Topics to cover:

1. Binary search tree in Java

1. Binary search tree

In this article, we will discuss the Binary search tree. This article will be very helpful and informative to the students with technical background as it is an important topic of their course.

Before moving directly to the binary search tree, let's first see a brief description of the tree.

What is a tree?

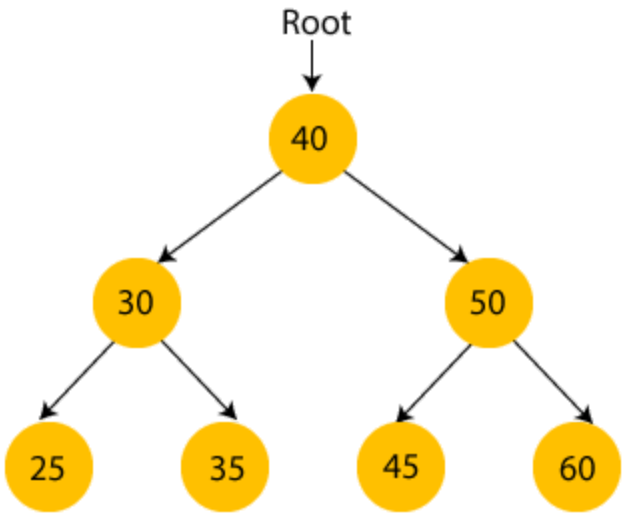
A tree is a kind of data structure that is used to represent the data in hierarchical form. It can be defined as a collection of objects or entities called as nodes that are linked together to simulate a hierarchy. Tree is a non-linear data structure as the data in a tree is not stored linearly or sequentially.

Now, let's start the topic, the Binary Search tree.

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 the 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.

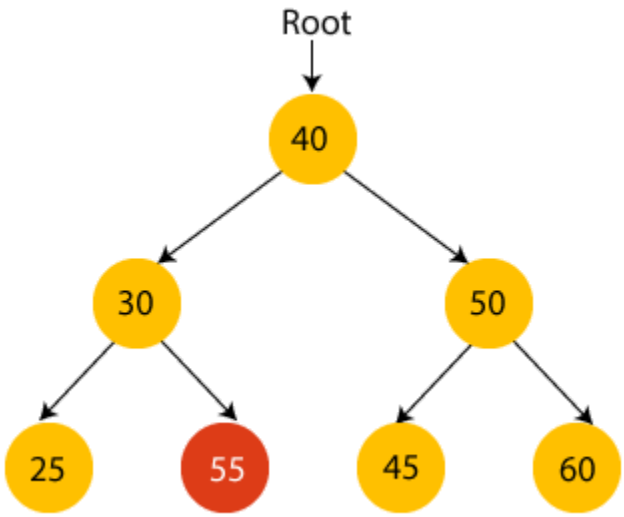
Let's understand the concept of Binary search tree with an example.



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.



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.

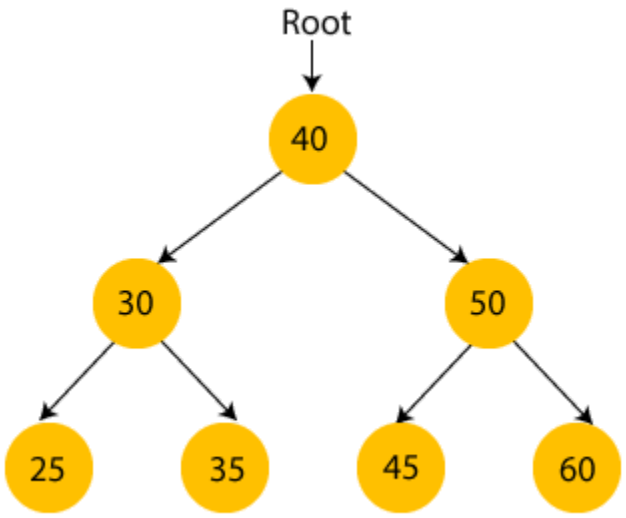
Advantages of Binary search tree

- Searching an element in the Binary search tree is easy as we always have a hint that which subtree has the desired element.
- As compared to array and linked lists, insertion and deletion operations are faster in BST.

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.

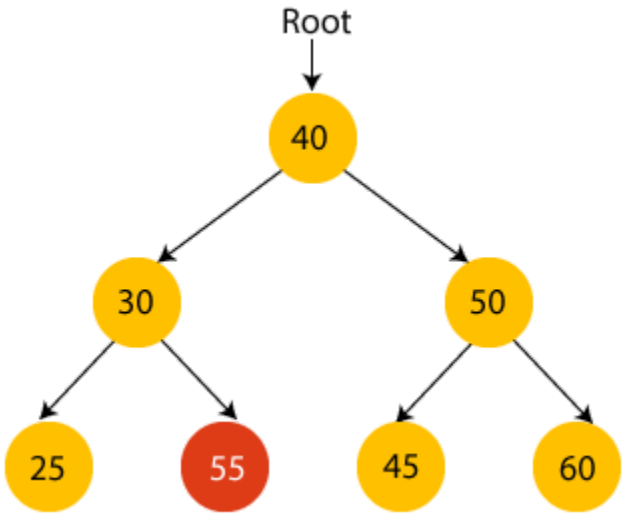
Let's understand the concept of Binary search tree with an example.



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.



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.

Advantages of Binary search tree

- Searching an element in the Binary search tree is easy as we always have a hint that which subtree has the desired element.
- As compared to array and linked lists, insertion and deletion operations are faster in BST.

Example of creating a binary search tree

Now, let's see the creation of binary search tree using an example.

Suppose the data elements are **- 45, 15, 79, 90, 10, 55, 12, 20, 50**

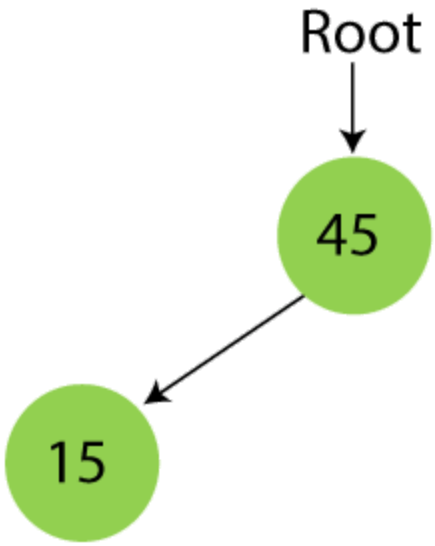
- First, we have to insert **45** into the tree as the root of the tree.
- Then, read the next element; if it is smaller than the root node, insert it as the root of the left subtree, and move to the next element.
- Otherwise, if the element is larger than the root node, then insert it as the root of the right subtree.

Now, let's see the process of creating the Binary search tree using the given data element. The process of creating the BST is shown below -

Step 1 - Insert 45.

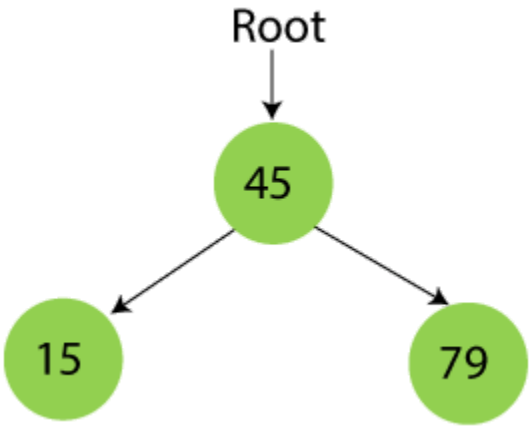
Step 2 - Insert 15.

As 15 is smaller than 45, so insert it as the root node of the left subtree.



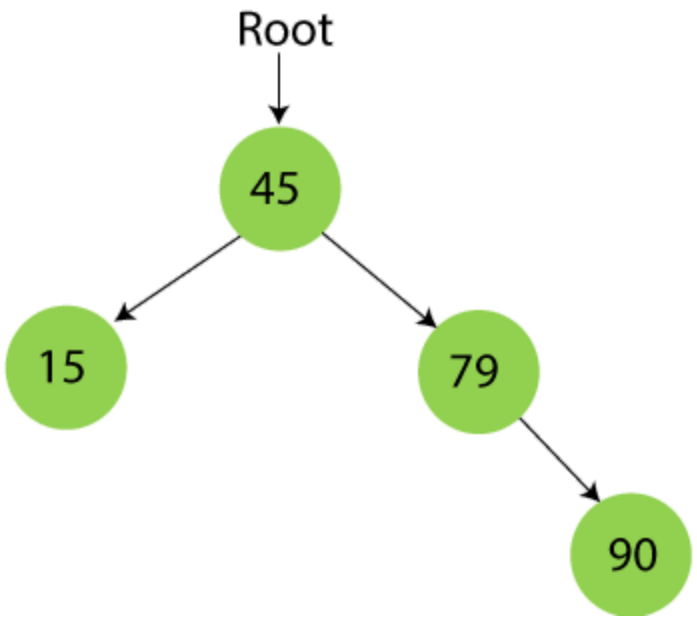
Step 3 - Insert 79.

As 79 is greater than 45, so insert it as the root node of the right subtree.



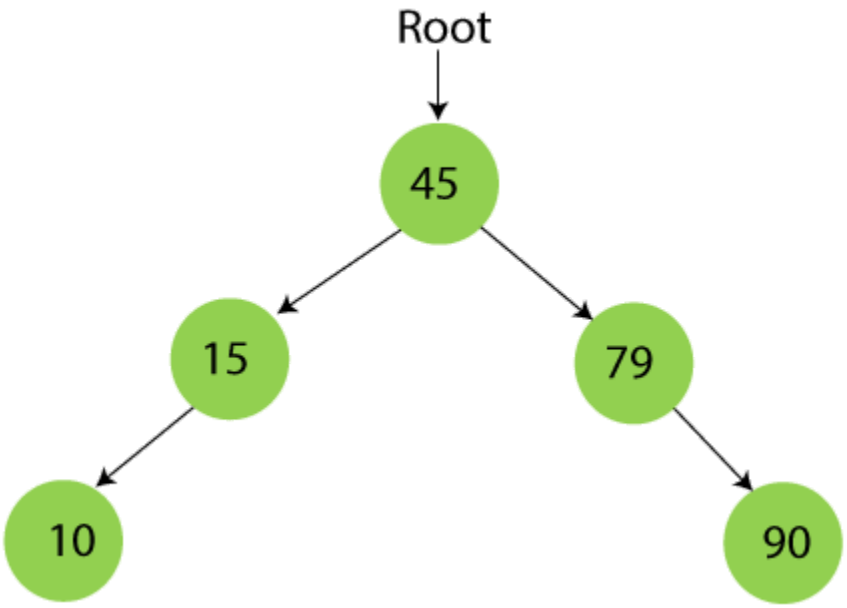
Step 4 - Insert 90.

90 is greater than 45 and 79, so it will be inserted as the right subtree of 79.



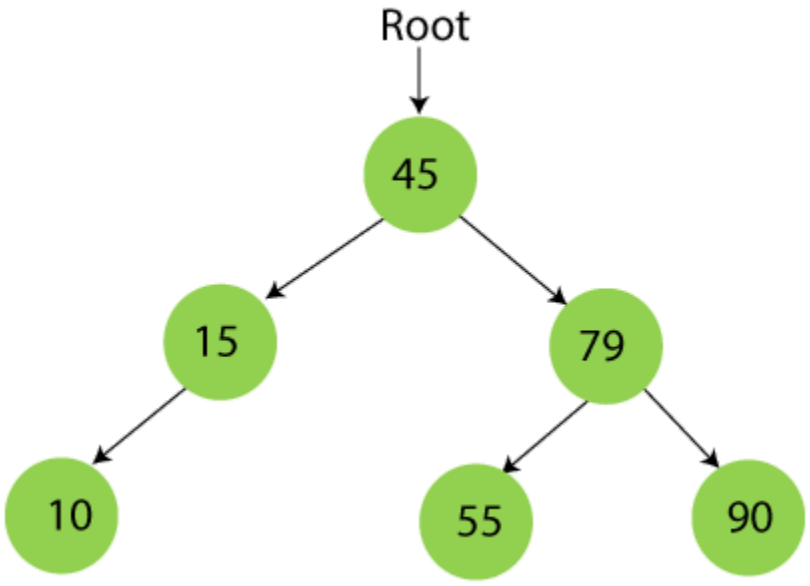
Step 5 - Insert 10.

10 is smaller than 45 and 15, so it will be inserted as a left subtree of 15.



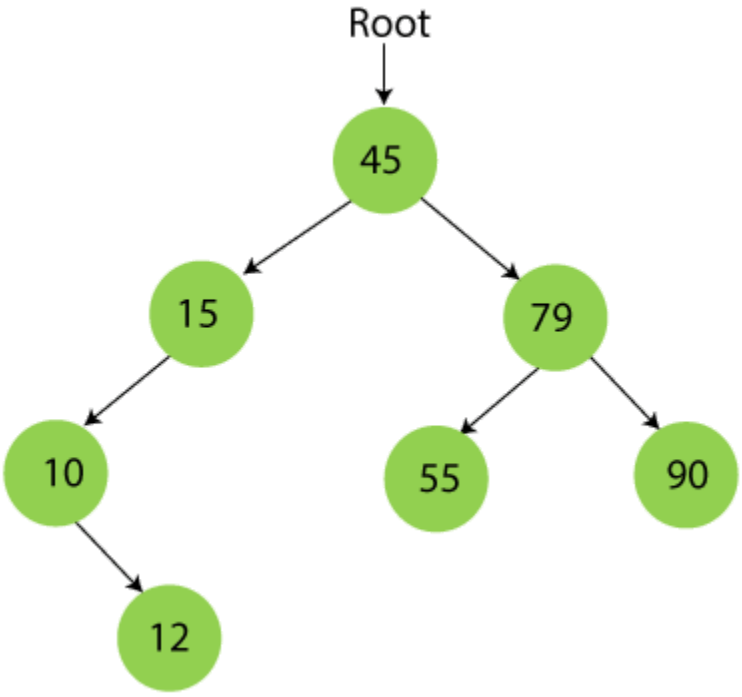
Step 6 - Insert 55.

55 is larger than 45 and smaller than 79, so it will be inserted as the left subtree of 79.



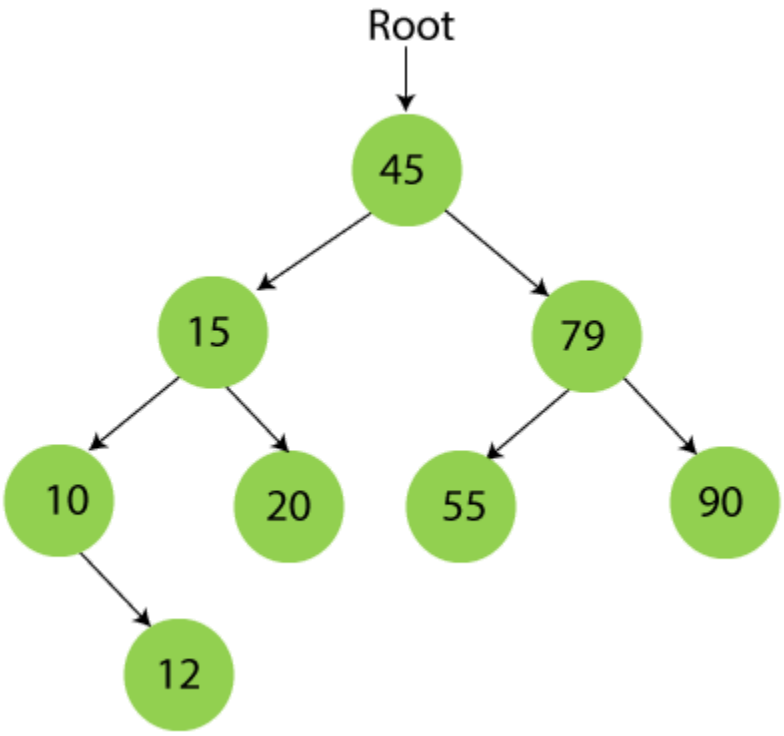
Step 7 - Insert 12.

12 is smaller than 45 and 15 but greater than 10, so it will be inserted as the right subtree of 10.



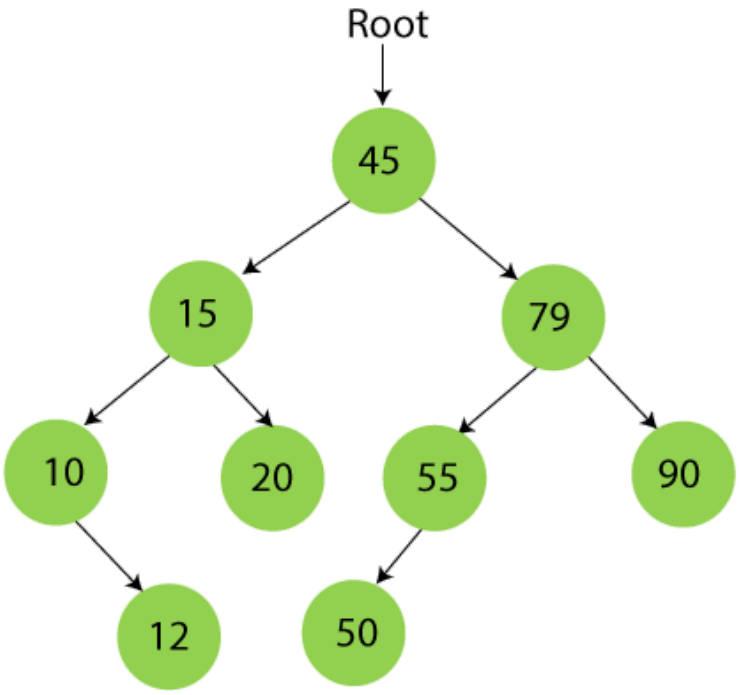
Step 8 - Insert 20.

20 is smaller than 45 but greater than 15, so it will be inserted as the right subtree of 15.



Step 9 - Insert 50.

50 is greater than 45 but smaller than 79 and 55. So, it will be inserted as a left subtree of 55.



Now, the creation of a binary search tree is completed. After that, let's move towards the operations that can be performed on the Binary search tree.

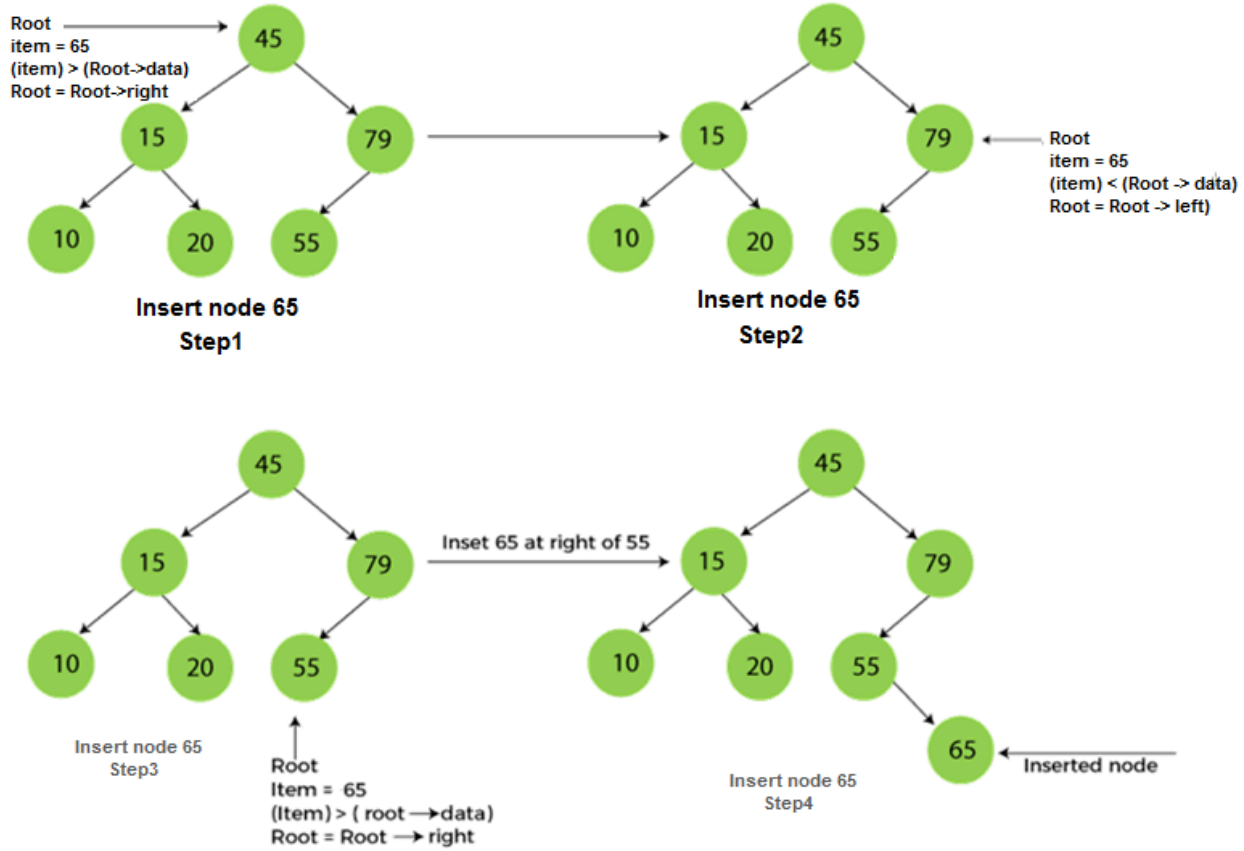
We can perform insert, delete and search operations on the binary search tree.

Let's understand how a search is performed on a binary search tree.

Insertion in Binary Search tree

A new key in BST is always inserted at the leaf. To insert an element in BST, we have to start searching from the root node; if the node to be inserted is less than the root node, then search for an empty location in the left subtree. Else, search for the empty location in the right subtree and insert the data. Insert in BST is similar to searching, as we always have to maintain the rule that the left subtree is smaller than the root, and right subtree is larger than the root.

Now, let's see the process of inserting a node into BST using an example.



Program 1: Create a Binary search tree

```
public class BSTClass {
        static class Node{
                int data;
                Node left;
                Node right;

                Node(int data){
                        this.data=data;
                        System.out.println("Node to insert: "+data);
                }
        }
```

```
public static Node insertNode(Node root,int data) {
	if(root==null) {
		root = new Node(data);
	}
	else if (data<root.data) {
		root.left = insertNode(root.left,data);
	}
	else if (data>root.data) {
		root.right = insertNode(root.right,data);
	}

	//System.out.println("current root: "+root.data);
	return root;
}

public static void inorderTraverse(Node root) {
	if(root==null) {
		return;
	}
	inorderTraverse(root.left);
	System.out.print("Current Node: "+root.data);
	if(root.left!=null) {
		System.out.print("\tLeft: "+root.left.data);
	}
	else {
		System.out.print("\tLeft: "+null);
	}
	if(root.right!=null) {
		System.out.print("\tRight: "+root.right.data);
	}
	else {
		System.out.print("\tRight: "+null);
	}
	System.out.println();
	inorderTraverse(root.right);
}

public static void main(String[] args) {
	Node root=null;

	int a[] = {3,5,1,4,2};
	for(int n:a) {
		root = insertNode(root,n);
```

```
            }
            System.out.println("Inorder traversal:");
            inorderTraverse(root);
        }
}
```

Output :

```
Node to insert: 3
Node to insert: 5
Node to insert: 1
Node to insert: 4
Node to insert: 2
Inorder traversal:
Current Node: 1     Left: null     Right: 2
Current Node: 2     Left: null     Right: null
Current Node: 3     Left: 1  Right: 5
Current Node: 4     Left: null     Right: null
Current Node: 5     Left: 4  Right: null
```

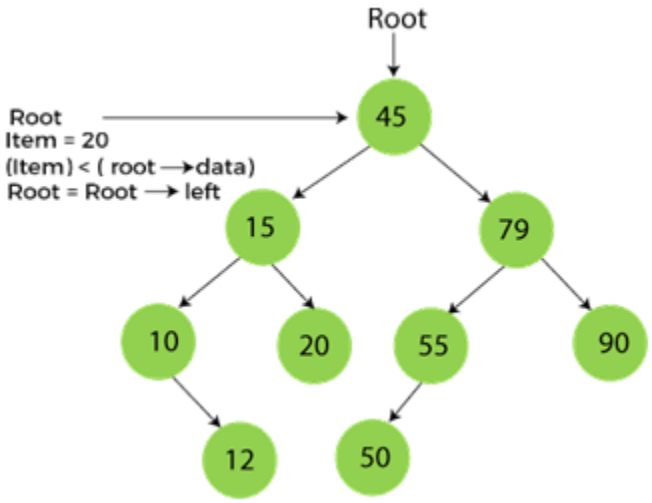
Searching in Binary search tree

Searching means to find or locate a specific element or node in a data structure. In Binary search tree, searching a node is easy because elements in BST are stored in a specific order. The steps of searching a node in Binary Search tree are listed as follows -

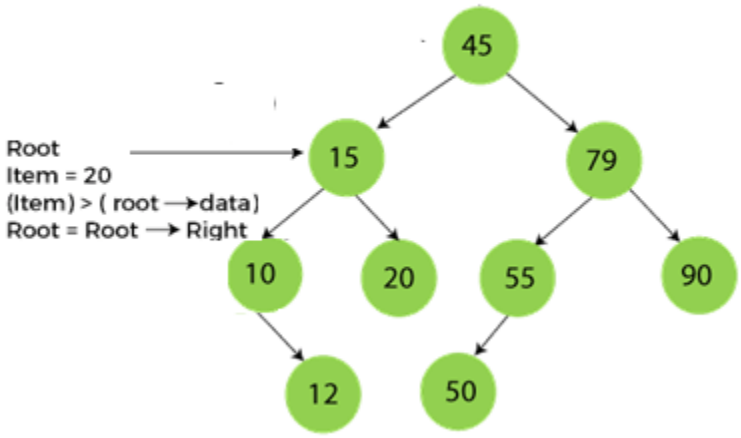
1. First, compare the element to be searched with the root element of the tree.
2. If the root is matched with the target element, then return the node's location.
3. If it is not matched, then check whether the item is less than the root element, if it is smaller than the root element, then move to the left subtree.
4. If it is larger than the root element, then move to the right subtree.
5. Repeat the above procedure recursively until the match is found.
6. If the element is not found or not present in the tree, then return NULL.

Now, let's understand the searching in binary tree using an example. We are taking the binary search tree formed above. Suppose we have to find node 20 from the below tree.

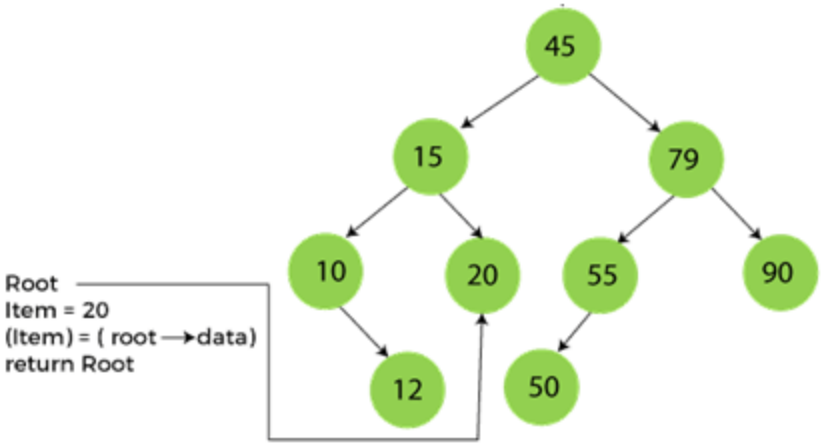
Step1:



Step2:



Step3:



Now, let's see the algorithm to search an element in the Binary search tree.

Now let's understand how the deletion is performed on a binary search tree. We will also see an example to delete an element from the given tree.

Program 2: Searching item in a Binary search tree

```
//We just need to add the following method to program 1 and call the method to search an element.

…..   // the dots mean the previous part of the following method in the whole program
public static boolean searchTree(Node root, int key) {
                if(root==null) {
                        return false;
                }
                else {
                        if(key<root.data) {
                                return searchTree(root.left, key);
                        }
                        else if(key==root.data) {
                                return true;
```

```
                    }
                    else {
                            return searchTree(root.right, key);
                    }
            }
    }
….. //the dots mean the following part of the above method in the whole program
```

Output :

…. // the output is very easy, just call the method and you can see the output

Deletion in Binary Search tree

In a binary search tree, we must delete a node from the tree by keeping in mind that the property of BST is not violated. To delete a node from BST, there are three possible situations occur -

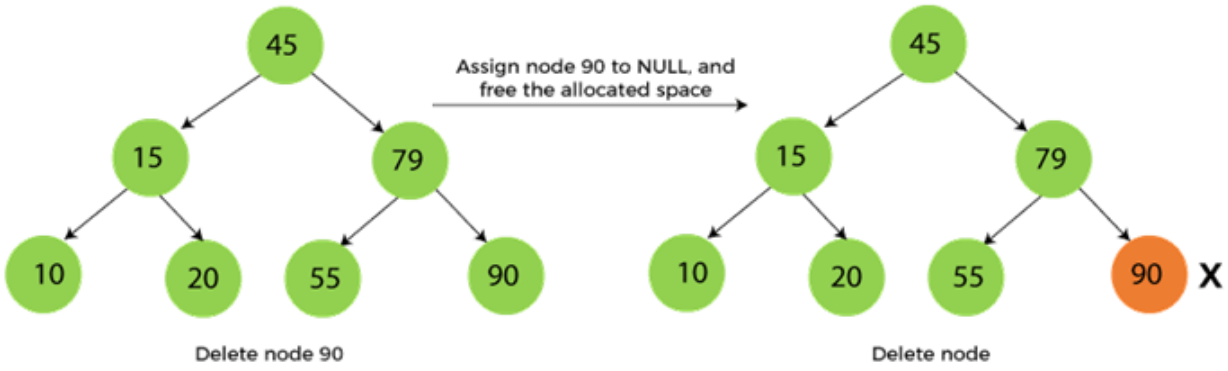
- The node to be deleted is the leaf node, or,
- The node to be deleted has only one child, and,
- The node to be deleted has two children

We will understand the situations listed above in detail.

When the node to be deleted is the leaf node

It is the simplest case to delete a node in BST. Here, we have to replace the leaf node with NULL and simply free the allocated space.

We can see the process to delete a leaf node from BST in the below image. In below image, suppose we have to delete node 90, as the node to be deleted is a leaf node, so it will be replaced with NULL, and the allocated space will free.

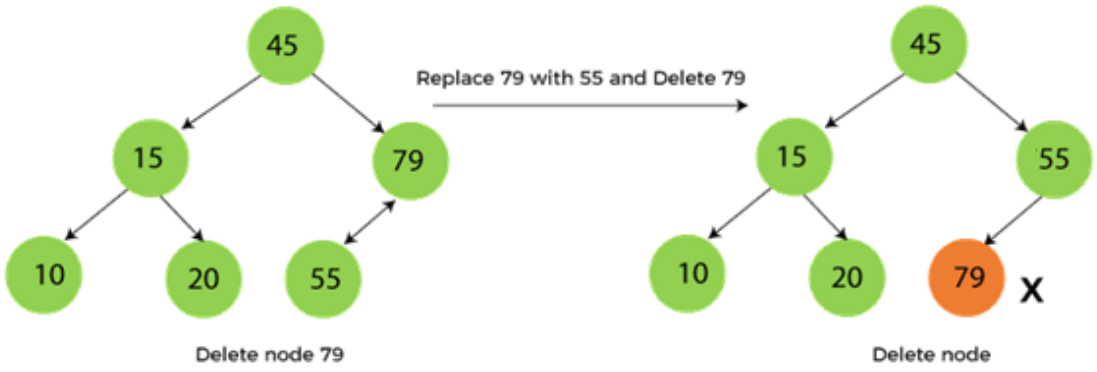


When the node to be deleted has only one child

In this case, we have to replace the target node with its child, and then delete the child node. It means that after replacing the target node with its child node, the child node will now contain the value to be deleted. So, we simply have to replace the child node with NULL and free up the allocated space.

We can see the process of deleting a node with one child from BST in the below image. In the below image, suppose we have to delete the node 79, as the node to be deleted has only one child, so it will be replaced with its child 55.

So, the replaced node 79 will now be a leaf node that can be easily deleted.



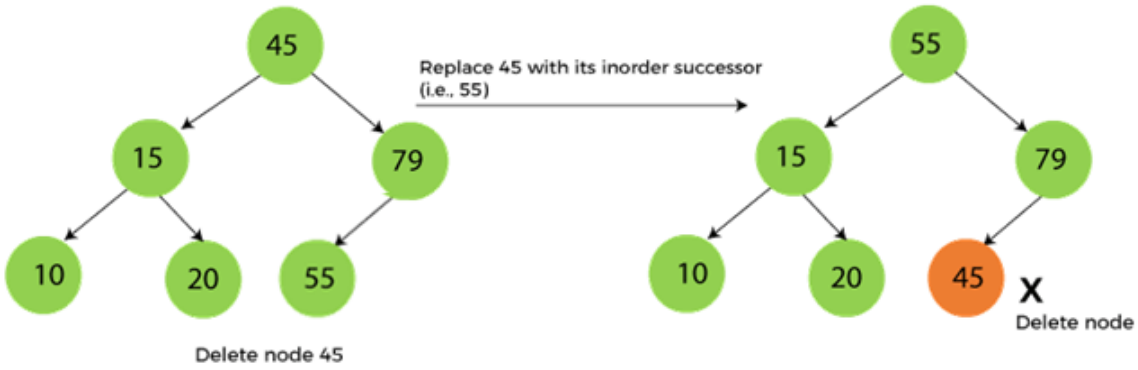
When the node to be deleted has two children

This case of deleting a node in BST is a bit complex among other two cases. In such a case, the steps to be followed are listed as follows -

- First, find the inorder successor of the node to be deleted.
- After that, replace that node with the inorder successor until the target node is placed at the leaf of tree.
- And at last, replace the node with NULL and free up the allocated space.

The inorder successor is required when the right child of the node is not empty. We can obtain the inorder successor by finding the minimum element in the right child of the node.

We can see the process of deleting a node with two children from BST in the below image. In the below image, suppose we have to delete node 45 that is the root node, as the node to be deleted has two children, so it will be replaced with its inorder successor. Now, node 45 will be at the leaf of the tree so that it can be deleted easily.



Now let's understand how insertion is performed on a binary search tree.

Program 3: Deleting item from a Binary search tree

```
///We need to add the following two methods in the whole program

….//The previous part

        public static Node deleteNode(Node root, int key) {
                if(key<root.data) {
```

```
                root.left=deleteNode(root.left,key);
            }
            else if(key>root.data) {
                root.right=deleteNode(root.right,key);
            }

            else {
                if(root.right==null && root.left==null) {
                    return null;
                }
                if(root.left==null) {
                    return root.right;
                }
                else if(root.right==null) {
                    return root.left;
                }
                Node inSc=inorderSuccessor(root.right);
                root.data = inSc.data;
                root.right=deleteNode(root.right,inSc.data);
            }

            return root;
        }

        public static Node inorderSuccessor(Node root) {
            while(root.left!=null) {
                root=root.left;
            }
            return root;
        }
….// The following part
```

Output :

…. // the output is very easy, just call the method and you can see the output

The complexity of the Binary Search tree

Let's see the time and space complexity of the Binary search tree. We will see the time complexity for insertion, deletion, and searching operations in best case, average case, and worst case.

1. Time Complexity

Operations	Best case time complexity	Average case time complexity	Worst case time complexity
Insertion	O(log n)	O(log n)	O(n)
Deletion	O(log n)	O(log n)	O(n)
Search	O(log n)	O(log n)	O(n)

Where 'n' is the number of nodes in the given tree.

2. Space Complexity

Operations	Space complexity
Insertion	O(n)
Deletion	O(n)
Search	O(n)

- The space complexity of all operations of Binary search tree is O(n).

Implementation of Binary search tree

Now, let's see the program to implement the operations of Binary Search tree.

Program: Write a program to perform operations of Binary Search tree in C++.

In this program, we will see the implementation of the operations of binary search tree. Here, we will see the creation, inorder traversal, insertion, and deletion operations of tree.

Here, we will see the inorder traversal of the tree to check whether the nodes of the tree are in their proper location or not. We know that the inorder traversal always gives us the data in ascending order. So, after performing the insertion and deletion operations, we perform the inorder traversal, and after traversing, if we get data in ascending order, then it is clear that the nodes are in their proper location.