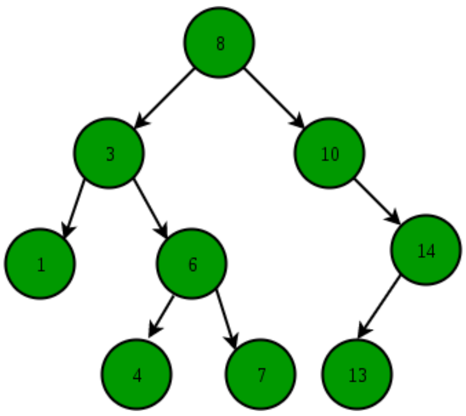
Binary Search Tree: Definition, Operations: Search, Insert, Min, Max, Successor, Delete

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.



Binary Search Tree Operation:

1. Search Operation-

There are three cases when we search for any element is an sorted array(Linear search):

Case 1 : element we are searching is in the beginning of an array (at index 0). This is a best case because we get that element just in 1 comparison.

Case 2: element we are searching is in the ending of an array (at index n-1). This is a worst case because we get that element after.

Case 3: element we are searching is in the middle somewhere. (index 1 to n-2). This is an average case because we get that element in more comparison than best case but less comparison than worst case.

Binary search can help us to implement optimized search by adopting following rules.

Rules-

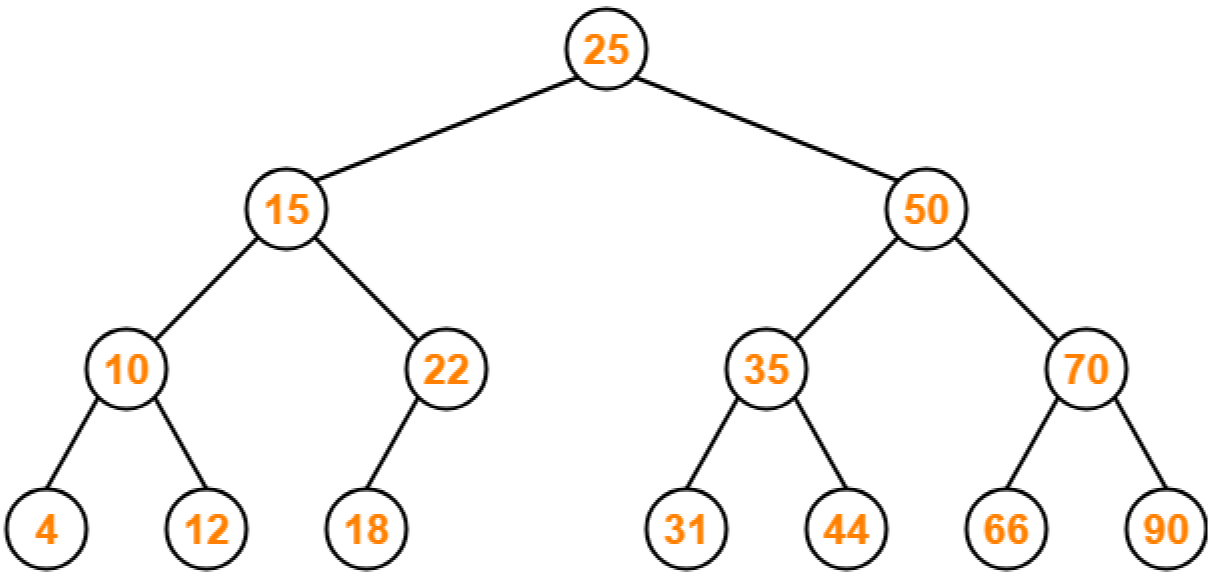
For searching a given key in the BST,

- Compare the key with the value of root node.
- If the key is present at the root node, then return the root node.

- If the key is greater than the root node value, then recur for the root node's right subtree.
- If the key is smaller than the root node value, then recur for the root node's left subtree.

Example-

Consider key = 45 has to be searched in the given BST-



Binary Search Tree

- We start our search from the root node 25.
- As 45 > 25, so we search in 25's right subtree.
- As 45 < 50, so we search in 50's left subtree.
- As 45 > 35, so we search in 35's right subtree.
- As 45 > 44, so we search in 44's right subtree but 44 has no subtrees.
- So, we conclude that 45 is not present in the above BST.

2. Insertion Operation-

Rules-

The insertion of a new key always takes place as the child of some leaf node.

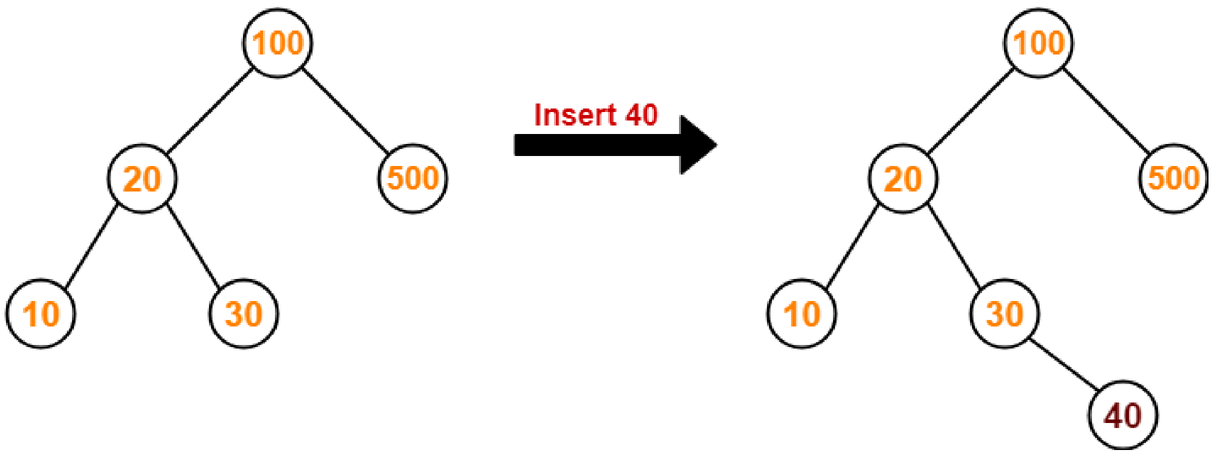
For finding out the suitable leaf node,

- Search the key to be inserted from the root node till some leaf node is reached.

- Once a leaf node is reached, insert the key as child of that leaf node.

Example-

Consider the following example where key = 40 is inserted in the given BST-



- We start searching for value 40 from the root node 100.
- As 40 < 100, so we search in 100's left subtree.
- As 40 > 20, so we search in 20's right subtree.
- As 40 > 30, so we add 40 to 30's right subtree.

3. Deletion Operation-

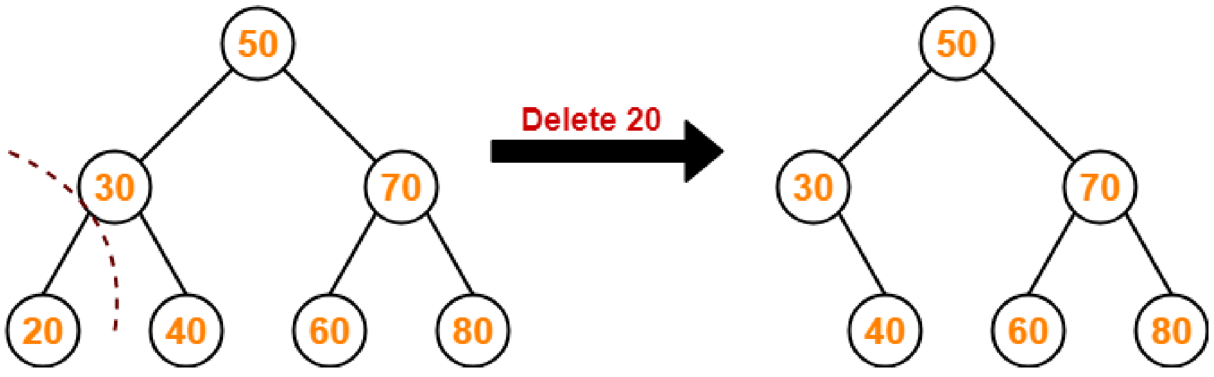
When it comes to deleting a node from the binary search tree, following three cases are possible-

Case-01: Deletion Of A Node Having No Child (Leaf Node)-

Just remove / disconnect the leaf node that is to deleted from the tree.

Example-

Consider the following example where node with value = 20 is deleted from the BST-

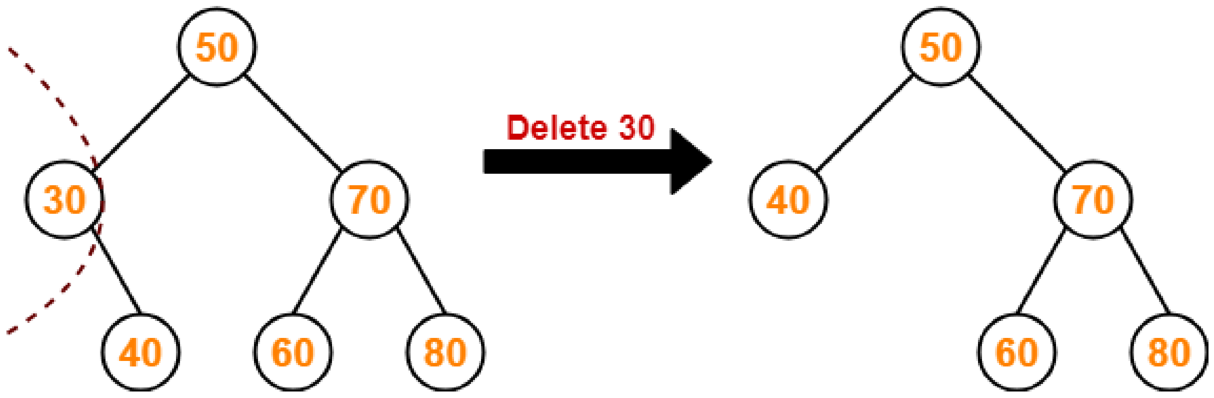


Case-02: Deletion Of A Node Having Only One Child-

Just make the child of the deleting node, the child of its grandparent.

Example-

Consider the following example where node with value = 30 is deleted from the BST-



Case-02: Deletion Of A Node Having Two Children-

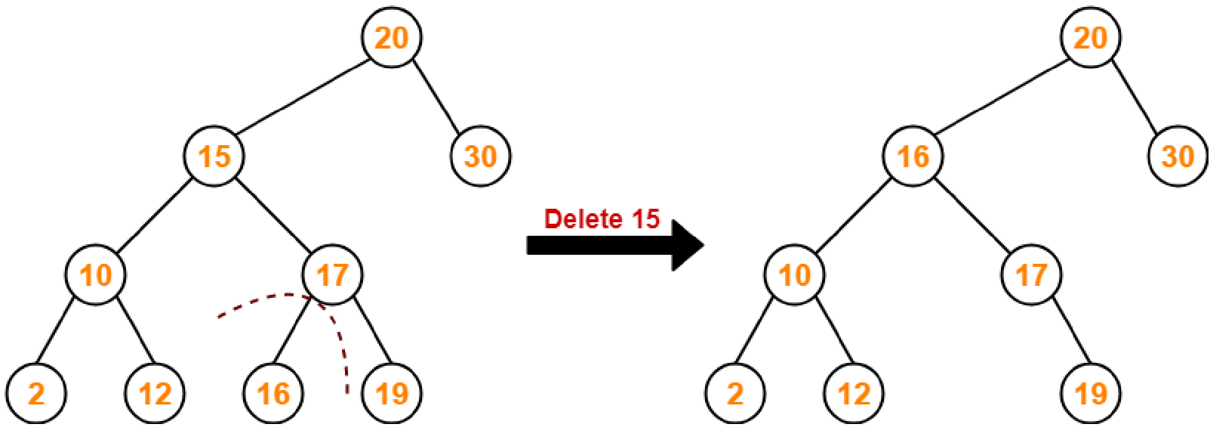
A node with two children may be deleted from the BST in the following two ways-

Method-01:

- Visit to the right subtree of the deleting node.
- Pluck the least value element called as inorder successor.
- Replace the deleting element with its inorder successor.

Example-

Consider the following example where node with value = 15 is deleted from the BST-



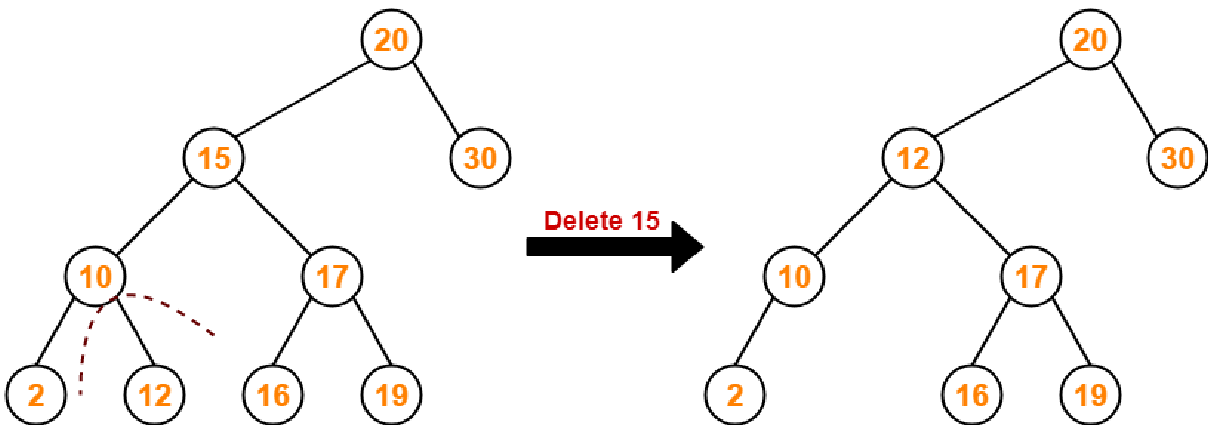
Method-02:

- Visit to the left subtree of the deleting node.

- Pluck the greatest value element called as inorder predecessor.
- Replace the deleting element with its inorder predecessor.

Example-

Consider the following example where node with value = 15 is deleted from the BST-



4. Maximum:

In Binary Search Tree, we can find maximum by traversing right pointers until we reach the rightmost node. But in Binary Tree, we must visit every node to figure out maximum. So the idea is to traverse the given tree and for every node return maximum of 3 values.

- Node's data.
- Maximum in node's left subtree.
- Maximum in node's right subtree.

5. Minimum:

In Binary Search Tree, we can find minimum by traversing left pointers until we reach the leftmost node. But in Binary Tree, we must visit every node to figure out minimum. So the idea is to traverse the given tree and for every node return minimum of 3 values.

- Node's data.
- Minimum in node's left subtree.
- Minimum in node's right subtree.

Some Interesting Facts:

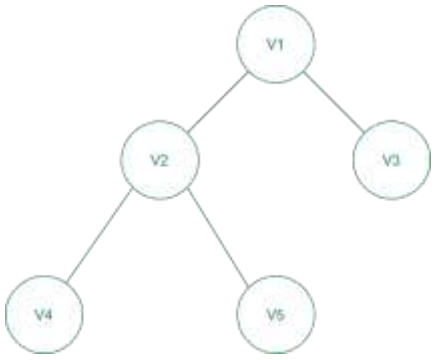
- Inorder traversal of BST always produces sorted output.
- We can construct a BST with only Preorder or Postorder or Level Order traversal. Note that we can always get inorder traversal by sorting the only given traversal.

- Number of unique BSTs with n distinct keys is Catalan Number as described in binary Tree.

Introduction to Height Balanced BSTs: AVL and Balance Mechanism

A **height-balanced binary tree** is defined as a binary tree in which the **height** of the left and the right subtree of any node differ by not more than 1.

Example: AVL tree, red-black tree are examples of height-balanced trees.



Height Balanced tree

Conditions for Height-Balanced Binary Tree:

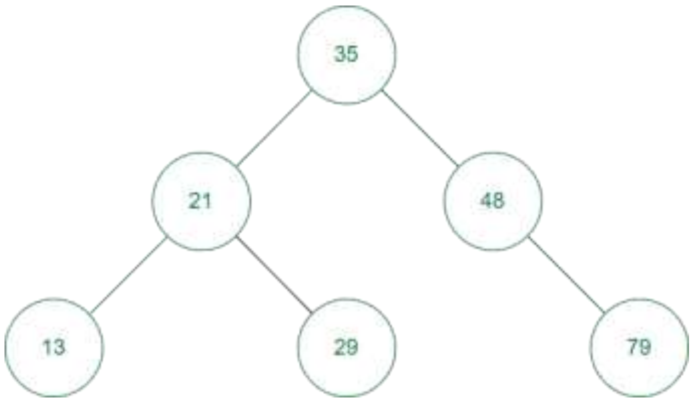
Following are the conditions for a height-balanced binary tree:

- The difference between the heights of the left and the right subtree for any node is not more than one. That is a balance factor can be either 0,1 and -1 only.
- The left subtree is balanced.
- The right subtree is balanced.

Note: An empty tree is also height-balanced.

Why do we need a Height-Balanced Binary Tree?

Let's understand the need for a balanced binary tree through an example.



- The above tree is a binary search tree and also a height-balanced tree.
- Suppose we want to want to find the value 79 in the above tree.
- First, we compare the value of the root node. Since the value of 79 is greater than 35, we move to its right child, i.e., 48. Since the value 79 is greater than 48, so we move to the right child of 48. The value of the right child of node 48 is 79. The number of hops required to search the element 79 is 2.
- Similarly, any element can be found with at most 2 jumps because the height of the tree is 2.

So it can be seen that any value in a balanced binary tree can be searched in **maximum logN** time where **N** is the number of nodes in the tree. But if the tree is not height-balanced then in the worst case, a search operation can take maximum **N** time(in case of skewed Tree).

Minimum number of nodes in a Height-Balanced Binary Tree at Height h is:

Num(h) = 1 + Num(h-1) + Num(h-2)

Where Num(h) is a function that shows minimum number of nodes required at height h

Applications of Height-Balanced Binary Tree:

- Balanced trees are mostly used for in-memory sorts of sets and dictionaries.
- Balanced trees are also used extensively in database applications in which insertions and deletions are fewer but there are frequent lookups for data required.
- It is used in applications that require improved searching apart from database applications.
- It has applications in storyline games as well.
- It is used mainly in corporate sectors where they have to keep the information about the employees working there and their change in shifts.

Advantages of Height-Balanced Binary Tree:

- It will improve the worst-case lookup time at the expense of making a typical case roughly one lookup less.
- As a general rule, a height-balanced tree would work better when the request frequencies across the data set are more evenly spread,
- It gives better search time complexity.

Disadvantages of Height-Balanced Binary Tree:

- Longer running times for the insert and remove operations.
- Must keep balancing info in each node.
- To find nodes to balance, must go back up in the tree.

AVL Tree

- AVL tree is a self-balancing binary search tree in which each node maintains extra information called a balance factor whose value is either -1, 0 or +1.
- AVL tree got its name after its inventor Georgy Adelson-Velsky and Landis.

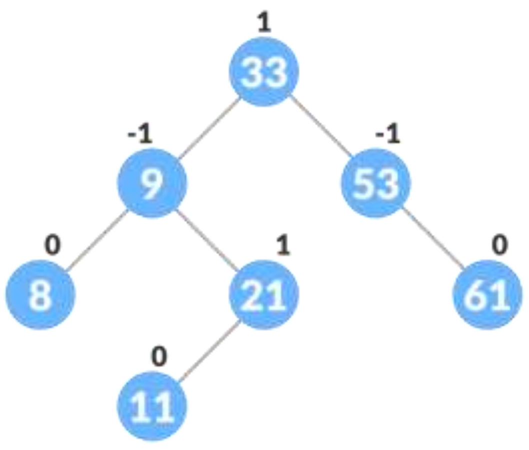
Why AVL Trees?

- Most of the BST operations (e.g., search, max, min, insert, delete.. etc) depends on height h of the BST.
- The cost of these operations may become almost the number of nodes for a skewed Binary tree.
- If we make sure that height of the tree be maximum logn after every insertion and deletion, then we can guarantee an upper bound of Logn for all these operations.
- The height of an AVL tree is always maximum Logn where n is the number of nodes in the tree

Balance Factor

- Balance factor of a node in an AVL tree is the difference between the height of the left subtree and that of the right subtree of that node.
- Balance Factor = (Height of Left Subtree - Height of Right Subtree) or (Height of Right Subtree - Height of Left Subtree)
- The self balancing property of an avl tree is maintained by the balance factor. The value of balance factor should always be -1, 0 or +1.

An example of a balanced avl tree is:



Avl tree

Operations on an AVL tree

Various operations that can be performed on an AVL tree are:

1. Rotating the subtrees in an AVL Tree

In rotation operation, the positions of the nodes of a subtree are interchanged.

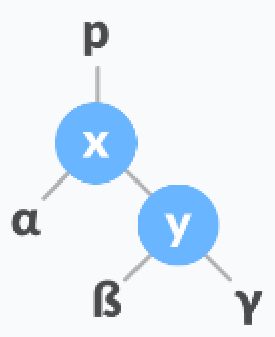
There are two types of rotations:

Left Rotate (Right – Right imbalance)

In left-rotation, the arrangement of the nodes on the right is transformed into the arrangements on the left node.

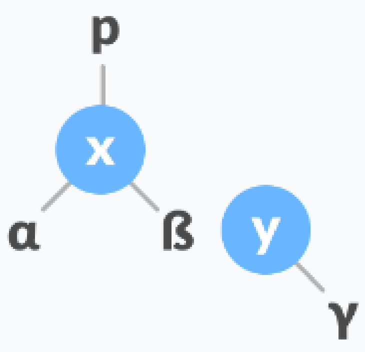
Algorithm:

1. Let the initial tree be:



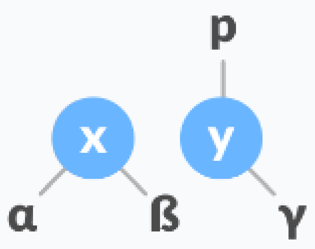
Left rotate

2. If y has a left subtree, assign x as the parent of the left subtree of y.



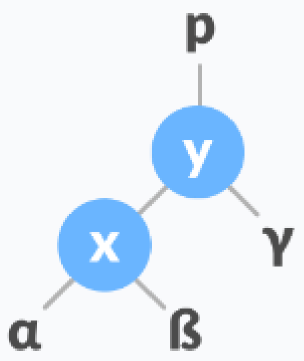
Assign x as the parent of the left subtree of y

3. If the parent of x is `NULL`, make y as the root of the tree.
4. Else if x is the left child of p, make y as the left child of p.
5. Else assign y as the right child of p.



Change the parent of x to that of y

6. Make y as the parent of x

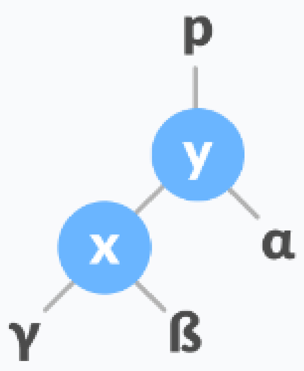


Assign y as the parent of x.

Right Rotate (Left Left imbalance)

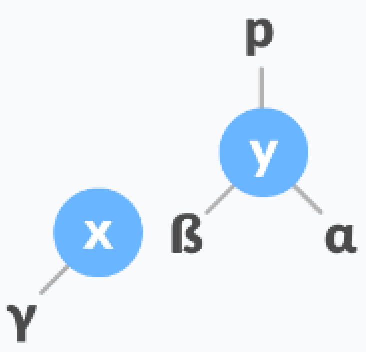
In left-rotation, the arrangement of the nodes on the left is transformed into the arrangements on the right node.

1. Let the initial tree be:



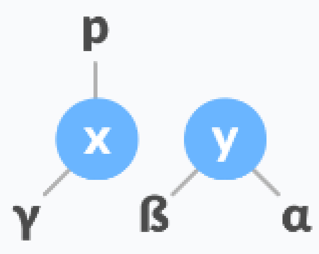
Initial tree

2. If x has a right subtree, assign y as the parent of the right subtree of x.



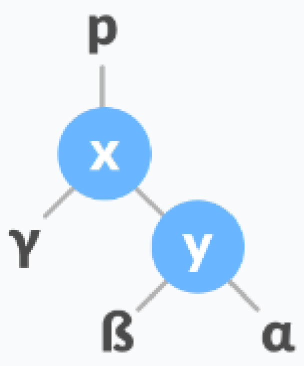
Assign y as the parent of the right subtree of x

3. If the parent of y is `NULL`, make x as the root of the tree.
4. Else if y is the right child of its parent p, make x as the right child of p.
5. Else assign x as the left child of p.



Assign the parent of y as the parent of x.

6. Make x as the parent of y.

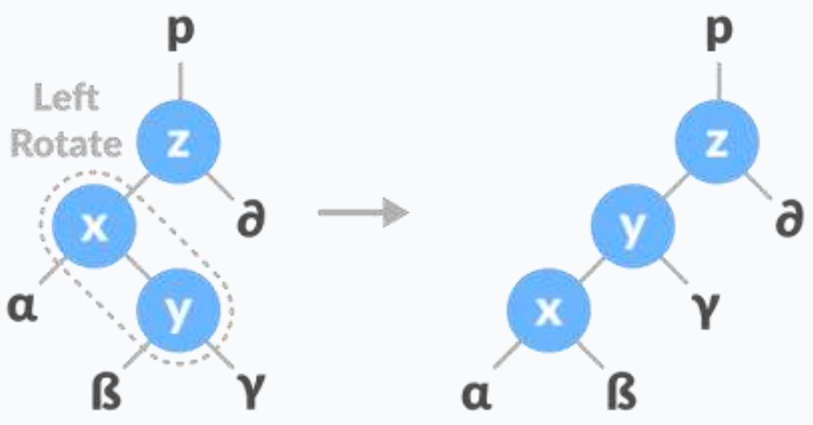


Assign x as the parent of y

Left-Right and Right-Left Rotate

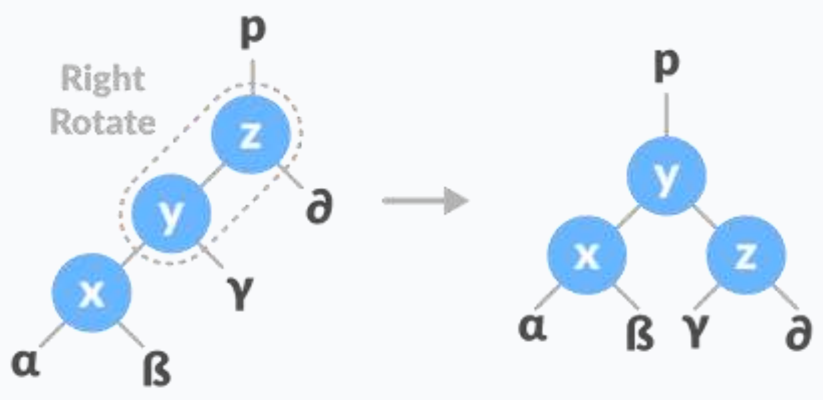
In left-right rotation, the arrangements are first shifted to the left and then to the right.

1. Do left rotation on x-y.



Left rotate x-y

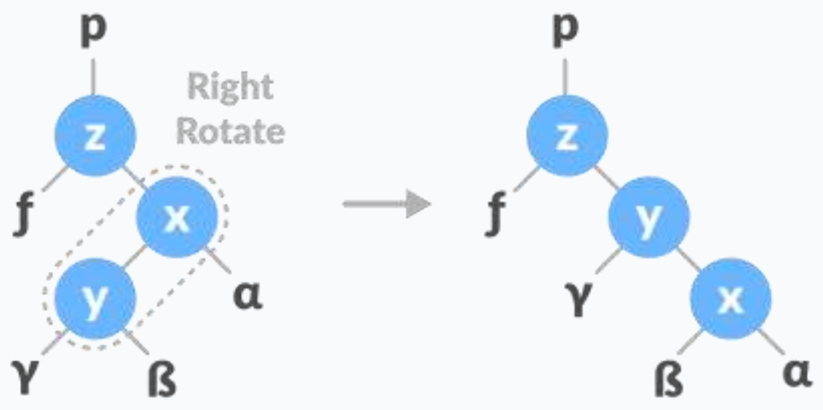
2. Do right rotation on y-z.



Right rotate z-y

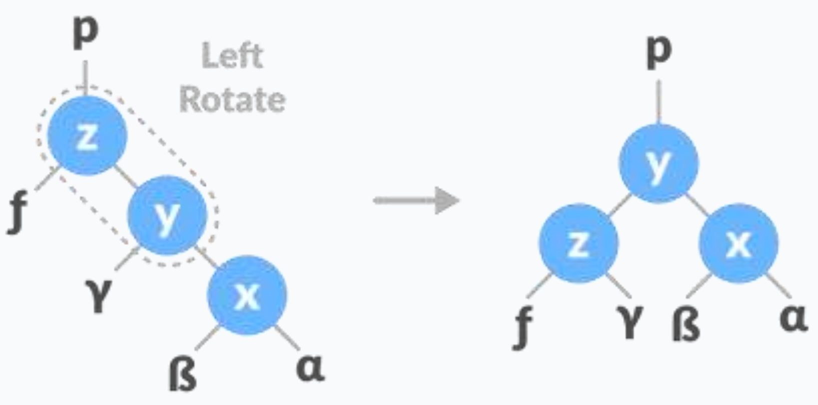
In right-left rotation, the arrangements are first shifted to the right and then to the left.

1. Do right rotation on x-y.



Right rotate x-y

2. Do left rotation on z-y.

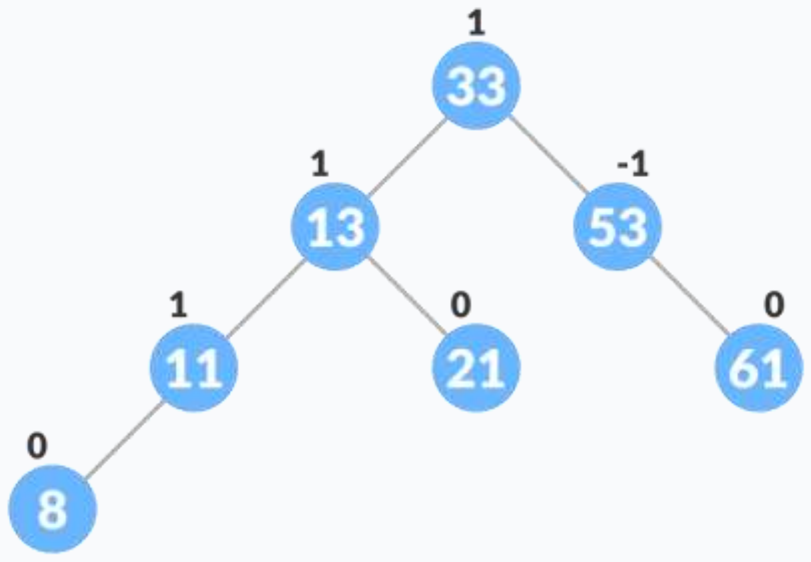


Left rotate z-y

Algorithm to insert a newNode

A `newNode` is always inserted as a leaf node with balance factor equal to 0.

1. Let the initial tree be:



Initial tree for insertion

Let the node to be inserted be:

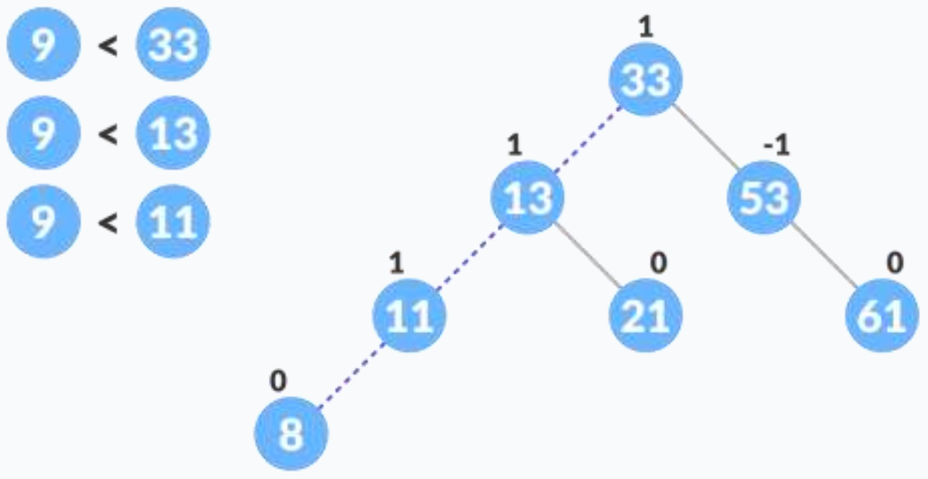
New node

2. Go to the appropriate leaf node to insert a `newNode` using the following recursive steps. Compare `newKey` with `rootKey` of the current tree.

a. If `newKey` < `rootKey`, call insertion algorithm on the left subtree of the current node until the leaf node is reached.

b. Else if `newKey` > `rootKey`, call insertion algorithm on the right subtree of current node until the leaf node is reached.

c. Else, return `leafNode`.

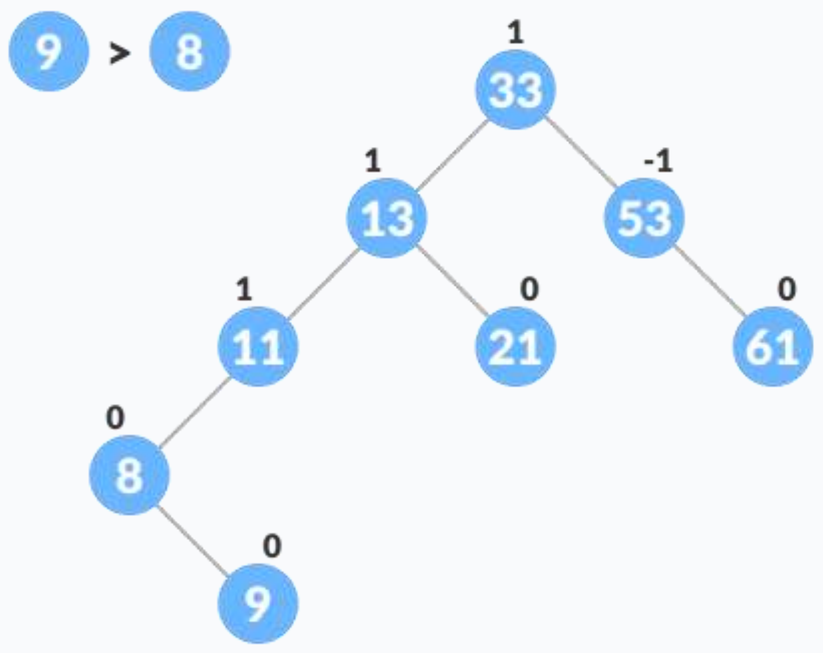


Finding the location to insert newNode

3. Compare `leafKey` obtained from the above steps with `newKey`:

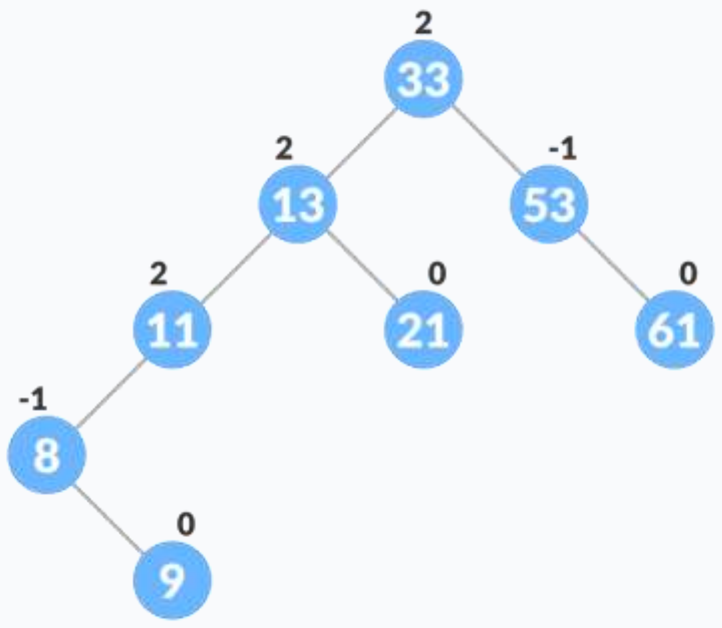
a. If `newKey` < `leafKey`, make newNode as the `leftChild` of `leafNode`.

b. Else, make `newNode` as rightChild of `leafNode`.



Inserting the new node

4. Update `balanceFactor` of the nodes.



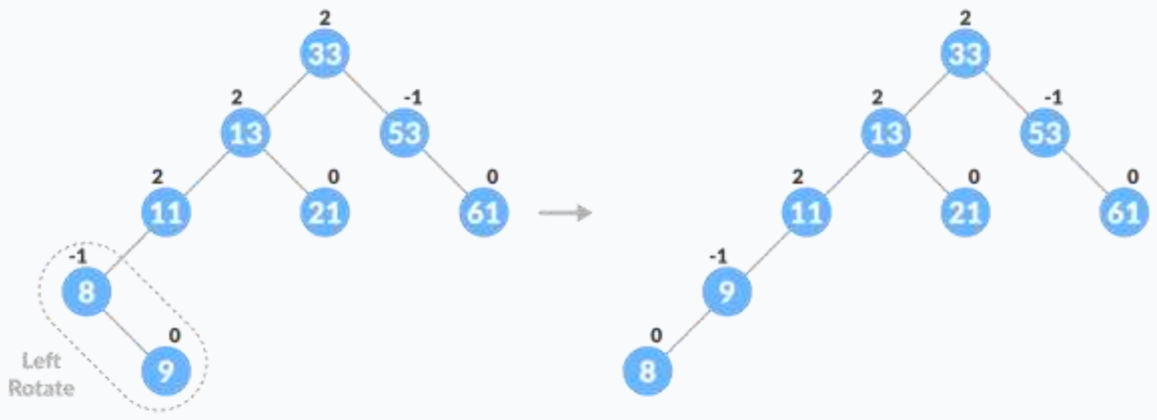
Updating the balance factor after insertion

5. If the nodes are unbalanced, then rebalance the node.

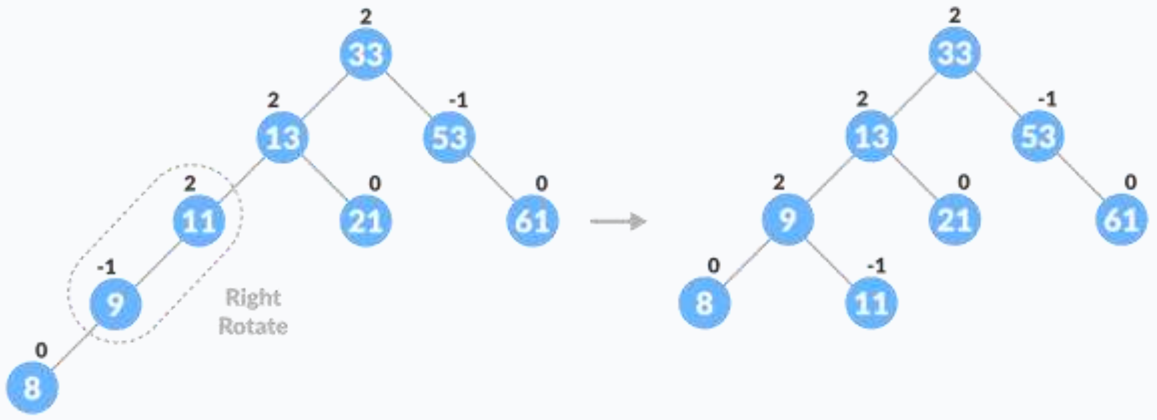
a. If `balanceFactor` > 1, it means the height of the left subtree is greater than that of the right subtree. So, do a right rotation or left-right rotation

a. If `newNodeKey` < `leftChildKey` do right rotation.

b. Else, do left-right rotation.



Balancing the tree with rotation



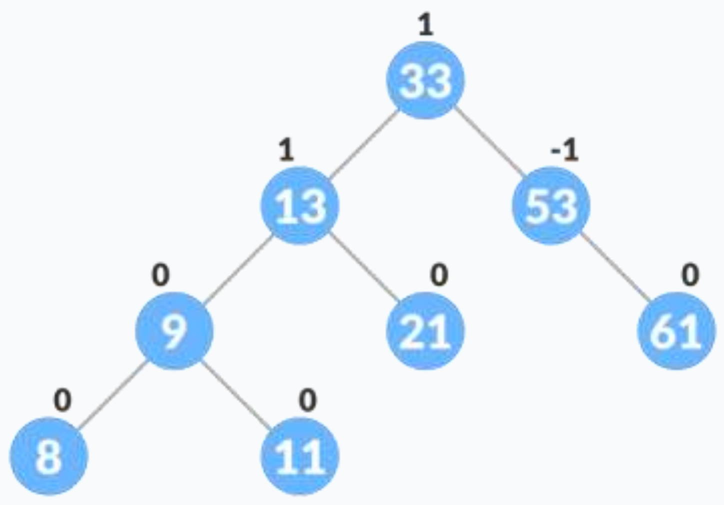
Balancing the tree with rotation

b. If `balanceFactor` < -1, it means the height of the right subtree is greater than that of the left subtree. So, do right rotation or right-left rotation

a. If `newNodeKey` > `rightChildKey` do left rotation.

b. Else, do right-left rotation

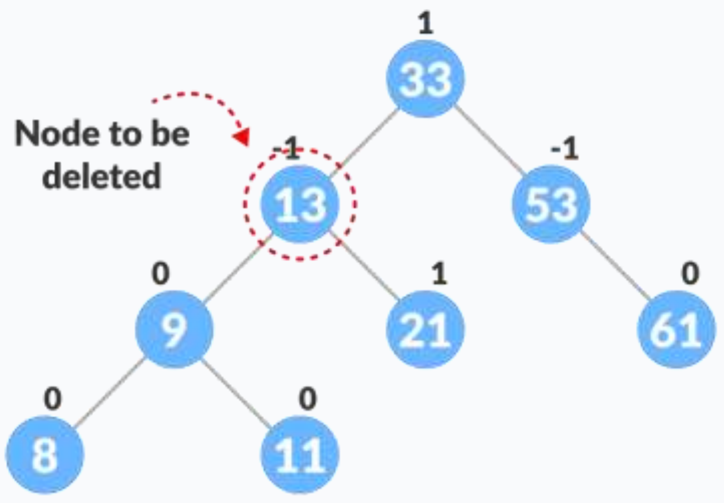
6. The final tree is:


Final balanced tree

Algorithm to Delete a node

A node is always deleted as a leaf node. After deleting a node, the balance factors of the nodes get changed. In order to rebalance the balance factor, suitable rotations are performed.

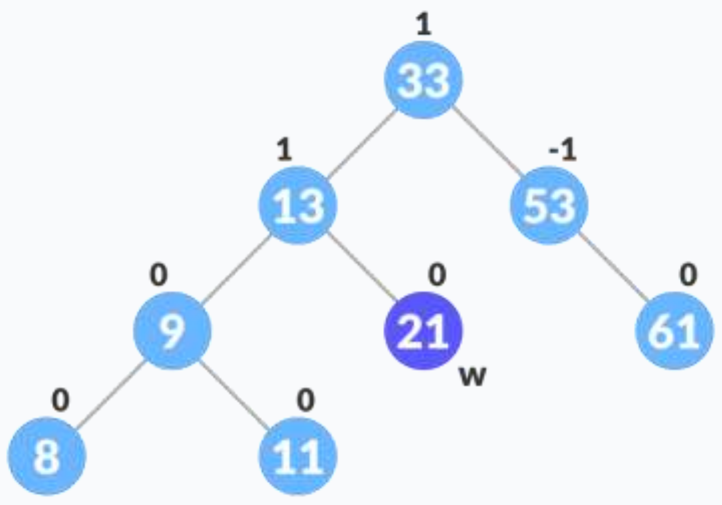
1. Locate `nodeToBeDeleted` (recursion is used to find `nodeToBeDeleted` in the code used below).



Locating the node to be deleted

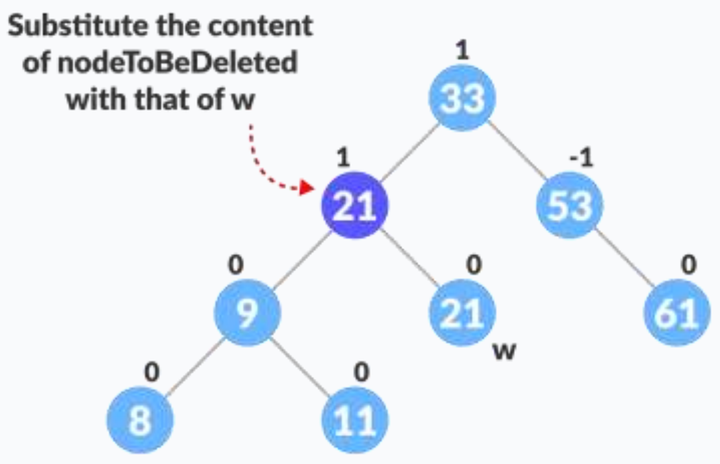
2. There are three cases for deleting a node:

a. If `nodeToBeDeleted` is the leaf node (ie. does not have any child), then remove `nodeToBeDeleted`.
b. If `nodeToBeDeleted` has one child, then substitute the contents of `nodeToBeDeleted` with that of the child. Remove the child.
c. If `nodeToBeDeleted` has two children, find the inorder successor `w` of `nodeToBeDeleted` (ie. node with a minimum value of key in the right subtree).



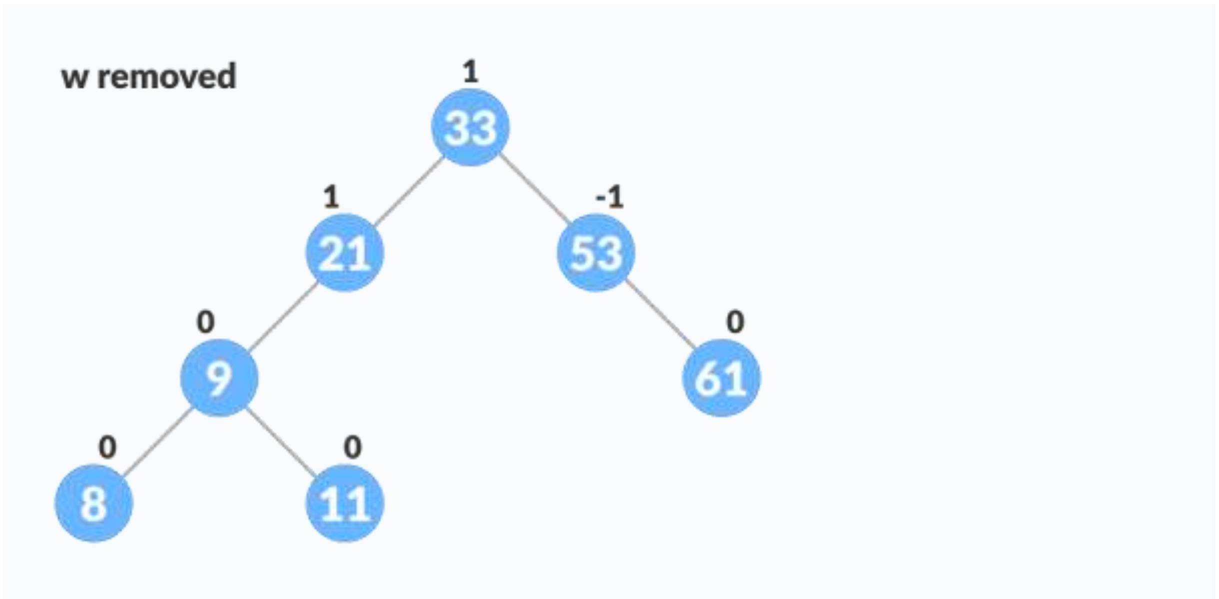
Finding the successor

a. Substitute the contents of `nodeToBeDeleted` with that of `w`.



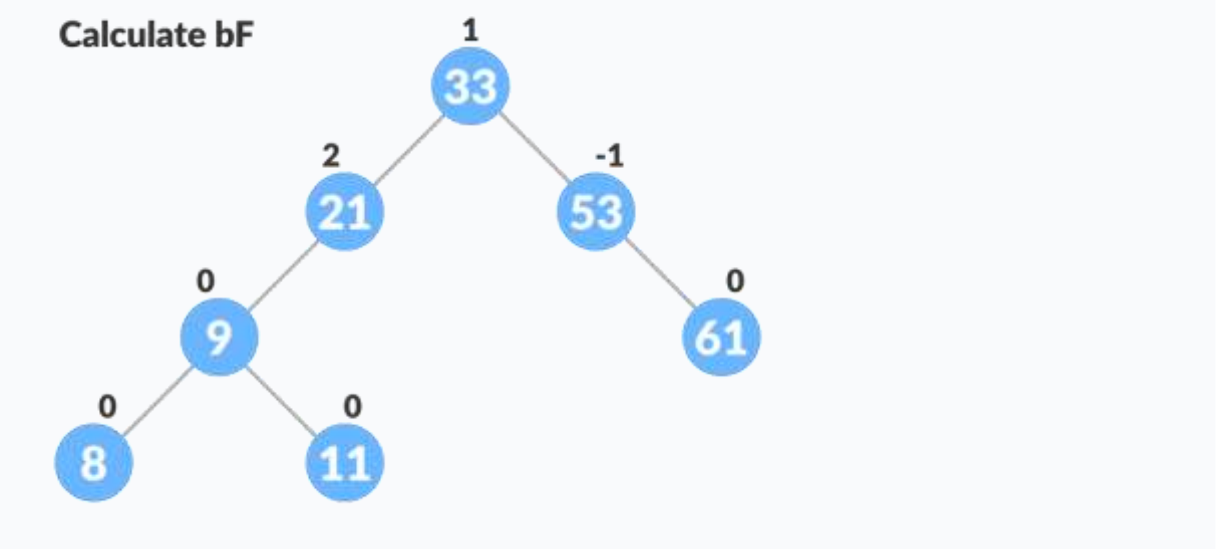
Substitute the node to be deleted

b. Remove the leaf node `w`.



Remove w

3. Update `balanceFactor` of the nodes.

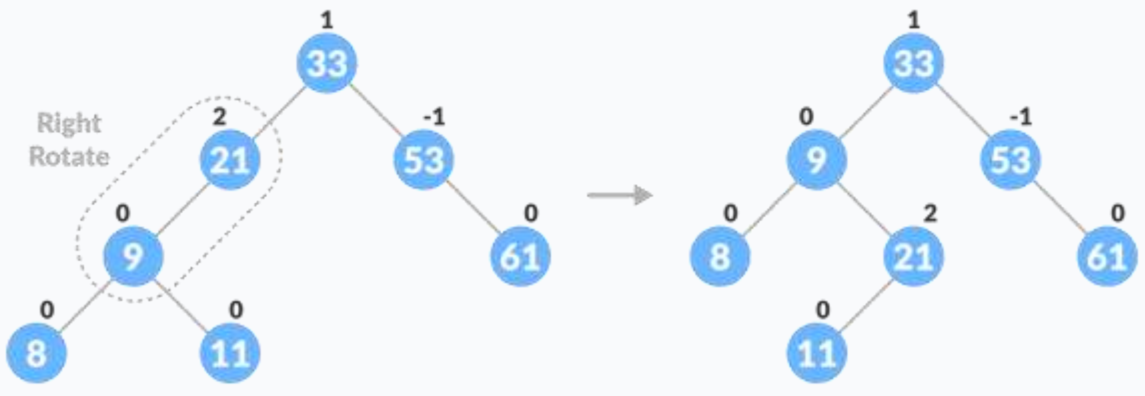


Update bf

4. Rebalance the tree if the balance factor of any of the nodes is not equal to -1, 0 or 1.

a. If `balanceFactor` of `currentNode` > 1,

a. If `balanceFactor` of `leftChild` >= 0, do right rotation.



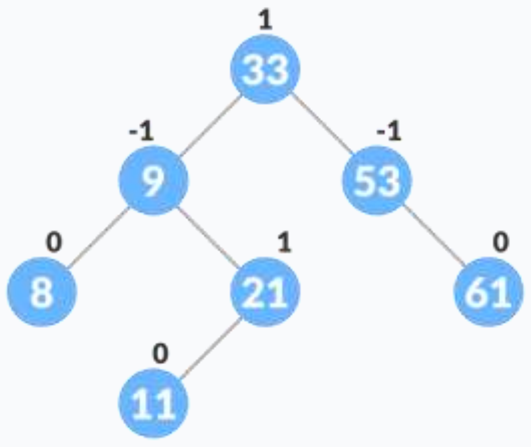
Right-rotate for balancing the tree

b. Else do left-right rotation.

b. If `balanceFactor` of `currentNode` < -1,

a. If `balanceFactor` of `rightChild` <= 0, do left rotation.

b. Else do right-left rotation.



5. The final tree is:

Example:

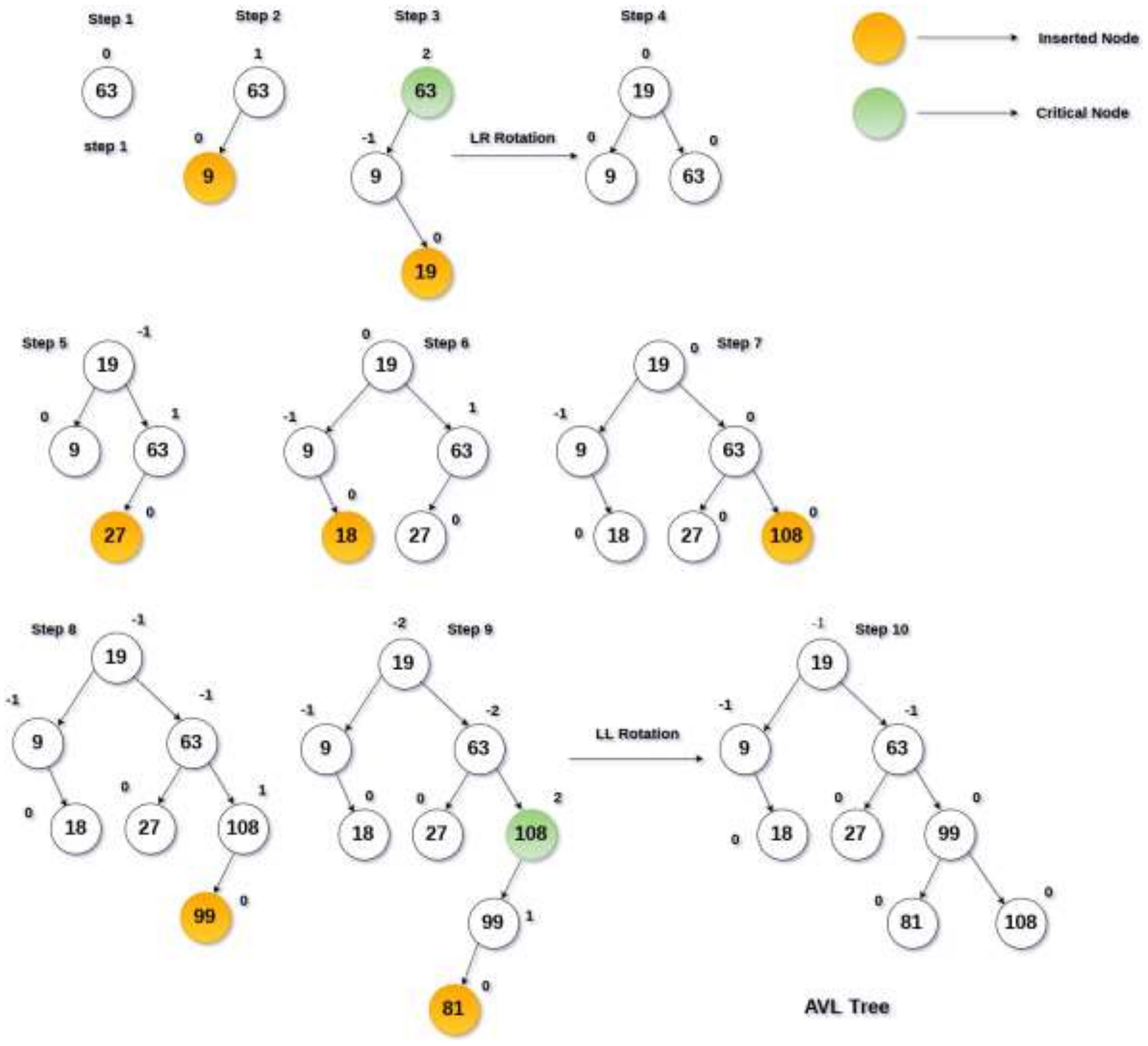
Construct an AVL tree by inserting the following elements in the given order:

63, 9, 19, 27, 18, 108, 99, 81

The process of constructing an AVL tree from the given set of elements is shown in the following figure.

At each step, we must calculate the balance factor for every node, if it is found to be more than 2 or less than -2, then we need a rotation to rebalance the tree. The type of rotation will be estimated by the location of the inserted element with respect to the critical node.

All the elements are inserted in order to maintain the order of binary search tree.



Advantages of AVL Trees

• It is always height balanced

• Height Never Goes Beyond LogN, where N is the number of nodes

• It give better search than compared to binary search tree

• It has self balancing capabilities

Summary of AVL Trees

• These are self-balancing binary search trees.

• Balancing Factor ranges -1, 0, and +1.

• When balancing factor goes beyond the range require rotations to be performed

• Insert, delete, and search time is O(log N).

• AVL tree are mostly used where search is more frequent compared to insert and delete operation.

Multi-way Search Tree: 2-3 Trees, B and B+ tree

Introduction of B-Tree

- B-Tree is a self-balancing search tree.
- In most of the other self-balancing search trees (like AVL and Red-Black Trees), it is assumed that everything is in main memory.
- To understand the use of B-Trees, we must think of the huge amount of data that cannot fit in main memory.
- When the number of keys is high, the data is read from disk in the form of blocks.
- Disk access time is very high compared to the main memory access time.
- The main idea of using B-Trees is to reduce the number of disk accesses.
- Most of the tree operations (search, insert, delete, max, min, ..etc) require disk accesses same as the height of the tree.
- B-tree is a fat tree.
- The height of B-Trees is kept low by putting maximum possible keys in a B-Tree node.
- Generally, the B-Tree node size is kept equal to the disk block size.
- Since the height of the B-tree is low so total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees like AVL Tree, Red-Black Tree, ..etc.

Properties of B-Tree:

1. All leaves are at the same level.
2. A B-Tree is defined by the term *order* ‘t’. The value of t depends upon disk block size.
3. Every node except root must contain at least $\left\lceil \frac{t}{2} \right\rceil$-1 keys. The root may contain minimum 1 key.
4. All nodes (including root) may contain at most t – 1 keys.
5. Number of children of a node is equal to the number of keys in it plus 1.
6. All keys of a node are sorted in increasing order. The child between two keys k1 and k2 contains all keys in the range from k1 and k2.

7. B-Tree grows and shrinks from the root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.
8. Insertion of a Node in B-Tree happens only at Leaf Node.

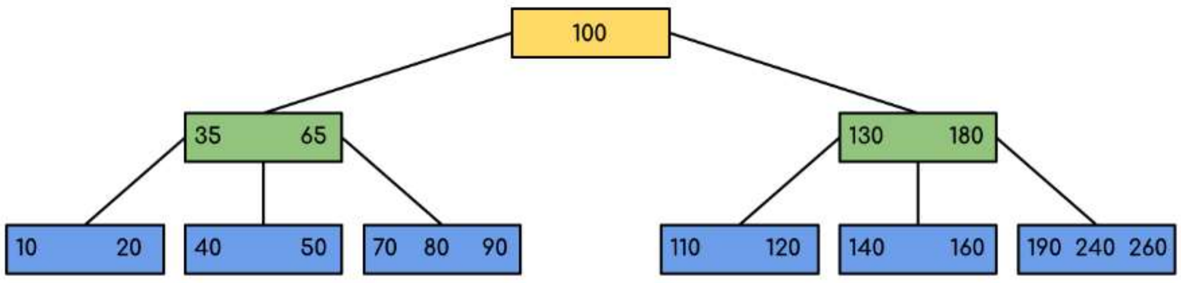
Traversal in B-Tree:

Traversal is also similar to Inorder traversal of Binary Tree. We start from the leftmost child, recursively print the leftmost child, then repeat the same process for remaining children and keys. In the end, recursively print the rightmost child.

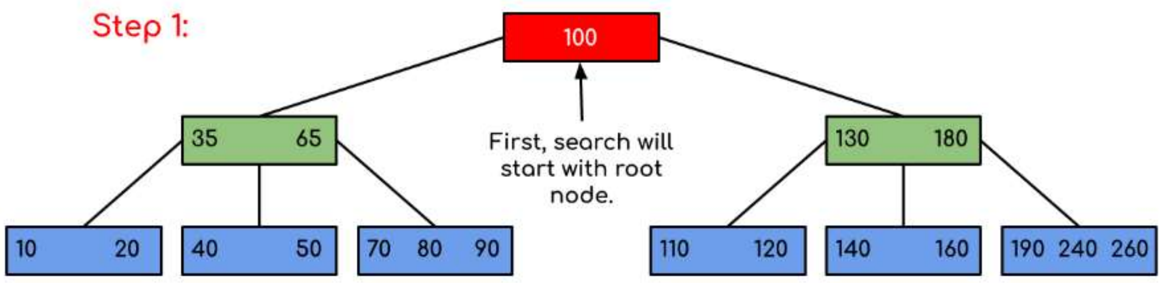
Search Operation in B-Tree:

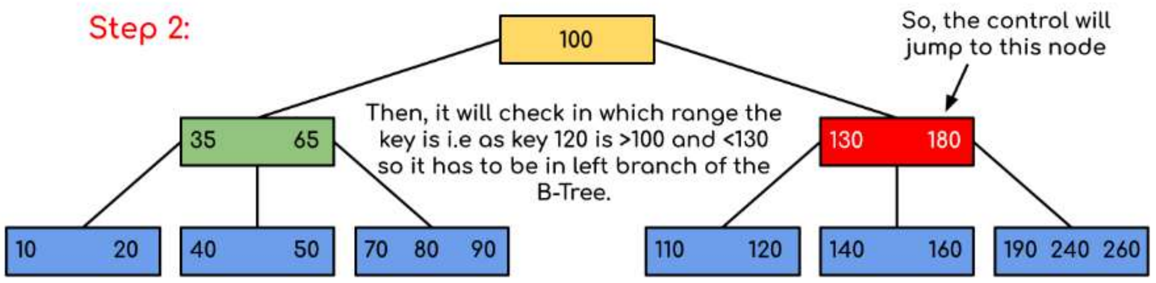
Search is similar to the search in Binary Search Tree. Let the key to be searched be k. We start from the root and recursively traverse down. For every visited non-leaf node, if the node has the key, we simply return the node. Otherwise, we recur down to the appropriate child (The child which is just before the first greater key) of the node. If we reach a leaf node and don't find k in the leaf node, we return NULL.

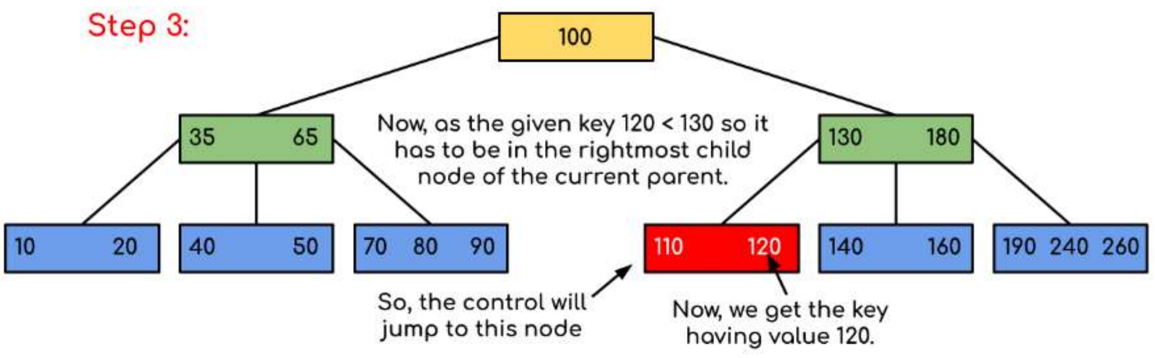
Example: Searching 120 in the given B-Tree.



Solution:




Step 2:
100
So, the control will jump to this node
Then, it will check in which range the key is i.e as key 120 is >100 and <130 so it has to be in left branch of the B-Tree.
35 65
130 180
10 20
40 50
70 80 90
110 120
140 160
190 240 260


Step 3:
100
35 65
130 180
Now, as the given key 120 < 130 so it has to be in the rightmost child node of the current parent.
10 20
40 50
70 80 90
110 120
140 160
190 240 260
So, the control will jump to this node
Now, we get the key having value 120.

In this example, we can see that our search was reduced by just limiting the chances where the key containing the value could be present. Similarly if within the above example we've to look for 180, then the control will stop at step 2 because the program will find that the key 180 is present within the current node. And similarly, if it's to seek out 90 then as 90 < 100 so it'll go to the left subtree automatically and therefore the control flow will go similarly as shown within the above example.

Insertion into a B-tree

- Inserting an element on a B-tree consists of two events: **searching the appropriate node** to insert the element and **splitting the node** if required.
- Insertion operation always takes place in the bottom-up approach.

Let us understand these events below.

Insertion Operation

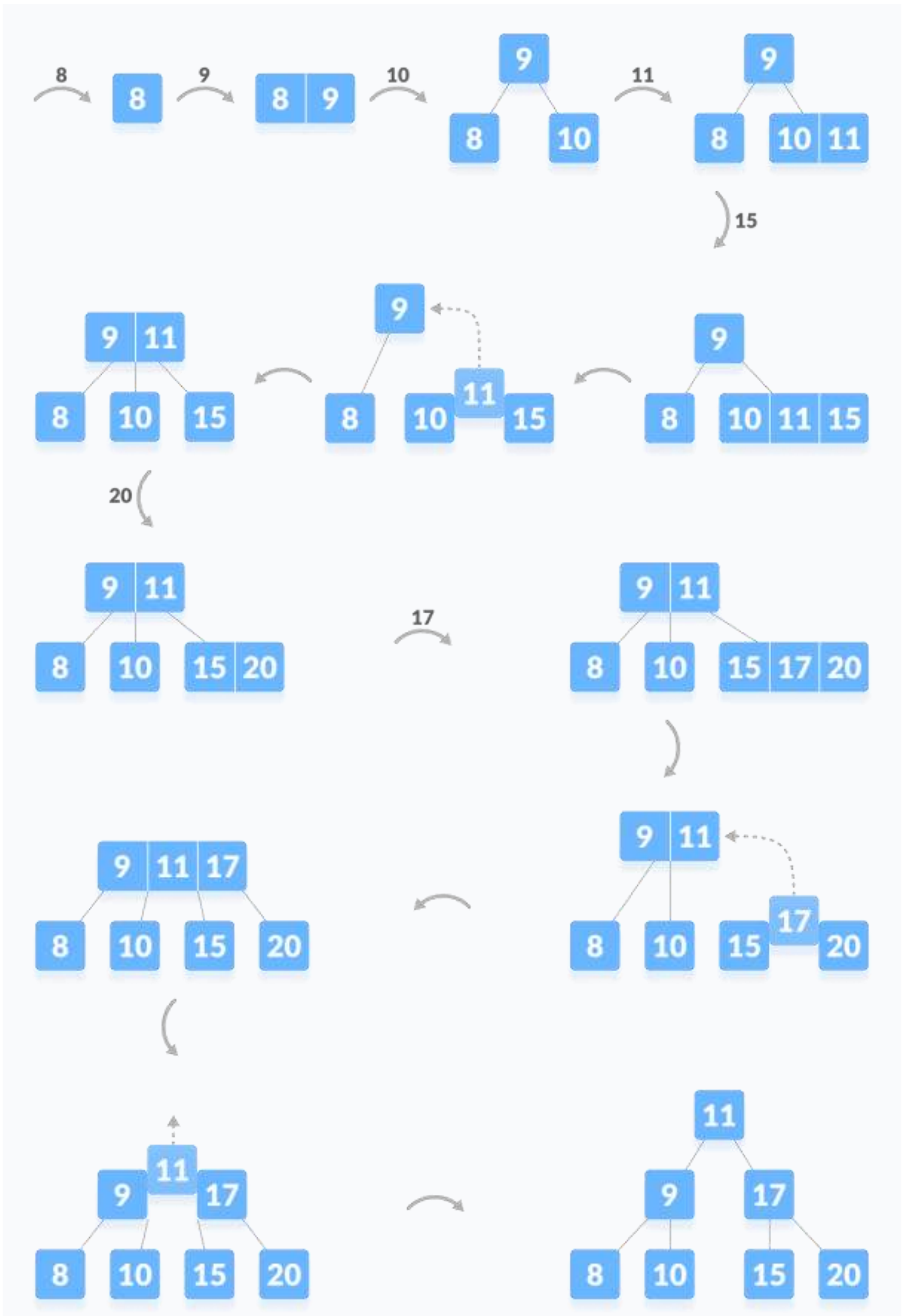
1. If the tree is empty, allocate a root node and insert the key.
2. Update the allowed number of keys in the node.
3. Search the appropriate node for insertion.
4. If the node is full, follow the steps below.
5. Insert the elements in increasing order.
6. Now, there are elements greater than its limit. So, split at the median.

7. Push the median key upwards and make the left keys as a left child and the right keys as a right child.
8. If the node is not full, follow the steps below.
9. Insert the node in increasing order.

Insertion Example

Let us understand the insertion operation with the illustrations below.

The elements to be inserted are 8, 9, 10, 11, 15, 20, 17.



Inserting elements into a B-tree

1. **Inorder Predecessor**

   The largest key on the left child of a node is called its inorder predecessor.

2. **Inorder Successor**

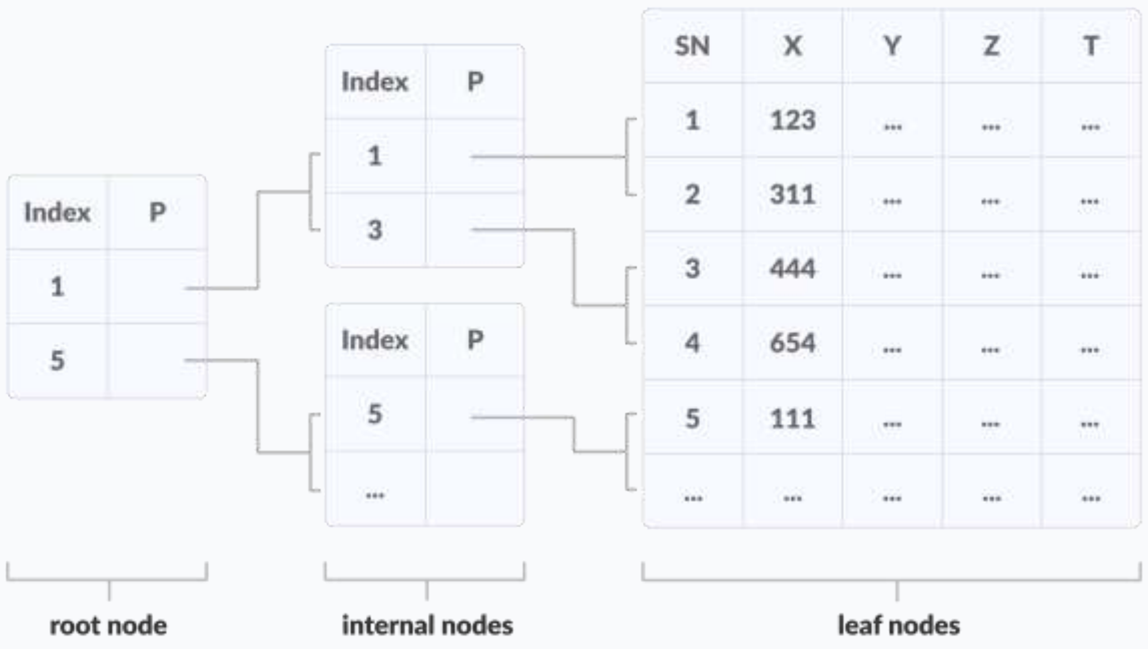
   The smallest key on the right child of a node is called its inorder successor.

Applications of B-Trees:

- It is used in large databases to access data stored in the disk
- Searching of data in a data set can be achieved in significantly less time using B tree
- With the indexing feature multilevel indexing can be achieved.
- Most of the servers also use B-tree approach.

B+ Tree

- A B+ tree is an advanced form of a self-balancing tree in which all the values are present in the leaf level.
- An important concept to be understood before learning B+ tree is multilevel indexing.
- In multilevel indexing, the index of indices is created as in figure below. It makes accessing the data easier and faster.

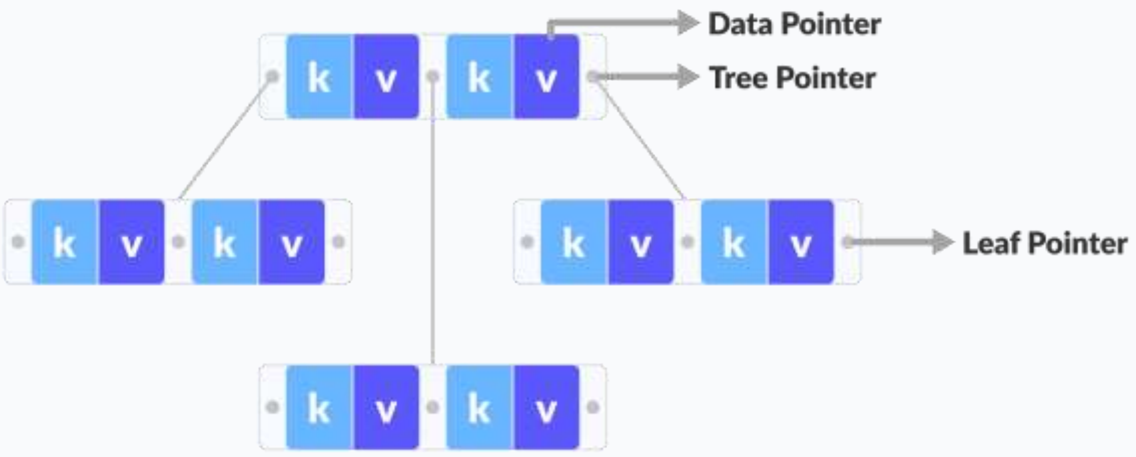


Multilevel Indexing using B+ tree

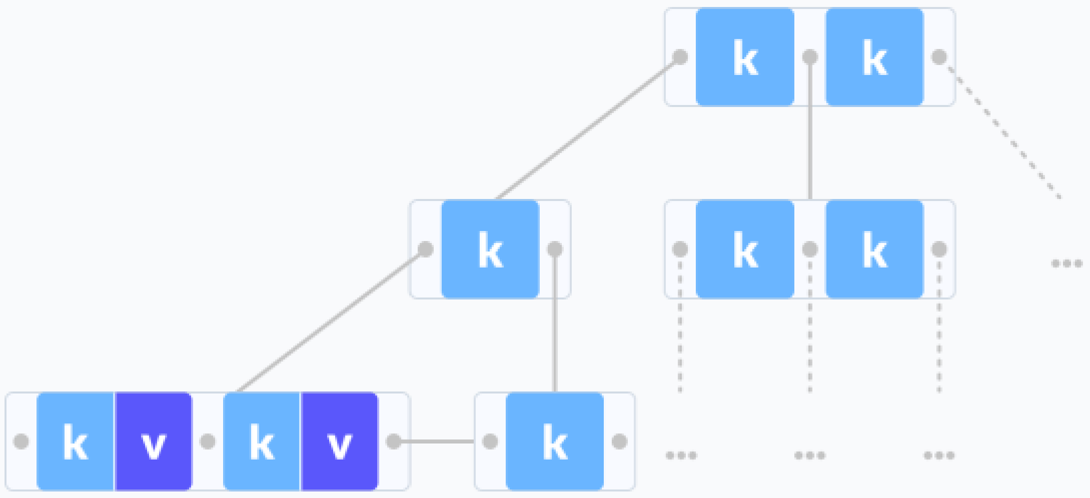
Properties of a B+ Tree

1. All leaves are at the same level.
2. The root has at least two children.
3. Each node except root can have a maximum of m children and at least m/2 children.
4. Each node can contain a maximum of m - 1 keys and a minimum of ⌈m/2⌉ - 1 keys.

Comparison between a B-tree and a B+ Tree



B-tree



B+ tree

- The data pointers are present only at the leaf nodes on a B+ tree whereas the data pointers are present in the internal, leaf or root nodes on a B-tree.
- The leaves are not connected with each other on a B-tree whereas they are connected on a B+ tree.
- Operations on a B+ tree are faster than on a B-tree.

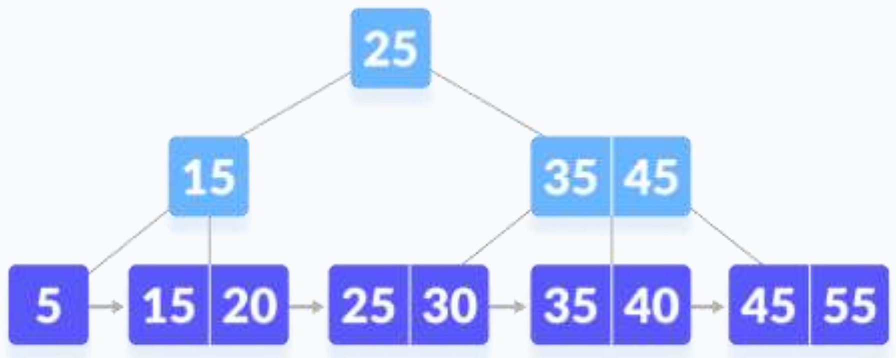
Searching on a B+ Tree

The following steps are followed to search for data in a B+ Tree of order m. Let the data to be searched be k.

1. Start from the root node. Compare k with the keys at the root node [k1, k2, k3,......km - 1].
2. If k < k1, go to the left child of the root node.
3. Else if k == k1, compare k2. If k < k2, k lies between k1 and k2. So, search in the left child of k2.
4. If k > k2, go for k3, k4,...km-1 as in steps 2 and 3.
5. Repeat the above steps until a leaf node is reached.
6. If k exists in the leaf node, return true else return false.

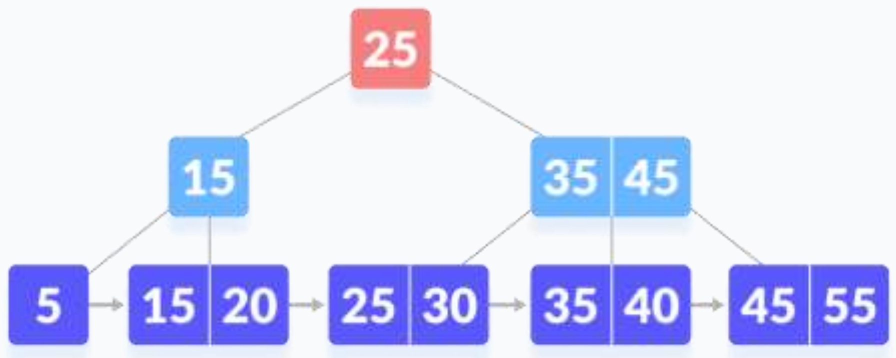
Searching Example on a B+ Tree

Let us search k = 45 on the following B+ tree.



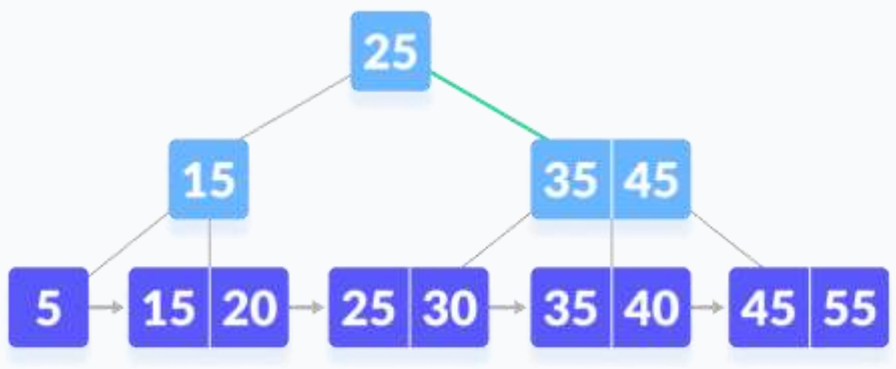
B+ tree

1. Compare k with the root node.



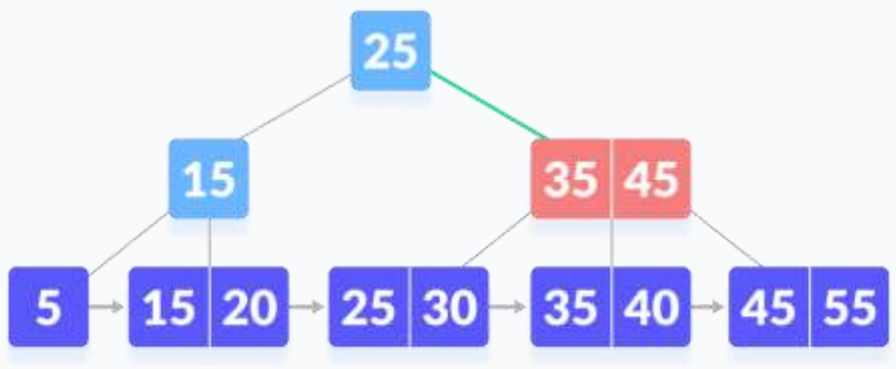
k is not found at the root

2. Since k > 25, go to the right child.



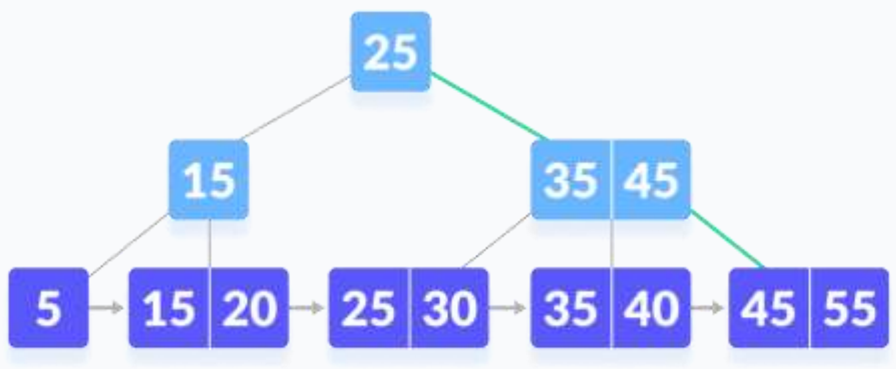
Go to right of the root

3. Compare k with 35. Since k > 30, compare k with 45.



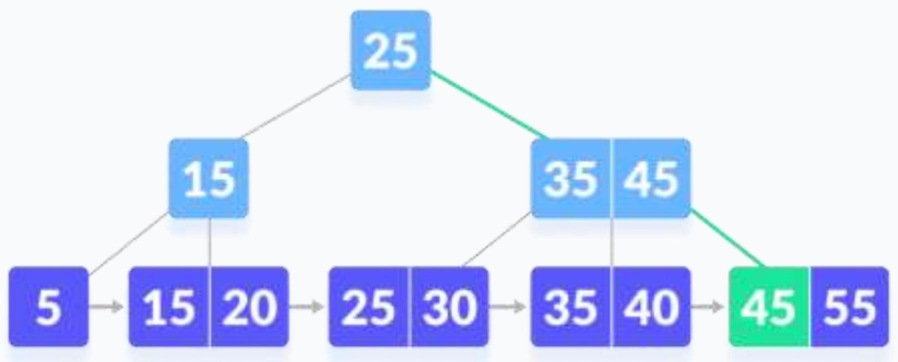
k not found

4. Since k ≥ 45, so go to the right child.



go to the right

5. k is found.



k is found

Insertion on a B+ Tree

- Inserting an element into a B+ tree consists of three main events: **searching the appropriate leaf**, **inserting** the element and **balancing/splitting** the tree.

Let us understand these events below.

Insertion Operation

Before inserting an element into a B+ tree, these properties must be kept in mind.

- The root has at least two children.
- Each node except root can have a maximum of m children and at least m/2 children.
- Each node can contain a maximum of m - 1 keys and a minimum of ⌈m/2⌉ - 1 keys.

The following steps are followed for inserting an element.

1. Since every element is inserted into the leaf node, go to the appropriate leaf node.
2. Insert the key into the leaf node.

Case I

1. If the leaf is not full, insert the key into the leaf node in increasing order.

Case II

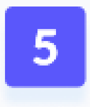
1. If the leaf is full, insert the key into the leaf node in increasing order and balance the tree in the following way.
2. Break the node at m/2th position.
3. Add m/2th key to the parent node as well.
4. If the parent node is already full, follow steps 2 to 3.

Insertion Example

Let us understand the insertion operation with the illustrations below.

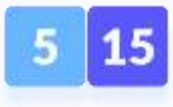
The elements to be inserted are 5,15, 25, 35, 45.

1. Insert 5.



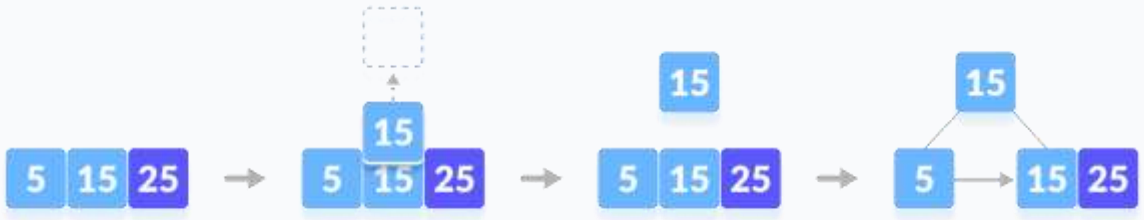
Insert 5

2. Insert 15.



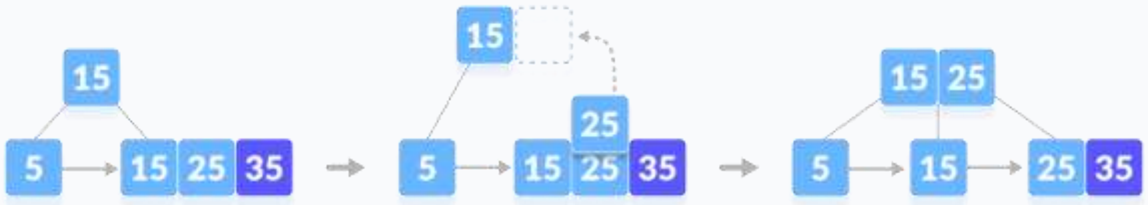
Insert 15

3. Insert 25.



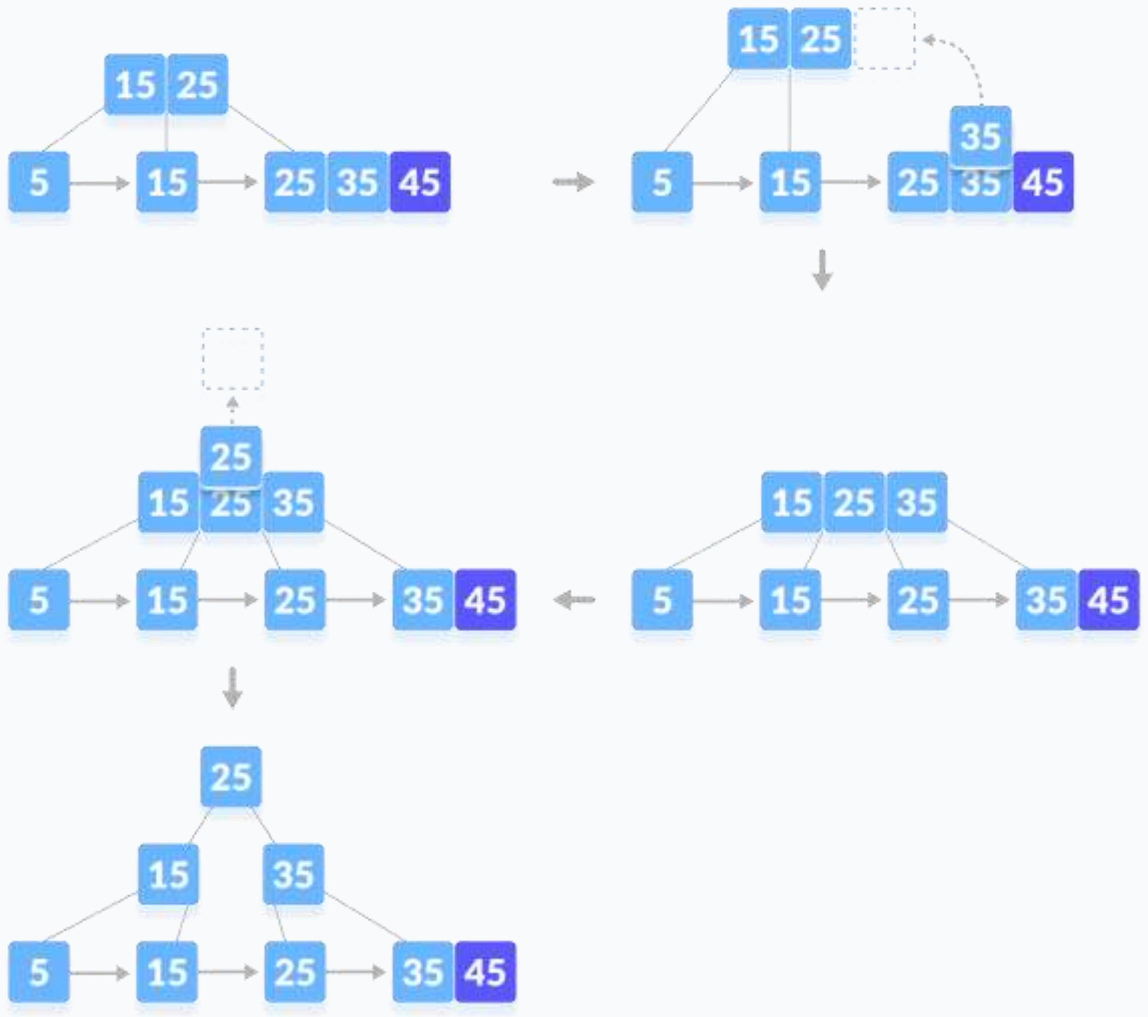
Insert 25

4. Insert 35.



Insert 35

5. Insert 45.



Insert 45