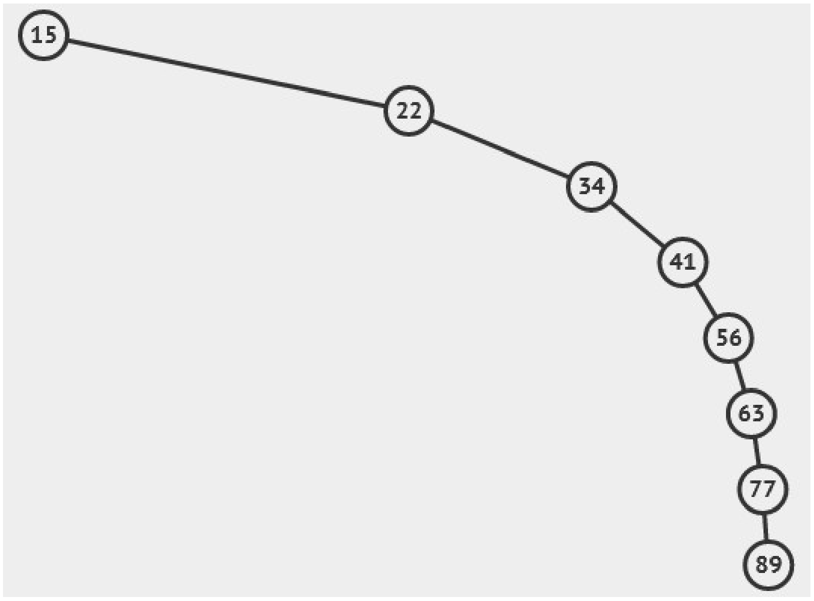
AVL Tree (Height balanced BST)

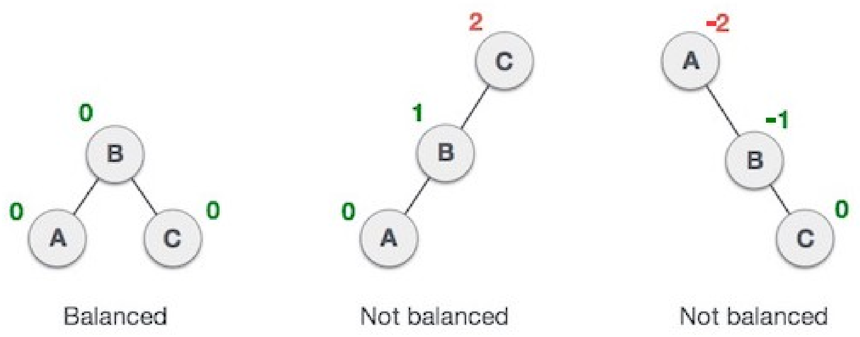
Problem in BST: What if the input to binary search tree comes in a sorted (ascending or descending) manner? It will then look like this



It is observed that BST's worst-case performance is closest to linear search algorithms, that is O(n). In real-time data, we cannot predict data pattern and their frequencies. So, a need arises to balance out the existing BST.

Named after their inventor **Adelson-Velski** & **Landis**, **AVL tree** is height balancing binary search tree. AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1 and not less than -1. This difference is called the **Balance Factor**.

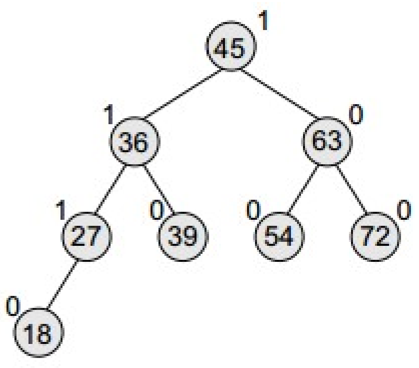
In the below figure, the first tree is balanced and the next two trees are not balanced −



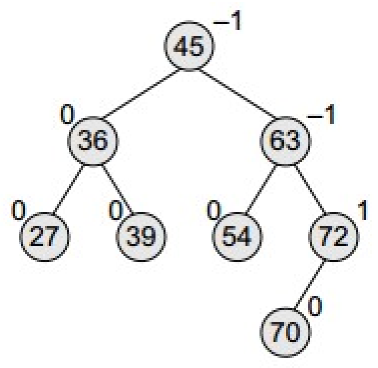
In the second tree, the left subtree of **C** has height 2 and the right subtree has height 0, so the difference is 2. In the third tree, the right subtree of **A** has height 2 and the left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1, 0 or -1.

```
BalanceFactor = height (left-sutree) − height (right-sutree)
```

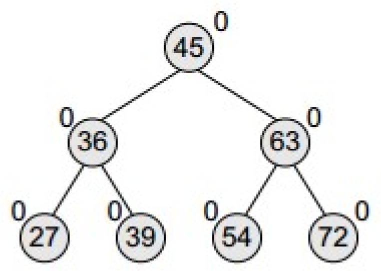
- **Left-Heavy Tree:** If the balance factor of a node is 1, then it means that the left sub-tree of the tree is one level higher than that of the right sub-tree. Such a tree is therefore called as a left-heavy tree.
- **Right-Heavy Tree:** If the balance factor of a node is –1, then it means that the 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.
- **Balanced 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.



Left-Heavy AVL Tree



Right-Heavy AVL Tree



Balanced AVL Tree

- The trees given in above figure are typical candidates of AVL trees because the balancing factor of every node is 1, 0, or –1.
- However, insertions and deletions from an AVL tree may disturb the balance factor of the nodes and, thus, rebalancing of the tree may have to be done.
- The tree is rebalanced by performing rotation at the critical node (a node where balance factor is other than -1, 0 and 1).

AVL Rotations

To balance itself, an AVL tree may perform the following four kinds of rotations

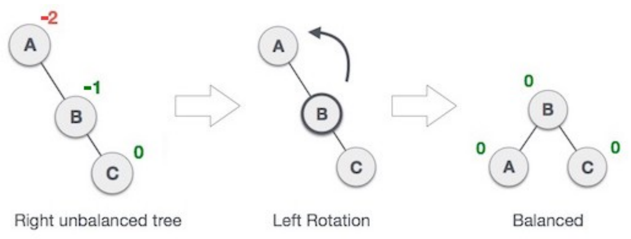
1. **Left rotation (Right-Right Case)**
2. **Right rotation (Left-Left Case)**
3. **Left-Right rotation (Left-Right Case)**
4. **Right-Left rotation (Right-Left Case)**

The first two rotations are single rotations and the next two rotations are double rotations.

Left Rotation

If a tree becomes imbalanced, when a node is inserted into the right subtree of the right subtree, then a single left rotation is performed to balance it.

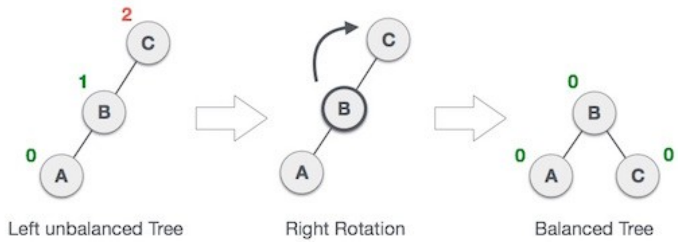
In this example, node **A** has become imbalanced as a node **C** is inserted in the right subtree of **A**'s right subtree. The left rotation is performed by making **A** the left-subtree of **B**.



1. Right Rotation

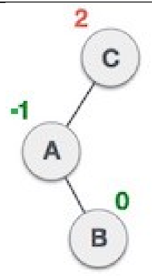
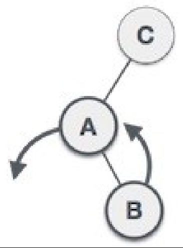
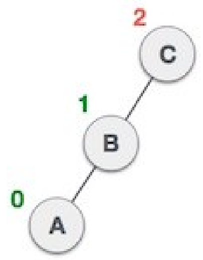
AVL tree may become imbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.

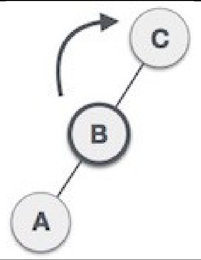
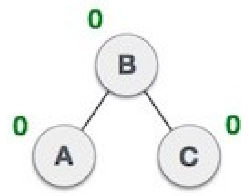
As depicted, the imbalanced node becomes the right child of its left child by performing a right rotation.



2. Left – Right Rotation

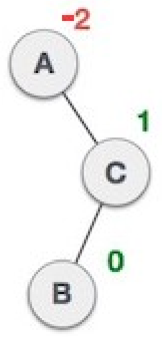
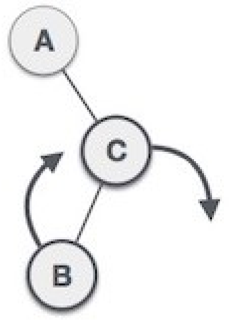
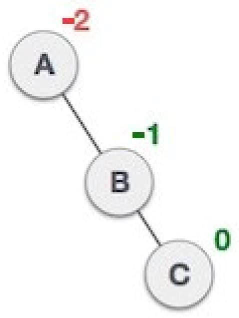
A left-right rotation is a combination of left rotation followed by right rotation.

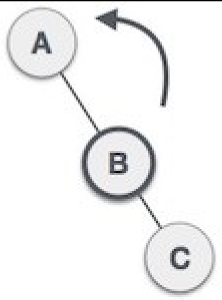
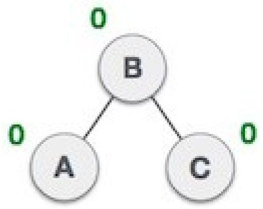
State	Action
	A node has been inserted into the right subtree of the left subtree. This makes **C** an imbalanced node. These scenarios cause AVL tree to perform left-right rotation.
	First perform the left rotation on the left subtree of **C**. This makes **A**, the left subtree of **B**.
	Node **C** is still imbalanced; however now, it is because of the left-subtree of the left-subtree.

	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.

3. Right – Left Rotation

A right - left rotation 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 imbalanced node with balance factor -2.
	First, 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 imbalanced 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.

Operations on an AVL Tree

The following operations are performed on an AVL tree...

1. **Search**
2. **Insertion**
3. **Deletion**

Search Operation in AVL Tree

In an AVL tree, the search operation is performed with **O(log n)** time complexity. The search operation is performed similar to Binary search tree search operation. The following steps are used to search an element in AVL tree...

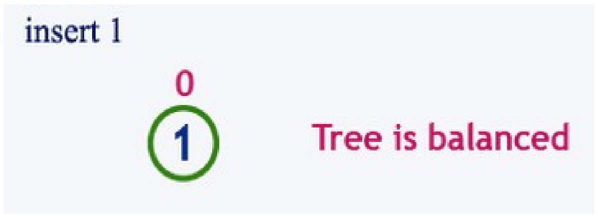
1. Read the search element from the user
2. Compare, the search element with the value of root node in the tree.
3. If both are matching, then display "Given node found!!!" and terminate the function
4. If both are not matching, then check whether search element is smaller or larger than that node value.
5. If search element is smaller, then continue the search process in left subtree.
6. If search element is larger, then continue the search process in right subtree.
7. Repeat the same until we found exact element or we completed with a leaf node
8. If we reach to the node with search value, then display "Element is found" and terminate the function.
9. If we reach to a leaf node and it is also not matching, then display "Element not found" and terminate the function.

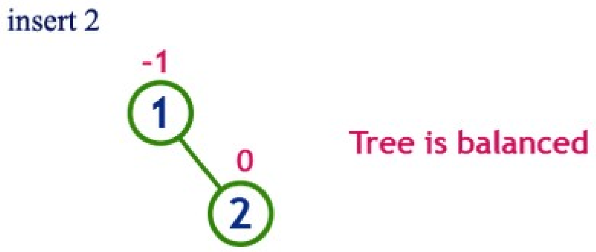
Insertion Operation in AVL Tree

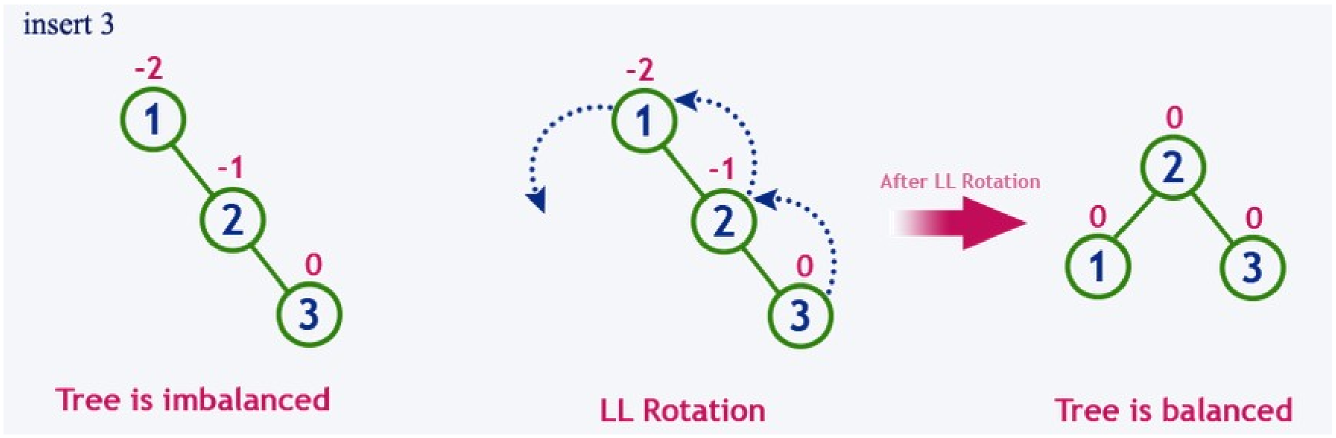
In an AVL tree, the insertion operation is performed with **O(log n)** time complexity. In AVL Tree, new node is always inserted as a leaf node. The insertion operation is performed as follows...

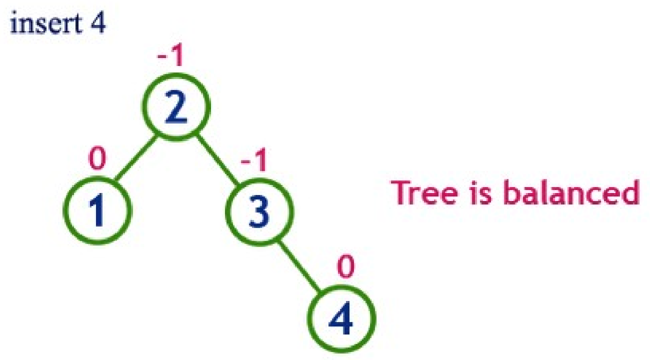
1. Insert the new element into the tree using Binary Search Tree insertion logic.
2. After insertion, check the **Balance Factor** of nodes present in path from newly inserted node to the root.
3. If the **Balance Factor** of every node is **0 or 1 or -1** then go for next operation.
4. If the **Balance Factor** of any node in path from newly inserted node to root is other than **0 or 1 or -1** then perform the suitable **Rotation** to make it balanced. Once rotation is completed, it is guaranteed that tree will be balanced.

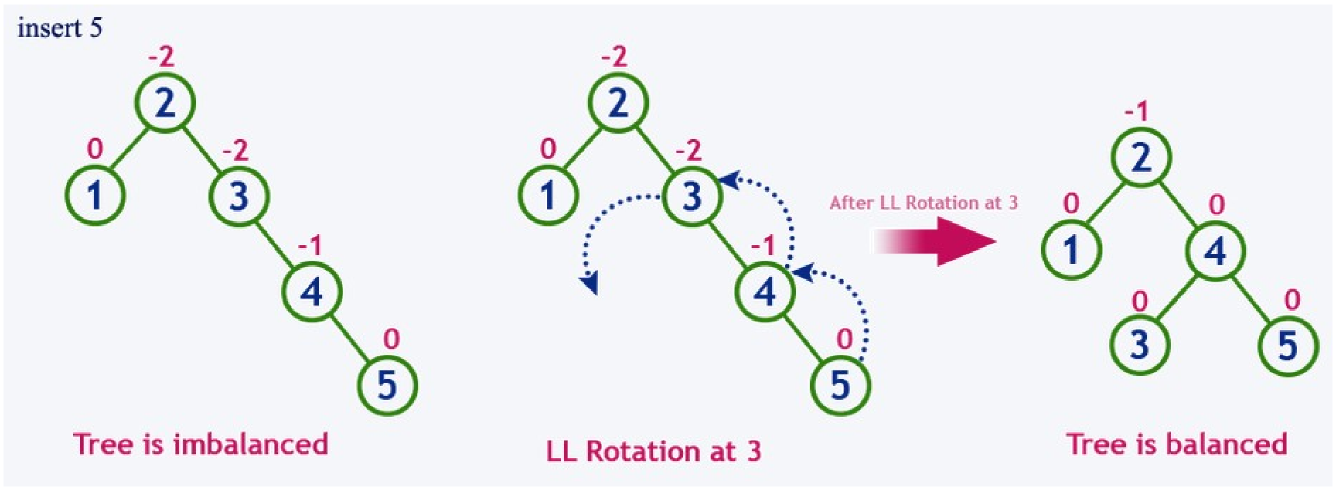
Example 1: Construct an AVL Tree by inserting numbers from 1 to 8.

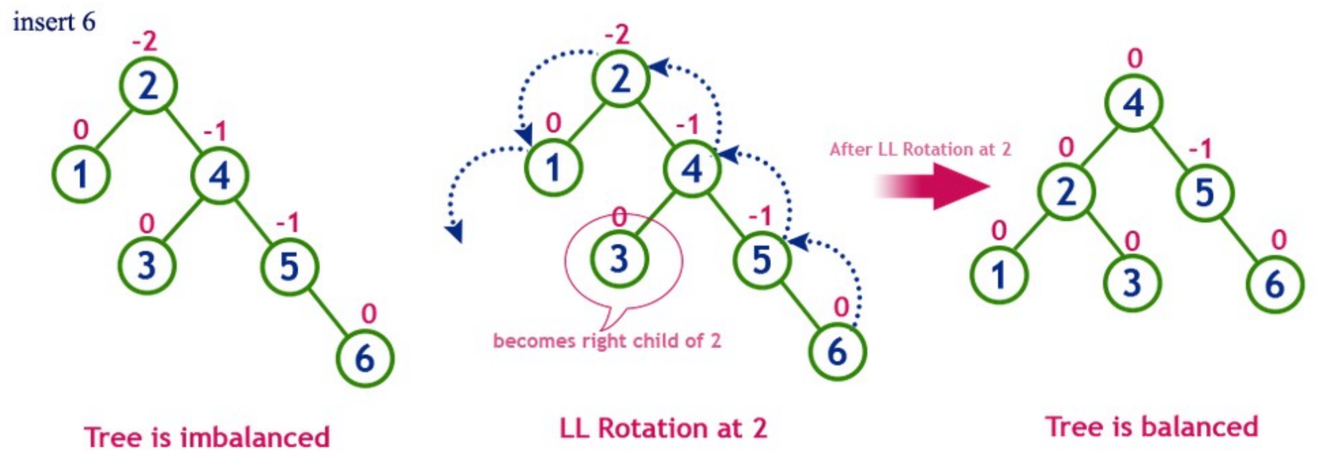


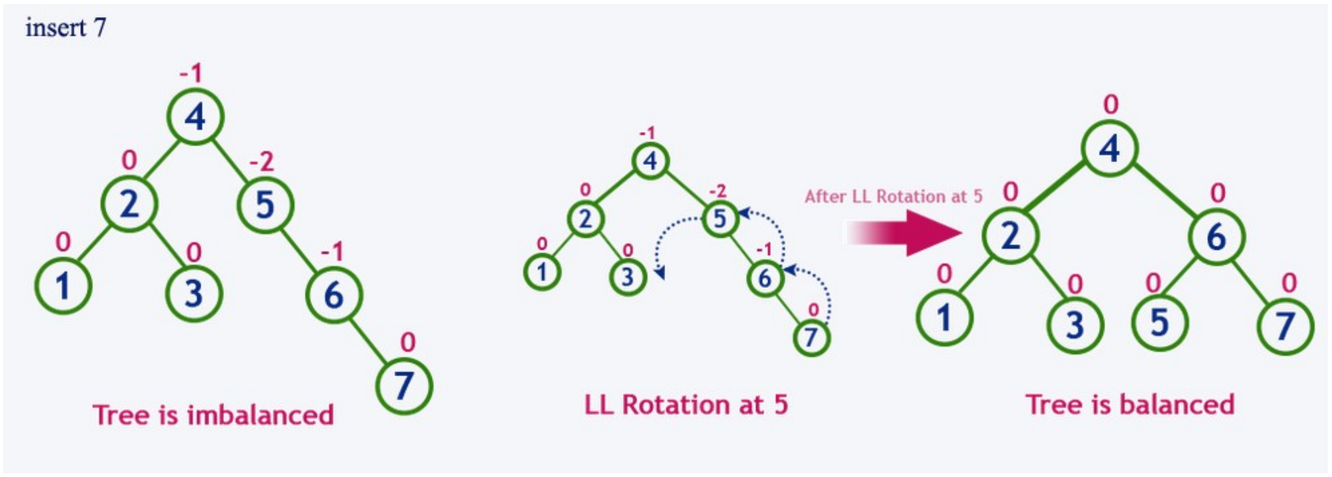


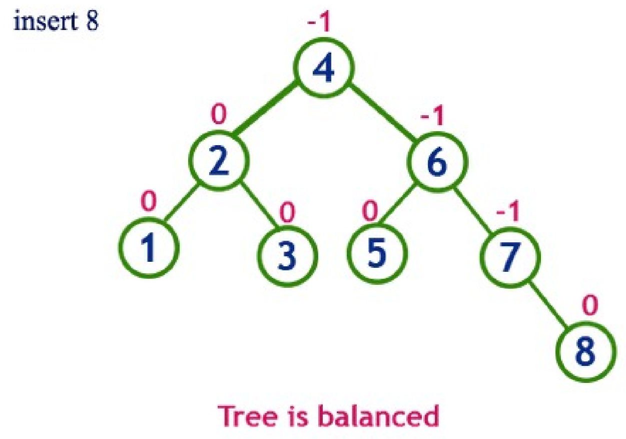












Example 2: Construct an AVL Tree by inserting numbers 14, 17, 11, 7, 53, 4, 13 into an empty AVL tree

Example 3: Construct an AVL Tree by inserting numbers 15, 20, 24, 10, 13, 7, 30, 36, 25 into an empty AVL tree

Deletion Operation in AVL Tree

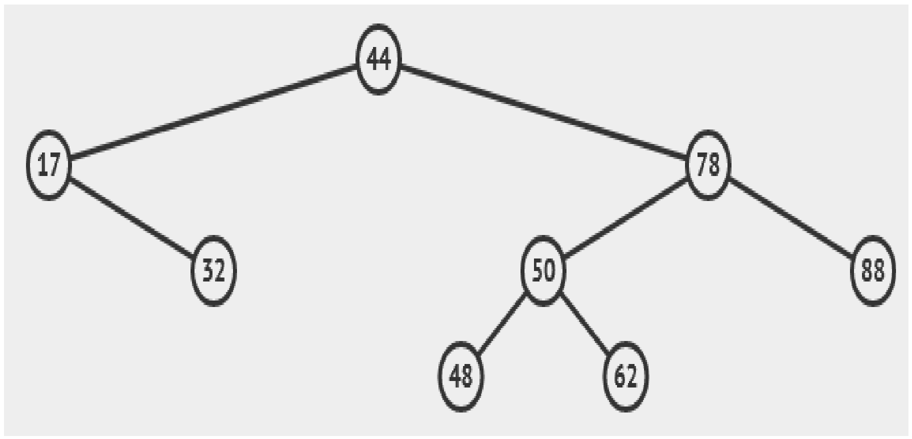
- Deletion of a node in an AVL tree is similar to that of binary search trees. But it goes one step ahead.
- Deletion may disturb the balance of the tree, so to rebalance the AVL tree; rotation(s) needs to be performed.
- On deletion of node X from the AVL tree, if node A becomes the critical node (closest ancestor node - on the path from X to the root node - that does not have its balance factor as 1, 0, or –1), then the type of rotation depends on whether X is in the left sub-tree of A or in its right sub-tree.
- Let A be the node where balance must be restored.
  - If the deleted node was in A's right subtree, then let B be the root of A's left subtree. Then:
    1. **B has balance factor 0 or +1 after deletion** -- then perform a single right rotation
    2. **B has balance factor -1 after deletion** -- then perform a left-right rotation
  - If the deleted node was in A's left subtree, then let B be the root of A's right subtree. Then:
    1. **B has balance factor 0 or -1 after deletion** -- then perform a single left rotation
    2. **B has balance factor +1 after deletion** -- then perform a right-left rotation

The deletion operation is performed as follows...

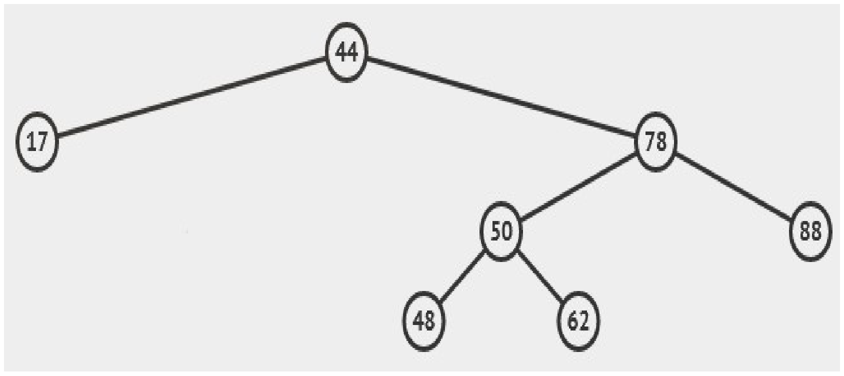
1. Let node to be deleted is **X**. Delete **X** from the tree using Binary Search Tree deletion logic.
2. After deletion, check the **Balance Factor** of nodes which are present in path from **X** to the root node.
3. If the **Balance Factor** of every node checked in step 2 is **0 or 1 or -1** then tree is said to be balanced and hence deletion process for a node has been completed.
4. If the **Balance Factor** of any node checked in step 2 is other than **0 or 1 or -1** then tree is said to be imbalanced. So perform the suitable **Rotation** (according to logic described above) to make it balanced. After rotation, new root of the sub-tree involved in rotation, becomes **X**. Go to step 3 and repeat.

Note: Unlike insertions, one rotation may not be enough to restore balance of the tree to make it AVL tree.

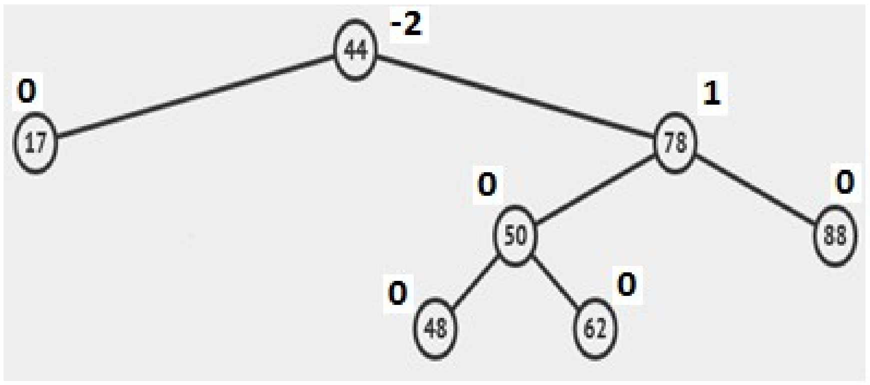
Example 1: Delete node **32** from the given tree and balance it, if required.



1. Delete node **32**



2. Check the balance factor of every node

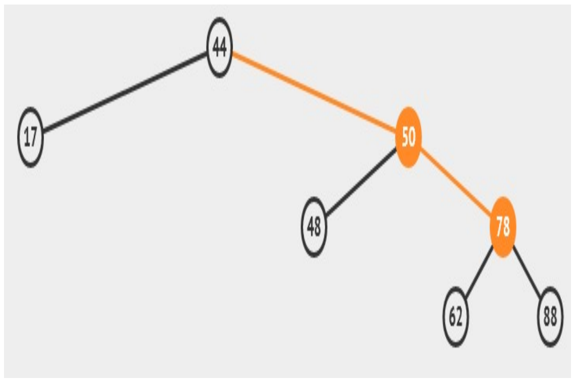


3. Balance the tree

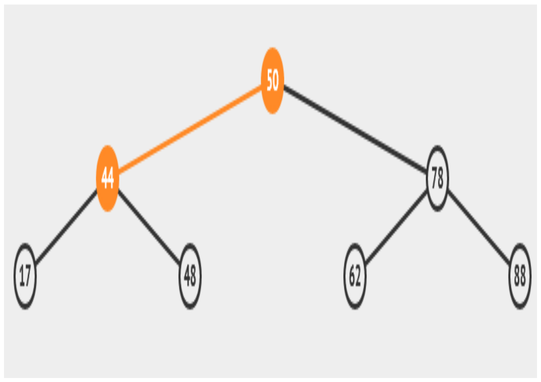
1.1 **44** is a critical node

1.2 Deletion is performed in left-subtree of critical node

1.3 **78** is a root node of right-subtree of a critical node with balance factor 1, so perform Right-Left rotation.

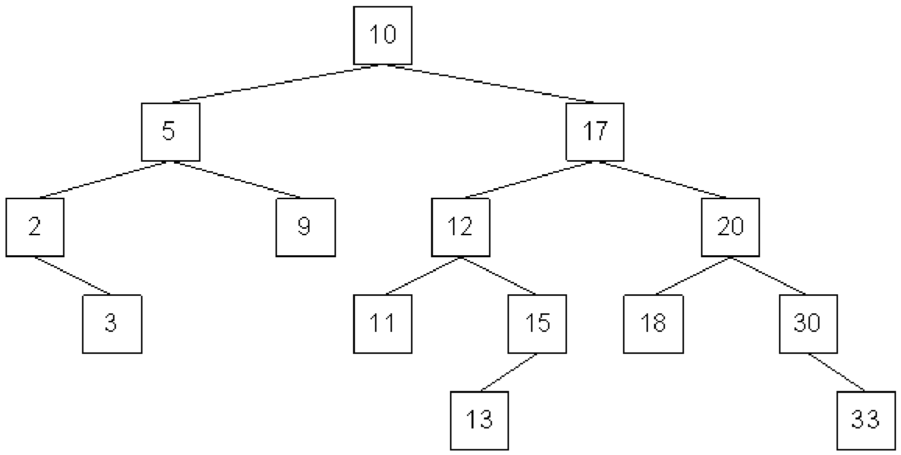


Right rotation on 78

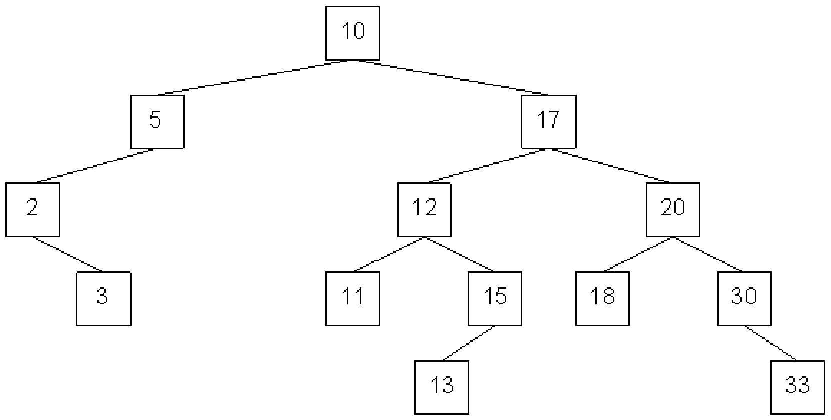


Left rotation on critical node 44

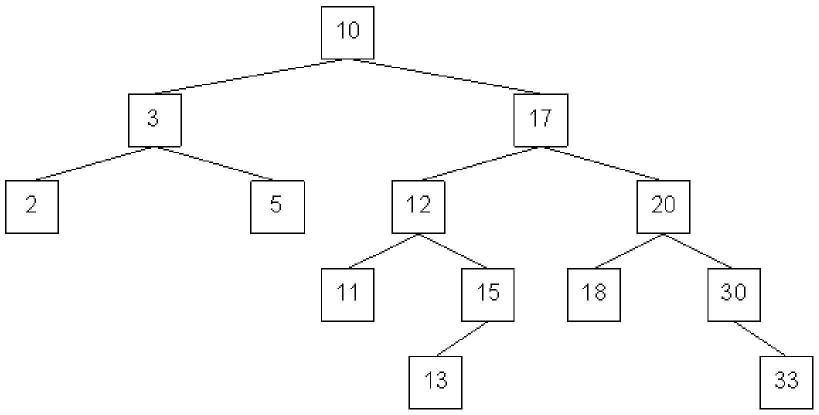
Example 2: Delete node **9** from the given tree and balance it, if required



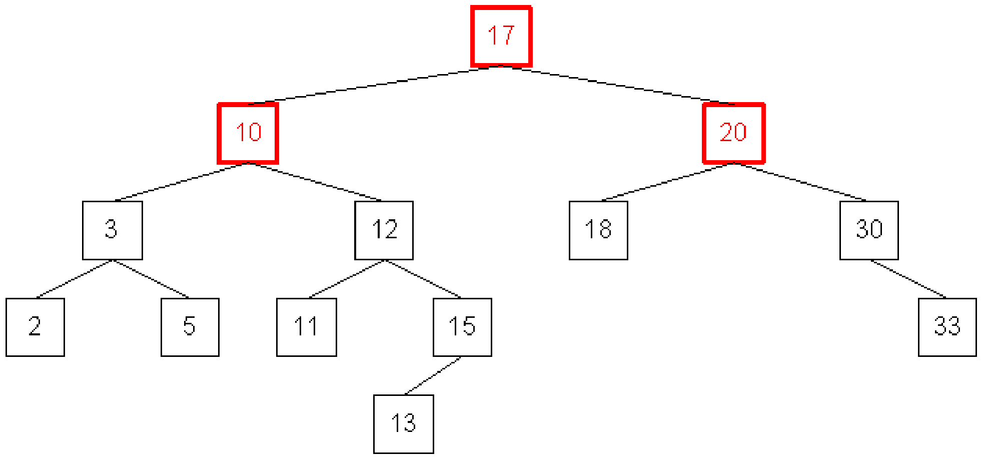
1. Delete node **9**



2. Balance tree at critical node **5 – LR** rotation is required

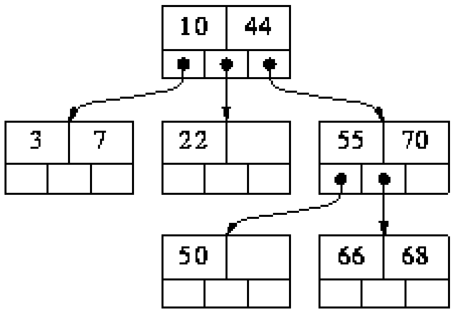


3. Balance tree at critical node **10 – L** rotation is required


17
10
20
3
12
18
30
2
5
11
15
33
13

Multiway (M-way) Search Trees

- A binary search tree has *one* key in each node and maximum *two* children.
- This notion easily generalizes to an M-way search tree, which has maximum **(M-1)** keys per node and maximum number_of_keys_in_node + 1 children. A node in M-way search tree may have **1** to **M-1** keys and may have **0** to **number_of_keys_in_node+1** children.
- **M** is called the *degree* of the tree. (Not to confuse with degree of a node in graph)
- A binary search tree, therefore, has degree 2. In other words, BST can be considered as 2-way search tree.
- Keys in a node are stored in ascending order, $K_1 < K_2 < ... K_n$ (n <= M-1). A child of a node can be placed at either end or in between two keys.
- All the keys in k_i's left subtree would be less than K_i; all the keys in K_i's right subtree would be greater than K_i. Hence all keys in subtree between K_i and K_{i+1} are greater than K_i and less than K_{i+1}.
- For example, here is a 3-way search tree:



Search operation

- The algorithm for searching for a key in an M-way search tree is the obvious generalization of the algorithm for searching in a binary search tree. If we are searching for key X are and currently at node containing keys $K_1...K_n$, there are four possible cases that can arise:
  - If $X < K_1$, recursively search for X in K_1's left subtree.
  - If $X > K_n$, recursively search for X in K_n's right subtree.
  - If $X=K_i$, for some i, then we are done (X has been found).
  - The only remaining possibility is that, for some i, $K_i < X < K_{i+1}$. In this case recursively search for X in the subtree that is in between K_i and K_{i+1}.
- For example, suppose we were searching for key 68 in the tree above. At the root, case (2) would apply, so we would continue the search in right subtree of 44 (K_2). At the root of this subtree, case (4) applies, 68 is between K_1=55 and K_2=70, so we would continue to search in the subtree between them. Now case (3) applies, 68=K_2, so we are done.
- If we had been searching for key 69, exactly the same processing would have occurred down to the last node. At that point, case (2) would apply, but the subtree we want to search in is empty. Therefore we conclude that 69 is not in the tree.

Insertion and Deletion operations

- The other algorithms for binary search trees - insertion and deletion - generalize in a similar way.
- As with binary search trees, inserting values in ascending order will result in a highly imbalanced M-way search tree; i.e. a tree whose height is O(N) instead of O(logN). This is a problem because all the important operations are O(height), and it is our aim to make them O(logN).
- One solution to this problem is to *force* the tree to be **height-balanced**.
- Such height-balanced M-way search tree is called **B-Tree**

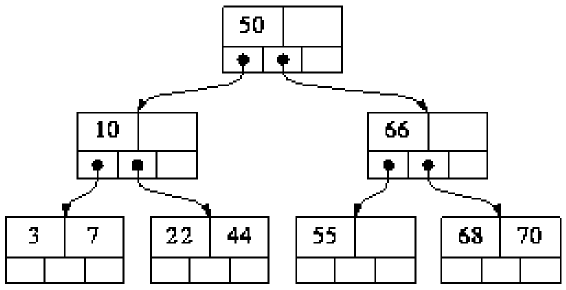
B – Tree

- B-Tree is a self-balanced M-way search tree.
- A B-Tree with order M has following properties:
  1. It is M-way search tree
  2. All the leaf nodes must be at same level.
  3. All non leaf nodes except root (i.e. all internal nodes) must have at least **ceiling (M/2)** children.
  4. If the root node is a non leaf node, then it must have at least 2 children. And if root node is leaf then it will have atleast 1 key.
  5. A tree is constructed in Bottom-Up manner. It makes sure that all internal nodes have exactly **number_of_keys_in_node+1** children.

Rules regarding number of keys and children can be summerized as follow:

		Keys	Children
Root	leaf	1 to M-1	0
	non-leaf	1 <= x <= M - 1	x+1
Non-Root	leaf	ceiling(M/2) – 1 to M-1	0
	non-leaf	ceiling(M/2) – 1 <= x <= M-1	x+1

- Example:



Insertion Operation

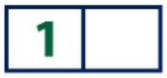
In a B-Tree, the new key must be added only at leaf node. The insertion operation is performed as follows...

1. Check whether tree is Empty.
2. If tree is **Empty**, then create a new node with new key and insert into the tree as a root node.
3. If tree is **Not Empty**, then find a leaf node to which the new key can be added using Binary Search Tree logic.
4. If that leaf node has an empty position, then add the new key to that leaf node by maintaining ascending order of keys within the node.
5. If that leaf node is already full, then **split** that leaf node by sending middle key to its parent node. Repeat the same until sending value is fixed into a node.
6. If the splitting is occurring at the root node, then the middle value becomes new root node for the tree and the height of the tree is increases by one.

Example 1: Construct a B-Tree of order 3 **(2-3 Tree)** by inserting numbers 1 to 10

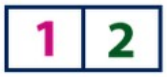
insert(1)

Since '1' is the first element into the tree that is inserted into a new node. It acts as the root node.



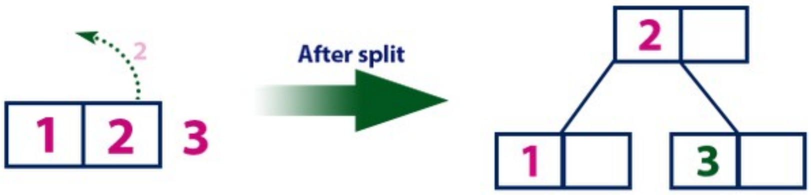
insert(2)

Element '2' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node has an empty position. So, new element (2) can be inserted at that empty position.



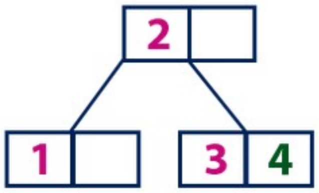
insert(3)

Element '3' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node doesn't has an empty position. So, we split that node by sending middle value (2) to its parent node. But here, this node doesn't has parent. So, this middle value becomes a new root node for the tree.



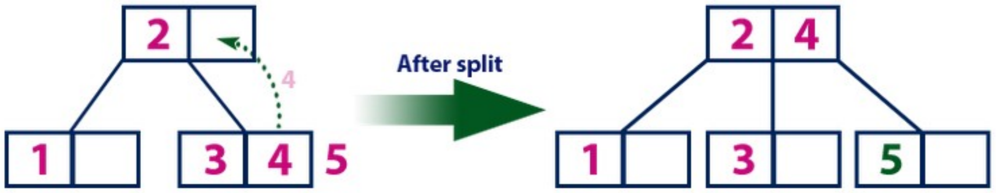
insert(4)

Element '4' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node with value '3' and it has an empty position. So, new element (4) can be inserted at that empty position.



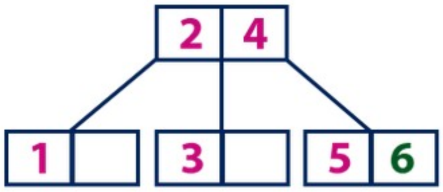
insert(5)

Element '5' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (4) to its parent node (2). There is an empty position in its parent node. So, value '4' is added to node with value '2' and new element '5' added as new leaf node.



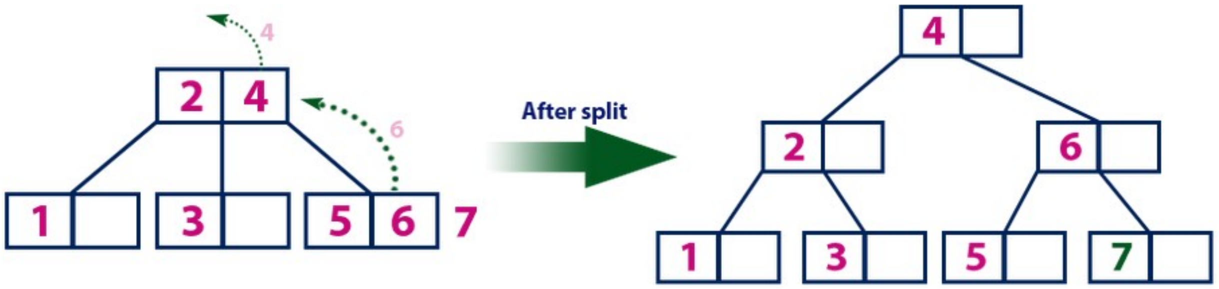
insert(6)

Element '6' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node with value '5' and it has an empty position. So, new element (6) can be inserted at that empty position.



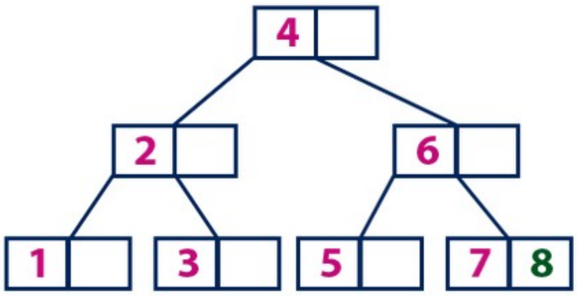
insert(7)

Element '7' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (6) to its parent node (2&4). But the parent (2&4) is also full. So, again we split the node (2&4) by sending middle value '4' to its parent but this node doesn't have parent. So, the element '4' becomes new root node for the tree.



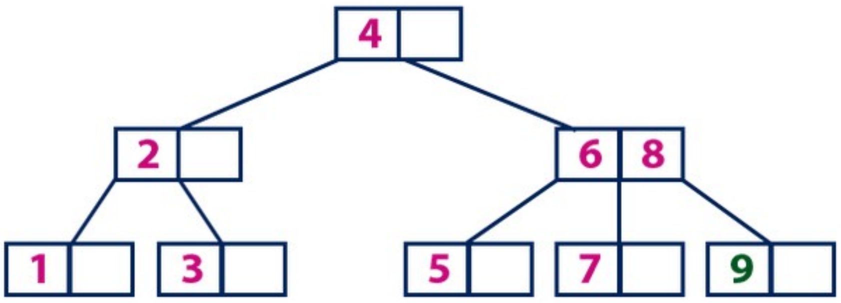
insert(8)

Element '8' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '8' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7) and it has an empty position. So, new element (8) can be inserted at that empty position.



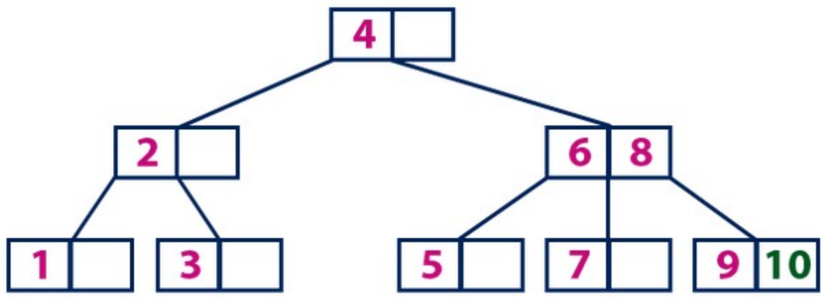
insert(9)

Element '9' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '9' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7 & 8). This leaf node is already full. So, we split this node by sending middle value (8) to its parent node. The parent node (6) has an empty position. So, '8' is added at that position. And new element is added as a new leaf node.



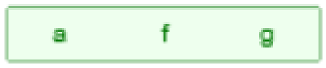
insert(10)

Element '10' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with values '6 & 8 '. '10' is larger than '6 & 8' and it is also not a leaf node. So, we move to the right of '8'. We reach to a leaf node (9). This leaf node has an empty position. So, new element '10' is added at that empty position.

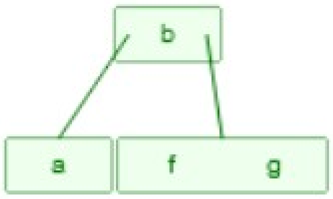


Example 2: Construct a B-Tree of order 4 **(2-3-4 Tree)** by inserting values **a, g, f, b, k, d, h, m, j, e, s, i, r, x, c, l, n, t, u, p.**

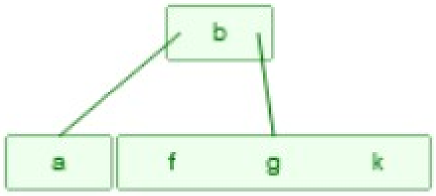
1. Insert **a, g, f**



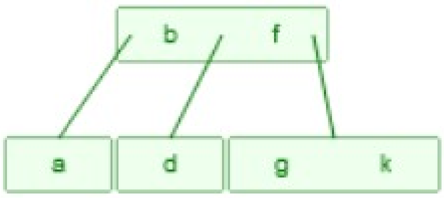
2. Insert **b**



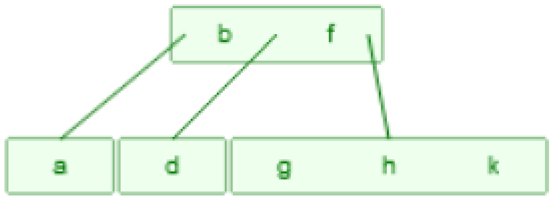
3. Insert **k**



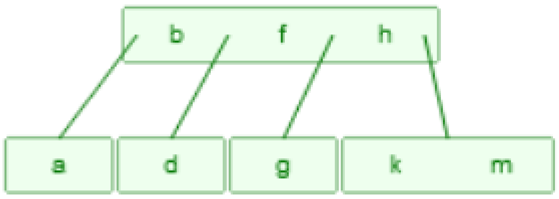
4. Insert **d**



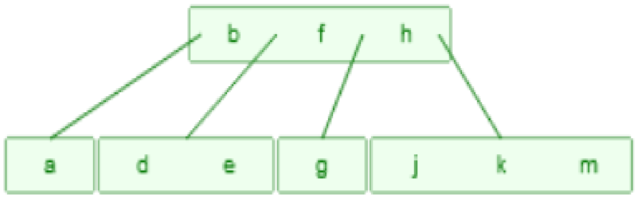
5. Insert **h**



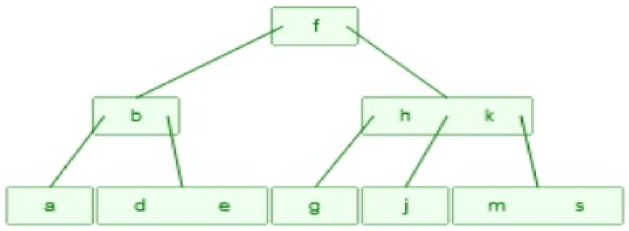
6. Insert **m**



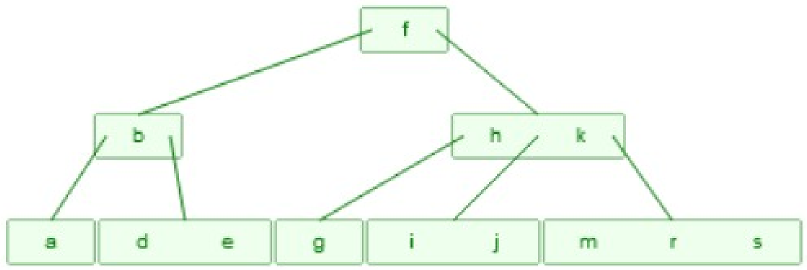
7. Insert **j,e**



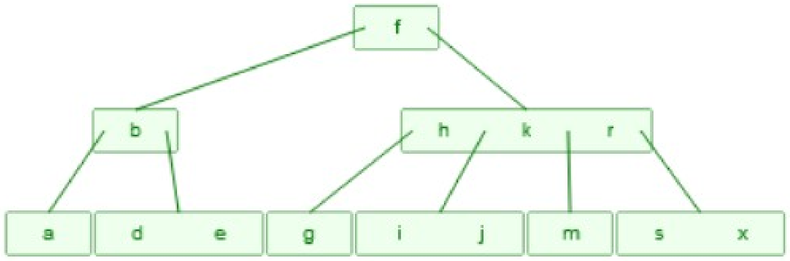
8. Insert **s**



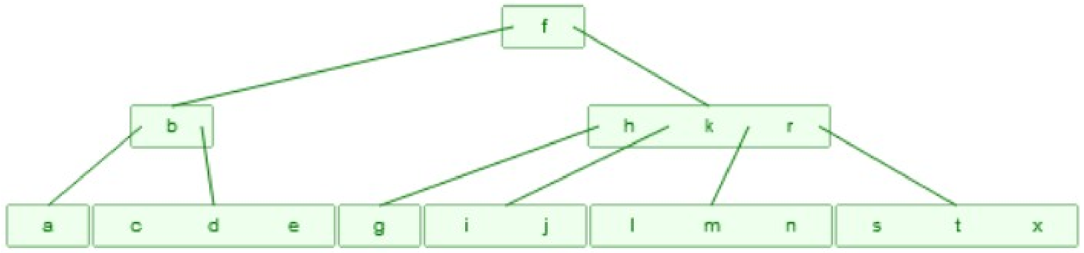
9. Insert **i,r**



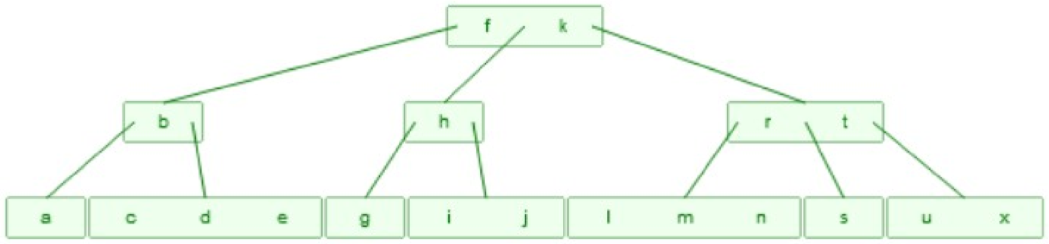
10. Insert **x**



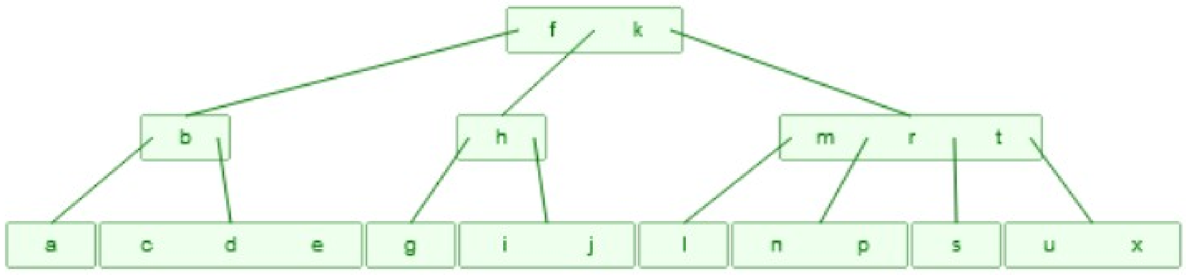
11. Insert **c,l,n,t**



12. Insert **u**



13. Insert **p**

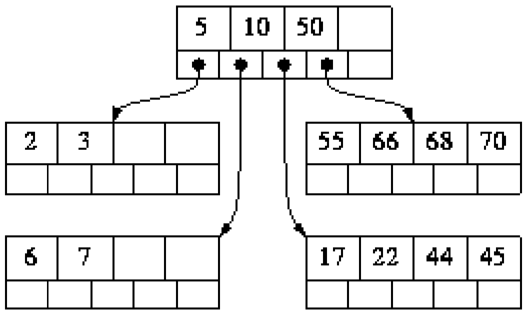


Deletion Operation

- Recall deletion algorithm for binary search trees: if the key to be deleted is in an internal node, it would be replaced by its inorder successor's (or predecessor's) key and then delete the node which originally contained successor (or predecessor) value.
- A similar strategy is used to delete a key from a B-tree. If the key to be deleted does not occur in a leaf, it will be replaced by its successor (the smallest key in its right subtree) or predecessor (the largest key in its left subtree) and then successor or predecessor key from the node that originally contained it will be deleted.
- As in a B-tree, the successor or predecessor is guaranteed to be in leaf. Therefore wherever the key to be deleted initially resides, the following deletion algorithm always begins at a leaf.
- To delete key X from a B-tree, starting at a leaf node, there are 2 steps:
    1. Remove X from the current node. Being a leaf node there are no subtrees to worry about.
    2. Removing X might cause the node containing it to have *too few* keys.
- It is required that the root to have at least **1** key in it and all other nodes to have at least **ceiling(M/2)–1** {*which is same as* ***floor((M-1)/2)***} keys in them. If the node has too few keys, it has ***underflowed***.
- If underflow does not occur, then the deletion process is finished.
- If it does occur, it must be fixed. The process for fixing a root is slightly different than the process for fixing the other nodes, and will be discussed afterwards.

How to fix a non-root node that has underflowed?

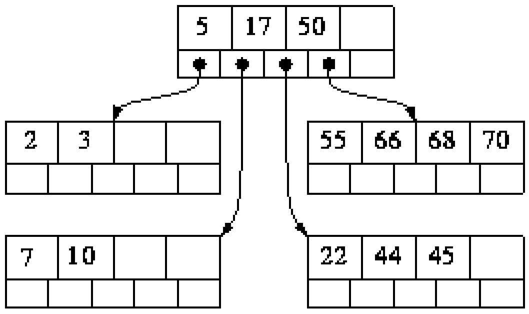
Let us take a specific example, deleting 6 from following B-tree (of degree 5):



- Removing 6 causes the node it is in to underflow, as it now contains just 1 value (7).
- The strategy for fixing this is to try to `borrow' keys from a neighbouring node.
- The current node and its most populous neighbour will be joined together to form a `combined node' - and combined node must also includes the key in the parent node that is in between these two nodes.
- In this example, node [7] will be joined with its more populous neighbour [17 22 44 45] and **'10'** will be put in between them, to create [7 10 17 22 44 45]
- How many keys might there be in this combined node?
    - The parent node contributes ***1*** key.
    - The node that underflowed contributes exactly ***floor((M-1)/2)-1*** keys.
    - The neighbouring node contributes somewhere between ***floor((M-1)/2)*** and ***(M-1)*** keys.
- The treatment of the combined node is different depending on whether the neighbouring node contributes exactly ***floor((M-1)/2)*** keys or more than this number.

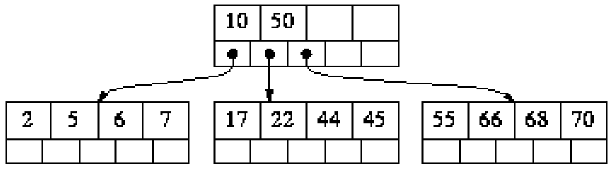
Case 1: Suppose that the most populous neighbouring node contains **more than *floor((M-1)/2)*** keys.

- In this case, the total number of keys in the combined node is strictly greater than
  1 + floor((M-1)/2)-1 + floor((M-1)/2)
  i.e. it is strictly greater than ***2*floor((M-1)/2)***. So it must contain ***2*floor((M-1)/2)+1*** keys or more.
- The combined node will be split into three pieces: Left, Middle, and Right, where Middle is a single key in the very middle of the combined node.
- Because the combined node has ***2*floor((M-1)/2)+1*** keys or more, Left and Right nodes are guaranteed to have atleast ***floor((M-1)/2)*** keys each, and therefore are legitimate nodes.
- The key borrowed from the parent will be replaced by Middle and Left and Right will be used as its two children.
- In this case the parent's size does not change.
- This is what happens in example of deleting 6 from the tree above. The combined node [7 10 17 22 44 45] contains more than 5 keys, so it is split into:
  - Left = [7 10]
  - Middle = 17
  - Right = [22 44 45]
- Then put Middle into the parent node (in the position where the `10' had been) with Left and Right as its children



Case 2: Suppose that the most populous neighbouring node contains **exactly *floor((M-1)/2)*** keys.

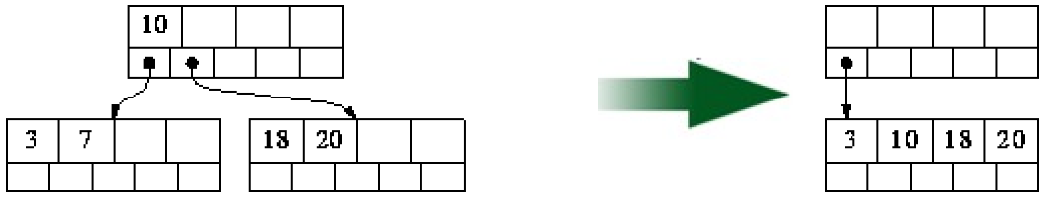
- Then the total number of keys in the combined node is
  1 + floor((M-1)/2)-1 + floor((M-1)/2) = 2*floor((M-1)/2) <= M-1
- In this case the combined node contains the right number of keys to be treated as a node.
- As a concrete example of this case, suppose that, in the above tree, 3 is deleted instead of 6. The node [2 3] underflows when 3 is removed.
- It would be combined with its more populous neighbour [6 7] and the intervening key from the parent (5) to create the combined node [2 5 6 7].
- This contains 4 keys, so it can be used without further processing. The result would be:



It is very important to note that the parent node now has one fewer key. This might cause *it* to underflow - imagine that 5 had been the *only* key in the parent node. If the parent node underflows, it would be treated in exactly the same way - combined with *it*s most populous neighbor etc. The underflow processing repeats at successive levels until no underflow occurs or until the root underflows.

How to fix a root node that has underflowed?

- For the root to underflow, it must have originally contained just one key, which now has been removed.
- If the root was also a leaf, then there is no problem: in this case the tree has become completely empty.
- If the root is not a leaf, it must originally have had two subtrees (because it originally contained one key). How could it possibly underflow?
- The deletion process always starts at a leaf and therefore the only way the root could have its key removed is through the Case 2 processing just described: the root's two children have been combined, along with the root's only key to form a single node. But if the root's two children are now a single node, then *that node* can be used as the new root, and the current root (which has underflowed) can simply be deleted.
- To illustrate this, suppose we delete 7 from the following B-tree (where M=5):



- The node [3 7] would underflow, and the combined node [3 10 18 20] would be created. This has 4 keys, which is acceptable when M=5. So it would be kept as a node, and **‘10’** would be removed from the parent node - the root. This is the only circumstance in which underflow can occur in a root that is not a leaf. The situation is this:
- Clearly, the current root node, now empty, can be deleted and its child used as the new root.

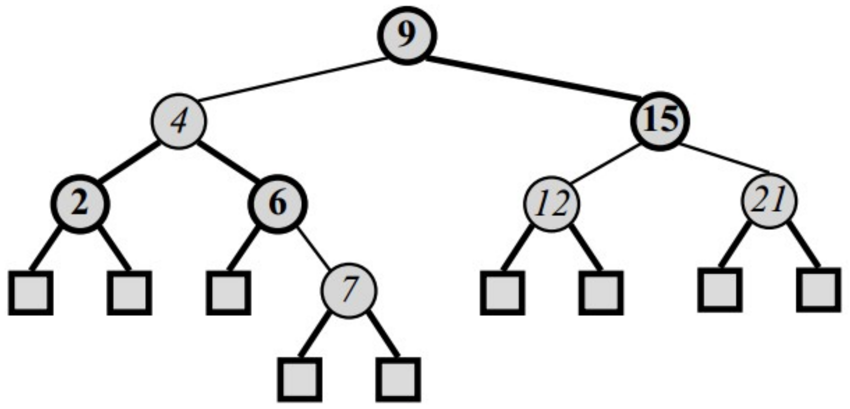
Example: Construct B-Tree of order 4 by inserting following keys and then delete the mentioned keys in the given order.

- Insert: 5, 3, 21, 9, 1, 13, 2, 7, 10, 12, 4, 8
- Delete: 2, 21, 10, 3, 4

RedBlack Tree (Symmetric Binary B-Tree)

Red-Black Tree is a self-balancing Binary Search Tree (BST) where every node follows following rules.

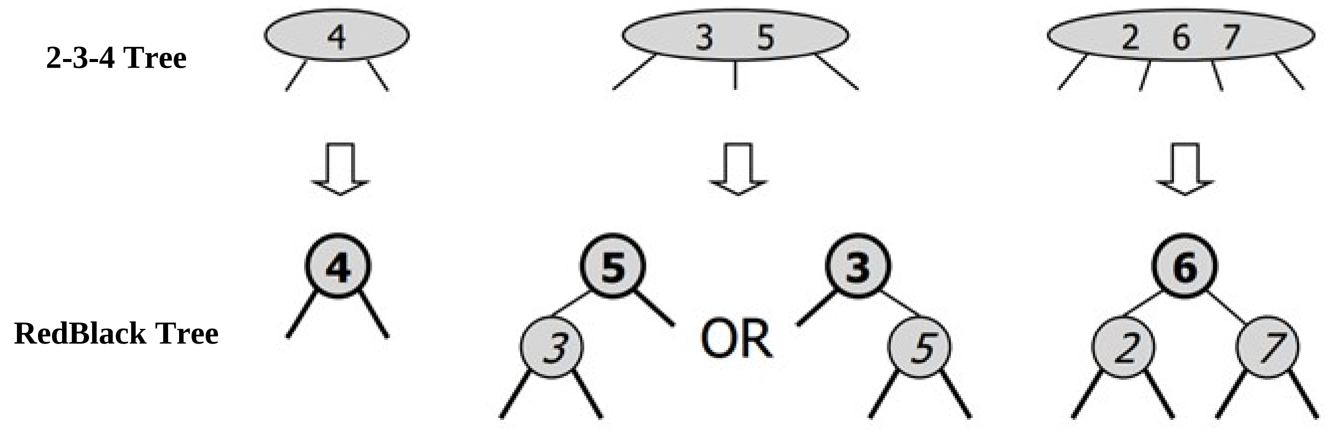
1) Every node has a colour either red or black.
2) Root of tree is always black.
3) There are no two adjacent red nodes (A red node cannot have a red parent or red child).
4) Every path from root to a NULL node has same number of black nodes (**black height of tree**).



RedBlack Tree

Relation with 2-3-4 Tree (i.e. B-Tree of order 4)

- A red-black tree is a representation of a 2-3-4 tree by means of a binary search tree whose nodes are coloured red or black.
- In comparison with its associated 2-3-4 tree, a red-black tree has
  - same logarithmic time performance
  - simpler implementation with a single node type
- Since black depth of all null nodes is the same, in the resultant 2-3-4 tree all leaves will be at same level.
- Let h be the black height of the tree with **n** nodes
  - If all nodes are black, it must be complete binary search tree where **$n=2^h-1$.** Least nodes possible in red-black tree of black height **h**.
  - If alternate levels of tree are red then **$n=2^{2h}-1$**, which is equal to **4^h-1** nodes. Hence, **$\log_4(n) < h < 1+\log_2(n)$ –** this is same as 2-3-4 tree.



Insertion

In Red-Black tree, two tools are used to do balancing.

1) Recoloring
2) Rotation

Recoloring is performed first, if recoloring doesn't work, then rotation is performed.

Following is detailed algorithm. The algorithm has mainly two cases depending upon the color of uncle. If uncle is red, we do recoloring. If uncle is black, we do rotations and/or recoloring.

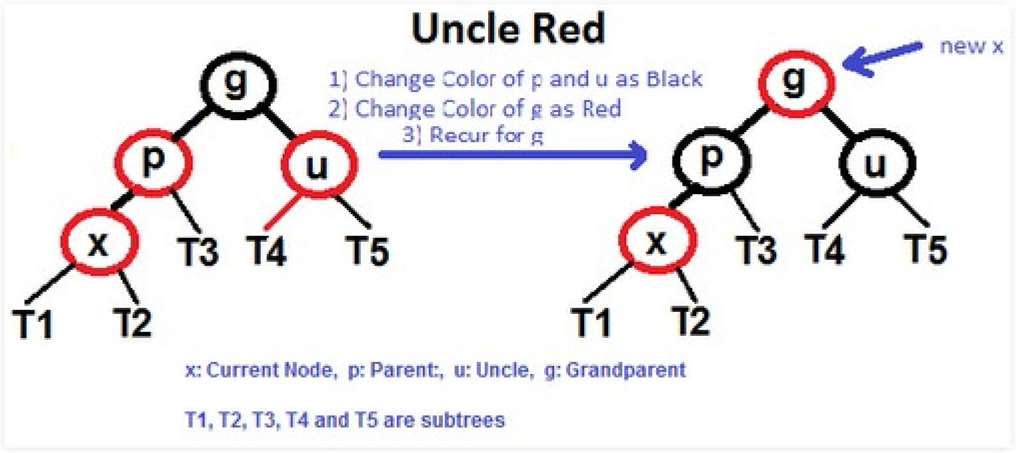
Color of a NULL node is considered as BLACK.

Algorithm:

1. Let x be the newly inserted node. Perform standard BST insertion and make the color of newly inserted node as RED. Hence black height of the tree remains unchanged but it can introduce double red problem.
2. If x is root, change color of x as BLACK (Black height of complete tree increases by 1).
3. Do following if x is not root and color of x's parent is not BLACK. If x's parent is black, no further action required as it can not introduce double red problem.

   Case1: If x's uncle is RED

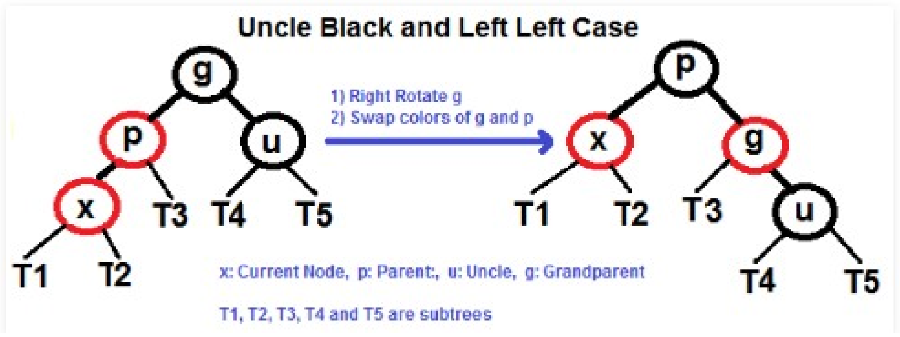
   a. Change color of parent and uncle as BLACK.
   b. Color of grand parent as RED.
   c. Change x = x's grandparent, repeat steps 2 and 3 for new x.


Uncle Red
1) Change Color of p and u as Black
2) Change Color of g as Red
3) Recur for g
new x
g
p
u
x
T1
T2
T3
T4
T5
x: Current Node, p: Parent:, u: Uncle, g: Grandparent
T1, T2, T3, T4 and T5 are subtrees

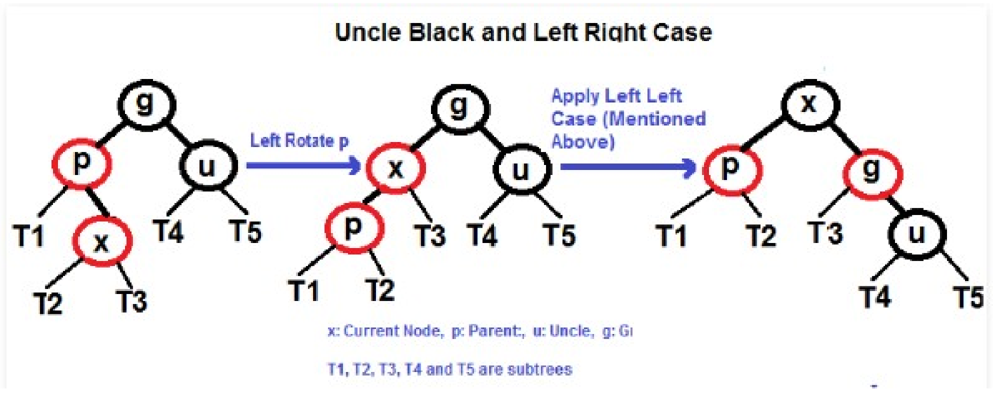
   Case2: If x's uncle is BLACK , then there can be 4 configurations for x, x's parent (p) and x's grandparent (g) similar to AVL Tree.

   1. Left Left Case (p is left child of g and x is left child of p)
   2. Left Right Case (p is left child of g and x is right child of p)
   3. Right Right Case (Mirror of case a)
   4. Right Left Case (Mirror of case c)

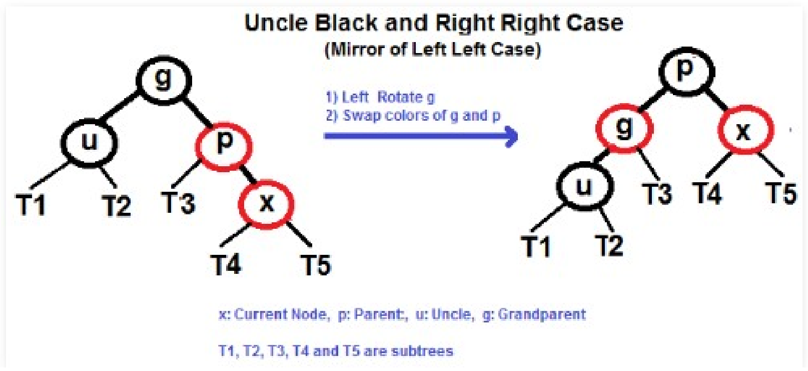
1. **Left Left Case**


Uncle Black and Left Left Case
g
p
u
x
T1
T2
T3
T4
T5
1) Right Rotate g
2) Swap colors of g and p
p
x
g
u
T1
T2
T3
T4
T5
x: Current Node, p: Parent:, u: Uncle, g: Grandparent
T1, T2, T3, T4 and T5 are subtrees

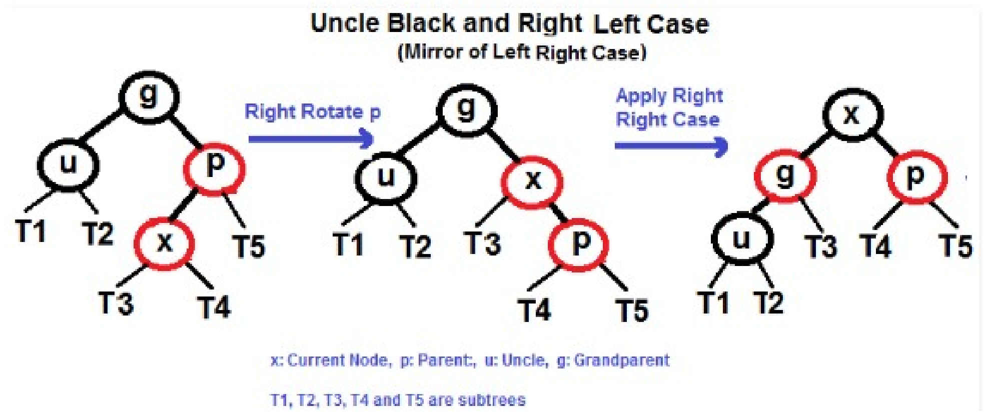
2. **Left Right Case**


Uncle Black and Left Right Case
g
p
u
T1
x
T2
T3
T4
T5
Left Rotate p
g
x
u
p
T1
T2
T3
T4
T5
Apply Left Left Case (Mentioned Above)
x
p
g
u
T1
T2
T3
T4
T5
x: Current Node, p: Parent:, u: Uncle, g: Gi
T1, T2, T3, T4 and T5 are subtrees

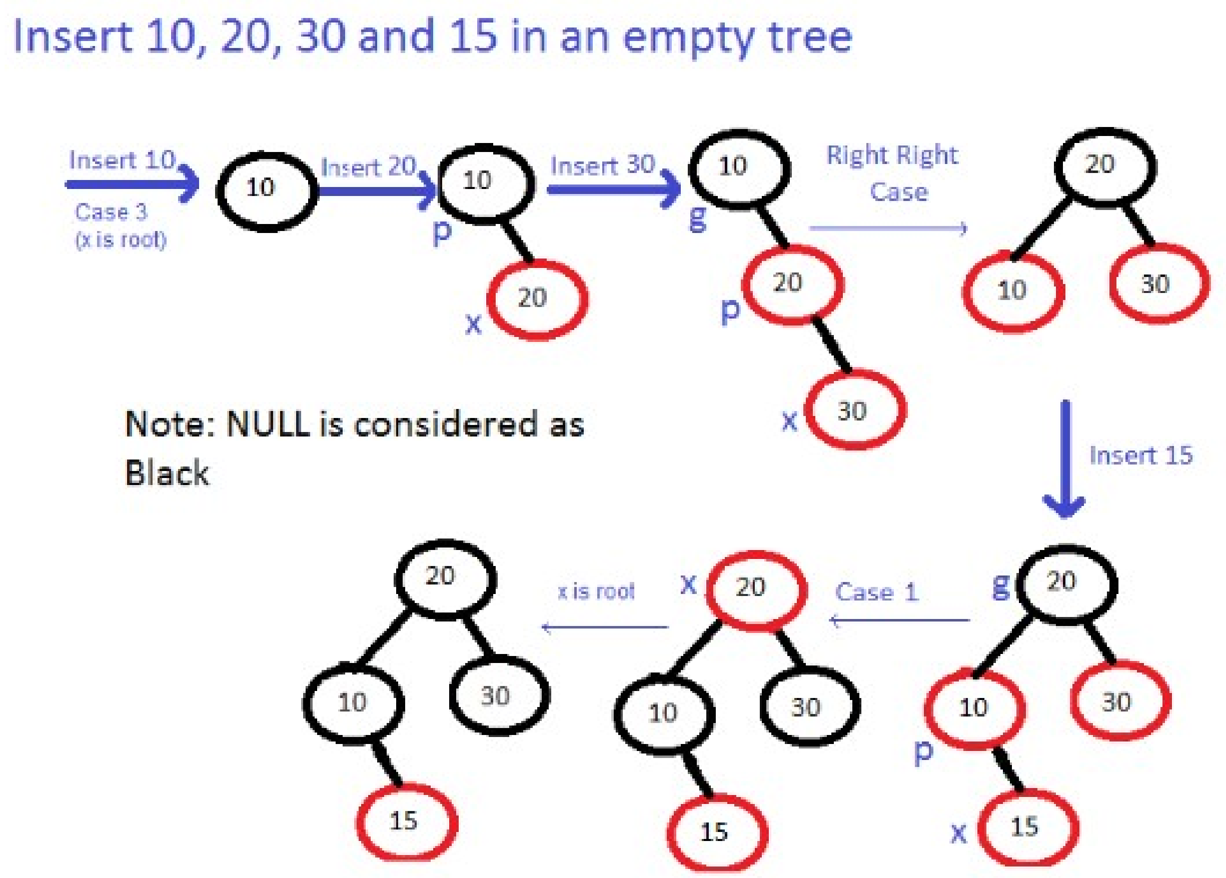
3. **Right Right Case**


Uncle Black and Right Right Case
(Mirror of Left Left Case)
g
u
p
x
T1
T2
T3
T4
T5
1) Left Rotate g
2) Swap colors of g and p
p
g
x
u
T1
T2
T3
T4
T5
x: Current Node, p: Parent:, u: Uncle, g: Grandparent
T1, T2, T3, T4 and T5 are subtrees

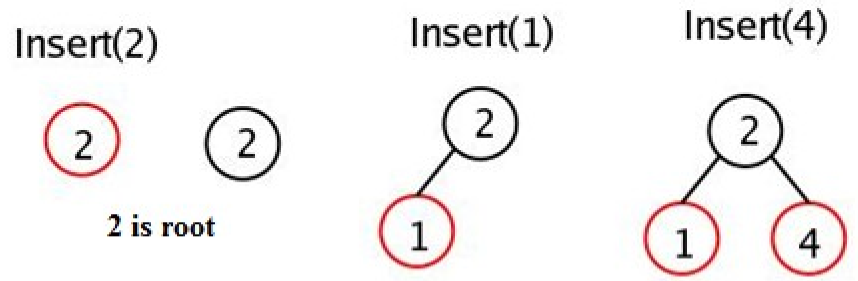
4. **Right Left Case**

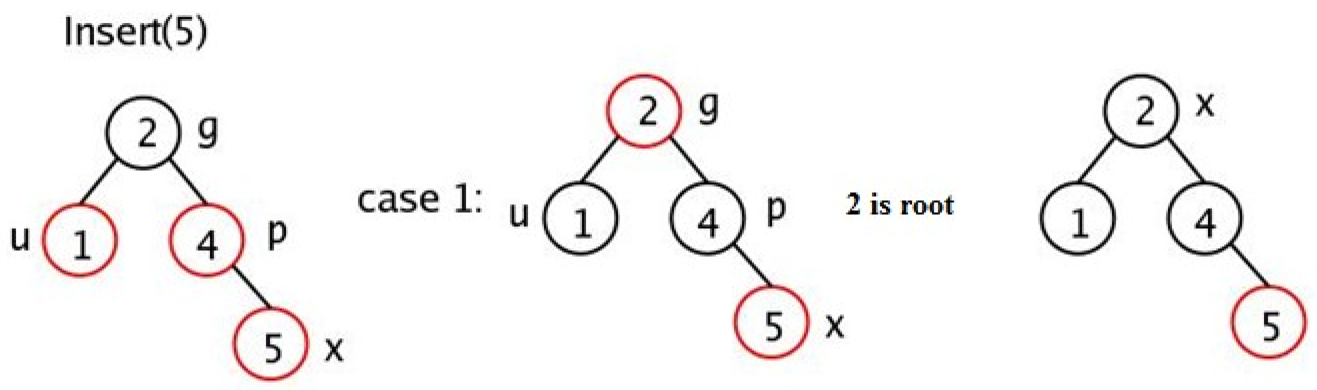

Uncle Black and Right Left Case
(Mirror of Left Right Case)
g
u
p
x
T1
T2
T3
T4
T5
Right Rotate p
g
u
x
p
T1
T2
T3
T4
T5
Apply Right Right Case
x
g
p
u
T1
T2
T3
T4
T5
x: Current Node, p: Parent:, u: Uncle, g: Grandparent
T1, T2, T3, T4 and T5 are subtrees

Example1:

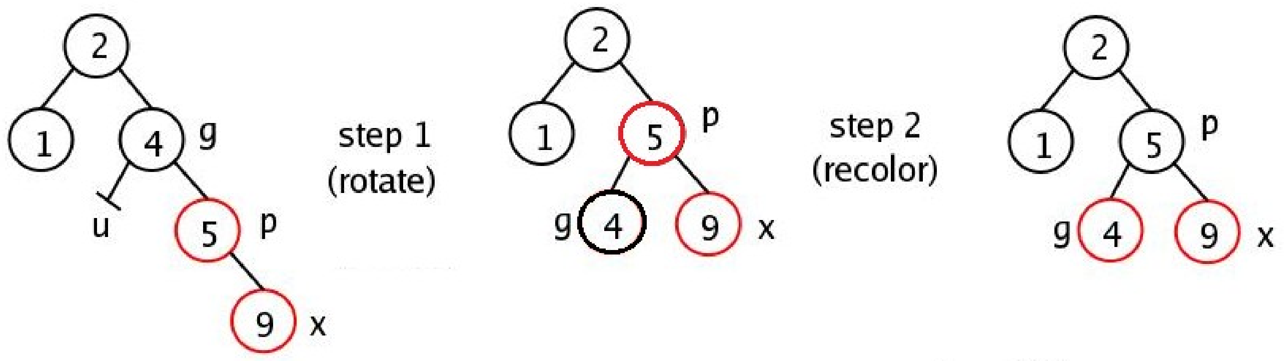


Example 2: Insert **2, 1, 4, 5, 9, 3, 6, 7**

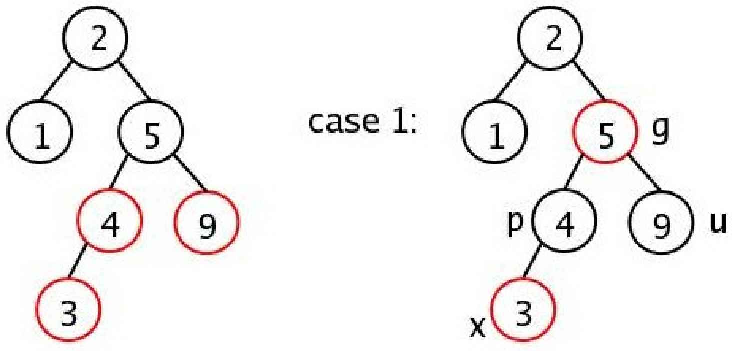




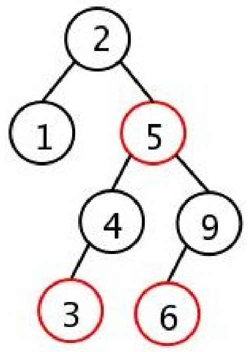
Insert(9)



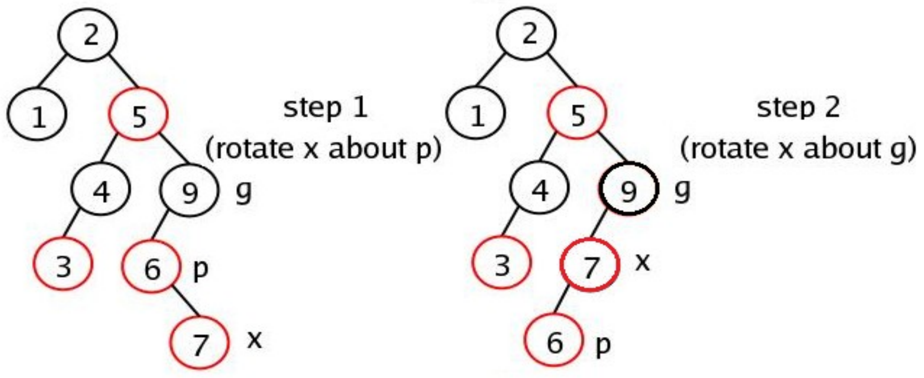
Insert(3)

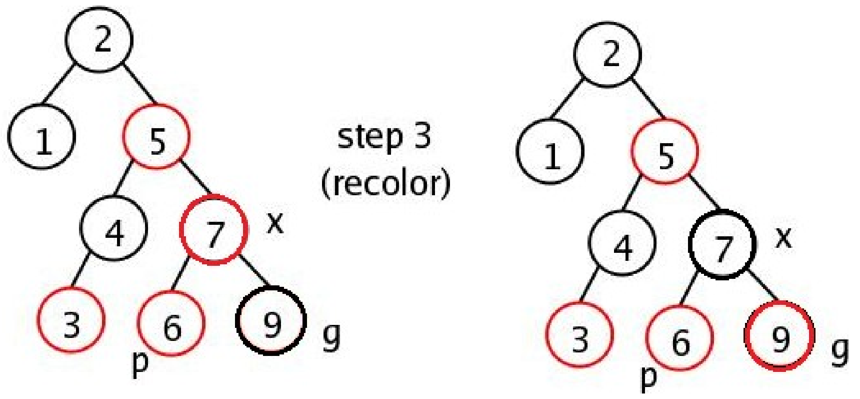


Insert(6)



Insert(7)





Deletion

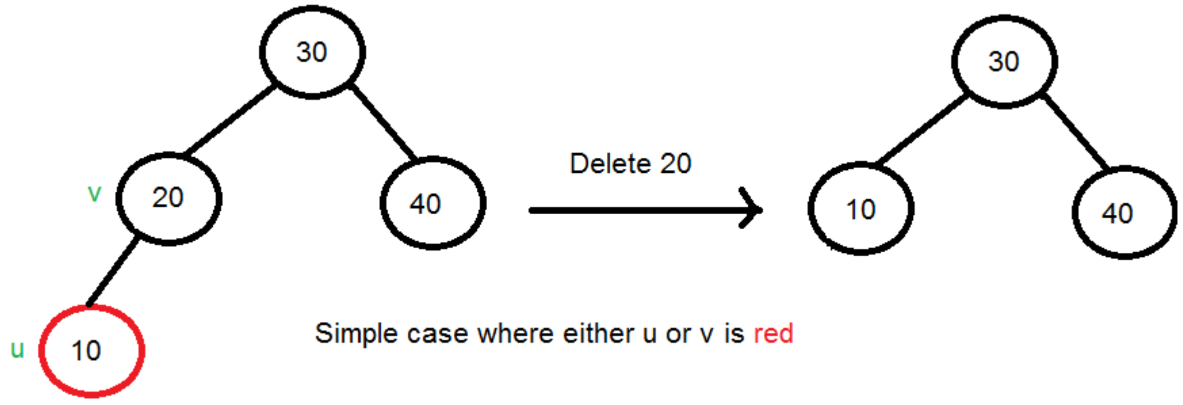
- Like Insertion, recoloring and rotations are used to maintain the Red-Black properties.
- In insert operation, color of uncle is checked to decide the appropriate case. In delete operation, ***color of sibling is checked*** to decide the appropriate case.
- The main property that violates after insertion is two consecutive reds. In delete, the main violated property is, change of black height in subtrees as deletion of a black node may cause reduced black height by one in root to leaf path.
- Deletion is fairly complex process. To understand deletion, notion of double black is used. When a black node is deleted and replaced by a black child, the child is marked as ***double black***. The main task now becomes to convert this double black to single black.

Deletion Steps

Following are detailed steps for deletion.

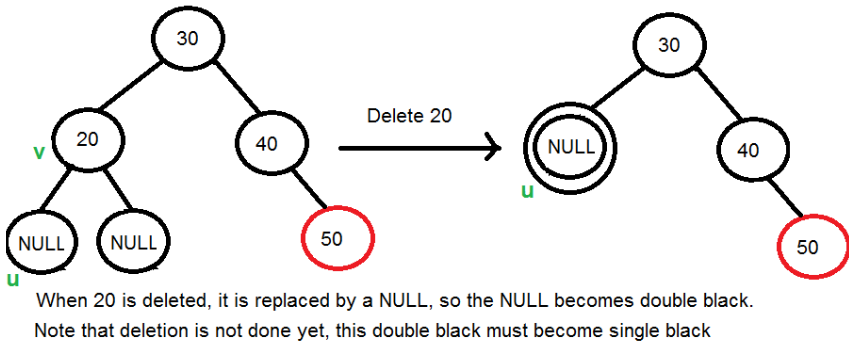
1. Perform standard BST delete. When standard delete operation is performed in BST, it always ends up deleting a node which is either leaf or has only one child (For an internal node, copy the successor and then recursively call delete for successor, successor is always a leaf node or a node with one child). So it is needed to handle only those cases where a node is leaf or has one child. Let ***v*** be the node to be deleted and ***u*** be the child that replaces ***v*** (Note that u is NULL when ***v*** is a leaf and color of NULL is considered as Black).

2. **Simple Case: If either u or v is red,** mark the replaced child as black (No change in black height). Note that both u and v cannot be RED as v is parent of u and two consecutive red nodes are not allowed in red-black tree.



3. **If Both *u* and *v* are Black**.

3.1. Color ***u*** as double black. Now task reduces to convert this double black to single black. Note that If ***v*** is leaf, then ***u*** is NULL and colour of NULL is considered as black. So the deletion of a black leaf also causes a double black.


30
20
v
40
NULL
NULL
u
50
Delete 20
30
NULL
u
40
50
When 20 is deleted, it is replaced by a NULL, so the NULL becomes double black.
Note that deletion is not done yet, this double black must become single black

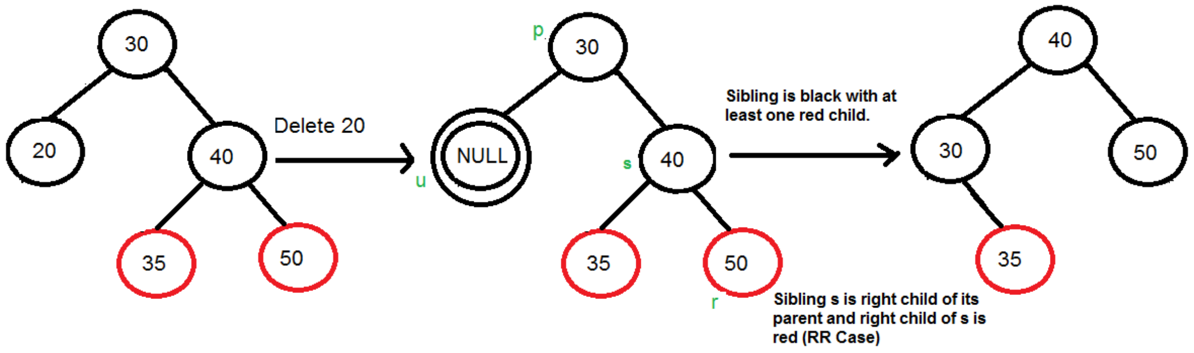
3.2. Do following while the current node u is double black and it is not root. Let sibling of node is ***s***.

a. **If sibling s is black and at least one of sibling's children is red**, perform rotation(s). Let the red child of s be **r**. This case can be divided in four subcases depending upon positions of s and r.

1. **Left Left Case:** ***s*** is left child of its parent and ***r*** is left child of ***s*** or both children of ***s*** are red. This is mirror of right-right case shown in below diagram.

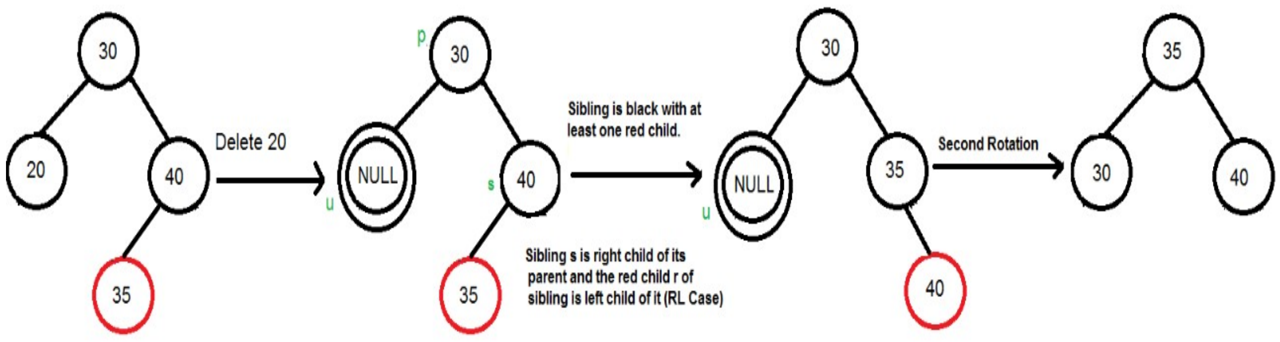
2. **Left Right Case**: ***s*** is left child of its parent and ***r*** is right child. This is mirror of right left case shown in below diagram.

3. **Right Right Case:** ***s*** is right child of its parent and ***r*** is right child of s or both children of ***s*** are red


30
20
40
35
50
Delete 20
p
30
NULL
u
s
40
35
50
r
Sibling is black with at least one red child.
40
30
50
35
Sibling s is right child of its parent and right child of s is red (RR Case)

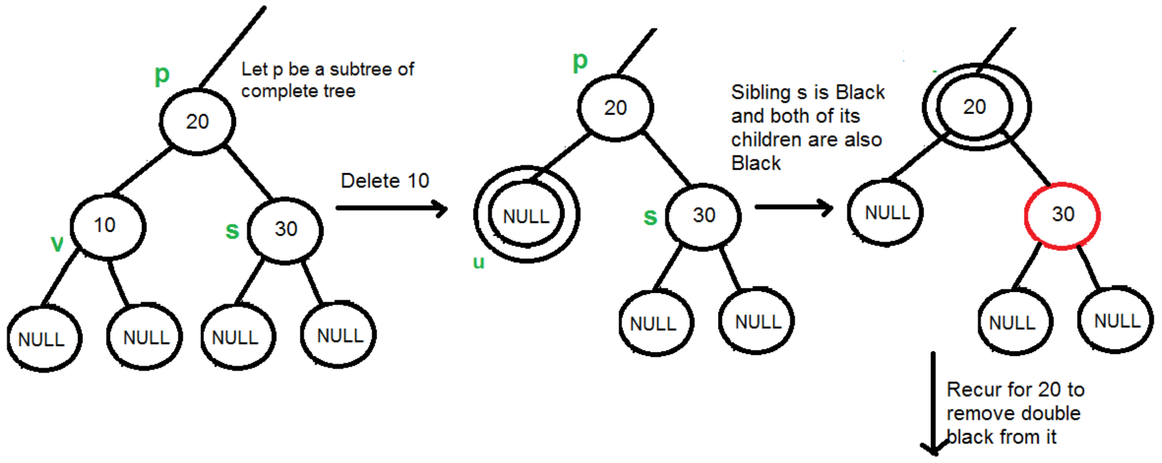
After rotation colors of ***p*** and ***s*** are intechanged and color of ***r*** is changed to black.

4. **Right Left Case: s** is right child of its parent and ***r*** is left child of ***s***


30
20
40
35
Delete 20
p
30
NULL
u
s
40
35
Sibling is black with at least one red child.
Sibling s is right child of its parent and the red child r of sibling is left child of it (RL Case)
30
NULL
u
35
40
Second Rotation
35
30
40

After first rotation colors of s and r, are interchanged. After second rotation colors of new sibling (got after first rotation – 35 here) and p (30 here) are interchanged and color of new red (got after first rotation – 40 here) is changed to black.

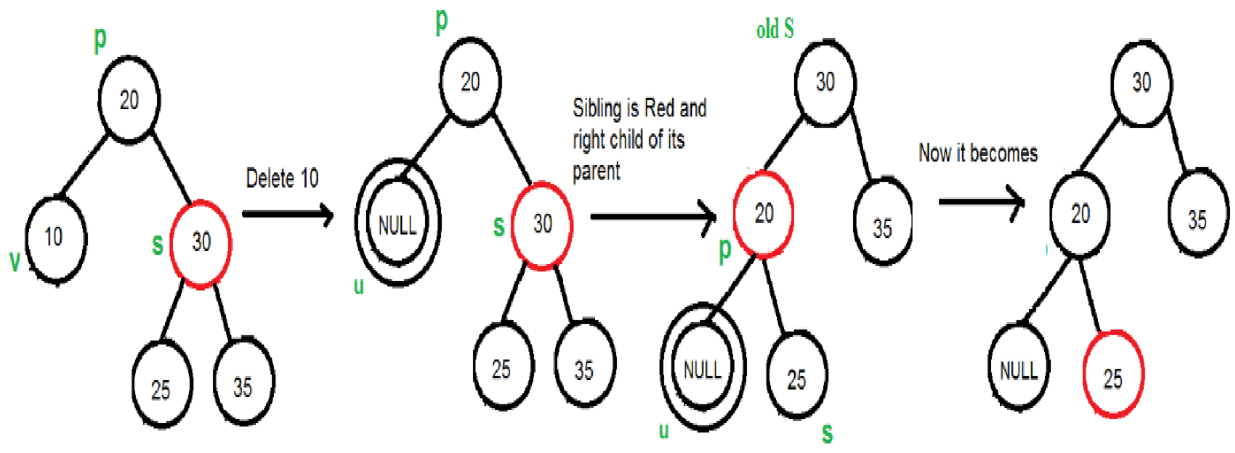
b. **If sibling s is black and its both children are black**, then perform recoloring, and recur for the parent if parent is black.


p
Let p be a subtree of complete tree
20
10
v
s
30
NULL
NULL
NULL
NULL
Delete 10
p
20
NULL
u
s
30
NULL
NULL
Sibling s is Black and both of its children are also Black
20
NULL
30
NULL
NULL
Recur for 20 to remove double black from it

In this case, if parent was red, then we didn't need to recur for parent, we can simply make it black (red + double black = single black)

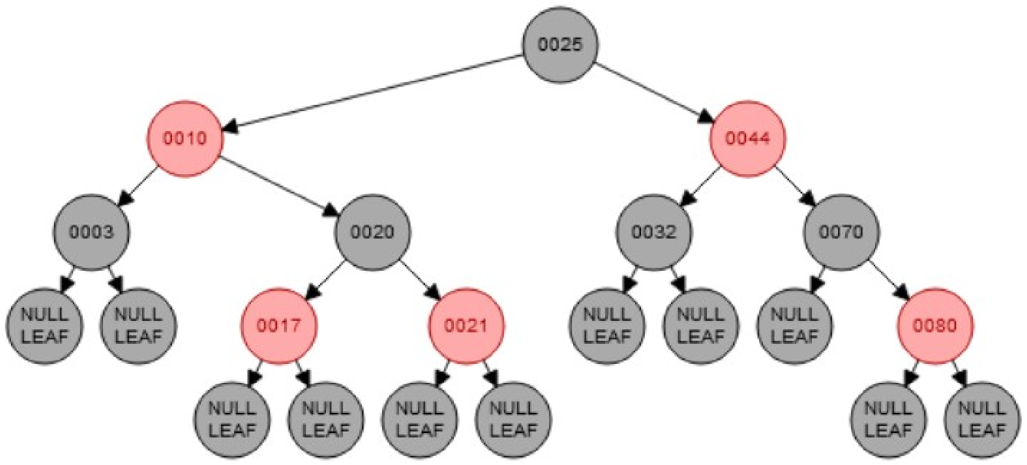
c. **If sibling is red,** perform a rotation to move old sibling up, recolor the old sibling and parent. The new sibling is always black (See the below diagram). This mainly converts the tree to black sibling case (by rotation) and leads to case (a) or (b). This case can be divided in two subcases.

1. **Left Case**: ***s*** is left child of its parent. This is mirror of right case shown in below diagram. Right rotate the parent ***p***.
2. **Right Case**: ***s*** is right child of its parent. Left rotate the parent ***p***.

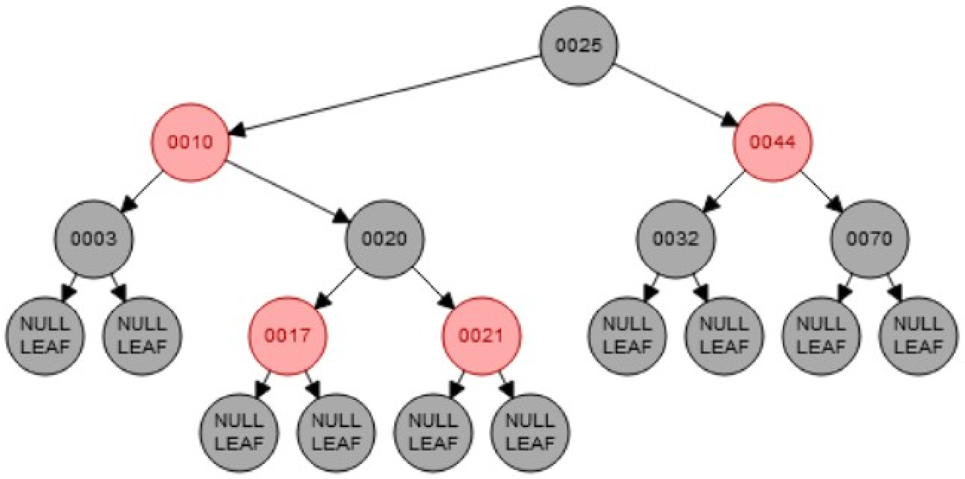


3.3. If ***u*** is root, make it single black and return. Black height of complete tree reduces by 1.

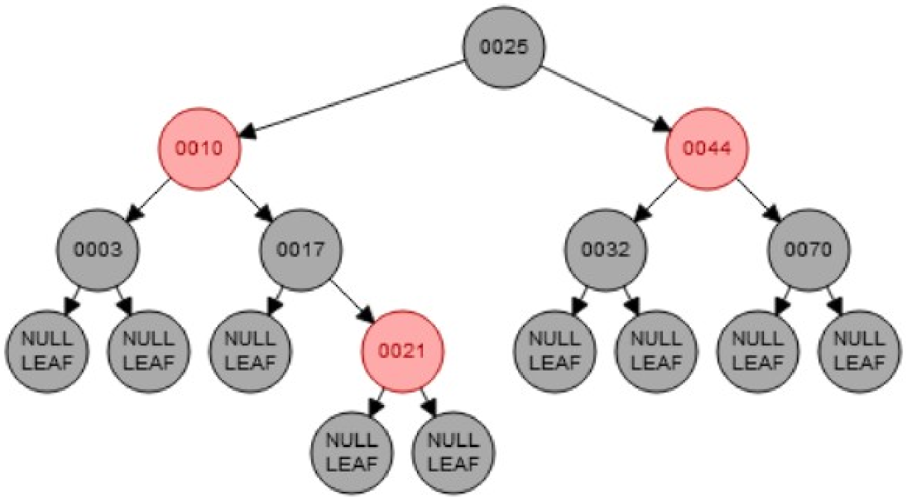
Example: Delete keys 80, 20, 32 from the following Red-Black Tree



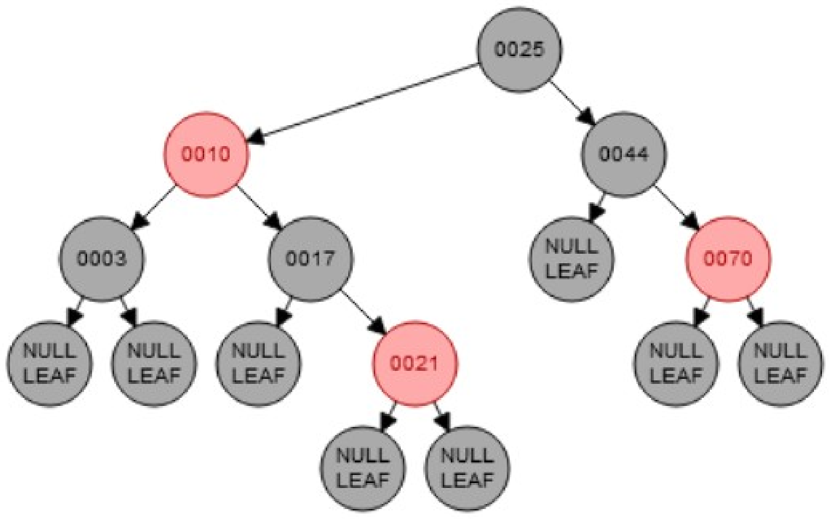
Delete 80: Simple case



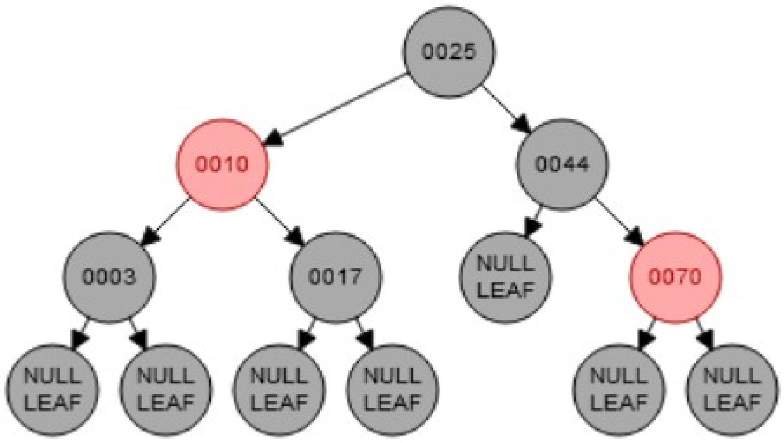
Delete 20: Node 20 will be replaced by predecessor 17 and 20 will be red leaf node, so deleted straight away



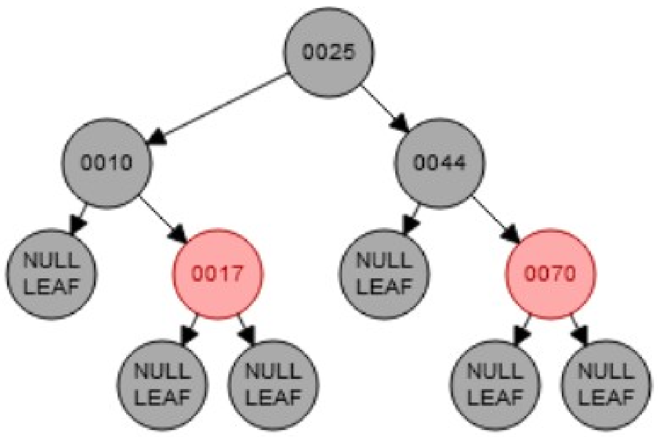
Delete 32: case 3.2.b



Delete 21: Simple case



Delete 3: case 3.2.b



Digital Search Trees

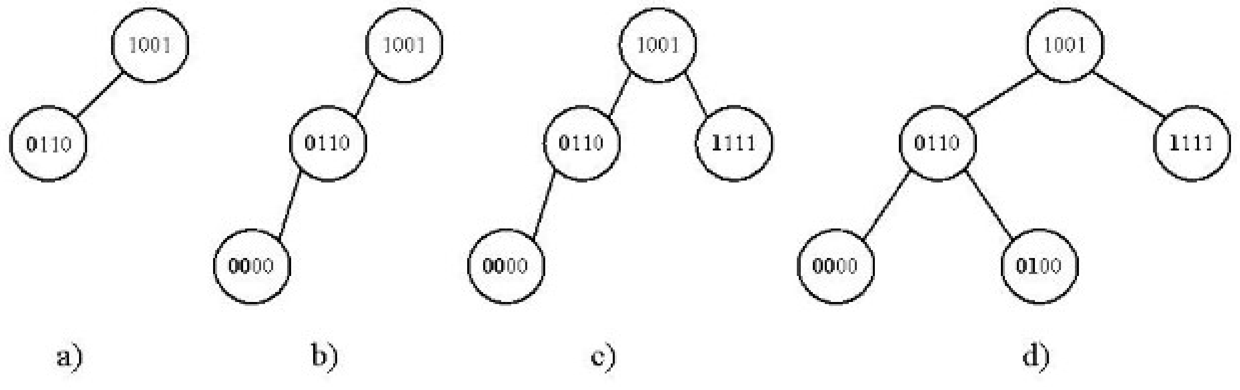
A digital search tree (DST) is a binary tree whose ordering of nodes is based on the values of bits in the binary representation of a node's key (Knuth,1997). The ordering principle is very simple: at each level of the tree a different bit of the key is checked; if the bit is 0, the search continues down the left subtree, if it is 1, the search continues down the right subtree. The search terminates when the corresponding link in the tree is NIL. Every node in a DST holds a key and links to the left and right child, just like in an ordinary binary search tree.

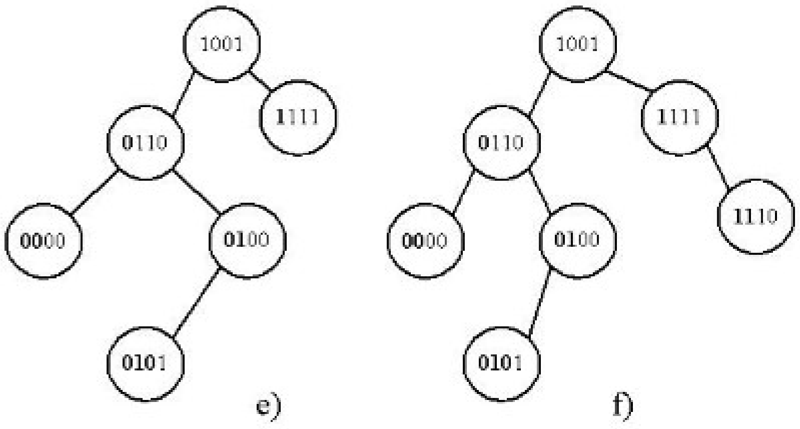
Creating and Searching DST

Let us build a tree with the following elements: 1001, 0110, 0000, 1111, 0100, 0101, and 1110. Since we start with an empty tree, the first element, 1001, becomes the root of the tree. The next key to be inserted is 0110. First, we have to check whether this key is equal to the key of the root node. Since it is not, we continue traversal down the tree. To decide where to place the node, we look at the first bit from the left. Since this bit is 0, we take the path down the left sub- tree and find that the left pointer is NIL. Therefore, we create a new node, which is the left child of the root node, as shown in Figure 1a. The next element to be inserted is 0000. Since the root of the tree is not NIL (and its key is not equal to the key being inserted), we look at the first bit (from the left) of the key to be inserted. Since this bit is 0, we take the path down the left subtree. Since the next node is not NIL either, we look at the next (second) bit of the key to be in- serted. The bit is 0, so we take the path down the left subtree. At this point we encounter a NIL pointer and place the new node in its place, as depicted in Figure 1b. We continue the procedure until all nodes have been added, and the tree in Figure 1f is obtained. Bits used to make the decisions while progressing down the tree are shown in bold in the figure. It is worth noting that scanning the bits from left to right, when making search decisions, is an arbitrary choice. The tree can be built in the same manner by scanning the bits from right to left.

The insertion method for DSTs is nearly identical to the insertion method for binary search trees. We first compare the key to be inserted with the key in the current node, except that for DSTs we test only for equality. In addition, DST algorithms test the corresponding bit of the new key to choose the next move.

To search for an element, we traverse the tree in the same way the insert procedure does. If long the way we find that the current node's key matches the key we are looking for, the search completes success fully. If during the traversal we reach a NIL node, this means that the search key does not exist in the tree.



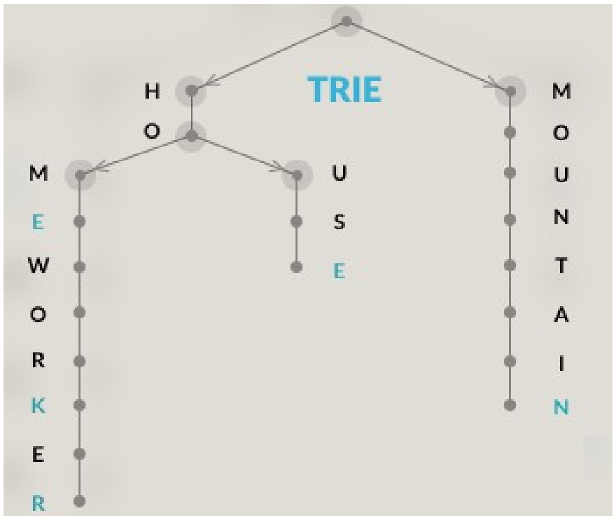


Deleting nodes from DST

Procedure to delete a node in the DST simply replaces the node with any leaf node in either of the node's subtrees. If the node to be deleted does not have any children, it is simply removed and replaced by a NIL pointer. For instance, removing the node 0110 from the tree in Figure 1f can be done by replacing it with either node 0000 or 0101. It is obvious that this method is similar, and in fact simpler than the deletion method for ordinary binary trees, which has to consider three distinct cases

Trie

- Trie is an efficient information re***Trie***val data structure. Using Trie, search complexities can be brought to optimal limit (key length).
- If keys are stored in binary search tree, a well balanced BST will need time proportional to **M * log N**, where M is maximum string length and N is number of keys in tree.
- Using Trie, the key can be searched in O(M) time. However the penalty is on Trie storage requirements.
- Every node of Trie consists of multiple branches. Each branch represents a possible character of keys.
- The last node of every key need to be marked as end of word node. A Trie node field ***isEndOfWord*** is used to distinguish the node as end of word node.

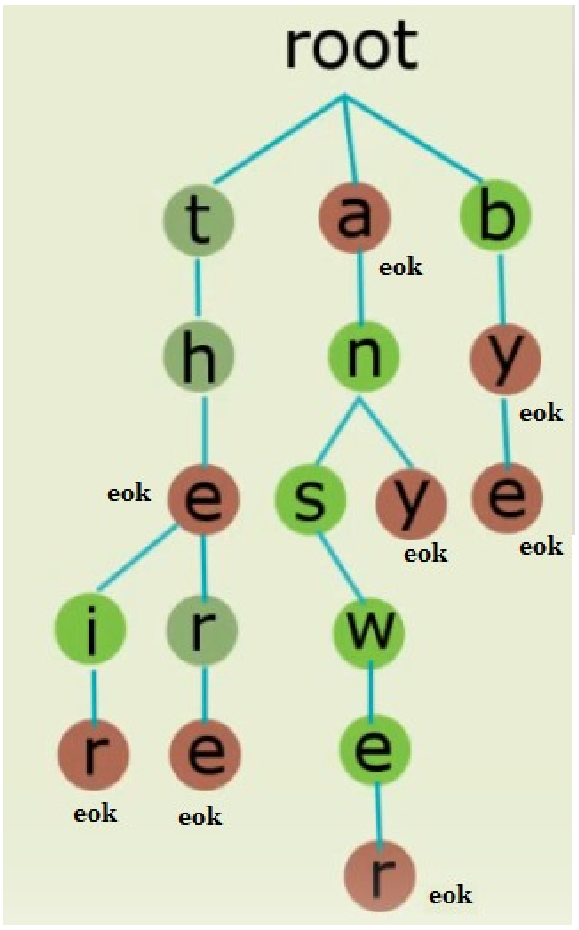


Inserting (Creating) Trie

- Inserting a key into Trie is simple approach. Every character of input key is inserted as an individual Trie node.
- Note that the *children* are pointers (or references) to next level trie nodes. The key character acts as an index into the *children* array.
- If the input key is new or an extension of existing key, it is needed to construct non-existing nodes of the key, and mark end of word for last node.
- If the input key is prefix of existing key in Trie, we simply mark the last node of key as end of word. The key length determines Trie depth.

Searching Trie

- Searching for a key is similar to insert operation, however here only compare the characters and move down.
- The search can terminate due to end of string or lack of key in trie.
- In the former case, if the *isEndofWord* field of last node is true, then the key exists in trie.
- In the second case, the search terminates without examining all the characters of key, since the key is not present in trie.


root
t
a
b
eok
h
n
y
eok
eok
e
s
y
e
eok
eok
i
r
w
r
e
e
eok
eok
r
eok