

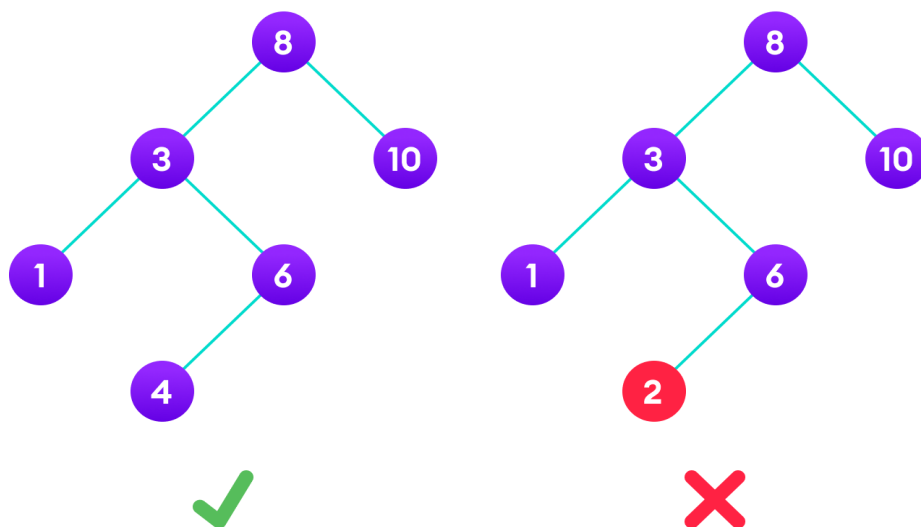
Binary Search Tree(BST)

Binary search tree is a data structure that quickly allows us to maintain a sorted list of numbers.

- It is called a binary tree because each tree node has a maximum of two children.
- It is called a search tree because it can be used to search for the presence of a number in $O(\log(n))$ time.

The properties that separate a binary search tree from a regular [binary tree](#) is

1. All nodes of left subtree are less than the root node
2. All nodes of right subtree are more than the root node
3. Both subtrees of each node are also BSTs i.e. they have the above two properties



A tree having a right subtree with one value smaller than the root is shown to demonstrate that it is not a valid binary search tree

The binary tree on the right isn't a binary search tree because the right subtree of the node "3" contains a value smaller than it.

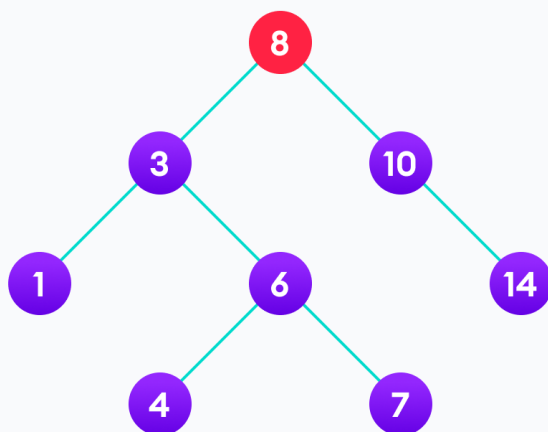
There are two basic operations that you can perform on a binary search tree:

Search Operation

The algorithm depends on the property of BST that if each left subtree has values below root and each right subtree has values above the root.

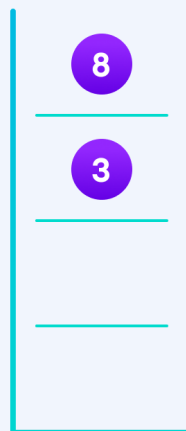
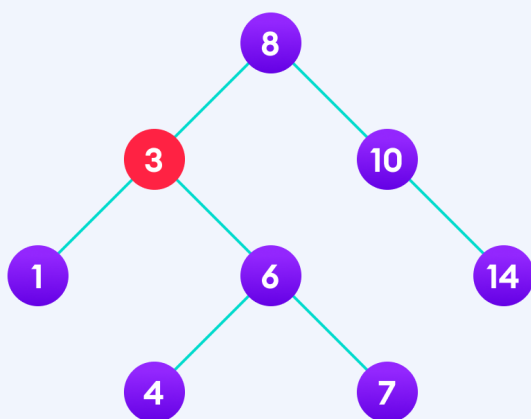
If the value is below the root, we can say for sure that the value is not in the right subtree; we need to only search in the left subtree and if the value is above the root, we can say for sure that the value is not in the left subtree; we need to only search in the right subtree.

Let us try to visualize this with a diagram.

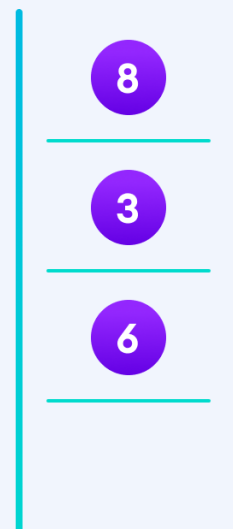
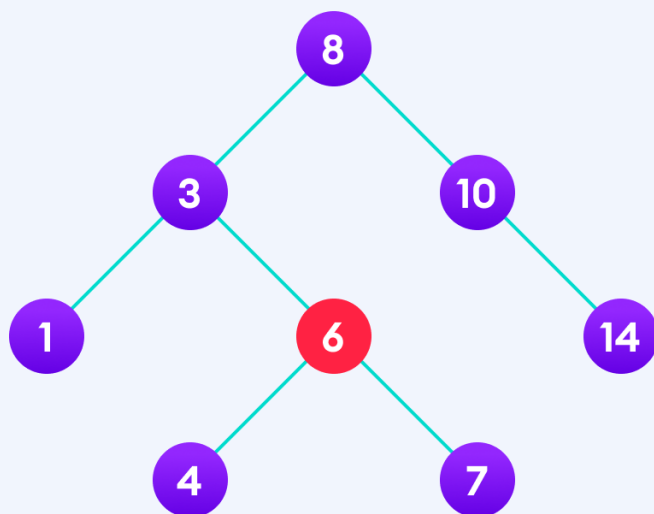


4 is not found

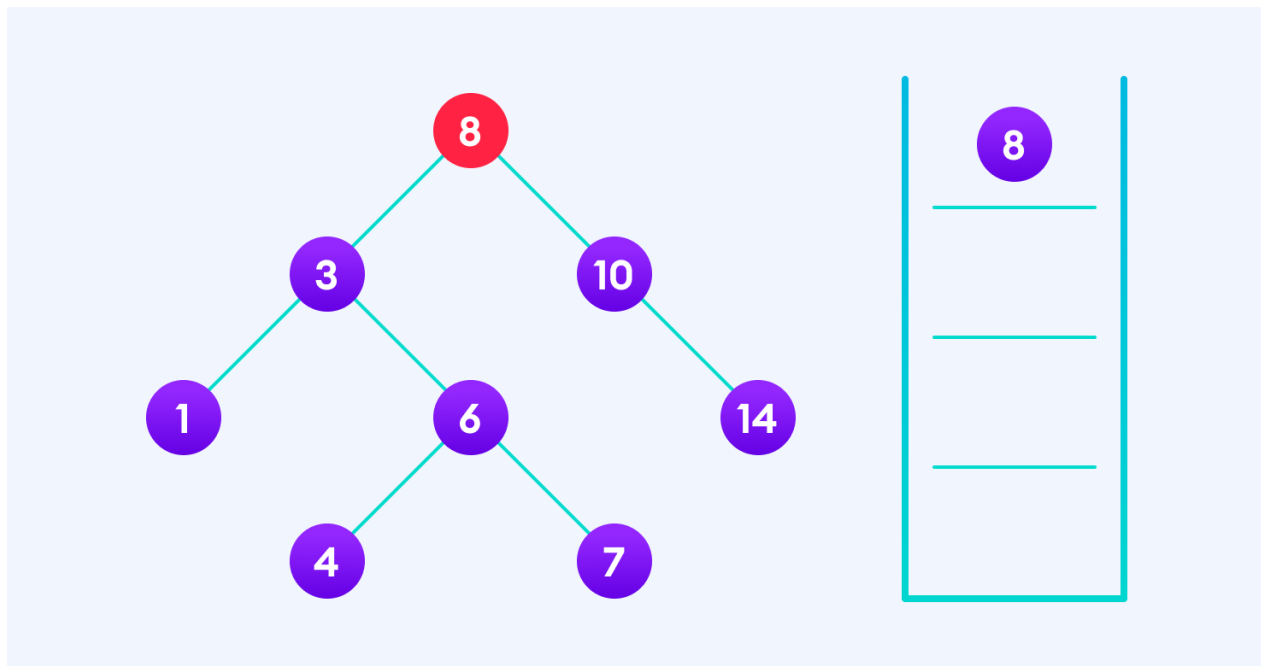
so, traverse through the left subtree of 8



4 is not found so,
traverse through the right subtree of 3



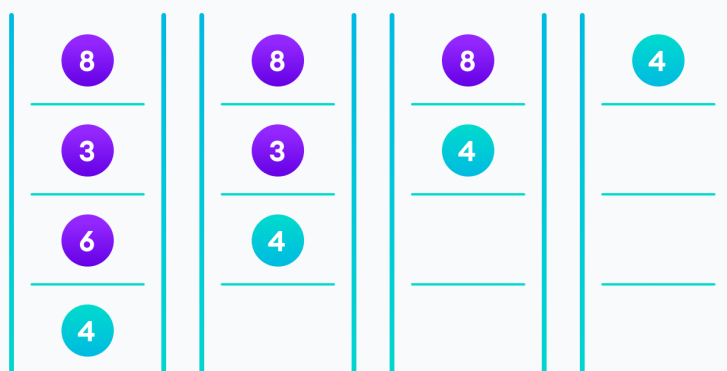
4 is not found so, traverse through the left subtree of 6



4 is found

If the value is found, we return the value so that it gets propagated in each recursion step as shown in the image below.

If you might have noticed, we have called `return search(struct node*)` four times. When we return either the new node or `NULL`, the value gets returned again and again until `search(root)` returns the final result.



If the value is found in any of the subtrees, it is propagated up so that in the end it is returned, otherwise null is returned

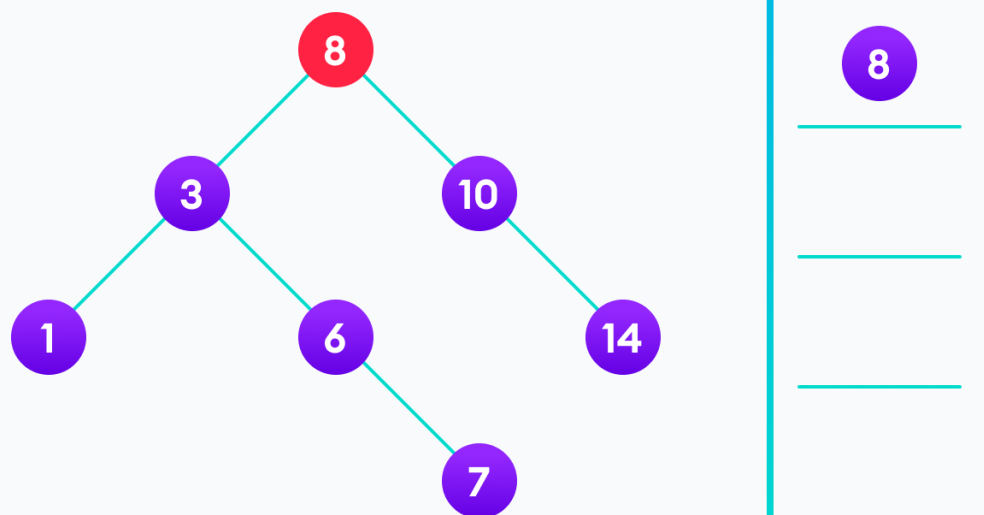
If the value is not found, we eventually reach the left or right child of a leaf node which is NULL and it gets propagated and returned.

Insert Operation

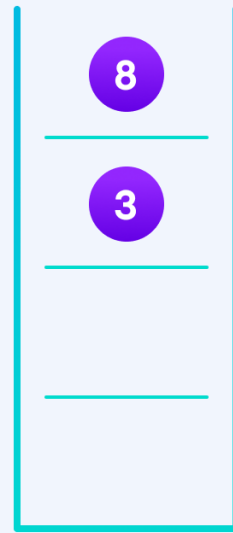
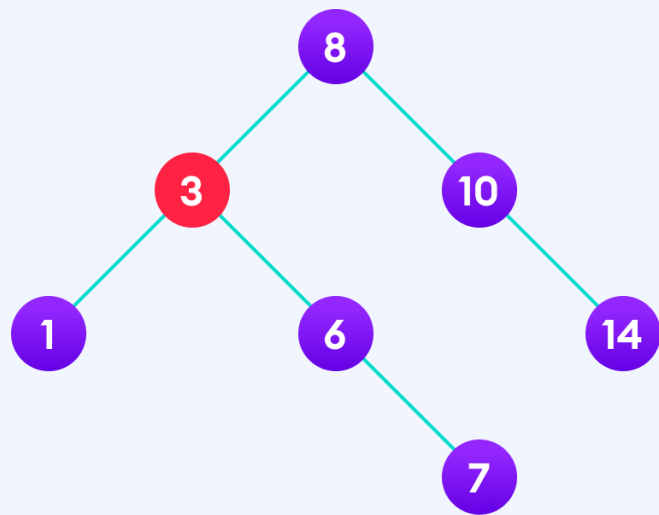
Inserting a value in the correct position is similar to searching because we try to maintain the rule that the left subtree is lesser than root and the right subtree is larger than root.

We keep going to either right subtree or left subtree depending on the value and when we reach a point left or right subtree is null, we put the new node there.

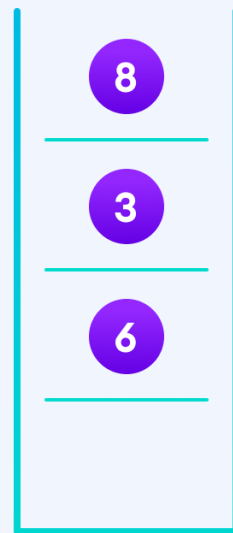
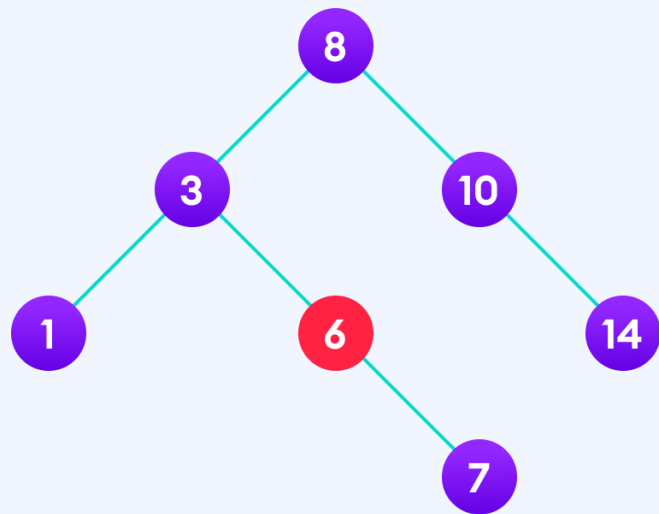
The algorithm isn't as simple as it looks. Let's try to visualize how we add a number to an existing BST.



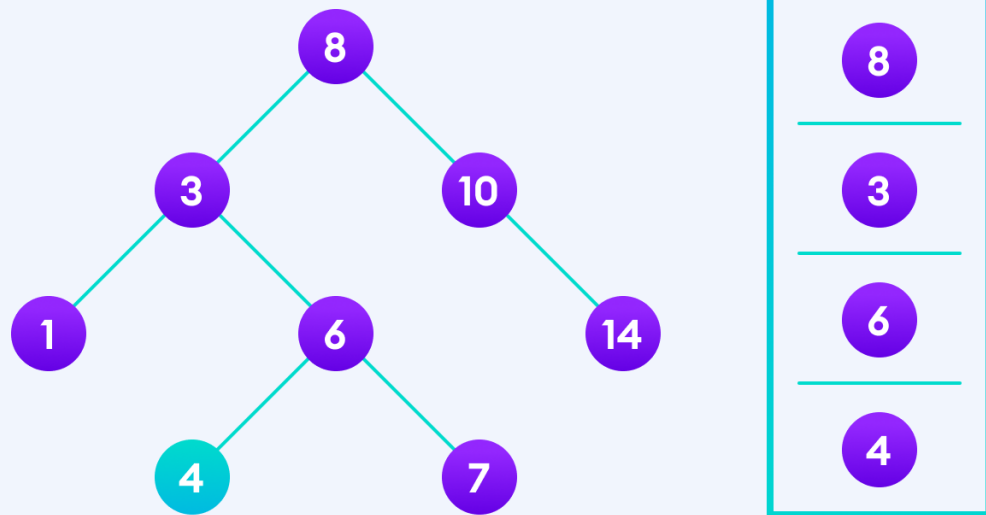
4<8 so, transverse through the left child of 8



4>3 so, transverse through the right child of 8



4<6 so, transverse through the left child of 6



Insert 4 as a left child of 6

We have attached the node but we still have to exit from the function without doing any damage to the rest of the tree. This is where the `return node;` at the end comes in handy. In the case of `NULL`, the newly created node is returned and attached to the parent node, otherwise the same node is returned without any change as we go up until we return to the root.

This makes sure that as we move back up the tree, the other node connections aren't changed.

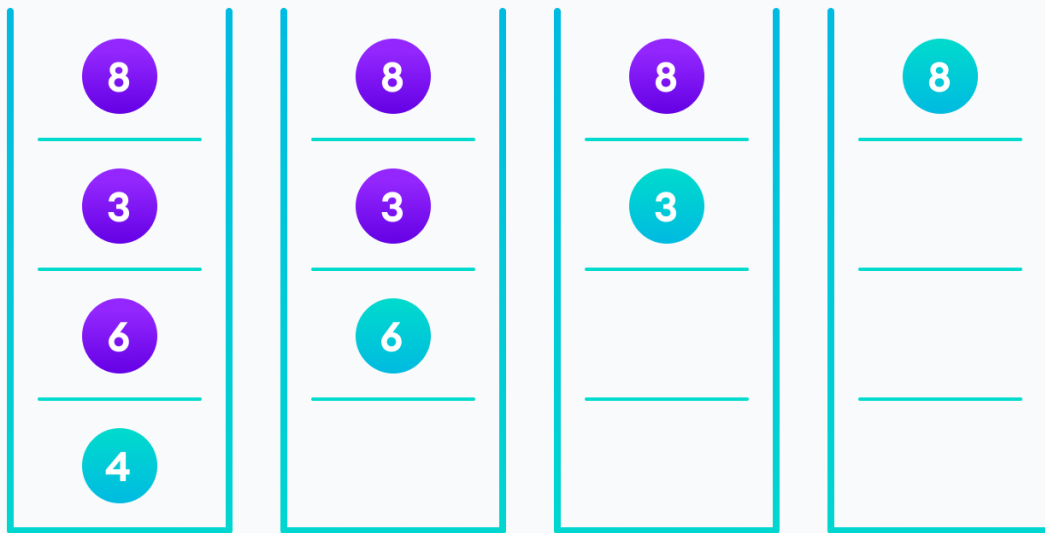


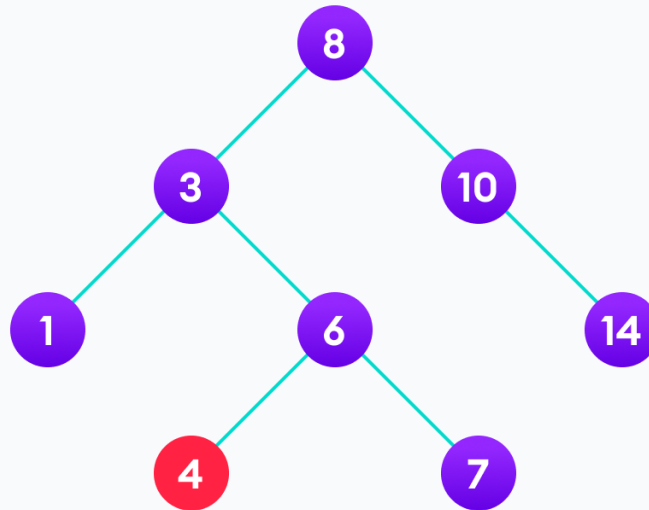
Image showing the importance of returning the root element at the end so that the elements don't lose their position during the upward recursion step.

Deletion Operation

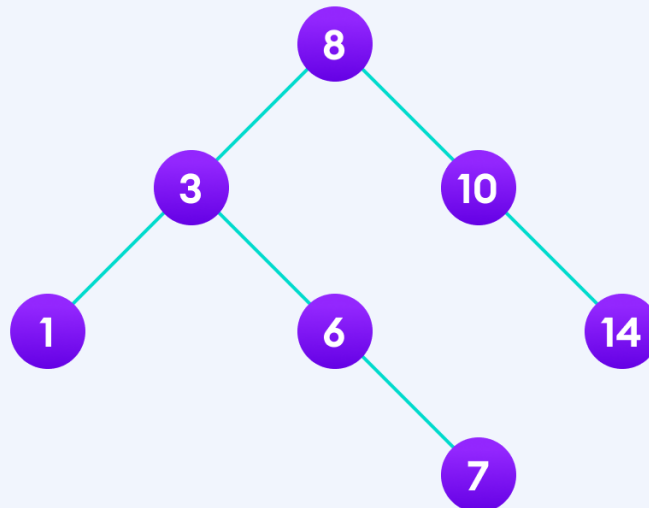
There are three cases for deleting a node from a binary search tree.

Case I

In the first case, the node to be deleted is the leaf node. In such a case, simply delete the node from the tree.



4 is to be deleted



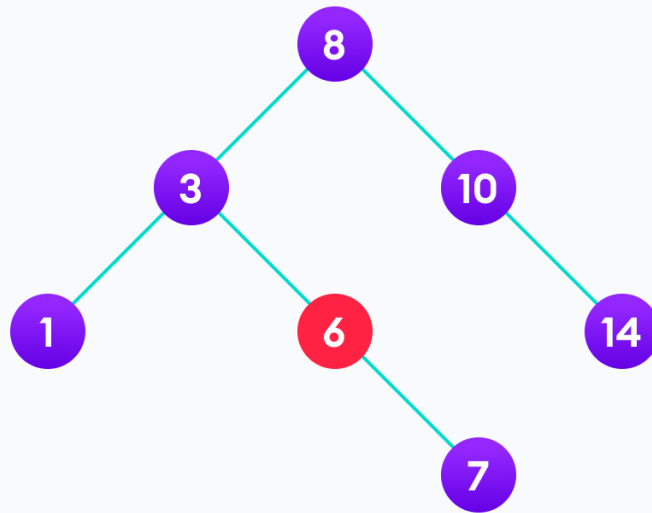
Delete the node

Case II

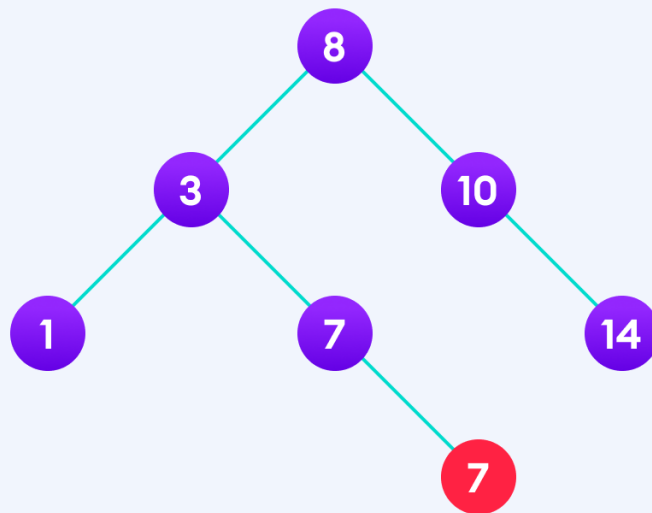
In the second case, the node to be deleted lies has a single child node. In such a case follow the steps below:

1. Replace that node with its child node.

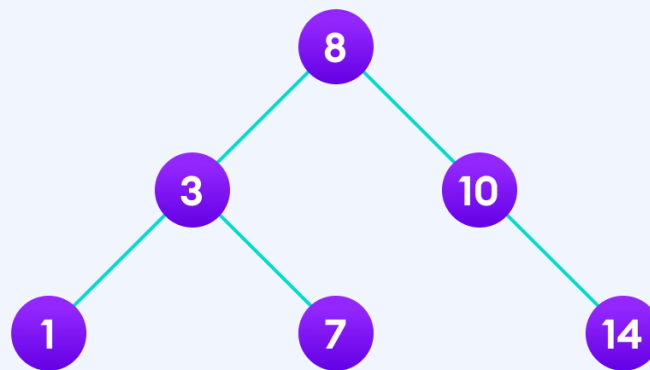
2. Remove the child node from its original position.



6 is to be deleted



copy the value of its child to the node and delete the child

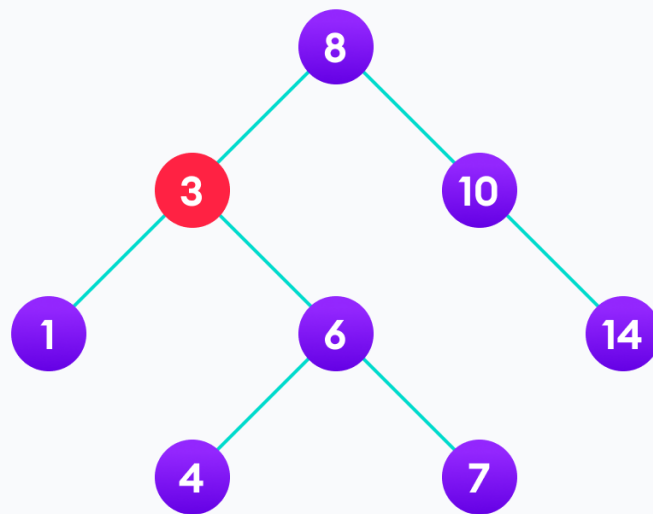


Final tree

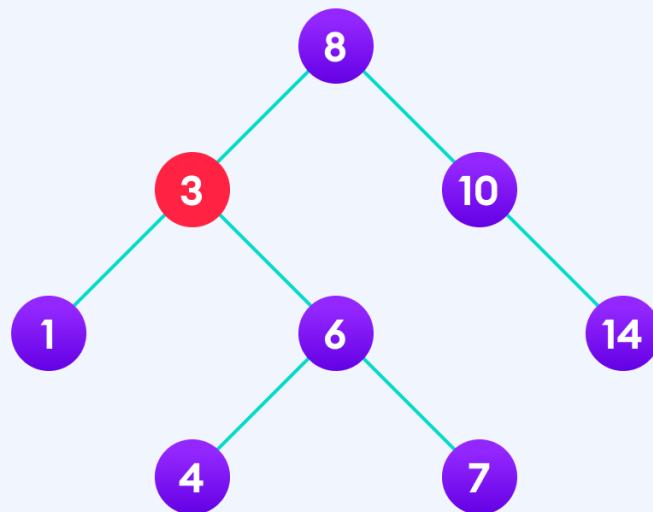
Case III

In the third case, the node to be deleted has two children. In such a case follow the steps below:

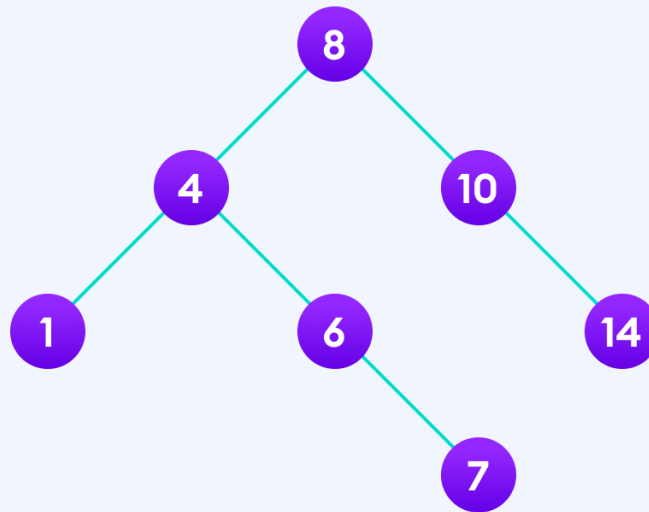
1. Get the inorder successor of that node.
2. Replace the node with the inorder successor.
3. Remove the inorder successor from its original position.



3 is to be deleted



Copy the value of the inorder successor (4) to the node



// Binary Search Tree operations in Java

```
class BinarySearchTree {
    class Node {
        int key;
        Node left, right;

        public Node(int item) {
            key = item;
            left = right = null;
        }
    }

    Node root;

    BinarySearchTree() {
        root = null;
    }

    void insert(int key) {
        root = insertKey(root, key);
    }
}
```

```

// Insert key in the tree
Node insertKey(Node root, int key) {
    // Return a new node if the tree is empty
    if (root == null) {
        root = new Node(key);
        return root;
    }

    // Traverse to the right place and insert the node
    if (key < root.key)
        root.left = insertKey(root.left, key);
    else if (key > root.key)
        root.right = insertKey(root.right, key);

    return root;
}

void inorder() {
    inorderRec(root);
}

// Inorder Traversal
void inorderRec(Node root) {
    if (root != null) {
        inorderRec(root.left);
        System.out.print(root.key + " -> ");
        inorderRec(root.right);
    }
}

void deleteKey(int key) {
    root = deleteRec(root, key);
}

Node deleteRec(Node root, int key) {
    // Return if the tree is empty
    if (root == null)
        return root;

    // Find the node to be deleted

```

```

    if (key < root.key)
        root.left = deleteRec(root.left, key);
    else if (key > root.key)
        root.right = deleteRec(root.right, key);
    else {
        // If the node is with only one child or no child
        if (root.left == null)
            return root.right;
        else if (root.right == null)
            return root.left;

        // If the node has two children
        // Place the inorder successor in position of the node to be deleted
        root.key = minValue(root.right);

        // Delete the inorder successor
        root.right = deleteRec(root.right, root.key);
    }

    return root;
}

// Find the inorder successor
int minValue(Node root) {
    int minv = root.key;
    while (root.left != null) {
        minv = root.left.key;
        root = root.left;
    }
    return minv;
}

// Driver Program to test above functions
public static void main(String[] args) {
    BinarySearchTree tree = new BinarySearchTree();

    tree.insert(8);
    tree.insert(3);
    tree.insert(1);
    tree.insert(6);

```

```
tree.insert(7);  
tree.insert(10);  
tree.insert(14);  
tree.insert(4);
```

```
System.out.print("Inorder traversal: ");  
tree.inorder();
```

```
System.out.println("\n\nAfter deleting 10");  
tree.deleteKey(10);  
System.out.print("Inorder traversal: ");  
tree.inorder();
```

```
}  
}
```