Date :

DSA

COMPLETE NOTES ---

- Prepared By :- TopperWorld

FOLLOW US ---

- Website- Topperworld.in
- linkedin- Topperworld
- Instagram- Topperworld.in

• Index •

Date :

Date :

Date :

Date :

2. Introduction to Data Structure

1. Overview of data structures and their significance

- Data structures are fundamental tools in computer and science that allows us to effectively organize and manipulate data.
- They provide a way to store and retrieve data, perform operations on the data, and represent relationships between different pieces of data
- By choosing appropriate data structures, we can optimize the effectively of algorithms and improve the performance of software systems.
- The significance of data structures lies in their ability to:

1. Efficient data storage

- Data structures enables us to store large volumes of data in a structured manner, ensuring quick access and retrieval.

Date :

2. Efficient data retrieval

- They allow us to access and retrieve data elements efficiently, reducing the time complexity of search operations

3. Data organization

- Data structures help organize data, making it easier to understand and manage complex relationships between different data elements.

4. Algorithm design

- Data structures helps and play a crucial role in designing efficient algorithms, as the choice of data structure can significally impact the overall algorithm's performance.

5. Memory Management

- Efficient memory usage is a crucial in software development and data structures aid in managing memory effectively.

• Linear data structure

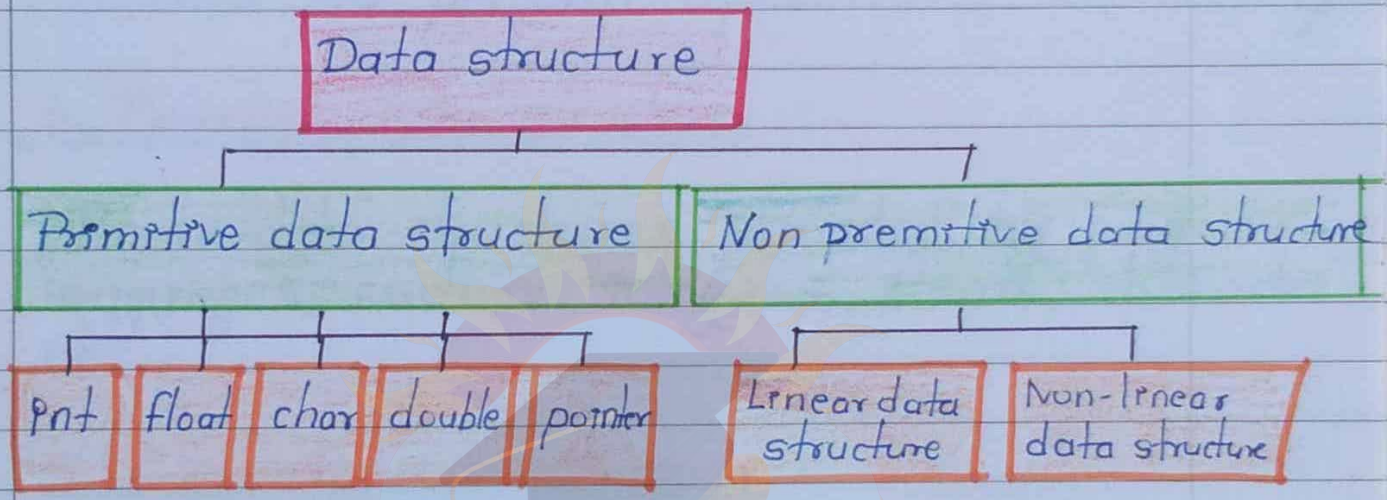
- The arrangement of data in sequential manner is known as Linear Data structure.
Example - Arrays, Stacks, Queue, Linked List

Date :

• Non-linear data structure :

• When one element is connected to 'n' number of elements it is known as Non Linear Datastructure.
Example- Trees, graphs

• Classification of data structure



• Basic terminologies and concepts in data structure

• Before diving deeper into specific data structures, its essential to understand some fundamental termendogies and concepts.

1. Data

• Any piece of information that can be processed or manipulated by a computer program.

Date :

2. Data element (or Node)

- A single unit of data, which may have one or more attributes

3. Data structure

- A way of organizing and storing data elements to perform operations efficiently.

4. Primitive data structure

- Basic data structures provided by programming languages, such as integers, floats, characters etc.

5. Composite data structure

- Data structures built by combining primitive data structures like arrays, linked list and trees.

6. Operations

- The actions performed on data structures, such as insertion, deletion, search, update etc

7. Linear data structure

- Data elements are organized in a linear sequence, such as arrays, linked lists, stacks

Date :

and queues

8. Non-linear data structures

- Data elements are organized hrerarchially such as trees and groups.

9. static data structures

- Fixed size data structures where the size cannot be changed after creation.

10. Dynamic data structures

- Data structures that can grow or shrink in size during program execution.

• Time and space complexity Analysis

- Time complexity is a measure of how the running time of an algorithm increases with the size of the input

- It allows us to understand how efficiently an algorithm performs as the input size approaches infinity

- It describes how the algorithm behaves for large inputs grows larger

- The time complexity of an algorithm is expressed using Bigo notation, which

Date :

provides an upper bound on the growth rate of the algorithm's running time.

- **Time complexity**

- Time complexity focuses on the growth rate of the algorithm's running time as the input size approaches infinity

- It describes how the algorithm behaves for large inputs

- **Space Complexity**

- Time complexity often considers the worst-case scenario, where the algorithm takes the maximum amount of time to execute for any given input.

- **Big O Notation**

- The Big O notation, an algorithms time complexity is represented as O(f(n)), where "f(n)" is a function describing the upper bound on the growth rate concerning the input size 'n'

- The notation "O" indicates an upper bound.

Date :

• Order of complexity

• Common time complexity classes include O(1) (Constant time), O(log n) (logarithmic time), O(n) (linear time), O(n log n) (linearithmic time), O(n^2) (quadratic time), O(2^2) (exponential time) and more.

• Space complexity

• Space complexity is a measure of how much additional memory (space) an algorithm or data structure requires to solve a problem

• As a function of the input size, it helps us to understand the efficiency of memory usage by the algorithm

• The space complexity of an algorithm is expressed using Big O notation.

• Example of space complexity

```
int calculatorSum (int n) {
    int * arr = new int [n]; // Allocate array
                                of size 'n'
    int sum = 0;
    for (int i=0; i<n; i++) {
        arr [i] = i+1;
        sum += arr [i];
    }
```

Date :

```
    delete [] arr; //Deallocate the array
    return sum;
}
```

• Space complexity analysis

- The function allocates an array of size "n" using dynamic memory allocation (keyword)

- The space complexity is O(n) because the size of array is directly proportional to the input size 'n'.

- The space complexity is primarily determined by the additional memory used to store the array of size "n".

• Advantages of Data structure

① Efficiency
② Reusability
③ Abstraction

• Efficiency-

- Data structure is a secure way of storing data on our system.

- It helps us to process data easily.

Date :

- Reusable

- Data structures are reusable as once we have to implemented particular data structure, we can use it at any place

- Abstraction

- It allows users to work with data structures without having to implementation details, which simply programming

- Operations on Data structure

Traversing

- We can access an element of data structure at least once

Searching

- We can easily search for data element in data structure

Sorting

- We can sort elements either in ascending or descending order

Date :

- Insertion

- We can insert new data elements in data structure

- Deletion

- We can delete data elements from data structure

- Updation

- We can update or replace existing elements from data structure

Date :

2. Concepts of Algorithm

- What is an algorithm ?

- An algorithm is a set of commands that must be followed for a computer to perform calculations

- Characteristics of an algorithm

Input

- Algorithm take input data, which can be in various formats such as number, text

Processing

- The algorithm processes the input data through series of logical and mathematical operations

Output

- After processing is complete, algorithm produces output

Efficiency

- Aiming to accomplish tasks quickly

Date :

• Optimization

• optimize algorithm to make them fast and reliable

• Implementation

• Algorithm are implemented in various programming languages, enabling comp--uters to execute them and produce outputs.

Algorithm Approaches

① Brute Force Algorithm
② Divide and conquer
③ Greedy Algorithm

• Brute Force Algorithm

• A brute force algorithm solves a problem through exhaustion : it goes through all possible choices until a solution is found.

• The time complexity of a brute force algorithm is often proportional to the input size.

• Brute force algorithms are simple and consistent, but very slow

Date :

• Divide and conquer Algorithm

• Divide-and conquer algorithm is recursively breaks down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly

• Greedy Algorithm

• An algorithm that finds a solution to problems in the shortest time possible

• It picks the path that seems optimal at the moment without regard for the overall optimization of the solution that would be formed

• Algorithm Analysis

• Algorithm analysis involves studying the efficiency and performance of algorithm in terms of their time complexity and space complexity

• By analyzing algorithms we can determine how they scale with input size and resources, helping us choose the most suitable algorithm for a specific problem

Date :

3. Arrays and Strings

• Array Concept

• Arrays are defined as collection of similar data type

• Array is the simplest data structure where each element can be randomly accessed by using its index number

Array Declaration:

int arr[10]; char arr[10], float arr[n];

• Complexity of Array Operation

1. Time complexity

Algorithm	Average case	Worst case
Access	o(1)	o(1)
Search	o(n)	o(n)
Insertion	o(n)	o(n)
Deletion	o(n)	o(n)

Date :

2. Space Complexity

The array space complexity for worst case is O(n).

- One-dimensional Array and their Operation

- One-Dimensional Array

- A one-dimensional array is a data structure that contains a set of elements of the same data type

- It stores a element in linear management under a single variable name

- Declaration and Initialization

- One dimensional array can be declared and initialized as follows:

```
//Declaration and Initialization of an array
datatype arrayName [size]; // Declaration of an array of 'size' elements of type 'datatype'

int numbers [5]= {10,20,30,40,50}
```

Date :

• Accessing elements :

• Elements of an array can be accessed using their index (starting from 0) :

```
int value = numbers[2]; // Accessing the element at index 2 (value will be 30)
```

• Common Operations

1. Insertion - Inserting an element into an array at specific index.

2. Deletion - Deleting an element from the array at a specific index.

3. Search - Searching for given element in the array

4. Traversal - Visiting each element of the array.

5. Sorting - Arranging element in specific order

• Multidimensional array and Matrices

• Multidimensional Array

• A multidimensional array is an array with more than one level or dimension.

• It is useful when you want to store data as a tabular form

Date :

• Declaration and Initialization

// Declaration and initialization of a 2D array
dataType arrayname [rows][columns];

// Example of 2D inter array initialization

```
int matrix [3][3] = {
    {1, 2, 3}
    {4, 5, 6}
    {7, 8, 9}};
```

• Accessing elements

• Elements in 2D array can be accessed using row and column indices

int value = matrix [1][2]; // Accessing element at row 1 column 2 (value will be 6)

• Common Operations :

• Transpose : Converting rows into column and vice versa.

• Addition and subtraction : Performing arithmetic operations on two matrices

• Multiplications : Multiplying two matrices

Date :

• Strings and string - manipulation techniques

• A string is an array of characters, terminated by the null character '\0'.

• c++ provides various library functions to manipulate string efficiently.

• String manipulation basically refers to the process of handling and analyzing strings

• Declaration and initialization

```
//Declaration and initialization of strings
char str1[] = "Hello"; // Automatically
                       includes the null values
char str2[10]; //Declaration of an empty
character array with enough with
space for 9 characters (+1 null chara-
-cters)
// c++ string (std::string) declaration
#include <string>
std::string myString = "Hello world";
```

• Common string manipulation techniques

1. Length: Getting the length of a string

• The number of characters it contains not counting the null.

Date :

```
int length = str.length(); //using the length()
method
```

2. Concatenation: Combining two strings

- Concatenation is the process of combining two or more strings into a single string

```
std::String str1 = "Hello";
std::string str2 = "World!";
std::string result= str1 + str2; //using
the + operator for concatenation.
```

3. Substring: Extracting a portion of a string

- Substring extraction allows us to obtain a part of a string based on the starting index and the length of the substring.

```
std::string str = "Hello World!";
std::string substring = str.substr(0,5);
//Extract substring from index 0 (inclusive)
to 5 (exclusive)
```

4. Comparison: Comparing the strings

- Comparison allows us to check if two strings are equal or determine their relative order

Date :

5. Modification: changing or replacing parts of a string

- String modification allows us to change or replace parts of a string

```
std::string str = "Welcome to DSA";
// Reclacing a string with another string
str.replace(0, 8, "Hello");
```

Date :

4. linked list

• Linked List Concept

• A linked list is the most sought - after data stucture when it comes to handling dynamic data elements

• A linked list consists of a data element known as a node.

• Each node consists of two fields:

• One field has data and in second field the node has an address that keeps a reference to the next node.

• Singly Linked List and their operations

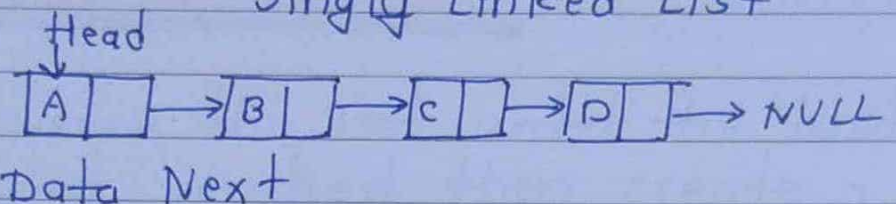
• It is the simplest type of linked list in which every node contains some data and a pointer to the next node of the same data type.

• In singly linked list, the individual element is called as "Node".

• Every "Node" contains two fields, data field and the next field.

Date :

Singly Linked List



- Null means it address part does not point to any node.
- The pointer that holds address of initial node is known as a header pointer

- **Structure of the singly Linked List:**

```
class Node {
public:
    int data
        //pointer to next node in LL
    Node* next;
};
```

- **Basic Operation**

- Insertion - The insertion operation are performed in three ways

① Insert at the beginning

- Create a new node with the given data and set its next pointer to the current head of the list, then update the head to point to the new node.

Date :

- Insert at the end:

- Traverse the list until the last node is created, reached, then create a new node with the given data and set the last nodes next pointer to the new node.

- Insert at a specific position:

- Traverse the list until reaching the node before the desired position. Then create a new node with the given data and set its next pointer to the next node, update the previous node next pointer to pointer to point the new node

- Deletion - There are also three main types of deletion in a singly linked list

- Delete at the beginning

- Update the head to point to the next node of and delete the previous head node

- Delete at the end

- Traverse the list until reaching the node before the desired position

- Update its next pointer to point to the node after the desired position and delete node of the last node.

Date :

- **Delete at the specific position**

- Traverse the list until reaching the node before desired position

- Update its next pointer to point the node after the desired position and delete the node of the desired position

- **Display**

- This process display the elements of a singly-linked list

- **Search**

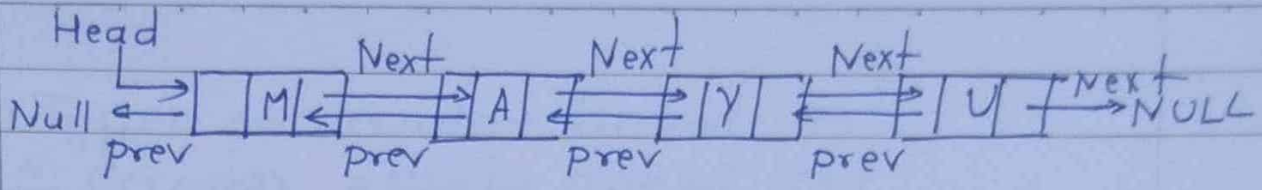
- It is a process of determining and retrieving a specific node either from the front, the end or anywhere in the list

- **Doubly Linked Lists and Circular Linked Lists**

- **Doubly Linked List**

- Traversal of items can be done in both forward and backward directions as every node contains an additional prev pointer that points to the previous node

Date :



Doubly Linked List

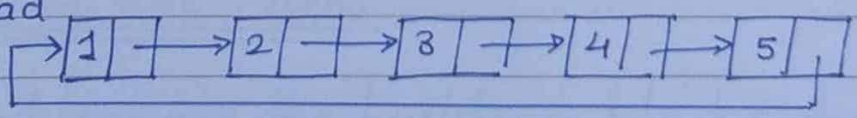
• Representation of Doubly linked list:

```
class Node{
public:
int data;
  Node * next;
  Node * prev;
};
```

• Circular Linked List

• The circular linked list is a linked list where all nodes are connected to form a circle.

• In a circular linked list, the first node and the last node are connected to each other which form a circle.



• Representation of circular Linked List

```
class Node {
    int value;
    Node next; }
```

Date :

- Types of circular linked list
① Circular singly Linked list
② Circular doubly Linked list

- Applications of linked list

- Linked lists are widely used in various applications, due to their dynamic nature and efficient operations

1. Dynamic Memory Allocation

- Linked list are used in memory management especially in cases where the size of data is unknown or may change during program execution

2. Stack and queues

Linked Lists are used to implement stacks and queues which are fundamental data structures used in algorithm design

3. Graph and Trees

Linked list are the building blocks for implementing more complex data structures

4. File system

- It manage the structure of file in file systems where files can be dynamically added or removed.

Date :

5. STACK and QUEUE

• Stack Concept

• Stack is a linear type of data structure that follows the LIFO (Last-In-First-Out) principle and allows insertion and deletion operations from one end of the stack data structure that is top

• Basic Operations on Stack

• push() - To insert an element into the stack

• pop() - To remove an element from the stack

• top() - Returns the top element of the stack

• isEmpty() - Returns true if stack is empty else false

• size() - returns the size of stack

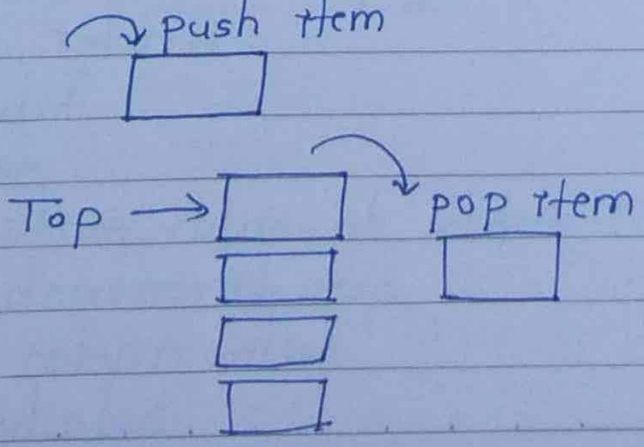


Fig. Stack

Date :

• Push :- Adds an item to the stack, if the stack is full, then it said to be an overflow condition

• Algorithm for push :

```
begin
   if stack is full
      return
   endif
   else
      increment top
      stack [top] assign value
   end else
   end procedure
```

• Pop :- Removes an item from the stack, the items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an underflow condition.

• Algorithm for pop :

```
begin
 if stack is empty
    return
 endif
 else
    store value of stack [top]
    decrement top
    return value
 end else
```

Date :

end procedure

- **Top** :- Returns the top element of the stack

- **Algorithm for Top** :-

```
begin
  return stack [top]
end procedure
```

- **isEmpty** :- Returns true if the stack is empty, else false

- **Algorithm for isEmpty**

```
begin
if top < 1
   return true
else
     return false
end procedure
```

- **Complexity Analysis**

- **Time Complexity**

Operations	Complexity
push()	O(1)
pop()	O(1)
isEmpty()	O(1)
size()	O(1)

Date :

• Types of Stacks

• Fixed Size Stack

• As the name suggests a fixed size stack has a fixed size and cannot grow or shrink dynamically

• Dynamic Size Stack

• A dynamic size stack can grow or shrink dynamically

• Infix to Postfix Stack

• This type of stack is used to convert infix expressions to postfix expressions

• Expression Evaluation Stack

• This type of stack is used to evaluate postfix expression

• Applications of stack

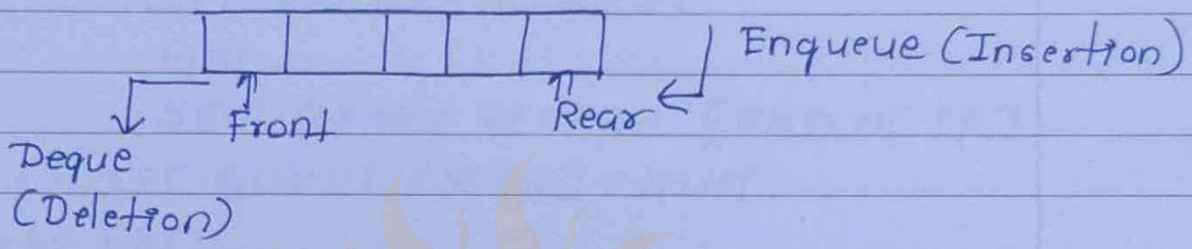
① Recursion
② Depth First Search
③ Backtracking
④ Memory Management

Date :

• Queue Concept

• A Queue can be defined as ordered list which enables insert operations to be performed at once end called REAR and delete operations to be performed at another end called FRONT.

• Queue can be referred as to be first in first out



• Operations on Queue

① Enqueue - Enqueue is used to insert element at rear end of queue. It returns void

② Dequeue - Dequeue operations performs the deletion from front end of queue

③ Peek - This returns element which is pointed by front pointer in the queue but does not delete it.

④ Queue overflow (is Full) - when queue is completely full then it shows overflow condition

⑤ Queue underflow (is empty) - whene there is no element in queue, it throws underflow condition

Date :

• Algorithm to Insert any element in a queue

• check if queue is already full by comparing rear to max-1.

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 queue

Step-1- IF FRONT=-1 OR FRONT > REAR
write UNDERFLOW
ELSE
SET VAL = QUEUE [FRONT]
SET FRONT = FRONT+1
[END OF IF]

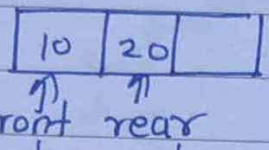
step-2- EXIT

• Types of Queue

① Linear Queue

• In Linear queue, an insertion take place from one end while deletion occurs from another end

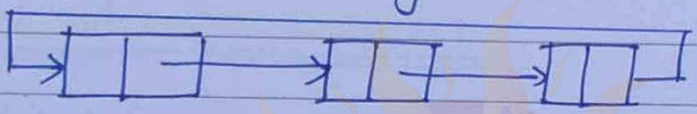
Date :



- The elements are inserted from rear end and if we insert more elements in queue, then rear values get incremented on every insertion.

Circular Queue

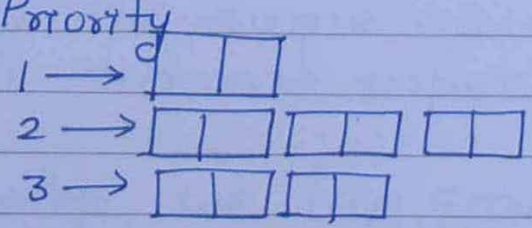
- In circular queue, the last element points to the first element making a circular link



- It provides better utilization

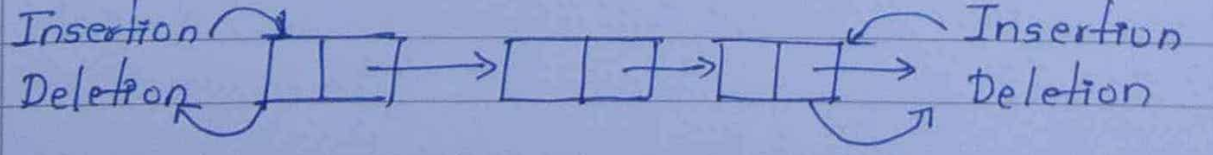
Priority Queue

- It is a type of queue served according to it priority



Deque (Double Ended Queue)

- Insertion and deletion performed on either front or rear



Date :

6. Recursion

• Recursion definition

• The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function

• Need of Recursion

• Recursion is an amazing technique with the help of which we can reduce the length of our code and make it easier to read and write

• Properties of Recursion

• Performing the same operations multiple times with different inputs

• In every step, we try smaller inputs to make the problem smaller

• Base condition is needed to stop the recursion otherwise infinite loop will occur

Date :

• For recursion to work correctly, there are two fundamental principle to follow

1. Base Case- Every recursive function must have a base case which is the simplest form of the problem that can be directly solved without further recursion.

2. Recursive call- In the body of the function, there should be a call to itself with a modified version of the original problem.

• Algorithm steps

• The algorithm steps for implementing recursion in a function are as follows.

Step 1- Define a base case:
• Identify the simplest case for which the solution is known or trivial.

• This is the stopping condition for recursion as it prevents the function from infinitely calling itself.

Step 2- Define a recursive case: Define the problem in terms of smaller subproblems

• Break the problem down into smaller versions of itself and call the function recursively to solve each subproblem

Date :

Step-3- Ensure the recursion terminates:

- Make sure that the recursive function eventually reaches the base case and does not enter an infinite loop

Step-4 -Combine the solutions:

- Combine the solutions of the subproblems to solve the original problem.

- Recursive solution to common problems:

1. Factorial-

- The factorial of a non-negative integer n, denoted by n! is the product of all positive integers less than or equal to n.

Recursive algorithm

```
unsigned int factorial (unsigned int n) {
 if (n==0 || n==1) {
   return 1; // Base case: 0! and 1! both 1
 } else {
   return n* factorial (n-1); // Recursive
   call to solve (n-1)!
  }
}
```

Date :

• Fibonacci series:

- The fibonacci series is a sequence of numbers in which each number (fibonacci number) is the sum of the two proceding ones usually starting with 0 and 1.

Recursive algorithm:

```
unsigned int fibonacci (unsigned int n)
 {
   if (n==0)
    {
      return 0; // Base case: f(0)=0
    }
   else if (n==1)
    {
      return 1;
    }
   else
    {
      return fibonacci (n-1) + fibonacci
        (n-2); // Recursive call to find
          f(n-1) + f(n-2)
    }
 }
```

Date :

7. TREES

• Introduction to tree data structure

- A tree data structure is a hrerarchical structure that is used to represent and organize data in a way that is easy to search and navigate

- It is a collection of nodes that are connected by edges and has a hrerarchical relationship between the nodes

- The topmost node of the tree is called the root, and the nodes below it are called the child nodes

- Each node can have multiple child nodes, and these child nodes can also have their own child nodes, forming a recursive structure

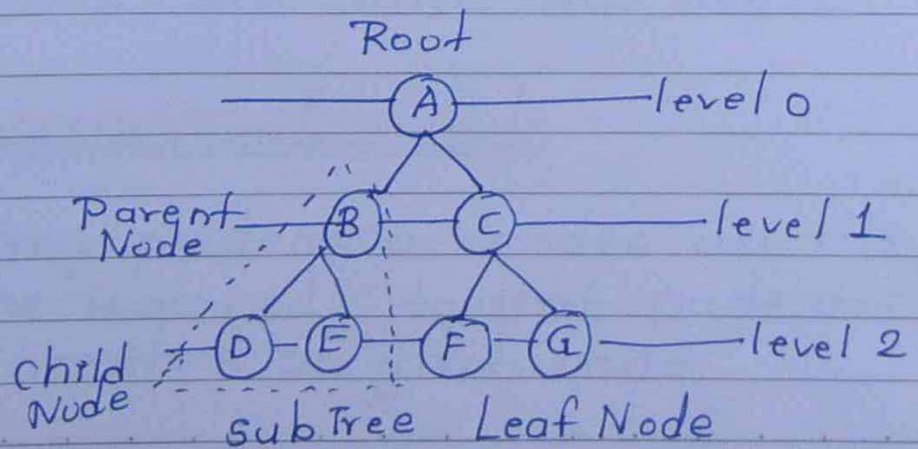


Fig. Tree

Date :

• 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

• Child Node :-

• The node which is the immediate successor of a node is called the child node of that node.

• Root Node :-

• The topmost node of a tree or the node which does not have any parent node is called the root node

• Leaf Node :-

• The node which do not have any child nodes are called leaf nodes.

• Ancestor of a node :-

• Any predecessor nodes on the path of the root node to that node are called ancestors of that node

Date :

- Descendant :-

- Any successor node on the path from the leaf node to that node are the descendants of the node

- Siblings :-

- Children of the same parent node are called siblings

- Level of a node :-

- The count of edges on the path from the root node to that node

- Internal Node :-

- A node with at least one child is called internal node.

- Neighbour of a Node :-

- Parent or child nodes of that node are called neighbors of that node

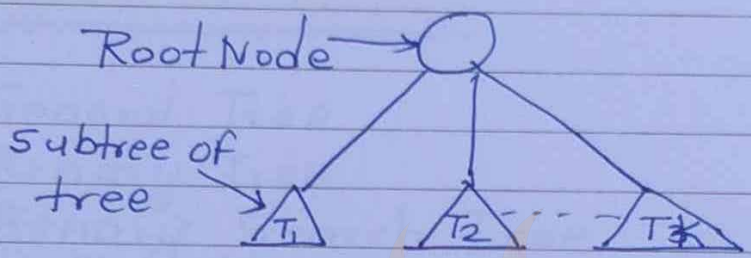
- Subtree :-

- Any node of the tree along with its descendant.

Date :

• Representation of tree data structure

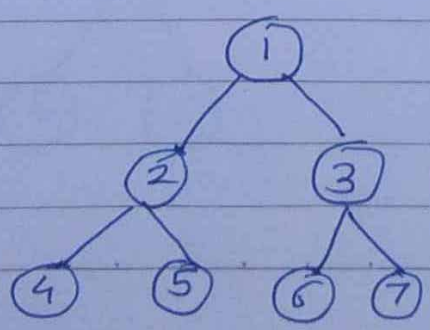
• A tree consists of a root and zero or more subtrees $T_1, T_2 \cdots T_k$ such that there is an edge from the root of the tree to the root of each subtree



• Representation of a Node in Tree Data Structure:

```
struct Node
{
  int data;
  struct Node *first_child;
  struct Node * second_child;
  struct Node * third_child;
    :
  struct Node *nth_child;
};
```

• Example of Tree data structure



Date :

Here,

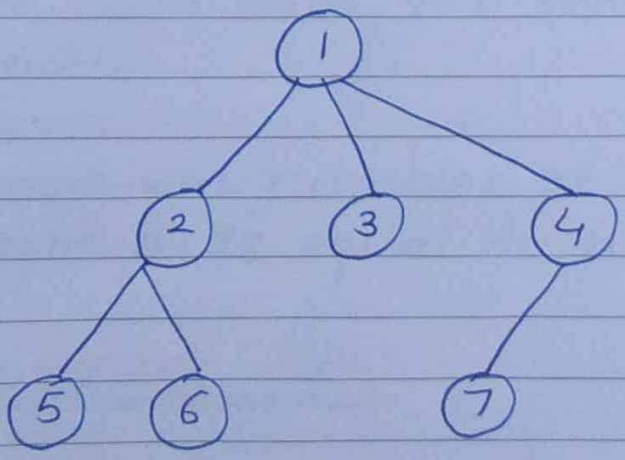
- Node 1 is the root node
- 1 is the parent of 2 and 3
- 2 and 3 are siblings
- 4, 5, 6 and 7 are leaf nodes
- 1 and 2 are the ancestors of 5

• Types of Trees :

① General Tree
② Binary Tree
③ Binary Search Tree
④ Balanced Search Tree
⑤ Ternary Tree
⑥

• General Tree :-

- General tree are unordered tree data structure where the root node has minimum 0 or maximum 'n' subtrees.

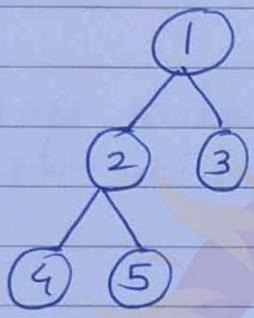


Date :

· Binary Tree :-

· In a binary tree, each node can have a maximum of two children linked to it

· Some common types of binary trees include full binary tree, complete binary tree, balanced binary tree



· Properties of Binary Tree

· At each level of i, maximum number of nodes is 2^i

· The height of the tree is defined as the longest path from the root node to the leaf node

· The maximum number of nodes possible at height h is equal to h+1.

· Full Binary Tree

· A full binary tree is a binary tree type where every node has either 0 or 2 children node.

Date :

• Complete Binary Tree

• A complete binary tree is a binary tree where all leaf nodes must be on same level

• Binary Search Tree (BST)

• A binary search tree is a binary tree with following properties

1. The left subtree of a node contains only nodes with value less than the node's value

2. The right subtree of a node contains only nodes with values greater than the node's value

3. Both the left and right subtrees are also binary search trees

• Balanced Search tree

• Balanced search trees are a special type of binary search trees that are designed to ensure that the tree remains balanced, preventing the trees from becoming skewed and reducing the height of tree to maintain efficient operations

Date :

- There are three types of Balanced search trees:
① AL/L Trees
② Red - Black Trees
③ Splay Trees

- **Ternary Tree**

- A ternary tree is a tree data structure in which each node has at most three child nodes, usually distingurshed as "left", "mid" and "right".

- **Basic Operation of Tree Data structure:**

- **Create** - Create a tree in the data structure

- **Insert** - Insert data in a tree

- **Search** - Searches specific data in a tree to check whether it is present or not.

- **Tree Traversal algorithms (pre-order, in-order, post-order)**

- Traversal is a process to visit all the nodes of a tree and may print their values too.

- Because all the nodes are connected via edges (links) we always start from the root (head) node.

Date :

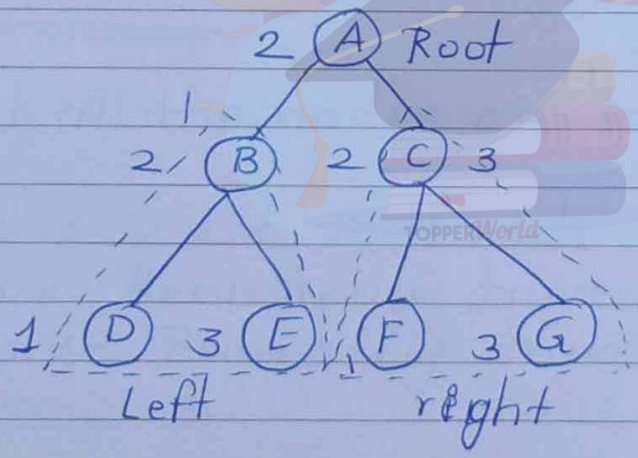
- That is, we cannot randomly access a node in a tree

- There are three ways which we used to traverse a tree -

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

• In-order Traversal

- In this traversal method, the left subtree is visited first then the root and later the right sub-tree



In-order → D → B → E → A → F → C → G

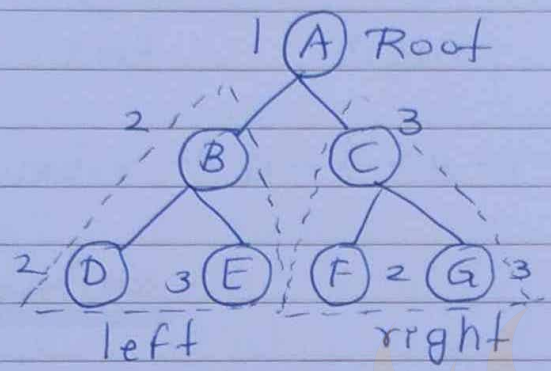
• Algorithm :-

Until all nodes are traversed
step-1 - Recursively traverse left subtree
step-2 - Visit root node
step-3 - Recursively traverse right subtree

Date :

• Pre-order Traversal :-

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



pre-order → A → B → D → E → C → F → G

• Algorithm

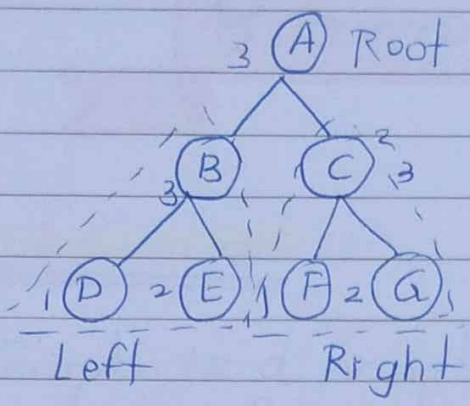
Until all the nodes are traversed

Step-1- Visit root node
Step-2- Recursively traverse left subtree
Step-3- Recursively traverse right subtree

• Post-order Traversal

• In this traversal, the root node is visited last, hence we traverse the left subtree, then the right subtree and finally the root node

Date : _____



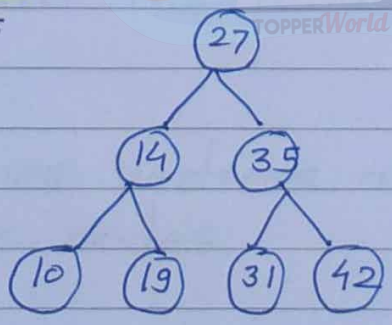
Pre-order → D → E → B → F → G → C → A

Algorithm :-

Until all nodes are traversed

Step-1- Recursively traverse left subtree
Step-2- Recursively traverse right subtree
Step-3- Visit root node

Example -



Pre-order- 27 14 10 19 35 31 42

In-order- 10 14 19 27 31 35 42

Post-order- 10 19 14 31 42 35 47

Date :

8. Graphs

• Graph Concept

• A Graph is a non-linear data structure consisting of vertices and edges

• The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph

• 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 unlabeled

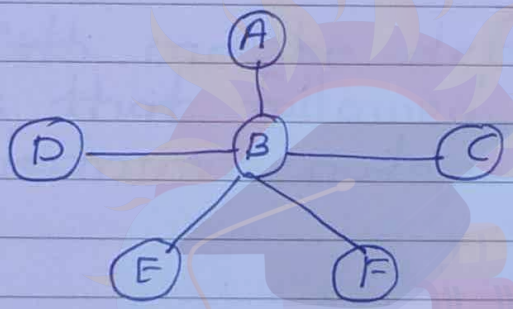
• Edges

• Edges are drawn or used to connect two nodes of the graph

Date :

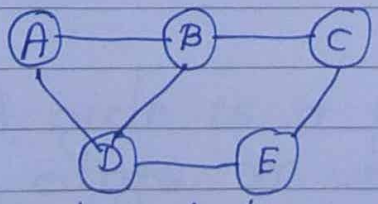
- It can be ordered pair of nodes in a directed graph
- Edges can connect any two nodes in any possible way
- There are no rules
- Sometimes edges are also known as arcs

- **Example of Graph data structure**



- **Directed and Undirected graph**

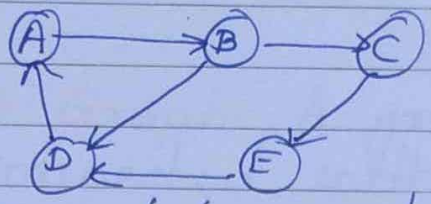
- A graph can be directed or undirected
- However in undirected graph, edges are not associated with directions with them.



Undirected Graph

Date :

· As the above figure edges are not attached with any of the directions



Directed Graph

· In the above figure, directed graph edges form an ordered pair

· Graph Terminology

· Path - Path can be defined as sequence of nodes that followed in order to reach some terminal node v from initial node v

· Closed path - A path will be called as closed if initial node is same as terminal node $V_0 = V_N$

· Simple path - If all node of graph are distinct with an expression $V_0 = V_N$, then such path is called as closed simple path

· cycle - A cycle is a path which has no repeated edges or vertices except first and last vertices

Date :

• Connected Graph- A graph in which some path exists between every two vertices (u, v) in V

• Complete Graph- A graph in which every node is connected with all other nodes

• A complete graph contains $\frac{n(n-1)}{2}$ edges where n is the number of nodes in graph

• Weighted Graph- In this graph each node is assigned with some data such as length or width.

• The weight of an edge can be given as w(e) which must be positive (+) value indicating cost of traversing edge

• Diagraph- A diagraph is directed graph in which each edge is associated with some direction and traversing can be done only in specified direction

• Loop:- An edge that is associated with similar end points can be called as loop

• Adjacent nodes- If two nodes u and v are connected via an edge e, then nodes u and v are called as neighbours

Date :

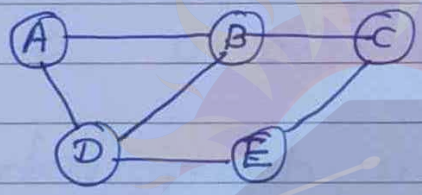
• Degree of a Node

• A degree of a node is a number of edges that are connected with that node

• A node with degree 0 is called isolated

• Sequential Representation

• Undirected Graph



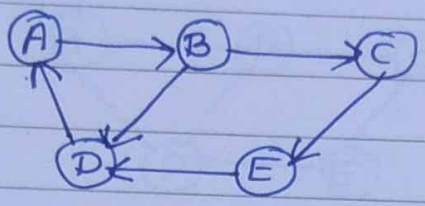
Adjacency matrix

	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

• In above figure, we can see mapping among vertices (A, B, C, D, E) is represented by using adjacency matrix

Date :

• Directed Graph

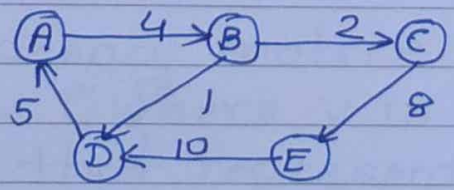


• Adjacency Matrix

	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

• A directed graph and its adjacency matrix representation shown in above figure

• Weighted Graph

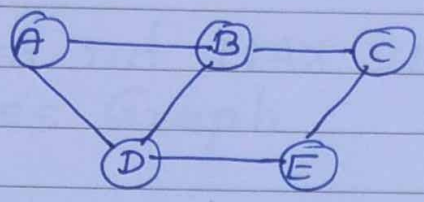


• Adjacency Matrix

	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

Date :

• Linked Representation :-



• Adjacency List

A → B → D X
B → A → D → C X
C → B → E X
D → A → B → E X
E → D → C

• A adjacency list is maintained for each node present in graph which stores node value and a pointer to next adjacent node to respective node

• Adjacency Matrix

• An adjacency matrix is a 2D array of size V x V (where V is the number of vertices) that represents a graph

• If there is an edge between vertices i and j, the matrix cell at (i, j) is marked with a 1 or a weight (for weighted graphs)

• otherwise it contains a 0 or a special

Date :

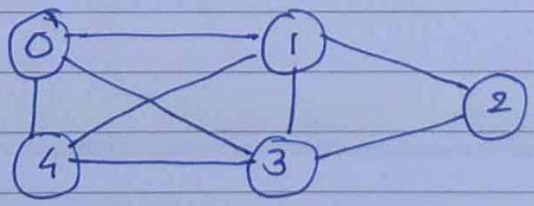
value to represent no edge

```
const int MAX_VERTICES = 100;
class Graph
{
  private:
  int V; // Number of vertices
  bool directed; // whether the graph
  is directed or undirected
  int matrix [MAX_VERTICES]
  [MAX_VERTICES];
  public:
  Graph (int vertices, bool isDirected
  = false): V(vertices), directed
  (isDirected)
  {
    for (int i=0; i<V; ++i)
    {
      for (int j=0; j<V; ++j)
      {
        matrix [i][j]=0;
      }
    }
  }
  void addEdge (int from, int to,
  int weight=1)
  {
    matrix [from][to]=weight;
    if (!directed) {
      matrix [to][from] = weight;
  }
} }
```

Date :

• Adjacency List :-

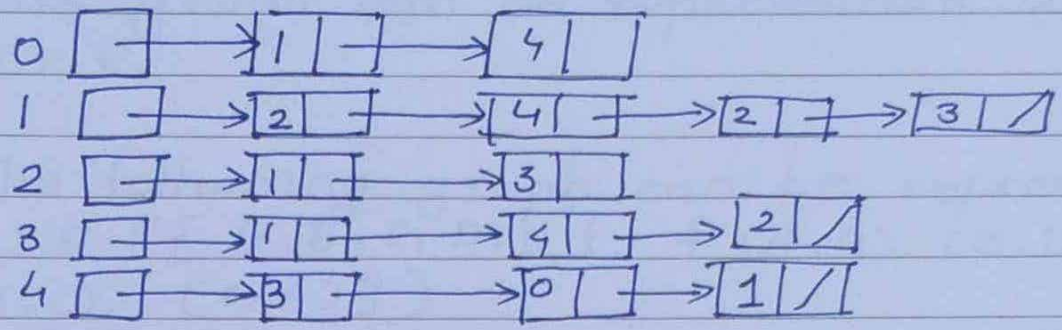
- An array of linked list is used
- The size of the array is equal to the number of vertices
- Let the array be an array [].
- An entry array [i] represents the linked list of vertices adjacent to the i^{th} vertex.
- This representation can also be used to represent a weighted graph
- The weights of edges can be represented as lists of pairs
- Consider the following graph:



- Example of undirected graph with 5 vertices

Date :

- Following is the adjacency list representation of the above graph



- **Advantage of Adjacency List**

- Saves space. Space taken is O(|V|+|E|)

- In the worst case, there can be C(V) number of edges in a graph thus consuming $O(V^2)$ space

- Adding a vertex is easier

- Computing all neighbors of vertex takes optimal time

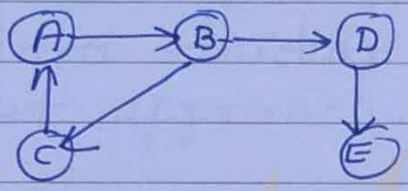
- **Graph Traversal Algorithm**

- The graph is one non-linear data structure

- That is consist of some nodes and their connected edges

Date :

- The edges may be directed or undirected
- This graph can be represented as G(V, E)
- The following graph can be represented as G({A, B, C, D, E}, {(A, B), (B, D), (D, E), (B, C), (C, A)})



- There are two graph traversal algorithms given below:

① BFS (Breadth First Search)
② DFS (Depth First Search)

• Breadth First Search

- Traversing or searching is one of the most used operations that are undertaken while working on graphs
- Therefore, in breadth-first-search (BFS), you start at a particular vertex and the algorithm tries to visit all the next level of traversal of vertices

Date :

- Unlike trees, graph may contain cyclic paths where the first and last vertices are remarkably the same always

- Thus in BFS, you need to keep node of all the track of the vertices you are visiting

- To implement such an order, you use a queue data structure which First-in, First-out approach.

- Algorithm:

1. start putting anyone vertices from the graph at the back of queue

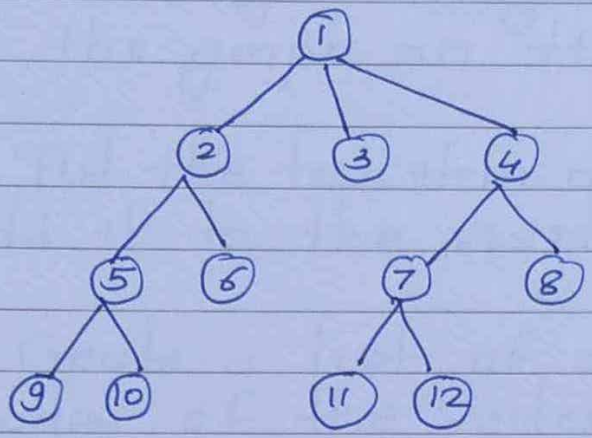
2. First, move the front queue item and add it to be the list of the visited node

3. Next, create nodes of the adjacent vertex of that list and add them which have not been visited yet.

4. keep repeating steps two and three until the queue is found to be empty

Date :

- Complexity :- O(V+E) where V is vertices and E is edges



- Depth First Search :-

- In Depth-First search (DFS), you start by particularly from the vertex and explore as much as you along all the branches before backtracking.

- In DFS, it is essential to keep note of the tracks of visited nodes and for this, you use stack data structure

- DFS finds its application when it comes to finding paths between two vertices and detecting cycles

- Also, topological sorting can be done using the DFS algorithm easily. DFS is also used for one-solution puzzles

Date :

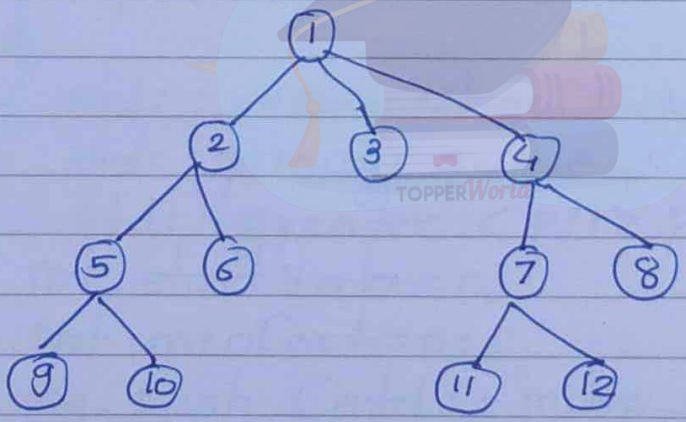
• Algorithm :-

1. start by putting one of the vertexes of the graph on the stack's top

2. Put the top item of the stack and add it to the visited vertex list

3. Create a list of all the adjacent nodes of the vertex and then add these nodes to unvisited at the top of the stack

4. keep representing step 2 and step 3 and the stack becomes empty



• Minimum spanning tree algorithm

• There are different algorithms

① Prim's algorithm
② kruskal's algorithm

Date :

• Prim's Algorithm

- Prim's algorithm is a greedy algorithm
- This algorithm always starts with a single node and moves through several adjacent nodes, in order to explore all of the connected edges along the way

```
#include <iostream>
#include <vector>
#include <queue>
typedef std::pair<int, int> pii; // pair
(weight, vertex)
int primMST(Graph &graph)
{
    std::vector<bool> visited(graph.v, false);
    std::priority_queue<pii, std::vector<pii>,
    std::greater<pii> pq; //Min-Heap
int startVertex = 0;
int minCost=0;
pq.push(std::make_pair(0, start
VERTEX));
while(!pq.empty())
{
    int currentVertex= pq.top().second;
    int weight= pq.top().first;
    pq.pop();
    if(!visited[currentVertex])
    {
```

Date :

```
    visited [currentVertex] = true;
    minCost += weight;
for(const auto & neighbor : graph.adjust
[currentVertex]
{
    int neighboursVertex = neighbor.first;
    int neighborVertex = neighbor.second;
    if(! visited [neighborVertex]
    {
        pq.push (std::make_pair (neighbor
        Weight, nerghbor.vertex));
    }
  }
 }
 return minCost;
}
```

• kruskal's Algorithm

• kruskal's algorithm is a greedy algorithm that finds the minimum spanning tree (MST) of a connected, undirected graph

• The MST is a subset of the edges that connects all the vertices of the graph with the minimum possible total edge weight

Date :

• Algorithm steps:

1. Sort all the edges of the graph in non-decreasing order of their weights

2. Initialize an empty set to store the MST

③ Iterate through the sorted edges, For each edge, if adding it to the MST does not create a ~~edge~~ cycle, including it in MST

④ Continue until there are V-I edges in the MST

```
#include <iostream>
#include <vector>
#include <algorithm>
struct edge
 {
    int from;
     int to;
     int weight;
     edge (int f, int t, int w) from (f), to(t),
     weight(w) { }
  };
  bool compareedges (const Edge & e1,
   const Edge & e2)
   {
     return e1. weight < e2. weight;
   }
```

```
int findParent(int vertex, std::vertex<int>
&parent)
{
  if(parent[vertex] == vertex)
   {
     return vertex;
   }
  return FindParent(parent[vertex], parent);
}
void Union(int x, int y, std::vector<int>
  &parent)
  {
    int xparent = findParent(x, parent);
    int yparent = findParent(y, parent);
     parent[xparent] = yparent;
  }
std::vector<Edge> kruskalMST(std::vector
<Edge>& edges, int v)
{
 std::sort(edges.begin(), edges.end(),
  compareEdges);
  std::vector<int> parent(v);
  for(int i=0; i<v; ++i)
   {
     parent[i] = i;
   }
 std::vector<Edge> MST;
 int edgesAdded = 0;
 for(const auto & edge: edges)
 {
   int from = edge.from;
```

Date :

```
int to = edge.to;
int weight = edge.weight;
if (findParent (from, parent) != findParent
(to, parent))
{
  Union (from, to parent);
   MST. push back (edge);
   edgesAdded ++;
 }
 if (edgesAdded == v-1)
  {
     break;
  }
 }
  return MST;
}
```

- Shortest Path Algorithms

- Dijkstra's Algorithm:

- Dijkstra's algorithm is a greedy algorithm that flows finds the shortest path from a single source vertex to all other vertices in a weighted graph with non-negative edge weights

- Algorithm

1. Create a set to store the shortest distances between from the source vertex to all other

vertices, initialize the distances to infinity and set the distance of the source vertex to 0.

2. Create a priority queue (min heap) to keep track of the vertices to be processed based on their current shortest distance

3. While the priority queue is not empty, do the following

a) Extract the vertex with the minimum distance from the source priority queue

b) For each neighbor of "current", calculate the distance from the source vertex through "current" to that neighbor

- **Bellman-Ford Algorithm**

- The Bellman-Ford algorithm is another shortest path algorithm that can handle graphs with negative edge weights

- It finds the shortest path from a single source vertex to all other vertices in a weighted graph, even if the graph contains negative-weight cycles (in which the sum of the weights in a cycle in a negative)

Date :

• Algorithm

1. Create an array to store the shortest distances from the source vertex to all other vertices initialize the distances to infinity, and set the distance of the source vertex to 0.

2. Repeat the following process v-1 (v is the number of vertices in the graph):

a. For each edge (u,v) with weight w, if distance to vertex u plus w is smaller than the current distance

3. After v-1 iterations, check for negative weight cycles if the distance to any vertex can still be improved the graph, contains a negative-weight cycle and the algorithm cannot find the shortest paths

• Floyd Warshall Algorithm-

• The Floyd Warshall Algorithm is an all pair shortest path algorithm unlike Dijkstra and Bellman Ford which are single source shortest path algorithms

• This algorithm works for both directed and undirected weighted graphs.

Date :

• Algorithm :-

- Intialize the solution matrix same as the input graph matrix as a first step

- Then updates the solution matrix by considering all vertices as an intermediate vertex

- The idea is to pick all vertices one by one and updates all shortest paths which include the picked vertex as an intermediate vertex in the shortest path

- When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2 .. k-1} as intermediate vertices

- For every pair (i, j) of the source and destination vertices respectively, there are two possible cases

ⓐ k is not an intermediate vertex in shortest path from i to j, We keep the value of dist [i][j] as it is.

ⓑ k is an intermediate vertex in shortest path from i to j, we update the value of dist [i][j] as dist [i][k] + dist [k][j], if dist [i][j] > dist [i][k] + dist [k][j]

Date :

9. SEARCING AND SORTING

• Searching Concept -

• Searching in data structure refers to finding a given element in the array of 'n' elements

• There are various types of searching techniques including linear search, binary search, hash search and tree search

• Search algorithms are designed to check or retrieve an element from any data structure where that element is being stored

• Types of Search Algorithms

1. Linear search
2. Binary search

• Linear search

• This algorithm works by sequentially iterating through the whole array or list from one end until the target element is found

Date :

- If the element is found, it returns its index, else -1.

- Algorithm

```
LINEAR SEARCH (A, N, VAL)
step 1: [INITIALIZE] SET POS = -1
step 2: [INITIALIZE] SET T = 1
step 3: Repeat step 4 and 5
step 4: If A[I] = VAL
            SET POS = I
            PRINT POS
            GO TO STEP 6
            [END OF IF]
            PRINT POS
            Goto step 6
            [END OF IF]
            SET I = I + 1
            [END OF LOOP]
step 5: IF POS = -1
            PRINT "value is not present in
            array"
            [END OF IF]
step 6: EXIT
```

- Complexity of Algorithm

complexity	Base case	Average case	Worst case
Time	O(1)	O(n)	O(n)
Space	-	-	O(1)

Date :

• Binary Search

- This type of searching algorithm is used to find the position of a specific value contained in a sorted array

- The binary search algorithms works on the principle of divide and conquer and it is considered the best searching algorithm because its faster to run

• Algorithm

```
BINARY SEARCH (A, LOWER_BOUND, UPPER_Bound, VAL)
step 1: [INITIALIZE] SET BEG = Lower bound
        END = upper bound, pos = -1
step 2: Repeat step 3 and 4 with
        BEG <= END
step 3: SET MID = (BEG + END)/2
step 4: IF A[MID] = VAL
        SET POS = MID
        PRINT POS GO To step 6
        ELSE IF A[MID] > VAL
        SET END = MID - 1
        ELSE SET BEG = MID + 1
step 5: IF POS = -1
        PRINT "value is not present in
        array"
step 6 : EXIT
```

Date :

• Complexity

SNO	PERFORMANCE	COMPLEXITY
1	WORST CASE	O(log n)
2	BEST CASE	O(1)
3	AVERAGE CASE	O(log n)
4	WORST CASE Space complexity	O(1)

• Sorting Concept :-

• Sorting refers to rearrangement of a given array or list of elements according to a comparison operator on the elements

• The comparison operator is used to decide the new order of elements in the respective data structure

• Types of Sorting Techniques

1. Comparison-based - We compare the elements in a comparison-based sorting algorithm

2. Non-comparison-based - We do not compare the elements in a non-comparison-based sorting algorithm

Date :

• Sorting algorithms :-

1. Selection sort

• Selection sort is another sorting technique in which we find the minimum element in every iteration and place it in the array beginning from the first index

• Thus a selection sort get divided into a sorted and unsorted subarray

Algorithm

step-1 - Set MIN to location 0
step-2 - Search the minimum element in the list
step-3 - Swap with value at location MIN
step-4 - Increment MIN to point to next element
step-5 - Repeat until the list is sorted

2. Bubble sort

• Bubble sorting is a simple sorting technique in sorting algorithm.

• In bubble sorting algorithm, arrange the elements of the list by forming pairs of adjacent elements.

Date :

- It means repeatedly step through the list which we want to sort, compare two items at a time and swap them if they are not in the right order

Steps:

- Traverse a collection of elements
- Move from the front to end
- Bubble the largest value to the end using pair-wise comparisons and swapping
- If we have N elements... and each time we bubble an element, we place it in its correct location... then we repeat the "bubble up" process N-1 times
- After pass 1 first largest element kept at last and same other passes.

Complexity

- Best Case - O(n)
- Worst Case - o(n^2)
- Average case - O(n^2)
- space complexity - o(1)

Date :

• Algorithm

• Here DATA is an array with N elements.

• This algorithm sorts the element in DATA

1. Repeat Steps 2 and 3 for k = 1 to N-1

2. Set PTR := 1 [Initializes pass pointer PTR]

3. Repeat while PTR <= N-k : [Execute pass]

   a) If DATA [PTR] > DATA [PTR+1], then :
      Interchanges DATA [PTR] and DATA [PTR+1]
      [END of IF structure]

   b) set PTR := PTR + 1

   [End of inner loop]

   [End step 1 of outer loop]

4. Exit

Date :

• Insertion sort

• Insertion sort is a simple sorting algorithm that builds the final sorted array one item at a time

• It is much less efficient large lists than more advanced algorithm

Algorithm

(Insertion Sort) INSERTION (A, N)

• This algorithm sorts the array A with N elements-

1. Set A[0] := -∞ [Initialize Sentinel element]

2. Repeat step 3 to 5 for k = 2, 3, 4 ... N.

3. set Temp := A[k] and PTR = k-1;

4. Repeat while TEMP < A[PTR]:
   a) set A[PTR+1] := A[PTR]
   b) set PTR := PTR-1
   [End of loop]

5. Set A[PTR+0] :- TEMP

[End of step 2 loop]

6. Return

Date :

• Radix Sort

- Radix sort is a linear sorting algorithm that is used for integers

- In Radix sort, there is digit by digit sorting is performed that is started from the least significant digit to the most significant digit

• Algorithm

radixSort (arr)

max = largest element in the given array

d = number of digits in the large element (or, max)

Now, create d bucket of size 0-9

for i -> 0 to d

sort the array elements using counting sort (or any stable sort) according to the digit at the ith place

Date :

· Quick sort

· Is a highly efficient sorting technique that divides a large data array into smaller once

· A vast array is divided into two arrays, one containing values smaller than the provided value, say pivot on which the partition is based

· Algorithm

```
· QuickSORT (array (A, start, end)
{
    if (start < end)
    {
        p = partion (A, start, end)
        QUICKSORT (A, start, p-1)
        QUICKSORT (A, p+1, end)
    }
}
```