# Experiment no.11

**Kareena Chinchkar[16] and Pooja Athare[18]**

October 21, 2023

## Title:Binary Search Tree

**Aim:Program to create a Binary Search Tree(BST) by considering the keys**

**Learning Objective:** Write functions to implement linear and non-linear data structure operations for solving a given problem

## 1 What is a tree?

A tree is a kind of data structure that is used to represent the data in hierarchical form. It can be defined as a collection of objects or entities called as nodes that are linked together to simulate a hierarchy. Tree is a non-linear data structure as the data in a tree is not stored linearly or sequentially.

## 2 What is a Binary Search tree?

A binary search tree follows some order to arrange the elements. In a Binary search tree, the value of left node must be smaller than the parent node, and the value of right node must be greater than the parent node. This rule is applied recursively to the left and right subtrees of the root.

Let's understand the concept of Binary search tree with an example.

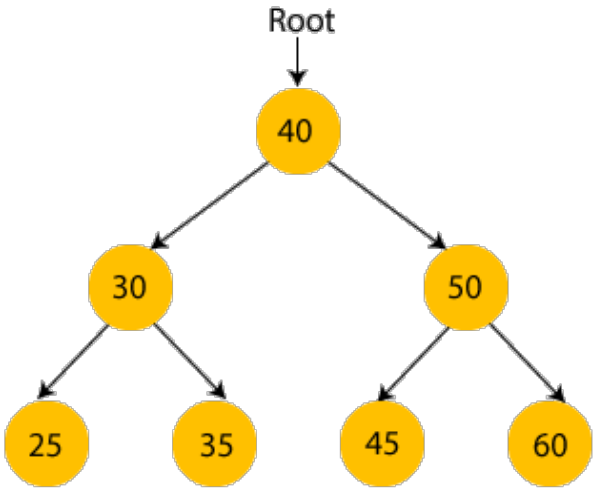



Figure 1: Binary Search Tree.

In the above figure, we can observe that the root node is 40, and all the nodes of the left subtree are smaller than the root node, and all the nodes of the right subtree are greater than the root node.

Similarly, we can see the left child of root node is greater than its left child and smaller than its right child. So, it also satisfies the property of binary search tree. Therefore, we can say that the tree in the above image is a binary search tree.

Suppose if we change the value of node 35 to 55 in the above tree, check whether the tree will be binary search tree or not.

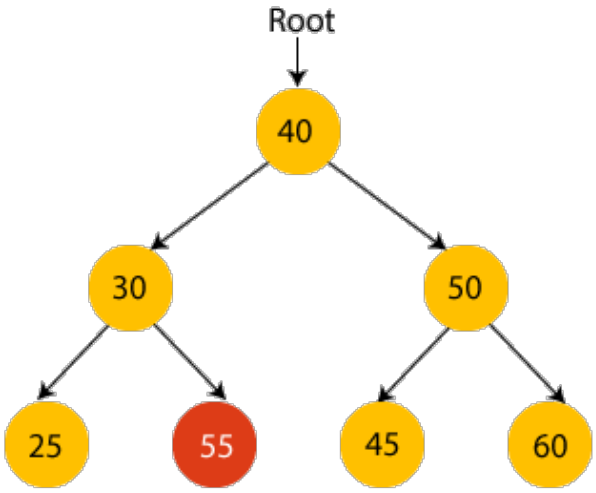



Figure 2: Binary Search Trees.

In the above tree, the value of root node is 40, which is greater than its left child 30 but smaller than right child of 30, i.e., 55. So, the above tree does not satisfy the property of Binary search tree. Therefore, the above tree is not a binary search tree.

## 3 Applications of Binary Search tree:

BSTs are used for indexing.
It is also used to implement various searching algorithms.
IT can be used to implement various data structures.
BSTs can be used in decision support systems to store and quickly retrieve data.
BSTs can be used to store and quickly retrieve data in computer simulations.
BSTs can be used to implement fast autocomplete systems.
BSTs can be used to implement decision trees, which are used in machine learning and artificial intelligence to model decisions and predict outcomes. Decision trees are used in many applications, including medical diagnosis, financial analysis, and marketing research.

## 4 Real-time Application of Binary Search tree:

BSTs are used for indexing in databases.
It is used to implement searching algorithms.
BSTs are used to implement Huffman coding algorithm.
It is also used to implement dictionaries.
Used for data caching.
Used in Priority queues.
Used in spell checkers.

### 4.1 Insertion using an key

A new key is always inserted at the leaf by maintaining the property of the binary search tree. We start searching for a key from the root until we hit a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node. The below steps are followed while we try to insert a node into a binary search tree:
1. Check the value to be inserted (say X) with the value of the current node (say val) we are in:
2. If X is less than val move to the left subtree.
3. Otherwise, move to the right subtree.
4.Once the leaf node is reached, insert X to its right or left based on the relation between X and the leaf node's value.
***Algorithm:***
1. Create a new BST node and assign values to it.

2. insert(node, key)
i) If root == NULL,
return the new node to the calling function.
ii) if root=¿data ¡ key
call the insert function with root=¿right and assign the return value in root=¿right.
root-¿right = insert(root=¿right,key)
iii) if root=¿data ¿ key
call the insert function with root-¿left and assign the return value in root=¿left.
root=¿left = insert(root=¿left,key)
3. Finally, return the original root pointer to the calling function.
***Example:***

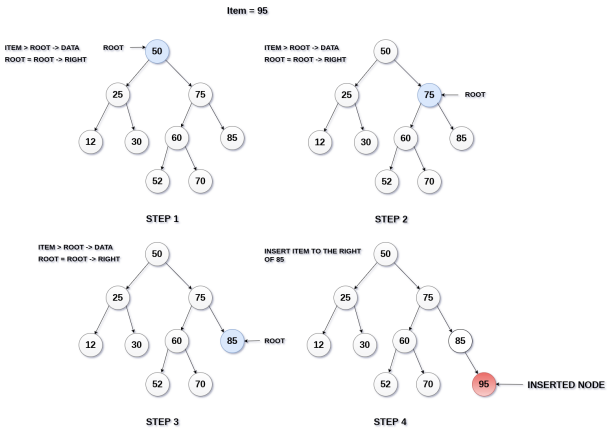



Figure 3: Insertion using an key.

## 4.2 Searching using an key

Let's say we want to search for the number X, We start at the root. Then:
We compare the value to be searched with the value of the root.
If it's equal we are done with the search if it's smaller we know that we need to go to the left subtree because in a binary search tree all the elements in the left subtree are smaller and all the elements in the right subtree are larger.
Repeat the above step till no more traversal is possible
If at any iteration, key is found, return True. Else False.
***Algorithm:***
Compare the element with the root of the tree.
If the item is matched then return the location of the node.
Otherwise check if item is less than the element present on root, if so then move to the left sub-tree.
If not, then move to the right sub-tree.
Repeat this procedure recursively until match found.
If element is not found then return NULL.
***Example:***

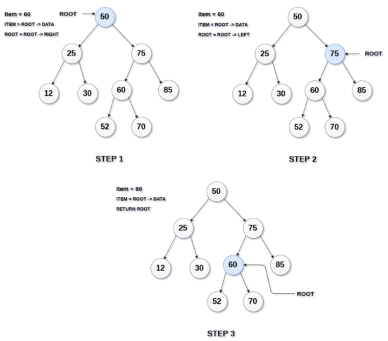



Figure 4: Searching using an key.

## 4.3 Minimum Value of node element

Traverse the node from root to left recursively until left is NULL.
The node whose left is NULL is the node with minimum value.
The basic idea behind this approach is to exploit the properties of a Binary Search Tree (BST). In a BST, the left subtree of a node contains all the nodes with values less than the node's value, and the right subtree contains all the nodes with values greater than the node's value.
***Algorithm:***
Follow the steps to implement the above idea:
1.Start at the root node of the BST.
2.If the left child of the current node is NULL, return the value of the current node. This is the minimum element in the BST. 3.If the value of the left child is less than the value of the current node, move to the left subtree and repeat step 2.
4.If the value of the left child is greater than or equal to the value of the current node, move to the right subtree and repeat step 2.
5.Repeat steps 2-4 until you reach a leaf node.
***Time Complexity:***
O(N) Worst case happens for left skewed trees in finding the minimum value.
O(1) Best case happens for right skewed trees in finding the minimum value.

## 4.4 Maximum Value of node element

Traverse the node from root to right recursively until right is NULL.
The node whose right is NULL is the node with maximum value.
***Algorithm:***
Follow the steps below to implement the above idea:
1.Initialize a variable maximum val to store the maximum value seen so far, and a pointer curr to point to the current node.
2.While curr is not NULL, do the following:
3.If the left subtree of curr is NULL, update maximum val with the value of curr, and move to the right subtree of curr.
4.If the left subtree of curr is not NULL, find the predecessor of curr in its left subtree. The predecessor is the rightmost node in the left subtree of curr.
5.If the right child of the predecessor is NULL, set it to curr and move to the left child of curr.
6.If the right child of the predecessor is curr, restore it to NULL, update maximum val with the value of curr, and move to the right child of curr.
7.Return maximum val. Steps 2-4 until you reach a leaf node.
***Time Complexity:***
O(N) Worst case happens for right skewed trees in finding the maximum value.
O(1) Best case happens for left skewed trees in finding the maximum value.
***Example:***

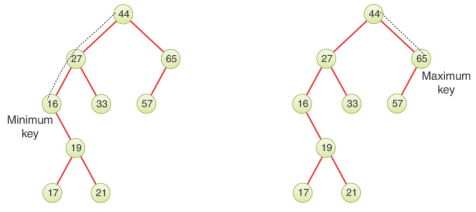



Figure 5: Min and Max using an key.

## 4.5 Transversal Value of node element

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. A Tree Data Structure can be traversed in following ways:
***Algorithm:***
### 1.Depth First Search or DFS
-¿Inorder Traversal :

Algorithm Inorder(tree)=
i.Traverse the left subtree, i.e., call Inorder(left-¿subtree)
ii.Visit the root.
iii.Traverse the right subtree, i.e., call Inorder(right-¿subtree) -¿Preorder Traversal :
Algorithm Preorder(tree)=
i.Visit the root.
ii.Traverse the left subtree, i.e., call Preorder(left-¿subtree)
iii.Traverse the right subtree, i.e., call Preorder(right-¿subtree)
-¿Postorder Traversal :
Algorithm Postorder(tree)=
i.Traverse the left subtree, i.e., call Postorder(left-¿subtree)
ii.Traverse the right subtree, i.e., call Postorder(right-¿subtree)
iii.Visit the root

**2.Level Order Traversal or Breadth First Search or BFS :**

For each node, first, the node is visited and then it's child nodes are put in a FIFO queue. Then again the first node is popped out and then it's child nodes are put in a FIFO queue and repeat until queue becomes empty.

**3.Boundary Traversal :**

The Boundary Traversal of a Tree includes:
i.left boundary (nodes on left excluding leaf nodes)
ii.leaves (consist of only the leaf nodes)
iii.right boundary (nodes on right excluding leaf nodes)

**4.Diagonal Traversal :**

In the Diagonal Traversal of a Tree, all the nodes in a single diagonal will be printed one by one.
***Example:***

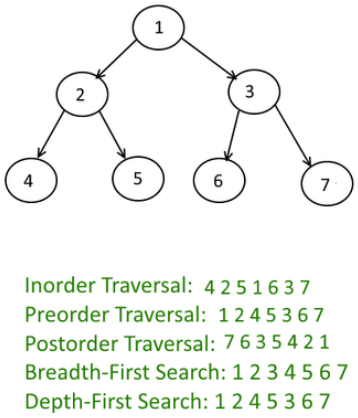



Figure 6: Traversal using an key.

# 5 Lab Outcome

Students should able to design and analyze simple linear and non linear data structures. It strengthen the ability to the students to identify and apply the suitable data structure for the given real world problem. It enables them to gain knowledge in practical applications of data structures .

# 6 Conclusion

A binary search tree (BST) is a fundamental data structure used in computer science and programming for efficient searching, insertion, and deletion of elements. Here's a concise conclusion about binary search trees:
1.Efficient Search:
Binary search trees provide efficient searching capabilities. They are organized in a way that allows for quick retrieval of data. When searching for a specific element, you can eliminate half of the remaining elements with each comparison, making the average time complexity for search operations O(log n).

2.Ordered Structure:
BSTs are ordered data structures, where each node has a value, and all nodes in the left subtree have values less than the node's value, while all nodes in the right subtree have values greater. This property makes it useful for applications like dictionaries or databases.
3.Insertion and Deletion:
Insertion and deletion operations in a BST also have average time complexities of O(log n). They maintain the BST's property by rearranging the tree as necessary.
4.Balanced BSTs:
While BSTs offer excellent average-case time complexities, their performance can degrade in the worst case if the tree becomes unbalanced. To address this, self-balancing BSTs like AVL trees and Red-Black trees are used to ensure that the tree remains balanced, resulting in consistent O(log n) performance for all operations.
5.In-Order Traversal:
In-order traversal of a BST visits the nodes in ascending order, which is beneficial when you need to process elements in sorted order.
In conclusion, binary search trees are a valuable data structure with efficient searching, insertion, and deletion operations when they are reasonably balanced. However, in cases where the tree becomes unbalanced, self-balancing variants or other data structures may be preferred to maintain consistent performance.

## Insertion using an key.

```c
#include <stdio.h>
#include <stdlib.h>

struct node {
    int key;
    struct node *left, *right;
};

// A utility function to create a new BST node
struct node* newNode(int item)
{
    struct node* temp
        = (struct node*)malloc(sizeof(struct node));
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

// A utility function to do inorder traversal of BST
void inorder(struct node* root)
{
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->key);
        inorder(root->right);
    }
}

// A utility function to insert
// a new node with given key in BST
struct node* insert(struct node* node, int key)
{
    // If the tree is empty, return a new node
    if (node == NULL)
        return newNode(key);

    // Otherwise, recur down the tree
    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);

    // Return the (unchanged) node pointer
    return node;
}

// Driver Code
int main()
{
    /* Let us create following BST
              50
           /     \
          30      70
          / \ / \
    20 40 60 80 */
    struct node* root = NULL;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    // Print inorder traversal of the BST
    inorder(root);

    return 0;
}
```

Figure 7: Insertion using an key.

**Output of Insertion using an key.**

**Output**

```
20 30 40 50 60 70 80
```

**Time Complexity:**

- The worst-case time complexity of insert operations is **O(h)** where **h** is the height of the Binary Search Tree.
- In the worst case, we may have to travel from the root to the deepest leaf node. The height of a skewed tree may become **n** and the time complexity of insertion operation may become **O(n).**

Figure 8: Output of Insertion using an key.

**Output of Searching using an key.**

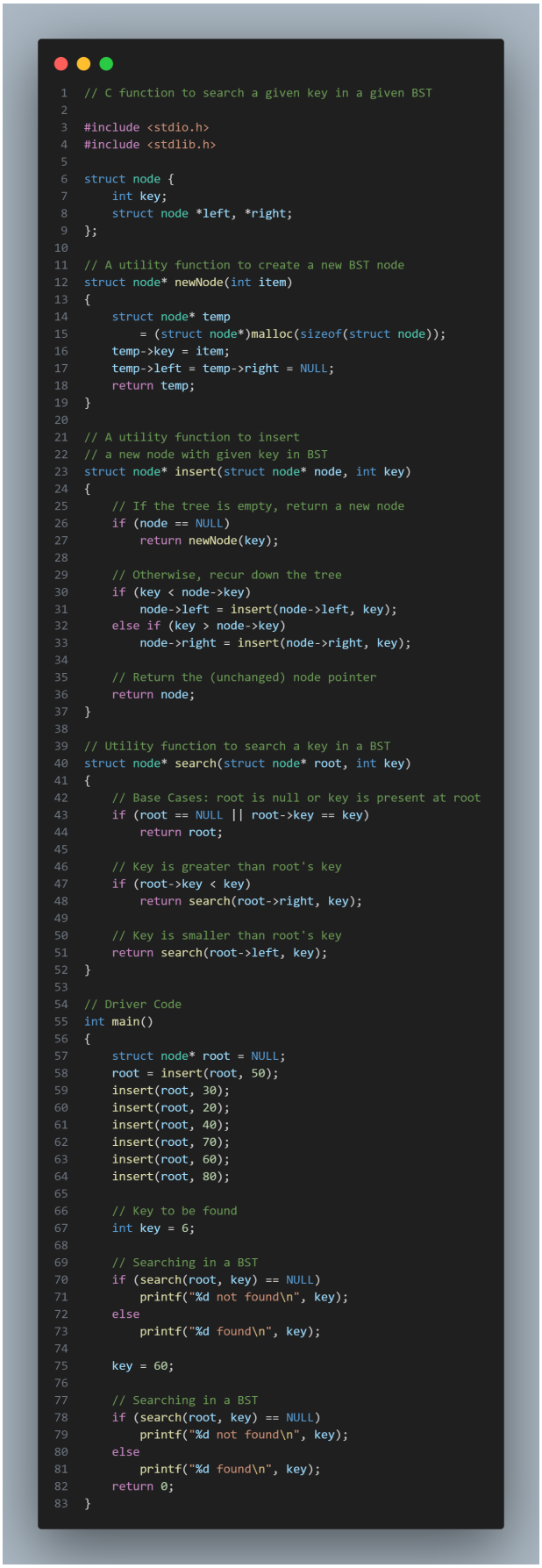
**Output**

```
6 not found
60 found
```

**Time complexity:** O(h), where h is the height of the BST.
**Auxiliary Space:** O(h), where h is the height of the BST. This is because the maximum amount of space needed to store the recursion stack would be **h**.

Figure 9: Output of Searching using an key.

## Searching using an key



```c
// C function to search a given key in a given BST

#include <stdio.h>
#include <stdlib.h>

struct node {
    int key;
    struct node *left, *right;
};

// A utility function to create a new BST node
struct node* newNode(int item)
{
    struct node* temp
        = (struct node*)malloc(sizeof(struct node));
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

// A utility function to insert
// a new node with given key in BST
struct node* insert(struct node* node, int key)
{
    // If the tree is empty, return a new node
    if (node == NULL)
        return newNode(key);

    // Otherwise, recur down the tree
    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);

    // Return the (unchanged) node pointer
    return node;
}

// Utility function to search a key in a BST
struct node* search(struct node* root, int key)
{
    // Base Cases: root is null or key is present at root
    if (root == NULL || root->key == key)
        return root;

    // Key is greater than root's key
    if (root->key < key)
        return search(root->right, key);

    // Key is smaller than root's key
    return search(root->left, key);
}

// Driver Code
int main()
{
    struct node* root = NULL;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    // Key to be found
    int key = 6;

    // Searching in a BST
    if (search(root, key) == NULL)
        printf("%d not found\n", key);
    else
        printf("%d found\n", key);

    key = 60;

    // Searching in a BST
    if (search(root, key) == NULL)
        printf("%d not found\n", key);
    else
        printf("%d found\n", key);
    return 0;
}
```

Figure 10: Searching using an key.

## Min and max using an key

```c
#include <stdio.h>
#include <stdlib.h>
#include <vector>
using namespace std;
struct node {
    int data;
    struct node* left;
    struct node* right;
};
struct node* newNode(int data)
{
    struct node* node
        = (struct node*)malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return (node);
}
struct node* insert(struct node* node, int data)
{
    if (node == NULL)
        return (newNode(data));
    else {
        if (data <= node->data)
            node->left = insert(node->left, data);
        else
            node->right = insert(node->right, data);

        return node;
    }
}

void inorder(struct node* node, vector<int>& sortedInorder)
{
    if (node == NULL)
        return;
    inorder(node->left, sortedInorder);

    sortedInorder.push_back(node->data);

    inorder(node->right, sortedInorder);
}

int main()
{
    struct node* root = NULL;
    root = insert(root, 4);
    insert(root, 2);
    insert(root, 1);
    insert(root, 3);
    insert(root, 6);
    insert(root, 4);
    insert(root, 5);

    vector<int> sortedInorder;
    inorder(
        root,
        sortedInorder);
    printf("\n Minimum value in BST is %d",
        sortedInorder[0]);
    getchar();
    return 0;
}
```

(a) Minimum

```c
#include <bits/stdc++.h>
using namespace std;

struct node {
    int data;
    struct node* left;
    struct node* right;
};

struct node* newNode(int data)
{
    struct node* newnode = new node();
    newnode->data = data;
    newnode->left = NULL;
    newnode->right = NULL;

    return (newnode);
}

struct node* insert(struct node* node, int data)
{

    if (node == NULL)
        return (newNode(data));
    else {
        if (data <= node->data)
            node->left = insert(node->left, data);
        else
            node->right = insert(node->right, data);

        return node;
    }
}

int maxValue(struct node* node)
{
    struct node* current = node;
    while (current->right != NULL)
        current = current->right;

    return (current->data);
}

int main()
{
    struct node* root = NULL;
    root = insert(root, 4);
    insert(root, 2);
    insert(root, 1);
    insert(root, 3);
    insert(root, 6);
    insert(root, 5);

    cout << "Maximum value in BST is " << maxValue(root);

    return 0;
}
```

(b) Maximum

Figure 11: Min and Max in Binary Search Tree

**Output of Min and Max.**

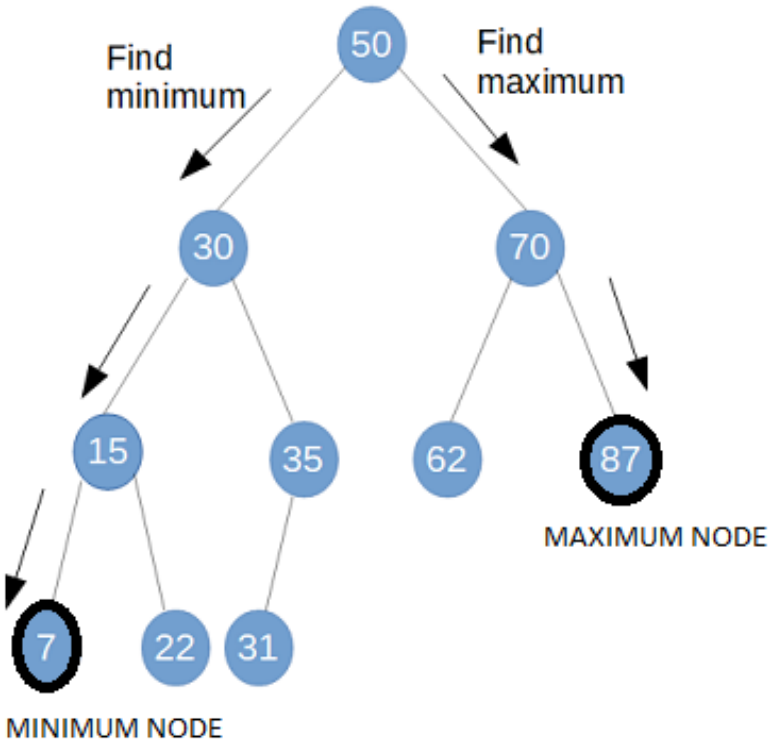
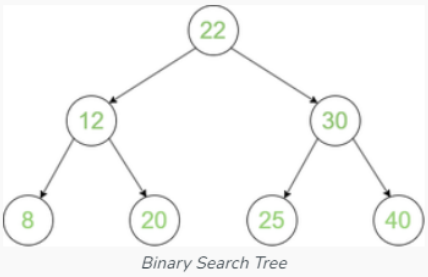



Figure 12: Output of Min and Max.

```c
#include <stdio.h>
#include <stdlib.h>
struct node {
    int data;
    struct node* left;
    struct node* right;
};
struct node* newNode(int data)
{
    struct node* node
        = (struct node*)malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return (node);
}
void printPostorder(struct node* node)
{
    if (node == NULL)
        return;

    // First recur on left subtree
    printPostorder(node->left);

    // Then recur on right subtree
    printPostorder(node->right);

    // Now deal with the node
    printf("%d ", node->data);
}

int main()
{
    struct node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    printf("Postorder traversal of binary tree is \n");
    printPostorder(root);

    getchar();
    return 0;
}
```

(a) Traverse Code




Binary Search Tree

***Output:***

*Inorder Traversal: 8 12 20 22 25 30 40*

*Preorder Traversal: 22 12 8 20 30 25 40*

*Postorder Traversal: 8 20 12 25 40 30 22*

(b) Output

Figure 13: Transversal Value of node element

## Traverse Code with Output