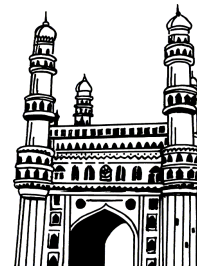


Rahul's ✓
Topper's Voice

AS PER
CBCS SYLLABUS



B.Sc.

III Year V Sem

Latest 2023 Edition

DATA STRUCTURES AND ALGORITHMS DATA SCIENCE PAPER - V

- 👉 Study Manual
- 👉 Short Question & Answers
- 👉 Multiple Choice Questions
- 👉 Fill in the blanks
- 👉 One Mark Answers
- 👉 Solved Model Papers

- by -

WELL EXPERIENCED LECTURER

Price
· 169-00



Rahul Publications™

Hyderabad. Ph : 66550071, 9391018098

All disputes are subjects to Hyderabad Jurisdiction only

B.Sc.

III Year V Sem

DATA STRUCTURES AND ALGORITHMS

DATA SCIENCE PAPER - V

Inspite of many efforts taken to present this book without errors, some errors might have crept in. Therefore we do not take any legal responsibility for such errors and omissions. However, if they are brought to our notice, they will be corrected in the next edition.

© No part of this publication should be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording and/or otherwise without the prior written permission of the publisher

Price ` . 169-00

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.

DATA STRUCTURES AND ALGORITHMS

DATA SCIENCE PAPER - V

STUDY MANUAL

Important Questions	IV - VI
Unit - I	1 - 40
Unit - II	41 - 78
Unit - III	79 - 104
Unit - IV	105 - 138

SOLVED MODEL PAPERS

MODEL PAPER - I	139 - 139
MODEL PAPER - II	140 - 141
MODEL PAPER - III	142 - 142

SYLLABUS

UNIT - I

Performance and Complexity Analysis: Space Complexity, Time Complexity, Asymptotic Notation (Big-Oh), Complexity Analysis Examples. Linear List-Array Representation: Vector Representation, Multiple Lists Single Array. Linear List-Linked Representation: Singly Linked Lists, Circular Lists, Doubly Linked Lists, Applications (Polynomial Arithmetic). Arrays and Matrices: Row and Column Major Representations, Sparse Matrices. Stacks: Array Representation, Linked Representation, Applications (Recursive Calls, Infix to Postfix, Postfix Evaluation). Queues: Array Representation, Linked Representation. Skip Lists and Hashing: Skip Lists Representation, Hash Table Representation, Application- Text Compression.

UNIT - II

Trees: Definitions and Properties, Representation of Binary Trees, Operations, Binary Tree Traversal. Binary Search Trees: Definitions, Operations on Binary Search Trees. Balanced Search Trees: AVL Trees, and B-Trees

UNIT - III

Graphs: Definitions and Properties, Representation, Graph Search Methods (Depth First Search and Breadth First Search) Application of Graphs: Shortest Path Algorithm (Dijkstra), Minimum Spanning Tree (Prim's and Kruskal's Algorithms).

UNIT - IV

Searching: Linear Search and Binary Search Techniques and their complexity analysis. Sorting and Complexity Analysis: Selection Sort, Bubble Sort, Insertion Sort, Quick Sort, Merge Sort, and Heap Sort. Algorithm Design Techniques: Greedy algorithm, divide-and-conquer, dynamic programming.

Contents

Topic	Page No.
UNIT - I	
1.1 Performance and Complexity Analysis	1
1.1.1 Space Complexity	2
1.1.2 Time Complexity	3
1.1.3 Asymptotic Notations (Big-Oh)	4
1.1.4 Complexity Analysis Examples	5
1.2 Linear List-Array Representation	5
1.2.1 Vector Representation	5
1.2.2 Multiple Lists Single Array	6
1.3 Linear List-Linked Representation	7
1.3.1 Singly linked list	7
1.3.2 Circular Lists	8
1.3.3 Doubly linked list	10
1.3.4 Application of Linked List (Polynomial Arithmetic)	11
1.4 Linear List-Linked Representation	16
1.4.1 Row and Column Major Representations	16
1.4.2 Sparse Matrices	18
1.5 Stacks	20
1.5.1 Array Representation of Stack	20
1.5.2 Linked list representation of stack	21
1.5.3 Applications (Recursive Calls, Infix to Postfix, Postfix Evaluation)	22
1.6 Queues	25
1.6.1 Array Representation	25
1.6.2 Linked Representation	27
1.7 Skip Lists and Hashing	28
1.7.1 Skip Lists Representation	28
1.7.2 Hash Table Representation, Application-Text Compression	29
➤ Short Questions and Answers	33 - 36
➤ Choose the Correct Answers	37 - 37
➤ Fill in the blanks	38 - 38
➤ Very short Answers	39 - 40

Topic	Page No.
-------	----------

UNIT - II

2.1	Trees	41
2.1.1	Definitions and Properties Representation of Binary Trees	41
2.1.2	Operations	45
2.1.3	Binary Tree Traversal	51
2.2	Binary Search Trees	53
2.2.1	Definitions, Operations on Binary Search Tress.	53
2.3	Balanced Search Trees	56
2.3.1	AVL Trees	56
2.3.2	B Trees	66
➤	Short Questions and Answers	70 - 74
➤	Choose the Correct Answers	75 - 75
➤	Fill in the blanks	76 - 76
➤	Very short Answers	77 - 78

UNIT - III

3.1	Graphs	79
3.1.1	Definitions	79
3.1.2	Graph Theory - Basic Properties	80
3.1.3	Representation of Graphs	82
3.1.4	Graph Search Methods (Depth First Search and Breadth First Search)	85
3.1.4.1	Depth First Search (DFS)	85
3.1.4.2	Breadth First Search (BFS)	87
3.2	Application of Graphs	89
3.2.1	Shortest Path Algorithm (Dijkstra)	89
3.2.2	Minimum Spanning Tree (Prim's and Kruskal's Algorithms)	93
3.2.2.1	Prim's Algorithm	93
3.2.2.2	Kruskal'sAlgorithm	95
➤	Short Questions and Answers	97 - 101
➤	Choose the Correct Answers	102 - 102
➤	Fill in the blanks	103 - 103
➤	Very short Answers	104 - 104

Topic**Page No.****UNIT - IV**

4.1	Searching	105
4.1.1	Linear Search	105
4.1.2	Binary Search	106
4.1.3	Complexity Analysis of Linear Search	108
4.1.4	Time Complexity Analysis of Binary Search	109
4.2	Sorting and Complexity Analysis	109
4.2.1	Selection Sort	109
4.2.2	Bubble Sort	111
4.2.3	Insertion Sort	113
4.2.4	Quick Sort	115
4.2.5	Merge Sort	117
4.2.6	Heap Sort	119
4.3	Algorithm Design Techniques	125
4.3.1	Greedy Algorithm	125
4.3.2	Divide and ConquerAlgorithm	127
4.3.3	Dynamic Programming	129
➤	Short Questions and Answers	130 - 134
➤	Choose the Correct Answers	135 - 136
➤	Fill in the blanks	137 - 137
➤	Very short Answers	138 - 138

Important Questions

UNIT - I

1. What is Performance Analysis of an algorithm?

Ans :

Refer Unit-I, Q.No. 1

2. What is Asymptotic Notation? Explain in detail about Big(O) notation.

Ans :

Refer Unit-I, Q.No. 4

3. Explain Singly Linked List and its Operations

Ans :

Refer Unit-I, Q.No. 8

4. Explain circular linked list and its operations.

Ans :

Refer Unit-I, Q.No. 9

5. What are the applications of Linked List?

Ans :

Refer Unit-I, Q.No. 11

6. How to represent Row and Column Major Representations.

Ans :

Refer Unit-I, Q.No. 12

7. Explain about Linked list Representation of stack.

Ans :

Refer Unit-I, Q.No. 15

8. Explain about Array representation of Queue

Ans :

Refer Unit-I, Q.No. 17

9. What is a skip list? Explain its Operations and Applications.

Ans :

Refer Unit-I, Q.No. 19

10. What is Hash Table? Explain its representation and Compression method

Ans :

Refer Unit-I, Q.No. 20

UNIT - II

1. What is a Tree data structure and How to represent a Binary tree? Explain its applications

Ans :

Refer Unit-II, Q.No. 1

2. Explain in detail about Binary Tree Traversal Technique?

Ans :

Refer Unit-II, Q.No. 3

3. Define a Binary Search Tree? And Explain its Operations.

Ans :

Refer Unit-II, Q.No. 4

4. What is AVL tree? Explain its operations.

Ans :

Refer Unit-II, Q.No. 5

5. Give an example how to construct an AVL Tree?

Ans :

Refer Unit-II, Q.No. 7

6. What is B Tree? Explain in detail about its Operations.

Ans :

Refer Unit-II, Q.No. 8

UNIT - III

1. What is Graph? Explain in detail about Graphs.

Ans :

Refer Unit-III, Q.No. 1

2. Explain about Graph basic properties in detail with an example.

Ans :

Refer Unit-III, Q.No. 2

3. Explain about DFS Search Algorithm.

Ans :

Refer Unit-III, Q.No. 4

4. Explain about Dijkstra's Shortest Path Algorithm.

Ans :

Refer Unit-III, Q.No. 6

5. Explain about prim's Algorithm

Ans :

Refer Unit-III, Q.No. 8

6. Explain about Kruskal's Algorithm.

Ans :

Refer Unit-III, Q.No. 9

UNIT - IV

1. Explain in detail about Linear Search with an Example?

Ans :

Refer Unit-IV, Q.No. 2

2. Explain in detail about Selection Sort with an example and its complexity.

Ans :

Refer Unit-IV, Q.No. 6

3. Explain in detail about Insertion Sort with an example and its complexity.

Ans :

Refer Unit-IV, Q.No. 8

4. Explain in detail about Quick Sort with an example and its complexity.

Ans :

Refer Unit-IV, Q.No. 9

5. Explain in detail about Heap Sort with an example and its complexity.

Ans :

Refer Unit-IV, Q.No. 11

6. Compare various Sorting Techniques with Real World Usage.

Ans :

Refer Unit-IV, Q.No. 12

7. Explain in detail about Greedy Algorithm with its applications.

Ans :

Refer Unit-IV, Q.No. 13

8. Explain in detail about Dynamic Prog-ramming with its applications.

Ans :

Refer Unit-IV, Q.No. 15

UNIT I

Performance and Complexity Analysis: Space Complexity, Time Complexity, Asymptotic Notation (Big-Oh), Complexity Analysis Examples. Linear List-Array Representation: Vector Representation, Multiple Lists Single Array. Linear List-Linked Representation: Singly Linked Lists, Circular Lists, Doubly Linked Lists, Applications (Polynomial Arithmetic). Arrays and Matrices: Row and Column Major Representations, Sparse Matrices. Stacks: Array Representation, Linked Representation, Applications (Recursive Calls, Infix to Postfix, Postfix Evaluation). Queues: Array Representation, Linked Representation. Skip Lists and Hashing: Skip Lists Representation, Hash Table Representation, Application- Text Compression.

1.1 PERFORMANCE AND COMPLEXITY ANALYSIS

Q1. What is Performance Analysis of an algorithm?

Ans:

(Imp.)

Introduction

Performance of an algorithm means predicting the resources which are required to an algorithm to perform its task.

If we want to go from city "A" to city "B", there can be many ways of doing this. We can go by flight, by bus, by train and also by bicycle. Depending on the availability and convenience, we choose the one which suits us. Similarly, in computer science, there are multiple algorithms to solve a problem. When we have more than one algorithm to solve a problem, we need to select the best one. Performance analysis helps us to select the best algorithm from multiple algorithms to solve a problem.

When there are multiple alternative algorithms to solve a problem, we analyze them and pick the one which is best suitable for our requirements. The formal definition is as follows...

Performance of an algorithm is a process of making evaluative judgement about algorithms.

It can also be defined as follows...

Performance of an algorithm means predicting the resources which are required to an algorithm to perform its task.

That means when we have multiple algorithms to solve a problem, we need to select a suitable algorithm to solve that problem.

We compare algorithms with each other which are solving the same problem, to select the best algorithm. To compare algorithms, we use a set of parameters or set of elements like memory required by that algorithm, the execution speed of that algorithm, easy to understand, easy to implement, etc.,

Generally, the performance of an algorithm depends on the following elements...

1. Whether that algorithm is providing the exact solution for the problem?
2. Whether it is easy to understand?
3. Whether it is easy to implement?
4. How much space (memory) it requires to solve the problem?
5. How much time it takes to solve the problem? Etc.,

When we want to analyse an algorithm, we consider only the space and time required by that particular algorithm and we ignore all the remaining elements.

Based on this information, performance analysis of an algorithm can also be defined as follows...

Performance analysis of an algorithm is the process of calculating 'space' and 'time' required by that algorithm.

Performance analysis of an algorithm is performed by using the following measures...

1. Space required to complete the task of that algorithm (Space Complexity). It includes program space and data space
2. Time required to complete the task of that algorithm (Time Complexity)

1.1.1 Space Complexity

Q2. What is Space complexity give with an example?

Ans :

When we design an algorithm to solve a problem, it needs some computer memory to complete its execution. For any algorithm, memory is required for the following purposes...

1. To store program instructions.
2. To store constant values.
3. To store variable values.
4. And for few other things like function calls, jumping statements etc.,.

Space complexity of an algorithm can be defined as follows...

Total amount of computer memory required by an algorithm to complete its execution is called as space complexity of that algorithm.

Generally, when a program is under execution it uses the computer memory for THREE reasons. They are as follows...

1. **Instruction Space:** It is the amount of memory used to store compiled version of instructions.
2. **Environmental Stack:** It is the amount of memory used to store information of partially executed functions at the time of function call.
3. **Data Space:** It is the amount of memory used to store all the variables and constants.

To calculate the space complexity, we must know the memory required to store different data type values (according to the compiler). For example, the C Programming Language compiler requires the following...

1. 2 bytes to store Integer value.
2. 4 bytes to store Floating Point value.
3. 1 byte to store Character value.
4. 6 (OR) 8 bytes to store double value.

Consider the following piece of code...

Example

```
int square(int a)
{
    return a*a;
}
```

In the above piece of code, it requires 2 bytes of memory to store variable 'a' and another 2 bytes of memory is used for return value.

That means, totally it requires 4 bytes of memory to complete its execution. And these 4 bytes of memory is fixed for any input value of 'a'. This space complexity is said to be Constant Space Complexity.

If any algorithm requires a fixed amount of space for all input values then that space complexity is said to be Constant Space Complexity.

If the amount of space required by an algorithm is increased with the increase of input value, then that space complexity is said to be Linear Space Complexity.

1.1.2 Time Complexity

Q3. What is Time complexity give with an example?

Ans :

Every algorithm requires some amount of computer time to execute its instruction to perform the task. This computer time required is called time complexity.

The time complexity of an algorithm can be defined as follows...

The time complexity of an algorithm is the total amount of time required by an algorithm to complete its execution.

Generally, the running time of an algorithm depends upon the following...

1. Whether it is running on Single processor machine or Multi processor machine.
2. Whether it is a 32 bit machine or 64 bit machine.
3. Read and Write speed of the machine.

4. The amount of time required by an algorithm to perform Arithmetic operations, logical operations, return value and assignment operations etc.,

5. Input data

Calculating Time Complexity of an algorithm based on the system configuration is a very difficult task because the configuration changes from one system to another system. To solve this problem, we must assume a model machine with a specific configuration. So that, we can able to calculate generalized time complexity according to that model machine.

To calculate the time complexity of an algorithm, we need to define a model machine. Let us assume a machine with following configuration...

- It is a Single processor machine
- It is a 32 bit Operating System machine
- It performs sequential execution
- It requires 1 unit of time for Arithmetic and Logical operations
- It requires 1 unit of time for Assignment and Return value
- It requires 1 unit of time for Read and Write operations

We calculate the time complexity of following example code by using the above-defined model machine...

Consider the following piece of code...

Example 1

```
int sum(int a, int b)
{
    return a + b;
}
```

In the above sample code, it requires 1 unit of time to calculate $a + b$ and 1 unit of time to return the value. That means, totally it takes 2 units of time to complete its execution. And it does not change based on the input values of a and b . That means for all input values, it requires the same amount of time i.e. 2 units.

If any program requires a fixed amount of time for all input values then its time complexity is said to be Constant Time Complexity.

Consider the following piece of code...

Example 2

```
int sum(int A[], int n)
{
    int sum = 0, i;
    for(i = 0; i < n; i++)
        sum = sum + A[i];
    return sum;
}
```

For the above code, time complexity can be calculated as follows...

Code	Cost Time required for line (Units)	Repeatability No. of Times Executed	Total Total Time required in worst case
int sumOfList(int A[], int n)			
{			
int sum = 0, i;	1	1	1
for(i = 0; i < n; i++)	1 + 1 + 1	1 + (n+1) + n	2n + 2
sum = sum + A[i];	2	n	2n
return sum;	1	1	1
}			
4n + 4 Total Time required			

In above calculation

Cost is the amount of computer time required for a single operation in each line.

Repeatability is the amount of computer time required by each operation for all its repetitions.

Total is the amount of computer time required by each operation to execute.

So above code requires ' $4n + 4$ ' Units of computer time to complete the task. Here the exact time is not fixed. And it changes based on the n value. If we increase the n value then the time required also increases linearly.

Totally it takes ' $4n + 4$ ' units of time to complete its execution and it is Linear Time Complexity.

If the amount of time required by an algorithm is increased with the increase of input value then that time complexity is said to be Linear Time Complexity.

1.1.3 Asymptotic Notations (Big-Oh)

Q4. What is Asymptotic Notation? Explain in detail about Big(Oh) notation.

Ans.:

(Imp.)

Whenever we want to perform analysis of an algorithm, we need to calculate the complexity of that algorithm. But when we calculate the complexity of an algorithm it does not provide the exact amount of resource required. So instead of taking the exact amount of resource, we represent that complexity in a general form (Notation) which produces the basic nature of that algorithm. We use that general form (Notation) for analysis process.

Asymptotic notation of an algorithm is a mathematical representation of its complexity.

For example, consider the following time complexities of two algorithms...

- Algorithm 1 : $5n^2 + 2n + 1$
- Algorithm 2 : $10n^2 + 8n + 3$

Generally, when we analyze an algorithm, we consider the time complexity for larger values of input data (i.e. 'n' value). In above two time complexities, for larger value of 'n' the term ' $2n + 1$ ' in algorithm 1 has least significance than the term ' $5n^2$ ', and the term ' $8n + 3$ ' in algorithm 2 has least significance than the term ' $10n^2$ '.

Here, for larger value of 'n' the value of most significant terms ($5n^2$ and $10n^2$) is very larger than the value of least significant terms ($2n + 1$ and $8n + 3$). So for larger value of 'n' we ignore the least significant terms to represent overall time required by an algorithm. In asymptotic notation, we use only the most significant terms to represent the time complexity of an algorithm.

Big - Oh Notation (O)

Big - Oh notation is used to define the upper bound of an algorithm in terms of Time Complexity.

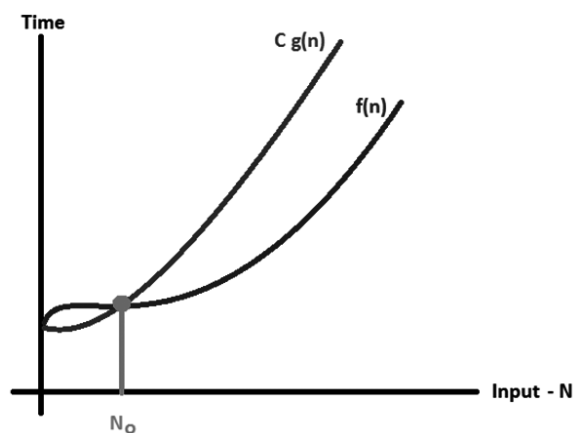
That means Big - Oh notation always indicates the maximum time required by an algorithm for all input values. That means Big - Oh notation describes the worst case of an algorithm time complexity.

Big - Oh Notation can be defined as follows...

Consider function $f(n)$ as time complexity of an algorithm and $g(n)$ is the most significant term. If $f(n) \leq C g(n)$ for all $n \geq n_0$, $C > 0$ and $n_0 \geq 1$. Then we can represent $f(n)$ as $O(g(n))$.

$$f(n) = O(g(n))$$

Consider the following graph drawn for the values of $f(n)$ and $C g(n)$ for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0 , always $C g(n)$ is greater than $f(n)$ which indicates the algorithm's upper bound.

Example

Consider the following $f(n)$ and $g(n)$...

$$f(n) = 3n + 2$$

$$g(n) = n$$

If we want to represent $f(n)$ as $O(g(n))$ then it must satisfy $f(n) \leq C g(n)$ for all values of $C > 0$ and $n_0 \geq 1$

$$f(n) \leq C g(n)$$

$$3n + 2 \leq C n$$

Above condition is always TRUE for all values of $C = 4$ and $n \geq 2$.

By using Big - Oh notation we can represent the time complexity as follows...

$$3n + 2 = O(n)$$

1.1.4 Complexity Analysis Examples

Q5. Explain about Complexity Analysis with an Examples.

Ans :

Time complexity of any algorithm is the time taken by the algorithm to complete. It is an important metric to show the efficiency of the algorithm and for comparative analysis. We tend to reduce the time complexity of algorithm that makes it more effective.

Example 1

Find the time complexity of the following code snippets

```
for(i= 0 ; i< n; i++) {
    cout<<i<< " ";
    i++;
}
```

The loop has maximum value n but the i will be incremented twice in the for loop which will make the time take half. So the time complexity is $O(n/2)$ which is equivalent to $O(n)$.

Example 2

Find the time complexity of the following code snippets.

```
for(i= 0 ; i< n; i++){
    for(j = 0; j<n ;j++){
        cout<<i<< " ";
    }
}
```

The inner loop and the outer loop both are executing n times. So for single value of i , j is looping n times, for n values of i , j will loop total $n*n = n^2$ times. So the time complexity is $O(n^2)$.

Example 3

Find the time complexity of the following code snippets

```
int i = n;
while(i){
    cout<<i<< " ";
    i = i/2;
}
```

In this case, after each iteration the value of i is turned into half of its previous value. So the series will be like: $n, n/2, n/4, \dots$. So the time complexity is $O(\log n)$.

Example 4

Find the time complexity of the following code snippets

```
if(i > j){
    j > 23 ? cout<<j : cout<<i;
}
```

There are two conditional statements in the code. Each conditional statement has time complexity = $O(1)$, for two of them it is $O(2)$ which is equivalent to $O(1)$ which is constant.

Example 5

Find the time complexity of the following code snippets

```
for(i= 0; i< n; i++){
    for(j = 1; j < n; j = j*2){
        cout<<i<< " ";
    }
}
```

The inner loop is executing $(\log n)$ times where the outer is executing n times. So for single value of i , j is executing $(\log n)$ times, for n values of i , j will loop total $n*(\log n) = (n \log n)$ times. So the time complexity is $O(n \log n)$.

1.2 LINEAR LIST-ARRAY REPRESENTATION

1.2.1 Vector Representation

Q6. How to represent the Vectors in Data Structure with an example

Ans :

Vectors

A collection of values that all have the same data type. The elements of a vector are all numbers, giving a numeric vector, or all character values, giving a character vector. A vector can be used to represent a single variable in a data set. Basic Vector Operations:

The vector class provides various methods to perform different operations on vectors.

1. Add elements
2. Access elements
3. Change elements
4. Delete elements

1. **Add Elements to a Vector:** To add a single element into a vector, we use the `push_back()` function. It inserts an element into the end of the vector.
2. **Access Elements of a Vector:** We use the index number to access the vector elements. Here, we use the `at()` function to access the element from the specified index.
3. **Change Vector Element:** We can change an element of the vector using the same `at()` function.
4. **Delete Elements from Vectors:** To delete a single element from a vector, we use the `pop_back()` function.

Vector Functions

The vector header file provides various functions that can be used to perform different operations on a vector.

Function	Description
<code>size()</code>	returns the number of elements present in the vector
<code>clear()</code>	removes all the elements of the vector
<code>front()</code>	returns the first element of the vector
<code>back()</code>	returns the last element of the vector
<code>empty()</code>	returns 1 (true) if the vector is empty
<code>capacity()</code>	check the overall size of a vector

1.2.2 Multiple Lists Single Array

Q7. Explain about Multiple Lists Single Array

Ans:

Array representation is basically wasteful of space when it is storing data that will change over time. To store some data, we allocate some space which is large enough to store multiple values in an array. Suppose we use the array doubling criteria to increase the size of the array. Consider the current array size is 8192. This is full. So we need to increase it by using array doubling technique. So new array size will be 16384. Then copy 8192 elements from old array to new array, then deallocate the old array. Now we can realize that before deallocating the space of the old array, the array size is thrice of 8192. The new array with double size and the old array. That is not so good approach.

When we want to store several lists we can share some larger array instead of creating new array for new lists. The multiple list in one array will be look like this.

10	20	30	40	50					60	70	80					2	3	5	7	11
List 1									List 2						List 3					

Though the multiple list in single array is memory efficient, but it has some problem also. Here insertion operation is more expensive. Because it may be necessary to move elements belonging to other lists to insert some element in the current list. And the representation is also harder to implement.

1.3 LINEAR LIST-LINKED REPRESENTATION

1.3.1 Singly linked list

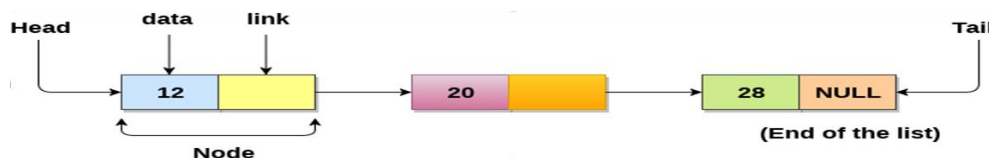
Q8. Explain singly linked list and its operations

Ans :

(Imp.)

A singly linked list is a type of linked list that is unidirectional, that is, it can be traversed in only one direction from head to the last node (tail). Each element in a linked list is called a node. A single node contains data and a pointer to the next node which helps in maintaining the structure of the list.

The first node is called the head; it points to the first node of the list and helps us access every other element in the list. The last node, also sometimes called the tail, points to NULL which helps us in determining when the list ends.



An example of a singly Linked List

Common Singly Linked List Operations:

- i) **Search for a node in the List:** It can determine and retrieve a specific node either from the front, the end, or anywhere in the list. The worst case Time Complexity for retrieving a node from anywhere in the list is $O(n)$.
- ii) **Add a node to the List:** It can add a node at the front, the end or anywhere in the linked list. The worst case Time Complexity for performing these operations is as follows:
 - Add item to the front of the list: $O(1)$
 - Add item to the end of the list: $O(n)$
 - Add item to anywhere in the list: $O(n)$
- iii) **Remove a node from the list:** It can remove a node either from the front, the end or from anywhere in the list.

The worst case Time Complexity for performing this operation is as follows:

- Remove item from the front of the list: $O(1)$
- Remove item from the end of the list: $O(n)$
- Remove item from anywhere in the list: $O(n)$

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.

S.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.

S.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.

1.3.2 Circular Lists

Q9. Explain circular linked list and its operations.

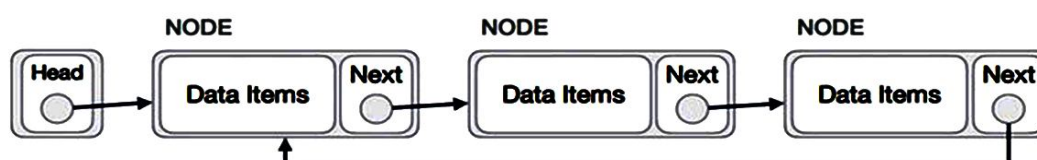
Ans:

(Imp.)

Circular Linked List is a variation of Linked list in which the first element points to the last element and the last element points to the first element. Both Singly Linked List and Doubly Linked List can be made into a circular linked list.

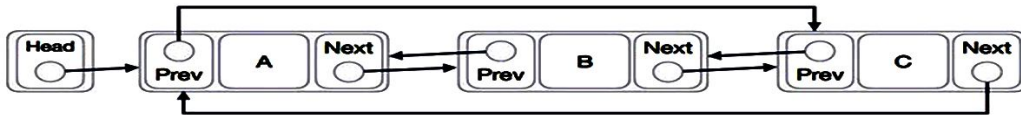
Singly Linked List as Circular

In singly linked list, the next pointer of the last node points to the first node.



Doubly Linked List as Circular

In doubly linked list, the next pointer of the last node points to the first node and the previous pointer of the first node points to the last node making the circular in both directions.



As per the above illustration, following are the important points to be considered.

- The last link's next points to the first link of the list in both cases of singly as well as doubly linked list.
- The first link's previous points to the last of the list in case of doubly linked list.

Basic Operations

Following are the important operations supported by a circular list.

- **insert:** Inserts an element at the start of the list.
- **delete:** Deletes an element from the start of the list.
- **display:** Displays the list.

Insertion Operation

Following code demonstrates the insertion operation in a circular linked list based on single linked list.

Example

```
insertFirst(data):
```

```
Begin
```

```
    create a new node
```

```
    node ->data := data
```

```
    if the list is empty, then
```

```
head := node
```

```
    next of node = head
```

```
else
```

```
    temp := head
```

```
    while next of temp is not head, do
```

```
temp := next of temp
```

```
done
```

```
    next of node := head
```

```
    next of temp := node
```

```
head := node
```

```
end if
```

```
End
```

Deletion Operation

Following code demonstrates the deletion operation in a circular linked list based on single linked list.deleteFirst():

```

deleteFirst():
Begin
    if head is null, then
        it is Underflow and return
    else if next of head = head, then
        head := null
        deallocate head
    else
        ptr := head
        while next of ptr is not head, do
            ptr := next of ptr
            next of ptr = next of head
        deallocate head
        head := next of ptr
    end if
End

```

Display List Operation

Following code demonstrates the display list operation in a circular linked list.

```

display():
Begin
    if head is null, then
        Nothing to print and return
    else
        ptr := head
        while next of ptr is not head, do
            display data of ptr
            ptr := next of ptr
            display data of ptr
        end if
    end if
End

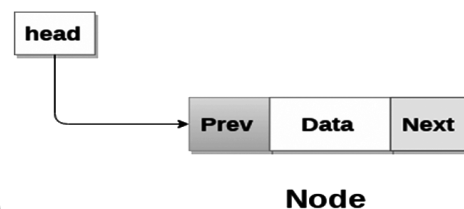
```

1.3.3 Doubly linked list

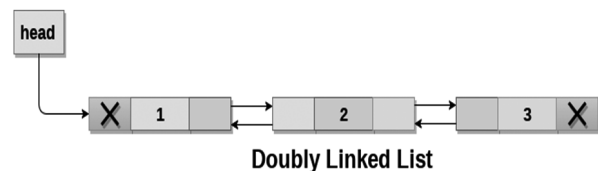
Q10. Explain Doubly linked list and its operations

Ans :

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.



The prev part of the first node and the next part of the last node will always contain null indicating end in each direction. In a singly linked list, we could traverse only in one direction, because each node contains address of the next node and it doesn't have any record of its previous nodes. However, doubly linked list overcome this limitation of singly linked list.

Due to the fact that, each node of the list contains the address of its previous node, we can find all the details about the previous node as well by using the previous address stored inside the previous part of each node.

All the operations regarding doubly linked list are described in the following table.

S. No.	Operations	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.

1.3.4 Application of Linked List (Polynomial Arithmetic)

Q11. What are the applications of Linked List?

Ans:

(Imp.)

A linked list is a linear data structure consisting of elements called nodes where each node is composed of two parts: an information part and a link part, also called the next pointer part.

Linked list is used in a wide variety of applications such as

- (i) Polynomial Manipulation
- (ii) Addition of long positive integers
- (iii) Representation of sparse matrices
- (iv) Addition of long positive integers
- (v) Symbol table creation
- (vi) Mailing list
- (vii) Memory management
- (viii) Linked allocation of files
- (ix) Multiple precision arithmetic etc

Polynomial Manipulation

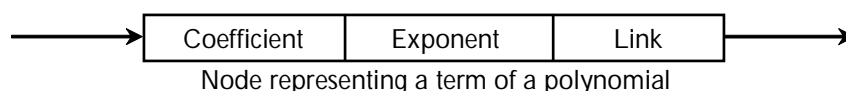
Polynomial manipulations are one of the most important applications of linked lists. Polynomials are an important part of mathematics not inherently supported as a data type by most languages. A polynomial is a collection of different terms, each comprising coefficients, and exponents. It can be represented using a linked list. This representation makes polynomial manipulation efficient.

While representing a polynomial using a linked list, each polynomial term represents a node in the linked list. To get better efficiency in processing, we assume that the term of every polynomial is stored within the linked list in the order of decreasing exponents. Also, no two terms have the same exponent, and no term has a zero coefficient and without coefficients. The coefficient takes a value of 1.

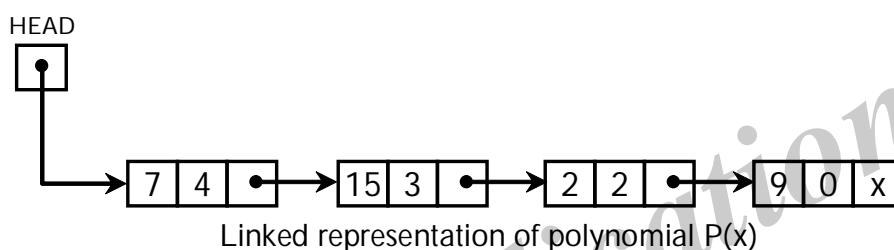
Each node of a linked list representing polynomial constitute three parts:

- The first part contains the value of the coefficient of the term.
- The second part contains the value of the exponent.
- The third part, LINK points to the next term (next node).

The structure of a node of a linked list that represents a polynomial is shown below:



Consider a polynomial $P(x) = 7x^2 + 15x^3 - 2x^2 + 9$. Here 7, 15, -2, and 9 are the coefficients, and 4, 3, 2, 0 are the exponents of the terms in the polynomial. On representing this polynomial using a linked list, we have



Observe that the number of nodes equals the number of terms in the polynomial. So we have 4 nodes. Moreover, the terms are stored to decrease exponents in the linked list. Such representation of polynomial using linked lists makes the operations like subtraction, addition, multiplication, etc., on polynomial very easy.

Addition of Polynomials

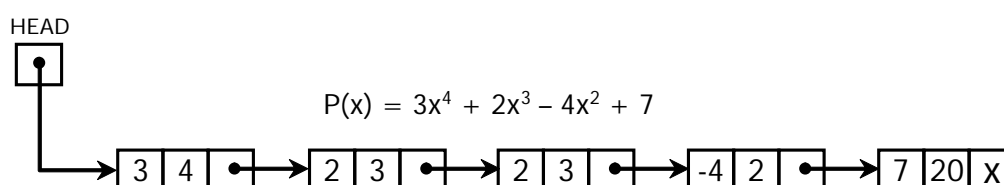
To add two polynomials, we traverse the list P and Q. We take corresponding terms of the list P and Q and compare their exponents. If the two exponents are equal, the coefficients are added to create a new coefficient. If the new coefficient is equal to 0, then the term is dropped, and if it is not zero, it is inserted at the end of the new linked list containing the resulting polynomial. If one of the exponents is larger than the other, the corresponding term is immediately placed into the new linked list, and the term with the smaller exponent is held to be compared with the next term from the other list. If one list ends before the other, the rest of the terms of the longer list is inserted at the end of the new linked list containing the resulting polynomial.

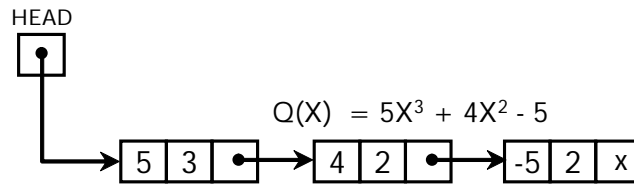
Let us consider an example an example to show how the addition of two polynomials is performed,

$$P(x) = 3x^4 + 2x^3 - 4x^2 + 7$$

$$Q(x) = 5x^3 + 4x^2 - 5$$

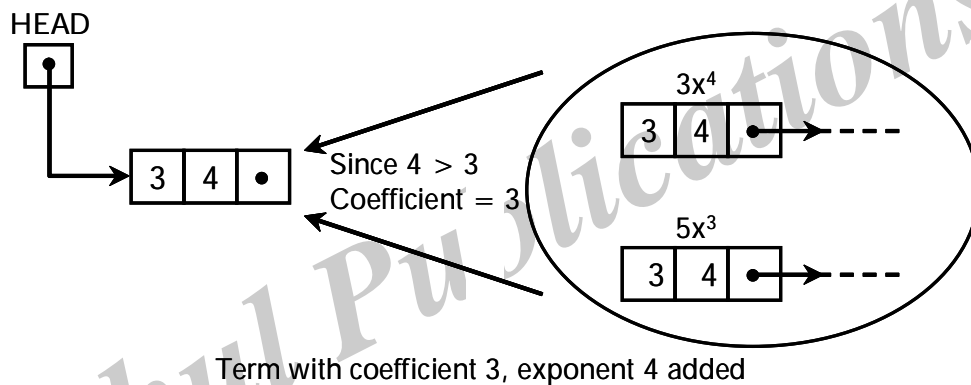
These polynomials are represented using a linked list in order of decreasing exponents as follows:



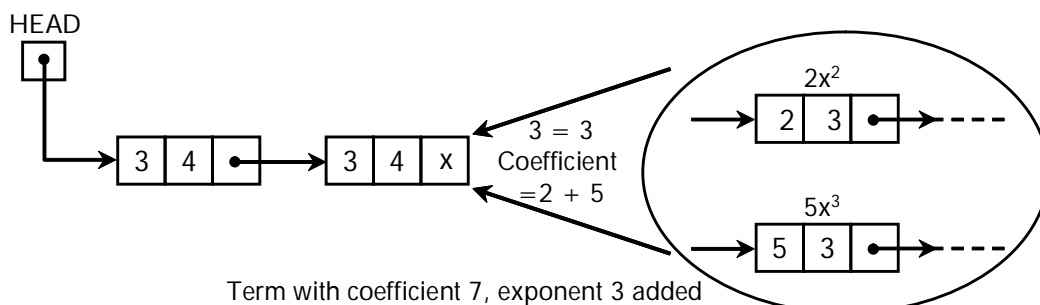


To generate a new linked list for the resulting polynomials that is formed on the addition of given polynomials $P(x)$ and $Q(x)$, we perform the following steps,

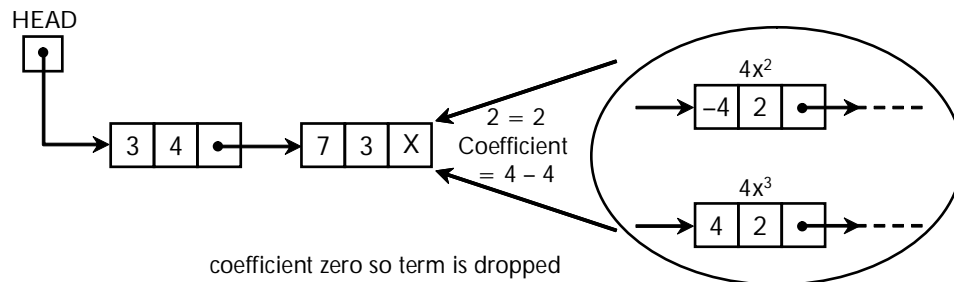
1. Traverse the two lists P and Q and examine all the nodes.
2. We compare the exponents of the corresponding terms of two polynomials. The first term of polynomials P and Q contain exponents 4 and 3, respectively. Since the exponent of the first term of the polynomial P is greater than the other polynomial Q , the term having a larger exponent is inserted into the new list. The new list initially looks as shown below:



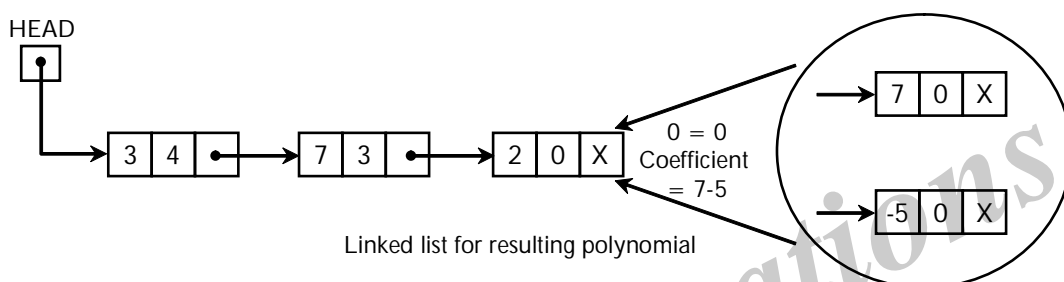
3. We then compare the exponent of the next term of the list P with the exponents of the present term of list Q . Since the two exponents are equal, so their coefficients are added and appended to the new list as follows:



4. Then we move to the next term of P and Q lists and compare their exponents. Since exponents of both these terms are equal and after addition of their coefficients, we get 0, so the term is dropped, and no node is appended to the new list after this,



5. Moving to the next term of the two lists, P and Q, we find that the corresponding terms have the same exponents equal to 0. We add their coefficients and append them to the new list for the resulting polynomial as shown below:



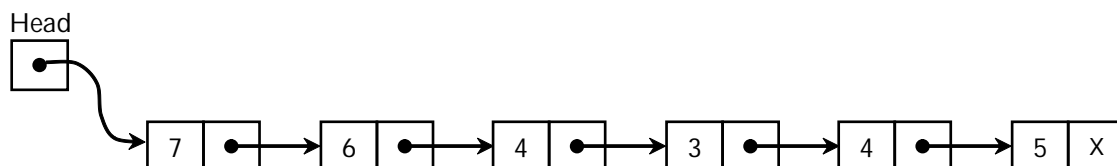
Addition of long positive integer using linked list

Most programming languages allow restrictions on the maximum value of integers stored. The maximum value of the largest integers is 32767, and the largest is 2147483647. Sometimes, applications such as security algorithms and cryptography require storing and manipulating integers of unlimited size. So in such a situation, it is desirable to use a linked list for storing and manipulating integers of arbitrary length.

Adding long positive integers can be performed effectively using a circular linked list. As we know that when we add two long integers, the digits of the given numbers are individually traversed from right to left, and the corresponding digits of the two numbers along with a carry from prior digits sum are added. So to accomplish addition, we must need to know how the digits of a long integer are stored in a linked list.

The digits of a long integer must be stored from right to left in a linked list so that the first node on the list contains the least significant digit, i.e., the right most digit, and the last node contains the most significant digit, i.e., left most digit.

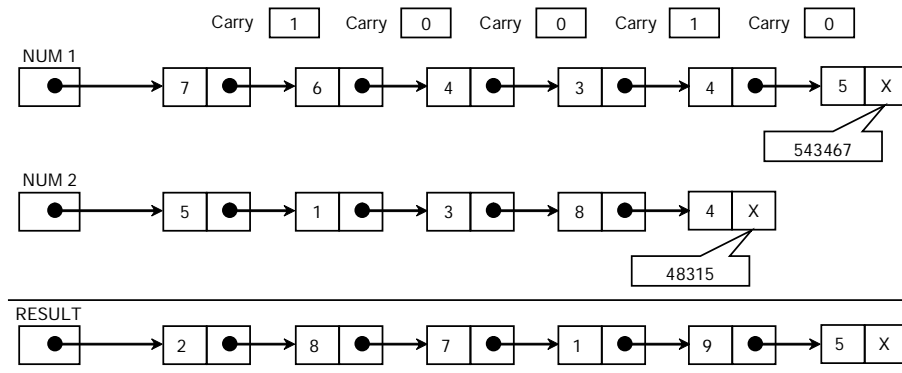
Example: An integer value 543467 can be represented using a linked list as



For performing the addition of two long integers, the following steps need to be followed:

- Traverse the two linked lists in parallel from left to right.
- During traversal, corresponding digits and a carry from prior digits sum are added, then stored in the new node of the resultant linked list.

The first positive long integer 543467 is represented using a linked list whose first node is pointed by NUM1 pointer. Similarly, the second positive long integer 48315 is represented using the second linked list whose first node is pointed by NUM2 pointer. These two numbers are stored in the third linked list whose first node is pointed to by the RESULT pointer.

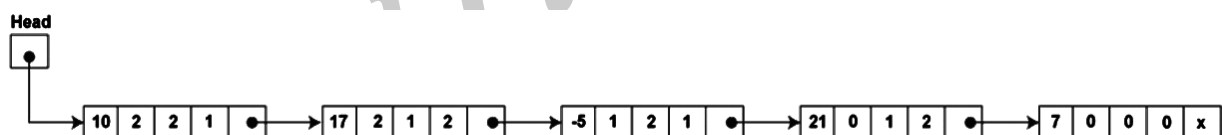


Polynomial of Multiple Variables

We can represent a polynomial with more than one variable, i.e., it can be two or three variables. Below is a node structure suitable for representing a polynomial with three variables X, Y, Z using a singly linked list.

COEFF	EXPX	EXPY	EMPZ	LINK
-------	------	------	------	------

Consider a polynomial $P(x, y, z) = 10x^2y^2z + 17x^2yz^2 - 5xy^2z + 21y^4z^2 + 7$. On representing this polynomial using linked list are:



Terms in such a polynomial are ordered accordingly to the decreasing degree in x. Those with the same degree in x are ordered according to decreasing degree in y. Those with the same degree in x and y are ordered according to decreasing degrees in z.

Some other applications of linked list

- **Memory Management:** Memory management is one of the operating system's key features. It decides how to allocate and reclaim storage for processes running on the system. We can use a linked list to keep track of portions of memory available for allocation.
- **Mailing List:** Linked lists have their use in email applications. Since it is difficult to predict multiple lists, maybe a mailer builds a linked list of addresses before sending a message.
- **LISP:** LISP is an acronym for LIST processor, an important programming language in artificial intelligence. This language extensively uses linked lists in performing symbolic processing.
- **Linked allocation of files:** A file of large size may not be stored in one place on a disk. So there must be some mechanism to link all the scattered parts of the file together. The use of a linked list allows an efficient file allocation method in which each block of a file contains a pointer to the file's text block. But this method is good only for sequential access.

- **Virtual Memory:** An interesting application of linked lists is found in the way systems support virtual memory.
- **Support for other data structures:** Some other data structures like stacks, queues, hash tables, graphs can be implemented using a linked list.

1.4 ARRAYS AND MATRICES

1.4.1 Row and Column Major Representations

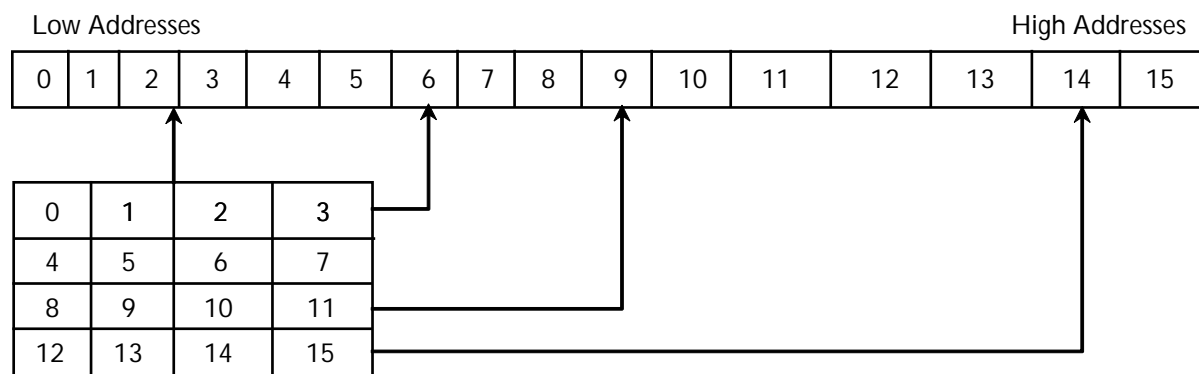
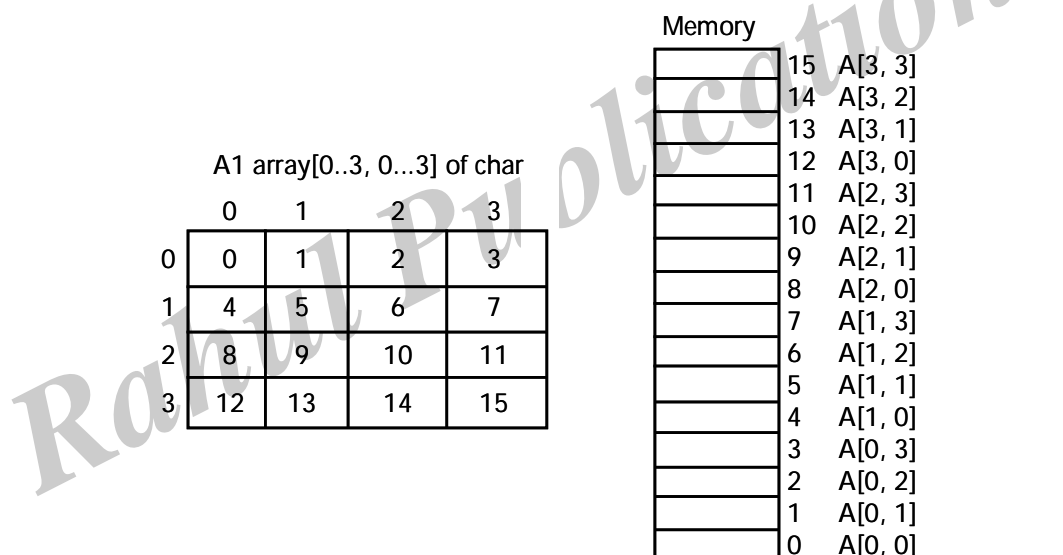
Q12. How to represent Row and Column Major Representations.

Ans :

(Imp.)

Row Major Ordering in an array

Row major ordering assigns successive elements, moving across the rows and then down the columns, to successive memory locations. In simple language, if the elements of an array are being stored in Row-Wise fashion. This mapping is demonstrated in the below figure: The formula to compute the Address (offset) for a two-dimension row-major ordered array as:



Address of $A[I][J] = \text{Base Address} + W * (C * I + j)$

Where **Base Address** is the address of the first element in an array.

- **W** = is the weight (size) of a data type.
- **C** = is Total No of Columns.
- **I** = is the Row Number
- **J** = is Column Number of an element whose address is to find out.

Column Major Ordering

If the element of an array is being stored in Column Wise fashion then it is called column-major ordering in an array. In row-major ordering, the right most index increased the fastest as you moved through consecutive memory locations. In column-major ordering, the leftmost index increases the fastest.

Pictorially, a column-major ordered array is organized as shown below.

A1 array[0..3, 0..3] of char				Memory	
	0	1	2	3	
0	0	1	2	3	15 A[3, 3]
1	4	5	6	7	14 A[3, 2]
2	8	9	10	11	13 A[3, 1]
3	12	13	14	15	12 A[3, 0]
					11 A[2, 3]
					10 A[2, 2]
					9 A[2, 1]
					8 A[2, 0]
					7 A[1, 3]
					6 A[1, 2]
					5 A[1, 1]
					4 A[1, 0]
					3 A[0, 3]
					2 A[0, 2]
					1 A[0, 1]
					0 A[0, 0]

The formulae for computing the address of an array element when using column-major ordering is very similar to that for row-major ordering. You simply reverse the indexes and sizes in the computation:

For a two-dimensional column-major **array**:

Address of $A[I][J] = \text{Base Address} + W * (R * J + I)$

Where **Base Address** is the address of the first element in an array.

W = is the weight (size) of a data type.

R = is Total No of Rows.

I = is the Row Number

J = is Column Number of an element whose address is to find out.

Address Calculation in single-dimensional Array

In 1-D array, there is no Row Major and no Column major concept, all the elements are stored in contiguous memory locations. We can calculate the address of the 1-D array by using the following formula:

Address of $A[I] = \text{Base Address} + W * I$.

Where I [is a location (Indexing) of element] whose address is to be found out. W (is the size of data type).

1.4.2 Sparse Matrices

Q13. What is Sparse Matrix? Explain the representation of Sparse Matrix.

Ans :

Matrix

A matrix can be defined as a two-dimensional array having 'm' rows and 'n' columns. A matrix with m rows and n columns is called $m \times n$ matrix. It is a set of numbers that are arranged in the horizontal or vertical lines of entries.

$$A = \begin{matrix} \xrightarrow{\text{Row (m)}} \\ \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \\ \downarrow \text{Columns (n)} \end{matrix}$$

Sparse Matrix

Sparse matrices are those matrices that have the majority of their elements equal to zero. In other words, the sparse matrix can be defined as the matrix that has a greater number of zero elements than the non-zero elements. The following benefits of using the sparse matrix.

Storage

We know that a sparse matrix contains lesser non-zero elements than zero, so less memory can be used to store elements. It evaluates only the non-zero elements.

Computing time

In the case of searching in sparse matrix, we need to traverse only the non-zero elements rather than traversing all the sparse matrix elements. It saves computing time by logically designing a data structure traversing non-zero elements.

Representation of sparse matrix

Now, let's see the representation of the sparse matrix. The non-zero elements in the sparse matrix can be stored using triplets that are rows, columns, and values. There are two ways to represent the sparse matrix that are listed as follows:

- Array representation
- Linked list representation

Array representation of the sparse matrix

Representing a sparse matrix by a 2D array leads to the wastage of lots of memory. This is because zeroes in the matrix are of no use, so storing

zeroes with non-zero elements is wastage of memory. To avoid such wastage, we can store only non-zero elements. If we store only non-zero elements, it reduces the traversal time and the storage space.

In 2D array representation of sparse matrix, there are three fields used that are named as -

row	column	value
-----	--------	-------

- **Row:** It is the index of a row where a non-zero element is located in the matrix.
- **Column:** It is the index of the column where a non-zero element is located in the matrix.
- **Value:** It is the value of the non-zero element that is located at the index (row, column).

Example

Let's understand the array representation of sparse matrix with the help of the example given below:

Consider the sparse matrix

Sparse Matrix →

	0	1	2	3
0	0	4	0	5
1	0	0	3	6
2	0	0	2	0
3	2	0	0	0
4	1	0	0	0

in the above figure, we can observe a 5x4 sparse matrix containing 7 non-zero elements and 13 zero elements. The above matrix occupies $5 \times 4 = 20$ memory space. Increasing the size of matrix will increase the wastage space.

The tabular representation of the above matrix is given below:

Table Structure

Row	Column	Value
0	1	4
0	3	5
1	2	3
1	3	6
2	2	2
3	0	2
4	0	1
5	4	7

In the above structure, first column represents the rows, the second column represents the columns, and the third column represents the non-zero value. The first row of the table represents the triplets. The first triplet represents that the value 4 is stored at 0th row and 1st column. Similarly, the second triplet represents that the value 5 is stored at the 0th row and 3rd column. In a similar manner, all triplets represent the stored location of the non-zero elements in the matrix.

The size of the table depends upon the total number of non-zero elements in the given sparse matrix. Above table occupies $8 \times 3 = 24$ memory space which is more than the space occupied by the sparse matrix. So, what's the benefit of using the sparse matrix? Consider the case if the matrix is 8×8 and there are only 8 non-zero elements in the matrix, then the space occupied by the sparse matrix would be $8 \times 8 = 64$, whereas the space occupied by the table represented using triplets would be $8 \times 3 = 24$.

Linked List representation of the sparse matrix:

In a linked list representation, the linked list data structure is used to represent the sparse matrix. The advantage of using a linked list to represent the sparse matrix is that the complexity of inserting or deleting a node in a linked list is lesser than the array.

Unlike the array representation, a node in the linked list representation consists of four fields. The four fields of the linked list are given as follows:

Row: It represents the index of the row where the non-zero element is located.

Column: It represents the index of the column where the non-zero element is located.

Value: It is the value of the non-zero element that is located at the index (row, column).

Next node: It stores the address of the next node.

The node structure of the linked list representation of the sparse matrix is shown in the below image:

Node Structure

Row	Column	Value	Pointer to Next Node
-----	--------	-------	----------------------

Example

The linked list representation of sparse matrix with the help of the example given below:

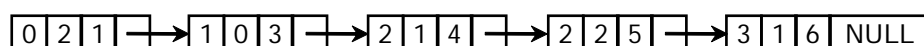
Consider the sparse matrix

Sparse Matrix \longrightarrow

	0	1	2	3
0	0	0	1	0
1	3	0	0	0
2	0	4	5	0
3	0	6	0	0

in the above figure, we can observe a 4×4 sparse matrix containing 5 non-zero elements and 11 zero elements. Above matrix occupies $4 \times 4 = 16$ memory space. Increasing the size of matrix will increase the wastage space.

The linked list representation of the above matrix is given below:



in the above figure, the sparse matrix is represented in the linked list form. In the node, the first field represents the index of the row, the second field represents the index of the column, the third field represents the value, and the fourth field contains the address of the next node. In the above figure, the

first field of the first node of the linked list contains 0, which means 0th row, the second field contains 2, which means 2nd column, and the third field contains 1 that is the non-zero element. So, the first node represents that element 1 is stored at the 0th row-2nd column in the given sparse matrix. In a similar manner, all of the nodes represent the non-zero elements of the sparse matrix.

1.5 STACKS

1.5.1 Array Representation of Stack

Q14. Explain about Array Representation of Stack

Ans :

In array representation, 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 onto the stack (push operation)

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 refer 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 overflowed 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

Time Complexity : O(1)

Deletion of an element from a stack (Pop operation)

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;
```

Time Complexity : O(1)

Visiting each element of the stack (Peek operation)

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

```

Time complexity: $O(n)$

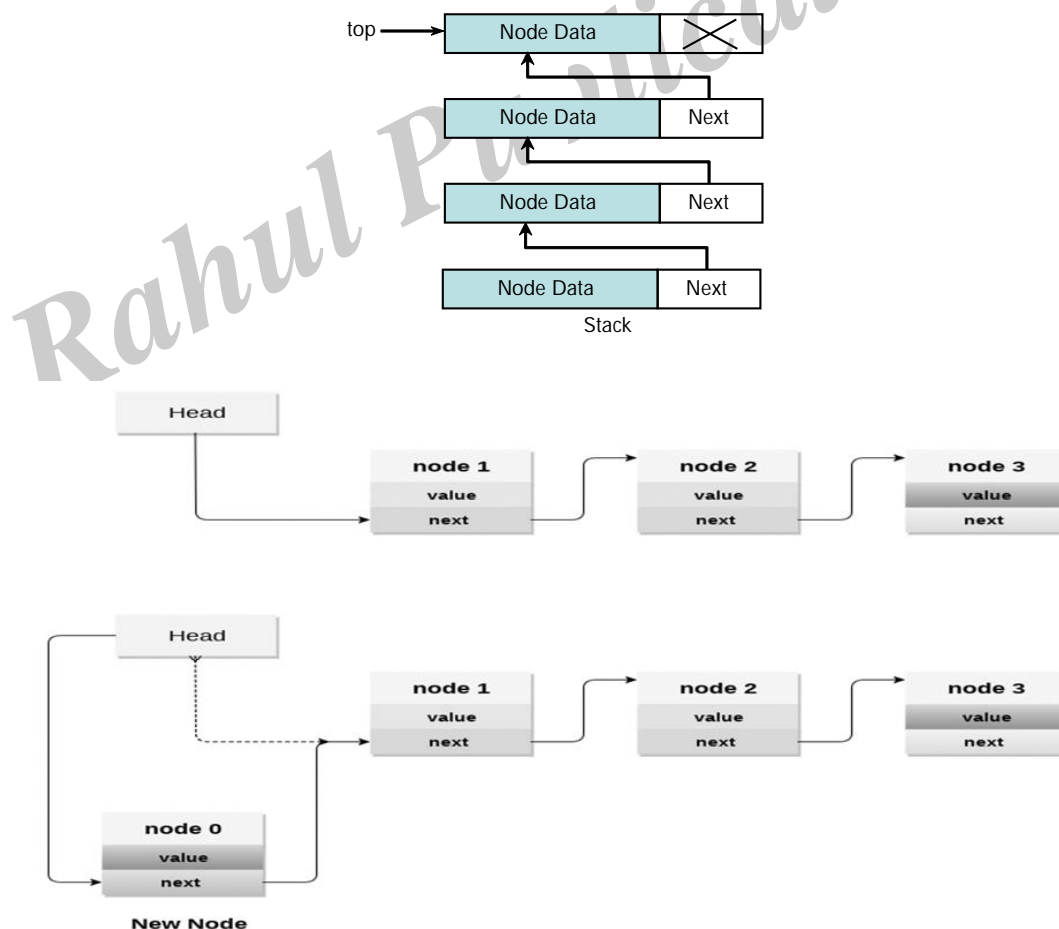
1.5.2 Linked list representation of stack**Q15. Explain about Linked list Representation of stack.**

Ans :

(Imp.)

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 representation 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.



Deleting a node from the stack (POP operation)

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.

Time Complexity : $O(n)$

Display the nodes (Traversing)

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.

Time Complexity : $O(n)$

1.5.3 Applications (Recursive Calls, Infix to Postfix, Postfix Evaluation)

Q16. Write short note on application of stack

- (i) Recursive
- (ii) Infix to Postfix
- (iii) Postfix Evaluation

Ans :

(i) Recursive Calls

Recursive functions use something called "the call stack." When a program calls a function, that function goes on top of the call stack. This is similar to a stack of books. You add things one at a time. Then, when you are ready to take something off, you always take off the top item.

The call stack in action with the factorial function. factorial(5) is written as 5! and it is defined like this: $5! = 5 * 4 * 3 * 2 * 1$. Here is a recursive function to calculate the factorial of a number:

```
function fact(x) {  
    if (x == 1) {  
        return 1;  
    } else {  
        return x * fact(x-1);  
    }  
}
```


(ii) Infix to Postfix

An infix and postfix are the expressions. An expression consists of constants, variables, and symbols. Symbols can be operators or parenthesis. All these components must be arranged according to a set of rules so that all these expressions can be evaluated using the set of rules.

Examples of expressions are:

$$5 + 6$$

$$A - B$$

$$(P * 5)$$

All the above expressions have a common structure, i.e., we have an operator between the two operands. An Operand is an object or a value on which the operation is to be performed. In the above expressions, 5, 6 are the operands while '+', '-', and '*' are the operators.

infix notation

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

$$\langle \text{operand} \rangle \langle \text{operator} \rangle \langle \text{operand} \rangle$$

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	\wedge
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

$$2 \wedge 2 \wedge 3 = 2 \wedge 8 \\ = 256$$

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.

(iii) Postfix Evaluation

The Postfix notation is used to represent algebraic expressions. The expressions written in postfix form are evaluated faster compared to infix notation as parenthesis is not required in postfix.

Evaluation of Postfix Expression

Example

Input: str = "2 3 1 * + 9 -"

Output: -4

Explanation:

Scan 2, it's a number, so push it to stack. Stack contains '2'

Scan 3, again a number, push it to stack, stack now contains '2 3' (from bottom to top)

Scan 1, again a number, push it to stack, stack now contains '2 3 1'

Scan *, it's an operator, pop two operands from stack, apply the * operator on operands, we get $3*1$ which results in 3. We push the result 3 to stack. The stack now becomes '2 3'.

Scan +, it's an operator, pop two operands from stack, apply the + operator on operands, we get $3 + 2$ which results in 5. We push the result 5 to stack. The stack now becomes '5'.

Scan 9, it's a number, so we push it to the stack. The stack now becomes '5 9'.

Scan -, it's an operator, pop two operands from stack, apply the - operator on operands, we get $5 - 9$ which results in -4. We push the result -4 to the stack. The stack now becomes '-4'.

There are no more elements to scan, we return the top element from the stack (which is the only element left in a stack).

Input: str = "100 200 + 2 / 5 * 7 +"

Output: 757

Evaluation of Postfix Expression Using Stack:

Follow the steps mentioned below to evaluate postfix expression using stack:

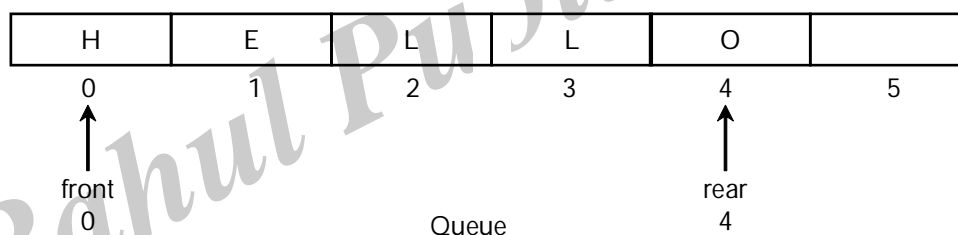
- Create a stack to store operands (or values).
- Scan the given expression from left to right and do the following for every scanned element.
- If the element is a number, push it into the stack
- If the element is an operator, pop operands for the operator from the stack. Evaluate the operator and push the result back to the stack
- When the expression is ended, the number in the stack is the final answer

1.6 QUEUES

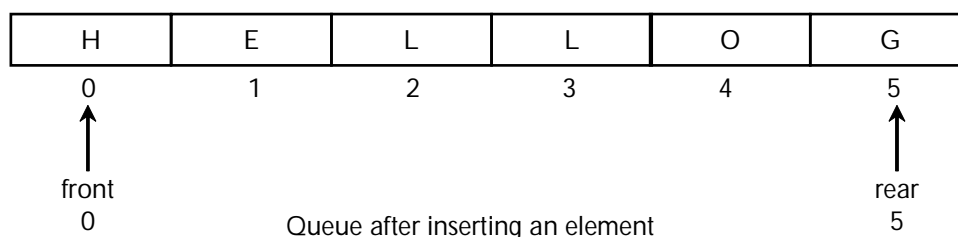
1.6.1 Array Representation**Q17. Explain about Array representation of Queue**

Ans : (Imp.)

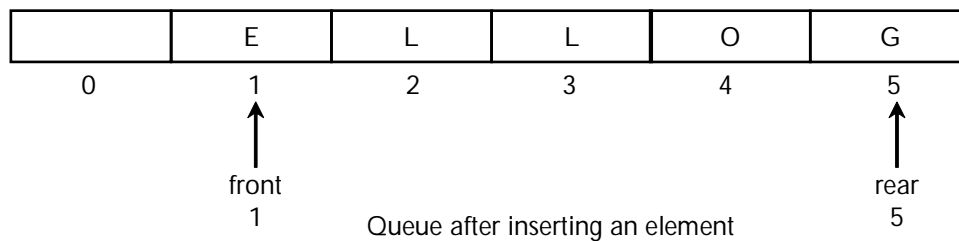
We can easily represent queue by using linear arrays. There are two variables i.e. front and rear, that are implemented in the case of every queue. Front and rear variables point to the position from where insertions and deletions are performed in a queue. Initially, the value of front and queue is -1 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.



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.



After deleting an element, the value of front will increase from -1 to 0. however, the queue will look something like following.



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

1.6.2 Linked Representation

Q18. Explain about Linked List Implementation of Queue

Ans :

The array implementation cannot be used for the large-scale applications where the queues are implemented. One of the alternatives of array implementation is linked list implementation of queue.

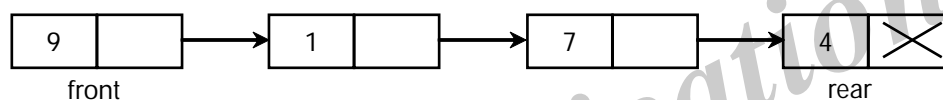
The storage requirement of linked representation of a queue with n elements is $O(n)$ while the time requirement for operations is $O(1)$.

In a linked queue, each node of the queue consists of two parts i.e. data part and the link part. Each element of the queue points to its immediate next element in the memory.

In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer. The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.

Insertion and deletions are performed at rear and front end respectively. If front and rear both are NULL, it indicates that the queue is empty.

The linked representation of queue is shown in the following figure.



Linked Queue

Operation on Linked Queue

There are two basic operations which can be implemented on the linked queues. The operations are Insertion and Deletion.

Insert operation

The insert operation appends the queue by adding an element to the end of the queue. The new element will be the last element of the queue.

Algorithm

- Step 1:** Allocate the space for the new node PTR
- Step 2:** SET PTR -> DATA = VAL
- Step 3:** IF FRONT = NULL
 SET FRONT = REAR = PTR
 SET FRONT -> NEXT = REAR -> NEXT = NULL
 ELSE
 SET REAR -> NEXT = PTR
 SET REAR = PTR
 SET REAR -> NEXT = NULL
 [END OF IF]
- Step 4:** END

Deletion

Deletion operation removes the element that is first inserted among all the queue elements. Firstly, we need to check either the list is empty or not. The condition $front == NULL$ becomes true if the list is empty, in this case, we simply write underflow on the console and make exit.

Algorithm

- Step 1:** IF FRONT = NULL
Write " Underflow "
Go to Step 5
[END OF IF]
- Step 2:** SET PTR = FRONT
- Step 3:** SET FRONT = FRONT -> NEXT
- Step 4:** FREE PTR
- Step 5:** END

1.7 SKIP LISTS AND HASHING**1.7.1 Skip Lists Representation****Q19. What is a skip list? Explain its Operations and Applications.***Ans :***(Imp.)**

A skip list is a probabilistic data structure. The skip list is used to store a sorted list of elements or data with a linked list. It allows the process of the elements or data to view efficiently. In one single step, it skips several elements of the entire list, which is why it is known as a skip list.

The skip list is an extended version of the linked list. It allows the user to search, remove, and insert the element very quickly. It consists of a base list that includes a set of elements which maintains the link hierarchy of the subsequent elements.

Skip list structure

It is built in two layers: The lowest layer and Top layer.

The lowest layer of the skip list is a common sorted linked list, and the top layers of the skip list are like an "express line" where the elements are skipped.

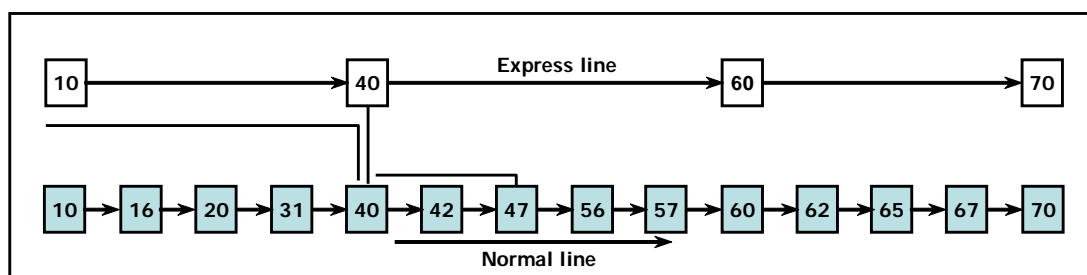
Working of the Skip list

In this example, we have 14 nodes, such that these nodes are divided into two layers, as shown in the diagram.

The lower layer is a common line that links all nodes, and the top layer is an express line that links only the main nodes, as you can see in the diagram.

Suppose you want to find 47 in this example. You will start the search from the first node of the express line and continue running on the express line until you find a node that is equal a 47 or more than 47.

You can see in the example that 47 does not exist in the express line, so you search for a node of less than 47, which is 40. Now, you go to the normal line with the help of 40, and search the 47, as shown in the diagram.



Skip List Basic Operations

There are the following types of operations in the skip list.

- **Insertion operation:** It is used to add a new node to a particular location in a specific situation.
- **Deletion operation:** It is used to delete a node in a specific situation.
- **Search Operation:** The search operation is used to search a particular node in a skip list.

Applications of the Skip list

1. It is used in distributed applications, and it represents the pointers and system in the distributed applications.
2. It is used to implement a dynamic elastic concurrent queue with low lock contention.
3. It is also used with the QMap template class.
4. The indexing of the skip list is used in running median problems.
5. The skip list is used for the delta-encoding posting in the Lucene search.

1.7.2 Hash Table Representation, Application-Text Compression

Q20. What is Hash Table? Explain its representation and Compression method

Ans :

(Imp.)

Hash Table

Hash table is one of the most important data structures that uses a special function known as a hash function that maps a given value with a key to access the elements faster.

A Hash table is a data structure that stores some information, and the information has basically two main components, i.e., key and value. The hash table can be implemented with the help of an associative array. The efficiency of mapping depends upon the efficiency of the hash function used for mapping.

For example, suppose the key value is John and the value is the phone number, so when we pass the key value in the hash function shown as below:

$\text{Hash}(\text{key}) = \text{index};$

When we pass the key in the hash function, then it gives the index.

$\text{Hash}(\text{john}) = 3;$

The above example adds the john at the index 3.

Hashing

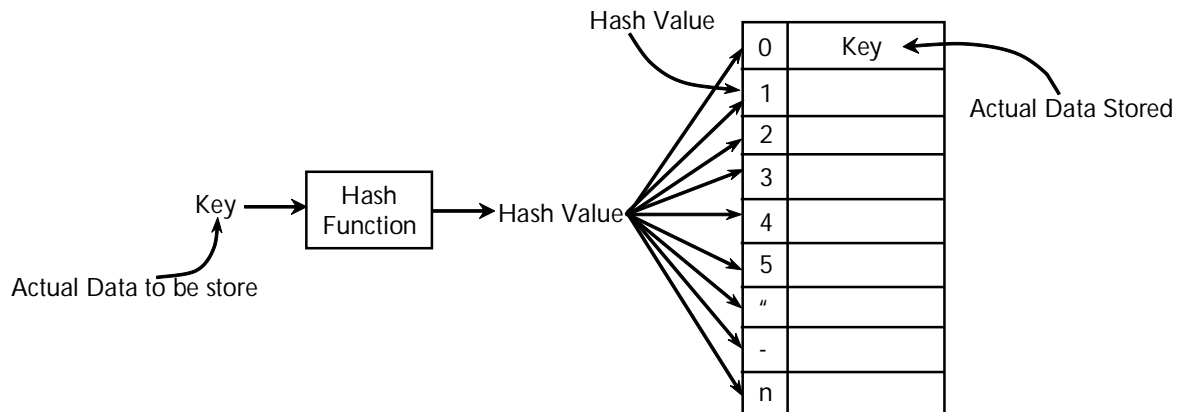
Hashing is one of the searching techniques that uses a constant time. The time complexity in hashing is $O(1)$. Till now, we read the two techniques for searching, i.e., linear search and binary search.

The worst time complexity in linear search is $O(n)$, and $O(\log n)$ in binary search. In both the searching techniques, the searching depends upon the number of elements but we want the technique that takes a constant time. So, hashing technique came that provides a constant time.

In Hashing technique, the hash table and hash function are used. Using the hash function, we can calculate the address at which the value can be stored.

The main idea behind the hashing is to create the (key/value) pairs. If the key is given, then the algorithm computes the index at which the value would be stored. It can be written as:

$$\text{Index} = \text{hash}(\text{key})$$



There are three ways of calculating the hash function:

- Division method
- Folding method
- Mid square method

In the division method, the hash function can be defined as:

$$h(k_i) = k_i \% m;$$

where **m** is the size of the hash table.

For example, if the key value is 6 and the size of the hash table is 10. When we apply the hash function to key 6 then the index would be:

$$h(6) = 6 \% 10 = 6$$

The index is 6 at which the value is stored.

Collision

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.

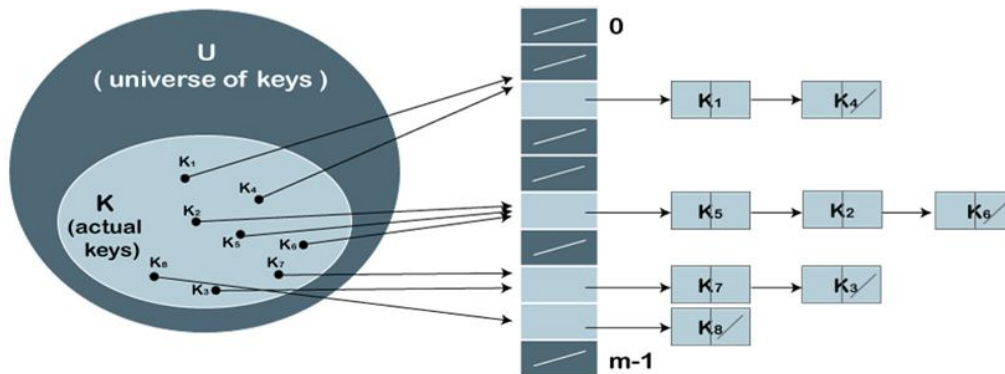
The following are the collision techniques:

- Open Hashing: It is also known as closed addressing.
- Closed Hashing: It is also known as open addressing.

Open Hashing

In Open Hashing, one of the methods used to resolve the collision is known as a chaining method.

Collision Resolution by Chaining



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 $h(k) = k/m$ as $h(k) = 2k+3$

- The index of key value 3 is:
 $\text{index} = h(3) = (2(3)+3)\%10 = 9$
 The value 3 would be stored at the index 9.
- The index of key value 2 is:
 $\text{index} = h(2) = (2(2)+3)\%10 = 7$
 The value 2 would be stored at the index 7.
- The index of key value 9 is:
 $\text{index} = h(9) = (2(9)+3)\%10 = 1$
 The value 9 would be stored at the index 1.
- The index of key value 6 is:
 $\text{index} = h(6) = (2(6)+3)\%10 = 5$
 The value 6 would be stored at the index 5.
- The index of key value 11 is:
 $\text{index} = h(11) = (2(11)+3)\%10 = 5$

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:

$$\text{index} = h(13) = (2(13)+3)\%10 = 9$$

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:

$$\text{index} = h(7) = (2(7)+3)\%10 = 7$$

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:

$$\text{index} = h(12) = (2(12)+3)\%10 = 7$$

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	$((2*12)+3)\%10 = 7$

Methods for Compressing

1. The Division Method:

$$h(k) = |k| \bmod N$$

N, the size of the array, should be a prime number so that it spreads out the generated codes for typical sequence of integers. Consider N=10 and sequence 10, 20, 30, ... or 5, 10, 15, 20, ...

We want to different integers two have probability 1/N for generating a bucket value.

2. The Mad Method

Multiply-Add-Divide

$$h(k) = |ak + b| \bmod N$$

Short Question and Answers

1. What is Space Complexity give with an Example?

Ans :

When we design an algorithm to solve a problem, it needs some computer memory to complete its execution. For any algorithm, memory is required for the following purposes...

- (a) To store program instructions.
- (b) To store constant values.
- (c) To store variable values.
- (d) And for few other things like function calls, jumping statements etc.,.

Space complexity of an algorithm can be defined as follows...

Total amount of computer memory required by an algorithm to complete its execution is called as space complexity of that algorithm.

Generally, when a program is under execution it uses the computer memory for THREE reasons. They are as follows...

- i) **Instruction Space:** It is the amount of memory used to store compiled version of instructions.
- ii) **Environmental Stack:** It is the amount of memory used to store information of partially executed functions at the time of function call.
- iii) **Data Space:** It is the amount of memory used to store all the variables and constants.

2. What is Time complexity give with an example?

Ans :

Every algorithm requires some amount of computer time to execute its instruction to perform the task. This computer time required is called time complexity.

The time complexity of an algorithm can be defined as follows...

The time complexity of an algorithm is the total amount of time required by an algorithm to complete its execution.

Generally, the running time of an algorithm depends upon the following...

- i) Whether it is running on Single processor machine or Multi processor machine.
- ii) Whether it is a 32 bit machine or 64 bit machine.
- iii) Read and Write speed of the machine.
- iv) The amount of time required by an algorithm to perform Arithmetic operations, logical operations, return value and assignment operations etc.,
- v) Input data

To calculate the time complexity of an algorithm, we need to define a model machine. Let us assume a machine with following configuration...

- i) It is a Single processor machine
- ii) It is a 32 bit Operating System machine
- iii) It performs sequential execution
- iv) It requires 1 unit of time for Arithmetic and Logical operations
- v) It requires 1 unit of time for Assignment and Return value
- vi) It requires 1 unit of time for Read and Write operations.

3. What is Hash Table? Explain its Representation and Compression method

Ans:

Hash Table

Hash table is one of the most important data structures that uses a special function known as a hash function that maps a given value with a key to access the elements faster.

A Hash table is a data structure that stores some information, and the information has basically two main components, i.e., key and value. The hash table can be implemented with the help of an associative array. The efficiency of mapping depends upon the efficiency of the hash function used for mapping.

For example, suppose the key value is John and the value is the phone number, so when we pass the key value in the hash function shown as below:

Hash(key) = index;

When we pass the key in the hash function, then it gives the index.

Hash(john) = 3;

The above example adds the john at the index 3.

4. Write the Compression Methods using Hash Table

Ans :

Methods for Compressing

i) The Division Method

$$h(k) = |k| \bmod N$$

N, the size of the array, should be a prime number so that it spreads out the generated codes for typical sequence of integers. Consider N=10 and sequence 10, 20, 30, ... or 5, 10, 15, 20, ...

We want to different integers two have probability 1/N for generating a bucket value.

ii) The Mad Method

Multiply-Add-Divide

$$h(k) = |ak + b| \bmod N$$

5. Describe the types of Data Structures?

Ans:

The following are the types of data structures:

Lists

A collection of related things linked to the previous or/and following data items.

Arrays

A collection of values that are all the same.

Records

A collection of fields, each of which contains data from a single data type.

Trees

A data structure that organizes data in a hierarchical framework. This form of data structure follows the ordered order of data item insertion, deletion, and modification.

Tables

The data is saved in the form of rows and columns. These are comparable to records in that the outcome or alteration of data is mirrored across the whole table.

6. What is a Linear Data Structure? Name a few examples.

Ans:

A data structure is linear if all its elements or data items are arranged in a sequence or a linear order. The elements are stored in a non-hierarchical way so that each item has successors and predecessors except the first and last element in the list.

Examples of linear data structures are Arrays, Stack, Strings, Queue, and Linked List.

7. What is a Data Structure? Give Applications of Data Structures?

Ans:

The Data Structure is the way data is organized (stored) and manipulated for retrieval and access. It also defines the way different sets of data relate to one another, establishing relationships and forming algorithms.

Applications of Data Structures

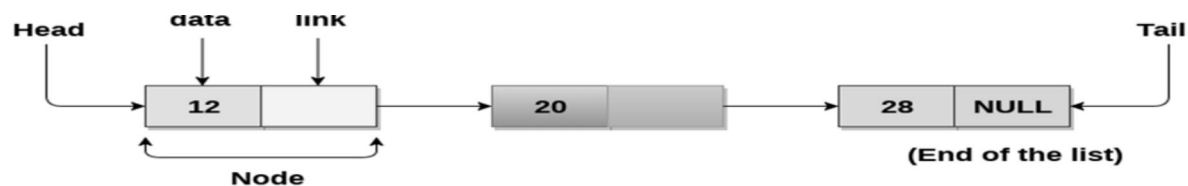
Numerical analysis, operating system, AI, compiler design, database management, graphics, statistical analysis, and simulation.

8. What is a linked list Data Structure?

Ans:

It's a linear Data Structure or a sequence of data objects where elements are not stored in adjacent memory locations. The elements are linked using pointers to form a chain. Each element is a separate object, called a node. Each node has two items: a data field and a reference to the next node. The entry point in a linked list is called the head. Where the list is empty, the head is a null reference and the last node has a reference to null.

A linked list is a dynamic data structure, where the number of nodes is not fixed, and the list has the ability to grow and shrink on demand.



9. What are the Advantages of a Linked List Over an Array?

Ans:

Advantages of a linked list over an array are:

1. Insertion and Deletion

Insertion and deletion of nodes is an easier process, as we only update the address present in the next pointer of a node. It's expensive to do the same in an array as the room has to be created for the new elements and existing elements must be shifted.

2. Dynamic Data Structure

As a linked list is a dynamic data structure, there is no need to give an initial size as it can grow and shrink at runtime by allocating and deallocating memory. However, the size is limited in an array as the number of elements is statically stored in the main memory.

3. No Wastage of Memory

As the size of a linked list can increase or decrease depending on the demands of the program, and memory is allocated only when required, there is no memory wasted. In the case of an array, there is memory wastage. For instance, if we declare an array of size 10 and store only five elements in it, then the space for five elements is wasted.

4. Implementation

Data structures like stack and queues are more easily implemented using a linked list than an array.

10. What is a stack? And where are stacks used?

Ans:

A stack is an abstract data type that specifies a linear data structure, as in a real physical stack or piles where you can only take the top item off the stack in order to remove things. Thus, insertion (push) and deletion (pop) of items take place only at one end called top of the stack, with a particular order: LIFO (Last In First Out) or FILO (First In Last Out).

Stacks used:

- Stacks Expression, evaluation, or conversion of evaluating prefix, postfix, and infix expressions
- Syntax parsing
- String reversal
- Parenthesis checking
- Backtracking

Choose the Correct Answers

1. Which one of the following is the process of inserting an element in the stack? [b]
(a) Add (b) Push
(c) Insert (d) None of the above
2. 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 _____. [b]
(a) Garbage collection (b) Underflow
(c) Overflow (d) None of the above
3. Which data structure is mainly used for implementing the recursive algorithm? [b]
(a) Queue (b) Stack
(c) Binary tree (d) Linked list
4. Which of the following is a linear data structure? [a]
(a) Array (b) AVL Trees
(c) Binary Trees (d) Graphs
5. Which of the following is not the type of queue? [b]
(a) Priority queue (b) Single-ended queue
(c) Circular queue (d) Ordinary queue
6. From following which is not the operation of data structure? [a]
(a) Operations that manipulate data in some way
(b) Operations that perform a computation
(c) Operations that check for syntax errors
(d) Operations that monitor an object for the occurrence of a controlling event
7. Which one of the following is an application of queue data structure? [d]
(a) When a resource is shared among multiple consumers.
(b) When data is transferred asynchronously
(c) Load Balancing
(d) All of the above
8. When a pop() operation is called on an empty queue, what is the condition called? [b]
(a) Overflow (b) Under flow
(c) Syntax Error (d) Garbage Value
9. Which of the following data structures can be used to implement queues? [b,c]
(a) Stack (b) Arrays
(c) Linked List (d) All of the Above
10. Which of the following is a Divide and Conquer algorithm? [d]
(a) Bubble Sort (b) Selection Sort
(c) Heap Sort (d) Merge Sort

Fill in the Blanks

1. A procedure that calls itself is called _____
2. What data structure is used for breadth first traversal of a graph is _____
3. If locality is a concern, you can use to _____ traverse the graph.
4. When the user tries to delete the element from the empty stack then the condition is said to be a _____
5. A list of elements in which enqueue operation takes place from one end, and dequeue operation takes place from one end is _____
6. _____ data structure is required to convert the infix to prefix notation
7. If the elements '1', '2', '3' and '4' are added in a stack, so what would be the order for the removal is _____
8. What is the outcome of the prefix expression +, -, *, 3, 2, /, 8, 4, 1 is _____
9. The minimum number of stacks required to implement a stack is _____
10. _____ is another name for the circular queue.

ANSWERS

1. Recursive
2. Queue
3. Depth first search
4. Underflow
5. Queue
6. Stack
7. '4', '3', '2' and '1'
8. 5
9. 2
10. Ring Buffer

One Mark Answers

1. What is an Algorithm?

Ans:

An algorithm is a step by step method of solving a problem or manipulating data. It defines a set of instructions to be executed in a certain order to get the desired output.

2. What is a Data Structure?

Ans:

The Data Structure is the way data is organized (stored) and manipulated for retrieval and access. It also defines the way different sets of data relate to one another, establishing relationships and forming algorithms.

3. Describe the types of Data Structures?

Ans:

The following are the types of data structures:

Lists, Arrays, Records, Trees and Tables

4. What is a Linear Data Structure? Name a few examples.

Ans:

A data structure is linear if all its elements or data items are arranged in a sequence or a linear order.

Examples: Arrays, Stack, Strings, Queue, and Linked List.

5. What are some applications of Data Structures?

Ans:

Numerical analysis, operating system, AI, compiler design, database management, graphics, statistical analysis, and simulation.

6. What is a linked list Data Structure?

Ans:

It's a linear Data Structure or a sequence of data objects where elements are not stored in adjacent memory locations. The elements are linked using pointers to form a chain. Each element is a separate object, called a node. Each node has two items: a data field and a reference to the next node.

7. Are linked lists considered linear or Non-linear Data Structures?

Ans:

Linked lists are considered both linear and non-linear data structures depending upon the application they are used for. When used for access strategies, it is considered as a linear data-structure. When used for data storage, it is considered a non-linear data structure.

8. What is a stack?*Ans:*

A stack is an abstract data type that specifies a linear data structure, insertion (push) and deletion (pop) of items take place only at one end called top of the stack, with a particular order: LIFO (Last In First Out) or FILO (First In Last Out).

9. Where are stacks used?*Ans:*

- Expression, evaluation, or conversion of evaluating prefix, postfix, and infix expressions
 - Syntax parsing
 - String reversal
 - Parenthesis checking
 - Backtracking
-

10. What is a Dequeue?*Ans:*

It is a double-ended queue, or a data structure, where the elements can be inserted or deleted at both ends (FRONT and REAR).

UNIT II

Trees: Definitions and Properties, Representation of Binary Trees, Operations, Binary Tree Traversal. Binary Search Trees: Definitions, Operations on Binary Search Trees. Balanced Search Trees: AVL Trees, and B-Trees

2.1 TREES

2.1.1 Definitions and Properties Representation of Binary Trees

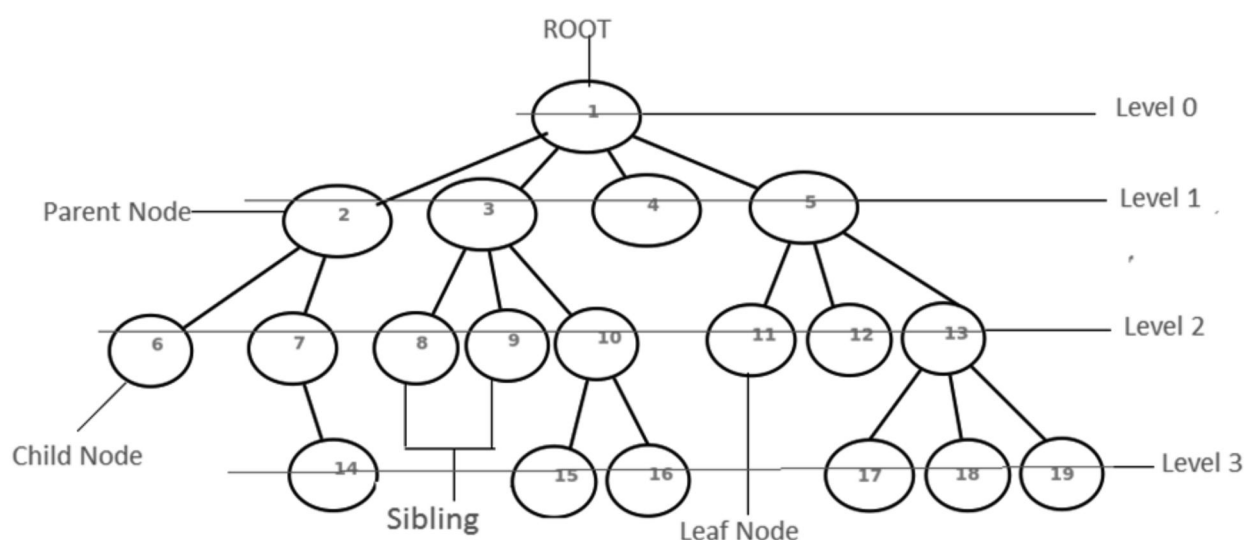
Q1. What is a Tree data structure and How to represent a Binary tree? Explain its applications

Ans :

(Imp.)

A tree is non-linear and a hierarchical data structure consisting of a collection of nodes such that each node of the tree stores a value and a list of references to other nodes (the "children").

This data structure is a specialized method to organize and store data in the computer to be used more effectively. It consists of a central node, structural nodes, and sub-nodes, which are connected via edges. We can also say that tree data structure has roots, branches, and leaves connected with one another.



Recursive Definition

A tree consists of a root, and zero or more subtrees T_1, T_2, \dots, T_k such that there is an edge from the root of the tree to the root of each subtree.

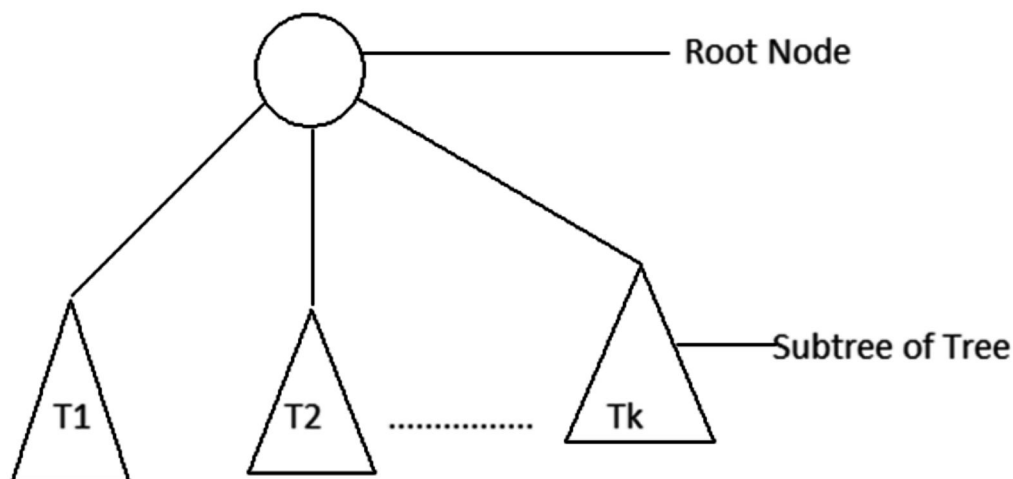


Fig.: Tree is considered a non-linear data structure

The data in a tree are not stored in a sequential manner i.e, they are not stored linearly. Instead, they are arranged on multiple levels or we can say it is a hierarchical structure. For this reason, the tree is considered to be a non-linear data structure.

Basic Terminologies In Tree Data Structure

- **Parent Node:** The node which is a predecessor of a node is called the parent node of that node. {2} is the parent node of {6, 7}.
- **Child Node:** The node which is the immediate successor of a node is called the child node of that node. Examples: {6, 7} are the child nodes of {2}.
- **Root Node:** The topmost node of a tree or the node which does not have any parent node is called the root node. {1} is the root node of the tree. A non-empty tree must contain exactly one root node and exactly one path from the root to all other nodes of the tree.
- **Leaf Node or External Node:** The nodes which do not have any child nodes are called leaf nodes. {6, 14, 8, 9, 15, 16, 4, 11, 12, 17, 18, 19} are the leaf nodes of the tree.
- **Ancestor of a Node:** Any predecessor nodes on the path of the root to that node are called Ancestors of that node. {1, 2} are the ancestor nodes of the node {7}.
- **Descendant:** Any successor node on the path from the leaf node to that node. {7, 14} are the descendants of the node. {2}.
- **Sibling:** Children of the same parent node are called siblings. {8, 9, 10} are called siblings.
- **Level of a node:** The count of edges on the path from the root node to that node. The root node has level 0.
- **Internal node:** A node with at least one child is called Internal Node.
- **Neighbor of a Node:** Parent or child nodes of that node are called neighbors of that node.
- **Subtree:** Any node of the tree along with its descendant.

Properties of a Tree:**➤ Number of edges**

An edge can be defined as the connection between two nodes. If a tree has N nodes then it will have $(N-1)$ edges. There is only one path from each node to any other node of the tree.

➤ Depth of a node

The depth of a node is defined as the length of the path from the root to that node. Each edge adds 1 unit of length to the path. So, it can also be defined as the number of edges in the path from the root of the tree to the node.

➤ Height of a node

The height of a node can be defined as the length of the longest path from the node to a leaf node of the tree.

➤ Height of the Tree

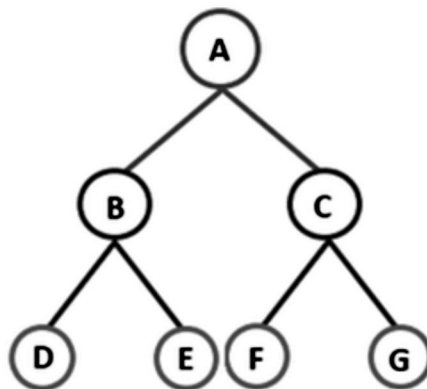
The height of a tree is the length of the longest path from the root of the tree to a leaf node of the tree.

➤ Degree of a Node

The total count of subtrees attached to that node is called the degree of the node. The degree of a leaf node must be 0. The degree of a tree is the maximum degree of a node among all the nodes in the tree.

Some more properties are:

- Traversing in a tree is done by depth first search and breadth first search algorithm.
- It has no loop and no circuit
- It has no self-loop
- Its hierarchical model.

Example of Binary Trees

Here,

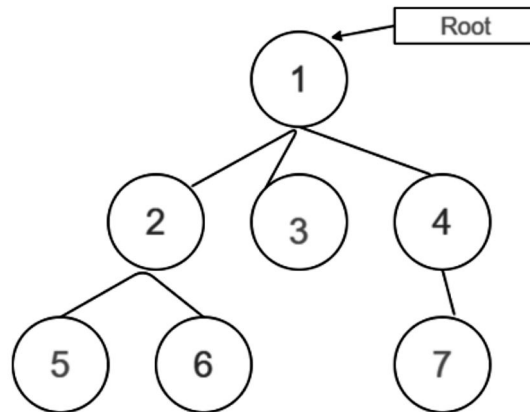
Node A is the root node

B is the parent of D and E

D and E are the siblings

D, E, F and G are the leaf nodes

A and B are the ancestors of E



Types

The different types of tree data structures are as follows:

1. General tree

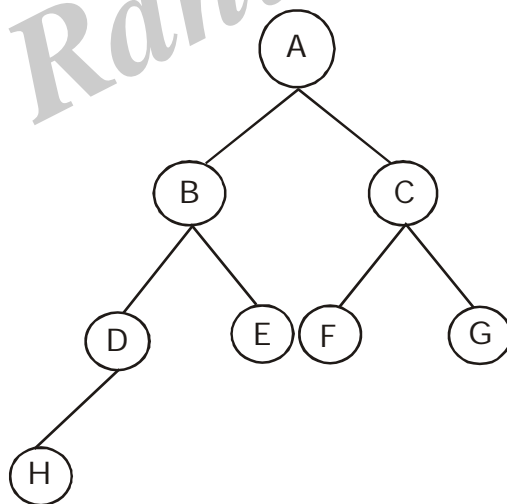
A general tree data structure has no restriction on the number of nodes. It means that a parent node can have any number of child nodes.

2. Binary tree

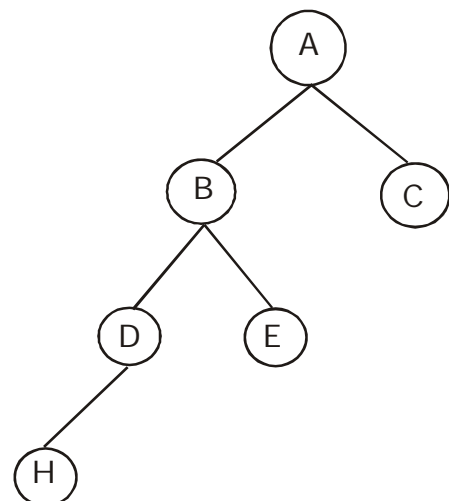
A node of a binary tree can have a maximum of two child nodes. In the given tree diagram, node B, D, and F are left children, while E, C, and G are the right children.

3. Balanced tree

If the height of the left sub-tree and the right sub-tree is equal or differs at most by 1, the tree is known as a balanced tree.



Balanced Tree



Unbalanced Tree

4. Binary search tree

As the name implies, binary search trees are used for various searching and sorting algorithms. The examples include AVL tree and red-black tree. It is a non-linear data structure. It shows that the value of the left node is less than its parent, while the value of the right node is greater than its parent.

➤ **Applications of Tree data structure:**

The applications of tree data structures are as follows:

1. Spanning trees

It is the shortest path tree used in the routers to direct the packets to the destination.

2. Binary Search Tree

It is a type of tree data structure that helps in maintaining a sorted stream of data.

1. Full Binary tree
2. Complete Binary tree
3. Skewed Binary tree
4. Stickily Binary tree
5. Extended Binary tree

3. Storing hierarchical data

Tree data structures are used to store the hierarchical data, which means data is arranged in the form of order.

4. Syntax tree

The syntax tree represents the structure of the program's source code, which is used in compilers.

5. Trie

It is a fast and efficient way for dynamic spell checking. It is also used for locating specific keys from within a set.

6. Heap

It is also a tree data structure that can be represented in a form of an array. It is used to implement priority queues.

2.1.2 Operations

Q2. What are the basic operations of Binary Tree?

Ans :

Basic Operations

The basic operations that can be performed on a binary search tree data structure, are the following

- **Insert** - Inserts an element in a tree/create a tree.
- **Search** - Searches an element in a tree.
- **Preorder Traversal** - Traverses a tree in a pre-order manner.
- **Inorder Traversal** - Traverses a tree in an in-order manner.
- **Postorder Traversal** - Traverses a tree in a post-order manner.

Insert Operation

The very first insertion creates the tree. Afterwards, whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

Algorithm

```
If root is NULL
then create root node
return
If root exists then
compare the data with node.data
while until insertion position is located
If data is greater than node.data
goto right subtree
else
goto left subtree
endwhile
insert data
endif
```

Binary Search Tree Construction

Let us understand the construction of a binary search tree using the following example-

Example

Construct a Binary Search Tree (BST) for the following sequence of numbers-

50, 70, 60, 20, 90, 10, 40, 100

When elements are given in a sequence,

- Always consider the first element as the root node.
- Consider the given elements and insert them in the BST one by one.

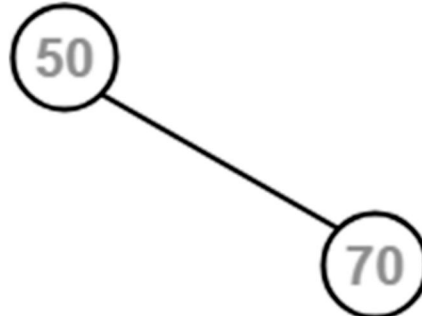
The binary search tree will be constructed as explained below-

Insert 50-

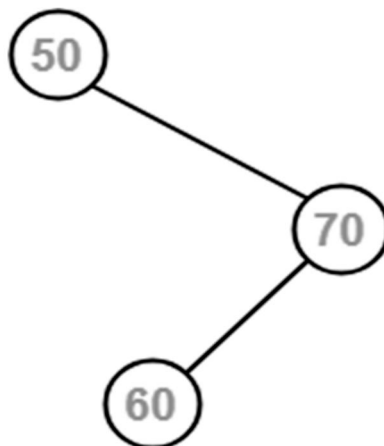
50

Insert 70-

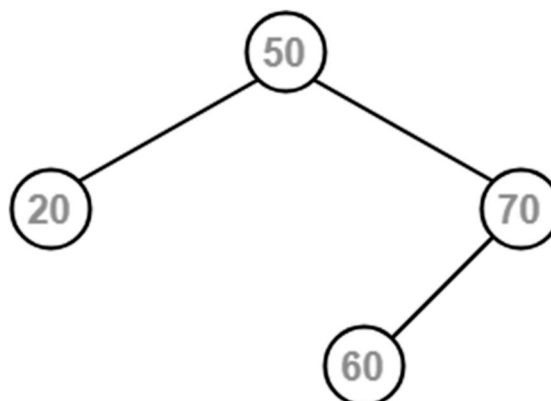
- As $70 > 50$, so insert 70 to the right of 50.

**Insert 60-**

- As $60 > 50$, so insert 60 to the right of 50.
- As $60 < 70$, so insert 60 to the left of 70.

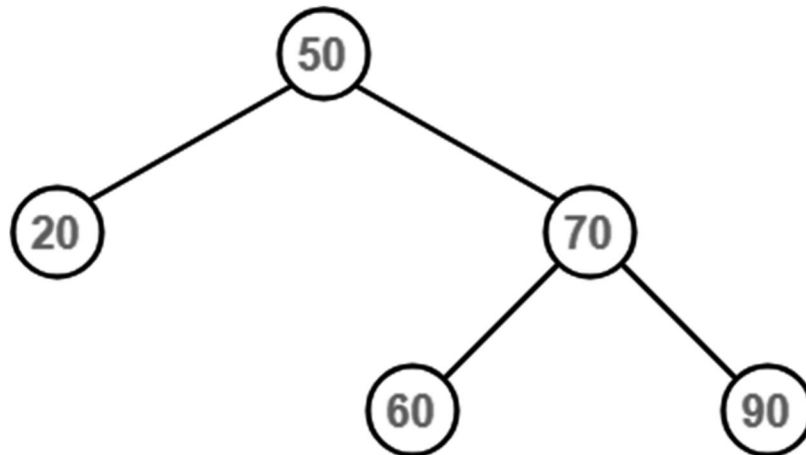
**Insert 20-**

- As $20 < 50$, so insert 20 to the left of 50.

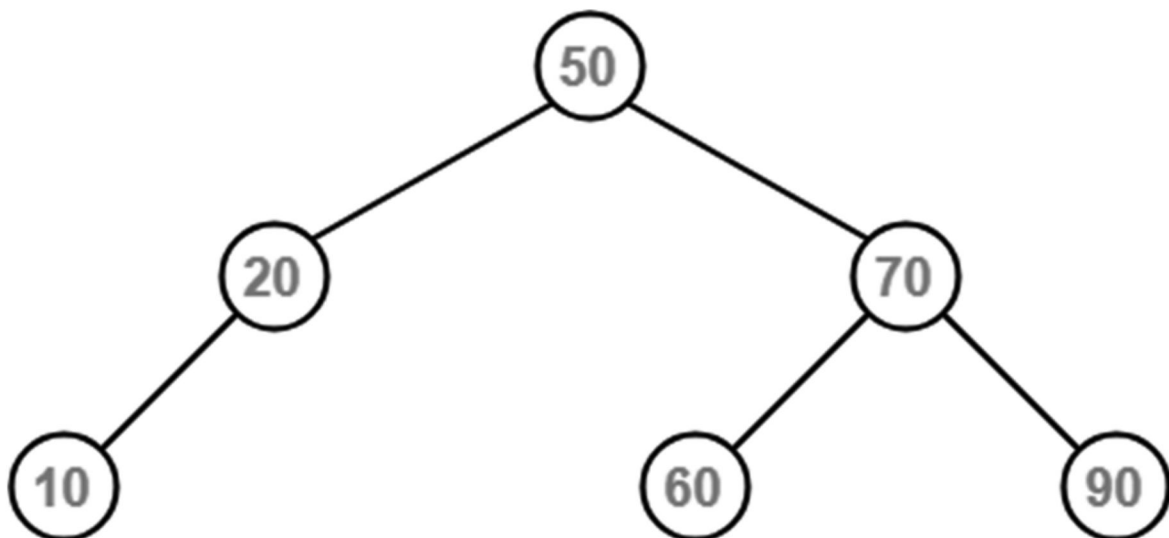


Insert 90-

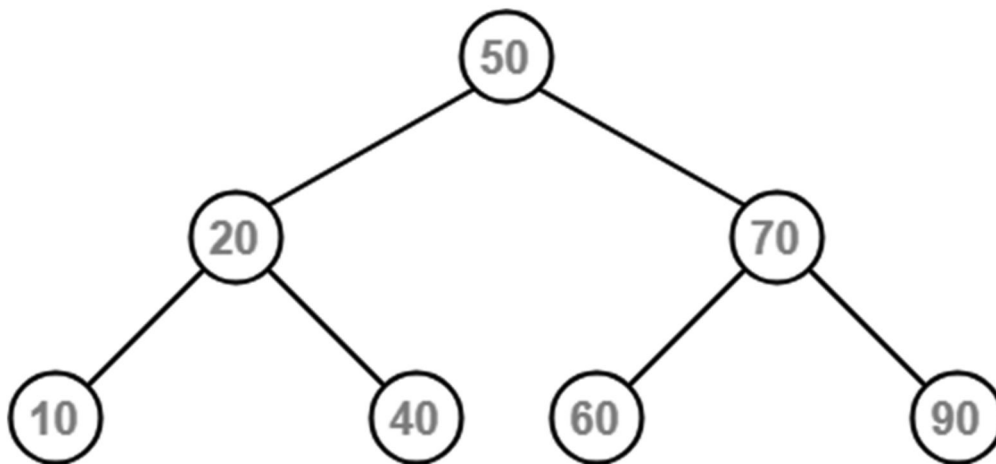
- As $90 > 50$, so insert 90 to the right of 50.
- As $90 > 70$, so insert 90 to the right of 70.

**Insert 10-**

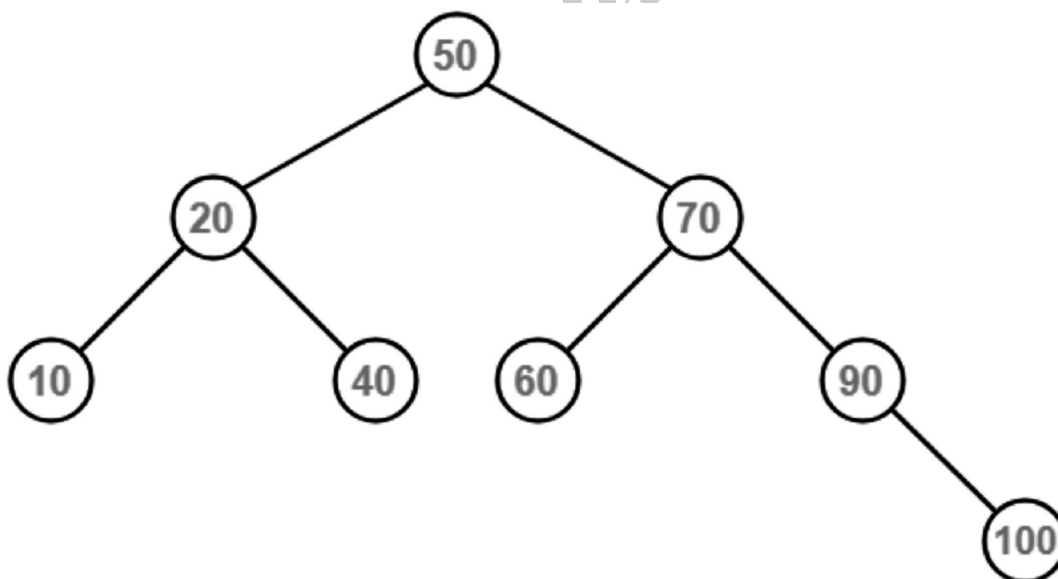
- As $10 < 50$, so insert 10 to the left of 50.
- As $10 < 20$, so insert 10 to the left of 20.

**Insert 40-**

- As $40 < 50$, so insert 40 to the left of 50.
- As $40 > 20$, so insert 40 to the right of 20.

**Insert 100-**

- As $100 > 50$, so insert 100 to the right of 50.
- As $100 > 70$, so insert 100 to the right of 70.
- As $100 > 90$, so insert 100 to the right of 90.

**Fig.: Binary Search Tree**

This is the required Binary Search Tree.

Search Operation

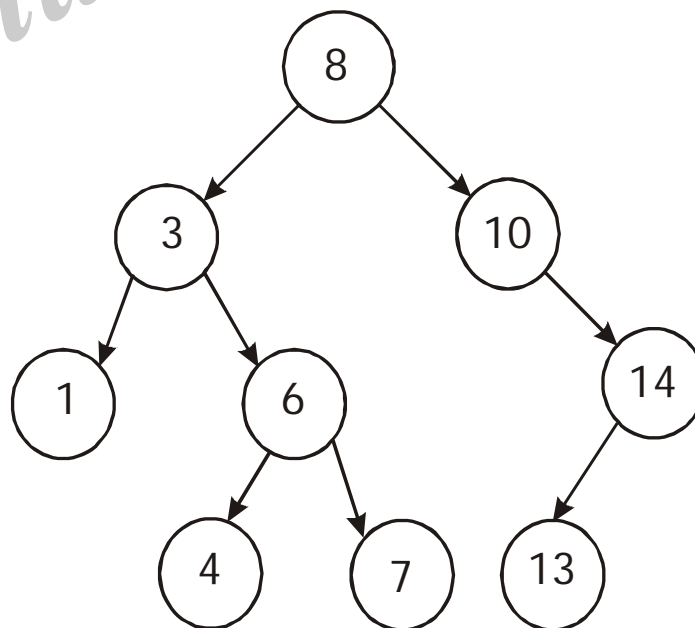
Whenever an element is to be searched, start searching from the root node, then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

Algorithm

```
If root.data is equal to search.data
return root
else
while data not found
If data is greater than node.data
goto right subtree
else
goto left subtree
If data found
return node
endwhile
return data not found
endif
```

Illustration to search 6 in below tree:

1. Start from the root.
2. Compare the searching element with root, if less than root, then recursively call left subtree, else recursively call right subtree.
3. If the element to search is found anywhere, return true, else return false.



2.1.3 Binary Tree Traversal

Q3. Explain in detail about Binary Tree Traversal Technique?

Ans :

(Imp.)

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree -

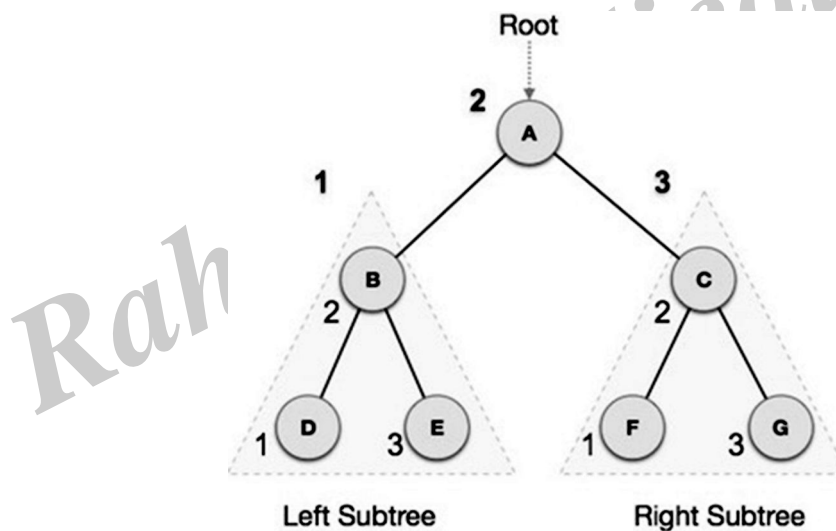
1. In-order Traversal
2. Pre-order Traversal
3. Post-order Traversal

Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

1. In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

If a binary tree is traversed in-order, the output will produce sorted key values in an ascending order.



We start from A, and following in-order traversal, we move to its left subtree B. B is also traversed in-order. The process goes on until all the nodes are visited. The output of in-order traversal of this tree will be-

$D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$

Algorithm

Until all nodes are traversed -

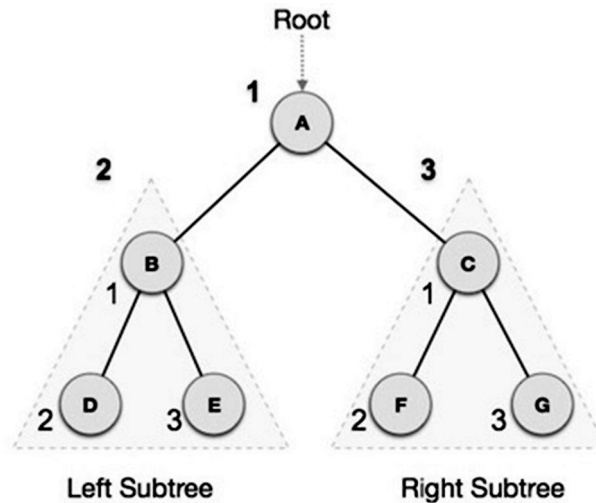
Step 1 - Recursively traverse left subtree.

Step 2 - Visit root node.

Step 3 - Recursively traverse right subtree.

2. Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



We start from A, and following pre-order traversal, we first visit A itself and then move to its left subtree B. B is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be -

A → B → D → E → C → F → G

Algorithm

Until all nodes are traversed -

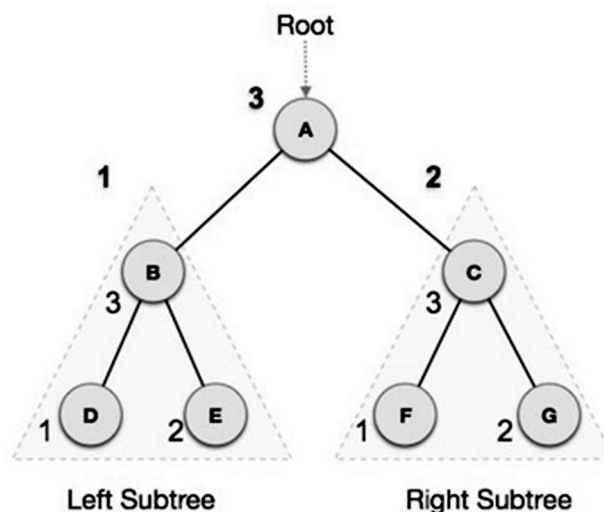
Step 1 - Visit root node.

Step 2 - Recursively traverse left subtree.

Step 3 - Recursively traverse right subtree.

3. Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



We start from A, and following Post-order traversal, we first visit the left subtree B. B is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be -

$D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$

Algorithm

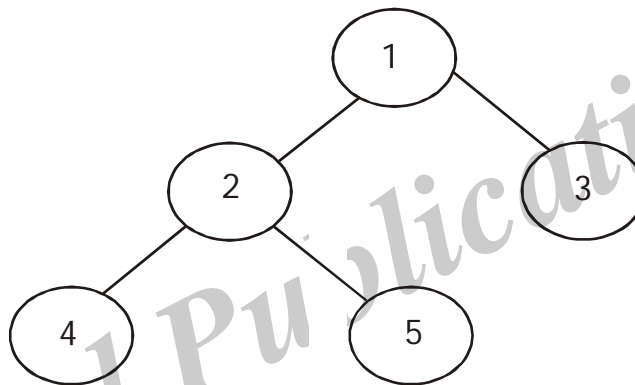
Until all nodes are traversed -

Step 1 - Recursively traverse left subtree.

Step 2 - Recursively traverse right subtree.

Step 3 - Visit root node.

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees.



Depth First Traversals:

(a) Inorder (Left, Root, Right) : 4 2 5 1 3

(b) Preorder (Root, Left, Right) : 1 2 4 5 3

(c) Postorder (Left, Right, Root) : 4 5 2 3 1

Breadth-First or Level Order Traversal: 1 2 3 4 5

2.2 BINARY SEARCH TREES

2.2.1 Definitions, Operations on Binary Search Trees.

Q4. Define a Binary Search Tree? And Explain its Operations.

Ans :

(Imp.)

Definitions:

Tree:

A tree is a kind of data structure that is used to represent the data in hierarchical form. It can be defined as a collection of objects or entities called as nodes that are linked together to simulate a hierarchy. Tree is a non-linear data structure as the data in a tree is not stored linearly or sequentially.

Binary Search tree:

A binary search tree follows some order to arrange the elements. In a Binary search tree, the value of left node must be smaller than the parent node, and the value of right node must be greater than the parent node. This rule is applied recursively to the left and right subtrees of the root.

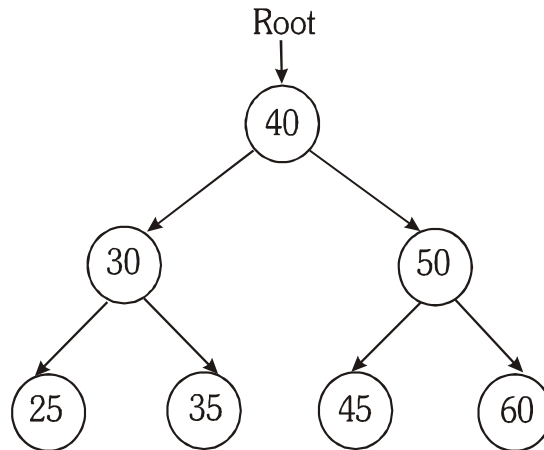
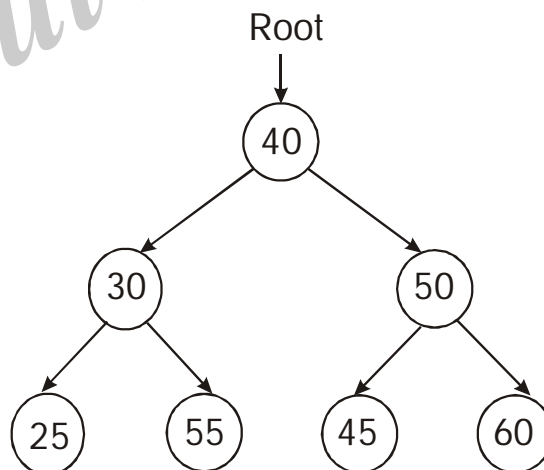


Fig.: Binary search tree with an example.

In the above figure, we can observe that the root node is 40, and all the nodes of the left subtree are smaller than the root node, and all the nodes of the right subtree are greater than the root node.

Similarly, we can see the left child of root node is greater than its left child and smaller than its right child. So, it also satisfies the property of binary search tree. Therefore, we can say that the tree in the above image is a binary search tree.

Suppose if we change the value of node 35 to 55 in the above tree, check whether the tree will be binary search tree or not.



In the above tree, the value of root node is 40, which is greater than its left child 30 but smaller than right child of 30, i.e., 55. So, the above tree does not satisfy the property of Binary search tree. Therefore, the above tree is not a binary search tree.

Advantages of Binary search tree

- Searching an element in the Binary search tree is easy as we always have a hint that which subtree has the desired element.

- As compared to array and linked lists, insertion and deletion operations are faster in BST.

Binary Search Tree Operations

The basic operations that can be performed on a binary search tree data structure, are the following

- Insert d Inserts an element in a tree/create a tree.
- Search d Searches an element in a tree.

Insert Operation

The very first insertion creates the tree. Afterwards, whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

Algorithm

If root is NULL

then create root node

return

If root exists then

compare the data with node.data

while until insertion position is located

If data is greater than node.data

goto right subtree

else

goto left subtree

endwhile

insert data

endif

Search Operation

Whenever an element is to be searched, start searching from the root node, then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

Algorithm

If root.data is equal to search.data

return root

else

while data not found

If data is greater than node.data

goto right subtree

else

goto left subtree

If data found

return node

endwhile

return data not found

endif

2.3 BALANCED SEARCH TREES

2.3.1 AVL Trees

Q5. What is AVL tree? Explain its operations.

Ans :

(Imp.)

AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honour of its inventors.

AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.

Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.

Balance Factor (k) = height (left(k)) - height (right(k))

- If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.
- If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.
- If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.

An AVL tree is given in the following figure. We can see that, balance factor associated with each node is in between -1 and +1. therefore, it is an example of AVL tree.

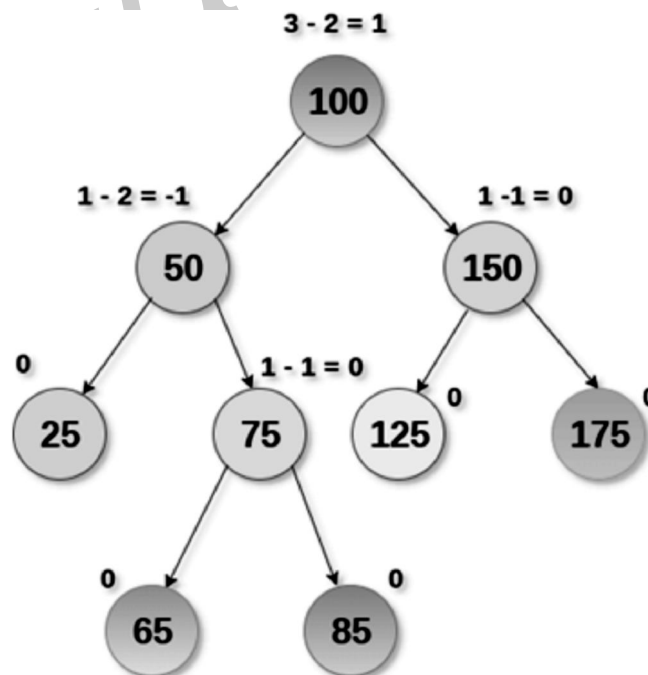


Fig.: AVL Tree

Complexity

Algorithm	Average case	Worst case
Space	$O(n)$	$O(n)$
Search	$O(\log n)$	$O(\log n)$
Insert	$O(\log n)$	$O(\log n)$
Delete	$O(\log n)$	$O(\log n)$

Operations on AVL tree

AVL tree is also a binary search tree therefore, all the operations are performed in the same way as they are performed in a binary search tree. Searching and traversing do not lead to the violation in property of AVL tree. However, insertion and deletion are the operations which can violate this property and therefore, they need to be revisited.

S.No.	Operation	Description
1	Insertion	Insertion in AVL tree is performed in the same way as it is performed in a binary search tree. However, it may lead to violation in the AVL tree property and therefore the tree may need balancing. The tree can be balanced by applying rotations.
2	Deletion	Deletion can also be performed in the same way as it is performed in a binary search tree. Deletion may also disturb the balance of the tree therefore, various types of rotations are used to rebalance the tree.

Use of AVL Tree is:

AVL tree controls the height of the binary search tree by not letting it to be skewed. The time taken for all operations in a binary search tree of height h is $O(h)$. However, it can be extended to $O(n)$ if the BST becomes skewed (i.e. worst case). By limiting this height to $\log n$, AVL tree imposes an upper bound on each operation to be $O(\log n)$ where n is the number of nodes.

Q6. Explain in detail about AVL Rotations*Ans :***AVL Rotations**

AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.

Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.

We perform rotation in AVL tree only in case if Balance Factor is other than -1, 0, and 1. There are basically four types of rotations which are as follows:

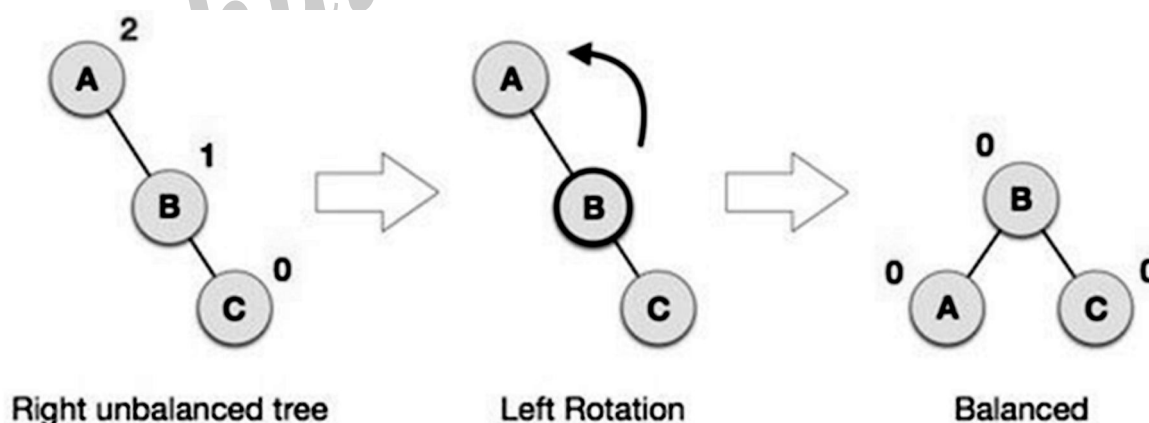
1. L L rotation: Inserted node is in the left subtree of left subtree of A
2. R R rotation : Inserted node is in the right subtree of right subtree of A
3. L R rotation : Inserted node is in the right subtree of left subtree of A
4. R L rotation : Inserted node is in the left subtree of right subtree of A

Where node A is the node whose balance Factor is other than -1, 0, 1.

The first two rotations LL and RR are single rotations and the next two rotations LR and RL are double rotations. For a tree to be unbalanced, minimum height must be at least 2, Let us understand each rotation.

1. RR Rotation

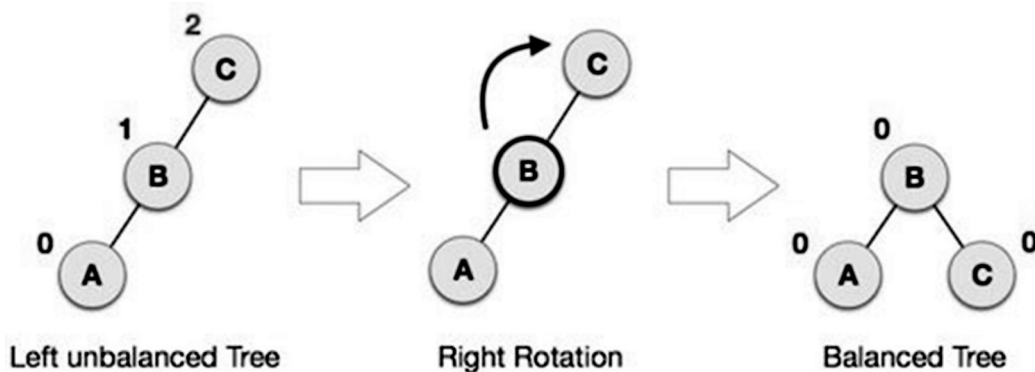
When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation, RR rotation is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2



In above example, node A has balance factor -2 because a node C is inserted in the right subtree of A right subtree. We perform the RR rotation on the edge below A.

2. LL Rotation

When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of C, then we perform LL rotation, LL rotation is clockwise rotation, which is applied on the edge below a node having balance factor 2.



In above example, node C has balance factor 2 because a node A is inserted in the left subtree of C left subtree. We perform the LL rotation on the edge below A.

3. LR Rotation

Double rotations are bit tougher than single rotation which has already explained above. LR rotation = RR rotation + LL rotation, i.e., first RR rotation is performed on subtree and then LL rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.


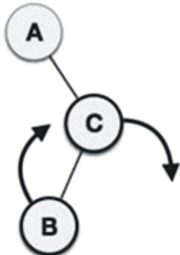
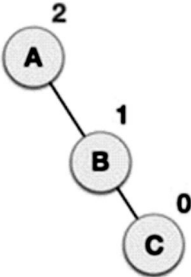
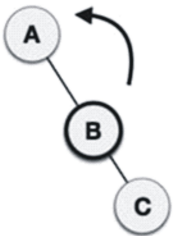
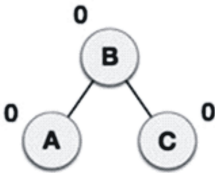
Let us understand each and every step very clearly:

State	Action
	A node B has been inserted into the right subtree of A the left subtree of C, because of which C has become an unbalanced node having balance factor 2. This case is L R rotation where: Inserted node is in the right subtree of left subtree of C
	As LR rotation = RR + LL rotation, hence RR (anticlockwise) on subtree rooted at A is performed first. By doing RR rotation, node A, has become the left subtree of B.
	After performing RR rotation, node C is still unbalanced, i.e., having balance factor 2, as inserted node A is in the left of left of C
	Now we perform LL clockwise rotation on full tree, i.e. on node C. node C has now become the right subtree of node B, A is left subtree of B
	Balance factor of each node is now either -1, 0, or 1, i.e. BST is balanced now.

4. RL Rotation

As already discussed, that double rotations are bit tougher than single rotation which has already explained above. R L rotation

= LL rotation + RR rotation, i.e., first LL rotation is performed on subtree and then RR rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

State	Action
	<p>A node B has been inserted into the left subtree of C the right subtree of A, because of which A has become an unbalanced node having balance factor - 2. This case is RL rotation where: Inserted node is in the left subtree of right subtree of A</p>
	<p>As RL rotation = LL rotation + RR rotation, hence, LL (clockwise) on subtree rooted at C is performed first. By doing RR rotation, node C has become the right subtree of B.</p>
	<p>After performing LL rotation, node A is still unbalanced, i.e. having balance factor -2, which is because of the right-subtree of the right-subtree node A.</p>
	<p>Now we perform RR rotation (anticlockwise rotation) on full tree, i.e. on node A. node C has now become the right subtree of node B, and node A has become the left subtree of B.</p>
	<p>Balance factor of each node is now either -1, 0, or 1, i.e., BST is balanced now.</p>

Q7. Give an example how to construct an AVL Tree?

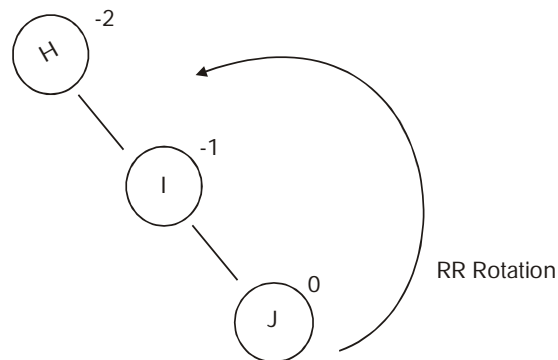
Ans :

(Imp.)

Construct an AVL tree having the following elements

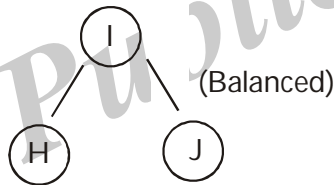
H, I, J, B, A, E, C, F, D, G, K, L

1. Insert H, I, J

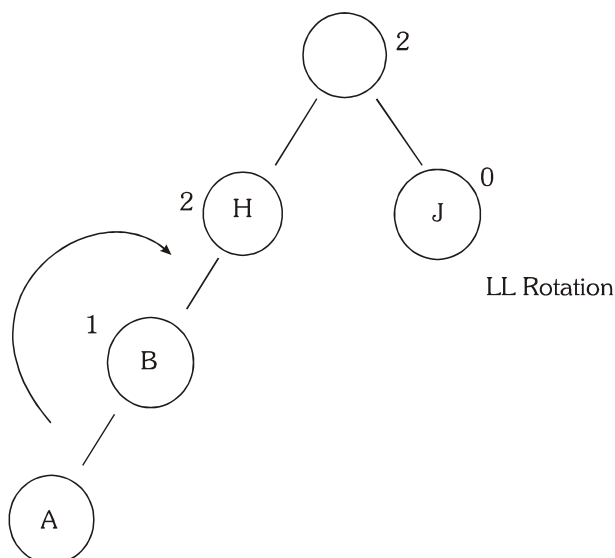


On inserting the above elements, especially in the case of H, the BST becomes unbalanced as the Balance Factor of H is -2. Since the BST is right-skewed, we will perform RR Rotation on node H.

The resultant balance tree is:

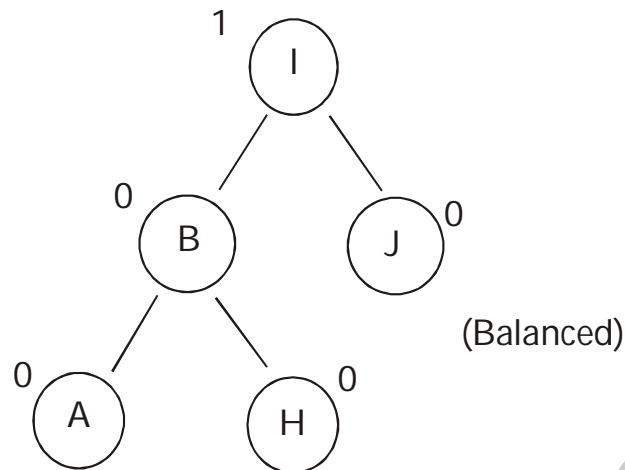


2. Insert B, A

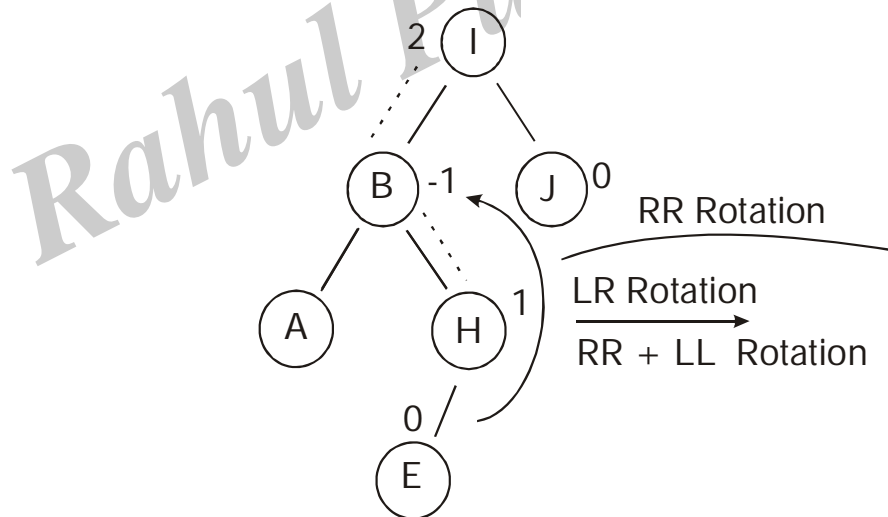


On inserting the above elements, especially in case of A, the BST becomes unbalanced as the Balance Factor of H and I is 2, we consider the first node from the last inserted node i.e. H. Since the BST from H is left-skewed, we will perform LL Rotation on node H.

The resultant balance tree is:



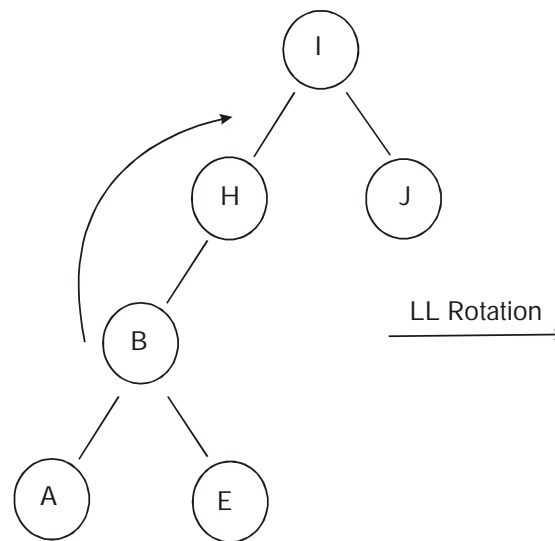
3. Insert E



On inserting E, BST becomes unbalanced as the Balance Factor of I is 2, since if we travel from E to I we find that it is inserted in the left subtree of right subtree of I, we will perform LR Rotation on node I.
LR = RR + LL rotation

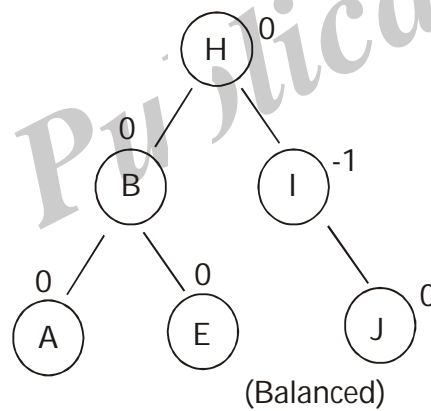
3a) We first perform RR rotation on node B

The resultant tree after RR rotation is:

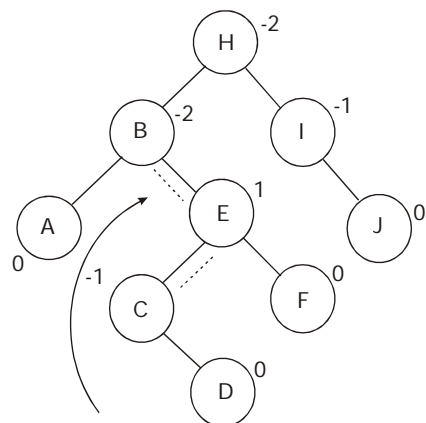


3b) We first perform LL rotation on the node I

The resultant balanced tree after LL rotation is:



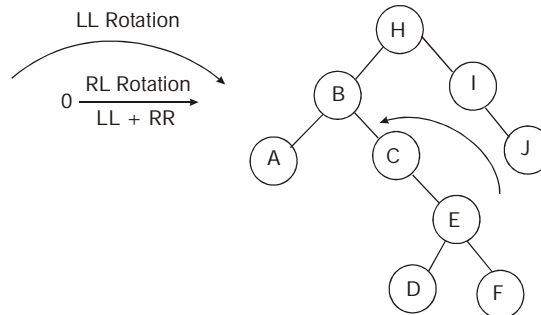
4. Insert C, F, D



On inserting C, F, D, BST becomes unbalanced as the Balance Factor of B and H is -2, since if we travel from D to B we find that it is inserted in the right subtree of left subtree of B, we will perform RL Rotation on node I. RL = LL + RR rotation.

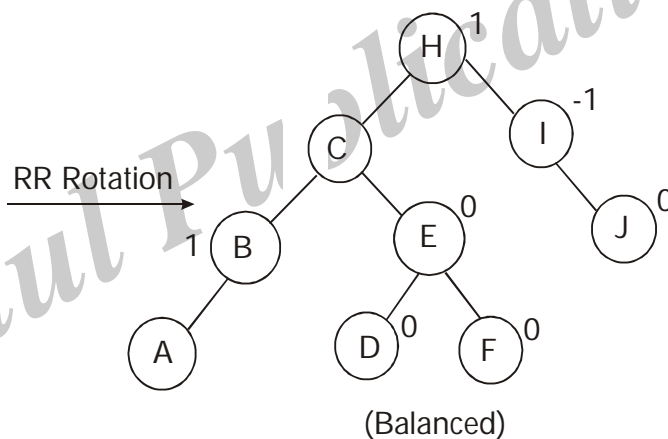
4a) We first perform LL rotation on node E

The resultant tree after LL rotation is:

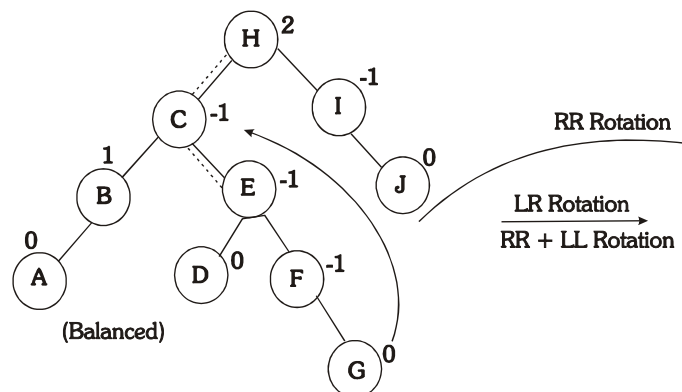


4b) We then perform RR rotation on node B

The resultant balanced tree after RR rotation is:



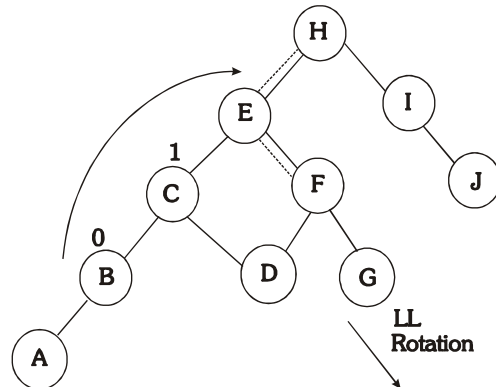
5. Insert G



On inserting G, BST become unbalanced as the Balance Factor of H is 2, since if we travel from G to H, we find that it is inserted in the left subtree of right subtree of H, we will perform LR Rotation on node I. LR = RR + LL rotation.

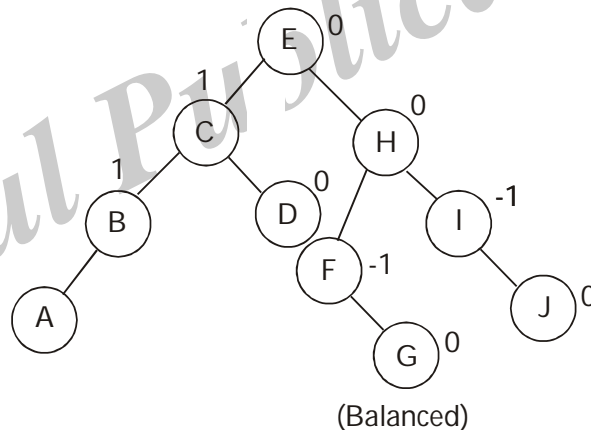
5 a) We first perform RR rotation on node C

The resultant tree after RR rotation is:

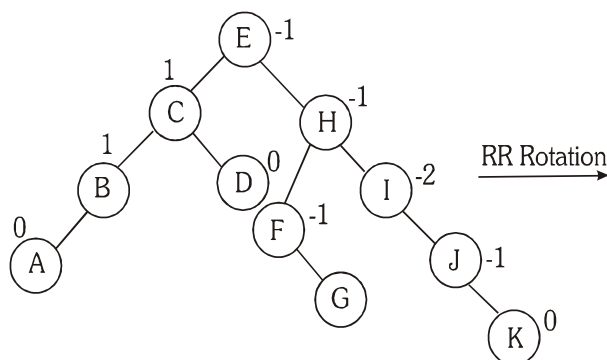


5b) We then perform LL rotation on node H

The resultant balanced tree after LL rotation is:

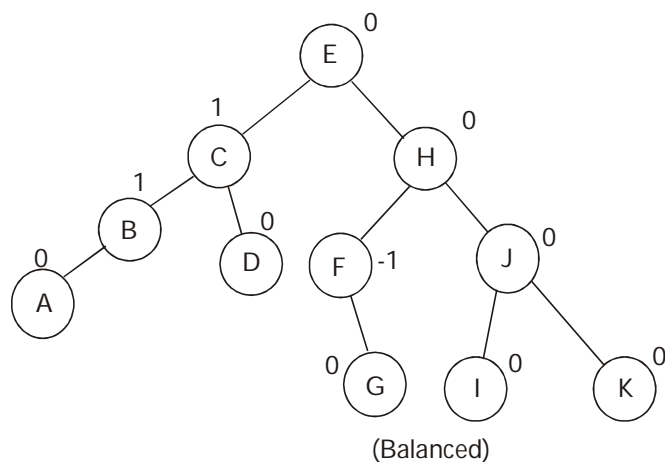


6. Insert K



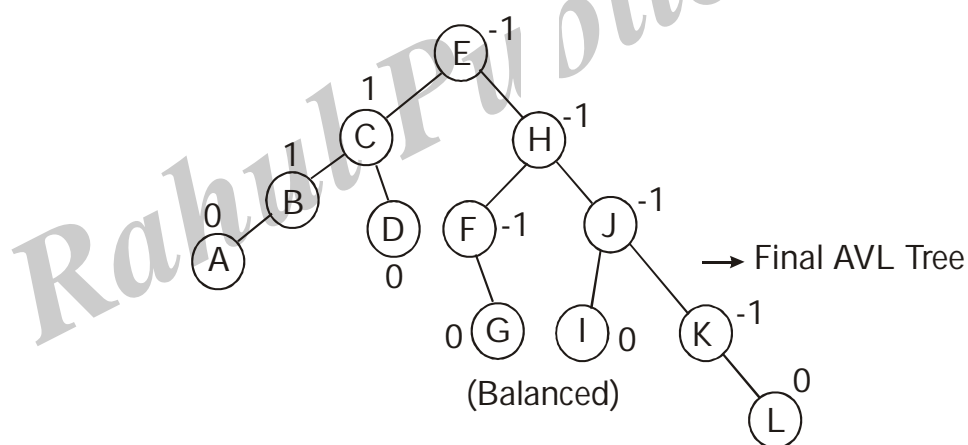
On inserting K, BST becomes unbalanced as the Balance Factor of I is -2. Since the BST is right-skewed from I to K, hence we will perform RR Rotation on the node I.

The resultant balanced tree after RR rotation is:



7. Insert L

On inserting the L tree is still balanced as the Balance Factor of each node is now either, -1, 0, +1. Hence the tree is a Balanced AVL tree



2.3.2 B Trees

Q8. What is B Tree? Explain in detail about its Operations.

Ans :

(Imp.)

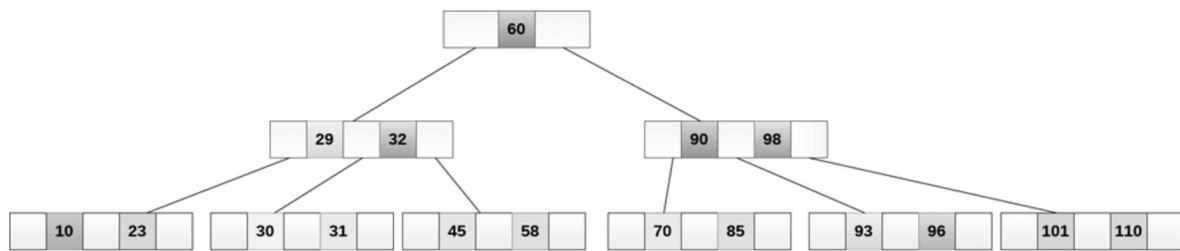
B Tree is a specialized m-way tree that can be widely used for disk access. A B-Tree of order m can have at most m-1 keys and m children. One of the main reasons of using B tree is its capability to store large number of keys in a single node and large key values by keeping the height of the tree relatively small.

A B tree of order m contains all the properties of an M way tree. In addition, it contains the following properties.

1. Every node in a B-Tree contains at most m children.
2. Every node in a B-Tree except the root node and the leaf node contain at least $m/2$ children.
3. The root nodes must have at least 2 nodes.
4. All leaf nodes must be at the same level.

It is not necessary that, all the nodes contain the same number of children but, each node must have $m/2$ number of nodes.

A B tree of order 4 is shown in the following image.



While performing some operations on B Tree, any property of B Tree may violate such as number of minimum children a node can have. To maintain the properties of B Tree, the tree may split or join.

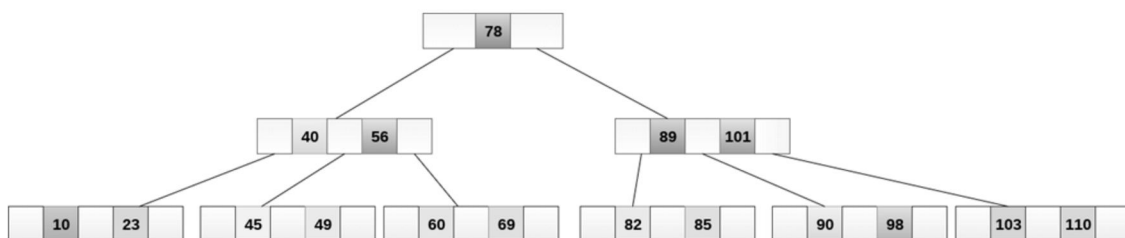
Operations

Searching :

Searching in B Trees is similar to that in Binary search tree. For example, if we search for an item 49 in the following B Tree. The process will something like following:

1. Compare item 49 with root node 78. since $49 < 78$ hence, move to its left sub-tree.
2. Since, $40 < 49 < 56$, traverse right sub-tree of 40.
3. $49 > 45$, move to right. Compare 49.
4. match found, return.

Searching in a B tree depends upon the height of the tree. The search algorithm takes $O(\log n)$ time to search any element in a B tree.



Inserting

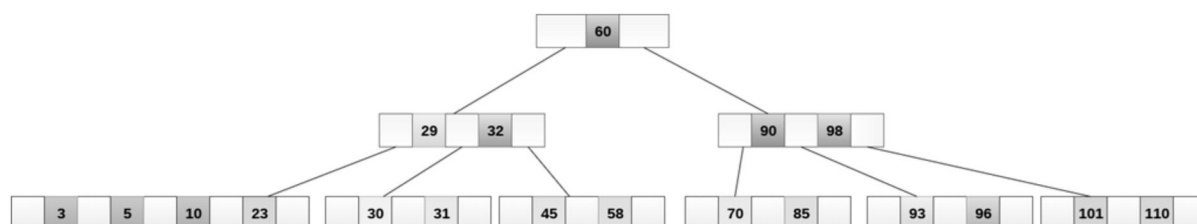
Insertions are done at the leaf node level. The following algorithm needs to be followed in order to insert an item into B Tree.

1. Traverse the B Tree in order to find the appropriate leaf node at which the node can be inserted.
2. If the leaf node contain less than $m-1$ keys then insert the element in the increasing order.
3. Else, if the leaf node contains $m-1$ keys, then follow the following steps.

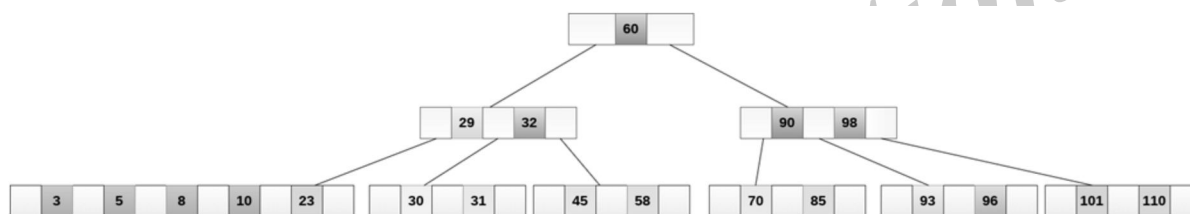
- Insert the new element in the increasing order of elements.
- Split the node into the two nodes at the median.
- Push the median element upto its parent node.
- If the parent node also contain m-1 number of keys, then split it too by following the same steps.

Example:

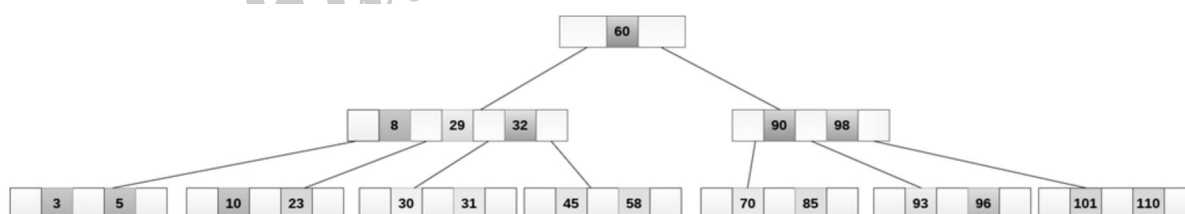
Insert the node 8 into the B Tree of order 5 shown in the following image.



8 will be inserted to the right of 5, therefore insert 8.



The node, now contain 5 keys which is greater than $(5 - 1 = 4)$ keys. Therefore split the node from the median i.e. 8 and push it up to its parent node shown as follows.

**Deletion**

Deletion is also performed at the leaf nodes. The node which is to be deleted can either be a leaf node or an internal node. Following algorithm needs to be followed in order to delete a node from a B tree.

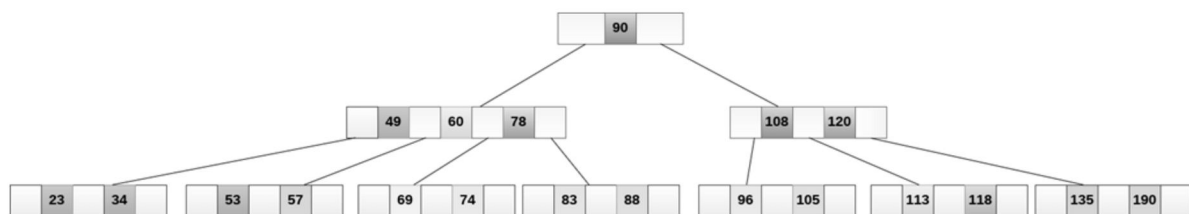
1. Locate the leaf node.
2. If there are more than $m/2$ keys in the leaf node then delete the desired key from the node.
3. If the leaf node doesn't contain $m/2$ keys then complete the keys by taking the element from right or left sibling.
 - If the left sibling contains more than $m/2$ elements then push its largest element up to its parent and move the intervening element down to the node where the key is deleted.

- If the right sibling contains more than $m/2$ elements then push its smallest element up to the parent and move intervening element down to the node where the key is deleted.
- 4. If neither of the sibling contain more than $m/2$ elements then create a new leaf node by joining two leaf nodes and the intervening element of the parent node.
- 5. If parent is left with less than $m/2$ nodes then, apply the above process on the parent too.

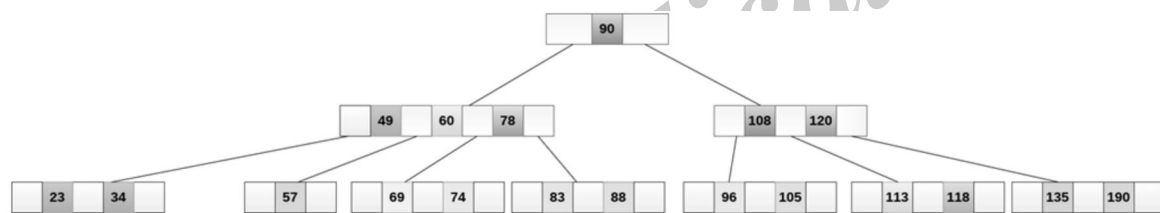
If the node which is to be deleted is an internal node, then replace the node with its in-order successor or predecessor. Since, successor or predecessor will always be on the leaf node hence, the process will be similar as the node is being deleted from the leaf node.

Example 1

Delete the node 53 from the B Tree of order 5 shown in the following figure.

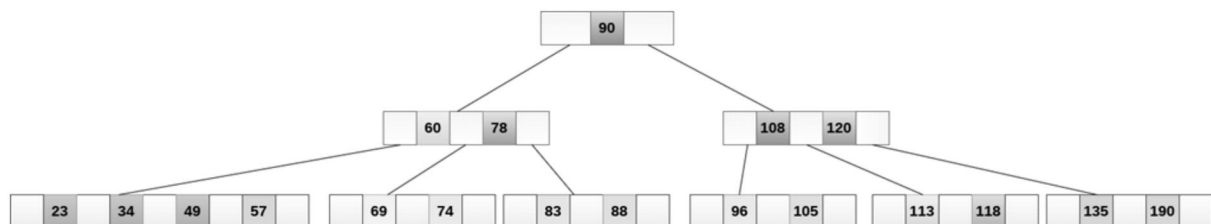


53 is present in the right child of element 49. Delete it.



Now, 57 is the only element which is left in the node, the minimum number of elements that must be present in a B tree of order 5, is 2. It is less than that, the elements in its left and right sub-tree are also not sufficient therefore, merge it with the left sibling and intervening element of parent i.e. 49.

The final B tree is shown as follows.



Application of B tree

B tree is used to index the data and provides fast access to the actual data stored on the disks since, the access to value stored in a large database that is stored on a disk is a very time-consuming process.

Searching an un-indexed and unsorted database containing n key values needs $O(n)$ running time in worst case. However, if we use B Tree to index this database, it will be searched in $O(\log n)$ time in worst case.

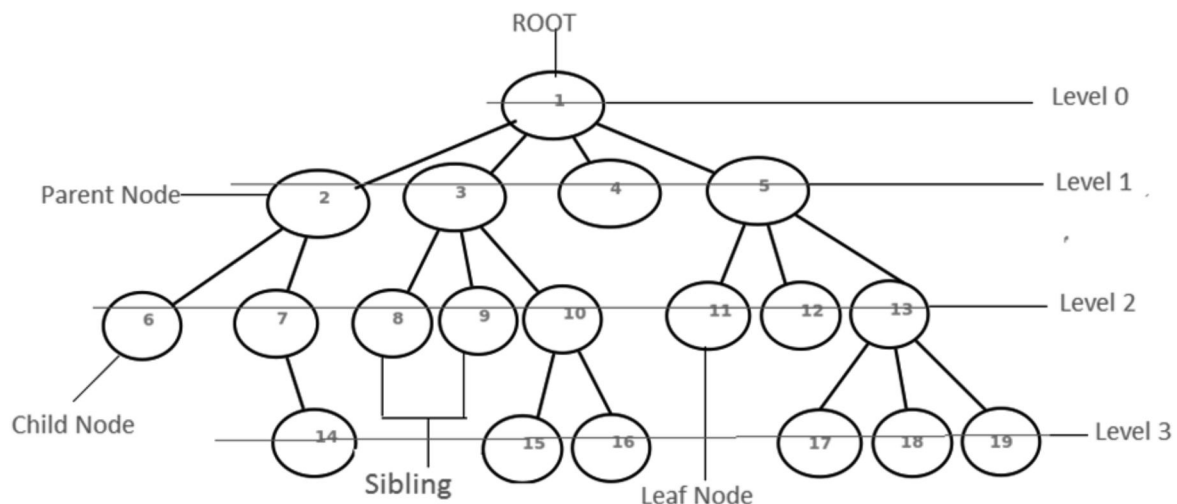
Short Question & Answers

1. What is a Tree data structure and How to represent a Binary tree?

Ans :

A tree is non-linear and a hierarchical data structure consisting of a collection of nodes such that each node of the tree stores a value and a list of references to other nodes (the "children").

This data structure is a specialized method to organize and store data in the computer to be used more effectively. It consists of a central node, structural nodes, and sub-nodes, which are connected via edges. We can also say that tree data structure has roots, branches, and leaves connected with one another.



2. What are the Properties of a BST Tree

Ans :

Properties of a Tree:

➤ Number of edges

An edge can be defined as the connection between two nodes. If a tree has N nodes then it will have $(N-1)$ edges. There is only one path from each node to any other node of the tree.

➤ Depth of a node

The depth of a node is defined as the length of the path from the root to that node. Each edge adds 1 unit of length to the path. So, it can also be defined as the number of edges in the path from the root of the tree to the node.

➤ Height of a node

The height of a node can be defined as the length of the longest path from the node to a leaf node of the tree.

➤ **Height of the Tree**

The height of a tree is the length of the longest path from the root of the tree to a leaf node of the tree.

➤ **Degree of a Node**

The total count of subtrees attached to that node is called the degree of the node. The degree of a leaf node must be 0. The degree of a tree is the maximum degree of a node among all the nodes in the tree.

Some more properties are:

- Traversing in a tree is done by depth first search and breadth first search algorithm.
- It has no loop and no circuit
- It has no self-loop
- Its hierarchical model.

3. What are the Applications of Tree data structure.

Ans :

The applications of tree data structures are as follows:

1. Spanning trees

It is the shortest path tree used in the routers to direct the packets to the destination.

2. Binary Search Tree

It is a type of tree data structure that helps in maintaining a sorted stream of data.

1. Full Binary tree
2. Complete Binary tree
3. Skewed Binary tree
4. Stickily Binary tree
5. Extended Binary tree

3. Storing hierarchical data

Tree data structures are used to store the hierarchical data, which means data is arranged in the form of order.

4. Syntax tree

The syntax tree represents the structure of the program's source code, which is used in compilers.

5. Trie

It is a fast and efficient way for dynamic spell checking. It is also used for locating specific keys from within a set.

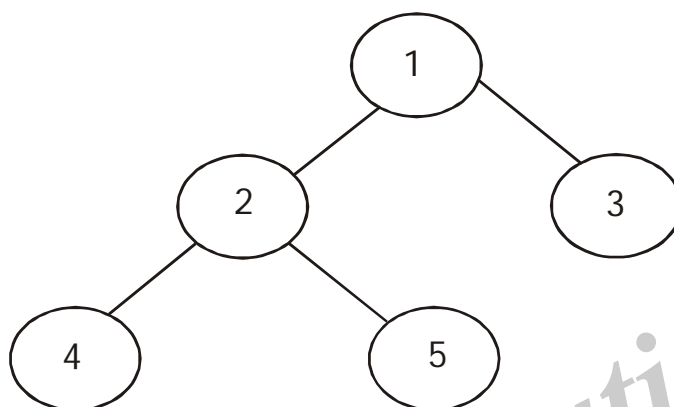
6. Heap

It is also a tree data structure that can be represented in a form of an array. It is used to implement priority queues.

4. Write tree traversal techniques with example.

Ans :

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees.



- (a) Inorder (Left, Root, Right) : 4 2 5 1 3
- (b) Preorder (Root, Left, Right) : 1 2 4 5 3
- (c) Postorder (Left, Right, Root) : 4 5 2 3 1

5. Explain AVL Tree operations.

Ans :

Operations on AVL tree

AVL tree is also a binary search tree therefore, all the operations are performed in the same way as they are performed in a binary search tree. Searching and traversing do not lead to the violation in property of AVL tree. However, insertion and deletion are the operations which can violate this property and therefore, they need to be revisited.

SNo.	Operation	Description
1	Insertion	Insertion in AVL tree is performed in the same way as it is performed in a binary search tree. However, it may lead to violation in the AVL tree property and therefore the tree may need balancing. The tree can be balanced by applying rotations.
2	Deletion	Deletion can also be performed in the same way as it is performed in a binary search tree. Deletion may also disturb the balance of the tree therefore, various types of rotations are used to rebalance the tree.

6. What is B Tree? Explain about its Operations.

Ans :

B Tree is a specialized m-way tree that can be widely used for disk access. A B-Tree of order m can have at most m-1 keys and m children. One of the main reasons of using B tree is its capability to store large number of keys in a single node and large key values by keeping the height of the tree relatively small.

A B tree of order m contains all the properties of an M way tree. In addition, it contains the following properties.

1. Every node in a B-Tree contains at most m children.
2. Every node in a B-Tree except the root node and the leaf node contain at least $m/2$ children.
3. The root nodes must have at least 2 nodes.
4. All leaf nodes must be at the same level.

It is not necessary that, all the nodes contain the same number of children but, each node must have $m/2$ number of nodes.

Operations

Searching: in B Trees searching is similar to Binary search tree

Inserting: Insertions are done at the leaf node level

Deletion: Deletion is also performed at the leaf nodes. The node which is to be deleted can either be a leaf node or an internal node.

7. How can AVL Tree be useful in all the operations as compared to Binary search tree?

Ans :

AVL tree controls the height of the binary search tree by not letting it be skewed. The time taken for all operations in a binary search tree of height h is $O(h)$. However, it can be extended to $O(n)$ if the BST becomes skewed (i.e. worst case). By limiting this height to $\log n$, AVL tree imposes an upper bound on each operation to be $O(\log n)$ where n is the number of nodes.

8. What are the applications of Graph data structure?

Ans :

The graph has the following applications:

- Graphs are used in circuit networks where points of connection are drawn as vertices and component wires become the edges of the graph.
- Graphs are used in transport networks where stations are drawn as vertices and routes become the edges of the graph.
- Graphs are used in maps that draw cities/states/regions as vertices and adjacency relations as edges.
- Graphs are used in program flow analysis where procedures or modules are treated as vertices and calls to these procedures are drawn as edges of the graph.

Q9. What are the various operations that can be performed on different Data Structures?*Ans :*

- **Insertion:** Add a new data item in the given collection of data items.
- **Deletion:** Delete an existing data item from the given collection of data items.
- **Traversal:** Access each data item exactly once so that it can be processed.
- **Searching:** Find out the location of the data item if it exists in the given collection of data items.
- **Sorting:** Arranging the data items in some order i.e. in ascending or descending order in case of numerical data and in dictionary order in case of alphanumeric data.

Q10. What is B Tree?*Ans :*

B Tree is a specialized m-way tree that can be widely used for disk access. A B-Tree of order m can have at most m-1 keys and m children. One of the main reasons of using B tree is its capability to store large number of keys in a single node and large key values by keeping the height of the tree relatively small.

A B tree of order m contains all the properties of an M way tree. In addition, it contains the following properties.

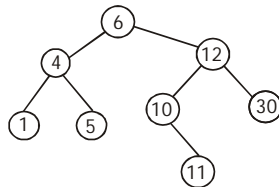
1. Every node in a B-Tree contains at most m children.
2. Every node in a B-Tree except the root node and the leaf node contain at least m/2 children.
3. The root nodes must have at least 2 nodes.
4. All leaf nodes must be at the same level.

It is not necessary that, all the nodes contain the same number of children but, each node must have m/2 number of nodes.

An example of B Tree

Choose the Correct Answers

1. What is the time complexity of the binary search algorithm? [c]
 (a) $O(n)$ (b) $O(1)$
 (c) $O(\log 2n)$ (d) $O(n^2)$
2. Kruskal's Algorithm for finding the Minimum Spanning Tree of a graph is a kind of a? [b]
 (a) DP Problem (b) Greedy Algorithm
 (c) Adhoc Problem (d) None of the above
3. Which of the following statements is false? [b]
 (a) Every tree is a bipartite graph (b) A tree contains a cycle
 (c) A tree with n nodes contains $n-1$ edges (d) A tree is a connected graph
4. If this tree is used for sorting, then new no 8 should be placed as the_____ [d]



- (a) Left child of the node labelled 30
 - (b) Right child of the node labelled 5
 - (c) Right child of the node labelled 30
 - (d) Left child of the node labelled 10
5. Traversing a binary tree first root and then left and right subtrees call _____ traversal. [b]
 (a) Postorder (b) Preorder
 (c) Inorder (d) None of these
6. A complete binary tree with the property that the value at each node is at least as large as the values at its children is called [d]
 (a) Binary search tree (b) Binary Tree
 (c) Completely balanced tree (d) Heap
7. Which algorithm is used in the top tree data structure? [a]
 (a) Divide and Conquer (b) Greedy
 (c) Backtracking (d) Branch
8. The number of edges from the node to the deepest leaf is called _____ of the tree. [a]
 (a) Height (b) Depth
 (c) Length (d) Width
9. What is a full binary tree? [a]
 (a) Each node has exactly zero or two children
 (b) Each node has exactly two children
 (c) All the leaves are at the same level
 (d) Each node has exactly one or two children
10. The number of edges from the root to the node is called _____ of the tree. [b]
 (a) Height (b) Depth
 (c) Length (d) Width

Fill in the Blanks

1. A data structure is a logical method of representing _____ .
2. In the _____ algorithm, the deletion procedure is complex.
3. _____ is the process of visiting every node in a tree at least once.
4. In _____ traversal, the root node is visited last.
5. Children of the same parent are called _____.
6. Nodes which are subtrees of another node are called _____.
7. The values in the left most child of a node must be smaller than the _____ value of that node.
8. The B-tree is derived from _____ trees.
9. In _____ the shortest path can be found.
10. in _____ algorithm, a sorted array of edges is required in order to construct a minimal spanning tree

ANSWERS

1. Data in memory
2. Binary tree
3. Traversal
4. Postorder
5. Siblings.
6. Children.
7. First
8. Multiway search
9. BFS
10. Dijkstra's

One Mark Answers

1. List the types of tree.

Ans:

There are six types of tree given as follows.

- General Tree
- Forests
- Binary Tree
- Binary Search Tree
- Expression Tree
- Tournament Tree

2. What are Binary trees?

Ans:

A binary Tree is a special type of generic tree in which, each node can have at most two children. Binary tree is generally partitioned into three disjoint subsets, i.e. the root of the node, left sub-tree and Right binary sub-tree.

3. List some applications of Tree-data structure?

Ans:

Applications of Tree- data structure:

- The manipulation of Arithmetic expression,
- Symbol Table construction,
- Syntax analysis
- Hierarchal data model

4. Define the graph data structure?

Ans:

A graph G can be defined as an ordered set $G(V, E)$ where $V(G)$ represents the set of vertices and $E(G)$ represents the set of edges which are used to connect these vertices. A graph can be seen as a cyclic tree, where the vertices (Nodes) maintain any complex relationship among them instead of having parent-child relations.

5. Which data structures are used in BFS and DFS algorithm?

Ans:

- In BFS algorithm, Queue data structure is used.
- In DFS algorithm, Stack data structure is used.

6. What are the advantages of Selection Sort?*Ans:*

- It is simple and easy to implement.
 - It can be used for small data sets.
 - It is 60 per cent more efficient than bubble sort.
-

7. List Some Applications of Multilinked Structures?*Ans:*

- Sparse matrix,
 - Index generation.
-

8. What is a leaf node?*Ans:*

Any node in a binary tree or a tree that does not have any children is called a leaf node.

9. What is a root node?*Ans:*

The first node or the top node in a tree is called the root node.

10. What is a Self-Balanced Tree?*Ans:*

Self-balanced binary search trees automatically keep their height as small as possible when operations like insertion and deletion take place.

UNIT III

Graphs: Definitions and Properties, Representation, Graph Search Methods (Depth First Search and Breadth First Search) Application of Graphs: Shortest Path Algorithm (Dijkstra), Minimum Spanning Tree (Prim's and Kruskal's Algorithms).

3.1 GRAPHS

3.1.1 Definitions

Q1. What is Graph? Explain in detail about Graphs.

Ans :

(Imp.)

Meaning

A graph can be defined as group of vertices and edges that are used to connect these vertices. A graph can be seen as a cyclic tree, where the vertices (Nodes) maintain any complex relationship among them instead of having parent child relationship.

Definition

A graph G can be defined as an ordered set $G(V, E)$ where $V(G)$ represents the set of vertices and $E(G)$ represents the set of edges which are used to connect these vertices.

A Graph $G(V, E)$ with 5 vertices (A, B, C, D, E) and six edges ((A,B), (B,C), (C,E), (E,D), (D,B), (D,A)) is shown in the following figure.

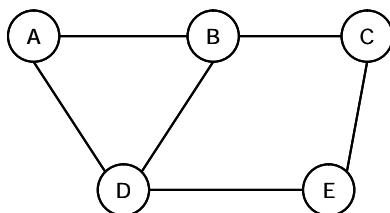


Fig.: Undirected Graph

Directed and Undirected Graph

A graph can be directed or undirected. However, in an undirected graph, edges are not associated with the directions with them. An undirected graph is shown in the above figure since its edges are not attached with any of the directions.

If an edge exists between vertex A and B then the vertices can be traversed from B to A as well as A to B.

In a directed graph, edges form an ordered pair. Edges represent a specific path from some vertex A to another vertex B. Node A is called initial node while node B is called terminal node.

A directed graph is shown in the following figure.

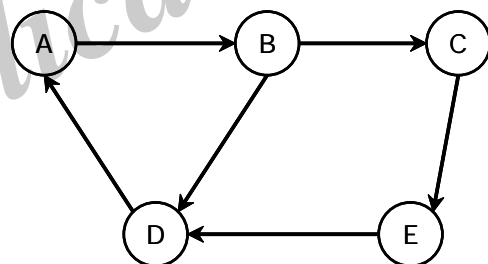


Fig.: Directed Graph

Graph Terminology Definitions

1. Path

A path can be defined as the sequence of nodes that are followed in order to reach some terminal node V from the initial node U .

2. Closed Path

A path will be called as closed path if the initial node is same as terminal node. A path will be closed path if $V_0 = V_N$.

3. Simple Path

If all the nodes of the graph are distinct with an exception $V_0 = V_N$, then such path P is called as closed simple path.

4. Cycle

A cycle can be defined as the path which has no repeated edges or vertices except the first and last vertices.

5. Connected Graph

A connected graph is the one in which some path exists between every two vertices (u, v) in V. There are no isolated nodes in connected graph.

6. Complete Graph

A complete graph is the one in which every node is connected with all other nodes. A complete graph contains $n(n-1)/2$ edges where n is the number of nodes in the graph.

7. Weighted Graph

In a weighted graph, each edge is assigned with some data such as length or weight. The weight of an edge e can be given as $w(e)$ which must be a positive (+) value indicating the cost of traversing the edge.

8. Digraph

A digraph is a directed graph in which each edge of the graph is associated with some direction and the traversing can be done only in the specified direction.

9. Loop

An edge that is associated with the similar end points can be called as Loop.

10. Adjacent Nodes

If two nodes u and v are connected via an edge e, then the nodes u and v are called as neighbors or adjacent nodes.

11. Degree of the Node

A degree of a node is the number of edges that are connected with that node. A node with degree 0 is called as isolated node.

3.1.2 Graph Theory - Basic Properties**Q2. Explain about Graph basic properties in detail with an example.**

Ans : (Imp.)

Graphs come with various properties which are used for characterization of graphs depending

on their structures. These properties are defined in specific terms pertaining to the domain of graph theory. In this chapter, we will discuss a few basic properties that are common in all graphs.

Distance between Two Vertices

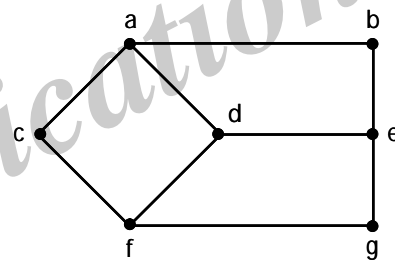
It is number of edges in a shortest path between Vertex U and Vertex V. If there are multiple paths connecting two vertices, then the shortest path is considered as the distance between the two vertices.

Notation - $d(U, V)$

There can be any number of paths present from one vertex to other. Among those, you need to choose only the shortest one.

Example

Take a look at the following graph -



Here, the distance from vertex 'd' to vertex 'e' or simply 'de' is 1 as there is one edge between them. There are many paths from vertex 'd' to vertex 'e' -

- da, ab, be
- df, fg, ge
- de (It is considered for distance between the vertices)
- df, fc, ca, ab, be
- da, ac, cf, fg, ge

Eccentricity of a Vertex

The maximum distance between a vertex to all other vertices is considered as the eccentricity of vertex.

Notation - $e(V)$

The distance from a particular vertex to all other vertices in the graph is taken and among those distances, the eccentricity is the highest of distances.

Example

In the above graph, the eccentricity of 'a' is 3.

The distance from 'a' to 'b' is 1 ('ab'),

from 'a' to 'c' is 1 ('ac'),

from 'a' to 'd' is 1 ('ad'),

from 'a' to 'e' is 2 ('ab'-'be') or ('ad'-'de'),

from 'a' to 'f' is 2 ('ac'-'cf') or ('ad'-'df'),

from 'a' to 'g' is 3 ('ac'-'cf'-'fg') or ('ad'-'df'-'fg').

So the eccentricity is 3, which is a maximum from vertex 'a' from the distance between 'ag' which is maximum.

In other words,

$$e(b) = 3$$

$$e(c) = 3$$

$$e(d) = 2$$

$$e(e) = 3$$

$$e(f) = 3$$

$$e(g) = 3$$

Radius of a Connected Graph

The minimum eccentricity from all the vertices is considered as the radius of the Graph G. The minimum among all the maximum distances between a vertex to all other vertices is considered as the radius of the Graph G.

Notation - $r(G)$

From all the eccentricities of the vertices in a graph, the radius of the connected graph is the minimum of all those eccentricities.

Example

In the above graph $r(G) = 2$, which is the minimum eccentricity for 'd'.

Diameter of a Graph

The maximum eccentricity from all the vertices is considered as the diameter of the Graph G. The maximum among all the distances between a vertex to all other vertices is considered as the diameter of the Graph G.

Notation - $d(G)$: From all the eccentricities of the vertices in a graph, the diameter of the connected graph is the maximum of all those eccentricities.

Example

In the above graph, $d(G) = 3$; which is the maximum eccentricity.

Central Point

If the eccentricity of a graph is equal to its radius, then it is known as the central point of the graph. If

$$e(V) = r(V),$$

then 'V' is the central point of the Graph 'G'.

Example

In the example graph, 'd' is the central point of the graph.

$$e(d) = r(d) = 2$$

Centre

The set of all central points of 'G' is called the centre of the Graph.

Example

In the example graph, {'d'} is the centre of the Graph.

Circumference

The number of edges in the longest cycle of 'G' is called as the circumference of 'G'.

Example

In the example graph, the circumference is 6, which we derived from the longest cycle a-c-f-g-e-b-a or a-c-f-d-e-b-a.

Girth

The number of edges in the shortest cycle of 'G' is called its Girth.

Notation

$$g(G).$$

Example:

In the example graph, the Girth of the graph is 4, which we derived from the shortest cycle a-c-f-d-a or d-f-g-e-d or a-b-e-d-a

3.1.3 Representation of Graphs

Q3. Explain in detail about how to represent a Graph.

Ans :

Graph Representation

A graph is a data structure that consist a set of vertices (called nodes) and edges. There are two ways to store Graphs into the computer's memory:

- Sequential representation (or, Adjacency matrix representation)
- Linked list representation (or, Adjacency list representation)

In sequential representation, an adjacency matrix is used to store the graph. Whereas in linked list representation, there is a use of an adjacency list to store the graph.

Sequential representation

In sequential representation, there is a use of an adjacency matrix to represent the mapping between vertices and edges of the graph. We can use an adjacency matrix to represent the undirected graph, directed graph, weighted directed graph, and weighted undirected graph.

If $\text{adj}[i][j] = w$, it means that there is an edge exists from vertex i to vertex j with weight w .

An entry A_{ij} in the adjacency matrix representation of an undirected graph G will be 1 if an edge exists between V_i and V_j . If an Undirected Graph G consists of n vertices, then the adjacency matrix for that graph is $n \times n$, and the matrix $A = [a_{ij}]$ can be defined as -

$$a_{ij} = 1 \text{ \{if there is a path exists from } V_i \text{ to } V_j\}$$

$$a_{ij} = 0 \text{ \{Otherwise\}}$$

It means that, in an adjacency matrix, 0 represents that there is no association exists between the nodes, whereas 1 represents the existence of a path between two edges.

If there is no self-loop present in the graph, it means that the diagonal entries of the adjacency matrix will be 0.

The adjacency matrix representation of an undirected graph.

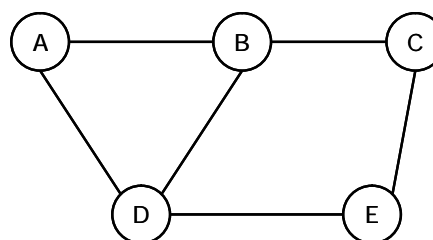


Fig.: Undirected Graph

	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	1	0
C	0	1	0	0	1
D	1	1	0	0	1
E	0	0	1	1	0

Adjacency Matrix

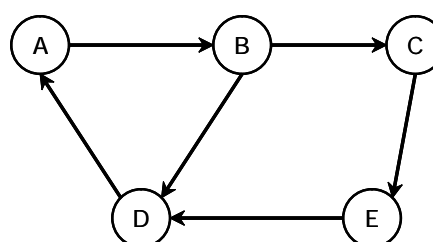
In the above figure, an image shows the mapping among the vertices (A, B, C, D, E), and this mapping is represented by using the adjacency matrix.

There exist different adjacency matrices for the directed and undirected graph. In a directed graph, an entry A_{ij} will be 1 only when there is an edge directed from V_i to V_j .

Adjacency matrix for a directed graph

In a directed graph, edges represent a specific path from one vertex to another vertex. Suppose a path exists from vertex A to another vertex B; it means that node A is the initial node, while node B is the terminal node.

Consider the below-directed graph and try to construct the adjacency matrix of it.



Directed Graph

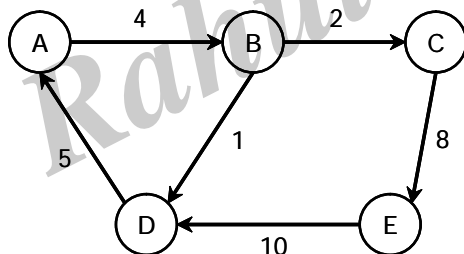
	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	1	0
C	0	0	0	0	1
D	1	0	0	0	0
E	0	0	0	1	0

Adjacency Matrix

In the above graph, we can see there is no self-loop, so the diagonal entries of the adjacent matrix are 0.

Adjacency matrix for a weighted directed graph

It is similar to an adjacency matrix representation of a directed graph except that instead of using the '1' for the existence of a path, here we have to use the weight associated with the edge. The weights on the graph edges will be represented as the entries of the adjacency matrix. We can understand it with the help of an example. Consider the below graph and its adjacency matrix representation. In the representation, we can see that the weight associated with the edges is represented as the entries in the adjacency matrix.

**Fig.: Weighted Directed Graph**

	A	B	C	D	E
A	0	4	0	0	0
B	0	0	2	1	0
C	0	0	0	0	8
D	5	0	0	0	0
E	0	0	0	10	0

Adjacency Matrix

In the above image, we can see that the adjacency matrix representation of the weighted

directed graph is different from other representations. It is because, in this representation, the non-zero values are replaced by the actual weight assigned to the edges.

Adjacency matrix is easier to implement and follow. An adjacency matrix can be used when the graph is dense and a number of edges are large.

Though, it is advantageous to use an adjacency matrix, but it consumes more space. Even if the graph is sparse, the matrix still consumes the same space.

Linked List Representation

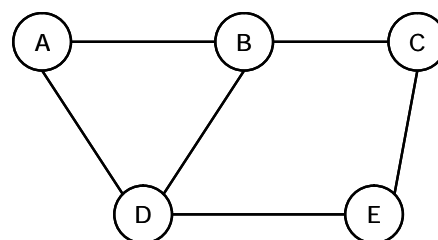
An adjacency list is used in the linked representation to store the Graph in the computer's memory. It is efficient in terms of storage as we only have to store the values for edges.

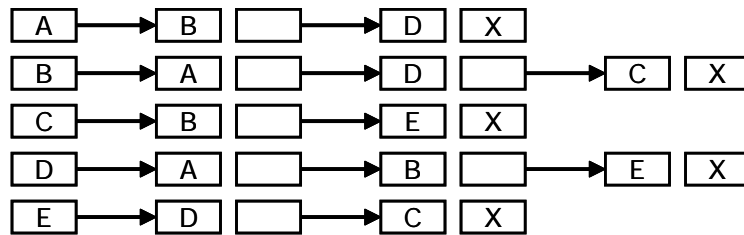
Let's see the adjacency list representation of an undirected graph.

In the above figure, we can see that there is a linked list or adjacency list for every node of the graph. From vertex A, there are paths to vertex B and vertex D. These nodes are linked to nodes A in the given adjacency list.

An adjacency list is maintained for each node present in the graph, which stores the node value and a pointer to the next adjacent node to the respective node. If all the adjacent nodes are traversed, then store the NULL in the pointer field of the last node of the list.

The sum of the lengths of adjacency lists is equal to twice the number of edges present in an undirected graph.

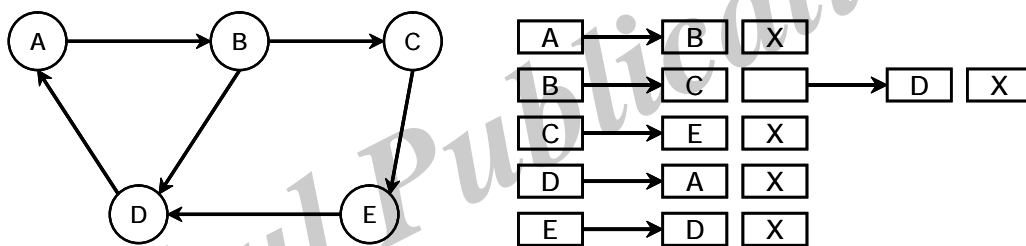
**Fig.: Undirected Graph**

**Adjacency List**

In the above figure, we can see that there is a linked list or adjacency list for every node of the graph. From vertex A, there are paths to vertex B and vertex D. These nodes are linked to nodes A in the given adjacency list.

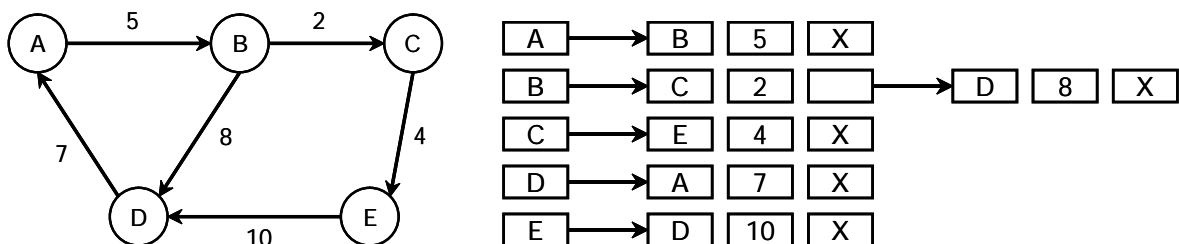
An adjacency list is maintained for each node present in the graph, which stores the node value and a pointer to the next adjacent node to the respective node. If all the adjacent nodes are traversed, then store the NULL in the pointer field of the last node of the list.

The sum of the lengths of adjacency lists is equal to twice the number of edges present in an undirected graph.

**Directed Graph****Adjacency List**

For a directed graph, the sum of the lengths of adjacency lists is equal to the number of edges present in the graph.

Now, consider the weighted directed graph, and let's see the adjacency list representation of that graph.

**Weighted Directed Graph****Adjacency List**

In the case of a weighted directed graph, each node contains an extra field that is called the weight of the node.

In an adjacency list, it is easy to add a vertex. Because of using the linked list, it also saves space.

3.1.4 Graph Search Methods (Depth First Search and Breadth First Search)

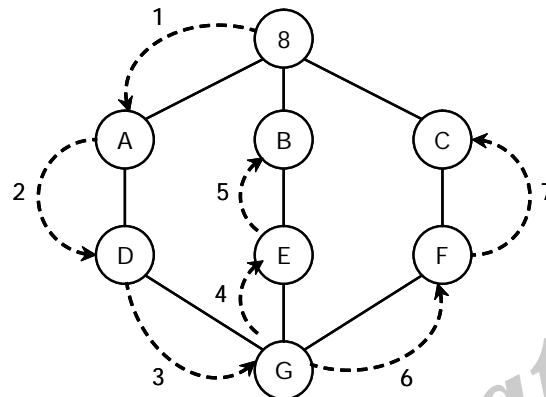
3.1.4.1 Depth First Search (DFS)

Q4. Explain about DFS Search Algorithm.

Ans :

(Imp.)

Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

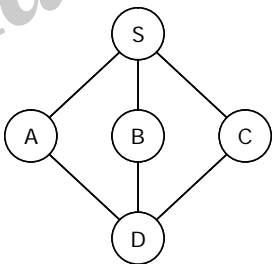
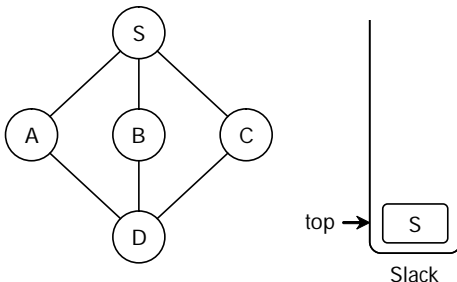


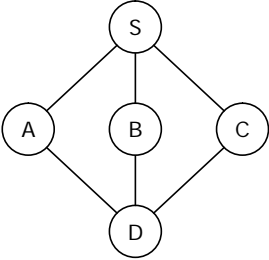
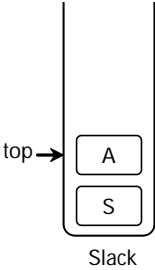
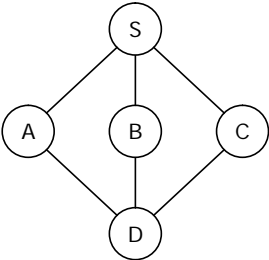
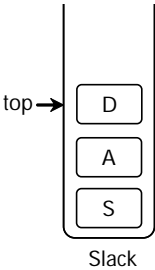
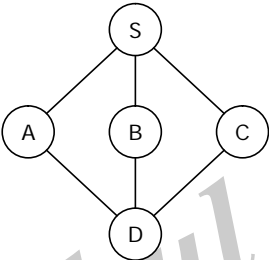
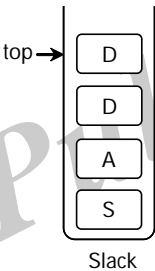
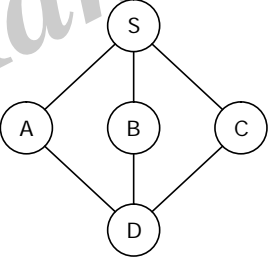
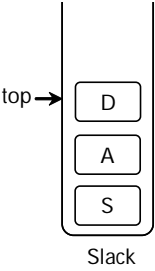
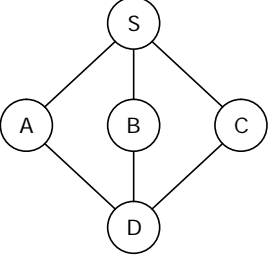
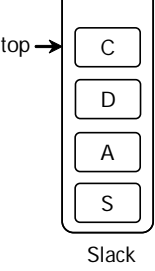
As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.

Rule 1: Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.

Rule 2: If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)

Rule 3: Repeat Rule 1 and Rule 2 until the stack is empty.

Step	Traversal	Description
1.	 Slack	Initialize the slack
2.	 Slack	Mark S as visited and put it onto the stack. Explore any unvisited adjacent node from S. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order

3.	 	Mark A as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both S and D are adjacent to A but we are concerned for unvisited nodes only.
4.	 	Visit D and mark it as visited and put onto the stack. Here, we have B and C nodes, which are adjacent to D and both are unvisited. However, we shall again choose in an alphabetical order.
5.	 	We choose B, mark it as visited and put onto the stack. Here B does not have any unvisited adjacent node. So, we pop B from the stack.
6.	 	We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find D to be on the top of the stack.
7.	 	Only unvisited adjacent node is from D is C now. So we visit C, mark it as visited and put it onto the stack.

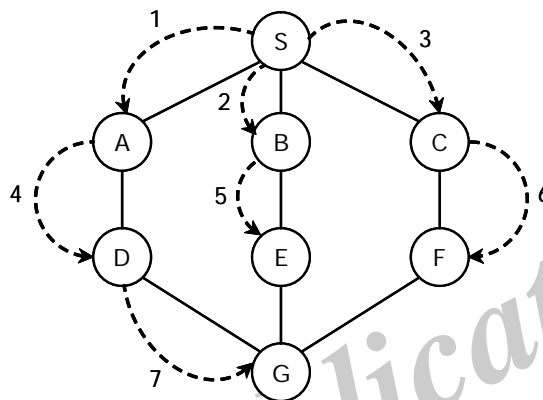
As C does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

3.1.4.2 Breadth First Search (BFS)

Q5. Explain about DFS Search Algorithm.

Ans :

Breadth First Search (BFS) algorithm traverses a graph in a breadth ward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

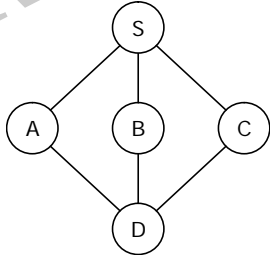
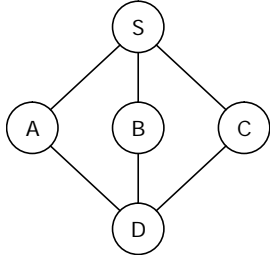


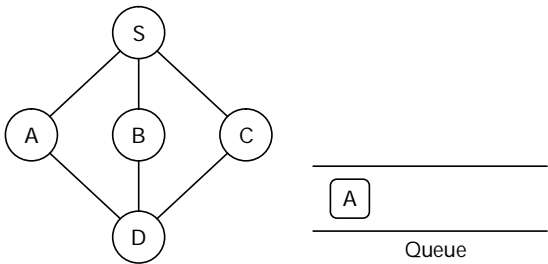
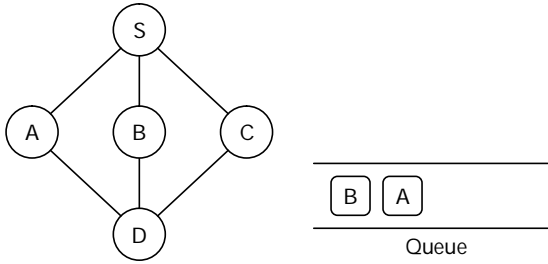
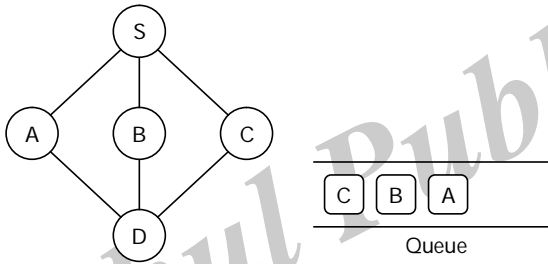
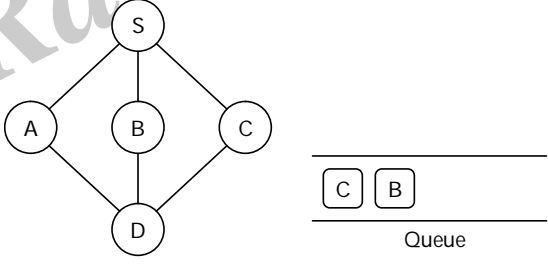
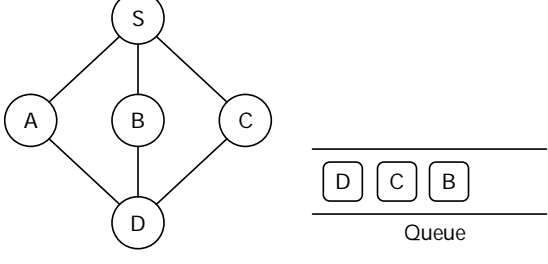
As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

Rule 1: Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.

Rule 2: If no adjacent vertex is found, remove the first vertex from the queue.

Rule 3: Repeat Rule 1 and Rule 2 until the queue is empty.

Step	Traversal	Description
1.	 <div style="display: flex; align-items: center; margin-top: 10px;"> <div style="flex-grow: 1; border-bottom: 1px solid black; margin-bottom: 2px;"></div> <div style="flex-grow: 1; border-bottom: 1px solid black; margin-bottom: 2px;"></div> </div> <div style="text-align: center; margin-top: 5px;">Queue</div>	Initialize the queue.
2.	 <div style="display: flex; align-items: center; margin-top: 10px;"> <div style="flex-grow: 1; border-bottom: 1px solid black; margin-bottom: 2px;"></div> <div style="flex-grow: 1; border-bottom: 1px solid black; margin-bottom: 2px;"></div> </div> <div style="text-align: center; margin-top: 5px;">Queue</div>	We start from visiting S (starting node), and mark it as visited.

3.		<p>We then see an unvisited adjacent node from S. In this example, we have three nodes but alphabetically we choose A, mark it as visited and enqueue it.</p>
4.		<p>Next, the unvisited adjacent node from S is B. We mark it as visited and enqueue it.</p>
5.		<p>Next, the unvisited adjacent node from S is C. We mark it as visited and enqueue it.</p>
6.		<p>Now, S is left with no unvisited adjacent nodes. So, we dequeue and find A.</p>
7.		<p>From A we have D as unvisited adjacent node. We mark it as visited and enqueue it.</p>

At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes.

3.2 APPLICATION OF GRAPHS

3.2.1 Shortest Path Algorithm (Dijkstra)

Q6. Explain about Dijkstra's Shortest Path Algorithm.

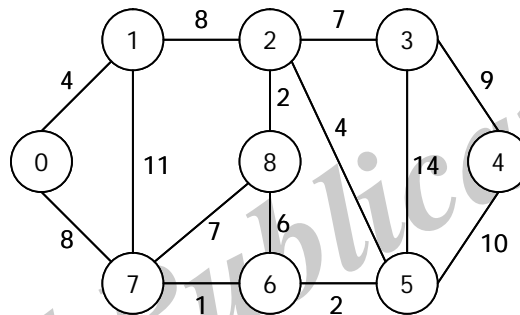
Ans :

(Imp.)

Given a graph and a source vertex in the graph, find the shortest paths from the source to all vertices in the given graph.

Examples:

Input: src = 0, the graph is shown below



Output: 0 4 12 19 21 11 9 8 14

Explanation

The distance from 0 to 1 = 4.

The minimum distance from 0 to 2 = 12. 0 -> 1 -> 2

The minimum distance from 0 to 3 = 19. 0 -> 1 -> 2 -> 3

The minimum distance from 0 to 4 = 21. 0 -> 7 -> 6 -> 5 -> 4

The minimum distance from 0 to 5 = 11. 0 -> 7 -> 6 -> 5

The minimum distance from 0 to 6 = 9. 0 -> 7 -> 6

The minimum distance from 0 to 7 = 8. 0 -> 7

The minimum distance from 0 to 8 = 14. 0 -> 1 -> 2 -> 8

Dijkstra's algorithm is very similar to Prim's algorithm for minimum spanning tree.

Like Prim's MST, generate a SPT (shortest path tree) with a given source as a root. Maintain two sets, one set contains vertices included in the shortest-path tree, other set includes vertices not yet included in the shortest-path tree. At every step of the algorithm, find a vertex that is in the other set (set not yet included) and has a minimum distance from the source.

Follow the steps below to solve the problem:

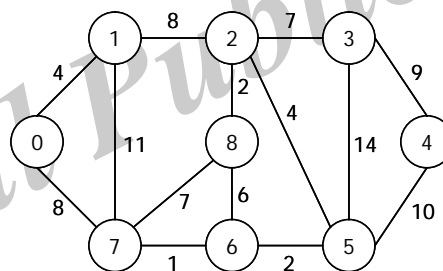
- Create a set sptSet (shortest path tree set) that keeps track of vertices included in the shortest-path tree, i.e., whose minimum distance from the source is calculated and finalized. Initially, this set is empty.
- Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign the distance value as 0 for the source vertex so that it is picked first.
- While sptSet doesn't include all vertices
 - Pick a vertex u which is not there in sptSet and has a minimum distance value.
 - Include u to sptSet.
 - Then update distance value of all adjacent vertices of u .
 - To update the distance values, iterate through all adjacent vertices.
 - For every adjacent vertex v , if the sum of the distance value of u (from source) and weight of edge $u-v$, is less than the distance value of v , then update the distance value of v .

Below is the illustration of the above approach:

Illustration :

To understand the Dijkstra's Algorithm let's take a graph and find the shortest path from source to all nodes.

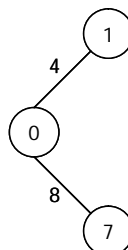
Consider below graph and $src = 0$



Step 1:

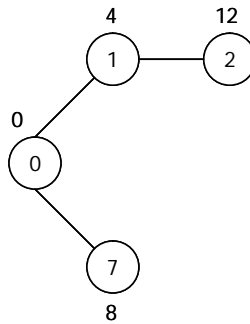
- The set sptSet is initially empty and distances assigned to vertices are $\{0, INF, INF, INF, INF, INF, INF, INF, INF\}$ where INF indicates infinite.
- Now pick the vertex with a minimum distance value. The vertex 0 is picked, include it in sptSet. So sptSet becomes $\{0\}$. After including 0 to sptSet, update distance values of its adjacent vertices.
- Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8.

The following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in green colour.

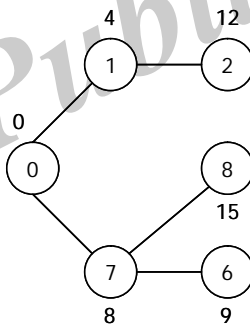


Step 2:

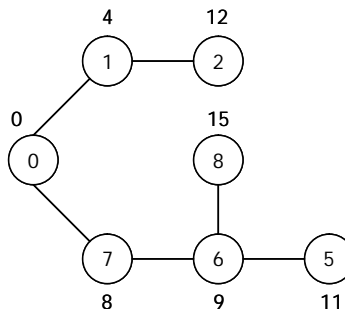
- Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). The vertex 1 is picked and added to sptSet.
- So sptSet now becomes {0, 1}. Update the distance values of adjacent vertices of 1.
- The distance value of vertex 2 becomes 12.

**Step 3:**

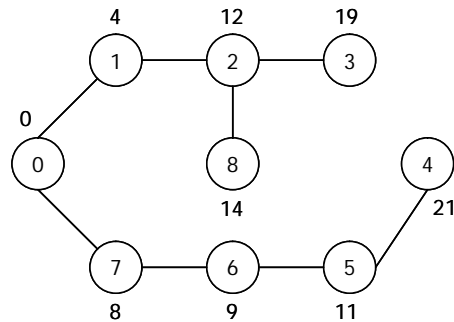
- Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 7 is picked. So sptSet now becomes {0, 1, 7}.
- Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).

**Step 4:**

- Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 6 is picked. So sptSet now becomes {0, 1, 7, 6}.
- Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.



We repeat the above steps until sptSet includes all vertices of the given graph. Finally, we get the following Shortest Path Tree (SPT).



Implementing Dijkstra Algorithm

Given a weighted, undirected and connected graph of V vertices and E edges, Find the shortest distance of all the vertex's from the source vertex S .

Example 1:

Input:

$$V = 2, E = 1$$

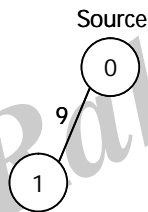
$$u = 0, v = 1, w = 9$$

$$\text{adj} [] = \{\{1, 9\}, \{0, 9\}\}$$

$$S = 0$$

Output:

$$0 \ 9$$



Explanation:

The source vertex is 0. Hence, the shortest distance of node 0 is 0 and the shortest distance from node 9 is $9 - 0 = 9$.

Example 2:

Input:

$$V = 3, E = 3$$

$$u = 0, v = 1, w = 1$$

$$u = 1, v = 2, w = 3$$

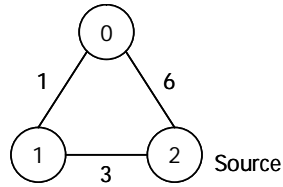
$$u = 0, v = 2, w = 6$$

$$\text{adj} = \{\{1, 1\}, \{2, 6\}\}, \{\{2, 3\}, \{0, 1\}\}, \{\{1, 3\}, \{0, 6\}\}$$

$$S = 2$$

Output:

4 3 0



Explanation:

For nodes 2 to 0, we can follow the path - 2 – 1 – 0. This has a distance of $1 + 3 = 4$, whereas the path 2 – 0 has a distance of 6. So, the Shortest path from 2 to 0 is 4.

3.2.2 Minimum Spanning Tree (Prim's and Kruskal's Algorithms)

Q7. Discuss about Minimum Spanning Tree.

Ans :

Spanning Tree

Spanning tree can be defined as a sub-graph of connected, undirected graph G that is a tree produced by removing the desired number of edges from a graph. In other words, spanning tree is a non-cyclic sub-graph of a connected and undirected graph G that connects all the vertices together. A graph G can have multiple spanning trees.

Minimum Spanning Tree

There can be weights assigned to every edge in a weighted graph. However, A minimum spanning tree is a spanning tree which has minimal total weight. In other words, minimum spanning tree is the one which contains the least weight among all other spanning trees of some particular graph.

Shortest path algorithms

There are two algorithms which are being used for this purpose.

- Prim's Algorithm
- Kruskal's Algorithm

3.2.2.1 Prim's Algorithm

Q8. Explain about prim's Algorithm

Ans :

(Imp.)

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.

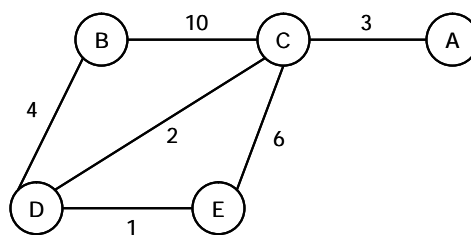
The algorithm is given as follows.

Algorithm

- Step 1: Select a starting vertex
- Step 2:** Repeat Steps 3 and 4 until there are fringe vertices
- Step 3:** Select an edge 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

Example:

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.

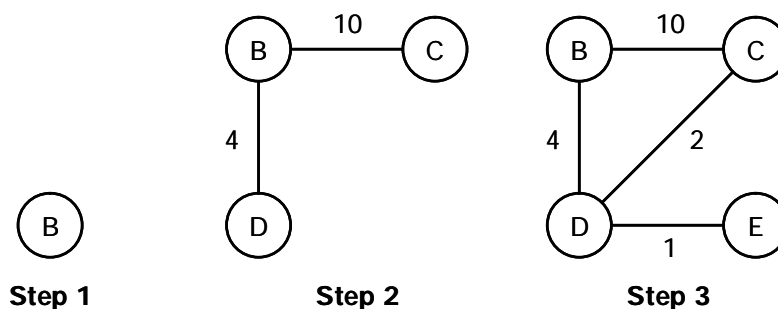
Step 3: Choose the edge with the minimum weight among all. i.e. B D and add it to MST. Add the adjacent vertices of D i.e. C and E.

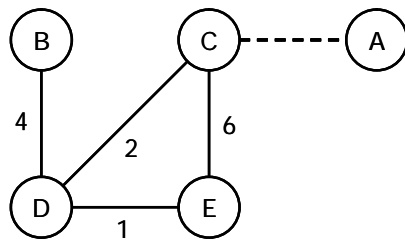
Step 3: Choose the edge with the minimum weight among all. In this case, the edges DE and CD are such edges. Add the mtoMST and explore the adjacent of C i.e. E and A.

Step 4 : Choose the edge with the minimum weight i.e. CA. We can't choose C E as it would cause cycle in the graph.

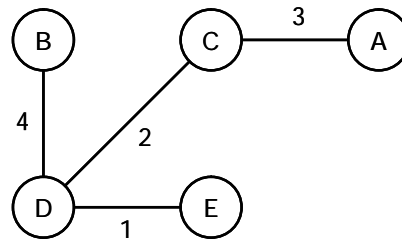
The graph produces in the step 4 is the minimum spanning tree of the graph shown in the above figure. The cost of MST will be calculated as,

$$\text{cost(MST)} = 4 + 2 + 1 + 3 = 10 \text{ units.}$$





Step 4



Step 5

3.2.2.2 Kruskal's Algorithm

Q9. Explain about Kruskal's Algorithm.

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).

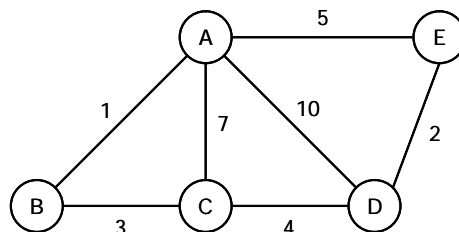
ELSE

Discard the edge

Step 6: END

Example:

Apply the Kruskal's algorithm on the graph given as follows.



Sol :

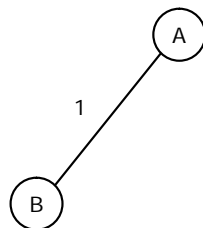
The weight of the edges given as:

Edge	AE	AD	AC	AB	BC	CD	DE
Weight	5	10	7	1	3	4	2

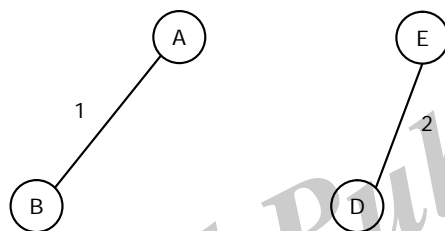
Sort the edges according to their weights.

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.

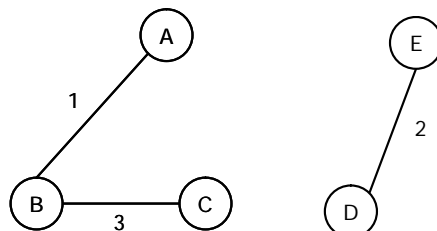
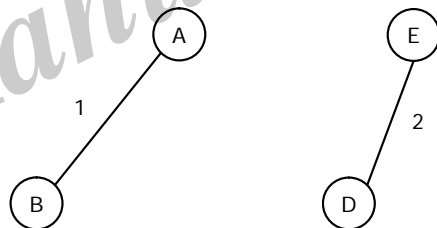


Add DE to the MST;



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 = $1 + 2 + 3 + 4 = 10$.

Short Question and Answers

1. What is Graph? Give an example

Ans:

A graph G can be defined as an ordered set $G(V, E)$ where $V(G)$ represents the set of vertices and $E(G)$ represents the set of edges which are used to connect these vertices.

A Graph $G(V, E)$ with 5 vertices (A, B, C, D, E) and six edges ((A,B), (B,C), (C,E), (E,D), (D,B), (D,A)) is shown in the following figure

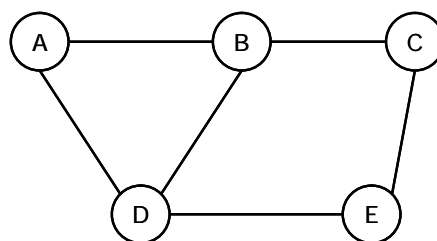


Fig.: Undirected Graph

2. Difference between Prim's Algorithm and Kruskal's Algorithm.

Ans :

Prim's Algorithm	Kruskal's Algorithm
It starts to build the Minimum Spanning Tree from any vertex in the graph.	It starts to build the Minimum Spanning Tree from the vertex carrying minimum weight in the graph.
It traverses one node more than one time to get the minimum distance.	It traverses one node only once.
Prim's algorithm has a time complexity of $O(V^2)$, V being the number of vertices and can be improved up to $O(E \log V)$ using Fibonacci heaps.	Kruskal's algorithm's time complexity is $O(E \log V)$, V being the number of vertices.
Prim's algorithm gives connected component as well as it works only on connected graph.	Kruskal's algorithm can generate forest(disconnected components) at any instant as well as it can work on disconnected components
Prim's algorithm runs faster in dense graphs.	Kruskal's algorithm runs faster in sparse graphs.
It generates the minimum spanning tree starting from the root vertex.	It generates the minimum spanning tree starting from the least weighted edge.
Applications of prim's algorithm are Travelling Salesman Problem, Network for roads and Rail tracks connecting all the cities etc.	Applications of Kruskal algorithm are LAN connection, TV Network etc.
Prim's algorithm prefer list data structures.	Kruskal's algorithm prefer heap data structures.

3. Differences between BFS and DFS.

Ans :

The following are the differences between the BFS and DFS:

Sr. No.	Key	BFS	DFS
1	Definition	BFS, stands for Breadth First Search.	DFS, stands for Depth First Search.
2	Data structure	BFS uses Queue to find the shortest path.	DFS uses Stack to find the shortest path.
3	Source	BFS is better when target is closer to Source.	DFS is better when target is far from source.
4	Suitability for decision tree	As BFS considers all neighbour so it is not suitable for decision tree used in puzzle games.	DFS is more suitable for decision tree. As with one decision, we need to traverse further to augment the decision. If we reach the conclusion, we won.
5	Speed	BFS is slower than DFS.	DFS is faster than BFS.
6	Time Complexity	Time Complexity of BFS = $O(V + E)$ where V is vertices and E is edges.	Time Complexity of DFS is also $O(V + E)$ where V is vertices and E is edges.

4. What is Minimum Spanning Tree?

Ans :

Spanning Tree

Spanning tree can be defined as a sub-graph of connected, undirected graph G that is a tree produced by removing the desired number of edges from a graph. In other words, spanning tree is a non-cyclic sub-graph of a connected and undirected graph G that connects all the vertices together. A graph G can have multiple spanning trees.

Minimum Spanning Tree

There can be weights assigned to every edge in a weighted graph. However, A minimum spanning tree is a spanning tree which has minimal total weight. In other words, minimum spanning tree is the one which contains the least weight among all other spanning tree of some particular graph.

5. What are the components of a Graph

Ans:

Components of a Graph

➤ Vertices

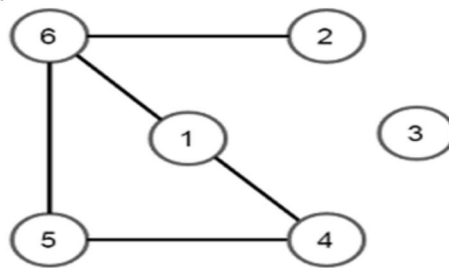
Vertices are the fundamental units of the graph. Sometimes, vertices are also known as vertex or nodes. Every node/vertex can be labeled or unlabelled.

➤ **Edges**

Edges are drawn or used to connect two nodes of the graph. It can be ordered pair of nodes in a directed graph. Edges can connect any two nodes in any possible way. There are no rules. Sometimes, edges are also known as arcs. Every edge can be labeled/unlabelled.

$$V = \{1, 2, 3, 4, 5, 6\}$$

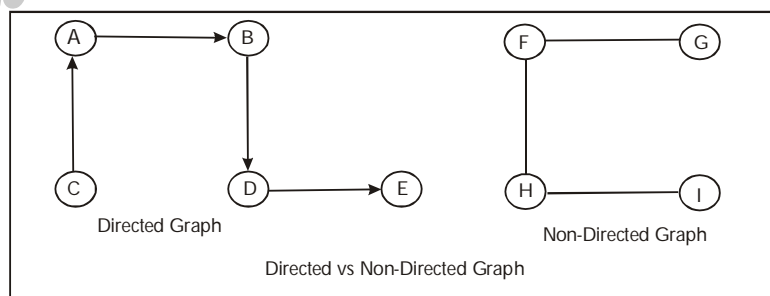
$$E = \{(1, 4), (1, 6), (2, 6), (4, 5), (5, 6)\}$$

**6. What is the difference between directed graph and non-directed graph?**

Ans:

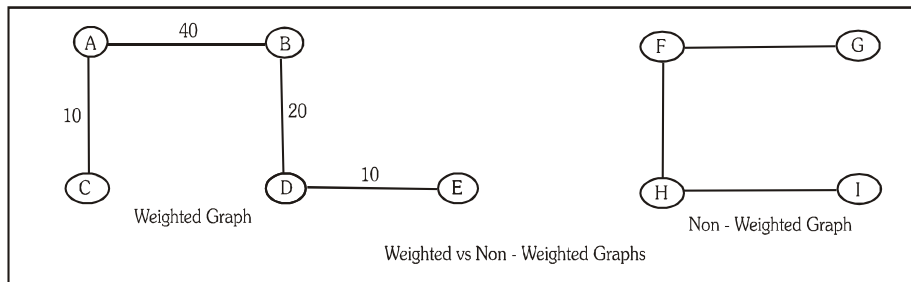
Directed graphs are graphs in which the edges have a direction. i.e. you can go from vertex A to vertex B, but you cannot go from vertex B to vertex A. An example of a directed graph is a one-way street city map. You can go only one direction on the edge (street) but not the other direction.

Non-directed graphs are graphs in which the edges do not have a direction. i.e. you can go from vertex A to vertex B and vice versa. An example of a non-directed graph is a freeway map connecting cities. You can go both directions on the edge

**7. What are weighted graphs?**

Ans:

Weighted graphs are graphs in which edges are given weights to represent the value of a variable. For example, in a graph of cities and freeways, the weight of an edge could represent the time it takes to drive from one city to the other. Similarly, in an airline routes graph, the weight of an edge could represent the cost of travel from one city to another.

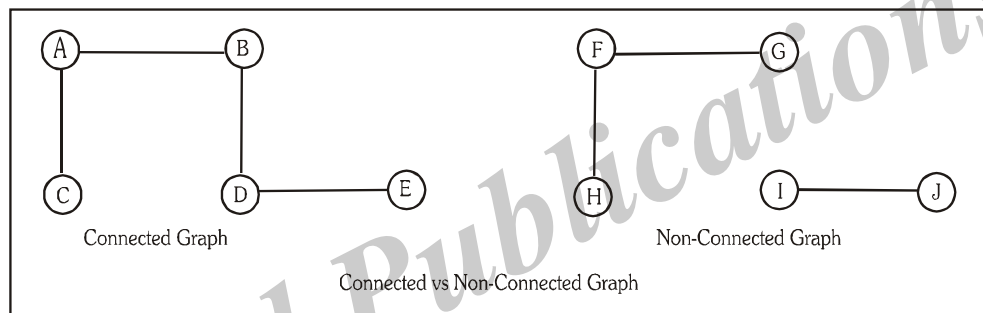


8. What is the difference between Connected Graph and Non-Connected Graph?

Ans:

In a connected graph there is at-least one path from every vertex to every other vertex.

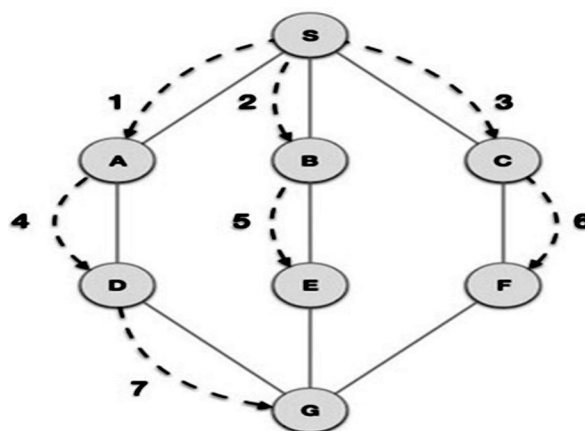
In a non-connected graph every vertex may not be connected to every other vertex.



9. Explain about BFS Search Algorithm

Ans:

Breadth First Search (BFS) algorithm traverses a graph in a breadth ward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



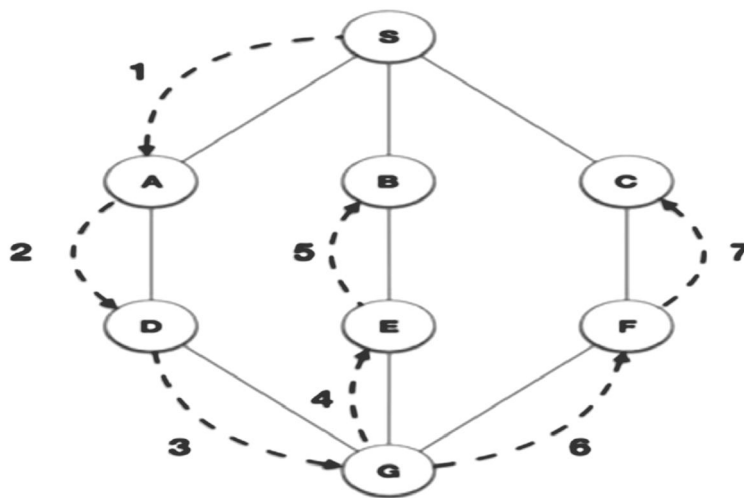
As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

- **Rule 1** - Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
- **Rule 2** - If no adjacent vertex is found, remove the first vertex from the queue.
- **Rule 3** - Repeat Rule 1 and Rule 2 until the queue is empty.

10. Explain about DFS Search Algorithm.

Ans:

Depth First Search (DFS) algorithm traverses a graph in a depth ward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

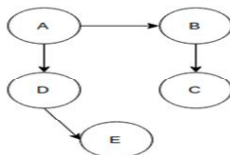


As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.

- **Rule 1** - Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- **Rule 2** - If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- **Rule 3** - Repeat Rule 1 and Rule 2 until the stack is empty.

Choose the Correct Answers

1. A graph in which all vertices have equal degree is known as _____ [a]
(a) Complete graph (b) Regular graph
(c) Multi graph (d) Simple graph
2. A graph is a tree if and only if graph is [b]
(a) Directed graph (b) Contains no cycles
(c) Planar (d) Completely connected
3. Which of the following data structure is required to convert arithmetic expression in infix to its equivalent postfix notation? [d]
(a) Queue (b) Linked list
(c) Binary search tree (d) None of above
4. If two trees have same structure and but different node content, then they are called _____ [d]
(a) Synonyms trees (b) Joint trees
(c) Equivalent trees (d) Similar trees
5. If two trees have same structure and node content, then they are called _____ [c]
(a) Synonyms trees (b) Joint trees
(c) Equivalent trees (d) Similar trees
6. The operation of processing each element in the list is known as _____ [d]
(a) Sorting (b) Merging
(c) Inserting (d) Traversal
7. Which of the following is non-linear data structure? [d]
(a) Stacks (b) List
(c) Strings (d) Trees
8. To represent hierarchical relationship between elements, which data structure is suitable? [c]
(a) Dequeue (b) Priority
(c) Tree (d) Graph
9. What would be the DFS traversal of the given Graph? [a]



- (a) ABCED (b) AEDCB
(c) EDCBA (d) ADECB
10. Which of the following properties does a simple graph not hold? [a]
(a) Must be connected (b) Must be unweighted
(c) Must have no loops or multiple edges (d) Must have no multiple edges

Fill in the Blanks

1. The process where two rotations are required to balance a tree is called_____
2. When the height of the left subtree and right subtree of a node in an AVL Tree are equal the balancing factor is_____
3. An Example of non-linear Data structure is _____
4. In RDBMS, the efficient data structure used in the Internal storage representation is_____
5. _____ is the most suitable data structure to represent a dictionary of word.
6. Depth First Search is equivalent to which of the traversal in the Binary Trees _____
7. The Data structure used in standard implementation of Breadth First Search is _____
8. The Depth First Search traversal of a graph will result into a _____
9. A person wants to visit some places. He starts from a vertex and then wants to visit every vertex till it finishes from one vertex, backtracks and then explore other vertex from same vertex. What algorithm he should use_____
10. In Depth First Search, how many times a node is visited _____

ANSWERS

1. Double rotation.
2. 0 (Zero)
3. Graph
4. B+ Tree
5. Binary Search Tree
6. Pre-order
7. Stack
8. Tree
9. DFS
10. Equivalent to number of indegree of the node

One Mark Answers

1. What is the difference between connected graph and non-connected graph?

Ans:

In a connected graph there is at-least one path from every vertex to every other vertex.

In a non-connected graph every vertex may not be connected to every other vertex.

2. What is DFS?

Ans:

Depth-first search is a simple preorder or postorder traversal for a tree, and it contains only tree edges

3. What is Dijkstra's Algorithm?

Ans:

It Finds the shortest path from one node to all other nodes in a weighted graph.

4. What is Topological Sort?

Ans:

It Arranges the nodes in a directed, acyclic graph in a special order based on incoming edges.

5. What is Minimum Spanning Tree?

Ans:

It Finds the cheapest set of edges needed to reach all nodes in a weighted graph.

6. Diameter of a Graph.

Ans:

The maximum eccentricity from all the vertices is considered as the diameter of the Graph G. The maximum among all the distances between a vertex to all other vertices is considered as the diameter of the Graph G.

7. List out the DFS applications

Ans:

Artificial intelligence and machine learning. These two algorithms are very common.

8. List out the Dijkstra Applications

Ans:

Google Maps uses Dijkstra to find the shortest path in navigation and IP Routing.

9. Which data structures are used in BFS and DFS algorithm?

Ans:

➤ Ans: In BFS algorithm, Queue data structure is used.

➤ In DFS algorithm, Stack data structure is used.

10. What are the applications of Graph data structure?

Ans:

Graphs are used in maps that draw cities/states/regions as vertices and adjacency relations as edges.

UNIT IV

Searching: Linear Search and Binary Search Techniques and their complexity analysis. Sorting and Complexity Analysis: Selection Sort, Bubble Sort, Insertion Sort, Quick Sort, Merge Sort, and Heap Sort. Algorithm Design Techniques: Greedy algorithm, divide-and-conquer, dynamic programming.

4.1 SEARCHING

Q1. Explain about searching.

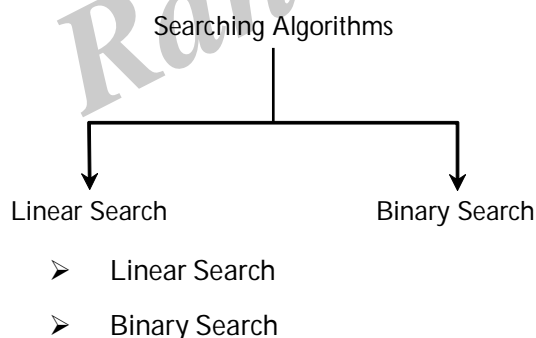
Ans:

Searching

- Searching is a process of finding a particular element among several given elements.
- The search is successful if the required element is found.
- Otherwise, the search is unsuccessful.

Searching Algorithms

Searching Algorithms are a family of algorithms used for the purpose of searching. The searching of an element in the given array may be carried out in the following two ways:



4.1.1 Linear Search

Q2. Explain in detail about Linear Search with an Example?

Ans:

(Imp.)

- Linear Search is the simplest searching algorithm.

- It traverses the array sequentially to locate the required element.
- It searches for an element by comparing it with each element of the array one by one.
- So, it is also called as Sequential Search.
- Linear Search Algorithm is applied when-
- No information is given about the array.
- The given array is unsorted or the elements are unordered.
- The list of data items is smaller.

Linear Search Algorithm

Consider

- There is a linear array 'a' of size 'n'.
- Linear search algorithm is being used to search an element 'item' in this linear array.
- If search ends in success, it sets loc to the index of the element otherwise it sets loc to -1.

Linear Search Algorithm is as follows-

Linear_Search (a , n , item , loc)

Begin

for i = 0 to (n - 1) by 1 do

if (a[i] = item) then

set loc = i

Exit

endif

endfor

set loc = -1

End

Linear Search Example**Consider**

- We are given the following linear array.
- Element 15 has to be searched in it using Linear Search Algorithm.

92	87	53	10	15	23	67
0	1	2	3	4	5	6

Linear Search Example

Now,

- Linear Search algorithm compares element 15 with all the elements of the array one by one.
- It continues searching until either the element 15 is found or all the elements are searched.
- Linear Search Algorithm works in the following steps:

Step - 01

- It compares element 15 with the 1st element 92.
- Since $15 \neq 92$, so required element is not found.
- So, it moves to the next element.

Step - 02

- It compares element 15 with the 2nd element 87.
- Since $15 \neq 87$, so required element is not found.
- So, it moves to the next element

Step - 03

- It compares element 15 with the 3rd element 53.
- Since $15 \neq 53$, so required element is not found.
- So, it moves to the next element.

Step - 04

- It compares element 15 with the 4th element 10.
- Since $15 \neq 10$, so required element is not found.
- So, it moves to the next element.

Step - 05

- It compares element 15 with the 5th element 15.
- Since $15 = 15$, so required element is found.
- Now, it stops the comparison and returns index 4 at which element 15 is present.

Application of Linear Search Algorithm

The linear search algorithm has the following applications:

- Linear search can be applied to both single-dimensional and multi-dimensional arrays.
- Linear search is easy to implement and effective when the array contains only a few elements.
- Linear Search is also efficient when the search is performed to fetch a single search in an unordered-List.

4.1.2 Binary Search**Q3. Explain in detail about Binary Search with an example?**

Ans :

Binary Search is a searching algorithm used in a sorted array by repeatedly dividing the search interval in half. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to $O(\log n)$.

Binary Search

- Binary Search is one of the fastest searching algorithms.
- It is used for finding the location of an element in a linear array.
- It works on the principle of divide and conquer technique.

Binary Search Algorithm can be applied only on Sorted arrays.

So, the elements must be arranged in-

- Either ascending order if the elements are numbers.
 - Or dictionary order if the elements are strings.
- To apply Binary Search on an unsorted array,
- First, sort the array using some sorting technique.
 - Then, use binary search algorithm.

Binary Search Algorithm

Consider

- There is a linear array 'a' of size 'n'.
- Binary search algorithm is being used to search an element 'item' in this linear array.
- If search ends in success, it sets loc to the index of the element otherwise it sets loc to -1.
- Variables beg and end keeps track of the index of the first and last element of the array or sub array in which the element is being searched at that instant.
- Variable mid keeps track of the index of the middle element of that array or sub array in which the element is being searched at that instant.

Binary Search Algorithm is as follows

Begin

Set beg = 0

Set end = n-1

Set mid = (beg + end) / 2

while ((beg <= end) and (a[mid] ≠ item)) do

if (item < a[mid]) then

Set end = mid - 1

else

Set beg = mid + 1

endif

Set mid = (beg + end) / 2

endwhile

if (beg > end) then

Set loc = -1

else

Set loc = mid

endif

End

Explanation

Binary Search Algorithm searches an element by comparing it with the middle most element of the array.

Then, following three cases are possible:

Case - 01

If the element being searched is found to be the middle most element, its index is returned.

Case - 02

If the element being searched is found to be greater than the middle most element, then its search is further continued in the right sub array of the middle most element.

Case - 03

If the element being searched is found to be smaller than the middle most element, then its search is further continued in the left sub array of the middle most element.

This iteration keeps on repeating on the sub arrays until the desired element is found (or) size of the sub array reduces to zero.

Binary Search Example**Consider**

- We are given the following sorted linear array.
- Element 15 has to be searched in it using Binary Search Algorithm.

3	10	15	20	35	40	60
---	----	----	----	----	----	----

a[0] a[1] a[2] a[3] a[4] a[5] a[6]

Binary Search Example

Binary Search Algorithm works in the following steps:

Step - 01

- To begin with, we take beg=0 and end=6.
- We compute location of the middle element as- mid

$$= (\text{beg} + \text{end}) / 2$$

$$= (0 + 6) / 2$$

$$= 3$$
- Here, $a[\text{mid}] = a[3] = 20 \neq 15$ and $\text{beg} < \text{end}$.
- So, we start next iteration.

Step - 02

- Since $a[\text{mid}] = 20 > 15$, so we take $\text{end} = \text{mid} - 1 = 3 - 1 = 2$ whereas beg remains unchanged.
- We compute location of the middle element as-mid

$$= (\text{beg} + \text{end}) / 2$$

$$= (0 + 2) / 2$$

$$= 1$$
- Here, $a[\text{mid}] = a[1] = 10 \neq 15$ and $\text{beg} < \text{end}$.
- So, we start next iteration.

Step - 03

- Since $a[\text{mid}] = 10 < 15$, so we take $\text{beg} = \text{mid} + 1 = 1 + 1 = 2$ whereas end remains unchanged.
- We compute location of the middle element as- mid

$$= (\text{beg} + \text{end}) / 2$$

$$= (2 + 2) / 2$$

$$= 2$$

- Here, $a[\text{mid}] = a[2] = 15$ which matches to the element being searched.

- So, our search terminates in success and index 2 is returned.

4.1.3 Complexity Analysis of Linear Search

Q4. Discuss in detail about Complexity Analysis of Linear Search.

Ans:

Time Complexity Analysis

Linear Search time complexity analysis is done below-

Best case

In the best possible case,

- The element being searched may be found at the first position.
- In this case, the search terminates in success with just one comparison.
- Thus in best case, linear search algorithm takes $O(1)$ operations.

Worst Case

In the worst possible case,

- The element being searched may be present at the last position or not present in the array at all.
- In the former case, the search terminates in success with n comparisons.
- In the later case, the search terminates in failure with n comparisons.
- Thus in worst case, linear search algorithm takes $O(n)$ operations.

Thus, we have-

Time Complexity of Linear Search Algorithm is $O(n)$

Here, n is the number of elements in the linear array.

Space Complexity Analysis of Linear Search:

In Linear Search, we are creating a Boolean variable to store if the element to be searched is present or not.

The variable is initialized to false and if the element is found, the variable is set to true. This variable can be used in other processes or returned by the function.

In Linear Search function, we can avoid using this Boolean variable as well and return true or false directly.

The input to Linear Search involves:

- A list/ array of N elements
- A variable storing the element to be searched.

As the amount of extra data in Linear Search is fixed, the Space Complexity is $O(1)$.

Therefore, Space Complexity of Linear Search is $O(1)$.

- Best Case Time Complexity of Linear Search: $O(1)$
- Average Case Time Complexity of Linear Search: $O(N)$
- Worst Case Time Complexity of Linear Search: $O(N)$
- Space Complexity of Linear Search: $O(1)$
- Number of comparisons in Best Case: 1
- Number of comparisons in Average Case: $N/2 + N/(N+1)$
- Number of comparisons in Worst Case: N

4.1.4 Time Complexity Analysis of Binary Search

Q5. Discuss in detail about Complexity Analysis of Binary Search.

Ans :

Binary Search time complexity analysis is done below:

- In each iteration or in each recursive call, the search gets reduced to half of the array.
- So for n elements in the array, there are $\log_2 n$ iterations or recursive calls.

Thus, we have

Time Complexity of Binary Search Algorithm is $O(\log_2 n)$:

Here, n is the number of elements in the sorted linear array.

This time complexity of binary search remains unchanged irrespective of the element position even if it is not present in the array.

Space Complexity Analysis of Binary Search:

In an iterative implementation of Binary Search, the space complexity will be $O(1)$.

This is because we need two variables to keep track of the range of elements that are to be checked. No other data is needed.

In a recursive implementation of Binary Search, the space complexity will be $O(\log N)$.

This is because in the worst case, there will be $\log N$ recursive calls and all these recursive calls will be stacked in memory. In fact, if 1 comparisons are needed, then 1 recursive calls will be stacked in memory and from our analysis of average case time complexity, we know that the average memory will be $O(\log N)$ as well.

Time and Space Complexity analysis of Binary Search is as follows:

- Best Case Time Complexity of Binary Search: $O(1)$
- Average Case Time Complexity of Binary Search: $O(\log N)$
- Worst Case Time Complexity of Binary Search: $O(\log N)$
- Space Complexity of Binary Search: $O(1)$ for iterative, $O(\log N)$ for recursive.

4.2 SORTING AND COMPLEXITY ANALYSIS

4.2.1 Selection Sort

Q6. Explain in detail about Selection Sort with an example and its complexity.

Ans: (Imp.)

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.

Working of Selection Sort

Follow the below steps to solve the problem:

- Initialize minimum value(min_idx) to location 0.
- Traverse the array to find the minimum element in the array.
- While traversing if any element smaller than min_idx is found then swap both the values.
- Then, increment min_idx to point to the next element.
- Repeat until the array is sorted.

Let's consider the following array as an example: arr[] = {64, 25, 12, 22, 11}.

First pass

- For the first position in the sorted array, the whole array is traversed from index 0 to 4 sequentially. The first position where 64 is stored presently, after traversing whole array it is clear that 11 is the lowest value.

64	25	12	22	11
----	----	----	----	----

- Thus, replace 64 with 11. After one iteration 11, which happens to be the least value in the array, tends to appear in the first position of the sorted list.

11	25	12	22	64
----	----	----	----	----

Second Pass

- For the second position, where 25 is present, again traverse the rest of the array in a sequential manner.

11	25	12	22	64
----	----	----	----	----

- After traversing, we found that 12 is the second lowest value in the array and it should appear at the second place in the array, thus swap these values.

11	12	25	22	64
----	----	----	----	----

Third Pass

- Now, for third place, where 25 is present again traverse the rest of the array and find the third least value present in the array.

11	12	25	22	64
----	----	----	----	----

- While traversing, 22 came out to be the third least value and it should appear at the third place in the array, thus swap 22 with element present at third position.

11	12	22	25	64
----	----	----	----	----

Fourth pass

- Similarly, for fourth position traverse the rest of the array and find the fourth least element in the array
- As 25 is the 4th lowest value hence, it will place at the fourth position.

11	12	22	25	64
----	----	----	----	----

Fifth Pass

- At last the largest value present in the array automatically get placed at the last position in the array
- The resulted array is the sorted array.

11	12	22	25	64
----	----	----	----	----

Selection sort complexity

The time complexity of selection sort in best case, average case, and in worst case. We will also see the space complexity of the selection sort.

1. Time Complexity

Case	Time Complexity
Best Case	$O(n^2)$
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

- **Best Case Complexity:** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of selection sort is $O(n^2)$.
- **Average Case Complexity :** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of selection sort is $O(n^2)$.
- **Worst Case Complexity:** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of selection sort is $O(n^2)$.

2. Space Complexity

Space Complexity

 $O(1)$

Stable

YES

The space complexity of selection sort is $O(1)$. It is because, in selection sort, an extra variable is required for swapping.

4.2.2 Bubble Sort**Q7. Explain in detail about Bubble Sort with an example and its complexity.***Ans :*

Bubble sort works on the repeatedly swapping of adjacent elements until they are not in the intended order. It is called bubble sort because the movement of array elements is just like the movement of air bubbles in the water. Bubbles in water rise up to the surface; similarly, the array elements in bubble sort move to the end in each iteration.

Although it is simple to use, it is primarily used as an educational tool because the performance of bubble sort is poor in the real world. It is not suitable for large data sets. The average and worst-case complexity of Bubble sort is $O(n^2)$, where n is a number of items.

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

The 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.

Bubble sort complexity

Now, let's see the time complexity of bubble sort in the best case, average case, and worst case. We will also see the space complexity of bubble sort.

1. Time Complexity

Case	Time Complexity
Best Case	$O(n)$
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

Best Case Complexity: It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of bubble sort is $O(n)$.

Average Case Complexity: It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of bubble sort is $O(n^2)$.

Worst Case Complexity: It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of bubble sort is $O(n^2)$.

2. Space Complexity

Space Complexity	$O(1)$
------------------	--------

The space complexity of bubble sort is $O(1)$. It is because, in bubble sort, an extra variable is required for swapping.

The space complexity of optimized bubble sort is $O(2)$. It is because two extra variables are required in optimized bubble sort.

4.2.3 Insertion Sort

Q8. Explain in detail about Insertion Sort with an example and its complexity.

Ans : (Imp.)

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.

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.

Step 2

Pick the next element, and store it separately in a **key**.

Step 3

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

Now, let's see the working of the 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	24	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.

Insertion sort complexity

The time complexity of insertion sort in best case, average case, and in worst case. We will also see the space complexity of insertion sort.

1. Time Complexity

Case	Time Complexity
Best Case	$O(n)$
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

- **Best Case Complexity:** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of insertion sort is $O(n)$.
- **Average Case Complexity:** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of insertion sort is $O(n^2)$.
- **Worst Case Complexity:** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of insertion sort is $O(n^2)$.

2. Space Complexity

Space Complexity $O(1)$

The space complexity of insertion sort is $O(1)$. It is because, in insertion sort, an extra variable is required for swapping.

4.2.4 Quick Sort

Q9. Explain in detail about Quick Sort with an example and its complexity.

Ans :

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

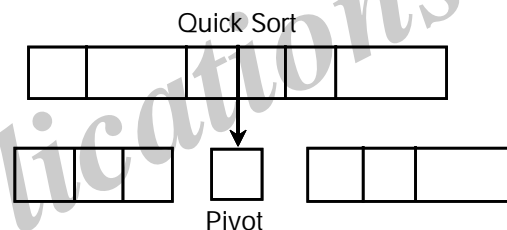
Recursively, sort two subarrays with Quicksort.

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.

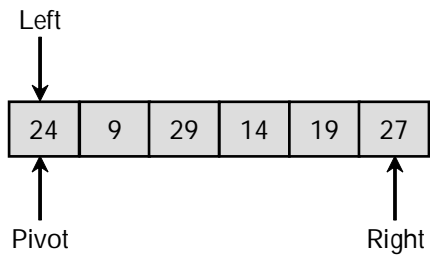
Working of Quick Sort Algorithm

Let the elements of array are -

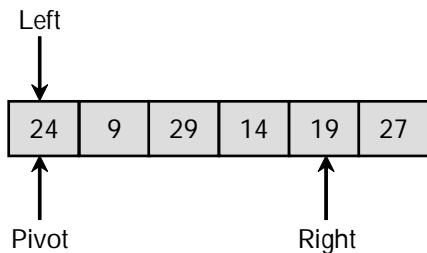
24	9	29	14	19	27
----	---	----	----	----	----

In the given array, we consider the leftmost element as pivot. So, in this case, $a[\text{left}] = 24$, $a[\text{right}] = 27$ and $a[\text{pivot}] = 24$.

Since, pivot is at left, so algorithm starts from right and move towards left.

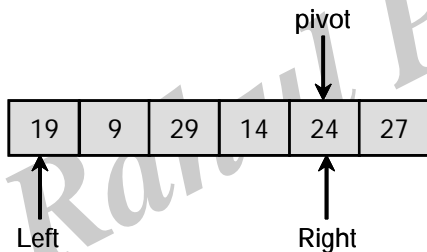


Now, $a[\text{pivot}] < a[\text{right}]$, so algorithm moves forward one position towards left, i.e. -



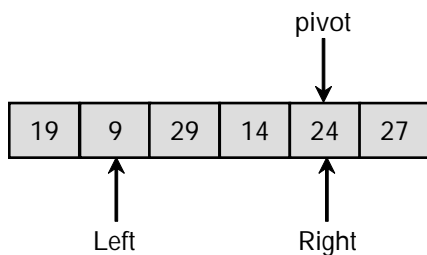
Now, $a[\text{left}] = 24$, $a[\text{right}] = 19$, and $a[\text{pivot}] = 24$.

Because, $a[\text{pivot}] > a[\text{right}]$, so, algorithm will swap $a[\text{pivot}]$ with $a[\text{right}]$, and pivot moves to right, as -

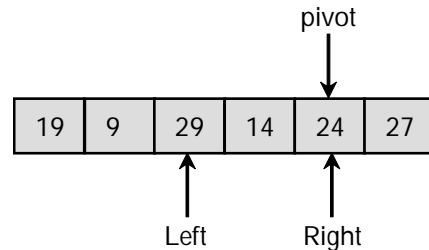


Now, $a[\text{left}] = 19$, $a[\text{right}] = 24$, and $a[\text{pivot}] = 24$. Since, pivot is at right, so algorithm starts from left and moves to right.

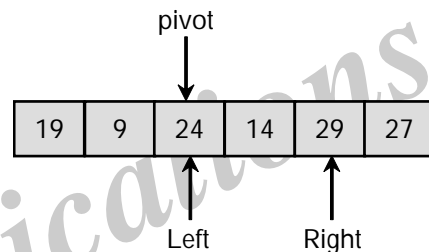
As $a[\text{pivot}] > a[\text{left}]$, so algorithm moves one position to right as -



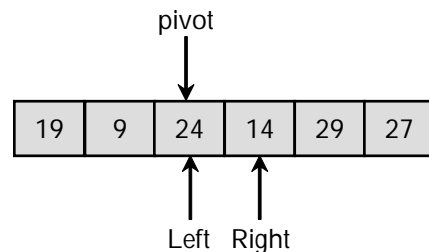
Now, $a[\text{left}] = 9$, $a[\text{right}] = 24$, and $a[\text{pivot}] = 24$. As $a[\text{pivot}] > a[\text{left}]$, so algorithm moves one position to right as -



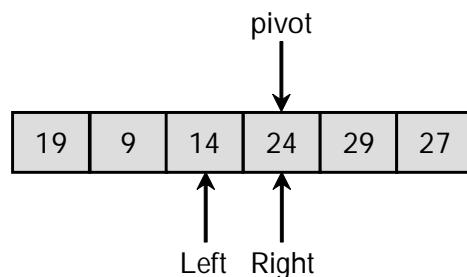
Now, $a[\text{left}] = 29$, $a[\text{right}] = 24$, and $a[\text{pivot}] = 24$. As $a[\text{pivot}] < a[\text{left}]$, so, swap $a[\text{pivot}]$ and $a[\text{left}]$, now pivot is at left, i.e. -



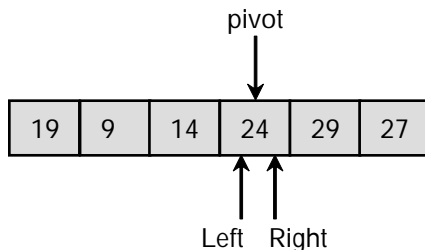
Since, pivot is at left, so algorithm starts from right, and move to left. Now, $a[\text{left}] = 24$, $a[\text{right}] = 29$, and $a[\text{pivot}] = 24$. As $a[\text{pivot}] < a[\text{right}]$, so algorithm moves one position to left, as -



Now, $a[\text{pivot}] = 24$, $a[\text{left}] = 24$, and $a[\text{right}] = 14$. As $a[\text{pivot}] > a[\text{right}]$, so, swap $a[\text{pivot}]$ and $a[\text{right}]$, now pivot is at right, i.e. -



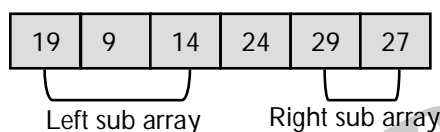
Now, $a[\text{pivot}] = 24$, $a[\text{left}] = 14$, and $a[\text{right}] = 24$. Pivot is at right, so the algorithm starts from left and move to right.



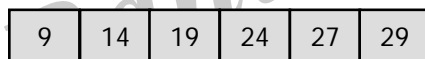
Now, $a[\text{pivot}] = 24$, $a[\text{left}] = 24$, and $a[\text{right}] = 24$. So, pivot, left and right are pointing the same element. It represents the termination of procedure.

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 -



Quicksort complexity

Now, let's see the time complexity of quicksort in best case, average case, and in worst case. We will also see the space complexity of quicksort.

1. Time Complexity

Case	Time Complexity
Best Case	$O(n \cdot \log n)$
Average Case	$O(n \cdot \log n)$
Worst Case	$O(n^2)$

- **Best Case Complexity** - In Quicksort, the best-case occurs when the pivot element is the middle element or near to the middle element. The best-case time complexity of quicksort is $O(n \cdot \log n)$.

- **Average Case Complexity:** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of quicksort is $O(n \cdot \log n)$.

- **Worst Case Complexity:** In quick sort, worst case occurs when the pivot element is either greatest or smallest element. Suppose, if the pivot element is always the last element of the array, the worst case would occur when the given array is sorted already in ascending or descending order. The worst-case time complexity of quicksort is $O(n^2)$.

2. Space Complexity

Space Complexity $O(n \cdot \log n)$

- The space complexity of quicksort is $O(n \cdot \log n)$.

4.2.5 Merge Sort

Q10. Explain in detail about Merge Sort with an example and its complexity.

Ans :

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.

1. MERGE_SORT(arr, beg, end)
- 2.
3. **if** beg < end
4. set mid = (beg + end)/2

5. MERGE_SORT(arr, beg, mid)
6. MERGE_SORT(arr, mid + 1, end)
7. MERGE (arr, beg, mid, end)
8. end of **if**
- 9.
10. 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 $A[\text{beg} \dots \text{mid}]$ and $A[\text{mid} + 1 \dots \text{end}]$, to build one sorted array $A[\text{beg} \dots \text{end}]$. So, the inputs of the MERGE function are $A[]$, beg, mid, and end.

Working of Merge sort Algorithm:

The working of merge sort Algorithm.

To understand the working of the merge sort algorithm, let's take an unsorted array. It will be easier to understand the merge sort via an example.

Let the elements of array are:

12	31	25	8	32	17	40	42
----	----	----	---	----	----	----	----

According to the merge sort, first divide the given array into two equal halves. Merge sort keeps dividing the list into equal parts until it cannot be further divided.

As there are eight elements in the given array, so it is divided into two arrays of size 4.

divide	12	31	25	8	32	17	40	42
--------	----	----	----	---	----	----	----	----

Now, again divide these two arrays into halves. As they are of size 4, so divide them into new arrays of size 2.

divide	12	31	25	8	32	17	40	42
--------	----	----	----	---	----	----	----	----

Now, again divide these arrays to get the atomic value that cannot be further divided.

divide	12	31	25	8	32	17	40	42
--------	----	----	----	---	----	----	----	----

Now, combine them in the same manner they were broken.

In combining, first compare the element of each array and then combine them into another array in sorted order.

So, first compare 12 and 31, both are in sorted positions. Then compare 25 and 8, and in the list of two values, put 8 first followed by 25. Then compare 32 and 17, sort them and put 17 first followed by 32. After that, compare 40 and 42, and place them sequentially.

merge	12	31	8	25	17	32	40	42
-------	----	----	---	----	----	----	----	----

In the next iteration of combining, now compare the arrays with two data values and merge them into an array of found values in sorted order.

merge	8	12	25	31	17	32	40	42
-------	---	----	----	----	----	----	----	----

Now, there is a final merging of the arrays. After the final merging of above arrays, the array will look like -

8	12	17	24	31	32	40	42
---	----	----	----	----	----	----	----

Now, the array is completely sorted.

Merge sort complexity

Now, let's see the time complexity of merge sort in best case, average case, and in worst case. We will also see the space complexity of the merge sort.

1. Time Complexity

Case	Time Complexity
Best Case	$O(n \cdot \log n)$
Average Case	$O(n \cdot \log n)$
Worst Case	$O(n \cdot \log n)$

- **Best Case Complexity:** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of merge sort is $O(n \cdot \log n)$.
- **Average Case Complexity:** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of merge sort is $O(n \cdot \log n)$.
- **Worst Case Complexity:** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of merge sort is $O(n \cdot \log n)$.

2. Space Complexity

Space Complexity $O(n)$

- The space complexity of merge sort is $O(n)$. It is because, in merge sort, an extra variable is required for swapping.

4.2.6 Heap Sort

Q11. Explain in detail about Heap Sort with an example and its complexity.

Ans :

(Imp.)

Heap sort processes the elements by creating the min-heap or max-heap using the elements of the given array. Min-heap or max-heap represents the ordering of array in which the root element represents the minimum or maximum element of the array.

Heap sort basically recursively performs two main operations -

- Build a heap H , using the elements of array.
- Repeatedly delete the root element of the heap formed in 1st phase.

Before knowing more about the heap sort, let's first see a brief description of **Heap**.

Heap

A heap is a complete binary tree, and the binary tree is a tree in which the node can have the utmost two children. 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.

Heap Sort

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.

Working of Heap sort Algorithm

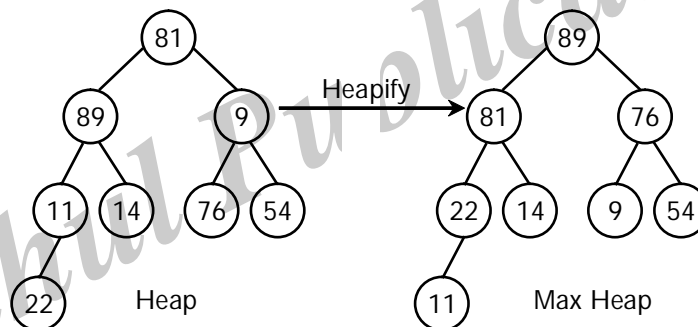
The working of the Heapsort Algorithm.

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.

81	89	9	11	14	76	54	22
----	----	---	----	----	----	----	----

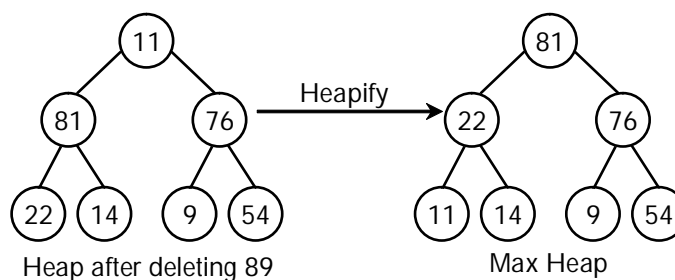
- First, we have to construct a heap from the given array and convert it into max heap.



- After converting the given heap into max heap, the array elements are -

89	81	76	22	14	9	54	11
----	----	----	----	----	---	----	----

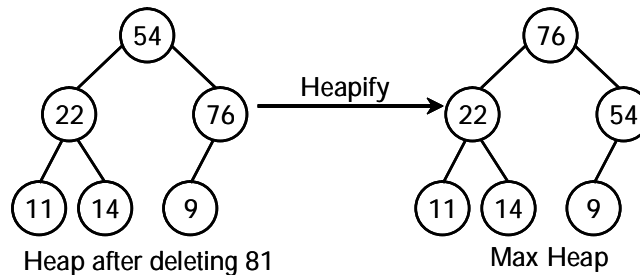
- Next, we have to delete the root element (**89**) from the max heap. To delete this node, we have to swap it with the last node, i.e. (**11**). After deleting the root element, we again have to heapify it to convert it into max heap.



- 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
----	----	----	----	----	---	----	----

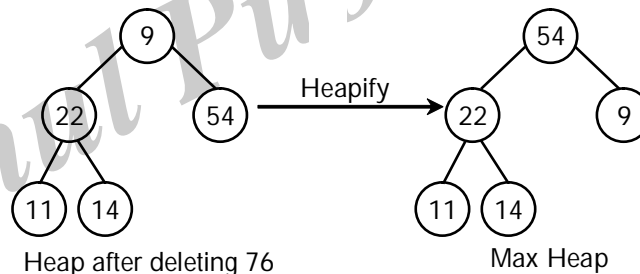
- In the next step, again, we have to delete the root element (**81**) from the max heap. To delete this node, we have to swap it with the last node, i.e. (**54**). After deleting the root element, we again have to heapify it to convert it into max heap.



- 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
----	----	----	----	----	---	----	----

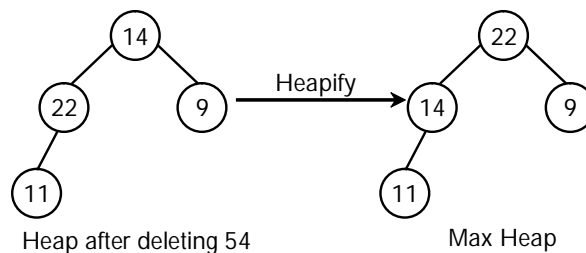
- In the next step, we have to delete the root element (**76**) from the max heap again. To delete this node, we have to swap it with the last node, i.e. (**9**). After deleting the root element, we again have to heapify it to convert it into max heap.



- 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
----	----	---	----	----	----	----	----

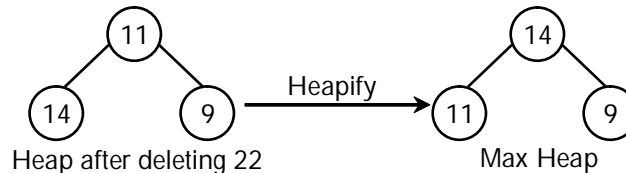
- In the next step, again we have to delete the root element (**54**) from the max heap. To delete this node, we have to swap it with the last node, i.e. (**14**). After deleting the root element, we again have to heapify it to convert it into max heap.



- 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
----	----	---	----	----	----	----	----

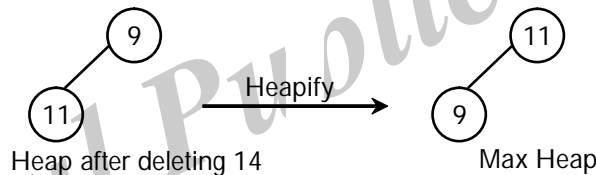
- In the next step, again we have to delete the root element (**22**) from the max heap. To delete this node, we have to swap it with the last node, i.e. (**11**). After deleting the root element, we again have to heapify it to convert it into max heap.



- 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
----	----	---	----	----	----	----	----

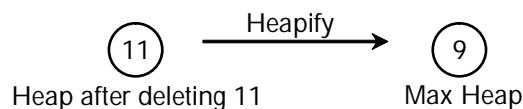
- In the next step, again we have to delete the root element (**14**) from the max heap. To delete this node, we have to swap it with the last node, i.e. (**9**). After deleting the root element, we again have to heapify it to convert it into max heap.



- 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
----	---	----	----	----	----	----	----

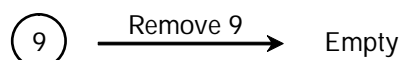
- In the next step, again we have to delete the root element (**11**) from the max heap. To delete this node, we have to swap it with the last node, i.e. (**9**). After deleting the root element, we again have to heapify it to convert it into max heap.



- 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.



- After completion of sorting, the array elements are -

9	11	14	22	54	76	81	89
---	----	----	----	----	----	----	----

- Now, the array is completely sorted.

Heap sort complexity

- Now, let's see the time complexity of Heap sort in the best case, average case, and worst case. We will also see the space complexity of Heapsort.

1. Time Complexity

Case	Time Complexity
Best Case	$O(n \log n)$
Average Case	$O(n \log n)$
Worst Case	$O(n \log n)$

- **Best Case Complexity** - It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of heap sort is **$O(n \log n)$** .
- **Average Case Complexity** - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of heap sort is **$O(n \log n)$** .
- **Worst Case Complexity** - It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of heap sort is **$O(n \log n)$** .

The time complexity of heap sort is $O(n \log n)$ in all three cases (best case, average case, and worst case). The height of a complete binary tree having n elements is $\log n$.

2. Space Complexity

Space Complexity $O(1)$

- The space complexity of Heap sort is $O(1)$.

Q12. Compare various sorting techniques with real world usage.

Ans :

(Imp.)

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 insertion sort in terms of efficiency even for small sequences.

Pros

- Again nothing, maybe just "catchy name"

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 $O(n \log n)$ 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 $O(n \log n)$
- 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 $O(n \log n)$
- 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 $O(n \log n)$ 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 $O(n \log n)$ 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 $O(n \log n)$ 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 $n \log n$ 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 $n \log n$

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	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$
Quick sort	$O(n^2)$	$O(n \log n)$	$O(n \log n)$	$O(\log n)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$

4.3 ALGORITHM DESIGN TECHNIQUES

4.3.1 Greedy Algorithm

Q13. Explain in detail about Greedy Algorithm with its applications.

Ans:

(Imp.)

The greedy method is one of the strategies like Divide and conquer used to solve the problems. This method is used for solving optimization problems. An optimization problem is a problem that demands either maximum or minimum results. Let's understand through some terms.

The Greedy method is the simplest and straightforward approach. It is not an algorithm, but it is a technique. The main function of this approach is that the decision is taken on the basis of the currently available information. Whatever the current information is present, the decision is made without worrying about the effect of the current decision in future.

This technique is basically used to determine the feasible solution that may or may not be optimal. The feasible solution is a subset that satisfies the given criteria. The optimal solution is the solution which is the best and the most favorable solution in the subset. In the case of feasible, if more than one solution satisfies the given criteria then those solutions will be considered as the feasible, whereas the optimal solution is the best solution among all the solutions.

Characteristics of Greedy method

The following are the characteristics of a greedy method:

- To construct the solution in an optimal way, this algorithm creates two sets where one set contains all the chosen items, and another set contains the rejected items.
- A Greedy algorithm makes good local choices in the hope that the solution should be either feasible or optimal.

Components of Greedy Algorithm

The components that can be used in the greedy algorithm are:

- **Candidate set:** A solution that is created from the set is known as a candidate set.
- **Selection function:** This function is used to choose the candidate or subset which can be added in the solution.
- **Feasibility function:** A function that is used to determine whether the candidate or subset can be used to contribute to the solution or not.
- **Objective function:** A function is used to assign the value to the solution or the partial solution.
- **Solution function:** This function is used to intimate whether the complete function has been reached or not.

Applications of Greedy Algorithm

- It is used in finding the shortest path.
- It is used to find the minimum spanning tree using the prim's algorithm or the Kruskal's algorithm.
- It is used in a job sequencing with a deadline.

This algorithm is also used to solve the fractional knapsack problem.

Understand through an example.

Suppose there is a problem 'P'. I want to travel from A to B shown as below:

P : A → B

The problem is that we have to travel this journey from A to B. There are various solutions to go from A to B. We can go from A to B by walk, car, bike, train, aeroplane etc. There is a constraint in the journey that we have to travel this journey within 12 hrs. If I go by train or aeroplane then only, I can cover this distance within 12 hrs. There are many solutions to this problem but there are only two solutions that satisfy the constraint.

If we say that we have to cover the journey at the minimum cost. This means that we have to travel this distance as minimum as possible, so this problem is known as a minimization problem. Till now, we have two feasible solutions, i.e., one by train and another one by air. Since travelling by train will lead to the minimum cost so it is an optimal solution. An optimal solution is also the feasible solution, but providing the best result so that solution is the optimal solution with the minimum cost. There would be only one optimal solution.

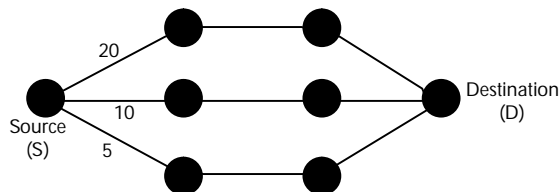
The problem that requires either minimum or maximum result then that problem is known as an optimization problem. Greedy method is one of the strategies used for solving the optimization problems.

Disadvantages of using Greedy algorithm

Greedy algorithm makes decisions based on the information available at each phase without considering the broader problem. So, there might be a possibility that the greedy solution does not give the best solution for every problem.

It follows the local optimum choice at each stage with a intend of finding the global optimum. Let's understand through an example.

Consider the graph which is given below:



We have to travel from the source to the destination at the minimum cost. Since we have three feasible solutions having cost paths as 10, 20, and 5. 5 is the minimum cost path so it is the optimal solution. This is the local optimum, and in this way, we find the local optimum at each stage in order to calculate the global optimal solution.

4.3.2 Divide and Conquer Algorithm

Q14. Explain in detail about Divide and Conquer Algorithm with its applications.

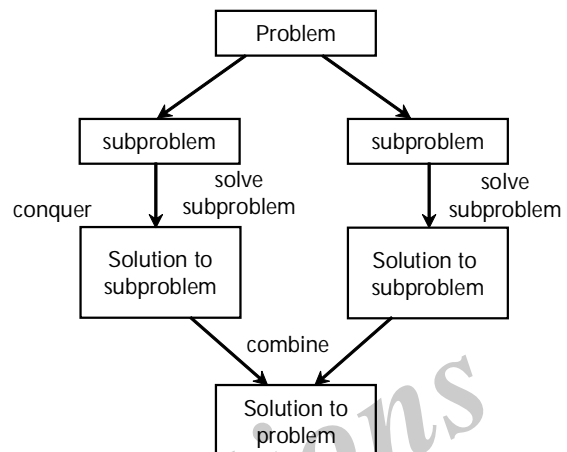
Ans :

Divide and Conquer is an algorithmic pattern. In algorithmic methods, the design is to take a dispute on a huge input, break the input into minor pieces, decide the problem on each of the small pieces, and then merge the piecewise solutions into a global solution. This mechanism of solving the problem is called the Divide & Conquer Strategy.

Divide and Conquer algorithm consists of a dispute using the following three steps.

1. **Divide** the original problem into a set of subproblems.

2. **Conquer:** Solve every subproblem individually, recursively.
3. **Combine:** Put together the solutions of the subproblems to get the solution to the whole problem.



Generally, we can follow the divide-and-conquer approach in a three-step process.

Examples

The specific computer algorithms are based on the Divide & Conquer approach:

1. Maximum and Minimum Problem
2. Binary Search
3. Sorting (merge sort, quick sort)
4. Tower of Hanoi.

Fundamental of Divide & Conquer Strategy:

There are two fundamentals of Divide & Conquer Strategy:

1. Relational Formula
 2. Stopping Condition
1. **Relational Formula:** It is the formula that we generate from the given technique. After generation of Formula we apply D&C Strategy, i.e. we break the problem recursively & solve the broken subproblems.
 2. **Stopping Condition:** When we break the problem using Divide & Conquer Strategy, then we need to know that for how much time, we need to apply divide & Conquer. So the condition where the need to stop our recursion steps of D&C is called as Stopping Condition.

Applications of Divide and Conquer Approach:

Following algorithms are based on the concept of the Divide and Conquer Technique:

1. **Binary Search:** The binary search algorithm is a searching algorithm, which is also called a half-interval search or logarithmic search. It works by comparing the target value with the middle element existing in a sorted array. After making the comparison, if the value differs, then the half that cannot contain the target will eventually eliminate, followed by continuing the search on the other half. We will again consider the middle element and compare it with the target value. The process keeps on repeating until the target value is met. If we found the other half to be empty after ending the search, then it can be concluded that the target is not present in the array.
2. **Quicksort:** It is the most efficient sorting algorithm, which is also known as partition-exchange sort. It starts by selecting a pivot value from an array followed by dividing the rest of the array elements into two sub-arrays. The partition is made by comparing each of the elements with the pivot value. It compares whether the element holds a greater value or lesser value than the pivot and then sort the arrays recursively.
3. **Merge Sort:** It is a sorting algorithm that sorts an array by making comparisons. It starts by dividing an array into sub-array and then recursively sorts each of them. After the sorting is done, it merges them back.
4. **Closest Pair of Points:** It is a problem of computational geometry. This algorithm emphasizes finding out the closest pair of points in a metric space, given n points, such that the distance between the pair of points should be minimal.
5. **Strassen's Algorithm:** It is an algorithm for matrix multiplication, which is named after Volker Strassen. It has proven to be much faster than the traditional algorithm when works on large matrices.

6. **Cooley-Tukey Fast Fourier Transform (FFT) algorithm:** The Fast Fourier Transform algorithm is named after J. W. Cooley and John Turkey. It follows the Divide and Conquer Approach and imposes a complexity of $O(n \log n)$.
7. **Karatsuba algorithm for fast multiplication:** It is one of the fastest multiplication algorithms of the traditional time, invented by Anatoly Karatsuba in late 1960 and got published in 1962. It multiplies two n -digit numbers in such a way by reducing it to at most single-digit.

Advantages of Divide and Conquer

- Divide and Conquer tend to successfully solve one of the biggest problems, such as the Tower of Hanoi, a mathematical puzzle. It is challenging to solve complicated problems for which you have no basic idea, but with the help of the divide and conquer approach, it has lessened the effort as it works on dividing the main problem into two halves and then solve them recursively. This algorithm is much faster than other algorithms.
- It efficiently uses cache memory without occupying much space because it solves simple subproblems within the cache memory instead of accessing the slower main memory.
- It is more proficient than that of its counterpart Brute Force technique.
- Since these algorithms inhibit parallelism, it does not involve any modification and is handled by systems incorporating parallel processing.

Disadvantages of Divide and Conquer

- Since most of its algorithms are designed by incorporating recursion, so it necessitates high memory management.
- An explicit stack may overuse the space.
- It may even crash the system if the recursion is performed rigorously greater than the stack present in the CPU.

4.3.3 Dynamic Programming:

Q15. Explain in detail about Dynamic Programming with its applications.

Ans .:

Dynamic programming approach is similar to divide and conquer in breaking down the problem into smaller and yet smaller possible sub-problems. But unlike, divide and conquer, these sub-problems are not solved independently. Rather, results of these smaller sub-problems are remembered and used for similar or overlapping sub-problems.

Dynamic programming is used where we have problems, which can be divided into similar sub-problems, so that their results can be re-used. Mostly, these algorithms are used for optimization. Before solving the in-hand sub-problem, dynamic algorithm will try to examine the results of the previously solved sub-problems. The solutions of sub-problems are combined in order to achieve the best solution.

So we can say that.,

- The problem should be able to be divided into smaller overlapping sub-problem.
- An optimum solution can be achieved by using an optimum solution of smaller sub-problems.
- Dynamic algorithms use Memoization.

Comparison

In contrast to greedy algorithms, where local optimization is addressed, dynamic algorithms are motivated for an overall optimization of the problem.

In contrast to divide and conquer algorithms, where solutions are combined to achieve an overall solution, dynamic algorithms use the output of a smaller sub-problem and then try to optimize a bigger sub-problem. Dynamic algorithms use Memoization to remember the output of already solved sub-problems.

Example

The following computer problems can be solved using dynamic programming approach:

- Fibonacci number series
- Knapsack problem
- Tower of Hanoi
- All pair shortest path by Floyd-Warshall
- Shortest path by Dijkstra
- Project scheduling

Dynamic programming can be used in both top-down and bottom-up manner. And of course, most of the times, referring to the previous solution output is cheaper than recomputing in terms of CPU cycles.

The various applications of Dynamic Programming are:

- Longest Common Subsequence.
- Finding Shortest Path.
- Finding Maximum Profit with other Fixed Constraints.
- Job Scheduling in Processor.
- Bioinformatics.
- Optimal search solutions.

Short Question and Answers

1. What is searching? Explain its types?

Ans :

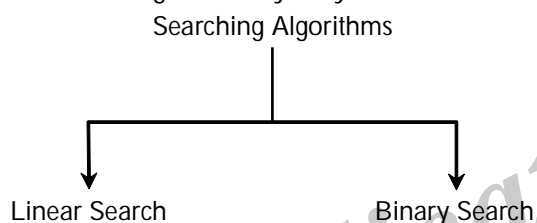
Searching

- Searching is a process of finding a particular element among several given elements.
- The search is successful if the required element is found.
- Otherwise, the search is unsuccessful.

Searching Algorithms

Searching Algorithms are a family of algorithms used for the purpose of searching.

The searching of an element in the given array may be carried out in the following two ways-



- Linear Search
- Binary Search

2. Differences between Divide and Conquer Method and Dynamic Programming.

Ans :

Divide and Conquer Method	Dynamic Programming
1. It deals (involves) three steps at each level of recursion: Divide the problem into a number of subproblems. Conquer the subproblems by solving them recursively. Combine the solution to the subproblems into the solution for original subproblems.	1. It involves the sequence of four steps: <ul style="list-style-type: none"> ➤ Characterize the structure of optimal solutions. ➤ Recursively defines the values of optimal solutions. ➤ Compute the value of optimal solutions in a Bottom-up minimum. ➤ Construct an Optimal Solution from computed information.
2. It is Recursive.	2. It is non Recursive.
3. It does more work on subproblems and hence has more time consumption.	3. It solves subproblems only once and then stores in the table.
4. It is a top-down approach.	4. It is a Bottom-up approach.
5. In this subproblems are independent of each other.	5. In this subproblems are interdependent.
6. For example: Merge Sort & Binary Search etc.	6. For example: Matrix Multiplication.

3. Write the advantages and disadvantages.*Ans :***Binary Search Algorithm Advantages**

The advantages of binary search algorithm are:

- It eliminates half of the list from further searching by using the result of each comparison.
- It indicates whether the element being searched is before or after the current position in the list.
- This information is used to narrow the search.
- For large lists of data, it works significantly better than linear search.

Binary Search Algorithm Disadvantages

The disadvantages of binary search algorithm are:

- It employs recursive approach which requires more stack space.
- Programming binary search algorithm is error prone and difficult.
- The interaction of binary search with memory hierarchy i.e. caching is poor.

4. Write the comparison between Bubble sort and Selection sort*Ans :***Comparison Chart**

Basis for comparison	Bubble sort	Selection sort
Basic	Adjacent element is compared and swapped	Largest element is selected and swapped with the last element (in case of ascending order).
Best case time complexity	$O(n)$	$O(n^2)$
Efficiency	Inefficient	Improved efficiency as compared to bubble sort
Stable	Yes	No
Method	Exchanging	Selection
Speed	Slow	Fast as compared to bubble sort

5. Differences between Linear search and Binary search*Ans :*

Basis of comparison	Linear search	Binary search
Definition	The linear search starts searching from the first element and compares each element with a searched element till the element is not found.	It finds the position of the searched element by finding the middle element of the array.
Sorted data	In a linear search, the elements don't need to be arranged in sorted order.	The pre-condition for the binary search is that the elements must be arranged in a sorted order.

Implementation	The linear search can be implemented on any linear data structure such as an array, linked list, etc.	The implementation of binary search is limited as it can be implemented only on those data structures that have two-way traversal.
Approach	It is based on the sequential approach.	It is based on the divide and conquer approach.
Size	It is preferable for the small-sized data sets.	It is preferable for the large-size data sets.
Efficiency	It is less efficient in the case of large-size data sets.	It is more efficient in the case of large-size data sets.
Worst-case scenario	In a linear search, the worst-case scenario for finding the element is $O(n)$.	In a binary search, the worst-case scenario for finding the element is $O(\log_2 n)$.
Best-case scenario	In a linear search, the best-case scenario for finding the first element in the list is $O(1)$.	In a binary search, the best-case scenario for finding the first element in the list is $O(1)$.
Dimensional array	It can be implemented on both a single and multidimensional array.	It can be implemented only on a multidimensional array.

6. Discuss about Linear search complexities

Ans:

The linear search algorithm iteratively searches all elements of the array. It has the best execution time of one and the worst execution time of n , where n is the total number of items in the search array.

It is the simplest search algorithm in data structure and checks each item in the set of elements until it matches the searched element till the end of data collection. When the given data is unsorted, a linear search algorithm is preferred over other search algorithms.

Complexities in linear search are given below:

Space Complexity

Since linear search uses no extra space, its space complexity is $O(n)$, where n is the number of elements in an array.

Time Complexity

Best-case complexity = $O(1)$ occurs when the searched item is present at the first element in the search array.

Worst-case complexity = $O(n)$ occurs when the required element is at the tail of the array or not present at all.

Average- case complexity = average case occurs when the item to be searched is in somewhere middle of the Array.

7. Discuss about Linear search complexities

Ans :

Binary Search

This algorithm locates specific items by comparing the middlemost items in the data collection. When a match is found, it returns the index of the item. When the middle item is greater than the search

item, it looks for a central item of the left sub-array. If, on the other hand, the middle item is smaller than the search item, it explores for the middle item in the right sub-array. It keeps looking for an item until it finds it or the size of the sub-arrays reaches zero.

Binary search needs sorted order of items of the array. It works faster than a linear search algorithm. The binary search uses the divide and conquers principle.

Run-time complexity = $O(\log n)$

Complexities in binary search are given below:

The worst-case complexity in binary search is $O(n \log n)$.

The average case complexity in binary search is $O(n \log n)$

Best case complexity = $O(1)$.

8. What is sorting?

Ans.:

Sorting refers to the operation or technique of arranging and rearranging sets of data in some specific order. A collection of records called a list where every record has one or more fields. The fields which contain a unique value for each record is termed as the key field. For example, a phone number directory can be thought of as a list where each record has three fields - 'name' of the person, 'address' of that person, and their 'phone numbers'. Being unique phone number can work as a key to locate any record in the list.

Sorting is the operation performed to arrange the records of a table or list in some order according to some specific ordering criterion. Sorting is performed according to some key value of each record.

The records are either sorted either numerically or alphanumerically. The records are then arranged in ascending or descending order depending on the numerical value of the key. Here is an example, where the sorting of a lists of marks obtained by a student in any particular subject of a class.

9. List the categories of sorting

Ans.:

The techniques of sorting can be divided into two categories. These are:

- Internal Sorting
- External Sorting

Internal Sorting

If all the data that is to be sorted can be adjusted at a time in the main memory, the internal sorting method is being performed.

External Sorting

When the data that is to be sorted cannot be accommodated in the memory at the same time and some has to be kept in auxiliary memory such as hard disk, floppy disk, magnetic tapes etc, then external sorting methods are performed.

10. List out the sorting techniques? And How to calculate the complexities of sorting techniques?

Ans :

Sorting techniques:

- Bubble Sort
- Selection Sort
- Merge Sort
- Insertion Sort
- Quick Sort
- Heap Sort

The complexity of sorting algorithm calculates the running time of a function in which 'n' number of items are to be sorted. The choice for which sorting method is suitable for a problem depends on several dependency configurations for different problems. The most noteworthy of these considerations are:

- The length of time spent by the programmer in programming a specific sorting program.
- Amount of machine time necessary for running the program.
- The amount of memory necessary for running the program.

Choose the Correct Answers

1. What is an external sorting algorithm? [a]
 - (a) Algorithm that uses tape or disk during the sort
 - (b) Algorithm that uses main memory during the sort
 - (c) Algorithm that involves swapping
 - (d) Algorithm that are considered 'in place'
2. What is an internal sorting algorithm? [b]
 - (a) Algorithm that uses tape or disk during the sort
 - (b) Algorithm that uses main memory during the sort
 - (c) Algorithm that involves swapping
 - (d) Algorithm that are considered 'in place'
3. What is the worst-case complexity of bubble sort? [d]
 - (a) $O(n \log n)$
 - (b) $O(\log n)$
 - (c) $O(n)$
 - (d) $O(n^2)$
4. The given array is $arr = \{1, 2, 4, 3\}$. Bubble sort is used to sort the array elements. How many iterations will be done to sort the array? [a]
 - (a) 4
 - (b) 2
 - (c) 1
 - (d) 0
5. Which of the following is not a stable sorting algorithm? [b]
 - (a) Insertion sort
 - (b) Selection sort
 - (c) Bubble sort
 - (d) Merge sort
6. If the given input array is sorted or nearly sorted, which of the following algorithm gives the best performance? [a]
 - (a) Insertion sort
 - (b) Selection sort
 - (c) Quick sort
 - (d) Merge sort
7. Which of the following is not an in-place sorting algorithm? [d]
 - (a) Selection sort
 - (b) Heap sort
 - (c) Quick sort
 - (d) Merge sort
8. Which of the following algorithm pays the least attention to the ordering of the elements in the input list? [b]
 - (a) Insertion sort
 - (b) Selection sort
 - (c) Quick sort
 - (d) None

9. If the given input array is sorted or nearly sorted, which of the following algorithm gives the best performance? [a]
- (a) Insertion sort (b) Selection sort
- (c) Quick sort (d) Merge sort
10. Consider the situation in which assignment operation is very costly. Which of the following sorting algorithm should be performed so that the number of assignment operations is minimized in general? [b]
- (a) Insertion sort (b) Selection sort
- (c) Heap sort (d) None

Rahul Publications

Fill in the Blanks

1. _____ is a stable sorting algorithm.
2. _____ is not a stable sorting algorithm.
3. _____ is not an in-place sorting algorithm.
4. _____ is not a non-comparison sort.
5. _____ algorithm pays the least attention to the ordering of the elements in the input list.
6. Time complexity of bubble sort in best case is _____.
7. _____ algorithms have lowest worst-case time complexity.
8. _____ sorting algorithms is stable.
9. Counting sort performs _____ Numbers of comparisons between input elements.
10. _____ sorting algorithm has the running time that is least dependant on the initial ordering of the input.

ANSWERS

1. Merge sort
2. Selection sort
3. Merge sort
4. Shell sort
5. Selection sort
6. $\theta(n)$
7. Heap sort
8. Counting sort (or) Bucket sort (or) Radix sort
9. $O(Zero)$
10. Selection sort

One Mark Answers

1. What are the advantages of Selection Sort?

Ans:

It is simple and easy to implement and It can be used for small data sets.

2. List out sorting techniques

Ans:

Bubble Sort, Selection Sort, Merge Sort, Insertion Sort, Quick Sort and Heap Sort.

3. Define internal sorting

Ans:

If all the data that is to be sorted can be adjusted at a time in the main memory, the internal sorting method is being performed.

4. Define external sorting.

Ans :

When the data that is to be sorted cannot be accommodated in the memory at the same time and some has to be kept in auxiliary memory such as hard disk, floppy disk, magnetic tapes etc.

5. What is sorting?

Ans :

Sorting refers to the operation or technique of arranging and rearranging sets of data in some specific order.

6. List Some Applications of Multilinked Structures?

Ans:

Sparse matrix and Index generation.

7. List out some of the examples of sorting in real-life scenarios

Ans :

Telephone Directory: The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily.

Dictionary: The dictionary stores words in an alphabetical order so that searching of any word becomes easy.

8. Why is sorting important?

Ans:

Efficient sorting is important for optimizing the efficiency of algorithms.

9. What is searching

Ans :

Searching is a process of finding a particular element among several given elements

10. What is bubble sort?

Ans:

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order.

FACULTY OF SCIENCE
B.Sc. III-Year V-Semester (CBCS) Examination
Model Paper - I
DATA STRUCTURES & ALGORITHMS
PAPER - V : (DATA SCIENCE)

Time: 3 Hours

Max. Marks: 80

SECTION - A ($8 \times 4 = 32$ Marks)

[Short Answer Type]

Note: Answer any **EIGHT** questions. All questions carry equal marks.

- | | |
|----------------------------------------------------------------------|-------------------|
| 1. What is a Data Structure? Give applications of Data Structures? | (Unit-I, SQA-7) |
| 2. What is Time complexity give with an example? | (Unit-I, SQA-3) |
| 3. What is a linked list Data Structure? | (Unit-I, SQA-8) |
| 4. What is a Tree data structure and How to represent a Binary tree? | (Unit-II, SQA-1) |
| 5. What are the Properties of a BST Tree. | (Unit-II, SQA-2) |
| 6. Write tree traversal techniques with example. | (Unit-II, SQA-4) |
| 7. What is Graph? Give an example. | (Unit-III, SQA-1) |
| 8. Differences between BFS and DFS. | (Unit-III, SQA-3) |
| 9. What are the components of a Graph | (Unit-III, SQA-5) |
| 10. What is searching? Explain its types? | (Unit-IV, SQA-1) |
| 11. Write the advantages and disadvantages of Binary search. | (Unit-IV, SQA-3) |
| 12. What is sorting? | (Unit-IV, SQA-8) |

SECTION - B ($4 \times 12 = 48$ Marks)

[Essay Answer Type]

Note: Attempt **ALL** questions. All questions carry equal marks.

- | | |
|-------------------------------------------------------------------------------------------------|---------------------|
| 13. (a) What is Performance Analysis of an algorithm? | (Unit-I, Q.No. 1) |
| (OR) | |
| (b) Explain about Linked List implementation of Queue. | (Unit-I, Q.No. 18) |
| 14. (a) What are the basic operations And Traversal Techniques of Binary Tree? | (Unit-II, Q.No. 2) |
| (OR) | |
| (b) What is a Tree data structure and How to represent a Binary tree? Explain its applications. | (Unit-II, Q.No. 1) |
| 15. (a) What is Graph? Explain in detail about Graphs | (Unit-III, Q.No. 1) |
| (OR) | |
| (b) Explain about DFS Search Algorithm. | (Unit-III, Q.No. 4) |
| 16. (a) Explain in detail about Linear Search with an example? | (Unit-IV, Q.No. 1) |
| (OR) | |
| (b) Explain in detail about Dynamic Programming with its applications. | (Unit-IV, Q.No. 13) |

FACULTY OF SCIENCE
B.Sc. III-Year V-Semester (CBCS) Examination
Model Paper - II
DATA STRUCTURES & ALGORITHMS
PAPER - V : (DATA SCIENCE)

Time: 3 Hours

Max. Marks: 80

SECTION - A (8 × 4 = 32 Marks)**[Short Answer Type]****Note:** Answer any **EIGHT** questions. All questions carry equal marks.

1. What is Space complexity give with an example? (Unit-I, SQA-1)
2. What is Hash Table? (Unit-I, SQA-3)
3. Describe the types of Data Structures? (Unit-I, SQA-5)
4. What are the Properties of a BST Tree. (Unit-II, SQA-2)
5. What are the Applications of Tree data structure. (Unit-II, SQA-3)
6. Write tree traversal techniques with example. (Unit-II, SQA-4)
7. Difference between Prim's Algorithm and Kruskal's Algorithm. (Unit-III, SQA-2)
8. What is Minimum Spanning tree? (Unit-III, SQA-4)
9. What is the difference between directed graph and non-directed graph? (Unit-III, SQA-6)
10. List the categories of sorting. (Unit-IV, SQA-9)
11. Write the comparison between Bubble sort and Selection sort. (Unit-IV, SQA-4)
12. List out the sorting techniques? And How to calculate the complexities of sorting techniques? (Unit-IV, SQA-10)

SECTION - B (4 × 12 = 48 Marks)**[Essay Answer Type]****Note:** Attempt **ALL** questions. All questions carry equal marks.

13. (a) What is Space complexity give with an example? (Unit-I, Q.No. 2)
(OR)
(b) What is Hash Table? Explain its representation and Compression method. (Unit-I, Q.No. 20)
14. (a) Explain in detail about Binary Tree Traversal techniques. (Unit-II, Q.No. 3)
(OR)
(b) Give an example how to construct an AVL Tree?
Construct an AVL tree having the following elements
H, I, J, B, A, E, C, F, D, G, K, L (Unit-II, Q.No. 6)

15. (a) Explain about BFS Search Algorithm. **(Unit-III, Q.No. 5)**
(OR)
(b) Explain about Dijkstra's Shortest Path Algorithm. **(Unit-III, Q.No. 10)**
16. (a) Explain in detail about Merge Sort with an example and its complexity. **(Unit-IV, Q.No. 9)**
(OR)
(b) Explain in detail about Greedy Algorithm with its applications. **(Unit-IV, Q.No. 11)**

FACULTY OF SCIENCE
B.Sc. III-Year V-Semester (CBCS) Examination
Model Paper - III
DATA STRUCTURES & ALGORITHMS
PAPER - V : (DATA SCIENCE)

Time: 3 Hours

Max. Marks: 80

SECTION - A ($8 \times 4 = 32$ Marks)

[Short Answer Type]

Note: Answer any **EIGHT** questions. All questions carry equal marks.

1. Write the compression methods using Hash Table. (Unit-I, SQA-4)
2. Describe the types of Data Structures? (Unit-I, SQA-5)
3. What is a Data Structure? Give applications of Data Structures? (Unit-I, SQA-7)
4. What are the Applications of Tree data structure. (Unit-II, SQA-3)
5. What is B Tree? Explain about its operations. (Unit-II, SQA-6)
6. What are the applications of Graph data structure? (Unit-II, SQA-8)
7. What are weighted graphs? (Unit-III, SQA-7)
8. What is Graph? Give an example? (Unit-III, SQA-1)
9. What are the components of a Graph? (Unit-III, SQA-5)
10. What are weighted graphs?. (Unit-IV, SQA-7)
11. Explain about BFS Search Algorithm. (Unit-IV, SQA-9)
12. Explain about DFS Search Algorithm. (Unit-IV, SQA-10)

SECTION - B ($4 \times 12 = 48$ Marks)

[Essay Answer Type]

Note: Attempt **ALL** questions. All questions carry equal marks.

13. (a) Explain singly linked list and its operations. (Unit-I, Q.No. 8)
 (OR)
 (b) What is Sparse Matrix? Explain the representation of Sparse Matrix. (Unit-I, Q.No. 13)
14. (a) What are the basic operations and Traversal Techniques of Binary Tree? (Unit-II, Q.No. 2)
 (OR)
 (b) What is B Tree? Explain in detail about its operations. (Unit-II, Q.No. 6)
15. (a) What is Spanning Tree? Explain about prim's Algorithm. (Unit-III, Q.No. 7)
 (OR)
 (b) What is Spanning Tree? Explain about Kruskal's Algorithm. (Unit-III, Q.No. 8)
16. (a) Explain in detail about Quick Sort with an example and its complexity. (Unit-IV, Q.No. 8)
 (OR)
 (b) Explain in detail about Divide and Conquer Algorithm with its applications. (Unit-IV, Q.No. 12)