# Lesson:


# Binary Search Tree

# Pre Requisites:

- Trees in Java
- Recursion in Java

# Topics to be covered

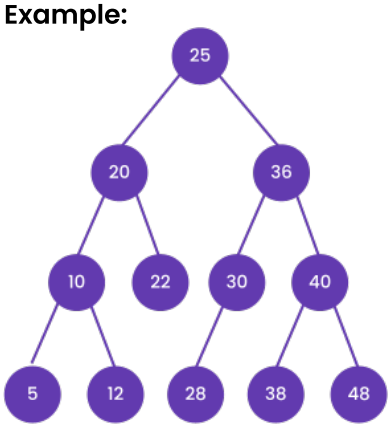
- Introduction to Binary Search Tree
- Insertion in BST
- Search In BST
- Deletion in BST
- Interview problem: checkBST
- Interview problem: kth smallest element in BST

# Introduction to Binary Search Tree(BST)

A Binary Search Tree (BST) is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child, and the topmost node in the tree is called the root.
It additionally satisfies the binary search property, which states that the key in each node must be greater than or equal to any key stored in the left subtree, and less than or equal to any key stored in the right subtree.
Advantages of BST:

- Binary search trees allow fast lookup, addition, and removal of items.
- They keep their keys in sorted order so that lookup and other operations can use the principle of binary search: when looking for a key in a tree (or a place to insert a new key), they traverse the tree from root to leaf, making comparisons to keys stored in the nodes of the tree and deciding, based on the comparison, to continue searching in the left or right subtrees.
- On average, this means that each comparison allows the operations to skip about half of the tree so that each lookup, insertion, or deletion takes time proportional to the logarithm of the number of items stored in the tree.
- This is much better than the linear time required to find items by key in an (unsorted) array but slower than the corresponding operations on hash tables.

**Example:**

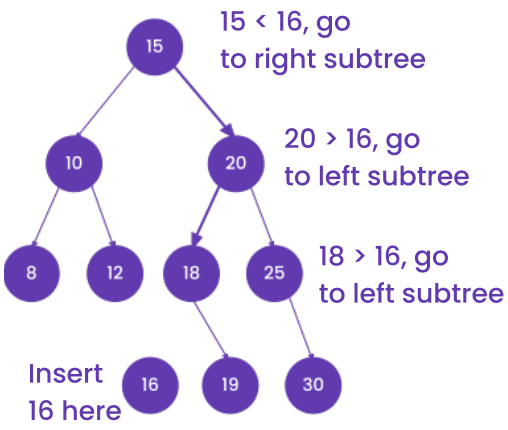



Binary Search Tree

Special property of BST: The inorder of BST is always in sorted order(ascending order). We will use this property in various examples and problems to verify our solution

# Insertion in BST

We will be given a root node and a sample node. Our task is to insert this node in BST such that it doesn't violate any property of BST. For eg if we want to insert 16 in the current tree then this is how the tree would look like.




**Insert (root, 16)**

We will be using recursive as well as iterative approach to solve this problem:

**Recursive approach:**

- When looking for a place to insert a new key, traverse the tree from root-to-leaf, making comparisons to keys stored in the tree's nodes and deciding based on the comparison to continue searching in the left or right subtrees.
- In other words, we examine the root and recursively insert the new node to the left subtree if its key is less than that of the root or the right subtree if its key is greater than or equal to the root.

**LP_Code1.java**

**Output:** We printed inorder of the given tree to verify if it's a BST or not.

```
8 10 12 15 16 20 25
```

**Iterative approach:**

- Another way to explain the insertion is to insert a new node into the tree. Initially, the key is compared with that of the root.
- If its key is less than the root's, it is then compared with the root's left child's key.
- If its key is greater, it is compared with the root's right child.
- This process continues until the new node is compared with a leaf node, and then it is added as this node's right or left child, depending on its key:
- if the key is less than the leaf's key, then it is inserted as the leaf's left child; otherwise, as to the leaf's right child.

**LP_Code2.java**

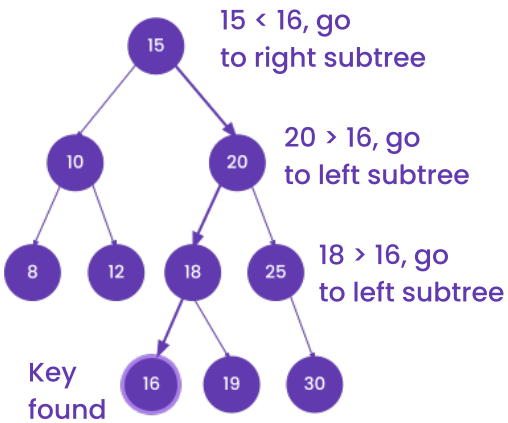
**Output:** We printed inorder of the given tree to verify if it's a BST or not.

```
8 10 12 15 16 20 25
```

# Search In BST

The algorithm should return the parent node of the key and print if the key is the left or right node of the parent node. If the key is not present in the BST, the algorithm should be able to determine that.

**For eg.** let's try to find 16 in the above created BST




**Search (root, 16)**

**Approach:**

- We begin by examining the root node. If the tree is null, the key we are searching for does not exist in the tree.
- Otherwise, if the key equals that of the root, the search is successful, we return the node. If the key is less than that of the root, we search the left subtree. Similarly, if the key is greater than that of the root, we search the right subtree.
- This process is repeated until the key is found or the remaining subtree is null. If the searched key is not found after a null subtree is reached, then the key is not present in the tree.

LP_Code3.java

**Output:** Searching 25 in the above BST

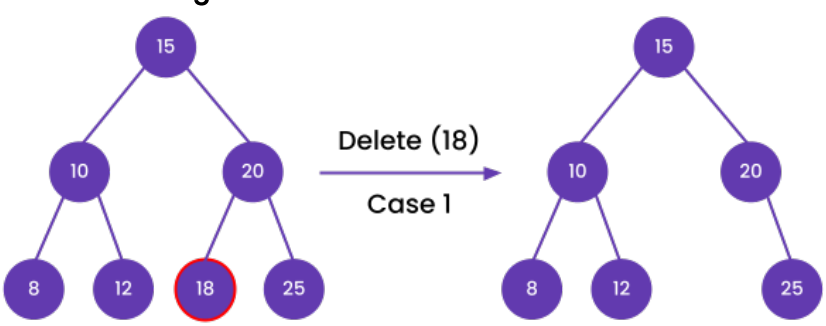
```
The given key is the right node of the node with key 20
```

On average, binary search trees with n nodes have O(log(n)) height. However, in the worst case, binary search trees can have O(n) height (for skewed trees where all the nodes except the leaf have one and only one child)

# Deletion in BST

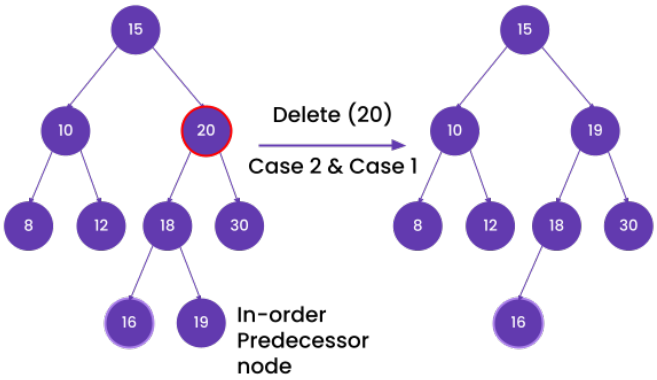
The idea is to delete a node from BST, and post that the order of BST should be maintained.
There would be 3 cases in deletion from a BST.

**Case 1: Deleting a node with no children: remove the node from the tree.**

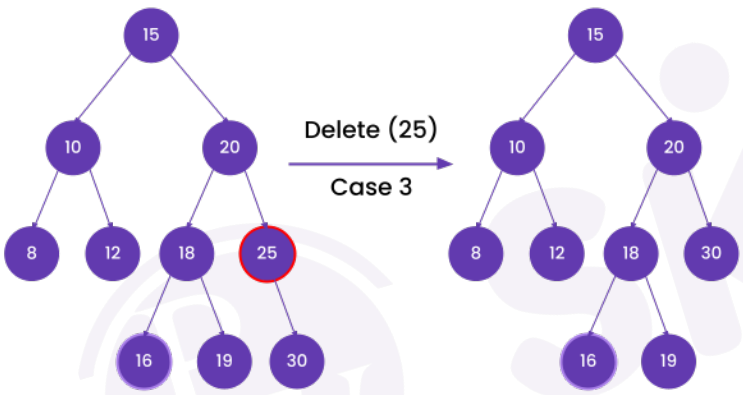


**Case 2: Deleting a node with two children: call the node to be deleted N.**

- Do not delete N. Instead, choose either its **inorder successor** node or its inorder predecessor node, R.
- Copy the value of R to N, then recursively call delete on R until reaching one of the first two cases.
- If we choose the inorder successor of a node, as the right subtree is not NULL (our present case is a node with 2 children), then its inorder successor is a node with the least value in its right subtree, which will have at a maximum of 1 subtree, so deleting it would fall in one of the first 2 cases.




**Case 3: Deleting a node with one child: remove the node and replace it with its child.**




**Note:** nodes with children are harder to delete. As with all binary trees, a node's inorder successor is its right subtree's leftmost child, and a node's inorder predecessor is the left subtree's rightmost child. In either case, this node will have zero or one child. Delete it according to one of the two simpler cases above.

**LP_Code4.java**

**Output:** Deleting 16 from the above BST and printing the inorder.

```
8 10 12 15 20
```

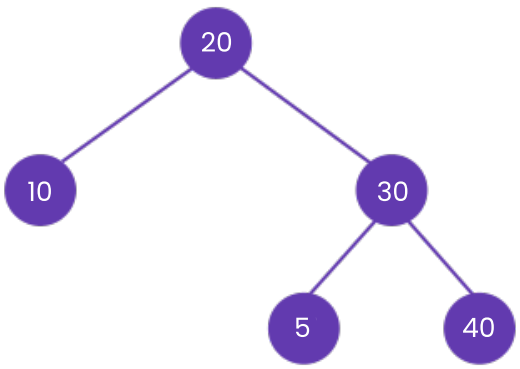
# Interview Problem: checkBST

The problem is to check whether for a given binary tree it is a BST or not.
The BST should hold this, the current node should be greater than the left node and smaller than the right node and this property should hold true for every node.

The easiest way is to find the inorder and check if it's not in sorted order then it's not a BST. This would take O(n) time as we would travel each node of the tree and construct an array.
Another approach is the recursion approach.We would go recursively to solve this problem and using property

of BST:

Traverse the tree, at every node check whether the node contains a value larger than the value at the left child and smaller than the value on the right child – does not work for all cases. Consider the following tree:




In the tree above, each node meets the condition that the node contains a value larger than its left child and smaller than its right child hold, and yet it's not a BST: the value 5 is on the right subtree of the node containing 20, a violation of the BST property.
Instead of deciding based solely on a node's values and its children, we also need information flowing down from the parent. In the tree above, if we could remember about the node containing the value 20, we would see that the node with value 5 is violating the BST property contract.
So, the condition we need to check at each node is:

- If the node is the left child of its parent, it must be smaller than (or equal to) the parent, and it must pass down the value from its parent to its right subtree to make sure none of the nodes in that subtree is greater than the parent.
- If the node is the right child of its parent, it must be larger than the parent, and it must pass down the value from its parent to its left subtree to make sure none of the nodes in that subtree is lesser than the parent.

**LP_Code5.java**

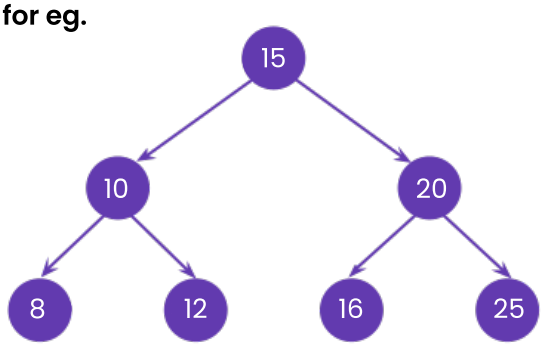
**Output:**

```
The tree is a BST.
```

The time complexity of the above solution is O(n), where n is the size of the BST, and requires space proportional to the tree's height for the call stack.

# Interview Problem: Kth smallest element in BST

The problem itself says that we need to return the kth smallest element in BST. so our counting would start from the smallest element of BST.
**for eg.**



For example, the 4th smallest node in the following BST is 15, and the 6th smallest is 20. The 8th smallest node does not exist.

Approach:

- The idea is to traverse the BST in an inorder fashion since the inorder traversal visits the nodes of a BST in the sorted order.
- Maintain a counter along with recursion that keeps track of the visited nodes, and when that counter reaches k, return that node.

The code is optimized to visit the right subtree only when the k'th smallest is not found in the left subtree.

**LP_Code6.java**

**Output:**

```
4'th smallest node is 15
```

# Next Class Teasers

- Priority queue
- Dynamic Programming