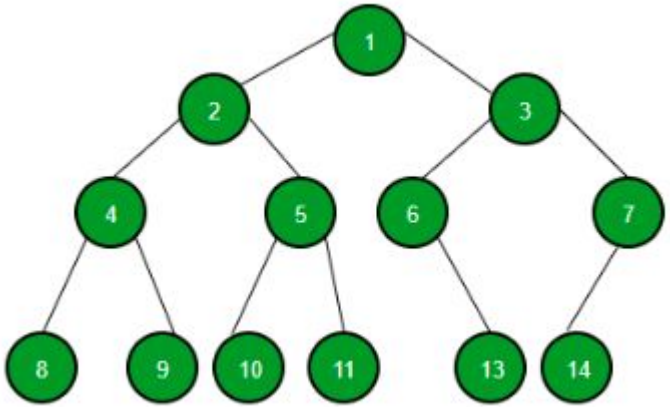
Binary Search Tree

Ar. Gör Elif Ece Erdem

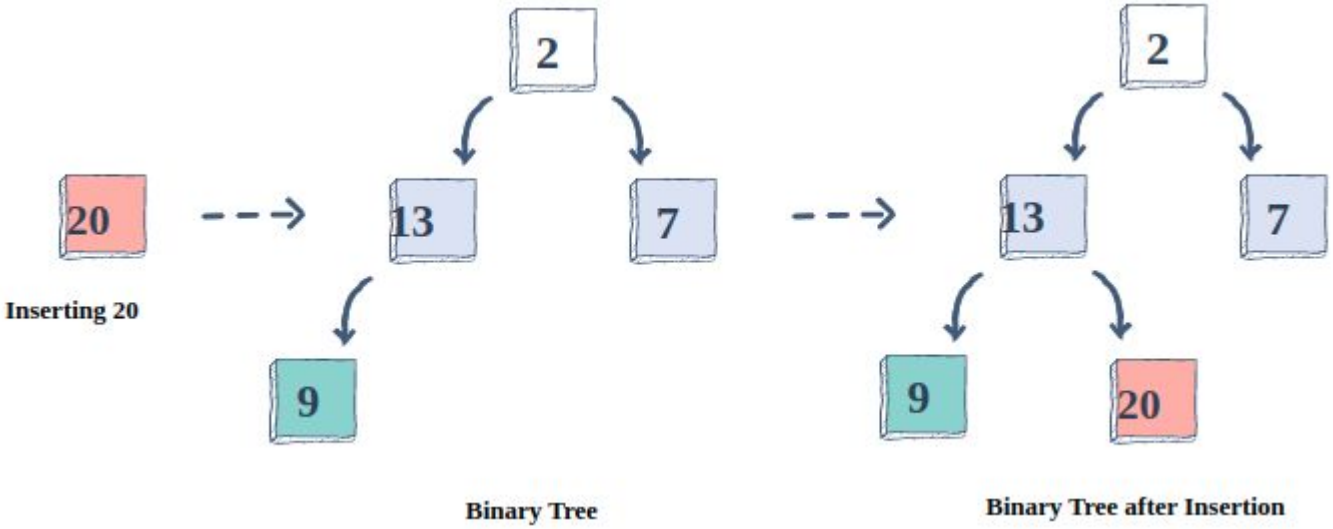
What is Binary Tree?

Binary Tree is defined as a tree data structure where each node has at most 2 children. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.



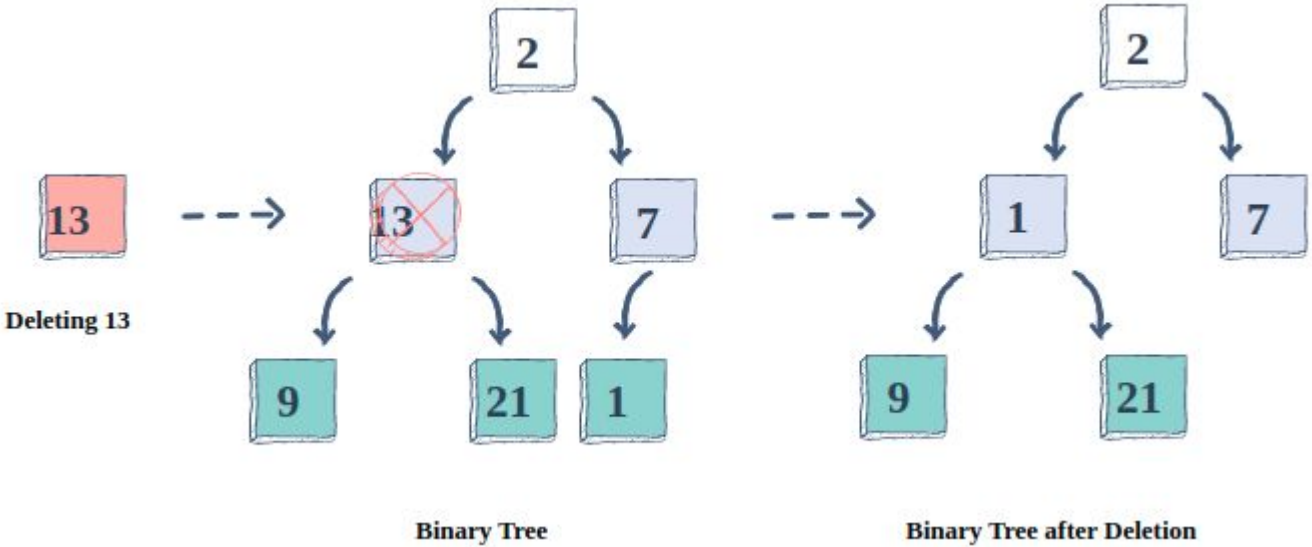
Binary Tree - Insertion

Upon finding an empty left *or* right child, the new element is inserted. By convention, the insertion always begins from the *left* child node.



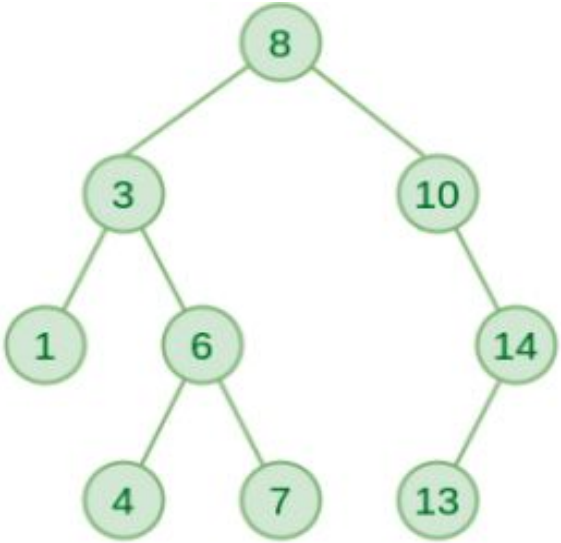
Binary Tree - Deletion

An element may also be removed from the binary tree. Since there is no particular order among the elements, upon deletion of a particular node, it is replaced with the right-most element.



What is Binary Search Tree?

A binary search tree is a data structure that allows for fast insertion, removal, and lookup of items while offering an efficient way to iterate them **in sorted order**.



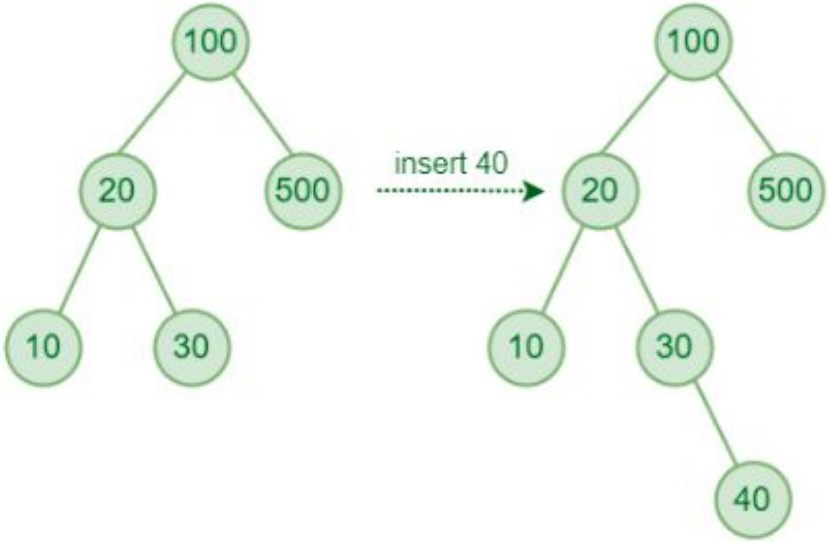
- The left subtree of a node contains only nodes with keys lesser than the node’s key.
- The right subtree of a node contains only nodes with keys greater than the node’s key.
- The left and right subtree each must also be a binary search tree.

Binary Search Tree - Insertion

Insertion in BST involves the comparison of the key values.

If the key value is lesser than or equal to root key then go to left subtree, find an empty space following to the search algorithm and insert the data

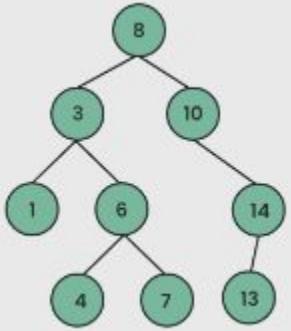
and if the key is greater than root key then go to right subtree, find an empty space following to the search algorithm and insert the data.



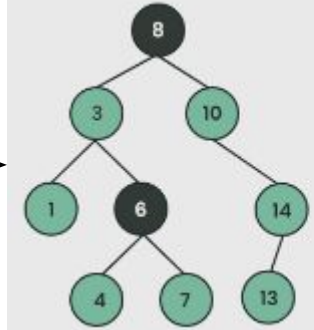
Binary Search Tree - Search

1. Check if tree is NULL, if the tree is not NULL then follow the following steps.
2. Compare the key to be searched with the root of the BST.
3. If the key is lesser than the root then search in the left subtree.
4. If the key is greater than the root then search in the right subtree.
5. If the key is equal to root then, return and print search successful.
6. Repeat step 3, 4 or 5 for the obtained subtree.

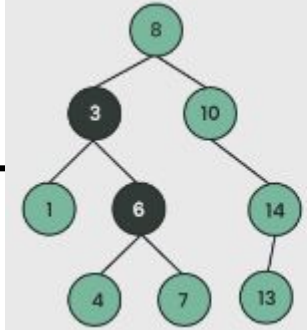
Binary Search Tree - Search



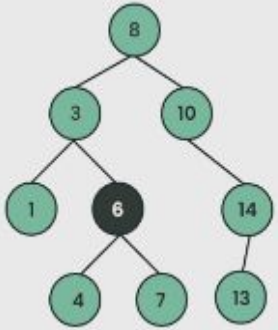
Consider The Following BST
Key = 6



Compare Key With Root, i.e 8 as 6<8, search in left subtree of 8



As Key (6) Is Greater Than 3, Search In The Right Subtree Of 3

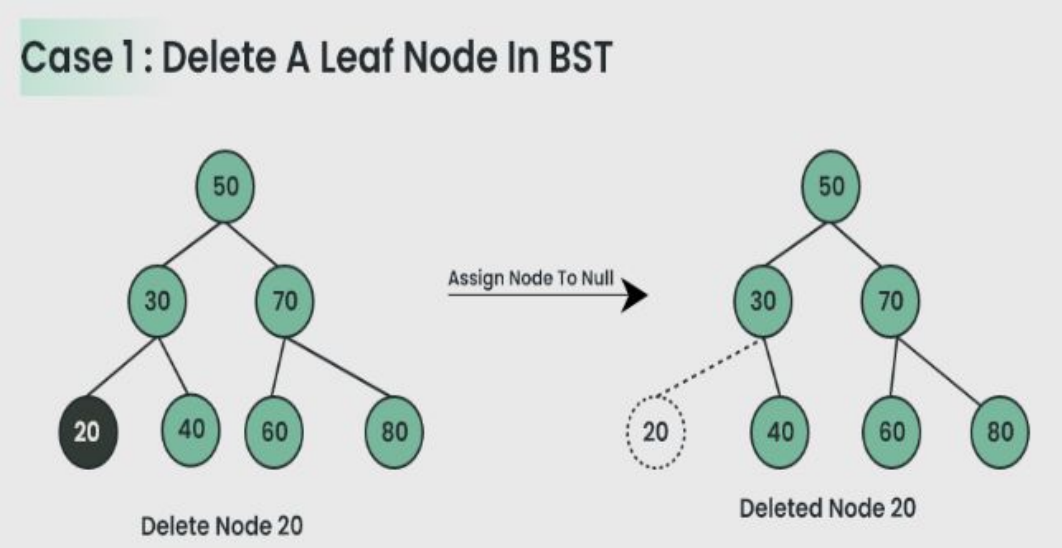


As 6 Is Equal To Key (6), So We Have Found The Key

Binary Search Tree - Deletion

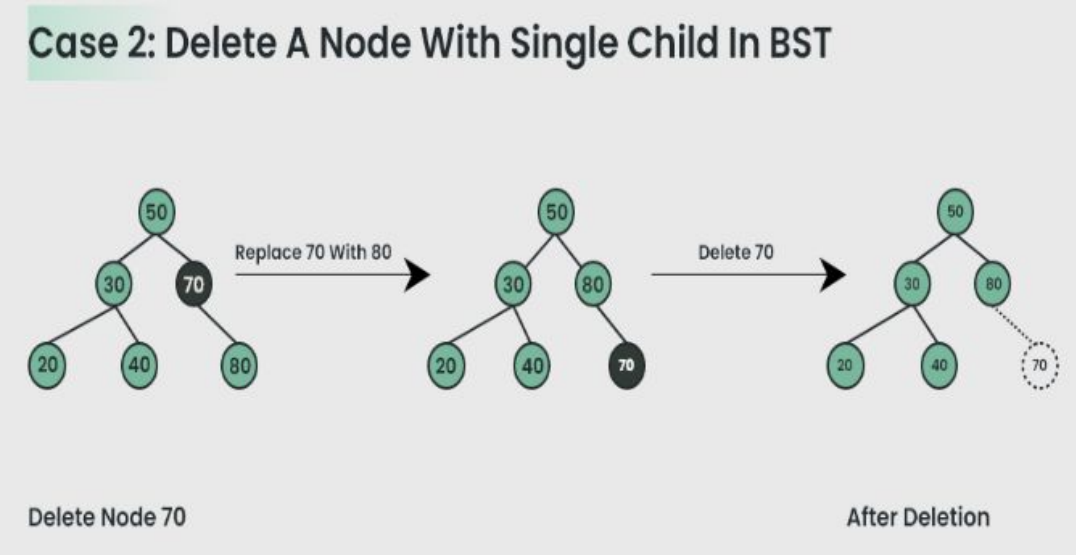
First, search the key to be deleted using searching algorithm and find the node. Then, find the number of children of the node to be deleted.

- **Case 1- If the node to be deleted is leaf node:** If the node to be deleted is a leaf node, then delete it.



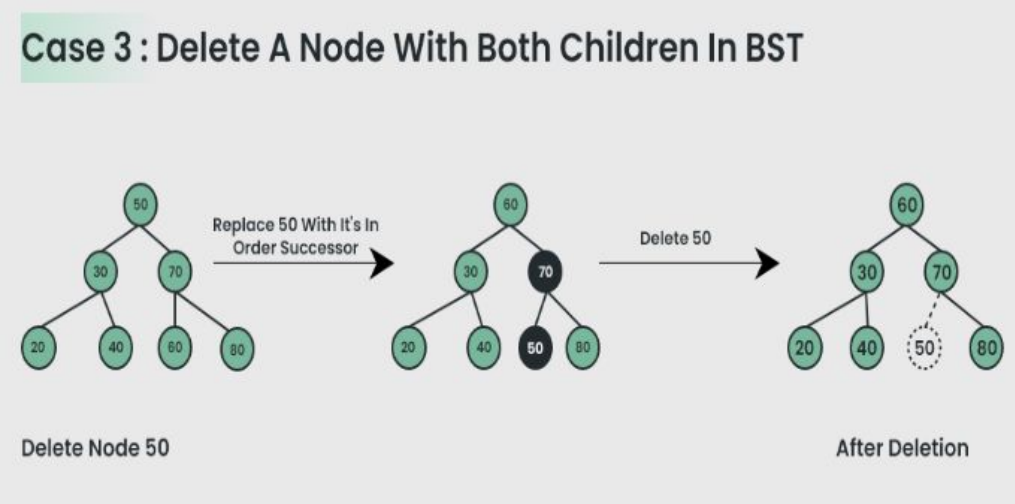
Binary Search Tree - Deletion

- **Case 2- If the node to be deleted has one child:** If the node to be deleted has one child then, delete the node and place the child of the node at the position of the deleted node.



Binary Search Tree - Deletion

- **Case 3- If the node to be deleted has two children:** If the node to be deleted has two children then, find the **inorder successor** or inorder predecessor of the node according to the nearest capable value of the node to be deleted. Delete the inorder successor or predecessor using the above cases. Replace the node with the inorder successor or predecessor.



Inorder successor: smallest value in the right subtree

Inorder predecessor: largest value in the left subtree

Time Complexities

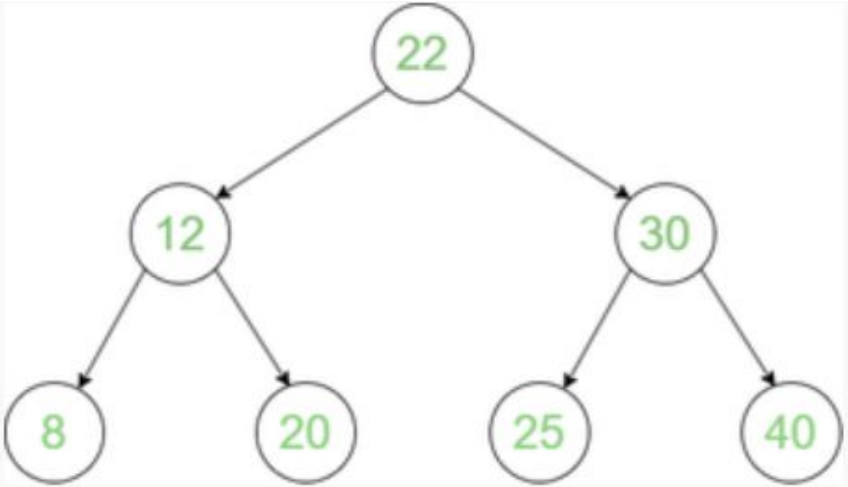
Operations	Best case time complexity	Average case time complexity	Worst case time complexity
Insertion	O(log n)	O(log n)	O(n)
Deletion	O(log n)	O(log n)	O(n)
Search	O(log n)	O(log n)	O(n)

Binary Search Tree - Depth First Traversal
Inorder Traversal

*"At first traverse **left subtree** then visit the **root** and then traverse the **right subtree**."*

Follow the below steps to implement the idea:

- Traverse left subtree
- Visit the root and print the data.
- Traverse the right subtree



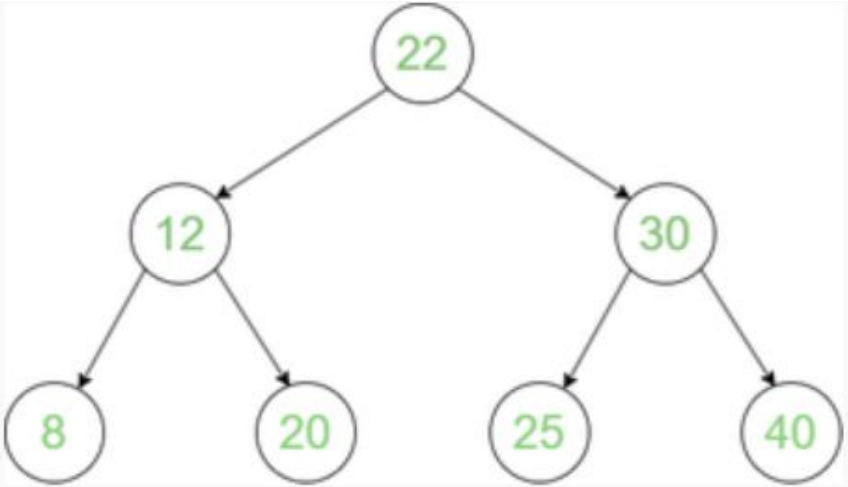
Inorder Traversal: *8 12 20 22 25 30 40*

Binary Search Tree - Depth First Traversal
Preorder Traversal

*“At first visit the **root** then traverse left subtree and then traverse the **right subtree**.”*

Follow the below steps to implement the idea:

- Visit the root and print the data.
- Traverse left subtree
- Traverse the right subtree



Preorder Traversal: *22 12 8 20 30 25 40*

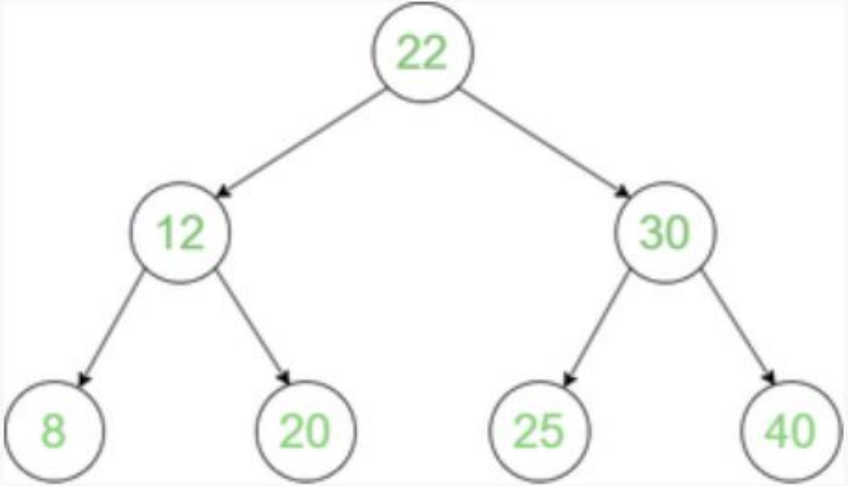
Binary Search Tree - Depth First Traversal
Postorder Traversal

*"At first traverse **left subtree** then traverse the **right subtree** and then visit the **root**."*

Follow the below steps to implement the idea:

- Traverse left subtree
- Traverse the right subtree
- Visit the root and print the data

Postorder Traversal: *8 20 12 25 40 30 22*



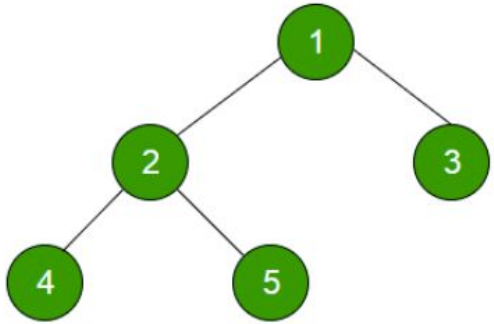
Binary Search Tree - Breadth First Traversal
Level Order Traversal

"Traverse a Tree such that all nodes present in the same level are traversed completely before traversing the next level."

Naive approach:

- Find height of tree.
- For each level, run a recursive function by maintaining current height.
- Whenever the level of a node matches, print that node.

Level Order Traversal:
1
2 3
4 5



Binary Search Tree - Breadth First Traversal
Level Order Traversal

"Traverse a Tree such that all nodes present in the same level are traversed completely before traversing the next level."

Using Queue:

- Push the nodes of a lower level in the queue.
- When any node is visited, pop that node from the queue
- and push the child of that node in the queue.

Level Order Traversal:
1
2 3
4 5