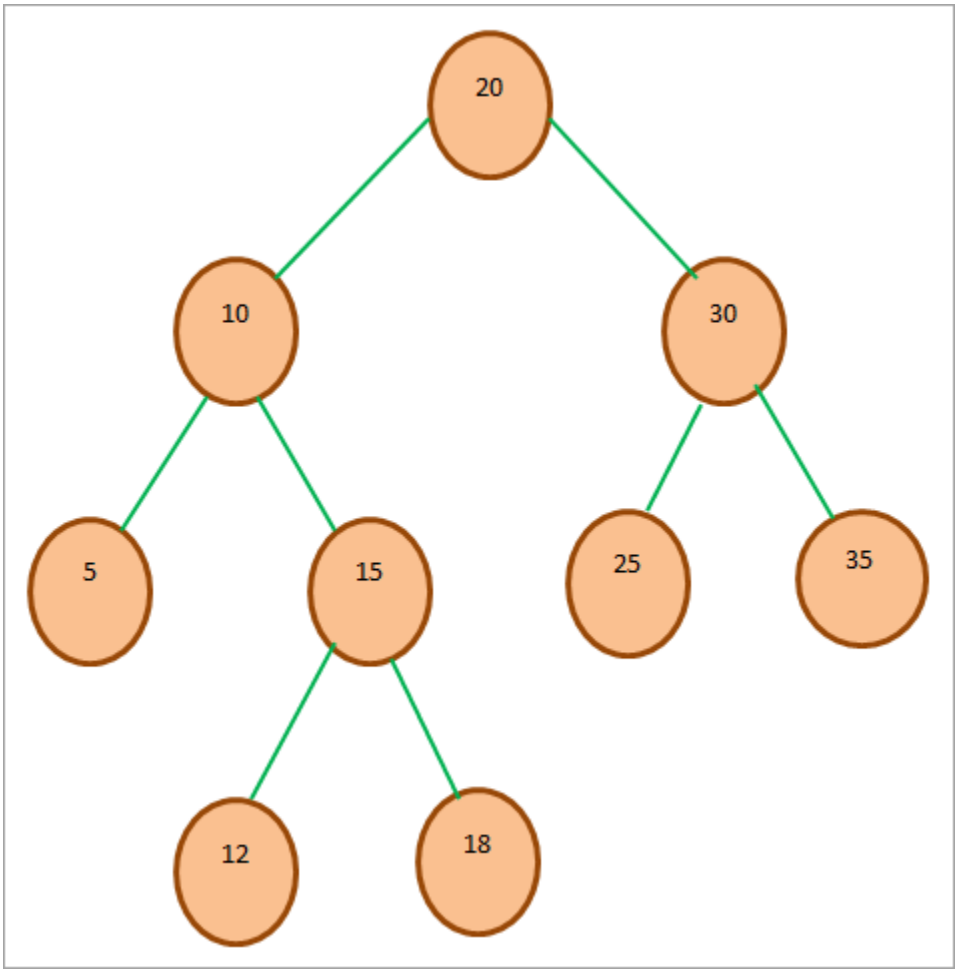
KHWAJA FAREED UEIT RAHIM YAR KHAN | Faculty of Information Technology

Binary Search Tree in Java

Binary Search Tree in Java

A BST does not allow duplicate nodes.

The below diagram shows a BST Representation:



Above shown is a sample BST. We see that 20 is the root node of this tree. The left subtree has all the node values that are less than 20. The right subtree has all the nodes that are greater than 20. We can say that the above tree fulfills the BST properties.

The BST data structure is considered to be very efficient when compared to Arrays and Linked list when it comes to insertion/deletion and searching of items.

BST takes O (log n) time to search for an element. As elements are ordered, half the subtree is discarded at every step while searching for an element. This becomes possible because we can easily determine the rough location of the element to be searched.

Similarly, insertion and deletion operations are more efficient in BST. When we want to insert a new element, we roughly know in which subtree (left or right) we will insert the element.

Creating A Binary Search Tree (BST)

Given an array of elements, we need to construct a BST.

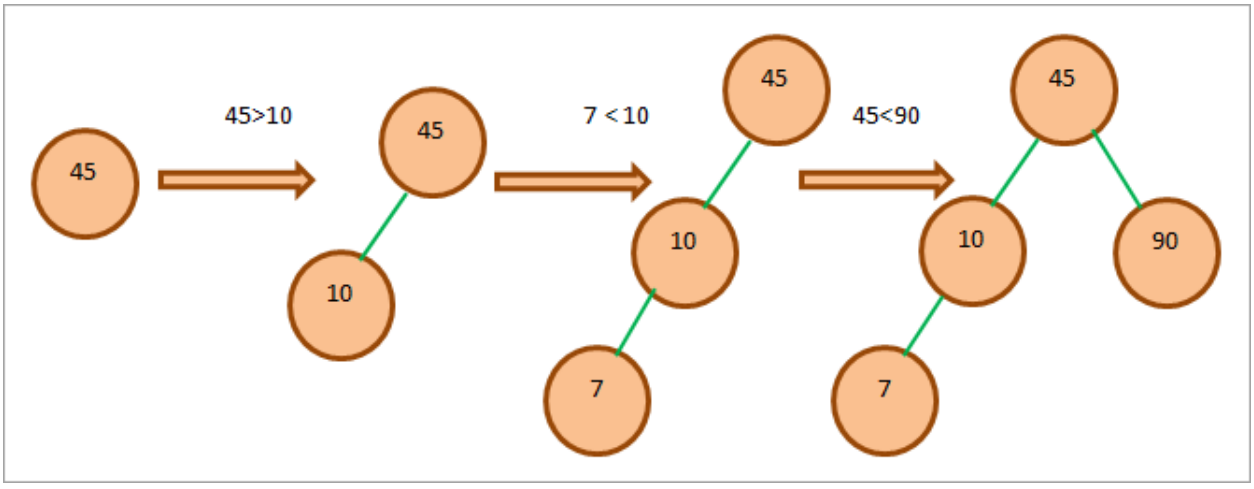
Let's do this as shown below:

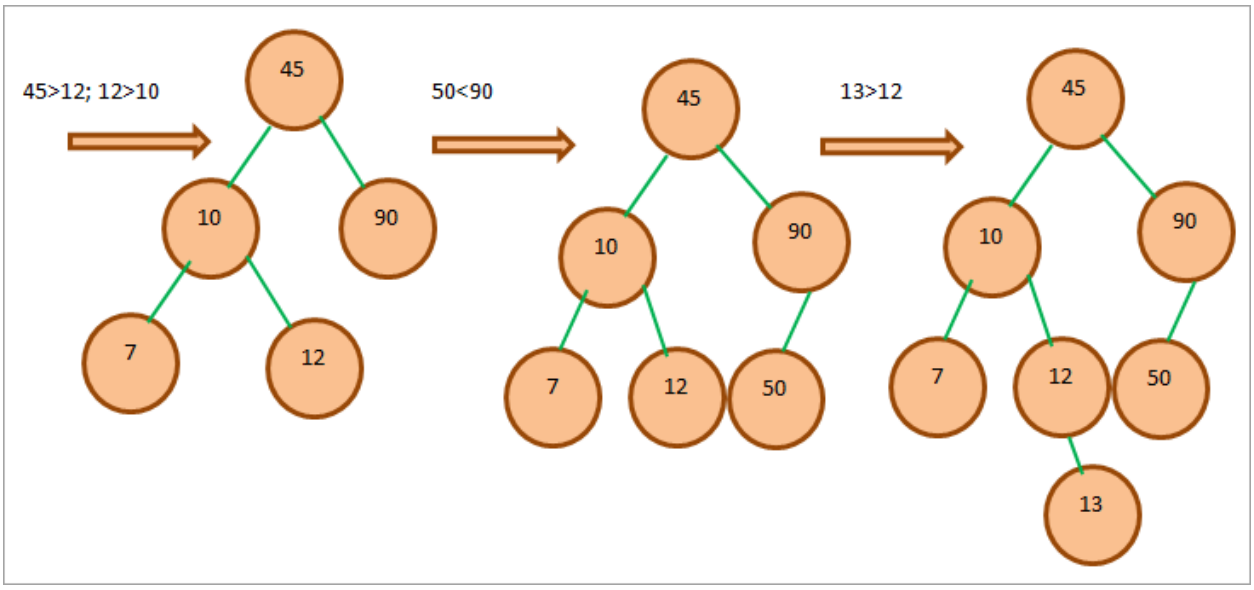
Given array: 45, 10, 7, 90, 12, 50, 13, 39, 57

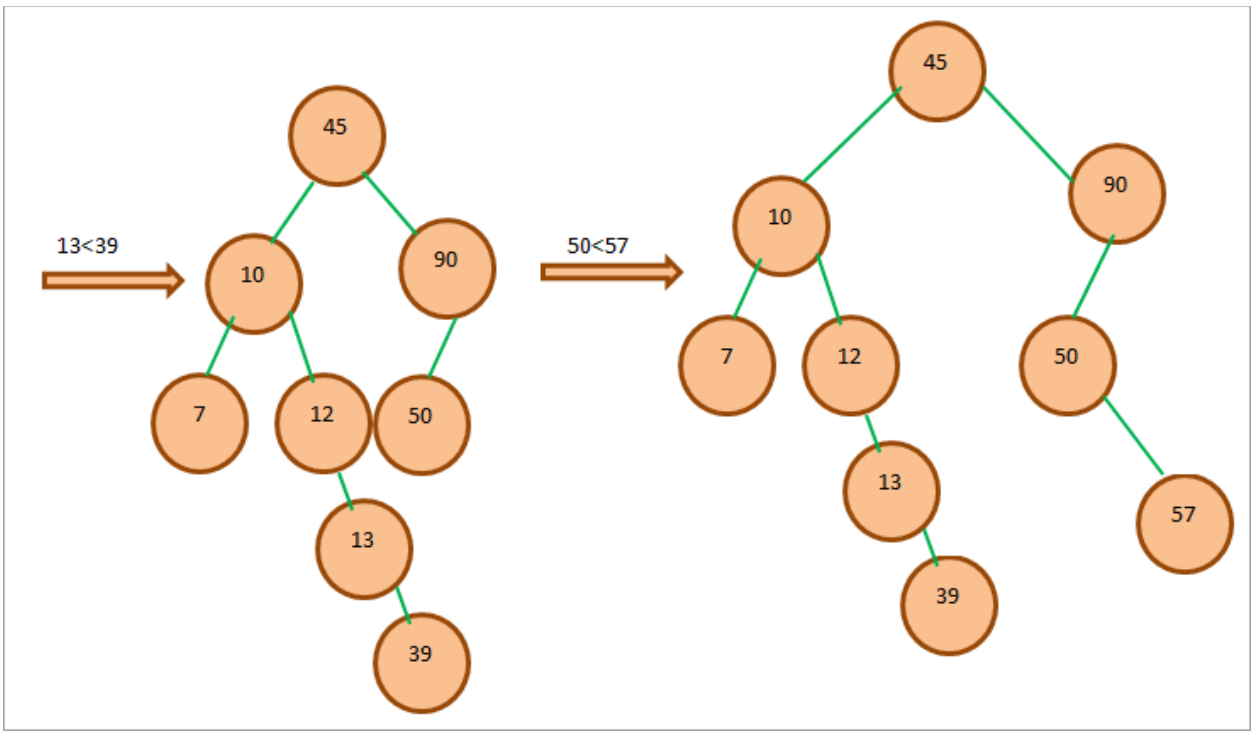
Let's first consider the top element i.e. 45 as the root node. From here we will go on creating the BST by considering the properties already discussed.

To create a tree, we will compare each element in the array with the root. Then we will place the element at an appropriate position in the tree.

The entire creation process for BST is shown below.







Completed BST

Binary Search Tree Operations

BST supports various operations. The following table shows the methods supported by BST in Java. We will discuss each of these methods separately.

Method/operation	Description
Insert	Add an element to the BST by not violating the BST properties.
Delete	Remove a given node from the BST. The node can be the root node, non-leaf, or leaf node.
Search	Search the location of the given element in the BST. This operation checks if the tree contains the specified key.

Insert An Element In BST

An element is always inserted as a leaf node in BST.

Given below are the steps for inserting an element.

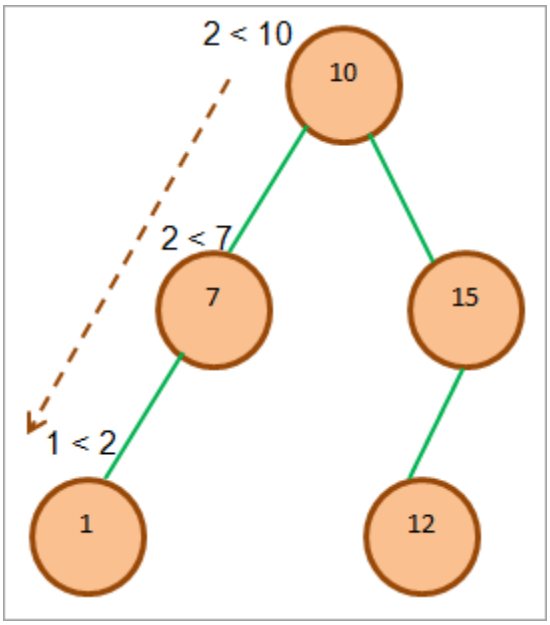
Start from the root.

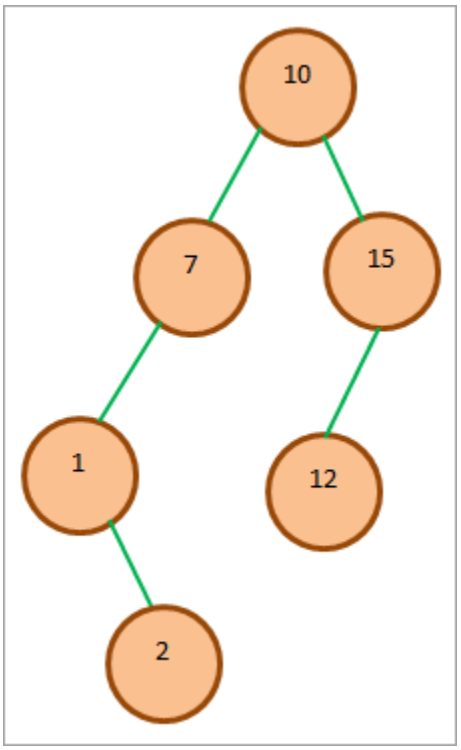
Compare the element to be inserted with the root node. If it is less than root, then traverse the left subtree or traverse the right subtree.

Traverse the subtree till the end of the desired subtree. Insert the node in the appropriate subtree as a leaf node.

Let’s see an illustration of the insert operation of BST.

Consider the following BST and let us insert element 2 in the tree.





The insert operation for BST is shown above. In fig (1), we show the path that we traverse to insert element 2 in the BST. We have also shown the conditions that are checked at each node. As a result of the recursive comparison, element 2 is inserted as the right child of 1 as shown in fig (2).

Search Operation In BST

To search if an element is present in the BST, we again start from the root and then traverse the left or right subtree depending on whether the element to be searched is less than or greater than the root.

Enlisted below are the steps that we have to follow.

Compare the element to be searched with the root node.

If the key (element to be searched) = root, return root node.

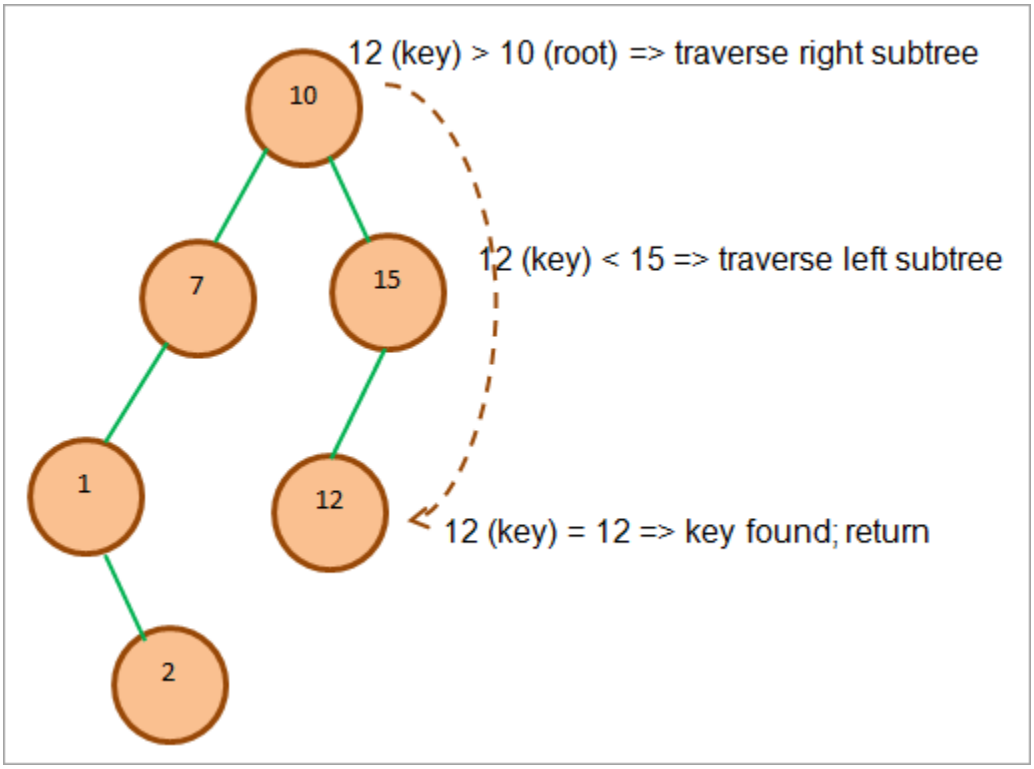
Else if key < root, traverse the left subtree.

Else traverse right subtree.

Repetitively compare subtree elements until the key is found or the end of the tree is reached.

Let's illustrate the search operation with an example. Consider that we have to search the key = 12.

In the below figure, we will trace the path we follow to search for this element.



As shown in the above figure, we first compare the key with root. Since the key is greater, we traverse the right subtree. In the right subtree, we again compare the key with the first node in the right subtree.

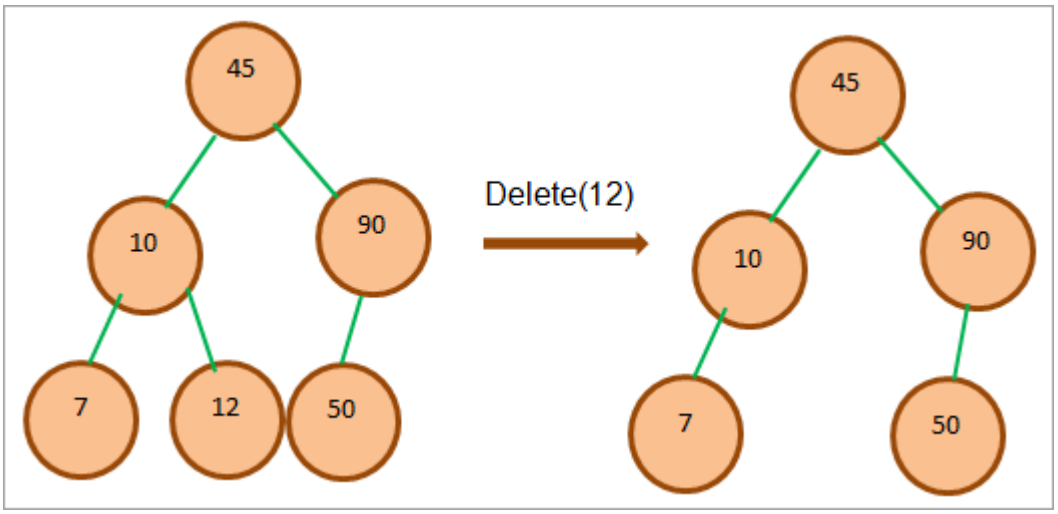
We find that the key is less than 15. So we move to the left subtree of node 15. The immediate left node of 15 is 12 that matches the key. At this point, we stop the search and return the result.

Remove Element From The BST

When we delete a node from the BST, then there are three possibilities as discussed below:

Node Is A Leaf Node

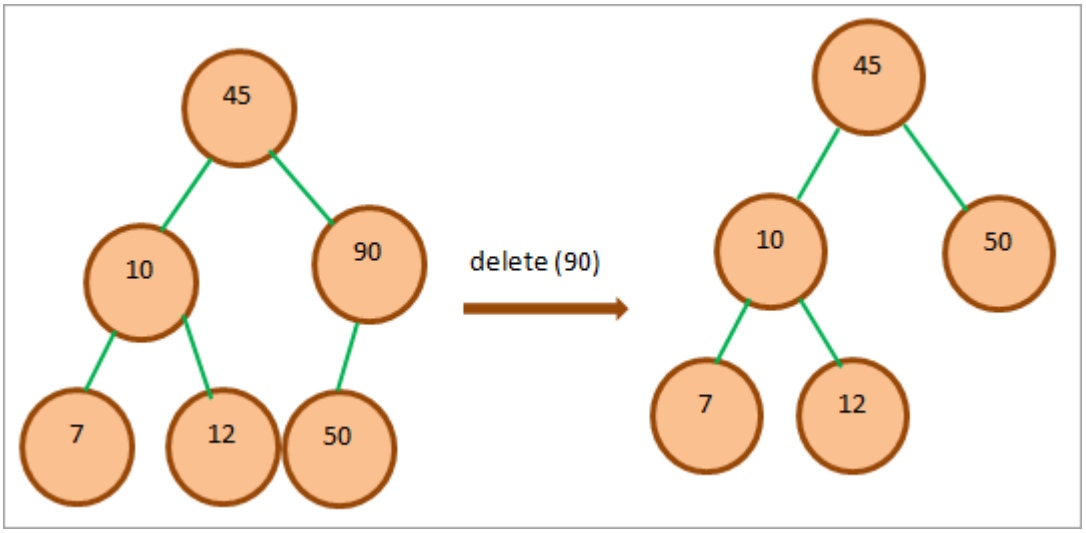
If a node to be deleted is a leaf node, then we can directly delete this node as it has no child nodes. This is shown in the below image.



As shown above, the node 12 is a leaf node and can be deleted straight away.

Node Has Only One Child

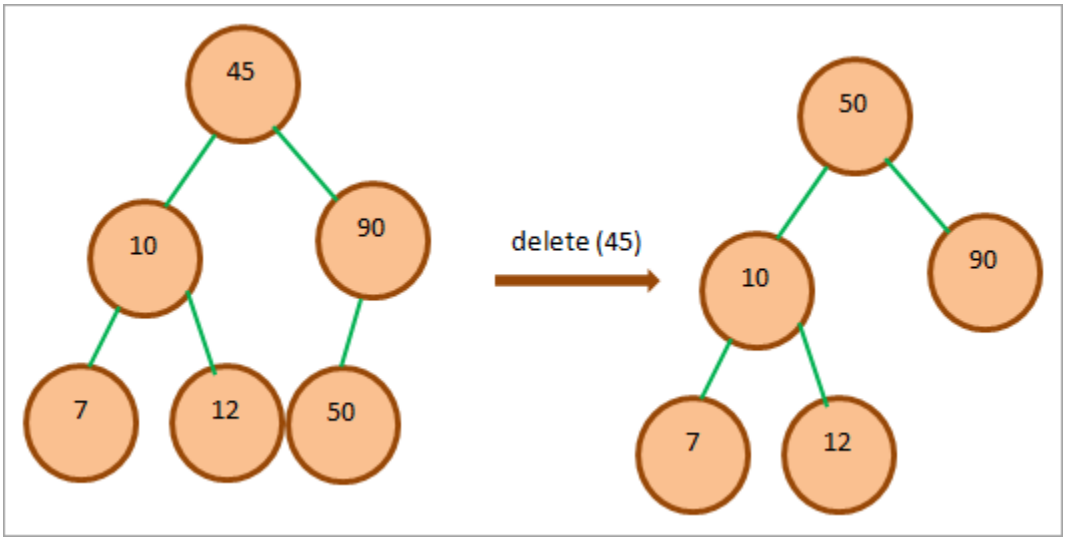
When we need to delete the node that has one child, then we copy the value of the child in the node and then delete the child.



In the above diagram, we want to delete node 90 which has one child 50. So we swap the value 50 with 90 and then delete node 90 which is a child node now.

Node Has Two Children

When a node to be deleted has two children, then we replace the node with the inorder (left-root-right) successor of the node or simply said the minimum node in the right subtree if the right subtree of the node is not empty. We replace the node with this minimum node and delete the node.



In the above diagram, we want to delete node 45 which is the root node of BST. We find that the right subtree of this node is not empty. Then we traverse the right subtree and find that node 50 is the minimum node here. So we replace this value in place of 45 and then delete 45.

If we check the tree, we see that it fulfills the properties of a BST. Thus the node replacement was correct.

Binary Search Tree (BST) Implementation In Java

The following program in Java provides a demonstration of all the above BST operation using the same tree used in illustration as an example.

```
class BST_class {
    //node class that defines BST node
    class Node {
        int key;
        Node left, right;
        public Node(int data){
            key = data;
            left = right = null;
        }
    }
    // BST root node
    Node root;
   // Constructor for BST =>initial empty tree
    BST_class(){
        root = null;
    }
    //delete a node from BST
    void deleteKey(int key) {
        root = delete_Recursive(root, key);
    }
    //recursive delete function
    Node delete_Recursive(Node root, int key)  {
        //tree is empty
        if (root == null)  return root;
        //traverse the tree
```

```
//traverse the tree
    if (key < root.key)     //traverse left subtree
        root.left = delete_Recursive(root.left, key);
    else if (key > root.key)  //traverse right subtree
        root.right = delete_Recursive(root.right, key);
    else  {
        // node contains only one child
        if (root.left == null)
            return root.right;
        else if (root.right == null)
            return root.left;

        // node has two children;
        //get inorder successor (min value in the right subtree)
        root.key = minValue(root.right);

        // Delete the inorder successor
        root.right = delete_Recursive(root.right, root.key);
    }
    return root;
  }
```

```
int minValue(Node root)  {
      //initially minval = root
      int minval = root.key;
      //find minval
      while (root.left != null)  {
         minval = root.left.key;
         root = root.left;
      }
      return minval;
   }
   // insert a node in BST
   void insert(int key)  {
      root = insert_Recursive(root, key);
   }
   //recursive insert function
   Node insert_Recursive(Node root, int key) {
       //tree is empty
      if (root == null) {
         root = new Node(key);
         return root;
      }
      //traverse the tree
      if (key < root.key)     //insert in the left subtree
         root.left = insert_Recursive(root.left, key);
```

```
else if (key > root.key)    //insert in the right subtree
            root.right = insert_Recursive(root.right, key);
          // return pointer
        return root;
}
 // method for inorder traversal of BST
    void inorder() {
        inorder_Recursive(root);
    }
   // recursively traverse the BST
    void inorder_Recursive(Node root) {
        if (root != null) {
            inorder_Recursive(root.left);
            System.out.print(root.key + " ");
            inorder_Recursive(root.right);
        }
    }
boolean search(int key)  {
        root = search_Recursive(root, key);
        if (root!= null)
            return true;
        else
            return false;
    }
```

```
    //recursive insert function
    Node search_Recursive(Node root, int key)  {
        // Base Cases: root is null or key is present at root
        if (root==null || root.key==key)
            return root;
        // val is greater than root's key
        if (root.key > key)
            return search_Recursive(root.left, key);
        // val is less than root's key
        return search_Recursive(root.right, key);
    }
}
```

```
class Main{
  public static void main(String[] args)  {
    //create a BST object
     BST_class bst = new BST_class();
     /* BST tree example
          45
        /    \
       10      90
      / \    /
     7   12  50   */
     //insert data into BST
     bst.insert(45);
     bst.insert(10);
     bst.insert(7);
     bst.insert(12);
     bst.insert(90);
     bst.insert(50);
     //print the BST
     System.out.println("The BST Created with input data(Left-root-right):");
        bst.inorder();

     //delete leaf node
```

```
        System.out.println("\nThe BST after Delete 12(leaf node):");
        bst.deleteKey(12);
        bst.inorder();
        //delete the node with one child
        System.out.println("\nThe BST after Delete 90 (node with 1 child):");
        bst.deleteKey(90);
        bst.inorder();

        //delete node with two children
        System.out.println("\nThe BST after Delete 45 (Node with two children):");
        bst.deleteKey(45);
        bst.inorder();
        //search a key in the BST
        boolean ret_val = bst.search (50);
        System.out.println("\nKey 50 found in BST:" + ret_val );
        ret_val = bst.search (12);
        System.out.println("\nKey 12 found in BST:" + ret_val );
    }
}
```

Output:

```
The BST Created with input data(Left-root-
right):
7 10 12 45 50 90

The BST after Delete 12(leaf node):

7 10 45 50 90

The BST after Delete 90 (node with 1 child):

7 10 45 50

The BST after Delete 45 (Node with two children):

7 10 50

Key 50 found in BST:true
```