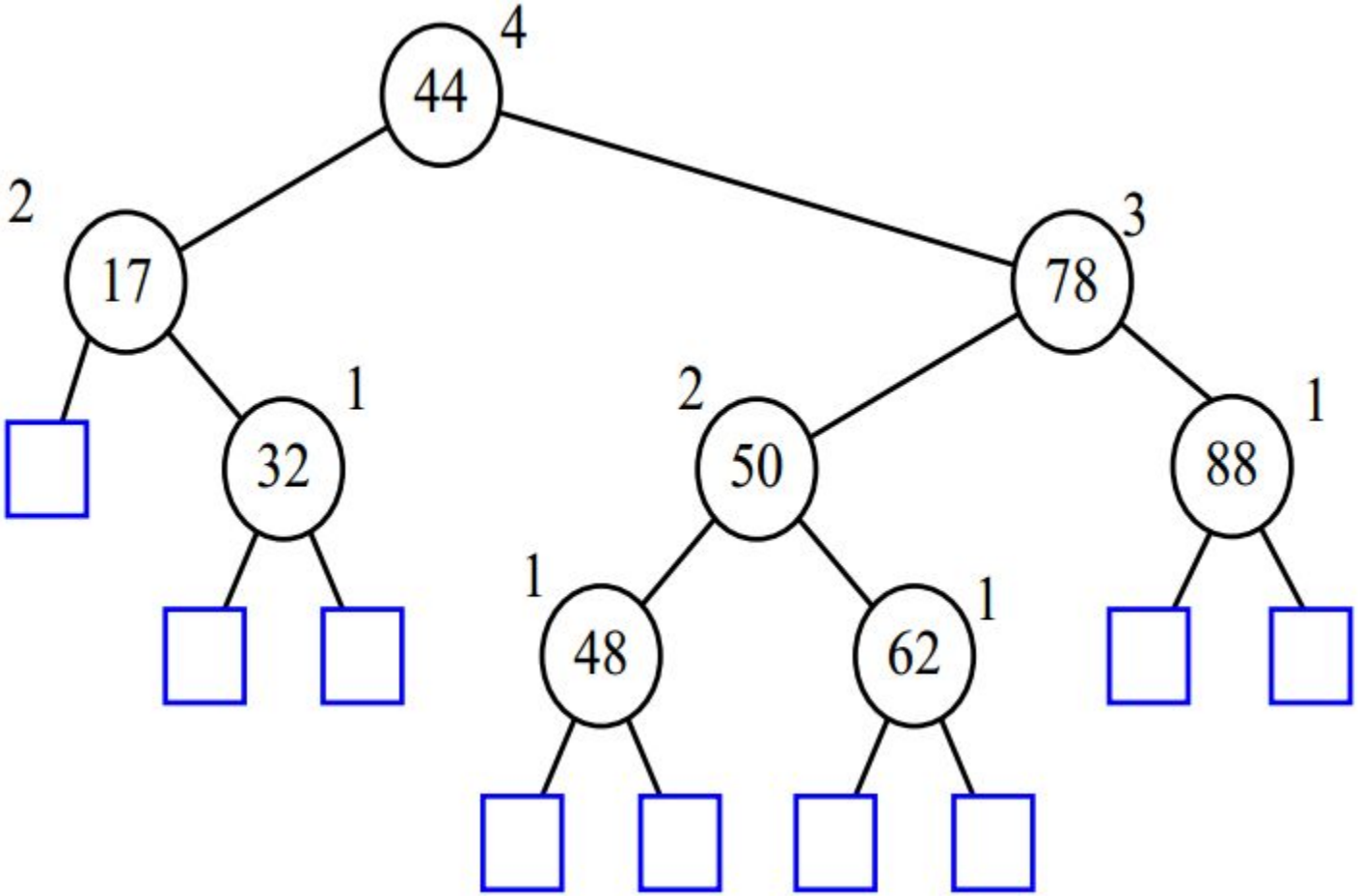
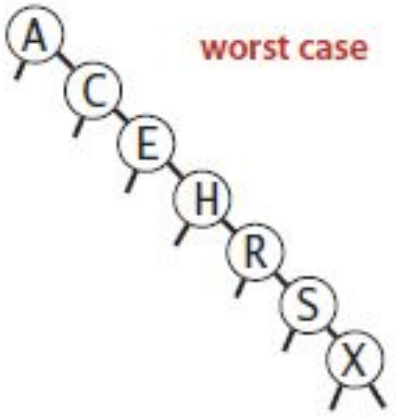
AVL Trees

- AVL tree (Adelson-Velskii and Landis' tree, named after the inventors) is a self-balancing binary search tree.
- An AVL tree is a binary search tree with a height balance property:
  - For each node v, the heights of the subtrees of v differ by at most 1.
- A subtree of an AVL tree is also an AVL tree.
- For each node of an AVL tree:

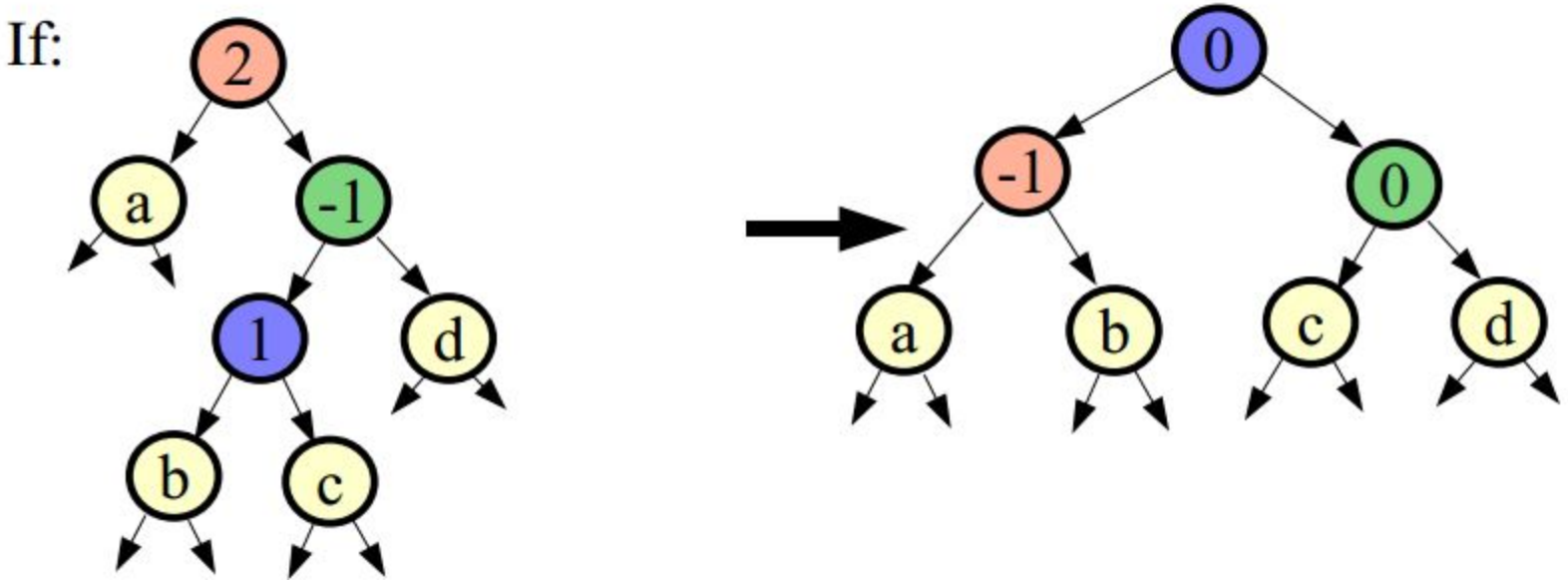
  Balance factor = height(right subtree) - height(left subtree)
- An AVL node can have a balance factor of -1, 0, or +1, otherwise it is unbalanced


4
44
2
17
78
3
1
32
2
50
88
1
1
48
62
1

- Insertion or deletion in an ordinary Binary Search Tree can cause large imbalances.
- In the worst case searching an imbalanced Binary Search Tree is O(n).

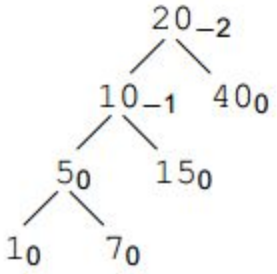


- If a tree becomes imbalanced due to an insertion or deletion, we rotate the tree such that it becomes balanced again.

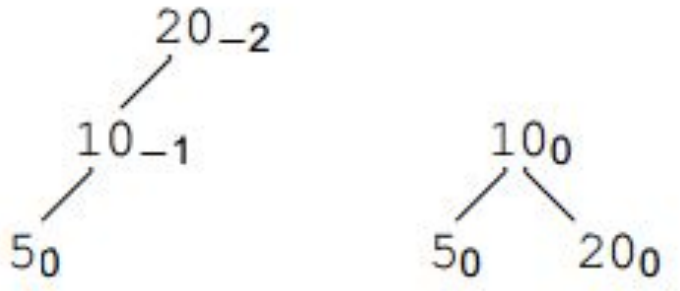


- The height-balance property ensures that the height of an AVL tree with n nodes is O(log n).
- Searching, insertion, and deletion are all O(log n).

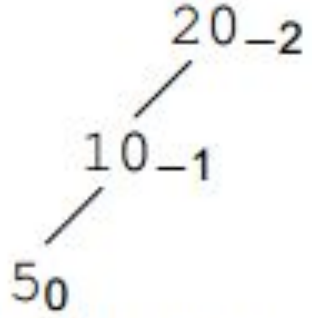
- Recall the definition:
- balance = depth(right tree) − depth(left tree).
- At every node we compute the balance, displayed as subscript



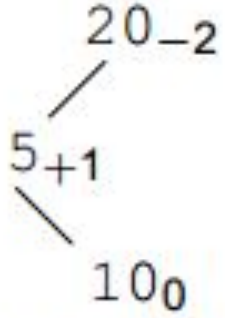
- The tree below is left heavy as the balance is −2.
- We also say that this is a left-left tree.
- Executing a right rotation balances the tree.



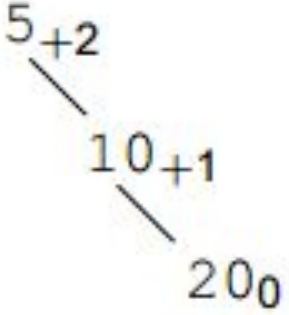
A tree is *critically unbalanced* if its balance is -2 or $+2$.



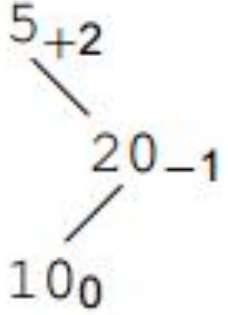
a left-left tree



a left-right tree



a right-right tree

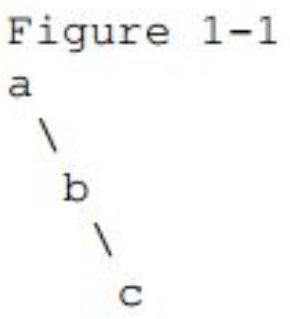


a right-left tree

LEFT ROTATION

Imagine we have this situation:

Figure 1-1



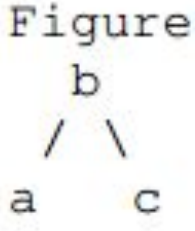
A right-right tree

To fix this, we must perform a left rotation, rooted at a. This is done in the following steps:

1. b becomes the new root.
2. a takes ownership of b's left child as its right child, or in this case, null.
3. b takes ownership of a as its left child.

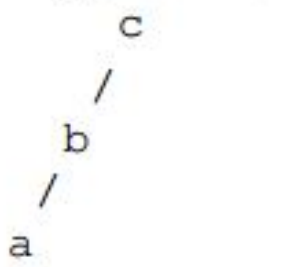
The resulting tree:

Figure 1-2



RIGHT ROTATION

Figure 1-3

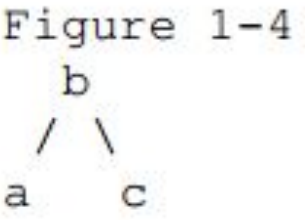


A left-left tree.
To fix this, we will perform a single right rotation, rooted at C. This is done in the following steps:

1. b becomes the new root.
2. c takes ownership of b's right child, as its left child. In this case, that value is null.
3. b takes ownership of c, as it's right child.

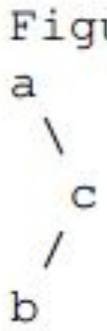
The resulting tree:

Figure 1-4



- DOUBLE ROTATION

Figure 1-6

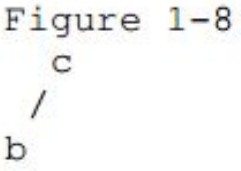


A right-left tree

We will perform a right rotation on the right subtree. We are not rotating on our current root. We are rotating on our right child. Think of our right subtree, isolated from our main tree, and perform a right rotation on it:

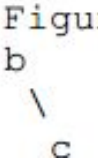
Before:

Figure 1-8



After:

Figure 1-9



So our tree becomes a right-right tree. After performing a rotation on our right subtree, we have prepared our root to be rotated left.

left rotation.

Figure 1-10

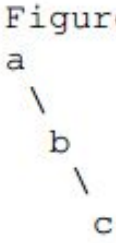
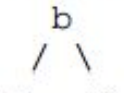
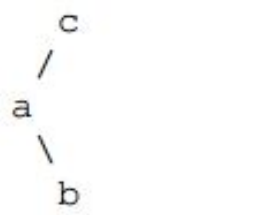


Figure 1-11



- To balance a tree which has a left subtree, that is right heavy.
- Or a left right tree.

Figure 1-12



The left subtree has a height of 2, and the right subtree has a height of 0. This makes the balance factor of our root node, c, equal to -2.

Balance factor = height(right subtree) - height(left subtree)

First make it a left-left tree, and then perform right rotation.

Figure 1-15

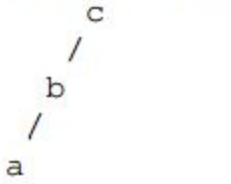
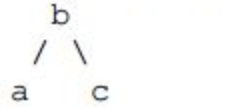


Figure 1-16



- IF tree is right heavy
- {
- IF tree's right subtree is left heavy
- {
- Perform Double Left rotation
- }
- ELSE
- {
- Perform Single Left rotation
- }
- }
- ELSE IF tree is left heavy
- {
- IF tree's left subtree is right heavy
- {
- Perform Double Right rotation
- }
- ELSE
- {
- Perform Single Right rotation
- }
- }

Insertions

- Let the newly inserted node be w
- Perform standard BST insert for w.
- 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.

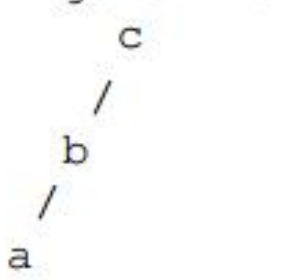
- //right rotate subtree rooted with y
- Node rightRotate(Node y) {
- Node x = y.left;
- Node T2 = x.right;

- // Perform rotation
- x.right = y;
- y.left = T2;

- // Update heights
- y.height = max(height(y.left), height(y.right)) + 1;
- x.height = max(height(x.left), height(x.right)) + 1;

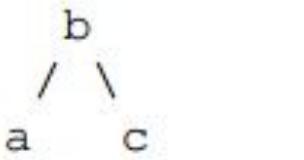
- // Return new root
- return x;
- }

Figure 1-3



The resulting tree:

Figure 1-4



1. b becomes the new root.
2. c takes ownership of b's right child, as its left child.
3. b takes ownership of c, as it's right child.

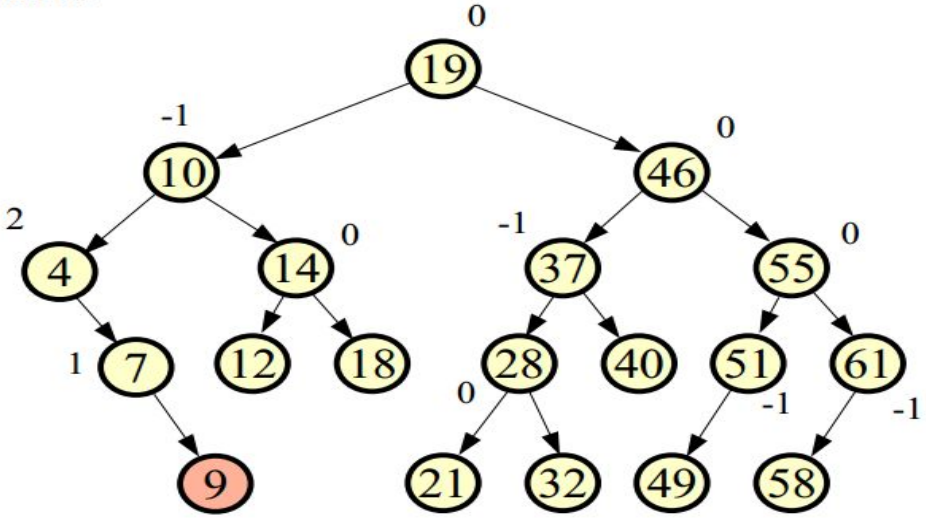
```
//left rotate subtree rooted with x
Node leftRotate(Node x) {
        Node y = x.right;
        Node T2 = y.left;

        // Perform rotation
        y.left = x;
        x.right = T2;

        // Update heights
        x.height = max(height(x.left), height(x.right)) + 1;
        y.height = max(height(y.left), height(y.right)) + 1;

        // Return new root
        return y;
  }
```

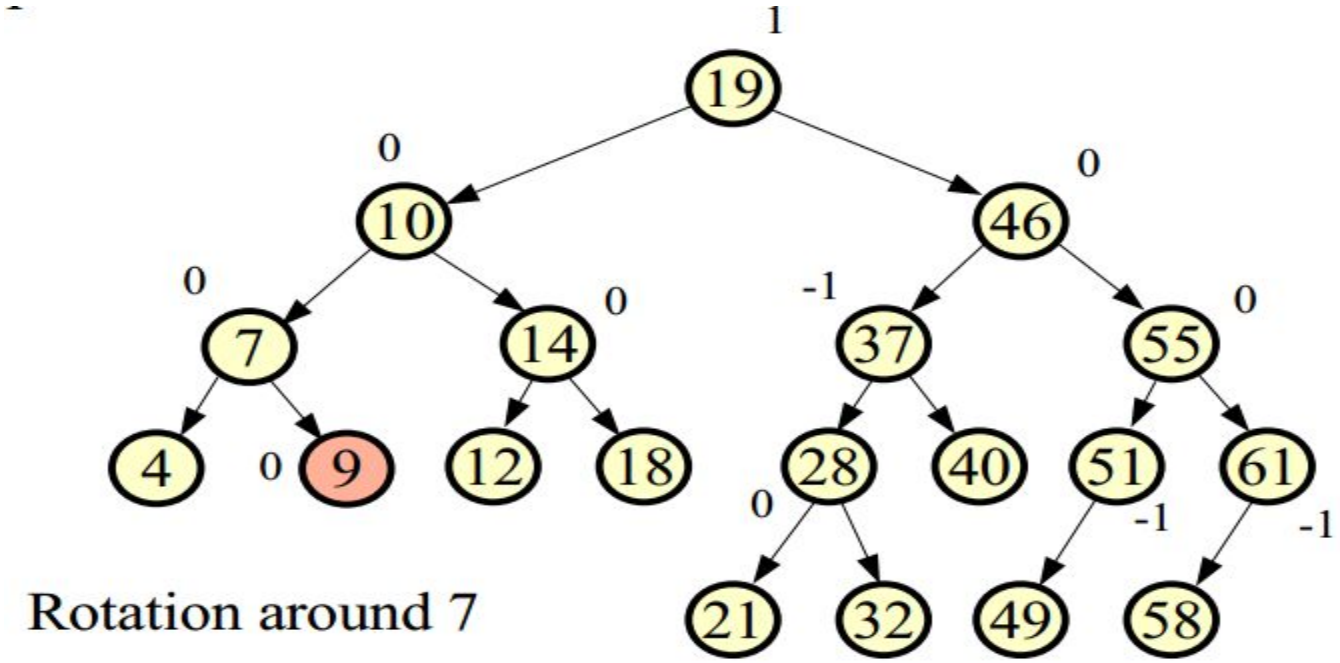
- Balance of node N
- int getBalance(Node N) {
- if (N == null)
- return 0;

- return height(N.left) - height(N.right);
- }

```
Node insert(Node node, int key) {
//Get the balance factor of this ancestor node to check whether this node became
unbalanced */
            int balance = getBalance(node);
// If this node becomes unbalanced, then there // are 4 cases
//Left Left Case
            if (balance > 1 && key < node.left.key)
                  return rightRotate(node);
// Right Right Case
            if (balance < -1 && key > node.right.key)
                  return leftRotate(node);
// Left Right Case
            if (balance > 1 && key > node.left.key) {
                  node.left = leftRotate(node.left);
                  return rightRotate(node);
            }
// Right Left Case
            if (balance < -1 && key < node.right.key) {
                  node.right = rightRotate(node.right);
                  return leftRotate(node);
            }
            return node;
      }
```

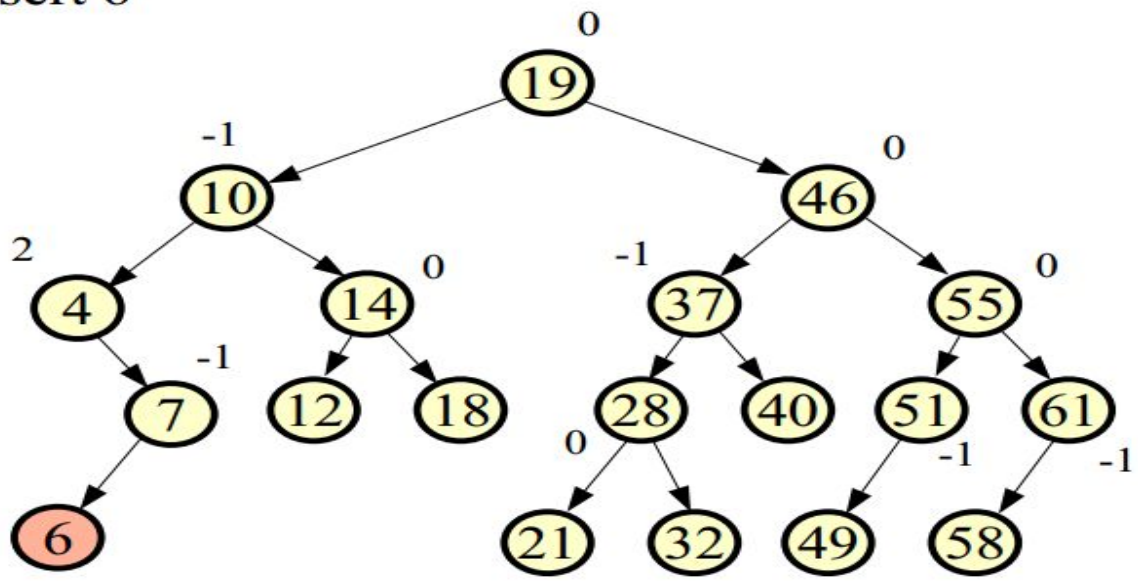
Example: Insert 9

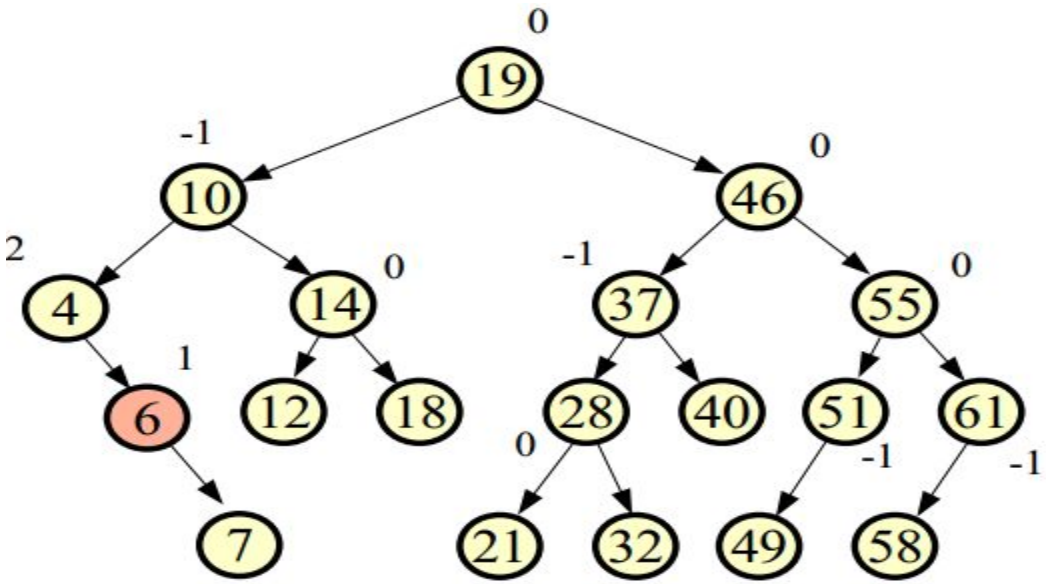




Rotation around 7

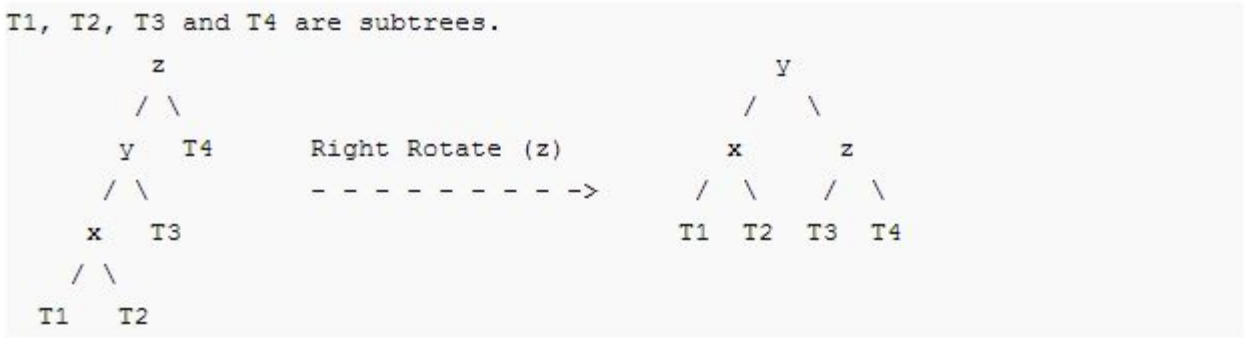
Example: Insert 6



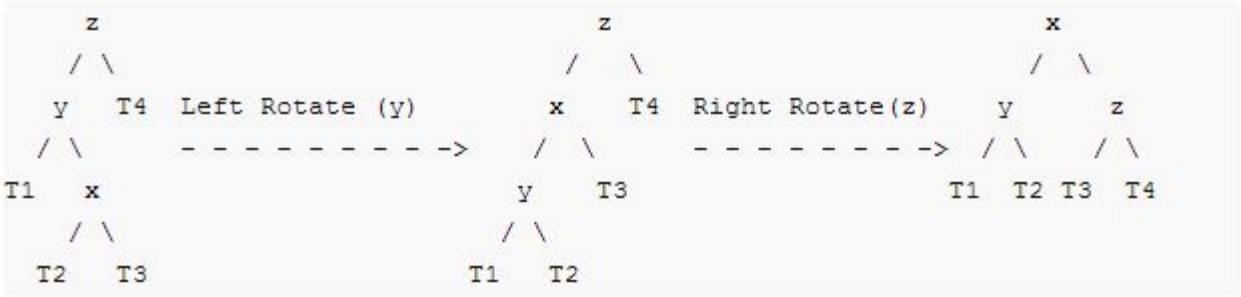


Double rotation

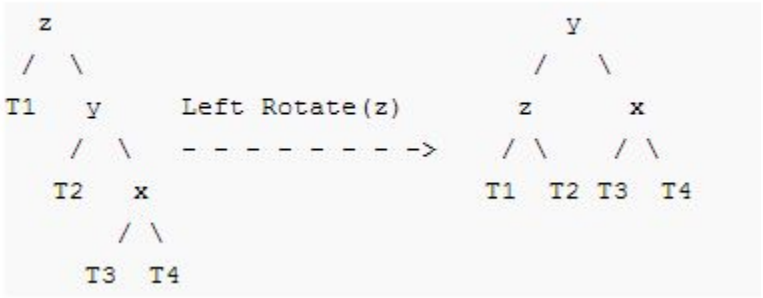
a) Left Left Case



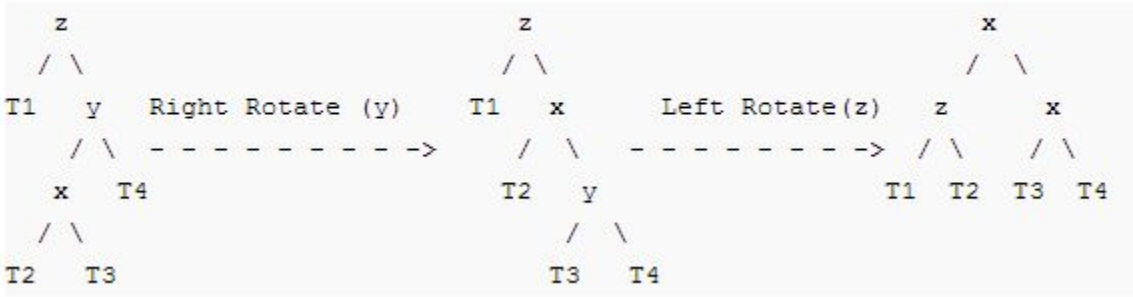
b) Left Right Case



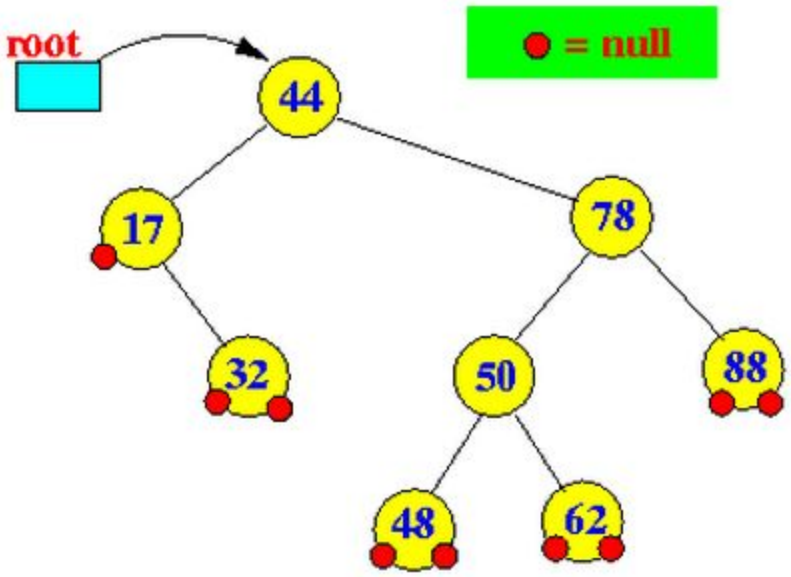
c) Right Right Case



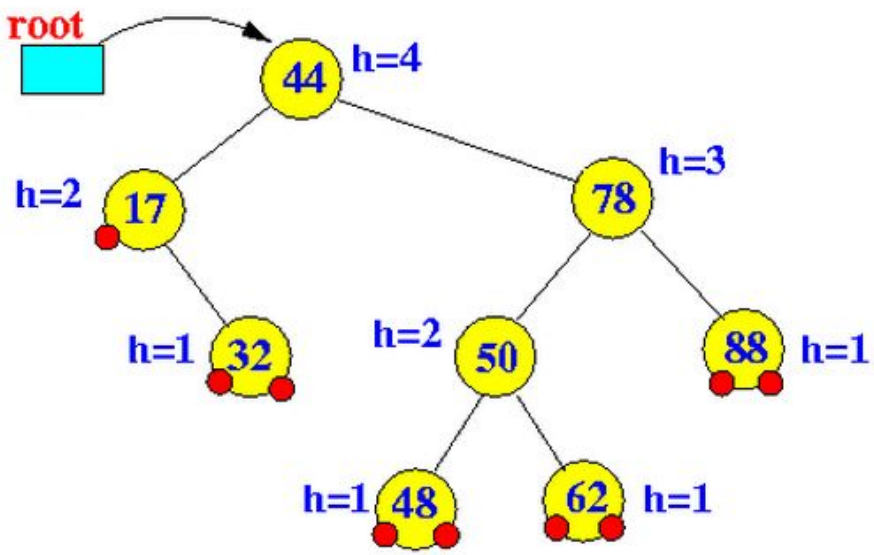
d) Right Left Case



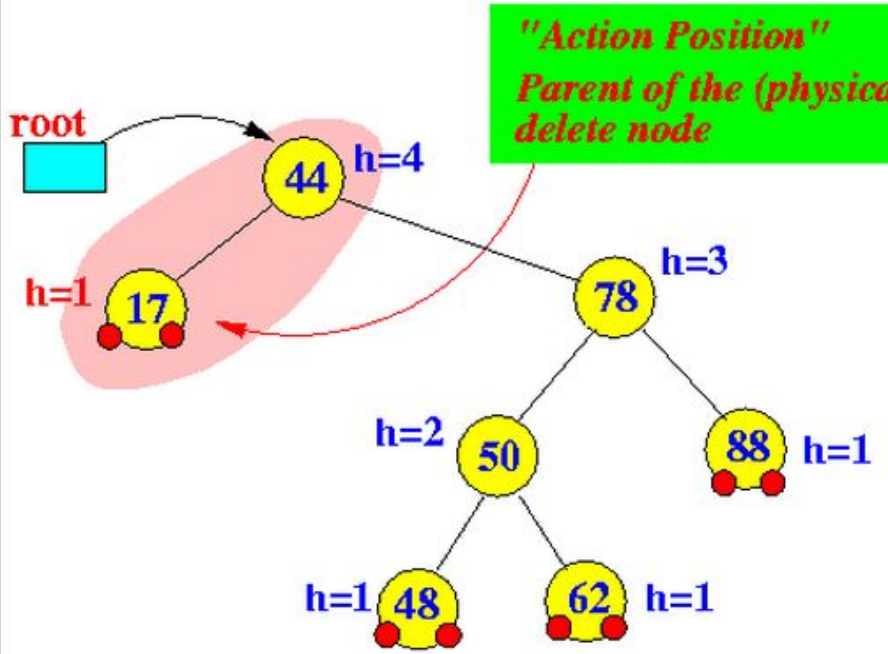
Deletion

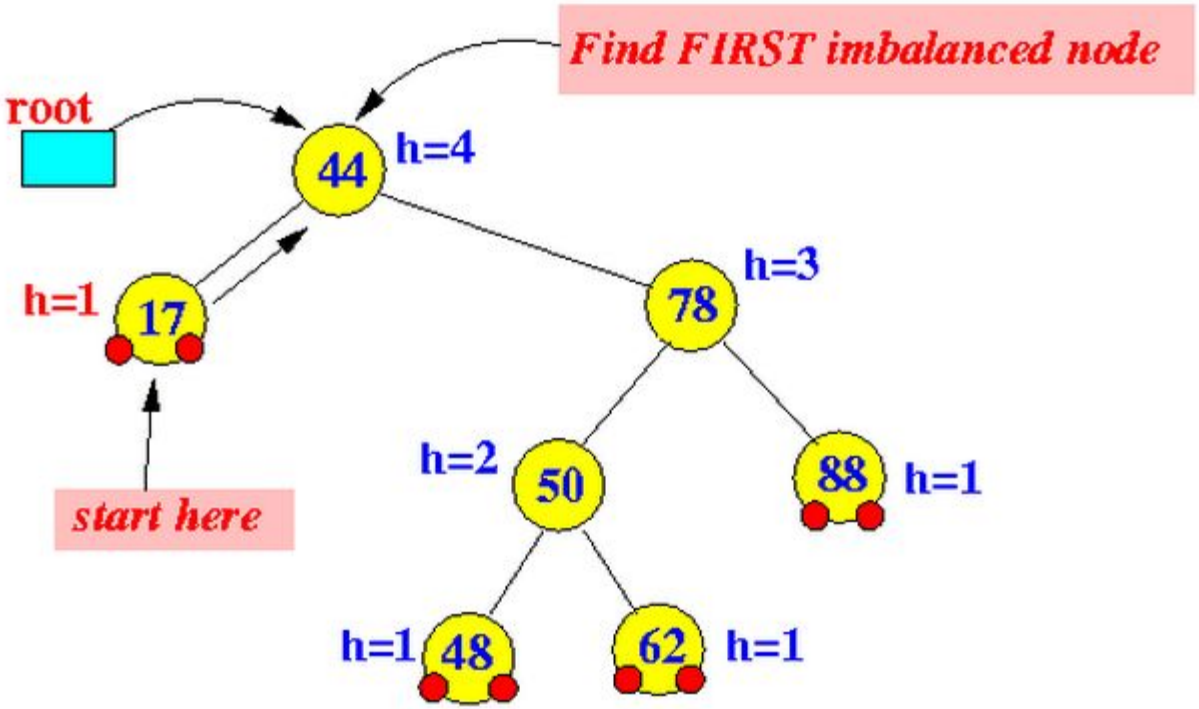


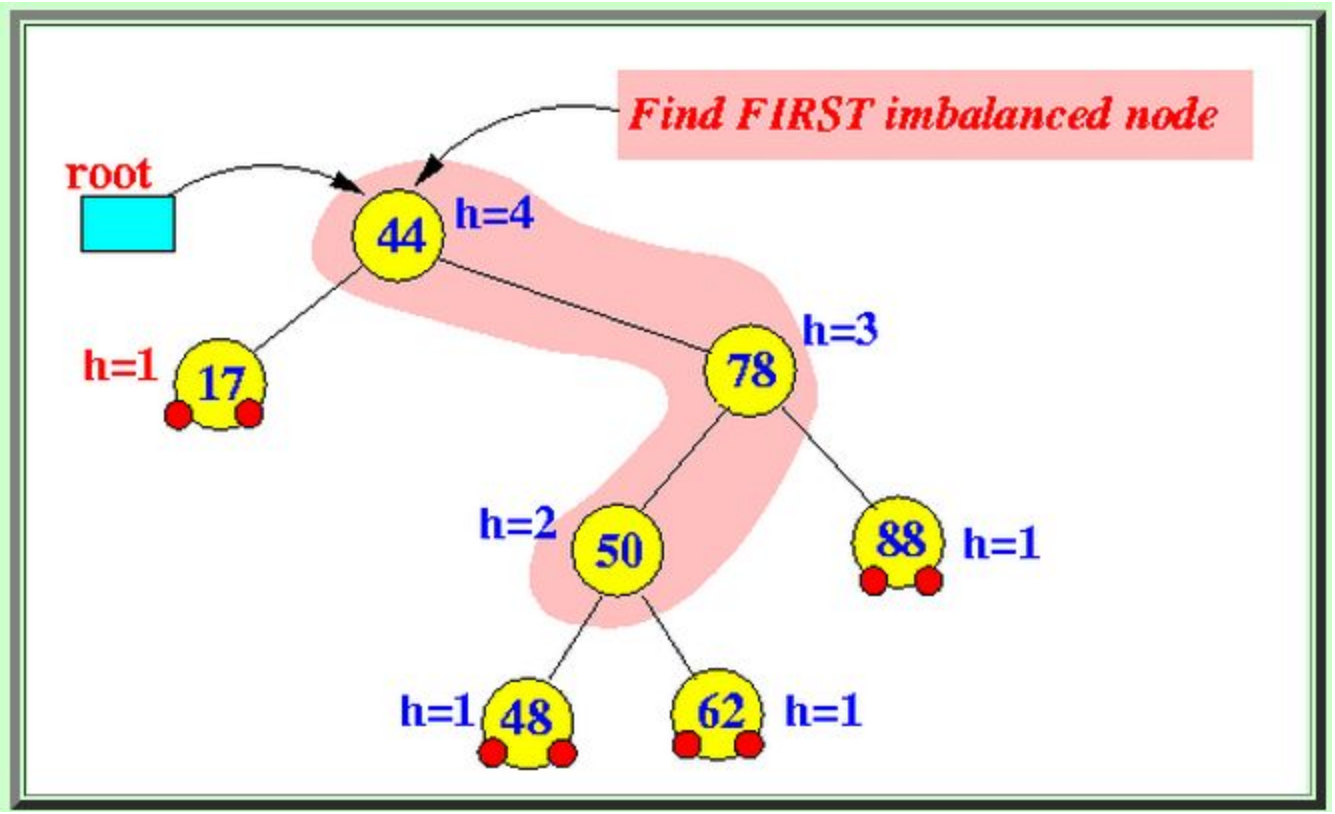
Before deleting node 32

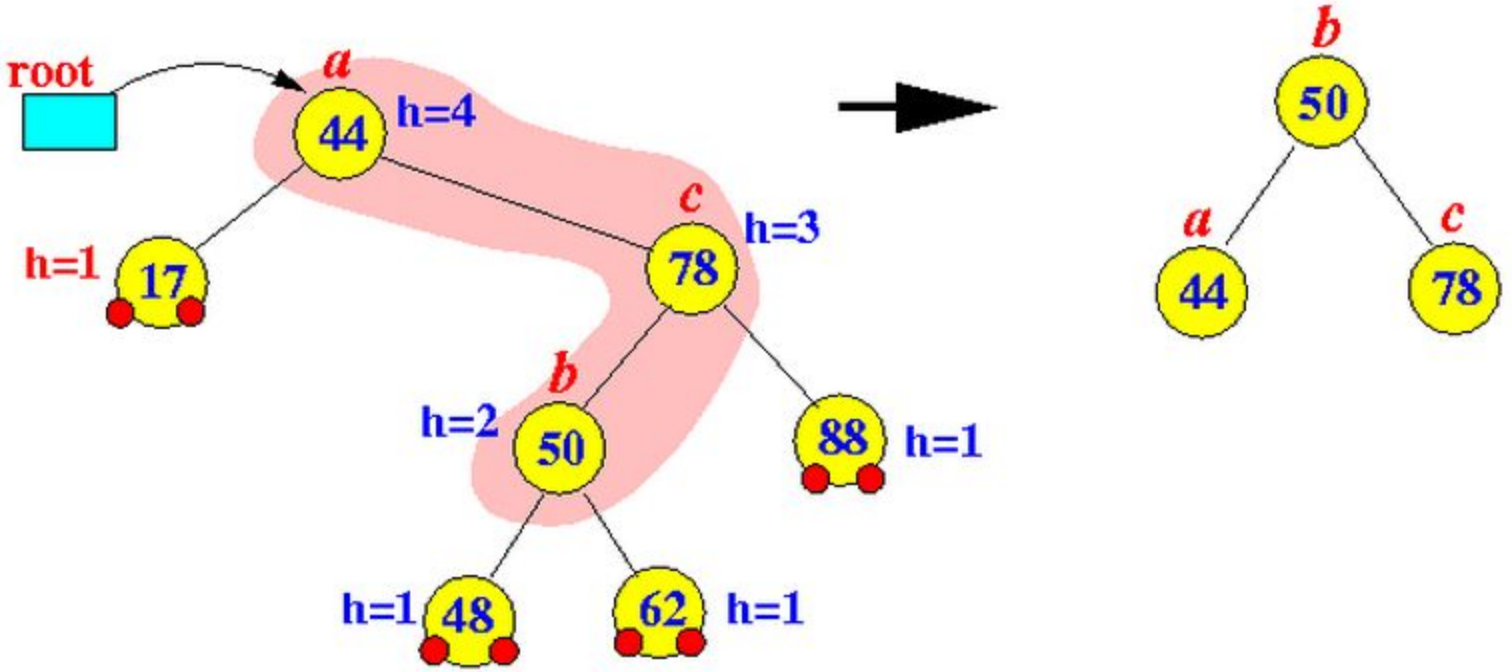


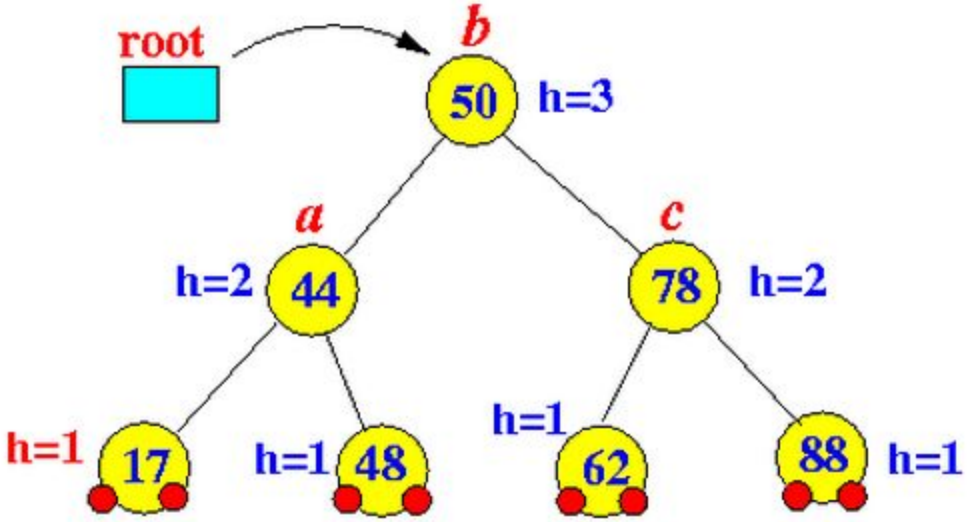
After deleting node 32




Find FIRST imbalanced node
root
44
h=4
h=1
17
78
h=3
start here
h=2
50
88
h=1
h=1
48
62
h=1


Find FIRST imbalanced node
root
44
h=4
h=1
17
78
h=3
h=2
50
88
h=1
h=1
48
62
h=1





1. b becomes the new root.
2. a takes ownership of b's left child as its right child.
3. b takes ownership of a as its left child.

References

- Wikipedia
- http://pages.cs.wisc.edu/~paton/readings/liblitVersion/AVL-Tree-Rotations.pdf
- http://gauss.ececs.uc.edu/Courses/C228/LectureNotes/Trees/BalancedBinaries/avl.pdf
- http://www.dcs.gla.ac.uk/~pat/52233/slides/AVLTrees1x1.pdf
- http://homepages.math.uic.edu/~jan/mcs360/balancing_search_trees.pdf
- http://cs-study.blogspot.com/2012/11/cases-of-rotation-of-avl-tree.html
- http://www.mathcs.emory.edu/~cheung/Courses/323/Syllabus/Trees/AVL-delete.html