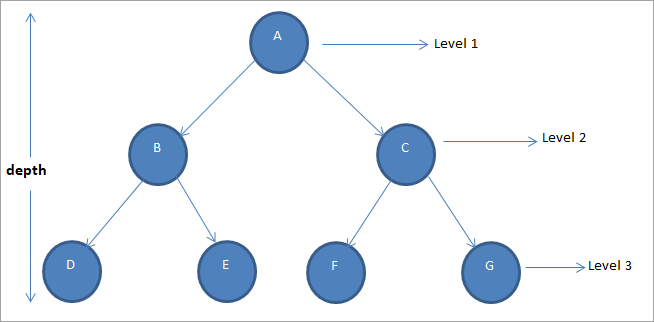
# **LEC-9 Tree**

Some Terminologies related to trees:



* **Root of a tree:**
* **Degree of a Node:**
  + No of sub-trees of a node
  + e:g of A is 3
* **Degree of tree:**
  + Maximum of all the degrees of nodes
  + e:g degree of tree in example is?
* **Leaf Node(s)/Terminal Node(s):**
  + Node(s) having 0 degree.
* **Non-Terminal Nodes**
* **Parent**
  + Root is also called Parent Node
* **Child Node(s)**
* **Siblings**
  + Nodes of same parent
* **Level of Node**
* **Depth/Height of a tree:**
  + Maximum level at which node exists
* **Binary Tree(s):**
  + Tree in which a node can have at most 2 child nodes.



* **Complete Binary tree:**
  + Binary Tree having leaf Nodes at adjacent level
  + E:g diagram 1 is complete binary tree.
* **Skewed Tree:**
  + Diagram 2 is left skewed Binary tree
* **Max no of Nodes at a specific level(k) 2^k-1**
* **Max no of nodes in a Binary Tree:**
  + 2^k-1
* **Full Binary Tree:**
  + Binary Tree with a max no of nodes.
* **Tree Traversal**
  + E:g

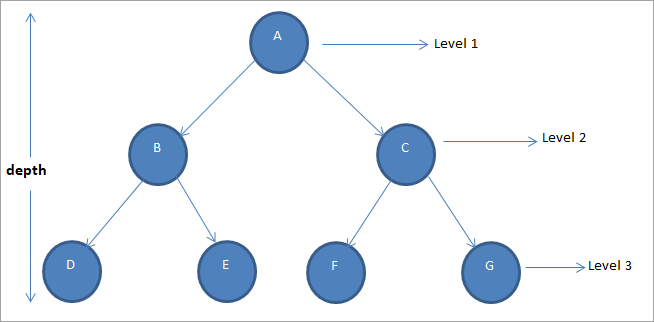
**N**

**R**

**L**

* **In-order traversal**
  + LNR or RNL
* **Pre-order:**
  + NLR or NRL
* **Post-order:**
  + LRN or RLN
* **Implementation of Binary Tree using array:**

|  |
| --- |
|  |



Root : will be at index 1

Left child : 2i

Right child : 2i + 1

Parent : [i/2]

**Drawback?**

**Solution:**

Implementation using LinkedList

#include <iostream>

using namespace std;

class BinaryTree;

class TreeNode{

    friend class BinaryTree;

    int data;

    TreeNode\* left, \*right;

};

class BinaryTree{

    TreeNode root;

    public:

    BinaryTree(){

        root=0;

    }

    void inOrder(){

        inOrder(root);

    }

};

## **Traversal Methods**

Define the following function with private access, before

* **inOrder(TreeNode\* t)**
* **PreOrder(TreeNode\* t)**
* **PostOrder(TreeNode\* t)**

    void inOrder(TreeNode\* t){

        if(t!=0){

            inOrder(t->left); //L

            cout<<t->data<<endl; //N

            inOrder(t->right); //R

        }

    }// LNR

    void PreOrder(TreeNode\* t){

        if(t!=0){

            cout<<t->data<<endl; //N

            PreOrder(t->left); //L

            PreOrder(t->right); //R

        }

    }// NLR

    void PostOrder(TreeNode\* t){

        if(t!=0){

            PostOrder(t->left); //L

            PostOrder(t->right); //R

            cout<<t->data<<endl; //N

        }

    }// LRN

* **Binary Search Tree**
  + A binary tree is BST if it has following:
    - Every Node has unique value/key
    - There should be a smaller value than root value on left side and right side of the tree should contain higher value than the root value/key
    - Left and right sub-tree(s) should also be a BST(s)
    - E:g 9,5,1,8,3,10,16 🡪9=Root
* Draw BST:

## **Implementation of BST:**

#include <iostream>

using namespace std;

class BST:

class TreeNode{

private:

    friend class BST;

    int data;

    TreeNode\* left, \*right;

public:

    TreeNode(int val){

        data=val;

        left=0;

        right=0;

    }//paramterized constructor

    TreeNode();// default constructor

};

class BST{

private:

    TreeNode\* root;

public:

    BST();

    ~BST();

    bool search(int val){

        TreeNode\* t=root;

        while(t!=0){

            if(t->data==val){

                return true;

            }

            else if(val<t->data){

                t=t->left;

            }

            else if(val>t->data){

                t=t->right;

            }

        }//end of while

        return false;

    }//end of function

};

## **Finding min,max in a Tree**

**Insert a Node in BST 2 methods:**

    void insert(TreeNode\* t,int val);

    void insert(int val){

        insert(root, val);

    }

    // definition of insert(TreeNode\* t,int val)

    void insert(TreeNode\* t,int val){

        if(t==0){

            t=new TreeNode;

            t->data=val;

            t->left=0;

            t->right=0;

        }

        else if(val<t->data){

            insert(t->left,val);

        }

        else if(val>t->data){

            insert(t->right,val);

        }

    }// end of insert method 1

    TreeNode\* insert2(TreeNode\* t,int val){

        if(t==0){

            return t=new TreeNode(val);

        }

        else if(val<t->data){

            insert(t->left,val);

        }

        else if(val>t->data){

            insert(t->right,val);

        }

    }// end of insert method 2

**Find minimum and maximum value of BST 2 methods:**

* **Minimum :**

    TreeNode\* mimimum(TreeNode\* t){

        while(t && t->left!=NULL){

            t=t->left;

        }

        return t;

    }//mimimum method 1

    TreeNode\* mimimum2(TreeNode\* t){

        if(t && t->left!=NULL){

           return mimimum2(t->left);

        }

        return t;

    }//mimimum method 2

* **Maximum :**

    TreeNode\* maximum(TreeNode\* t){

        while(t && t->right!=NULL){

            t=t->right;

        }

        return t;

    }//mimimum method 1

    TreeNode\* maximum2(TreeNode\* t){

        if(t && t->right!=NULL){

           return maximum2(t->right);

        }

        return t;

    }//mimimum method 2

## **Delete a node**

**Delete Tree Node:**

   bool remove(int val){

        TreeNode\* curr=root,\*parent=0;

        while(curr!=0 && curr->data!=val){

            parent=curr;

            if(val<curr->data){

                curr=curr->left;

            }

            else{

                curr=curr->right;

            }

            if(curr==0){

                return false;

            }

            else{

                if(curr!=0 && curr->right!=0){

                    TreeNode\* l=curr->left,\*pl=curr;

                    while(l->right!=0){

                        pl=l;

                        l=l->right;

                    }

                    curr->data=l->data;

                    curr=l;

                    parent=pl;

                }

                TreeNode\* child;

                if(curr->left!=0){

                    child=curr->left;

                }

                else{

                    child=curr->right;

                }

                if(curr==root){

                    root=child;

                }

                else{

                    if(curr==parent->right){

                        parent->right=child;

                    }

                    else{

                        parent->left=child;

                    }

                    delete curr;

                    curr=0;

                    return true;

                }

            }

        }

    }