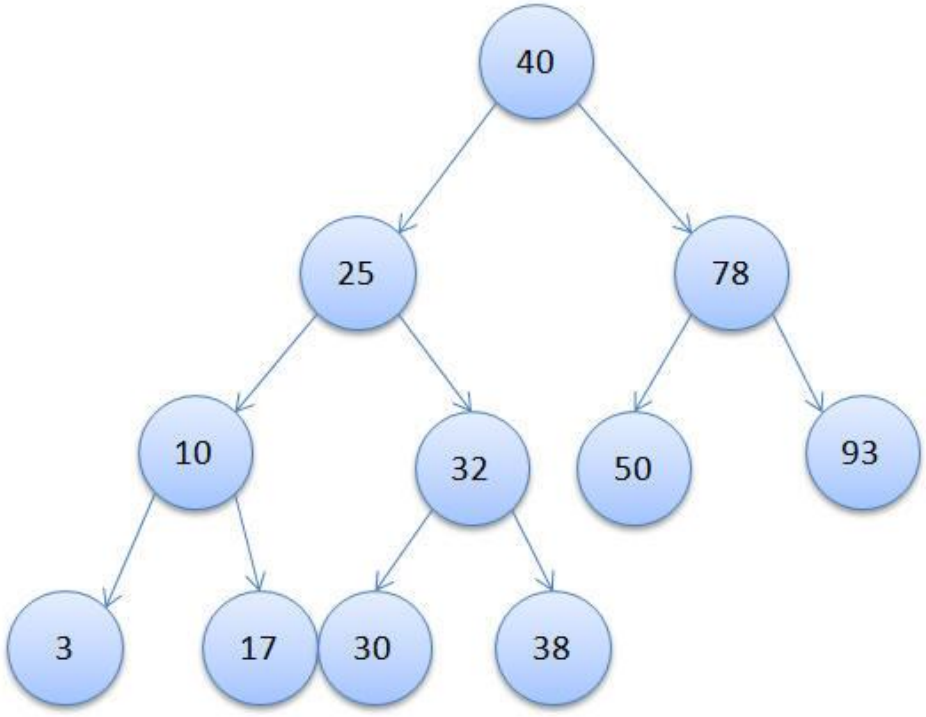
Revision: Binary Search Tree and Tree Traversal – Inorder, Preorder, Postorder

What is a Binary Search Tree (BST)?

Binary Search Tree (BST) is a binary tree data structure with a special feature where in the value store at each node is greater than or equal to the value stored at its left sub child and lesser than the value stored at its right sub child. Let's look at an example of a BST:



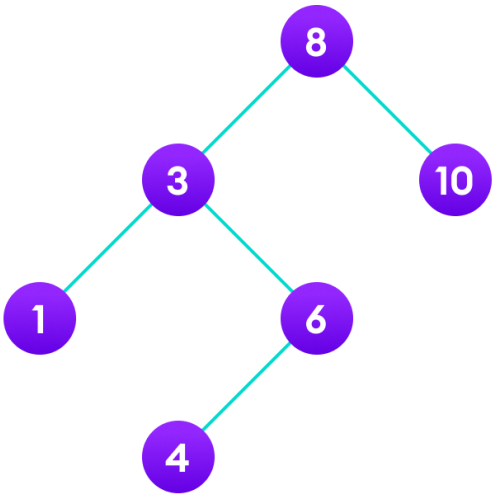
In the above example you can see that at each node the value in the left child is lesser than or equal to the value in the node and the value in the right child is greater than the value in the node.

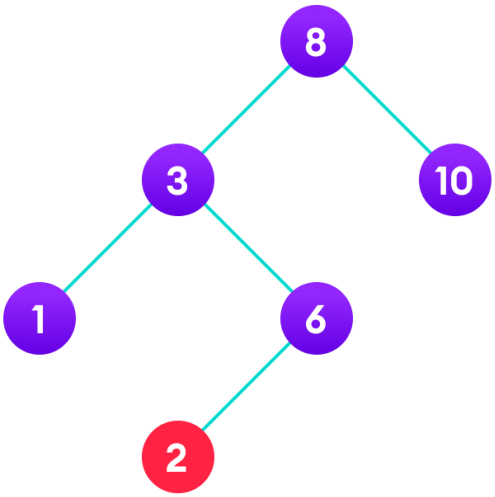
Here's a more formal definition:

Binary Search Tree is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- **The left and right subtree each must also be a binary search tree.**

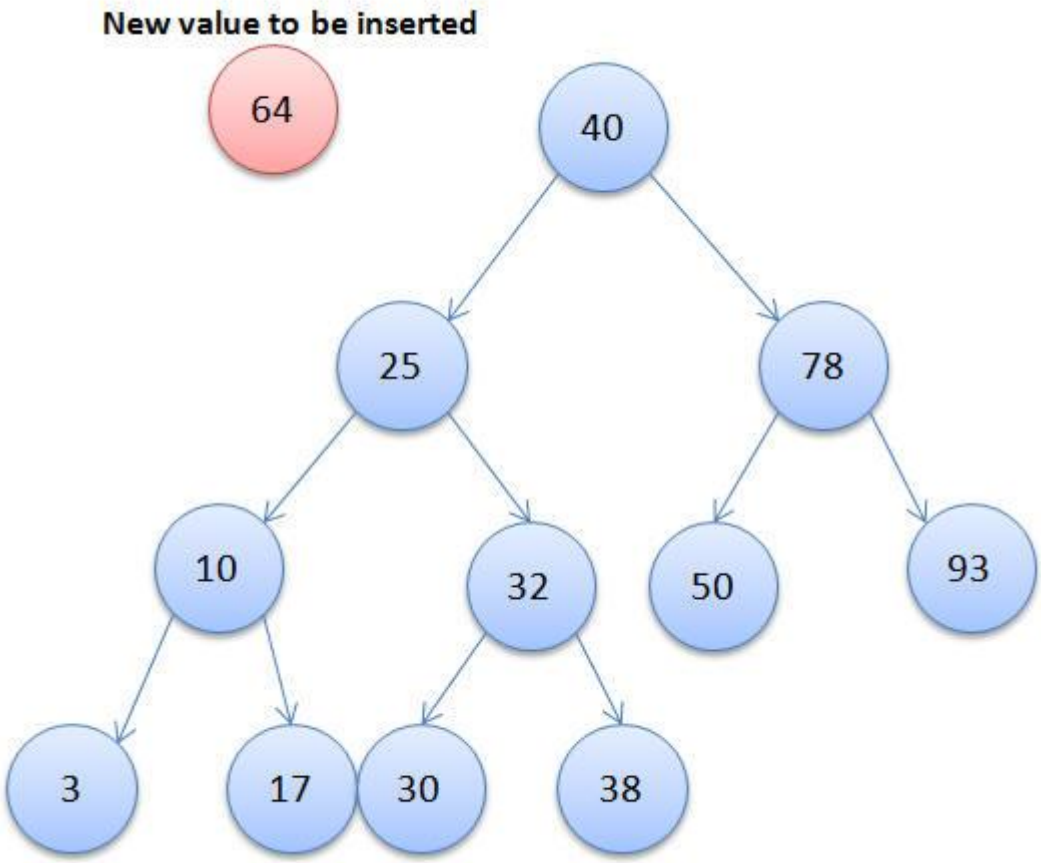
On the next page you'll find an example of valid and invalid BSTs. Question: Assuming the node with the key of '2' was the last one to be inserted, where should it have gone?

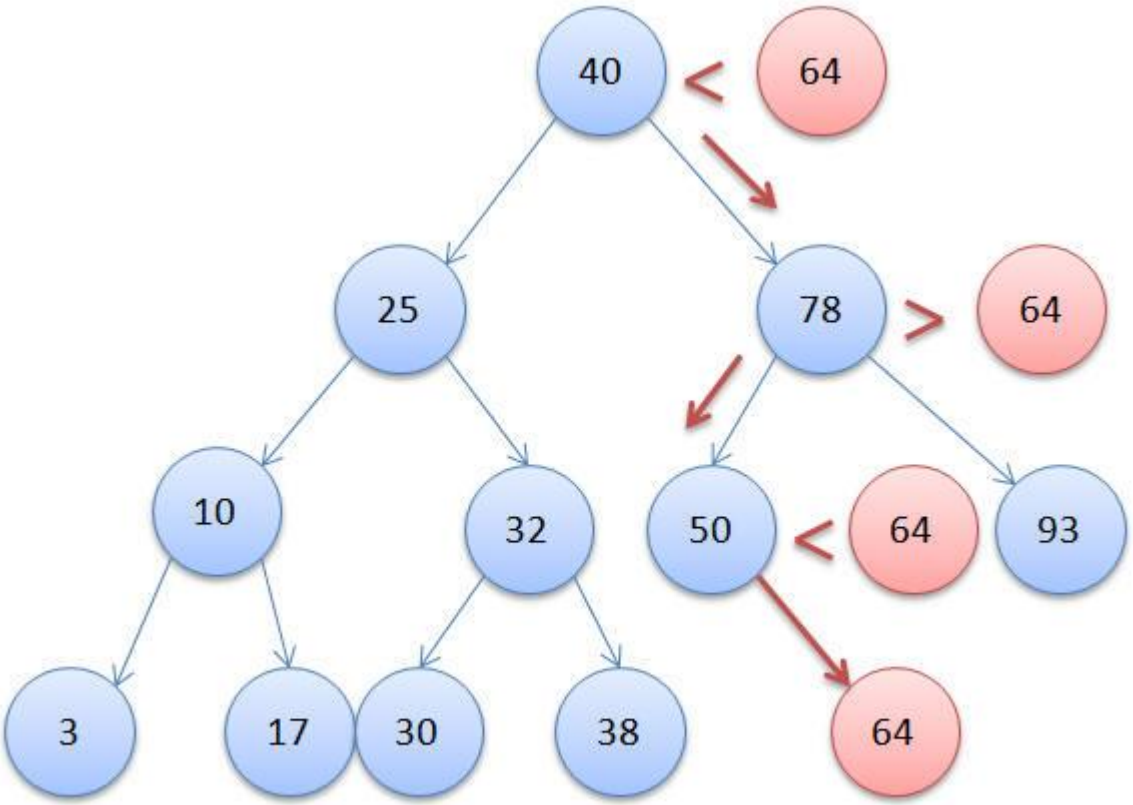

8
3
10
1
6
4


8
3
10
1
6
2

Building a Binary Search Tree (BST)

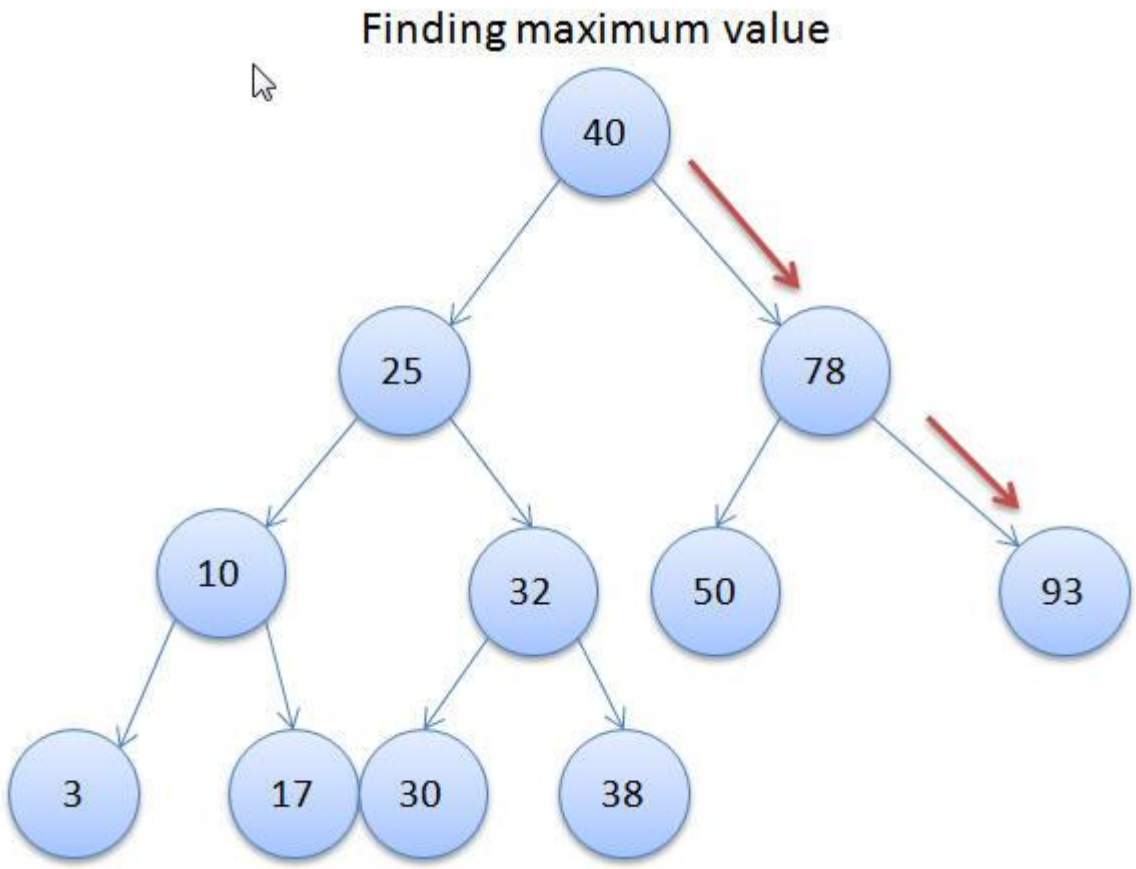
Now that we have seen how a BST looks, let me show you how one can build a BST in terms of inserting new nodes into a pre-existing tree. **The basic idea is that at each node we compare with the value being inserted. If the value is lesser then we traverse through the left sub tree and if the value is greater we traverse through the right subtree.** Suppose we have to insert the value **64** in the above BST, let's look at the nodes traversed before its inserted at the right place:

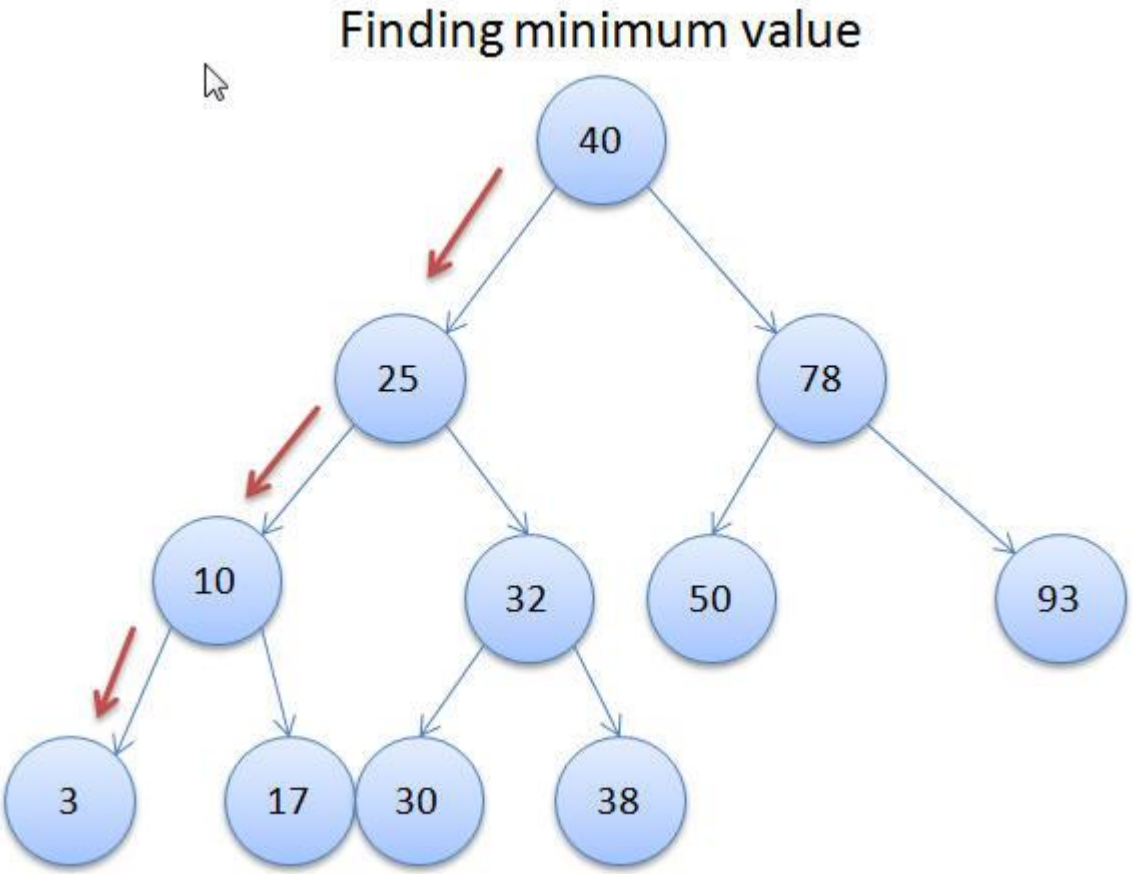



40
64
25
78
64
10
32
50
64
93
3
17
30
38
64

Finding Maximum and Minimum Value in BST

You may have noticed in the above example that the leftmost node has the lowest value and the rightmost node has the highest value. This is due to the sorted nature of the tree.




Finding minimum value
40
25
78
10
32
50
93
3
17
30
38

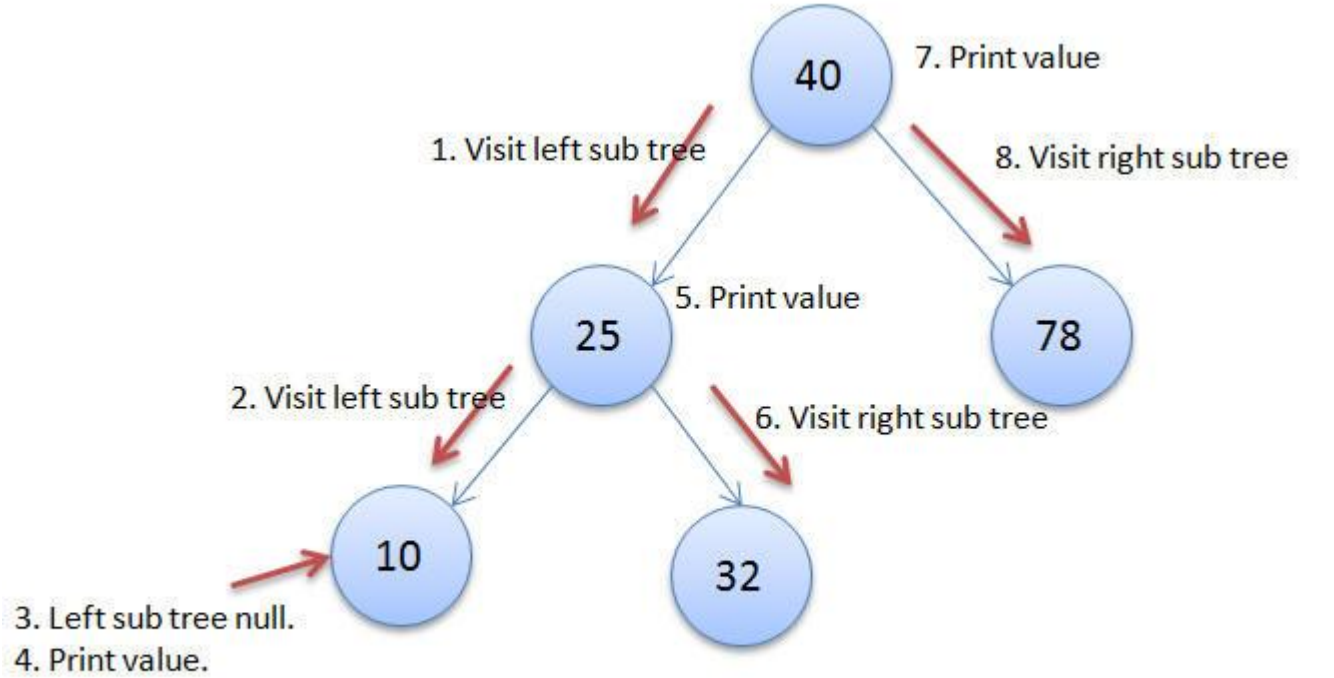
Traversing the Binary Search Tree (BST)

Traversing the tree or BST in this case is visiting each of the nodes present in the tree and performing some operation with the value present in the node which in this case will be printing the value present in the node. **When we traverse the tree we have to visit the value present in the node, then node's right sub tree and the left sub tree**. Visiting the right and left sub tree will be a ***recursive*** operation. The order in which we perform the three operations i.e. visiting the value, right sub tree and left sub tree gives rise to three traversal techniques:

1. Inorder Traversal
2. Preorder Traversal
3. Postorder Traversal

Inorder Traversal

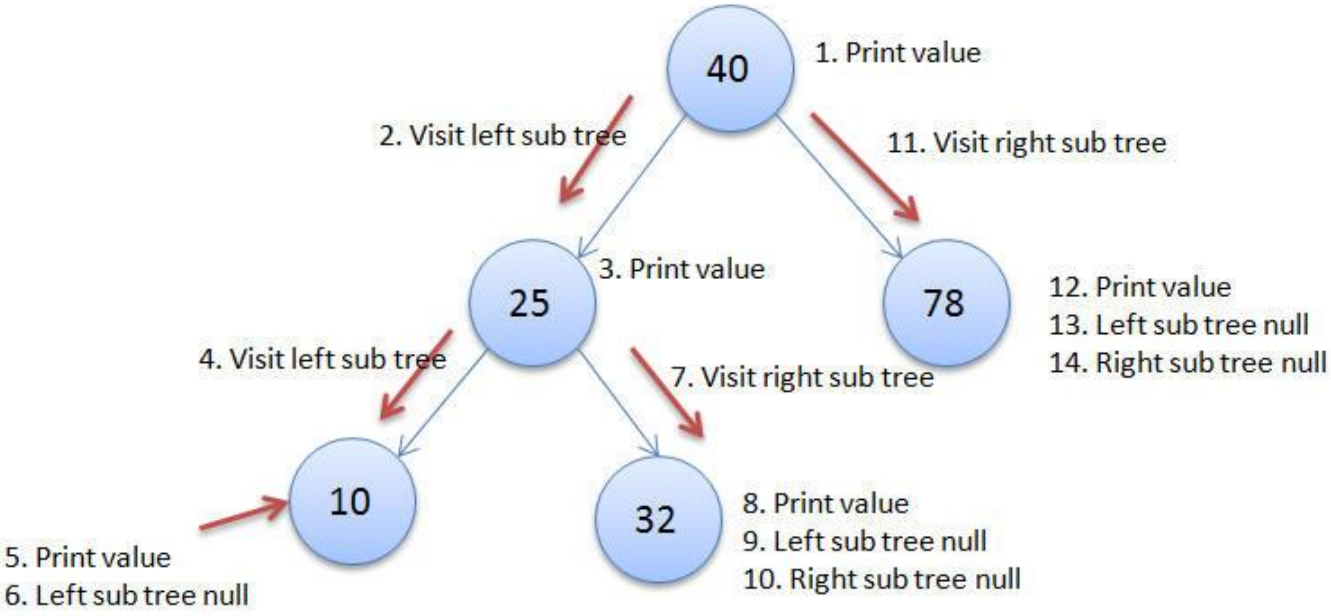
In this traversal the left sub tree of the given node is visited first, then the value at the given node is printed and then the right sub tree of the given node is visited. This process is applied ***recursively*** to all the nodes in the tree.



The above INORDER traversal gives: **10, 25, 32, 40, 78**

Preorder traversal

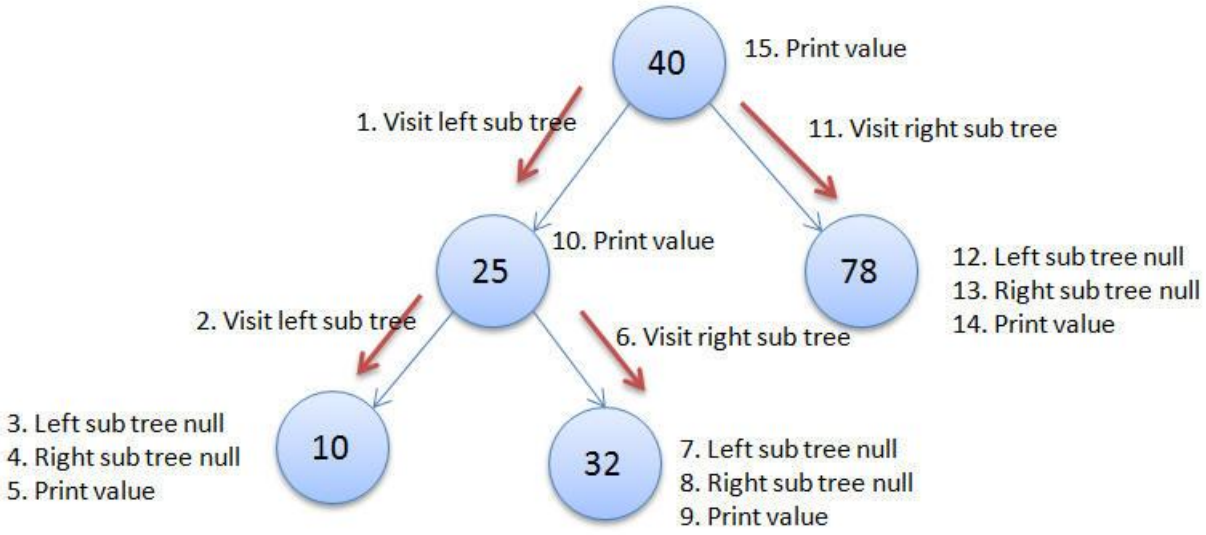
In this traversal the value at the given node is printed first and then the left sub tree of the given node is visited and then the right sub tree of the given node is visited. This process is applied recursively all the node in the tree until either the left sub tree is empty or the right sub tree is empty.



The above PREORDER traversal gives: **40, 25, 10, 32, 78**

Postorder Traversal

In this traversal the left sub tree of the given node is traversed first, then the right sub tree of the given node is traversed and then the value at the given node is printed. This process is applied recursively all the node in the tree until either the left sub tree is empty or the right sub tree is empty.



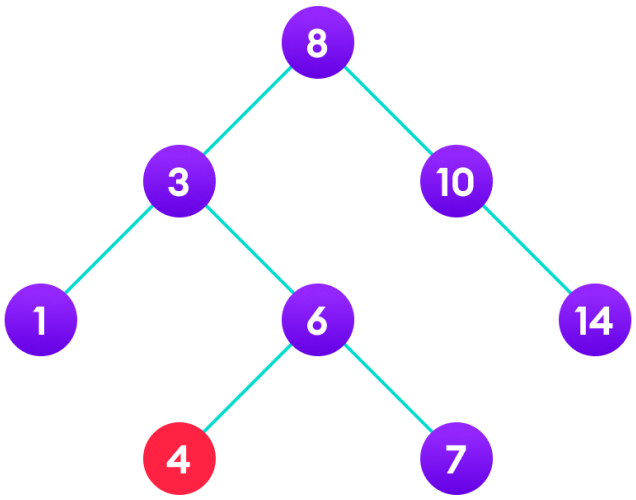
The above POSTORDER traversal gives: **10, 32, 25, 78, 40**

Deletion Operation (from programiz.com)

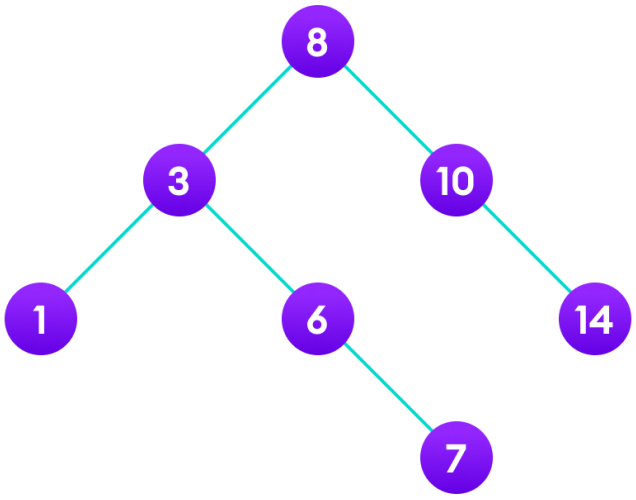
There are three cases for deleting a node from a binary search tree.

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



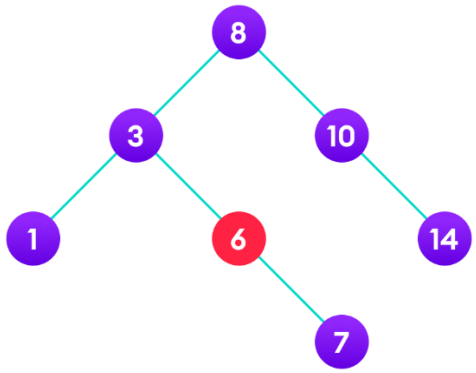
Delete the node

Case II

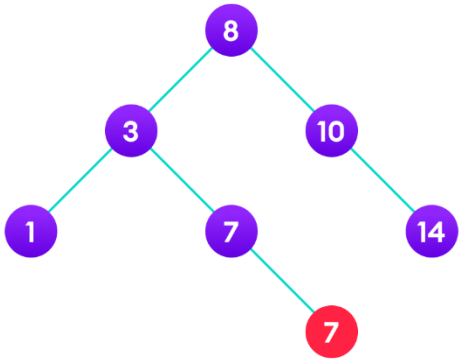
In the second case, the node to be deleted lies has a single child node. In such a case follow the steps below:

Replace that node with its child node.

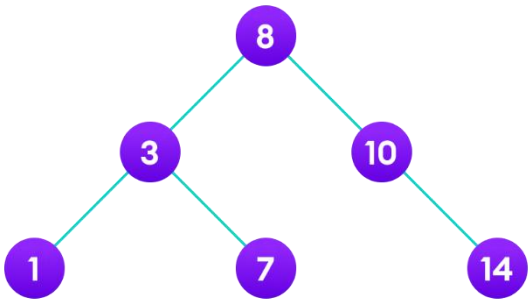
Remove the child node from its original position.



6 is to be deleted



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



Final tree

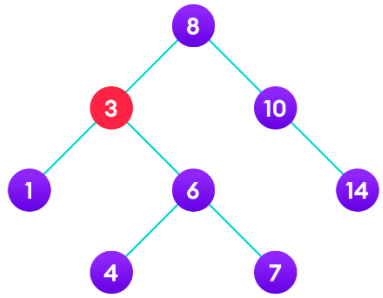
Case III

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

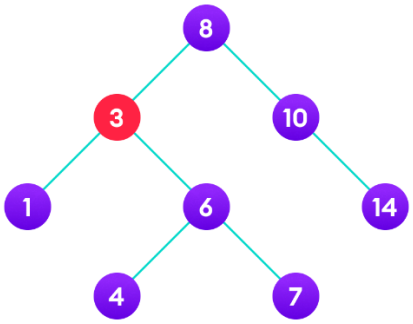
Get the inorder successor of that node.

Replace the node with the inorder successor.

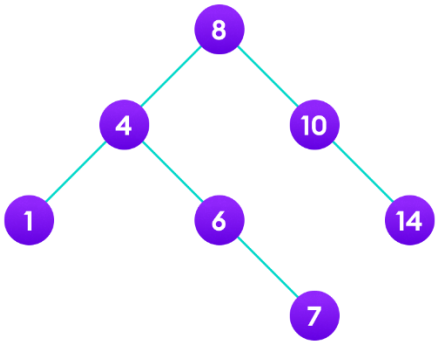
Remove the inorder successor from its original position.



3 is to be deleted



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



Delete the inorder successor

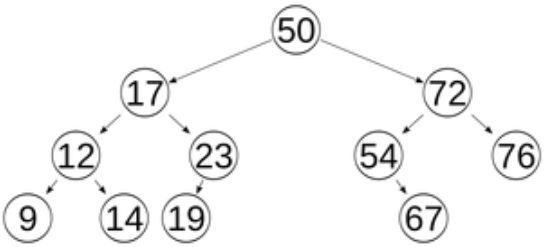
Why do we need binary search trees?

Let's say we have a list as an input and we need to find some element in it. What is the best way to do it?

Naive implementation could be simply iterating through all the elements and check each one. That maybe be fine for small lists and simple use cases. But can we do better?

Probably not with a simple random array. However if we sort it beforehand it is possible to "divide and conqueror". That is we can check element in the middle and if the one we are looking for is larger we do the same with sub list n/2 to n (where n is number of elements in the list). So we cut through search space considerably (O(log n) complexity instead of O(n)).

That is fine but we need to sort the input before with some sorting algorithm. And if we are constantly modifying (inserting/deleting) the input and searching through it, it's not practical to sort it each time before search. Here is where binary search trees steps in. We modify the way data is kept. Not in a simple list/array but in a tree like data structure. Looking at the picture you probably already see why it is easier to search in such a structure.



However, a binary search tree can get unbalanced and then lose its efficiency. To solve that problem **self-balancing binary search trees** were invented. That's a new topic(??) that we'll be exploring.

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)