Binary Search Tree

A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties −

- The left sub-tree of a node has a key less than or equal to its parent node's key.
- The right sub-tree of a node has a key greater than or equal to its parent node's key.

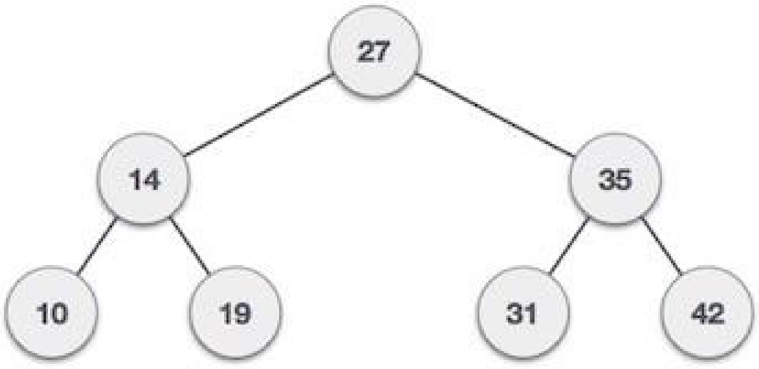
Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as −

```
left_subtree (keys) ≤ node (key) ≤ right_subtree (keys)
```

Binary Tree Representation

BST is a collection of nodes arranged in a way where they maintain BST properties. Each node has a key and an associated value. While searching, the desired key is compared to the keys in BST and if found, the associated value is retrieved.

Following is a pictorial representation of BST −



We observe that the root node key (27) has all less-valued keys on the left sub-tree and the higher valued keys on the right sub-tree.

Basic Operations

Following are the basic operations of a Binary Search Tree −

- **Search** − Searches an element in a tree.
- **Insert** − Inserts an element in a tree.
- **Pre-order Traversal** − Traverses a tree in a pre-order manner.
- **In-order Traversal** − Traverses a tree in an in-order manner.
- **Post-order Traversal** − Traverses a tree in a post-order manner.

Defining a Node

Define a node that stores some data, and references to its left and right child nodes.

```
struct node {
   int data;
   struct node *leftChild;
   struct node *rightChild;
};
```

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

```
1. START
2. Check whether the tree is empty or not
3. If the tree is empty, search is not possible
4. Otherwise, first search the root of the tree.
5. If the key does not match with the value in the root,
   search its subtrees.
6. If the value of the key is less than the root value,
   search the left subtree
7. If the value of the key is greater than the root value,
   search the right subtree.
8. If the key is not found in the tree, return unsuccessful search.
9. END
```

Example

Following are the implementations of this operation in various programming languages −

C C++ Java Python

```c
#include <stdlib.h>
struct node {
   int data;
   struct node *leftChild, *rightChild;
};
struct node *root = NULL;
struct node *newNode(int item){
   struct node *temp = (struct node *)malloc(sizeof(struct node)
   temp->data = item;
   temp->leftChild = temp->rightChild = NULL;
   return temp;
}
void insert(int data){
   struct node *tempNode = (struct node*) malloc(sizeof(struct n
   struct node *current;
   struct node *parent;
   tempNode->leftChild = NULL;
   tempNode->rightChild = NULL;

   //if tree is empty
   if(root == NULL) {
      root = tempNode;
   } else {
      current = root;
      parent = NULL;
      while(1) {
         parent = current;

         //go to left of the tree
         if(data < parent->data) {
            current = current->leftChild;

            //insert to the left
            if(current == NULL) {
               parent->leftChild = tempNode;
               return;
            }
         }//go to right of the tree
         else {
            current = current->rightChild;

            //insert to the right
            if(current == NULL) {
               parent->rightChild = tempNode;
               return;
            }
         }
      }
   }
}
struct node* search(int data){
   struct node *current = root;
   while(current->data != data) {
         //go to left tree
```

```
            current = current->leftChild;
         }//else go to right tree
         else {
            current = current->rightChild;
         }

         //not found
         if(current == NULL) {
            return NULL;
         }
      }
   return current;
}
void printTree(struct node* Node){
   if(Node == NULL)
      return;
   printTree(Node->leftChild);
   printf(" --%d", Node->data);
   printTree(Node->rightChild);
}
int main(){
   insert(55);
   insert(20);
   insert(90);
   insert(50);
   insert(35);
   insert(15);
   insert(65);
   printf("Insertion done");
   printf("\nBST: \n");
   printTree(root);
   struct node* k;
   int ele = 35;
   printf("\nElement to be searched: %d", ele);
   k = search(35);
   if(k != NULL)
      printf("\nElement %d found", k->data);
   else
   return 0;
}
```

Output

```
Insertion done
BST:
 --15 --20 --35 --50 --55 --65 --90
Element to be searched: 35
Element 35 found
```

Insertion Operation

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

```
1. START
2. If the tree is empty, insert the first element as the root node c
   tree. The following elements are added as the leaf nodes.
3. If an element is less than the root value, it is added into the l
   subtree as a leaf node.
4. If an element is greater than the root value, it is added into th
   subtree as a leaf node.
5. The final leaf nodes of the tree point to NULL values as their
   child nodes.
6. END
```

Example

Following are the implementations of this operation in various programming languages −

C | C++ | Java | Python

```
#include <stdio.h>
#include <stdlib.h>
struct node {
   int data;
   struct node *leftChild, *rightChild;
};
struct node *root = NULL;
struct node *newNode(int item){
   struct node *temp = (struct node *)malloc(sizeof(struct node)
   temp->data = item;
   temp->leftChild = temp->rightChild = NULL;
   return temp;
}
void insert(int data){
   struct node *tempNode = (struct node*) malloc(sizeof(struct n
   struct node *current;
   struct node *parent;
   tempNode->data = data;
   tempNode->leftChild = NULL;
   tempNode->rightChild = NULL;

   //if tree is empty
   if(root == NULL) {
      root = tempNode;
   } else {
      current = root;
      parent = NULL;
      while(1) {
         parent = current;

         //go to left of the tree
         if(data < parent->data) {
            current = current->leftChild;
```

```
              //insert to the left
              if(current == NULL) {
                 parent->leftChild = tempNode;
                 return;
              }
           }//go to right of the tree
           else {
              current = current->rightChild;

              //insert to the right
              if(current == NULL) {
                 parent->rightChild = tempNode;
                 return;
              }
           }
        }
     }
  }
  void printTree(struct node* Node){
     if(Node == NULL)
        return;
     printTree(Node->leftChild);
     printf(" --%d", Node->data);
     printTree(Node->rightChild);
  }
  int main(){
     insert(55);
     insert(20);
     insert(90);
     insert(50);
     insert(35);
     insert(15);
     insert(65);
     printf("Insertion done\n");
     printf("BST: \n");
     printTree(root);
     return 0;
```

Output

```
Insertion done
BST:
 --15 --20 --35 --50 --55 --65 --90
```

Inorder Traversal

The inorder traversal operation in a Binary Search Tree visits all its nodes in the following order −

- Firstly, we traverse the left child of the root node/current node, if any.
- Next, traverse the current node.
- Lastly, traverse the right child of the current node, if any.

Algorithm

```
1. START
2. Traverse the left subtree, recursively
3. Then, traverse the root node
4. Traverse the right subtree, recursively.
5. END
```

Example

Following are the implementations of this operation in various programming languages −

C | C++ | Java | Python

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

// Inorder Traversal
void inorder(struct node *root){
   if (root != NULL) {
      inorder(root->left);
      printf("%d -> ", root->key);
      inorder(root->right);
   }
}

// Insertion operation
struct node *insert(struct node *node, int key){
   if (node == NULL) return newNode(key);
   if (key < node->key)
      node->left = insert(node->left, key);
   else
      node->right = insert(node->right, key);
   return node;
}
int main(){
   struct node *root = NULL;
   root = insert(root, 55);
   root = insert(root, 20);
   root = insert(root, 90);
   root = insert(root, 50);
   root = insert(root, 35);
   root = insert(root, 15);
   root = insert(root, 65);
   printf("Inorder traversal: ");
   inorder(root);
```

Output

```
Inorder traversal: 15 -> 20 -> 35 -> 50 -> 55 -> 65 -> 90 ->
```

Preorder Traversal

The preorder traversal operation in a Binary Search Tree visits all its nodes. However, the root node in it is first printed, followed by its left subtree and then its right subtree.

Algorithm

```
1. START
2. Traverse the root node first.
3. Then traverse the left subtree, recursively
4. Later, traverse the right subtree, recursively.
5. END
```

Example

Following are the implementations of this operation in various programming languages −

C C++ Java Python

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

```
}

// Preorder Traversal
void preorder(struct node *root){
   if (root != NULL) {
      printf("%d -> ", root->key);
      preorder(root->left);
      preorder(root->right);
   }
}

// Insertion operation
struct node *insert(struct node *node, int key){
   if (node == NULL) return newNode(key);
   if (key < node->key)
      node->left = insert(node->left, key);
   else
      node->right = insert(node->right, key);
   return node;
}
int main(){
   struct node *root = NULL;
   root = insert(root, 55);
   root = insert(root, 20);
   root = insert(root, 90);
   root = insert(root, 50);
   root = insert(root, 35);
   root = insert(root, 15);
   root = insert(root, 65);
   printf("Preorder traversal: ");
   preorder(root);
```

Output

```
Preorder traversal: 55 -> 20 -> 15 -> 50 -> 35 -> 90 -> 65 ->
```

Postorder Traversal

Like the other traversals, postorder traversal also visits all the nodes in a Binary Search Tree and displays them. However, the left subtree is printed first, followed by the right subtree and lastly, the root node.

Algorithm

```
1. START
2. Traverse the left subtree, recursively
3. Traverse the right subtree, recursively.
4. Then, traverse the root node
5. END
```

Example

Following are the implementations of this operation in various programming languages −

C C++ Java Python

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

// Postorder Traversal
void postorder(struct node *root){
   if (root != NULL) {
        printf("%d -> ", root->key);
        postorder(root->left);
        postorder(root->right);
    }
}

// Insertion operation
struct node *insert(struct node *node, int key){
   if (node == NULL) return newNode(key);
   if (key < node->key)
      node->left = insert(node->left, key);
   else
      node->right = insert(node->right, key);
   return node;
}
int main(){
   struct node *root = NULL;
   root = insert(root, 55);
   root = insert(root, 20);
   root = insert(root, 90);
   root = insert(root, 50);
   root = insert(root, 35);
   root = insert(root, 15);
   root = insert(root, 65);
   printf("Postorder traversal: ");
   postorder(root);
```

Output

```
Postorder traversal: 55 -> 20 -> 15 -> 50 -> 35 -> 90 > 65 ->
```

Complete implementation

Following are the complete implementations of Binary Search Tree in various programming languages –

C C++ Java Python

```
#include <stdlib.h>
struct node {
   int data;
   struct node *leftChild, *rightChild;
};
struct node *root = NULL;
struct node *newNode(int item){
   struct node *temp = (struct node *)malloc(sizeof(struct node)
   temp->data = item;
   temp->leftChild = temp->rightChild = NULL;
   return temp;
}
void insert(int data){
   struct node *tempNode = (struct node*) malloc(sizeof(struct n
   struct node *current;
   struct node *parent;
   tempNode->data = data;
   tempNode->leftChild = NULL;
   tempNode->rightChild = NULL;

   //if tree is empty
   if(root == NULL) {
      root = tempNode;
   } else {
      current = root;
      parent = NULL;
      while(1) {
         parent = current;

         //go to left of the tree
         if(data < parent->data) {
            current = current->leftChild;

            //insert to the left
            if(current == NULL) {
               parent->leftChild = tempNode;
               return;
         }//go to right of the tree
         else {
            current = current->rightChild;

            //insert to the right
            if(current == NULL) {
               parent->rightChild = tempNode;
               return;
            }
         }
      }
   }
}
struct node* search(int data){
   struct node *current = root;
   while(current->data != data) {
      if(current != NULL) {
         //go to left tree
         if(current->data > data) {
            current = current->leftChild;
         }//else go to right tree
         else {
            current = current->rightChild;
         }

         //not found
         if(current == NULL) {
            return NULL;
         }
      }
   }
   return current;
}

// Inorder Traversal
void inorder(struct node *root){
   if (root != NULL) {
      inorder(root->leftChild);
      inorder(root->rightChild);
   }
}

// Preorder Traversal
void preorder(struct node *root){
   if (root != NULL) {
      printf("%d -> ", root->data);
      preorder(root->leftChild);
      preorder(root->rightChild);
   }
}

// Postorder Traversal
void postorder(struct node *root){
   if (root != NULL) {
      printf("%d -> ", root->data);
      postorder(root->leftChild);
      postorder(root->rightChild);
   }
}
int main(){
   insert(55);
   insert(20);
   insert(90);
   insert(50);
   insert(35);
   insert(15);
   insert(65);
   printf("Insertion done");
   printf("\nPreorder Traversal: ");
   preorder(root);
   printf("\nInorder Traversal: ");
   inorder(root);
   printf("\nPostorder Traversal: ");
   postorder(root);
   struct node* k;
   int ele = 35;
   k = search(35);
   if(k != NULL)
      printf("\nElement %d found", k->data);
   else
      printf("\nElement not found");
   return 0;
}
```

Output

```
Insertion done
Preorder Traversal: 55 -> 20 -> 15 -> 50 -> 35 -> 90 -> 65 ->
Inorder Traversal: 15 -> 20 -> 35 -> 50 -> 55 -> 65 -> 90 ->
Postorder Traversal: 55 -> 20 -> 15 -> 50 -> 35 -> 90 -> 65 ->
Element to be searched: 35
Element 35 found
```