Lab 10

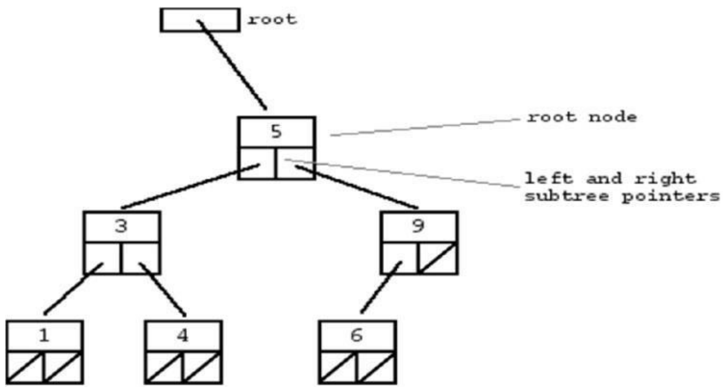
Binary trees, insertion/deletion in binary trees and tree traversal

Objective:

- To learn the basic concepts and implementation of trees in Java.

Introduction

Tree is a nonlinear data structure that models a hierarchical organization. The characteristic features are that each element may have several successors (called its“children”) and every element except one (called the “root”) has a unique predecessor (called its “parent”). A Binary Tree is made of nodes, where each node contains a "left" pointer, a "right" pointer, and a data element. The "root" pointer points to the topmostnode in the tree. The left and right pointers recursively point to smaller "subtrees" on either side. A null pointer represents a binary tree with no elements the empty tree. A binary tree is either empty (represented by a null pointer), or is made of a single node, where the left and right pointers (recursive definition ahead) each point to a binary tree. A rooted tree is a tree in which a special ("labeled") node is singled out. This node is called the "root" or (less commonly) "eve" of the tree. Rooted trees are equivalent oriented trees.



A "binary search tree" (BST) or "ordered binary tree" is a type of binary tree where the nodes are arranged in order: for each node, all elements in its left subtree are lessorequalto the node (<=), and all the elements in its right subtree are greater than the node (>). The tree shown above is a binary search tree the "root" node is a 5, and its left subtree nodes (1, 3, 4) are <= 5, and its right subtree nodes (6, 9) are > 5. Recursively, each of the subtrees must also obey the binary search tree constraint: in the (1, 3, 4) subtree, the 3 is the root, the 1 <= 3 and 4 > 3. Watch out for the exact wording in the problems a "binary search tree" is different from a "binary tree".

The nodes at the bottom edge of the tree have empty subtrees and are called "leaf" nodes (1, 4, 6) while the others are "internal" nodes (3, 5, 9).

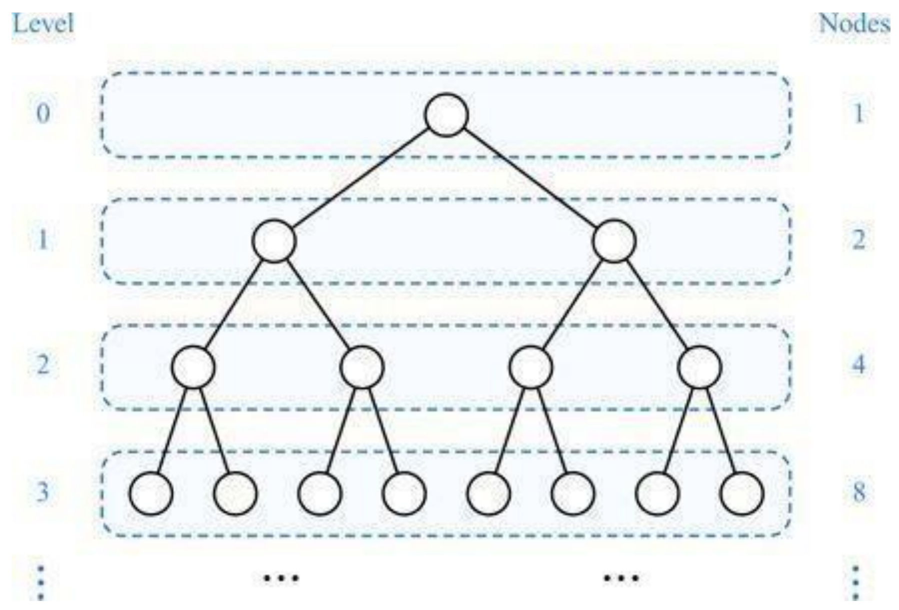
Programming Strategy:

When thinking about a binary tree problem, it's often a good idea to draw a few little trees to think about the various cases. For any problem related to binary trees, there are two things to understand.

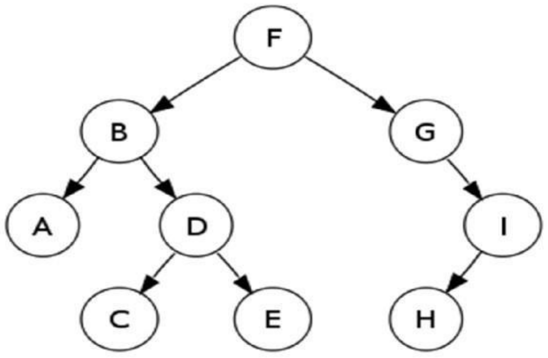
- The node/pointer structure that makes up the tree and the code that manipulates it.
- The algorithm, typically recursive, that iterates over the tree.

Criteria of Binary Trees:

- Binary trees have several interesting properties dealing with relationships between their heights and number of nodes.
- We denote the set of all nodes of a tree T at the same depth d as the level d of T.
- In a binary tree, level 0 has at most one node (the root), level 1 has at most two nodes (the children of the root), and level 2 has at most four nodes, and so on.
- We can see that the maximum number of nodes on the levels of a binary tree grows exponentially as we go down the tree.



Traversals of Binary Trees:



Depth First Search:

- **Preorder** traversal sequence: F, B, A, D, C, E, G, I, H (root, left, right)
- **Inorder** traversal sequence: A, B, C, D, E, F, G, H, I (left, root, right); note how this produces a sorted sequence
- **Postorder** traversal sequence: A, C, E, D, B, H, I, G, F (left, right, root)

Breadth First:

- **Levelorder** traversal sequence: F, B, G, A, D, I, C, E, H

Insertion into binary Tree

```
  struct TreeNode
{
      int value;
TreeNode *left;
      TreeNode *right;
};
TreeNode *root=NULL;
bool printGivenLevel(TreeNode* root, int level); void
insertNode(int num)
{
 TreeNode *temp= new TreeNode; // Pointer to traverse the
tree   TreeNode *nodePtr= new TreeNode; // Create a new node
temp->value = num;   temp->left = temp->right = NULL;
     if (!root) // Is the tree empty?
   root = temp;       else{
     nodePtr = root;
     while (nodePtr != NULL)
        {
             if (num < nodePtr->value)
              {
               if (nodePtr->left)
           nodePtr = nodePtr->left;
           else
                 {
                        nodePtr->left = temp;
                         break;
                    }
              }
               else if (num > nodePtr->value)
              {
                    if (nodePtr->right)
                          nodePtr = nodePtr->right;
```

```
                              else
                              {
                                     nodePtr->right = temp;
             break;
                    }
       }                         else

                        {
                               cout << "Duplicate value found in tree.\n";

                              break;
                    }}}}
```

Inorder, Preorder, Postorder Traversal

```
void displayInorder(TreeNode *nodePtr)
{
 if(nodePtr)
 {
   displayInorder(nodePtr->left);
cout<<nodePtr->value<<",";
         displayInorder(nodePtr->right);
 }  }

void displayPreorder(TreeNode *nodePtr)
{
 if(nodePtr)
 {
   cout<<nodePtr->value<<",";
 displayPreorder(nodePtr->left);
         displayPreorder(nodePtr->right);

 }}

void displayPostorder(TreeNode *nodePtr)
{
 if(nodePtr)  {
         displayPostorder(nodePtr->left);
displayPostorder(nodePtr->right);
cout<<nodePtr->value<<",";
}}
```

Level-order Traversal

```
void printLevelOrder(TreeNode* nodePtr)
{
 if(nodePtr == NULL){
        return;
 }
    int level = 1;

       while(printGivenLevel(nodePtr, level)){
level++;
    }  }    bool printGivenLevel(TreeNode* nodePtr,
int level)
{
    if (nodePtr == NULL)         return
false;     if
(level == 1)
{
        cout<<nodePtr->value<<",";
return true;      }
        bool left = printGivenLevel(nodePtr->left, level-1);
bool right = printGivenLevel(nodePtr->right, level-1);
             return left
|| right;
}  int
main(void) {

 cout << "Inserting nodes. ";
insertNode(5);   insertNode(3);   insertNode(9);
insertNode(1);   insertNode(4);   insertNode(6);
insertNode(10);  insertNode(-1);  insertNode(2);
 cout << "Done.\nInorder Display:\n";  displayInorder(root);
cout  <<  "\nPreorder  Display:\n";    displayPreorder(root);
cout << "\nPostorder Display:\n";  displayPostorder(root);
         cout << "\nLevelorder Display:\n";
  printLevelOrder(root);
return 0;
}
```