NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES
CS 201–DATA STRUCTURES LAB
Lab Session 13

Instructors: Aqsa Zahid, Mubashra Fayyaz

Trees

- Tree is a hierarchical data structure which stores the information naturally in the form of hierarchy style.
- Tree is one of the most powerful and advanced data structures.
- It is a non-linear data structure compared to arrays, linked lists, stack and queue.
- It represents the nodes connected by edges.

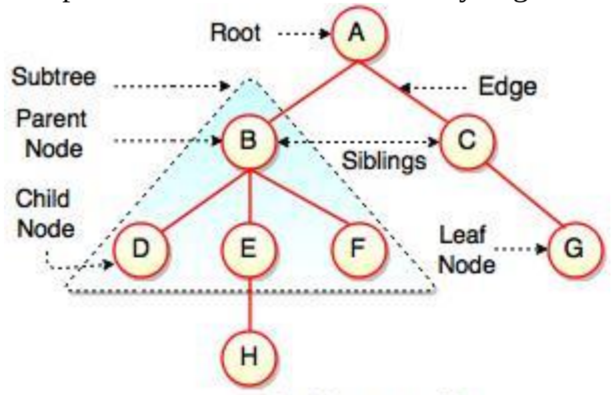


Fig. Structure of Tree

The above figure represents structure of a tree. Tree has 2 subtrees.
A is a parent of B and C.
B is called a child of A and also parent of D, E, F.
Tree is a collection of elements called Nodes, where each node can have arbitrary number of children.

Field	Description
Root	Root is a special node in a tree. The entire tree is referenced through it. It does not have a parent.
Parent Node	Parent node is an immediate predecessor of a node.
Child Node	All immediate successors of a node are its children.
Siblings	Nodes with the same parent are called Siblings.
Path	Path is a number of successive edges from source node to destination node.
Height of Node	Height of a node represents the number of edges on the longest path between that node and a leaf.
Height of Tree	Height of tree represents the height of its root node.

Depth of Node	Depth of a node represents the number of edges from the tree's root node to the node.
Degree of Node	Degree of a node represents a number of children of a node.
Edge	Edge is a connection between one node to another. It is a line between two nodes or a node and a leaf.

In the above figure, D, F, H, G are **leaves**. B and C are **siblings**. Each node excluding a root is connected by a direct edge from exactly one other node
parent → children.

Levels of a node

Levels of a node represents the number of connections between the node and the root. It represents generation of a node. If the root node is at level 0, its next node is at level 1, its grand child is at level 2 and so on. Levels of a node can be shown as follows:

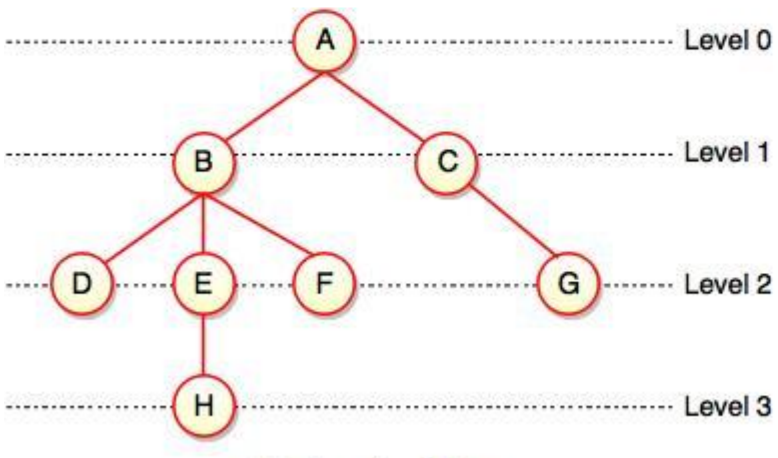


Fig. Levels of Tree

Note:

- If node has no children, it is called **Leaves** or **External Nodes.**

- Nodes which are not leaves, are called **Internal Nodes**. Internal nodes have at least one child.

- A tree can be empty with no nodes or a tree consists of one node called the **Root**.

Height of a Node

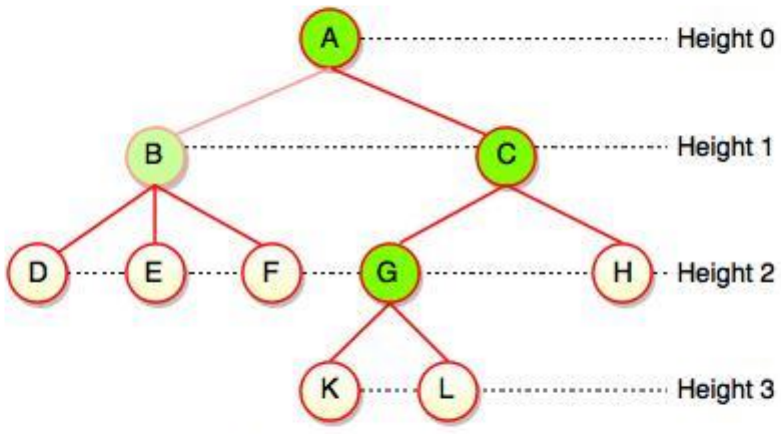


Fig. Height of a Node

height of a node is a number of edges on the longest path between that node and a leaf. Each node has height.

In the above figure, A, B, C, D can have height. Leaf cannot have height as there will be no path starting from a leaf. Node A's height is the number of edges of the path to K not to D. And its height is 3.

Note:

- Height of a node defines the longest path from the node to a leaf.
- Path can only be downward.

Depth of a Node

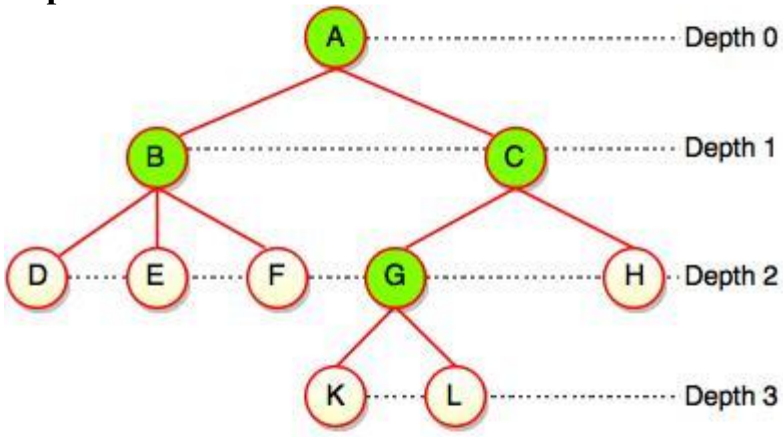


Fig. Depth of a Node

While talking about the height, it locates a node at bottom where for depth, it is located at top which is root level and therefore we call it depth of a node.

In the above figure, Node G's depth is 2. In depth of a node, we just count how many edges between the targeting node & the root and ignoring the directions.

Note: Depth of the root is 0.

Advantages of Tree

- Tree reflects structural relationships in the data.
- It is used to represent hierarchies.
- It provides an efficient insertion and searching operations.
- Trees are flexible. It allows to move subtrees around with minimum effort.

Binary Tree

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

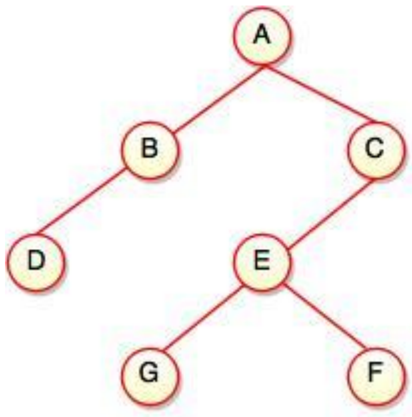


Fig. Binary Tree

The above tree represents binary tree in which node A has two children B and C. Each children have one child namely D and E respectively.

Representation of Binary Tree using Array

Binary tree using array represents a node which is numbered sequentially level by level from left to right. Even empty nodes are numbered.

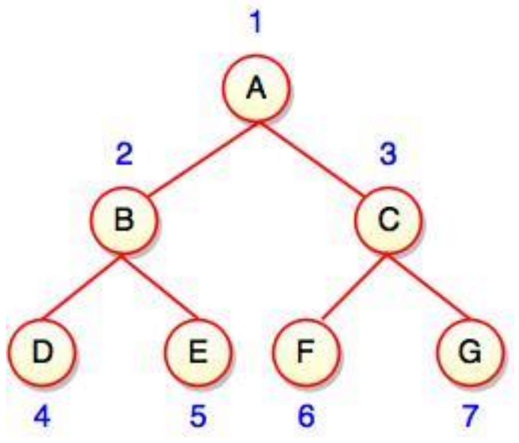


Fig. Binary Tree using Array

Array index is a value in tree nodes and array value gives to the parent node of that particular index or node. Value of the root node index is always -1 as there is no parent for root. When the data item of the tree is sorted in an array, the number appearing against the node will work as indexes of the node in an array.

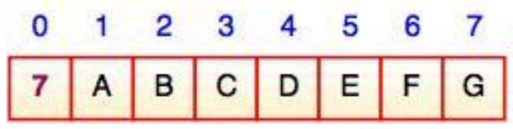


Fig. Location Number of an Array in a Tree

Location number of an array is used to store the size of the tree. The first index of an array that is '0', stores the total number of nodes. All nodes are numbered from left to right level by level from top to bottom. In a tree, each node having an index i is put into the array as its ith element.

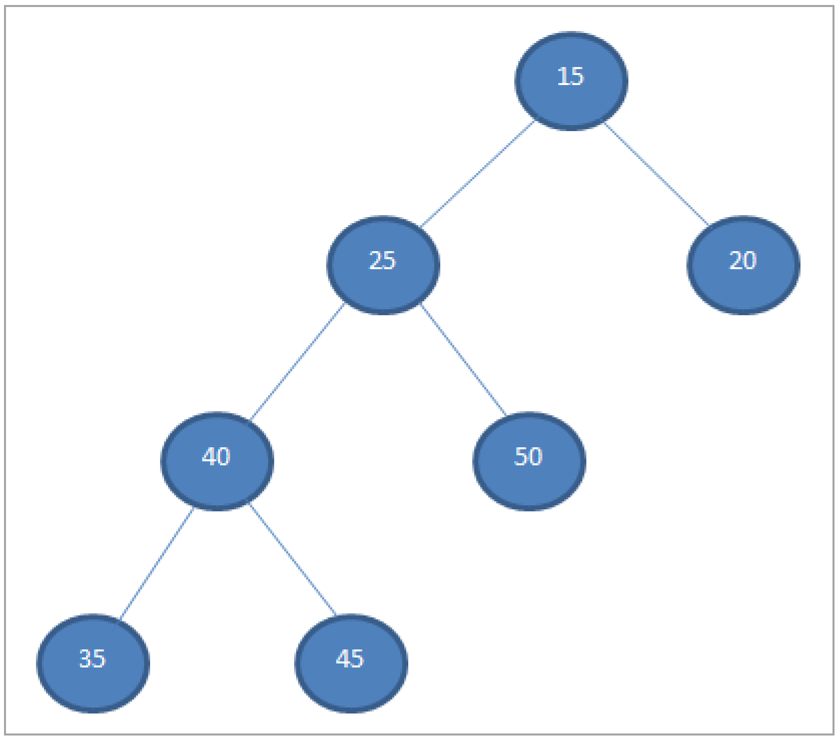
The above figure shows how a binary tree is represented as an array. Value '7' is the total number of nodes. If any node does not have any of its child, null value is stored at the corresponding index of the array.

Types Of Binary Tree

Following are the most common types of binary trees.

1. Full Binary Tree

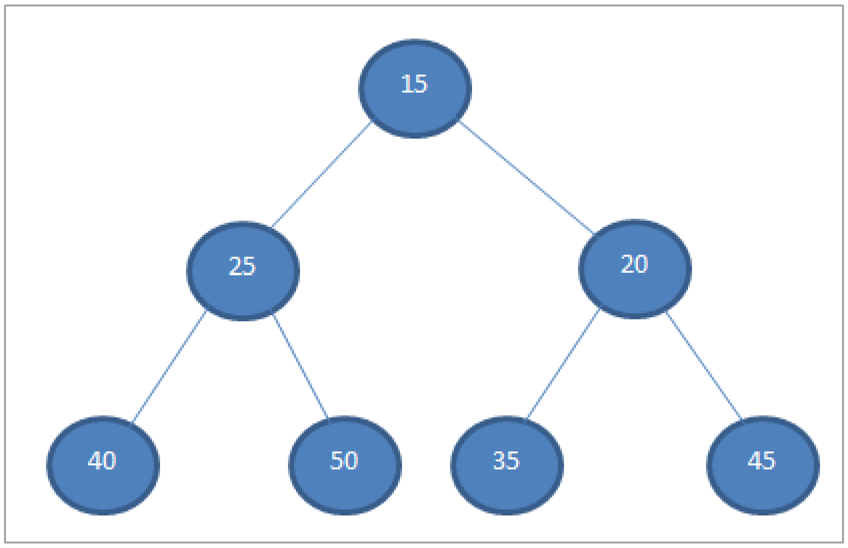
A binary tree in which every node has 0 or 2 children is termed as a full binary tree.



Above shown is a full binary tree in which we can see that all its nodes except the leaf nodes have two children. If L is the number of leaf nodes and ‘l’ is the number of internal or non-leaf nodes, then for a full binary tree, L = l + 1.

2. Complete Binary Tree

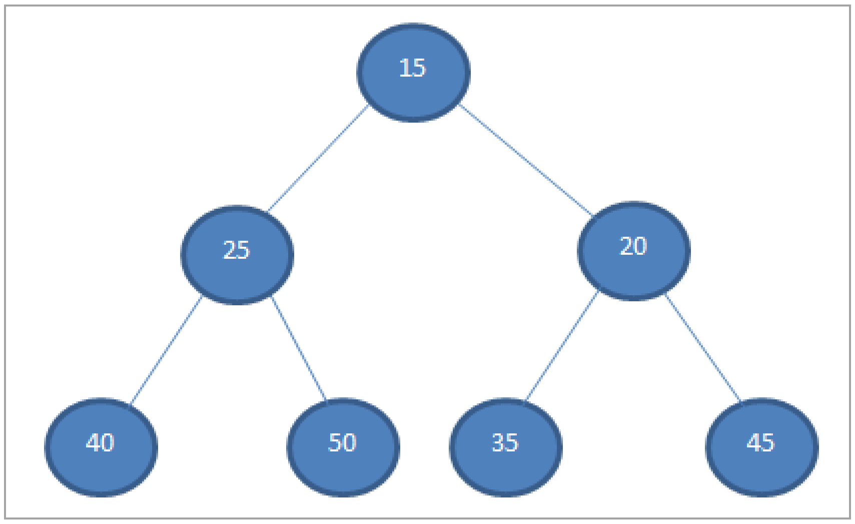
A complete binary tree has all the levels filled except for the last level and the last level has all its nodes as much as to the left.



The tree shown above is a complete binary tree. A typical example of a complete binary tree is a binary heap which we will discuss in the later tutorials.

3. Perfect Binary Tree

A binary tree is termed perfect when all its internal nodes have two children and all the leaf nodes are at the same level.

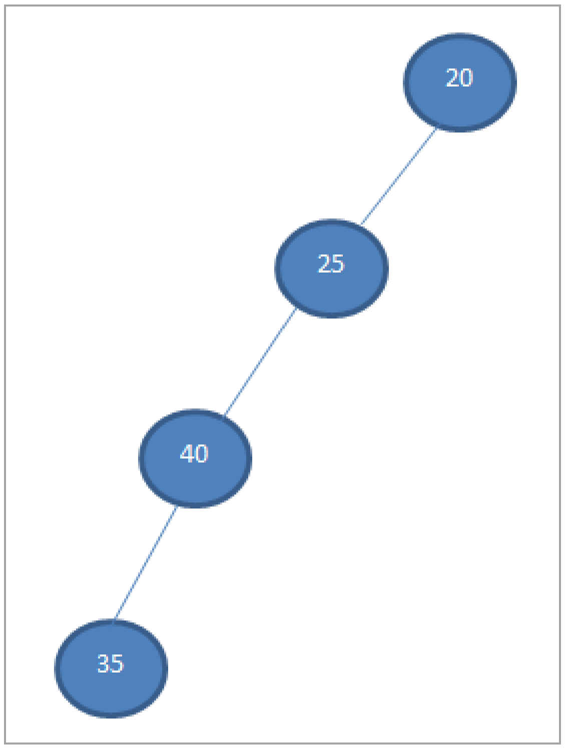


A binary tree example shown above is a perfect binary tree as each of its nodes has two children and all the leaf nodes are at the same level.

A perfect binary tree of height h has $2^h - 1$ number of nodes.

4. A Degenerate Tree

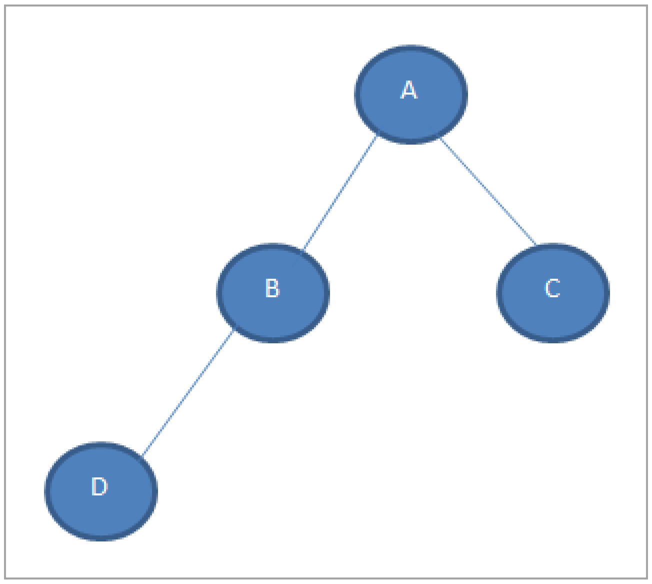
A binary tree where each internal node has only one child is called a degenerate tree.



The tree shown above is a degenerate tree. As far as the performance of this tree is concerned, the degenerate trees are the same as linked-lists.

5. Balanced Binary Tree

A binary tree in which the depth of the two subtrees of every node never differs by more than 1 is called a balanced binary tree.

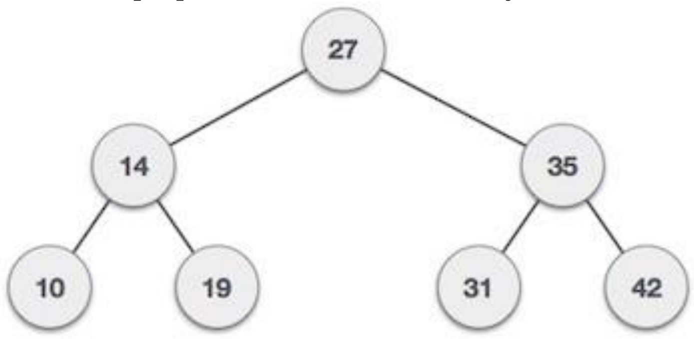


The binary tree shown above is a balanced binary tree as the depth of the two subtrees of every node is not more than 1.

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);
}
```

Lab Task:

- Implement Binary tree, binary search tree create basic functions of insertion, search and all pre-order, post-order, in-order traversal functions (recursive and non recursive both).