UNIT IV

Trees: Introduction, Binary Trees, Representation of Binary Tree, Binary Tree Traversal, Binary Search Tree, Operations on Binary Search Tree, Heap tree, B-tree.

Graphs: Terminology, Types, Representation of Graph, Elementary Graph operations- DFS and BFS.

4.1 Introduction to Trees

Q1. What is tree data structure? Write about the basic terminology of tree data structure. **(Imp.)**

Ans :

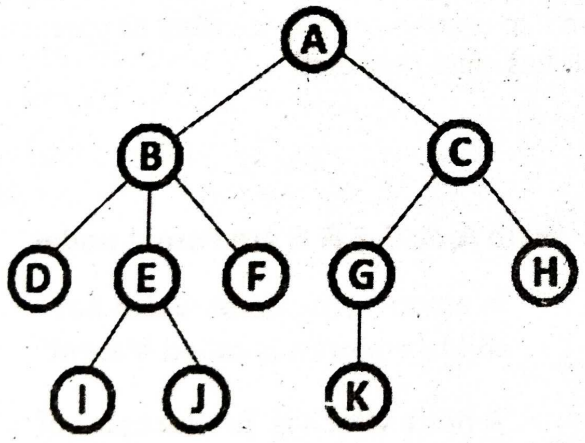
A tree data structure can also be defined as follows.

Tree data structure is a collection of data (Node) which is organized in hierarchical structure and this is a recursive definition

In tree data structure, every individual element is called as Node. Node in a tree data structure, stores the actual data of that particular element and link to next element in hierarchical structure.

In a tree data structure, if we have N number of nodes then we can have a maximum of N-1 number of links.

Example



TREE with 11 nodes and 10 edges

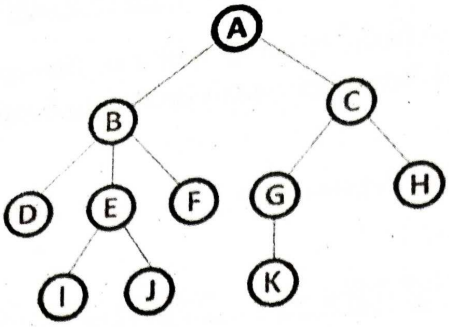
- In any tree with 'N' nodes there will be maximum of 'N-1' edges
- In a tree every individual element is called as 'NODE'

Terminology

In a tree data structure, we use the following terminology...

1. **Root**

In a tree data structure, the first node is called as Root Node. Every tree must have root node. We can say that root node is the origin of tree data structure. In any tree, there must be only one root node. We never have multiple root nodes in a tree.

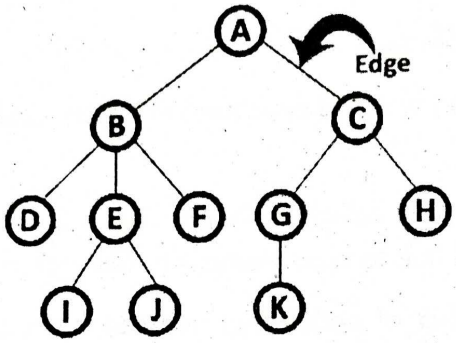


Here 'A' is the 'root' node

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

2. Edge

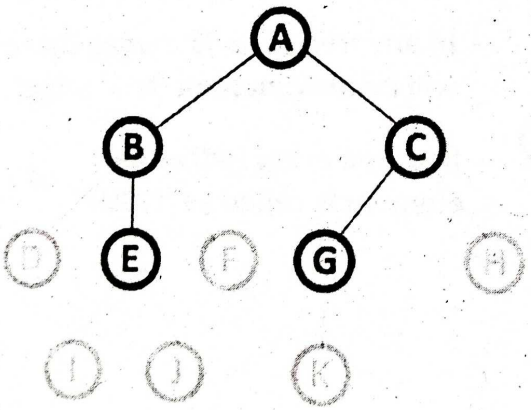
In a tree data structure, the connecting link between any two nodes is called as EDGE. In a tree with 'N' number of nodes there will be a maximum of 'N-1' number of edges.



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

3. Parent

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

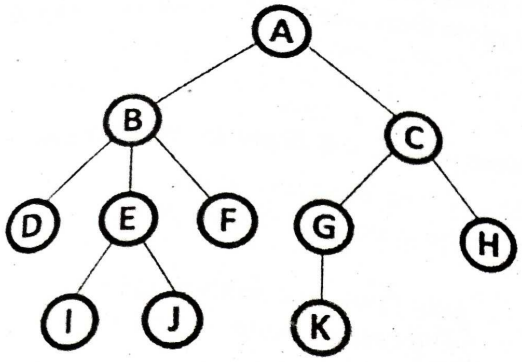


Here A, B, C, E & G are Parent nodes

- **In any tree the 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.

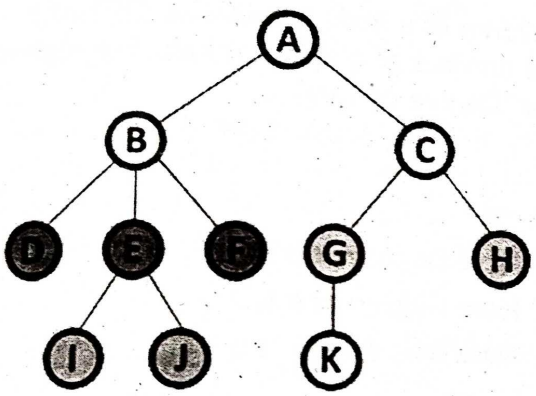


Here B & C are Children of A
Here G & H are Children of C
Here K is Child of G

- descendant of any node is called as **CHILD Node**

Siblings

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



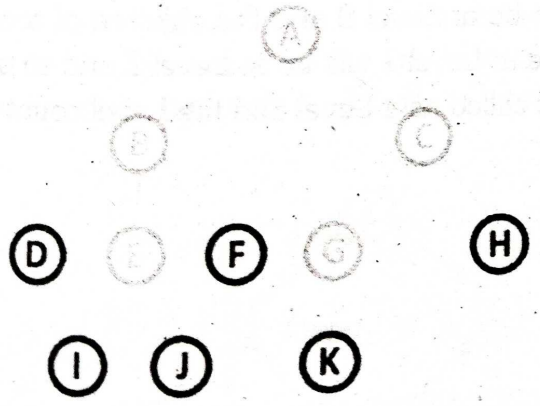
Here are Siblings
Here D E & F are Siblings
Here are Siblings
Here I & J are Siblings

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

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.



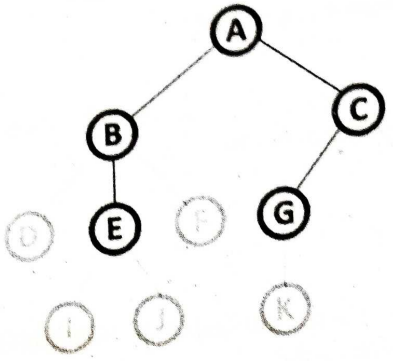
Here D, I, J, F, K & H are Leaf nodes

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

Internal Nodes

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

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

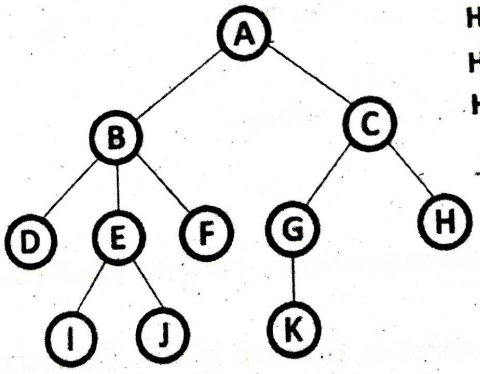


Here A, B, C, E & G are Internal nodes

- In any tree the node which has atleast one child is called **'Internal' node**
- Every non-leaf node is called as **'Internal' node**

8. Degree

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

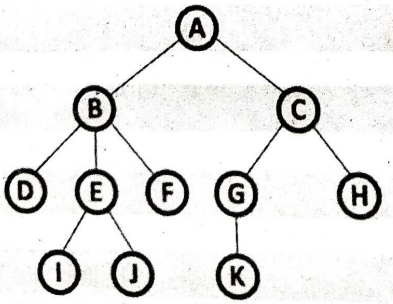


Here Degree of B is 3
Here Degree of A is 2
Here Degree of F is 0

- In any tree, 'Degree' a node is total number of children it has.

9. Level

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

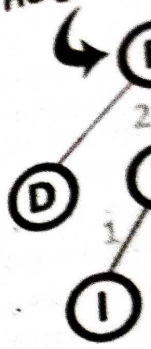


Level 0
Level 1
Level 2
Level 3

10. Height

In a tree data stru
path is called as
the tree. In a tree

Height is



11. Depth

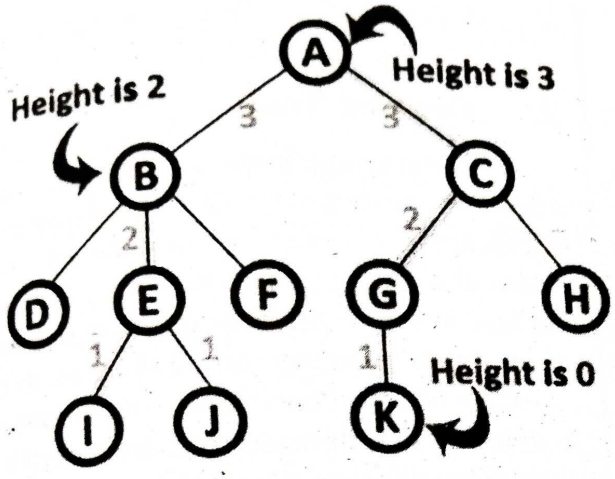
In a tree data
as DEPTH o
longest path i
a tree is said

12. Path

In a tree da
as PATH t
below exan

10. Height

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

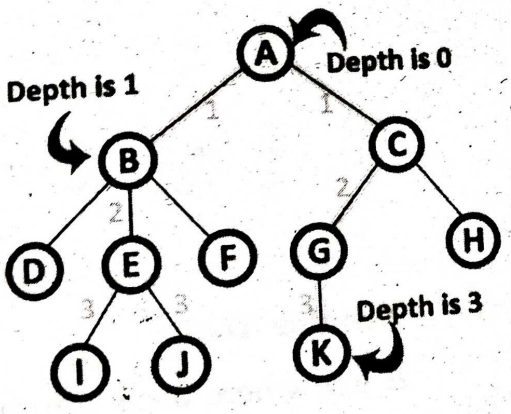


Here Height of tree is 3

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

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'.

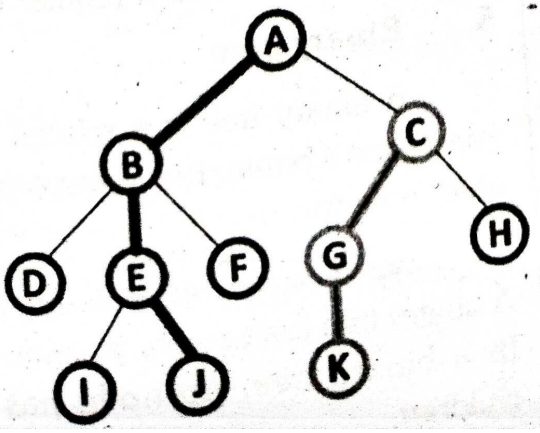


Here Depth of tree is 3

- 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.



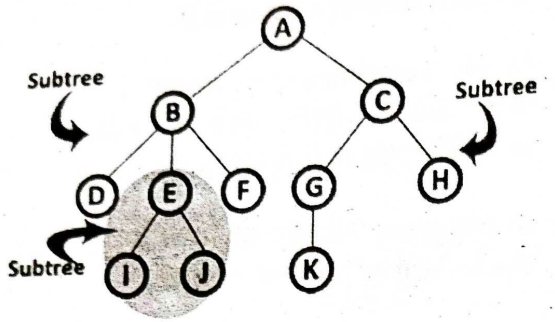
- In any tree, 'Path' is a sequence of nodes and edges between two nodes.

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.



3.1.2 Types of Trees

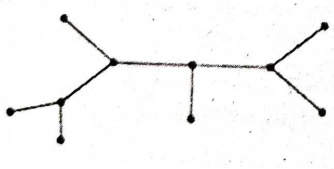
Q2. Write about various types of trees ?

Ans :

1. Free tree
2. Rooted tree
3. Ordered tree
4. Regular tree
5. Binary tree
6. Complete tree
7. Position tree

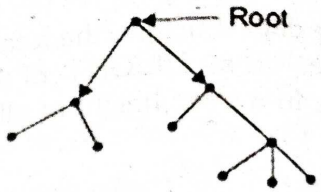
1. Free tree

A free tree is a connected, acyclic graph. It is an undirected graph. It has no node designated as a root. As it is connected, any node can be reached from any other node through a unique path. The following is an example of a free tree.



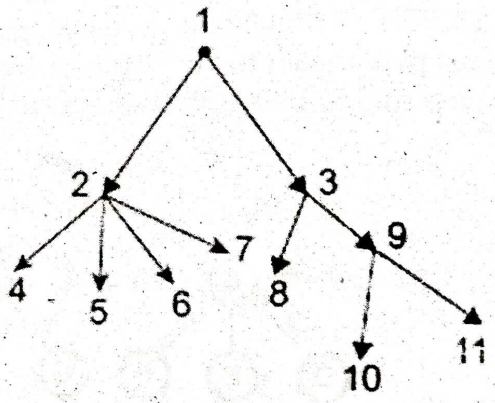
2. Rooted Tree

Unlike free tree, a rooted tree is a directed graph where one node is designated as root, whose incoming degree is zero, whereas for all other nodes, the incoming degree is one.



3. Ordered Tree

In many applications, the relative order of the nodes at any particular level assumes some significance. It is easy to impose an order on the nodes at a level by referring to a particular node as the first node, to another node as the second, and so on. Such ordering can be done from left to right. Just like nodes at each level, we can prescribe order to edges. If in a directed tree, an ordering of a node at each level is prescribed, then such a tree is called an *ordered tree.*



4. Regular Tree

A tree where each branch node vertex has the same outdegree is called a *regular tree.* If in a directed tree, the outdegree of every node is less than or equal to *m*, then the tree is called an *m-ary tree.* If the outdegree of every node is exactly equal to *m* (the branch nodes) or zero (the leaf nodes), then the tree is called a regular m-ary tree.

5. Binary tree

A binary tree is a special form of an *m*-ary tree. Since a binary tree is important, it is frequently used in various applications of computer science.

We have defined an *m*-ary tree (general tree). A binary tree is an *m*-ary position tree when $m = 2$. In a binary tree, no node has more than two children.

6. Complete tree

A tree with n nodes and of depth k is complete if and only if its nodes correspond to the nodes that are numbered from 1 to n in the full tree of depth k.

A binary tree of height h is complete if and only if one of the following holds good:

i) It is empty.

ii) Its left subtree is complete of height $h>=1$ and its right subtree is completely full of height $h=2$.

iii) Its left subtree is completely full of height $h>=1$ and its right subtree is complete of height $h=1$.

A binary tree is completely full if it is of height h and has $(2h+1 -1)$ nodes.

Full Binary Tree

A binary tree is a full binary tree if it contains the maximum possible number of nodes in all levels. Figure shows a full binary tree of height 2.

In a full binary tree, each node has two children or no child at all. The total number of nodes in a full binary tree of height h is $2h+1-1$ considering the root at level 0.

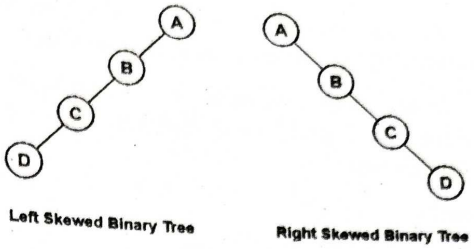
It can be calculated by adding the number of nodes of each level as in the following equation: $20 <21 <22 <... <2h <2h+1 -1$

Complete Binary Tree

A binary tree is said to be a complete binary tree if all its levels except the last level have the maximum number of possible nodes, and all the nodes of the last level appear as far left as possible. In a complete binary tree, all the leaf nodes are at the last and the second last level, and the levels are filled from left to right.

Left skewed binary tree

If the right subtree is missing in every node of a tree, we call it a left skewed tree . If the left subtree is missing in every node of a tree, we call it as right subtree.



Left Skewed Binary Tree Right Skewed Binary Tree

Strictly Binary Tree

If every non-terminal node in a binary tree consists of non-empty left and right subtrees, then such a tree is called a strictly binary tree.

In Fig., the non-empty nodes D and E have left and right subtrees. Such expression trees are known as strictly binary trees.

Extended Binary Tree

A binary tree T with each node having zero or two children is called an extended binary tree. The nodes with two children are called internal nodes, and those with zero children are called external nodes. Trees can be converted into extended trees by adding a node.

7. Position Tree

A position tree, also known as a suffix tree, is one that represents the suffixes of a string S and such representation facilitates string operations being performed faster. Such a tree's edges are labelled with strings, such that each suffix of S corresponds to exactly one path from the tree's root to a leaf node. The space and time requirement is linear in the length of S. After its construction, several operations can be performed quickly, such as locating a substring in S, locating a substring if a certain number of mistakes are allowed, locating matches for a regular expression pattern, and so.

4.1.1 Binary Trees

Q3. What is binary tree? (Imp.)

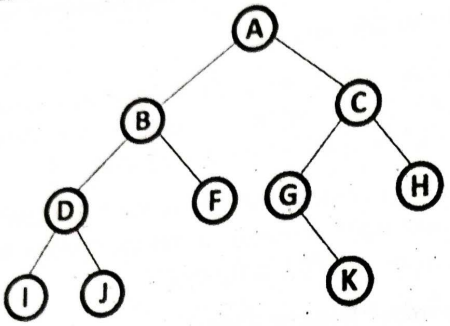
Ans :

In a normal tree, every node can have any number of children. Binary tree is a special type of tree data structure in which every node can have a maximum of 2 children. One is known as left child and the other is known as right child.

A tree in which every node can have a maximum of two children is called as Binary Tree.

In a binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2 children.

Example



Q4. Write briefly about various types of binary trees? **(Imp.)**

Ans :

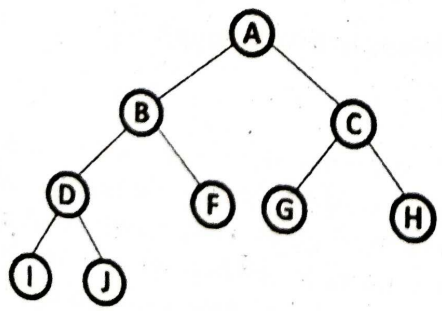
There are different types of binary trees and they are...

1. Strictly Binary Tree

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none. That means every internal node must have exactly two children. A strictly Binary Tree can be defined as follows...

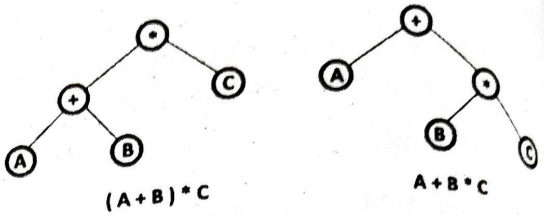
A binary tree in which every node has either two or zero number of children is called Strictly Binary Tree

Strictly binary tree is also called as Full Binary Tree or Proper Binary Tree or 2-Tree



Strictly binary tree data structure is used to represent mathematical expressions.

Example

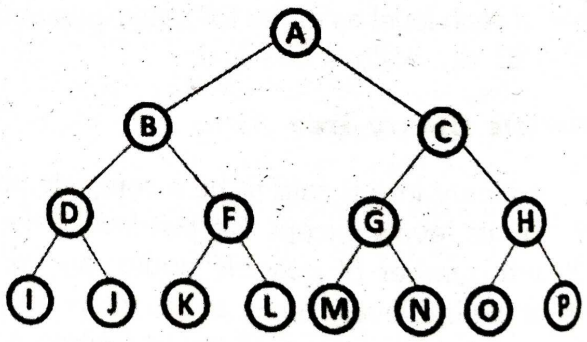


2. Complete Binary Tree

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none and in complete binary tree all the nodes must have exactly two children and at every level of complete binary tree there must be 2^{level} number of nodes. For example at level 2 there must be $2^2 = 4$ nodes and at level 3 there must be $2^3 = 8$ nodes.

A binary tree in which every internal node has exactly two children and all leaf nodes are at same level is called Complete Binary Tree.

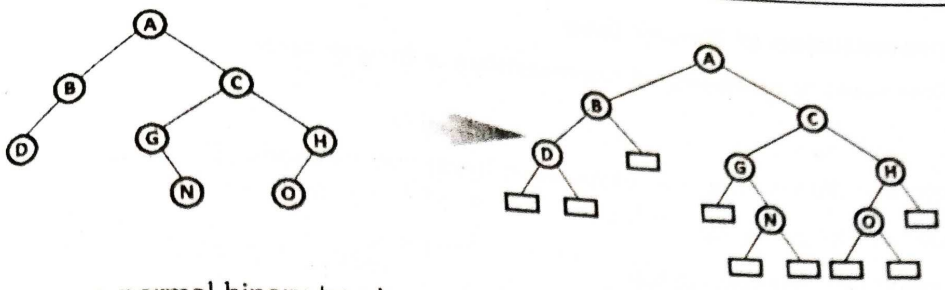
Complete binary tree is also called as Perfect Binary Tree.



3. Extended Binary Tree

A binary tree can be converted into Full Binary tree by adding dummy nodes to existing nodes wherever required.

The full binary tree obtained by adding dummy nodes to a binary tree is called as Extended Binary Tree.



In above figure, a normal binary tree is converted into full binary tree by adding dummy nodes (In pink colour).

Q5. Explain ADT of Binary Tree.

Ans :

A binary tree consists of nodes with one predecessor and at most two successors (called the left child and the right child). The only exception is the root node of the tree that does not have a predecessor. A node can contain any amount and any type of data.

- **create** - A binary tree may be created in several states. The most common include ...
  - an empty tree (i.e. no nodes) and hence the constructor will have no parameters
  - a tree with one node (i.e. the root) and hence the constructor will have a single parameter (the data for the root node)
  - a tree with a new root whose children are other existing trees and hence the constructor will have three parameters (the data for the root node and references to its subtrees)
- **isEmpty()** - Returns true if there are no nodes in the tree, false otherwise.
- **isFull()** - May be required by some implementations. Returns true if the tree is full, false otherwise.
- **clear()** - Removes all of the nodes from the tree (essentially reinitializing it to a new empty tree).
- **add(value)** - Adds a new node to the tree with the given value. The actual implementation of this method is determined by the purpose for the tree and how the tree is to be maintained and/or processed. To begin with, we will assume no particular purpose or order and therefore add new nodes in such a way that the tree will remain nearly balanced.

remove() - Removes the root node of the tree and returns its data. The actual implementation of this method and other forms of removal will be determined by the purpose for the tree and how the tree is to be maintained and/or processed. For example, you might need a remove() method with a parameter that indicates which node of the tree is to be removed (based on position, key data, or reference). To begin with, we will assume no particular purpose or order and therefore remove the root in such a way that the tree will remain nearly balanced.

Other operations that may be included as needed:

- **height()** - Determines the height of the tree. An empty tree will have height 0.
- **size()** - Determines the number of nodes in the tree.
- **getRootData()** - Returns the data (primative data) or reference to the data (objects) of the tree's root.
- **getLeftSubtree()** - Returns a reference to the left subtree of this tree.
- **getRightSubtree()** - Returns a reference to the right subtree of this tree.

4.1.2 Representation of Binary Tree

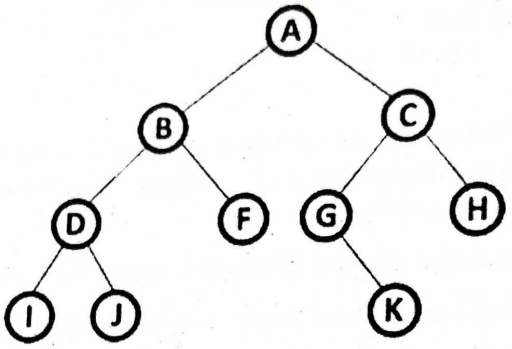
Q6. Explain various methods of representing a binary tree. (Imp

Ans :

A binary tree data structure is represented using two methods. Those methods are as follows...

1. Array Representation
2. Linked List Representation

Consider the following binary tree...



1. Array Representation

In array representation of binary tree, we use a one dimensional array (1-D Array) to represent a binary tree.

Consider the above example of binary tree and it is represented as follows...

A	B	C	D	F	G	H	I	J	-	-	-	K	-	-	-	-	-	-	-	-

To represent a binary tree of depth **'n'** using array representation, we need one dimensional array with a maximum size of $2^{n+1} - 1$.

Advantages

The various merits of representing binary trees using arrays are as follows:

1. Any node can be accessed from any other node by calculating the index.
2. Here, the data is stored without any pointers to its successor or predecessor.
3. In the programming languages, where dynamic memory allocation is not possible (such as BASIC, fortran), array representation is the only means to store a tree.

Disadvantages

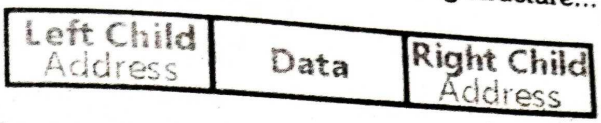
The various demerits when representing binary trees using arrays are as follows:

1. Other than full binary trees, majority of the array entries may be empty.
2. It allows only static representation. The array size cannot be changed during the execution.
3. Inserting a new node to it or deleting a node from it is inefficient with this representation, because it requires considerable data movement up and down the array, which demand excessive amount of processing time.

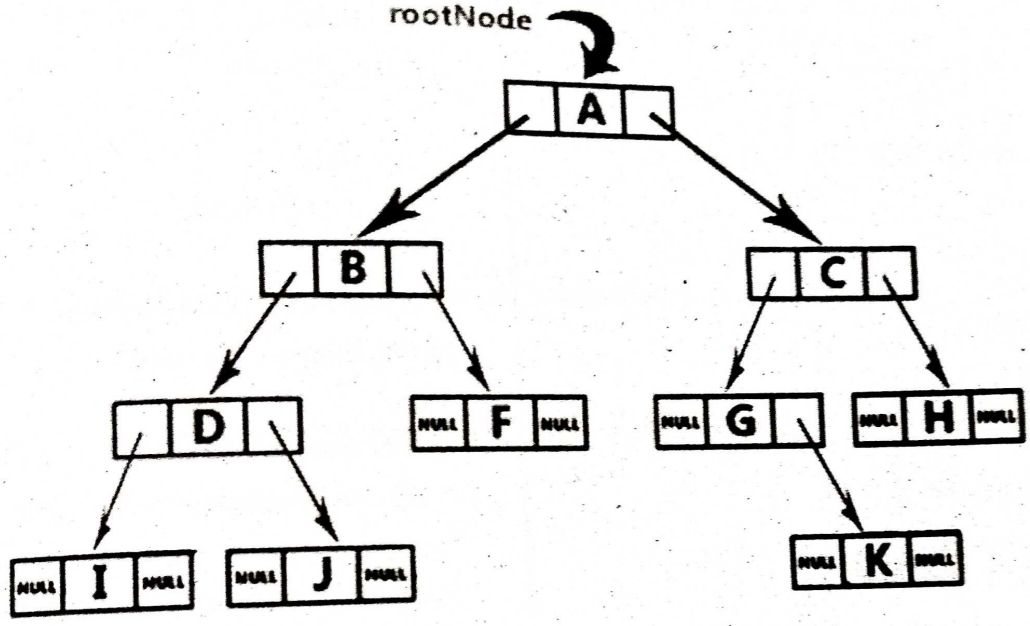
Linked List Representation

We use double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address.

In this linked list representation, a node has the following structure...



The above example of binary tree represented using Linked list representation is shown as follows...



Q7. Write a C++ program to represent binary tree using array. **(Imp.)**

Ans :

```
#include<bits/stdc++.h>
using namespace std;
char tree[10];
int rootnode(char key){
   if(tree[0] != '\0')
      cout<<"Tree already had root";
   else
      tree[0] = key;
   return 0;
}
int leftchild(char key, int parent){
   if(tree[parent] == '\0')
```

```
        cout <<"\nCan't set child at"<<(parent * 2)
    + 1<<" , no parent found";
    else
            tree[(parent * 2) + 1] = key;
        return 0;
    }
    int rightchild(char key, int parent){
        if(tree[parent] == '\0')
        cout<<"\nCan't set child at"<<(parent * 2)
+       2<<" , no parent found";
        else
            tree[(parent * 2) + 2] = key;
        return 0;
    }
    int traversetree(){
        cout << "\n";
        for(int i = 0; i < 10; i++){
            if(tree[i] != '\0')
            cout<<tree[i];
            else
            cout<<"-";
        }
        return 0;
}
int main(){
    rootnode('A');
    rightchild('C', 2);
    leftchild('D', 0);
    rightchild('E', 1);
    rightchild('F', 2);
    traversetree();
    return 0;
}
```

Q8. Write aC++ program to create a Complete Binary tree from its Linked List Representation.

Ans :

// C++ program to create a Complete Binary tree from its Linked List Representation

```
#include <iostream>
#include <string>
#include <queue>
using namespace std;
// Linked list node
struct ListNode
{
    int data;
    ListNode* next;
};
// Binary tree node structure
struct BinaryTreeNode
{
    int data;
    BinaryTreeNode *left, *right;
};
// Function to insert a node at the beginning of
the Linked List
void push(struct ListNode** head_ref, int
new_data)
{
    // allocate node and assign data
    struct ListNode* new_node = new ListNode;
    new_node->data = new_data;

    // link the old list off the new node
    new_node->next = (*head_ref);

    // move the head to point to the new node
    (*head_ref)    = new_node;
}
// method to create a new binary tree node from
the given data
BinaryTreeNode* newBinaryTreeNode(int data)
{
```

```
    BinaryTreeNode *temp = new BinaryTreeNode;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// converts a given linked list representing a complete binary tree into the
// linked representation of binary tree.
void convertList2Binary(ListNode *head, BinaryTreeNode* &root)
{
    // queue to store the parent nodes
    queue<BinaryTreeNode *> q;

    // Base Case
    if (head == NULL)
    {
        root = NULL; // Note that root is passed by reference
        return;
    }
    // 1.) The first node is always the root node, and add it to the queue
    root = newBinaryTreeNode(head->data);
    q.push(root);

    // advance the pointer to the next node
    head = head->next;

    // until the end of linked list is reached, do the following steps
    while (head)
    {
        // 2.a) take the parent node from the q and remove it from q
        BinaryTreeNode* parent = q.front();
        q.pop();

        // 2.c) take next two nodes from the linked list. We will add
        // them as children of the current parent node in step 2.b. Push them
        // into the queue so that they will be parents to the future nodes
        BinaryTreeNode *leftChild = NULL, *rightChild = NULL;
        leftChild = newBinaryTreeNode(head->data);
        q.push(leftChild);
        head = head->next;
```

```
            if (head)
            {
                rightChild = newBinaryTreeNode(head->data);
                q.push(rightChild);
                head = head->next;
            }
            // 2.b) assign the left and right children of parent
            parent->left = leftChild;
            parent->right = rightChild;
        }
    }
    // Utility function to traverse the binary tree after conversion
    void inorderTraversal(BinaryTreeNode* root)
    {
        if (root)
        {
            inorderTraversal( root->left );
            cout << root->data << " ";
            inorderTraversal( root->right );
        }
    }

    // Driver program to test above functions
    int main()
    {
        // create a linked list shown in above diagram
        struct ListNode* head = NULL;
        push(&head, 36);  /* Last node of Linked List */
        push(&head, 30);
        push(&head, 25);
        push(&head, 15);
        push(&head, 12);
        push(&head, 10); /* First node of Linked List */
        BinaryTreeNode *root;
        convertList2Binary(head, root);
        cout << "Inorder Traversal of the constructed Binary Tree is: \n";
        inorderTraversal(root);
        return 0;
    }
```

Run on IDE

Output:

Inorder Traversal of the constructed Binary Tree is: 25 12 30 10 36 15.

Q9. How to

Ans :

The inser
could be a bran
are usually in t

Let us st

The inse

Search f
is to be i
insertion

Link a n

Example

1. sta
2. 60
3. 60
4. 60

Q10. Write a

Ans :

C++ progra

#include <std

// tree node

struct Node

{

int data;

Node *l

9. How to insert a node in binary tree? Explain with an example.

Ans :

The insert() operation inserts a new node at any position in a binary tree. The node to be inserted could be a branch node or a leaf node. The branch node insertion is generally based on some criteria that are usually in the context of a special form of a binary tree.

Let us study a commonly used case of inserting a node as a leaf node.

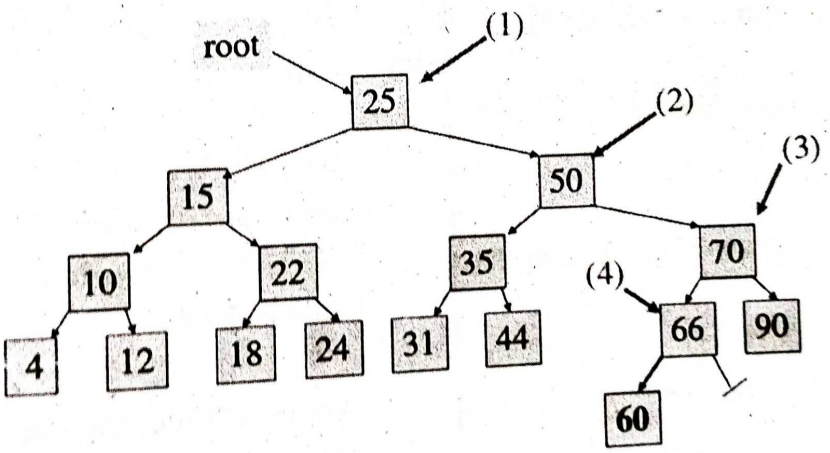
The insertion procedure is a two-step process.

Search for the node whose child node is to be inserted. This is a node at some level *i*, and a node is to be inserted at the level $i > 1$ as either its left child or right child. This is the node after which the insertion is to be made.

Link a new node to the node that becomes its parent node, that is, either the Lchild or the Rchild.

Example: insert 60 in the tree:

1. start at the root, 60 is greater than 25, search in right subtree
2. 60 is greater than 50, search in 50's right subtree
3. 60 is less than 70, search in 70's left subtree
4. 60 is less than 66, add 60 as 66's left child



10. Write a C++ program to insert an element into binary tree.

Ans :

C++ program to insert an element into binary tree

```
#include <stdio.h>
// tree node
struct Node

    int data;
    Node *left, *right;
```

```
    // returns a new tree Node
    Node* newNode(int data)
    {
        Node* temp = new Node();
        temp->data = data;
        temp->left = temp->right = NULL;
        return temp;
    }
    // A function to create binary tree.
    Node* Tree(Node* temp, int data )
    {
    // If the tree is empty, return a new node
      if (temp == NULL)
        return newNode(data);

    // Otherwise, recur down the tree
      if (data < temp->data)
        temp->left = Tree(temp->left, data);
      else
        temp->right = Tree(temp->right, data);

    //return the (unchanged) node pointer
      return temp;
    }
    //function to display all the element present in the
    binary tree
    void display(struct Node* root)
    {

      if (!root)
      return;
        display(root->left);
        cout<<root->data<<" ";
        display(root->right);

    }
    //function to insert element in binary tree
    void insert(struct Node* root , int value)
    {
      queue<struct Node*> q;
      q.push(root);
    // Do level order traversal until we find an em
    place.
      while (!q.empty()) {
        struct Node* root = q.front();
        q.pop();

        if (!root->left) {
          root->left = newNode(value);
          break;
        } else
          q.push(root->left);

        if (!root->right) {
          root->right = newNode(value);
          break;
        } else
          q.push(root->right);
      }
    }

    int main()
    {
        char ch;
        int n, arr[20],size;
        Node *root = new Node;
        root = NULL;

        cout<<"Enter the size of array : ";
        cin>>size;

        cout<<"Enter the elements in array : ";
        for(int i=0;i<size;i++)
      {
        cin>>arr[i];
      }
        // Construct the binary tree.
        for(int i = 0; i < size; i++)
        {
            root = Tree(root, arr[i]);
        }
        cout<<"\nEnter the Element to be insert
```



```
cin>>n;
insert(root
cout<<"
cout<<"
cout<<"E
display(ro
cout<<e
return 0;
}
```

4.1.3 Binary

Q11. Explai
techniq

Ans :

When w
need to follow
that binary tr
tree displayin
traversal meth

Displaying
binary tree

There a

1. In - Or
2. Pre -
3. Post -

Consid

1.

In -
right

In In
between le
the left chil

```
cin>>n;
insert(root,n);
cout<<"\nElement Inserted"<<endl;
cout<<"\nAfter Inserting "<<endl;
cout<<"Elements are: ";
display(root);
cout<<endl;
return 0;
```

4.1.3 Binary Tree Traversal

Q11. Explain about various traversal techniques of binary tree.

(Imp.)

Ans :

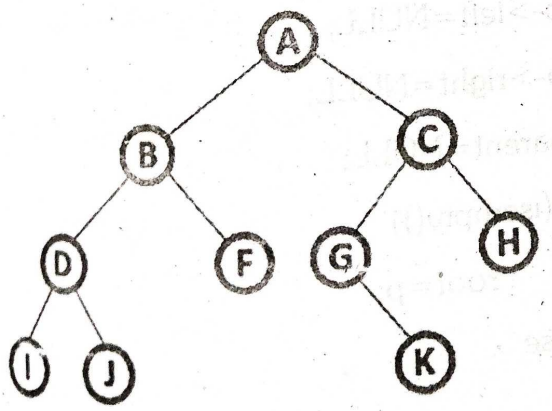
When we wanted to display a binary tree, we need to follow some order in which all the nodes of that binary tree must be displayed. In any binary tree displaying order of nodes depends on the traversal method.

Displaying (or) visiting order of nodes in a binary tree is called as Binary Tree Traversal

There are three types of binary tree traversals.

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

Consider the following binary tree...



1. In - Order Traversal (left Child - root - right Child)

In In-Order traversal, the root node is visited between left child and right child. In this traversal, the left child node is visited first, then the root node is visited and later we go for visiting right child node. This in-order traversal is applicable for every root node of all subtrees in the tree. This is performed recursively for all nodes in the tree.

In the above example of binary tree, first we try to visit left child of root node 'A', but A's left child is a root node for left subtree. so we try to visit its (B's) left child 'D' and again D is a root for subtree with nodes D, I and J. So we try to visit its left child 'I' and it is the left most child. So first we visit 'I' then go for its root node 'D' and later we visit D's right child 'J'. With this we have completed the left part of node B. Then visit 'B' and next B's right child 'F' is visited. With this we have completed left part of node A. Then visit root node 'A'. With this we have completed left and root parts of node A. Then we go for right part of the node A. In right of A again there is a subtree with root C. So go for left child of C and again it is a subtree with root G. But G does not have left part so we visit 'G' and then visit G's right child K. With this we have completed the left part of node C. Then visit root node 'C' and next visit C's right child 'H' which is the right most child in the tree so we stop the process.

That means here we have visited in the order of I - D - J - B - F - A - G - K - C - H using In-Order Traversal.

In-Order Traversal for above example of binary tree is

I - D - J - B - F - A - G - K - C - H

2. Pre - Order Traversal (root - leftChild - rightChild)

In Pre-Order traversal, the root node is visited before left child and right child nodes. In this traversal, the root node is visited first, then its left child and later its right child. This pre-order traversal is applicable for every root node of all subtrees in the tree.

In the above example of binary tree, first we visit root node 'A' then visit its left child 'B' which is a root for D and F. So we visit B's left child 'D' and again D is a root for I and J. So we visit D's left child 'I' which is the left most child. So next we go for visiting D's right child 'J'. With this we have completed root, left and right parts of node D and root, left parts of node B. Next visit B's right

child 'F'. With this we have completed root and left parts of node A. So we go for A's right child 'C' which is a root node for G and H. After visiting C, we go for its left child 'G' which is a root for node K. So next we visit left of G, but it does not have left child so we go for G's right child 'K'. With this we have completed node C's root and left parts. Next visit C's right child 'H' which is the right most child in the tree. So we stop the process.

That means here we have visited in the order of A-B-D-I-J-F-C-G-K-H using Pre-Order Traversal.

Pre-Order Traversal for above example binary tree is

A - B - D - I - J - F - C - G - K - H

2. Post - Order Traversal (leftChild - rightChild - root)

In Post-Order traversal, the root node is visited after left child and right child. In this traversal, left child node is visited first, then its right child and then its root node. This is recursively performed until the right most node is visited.

Here we have visited in the order of I - J - D - F - B - K - G - H - C - A using Post-Order Traversal.

Post-Order Traversal for above example binary tree is

I - J - D - F - B - K - G - H - C - A

Q12. Write a C++ program for creation and traversal of a Binary Tree.

Ans :

```
#include<iostream.h>
#include<conio.h>
#include<process.h>
struct tree_node
{
    tree_node *left;
    tree_node *right;
    int data;
};
class bst
{
    tree_node *root;
    public:
    bst()
    {
        root=NULL;
    }
    int isempty()
    {
        return(root==NULL);
    }
    void insert(int item);
    void inordertrav();
    void inorder(tree_node *);
    void postordertrav();
    void postorder(tree_node *);
    void preordertrav();
    void preorder(tree_node *);
};
void bst::insert(int item)
{
    tree_node *p=new tree_node;
    tree_node *parent;
    p->data=item;
    p->left=NULL;
    p->right=NULL;
    parent=NULL;
    if(isempty())
        root=p;
    else
    {
        tree_node *ptr;
        ptr=root;
        while(ptr!=NULL)
        {
```

```
            parent=ptr;
            if(item>ptr->data)
                ptr=ptr->right;
            else
                ptr=ptr->left;
        }
        if(item<parent->data)
            parent->left=p;
        else
            parent->right=p;
    }
}
void bst::inordertrav()
{
    inorder(root);
}
void bst::inorder(tree_node *ptr)
{
    if(ptr!=NULL)
    {
        inorder(ptr->left);
        cout<<" "<<ptr->data<<"    ";
        inorder(ptr->right);
    }
}
void bst::postordertrav()
{
    postorder(root);
}
void bst::postorder(tree_node *ptr)
{
    if(ptr!=NULL)
    {
        postorder(ptr->left);
        postorder(ptr->right);
        cout<<" "<<ptr->data<<"    ";
    }
}
void bst::preordertrav()
{
    preorder(root);
}
void bst::preorder(tree_node *ptr)
{
    if(ptr!=NULL)
    {
        cout<<" "<<ptr->data<<"    ";
        preorder(ptr->left);
        preorder(ptr->right);
    }
}
void main()
{
    bst b;
    b.insert(52);
    b.insert(25);
    b.insert(50);
    b.insert(15);
    b.insert(40);
    b.insert(45);
    b.insert(20); cout<<"inorder"<<endl;
    b.inordertrav();
    cout<<endl<<"postorder"<<endl;
    b.postordertrav();
    cout<<endl<<"preorder"<<endl;
    b.preordertrav();
    getch();
}
```

4.1.4 Binary Search Tree

Q13. What is Binary Search Tree? How to represent it. **(Imp.)**

Ans :

A binary search tree follows some order to arrange the elements. In a Binary search tree, the value of left node must be smaller than the parent

node, and the value of right node must be greater than the parent node. This rule is applied recursively to the left and right subtrees of the root.

Let's understand the concept of Binary search tree with an example.

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

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

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

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

Advantages of Binary Search Tree

1. Searching an element in the Binary search tree is easy as we always have a hint that which subtree has the desired element.
2. As compared to array and linked lists, insertion and deletion operations are faster in BST.

Q14. Create a binary tree by using a following example.

Ans :

Example of Creating a Binary Search Tree

Now, let's see the creation of binary search tree using an example.

Suppose the data elements are - 45, 15, 79, 90, 10, 55, 12, 20, 50

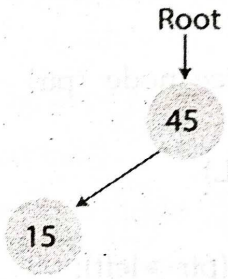
1. First, we have to insert 45 into the tree as the root of the tree.
2. Then, read the next element; if it is smaller than the root node, insert it as the root of the left subtree, and move to the next element.
3. Otherwise, if the element is larger than the root node, then insert it as the root of the right subtree.

Now, let's see the process of creating the Binary search tree using the given data element. The process of creating the BST is shown below.

Step 1 - Insert 45

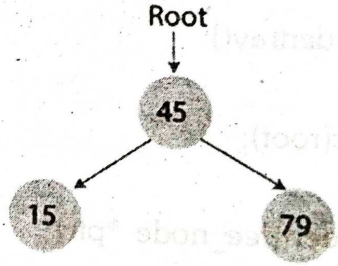
Step 2 - Insert 15.

As 15 is smaller than 45, so insert it as the root node of the left subtree.



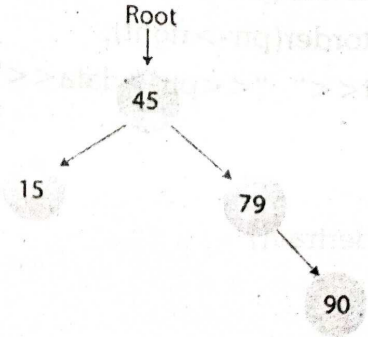
Step 3 - Insert 79.

As 79 is greater than 45, so insert it as the root node of the right subtree.



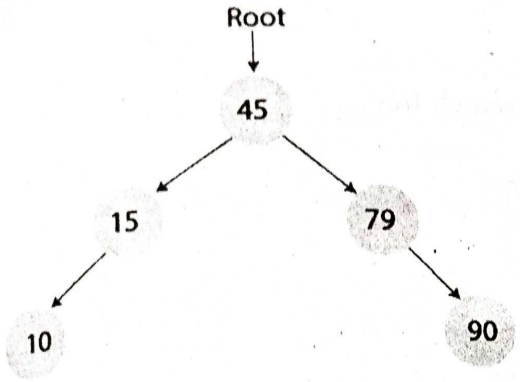
Step 4 - Insert 90.

90 is greater than 45 and 79, so it will be inserted as the right subtree of 79.



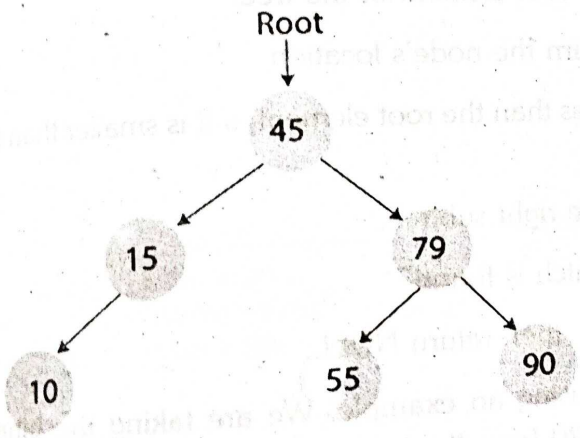
Step 5 - Insert 10.

10 is smaller than 45 and 15, so it will be inserted as a left subtree of 15.



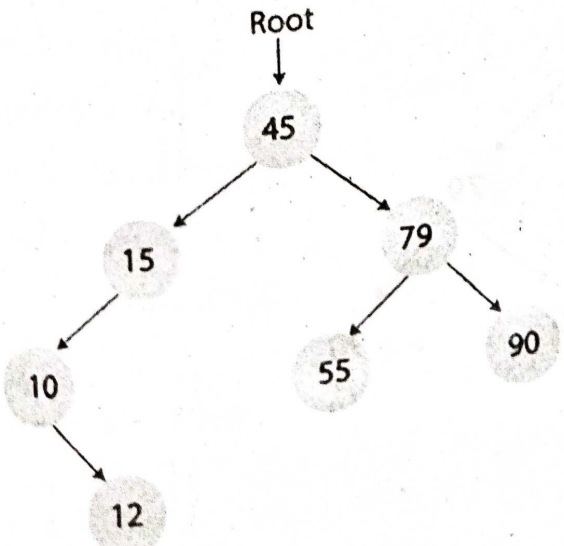
Step 6 - Insert 55.

55 is larger than 45 and smaller than 79, so it will be inserted as the left subtree of 79.



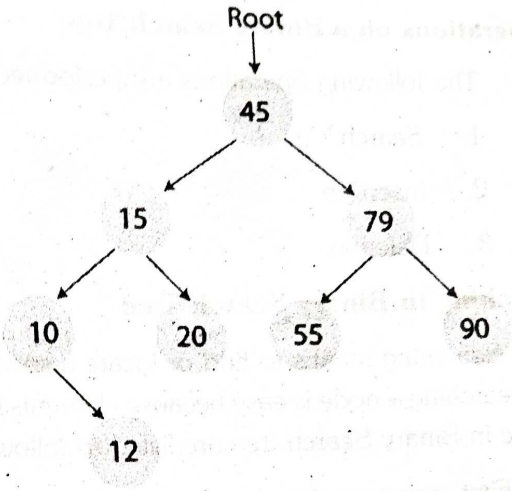
Step 7 - Insert 12.

12 is smaller than 45 and 15 but greater than 10, so it will be inserted as the right subtree of 10.



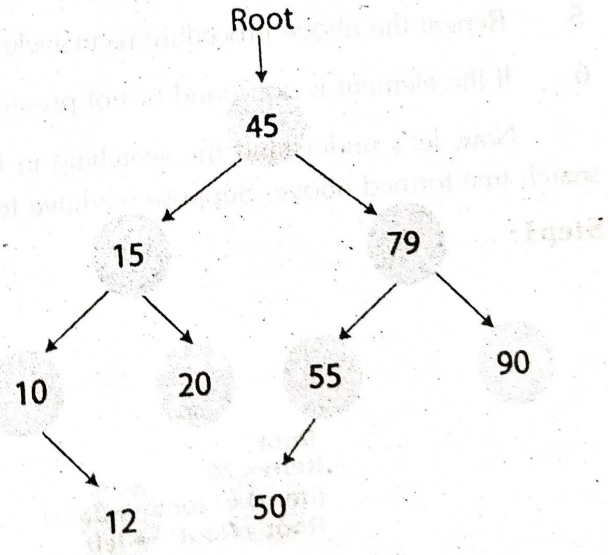
Step 8 - Insert 20.

20 is smaller than 45 but greater than 15, so it will be inserted as the right subtree of 15.



Step 9 - Insert 50.

50 is greater than 45 but smaller than 79 and 55. So, it will be inserted as a left subtree of 55.



Now, the creation of binary search tree is completed. After that, let's move towards the operations that can be performed on Binary search tree.

We can perform insert, delete and search operations on the binary search tree.

4.1.5 Operations on Binary Search Tree

Q15. Explain the operations that can be performed on BSTs. **(Imp.)**

Ans :

Operations on a Binary Search Tree

The following operations are performed on a binary search tree...

1. Search
2. Insertion
3. Deletion

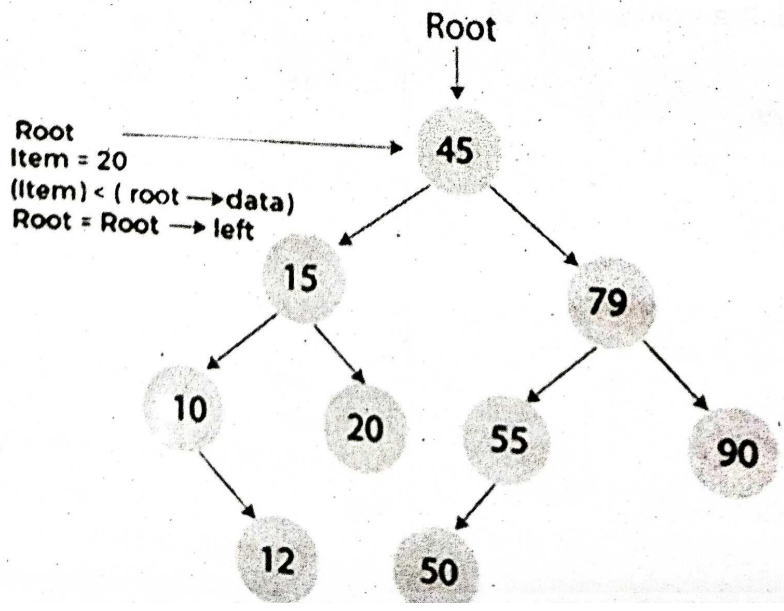
Searching in Binary Search Tree

Searching means to find or locate a specific element or node in a data structure. In Binary search tree, searching a node is easy because elements in BST are stored in a specific order. The steps of searching a node in Binary Search tree are listed as follows:

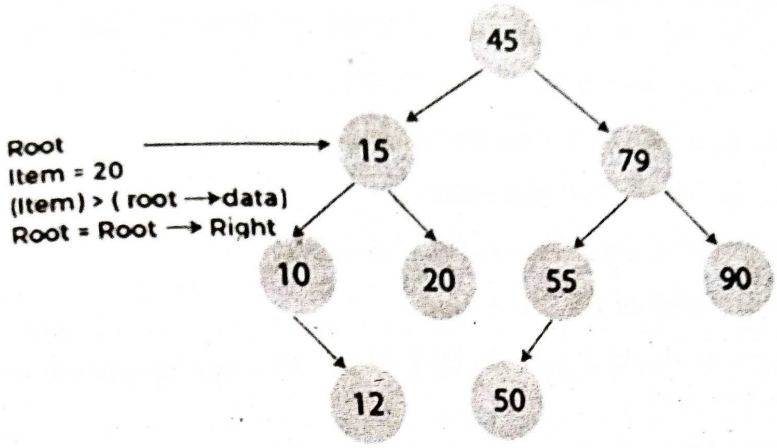
1. First, compare the element to be searched with the root element of the tree.
2. If root is matched with the target element, then return the node's location.
3. If it is not matched, then check whether the item is less than the root element, if it is smaller than the root element, then move to the left subtree.
4. If it is larger than the root element, then move to the right subtree.
5. Repeat the above procedure recursively until the match is found.
6. If the element is not found or not present in the tree, then return NULL.

Now, let's understand the searching in binary tree using an example. We are taking the binary search tree formed above. Suppose we have to find node 20 from the below tree.

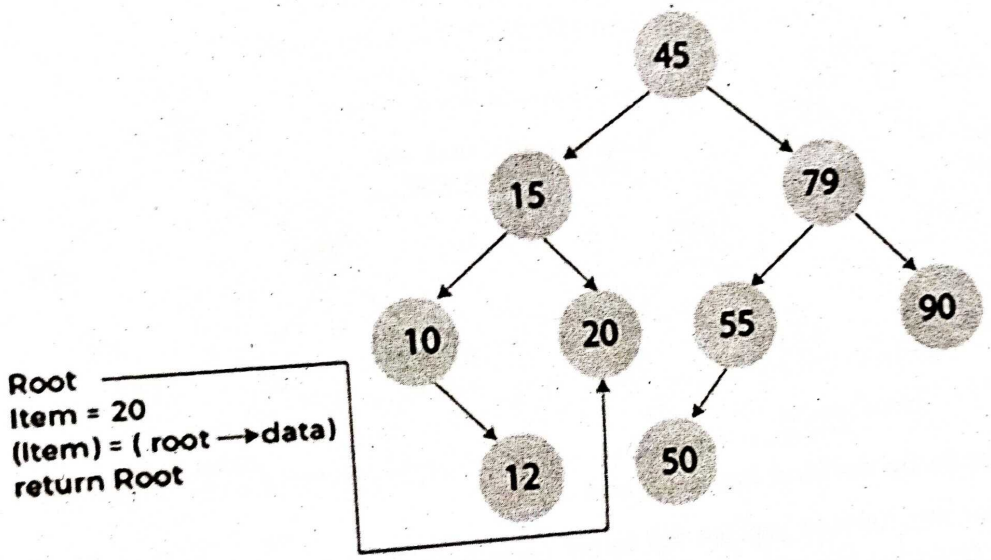
Step1:



Step 2:



Step 3:



Algorithm to search an element in Binary search tree

Search (root, item)

Step 1 - if (item = root → data) or (root = NULL)

return root

else if (item **< root** → data)

return Search(root → left, item)

else

return Search(root → right, item)

END if

Step 2 - END

Now let's understand how the deletion is performed on a binary search tree. We will also see an example to delete an element from the given tree.

Deletion in Binary Search Tree

In a binary search tree, we must delete a node from the tree by keeping in mind that the property of BST is not violated. To delete a node from BST, there are three possible situations occur -

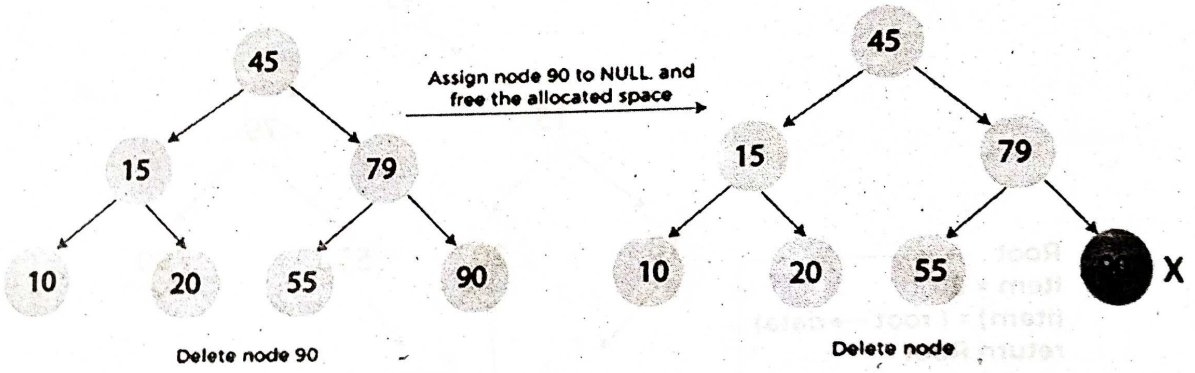
- The node to be deleted is the leaf node, or,
- The node to be deleted has only one child, and,
- The node to be deleted has two children

We will understand the situations listed above in detail.

When the node to be Deleted is the Leaf Node

It is the simplest case to delete a node in BST. Here, we have to replace the leaf node with NULL and simply free the allocated space.

We can see the process to delete a leaf node from BST in the below image. In below image, suppose we have to delete node 90, as the node to be deleted is a leaf node, so it will be replaced with NULL, and the allocated space will free.

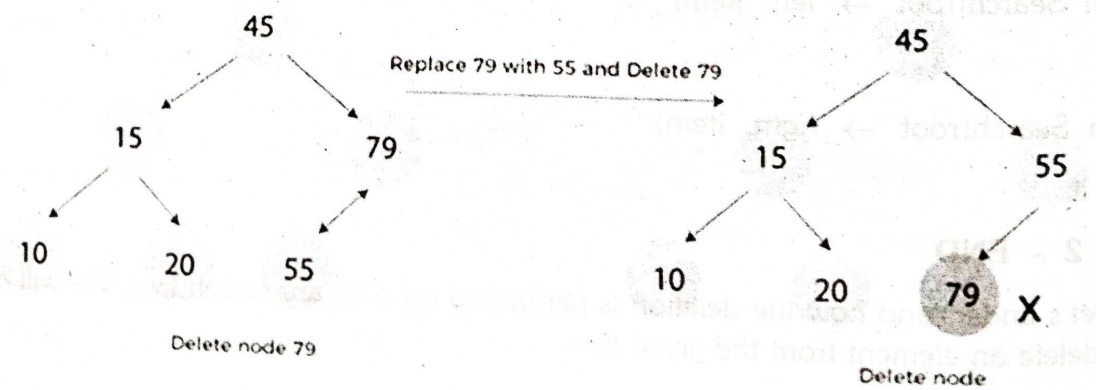


When the node to be deleted has only one child

In this case, we have to replace the target node with its child, and then delete the child node. It means that after replacing the target node with its child node, the child node will now contain the value to be deleted. So, we simply have to replace the child node with NULL and free up the allocated space.

We can see the process of deleting a node with one child from BST in the below image. In the below image, suppose we have to delete the node 79, as the node to be deleted has only one child, so it will be replaced with its child 55.

So, the replaced node 79 will now be a leaf node that can be easily deleted.



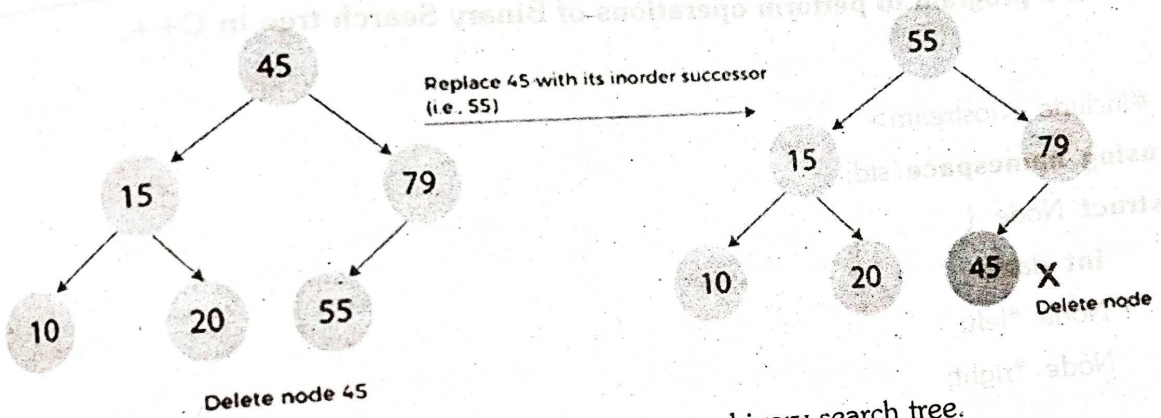
When the Node to be Deleted has two Children

This case of deleting a node in BST is a bit complex among other two cases. In such a case, the steps to be followed are listed as follows:

1. First, find the inorder successor of the node to be deleted.
2. After that, replace that node with the inorder successor until the target node is placed at the leaf of tree.
3. And at last, replace the node with NULL and free up the allocated space.

The inorder successor is required when the right child of the node is not empty. We can obtain the inorder successor by finding the minimum element in the right child of the node.

We can see the process of deleting a node with two children from BST in the below image. In the below image, suppose we have to delete node 45 that is the root node, as the node to be deleted has two children, so it will be replaced with its inorder successor. Now, node 45 will be at the leaf of the tree so that it can be deleted easily.

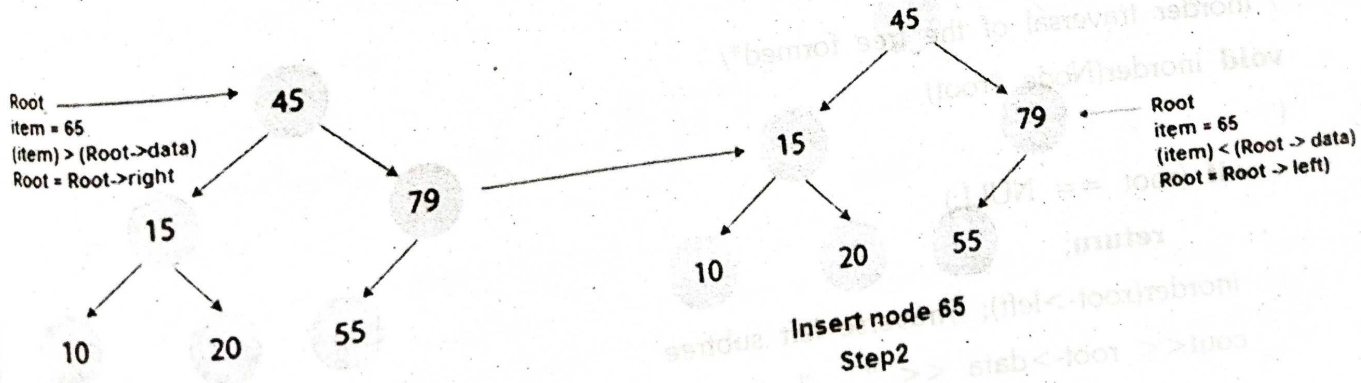


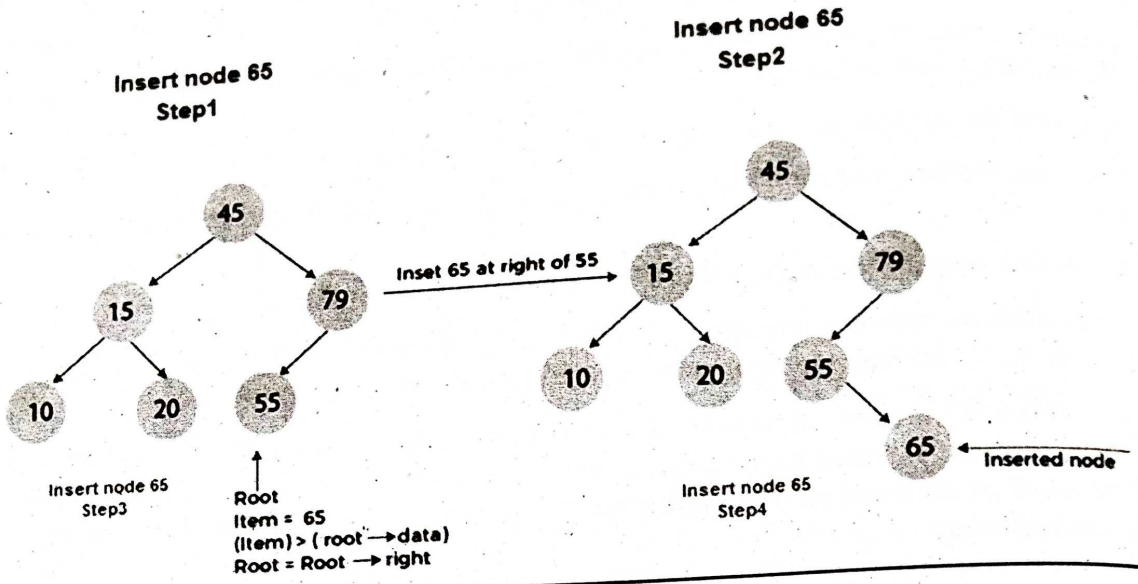
Now let's understand how insertion is performed on a binary search tree.

Insertion in Binary Search Tree

A new key in BST is always inserted at the leaf. To insert an element in BST, we have to start searching from the root node; if the node to be inserted is less than the root node, then search for an empty location in the left subtree. Else, search for the empty location in the right subtree and insert the data. Insert in BST is similar to searching, as we always have to maintain the rule that the left subtree is smaller than the root, and right subtree is larger than the root.

Now, let's see the process of inserting a node into BST using an example.





Q16. Write a program to perform operations of Binary Search tree in C++. (Imp.)

Ans :

```
#include <iostream>
using namespace std;
struct Node {
    int data;
    Node *left;
    Node *right;
};
Node* create(int item)
{
    Node* node = new Node;
    node->data = item;
    node->left = node->right = NULL;
    return node;
}
/*Inorder traversal of the tree formed*/
void inorder(Node *root)
{
    if (root == NULL)
        return;
    inorder(root->left); //traverse left subtree
    cout<< root->data << "    "; //traverse root node
```

```
inorder(root->right); //traverse right subtree
}
Node* findMinimum(Node* cur) /* To find
the inorder successor*/
{
    while(cur->left != NULL) {
        cur = cur->left;
    }
    return cur;
}
Node* insertion(Node* root, int item) /
    *Insert a node*/
{
    if (root == NULL)
    return create(item); /*return new
    node if tree is empty*/
    if (item < root->data)
        root->left = insertion(root-
            >left, item);
    else
        root->right = insertion(root-
            >right, item);
    return root;
}
void search (Node * & cur, int item, Node
* & parent)
{
while (cur != NULL && cur->data
        != item)
    {
        parent = cur;
        if (item < cur->data)
            cur = cur->left;
        else
        cur = cur->right;
    }
}
void deletion(Node*& root, int item) /
    *function to delete a node*/
{
    Node* parent = NULL;
    Node* cur = root;
    search(cur, item, parent); /*find the
    node to be deleted*/
    if (cur == NULL)
        return;
    if (cur->left == NULL && cur-
    >right == NULL) /*When node has
    no children*/
    {
        if (cur != root)
        {
          if (parent->left == cur)
          parent->left = NULL;
          else
          parent->right = NULL;
          }
          else
              root = NULL;
          free(cur);
    }
    else if (cur->left && cur->right)
    {
    Node* succ = findMinimum(cur->
        right);
        int val = succ->data;
        deletion (root, succ->data);
        cur->data = val;
    }
    else
    {
    Node* child = (cur->left)? cur-
                >left: cur->right;
```

```
            if (cur != root)
            {
            if (cur == parent->left)
            parent->left = child;
            else
            parent->right = child;
            }
            else
                root = child;
            free(cur);
        }
    }
    int main()
    {
      Node* root = NULL;
      root = insertion(root, 45);
      root = insertion(root, 30);
      root = insertion(root, 50);
      root = insertion(root, 25);
      root = insertion(root, 35);
      root = insertion(root, 45);
      root = insertion(root, 60);
      root = insertion(root, 4);
    printf("The inorder traversal of the given binary tree is - \n");
      inorder(root);
      deletion(root, 25);
      printf("\nAfter deleting node 25, the inorder traversal of the given binary tree is - \n");
      inorder(root);
      insertion(root, 2);
    printf("\nAfter inserting node 2, the inorder traversal of the given binary tree is - \n");
      inorder(root);
      return 0;
    }
```

4.1.6 Heap Tree

Q17. What is Heap?

Ans :

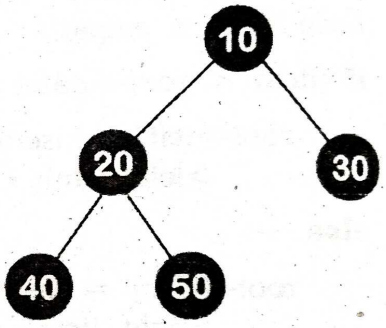
A heap is a complete binary tree, and the binary tree is a tree in which the node can have utmost two children. Before knowing more about the heap data structure, we should know about the complete binary tree.

Q18. What is a complete binary tree?

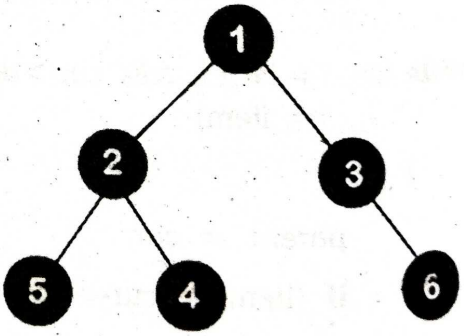
Ans :

A complete binary tree is a binary tree in which all the levels except the last level, i.e., leaf node should be completely filled, and all the nodes should be left-justified.

Let's understand through an example.



In the above figure, we can observe that all the internal nodes are completely filled except the leaf node; therefore, we can say that the above tree is a complete binary tree.



The above figure shows that all the internal nodes are completely filled except the leaf node but the leaf nodes are added at the right part; therefore, the above tree is not a complete binary tree.

Q19. What are the different types of heaps? Explain Max heap algorithm with an example.

Ans :

There are two types of the heap:

1. Min Heap
2. Max heap

1. Min Heap

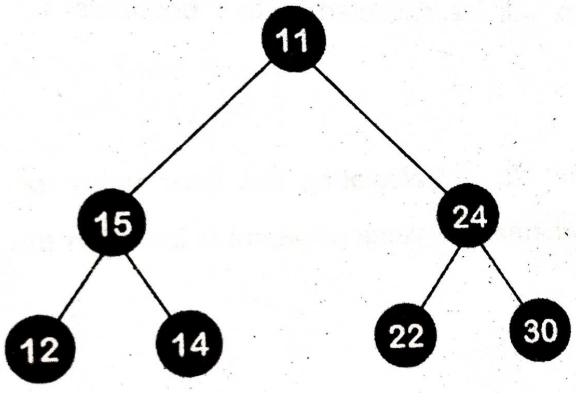
The value of the parent node should be less than or equal to either of its children.

Or

In other words, the min-heap can be defined as, for every node i, the value of node i is greater than or equal to its parent value except the root node. Mathematically, it can be defined as:

A[Parent(i)] <= A[i]

Let's understand the min-heap through an example.



In the above figure, 11 is the root node, and the value of the root node is less than the value of all the other nodes (left child or a right child).

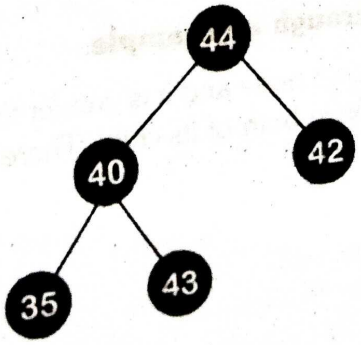
2. Max Heap

The value of the parent node is greater than or equal to its children.

Or

In other words, the max heap can be defined as for every node i; the value of node i is less than or equal to its parent value except the root node. Mathematically, it can be defined as:

A[Parent(i)] >= A[i]



The above tree is a max heap tree as it satisfies the property of the max heap. Now, let's see the array representation of the max heap.

Time complexity in Max Heap

The total number of comparisons required in the max heap is according to the height of the tree. The height of the complete binary tree is always logn; therefore, the time complexity would also be O(logn).

Algorithm of insert operation in the max heap.

```
/algorithm to insert an element in the max heap.
insert Heap(A, n, value)
{
n=n+1;          // n is incremented to insert the new element
A[n]=value;     // assign new value at the nth position
i = n;          // assign the value of n to i
                // loop will be executed until i becomes 1.
while(i>1)
{
parent= floor value of i/2; // Calculating the floor value of i/2
// Condition to check whether the value of parent is less than the given node or not
if(A[parent]<A[i])
{
swap(A[parent], A[i]);
i = parent;
}
else
{
return;
}
}
}
```

Let's understand the max heap through an example.

In the above figure, 55 is the parent node and it is greater than both of its child, and 11 is the parent of 9 and 8, so 11 is also greater than from both of its child. Therefore, we can say that the above tree is a max heap tree.

Insertion in the Heap tree

44, 33, 77, 11, 55, 88, 66

Suppose we want to create the max heap tree. To create the max heap tree, we need to consider the following two cases:

First, we have to insert the element in such a way that the property of the complete binary tree must be maintained.

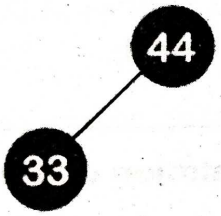
Secondly, the value of the parent node should be greater than the either of its child.

Step 1

First we add the 44 element in the tree as shown below:

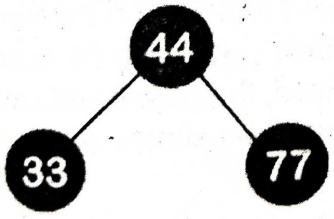
Step 2

The next element is 33. As we know that insertion in the binary tree always starts from the left side so 44 will be added at the left of 33 as shown below:

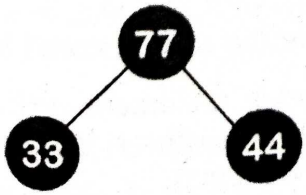


Step 3

The next element is 77 and it will be added to the right of the 44 as shown below:

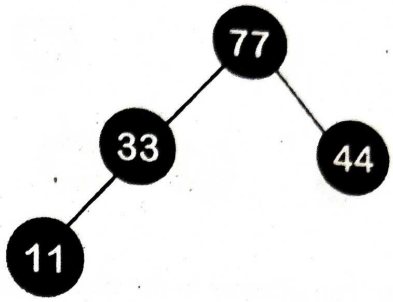


As we can observe in the above tree that it does not satisfy the max heap property, i.e., parent node 44 is less than the child 77. So, we will swap these two values as shown below:



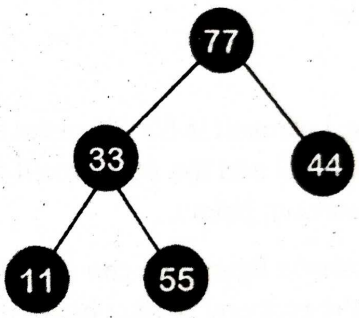
Step 4

The next element is 11. The node 11 is added to the left of 33 as shown below:

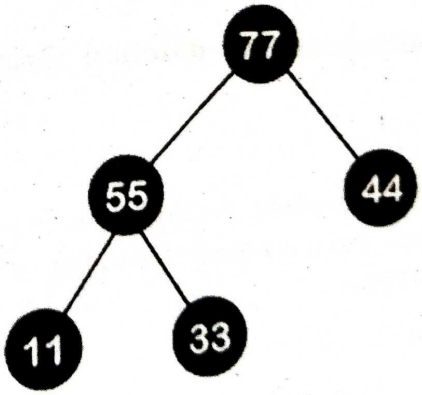


Step 5

The next element is 55. To make it a complete binary tree, we will add the node 55 to the right of 33 as shown below:

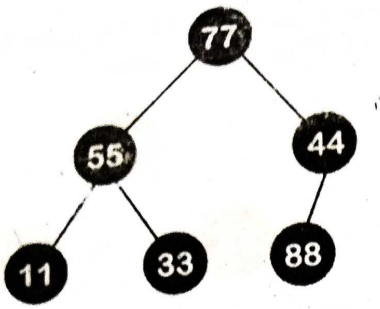


As we can observe in the above figure that it does not satisfy the property of the max heap because 33<55, so we will swap these two values as shown below:



Step 6

The next element is 88. The left subtree is completed so we will add 88 to the left of 44 as shown below:



As we can observe in the above figure that it does not satisfy the property of the max heap because 44<88, so we will swap these two values as shown below:

Again, it is violating the max heap property because 88>77 so we will swap these two values as shown below:

Step 7:

The next element is 66. To make a complete binary tree, we will add the 66 element to the right side of 77 as shown below:

In the above figure, we can observe that the tree satisfies the property of max heap; therefore, it is a heap tree.

Deletion in Heap Tree

In Deletion in the heap tree, the root node is always deleted and it is replaced with the last element.

Let's understand the deletion through an example.

Step 1:

In the above tree, the first 30 node is deleted from the tree and it is replaced with the 15 element as shown below:

Now we will heapify the tree. We will check whether the 15 is greater than either of its child or not. 15 is less than 20 so we will swap these two values as shown below:

Again, we will compare 15 with its child. Since 15 is greater than 10 so no swapping will occur.

Algorithm to Heapify the Tree

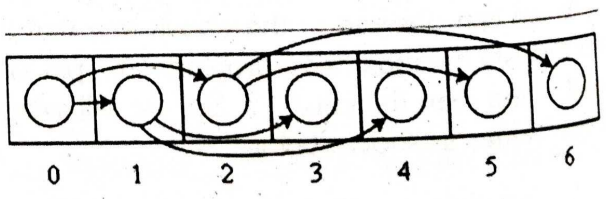
```
Max Heapify(A, n, i)
{
int largest =i;
int l= 2i;
int r= 2i+1;
while(l<=n && A[l]>A[largest])
{
largest=l;
}
while(r<=n && A[r]>A[largest])
{
largest=r;
}
if(largest!=i)
{
swap(A[largest], A[i]);
heapify(A, n, largest);
}}
```

4.5.2 Implementation of Heap

Q20. Implement and build heap using an example.

Ans :

A more common approach is to store the heap in an array. Since heap is always a complete binary tree, it can be stored compactly. No space is required for pointers; instead, the parent and children of each node can be found by simple arithmetic on array indices.



The rules (assume the root is stored in *arr*[0]):

For each index *i*, element *arr*[*i*] has children at *arr*[2*i* + 1] and *arr*[2*i* + 2], and the parent at *arr*[floor((*i* - 1)/2)].

This implementation is particularly useful in the heapsort algorithm, where it allows the space in the input array to be reused to store the heap (i.e., the algorithm is in-place). However it requires allocating the array before filling it, which makes this method not that useful in priority queues implementation, where the number of elements is unknown.

It is perfectly acceptable to use a traditional binary tree data structure to implement a binary heap. There is an issue with finding the adjacent element on the last level on the binary heap when adding an element.

Building a Heap

A heap could be built by successive insertions. This approach requires $O(n \log n)$ time for n elements. Why?

The Optimal Method

- Starts by arbitrarily putting the elements on a binary tree.
- Starting from the lowest level and moving upwards until the heap property is restored by shifting the root of the subtree downward as in the removal algorithm.
- If all the subtrees at some height h (measured from the bottom) have already been "heapified", the trees at hight $h+1$ can be heapified by sending their root down. This process takes $O(h)$ swaps.

As an example, let's build a heap with the following values: 20, 35, 23, 22, 4, 45, 21, 5, 42 and 19.

As has been proved , this optimal method requires $O(n)$ time for n elements.

Q21. Write about heap data structure.

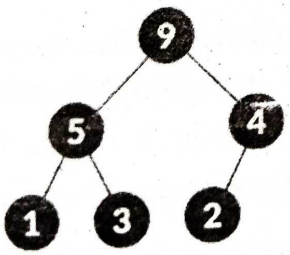
OR

Explain, what is heapify?

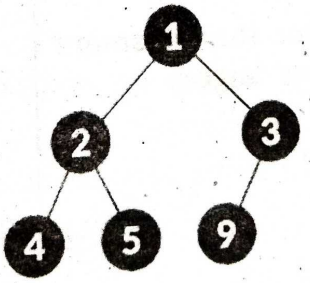
Ans :

Heap data structure is a complete binary tree that satisfies the heap property, where any given node is

- always greater than its child node/s and the key of the root node is the largest among all other nodes. This property is also called max heap property.
- always smaller than the child node/s and the key of the root node is the smallest among all other nodes. This property is also called min heap property.



Max-heap



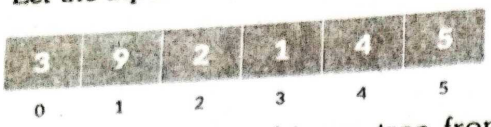
Min-heap

This type of data structure is also called a binary heap.

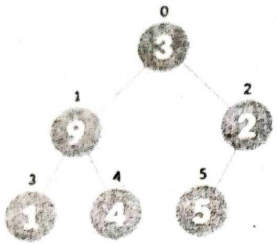
Heapify

Heapify is the process of creating a heap data structure from a binary tree. It is used to create a Min-Heap or a Max-Heap.

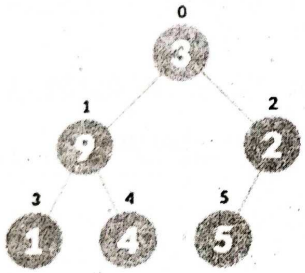
1. Let the input array be Initial Array



2. Create a complete binary tree from the arrayComplete binary tree.



3. Start from the first index of non-leaf node whose index is given by n/2 – 1.



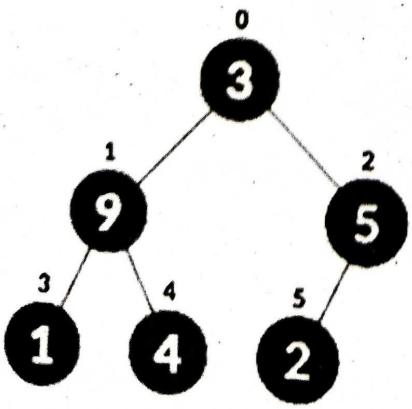
4. Start from the first on leaf node.

5. Set current element i as largest.

6. The index of left child is given by 2i + 1 and the right child is given by 2i + 2.

If leftChild is greater than current Element (i.e. element at ith index), set left ChildIndex as largest.

If right Child is greater than element in largest, set rightChildIndex as largest.



7. Swap largest with currentElementSwap if necessary.

8. Repeat steps 3-7 until the subtrees are also heapified.

Algorithm

```
Heapify(array, size, i)
    set i as largest
    leftChild = 2i + 1
    rightChild = 2i + 2
    if leftChild > array[largest]
    set leftChildIndex as largest
    if rightChild > array[largest]
    set rightChildIndex as largest
```

4.1.7 B-Tree

Q22. What is B-Tree ? Explain the operations of B-Tree.

Ans : **(Imp.)**

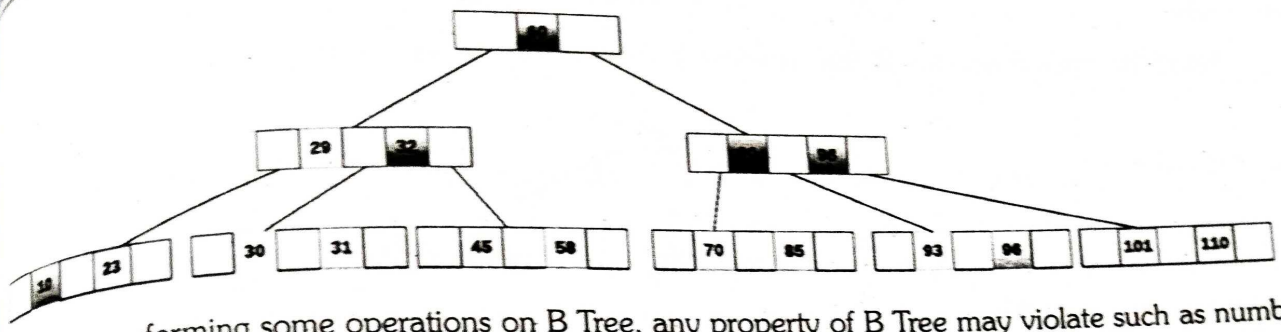
B Tree is a specialized m-way tree that can be widely used for disk access. A B-Tree of order m can have at most m-1 keys and m children. One of the main reason of using B tree is its capability to store large number of keys in a single node and large key values by keeping the height of the tree relatively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

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

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

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

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



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

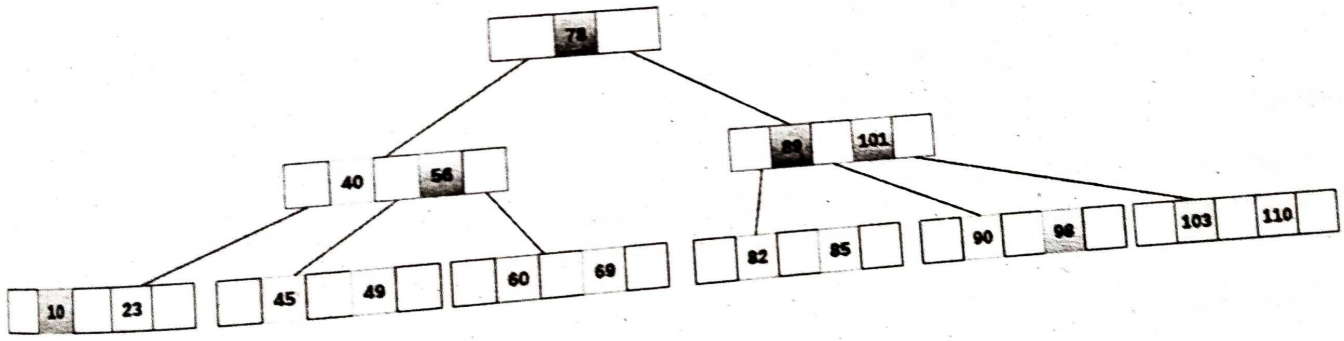
Operations

Searching :

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

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

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



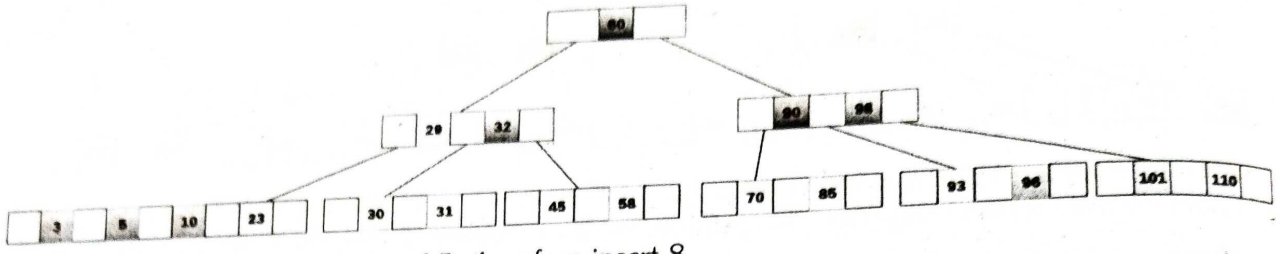
Inserting

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

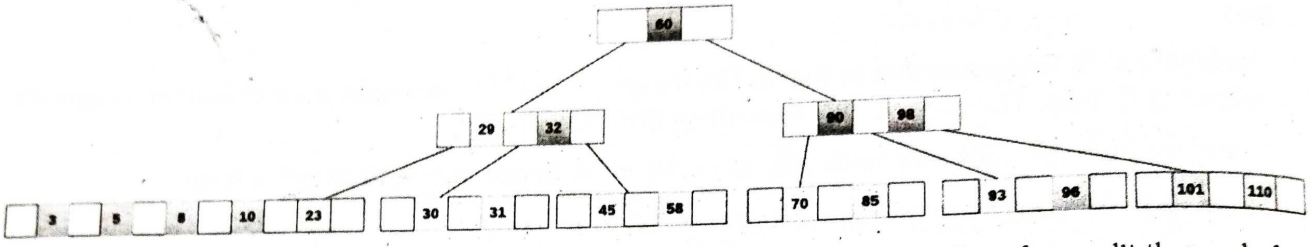
1. Traverse the B Tree in order to find the appropriate leaf node at which the node can be inserted.
2. If the leaf node contain less than m-1 keys then insert the element in the increasing order.
3. Else, if the leaf node contains m-1 keys, then follow the following steps.
   - Insert the new element in the increasing order of elements.
   - Split the node into the two nodes at the median.
   - Push the median element upto its parent node.
   - If the parent node also contain m-1 number of keys, then split it too by following the same steps.

Example:

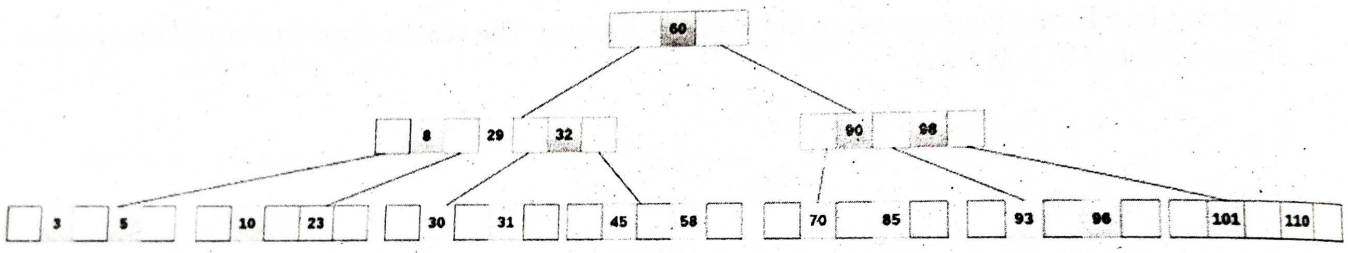
Insert the node 8 into the B Tree of order 5 shown in the following image.



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



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



Deletion

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

1. Locate the leaf node.
2. If there are more than m/2 keys in the leaf node then delete the desired key from the node.
3. If the leaf node doesn't contain m/2 keys then complete the keys by taking the element from eight or left sibling.

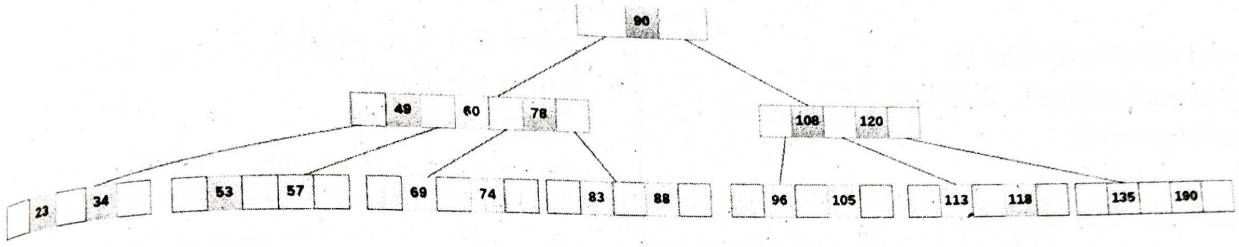
- o If the left sibling contains more than m/2 elements then push its largest element up to its parent and move the intervening element down to the node where the key is deleted.
- o If the right sibling contains more than m/2 elements then push its smallest element up to the parent and move intervening element down to the node where the key is deleted.

4. If neither of the sibling contain more than m/2 elements then create a new leaf node by joining two leaf nodes and the intervening element of the parent node.
5. If parent is left with less than m/2 nodes then, apply the above process on the parent too.

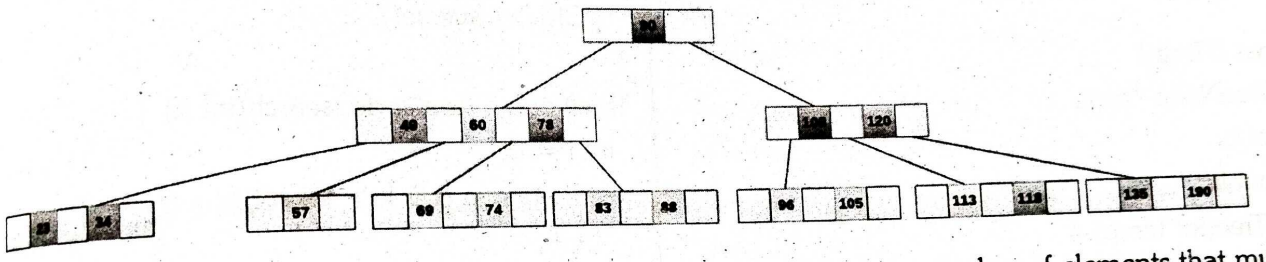
If the the node which is to be deleted is an internal node, then replace the node with its in-order successor or predecessor. Since, successor or predecessor will always be on the leaf node hence, the process will be similar as the node is being deleted from the leaf node.

Example 1

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

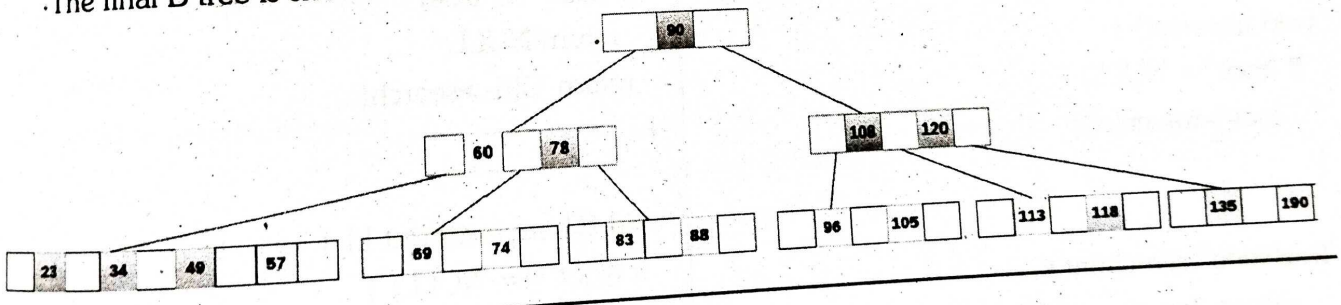


53 is present in the right child of element 49. Delete it.



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

.The final B tree is shown as follows.



Q23. Write a program to search a key in B-Tree.

Ans :

```
// Searching a key on a B-tree in C++
#include <iostream>
using namespace std;

class TreeNode {
 int *keys;
 int t;
 TreeNode **C;
```

```
  int n;
  bool leaf;

  public:
  TreeNode(int temp, bool bool_leaf);

  void insertNonFull(int k);
  void splitChild(int i, TreeNode *y);
  void traverse();

  TreeNode *search(int k);

  friend class BTree;
};
class BTree {
  TreeNode *root;
  int t;
  public:
  BTree(int temp) {
   root = NULL;
   t = temp;
  }

  void traverse() {
   if (root != NULL)
    root->traverse();
  }

  TreeNode *search(int k) {
    return (root == NULL) ? NULL : root-
>search(k);
  }
  void insert(int k);
};

TreeNode::TreeNode(int t1, bool leaf1) {
 t = t1;
 leaf = leaf1;

 keys = new int[2 * t - 1];
  C = new TreeNode *[2 * t];

  n = 0;
}
void TreeNode::traverse() {
 int i;
 for (i = 0; i < n; i++) {
  if (leaf == false)
   C[i]->traverse();
  cout << " " << keys[i];
 }

 if (leaf == false)
  C[i]->traverse();
}
TreeNode *TreeNode::search(int k) {
 int i = 0;
 while (i < n && k > keys[i])
  i++;

 if (keys[i] == k)
  return this;
 if (leaf == true)
  return NULL;
 return C[i]->search(k);
}

void BTree::insert(int k) {
 if (root == NULL) {
  root = new TreeNode(t, true);
  root->keys[0] = k;
  root->n = 1;
 } else {
  if (root->n == 2 * t - 1) {
   TreeNode *s = new TreeNode(t, false);
   s->C[0] = root;
   s->splitChild(0, root);
   int i = 0;
   if (s->keys[0] < k)
    i++;
```

```
    s->C[i]->insertNonFull(k);

    root = s;
  } else
    root->insertNonFull(k);

}
void TreeNode::insertNonFull(int k) {
 int i = n - 1;
 if (leaf == true) {
  while (i >= 0 && keys[i] > k) {
   keys[i + 1] = keys[i];
   i--;
  }
  keys[i + 1] = k;
  n = n + 1;
 } else {
  while (i >= 0 && keys[i] > k)
   i--;
  if (C[i + 1]->n == 2 * t - 1) {
   splitChild(i + 1, C[i + 1]);
   if (keys[i + 1] < k)
    i++;
  }
  C[i + 1]->insertNonFull(k);
 }
}

void TreeNode::splitChild(int i, TreeNode *y) {
 TreeNode *z = new TreeNode(y->t, y->leaf);
 z->n = t - 1;
 for (int j = 0; j < t - 1; j++)
  z->keys[j] = y->keys[j + t];

 if (y->leaf == false) {
  for (int j = 0; j < t; j++)
   z->C[j] = y->C[j + t];
 }
 y->n = t - 1;
 for (int j = n; j >= i + 1; j--)
  C[j + 1] = C[j];
 C[i + 1] = z;
 for (int j = n - 1; j >= i; j--)
  keys[j + 1] = keys[j];

 keys[i] = y->keys[t - 1];
 n = n + 1;
}

int main() {
 BTree t(3);
 t.insert(8);
 t.insert(9);
 t.insert(10);
 t.insert(11);
 t.insert(15);
 t.insert(16);
 t.insert(17);
 t.insert(18);
 t.insert(20);
 t.insert(23);

 cout << "The B-tree is: ";
 t.traverse();
 int k = 10;
 (t.search(k) != NULL) ? cout << endl
<< k << " is found"
          : cout << endl
<< k << " is not Found";
 k = 2;
 (t.search(k) != NULL) ? cout << endl
<< k << " is found"
          : cout << endl
<< k << " is not Found\n";
}
```

4.2 GRAPHS

4.2.1 Terminology

Q24. What is graph? Discuss the terminology of graph. **(Imp.)**

Ans :

Graph is a non linear data structure, it contains a set of points known as nodes (or vertices) and set of links known as edges (or Arcs) which connects the vertices. A graph is defined as follows...

- Graph is a collection of vertices and arcs which connects vertices in the graph
- Graph is a collection of nodes and edges which connects nodes in the graph

Generally, a graph G is represented as G = (V , E), where V is set of vertices and E is set of edges.

Example

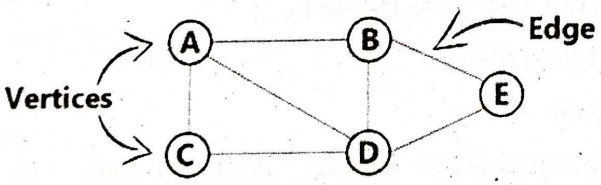
The following is a graph with 5 vertices and 6 edges.

This graph G can be defined as G = (V , E)

Where

V = {A,B,C,D,E} and

E = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}.



We use the following terms in graph data structure...

Vertex

A individual data element of a graph is called as Vertex. Vertex is also known as node. In above example graph, A, B, C, D & E are known as vertices.

Edge

An edge is a connecting link between two vertices. Edge is also known as Arc. An edge is represented as (startingVertex, endingVertex). For example, in above graph, the link between vertices A and B is represented as (A,B). In above example graph, there are 7 edges (i.e., (A,B), (A,C), (A,D), (B,D), (B,E), (C,D), (D,E)).

Edges are three types.

1. **Undirected Edge :** An undirected egde is a bidirectional edge. If there is a undirected edge between vertices A and B then edge (A , B) is equal to edge (B , A).
2. **Directed Edge:** A directed egde is a unidirectional edge. If there is a directed edge between vertices A and B then edge (A , B) is not equal to edge (B , A).
3. **Weighted Edge:** A weighted egde is an edge with cost on it.

Undirected Graph

A graph with only undirected edges is said to be undirected graph.

Directed Graph

A graph with only directed edges is said to be directed graph.

Mixed Graph

A graph with undirected and directed edges is said to be mixed graph.

End vertices or Endpoints

The two vertices joined by an edge are called the end vertices (or endpoints) of the edge.

Origin

If an edge is directed, its first endpoint is said to be origin of it.

Destination

If an edge is directed, its first endpoint is said to be origin of it and the other endpoint is said to be the destination of the edge.

Adjacent

If there is an edge between vertices A and B then both A and B are said to be adjacent. In other words, Two vertices A and B are said to be adjacent if there is an edge whose end vertices are A and B.

Incident

An edge is said to be incident on a vertex if the vertex is one of the endpoints of that edge.

Outgoing Edge

A directed edge is said to be outgoing edge on its orign vertex.

Incoming Edge

A directed edge is said to be incoming edge on its destination vertex.

Degree

Total number of edges connected to a vertex is said to be degree of that vertex.

Indegree

Total number of incoming edges connected to a vertex is said to be indegree of that vertex.

Outdegree

Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.

Parallel edges or Multiple edges

If there are two undirected edges to have the same end vertices, and for two directed edges to have the same origin and the same destination. Such edges are called parallel edges or multiple edges.

Self-loop

An edge (undirected or directed) is a self-loop if its two endpoints coincide.

Simple Graph

A graph is said to be simple if there are no parallel and self-loop edges.

Path

A path is a sequence of alternating vertices and edges that starts at a vertex and ends at a vertex such that each edge is incident to its predecessor and successor vertex.

4.2.2 Types

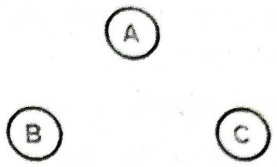
Q25. Write about various types of graphs. **(Imp.)**

Ans :

So depending upon the position of these nodes and vertices, there are different types of graphs, such as:

1. Null Graph

The Null Graph is also known as the order zero graph. The term "null graph" refers to a graph with an empty edge set. In other words, a null graph has no edges, and the null graph is present with only isolated vertices in the graph.



The image displayed above is a null or zero graphs because it has zero edges between the three vertices of the graph.

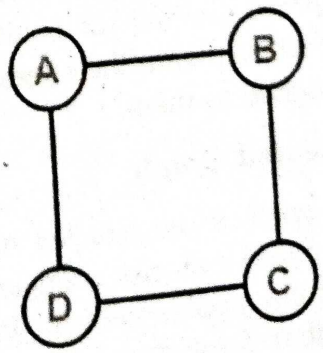
2. Trivial Graph

A graph is called a trivial graph if it has only one vertex present in it. The trivial graph is the smallest possible graph that can be created with the least number of vertices that is one vertex only.

The above is an example of a trivial graph having only a single vertex in the whole graph named vertices A.

3. Non-Directed Graph

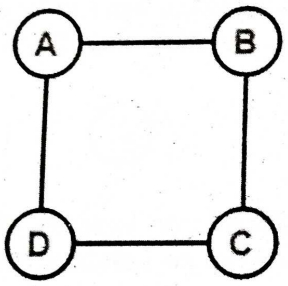
A graph is called a non-directed graph if all the edges present between any graph nodes are non-directed. By non-directed edges, we mean the edges of the graph that cannot be determined from the node it is starting and at which node it is ending. All the edges for a graph need to be non-directed to call it a non-directed graph. All the edges of a non-directed graph don't have any direction.



The graph that is displayed above is an example of a disconnected graph. This graph is called a disconnected graph because there are four vertices named vertex A, vertex B, vertex C, and vertex D. There are also exactly four edges between these vertices of the graph. And all the vertices that are present between the different nodes of the graph are not directed, which means the edges don't have any specific direction.

4. Directed Graph

Another name for the directed graphs is digraphs. A graph is called a directed graph or digraph if all the edges present between any vertices or nodes of the graph are directed or have a defined direction. By directed edges, we mean the edges of the graph that have a direction to determine from which node it is starting and at which node it is ending.

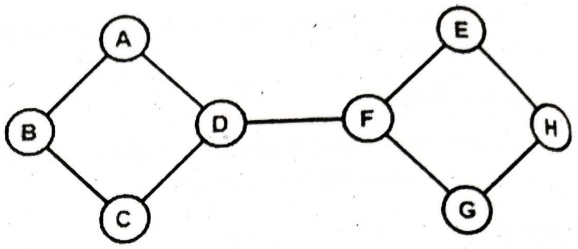


All the edges for a graph need to be directed to call it a directed graph or digraph. All the edges of a directed graph or digraph have a direction that will start from one vertex and end at another.

The graph that is displayed above is an example of a connected graph. This graph is called a connected graph because there are four vertices in the graph named vertex A, vertex B, vertex C, and vertex D. There are also exactly four edges between these vertices of the graph and all the vertices that are present between the different nodes of the graph are directed (or pointing to some of the vertices) which means the edges have a specific direction assigned to them.

5. Connected Graph

For a graph to be labelled as a connected graph, there must be at least a single path between every pair of the graph's vertices. In other words, we can say that if we start from one vertex, we should be able to move to any of the vertices that are present in that particular graph, which means there exists at least one path between all the vertices of the graph.



The graph shown above is an example of a connected graph because we start from any one of the vertices of the graph and start moving towards any other remaining vertices of the graph. There will exist at least one path for traversing the graph.

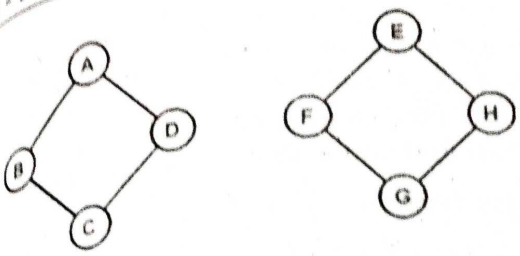
For example, if we begin from vertex B and traverse to vertex H, there are various paths for traversing. One of the paths is

Vertice B -> vertice C -> vertice D -> vertice F -> vertice E -> vertice H.

Similarly, there are other paths for traversing the graph from vertex B to vertex H. there is at least one path between all the graph nodes. In other words, we can say that all the vertices or nodes of the graph are connected to each other via edge or number of edges.

Disconnected Graph

A graph is said to be a disconnected graph where there does not exist any path between at least one pair of vertices. In other words, we can say that if we start from any one of the vertices of the graph and try to move to the remaining present vertices of the graph and there exists not even a single path to move to that vertex, then it is the case of the disconnected graph. If any one of such a pair of vertices doesn't have a path between them, it is called a disconnected graph.

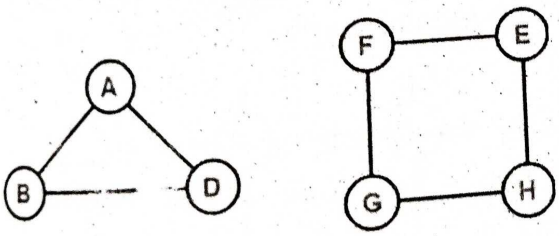


The graph shown above is a disconnected graph. The above graph is called a disconnected graph because at least one pair of vertices doesn't have a path to traverse starting from either node.

For example, a single path between both vertices doesn't exist if we want to traverse from vertex A to vertex G. In other words, we can say that all the vertices or nodes of the graph are not connected to each other via edge or number of edges so that they can be traversed.

Regular Graph

For a graph to be called a regular, it should satisfy one primary condition: all graph vertices should have the same degree. By the degree of vertices, we mean the number of nodes associated with a particular vertex. If all the graph nodes have the same degree value, then the graph is called a regular graph. If all the vertices of a graph have the degree value of 6, then the graph is called a 6-regular graph. If all the vertices in a graph are of degree 'k', then it is called a "k-regular graph".



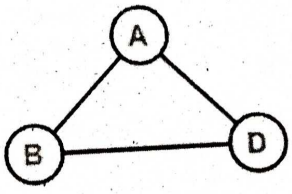
The graphs that are displayed above are regular graphs. In graph 1, there are three vertices named vertex A, vertex B, and vertex C, All the vertices in graph 1, have the degree of each node as 2. The degree of each vertex is calculated by counting the number of edges connected to that particular vertex.

For vertex A in graph 1, there are two edges associated with vertex A, one from vertex B and another from vertex D. Thus, the degree of vertex A of graph one is 2. Similarly, for other vertices of the graph, there are only two edges associated with each vertex, vertex B and vertex D. Therefore, vertex B and vertex D are 2. As the degree of all the three nodes of the graph is the same, that is 2. Therefore, this graph is called a 2-regular graph.

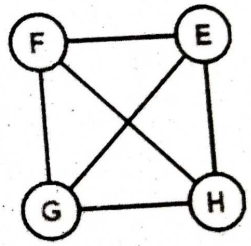
Similarly, for the second graph shown above, there are four vertices named vertex E, vertex F, vertex G, and vertex F. The degree of all the four vertices of this graph is 2. Each vertex of the graph is 2 because only two edges are associated with all of the graph's vertices. As all the nodes of this graph have the same degree of 2, this graph is called a regular graph.

Complete Graph

A graph is said to be a complete graph if, for all the vertices of the graph, there exists an edge between every pair of the vertices. In other words, we can say that all the vertices are connected to the rest of all the vertices of the graph. A complete graph of 'n' vertices contains exactly nC2 edges, and a complete graph of 'n' vertices is represented as Kn.



K_3

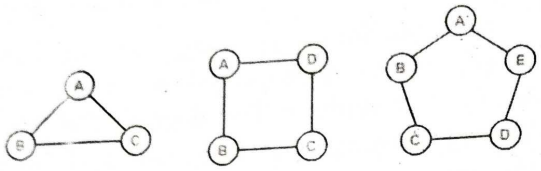


K_4

There are two graphs name K3 and K4 shown in the above image, and both graphs are complete graphs. Graph K3 has three vertices, and each vertex has at least one edge with the rest of the vertices. Similarly, for graph K4, there are four nodes named vertex E, vertex F, vertex G, and vertex H. For example, the vertex F has three edges connected to it to connect it to the respective three remaining vertices of the graph. Likewise, for the other three reaming vertices, there are three edges associated with each one of them. As all the vertices of this graph have a separate edge for other vertices, it is called a complete graph.

Cycle Graph

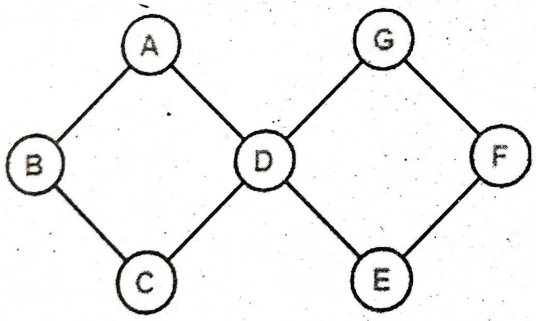
If a graph with many vertices greater than three and edges form a cycle, then the graph is called a cycle graph. In a graph of cycle type, the degree of all the vertices of the cycle graph will be 2.



There are three graphs shown in the above image, and all of them are examples of the cyclic graph because the number of nodes for all of these graphs is greater than two and the degree of all the vertices of all these graphs is exactly 2.

Cyclic Graph

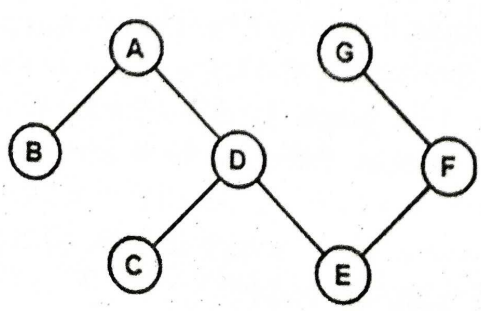
For a graph to be called a cyclic graph, it should consist of at least one cycle. If a graph has a minimum of one cycle present, it is called a cyclic graph.



The graph shown in the image has two cycles present, satisfying the required condition for a graph to be cyclic, thus making it a cyclic graph.

Acyclic Graph

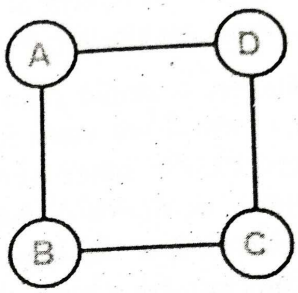
A graph is called an acyclic graph if zero cycles are present, and an acyclic graph is the complete opposite of a cyclic graph.



The graph shown in the above image is acyclic because it has zero cycles present in it. That means if we begin traversing the graph from vertex B, then a single path doesn't exist that will traverse all the vertices and end at the same vertex that is vertex B.

Finite Graph

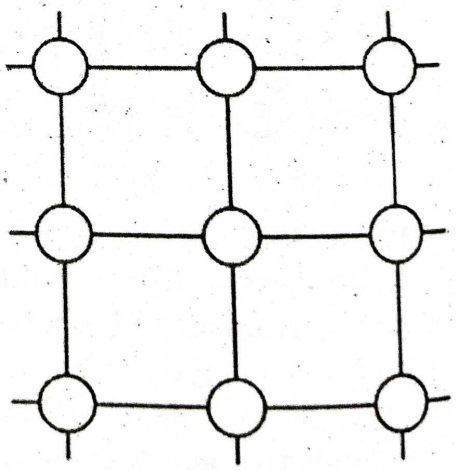
If the number of vertices and the number of edges that are present in a graph are finite in number, then that graph is called a finite graph.



The graph shown in the above image is the finite graph. There are four vertices named vertex A, vertex B, vertex C, and vertex D, and the number of edges present in this graph is also four, as both the number of nodes and vertices of this graph is finite in number it is called a finite graph.

Infinite Graph

If the number of vertices in the graph and the number of edges in the graph are infinite in number, that means the vertices and the edges of the graph cannot be counted, then that graph is called an infinite graph.

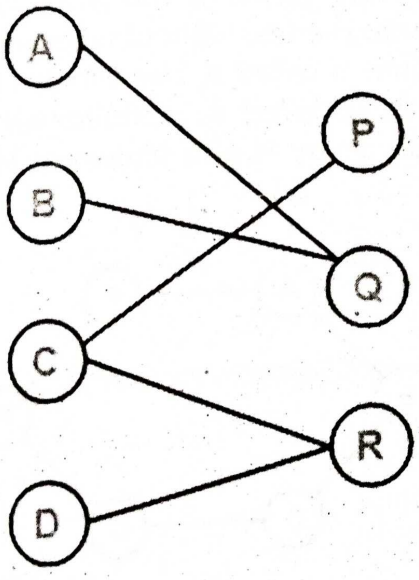


As we can see in the above image, the number of vertices in the graph and the number of edges in the graph are infinite, so this graph is called an infinite graph.

Bipartite Graph

For a graph to be a Bipartite graph, it needs to satisfy some of the basic preconditions. These conditions are:

- All the vertices of the graph should be divided into two distinct sets of vertices X and Y.
- All the vertices present in the set X should only be connected to the vertices present in the set Y with some edges. That means the vertices present in a set should not be connected to the vertex that is present in the same set.
- Both the sets that are created should be distinct that means both should not have the same vertices in them.



The graph shown in the above image is divided into two vertices named set X and set Y. The contents of these sets are,

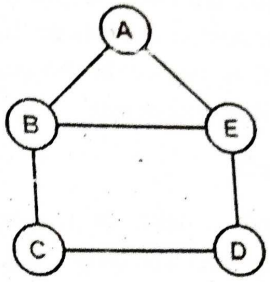
Set X = {vertex A, vertex B, vertex C, vertex D}

Set Y = {vertex P, vertex Q, vertex R}

The vertex A of the set X is associated with the vertex Q of the set Y. And the vertex B is also connected to the vertex Q. The vertex C of the set X is connected to the two vertices of the set Y named vertex P and vertex R. The vertex D of the set X is associated with the vertex Q of the set R.

Planar Graph

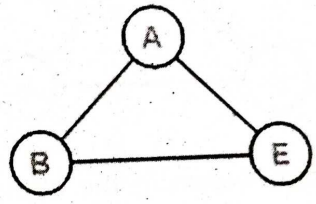
A graph is called a planar graph if that graph can be drawn in a single plane with any two of the edges intersecting each other.



The graph shown in the above image can be drawn in a single plane with any two edges intersecting. Thus it is a planar graph.

Simple Graph

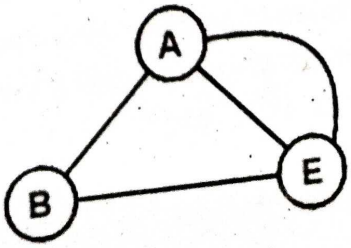
A graph is said to be a simple graph if the graph doesn't consist of no self-loops and no parallel edges in the graph.



We have three vertices and three edges for the graph that is shown in the above image. This graph has no self-loops and no parallel edges; therefore, it is called a simple graph.

Multi Graph

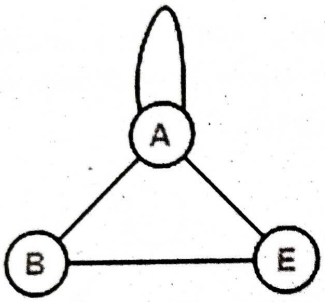
A graph is said to be a multigraph if the graph doesn't consist of any self-loops, but parallel edges are present in the graph. If there is more than one edge present between two vertices, then that pair of vertices is said to be having parallel edges.



We have three vertices and three edges for the graph that is shown in the above image. There are no self-loops, but two edges connect these two vertices between vertex A and vertex E of the graph. In other words, we can say that if two vertices of a graph are connected with more than one edge in a graph, then it is said to be having parallel edges, thus making it a multigraph.

Pseudo Graph

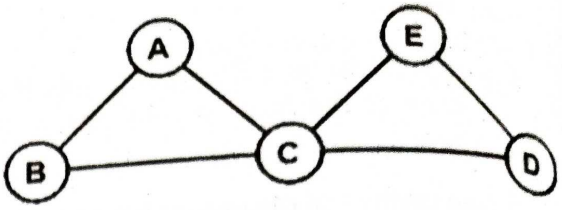
If a graph consists of no parallel edges, but self-loops are present in a graph, it is called a pseudo graph. The meaning of a self-loop is that there is an edge present in the graph that starts from one of the graph's vertices, and if that edge ends on the same vertex, then it is called a pseudo graph.



The graph shown in the above image has vertex A, vertex B and vertex E. There are four edges in this graph, and there are three edges associated with vertex A, and among these three edges, one of the edges is a self-loop. And among these four edges present in there is no parallel edge in it. Since the graph shown above has a self-loop and no parallel edge present in it, thus it is a pseudo graph.

Euler Graph

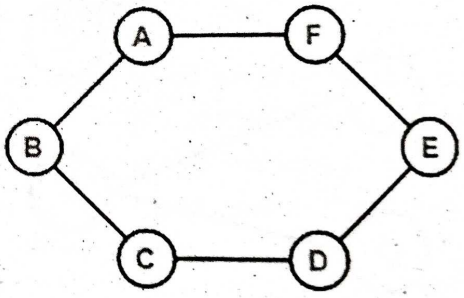
If all the vertices present in a graph have an even degree, then the graph is known as an Euler graph. By degree of a vertex, we mean the number of edges that are associated with a vertex. So for a graph to be an Euler graph, it is required that all the vertices in the graph should be associated with an even number of edges.



In the graph shown in the above image, we have five vertices named vertex A, vertex B, vertex C, vertex D and vertex E. All the vertices except vertex C have a degree of 2, which means they are associated with two edges each of the vertex. At the same time, vertex C is associated with four edges, thus making it degree 4. The degree of vertex C and other vertices is 4 and 2, respectively, which are even. Therefore, the graph displayed above is an Euler graph.

Hamilton Graph

Suppose a closed walk in the connected graph that visits every vertex of the graph exactly once (except starting vertex) without repeating the edges. Such a graph is called a Hamiltonian graph, and such a walk is called a Hamiltonian path. The Hamiltonian circuit is also known as Hamiltonian Cycle.



In other words, A Hamiltonian path that starts and ends at the same vertex is called a Hamiltonian circuit. Every graph that contains a Hamiltonian circuit also contains a Hamiltonian path, but vice versa is not true. There may exist more than one Hamiltonian path and Hamiltonian circuit in a graph.

The graph shown in the above image consists of a closed path ABCDEFA which starts from vertex A and traverses all other vertices or nodes without traversing any of the nodes twice other than vertex A in the path of traversal. Therefore, the graph shown in the above image is a Hamilton graph.

2.3 Representation of Graph

Q26. What are the various ways to represent graphs.

Ans : **(Imp.)**

Graph data structure is represented using following representations...

Adjacency Matrix

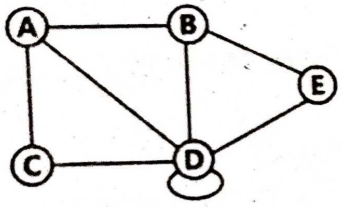
Incidence Matrix

Adjacency List

Adjacency Matrix

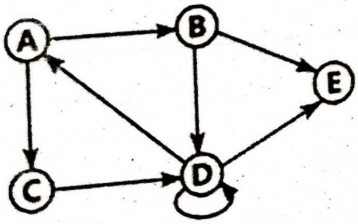
In this representation, graph can be represented using a matrix of size total number of vertices by total number of vertices. That means if a graph with 4 vertices can be represented using a matrix of 4X4 class. In this matrix, rows and columns both represents vertices. This matrix is filled with either 1 or 0. Here, 1 represents there is an edge from row vertex to column vertex and 0 represents there is no edge from row vertex to column vertex.

For example, consider the following undirected graph representation...



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

Directed graph representation...

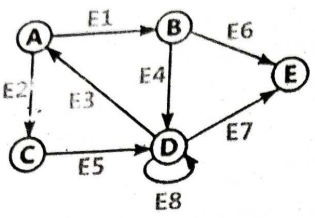


	A	B	C	D	E
A	0	1	1	0	0
B	0	0	0	1	1
C	0	0	0	1	0
D	1	0	0	1	1
E	0	0	0	0	0

Incidence Matrix

In this representation, graph can be represented using a matrix of size total number of vertices by total number of edges. That means if a graph with 4 vertices and 6 edges can be represented using a matrix of 4X6 class. In this matrix, rows represents vertices and columns represents edges. This matrix is filled with either 0 or 1 or -1. Here, 0 represents row edge is not connected to column vertex, 1 represents row edge is connected as outgoing edge to column vertex and -1 represents row edge is connected as incoming edge to column vertex.

For example, consider the following directed graph representation...

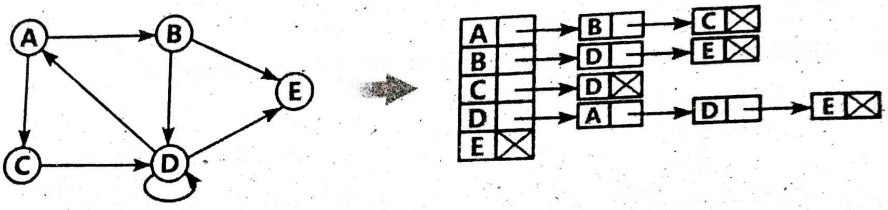


	E1	E2	E3	E4	E5	E6	E7	E8
A	1	1	-1	0	0	0	0	0
B	-1	0	0	1	0	1	0	0
C	0	-1	0	0	1	0	0	0
D	0	0	1	-1	-1	0	1	1
E	0	0	0	0	0	-1	-1	0

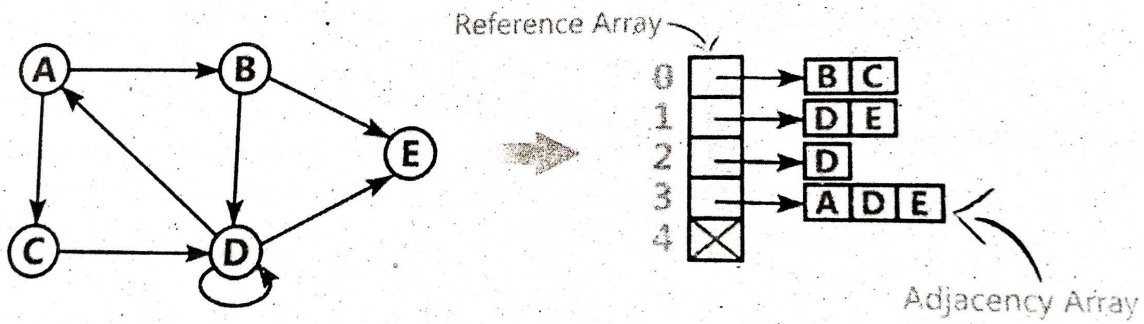
Adjacency List

In this representation, every vertex of graph contains list of its adjacent vertices.

For example, consider the following directed graph representation implemented using linked list...



This representation can also be implemented using array as follows..



4.2.4 Elementary Graph Operations- Dfs And Bfs

Q27. Explain DFS algorithm with an example.

Ans :

(Imp.)

DFS (Depth First Search)

DFS traversal of a graph, produces a spanning tree as final result. Spanning Tree is a graph without any loops. We use Stack data structure with maximum size of total number of vertices in the graph to implement DFS traversal of a graph.

We use the following steps to implement DFS traversal...

- **Step 1:** Define a Stack of size total number of vertices in the graph.
- **Step 2:** Select any vertex as **starting point** for traversal. Visit that vertex and push it on to the Stack.

Step 3: Visit any one of the **adjacent** vertex of the vertex which is at top of the stack which is not visited and push it on to the stack.

Step 4: Repeat step 3 until there are no new vertex to be visit from the vertex on top of the stack.

Step 5: When there is no new vertex to be visit then use **back tracking** and pop one vertex from the stack.

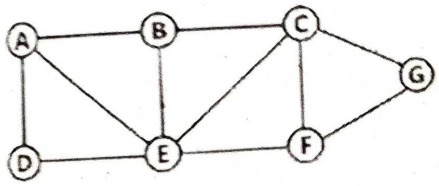
Step 6: Repeat steps 3, 4 and 5 until stack becomes Empty.

Step 7: When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

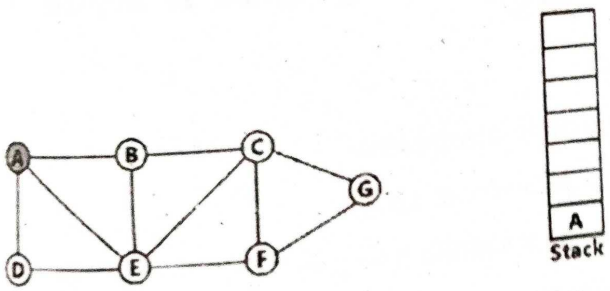
Back tracking is coming back to the vertex from which we came to current vertex.

Example

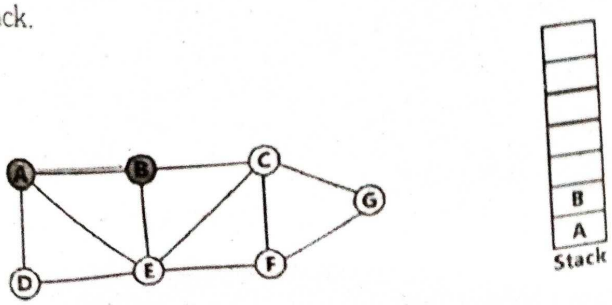
Consider the following example graph to perrom DFS traversar.



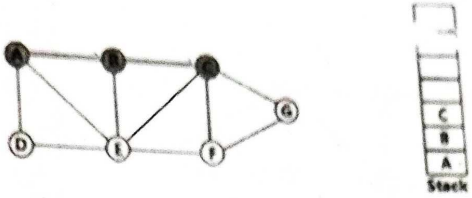
Step 1 : Select the vertex A as starting point (visit A)



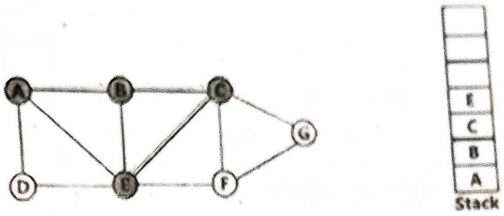
Step 2 : Visit any adjacent vertex of A which is not visited (B). Push newly visited vertex B on to the stack.



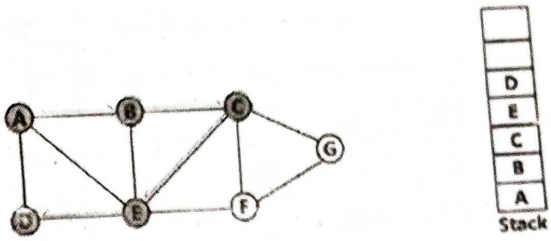
Step 3 : Visit any adjacent vertext of B which is not visited (C) Push control the stack.



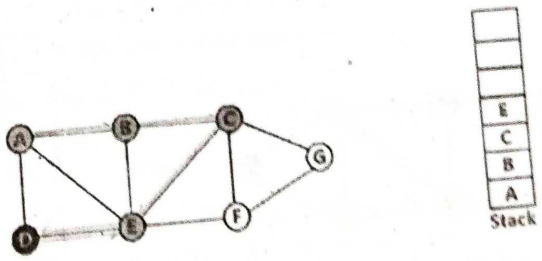
Step 4 : Visit any adjacent vertex of C which is not visited (E) Push E on to the stack.



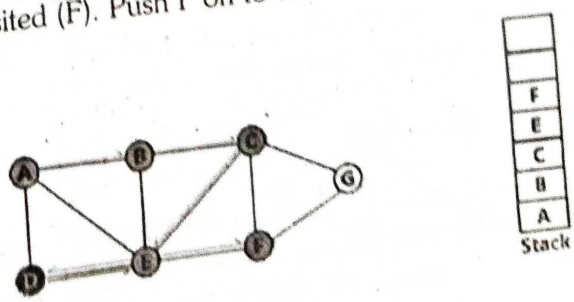
Step 5: Visit any adjacent vertext of E which is not visited (D) Push D on to the Stack.



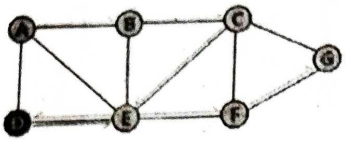
Step 6: There is no new vertiex to be visited from D. So use back track. Pop D from the stack.

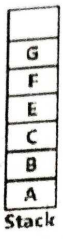


Step 7 : Visit any adjacent vertex of E which is not visited (F). Push F on to the stack.

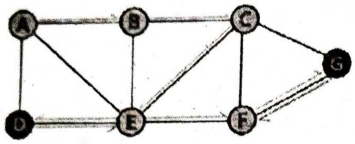


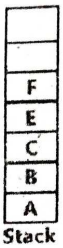
Step 8: Vist any adjacent vertex of F which is not visited (G). Push G on to the Stack.



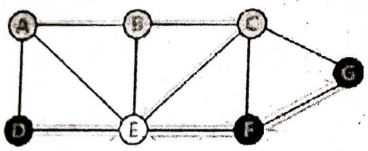


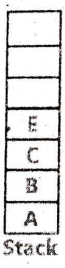
Step 9: There is no new vertiex to be visited from F. So use back track. Pop g from the stack.



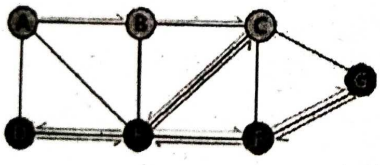


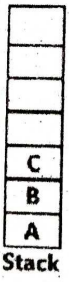
Step 10: There is no new vertiex to be visited from F. So use back track. Pop F from the stack.



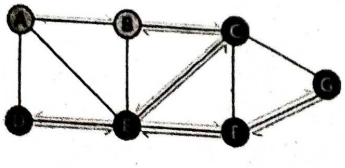


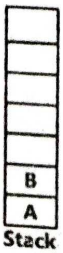
Step 11: There is no new vertiex to be visited from E. So use back track. Pop F from the Stack.



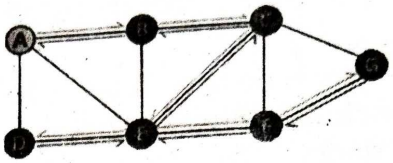


Step 12: There is no new vertiex to be visited from C. So use back track. Pop C from the stack.

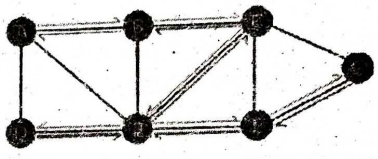


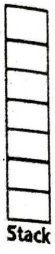


Step 13: There is no vertiex to be visited from B. So use back track. Pop B from stack.

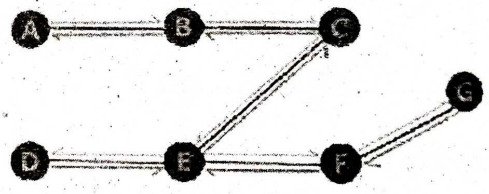


Step 14: There is no new vertex to be visited from A. So use back track. Pop A from the stack.





Stack became Empty. So stop DFS Treversa. Final result of DFS traversa is following spanning tree.



Q28. Write a C++ program to implement DFS .

Ans :

```
#include<iostream>
#include<conio.h>
#include<stdlib.h>
using namespace std;
intcost[10][10],i,j,k,n,stk[10], top, v,visit[10],
visited[10];

main()
{
int m;
cout <<"enterno of vertices";
```



```
cin >> n;
cout <<"e
cin >> m;
cout <<"
for(k=1;k
{
cin >>i>
cost[i][j]=
}

cout <<
cin >>v;
cout <<
cout <<
visited[v]
k=1;
while(k<
{
for(j=n;j
if(cost[v]
{
visit[j]=
stk[top]
top++;
}
v=stk[-
cout<<
k++;
visit[v]=
}
}
```

OUTPUT

```
enterno
ente no
```

```
cin >> n;
cout <<"ente no of edges";
cin >> m;
cout <<"\nEDGES \n";
for(k=1;k<=m;k++)
{
cin >>i>>j;
cost[i][j]=1;
}

cout <<"enter initial vertex";
cin >>v;
cout <<"ORDER OF VISITED VERTICES";
cout << v <<" ";
visited[v]=1;
k=1;
while(k<n)
{
for(j=n;j>=1;j—)
if(cost[v][j]!=0 && visited[j]!=1 && visit[j]!=1)
{
visit[j]=1;
stk[top]=j;
top++;
}
v=stk[—top];
cout<<v << " ";
k++;
visit[v]=0; visited[v]=1;
}
}
```

OUTPUT

```
enterno of vertices9
ente no of edges9
EDGES
1 2
2 3
2 6
1 5
1 4
4 7
5 7
7 8
8 9
enter initial vertex1
ORDER OF VISITED VERTICES1 2 3 6 4 7 8 9 5
```

Q29. Explain about BFS algorithm with an example.

Ans :

BFS (Breadth First Search)

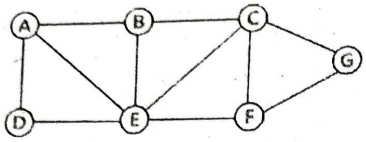
BFS traversal of a graph, produces a spanning tree as final result. Spanning Tree is a graph without any loops. We use Queue data structure with maximum size of total number of vertices in the graph to implement BFS traversal of a graph.

We use the following steps to implement BFS traversal...

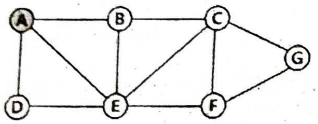
- **Step 1:** Define a Queue of size total number of vertices in the graph.
- **Step 2:** Select any vertex as starting point for traversal. Visit that vertex and insert it into the Queue.
- **Step 3:** Visit all the adjacent vertices of the verex which is at front of the Queue which is not visited and insert them into the Queue.
- **Step 4:** When there is no new vertex to be visit from the vertex at front of the Queue then delete that vertex from the Queue.
- **Step 5:** Repeat step 3 and 4 until queue becomes empty.
- **Step 6:** When queue becomes Empty, then produce final spanning tree by removing unused edges from the graph.

Examples

Consider the following example graph to performs BFS traversal.



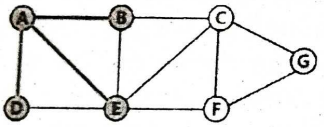
➢ **Step 1:** Select the vertex A as starting point (visit A) Insert A into the Queue.



Queue

A						

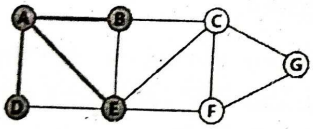
➢ **Step 2 :** Visit all adjacent vertices of A which are not visited (D, E, B). Insert newly visited vertices into the Queue and delete A from the Queue.



Queue

	D	E	B			

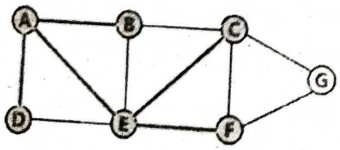
➢ **Step 3:** Visit all ajacent vertices of D which are not visited (there is no vertex). Delete D from the Queue.



Queue

		E	B			

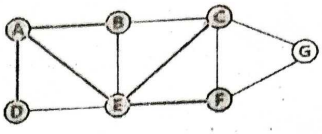
➢ **Step 4 :** Visit all adjacent verlices of E which are not visited (C, F). Insert newly visited vertices into the Queue and delete E from the Queue.



Queue

			B	C	F	

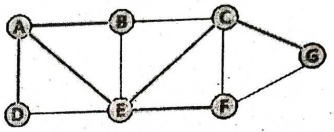
➢ **Step 5:** Visit all adjacent vertices of B which are not visited (there is no vertex). Delete B from the Queue.



Queue

				C	F	

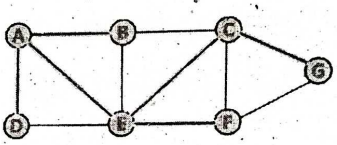
➢ **Step 6 :** Visit all adjacent vertices of C which are not visited (G). Insert newly visited into the Queue and delete C from the Queue.



Queue

					F	G

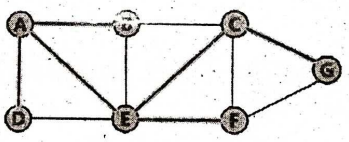
➢ **Step 7:** Visit all adjacent vertices of F which are not visited (there is no vertex). Delete F from the Queue.



Queue

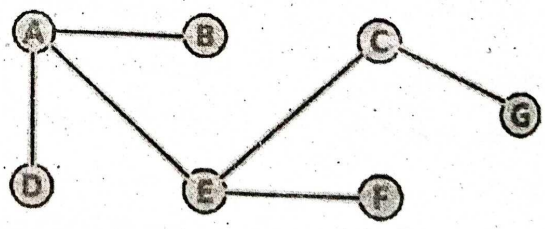
						G

➢ **Step 8:** Visit all adjacent vertices of G which are not visited (there is no vertex). Delete G from the Queue.



Queue

Queue became empty. So, stop the BFS process. Final result of BFS is a Spanning Tree as shown below.



Q30. Write a C++ program for implementation on BFS.

Ans :

```
#include <iostream>
#include <conio.h>
usingnamespace std;
```

```
int c =0, t =0;
struct node_info
{
int no;
int st_time;
}*q =NULL, *r =NULL, *x =NULL;
struct node
{
   node_info *pt;
   node *next;
}*front =NULL, *rear =NULL, *p =NULL, *np
=NULL;
void push(node_info *ptr)
{
   np =new node;
   np->pt = ptr;
   np->next =NULL;
if(front ==NULL)
{
      front = rear = np;
      rear->next =NULL;
}
else
{
      rear->next = np;
      rear = np;
      rear->next =NULL;
}
}
node_info *remove()
{
if(front ==NULL)
{
cout<<"empty queue\n";
}
else
{
      p = front;
      x = p->pt;
       front = front->next;
delete(p);
return(x);
}
}
void bfs(int*v,int am[][7],int i)
{
if(c ==0)
{
       q =new node_info;
       q->no = i;
       q->st_time = t++;
 cout<<"time of visitation for node "<<q-
>no<<":"<<q->st_time<<"\n\n";
       v[i]=1;
       push(q);
 }
    c++;
 for(int j =0; j <7; j++)
 {
 if(am[i][j]==0||(am[i][j]==1&& v[j]==1))
 continue;
 elseif(am[i][j]==1&& v[j]==0)
 {
          r =new node_info;
          r->no = j;
          r->st_time = t++;
  cout<<"time of visitation for node "<<r-
>no<<":"<<r->st_time<<"\n\n";
          v[j]=1;
          push(r);
   }
   }
   remove();
   if(c <=6&& front !=NULL)
```

```
        bfs(v, am, remove()->no);
    }
    int main()
    {
    int v[7], am[7][7];
    for(int i =0; i <7; i++)
        v[i]=0;
    for(int i =0; i <7; i++)
    {
    cout<<"enter the values for adjacency matrix row:"<<i+1<<endl;
    for(int j =0; j <7; j++)
    {
    cin>>am[i][j];
    }
    }
      bfs(v, am, 0);
      getch();
  1. }
```

Output

```
enter the values for adjacency matrix row:1
0
1
1
0
0
1
1
enter the values for adjacency matrix row:2
1
0
0
0
0
0
0
enter the values for adjacency matrix row:3
1
0
0
0
0
0
1
time of visitation for node 0:0
time of visitation for node 1:1
time of visitation for node 2:2
```

Short Question and Answers

1. What is tree data structure?

Ans :

A tree data structure can also be defined as follows.

Tree data structure is a collection of data (Node) which is organized in hierarchical structure and this is a recursive definition

In tree data structure, every individual element is called as Node. Node in a tree data structure, stores the actual data of that particular element and link to next element in hierarchical structure.

In a tree data structure, if we have N number of nodes then we can have a maximum of N-1 number of links.

2. What is binary tree?

Ans :

In a normal tree, every node can have any number of children. Binary tree is a special type of tree data structure in which every node can have a maximum of 2 children. One is known as left child and the other is known as right child.

A tree in which every node can have a maximum of two children is called as Binary Tree.

In a binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2 children.

3. Explain ADT of Binary Tree.

Ans :

A binary tree consists of nodes with one predecessor and at most two successors (called the left child and the right child). The only exception is the root node of the tree that does not have a predecessor. A node can contain any amount and any type of data.

- **Create:** A binary tree may be created in several states. The most common include ...
  - An empty tree (i.e. no nodes) and hence the constructor will have no parameters
  - A tree with one node (i.e. the root) and hence the constructor will have a single parameter (the data for the root node)
  - A tree with a new root whose children are other existing trees and hence the constructor will have three parameters (the data for the root node and references to its subtrees)
- **isEmpty():** Returns true if there are no nodes in the tree, false otherwise.
- **isFull():** May be required by some implementations. Returns true if the tree is full, false otherwise.
- **clear():** Removes all of the nodes from the tree (essentially reinitializing it to a new empty tree).
- **add(value):** Adds a new node to the tree with the given value. The actual implementation of this method is determined by the purpose for the tree and how the tree is to be maintained and/or processed. To begin with, we will assume no particular purpose or order and therefore add new nodes in such a way that the tree will remain nearly balanced.

4 Binary Tree Traversal

Ans :

When we wanted to display a binary tree, we need to follow some order in which all the nodes of that binary tree must be displayed. In any binary tree displaying order of nodes depends on the traversal method.

Displaying (or) visiting order of nodes in a binary tree is called as Binary Tree Traversal

There are three types of binary tree traversals.

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

5. Binary Search Tree

Ans :

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

Let's understand the concept of Binary search tree with an example.

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

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

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

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

6. What is a complete binary tree?

Ans :

A complete binary tree is a binary tree in which all the levels except the last level, i.e., leaf node should be completely filled, and all the nodes should be left-justified.

7. What is Heap?

Ans :

A heap is a complete binary tree, and the binary tree is a tree in which the node can have utmost two children. Before knowing more about the heap data structure, we should know about the complete binary tree.

8. Heap data structure.

Ans :

Heap data structure is a complete binary tree that satisfies the heap property, where any given node is

- always greater than its child node/s and the key of the root node is the largest among all other nodes. This property is also called max heap property.
- always smaller than the child node/s and the key of the root node is the smallest among all other nodes. This property is also called min heap property.

9. What is B-Tree?

Ans :

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

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

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

10. What is graph?

Ans :

Graph is a non linear data structure, it contains a set of points known as nodes (or vertices) and set of links known as edges (or Arcs) which connects the vertices. A graph is defined as follows...

- Graph is a collection of vertices and arcs which connects vertices in the graph
- Graph is a collection of nodes and edges which connects nodes in the graph

Generally, a graph G is represented as G = (V , E), where V is set of vertices and E is set of edges.

Choose the Correct Answers

1. A binary tree in which if all its levels except possibly the last, have the maximum number of nodes and all the nodes at the last level appear as far left as possible, is called [d]
   (a) Full binary tree (b) Binary Search Tree
   (c) Threaded tree (d) Complete binary tree

2. A full binary tree with n leaves contains [c]
   (a) n nodes (b) log2n nodes
   (c) 2n - 1 nodes (d) 2n+1 nodes

3. If each node in a tree has value greater the every value in its left subtree and has value less than every value in its right subtree, the tree is called [c]
   (a) Complete tree (b) Full binary tree
   (c) Binary search tree (d) AVL tree

4. The smallest number of key that will force a B-tree of order 3 to have a height 3 is [c]
   (a) 12 (b) 10
   (c) 7 (d) None of these

5. B+- trees are preferred to binary trees in databases because [b]
   (a) Disk capacities are greater than memory capacities
   (b) Disk access is much slower than memory Access
   (c) Disk data transfer rates are much less than memory data transfer rates.
   (d) All of above

6. What is the time complexity for checking if an undirected graph with E edges and V vertices is Bipartite, given its adjacency matrix? [d]
   (a) O(E) (b) O(V)
   (c) O(E*E) (d) O(V*V)

7. A graph in which all vertices have equal degree is known as _____ [a]
   (a) Complete graph (b) Regular graph
   (c) Multi graph (d) Simple graph

8. The number of edges in a complete graph of n vertices is [b]
   (a) n(n+1)/2 (b) n(n-1)/2
   (c) n2/2 (d) n

9. A graph in which all vertices have equal degree is known as _____ [a]
   (a) Complete graph (b) Regular graph
   (c) Multi graph (d) Simple graph

10. A vertex of in-degree zero in a directed graph is called a/an [c]
    (a) Root vertex (b) Isolated vertex
    (c) Sink (d) Articulation point

Fill in the Blanks

1. A full binary tree with n non-leaf nodes contains _____ nodes.
2. Traversing a binary tree first root and then left and right subtrees called _____ traversal.
3. A _____ binary tree in which all its levels except the last, have maximum numbers of nodes, and all the nodes in the last level have only one child it will be its left child.
4. The maximum number of nodes on level i of a binary tree is _____
5. If two trees have same structure and but different node content, then they are called _____
6. A full binary tree with n leaves contains _____ nodes
7. A graph is a tree if and only if graph is _____
8. The number of possible undirected graphs which may have self loops but no multiple edges and have n vertices is _____
9. Number of vertices with odd degrees in a graph having a eulerian walk is _____
10. What is the number of vertices of degree 2 in a path graph having n vertices,here n>2.

Answers

1. $2n + 1$ nodes
2. Preorder
3. Complete binary tree
4. $2^{i}-1$
5. Similar trees
6. $2n - 1$ nodes
7. Contains no cycles
8. $2^{((n*n)/2)}$
9. Either 0 or 2
10. n-2