Faculty of Computing



Data Structure and Algorithm

Lab Manual

# **Lab 13: Trees**

1. **Activity Time boxing**

|  |  |  |  |
| --- | --- | --- | --- |
| **Task No.** | **Activity Name** | **Activity time** | **Total Time** |
| 1 | Lecture | 20 min |  |
| 2 | Walkthrough Tasks | 10 min |  |
| 3 | Practice tasks | 110 min |  |
| 4 | Evaluation | 40 min | 180 |

1. **Concept Map**

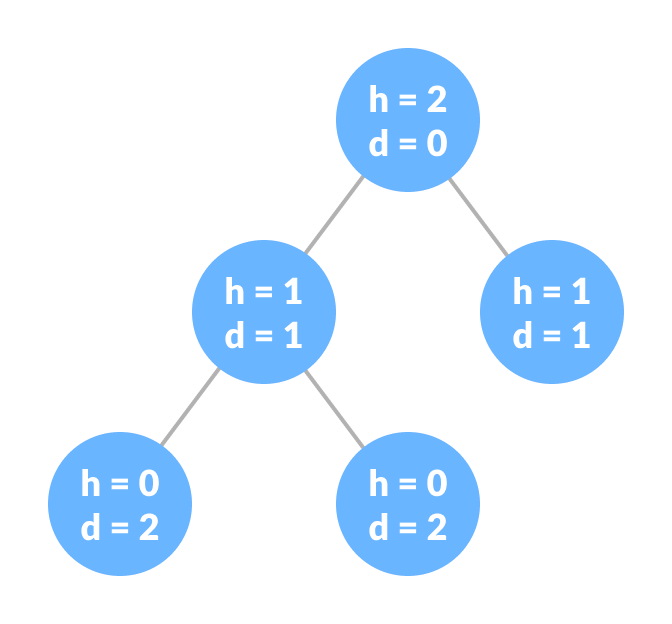
* Trees
* Binary Trees
* Binary Search Tree
* Insertion
* Deletion

## **Trees**

Trees are non-linear hierarchical data structures. A tree is a collection of nodes connected to each other by means of “edges” which are either directed or undirected. One of the nodes is designated as “Root node” and the remaining nodes are called child nodes or the leaf nodes of the root node.

**3.1 Some basic terms that we use for trees.**

* **Root node:** This is the topmost node in the tree hierarchy. In the above diagram, Node A is the root node. Note that the root node doesn’t have any parent.
* **Leaf node:** It is the Bottom most node in a tree hierarchy. Leaf nodes are the nodes that do not have any child nodes. They are also known as external nodes. Nodes E, F, G, H and C in the above tree are all leaf nodes.
* **Subtree**: Subtree represents various descendants of a node when the root is not null. A tree usually consists of a root node and one or more subtrees. In the above diagram, (B-E, B-F) and (D-G, D-H) are subtrees.
* **Parent node**: Any node except the root node that has a child node and an edge upward towards the parent.
* **Ancestor Node:** It is any predecessor node on a path from the root to that node. Note that the root does not have any ancestors. In the above diagram, A and B are the ancestors of E.
* **Key**: It represents the value of a node.
* **Level**: Represents the generation of a node. A root node is always at level 1. Child nodes of the root are at level 2, grandchildren of the root are at level 3 and so on. In general, each node is at a level higher than its parent.
* **Path**: The path is a sequence of consecutive edges. In the above diagram, the path to E is A=>B->E.
* **Degree**: Degree of a node indicates the number of children that a node has. In the above diagram, the degree of B and D is 2 each whereas the degree of C is 0.
* **Height of a Node:** The height of a node is the number of edges from the node to the deepest leaf (ie. the longest path from the node to a leaf node).

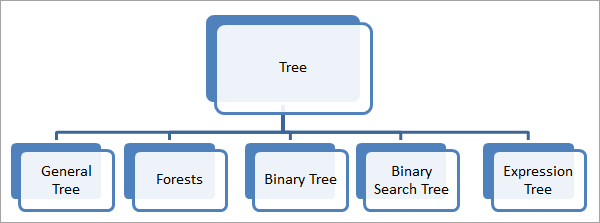


## **3.2. Why Tree Data Structure?**

Other data structures such as arrays, linked list, stack, and queue are linear data structures that store data sequentially. To perform any operation in a linear data structure, the time complexity increases with the increase in the data size. But it is not acceptable in today's computational world. Different tree data structures allow quicker and easier access to the data as it is a non-linear data structure.

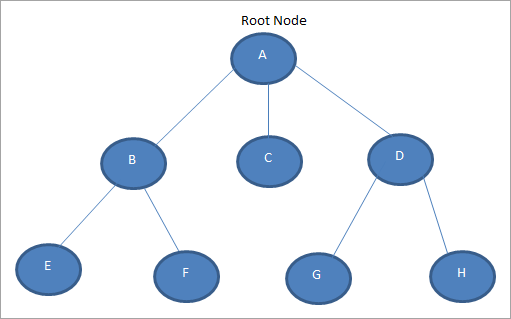
## **3.3. Types Of C++ Trees**

The tree data structure can be classified into the following subtypes as shown in the below diagram.

[](https://cdn.softwaretestinghelp.com/wp-content/qa/uploads/2019/08/2-tree-data-structure.png)

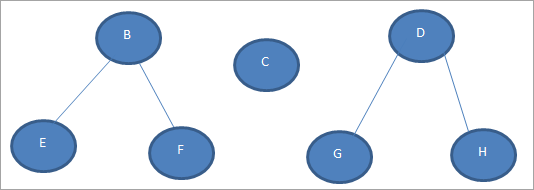
### 3.3.1. General Tree

The general tree is the basic representation of a tree. It has a node and one or more child nodes. The top-level node i.e. the root node is present at level 1 and all the other nodes may be present at various levels.

[](https://cdn.softwaretestinghelp.com/wp-content/qa/uploads/2019/08/3-General-Tree.png)

#### **3.3.2. Forests**

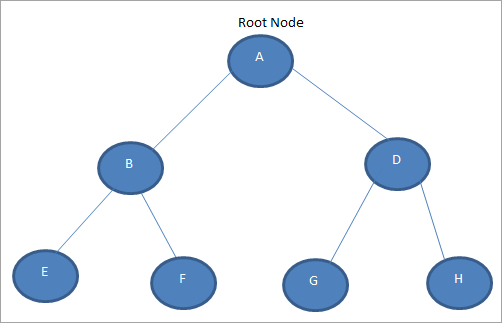
Whenever we delete the root node from the tree and the edges joining the next level elements and the root, we obtain disjoint sets of trees as shown below.

[](https://cdn.softwaretestinghelp.com/wp-content/qa/uploads/2019/08/4.-Forests.png)

In the above figure, we obtained two forests by deleting the root node A and the three edges that were connecting the root node to nodes B, C, and D

#### **3.3.3. Binary Tree:** A tree data structure in which each node has at most two child nodes is called a binary tree. A binary tree is the most popular tree data structure and is used in a range of applications like expression evaluation, databases, etc.

**The following figure shows a binary tree.**

[](https://cdn.softwaretestinghelp.com/wp-content/qa/uploads/2019/08/5.-Binary-tree.png)

In the above figure, we see that nodes A, B, and D have two children each. A binary tree in which each node has exactly zero or two children is called a full binary tree. In this tree, there are no nodes that have one child.

**Binary Search Tree:**

A binary search tree extends upon the concept of a binary tree. A binary search tree is set such that:-

1) Every left node is always lesser than its parent node

2) Every right node is always greater than its parent node

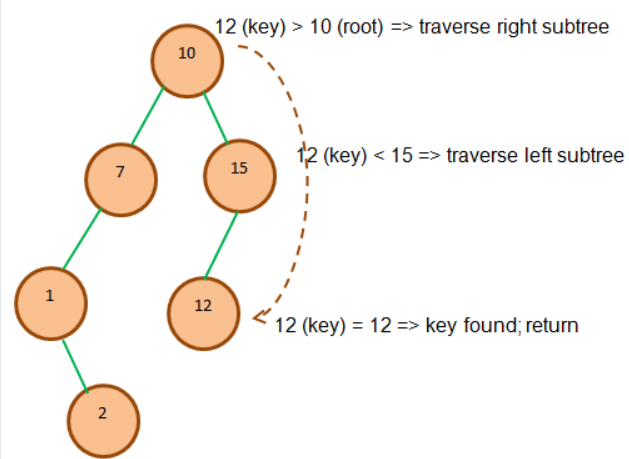
At the time of insertion of nodes, the decision about the position of the node is made. These properties help solve a lot of algorithmic challenges, and as such was designed for that purpose. Binary search trees support all operations that can be performed on binary trees, allowing some of the tasks to be done in lesser time.

**Trees Operation:**

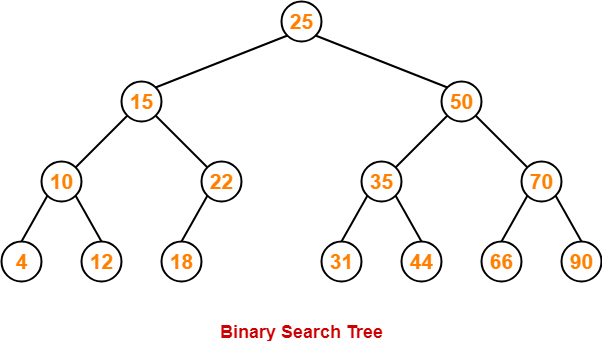
1. **Search Operation:** For searching a given key in the BST,

* Compare the key with the value of root node.
* If the key is present at the root node, then return the root node.
* If the key is greater than the root node value, then recur for the root node’s right subtree.
* If the key is smaller than the root node value, then recur for the root node’s left subtree.

Example



Consider key = 45 has to be searched in the given BST-



* We start our search from the root node 25.
* As 45 > 25, so we search in 25’s right subtree.
* As 45 < 50, so we search in 50’s left subtree.
* As 45 > 35, so we search in 35’s right subtree.
* As 45 > 44, so we search in 44’s right subtree but 44 has no subtrees.
* So, we conclude that 45 is not present in the above BST.

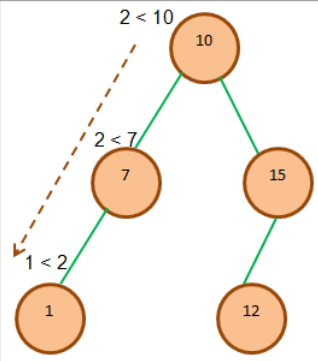
1. **Insertion Operation**

The insertion of a new key always takes place as the child of some leaf node.

**Steps for inserting an element.**

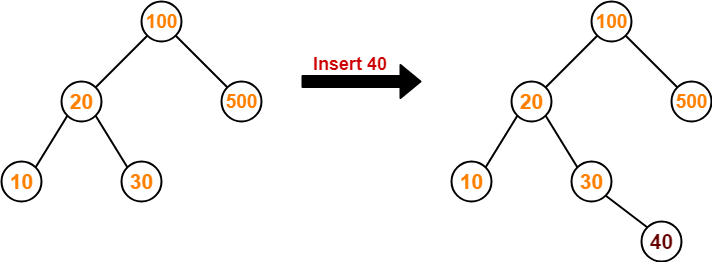
* Start from the root.
* Compare the element to be inserted with the root node. If it is less than root, then traverse the left subtree or traverse the right subtree.
* Traverse the subtree till the end of the desired subtree. Insert the node in the appropriate subtree as a leaf node.

**Example-**



Insert 2

Consider the following example where key = 40 is inserted in the given BST-



* We start searching for value 40 from the root node 100.
* As 40 < 100, so we search in 100’s left subtree.
* As 40 > 20, so we search in 20’s right subtree.
* As 40 > 30, so we add 40 to 30’s right subtree.

1. **Delete Operation:**

When we delete a node from the BST, then there are three possibilities as discussed below:

 Node delete\_Recursive(Node root, **int** key)  {

        //tree is empty

**if** (root == **null**)  **return** root;

//traverse the tree

**if** (key < root.key)     //traverse left subtree

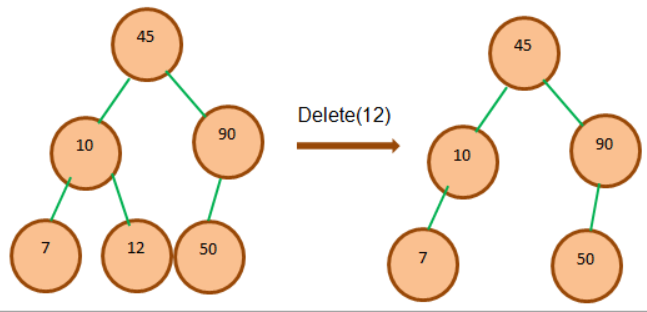
            root.left = delete\_Recursive(root.left, key);

**else** **if** (key > root.key)  //traverse right subtree

            root.right = delete\_Recursive(root.right, key);

**Node Is A Leaf Node**

If a node to be deleted is a leaf node, then we can directly delete this node as it has no child nodes. This is shown in the below image.



**Node Has Only One Child**

When we need to delete the node that has one child, then we copy the value of the child in the node and then delete the child.

Diagram

Description automatically generated

**if** (root.left == **null**)

**return** root.right;

**else** **if** (root.right == **null**)

**return** root.left;

**Node Has Two Children**

If the node has two children, it becomes a little tricky. We need to find the node which has the smallest value in the right subtree (among the elements that have a greater value than the node to be deleted) for the node and use that to replace the deleted node. we replace the node with the inorder (left-root-right) successor of the node or simply said the minimum node in the right subtree if the right subtree of the node is not empty.

Note that the smallest value in the right subtree is the node that comes immediately after the node to be deleted.

For Left subtree: find largest value

For Right subtree: find smallest value

// node has two children;

            //get inorder successor (min value in the right subtree)

            root.key = minValue(root.right);

            // Delete the inorder successor

            root.right = delete\_Recursive(root.right, root.key);

        }

**return** root;

    }

**int** minValue(Node root)  {

        //initially minval = root

**int** minval = root.key;

        //find minval

**while** (root.left != **null**)  {

            minval = root.left.key;

            root = root.left;

        }

**return** minval;

    }

Diagram

Description automatically generated

For Right Subtree:

   //recursive delete function

    Node delete\_Recursive(Node root, **int** key)  {

        //tree is empty

**if** (root == **null**)  **return** root;

        //traverse the tree

**if** (key < root.key)     //traverse left subtree

            root.left = delete\_Recursive(root.left, key);

**else** **if** (key > root.key)  //traverse right subtree

            root.right = delete\_Recursive(root.right, key);

**else**  {

            // node contains only one child

**if** (root.left == **null**)

**return** root.right;

**else** **if** (root.right == **null**)

**return** root.left;

            // node has two children;

            //get inorder successor (min value in the right subtree)

            root.key = minValue(root.right);

            // Delete the inorder successor

            root.right = delete\_Recursive(root.right, root.key);

        }

**return** root;

    }

**int** minValue(Node root)  {

        //initially minval = root

**int** minval = root.key;

        //find minval

**while** (root.left != **null**)  {

            minval = root.left.key;

            root = root.left;

        }

**return** minval;

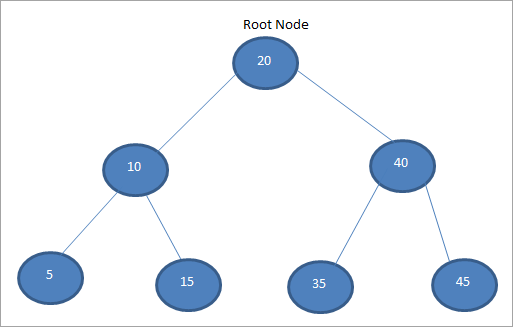
    }

Try left subtree coding, in which you have

#### **3.3.4. Binary Search Tree**

The binary tree that is ordered is called the binary search tree. In a binary search tree, the nodes to the left are less than the root node while the nodes to the right are greater than or equal to the root node.

**An example of a binary search tree is shown below.**

[](https://cdn.softwaretestinghelp.com/wp-content/qa/uploads/2019/08/6.-Binary-Search-Tree.png)

In the above figure, we can see that the left nodes are all less than 20 which is the root element. The right nodes, on the other hand, are greater than the root node. The binary search tree is used in searching and sorting techniques.

## **3.4.** **Implementation of a Binary tree**

class Node

{

int key;

Node left, right;

public Node(int item)

{

key = item;

left = right = null;

}

}

// A Java program to introduce Binary Tree

class BinaryTree

{ // Root of Binary Tree

Node root;

// Constructors

BinaryTree(int key)

{

root = new Node(key);

}

BinaryTree()

{

root = null;

}

public static void main(String[] args)

{

BinaryTree tree = new BinaryTree();

/\*create root\*/

tree.root = new Node(1);

/\* following is the tree after above statement

1

/ \

null null \*/

tree.root.left = new Node(2);

tree.root.right = new Node(3);

tree.root.left.left = new Node(4);

}

}

### 3.5. Insert value in Binary Search Tree(BST)

Inserting a value in the correct position is similar to searching because we try to maintain the rule that left subtree is lesser than root and right subtree is larger than root.

We keep going to either right subtree or left subtree depending on the value and when we reach a point left or right subtree is null, we put the new node there.

Algorithm:

1. If node == NULL
2. return createNode(data)
3. if (data < node->data)
4. node->left = insert(node->left, data);
5. else if (data > node->data)
6. node->right = insert(node->right, data);
7. return node.

### 3.6. Tree Traversal Techniques

We have seen linear data structures like arrays, linked lists, stacks, queues, etc. All these data structures have a common traversing technique that traverses the structure only in one way i.e. linearly. But in the case of trees, we have different traversal techniques as listed below:

1. **In order:** In this traversal technique, we traverse the left subtree first till there are no more nodes in the left subtree. After this, we visit the root node and then proceed to traverse the right subtree until there are no more nodes to traverse in the right subtree. Thus the order of inOrder traversal is left->root->right.
2. **Pre-order:** For preorder traversal technique, we process the root node first, then we traverse the entire left subtree and finally, we traverse the right subtree. Hence the order of preorder traversal is root->left->right.
3. **Post-order:** In the post-order traversal technique, we traverse the left subtree, then the right subtree and finally the root node. The order of traversal for the postorder technique is left->right->root.

**Tree Traversal Algorithm:**

If n is the root node and ‘l’ and ’r’ are left and right nodes of the tree respectively, then the tree traversal algorithms are as follows:

**In order (lnr) algorithm:**

1. Traverse left subtree using inOrder(left- Subtree).
2. Visit the root node(n).
3. Traverse right subtree using inOrder(right- subtree).

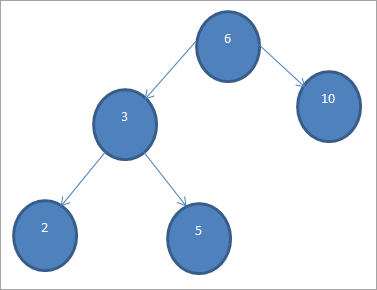
**Preorder (nlr) algorithm:**

1. Visit the root node(n).
2. Traverse left subtree using preorder(left-subtree).
3. Traverse the right subtree using preorder(right-subtree).

**Postorder (lrn) algorithm:**

1. Traverse left subtree using postOrder(left-subtree).
2. Traverse the right subtree using postOrder(right-subtree).
3. Visit the root node(n).

From the above algorithms of traversal techniques, we see that the techniques can be applied to a given tree in a recursive fashion to get the desired result. Consider the following tree.

[](https://cdn.softwaretestinghelp.com/wp-content/qa/uploads/2019/08/8.-Postorder-lrn-algorithm.png)

**Using the above traversal techniques, the traversal sequence for the above tree is given below:**

1. InOrder traversal : 2 3 5 6 10
2. PreOrder traversal : 6 3 2 5 10
3. PostOrder traversal: 2 5 3 10 6

3. 3. **Depth First Search (DFS)**

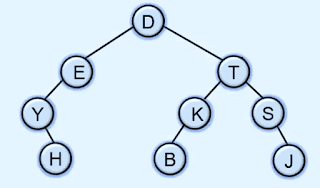
Depth-first search (DFS) is yet another technique used to traverse a tree or a graph. DFS starts with a root node or a start node and then explores the adjacent nodes of the current node by going deeper into the graph or a tree. This means that in DFS the nodes are explored depth-wise until a node with no children is encountered.

### 3.3.1. DFS Algorithm

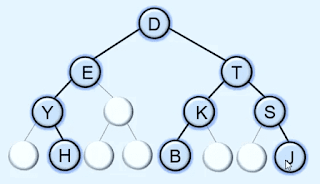
* **Step 1:** Insert the root node or starting node of a tree or a graph in the stack.
* **Step 2:** Pop the top item from the stack and add it to the visited list.
* **Step 3:** Find all the adjacent nodes of the node marked visited and add the ones that are not yet visited, to the stack.
* **Step 4**: Repeat steps 2 and 3 until the stack is empty.

# **Array representation of Binary tree**

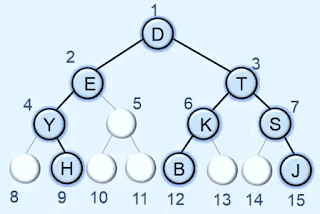
Torepresent a binary tree using array first we need to convert a **binary tree into a full binary tree.** and then we give the number to each node and store it into their respective locations.  
let's take an example to understand how to represent a binary tree using an array.  
to do this first we need to convert a binary tree into a full binary tree.

[](https://4.bp.blogspot.com/-Ll9mBUvZUBQ/XtM755oPpdI/AAAAAAAAM7s/9PwlNosx-Uk59SWFsWwPU5eEoAMOJoGBwCK4BGAYYCw/s1600/array-representation-of-binary-tree.png)

here in the above example to convert this binary tree into a full binary tree we need to add nodes that don't have child nodes till the last level of the tree.

[](https://4.bp.blogspot.com/-LgFaHUjdW7w/XtM83LduqyI/AAAAAAAAM74/BAbs2PUplIoSHGjV7hiQOPQJfH614Pi2ACK4BGAYYCw/s1600/array-representation-of-binary-tree.png)

So now the tree becomes a full binary tree. after that to represent it using an array we need to give the numbers to each and every node but level by level.

[](https://2.bp.blogspot.com/-1Ro93-pzJsU/XtM-Kw5xWeI/AAAAAAAAM8E/VnyRsZ9KNbUzN6QynyyeKbRzPZgqYFWTQCK4BGAYYCw/s1600/array-representation-of-binary-tree.png)

after giving the number to each and every node now we need to create an array of size 15 + 1.

[sequential Array representation of Binary tree in data structures](https://4.bp.blogspot.com/-Q2fwWSsbJCI/XtNDzGcujpI/AAAAAAAAM8Q/WNCgCRbxYcs_vL9hH9iDkuHLRQ24LAKEwCK4BGAYYCw/s1600/array-representation-of-binary-tree.png)

after that store each one node in array in their respective index points. like D has number 1 then we store it in the array at index 1 and E has number 2 then we store it at index 2 in the array.

[sequential Array representation of Binary tree in data structures](https://3.bp.blogspot.com/-R-t70FYu_Qw/XtNEnn0HekI/AAAAAAAAM8c/q1TFw3uBfIs2jDMnw1LC3FgJYfLRvuK8QCK4BGAYYCw/s1600/array-representation-of-binary-tree.png)

so this is the array representation of a binary tree.

The root is always stored at index 1 in the array.  
  
**Note: If any node is stored at K position then the left child of a node is stored at index 2k and the right child is stored at index 2K + 1 and the parent of a node is stored at floor(K/2) index.**

**Code**

**Tree Travsersal Code**

**// Binary Tree in C++**

// Binary Tree in C++

#include <stdlib.h>

#include <iostream>

using namespace std;

struct node {

int data;

struct node \*left;

struct node \*right;

};

// New node creation

struct node \*newNode(int data) {

struct node \*node = (struct node \*)malloc(sizeof(struct node));

node->data = data;

node->left = NULL;

node->right = NULL;

return (node);

}

// Traverse Preorder

void traversePreOrder(struct node \*temp) {

if (temp != NULL) {

cout << " " << temp->data;

traversePreOrder(temp->left);

traversePreOrder(temp->right);

}

}

// Traverse Inorder

void traverseInOrder(struct node \*temp) {

if (temp != NULL) {

traverseInOrder(temp->left);

cout << " " << temp->data;

traverseInOrder(temp->right);

}

}

// Traverse Postorder

void traversePostOrder(struct node \*temp) {

if (temp != NULL) {

traversePostOrder(temp->left);

traversePostOrder(temp->right);

cout << " " << temp->data;

}

}

int main() {

struct node \*root = newNode(1);

root->left = newNode(2);

root->right = newNode(3);

root->left->left = newNode(4);

cout << "preorder traversal: ";

traversePreOrder(root);

cout << "\nInorder traversal: ";

traverseInOrder(root);

cout << "\nPostorder traversal: ";

traversePostOrder(root);

}

**Evaluation criteria**

The evaluation criteria for this lab will be based on the completion of the following tasks. Each task is assigned the marks percentage which will be evaluated by the instructor in the lab whether the student has finished the complete/partial task(s).

Evaluation of the Lab

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr. No.** | **Task No** | **Description** | **Marks** |
| 1 | Task 1 | Lab Task 1 | 10 |
| 2 | Task 2 | Lab Task 2 | 10 |
| 3 | Task 3 | Lab Task 3 | 5 |
| 4 | Task 4 | Lab Task 4 | 5 |

**Further Readin****g and Books**

Uploaded on Moellim