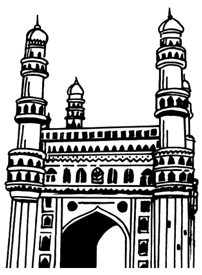
Rahul's ✔
Topper's Voice

AS PER CBCS SYLLABUS



B.Sc.

II Year III Sem

Latest 2022 Edition

DATA STRUCTURES USING C++

- Study Manual
- FAQ's and Important Questions
- Short Question and Answers
- Choose the Correct Answer
- Fill in the blanks
- Practical Programs
- Solved Previous Question Papers
- Solved Model Papers

` .199/-

- by -
WELL EXPERIENCED LECTURER

Rahul Publications™
Hyderabad. Ph : 66550071, 9391018098

B.Sc.

II Year III Sem

(Osmania University)

DATA STRUCTURES USING C++

Price ` 199-00

Sole Distributors : ☎ : **66550071, Cell : 9391018098**

VASU BOOK CENTRE

Shop No. 2, Beside Gokul Chat, Koti, Hyderabad.

Maternity Hospital Opp. Lane, Narayan Naik Complex, Koti, Hyderabad.
Near Andhra Bank, Subway, Sultan Bazar, Koti, Hyderabad -195.

C
O
N
T
E
N
T
S

DATA STRUCTURES USING C++

STUDY MANUAL

FAQ's and Important Questions	**V - VIII**
Unit - I	**1 - 88**
Unit - II	**89 - 180**
Unit - III	**181 - 274**
Unit - IV	**275 - 336**
Practical Programs	**337 - 430**

SOLVED MODEL PAPERS

Model Paper - I	**431 - 431**
Model Paper - II	**432 - 432**
Model Paper - III	**433 - 433**

PREVIOUS QUESTION PAPERS

July - 2021	**434 - 434**
November / December - 2019	**435 - 439**
June / July - 2019	**440 - 441**
November / December - 2018	**442 - 446**
May / June - 2018	**447 - 452**
November / December - 2017	**453 - 458**

SYLLABUS

UNIT - I

Basic data Structure: Introduction to Data Structures, Types of Data Structures, and Introduction to Algorithms, Pseudo code, and Relationship among data, data structures,andalgorithms, Implementation of data structures, Analysis of Algorithms.

Stacks: Concept of Stacks and Queues, Stacks, Stack Abstract Data Type, Representation of Stacks Using Sequential Organization (Arrays), Multiple Stacks, Applications of Stack, Expression Evaluation and Conversion, Polish notation and expression conversion, Processing of Function Calls, Reversing a String with a Stack, Recursion.

UNIT - II

Recursion: Introduction, Recurrence, Use of Stack in Recursion, Variants of Recursion, Recursive Functions, Iteration versus Recursion.

Queues: Concept of Queues, Queue as Abstract Data Type, Realization of Queues Using Arrays, Circular Queue, Multi-queues, Dequeue, Priority Queue, Applications of Queues,

Linked Lists: Introduction, Linked List, Linked List Abstract Data Type, Linked List Variants, Doubly Linked List, Circular Linked List, Representation of Sparse Matrix Using Linked List, Linked Stack, Linked Queue.

UNIT - III

Trees: Introduction, Types of Trees, Binary Tree, Binary Tree Abstract Data Type, Realization of a Binary Tree, Insertion of a Node in Binary Tree, Binary Tree Traversal, Other Tree Operations, Binary Search Tree, Threaded Binary Tree, Applications of Binary Trees.

Searching and Sorting: Search Techniques-Linear Search, Binary Search, Sorting TechniquesSelection Sort, Bubble Sort, Insertion Sort, Merge Sort, Quick Sort, Comparison of All Sorting Methods, Search Trees: Symbol Table, Optimal Binary Search Tree, AVL Tree (Heightbalanced Tree).

UNIT - IV

Graphs: Introduction, Representation of Graphs, Graph Traversal – Depth First Search, Breadth First Search, Spanning Tree, Prim’s Algorithm, Kruskal’s Algorithm.

Hashing: Introduction, Key Terms and Issues, Hash Functions, Collision Resolution Strategies, Hash Table Overflow, Extendible Hashing

Heaps: Basic Concepts, Implementation of Heap, Heap as Abstract Data Type, Heap Sort, Heap Applications.

Contents

UNIT - I

Topic **Page No.**

1.1 Basic Data Structures 1

1.1.1 Introduction to data structures 1

1.1.2 Types of data structures and Introduction to algorithm 3

1.1.3 Pseudo code and Relationship among data, data structures and algorithms 8

1.1.4 Implementation of data structures 16

1.1.5 Analysis of Algorithms 17

1.2 Stacks 26

1.2.1 Concepts of Stacks and Queues, Stacks, Stack Abstract Data type 26

1.2.2 Representation of Stacks Using sequential organizations(Arrays) 30

1.2.3 Multiple Stack 35

1.2.4 Applications of Stack, Expression evaluation and conversion, polish notation and Expression convertion 38

1.2.5 Processing of Function calls 65

1.2.6 Reversing a string with stack 67

1.2.7 Recursion 69

➢ Short Question & Answers .... 71 - 84

➢ Choose the Correct Answers .... 85 - 86

➢ Fill in the blanks .... 87 - 87

➢ One Mark Answers .... 88 - 88

UNIT - II

2.1 Recursion 89

2.1.1 Introduction 89

2.1.2 Recurrence 94

2.1.3 Use of Stack in Recursion 95

2.1.4 Variants of Recursion , Recursive functions 97

2.1.5 Iteration Versus Recrusion 101

2.2 Queues 102

2.2.1 Concept of Queues, Queue Abstract Data Type 102

2.2.2 Realization of Queues using Arrays 106

Topic **Page No.**

2.2.3 Circular Queue, Multi Queues 112
2.2.4 Dequeue, Priority Queue 116
2.2.5 Applications of Queues 124
2.3 Linked List 125
2.3.1 Introduction to Linked List 125
2.3.2 Linked List Abstract Data Type 134
2.3.3 Liked List variants, Doubly Linked List, Circular Linked List. 137
2.3.4 Representation of Sparse Matrix using Linked List 156
2.3.5 Linked Stack , Linked Queue 158
➢ Short Question & Answers .... 165 - 177
➢ Choose the Correct Answers .... 178 - 178
➢ Fill in the blanks .... 179 - 179
➢ One Mark Answers .... 189 - 180

UNIT - III

3.1 Trees 181
3.1.1 Introduction 181
3.1.2 Types of Trees 186
3.1.3 Binary Tree 188
3.1.4 Binary Tree Abstract Data Type 190
3.1.5 Realization of a Binary Tree 191
3.1.6 Insertion of a Node in Binary Tree 196
3.1.7 Binary Tree Traversal 198
3.1.8 Other Tree Operations 201
3.1.9 Binary Search Tree 204
3.1.10 Threaded Binary Tree 215
3.1.11 Applications of Binary Trees. 220
3.2 Searching and Sorting 221
3.2.1 Search Techniques 221
3.2.1.1 Linear Search 222
3.2.1.2 Binary Search 224
3.2.2 Sorting Techniques 227
3.2.2.1 Selection Sort 227

Topic **Page No.**

3.2.2.2 Bubble Sort 230

3.2.2.3 Insertion Sort 232

3.2.2.4 Merge Sort 235

3.2.2.5 Quick Sort 236

3.2.3 Comparison of All Sorting Methods 240

3.3 Search Trees 242

3.3.1 Symbol Table 242

3.3.2 Optimal Binary Search Tree 243

3.3.3 AVL Tree (Heightbalanced Tree) 249

➢ Short Question & Answers .... 264 - 271

➢ Choose the Correct Answers .... 272 - 272

➢ Fill in the blanks .... 273 - 273

➢ One Mark Answers .... 274 - 274

UNIT - IV

4.1 Graphs 275

4.1.1 Introduction 275

4.1.2 Representation of Graphs 276

4.2 Graph Traversal 278

4.2.1 Depth First Search 278

4.2.2 Breadth First Search 283

4.3 Spanning Tree 288

4.3.1 Prim’s Algorithm 292

4.3.2 Kruskal’s Algorithm. 296

4.4 Hashing 300

4.4.1 Introduction 300

4.4.2 Key Terms and Issues 301

4.4.3 Hash Functions 301

4.4.4 Collision Resolution Strategies 304

4.4.5 Hash Table Overflow 310

4.4.6 Extendible Hashing 310

Topic **Page No.**

4.5 Heaps 312

4.5.1 Basic Concepts 312

4.5.2 Implementation of Heap 317

4.5.2 Heap as Abstract Data Type 318

4.5.3 Heap Sort 320

4.5.4 Heap Applications 327

➢ Short Question & Answers .... 328 - 333

➢ Choose the Correct Answers .... 334 - 334

➢ Fill in the blanks .... 335 - 335

➢ One Mark Answers .... 336 - 336

Frequently Asked & Important Questions

UNIT - I

1. Define an algorithm. Write a flow chart and a pseudo code to compute the sum of the first N natural numbers.

Ans : **(Dec.-17, Imp.)**

Refer Unit-I, Q.No. 15

2. What is a Stack? List out the standard operations of Stack ?

Ans : **(June-18, Dec.-17)**

Refer Unit-I, Q.No. 28

3. Discuss in detail Stack ADT.

Ans : **(July-21, Dec.-17)**

Refer Unit-I, Q.No. 30

4. Explain the representation of stack using arrays implementation.

Ans : **(June-18, Imp.)**

Refer Unit-I, Q.No. 31

5. What is the use of function calls in stack implementation?

Ans : **(July-19, Imp.)**

Refer Unit-I, Q.No. 51

6. Write an algorithm to reverse a string using stack implementation.

Ans : **(July-21, July-19)**

Refer Unit-I, Q.No. 52

7. Write a C++ program to reverse a string using stack.

Ans : **(July-21, July-19)**

Refer Unit-I, Q.No. 53

8. Reversing the string "ABCDEF" using stack.

Ans : **(July-21, Dec.-17)**

Refer Unit-I, Q.No. 54

UNIT - II

1. Explain Towers of Hanoi using recursion.

Ans : **(July-19, Imp.)**

Refer Unit-II, Q.No. 16

2. What is mean by Queue? What are the basic operations of Queue?

Ans : **(June-18, Imp.)**

Refer Unit-II, Q.No. 19

3. Write a C++ program to implement Queue ADT.

Ans : **(June-18, Dec.-17)**

Refer Unit-II, Q.No. 22

4. What is mean by Circular Queue? What is the need of circular queue?

Ans : **(July-21, Dec.-19)**

Refer Unit-II, Q.No. 30

5. How can we search a specific element in the linked list?

Ans : **(Dec.-17, Imp.)**

Refer Unit-II, Q.No. 48

6. How can we search a specific element in the linked list?

Ans : **(Dec.-17, Imp.)**

Refer Unit-II, Q.No. 49

7. Discuss in detail about Doubly Linked List. Explain the basic operations of Doubly linked list.

Ans : **(Dec.-19, Imp.)**

Refer Unit-II, Q.No. 55

UNIT - III

1. Explain about various traversal techniques of binary tree.

Ans : **(July-21, Dec.-19, June-19, Dec.-18, June-18)**

Refer Unit-III, Q.No. 10

2. Write a C++ program for creation and traversal of a Binary Tree

Ans : **(Dec.-19, July-19, June-18)**

Refer Unit-III, Q.No. 11

3. What is Binary Search Tree? How to represent it.

Ans : **(July-21, Nov.-17)**

Refer Unit-III, Q.No. 13

4. Explain the operations that can be performed on BSTs.

Ans : **(Nov.-17)**

Refer Unit-III, Q.No. 14

5. Explain Binary Search with an example.

Ans : **(Junly-2019)**

Refer Unit-III, Q.No. 23

6. Explain Selection Sort with an example.

Ans : **(June-2018, Imp.)**

Refer Unit-III, Q.No. 26

7. Write a program to implement selection sort

Ans : **(June-18, Imp.)**

Refer Unit-III, Q.No. 27

8. Explain bubble sort with an example.

Ans : **(Dec.-19, June - 18)**

Refer Unit-III, Q.No. 28

9. What is insertion sort? Explain its working with an example.

Ans : **(Dec.-19, Dec.-18, Nov.-18)**

Refer Unit-III, Q.No. 29

10. What is merge sort? Explain it with an example.

Ans : **(Aug.-21, Dec.-19)**

Refer Unit-III, Q.No. 31

11. Compare various sorting techniques with real world usage.

Ans : **(July-21, Nov.-19)**

Refer Unit-III, Q.No. 34

UNIT - IV

1. What are the various ways to represent graphs?

Ans : **(Dec.-19, Dec.-18, Nov.-17)**

Refer Unit-IV, Q.No. 2

2. Explain DFS algorithm with an example.

Ans : **(June-18)**

Refer Unit-IV, Q.No. 3

3. Explain about BFS algorithm with an example.

Ans : **(June-18, Imp.)**

Refer Unit-IV, Q.No. 5

4. Explain the representation of Minimum Spanning tree.

Ans : **(June-18, Imp.)**

Refer Unit-IV, Q.No. 8

5. Write a program to implement Prim's Algorithm.

Ans : **(Nov.-17, Imp.)**

Refer Unit-IV, Q.No. 11

6. Explain Kruskal's Algorithm with an example.

Ans : **(Imp.)**

Refer Unit-IV, Q.No. 12

7. Write about heap data structure.

Ans : **(July-21, Dec.-19, Nov.-17)**

Refer Unit-IV, Q.No. 27

8. Write the applications of heap sort.

Ans : **(Imp.)**

Refer Unit-IV, Q.No. 30

UNIT I

Basic data Structure: Introduction to Data Structures, Types of Data Structures, and Introduction to Algorithms, Pseudo code, and Relationship among data, data structures,andalgorithms, Implementation of data structures, Analysis of Algorithms.
Stacks: Concept of Stacks and Queues, Stacks, Stack Abstract Data Type, Representation of Stacks Using Sequential Organization (Arrays), Multiple Stacks, Applications of Stack, Expression Evaluation and Conversion, Polish notation and expression conversion, Processing of Function Calls, Reversing a String with a Stack, Recursion.

1.1 Basic Data Structures

1.1.1 Introduction to data structures

Q1. What is a Data Structure? What are the basic operations of Data Structures.

Ans :

Data Structure is a way to store and organize data so that it can be used efficiently.The data structure name indicates itself that organizing the data in memory. There are many ways of organizing the data in the memory as we have already seen one of the data structures, i.e., array in C language. Array is a collection of memory elements in which data is stored sequentially, i.e., one after another. In other words, we can say that array stores the elements in a continuous manner. This organization of data is done with the help of an array of data structures. There are also other ways to organize the data in memory.

The major or the common operations that can be performed on the data structures are:

- **Searching:** We can search for any element in a data structure.
- **Sorting:** We can sort the elements of a data structure either in an ascending or descending order.
- **Insertion:** We can also insert the new element in a data structure.
- **Updation:** We can also update the element, i.e., we can replace the element with another element.
- **Deletion:** We can also perform the delete operation to remove the element from the data structure.

Q2. Discuss the basic terminology of Data Structures.

Ans :

Data structures are the building blocks of any program or the software. Choosing the appropriate data structure for a program is the most difficult task for a programmer. Following terminology is used as far as data structures are concerned.

Data

Data can be defined as an elementary value or the collection of values, for example, student's name and its id are the data about the student.

Group Items

Data items which have subordinate data items are called Group item, for example, name of a student can have first name and the last name.

Record

Record can be defined as the collection of various data items, for example, if we talk about the student entity, then its name, address, course and marks can be grouped together to form the record for the student.

File

A File is a collection of various records of one type of entity, for example, if there are 60 employees in the class, then there will be 20 records in the related file where each record contains the data about each employee.

Attribute and Entity

An entity represents the class of certain objects. it contains various attributes. Each attribute represents the particular property of that entity.

Field

Field is a single elementary unit of information representing the attribute of an entity.

Q3. What is the need of data structures?

Ans :

As applications are getting complexed and amount of data is increasing day by day, there may arrise the following problems:

Processor speed

To handle very large amout of data, high speed processing is required, but as the data is growing day by day to the billions of files per entity, processor may fail to deal with that much amount of data.

Data Search

Consider an inventory size of 106 items in a store, If our application needs to search for a particular item, it needs to traverse 106 items every time, results in slowing down the search process.

Multiple requests

If thousands of users are searching the data simultaneously on a web server, then there are the chances that a very large server can be failed during that process.

In order to solve the above problems, data structures are used. Data is organized to form a data structure in such a way that all items are not required to be searched and required data can be searched instantly.

Q4. List and explain advantages of data structures ?

Ans :

Efficiency

Efficiency of a program depends upon the choice of data structures. For example: suppose, we have some data and we need to perform the search for a perticular record. In that case, if we organize our data in an array, we will have to search sequentially element by element. hence, using array may not be very efficient here. There are better data structures which can make the search process efficient like ordered array, binary search tree or hash tables.

Reusability

Data structures are reusable, i.e. once we have implemented a particular data structure, we can use it at any other place. Implementation of data structures can be compiled into libraries which can be used by different clients.

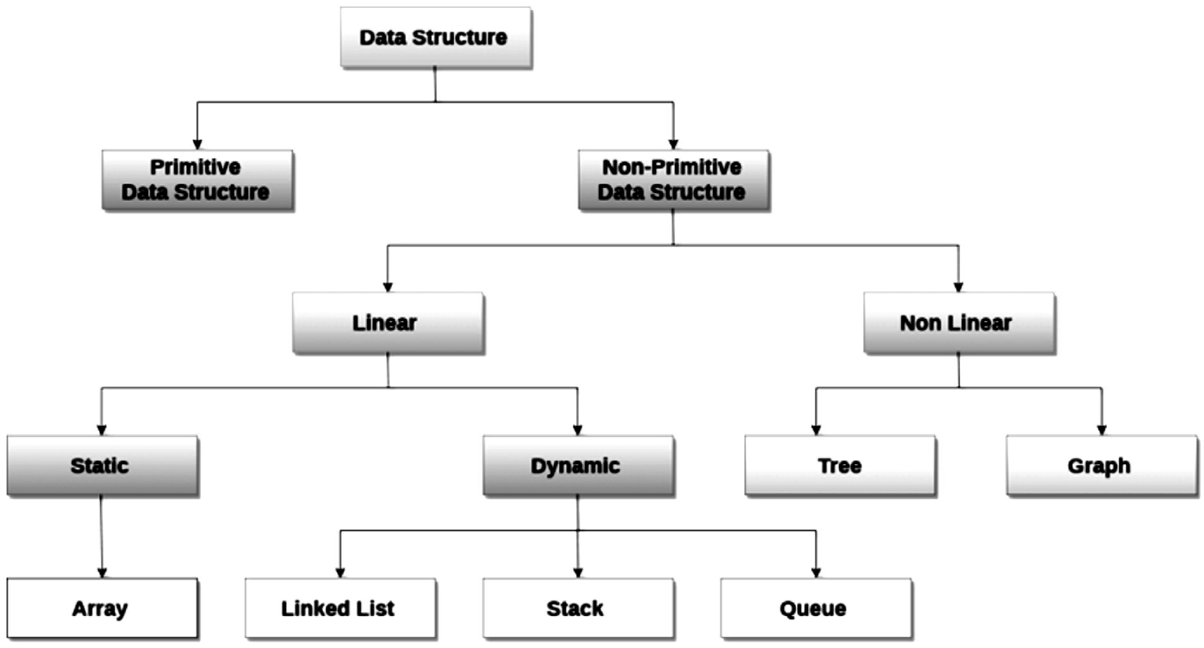
Abstraction

Data structure is specified by the ADT which provides a level of abstraction. The client program uses the data structure through interface only, without getting into the implementation details.

1.1.2 Types of data structures and Introduction to algorithm

Q5. Discuss various types of data structures.

Ans :



Linear Data Structures

A data structure is called linear if all of its elements are arranged in the linear order. In linear data structures, the elements are stored in non-hierarchical way where each element has the successors and predecessors except the first and last element.

Types of Linear Data Structures are given below:

Arrays

An array is a collection of similar type of data items and each data item is called an element of the array. The data type of the element may be any valid data type like char, int, float or double.

The elements of array share the same variable name but each one carries a different index number known as subscript. The array can be one dimensional, two dimensional or multidimensional.

The individual elements of the array age are:

age[0], age[1], age[2], age[3],......... age[98], age[99].

Linked List

Linked list is a linear data structure which is used to maintain a list in the memory. It can be seen as the collection of nodes stored at non-contiguous memory locations. Each node of the list contains a pointer to its adjacent node.

Stack

Stack is a linear list in which insertion and deletions are allowed only at one end, called top.

A stack is an abstract data type (ADT), can be implemented in most of the programming languages. It is named as stack because it behaves like a real-world stack, for example: - piles of plates or deck of cards etc.

Queue

Queue is a linear list in which elements can be inserted only at one end called rear and deleted only at the other end called front.

It is an abstract data structure, similar to stack. Queue is opened at both end therefore it follows First-In-First-Out (FIFO) methodology for storing the data items.

Non Linear Data Structures

This data structure does not form a sequence i.e. each item or element is connected with two or more other items in a non-linear arrangement. The data elements are not arranged in sequential structure.

Types of Non Linear Data Structures are given below:

Trees

Trees are multilevel data structures with a hierarchical relationship among its elements known as nodes. The bottommost nodes in the herierchy are called leaf node while the topmost node is called root node. Each node contains pointers to point adjacent nodes.

Tree data structure is based on the parent-child relationship among the nodes. Each node in the tree can have more than one children except the leaf nodes whereas each node can have atmost one parent except the root node. Trees can be classfied into many categories which will be discussed later in this tutorial.

Graphs

Graphs can be defined as the pictorial representation of the set of elements (represented by vertices) connected by the links known as edges. A graph is different from tree in the sense that a graph can have cycle while the tree can not have the one.

Q6. What is Algorithm? What are the characteristics of an algorithm ?

Ans :

An algorithm is a process or a set of rules required to perform calculations or some other problem-solving operations especially by a computer. The formal definition of an algorithm is that it contains the finite set of instructions which are being carried in a specific order to perform the specific task. It is not the complete program or code; it is just a solution (logic) of a problem, which can be represented either as an informal description using a Flowchart or Pseudo code.

The following are the characteristics of an algorithm:

➢ **Input**

An algorithm has some input values. We can pass 0 or some input value to an algorithm.

➢ **Output**

We will get 1 or more output at the end of an algorithm.

➢ **Unambiguity**

An algorithm should be unambiguous which means that the instructions in an algorithm should be clear and simple.

➢ **Finiteness**

An algorithm should have finiteness. Here, finiteness means that the algorithm should contain a limited number of instructions, i.e., the instructions should be countable.

➢ **Effectiveness**

An algorithm should be effective as each instruction in an algorithm affects the overall process.

➢ **Language independent**

An algorithm must be language-independent so that the instructions in an algorithm can be implemented in any of the languages with the same output.

Q7. Why do we need algorithms ? List out the characteristics of an algorithm.

Ans :

We need algorithms because of the following reasons:

- **Scalability**

  It helps us to understand the scalability. When we have a big real-world problem, we need to scale it down into small-small steps to easily analyze the problem.

- **Performance**

  The real-world is not easily broken down into smaller steps. If the problem can be easily broken into smaller steps means that the problem is feasible.

Let's understand the algorithm through a real-world example. Suppose we want to make a lemon juice, so following are the steps required to make a lemon juice:

Step 1: First, we will cut the lemon into half.

Step 2: Squeeze the lemon as much you can and take out its juice in a container.

Step 3: Add two tablespoon sugar in it.

Step 4: Stir the container until the sugar gets dissolved.

Step 5: When sugar gets dissolved, add some water and ice in it.

Step 6: Store the juice in a fridge for 5 to minutes.

Step 7: Now, it's ready to drink.

The above real-world can be directly compared to the definition of the algorithm. We cannot perform the step 3 before the step 2, we need to follow the specific order to make lemon juice. An algorithm also says that each and every instruction should be followed in a specific order to perform a specific task.

Now we will look an example of an algorithm in programming.

We will write an algorithm to add two numbers entered by the user.

The following are the steps required to add two numbers entered by the user:

Step 1: Start

Step 2: Declare three variables a, b, and sum.

Step 3: Enter the values of a and b.

Step 4: Add the values of a and b and store the result in the sum variable, i.e., sum = a + b.

Step 5: Print sum

Step 6: Stop

Characteristics of an Algorithm

- **Unambiguous**

  Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.

- **Input**

  An algorithm should have 0 or more well-defined inputs.

- **Output**

  An algorithm should have 1 or more well-defined outputs, and should match the desired output.

- **Finiteness**

  Algorithms must terminate after a finite number of steps.

- **Feasibility**

  Should be feasible with the available resources.

- **Independent**

  An algorithm should have step-by-step directions, which should be independent of any programming code.

Q8. What are the major factors considered to design an algorithm ?

Ans :

The following are the factors that we need to consider for designing an algorithm:

- **Modularity**

  If any problem is given and we can break that problem into small-small modules or small-small steps, which is a basic definition of an algorithm, it means that this feature has been perfectly designed for the algorithm.

- **Correctness**

  The correctness of an algorithm is defined as when the given inputs produce the desired output, which means that the algorithm has been designed algorithm. The analysis of an algorithm has been done correctly.

- **Maintainability**

  Here, maintainability means that the algorithm should be designed in a very simple structured way so that when we redefine the algorithm, no major change will be done in the algorithm.

- **Functionality**

  It considers various logical steps to solve the real-world problem.

- **Robustness**

  Robustness means that how an algorithm can clearly define our problem.

- **User-friendly**

  If the algorithm is not user-friendly, then the designer will not be able to explain it to the programmer.

- **Simplicity**

  If the algorithm is simple then it is easy to understand.

- **Extensibility**

  If any other algorithm designer or programmer wants to use your algorithm then it should be extensible.

Q9. Explain Importance, Issues and approaches of an algorithm?

Ans :

Importance of Algorithms

1. Theoretical importance

When any real-world problem is given to us and we break the problem into small-small modules. To break down the problem, we should know all the theoretical aspects.

2. Practical importance

As we know that theory cannot be completed without the practical implementation. So, the importance of algorithm can be considered as both theoretical and practical.

Issues of Algorithms

The following are the issues that come while designing an algorithm:

- **How to design algorithms**

  As we know that an algorithm is a step-by-step procedure so we must follow some steps to design an algorithm.

- How to analyze algorithm efficiency

Approaches of Algorithm

The following are the approaches used after considering both the theoretical and practical importance of designing an algorithm:

- **Brute force algorithm**

  The general logic structure is applied to design an algorithm. It is also known as an exhaustive search algorithm that searches all the possibilities to provide the required solution. Such algorithms are of two types:

  1. **Optimizing:** Finding all the solutions of a problem and then take out the best solution or if the value of the best solution is known then it will terminate if the best solution is known.
  2. **Sacrificing:** As soon as the best solution is found, then it will stop.

- **Divide and conquer**

  It is a very implementation of an algorithm. It allows you to design an algorithm in a step-by-step variation. It breaks down the algorithm to solve the problem in different methods. It allows you to break down the problem into different methods, and valid output is produced for the valid input. This valid output is passed to some other function.

- **Greedy algorithm**

  It is an algorithm paradigm that makes an optimal choice on each iteration with the hope of getting the best solution. It is easy to implement and has a faster execution time. But, there are very rare cases in which it provides the optimal solution.

- **Dynamic programming**

  It makes the algorithm more efficient by storing the intermediate results. It follows five different steps to find the optimal solution for the problem:

  1. It breaks down the problem into a subproblem to find the optimal solution.
  2. After breaking down the problem, it finds the optimal solution out of these subproblems.
  3. Stores the result of the subproblems is known as memorization.
  4. Reuse the result so that it cannot be recomputed for the same subproblems.
  5. Finally, it computes the result of the complex program.

- **Branch and Bound Algorithm**

  The branch and bound algorithm can be applied to only integer programming problems. This approach divides all the sets of feasible solutions into smaller subsets. These subsets are further evaluated to find the best solution.

- **Randomized Algorithm**

  As we have seen in a regular algorithm, we have predefined input and required output. Those algorithms that have some defined set of inputs and required output, and follow some described steps are known as deterministic algorithms. What happens that when the random variable is introduced in the randomized algorithm?. In a randomized algorithm, some random bits are introduced by the algorithm and added in the input to produce the output, which is random in nature. Randomized algorithms are simpler and efficient than the deterministic algorithm.

- **Backtracking**

  Backtracking is an algorithmic technique that solves the problem recursively and removes the solution if it does not satisfy the constraints of a problem.

Q10. List out major categories of algorithms.

Ans :

The major categories of algorithms are given below:

- **Sort:** Algorithm developed for sorting the items in a certain order.
- **Search:** Algorithm developed for searching the items inside a data structure.
- **Delete:** Algorithm developed for deleting the existing element from the data structure.
- **Insert:** Algorithm developed for inserting an item inside a data structure.
- **Update:** Algorithm developed for updating the existing element inside a data structure.

Q11. Explain how algorithms are analysed?

Ans :

The algorithm can be analyzed in two levels, i.e., first is before creating the algorithm, and second is after creating the algorithm. The following are the two analysis of an algorithm:

- **Priori Analysis**

  Here, priori analysis is the theoretical analysis of an algorithm which is done before implementing the algorithm. Various factors can be considered before implementing the algorithm like processor speed, which has no effect on the implementation part.

- **Posterior Analysis**

  Here, posterior analysis is a practical analysis of an algorithm. The practical analysis is achieved by implementing the algorithm using any programming language. This analysis basically evaluate that how much running time and space taken by the algorithm.

  The performance of the algorithm can be measured in two factors:

- **Time complexity**

  The time complexity of an algorithm is the amount of time required to complete the execution. The time complexity of an algorithm is denoted by the big O notation. Here, big O notation is the asymptotic notation to represent the time complexity.

The time complexity is mainly calculated by counting the number of steps to finish the execution. Let's understand the time complexity through an example.

```
sum=0;
// Suppose we have to calculate
the sum of n numbers.
for i=1 to n
sum=sum+i;
// when the loop ends then sum
holds the sum of the n numbers
return sum;
```

In the above code, the time complexity of the loop statement will be atleast n, and if the value of n increases, then the time complexity also increases. While the complexity of the code, i.e., return sum will be constant as its value is not dependent on the value of n and will provide the result in one step only. We generally consider the worst-time complexity as it is the maximum time taken for any given input size.

➢ **Space complexity**

An algorithm's space complexity is the amount of space required to solve a problem and produce an output. Similar to the time complexity, space complexity is also expressed in big O notation.

For an algorithm, the space is required for the following purposes:

1. To store program instructions
2. To store constant values
3. To store variable values
4. To track the function calls, jumping statements, etc.

Auxiliary space

The extra space required by the algorithm, excluding the input size, is known as an auxiliary space. The space complexity considers both the spaces, i.e., auxiliary space, and space used by the input.

So,

Space complexity = Auxiliary space + Input size.

1.1.3 Pseudo code and Relationship among data, data structures and algorithms

Q12. What is pseudo code? Explain with an example. List out its advantages and disadvantages?

Ans :

Pseudocode is not a programming language, it is a simple way of describing a set of instructions that does not have to use specific syntax.

Writing in pseudocode is similar to writing in a programming language. Each step of the algorithm is written on a line of its own in sequence. Usually, instructions are written in uppercase, variables in lowercase and messages in sentence case.

In pseudocode, INPUT asks a question. OUT PUT prints a message on screen.

Pseudocode is an informal way of programming description that does not require any strict programming language syntax or underlying technology considerations. It is used for creating an outline or a rough draft of a program. Pseudocode summarizes a program's flow, but excludes underlying details. System designers write pseudocode to ensure that programmers understand a software project's requirements and align code accordingly.

It is written in the form of annotations and informational text that is written in plain English only. Just like programming languages, it doesn't have any syntax, so it cannot be compiled or interpreted by the compiler.

Below are some points which we need to keep in mind while designing the pseudocode

➢ We should have to use the appropriate naming convention. By doing that, it is very easy to understand the pseudocode. So, the naming should be simple and distinct.

➢ We should have to use the appropriate sentence casings. For methods, we use the CamelCase, for constants, we use the upper case, and for variables, we use the lower case.

➢ The pseudocode should not be abstract, and the thing which is going to happen in the actual code should be elaborated.

- We use the 'if-then, 'for', 'while', 'cases' standard programming structures in the same way as we use it in programming.
- All the sections of the pseudocode should be completed, finite and clear to understand.
- The pseudocode should be as simple as it can be understood by a layman having no sufficient knowledge of technical terms. So, we don't write the pseudocode in a complete programmatic manner.

Pseudocode is also written using some specific words and characters, which are shown below:

- To begin the comment double forward slash are used "//".
- Matching braces "{and}" are used to present blocks where a compound statement (set of simple statements) can be illustrated as a block and terminated by a semicolon";". The body of a procedure constructs a block as well.
- All the identifiers start with a letter and the datatype of the variables are not declared explicitly.
- An assignment statement is used for the assigning values to the variables.
- To produce the boolean values (i.e., true and false) the logical operators and, or and not and the relational operators <, d", =, =, e" and > are provided.

  Input and output are presented by read and write instructions.
- "if and then" expressions are used to express a conditional statement.

Example:

The pseudo code of the Armstrong number can be written in the following way:

```
1.  Initialize c to zero.
2.  Initialize n to a random number to check Armstrong.
3.  Initialize temp to n.
4.  Repeat steps until the value of n are greater than zero.
5.  Find a reminder of n by using n%10.
6.  Remove the last digit from the number by using n/10.
7.  Find the thrice of the reminder and add it to c.
8.  If temp = = c
    Print "Armstrong number"
9.  else
    Not an Armstrong number"
```

The algorithm of the above program can be written in the following way:

Input the number.

```
1.  Initialize c = 0 and temp = n.
2.  Repeat until (temp != 0)
3.  a = temp % 10 //remainder
4.  c = c + (a * a * a)
5.  temp = temp / 10
6.  if (c = = n)
7.  Display "Armstrong number"
8.  Else
9.  Display "Not an Armstrong number"
```

Advantages

- In order to improve the readability of any approach, pseudo code plays a very important role.
- In between the program and the algorithm, Pseudo code work as a bridge. It is treated as a document so that the developer can understand the program easily.
- Pseudo code focuses on explaining the working on each line of the program. Due to this, it is very easy for the programmer to construct the code.

Disadvantages

- The visual representation of the programming code can be easily understood, and the pseudocode doesn't provide it.
- There is no well-defined format to write the pseudocode.

- There are no standards available for pseudocode. Companies use their own standards to write it.
- If we use pseudocode, we need to maintain one more document for our code.

Q13. Write a Pseudo code and algorithm for calculating area of circle.

Ans :

Code:

```
AreaofCircle()
{
BEGIN
Read: Number radius, Area;
Input r;
Area = 3.14 * r * r;
Output Area;
END
}
```

Algorithm:

```
1.  Start.
2.  Read the radius value r as the input given by the user.
3.  Calculate the area as Area: 3.14 * r * r.
4.  Display the Area.
5.  End.
```

Q14. Write an algorithm to determine if a student is pass or fail based on the grades. Grades are the average of total marks obtained in all the subjects.

Ans :

Algorithm

Steps are given below:

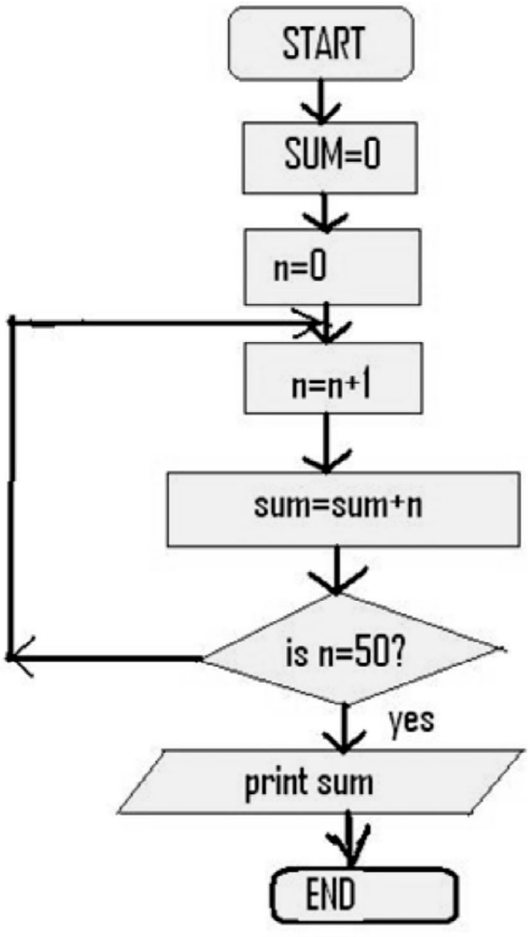
```
1.  Start
2.  Input Marks1, Marks2, Marks3, Marks4
3.  Grade= (Marks1+Marks2 +Marks3 + Marks4)/4
4.  If (Grade<50) then
5.  Print "Fail"
6.  Else
7.  Print "Pass"
8.  End if
9.  Stop
```

Q15. Define an algorithm. Write a flow chart and a pseudo code to compute the sum of the first N natural numbers.

Ans : **(Dec.-17, Imp.)**

The word Algorithm means "a process or set of rules to be followed in calculations or other problem-solving operations". Therefore Algorithm refers to a set of rules/instructions that step-by-step define how a work is to be executed upon in order to get the expected results.

Flow chart for some of the first N natural numbers



Pseudo code for some of N natural numbers

```
BEGIN
NUMBER counter, sum=0
FOR counter=1 TO 100 STEP 1 DO
sum=sum+counter
ENDFOR
OUTPUT sum
END
```

Q16. Write an algorithm to compute the following using a subalgorithm P = n! / (n-r)!

Ans :

```
Start
Step 1 -> Declare function for calculating factorial
    int cal_n(int n)
    int temp = 1
    Loop for int i = 2 and i <= n and i++
        Set temp = temp * i
    End
    return temp
step 2 -> declare function to calculate ncr
    int nCr(int n, int r)
        return cal_n(n) / (cal_n(r) * cal_n(n - r))
step 3 -> In main()
    declare variable as int n = 12, r = 4
    print nCr(n, r)
Stop
```

Q17. What are the differences between Pseudocode and Algorithms.

Ans :

S. NO	Pseudocode	Algorithm
1.	It is the method used for writing the computer program in easy ad readable format or in the English language to make it the user or developer easier to understand the logic for further problem resolution.	It is another method of describing the computer program in a readable ad under standable with the sequence of steps written to follow while writing the code.
2.	It is said to be the representation of the algorithm as it is simpler to read and understand than the algorithm.	It is said to be the sequence of steps of explanation for the code with logical steps that makes it easier but a bit difficult to understand than pseudocode.
3.	It is one of the simpler methods or versions for writing code in any programming language.	It is one of the systematic and logical methods or versions for writing code in any progra-mming language.
4.	Pseudocode is not a programming language and is written in simple	Algorithms use natural languages ad flow-charts for representing any code and hence

	English and hence it is easy to understand and read.	it is easy to read but difficult in understanding the algorithm.
5.	Pseudocode can be considered a method for describing the program into high-level description with the operating principle of any computer program or algorithm.	The algorithm is considered to be an unambiguous method of describing the code for which it specifies how to solve the problem.
6.	Pseudocode uses simple English language or phrases as there is no particular syntax followed to write ay pseudocode as it is the method.	The algorithm uses high-level constructs such as code snippets which may sometimes make it difficult for interpreting and under standing.
7.	Pseudocode is quite easier to construct because it is written in easy under standing simpler for debugging.	The algorithm is quite complex when constructing as it sometimes involves code snippets in it and hence it is a bit difficult when it comes to debugging.
8.	Pseudocodes are not used in any complicated programming languages as algorithms because it is widely used when the combination of programming and simple natural languages is required for constructing the pseudocode.	Algorithms can be used in any complex programming language as it uses simple logical code snippets sometimes which is more than natural language as used in Pseudocode constructions.
9.	The pseudocode also uses few specific words or phrases or special symbols to differentiate it from the algorithm, though pseudocodes can be considered as a method of writing algorithms. It includes a forward slash for beginning comments, uses flower braces for in-cluding logical blocks, italso uses some operators to specify the logic of the code.	The algorithm is written using some specific criteria such as it includes input statement, output statement, also includes with effectiveness and efficiency of the algorithm code written for a particular program, clear statements which are free from ambiguity, also includes certain or proper conditions to be displayed for correctly terminating after few steps when required in the code, and sometimes small logical codes can also be written to specify some logic directly.
10.	Pseudocode cannot be considered algorithms.	Algorithms can be considered as pseudocode.

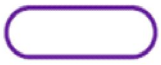
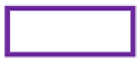
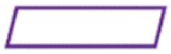
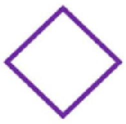
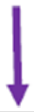
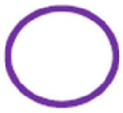
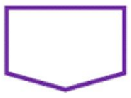
Q18. What is flow chart? Discuss various symbols used in flow chart representation.

Ans :

Flowchart is a diagrammatic representation of sequence of logical steps of a program. Flowcharts use simple geometric shapes to depict processes and arrows to show relationships and process/data flow.

Flowchart Symbols

Here is a chart for some of the common symbols used in drawing flowcharts.

Symbol	Symbol Name	Purpose
	Start/Stop	Used at the beginning and end of the algorithm to show start and end of the program.
	Process	Indicates processes like mathematical operations.
	Input/ Output	Used for denoting program inputs and outputs.
	Decision	Stands for decision statements in a program, where answer is usually Yes or No.
	Arrow	Shows relationships between different shapes.
	On-page Connector	Connects two or more parts of a flowchart, which are on the same page.
	Off-page Connector	Connects two parts of a flowchart which are spread over different pages.

Guidelines for Developing Flowcharts

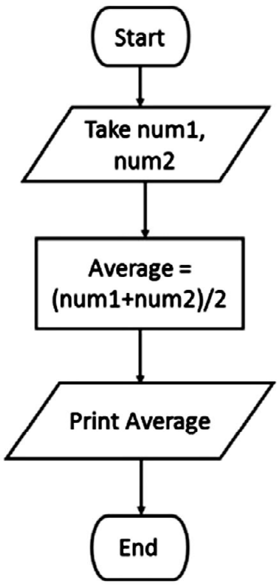
These are some points to keep in mind while developing a flowchart -

- Flowchart can have only one start and one stop symbol
- On-page connectors are referenced using numbers
- Off-page connectors are referenced using alphabets
- General flow of processes is top to bottom or left to right
- Arrows should not cross each other

Q19. Draw a flow chart to calculate the average of two numbers.

Ans :

Here is a flowchart to calculate the average of two numbers.



Q20. List out the advantages and disadvantages of flow charts

Ans :

Advantages

Communication

Flowcharts are better way of communicating the logic of a system to all concerned.

Effective analysis

With the help of flowchart, problem can be analysed in more effective way.

Proper documentation

Program flowcharts serve as a good program documentation, which is needed for various purposes.

Efficient Coding

The flowcharts act as a guide or blueprint during the systems analysis and program development phase.

Proper Debugging: The flowchart helps in debugging process.

Efficient Program Maintenance

The maintenance of operating program becomes easy with the help of flowchart. It helps the programmer to put efforts more efficiently on that part.

Disadvanatges

Although a flowchart is a very useful tool, there are a few limitations in using flowcharts which are listed below:

Complex logic

Sometimes, the program logic is quite complicated. In that case, flowchart becomes complex and clumsy.

Alterations and Modifications

If alterations are required the flowchart may require re-drawing completely.

Reproduction

As the flowchart symbols cannot be typed, reproduction of flowchart becomes a problem.

The essentials of what is done can easily be lost in the technical details of how it is done.

Q21. What are the differences between algorithm and flow chart.

Ans :

Flowchart	Algorithm
Block by block information diagram representing the data flow.	Step by step instruction representing the process of any solution.
It is a pictorial representation of a process.	It is a stepwise analysis of the work to be done.
The solution is shown in a graphical format.	The solution is shown in a non-computer language like English.
Easy to understand as compared to the algorithm.	It is somewhat difficult to understand.
Easy to show branching and looping.	Difficult to show branching and looping
Flowchart for a big problem is impractical	The algorithm can be written for any problem
Difficult to debug errors.	Easy to debug errors.
It is easy to make a flowchart.	It is difficult to write an algorithm as compared to a flowchart.

1.1.4 Implementation of data structures

Q22. Discuss possible ways to implement data structures.

Ans :

Data structures are implemented by using

1. Arrays
2. Linked list
3. Stack
4. Queue

1. Arrays

An array is a structure of fixed-size, w hich can hold items of the same data type. Arrays are indexed, meaning that random access is possible. An array is usually presented as a native data structure in many programming languages.

2. Linked Lists

A linked list is a sequential structure that consists of a sequence of items in linear order which are linked to each other. You are only required to know one end of the chain to traverse through this data structure. In a scenario where the size changes frequency, using an array might not be advantageous unless data is accessed randomly (expansions can cause higher copy times and use more memory unless you implement a shrink operation). So a linked list can be considered as a data structure that supports frequent size variations and sequential access.

3. Stacks

A stack is a LIFO (Last In First Out - the element placed at last can be accessed at first) structure. Although this data structure has a different behaviour, this can be considered as a derivation of the linked list with only a head or access to the top element.

4. Queues

A queue is a FIFO (First In First Out - the element placed at first can be accessed at first) structure. This can be considered as the reverse scenario of the stack. In simpler terms, it is a linked list where we add from one end read from the other end.

1.1.5 Analysis of Algorithms

Q23. Explain how algorithms are analysed.

Ans :

In theoretical analysis of algorithms, it is common to estimate their complexity in the asymptotic sense, i.e., to estimate the complexity function for arbitrarily large input. The term "analysis of algorithms" was coined by Donald Knuth.

Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem. Most algorithms are designed to work with inputs of arbitrary length. Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

Usually, the efficiency or running time of an algorithm is stated as a function relating the input length to the number of steps, known as time complexity, or volume of memory, known as space complexity.

By considering an algorithm for a specific problem, we can begin to develop pattern recognition so that similar types of problems can be solved by the help of this algorithm.

Algorithms are often quite different from one another, though the objective of these algorithms are the same. For example, we know that a set of numbers can be sorted using different algorithms. Number of comparisons performed by one algorithm may vary with others for the same input. Hence, time complexity of those algorithms may differ. At the same time, we need to calculate the memory space required by each algorithm.

Analysis of algorithm is the process of analyzing the problem-solving capability of the algorithm in terms of the time and size required (the size of memory for storage while implementation). However, the main concern of analysis of algorithms is the required time or performance. Generally, we perform the following types of analysis -

- **Worst-case**

  The maximum number of steps taken on any instance of size a.

- **Best-case**

  The minimum number of steps taken on any instance of size a.

- **Average case**

  An average number of steps taken on any instance of size a.

- **Amortized**

  A sequence of operations applied to the input of size a averaged over time.

Q24. What is time complexity ? Explain how to calculate time complexity with an example.

Ans :

Time complexity of an algorithm signifies the total time required by the program to run till its completion.

The time complexity of algorithms is most commonly expressed using the big O notation. It's an asymptotic notation to represent the time complexity.Time Complexity is most commonly estimated by counting the number of elementary steps performed by any algorithm to finish execution.

Calculating Time Complexity

the most common metric for calculating time complexity is Big O notation. This removes all constant factors so that the running time can be estimated in relation to N, as N approaches infinity. In general you can think of it like this :

```
statement;
```

Above we have a single statement. Its Time Complexity will be Constant. The running time of the statement will not change in relation to N.

```
for(i=0; i< N; i++)
{
statement;
}
```

The time complexity for the above algorithm will be Linear. The running time of the loop is directly proportional to N. When N doubles, so does the running time.

```
for(i=0; i< N; i++)
{
for(j=0; j <N;j++)
  {
statement;
  }
}
```

This time, the time complexity for the above code will be Quadratic. The running time of the two loops is proportional to the square of N. When N doubles, the running time increases by N * N.

```
while(low <= high)
{
mid=(low + high)/2;
if(target < list[mid])
high= mid -1;
elseif(target > list[mid])
low= mid +1;
elsebreak;
}
```

This is an algorithm to break a set of numbers into halves, to search a particular field(we will study this in detail later). Now, this algorithm will have a Logarithmic Time Complexity. The running time of the algorithm is proportional to the number of times N can be divided by 2(N is high-low here). This is because the algorithm divides the working area in half with each iteration.

```
voidquicksort(int list[],int left,int right)
{
int pivot =partition(list, left, right);
quicksort(list, left, pivot -1);
quicksort(list, pivot +1, right);
}
```

Types of Notations for Time Complexity

Now we will discuss and understand the various notations used for Time Complexity.

1. **Big Oh** denotes "fewer than or the same as" <expression> iterations.
2. **Big Omega** denotes "more than or the same as" <expression> iterations.
3. **Big Theta** denotes "the same as" <expression> iterations.
4. **Little Oh** denotes "fewer than" <expression> iterations.
5. **Little Omega** denotes "more than" <expression> iterations.

O(expression) is the set of functions that grow slower than or at the same rate as expression. It indicates the maximum required by an algorithm for all input values. It represents the worst case of an algorithm's time complexity.

Omega (expression) is the set of functions that grow faster than or at the same rate as expression. It indicates the minimum time required by an algorithm for all input values. It represents the best case of an algorithm's time complexity.

Theta(expression) consist of all the functions that lie in both O(expression) and Omega(expression). It indicates the average bound of an algorithm. It represents the average case of an algorithm's time complexity.

Q25. What is space complexity? Explain with an example.

Ans :

Space complexity is an amount of memory used by the algorithm (including the input values of the algorithm), to execute it completely and produce the result.

We know that to execute an algorithm it must be loaded in the main memory. The memory can be used in different forms:

- Variables (This includes the constant values and temporary values)
- Program Instruction
- Execution

Auxiliary Space

Auxiliary space is extra space or temporary space used by the algorithms during its execution.

Memory Usage during program execution

- Instruction Space is used to save compiled instruction in the memory.
- Environmental Stack is used to storing the addresses while a module calls another module or functions during execution.
- Data space is used to store data, variables, and constants which are stored by the program and it is updated during execution.

Space Complexity = Auxiliary Space + Input space

While executing, algorithm uses memory space for three reasons:

1. **Instruction Space**

   It's the amount of memory used to save the compiled version of instructions.

2. **Environmental Stack**

   Sometimes an algorithm(function) may be called inside another algorithm(function). In such a situation, the current variables are pushed onto the system stack, where they wait for further execution and then the call to the inside algorithm(function) is made.

   For example, If a function A() calls function B() inside it, then all th variables of the function A() will get stored on the system stack temporarily, while the function B() is called and executed inside the funciton A().

3. **Data Space**

   Amount of space used by the variables and constants.

   But while calculating the Space Complexity of any algorithm, we usually consider only Data Space and we neglect the Instruction Space and Environmental Stack.

   For calculating the space complexity, we need to know the value of memory used by different type of datatype variables, which generally varies for different operating systems, but the method for calculating the space complexity remains the same.

Type	Size
bool, char, unsigned char, signed char, __int8	1 byte
__int16, short, unsigned short, wchar_t, __wchar_t	2 bytes
float, __int32, int, unsigned int, long, unsigned long	4 bytes
double, __int64, long double, long long	8 bytes

Now let's learn how to compute space complexity by taking a few examples:

```
{
int z = a + b + c;
return(z);
```

```
        }
```

In the above expression, variables a, b, c and z are all integer types, hence they will take up 4 bytes each, so total memory requirement will be (4(4) + 4) = 20 bytes, this additional 4 bytes is for return value. And because this space requirement is fixed for the above example, hence it is called Constant Space Complexity.

Let's have another example, this time a bit complex one,

```
        intsum(int a[], int n)
{
        int x = 0; // 4 bytes for x
        for(inti = 0; i< n; i++)   // 4 bytes for i
        {
        x  = x + a[i];
        }
        return(x);
}
```

- In the above code, 4*n bytes of space is required for the array a[] elements.
- 4 bytes each for x, n, i and the return value.
- Hence the total memory requirement will be (4n + 12), which is increasing linearly with the increase in the input value n, hence it is called as Linear Space Complexity.
- Similarly, we can have quadratic and other complex space complexity as well, as the complexity of an algorithm increases.
- But we should always focus on writing algorithm code in such a way that we keep the space complexity minimum.

Q26. What is the use of asymptotic analysis?

Ans :

Using asymptotic analysis, we can get an idea about the performance of the algorithm based on the input size. We should not calculate the exact running time, but we should find the relation between the running time and the input size. We should follow the running time when the size of input is increased.

For the space complexity, our goal is to get the relation or function that how much space in the main memory is occupied to complete the algorithm.

Asymptotic Behavior

For a function f(n) the asymptotic behavior is the growth of f(n) as n gets large. Small input values are not considered. Our task is to find how much time it will take for large value of the input.

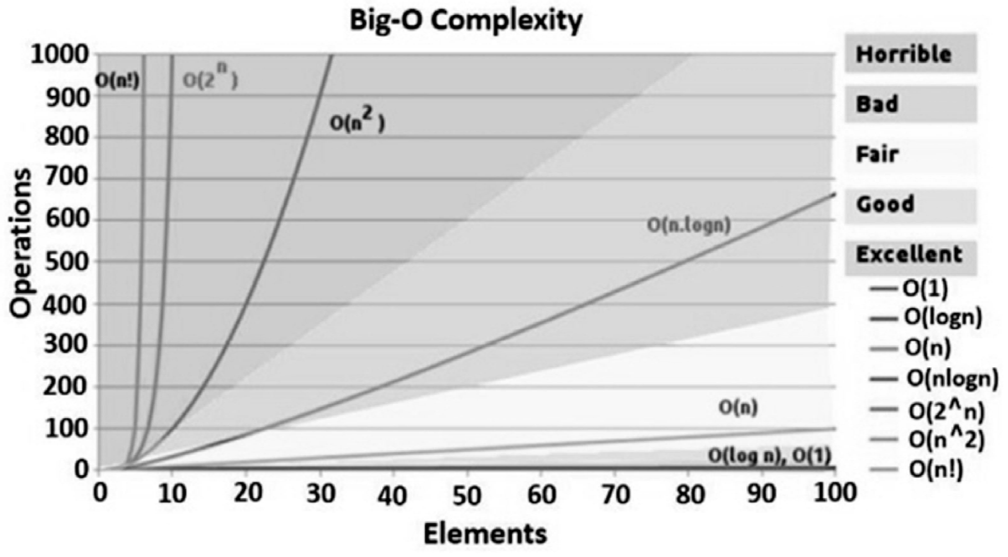
For example, $f(n) = c * n + k$ as linear time complexity. $f(n) = c * n^2 + k$ is quadratic time complexity.

The analysis of algorithms can be divided into three different cases. The cases are as follows -

Best Case - Here the lower bound of running time is calculated. It describes the behavior of algorithm under optimal conditions.

Average Case - In this case we calculate the region between upper and lower bound of the running time of algorithms. In this case the number of executed operations are not minimum and not maximum.

Worst Case - In this case we calculate the upper bound of the running time of algorithms. In this case maximum number of operations are executed.



Q27. Discuss in detail asymptotic notaions.

Ans :

The commonly used asymptotic notations used for calculating the running time complexity of an algorithm is given below:

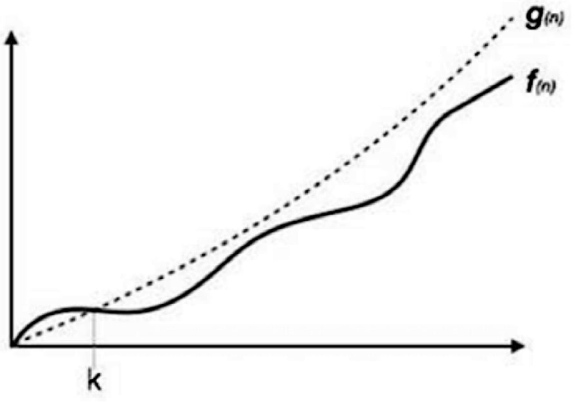
- Big oh Notation (?)
- Omega Notation (Ω)
- Theta Notation (θ)

Big oh Notation (O)

➢ Big O notation is an asymptotic notation that measures the performance of an algorithm by simply providing the order of growth of the function.

➢ This notation provides an upper bound on a function which ensures that the function never grows faster than the upper bound. So, it gives the least upper bound on a function so that the function never grows faster than this upper bound.

It is the formal way to express the upper boundary of an algorithm running time. It measures the worst case of time complexity or the algorithm's longest amount of time to complete its operation. It is represented as shown below:



For example:

If f(n) and g(n) are the two functions defined for positive integers, then f(n) = O(g(n)) as f(n) is big oh of g(n) or f(n) is on the order of g(n)) if there exists constants c and no such that:

f(n)d$\leq$c.g(n) for all n$\geq$no

This implies that f(n) does not grow faster than g(n), or g(n) is an upper bound on the function f(n). In this case, we are calculating the growth rate of the function which eventually calculates the worst time complexity of a function, i.e., how worst an algorithm can perform.

Let's understand through examples

Example 1: f(n)=2n+3 , g(n)=n

Now, we have to find Is f(n)=O(g(n))?

To check f(n)=O(g(n)), it must satisfy the given condition:

f(n)<=c.g(n)

First, we will replace f(n) by 2n+3 and g(n) by n.

2n+3 <= c.n

Let's assume c=5, n=1 then

2*1+3<=5*1

5<=5

For n=1, the above condition is true.

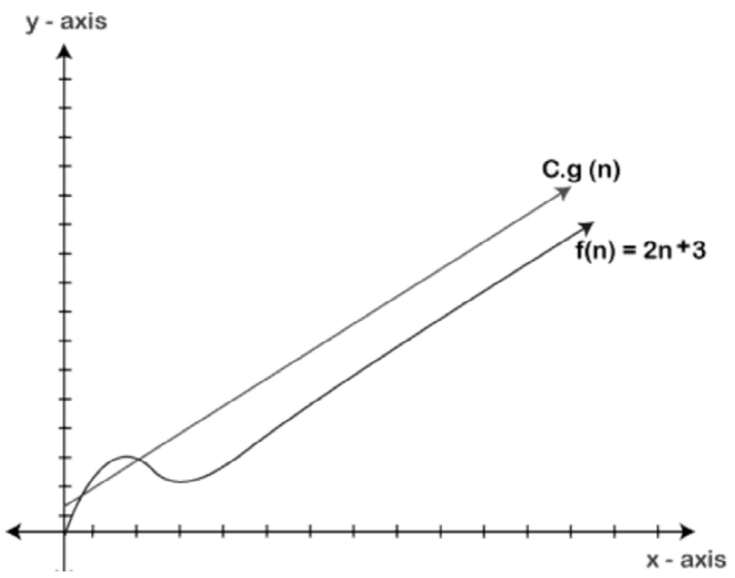
If n=2

2*2+3<=5*2

7<=10

For n=2, the above condition is true.

We know that for any value of n, it will satisfy the above condition, i.e., 2n+3<=c.n. If the value of c is equal to 5, then it will satisfy the condition 2n+3<=c.n. We can take any value of n starting from 1, it will always satisfy. Therefore, we can say that for some constants c and for some constants n0, it will always satisfy 2n+3<=c.n. As it is satisfying the above condition, so f(n) is big oh of g(n) or we can say that f(n) grows linearly. Therefore, it concludes that c.g(n) is the upper bound of the f(n). It can be represented graphically as:



The idea of using big o notation is to give an upper bound of a particular function, and eventually it leads to give a worst-time complexity. It provides an assurance that a particular function does not behave suddenly as a quadratic or a cubic fashion, it just behaves in a linear manner in a worst-case.

Omega Notation (Ω)

- It basically describes the best-case scenario which is opposite to the big o notation.
- It is the formal way to represent the lower bound of an algorithm's running time. It measures the best amount of time an algorithm can possibly take to complete or the best-case time complexity.
- It determines what is the fastest time that an algorithm can run.

If we required that an algorithm takes at least certain amount of time without using an upper bound, we use big - Ω notation i.e. the Greek letter "omega". It is used to bound the growth of running time for large input size.

If f(n) and g(n) are the two functions defined for positive integers, then f(n) = Ω (g(n)) as f(n) is Omega of g(n) or f(n) is on the order of g(n)) if there exists constants c and no such that:

f(n)>=c.g(n) for all n ≥ no and c>0

Let's consider a simple example.

If f(n) = 2n+3, g(n) = n,

Is f(n)=Ω(g(n))?

It must satisfy the condition:

f(n)>=c.g(n)

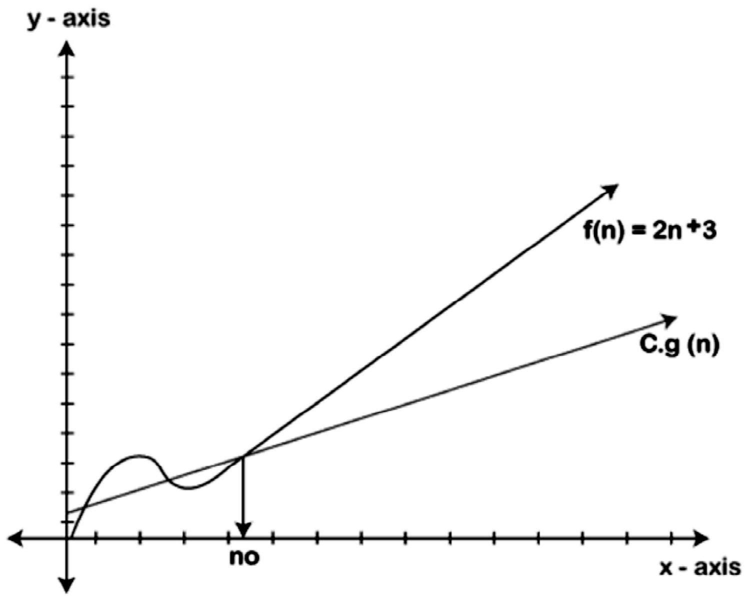
To check the above condition, we first replace f(n) by 2n+3 and g(n) by n.

2n+3>=c*n

Suppose c=1

2n+3>=n (This equation will be true for any value of n starting from 1).

Therefore, it is proved that g(n) is big omega of 2n+3 function.



As we can see in the above figure that g(n) function is the lower bound of the f(n) function when the value of c is equal to 1. Therefore, this notation gives the fastest running time. But, we are not more interested in finding the fastest running time, we are interested in calculating the worst-case scenarios because we want to check our algorithm for larger input that what is the worst time that it will take so that we can take further decision in the further process.

Theta Notation (θ)

- The theta notation mainly describes the average case scenarios.
- It represents the realistic time complexity of an algorithm. Every time, an algorithm does not perform worst or best, in real-world problems, algorithms mainly fluctuate between the worst-case and best-case, and this gives us the average case of the algorithm.
- Big theta is mainly used when the value of worst-case and the best-case is same.
- It is the formal way to express both the upper bound and lower bound of an algorithm running time.

Let's understand the big theta notation mathematically:

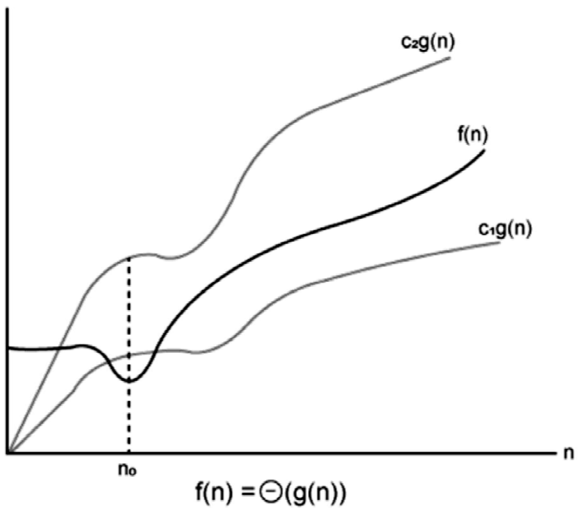
Let f(n) and g(n) be the functions of n where n is the steps required to execute the program then:

$f(n) = \theta g(n)$

The above condition is satisfied only if when

$c1.g(n) <= f(n) <= c2.g(n)$

where the function is bounded by two limits, i.e., upper and lower limit, and f(n) comes in between. The condition $f(n) = \theta g(n)$ will be true if and only if c1.g(n) is less than or equal to f(n) and c2.g(n) is greater than or equal to f(n). The graphical representation of theta notation is given below:



Let's consider the same example where

f(n)=2n+3

g(n)=n

As c1.g(n) should be less than f(n) so c1 has to be 1 whereas c2.g(n) should be greater than f(n) so c2 is equal to 5. The c1.g(n) is the lower limit of the of the f(n) while c2.g(n) is the upper limit of the f(n).

c1.g(n)<=f(n)<=c2.g(n)

Replace g(n) by n and f(n) by 2n+3

c1.n <=2n+3<=c2.n

if c1=1, c2=2, n=1

1*1 <=2*1+3 <=2*1

1<= 5 <= 2 // for n=1, it satisfies the condition c1.g(n)<=f(n)<=c2.g(n)

If n=2

1*2<=2*2+3<=2*2

2<=7<=4 // for n=2, it satisfies the condition c1.g(n)<=f(n)<=c2.g(n)

Therefore, we can say that for any value of n, it satisfies the condition c1.g(n)<=f(n) <=c2.g(n). Hence, it is proved that f(n) is big theta of g(n). So, this is the average-case scenario which provides the realistic time complexity.

1.2 Stacks

1.2.1 Concepts of Stacks and Queues, Stacks, Stack Abstract Data type

Q28. What is a Stack? List out the standard operations of Stack ?

Ans : **(June-18, Dec.-17)**

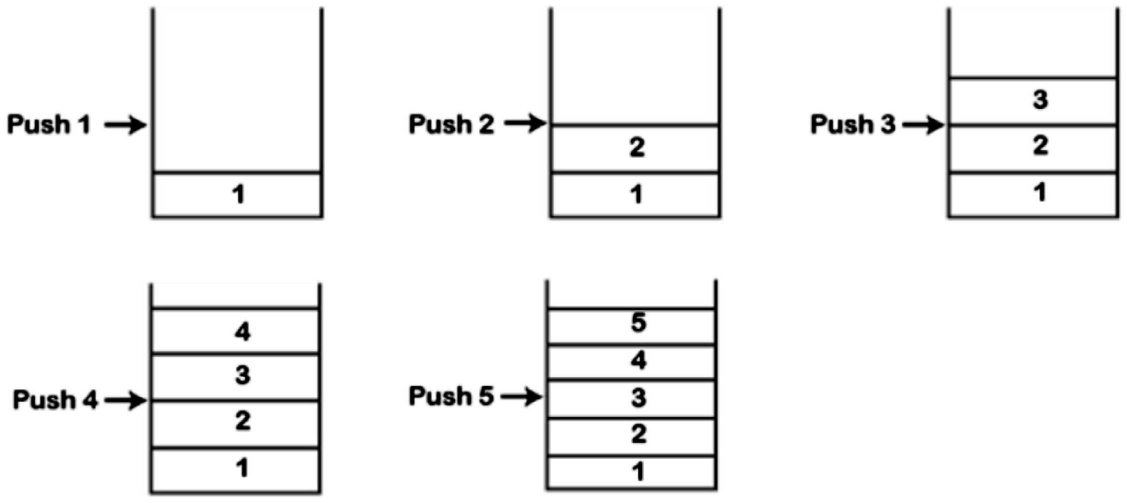
A Stack is a linear data structure that follows the LIFO (Last-In-First-Out) principle. Stack has one end, whereas the Queue has two ends (front and rear). It contains only one pointer top pointer pointing to the topmost element of the stack. Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the stack. In other words, a stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.

- It is called as stack because it behaves like a real-world stack, piles of books, etc.
- A Stack is an abstract data type with a pre-defined capacity, which means that it can store the elements of a limited size.
- It is a data structure that follows some order to insert and delete the elements, and that order can be LIFO or FILO.

Working of Stack

Stack works on the LIFO pattern. As we can observe in the below figure there are five memory blocks in the stack; therefore, the size of the stack is 5.

Suppose we want to store the elements in a stack and let's assume that stack is empty. We have taken the stack of size 5 as shown below in which we are pushing the elements one by one until the stack becomes full.



Since our stack is full as the size of the stack is 5. In the above cases, we can observe that it goes from the top to the bottom when we were entering the new element in the stack. The stack gets filled up from the bottom to the top.

When we perform the delete operation on the stack, there is only one way for entry and exit as the other end is closed. It follows the LIFO pattern, which means that the value entered first will be removed last. In the above case, the value 5 is entered first, so it will be removed only after the deletion of all the other elements.

The following are some common operations implemented on the stack:

- **push():** When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
- **pop():** When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.

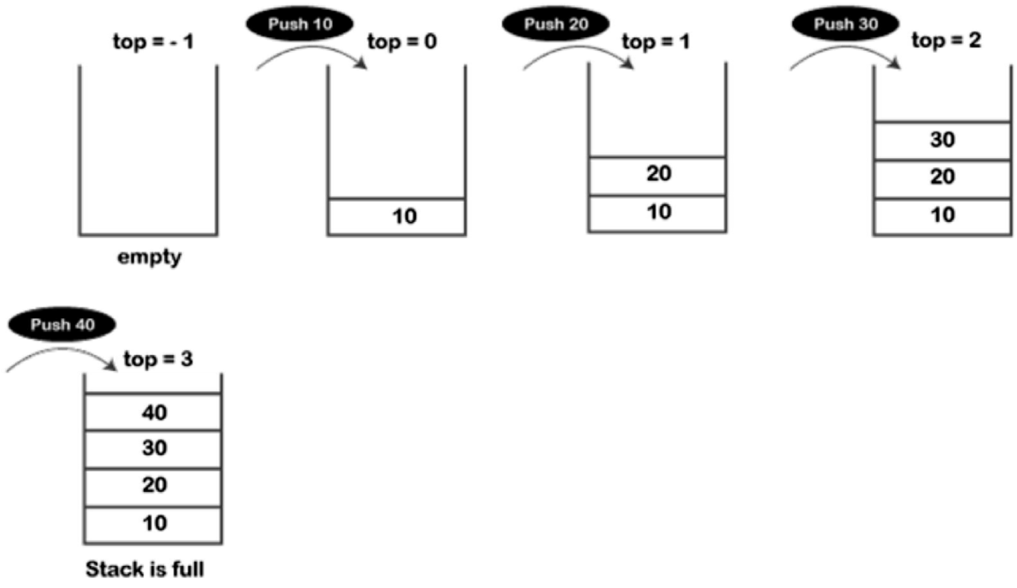
- **isEmpty():** It determines whether the stack is empty or not.
- **isFull():** It determines whether the stack is full or not.'
- **peek():** It returns the element at the given position.
- **count():** It returns the total number of elements available in a stack.
- **change():** It changes the element at the given position.
- **display():** It prints all the elements available in the stack.

Q29. Explain push and pop operations of a stack .

Ans :

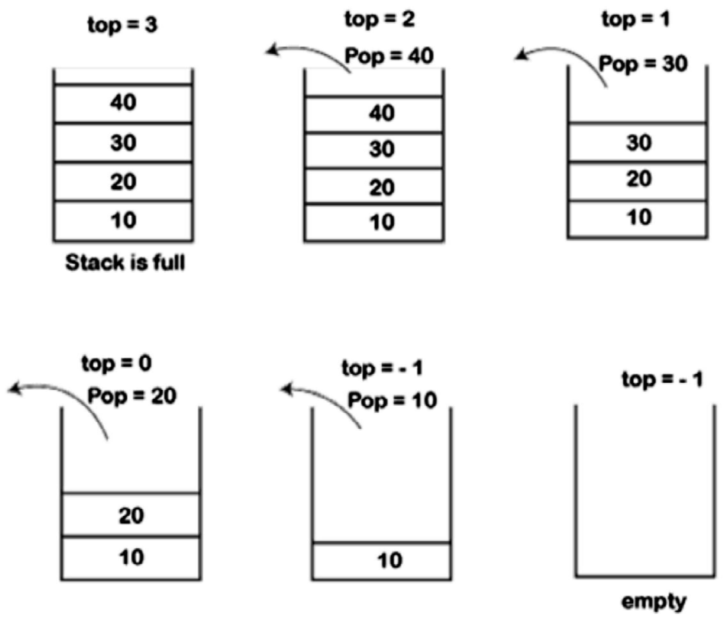
The steps involved in the PUSH operation is given below:

- Before inserting an element in a stack, we check whether the stack is full.
- If we try to insert the element in a stack, and the stack is full, then the overflow condition occurs.
- When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
- When the new element is pushed in a stack, first, the value of the top gets incremented, i.e., top=top+1, and the element will be placed at the new position of the top.
- The elements will be inserted until we reach the max size of the stack.



The steps involved in the POP operation is given below:

- Before deleting the element from the stack, we check whether the stack is empty.
- If we try to delete the element from the empty stack, then the underflow condition occurs.
- If the stack is not empty, we first access the element which is pointed by the top
- Once the pop operation is performed, the top is decremented by 1, i.e., top=top-1.



Q30. Discuss in detail Stack ADT.

Ans : **(July-21, Dec.-17)**

The abstract datatype is special kind of datatype, whose behavior is defined by a set of values and set of operations. The keyword “Abstract” is used as we can use these datatypes, we can perform different operations. But how those operations are working that is totally hidden from the user. The ADT is made of with primitive datatypes, but operation logics are hidden.

Here we will see the stack ADT. These are few operations or functions of the Stack ADT.

- isFull(), This is used to check whether stack is full or not
- isEmpry(), This is used to check whether stack is empty or not
- push(x), This is used to push x into the stack
- pop(), This is used to delete one element from top of the stack
- peek(), This is used to get the top most element of the stack
- size(), this function is used to get number of elements present into the stack

Example:

```
#include<iostream>
#include<stack>
usingnamespacestd;
main(){
    stack<int>stk;
    if(stk.empty()){
       cout<<“Stack is empty”<<endl;
    }else{
        cout<<"Stack is not empty"<<endl;
    }
    //insert elements into stack
    stk.push(10);
    stk.push(20);
    stk.push(30);
    stk.push(40);
    stk.push(50);
    cout<<"Size of the stack: "<<stk.size()<<endl;
    //pop and dispay elements
    while(!stk.empty()){
        int item =stk.top();// same as peek operation
        stk.pop();
        cout<< item <<" ";
    }
}
```

Output

```
Stack is empty
Size of the stack: 5
50 40 30 20 10
```

1.2.2 Representation of Stacks Using sequential organizations(Arrays)

Q31. Explain the representation of stack using arrays implementation.

Ans : **(June-18, Imp.)**

In array implementation, the stack is formed by using the array. All the operations regarding the stack are performed using arrays. Lets see how each operation can be implemented on the stack using array data structure.

Adding an element into the top of the stack is referred to as push operation. Push operation involves following two steps.

1. Increment the variable Top so that it can now refere to the next memory location.
2. Add element at the position of incremented top. This is referred to as adding new element at the top of the stack.

Stack is overflown when we try to insert an element into a completely filled stack therefore, our main function must always avoid stack overflow condition.

Algorithm:

```
begin
if top = n then stack full
top = top + 1
stack (top) : = item;
end
```

Deletion of an element from the top of the stack is called pop operation. The value of the variable top will be incremented by 1 whenever an item is deleted from the stack. The top most element of the stack is stored in an another variable and then the top is decremented by 1. the operation returns the deleted value that was stored in another variable as the result.

The underflow condition occurs when we try to delete an element from an already empty stack.

Algorithm :

```
begin
if top = 0 then stack empty;
item := stack(top);
top = top - 1;
end;
```

Peek operation involves returning the element which is present at the top of the stack without deleting it. Underflow condition can occur if we try to return the top element in an already empty stack.

Algorithm :

```
PEEK (STACK, TOP)
Begin
    if top = -1 then stack empty
    item = stack[top]
    return item
End
```

Example:

```
#include<iostream>
usingnamespacestd;
```

```
int stack[100], n=100, top=-1;
void push(intval){
    if(top>=n-1)
    cout<<"Stack Overflow"<<endl;
    else{
       top++;
       stack[top]=val;
    }
}
void pop(){
    if(top<=-1)
    cout<<"Stack Underflow"<<endl;
    else{
         cout<<"The popped element is "<<
stack[top]<<endl;
       top—;
    }
}
void display(){
    if(top>=0){
       cout<<"Stack elements are:";
       for(inti=top;i>=0;i—)
       cout<<stack[i]<<" ";
       cout<<endl;
    }else
    cout<<"Stack is empty";
}
int main(){
    intch,val;
    cout<<"1) Push in stack"<<endl;
    cout<<"2) Pop from stack"<<endl;
    cout<<"3) Display stack"<<endl;
    cout<<"4) Exit"<<endl;
    do{
       cout<<"Enter choice: "<<endl;
       cin>>ch;
       switch(ch){
          case1:{
                cout<<"Enter value to be
pushed:"<<endl;
             cin>>val;
             push(val);
             break;
          }
          case2:{
             pop();
             break;
          }
          case3:{
             display();
             break;
          }
          case4:{
             cout<<"Exit"<<endl;
             break;
          }
          default:{
             cout<<"Invalid Choice"<<endl;
          }
       }
    }while(ch!=4);
    return0;
}
```

Output

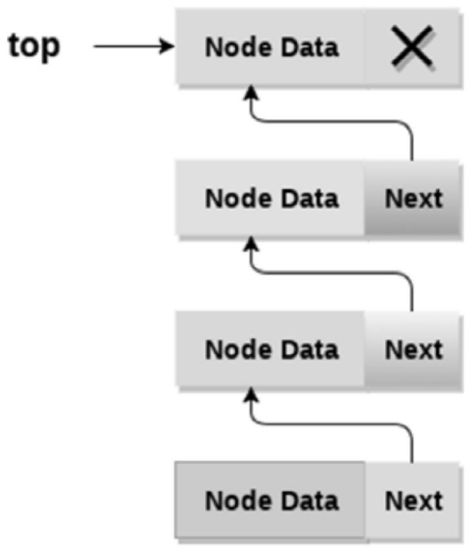
```
1)    Push in stack
2)    Pop from stack
3)    Display stack
4)    Exit
      Enter choice: 1
      Enter value to be pushed: 2
      Enter choice: 1
      Enter value to be pushed: 6
      Enter choice: 1
      Enter value to be pushed: 8
      Enter choice: 1
Enter value to be pushed: 7
Enter choice: 2
The popped element is 7
Enter choice: 3
Stack elements are:8 6 2
Enter choice: 5
Invalid Choice
Enter choice: 4
Exit
```

Q32. Explain linked list representation for implementing Stack.

Ans : **(July-21)**

Instead of using array, we can also use linked list to implement stack. Linked list allocates the memory dynamically. However, time complexity in both the scenario is same for all the operations i.e. push, pop and peek.

In linked list implementation of stack, the nodes are maintained non-contiguously in the memory. Each node contains a pointer to its immediate successor node in the stack. Stack is said to be overflown if the space left in the memory heap is not enough to create a node.

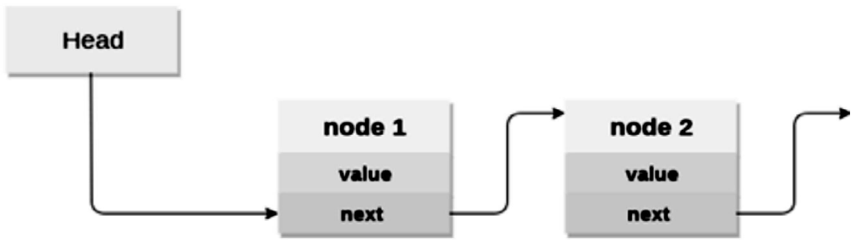


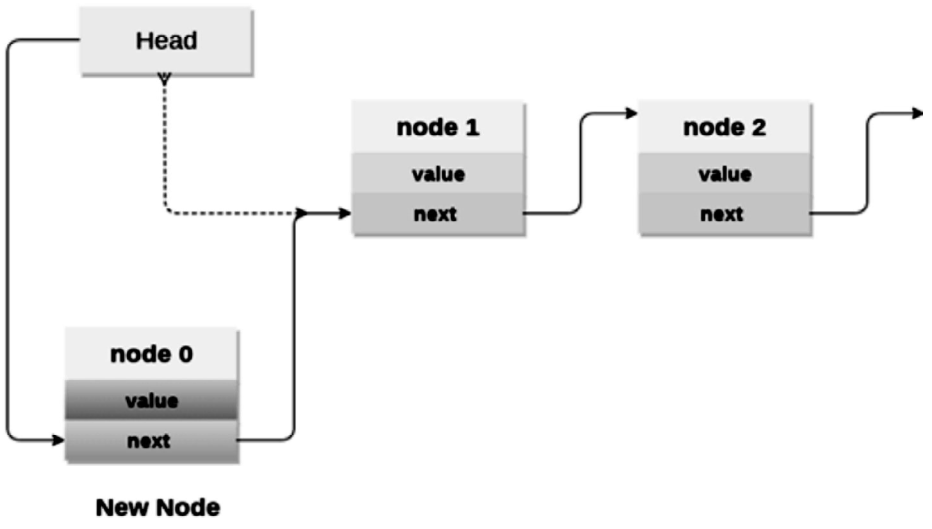
Stack

The top most node in the stack always contains null in its address field. Lets discuss the way in which, each operation is performed in linked list implementation of stack.

Adding a node to the stack is referred to as push operation. Pushing an element to a stack in linked list implementation is different from that of an array implementation. In order to push an element onto the stack, the following steps are involved.

1. Create a node first and allocate memory to it.
2. If the list is empty then the item is to be pushed as the start node of the list. This includes assigning value to the data part of the node and assign null to the address part of the node.
3. If there are some nodes in the list already, then we have to add the new element in the beginning of the list (to not violate the property of the stack). For this purpose, assign the address of the starting element to the address field of the new node and make the new node, the starting node of the list.
4. **Time Complexity : o(1)**





Deleting a node from the top of stack is referred to as pop operation. Deleting a node from the linked list implementation of stack is different from that in the array implementation. In order to pop an element from the stack, we need to follow the following steps :

1. **Check for the underflow condition:** The underflow condition occurs when we try to pop from an already empty stack. The stack will be empty if the head pointer of the list points to null.
2. **Adjust the head pointer accordingly:** In stack, the elements are popped only from one end, therefore, the value stored in the head pointer must be deleted and the node must be freed. The next node of the head node now becomes the head node.

Displaying all the nodes of a stack needs traversing all the nodes of the linked list organized in the form of stack. For this purpose, we need to follow the following steps.

1. Copy the head pointer into a temporary pointer.
2. Move the temporary pointer through all the nodes of the list and print the value field attached to every node.

Example:

```
#include<iostream>
usingnamespacestd;
structNode{
    int data;
    structNode*next;
};
structNode* top = NULL;
void push(intval){
    structNode*newnode=(structNode*)
        malloc(sizeof(structNode));
    newnode->data =val;
    newnode->next= top;
    top=newnode;
}
void pop(){
    if(top==NULL)
    cout<<"Stack Underflow"<<endl;
    else{
        cout<<"The popped element is "<< top-
>data <<endl;
        top= top->next;
    }
}
void display(){
    structNode*ptr;
    if(top==NULL)
    cout<<"stack is empty";
    else{
        ptr= top;
        cout<<"Stack elements are: ";
        while(ptr!= NULL){
            cout<<ptr->data <<" ";
            ptr=ptr->next;
        }
    }
    cout<<endl;
}
int main(){
    intch,val;
    cout<<"1) Push in stack"<<endl;
    cout<<"2) Pop from stack"<<endl;
    cout<<"3) Display stack"<<endl;
    cout<<"4) Exit"<<endl;
    do{
        cout<<"Enter choice: "<<endl;
        cin>>ch;
        switch(ch){
            case1:{
        cout<<"Enter value to be pushed:"<<endl;
                cin>>val;
                push(val);
                break;
            }
            case2:{
                pop();
                break;
            }
            case3:{
                display();
                break;
            }
            case4:{
                cout<<"Exit"<<endl;
                break;
            }
            default:{
                cout<<"Invalid Choice"<<endl;
            }
        }
    }while(ch!=4);
    return0;
}
```

Output

```
1)    Push in stack
2)    Pop from stack
3)    Display stack
4)    Exit
      Enter choice: 1
Enter value to be pushed: 2
Enter choice: 1
Enter value to be pushed: 6
Enter choice: 1
Enter value to be pushed: 8
Enter choice: 1
Enter value to be pushed: 7
Enter choice: 2
The popped element is 7
Enter choice: 3
Stack elements are:8 6 2
Enter choice: 5
Invalid Choice
Enter choice: 4
Exit
```

1.2.3 Multiple Stack

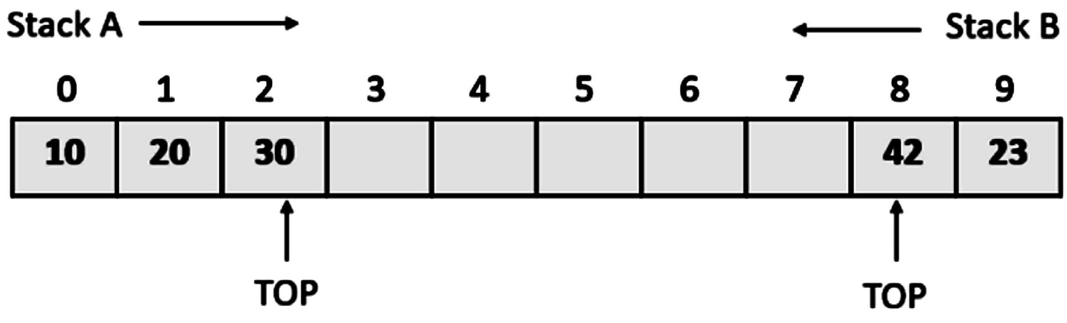
Q33. What is multiple stack ? What is the use of Multiple stack ? Explain with an example program.

Ans :

A single stack is sometime not sufficient to store large amount of data. To overcome this problem we can use multiple stack. For this, we have used a single array having more than one stack. The array is divided for multiple stacks.

Suppose, there is an array STACK[n] divided into two stack STACK A and STACK B, where n = 10.

- STACK A expands from the left to right, i.e. from 0th element.
- STACK B expands from the right to left, i.e, from 10th element.
- The combined size of both STACK A and STACK B never exceed 10.



Example:

```
#include<iostream>
using namespace std;
#define MAX 5
class Stack
{
public:
int a[MAX];
int top;
Stack()
  {
top=-1;
  }
intgetdata();
boolIsEmpty();
boolIsFull();
void push();
void pop();
};
int Stack::getdata()
{
int data;
cout<<"\nEnter the Element ";
cin>>data;
return data;
}
bool Stack::IsEmpty()
{
if(top==-1)
return 1;
return 0;
}
bool Stack::IsFull()
{
if(top==MAX)
return 1;
return 0;
}
void display(Stack &r,Stack&g,Stack&b)
{
inti;
if(r.IsEmpty())
cout<<"\nRed stack is empty";
else
{
cout<<"\nItems in  the Red stack \n";
for(i=r.top;i>=0;i—)
cout<<r.a[i]<<endl;
}
if(g.IsEmpty())
cout<<"\nGreen stack is empty";
else
{
cout<<"\nItems in  the Green stack \n";
for(i=g.top;i>=0;i—)
cout<<g.a[i]<<endl;
}
if(b.IsEmpty())
cout<<"\nBlue stack is empty";
else
{
cout<<"\nItems in  the Blue stack \n";
for(i=b.top;i>=0;i—)
cout<<b.a[i]<<endl;
 }
}
void Stack::push()
{
if(IsFull())
cout<<"\n stack is full";
else
 {
top++;
a[top]=getdata();
 }
}
void Stack::pop()
{
if(IsEmpty())
cout<<"\nstack is empty";
else
top—;
}
int main()
{
 Stack red,green,blue;
intch,temp;
do
  {
cout<<"\n\t Main menu";
cout<<"\n 1:Push red";
cout<<"\n 2:Push green";
cout<<"\n 3:Push blue";
cout<<"\n 4:Pop red";
cout<<"\n 5:Pop green";
cout<<"\n 6:Pop blue";
cout<<"\n 7:Show";
cout<<"\n 8:Exit";
cout<<"\n Enter Your Choice ";
cin>>ch;
switch(ch)
   {
case 1: red.push(); break;
case 2: green.push(); break;
case 3: blue.push(); break;
case 4: red.pop(); break;
case 5: green.pop(); break;
case 6: blue.pop(); break;
case 7: display(red,green,blue); break;
case 8: temp=1; break;
default: cout<<"\n wrong choice";
   }
}while(ch!=8 && temp!=1);
}
```

Output

```
Main menu
1.    Push red
2.    Push green
3.    Push blue
4.    Pop red
5.    Pop green
6.    Pop blue
7.    Show
8.    Exit
      Enter Your Choice 1
      Enter the Element 10
Main menu
1.    Push red
2.    Push green
3.    Push blue
4.    Pop red
5.    Pop green
6.    Pop blue
7.    Show
8.    Exit
      Enter Your Choice 1
      Enter the Element 20
Main menu
1.    Push red
2.    Push green
3.    Push blue
4.    Pop red
5.    Pop green
6.    Pop blue
7.    Show
8.    Exit
      Enter Your Choice 1
      Enter the Element 30
Main menu
1.    Push red
2.    Push green
3.    Push blue
4.    Pop red
5.    Pop green
6.    Pop blue
7.    Show
8.    Exit
      Enter Your Choice 2
      Enter the Element 100
```

```
Main menu
1.      Push red
2.      Push green
3.      Push blue
4.      Pop red
5.      Pop green
6.      Pop blue
7.      Show
8.      Exit
        Enter Your Choice 2
        Enter the Element 200
Main menu
1.      Push red
2.      Push green
3.      Push blue
4.      Pop red
5.      Pop green
6.      Pop blue
7.      Show
8.      Exit
        Enter Your Choice 7
Items in  the Red stack
        30
        20
        10
Items in  the Green stack
        200
        100
Blue stack is empty
Main menu
1.      Push red
2.      Push green
3.      Push blue
4.      Pop red
5.      Pop green
6.      Pop blue
7.      Show
8.      Exit
        Enter Your Choice 4
Main menu
1.      Push red
2.      Push green
3.      Push blue
4.      Pop red
5.      Pop green
6.      Pop blue
7.      Show
8.      Exit
        Enter Your Choice 7
Items in  the Red stack
        20
        10
Items in  the Green stack
        200
        100
Blue stack is empty
Main menu
1.      Push red
2.      Push green
3.      Push blue
4.      Pop red
5.      Pop green
6.      Pop blue
7.      Show
8.      Exit
        Enter Your Choice 8
```

1.2.4 Applications of Stack, Expression evaluation and conversion, polish notation and Expression convertion

Q34. List out the applications of Stack.

Ans :

The following are the applications of the stack:

- ➢ Balancing of symbols: Stack is used for balancing a symbol. For example, we have the following program:

```
int  main()
{
```

```
cout<<"Hello";
cout<<"Student";
}
```

As we know, each program has an opening and closing braces; when the opening braces come, we push the braces in a stack, and when the closing braces appear, we pop the opening braces from the stack. Therefore, the net value comes out to be zero. If any symbol is left in the stack, it means that some syntax occurs in a program.

- **String reversal**

  Stack is also used for reversing a string. For example, we want to reverse a "Student" string, so we can achieve this with the help of a stack.

  First, we push all the characters of the string in a stack until we reach the null character.

  After pushing all the characters, we start taking out the character one by one until we reach the bottom of the stack.

- **UNDO/REDO**

  It can also be used for performing UNDO/ REDO operations. For example, we have an editor in which we write 'a', then 'b', and then 'c'; therefore, the text written in an editor is abc. So, there are three states, a, ab, and abc, which are stored in a stack. There would be two stacks in which one stack shows UNDO state, and the other shows REDO state.

  If we want to perform UNDO operation, and want to achieve 'ab' state, then we implement pop operation.

- **Recursion**

  The recursion means that the function is calling itself again. To maintain the previous states, the compiler creates a system stack in which all the previous records of the function are maintained.

- **DFS(Depth First Search)**

  This search is implemented on a Graph, and Graph uses the stack data structure.

- **Backtracking**

  Suppose we have to create a path to solve a maze problem. If we are moving in a particular path, and we realize that we come on the wrong way. In order to come at the beginning of the path to create a new path, we have to use the stack data structure.

- **Expression conversion**

  Stack can also be used for expression conversion. This is one of the most important applications of stack. The list of the expression conversion is given below:

  Infix to prefix

  Infix to postfix

  Prefix to infix

  Prefix to postfix

  Postfix to infix

- **Memory management**

  The stack manages the memory. The memory is assigned in the contiguous memory blocks. The memory is known as stack memory as all the variables are assigned in a function call stack memory. The memory size assigned to the program is known to the compiler. When the function is created, all its variables are assigned in the stack memory. When the function completed its execution, all the variables assigned in the stack are released.

Q35. Explain various types of notations used to evaluate the expression using data structures.

Ans : **(July-21)**

The way to write arithmetic expression is known as a notation. An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are "

- Infix Notation
- Prefix (Polish) Notation
- Postfix (Reverse-Polish) Notation

These notations are named as how they use operator in expression. We shall learn the same here in this chapter.

Infix Notation

We write expression in infix notation, e.g. a – b + c, where operators are used in-between operands. It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices. An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.

Prefix Notation

In this notation, operator is prefixed to operands, i.e. operator is written ahead of operands. For example, +ab. This is equivalent to its infix notation a + b. Prefix notation is also known as Polish Notation.

Postfix Notation

This notation style is known as Reversed Polish Notation. In this notation style, the operator is postfixed to the operands i.e., the operator is written after the operands. For example, ab+. This is equivalent to its infix notation a + b.

The following table briefly tries to show the difference in all three notations -

Sr.No.	Infix Notation	Prefix Notation	Postfix Notation
1	a + b	+ a b	a b +
2	(a + b) * c	* + a b c	a b + c *
3	a * (b + c)	* a + b c	a b c + *
4	a / b + c / d	+ / a b / c d	a b / c d / +
5	(a + b) * (c + d)	* + a b + c d	a b + c d + *
6	((a + b) * c) – d	- * + a b c d	a b + c * d -

Q36. What is mean by precedence?

Ans :

Parsing Expressions

It is not a very efficient way to design an algorithm or program to parse infix notations. Instead, these infix notations are first converted into either postfix or prefix notations and then computed.

To parse any arithmetic expression, we need to take care of operator precedence and associativity also.

Precedence

When an operand is in between two different operators, which operator will take the operand first, is decided by the precedence of an operator over others. For example -

$$a + b*c \Rightarrow a + (b*c)$$

As multiplication operation has precedence over addition, b * c will be evaluated first. A table of operator precedence is provided later.

Associativity

Associativity describes the rule where operators with the same precedence appear in an expression. For example, in expression a + b " c, both + and – have the same precedence, then which part of the expression will be evaluated first, is determined by associativity of those operators. Here, both + and " are left associative, so the expression will be evaluated as (a + b) – c.

Precedence and associativity determines the order of evaluation of an expression. Following is an operator precedence and associativity table (highest to lowest) -

Sr.No.	Operator	Precedence	Associativity
1	Exponentiation ^	Highest	Right Associative
2	Multiplication (*) & Division (/)	Second Highest	Left Associative
3	Addition (+) & Subtraction (–)	Lowest	Left Associative

The above table shows the default behavior of operators. At any point of time in expression evaluation, the order can be altered by using parenthesis. For example "

In a + b*c, the expression part b*c will be evaluated first, with multiplication as precedence over addition. We here use parenthesis for a + b to be evaluated first, like (a + b)*c.

Q37. What is Polish Notation ?

Ans :

Polish notation is a notation form for expressing arithmetic, logic and algebraic equations. Its most basic distinguishing feature is that operators are placed on the left of their operands. If the operator has a defined fixed number of operands, the syntax does not require brackets or parenthesis to lessen ambiguity.

Polish notation is also known as prefix notation, prefix Polish notation, normal Polish notation, Warsaw notation and Lukasiewicz notation.

Q38. What is reverse polish notation?

Ans :

In reverse polish notation, the operator is placed after the operands like xy+, and it is also called Postfix notation.

In both polish and reverse polish notation we don't require the parentheses because all the operators are arranged in their precedence associativity rule.

^ > * = / > - = +

Q39. Discuss various notations used for conversions.

Ans :

The name comes from the Polish mathematician/logician Lukasiewicz, who introduced it. There are 3 different ways to write an algebraic expressions:

- Infix form
- Prefix form
- Postfix form

➢ **Infix form**

Is exactly the fully parenthesized notation we have just introduced. Let me remind you once again the Recursive definition

infix-expression:=(infix-expression operand infix-expression)

infix-expression:= atom

Examples

(3 * 7)

((1 + 3) * 2)

((1 + 3) * (2 - 3))

Main Feature: the binary operator is between the two operands.

Question

what if we do not put all the parentheses? Then there are ambiguities on how to interpret an expression: is 1+2*3 the same as (1+2)*3 or the same as 1+(2*3)? The precedence of operators solves this problem.

Prefix form

Main Feature:the operator preceeds the two operands.

Recursive definition of fully parenthesized version:

prefix-expression:=(operand prefix-expression prefix-expression)

prefix-expression:= atom

Recursive definition of classic version, without parentheses (we do not need them, because there is no longer any ambiguity on how to match the operands to the operators):

prefix-expression:= operand prefix-expression prefix-expression

prefix-expression:= atom

Examples

(* 3 7) or simply * 3 7

(* (+ 1 3) 2) or simply * + 1 3 2

(* (+ 1 3) (- 2 3)) or simply * + 1 3 - 2 3

Postfix form

Main Feature: the operator is after the two operands. Recursive definition

postfix-expression:=(operand postfix-expression postfix-expression) postfix-expression:= atom

Recursive definition of classic version, without parentheses (we do not need them, because there is no longer any ambiguity on how to match the operands to the operators):

postfix-expression:= operand postfix-expression postfix-expression

postfix-expression:= atom

Examples

(3 7 *) or simply 3 7 *

((1 3 +) 2 *) or simply 1 3 + 2 *

((1 3 +) (2 3 -) *) or simply 1 3 + 2 3 - *

Associativity

Associativity describes the rule where operators with the same precedence appear in an expression. For example, in the expression a + b – c, both + and – have the same precedence, then which part of the expression will be evaluated first, is determined by associativity of those operators. Here, both + and " are left associative, so the expression will be evaluated as (a + b) – c.

Precedence and associativity determine the order of evaluation of an expression. Following is an operator precedence and associativity table (highest to lowest)–

Sr.No.	Operator	Precedence	Associativity
1	Exponentiation ^	Highest	Right Associative
2	Multiplication (*) & Division (/)	Second Highest	Left Associative
3	Addition (+) & Subtraction (–)	Lowest	Left Associative

The above table shows the default behavior of operators. At any point of time in expression evaluation, the order can be altered by using parenthesis. For example-

In a + b*c, the expression part b*c will be evaluated first, with multiplication as precedence over addition. We here use parenthesis for a + b to be evaluated first, like (a + b)*c.

Examples of Infix, Prefix, Postfix

- Infix A + B, 3*x – y
- Prefix +AB, -*3xy
- Postfix AB+, 3x*y-

Converting to Infix to Postfix

A + B * C

=> A + [B*C]

=> A+[BC*]

=> A [BC*] +

=> ABC * +

Converting to Infix to Prefix

A + B * C

=> A + [B*C]

=> A+[*BC]

=> + A [*BC]

=> + A * BC

Q40. Explain infix notation in detail. What are the problems associated with infix notation?

Ans :

When the operator is written in between the operands, then it is known as infix notation. Operand does not have to be always a constant or a variable; it can also be an expression itself.

For example,

(p + q) * (r + s)

In the above expression, both the expressions of the multiplication operator are the operands, i.e., (p + q), and (r + s) are the operands.

In the above expression, there are three operators. The operands for the first plus operator are p and q, the operands for the second plus operator are r and s. While performing the operations on the expression, we need to follow some set of rules to evaluate the result. In the above expression, addition operation would be performed on the two expressions, i.e., p+q and r+s, and then the multiplication operation would be performed.

Syntax of infix notation is given below:

<operand><operator><operand>

If there is only one operator in the expression, we do not require applying any rule. For example, 5 + 2; in this expression, addition operation can be performed between the two operands (5 and 2), and the result of the operation would be 7.

If there are multiple operators in the expression, then some rule needs to be followed to evaluate the expression.

If the expression is:

4 + 6 * 2

If the plus operator is evaluated first, then the expression would look like:

10 * 2 = 20

If the multiplication operator is evaluated first, then the expression would look like:

4 + 12 = 16

The above problem can be resolved by following the operator precedence rules. In the algebraic expression, the order of the operator precedence is given in the below table:

Operators	Symbols
Parenthesis	( ),{ },[ ]
Exponents	^
Multiplication and Division	*,/
Addition and Subtraction	+,−

The first preference is given to the parenthesis; then next preference is given to the exponents. In the case of multiple exponent operators, then the operation will be applied from right to left.

For example:

2 ^ 2 ^ 3 = 2 ^ 8

= 256

After exponent, multiplication, and division operators are evaluated. If both the operators are present in the expression, then the operation will be applied from left to right.

The next preference is given to addition and subtraction. If both the operators are available in the expression, then we go from left to right.

The operators that have the same precedence termed as operator associativity. If we go from left to right, then it is known as left-associative. If we go from right to left, then it is known as right-associative.

Problem with infix notation

To evaluate the infix expression, we should know about the operator precedence rules, and if the operators have the same precedence, then we should follow the associativity rules. The use of parenthesis is very important in infix notation to control the order in which the operation to be performed. Parenthesis improves the readability of the expression. An infix expression is the most common way of writing expression, but it is not easy to parse and evaluate the infix expression without ambiguity. So, mathematicians and logicians studied this problem and discovered two other ways of writing expressions which are prefix and postfix. Both expressions do not require any parenthesis and can be parsed without ambiguity. It does not require operator precedence and associativity rules.

Q41. Discuss in detail postfix notaion. Write the algorithm to evaluate postfix notation.

Ans :

Postfix Expression

The postfix expression is an expression in which the operator is written after the operands. For example, the postfix expression of infix notation (2+3) can be written as 23+.

Some key points regarding the postfix expression are:

- In postfix expression, operations are performed in the order in which they have written from left to right.
- It does not any require any parenthesis.

- We do not need to apply operator precedence rules and associativity rules.

Algorithm to evaluate the postfix expression

1. Scan the expression from left to right until we encounter any operator.
2. Perform the operation
3. Replace the expression with its computed value.
4. Repeat the steps from 1 to 3 until no more operators exist.

Let's understand the above algorithm through an example.

Infix expression: 2 + 3 * 4

We will start scanning from the left most of the expression. The multiplication operator is an operator that appears first while scanning from left to right. Now, the expression would be:

Expression = 2 + 34*

= 2 + 12

Again, we will scan from left to right, and the expression would be:

Expression = 2 12 +

= 14

Evaluation of postfix expression using stack.

1. Scan the expression from left to right.
2. If we encounter any operand in the expression, then we push the operand in the stack.
3. When we encounter any operator in the expression, then we pop the corresponding operands from the stack.
4. When we finish with the scanning of the expression, the final value remains in the stack.

Let's understand the evaluation of postfix expression using stack.

Example 1: Postfix expression: 2 3 4 * +

Input	**Stack**	
2 3 4 * +	empty	Push 2
3 4 * +	2	Push 3
4 * +	3 2	Push 4
* +	4 3 2	Pop 4 and 3, and perform 4*3 = 12. Push 12 into the stack.
+	12 2	Pop 12 and 2 from the stack, and perform 12 + 2 = 14. Push 14 into the stack.

The result of the above expression is 14.

Algorithm to evaluate postfix expression

1. Read a character
2. If the character is a digit, convert the character into int and push the integer into the stack.
3. If the character is an operator,
   - Pop the elements from the stack twice obtaining two operands.
   - Perform the operation
   - Push the result into the stack.

Q42. Explain the process of converting infix to postfix expression.

Ans :

Algorithm to Convert Infix to Postfix Expression Using Stack

1. Initialize the Stack.
2. Scan the operator from left to right in the infix expression.
3. If the leftmost character is an operand, set it as the current output to the Postfix string.
4. And if the scanned character is the operator and the Stack is empty or contains the '(', ')' symbol, push the operator into the Stack.
5. If the scanned operator has higher precedence than the existing **precedence** operator in the Stack or if the Stack is empty, put it on the Stack.
6. If the scanned operator has lower precedence than the existing operator in the Stack, pop all the Stack operators. After that, push the scanned operator into the Stack.
7. If the scanned character is a left bracket '(', push it into the Stack.
8. If we encountered right bracket ')', pop the Stack and print all output string character until '(' is encountered and discard both the bracket.
9. Repeat all steps from 2 to 8 until the infix expression is scanned.
10. Print the Stack output.
11. Pop and output all characters, including the operator, from the Stack until it is not empty.
12. Let's translate an infix expression into postfix expression in the stack:
13. Here, we have infix expression ((A * (B + D)/E) - F * (G + H / K))) to convert into its equivalent postfix expression:

Label No.	Symbol Scanned	Stack	Expression
1	(	(	
2	(	((	
3	A	((	A
4	*	((*	A
5	(	((*(	A
6	B	((*(	AB
7	+	((*(+	AB
8	D	((*(+	ABD
9	)	((*	ABD+
10	/	((*/	ABD+
11	E	((*/	ABD+E
12	)	(	ABD+E/*
13	-	(-	ABD+E/*
14	(	(-(	ABD+E/*
15	F	(-(	ABD+E/*F
16	*	(-(*	ABD+E/*F
17	(	(-(*(	ABD+E/*F
18	G	(-(*(	ABD+E/*FG
19	+	(-(*(+	ABD+E/*FG
20	H	(-(*(+	ABD+E/*FGH
21	/	(-(*(+/	ABD+E/*FGH
22	K	(-(*(+/	ABD+E/*FGHK
23	)	(-(*	ABD+E/*FGHK/+
24	)	(-	ABD+E/*FGHK/+*
25	)		ABD+E/*FGHK/+*-

Q43. Convert the following given expression from infix to postfix A+(B*C+D)/E

Ans :

Input Token	Stack	Postfix Expression	Action
A		A	Add A into expression string
+	+	A	Push '+' into stack
(	+(	A	Push (into stack
B	+(	AB	Add B into expression string
*	+(*	AB	Push '*' into stack
C	+(*	ABC	'+' operator has less precedence than '*', so pop * and add to expression string
D	+(+	ABC*D	Add D into expression string
)	+	ABC*D+	) has come so pop + and add it to expression string.
/	+/	ABC*D+	/ has higher precedence than + so push / into stack
E	+/	ABC*D+E/+	Add E into expression string and pop all operators one by one from stack and add it to expression string.

Q44. Write a C++ program to convert infix expression to postfix expression.

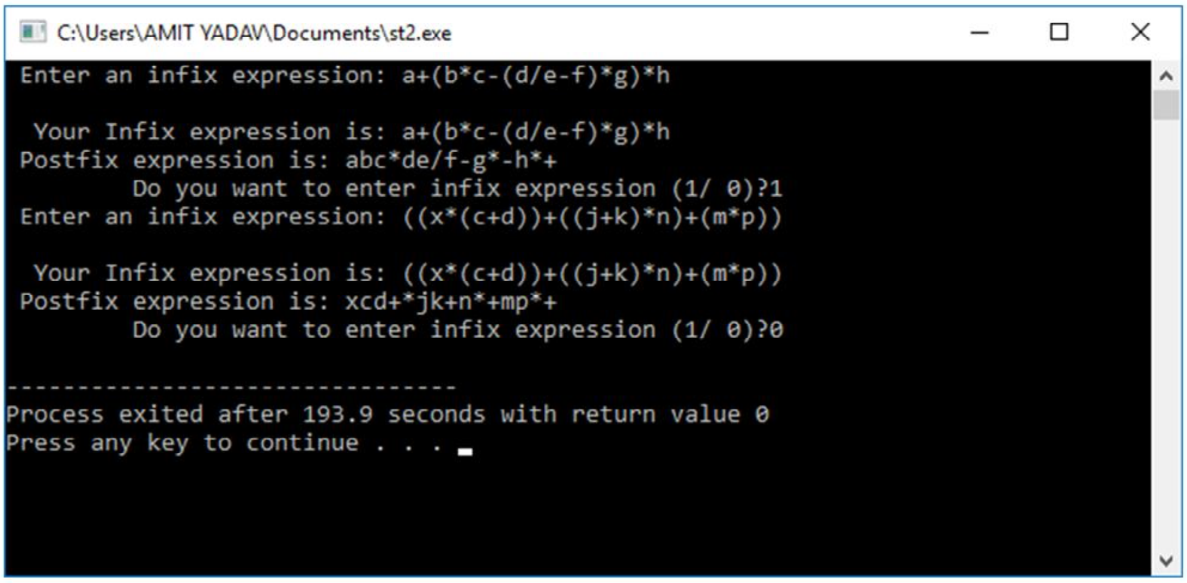
Ans :

```
#include<iostream>
#include<stack>
using namespace std;
// defines the Boolean function for operator, operand, equalOrhigher precedence and the string conversion function.
bool IsOperator(char);
bool IsOperand(char);
bool eqlOrhigher(char, char);
string convert(string);
int main()
```

```
{
string infix_expression, postfix_expression;
int ch;
do
{
cout << " Enter an infix expression: ";
cin >> infix_expression;
postfix_expression = convert(infix_expression);
cout << "\n Your Infix expression is: " << infix_expression;
cout << "\n Postfix expression is: " << postfix_expression;
cout << "\n \t Do you want to enter infix expression (1/ 0)?";
cin >> ch;
//cin.ignore();
} while(ch == 1);
return 0;
}
// define the IsOperator() function to validate whether any symbol is operator.
/* If the symbol is operator, it returns true, otherwise false. */
bool IsOperator(char c)
{
if(c == '+' || c == '-' || c == '*' || c == '/' || c == '^' )
return true;
return false;
}
// IsOperand() function is used to validate whether the character is operand.
bool IsOperand(char c)
{
if(c >= 'A' && c <= 'Z')  /
* Define the character in between A to Z. If not, it returns False. */
return true;
if (c >= 'a' && c <= 'z')
// Define the character in between a to z. If not, it returns False. */
return true;
if(c >= '0' && c <= '9')
// Define the character in between 0 to 9. If not, it returns False. */
return true;
return false;
}
// here, precedence() function is used to define the precedence to the operator.
int precedence(char op)
{
```

```
if(op == '+' || op == '-')    /* it defines the lowest precedence */
return 1;
if (op == '*' || op == '/')
return 2;
if(op == '^')
/* exponent operator has the highest precedence *
return 3;
return 0;
}
/* The eqlOrhigher() function is used to check the higher or equal precedence of the two
operators in infix expression. */
bool eqlOrhigher (char op1, char op2)
{
int p1 = precedence(op1);
int p2 = precedence(op2);
if (p1 == p2)
{
if (op1 == '^' )
return false;
return true;
}
return  (p1>p2 ? true : false);
}
/* string convert() function is used to convert the infix
expression to the postfix expression of the Stack */
string convert(string infix)
{
stack <char> S;
string postfix ="";
char ch;
S.push( '(' );
infix += ')';
for(int i = 0; i<infix.length(); i++)
{
ch = infix[i];
if(ch == ' ')
continue;
else if(ch == '(')
S.push(ch);
else if(IsOperand(ch))
postfix += ch;
else if(IsOperator(ch))
{
while(!S.empty() && eqlOrhigher(S.top(), ch))
{
postfix += S.top();
S.pop();
}
S.push(ch);
}
else if(ch == ')')
{
while(!S.empty() && S.top() != '(')
{
postfix += S.top();
S.pop();
}
S.pop();
}
}
return postfix;
}
```

Output:



```
Enter an infix expression: a+(b*c-(d/e-f)*g)*h

 Your Infix expression is: a+(b*c-(d/e-f)*g)*h
Postfix expression is: abc*de/f-g*-h*+
        Do you want to enter infix expression (1/ 0)?1
Enter an infix expression: ((x*(c+d))+((j+k)*n)+(m*p))

 Your Infix expression is: ((x*(c+d))+((j+k)*n)+(m*p))
Postfix expression is: xcd+*jk+n*+mp*+
        Do you want to enter infix expression (1/ 0)?0

--------------------------------
Process exited after 193.9 seconds with return value 0
Press any key to continue . . .
```

Q45. Explain the process of converting infix to prefix notation.

Ans :

Infix to Prefix Conversion Algorithm

Iterate the given expression from left to right, one character at a time

Step 1: First reverse the given expression

Step 2: If the scanned character is an operand, put it into prefix expression.

Step 3: If the scanned character is an operator and operator's stack is empty, push operator into operators' stack.

Step 4: If the operator's stack is not empty, there may be following possibilities.

If the precedence of scanned operator is greater than the top most operator of operator's stack, push this operator into operator 's stack.

If the precedence of scanned operator is less than the top most operator of operator's stack, pop the operators from operator's stack untill we find a low precedence operator than the scanned character.

If the precedence of scanned operator is equal then check the associativity of the operator. If associativity left to right then simply put into stack. If associativity right to left then pop the operators from stack until we find a low precedence operator.

If the scanned character is opening round bracket ('('), push it into operator's stack.

If the scanned character is closing round bracket (')'), pop out operators from operator's stack until we find an opening bracket ('(').

Repeat Step 2,3 and 4 till expression has character

Step 5: Now pop out all the remaining operators from the operator's stack and push into postfix expression.

Step 6: Exit

Infix to Prefix conversion example

Infix Expression: (A+B)+C-(D-E)^F

First reverse the given infix expression

After Reversing: F^)E-D(-C+)B+A(

Input Token	Stack	Expression	Action
F		F	Add F into expression string
^	^	F	Push ' ^ ' into stack
)	^)	FE	Push ')' into stack
E	^)	FE	Add E into expression string
–	^)-	FE	Push '–' into stack
D	^)-	FED	Add D into expression string
(	^)-	FED-	'(' Pair matched, so pop operator '–'

–	–	FED- ^	'–' operator has less precedence than ' ^ ', so pop ^ and add to expression string
C	–	FED- ^ C	Add C into expression string
+	+	FED- ^ C-	Pop-because - and + have left associativity
)	+)	FED- ^ C-	Push ‘ γ ’ into stack
B	+)	FED- ^ C-B	Add B into expression string
+	+)	FED- ^ C-B	Push ‘+’ into stack
A	+)+	FED- ^ C-BA	Add A into expression string
(	+	FED- ^ C-BA+	'(' Pair matched, so pop operator ‘+'
		FED- ^ C-BA++	Pop all operators one by one as we have reached end of the expression

Now reverse the expression to get prefix expression ++AB-C^-DEF

Q46. Write a C++ Program to convert infix to prefix expression.

Ans :

```
#include<iostream>
#include<stack>
#include<locale>//for function isalnum()
#include<algorithm>
usingnamespacestd;
intpreced(charch){
   if(ch=='+'||ch=='-'){
      return1;   //Precedence of + or - is 1
   }elseif(ch=='*'||ch=='/'){
      return2;    //Precedence of * or / is 2
   }elseif(ch=='^'){
      return3;    //Precedence of ^ is 3
   }else{
      return0;
   }
}
stringinToPost(string infix){
   stack<char>stk;
   stk.push('#');     //add some extra character to avoid underflow
   string postfix ="";   //initially the postfix string is empty
   string::iterator it;
   for(it =infix.begin(); it!=infix.end(); it++){
      if(isalnum(char(*it)))
```

```
            postfix+=*it;      //add to postfix when character is letter or number
        elseif(*it =='(')
             stk.push('(');
        elseif(*it =='^')
             stk.push('^');
        elseif(*it ==')'){
            while(stk.top()!='#'&&stk.top()!='('){
               postfix+=stk.top();      //store and pop until ( has found
               stk.pop();
            }
            stk.pop();      //remove the '(' from stack
        }else{
            if(preced(*it)>preced(stk.top()))
               stk.push(*it);      //push if precedence is high
            else{
                 while(stk.top()!='#'&&preced(*it)<=preced(stk.top())){
                    postfix+=stk.top();      //store and pop until higher precedence is found
                    stk.pop();
                 }
                 stk.push(*it);
            }
        }
    }
    while(stk.top()!='#'){
        postfix+=stk.top();      //store and pop until stack is not empty
        stk.pop();
    }
    return postfix;
}
stringinToPre(string infix){
    string prefix;
    reverse(infix.begin(),infix.end());
        / /reverse the infix expression
    string::iterator it;
    for(it =infix.begin(); it !=infix.end(); it++){      //reverse the parenthesis after reverse
        if(*it =='(')
            *it =')';
        elseif(*it ==')')
            *it ='(';
    }
    prefix=inToPost(infix);
```

```
        // convert new reversed infix to postfix form.
    reverse(prefix.begin(),prefix.end());  //again reverse the result to get final prefix form
    return prefix;
}
int main(){
    string infix =”x^y/(5*z)+2";
cout<<“Prefix Form Is: “<<inToPre (infix) <<endl;
}
```

Output

```
        Prefix Form Is: +/^xy*5z2
```

Q47. Explain the process of converting postfix to infix.

Ans :

he steps required for Postfix to Infix Conversion are as follows:

1. Scan the postfix expression from left to right.
2. Initialize an empty string stack.
3. If the scanned character is operand, push it into stack.
4. Else if the scanned character is operator, pop two strings from stack, namely, temp1 and temp2, and push: (temp2 operator temp1) into stack.
5. Repeat steps from 3 to 4 until all the characters from the string are scanned.
6. In the end, only one valid infix string will be present in the stack, pop it and return it.

Example:

abc * de – /+ = ((a + ((b * c) / (d – e))))

SYMBOL	STACK
a	a
b	a | b
c	a | b | c
*	a | (b * c)
d	a | (b * c) | d
e	a | (b * c) | d | e
–	a | (b * c) | (d – e)
/	a | ((b * c) / (d – e))
+	(a + ((b * c) / (d – e)))

Hence, the equivalent infix expression: (a+((b*c)/(d-e))).

Program :

```
/* Program for Postfix to Infix Conversion */
#include <iostream>
#include<string>
#define sizes 100
using namespace std;
class stack
{
    string item[100];
    int top;
    public:
        stack()
        {
            top=-1;
        }
        void push(string str)
        {
            if(top==sizes-1)
            {
                cout<<"stack overflow!!\n";
                return;
            }
            top++;
            item[top]=str;
        }
        string pop()
        {
            inti;
            string temp;
            if(top==-1)
            {
                cout<<"stack underflow!!\n";
                return "abc";
            }
            temp = item[top];
            top—;
            return temp;
     }
};
int main(intargc, char** argv)
{
    inti,j=0;
    stackst;
    stringpostfix,infix;
    cout<<"Enter postfix expression:\n";
    cin>>postfix;
    for(i=0;i<postfix.size();i++)
    {
if(postfix[i]=='+' || postfix[i]=='-'
|| postfix[i]=='*' || postfix[i]=='/'
|| postfix[i]==' ^ ')
        {
            string temp,op1,op2;
            op2=st.pop();
            op1=st.pop();
    temp='('+op1+postfix[i]+op2+')';
            st.push(temp);
        }
        else
        {
            string flag;
            flag=flag+postfix[i];
            st.push(flag);
        }
    }
    cout<<"The equivalent infix expression is:\n"<<st.pop();
    return 0;
}
```

Output:

```
Enter postfix expression:
abc*de-/+
The equivalent infix expression:
((a+((b*c)/(d-e))))
```

Q48. Explain the process of converting an expression from postfix to prefix with an example.

Ans :

The Steps required to convert Postfix to Prefix Expression are as follows:

1. Scan the string from left to right.
2. Initialize an empty string stack.
3. If the scanned character is operand, push it into stack.
4. Else if the scanned character is operator, pop two strings from stack, namely, temp1 and temp2, and push "operator temp1 temp2" into stack.
5. Repeat steps from 3 to 4 until all the characters of the strings are scanned.
6. In the last only a single valid prefix string will be in the stack, pop it and return it.

Example:

abc*de-/+ = +a/*bc-de

SYMBOL	STACK
a	a
b	a | b
c	a | b | c
*	a | *bc
d	a | *bc | d
e	a | *bc | d | e
–	a | *bc | –de
/	a | / * bc – de
+	+a / *bc – de

Hence, the equivalent infix expression: +a/*bc-de

Program:

```
/* Program for Postfix to Prefix Conversion */
#include <iostream>
#include<string>
#define sizes 100
using namespace std;
class stack
{
    string item[sizes];
    int top;
    public:
        stack()
```

```
            {
                  top=-1;
            }
            void push(string str)
            {
                  if(top==sizes-1)
                  {
                        cout<<"stack overflow!!";
                        return;
            }
                  top++;
                  item[top]=str;
                  }
                  string pop()
                  {
                  string temp;
                  if(top==-1)
                  {
                        cout<<"stack underflow!!";
                        return "abc";
                  }
                  temp= item[top];
                  top - -;
                  return temp;
             }
};
      int main(intargc, char** argv) {
      stack st;
      string postfix;
      inti;
      cout<<"Enter the valid postfix string:\n";
      cin>>postfix;
      for(i=0;i<postfix.size();i++)
      {
            if(postfix[i]= = '+' || postfix[i]= ='-' || postfix[i]= ='*' || postfix[i]= ='/'
            || postfix[i]= ='^')
            {
            string op1,op2,temp;
                  op2=st.pop();
                  op1=st.pop();
                  temp=postfix[i]+op1+op2;
                  st.push(temp);
      }
      else
      {
            string flag;
            flag=flag+postfix[i];
            st.push(flag);
            }
      }
      cout<<"The equivalent prefix expression:\n"<<st.pop();
      return 0;
}
```

OUTPUT

```
      Enter valid postfix expression:
      abc*de-/+
      The equivalent prefix expression:
      +a/*bc-de
```

Q49. Explain the process of converting prefix to infix with an example.

Ans :

The Steps to convert Prefix to Infix Using Stack Expression are as follows:

1. Scan the Prefix Expression form right to left.
2. Initialize the string stack.
3. If the scanned character is operand, push it into stack.
4. Else if the scanned character is operator, pop two strings from the stack namely, temp1 and temp2, then push (temp1 operator temp2) into stack.

5. Repeat steps from 3 to 4 until we scanned all the characters.
6. At the end, only infix string will be left into stack, pop and return it.

Example:

+a/*bc-de = (a+((b*c)/(d-e)))

NOTE : Read the prefix string in reverse order.

SYMBOL	STACK
e	e
d	e | d
–	(d – e)
c	(d – e) | c
b	(d – e) | c | b
*	(d – e) | (b * c)
/	((b * c) / (d – e))
a	((b * c) / (d – e)) | a
+	(a + ((b * c) / (d – e)))

```
/* Program for converting Prefix to Infix Using Stack */
# include <iostream>
# include<string>
#define sizes 100
using namespace std;
class stack
{
    string item[sizes];
    int top;
    public:
        stack()
        {
            top=-1;
    }
    void push(string st)
    {
        if(top==sizes-1)
        {
            cout<<"stack overflow!!";
```

```
                return;
            }
                top++;
                item[top]=st;
            }
            string pop()
{
                inti;
                string temp;
                if(top==-1)
                {
                    cout<<"stack underflow!!\n";
                    return "abc";
                }
                temp = item[top];
                top—;
                return temp;
            }
        };
        int main(intargc, char** argv)
        {
            inti;
            stack st;
            string prefix;
            cout<<"Enter valid prefix string: ";
            cin>>prefix;
            for(i=prefix.size()-1;i>=0;i—)
            {
        if(prefix[i]=='+' || prefix[i]=='-' || prefix[i]=='*' || prefix[i]=='/' || prefix[i]=='^')
                {
                    string op1,op2,temp;
                    op1=st.pop();
                    op2=st.pop();
                    temp='('+op1+prefix[i]+op2+')';
                    st.push(temp);
                }
                else
                {
                    string flag;
                    flag=flag+prefix[i];
                    st.push(flag);
               }
     }
     cout<<"The resultant infix string is: "<<st.pop();
     return 0;}
```

OUTPUT

```
Enter valid Prefix String: +a/*bc-de
The resultant infix string: (a+((b*c)/(d-e)))
```

Q50. Explain the process of converting prefix to postfix expression.

Ans :

The steps required for Prefix to Postfix Conversion are as follows:

1. Scan the prefix string in reverse order.
2. Initialize an empty string stack.
3. If the scanned character is operand, push it into stack.
4. Else if, the scanned character is operator, pop two strings from the stack, namely, temp1 and temp2, then push temp1 temp2 operator into stack.
5. Repeat steps from 3 to 4 until all the characters of the strings are scanned.
6. Lastly, single postfix string will be left in stack, pop it and return it.

Example:

+a / *bc – de = abc * de – / +

NOTE : Scan the prefix string in reverse order.

SYMBOL	STACK
e	e
d	e | d
–	de –
c	de– | c
b	de– | c | b
*	de– | bc *
/	bc * de – /
a	bc * de – / | a
+	abc * de – / +

Hence, the equivalent postfix expression :abc*de-/+

```
/* Program for Prefix to Postfix Conversion */
#include <iostream>
#include <string>
#define sizes 100
using namespace std;
class stack
{
        string item[sizes];
        int top;
        public:
                stack()
                {
                        top=-1;
                }
                void push(string str)
                {
                        if(top==sizes-1)
                        {
                                cout<<"stack overflow!!";
                                return;
                        }
                top++;
                item[top]=str;
}
                string pop()
                {
                        if(top==-1)
                        {
                                cout<<"stack underflow!!";
                                return "abc";
                        }
                        string temp;
                        temp=item[top];
                        top - -;
                        return temp;
            }
        };
        int main(intargc, char** argv)
        {
                inti;
            string prefix;
            stack st;
            cout<<"Enter valid prefix expression: ";
            cin>>prefix;
            for(i=prefix.size()-1;i>=0;i--)
            {
                if(prefix[i]=='+' || prefix[i]=='-' || prefix[i]=='*' || prefix[i]=='/'
                || prefix[i]=='^')
                {
                        string op1,op2,temp;
                        op1=st.pop();
                        op2=st.pop();
                        temp=op1+op2+prefix[i];
                        st.push(temp);
                }
                else
                {
                        string flag;
                        flag=flag+prefix[i];
                        st.push(flag);
                }
        }
        cout<<"The equivalent postfix expression: "<<st.pop();
        return 0;
}
```

OUTPUT

```
Enter valid prefix expression: +a/*bc-de
The equivalent Postfix expression: abc*de-/+
```

1.2.5 Processing of Function calls

Q51. What is the use of function calls in stack implementation?

Ans : **(July-19, Imp.)**

Some computer programming languages allow a module or function to call itself. This technique is known as recursion. In recursion, a function α either calls itself directly or calls a function β that in turn calls the original function α. The function α is called recursive function.

Example:

```
    int function1(int value1){
    if(value1 <1)
    return;
    function2(value1 -1);
```

```
cout<<"%d ",value1;
}
int function2(int value2){
function1(value2);
}
```

Properties

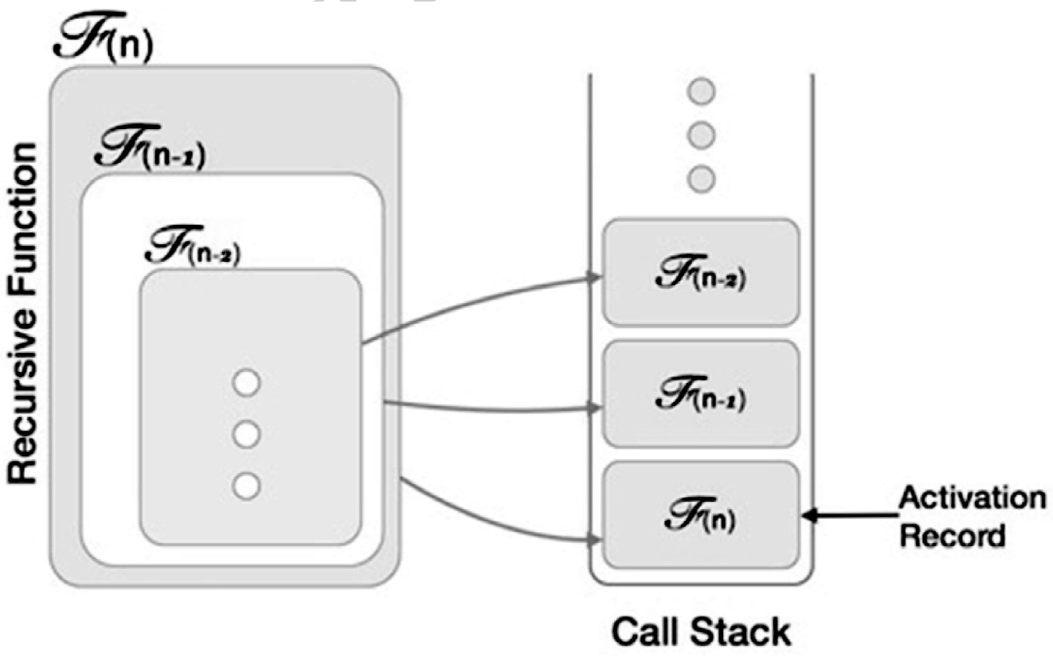
A recursive function can go infinite like a loop. To avoid infinite running of recursive function, there are two properties that a recursive function must have -

- Base criteria d There must be at least one base criteria or condition, such that, when this condition is met the function stops calling itself recursively.
- Progressive approach d The recursive calls should progress in such a way that each time a recursive call is made it comes closer to the base criteria.

Implementation

Many programming languages implement recursion by means of stacks. Generally, whenever a function (caller) calls another function (callee) or itself as callee, the caller function transfers execution control to the callee. This transfer process may also involve some data to be passed from the caller to the callee.

This implies, the caller function has to suspend its execution temporarily and resume later when the execution control returns from the callee function. Here, the caller function needs to start exactly from the point of execution where it puts itself on hold. It also needs the exact same data values it was working on. For this purpose, an activation record (or stack frame) is created for the caller function.



This activation record keeps the information about local variables, formal parameters, return address and all information passed to the caller function.

One may argue why to use recursion, as the same task can be done with iteration. The first reason is, recursion makes a program more readable and because of latest enhanced CPU systems, recursion is more efficient than iterations.

In case of iterations, we take number of iterations to count the time complexity. Likewise, in case of recursion, assuming everything is constant, we try to figure out the number of times a recursive call is being made. A call made to a function is Ï(1), hence the (n) number of times a recursive call is made makes the recursive function Ï(n).

Space complexity is counted as what amount of extra space is required for a module to execute. In case of iterations, the compiler hardly requires any extra space. The compiler keeps updating the values of variables used in the iterations. But in case of recursion, the system needs to store activation record each time a recursive call is made. Hence, it is considered that space complexity of recursive function may go higher than that of a function with iteration.

As an example, consider the following program:

```
voidbar() {
}
voidfoo() {
bar();
}
intmain() {
foo();
}
```

When the program is run, the main() function is called, so an activation record is created and added to the top of the stack. Then main() calls foo(), which places an activation record for foo() on the top of the stack. Then bar() is called, so its activation record is put on the stack. When bar() returns, its activation record is removed from the stack. Then foo() completes, removing its activation record. Finally, the activation record for main() is destroyed when the function returns. Following figure shows the state of the stack after each call and return.

The image part with relationship ID rId35 was not found in the file.

1.2.6 Reversing a string with stack

Q52. Write an algorithm to reverse a string using stack implementation.

Ans : **(July-21, July-19)**

1. Initialize a string of length n.
2. Create a stack of the same size to store the characters.
3. Traverse the string and push each and every character into the stack one by one.
4. Traverse again and start popping the characters out and concatenate them together into a string.
5. Print the reversed string.

Q53. Write a C++ program to reverse a string using stack.

Ans : **(July-21, July-19)**

```
#include <bits/stdc++.h>
usingnamespacestd;
class Stack{
public:
int top;
unsigned capacity;
char* array;
};
Stack* createStack(unsigned capacity){
Stack* stack = newStack();
stack->capacity = capacity;
stack->top = -1;
stack->array = newchar[(stack->capacity * sizeof(char))];
return stack;
}
intisFull(Stack* stack){
return stack->top == stack->capacity - 1;
}
intisEmpty(Stack* stack){
return stack->top == -1;
}
voidpush(Stack* stack, char item){
if(isFull(stack))
return;
stack->array[++stack->top] = item;
}
charpop(Stack* stack){
if(isEmpty(stack))
return -1;
return stack->array[stack->top—];
}
voidreverse(char s[]){
int n = strlen(s);
Stack* stack = createStack(n);
    inti;
    for(i = 0; i< n; i++){
push(stack, s[i]);
    }
    for(i = 0; i< n; i++){
    s[i] = pop(stack);
    }
    }
    intmain(){
    char s[] = "TutorialCup";
    reverse(s);
    cout<<s;
    return 0;
    }
```

Output

```
puClairotuT
```

Q54. Reversing the string “ABCDEF” using stack.

Ans : **(July-21, Dec.-17)**

Following are the steps involved to push the elements into the stack.

String: “ABCDEF’

STEP1:

Push(A)

TOP is A

STEP2 :

Push(B)

TOP is B

STEP3 :

Push(C)

TOP is C

STEP4 :

Push(D)

TOP is D

STEP5 :

Push(E)

TOP is E

STEP6 :

Push(F)

TOP is F

Now take other stack. Remove the elements of the first stack and insert into the second stack.

Stack 1	**Stack2**
POP(F)	Push(F)
POP(E)	Push(E)
POP(D)	Push(D)
POP(C)	Push(C)
POP(B)	Push(B)
POP(A)	Push(A)

Now the elements present in stack are displayed by using display()

Output:

“ FEDCBA”

1.2.7 Recursion

Q55. What is recursion ?

Ans :

The process in which a function calls itself is known as recursion and the corresponding function is called the recursive function. The popular example to understand the recursion is factorial function.

Factorial function: f(n) = n*f(n-1), base condition: if n<=1 then f(n) = 1. Don't worry we wil discuss what is base condition and why it is important.

In the following diagram. I have shown that how the factorial function is calling itself until the function reaches to the base condition.

Factorial function: f(n) = n*f(n–1)

Lets say we want to find out the factorial of 5 which means n = 5

f(5)=5*f(5 – 1)=5*f(4)

↓

5*4*f(4–1)=20*f(3)

↓

20*3*f(3 –1)=60*f(2)

↓

60*2*f(2–1)=120*f(1)

↓

120*1*f(1–1) = 120*f(0)

↓

120*1=120

Q56. Write a C++ Program to find factorial of a given number using recursion.

Ans :

```
#include<iostream>
usingnamespacestd;
//Factorial function
int f(int n){
/* This is called the base condition, it is
   * very important to specify the base condition
   * in recursion, otherwise your program will throw
   * stack overflow error.
   */
   if(n <=1)
        return1;
   else
```

```
            return n*f(n-1);
}
int main(){
intnum;
     cout<<"Enter a number: ";
      cin>>num;
      cout<<"Factorial of entered number: "<<f(num);
return0;
}
```

Output:

```
Enter a number:5
Factorial of entered number:120
```

Q57. List out advantages and disadvantages of recursion.

Ans :

Advantages of Recursion

- ➢ It requires few variables which make program clean.
- ➢ It shorten the complex and nested code.

Disadvantages of Recursion

- ➢ It is hard to debug recursive function.
- ➢ It is tough to understand the logic of a recursive function.
- ➢ It can cause infinite loop or unexpected results if not written correctly.
- ➢ Recursive program has greater memory space requirements than iterative program.
- ➢ Recursive programs are slower than non recursive programs.

Short Question & Answers

1. What is a Data Structure? What are the basic operations of Data Structures.

Ans :

Data Structure is a way to store and organize data so that it can be used efficiently.The data structure name indicates itself that organizing the data in memory. There are many ways of organizing the data in the memory as we have already seen one of the data structures, i.e., array in C language. Array is a collection of memory elements in which data is stored sequentially, i.e., one after another. In other words, we can say that array stores the elements in a continuous manner. This organization of data is done with the help of an array of data structures. There are also other ways to organize the data in memory.

The major or the common operations that can be performed on the data structures are:

- **Searching:** We can search for any element in a data structure.
- **Sorting:** We can sort the elements of a data structure either in an ascending or descending order.
- **Insertion:** We can also insert the new element in a data structure.
- **Updation:** We can also update the element, i.e., we can replace the element with another element.
- **Deletion:** We can also perform the delete operation to remove the element from the data structure.

2. List and explain advantages of data structures ?

Ans :

Efficiency

Efficiency of a program depends upon the choice of data structures. For example: suppose, we have some data and we need to perform the search for a perticular record. In that case, if we organize our data in an array, we will have to search sequentially element by element. hence, using array may not be very efficient here. There are better data structures which can make the search process efficient like ordered array, binary search tree or hash tables.

Reusability

Data structures are reusable, i.e. once we have implemented a particular data structure, we can use it at any other place. Implementation of data structures can be compiled into libraries which can be used by different clients.

Abstraction

Data structure is specified by the ADT which provides a level of abstraction. The client program uses the data structure through interface only, without getting into the implementation details.

3. What is Algorithm? What are the characteristics of an algorithm ?

Ans :

An algorithm is a process or a set of rules required to perform calculations or some other problem-solving operations especially by a computer. The formal definition of an algorithm is that it contains the finite set of instructions which are being carried in a specific order to perform the specific task. It is not the complete program or code; it is just a solution (logic) of a problem, which can be represented either as an informal description using a Flowchart or Pseudo code.

The following are the characteristics of an algorithm:

- **Input**

  An algorithm has some input values. We can pass 0 or some input value to an algorithm.

- **Output**

  We will get 1 or more output at the end of an algorithm.

- **Unambiguity**

  An algorithm should be unambiguous which means that the instructions in an algorithm should be clear and simple.

- **Finiteness**

  An algorithm should have finiteness. Here, finiteness means that the algorithm should contain a limited number of instructions, i.e., the instructions should be countable.

- **Effectiveness**

  An algorithm should be effective as each instruction in an algorithm affects the overall process.

- **Language independent**

  An algorithm must be language-independent so that the instructions in an algorithm can be implemented in any of the languages with the same output.

4. List out the basic characteristics of an algorithm.

Ans :

Characteristics of an Algorithm

- **Unambiguous**

  Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/ outputs should be clear and must lead to only one meaning.

- **Input**

  An algorithm should have 0 or more well-defined inputs.

- **Output**

  An algorithm should have 1 or more well-defined outputs, and should match the desired output.

- **Finiteness**

  Algorithms must terminate after a finite number of steps.

- **Feasibility**

  Should be feasible with the available resources.

- **Independent**

  An algorithm should have step-by-step directions, which should be independent of any programming code.

5. Explain how algorithms are analysed?

Ans :

The algorithm can be analyzed in two levels, i.e., first is before creating the algorithm, and second is after creating the algorithm. The following are the two analysis of an algorithm:

- **Priori Analysis**

  Here, priori analysis is the theoretical analysis of an algorithm which is done before implementing the algorithm. Various factors can be considered before implementing the algorithm like processor speed, which has no effect on the implementation part.

- **Posterior Analysis**

  Here, posterior analysis is a practical analysis of an algorithm. The practical analysis is achieved by implementing the algorithm using any programming language. This analysis basically evaluate that how much running time and space taken by the algorithm.

The performance of the algorithm can be measured in two factors:

- **Time complexity**

  The time complexity of an algorithm is the amount of time required to complete the execution. The time complexity of an algorithm is denoted by the big O notation. Here, big O notation is the asymptotic notation to represent the time complexity. The time complexity is mainly calculated by counting the number of steps to finish the execution. Let's understand the time complexity through an example.

```
sum=0;
// Suppose we have to calculate
the sum of n numbers.
for i=1 to n
sum=sum+i;
// when the loop ends then sum
```

holds the sum of the n numbers

return sum;

In the above code, the time complexity of the loop statement will be atleast n, and if the value of n increases, then the time complexity also increases. While the complexity of the code, i.e., return sum will be constant as its value is not dependent on the value of n and will provide the result in one step only. We generally consider the worst-time complexity as it is the maximum time taken for any given input size.

- **Space complexity**

  An algorithm's space complexity is the amount of space required to solve a problem and produce an output. Similar to the time complexity, space complexity is also expressed in big O notation.

  For an algorithm, the space is required for the following purposes:

  1. To store program instructions
  2. To store constant values
  3. To store variable values
  4. To track the function calls, jumping statements, etc.

Auxiliary space

The extra space required by the algorithm, excluding the input size, is known as an auxiliary space. The space complexity considers both the spaces, i.e., auxiliary space, and space used by the input.

So,

Space complexity = Auxiliary space + Input size.

6. What are the differences between Pseudocode and Algorithms.

Ans :

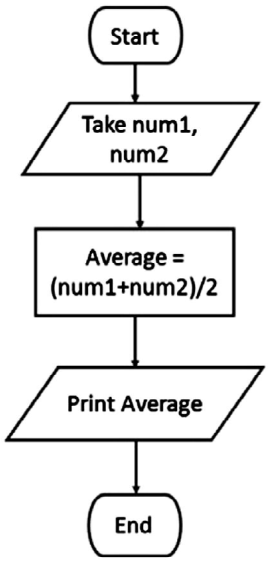
S. NO	Pseudocode	Algorithm
1.	It is the method used for writing the computer program in easy ad readable format or in the English language to make it the user or developer easier to understand the logic for further problem resolution.	It is another method of describing the computer program in a readable ad under standable with the sequence of steps written to follow while writing the code.
2.	It is said to be the representation of the algorithm as it is simpler to read and understand than the algorithm.	It is said to be the sequence of steps of explanation for the code with logical steps that makes it easier but a bit difficult to understand than pseudocode.
3.	It is one of the simpler methods or versions for writing code in any programming language.	It is one of the systematic and logical methods or versions for writing code in any progra-mming language.

4.	Pseudocode is not a programming language and is written in simple English and hence it is easy to understand and read.	Algorithms use natural languages ad flow-charts for representing any code and hence it is easy to read but difficult in understanding the algorithm.
5.	Pseudocode can be considered a method for describing the program into high-level description with the operating principle of any computer program or algorithm.	The algorithm is considered to be an unambiguous method of describing the code for which it specifies how to solve the problem.
6.	Pseudocode uses simple English language or phrases as there is no particular syntax followed to write ay pseudocode as it is the method.	The algorithm uses high-level constructs such as code snippets which may sometimes make it difficult for interpreting and under standing.
7.	Pseudocode is quite easier to construct because it is written in easy under standing simpler for debugging.	The algorithm is quite complex when constructing as it sometimes involves code snippets in it and hence it is a bit difficult when it comes to debugging.
8.	Pseudocodes are not used in any complicated programming languages as algorithms because it is widely used when the combination of programming and simple natural languages is required for constructing the pseudocode.	Algorithms can be used in any complex programming language as it uses simple logical code snippets sometimes which is more than natural language as used in Pseudocode constructions.
9.	The pseudocode also uses few specific words or phrases or special symbols to differentiate it from the algorithm, though pseudocodes can be considered as a method of writing algorithms. It includes a forward slash for beginning comments, uses flower braces for in-cluding logical blocks, italso uses some operators to specify the logic of the code.	The algorithm is written using some specific criteria such as it includes input statement, output statement, also includes with effectiveness and efficiency of the algorithm code written for a particular program, clear statements which are free from ambiguity, also includes certain or proper conditions to be displayed for correctly terminating after few steps when required in the code, and sometimes small logical codes can also be written to specify some logic directly.
10.	Pseudocode cannot be considered algorithms.	Algorithms can be considered as pseudocode.

7. Draw a flow chart to calculate the average of two numbers.

Ans :

Here is a flowchart to calculate the average of two numbers.



8. What are the differences between algorithm and flow chart.

Ans :

Flowchart	Algorithm
Block by block information diagram representing the data flow.	Step by step instruction representing the process of any solution.
It is a pictorial representation of a process.	It is a stepwise analysis of the work to be done.
The solution is shown in a graphical format.	The solution is shown in a non-computer language like English.
Easy to understand as compared to the algorithm.	It is somewhat difficult to understand.
Easy to show branching and looping.	Difficult to show branching and looping
Flowchart for a big problem is impractical	The algorithm can be written for any problem
Difficult to debug errors.	Easy to debug errors.
It is easy to make a flowchart.	It is difficult to write an algorithm as compared to a flowchart.

9. What is time complexity?

Ans :

Time complexity of an algorithm signifies the total time required by the program to run till its completion.

The time complexity of algorithms is most commonly expressed using the big O notation. It's an asymptotic notation to represent the time complexity.Time Complexity is most commonly estimated by counting the number of elementary steps performed by any algorithm to finish execution.

10. What is mean by space complexity.

Ans :

Space complexity is an amount of memory used by the algorithm (including the input values of the algorithm), to execute it completely and produce the result.

We know that to execute an algorithm it must be loaded in the main memory. The memory can be used in different forms:

- Variables (This includes the constant values and temporary values)
- Program Instruction
- Execution

Auxiliary Space

Auxiliary space is extra space or temporary space used by the algorithms during its execution.

Memory Usage during program execution

- Instruction Space is used to save compiled instruction in the memory.
- Environmental Stack is used to storing the addresses while a module calls another module or functions during execution.
- Data space is used to store data, variables, and constants which are stored by the program and it is updated during execution.

Space Complexity = Auxiliary Space + Input space

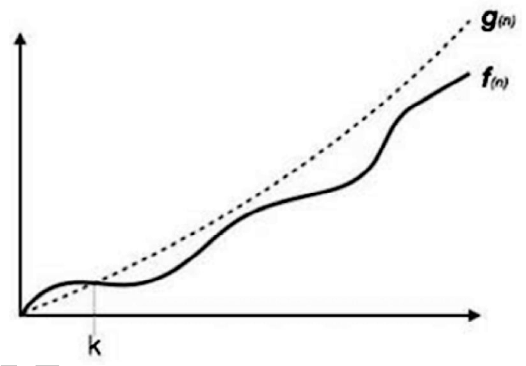
11. What is Big oh Notation?

Ans :

Big oh Notation (O)

- Big O notation is an asymptotic notation that measures the performance of an algorithm by simply providing the order of growth of the function.
- This notation provides an upper bound on a function which ensures that the function never grows faster than the upper bound. So, it gives the least upper bound on a function so that the function never grows faster than this upper bound.

It is the formal way to express the upper boundary of an algorithm running time. It measures the worst case of time complexity or the algorithm's longest amount of time to complete its operation. It is represented as shown below:



12. Define a stack.

Ans :

A Stack is a linear data structure that follows the LIFO (Last-In-First-Out) principle. Stack has one end, whereas the Queue has two ends (front and rear). It contains only one pointer top pointer pointing to the topmost element of the stack. Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the stack. In other words, a stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.

- It is called as stack because it behaves like a real-world stack, piles of books, etc.
- A Stack is an abstract data type with a pre-defined capacity, which means that it can store the elements of a limited size.
- It is a data structure that follows some order to insert and delete the elements, and that order can be LIFO or FILO.

13. What are the operations of a stack.

Ans :

The following are some common operations implemented on the stack:

- **push():** When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
- **pop():** When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.
- **isEmpty():** It determines whether the stack is empty or not.
- **isFull():** It determines whether the stack is full or not.’
- **peek():** It returns the element at the given position.
- **count():** It returns the total number of elements available in a stack.
- **change():** It changes the element at the given position.
- **display():** It prints all the elements available in the stack.

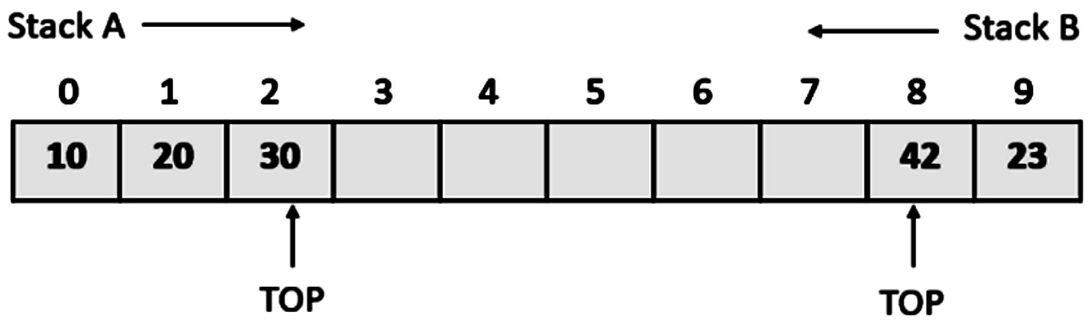
14. What is multiple stack.

Ans :

A single stack is sometime not sufficient to store large amount of data. To overcome this problem we can use multiple stack. For this, we have used a single array having more than one stack. The array is divided for multiple stacks.

Suppose, there is an array STACK[n] divided into two stack STACK A and STACK B, where n = 10.

- STACK A expands from the left to right, i.e. from 0th element.
- STACK B expands from the right to left, i.e, from 10th element.
- The combined size of both STACK A and STACK B never exceed 10.



15. Lest out the applications of a stack.

Ans :

The following are the applications of the stack:

- Balancing of symbols: Stack is used for balancing a symbol. For example, we have the following program:

```
    int  main()
    {
  cout<<"Hello";
  cout<<"Student";
    }
```

As we know, each program has an opening and closing braces; when the opening braces come, we push the braces in a stack, and when the closing braces appear, we pop the opening braces from the stack. Therefore, the net value comes out to be zero. If any symbol is left in the stack, it means that some syntax occurs in a program.

- **String reversal**

  Stack is also used for reversing a string. For example, we want to reverse a "Student" string, so we can achieve this with the help of a stack.

  First, we push all the characters of the string in a stack until we reach the null character.

  After pushing all the characters, we start taking out the character one by one until we reach the bottom of the stack.

- **UNDO/REDO**

  It can also be used for performing UNDO/REDO operations. For example, we have an editor in which we write 'a', then 'b', and then 'c'; therefore, the text written in an editor is abc. So, there are three states, a, ab, and abc, which are stored in a stack. There would be two stacks in which one stack shows UNDO state, and the other shows REDO state.

  If we want to perform UNDO operation, and want to achieve 'ab' state, then we implement pop operation.

- **Recursion**

  The recursion means that the function is calling itself again. To maintain the previous states, the compiler creates a system stack in which all the previous records of the function are maintained.

- **DFS(Depth First Search)**

  This search is implemented on a Graph, and Graph uses the stack data structure.

- **Backtracking**

  Suppose we have to create a path to solve a maze problem. If we are moving in a particular path, and we realize that we come on the wrong way. In order to come at the beginning of the path to create a new path, we have to use the stack data structure.

- **Expression conversion**

  Stack can also be used for expression conversion. This is one of the most important applications of stack. The list of the expression conversion is given below:

  Infix to prefix

  Infix to postfix

  Prefix to infix

Prefix to postfix

Postfix to infix

➢ **Memory management**

The stack manages the memory. The memory is assigned in the contiguous memory blocks. The memory is known as stack memory as all the variables are assigned in a function call stack memory. The memory size assigned to the program is known to the compiler. When the function is created, all its variables are assigned in the stack memory. When the function completed its execution, all the variables assigned in the stack are released.

16. Explain various types of notations used to evaluate the expression using data structures.

Ans :

The way to write arithmetic expression is known as a notation. An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are -

- Infix Notation
- Prefix (Polish) Notation
- Postfix (Reverse-Polish) Notation

These notations are named as how they use operator in expression. We shall learn the same here in this chapter.

Infix Notation

We write expression in infix notation, e.g. a – b + c, where operators are used in-between operands. It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices. An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.

Prefix Notation

In this notation, operator is prefixed to operands, i.e. operator is written ahead of operands. For example, +ab. This is equivalent to its infix notation a + b. Prefix notation is also known as Polish Notation.

Postfix Notation

This notation style is known as Reversed Polish Notation. In this notation style, the operator is postfixed to the operands i.e., the operator is written after the operands. For example, ab+. This is equivalent to its infix notation a + b.

The following table briefly tries to show the difference in all three notations -

Sr.No.	Infix Notation	Prefix Notation	Postfix Notation
1	a + b	+ a b	a b +
2	(a + b) * c	* + a b c	a b + c *
3	a * (b + c)	* a + b c	a b c + *
4	a / b + c / d	+ / a b / c d	a b / c d / +
5	(a + b) * (c + d)	* + a b + c d	a b + c d + *
6	((a + b) * c) – d	- * + a b c d	a b + c * d -

17. What is Polish Notation ?

Ans :

Polish notation is a notation form for expressing arithmetic, logic and algebraic equations. Its most basic distinguishing feature is that operators are placed on the left of their operands. If the operator has a defined fixed number of operands, the syntax does not require brackets or parenthesis to lessen ambiguity.

Polish notation is also known as prefix notation, prefix Polish notation, normal Polish notation, Warsaw notation and Lukasiewicz notation.

18. What is reverse polish notation?

Ans :

In reverse polish notation, the operator is placed after the operands like xy+, and it is also called Postfix notation.

In both polish and reverse polish notation we don't require the parentheses because all the operators are arranged in their precedence associativity rule.

^ > * = / > - = +

19. Discuss in detail postfix notaion. Write the algorithm to evaluate postfix notation.

Ans :

Postfix Expression

The postfix expression is an expression in which the operator is written after the operands. For example, the postfix expression of infix notation (2+3) can be written as 23+.

Some key points regarding the postfix expression are:

- In postfix expression, operations are performed in the order in which they have written from left to right.
- It does not any require any parenthesis.
- We do not need to apply operator precedence rules and associativity rules.

Algorithm to evaluate the postfix expression

1. Scan the expression from left to right until we encounter any operator.
2. Perform the operation
3. Replace the expression with its computed value.
4. Repeat the steps from 1 to 3 until no more operators exist.

Let's understand the above algorithm through an example.

Infix expression: 2 + 3 * 4

We will start scanning from the left most of the expression. The multiplication operator is an operator that appears first while scanning from left to right. Now, the expression would be:

Expression = 2 + 34*

= 2 + 12

Again, we will scan from left to right, and the expression would be:

Expression = 2 12 +

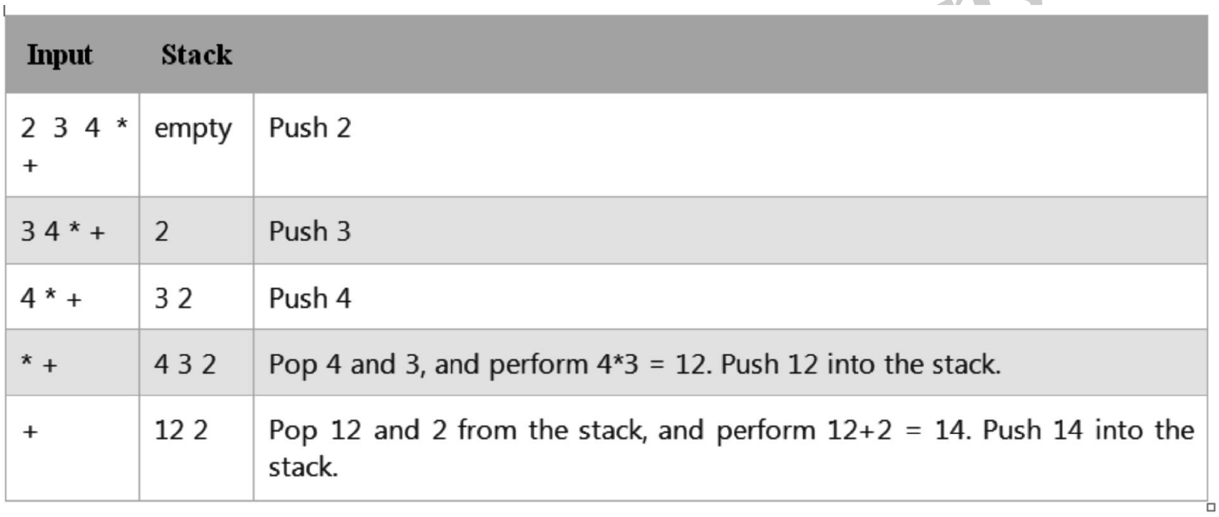
= 14

Evaluation of postfix expression using stack.

1. Scan the expression from left to right.
2. If we encounter any operand in the expression, then we push the operand in the stack.
3. When we encounter any operator in the expression, then we pop the corresponding operands from the stack.
4. When we finish with the scanning of the expression, the final value remains in the stack.

Let's understand the evaluation of postfix expression using stack.

Example 1: Postfix expression: 2 3 4 * +



Input	Stack	
2 3 4 * +	empty	Push 2
3 4 * +	2	Push 3
4 * +	3 2	Push 4
* +	4 3 2	Pop 4 and 3, and perform 4*3 = 12. Push 12 into the stack.
+	12 2	Pop 12 and 2 from the stack, and perform 12+2 = 14. Push 14 into the stack.

The result of the above expression is 14.

Algorithm to evaluate postfix expression

1. Read a character
2. If the character is a digit, convert the character into int and push the integer into the stack.
3. If the character is an operator,
   - Pop the elements from the stack twice obtaining two operands.
   - Perform the operation
   - Push the result into the stack.

20. Write an algorithm to reverse a string using stack implementation.

Ans :

1. Initialize a string of length n.
2. Create a stack of the same size to store the characters.
3. Traverse the string and push each and every character into the stack one by one.
4. Traverse again and start popping the characters out and concatenate them together into a string.
5. Print the reversed string.

21. What is recursion ?

Ans :

The process in which a function calls itself is known as recursion and the corresponding function is called the recursive function. The popular example to understand the recursion is factorial function.

Factorial function: f(n) = n*f(n-1), base condition: if n<=1 then f(n) = 1. Don't worry we wil discuss what is base condition and why it is important.

In the following diagram. I have shown that how the factorial function is calling itself until the function reaches to the base condition.

Factorial function: f(n) = n*f(n–1)

Lets say we want to find out the factorial of 5 which means n = 5

f(5)=5*f(5 – 1)=5*f(4)

↓

5*4*f(4–1)=20*f(3)

↓

20*3*f(3 –1)=60*f(2)

↓

60*2*f(2–1)=120*f(1)

↓

120*1*f(1–1) = 120*f(0)

↓

120*1=120

22. Transform the following infix expression into the equivalent postfix expression.

Ans :

(i) A + B * (C+D) / F + D * E

A+B*(C+D)F/+D*E

A+B*(C+D)F/+DE*

A+B*(C+D)F/DE*+

A+B(C+D)*F/DE*+

A+BCD*F/+DE*+

ABCD+*F/+DE*+

(ii) (A+B ^ D) / (E-F) + G

(A + B ^ D) / (E – F) + G

= (A + B ^ D)(E – F)/ + G

= (A + B ^ D)(E – F)/G+

= A + BD ^ (E – F)/G+

= ABD ^ +EF-/G+

23. What are the differences between Atomic and composite data with examples?

Ans :

Atomic Data

The Atomic data is the data which is viewed as single and non decomposable entity by the user. Due to its numerical properties, the atomic data can also be called as Scalar data.

Example: Consider an integer 5678.It can be decomposed into single digit i.e., (5, 6, 7, 8). However, after decomposition these digits doesn't hold the same characteristics as the actual integer i.e.,(5678)does. Thus,the Atomic data is considered as single and non decomposable data.

Composite Data

The composite data is defined as the data which can be decomposed into several meaningful subfield.It can be also be called as structured data and it is implemented in C++ by using structure,class etc..

Example: Consider an employee's record which contains fields such as employee ID, name, salary etc.. Each employee record is decomposed into several sub-fields(i.e., employee ID,name, salary). Thus, the composite data can be decomposed without any change in its meaning.

24. What is sequential organization ? Briefly explain its advantages and dis-advantages.

Ans :

The Sequential file organization is a popular file organization in the database management system (DBMS). It is a simple technique for file organization structure. This technique stores the data element in the sequence manner that is organized one after another in binary format. The File organization in DBMS supports various data operations such as insert, update, delete, and retrieve the data.

Advantages

- ➢ The sequential file organization is efficient and process faster for the large volume of data.
- ➢ It is a simple file organization compared to other available file organization methods.
- ➢ This method can be implemented using cheaper storage devices such as magnetic tapes.
- ➢ It requires fewer efforts to store and maintain data elements.
- ➢ The sequential file organization technique is useful for report generation and statistical computation process.
- ➢ This file organization is a preferred method for calculating aggregates that involve most of the data elements that have to be accessed while performing the computation process. Some of the popular use cases are calculating grades for the students, generating payslips for the employees, and generating the invoices in the business.

Disadvantages

- ➢ The shorting operation is a time-consuming process and more memory space for the shorted file method in the sequential file organization.
- ➢ The shorting operations iterate for every writes operation such as insert, update, or delete.
- ➢ The traversing time is high in the sequential file organization as for each writes operation, the system or the program control cannot find a particular data item directly at one go, it has to traverse through the sequence of data items.

25. Define a data structure in terms of the triplet (D,F,A) give an example.

Ans :

The data structure can be defined as the collection of elements and all the possible operations which are required for those set of elements. Formally data structure can be defined as a data structure is a set of domains D, a set of domains F and a set of axioms A. this triple (D, F, A) denotes the data structure d.

26. What are the advantages of prefix and postfix expressions?

Ans :

Advantages of infix

- Much easier to translate to a format that is suitable for direct execution. Either format can trivially be turned into a tree for further processing, and postfix can be directly translated to code if you use a stack-based processor or virtual machine
- Entirely unambiguous. Infix notation requires precedence and associativity rules to disambiguate it, or addition of extra parentheses that are not usually considered part of the notation. As long as the number of arguments to each operator are known in advance, both prefix and postfix notation are entirely unambiguous: "* + 5 6 3" is (5+6)*3, and cannot be interpreted as 5+(6*3), whereas parenthesis is required to achieve with infix.
- Supports operators with different numbers of arguments without variation of syntax. "unary-op 5" and "ternary-op 1 2 3" both work fine, but need special syntax to make them work in infix.

Advantages of postfix

Postfix notation is when we use operator after the operands. Let's discuss the advantages are:

1. Any formula can be expressed without parenthesis.
2. It is very convenient for evaluating formulas on computer with stacks.
3. Postfix expression doesn't has the operator precedence.
4. Postfix is slightly easier to evaluate.
5. It reflects the order in which operations are performed.
6. You need to worry about the left and right associativity.

Choose the Correct Answers

1. How can we describe an array in the best possible way? [c]
   (a) The Array shows a hierarchical structure
   (b) Arrays are immutable.
   (c) Container that stores the elements of similar types
   (d) The Array is not a data structure

2. When the user tries to delete the element from the empty stack then the condition is said to be a ________ [a]
   (a) Underflow (b) Garbage collection
   (c) Overflow (d) None of the above

3. If the size of the stack is 10 and we try to add the 11th element in the stack then the condition is known as ________ [c]
   (a) Underflow (b) Garbage collection
   (c) Overflow (d) None of the above

4. Which one of the following is not the application of the stack data structure [d]
   (a) String reversal (b) Recursion
   (c) Backtracking (d) Asynchronous data transfer

5. Which data structure is mainly used for implementing the recursive algorithm? [b]
   (a) Queue (b) Stack
   (c) Binary tree (d) Linked list

6. Which data structure is required to convert the infix to prefix notation? [a]
   (a) Stack (b) Linked list
   (c) Binary tree (d) Queue

7. Which of the following is the infix expression? [a]
   (a) A+B*C (b) +A*BC
   (c) ABC+* (d) None of the above

8. Which of the following is the prefix form of A+B*C? [d]
   (a) A+(BC*) (b) +AB*C
   (c) ABC+* (d) +A*BC

9. If the elements '1', '2', '3' and '4' are added in a stack, so what would be the order for the removal? [c]

(a) 1234 (b) 2134

(c) 4321 (d) None of the above

10. What is the outcome of the prefix expression +, -, *, 3, 2, /, 8, 4, 1? [c]

(a) 12 (b) 11

(c) 5 (d) 4

Fill in the blanks

1. The process in which a function calls itself is known as ________
2. ________ is a way to store and organize data so that it can be used efficiently.
3. A data structure is called linear if all of its elements are arranged in the ________
4. An ________ is a process or a set of rules required to perform calculations or some other problem-solving operations especially by a computer.
5. An algorithm should be ________which means that the instructions in an algorithm should be clear and simple.
6. ________ of an algorithm is the amount of time required to complete the execution.
7. An algorithm's ________ is the amount of space required to solve a problem and produce an output.
8. Space complexity = ________
9. ________ is a diagrammatic representation of sequence of logical steps of a program
10. An ________ is a structure of fixed-size, which can hold items of the same data type.

ANSWESRS

1. Recursion
2. Data Structure
3. Linear order.
4. Algorithm
5. Unambiguous
6. The time complexity
7. Space complexity
8. Auxiliary space + Input size.
9. Flowchart
10. Array

One Mark Answers

1. Linear data structure.

Ans :

A data structure is called linear if all of its elements are arranged in the linear order. In linear data structures, the elements are stored in non-hierarchical way where each element has the successors and predecessors except the first and last element.

2. Stack.

Ans :

Stack is a linear list in which insertion and deletions are allowed only at one end, called top.

3. Algorithm.

Ans :

The formal definition of an algorithm is that it contains the finite set of instructions which are being carried in a specific order to perform the specific task.

4. List any 2 disadvantages of pseudo code.

Ans :

- The visual representation of the programming code can be easily understood, and the pseudocode doesn't provide it.
- There is no well-defined format to write the pseudocode.
- There are no standards available for pseudocode. Companies use their own standards to write it.
- If we use pseudocode, we need to maintain one more document for our code.

5. Flow chart?

Ans :

Flowchart is a diagrammatic representation of sequence of logical steps of a program. Flowcharts use simple geometric shapes to depict processes and arrows to show relationships and process/data flow.

UNIT II

Recursion: Introduction, Recurrence, Use of Stack in Recursion, Variants of Recursion, Recursive Functions, Iteration versus Recursion.

Queues: Concept of Queues, Queue as Abstract Data Type, Realization of Queues Using Arrays, Circular Queue, Multi-queues, Dequeue, Priority Queue, Applications of Queues,

Linked Lists: Introduction, Linked List, Linked List Abstract Data Type, Linked List Variants, Doubly Linked List, Circular Linked List, Representation of Sparse Matrix Using Linked List, Linked Stack, Linked Queue.

2.1 Recursion

2.1.1 Introduction

Q1. What is Recursion? Explain with an example.

Ans :

The process in which a function calls itself is known as recursion and the corresponding function is called the recursive function. The popular example to understand the recursion is factorial function.

Factorial function

f(n) = n*f(n-1), base condition: if n<=1 then f(n) = 1. Don't worry we will discuss what is base condition and why it is important.

In the following diagram. I have shown that how the factorial function is calling itself until the function reaches to the base condition.

Factorial function:f(n) = n*f(n–1)

Lets say we want to find out the factorial of 5 which means n = 5.

```
f(5)=5*f(5–1)=5*f(4)
                  ↓
              5*4*f(4–1)=  20*f(3)
                                ↓
                          20*3*f(3-1)=60*f(2)
                                           ↓
                                   60*2*f(2-1)  =  120*f(1)
                                                       ↓
                                            120*1*f(1-1)  =  120*f(0)
                                                                  ↓
                                                            120*1=120
```

```
#include<iostream>
usingnamespace std;
//Factorial function
int f(int n){
/* This is called the base condition, it is
       * very important to specify the base condition
       * in recursion, otherwise your program will throw
       * stack overflow error.
*/
```

```
    if (n <= 1)
          return1;
    else
         return n*f(n-1);
}
int main(){
int num;
        cout<<"Enter a number: ";
        cin>>num;
        cout<<"Factorial of entered number: "<<f(num);
return0;
}
```

Output

```
Enter a number: 5
Factorial of entered number: 120
```

Base condition

In the above program, you can see that I have provided a base condition in the recursive function. The condition is:

```
if(n <=1)
      return1;
```

The purpose of recursion is to divide the problem into smaller problems till the base condition is reached. For example in the above factorial program I am solving the factorial function f(n) by calling a smaller factorial function f(n-1), this happens repeatedly until the n value reaches base condition(f(1)=1). If you do not define the base condition in the recursive function then you will get stack overflow error.

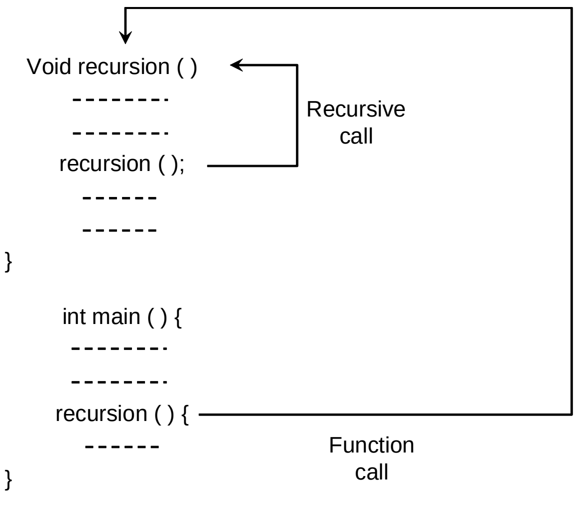
Q2. How does a recursion works?

Ans :

The function of the recursion approach is to solve the problem effectively in comparison to another problem-solving approach. The recursion process divides the problem into the subtask as a function and continuously calls the same function again to get one step closer to the final solution. This process continues until we find the final solution to the problem. Each time the part of the solution is found, it is stored in the form of stack data structure in the memory and at last, popped out to get the final solution. As we approach the final solution, there is a base condition that should be checked to get out of the recursion process. This base condition is checked using the conditional statement and hence avoids the recursion process to get into the infinite loop. If for any reason the base case fails to work, the program will fall into an infinite loop and we will never reach the solution of the program. Below is the working of recursion in C++.

```
voidrecursion()
{
        ... .. ...
        recursion();
        ... .. ...
}
intmain()
{
        ... .. ...
        recursion();
        ... .. ...
}
```

The figure below shows how recursion works by calling the recursive function again and again.



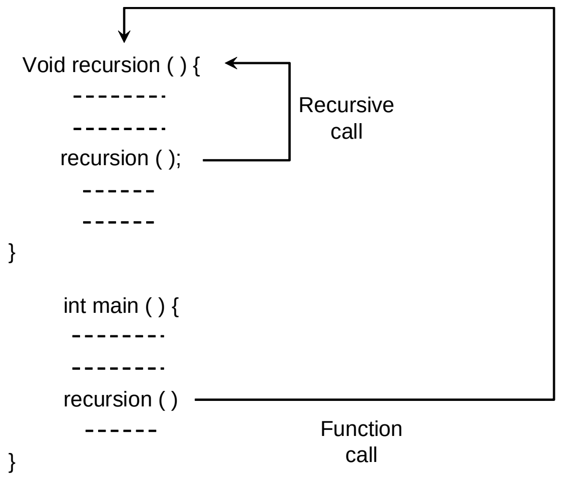
There are two types of recursive function i.e. direct recursion and indirect recursion. Direct recursion is when the function calls itself just like we saw in the above program. Indirect recursion is when the function calls another function and then that function calls the calling function.

The function of the recursion approach is to solve the problem effectively in comparison to another problem-solving approach. The recursion process divides the problem into the subtask as a function and continuously calls the same function again to get one step closer to the final solution. This process continues until we find the final solution to the problem. Each time the part of the solution is found, it is stored in the form of stack data structure in the memory and at last, popped out to get the final solution. As we approach the final solution, there is a base condition that should be checked to get out of the recursion process. This base condition is checked using the conditional statement and hence avoids the recursion process to get into the infinite loop. If for any reason the base case fails to work, the program will fall into an infinite loop and we will never reach the solution of the program. Below is the working of recursion in C++.

```
voidrecursion()
{
        ... .. ...
        recursion();
        ... .. ...
}
intmain()
{
        ... .. ...
        recursion();
        ... .. ...
}
```

The figure below shows how recursion works by calling the recursive function again and again.

There are two types of recursive function i.e. direct recursion and indirect recursion. Direct recursion is when the function calls itself just like we saw in the above program. Indirect recursion is when the function calls another function and then that function calls the calling function.



Q3. What is the main difference between direct recursion and indirect recursion.

Ans :

Direct recursion

When function calls itself, it is called direct recursion, the example we have seen above is a direct recursion example.

Indirect recursion

When function calls another function and that function calls the calling function, then this is called indirect recursion. For example: function A calls function B and Function B calls function A.

Example for indirect recursion

```
#include<iostream>
usingnamespace std;
int fa(int);
int fb(int);
int fa(int n){
if(n<=1)
return1;
else
return n*fb(n-1);
}
int fb(int n){
if(n<=1)
return1;
```

```
        else
        return n*fa(n-1);
        }
        int main(){
        int num=5;
              cout<<fa(num);
        return0;
        }
```

Output

```
120
```

Q4. List out the advantages and disadvantages of recursion ?

Ans :

Advantages

- ➢ Less number code lines are used in the recursion program and hence the code looks shorter and cleaner.
- ➢ Recursion is easy to approach to solve the problems involving data structure and algorithms like graph and tree.
- ➢ Recursion helps to reduce the time complexity.
- ➢ It helps to reduce unnecessary calling of the function.
- ➢ It helps to solve the stack evolutions and prefix, infix, postfix evaluation.
- ➢ Recursion is the best method to define objects that have repeated structural forms.

Disadvantages

- ➢ It consumes a lot of stack space
- ➢ It takes more time to process the program
- ➢ If an error is accrued in the program, it is difficult to debug the error in comparison to the iterative program.

Q5. Write a C++ program to print Fibonacci series using recursion.

Ans :

```
#include <iostream>
using namespace std;
int fibonnaci(int x) {
if((x==1)||(x==0)) {
return(x);
        }else {
        return(fibonnaci(x-1)+fibonnaci(x-2));
        }
}
int main() {
int x , i=0;
        cout <<"Enter the number of terms of
                  series : ";
        cin >> x;
        cout <<"\nFibonnaci Series : ";
while(i < x) {
        cout <<" "<< fibonnaci(i);
        i++;
        }
        return0;
}
```

Output

```
Enter the number of terms of series : 15
Fibonnaci Series : 0 1 1 2 3 5 8 13 21 34 55 89
144 233 377
```

Q6. Write a C++ program to find factorial of given number using recursion.

Ans :

```
#include <iostream>
using namespace std;
int fact(int n);
int main()
{
int n;
        cout <<"Enter a positive integer: ";
        cin >> n;
        cout <<"Factorial of "<< n <<" = "<<
        fact(n);
return0;
}
int fact(int n)
{
if(n >1)
return n * fact(n -1);
else
return1;
}
```

Output

```
Enter an positive integer: 6
Factorial of 6 = 720
```

Q7. Write a c++program to calculate power using recursion in c++.

Ans :

```
#include <iostream>
using namespace std;
int calculate(int, int);
int main()
{
int base, power, result;
     cout <<"Enter base number: ";
     cin >> base;
     cout <<"Enter power number(positive integer): ";
     cin >> power;
     result = calculate(base, power);
     cout << base <<" ^ "<< power <<" = "<< result;
return0;
}
int calculate(int base, int power)
{
if (power !=0)
return (base*calculate(base, power-1));
else
return1;
}
```

Output

```
Enter base number: 3
Enter power number (positive integer): 4
3 ^ 4 = 81
```

Q8. Write a c++ program to reverse a number using recursion.

Ans :

```
#include <iostream.h>
using namespace std;
int reverseNumber(int n) {
        static temp,sum;
        if(n>0){
        temp = n%10;
        sum=sum*10+ temp;
reverseNumber(n/10);
}
else
{
returnsum;
}
}
int main()
{
int n,reverse;
cout<<"Enter number";
cin >> n;
reverse = reverseNumber(n);
cout <<"Reverse of number is"<< reverse;
return0;
}
```

Output

```
Enter number : 3456
Reverse of number is : 6543
```

Q9. Write a C++ program to check whether the given number is a prime or not using recursion.

Ans :

```
#include <bits/stdc++.h>
using namespace std;
bool isprime(int n, int i =2)
{
        if (n <=2)
                return (n ==2) ? true : false;
        if (n % i ==0)
                return false;
        if (i * i > n)
                return true;
        return isprime(n, i +1);
}
int main()
{
        int n =18;
        if (isprime(n))
                cout <<"Yes, Number is Prime
                Number";
        else
                cout <<"No, Number is not Prime
                Number";
        return0;
}
```

Output

```
No, Number is not Prime Number
```

2.1.2 Recurrence

Q10. Define Recurrence.

Ans :

Recurrence means, the number of times, that a function has recused. It may also mean, simply, that something has been repeated. It may even be a variable.

A recurrence is a well-defined mathematical function where the function being defined is applied within its own definition. The mathematical function factorial of n can also be defined recursively as n! = n × (n - 1)!, where 1! = 1 Fibonacci sequence as an example. The Fibonacci sequence is the sequence of numbers 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ... The first two numbers of the sequence are both 1, whereas each succeeding number is the sum of the preceding two numbers (we arrived at 55 as the 10th number; it is the sum of 21 and 34, the eighth and ninth numbers).

2.1.3 Use of Stack in Recursion

Q11. Explain how recursion used to reversing the stack?

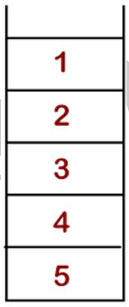
Ans :

There are various ways to reverse a stack using recursion. The most common way of reversing a stack is to use an auxiliary stack. First, we will pop all the elements from the stack and push them into the auxiliary stack. Once all the elements are pushed into the auxiliary stack, then it contains the elements in the reverse order and we simply print them. But, here, we will not use the auxiliary stack. We will use a recursion method to reverse a stack where recursion means calling the function itself again and again.

In the recursion method, we first pop all the elements from the input stack and push all the popped items into the function call stack until the stack becomes empty. When the stack becomes empty, all the items will be pushed at the stack. Let's understand this scenario through an example.

For example:

Input stack: 1, 2, 3, 4, 5

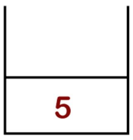


Output: 5, 4, 3, 2, 1

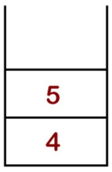
Solution

Firstly, all the elements from the input stack are pushed into the function call stack

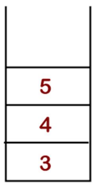
Step 1: Element 5 is pushed at the bottom of the stack shown as below:



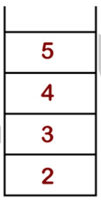
Step 2: Element 4 is pushed at the bottom of the stack shown as below:



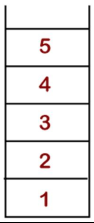
Step 3: Element 3 is pushed at the bottom of the stack shown as below:



Step 4: Element 2 is pushed at the bottom of the stack shown as below:



Step 5: Element 1 is pushed at the bottom of the stack shown as below:



Q12. Write an algorithm to sort a stack.

Ans :

1. Initialize a stack and push elements in it.
2. Create a function to insert the elements in sorted order in the stack which accepts a stack and an element as a parameter. Check if the stack is empty or the given element is greater than the element at the top of the stack, push the element in the stack and return.
3. Create a temporary variable and store the top of the stack in it. Pop the element at the top of the stack and make the recursive call to the function itself. Push the temporary variable in the stack.

4. Similarly, create a function sort() that accepts a stack as a parameter. Check if the stack is not empty, create a variable x, and store the top of the stack in it. Pop the element at the top of the stack. Make a recursive call to the function itself. Call the function to insert the elements in sorted order in the stack.

5. Call the sort function in the main().

6. Create a function to print the sorted stack which accepts a stack as the parameter. Traverse while the stack is not empty, print the data of the stack and move to the next element.

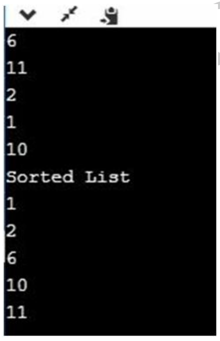
Q13. Write a C++ program to sort a stack using recursion.

Ans :

```
#include <iostream>
usingnamespacestd;
structNODE
{
        int Data;
        NODE *Next;
};
classStack
{
        NODE *Top;
                public:
                Stack ()
        {
        Top = NULL;
        }
voidPush (int x);
intPop ();
        voidDisp ();
        voidISort (Stack & A, int t);
        voidSort (Stack & A);
        };
        void Stack::Push (int x)
        {
                NODE *Temp;
                Temp = new NODE;
                Temp->Data = x;
                Temp->Next = Top;
                Top = Temp;
        }
        int Stack::Pop ()
        {
        if (Top != NULL)
        {
                NODE *Temp = Top;
                Top = Top->Next;
                int b = Temp->Data;
                delete Temp;
        return b;
        }
        else
        cout<<"Stack Empty..";
        }
        void Stack::Disp ()
        {
                NODE *Temp = Top;
                while (Temp != NULL)
        {
                cout<< Temp->Data <<endl;
                Temp = Temp->Next;
        }
}
void Stack::ISort (Stack & A, int t)
{
        NODE *Temp = Top;
                if (Top == NULL || Top->Data > t)
                A.Push (t);
                //If Stack empty,insert the element or t
                // is smaller than Top
        else
        {
        int x = A.Pop (); // Pop the Top element
        ISort (A, t);       // Sorting
        A.Push (x);        // Pushing all the popped
                           // elements
        }
}
void Stack :: Sort (Stack &A)
```

```
{
        if (Top != NULL)
        {
        int f= A.Pop();          // Popping Top
        Sort(A);                 // Sorting
        ISort(A, f);             // Inserting
    }
}
intmain ()
{
int a;
        Stack ST;
char Ch;
        ST.Push (10);
        ST.Push (1);
        ST.Push (2);
        ST.Push (11);
        ST.Disp ();
        cout<<"Sorted List\n";
        ST.Sort(ST);
        ST.Disp();
        return0;
        }
```

Output



```
6
11
2
1
10
Sorted List
1
2
6
10
11
```

2.1.4 Variants of Recursion , Recursive functions

Q14. Explain various types of recursions.

Ans : **(June-18)**

Recursion is the process in which a function calls itself up to n-number of times. If a program allows the user to call a function inside the same function recursively, the procedure is called a recursive call of the function. Furthermore, a recursive function can call itself directly or indirectly in the same program.

Syntax of the Recursion function

```
void recursion ()
{
recursion();
        // The recursive function calls itself inside the
        // same function
}
int main ()
{
recursion (); // function call
}
```

In the above syntax, the main() function calls the recursion function only once. After that, the recursion function calls itself up to the defined condition, and if the user doesn't define the condition, it calls the same function infinite times.

Recursive functions can be classified on the basis of :

a) If the functions call itself directly or indirectly – **Direct / Indirect**

b) If an operation is pending at each recursive call – **Tail Recursive/ Not**

c) based on the structure of the function calling pattern – **Linear / Tree**

(i) Direct Recursion

➢ If a function explicitly calls itself it is called directly recursive.

➢ When the method invokes itself it is direct.

```
int testfunc(int num) {
        if (num == 0)
                return 0;
        else
                return (testfunc(num - 1));
}
```

Here, the function 'testfunc' calls itself for all positive values of num.

(ii) Indirect Recursion

- This occurs when the function invokes other method which again causes the original function to be called again.
- If a method 'X' , calls method 'Y', which calls method 'Z' which again leads to 'X' being invoked is called indirect recursive or mutually recursive as well.

```
int testfunc1(int num) {
        if (num == 0)
        return 0;
        else
        return (testfunc2(num - 1));
}
        int testfunc2(int num2) {
        return testfunc1(num2 - 1);
}
```

(iii) Tail / Bottom Recursion

A function is said to be tail-recursive, if no operations are pending when the recursive function returns to its caller.

- Such functions, immediately return the return value from the calling function.
- It is an efficient method as compared to others, as the stack space required is less and even compute overhead will get reduced.
- Recollect the previously discussed example, factorial of a number. We had written it in non tail recursive way, as after call operation is still pending.

```
int fact(int n){
if (n == 1)
        return 1;
        else
        return (n * fact(n - 1));
}
```

In order to make it tail recursive, information about pending tasks has to be tracked.

```
int fact(int n){
return (n * fact2(n - 1));
        }
        int fact2(int n, int result) {
        if (n == 1) return result;
        return fact2(n - 1, n * result);
}
```

If you observe, the 'fact2' has a similar syntax to the original fact. Now, 'fact' in tail recursion case does not have pending calculations/operations to perform on return from recursive function calls.

The value computed by fact2 is simply returned. Thus, the amount of space required on stack reduces considerably. (just for value of n and result, space required.

(iv) Linear and Tree Recursion

Depending on the structure the recursive function calls take, or grows it can be either linear or non linear.

It is linearly recursive when, the pending opera-tions do not involve another recursive call to the function. Our Factorial recursive function is linearly recursive as it only involves multiplying the returned values and no further calls to function.

Tree recursion is when, pending operations involve another recursive call to function.

Q15. What is recursive function?

Ans :

Recursive Functions

Recursive Functions is the process of defining something in terms of itself. It is a function that calls itself again in the body of the function.

A function fact ( ), that computes the factorial of an integer 'N' ,which is the product of all whole numbers from 1 to N.

When fact ( ) is called with an argument of 1 (or) 0, the function returns 1. Otherwise, it returns the product of n*fact (n-1), this happens until 'n' equals 1.

```
Fact (5) = 5* fact (4)
         = 5*4*3* fact (3)
         = 5*4*3*2* fact (2)
         = 5*4*3*2*1 fact (1)
         = 5*4*3*2*1
         = 120.
```

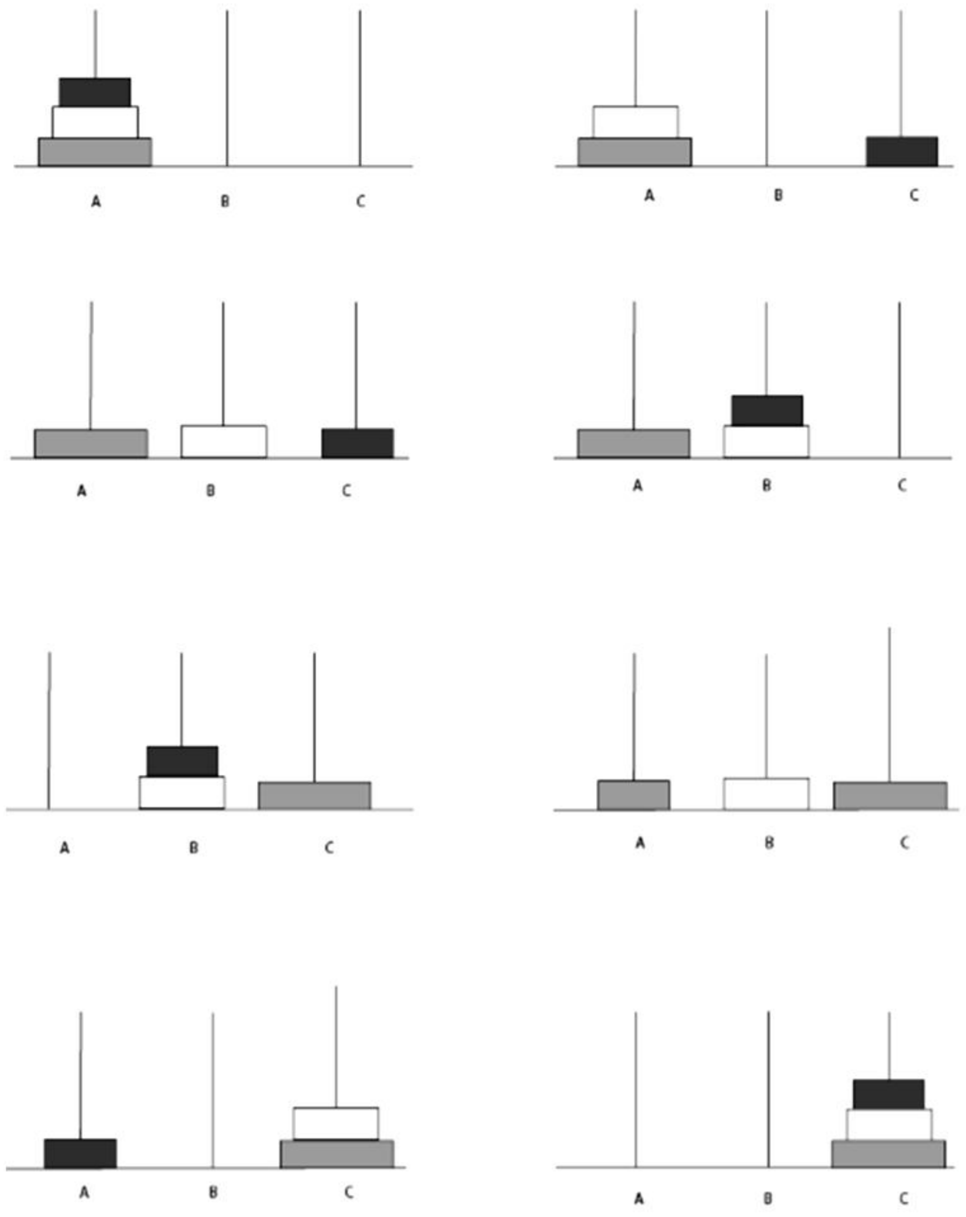
Q16. Explain Towers of Hanoi using recursion.

Ans : **(July-19, Imp.)**

Tower of Hanoi

1. It is a classic problem where you try to move all the disks from one peg to another peg using only three pegs.
2. Initially, all of the disks are stacked on top of each other with larger disks under the smaller disks.
3. You may move the disks to any of three pegs as you attempt to relocate all of the disks, but you cannot place the larger disks over smaller disks and only one disk can be transferred at a time.

Let us we have three disks stacked on a peg



Program

```
#include<iostream.h>
#include<conio.h>
        usingnamespace std;
        void TOH(int d, char t1, char t2, char t3)
        {
        if(d==1)
        {
        cout<<"\nShift top disk from tower" <<t1<<" to tower"<<t2;
        return;
        }
        TOH(d-1,t1,t3,t2);
        cout<<"\nShift top disk from tower" <<t1 <<" to tower"<<t2;
        TOH(d-1,t3,t2,t1);
        }
        int main()
        {
        int disk;
        cout<<"Enter the number of disks:"; cin>>disk;
        if(disk<1)
        cout<<"There are no disks to shift";
        else
        cout<<"There are "<<disk<<"disks in tower 1\n";
        TOH(disk, '1','2','3');
        cout<<"\n\n"<<disk<<"disks in tower 1 are shifted to tower 2";
        getch();
        return0;
        }
```

Output

```
        Enter the number of disks: 3
        There are 3 disks in tower 1
        Shift top disk from tower 1 to tower 2
        Shift top disk from tower 1 to tower 3
        Shift top disk from tower 2 to tower 3
        Shift top disk from tower 1 to tower 2
        Shift top disk from tower 3 to tower 1
        Shift too disk from tower 3 to tower 2
        Shift top disk from tower 1 to tower 2
        3 disks in tower 1 are shifted to tower 2
```

2.1.5 Iteration Versus Recrusion

Q17. What is Iteration?

Ans :

- Iteration uses repetition structure.
- An infinite loop occurs with iteration if the loop condition test never becomes false and Infinite looping uses CPU cycles repeatedly.
- An iteration terminates when the loop condition fails.
- An iteration does not use the stack so it's faster than recursion.
- Iteration consumes less memory.
- Iteration makes the code longer.

Q18. What are the differences between Iteration and recursion.

Ans :

On the basis	Recursion	Iteration
Basic	Recursion is the process of calling a function itself within its own code.	In iteration, there is a repeated execution of the set of instructions. In Iteration, loops are used to execute the set of instructions repetitively until the condition is false.
Syntax	There is a termination condition is specified.	The format of iteration includes initialization, condition, and increment/decrement of a variable.
Termination	The termination condition is defined within the recursive function.	Here, the termination condition is defined in the definition of the loop.
Code size	The code size in recursion is smaller than the code size in iteration.	The code size in iteration is larger than the code size in recursion.
Infinite	If the recursive function does not meet to a termination condition, it leads to an infinite recursion. There is a chance of system crash in infinite recursion.	Iteration will be infinite, if the control condition of the iteration statement never becomes false. On infinite loop, it repeatedly used CPU cycles.
Applied	It is always applied to functions.	It is applied to loops.
Speed	It is slower than iteration.	It is faster than recursion.
Usage	Recursion is generally used where there is no issue of time complexity, and code size requires being small.	It is used when we have to balance the time complexity against a large code size.
Time complexity	It has high time complexity.	The time complexity in iteration is relatively lower. We can calculate its time complexity by finding the no. of cycles being repeated in a loop.
Stack	It has to update and maintain the stack.	There is no utilization of stack.
Memory	It uses more memory as compared to iteration.	It uses less memory as compared to recursion.
Overhead	There is an extensive overhead due to updating and maintaining the stack.	There is no overhead in iteration.

2.2 QUEUES

2.2.1 Concept of Queues, Queue Abstract Data Type

Q19. What is mean by Queue? What are the basic operations of Queue?

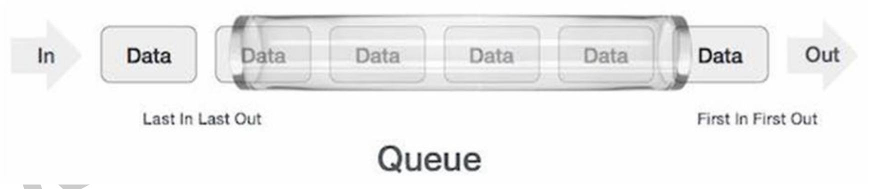
Ans : **(June-18, Imp.)**

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

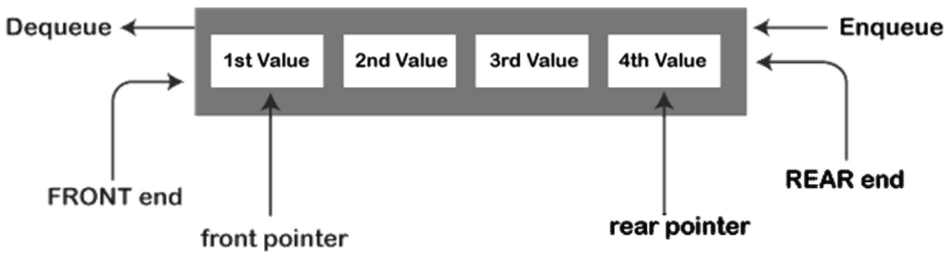
Queue Representation

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure:



As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.

A queue in the data structure can be considered similar to the queue in the real-world. A queue is a data structure in which whatever comes first will go out first. It follows the FIFO (First-In-First-Out) policy. In Queue, the insertion is done from one end known as the rear end or the tail of the queue, whereas the deletion is done from another end known as the front end or the head of the queue. In other words, it can be defined as a list or a collection with a constraint that the insertion can be performed at one end called as the rear end or tail of the queue and deletion is performed on another end called as the front end or the head of the queue.



Operations on Queue

There are two fundamental operations performed on a Queue:

- **Enqueue:** The enqueue operation is used to insert the element at the rear end of the queue. It returns void.
- **Dequeue:** The dequeue operation performs the deletion from the front-end of the queue. It also returns the element which has been removed from the front-end. It returns an integer value. The dequeue operation can also be designed to void.
- **Peek:** This is the third operation that returns the element, which is pointed by the front pointer in the queue but does not delete it.
- **Queue overflow (isfull):** When the Queue is completely full, then it shows the overflow condition.
- **Queue underflow (isempty):** When the Queue is empty, i.e., no elements are in the Queue then it throws the underflow condition.

Q20. Explain Enqueue and Dequeue operations.

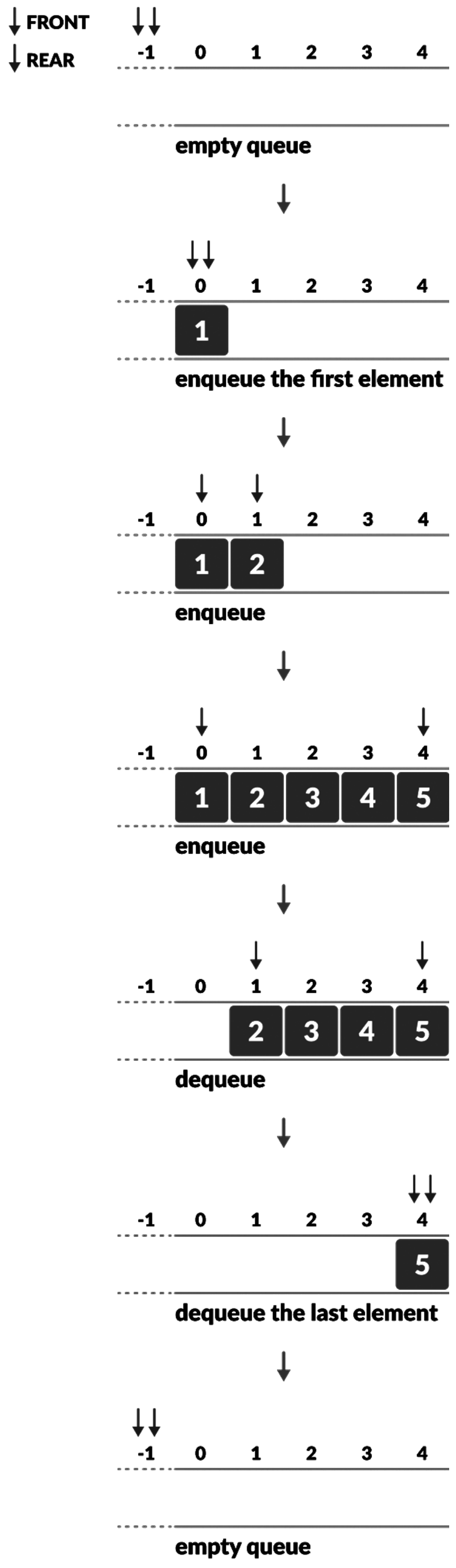
Ans :

Enqueue Operation

- Check if the queue is full
- For the first element, set the value of FRONT to 0
- Increase the REAR index by 1
- Add the new element in the position pointed to by REAR

Dequeue Operation

- Check if the queue is empty
- Return the value pointed by FRONT
- Increase the FRONT index by 1
- For the last element, reset the values of FRONT and REAR to -1.



Q21. Discuss Queue Abstract Data Type (ADT).

Ans : **(June-18)**

1. The queue abstract data type is defined by the following structure and operations.
2. A queue is structured, as described above, as an ordered collection of items which are added at one end, called the "rear," and removed from the other end, called the "front."
3. Queues maintain a FIFO ordering property. The queue operations are given below.
   - Queue() creates a new queue that is empty. It needs no parameters and returns an empty queue.
   - enqueue(item) adds a new item to the rear of the queue. It needs the item and returns nothing.
   - dequeue() removes the front item from the queue. It needs no parameters and returns the item. The queue is modified.
   - isEmpty() tests to see whether the queue is empty. It needs no parameters and returns a boolean value.
   - size() returns the number of items in the queue. It needs no parameters and returns an integer.

Q22. Write a C++ program to implement Queue ADT.

Ans : **(June-18, Dec.-17)**

```
// Queue implementation in C++
#include <iostream>
#define SIZE 5
using namespace std;
class Queue {
    private:
    int items[SIZE], front, rear;
    public:
    Queue() {
    front = -1;
    rear = -1;
    }
        bool isFull() {
        if (front == 0 && rear == SIZE - 1) {
        return true;
        }
        return false;
}
        bool isEmpty() {
        if (front == -1)
        return true;
        else
        return false;
}
        void enQueue(int element) {
        if (isFull()) {
        cout << "Queue is full";
        } else {
        if (front == -1) front = 0;
        rear++;
        items[rear] = element;
         cout << endl
        << "Inserted " << element << endl;
        }
  }
        int deQueue() {
        int element;
        if (isEmpty()) {
        cout << "Queue is empty" << endl;
        return (-1);
        } else {
        element = items[front];
        if (front >= rear) {
        front = -1;
        rear = -1;
        } /* Q has only one element, so we reset the
              queue after deleting it. */
        else {
        front++;
        }
        cout << endl
        << "Deleted -> " << element << endl;
        return (element);
        }
}
        void display() {
        /* Function to display elements of Queue */
        int i;
        if (isEmpty()) {
        cout << endl
        << "Empty Queue" << endl;
        } else {
        cout << endl
        << "Front index-> " << front;
        cout << endl
        << "Items -> ";
        for (i = front; i <= rear; i++)
        cout << items[i] << "  ";
        cout << endl
        << "Rear index-> " << rear << endl;
        }
    }
};
int main()
{
Queue q;
//deQueue is not possible on empty queue
q.deQueue();
        //enQueue 5 elements
        q.enQueue(1);
        q.enQueue(2);
        q.enQueue(3);
        q.enQueue(4);
        q.enQueue(5);
        // 6th element can't be added to because the
        //queue is full
        q.enQueue(6);
        q.display();
    // deQueue removes element entered first i.e. 1
        q.deQueue();
        //Now we have just 4 elements
        q.display();
        return 0;
        }
```

Q23. Discuss various types of Queues

Ans :

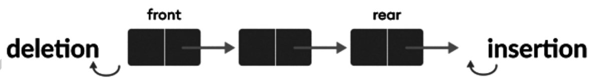
A queue is a useful data structure in programming. It is similar to the ticket queue outside a cinema hall, where the first person entering the queue is the first person who gets the ticket.

There are four different types of queues:

- Simple Queue
- Circular Queue
- Priority Queue
- Double Ended Queue

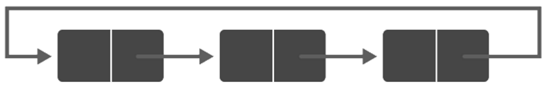
- **Simple Queue**

In a simple queue, insertion takes place at the rear and removal occurs at the front. It strictly follows the FIFO (First in First out) rule.



- **Circular Queue**

In a circular queue, the last element points to the first element making a circular link.

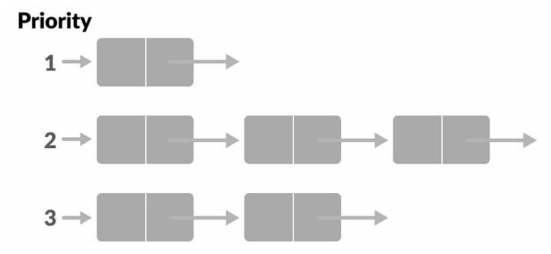


- **Circular Queue Representation**

The main advantage of a circular queue over a simple queue is better memory utilization. If the last position is full and the first position is empty, we can insert an element in the first position. This action is not possible in a simple queue.

- **Priority Queue**

A priority queue is a special type of queue in which each element is associated with a priority and is served according to its priority. If elements with the same priority occur, they are served according to their order in the queue.

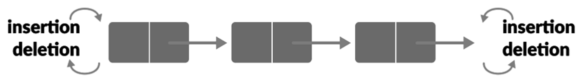


- **Priority Queue Representation**

  Insertion occurs based on the arrival of the values and removal occurs based on priority.

- **Deque (Double Ended Queue)**

  In a double ended queue, insertion and removal of elements can be performed from either from the front or rear. Thus, it does not follow the FIFO (First In First Out) rule.



- **Deque Representation**

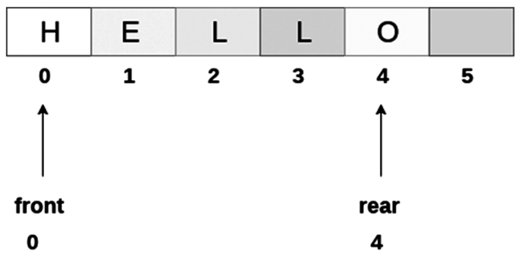
2.2.2 Realization of Queues using Arrays

Q24. Explain queue implementation using arrays.

Ans :

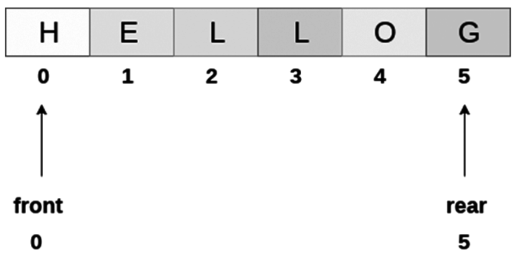
Array representation of Queue

We can easily represent queue by using linear arrays. There are two variables i.e. front and rear, that are implemented in the case of every queue. Front and rear variables point to the position from where insertions and deletions are performed in a queue. Initially, the value of front and queue is -1 which represents an empty queue. Array representation of a queue containing 5 elements along with the respective values of front and rear, is shown in the following figure.



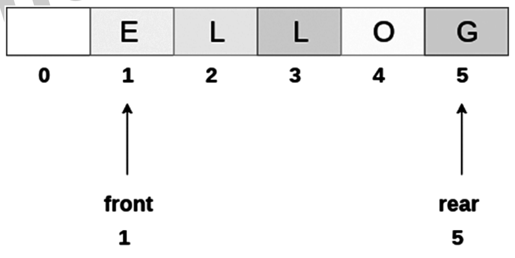
Queue

The above figure shows the queue of characters forming the English word **"HELLO"**. Since, No deletion is performed in the queue till now, therefore the value of front remains -1 . However, the value of rear increases by one every time an insertion is performed in the queue. After inserting an element into the queue shown in the above figure, the queue will look something like following. The value of rear will become 5 while the value of front remains same.



Queue after inserting an element

After deleting an element, the value of front will increase from -1 to 0. however, the queue will look something like following.



Queue after deleting an element

Q25. Explain Enqueue(), DeQueue(), Display() operations.

Ans :

Before we implement actual operations, first follow the below steps to create an empty queue.

- **Step 1:** Include all the header files which are used in the program and define a constant **'SIZE'** with specific value.
- **Step 2:** Declare all the user defined functions which are used in queue implementation.
- **Step 3:** Create a one dimensional array with above defined SIZE (int queue[SIZE]).

- **Step 4:** Define two integer variables 'front' and 'rear' and initialize both with '–1'. (int front = -1, rear = -1)
- **Step 5:** Then implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on queue.

enQueue(value) - Inserting value into the queue

In a queue data structure, enQueue() is a function used to insert a new element into the queue. In a queue, the new element is always inserted at **rear** position. The enQueue() function takes one integer value as a parameter and inserts that value into the queue. We can use the following steps to insert an element into the queue...

- **Step 1:** Check whether queue is FULL. (rear == SIZE-1)
- **Step 2:** If it is FULL, then display "Queue is FULL!!! Insertion is not possible!!!" and terminate the function.
- **Step 3:** If it is NOT FULL, then increment rear value by one (rear++) and set queue[rear] = value.

deQueue() - Deleting a value from the Queue

In a queue data structure, deQueue() is a function used to delete an element from the queue. In a queue, the element is always deleted from front position. The deQueue() function does not take any value as parameter. We can use the following steps to delete an element from the queue...

- **Step 1:** Check whether queue is EMPTY. (front == rear).
- **Step 2:** If it is EMPTY, then display "Queue is EMPTY!!! Deletion is not possible!!!" and terminate the function.
- **Step 3:** If it is NOT EMPTY, then increment the front value by one (front ++). Then display queue[front] as deleted element. Then check whether both front and rear are equal (front == rear), if it TRUE, then set both front and rear to '-1' (front = rear = -1).

display() - Displays the elements of a Queue

We can use the following steps to display the elements of a queue...

- **Step 1:** Check whether queue is EMPTY. (front == rear)
- **Step 2:** If it is EMPTY, then display "Queue is EMPTY!!!" and terminate the function.
- **Step 3:** If it is NOT EMPTY, then define an integer variable 'i' and set 'i = front+1'.
- **Step 4:** Display 'queue[i]' value and increment 'i' value by one (i++). Repeat the same until 'i' value reaches to rear (i <= rear).

Q26. Write an algorithm to insert an element to the queue and delete an element from the queue.

Ans :

Algorithm to insert any element in a queue

Check if the queue is already full by comparing rear to max - 1. if so, then return an overflow error.

If the item is to be inserted as the first element in the list, in that case set the value of front and rear to 0 and insert the element at the rear end.

Otherwise keep increasing the value of rear and insert each element one by one having rear as the index.

Algorithm

```
Step 1: IF REAR = MAX - 1
            Write OVERFLOW
            Go to step
        [END OF IF]
Step 2: IF FRONT = -1 and REAR = -1
        SET FRONT = REAR = 0
        ELSE
        SET REAR = REAR + 1
        [END OF IF]
Step 3: Set QUEUE[REAR] = NUM
Step 4: EXIT
```

Algorithm to delete an element from the queue

If, the value of front is -1 or value of front is greater than rear , write an underflow message and exit.

Otherwise, keep increasing the value of front and return the item stored at the front end of the queue at each time.

Algorithm

```
➢ Step 1: IF FRONT = -1 or FRONT > REAR
                Write UNDERFLOW
                ELSE
                SET VAL = QUEUE[FRONT]
                SET FRONT = FRONT + 1
                [END OF IF]
➢ Step 2: EXIT
```

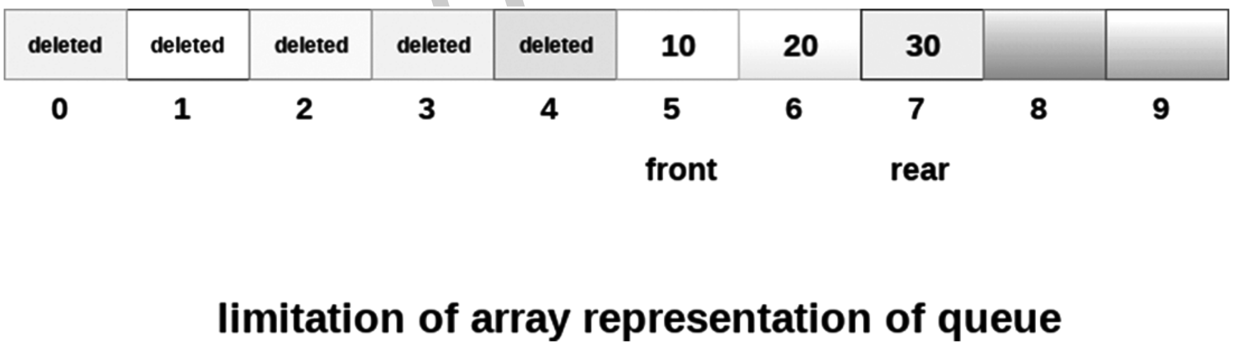
Q27. Discuss the major draw backs of Queue Array implementation.

Ans :

Some drawbacks of using this technique to implement a queue.

➢ **Memory wastage**

The space of the array, which is used to store queue elements, can never be reused to store the elements of that queue because the elements can only be inserted at front end and the value of front might be so high so that, all the space before that, can never be filled.



The above figure shows how the memory space is wasted in the array representation of queue. In the above figure, a queue of size 10 having 3 elements, is shown. The value of the front variable is 5, therefore, we can not reinsert the values in the place of already deleted element before the position of front. That much space of the array is wasted and can not be used in the future (for this queue).

➢ **Deciding the array size**

On of the most common problem with array implementation is the size of the array which requires to be declared in advance. Due to the fact that, the queue can be extended at runtime depending upon the problem, the extension in the array size is a time taking process and almost impossible to be performed at runtime since a lot of reallocations take place. Due to this reason, we can declare the array large enough so that we can store queue elements as enough as possible but the main problem with this declaration is that, most of the array slots (nearly half) can never be reused. It will again lead to memory wastage.

Q28. Write a C++ program to implement queues using arrays.

Ans :

```
#include<iostream>
usingnamespace std;
int queue[100], n =100, front =-1, rear =-1;
voidInsert(){
        int val;
        if(rear == n -1)
        cout<<"Queue Overflow"<<endl;
        else{
        if(front ==-1)
        front =0;
        cout<<"Insert the element in queue : "<<endl;
        cin>>val;
        rear++;
        queue[rear]= val;
    }
}
voidDelete(){
        if(front ==-1|| front > rear){
        cout<<"Queue Underflow ";
        return;
        }else{
        cout<<"Element deleted from queue is : "<< queue[front]<<endl;
        front++;;
    }
}
voidDisplay(){
        if(front ==-1)
        cout<<"Queue is empty"<<endl;
        else{
        cout<<"Queue elements are : ";
        for(int i = front; i <= rear; i++)
        cout<<queue[i]<<" ";
              cout<<endl;
    }
}
int main(){
        int ch;
        cout<<"1) Insert element to queue"<<endl;
        cout<<"2) Delete element from queue"<<endl;
        cout<<"3) Display all the elements of queue"<<endl;
        cout<<"4) Exit"<<endl;
        do{
        cout<<"Enter your choice : "<<endl;
        cin<<ch;
        switch(ch){
              case1:Insert();
              break;
              case2:Delete();
              break;
              case3:Display();
              break;
              case4: cout<<"Exit"<<endl;
              break;
              default: cout<<"Invalid choice"<<endl;
              }
        }while(ch!=4);
    return0;
}
```

The output of the above program is as follows

```
1)    Insert element to queue
2)    Delete element from queue
3)    Display all the elements of queue
4)    Exit
      Enter your choice : 1
      Insert the element in queue : 4
      Enter your choice : 1
      Insert the element in queue : 3
      Enter your choice : 1
      Insert the element in queue : 5
      Enter your choice : 2
      Element deleted from queue is : 4
      Enter your choice : 3
```

```
Queue elements are : 3 5
Enter your choice : 7
Invalid choice
Enter your choice : 4
Exit
```

Q29. Explain the representation of Queue using Linked List.

Ans : **(July-19)**

Implementing Queues using Linked Lists or Pointers is an effective way of the utilization of memory.

The queue is a Linear Data Structure like Arrays and Stacks.

It is a homogenous (similar) collection of elements in which new elements are inserted at one end Called the Rear end, and the existing elements are deleted from the other end called the Front end.

A Queue is logically a First in First Out (FIFO) type of data structure Or Queue simply means a line and follows FIFO logical design, which means the element inserted first will be deleted first.

Implementing Queues using Pointers is an effective way of the utilization of memory.

Since it is dynamic therefore you don't need to worry about the size of the queue It will grow and shrink at the run time.

In Arrays, if you want to add or delete elements in the middle then you have to shift all the existing elements.

But in Linked Lists, You only have to insert a new node and adjust the links of the newly inserted node with the rest of the nodes.

Algorithm for Insertion

```
Queue_Linked Ptr, temp
➢ Temp=start
➢ Ptr=new node
➢ Ptr->data=NULL
➢ If Rear == NULL
     Set front=Ptr
     Set rear=Ptr
➢ Else
➢ While(temp->next!=NULL)
➢ Temp=temp->next
➢ Temp->next=ptr
```

Algorithm for Deletion

```
➢ If( Front == NULL )
     Write Queue empty & return
➢ Else,
          Temp=start
          Value=temp->data
          Start=start->next
          Delete temp
          Return (value)
➢ Eixt
```

Advantages of Queue using linked lists

Memory efficient

Linked representation of queues is the effective and proper utilization of memory.

There is no memory wasted in linked lists because the data elements are stored at distant places in the memory whose addresses are linked by pointers.

Time efficient

There is no need to worry about deleting the element in the middle because it requires same amount of time to Insert at front and middle.

It is a very useful data structure that is used in various aspects of programming fields and in the software development process.

Example

```
#include<iostream>
usingnamespace std;
struct node {
      int data;
      struct node *next;
};
      struct node* front = NULL;
      struct node* rear = NULL;
      struct node* temp;
      voidInsert(){
      int val;
```

```
        cout<<"Insert the element in queue : "<<endl;
        cin>>val;
        if(rear == NULL){
        rear =(struct node *)malloc(sizeof(struct node));
        rear->next= NULL;
        rear->data = val;
        front = rear;
                }else{
        temp=(struct node *)malloc(sizeof(struct node));
        rear->next= temp;
        temp->data = val;
        temp->next= NULL;
        rear = temp;
    }
}
voidDelete(){
    temp = front;
    if(front == NULL){
        cout<<"Underflow"<<endl;
        return;
    }
    else
    if(temp->next!= NULL){
        temp = temp->next;
        cout<<"Element deleted from queue is : "<<front->data<<endl;
        free(front);
        front = temp;
        }else{
        cout<<"Element deleted from queue is : "<<front->data<<endl;
        free(front);
        front = NULL;
        rear = NULL;
    }
}
voidDisplay(){
        temp = front;
        if((front == NULL)&&(rear == NULL)){
        cout<<"Queue is empty"<<endl;
        return;
        }
        cout<<"Queue elements are: ";
        while(temp != NULL){
        cout<<temp->data<<" ";
        temp = temp->next;
    }
    cout<<endl;
}
int main(){
    int ch;
        cout<<"1) Insert element to queue"<<endl;
        cout<<"2) Delete element from queue"<<endl;
        cout<<"3) Display all the elements of queue"<<endl;
        cout<<"4) Exit"<<endl;
    do{
        cout<<"Enter your choice : "<<endl;
        cin>>ch;
        switch(ch){
            case1:Insert();
            break;
            case2:Delete();
            break;
            case3:Display();
            break;
            case4: cout<<"Exit"<<endl;
            break;
            default: cout<<"Invalid choice"<<endl;
        }
        }while(ch!=4);
    return0;
}
```

Output

The output of the above program is as follows

```
1)  Insert element to queue
2)  Delete element from queue
3)  Display all the elements of queue
4)  Exit
```

```
Enter your choice : 1
Insert the element in queue : 4
Enter your choice : 1
Insert the element in queue : 3
Enter your choice : 1
Insert the element in queue : 5
Enter your choice : 2
Element deleted from queue is : 4
Enter your choice : 3
Queue elements are : 3 5
Enter your choice : 7
Invalid choice
Enter your choice : 4
Exit
```

2.2.3 Circular Queue, Multi Queues

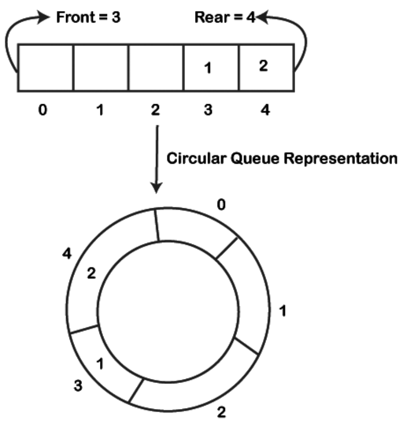
Q30. What is mean by Circular Queue? What is the need of circular queue?

Ans : **(July-21, Dec.-19)**

A circular queue is similar to a linear queue as it is also based on the FIFO (First In First Out) principle except that the last position is connected to the first position in a circular queue that forms a circle. It is also known as a Ring Buffer.

There was one limitation in the array implementation of Queue.

If the rear reaches to the end position of the Queue then there might be possibility that some vacant spaces are left in the beginning which cannot be utilized. So, to overcome such limitations, the concept of the circular queue was introduced.



As we can see in the above image, the rear is at the last position of the Queue and front is pointing somewhere rather than the 0th position. In the above array, there are only two elements and other three positions are empty. The rear is at the last position of the Queue; if we try to insert the element then it will show that there are no empty spaces in the Queue. There is one solution to avoid such wastage of memory space by shifting both the elements at the left and adjust the front and rear end accordingly. It is not a practically good approach because shifting all the elements will consume lots of time. The efficient approach to avoid the wastage of the memory is to use the circular queue data structure.

Q31. Discuss the operations and applications of circular queue.

Ans : **(July-21)**

Operations on Circular Queue

The following are the operations that can be performed on a circular queue:

- **Front:** It is used to get the front element from the Queue.
- **Rear:** It is used to get the rear element from the Queue.
- **enQueue(value):** This function is used to insert the new value in the Queue. The new element is always inserted from the rear end.
- **deQueue():** This function deletes an element from the Queue. The deletion in a Queue always takes place from the front end.

Applications of Circular Queue

The circular Queue can be used in the following scenarios:

Memory management

The circular queue provides memory management. As we have already seen that in linear queue, the memory is not managed very efficiently. But in case of a circular queue, the memory is managed efficiently by placing the elements in a location which is unused.

CPU Scheduling

The operating system also uses the circular queue to insert the processes and then execute them.

Traffic system

In a computer-control traffic system, traffic light is one of the best examples of the circular queue. Each light of traffic light gets ON one by one after every jinterval of time. Like red light gets ON for one minute then yellow light for one minute and then green light. After green light, the red light gets ON.

Q32. Explain in detail Enqueue and Dequeue operations with the context of circular queue.

Ans : **(July-21)**

Enqueue operation

The steps of enqueue operation are given below:

- First, we will check whether the Queue is full or not.
- Initially the front and rear are set to -1. When we insert the first element in a Queue, front and rear both are set to 0.
- When we insert a new element, the rear gets incremented, *i.e.*, rear = rear + 1.

Scenarios for inserting an element

There are two scenarios in which queue is not full:

- If rear != max - 1, then rear will be incremented to mod(maxsize) and the new value will be inserted at the rear end of the queue.
- If front != 0 and rear = max - 1, it means that queue is not full, then set the value of rear to 0 and insert the new element there.

There are two cases in which the element cannot be inserted:

- When front ==0 && rear = max-1, which means that front is at the first position of the Queue and rear is at the last position of the Queue.
- front== rear + 1;

Algorithm to insert an element in a circular queue

```
Step1:  IF (REAR+1)%MAX = FRONT
            Write " OVERFLOW "
            Goto step 4
        [End OF IF]
Step 2: IF FRONT = -1 and REAR = -1
            SET FRONT = REAR = 0
        ELSE IF REAR = MAX - 1 and FRONT ! = 0
            SET REAR = 0
        ELSE
            SET REAR = (REAR + 1) % MAX
        [END OF IF]
Step 3: SET QUEUE[REAR] = VAL
Step 4: EXIT
```

Dequeue Operation

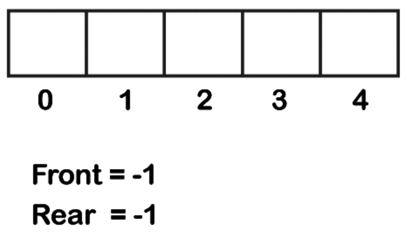
The steps of dequeue operation are given below:

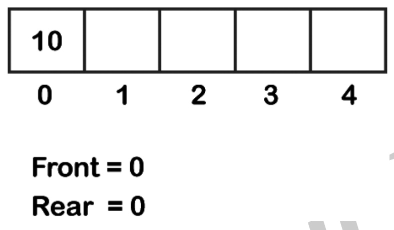
- First, we check whether the Queue is empty or not. If the queue is empty, we cannot perform the dequeue operation.
- When the element is deleted, the value of front gets decremented by 1.
- If there is only one element left which is to be deleted, then the front and rear are reset to -1.

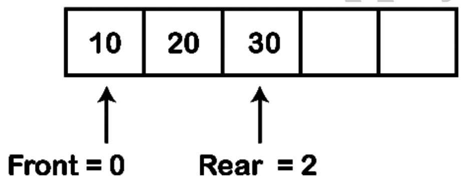
Algorithm to delete an element from the circular queue

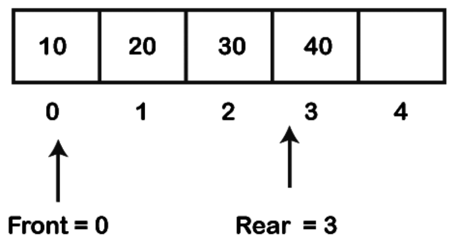
```
Step 1: IF FRONT = -1
            Write "UNDERFLOW"
            Goto Step 4
        [END of IF]
Step 2: SET VAL = QUEUE[FRONT]
Step 3: IF FRONT = REAR
            SET FRONT = REAR = -1
        ELSE
        IF FRONT = MAX -1
        SET FRONT = 0
        ELSE
        SET FRONT = FRONT + 1
        [END of IF]
        [END OF IF]
Step 4: EXIT
```

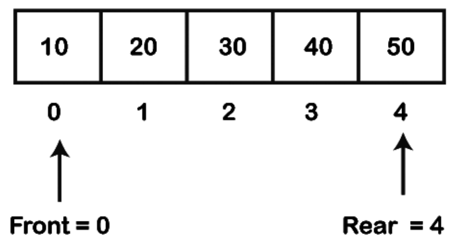
Let's understand the enqueue and dequeue operation through the diagrammatic representation.

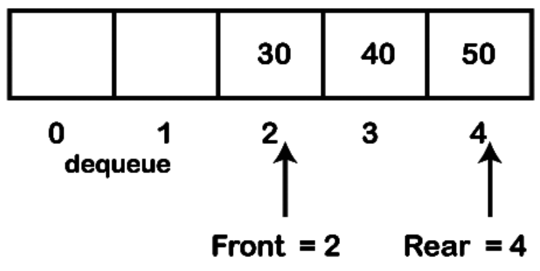


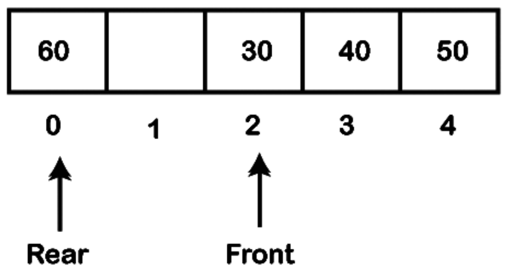


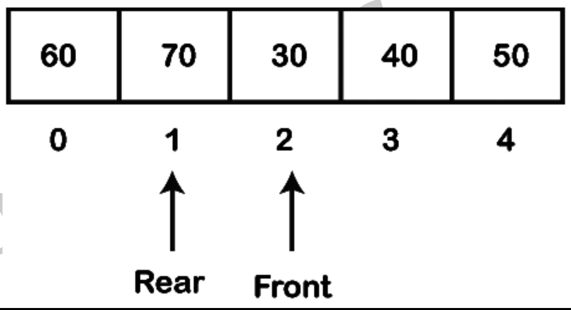












Q33. Write a C++ program to implement Circular Queue .

Ans :

```
#include<iostream>
usingnamespace std;
int cqueue[5];
int front =-1, rear =-1, n=5;
void insertCQ(int val){
        if((front ==0&& rear == n-1)||(front ==
        rear+1)){
        cout<<"Queue Overflow \n";
        return;
        }
    if(front ==-1){
        front =0;
        rear =0;
```

```
   }else{
     if(rear == n -1)
     rear =0;
     else
     rear = rear +1;
   }
   cqueue[rear]= val ;
}
void deleteCQ(){
   if(front ==-1){
     cout<<"Queue Underflow\n";
     return;
   }
   cout<<"Element deleted from queue is :
     "<<cqueue[front]<<endl;
   if(front == rear){
     front =-1;
     rear =-1;
   }else{cout<<cqueue[f]<<" ";
       f++;
     }
     }else{
     while(f <= n -1){
         cout<<cqueue[f]<<" ";
         f++;
     }
     f =0;
     while(f <= r){
         cout<<cqueue[f]<<" ";
         f++;zzzint main(){
   int ch, val;
   cout<<"1)Insert\n";
   cout<<"2)Delete\n";
   cout<<"3)Display\n";
   cout<<"4)Exit\n";
   do{
     cout<<"Enter choice : "<<endl;
     cin>>ch;
     switch(ch){
       case1:
       cout<<"Input for insertion: "<<endl;
       cin>>val;
       insertCQ(val);
       break;
       case2:
       deleteCQ();
       break;
       case3:
       displayCQ();
       break;
       case4:
       cout<<"Exit\n";
       break;
       default: cout<<"Incorrect!\n";
     }
   }while(ch !=4);
   return0;
}
```

Output

The output of the above program is as follows

```
1)   Insert
2)   Delete
3)   Display
4)   Exit
Enter choice : 1
Input for insertion:
Enter choice : 1
Input for insertion:
Enter choice : 1
```

Input for insertion:

Enter choice : 1

Input for insertion:

Enter choice : 1

Input for insertion:

Enter choice : 2

Element deleted from queue is : 5

Enter choice : 2

Element deleted from queue is : 3

Enter choice : 2

Element deleted from queue is : 2

Enter choice : 1

Input for insertion: 6

Enter choice : 3

Queue elements are :

7 9 6

Enter choice : 4

Exit

2.2.4 Dequeue, Priority Queue

Q34. Explain in detail about Dequeue.

Ans : **(Dec.-19)**

The dequeue stands for Double Ended Queue. In the queue, the insertion takes place from one end while the deletion takes place from another end. The end at which the insertion occurs is known as the rear end whereas the end at which the deletion occurs is known as front end.

Deque is a linear data structure in which the insertion and deletion operations are performed from both ends. We can say that deque is a generalized version of the queue.

Properties of deque

Deque can be used both as **stack** and **queue** as it allows the insertion and deletion operations on both ends.

In deque, the insertion and deletion operation can be performed from one side. The stack follows the LIFO rule in which both the insertion and deletion can be performed only from one end; therefore, we conclude that dequeue can be considered as a stack.

In deque, the insertion can be performed on one end, and the deletion can be done on another end. The queue follows the FIFO rule in which the element is inserted on one end and deleted from another end. Therefore, we conclude that the deque can also be considered as the queue.

There are two types of Queues, Input-restricted queue, and output-restricted queue.

1. **Input-restricted queue:** The input-restricted queue means that some restrictions are applied to the insertion. In input-restricted queue, the insertion is applied to one end while the deletion is applied from both the ends.
2. **Output-restricted queue:** The output-restricted queue means that some restrictions are applied to the deletion operation. In an output-restricted queue, the deletion can be applied only from one end, whereas the insertion is possible from both ends.

Q35. List out the operations and apllications of Dequeue.

Ans :

The following are the operations applied on deque:

- ➢ Insert at front
- ➢ Delete from end
- ➢ insert at rear
- ➢ delete from rear

Other than insertion and deletion, we can also perform peek operation in deque. Through peek operation, we can get the front and the rear element of the dequeue.

We can perform two more operations on dequeue:

- ➢ **isFull():** This function returns a true value if the stack is full; otherwise, it returns a false value.
- ➢ **isEmpty():** This function returns a true value if the stack is empty; otherwise it returns a false value.

Applications of Deque

- The deque can be used as a stack and queue; therefore, it can perform both redo and undo operations.
- It can be used as a palindrome checker means that if we read the string from both ends, then the string would be the same.
- It can be used for multiprocessor scheduling. Suppose we have two processors, and each processor has one process to execute. Each processor is assigned with a process or a job, and each process contains multiple threads. Each processor maintains a deque that contains threads that are ready to execute. The processor executes a process, and if a process creates a child process then that process will be inserted at the front of the deque of the parent process. Suppose the processor P_2 has completed the execution of all its threads then it steals the thread from the rear end of the processor P_1 and adds to the front end of the processor P_2. The processor P_2 will take the thread from the front end; therefore, the deletion takes from both the ends, i.e., front and rear end. This is known as the A-steal algorithm for scheduling.

Q36. Explain Enqueue and Dequeue operations in detail.

Ans :

The following are the steps to perform the operations on the Deque:

Enqueue operation

1. Initially, we are considering that the deque is empty, so both front and rear are set to -1, i.e., f = -1 and r = -1.
2. As the deque is empty, so inserting an element either from the front or rear end would be the same thing. Suppose we have inserted element 1, then front is equal to 0, and the rear is also equal to 0.
3. Suppose we want to insert the next element from the rear. To insert the element from the rear end, we first need to increment the rear, i.e., rear=rear+1. Now, the rear is pointing to the second element, and the front is pointing to the first element.
4. Suppose we are again inserting the element from the rear end. To insert the element, we will first increment the rear, and now rear points to the third element.
5. If we want to insert the element from the front end, and insert an element from the front, we have to decrement the value of front by 1. If we decrement the front by 1, then the front points to -1 location, which is not any valid location in an array. So, we set the front as (n -1), which is equal to 4 as n is 5. Once the front is set, we will insert the value as shown in the below figure:

Dequeue Operation

1. If the front is pointing to the last element of the array, and we want to perform the delete operation from the front. To delete any element from the front, we need to set front=front+1. Currently, the value of the front is equal to 4, and if we increment the value of front, it becomes 5 which is not a valid index. Therefore, we conclude that if front points to the last element, then front is set to 0 in case of delete operation.
2. If we want to delete the element from rear end then we need to decrement the rear value by 1, i.e., rear=rear-1 as shown in the below figure:
3. If the rear is pointing to the first element, and we want to delete the element from the rear end then we have to set rear=n-1 where n is the size of the array as shown in the below figure:

Q37. Write a C++ program for Deque implementation.

Ans :

```
#include<iostream>
usingnamespace std;
#define SIZE 10
class dequeue {
        int a[20],f,r;
        public:
```

```
        dequeue();
        void insert_at_beg(int);
        void insert_at_end(int);
        void delete_fr_front();
        void delete_fr_rear();
        void show();
};
dequeue::dequeue(){
        f=-1;
        r=-1;
}
void dequeue::insert_at_end(int i){
        if(r>=SIZE-1){
        cout<<"\n insertion is not possible, overflow!!!!";
        }else{
        if(f==-1){
              f++;
              r++;
        }else{
              r=r+1;
        }
        a[r]=i;
        cout<<"\nInserted item is"<<a[r];
    }
}
void dequeue::insert_at_beg(int i){
        if(f==-1){
        f=0;
        a[++r]=i;
        cout<<"\n inserted element is:"<<i;
        }elseif(f!=0){
        a[—f]=i;
        cout<<"\n inserted element is:"<<i;
        }else{
        cout<<"\n insertion is not possible, overflow!!!";
    }
}
        void dequeue::delete_fr_front(){
        if(f==-1){
        cout<<"deletion is not possible::dequeue is empty";
        return;
    }
    else{
        cout<<"the deleted element is:"<<a[f];
        if(f==r){
            f=r=-1;
            return;
        }else
            f=f+1;
        }
    }
    void dequeue::delete_fr_rear(){
        if(f==-1){
        cout<<"deletion is not possible::dequeue is empty";
        return;
        }
        else{
            cout<<"the deleted element is:"<<a[r];
            if(f==r){
                f=r=-1;
            }else
                r=r-1;
        }
    }
    void dequeue::show(){
        if(f==-1){
            cout<<"Dequeue is empty";
        }else{
            for(int i=f;i<=r;i++){
                cout<<a[i]<<" ";
            }
        }
    }
    int main(){
        int c,i;
        dequeue d;
        Do//perform switch opeartion {
        cout<<"\n 1.insert at beginning";
        cout<<"\n 2.insert at end";
        cout<<"\n 3.show";
        cout<<"\n 4.deletion from front";
        cout<<"\n 5.deletion from rear";
        cout<<"\n 6.exit";
        cout<<"\n enter your choice:";
        cin>>c;
        switch(c){
            case1:
        cout<<"enter the element to be inserted";
                cin>>i;
                d.insert_at_beg(i);
            break;
            case2:
        cout<<"enter the element to be inserted";
        cin>>i;
        d.insert_at_end(i);
            break;
            case3:
                d.show();
            break;
            case4:
                d.delete_fr_front();
            break;
            case5:
                d.delete_fr_rear();
            break;
            case6:
                exit(1);
            break;
            default:
                cout<<"invalid choice";
            break;
        }
    }while(c!=7);
}
```

Output

```
1.    insert at beginning
2.    insert at end
3.    show
4.    deletion from front
5.    deletion from rear
6.    exit
enter your choice: 1
deletion is not possib
1.    nsert at end
3.    show
4.    deletion from front
5.    deletion from rear
6.    exit
enter your choice:5
deletion is not possible::dequeue is empty
1.    insert at beginning
2.    insert at end
3.    show
4.    deletion from front
5.    deletion from rear
6.    exit
```

```
enter your choice:1
enter the element to be inserted7
inserted element is:7
1.    insert at beginning
2.    insert at end
3.    show
4.    deletion from front
5.    deletion from rear
6.    exit
enter your choice:1
enter the element to be inserted6
insertion is not possible, overflow!!!
1.    insert at beginning
2.    insert at end
3.    show
4.    deletion from front
5.    deletion from rear
6.    exit
enter your choice:1
enter the element to be inserted4
insertion is not possible, overflow!!!
1.    insert at beginning
2.    insert at end
3.    show
4.    deletion from front
5.    deletion from rear
6.    exit
enter your choice:2
enter the element to be inserted6
Inserted item is6
1.    insert at beginning
2.    insert at end
3.    show
4.    deletion from front
5.    deletion from rear
6.    exit
enter your choice:2
enter the element to be inserted4
Inserted item is4
1.    insert at beginning
2.    insert at end
3.    show
4.    deletion from front
5.    deletion from rear
6.    exit
enter your choice:3
7 6 4
1.    insert at beginning
2.    insert at end
3.    show
4.    deletion from front
5.    deletion from rear
6.    exit
enter your choice:4
the deleted element is:7
1.    insert at beginning
2.    insert at end
3.    show
4.    deletion from front
5.    deletion from rear
6.    exit
enter your choice:5
the deleted element is:4
1.    insert at beginning
2.    insert at end
3.    show
4.    deletion from front
5.    deletion from rear
6.    exit
enter your choice:1
enter the element to be inserted7
inserted element is:7
1.    insert at beginning
2.    insert at end
```

```
3.    show
4.    deletion from front
5.    deletion from rear
6.    exit
enter your choice:3
7 6
1.    insert at beginning
2.    insert at end
3.    show
4.    deletion from front
5.    deletion from rear
6.    exit
enter your choice:6
```

Q38. What is a Priority Queue? What are the characteristics of priority queue.

Ans :

A priority queue is an abstract data type that behaves similarly to the normal queue except that each element has some priority, i.e., the element with the highest priority would come first in a priority queue. The priority of the elements in a priority queue will determine the order in which elements are removed from the priority queue.

The priority queue supports only comparable elements, which means that the elements are either arranged in an ascending or descending order.

For example, suppose we have some values like 1, 3, 4, 8, 14, 22 inserted in a priority queue with an ordering imposed on the values is from least to the greatest. Therefore, the 1 number would be having the highest priority while 22 will be having the lowest priority.

Characteristics of a Priority queue

- ➢ Every element in a priority queue has some priority associated with it.
- ➢ An element with the higher priority will be deleted before the deletion of the lesser priority.
- ➢ If two elements in a priority queue have the same priority, they will be arranged using the FIFO principle.

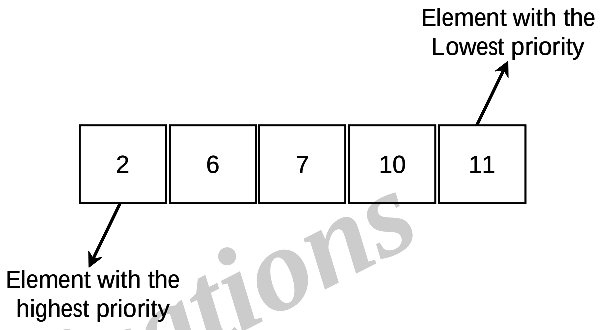
Q39. Discuss various types of priority queues.

Ans :

There are two types of priority queue:

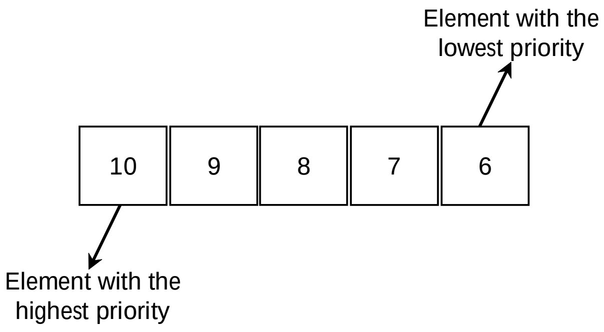
Ascending order priority queue

In ascending order priority queue, a lower priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in an ascending order like 1,2,3,4,5; therefore, the smallest number, i.e., 1 is given as the highest priority in a priority queue.


Element with the Lowest priority
2
6
7
10
11
Element with the highest priority

Descending order priority queue

In descending order priority queue, a higher priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in descending order like 5, 4, 3, 2, 1; therefore, the largest number, i.e., 5 is given as the highest priority in a priority queue.

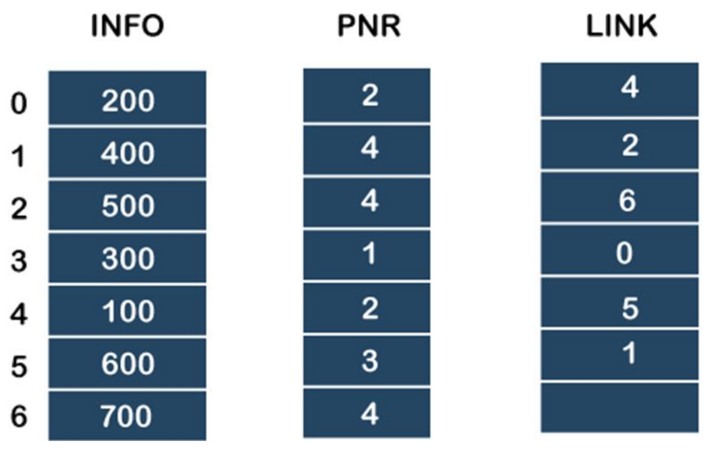

Element with the lowest priority
10
9
8
7
6
Element with the highest priority

Representation of priority queue

Now, we will see how to represent the priority queue through a one-way list.

We will create the priority queue by using the list given below in which INFO list contains the data elements, PRN list contains the priority numbers of each data element available in the INFO list, and LINK basically contains the address of the next node.

In the case of priority queue, lower priority number is considered the higher priority, i.e., lower priority number = higher priority.



Step 1

In the list, lower priority number is 1, whose data value is 333, so it will be inserted in the list as shown in the below diagram:

Step 2

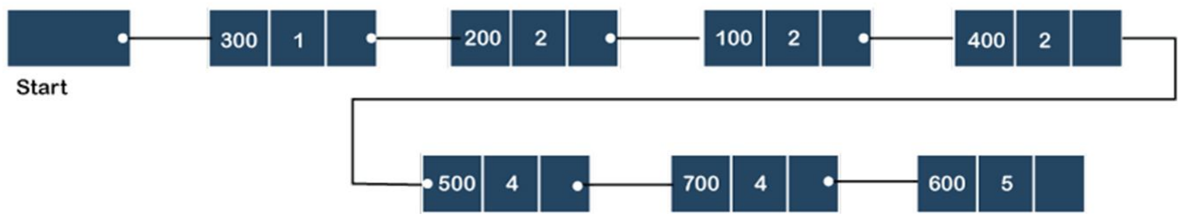
After inserting 333, priority number 2 is having a higher priority, and data values associated with this priority are 222 and 111. So, this data will be inserted based on the FIFO principle; therefore 222 will be added first and then 111.

Step 3

After inserting the elements of priority 2, the next higher priority number is 4 and data elements associated with 4 priority numbers are 444, 555, 777. In this case, elements would be inserted based on the FIFO principle; therefore, 444 will be added first, then 555, and then 777.

Step 4

After inserting the elements of priority 4, the next higher priority number is 5, and the value associated with priority 5 is 666, so it will be inserted at the end of the queue.



The following are the applications of the priority queue:

- It is used in the Dijkstra's shortest path algorithm.
- It is used in prim's algorithm
- It is used in data compression techniques like Huffman code.
- It is used in heap sort.
- It is also used in operating system like priority scheduling, load balancing and interrupt handling.

Q40. Write a c++ program to implement priority Queue.

Ans :

```
#include<iostream>
#include<queue>
usingnamespace std;
     void dequeElements(priority_queue <int> que){
           priority_queue <int> q = que;
                 while(!q.empty()){
                 cout << q.top()<<" ";
                 q.pop();
           }
           cout << endl;
           }
           int main(){
           priority_queue <int> que;
           que.push(10);
           que.push(20);
           que.push(30);
           que.push(5);
           que.push(1);
           cout <<"Currently que is holding : ";
           dequeElements(que);
           cout <<"Size of queue : "<<que.size()<< endl;
           cout <<"Element at top position : "<< que.top()<< endl;
           cout <<"Delete from queue : ";
           que.pop();
           dequeElements(que);
           cout <<"Delete from queue : ";
           que.pop();
     dequeElements(que);
}
```

Output

```
Currently que is holding : 30 20 10 5 1
Size of queue : 5
Element at top position : 30
Delete from queue : 20 10 5 1
Delete from queue : 10 5 1
```

2.2.5 Applications of Queues

Q41. List out the applications of Queues.

Ans :

Applications of Queue

Due to the fact that queue performs actions on first in first out basis which is quite fair for the ordering of actions. There are various applications of queues discussed as below.

1. Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
2. Queues are used in asynchronous transfer of data (where data is not being transferred at the same rate between two processes) for eg. pipes, file IO, sockets.
3. Queues are used as buffers in most of the applications like MP3 media player, CD player, etc.
4. Queue are used to maintain the play list in media players in order to add and remove the songs from the play-list.
5. Queues are used in operating systems for handling interrupts.

Q42. What are the differences between stack and Queue.

Ans :

Basis for comparison	Stack	Queue
Principle	It follows the principle LIFO (Last In-First Out), which implies that the element which is inserted last would be the first one to be deleted.	It follows the principle FIFO (First In - First Out), which implies that the element which is added first would be the first element to be removed from the list.
Structure	It has only one end from which both the insertion and deletion take place, and that end is known as a top.	It has two ends, i.e., front and rear end. The front end is used for the deletion while the rear end is used for the insertion.
Number of pointers used	It contains only one pointer known as a top pointer. The top pointer holds the address of the last inserted or the topmost element of the stack.	It contains two pointers front and rear pointer. The front pointer holds the address of the first element, whereas the rear pointer holds the address of the last element in a queue.
Operations performed	It performs two operations, push and pop. The push operation inserts the element in a list while the pop operation removes the element from the list.	It performs mainly two operations, enqueue and dequeue. The enqueue operation performs the insertion of the elements in a queue while the dequeue operation performs the deletion of the elements from the queue.
Examination of the empty condition	If top==-1, which means that the stack is empty.	If front== -1 or front = rear +1, which means that the queue is empty.

Examination of full condition	If top= = max-1, this condition implies that the stack is full.	If rear= =max-1, this condition implies that the stack is full.
Variants	It does not have any types.	It is of three types like priority queue, circular queue and double ended queue.
Implementation	It has a simpler implementation.	It has a comparatively complex implementation than a stack.
Visualization	A Stack is visualized as a vertical collection.	A Queue is visualized as a horizontal collection.

Q43. What are the differences between linear queue and circular queue.

Ans :

Basis of comparison	Linear Queue	Circular Queue
Meaning	The linear queue is a type of linear data structure that contains the elements in a sequential manner.	The circular queue is also a linear data structure in which the last element of the Queue is connected to the first element, thus creating a circle.
Insertion and Deletion	In linear queue, insertion is done from the rear end, and deletion is done from the front end.	In circular queue, the insertion and deletion can take place from any end.
Memory space	The memory space occupied by the linear queue is more than the circular queue.	It requires less memory as compared to linear queue.
Memory utilization	The usage of memory is inefficient.	The memory can be more efficiently utilized.
Order of execution	It follows the FIFO principle in order to perform the tasks.	It has no specific order for execution.

2.3 Linked List

2.3.1 Introduction to Linked List

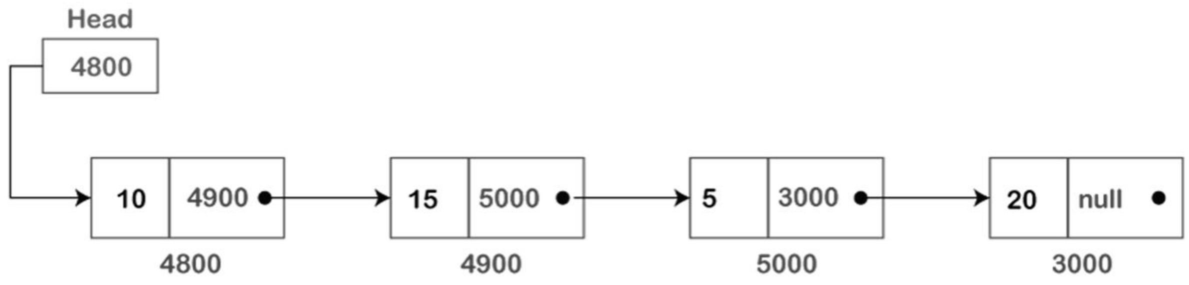
Q44. What is Linked List? List out the advantages and disadvantages of linked list.

Ans :

When we want to work with an unknown number of data values, we use a linked list data structure to organize that data. The linked list is a linear data structure that contains a sequence of elements such that each element links to its next element in the sequence. Each element in a linked list is called “Node”.

These elements are linked to each other by providing one additional information along with an element, i.e., the address of the next element. The variable that stores the address of the next element is known as a pointer. Therefore, we conclude that the linked list contains two parts, i.e., the first one is the data element, and the other is the pointer.

A linked list can also be defined as the collection of the nodes in which one node is connected to another node, and node consists of two parts, i.e., one is the data part and the second one is the address part, as shown in the below figure:



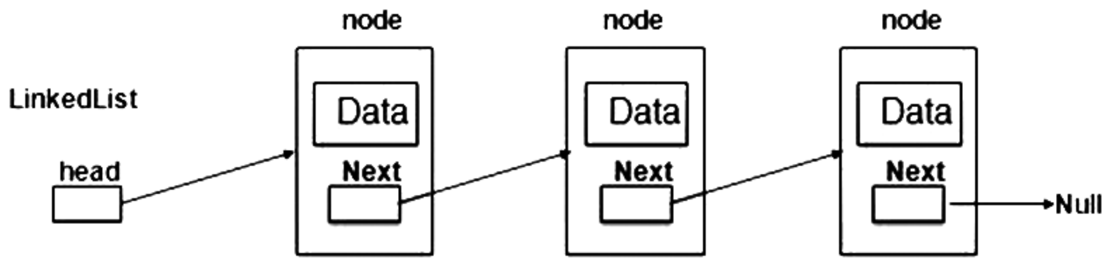
In the above figure, we can observe that each node contains the data and the address of the next node. The last node of the linked list contains the NULL value in the address part.

A linked-list is a sequence of data structures which are connected together via links.

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list the second most used data structure after array. Following are important terms to understand the concepts of Linked List.

- **Link:** Each Link of a linked list can store a data called an element.
- **Next:** Each Link of a linked list contain a link to next link called Next.
- **LinkedList:** A LinkedList contains the connection link to the first Link called First.

Linked List Representation



As per above shown illustration, following are the important points to be considered.

- LinkedList contains an link element called first.
- Each Link carries a data field(s) and a Link Field called next.
- Each Link is linked with its next link using its next link.
- Last Link carries a Link as null to mark the end of the list.

The following are the advantages of using a linked list over an array:

- **Dynamic data structure**

  The size of the linked list is not fixed as it can vary according to our requirements.

- **Insertion and Deletion**

  Insertion and deletion in linked list are easier than array as the elements in an array are stored in a consecutive location. In contrast, in the case of a linked list, the elements are stored in a random location. The complexity for insertion and deletion of elements from the beginning is O(1) in the linked list, while in the case of an array, the complexity would be O(n). If we want to insert or delete the element in an array, then we need to shift the elements for creating the space. On the other hand, in the linked list, we do not have to shift the elements. In the linked list, we just need to update the address of the pointer in the node.

- **Memory efficient**

  Its memory consumption is efficient as the size of the linked list can grow or shrink according to our requirements.

- **Implementation**

  Both the stacks and queues can be implemented using a linked list.

  The following are the disadvantages of linked list:

- **Memory usage**

  The node in a linked list occupies more memory than array as each node occupies two types of variables, i.e., one is a simple variable, and another is a pointer variable that occupies 4 bytes in the memory.

- **Traversal**

  In a linked list, the traversal is not easy. If we want to access the element in a linked list, we cannot access the element randomly, but in the case of an array, we can randomly access the element by index. For example, if we want to access the 3rd node, then we need to traverse all the nodes before it. So, the time required to access a particular node is large.

- **Reverse traversing**

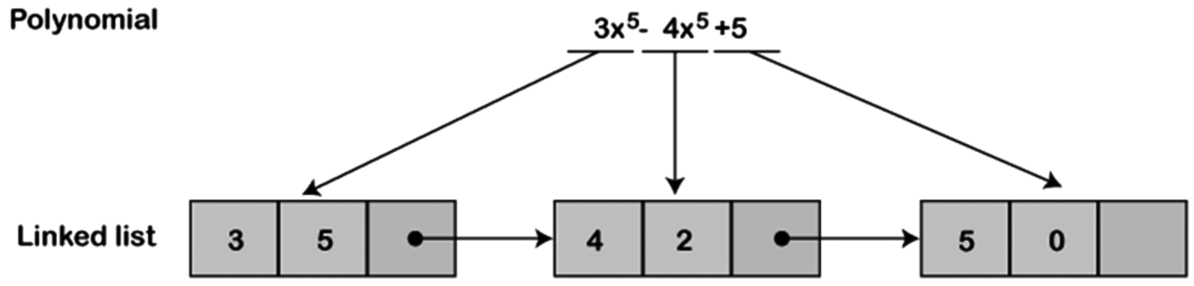
  In a linked list, backtracking or reverse traversing is difficult. In a doubly linked list, it is easier but requires more memory to store the back pointer.

Q45. What are the applications of linked list ?

Ans :

The applications of the linked list are given below:

With the help of a linked list, the polynomials can be represented as well as we can perform the operations on the polynomial. We know that polynomial is a collection of terms in which each term contains coefficient and power. The coefficients and power of each term are stored as node and link pointer points to the next element in a linked list, so linked list can be used to create, delete and display the polynomial.



- A sparse matrix is used in scientific compu-tation and numerical analysis. So, a linked list is used to represent the sparse matrix.

- The various operations like student's details, employee's details or product details can be implemented using the linked list as the linked list uses the structure data type that can hold different data types.

- Stack, Queue, tree and various other data structures can be implemented using a linked list.
- The graph is a collection of edges and vertices, and the graph can be represented as an adjacency matrix and adjacency list. If we want to represent the graph as an adjacency matrix, then it can be implemented as an array. If we want to represent the graph as an adjacency list, then it can be implemented as a linked list.
- To implement hashing, we require hash tables. The hash table contains entries that are implemented using linked list.
- A linked list can be used to implement dynamic memory allocation. The dynamic memory allocation is the memory allocation done at the run-time.

Following are the basic operations supported by a list.

- **Insertion:** add an element at the beginning of the list.
- **Deletion:** delete an element at the beginning of the list.
- **Display:** displaying complete list.
- **Search:** search an element using given key.
- **Delete:** delete an element using given key.

Q46. Explain the process of inserting a new element into the linked list.

Ans : **(Dec.-17)**

You can add elements to either the beginning, middle or end of the linked list.

1. Insert at the beginning

- Allocate memory for new node
- Store data
- Change next of new node to point to head
- Change head to point to recently created node

```
structnode *newNode;
newNode = malloc(sizeof(structnode));
newNode->data = 4;
newNode->next = head;
head = newNode;
```

2. Insert at the End

- Allocate memory for new node
- Store data
- Traverse to last node
- Change next of last node to recently created node

```
structnode *newNode;
newNode = malloc(sizeof(structnode));
newNode->data = 4;
newNode->next = NULL;
structnode *temp = head;
while(temp->next != NULL){
temp = temp->next;
}
temp->next = newNode;
```

3. Insert at the Middle

- Allocate memory and store data for new node.
- Traverse to node just before the required position of new node.
- Change next pointers to include new node in between.

```
structnode *newNode;
newNode = malloc(sizeof(structnode));
newNode->data = 4;
structnode *temp = head;
for(int i=2; i < position; i++) {
if(temp->next != NULL) {
temp = temp->next;
}
}
newNode->next = temp->next;
temp->next = newNode;
```

Q47. How can we delete an element from the linked list ?

Ans : **(Dec.-17)**

You can delete either from the beginning, end or from a particular position.

1. Delete from beginning
   - ➢ Point head to the second node

```
head = head->next;
```

2. Delete from end
   - ➢ Traverse to second last element
   - ➢ Change its next pointer to null

```
struct node* temp = head;
while(temp->next->next!=NULL){
        temp = temp->next;
}
temp->next = NULL;
```

3. Delete from middle
   - ➢ Traverse to element before the element to be deleted
   - ➢ Change next pointers to exclude the node from the chain

```
for(int i=2; i< position; i++) {
        if(temp->next!=NULL) {
        temp = temp->next;
        }
}
temp->next = temp->next->next;
```

Q48. How can we search a specific element in the linked list?

Ans : **(Dec.-17, Imp.)**

You can delete either from the beginning, end or from a particular position.

1. Delete from beginning
   - ➢ Point head to the second node

```
head = head->next;
```

2. Delete from end
   - ➢ Traverse to second last element
   - ➢ Change its next pointer to null

```
struct node* temp = head;
while(temp->next->next!=NULL){
temp = temp->next;
}
temp->next = NULL;
```

3. Delete from middle
   - ➢ Traverse to element before the element to be deleted
   - ➢ Change next pointers to exclude the node from the chain

```
for(int i=2; i< position; i++) {
if(temp->next!=NULL) {
        temp = temp->next;
        }
}
temp->next = temp->next->next;
```

Q49. How can we search a specific element in the linked list?

Ans : **(Dec.-17, Imp.)**

You can search an element on a linked list using a loop using the following steps. We are finding item on a linked list.

- ➢ Make head as the current node.
- ➢ Run a loop until the current node is NULL because the last element points to NULL.
- ➢ In each iteration, check if the key of the node is equal to item. If it the key matches the item, return true otherwise return false.

```
// Search a node
bool searchNode(structNode** head_ref, int key) {
structNode* current = *head_ref;
while (current != NULL) {
if (current->data == key) returntrue;
current = current->next;
}
returnfalse;
}
```

Q50. Explain the process of sorting the elements of a linked list ?

Ans :

Sort Elements of a Linked List

We will use a simple sorting algorithm, Bubble Sort, to sort the elements of a linked list in ascending order below.

1. Make the head as the current node and create another node index for later use.
2. If head is null, return.
3. Else, run a loop till the last node (i.e. NULL).
4. In each iteration, follow the following step 5-6.
5. Store the next node of current in index.
6. Check if the data of the current node is greater than the next node. If it is greater, swap current and index.

Check the article on bubble sort for better understanding of its working.

```
// Sort the linked list
        void sortLinkedList(struct Node** head_ref) {
        struct Node *current = *head_ref, *index = NULL;
        int temp;
                if (head_ref == NULL) {
                return;
                } else {
                while (current != NULL) {
                        // index points to the node next to current
                index = current->next;
                        while (index != NULL) {
                if (current->data > index->data) {
                temp = current->data;
                        current->data = index->data;
                         index->data = temp;
                }
          index = index->next;
        }
        current = current->next;
        }
    }
}
```

Q51. Write a C++ program to perform basic opertaions on a linked list.

Ans :

```
// Linked list operations in C++
#include <stdlib.h>
#include <iostream>
using namespace std;
// Create a node
    struct Node {
        int data;
        struct Node* next;
    };
void insertAtBeginning(struct Node** head_ref, int new_data) {
// Allocate memory to a node
struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
// insert the data
new_node->data = new_data;
new_node->next = (*head_ref);
// Move head to new node
(*head_ref) = new_node;
}
// Insert a node after a node
void insertAfter(struct Node* prev_node, int new_data) {
if (prev_node == NULL) {
cout << "the given previous node cannot be NULL";
return;
}
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = prev_node->next;
    prev_node->next = new_node;
}
// Insert at the end
void insertAtEnd(struct Node** head_ref, int new_data) {
struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
struct Node* last = *head_ref; /* used in step 5*/
     new_node->data = new_data;
    new_node->next = NULL;
```

```
  if (*head_ref == NULL) {
  *head_ref = new_node;
return;
}
while (last->next != NULL) last = last->next;
last->next = new_node;
return;
}
// Delete a node
void deleteNode(struct Node** head_ref, int key) {
struct Node *temp = *head_ref, *prev;
if (temp != NULL && temp->data == key) {
*head_ref = temp->next;
free(temp);
return;
}
// Find the key to be deleted
while (temp != NULL && temp->data != key) {
prev = temp;
temp = temp->next;
}
// If the key is not present
if (temp == NULL) return;
// Remove the node
prev->next = temp->next;
free(temp);
}
// Search a node
bool searchNode(struct Node** head_ref, int key) {
struct Node* current = *head_ref;
while (current != NULL) {
if (current->data == key) return true;
current = current->next;
}
return false;
}
```

```
        // Sort the linked list
                void sortLinkedList(struct Node** head_ref) {
                struct Node *current = *head_ref, *index = NULL;
                int temp;
                        if (head_ref == NULL) {
                        return;
                        } else {
                                while (current != NULL) {
                                // index points to the node next to current
                                index = current->next;
                        while (index != NULL) {
                        if (current->data > index->data) {
                        temp = current->data;
                current->data = index->data;
                index->data = temp;
                }
                index = index->next;
                }
                current = current->next;
                }
        }
}
        // Print the linked list
        void printList(struct Node* node) {
        while (node != NULL) {
        cout << node->data << " ";
        node = node->next;
        }
}
// Driver program
int main() {
        struct Node* head = NULL;
        insertAtEnd(&head, 1);
        insertAtBeginning(&head, 2);
        insertAtBeginning(&head, 3);
        insertAtEnd(&head, 4);
        insertAfter(head->next, 5);
```

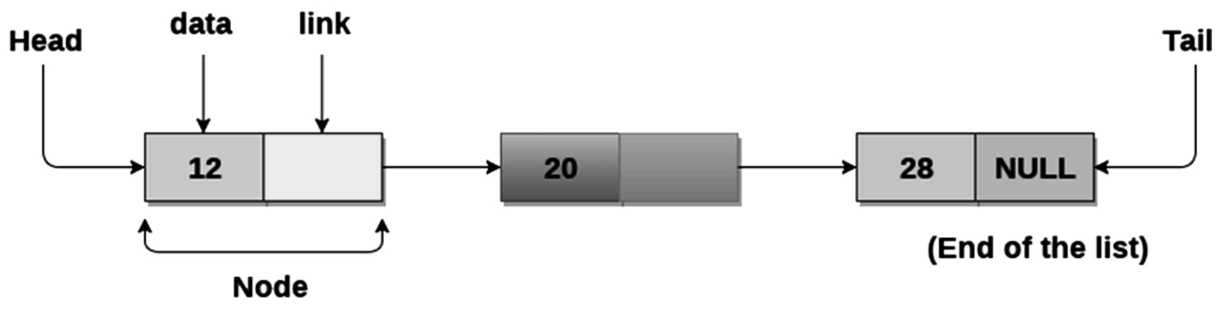
```
    cout << "Linked list: ";
    printList(head);
    cout << "\nAfter deleting an element: ";
    deleteNode(&head, 3);
    printList(head);
    int item_to_find = 3;
    if (searchNode(&head, item_to_find)) {
    cout << endl << item_to_find << " is found";
    } else {
    cout << endl << item_to_find << " is not found";
    }
        sortLinkedList(&head);
        cout << "\nSorted List: ";
        printList(head);
    }
```

2.3.2 Linked List Abstract Data Type

Q52. Discuss in detail Linked List representation using ADT.

Ans :

- Linked List can be defined as collection of objects called **nodes** that are randomly stored in the memory.
- A node contains two fields i.e. data stored at that particular address and the pointer which contains the address of the next node in the memory.
- The last node of the list contains pointer to the null.



Basic operations of linked list

Insertion

The insertion into a singly linked list can be performed at different positions. Based on the position of the new node being inserted, the insertion is categorized into the following categories.

Sl.No.	Operation	Description
1.	Insertion at beginning	It involves inserting any element at the front of the list. We just need to a few link adjustments to make the new node as the head of the list.
2.	Insertion at end of the list	It involves insertion at the last of the linked list. The new node can be inserted as the only node in the list or it can be inserted as the last one. Different logics are implemented in each scenario.
3.	Insertion after specified node	It involves insertion after the specified node of the linked list. We need to skip the desired number of nodes in order to reach the node after which the new node will be inserted.

Deletion and Traversing

The Deletion of a node from a singly linked list can be performed at different positions. Based on the position of the node being deleted, the operation is categorized into the following categories.

Sl.No.	Operation	Description
1.	Deletion at beginning	It involves deletion of a node from the beginning of the list. This is the simplest operation among all. It just need a few adjustments in the node pointers.
2.	Deletion at the end of the list	It involves deleting the last node of the list. The list can either be empty or full. Different logic is implemented for the different scenarios.
3.	Deletion after specified node	It involves deleting the node after the specified node in the list. we need to skip the desired number of nodes to reach the node after which the node will be deleted. This requires traversing through the list.
4.	Traversing	In traversing, we simply visit each node of the list at least once in order to perform some specific operation on it, for example, printing data part of each node present in the list.
5.	Searching	In searching, we match each element of the list with the given element. If the element is found on any of the location then location of that element is returned otherwise null is returned.

Q53. Write a C++ program to implement Linked List ADT.

Ans :

```
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
class Linked_list_Stack
{
        private:
```

```
    struct node
{
int data;
node *next;
};
node *top;
node *entry;
node *print;
node *bottom;
node *last_entry;
node *second_last_entry;
public:
Linked_list_Stack( );
void pop( );
void push( );
void print_list( );
void show_working( );
};
Linked_list_Stack::Linked_list_Stack ( )
{
top=NULL;
bottom=NULL;
}
void Linked_list_Stack::push( )
{
int num;
cout<<"\n\t Enter value to push onto Stack : ";
cin>>num;
entry=new node;
if(bottom==NULL)
{
entry->data=num;
entry->next=NULL;
bottom=entry;
top=entry;
}
else
{
    entry->data=num;
    entry->next=NULL;
    top->next=entry;
    top=entry;
}
cout<<"\n\t *** "<<num<<" is pushed onto the
Stack."<<endl;
}
void Linked_list_Stack::pop( )
{
if(bottom==NULL)
cout<<"\n\t *** Error : Stack is empty. \n"<<endl;
else
{
for(last_entry=bottom;last_entry->next!=NULL;
last_entry=last_entry->next)
second_last_entry=last_entry;
if(top==bottom)
bottom=NULL;
int poped_element=top->data;
delete top;
top=second_last_entry;
top->next=NULL;
cout<<"\n\t*** "<<poped_element<<" is poped from
the Stack."<<endl;
}
```

```
}
void Linked_list_Stack::print_list( )
{
print=bottom;
if(print!=NULL)
cout<<"\n\t Values pushed onto Stack are : \n"<<endl;
else
cout<<"\n\t *** Nothing to show. "<<endl;
while(print!=NULL)
{
cout<<"\t "<<print->data<<endl;
print=print->next;
}
}
void Linked_list_Stack::show_working( )
{
int choice;
clrscr( );
cout<<"\n\n********** Implementation of Linked List as a
Stack **********"<<endl;
cout<<"\n1.Push elements to stack"<<endl;
cout<<"2.Pop elements to stack"<<endl;
cout<<"3.Print the elements of stack"<<endl;
cout<<"4.Exit"<<endl;
do
{
cout<<"\nEnter your Choice : ";
cin>>choice;
switch(choice)
{
case 1: push();
break;
case 2: pop();
break;
case 3: print_list( );
break;
case 4: exit(0);
break;
default:
cout<<"enter the valid choice";
}
}while(1);
}
int main( )
{
        Linked_list_Stack obj;
        obj.show_working( );
        return 0;
}
```

2.3.3 Liked List variants, Doubly Linked List, Circular Linked List.

Q54. Discuss in short types of Linked Lists.

Ans :

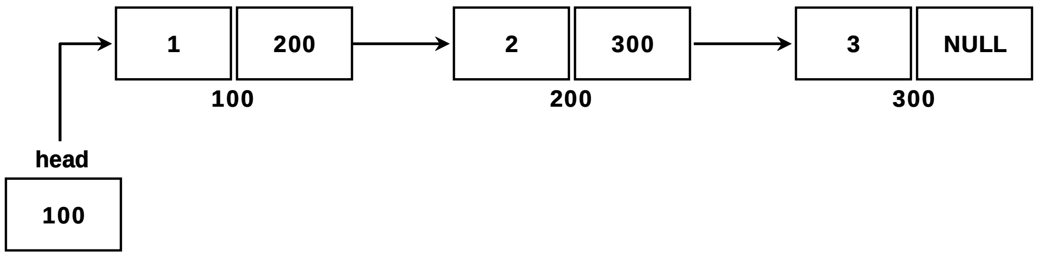
The following are the types of linked list:

- Singly Linked list
- Doubly Linked list
- Circular Linked list
- Doubly Circular Linked list

1. Singly Linked list

It is the commonly used linked list in programs. If we are talking about the linked list, it means it is a singly linked list. The singly linked list is a data structure that contains two parts, i.e., one is the data part, and the other one is the address part, which contains the address of the next or the successor node. The address part in a node is also known as a **pointer**.

Suppose we have three nodes, and the addresses of these three nodes are 100, 200 and 300 respectively. The representation of three nodes as a linked list is shown in the below figure:

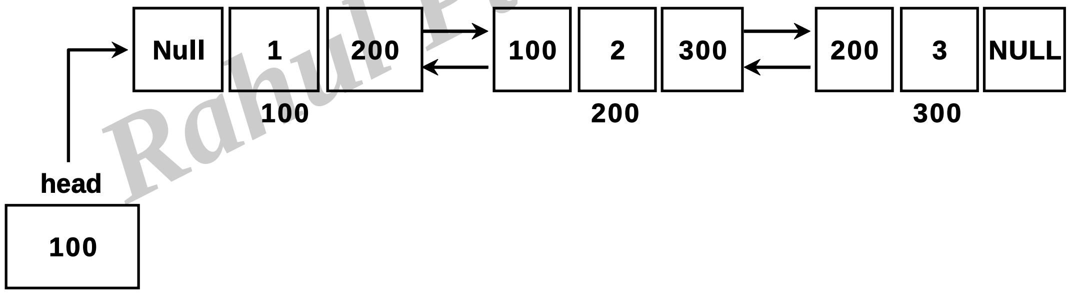


- We can observe in the above figure that there are three different nodes having address 100, 200 and 300 respectively. The first node contains the address of the next node, i.e., 200, the second node contains the address of the last node, i.e., 300, and the third node contains the NULL value in its address part as it does not point to any node. The pointer that holds the address of the initial node is known as a head pointer.
- The linked list, which is shown in the above diagram, is known as a singly linked list as it contains only a single link. In this list, only forward traversal is possible; we cannot traverse in the backward direction as it has only one link in the list.

(2) Doubly linked list

- As the name suggests, the doubly linked list contains two pointers. We can define the doubly linked list as a linear data structure with three parts: the data part and the other two address part. In other words, a doubly linked list is a list that has three parts in a single node, includes one data part, a pointer to its previous node, and a pointer to the next node.

  Suppose we have three nodes, and the address of these nodes are 100, 200 and 300, respectively. The representation of these nodes in a doubly-linked list is shown below:

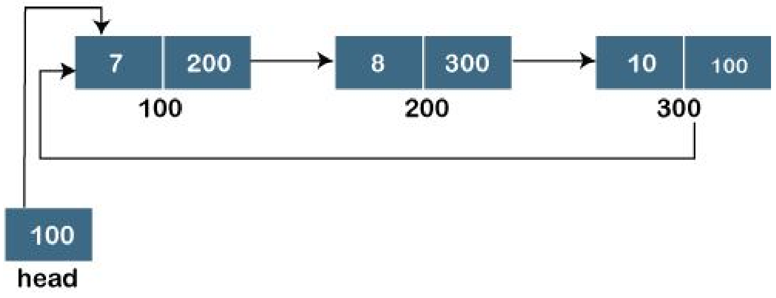


- As we can observe in the above figure, the node in a doubly-linked list has two address parts; one part stores the ***address of the next*** while the other part of the node stores the ***previous node's address***. The initial node in the doubly linked list has the **NULL** value in the address part, which provides the address of the previous node.

(3) Circular linked list

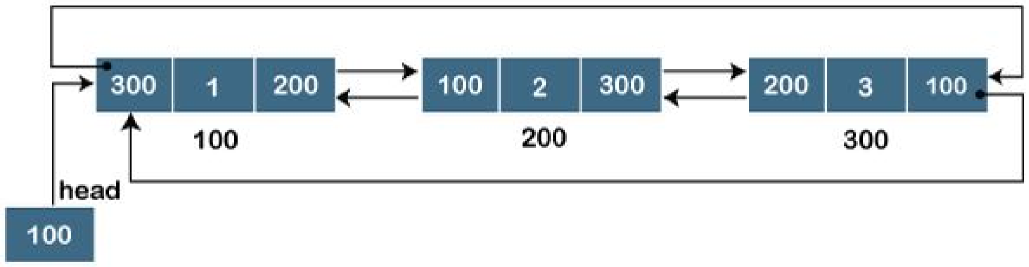
A circular linked list is a variation of a singly linked list. The only difference between the ***singly linked list*** and a ***circular linked*** list is that the last node does not point to any node in a singly linked list, so its link part contains a NULL value. On the other hand, the circular linked list is a list in which the last node connects to the first node, so the link part of the last node holds the first node's address. The circular linked list has no starting and ending node.

A circular linked list is a sequence of elements in which each node has a link to the next node, and the last node is having a link to the first node. The representation of the circular linked list will be similar to the singly linked list, as shown below:



(4) Doubly Circular linked list

The doubly circular linked list has the features of both the circular linked list and doubly linked list.

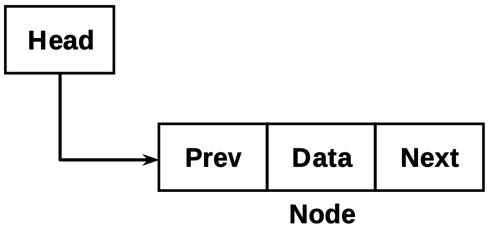


The above figure shows the representation of the doubly circular linked list in which the last node is attached to the first node and thus creates a circle. It is a doubly linked list also because each node holds the address of the previous node also. The main difference between the doubly linked list and doubly circular linked list is that the doubly circular linked list does not contain the NULL value in the previous field of the node. As the doubly circular linked contains three parts, i.e., two address parts and one data part so its representation is similar to the doubly linked list.

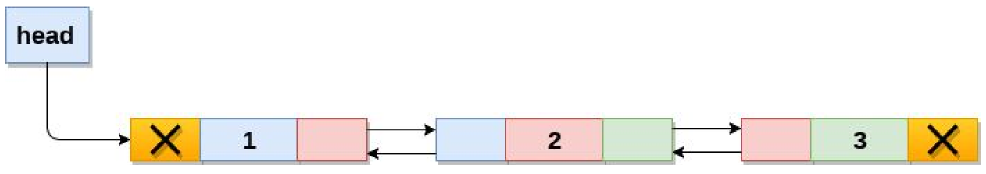
Q55. Discuss in detail about Doubly Linked List. Explain the basic operations of Doubly linked list.

Ans : **(Dec.-19)**

Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer) , pointer to the previous node (previous pointer). A sample node in a doubly linked list is shown in the figure.



A doubly linked list containing three nodes having numbers from 1 to 3 in their data part, is shown in the following image.



Doubly Linked List

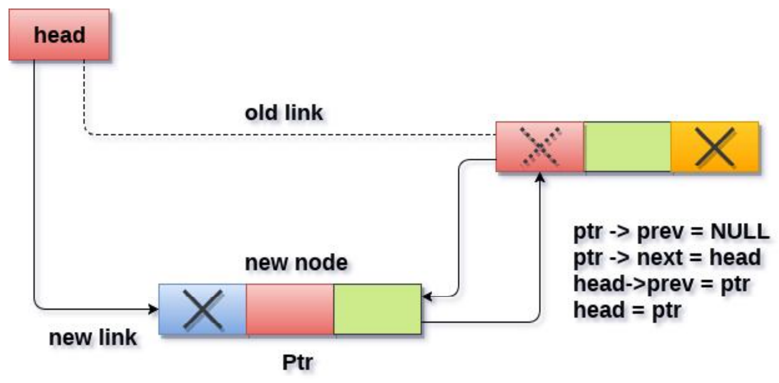
Following are the operations of Doubly linked list

Sl.No.	Operation	Description
1	Insertion at beginning	Adding the node into the linked list at beginning.
2	Insertion at end	Adding the node into the linked list to the end.
3	Insertion after specified node	Adding the node into the linked list after the specified node.
4	Deletion at beginning	Removing the node from beginning of the list
5	Deletion at the end	Removing the node from end of the list.
6	Deletion of the node having given data	Removing the node which is present just after the node containing the given data.
7	Searching	Comparing each node data with the item to be searched and return the location of the item in the list if the item found else return null.
8	Traversing	Visiting each node of the list at least once in order to perform some specific operation like searching, sorting, display, etc.

Insertion in doubly linked list at beginning

Algorithm

```
Step 1:  IF ptr = NULL
             Write OVERFLOW
             Go to Step 9
             [END OF IF]
Step 2:  SET NEW_NODE = ptr
Step 3:  SET ptr = ptr -> NEXT
Step 4:  SET NEW_NODE -> DATA = VAL
Step 5:  SET NEW_NODE -> PREV = NULL
Step 6:  SET NEW_NODE -> NEXT = START
Step 7:  SET head -> PREV = NEW_NODE
Step 8:  SET head = NEW_NODE
Step 9:  EXIT
```

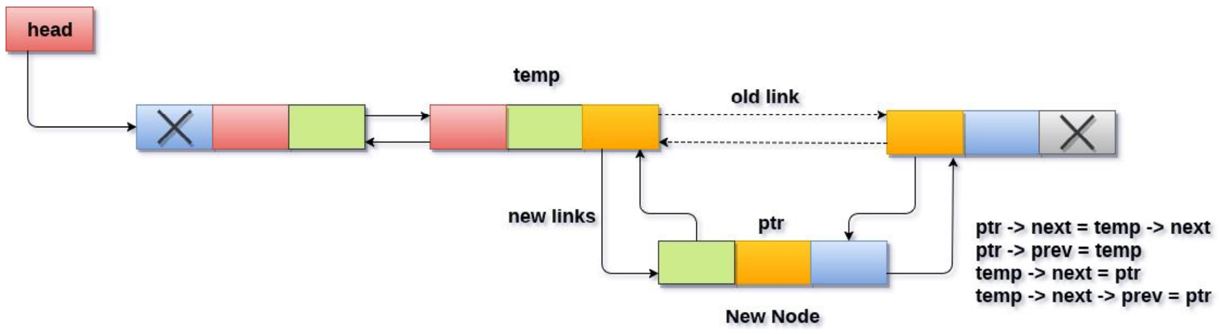


Insertion into doubly linked list at beginning

Insertion in doubly linked list after Specified node

Algorithm

```
Step 1:  IF PTR = NULL
         Write OVERFLOW
         Go to Step 15
         [END OF IF]
Step 2:  SET NEW_NODE = PTR
Step 3:  SET PTR = PTR -> NEXT
Step 4:  SET NEW_NODE -> DATA = VAL
Step 5:  SET TEMP = START
Step 6:  SET I = 0
Step 7:  REPEAT 8 to 10 until I<="" li="">
Step 8:  SET TEMP = TEMP -> NEXT
Step 9:  IF TEMP = NULL
Step 10: WRITE "LESS THAN DESIRED NO. OF ELEMENTS"
         GOTO STEP 15
         [END OF IF]
         [END OF LOOP]
Step 11: SET NEW_NODE -> NEXT = TEMP -> NEXT
Step 12: SET NEW_NODE -> PREV = TEMP
Step 13 : SET TEMP -> NEXT = NEW_NODE
Step 14: SET TEMP -> NEXT -> PREV = NEW_NODE
Step 15: EXIT
```



Insertion into doubly linked list after specified node

Q56. Write a C++ program to demonstrate on Doubly Linked List.

Ans :

```
#include<iostream>
usingnamespace std;
structNode{
        int data;
        structNode*prev;
        structNode*next;
    };
    structNode* head = NULL;
    void insert(int newdata){
        structNode* newnode =(structNode*) malloc(sizeof(structNode));
        newnode->data = newdata;
        newnode->prev = NULL;
        newnode->next= head;
        if(head != NULL)
        head->prev = newnode ;
        head = newnode;
    }
    void display(){
        structNode* ptr;
        ptr = head;
        while(ptr != NULL){
        cout<< ptr->data <<" ";
```

```
            ptr = ptr->next;
    }
}
        int main(){
        insert(3);
        insert(1);
        insert(7);
        insert(2);
        insert(9);
        cout<<"The doubly linked list is: ";
        display();
        return0;
}
```

Output

```
The doubly linked list is: 9 2 7 1 3
```

Q57. Discuss In detail deletion, search in doubly linked list.

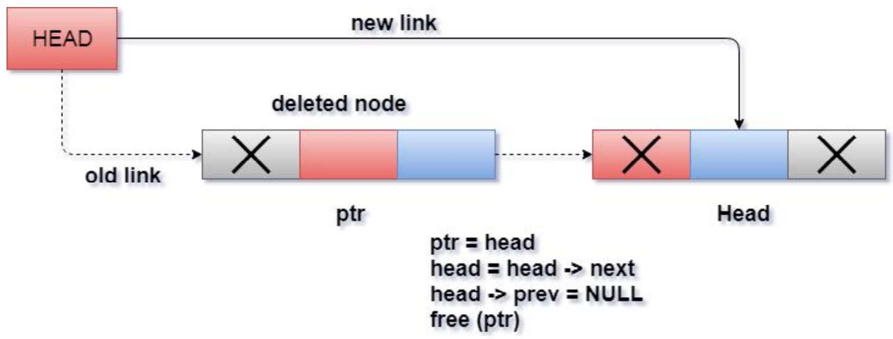
Ans : **(Dec.-19)**

Operations on Circular Singly linked list

Deletion at beginning

Algorithm

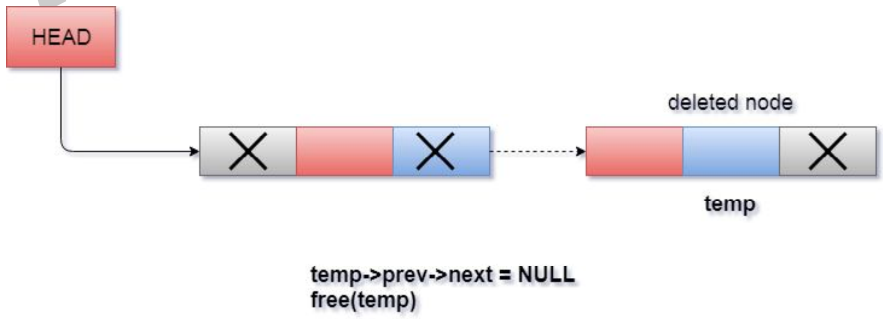
```
STEP 1: IF HEAD = NULL
            WRITE UNDERFLOW
            GOTO STEP 6
STEP 2: SET PTR = HEAD
STEP 3: SET HEAD = HEAD '! NEXT
STEP 4: SET HEAD '! PREV = NULL
STEP 5: FREE PTR
STEP 6: EXIT
```



Deletion in doubly linked list from beginning

Deletion in doubly linked list at the end

```
Step 1:  IF HEAD = NULL
         Write UNDERFLOW
         Go to Step 7
         [END OF IF]
Step 2:  SET TEMP = HEAD
Step 3:  REPEAT STEP 4 WHILE TEMP->NEXT != NULL
Step 4:  SET TEMP = TEMP->NEXT
         [END OF LOOP]
Step 5:  SET TEMP ->PREV-> NEXT = NULL
Step 6:  FREE TEMP
Step 7:  EXIT
```



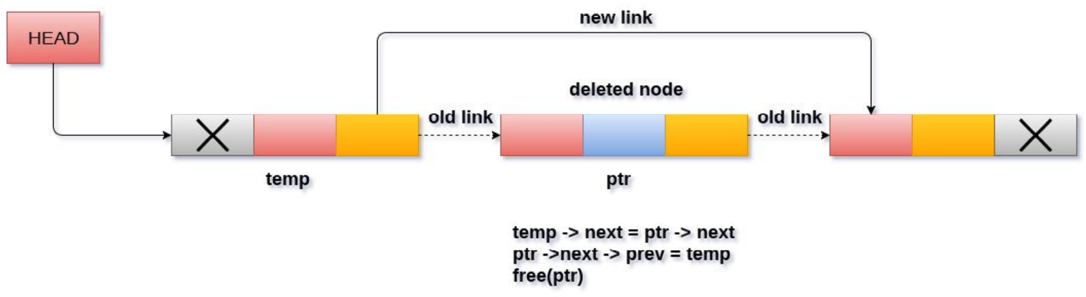
Deletion in doubly linked list at the end

Deletion in doubly linked list after the specified node

Algorithm

```
Step 1:  IF HEAD = NULL
         Write UNDERFLOW
```

```
          Go to Step 9
          [END OF IF]
Step 2:   SET TEMP = HEAD
Step 3:   Repeat Step 4 while TEMP -> DATA != ITEM
Step 4:   SET TEMP = TEMP -> NEXT
          [END OF LOOP]
Step 5:   SET PTR = TEMP -> NEXT
Step 6:   SET TEMP -> NEXT = PTR -> NEXT
Step 7:   SET PTR -> NEXT -> PREV = TEMP
Step 8:   FREE PTR
Step 9:   EXIT
```



Deletion of a specified node in doubly linked list

Searching for a specific node in Doubly Linked List

Algorithm

```
Step 1:   IF HEAD = = NULL
          WRITE "UNDERFLOW"
          GOTO STEP 8
          [END OF IF]
Step 2:   Set PTR = HEAD
Step 3:   Set i = 0
Step 4:   Repeat step 5 to 7 while PTR != NULL
Step 5:   IF PTR '! data = item
          return i
          [END OF IF]
Step 6:   i = i + 1
Step 7:   PTR = PTR '! next
Step 8:   Exit
```

Q58. Write a C++ program to demonstrate on Doubly Linked List.

Ans :

```
        #include<iostream>
        usingnamespace std;
        structNode{
                    int data;
                    structNode*prev;
                    structNode*next;
        };
              structNode* head = NULL;
              void insert(int newdata){
                    structNode* newnode =(structNode*) malloc(sizeof(structNode));
                    newnode->data = newdata;
                    newnode->prev = NULL;
                    newnode->next= head;
                    if(head != NULL)
              head->prev = newnode ;
              head = newnode;
        }
        void display(){
        structNode* ptr;
        ptr = head;
        while(ptr != NULL){
        cout<< ptr->data <<" ";
        ptr = ptr->next;
    }
}
int main(){
        insert(3);
        insert(1);
        insert(7);
        insert(2);
        insert(9);
        cout<<"The doubly linked list is: ";
        display();
        return0;
}
```

Output

```
The doubly linked list is: 9 2 7 1 3
```

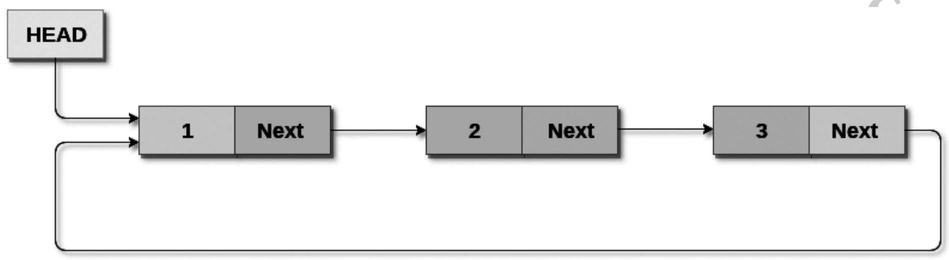
Q59. Discuss In detail about Circular Single Linked List.

Ans :

In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.

We traverse a circular singly linked list until we reach the same node where we started. The circular singly liked list has no beginning and no ending. There is no null value present in the next part of any of the nodes.

The following image shows a circular singly linked list.



Circular Singly Linked List

Operations on Circular Singly Linked List

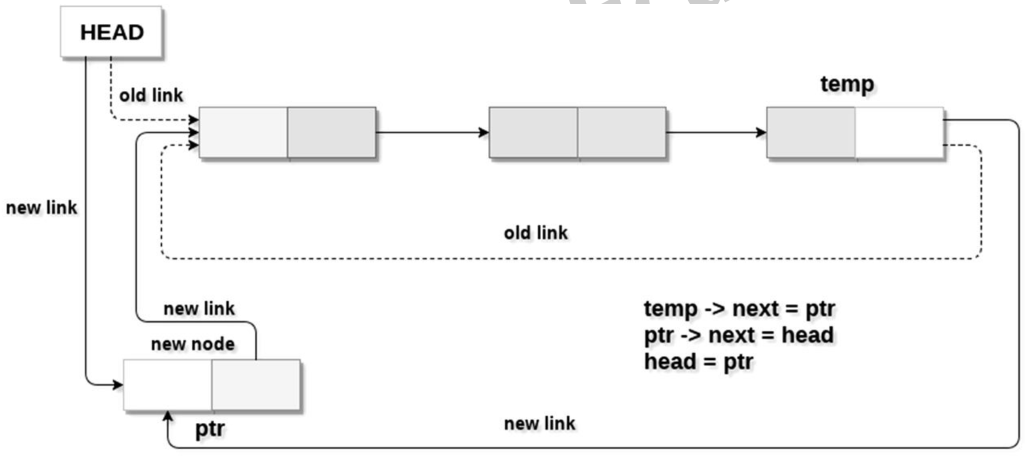
Sl. No.	Operation	Description
1.	Insertion at beginning	Adding a node into circular singly linked list at the beginning.
2.	Insertion at the end	Adding a node into circular singly linked list at the end.

Deletion and Traversing

Sl. No.	Operation	Description
1	Deletion at beginning	Removing the node from circular singly linked list at the beginning.
2	Deletion at the end	Removing the node from circular singly linked list at the end.
3	Searching	Compare each element of the node with the given item and return the location at which the item is present in the list otherwise return null.
4	Traversing	Visiting each element of the list at least once in order to perform some specific operation.

Insertion into circular singly linked list at beginning

```
Step 1:  IF PTR = NULL
         Write OVERFLOW
         Go to Step 11
         [END OF IF]
Step 2:  SET NEW_NODE = PTR
Step 3:  SET PTR = PTR -> NEXT
Step 4:  SET NEW_NODE -> DATA = VAL
Step 5:  SET TEMP = HEAD
Step 6:  Repeat Step 8 while TEMP -> NEXT != HEAD
Step 7:  SET TEMP = TEMP -> NEXT
         [END OF LOOP]
Step 8:  SET NEW_NODE -> NEXT = HEAD
Step 9:  SET TEMP ’! NEXT = NEW_NODE
Step 10: SET HEAD = NEW_NODE
Step 11: EXIT
```



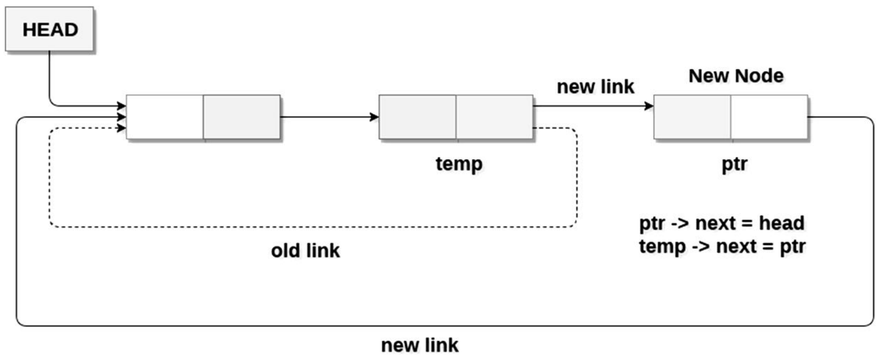
Insertion into circular singly linked list at beginning

Insertion into circular singly linked list at the end

Algorithm

```
Step 1:  IF PTR = NULL
         Write OVERFLOW
         Go to Step 1
         [END OF IF]
```

```
Step 2:   SET NEW_NODE = PTR
Step 3:   SET PTR = PTR -> NEXT
Step 4:   SET NEW_NODE -> DATA = VAL
Step 5:   SET NEW_NODE -> NEXT = HEAD
Step 6:   SET TEMP = HEAD
Step 7:   Repeat Step 8 while TEMP -> NEXT != HEAD
Step 8:   SET TEMP = TEMP -> NEXT
          [END OF LOOP]
Step 9:   SET TEMP -> NEXT = NEW_NODE
Step 10:  EXIT
```

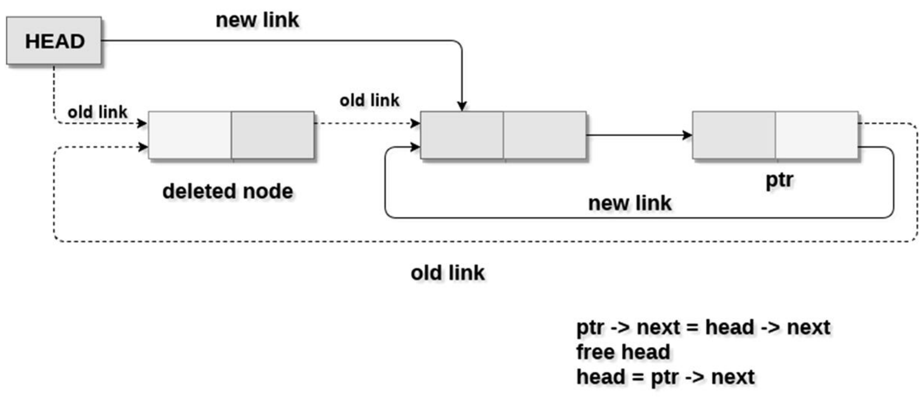


Insertion into circular singly linked list at end

Deletion in circular singly linked list at beginning

Algorithm

```
Step 1:   IF HEAD = NULL
          Write UNDERFLOW
          Go to Step 8
          [END OF IF]
Step 2:   SET PTR = HEAD
Step 3:   Repeat Step 4 while PTR ’! NEXT != HEAD
Step 4:   SET PTR = PTR ’! next
          [END OF LOOP]
Step 5:   SET PTR ’! NEXT = HEAD ’! NEXT
Step 6:   FREE HEAD
Step 7:   SET HEAD = PTR ’! NEXT
Step 8:   EXIT
```

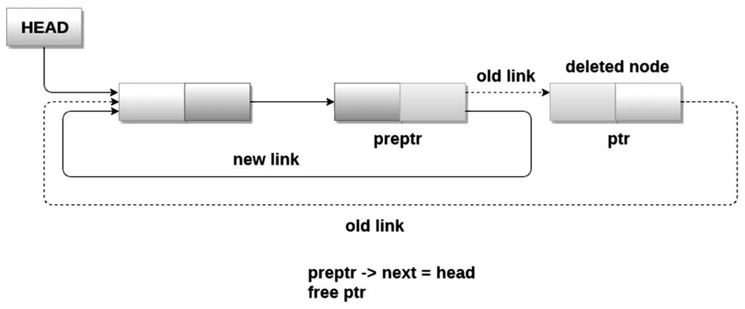


Deletion in circular singly linked list at beginning

Deletion in Circular singly linked list at the end

Algorithm

```
Step 1:  IF HEAD = NULL
         Write UNDERFLOW
         Go to Step 8
         [END OF IF]
Step 2:  SET PTR = HEAD
Step 3:  Repeat Steps 4 and 5 while PTR -> NEXT != HEAD
Step 4:  SET PREPTR = PTR
Step 5:  SET PTR = PTR -> NEXT
         [END OF LOOP]
Step 6:  SET PREPTR -> NEXT = HEAD
Step 7:  FREE PTR
Step 8:  EXIT
```



Deletion in circular singly linked list at end

Q60. Write a C++ program to implement Circular Linked List.

Ans ;

```
        #include<iostream>
        usingnamespace std;
        structNode{
              int data;
              structNode*next;
        };
              structNode* head = NULL;
              void insert(int newdata){
                     structNode*newnode =(structNode*)malloc(sizeof(structNode));
                     structNode*ptr = head;
                     newnode->data = newdata;
                     newnode->next= head;
                     if(head!= NULL){
              while(ptr->next!= head)
              ptr = ptr->next;
              ptr->next= newnode;
        } else
        newnode->next= newnode;
        head = newnode;
}
void display(){
        structNode* ptr;
        ptr = head;
        do{
        cout<<ptr->data <<" ";
        ptr = ptr->next;
        }while(ptr != head);
}
int main(){
        insert(3);
        insert(1);
        insert(7);
        insert(2);
        insert(9);
        cout< < "The circular linked list is: ";
        display();
        return0;
        }
```

Output

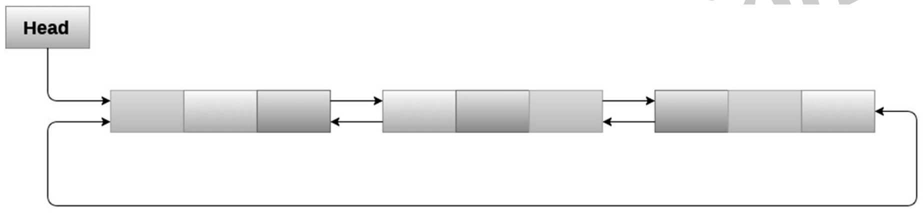
```
The circular linked list is: 9 2 7 1 3
```

Q61. Discuss in detail about Circular Doubly linked list.

Ans :

Circular doubly linked list is a more complexed type of data structure in which a node contain pointers to its previous node as well as the next node. Circular doubly linked list doesn't contain NULL in any of the node. The last node of the list contains the address of the first node of the list. The first node of the list also contain address of the last node in its previous pointer.

A circular doubly linked list is shown in the following figure.



Circular Doubly Linked List

Operations on circular doubly linked list

There are various operations which can be performed on circular doubly linked list. The node structure of a circular doubly linked list is similar to doubly linked list. However, the operations on circular doubly linked list is described in the following table.

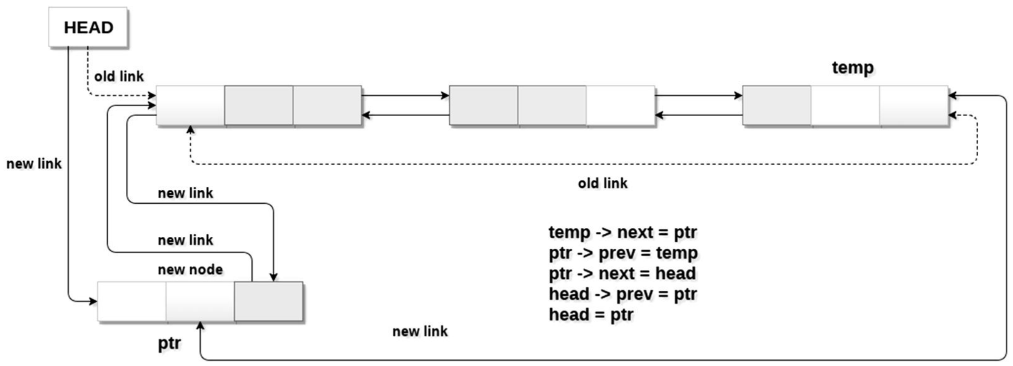
Sl. No.	Operation	Description
1	Insertion at beginning	Adding a node in circular doubly linked list at the beginning.
2	Insertion at end	Adding a node in circular doubly linked list at the end.
3	Deletion at beginning	Removing a node in circular doubly linked list from beginning.
4	Deletion at end	Removing a node in circular doubly linked list at the end.

Insertion in circular doubly linked list at beginning

Algorithm

```
Step 1: IF PTR = NULL
            Write OVERFLOW
            Go to Step 13
            [END OF IF]
```

```
Step 2:   SET NEW_NODE = PTR
Step 3:   SET PTR = PTR -> NEXT
Step 4:   SET NEW_NODE -> DATA = VAL
Step 5:    SET TEMP = HEAD
Step 6:   Repeat Step 7 while TEMP -> NEXT != HEAD
Step 7:   SET TEMP = TEMP -> NEXT
          [END OF LOOP]
Step 8:   SET TEMP -> NEXT = NEW_NODE
Step 9:   SET NEW_NODE -> PREV = TEMP
Step 1:   SET NEW_NODE -> NEXT = HEAD
Step 11:  SET HEAD -> PREV = NEW_NODE
Step 12:  SET HEAD = NEW_NODE
Step 13:  EXIT
```

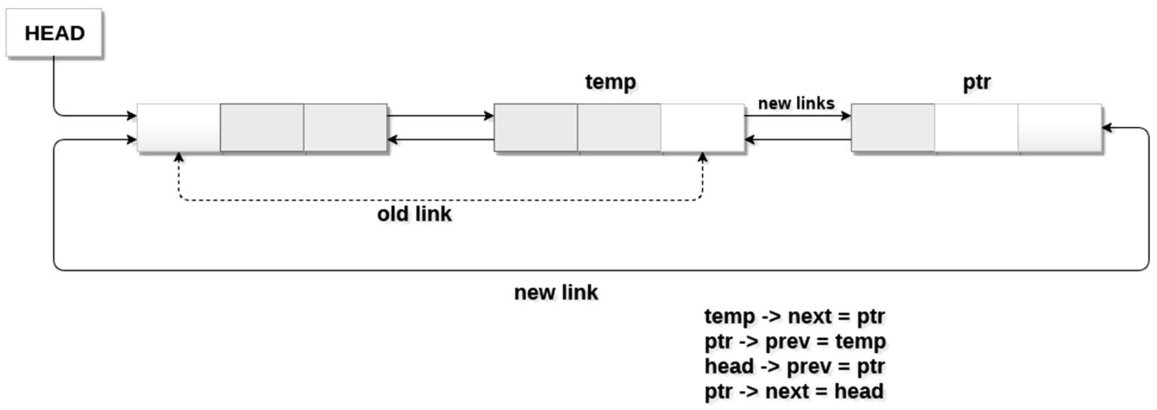


Insertion into circular doubly linked list at beginning

Insertion in circular doubly linked list at end

Algorithm

```
Step 1:   IF PTR = NULL
              Write OVERFLOW
              Go to Step 12
          [END OF IF]
Step 2:   SET NEW_NODE = PTR
Step 3:   SET PTR = PTR -> NEXT
Step 4:   SET NEW_NODE -> DATA = VAL
Step 5:   SET NEW_NODE -> NEXT = HEAD
Step 6:   SET TEMP = HEAD
Step 7:   Repeat Step 8 while TEMP -> NEXT != HEAD
Step 8:       SET TEMP = TEMP -> NEXT
          [END OF LOOP]
Step 9:   SET TEMP -> NEXT = NEW_NODE
Step 10:  SET NEW_NODE -> PREV = TEMP
Step 11:  SET HEAD -> PREV = NEW_NODE
Step 12:  EXIT
```

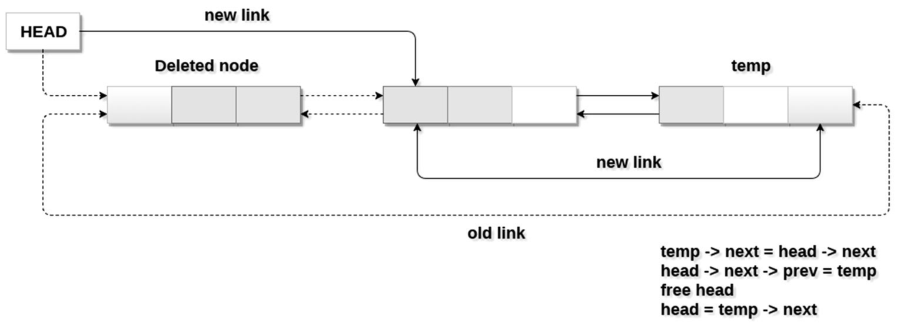


Insertion into circular doubly linked list at end

Deletion in Circular doubly linked list at beginning

Algorithm

```
Step 1:  IF HEAD = NULL
             Write UNDERFLOW
             Go to Step 8
         [END OF IF]
Step 2:  SET TEMP = HEAD
Step 3:  Repeat Step 4 while TEMP -> NEXT != HEAD
Step 4:  SET TEMP = TEMP -> NEXT
         [END OF LOOP]
Step 5:  SET TEMP -> NEXT = HEAD -> NEXT
Step 6:  SET HEAD -> NEXT -> PREV = TEMP
Step 7:  FREE HEAD
Step 8:  SET HEAD = TEMP -> NEXT
```



Deletion in circular doubly linked list at beginning

Deletion in circular doubly linked list at end

Algorithm

```
Step 1:  IF HEAD = NULL
         Write UNDERFLOW
         Go to Step 8
         [END OF IF]
Step 2:  SET TEMP = HEAD
Step 3:  Repeat Step 4 while TEMP -> NEXT != HEAD
Step 4:  SET TEMP = TEMP -> NEXT
         [END OF LOOP]
Step 5:  SET TEMP -> PREV -> NEXT = HEAD
Step 6:  SET HEAD -> PREV = TEMP -> PREV
Step 7:  FREE TEMP
Step 8:  EXIT
```

Q62. What are the differences between single linked list and doubly linked list.

Ans :

Basis of comparison	Singly linked list	Doubly linked list
Definition	A single linked list is a list of nodes in which node has two parts, the first part is the data part, and the next part is the pointer pointing to the next node in the sequence of nodes.	A doubly linked list is also a collection of nodes in which node has three fields, the first field is the pointer containing the address of the previous node, the second is the data field, and the third is the pointer containing the address of the next node.
Access	The singly linked list can be traversed only in the forward direction.	The doubly linked list can be accessed in both directions.
List pointer	It requires only one list pointer variable, i.e., the head pointer pointing to the node.	It requires two list pointer variables, head and last. The head pointer points to the first node, and the last pointer points to the last node of the list.
Memory space	It utilizes less memory space.	It utilizes more memory space.
Efficiency	It is less efficient as compared to a doubly-linked list.	It is more efficient.
Implementation	It can be implemented on the stack.	It can be implemented on stack, heap and binary tree.
Complexity	In a singly linked list, the time complexity for inserting and deleting an element from the list is O(n).	In a doubly-linked list, the time complexity for inserting and deleting an element is O(1).

2.3.4 Representation of Sparse Matrix using Linked List

Q63. What is Sparse Matrix?

Ans :

In computer programming, a matrix can be defined with a 2-dimensional array. Any array with ‘m’ columns and ‘n’ rows represent a m X n matrix. There may be a situation in which a matrix contains more number of ZERO values than NON-ZERO values. Such matrix is known as sparse matrix.

When a sparse matrix is represented with a 2-dimensional array, we waste a lot of space to represent that matrix. For example, consider a matrix of size 100 × 100 containing only 10 non-zero elements. In this matrix, only 10 spaces are filled with non-zero values and remaining spaces of the matrix are filled with zero. That means, totally we allocate 100 × 100 × 2 = 20000 bytes of space to store this integer matrix. And to access these 10 non-zero elements we have to make scanning for 10000 times. To make it simple we use the following sparse matrix representation.Sparse Matrix Representations

A sparse matrix can be represented by using TWO representations, those are as follows...

1. Triplet Representation (Array Representation)
2. Linked Representation

1. Triplet Representation (Array Representation)

In this representation, we consider only non-zero values along with their row and column index values. In this representation, the 0th row stores the total number of rows, total number of columns and the total number of non-zero values in the sparse matrix.

For example, consider a matrix of size 5 X 6 containing 6 number of non-zero values. This matrix can be represented as shown in the image...

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 9 & 0 \\ 0 & 8 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 2 & 0 & 0 & 0 \end{bmatrix} \Rightarrow$$

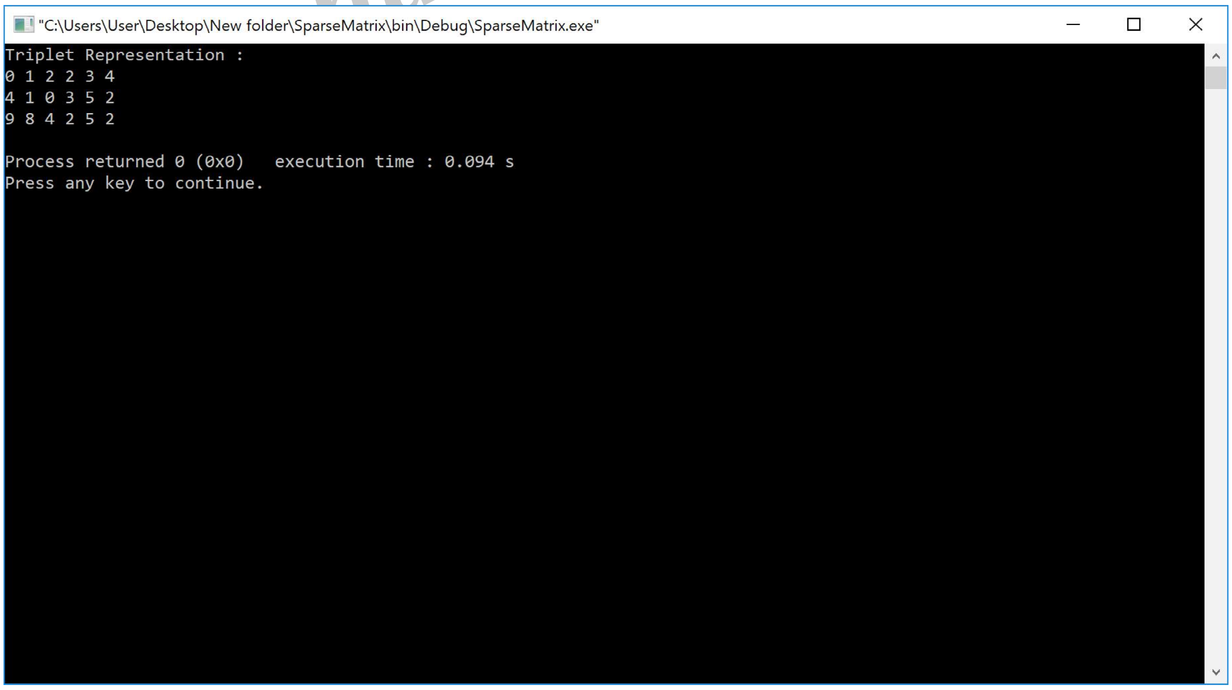
Rows	Colums	Values
5	6	6
0	4	9
1	1	8
2	0	4
2	3	2
2	3	2
3	5	5
4	2	2

In above example matrix, there are only 6 non-zero elements (those are 9, 8, 4, 2, 5 and 2) and matrix size is 5 × 6. We represent this matrix as shown in the above image. Here the first row in the right side table is filled with values 5, 6 & 6 which indicates that it is a sparse matrix with 5 rows, 6 columns & 6 non-zero values. The second row is filled with 0, 4, & 9 which indicates the non-zero value 9 is at the 0th-row 4th column in the Sparse matrix. In the same way, the remaining non-zero values also follow a similar pattern.

```
#include<iostream>
using namespace std;
int main()
{
  // sparse matrix of class 5x6 with 6 non-zero values
   int sparseMatrix[5][6] =
   {
      {0 , 0 , 0 , 0 , 9, 0 },
      {0 , 8 , 0 , 0 , 0, 0 },
      {4 , 0 , 0 , 2 , 0, 0 },
      {0 , 0 , 0 , 0 , 0, 5 },
      {0 , 0 , 2 , 0 , 0, 0 }
   };
      //Finding total non-zero values in the sparse
      matrix
      int size = 0;
      for (int row = 0; row < 5; row++)
      for (int column = 0; column < 6;
           column++)
      if (sparseMatrix[row][column] != 0)
          size++;
      // Defining result Matrix
      int resultMatrix[3][size];
      // Generating result matrix
      int k = 0;
      for (int row = 0; row < 5; row++)
      for (int column = 0; column < 6; column
      ++)
```

```
        if (sparseMatrix[row][column] != 0)
        {
            resultMatrix[0][k] = row;
            resultMatrix[1][k] = column;
            resultMatrix[2][k] = sparseMatrix[row][column];
            k++;
        }
        // Displaying result matrix
        cout<<"Triplet Representation : "<<endl;
        for (int row=0; row<3; row++)
        {
        for (int column = 0; column<size; column++)
        cout<<resultMatrix[row][column]<<" ";
       cout<<endl;
    }
    return 0;
}
```

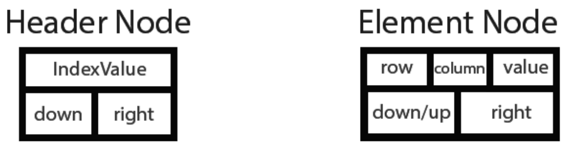
Output


"C:\Users\User\Desktop\New folder\SparseMatrix\bin\Debug\SparseMatrix.exe"

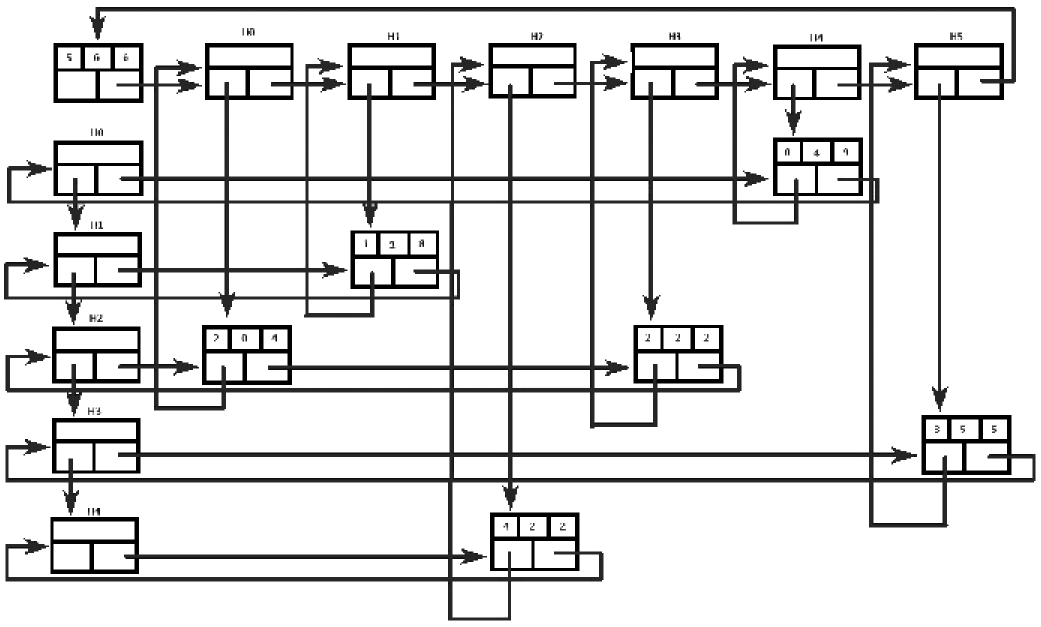
```
Triplet Representation :
0 1 2 2 3 4
4 1 0 3 5 2
9 8 4 2 5 2

Process returned 0 (0x0)   execution time : 0.094 s
Press any key to continue.
```

Linked Representation

In linked representation, we use a linked list data structure to represent a sparse matrix. In this linked list, we use two different nodes namely **header node** and **element node**. Header node consists of three fields and element node consists of five fields as shown in the image...



Consider the above same sparse matrix used in the Triplet representation. This sparse matrix can be represented using linked representation as shown in the below image...



In the above representation, H0, H1,..., H5 indicates the header nodes which are used to represent indexes. Remaining nodes are used to represent non-zero elements in the matrix, except the very first node which is used to represent abstract information of the sparse matrix (i.e., It is a matrix of 5 X 6 with 6 non-zero elements).

In this representation, in each row and column, the last node right field points to its respective header node.

2.3.5 Linked Stack , Linked Queue

Q64. What is Linked Stack ? Explain in detail.

(OR)

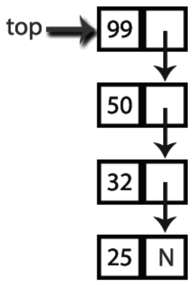
Explain the implementation of stack using linked list representation.

Ans :

The major problem with the stack implemented using an array is, it works only for a fixed number of data values. That means the amount of data must be specified at the beginning of the implementation itself. Stack implemented using an array is not suitable, when we don't know the size of data which we are going to use. A stack data structure can be implemented by using a linked list data structure. The stack implemented using linked list can work for an unlimited number of values. That means, stack implemented using linked list works for the variable size of data. So, there is no need to fix the size at the beginning of the implementation. The Stack implemented using linked list can organize as many data values as we want.

In linked list implementation of a stack, every new element is inserted as '**top**' element. That means every newly inserted element is pointed by '**top**'. Whenever we want to remove an element from the stack, simply remove the node which is pointed by '**top**' by moving '**top**' to its previous node in the list. The **next** field of the first element must be always **NULL**.

Example



In the above example, the last inserted node is 99 and the first inserted node is 25. The order of elements inserted is 25, 32,50 and 99.

Stack Operations using Linked List

To implement a stack using a linked list, we need to set the following things before implementing actual operations.

Step 1: Include all the header files which are used in the program. And declare all the user defined functions.

Step 2: Define a 'Node' structure with two members data and next.

Step 3: Define a **Node** pointer '**top**' and set it to **NULL**.

Step 4: Implement the **main** method by displaying Menu with list of operations and make suitable function calls in the **main** method.

Push(value) - Inserting an element into the Stack

We can use the following steps to insert a new node into the stack...

Step 1: Create a **newNode** with given value.

Step 2: Check whether stack is **Empty** (**top** = = **NULL**)

Step 3: If it is **Empty**, then set **newNode'™ next** = **NULL**.

Step 4: If it is **Not Empty**, then set **new Node'™ next** = **top**.

Step 5: Finally, set **top** = **newNode**.

pop() - Deleting an Element from a Stack

We can use the following steps to delete a node from the stack...

Step 1: Check whether **stack** is **Empty** (**top == NULL**).

Step 2: If it is **Empty**, then display **"Stack is Empty!!! Deletion is not possible!!!"** and terminate the function.

Step 3: If it is **Not Empty**, then define a **Node** pointer '**temp**' and set it to '**top**'.

Step 4: Then set '**top** = **top '! next**'.

Step 5: Finally, delete '**temp**'. (**free(temp)**). **display() - Displaying stack of elements**

We can use the following steps to display the elements (nodes) of a stack...

Step 1: Check whether stack is **Empty**(**top == NULL**).

Step 2: If it is **Empty**, then display **'Stack is Empty!!!'** and terminate the function.

Step 3: If it is **Not Empty**, then define a Node pointer **'temp'** and initialize with **top**.

Step 4: Display '**temp '! data** —>' and move it to the next node. Repeat the same until **temp** reaches to the first node in the stack. (**temp '! next** != **NULL**).

Step 5: Finally! Display '**temp '! data** —> **NULL**'.

Q65. Write a C++ program to implement stack using linked list representation.

Ans :

```
        #include<iostream>
        usingnamespace std;
        structNode{
              int data;
              structNode*next;
        };
        structNode* top = NULL;
        void push(int val){
              structNode* newnode =(structNode*) malloc(sizeof(structNode));
              newnode->data = val;
              newnode->next= top;
              top = newnode;
        }
        void pop(){
              if(top==NULL)
              cout<<"Stack Underflow"<<endl;
        else{
        cout<<"The popped element is "<< top->data <<endl;
        top = top->next;
    }
}
void display(){
        structNode* ptr;
        if(top==NULL)
        cout<<"stack is empty";
        else{
        ptr = top;
        cout<<"Stack elements are: ";
        while(ptr != NULL){
            cout<< ptr->data <<" ";
            ptr = ptr->next;
        }
    }
    cout<<endl;
}
```

```
int main(){
    int ch, val;
    cout<<"1) Push in stack"<<endl;
    cout<<"2) Pop from stack"<<endl;
    cout<<"3) Display stack"<<endl;
    cout<<"4) Exit"<<endl;
    do{
        cout<<"Enter choice: "<<endl;
        cin>>ch;
        switch(ch){
            case1:{
                cout<<"Enter value to be
                            pushed:"<<endl;
                cin>>val;
                push(val);
                break;
            }
            case2:{
                pop();
                break;
            }
            case3:{
                display();
                break;
            }
            case4:{
                cout<<"Exit"<<endl;
                break;
            }
            default:{
                cout<<"Invalid Choice"<<endl;
            }
        }
    }while(ch!=4);
    return0;
}
```

Output

```
1)    Push in stack
2)    Pop from stack
3)    Display stack
4)    Exit
Enter choice: 1
Enter value to be pushed: 2
Enter choice: 1
Enter value to be pushed: 6
Enter choice: 1
Enter value to be pushed: 8
Enter choice: 1
Enter value to be pushed: 7
Enter choice: 2
The popped element is 7
Enter choice: 3
Stack elements are:8 6 2
Enter choice: 5
Invalid Choice
Enter choice: 4
Exit
```

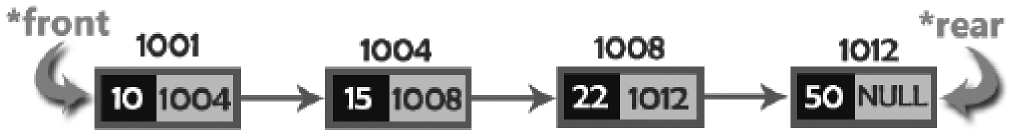
Q66. Explain the implementation of queue using linked list representation.

Ans :

The major problem with the queue implemented using an array is, It will work for an only fixed number of data values. That means, the amount of data must be specified at the beginning itself. Queue using an array is not suitable when we don't know the size of data which we are going to use. A queue data structure can be implemented using a linked list data structure. The queue which is implemented using a linked list can work for an unlimited number of values. That means, queue using linked list can work for the variable size of data (No need to fix the size at the beginning of the implementation). The Queue implemented using linked list can organize as many data values as we want.

In linked list implementation of a queue, the last inserted node is always pointed by '**rear**' and the first node is always pointed by '**front**'.

Example



In above example, the last inserted node is 50 and it is pointed by '**rear**' and the first inserted node is 10 and it is pointed by '**front**'. The order of elements inserted is 10, 15, 22 and 50.

Operations

To implement queue using linked list, we need to set the following things before implementing actual operations.

Step 1 - Include all the **header files** which are used in the program. And declare all the **user defined functions**.

Step 2 - Define a '**Node**' structure with two members **data** and **next**.

Step 3 - Define two **Node** pointers '**front**' and '**rear**' and set both to **NULL**.

Step 4 - Implement the **main** method by displaying Menu of list of operations and make suitable function calls in the **main** method to perform user selected operation.

enQueue(value) - Inserting an element into the Queue

We can use the following steps to insert a new node into the queue...

Step 1 - Create a **newNode** with given value and set '**newNode '™ next**' to **NULL**.

Step 2 - Check whether queue is **Empty** (**rear** == **NULL**)

Step 3 - If it is **Empty** then, set **front** = **newNode** and **rear** = **newNode**.

Step 4 - If it is **Not Empty** then, set **rear '™ next** = **newNode** and **rear** = **newNode**.

deQueue() - Deleting an Element from Queue

We can use the following steps to delete a node from the queue...

Step 1 - Check whether **queue** is **Empty** (**front == NULL**).

Step 2 - If it is **Empty**, then display **"Queue is Empty!!! Deletion is not possible!!!"** and terminate from the function

Step 3 - If it is **Not Empty** then, define a Node pointer '**temp**' and set it to '**front**'.

Step 4 - Then set '**front** = **front '! next**' and delete '**temp**' (**free(temp)**).

display() - Displaying the elements of Queue

We can use the following steps bto display the elements (nodes) of a queue...

Step 1 - Check whether queue is **Empty** (**front** == **NULL**).

Step 2 - If it is **Empty** then, display **'Queue is Empty!!!'** and terminate the function.

Step 3 - If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **front**.

Step 4 - Display '**temp '! data** —>' and move it to the next node. Repeat the same until '**temp**' reaches to '**rear**' (**temp '! next** != **NULL**).

Step 5 - Finally! Display '**temp '! data** —> **NULL**'.

Q67. Write a C++ program to implement queue using linked list representation

Ans :

```
#include<iostream>
usingnamespace std;
struct node {
        int data;
        struct node *next;
};
struct node* front = NULL;
struct node* rear = NULL;
struct node* temp;
voidInsert(){
        int val;
        cout<<"Insert the element in queue : "<<endl;
        cin>>val;
        if(rear == NULL){
        rear =(struct node *)malloc(sizeof(struct node));
        rear->next= NULL;
        rear->data = val;
        front = rear;
        }else{
        temp=(struct node *)malloc(sizeof(struct node));
        rear->next= temp;
        temp->data = val;
        temp->next= NULL;
        rear = temp;
    }
}
voidDelete(){
        temp = front;
        if(front == NULL){
        cout<<"Underflow"<<endl;
        return;
    }
    else
        if(temp->next!= NULL){
        temp = temp->next;
        cout<<"Element deleted from queue is : "<<front->data<<endl;
        free(front);
        front = temp;
        }else{
        cout<<"Element deleted from queue is : "<<front->data<<endl;
        free(front);
        front = NULL;
        rear = NULL;
    }
}
voidDisplay(){
    temp = front;
    if((front == NULL)&&(rear == NULL)){
        cout<<"Queue is empty"<<endl;
        return;
    }
    cout<<"Queue elements are: ";
    while(temp != NULL){
        cout<<temp->data<<" ";
        temp = temp->next;
    }
    cout<<endl;
}
int main(){
    int ch;
        cout<<"1) Insert element to queue"<<endl;
        cout<<"2) Delete element from queue"<<endl;
        cout<<"3) Display all the elements of queue"<<endl;
        cout<<"4) Exit"<<endl;
        do{
        cout<<"Enter your choice : "<<endl;
        cin>>ch;
        switch(ch){
            case1:Insert();
            break;
            case2:Delete();
            break;
            case3:Display();
            break;
            case4: cout<<"Exit"<<endl;
            break;
            default: cout<<"Invalid choice"<<endl;
        }
    }while(ch!=4);
    return0;
}
```

Output

The output of the above program is as follows

```
1)	Insert element to queue
2)	Delete element from queue
3)	Display all the elements of queue
4)	Exit
Enter your choice : 1
Insert the element in queue : 4
Enter your choice : 1
Insert the element in queue : 3
Enter your choice : 1
Insert the element in queue : 5
Enter your choice : 2
Element deleted from queue is : 4
Enter your choice : 3
Queue elements are : 3 5
Enter your choice : 7
Invalid choice
Enter your choice : 4
Exit
```

Short Question and Answers

1. Explain the use of stack to find the factorial of a number using recrussion.

Ans :

The process in which a function calls itself is known as recursion and the corresponding function is called the recursive function. The popular example to understand the recursion is factorial function.

Factorial function

f(n) = n*f(n-1), base condition: if n<=1 then f(n) = 1. Don't worry we wil discuss what is base condition and why it is important.

In the following diagram. I have shown that how the factorial function is calling itself until the function reaches to the base condition.

Factorial function:f(n) = n*f(n–1)

Lets say we want to find out the factorial of 5 which means n = 5.

```
f(5)=5*f(5–1)=5*f(4)
                  ↓
             5*4*f(4–1)=  20*f(3)
                              ↓
                      20*3*f(3-1)=60*f(2)
                                      ↓
                             60*2*f(2-1)  =  120*f(1)
                                                 ↓
                                     120*1*f(1-1)  =  120*f(0)
                                                          ↓
                                                     120*1=120
```

```
#include<iostream>
usingnamespace std;
//Factorial function
int f(int n){
/* This is called the base condition, it is
     * very important to specify the base condition
     * in recursion, otherwise your program will throw
     * stack overflow error.
*/
if (n <= 1)
      return1;
else
      return n*f(n-1);
}
```

```
int main(){
int num;
        cout<<"Enter a number: ";
        cin>>num;
        cout<<"Factorial of entered number: "<<f(num);
return0;
}
```

Output

```
Enter a number: 5
Factorial of entered number: 120.
```

2. List out the advantages and disadvantages of recursion ?

Ans :

Advantages

- ➢ Less number code lines are used in the recursion program and hence the code looks shorter and cleaner.
- ➢ Recursion is easy to approach to solve the problems involving data structure and algorithms like graph and tree.
- ➢ Recursion helps to reduce the time complexity.
- ➢ It helps to reduce unnecessary calling of the function.
- ➢ It helps to solve the stack evolutions and prefix, infix, postfix evaluation.
- ➢ Recursion is the best method to define objects that have repeated structural forms.

Disadvantages

- ➢ It consumes a lot of stack space
- ➢ It takes more time to process the program
- ➢ If an error is accrued in the program, it is difficult to debug the error in comparison to the iterative program.

3. Write a C++ program to find factorial of given number using recursion.

Ans :

```
#include <iostream>
using namespace std;
int fact(int n);
int main()
{
int n;
        cout <<"Enter a positive integer: ";
        cin >> n;
        cout <<"Factorial of "<< n <<" = "<< fact(n);
return0;
        }
        int fact(int n)
        {
        if(n >1)
        return n * fact(n -1);
        else
        return1;
        }
```

Output

```
Enter an positive integer: 6
Factorial of 6 = 720
```

4. Write a c++ program to reverse a number using recursion.

Ans :

```
        #include <iostream.h>
        using namespace std;
        int reverseNumber(int n) {
                static temp,sum;
                if(n>0){
                temp = n%10;
                sum=sum*10+ temp;
        reverseNumber(n/10);
        }
        else
        {
        returnsum;
        }
        }
        int main()
        {
        int n,reverse;
        cout<<"Enter number";
        cin >> n;
        reverse = reverseNumber(n);
        cout <<"Reverse of number is"<< reverse;
        return0;
}
```

Output

```
Enter number : 3456
Reverse of number is : 6543
```

5. Define Recurrence.

Ans :

Recurrence means, the number of times, that a function has recused. It may also mean, simply, that something has been repeated. It may even be a variable.

A recurrence is a well-defined mathematical function where the function being defined is applied within its own definition. The mathematical function factorial of n can also be defined recursively as n! = n × (n - 1)!, where 1! = 1 Fibonacci sequence as an example. The Fibonacci sequence is the sequence of numbers 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ... The first two numbers of the sequence are both 1, whereas each succeeding number is the sum of the preceding two numbers (we arrived at 55 as the 10th number; it is the sum of 21 and 34, the eighth and ninth numbers).

6. What is recursive function?

Ans :

Recursive Functions is the process of defining something in terms of itself. It is a function that calls itself again in the body of the function.

A function fact ( ), that computes the factorial of an integer 'N' ,which is the product of all whole numbers from 1 to N.

When fact ( ) is called with an argument of 1 (or) 0, the function returns 1. Otherwise, it returns the product of n*fact (n-1), this happens until 'n' equals 1.

```
Fact (5) = 5* fact (4)
         = 5*4*3* fact (3)
         = 5*4*3*2* fact (2)
         = 5*4*3*2*1 fact (1)
         = 5*4*3*2*1
         = 120.
```

7. What are the differences between Iteration and recursion.

Ans :

On the basis	Recursion	Iteration
Basic	Recursion is the process of calling a function itself within its own code.	In iteration, there is a repeated execution of the set of instructions. In Iteration, loops are used to execute the set of instructions repetitively until the condition is false.
Syntax	There is a termination condition is specified.	The format of iteration includes initialization, condition, and increment/decrement of a variable.
Termination	The termination condition is defined within the recursive function.	Here, the termination condition is defined in the definition of the loop.
Code size	The code size in recursion is smaller than the code size in iteration.	The code size in iteration is larger than the code size in recursion.
Infinite	If the recursive function does not meet to a termination condition, it leads to an infinite recursion. There is a chance of system crash in infinite recursion.	Iteration will be infinite, if the control condition of the iteration statement never becomes false. On infinite loop, it repeatedly used CPU cycles.
Applied	It is always applied to functions.	It is applied to loops.
Speed	It is slower than iteration.	It is faster than recursion.
Usage	Recursion is generally used where there is no issue of time complexity, and code size requires being small.	It is used when we have to balance the time complexity against a large code size.
Time complexity	It has high time complexity.	The time complexity in iteration is relatively lower. We can calculate its time complexity by finding the no. of cycles being repeated in a loop.
Stack	It has to update and maintain the stack.	There is no utilization of stack.
Memory	It uses more memory as compared to iteration.	It uses less memory as compared to recursion.
Overhead	There is an extensive overhead due to updating and maintaining the stack.	There is no overhead in iteration.

8. What is mean by Queue? What are the basic operations of Queue?

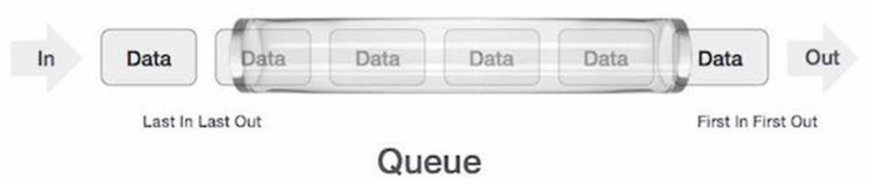
Ans :

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

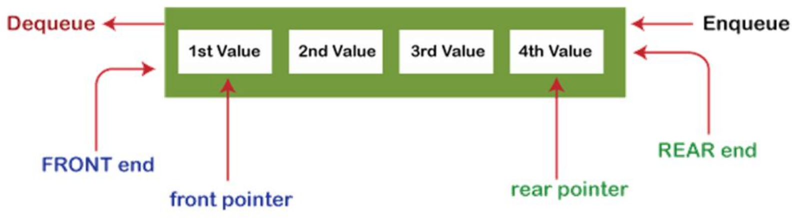
Queue Representation

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure:



As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.

A queue in the data structure can be considered similar to the queue in the real-world. A queue is a data structure in which whatever comes first will go out first. It follows the FIFO (First-In-First-Out) policy. In Queue, the insertion is done from one end known as the rear end or the tail of the queue, whereas the deletion is done from another end known as the front end or the head of the queue. In other words, it can be defined as a list or a collection with a constraint that the insertion can be performed at one end called as the rear end or tail of the queue and deletion is performed on another end called as the front end or the head of the queue.



9. List out the basic operation of a Queue.

Ans :

There are two fundamental operations performed on a Queue:

- **Enqueue:** The enqueue operation is used to insert the element at the rear end of the queue. It returns void.
- **Dequeue:** The dequeue operation performs the deletion from the front-end of the queue. It also returns the element which has been removed from the front-end. It returns an integer value. The dequeue operation can also be designed to void.
- **Peek:** This is the third operation that returns the element, which is pointed by the front pointer in the queue but does not delete it.
- **Queue overflow (isfull):** When the Queue is completely full, then it shows the overflow condition.
- **Queue underflow (isempty):** When the Queue is empty, i.e., no elements are in the Queue then it throws the underflow condition.

10. Explain in detail about Dequeue.

Ans :

The dequeue stands for Double Ended Queue. In the queue, the insertion takes place from one end while the deletion takes place from another end. The end at which the insertion occurs is known as the rear end whereas the end at which the deletion occurs is known as front end.

Deque is a linear data structure in which the insertion and deletion operations are performed from both ends. We can say that deque is a generalized version of the queue.

Properties of deque

Deque can be used both as **stack** and **queue** as it allows the insertion and deletion operations on both ends.

In deque, the insertion and deletion operation can be performed from one side. The stack follows the LIFO rule in which both the insertion and deletion can be performed only from one end; therefore, we conclude that dequeue can be considered as a stack.

In deque, the insertion can be performed on one end, and the deletion can be done on another end. The queue follows the FIFO rule in which the element is inserted on one end and deleted from another end. Therefore, we conclude that the deque can also be considered as the queue.

There are two types of Queues, Input-restricted queue, and output-restricted queue.

1. **Input-restricted queue:** The input-restricted queue means that some restrictions are applied to the insertion. In input-restricted queue, the insertion is applied to one end while the deletion is applied from both the ends.

2. **Output-restricted queue:** The output-restricted queue means that some restrictions are applied to the deletion operation. In an output-restricted queue, the deletion can be applied only from one end, whereas the insertion is possible from both ends.

11. What is a Priority Queue?

Ans :

A priority queue is an abstract data type that behaves similarly to the normal queue except that each element has some priority, i.e., the element with the highest priority would come first in a priority queue. The priority of the elements in a priority queue will determine the order in which elements are removed from the priority queue.

The priority queue supports only comparable elements, which means that the elements are either arranged in an ascending or descending order.

For example, suppose we have some values like 1, 3, 4, 8, 14, 22 inserted in a priority queue with an ordering imposed on the values is from least to the greatest. Therefore, the 1 number would be having the highest priority while 22 will be having the lowest priority.

Characteristics of a Priority queue

- Every element in a priority queue has some priority associated with it.
- An element with the higher priority will be deleted before the deletion of the lesser priority.
- If two elements in a priority queue have the same priority, they will be arranged using the FIFO principle.

12. List out the applications of Queues.

Ans :

Applications of Queue

Due to the fact that queue performs actions on first in first out basis which is quite fair for the ordering of actions. There are various applications of queues discussed as below.

1. Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
2. Queues are used in asynchronous transfer of data (where data is not being transferred at the same rate between two processes) for eg. pipes, file IO, sockets.
3. Queues are used as buffers in most of the applications like MP3 media player, CD player, etc.

4. Queue are used to maintain the play list in media players in order to add and remove the songs from the play-list.

5. Queues are used in operating systems for handling interrupts.

13. What are the differences between stack and Queue.

Ans :

Basis for comparison	**Stack**	**Queue**
Principle	It follows the principle LIFO (Last In-First Out), which implies that the element which is inserted last would be the first one to be deleted.	It follows the principle FIFO (First In - First Out), which implies that the element which is added first would be the first element to be removed from the list.
Structure	It has only one end from which both the insertion and deletion take place, and that end is known as a top.	It has two ends, i.e., front and rear end. The front end is used for the deletion while the rear end is used for the insertion.
Number of pointers used	It contains only one pointer known as a top pointer. The top pointer holds the address of the last inserted or the topmost element of the stack.	It contains two pointers front and rear pointer. The front pointer holds the address of the first element, whereas the rear pointer holds the address of the last element in a queue.
Operations performed	It performs two operations, push and pop. The push operation inserts the element in a list while the pop operation removes the element from the list.	It performs mainly two operations, enqueue and dequeue. The enqueue operation performs the insertion of the elements in a queue while the dequeue operation performs the deletion of the elements from the queue.
Examination of the empty condition	If top==-1, which means that the stack is empty.	If front== -1 or front = rear +1, which means that the queue is empty.
Examination of full condition	If top== max-1, this condition implies that the stack is full.	If rear==max-1, this condition implies that the stack is full.
Variants	It does not have any types.	It is of three types like priority queue, circular queue and double ended queue.
Implementation	It has a simpler implementation.	It has a comparatively complex implementation than a stack.
Visualization	A Stack is visualized as a vertical collection.	A Queue is visualized as a horizontal collection.

14. What are the differences between linear queue and circular queue.

Ans :

Basis of comparison	Linear Queue	Circular Queue
Meaning	The linear queue is a type of linear data structure that contains the elements in a sequential manner.	The circular queue is also a linear data structure in which the last element of the Queue is connected to the first element, thus creating a circle.
Insertion and Deletion	In linear queue, insertion is done from the rear end, and deletion is done from the front end.	In circular queue, the insertion and deletion can take place from any end.
Memory space	The memory space occupied by the linear queue is more than the circular queue.	It requires less memory as compared to linear queue.
Memory utilization	The usage of memory is inefficient.	The memory can be more efficiently utilized.
Order of execution	It follows the FIFO principle in order to perform the tasks.	It has no specific order for execution.

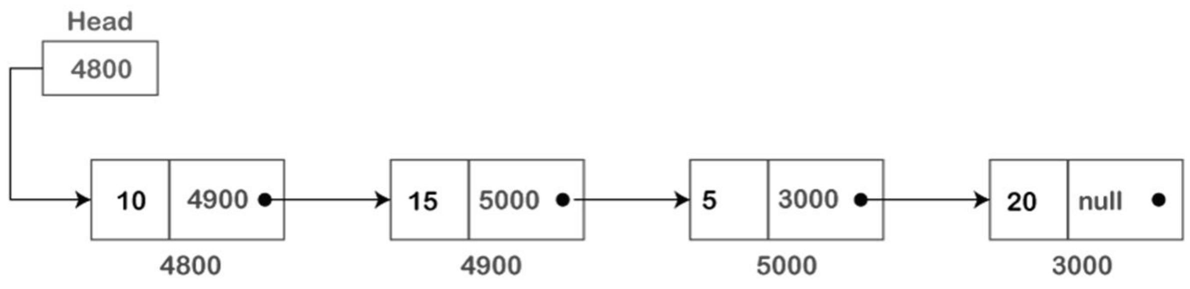
15. What is mean by linked list.

Ans :

When we want to work with an unknown number of data values, we use a linked list data structure to organize that data. The linked list is a linear data structure that contains a sequence of elements such that each element links to its next element in the sequence. Each element in a linked list is called “Node”.

These elements are linked to each other by providing one additional information along with an element, i.e., the address of the next element. The variable that stores the address of the next element is known as a pointer. Therefore, we conclude that the linked list contains two parts, i.e., the first one is the data element, and the other is the pointer.

A linked list can also be defined as the collection of the nodes in which one node is connected to another node, and node consists of two parts, i.e., one is the data part and the second one is the address part, as shown in the below figure:



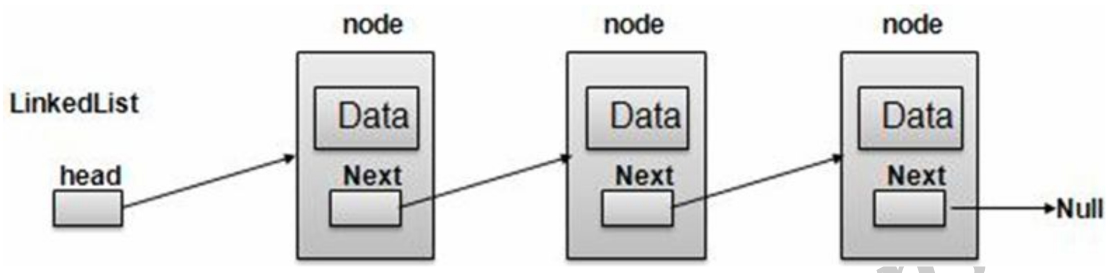
In the above figure, we can observe that each node contains the data and the address of the next node. The last node of the linked list contains the NULL value in the address part.

A linked-list is a sequence of data structures which are connected together via links.

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list the second most used data structure after array. Following are important terms to understand the concepts of Linked List.

- **Link:** Each Link of a linked list can store a data called an element.
- **Next:** Each Link of a linked list contain a link to next link called Next.
- **LinkedList:** A LinkedList contains the connection link to the first Link called First.

Linked List Representation



As per above shown illustration, following are the important points to be considered.

- LinkedList contains an link element called first.
- Each Link carries a data field(s) and a Link Field called next.
- Each Link is linked with its next link using its next link.
- Last Link carries a Link as null to mark the end of the list.

16. What are the advantges of a linked list.

Ans :

The following are the advantages of using a linked list over an array:

- **Dynamic data structure**

  The size of the linked list is not fixed as it can vary according to our requirements.

- **Insertion and Deletion**

  Insertion and deletion in linked list are easier than array as the elements in an array are stored in a consecutive location. In contrast, in the case of a linked list, the elements are stored in a random location. The complexity for insertion and deletion of elements from the beginning is O(1) in the linked list, while in the case of an array, the complexity would be O(n). If we want to insert or delete the element in an array, then we need to shift the elements for creating the space. On the other hand, in the linked list, we do not have to shift the elements. In the linked list, we just need to update the address of the pointer in the node.

- **Memory efficient**

  Its memory consumption is efficient as the size of the linked list can grow or shrink according to our requirements.

- **Implementation**

  Both the stacks and queues can be implemented using a linked list.

  The following are the disadvantages of linked list:

- **Memory usage**

  The node in a linked list occupies more memory than array as each node occupies two types of variables, i.e., one is a simple variable, and another is a pointer variable that occupies 4 bytes in the memory.

- **Traversal**

  In a linked list, the traversal is not easy. If we want to access the element in a linked list, we cannot access the element randomly, but in the case of an array, we can randomly access the element by index. For example, if we want to access the 3rd node, then we need to traverse all the nodes before it. So, the time required to access a particular node is large.

- **Reverse traversing**

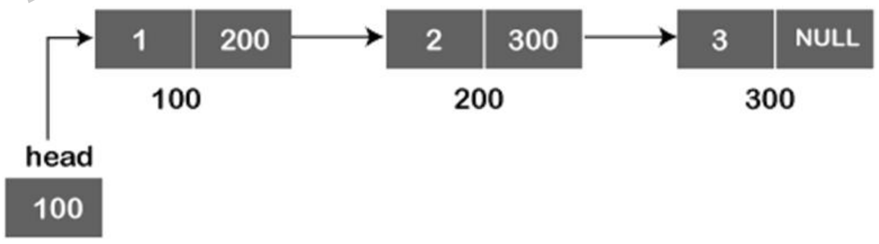
  In a linked list, backtracking or reverse traversing is difficult. In a doubly linked list, it is easier but requires more memory to store the back pointer.

17. Explain the insertion of a node in single linked list with example.

Ans :

It is the commonly used linked list in programs. If we are talking about the linked list, it means it is a singly linked list. The singly linked list is a data structure that contains two parts, i.e., one is the data part, and the other one is the address part, which contains the address of the next or the successor node. The address part in a node is also known as a **pointer**.

Suppose we have three nodes, and the addresses of these three nodes are 100, 200 and 300 respectively. The representation of three nodes as a linked list is shown in the below figure:



- We can observe in the above figure that there are three different nodes having address 100, 200 and 300 respectively. The first node contains the address of the next node, i.e., 200, the second node contains the address of the last node, i.e., 300, and the third node contains the NULL value in its address part as it does not point to any node. The pointer that holds the address of the initial node is known as a head pointer.

- The linked list, which is shown in the above diagram, is known as a singly linked list as it contains only a single link. In this list, only forward traversal is possible; we cannot traverse in the backward direction as it has only one link in the list.

18. What are the differences between single linked list and doubly linked list.

Ans :

Basis of comparison	Singly linked list	Doubly linked list
Definition	A single linked list is a list of nodes in which node has two parts, the first part is the data part, and the next part is the pointer pointing to the next node in the sequence of nodes.	A doubly linked list is also a collection of nodes in which node has three fields, the first field is the pointer containing the address of the previous node, the second is the data field, and the third is the pointer containing the address of the next node.
Access	The singly linked list can be traversed only in the forward direction.	The doubly linked list can be accessed in both directions.
List pointer	It requires only one list pointer variable, i.e., the head pointer pointing to the node.	It requires two list pointer variables, head and last. The head pointer points to the first node, and the last pointer points to the last node of the list.
Memory space	It utilizes less memory space.	It utilizes more memory space.
Efficiency	It is less efficient as compared to a doubly-linked list.	It is more efficient.
Implementation	It can be implemented on the stack.	It can be implemented on stack, heap and binary tree.
Complexity	In a singly linked list, the time complexity for inserting and deleting an element from the list is O(n).	In a doubly-linked list, the time complexity for inserting and deleting an element is O(1).

19. Write a recursive algorithm to find X^Y .

Ans :

Algorithm for calculating X to the Power of Y i.e X^Y :

```
Step 1:  Start
Step 2:  Declare variable pow and i.
Step 3:  Initialize pow= 1 and i= 1.
Step 4:  Read base X and power Y from user.
Step 5:  Repeat step until i is less than equal to Y i.e i< =Y
         5.1: Set, pow= pow * x
         5.2: Increment the value of i by 1
Step 6:  The value stored in pow is the required value.
Step 7:  Stop
```

20. Why linked list is called dynamic data structure? What are the advantages of using linked list over array.

Ans :

Dynamic Data Structure is that kind of data structure that changes its size during runtime. The values store in the data structure can be changed easily either it be static or dynamic data structure. But the dynamic data are designed in such a way that both the data and the size of the data structure can be easily changed at the runtime.

The main use case for which the Dynamic Data Structures are defined is to easily facilitate the change in the size of the data structure at the runtime without hindering the other operations that are associated with that data structure before increasing or decreasing the size of the data structure.

Some of the major examples of dynamic data structures are:

- Singly Linked List
- Doubly Linked List
- Vector
- Stack
- Queue
- Tree

Advantages of using linked list over arrays

The principal benefit of a linked list over a conventional array is that the list elements can be easily inserted or removed without reallocation or reorganization of the entire structure because the data items need not be stored contiguously in memory or on disk, while restructuring an array at run-time is a much more.

21. Describe execution of recursive calls with example.

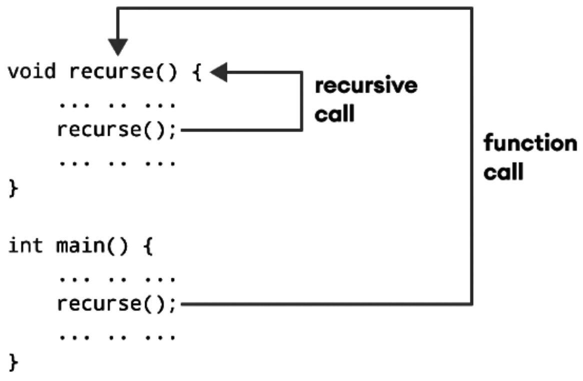
Ans :

A recursive call is a command within a subroutine or function that tells the program to run the same subroutine again. The repeat performance may be the direct result of the function, or a second function may be triggered that, in turn, refers back to the first function.

Working of Recursion in C++

```
voidrecurse()
{
    ... .. ...
recurse();
    ... .. ...
}

intmain()
{
    ... .. ...
recurse();
    ... .. ...
}
```

The figure below shows how recursion works by calling itself over and over again.



```
void recurse() {
    ... .. ...
    recurse();
    ... .. ...
}

int main() {
    ... .. ...
    recurse();
    ... .. ...
}
```

Choose the Correct Answers

1. A list of elements in which enqueue operation takes place from one end, and dequeue operation takes place from one end is ________. [c]

   (a) Binary tree (b) Stack

   (c) Queue (d) Linked list

2. Which of the following principle does Queue use? [b]

   (a) LIFO principle (b) FIFO principle

   (c) Linear tree (d) Ordered array

3. Which one of the following is not the type of the Queue? [d]

   (a) Linear Queue (b) Circular Queue

   (c) Double ended Queue (d) Single ended Queue

4. The time complexity of enqueue operation in Queue is ________. [a]

   (a) O(1) (b) O(n)

   (c) O(logn) (d) O(nlogn)

5. Which one of the following is the correct way to increment the rear end in a circular queue? [b]

   (a) rear = rear + 1 (b) (rear + 1) % max

   (c) (rear % max) + 1 (d) None of the above

6. In the linked list implementation of queue, where will the new element be inserted? [c]

   (a) At the middle position of the linked list (b) At the head position of the linked list

   (c) At the tail position of the linked list (d) None of the above

7. How many Queues are required to implement a Stack? [b]

   (a) 3 (b) 2

   (c) 1 (d) 4

8. Which one of the following is not the application of the Queue data structure? [d]

   (a) Resource shared between various systems (b) Data is transferred asynchronously

   (c) Load balancing (d) Balancing of symbols

9. Which of the following principle is used if two elements in the priority queue have the same priority? [b]

   (a) LIFO (b) FIFO

   (c) Linear tree (d) None of the above

10. A linear data structure in which insertion and deletion operations can be performed from both the ends is ________. [b]

    (a) Queue (b) Deque

    (c) Priority queue (d) Circular queue

Fill in the blanks

1. In the Deque implementation using singly linked list, ________ be the time complexity of deleting an element from the rear end.
2. The process in which a function calls itself is known as recursion and the corresponding function is called the ________.
3. When function calls itself, it is called ________.
4. ________ tests to see whether the queue is empty. It needs no parameters and returns a boolean value.
5. A circular queue is also known as ________.
6. The dequeue stands for ________.
7. The ________ supports only comparable elements, which means that the elements are either arranged in an ascending or descending order.
8. Linked List can be defined as collection of objects called ________ that are randomly stored in the memory.
9. The doubly linked list can be accessed in ________ directions.
10. ________ is a more complexed type of data structure in which a node contain pointers to its previous node as well as the next node.

ANSWERS

1. O(n)
2. recursive function
3. direct recursion
4. isEmpty()
5. Ring Buffer
6. Double Ended Queue
7. priority queue
8. nodes
9. both
10. Circular doubly linked list

One Mark Question & Answers

1. Circular Linked List

Ans :

Circular doubly linked list is a more complexed type of data structure in which a node contain pointers to its previous node as well as the next node.

2. Single Linked List

Ans :

The singly linked list is a data structure that contains two parts, i.e., one is the data part, and the other one is the address part, which contains the address of the next or the successor node. The address part in a node is also known as a pointer.

3. Linked List

Ans :

A linked list can also be defined as the collection of the nodes in which one node is connected to another node, and node consists of two parts, i.e., one is the data part and the second one is the address part.

4. What is recursion?

Ans :

The process in which a function calls itself is known as recursion and the corresponding function is called the recursive function

5. Priority Queue

Ans :

A priority queue is a special type of queue in which each element is associated with a priority and is served according to its priority. If elements with the same priority occur, they are served according to their order in the queue.

UNIT III

Trees: Introduction, Types of Trees, Binary Tree, Binary Tree Abstract Data Type, Realization of a Binary Tree, Insertion of a Node in Binary Tree, Binary Tree Traversal, Other Tree Operations, Binary Search Tree, Threaded Binary Tree, Applications of Binary Trees.
Searching and Sorting: Search Techniques-Linear Search, Binary Search, Sorting TechniquesSelection Sort, Bubble Sort, Insertion Sort, Merge Sort, Quick Sort, Comparison of All Sorting Methods, Search Trees: Symbol Table, Optimal Binary Search Tree, AVL Tree (Heightbalanced Tree).

3.1 Trees

3.1.1 Introduction

Q1. What is tree data structure? Write about the basic terminology of tree data structure.

Ans :

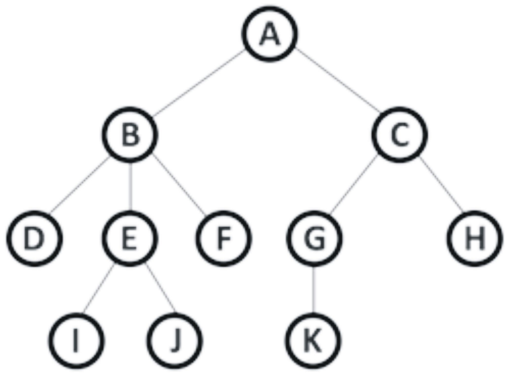
A tree data structure can also be defined as follows:

Tree data structure is a collection of data (Node) which is organized in hierarchical structure and this is a recursive definition

In tree data structure, every individual element is called as Node. Node in a tree data structure, stores the actual data of that particular element and link to next element in hierarchical structure.

In a tree data structure, if we have N number of nodes then we can have a maximum of N-1 number of links.

Example:



TREE with 11nodes and 10 edges

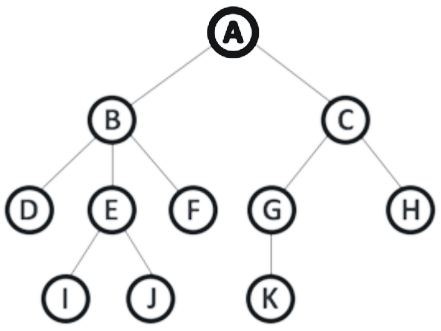
- In any tree with 'N'nodes there will be maximum of 'N-1' edges
- In a tree every individual element is called as 'NODE'

Terminology

In a tree data structure, we use the following terminology.

1. Root

In a tree data structure, the first node is called as Root Node. Every tree must have root node. We can say that root node is the origin of tree data structure. In any tree, there must be only one root node. We never have multiple root nodes in a tree.

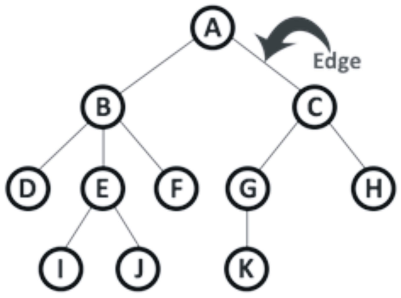


Here 'A' is the 'root' node

- In any tree the first node is called as ROOT node

2. Edge

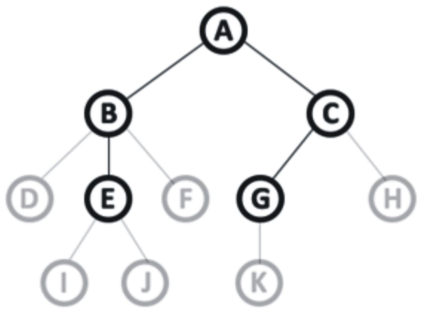
In a tree data structure, the connecting link between any two nodes is called as EDGE. In a tree with 'N' number of nodes there will be a maximum of 'N-1' number of edges.



- In any tree, **'Edge'** is a connecting link between two nodes.

3. Parent

In a tree data structure, the node which is predecessor of any node is called as PARENT NODE. In simple words, the node which has branch from it to any other node is called as parent node. Parent node can also be defined as "The node which has child / children".

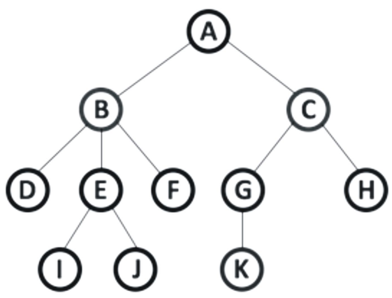


Here A, B, C, E & G are parent nodes

- In any tree the mode which has child / children is called **'Parent'**
- A node which is predecessor of any other node is called **'Parent'**

4. Child

In a tree data structure, the node which is descendant of any node is called as CHILD Node. In simple words, the node which has a link from its parent node is called as child node. In a tree, any parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes.



Here B & C are Children of A

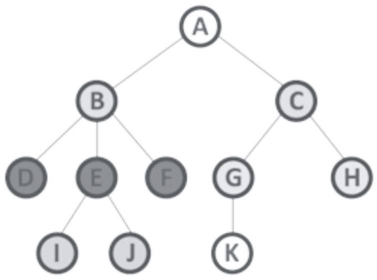
Here G & H are Children of C

Here K is Child of G

- descendant of any node is called as CHILD Node

5. Siblings

In a tree data structure, nodes which belong to same Parent are called as SIBLINGS. In simple words, the nodes with same parent are called as Sibling nodes.



Here B & C are Siblings

Here DE & F are Siblings

Here G & H are Siblings

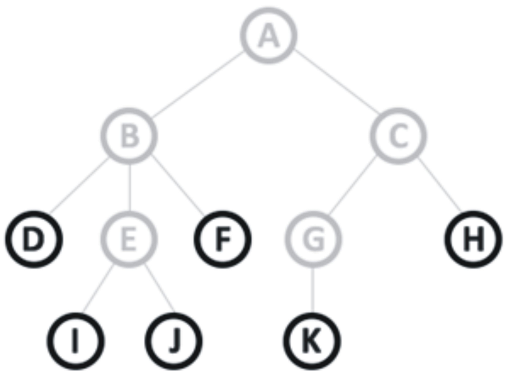
Here I & J are Siblings

- In any tree the nodes which has same Parent are called 'Siblings'
- The children of a Parent are called 'Siblings'

6. Leaf

In a tree data structure, the node which does not have a child is called as LEAF Node. In simple words, a leaf is a node with no child.

In a tree data structure, the leaf nodes are also called as External Nodes. External node is also a node with no child. In a tree, leaf node is also called as 'Terminal' node.



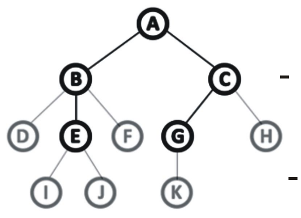
Here D, I, J, F, K & H are Leaf nodes'

- In any tree the node which does not have children is called 'Leaf'
- A node without successors is called 'leaf' node

7. Internal Nodes

In a tree data structure, the node which has atleast one child is called as INTERNAL Node. In simple words, an internal node is a node with atleast one child.

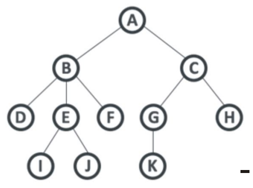
In a tree data structure, nodes other than leaf nodes are called as Internal Nodes. The root node is also said to be Internal Node if the tree has more than one node. Internal nodes are also called as 'Non-Terminal' nodes.



Here A, B, C, E & G are Internal nodes
- In any tree the node which has atleast one child is called **'Internal'** node
- Every non-leaf node is called as **'Internal'** node

8. Degree

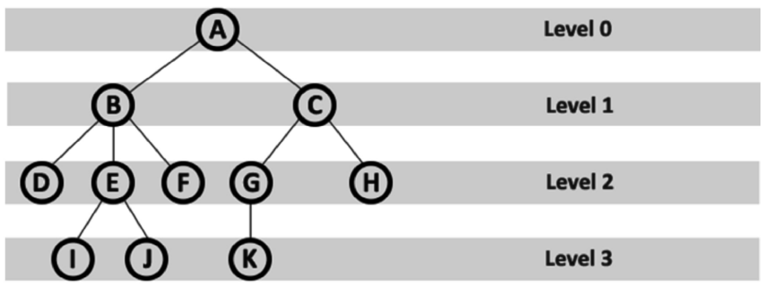
In a tree data structure, the total number of children of a node is called as DEGREE of that Node. In simple words, the Degree of a node is total number of children it has. The highest degree of a node among all the nodes in a tree is called as 'Degree of Tree'



Here Degree of B is 3
Here Degree of A is 2
Here Degree of F is 0
- In any tree, **'Degree'** a node is total number of children it has.

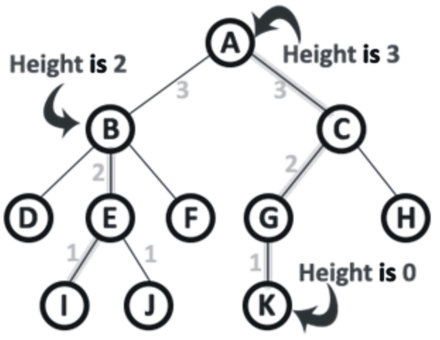
9. Level

In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on... In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step).



10. Height

In a tree data structure, the total number of egdes from leaf node to a particular node in the longest path is called as HEIGHT of that Node. In a tree, height of the root node is said to be height of the tree. In a tree, height of all leaf nodes is '0'.

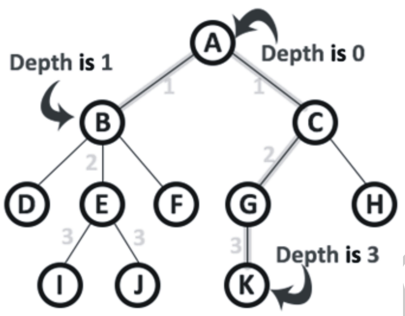


Here Height of tree is 3

- In any tree, **'Height of Node'** is total number of Edges from leaf to that node in longest path
- In any tree, **'Depth of Tree'** is total number of edges from root to leaf in the longest path.

11. Depth

In a tree data structure, the total number of egdes from root node to a particular node is called as DEPTH of that Node. In a tree, the total number of edges from root node to a leaf node in the longest path is said to be Depth of the tree. In simple words, the highest depth of any leaf node in a tree is said to be depth of that tree. In a tree, depth of the root node is '0'.

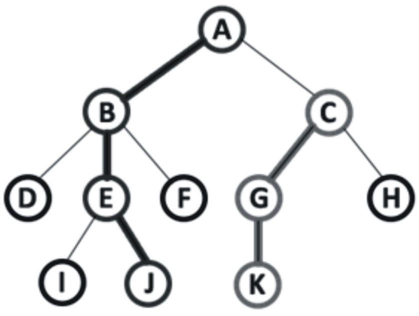


Here Depth of tree is 3

- In any tree, **'Depth of Node'** is total number of Edges from root to that node.
- In any tree, **'Depth of Tree'** is total number of edges from root to leaf in the longest path.

12. Path

In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as PATH between that two Nodes. Length of a Path is total number of nodes in that path. In below example the path A - B - E - J has length 4.



- In any tree, **'Path'** is a sequence of nodes and edges between two nodes.

Here, 'Path' between A & J is

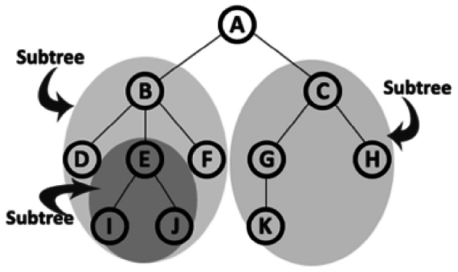
$\boxed{\text{A - B - E - J}}$

Here, 'Path' between C & K is

$\boxed{\text{C-G-K}}$

13. Sub Tree

In a tree data structure, each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node.



3.1.2 Types of Trees

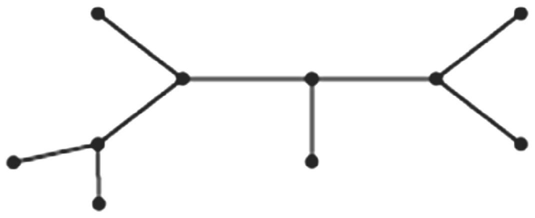
Q2. Write about various types of trees ?

Ans :

1. Free tree
2. Rooted tree
3. Ordered tree
4. Regular tree
5. Binary tree
6. Complete tree
7. Position tree

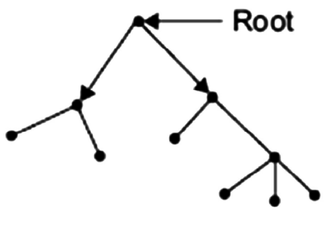
1. Free tree

A free tree is a connected, acyclic graph. It is an undirected graph. It has no node designated as a root. As it is connected, any node can be reached from any other node through a unique path. The following is an example of a free tree.



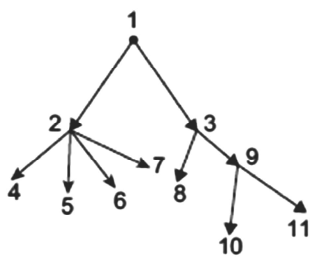
2. Rooted tree

Unlike free tree, a rooted tree is a directed graph where one node is designated as root, whose incoming degree is zero, whereas for all other nodes, the incoming degree is one.



3. Ordered tree

In many applications, the relative order of the nodes at any particular level assumes some significance. It is easy to impose an order on the nodes at a level by referring to a particular node as the first node, to another node as the second, and so on. Such ordering can be done from left to right . Just like nodes at each level, we can prescribe order to edges. If in a directed tree, an ordering of a node at each level is prescribed, then such a tree is called an ordered tree.



4. Regular tree

A tree where each branch node vertex has the same outdegree is called a regular tree. If in a directed tree, the outdegree of every node is less than or equal to m, then the tree is called an m-ary tree. If the outdegree of every node is exactly equal to m (the branch nodes) or zero (the leaf nodes), then the tree is called a regular m-ary tree.

5. Binary tree

A binary tree is a special form of an m-ary tree. Since a binary tree is important, it is frequently used in various applications of computer science.

We have defined an m-ary tree (general tree). A binary tree is an m-ary position tree

when m =2. In a binary tree, no node has more than two children.

6. Complete tree

A tree with n nodes and of depth k is complete if and only if its nodes correspond to the nodes that are numbered from 1 to n in the full tree of depth k.

A binary tree of height h is complete if and only if one of the following holds good:

1. It is empty.
2. Its left subtree is complete of height h> =1 and its right subtree is completely full of

   height h =2.
3. Its left subtree is completely full of height h > =1 and its right subtree is complete of

   height h =1.

   A binary tree is completely full if it is of height h and has (2h+1 -1) nodes.

Full binary tree : A binary tree is a full binary tree if it contains the maximum possible number of nodes in all levels.

In a full binary tree, each node has two children or no child at all. The total number of nodes in a full binary tree of heighth is 2h+1-1 considering the root at level 0.

It can be calculated by adding the number of nodes of each level as in the following equation:

20 <21 <22 <... <2h <2h+1 -1

Complete binary tree A binary tree is said to be a complete binary tree if all its levels except the last level have the maximum number of possible nodes, and all the nodes of the last level appear as far left as possible. In a complete binary tree, all the leaf nodes are at the last and the second last level, and the levels are filled from left to right.

Left skewed binary tree If the right subtree is missing in every node of a tree, we call it a left skewed tree . If the left subtree is missing in every node of a tree, we call it as right subtree.

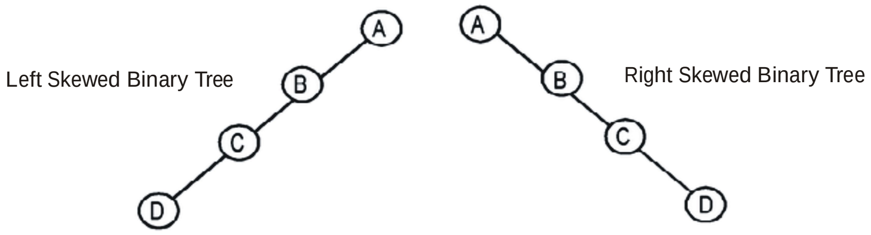


Fig.: Skewed Binary Tree

Strictly binary tree If every non-terminal node in a binary tree consists of non-empty left and right subtrees, then such a tree is called a strictly binary tree.

In Fig. the non-empty nodes D and E have left and right subtrees. Such expression trees are known as strictly binary trees.

Extended binary tree A binary tree T with each node having zero or two children is called an extended binary tree. The nodes with two children are called internal nodes, and those with zero children are called external nodes. Trees can be converted into extended trees by adding a node.

7. Position tree

A position tree, also known as a suffix tree, is one that represents the suffixes of a string S and such representation facilitates string operations being performed faster. Such a tree's edges are labelled with strings, such that each suffix of S corresponds to exactly one path from the tree's root to a leaf node. The space and time requirement is linear in the length of S. After its construction, several operations can be performed quickly, such as locating a substring in S, locating a substring if a certain number of mistakes are allowed, locating matches for a regular expression pattern, and so

3.1.3 Binary Tree

Q3. What is binary tree?

(OR)

Properties of binary tree.

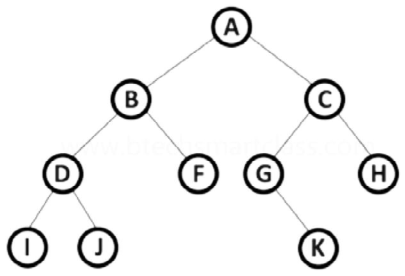
Ans : **(Nov.-2017)**

In a normal tree, every node can have any number of children. Binary tree is a special type of tree data structure in which every node can have a maximum of 2 children. One is known as left child and the other is known as right child.

A tree in which every node can have a maximum of two children is called as Binary Tree.

In a binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2 children.

Example



Q4. Write briefly about various types of binary trees?

Ans :

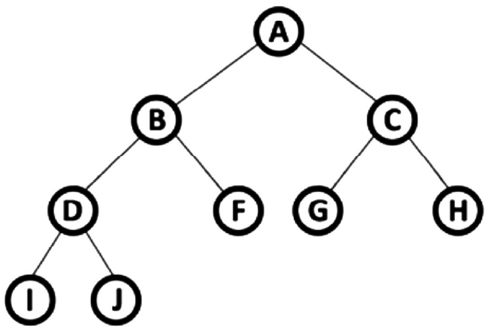
There are different types of binary trees and they are:

1. **Strictly Binary Tree**

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none. That means every internal node must have exactly two children. A strictly Binary Tree can be defined as follows:

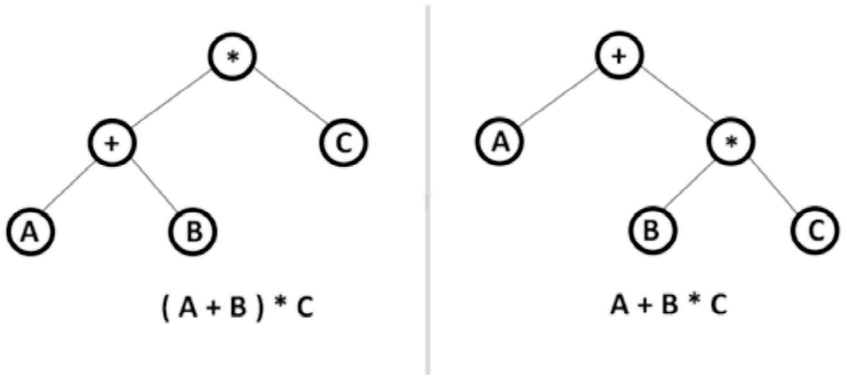
A binary tree in which every node has either two or zero number of children is called Strictly Binary Tree

Strictly binary tree is also called as Full Binary Tree or Proper Binary Tree or 2-Tree



Strictly binary tree data structure is used to represent mathematical expressions.

Example

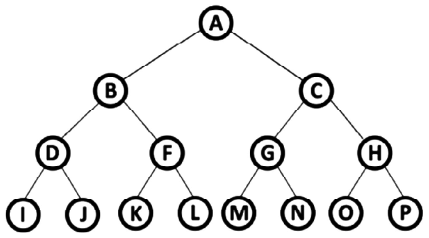


2. Complete Binary Tree

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none and in complete binary tree all the nodes must have exactly two children and at every level of complete binary tree there must be 2^{level} number of nodes. For example at level 2 there must be $2^2 = 4$ nodes and at level 3 there must be $2^3 = 8$ nodes.

A binary tree in which every internal node has exactly two children and all leaf nodes are at same level is called Complete Binary Tree.

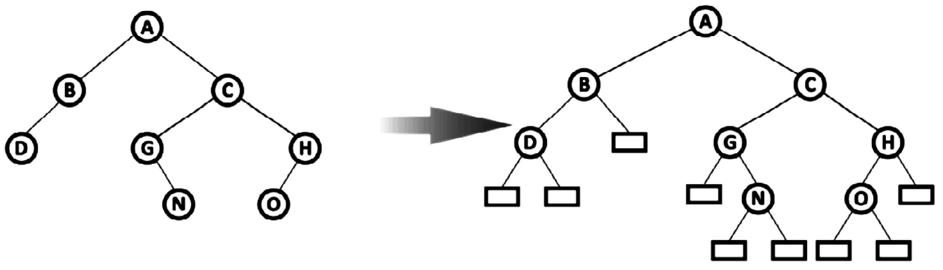
Complete binary tree is also called as Perfect Binary Tree



3. Extended Binary Tree

A binary tree can be converted into Full Binary tree by adding dummy nodes to existing nodes wherever required.

The full binary tree obtained by adding dummy nodes to a binary tree is called as Extended Binary Tree.



In above figure, a normal binary tree is converted into full binary tree by adding dummy nodes (In pink colour).

3.1.4 Binary Tree Abstract Data Type

Q5. Explain ADT of Binary Tree

Ans :

A binary tree consists of nodes with one predecessor and at most two successors (called the left child and the right child). The only exception is the root node of the tree that does not have a predecessor. A node can contain any amount and any type of data.

1. **create:** A binary tree may be created in several states. The most common include.
   - an empty tree (i.e. no nodes) and hence the constructor will have no parameters
   - a tree with one node (i.e. the root) and hence the constructor will have a single parameter (the data for the root node)

- a tree with a new root whose children are other existing trees and hence the constructor will have three parameters (the data for the root node and references to its subtrees)

2. **isEmpty():** Returns true if there are no nodes in the tree, false otherwise.
   - **isFull():** May be required by some implementations. Returns true if the tree is full, false otherwise.
3. **clear():** Removes all of the nodes from the tree (essentially reinitializing it to a new empty tree).
4. **add(value):** Adds a new node to the tree with the given value. The actual implementation of this method is determined by the purpose for the tree and how the tree is to be maintained and/or processed. To begin with, we will assume no particular purpose or order and therefore add new nodes in such a way that the tree will remain nearly balanced.
5. **remove():** Removes the root node of the tree and returns its data. The actual implementation of this method and other forms of removal will be determined by the purpose for the tree and how the tree is to be maintained and/or processed. For example, you might need a remove() method with a parameter that indicates which node of the tree is to be removed (based on position, key data, or reference). To begin with, we will assume no particular purpose or order and therefore remove the root in such a way that the tree will remain nearly balanced.

   Other operations that may be included as needed:

- **height():** Determines the height of the tree. An empty tree will have height 0.
- **size():** Determines the number of nodes in the tree.
- **getRootData():** Returns the data (primative data) or reference to the data (objects) of the tree's root.
- **getLeftSubtree():** Returns a reference to the left subtree of this tree.
- **getRightSubtree():** Returns a reference to the right subtree of this tree.

3.1.5 Realization of a Binary Tree

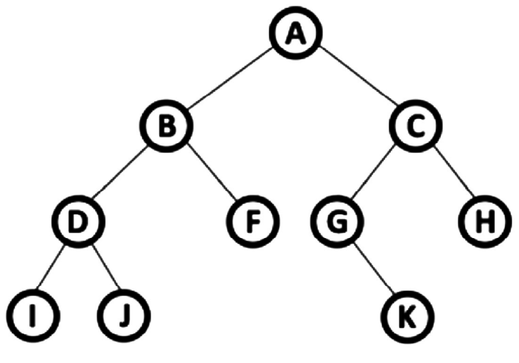
Q6. Explain various methods of representing a binary tree.

Ans :

A binary tree data structure is represented using two methods. Those methods are as follows:

1. Array Representation
2. Linked List Representation

Consider the following binary tree:



1. Array Representation

In array representation of binary tree, we use a one dimensional array (1-D Array) to represent a binary tree.

Consider the above example of binary tree and it is represented as follows:

A	B	C	D	F	G	H	I	J	–	–	–	K	–	–	–	–	–	–	–	–

To represent a binary tree of depth 'n' using array representation, we need one dimensional array with a maximum size of $2^{n+1} - 1$.

Advantages

The various merits of representing binary trees using arrays are as follows:

1. Any node can be accessed from any other node by calculating the index.
2. Here, the data is stored without any pointers to its successor or predecessor.
3. In the programming languages, where dynamic memory allocation is not possible (such as BASIC, fortran), array represen-tation is the only means to store a tree.

Disadvantages

The various demerits when representing binary trees using arrays are as follows:

1. Other than full binary trees, majority of the array entries may be empty.
2. It allows only static representation. The array size cannot be changed during the execution.
3. Inserting a new node to it or deleting a node from it is inefficient with this representation, because it requires considerable data movement up and down the array, which demand excessive amount of processing time.

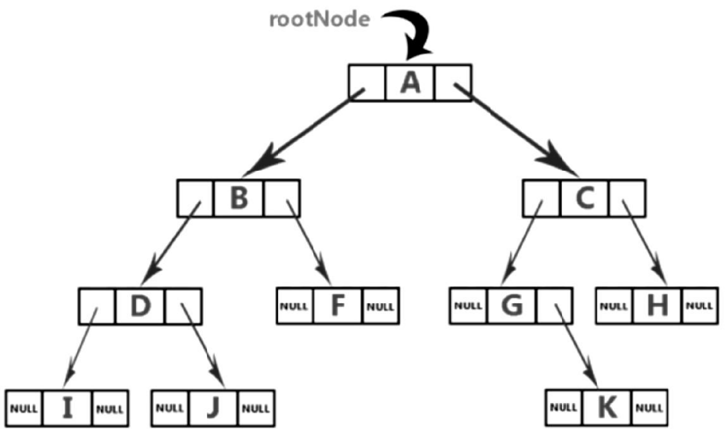
2. Linked List Representation

We use double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address.

In this linked list representation, a node has the following structure.

Left Child Address	**Date**	**Right Child** Address

The above example of binary tree represented using Linked list representation is shown as follows:



Q7. Write aC++ program to create a Complete Binary tree from its Linked List Representation.

Ans :

```
// C++ program to create a Complete Binary tree from its Linked ListRepresentation
#include <iostream>
#include <string>
#include <queue>
using namespace std;
// Linked list node
struct ListNode
{
        int data;
        ListNode* next;
};
// Binary tree node
structurestruct BinaryTreeNode
{
        int data;
        BinaryTreeNode *left, *right;};
// Function to insert a node at the beginning of the Linked List
void push(struct ListNode** head_ref, int new_data)
{
        // allocate node and assign data
        struct ListNode* new_node = new ListNode;
        new_node->data = new_data;
        // link the old list off the new node
        new_node->next = (*head_ref);
        // move the head to point to the new node
        (*head_ref)     = new_node;
}
// method to create a new binary tree node from the given data
BinaryTreeNode* newBinaryTreeNode(int data)
{
        BinaryTreeNode *temp = new BinaryTreeNode;
        temp->data = data;
        temp->left = temp->right = NULL;
        return temp;
}
        // converts a given linked list representing a complete binary tree into the
        // linked representation of binary tree.
        void convertList2Binary(ListNode *head, BinaryTreeNode* &root)
{
        // queue to store the parent nodes
        queue<BinaryTreeNode *> q;
        // Base Case
        if (head == NULL)
{
        root = NULL; // Note that root is passed by reference
        return;
}
        // 1.) The first node is always the root node, and add it to the queue
        root = newBinaryTreeNode(head->data);
        q.push(root);
        // advance the pointer to the next node
        head = head->next;
        // until the end of linked list is reached, do the following steps
        while (head)
{
        // a) take the parent node from the q and remove it from q
        BinaryTreeNode* parent = q.front();
        q.pop();
        // c) take next two nodes from the linked list. We will add
        // them as children of the current parent node in step 2.b. Push them
        // into the queue so that they will be parents to the future nodes
        BinaryTreeNode *leftChild = NULL, *rightChild = NULL;
        leftChild = newBinaryTreeNode(head->data);
        q.push(leftChild);
        head = head->next;
        if (head)
    {
        rightChild = newBinaryTreeNode(head->data);
        q.push(rightChild);
        head = head->next;
    }
        // b) assign the left and right children of parent
        parent->left = leftChild;
        parent->right = rightChild;
        }
    }
        // Utility function to traverse the binary tree after conversion
        void inorderTraversal(BinaryTreeNode* root)
    {     if (root)
        {
              inorderTraversal( root->left );
              cout << root->data << " ";
              inorderTraversal( root->right );
        }
    }
        // Driver program to test above functions
        int main()
        {
        // create a linked list shown in above diagram
        struct ListNode* head = NULL;
        push(&head, 36);  /* Last node of Linked List */
        push(&head, 30);
        push(&head, 25);
        push(&head, 15);
        push(&head, 12);
        push(&head, 10); /* First node of Linked List */
        BinaryTreeNode *root;
        convertList2Binary(head, root);
        cout << "Inorder Traversal of the constructed Binary Tree is: \n";
        inorderTraversal(root);
        return 0;
    }
    Run on IDE
```

Output:

```
    Inorder Traversal of the constructed Binary Tree is:
    25 12 30 10 36 15
```

3.1.6 Insertion of a Node in Binary Tree

Q8. How to insert a node in binary tree? Explain with an example.

Ans :

The insert() operation inserts a new node at any position in a binary tree. The node to be inserted could be a branch node or a leaf node. The branch node insertion is generally based on some criteria that are usually in the context of a special form of a binary tree.

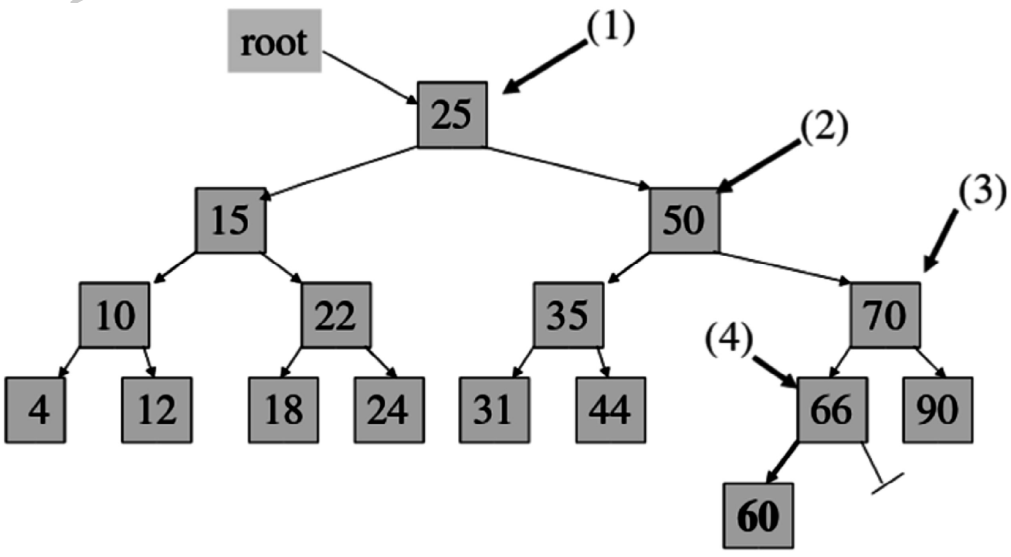
Let us study a commonly used case of inserting a node as a leaf node.

The insertion procedure is a two-step process.

1. Search for the node whose child node is to be inserted. This is a node at some level i, and a node is to be inserted at the level i >1 as either its left child or right child. This is the node after which the insertion is to be made.
2. Link a new node to the node that becomes its parent node, that is, either the Lchild or the Rchild.

Example: insert 60 in the tree:

1. start at the root, 60 is greater than 25, search in right subtree
2. 60 is greater than 50, search in 50's right subtree
3. 60 is less than 70, search in 70's left subtree
4. 60 is less than 66, add 60 as 66's left child



Q9. Write a C++ program to insert an element into binary tree

Ans

```
//C++ program to insert an element into binary tree
#include <stdio.h>
// tree node
structNode
{
    int data;
    Node *left, *right;
};

// returns a new tree Node
Node* newNode(int data)
{
    Node* temp = new Node();
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}
// A function to create binary tree.
Node* Tree(Node* temp, int data )
{
// If the tree is empty, return a new node
if (temp == NULL)
return newNode(data);
// Otherwise, recur down the tree
if (data < temp->data)
 temp->left = Tree(temp->left, data);
else
    temp->right = Tree(temp->right, data);
//return the (unchanged) node pointer
return temp;
}
//function to display all the element present in the binary tree
voiddisplay(struct Node* root)
{
if (!root)
return;
    display(root->left);
cout<<root->data<<" ";
    display(root->right);
}
//function to insert element in binary tree
voidinsert(struct Node* root , int value)
{
queue<struct Node*> q;
    q.push(root);
// Do level order traversal until we find an empty place.
while (!q.empty()) {
structNode* root = q.front();
    q.pop();
if (!root->left) {
    root->left = newNode(value);
break;
    } else
        q.push(root->left);
if (!root->right) {
    root->right = newNode(value);
break;
    } else
        q.push(root->right);
    }
}
intmain()
{
    char ch;
```

```
        int n, arr[20],size;
        Node *root = new Node;
        root = NULL;
        cout<<"Enter the size of array : ";
        cin>>size;
        cout<<"Enter the elements in array : ";
        for(int i=0;i<size;i++)
    {
        cin>>arr[i];
    }
        // Construct the binary tree.
        for(int i = 0; i < size; i++)
    {
        root = Tree(root, arr[i]);
    }
        cout<<"\nEnter the Element to be insert : ";
        cin>>n;
        insert(root,n);
        cout<<"\nElement Inserted"<<endl;
        cout<<"\nAfter Inserting "<<endl;
    cout<<"Elements are: ";
        display(root);
    cout<<endl;
    return0;
    }
```

3.1.7 Binary Tree Traversal

Q10. Explain about various traversal techniques of binary tree.

Ans : **(July-21, Dec.-19, June-19, Dec.-18, June-18)**

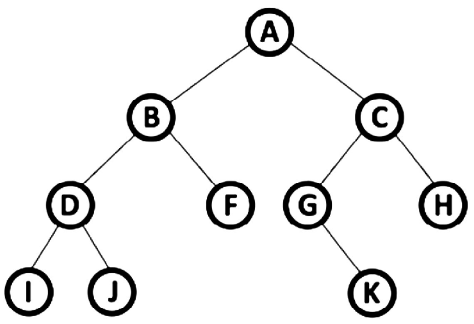
When we wanted to display a binary tree, we need to follow some order in which all the nodes of that binary tree must be displayed. In any binary tree displaying order of nodes depends on the traversal method.

Displaying (or) visiting order of nodes in a binary tree is called as Binary Tree Traversal.

There are three types of binary tree traversals.

1. In - Order Traversal
2. Pre - Order Traversal
3. Post - Order Traversal

Consider the following binary tree:



1. **In - Order Traversal (leftChild - root - rightChild)**

In In-Order traversal, the root node is visited between left child and right child. In this traversal, the left child node is visited first, then the root node is visited and later we go for visiting right child node. This in-order traversal is applicable for every root node of all subtrees in the tree. This is performed recursively for all nodes in the tree.

In the above example of binary tree, first we try to visit left child of root node 'A', but A's left child is a root node for left subtree. so we try to visit its (B's) left child 'D' and again D is a root for subtree with nodes D, I and J. So we try to visit its left child 'I' and it is the left most child. So first we visit 'I' then go for its root node 'D' and later we visit D's right child 'J'. With this we have completed the left part of node B. Then visit 'B' and next B's right child 'F' is visited. With this we have completed left part of node A. Then visit root node 'A'. With this we have completed left and root parts of node A. Then we go for right part of the node A. In right of A again there is a subtree with root C. So go for left child of C and again it is a subtree with root G. But G does not have left part so we visit 'G' and then visit G's right child K. With this we have completed the left part of node C. Then visit root node 'C' and next visit C's right child 'H' which is the right most child in the tree so we stop the process.

That means here we have visited in the order of I - D - J - B - F - A - G - K - C - H using In-Order Traversal.

In-Order Traversal for above example of binary tree is

I - D - J - B - F - A - G - K - C - H

2. **Pre - Order Traversal (root - leftChild - rightChild)**

In Pre-Order traversal, the root node is visited before left child and right child nodes. In this traversal, the root node is visited first, then its left child and later its right child. This pre-order traversal is applicable for every root node of all subtrees in the tree.

In the above example of binary tree, first we visit root node 'A' then visit its left child 'B' which is a root for D and F. So we visit B's left child 'D' and again D is a root for I and J. So we visit D's left child 'I' which is the left most child. So next we go for visiting D's right child 'J'. With this we have completed root, left and right parts of node D and root, left parts of node B. Next visit B's right child 'F'. With this we have completed root and left parts of node A. So we go for A's right child 'C' which is a root node for G and H. After visiting C, we go for its left child 'G' which is a root for node K. So next we visit left of G, but it does not have left child so we go for G's right child 'K'. With this we have completed node C's root and left parts. Next visit C's right child 'H' which is the right most child in the tree. So we stop the process.

That means here we have visited in the order of A-B-D-I-J-F-C-G-K-H using Pre-Order Traversal.

Pre-Order Traversal for above example binary tree is

A - B - D - I - J - F - C - G - K - H

2. Post - Order Traversal (leftChild - rightChild - root)

In Post-Order traversal, the root node is visited after left child and right child. In this traversal, left child node is visited first, then its right child and then its root node. This is recursively performed until the right most node is visited.

Here we have visited in the order of I - J - D - F - B - K - G - H - C - A using Post-Order Traversal.

Post-Order Traversal for above example binary tree is

I - J - D - F - B - K - G - H - C - A

Q11. Write a C++ program for creation and traversal of a Binary Tree

Ans : **(Dec.-19, July-19, June-18)**

```
#include<iostream.h>
#include<conio.h>
#include<process.h>
struct tree_node
{
        tree_node *left;
        tree_node *right;
        int data;
} ;
class bst
{
        tree_node *root;
        public:
        bst()
        {
                root=NULL;
        }
        int isempty()
        {
                return(root= =NULL);
        }
        void insert(int item);
        void inordertrav();
        void inorder(tree_node *);
        void postordertrav();
        void postorder(tree_node *);
        void preordertrav();
        void preorder(tree_node *);
};
void bst::insert(int item)
{
        tree_node *p=new tree_node;
        tree_node *parent;
        p->data=item;
        p->left=NULL;
        p->right=NULL;
        parent=NULL;
        if(isempty())
                root=p;
        else
        {
                tree_node *ptr;
                ptr=root;
                while(ptr!=NULL)
                {
                        parent=ptr;
                        if(item>ptr->data)
                                ptr=ptr->right;
                        else
                                ptr=ptr->left;
                }
                if(item<parent->data)
                        parent->left=p;
```

```
            else
                  parent->right=p;
        }
}
void bst::inordertrav()
{
        inorder(root);
}
void bst::inorder(tree_node *ptr)
{
        if(ptr!=NULL)
        {
              inorder(ptr->left);
              cout<<"  "<<ptr->data<<"     ";
              inorder(ptr->right);
        }
}
void bst::postordertrav()
{
        postorder(root);
}
void bst::postorder(tree_node *ptr)
{
        if(ptr!=NULL)
        {
              postorder(ptr->left);
              postorder(ptr->right);
              cout<<"  "<<ptr->data<<"     ";
        }
}
void bst::preordertrav()
{
        preorder(root);
}
void bst::preorder(tree_node *ptr)
{
        if(ptr!=NULL)
        {
              cout<<"  "<<ptr->data<<"     ";
              preorder(ptr->left);
              preorder(ptr->right);
        }
}
void main()
{
        bst b;
        b.insert(52);
        b.insert(25);
        b.insert(50);
        b.insert(15);
        b.insert(40);
        b.insert(45);
        b.insert(20); cout<<"inorder"<<endl;
        b.inordertrav();
        cout<<endl<<"postorder"<<endl;
        b.postordertrav();
        cout<<endl<<"preorder"<<endl;
        b.preordertrav();
        getch();
}
```

3.1.8 Other Tree Operations

Q12. Explain about other tree operations on a binary tree.

Ans :

Using traversal as a basic operation, many other operations can be performed on a tree, such as finding the height of the tree, computing the total number of nodes, leaf nodes, and so on. Let us study a few of such operations.

- **counting nodes** : CountNode() is the function that returns the total count of nodes in a linked binarytree.

```
int BinaryTree :: CountNode(TreeNode *Root)
{
if(Root == Null)
return 0;
else
return(1 + CountNode(Root->Rchild) + CountNode(Root->Lchild));
}
```

- **counting leaf nodes :** The CountLeaf() operation counts the total number of leaf nodes in a linked binary tree.Leaf nodes are those with no left or right children.

```
int BinaryTree :: CountLeaf(TreeNode *Root)
{
if(Root == Null)
return 0;
else if((Root->Rchild == Null) && (Root->Lchild == Null))
return(1);
else
return(CountLeaf(Root->Lchild) + CountLeaf(Root->Rchild));
}
```

- **Computing Height of Binary Tree :** The TreeHeight() operation computes the height of a linked binary tree. Height of atree is the maximum path length in the tree. We can get the path length by traversing thetree depthwise. Let us consider that an empty tree's height is 0 and the tree with only onenode has the height 1.

```
int BinaryTree :: TreeHeight(TreeNode *Root)
{
int heightL, heightR;
if(Root == Null)
return 0;
if(Root->Lchild == Null && Root->Rchild == Null)
return 0;
heightL = TreeHeight(Root->Lchild);
heightR = TreeHeight(Root->Rchild);
if(heightR > heightL)
return(heightR + 1);
return(heightL + 1);
}
```

- **Getting Mirror**, Replica, or Tree Interchange of Binary Tree : The Mirror() operation finds the mirror of the tree that will interchange all left and rightsubtrees in a linked binary tree.

```
void BinaryTree :: Mirror(TreeNode *Root)
{
```

```
TreeNode *Tmp;
if(Root != Null)
{
Tmp = Root->Lchild;
Root->Lchild = Root->Rchild;
Root->Rchild = Tmp;
Mirror(Root->Lchild);
Mirror(Root->Rchild);
}
}
```

- **Copying Binary Tree:** The TreeCopy() operation makes a copy of the linked binary tree. The function shouldallocate the necessary nodes and copy the respective contents into them.

```
TreeNode *BinaryTree :: TreeCopy()
{
TreeNode *Tmp;
if(Root == Null)
return Null;
Tmp = new TreeNode;
Tmp?Lchild = TreeCopy(Root?Lchild);
Tmp?Rchild = TreeCopy(Root?Rchild);
Tmp?Data = Root?Data;
return Tmp;
}
```

- **Equality Test :** The BTree_Equal() operation checks whether two binary trees are equal. Two trees aresaid to be equal if they have the same topology, and all the corresponding nodes are equal.The same topology refers to the fact that each branch in the first tree corresponds to abranch in the second tree in the same order and vice versa.

```
int BinaryTree :: BTree_Equal(Binarytree T1 , BinaryTree T2)
{
if(Root == Null && T2.Root == Null)
return 1;
return(Root && T2.Root);
&&(Root->Data == T2.Root->Data);
&&BTree_Equal(Root->Lchild ,T2.Root->Lchild);
&&BTree_Equal(Root->Rchild, T2.Root->Rchild));
}
```

3.1.9 Binary Search Tree

Q13. What is Binary Search Tree? How to represent it.

Ans : **(July-21, Nov.-17)**

A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties:

➢ The left sub-tree of a node has a key less than or equal to its parent node's key.

➢ The right sub-tree of a node has a key greater than or equal to its parent node's key.

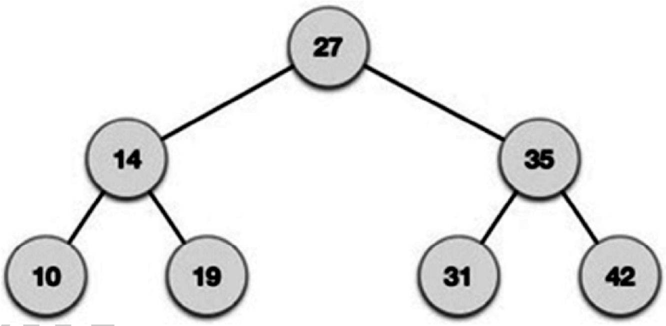
Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as:

left_subtree (keys) d" node (key) d" right_subtree (keys)

Representation of BST

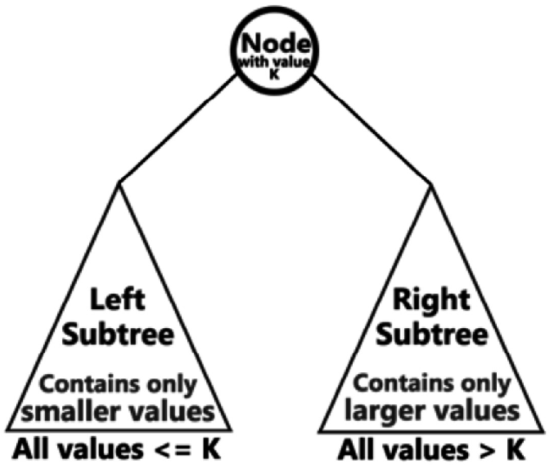
BST is a collection of nodes arranged in a way where they maintain BST properties. Each node has a key and an associated value. While searching, the desired key is compared to the keys in BST and if found, the associated value is retrieved.

Following is a pictorial representation of BST:



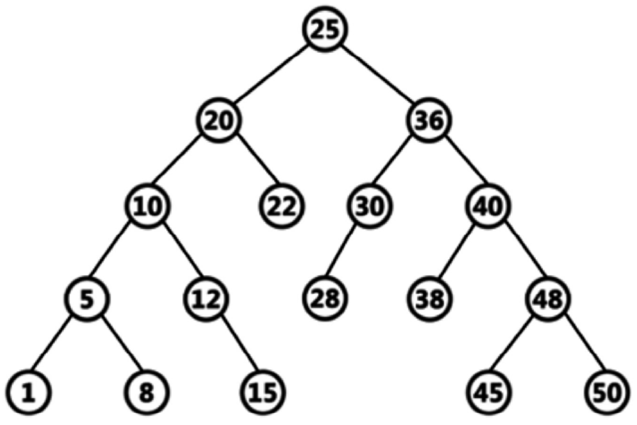
We observe that the root node key (27) has all less-valued keys on the left sub-tree and the higher valued keys on the right sub-tree.

In a binary search tree, all the nodes in left subtree of any node contains smaller values and all the nodes in right subtree of that contains larger values as shown in following figure.



Example

The following tree is a Binary Search Tree. In this tree, left subtree of every node contains nodes with smaller values and right subtree of every node contains larger values.



Every Binary Search Tree is a binary tree but all the Binary Trees need not to be binary search trees.

Q14. Explain the operations that can be performed on BSTs.

Ans : **(Nov.-17)**

Operations on a Binary Search Tree

The following operations are performed on a binary search tree...

1. Search
2. Insertion
3. Deletion

1. Search Operation in BST

In a binary search tree, the search operation is performed with O(log n) time complexity. The search operation is performed as follows...

- **Step 1:** Read the search element from the user
- **Step 2:** Compare, the search element with the value of root node in the tree.
- **Step 3:** If both are matching, then display "Given node found!!!" and terminate the function
- **Step 4:** If both are not matching, then check whether search element is smaller or larger than that node value.
- **Step 5:** If search element is smaller, then continue the search process in left subtree.
- **Step 6:** If search element is larger, then continue the search process in right subtree.
- **Step 7:** Repeat the same until we found exact element or we completed with a leaf node
- **Step 8:** If we reach to the node with search value, then display "Element is found" and terminate the function.
- **Step 9:** If we reach to a leaf node and it is also not matching, then display "Element not found" and terminate the function.

Algorithm

```
struct node* search(int data){
struct node *current = root;
        printf("Visiting elements: ");
while(current->data != data){
if(current != NULL){
        printf("%d ",current->data);
//go to left tree
if(current->data > data){
        current = current->leftChild;
}//else go to right tree
else{
        current = current->rightChild;
}
//not found
if(current == NULL){
return NULL;
}}}
return current;
}
```

2. Insertion Operation in BST

In a binary search tree, the insertion operation is performed with O(log n) time complexity. In binary search tree, new node is always inserted as a leaf node. The insertion operation is performed as follows:

- **Step 1:** Create a newNode with given value and set its left and right to NULL.
- **Step 2:** Check whether tree is Empty.
- **Step 3:** If the tree is Empty, then set set root to newNode.
- **Step 4:** If the tree is Not Empty, then check whether value of newNode is smaller or larger than the node (here it is root node).
- **Step 5:** If newNode is smaller than or equal to the node, then move to its left child. If newNode is larger than the node, then move to its right child.
- **Step 6:** Repeat the above step until we reach to a leaf node (e.i., reach to NULL).
- **Step 7:** After reaching a leaf node, then isert the newNode as left child if newNode is smaller or equal to that leaf else insert it as right child.

```
void insert(int data) {
struct node *tempNode = (struct node*) malloc(sizeof(struct node));
```

```
          struct node *current;
          struct node *parent;
          tempNode->data = data;
          tempNode->leftChild = NULL;
          tempNode->rightChild = NULL;
     //if tree is empty
     if(root == NULL) {
     root = tempNode;
     } else {
     current = root;
     parent = NULL;
     while(1) {
     parent = current;
     //go to left of the tree
     if(data < parent->data) {
          current = current->leftChild;
          //insert to the left
     if(current == NULL) {
          parent->leftChild = tempNode;
          return;
       }
     }//go to right of the tree
     else {
          current = current->rightChild;
          //insert to the right
          if(current == NULL) {
          parent->rightChild = tempNode;
          return;
       }       }
   }
 }}
```

3. Deletion Operation in BST

In a binary search tree, the deletion operation is performed with O(log n) time complexity. Deleting a node from Binary search tree has follwing three cases...

- Case 1: Deleting a Leaf node (A node with no children)
- Case 2: Deleting a node with one child
- Case 3: Deleting a node with two children

Case 1: Deleting a leaf node

We use the following steps to delete a leaf node from BST

- Step 1: Find the node to be deleted using search operation
- Step 2: Delete the node using free function (If it is a leaf) and terminate the function.

Case 2: Deleting a node with one child

We use the following steps to delete a node with one child from BST...

- **Step 1:** Find the node to be deleted using search operation
- **Step 2:** If it has only one child, then create a link between its parent and child nodes.
- **Step 3:** Delete the node using free function and terminate the function.

Case 3: Deleting a node with two children

We use the following steps to delete a node with two children from BST...

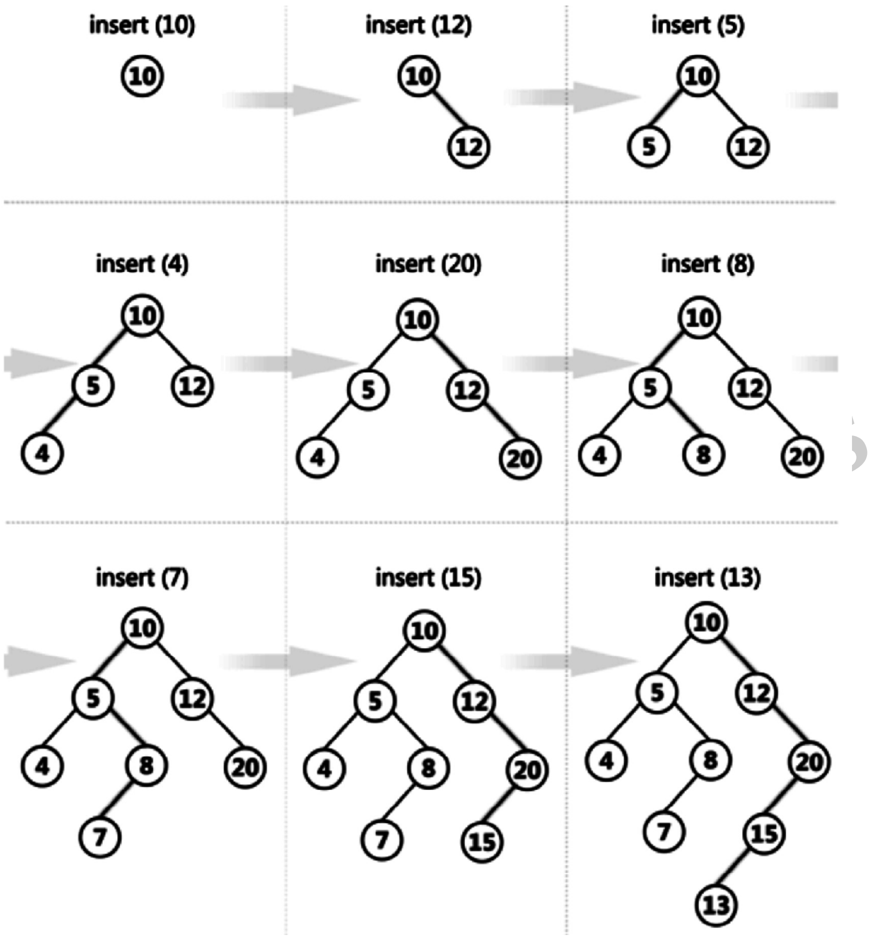
- **Step 1:** Find the node to be deleted using search operation
- **Step 2:** If it has two children, then find the largest node in its left subtree (OR) the smallest node in its right subtree.
- **Step 3:** Swap both deleting node and node which found in above step.
- **Step 4:** Then, check whether deleting node came to case 1 or case 2 else goto steps 2
- **Step 5:** If it comes to case 1, then delete using case 1 logic.
- **Step 6:** If it comes to case 2, then delete using case 2 logic.
- **Step 7:** Repeat the same process until node is deleted from the tree.

Example

Construct a Binary Search Tree by inserting the following sequence of numbers.

10,12,5,4,20,8,7,15 and 13

Above elements are inserted into a Binary Search Tree as follows.



Q15. Write a program to implement BST.

Ans :

```
/*  * C++ Program To Implement BST  */
# include <iostream>
# include <cstdlib>
using namespace std;
/*  * Node Declaration  */
struct node
{
      int info;
```

```
        struct node *left;
        struct node *right;
}*root;
/*  * Class Declaration  */
class BST
{
        public:
        void find(int, node **, node **);
        void insert(int);
        void del(int);
        void case_a(node *,node *);
        void case_b(node *,node *);
        void case_c(node *,node *);
        void preorder(node *);
        void inorder(node *);
        void postorder(node *);
        void display(node *, int);
        BST()
{
        root = NULL;
}
};
/*  * Main Contains Menu  */
int main()
{
        int choice, num;
        BST bst;
        node *temp;
        while (1)
{
        cout<<"————————————"<<endl;
        cout<<"Operations on BST"<<endl;
        cout<<"————————————"<<endl;
        cout<<"1.Insert Element "<<endl;
        cout<<"2.Delete Element "<<endl;
        cout<<"3.Inorder Traversal"<<endl;
        cout<<"4.Preorder Traversal"<<endl;
        cout<<"5.Postorder Traversal"<<endl;
        cout<<"6.Display"<<endl;
        cout<<"7.Quit"<<endl;
        cout<<"Enter your choice : ";
```

```
          cin>>choice;
          switch(choice)
    {
    case 1:
          temp = new node;
          cout<<"Enter the number to be inserted : ";
               cin>>temp->info;
          bst.insert(root, temp);
     case 2:
          if (root == NULL)
     {
          cout<<"Tree is empty, nothing to delete"<<endl;
          continue;
     }
          cout<<"Enter the number to be deleted : ";
          cin>>num;
          bst.del(num);
          break;
     case 3:
       cout<<"Inorder Traversal of BST:"<<endl;
       bst.inorder(root);
       cout<<endl;
       break;
     case 4:
          cout<<"Preorder Traversal of BST:"<<endl;
          bst.preorder(root);
          cout<<endl;
          break;
     case 5:
     cout<<"Postorder Traversal of BST:"<<endl;
     bst.postorder(root);
     cout<<endl;
     break;
    case 6:
     cout<<"Display BST:"<<endl;
     bst.display(root,1);
     cout<<endl;
     break;
    case 7:
     exit(1);
     default:
        cout<<"Wrong choice"<<endl;
       }
   }
}
        /* * Find Element in the Tree */
        void BST::find(int item, node **par, node
        **loc)
        {
              node *ptr, *ptrsave;
              if (root == NULL)
        {
              *loc = NULL;
              *par = NULL;
              return;
        }
              if (item == root->info)
        {
              *loc = root;
              *par = NULL;
              return;
        }
              if (item < root->info)
              ptr = root->left;
              else
              ptr = root->right;
              ptrsave = root;
              while (ptr != NULL)
        {
              if (item == ptr->info)
        {
              *loc = ptr;
              *par = ptrsave;
              return;
        }
              ptrsave = ptr;
              if (item < ptr->info)
                     ptr = ptr->left;
                            else
                                   ptr = ptr->right;
        }
        *loc = NULL;
        *par = ptrsave;
}
        /** Inserting Element into the Tree */
        void BST::insert(node *tree, node *newnode)
        {
        if (root == NULL)
        {
              root = new node;
              root->info = newnode->info;
              root->left = NULL;
              root->right = NULL;
              cout<<"Root Node is Added"<<endl;
              return;
        }
        if (tree->info == newnode->info)
        {
              cout<<"Element already in the
        tree"<<endl;
              return;
        }
        if (tree->info > newnode->info)
        {
              if (tree->left != NULL)
              {
              insert(tree->left, newnode);
                     }
                     else
                     {
              tree->left = newnode;
              (tree->left)->left = NULL;
              (tree->left)->right = NULL;
              cout<<"Node Added To Left"<<endl;
              return;
              }
        }
        else
        {
              if (tree->right != NULL)
              {
              insert(tree->right, newnode);
       }
```

```
    else
    {
          tree->right = newnode;
          (tree->right)->left = NULL;
          (tree->right)->right = NULL;
       cout<<"Node Added To Right"<<endl;
       return;
     }
 }}
     /* * Delete Element from the tree */
     void BST::del(int item)
     {
          node *parent, *location;
          if (root == NULL)
     {
          cout<<"Tree empty"<<endl;
          return;
     }
     find(item, &parent, &location);
     if (location == NULL)
     {
          cout<<"Item not present in tree"<<endl;
          return;
     }
     if (location->left == NULL && location->right == NULL)
       case_a(parent, location);
     if (location->left != NULL && location->right == NULL)
       case_b(parent, location);
     if (location->left == NULL && location->right != NULL)
       case_b(parent, location);
     if (location->left != NULL && location->right != NULL)
       case_c(parent, location);
      free(location);
     }
     /* * Case A */
     void BST::case_a(node *par, node *loc )
     {
          if (par == NULL)
          {
                root = NULL;
```

```
        }
        else
        {
              if (loc == par->left)
                     par->left = NULL;
              else
          par->right = NULL;
        }
}
        /*  * Case B  */
        void BST::case_b(node *par, node *loc)
        {
        node *child;
        if (loc->left != NULL)
              child = loc->left;
        else
        child = loc->right;
        if (par == NULL)
        {
              root = child;
        }
        else
        {
              if (loc == par->left)
                     par->left = child;
              else
                     par->right = child;
        }
}
/*  * Case C  */
void BST::case_c(node *par, node *loc)
        {
        node *ptr, *ptrsave, *suc, *parsuc;
        ptrsave = loc;
        ptr = loc->right;
        while (ptr->left != NULL)
        {
              ptrsave = ptr;
              ptr = ptr->left;
        }
        suc = ptr;
        parsuc = ptrsave;
        if (suc->left == NULL && suc->right
                     == NULL)
              case_a(parsuc, suc);
        else
              case_b(parsuc, suc);
        if (par == NULL)
        {
              root = suc;
        }
        else
        {
              if (loc == par->left)
                     par->left = suc;
              else
                     par->right = suc;
        }
        suc->left = loc->left;
        suc->right = loc->right;
        }
        /*  * Pre Order Traversal  */
        void BST::preorder(node *ptr)
        {
              if (root == NULL)
              {
              cout<<"Tree is empty"<<endl;
              return;
        }
        if (ptr != NULL)
        {
              cout<<ptr->info<<"  ";
              preorder(ptr->left);
              preorder(ptr->right);
        }
        }
        /*  * In Order Traversal  */

        void BST::inorder(node *ptr)
        {
              if (root == NULL)
              {
                    cout<<"Tree is empty"<<endl;
                    return;
            }
            if (ptr != NULL)
            {
                inorder(ptr->left);
                cout<<ptr->info<<"  ";
                inorder(ptr->right);
            }
}
        /* * Postorder Traversal */
        void BST::postorder(node *ptr)
        {
                if (root == NULL)
                {
                    cout<<"Tree is empty"<<endl;
                    return;
            }
            if (ptr != NULL)
            {
                postorder(ptr->left);
                postorder(ptr->right);
                cout<<ptr->info<<"  ";
            }
}
        /* * Display Tree Structure */
        void BST::display(node *ptr, int level)
        {
                int i;
                if (ptr != NULL)
                {
                display(ptr->right, level+1);
                cout<<endl;
                if (ptr == root)
                cout<<"Root->:  ";
        else
        {
                for (i = 0;i < level;i++)
                cout<<"       ";
                    }
                cout<<ptr->info;
                display(ptr->left, level+1);
                }
}
```

OUTPUT

```
        ________________
        Operations on BST
        ________________
        1.   Insert Element
        2.   Delete Element
        3.   Inorder Traversal
        4.   Preorder Traversal
        5.   Postorder Traversal
        6.   Display
        7.   Quit
        Enter your choice : 1
        Enter the number to be inserted : 8
        Root Node is Added
        ________________
        Operations on BST
        ________________
        1.   Insert Element
        2.   Delete Element
        3.   Inorder Traversal
        4.   Preorder Traversal
        5.   Postorder Traversal
        6.   Display
        7.   Quit
        Enter your choice : 6
        Display BST:
        Root->:  8
        ________________
        Operations on BST
        ________________
        1.   Insert Element
        2.   Delete Element
        3.   Inorder Traversal
        4.   Preorder Traversal
```

```
5.    Postorder Traversal
6.    Display
7.    Quit
Enter your choice : 1
Enter the number to be inserted : 9
Node Added To Right
_______________
Operations on BST
_______________
1.    Insert Element
2.    Delete Element
3.    Inorder Traversal
4.    Preorder Traversal
5.    Postorder Traversal
6.    Display
7.    Quit
Enter your choice : 6
Display BST : 9
Root->:  8
```

3.1.10 Threaded Binary Tree

Q16. What is threaded binary tree? Explain the representation of threaded binary tree.

Ans :

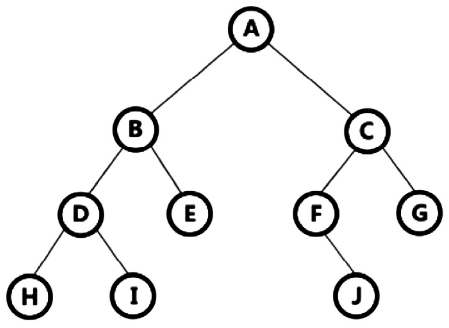
A binary tree is represented using array representation or linked list representation. When a binary tree is represented using linked list representation, if any node is not having a child we use NULL pointer in that position. In any binary tree linked list representation, there are more number of NULL pointer than actual pointers. Generally, in any binary tree linked list representation, if there are 2N number of reference fields, then N+1 number of reference fields are filled with NULL (N+1 are NULL out of 2N). This NULL pointer does not play any role except indicating there is no link (no child).

A binary tree called "Threaded Binary Tree", which make use of NULL pointer to improve its traversal processes. In threaded binary tree, NULL pointers are replaced by references to other nodes in the tree, called threads.

Threaded Binary Tree is also a binary tree in which all left child pointers that are NULL (in Linked list representation) points to its in-order predecessor, and all right child pointers that are NULL in Linked list representation) points to its in-order successor.

If there is no in-order predecessor or in-order successor, then it point to root node.

Consider the following binary tree:



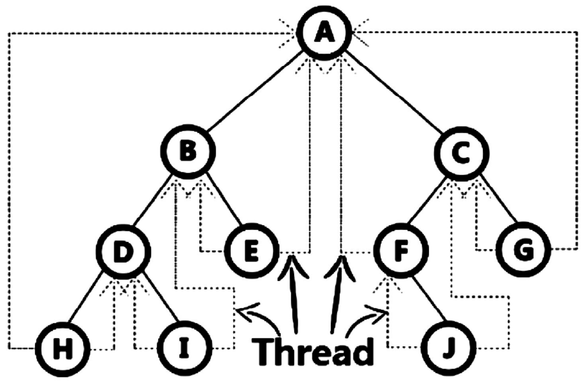
To convert above binary tree into threaded binary tree, first find the in-order traversal of that tree.

In-order traversal of above binary tree:

H - D - I - B - E - A - F - J - C - G

When we represent above binary tree using linked list representation, nodes H, I, E, F, J and G left child pointers are NULL. This NULL is replaced by address of its in-order predecessor, respectively (I to D, E to B, F to A, J to F and G to C), but here the node H does not have its in-order predecessor, so it points to the root node A. And nodes H, I, E, J and G right child pointers are NULL. This NULL ponters are replaced by address of its in-order successor, respectively (H to D, I to B, E to A, and J to C), but here the node G does not have its in-order successor, so it points to the root node A.

Above example binary tree become as follows after converting into threaded binary tree.



In above figure threads are indicated with dotted links.

Q17. Write a program to implement threaded Binary tree.

Ans :

```
//* C++ Program to Implement Threaded
Binary Tree
#include <iostream>
#include <cstdlib>
#define MAX_VALUE 65536
usingnamespace std;
/* Class Node */
class Node
{
      public:
int key;
      Node *left, *right;
bool leftThread, rightThread;
};
/* Class ThreadedBinarySearchTree */
class ThreadedBinarySearchTree
{
      private:
Node *root;
public:
/* Constructor */
ThreadedBinarySearchTree()
{
      root =new Node();
      root->right = root->left = root;
      root->leftThread =true;
      root->key = MAX_VALUE;
}
/* Function to clear tree */
void makeEmpty()
{
      root =new Node();
      root->right = root->left = root;
      root->leftThread =true;
      root->key = MAX_VALUE;
}
/* Function to insert a key */
void insert(int key)
{
      Node *p = root;
for(;;)
{
if(p->key < key)
{
if(p->rightThread)
break;
      p = p->right;
}
elseif(p->key > key)
{
if(p->leftThread)
break;
        p = p->left;
}
else
{
/* redundant key */
return;
```

```
}
}
      Node *tmp =new Node();
      tmp->key = key;
      tmp->rightThread = tmp->left
      Thread =true;
if(p->key < key)
{
/* insert to right side */
      tmp->right = p->right;
      tmp->left = p;
      p->right = tmp;
      p->rightThread =false;
}
else
{
      tmp->right = p;
      tmp->left = p->left;
      p->left = tmp;
      p->leftThread =false;
}
}
/* Function to search for an element */
bool search(int key)
{
      Node *tmp = root->left;
for(;;)
{
if(tmp->key < key)
{
if(tmp->rightThread)
returnfalse;
      tmp = tmp->right;
}
elseif(tmp->key > key)
{
if(tmp->leftThread)
returnfalse;
      tmp = tmp->left;
}
else
{
returntrue;
}
}}
/* Fuction to delete an element */
void Delete(int key)
{
      Node *dest = root->left, *p = root;
for(;;)
{
if(dest->key < key)
{
/* not found */
if(dest->rightThread)
return;
      p = dest;
      dest = dest->right;
}
elseif(dest->key > key)
{
/* not found */
if(dest->leftThread)
return;
        p = dest;
        dest = dest->left;
}
else
{
/* found */
```

```
break;
}
}
        Node *target = dest;
if(!dest->rightThread &&!dest->leftThread)
{
/* dest has two children*/
        p = dest;
/* find largest node at left child */
        target = dest->left;
while(!target->rightThread)
{
        p = target;
        target = target->right;
}
/* using replace mode*/
        dest->key = target->key;
}
if(p->key >= target->key)
{
if(target->rightThread && target->left
Thread)
{
        p->left = target->left;
        p->leftThread =true;
}
elseif(target->rightThread)
{
        Node *largest = target->left;
while(!largest->rightThread)
{
        largest = largest->right;
}
        largest->right = p;
        p->left = target->left;
}
else
{
        Node *smallest = target->right;
while(!smallest->leftThread)
{
        smallest = smallest->left;
}
        smallest->left = target->left;
        p->left = target->right;
}
}
else
{
if(target->rightThread && target->left
Thread)
{
        p->right = target->right;
        p->rightThread =true;
}
elseif(target->rightThread)
{
        Node *largest = target->left;
while(!largest->rightThread)
{
        largest = largest->right;
}
        largest->right =  target->right;
        p->right = target->left;
}
else
{
        Node *smallest = target->right;
while(!smallest->leftThread)
{
        smallest = smallest->left;
}
```

```
        smallest->left = p;
        p->right = target->right;
}
}
}
/* Function to print tree */
void printTree()
{
Node *tmp = root, *p;
for(;;)
{
        p = tmp;
        tmp = tmp->right;
if(!p->rightThread)
{
while(!tmp->leftThread)
{
        tmp = tmp->left;
}
}
if(tmp == root)
break;
cout<<tmp->key<<"   ";
}
cout<<endl;
}
};
/* Main Contains Menu */
int main()
{
ThreadedBinarySearchTree tbst;
cout<<"ThreadedBinarySearchTree Test\n";
char ch;
int choice, val;
/*  Perform tree operations  */
do
{
cout<<"\nThreadedBinarySearchTree
Operations\n";
cout<<"1. Insert "<<endl;
cout<<"2. Delete"<<endl;
cout<<"3. Search"<<endl;
cout<<"4. Clear"<<endl;
cout<<"Enter Your Choice: ";
cin>>choice;
switch(choice)
{
case1:
cout<<"Enter integer element to insert: ";
cin>>val;
        tbst.insert(val);
break;
case2:
cout<<"Enter integer element to delete: ";
cin>>val;
        tbst.Delete(val);
break;
case3:
cout<<"Enter integer element to search: ";
cin>>val;
if(tbst.search(val)==true)
cout<<"Element "<<val<<" found in the
tree"<<endl;
else
cout<<"Element "<<val<<" not found in
the tree"<<endl;
break;
case4:
cout<<"\nTree Cleared\n";
        tbst.makeEmpty();
```

```
break;
default:
cout<<"Wrong Entry \n ";
break;
}
/*  Display tree  */
cout<<"\nTree = ";
tbst.printTree();
cout<<"\nDo you want to continue (Type y or n): ";
cin>>ch;
}
while(ch =='Y'|| ch =='y');
return0;
}
```

OUTPUT

```
ThreadedBinarySearchTree Test
ThreadedBinarySearchTree Operations
1. Insert
2. Delete
3. Search
4. Clear
Enter Your Choice: 1
Enter integer element to insert: 28
Tree = 28
Do you want to continue(Type y or n): y
ThreadedBinarySearchTree Operations
1. Insert
2. Delete
3. Search
4. Clear
Enter Your Choice: 1
Enter integer element to insert: 5
Tree = 528
```

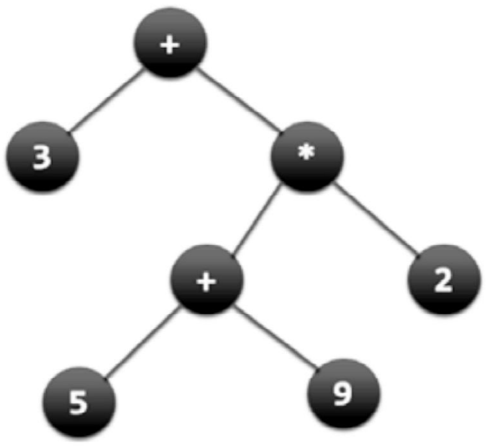
3.1.11 Applications of Binary Trees.

Q18. What is expression tree? Write about the construction of expression tree.

Ans :

Expression Tree

Expression tree is a binary tree in which each internal node corresponds to operator and each leaf node corresponds to operand so for example expression tree for 3 + ((5+9)*2) would be:



Inorder traversal of expression tree produces infix version of given postfix expression (same with preorder traversal it gives prefix expression)

Evaluating the expression represented by expression tree:

```
Let t be the expression tree
If  t is not null then
If t.value is operand then
      Return  t.value
A = solve(t.left)
 B = solve(t.right)
 // calculate applies operator 't.value'
// on A and B, and returns value
Return calculate(A, B, t.value)
```

Construction of Expression Tree:

Now For constructing expression tree we use a stack. We loop through input expression and do following for every character.

1. If character is operand push that into stack
2. If character is operator pop two values from stack make them its child and push current node again.

At the end only element of stack will be root of expression tree.

Q19. What is decision tree? Write about it.

Ans :

Decision tree

A decision tree is a flowchart-like structure in which each internal node represents a “test” on an attribute (e.g. whether a coin flip comes up heads or tails), each branch represents the outcome of the test and each leaf node represents a class label (decision taken after computing all attributes). The paths from root to leaf represents classification rules.

In decision analysis a decision tree and the closely related influence diagram are used as a visual and analytical decision support tool, where the expected values (or expected utility) of competing alternatives are calculated.

A decision tree consists of 3 types of nodes:

1. Decision nodes - commonly represented by squares
2. Chance nodes - represented by circles
3. End nodes - represented by triangles

Decision trees are commonly used in operations research and operations management. If in practice decisions have to be taken online with no recall under incomplete knowledge, a decision tree should be paralleled by a probability model as a best choice model or online selection model algorithm. Another use of decision trees is as a descriptive means for calculating conditional probabilities.

Decision trees, influence diagrams, utility functions, and other decision analysis tools and methods are taught to undergraduate students in schools of business, health economics, and public health, and are examples of operations research or management science methods.

3.2 Searching and Sorting

3.2.1 Search Techniques

Q20. What is Search? What are the basic searching techniques?

Ans :

Search is a process of finding a value in a list of values. In other words, searching is the process of locating given value position in a list of values.

Searching can be done on internal data structures or on external data structures. Information retrieval in the required format is the central activity in all computer applications.

Basic Searching Techniques

Consider a list of n elements or can represent a file of n records, where each element is a key number. The task is to find a particular key in the list in the shortest possible time. If you know you are going to search for an item in a set, you will need to think carefully about what type of data structure you will use for that set. At low level, the only searches that get mentioned are for sorted and unsorted arrays. However, these are not the only data types that are useful for searching.

1. **Linear search**

   Start at the beginning of the list and check every element of the list. Very slow [order O(n)] but works on an unsorted list.

2. **Binary Search**

   This is used for searching in a sorted array. Test the middle element of the array. If it is too big. Repeat the process in the left half of the array, and the right half if it’s too small. In this way, the amount of space that needs to be searched is halved every time, so the time is O(log n).

3. **Hash Search**

   Searching a hash table is easy and extremely fast, just find the hash value for the item you’re looking for then go to that index and start searching the array until you find what you are looking for or you hit a blank spot. The order is pretty close to o(1), depending on how full your hash table is.

4. Binary Tree search

Search a binary tree is just as easy as searching a hash table, but it is usually slower (especially if the tree is badly unbalanced). Just start at the root. Then go down the left subtree if the root is too big and the right subtree if is too small. Repeat until you find what you want or the sub-tree you want isn't there. The running time is O(log n) on average and O(n) in the worst case.

3.2.1.1Linear Search

Q21. Explain Linear Search with an example.

Ans :

Linear search algorithm finds given element in a list of elements with O(n) time complexity where n is total number of elements in the list. This search process starts comparing of search element with the first element in the list. If both are matching then results with element found otherwise search element is compared with next element in the list. If both are matched, then the result is "element found". Otherwise, repeat the same with the next element in the list until search element is compared with last element in the list, if that last element also doesn't match, then the result is "Element not found in the list". That means, the search element is compared with element by element in the list.

Linear search is implemented using following steps.

- **Step 1:** Read the search element from the user
- **Step 2:** Compare, the search element with the first element in the list.
- **Step 3:** If both are matching, then display "Given element found!!!" and terminate the function
- **Step 4:** If both are not matching, then compare search element with the next element in the list.
- **Step 5:** Repeat steps 3 and 4 until the search element is compared with the last element in the list.
- **Step 6:** If the last element in the list is also doesn't match, then display "Element not found!!!" and terminate the function.

Example

Consider the following list of element and search element.

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99

Search element 12

Step 1:

search element (12) is compared with first element (65)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99
	12							

Both are not matching. So move to next element

Step 2:

search element (12) is compared with next element (20)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99
		12						

Both are not matching. So move to next element

Step 3:

search element (12) is compared with next element (10)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99
			12					

Both are not matching. So move to next element

Step 4:

search element (12) is compared with next element (55)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99
				12				

Both are not matching. So move to next element

Step 5:

search element (12) is compared with next element (32)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99
					12			

both are not matching. So move to next element

Step 6:

search element (12) is compared with next element (12)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99
						12		

both are matching. So we stop comparing and display element found at index 5.

Q22. Write a program to implement Linear Search.

Ans :

Implementation of Linear search

```
#include<iostream>
using namespace std;  int main()
{
        int a[20],n,x,i,flag=0;
        cout<<"How many elements?";
        cin>>n;
        cout<<"\nEnter elements of the
                                array\n";
        for(i=0;i<n;++i)
        cin>>a[i];  cout<<"\n
        Enter element to search:";
        cin>>x;
        for(i=0;i<n;++i)
        {
        if(a[i]==x)

{       flag=1;
        break;
}
 }
if(flag)
cout<<"\nElement is found at position
"<<i+1;
else
        cout<<"\n
        Element not found";
        return 0;
}
```

Output

```
How many elements?4
Enter elements of the array
5 9 12 4
Enter element to search:9
Element is found at position 2
```

3.2.1.2 Binary Search

Q23. Explain Binary Search with an example.

Ans : **(Junly-2019)**

Binary search algorithm finds given element in a list of elements with O(log n) time complexity where n is total number of elements in the list. The binary search algorithm can be used with only sorted list of element. That means, binary search can be used only with list of element which are already arraged in a order. The binary search can not be used for list of element which are in random order. This search process starts comparing of the search element with the middle element in the list. If both are matched, then the result is "element found". Otherwise, we check whether the search element is smaller or larger than the middle element in the list. If the search element is smaller, then we repeat the same process for left sublist of the middle element. If the search element is larger, then we repeat the same process for right sublist of the middle element. We repeat this process until we find the search element in the list or until we left with a sublist of only one element. And if that element also doesn't match with the search element, then the result is "Element not found in the list".

Binary search is implemented using following steps...

- **Step 1:** Read the search element from the user
- **Step 2:** Find the middle element in the sorted list
- **Step 3:** Compare, the search element with the middle element in the sorted list.
- **Step 4:** If both are matching, then display "Given element found!!!" and terminate the function
- **Step 5:** If both are not matching, then check whether the search element is smaller or larger than middle element.
- **Step 6:** If the search element is smaller than middle element, then repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.

- **Step 7:** If the search element is larger than middle element, then repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.
- **Step 8:** Repeat the same process until we find the search element in the list or until sublist contains only one element.
- **Step 9:** If that element also doesn't match with the search element, then display "Element not found in the list!!!" and terminate the function.

Example : Consider the following list of element and search element

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99

search element 12

Step 1:

search element (12) is compared with middle element (SO)

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99
					12				

Both are not matching. And 12 is smaller than SO. So we search only in the left sublist (i.e. 10.12. 20 & 32).

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99

Step 2:

search element (12) is compared with middle element (12)

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99
		12							

Both are matching. So the result is "Element found at index 1'

search element 80

Step 1:

search element (80) is compared with middle element (50)

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99
					80				

Both are not matching. And 80 is larger than 50. So we search only in the right sublist (i.e. 55,65,80 & 99).

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99

Step 2:

search element (80) is compared with middle element (65)

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99
							80		

Both are not matching. And 80 is larger than 65. So we search only in the right sublist (i.e. 80 8t 99).

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99

Step 3:

search element (80) is compared with middle element (80)

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99
								80	

Both are not matching. So the result is "Element found at index 7"

Q24. Write a program to implement binary search.

Ans :

Binary Search implementation

```
#include<iostream>
using namespace std;
int main()
{
    int search(int [],int,int);
    int n,i,a[100],e,res;
    cout<<"How Many Elements:";
    cin>>n;
    cout<<"\nEnter Elements of Array in Ascending order\n";
    for(i=0;i<n;++i)
    {
    cin>>a[i];
    }
    cout<<"\nEnter element to search:";
    cin>>e;
    res=search(a,n,e);
    if(res!=0)
            cout<<"\nElement found at
             position "<<res+1;
    else
    cout<<"\nElement is not found....!!!";
    return 0;
}
int search(int a[],int n,int e)
{
    int f,l,m;
    f=0;
    l=n-1;
    while(f<=l)
    {
            m=(f+l)/2;
            if(e==a[m])
                    return(m);
    else
    if(e>a[m])
    f=m+1;
    else
    l=m-1;
    }
    return 0;
}
```

Output

```
    How Many Elements:5
    Enter Elements of Array in Ascending order
    12 39 40 68 77
    Enter element to search:40
    Element found at position 3
```

3.2.2 Sorting Techniques

Q25. What is sorting? Write about various sorting techniques.

Ans :

Sorting is a technique to rearrange the elements of a list in ascending or descending order, which can be numerical, lexicographical, or any user-defined order. Sorting is a process through which the data is arranged in ascending or descending order.

Sorting can be classified in two types;

Internal Sorts: This method uses only the primary memory during sorting process. All data items are held in main memory and no secondary memory is required this sorting process. If all the data that is to be sorted can be accommodated at a time in memory is called internal sorting. There is a limitation for internal sorts; they can only process relatively small lists due to memory constraints. There are 3 types of internal sorts.

- Selection sort
- Heap Sort
- Insertion sort
- Shell Sort algorithm
- Bubble Sort
- Quick sort algorithm

External Sorts: Sorting large amount of data requires external or secondary memory. This process uses external memory such as HDD, to store the data which is not fit into the main memory. So, primary memory holds the currently being sorted data only. All external sorts are based on process of merging. Different parts of data are sorted separately and merged together. **Ex:** Merge Sort

3.2.2.1 Selection Sort

Q26. Explain Selection Sort with an example.

Ans : **(June-2018)**

In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array. It is also the simplest algorithm. It is an in-place comparison sorting algorithm. In this algorithm, the array is divided into two parts, first is sorted part, and another one is the unsorted part. Initially, the sorted part of the array is empty, and unsorted part is the given array. Sorted part is placed at the left, while the unsorted part is placed at the right.

In selection sort, the first smallest element is selected from the unsorted array and placed at the first position. After that second smallest element is selected and placed in the second position. The process continues until the array is entirely sorted.

The average and worst-case complexity of selection sort is $O(n^2)$, where n is the number of items. Due to this, it is not suitable for large data sets.

Algorithm

```
SELECTION SORT(arr, n)
Step 1: Repeat Steps 2 and 3 for i = 0 to n-1
Step 2: CALL SMALLEST(arr, i, n, pos)
Step 3: SWAP arr[i] with arr[pos]
[END OF LOOP]
Step 4: EXIT
SMALLEST (arr, i, n, pos)
Step 1: [INITIALIZE] SET SMALL = arr[i]
Step 2: [INITIALIZE] SET pos = i
Step 3: Repeat for j = i+1 to n
if (SMALL > arr[j])
    SET SMALL = arr[j]
SET pos = j
[END OF if]
[END OF LOOP]
Step 4: RETURN pos
```

Working of Selection sort Algorithm

To understand the working of the Selection sort algorithm, let's take an unsorted array. It will be easier to understand the Selection sort via an example.

Let the elements of array are:

12	29	25	8	32	17	40

Now, for the first position in the sorted array, the entire array is to be scanned sequentially.

At present, 12 is stored at the first position, after searching the entire array, it is found that 8 is the smallest value.

12	29	25	8	32	17	40

So, swap 12 with 8. After the first iteration, 8 will appear at the first position in the sorted array.

8	29	25	12	32	17	40

For the second position, where 29 is stored presently, we again sequentially scan the rest of the items of unsorted array. After scanning, we find that 12 is the second lowest element in the array that should be appeared at second position.

8	29	25	12	32	17	40

Now, swap 29 with 12. After the second iteration, 12 will appear at the second position in the sorted array. So, after two iterations, the two smallest values are placed at the beginning in a sorted way.

8	12	25	29	32	17	40

The same process is applied to the rest of the array elements. Now, we are showing a pictorial representation of the entire sorting process.

8	12	25	29	32	17	40

8	12	25	29	32	17	40

8	12	17	29	32	25	40

8	12	17	29	32	25	40

8	12	17	29	32	25	40

8	12	17	25	32	29	40

8	12	17	25	32	29	40

8	12	17	25	32	29	40

8	12	17	25	29	32	40

8	12	17	25	29	32	40

Now, the array is completely sorted.

Complexity Analysis of Selection Sorting

- Worst Case Time Complexity : O(n^2)
- Best Case Time Complexity : O(n^2)
- Average Time Complexity : O(n^2)
- Space Complexity : O(1)

Q27. Write a program to implement selection sort

Ans : **(June-18)**

Implementation of selection sort

```
#include<iostream>
```

```
using namespace std;
int main()
{
        int i,j,n,loc,temp,min,a[30];
        cout<<"Enter the number of elements:";
        cin>>n;
        cout<<"\nEnter the elements\n";
        for(i=0;i<n;i++)
        {
                cin>>a[i];
        }
        for(i=0;i<n-1;i++)
        {
        min=a[i];
        loc=i;
        for(j=i+1;j<n;j++)
        {
                if(min>a[j])
                {
                        min=a[j];
                        loc=j;
                }       }

      temp=a[i];
      a[i]=a[loc];
      a[loc]=temp;
}

cout<<"\nSorted list is as follows\n";
for(i=0;i<n;i++)
{
        cout<<a[i]<<" ";
}
return 0;
}
```

3.2.2.2 Bubble Sort

Q28. Explain bubble sort with an example.

Ans : **(Dec.-19, June - 18)**

Bubble Sort is an algorithm which is used to sort N elements that are given in a memory for eg: an Array with N number of elements. Bubble Sort compares all the element one by one and sort them based on their values.

It is called Bubble sort, because with each iteration the smaller element in the list bubbles up towards the first place, just like a water bubble rises up to the water surface.

Sorting takes place by stepping through all the data items one-by-one in pairs and comparing adjacent data items and swapping each pair that is out of order.

Algorithm

In the algorithm given below, suppose arr is an array of n elements. The assumed swap function in the algorithm will swap the values of given array elements.

```
begin BubbleSort(arr)
    for all array elements
    if arr[i] > arr[i+1]
    swap(arr[i], arr[i+1])
    end if
    end for
    return arr
end BubbleSort
```

Working of Bubble sort Algorithm

To understand the working of bubble sort algorithm, let's take an unsorted array. We are taking a short and accurate array, as we know the complexity of bubble sort is O(n^2).

Let the elements of array are:

13	32	26	35	10

First Pass

Sorting will start from the initial two elements. Let compare them to check which is greater.

13	32	26	35	10

Here, 32 is greater than 13 (32 > 13), so it is already sorted. Now, compare 32 with 26.

13	32	26	35	10

Here, 26 is smaller than 36. So, swapping is required. After swapping new array will look like -

13	26	32	35	10

Now, compare 32 and 35.

13	26	32	35	10

Here, 35 is greater than 32. So, there is no swapping required as they are already sorted.

Now, the comparison will be in between 35 and 10.

13	26	32	35	10

Here, 10 is smaller than 35 that are not sorted. So, swapping is required. Now, we reach at the end of the array. After first pass, the array will be:

13	26	32	10	35

Now, move to the second iteration.

Second Pass

The same process will be followed for second iteration.

13	26	32	10	35
13	26	32	10	35
13	26	32	10	35

Here, 10 is smaller than 32. So, swapping is required. After swapping, the array will be:

13	26	10	32	35
13	26	10	32	35

Now, move to the third iteration.

Third Pass

The same process will be followed for third iteration.

13	26	10	32	35
13	26	10	32	35

Here, 10 is smaller than 26. So, swapping is required. After swapping, the array will be:

13	10	26	32	35
13	10	26	32	35
13	10	26	32	35

Now, move to the fourth iteration.

Fourth pass

Similarly, after the fourth iteration, the array will be

10	13	26	32	35

Hence, there is no swapping required, so the array is completely sorted.

Time Complexity

Case Time	Complexity
Best Case	O(n)
Average Case	O(n^2)
Worst Case	O(n^2)

```
int a[6] = {5, 1, 6, 2, 4, 3};
int i, j, temp;
for(i=0; i<6; i++)
{
int flag = 0;    //taking a flag variable
for(j=0; j<6-i-1; j++)
{
    if( a[j] > a[j+1])
{
    temp = a[j];
    a[j] = a[j+1];
```

```
        a[j+1] = temp;
    flag = 1;       //setting flag as 1, if swapping occurs
    }  }
    if(!flag)          //breaking out of for loop if no swapping takes place
    {
    break;
    }}
```

3.2.2.3 Insertion Sort

Q29. What is insertion sort? Explain its working with an example.

Ans : **(Dec.-19, Dec.-18, Nov.-18)**

Insertion sort works similar to the sorting of playing cards in hands. It is assumed that the first card is already sorted in the card game, and then we select an unsorted card. If the selected unsorted card is greater than the first card, it will be placed at the right side; otherwise, it will be placed at the left side. Similarly, all unsorted cards are taken and put in their exact place.

The same approach is applied in insertion sort. The idea behind the insertion sort is that first take one element, iterate it through the sorted array. Although it is simple to use, it is not appropriate for large data sets as the time complexity of insertion sort in the average case and worst case is $O(n^2)$, where n is the number of items. Insertion sort is less efficient than the other sorting algorithms like heap sort, quick sort, merge sort, etc.

Insertion sort has various advantages such as:

➢ Simple implementation

➢ Efficient for small data sets

➢ Adaptive, i.e., it is appropriate for data sets that are already substantially sorted.

Algorithm

The simple steps of achieving the insertion sort are listed as follows:

Step 1:

If the element is the first element, assume that it is already sorted. Return 1.

Step2: Pick the next element, and store it separately in a key.

Step3: Now, compare the key with all elements in the sorted array.

Step 4: If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.

Step 5: Insert the value.

Step 6: Repeat until the array is sorted.

Working of Insertion sort Algorithm

To understand the working of the insertion sort algorithm, let's take an unsorted array. It will be easier to understand the insertion sort via an example.

Let the elements of array are:

12	31	25	8	32	17

Initially, the first two elements are compared in insertion sort.

12	31	25	8	32	17

Here, 31 is greater than 12. That means both elements are already in ascending order. So, for now, 12 is stored in a sorted sub-array.

12	31	25	8	32	17

Now, move to the next two elements and compare them.

12	31	25	8	32	17
12	31	25	8	32	17

Here, 25 is smaller than 31. So, 31 is not at correct position. Now, swap 31 with 25. Along with swapping, insertion sort will also check it with all elements in the sorted array.

For now, the sorted array has only one element, i.e. 12. So, 25 is greater than 12. Hence, the sorted array remains sorted after swapping.

12	25	31	8	32	17

Now, two elements in the sorted array are 12 and 25. Move forward to the next elements that are 31 and 8.

12	25	31	8	32	17
12	25	31	8	32	17

Both 31 and 8 are not sorted. So, swap them.

12	25	8	31	32	17

After swapping, elements 25 and 8 are unsorted.

12	25	8	31	32	17

So, swap them.

12	8	25	31	32	17

Now, elements 12 and 8 are unsorted.

12	8	25	31	32	17

So, swap them too.

8	12	25	31	32	17

Now, the sorted array has three items that are 8, 12 and 25. Move to the next items that are 31 and 32.

8	12	25	31	32	17

Hence, they are already sorted. Now, the sorted array includes 8, 12, 25 and 31.

8	12	25	31	32	17

Move to the next elements that are 32 and 17.

8	12	25	31	32	17

17 is smaller than 32. So, swap them.

8	12	25	31	17	32

8	12	25	31	17	32

Swapping makes 31 and 17 unsorted. So, swap them too.

8	12	25	17	31	32

8	12	25	17	31	32

Now, swapping makes 25 and 17 unsorted. So, perform swapping again.

8	12	17	25	31	32

Now, the array is completely sorted.

Time Complexity

➢ Best Case	O(n)
➢ Average Case	O(n^2)
➢ Worst Case	O(n^2)

Q30. Write a program to implement Insertion sort.

Ans :

Insertion Sorting in C++

```
#include <stdlib.h>
#include <iostream.h>
using namespace std;
//member functions declaration
void insertionSort(int arr[], int length);
void printArray(int array[],int size);
int main() {
int array[5]= {5,4,3,2,1};
insertionSort(array,5);
return 0;
}
```

```
void insertionSort(int arr[], int length) {
int i, j ,tmp;
for (i = 1; i < length; i++) {
      j = i;
      while (j > 0 && arr[j - 1] > arr[j]) {
            tmp = arr[j];
            arr[j] = arr[j - 1];
            arr[j - 1] = tmp;
            j—;
      }
printArray(arr,5);
}}
void printArray(int array[], int size){
cout<< "Sorting tha array using Insertion sort... ";
int j;
for (j=0; j < size;j++)
      for (j=0; j < size;j++)
            cout <<" "<< array[j];
cout << endl;
}
```

Complexity Analysis of Insertion Sorting

Worst Case Time Complexity : O(n^2)

Best Case Time Complexity : O(n)

Average Time Complexity : O(n^2)

Space Complexity : O(1)

3.2.2.4 Merge Sort

Q31. What is merge sort? Explain it with an example.

Ans : **(Aug.-21, Dec.-19)**

Merge sort is similar to the quick sort algorithm as it uses the divide and conquer approach to sort the elements. It is one of the most popular and efficient sorting algorithm. It divides the given list into two equal halves, calls itself for the two halves and then merges the two sorted halves. We have to define the merge() function to perform the merging.

The sub-lists are divided again and again into halves until the list cannot be divided further. Then we combine the pair of one element lists into two-element lists, sorting them in the process. The sorted two-element pairs is merged into the four-element lists, and so on until we get the sorted list.

Algorithm

In the following algorithm, arr is the given array, beg is the starting element, and end is the last element of the array.

```
MERGE_SORT(arr, beg, end)
      if beg < end
set mid = (beg + end)/2
MERGE_SORT(arr, beg, mid)
MERGE_SORT(arr, mid + 1, end)
MERGE (arr, beg, mid, end)
end of if
END MERGE_SORT
```

The important part of the merge sort is the MERGE function. This function performs the merging of two sorted sub-arrays that are A[beg…mid] and A[mid+1…end], to build one sorted array A[beg…end]. So, the inputs of the MERGE function are A[], beg, mid, and end.

The implementation of the MERGE function is given as follows:

Sorting using Merge Sort Algorithm

/* a[] is the array, p is starting index, that is 0, and r is the last index of array. */

Lets take a[5] = {32, 45, 67, 2, 7} as the array to be sorted.

```
void mergesort(int a[], int p, int r)
{
      int q;
```

```
        if(p < r)
        {
        q = floor( (p+r) / 2);
        mergesort(a, p, q);
        mergesort(a, q+1, r);
        merge(a, p, q, r);
    }
    }
    void merge(int a[], int p, int q, int r)
    {
        int b[5];     //same size of a[]
        int i, j, k;
        k = 0;
        i = p;
        j = q+1;
        while(i <= q && j <= r)
    {
        if(a[i] < a[j])
    {
        b[k++] = a[i++];
        // same as b[k]=a[i]; k++; i++;
    }
    else
    {
        b[k++] = a[j++];
    } }
    while(i <= q)
    {
        b[k++] = a[i++];
    }
        while(j <= r)
    {
        b[k++] = a[j++];
    }
        for(i=r; i >= p; i—)
    {
        a[i] = b[—k];
         // copying back the sorted list to a[]
    }
}
```

Complexity Analysis of Merge Sort

Worst Case Time Complexity : O(n log n)

Best Case Time Complexity : O(n log n)

Average Time Complexity : O(n log n)

Space Complexity : O(n)

- Time complexity of Merge Sort is O(n Log n) in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves.
- It requires equal amount of additional space as the unsorted list. Hence its not at all recommended for searching large unsorted lists.
- It is the best Sorting technique for sorting Linked Lists.

3.2.2.5 Quick Sort

Q32. What is quick sort? Explain it with an example?

Ans : **(July-2019)**

Sorting is a way of arranging items in a systematic manner. Quicksort is the widely used sorting algorithm that makes n log n comparisons in average case for sorting an array of n elements. It is a faster and highly efficient sorting algorithm. This algorithm follows the divide and conquer approach. Divide and conquer is a technique of breaking down the algorithms into subproblems, then solving the subproblems, and combining the results back together to solve the original problem.

Divide

In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

Conquer

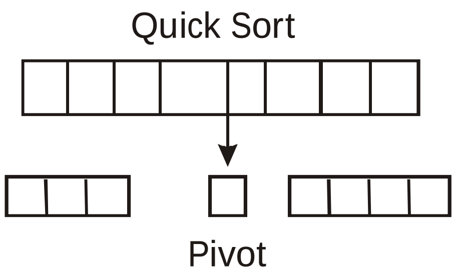
Recursively, sort two subarrays with Quicksort.

Combine

Combine the already sorted array.

Quicksort picks an element as pivot, and then it partitions the given array around the picked pivot element. In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot.

After that, left and right sub-arrays are also partitioned using the same approach. It will continue until the single element remains in the sub-array.



Choosing the pivot

Picking a good pivot is necessary for the fast implementation of quicksort. However, it is typical to determine a good pivot. Some of the ways of choosing a pivot are as follows:

- Pivot can be random, i.e. select the random pivot from the given array.
- Pivot can either be the rightmost element of the leftmost element of the given array.
- Select median as the pivot element.

Algorithm:

```
QUICKSORT (array A, start, end)
{
if (start < end)
{
p = partition(A, start, end)
QUICKSORT (A, start, p - 1)
QUICKSORT (A, p + 1, end)
}
}
```

Partition Algorithm

The partition algorithm rearranges the sub-arrays in a place.

```
PARTITION (array A, start, end)
{
pivot ? A[end]
i ? start-1
for j ? start to end -1 {
do if (A[j] < pivot) {
then i ? i + 1
swap A[i] with A[j]
}}
swap A[i+1] with A[end]
return i+1
}
```

Working of Quick Sort Algorithm

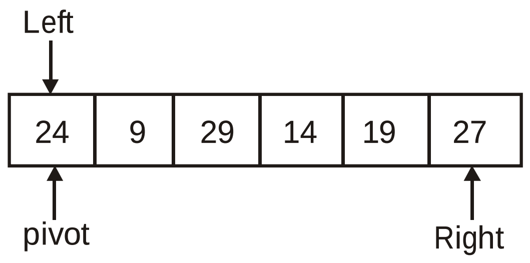
To understand the working of quick sort, let's take an unsorted array. It will make the concept more clear and understandable.

Let the elements of array are:

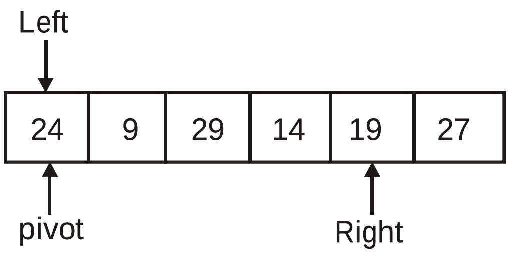
24	9	29	14	19	27

In the given array, we consider the leftmost element as pivot. So, in this case, a[left] = 24, a[right] = 27 and a[pivot] = 24.

Since, pivot is at left, so algorithm starts from right and move towards left.

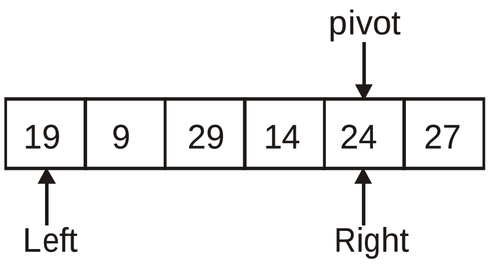


Now, a[pivot] < a[right], so algorithm moves forward one position towards left, i.e.



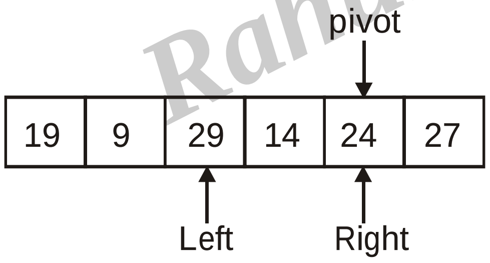
Now, a[left] = 24, a[right] = 19, and a[pivot] = 24.

Because, a[pivot] > a[right], so, algorithm will swap a[pivot] with a[right], and pivot moves to right, as -

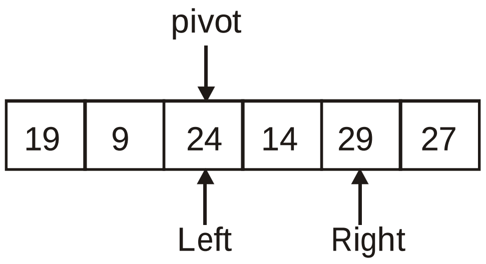


Now, a[left] = 19, a[right] = 24, and a[pivot] = 24. Since, pivot is at right, so algorithm starts from left and moves to right.

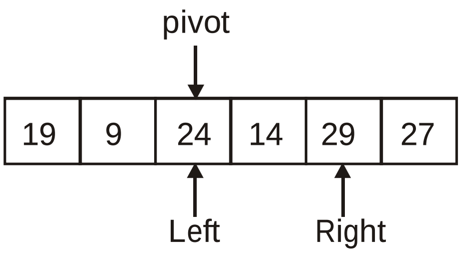
As a[pivot] > a[left], so algorithm moves one position to right as -



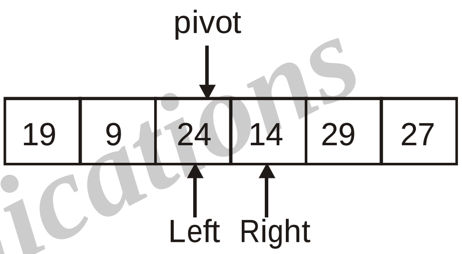
Now, a[left] = 9, a[right] = 24, and a[pivot] = 24. As a[pivot] > a[left], so algorithm moves one position to right as -



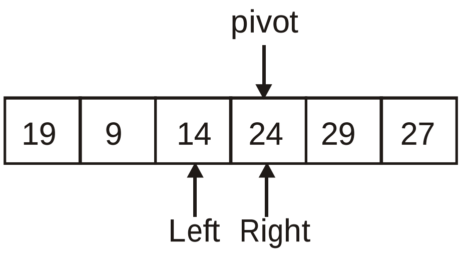
Now, a[left] = 29, a[right] = 24, and a[pivot] = 24. As a[pivot] < a[left], so, swap a[pivot] and a[left], now pivot is at left, i.e. -



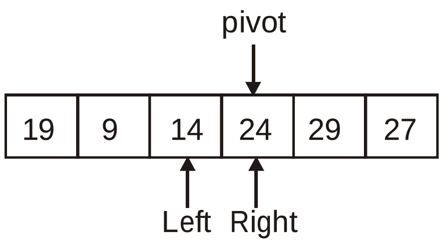
Since, pivot is at left, so algorithm starts from right, and move to left. Now, a[left] = 24, a[right] = 29, and a[pivot] = 24. As a[pivot] < a[right], so algorithm moves one position to left, as -



Now, a[pivot] = 24, a[left] = 24, and a[right] = 14. As a[pivot] > a[right], so, swap a[pivot] and a[right], now pivot is at right, i.e. -



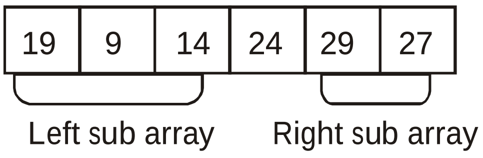
Now, a[pivot] = 24, a[left] = 14, and a[right] = 24. Pivot is at right, so the algorithm starts from left and move to right.



Now, a[pivot] = 24, a[left] = 24, and a[right] = 24. So, pivot, left and right are pointing the same element. It represents the termination of procedure.

Element 24, which is the pivot element is placed at its exact position.

Elements that are right side of element 24 are greater than it, and the elements that are left side of element 24 are smaller than it.



Now, in a similar manner, quick sort algorithm is separately applied to the left and right sub-arrays. After sorting gets done, the array will be -

9	14	19	24	27	29

- **Time Complexity**

Best Case	O(n*logn)
Average Case	O(n*logn)
Worst Case	O(n^2)

Q33. Write a program to implement quick sort.

Ans :

```
#include<iostream>
#include<cstdlib>
usingnamespace std;
void swap(int*a,int*b){
    int temp;
    temp =*a;
    *a =*b;
    *b = temp;
}
intPartition(int a[],int l,int h){
    int pivot, index, i;
    index = l;
    pivot = h;
    for(i = l; i < h; i++){
        if(a[i]< a[pivot]){
        swap(&a[i],&a[index]);
        index++;
    }
}
swap(&a[pivot],&a[index]);
return index;
}
intRandomPivotPartition(int a[],int l,int h){
int pvt, n, temp;
n = rand();
pvt = l + n%(h-l+1);
swap(&a[h],&a[pvt]);
returnPartition(a, l, h);
}
intQuickSort(int a[],int l,int h){
int pindex;
if(l < h){
pindex =RandomPivotPartition(a, l, h);
QuickSort(a, l, pindex-1);
QuickSort(a, pindex+1, h);
}
return0;
}
int main(){
int n, i;
cout<<"\nEnter the number of data element to be sorted: ";
cin>>n;
int arr[n];
for(i =0; i < n; i++){
cout<<"Enter element "<<i+1<<": ";
cin>>arr[i];
}
QuickSort(arr,0, n-1);
cout<<"\nSorted Data ";
for(i =0; i < n; i++)
cout<<"->"<<arr[i];
return0;
}
```

Output

```
Enter the number of data element to be sorted: 4 Enter element 1: 3
Enter element 2: 4
Enter element 3: 7
Enter element 4: 6
Sorted Data ->3->4->6->7
```

3.2.3 Comparison of All Sorting Methods

Q34. Compare various sorting techniques with real world usage.

Ans : **(July-21, Nov.-2019)**

Selection sort

Selection sort is an exception in our list. This is considered an academic sorting algorithm. Because the time efficiency is always O(n^2) which is not acceptable. There is no real world usage for selection sort except passing the data structure course exam.

Pros

➢ Nothing

cons

➢ Always run at O(n^2) even at best case scenario

Bubble sort

This is the other exception in the list because bubble sort is too slow to be practical. Unless the sequence is almost sorted feasibility of bubble sort is zero and the running time is O(n^2). This is one of the three simple sorting algorithms alongside selection sort and insertion sort but like selection sort falls short of insertions sort in terms of efficiency even for small sequences.

Pros

➢ Again nothing, maybe just "catchy name[1]"

Cons

➢ With polynomial O(n^2) it is too slow

Insertion sort

Insertion sort is definitely not the most efficient algorithm out there but its power lies in its simplicity. Since it is very easy to implement and adequately efficient for small number of elements, it is useful for small applications or trivial ones. The definition of small is vague and depends on a lot of things but a safe bet is if under 50, insertion sort is fast enough. Another situation that insertion sort is useful is when the sequence is almost sorted. Such sequences may seem like exceptions but in real world applications often you encounter almost sorted elements. The run time of insertions sort is O(n^2) at worst case scenario. So far we have another useless alternative for selection sort. But if implemented well the run time can be reduced to O(n+k). n is the number of elements and k is the number of inversions (the number of pair of elements out of order). With this new run time in mind you can see if the sequence is almost sorted (k is small) the run time can be almost linear which is a huge improvement over the polynomial n^2.

Pros

➢ Easy to implement

➢ The more the sequence is ordered the closer is run time to linear time O(n)

Cons

➢ Not suitable for large data sets

➢ Still polynomial at worst case

Heap sort

This is the first general purpose sorting algorithm we are introducing here. Heap sort runs at O(nlogn) which is optimal for comparison based sorting algorithms. Though heap sort has the same run time as quick sort and merge sort but it is usually outperformed in real world scenarios. If you are asking then why should anyone use it, the answer lies in space efficiency. Nowadays computers come with huge amount of memory, enough for many applications. Does this mean heap sort is losing its shine? No, still when writing programs for environments with limited memory, such as embedded systems or space efficiency is much more important than time efficiency. A rule of thumb is if the sequence is small enough to easily fit in main memory then heap sort is good choice.

Pros

- Runs at O(nlogn)
- Can be easily implemented to be executed in place

Cons

- Not as fast as other comparison based algorithms in large data sets
- It doesn't provide stable sorting

Quick sort

One of the most widely used sorting algorithms in computer industry. Surprisingly quick sort has a running time of O(n^2) that makes it susceptible in real-time applications. Having a polynomial worst case scenario still quick sort usually outperforms both quick sort and merge sort (coming next). The reason behind the popularity of quick sort despite the short comings is both being fast in real world scenarios (not necessarily worst case) and the ability to be implemented as an in place algorithm.

Pros

- Most often than not runs at O(nlogn)
- Quick sort is tried and true, has been used for many years in industry so you can be assured it is not going to fail you
- High space efficiency by executing in place

Cons

- Polynomial worst case scenario makes it susceptible for time critical applications
- Provides non stable sort due to swapping of elements in partitioning step

Merge sort

Having a O(nlogn) worst case scenario run time makes merge sort a powerful sorting algorithm. The main drawback of this algorithm is its space inefficiency. That is in the process of sorting lots of temporary arrays have to be created and many copying of elements is involved. This doesn't mean merge sort is not useful. When the data to be sorted is distributed across different locations like cache, main memory etc then copying data is inevitable. Merge sort mainly owes its popularity to Tim Peters who designed a variant of it which is in essence a bottom-up merge sort and is known as Tim sort.

Pros

- Excellent choice when data is fetched from resources other than main memory
- Having a worst case scenario run time of O(nlogn) which is optimal
- Tim sort variant is really powerful

Cons

- Lots of overhead in copying data between arrays and making new arrays
- Extremely difficult to implement it in place for arrays
- Space inefficiency

Special purpose sorting algorithms

Though currently O(nlogn) seems like an unbreakable cap for sorting algorithms, this just holds true for general purpose sorts. If the entities to be sorted are integers, strings or d-tuples then you are not limited by the sorting algorithms above. Radix sort and Bucket sort are two of most famous special purpose sorting algorithms. their worst case scenario run time is O(f(n+r)). [0, r-1] is the range of integers and f=1 for bucket sort. All in all this means if f(n+r) is significantly below nlogn function then these methods are faster than three powerful general purpose sorting algorithms, merge sort, quick sort and heap sort.

pros

- They can run faster than nlogn

cons

- Cannot be used for every type of data
- Not necessarily always run faster than general purpose algorithms

Comparison table:

	worst case time	average case time	best case time	worst case space
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(n)
Bubble sort	$O(n^2)$	$O(n^2)$	O(n)	O(1)
Insertion sort	$O(n^2)$	$O(n^2)$	O(n)	O(n)
Heap sort	O(nlogn)	O(nlogn)	O(nlogn)	O(1)
Quick sort	$O(n^2)$	O(nlogn)	O(nlogn)	O(logn)
Merge sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)

3.3 Search Trees

3.3.1 Symbol Table

Q35. What is symbol table? How to represent it?

Ans :

Symbol Table

The symbol table is a kind of a 'keyed table' which stores <key, information> pairs with no additional logical structure.

The operations performed on symbol tables are the following:

1. Inserting the <key, information> pairs into the collection.
2. Removing the <key, information> pairs by specifying the key.
3. Searching for a particular key.
4. Retrieving the information associated with a key.

Representation of Symbol Table

There are two different techniques for implementing a keyed table, namely, the symbol table and the tree table.

Static Tree Tables

When symbols are known in advance and no insertion and deletion is allowed, such a structure i called a static tree table. An example of this type of table is a reserved word table in a compiler.

There are four options for searching:

1. Static tree table can be stored as a sorted sequential list and binary search (O(log2n)) can be use to search a symbol.
2. Balanced BST can be used to find symbols having equal probabilities.
3. Hash tables, having the search time O(1), can be used to store a symbol table.
4. OBST is used when different symbols are searched with different probabilities.

Dynamic Tree Tables

A dynamic tree table is used when symbols are not known in advance but are inserted as they com and deleted if not required. Dynamic keyed tables are those that are built onthe-fly. The keys have n history associated with their use. The dynamically built tree that is a balanced BST is the best choice.

3.3.2 Optimal Binary Search Tree

Q36. Explain OBST with an example.

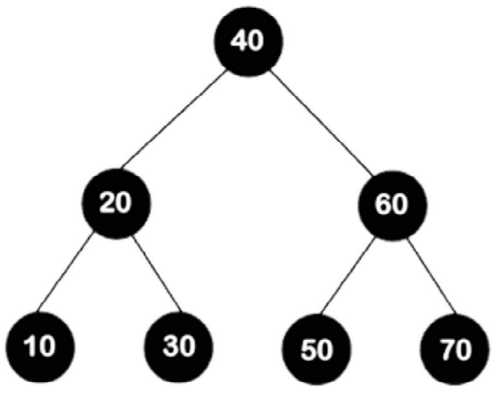
Ans :

As we know that in binary search tree, the nodes in the left subtree have lesser value than the roo node and the nodes in the right subtree have greater value than the root node.

We know the key values of each node in the tree, and we also know the frequencies of each nod in terms of searching means how much time is required to search a node. The frequency and key-valu determine the overall cost of searching a node. The cost of searching is a very important factor in variou applications. The overall cost of searching a node should be less. The time required to search a node i BST is more than the balanced binary search tree as a balanced binary search tree contains a lesse number of levels than the BST. There is one way that can reduce the cost of a binary search treeis know as an optimal binary search tree.

Let's understand through an example.

If the keys are 10, 20, 30, 40, 50, 60, 70



In the above tree, all the nodes on the left subtree are smaller than the value of the root node, and all the nodes on the right subtree are larger than the value of the root node. The maximum time required to search a node is equal to the minimum height of the tree, equal to logn.

Now we will see how many binary search trees can be made from the given number of keys.

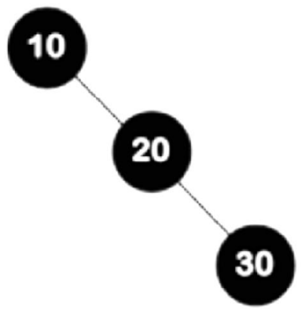
For example: 10, 20, 30 are the keys, and the following are the binary search trees that can be made out from these keys.

The Formula for calculating the number of trees:

$$\frac{{}^{2n}C_n}{n+1}$$

When we use the above formula, then it is found that total 5 number of trees can be created.

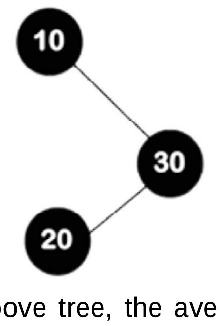
The cost required for searching an element depends on the comparisons to be made to search an element. Now, we will calculate the average cost of time of the above binary search trees.



In the above tree, total number of 3 comparisons can be made. The average number of comparisons can be made as:

Average number of comparisons

$$= \frac{1+2+3}{3} = 2$$



In the above tree, the average number of comparisons that can be made as:

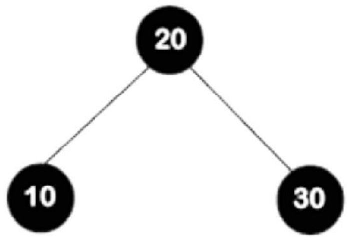
Average number of comparisons

$$= \frac{1+2+3}{3} = 2$$

In the above tree, the average number of comparisons that can be made as:

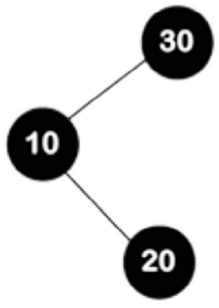
Average number of comparisons

$$= \frac{1+2+3}{3} = 2$$



Average number of comparisons

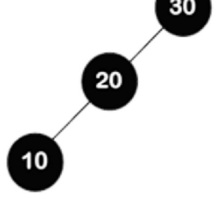
$$= \frac{1+2+2}{3} = 5/3$$



In the above tree, the total number of comparisons can be made as 3. Therefore, the average number of comparisons that can be made as:

Average number of comparisons

$$= \frac{1+2+2}{3} = 2$$



In the above tree, the total number of comparisons can be made as 3. Therefore, the average number of comparisons that can be made as:

Average number of comparisons $= \frac{1+2+2}{3} = 2$

In the third case, the number of comparisons is less because the height of the tree is less, so it’s a balanced binary search tree.

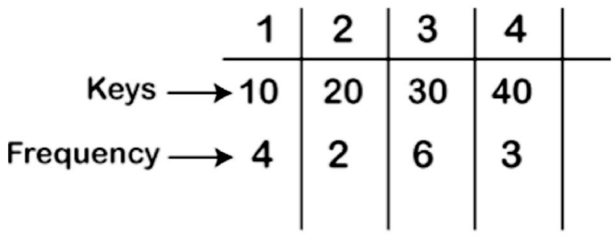
Till now, we read about the height-balanced binary search tree. To find the optimal binary search tree, we will determine the frequency of searching a key.

Let’s assume that frequencies associated with the keys 10, 20, 30 are 3, 2, 5.

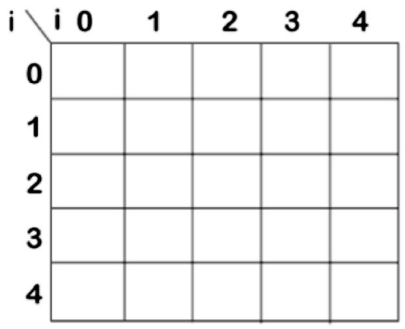
The above trees have different frequencies. The tree with the lowest frequency would be considered the optimal binary search tree. The tree with the frequency 17 is the lowest, so it would be considered as the optimal binary search tree.

Dynamic Approach

Consider the below table, which contains the keys and frequencies.



	1	2	3	4
Keys →	10	20	30	40
Frequency →	4	2	6	3



i \ i	0	1	2	3	4
0					
1					
2					
3					
4					

First, we will calculate the values where j-i is equal to zero.

When i=0, j=0, then j-i = 0

When i = 1, j=1, then j-i = 0

When i = 2, j=2, then j-i = 0

When i = 3, j=3, then j-i = 0

When i = 4, j=4, then j-i = 0

Therefore, c[0, 0] = 0, c[1 , 1] = 0, c[2,2] = 0, c[3,3] = 0, c[4,4] = 0

Now we will calculate the values where j-i equal to 1.

When j=1, i=0 then j-i = 1

When j=2, i=1 then j-i = 1

When j=3, i=2 then j-i = 1

When j=4, i=3 then j-i = 1

Now to calculate the cost, we will consider only the jth value.

The cost of c[0,1] is 4 (The key is 10, and the cost corresponding to key 10 is 4).

The cost of c[1,2] is 2 (The key is 20, and the cost corresponding to key 20 is 2).

The cost of c[2,3] is 6 (The key is 30, and the cost corresponding to key 30 is 6)

The cost of c[3,4] is 3 (The key is 40, and the cost corresponding to key 40 is 3)

	0	1	2	3	4
0	0	4			
1		0	2		
2			0	6	
3				0	3
4					0

Now we will calculate the values where j-i = 2

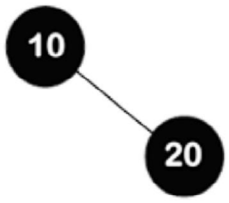
When j=2, i=0 then j-i = 2

When j=3, i=1 then j-i = 2

When j=4, i=2 then j-i = 2

In this case, we will consider two keys.

➢ When i=0 and j=2, then keys 10 and 20. There are two possible trees that can be made out from these two keys shown below:



In the first binary tree, cost would be: 4*1 + 2*2 = 8

In the second binary tree, cost would be: 4*2 + 2*1 = 10

The minimum cost is 8; therefore, c[0,2] = 8

	0	1	2	3	4
0	0	4	8		
1		0	2		
2			0	6	
3				0	3
4					0

➢ When i=1 and j=3, then keys 20 and 30. There are two possible trees that can be made out from these two keys shown below:

In the first binary tree, cost would be:

1*2 + 2*6 = 14

➢ In the second binary tree, cost would be:

1*6 + 2*2 = 10

The minimum cost is 10;

therefore, c[1,3] = 10

➢ When i=2 and j=4, we will consider the keys at 3 and 4, i.e., 30 and 40. There are two possible trees that can be made out from these two keys shown as below:

In the first binary tree, cost would be:

1*6 + 2*3 = 12

In the second binary tree, cost would be:

1*3 + 2*6 = 15

The minimum cost is 12,

therefore, c[2,4] = 12

i \ j	0	1	2	3	4
0	0	4	8^1		
1		0	2	10^3	
2			0	6	12^3
3				0	3
4					0

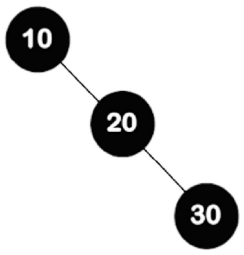
Now we will calculate the values when j-i = 3

When j=3, i=0 then j-i = 3

When j=4, i=1 then j-i = 3

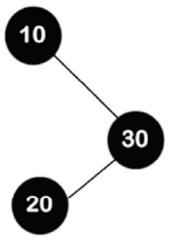
➢ When i=0, j=3 then we will consider three keys, i.e., 10, 20, and 30.

The following are the trees that can be made if 10 is considered as a root node.



In the above tree, 10 is the root node, 20 is the right child of node 10, and 30 is the right child of node 20.

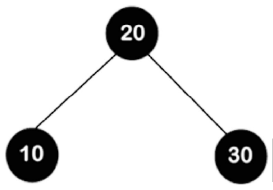
Cost would be: 1*4 + 2*2 + 3*6 = 26



In the above tree, 10 is the root node, 30 is the right child of node 10, and 20 is the left child of node 20.

Cost would be: 1*4 + 2*6 + 3*2 = 22

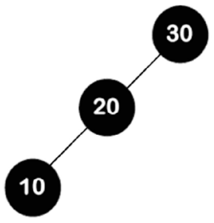
The following tree can be created if 20 is considered as the root node.



In the above tree, 20 is the root node, 30 is the right child of node 20, and 10 is the left child of node 20.

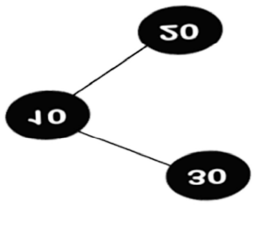
Cost would be: 1*2 + 4*2 + 6*2 = 22

The following are the trees that can be created if 30 is considered as the root node.



In the above tree, 30 is the root node, 20 is the left child of node 30, and 10 is the left child of node 20.

Cost would be: 1*6 + 2*2 + 3*4 = 22



In the above tree, 30 is the root node, 10 is the left child of node 30 and 20 is the right child of node 10.

Cost would be: 1*6 + 2*4 + 3*2 = 20

Therefore, the minimum cost is 20 which is the 3rd root. So, c[0,3] is equal to 20.

- When i=1 and j=4 then we will consider the keys 20, 30, 40

  c[1,4] = min{ c[1,1] + c[2,4], c[1,2] + c[3,4], c[1,3] + c[4,4] } + 11

  = min{0+12, 2+3, 10+0}+ 11

  = min{12, 5, 10} + 11

  The minimum value is 5; therefore, c[1,4] = 5+11 = 16

i\j	0	1	2	3	4
0	0	4	8^1	20^3	
1		0	2	10^3	16^3
2			0	6	12^3
3				0	3
4					0

- Now we will calculate the values when j-i = 4

  When j=4 and i=0 then j-i = 4

  In this case, we will consider four keys, i.e., 10, 20, 30 and 40. The frequencies of 10, 20, 30 and 40 are 4, 2, 6 and 3 respectively.

  w[0, 4] = 4 + 2 + 6 + 3 = 15

  If we consider 10 as the root node then

  C[0, 4] = min {c[0,0] + c[1,4]}+ w[0,4]= min {0 + 16} + 15= 31

  If we consider 20 as the root node then

  C[0,4] = min{c[0,1] + c[2,4]} + w[0,4]= min{4 + 12} + 15= 16 + 15 = 31

  If we consider 30 as the root node then,

  C[0,4] = min{c[0,2] + c[3,4]} +w[0,4] = min {8 + 3} + 15 = 26

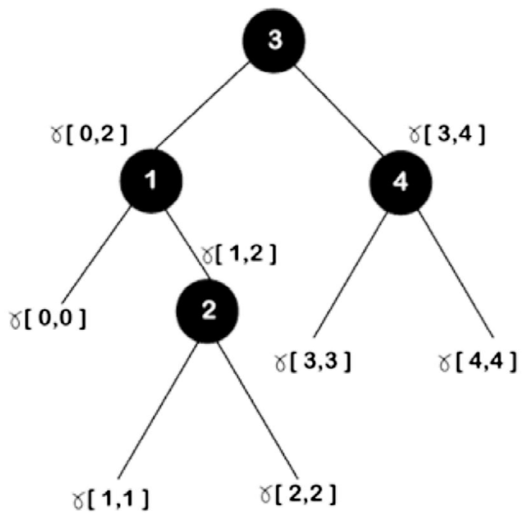
  If we consider 40 as the root node then,

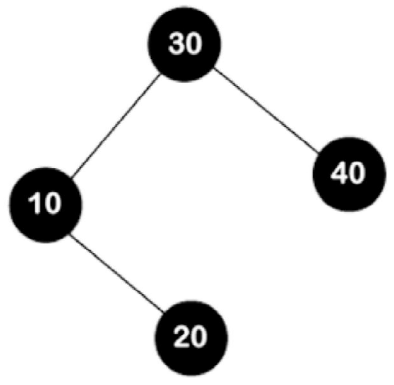
  C[0,4] = min{c[0,3] + c[4,4]} + w[0,4]= min{20 + 0} + 15= 35

  In the above cases, we have observed that 26 is the minimum cost; therefore, c[0,4] is equal to 26.

i\j	0	1	2	3	4
0	0	4	8^1	20^3	26^3
1		0	2	10^3	16^3
2			0	6	12^3
3				0	3
4					0

The optimal binary tree can be created as:





General formula for calculating the minimum cost is:

C[i,j] = min{c[i, k-1] + c[k,j]} + w(i,j)

3.3.3 AVL Tree (Heightbalanced Tree)

Q37. What is AVL tree? Explain the rotations in AVL tree.

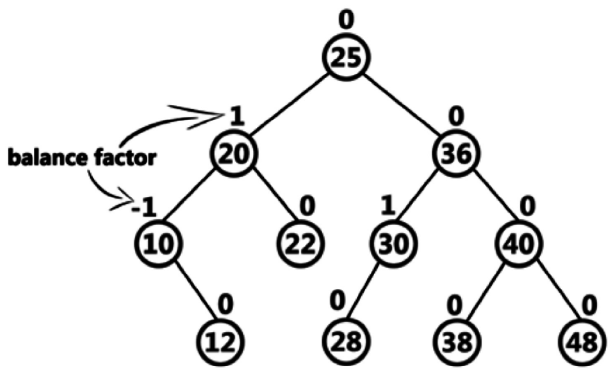
Ans :

An AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either -1, 0 or +1.

Balance factor of a node is the difference between the heights of left and right subtrees of that node. The balance factor of a node is calculated either height of left subtree - height of right subtree (OR) height of right subtree - height of left subtree. In the following explanation, we are calculating as follows...

Balance factor = heightOfLeftSubtree – height Of RightSubtree

Example



The above tree is a binary search tree and every node is satisfying balance factor condition. So this tree is said to be an AVL tree.

Every AVL Tree is a binary search tree but all the Binary Search Trees need not to be AVL trees.

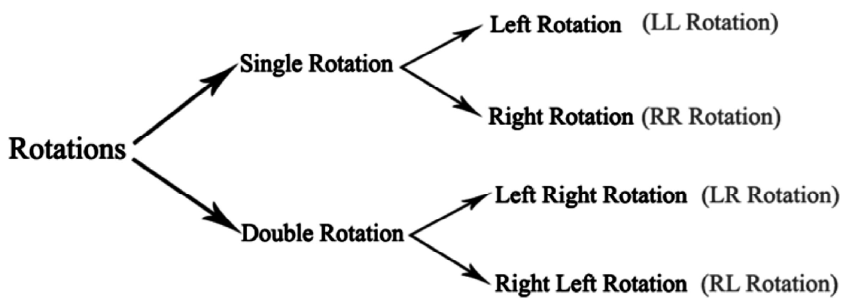
AVL Tree Rotations

In AVL tree, after performing every operation like insertion and deletion we need to check the balance factor of every node in the tree. If every node satisfies the balance factor condition then we conclude the operation otherwise we must make it balanced. We use rotationoperations to make the tree balanced whenever the tree is becoming imbalanced due to any operation.

Rotation operations are used to make a tree balanced.

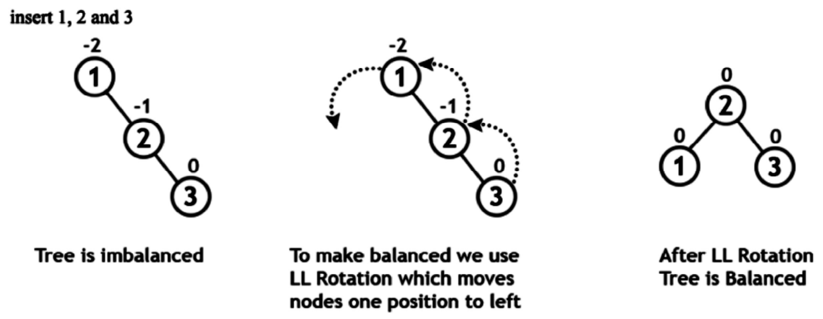
Rotation is the process of moving the nodes to either left or right to make tree balanced.

There are four rotations and they are classified into two types.


Rotations
Single Rotation
Left Rotation (LL Rotation)
Right Rotation (RR Rotation)
Double Rotation
Left Right Rotation (LR Rotation)
Right Left Rotation (RL Rotation)

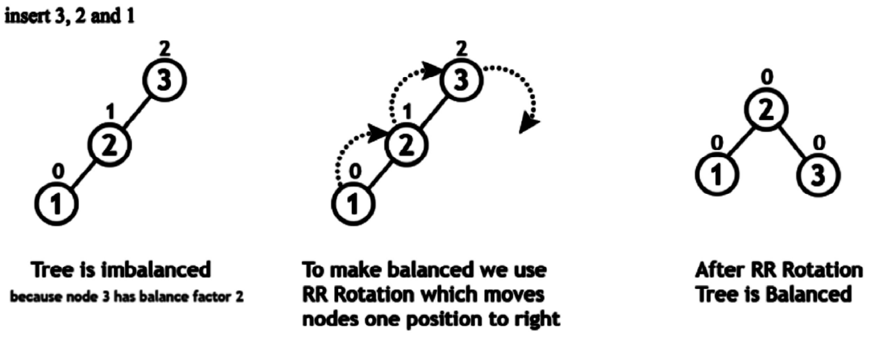
Single Left Rotation (LL Rotation)

In LL Rotation every node moves one position to left from the current position. To understand LL Rotation, let us consider following insertion operations into an AVL Tree.


insert 1, 2 and 3
Tree is imbalanced
To make balanced we use LL Rotation which moves nodes one position to left
After LL Rotation Tree is Balanced

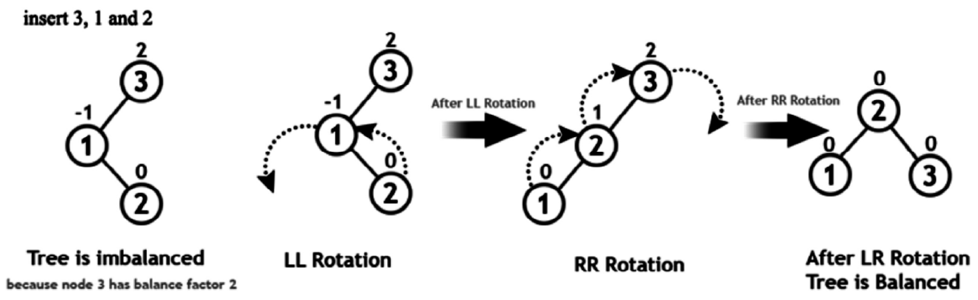
Single Right Rotation (RR Rotation)

In RR Rotation every node moves one position to right from the current position. To understand RR Rotation, let us consider following insertion operations into an AVL Tree.


insert 3, 2 and 1
Tree is imbalanced
because node 3 has balance factor 2
To make balanced we use RR Rotation which moves nodes one position to right
After RR Rotation Tree is Balanced

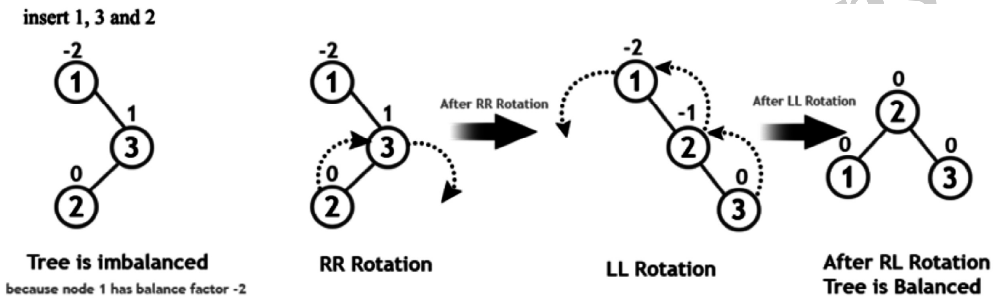
Left Right Rotation (LR Rotation)

The LR Rotation is combination of single left rotation followed by single right rotation. In LR Roration, first every node moves one position to left then one position to right from the current position. To understand LR Rotation, let us consider following insertion operations into an AVL Tree.



Right Left Rotation (RL Rotation)

The RL Rotation is combination of single right rotation followed by single left rotation. In RL Roration, first every node moves one position to right then one position to left from the current position. To understand RL Rotation, let us consider following insertion operations into an AVL Tree.



Q38. Explain the operations on an AVL tree with an example.

Ans :

Operations on an AVL Tree

The following operations are performed on an AVL tree.

1. Search
2. Insertion
3. Deletion

Search Operation in AVL Tree

In an AVL tree, the search operation is performed with O(log n) time complexity. The search operation is performed similar to Binary search tree search operation. We use the following steps to search an element in AVL tree..

- **Step 1:** Read the search element from the user
- **Step 2:** Compare, the search element with the value of root node in the tree.
- **Step 3:** If both are matching, then display "Given node found!!!" and terminate the function
- **Step 4:** If both are not matching, then check whether search element is smaller or larger than that node value.

- **Step 5:** If search element is smaller, then continue the search process in left subtree.
- **Step 6:** If search element is larger, then continue the search process in right subtree.
- **Step 7:** Repeat the same until we found exact element or we completed with a leaf node
- **Step 8:** If we reach to the node with search value, then display "Element is found" and terminate the function.
- **Step 9:** If we reach to a leaf node and it is also not matching, then display "Element not found" and terminate the function.

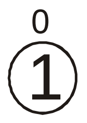
Insertion Operation in AVL Tree

In an AVL tree, the insertion operation is performed with O(log n) time complexity. In AVL Tree, new node is always inserted as a leaf node. The insertion operation is performed as follows..

- **Step 1:** Insert the new element into the tree using Binary Search Tree insertion logic.
- **Step 2:** After insertion, check the Balance Factor of every node.
- **Step 3:** If the Balance Factor of every node is 0 or 1 or -1 then go for next operation.
- **Step 4:** If the Balance Factor of any node is other than 0 or 1 or -1 then tree is said to be imbalanced. Then perform the suitable Rotation to make it balanced. And go for next operation.

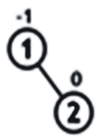
Example: Construct an AVL Tree by inserting numbers from 1 to 8.

insert 1



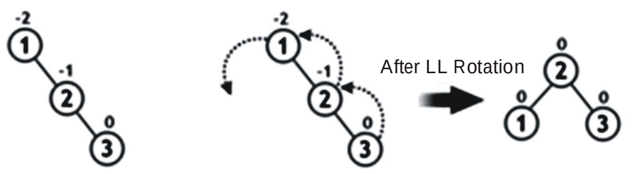
Tree is balanced

inset 2



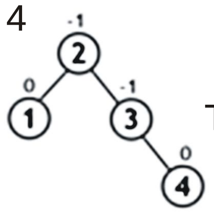
Tree is balanced

insert 3

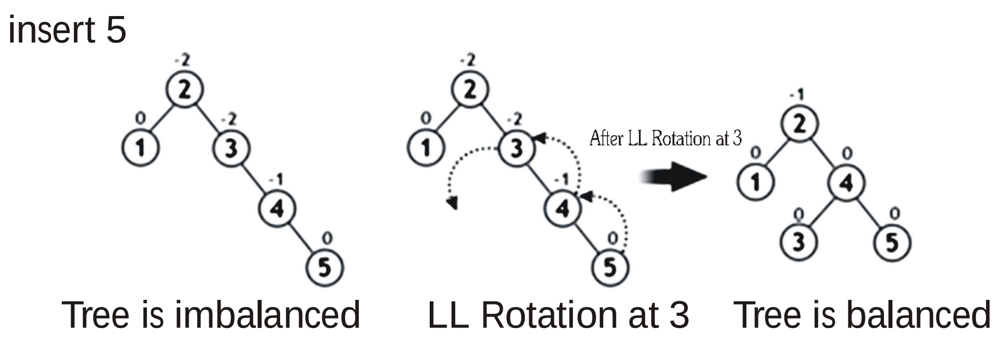


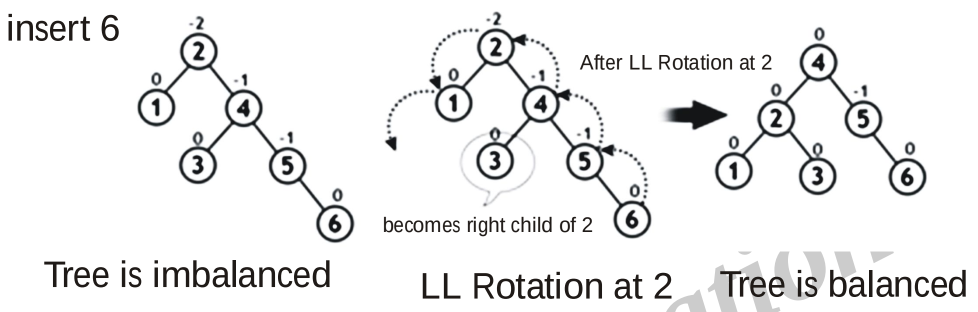
Tree is imbalanced LL Rotation Tree is balanced

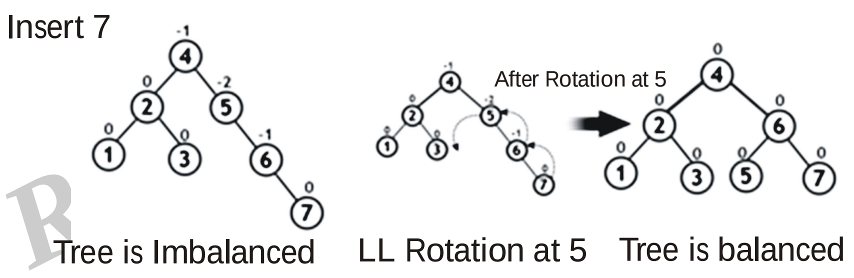
insert 4

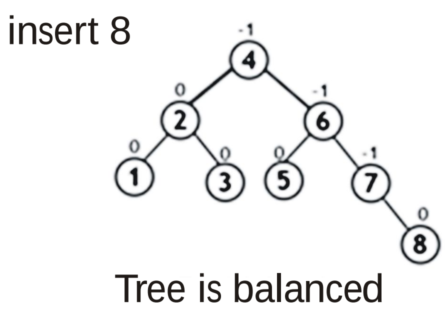


Tree is balanced









Deletion Operation in AVL Tree

In an AVL Tree, the deletion operation is similar to deletion operation in BST. But after every deletion operation we need to check with the Balance Factor condition. If the tree is balanced after deletion then go for next operation otherwise perform the suitable rotation to make the tree Balanced.

Q39. Write a C++ program to implement AVL trees

Ans :

```
#include <iostream.h>
#include <stdlib.h>
#include<constream.h>
#define FALSE 0
#define TRUE 1
struct AVLNode
{
        int data ;
        int balfact ;
        AVLNode *left ;
        AVLNode *right ;
} ;

class avltree
{
        private :
                AVLNode *root ;
        public :
                avltree( ) ;
                AVLNode*  insert ( int data, int *h ) ;
                static AVLNode* buildtree ( AVLNode *root, int data, int *h ) ;
                void display( AVLNode *root ) ;
                AVLNode* deldata ( AVLNode* root, int data, int *h ) ;
                static AVLNode* del ( AVLNode *node, AVLNode* root, int *h ) ;
                static AVLNode* balright ( AVLNode *root, int *h ) ;
                static AVLNode* balleft ( AVLNode* root, int *h ) ;
                void setroot ( AVLNode *avl ) ;
                ~avltree( ) ;
                static void deltree ( AVLNode *root ) ;
} ;
avltree :: avltree( )
{
        root = NULL ;
}
AVLNode* avltree :: insert ( int data, int *h )
{
        root = buildtree ( root, data, h ) ;
        return root ;
```

```
}
AVLNode* avltree :: buildtree ( AVLNode *root, int data, int *h )
{
    AVLNode *node1, *node2 ;

    if ( root = = NULL )
    {
        root = new AVLNode ;
        root -> data = data ;
        root -> left = NULL ;
        root -> right = NULL ;
        root -> balfact = 0 ;
        *h = TRUE ;
        return ( root ) ;
    }
    if ( data < root -> data )
    {
        root -> left = buildtree ( root -> left, data, h ) ;

        // If left subtree is higher
        if ( *h )
        {
            switch ( root -> balfact )
            {
                case 1 :
                    node1 = root -> left ;
                    if ( node1 -> balfact = = 1 )
                    {
                        cout < < "\nRight rotation." ;
                        root -> left = node1 -> right ;
                        node1 -> right = root ;
                        root -> balfact = 0 ;
                        root = node1 ;
                    }
                    else
                    {
                    cout < < "\nDouble rotation, left then right." ;
                        node2 = node1 -> right ;
                        node1 -> right = node2 -> left ;
                        node2 -> left = node1 ;
                        root -> left = node2 -> right ;
                        node2 -> right = root ;
                        if ( node2 -> balfact = = 1 )
                                root -> balfact = -1 ;
                        else
                                root -> balfact = 0 ;
                        if ( node2 -> balfact = = -1 )
                                node1 -> balfact = 1 ;
                        else
                                node1 -> balfact = 0 ;
                        root = node2 ;
                    }
                    root -> balfact = 0 ;
                    *h = FALSE ;
                    break ;

                case 0 :
                    root -> balfact = 1 ;
                    break ;
                case -1 :
                    root -> balfact = 0 ;
                    *h = FALSE ;
            }
        }
}

if ( data > root -> data )
{
        root -> right = buildtree ( root -> right, data, h ) ;

        if ( *h )
        {
            switch ( root -> balfact )
            {
                case 1 :
                    root -> balfact = 0 ;
                    *h = FALSE ;
                    break ;
                case 0 :
                    root -> balfact = -1 ;
```

```
                                break ;
                        case -1 :
                                node1 = root -> right ;
                                if ( node1 -> balfact = = -1 )
                                {
                                    cout < < "\nLeft rotation." ;
                                    root -> right = node1 -> left ;
                                    node1 -> left = root ;
                                    root -> balfact = 0 ;
                                    root = node1 ;
                                }
                                else
                                {
cout < < "\nDouble rotation, right then left." ;
                                    node2 = node1 -> left ;
                                    node1 -> left = node2 -> right ;
                                    node2 -> right = node1 ;
                                    root -> right = node2 -> left ;
                                    node2 -> left = root ;
                                    if ( node2 -> balfact = = -1 )
                                            root -> balfact = 1 ;
                                    else
                                            root -> balfact = 0 ;
                                    if ( node2 -> balfact = = 1 )
                                            node1 -> balfact = -1 ;
                                    else
                                            node1 -> balfact = 0 ;
                                    root = node2 ;
                                }
                                root -> balfact = 0 ;
                                *h = FALSE ;
                    }
                }
        }
        return ( root ) ;
}
void avltree :: display ( AVLNode* root )
{
        if ( root != NULL )
        {
          display ( root -> left ) ;
          cout < < root -> data < < "\t" ;
          display ( root -> right ) ;
     }
}
AVLNode* avltree :: deldata ( AVLNode *root, int data, int *h )
{
     AVLNode *node ;
     if ( root -> data = = 13 )
          cout < < root -> data ;
     if ( root = = NULL )
     {
          cout < < "\nNo such data." ;
          return ( root ) ;
     }
     else
     {
          if ( data < root -> data )
          {
               root -> left = deldata ( root -> left, data, h ) ;
               if ( *h )
                    root = balright ( root, h ) ;
          }
          else
          {
               if ( data > root -> data )
               {
                    root -> right = deldata ( root -> right, data, h ) ;
                    if ( *h )
                         root = balleft ( root, h ) ;
               }
               else
               {
                    node = root ;
                    if ( node -> right = = NULL )
                    {
                         root = node -> left ;
                         *h = TRUE ;
                         delete ( node ) ;
                    }
                    else
                    {
                         if ( node -> left = = NULL )
                         {
                             root = node -> right ;
                             *h = TRUE ;
                             delete ( node ) ;
                         }
                         else
                         {
                             node -> right = del ( node -> right, node, h ) ;
                             if ( *h )
                                   root = balleft ( root, h ) ;
                         }
                    }
               }
          }
     }
     return ( root ) ;
}
AVLNode* avltree :: del ( AVLNode *succ, AVLNode *node, int *h )
{
     AVLNode *temp = succ ;

     if ( succ -> left != NULL )
     {
          succ -> left = del ( succ -> left, node, h ) ;
          if ( *h )
               succ = balright ( succ, h ) ;
     }
     else
     {
          temp = succ ;
          node -> data = succ -> data ;
          succ = succ -> right ;
          delete ( temp ) ;
          *h = TRUE ;
     }
     return ( succ ) ;
}
```

```
AVLNode* avltree :: balright ( AVLNode *root, int *h )
{
        AVLNode *temp1, *temp2 ;
        switch ( root -> balfact )
        {
              case 1 :
                     root -> balfact = 0 ;
                     break ;
              case 0 :
                     root -> balfact = -1 ;
                     *h  = FALSE ;
                     break ;
              case -1 :
                     temp1 = root -> right ;
                     if ( temp1 -> balfact < = 0 )
                     {
                            cout < < "\nLeft rotation." ;
                            root -> right = temp1 -> left ;
                            temp1 -> left = root ;
                            if ( temp1 -> balfact = = 0 )
                            {
                                   root -> balfact = -1 ;
                                   temp1 -> balfact = 1 ;
                                   *h = FALSE ;
                            }
                            else
                            {
                                   root -> balfact = temp1 -> balfact = 0 ;
                            }
                            root = temp1 ;
                     }
                     else
                     {
                            cout < < "\nDouble rotation, right then left." ;
                            temp2 = temp1 -> left ;
                            temp1 -> left = temp2 -> right ;
                            temp2 -> right = temp1 ;
                            root -> right = temp2 -> left ;
                            temp2 -> left = root ;
                            if ( temp2 -> balfact = = -1 )
```

```
                        root -> balfact = 1 ;
                    else
                        root -> balfact = 0 ;
                    if ( temp2 -> balfact = = 1 )
                        temp1 -> balfact = -1 ;
                    else
                        temp1 -> balfact = 0 ;
                    root = temp2 ;
                    temp2 -> balfact = 0 ;
                }
    }
    return ( root ) ;
}
AVLNode* avltree :: balleft ( AVLNode *root, int *h )
{
    AVLNode *temp1, *temp2 ;
    switch ( root -> balfact )
    {
        case -1 :
            root -> balfact = 0 ;
            break ;

        case 0 :
            root -> balfact = 1 ;
            *h = FALSE ;
            break ;

        case 1 :
            temp1 = root -> left ;
            if ( temp1 -> balfact > = 0 )
            {
                cout < < "\nRight rotation." ;
                root -> left = temp1 -> right ;
                temp1 -> right = root ;

                if ( temp1 -> balfact = = 0 )
                {
                    root -> balfact = 1 ;
                    temp1 -> balfact = -1 ;
                    *h = FALSE ;
                    }
                    else
                    {
                          root -> balfact = temp1 -> balfact = 0 ;
                    }
                    root = temp1 ;
              }
              else
              {
                    cout << "\nDouble rotation, left then right." ;
                    temp2 = temp1 -> right ;
                    temp1 -> right = temp2 -> left ;
                    temp2 -> left = temp1 ;
                    root -> left = temp2 -> right ;
                    temp2 -> right = root ;
                    if ( temp2 -> balfact == 1 )
                          root -> balfact = -1 ;
                    else
                          root -> balfact = 0 ;
                    if ( temp2-> balfact == -1 )
                          temp1 -> balfact = 1 ;
                    else
                          temp1 -> balfact = 0 ;
                    root = temp2 ;
                    temp2 -> balfact = 0 ;
              }
      }
      return ( root ) ;
}
void avltree :: setroot ( AVLNode *avl )
{
      root = avl ;
}
avltree :: ~avltree( )
{
      deltree ( root ) ;
}
void avltree :: deltree ( AVLNode *root )
{
      if ( root != NULL )
```

```
        {
                deltree ( root -> left ) ;
                deltree ( root -> right ) ;
        }
        delete ( root ) ;
}
void main( )
{
        avltree at ;
        AVLNode *avl = NULL ;
        int h ;
        clrscr();
        avl = at.insert ( 20, &h ) ;
        at.setroot ( avl ) ;
        avl = at.insert ( 6, &h ) ;
        at.setroot ( avl ) ;
        avl = at.insert ( 29, &h ) ;
        at.setroot ( avl ) ;
        avl = at.insert ( 5, &h ) ;
        at.setroot ( avl ) ;
        avl = at.insert ( 12, &h ) ;
        at.setroot ( avl ) ;
        avl = at.insert ( 25, &h ) ;
        at.setroot ( avl ) ;
        avl = at.insert ( 32, &h ) ;
        at.setroot ( avl ) ;
        avl = at.insert ( 10, &h ) ;
        at.setroot ( avl ) ;
        avl = at.insert ( 15, &h ) ;
        at.setroot ( avl ) ;
        avl = at.insert ( 27, &h ) ;
        at.setroot ( avl ) ;
        avl = at.insert ( 13, &h ) ;
        at.setroot ( avl ) ;
        cout << endl << "AVL tree:\n" ;
        at.display ( avl ) ;
        avl = at.deldata ( avl, 20, &h ) ;
        at.setroot ( avl ) ;
        avl = at.deldata ( avl, 12, &h ) ;
        at.setroot ( avl ) ;
        cout << endl << "AVL tree after deletion
        of a node:\n" ;
        at.display ( avl ) ;
        getch();
}
```

Short Question & Answers

1. What is binary tree?

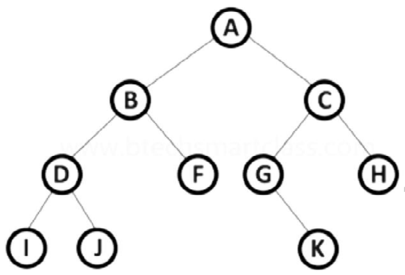
Ans :

In a normal tree, every node can have any number of children. Binary tree is a special type of tree data structure in which every node can have a maximum of 2 children. One is known as left child and the other is known as right child.

A tree in which every node can have a maximum of two children is called as Binary Tree.

In a binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2 children.

Example



2. Write a C++ program for creation and traversal of a Binary Tree

Ans :

```
#include<iostream.h>
#include<conio.h>
#include<process.h>
struct tree_node
{
        tree_node *left;
        tree_node *right;
        int data;
} ;
class bst
{
        tree_node *root;
        public:
        bst()
        {
                root=NULL;
        }
        int isempty()
        {
                return(root= =NULL);
        }
        void insert(int item);
        void inordertrav();
        void inorder(tree_node *);
        void postordertrav();
        void postorder(tree_node *);
        void preordertrav();
        void preorder(tree_node *);
};
void bst::insert(int item)
{
        tree_node *p=new tree_node;
        tree_node *parent;
        p->data=item;
        p->left=NULL;
        p->right=NULL;
        parent=NULL;
        if(isempty())
                root=p;
        else
        {
                tree_node *ptr;
                ptr=root;
                while(ptr!=NULL)
                {
                        parent=ptr;
                        if(item>ptr->data)
                                ptr=ptr->right;
```

```
            else
                  ptr=ptr->left;
        }
        if(item<parent->data)
            parent->left=p;
        else
            parent->right=p;
    }
}
void bst::inordertrav()
{
    inorder(root);
}
void bst::inorder(tree_node *ptr)
{
    if(ptr!=NULL)
    {
        inorder(ptr->left);
        cout<<" "<<ptr->data<<"    ";
        inorder(ptr->right);
    }
}
void bst::postordertrav()
{
    postorder(root);
}
void bst::postorder(tree_node *ptr)
{
    if(ptr!=NULL)
    {
        postorder(ptr->left);
        postorder(ptr->right);
        cout<<" "<<ptr->data<<"    ";
    }
}
void bst::preordertrav()
{
    preorder(root);
}
void bst::preorder(tree_node *ptr)
{
    if(ptr!=NULL)
    {
        cout<<" "<<ptr->data<<"    ";
        preorder(ptr->left);
        preorder(ptr->right);
    }
}
void main()
{
    bst b;
    b.insert(52);
    b.insert(25);
    b.insert(50);
    b.insert(15);
    b.insert(40);
    b.insert(45);
    b.insert(20); cout<<"inorder"<<endl;
    b.inordertrav();
    cout<<endl<<"postorder"<<endl;
    b.postordertrav();
    cout<<endl<<"preorder"<<endl;
    b.preordertrav();
    getch();
}
```

3. Explain ADT of Binary Tree

Ans :

A binary tree consists of nodes with one predecessor and at most two successors (called the left child and the right child). The only exception is the root node of the tree that does not have a predecessor. A node can contain any amount and any type of data.

1. **create:** A binary tree may be created in several states. The most common include.
   - an empty tree (i.e. no nodes) and hence the constructor will have no parameters
   - a tree with one node (i.e. the root) and hence the constructor will have a single parameter (the data for the root node)
   - a tree with a new root whose children are other existing trees and hence the

constructor will have three parameters (the data for the root node and references to its subtrees)

2. **isEmpty():** Returns true if there are no nodes in the tree, false otherwise.

   - **isFull():** May be required by some implementations. Returns true if the tree is full, false otherwise.

3. **clear():** Removes all of the nodes from the tree (essentially reinitializing it to a new empty tree).

4. **add(value):** Adds a new node to the tree with the given value. The actual implementation of this method is determined by the purpose for the tree and how the tree is to be maintained and/or processed. To begin with, we will assume no particular purpose or order and therefore add new nodes in such a way that the tree will remain nearly balanced.

5. **remove():** Removes the root node of the tree and returns its data. The actual implementation of this method and other forms of removal will be determined by the purpose for the tree and how the tree is to be maintained and/or processed. For example, you might need a remove() method with a parameter that indicates which node of the tree is to be removed (based on position, key data, or reference). To begin with, we will assume no particular purpose or order and therefore remove the root in such a way that the tree will remain nearly balanced.

   Other operations that may be included as needed:

- **height():** Determines the height of the tree. An empty tree will have height 0.
- **size():** Determines the number of nodes in the tree.
- **getRootData():** Returns the data (primative data) or reference to the data (objects) of the tree's root.
- **getLeftSubtree():** Returns a reference to the left subtree of this tree.
- **getRightSubtree():** Returns a reference to the right subtree of this tree.

4. What is Search ?

Ans :

Search is a process of finding a value in a list of values. In other words, searching is the process of locating given value position in a list of values.

Searching can be done on internal data structures or on external data structures. Information retrieval in the required format is the central activity in all computer applications.

Basic Searching Techniques

Consider a list of n elements or can represent a file of n records, where each element is a key number. The task is to find a particular key in the list in the shortest possible time. If you know you are going to search for an item in a set, you will need to think carefully about what type of data structure you will use for that set. At low level, the only searches that get mentioned are for sorted and unsorted arrays. However, these are not the only data types that are useful for searching.

1. **Linear search**

   Start at the beginning of the list and check every element of the list. Very slow [order O(n)] but works on an unsorted list.

2. **Binary Search**

   This is used for searching in a sorted array. Test the middle element of the array. If it is too big. Repeat the process in the left half of the array, and the right half if it's too small. In this way, the amount of space that needs to be searched is halved every time, so the time is O(log n).

3. **Hash Search**

   Searching a hash table is easy and extremely fast, just find the hash value for the item you're looking for then go to that index and start searching the array until you find what you are looking for or you hit a blank spot. The order is pretty close to o(1), depending on how full your hash table is.

4. **Binary Tree search**

   Search a binary tree is just as easy as searching a hash table, but it is usually slower (especially if the tree is badly unbalanced). Just start at the root. Then go down the left subtree if the root is too big and the right subtree if is too small. Repeat until you find what you want or the sub-tree you want isn't there. The running time is O(log n) on average and O(n) in the worst case.

5. Compare various sorting techniques with real world usage.

Ans :

Selection sort is an exception in our list. This is considered an academic sorting algorithm. Because the time efficiency is always O(n^2) which is not acceptable. There is no real world usage for selection sort except passing the data structure course exam.

Pros

- Nothing

cons

- Always run at O(n^2) even at best case scenario

Bubble sort

This is the other exception in the list because bubble sort is too slow to be practical. Unless the sequence is almost sorted feasibility of bubble sort is zero and the running time is O(n^2). This is one of the three simple sorting algorithms alongside selection sort and insertion sort but like selection sort falls short of insertions sort in terms of efficiency even for small sequences.

Pros

- Again nothing, maybe just "catchy name[1]"

Cons

- With polynomial O(n^2) it is too slow

Insertion sort

Insertion sort is definitely not the most efficient algorithm out there but its power lies in its simplicity. Since it is very easy to implement and adequately efficient for small number of elements, it is useful for small applications or trivial ones. The definition of small is vague and depends on a lot of things but a safe bet is if under 50, insertion sort is fast enough. Another situation that insertion sort is useful is when the sequence is almost sorted. Such sequences may seem like exceptions but in real world applications often you encounter almost sorted elements. The run time of insertions sort is O(n^2) at worst case scenario. So far we have another useless alternative for selection sort. But if implemented well the run time can be reduced to O(n+k). n is the number of elements and k is the number of inversions (the number of pair of elements out of order). With this new run time in mind you can see if the sequence is almost sorted (k is small) the run time can be almost linear which is a huge improvement over the polynomial n^2.

Pros

- Easy to implement
- The more the sequence is ordered the closer is run time to linear time O(n)

Cons

- Not suitable for large data sets
- Still polynomial at worst case

6. Apply the bubble sort algorithm to sort the list of data: 2, 10, 6, 4, 8.

Ans :

Bubble Sort

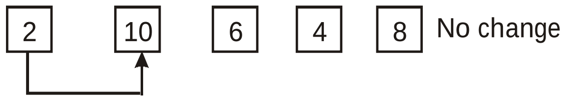
The given list of data is 2, 10, 6, 4, 8

Consider the following list of data 2, 10, 6, 4, 8

Iteration 1

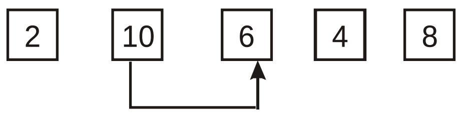
Step 1

Compare the first two elements since, 2 < 10, exchanging of element's position is not required.

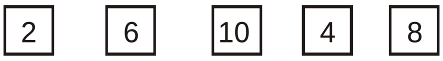


Step 2

Compare the next two elements

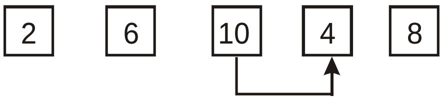


Since 10 > 6, exchange position of 10 with position of element 6.

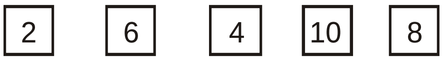


Step 3

Compare element 10 with element 4

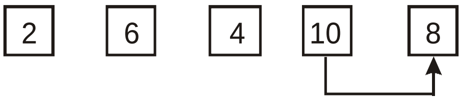


Since, 10 > 4, exchange position of 10 with position of 4.

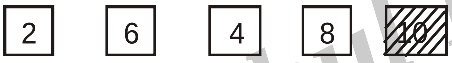


Step 4

Compare element 10 and 8



Since,10 > 8 exchange position of 10 with position of 8.

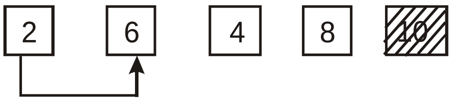


As the end of array is reached the iteration 1 terminates. the last element 10 does not require further processing.

Iteration 2

Step 1

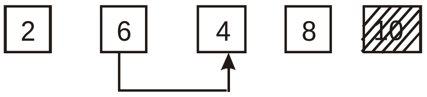
Compare element 2 with 6



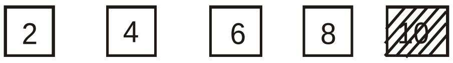
Since, 2< 6 exchanging of an element's position is not required.

Step 2

Compare element 6 with 4

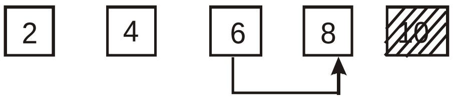


Since, 6 > 4 exchange position of 4 with position of 6

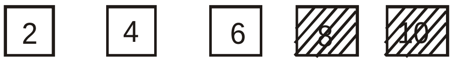


Step 3

Compare the element with 8



Since, 6 < 8 exchanging of an element's position is not required.

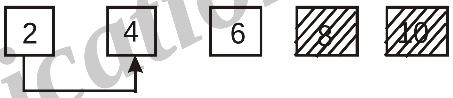


This is end of iteration 2, 8 is patched before with the element 10.

Iteration 3

Step 1

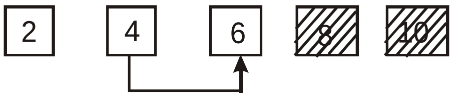
Compare element 2 with element 4



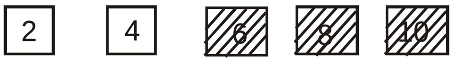
Since, 2 < 4 exchanging of an element's position is not required.

Step 2

Compare element 4 with 6



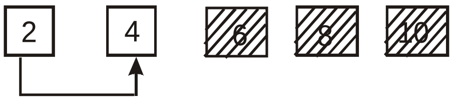
Since, 4 < 6 exchanging of an element's position is not required.



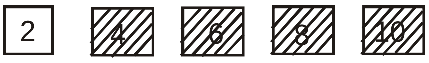
This is end of the iteration 3, the element 6 is patched before the element 8.

Iteration 4

Compare element 2 with 4



Since, 2 < 4 exchanging of an element's position is not required.



This is end of the iteration 4, the element 4 is patched before the element 6.

Since, the only element left is 2, it is patched before 4.

Finally, the completely sorted list of elements obtained is given below,

2	4	6	8	10

7. Construct a binary tree using the following two traversals.

Inorder	**D**	**B**	**H**	**E**	**A**	**I**	**F**	**J**	**C**	**G**
Preorder	**A**	**B**	**D**	**E**	**H**	**C**	**F**	**I**	**J**	**G**

Ans :

Given that,

Inorder	D	B	H	E	A	I	F	J	C	G
Preorder	A	B	D	E	H	C	F	I	J	G

In preorder traversal of a binary tree, the root node is traversed first. So, the first value in the preorder traversal gives the root of the binary tree. Then the left subtree followed by the right subtree are traversed.

In the inorder traversal, the left subtree is traversed initially and then the root is visited. Finally, the right subtree is traversed.

A binary tree can be constructed from the given preorder and inorder traversals. Consider the above given data, where the first element in preorder is 'A', so the element 'A' is the root node. The elements before 'A' in the inorder list i.e., (DBHE) becomes the left subtree and the elements after 'A' in the inorder list i.e., (1FJCG) becomes the right subtree. This can be shown in figure (a).

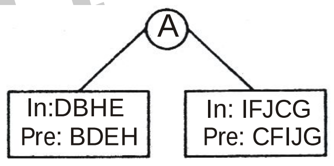


Figure (a)

In the left subtree, the first element in preorder list is 'B'. So, 'B' becomes the root node. The elements on the left side of 'B' in the inorder list i.e., (D) becomes the left subtree and the elements on the right side of B in the inorder list i.e., (HE) becomes the right subtree. This can be shown in figure (b).

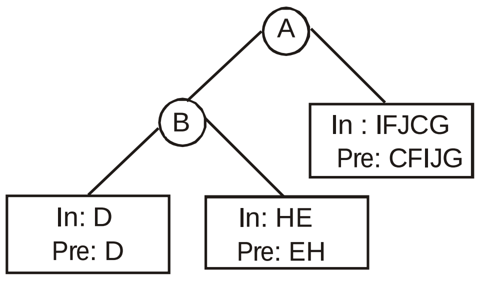


Figure (b)

Now, consider the left subtree of node B, the only element in preorder is ‘D’ So, ‘D’ becomes the leaf node. But, in the inorder list, there is only element i.e., ‘D’. So both left subtree and right subtree cannot be formed. This can be shown in figure (c).

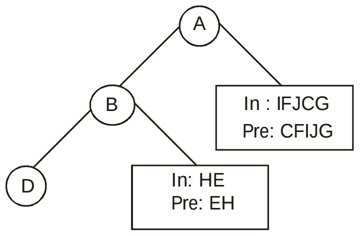


Figure (c)

Now, consider the right subtree of node ‘B’, the first element in preorder list is E. So, ‘E’ becomes the root mode of the right subtree. In the inorder list, the elements on the leftside of ‘E’ i.e., (H) becomes the left subtree of ‘E’ and since there are no elements on the rightsides of’E’, there is no right subtree for ‘E’. This can be shown in figure (d).

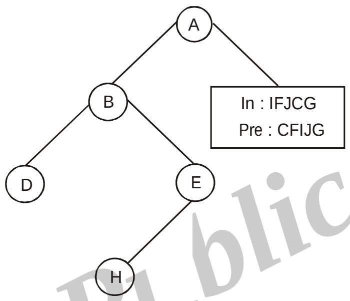


Figure (d)

Hence, the left subtree of node ‘A’ is completed.

Now, consider the right subtree of root node ‘A’. In the right subtree, the first element in preorder is ‘C’, So ‘C’ becomes the root node. In the inorder list, the elements on leftside of‘C’ i.e., (IFJ) becomes left subtree and the element on rightside of‘C’ i.e., ‘G’ becomes right subtree. This can be shown in figure (e).

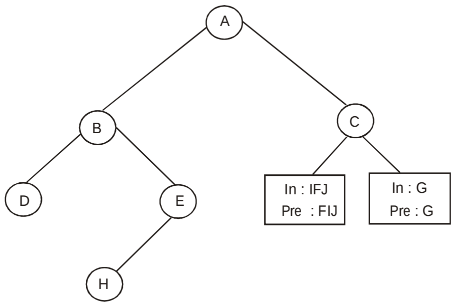


Figure (e)

Now, consider the left subtree of node ‘C’. The first element in the preorder i.e., ‘F’ becomes root node. In the inorder list, the elements on the leftside of “F’ i.e., (I) becomes left subtree and the elements on the rightside of ‘F’ i.e., (J) becomes right subtree. This can be shown in figure (f).

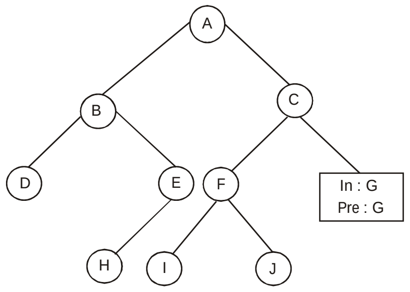


Figure (f)

Now, consider the right subtree of node ‘C’, the only element in preorder is ‘G’ so it becomes leaf node. In the inorder list, there is only one element i.e., ‘G’. So both left subtree and right subtree cannot be formed. This can be shown in figure (g).

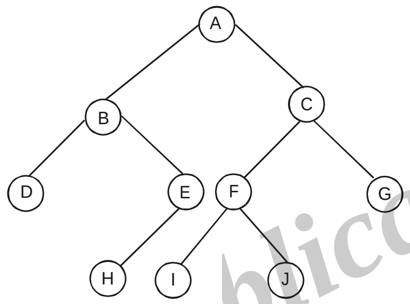


Figure (g)

Hence, the right subtree of node ’A’ is completed and figure (g) gives the final binary tree constructed for the given preorder and inorder lists.

8. Define internal and external sorting.

Ans :

Internal Sorting : If the Input data is such that it can be adjusted in the main memory.

External Sorting : If the input data is such that it cannot be adjusted in the memory entirely at once, It needs to be stored is external memory.

9. What are the pros of binary search.

Ans :

The binary search algorithm is more efficient than the linear search algorithm because it takes less time to search through the list. It has a logarithmic relationship between the number of elements (N) in the list, and the number of comparisons required (C), given by the following formula.

10. What are the applications of trees?

Ans :

Some of the applications of trees are,

1. Manipulation of arithmetic expression
2. Constructing symbol table
3. Maintaining symbol table

Choose the Correct Answers

1. The operations of processing each element in the list is known as [d]

   (a) Sorting (b) Merging
   (c) Inserting (d) Traversal

2. The other name for directed graph is ________. [b]

   (a) Directed Graph (b) Digraph
   (c) Dirgraph (d) Dia-Graph

3. Binary trees with thread are called as ________. [a]

   (a) Threaded Tree (b) Pointer Tree
   (c) Special Tree (d) Special Pointer Tree

4. A terminal node in a binary tree is called ________. [b]

   (a) Root (b) Leaf
   (c) Inserting (d) Branch

5. Which indicated Pre-Order Traversal [c]

   (a) Left SubTree, Right SubTree, Root (b) Right SubTree, Left SubTree, Root
   (c) Root, Left SubTree Right SubTree (d) Right SubTree, Root, Left SubTree

6. Linked Representation of binary tree need ________ Practical Arrays [c]

   (a) 4 (b) 2
   (c) 3 (d) 5

7. Graphs are represented using [b]

   (a) Adjacency Tree (b) Adjacency Linked List/ Matrix
   (c) AdjacencyGraph (d) Adjacency Queue

8. The spanning tree of connected graph with 10 vertices contain ________ [a]

   (a) 9 Edeges (b) 11 Edeges
   (c) 10 Edeges (d) 9 Vertices

9. In Binary tree nodes with no successor are called [b]

   (a) End Nodes (b) Terminal Nodes
   (c) Final Nodes (d) Last Nodes

10. Which of the following tree traversal visit root node last [a]

    (a) Post Order (b) In-Order
    (c) none (d) Pre-Order

Fill in the Blanks

1. ________ is a nonlinear data structure which organizes data in hierarchical structure.
2. If we have N number of nodes then we have a maximum of ________ number of links.
3. The connecting links between any two nodes in a tree is called ________.
4. A ________ tree can have maximum of 2 childeren.
5. A full binary tree is also known as ________.
6. Visiting order of node in a binary tree is called as ________.
7. Graph is nonlinear data structure which contains set of points known as ________ and ________.
8. A Graph with un directed edges is called ________.
9. A graph is said to be ________ graph if all its vertices have the same degree.
10. DFS and BFS traversal of a graph results ________ tree.
11. We use ________ Data structure to implement minimum spanning tree.
12. ________ is used to index and retrieve elements.

ANSWERS

1. Tree
2. N-1
3. Edge
4. Binary
5. Proper Binary Tree or 2 - Tree
6. Binary Tree Traversal
7. Nodes, Links
8. Undirected Graphs
9. Regular
10. Spanning
11. Queue
12. Hasing

One Mark Answers

1. Define a tree.

Ans :

A ‘tree’ is a non-linear data structure represented in a hierarchical manner. It contains a finite set of elements called ‘node”.

2. Define static tree table.

Ans :

The static tree table is a structure in which operations like insertion and deletion are not allowed and symbols are already known.

3. Define binary tree.

Ans :

A binary tree ‘T’ is defined by finite set of elements called nodes such that,

(a) A binary tree is empty or

(b) It consists of a node called ‘root’ which includes two binary trees called left subtree and right subtree of the root.

4. Define searching.

Ans :

Searching is a process done to know the correct location of an element from a list or array of elements. In an array, elements are stored in consecutive memory locations. Searching is done by comparing a particular element with the remaining elements until the exact match is found.

UNIT IV

Graphs: Introduction, Representation of Graphs, Graph Traversal – Depth First Search, Breadth First Search, Spanning Tree, Prim's Algorithm, Kruskal's Algorithm.

Hashing: Introduction, Key Terms and Issues, Hash Functions, Collision Resolution Strategies, Hash Table Overflow, Extendible Hashing

Heaps: Basic Concepts, Implementation of Heap, Heap as Abstract Data Type, Heap Sort, Heap Applications.

4.1 Graphs

4.1.1 Introduction

Q1. What is graph? Discuss the terminology of graph.

Ans :

Graph is a non linear data structure, it contains a set of points known as nodes (or vertices) and set of links known as edges (or Arcs) which connects the vertices. A graph is defined as follows:

Graph is a collection of vertices and arcs which connects vertices in the graph

Graph is a collection of nodes and edges which connects nodes in the graph

Generally, a graph G is represented as G = (V , E), where V is set of vertices and E is set of edges.

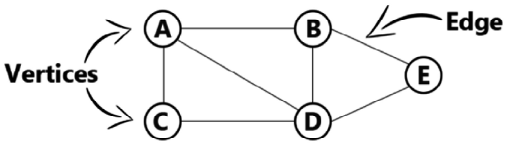
Example

The following is a graph with 5 vertices and 6 edges. This graph G can be defined as G = (V , E)

Where,

V = {A,B,C,D,E} and

E = {(A,B),(A,C)(A,D),(B,D), (C,D),(B,E), (E,D)}.



We use the following terms in graph data structure.

Vertex

A individual data element of a graph is called as Vertex. Vertex is also known as node. In above example graph, A, B, C, D & E are known as vertices.

Edge

An edge is a connecting link between two vertices. Edge is also known as Arc. An edge is represented as (startingVertex, endingVertex). For example, in above graph, the link between vertices A and B is represented as (A,B). In above example graph, there are 7 edges (i.e., (A, B), (A, C), (A,D), (B, D), (B, E), (C,D), (D, E)).

Edges are three types.

1. **Undirected Edge**: An undirected egde is a bidirectional edge. If there is a undirected edge between vertices A and B then edge (A, B) is equal to edge (B, A).

2. **Directed Edge:** A directed egde is a unidirectional edge. If there is a directed edge between vertices A and B then edge (A , B) is not equal to edge (B, A).

3. **Weighted Edge:** A weighted egde is an edge with cost on it.

Undirected Graph

A graph with only undirected edges is said to be undirected graph.

Directed Graph

A graph with only directed edges is said to be directed graph.

Mixed Graph

A graph with undirected and directed edges is said to be mixed graph.

End vertices or Endpoints

The two vertices joined by an edge are called the end vertices (or endpoints) of the edge.

Origin

If an edge is directed, its first endpoint is said to be origin of it.

Destination

If an edge is directed, its first endpoint is said to be origin of it and the other endpoint is said to be the destination of the edge.

Adjacent

If there is an edge between vertices A and B then both A and B are said to be adjacent. In other words, Two vertices A and B are said to be adjacent if there is an edge whose end vertices are A and B.

Incident

An edge is said to be incident on a vertex if the vertex is one of the endpoints of that edge.

Outgoing Edge

A directed edge is said to be outgoing edge on its orign vertex.

Incoming Edge

A directed edge is said to be incoming edge on its destination vertex.

Degree

Total number of edges connected to a vertex is said to be degree of that vertex.

Indegree

Total number of incoming edges connected to a vertex is said to be indegree of that vertex.

Outdegree

Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.

Parallel edges or Multiple edges

If there are two undirected edges to have the same end vertices, and for two directed edges to have the same origin and the same destination. Such edges are called parallel edges or multiple edges.

Self-loop

An edge (undirected or directed) is a self-loop if its two endpoints coincide.

Simple Graph

A graph is said to be simple if there are no parallel and self-loop edges.

Path

A path is a sequence of alternating vertices and edges that starts at a vertex and ends at a vertex such that each edge is incident to its predecessor and successor vertex.

4.1.2 Representation of Graphs

Q2. What are the various ways to represent graphs?

Ans : **(Dec.-19, Dec.-18, Nov.-17)**

Graph Representations

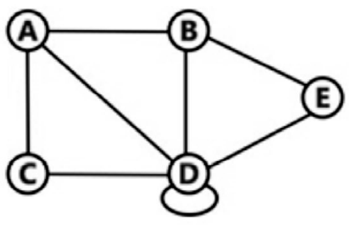
Graph data structure is represented using following representations...

1. Adjacency Matrix
2. Incidence Matrix
3. Adjacency List

1. Adjacency Matrix

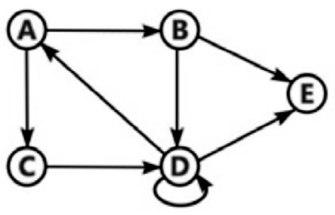
In this representation, graph can be represented using a matrix of size total number of vertices by total number of vertices. That means if a graph with 4 vertices can be represented using a matrix of 4X4 class. In this matrix, rows and columns both represents vertices. This matrix is filled with either 1 or 0. Here, 1 represents there is an edge from row vertex to column vertex and 0 represents there is no edge from row vertex to column vertex.

For example, consider the following undirected graph representation...



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 1 & 0 \\ B & 1 & 0 & 0 & 1 & 1 \\ C & 1 & 0 & 0 & 1 & 0 \\ D & 1 & 1 & 1 & 1 & 1 \\ E & 0 & 1 & 0 & 1 & 0 \end{array}$$

Directed graph representation...

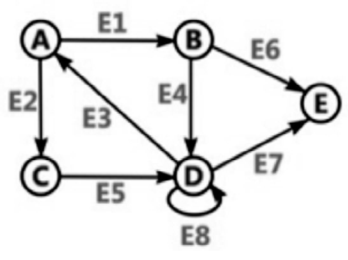


$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 0 & 0 \\ B & 0 & 0 & 0 & 1 & 1 \\ C & 0 & 0 & 0 & 1 & 0 \\ D & 1 & 0 & 0 & 1 & 1 \\ E & 0 & 0 & 0 & 0 & 0 \end{array}$$

2. Incidence Matrix

In this representation, graph can be represented using a matrix of size total number of vertices by total number of edges. That means if a graph with 4 vertices and 6 edges can be represented using a matrix of 4X6 class. In this matrix, rows represents vertices and columns represents edges. This matrix is filled with either 0 or 1 or -1. Here, 0 represents row edge is not connected to column vertex, 1 represents row edge is connected as outgoing edge to column vertex and -1 represents row edge is connected as incoming edge to column vertex.

For example, consider the following directed graph representation...

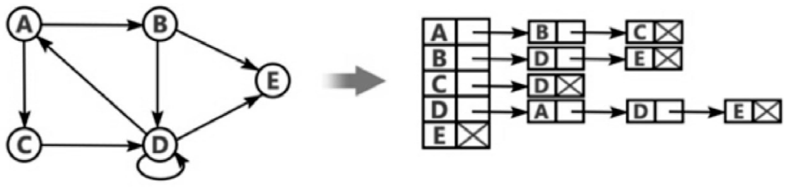


$$\begin{array}{c|cccccccc} & E1 & E2 & E3 & E4 & E5 & E6 & E7 & E8 \\ \hline A & 1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ B & -1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ C & 0 & -1 & 0 & 0 & 1 & 0 & 0 & 0 \\ D & 0 & 0 & 1 & -1 & -1 & 0 & 1 & 1 \\ E & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 \end{array}$$

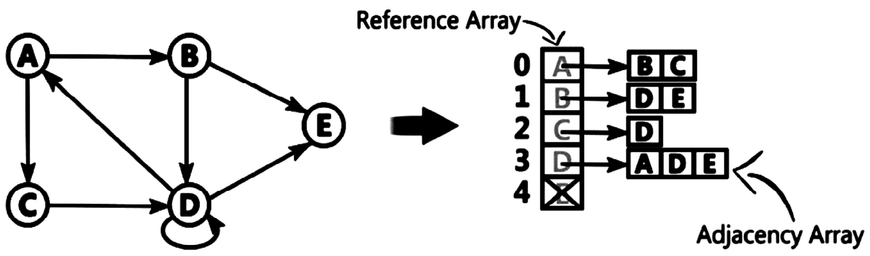
3. Adjacency List

In this representation, every vertex of graph contains list of its adjacent vertices.

For example, consider the following directed graph representation implemented using linked list...



This representation can also be implemented using array as follows..



4.2 Graph Traversal

4.2.1 Depth First Search

Q3. Explain DFS algorithm with an example.

Ans : **(June-18)**

DFS (Depth First Search)

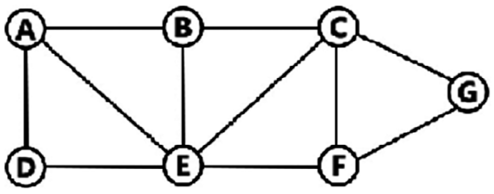
DFS traversal of a graph, produces a spanning tree as final result. Spanning Tree is a graph without any loops. We use Stack data structure with maximum size of total number of vertices in the graph to implement DFS traversal of a graph.

We use the following steps to implement DFS traversal...

- **Step 1:** Define a Stack of size total number of vertices in the graph.
- **Step 2:** Select any vertex as starting point for traversal. Visit that vertex and push it on to the Stack.
- **Step 3:** Visit any one of the adjacent vertex of the verex which is at top of the stack which is not visited and push it on to the stack.
- **Step 4:** Repeat step 3 until there are no new vertex to be visit from the vertex on top of the stack.
- **Step 5:** When there is no new vertex to be visit then use back tracking and pop one vertex from the stack.
- **Step 6:** Repeat steps 3, 4 and 5 until stack becomes Empty.
- **Step 7:** When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph Back tracking is coming back to the vertex from which we came to current vertex.

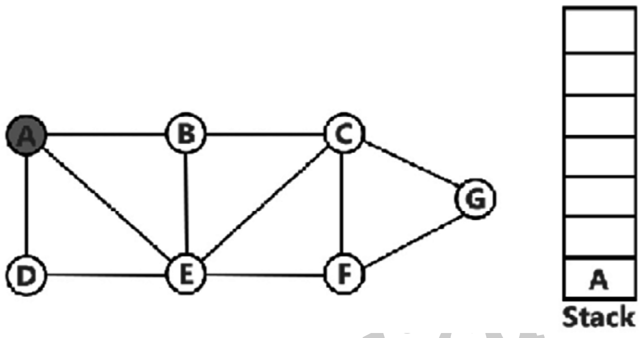
Example

Consider the following example graph to perform UPS traversal



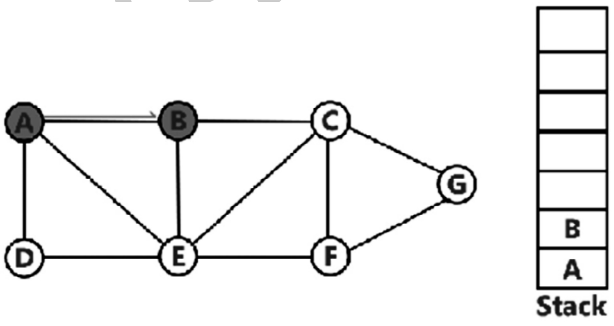
Step 1:

- Select the vertex A as starting point (visit A).
- Push A on to the Stack



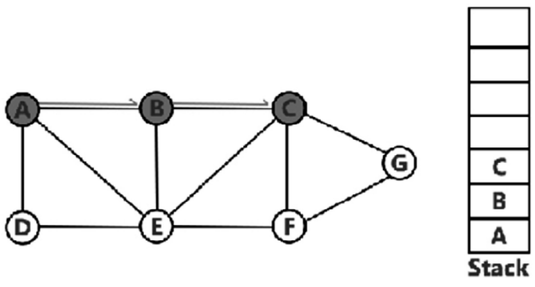
Step 2:

- Visit any adjacement vertex of A which is not visited(B).
- Push newly visited vertex B on to the Stack



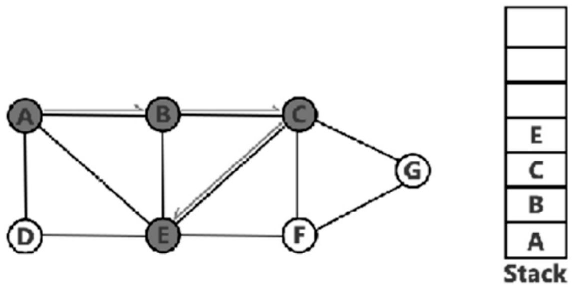
Step 3:

- Visit any adjacement vertex of B which is not visited(C)
- Push C on to the Stack



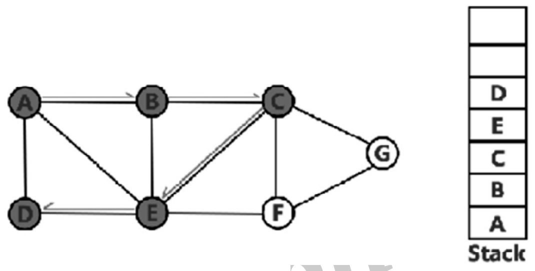
Step 4:

- Visit any adjacement vertex of C which is not visited(E)
- Push E on to the Stack



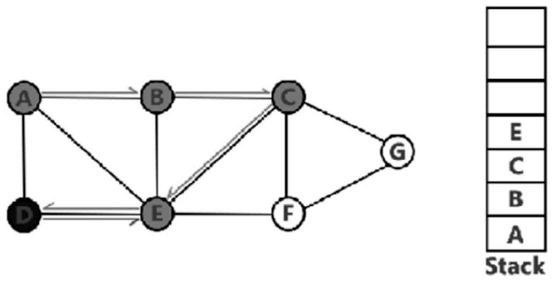
Step 5:

- Visit any adjacement vertex of E which is not visited(D)
- Push D on to the Stack



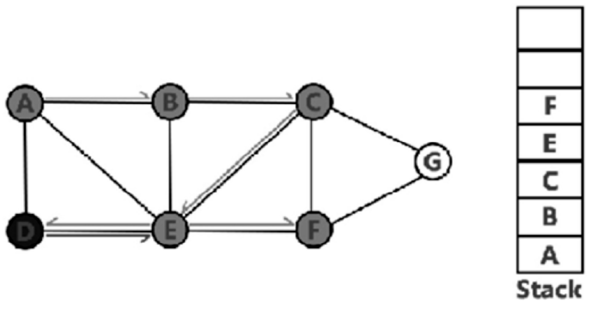
Step 6:

- There is no new vertiex to be visited from D. So use back track
- Pop D from the Stack



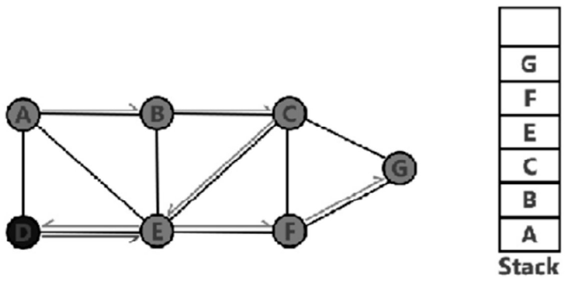
Step 7:

- Visit any adjacement vertex of E which is not visited(F)
- Push F on to the Stack



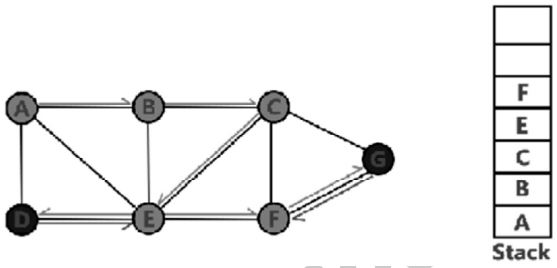
Step 8:

- Visit any adjacement vertex of F which is not visited(G)
- Push G on to the Stack



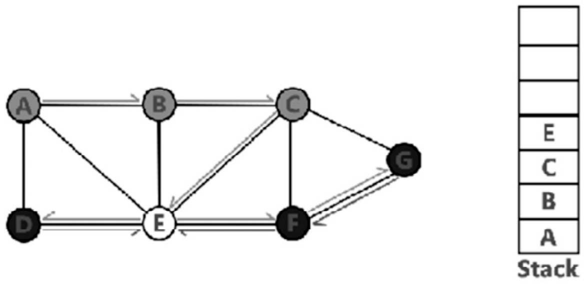
Step 9:

- There is no new vertiex to be visited from G. So use back track
- Pop G from the Stack



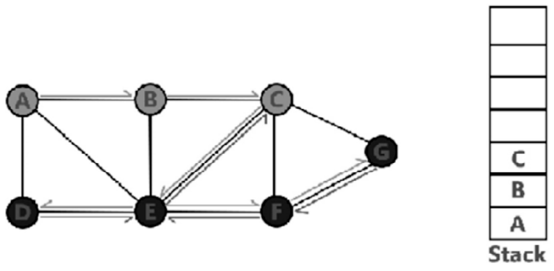
Step 10:

- There is no new vertiex to be visited from F. So use back track
- Pop F from the Stack



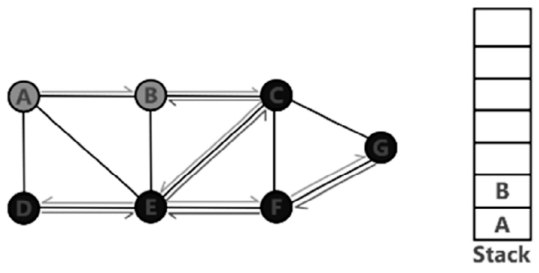
Step 11:

- There is no new vertiex to be visited from E. So use back track
- Pop F from the Stack



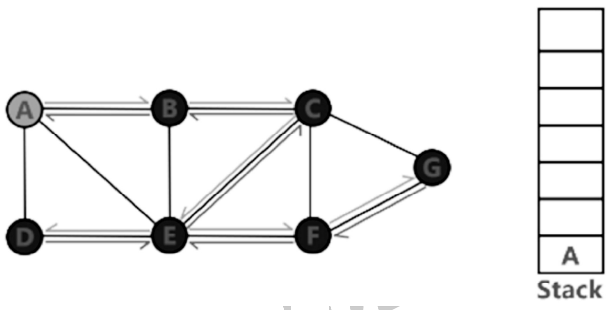
Step 12:

- There is no new vertiex to be visited from C. So use back track
- Pop C from the Stack



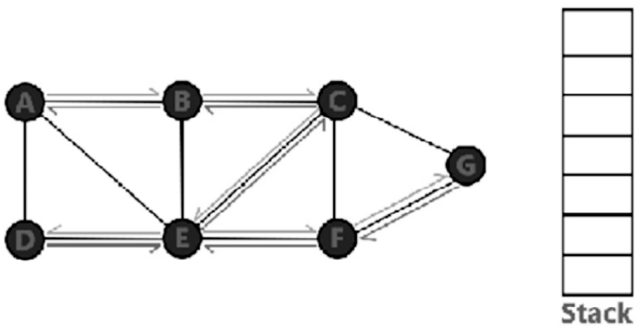
Step 13:

- There is no new vertiex to be visited from B. So use back track
- Pop B from the Stack



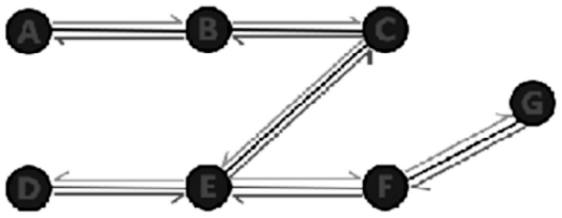
Step 14:

- There is no new vertiex to be visited from A. So use back track
- Pop A from the Stack



Step 15:

- Stack became Empty. So stop DFS Treversa
- Final result of DFS traversa is following spanning tre



Q4. Write a C++ program to IMPLEMENT DFS

Ans :

```
#include<iostream>
#include<conio.h>
#include<stdlib.h>
using namespace std;
intcost[10][10],i,j,k,n,stk[10],top,v, visit[10], visited[10];
main()
{
int m;
cout <<"enterno of vertices";
cin >> n;
cout <<"ente no of edges";
cin >> m;
cout <<"\nEDGES \n";
for(k=1;k<=m;k++)
{
cin >>i>>j;
cost[i][j]=1;
}
cout <<"enter initial vertex";
cin >>v;
cout <<"ORDER OF VISITED VERTICES";
cout << v <<" ";
visited[v]=1;
k=1;
while(k<n)
{
for(j=n;j>=1;j—)
if(cost[v][j]!=0 && visited[j]!=1 && visit[j]!=1)
{
visit[j]=1;
stk[top]=j;
top++;
}
v=stk[—top];
cout<<v << " ";
k++;
visit[v]=0; visited[v]=1;
}
}
```

OUTPUT

```
enterno of vertices9
ente no of edges9
EDGES
1 2
2 3
2 6
1 5
1 4
4 7
5 7
7 8
8 9
enter initial vertex1
ORDER OF VISITED VERTICES1 2 3 6 4 7 8 9 5
```

4.2.2 Breadth First Search

Q5. Explain about BFS algorithm with an example.

Ans : **(June-18)**

BFS (Breadth First Search)

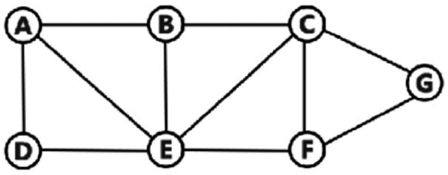
BFS traversal of a graph, produces a spanning tree as final result. Spanning Tree is a graph without any loops. We use Queue data structure with maximum size of total number of vertices in the graph to implement BFS traversal of a graph.

We use the following steps to implement BFS traversal.

- **Step 1:** Define a Queue of size total number of vertices in the graph.
- **Step 2:** Select any vertex as starting point for traversal. Visit that vertex and insert it into the Queue.
- **Step 3:** Visit all the adjacent vertices of the verex which is at front of the Queue which is not visited and insert them into the Queue.
- **Step 4:** When there is no new vertex to be visit from the vertex at front of the Queue then delete that vertex from the Queue.
- **Step 5:** Repeat step 3 and 4 until queue becomes empty.
- **Step 6:** When queue becomes Empty, then produce final spanning tree by removing unused edges from the graph

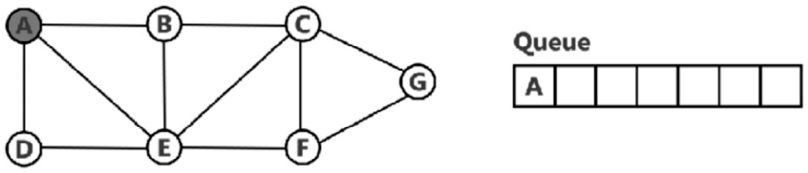
Example

Consider the following example graph to perform BFS traversal



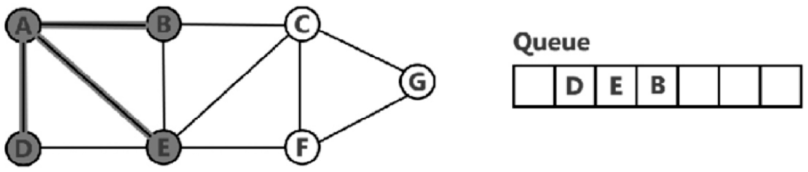
Step 1:

- Select the vertex A as starting point (visit A)
- Insert A into the Queue



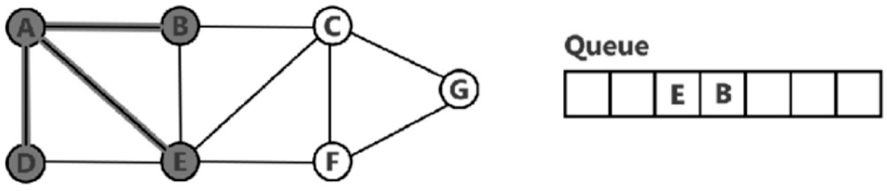
Step 2:

- Visit all adjacent vertices of A which are not visited (D, E, B)
- Insert newly visited vertices into the Queue and delete A from the Queue



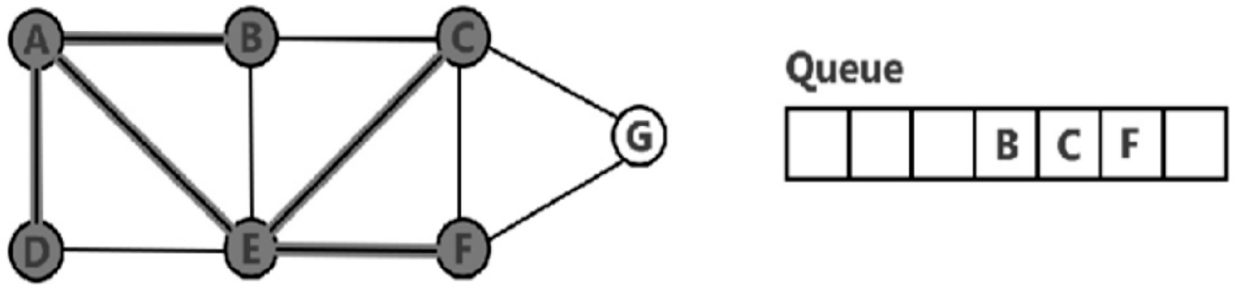
Step 3:

- Visit all adjacent vertices of D which are not visited (there is no vertex)
- Delete D from the Queue



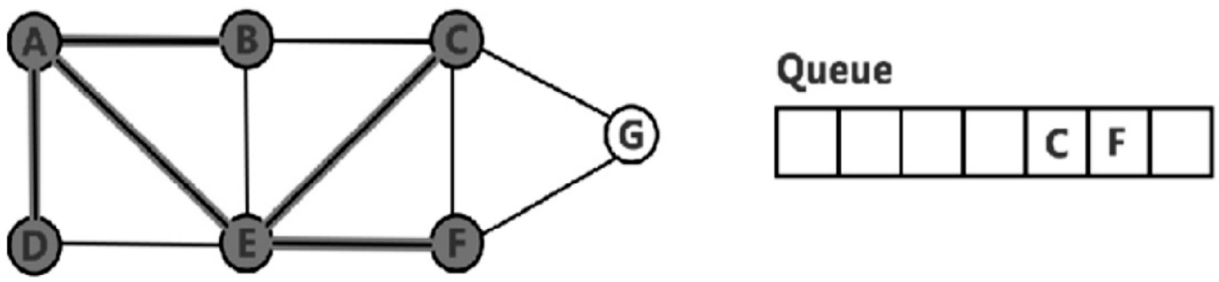
Step 4:

- Visit all adjacent vertices of E which are not visited (C, F)
- Insert newly visited vertices into the Queue and delete E from the Queue



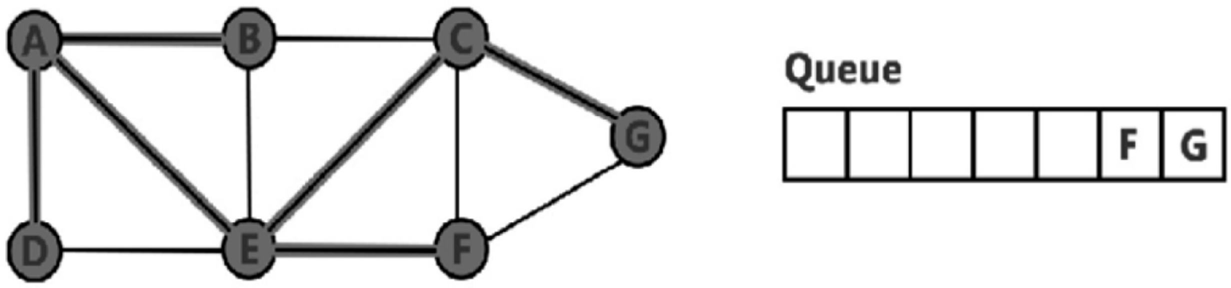
Step 5:

- Visit all adjacent vertices of B which are not visited (there is no vertex)
- Delete B from the Queue



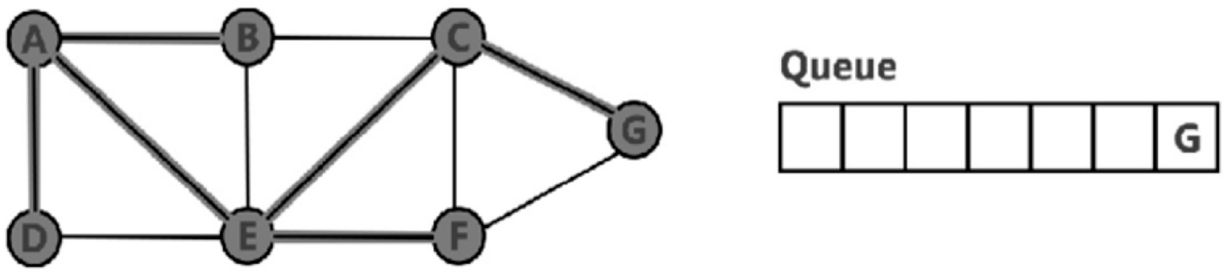
Step 6:

- Visit all adjacent vertices of C which are not visited (G)
- Insert newly visited vertices into the Queue and delete C from the Queue



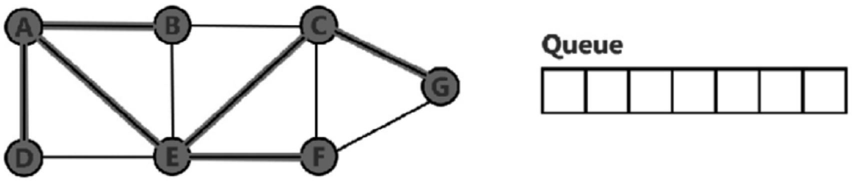
Step 7:

- Visit all adjacent vertices of F which are not visited (there is no vertex)
- Delete F form the Queue



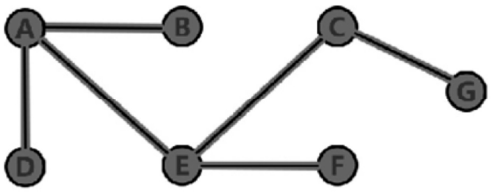
Step 8:

- Visit all adjacent vertices of G which are not visited (there is no vertex)
- Delete G form the Queue



Step 9:

- Queue became Empty. So, stop the BFS process
- Final result of BFS is a Spaning Tree as shown below



Q6. Write a C++ program for implementation on BFS

Ans :

```
#include <iostream>
#include <conio.h>
usingnamespace std;
int c =0, t =0;
struct node_info
{
int no;
int st_time;
}*q =NULL, *r =NULL, *x =NULL;
struct node
{
   node_info *pt;
   node *next;
}*front =NULL, *rear =NULL, *p =NULL, *np =NULL;
void push(node_info *ptr)
{
   np =new node;
   np->pt = ptr;
   np->next =NULL;
if(front ==NULL)
{
       front = rear = np;
       rear->next =NULL;
}
else
{
       rear->next = np;
       rear = np;
       rear->next =NULL;
}
}
node_info *remove()
{
if(front ==NULL)
{
cout<<"empty queue\n";
}
else
{
       p = front;
       x = p->pt;
       front = front->next;
delete(p);
return(x);
}
}
void bfs(int*v,int am[][7],int i)
{
if(c ==0)
{
       q =new node_info;
       q->no = i;
       q->st_time = t++;
cout<<"time of visitation for node
"<<q->no<<":"<<q->st_time<<"\n\n";
       v[i]=1;
       push(q);
}
   c++;
for(int j =0; j <7; j++)
{
if(am[i][j]==0||(am[i][j]==1&& v[j]==1))
continue;
elseif(am[i][j]==1&& v[j]==0)
{
          r =new node_info;
          r->no = j;
          r->st_time = t++;
cout<<"time of visitation for node "<<r-
>no<<":"<<r->st_time<<"\n\n";
          v[j]=1;
          push(r);
}
}
remove();
if(c <=6&& front !=NULL)
        bfs(v, am, remove()->no);
}
int main()
{
int v[7], am[7][7];
for(int i =0; i <7; i++)
        v[i]=0;
for(int i =0; i <7; i++)
{
cout<<"enter the values for adjacency matrix
row:"<<i+1<<endl;
for(int j =0; j <7; j++)
{
cin>>am[i][j];
}
}
    bfs(v, am, 0);
    getch();
        1.   }
```

Output

```
        enter the values for adjacency matrix row:1
        0
        1
        1
```

```
0
0
1
1
enter the values for adjacency matrix row:2
1
0
0
0
0
0
0
enter the values for adjacency matrix row:3
1
0
0
0
0
0
1
time of visitation for node 0:0
time of visitation for node 1:1
time of visitation for node 2:2
```

4.3 Spanning Tree

Q7. What is a Spanning tree? Represent with an example

Ans : **(Nov.-17)**

If we have a graph containing V vertices and E edges, then the graph can be represented as:

G(V, E)

If we create the spanning tree from the above graph, then the spanning tree would have the same number of vertices as the graph, but the vertices are not equal. The edges in the spanning tree would be equal to the number of edges in the graph minus 1.

Suppose the spanning tree is represented as:

G‘(V‘, E‘)

where,

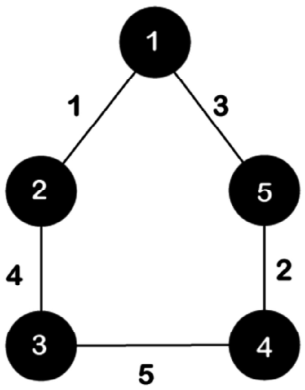
V = V'

E' ∈ E -1

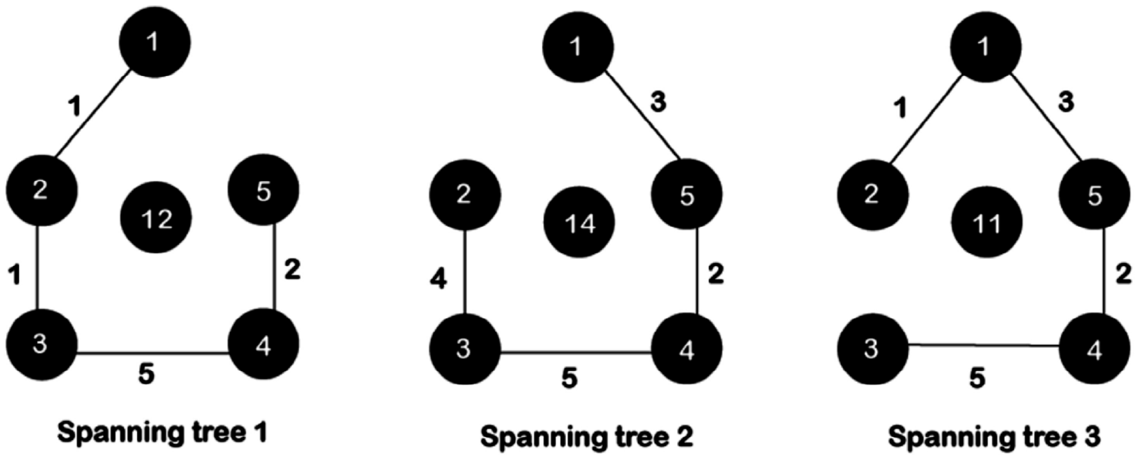
E' = |V| - 1

Let's understand through an example.

Suppose we want to create the spanning tree of the graph, which is shown below:



As we know, that spanning tree contains the same number of vertices as the graph, so the total number of vertices in the graph is 5; therefore, the spanning tree will also contain the 5 vertices. The edges in the spanning tree are equal to the number of vertices in the graph minus 1; therefore, the number of edges is 4. Three spanning trees can be created, which are shown below:

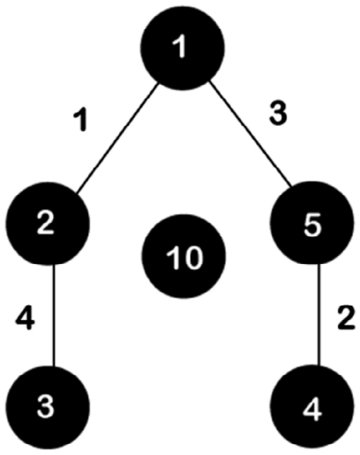


Q8. Explain the representation of Minimum Spanning tree.

Ans : **(June-18)**

Minimum Spanning Trees

The minimum spanning tree is the tree whose sum of the edge weights is minimum. From the above spanning trees, the total edge weight of the spanning tree 1 is 12, the total edge weight of the spanning tree 2 is 14, and the total edge weight of the spanning tree 3 is 11; therefore, the total edge weight of the spanning tree 3 is minimum. From the above graph, we can also create one more spanning tree as shown below:

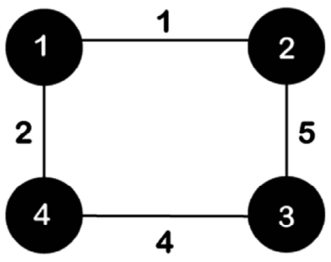


In the above tree, the total edge weight is 10 which is less than the above spanning trees; therefore, the minimum spanning tree is a tree which is having an edge weight, i.e., 10.

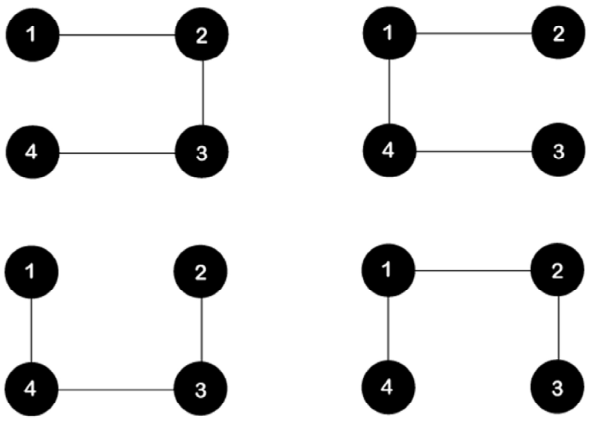
Properties of Spanning tree

- A connected graph can contain more than one spanning tree. The spanning trees which are minimally connected or we can say that the tree which is having a minimum total edge weight would be considered as the minimum spanning tree.
- All the possible spanning trees that can be created from the given graph G would have the same number of vertices, but the number of edges in the spanning tree would be equal to the number of vertices in the given graph minus 1.
- The spanning tree does not contain any cycle. Let's understand this property through an example.

  Suppose we have the graph which is given below:



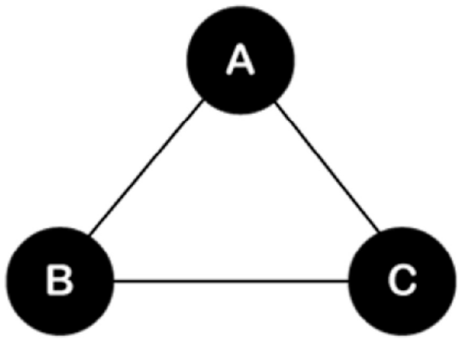
We can create the spanning trees of the above graph, which are shown below:



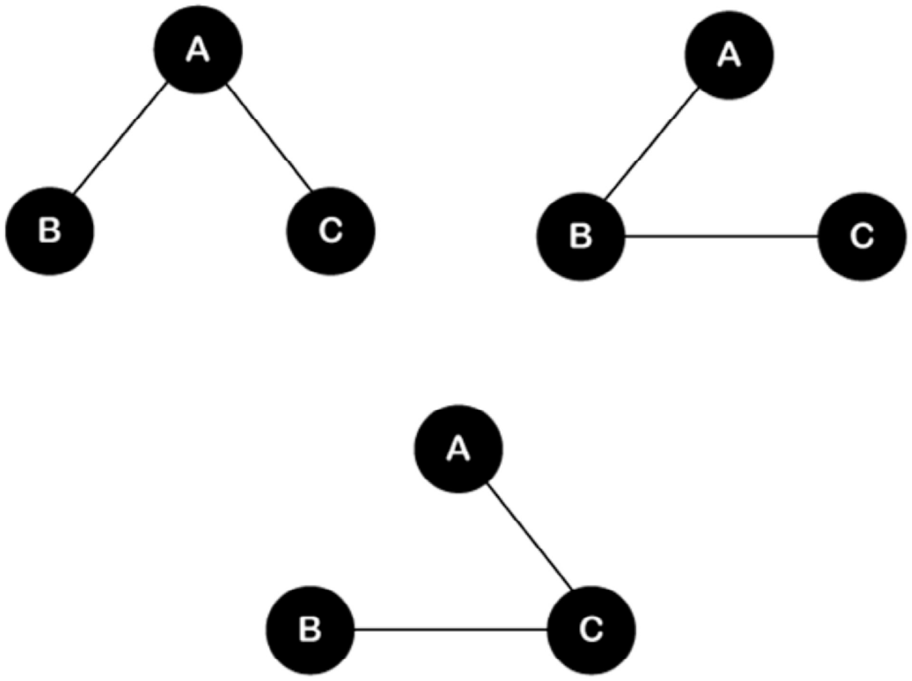
- As we can observe in the above spanning trees that one edge has been removed. If we do not remove one edge from the graph, then the tree will form a cycle, and that tree will not be considered as the spanning tree.
- The spanning tree cannot be disconnected. If we remove one more edge from any of the above spanning trees as shown below:

  The above tree is not a spanning tree because it is disconnected now.
- If two or three edges have the same edge weight, then there would be more than two minimum spanning trees. If each edge has a distinct weight, then there will be only one or unique minimum spanning tree.
- A complete undirected graph can have n^{n-2} number of spanning trees where n is the number of vertices in the graph. For example, the value of n is 5 then the number of spanning trees would be equal to 125.
- Each connected and undirected graph contains at least one spanning tree.
- The disconnected graph does not contain any spanning tree, which we have already discussed.
- If the graph is a complete graph, then the spanning tree can be constructed by removing maximum (e-n+1) edges. Let's understand this property through an example.

  A complete graph is a graph in which each pair of vertices are connected. Consider the complete graph having 3 vertices, which is shown below:



We can create three spanning trees from the above graph shown as below:



- According to this property, the maximum number of edges from the graph can be formulated as (e-n+1) where e is the number of edges, n is the number of vertices. When we substitute the value of e and n in the formula, then we get 1 value. It means that we can remove maximum 1 edge from the graph to make a spanning tree. In the above spanning trees, the one edge has been removed.

  Applications of Spanning trees

  The following are the applications of the spanning trees:

- Building a network: Suppose there are many routers in the network connected to each other, so there might be a possibility that it forms a loop. So, to avoid the formation of the loop, we use the tree data structure to connect the routers, and a minimum spanning tree is used to minimally connect them.
- Clustering: Here, clustering means that grouping the set of objects in such a way that similar objects belong to the same group than to the different group. Our goal is to divide the n objects into k groups such that the distance between the different groups gets maximized.

4.3.1 Prim's Algorithm

Q9. What is Prim's Algorithm?

Ans :

Prim's Algorithm is used to find the minimum spanning tree from a graph. Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.

Prim's algorithm starts with the single node and explore all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.

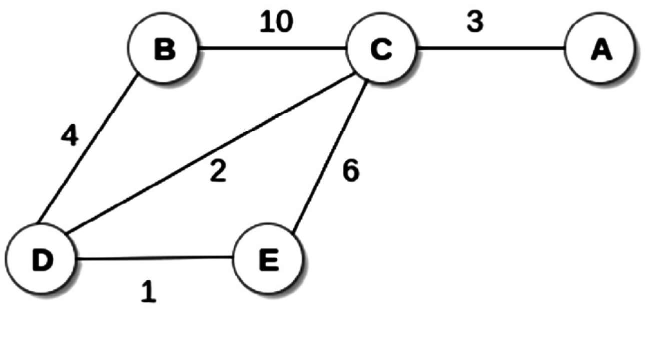
The algorithm is given as follows.

Algorithm

- **Step 1:** Select a starting vertex
- **Step 2:** Repeat Steps 3 and 4 until there are fringe vertices
- **Step 3:** Select an edge e connecting the tree vertex and fringe vertex that has minimum weight
- **Step 4:** Add the selected edge and the vertex to the minimum spanning tree T

  [END OF LOOP]
- **Step 5:** EXIT

Q10. Construct a minimum spanning tree of the graph given in the following figure by using prim's algorithm.



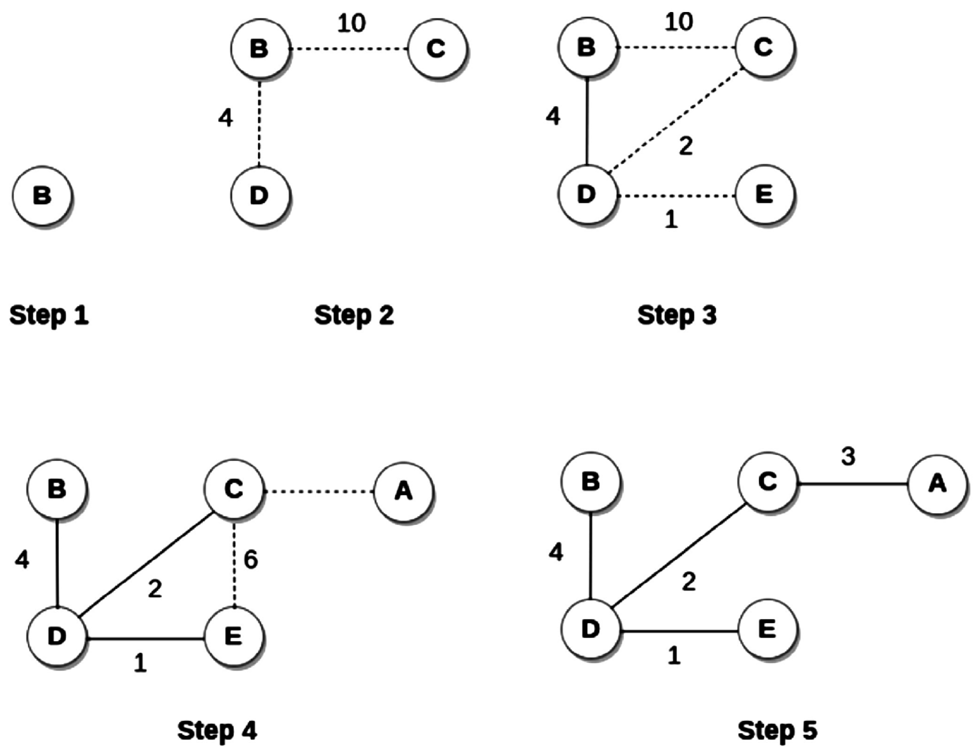
Sol :

- **Step 1 :** Choose a starting vertex B.
- **Step 2:** Add the vertices that are adjacent to A. the edges that connecting the vertices are shown by dotted lines.
- **Step 3:** Choose the edge with the minimum weight among all. i.e. BD and add it to MST. Add the adjacent vertices of D i.e. C and E.
- **Step 3:** Choose the edge with the minimum weight among all. In this case, the edges DE and CD are such edges. Add them to MST and explore the adjacent of C i.e. E and A.
- **Step 4:** Choose the edge with the minimum weight i.e. CA. We can't choose CE as it would cause cycle in the graph.

The graph produces in the step 4 is the minimum spanning tree of the graph shown in the above figure.

The cost of MST will be calculated as;

cost(MST) = 4 + 2 + 1 + 3 = 10 units.



Q11. Write a program to implement Prim's Algorithm.

Ans : **(Nov.-17)**

```
#include<iostream>
usingnamespace std;
// Number of vertices in the graph
constint V=6;
// Function to find the vertex with minimum key value
intmin_Key(int key[],bool visited[])
{
int min =999, min_index;// 999 represents an Infinite value
for(int v =0; v < V; v++){
if(visited[v]==false&& key[v]< min){
        // vertex should not be visited
          min = key[v];
```

```
                        min_index = v;
}
}
return min_index;
}
// Function to print the final MST stored in parent[]
intprint_MST(int parent[],int cost[V][V])
{
int minCost=0;
         cout<<"Edge \tWeight\n";
for(int i =1; i< V; i++){
                cout<<parent[i]<<" - "<<i<<" \t"<<cost[i][parent[i]]<<" \n";
                minCost+=cost[i][parent[i]];
}
         cout<<"Total cost is"<<minCost;
}
// Function to find the MST using adjacency cost matrix representation
voidfind_MST(int cost[V][V])
{
int parent[V], key[V];
bool visited[V];
// Initialize all the arrays
for(int i =0; i< V; i++){
        key[i]=999;// 99 represents an Infinite value
        visited[i]=false;
        parent[i]=-1;
}
    key[0]=0;// Include first vertex in MST by setting its key vaue to 0.
    parent[0]=-1;// First node is always root of MST
// The MST will have maximum V-1 vertices
for(int x =0; x < V -1; x++)
{
// Finding the minimum key vertex from the set of vertices not yet included in MST
int u =min_Key(key, visited);
visited[u]=true;// Add the minimum key vertex to the MST
// Update key and parent arrays
for(int v =0; v < V; v++)
{
// cost[u][v] is non zero only for adjacent vertices of u
// visited[v] is false for vertices not yet included in MST
// key[] gets updated only if cost[u][v] is smaller than key[v]
if(cost[u][v]!=0&& visited[v]==false&& cost[u][v]< key[v])
{
            parent[v]= u;
            key[v]= cost[u][v];
}}}
// print the final MST
      print_MST(parent, cost);
}
// main function
intmain()
{
int cost[V][V];
      cout<<"Enter the vertices for a graph with 6 vetices";
for(int i=0;i<V;i++)
{
for(int j=0;j<V;j++)
{
                  cin>>cost[i][j];
}}
      find_MST(cost);
return0;
}
```

The input graph is the same as the graph in Fig 2.

```
Enter the vertices for a graph with 6 vetices
   0  4  0  0  0  2
   4  0  6  0  0  3
   0  6  0  3  0  1
   0  0  3  0  2  0
   0  0  0  2  0  4
   2  3  1  0  4  0
   Edge      Weight
   5 - 1     3
   5 - 2     1
   2 - 3     3
   3 - 4     2
   0 - 5     2
   Total cost is 11
```

4.3.2 Kruskal's Algorithm.

Q12. Explain Kruskal's Algorithm with an example.

Ans : **(Imp.)**

Kruskal's Algorithm is used to find the minimum spanning tree for a connected weighted graph. The main target of the algorithm is to find the subset of edges by using which, we can traverse every vertex of the graph. Kruskal's algorithm follows greedy approach which finds an optimum solution at every stage instead of focusing on a global optimum.

The Kruskal's algorithm is given as follows.

Algorithm

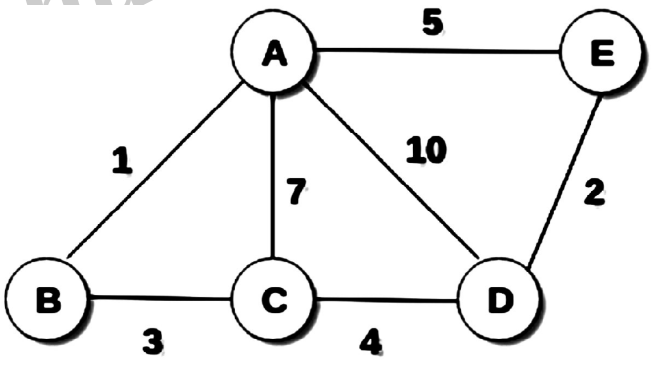
- **Step 1:** Create a forest in such a way that each graph is a separate tree.
- **Step 2:** Create a priority queue Q that contains all the edges of the graph.
- **Step 3:** Repeat Steps 4 and 5 while Q is NOT EMPTY
- **Step 4:** Remove an edge from Q
- **Step 5:** IF the edge obtained in Step 4 connects two different trees, then Add it to the forest (for combining two trees into one tree).

  ELSE

  Discard the edge
- **Step 6:** END

Example :

Apply the Kruskal's algorithm on the graph given as follows.



Sol:

the weight of the edges given as :

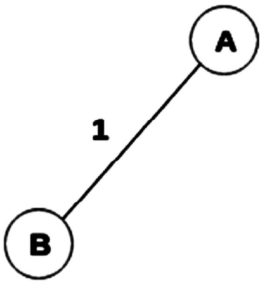
Edge	AE	AD	AC	AB	BC	CD	DE
Weight	5	10	7	1	3	4	2

Sort the edges according to their weights.

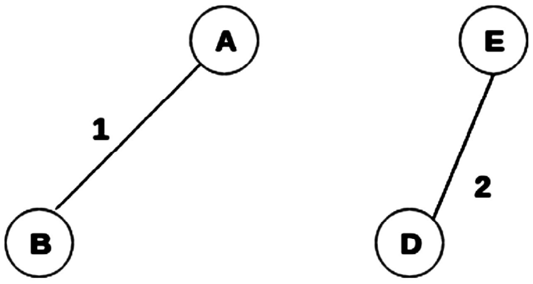
Edge	AB	DE	BC	CD	AE	AC	AD
Weight	1	2	3	4	5	7	10

Start constructing the tree;

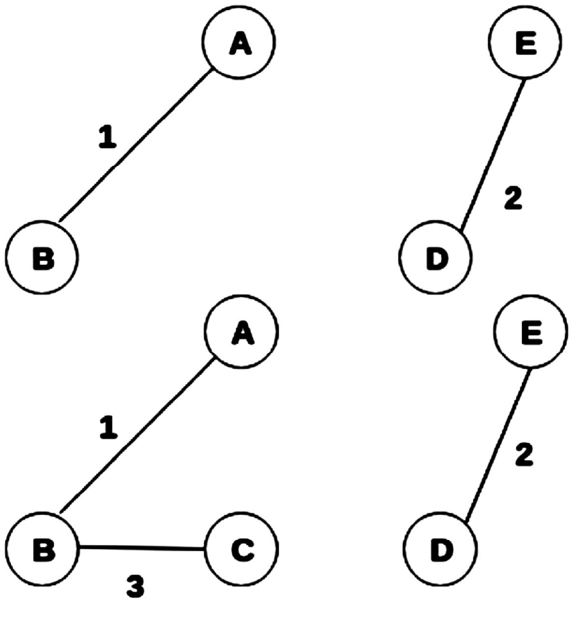
Add AB to the MST;



Add DE to the MST;



Add BC to the MST;



The next step is to add AE, but we can't add that as it will cause a cycle.

The next edge to be added is AC, but it can't be added as it will cause a cycle.

The next edge to be added is AD, but it can't be added as it will contain a cycle.

Hence, the final MST is the one which is shown in the step 4.

the cost of MST = 1 + 2 + 3 + 4 = 10.

The steps for implementing Kruskal's algorithm are as follows:

1. Sort all the edges from low weight to high
2. Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.
3. Keep adding edges until we reach all vertices.

Q13. Write a program to implement kruskal's algorithm

Ans :

```
// Kruskal's algorithm in C++
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
#define edge pair<int, int>
class Graph {
   private:
  vector<pair<int, edge>> G;  // graph
  vector<pair<int, edge>> T;  // mst
  int *parent;
  int V;  // number of vertices/nodes in graph
   public:
  Graph(int V);
  void AddWeightedEdge(int u, int v, int w);
  int find_set(int i);
  void union_set(int u, int v);
  void kruskal();
  void print();
};
Graph::Graph(int V) {
  parent = new int[V];
  //i 0 1 2 3 4 5
  //parent[i] 0 1 2 3 4 5
  for (int i = 0; i < V; i++)
    parent[i] = i;
  G.clear();
  T.clear();
}
void Graph::AddWeightedEdge(int u, int v, int w) {
  G.push_back(make_pair(w, edge(u, v)));
}
int Graph::find_set(int i) {
  // If i is the parent of itself
  if (i == parent[i])
    return i;
  else
    // Else if i is not the parent of itself
    // Then i is not the representative of his set,
    // so we recursively call Find on its parent
     return find_set(parent[i]);
}
void Graph::union_set(int u, int v) {
  parent[u] = parent[v];
}
void Graph::kruskal() {
  int i, uRep, vRep;
  sort(G.begin(), G.end());  // increasing weight
  for (i = 0; i < G.size(); i++) {
    uRep = find_set(G[i].second.first);
    vRep = find_set(G[i].second.second);
    if (uRep != vRep) {
      T.push_back(G[i]);  // add to tree
      union_set(uRep, vRep);
    }
  }
}
void Graph::print() {
  cout << "Edge :"
<< " Weight" << endl;
         for (int i = 0; i < T.size(); i++) {
         cout << T[i].second.first << " - " << T[i].second.second << " : "
<< T[i].first;
    cout << endl;
  }
}
int main() {
  Graph g(6);
  g.AddWeightedEdge(0, 1, 4);
  g.AddWeightedEdge(0, 2, 4);
  g.AddWeightedEdge(1, 2, 2);
  g.AddWeightedEdge(1, 0, 4);
  g.AddWeightedEdge(2, 0, 4);
  g.AddWeightedEdge(2, 1, 2);
  g.AddWeightedEdge(2, 3, 3);
  g.AddWeightedEdge(2, 5, 2);
  g.AddWeightedEdge(2, 4, 4);
  g.AddWeightedEdge(3, 2, 3);
  g.AddWeightedEdge(3, 4, 3);
  g.AddWeightedEdge(4, 2, 4);
  g.AddWeightedEdge(4, 3, 3);
  g.AddWeightedEdge(5, 2, 2);
  g.AddWeightedEdge(5, 4, 3);
  g.kruskal();
  g.print();
  return 0;
}
```

4.4 Hashing

4.4.1 Introduction

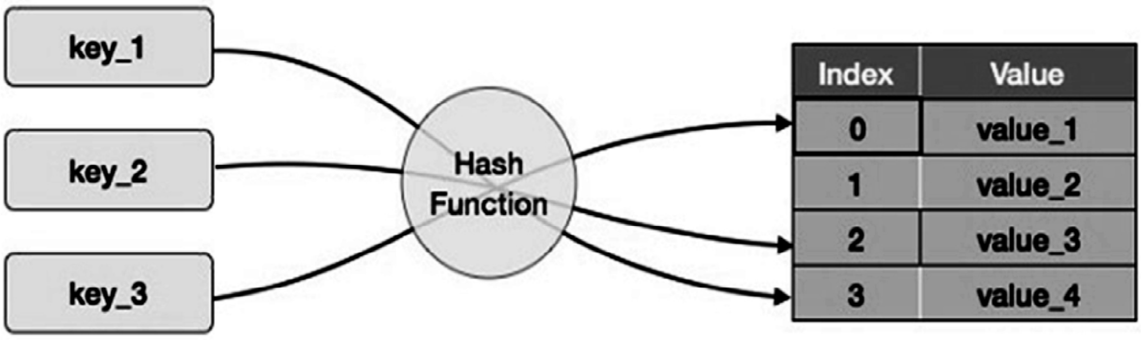
Q14. What is Hashing? Define it with an example.

Ans : **(July-21)**

- Hashing is the process of mapping large amount of data item to smaller table with the help of hashing function.
- Hashing is also known as Hashing Algorithm or Message Digest Function.
- It is a technique to convert a range of key values into a range of indexes of an array.
- It is used to facilitate the next level searching method when compared with the linear or binary search.
- Hashing allows to update and retrieve any data entry in a constant time O(1).
- Constant time O(1) means the operation does not depend on the size of the data.
- Hashing is used with a database to enable items to be retrieved more quickly.
- It is used in the encryption and decryption of digital signatures.

Example

Consider an example of hash table of size 20, and the following items are to be stored. Item are in the (key,value) format.



(1,20) , (2,70) , (42,80) , (4,25) , (12,44) , (14,32) , (17,11) , (13,78) , (37,98)

Sr. No.	Key	Hash	Array Index
1	1	1 % 20 = 1	1
2	2	2 % 20 = 2	2
3	42	42 % 20 = 2	2
4	4	4 % 20 = 4	4
5	12	12 % 20 = 12	12
6	14	14 % 20 = 14	14
7	17	17 % 20 = 17	17
8	13	13 % 20 = 13	13
9	37	37 % 20 = 17	17

4.4.2 Key Terms and Issues

Q15. Mention the key terms and issues of hash table.

Ans :

- Hash table Hash table is an array [0 to Max – 1] of size Max.
- Bucket A bucket is an index position in a hash table that can store more than one record. When the same index is mapped with two keys, both the records are stored in the same bucket. The assumption is that the buckets are equal in size.
- Probe Each action of address calculation and check for success is called as a probe.
- Collision The result of two keys hashing into the same address is called collision.
- Synonym Keys that hash to the same address are called synonyms.
- Overflow The result of many keys hashing to a single address and lack of room in the bucket is known as an overflow. Collision and overflow are synonymous when the bucket is of size 1.
- Open or external hashing When we allow records to be stored in potentially unlimited space, it is called as open or external hashing.
- Closed or internal hashing When we use fixed space for storage eventually limiting the number of records to be stored, it is called as closed or internal hashing.
- Hash function Hash function is an arithmetic function that transforms a key into an
- address which is used for storing and retrieving a record.
- Perfect hash function The hash function that transforms different keys into different
- addresses is called a perfect hash function. The worth of a hash function depends on how
- well it avoids collision.
- Load density The maximum storage capacity, that is, the maximum number of records
- that can be accommodated, is called as loading density.
- Full table A full table is one in which all locations are occupied. Owing to the
- characteristics of hash functions, there are always empty locations, rather a hash function should not allow the table to get filled in more than 75%.
- Load factor Load factor is the number of records stored in a table divided by the
- maximum capacity of the table, expressed in terms of percentage.
- Rehashing Rehashing is with respect to closed hashing. When we try to store the record with Key1 at the bucket position Hash(Key1) and find that it already holds a record, it is collision situation. To handle collision, we use a strategy to choose a sequence of alternative locations Hash1(Key1), Hash2(Key1), and so on within the bucket table so as to place the record with Key1. This is known as rehashing.

Issues in hashing In case of collision, there are two main issues to be considered:

1. We need a good hashing function that minimizes the number of collisions.
2. We want an efficient collision resolution strategy so as to store or locate synonyms.

4.4.3 Hash Functions

Q16. What is Hash Function?

Ans : **(June-18)**

- A fixed process converts a key to a hash key is known as a Hash Function.
- This function takes a key and maps it to a value of a certain length which is called a Hash value or Hash.
- Hash value represents the original string of characters, but it is normally smaller than the original.
- It transfers the digital signature and then both hash value and signature are sent to the receiver. Receiver uses the same hash function to generate the hash value and then compares it to that received with the message.
- If the hash values are same, the message is transmitted without errors.

Q17. Explain various methods of hash functions.

Ans :

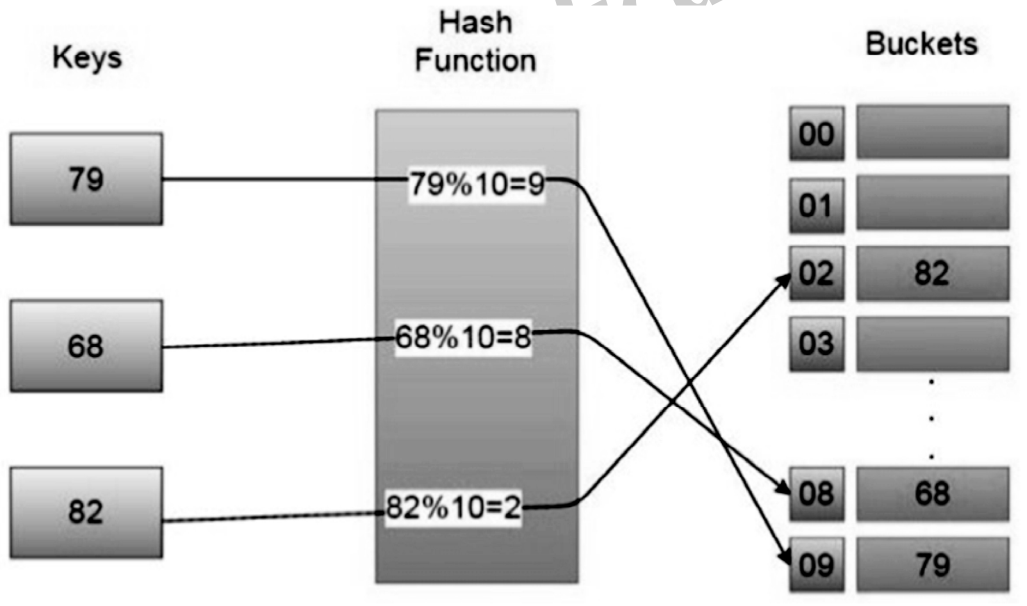
Hash function Hash function is one that maps a key in the range [0 to Max – 1], the result of which is used as an index (or address) in the hash table for storing and retrieving records. One more way to define a hash function is as the function that transforms a key into an address. The address generated by a hashing function is called the home address. All home addresses refer to a particular area of the memory called the prime area.

Division Hash Method

- The key K is divided by some number m and the remainder is used as the hash address of K.
  - h(k)=k mod m
- This gives the indexes in the range 0 to m-1 so the hash table should be of size m
- This is an example of uniform hash function if value of m will be chosen carefully.
- Generally a prime number is a best choice which will spread keys evenly.

A uniform hash function is designed to distribute the keys roughly evenly into the available positions within the array (or hash table).

For example: Let us say apply division approach to find hash value for some values considering number of buckets be 10 as shown below.



The Multiplication Method

The multiplication method for creating a hash function operates in two steps.

- **Step 1.** Multiply the key k by a constant A in the range 0 < A < 1, and extract the fractional part of kA.
- **Step 2.** Multiply this value by m and take the floor of the result. Why?
- In short, the hash function is

- h(k) = bm·(kA mod 1)c,
- where (kA mod 1) denotes the fractional part of kA, that is, kA”bkAc.

E.g., m = 10000, k = 123456, and A = p5 –1

2 = 0.618033,

then

h(k) = b10000 ·(123456 ·0.61803. . .mod 1)c

= b10000 ·(76300.0041151. . . mod 1)c

= b10000 ·0.0041151. . . c

= b41.151. . . c

= 41.

The advantage of this method is that the value choice of m is not critical.

The Folding Method

➢ The key K is partitioned into a number of parts ,each of which has the same length as the required address with the possible exception of the last part .

➢ The parts are then added together , ignoring the final carry, to form an address.

➢ **Example:** If key=356942781 is to be transformed into a three digit address.

P1=356, P2=942, P3=781 are added to yield 079.

The Mid- Square Method

➢ The key K is multiplied by itself and the address is obtained by selecting an appropriate number of digits from the middle of the square.

➢ The number of digits selected depends on the size of the table.

➢ Example: If key=123456 is to be transformed.

(123456)2=15241383936

If a three-digit address is required, positions 5 to 7 could be chosen giving address 138.

Universal hashing

If a malicious adversary chooses keys to be hashed, then he can choose n keys that all hash to the same slot, yielding an average retrieval time of Q(n). Any fixed hash function is vulnerable to this sort of worst case behavior. The only effective way to improve the situation is to choose a hash function randomly in a way that is independent of the keys that are actually going to be stored.

This approach is referred to as universal hashing. In other words, universal hashing is to select the hash function at random and at run time from a carefully designed collection of hash functions.

Let H = {h1,h2, . . . ,hl} be a finite collection of hash functions that map a given

universe U of keys into a range {0,1, . . . ,m–1}.

Such a collection is said to be universal if for each pair of distinct keys x, y 2U, the number of hash functions h 2 H for which h(x) = h(y) is at most $\frac{|H|}{m} = \frac{l}{m}$

Characteristics of a Good Hash Function

- The hash value is fully determined by the data being hashed.
- The hash function uses all the input data.
- The hash function "uniformly" distributes the data across the entire set of possible hash values.
- The hash function generates very different hash values for similar strings.

4.4.4 Collision Resolution Strategies

Q18. What is collision resolution technique?

Ans :

When the two different values have the same value, then the problem occurs between the two values, known as a collision. In the above example, the value is stored at index 6. If the key value is 26, then the index would be:

h(26) = 26%10 = 6

Therefore, two values are stored at the same index, i.e., 6, and this leads to the collision problem. To resolve these collisions, we have some techniques known as collision techniques.

The following are the collision techniques:

- **Open Hashing:** It is also known as closed addressing.
- **Closed Hashing:** It is also known as open addressing.

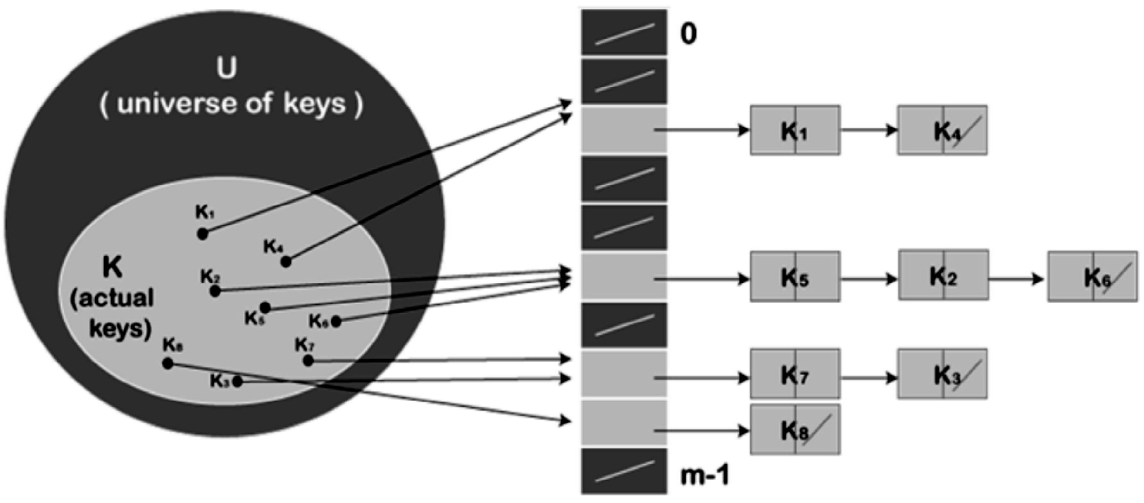
Q19. Explain Open Hashing method with an example.

Ans :

Open Hashing

In Open Hashing, one of the methods used to resolve the collision is known as a chaining method.

Collision Resolution by Chaining



Let’s first understand the chaining to resolve the collision.

Suppose we have a list of key values

A = 3, 2, 9, 6, 11, 13, 7, 12 where m = 10, and h(k) = 2k+3

In this case, we cannot directly use h(k) = k_i/m as h(k) = 2k+3

- The index of key value 3 is:

index = h(3) = (2(3)+3)%10 = 9

The value 3 would be stored at the index 9.

- The index of key value 2 is:

index = h(2) = (2(2)+3)%10 = 7

The value 2 would be stored at the index 7.

- The index of key value 9 is:

index = h(9) = (2(9)+3)%10 = 1

The value 9 would be stored at the index 1.

- The index of key value 6 is:

index = h(6) = (2(6)+3)%10 = 5

The value 6 would be stored at the index 5.

- The index of key value 11 is:

index = h(11) = (2(11)+3)%10 = 5

The value 11 would be stored at the index 5. Now, we have two values (6, 11) stored at the same index, i.e., 5. This leads to the collision problem, so we will use the chaining method to avoid the collision. We will create one more list and add the value 11 to this list. After the creation of the new list, the newly created list will be linked to the list having value 6.

- The index of key value 13 is:

index = h(13) = (2(13)+3)%10 = 9

The value 13 would be stored at index 9. Now, we have two values (3, 13) stored at the same index, i.e., 9. This leads to the collision problem, so we will use the chaining method to avoid the collision. We will create one more list and add the value 13 to this list. After the creation of the new list, the newly created list will be linked to the list having value 3.

- The index of key value 7 is:

index = h(7) = (2(7)+3)%10 = 7

The value 7 would be stored at index 7. Now, we have two values (2, 7) stored at the same index, i.e., 7. This leads to the collision problem, so we will use the chaining method to avoid the collision. We will create one more list and add the value 7 to this list. After the creation of the new list, the newly created list will be linked to the list having value 2.

- The index of key value 12 is:

index = h(12) = (2(12)+3)%10 = 7

According to the above calculation, the value 12 must be stored at index 7, but the value 2 exists at index 7. So, we will create a new list and add 12 to the list. The newly created list will be linked to the list having a value 7.

The calculated index value associated with each key value is shown in the below table:

key	Location(u)
3	((2*3)+3)%10 = 9
2	((2*2)+3)%10 = 7
9	((2*9)+3)%10 = 1
6	((2*6)+3)%10 = 5
11	((2*11)+3)%10 = 5
13	((2*13)+3)%10 = 9
7	((2*7)+3)%10 = 7
12	((2*12)+3)%10 = 7

Q20. What is Closed Hashing? Explain Linear probing with example.

Ans :

Closed Hashing

In Closed hashing, three techniques are used to resolve the collision:

1. Linear probing
2. Quadratic probing
3. Double Hashing technique

1. Linear Probing

Linear probing is one of the forms of open addressing. As we know that each cell in the hash table contains a key-value pair, so when the collision occurs by mapping a new key to the cell already occupied by another key, then linear probing technique searches for the closest free locations and adds a new key to that empty cell. In this case, searching is performed sequentially, starting from the position where the collision occurs till the empty cell is not found.

Let's understand the linear probing through an example.

Consider the above example for the linear probing:

A = 3, 2, 9, 6, 11, 13, 7, 12 where m = 10, and h(k) = 2k+3

The key values 3, 2, 9, 6 are stored at the indexes 9, 7, 1, 5 respectively. The calculated index value of 11 is 5 which is already occupied by another key value, i.e., 6. When linear probing is applied, the nearest empty cell to the index 5 is 6; therefore, the value 11 will be added at the index 6.

The next key value is 13. The index value associated with this key value is 9 when hash function is applied. The cell is already filled at index 9. When linear probing is applied, the nearest empty cell to the index 9 is 0; therefore, the value 13 will be added at the index 0.

The next key value is 7. The index value associated with the key value is 7 when hash function is applied. The cell is already filled at index 7. When linear probing is applied, the nearest empty cell to the index 7 is 8; therefore, the value 7 will be added at the index 8.

The next key value is 12. The index value associated with the key value is 7 when hash function is applied. The cell is already filled at index 7. When linear probing is applied, the nearest empty cell to the index 7 is 2; therefore, the value 12 will be added at the index 2.

2. Quadratic Probing

In case of linear probing, searching is performed linearly. In contrast, quadratic probing is an open addressing technique that uses quadratic polynomial for searching until a empty slot is found.

It can also be defined as that it allows the insertion ki at first free location from $(u+i^2)\%m$ where i=0 to m-1.

Let's understand the quadratic probing through an example.

Consider the same example which we discussed in the linear probing.

A = 3, 2, 9, 6, 11, 13, 7, 12 where m = 10, and h(k) = 2k+3

The key values 3, 2, 9, 6 are stored at the indexes 9, 7, 1, 5, respectively. We do not need to apply the quadratic probing technique on these key values as there is no occurrence of the collision.

The index value of 11 is 5, but this location is already occupied by the 6. So, we apply the quadratic probing technique.

When i = 0

Index= $(5+ 0^2)\%10 = 5$

When i=1

Index = $(5+1^2)\%10 = 6$

Since location 6 is empty, so the value 11 will be added at the index 6.

The next element is 13. When the hash function is applied on 13, then the index value comes out to be 9, which we already discussed in the chaining method. At index 9, the cell is occupied by another value, i.e., 3. So, we will apply the quadratic probing technique to calculate the free location.

When i=0

Index = $(9+0^2)\%10 = 9$

When i=1

Index = $(9+1^2)\%10 = 0$

Since location 0 is empty, so the value 13 will be added at the index 0.

The next element is 7. When the hash function is applied on 7, then the index value comes out to be 7, which we already discussed in the chaining method. At index 7, the cell is occupied by another value, i.e., 7. So, we will apply the quadratic probing technique to calculate the free location.

When i=0

Index = $(7+0^2)\%10 = 7$

When i=1

Index = $(7+1^2)\%10 = 8$

Since location 8 is empty, so the value 7 will be added at the index 8.

The next element is 12. When the hash function is applied on 12, then the index value comes out to be 7. When we observe the hash table then we will get to know that the cell at index 7 is already occupied by the value 2. So, we apply the Quadratic probing technique on 12 to determine the free location.

When i=0

Index= $(7+0^2)\%10 = 7$

When i=1

Index = $(7+1^2)\%10 = 8$

When i=2

Index = $(7+2^2)\%10 = 1$

When i=3

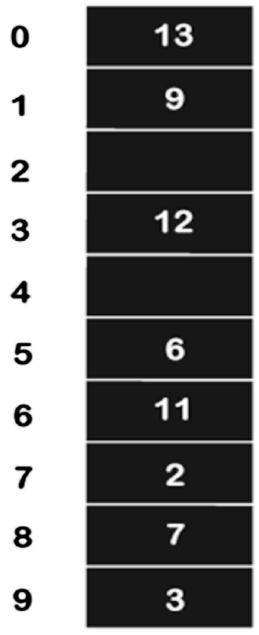
Index = $(7+3^2)\%10 = 6$

When i=4

Index = $(7+4^2)\%10 = 3$

Since the location 3 is empty, so the value 12 would be stored at the index 3.

The final hash table would be:



Therefore, the order of the elements is 13, 9, _, 12, _, 6, 11, 2, 7, 3.

3. Double Hashing

Double hashing is an open addressing technique which is used to avoid the collisions. When the collision occurs then this technique uses the secondary hash of the key. It uses one hash value as an index to move forward until the empty location is found.

In double hashing, two hash functions are used. Suppose $h_1(k)$ is one of the hash functions used to calculate the locations whereas $h_2(k)$ is another hash function. It can be defined as "insert k_i at first free place from (u+v*i)%m where i=(0 to m-1)". In this case, u is the location computed using the hash function and v is equal to ($h_2(k)$%m).

Consider the same example that we use in quadratic probing.

A = 3, 2, 9, 6, 11, 13, 7, 12 where m = 10, and

$h_1(k) = 2k+3$

$h_2(k) = 3k+1$

key	Location (u)	v	probe
3	((2*3)+3)%10 = 9	-	1
2	((2*2)+3)%10 = 7	-	1
9	((2*9)+3)%10 = 1	-	1
6	((2*6)+3)%10 = 5	-	1
11	((2*11)+3)%10 = 5	(3(11)+1)%10 =4	3
13	((2*13)+3)%10 = 9	(3(13)+1)%10 = 0	
7	((2*7)+3)%10 = 7	(3(7)+1)%10 = 2	
12	((2*12)+3)%10 = 7	(3(12)+1)%10 = 7	2

As we know that no collision would occur while inserting the keys (3, 2, 9, 6), so we will not apply double hashing on these key values.

On inserting the key 11 in a hash table, collision will occur because the calculated index value of 11 is 5 which is already occupied by some another value. Therefore, we will apply the double hashing technique on key 11. When the key value is 11, the value of v is 4.

Now, substituting the values of u and v in (u+v*i)%m

When i=0

Index = (5+4*0)%10 =5

When i=1

Index = (5+4*1)%10 = 9

When i=2

Index = (5+4*2)%10 = 3

Since the location 3 is empty in a hash table; therefore, the key 11 is added at the index 3.

The next element is 13. The calculated index value of 13 is 9 which is already occupied by some another key value. So, we will use double hashing technique to find the free location. The value of v is 0.

Now, substituting the values of u and v in (u+v*i)%m

When i=0

Index = (9+0*0)%10 = 9

We will get 9 value in all the iterations from 0 to m-1 as the value of v is zero. Therefore, we cannot insert 13 into a hash table.

The next element is 7. The calculated index value of 7 is 7 which is already occupied by some another key value. So, we will use double hashing technique to find the free location. The value of v is 2.

Now, substituting the values of u and v in

(u+v*i)%m

When i=0

Index = (7 + 2*0)%10 = 7

When i=1

Index = (7+2*1)%10 = 9

When i=2

Index = (7+2*2)%10 = 1

When i=3

Index = (7+2*3)%10 = 3

When i=4

Index = (7+2*4)%10 = 5

When i=5

Index = (7+2*5)%10 = 7

When i=6

Index = (7+2*6)%10 = 9

When i=7

Index = (7+2*7)%10 = 1

When i=8

Index = (7+2*8)%10 = 3

When i=9

Index = (7+2*9)%10 = 5

Since we checked all the cases of i (from 0 to 9), but we do not find suitable place to insert 7. Therefore, key 7 cannot be inserted in a hash table.

The next element is 12. The calculated index value of 12 is 7 which is already occupied by some another key value. So, we will use double hashing technique to find the free location. The value of v is 7.

Now, substituting the values of u and v in (u+v*i)%m

When i=0

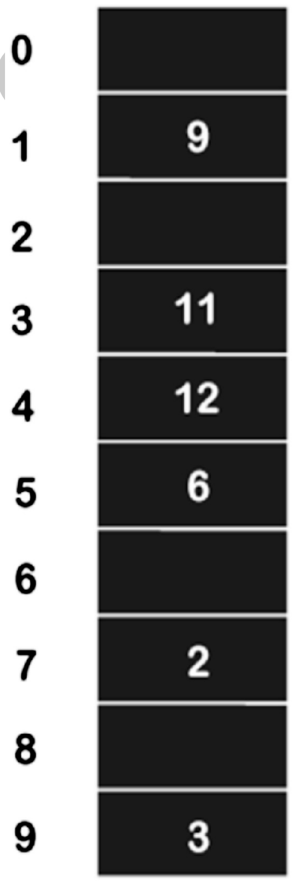
Index = (7+7*0)%10 = 7

When i=1

Index = (7+7*1)%10 = 4

Since the location 4 is empty; therefore, the key 12 is inserted at the index 4.

The final hash table would be:



The order of the elements is _, 9, _, 11, 12, 6, _, 2, _, 3.

4.4.5 Hash Table Overflow

Q21. How the overflow occurs in hash table? explain.

Ans :

An overflow occurs at the time of the home bucket for a new pair (key, element) is full.

We may tackle overflows bySearch the hash table in some systematic manner for a bucket that is not full.

- ➢ Linear probing (linear open addressing).
- ➢ Quadratic probing.
- ➢ Random probing.

Eliminate overflows by allowing each bucket to keep a list of all pairs for which it is the home bucket.

- ➢ Array linear list.
- ➢ Chain.

Open addressing is performed to ensure that all elements are stored directly into the hash table, thus it attempts to resolve collisions implementing various methods.

Linear Probing is performed to resolve collisions by placing the data into the next open slot in the table.

Performance of Linear Probing

- ➢ Worst-case find/insert/erase time is è(m), where m is treated as the number of pairs in the table.
- ➢ This occurs when all pairs are in the same cluster.

Problem of Linear Probing

- ➢ Identifiers are tending to cluster together
- ➢ Adjacent clusters are tending to coalesce
- ➢ Increase or enhance the search time
- ➢ **Quadratic Probing**

  Linear probing searches buckets (H(x)+i2)%b; H(x) indicates Hash function of x

  Quadratic probing implements a quadratic function of i as the increment

Examine buckets H(x), (H(x)+i2)%b, (H(x)-i2)%b, for 1<=i<=(b-1)/2

b is indicated as a prime number of the form 4j+3, j is an integer

- ➢ **Random Probing**

  Random Probing performs incorporating with random numbers.

  H(x):= (H'(x) + S[i]) % b

  S[i] is a table along with size b-1

  S[i] is indicated as a random permutation of integers [1, b-1].

4.4.6 Extendible Hashing

Q22. What is extendible hashing ? Explain with a help of an example.

Ans :

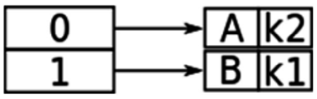
- ➢ It is an approach that tries to make hashing dynamic,i.e. to allow insertions and deletions to occur without resulting in poor performance after many of these operations.
- ➢ Extendible hashing combines two ingredients: Hashing and tries.
- ➢ tries are digital trees like the one used in Lempel-¡ iv
- ➢ Keys are placed into buckets,which are independent parts of a file in disk.
- ➢ Keys having a hashing address with the same prefix share the same bucket.
- ➢ Atrie is used for fast access to the buckets. It uses a prefix of the
- ➢ hashing address in order to locate the desired bucket.

Example

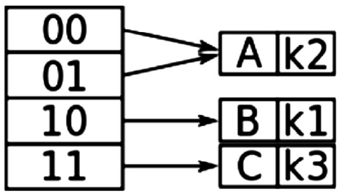
Assume that the hash function {\displaystyle h(k)}h returns a string of bits. The first i bits of each string will be used as indices to figure out where they will go in the "directory" (hash table). Additionally, i is the smallest number such that the index of every item in the table is unique.

Keys to be used

Let's assume that for this particular example, the bucket size is 1. The first two keys to be inserted, k_1 and k_2, can be distinguished by the most significant bit, and would be inserted into the table as follows:



Now, if k_3 were to be hashed to the table, it wouldn't be enough to distinguish all three keys by one bit (because both k_3 and k_1 have 1 as their leftmost bit). Also, because the bucket size is one, the table would overflow. Because comparing the first two most significant bits would give each key a unique location, the directory size is doubled as follows:



And so now k_1 and k_3 have a unique location, being distinguished by the first two leftmost bits. Because k_2 is in the top half of the table, both 00 and 01 point to it because there is no other key to compare to that begins with a 0.

Now, k_4 needs to be inserted, and it has the first two bits as 01..(1110), and using a 2 bit depth in the directory, this maps from 01 to Bucket A. Bucket A is full (max size 1), so it must be split; because there is more than one pointer to Bucket A, there is no need to increase the directory size.

What is needed is information about:

1. The key size that maps the directory (the global depth), and
2. The key size that has previously mapped the bucket (the local depth)

In order to distinguish the two action cases:

1. Doubling the directory when a bucket becomes full
2. Creating a new bucket, and re-distributing the entries between the old and the new bucket

Examining the initial case of an extendible hash structure, if each directory entry points to one bucket, then the local depth should be equal to the global depth.

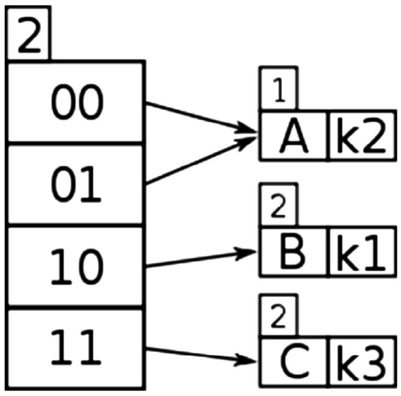
The number of directory entries is equal to $2^{\text{global depth}}$, and the initial number of buckets is equal to $2^{\text{local depth}}$.

Thus if global depth = local depth = 0, then $2^0 = 1$, so an initial directory of one pointer to one bucket.

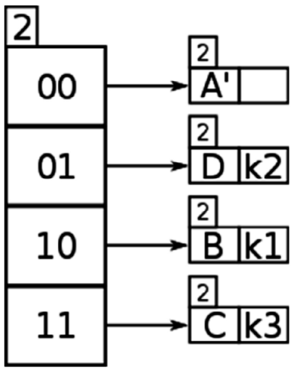
Back to the two action cases

If the local depth is equal to the global depth, then there is only one pointer to the bucket, and there is no other directory pointers that can map to the bucket, so the directory must be doubled (case1).

If the bucket is full, if the local depth is less than the global depth, then there exists more than one pointer from the directory to the bucket, and the bucket can be split (case 2).

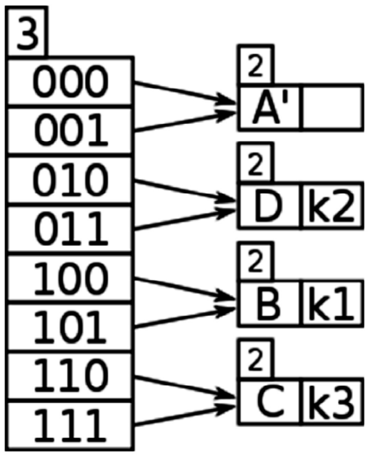


Key 01 points to Bucket A, and Bucket A's local depth of 1 is less than the directory's global depth of 2, which means keys hashed to Bucket A have only used a 1 bit prefix (i.e. 0), and the bucket needs to have its contents split using keys 1 + 1 = 2 bits in length; in general, for any local depth d where d is less than D, the global depth, then d must be incremented after a bucket split, and the new d used as the number of bits of each entry's key to redistribute the entries of the former bucket into the new buckets.

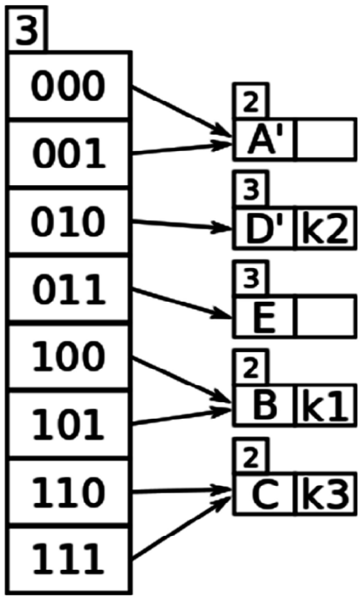


Now, h is tried again, with 2 bits 01.., and now key 01 points to a new bucket but there is still B K1 in it (B K1 and also begins with 01). If ADBC had been 000110, with key 00, there would have been no problem, because 00 would have remained in the new bucket A' and bucket D would have been empty.

So Bucket D needs to be split, but a check of its local depth, which is 2, is the same as the global depth, which is 2, so the directory must be split again, in order to hold keys of sufficient detail, e.g. 3 bits.



1. Bucket D needs to split due to being full.
2. As D's local depth = the global depth, the directory must double to increase bit detail of keys.
3. Global depth has incremented after directory split to 3.
4. The new entry D is rekeyed with global depth 3 bits and ends up in D which has local depth 2, which can now be incremented to 3 and D can be split to D' and E.
5. The contents of the split bucket D, K, has been re-keyed with 3 bits, and it ends up in D.
6. K4 is retried and it ends up in E which has a spare slot.



Now, k2 is in D and E is tried again, with 3 bits 011.., and it points to bucket D which already contains k2 so is full; D's local depth is 2 but now the global depth is 3 after the directory doubling, so now D can be split into bucket's D' and E, the contents of D, k2 has its retried with a new global depth bitmask of 3 and ends up in D', then the new entry is retried with bitmasked using the new global depth bit count of 3 and this gives 011 which now points to a new bucket E which is empty. So k1 goes in Bucket E.

4.5 Heaps

4.5.1 Basic Concepts

Q23. What is Heap?

Ans : **(July-19)**

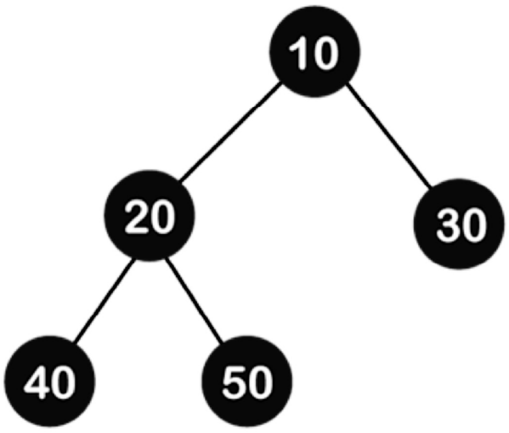
A heap is a complete binary tree, and the binary tree is a tree in which the node can have utmost two children. Before knowing more about the heap data structure, we should know about the complete binary tree.

Q24. What is a complete binary tree?

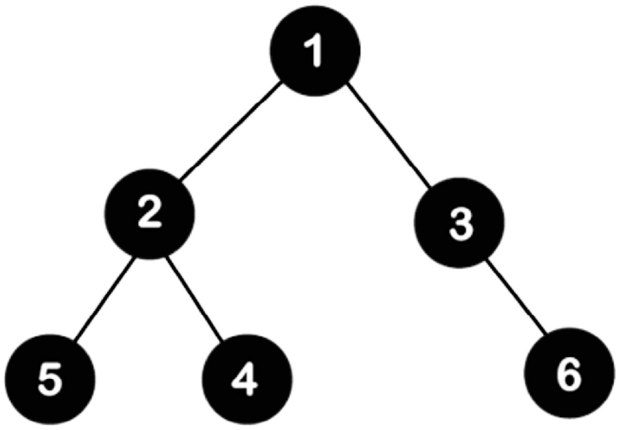
Ans :

A complete binary tree is a binary tree in which all the levels except the last level, i.e., leaf node should be completely filled, and all the nodes should be left-justified.

Let's understand through an example.



In the above figure, we can observe that all the internal nodes are completely filled except the leaf node; therefore, we can say that the above tree is a complete binary tree.



The above figure shows that all the internal nodes are completely filled except the leaf node, but the leaf nodes are added at the right part; therefore, the above tree is not a complete binary tree.

Q25. What are the different types of heaps? Explain Max heap algorithm with an example

Ans :

There are two types of the heap:

- Min Heap
- Max heap

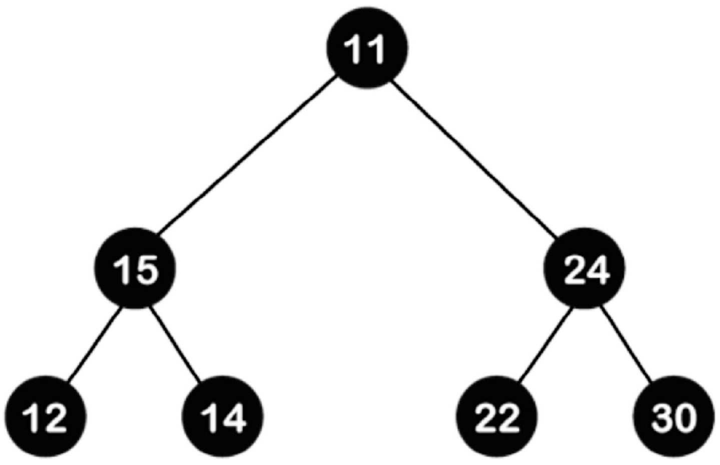
➢ **Min Heap:** The value of the parent node should be less than or equal to either of its children.

Or

In other words, the min-heap can be defined as, for every node i, the value of node i is greater than or equal to its parent value except the root node. Mathematically, it can be defined as:

A[Parent(i)] <= A[i]

Let's understand the min-heap through an example.



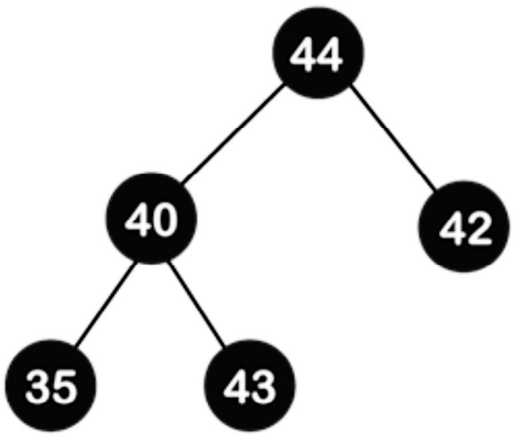
In the above figure, 11 is the root node, and the value of the root node is less than the value of all the other nodes (left child or a right child).

- **Max Heap:** The value of the parent node is greater than or equal to its children.

Or

In other words, the max heap can be defined as for every node i; the value of node i is less than or equal to its parent value except the root node. Mathematically, it can be defined as:

A[Parent(i)] >= A[i]



The above tree is a max heap tree as it satisfies the property of the max heap. Now, let's see the array representation of the max heap.

Time complexity in Max Heap

The total number of comparisons required in the max heap is according to the height of the tree. The height of the complete binary tree is always logn; therefore, the time complexity would also be O(logn).

Algorithm of insert operation in the max heap.

/ algorithm to insert an element in the max heap.

```
insertHeap(A, n, value)
{
n=n+1; // n is incremented to insert the new element
A[n]=value; // assign new value at the nth position
i = n; // assign the value of n to i
// loop will be executed until i becomes 1.
while(i>1)
{
parent= floor value of i/2; // Calculating the floor value of i/2
// Condition to check whether the value of parent is less than the given node or not
if(A[parent]<A[i])
{
swap(A[parent], A[i]);
i = parent;
}
else
{
return;
}
}
}
```

Let's understand the max heap through an example.

In the above figure, 55 is the parent node and it is greater than both of its child, and 11 is the parent of 9 and 8, so 11 is also greater than from both of its child. Therefore, we can say that the above tree is a max heap tree.

Insertion in the Heap tree

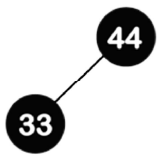
44, 33, 77, 11, 55, 88, 66

Suppose we want to create the max heap tree. To create the max heap tree, we need to consider the following two cases:

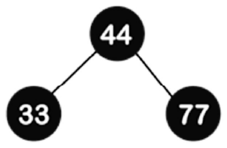
- First, we have to insert the element in such a way that the property of the complete binary tree must be maintained.
- Secondly, the value of the parent node should be greater than the either of its child.

Step 1: First we add the 44 element in the tree as shown below:

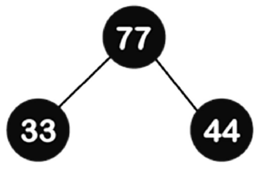
Step 2: The next element is 33. As we know that insertion in the binary tree always starts from the left side so 44 will be added at the left of 33 as shown below:



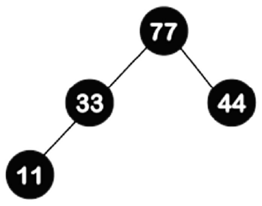
Step 3: The next element is 77 and it will be added to the right of the 44 as shown below:



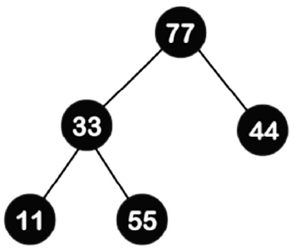
As we can observe in the above tree that it does not satisfy the max heap property, i.e., parent node 44 is less than the child 77. So, we will swap these two values as shown below:



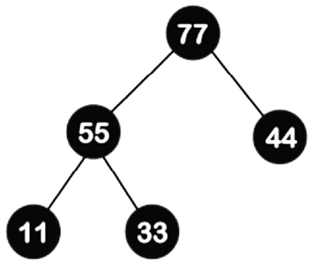
Step 4: The next element is 11. The node 11 is added to the left of 33 as shown below:



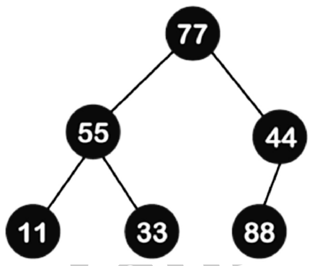
Step 5: The next element is 55. To make it a complete binary tree, we will add the node 55 to the right of 33 as shown below:



As we can observe in the above figure that it does not satisfy the property of the max heap because 33<55, so we will swap these two values as shown below:



Step 6: The next element is 88. The left subtree is completed so we will add 88 to the left of 44 as shown below:



As we can observe in the above figure that it does not satisfy the property of the max heap because 44<88, so we will swap these two values as shown below:

Again, it is violating the max heap property because 88>77 so we will swap these two values as shown below:

Step 7: The next element is 66. To make a complete binary tree, we will add the 66 element to the right side of 77 as shown below:

In the above figure, we can observe that the tree satisfies the property of max heap; therefore, it is a heap tree.

Deletion in Heap Tree

In Deletion in the heap tree, the root node is always deleted and it is replaced with the last element.

Let's understand the deletion through an example.

Step 1: In the above tree, the first 30 node is deleted from the tree and it is replaced with the 15 element as shown below:

Now we will heapify the tree. We will check whether the 15 is greater than either of its child or not. 15 is less than 20 so we will swap these two values as shown below:

Again, we will compare 15 with its child. Since 15 is greater than 10 so no swapping will occur.

Algorithm to heapify the tree

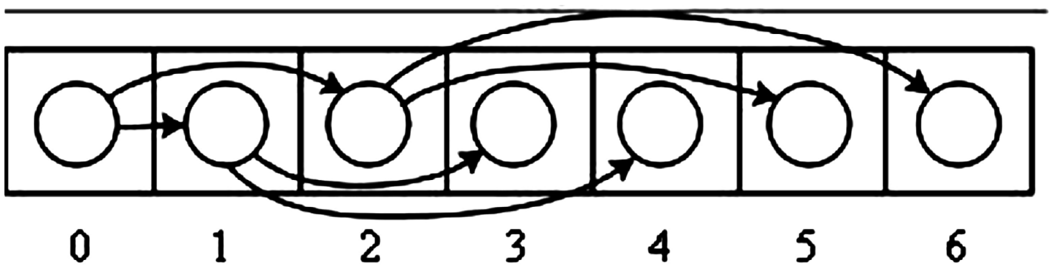
```
MaxHeapify(A, n, i)
{
int largest =i;
int l= 2i;
int r= 2i+1;
while(l<=n && A[l]>A[largest])
{
largest=l;
}
while(r<=n && A[r]>A[largest])
{
largest=r;
}
if(largest!=i)
{
swap(A[largest], A[i]);
heapify(A, n, largest);
}}
```

4.5.2 Implementation of Heap

Q26. Implement and build heap using an example

Ans :

A more common approach is to store the heap in an array. Since heap is always a complete binary tree, it can be stored compactly. No space is required for pointers; instead, the parent and children of each node can be found by simple arithmetic on array indices.



The rules (assume the root is stored in *arr*[0]):

- For each index *i*, element *arr*[*i*] has children at *arr*[2*i* + 1] and *arr*[2*i* + 2], and the parent at *arr*[floor((*i* - 1)/2)].

This implementation is particularly useful in the heapsort algorithm, where it allows the space in the input array to be reused to store the heap (i.e., the algorithm is in-place). However it requires allocating the array before filling it, which makes this method not that useful in priority queues implementation, where the number of elements is unknown.

It is perfectly acceptable to use a traditional binary tree data structure to implement a binary heap. There is an issue with finding the adjacent element on the last level on the binary heap when adding an element.

Building a Heap

A heap could be built by successive insertions. This approach requires O(n log n) time for n elements. Why?

The optimal method:

- Starts by arbitrarily putting the elements on a binary tree.
- Starting from the lowest level and moving upwards until the heap property is restored by shifting the root of the subtree downward as in the removal algorithm.
- If all the subtrees at some height h (measured from the bottom) have already been "heapified", the trees at hight h+1 can be heapified by sending their root down. This process takes O(h) swaps.

  As an example, let's build a heap with the following values: 20, 35, 23, 22, 4, 45, 21, 5, 42 and 19.

  As has been proved, this optimal method requires O(n) time for n elements.

4.5.2 Heap as Abstract Data Type

Q27. Write about heap data structure

OR

Explain , what is heapify?

Ans : **(July-21, Dec.-19, Nov.-17)**

Heap data structure is a complete binary tree that satisfies the heap property, where any given node is

- Always greater than its child node/s and the key of the root node is the largest among all other nodes. This property is also called max heap property.
- Always smaller than the child node/s and the key of the root node is the smallest among all other nodes. This property is also called min heap property.

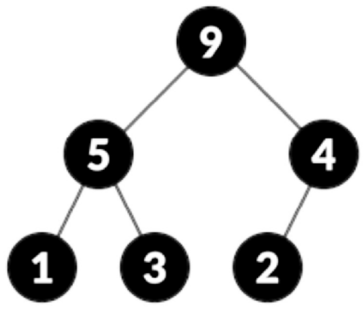


Fig.: Max heap

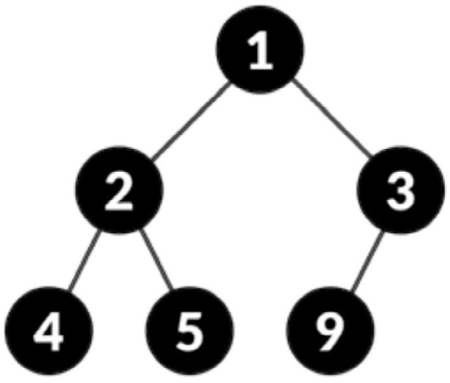
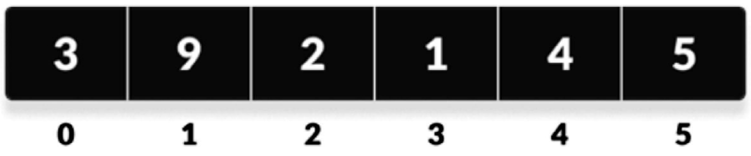
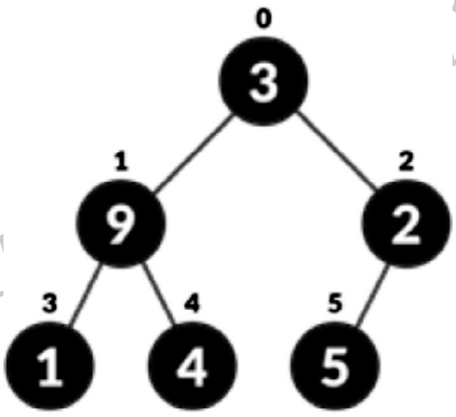


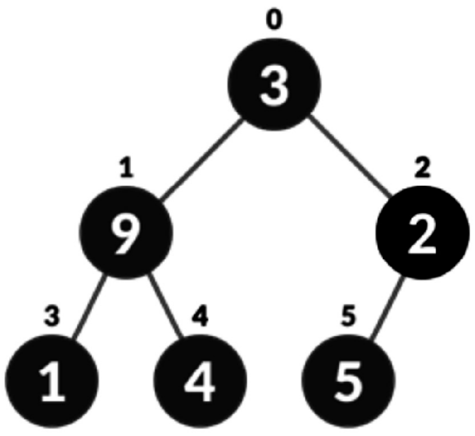
Fig.: Min heap



2. Create a complete binary tree from the arrayComplete binary tree



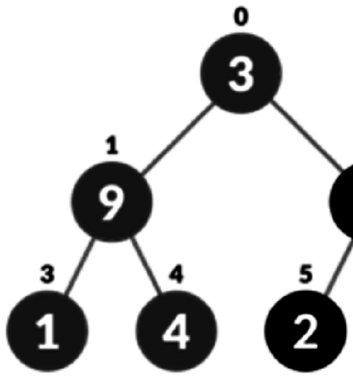
3. Start from the first index of non-leaf node whose index is given by n/2 – 1.



4. Start from the first on leaf node
5. Set current element i as largest.
6. The index of left child is given by 2i + 1 and the right child is given by 2i + 2.

   If leftChild is greater than currentElement (i.e. element at ith index), set leftChildIndex as largest.

   If right Child is greater than element in largest, set rightChildIndex as largest.



7. Swap largest with currentElementSwap if necessary
8. Repeat steps 3-7 until the subtrees are also heapified.

Algorithm

```
Heapify(array, size, i)
set i as largest
leftChild = 2i + 1
rightChild = 2i + 2
if leftChild >array[largest]
set leftChildIndex as largest
if rightChild >array[largest]
set rightChildIndex as largest
swap array[i] andarray[largest]
```

4.5.3 Heap Sort

Q28. What is heap sort? Write the heap sort algorithm? Explain with an example.

Ans : **(Dec.-18)**

Heapsort is a popular and efficient sorting algorithm. The concept of heap sort is to eliminate the elements one by one from the heap part of the list, and then insert them into the sorted part of the list.

Heapsort is the in-place sorting algorithm.

Algorithm

```
HeapSort(arr)
BuildMaxHeap(arr)
for i = length(arr) to 2
swap arr[1] with arr[i]
    heap_size[arr] = heap_size[arr] ? 1
    MaxHeapify(arr,1)
End
```

BuildMaxHeap(arr)

```
BuildMaxHeap(arr)
     heap_size(arr)
     = length(arr)
for i = length(arr)/2 to 1
MaxHeapify(arr,i)
End
```

MaxHeapify(arr,i)

```
MaxHeapify(arr,i)
      L = left(i)
      R = right(i)
if L ? heap_size[arr] and arr[L] > arr[i]
largest = L
else
largest = i
if R ? heap_size[arr] and arr[R] > arr[largest]
largest = R
if largest != i
swap arr[i] with arr[largest]
MaxHeapify(arr,largest)
End
```

Working of Heap sort Algorithm

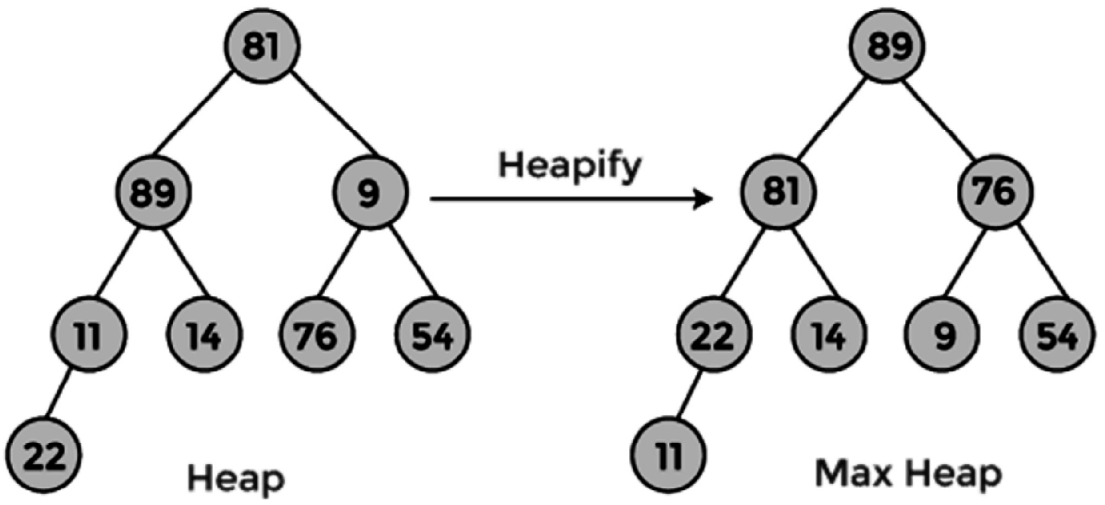
In heap sort, basically, there are two phases involved in the sorting of elements. By using the heap sort algorithm, they are as follows -

- The first step includes the creation of a heap by adjusting the elements of the array.
- After the creation of heap, now remove the root element of the heap repeatedly by shifting it to the end of the array, and then store the heap structure with the remaining elements.

Now let's see the working of heap sort in detail by using an example. To understand it more clearly, let's take an unsorted array and try to sort it using heap sort. It will make the explanation clearer and easier.

81	89	9	11	14	76	54	22

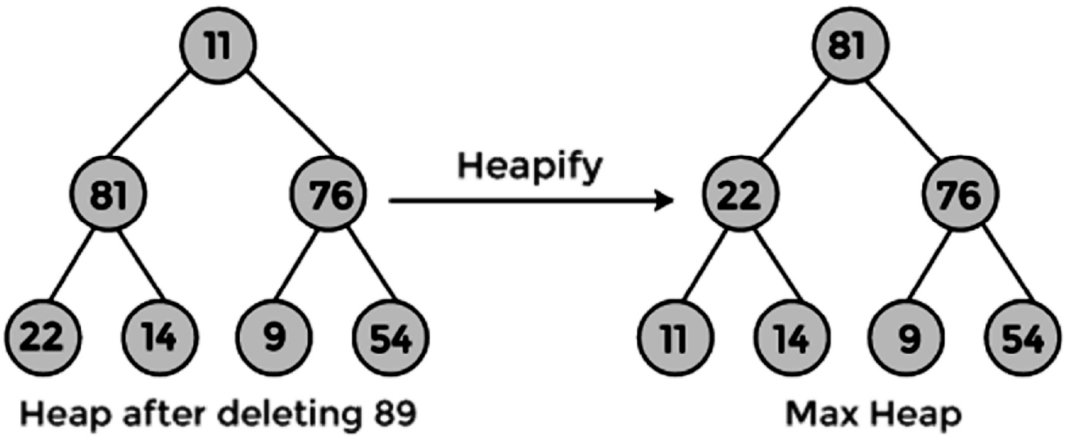
First, we have to construct a heap from the given array and convert it into max heap.



After converting the given heap into max heap, the array elements are -

89	81	76	22	14	9	54	11

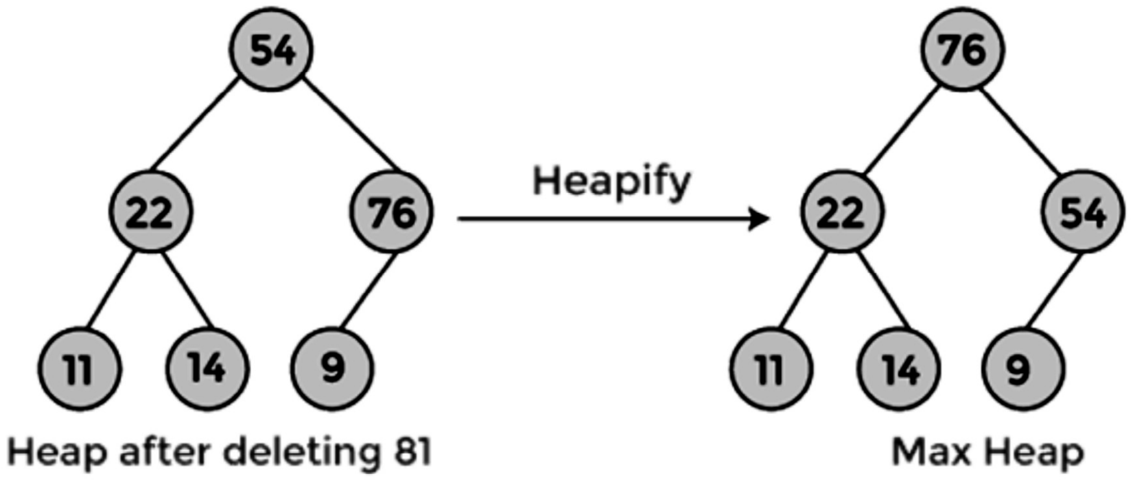
Next, we have to delete the root element (89) from the max heap. To delete this node, we have to swap it with the last node, i.e. (11). After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element 89 with 11, and converting the heap into max-heap, the elements of array are -

81	22	76	11	14	9	54	89

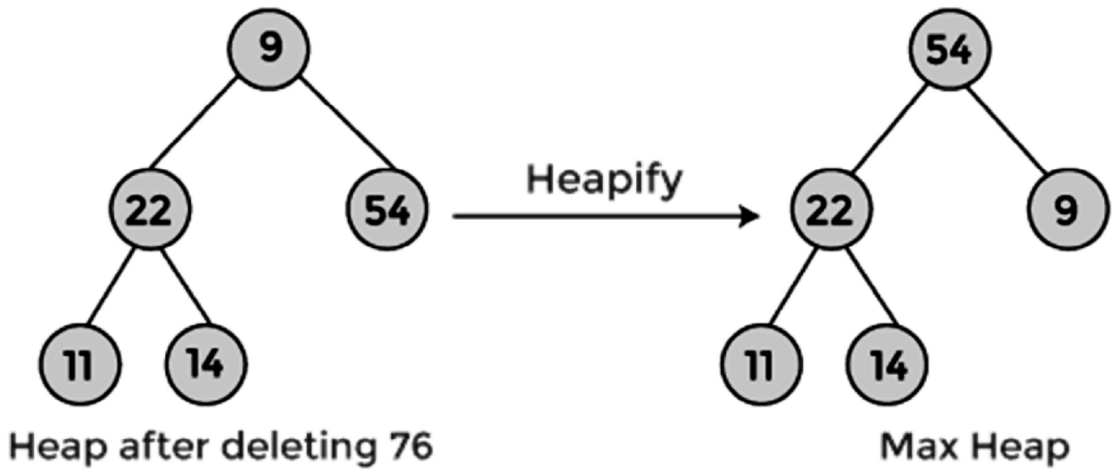
In the next step, again, we have to delete the root element (81) from the max heap. To delete this node, we have to swap it with the last node, i.e. (54). After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element 81 with 54 and converting the heap into max-heap, the elements of array are -

76	22	54	11	14	9	81	89

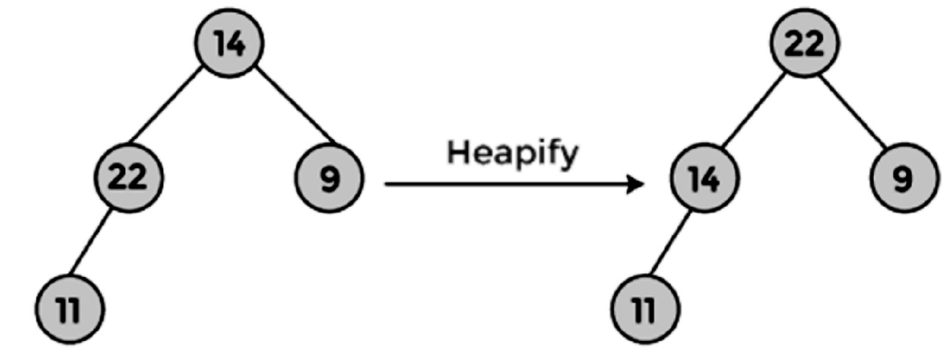
In the next step, we have to delete the root element (76) from the max heap again. To delete this node, we have to swap it with the last node, i.e. (9). After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element 76 with 9 and converting the heap into max-heap, the elements of array are -

54	22	9	11	14	76	81	89

In the next step, again we have to delete the root element (54) from the max heap. To delete this node, we have to swap it with the last node, i.e. (14). After deleting the root element, we again have to heapify it to convert it into max heap.

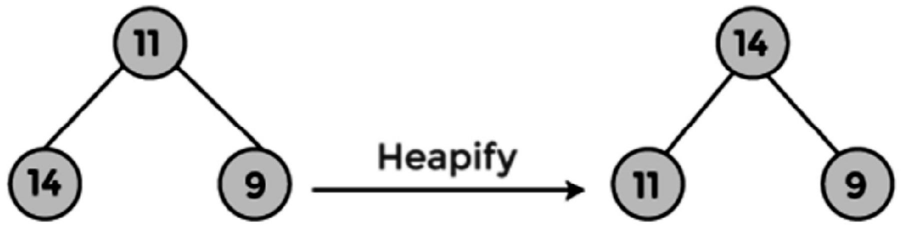


Heap after deleting 54 Max Heap

After swapping the array element 54 with 14 and converting the heap into max-heap, the elements of array are -

22	14	9	11	54	76	81	89

In the next step, again we have to delete the root element (22) from the max heap. To delete this node, we have to swap it with the last node, i.e. (11). After deleting the root element, we again have to heapify it to convert it into max heap.

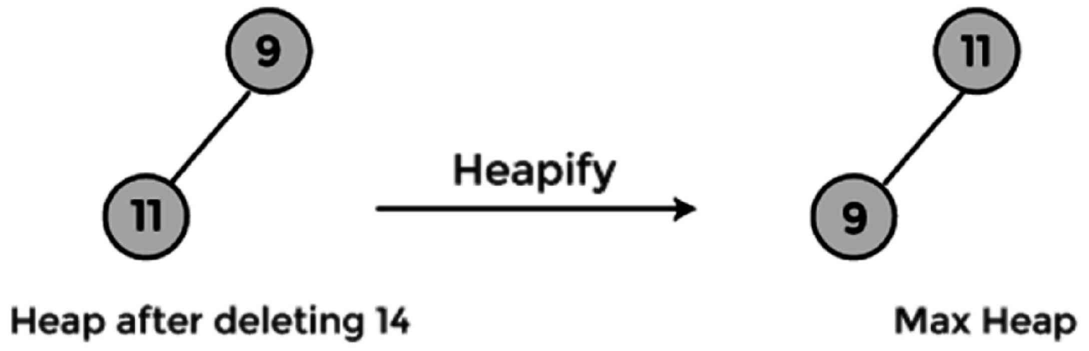


Heap after deleting 22 Max Heap

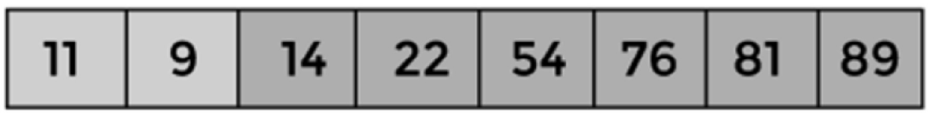
After swapping the array element 22 with 11 and converting the heap into max-heap, the elements of array are -

14	11	9	22	54	76	81	89

In the next step, again we have to delete the root element (14) from the max heap. To delete this node, we have to swap it with the last node, i.e. (9). After deleting the root element, we again have to heapify it to convert it into max heap.

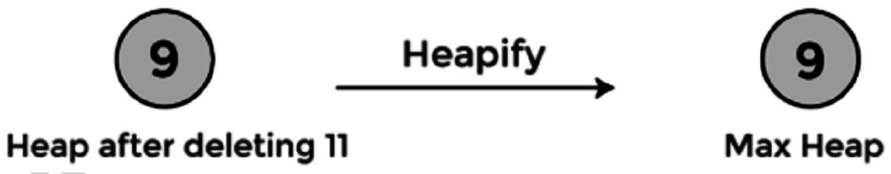


After swapping the array element 14 with 9 and converting the heap into max-heap, the elements of array are -

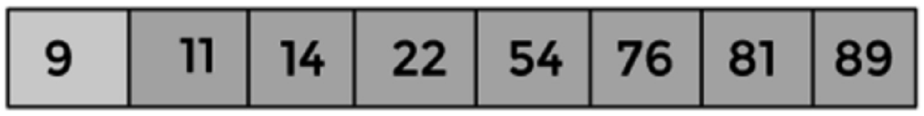


11	9	14	22	54	76	81	89

In the next step, again we have to delete the root element (11) from the max heap. To delete this node, we have to swap it with the last node, i.e. (9). After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element 11 with 9, the elements of array are -

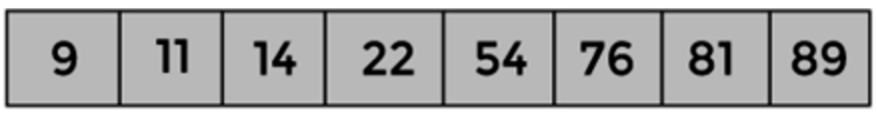


9	11	14	22	54	76	81	89

Now, heap has only one element left. After deleting it, heap will be empty.

9
Remove 9
Empty

After completion of sorting, the array elements are -



9	11	14	22	54	76	81	89

Now, the array is completely sorted.

Time Complexity

- Best Case O(n logn)
- Average Case O(n log n)
- Worst Case O(n log n)

Q29. write a program to implement heap sort

Ans :

```
// C++ program for implementation of Heap Sort
#include <iostream>
using namespace std;
   // To heapify a subtree rooted with node i which is
// an index in arr[]. n is size of heap
void heapify(int arr[], int n, int i)
{
      int largest = i; // Initialize largest as root
      int l = 2 * i + 1; // left = 2*i + 1
      int r = 2 * i + 2; // right = 2*i + 2
         // If left child is larger than root
      if (l < n && arr[l] > arr[largest])
            largest = l;
      // If right child is larger than largest so far
      if (r < n && arr[r] > arr[largest])
            largest = r;
         // If largest is not root
      if (largest != i) {
            swap(arr[i], arr[largest]);
           // Recursively heapify the affected sub-tree
            heapify(arr, n, largest);
      }}
  // main function to do heap sort
void heapSort(int arr[], int n)
{
      // Build heap (rearrange array)
      for (int i = n / 2 - 1; i >= 0; i—)
            heapify(arr, n, i);
         // One by one extract an element from heap
      for (int i = n - 1; i >= 0; i—) {
            // Move current root to end
            swap(arr[0], arr[i]);
              // call max heapify on the reduced heap
            heapify(arr, i, 0);
      }
}
   /* A utility function to print array of size n */
void printArray(int arr[], int n)
{
      for (int i = 0; i < n; ++i)
            cout << arr[i] << " ";
      cout << "\n";
}
   // Driver program
int main()
{
      int arr[] = { 12, 11, 13, 5, 6, 7 };
      int n = sizeof(arr) / sizeof(arr[0]);
         heapSort(arr, n);
         cout << "Sorted array is \n";
      printArray(arr, n);
}
```

Output:

```
Sorted array is
5 6 7 11 12 13
```

4.5.4 Heap Applications

Q30. Write the applications of heap sort.

Ans : **(Imp.)**

Heap Data Structure is generally taught with Heapsort. Heapsort algorithm has limited uses because Quicksort is better in practice. Nevertheless, the Heap data structure itself is enormously used. Following are some uses other than Heapsort.

Priority Queues

Priority queues can be efficiently implemented using Binary Heap because it supports insert(), delete() and extractmax(), decreaseKey() operations in O(logn) time. Binomoial Heap and Fibonacci Heap are variations of Binary Heap. These variations perform union also in O(logn) time which is a O(n) operation in Binary Heap. Heap Implemented priority queues are used in Graph algorithms like Prim's Algorithm and Dijkstra's algorithm.

Order statistics

The Heap data structure can be used to efficiently find the kth smallest (or largest) element in an array. See method 4 and 6 of this post for details.

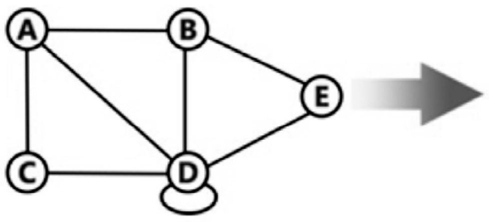
Short Question and Answers

1. Adjacency Matrix.

Ans :

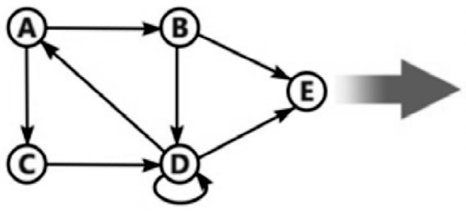
In this representation, graph can be represented using a matrix of size total number of vertices by total number of vertices. That means if a graph with 4 vertices can be represented using a matrix of 4X4 class. In this matrix, rows and columns both represents vertices. This matrix is filled with either 1 or 0. Here, 1 represents there is an edge from row vertex to column vertex and 0 represents there is no edge from row vertex to column vertex.

For example, consider the following undirected graph representation...



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 1 & 0 \\ B & 1 & 0 & 0 & 1 & 1 \\ C & 1 & 0 & 0 & 1 & 0 \\ D & 1 & 1 & 1 & 1 & 1 \\ E & 0 & 1 & 0 & 1 & 0 \end{array}$$

Directed graph representation...



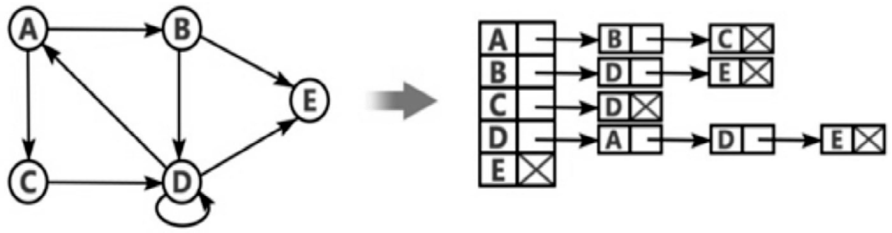
$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 0 & 0 \\ B & 0 & 0 & 0 & 1 & 1 \\ C & 0 & 0 & 0 & 1 & 0 \\ D & 1 & 0 & 0 & 1 & 1 \\ E & 0 & 0 & 0 & 0 & 0 \end{array}$$

2. Adjacency List.

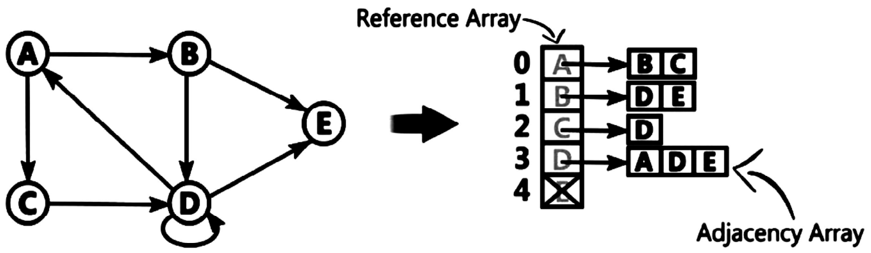
Ans :

In this representation, every vertex of graph contains list of its adjacent vertices.

For example, consider the following directed graph representation implemented using linked list...



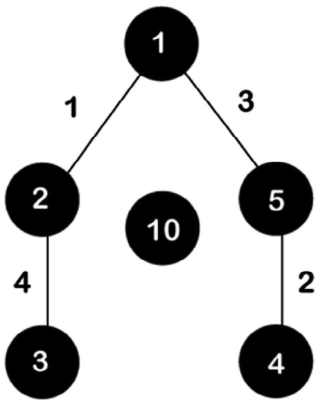
This representation can also be implemented using array as follows:



3. Minimum spanning trees.

Ans :

The minimum spanning tree is the tree whose sum of the edge weights is minimum. From the above spanning trees, the total edge weight of the spanning tree 1 is 12, the total edge weight of the spanning tree 2 is 14, and the total edge weight of the spanning tree 3 is 11; therefore, the total edge weight of the spanning tree 3 is minimum. From the above graph, we can also create one more spanning tree as shown below:



In the above tree, the total edge weight is 10 which is less than the above spanning trees; therefore, the minimum spanning tree is a tree which is having an edge weight, i.e., 10.

Properties of Spanning tree

- A connected graph can contain more than one spanning tree. The spanning trees which are minimally connected or we can say that the tree which is having a minimum total edge weight would be considered as the minimum spanning tree.
- All the possible spanning trees that can be created from the given graph G would have the same number of vertices, but the number of edges in the spanning tree would be equal to the number of vertices in the given graph minus 1.
- The spanning tree does not contain any cycle. Let's understand this property through an example.

4. What is Heap?

Ans :

A heap is a complete binary tree, and the binary tree is a tree in which the node can have utmost two children. Before knowing more about the heap data structure, we should know about the complete binary tree.

5. What is Hash Function?

Ans :

- A fixed process converts a key to a hash key is known as a Hash Function.
- This function takes a key and maps it to a value of a certain length which is called a Hash value or Hash.
- Hash value represents the original string of characters, but it is normally smaller than the original.
- It transfers the digital signature and then both hash value and signature are sent to the receiver. Receiver uses the same hash function to generate the hash value and then compares it to that received with the message.
- If the hash values are same, the message is transmitted without errors.

6. Min Heap

Ans :

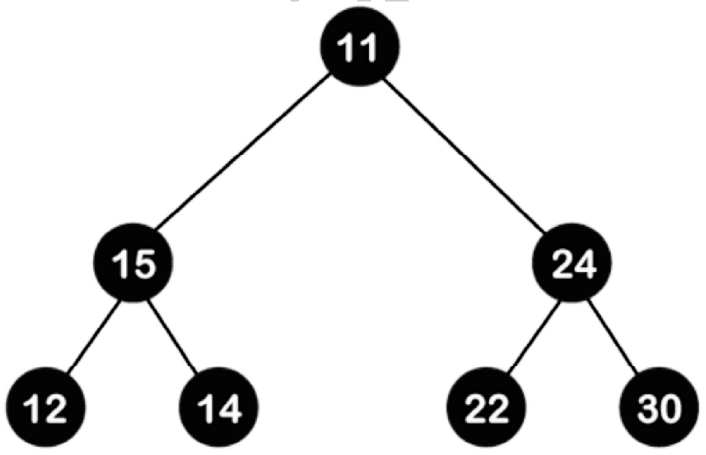
The value of the parent node should be less than or equal to either of its children.

Or

In other words, the min-heap can be defined as, for every node i, the value of node i is greater than or equal to its parent value except the root node. Mathematically, it can be defined as:

A[Parent(i)] < = A[i]

Let's understand the min-heap through an example.



In the above figure, 11 is the root node, and the value of the root node is less than the value of all the other nodes (left child or a right child).

7. Max Heap

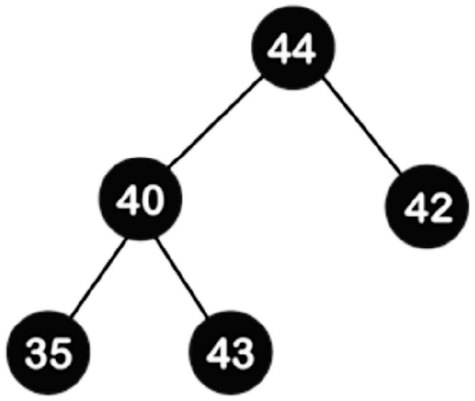
Ans :

The value of the parent node is greater than or equal to its children.

Or

In other words, the max heap can be defined as for every node i; the value of node i is less than or equal to its parent value except the root node. Mathematically, it can be defined as:

A[Parent(i)] > = A[i]



The above tree is a max heap tree as it satisfies the property of the max heap. Now, let's see the array representation of the max heap.

8. What is graph?

Ans :

Graph is a non linear data structure, it contains a set of points known as nodes (or vertices) and set of links known as edges (or Arcs) which connects the vertices. A graph is defined as follows:

Graph is a collection of vertices and arcs which connects vertices in the graph

Graph is a collection of nodes and edges which connects nodes in the graph

Generally, a graph G is represented as G = (V , E), where V is set of vertices and E is set of edges.

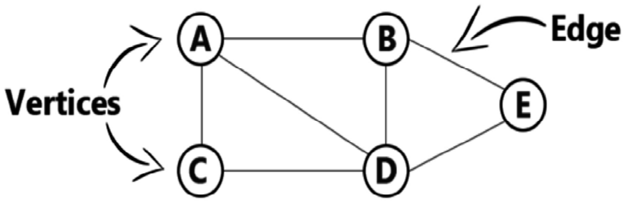
Example

The following is a graph with 5 vertices and 6 edges. This graph G can be defined as G = (V , E)

Where,

V = {A,B,C,D,E} and

E = {(A,B),(A,C)(A,D),(B,D), (C,D),(B,E), (E,D)}.



We use the following terms in graph data structure.

9. Representation of Graphs.

Ans :

Graph data structure is represented using following representations...

1. Adjacency Matrix
2. Incidence Matrix
3. Adjacency List

10. What is collision resolution technique?

Ans :

When the two different values have the same value, then the problem occurs between the two values, known as a collision. In the above example, the value is stored at index 6. If the key value is 26, then the index would be:

h(26) = 26%10 = 6

Therefore, two values are stored at the same index, i.e., 6, and this leads to the collision problem. To resolve these collisions, we have some techniques known as collision techniques.

The following are the collision techniques:

- Open Hashing: It is also known as closed addressing.
- Closed Hashing: It is also known as open addressing.

11. Heap as abstract data type.

Ans :

Heap data structure is a complete binary tree that satisfies the heap property, where any given node is

- Always greater than its child node/s and the key of the root node is the largest among all other nodes. This property is also called max heap property.
- Always smaller than the child node/s and the key of the root node is the smallest among all other nodes. This property is also called min heap property.

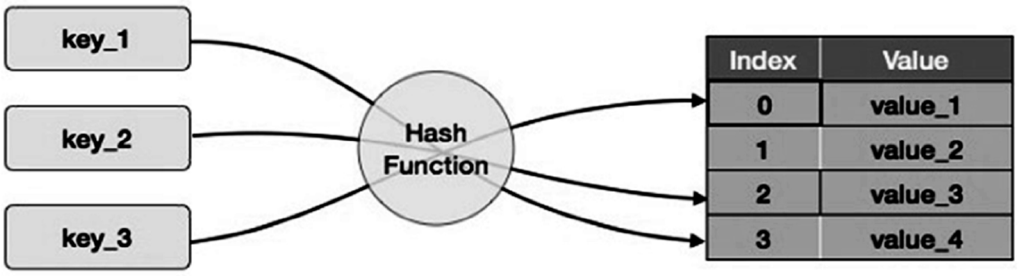
12. What is Hashing? Define it with an example.

Ans :

- Hashing is the process of mapping large amount of data item to smaller table with the help of hashing function.
- Hashing is also known as Hashing Algorithm or Message Digest Function.
- It is a technique to convert a range of key values into a range of indexes of an array.
- It is used to facilitate the next level searching method when compared with the linear or binary search.
- Hashing allows to update and retrieve any data entry in a constant time O(1).
- Constant time O(1) means the operation does not depend on the size of the data.
- Hashing is used with a database to enable items to be retrieved more quickly.
- It is used in the encryption and decryption of digital signatures.

Example

Consider an example of hash table of size 20, and the following items are to be stored. Item are in the (key,value) format. .



(1,20) , (2,70) , (42,80) , (4,25) , (12,44) , (14,32) , (17,11) , (13,78) , (37,98)

Sr. No.	Key	Hash	Array Index
1	1	1 % 20 = 1	1
2	2	2 % 20 = 2	2
3	42	42 % 20 = 2	2
4	4	4 % 20 = 4	4
5	12	12 % 20 = 12	12
6	14	14 % 20 = 14	14
7	17	17 % 20 = 17	17
8	13	13 % 20 = 13	13
9	37	37 % 20 = 17	17

Choose the Correct Answers

1. Which of the following algorithm design technique is used in the Quick Sort Algorithm? [c]

   (a) Dynamic Programming (b) Back tracking

   (c) Divide and Conqure (d) Greedy Method

2. Merge Sort uses [a]

   (a) Dynamic Programming (b) Back tracking

   (c) Heuristic Approach (d) Greedy Method

3. The Complexity of Bubble sort in best case is [a]

   (a) $\theta(n)$ (b) $\theta(n\log n)$

   (c) $\theta(n^2)$ (d) $\theta(n(\text{long } n)^2)$

4. Which of the following algorithm pays the least attention to the ordering of the elements in the input? [b]

   (a) Insert Sort (b) Selection Sort

   (c) Quick Sort (d) None

5. Which of the following is not a stable Algorithm [b]

   (a) Insertion Sort (b) Selection Sort

   (c) Bubble Sort (d) Merge Sort

6. Finding the location of the elements with a given values is [b]

   (a) Traversal (b) Searching

   (c) Sorting (d) None

7. Which of the following is not a non-comparison sort [d]

   (a) Counting Sort (b) Bucket Sort

   (c) Radix Sort (d) Shell Sort

8. _________ is putting an element in the appropriate place in a sorted list. [a]

   (a) Insertion Sort (b) Extraction Sort

   (c) Selection Sort (d) Merge Sort

9. Partition and Exchange sorting is _________ [a]

   (a) Quick Sort (b) Tree Sort

   (c) Heap Sort (d) Bubble Sort

10. Which of the following sorting algorithm of priority queue uses [a]

    (a) Bubble Sort (b) Insertion Sort

    (c) Merge Sort (d) Selection Sort

Fill in the Blanks

1. ________ is a process of locating a particular element in a set of Elements.
2. Sorting refers to ________ in a increasing or decreasing fashion.
3. Quick sort partitions an array which are called as ________
4. Merge sort technique is based on ________
5. Bubble sort algorithm is based on ________
6. The complexity of Bubble Sort algorithm is ________
7. Heap data structure is a ________ based data structure.
8. ________ is a fully binary tree.
9. As per ADT ________ is a data type which stores a collection of elements.
10. ________ is one of the best sorting method with no worst case.

ANSWERS

1. Searching
2. Ordering /Arranging Data
3. Sub Array
4. Divide and Conqure
5. Comparision
6. O(n^2)
7. Binary Tree
8. Max Heap
9. Heap
10. Heap Sort

One Mark Answers

1. Define a graph.

Ans :

A graph is defined as G = (V,E) where,

(i) V is the set of elements called nodes or vertices or points.

(ii) E is the set of edges of the graph identified with a unique pair (U, V) of nodes.

Here (U, V) pair denotes that there is an edge from node U to node V.

2. Define hashing.

Ans :

Hashing is a technique that makes use of a hash function for mapping pairs to the corresponding (entries) positions in a hash table. Ideally a pair 'P' with a key 'K' is stored at a position f(K) by the hash function '*f*'. Each hash table position can store one pair.

3. Define hash function.

Ans :

It is a function that is used to map the dictionary pairs to the corresponding entries in the hash table.

If the dictionary pair is 'pair' and if it consists of the key 'key and if '/ denotes the hash function then the pair "pair" is stored at f (key) position in a table.

In order to search a pair with the key K, first we find f(K) and determine whether a pair is at f (K) position in the hash table. If it is not present at f(K) then we insert the pair at f(K) else if it occurs f(K) then we find the desired pair that can be deleted if required.

4. Define hash table.

Ans :

It is a data structure that is used to store dictionary pairs. It is of fixed size. The 'search' operation is applied to a part of the dictionary pair called the key. The size of the hash table is represented by a variable called the table size which ranges from q to (table size –1).

5. Define heap.

Ans :

A complete binary tree in which the value node is either greater than, lesser than or equal to values in child nodes is called a heap.

Heap can be classified as,

(i) Ascending heap (min heap)

(ii) Descending heap (max heap)

Practicals

1. Write C++ programs to implement the following using an array

Ans :

Stack ADT

```
#include<iostream>
#include<conio.h>
using namespace std;
class Stack
{
int stk[10];
int top;
public:
Stack()
   {
top=-1;
   }
//This funtion is used to push an elemnt into stack
void push(int x)
   {
if(isFull())
       {
cout<<"Stack is full";
return;
       }
top++;
stk[top]=x;
cout<<"Inserted Element:"<<" "<<x;
   }
   //This functionn is used to check if the stack is full
bool isFull()
   {
int size=sizeof(stk)/sizeof(*stk);
if(top == size-1)
        {
return true;
        }
else
        {
return false;
        }
    }

    //This function is used to check if the stack is empty
bool isEmpty()
    {
if(top==-1)
        {
return true;
        }
else
        {
return false;
        }
    }

    //This function is used to pop an element of stack
void pop()
    {
if(isEmpty())
        {
cout<<"Stack is Empty";
return;
        }
cout<<"Deleted element is:" <<" "<<stk[top--];
    }
    //This function is used to display the elements of the stack
void display()
```

```
   {
if(top==-1)
       {
cout<<" Stack is Empty!!";
return;
       }
for(int i=top;i>=0;i--)
       {
cout<<stk[i] <<" ";
       }
   }
};

void main()
{
int ch;
    Stack st;
while(1)
       {
cout<<"\n1.Push  2.Pop  3.Display  4.Exit\nEnter ur choice";
cin>> ch;
switch(ch)
            {
case 1:  cout <<"Enter the element you want to push";
cin>> ch;
st.push(ch);
break;
case 2:  st.pop();
break;
case 3:  st.display();
break;
case 4:  exit(0);
             }
        }
}
```

Sample Output:

```
1.Push  2.Pop  3.Display  4.Exit
Enter ur choice 1
Enter the element you want to push 11
Inserted Element: 11
1.Push  2.Pop  3.Display  4.Exit
Enter ur choice 1
Enter the element you want to push 22
Inserted Element: 22
1.Push  2.Pop  3.Display  4.Exit
Enter ur choice 1
Enter the element you want to push 33
Inserted Element: 33
1.Push  2.Pop  3.Display  4.Exit
Enter ur choice 1
Enter the element you want to push 44
Inserted Element: 44
1.Push  2.Pop  3.Display  4.Exit
Enter ur choice 1
Enter the element you want to push 55
Inserted Element: 55
1.Push  2.Pop  3.Display  4.Exit
Enter ur choice 3
55 44 33 22 11
```

b) Queue ADT

```
#include <iostream>
using namespace std;
const int Max=10;
void Qinsert(int Q[],int &R,int F)
{
if ((R+1)%Max!=F)
   {
      R=(R+1)%Max;
cout<<"Data:";
cin>>Q[R];
```

```
   }
else
cout<<"Queue is Full!"<<endl;
}
void Qdelete(int Q[],int R,int &F)
{
if (R!=F)
   {
      F=(F+1)%Max;
cout<<Q[F]<<" Deleted!";
   }
else
cout<<"Queue is empty"<<endl;
}
void Qdisplay(int Q[],int R,int F)
{
int Cn=F;
while (Cn!=R)
   {
      Cn=(Cn+1)%Max;
cout<<Q[Cn]<<endl;
   }
}
int main()
{  //Initialisation Steps
int Que[Max],Rear=0,Front=0;
char Ch;
do
   {
cout<<"\n I:Insert/D:Delete/S:Show/Q:Quit ";
cin>>Ch;
switch(Ch)
      {
case 'I':Qinsert(Que,Rear,Front);
break;
case 'D':Qdelete(Que,Rear,Front);
break;
case 'S':Qdisplay(Que,Rear,Front);
break;
        }
}while (Ch!='Q');
return 0 ;
}
Output:
        I:Insert/D:Delete/S:Show/Q:Quit I
Data:10
        I:Insert/D:Delete/S:Show/Q:Quit I
Data:20
        I:Insert/D:Delete/S:Show/Q:Quit D
10 Deleted!
        I:Insert/D:Delete/S:Show/Q:Quit I
Data:30
        I:Insert/D:Delete/S:Show/Q:Quit I
Data:40
        I:Insert/D:Delete/S:Show/Q:Quit S
20
30
40
        I:Insert/D:Delete/S:Show/Q:Quit Q
...Program finished with exit code 0
Press ENTER to exit console.
```

2. Write a C++ program to implement Circular queue using array.

Ans :

```
/* Circular Queue */
#include<iostream>
#include<climits>
#define MAX 50
using namespace std;
// gloabal variables - queue_array, front, rear
```

```
int queue[MAX];
int front=-1,rear=-1;
// Utility to Enqueue an element to the Queue
void enqueue(int num)
{
        if((rear==MAX-1 && front==0) || (front==rear+1))
        {
              cout<<"Queue is Full !!\n";
              return;
        }
        if(rear == MAX-1) // front != 0
              rear = -1;
        rear++;
        queue[rear] = num;
        if(front == -1)
              front=0;
}
// Utility to Dequeue an element from Queue
void dequeue()
{
        if(front==-1)
        {
              cout<<"Queue is Empty !!\n";
              return;
        }
        int number;
        number = queue[front];
        queue[front] = INT_MAX;

        if(front == rear)
        {
              front = -1;
              rear = -1;
        }
        else if(front == MAX-1) //rear != MAX-1
```

```
            front = 0;
        else
            front++;

        cout<<"Dequeued element = "<<number<<"\n";
}
// Utility to display Front of Queue
void queue_front()
{
        if(front == -1)
        {
            cout<<"Queue is Empty !!\n";
            return;
        }
        cout<<"Front of the queue = "<<queue[front]<<"\n";
}
// Utility to display Rear of Queue
void queue_rear()
{
        if(rear == -1)
        {
            cout<<"Queue is Empty !!\n";
            return;
        }
        cout<<"Rear of the  queue = "<<queue[rear]<<"\n";
}
// Utility to display entire Queue
void disp_queue()
{
        if(front == -1)
        {
            cout<<"Queue is Empty !!\n";
            return;
        }
        int i;
        for(i=0;i<MAX;i++)
        {
              if(queue[i] == INT_MAX)
                     continue;
              cout<<queue[i]<<" ";
        }
        cout<<"\n";
}
// Driver Code
int main()
{
        int optn,number;

        for(int i=0;i<MAX;i++)
              queue[i] = INT_MAX;
        while(1)
        {
              cout<<"\nOperations available : \n";
              cout<<"1. ENQUEUE\n2. DEQUEUE\n3. Front\n4. Rear\n5. DISPLAY queue\n6. Exit\n\n";
              cout<<"Enter your choice : ";
              cin>>optn;
                     switch(optn)
              {
                     case(1):
                            cout<<"Enter the number : ";
                            cin>>number;
                            enqueue(number);
                            break;
                     case(2):
                            dequeue();
                            break;
                     case(3):
                            queue_front();
                            break;
```

```
                        case(4):
                                queue_rear();
                                break;
                        case(5):
                                disp_queue();
                                break;
                        case(6):
                                exit(0);
                }
        }
        return 0;
}
```

Output :

```
Operations available :
1. ENQUEUE
2. DEQUEUE
3. Front
4. Rear
5. DISPLAY queue
6. Exit
Enter your choice : 1
Enter the number : 12
Operations available :
1. ENQUEUE
2. DEQUEUE
3. Front
4. Rear
5. DISPLAY queue
6. Exit
Enter your choice : 1
Enter the number : 13
Operations available :
1. ENQUEUE
2. DEQUEUE
3. Front
```

```
4. Rear
5. DISPLAY queue
6. Exit
Enter your choice : 1
Enter the number : 14
Operations available :
1. ENQUEUE
2. DEQUEUE
3. Front
4. Rear
5. DISPLAY queue
6. Exit
Enter your choice : 5
12 13 14
```

3. Write C++ programs to implement the following using a single linked list.

Ans :

Stack ADT

```
#include <iostream>
#include <string.h>
usingnamespace std;
struct Node{
int stu_no;
char stu_name[50];
int p;
   Node *next;
};
Node *top;
class stack{
public:
void push(int n,char name[],int perc);
void pop();
void display();
};
void stack :: push(int n,char name[],int perc)
```

```
{
struct Node *newNode=new Node;
   //fill data part
newNode->stu_no=n;
newNode->p=perc;
strcpy(newNode->stu_name,name);
   //link part
newNode->next=top;
   //make newnode as top/head
top=newNode;
}
void stack ::pop()
{
if(top= =NULL){
cout<<"List is empty!"<<endl;
return;
   }
    cout<<top->stu_name<<" is removed."<<endl;
top=top->next;
}
void stack:: display()
{
if(top= =NULL){
cout<<"List is empty!"<<endl;
return;
   }
struct Node *temp=top;
while(temp!=NULL){
cout<<temp->stu_no<<" ";
cout<<temp->stu_name<<" ";
cout<<temp->p<<" ";
cout<<endl;
temp=temp->next;
   }
```

```
cout<<endl;
}
int main(){

stack s;
char ch;
do{
int n;
cout<<"ENTER CHOICE\n"<<"1.Push\n"<<"2.Pop\n"<<"3.Display\n";
cout<<"Make a choice: ";
cin>>n;
switch(n){
case1:
        Node n;
cout<<"Enter details of the element to be pushed : \n";
cout<<"Roll Number : ";
cin>>n.stu_no;
cout<<"Enter Name: ";
std::cin.ignore(1);
cin.getline(n.stu_name,50);
cout<<"Enter Percentage: ";
cin>>n.p;
        //push data into the stack
s.push(n.stu_no,n.stu_name,n.p);
break;
case2 :
        //pop data from stack
s.pop();
break;
case3 :
        //display data
s.display();
break;
default :
```

```
cout<<"Invalid Choice\n";
   }
cout<<"Do you want to continue ? : ";
cin>>ch;
}while(ch=='Y'||ch=='y');
return 0;
}
```

OUTPUT

```
R CHOICE
1. Push
2. Pop
3.Display
Make a choice: 1
Enter details of the element to be pushed :
Roll Number : 101
Enter Name: PRIYA
Enter Percentage: 99
Do you want to continue ? : y
ENTER CHOICE
1.Push
2.Pop
3.Display
Make a choice: 1
Enter details of the element to be pushed :
Roll Number : 102
Enter Name: Mahak
Enter Percentage: 95
Do you want to continue ? : y
ENTER CHOICE
1.Push
2.Pop
3.Display
Make a choice: 3
102 Mahak 95
101 PRIYA 99
```

b) Queue ADT

```
#include<iostream>
usingnamespace std;
struct Node{
int data;
    Node *next;
};
class Queue{
    public:
    Node *front,*rear;
Queue(){front=rear=NULL;}
void insert(int n);
void deleteitem();
void display();
     ~Queue();
};
void Queue::insert(int n){
    Node *temp=new Node;
if(temp==NULL){
cout<<"Overflow"<<endl;
return;
    }
temp->data=n;
temp->next=NULL;
    //for first node
if(front==NULL){
front=rear=temp;
    }
else{
rear->next=temp;
rear=temp;
    }
    cout<<n<<" has been inserted successfully."<<endl;
}
```

```
void Queue::display(){
if(front= =NULL){
cout< <"Underflow."< <endl;
return;
   }
   Node *temp=front;
   //will check until NULL is not found
while(temp){
cout< <temp->data< <" ";
temp=temp->next;
   }
cout< <endl;
}
void Queue :: deleteitem()
   {
if (front= =NULL){
cout< <"underflow"< <endl;
return;
   }
cout< <front->data< <" is being deleted "< <endl;
if(front= =rear)//if only one node is there
front=rear=NULL;
else
front=front->next;
}
Queue ::~Queue()
{
while(front!=NULL)
   {
      Node *temp=front;
front=front->next;
delete temp;
   }
rear=NULL;
}
```

```
int main(){
   Queue Q;
Q.display();
Q.insert(10);
Q.insert(24);
Q.insert(28);
Q.insert(32);
Q.insert(30);
Q.display();
Q.deleteitem();
Q.deleteitem();
Q.deleteitem();
Q.deleteitem();
Q.deleteitem();
return 0;
}
```

Output

```
Underflow.
   10 has been inserted successfully.
   24 has been inserted successfully.
   28 has been inserted successfully.
   32 has been inserted successfully.
   30 has been inserted successfully.
   10 24 28 32 30
   10 is being deleted
   24 is being deleted
   28 is being deleted
   32 is being deleted
   30 is being deleted
```

4. Write a C++ program to implement Circular queue using Single linked list.

Ans :

```
#include <iostream>;
using namespace std;
class Queue {
   public:
   // Initialize front and rear
   int rear, front;
   // Circular Queue
   int size;
   int *circular_queue;
   Queue(int sz) {
     front = rear = -1;
     size = sz;
     circular_queue = new int[sz];
   }
   void enQueue(int elem);
   int deQueue();
   void displayQueue();
};
/* Function to create Circular queue */
void Queue::enQueue(int elem)
{
   if ((front == 0 &amp;&amp; rear == size-1) || (rear == (front-1)%(size-1))) {
       cout&lt;&lt;"\nQueue is Full";
       return;
   }
   else if (front == -1) {     /* Insert First Element */
       front = rear = 0;
       circular_queue[rear] = elem;
   }
   else if (rear == size-1 &amp;&amp; front != 0) {
       rear = 0;
       circular_queue[rear] = elem;
   }
   else {
       rear++;
       circular_queue[rear] = elem;
   }
}
```

```
// Function to delete element from Circular Queue
int Queue::deQueue()
{
    if (front = = -1)  {
        cout&lt;&lt;"\nQueue is Empty";
        return -1;
    }
    int data = circular_queue[front];
    circular_queue[front] = -1;
    if (front = = rear)  {
        front = -1;
        rear = -1;
    }
    else if (front = = size-1)
        front = 0;
    else
        front++;
    return data;
}
//display elements of Circular Queue
void Queue::displayQueue()
{
    if (front = = -1) {
        cout&lt;&lt;"\nQueue is Empty"&lt;&lt;endl;
        return;
    }
    cout&lt;&lt;"\nCircular Queue elements: ";
    if (rear &gt;= front) {
        for (int i = front; i &lt;= rear; i++)
            cout&lt;&lt;circular_queue[i]&lt;&lt;" ";
    }
    Else  {
        for (int i = front; i &lt; size; i++)
            cout&lt;&lt;circular_queue[i]&lt;&lt;" ";
        for (int i = 0; i &lt;= rear; i++)
```

```
            cout&lt;&lt;circular_queue[i]&lt;&lt;" ";
    }
}
//main program
int main()
{
    Queue pq(5);
    // Insert elements in Circular Queue
     pq.enQueue(2);
     pq.enQueue(4);
     pq.enQueue(6);
     pq.enQueue(8);
    // Display elements present in Circular Queue
    pq.displayQueue();
    // Delete elements from Circular Queue
     cout&lt;&lt;"\nElement Dequeued = "&lt;&lt;pq.deQueue();
     cout&lt;&lt;"\nElement Dequeued = "&lt;&lt;pq.deQueue();
    pq.displayQueue();
     pq.enQueue(10);
     pq.enQueue(12);
     pq.enQueue(14);
    pq.displayQueue();
     pq.enQueue(10);
    return 0;
}
```

Output:

```
Circular Queue elements: 2 4 6 8
Element Dequeued = 2
Element Dequeued = 4
Circular Queue elements: 6 8
Circular Queue elements: 6 8 10 12 14
Queue is Full
```

5. Write a C++ program to implement the double ended queue ADT using double linked list.

Ans :

```
#include<iostream>
#include<conio.h>
#include<stdlib.h>
using namespace std;
class node
{
public:
   int data;
   class node *next;
   class node *prev;
};
class dqueue: public node
{
   node *head,*tail;
    int top1,top2;
public:
    dqueue()
   {
       top1=0;
       top2=0;
      head=NULL;
      tail=NULL;
   }
   void push(int x)
   {
       node *temp;
      int ch;
       if(top1+top2 >=5)
      {
          cout <<"dqueue overflow";
         return ;
```

```
}
if( top1+top2 == 0)
{
    head = new node;
    head->data=x;
    head->next=NULL;
    head->prev=NULL;
    tail=head;
    top1++;
}
else
{
    cout <<" Add element 1.FIRST 2.LAST\n Enter Your Choice:";
    cin >> ch;
    if(ch==1)
    {
        top1++;
        temp=new node;
        temp->data=x;
        temp->next=head;
        temp->prev=NULL;
        head->prev=temp;
        head=temp;
    }
    else
    {
        top2++;
        temp=new node;
        temp->data=x;
        temp->next=NULL;
        temp->prev=tail;
        tail->next=temp;
        tail=temp;
    }
    }
}
void pop()
{
    int ch;
    cout <<"Delete 1.First Node 2.Last Node\n Enter Your Choice:";
    cin >>ch;
    if(top1 + top2 <=0)
    {
        cout <<"\nDqueue under flow";
        return;
    }
    if(ch==1)
    {
        head=head->next;
        head->prev=NULL;
        top1--;
    }
    else
    {
        top2--;
        tail=tail->prev;
        tail->next=NULL;
    }
}
void display()
{
    int ch;
    node *temp;
    cout <<"Display From 1.STARTING 2.ENDING\n Enter Your Choice";
    cin >>ch;
    if(top1+top2 <=0)
    {
        cout <<"under flow";
```

```
            return ;
        }
        if (ch==1)
        {
            temp=head;
            while(temp!=NULL)
            {
                cout << temp->data <<" ";
                temp=temp->next;
            }
        }
        else
        {
            temp=tail;
            while( temp!=NULL)
            {
                cout <<temp->data << " ";
                temp=temp->prev;
            }
        }
    }
};
int main()
{
    dqueue d1;
    int ch;
    while (1)
    {
        cout <<"1.INSERT 2.DELETE 3.DISPLAY 4.EXIT\n Enter Your Choice:";
        cin >>ch;
        switch(ch)
        {
        case 1:
            cout <<"Enter the Element: ";
```

```
          cin > > ch;
           d1.push(ch);
          break;
      case 2:
           d1.pop();
          break;
      case 3:
          d1.display();
          break;
      case 4:
          exit(1);
      }
   }<br>  return 0;
}
```

```
 OUTPUT:
1.INSERT 2.DELETE 3.DISPLAY 4.EXIT
 Enter Your Choice:1
Enter the Element: 10
1.INSERT 2.DELETE 3.DISPLAY 4.EXIT
 Enter ur choice:1
Enter the Element: 15
 Add element 1.FIRST 2.LAST
 Enter Your Choice:2
1.INSERT 2.DELETE 3.DISPLAY 4.EXIT
 Enter Your Choice:3
Display From 1.STARTING 2.ENDING
 Enter Your Choice:1
10 15 1.INSERT 2.DELETE 3.DISPLAY 4.EXIT
 Enter Your Choice:4
```

6. Write a C++ program to solve tower of Hanoi problem recursively

Ans :

```
#include<iostream>
usingnamespace std;
//tower of HANOI function implementation
```

```
void TOH(int n,char Sour, char Aux,char Des)
{
        if(n==1)
        {
              cout<<"Move Disk "<<n<<" from "<<Sour<<" to "<<Des<<endl;
              return;
        }
        TOH(n-1,Sour,Des,Aux);
        cout<<"Move Disk "<<n<<" from "<<Sour<<" to "<<Des<<endl;
        TOH(n-1,Aux,Sour,Des);
}
//main program
int main()
{
        int n;
        cout<<"Enter no. of disks:";
        cin>>n;
        //calling the TOH
        TOH(n,'A','B','C');
        return 0;
}
```

Output

```
Enter no. of disks:3
Move Disk 1 from A to C
Move Disk 2 from A to B
Move Disk 1 from C to B
Move Disk 3 from A to C
Move Disk 1 from B to A
Move Disk 2 from B to C
Move Disk 1 from A to C
```

7. Write C++ program to perform the following operations:

Ans :

Insert an element into a binary search tree.

```
#include<iostream>
```

```
usingnamespace std;
template<classt>
classbst
{
struct NODE
{
t data;
  NODE* left, *right;
};
public:
NODE* createnewnode(t data)
{
   NODE* createnewnode = newNODE();
createnewnode->data = data;
createnewnode->left = createnewnode->right = NULL;
        return createnewnode;
}
NODE* Insert(NODE* root,t data)
{
        if(root == NULL)
        {
        root = createnewnode(data);
        }
        elseif(data <= root->data)
        {
        root->left = Insert(root->left,data);
        }
        else
        {
        root->right = Insert(root->right,data);
        }
        return root;
}
voidinorder(NODE* root )
```

```
{
        if(root != NULL)
        {
        inorder(root->left);
        cout<<root->data<<" ";
        inorder(root->right);
        }
}
bst()
{
NODE* root = NULL;
t a,b,c,d,e,f;
cout<<endl<<"Enter the data"<<endl;
cin>>a;
cin>>b;
cin>>c;
cin>>d;
cin>>e;
cin>>f;
        root = Insert(root,a);
        root = Insert(root,b);
        root = Insert(root,c);
        root = Insert(root,d);
        root = Insert(root,e);
        root = Insert(root,f);
        cout<<"***THE BINARY SEARCH TREE*** "<<endl;
        cout<<"Inorder:";
        inorder(root);
        cout<<endl;
        insertion(root);
}
NODE* insertion(NODE* root)
{
        t n;
        cout<<"Enter the ELEMENT to be inserted in the BST"<<endl;
        cin>>n;
root = Insert(root,n);
cout<<"***THE BINARY SEARCH TREE(2)***"<<endl;
cout<<"Inorder:";
        inorder(root);
}
};
intmain()
{
        bst<int>a;
        cout<<endl<<"-------------------------------------------"<<endl;
        bst<char>b;
        return0;
}
```

```
25
31
12
15
16
13
***THE BINARY SEARCH TREE***
Inorder:12 13 15 16 25 31
Enter the ELEMENT to be inserted in the BST
17
***THE BINARY SEARCH TREE(2)***
Inorder:12 13 15 16 17 25 31
Enter the data
g
u
q
i
***THE BINARY SEARCH TREE***
Inorder: a g i q t u
```

```
Enter the ELEMENT to be inserted in the BST
y
***THE BINARY SEARCH TREE(2)***
Inorder: a g i q t u y
Process exited after 39.67 seconds with return value 0
Press any key to continue . . .
```

b) Delete an element from binary search tree.

```
#include<iostream>
usingnamespace std;
bool c=false;
struct node
{
        int data;
        node* left;
        node* right;
};
struct node* getNode(int data)
{
        node* newNode=new node();
        newNode->data=data;
        newNode->left=NULL;
        newNode->right=NULL;
        return newNode;
}
void inorder(struct node* root)
{
if (root != NULL)
    {
inorder(root->left);
cout<<root->data<<" ";
inorder(root->right);
   }
}
node* findMin(node*root)
```

```
{
while(root->left!=NULL)
   {
root=root->left;
   }
return root;
}
struct node* Insert(struct node* root, int data)
{
        if (root == NULL)
               return getNode(data);
        if (data < root->data)
               root->left  = Insert(root->left, data);
        elseif (data > root->data)
               root->right = Insert(root->right, data);
        return root;
}
bool Search(node*root,int value)
{
        if(root==NULL)
               returnfalse;
        elseif(root->data == value)
        {
               returntrue;
        }
        elseif(value < root-> data)
               Search(root->left,value);
        elseif(value > root->data)
               Search(root->right,value);
}
node* Delete( node* root,int value)
{
        c=Search(root,value);
        if(root==NULL)
               return root;
        elseif(value< root->data)
        {
               root->left= Delete(root->left,value);
        }
        elseif(value> root->data)
        {
               root->right= Delete(root->right,value);
        }
        // Node deletion
        else
        {
               //case 1: Leaf Node
               if(root->left==NULL&&root->right==NULL)
               {
                   delete root;
                   root=NULL;
                   return root;
               }
               //case 2: one child
               elseif(root->left==NULL)
               {
                   struct node* temp=root;
                   root=root->right;
                   delete temp;
                   return root;
               }
               elseif(root->right==NULL)
               {
                   struct node* temp=root;
                   root=root->left;
                   delete temp;
                   return root;
               }
               //case 3: 2 child
               else
               {
                   struct node*temp=findMin(root->right);
                   root->data=temp->data;
                   root->right=Delete(root->right,temp->data);
               }
        }
        return root;
    }
    int main()
    {
        node* root=NULL;
        root=Insert(root,20);
        Insert(root,15);
        Insert(root,25);
        Insert(root,18);
        Insert(root,10);
        Insert(root,16);
        Insert(root,19);
        Insert(root,17);
        cout<<"Before Deletion: "<<endl;
        cout<<"Inorder: ";
        inorder(root);
        cout<<endl<<endl;
        Delete(root,15);
        if(c)
        {
               cout<<"Node Deleted"<<endl;
               cout<<"\nAfter Deletion: "<<endl;
               cout<<"Inorder: ";
               inorder(root);
               cout<<endl;
        }
        else
```

```
                cout<<"Node Not Found"<<endl;

        return 0;
}
```

Output

```
Before Deletion:
Inorder: 10 15 16 17 18 19 20 25
Node Deleted
After Deletion:
Inorder: 10 16 17 18 19 20 25
```

c) Search for a key in a binary search tree.

```
#include <iostream>
using namespace std;
// Data structure to store a BST node
struct Node
{
    int data;
    Node* left = nullptr, *right = nullptr;
    Node() {}
    Node(int data): data(data) {}
};
 // Recursive function to insert a key into a BST
Node* insert(Node* root, int key)
{
    // if the root is null, create a new node and return it
    if (root == nullptr) {
        return new Node(key);
    }
    // if the given key is less than the root node, recur for the left subtree
    if (key < root->data) {
        root->left = insert(root->left, key);
    }
    // if the given key is more than the root node, recur for the right subtree
    else {
        root->right = insert(root->right, key);
    }
    return root;
}
// Recursive function to search in a given BST
void search(Node* root, int key, Node* parent)
{
    // if the key is not present in the key
    if (root == nullptr)
    {
        cout << "Key not found";
        return;
    }
    // if the key is found
    if (root->data == key)
    {
        if (parent == nullptr) {
            cout << "The node with key " << key << " is root node";
        }
        else if (key < parent->data)
        {
            cout << "The given key is the left node of the node with key "
                 << parent->data;
        }
        else {
            cout << "The given key is the right node of the node with key "
                 << parent->data;
        }
        return;
    }
    // if the given key is less than the root node, recur for the left subtree;
    // otherwise, recur for the right subtree
    if (key < root->data) {
        search(root->left, key, root);
    } else {
        search(root->right, key, root);
    }
}
 int main()
{
    int keys[] = { 15, 10, 20, 8, 12, 16, 25 };
     Node* root = nullptr;
    for (int key: keys) {
        root = insert(root, key);
    }
     search(root, 25, nullptr);
     return 0;
}
```

Output:

```
The given key is the right node of the node with key 20
```

8. Write C++ programs for the implementation tree traversal technique BFS.

Ans :

```
#include <iostream>
#include <conio.h>
usingnamespace std;
int c =0, t =0;
struct node_info
{
int no;
int st_time;
}*q =NULL, *r =NULL, *x =NULL;
struct node
{
    node_info *pt;
node*next;
}*front =NULL, *rear =NULL, *p =NULL, *np =NULL;
void push(node_info *ptr)
{
np=new node;
```

```
np->pt = ptr;
np->next =NULL;
if(front ==NULL)
{
front= rear = np;
rear->next =NULL;
}
else
{
rear->next = np;
rear= np;
rear->next =NULL;
}
}
node_info *remove()
{
if(front ==NULL)
{
cout<<"empty queue\n";
}
else
{
      p = front;
      x = p->pt;
front= front->next;
delete(p);
return(x);
}
}
void bfs(int*v,int am[][7],int i)
{
if(c ==0)
{
      q =new node_info;
      q->no = i;
```

```
        q->st_time = t++;
cout<<"time of visitation for node "<<q->no<<":"<<q->st_time<<"\n\n";
v[i]=1;
push(q);
}
c++;
for(int j =0; j <7; j++)
{
if(am[i][j]==0||(am[i][j]==1&& v[j]==1))
continue;
elseif(am[i][j]==1&& v[j]==0)
{
            r =new node_info;
r->no = j;
r->st_time = t++;
cout<<"time of visitation for node "<<r->no<<":"<<r->st_time<<"\n\n";
v[j]=1;
push(r);
}
}
remove();
if(c <=6&& front !=NULL)
bfs(v, am, remove()->no);
}
int main()
{
int v[7], am[7][7];
for(int i =0; i <7; i++)
v[i]=0;
for(int i =0; i <7; i++)
{
cout<<"enter the values for adjacency matrix row:"<<i+1<<endl;
for(int j =0; j <7; j++)
```

```
{
cin> >am[i][j];
}
}
bfs(v, am, 0);
getch();
}
```

Output

```
enter the values for adjacency matrix row:1
0
1
1
0
0
1
1
enter the values for adjacency matrix row:2
1
0
0
0
0
0
0
enter the values for adjacency matrix row:3
1
0
0
0
0
0
1
time of visitation for node 0:0
time of visitation for node 1:1
time of visitation for node 2:2
```

9. Write a C++ program that uses recursive functions to traverse a binary search tree.

Ans :

a) Pre-order

b) In-order

c) Post-order

```
// C++ program for different tree traversals
#include <iostream>
using namespace std;
 /* A binary tree node has data, pointer to left childand a pointer to right child */
struct Node {
    int data;
    struct Node *left, *right;
    Node(int data)
    {
        this->data = data;
        left = right = NULL;
    }
};
 /* Given a binary tree, print its nodes according to the
"bottom-up" postorder traversal. */
void printPostorder(struct Node* node)
{
    if (node == NULL)
        return;
     // first recur on left subtree
     printPostorder(node->left);
     // then recur on right subtree
     printPostorder(node->right);
     // now deal with the node
    cout << node->data << " ";
}
 /* Given a binary tree, print its nodes in inorder*/
void printInorder(struct Node* node)
{
    if (node == NULL)
        return;
     /* first recur on left child */
     printInorder(node->left);
     /* then print the data of node */
    cout << node->data << " ";
     /* now recur on right child */
     printInorder(node->right);
}
 /* Given a binary tree, print its nodes in preorder*/
void printPreorder(struct Node* node)
{
    if (node == NULL)
        return;
     /* first print data of node */
    cout << node->data << " ";
     /* then recur on left subtree */
     printPreorder(node->left);
     /* now recur on right subtree */
     printPreorder(node->right);
}
 /* Driver program to test above functions*/
int main()
{
    struct Node* root = new Node(1);
    root->left = new Node(2);
    root->right = new Node(3);
    root->left->left = new Node(4);
    root->left->right = new Node(5);
     cout << "\nPreorder traversal of binary tree is \n";
     printPreorder(root);
     cout << "\nInorder traversal of binary tree is \n";
     printInorder(root);
     cout << "\nPostorder traversal of binary tree is \n";
     printPostorder(root);
    return 0;
}
```

Output:

```
Preorder traversal of binary tree is
1 2 4 5 3
Inorder traversal of binary tree is
4 2 5 1 3
Postorder traversal of binary tree is
4 5 2 3 1
```

10. Write a C++ program to find height of a tree.

Ans :

```
#include <iostream>
using namespace std;
// Data structure to store a Binary Tree node
struct Node
{
int key;
   Node *left, *right;

Node(int key)
   {
this->key = key;
this->left = this->right = NULL;
   }
};
// Recursive function to calculate height of given binary tree
int height(Node* root)
{
// Base case: empty tree has height 0
if (root == NULL)
return 0;
// recur for left and right subtree and consider maximum depth
return 1 + max(height(root->left), height(root->right));
}
```

```
// main function
int main()
{
   Node* root = NULL;
root = new Node(10);
root->left = new Node(1);
root->right = new Node(25);
root->left->left = new Node(9);
root->left->right = new Node(10);
root->right->left = new Node(13);
root->right->right = new Node(20);
cout<< "The height of the binary tree is " << height(root);
return 0;
}
```

Output :

```
The height of binary tree is 3.
```

11 Write a C++ program to find MIN and MAX element of a BST.

Ans :

```
#include<iostream>
using namespace std;
struct BSTNode
{
int data;
BSTNode *left;
   BSTNode *right;
};
 BSTNode* getnewnode(int data)
{
 BSTNode* temp=new BSTNode();
temp->data=data;
     temp->left=temp->right=NULL;
      return temp;
}
BSTNode* insertnode(struct BSTNode *root,int data)
```

```
{
if(root= =NULL)
{
     root=getnewnode(data);
}
else if(data< =root->data)
{
root->left=insertnode(root->left,data);
}
else
{
root->right=insertnode(root->right,data);
}
return root;
}
bool search(BSTNode *root,int data)
{
if(root= =NULL) return false;
else if(root->data= =data) return true;
else if(data<root->data)
 return search(root->left,data);
 else
 return search(root->right,data);
}
void preorder(BSTNode* root)
{
if(root= =NULL)
return ;
else
{
cout< <root->data< <" ";
preorder(root->left);
preorder(root->right);
}
```

```
}
void inorder(BSTNode* root)
{
if(root= =NULL)
return ;
else
{
inorder(root->left);
cout< <root->data< <" ";
inorder(root->right);
}
}
void postorder(BSTNode* root)
{
if(root= =NULL)
return ;
else
{
    postorder(root->left);
postorder(root->right);
    cout< <root->data< <" ";
}
}
int findmin(BSTNode* root) //itterative method
{
if(root= =NULL)
{
cout< <"\nerror: tree is empty ";
return -1;
}
BSTNode* current=root;
while(current->left)
current=current->left;
return current->data;
```

```
}
int findmax(BSTNode* root)   //itterative method find the maximum element
{
if(root==NULL)
{
cout<<"\n error: tree is empty ";
return -1;
}
BSTNode* current=root;
while(current->right)
current=current->right;
return current->data;
}
int recursive_findMin(BSTNode* root)
{
if(root==NULL)
{
cout<<"\n error: tree is empty ";
return -1;
}
else if(root->left==NULL)
 return root->data;
    return recursive_findMin(root->left);
}
int recursive_findMax(BSTNode *root)
{
if(root==NULL)
{
cout<<"\n error: tree is empty ";
return -1;
}
else if(root->right==NULL)
 return root->data;
    return recursive_findMax(root->right);
```

```
}
int findHeight(BSTNode* root)
{
if(root==NULL)
return -1;
return max(findHeight(root->left),findHeight(root->right))+1;
}
int main()
{
int x;
struct BSTNode* root=NULL;
root=insertnode(root,10);
     root=insertnode(root,15);
root=insertnode(root,9);
root=insertnode(root,8);
     root=insertnode(root,7);
root=insertnode(root,6);
cout<<"enter the data to search ";
cin>>x;
if(search(root,x)==true)
    cout<<"\n data is found ";
else
cout<<"\n data is not found";
cout<<"\npreorder traversal ";
preorder(root);
cout<<"\ninorder traversal ";
inorder(root);
cout<<"\npostorder traversal ";
postorder(root);
cout<<"\n height of tree= "<<findHeight(root);
 cout<<"\n minimum element by itterative method: "<<findmin(root);
 cout<<"\n maximum element by itterative method: "<<findmax(root);
 cout<<"\n maximum element by recursive method : "<< recursive_findMax(root);
  return 0;
}
```

Output:-

```
enter the data to search 15
        data Is found
preorder traversal 10 9 8 7 6 15
inorder traversal 6 7 8 9 10 15
postorder traversal 6 7 8 9 15 10
        height of tree= 4
        minimum element by itterative method: 6
        maximum element by itterative method: 15
        minimum element by recursive method : 6
        maximum element by recursive method : 15
Process exited after 9.164 seconds with return value 0
Press any key to continue ...
```

12 Write a C++ program to find Inorder Successor of a given node.

Ans :

```
#include <iostream>
using namespace std;
 // Data structure to store a BST node
struct Node
{
   int data;
   Node* left = nullptr, *right = nullptr;
    Node() {}
   Node(int data): data(data) {}
};
 // Recursive function to insert a key into a BST
Node* insert(Node* root, int key)
{
   // if the root is null, create a new node and return it
   if (root == nullptr) {
       return new Node(key);
   }
   // if the given key is less than the root node, recur for the left subtree
   if (key < root->data) {
       root->left = insert(root->left, key);
   }
    // if the given key is more than the root node, recur for the right subtree
   else {
       root->right = insert(root->right, key);
   }
    return root;
}
 // Helper function to find minimum value node in a given BST
Node* findMinimum(Node* root)
{
   while (root->left) {
       root = root->left;
   }
    return root;
}
 // Recursive function to find an inorder successor for the given key in a BST.
// Note that successor `succ` is passed by reference to the function
Node* findSuccessor(Node* root, Node* succ, int key)
{
   // base case
   if (root == nullptr) {
       return succ;
   }
    // if a node with the desired value is found, the successor is the minimum value
   // node in its right subtree (if any)
   if (root->data == key)
   {
       if (root->right != nullptr) {
           return findMinimum(root->right);
       }
   }
   // if the given key is less than the root node, recur for the left subtree
   else if (key < root->data)
   {
       // update successor to the current node before recursing in the left subtree
       succ = root;
       return findSuccessor(root->left, succ, key);
   }
    // if the given key is more than the root node, recur for the right subtree
   else {
       return findSuccessor(root->right, succ, key);
   }
    return succ;
}
 int main()
{
   int keys[] = { 15, 10, 20, 8, 12, 16, 25 };

   /* Construct the following tree
             15
           /    \
          /      \
         10       20
        / \      / \
       /   \    /   \
       8    12  16    25
   */
    Node* root = nullptr;
   for (int key: keys) {
       root = insert(root, key);
   }

   // find inorder successor for each key
   for (int key: keys)
   {
       Node* succ = findSuccessor(root, nullptr, key);
       if (succ != nullptr) {
           cout << "The successor of node " << key << " is " << succ->data;
       }
       else {
           cout << "No Successor exists for node " << key;
       }
        cout << endl;
   }
    return 0;
}
```

Output:

```
The successor of node 15 is 16
The successor of node 10 is 12
The successor of node 20 is 25
The successor of node 8 is 10
The successor of node 12 is 15
The successor of node 16 is 20
No Successor exists for node 25
```

13. Write C++ programs to perform the following operations on B-Trees and AVL Trees.

Ans :

a) Insertion b) Deletion

```
#include<iostream>
#include<stdlib.h>
using namespace std;
#define TRUE 1
#define FALSE 0
#define NULL 0
class AVL;
class AVLNODE
{
   friend class AVL;
private:
    int data;
    AVLNODE *left,*right;
    int bf;
```

```
};
class AVL
{
private:
   AVLNODE *root;
public:
   AVLNODE *loc,*par;
   AVL()
   {
      root=NULL;
   }
   int insert(int);
   void displayitem();
   void display(AVLNODE *);
   void removeitem(int);
   void remove1(AVLNODE *,AVLNODE *,int);
   void remove2(AVLNODE *,AVLNODE *,int);
   void search(int x);
   void search1(AVLNODE *,int);
};
int AVL::insert(int x)
{
   AVLNODE *a,*b,*c,*f,*p,*q,*y,*clchild,*crchild;
   int found,unbalanced;
   int d;
   if(!root) //special case empty tree
   {
      y=new AVLNODE;
      y->data=x;
      root=y;
      root->bf=0;
      root->left=root->right=NULL;
      return TRUE;
   }
```

```
//phase 1:locate insertion point for x.a keeps track of the most
// recent node with balance factor +/-1,and f is the parent of a
// q follows p through the tree.
   f=NULL;
   a=p=root;
   q=NULL;
   found=FALSE;
   while(p&&!found)
   {
      //search for insertion point for x
      if(p->bf)
      {
         a=p;
         f=q;
      }
      if(x<p->data) //take left branch
      {
         q=p;
         p=p->left;
      }
      else if(x>p->data)
      {
         q=p;
         p=p->right;
      }
      else
      {
         y=p;
         found=TRUE;
      }
   } //end while
//phase 2:insert and rebalance.x is not in the tree and
// may be inserted as the appropriate child of q.
   if(!found)
```

```
  {
      y = new AVLNODE;
      y->data=x;
      y->left=y->right=NULL;
      y->bf=0;
      if(x<q->data) //insert as left child
          q->left=y;
      else
          q->right=y; //insert as right child
//adjust balance factors of nodes on path from a to q
//note that by the definition of a,all nodes on this
//path must have balance factors of 0 and so will change
//to +/- d=+1 implies that x is inserted in the left
// subtree of a d=-1 implies
//to that x inserted in the right subtree of a.
      if(x>a->data)
      {
          p=a->right;
          b=p;
          d=-1;
      }
      else
      {
          p=a->left;
          b=p;
          d=1;
      }
      while(p!=y)
          if(x>p->data) //height of right increases by 1
          {
              p->bf=-1;
              p=p->right;
          }
          else //height of left increases by 1
          {
              p->bf=1;
              p=p->left;
          }
//is tree unbalanced
        unbalanced=TRUE;
         if(!(a->bf)||!(a->bf+d))
        {
           //tree still balanced
            a->bf+=d;
           unbalanced=FALSE;
        }
         if(unbalanced) //tree unbalanced,determine rotation type
        {
            if(d==1)
            {
               //left imbalance
                if(b->bf==1) //rotation type LL
               {
                   a->left=b->right;
                   b->right=a;
                   a->bf=0;
                   b->bf=0;
               }
               else //rotation type LR
               {
                   c=b->right;
                   b->right=c->left;
                   a->left=c->right;
                   c->left=b;
                   c->right=a;
                   switch(c->bf)
                   {
                   case 1:
                        a->bf=-1; //LR(b)
          b->bf=0;
          break;
        case -1:
          b->bf=1; //LR(c)
          a->bf=0;
          break;
        case 0:
          b->bf=0; //LR(a)
          a->bf=0;
          break;
        }
        c->bf=0;
        b=c; //b is the new root
      } //end of LR
    } //end of left imbalance
    else //right imbalance
    {
      if(b->bf==-1) //rotation type RR
      {
        a->right=b->left;
        b->left=a;
        a->bf=0;
        b->bf=0;
      }
      else //rotation type LR
      {
        c=b->right;
        b->right=c->left;
        a->right=c->left;
        c->right=b;
        c->left=a;
        switch(c->bf)
        {
        case 1:
                a->bf=-1; //LR(b)
                b->bf=0;
                break;
            case -1:
                b->bf=1; //LR(c)
                a->bf=0;
                break;
            case 0:
                b->bf=0; //LR(a)
                a->bf=0;
                break;
            }
            c->bf=0;
            b=c; //b is the new root
        } //end of LR
    }
//subtree with root b has been rebalanced and is the new subtree
    if(!f)
        root=b;
    else if(a==f->left)
        f->left=b;
    else if(a==f->right)
        f->right=b;
  } //end of if unbalanced
  return TRUE;
 } //end of if(!found)
 return FALSE;
} //end of AVL INSERTION
void AVL::displayitem()
{
 display(root);
}
void AVL::display(AVLNODE *temp)
{
   if(temp==NULL)
      return;
   cout<<temp->data<<" ";
   display(temp->left);
   display(temp->right);
}
void AVL::removeitem(int x)
{
   search(x);
   if(loc==NULL)
   {
      cout<<"\nitem is not in tree :(";
      return;
   }
   if(loc->right!=NULL&&loc->left!=NULL)
      remove1(loc,par,x);
   else
      remove2(loc,par,x);
}
void AVL::remove1(AVLNODE *l,AVLNODE *p,int x)
{
   AVLNODE *ptr,*save,*suc,*psuc;
   ptr=l->right;
   save=l;
   while(ptr->left!=NULL)
   {
      save=ptr;
      ptr=ptr->left;
   }
   suc=ptr;
   psuc=save;
   remove2(suc,psuc,x);
   if(p!=NULL)
      if(l==p->left)
```

```
            p->left=suc;
        else
            p->right=suc;
    else
        root=l;
    suc->left=l->left;
    suc->right=l->right;
    return;
}
void AVL::remove2(AVLNODE *s,AVLNODE *p,int x)
{
    AVLNODE *child;
    if(s->left==NULL && s->right==NULL)
        child=NULL;
    else if(s->left!=NULL)
        child=s->left;
    else
        child=s->right;
    if(p!=NULL)
        if(s==p->left)
            p->left=child;
        else
            p->right=child;
    else
        root=child;
}
void AVL::search(int x)
{
    search1(root,x);
}
void AVL::search1(AVLNODE *temp,int x)
{
    AVLNODE *ptr,*save;
    int flag;
```

```
if(temp= =NULL)
{
    cout< <"\nthe tree is empty";
    return;
}
if(temp->data= =x)
{
    cout< <"\nThe item is root and is found :)";
    par=NULL;
    loc=temp;
    par->left=NULL;
    par->right=NULL;
    return;
}
if( x < temp->data)
{
    ptr=temp->left;
    save=temp;
}
else
{
    ptr=temp->right;
    save=temp;
}
while(ptr!=NULL)
{
    if(x= =ptr->data)
    {
        flag=1;
        cout< <"\nItemfound :)";
        loc=ptr;
        par=save;
    }
    if(x<ptr->data)
```

```
            ptr=ptr->left;
         else
            ptr=ptr->right;
    }
    if(flag!=1)
    {
        cout<<"Item is not there in tree :(";
        loc=NULL;
        par=NULL;
        cout<<loc;
        cout<<par;
    }
}
int main()
{
   AVL a;
    int x,y,c;
    char ch;
    do
    {
        cout<<"\n1.Insert";
        cout<<"\n2.Display";
        cout<<"\n3.Delete";
        cout<<"\n4.Search";
        cout<<"\n5.Exit";
        cout<<"\nEnter your choice of operation on AVL Tree :";
        cin>>c;
        switch(c)
        {
        case 1:
             cout<<"\nEnter an Element to be inserted into Tree :";
            cin>>x;
            a.insert(x);
            break;
        case 2:
            a.displayitem();
            break;
        case 3:
             cout<<"\nEnter an item for Deletion :";
            cin>>y;
             a.removeitem(y);
            break;
        case 4:
             cout<<"\nEnter an element to perform Search :";
            cin>>c;
            a.search(c);
            break;
        case 5:
            exit(0);
            break;
        default :
            cout<<"\nInvalid option try again";
        }
         cout<<"\nDo u want to continue (y/n) :";
        cin>>ch;
    }
    while(ch=='y'||ch=='Y');<br>     return 0;
}
```

OUTPUT:

```
1.Insert
2.Display
3.Delete
4.Search
5.Exit
Enter your choice of operation on AVL Tree :1
Enter an Element to be inserted into Tree :10
Do u want to continue (y/n) :y
1.Insert
```

```
2.Display
3.Delete
4.Search
5.Exit
Enter your choice of operation on AVL Tree :1
Enter an Element to be inserted into Tree :14
Do u want to continue (y/n) :y
1.Insert
2.Display
3.Delete
4.Search
5.Exit
Enter your choice of operation on AVL Tree :1
Enter an Element to be inserted into Tree :9
Do u want to continue (y/n) :y
1.Insert
2.Display
3.Delete
4.Search
5.Exit
Enter your choice of operation on AVL Tree :2
10 9 14
```

14 Write C++ programs for sorting a given list of elements in ascending order using the following sorting methods.

Ans :

a) Quick sort

```
#include<iostream>
#include<cstdlib>
usingnamespace std;
void swap(int*a,int*b){
   int temp;
   temp=*a;
   *a =*b;
   *b = temp;
}
```

```
intPartition(int a[],int l,int h){
  int pivot, index, i;
  index= l;
  pivot= h;
  for(i = l; i < h; i++){
    if(a[i]< a[pivot]){
      swap(&a[i],&a[index]);
      index++;
    }
  }
  swap(&a[pivot],&a[index]);
  return index;
}
intRandomPivotPartition(int a[],int l,int h){
  int pvt, n, temp;
  n =rand();
  pvt= l + n%(h-l+1);
  swap(&a[h],&a[pvt]);
  returnPartition(a, l, h);
}
intQuickSort(int a[],int l,int h){
  int pindex;
  if(l < h){
    pindex=RandomPivotPartition(a, l, h);
    QuickSort(a, l, pindex-1);
    QuickSort(a, pindex+1, h);
  }
  return0;
}
int main(){
  int n, i;
  cout<<"\nEnter the number of data element to be sorted: ";
  cin>>n;
  int arr[n];
  for(i =0; i < n; i++){
     cout<<"Enter element "<<i+1<<": ";
    cin>>arr[i];
  }
   QuickSort(arr,0, n-1);
  cout<<"\nSorted Data ";
  for(i =0; i < n; i++)
     cout<<"->"<<arr[i];
  return0;
}
```

Output

```
Enter the number of data element to be sorted: 4 Enter element 1: 3
Enter element 2: 4
Enter element 3: 7
Enter element 4: 6
Sorted Data ->3->4->6->7
```

b) Merge sort

```
#include<iostream>
#include<conio.h>
#include<stdlib.h>
#define MAX_SIZE 5
usingnamespacestd;
voidmerge_sort(int, int);
voidmerge_array(int, int, int, int);
int arr_sort[MAX_SIZE];
intmain(){
int i;
cout<<"Simple C++ Merge Sort Example - Functions and Array\n";
cout<<"\nEnter "<< MAX_SIZE <<" Elements for Sorting : "<<endl;
for (i = 0; i < MAX_SIZE; i++)
cin>> arr_sort[i];
cout<<"\nYour Data   :";
for (i = 0; i < MAX_SIZE; i++) {
cout<<"\t"<< arr_sort[i];
```

```
   }
   merge_sort(0, MAX_SIZE - 1);
cout<<"\n\nSorted Data :";
for (i = 0; i < MAX_SIZE; i++) {
cout<<"\t"<< arr_sort[i];
   }
getch();
}
voidmerge_sort(int i, int j){
int m;
if (i < j) {
      m = (i + j) / 2;
      merge_sort(i, m);
      merge_sort(m + 1, j);
// Merging two arrays
      merge_array(i, m, m + 1, j);
   }
}
voidmerge_array(int a, int b, int c, int d){
int t[50];
int i = a, j = c, k = 0;
while (i <= b && j <= d) {
if (arr_sort[i] < arr_sort[j])
t[k++] = arr_sort[i++];
else
t[k++] = arr_sort[j++];
   }
//collect remaining elements
while (i <= b)
t[k++] = arr_sort[i++];
while (j <= d)
t[k++] = arr_sort[j++];
for (i = a, j = 0; i <= d; i++, j++)
      arr_sort[i] = t[j];
```

```
}
Output
Enter 5 Elements for Sorting
67
57
45
32
13
Your Data   :   6757453213
Sorted Data :1332455767
```

15. Write a C++ program to find optimal ordering of matrix multiplication.

Ans :

```
#include<stdio.h>
#include<limits.h>
// Matrix Ai has dimension p[i-1] x p[i] for i = 1..n
int MatrixChainMultiplication(int p[], int n)
{
    int m[n][n];
   int i, j, k, L, q;
   for (i=1; i<n; i++)
       m[i][i] = 0;    //number of multiplications are 0(zero) when there is only one matrix
   //Here L is chain length. It varies from length 2 to length n.
   for (L=2; L<n; L++)
   {
       for (i=1; i<n-L+1; i++)
       {
          j = i+L-1;
          m[i][j] = INT_MAX;  //assigning to maximum value
          for (k=i; k<=j-1; k++)
          {
               q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
             if (q < m[i][j])
             {
                 m[i][j] = q;    //if number of multiplications found less that number will be updated.
```

```
                }
            }
        }
    }
    return m[1][n-1];    //returning the final answer which is M[1][n]
}
int main()
{
    int n,i;
    printf("Enter number of matrices\n");
    scanf("%d",&n);
    n++;
    int arr[n];
    printf("Enter dimensions \n");
    for(i=0;i<n;i++)
    {
        printf("Enter d%d :: ",i);
        scanf("%d",&arr[i]);
    }
    int size = sizeof(arr)/sizeof(arr[0]);
    printf("Minimum number of multiplications is %d ", MatrixChainMultiplication(arr, size));
    return 0;
}
```

Output

```
Enter number of matrices
4
Enter dimensions
Enter d0 :: 10
Enter d1 :: 100
Enter d2 :: 20
Enter d3 :: 5
Enter d4 :: 80
Minimum number of multiplications is 19000
```

16. Write a C++ program that uses dynamic programming algorithm to solve the optimal binary search tree problem

Ans :

```
#include <iostream>
#include <climits>
using namespace std;
// Find optimal cost to construct a binary search tree from keys
// `i` to `j`, where each key `k` occurs `freq[k]` number of times
int findOptimalCost(int freq[], int i, int j, int level)
{
   // base case
   if (j < i) {
       return 0;
   }
    int optimalCost = INT_MAX;
    // consider each key as a root and recursively find an optimal solution
   for (int k = i; k <= j; k++)
   {
       // recursively find the optimal cost of the left subtree
       int leftOptimalCost = findOptimalCost(freq, i, k - 1, level + 1);
        // recursively find the optimal cost of the right subtree
       int rightOptimalCost = findOptimalCost(freq, k + 1, j, level + 1);
        // current node's cost is `freq[k]×level`
       int cost = freq[k] * level + leftOptimalCost + rightOptimalCost;
        // update the optimal cost
       optimalCost = min (optimalCost, cost);
   }
    // Return minimum value
   return optimalCost;
}
 int main()
{
   int freq[] = { 25, 10, 20 };
   int n = sizeof(freq) / sizeof(freq[0]);
    cout << "The optimal cost of constructing BST is "
```

```
        << findOptimalCost(freq, 0, n - 1, 1);
    return 0;
}
```

Output:

```
The optimal cost of constructing BST is 95
```

17. Write a C++ program to implement Hash Table

Ans :

```
#include <iostream>
#include<list>
using namespace std;
class hashclass
{
   int bucket; //no of buckets you want
   list<int>*hashtable;//container
   public:
   hashclass(int a)//constructor
    {
      bucket=a;
      hashtable=new list<int>[bucket];
    }
    void insert_element(int key)//function used to insert elements to the hashtable
    {
//to get the hash index of key
        int indexkey=hashFunction(key);
        hashtable[indexkey].push_back(key);
    }
    void delete_element(int key)//function used to delete elements from the hashtable
    {
        int indexkey=hashFunction(key);
        list<int>::iterator i=hashtable[indexkey].begin();
        for(;i!=hashtable[indexkey].end();i++)
        {
            if(*i==key)
                break;
        }
        if(i!=hashtable[indexkey].end())
        {
            hashtable[indexkey].erase(i);
        }
    }
    int hashFunction(int a)//function used to map values to key
    {
        return (a%bucket);
    }
    void display_table()//used to display hashtable values
    {

        for(int i=0;i<bucket;i++)
        {
            cout<<i;
            list<int>::iterator j=hashtable[i].begin();
        for(;j!=hashtable[i].end();j++)
            {
                cout<<"---->"<<*j;
            }
            cout<<endl;
        }
    }
    void search_element(int key)//used to search element
    {
        int a=0;//will be 1 if element exist otherwise 0
        int indexkey=hashFunction(key);
                list<int>::iterator i=hashtable[indexkey].begin();
        for(;i!=hashtable[indexkey].end();i++)
        {
            if(*i==key)
            {
                a=1;
                break;
```

```
          }
      }
          if(a==1)
          {
              cout<<"The element you wanted to search is present in the hashtable."<<endl;
          }
          else
          {
              cout<<"Element not present"<<endl;
          }
    }
    ~hashclass()//destructor
    {
        delete[]hashtable;
    }
};
int main()
{
int bucketn,ch,element;
cout<<"enter the no of buckets"<<endl;
cin>>bucketn;
hashclass hashelement(bucketn);
while(1)
{
    cout<<"1.insert element to the hashtable"<<endl;
    cout<<"2.search element from the hashtable"<<endl;
    cout<<"3.delete element from the hashtable"<<endl;
    cout<<"4.display elements of the hashtable"<<endl;
    cout<<"5.exit"<<endl;
    cout<<"enter your choice"<<endl;
    cin>>ch;
    switch(ch)
    {
        case 1:cout<<"enter the element"<<endl;
        cin>>element;
        hashelement.insert_element(element);
        break;
        case 2:cout<<"enter the element you want to search"<<endl;
        cin>>element;
        hashelement.search_element(element);
        break;
        case 3:cout<<"enter the element to be deleted"<<endl;
        cin>>element;
        hashelement.delete_element(element);
        break;
        case 4:hashelement.display_table();
        break;
        case 5:return 0;
        default:cout<<"enter a valid value"<<endl;
}
}
  return 0;
}
```

SAMPLE OUTPUT:

```
enter the no of buckets
4
1.insert element to the hashtable
2.search element from the hashtable
3.delete element from the hashtable
4.display elements of the hashtable
5.exit
enter your choice
1
enter the element
13
1.insert element to the hashtable
2.search element from the hashtable
3.delete element from the hashtable
4.display elements of the hashtable
5.exit
```

```
enter your choice
1
enter the element
16
1.insert element to the hashtable
2.search element from the hashtable
3.delete element from the hashtable
4.display elements of the hashtable
5.exit
```

18. Write C++ programs to perform the following on Heap

Ans :

a) Build Heap

```
#include <iostream>
using namespace std;
int n; // size of array
// To heapify a subtree rooted with node i which is
// an index in arr[]. N is size of heap
// we are using an array to create min heap
void heapify(int arr[], int i)
{
   int smallest = i; // The node which will be heapified
   int l = 2 * i + 1; // left child node
   int r = 2 * i + 2; // right child node
   // Check if left child is smaller than its parent
   if (l < n && arr[l] < arr[smallest])
      smallest = l;
   // Check if right child is smaller than smallest
   if (r < n && arr[r] < arr[smallest])
      smallest = r;
   // If smallest is not parent
   if (smallest != i) {
      swap(arr[i], arr[smallest]);
      // Recursively heapify the affected sub-tree
      heapify(arr, smallest);
```

```
    }
}
// Function to build a Max-Heap from the given array
void buildHeap(int arr[])
{
    // Perform level order traversal
    // from last non-leaf node and heapify each node
    for (int i = n; i >= 0; i--) {
        heapify(arr, i);
    }
}
// Driver Code
int main()
{
    // Sample array
    int arr[] = { 1, 5, 6, 8, 9, 7, 3};
    // calculating size of array
    n = sizeof(arr) / sizeof(arr[0]);
    cout<< "Array representation before buildHeap is: "<<endl;
    for (int i = 0; i < n; ++i)
        cout<< arr[i] << " ";
    cout<<endl;
    // call the buildheap method on the array
    buildHeap(arr);
  cout<< "Array representation after buildHeap is: "<<endl;
// print the heap
    for (int i = 0; i < n; ++i)
        cout<< arr[i] << " ";
        return 0;
}
```

Output

```
1.32s
Array representation before buildHeap is:
1 5 6 8 9 7 3
Array representation after buildHeap is:
1 5 3 8 9 7 6
```

b) Insertion

```
#include<iostream>
using namespace std;
/*Heapify using Bottom up approach*/
void heapify(int a[],int n,int i)
{
        int parent;
        parent = (i-1)/2; //for finding the parent of child
        if(parent>=0)     //Check until index < 0
        {
              if(a[parent]<a[i])
              {
                    swap(a[parent],a[i]);
                    heapify(a,n,parent); //recursive heapify function
              }
        }
}
/*Inserting the New Element into the Heap*/
void insert(int a[],int &n,int val)
{
        /*Increase the Size of the Array by 1*/
        n=n+1;
        /*Insert the new element at the end of the Array*/
        a[n-1]=val;
        /*Heapify function*/
        heapify(a,n,n-1);
}
/*Printing the Heap*/
void print(int a[],int n)
{
        cout<<"\nThe Array Representation of Heap is\n";
        for(int i=0;i<n;i++)
        {
              cout<<a[i]<<" ";
        }
}
/*Driver Function*/
int main()
{
        /*Initial Max Heap is */
        int a[100]={10,5,3,2,4};
        int n = 5;
        /*The Element to be insert is 15*/
        int val = 15;
        /*Printing the Array*/
        print(a,n);
        /*Insert Function*/
        insert(a,n,val); //here we are passing the 'n' value by reference
        /*Printing the Array*/
        print(a,n);
        return 0;
}
```

OUTPUT

```
The Array Representation of Heap is
10 5 3 2 4
The Array Representation of Heap is //After inserting new element 15 is
15 5 10 2 4 3
```

c) Deletion

```
#include<iostream>
using namespace std;
/*Heapify the heap using top down approach*/
void heapify(int a[],int n,int i)
{
        int parent = i;
//checking whether the parent index is valid in the array or not
        if(i<n)
        {
              //formula for finding left subtree of parent
```

```
            int left = 2*i+1;
            //formula for finding Right subtree of parent
            int right = 2*i+2;
            //checking whether the children index is valid in the array or not
            if(left<n || right<n)
            {
                  1//checking whether parent is less than children or not
                  if(a[parent]<a[left] || a[parent]<a[right])
                  {
//finding the largest children
                        if(a[left]<a[right])
                        {
//swaping the parent with largest children
                              swap(a[right],a[parent]);
                              heapify(a,n,right);  //Recursively heapify the heap
                        }
                        else
                        {
                              swap(a[left],a[parent]);
                              heapify(a,n,right);
                        }
                  }
            }
      }
}
// Function to delete the root from Heap
void DeleteElement(int a[],int &n)
{
      swap(a[0],a[n-1]); // swaping root and the last element
      n=n-1; // Decrease size of heap by 1
      heapify(a,n,0); //heapify the root node
}
/*Printing the Heap*/
void printArray(int a[],int n)
```

```
{
        for(int i=0;i<n;i++)
        {
              cout<<a[i]<<" ";
        }
}
/*Driver Function*/
int main()
{
        /*Initial Max Heap is */
        int a[] = {10,5,3,2,4};
        int n=5;
        cout<<"Before Deletion"<<endl;
        /*Printing the Array*/
        printArray(a,n);
        /*Deleting the root*/
        DeleteElement(a,n);
        cout<<"\nAfter Deletion"<<endl;
        /*Printing the Array*/
        printArray(a,n);
        return 0;
}
```

OUTPUT

```
Before Deletion
10 5 3 2 4
After Deletion
5 4 3 2
```

19. Write C++ programs to perform following operations on Skip List

Ans :

a) Insertion b) Deletion

```
#include <iostream>
#include <cstdlib>
#include <cmath>
#include <cstring>
```

```
#define MAX_LEVEL 6
constfloat P =0.5;
usingnamespace std;
/*
 * Skip Node Declaration
 */
struct snode
{
int value;
snode**forw;
snode(int level, int&value)
{
forw=new snode *[level +1];
memset(forw, 0, sizeof(snode*)*(level +1));
this->value = value;
}
    ~snode()
{
delete[] forw;
}
};
/*
 * Skip List Declaration
 */
struct skiplist
{
snode*header;
int value;
int level;
skiplist()
{
header=new snode(MAX_LEVEL, value);
level=0;
}
```

```
    ~skiplist()
{
delete header;
}
void display();
bool contains(int&);
void insert_element(int&);
void delete_element(int&);
};
/*
 * Main: Contains Menu
 */
int main()
{
skiplist ss;
int choice, n;
while(1)
{
cout<<endl<<"-----------------------"<<endl;
cout<<endl<<"Operations on Skip list"<<endl;
cout<<endl<<"-----------------------"<<endl;
cout<<"1.Insert Element"<<endl;
cout<<"2.Delete Element"<<endl;
cout<<"3.Search Element"<<endl;
cout<<"4.Display List "<<endl;
cout<<"5.Exit "<<endl;
cout<<"Enter your choice : ";
cin>>choice;
switch(choice)
{
case1:
cout<<"Enter the element to be inserted: ";
cin>>n;
        ss.insert_element(n);
if(ss.contains(n))
cout<<"Element Inserted"<<endl;
break;
case2:
cout<<"Enter the element to be deleted: ";
cin>>n;
if(!ss.contains(n))
{
cout<<"Element not found"<<endl;
break;
}
        ss.delete_element(n);
if(!ss.contains(n))
cout<<"Element Deleted"<<endl;
break;
case3:
cout<<"Enter the element to be searched: ";
cin>>n;
if(ss.contains(n))
cout<<"Element "<<n<<" is in the list"<<endl;
else
cout<<"Element not found"<<endl;
case4:
cout<<"The List is: ";
ss.display();
break;
case5:
exit(1);
break;
default:
cout<<"Wrong Choice"<<endl;
}
}
return0;
```

```
}
/*
 * Random Value Generator
 */
float frand()
{
return(float)rand()/RAND_MAX;
}
/*
 * Random Level Generator
 */
int random_level()
{
staticbool first =true;
if(first)
{
srand((unsigned)time(NULL));
first=false;
}
int lvl =(int)(log(frand())/log(1.-P));
return lvl < MAX_LEVEL ?lvl : MAX_LEVEL;
}
/*
 * Insert Element in Skip List
 */
void skiplist::insert_element(int&value)
{
snode*x = header;
snode*update[MAX_LEVEL +1];
memset(update, 0, sizeof(snode*)*(MAX_LEVEL +1));
for(int i = level;i >=0;i--)
{
while(x->forw[i]!=NULL&& x->forw[i]->value < value)
{
```

```
            x = x->forw[i];
}
update[i]= x;
}
    x = x->forw[0];
if(x ==NULL|| x->value != value)
{
int lvl = random_level();
if(lvl > level)
{
for(int i = level +1;i <= lvl;i++)
{
update[i]= header;
}
level= lvl;
}
        x =newsnode(lvl, value);
for(int i =0;i <= lvl;i++)
{
x->forw[i]= update[i]->forw[i];
update[i]->forw[i]= x;
}
}
}
/*
 * Delete Element from Skip List
 */
void skiplist::delete_element(int&value)
{
snode*x = header;
snode*update[MAX_LEVEL +1];
memset(update, 0, sizeof(snode*)*(MAX_LEVEL +1));
for(int i = level;i >=0;i--)
{
```

```
while(x->forw[i]!=NULL&& x->forw[i]->value < value)
{
            x = x->forw[i];
}
update[i]= x;
}
    x = x->forw[0];
if(x->value == value)
{
for(int i =0;i <= level;i++)
{
if(update[i]->forw[i]!= x)
break;
update[i]->forw[i]= x->forw[i];
}
delete x;
while(level >0&& header->forw[level]==NULL)
{
level--;
}
}
}
/*
 * Display Elements of Skip List
 */
void skiplist::display()
{
const snode *x = header->forw[0];
while(x !=NULL)
{
cout<< x->value;
        x = x->forw[0];
if(x !=NULL)
cout<<" - ";
```

```
}
cout<<endl;
}
/*
 * Search Elemets in Skip List
 */
bool skiplist::contains(int&s_value)
{
snode*x = header;
for(int i = level;i >=0;i--)
{
while(x->forw[i]!=NULL&& x->forw[i]->value < s_value)
{
            x = x->forw[i];
}
}
    x = x->forw[0];
return x !=NULL&& x->value == s_value;
}
```

```
OUTPUT
---------------------------------
Operations on Skip list
---------------------------------
1.Insert Element
2.Delete Element
3.Search Element
4.Display List
5.Exit
Enter your choice :1
Enter the element to be inserted: 7
Element Inserted
---------------------------------
Operations on Skip list
---------------------------------
```

```
1.Insert Element
2.Delete Element
3.Search Element
4.Display List
5.Exit
Enter your choice :1
Enter the element to be inserted: 9
Element Inserted
---------------------------------
Operations on Skip list
---------------------------------
1.Insert Element
2.Delete Element
3.Search Element
4.Display List
5.Exit
Enter your choice :4
The List is: 7 - 9
```

20. Write a C++ Program to Create a Graph using Adjacency Matrix Representation.

Ans :

```
#include <iostream>
using namespace std;
// stores adjacency list items
struct adjNode {
    int val, cost;
    adjNode* next;
};
// structure to store edges
struct graphEdge {
    int start_ver, end_ver, weight;
};
class DiaGraph{
    // insert new nodes into adjacency list from given graph
    adjNode* getAdjListNode(int value, int weight, adjNode* head)   {
        adjNode* newNode = new adjNode;
        newNode->val = value;
        newNode->cost = weight;

        newNode->next = head;   // point new node to current head
        return newNode;
    }
    int N;  // number of nodes in the graph
public:
    adjNode **head;                //adjacency list as array of pointers
    // Constructor
    DiaGraph(graphEdge edges[], int n, int N)  {
        // allocate new node
        head = new adjNode*[N]();
        this->N = N;
        // initialize head pointer for all vertices
        for (int i = 0; i < N; ++i)
            head[i] = nullptr;
        // construct directed graph by adding edges to it
        for (unsigned i = 0; i < n; i++)  {
            int start_ver = edges[i].start_ver;
            int end_ver = edges[i].end_ver;
            int weight = edges[i].weight;
            // insert in the beginning
            adjNode* newNode = getAdjListNode(end_ver, weight, head[start_ver]);
                    // point head pointer to new node
            head[start_ver] = newNode;
            }
    }
     // Destructor
     ~DiaGraph() {
    for (int i = 0; i < N; i++)
        delete[] head[i];
        delete[] head;
    }
};
// print all adjacent vertices of given vertex
void display_AdjList(adjNode* ptr, int i)
{
   while (ptr != nullptr) {
      cout << "(" << i << ", " << ptr->val
         << ", " << ptr->cost << ") ";
      ptr = ptr->next;
   }
   cout << endl;
}
// graph implementation
int main()
{
   // graph edges array.
   graphEdge edges[] = {
      // (x, y, w) -> edge from x to y with weight w
       {0,1,2},{0,2,4},{1,4,3},{2,3,2},{3,1,4},{4,3,3}
   };
   int N = 6;     // Number of vertices in the graph
   // calculate number of edges
   int n = sizeof(edges)/sizeof(edges[0]);
   // construct graph
   DiaGraph diagraph(edges, n, N);
   // print adjacency list representation of graph
   cout<<"Graph adjacency list "<<endl<<"(start_vertex, end_vertex, weight):"<<endl;
   for (int i = 0; i < N; i++)
   {
      // display adjacent vertices of vertex i
      display_AdjList(diagraph.head[i], i);
   }
   return 0;
}
```

```
Output:
Graph adjacency list
(start_vertex, end_vertex, weight):
(0, 2, 4) (0, 1, 2)
(1, 4, 3)
(2, 3, 2)
(3, 1, 4)
(4, 3, 3)
```

21. Write a C++ program to implement graph traversal techniques

Ans :

a) BFS

```
#include<iostream>
#include<conio.h>
#include<stdlib.h>
int cost[10][10],i,j,k,n,qu[10],front,rare,v,visit[10],visited[10];
int main()
{
   int m;
   cout <<"Enter no of vertices:";
   cin >> n;
   cout <<"Enter no of edges:";
   cin >> m;
   cout <<"\nEDGES \n";
   for(k=1; k<=m; k++)
   {
      cin >>i>>j;
      cost[i][j]=1;
   }
   cout <<"Enter initial vertex to traverse from:";
   cin >>v;
   cout <<"Visitied vertices:";
   cout <<v<<" ";
   visited[v]=1;
   k=1;
   while(k<n)
   {
      for(j=1; j<=n; j++)
         if(cost[v][j]!=0 && visited[j]!=1 && visit[j]!=1)
         {
            visit[j]=1;
            qu[rare++]=j;
         }
      v=qu[front++];
      cout<<v <<" ";
      k++;
      visit[v]=0;
      visited[v]=1;
   }
   return 0;
}
```

OUTPUT

```
Enter no of vertices:4
Enter no of edges:4
EDGES
1 2
1 3
2 4
3 4
Enter initial vertex to traverse from:1
Visited vertices:1 2 3 4
12
```

b) DFS

```
#include<iostream>
#include<conio.h>
#include<stdlib.h>
int cost[10][10],i,j,k,n,stk[10],top,v,visit[10],visited[10];
int main()
{
  int m;
  cout <<"Enter no of vertices:";
  cin >> n;
  cout <<"Enter no of edges:";
  cin >> m;
  cout <<"\nEDGES \n";
  for(k=1; k<=m; k++)
  {
     cin >>i>>j;
     cost[i][j]=1;
  }
  cout <<"Enter initial vertex to traverse from:";
  cin >>v;
  cout <<"DFS ORDER OF VISITED VERTICES:";
  cout << v <<" ";
  visited[v]=1;
  k=1;
  while(k<n)
  {
      for(j=n; j>=1; j--)
         if(cost[v][j]!=0 && visited[j]!=1 && visit[j]!=1)
         {
            visit[j]=1;
            stk[top]=j;
            top++;
         }
      v=stk[--top];
      cout<<v << " ";
      k++;
      visit[v]=0;
      visited[v]=1;
  }
  return 0;
}
```

```
OUTPUT
Enter no of vertices:4
Enter no of edges:4
EDGES
1 2
1 3
2 4
3 4
Enter initial vertex to traverse from:1
DFS ORDER OF VISITED VERTICES:1 2 4 3
```

22. Write a C++ program to Heap sort using tree structure.

Ans :

```
#include <iostream>
using namespace std;
// function to heapify the tree
void heapify(int arr[], int n, int root)
{
  int largest = root; // root is the largest element
  int l = 2*root + 1; // left = 2*root + 1
  int r = 2*root + 2; // right = 2*root + 2
  // If left child is larger than root
  if (l < n && arr[l] > arr[largest])
  largest = l;
  // If right child is larger than largest so far
  if (r < n && arr[r] > arr[largest])
  largest = r;
  // If largest is not root
  if (largest != root)
    {
    //swap root and largest
     swap(arr[root], arr[largest]);
    // Recursively heapify the sub-tree
    heapify(arr, n, largest);
    }
}
// implementing heap sort
void heapSort(int arr[], int n)
```

```
{
  // build heap
  for (int i = n / 2 - 1; i >= 0; i--)
  heapify(arr, n, i);
  // extracting elements from heap one by one
  for (int i=n-1; i>=0; i--)
  {
     // Move current root to end
     swap(arr[0], arr[i]);
    // again call max heapify on the reduced heap
    heapify(arr, i, 0);
  }
}
/* print contents of array - utility function */
void displayArray(int arr[], int n)
{
  for (int i=0; i<n; ++i)
  cout << arr[i] << " ";
  cout << "\n";
}
// main program
int main()
{
   int heap_arr[] = {4,17,3,12,9,6};
   int n = sizeof(heap_arr)/sizeof(heap_arr[0]);
   cout<<"Input array"<<endl;
  displayArray(heap_arr,n);
   heapSort(heap_arr, n);
   cout << "Sorted array"<<endl;
  displayArray(heap_arr, n);
}
```

Output:

```
Input array
4 17 3 12 9 6
Sorted array
3 4 6 9 12 17
```

FACULTY OF SCIENCE

B.Sc. II Year III - Semester (CBCS) Examination

Model Paper - I

DATA STRUCTURES USING C++

PAPER - III : (COMPUTER SCIENCE)

Time : 3 Hours] [Max. Marks : 80

PART - A (8 × 4 = 32 Marks)
(Short Answer Type)

Note : *Answer any* ***Eight*** *of the following questions.*

1.	What is Algorithm? What are the characteristics of an algorithm ?	**(Unit-I, SQA-3)**
2.	What are the differences between Pseudocode and Algorithms.	**(Unit-I, SQA-6)**
3.	What is multiple stack.	**(Unit-I, SQA-14)**
4.	What are the differences between Iteration and recursion.	**(Unit-II, SQA-7)**
5.	List out the basic operation of a Queue.	**(Unit-II, SQA-9)**
6.	List out the applications of Queues.	**(Unit-II, SQA-12)**
7.	Explain ADT of Binary Tree	**(Unit-III, SQA-3)**
8.	Apply the bubble sort algorithm to sort the list of data: 2, 10, 6, 4, 8.	**(Unit-III, SQA-6)**
9.	What are the pros of binary search.	**(Unit-III, SQA-9)**
10.	Minimum spanning trees.	**(Unit-IV, SQA-3)**
11.	Adjacency Matrix.	**(Unit-IV, SQA-1)**
12.	Min Heap	**(Unit-IV, SQA-6)**

PART - B (4 × 12 = 48 Marks)
(Essay Answer Type)

Note : *Answer* **ALL** *the questions.*

13. (a) What is a Stack? List out the standard operations of Stack ? **(Unit-I, Q.No. 28)**

OR

(b) Explain the process of converting infix to prefix notation. **(Unit-I, Q.No. 45)**

14. (a) Explain various types of recursions. **(Unit-II, Q.No. 14)**

OR

(b) What is mean by Queue? What are the basic operations of Queue? **(Unit-II, Q.No. 19)**

15. (a) Explain about various traversal techniques of binary tree. **(Unit-III, Q.No. 10)**

OR

(b) What is insertion sort? Explain its working with an example. **(Unit-III, Q.No. 29)**

16. (a) What are the various ways to represent graphs? **(Unit-IV, Q.No. 2)**

OR

(b) Explain Kruskal's Algorithm with an example. **(Unit-IV, Q.No. 12)**

FACULTY OF SCIENCE

B.Sc. II Year III - Semester (CBCS) Examination

Model Paper - II

DATA STRUCTURES USING C++

PAPER - III : (COMPUTER SCIENCE)

Time : 3 Hours] [Max. Marks : 80

PART - A (8 × 4 = 32 Marks)

(Short Answer Type)

Note : *Answer any* ***Eight*** *of the following questions.*

1. List out the basic characteristics of an algorithm. **(Unit-I, SQA-4)**
2. What are the differences between algorithm and flow chart. **(Unit-I, SQA-8)**
3. Lest out the applications of a stack. **(Unit-I, SQA-15)**
4. List out the advantages and disadvantages of recursion ? **(Unit-II, SQA-2)**
5. Write a c++ program to reverse a number using recursion. **(Unit-II, SQA-4)**
6. What is recursive function? **(Unit-II, SQA-6)**
7. What is binary tree? **(Unit-III, SQA-1)**
8. Explain ADT of Binary Tree **(Unit-III, SQA-3)**
9. Compare various sorting techniques with real world usage. **(Unit-III, SQA-5)**
10. What is Heap? **(Unit-IV, SQA-4)**
11. Representation of Graphs. **(Unit-IV, SQA-9)**
12. What is collision resolution technique? **(Unit-IV, SQA-10)**

PART - B (4 × 12 = 48 Marks)

(Essay Answer Type)

Note : *Answer* **ALL** *the questions.*

13. (a) What is the use of function calls in stack implementation? **(Unit-I, Q.No. 51)**

OR

(b) Explain linked list representation for implementing Stack. **(Unit-I, Q.No. 32)**

14. (a) Explain the representation of Queue using Linked List. **(Unit-II, Q.No. 29)**

OR

(b) Discuss in detail about Doubly Linked List. Explain the basic operations of Doubly linked list. **(Unit-II, Q.No. 55)**

15. (a) Compare various sorting techniques with real world usage. **(Unit-III, Q.No. 34)**

OR

(b) How to insert a node in binary tree? Explain with an example. **(Unit-III, Q.No. 8)**

16. (a) Explain about BFS algorithm with an example. **(Unit-IV, Q.No. 5)**

OR

(b) What is heap sort? Write the heap sort algorithm? Explain with an example. **(Unit-IV, Q.No. 28)**

FACULTY OF SCIENCE

B.Sc. II Year III - Semester (CBCS) Examination

Model Paper - III

DATA STRUCTURES USING C++

PAPER - III : (COMPUTER SCIENCE)

Time : 3 Hours] [Max. Marks : 80

PART - A (8 × 4 = 32 Marks)

(Short Answer Type)

Note : *Answer any* ***Eight*** *of the following questions.*

1. Draw a flow chart to calculate the average of two numbers. **(Unit-I, SQA-7)**
2. What is recursion ? **(Unit-I, SQA-21)**
3. What are the advantages of prefix and postfix expressions? **(Unit-I, SQA-26)**
4. What are the differences between stack and Queue. **(Unit-II, SQA-13)**
5. What are the advantges of a linked list. **(Unit-II, SQA-16)**
6. Why linked list is called dynamic data structure? What are the advantages of using linked list over array. **(Unit-II, SQA-20)**
7. Write a C + + program for creation and traversal of a Binary Tree **(Unit-III, SQA-2)**
8. What is Search ? **(Unit-III, SQA-4)**
9. What are the applications of trees? **(Unit-III, SQA-10)**
10. What is graph? **(Unit-IV, SQA-8)**
11. Representation of Graphs. **(Unit-IV, SQA-9)**
12. What is Hashing? Define it with an example. **(Unit-IV, SQA-12)**

PART - B (4 × 12 = 48 Marks)

(Essay Answer Type)

Note : *Answer* **ALL** *the questions.*

13. (a) What are the differences between Pseudocode and Algorithms. **(Unit-I, Q.No. 17)**

OR

(b) Discuss in detail Stack ADT. **(Unit-I, Q.No. 30)**

14. (a) What is mean by Circular Queue? What is the need of circular queue? **(Unit-II, Q.No. 30)**

OR

(b) Write a C + + program to implement queues using arrays. **(Unit-II, Q.No. 28)**

15. (a) Explain the operations that can be performed on BSTs. **(Unit-III, Q.No. 14)**

OR

(b) Explain Binary Search with an example. **(Unit-III, Q.No. 23)**

16. (a) What is Closed Hashing? Explain Linear probing with example. **(Unit-IV, Q.No. 20)**

OR

(b) What are the different types of heaps? Explain Max heap algorithm with an example. **(Unit-IV, Q.No. 25)**

FACULTY OF SCIENCE

B.sc. II Year III-Semester(CBSC) Examination

July - 2021

DATA STRUCTURES

Paper - III : (Computer Science)

Time : 2 Hours [Max. Marks : 80

PART - A - (5 × 4 = 20 Marks)

[Short Answer type]

Note : Answer any **FIVE** questions.

ANSWERS

1.	Explain the attributes of ordered list with examples.	
2.	What is a stack? Write various applications of stack.	**(Unit-I, Q.No.34)**
3.	Demonstrate recursive call with an example.	**(Unit-II, SQA-21)**
4.	Write all the applications of linked list.	**(Unit-II, SQA-45)**
5.	What is binary tree abstract data type?	**(Unit-III, SQA-3)**
6.	Explain hashing with an example.	**(Unit-IV, SQA-12)**
7.	Describe searching process using an array.	**(Unit-III, SQA-4)**
8.	Write the comparisons of sorting techniques.	**(Unit-III, SQA-5)**
9.	Define Flowchart. Discuss the different symbols in Flowchart.	**(Unit-I, Q.No.18)**
10.	Write short notes on Double-ended queue.	**(Unit-I, Q.No.34)**
11.	Write short notes on Collision Resolution strategies.	**(Unit-IV, SQA-10)**
12.	What is a Heap Abstract Data Type?	**(Unit-IV, SQA-11)**

PART - B - (3 × 20 = 60 Marks)

[Essay Answer Type]

Note : Answer any **THREE** questions.

13.	Write string manipulation instructions. Demonstrate those instructions with examples and algorithms.	**(Unit-I, Q.No.52,53,54)**
14.	Describe arithmetic expression evaluation with examples.	**(Unit-I, Q.No.35)**
15.	Demonstrate and explain circular queue with examples and algorithm.	**(Unit-II, Q.No.30,31,32)**
16.	What is recursion and write an example for recursion?	**(Unit-II, Q.No.1)**
17.	Explain binary tree representation with traversals.	**(Unit-III, Q.No.10)**
18.	Describe Kruskal's algorithm.	**(Unit-IV, Q.No.12)**
19.	Explain Binary search algorithm with an example.	**(Unit-III, Q.No.13)**
20.	What is merge sort? Explain with an example and an algorithm.	**(Unit-III, Q.No.31)**

FACULTY OF SCIENCE

B.sc. II Year III-Semester(CBSC) Examination

November / December - 2019

DATA STRUCTURES

Paper - III : (Computer Science)

Time : 3 Hours [Max. Marks : 80

PART - A - (5 × 4 = 20 Marks)

[Short Answer type]

Note : Answer any **FIVE** of the following questions.

ANSWERS

1. Define data structure and describe the types of data structure. **(Unit-I, SQA-1, Q.No.5)**
2. What is stack? List out applications of stack. **(Unit-I, Q.No.34)**
3. Why linked list is called dynamic data structure? What are the advantages of using linked list over arrays? **(Unit-II, SQA-20)**
4. Describe execution of recursive calls with example. **(Unit-II, SQA-21)**
5. What are the binary tree application? **(Unit-III, SQA-10)**
6. Define graph and explain graph representation. **(Unit-IV, SQA-8,9)**
7. Write a program for sequential search. **(Out of Syllabus)**
8. What is heap? Explain heap construction process. **(Unit-IV, SQA-3)**

PART - B - (4 × 15 = 60 Marks)

[Essay Answer Type]

Note : Answer **ALL** the questions from the following.

9. (a) (i) What is an array and explain its advantages and disadvantages? **(Out of Syllabus)**

(ii) Explain memory representation and address calculation of 1-D and 2-D arrays. **(Out of Syllabus)**

OR

(b) Write a program to implement the stack abstract data type using an array. **(Unit-I, SQA-30)**

10. (a) (i) What is recursion and write an example for recursion? **(Unit-II, Q.No.1)**

(ii) What is queue? And explain about circular queue and double ended queue. **(Unit-II, Q.No.19,30,34)**

OR

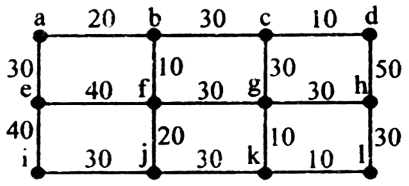
(b) Write a program to create a double linked list insert, delete and search for an element operations. **(Unit-II, Q.No.55,57)**

11. (a) (i) Define the binary tree and explain its properties. Explain the binary tree traversal techniques with example. **(Unit-III, Q.No.3,10)**

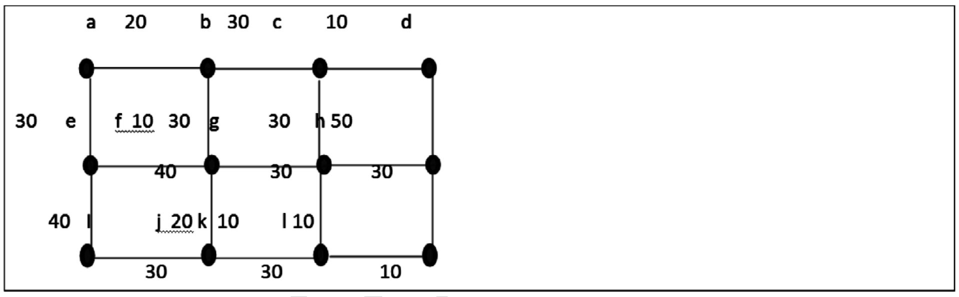
(ii) Write a program to travel binary tree in pre-order, post-order and in-order. **(Unit-III, Q.No.11)**

OR

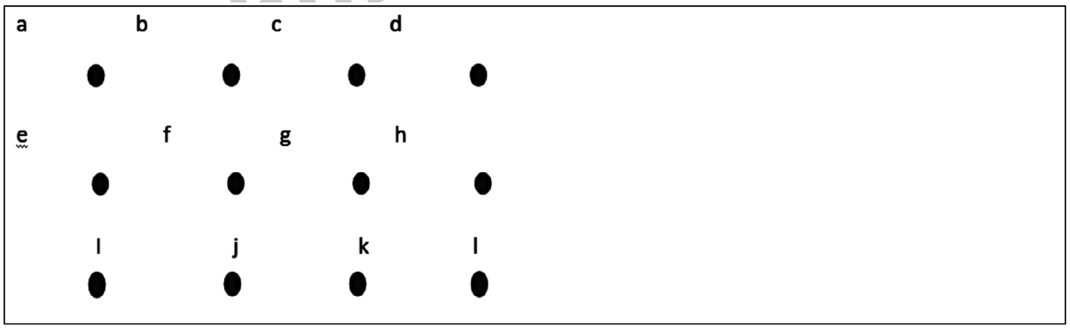
(b) What is spanning tree and minimum spanning tree? Construct minimum spanning tree using Kruskal's algorithm. **(Unit-IV, Q.No.7,8)**

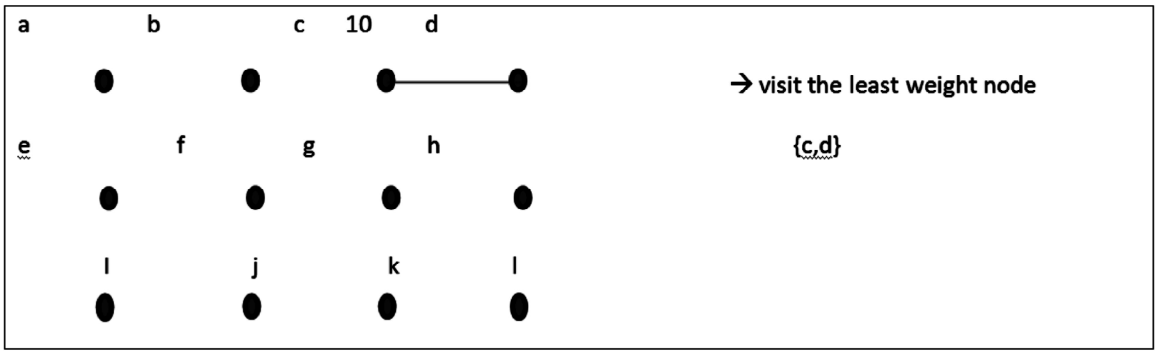


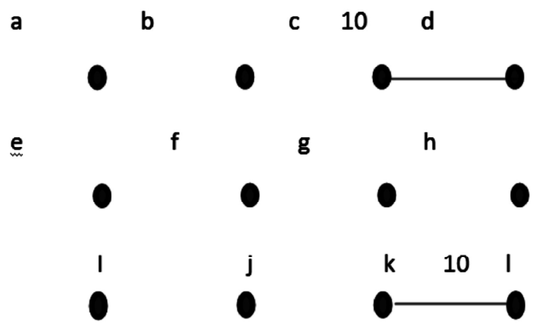
Ans :



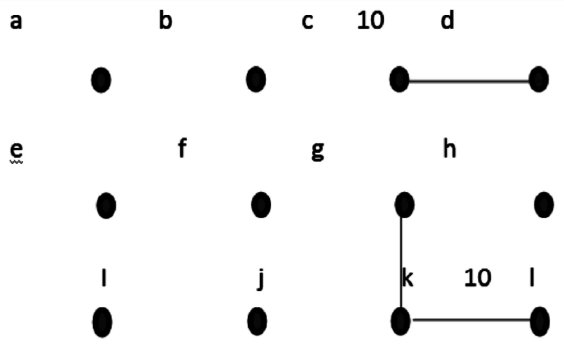
Steps: Mark all the nodes. Visit the least weight edges until all the nodes are visited with out loops



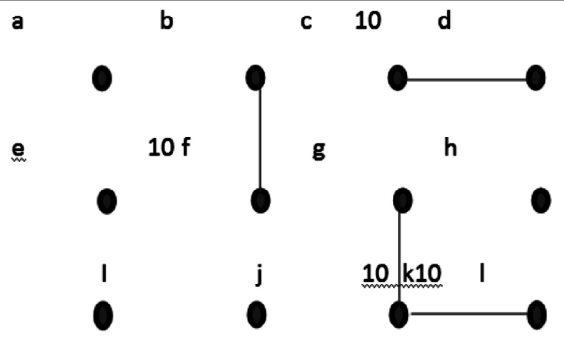




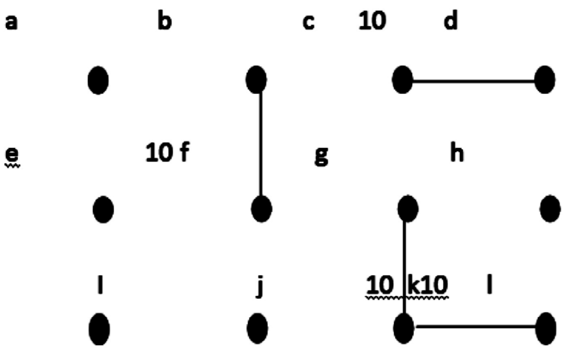
→{c,d} {k,l}



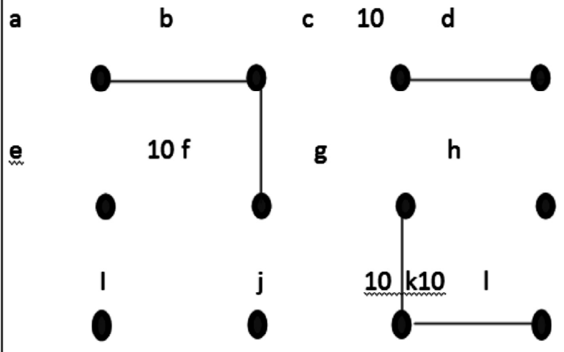
→{c,d} {k,l}{g,k}



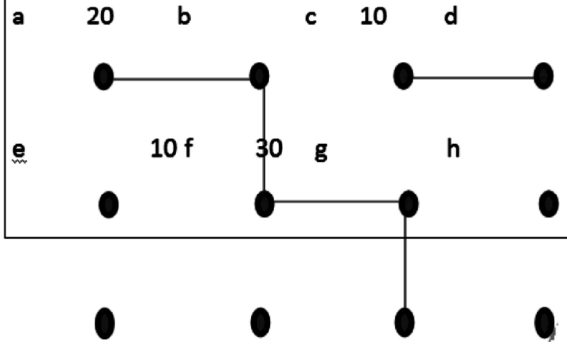
→{c,d} {k,l} {g,k}, {b,f}



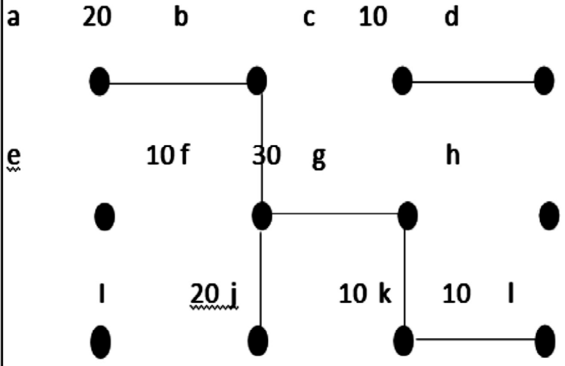
→{c,d} {k,l} {g,k}, {b,f}



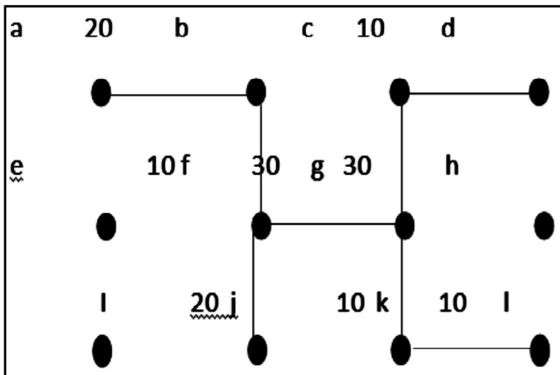
→{c,d} {k,l} {g,k}, {b,f}, {a,b}



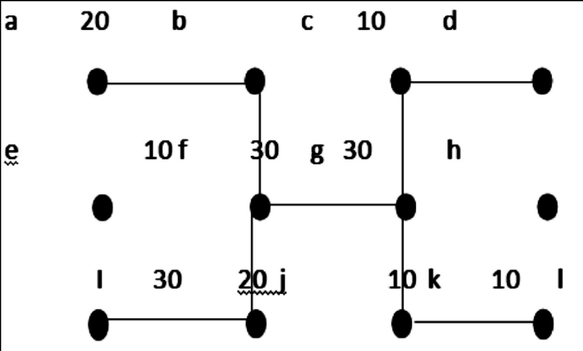
→{c,d} {k,l} {g,k}, {b,f}, {a,b}



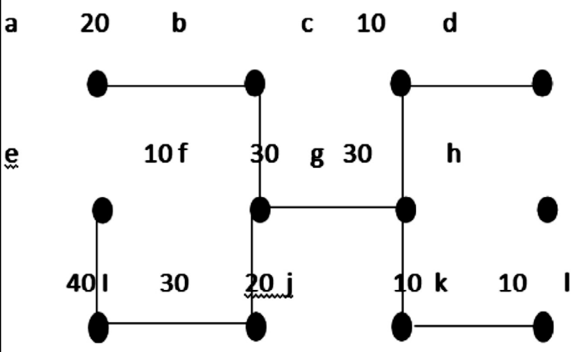
→{c,d} {k,l} {g,k}, {b,f}, {a,b} , {f,j}



→{c,d} {k,l} {g,k}, {b,f}, {a,b}{f,j}{c,g}



$\rightarrow$ {c,d} {k,l} {g,k}, {b,f}, {a,b}{f,j}{c,g}{I,j}



$\rightarrow$ {c,d} {k,l} {g,k}, {b,f}, {a,b}{f,j}{c,g}{I,j}{e,f}

Now the weight of the graph is

$10+10+10+30+30+20+10+20+30+40 = 210$

12. (a) (i) Sort the given list of numbers 76, 67, 36, 55, 23, 14, 6 using

(ii) Write a program for sorting the numbers in ascending order using bubble sort. **(Unit-III, Q.No.28, Refer 12(a) Dec.-17)**

OR

(b) Explain the merge sort technique and write a program for sorting the numbers in ascending order using merge sort. **(Unit-III, Q.No.31)**

FACULTY OF SCIENCE

B.sc. II Year III-Semester(CBSC) Examination

June / July - 2019

DATA STRUCTURES

Paper - III : (Computer Science)

Time : 3 Hours [Max. Marks : 80

PART - A - (5 × 4 = 20 Marks)

[Short Answer type]

Note : Answer any **FIVE** of the following questions.

		ANSWERS
1.	Define a Data structure in terms of the triplet (D, F, A). Give an example.	**(Unit-I, SQA-25)**
2.	What are the advantages of prefix and postfix expression?	**(Unit-I, SQA-26)**
3.	What are the applications of stacks?	**(Unit-I, Q.No.34)**
4.	Write a recursive algorithm to find X^Y.	**(Unit-II, SQA-19)**
5.	What are the applications of trees?	**(Out of Syllabus)**
6.	Write the algorithm for post order traversal on a binary tree and give an example.	**(Unit-III, SQA-2)**
7.	Define internal and external sorting.	**(Unit-III, SQA-8)**
8.	Define a heap with an example.	**(Unit-IV, SQA-3)**

PART - B - (4 × 15 = 60 Marks)

[Essay Answer Type]

Note : Answer **ALL** the questions.

9. (a) Explain the concept of subalgorithms. Write an algorithm to compute the following using a subalgorithm: P = n! / (n – r)!

Ans :

```
Start
Step 1 -> Declare function for calculating factorial
    intcal_n(int n)
    int temp = 1
    Loop for inti = 2 and i<= n and i++
        Set temp = temp * i
    End
    return temp
```

```
step 2 -> declare function to calculate ncr
    intnCr(int n, int r)
        returncal_n(n) / (cal_n(r) * cal_n(n - r))
step 3 -> In main()
    declare variable as int n = 12, r = 4
    printnCr(n, r)
Stop
```

OR

(b) Explain how a stack is used in the following applications,

(i) Processing of function calls **(Unit-I, Q.No.51)**

(ii) Reversing a string **(Unit-I, Q.No.52,53)**

(iii) Checking correctness of well-formed parentheses. **(Out of Syllabus)**

10. (a) Explain Tower of Hanoi using recursion. Write the algorithm and draw the tree representation of recursive calls of the algorithm for 3 disks. **(Unit-II, Q.No.16)**

OR

(b) Explain tire representation of Queue using a linked list. **(Unit-II, Q.No.29)**

11. (a) Define Inorder traversal algorithm with an example. Write a C ++ member function to implement non-recursive inorder traversal algorithm of a binary tree. **(Unit-III, Q.No.10,11)**

OR

(b) Write Graph ADT. Explain the following operations on a graph with examples.

(i) Insert Vertex

(ii) Delete Vertex

(iii) Insert Edge

(iv) Delete Edge **(Out of Syllabus)**

12. (a) Write an algorithm for Quick sort. Explain the step-by-step procedure of quick sort on the list 25, 57, 48, 37, 12, 92, 86, 33 **(Unit-III, Q.No.32, Refer 12(b) Dec.-17)**

OR

(b) Discuss Binary Search algorithm with an example. Write down the Pros and Cons of binary search method. **(Unit-III, Q.No.23)**

FACULTY OF SCIENCE

B.sc. II Year III-Semester(CBSC) Examination

November / December - 2018

DATA STRUCTURES

Paper - III : (Computer Science)

Time : 3 Hours [Max. Marks : 80

PART - A - (5 × 4 = 20 Marks)

[Short Answer type]

Note : Answer any **FIVE** of the following questions.

ANSWERS

1. What is the difference between Atomic and Composite data with examples? **(Unit-I, SQA-23)**
2. What is Sequential Organization? Briefly explain its advantages and disadvantages. **(Unit-I, SQA-24)**
3. Explain the use of Stack to find the factorial of a number using recursion. **(Unit-II, SQA-1)**
4. Define recursion? **(Unit-II, SQA-5)**
5. Construct a binary tree using the following two traversals.

Inorder	D	B	H	E	A	I	F	J	C	G
Preorder	A	B	D	E	H	C	F	I	J	G

(Unit-III, SQA-7)

6. Adjacency Matrix. **(Unit-IV, SQA-1)**
7. What are the Pros of binary search? **(Unit-III, SQA-9)**
8. Define minheap and maxheap with examples. **(Unit-IV, SQA-6,7)**

PART - B - (4 × 15 = 60 Marks)

[Essay Answer Type]

Note : Answer **ALL** the questions.

9. (a) Define ADT for the “Integer” and explain all its functions and axioms in detail. **(Out of Syllabus)**

OR

(b) Write a C++ program to implement the following operations on Arrays.

(i) To insert an element at a given position **(Out of Syllabus)**

(ii) To delete an element at a given position. **(Out of Syllabus)**

10. (a) Explain the representation of stack using a linked list. **(Unit-II, Q.No.64,65)**

OR

(b) Explain the operations of inserting a node, deleting a node and traversal in a circular linked list with examples. **(Unit-II, Q.No.59,60)**

11. (a) Write the algorithm for the construction of Expression Tree. Explain the steps to construct an expression tree for the expression E = (a + b × c)/d. **(Out of Syllabus)**

OR

(b) Explain the representation of graphs using the adjacency matrix, adjacency list and adjacency multi-list. **(Unit-IV, Q.No.2)**

12. (a) Write a C++ program for insertion sort. Show the steps of the insertion algorithm for the list of data. 76, 67, 36, 55, 23, 14, 6.

(Unit-IV, Q.No.29, Refer 12(a) Dec.-17)

OR

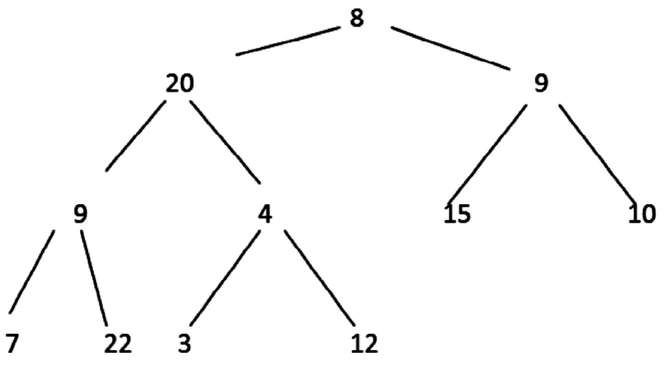
(b) Explain the step-by-step procedure to construct a heap tree using the list of keys. 8, 20, 9, 4, 15, 10, 7, 22, 3, 12. **(Unit-IV, Q.No.28)**

Ans :

8, 20, 9, 4, 15, 10, 7, 22, 3, 12

We can heapify this by using max heap or min heap. Here we use Max heap.

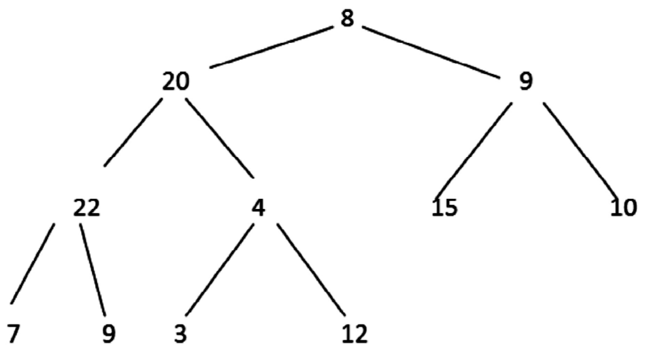
Step 1: construct a complete binary tree using the given elements



Step 2: We ensure that the tree is a max heap.

Node 22 contains greater element in its right child node.

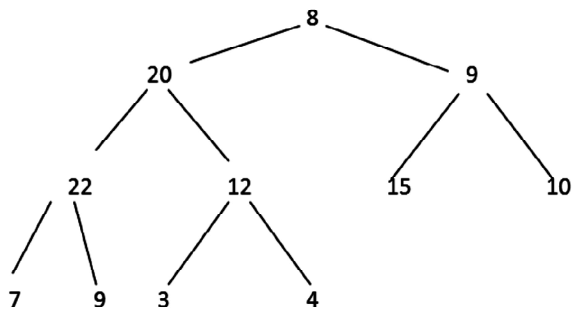
So, we swap node22 and node 9



Step 3:

Node 12 contains greater element in its right child node.

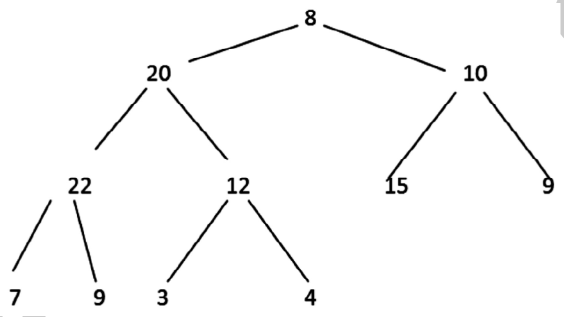
So, we swap node 12 and node 4



Step 4:

Node 10 contains greater element in its right child node.

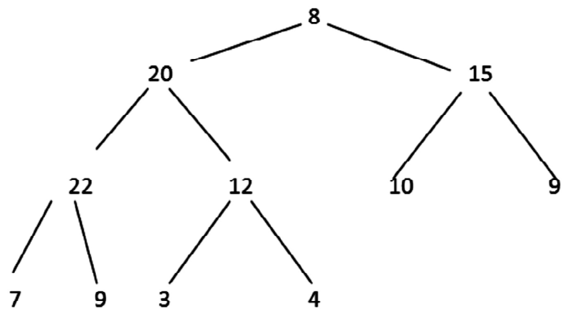
So, we swap node 10 and node 9



Step 5:

Node 10 contains greater element in its left child node.

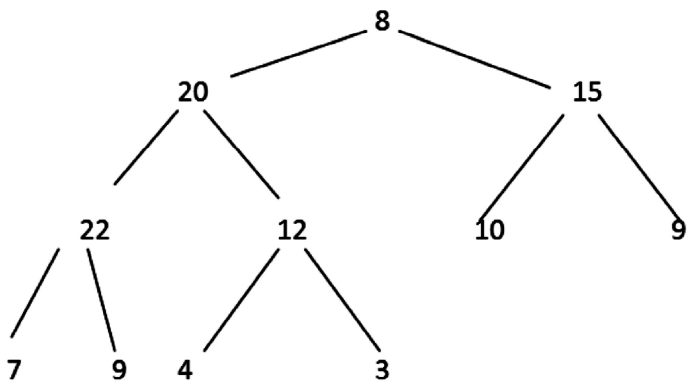
So, we swap node 10 and node 15



Step 6:

Node 4 containssmaller element in its left child node.

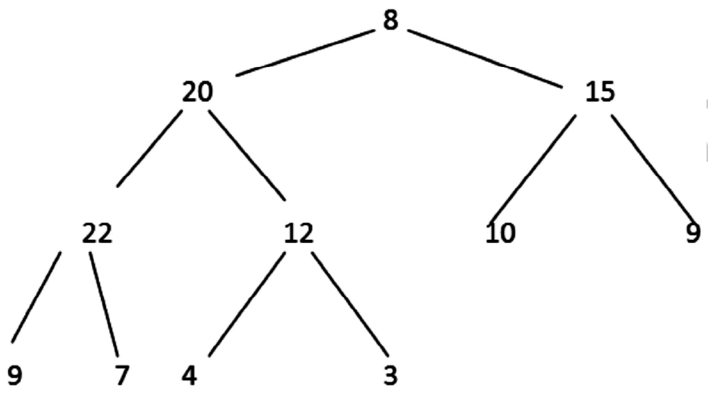
So, we swap node 4 and node 3



Step 7:

Node 9 contains lesser element in its left child node.

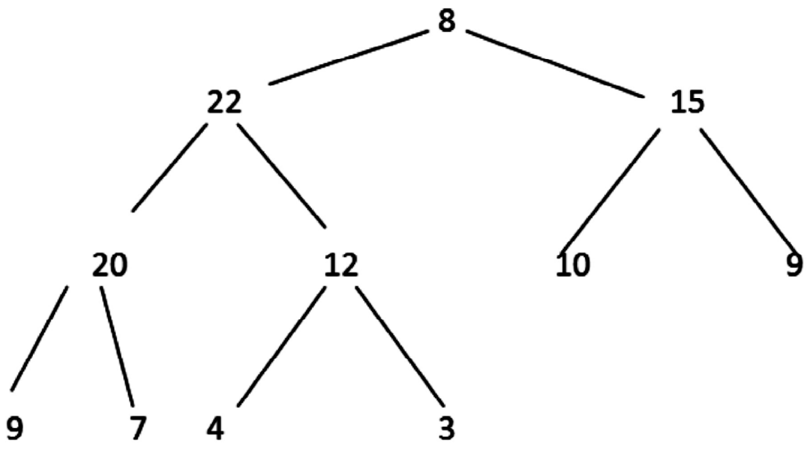
So, we swap node 9 and node 7



Step 8:

Node 20 contains greater element in its left child node.

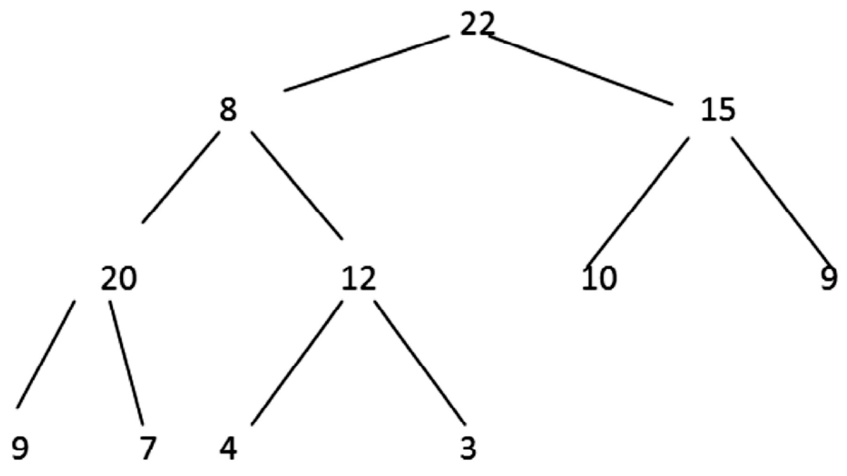
So, we swap node 22 and node 20



Step 9:

Node8 contains greater element in its left child node.

So, we swap node 8 and node 22



Final heap tree for the given list of values.

FACULTY OF SCIENCE

B.sc. II Year III-Semester(CBSC) Examination

May / June - 2018

DATA STRUCTURES

Paper - III : (Computer Science)

Time : 3 Hours [Max. Marks : 80

PART - A - (5 × 4 = 20 Marks)

[Short Answer type]

Note : Answer any **FIVE** of the following questions.

ANSWERS

1. Define an algorithm. Write the essential characteristics of an algorithm. **(Unit-I, SQA-3,4)**
2. Transform the following infix expressions into their equivalent postfix expressions:
   A + B × (C + D) / F + D × E and (A + B ^ D) / (E – F) + G **(Unit-I, SQA-22)**
3. Write a short notes on applications of queues. **(Unit-II, SQA-12)**
4. Explain the insertion of a node in singly linked lists with examples. **(Unit-II, SQA-17)**
5. Define the terms minimum spanning tree and connected components with examples. **(Unit-IV, SQA-4)**
6. Explain the hash function with any two implementation methods. **(Unit-IV, SQA-5)**
7. Apply the bubble sort algorithm to sort the list of data: 2,10,6,4,8. **(Unit-III, SQA-6)**
8. Explain the concepts of min-heap and max-heap with examples. **(Unit-IV, SQA-6,7)**

PART - B - (4 × 15 = 60 Marks)

[Essay Answer Type]

Note : Answer **ALL** the questions from the following.

9. (a) (i) Explain row-major and column-major memory representations for 2-D arrays. **(Out of Syllabus)**

   (ii) Consider an integer array, int A[3][4] in C+h If the base address s 1050, find the address of the element A[2]|3] with row-major and column-major representations of the array. **(Out of Syllabus)**

OR

   (b) Explain the stack operations with program codes and examples. **(Unit-I, Q.No.28,31)**

10. (a) Briefly describe about the direct recursion, indirect recursion and tree recursion with examples. **(Unit-II, Q.No.14)**

OR

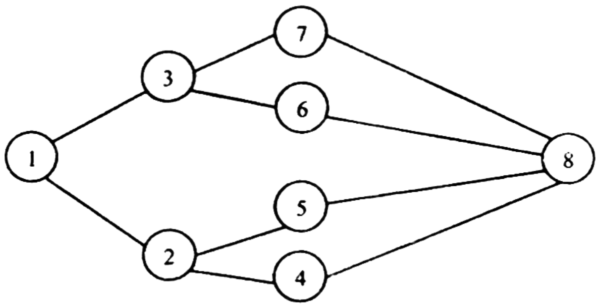
   (b) (i) What is a queue? Give the ADT for a queue. **(Unit-II, Q.No.19,21,22)**

(ii) Explain implementation of circular queue using arrays. **(Unit-II, Q.No.30,33)**

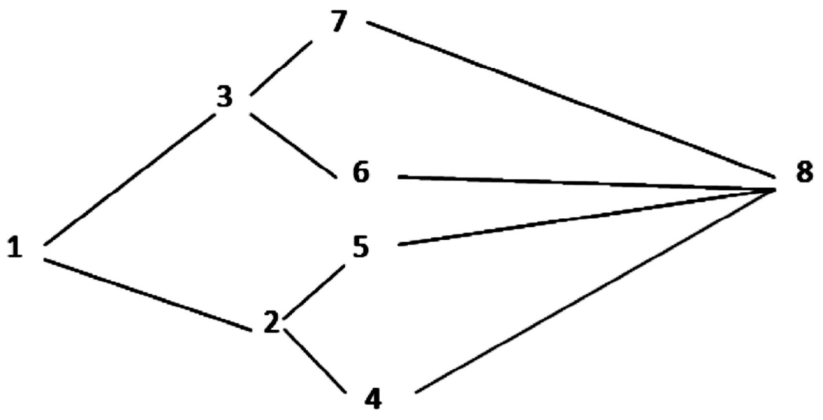
11. (a) Define binary search tree. Explain pre-order, in-order, post-order binary search tree traversals with examples. **(Unit-III, Q.No.10,11)**

OR

(b) Briefly describe about the graph traversal techniques. For the following graph, give the result of DFS and BFS traversals. **(Unit-III, Q.No.3,5)**



Ans :

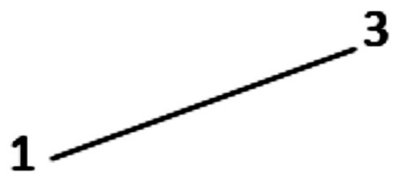


Visited:								
Stack								

Step1: We start from vertex 1, DFS puts all its adjacent nodes in stack

Visited:	1							
Stack	3	2						

Step 2: Next visit the element at the top of the stack, i.e 3 and go its adjacent nodes. And push them into top of the stack.

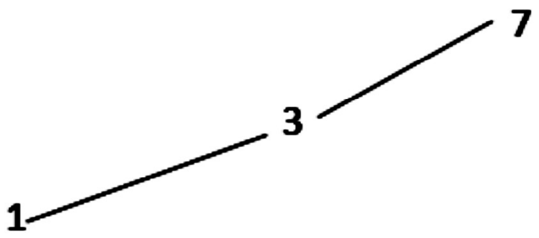


Visited:

1	3						

Stack

7	6	2					



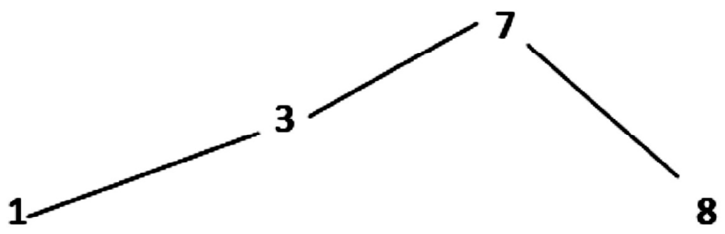
Visited:

1	3	7					

Stack

8	6	2					

Step4: Next visit the node from the top of the stack, ie 8. And push its unvisited adjacent nodes to the top of the stack.



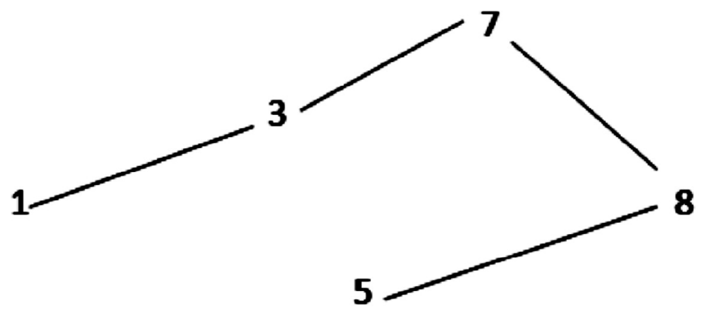
Visited:

1	3	7	8				

Stack

5	4	6	2				

Step 5: Next visit the node from the top of the stack, ie 5. And push its unvisited adjacent nodes to the top of the stack.



Visited:

1	3	7	8	5	2		

Stack

4	6	2					

Step6: Next visit the node from the top of the stack, ie 4. And push its unvisited adjacent nodes to the top of the stack. There are no unvisited node for 4 so remove 4 from the stack.

Visited:

1	3	7	8	5	2		

Stack

6	2						

Step7: Next visit the node from the top of the stack, ie 6. And push its unvisited adjacent nodes to the top of the stack. There are no unvisited node for 4 so remove 4 from the stack.

Visited:

1	3	7	8	5	2		

Stack

2							

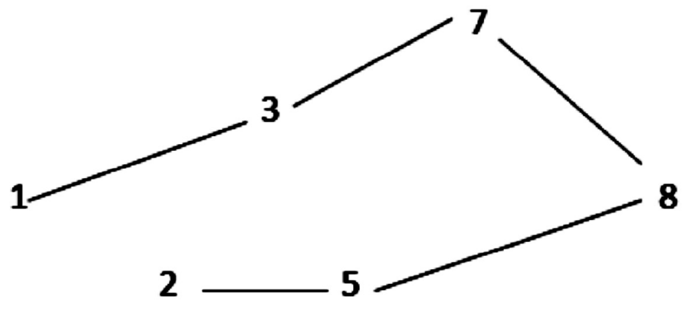
Step7: Next visit the node from the top of the stack, ie 6. And push its unvisited adjacent nodes to the top of the stack. There are no unvisited node for 4 so remove 4 from the stack.

Visited:

1	3	7	8	5	2		

Stack

Now the stack is empty. So all the nodes are visited. The final graph after the DFS i s



12. (a) Write the binary search algorithm and trace the steps followed to find 88 in the following list 8,12,24,28,32,45,48,51,56,62.

Ans :

0	1	2	3	4	5	6	7	8	9
8	12	24	28	32	45	48	51	56	62

$\rightarrow$ Index of thearray

Key to find K = 88

Find the mid value of the array by using

mid = (beg + end)/ 2 $\rightarrow$ mid = (1+9) / 2 = 5

Here mid element = 5th element = 45

Now compare mid element with key $\rightarrow$ k > mid (88> 45).

So search the right array. Now split the array

8	12	24	28	32

45	48	51	56	62

Mid

Find mid in splitted array

0	1	2	3
48	51	56	62

Mid = (0+3) / 2 = 2 mid value = 56

Now k > mid $\rightarrow$ 88> 56

Split array again after the mid element. Only one element left in the array

$\boxed{62}$ 62< 88 $\rightarrow$ no more elements to search.

Search ended . Item not found.

OR

(b) Write a program code for selection sort algorithm. Show the stepwise execution of the algorithm for the following list of data: 76, 67, 36, 55, 23, 14, 6. **(Unit-IV, Q.No.26,27)**

Ans :

76	67	36	55	23	14	6

$\rightarrow$ Unsorted array

↓

76	67	36	55	23	14	6

We started here , and find the minimum element and swap it with the first element of the array.

Minimum element is 6 swap 76 with 6

↓

6	67	36	55	23	14	76

Now we start with this and find the minimum element and swap it with the second element of the array. Now minimum element is 14 swap with 67

↓

6	14	36	55	23	67	76

Now we start with this and find the minimum element and swap it with the third element of the array. Now minimum element is 23 swap with 36

↓

6	14	23	55	36	67	76

Now we start with this and find the minimum element and swap it with the next element of the array. Now minimum element is 36 swap with 55

↓

6	14	23	36	55	67	76

Now we start with this and find the minimum element and swap it with the next element of the array. There is no minimum element in the array.

So array is sorted.

Sorted array

6	14	23	36	55	67	76

FACULTY OF SCIENCE

B.sc. II Year III-Semester(CBSC) Examination

November / December - 2017

DATA STRUCTURES

Paper - III : (Computer Science)

Time : 3 Hours　　[Max. Marks : 80

PART - A - (5 × 4 = 20 Marks)

[Short Answer type]

Note : Answer any **FIVE** of the following questions.

ANSWERS

1. Write the advantages and disadvantages of arrays. **(Out of Syllabus)**
2. Explain the postfix expression evaluation with an example. **(Unit-I, SQA-19)**
3. Differentiate between iteration and recursion approaches in problem solving. **(Unit-II, SQA-7)**
4. Write a short notes on double-ended queue (DEQUE). **(Unit-II, SQA-10)**
5. Briefly describe about the properties of a binary tree. **(Unit-III, SQA-1)**
6. Describe the adjacency matrix and adjacency list graph representations with example. **(Unit-IV, SQA-1,2)**
7. Explain the sequential search algorithm with an example. **(Out of Syllabus)**
8. Define a heap. Build the heap tree for the list of data. **(Unit-IV, SQA-3)**

PART - B - (4 × 15 = 60 Marks)

[Essay Answer Type]

Note : Answer **ALL** the questions from the following.

9. (a) (i) Briefly describe about the various types of data structure. **(Unit-I, Q.No.5)**

(ii) Define an algorithm. Write a flowchart and a pseudo-code to compute the sum of the first N natural numbers. **(Unit-I, Q.No.15)**

OR

(b) (i) What is a stack? Give the ADT for a stack. **(Unit-I, Q.No.28,30)**

(ii) Reverse the string “ABCDEF” using stack. **(Unit-I, Q.No.54)**

10. (a) Explain the queue operations with a program code and examples. **(Unit-II, Q.No.19,22)**

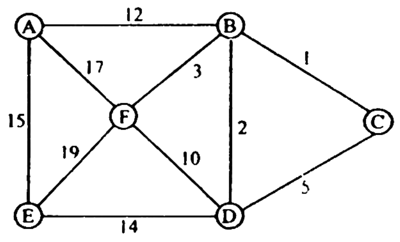
OR

(b) Explain the operations of insertion of a node, deletion of node and traversal in linked list with examples. **(Unit-III, Q.No.46,47,48,44)**

11. (a) Define binary search tree. Construct a binary search tree and explain the operations inserting a node, searching for a key, deleting a node with examples. **(Unit-III, Q.No.13,14)**

OR

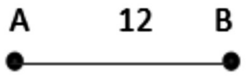
(b) Define the terms graph, tree, spanning tree and minimum spanning tree. Construct a minimum spanning tree (step-by-step) from the following graph using prim's algorithm. **(Unit-IV, Q.No.1,7,10)**



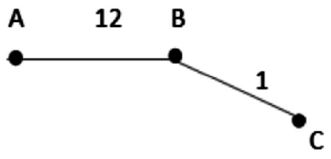
Ans :

Step 1: choose any vertex

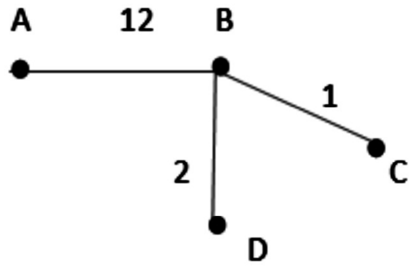
Step 2: Choose shortest edge from this vertex



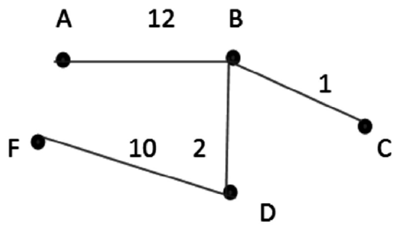
Step 3: choose the nearest vertex not in the solution.



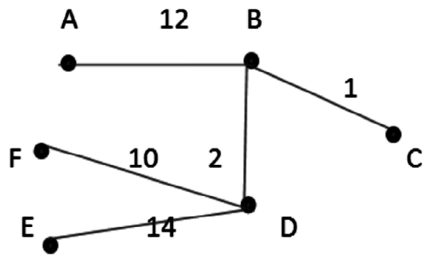
Step 4: choose the nearest edge not in the solution, if there are multiple choices, choose the random one.



Step 5: choose the nearest edge not in the solution, if there are multiple choices, choose the random one.



Step 6: choose the nearest edge not in the solution, if there are multiple choices, choose the random one.

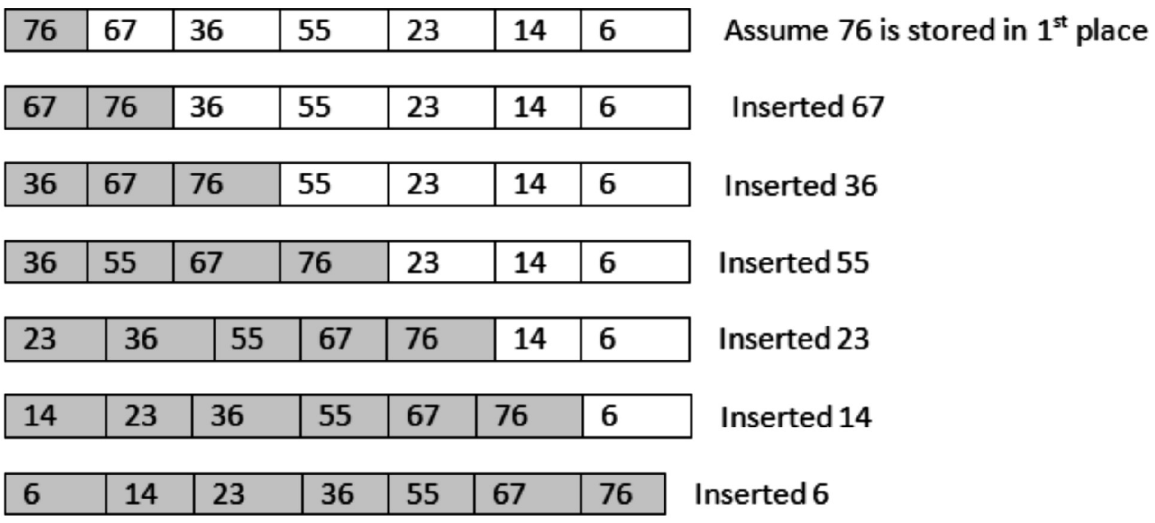


Finally we got the spanning tree. The weight of the spanning tree is

12+1+2+14+10 = 39

12. (a) Write a program code for insertion sort algorithm. Show the stepwise execution of the algorithm for the following list of data: 76, 67, 36, 55, 23, 14, 6.

Ans :



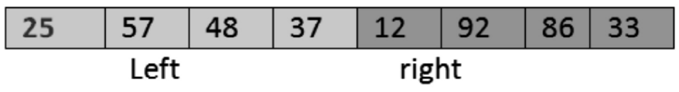
76	67	36	55	23	14	6	Assume 76 is stored in 1st place
67	76	36	55	23	14	6	Inserted 67
36	67	76	55	23	14	6	Inserted 36
36	55	67	76	23	14	6	Inserted 55
23	36	55	67	76	14	6	Inserted 23
14	23	36	55	67	76	6	Inserted 14
6	14	23	36	55	67	76	Inserted 6

OR

(b) Write an algorithm for quick sort. Show the stepwise execution of the algorithm for the following list of data: 25,57,48,37,12,92,86,33. **(Unit-IV, Q.No.32,33)**

Ans :

Pivot



25	57	48	37	12	92	86	33
	Left			right			

Let 37 be the Pivot element .Move Pivot to the end

25	57	48	33	12	92	86	37

Partion the sub array

R						L	
25	57	48	33	12	92	86	37

Move the left bound(L) to the right until it reaches grater than or equal to Pivot

R						L	
25	57	48	33	12	92	86	**37**

Step right (now pivot is 57)

	R					L	
25	57	48	33	12	92	86	**37**

Move right bound to the left until it finds the lesser than values to the pivot

Step left

	R				L		
25	57	48	33	12	92	86	**37**

Step left

	R			L			
25	57	48	33	12	92	86	**37**

Swap 12, 57

	R			L			
25	12	48	33	57	92	86	**37**

Move the left bound to the right until it reaches greater than or equal to pivot

	R			L			
25	12	48	33	57	92	86	**37**

Step right

R L

25	12	48	33	57	92	86	**37**

R L

25	12	48	33	57	92	86	**37**

R L

25	12	33	48	57	92	86	37

RL

25	12	33	48	57	92	86	37

When the right bound crosses, all the left side array is sorted.

Move the Pivot to the final location.

25	12	33	37	57	92	86	48

Apply the quick sort on the left sublist

25	12	33	37	57	92	86	48

Select the pivot (Now 12 is the Pivot)

Move the Pivot to the end.

R L

25	33	12	37	57	92	86	48

Move the right bound to the left unit it crosses. Then swap the Pivot to the left most element.

RL

12	33	25	37	57	92	86	48

Select the Pivot (Now 33) .as there are no elements to b sorted send the pivot the end

12	33	25	37	57	92	86	48

12	25	33	37	57	92	86	48

Left array is completely sorted

Apply the quick sort on right sublist

12	25	33	37	57	92	86	48

Select the Pivot(92) move it to the end

				R		L	
12	25	33	37	57	48	86	92

Step right

					R	L	
12	25	33	37	57	48	86	92

Move the right bound to the left until it crosses. Now all the elements of right are grater than pivot. Move the pivot to the final location. 92 is already in final location.

Apply the quick sort on left sublist.

Select the pivot 48 move the pivot to the last of the sublist

				R	L		
12	25	33	37	57	86	48	92

Partition the subarray. Move the r and l bounds until they cross .when the right bound crosses to the left all the elements of right list are greater than the pivot. Move the pivot to the last.

12	25	33	37	48	86	57	92

Call quicksort on the right sublist. Select pivot (86).

12	25	33	37	48	86	57	92

Place the Pivot at the end of sublist.

12	25	33	37	48	57	86	92

Now the list is sorted