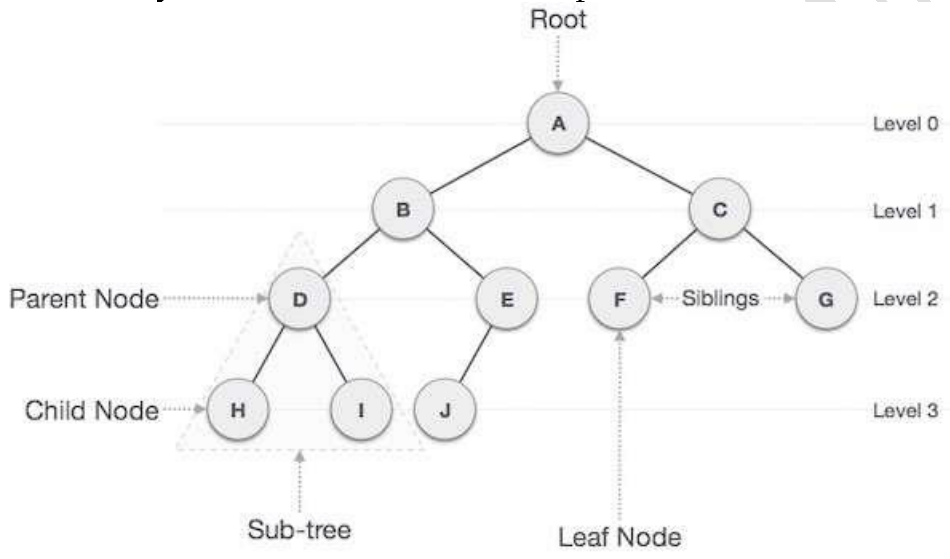
Lecture 1- Tree & Binary Search Tree

Tree represents the nodes connected by edges. We will discuss binary tree or binary search tree specifically.

Binary Tree

Binary Tree is a special data structure used for data storage purposes. A binary tree has a special condition that each node can have a maximum of two children. A binary tree has
the benefits of both an ordered array and a linked list as search is as quick as in a sorted array and insertion or deletion operation are as fast as in linked list.



Important Terms
Following are the important terms with respect to tree.
□□**Path** − Path refers to the sequence of nodes along the edges of a tree.
□□**Root** – The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
□□**Parent** − Any node except the root node has one edge upward to a node called parent.
□□**Child** – The node below a given node connected by its edge downward is called its child node.
□□**Leaf** – The node which does not have any child node is called the leaf node.

□□**Subtree** − Subtree represents the descendants of a node.

□□**Visiting** − Visiting refers to checking the value of a node when control is on the node.
□□**Traversing** − Traversing means passing through nodes in a specific order.
□□**Levels** − Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
□□**Keys** − Key represents a value of a node based on which a search operation is to be carried out for a node.

Improving the Search Time with Binary Search Trees (BSTs)

A *binary search tree* is a special kind of binary tree designed to improve the efficiency of searching through the contents of a binary tree. Binary search trees exhibit the following property: for any node *n*, every descendant node's value in the left *subtree* of *n* is less than the value of *n*, and every descendant node's value in the right *subtree* is greater than the value of *n*.

1-Search Nodes in a BST:
our searching algorithm goes as follows. We have a node *n* we wish to find (or determine if it exists), and we have a reference to the BST's root. This algorithm performs a number of comparisons until a null reference is hit or until the node we are searching for is found. At each step we are dealing with two nodes: a node in the tree, call it *c*, that we are currently comparing with *n*, the node we are looking for. Initially, *c* is the root of the BST. We apply the following steps:

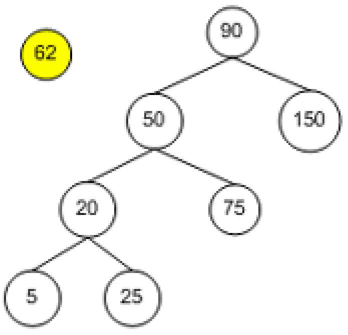
1. If *c* is a null reference, then exit the algorithm. *n* is not in the BST.
2. Compare *c*'s value and *n*'s value.
3. If the values are equal, then we found *n*.
4. If *n*'s value is less than *c*'s then *n*, if it exists, must be in the *c*'s left subtree. Therefore, return to step 1, letting *c* be *c*'s left child.
5. If *n*'s value is greater than *c*'s then *n*, if it exists, must be in the *c*'s right subtree. Therefore, return to step 1, letting *c* be *c*'s right child.

2-Insert Nodes into a BST
When adding a new node we can't arbitrarily add the new node; rather, we have to add the new node such that the binary search tree property is maintained.

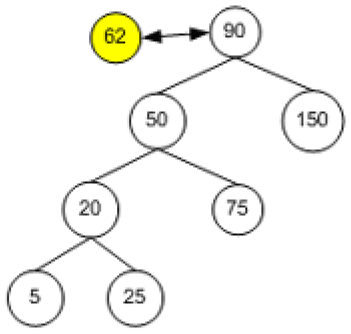
When inserting a new node we will always insert the new node as a leaf node. The only challenge, then, is finding the node in the BST which will become this new node's parent. Like with the searching algorithm, we'll be making comparisons between a node *c* and the node to be inserted, *n*. We'll also need to keep track of *c*'s parent node. Initially, *c* is the BST root and *parent* is a null reference. Locating the new parent node is accomplished by using the following algorithm:

1. If *c* is a null reference, then *parent* will be the parent of *n*. If *n*'s value is less than *parent*'s value, then *n* will be *parent*'s new left child; otherwise *n* will be *parent*'s new right child.
2. Compare *c* and *n*'s values.
3. If *c*'s value equals *n*'s value, then the user is attempting to insert a duplicate node. Either simply discard the new node, or raise an exception. (Note that the nodes' values in a BST must be unique.)
4. If *n*'s value is less than *c*'s value, then *n* must end up in *c*'s left subtree. Let *parent* equal *c* and *c* equal *c*'s left child, and return to step 1.
5. If *n*'s value is greater than *c*'s value, then *n* must end up in *c*'s right subtree. Let *parent* equal *c* and *c* equal *c*'s right child, and return to step 1.

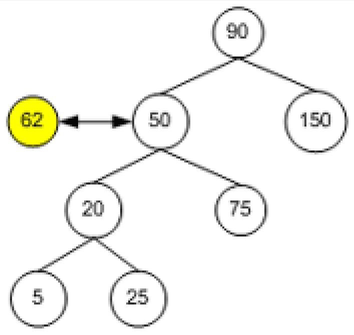
Given the following BST, we want to insert a node with the value 62...



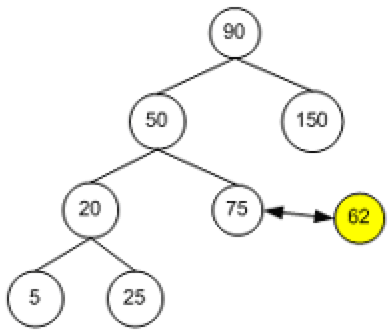
We start by comparing the node to insert (62) with the root (90). We see that 62 is less than 90, so we know 62 must be added somewhere to the root's left subtree.



We next compare 62 to 50. Since 62 is greater than 50, 62 must belong somewhere in 50's right subtree.

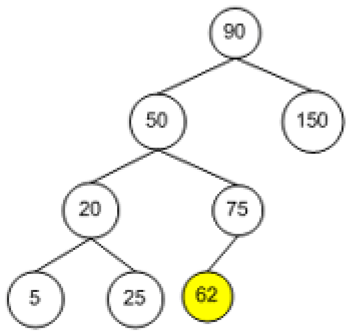


We next compare 62 to 75. Since 75 is greater than 62, 62 must exist somewhere in 75's left subtree.



Since 75's left child is a null reference, we have found the new location for node 62!

All that's left to do is set 75's left child to 62, which adds 62 to the BST tree and maintains the binary search tree property.



The First Step: Creating Node Class

Node class has three member variables:

- number. This member variable contains the data stored in the node of the type interger.
- rightLeaf, of type Node This member variable represents the node's right children.
- leftLeaf, of type Node This member variable represents the node's left children.

Method insertData works in recursive way to insert a new node in binary tree, It check the value of Node and decide to insert the new node to left or right.

Method search works also in recursive way, searching for node with required value.

```
class Node
    {
      private int number;
      public Node rightLeaf;
      public Node leftLeaf;

      public Node(int value)
      {
        number = value;
        rightLeaf = null;
        leftLeaf = null;
      }

      public bool isLeaf( Node node)
      {
        return (node.rightLeaf == null && node.leftLeaf == null);

      }

      public void insertData(ref Node node, int data)
```

```
{
   if (node == null)
   {
      node = new Node(data);

   }
   else if (node.number < data)
   {
      insertData(ref node.rightLeaf, data);
   }

   else if (node.number > data)
   {
      insertData(ref node.leftLeaf, data);
   }
}

public bool search(Node node, int s)
{
   if (node == null)
      return false;

   if (node.number == s)
   {
      return true;
   }
   else if (node.number < s)
   {
      return search(node.rightLeaf, s);
   }
   else if (node.number > s)
   {
      return search(node.leftLeaf, s);
   }

   return false;
}

public void display(Node n)
{
```

```
                if (n == null)
                    return;

                display(n.leftLeaf);
                Console.Write(" " + n.number);
                display(n.rightLeaf);
            }

        }
```

Creating the BinaryTree Class

The BinaryTree class contains a private member variable root. root is of type Node and represents the root of the binary tree.
Count represent number of nodes in tree.

```
class BinaryTree
        {
            private Node root;
            private int count;

            public BinaryTree()
            {
                root = null;
                count = 0;
            }
            public bool isEmpty()
            {
                return root == null;
            }

            public void insert(int d)
            {
                if (isEmpty())
                {
                    root = new Node(d);
                }
                else
                {
```

```
            root.insertData(ref root, d);
        }

        count++;
    }

    public bool search(int s)
    {
        return root.search(root, s);
    }



    public void display()
    {
        if (!isEmpty())
            root.display(root);
    }

    public int Count()
    {
        return count;
    }
}
```