UNIT V

TREES AND GRAPHS

In linear data structure data is organized in sequential order and in non-linear data structure data is organized in random order. A tree is a very popular non-linear data structure used in a wide range of applications. A tree data structure can be defined as follows...

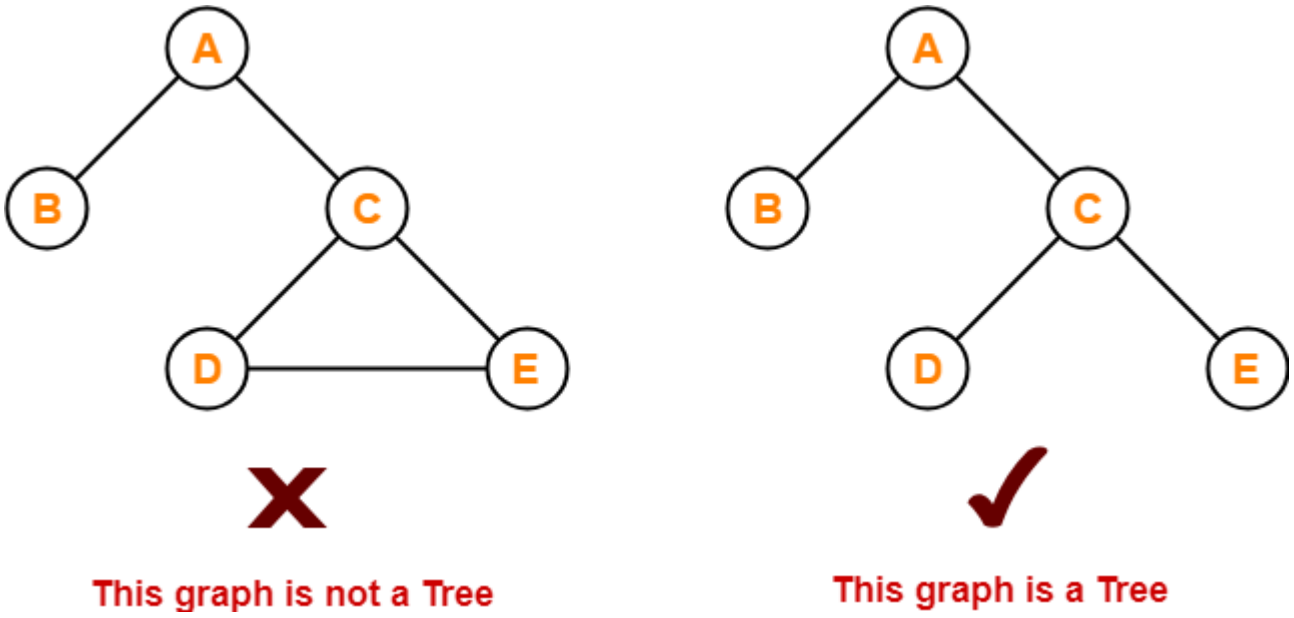
Tree is a non-linear data structure which organizes data in a hierarchical structure and this is a recursive definition.

OR

A tree is a connected graph without any circuits.

OR

If in a graph, there is one and only one path between every pair of vertices, then graph is called as a tree.



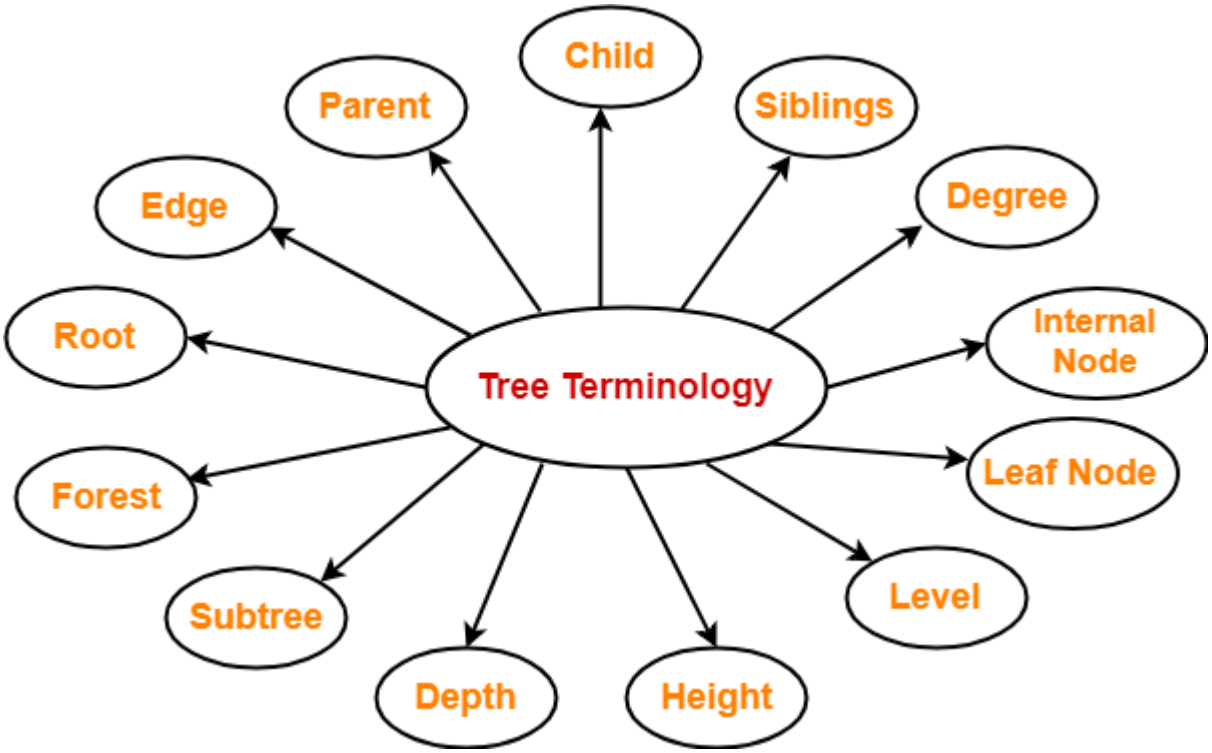
Properties-

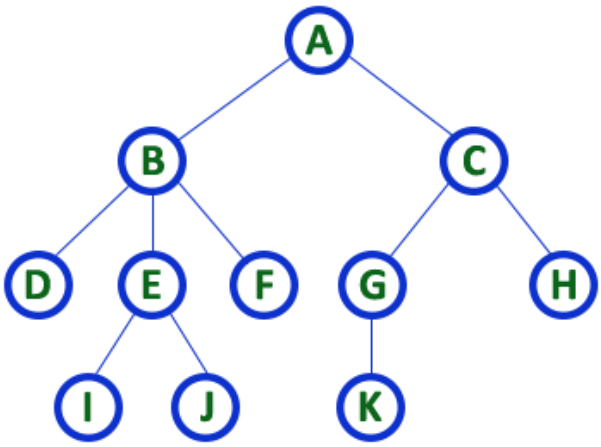
The important properties of tree data structure are-

- There is one and only one path between every pair of vertices in a tree.
- A tree with n vertices has exactly (n-1) edges.
- A graph is a tree if and only if it is minimally connected.
- Any connected graph with n vertices and (n-1) edges is a tree.

Tree Terminology-

The important terms related to tree data structure are-





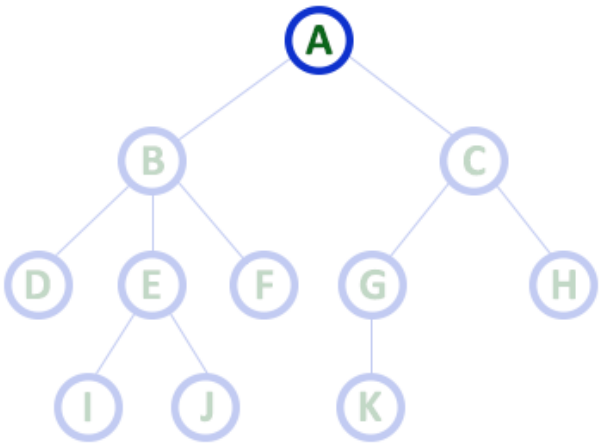
TREE with 11 nodes and 10 edges

- In any tree with 'N' nodes there will be maximum of 'N-1' edges
- In a tree every individual element is called as 'NODE'

1. Root

In a tree data structure, the first node is called as **Root Node.** Every tree must have a root node. We can say that the root node is the origin of the tree data structure. In any tree, there

must be only one root node. We never have multiple root nodes in a tree.

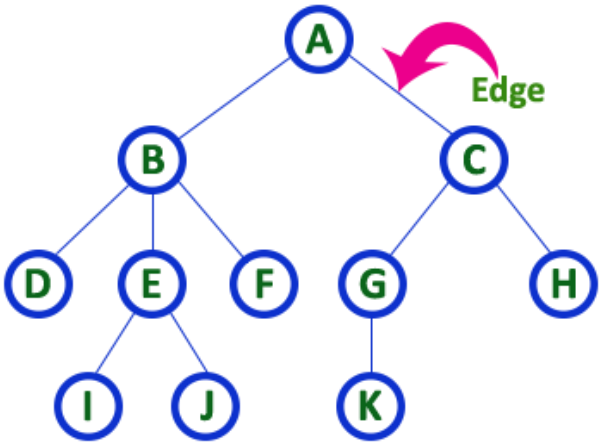


Here 'A' is the 'root' node

- In any tree the first node is called as ROOT node

2. Edge

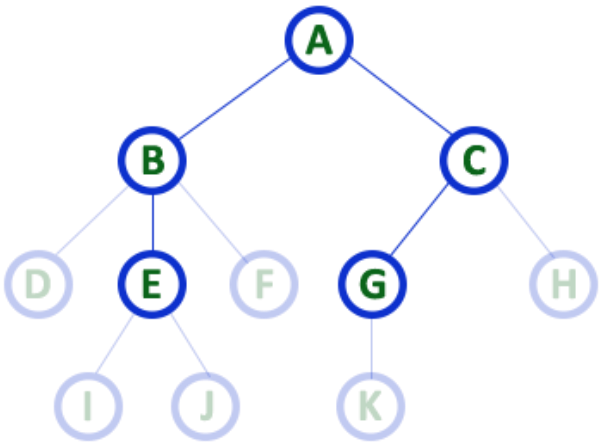
In a tree data structure, the connecting link between any two nodes is called as **EDGE.** In a tree with **'N'** number of nodes there will be a maximum of **'N-1'** number of edges.



- In any tree, 'Edge' is a connecting link between two nodes.

3. Parent

In a tree data structure, the node which is a predecessor of any node is called as **PARENT NODE.** In simple words, the node which has a branch from it to any other node is called a parent node. Parent node can also be defined as **"The node which has child / children".**

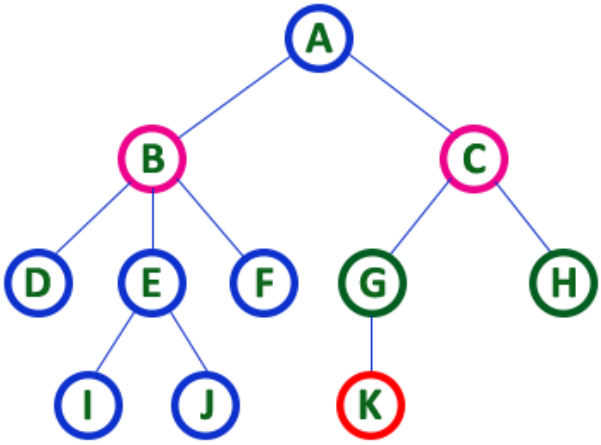


Here A, B, C, E & G are Parent nodes

- In any tree the node which has child / children is called 'Parent'
- A node which is predecessor of any other node is called 'Parent'

4. Child

In a tree data structure, the node which is descendant of any node is called as **CHILD Node**. In simple words, the node which has a link from its parent node is called as child node. In a tree, any parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes.



Here B & C are Children of A

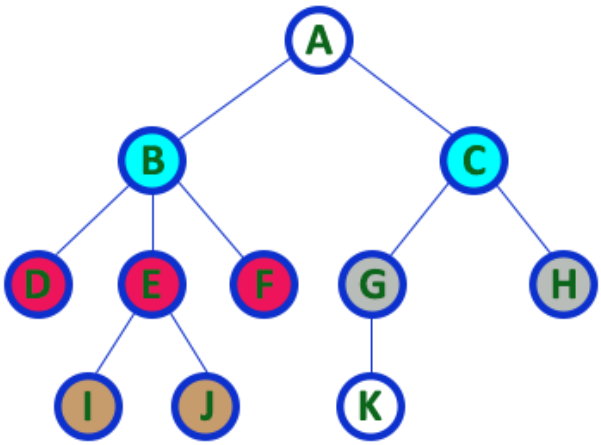
Here G & H are Children of C

Here K is Child of G

- descendant of any node is called as CHILD Node

5. Siblings

In a tree data structure, nodes which belong to same Parent are called as **SIBLINGS**. In simple words, the nodes with the same parent are called Sibling nodes.



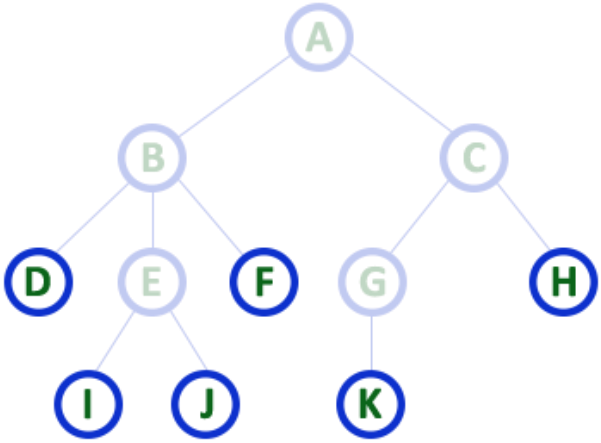
Here B & C are Siblings
Here D E & F are Siblings
Here G & H are Siblings
Here I & J are Siblings

- In any tree the nodes which has same Parent are called 'Siblings'
- The children of a Parent are called 'Siblings'

6. Leaf

In a tree data structure, the node which does not have a child is called as **LEAF Node**. In simple words, a leaf is a node with no child.

In a tree data structure, the leaf nodes are also called as **External Nodes**. External node is also a node with no child. In a tree, leaf node is also called as **'Terminal'** node.



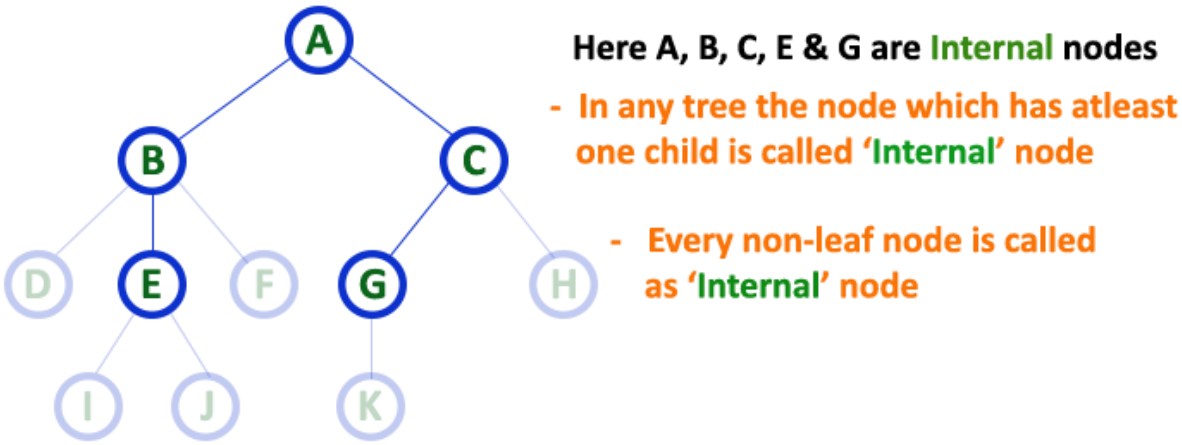
Here D, I, J, F, K & H are Leaf nodes

- In any tree the node which does not have children is called 'Leaf'
- A node without successors is called a 'leaf' node

7. Internal Nodes

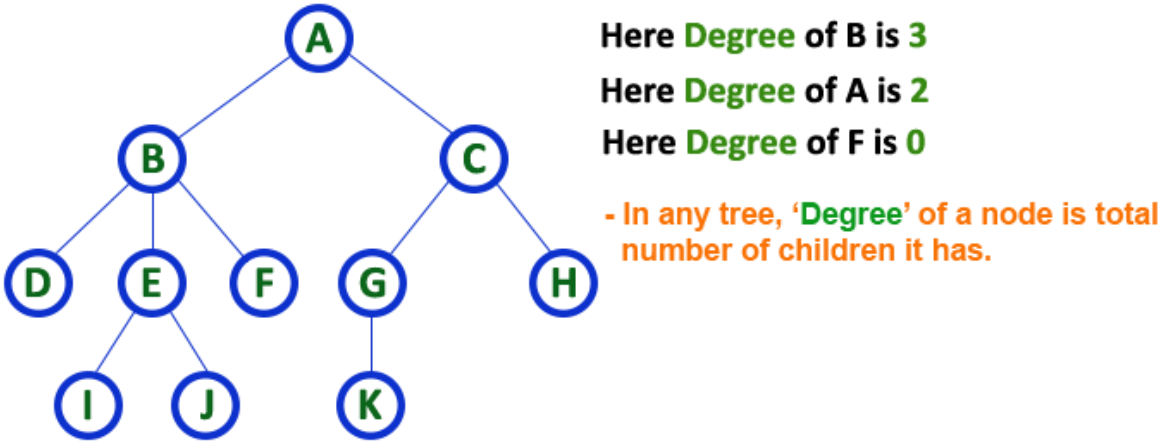
In a tree data structure, the node which has atleast one child is called as **INTERNAL Node**. In simple words, an internal node is a node with atleast one child.

In a tree data structure, nodes other than leaf nodes are called as **Internal Nodes**. **The root node is also said to be Internal Node** if the tree has more than one node. Internal nodes are also called as **'Non-Terminal'** nodes.



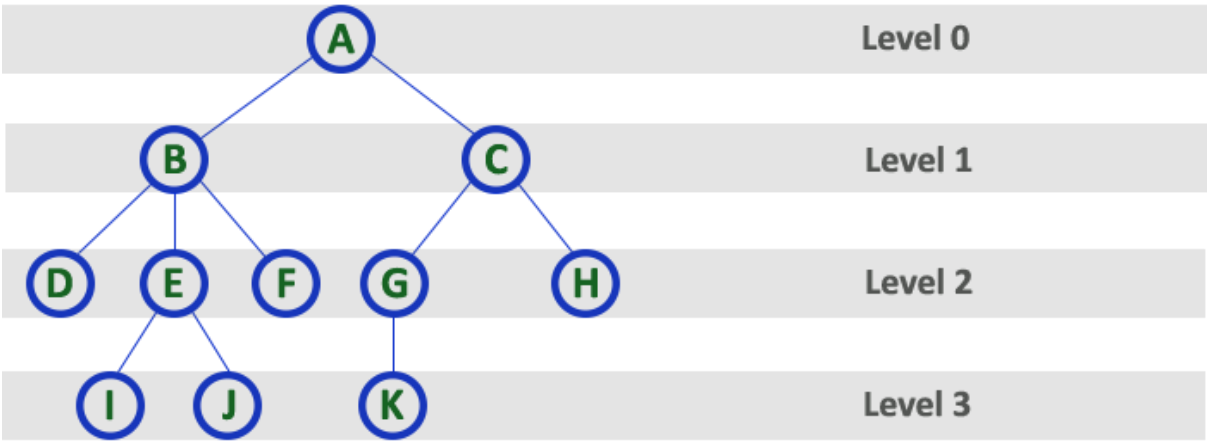
8. Degree

In a tree data structure, the total number of children of a node is called as **DEGREE** of that Node. In simple words, the Degree of a node is total number of children it has. The highest degree of a node among all the nodes in a tree is called as **'Degree of Tree'**



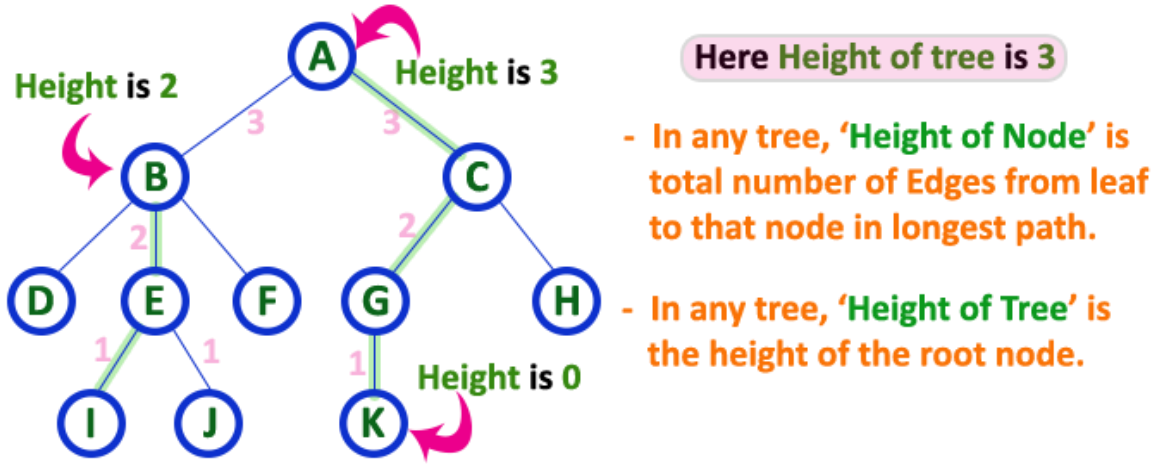
9. Level

In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on... In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step).



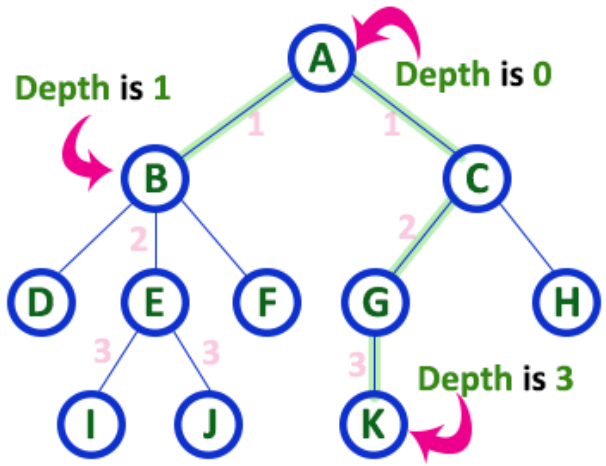
10. Height

In a tree data structure, the total number of edges from leaf node to a particular node in the longest path is called as **HEIGHT** of that Node. In a tree, height of the root node is said to be **height of the tree**. In a tree, **height of all leaf nodes is '0'.**



11. Depth

In a tree data structure, the total number of egdes from root node to a particular node is called as **DEPTH** of that Node. In a tree, the total number of edges from root node to a leaf node in the longest path is said to be **Depth of the tree**. In simple words, the highest depth of any leaf node in a tree is said to be depth of that tree. In a tree, **depth of the root node is '0'.**

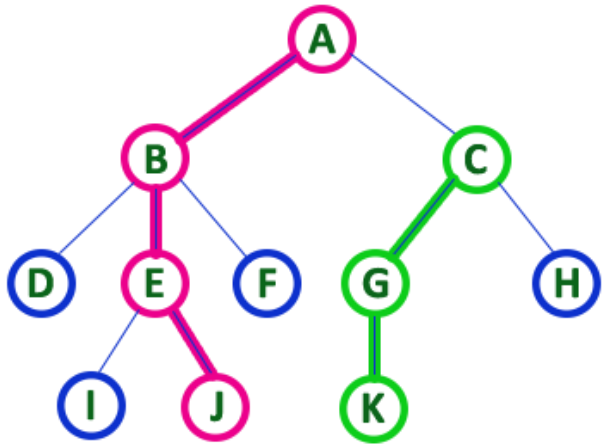


Here Depth of tree is 3

- In any tree, 'Depth of Node' is total number of Edges from root to that node.
- In any tree, 'Depth of Tree' is total number of edges from root to leaf in the longest path.

12. Path

In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as **PATH** between that two Nodes. **Length of a Path** is total number of nodes in that path. In below example **the path A - B - E - J has length 4.**



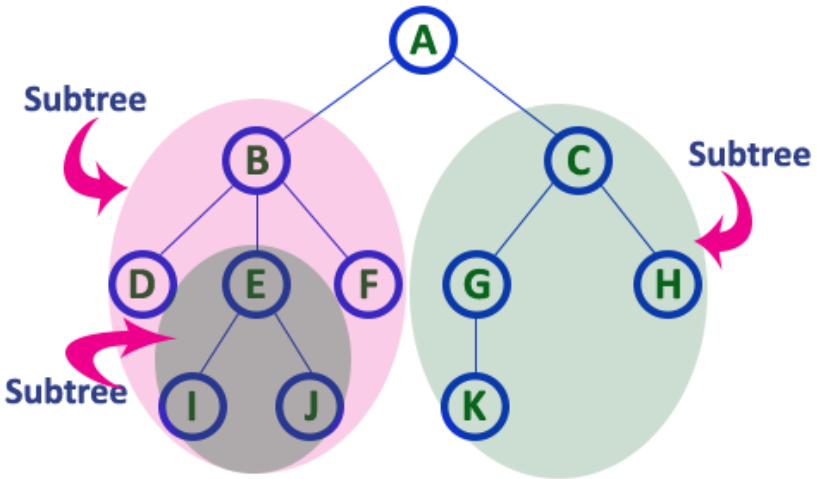
- In any tree, 'Path' is a sequence of nodes and edges between two nodes.

Here, 'Path' between A & J is
A - B - E - J

Here, 'Path' between C & K is
C - G - K

13. Sub Tree

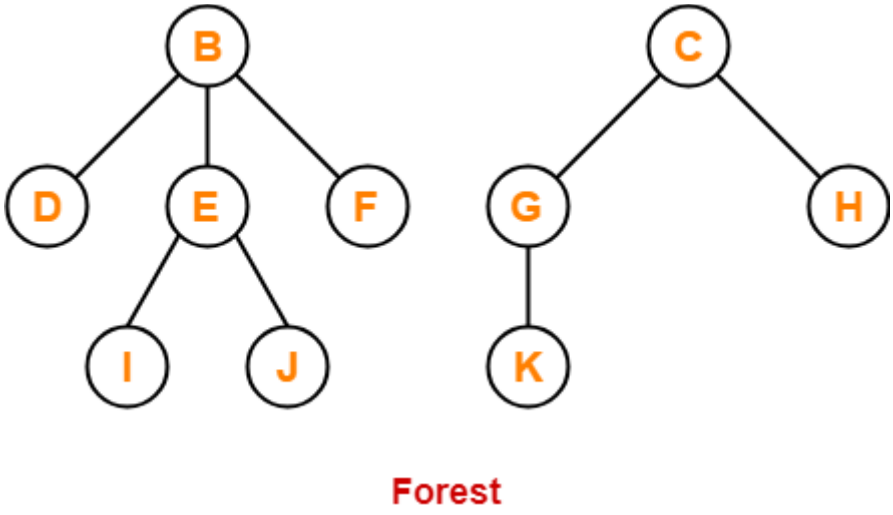
In a tree data structure, each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node.



14. Forest-

A forest is a set of disjoint trees.

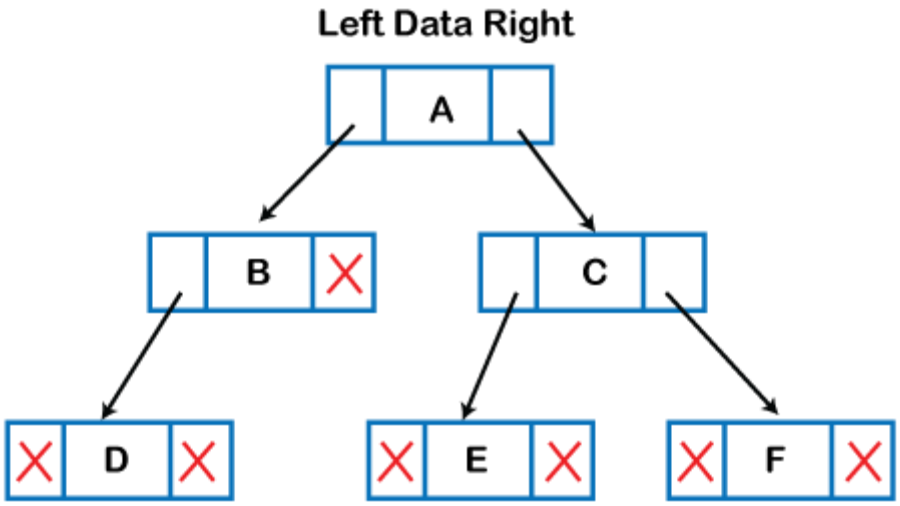
Example-



Here, nodes D, I, J, F, K and H are leaf nodes.

Implementation of Tree

The tree data structure can be created by creating the nodes dynamically with the help of the pointers. The tree in the memory can be represented as shown below:



The above figure shows the representation of the tree data structure in the memory. In the above structure, the node contains three fields. The second field stores the data; the first field stores the address of the left child, and the third field stores the address of the right child.

In programming, the structure of a node can be defined as:

```
struct node
{
  int data;
struct node *left;
struct node *right;
}
```

The above structure can only be defined for the binary trees because the binary tree can have utmost two children, and generic trees can have more than two children. The structure of the node for generic trees would be different as compared to the binary tree.

Basic Operation Of Tree:

Create – create a tree in data structure.
Insert – Inserts data in a tree.
Search – Searches specific data in a tree to check it is present or not.
Preorder Traversal – perform Traveling a tree in a pre-order manner in data structure .
In order Traversal – perform Traveling a tree in an in-order manner.

Post order Traversal -perform Traveling a tree in a post-order manner.

```cpp
#include <bits/stdc++.h>
using namespace std;

void addEdge(int x, int y, vector<vector<int> >& adj)
{
    adj[x].push_back(y);
    adj[y].push_back(x);
}

void printParents(int node, vector<vector<int> >& adj,
                  int parent)
{

    if (parent == 0)
        cout << node << "->Root" << endl;
    else
        cout << node << "->" << parent << endl;

    for (auto cur : adj[node])
        if (cur != parent)
            printParents(cur, adj, node);
}

void printChildren(int Root, vector<vector<int> >& adj)
{

    queue<int> q;

    q.push(Root);

    int vis[adj.size()] = { 0 };

    while (!q.empty()) {
        int node = q.front();
        q.pop();
        vis[node] = 1;
        cout << node << "-> ";
        for (auto cur : adj[node])
            if (vis[cur] == 0) {
                cout << cur << " ";
                q.push(cur);
            }
        cout << endl;
    }
}
```

```
void printLeafNodes(int Root, vector<vector<int> >&
adj)
{

    for (int i = 1; i < adj.size(); i++)
        if (adj[i].size() == 1 && i != Root)
            cout << i << " ";
    cout << endl;
}

void printDegrees(int Root, vector<vector<int> >& adj)
{
    for (int i = 1; i < adj.size(); i++) {
        cout << i << ": ";

        if (i == Root)
            cout << adj[i].size() << endl;
        else
            cout << adj[i].size() - 1 << endl;
    }
}

int main()
{

    int N = 7, Root = 1;

    vector<vector<int> > adj(N + 1, vector<int>());

    addEdge(1, 2, adj);
    addEdge(1, 3, adj);
    addEdge(1, 4, adj);
    addEdge(2, 5, adj);
    addEdge(2, 6, adj);
    addEdge(4, 7, adj);

    cout << "The parents of each node are:" << endl;
    printParents(Root, adj, 0);


    cout << "The children of each node are:" << endl;
    printChildren(Root, adj);


    cout << "The leaf nodes of the tree are:" << endl;
    printLeafNodes(Root, adj);
```

```
    cout << "The degrees of each node are:" << endl;
    printDegrees(Root, adj);

    return 0;
}
```

Output

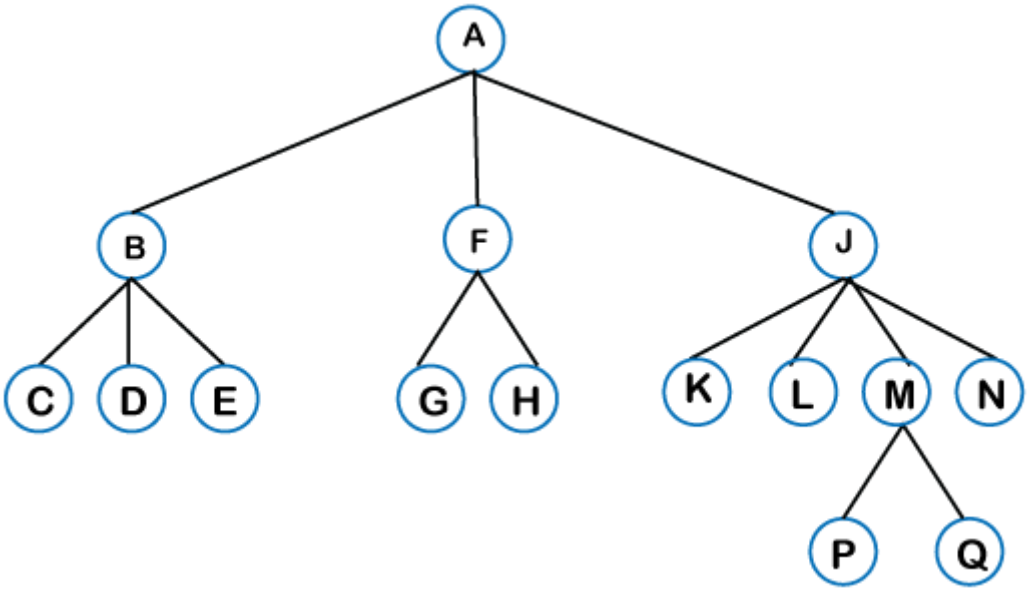
```
The parents of each node are:
1->Root
2->1
5->2
6->2
3->1
4->1
7->4
The children of each node are:
1-> 2 3 4
2-> 5 6
3->
4-> 7
5->
6->
7->
The leaf nodes of the tree are:
3 5 6 7
The degrees of each node are:
1: 3
2: 2
3: 0
4: 1
5: 0
6: 0
7: 0
```

Types of Tree data structures

The different types of tree data structures are as follows:

1. **General tree**

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

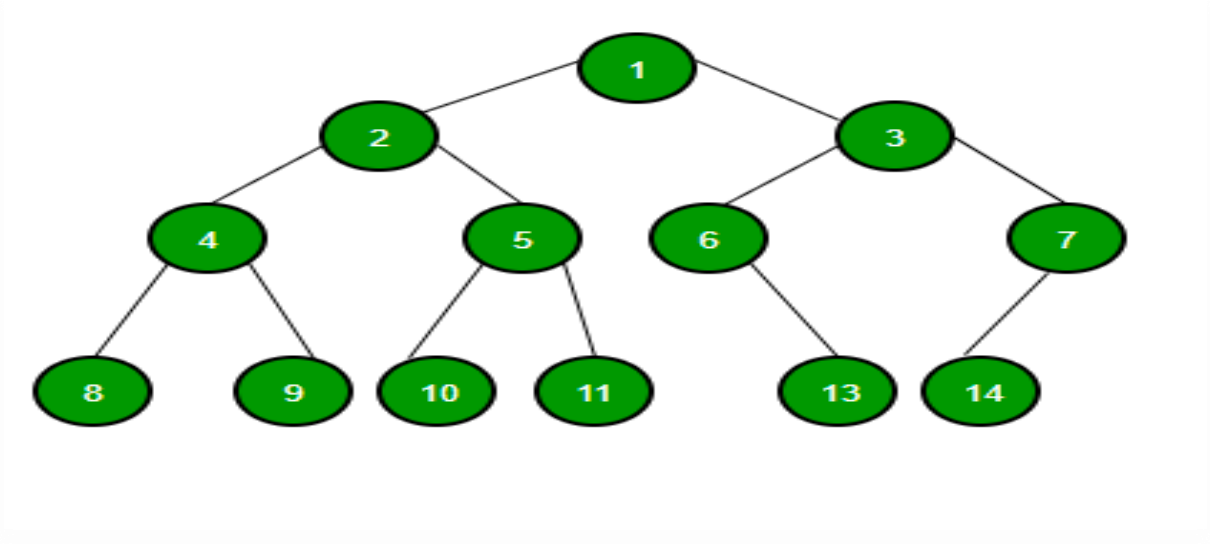


There can be ***n*** number of subtrees in a general tree. In the general tree, the subtrees are unordered as the nodes in the subtree cannot be ordered.

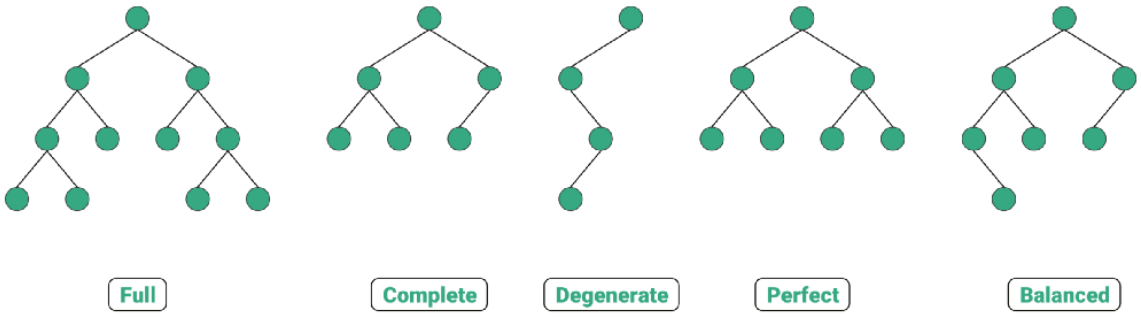
Every non-empty tree has a downward edge, and these edges are connected to the nodes known as ***child nodes***. The root node is labeled with level 0. The nodes that have the same parent are known as ***siblings***.

2. **Binary tree**

Binary Tree is defined as a Tree data structure with at most 2 children. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.



Different Types of Binary Tree



Binary Tree Representation

A Binary tree is represented by a pointer to the topmost node of the tree. If the tree is empty, then the value of the root is NULL.

Binary Tree node contains the following parts:

- ✓ Data
- ✓ Pointer to left child
- ✓ Pointer to right child

Basic Operation On Binary Tree:

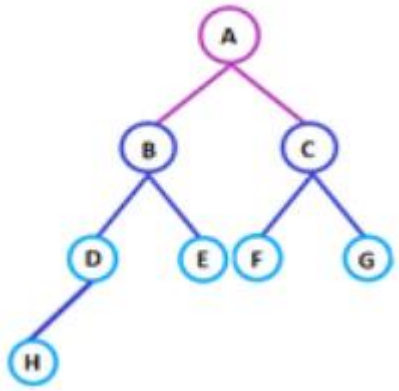
- ✓ Inserting an element.
- ✓ Removing an element.
- ✓ Searching for an element.
- ✓ Traversing an element.

Auxiliary Operation On Binary Tree:

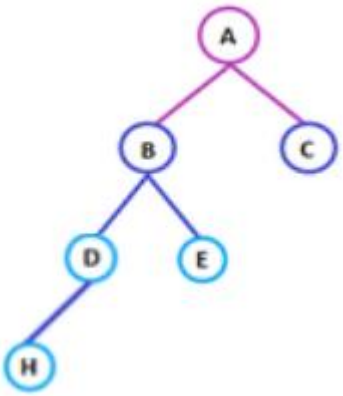
- ✓ Finding the height of the tree
- ✓ Find the level of the tree
- ✓ Finding the size of the entire tree.

3. **Balanced tree**

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



Balanced Tree



Unbalanced Tree

4. **Binary search tree**

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

Applications of Tree data structure: The applications of tree data structures are as follows:

1. Spanning trees: It is the shortest path tree used in the routers to direct the packets to the destination.

2. Binary Search Tree: It is a type of tree data structure that helps in maintaining a sorted stream of data.

- ✓ Full Binary tree
- ✓ Complete Binary tree
- ✓ Skewed Binary tree
- ✓ Stickily Binary tree
- ✓ Extended Binary tree

3. Storing hierarchical data: Tree data structures are used to store the hierarchical data, which means data is arranged in the form of order.
4. Syntax tree: The syntax tree represents the structure of the program's source code, which is used in compilers.
5. Trie: It is a fast and efficient way for dynamic spell checking. It is also used for locating specific keys from within a set.
6. Heap: It is also a tree data structure that can be represented in a form of an array. It is used to implement priority queues.

Tree Traversal in Data Structure

Tree traversal means visiting each node of the tree. The tree is a non-linear data structure, and therefore its traversal is different from other linear data structures. There is only one way to visit each node/element in linear data structures, i.e. starting from the first value and traversing in a linear order. However, in tree data structures, there are multiple ways to traverse it.

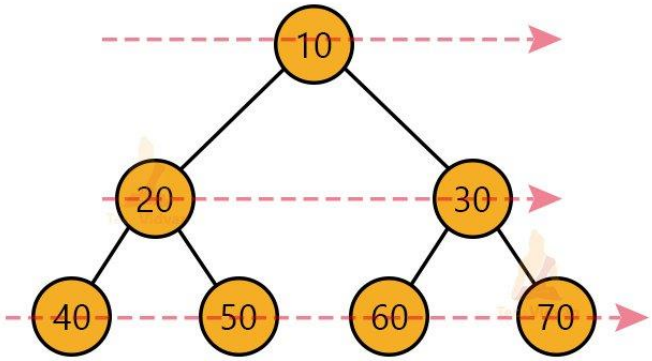
Types of Tree Traversal:

Broadly, tree traversal is classified into two categories: Level order or breadth-first traversal and depth-first traversal. The depth-first traversal further has 3 different categories that we are going to study in detail.

Level Order Traversal:

In a level order traversal, the nodes are covered in a level-wise manner from left to right. We can implement a level order traversal with the help of a queue data structure.

For example, the following diagram depicts the order of traversing the nodes of the tree.



Here, the level order traversal will print the nodes in the following order: 10, 20, 30, 40, 50, 60, 70.

Depth First Traversal:

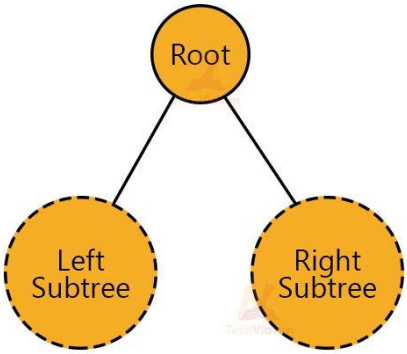
In depth-first traversal, we go in one direction up to the bottom first and then come back and go to the other direction. There are three types of depth-first traversals. These are:

1. Preorder traversal

2. Inorder traversal

3. Postorder traversal

A binary tree is a recursive data structure. Also, each binary tree has 3 parts: a root node, a left subtree and a right subtree. These left and right subtrees are also trees in themselves and thus, again further described recursively.



A tree comprises multiple nodes. The structure of one node is:

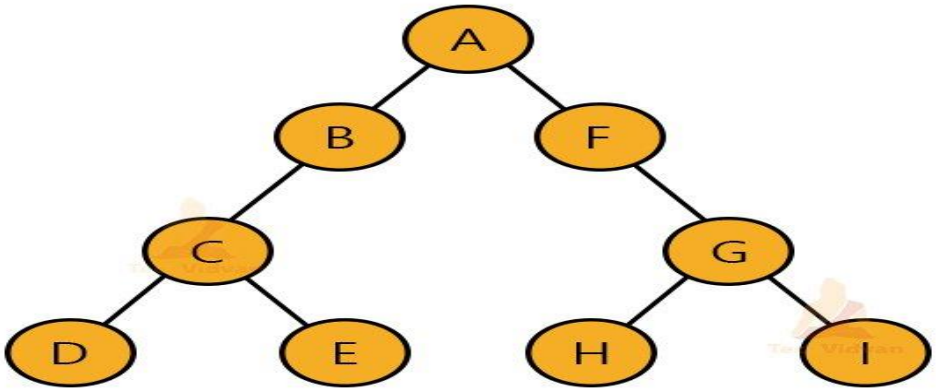
```
struct node
     {
       int data;
     struct node *left;
     struct node *right;
     }
```

Thus, each node has 3 parts: A data item or key, a pointer to the left child and a pointer to the right child.

Preorder Traversal:

In a preorder traversal, we process/visit the root node first. Then we traverse the left subtree in a preorder manner. Finally, we visit the right subtree again in a preorder manner.

For example, consider the following tree:



Here, the root node is A. All the nodes on the left of A are a part of the left subtree whereas all the nodes on the right of A are a part of the right subtree. Thus, according to preorder traversal, we will first visit the root node, so A will print first and then move to the left subtree.

B is the root node for the left subtree. So B will print next and we will visit the left and right nodes of B. In this manner, we will traverse the whole left subtree and then move to the right subtree. Thus, the order of visiting the nodes will be: A→B→C→D→E→F→G→H→I.

Algorithm for Preorder Traversal:

for all nodes of the tree:
Step 1: Visit the root node.
Step 2: Traverse left subtree recursively.
Step 3: Traverse right subtree recursively.

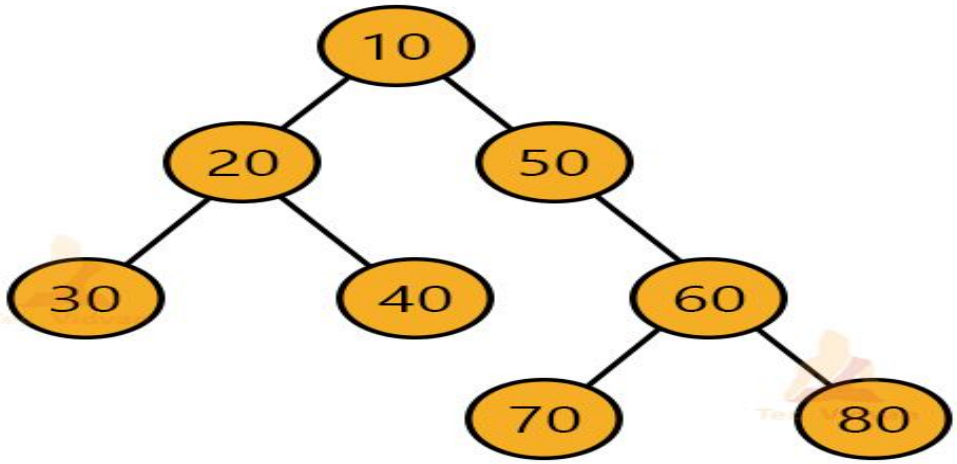
Uses of Preorder Traversal:

- ✓ If we want to create a copy of a tree, we make use of preorder traversal.
- ✓ Preorder traversal helps to give a prefix expression for the expression tree.

Inorder Traversal:

In an inorder traversal, we first visit the left subtree, then the root node and then the right subtree in an inorder manner.

Consider the following tree:



In this case, as we visit the left subtree first, we get the node with the value 30 first, then 20 and then 40. After that, we will visit the root node and print it. Then comes the turn of the right subtree. We will traverse the right subtree in a similar manner. Thus, after performing the inorder traversal, the order of nodes will be 30→20→40→10→50→70→60→80.

Algorithm for Inorder Traversal:

for all nodes of the tree:
Step 1: Traverse left subtree recursively.
Step 2: Visit the root node.
Step 3: Traverse right subtree recursively.

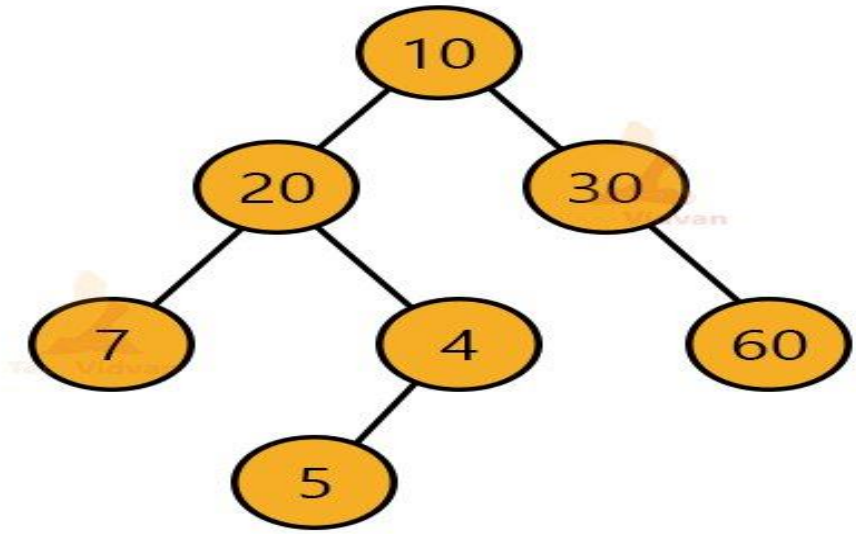
Uses of Inorder Traversal:

- ✓ Used to print the nodes of a binary search tree in non-decreasing order.
- ✓ We can also use the inorder traversal to get the non-increasing order of a BST.

Postorder Traversal:

Postorder traversal is a kind of traversal in which we first traverse the left subtree in a postorder manner, then traverse the right subtree in a postorder manner and at the end visit the root node.

For example, in the following tree:



The postorder traversal will be 7→5→4→20→60→30→10.

Algorithm for Postorder Traversal:

for all nodes of the tree:

Step 1: Traverse left subtree recursively.
Step 2: Traverse right subtree recursively.
Step 3: Visit the root node.

Uses of Postorder Traversal:

- ✓ It helps to delete the tree.
- ✓ It helps to get the postfix expression in an expression tree.

Traversal Implementation in C++:

```
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int key;
    struct Node* left;
    struct Node* right;
};

void inorder_traversal(struct Node* root) {
    if (root == NULL)
        return;
    inorder_traversal(root->left);
```

```
    cout << "->" << root->key;
    inorder_traversal(root->right);
}

void preorder_traversal(struct Node* root) {
    if (root == NULL)
        return;
    cout << "->" << root->key;
    preorder_traversal(root->left);
    preorder_traversal(root->right);
}

void postorder_traversal(struct Node* root) {
    if (root == NULL)
        return;
    postorder_traversal(root->left);
    postorder_traversal(root->right);
    cout << "->" << root->key;
}

struct Node* create_node( int value) {
        struct Node* new_node = (struct Node
*)malloc(sizeof(struct Node));
    new_node->key = value;
    new_node->left = NULL;
    new_node->right = NULL;
    return new_node;
}

struct Node* insert_left(struct Node* root, int value) {
    root->left = create_node(value);
    return root->left;
}

struct Node* insert_right(struct Node* root, int value) {
    root->right = create_node(value);
    return root->right;
}

int main() {
    struct Node* root = create_node(1);
    insert_left(root, 12);
    insert_right(root, 9);

    insert_left(root->left, 5);
    insert_right(root->left, 6);

    cout << "Inorder traversal \n";
    inorder_traversal(root);

    cout << "\nPreorder traversal \n";
```

```
    preorder_traversal(root);

    cout << "\nPostorder traversal \n";
    postorder_traversal(root);
}
```

Output:

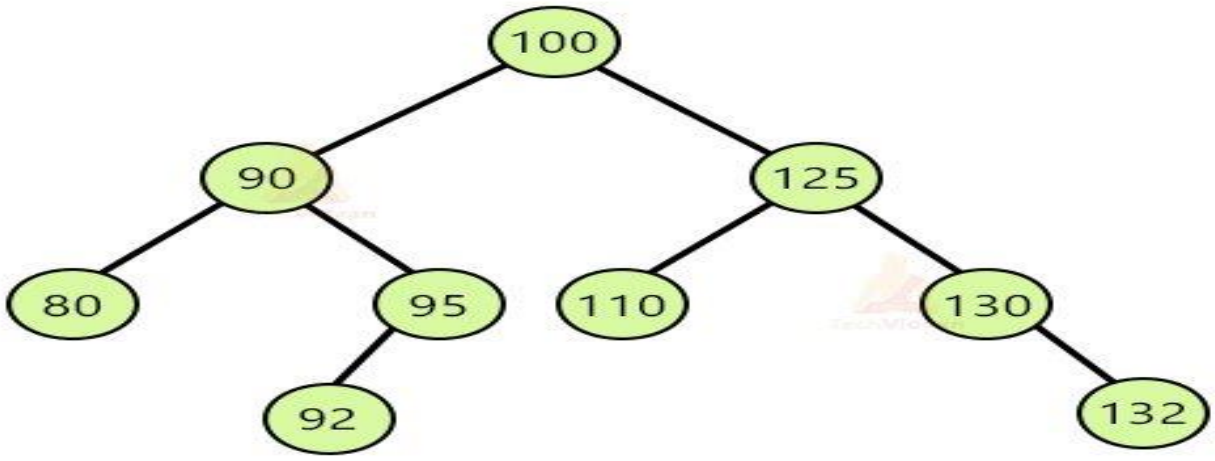
```
Inorder traversal
→5→12→6→1→9
Preorder traversal
→1→12→5→6→9
Post traversal
→5→6→12→9→1
```

Binary Search Tree in Data Structure

A binary search tree(BST) is a special type of binary tree and is also known as a sorted or ordered binary tree. In a binary search tree:

- The value of the left node is less than the value of the parent node.
- The value of the right node is greater than the value of the parent node.

Thus, in a binary search tree, all the values in the left subtree are smaller than the value at the root node and all the values in the right subtree are larger than the value at the root node. A binary search tree usually does not have any duplicate nodes. The following tree represents a binary search tree:



The benefit of using a binary search tree over a binary tree is that search operation can be performed in O(log n) time because of ordering in the BST. A binary search tree performs better in search, insert and delete operations as compared to a simple binary tree.

Representation of Binary Search Tree:

Like a binary tree, a binary search tree is also a collection of nodes where each parent node can have a maximum of two children. A node in a binary search tree has the following structure:

```
struct Node {
   int key;
   struct Node *left_child;
   struct Node *right_child;
}
```

Operations on Binary Search Tree:

1. Searching
2. Insertion
3. Deletion
4. Traversal

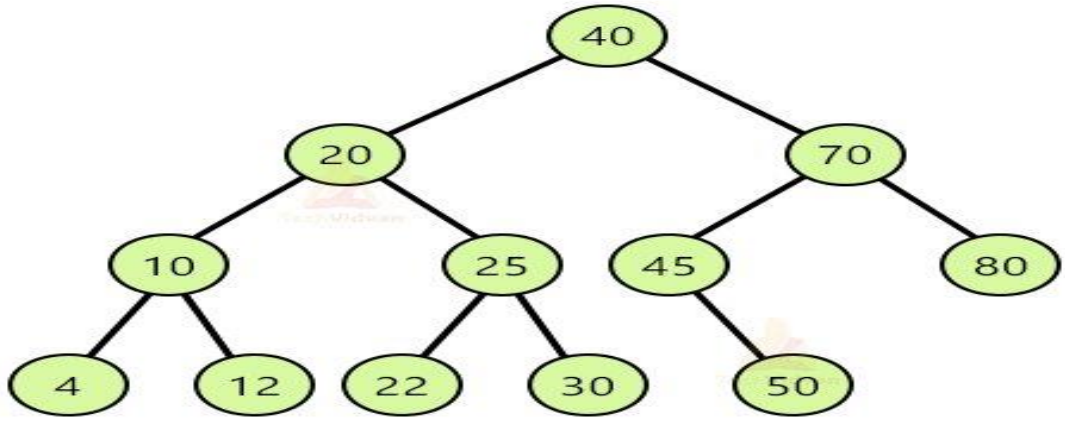
The traversal operation is of three different types:

1. Pre-order traversal: It will traverse the root first, then the left child and at last the right child.
2. Post-order traversal: It will traverse the left subtree first, then root and then the right subtree.
3. Inorder traversal: It will traverse the left subtree, then root and then the right subtree.

Search Operation in a BST:

This operation is used to search a key value in a binary search tree. Searching in a BST is very efficient because the values are ordered. While searching for a value, we first compare it with the root node. If the value is less than the root node, then it must surely be present in the left subtree. If the value is greater than the root node, then it will be present in the right subtree. Thus, we will traverse the subtrees accordingly until we reach the desired location. The search operation takes O(log n) time.

Let us understand with the help of the following example:



Suppose we wish to search 10. We will start from the root node i.e.40. The value of 40 is more than the value of 10, therefore 10 must be present in the left subtree. Thus, we will discard the right subtree and start traversing the left subtree in the same manner. Now we reach the node having a value of 20. Again 20>10, so will discard the right subtree and move further in the left subtree. Moving this direction, we finally get our desired value i.e. 10. This is how search operation works in a binary search tree.

Algorithm for Searching in a BST:

```
if (root == NULL)
    return NULL;
if (num == root->key)
    return root->key;
if (num < root->key)
    return Binary_search(root->left)
if (num > root->key)
    return Binary_search(root->right)
```

Insertion in a Binary Search Tree:

Insertion operation is used to insert values inside the tree. While inserting a value, we need to take care that the order

of the binary search tree is not disturbed. Thus, every value must be inserted at its right place according to its order.

While performing the insert operation, we first compare the value to be inserted with the root node value. If the value is less than the root node, we place it in the left subtree. If the value is more than the root node, we will place it in the right subtree. For inserting a value at the right place, we need to search the correct location first.

Algorithm for Insertion in a BST:

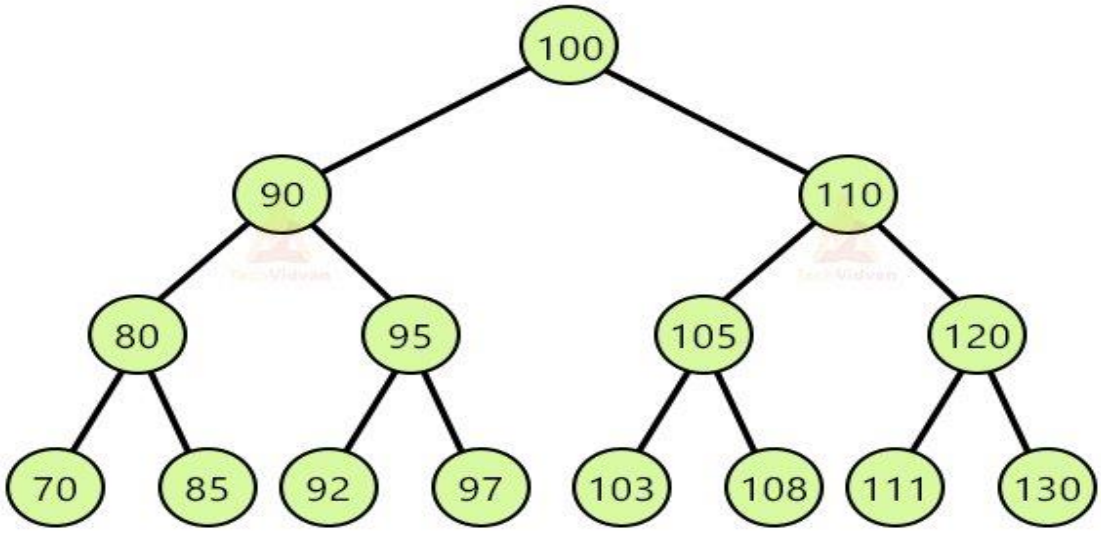
```
if (Node == NULL)

return create_node(key)

if (key < Node->key)

Node->left_child = insert(Node->left_child, key);

else if (key > Node->key)

Node->right_child = insert(Node->right_child, key);

return Node;
```

Deletion in a BST

The deletion operation includes removing a node from the BST. For performing the deletion operation, first, we have to search the location of the element. The node could be a leaf node, an internal node with one child or an internal node with two children. Let us discuss all the different cases.

Case 1: When the node is a leaf node:
If we wish to delete a leaf node, we will simply remove it without affecting the other nodes of the tree and set it to NULL. For deleting a node, we need to know its parent node.

Consider the following example:

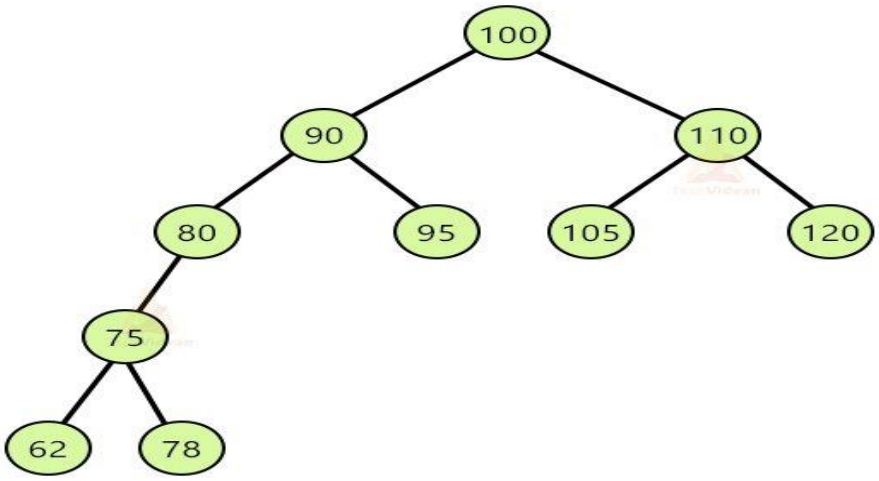


Suppose we wish to delete 70 from this tree. We will first go to the root node and compare the value of 70 to it. As 70<100, therefore, we will start looking for 70 in the left subtree. In the left subtree, we first encounter 90 which is again greater than 70. So, we will again traverse the left part of the subtree. Again 70<80, therefore we traverse the left subtree. Finally, after this step, we reach 70.

We will delete 70 from the tree and set the left pointer of 80 i.e. the parent of 70 to NULL. Thus, two operations are happening here: removing 70 from the tree and setting the left pointer of 80 to NULL. In case of deletion from the leaf node, if the height of the tree is h, then the number of comparisons will be **h+1.**

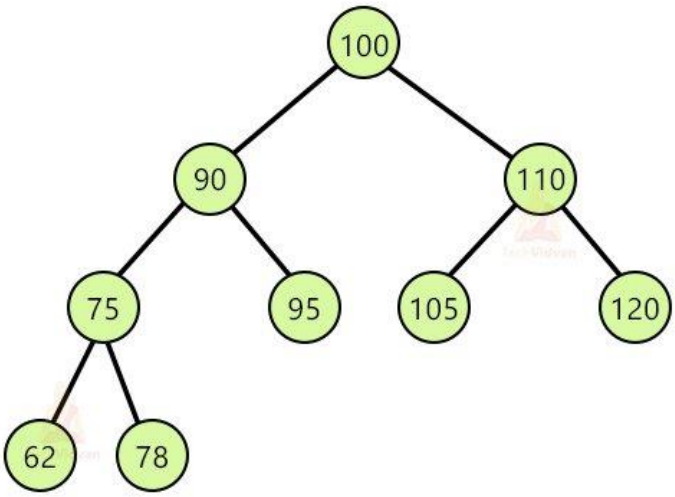
Case 2: Deletion of an internal node having one child:

Let the binary search tree be:



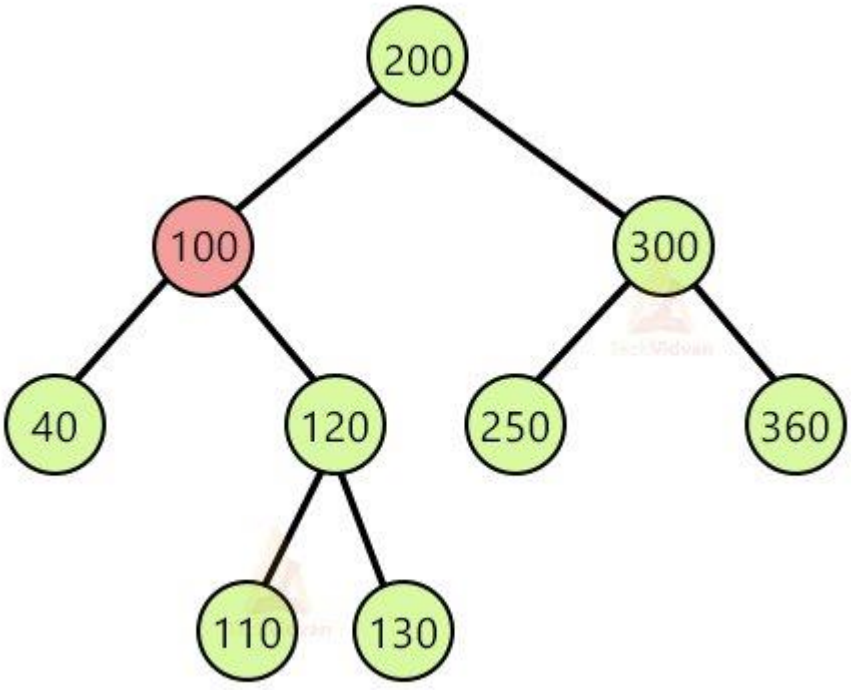
Here, 80 has only 1 child. Suppose we wish to delete 80 from the tree. To delete 80, we need to know the inorder traversal of the tree. In this case, the inorder traversal is: 62, 75,

78, **80**, 90, 95, 100, 105, 110, 120.
Once we remove 80, the inorder traversal will be: 62, 75, 78, 90, 95, 100, 105, 110, 120. Thus, after removing 80, the tree will be:

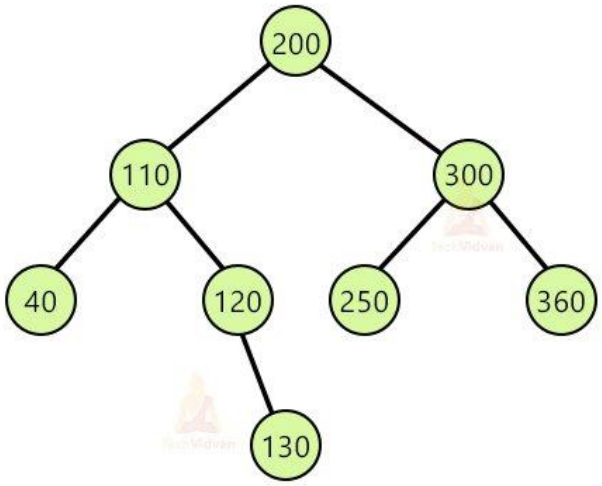


Case 3: Deletion of an internal node having two child nodes:
To delete an internal node having 2 child nodes, first find the in order successor of the node to be deleted and then swap it with the node. Now we have our node at the leaf position, therefore, we will simply remove it.

For example, consider the following tree:



Suppose we wish to remove 100 from the tree. We will find the inorder successor of 100 which comes out to be 110. We will swap 110 and 100 and then delete 100. Thus, the final tree will be:



Binary Search Tree Implementation in C++:

```
#include <bits/stdc++.h>

using namespace std;

struct Node {

int key;

struct Node *left_child, *right_child;

};

struct Node *new_node(int value) {

struct Node *temp = (struct Node *)malloc(sizeof(struct
Node));

temp->key = value;

temp->left_child = temp->right_child = NULL;

return temp;

}

void inorder_traversal(struct Node *Root) {
```

```
if (Root != NULL) {
inorder_traversal(Root->left_child);
cout<<" -> "<< Root->key;
inorder_traversal(Root->right_child);
}
}
struct Node *insertion(struct Node *Node, int key) {
if (Node == NULL)
return new_node(key);
if (key < Node->key)
Node->left_child = insertion(Node->left_child, key);
else
Node->right_child = insertion(Node->right_child, key);
return Node;
}
struct Node *min_value(struct Node *Node) {
struct Node *current = Node;
while (current && current->left_child != NULL)
current = current->left_child;
return current;
}
struct Node *deletion(struct Node *Root, int key) {
if (Root == NULL)
return Root;
if (key < Root->key)
```

```
Root->left_child = deletion(Root->left_child, key);
else if (key > Root->key)
Root->right_child = deletion(Root->right_child, key);
else {
if (Root->left_child == NULL) {
struct Node *temp = Root->right_child;
free(Root);
return temp;
}
else if (Root->right_child == NULL) {
struct Node *temp = Root->left_child;
free(Root);
return temp;
}
struct Node *temp = min_value(Root->right_child);
Root->key = temp->key;
Root->right_child = deletion(Root->right_child, temp->key);
}
return Root;
}
int main() {
struct Node *root = NULL;
root = insertion(root, 200);
root = insertion(root, 100);
root = insertion(root, 40);
```

```
root = insertion(root, 120);

root = insertion(root, 130);

root = insertion(root, 300);

root = insertion(root, 250);

root = insertion(root, 360);

root = insertion(root, 110);

cout<<"Inorder traversal: ";

inorder_traversal(root);

cout<<"\nAfter deleting 100\n";

root = deletion(root, 100);

cout<<"Inorder traversal: ";

inorder_traversal(root);

}
```

Output:

```
Inorder traversal: 40→ 100→110→120→130→200→250→300→360→

After deleting 100

Inorder traversal: 40→ 110→120→130→200→250→300→360→
```

Complexity Analysis of Binary Search Tree:

Consider a tree having n nodes. Then, the time complexity of Binary Search Tree will be:

Scenario/Operation	Search	Insert	Delete
Worst case	O(n)	O(n)	O(n)
Average case	O(log n)	O(log n)	O(log n)
Best case	O(log n)	O(log n)	O(log n)

The space complexity in all operations is O(n).

Advantages of Binary Search Tree:

- Searching is more efficient as compared to a simple binary tree.
- The insertion and deletion operations are faster in a binary search tree in comparison to arrays and linked lists.
- At each step, the binary search tree removes half the subtree and this makes it work faster.

Applications of Binary Search Tree:

- It is used in multi-level indexing in the database management system.
- Used in UNIX kernel to manage virtual memory areas
- Used in dynamic sorting of elements.

GRAPHS

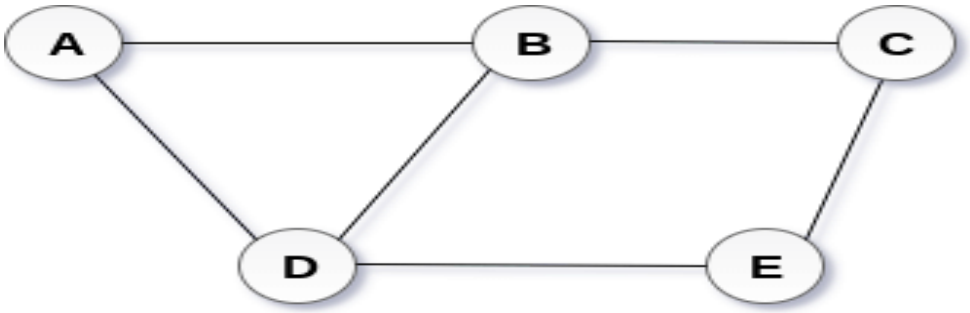
Graph

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

Definition

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

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



Undirected Graph

Directed and Undirected Graph

A graph can be directed or undirected. However, in an undirected graph, edges are not associated with the directions with them. An undirected graph is shown in the above figure since its edges are not attached with any of the directions. If an edge exists between vertex A and B then the vertices can be traversed from B to A as well as A to B.

Weighted Graph

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

Digraph

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

Loop

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

Adjacent Nodes

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

Degree of the Node

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

Graph representation

In this article, we will discuss the ways to represent the graph. By Graph representation, we simply mean the technique to be used to store some graph into the computer's memory.

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

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

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

In this tutorial, we will discuss each one of them in detail.

Now, let's start discussing the ways of representing a graph in the data structure.

Sequential representation

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

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

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

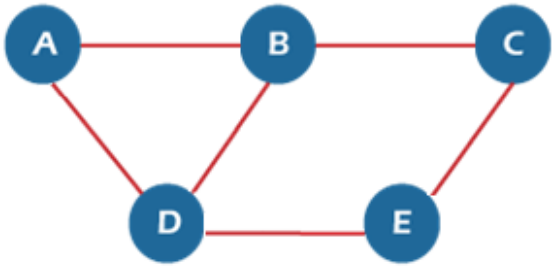
a_{ij} = 1 {if there is a path exists from V_i to V_j}

a_{ij} = 0 {Otherwise}

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

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

Now, let's see the adjacency matrix representation of an undirected graph.



Undirected Graph

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

Adjacency Matrix

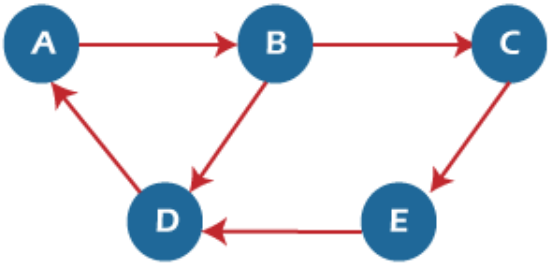
In the above figure, an image shows the mapping among the vertices (A, B, C, D, E), and this mapping is represented by using the adjacency matrix.

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

Adjacency matrix for a directed graph

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

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



Directed Graph

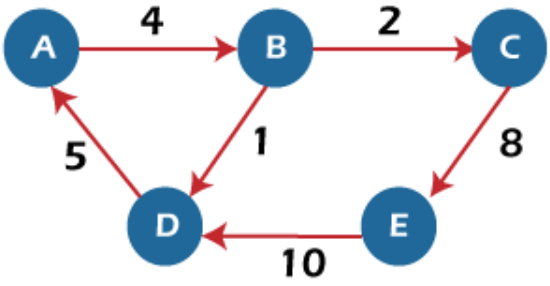
	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	1	0
C	0	0	0	0	1
D	1	0	0	0	0
E	0	0	0	1	0

Adjacency Matrix

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

Adjacency matrix for a weighted directed graph

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



weighted Directed Graph

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

Adjacency Matrix

In the above image, we can see that the adjacency matrix representation of the weighted directed graph is different from other representations. It is because, in this representation, the non-zero values are replaced by the actual weight assigned to the edges.

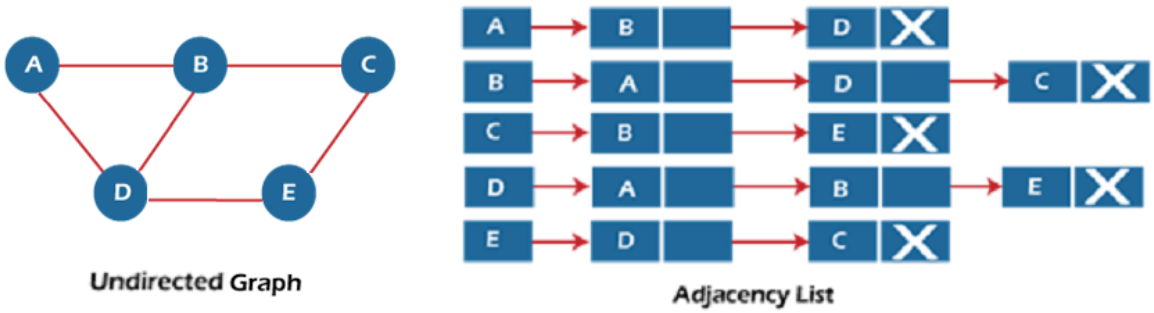
Adjacency matrix is easier to implement and follow. An adjacency matrix can be used when the graph is dense and a number of edges are large.

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

Linked list representation

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

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

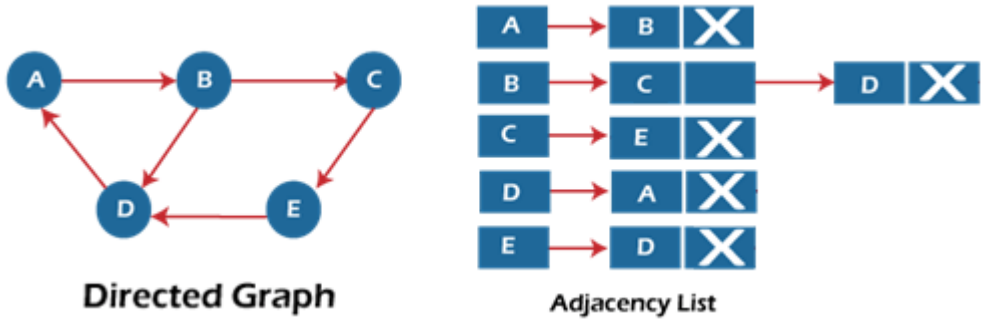


In the above figure, we can see that there is a linked list or adjacency list for every node of the graph. From vertex A, there are paths to vertex B and vertex D. These nodes are linked to nodes A in the given adjacency list.

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

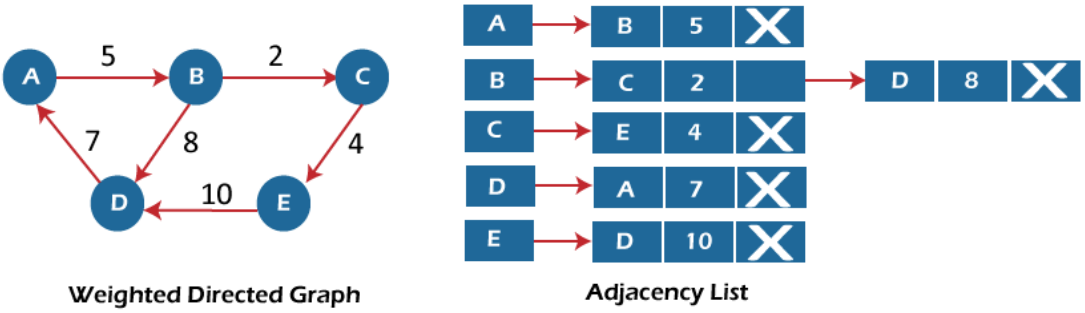
The sum of the lengths of adjacency lists is equal to twice the number of edges present in an undirected graph.

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



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

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



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

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

Implementation of adjacency matrix representation of Graph

Now, let's see the implementation of adjacency matrix representation of graph in C.

In this program, there is an adjacency matrix representation of an undirected graph. It means that if there is an edge exists from vertex A to vertex B, there will also an edge exists from vertex B to vertex A.

Here, there are four vertices and five edges in the graph that are non-directed.

```
#include <bits/stdc++.h>
using namespace std;
#include <stdio.h>
#define V 4
```

```
void init(int arr[][V]) {
  int i, j;
  for (i = 0; i < V; i++)
    for (j = 0; j < V; j++)
      arr[i][j] = 0;
}

void insertEdge(int arr[][V], int i, int j) {
  arr[i][j] = 1;
  arr[j][i] = 1;
}

void printAdjMatrix(int arr[][V]) {
  int i, j;
  for (i = 0; i < V; i++) {
    cout<<i<<" : ";
    for (j = 0; j < V; j++) {
      cout<< arr[i][j]<<" ";
    }
    cout<<"\n";
  }
}

int main() {
  int adjMatrix[V][V];
  init(adjMatrix);
  insertEdge(adjMatrix, 0, 1);
  insertEdge(adjMatrix, 0, 2);
  insertEdge(adjMatrix, 1, 2);
  insertEdge(adjMatrix, 2, 0);
  insertEdge(adjMatrix, 2, 3);
  printAdjMatrix(adjMatrix);
  return 0;
}
```

Output:

After the execution of the above code, the output will be -

```
0: 0 1 1 0
1: 1 0 1 0
2: 1 1 0 1
3: 0 0 1 0
```

Implementation of adjacency list representation of Graph

Now, let's see the implementation of adjacency list representation of graph in C.

In this program, there is an adjacency list representation of an undirected graph. It means that if there is an edge exists from vertex A to vertex B, there will also an edge exists from vertex B to vertex A.

```
#include <bits/stdc++.h>
using namespace std;
#include <stdio.h>
#include <stdlib.h>

struct AdjNode {
    int dest;
    struct AdjNode* next;
};


struct AdjList {
    struct AdjNode* head;
};
struct Graph {
   int V;
    struct AdjList* array;
};
struct AdjNode* newAdjNode(int dest)
{
    struct AdjNode* newNode = (struct AdjNode*)malloc(sizeof(struct
AdjNode));
    newNode->dest = dest;
    newNode->next = NULL;
    return newNode;
}
struct Graph* createGraph(int V)
{
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct
Graph));
    graph->V = V;
    graph->array = (struct AdjList*)malloc(V * sizeof(struct
AdjList));
    int i;
    for (i = 0; i < V; ++i)
        graph->array[i].head = NULL;
    return graph;
}

void addEdge(struct Graph* graph, int src, int dest)
{

    struct AdjNode* check = NULL;
    struct AdjNode* newNode = newAdjNode(dest);
   if (graph->array[src].head == NULL) {
        newNode->next = graph->array[src].head;
        graph->array[src].head = newNode;
    }
    else {
        check = graph->array[src].head;
```

```
        while (check->next != NULL) {
            check = check->next;
        }

        check->next = newNode;
    }

    newNode = newAdjNode(src);
    if (graph->array[dest].head == NULL) {
        newNode->next = graph->array[dest].head;
        graph->array[dest].head = newNode;
    }
    else {
        check = graph->array[dest].head;
        while (check->next != NULL) {
            check = check->next;
        }
        check->next = newNode;
    }
}

void print(struct Graph* graph)
{
    int v;
    for (v = 0; v < graph->V; ++v) {
        struct AdjNode* pCrawl = graph->array[v].head;
        cout<<"\n The Adjacency list of vertex "<< v << " is: \n
head ";
        while (pCrawl) {
            cout<<"-> "<< pCrawl->dest;
            pCrawl = pCrawl->next;
        }
        cout<<"\n";
    }
}
int main()
{
    int V = 4;
    struct Graph* g = createGraph(V);
    addEdge(g, 0, 1);
    addEdge(g, 0, 3);
    addEdge(g, 1, 2);
    addEdge(g, 1, 3);
    addEdge(g, 2, 4);
    addEdge(g, 2, 3);
    addEdge(g, 3, 4);
    print(g);
    return 0;
}
```

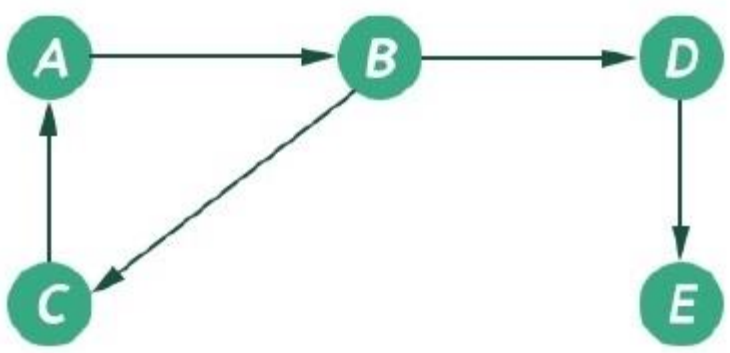
Output:

In the output, we will see the adjacency list representation of all the vertices of the graph. After the execution of the above code, the output will be -

```
The Adjacency list of vertex 0 is:
head -> 1-> 3

The Adjacency list of vertex 1 is:
head -> 0-> 2-> 3

The Adjacency list of vertex 2 is:
head -> 1-> 4-> 3

The Adjacency list of vertex 3 is:
head -> 0-> 1-> 2-> 4
```

Graph Traversal

The graph is one non-linear data structure. That is consists of some nodes and their connected edges. The edges may be director 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)})



The graph has two types of traversal algorithms. These are called the Breadth First Search and Depth First Search.

Breadth First Search (BFS)

The Breadth First Search (BFS) traversal is an algorithm, which is used to visit all of the nodes of a given graph. In this traversal algorithm one node is selected and then all of the adjacent nodes are visited one by one. After completing all of the adjacent vertices, it moves further to check another vertices and checks its adjacent vertices again.

A standard BFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The algorithm works as follows:

1. Start by putting any one of the graph's vertices at the back of a queue.
2. Take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.

The graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the BFS algorithm on every node

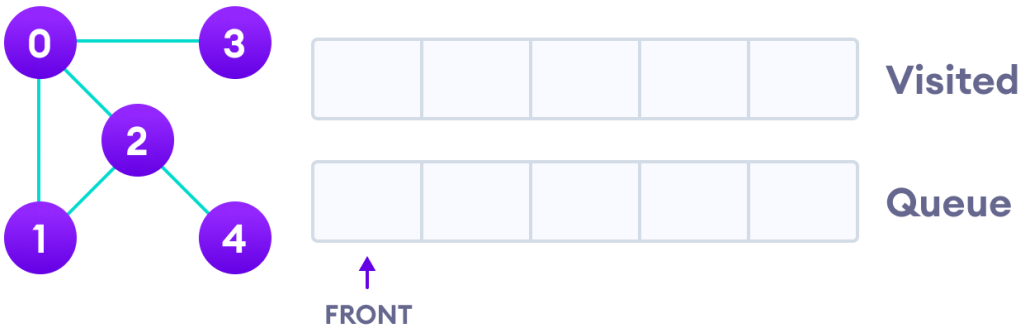
Algorithm

```
bfs(vertices, start)
Input: The list of vertices, and the start vertex.
Output: Traverse all of the nodes, if the graph is connected.
Begin
   define an empty queue que
   at first mark all nodes status as unvisited
   add the start vertex into the que
   while que is not empty, do
      delete item from que and set to u
      display the vertex u
      for all vertices 1 adjacent with u, do
         if vertices[i] is unvisited, then
            mark vertices[i] as temporarily visited
            add v into the queue
```

```
          mark
       done
       mark u as completely visited
   done
End
```

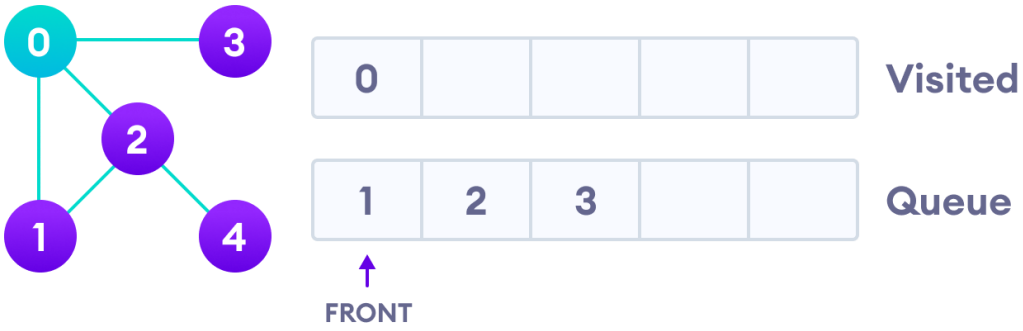
BFS example

Let's see how the Breadth First Search algorithm works with an example. We use an undirected graph with 5 vertices.



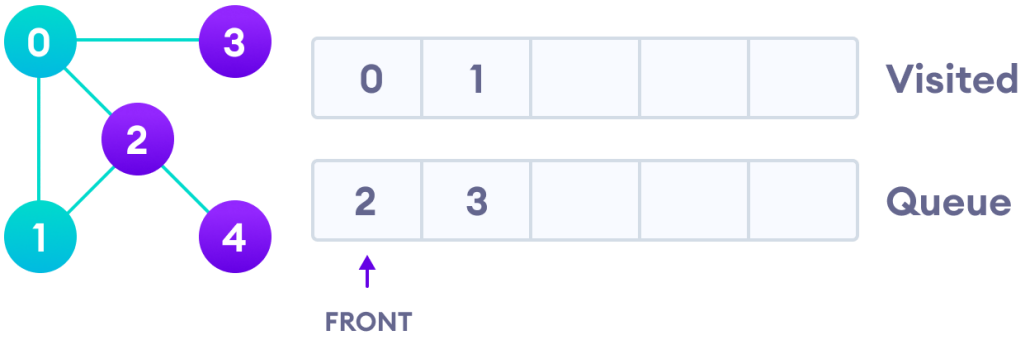
Undirected graph with 5 vertices

We start from vertex 0, the BFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.



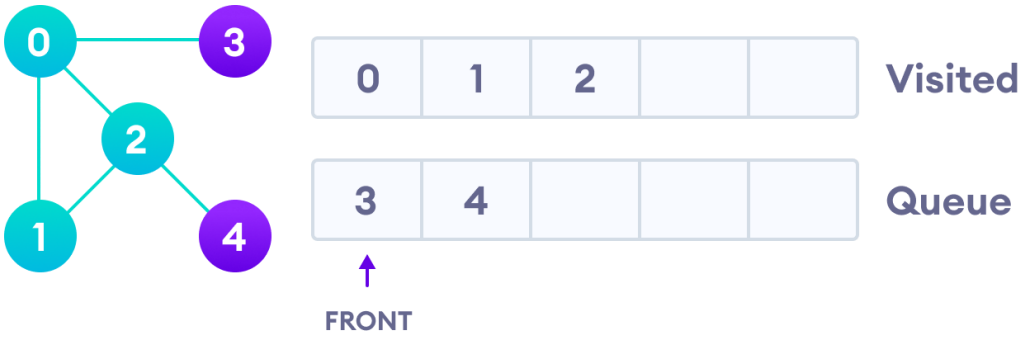
Visit start vertex and add its adjacent vertices to queue

Next, we visit the element at the front of queue i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.

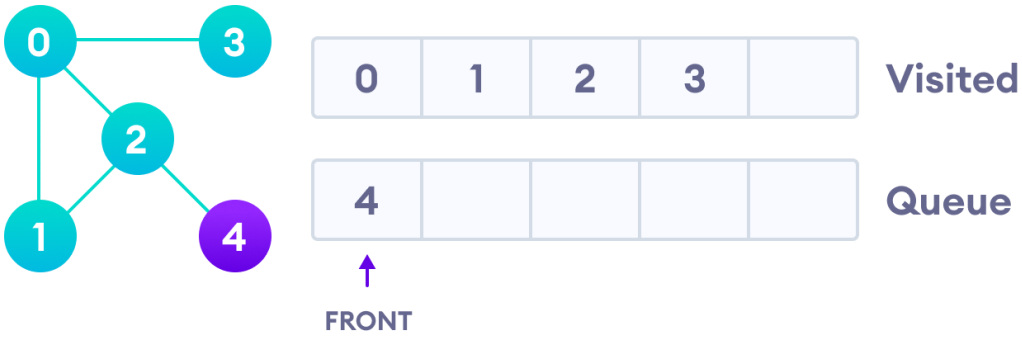


Visit the first neighbour of start node 0, which is 1

Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the back of the queue and visit 3, which is at the front of the queue.



Visit 2 which was added to queue earlier to add its neighbours



4 remains in the queue

Only 4 remains in the queue since the only adjacent node of 3 i.e. 0 is already visited. We visit it.

Visit last remaining item in the queue to check if it has unvisited neighbors

Since the queue is empty, we have completed the Breadth First Traversal of the graph.

BFS pseudocode

```
create a queue Q

mark v as visited and put v into Q

while Q is non-empty

    remove the head u of Q

    mark and enqueue all (unvisited) neighbours of u
```

Implementation of BFS in C++

```cpp
#include <iostream>
#include <queue>
#include <vector>
using namespace std;
#define N 10
vector<int> graph[N];
bool vis[N];
int main() {
    for(int i=0;i<=N;i++)
        vis[i]=0;
    int n,m;
    cin>>n>>m;
    int x,y;
    for(int i=0;i<m;i++)
    {
        cin>>x>>y;
        graph[x].push_back(y);
        graph[y].push_back(x);
    }
    queue<int> q;
    q.push(1);
    vis[1]=true;
    while(!q.empty())
    {
        int node=q.front();
        q.pop();
        cout<<node<<"->";
        vector<int>::iterator it;
        for(it=graph[node].begin();it!=graph[node].end();it++)
        {
            if(!vis[*it])
            {
                vis[*it]=1;
                q.push(*it);
```

```
            }
        }
    }

  return 0;
}
```

Depth First Search (DFS)

The Depth First Search (DFS) is a graph traversal algorithm. In this algorithm one starting vertex is given, and when an adjacent vertex is found, it moves to that adjacent vertex first and try to traverse in the same manner.

Working principle:

A standard DFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

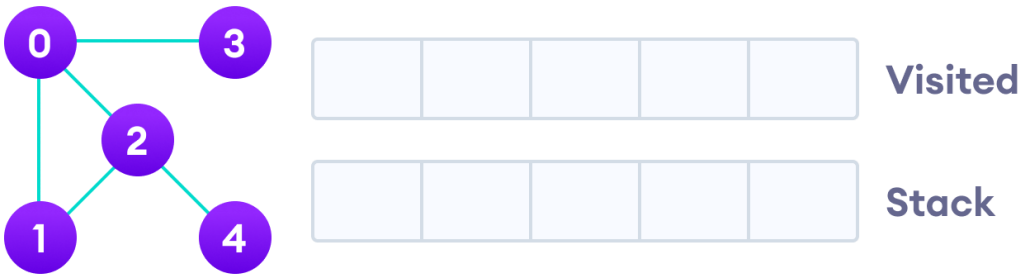
The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The DFS algorithm works as follows:

1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

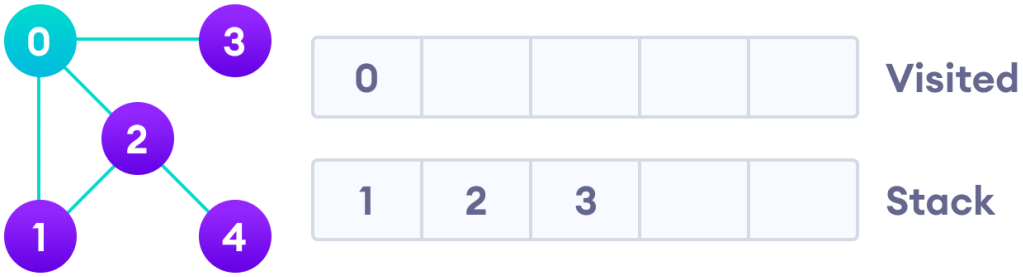
Depth First Search Example

Let's see how the Depth First Search algorithm works with an example. We use an undirected graph with 5 vertices.



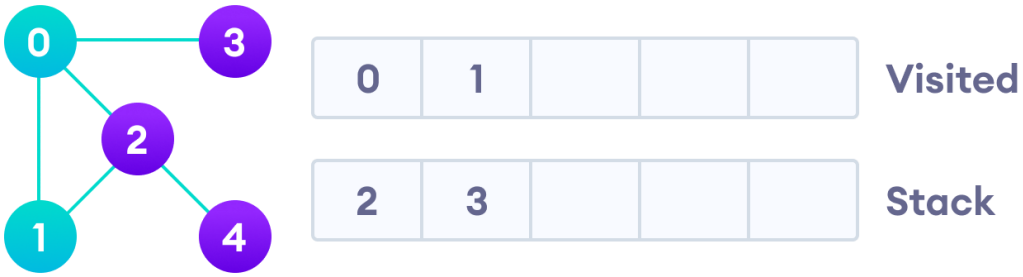
Undirected graph with 5 vertices

We start from vertex 0, the DFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.



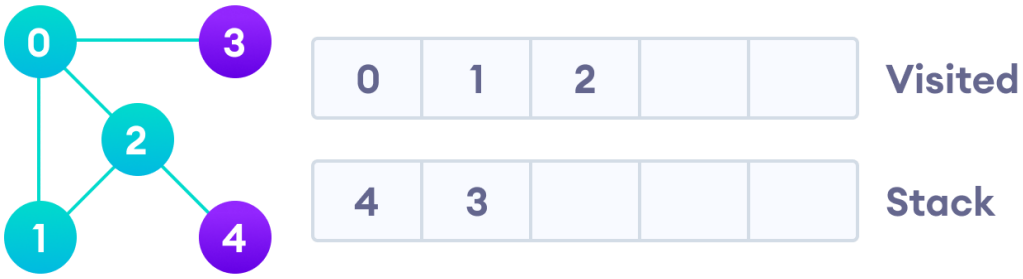
Visit the element and put it in the visited list

Next, we visit the element at the top of stack i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.

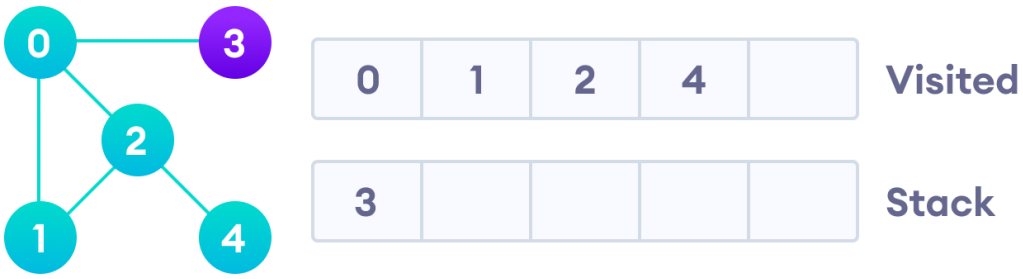


Visit the element at the top of stack

Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.

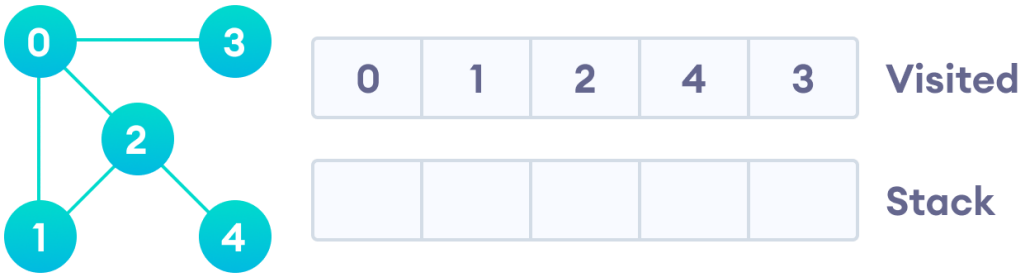


Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.



Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.

After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.



After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.

DFS Pseudocode (recursive implementation)

The pseudocode for DFS is shown below. In the init() function, notice that we run the DFS function on every node. This is because the graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the DFS algorithm on every node.

```
DFS(G, u)
    u.visited = true
    for each v ∈ G.Adj[u]
        if v.visited == false
            DFS(G,v)
init() {
    For each u ∈ G
        u.visited = false
     For each u ∈ G
       DFS(G, u)
}
```

Algorithm

```
dfs(vertices, start)
```

```
Input: The list of all vertices, and the start node.
Output: Traverse all nodes in the graph.
Begin
   initially make the state to unvisited for all nodes
   push start into the stack
   while stack is not empty, do
      pop element from stack and set to u
      display the node u
      if u is not visited, then
         mark u as visited
         for all nodes i connected to u, do
            if ith vertex is unvisited, then
               push ith vertex into the stack
               mark ith vertex as visited
         done
   done
End
```

Implementation of DFS in C++

```
#include <iostream>
#include <list>
using namespace std;
class Graph {
  int numVertices;
  list<int> *adjLists;
  bool *visited;

   public:
  Graph(int V);
  void addEdge(int src, int dest);
  void DFS(int vertex);
};

Graph::Graph(int vertices) {
  numVertices = vertices;
  adjLists = new list<int>[vertices];
```

```
  visited = new bool[vertices];
}

void Graph::addEdge(int src, int dest) {
  adjLists[src].push_front(dest);
}

void Graph::DFS(int vertex) {
  visited[vertex] = true;
  list<int> adjList = adjLists[vertex];
 cout << vertex << " ";
  list<int>::iterator i;
  for (i = adjList.begin(); i != adjList.end(); ++i)
    if (!visited[*i])
      DFS(*i);
}
int main() {
  Graph g(4);
  g.addEdge(0, 1);
  g.addEdge(0, 2);
  g.addEdge(1, 2);
  g.addEdge(2, 3);
  g.DFS(2);

  return 0;
}
```