AVL TREES

Background

So far …

- Binary search trees store linearly ordered data
- Best case height: $\Theta(\ln(n))$
- Worst case height: $\mathbf{O}(n)$

Requirement:

- Define and maintain a *balance* to ensure $\Theta(\ln(n))$ height

Recall:

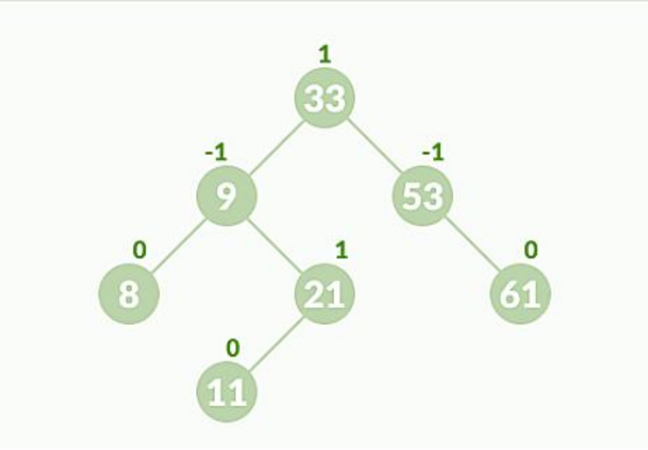
- An empty tree has height -1
- A tree with a single node has height 0

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.

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:

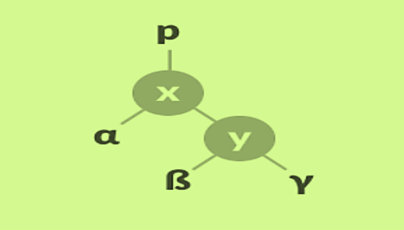


Operations on AVL Tree

- Insert
- Delete
- Search
- Rotate
  - Left Rotate
  - Right Rotate
  - Left-Right Rotate
  - Right-Left Rotate

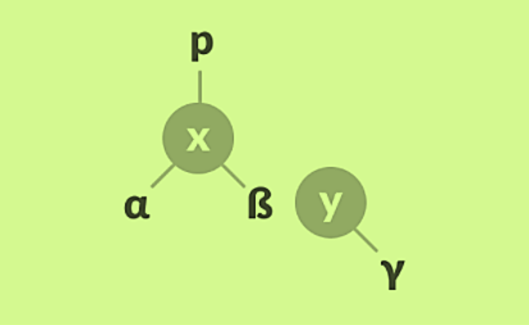
Left Rotate

- 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

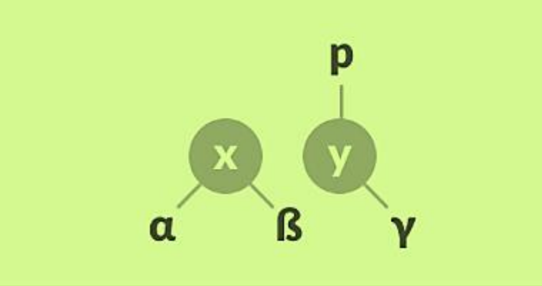
  2. If y has a left subtree, assign x as the parent of the left subtree of y



Assign x as the parent of the left subtree of y

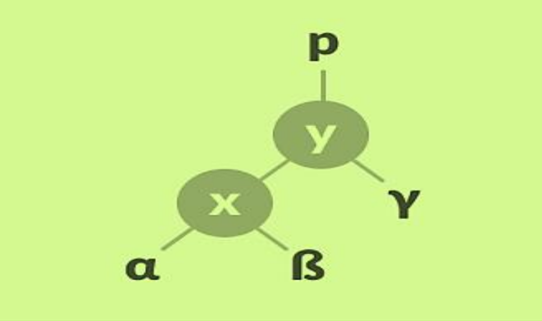
Left Rotate

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

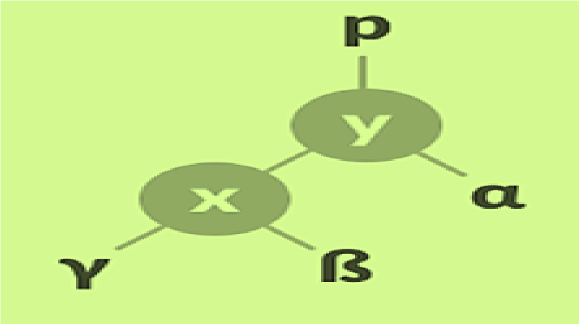


Assign y as the parent of x.

Right Rotate

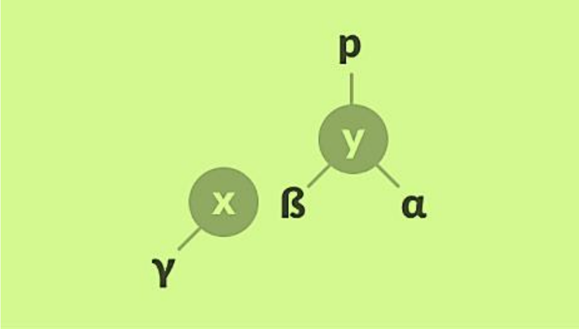
- In right-rotation, the arrangement of the nodes on the left is transformed into the arrangements on the right node.
- Algorithm:

1. Let the initial tree be:



Initial tree

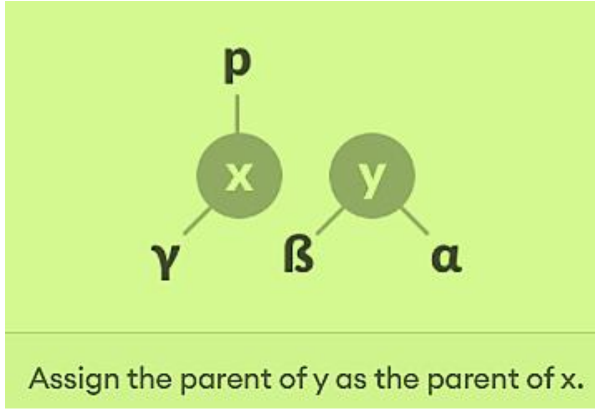
2. If x has a right subtree, assign y as the parent of the right subtree of x



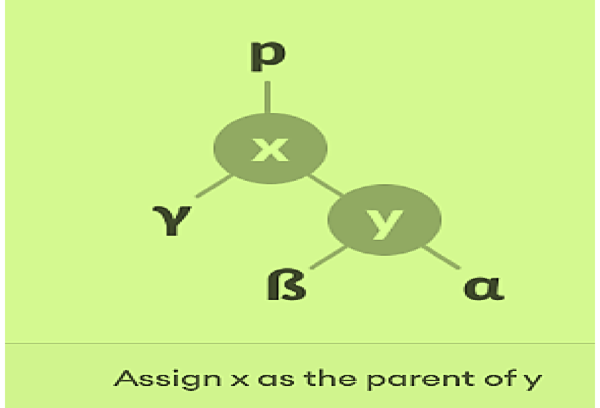
Assign y as the parent of the right subtree of x

Right Rotate

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



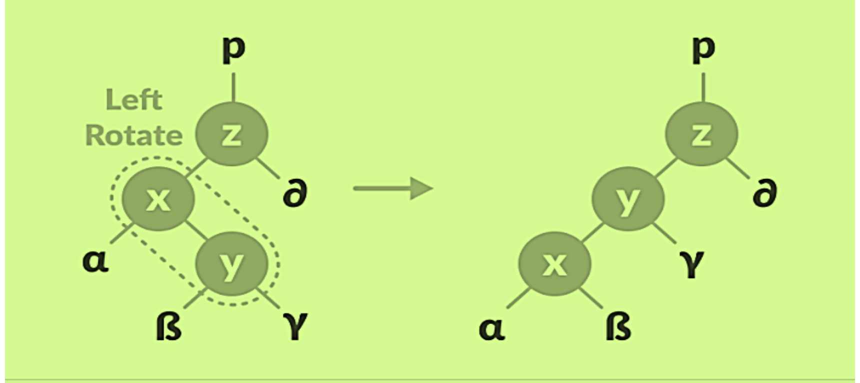
6. Make x as the parent of y



Left -Right Rotate

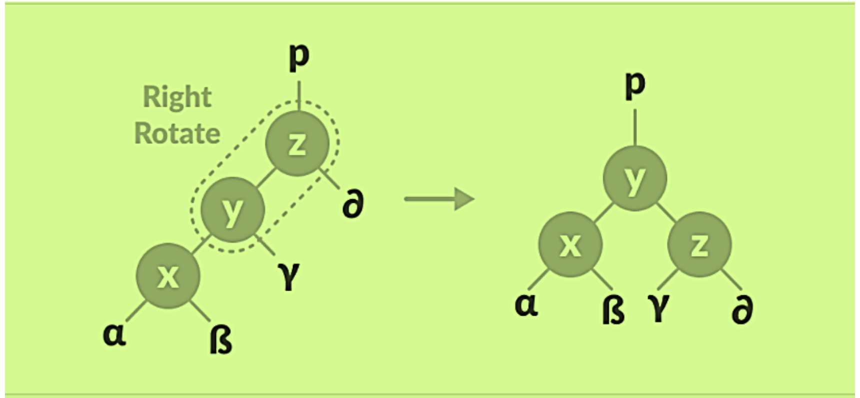
- In left-right rotation, the arrangements are first shifted to the left and then to the right.
- Algorithm:

1. Do left rotation on x-y.



Left rotate x-y

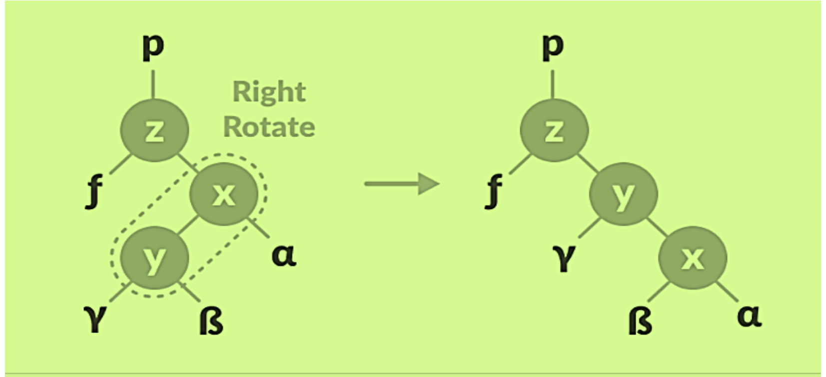
2. Do right rotation on y-z.



Right rotate z-y

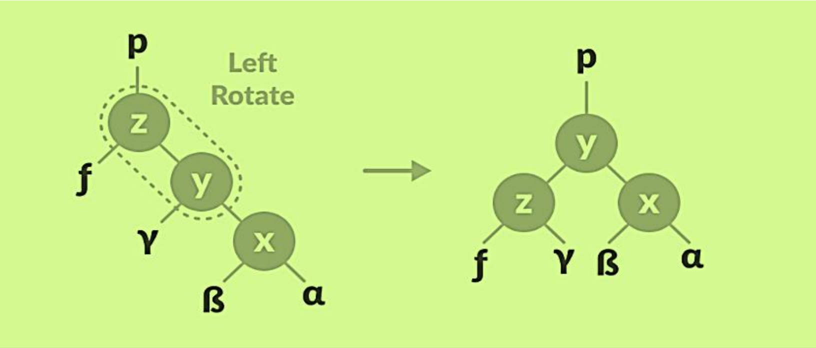
Right-Left Rotate

- In right-left rotation, the arrangements are first shifted to the right and then to the left.
- Algorithm:
  1. Do right rotation on x-y.



Right rotate x-y

  2. Do left rotation on z-y.

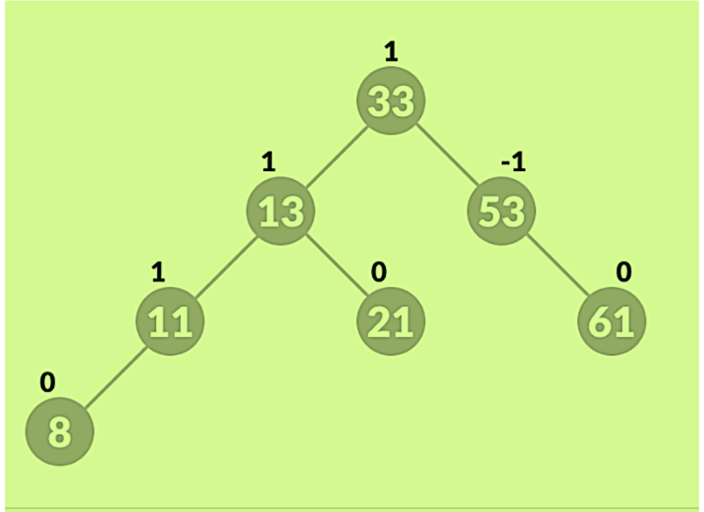


Left rotate z-y

Algorithm to insert a newNode

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

1. Let the initial tree be:



Initial tree for insertion

2. Let the node to be inserted be:

New node

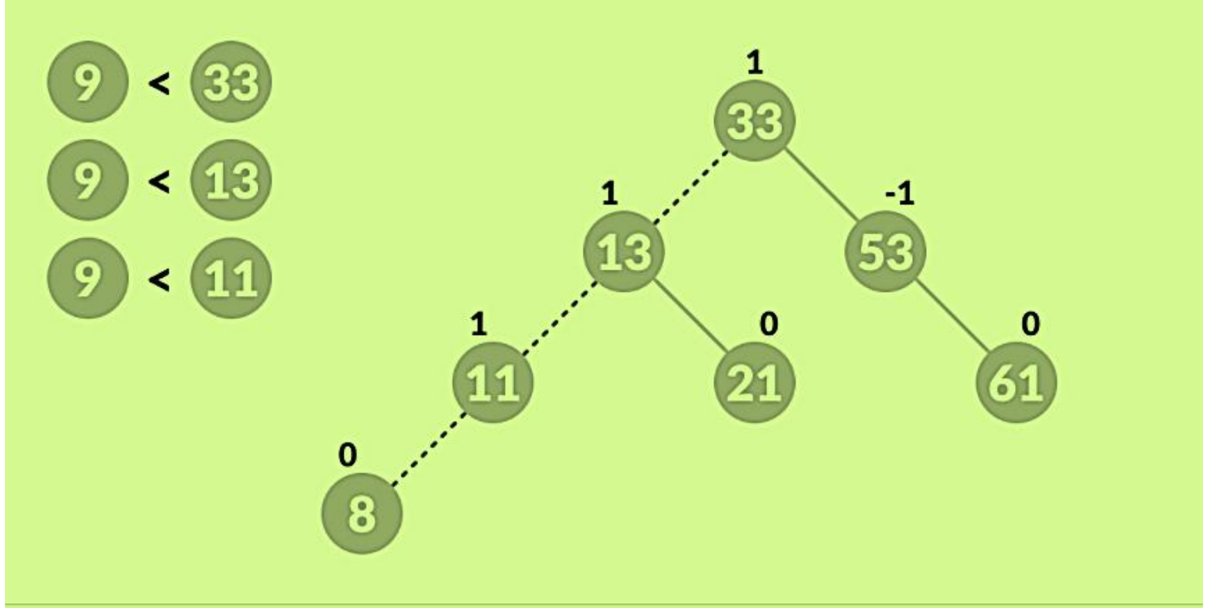
AVL Tree Insertion

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

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

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

   3.3 Else, return leafNode



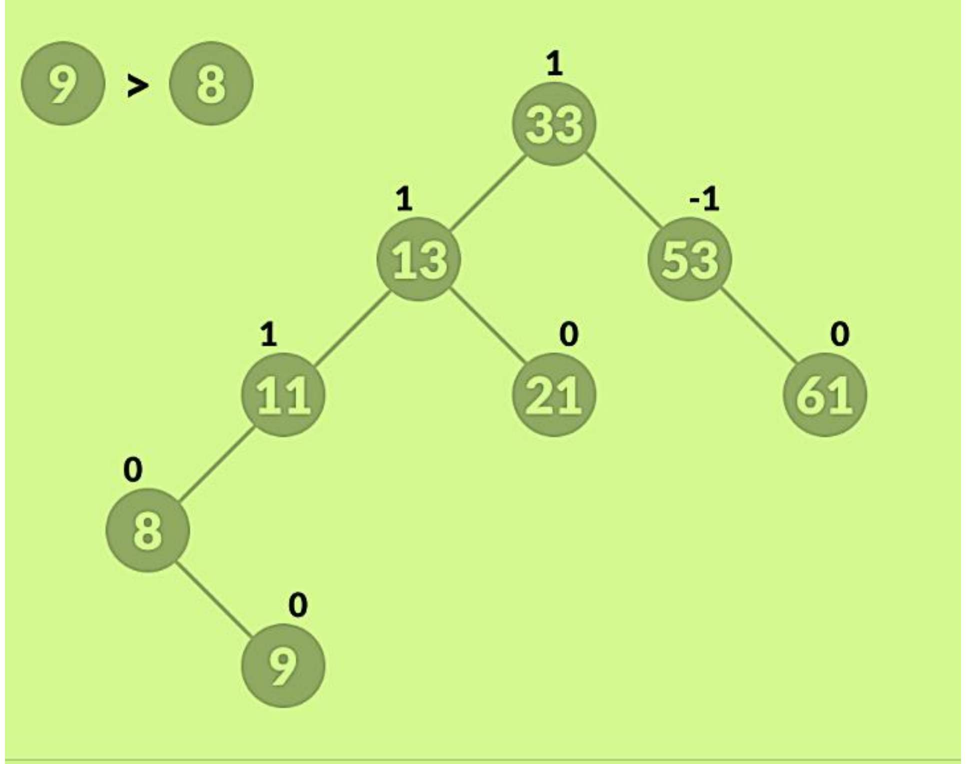
Finding the location to insert newNode

AVL Tree Insertion

4. Compare leafKey obtained from the above steps with newKey:

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

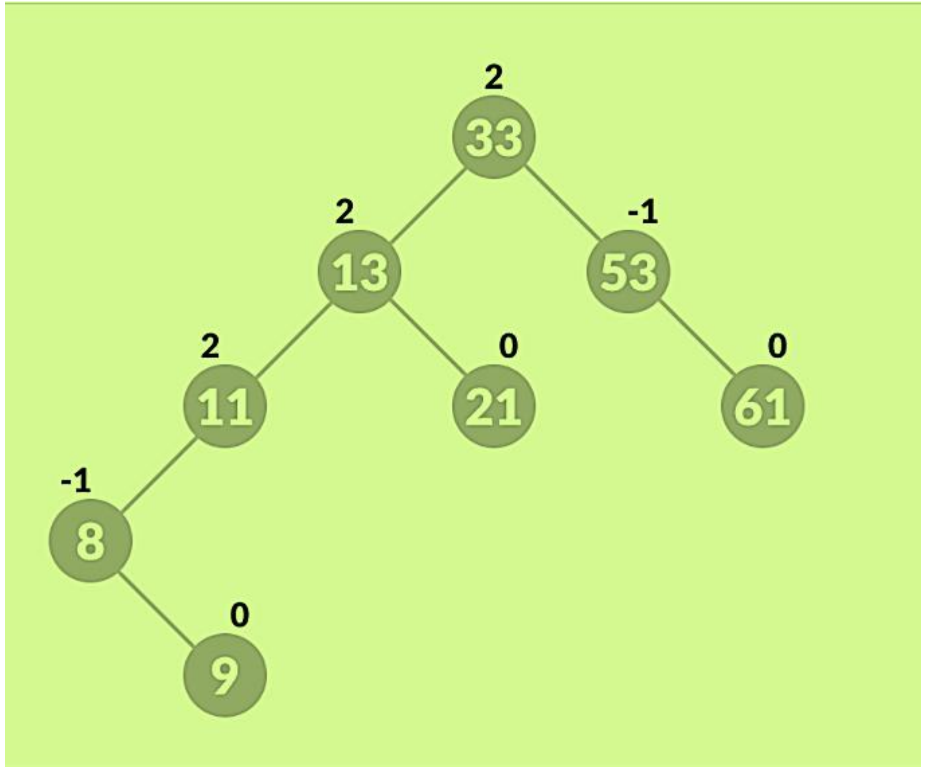
    4.2 Else, make newNode as rightChild of leafNode



Inserting the new node

AVL Tree Insertion

5. Update balanceFactor of the nodes.



Updating the balance factor after insertion

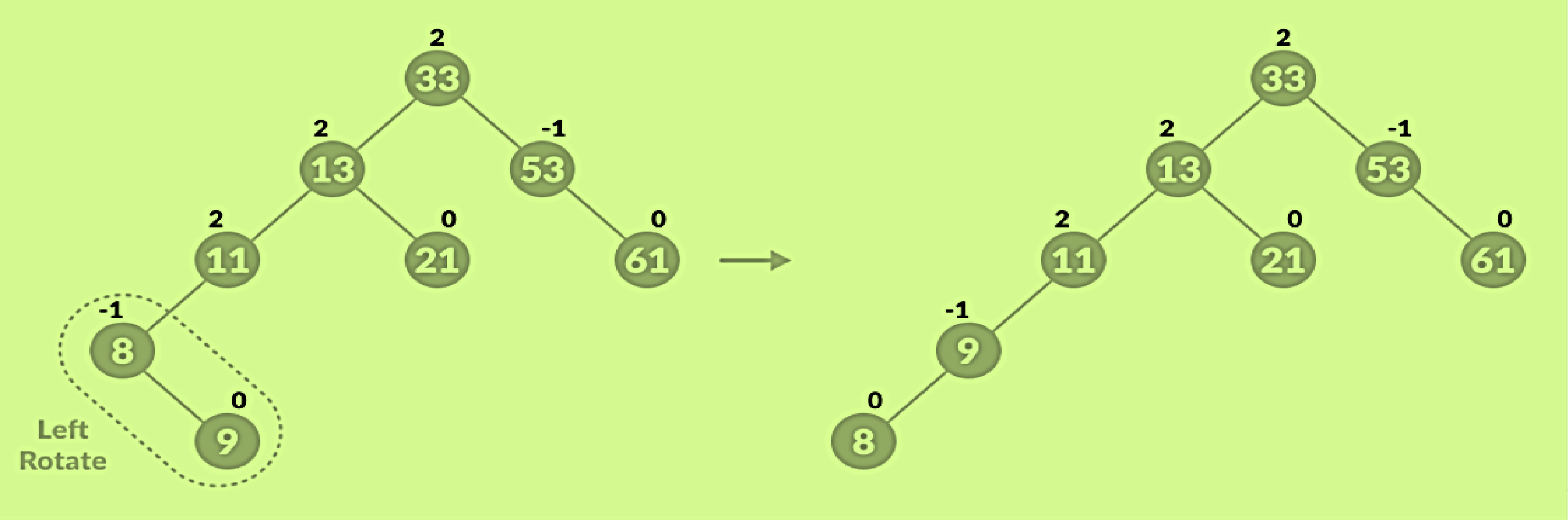
AVL Tree Insertion

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

   6.1 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

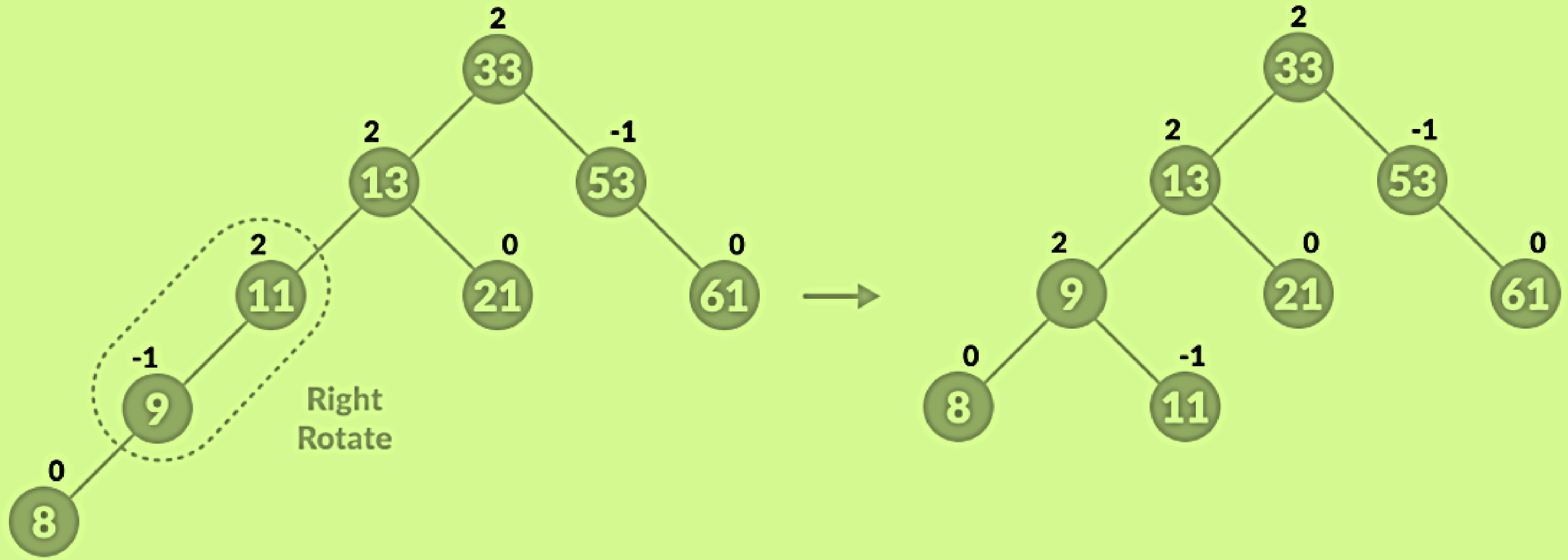
   6.1.1 If newNodeKey < leftChildKey do right rotation.

   6.1.2 Else, do left-right rotation.



Balancing the tree with rotation

AVL Tree Insertion



Balancing the tree with rotation

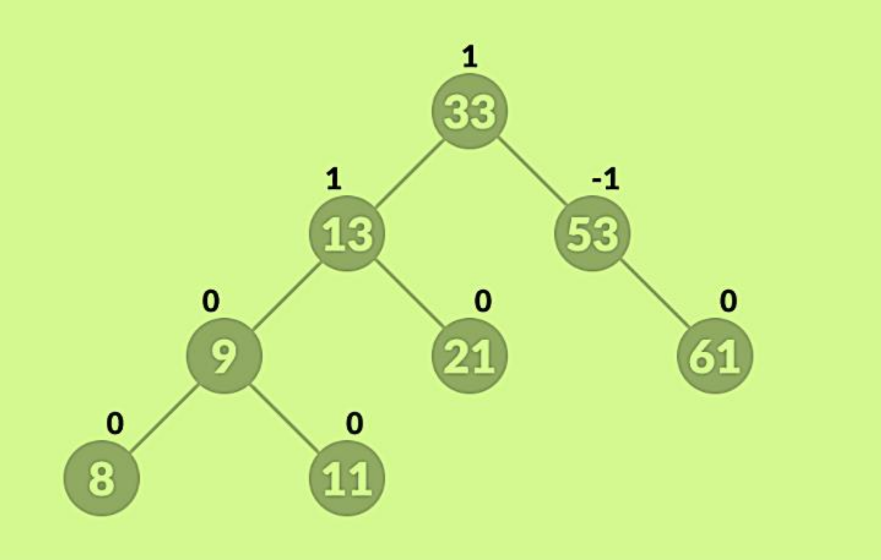
AVL Tree Insertion

6.2 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

6.2.1 If newNodeKey > rightChildKey do left rotation.

6.2.2 Else, do right-left rotation

7. The final Tree is :

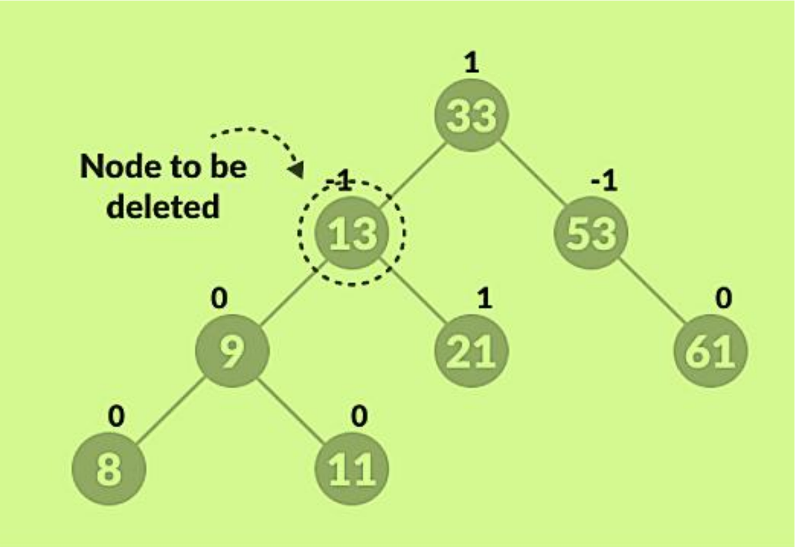


Final balanced tree

Algorithm to Delete a 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 .



Locating the node to be deleted

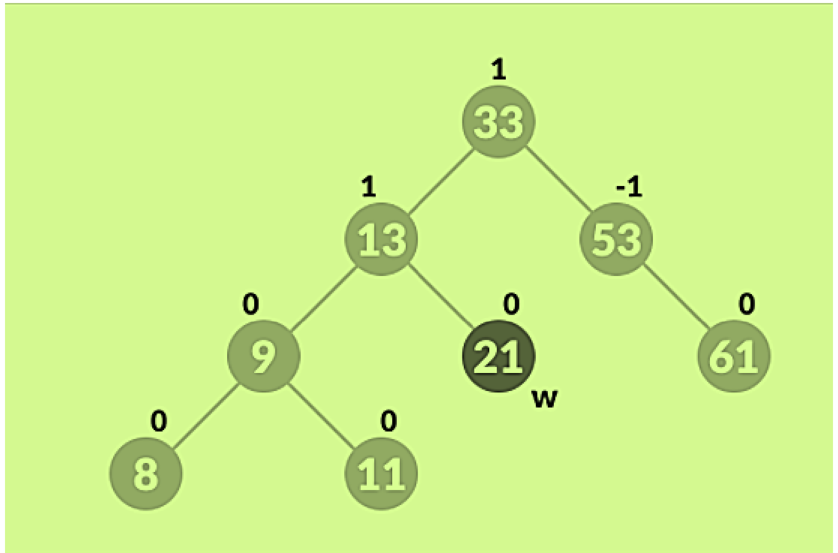
AVL Tree Deletion

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.

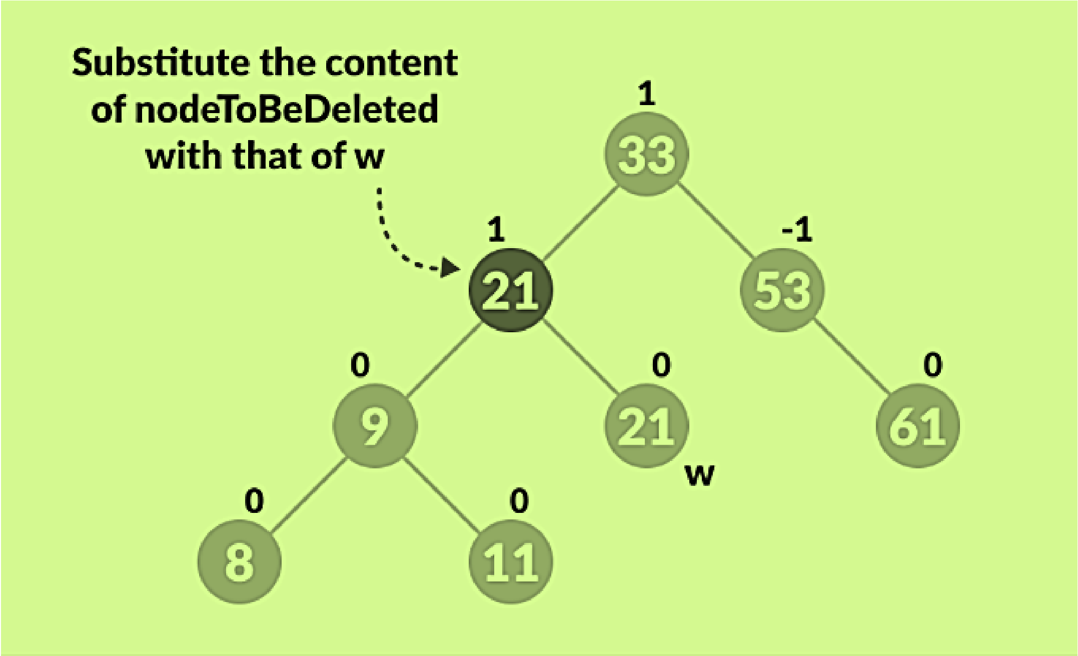
c. If nodeToBeDeleted has two children, find the inorder successor **w** of nodeToBeDeleted (ie. node with a minimum value of key in the right subtree).



Finding the successor

AVL Tree Deletion

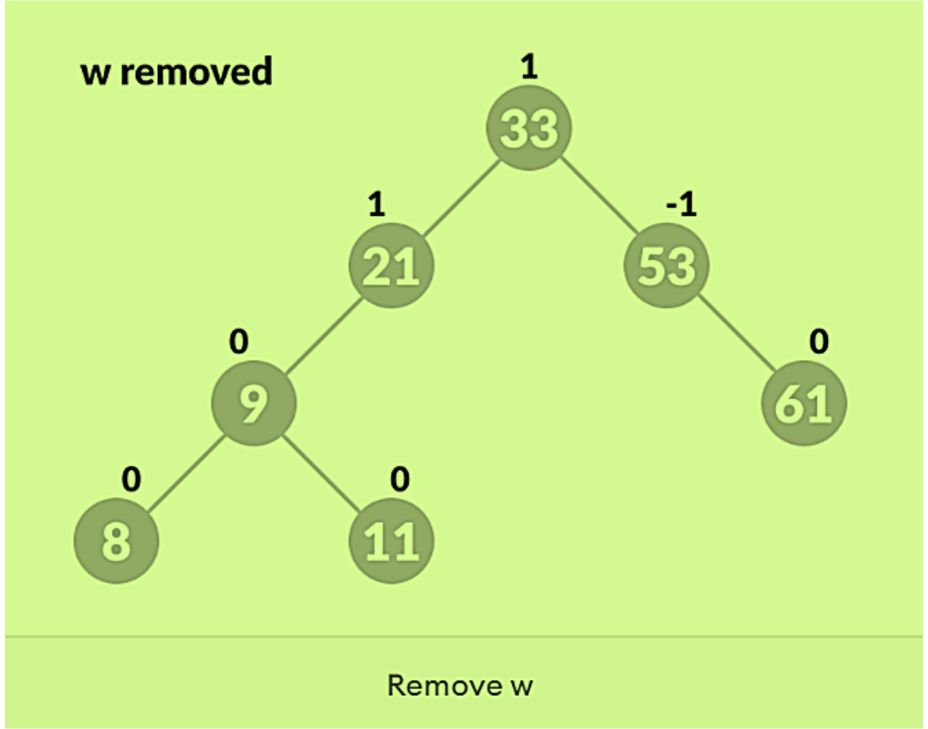
3. Substitute the contents of nodeToBeDeleted with that of **w**



Substitute the node to be deleted

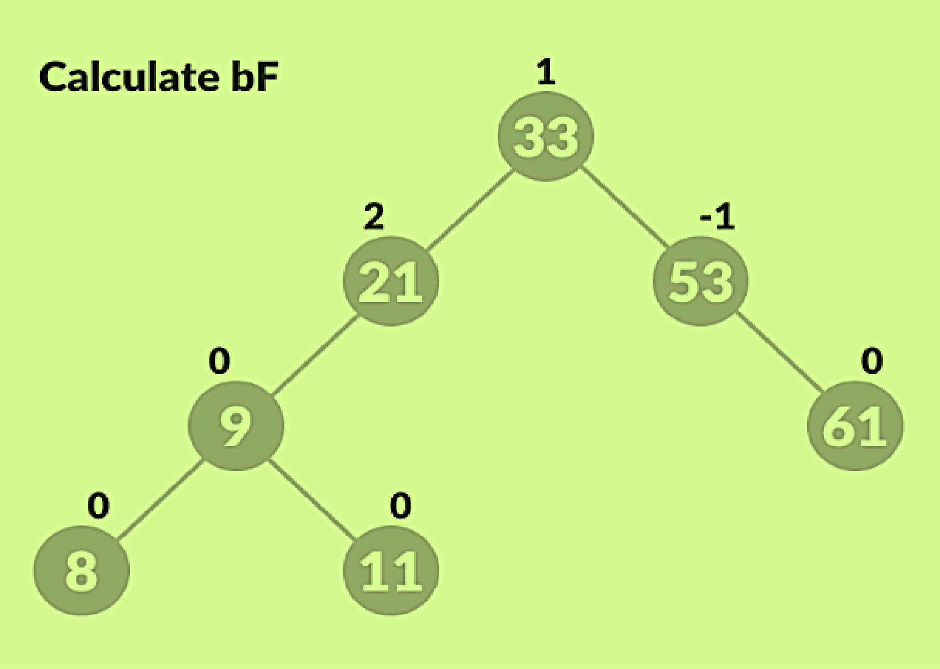
AVL Tree Deletion

4. Remove the leaf node **w**.



AVL Tree Deletion

5. Update balanceFactor of the nodes.



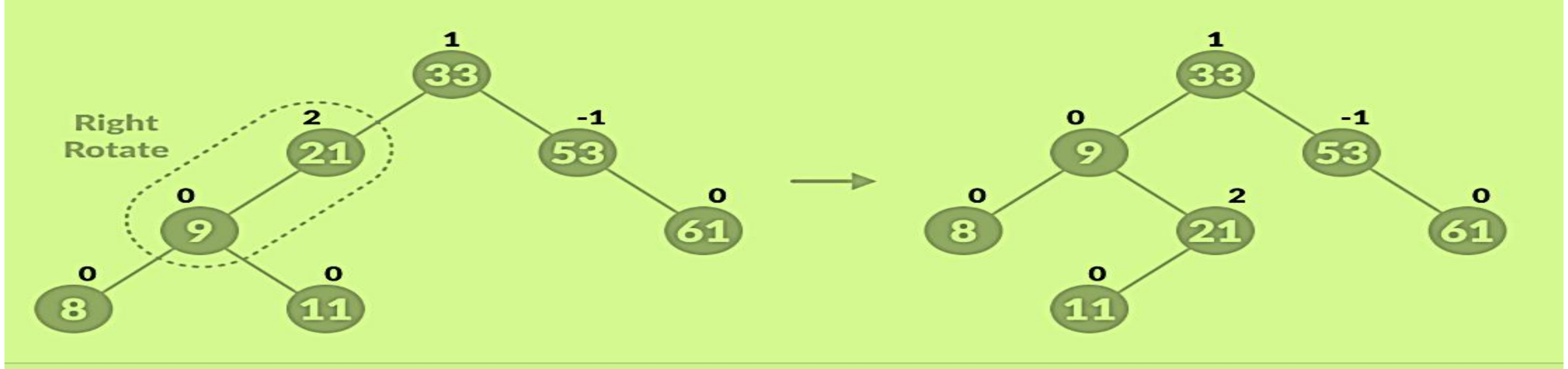
Update bf

AVL Tree Deletion

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

6.1 If balanceFactor of currentNode > 1,

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



Right-rotate for balancing the tree

6.1.2 Else do left-right rotation.

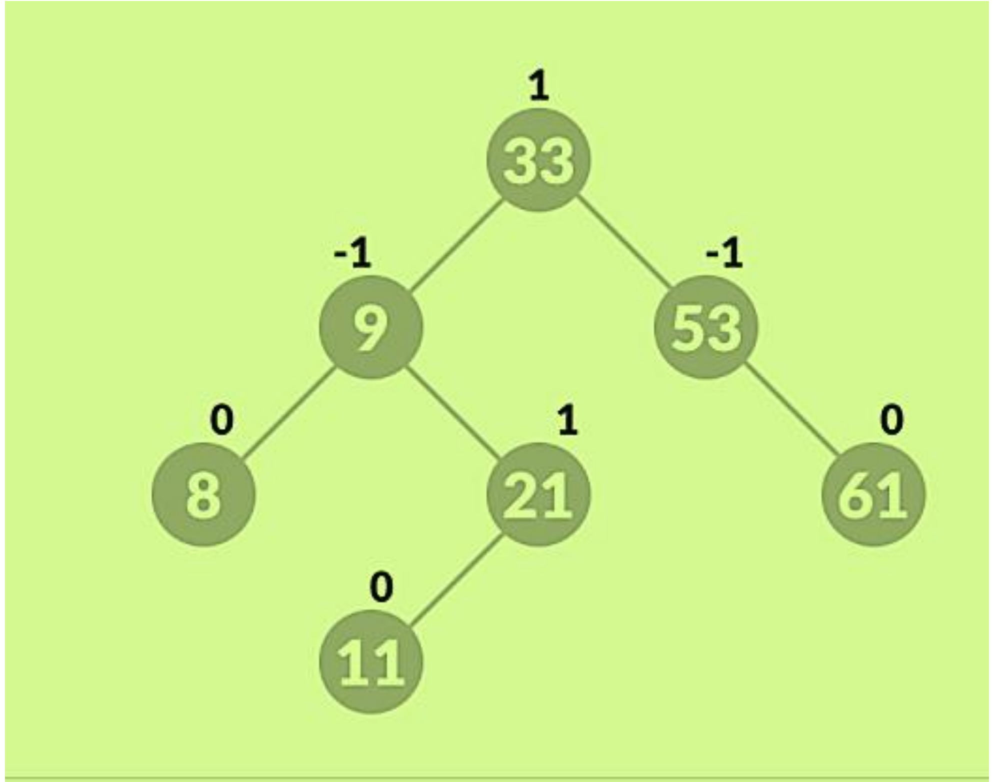
6.2 If balanceFactor of currentNode < -1,

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

6.2.2 Else do right-left rotation.

AVL Tree Deletion

7. The final tree after deletion is:



Complexities of Different Operations on an AVL Tree

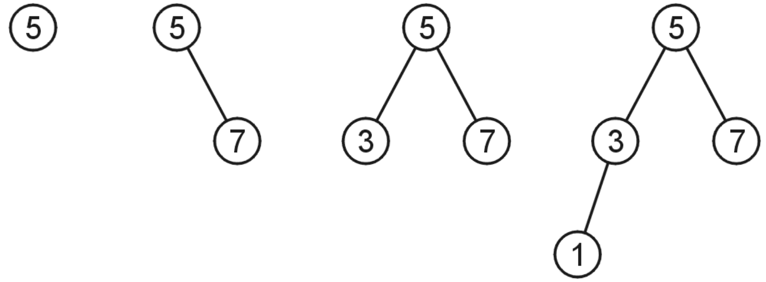
Insertion	Deletion	Search
O(log n)	O(log n)	O(log n)

Height of an AVL Tree

- By the definition of complete trees, any complete binary search tree is an AVL tree
- Thus an upper bound on the number of nodes in an AVL tree of height h a perfect binary tree with $2^{h+1} - 1$ nodes
  - What is an lower bound?

Height of an AVL Tree

Let $\mathrm{F}(h)$ be the fewest number of nodes in a tree of height h



$\mathrm{F}(0) = 1$

$\mathrm{F}(1) = 2$

$\mathrm{F}(2) = 4$

Can we find $\mathrm{F}(h)$?

Height of an AVL Tree

The worst-case AVL tree of height h would have:

- A worst-case AVL tree of height $h-1$ on one side,
- A worst-case AVL tree of height $h-2$ on the other, and
- The root node

We get: $\mathrm{F}(h) = \mathrm{F}(h-1) + 1 + \mathrm{F}(h-2)$

Height of an AVL Tree

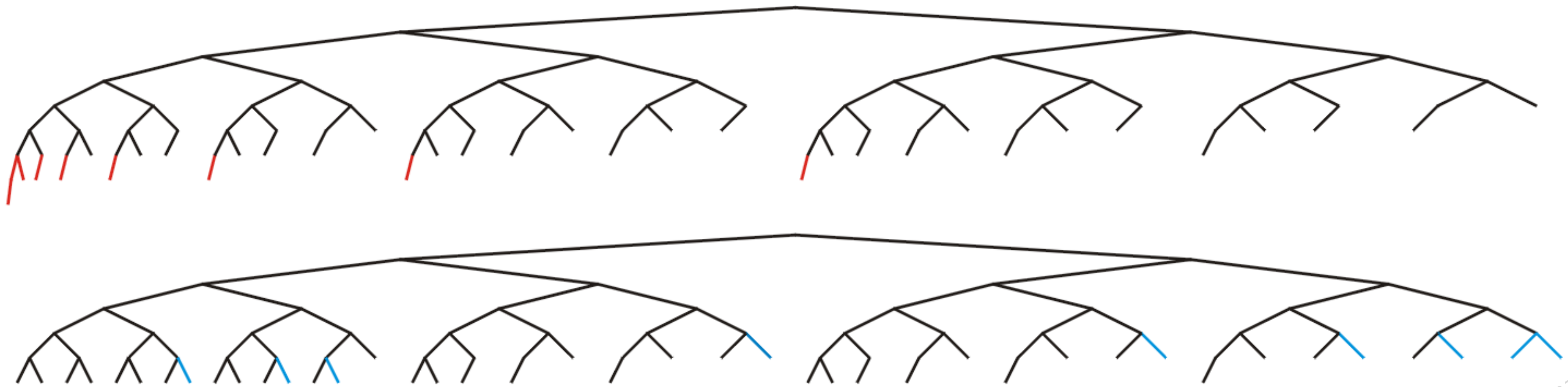
This is a recurrence relation:

$$\mathrm{F}(h) = \begin{cases} 1 & h = 0 \\ 2 & h = 1 \\ \mathrm{F}(h-1) + \mathrm{F}(h-2) + 1 & h > 1 \end{cases}$$

The solution?

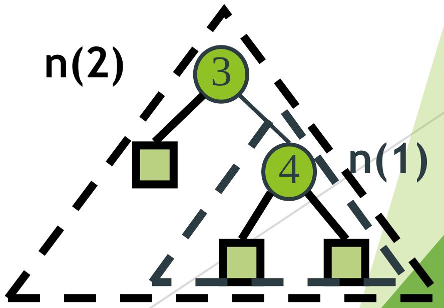
Height of an AVL Tree

In this example, $n = 88$, the worst- and best-case scenarios differ in height by only 2



Height of an AVL Tree

- **Fact**: The ***height*** of an AVL tree storing n keys is O(log n).
- **Proof**: Let us bound **n(h):** the minimum number of internal nodes of an AVL tree of height h.
- We easily see that n(1) = 1 and n(2) = 2
- For n > 2, an AVL tree of height h contains the root node, one AVL subtree of height h-1 and another of height h-2.
- That is, n(h) = 1 + n(h-1) + n(h-2)
- Knowing n(h-1) > n(h-2), we get n(h) > 2n(h-2). So
  - n(h) > 2n(h-2), n(h) > 4n(h-4), n(h) > 8n(n-6), … (by induction),
  - n(h) > 2^in(h-2i)
- Solving the base case we get: n(h) > $2^{h/2-1}$
- Taking logarithms: h < 2log n(h) +2
- Thus the height of an AVL tree is O(log n)



Pros and Cons of AVL Trees

Arguments for AVL trees:

1. Search is O(log N) since AVL trees are always balanced.
2. Insertion and deletions are also O(logn)
3. The height balancing adds no more than a constant factor to the speed of insertion.

Arguments against using AVL trees:

1. Difficult to program & debug; more space for balance factor.
2. Asymptotically faster but rebalancing costs time.
3. Most large searches are done in database systems on disk and use other structures (e.g. B-trees).
4. May be OK to have O(N) for a single operation if total run time for many consecutive operations is fast (e.g. Splay trees).