COMPLETE NOTES ON Data Structures in C++

Instagram : @ curious_.programmer

Telegram : @ curious_coder

Written by :-

Damini Patil (CSE student)

Index

Sr. No	chapters	Page No.
	tation using arrays and linked lists 4.4 Queue operations	
5.	Recursion 5.1 Introduction to recursion and its principle. 5.2 Recursive algorithms and their analysis. 5.3 Recursive solutions to common problems (e.g factorial, fibonacci series)	25-27
6.	Trees 6.1 Introduction to trees and their properties 6.2 Binary trees, binary search trees, and balanced search trees 6.3 Tree traversal algorithms (pre-order, in-order, post-order) 6.4 Binary heaps and priority queue.	28-35
7.	Graphs 7.1 Graph representations 7.2 Graph traversal algorithms 7.3 Minimum spanning tree algorithms 7.4 shortest path algorithms (Dijkstra's algorithm, Bellman-Ford algorithm)	36-47
	..— End ——..	

1. Introduction to Data Structures

1.1 Overview of Data structures and Their Significance

Data structures are fundamental tools in computer science that allow 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 diffreent pieces of data. By choosing appropriate data structures, we can optimize the efficiency 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.

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

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 significantly 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.

1.2 Basic terminologies and concepts in data structures

Before diving deeper into specific data structures, it's essential to understand some fundamental terminologies and concepts:

1. Data:
Any piece of information that can be processed or manipulated by a computer program.

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 perf-

orm operations efficiently.

4. Primitive data structures:
Basic data structures provided by programming languages, such as integers, floats, characters, etc.

5. Composite data structures:
Data structures built by combining primitive data structures, like arrays, linked lists, and trees.

6. Operations:
The actions performed on data structures, such as insertion, deletion, search, update, etc.

7. Linear data structures
Data elements are organized in a linear sequence, such as arrays, linked lists, stacks, and queues.

8. Non-linear data structures
Data elements are organized hierarchically, such as trees and graphs.

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

10. Dynamic data structure
Data structures that can grow or shrink in size during program execution.

1.3 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 grows larger. The time complexity of an algorithm is expressed using Big O notation, which 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:

In Big O notation, an algorithm's 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.

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^n) (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 also expressed using Big O notation

Example of space complexity

```
int calculatorSum(int n) {
    int* arr = new int[n]; // Allocates an array of size "n"
    int sum = 0;
    for (int i = 0; i<n; i++) {
        arr[i] = i+1;
        sum+ = arr[i];
    }
    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 drray of size "n".

2. Arrays and Strings

2.1 One-dimensional Arrays and their operations

One-dimensional arrays:

An array is a data structure that allows us to store multiple elements of the same data type in contiguous memory locations. One-dimensional arrays, also known as vectors, represent a list of elements arranged in a single row.

Declaration and Initialization:

One dimensional arrays 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};
```

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

- **Insertion**: Inserting an element into an array at specific index.
- **Deletion** : Deleting an element from the array at a specific index.
- **Search**: Searching for a given element in the array.
- **Traversal**: Visiting each element of the array.
- **Sorting**: Arranging the elements in a specific order

2.2 Multidimensional array and Matrices:

Multidimensional Arrays:

A multidimensional array is an array of arrays, forming a matrix-like structure. The most common type is a two-dimensional array, representing rows and columns

Declaration and Initialization

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

// Example of 2D integer 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 the element at row 1, column 2 (value will be 6)
```

Common Operations:

Transpose: Converting rows into columns and vice versa.

Addition and substraction: Performing arithmetic operations on two matrices.

Multiplications: Multiplying two matrices.

2.3 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 strings efficiently

Declaration and Initialization:

```
// Declaration and initialization of strings
char str1[] = "Hello"; // Automatically includes the null characters.
char str2[10]; // Declaration of an empty character array with enough with space for 9 characters (+1 null characters)

// 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 length of string is the number of characters it contains. In many programming, there are built-in functions or methods to find the length of a string.

```
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 two strings

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

5. Modification : changing or Replacing parts of a string

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

```
std :: string str = "Hello, World!";
// Replacing a substring with another string
str.replace (0,5, "Hi");
```

3. Linked Lists

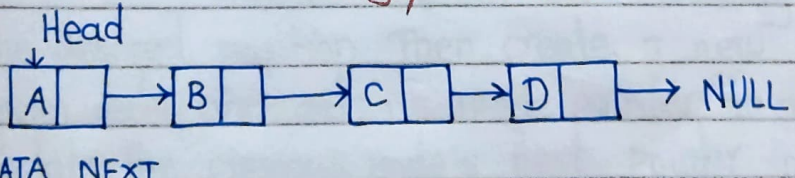
Linked Lists are linear data structures used to store and organize elements in memory. The elements in a linked list are linked using pointers. In simple worlds, a linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.

3.1 Singly linked lists 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

The node contains a pointer to the next node means that the node stores the address of the next node in the sequence. A single linked list allows the traversal of data only in one way.

Singly linked list



Structure of the singly linked list :

```
class Node {
public:
    int data;
    // Pointer to next node in LL
```

```
Node* next;
};
```

Basic Operations:

Insertion: There are three main types of insertion in singly linked list.

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.

Insert at the End:

Traverse the list untill 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 untill 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's next pointer to pointer to point to 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 untill reaching the node before the desired position. Update its next pointer to point to the node after the desired position and delete the node at the last node.

Delete at the specific position

Transeverse the list untill reaching the node before desired position. Update its next pointer to point the node after the desired position and delete the node at the desired position.

Traversal :

Start from the head of the list and visit each node sequentially, performing the necessary operations on the data.

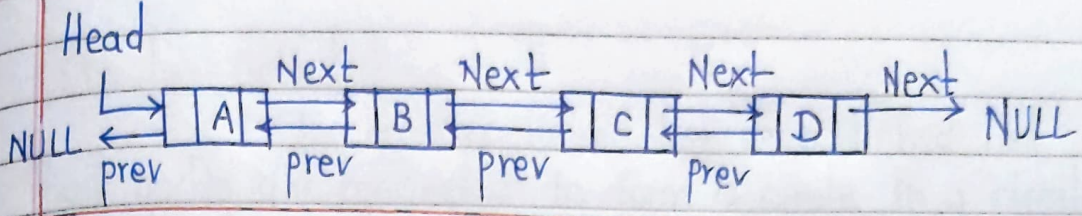
Searching:

Traverse the necessary list, comparing the data in each node with the target value untill a match is found or the end of the list is reached.

3.2 Doubly Linked Lists and Circular linked lists.

Doubly Linked Lists:

In a doubly linked list, each node has two pointers: One pointing to the next ti node (as in singly linked lists) another pointing to the previous node. This is bidirectional linking allows for efficient traversal in both directions.



Doubly Linked list

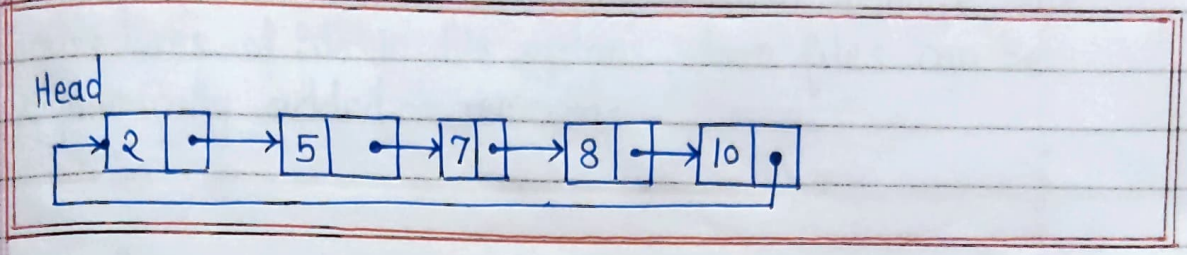
```
// Node of a doubly linked list
class Node{
public:
  int data;
   // pointer to next node in DLL
  Node* next;
   // pointer to previous node in DLL
  Node* prev;
};
```

Advantages of Doubly Linked List over the singly linked list:

- A DLL can be traversed in both forward and backward directions.
- The DLL has delete operation which is more efficient if a pointer to the node to be deleted is given.
- We can quickly insert a new node before a given node.
- In a singly linked list, to delete a node, a pointer to the previous node is needed. To get this previous node, sometimes the list is traversed. In DLL, we get the previous node using the previous pointer.

Circular linked lists:

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 forms a circle. There is no NULL at the end.



There are generally two types of circular linked lists.
1) Circular Singly linked list.
2) Circular doubly linked list.

3.3 Applications of Linked Lists.

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

1. Dynamic Memory Allocation

Linked lists 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. Graphs and Trees:

Linked lists are the building blocks for implementing more complex data structures like graphs and trees.

4. File systems:

Linked lists are the used to manage the structures of file in file systems, where files can be dynamically added or removed.

4. Stack and Queues

Stack data structure

A stack is a linear data structure that follows the Last In, First Out (LIFO) principle. This means that the last element added to the stack will be the first one to be removed. It has two main operations : push (to add an element) and pop (to remove an element).

Implementation using arrays :

```
#include <iostream>
const int MAX_SIZE = 100; // Maximum size of stack
class stack {
private:
   int arr[MAX_SIZE];
   int top; // Index of the top element
public:
   stack() {
      top = -1; // Initialize the stack as array empty
   }
bool isEmpty() {
   return (top == -1);
}
bool isFull() {
   return (top == MAX_SIZE - 1);
}
Void push (int value) {
   if (isFull()) {
      std:: cout << "stack overflow! cannot push element."
```

```
    << std :: endl;
    return;
  }
  top++;
  arr[top] = value;
 }
void pop () {
  if (isEmpty()) {
   std :: cout << "stack overflow underflow ! cannot pop
   element." << std :: endl;
   return;
  }
  top--;
 }
}
```

4.2 Stack operations (push, pop, peek)

1. Push:

Adds an element to the top of the stack. Its an operation in which the elements are inserted at the top of the stack. In the push functions, we need to pass an element which we want to insert in a stack.

2. Pop():

It is an operation in which the elements are deleted from the top of the stack. In the pop() function, we do not have to pass any argument!

3. peek():

This function returns the value of the topmost element available in the stack. Like pop(), it returns the value of the topmost element but does not remove that element from the stack.

4.3 Queue data structure and its' implementation using arrays and linked lists

A queue is a linear data structure that follows the first In, First Out (FIFO) principle. This means that the first element added to the queue will be the first one to be removed. It has two main operation: enqueue (to add an element) and dequeue (to remove an element)

Implementation using Arrays:

```
#include<iostream>
const int MAX_SIZE = 100; // Maximum size of queue
class queue {
private:
 int arr[MAX_SIZE];
 int front, rear; // Index of front and rear elements.
 int size;
public:
 queue() {
  front = rear = -1; // Initialize queue as empty
   size = 0;
 }
```

```
bool isEmpty(){
    return (size==0);
}
bool isFull(){
    return (size== MAX_SIZE);
}
void enqueue(int value){
    if (isFull()) {
    std::cout << "Queue overflow! Cannot enqueue element."
     << std::endl;
     return;
    }
if (isEmpty()){
    front = rear = 0;
} else {
    rear = (rear+1) % MAX_SIZE; // circular queue implem-
    entation
}
  arr[rear] = value;
   size++;
}
void dequeue(){
    if (isEmpty()){
    std:: cout << "queue overflow underflow! cannot
     dequeue element." << std::endl;
     return;
    }
   if (front == rear){
    front = rear = -1;
   } else {
```

```
    front = (front + 1) % MAX_SIZE;
    }
    size--;
}
int peek(){
    if (isEmpty()){
        std::cout << "queue is empty ! cannot peek. " << std::endl;
        return -1 ;
    }
    return arr[front];
}
};
```

Implementation using linked lists.

```
#include <iostream>
class Node{
public:
    int data;
    Node* next;
    Node(int value){
        data = value;
        next = nullptr;
    }
};
class Queue{
private:
    Node* front;
    Node* rear;
```

```
public:
  Queue() {
     front = rear = nullptr;
  }
  bool isEmpty () {
    return (front == nullptr);
  }
  void enqueue(int value){
   Node* newnode = new Node(value);
   if (isEmpty ()) {
      front = rear = newNode;
   } else{
      rear -> next =newnode;
      rear= newNode;
    }
  }
Void dequeue () {
   if (isEmpty()){
   std:: cout << "Queue underflow" << std:: endl;
   return;
  }
  Node* temp =front;
  front = front -> next;
  delete temp;
  if (front ==nullptr){
    rear = nullptr;
  }}
}
int peek (){
  if (isEmpty()){
```

```
        std::cout << "Queue is Empty!" << endl;
        return -1;
    }
    return front->data;
    }
};
```

Queue operations:

Enqueue: Adds an element to the rear of the queue
Dequeue: Removes the front element from the queue.

5. Recursion

5.1 Introduction to recursion and its principle

Recursion is a powerful programming technique in which function calls itself to solve a problem. It is a way of solving problems that involve breaking them down into smaller, similar subproblems. The idea is to solve the base case directly and then reduce the problem to a smaller version of itself and call the function recursively untill it reaches the base cases.

For recursion to work correctly, there are two fundamental principles 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. It acts as a termination condition for the recursive calls and prevents infinite 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. This step breaks the original problem into smaller more managable subproblems.

5.2 Recursive Algorithms and Their analysis

Recursive algorithms can be an elegant and

consise way to solve certain problems. However, they come with the cost of additional function calls and memory usage, which can lead to stack overflow or excessive time complexity if not implemented carefully.

To analyze the time complexity of a recursive algorithm, we can use the recurrence relation, which expressess the time complexity in terms of smaller subproblems. The recurrence relation helps us understand how many times the function will be called and how much work each call performs.

5.3 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)!
 }
}
```

2. Fibonacci series :

The Fibonacci series is a sequense of numbers in which each number (Fibonacci number) is the sum of the two preceeding ones, usually starting with 0 and 1.

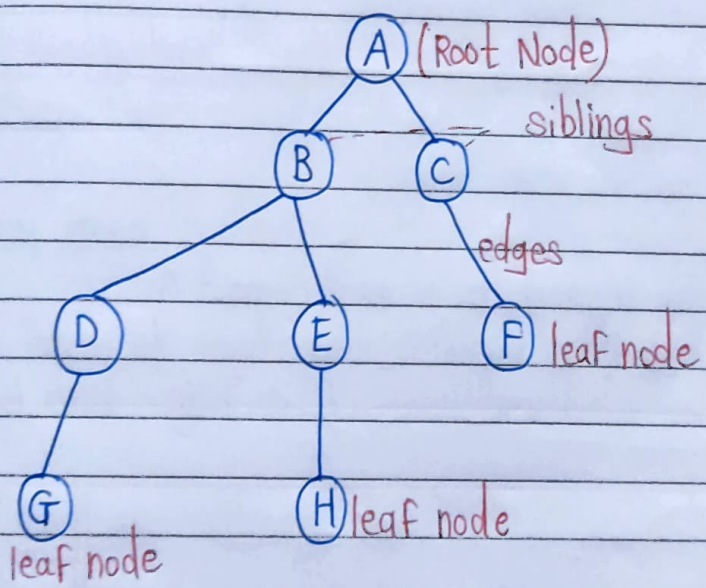
Recursive algorithm:

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

6. Trees

6.1 Introduction to Trees and their properties

A tree is a hierarchical data structure that consists of nodes connected by edges. It is a non-linear data structure, meaning it does not have a linear sequence like arrays or linked lists. In tree, there is a single node called root from which all other nodes are reachable via directed edges.



Tree terminology:

- Node
- parent
- Leaf
- Depth
- siblings
- Root
- child
- Edge
- Height

1. Node: Each element of a tree is called 'Node'. It contains

some data and may have links to other nodes

2. Root : The topmost node in a tree from which all other nodes are descendants.

3. Parent : A node that has child node connected to it.

4. child : Nodes connected to a parent node

5. leaf : Nodes with node no children

6. Edge : The link or connection between two nodes

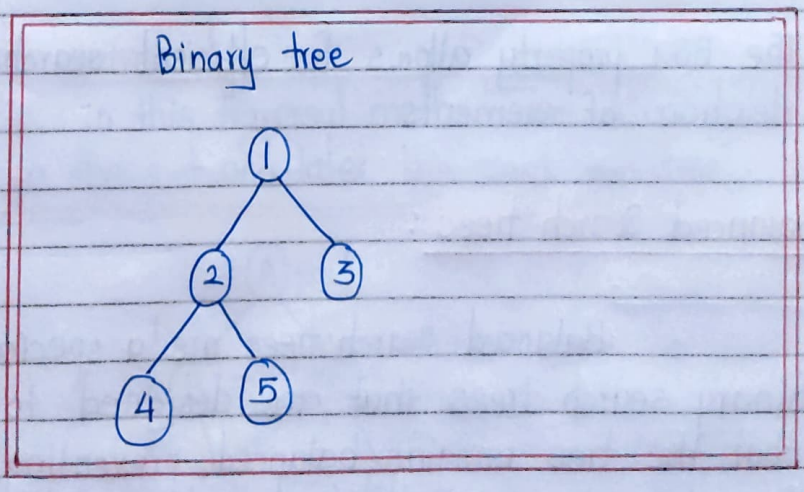
7. Depth : The length of the path from the last root to a particular node.

8. Height : The length of the longest path from a node to a leaf in its subtree.

6.2 Binary trees, binary search trees and balanced search trees.

Binary trees :

A binary tree is a tree in which each node can can have at most two children, referred to as the left child and right child



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.

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.

The BST property allows for efficient searching, insertion deletion of elements

Balanced Search trees:

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

tree to maintain efficient operations.

There are three types of Balanced search trees:
1) AVL Trees
2) Red-Black Trees
3) splay Trees

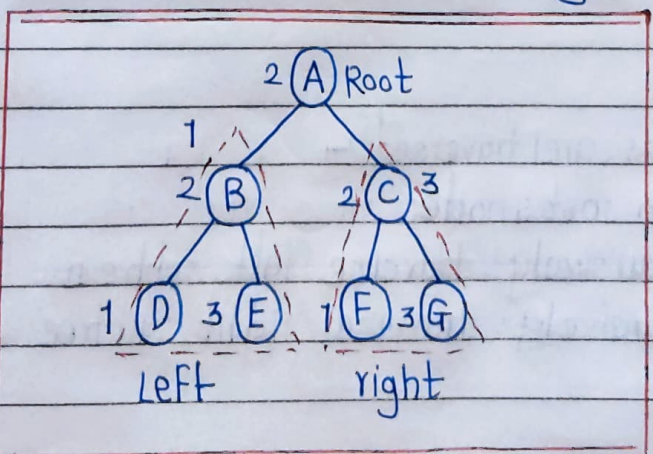
6.3 Tree traversal algorithms (pre-order, in-order, post-order)

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

- In-order Traversal
- Pre-order Traversal
- 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.

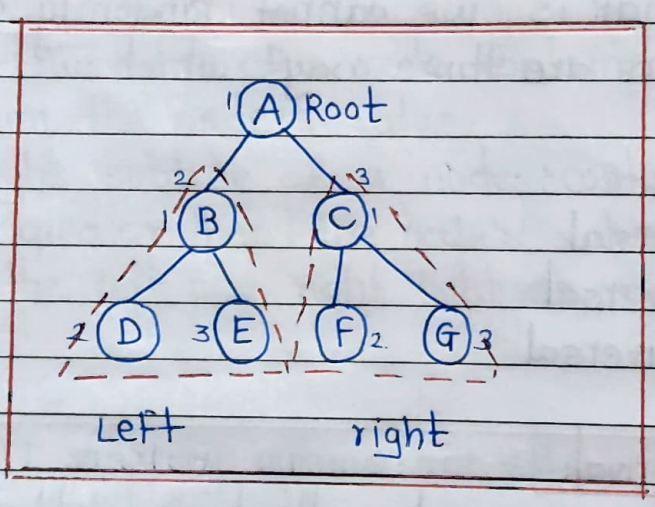


In-order → D-B-E-A-F-C-G

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

Pre-order Traversal:

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

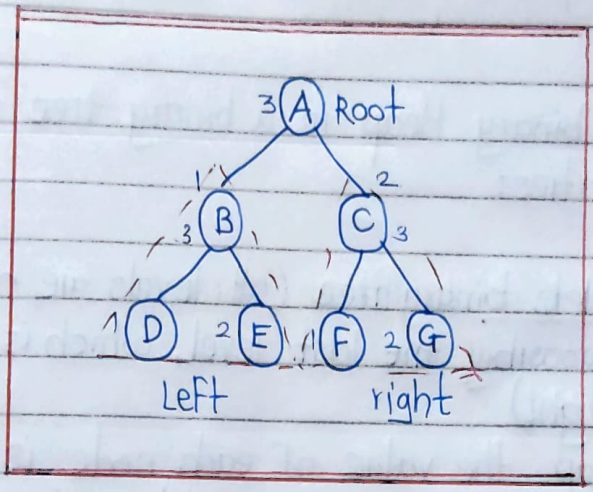


pre-order → A-B-D-E-C-F-G

Algorithm:
Untill all 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.

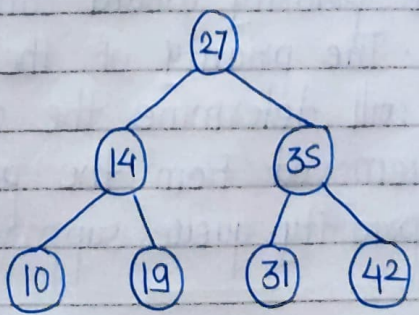


Post-order → D-E-B-F-G-C-A

Algorithm:

Untill all nodes are traversed -
step1 - Recursively traverse left subtree
step2 - Recursively traverse right subtree
step3 - 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

6.4 Binary Heaps and Priority Queues

Binary Heap:

A binary Heap is a binary tree with the following properties:

1. It is a complete binary tree (all levels are completely filled except possibly the last level, which is filled from left to right)
2. In a min-heap, the value of each node is less than or equal to the values of its children (root has the minimum value)
3. In a max-heap, the value of each node is greater than or equal to the values of its children (root has the maximum value)

Priority Queue:

A priority queue is an abstract data type that behaves similarly to an normal queue except that each element has some priority i.e the element with the highest priority would come first in a priority queue. The priority of the elements in a priority queue will determine the order in which elements are removed from the priority queue.

The priority queue supports only comparable

elements, which means that the elements are either arranged in an ascending or descending order

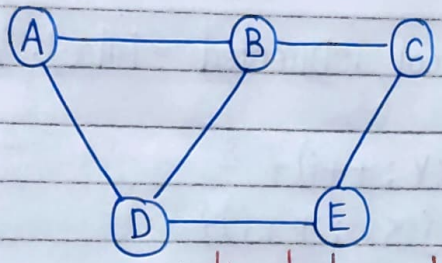
7. Graphs

Graph:

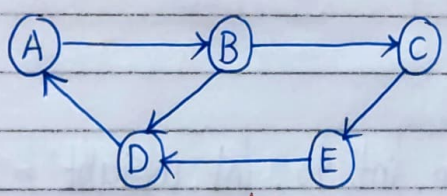
A graph G can be defined as an ordered set G(V,E) where,

V(G) represents the set of vertices
E(G) represents the set of edges

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



•Undirected graph•



•Directed graph•

7.1 Graph representation

Adjancency Matrix:

An adjancency matrix is a 2D array of size V×V (where V is the number of vertices) that represents a graph. If there is an edge between vertices i and j.

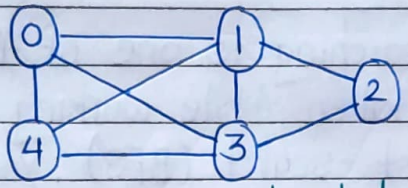
, the matrix cell at (i,j) is marked with a 1 or a weight (for weighted graphs). Otherwise, it contains a 0 or a special value to represent no edge.

```
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;
    }
  }
}
  voidaddEdge(int from, int to, int weight = 1){
   matrix [from][to] = weight;
    if (!directed) {
      matrix[to] [from] = weight;
  }
  }
}
```

Adjancency List :

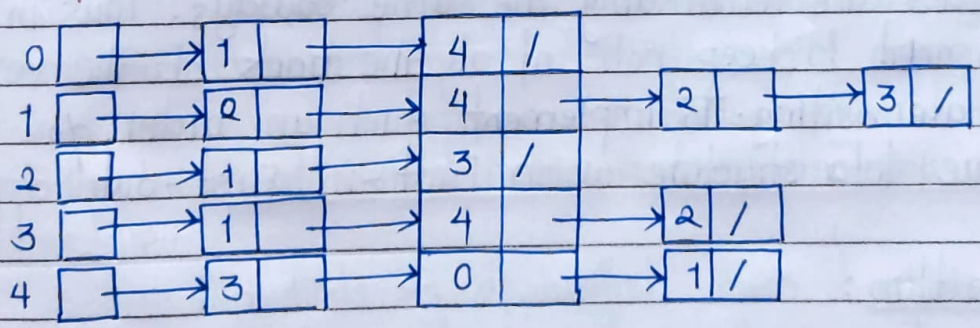
An array of linked list is used. The size of the array is equal to the number of vertices. Let the array be an array[]. An entry array[i] represents the linked list of vertices adjacent to the i^{th} vertex.
This representation can also be used to represent a weighted graph. The weights of edges can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

Consider the following graph:



• Examples of undirected graph with 5 vertices.

Following is the adjancency list representation of the above graph.



Advantages of Adjancency 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.

7.2 Graph traversal Algorithms

There are two graph traversal algorithms given below:
1) BFS - Breadth First search
2) DFS - Depth First search

Breadth First search:

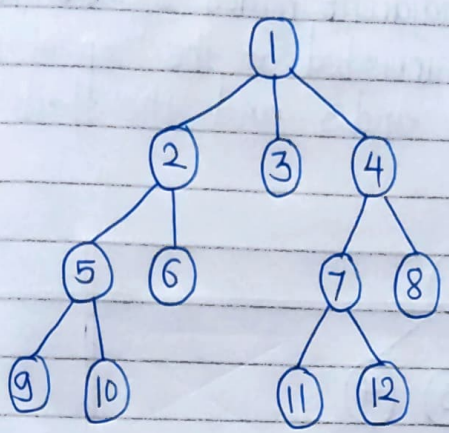
Traversing or searching is one of the most used operations that are undertaken while working on graphs. Therefore, in breadth-first-search (BFS), you start at a particular vertex, and the algorithm tries to visit all the neighbours at the given depth before moving on to the next level of traversal of vertices. 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 note 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 the queue
2. First, move the front queue item and add it to 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 untill the queue is found to be empty.

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



∘ Breadth First Search ∘

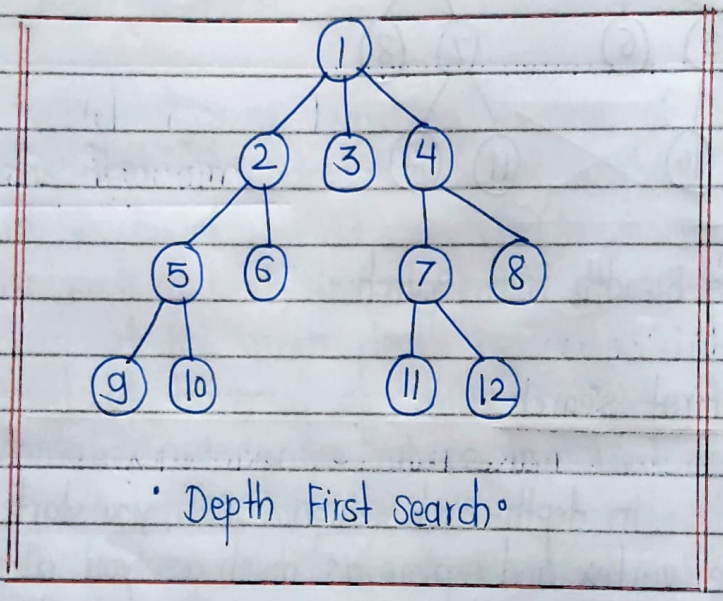
Depth-First search:

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.

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 those nodes to unvisited at the top of the stack.
4. keep representing steps 2 and 3 and the stack becomes empty.



• Depth First Search •

7.3 Minimum spanning tree algorithm

There are two different algorithms

1) Prim's algorithm
2) Kruskal's algorithm.

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, startVertex));

while (!pq.empty()) {
    int currentVertex = pq.top(). second;
    int weight = pq.top() . first;
    pq.pop();

if (! visited [currentVertex]) {
    visited [currentVertex] = true;
    minCost += weight;

for (const auto& neighbor : graph.adjList [currentVertex]) {
```

```
        int neighboursVertex = neighbor.first;
        int neighborVertex = neighbor.second;

        if (!visited[neighborVertex]) {
          pq.push(std::make_pair(neighborWeight, neighbor-
          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.

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 cycle, including it in the MST
4. Continue untill there are V-1 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;
  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;
}
```

7.4 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)

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.