Trees – II

Non-Recursive PreOrder Traversal
Height of a binary tree
Binary Search Tree

- **Searching**
- **Insertion**
- **Traversal**
- **Creation**

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. In this section we shall develop a non-recursive algorithm for the preorder traversal.
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(“%d “,p->data);
       if ( p->right_child != NULL)
            push ( p-> right_child);
       if (p ->left_child != NULL)
           push (p -> left_child);
      }
}
```

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

Consider the following tree

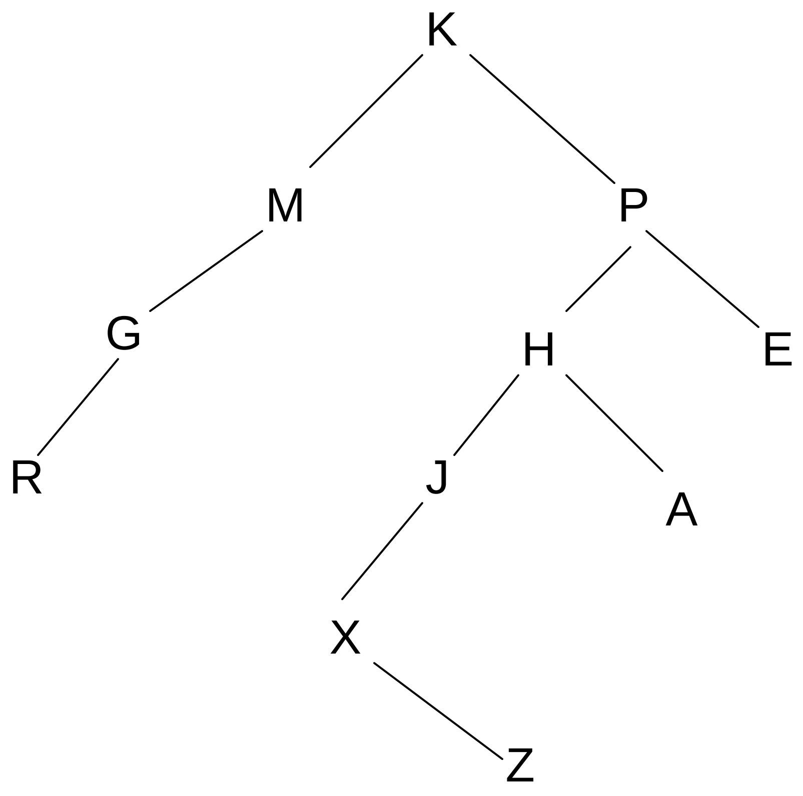
```
           M
          /  \
         /     \
       P        Q
      /  \        /
     /     \     /
   R      S    T
```

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**
-	**M P R S Q**
T	_ _ _
-	**M P R S Q T**

Height of a binary tree:

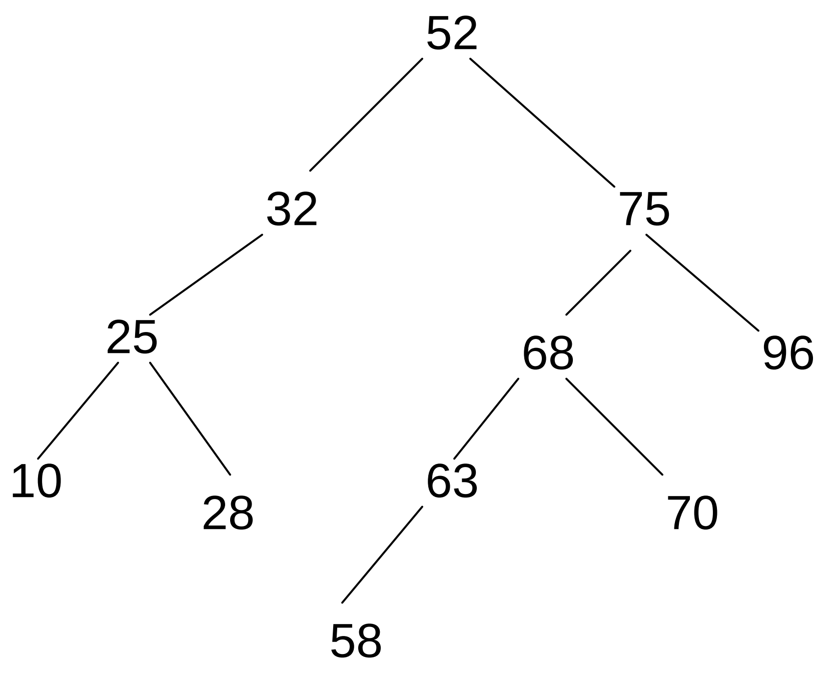
The height of a tree is given by the height of the root node, which is a count of number of nodes on the longest path from the root to the leaf node. In the following tree the longest



path is from K to Z which contains 6 nodes, so the height of root node K is 6.
The height of the root node can be found by considering the left and right subtrees separately and taking the maximum height. For the above tree, height of M is 3, and height of P is 5, the max of 3 and 5 is 5. Add 1 to it (for the root node itself) to get the total height of 6. Here is a function which returns the height of a binary tree.

```
int height (struct treenode *ptr)
{
     int leftheight, rightheight;

     if (ptr == NULL)
        return 0;
     else
     {     leftheight = height(ptr->left);
              rightheight = height(ptr->right);

              if (leftheight > rightheight)
                  return(leftheight + 1);
              else
                  return(rightheight + 1);
     }
}
```

Binary Search Tree (BST)



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 one has to check only one node at each level of the height, so complexity is given by the height of the tree, and 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.

<u>Traversal of a Binary Search Tree</u>

The above BST can be traversed starting with the root node *(Preorder traversal)* to result in the sequence 52, 32, 25, 10, 28, 75, 68, 63, 58, 70, 96.

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

10, 25, 28, 32, 52, 58, 63, 68, 70, 75, 96

What do you notice? This is nothing but an **ordered listing** of the values of BST nodes in the increasing order, with the left most node being the smallest element and the right most node being the largest element.

<u>Searching for a target in the Binary Search Tree</u>

The binary search tree definition allows you 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 the given value is smaller than the root value,look into the left subtree,
else look into the right subtree.
If subtree is null, return 0.

```
struct tree_node{
     int data;
     struct tree_node *left;
     struct tree_node *right;
};

int treeSearch( struct tree_node *p, int target)
{
     if (p!=NULL)
        {
          if (p->data == target)
                    return 1;
          else if (p->data > target)
```

```
                    retun treeSearch(p->left, target);
               else
                    return treeSearch(p->right, target);
         }
      return 0;
}
```

Apply this function on the binary search tree shown earlier to locate 63. You have to just search 4 nodes to locate the value. What is the complexity of this function? If the tree is balanced it should be *O(log n)*, where *n* is the number of nodes on the tree. In the worst case, when the tree is skewed in one direction, it would be *O(n)*.

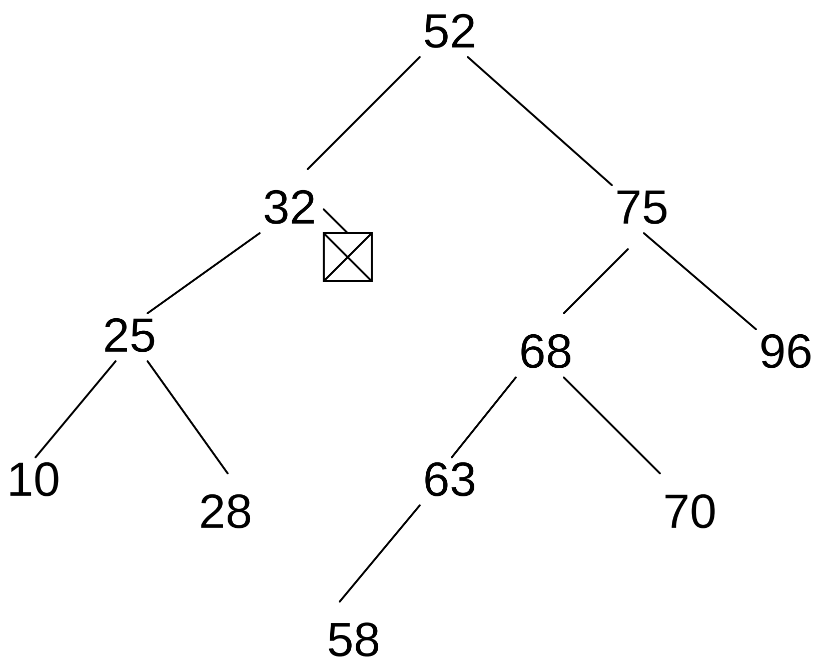
Inserting a node in a BST

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.

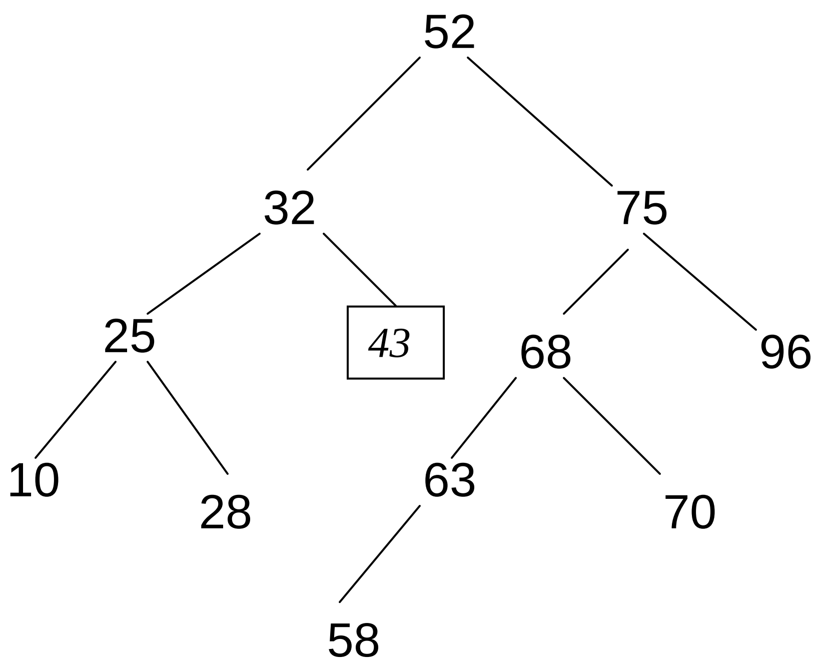
Inserting a new node into a BST **always** occurs at a NULL pointer.
There is never a case when existing nodes need to be rearranged to accommodate the new node.

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, which is the right pointer of 32.



After insertion the tree takes the form:



Inserting a new value in a BST:

The following function inserts a value “d” in its proper place in a Binary Search Tree *(with distinct values in the nodes, no repeated values).* 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. Note how the new node gets automatically linked to its parent node.

//Inserts a node with value d in a tree with root p

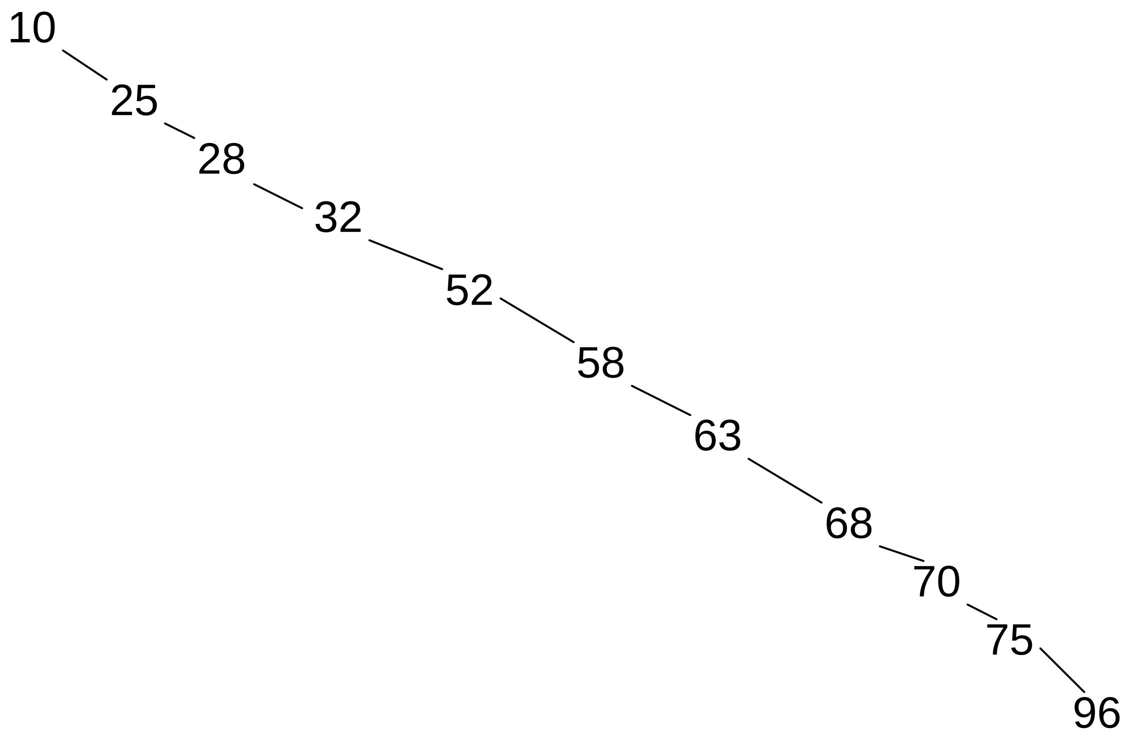
```
struct treenode* insert( int d , struct treenode * p)
{
//Inserting the value in a new node
   if( p== null)
        {
             p = (struct treenode*)(malloc(sizeof(struct
                   treenode)));
                p->data = d;
                p->left = NULL;
                p->right = NULL;
        }

//Inserting a value less than the root value, move left
   else  if(d  <  p->data)
             p->left = insert( d, p->left);

//Inserting a value greater than the root value, move right

   else  if(d  >  p->data)
             p->right = insert( d, p->right );


   return p;
}
```

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 earlier BST tree was created from the sequence
52, 32, 25, 75, 68, 96, 63, 10, 70, 28, 58

If the values arrived in the following sequence: 10, 25, 28, 32, 52, 58, 63, 68, 70, 75, 96

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 the elements are in perfectly sorted 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 is chosen as 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.
- We have to 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(  sequence[mid], p);
      balance(sequence, first, mid-1);
      balance(sequence, mid+1, last);
   }
}
```