Trees – II

Height of a Binary tree
Binary Search Tree
- **Searching**
- **Insertion**
- **Traversal**
- **Creation**

Non-Recursive PreOrder Traversal

Height of a binary tree:

The height of a binary tree is the height of the root node. The height of any node in a tree by itself is 0. If it has any children, then its height increases by 1.
Thus the height of a node is *max (height of left subtree, height of right subtree) + 1* .
If a node does not have any children, both the left and right height is going to be – 1, and thus its own height is going to be 0.

```
int height (Lnode p)
{
     int leftheight, rightheight;
     if (p == NULL)
          return -1;
     else
     {
          leftheight = height(p.left);
          rightheight = height(p.right);
          if (leftheight > rightheight)
               return(leftheight + 1);
          else
               return(rightheight + 1);
     }
}
```

Binary Search Tree (BST)

We have seen earlier that if the values in nodes of a binary tree are arranged in a specific order, with all elements smaller than the root stored in left subtree and all elements greater than the root stored as right subtree, it represents a sorted list. The search complexity reduces considerably, as the height of the tree is much less than total number of elements. Of course, one has to keep in mind that the tree has to be organized in a specific manner. Such a tree is known as a Binary Search tree, because it permits us to carry out a search similar to the **Binary search** method that we have used on a sorted array.

Let us first of all define a BST.

A Binary search tree (BST) is a binary tree that is

either empty , or

each node contains a data value satisfying the following:

a) all data values in the left subtree are smaller than the data value in the root.

b) the data value in the root is smaller than all values in its right subtree.

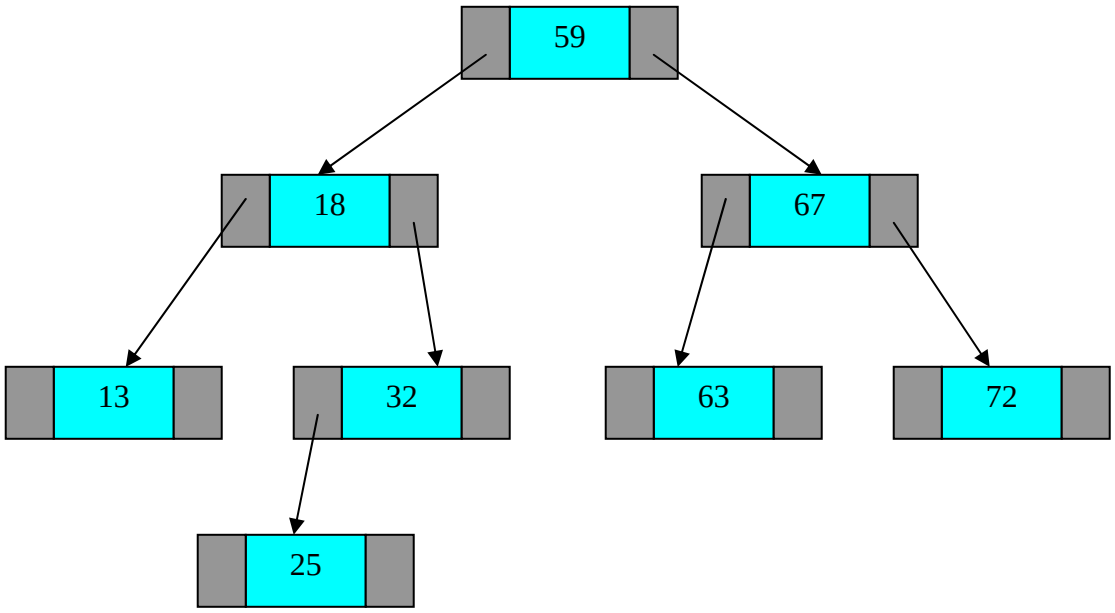
c) the left and right subtrees are also binary search tees.

Searching for a target in the Binary Search Tree

The definition of a binary search tree allows us to quickly search for a particular value in the BST. Check the given value with the value in the root node. If it matches, return 1, else if given value is smaller , look into left subtree, else look into right subtree. If subtree is null, return 0.

```
Public boolean treeSearch( Lnode p, int target)
{    if (p!=NULL)
        {
          if (p.data == target)
            return true;
            else if (p.data > target)
               treeSearch(p.left);
            else
                 treeSearch(p.right);
         }
      return false;
}
```

Example Tree:



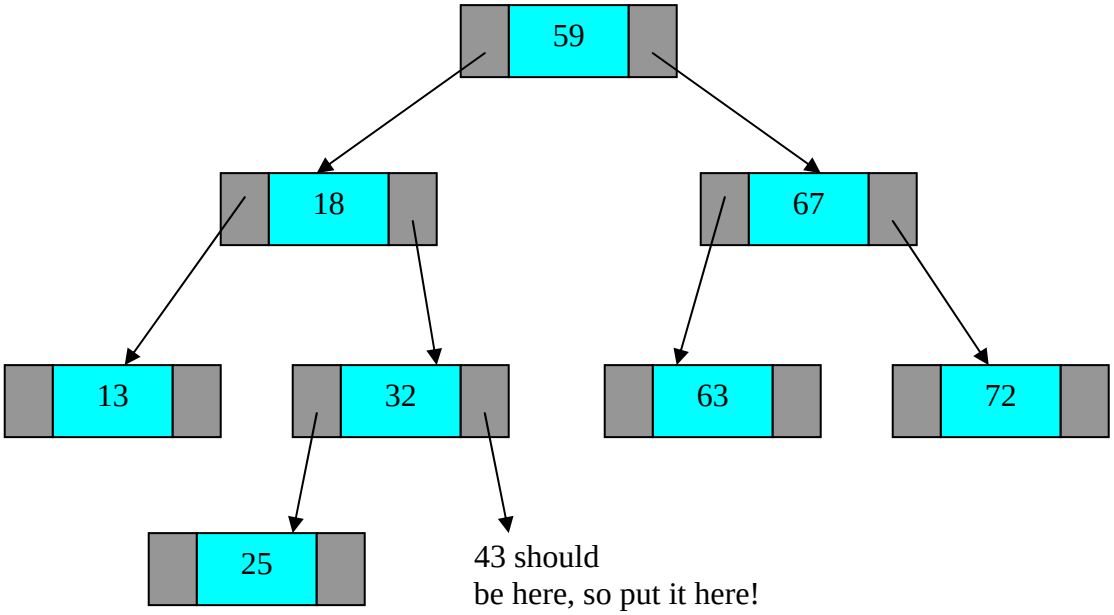
Inserting a node in a BST

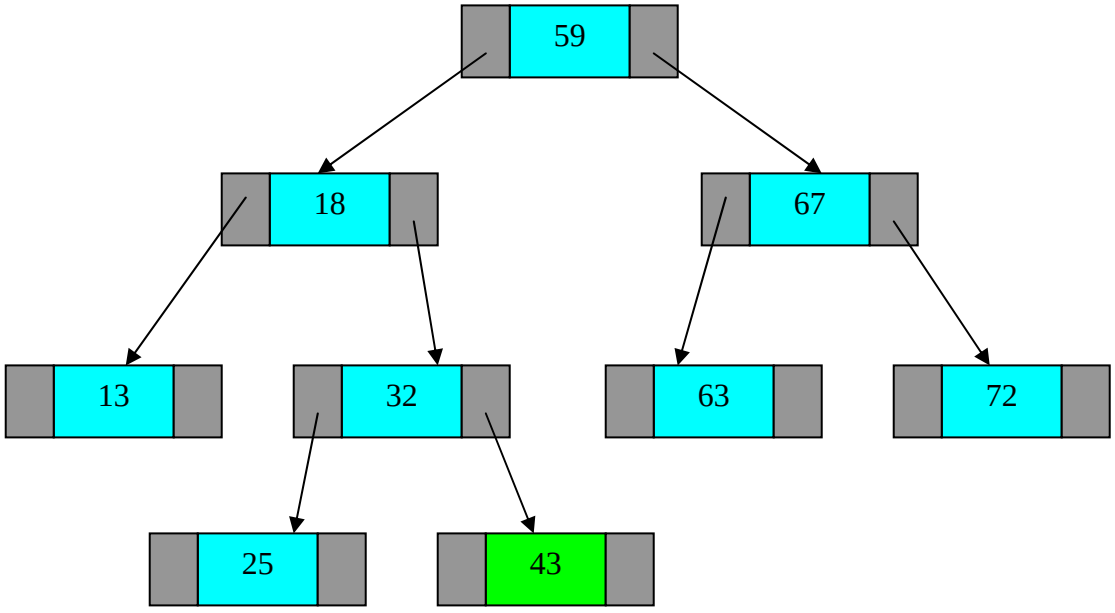
Insertion of a new node in a BST has to be done only at the appropriate place for it, so that overall BST structure is still maintained, i.e. the value at any node should be less than that at right node and less than that at the left node.

Inserting a new node into a BST **always** occurs at a NULL pointer.
There is never a case when existing nodes need to be rearranged to accommodate the new node.

As an example, consider inserting the new value 43 into the BST shown above. Where is the new node supposed to go?

Hint: search for node containing 43. Obviously, you won't find it, but the search algorithm has taken you to the NULL pointer where it should be placed, so this is the appropriate place to insert it.





Inserting a new value in a BST:

The following function inserts a value “d” in its proper place in a Binary Search Tree. It recursively searches for the proper place to insert the new value, and returns with a pointer to the position of the new node. Then a new node is created and the value is stored in it.

```
//Inserts a node with value d in a tree with root p
Public Lnode insert( int d , Lnode p)
{
//Inserting the root
   if( p== null)
             p = new Lnode (d );

//Inserting element less than the root, go to left
   else  if(d  <=  p.info)
             p.left = insert( d, p.left);

//Inserting element greater than root, go to right
   else  if(d  >  p.info)
             p.right = insert( d, p.right );

   else
      ;

   return p;
}
```

Traversal of a Binary Search Tree

The above BST can be traversed starting with the root node (Preorder traversal) to result in the sequence 59,18,13,32,25,43,67,63,72

However, an interesting feature is revealed with the Inorder Traversal which yields

13, 18, 25, 32, 43, 59, 63, 67, 72

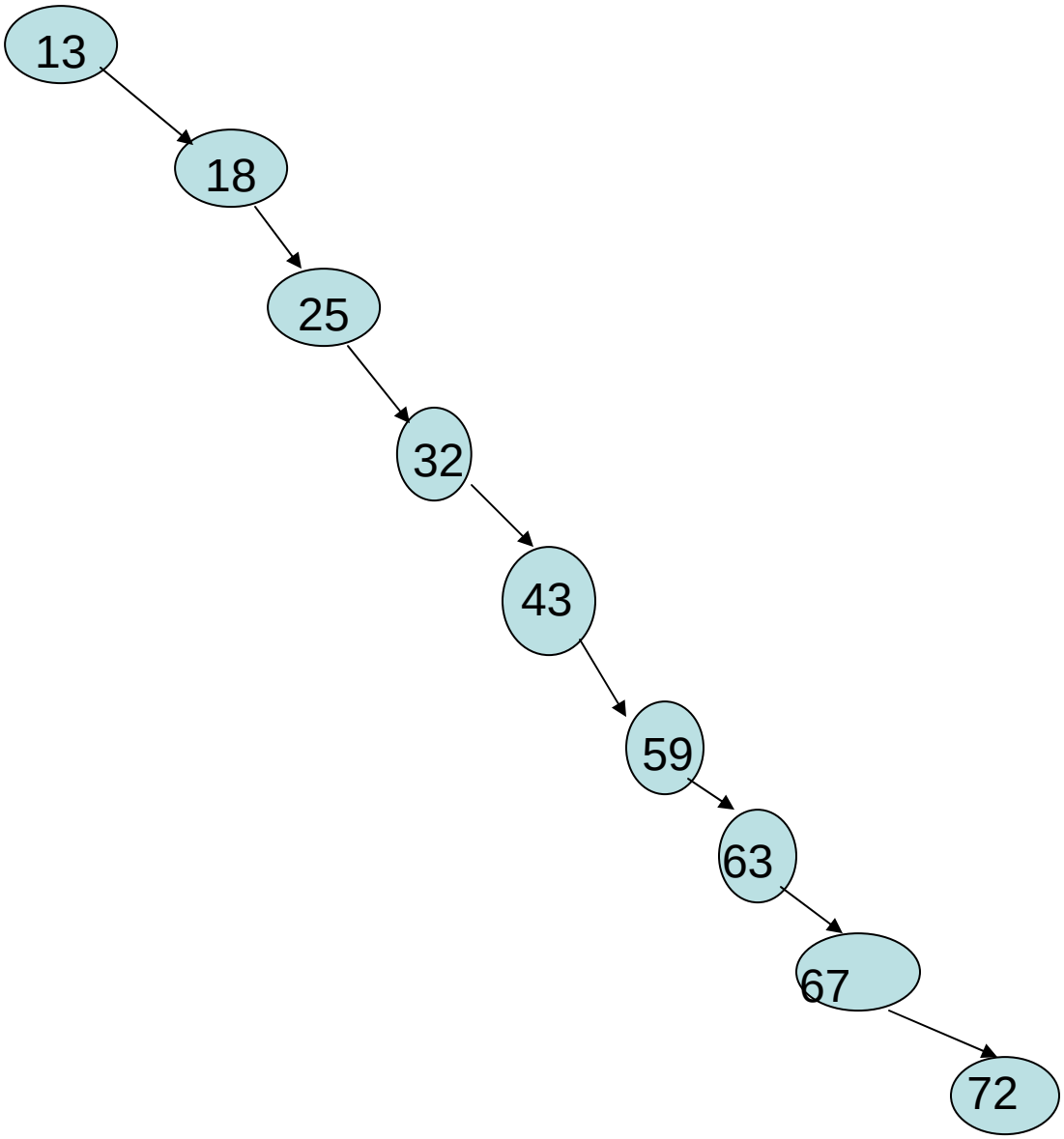
What do you notice?
This is an **ordered listing** of the values of BST nodes, with the left most node being the smallest element and the right most node being the largest element.

Creating a Binary Search Tree

To create a binary search tree, keep on inserting the nodes as and when they arrive. Note that the shape of the BST will depend on the order of insertion of the nodes.
The above tree was created from the sequence 59, 18, 13, 67, 32, 72, 25, 63, 43

If the values arrived in the following sequence: 13, 18, 25, 32, 43, 59, 63, 67, 72

the BST would take the following shape



Creating a balanced tree from sorted data:

Notice that the above tree is not a balanced tree, but skewed towards right, as all elements are in perfect order. In such a case, the search and insertion complexity would be O(n) instead of O(log n). If the sequence were entered in descending order then it would result in a left-skewed tree. For any other ordering of the sequence the complexity would lie in between O(n) and O(log n).

Suppose we were interested in generating a balanced BST, given any arbitrary sequence of values.
We could this by first storing all the elements in an array and sorting them in ascending order.

- Once sorted, the element at the midpoint of the array will become the root of the BST. The array can now be viewed as consisting of two subarrays, one to the left of the midpoint and one to the right of the midpoint.
- The middle element in the left subarray becomes the left child of the root node and the middle element in the right subarray becomes the right child of the root.
- This process continues with further subdivision of the original array until all the elements in the array have been positioned in the BST.
- Take care to completely generate the left subtree of the root before generating the right subtree of the root. If this is done, a simple recursive procedure can be used to generate a balanced BST.

```
void balance( int sequence[], int first, int last)
{
     int mid;
        if (first <= last) {
      mid = (first + last)/2;
      insert_BST(  info[mid], p);
      balance(sequence, first, mid-1);
      balance(sequence, mid+1, last);
   }
}
```

where the function insert_BST creates a new node with the value sequence[mid] and calls the ‘insert ‘ function to insert the new node into the BST.

Preorder tree traversal (Non Recursive version)

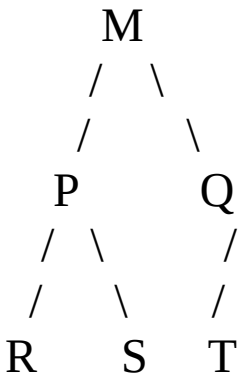
We have earlier seen a recursive algorithm for preorder traversal. In a preorder traversal, we visit the root node first, then we visit its left subtree (all the nodes) and finally visit its right subtree. Here we shall develop a non-recursive algorithm. Since we can visit only one node at a time, we shall make use of a stack to store the other nodes to be visited later.

To start with we push the root node on the stack.
Then we push the right child and the left child of the current node on a stack recursively.
Then we pop them and print them. Look at the following **algorithm**:

```
p = root;
if (p != NULL)
{
   push(p);
   while ( stack not empty)    {
        p = pop stack;
           printf(p.info);
       if ( p.right != NULL)
            push ( p. right);
       if (p.left != NULL)
           push (p . left);
      }
}
```

/ note the left child is pushed on top of right node, so that it gets popped up first */*

Consider the following tree



Non Recursive Preorder Traversal:

Stack position	Popped values
M	
	M
Q P	
Q	**M P**
Q S R	
Q S	**M P R**
Q	**M P R S**
	M P R S Q
T	
-	**M P R S Q T**