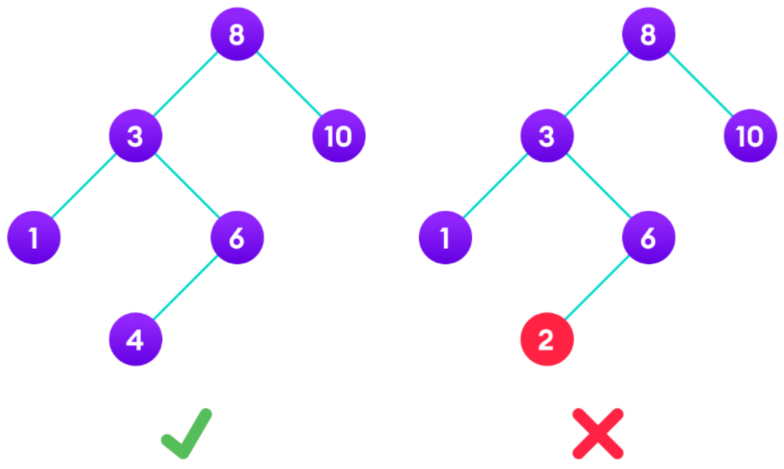
Binary Search Tree(BST)

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



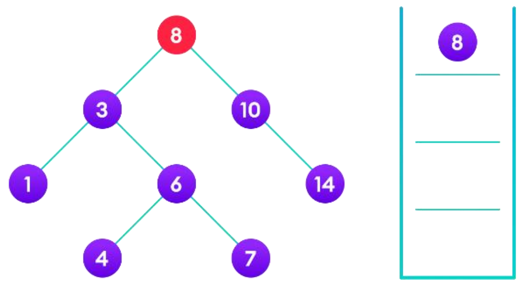
Search Operation

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.

Algorithm:

```
If root == NULL
    return NULL;
If number == root->data
    return root->data;
If number < root->data
    return search(root->left)
If number > root->data
    return search(root->right)
```



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

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.

Algorithm:

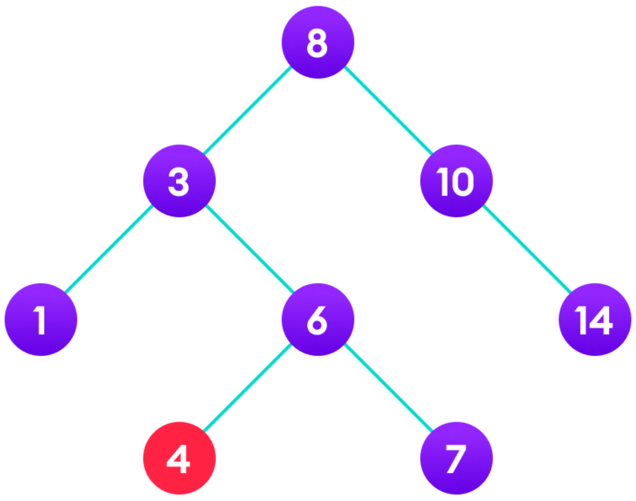
```
If node == NULL
    return createNode(data)
if (data < node->data)
    node->left  = insert(node->left, data);
else if (data > node->data)
    node->right = insert(node->right, data);
return node;
```

Deletion Operation

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

Case I

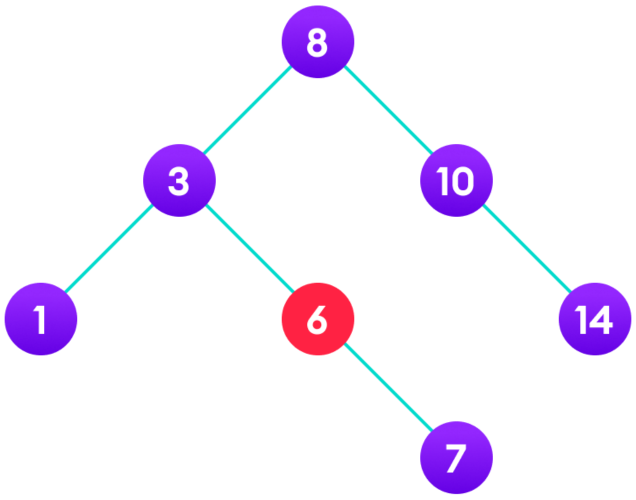
In the first case, the node to be deleted is the leaf node. In such a case, simply delete the node from the tree.



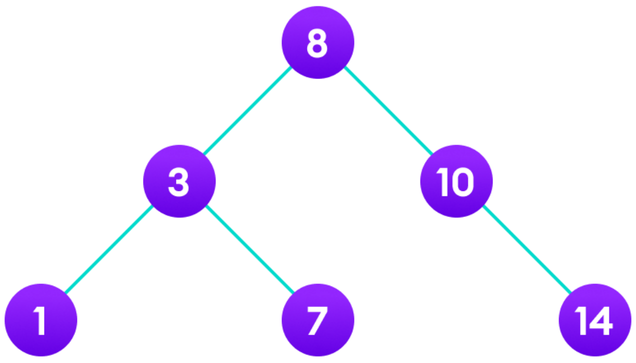
Case II

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.



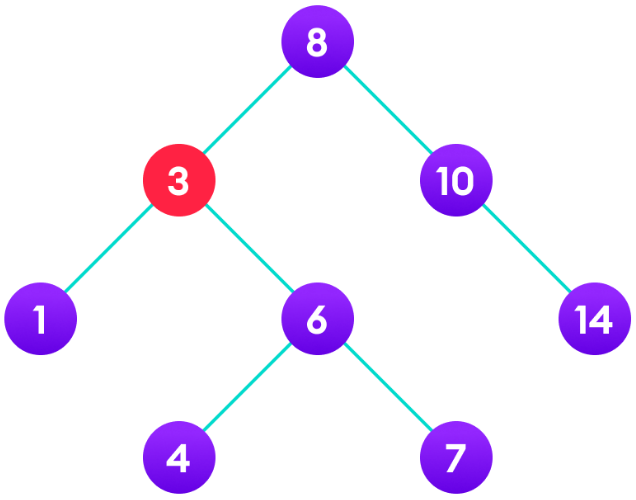
Final Tree



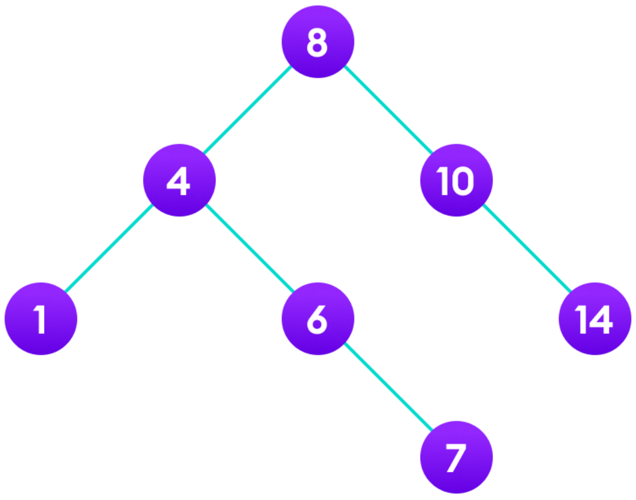
Case III

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.



Final Tree



Binary Search Tree Complexities

Time Complexity

Operation	Best Case Complexity	Average Case Complexity	Worst Case Complexity
Search	O(log n)	O(log n)	O(n)
Insertion	O(log n)	O(log n)	O(n)
Deletion	O(log n)	O(log n)	O(n)

Space Complexity

The space complexity for all the operations is `O(n)`.

Binary Search Tree Applications

1. In multilevel indexing in the database
2. For dynamic sorting
3. For managing virtual memory areas in Unix kernel

```
#include<iostream>
using namespace std;

class BSTNode
{
public:
    int Key;
    BSTNode * Left;
    BSTNode * Right;
    BSTNode * Parent;
};

// Function to create a new Node in heap
BSTNode * BST::Insert(BSTNode * node, int key)
{
    // If BST doesn't exist
    // create a new node as root
    // or it's reached when
    // there's no any child node
    // so we can insert a new node here
    if(node == NULL)
    {
        node = new BSTNode;
        node->Key = key;
        node->Left = NULL;
        node->Right = NULL;
        node->Parent = NULL;
    }
    // If the given key is greater than
    // node's key then go to right subtree
    else if(node->key < key){
        node->Right = Insert(node->Right, key);
        node->Right->Parent = node;
    }
    // If the given key is smaller than
    // node's key then go to left subtree
    else
    {
        node->Left = Insert(node->Left, key);
        node->Left->Parent = node;
    }
int main() {
        BSTNode* root = NULL;    // Creating an empty tree
        /*Code to test the logic*/
    void BST::Insert(int key){
        root = Insert(root,15);
        root = Insert(root,10);
        root = Insert(root,20);
        root = Insert(root,25);
        root = Insert(root,8);
        root = Insert(root,12);
    }
```

Program Implementation

```
// Binary Search Tree operations in C++
#include <iostream>
using namespace std;
struct node {
  int key;
  struct node *left, *right;
};
// Create a 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;
}
// Inorder Traversal
void inorder(struct node *root) {
  if (root != NULL) {
    // Traverse left
    inorder(root->left);
    // Traverse root
    cout << root->key << " -> ";
    // Traverse right
    inorder(root->right);
  }}
// Insert a node
struct node *insert(struct node *node, int key) {
  // Return a new node if the tree is empty
  if (node == NULL) return newNode(key);
  // Traverse to the right place and insert the node
  if (key < node->key)
    node->left = insert(node->left, key);
  else
    node->right = insert(node->right, key);
  return node;
}

// Find the inorder successor
struct node *minValueNode(struct node *node) {
  struct node *current = node;
  // Find the leftmost leaf
  while (current && current->left != NULL)
    current = current->left;
  return current;
}

// Deleting a node
struct node *deleteNode(struct 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 = deleteNode(root->left, key);
else if (key > root->key)
  root->right = deleteNode(root->right, key);
else {
  // If the node is with only one child or no child
  if (root->left == NULL) {
    struct node *temp = root->right;
    free(root);
    return temp;
  } else if (root->right == NULL) {
    struct node *temp = root->left;
    free(root);
    return temp;
  }

  // If the node has two children
  struct node *temp = minValueNode(root->right);

  // Place the inorder successor in position of the node to be deleted
  root->key = temp->key;
```

```
    // Delete the inorder successor
    root->right = deleteNode(root->right, temp->key);
  }
  return root;
}

// Driver code
int main() {
  struct node *root = NULL;
  root = insert(root, 8);
  root = insert(root, 3);
  root = insert(root, 1);
  root = insert(root, 6);
  root = insert(root, 7);
  root = insert(root, 10);
  root = insert(root, 14);
  root = insert(root, 4);

  cout << "Inorder traversal: ";
  inorder(root);

  cout << "\nAfter deleting 10\n";
  root = deleteNode(root, 10);
  cout << "Inorder traversal: ";
  inorder(root);
```

```
}
```