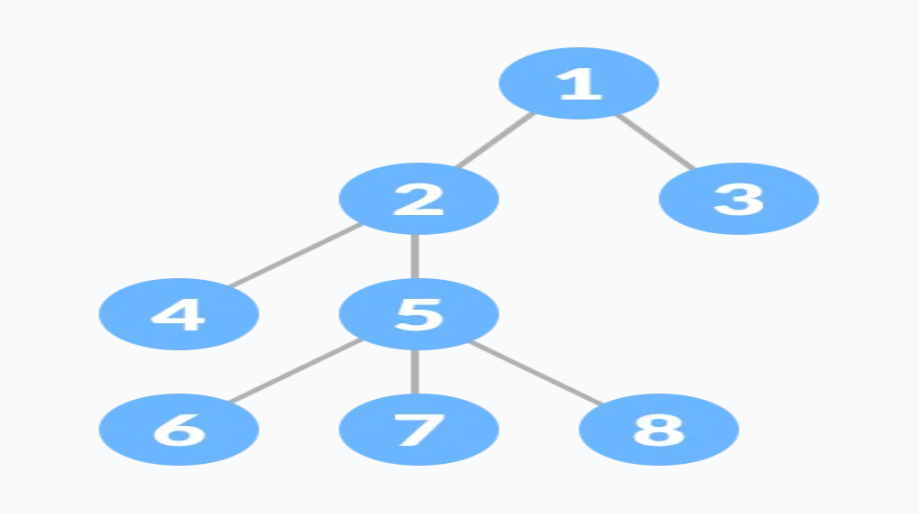
TREE

A tree is a nonlinear hierarchical data structure that consists of nodes connected by edges.



Why Tree Data Structure?

Other data structures such as arrays, linked list, stack, and queue are linear data structures that store data sequentially. In order to perform any operation in a linear data structure, the time complexity increases with the increase in the data size. But, it is not acceptable in today's computational world.

Different tree data structures allow quicker and easier access to the data as it is a non-linear data structure.

Tree Terminologies

Node

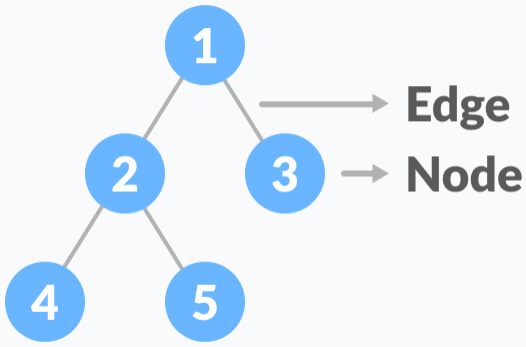
A node is an entity that contains a key or value and pointers to its child nodes.

The last nodes of each path are called leaf nodes or external nodes that do not contain a link/pointer to child nodes.

The node having at least a child node is called an internal node.

Edge

It is the link between any two nodes.



Nodes and edges of a tree

Root

It is the topmost node of a tree.

Height of a Node

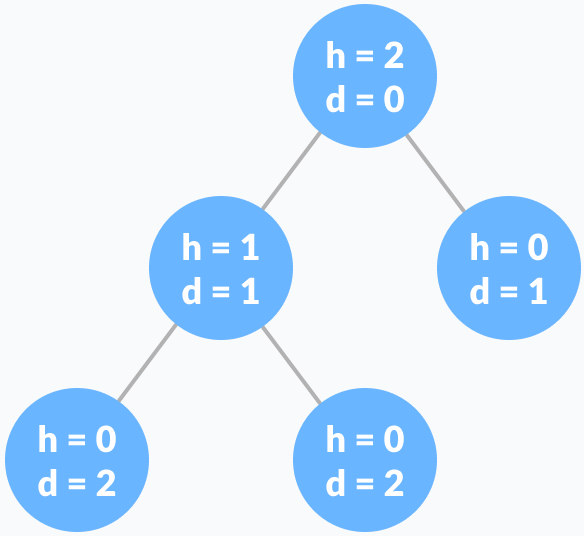
The height of a node is the number of edges from the node to the deepest leaf (ie. the longest path from the node to a leaf node).

Depth of a Node

The depth of a node is the number of edges from the root to the node.

Height of a Tree

The height of a Tree is the height of the root node or the depth of the deepest node.



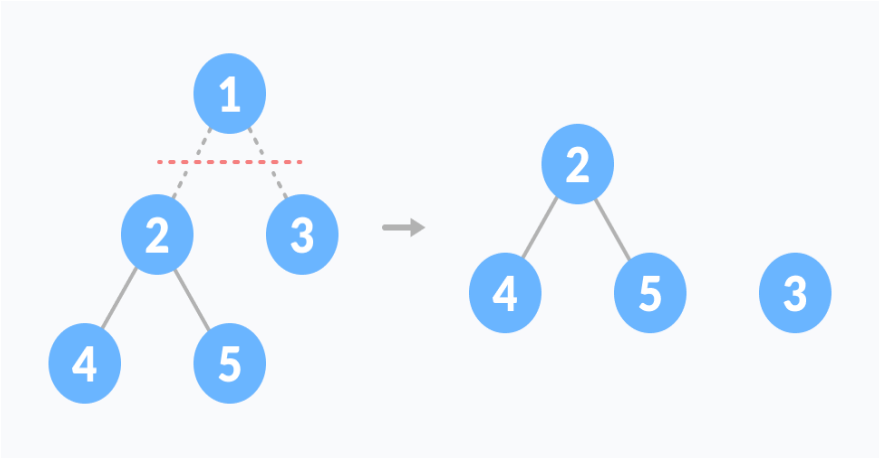
Height and depth of each node in a tree

Degree of a Node

The degree of a node is the total number of branches of that node.

Forest

A collection of disjoint trees is called a forest.



creating forest from a tree

You can create a forest by cutting the root of a tree.

Types of Tree

1. Binary Tree
2. Binary Search Tree
3. AVL Tree
4. B-Tree

Tree Traversal

In order to perform any operation on a tree, you need to reach to the specific node. The tree traversal algorithm helps in visiting a required node in the tree.

To learn more, please visit tree traversal.

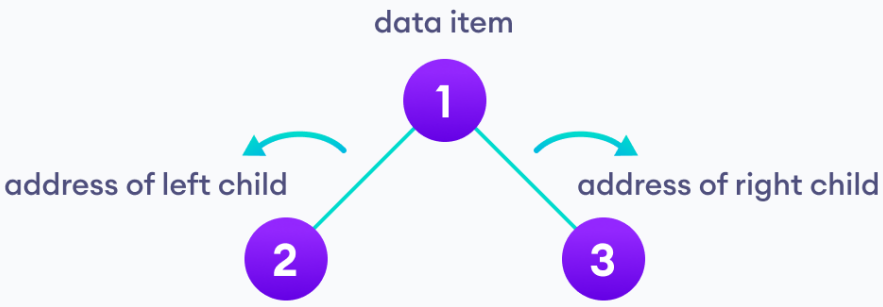
Tree Applications

- Binary Search Trees(BSTs) are used to quickly check whether an element is present in a set or not.
- Heap is a kind of tree that is used for heap sort.
- A modified version of a tree called Tries is used in modern routers to store routing information.
- Most popular databases use B-Trees and T-Trees, which are variants of the tree structure we learned above to store their data
- Compilers use a syntax tree to validate the syntax of every program you write.

1. BINARY TREE

A binary tree is a tree data structure in which each parent node can have at most two children. Each node of a binary tree consists of three items:

- data item
- address of left child
- address of right child

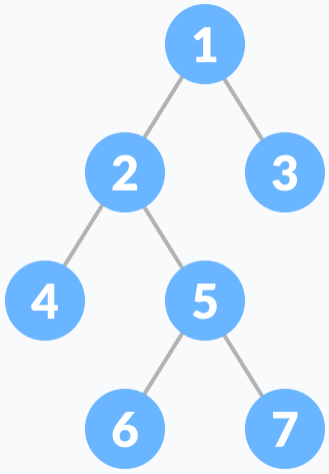


Binary Tree

Types of Binary Tree

1. Full Binary Tree

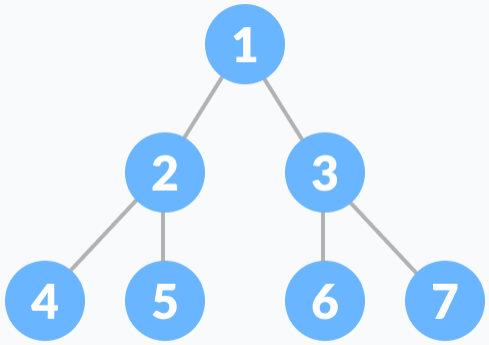
A full Binary tree is a special type of binary tree in which every parent node/internal node has either two or no children.



Full Binary Tree.

2. Perfect Binary Tree

A perfect binary tree is a type of binary tree in which every internal node has exactly two child nodes and all the leaf nodes are at the same level.

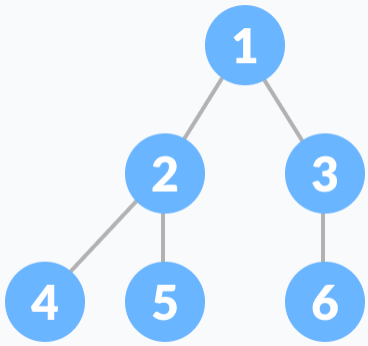


Perfect Binary Tree

3. Complete Binary Tree

A complete binary tree is just like a full binary tree, but with two major differences

1. Every level must be completely filled
2. All the leaf elements must lean towards the left.
3. The last leaf element might not have a right sibling i.e. a complete binary tree doesn't have to be a full binary tree.

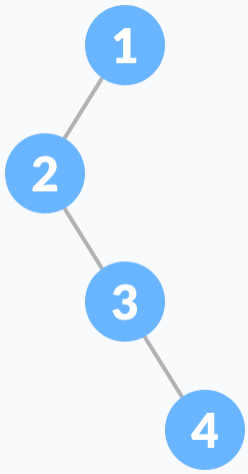


Complete Binary Tree

To learn more, please visit complete binary tree.

4. Degenerate or Pathological Tree

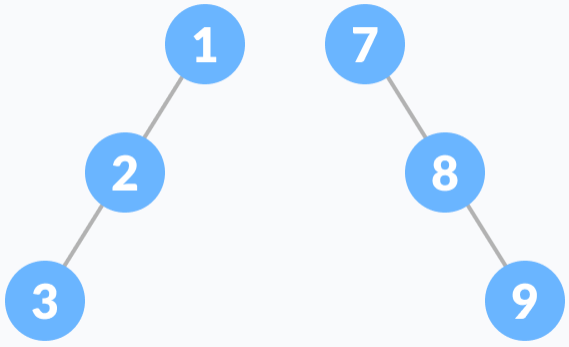
A degenerate or pathological tree is the tree having a single child either left or right.



Degenerate Binary Tree

5. Skewed Binary Tree

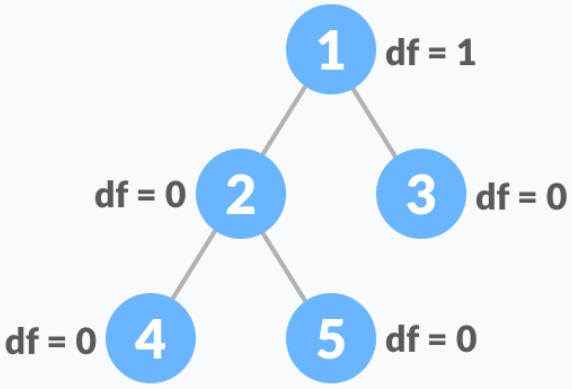
A skewed binary tree is a pathological/degenerate tree in which the tree is either dominated by the left nodes or the right nodes. Thus, there are two types of skewed binary tree: left-skewed binary tree and right-skewed binary tree.



Skewed Binary Tree

6. Balanced Binary Tree

It is a type of binary tree in which the difference between the height of the left and the right subtree for each node is either 0 or 1.



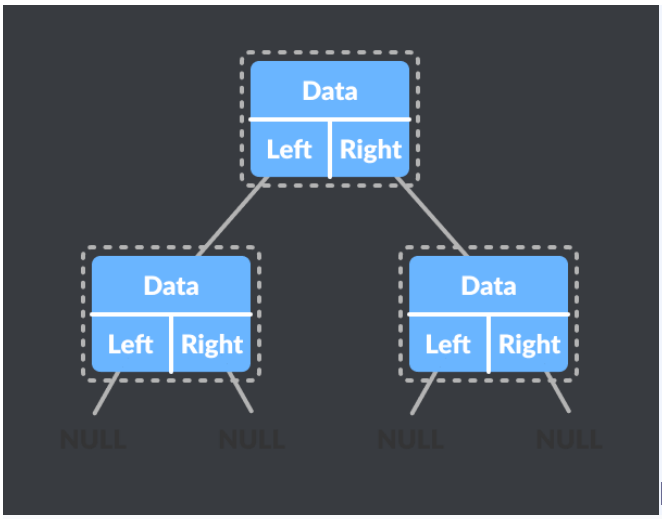
Balanced Binary Tree

To learn more, please visit balanced binary tree.

Binary Tree Representation

A node of a binary tree is represented by a structure containing a data part and two pointers to other structures of the same type.

```
struct node
{
 int data;
 struct node *left;
 struct node *right;
};
```



Binary Tree Representation

```
#include <bits/stdc++.h>
using namespace std;

struct node {
  int data;
  struct node *left;
  struct node *right;
};

struct node *newNode(int data) {
  struct node *node = (struct node
*)malloc(sizeof(struct node));
  node->data = data;
  node->left = NULL;
  node->right = NULL;
  return (node);
}

void traversePreOrder(struct node
*temp) {
  if (temp != NULL) {
    cout << " " << temp->data;
    traversePreOrder(temp->left);
    traversePreOrder(temp->right);
  }
}
```

```
void traverseInOrder(struct node
*temp) {
  if (temp != NULL) {
    traverseInOrder(temp->left);
    cout << " " << temp->data;
    traverseInOrder(temp->right);
  }

void traversePostOrder(struct node
*temp) {
  if (temp != NULL) {
    traversePostOrder(temp->left);
    traversePostOrder(temp->right);
    cout << " " << temp->data;
  }
}

int main() {
  struct node *root = newNode(1);
  root->left = newNode(2);
  root->right = newNode(3);
  root->left->left = newNode(4);
  cout << "preorder traversal: ";
  traversePreOrder(root);
  cout << "\nInorder traversal: ";
  traverseInOrder(root);
 cout << "\nPostorder traversal: ";
  traversePostOrder(root);
}
```

Binary Tree Applications

- For easy and quick access to data
- In router algorithms
- To implement heap data structure
- Syntax tree

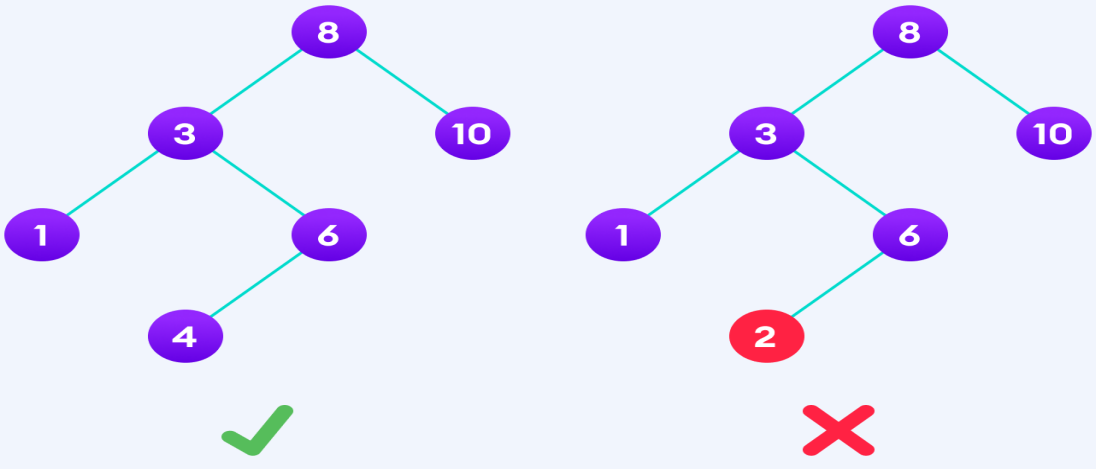
2. BINARY SEARCH TREE(BST)

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

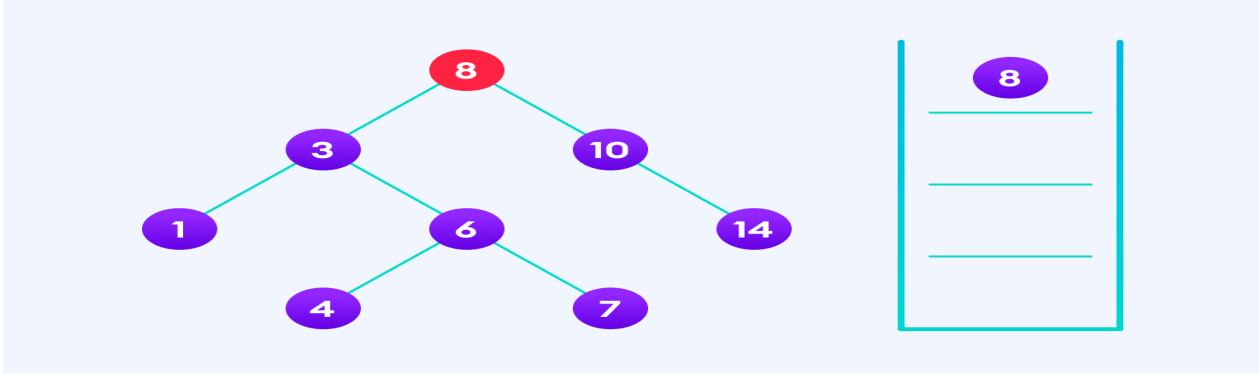
The algorithm depends on the property of BST that if each left subtree has values below root and each right subtree has values above the root.

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.

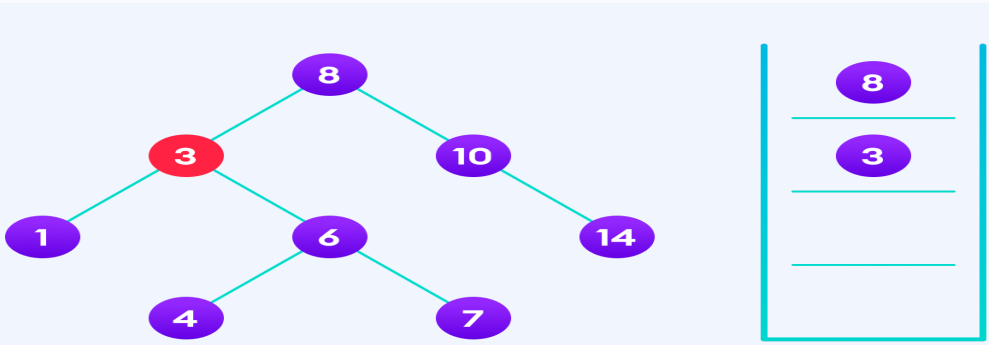
Algorithm:

```
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)
```

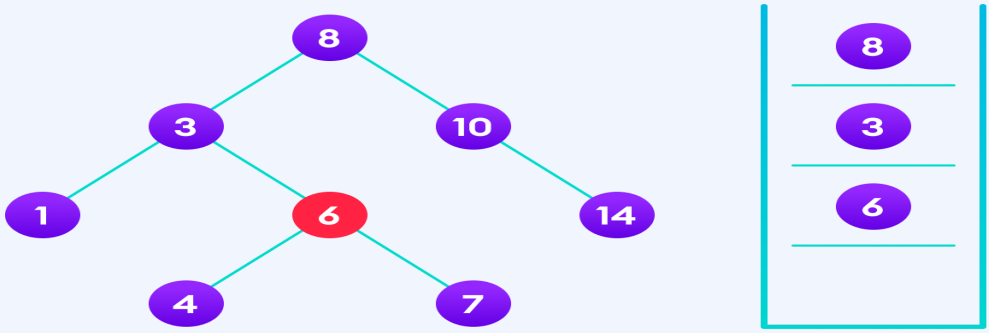
.



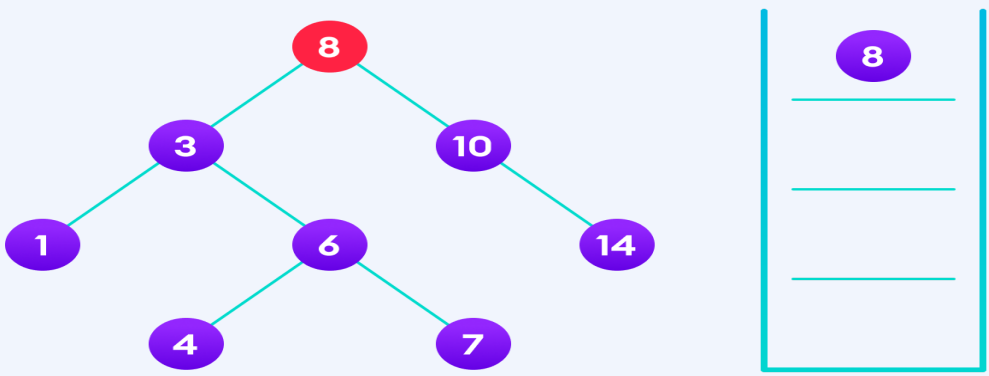
4 is not found so, traverse through the left subtree of 8



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



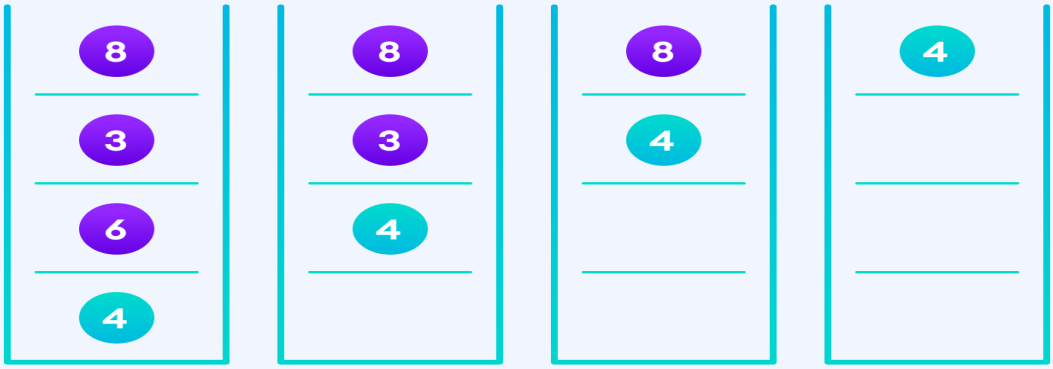
4 is not found so, traverse through the left subtree of 6



4 is found

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

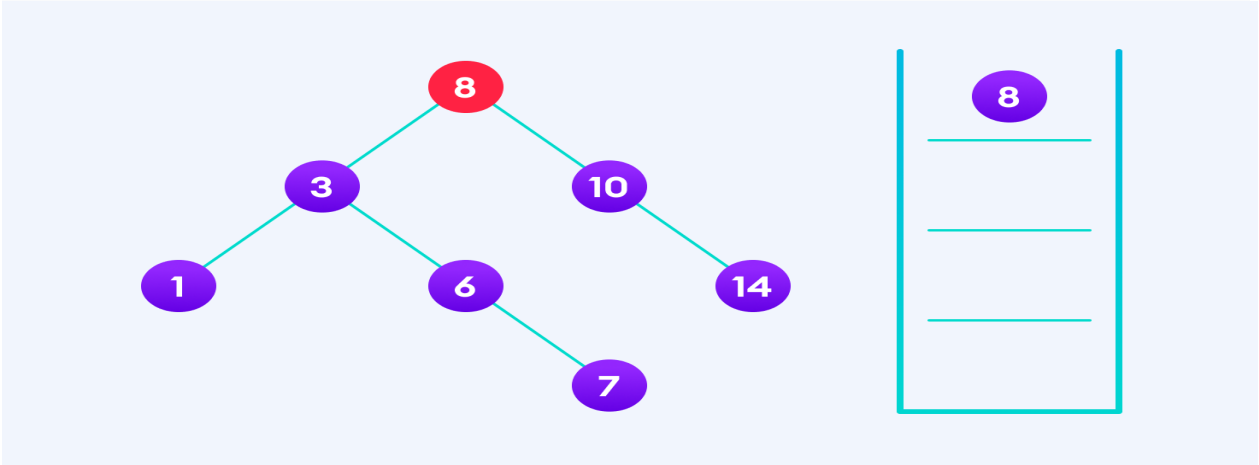
Inserting a value in the correct position is similar to searching because we try to maintain the rule that the left subtree is lesser than root and the right subtree is larger than root.

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.

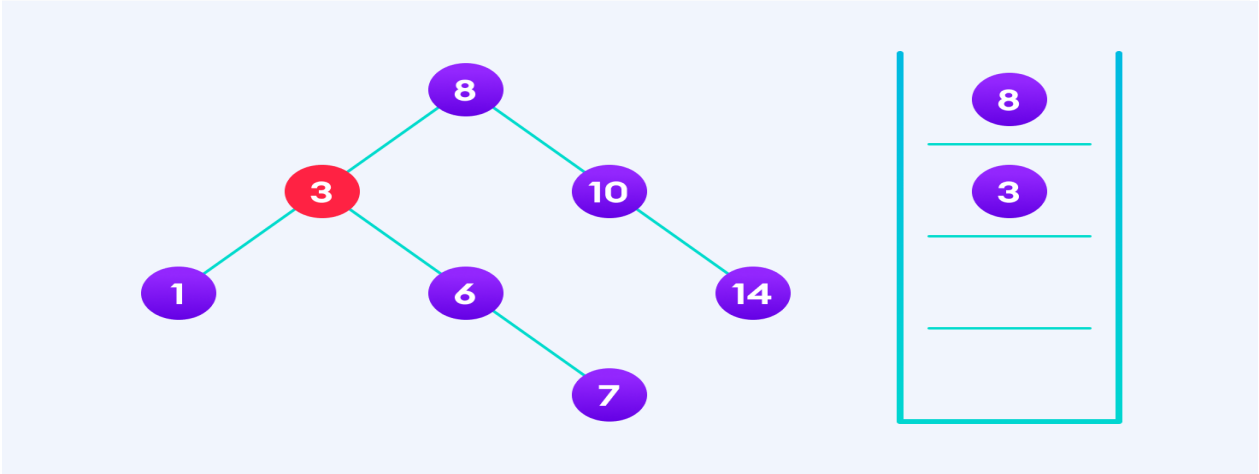
Algorithm:

```
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;
```

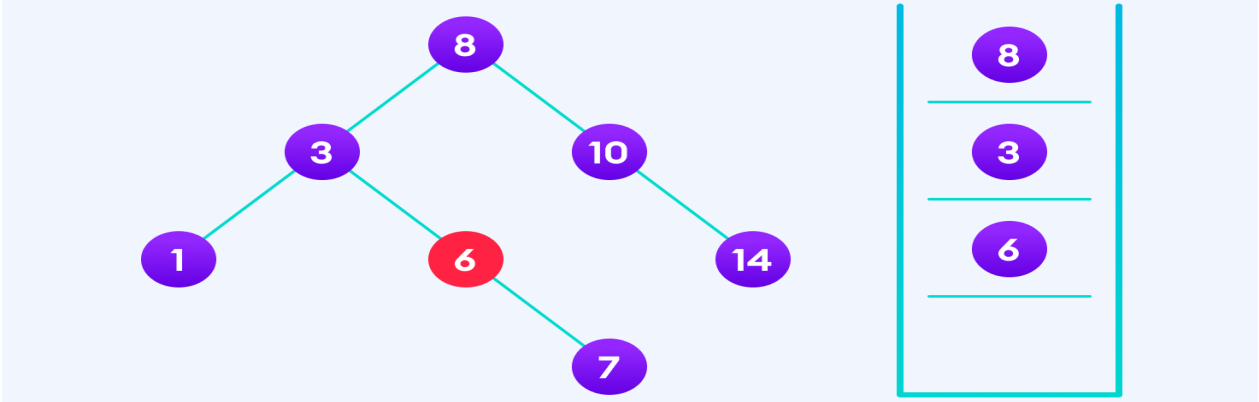
The algorithm isn't as simple as it looks. Let's try to visualize how we add a number to an existing BST.



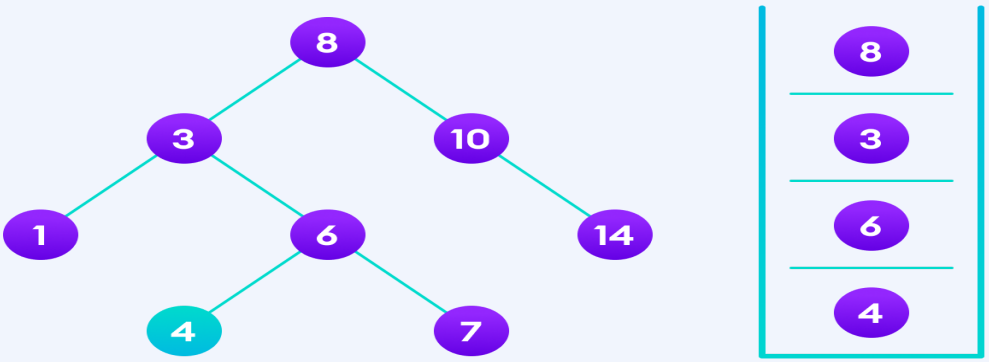
4<8 so, transverse through the left child of 8



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.

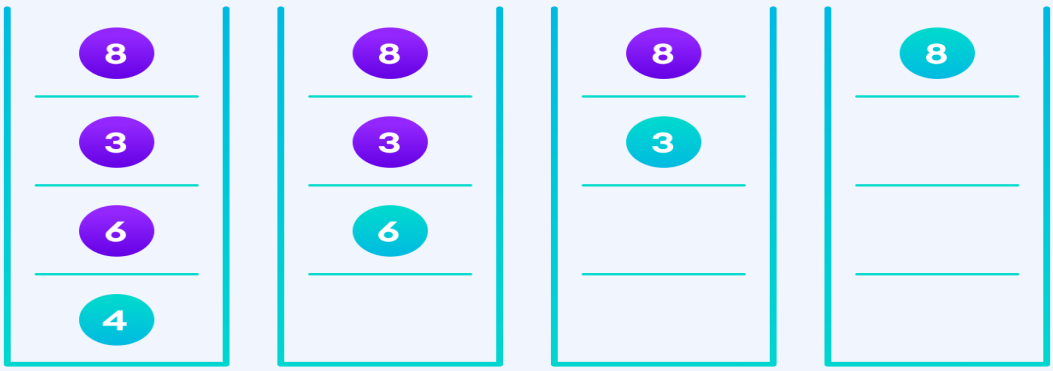


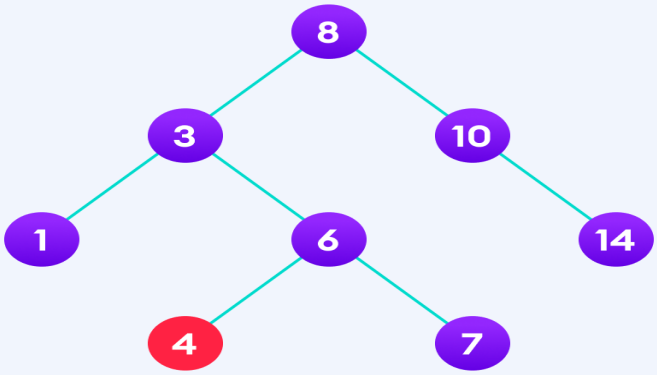
Image showing the importance of returning the root element at the end so that the elements don't lose their position during the upward recursion step.

Deletion Operation

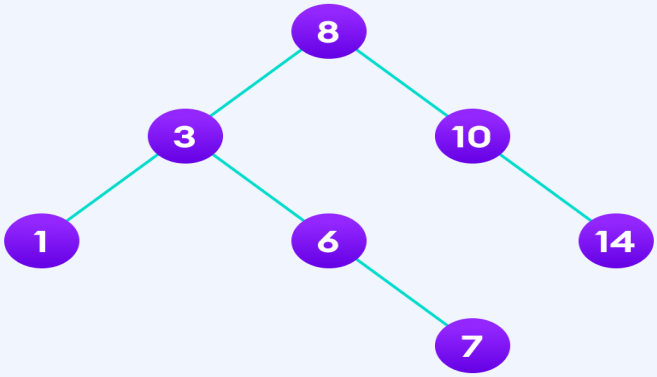
There are three cases for deleting a node from a binary search tree.

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

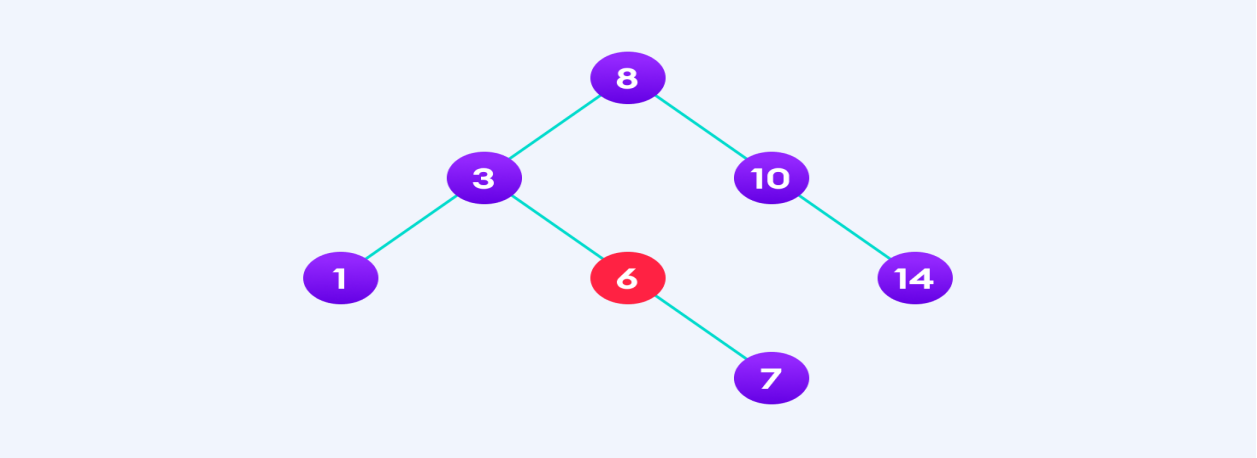


Delete the node

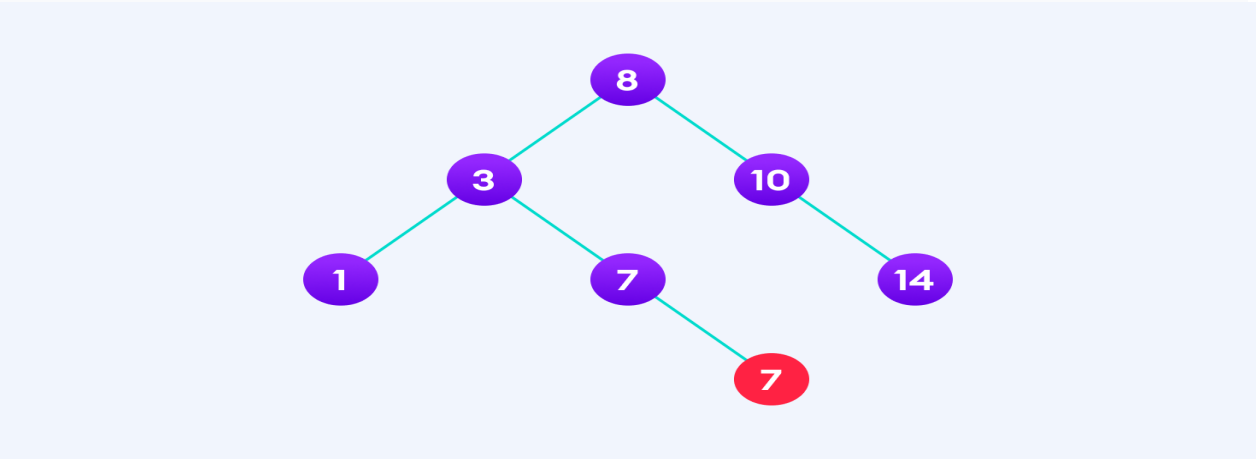
Case II

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

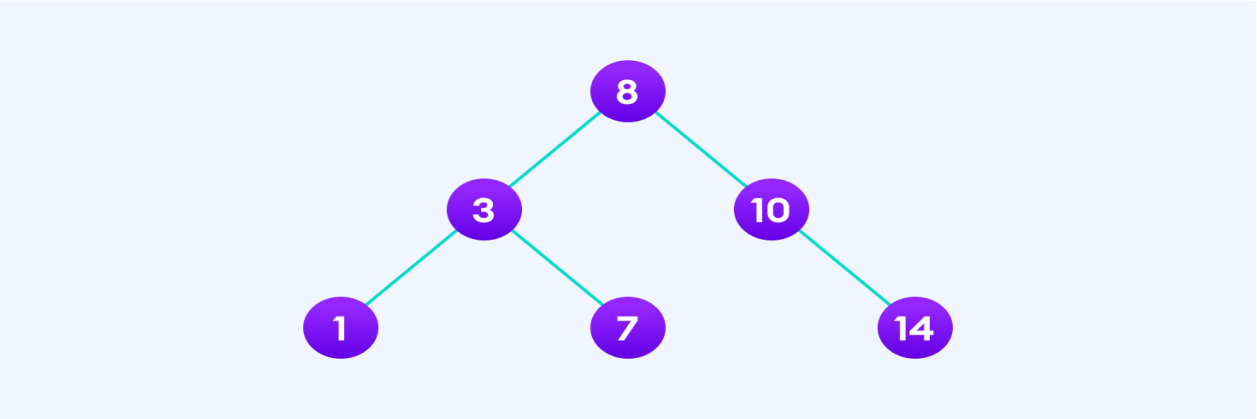
1. Replace that node with its child node.
2. Remove the child node from its original position.



6 is to be deleted



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



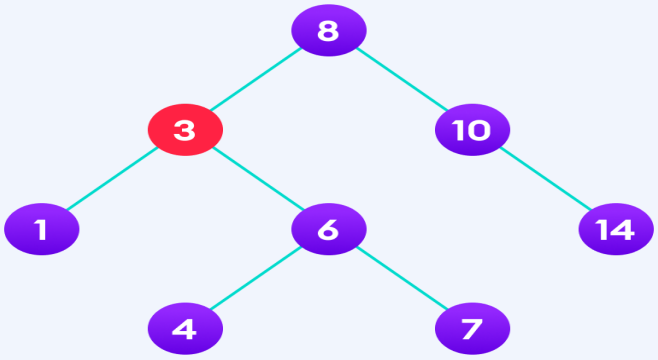
Final tree

Case III

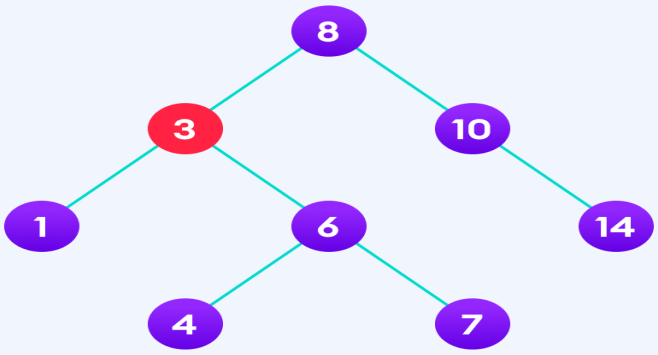
In the third case, the node to be deleted has two children. In such a case follow the steps below:

1. Get the inorder successor of that node.

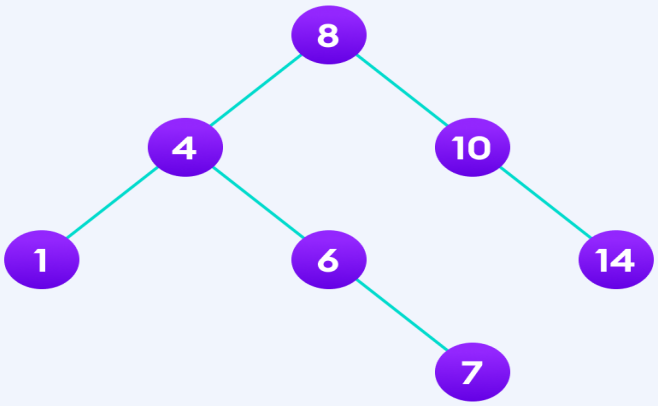
2. Replace the node with the inorder successor.
3. Remove the inorder successor from its original position.



3 is to be deleted



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



Delete the inorder successor

```
#include <bits/stdc++.h>
using namespace std;

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;
}

void inorder(struct node *root) {
  if (root != NULL) {
   inorder(root->left);
   cout << root->key << " -> ";
    inorder(root->right);
  }
}

struct node *minValueNode(struct
node *node) {
  struct node *current = node;
  while (current && current->left
!= NULL)
current = current->left;
return current;
}
```

```
struct node *deleteNode(struct node
*root, int key) {
    if (root == NULL) return root;
  if (key < root->key)
   root->left =
deleteNode(root->left, key);
  else if (key > root->key)
    root->right =
deleteNode(root->right, key);
  else {
      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;
    }
struct node *temp =
minValueNode(root->right);
    root->key = temp->key;
    root->right =
deleteNode(root->right, temp->key);
  }return root;
}

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);
}
```

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)

Here, `n` is the number of nodes in the tree.

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