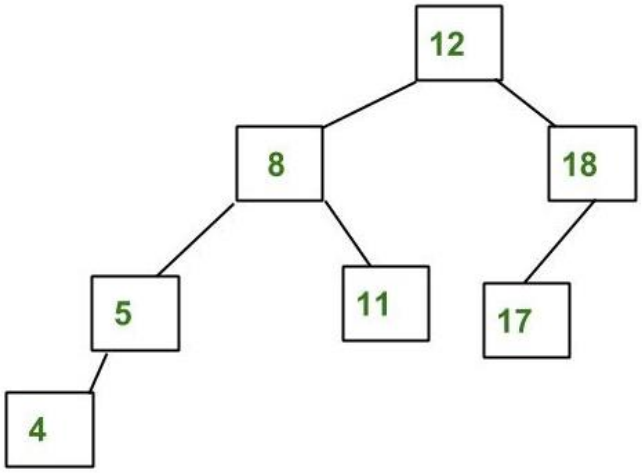
AVL Tree

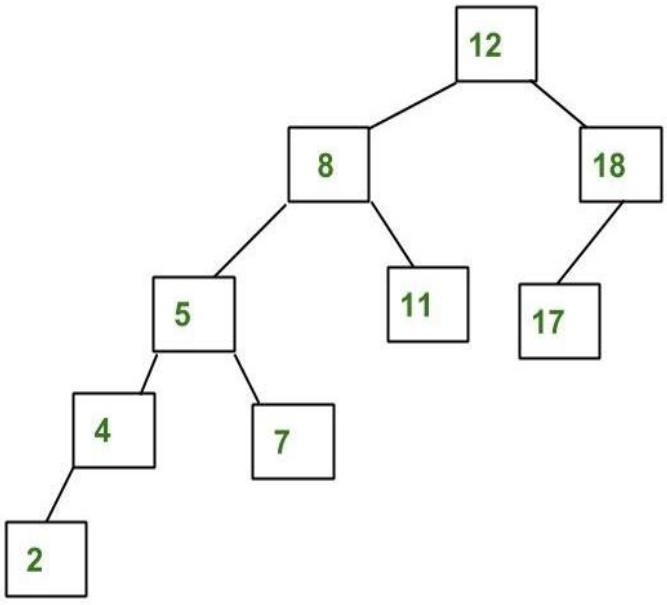
AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.

An Example Tree that is an AVL Tree



The above tree is AVL because differences between heights of left and right subtrees for every node is less than or equal to 1.

An Example Tree that is NOT an AVL Tree



The above tree is not AVL because differences between heights of left and right subtrees for 8 and 18 is greater than 1.

Why AVL Trees?
Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where

h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that height of the tree remains O(Logn) after every insertion and deletion, then we can guarantee an upper bound of O(Logn) for all these operations. The height of an AVL tree is always O(Logn) where n is the number of nodes in the tree (See this video lecture for proof).

Insertion

To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property (keys(left) < key(root) < keys(right)).
1) Left Rotation
2) Right Rotation

```
T1, T2 and T3 are subtrees of the tree
rooted with y (on the left side) or x (on
the right side)
     y                               x
    / \     Right Rotation          /  \
   x   T3   - - - - - - - >        T1   y
  / \       < - - - - - - -            / \
 T1  T2     Left Rotation            T2  T3
Keys in both of the above trees follow the
following order
 keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)
So BST property is not violated anywhere.
```

Steps to follow for insertion
Let the newly inserted node be w
1) Perform standard BST insert for w.
2) Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the child of z that comes on the path from w to z and x be the grandchild of z that comes on the path from w to z.
3) Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements:
a) y is left child of z and x is left child of y (Left Left Case)
b) y is left child of z and x is right child of y (Left Right Case)
c) y is right child of z and x is right child of y (Right Right Case)
d) y is right child of z and x is left child of y (Right Left Case)

Following are the operations to be performed in above mentioned 4 cases. In all of the cases, we only need to re-balance the subtree rooted with z and the complete tree becomes balanced as the height of subtree (After appropriate rotations) rooted with z becomes same as it was before insertion. (See this video lecture for proof)

a) Left Left Case

```
T1, T2, T3 and T4 are subtrees.
         z                                      y
        / \                                   /   \
       y   T4      Right Rotate (z)          x      z
      / \          - - - - - - - - ->       /  \    /  \
     x   T3                                T1  T2  T3  T4
    / \
  T1   T2
```

b) Left Right Case

```
     z                                z                               x
    / \                              /   \                           /  \
   y   T4  Left Rotate (y)          x    T4  Right Rotate(z)        y      z
  / \      - - - - - - - - - ->     /  \      - - - - - - - - ->   / \    / \
T1   x                           y    T3                        T1  T2 T3  T4
    / \                         / \
  T2   T3                     T1   T2
```

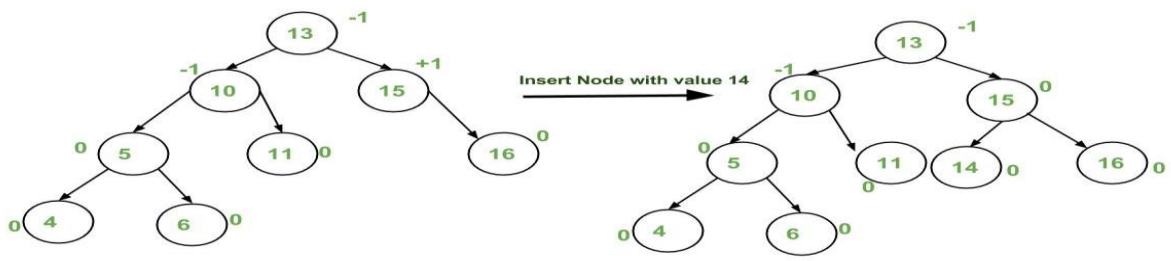
c) Right Right Case

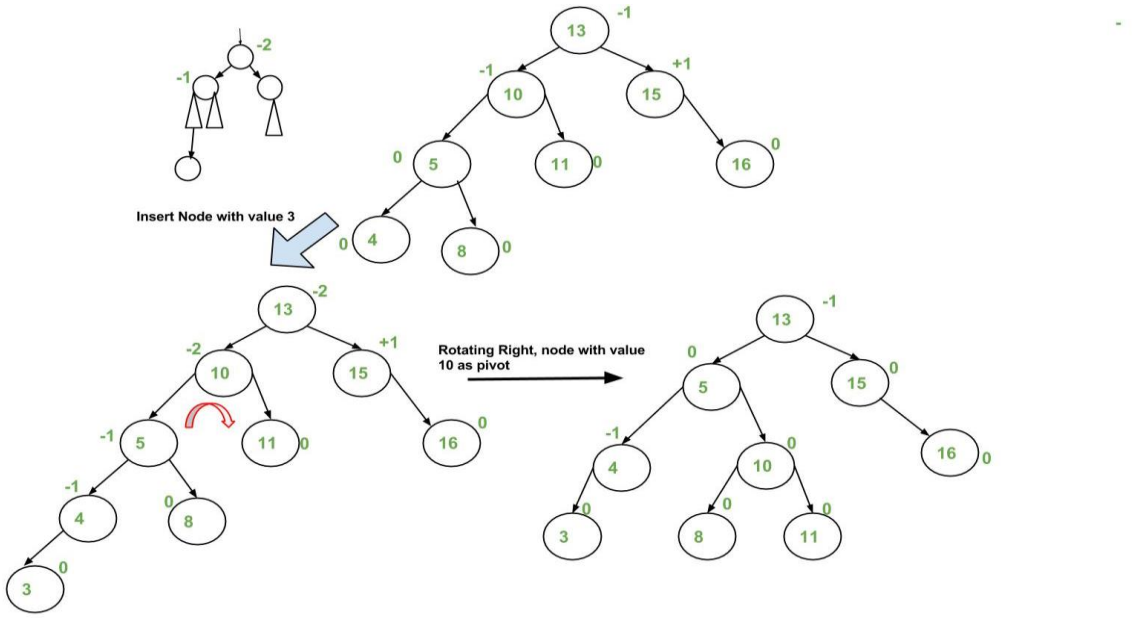
```
  z                                 y
 /  \                             /   \
T1   y     Left Rotate(z)        z      x
    /  \   - - - - - - - - ->    / \    / \
   T2   x                      T1  T2 T3  T4
       / \
     T3  T4
```

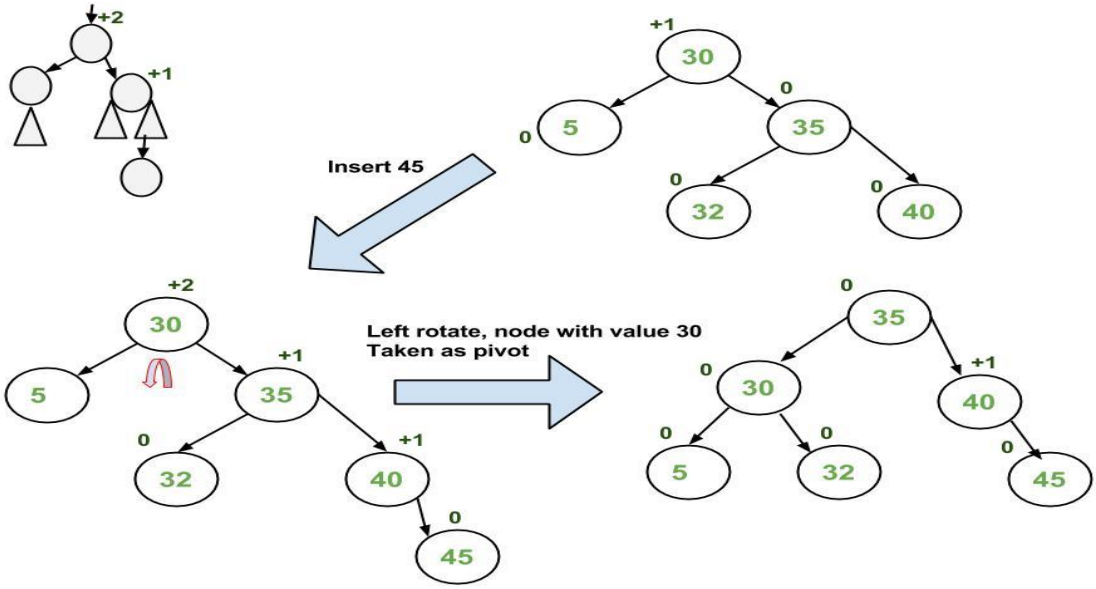
d) Right Left Case

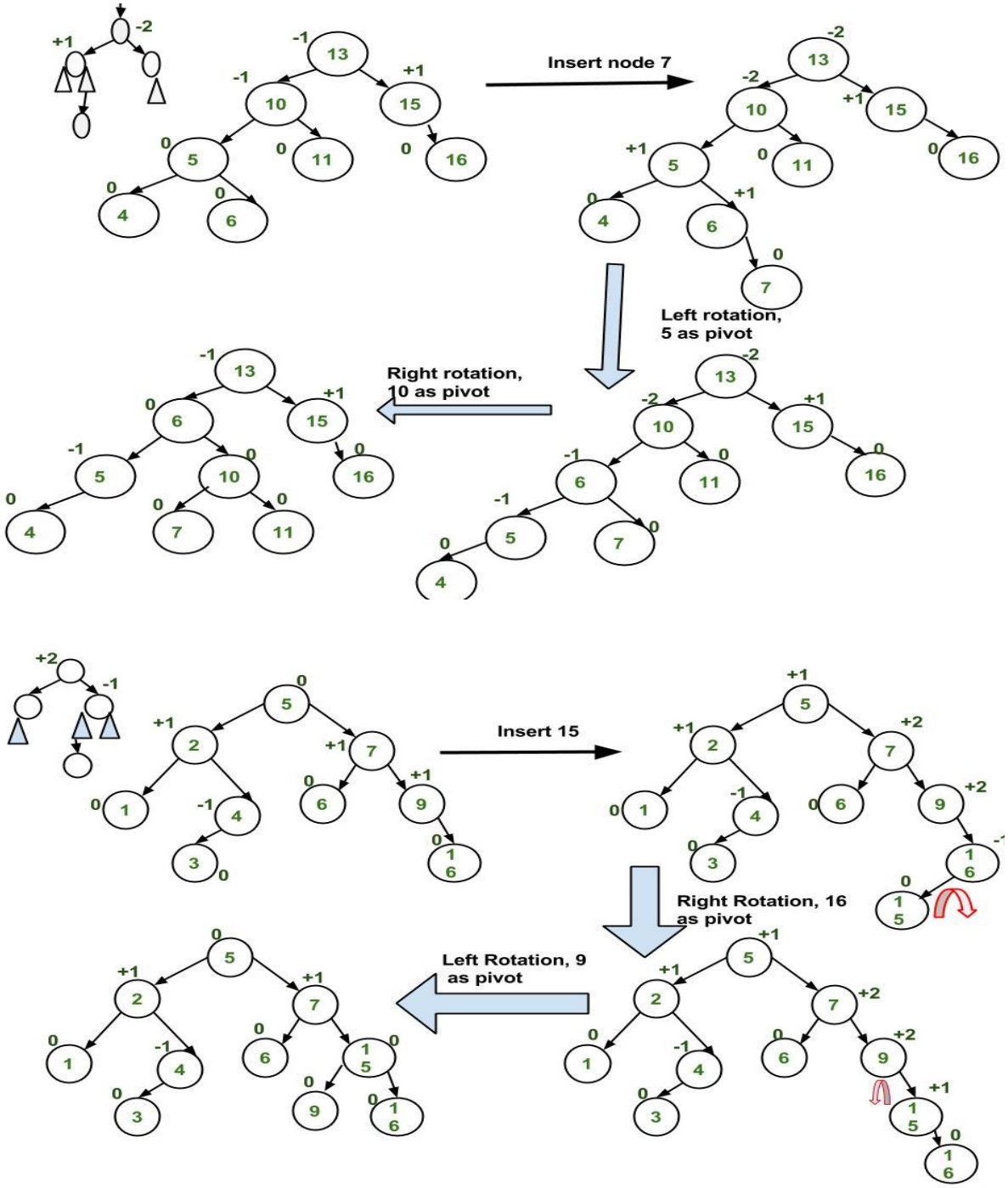
```
   z                                z                               x
  / \                              / \                             /  \
T1   y   Right Rotate (y)        T1   x      Left Rotate(z)      z      y
    / \  - - - - - - - - - ->        /  \    - - - - - - - - ->  / \    / \
   x   T4                          T2   y                     T1  T2  T3  T4
  / \                                  /  \
T2   T3                              T3   T4
```

Insertion Examples:




Insert Node with value 3
Rotating Right, node with value
10 as pivot


Insert 45
Left rotate, node with value 30
Taken as pivot



Implementation

Following is the implementation for AVL Tree Insertion. The following implementation uses the recursive BST insert to insert a new node. In the recursive BST insert, after insertion, we get pointers to all ancestors one by one in a bottom-up manner. So we don't need parent pointer to travel up. The recursive code itself travels up and visits all the ancestors of the newly inserted node.

1) Perform the normal BST insertion.

2) The current node must be one of the ancestors of the newly inserted node. Update the height of the current node.

3) Get the balance factor (left subtree height – right subtree height) of the current node.

4) If balance factor is greater than 1, then the current node is unbalanced and we are either in Left Left case or left Right case. To check whether it is left left case or not, compare the newly inserted key with the key in left subtree root.
5) If balance factor is less than -1, then the current node is unbalanced and we are either in Right Right case or Right-Left case. To check whether it is Right Right case or not, compare the newly inserted key with the key in right subtree root.

Time Complexity: The rotation operations (left and right rotate) take constant time as only a few pointers are being changed there. Updating the height and getting the balance factor also takes constant time. So the time complexity of AVL insert remains same as BST insert which is O(h) where h is the height of the tree. Since AVL tree is balanced, the height is O(Logn). So time complexity of AVL insert is O(Logn).

Comparison with Red Black Tree
The AVL tree and other self-balancing search trees like Red Black are useful to get all basic operations done in O(log n) time. The AVL trees are more balanced compared to Red-Black Trees, but they may cause more rotations during insertion and deletion. So if your application involves many frequent insertions and deletions, then Red Black trees should be preferred. And if the insertions and deletions are less frequent and search is the more frequent operation, then AVL tree should be preferred over Red Black Tree.

Deletion

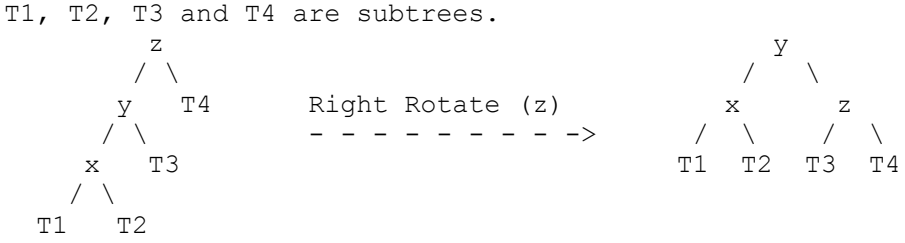
Steps to follow for deletion.
To make sure that the given tree remains AVL after every deletion, we must augment the standard BST delete operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property (keys(left) < key(root) < keys(right)).
1) Left Rotation
2) Right Rotation

```
T1, T2 and T3 are subtrees of the tree rooted with y (on left side)
or x (on right side)
                y                               x
               / \     Right Rotation          /  \
              x   T3   - - - - - - - >        T1   y
             / \       < - - - - - - -            / \
            T1  T2     Left Rotation            T2  T3
Keys in both of the above trees follow the following order
      keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)
So BST property is not violated anywhere.
```

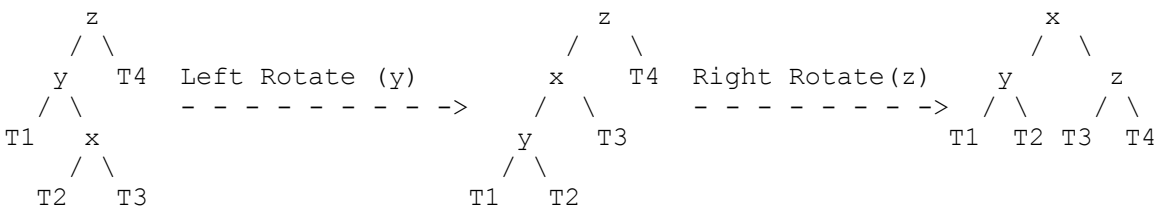
Let w be the node to be deleted
1) Perform standard BST delete for w.
2) Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the larger height child of z, and x be the larger height child of y. Note that the definitions of x and y are different from insertion here.
3) Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements:
a) y is left child of z and x is left child of y (Left Left Case)
b) y is left child of z and x is right child of y (Left Right Case)
c) y is right child of z and x is right child of y (Right Right Case)
d) y is right child of z and x is left child of y (Right Left Case)

Like insertion, following are the operations to be performed in above mentioned 4 cases. Note that, unlike insertion, fixing the node z won't fix the complete AVL tree. After fixing z, we may have to fix ancestors of z as well

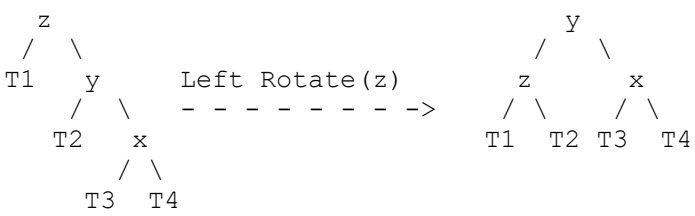
a) Left Left Case



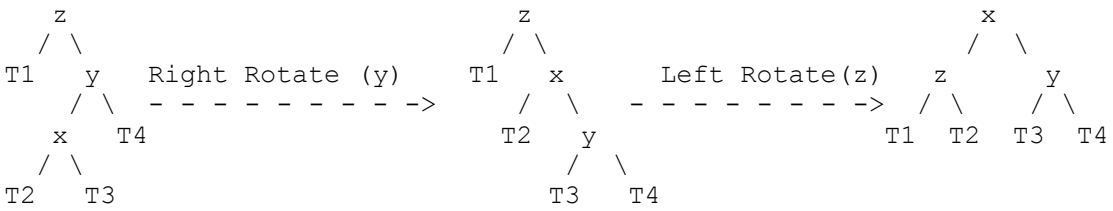
b) Left Right Case



c) Right Right Case

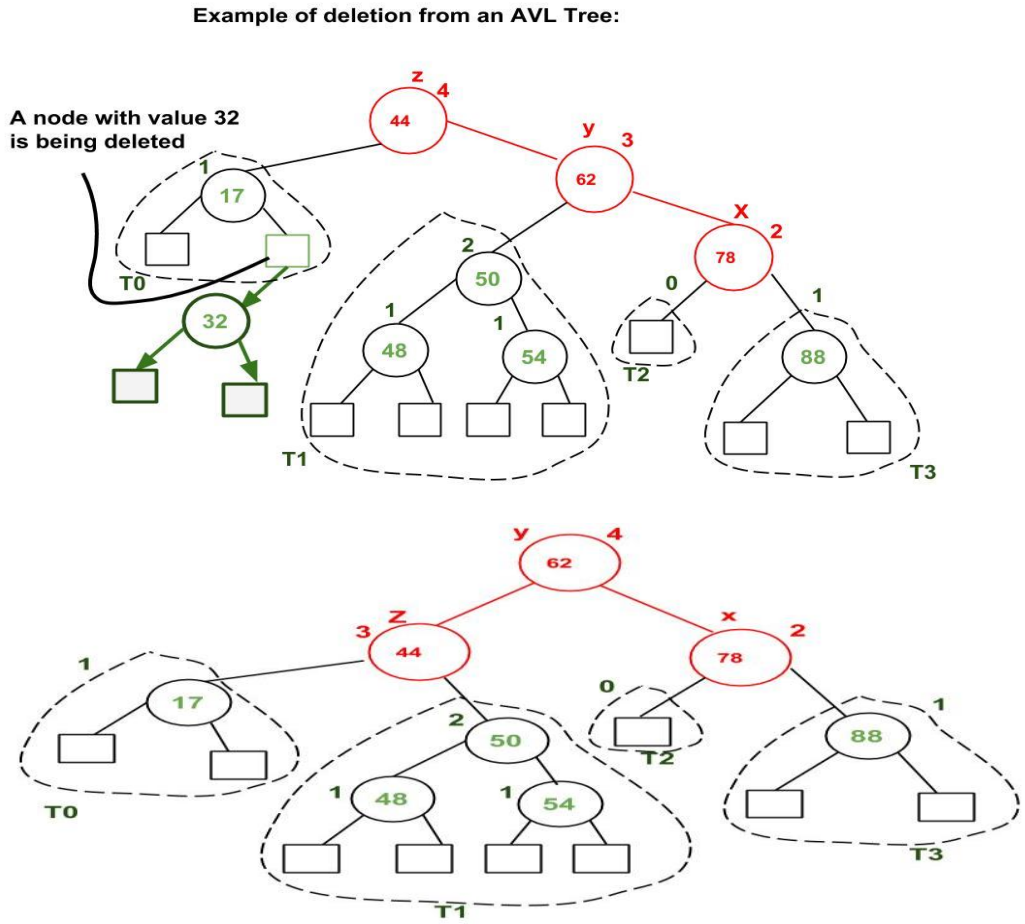


d) Right Left Case



Unlike insertion, in deletion, after we perform a rotation at z, we may have to perform a rotation at ancestors of z. Thus, we must continue to trace the path until we reach the root.

Example:



A node with value 32 is being deleted. After deleting 32, we travel up and find the first unbalanaced node which is 44. We mark it as z, its higher height child as y which is 62, and y's higher height child as x which could be either 78 or 50 as both are of same height. We have considered 78. Now the case is Right Right, so we perform left rotation.

C++implementation

Following is the C++ implementation for AVL Tree Deletion. The following C++implementation uses the recursive BST delete as basis. In the recursive BST delete, after deletion, we get pointers to all ancestors one by one in bottom up manner. So we don't need parent pointer to travel up. The recursive code itself travels up and visits all the ancestors of

the deleted node.
1) Perform the normal BST deletion.
2) The current node must be one of the ancestors of the deleted node. Update the height of the current node.
3) Get the balance factor (left subtree height – right subtree height) of the current node.
4) If balance factor is greater than 1, then the current node is unbalanced and we are either in Left Left case or Left Right case. To check whether it is Left Left case or Left Right case, get the balance factor of left subtree. If balance factor of the left subtree is greater than or equal to 0, then it is Left Left case, else Left Right case.
5) If balance factor is less than -1, then the current node is unbalanced and we are either in Right Right case or Right Left case. To check whether it is Right Right case or Right Left case, get the balance factor of right subtree. If the balance factor of the right subtree is smaller than or equal to 0, then it is Right Right case, else Right Left case.

Time Complexity: The rotation operations (left and right rotate) take constant time as only few pointers are being changed there. Updating the height and getting the balance factor also take constant time. So the time complexity of AVL delete remains same as BST delete which is O(h) where h is height of the tree. Since AVL tree is balanced, the height is O(Logn). So time complexity of AVL delete is O(Log n).

SOME IMPORTANT THEORY:

AVL trees are often compared with red–black trees because both support the same set of operations and take time for the basic operations. For lookup-intensive applications, AVL trees are faster than red–black trees because they are more strictly balanced.[4] Similar to red–black trees, AVL trees are height-balanced.

Balance factor

In a binary tree the *balance factor* of a node is defined to be the height difference

of its two child sub-trees. A binary tree is defined to be an *AVL tree* if the invariant holds for every node in the tree. A node with is called "left-heavy", one with is called "right-heavy", and one with is sometimes simply called "balanced".

APPLICATIONS OF AVL TREES

1) Trains in a railway system.

AVL trees are beneficial in the cases where you are designing some database where insertions and deletions are not that frequent but you have to frequently look-up for the items present in there.

2) Linux Completely Fair Scheduler

3)Java Hashmap

Advantages of an AVL tree:

AVL Trees are self- balancing Binary Search Trees (BSTs). A normal BST may be skewed to either side which will result in a much greater effort to search for a key (the order will be much more than $O(\log 2n)$) and sometimes equal $O(n)$) at times giving a worst case result which is the same effort spent in sequential searching. This makes BSTs useless as a practically efficient Data Structure.

AVL Tree(Hight - Balance Tree)

It is nothing but a *Binary Search Tree* (BST) with balance factor.

Balance Factor can be -1, 0, 1.

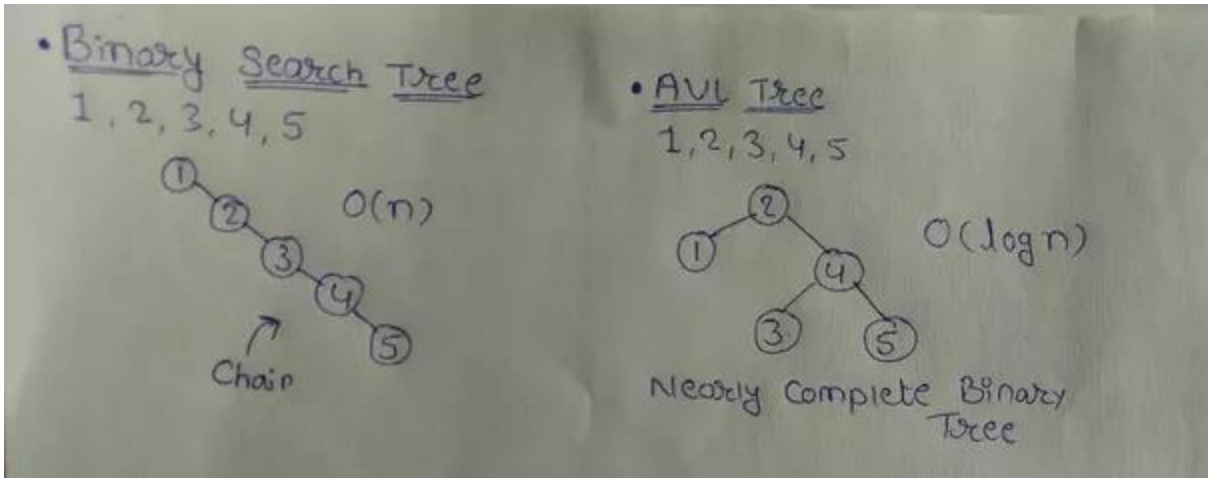
Importance of AVL tree

Searching can be done by *Binary Search Tree* (BST) but if element are already sorted BST take form of chain and time complexity is O(n).

To overcome this problem AVL tree is introduced.

Due to balance factor it never take a form of chain, in fact AVL tree is nearly complete Binary tree.

Operations like Insertion, Deletion, Searching in a tree take O(log n) Time in worst case and Average case.



Disadvantages of AVL trees:

- The code for AVL tree is much more complex than the code for BST and we have to handle a lot of corner cases.
- Since the height has to be maintained in AVL trees frequent rotations are done which inturn will increase the pointer overhead which even tends to some negligiable extra space for very big inputs.
- Deletion operations cost high in AVL trees as they involve a lot of pointer changes and rotations.

So AVL trees can be implemented at the cost of some extra space and much more complex code when compared to the traditional BSTs. Red black trees can be preferred as they have much less pointer overhead but still the code for Red Black trees is very complex and understanding and implementing that is indeed a tedious job.

1. Slow Inserts and Deletes
The largest disadvantage to using an AVL tree is the fact that in the event that it is slightly unbalanced it can severely impact the amount of time that it takes to perform inserts and deletes. Computer science professionals find that since AVL trees don't allow anything outside of -1 to 1, it can drastically slow these 2 functions down.

2. Fast Updating Systems
In the event that you're working on a system that has the tendency to update quickly then it is advisable to avoid AVL trees. This is because they work much better with systems that are

rarely updated, giving the system time to acquire all of the information that it needs to perform at its best. If you are working with controlled data, an AVL tree is your best option.

What is the difference between a Binary tree, and an AVL tree?

AVL tree is a self balancing BST. Which means it will undergo alterations (LL,LR,RL,RR roatations here) when a new node is added which does not comply with its self balancing property.

The self balancing property is - if the difference in height of left subtree and right subtree of a node is greater than one, then it is made to be zero or equal to one by undergoing alterations/rotations.

Binary tree on the other hand does not have any such property and can be unbalanced or skewed.

So to summarize it :

Binary tree + BST property + self balancing property = AVL tree

BST property - >every element on left subtree of a node should be smaller than and every element on rhe right subtree of a node should be greater than the element on that node. And this property is respected when adding nodes.

END