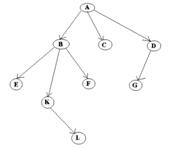
|  |
| --- |
| **NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES**  **CS 218–DATA STRUCTURES LAB** |
| **Instructors:** Mubashra |

**Trees :**

A tree is a collection of nodes connected by directed (or undirected) edges. A tree is a nonlinear data structure, compared to arrays, linked lists, stacks and queues which are linear data structures. A tree can be empty with no nodes or a tree is a structure consisting of one node called the **root** and zero or one or more sub trees. A tree has following general properties:

* One node is distinguished as a root;
* Every node (exclude a root) is connected by a directed edge from exactly one other node; A direction is: parent -> children



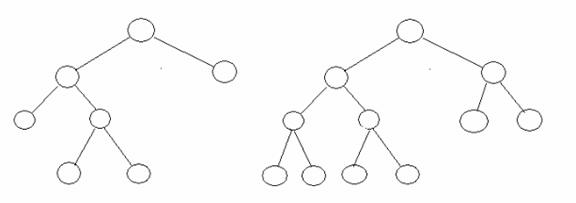
A is a parent of B, C, D,   
B is called a child of A.   
on the other hand, B is a parent of E, F, K

* In the above picture, the root has 3 sub-trees.
* Each node can have arbitrary number of children.
* Nodes with no children are called leaves, or external nodes.
* In the above picture, C, E, F, L, G are leaves.
* Nodes, which are not leaves, are called internal nodes. Internal nodes have at least one child.
* Nodes with the same parent are called siblings.
* In the picture, B, C, D is called siblings.
* The depth of a node is the number of edges from the root to the node.
* The depth of K is 2.
* The height of a node is the number of edges from the node to the deepest leaf.
* The height of B is 2.
* The height of a tree is a height of a root.

**Binary Trees**

A binary tree in which each node has exactly zero or two children is called a full binary tree. In a full tree, there are no nodes with exactly one child.

A complete binary tree is a tree, which is completely filled, with the possible exception of the bottom level, which is filled from left to right. A complete binary tree of the height h has between 2h and 2(h+1)-1 nodes. Here are some examples:



**Binary Search Trees**

Binary search tree (BST) or a lexicographic tree is a binary tree data structure which has the following binary search tree properties:

* Each node has a value.
* The key value of the left child of a node is less than to the parent's key value.
* The key value of the right child of a node is greater than (or equal) to the parent's key value.
* And these properties holds true for every node in the tree.

If a BST allows duplicate values, then it represents a multi-set. This kind of tree uses non-strict inequalities (<=, >=). Everything in the left sub-tree of a node is strictly less than the value of the node, but everything in the right sub-tree is either greater than or equal to the value of the node.

If a BST doesn't allow duplicate values, then the tree represents a set with unique values, like the mathematical set. Trees without duplicate values use strict inequalities, meaning that the left sub-tree of a node only contains nodes with values that are less than the value of the node, and the right sub-tree only contains values that are greater.

The choice of storing equal values in the right sub-tree only is arbitrary; the left would work just as well. One can also permit non-strict equality in both sides. This allows a tree containing many duplicate values to be balanced better, but it makes searching more complex.

**Traversals**

Stepping through the items of a tree, by means of the connections between parents and children, is called walking the tree and the action is a walk of the tree (traverse). Often, an operation might be performed when a pointer arrives at a particular node (visiting the node – for example, printing the value/s that the node contains).

The binary search tree property allows us to obtain all the keys in a binary search tree in a sorted order by a simple traversing algorithm, called an in order tree walk, that traverses the left sub tree of the root in in order traverse, then accessing the root node itself, then traversing in in-order the right sub tree of the root node.

The tree may also be traversed in preorder or post order traversals. By first accessing the root, and then the left and the right sub-tree or the right and then the left sub-tree to be traversed in preorder. And the opposite for the post order.

The algorithms are described below, with Node initialized to the tree’s root.

• Preorder Traversal  
 1. Visit Node.  
 2. Traverse Node’s left sub-tree.  
 3. Traverse Node’s right sub-tree.  
  
• In-order Traversal  
 1. Traverse Node’s left sub-tree.  
 2. Visit Node.  
 3. Traverse Node’s right sub-tree   
  
• Post-order Traversal  
 1. Traverse Node’s left sub-tree.  
 2. Traverse Node’s right sub-tree.  
 3. Visit Node.

**Insertion**

Insertion begins as a search would begin; if the root is not equal to the value, we search the left or right sub-trees as before. Eventually, we will reach a leaf and add the value as its right or left child, depending on the node's value.

**Step 1:**

**Create class Nodes**

class Node {

private:

int key;

string name;

Node leftChild;

Node rightChild;

public:

Node(int key, string name) {

this.key = key;

this.name = name;

}

string toString() {

return cout<<name<< " has the key " <<key<<endl;

}

};

Step 2:

Create class BinaryTree and create a function which add nodes in BST

class BinaryTree {

private:

Node root;

public:

void addNode(int key, string name) {

// Create a new Node and initialize it

Node newNode = new Node(key, name);

// If there is no root this becomes root

if (root == NULL) {

root = newNode;

} else {

// Set root as the Node we will start

// with as we traverse the tree

Node focusNode = root;

// Future parent for our new Node

Node parent;

while (true) {

// root is the top parent so we start

// there

parent = focusNode;

// Check if the new node should go on

// the left side of the parent node

if (key < focusNode.key) {

// Switch focus to the left child

focusNode = focusNode.leftChild;

// If the left child has no children

if (focusNode == NULL) {

// then place the new node on the left of it

parent.leftChild = newNode;

return; // All Done

}

} else { // If we get here put the node on the right

focusNode = focusNode.rightChild;

// If the right child has no children

if (focusNode == NULL) {

// then place the new node on the right of it

parent.rightChild = newNode;

return; // All Done

}

}

}

}

}

void in\_orderTraverseTree(Node focusNode)//Recursive {

}

void preorderTraverseTree(Node focusNode) //Recursive{

if (focusNode != NULL) {

cout<<focusNode<<” ”;

preorderTraverseTree(focusNode.leftChild);

preorderTraverseTree(focusNode.rightChild);

}

void post\_orderTraverseTree(Node focusNode) //Recursive{

}

void in\_orderTraverseTreeNR(Node focusNode)//Non Recursive {

}

void preorderTraverseTreeNR(Node focusNode)// Non Recursive

{

}

void post\_orderTraverseTreeNR(Node focusNode) // Non Recursive

{

}

Create class having main method

int main() {

BinaryTree theTree = new BinaryTree();

theTree.addNode(50, "Boss");

theTree.addNode(25, "Vice President");

theTree.addNode(15, "Office Manager");

theTree.addNode(30, "Secretary");

theTree.addNode(75, "Sales Manager");

theTree.addNode(85, "Salesman 1");

// Different ways to traverse binary trees

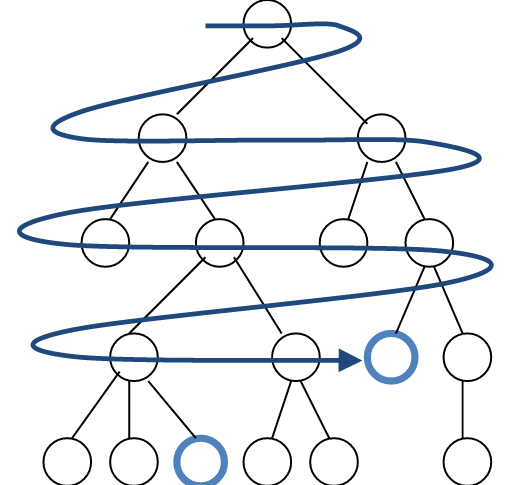
theTree.in\_orderTraverseTree(theTree.root);

theTree.preorderTraverseTree(theTree.root);

theTree.post\_orderTraverseTree(theTree.root);

}

**Breadth-first searching:**

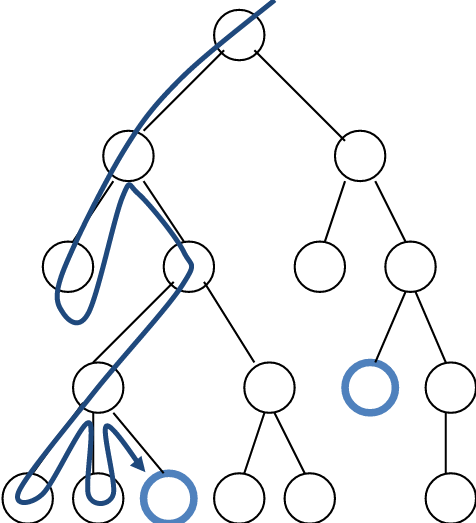


* A breadth-first search (BFS) explores nodes nearest the root before exploring nodes further away
* For example, after searching A, then B, then C, the search proceeds with D, E, F, G
* Node are explored in the order A B C D E F G H I J K L M N O P Q
* J will be found before N

**How to do breadth-first searching:**

Put the root node on a queue;  
while (queue is not empty) {  
 remove a node from the queue;  
 if (node is a goal node) return success;  
 put all children of node onto the queue;  
}  
return failure;

**Depth-first searching**



* A depth-first search (DFS) explores a path all the way to a leaf before backtracking and exploring another path
* For example, after searching A, then B, then D, the search backtracks and tries another path from B
* Node are explored in the order A B D E H L M N I O P C F G J K Q
* N will be found before J

**How to do depth-first searching:**

Put the root node on a stack;  
while (stack is not empty) {  
 remove a node from the stack;  
 if (node is a goal node) return success;  
 put all children of node onto the stack;  
}  
return failure;

**Lab Task:**

* Implement the following functions for BST.
  + Insertion
  + Deletion
  + Searching
  + Height
  + No of Leaf Nodes.
* Implement Breadth first and Depth first searching.
* Implement all pre-order, post-order, in-order traversal functions (recursive and non recursive both).

**Home Task:**

Implement all the functions of B-tree.

1. Insertion
2. Deletion
3. Searching
4. Sum of all Nodes