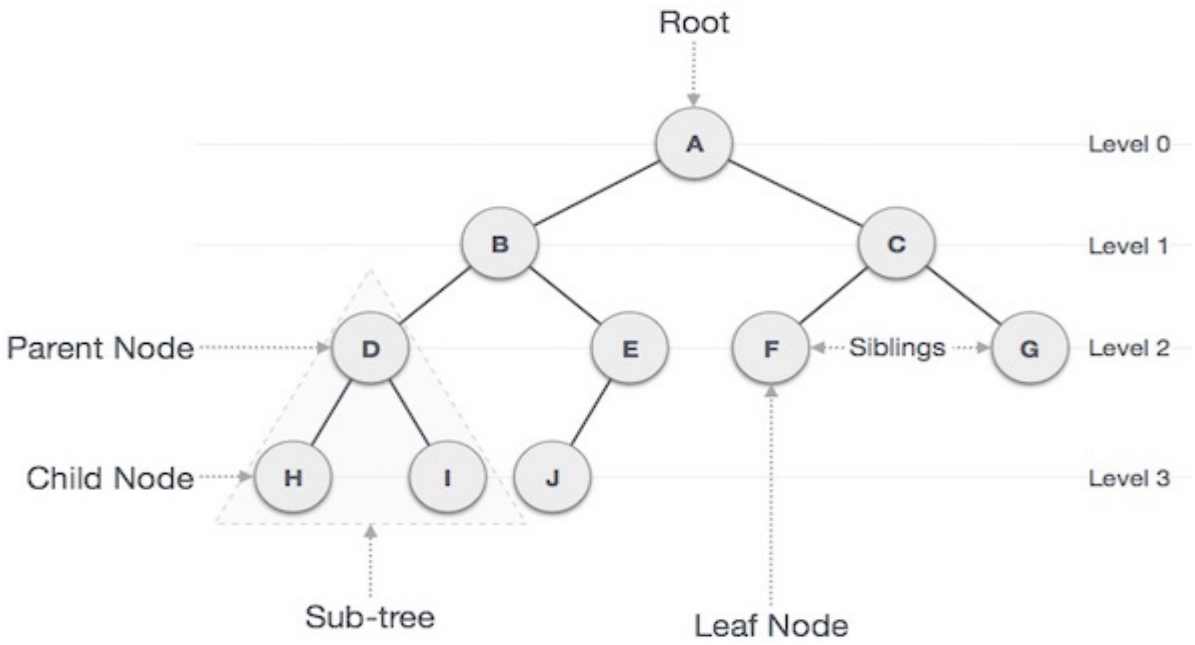
TREES

There are many basic data structures that can be used to solve application problems. Array is a good static data structure that can be accessed randomly and is fairly easy to implement. Linked Lists on the other hand is dynamic and is ideal for application that requires frequent operations such as add, delete, and update. One drawback of linked list is that data access is sequential. Then there are other specialized data structures like, stacks and queues that allows us to solve complicated problems (eg: Maze traversal) using these restricted data structures. One other data structure is the hash table that allows users to program applications that require frequent search and updates. They can be done in O(1) in a hash table.

One of the disadvantages of using an array or linked list to store data is the time necessary to search for an item. Since both the arrays and Linked Lists are linear structures the time required to search a "linear" list is proportional to the size of the data set. For example, if the size of the data set is n, then the number of comparisons needed to find (or not find) an item may be as bad as some multiple of n. So imagine doing the search on a linked list (or array) with $n = 10^6$ nodes. Even on a machine that can do million comparisons per second, searching for m items will take roughly m seconds. This not acceptable in today's world where speed at which we complete operations is extremely important. Time is money. Therefore it seems that better (more efficient) data structures are needed to store and search data.

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 subtrees. 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



***Binary Tree**

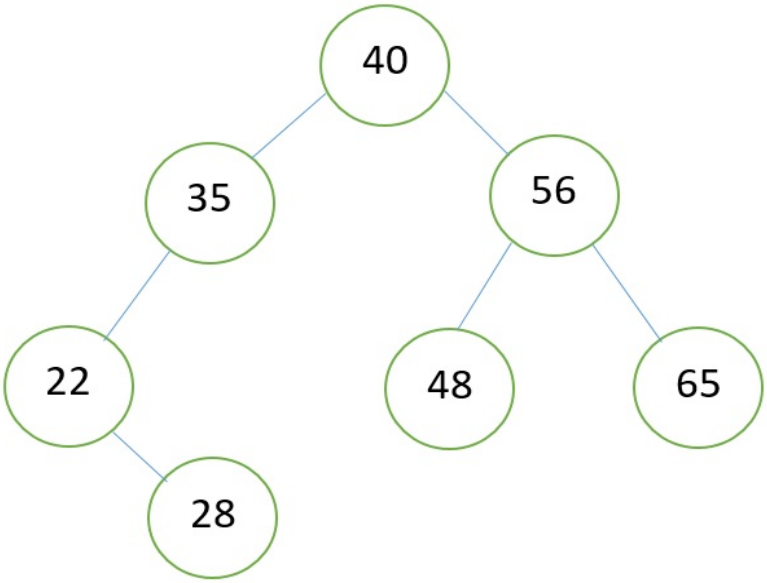
A binary tree is a finite set of nodes that is either empty or consist a root node and two disjoint binary trees called the left subtree and the right subtree.

In other words, a binary tree is a non-linear data structure in which each node has maximum of two child nodes. The tree connections can be called as branches.

According to graph theory binary trees defined here are actually arborescence. A binary tree is also known as old programming term bifurcating arborescence, before the modern computer science terminology prevailed.Binary tree is also known as rooted binary tree because some author uses this term to emphasize the fact that the tree is rooted, but as defined above, a binary tree is always rooted. A binary tree is a special case of an ordered binary tree, where k is 2.

1. Trees are used to represent data in hierarchical form.
2. Binary tree is the one in which each node has maximum of two child- node.
3. The order of binary tree is '2'.
4. Binary tree does not allow duplicate values.
5. While constructing a binary, if an element is less than the value of its parent node, it is placed on the left side of it otherwise right side.
6. A binary tree is shown for the element 40, 56, 35, 48, 22, 65, 28.

Following there is an example of binary search tree:



Advantages of Binary Tree:

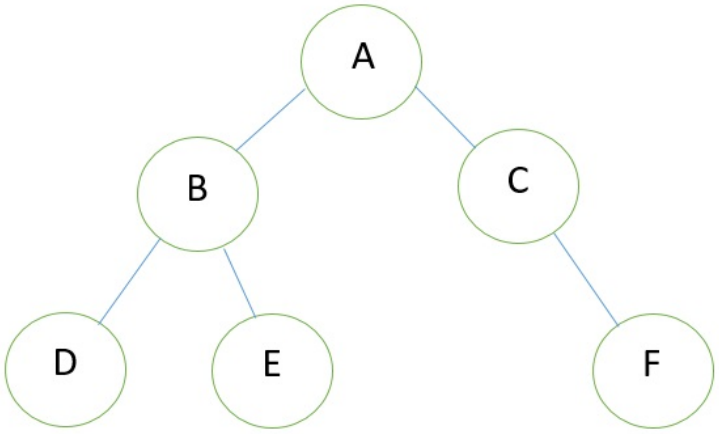
1. Searching in Binary tree become faster.
2. Binary tree provides six traversals.
3. Two of six traversals give sorted order of elements.
4. Maximum and minimum elements can be directly picked up.
5. It is used for graph traversal and to convert an expression to postfix and prefix forms.

1) ****Complete Binary Tree**

A binary tree T is said to be complete binary tree if -

1. All its levels, except possibly except possibly the last, have the maximum number of nodes and
2. All the nodes at the last level appears as far left as possible.

Consider the following tree, which is complete binary tree:



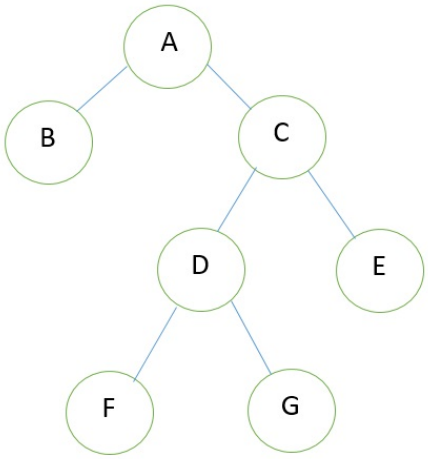
Note: Full binary tree is also called complete binary tree.

IF L is the level of complete binary tree then $2^L - 1$ nodes present in the tree.

2) ****Strictly Binary Tree**

When every non leaf node in a binary tree is filled with left and right subtrees, the tree is called a strictly binary tree.

Following figure shows a strictly binary tree:



In above figure nodes A, C and D provide two nodes each.

3) **Extended Binary Tree**

The binary tree that is extended with zero (no nodes) or left or right node or both the nodes is called an extended binary tree or a 2- tree.

The extended binary tree is shown in figure above. The extended nodes are indicated by square box. Maximum nodes in the binary tree have one child (nodes) shown at either left or right by adding one or more children, the tree can be extended. The extended binary tree is very useful for representation of algebraic expressions. The left and right child nodes denote operands and parent node indicates operator.

4) **Full Binary Tree**

A Binary Tree is full binary tree if and only if -

1. Each non- leaf node has exactly two child nodes.
2. All leaf nodes are at the same level.

Properties of Full Binary Tree

1. A binary tree of height h with no missing node.
2. All leaves are at level h and all other nodes have two children.
3. All the nodes that are at a level less than h have two children each.

*****BST Basic Operations**

The basic operations that can be performed on a binary search tree data structure, are the following −

- Insert − Inserts an element in a tree/create a tree.
- Search − Searches an element in a tree.
- Preorder Traversal − Traverses a tree in a pre-order manner.
- Inorder Traversal − Traverses a tree in an in-order manner.
- Postorder Traversal − Traverses a tree in a post-order manner.

We shall learn creating (inserting into) a tree structure and searching a data item in a tree in this chapter. We shall learn about tree traversing methods in the coming chapter.

****Insert Operation**

The very first insertion creates the tree. Afterwards, whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

*****Algorithm**

```
If root is NULL
   then create root node
return
```

```
If root exists then
  compare the data with node.data

  while until insertion position is located

    If data is greater than node.data
      goto right subtree
    else
      goto left subtree

  endwhile

  insert data

end If
```

*****Search Operation**

Whenever an element is to be searched, start searching from the root node, then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

Algorithm

```
If root.data is equal to search.data
  return root
else
  while data not found

    If data is greater than node.data
      goto right subtree
    else
      goto left subtree

    If data found
      return node
  endwhile

  return data not found

end if
```

****Traversal** is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree −

- In-order Traversal

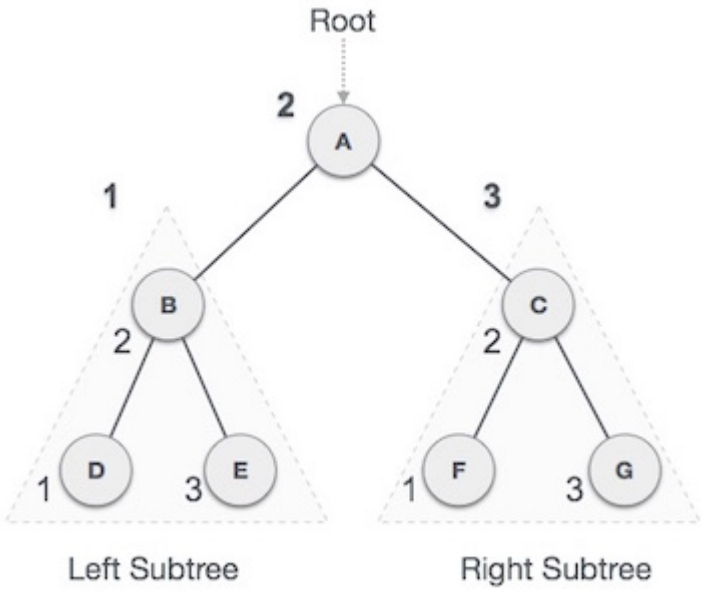
- Pre-order Traversal
- Post-order Traversal

Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

If a binary tree is traversed in-order, the output will produce sorted key values in an ascending order.



We start from A, and following in-order traversal, we move to its left subtree B. B is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be −

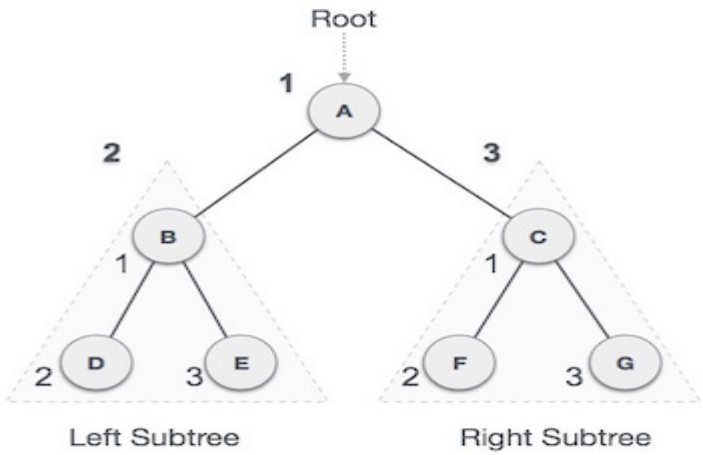
D → B → E → A → F → C → G

Algorithm

Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Visit root node.
Step 3 − Recursively traverse right subtree.

Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



We start from A, and following pre-order traversal, we first visit A itself and then move to its left subtree B. B is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be −

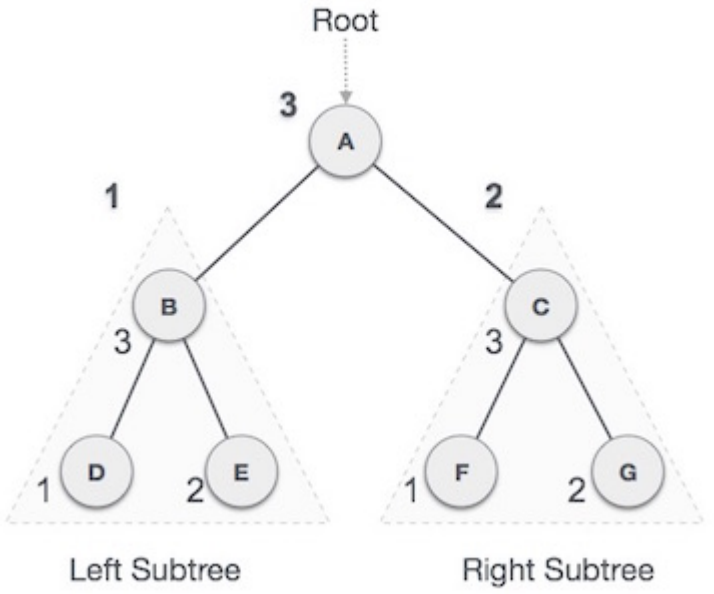
A → B → D → E → C → F → G

Algorithm

Until all nodes are traversed −
Step 1 − Visit root node.
Step 2 − Recursively traverse left subtree.
Step 3 − Recursively traverse right subtree.

Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



We start from A, and following Post-order traversal, we first visit the left subtree B. B is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be −

D → E → B → F → G → C → A

Algorithm

Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Recursively traverse right subtree.
Step 3 − Visit root node.

****A threaded binary tree** is defined as follows:, "A binary tree is threaded by making all right child pointers that would normally be null point to the in-order successor of the node (if it exists), and all left child pointers that would normally be null point to the in-order predecessor of the node."

This definition assumes the traversal order is the same as in-order traversal of the tree. However, pointers can instead (or in addition) be added to tree nodes, rather than replacing linked lists thus defined are also commonly called "threads", and can be used to enable traversal in any order(s) desired. For example, a tree whose nodes represent information about people might be sorted by name, but have extra threads allowing quick traversal in order of birth date, weight, or any other known characteristic.

In computing, a threaded binary tree is a binary tree variant that facilitates traversal in a particular order (often the same order already defined for the tree).

An entire binary sort tree can be easily traversed in order of the main key, but given only a pointer to a node, finding the node which comes next may be slow or impossible. For example, leaf nodes by definition have no descendants, so no other node can be reached given only a pointer to a leaf node -- of course that includes the desired "next" node. A threaded tree adds extra information in some or all nodes, so the "next" node can be found quickly. It can also be traversed without recursion and the extra storage (proportional to the tree's depth) that requires.

******Height balanced /AVL trees***

A height balanced tree is one where there is a bound on the difference between the heights of the subtrees. One of the classic examples of height balanced tree is AVL trees. In AVL trees each node has an attribute associated to it called the balance factor. Balance factor of a node is nothing but the difference between the heights of the subtrees rooted at that particular node. In AVL tree the constraint is that the heights may differ by atmost 1. In other words balance factor of any node may be one of the 3 values namely -1 , 0 or 1.

AVL Rotations

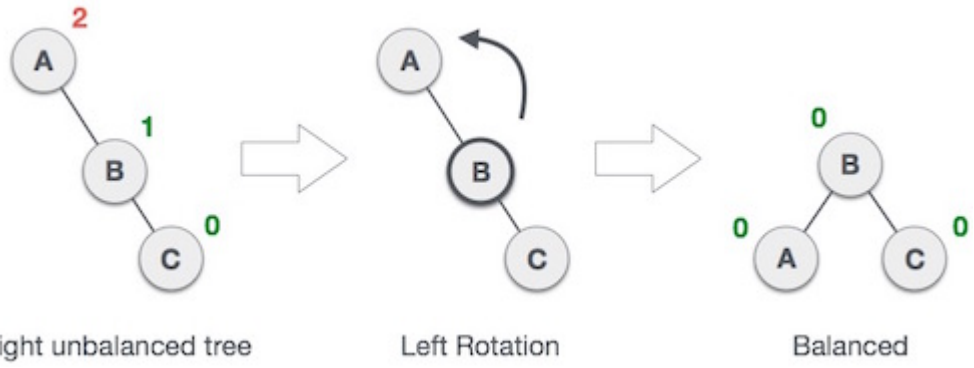
To balance itself, an AVL tree may perform the following four kinds of rotations −

- Left rotation
- Right rotation
- Left-Right rotation
- Right-Left rotation

The first two rotations are single rotations and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2. With this simple tree, let's understand them one by one.

Left Rotation

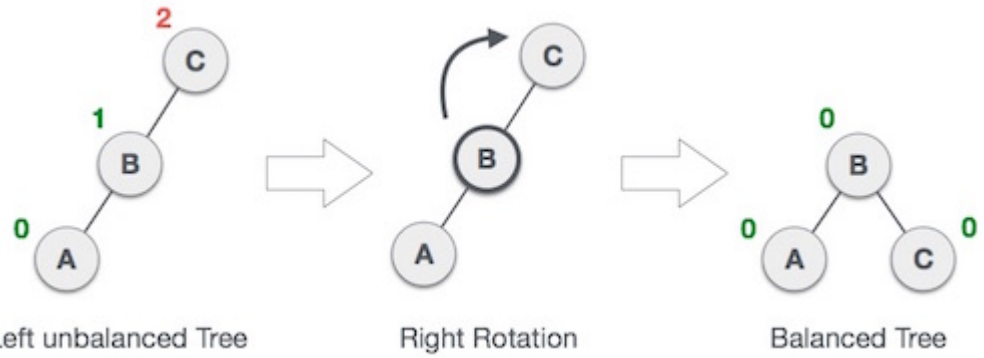
If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation −



In our example, node A has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making A the left-subtree of B.

Right Rotation

AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.



As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

Left-Right Rotation

Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right rotation.

Right-Left Rotation

The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.

*****Insertion :**

- Trace from path of inserted leaf towards the root, and check if the AVL tree property is violated perform rotation if necessary
- For insertion, once we perform (Single or doubles) rotation at a node x, The AVL tree property is already restored. We need not to perform any rotation at any ancestor of X.

- Thus one rotation is enough to restore the AVL tree properly
- There are 4 different cases (actually 2) , so don't mix up them
- The time complexity to perform rotation is 0(1)
- The time completely to insert, and find a node that violates the AVL property is dependent on the height of the tree, which is 0(log(n))
- So insertion takes o(log(n))

*****Deletion :**

- Delete a node X as in ordinary BST(Note that X is either a leaf or X has exactly one child)
- Check and restore the AVL tree property.
- Trace from path of deleted node towards the root and check if the AVL tree property is violated
- Similar to an insertion operation, there are four cases to restore the AVL tree property.
- The only difference from insertion is that after we perform a rotation at x, we may have to perform, a rotation at some ancestors of x, It may involve several rotations.
- Therefore, we must continue to trace the path until we reach the root.
- The time complexity to delete a node is dependent on the height of the tree, which is also o(log(n))