What is a Binary search tree?

A binary search tree is a tree data structure that uses node representation wherein each node has a key associated to it. It maintains 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.

Properties of a BST:

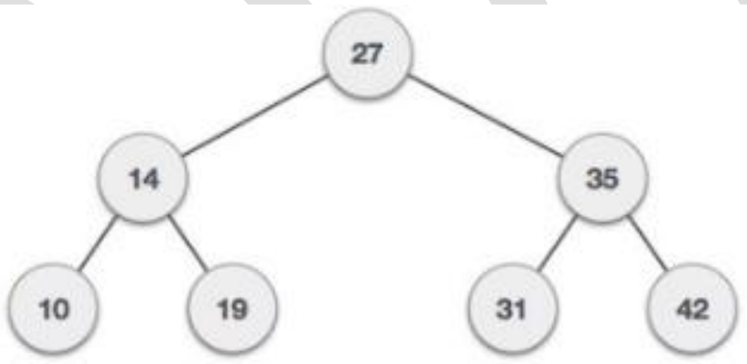
1. All values stored in nodes of left subtree are lesser than the root node
2. All values stored in nodes of right subtree are greater than the root node
3. Both sub-trees of each node are also BSTs i.e. they have the above two properties

left_subtree (keys) < node (key) ≤ right_subtree (keys)

Representation:

BST is a collection of nodes arranged in a way where they maintain BST properties. Each node has a key and an associated value.

Pictorial Representation:



Operations on Binary Search Tree:

There are many operations which can be performed on a binary search tree.

SN	Operation	Description
1	Searching in BST	Finding the location of some specific element in a binary search tree.
2	Insertion in BST	Adding a new element to the binary search tree at the appropriate location so that the property of BST do not violate.
3	Deletion in BST	Deleting some specific node from a binary search tree. However, there can be various cases in deletion depending upon the number of children, the node have.

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.

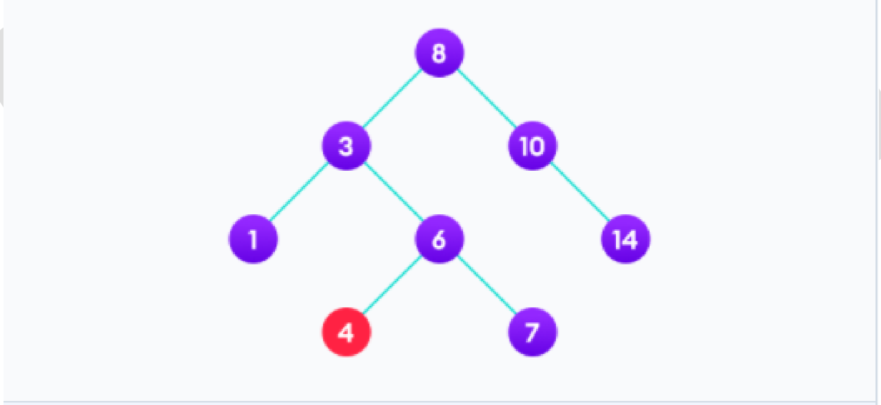
```
If node == NULL
   return createNode(data)
if (data < node->data)
   node->left  = insert(node->left, data);
else if (data > node->data)
   node->right = insert(node->right, data);
return node;
```

Delete Operation:

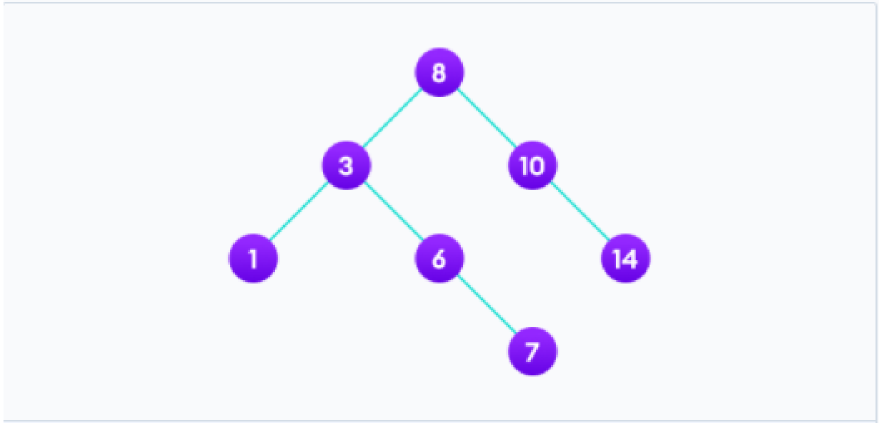
There are three cases for deleting a node from a binary search tree.

Case 1:

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

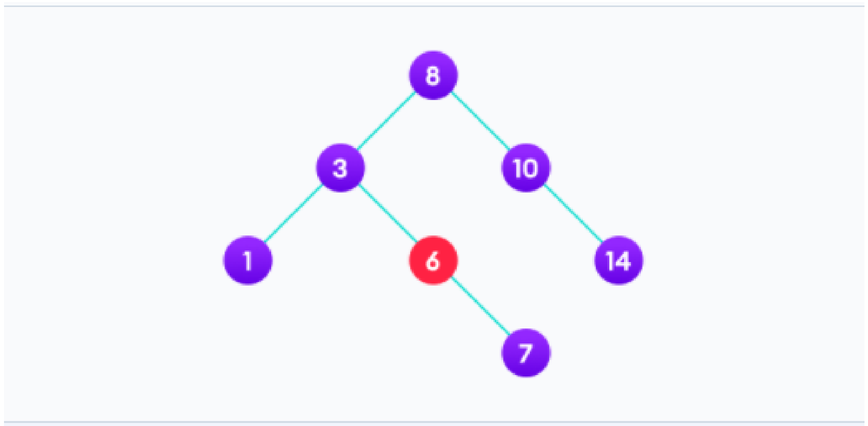


Delete the node

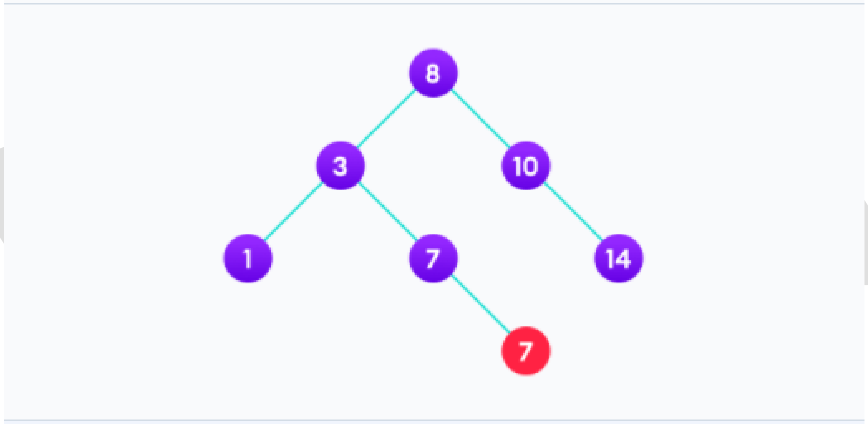
Case 2:

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

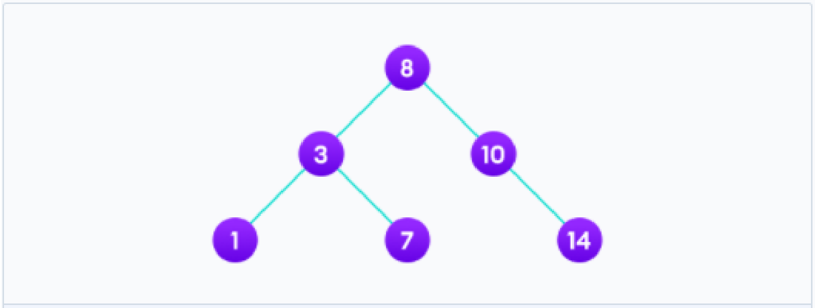
1. Replace that node with its child node.
2. Remove the child node from its original position.



6 is to be deleted



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

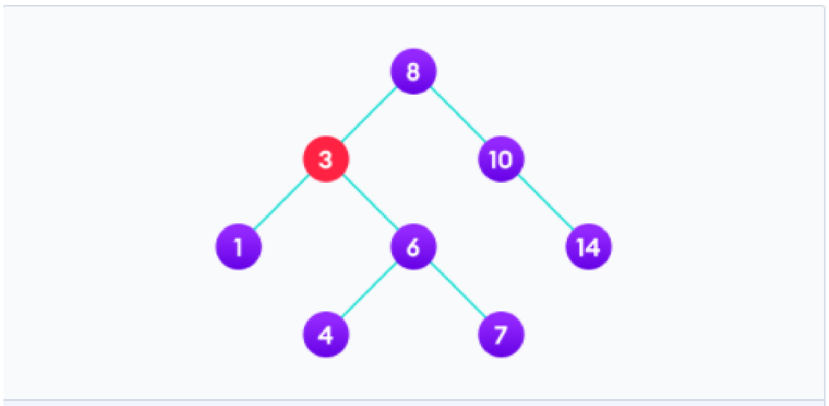


Final tree

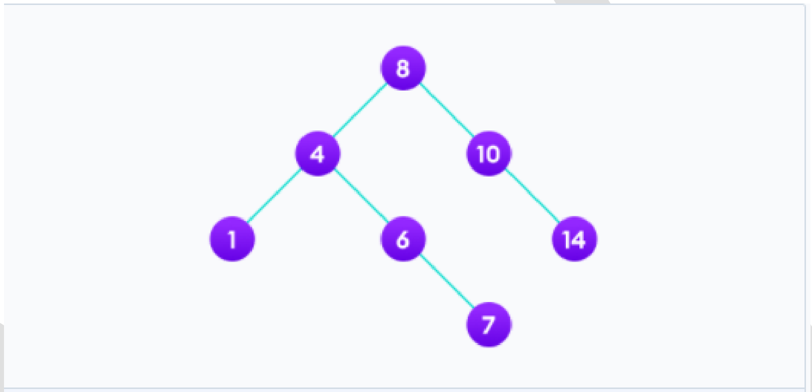
Case III

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.



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



Algorithm:

```
Node deleteNode(Node root, int valueToDelete) {
 if root = null
  return node
 if root.value < valueToDelete
  deleteNode(root.right, valueToDelete)
 if root.value > valueToDelete
  deleteNode(root.left, valueToDelete)
 else
  if (isLeafNode(root))
   return null
  if (root.right == null)
   return root.left
  if (root.left == null)
   return root.right
  else
   minValue = findMinInRightSubtree(root)
   root.value = minValue
   removeDuplicateNode(root)
   return root
```

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)
```

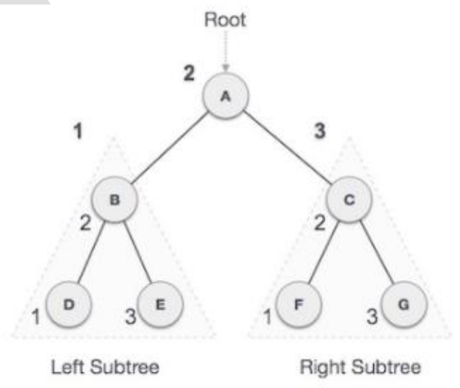
BST Traversal:

There are three ways which we use to traverse a tree:

1. Inorder Traversal:
In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.

Pseudocode:

```
inorder(root->left)
display(root->data)
inorder(root->right)
```

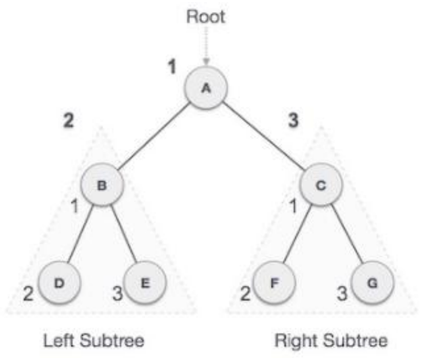


We start from **A**, and following in-order traversal, we move to its left subtree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be –

D → B → E → A → F → C → G

Pre-order Traversal
In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.

Pseudocode:
```
display(root->data)
preorder(root->left)
preorder(root->right)
```



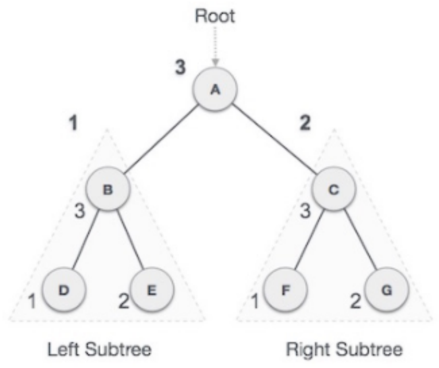
We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be –

A → B → D → E → C → F → G

Postorder Traversal:

Pseudocode:
```
postorder(root->left)
postorder(root->right)
display(root->data)
```

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



We start from **A**, and following Post-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be –

D → E → B → F → G → C → A

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

Define AVL tree and ·discuss various rotations on AVL tree and height of AVL tree.

- AVL tree is a self-balancing binary search tree invented by G.M. Adelson-Velsky and E.M. Landis in 1962.
- The tree is named AVL in honor of its inventors.
- In an AVL tree, the heights of the two sub-trees of a node may differ by at most one.
- Due to this property, the AVL tree is also known as a height-balanced tree.
- The key advantage of using an AVL tree is that it takes O(log n) time to perform search, insert, and delete operations in an average case as well as the worst case because the height of the tree is limited to O(log n).
- The structure of an AVL tree is the same as that of a binary search tree but with a little difference. In its structure, it stores an additional variable called the BalanceFactor.
- Thus, every node has a balance factor associated with it.
- The balance factor of a node is calculated by subtracting the height of its right sub-tree from the height of its left sub-tree.
- A binary search tree in which every node has a balance factor of –1, 0, or 1 is said to be height balanced.
- A node with any other balance factor is considered to be unbalanced and requires rebalancing of the tree.
- Balance factor = Height (left sub-tree) – Height (right sub-tree)
- If the balance factor of a node is 1, then it means that the left sub-tree of the tree is one level higher than that of the right sub-tree. Such a tree is therefore called as a left-heavy tree.
- If the balance factor of a node is 0, then it means that the height of the left sub-tree (longest path in the left sub-tree) is equal to the height of the right sub-tree.
- If the balance factor of a node is –1, then it means that the left sub-tree of the tree is one level lower than that of the right sub-tree. Such a tree is therefore called as a right-heavy tree.

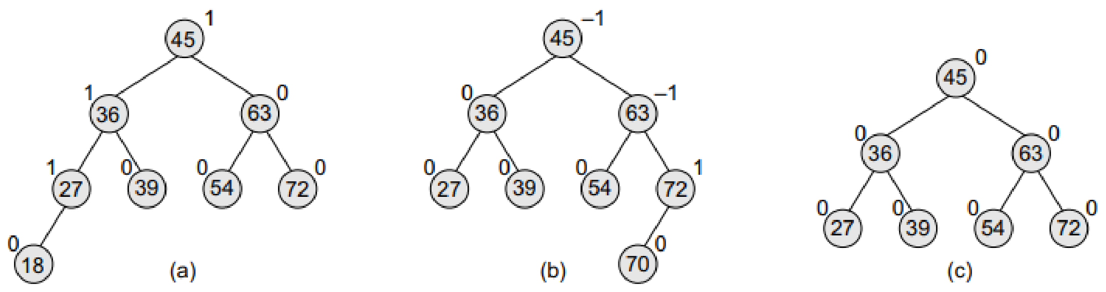


Figure 10.35 (a) Left-heavy AVL tree, (b) right-heavy tree, (c) balanced tree

Operations on AVL Trees:

Searching: Searching in an AVL tree is performed exactly the same way as it is performed in a binary search tree. Due to the height-balancing of the tree, the search operation takes O(log n) time to complete.

Inserting a New Node in an AVL Tree:
Insertion is similar to that of a binary search tree. In AVL trees, it followed by an additional step of **rotation.** Rotation is done to restore the balance of the tree. However, if insertion of the new node does not disturb the balance factor, that is, **if the balance factor of every node is still –1, 0, or 1**, then **rotations are not required.**

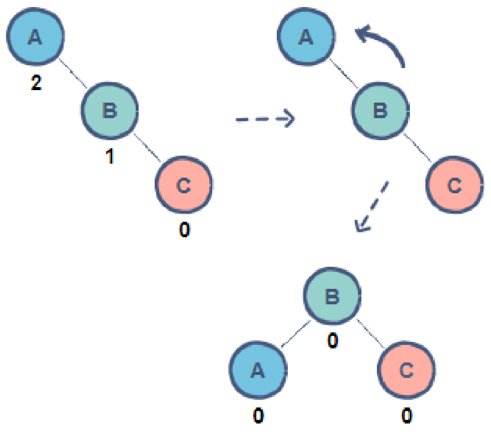
Three changes can happen:

1. Initially, the node was either left- or right-heavy and after insertion, it becomes balanced.
2. Initially, the node was balanced and after insertion, it becomes either left- or right-heavy.
3. Initially, the node was heavy (either left or right) and the new node has been inserted in the heavy sub-tree, thereby creating an unbalanced sub-tree. Such a node is said to be a **critical node.**

There are four types of rebalancing rotations:

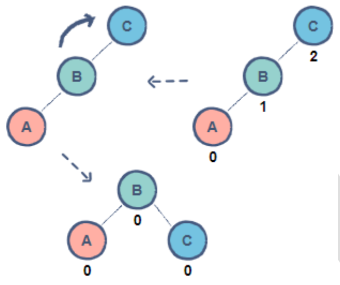
Left Rotation:

A single rotation applied when a node is inserted in the right subtree of a right subtree. In the given example, node A has a balance factor of 2 after the insertion of node C. By rotating the tree left, node B becomes the root resulting in a balanced tree.

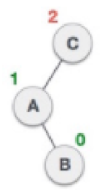
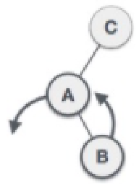
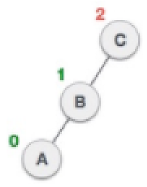
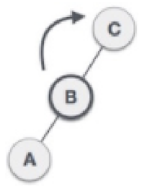
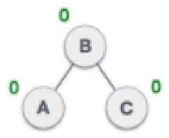


Right Rotation:

A single rotation applied when a node is inserted in the left subtree of a left subtree. In the given example, node C now has a balance factor of 2 after the insertion of node A. By rotating the tree right, node B becomes the root resulting in a balanced tree.

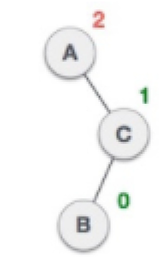
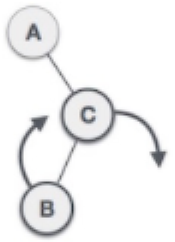
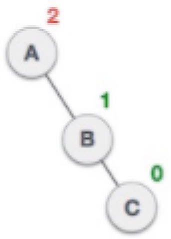
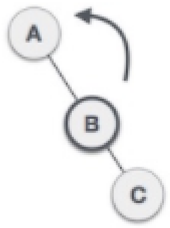
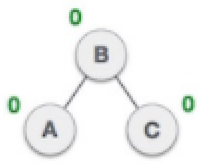


Left-Right Rotation
A double rotation in which a *left rotation* is followed by a *right rotation*.

State	Action
	A node has been inserted into the right subtree of the left subtree. This makes C an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.
	We first perform the left rotation on the left subtree of C. This makes A, the left subtree of B.
	Node C is still unbalanced, however now, it is because of the left-subtree of the left-subtree.
	We shall now right-rotate the tree, making B the new root node of this subtree. C now becomes the right subtree of its own left subtree.
	The tree is now balanced.

Right Left Rotation:

It is a combination of right rotation followed by left rotation.

State	Action
	A node has been inserted into the left subtree of the right subtree. This makes **A**, an unbalanced node with balance factor 2.
	First, we perform the right rotation along **C** node, making **C** the right subtree of its own left subtree **B**. Now, **B** becomes the right subtree of **A**.
	Node **A** is still unbalanced because of the right subtree of its right subtree and requires a left rotation.
	A left rotation is performed by making **B** the new root node of the subtree. **A** becomes the left subtree of its right subtree **B**.
	The tree is now balanced.

Deletion: