LAB SESSION # 06

Outline

- Deletion in BST
- Searching in AVL
- Insertion in AVL
- Exercise

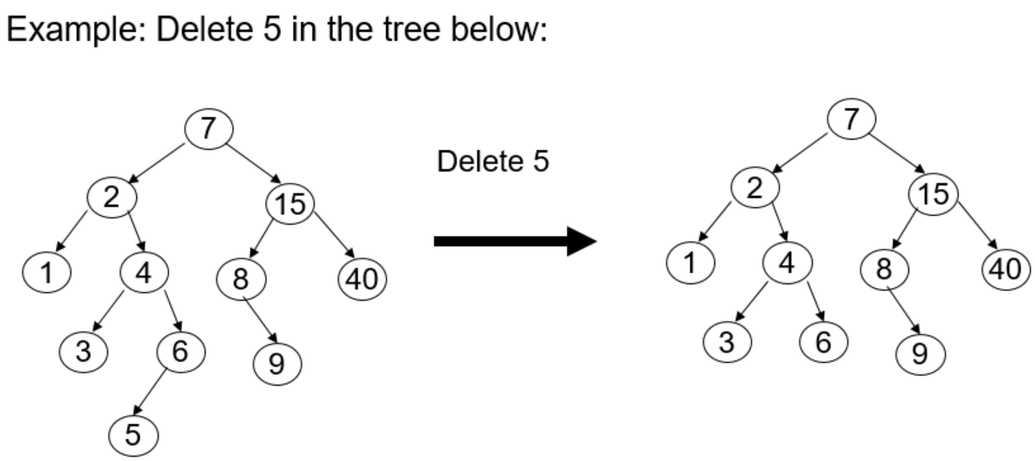
DELETION IN BST

There are three cases:

1. The node to be deleted is a leaf node.
2. The node to be deleted has one non-empty child.
3. The node to be deleted has two non-empty children.

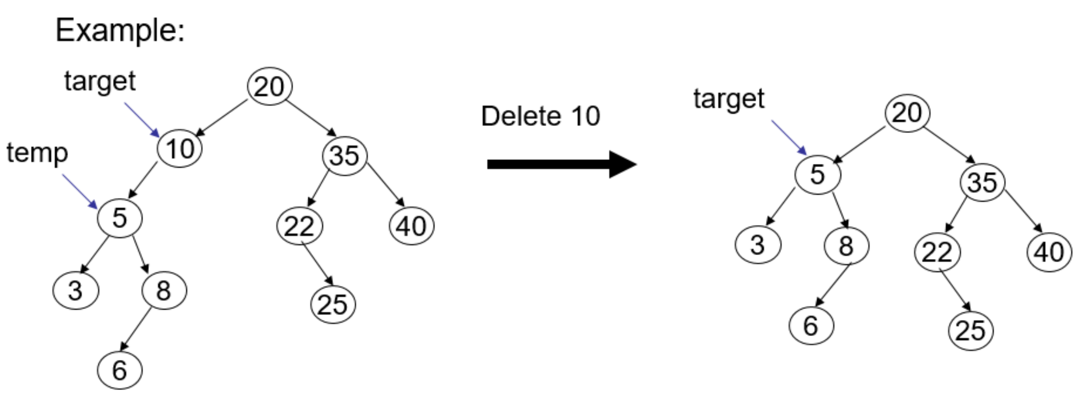
CASE 1: DELETING A LEAF NODE

Convert the leaf node into an empty tree by using the detachKey method:



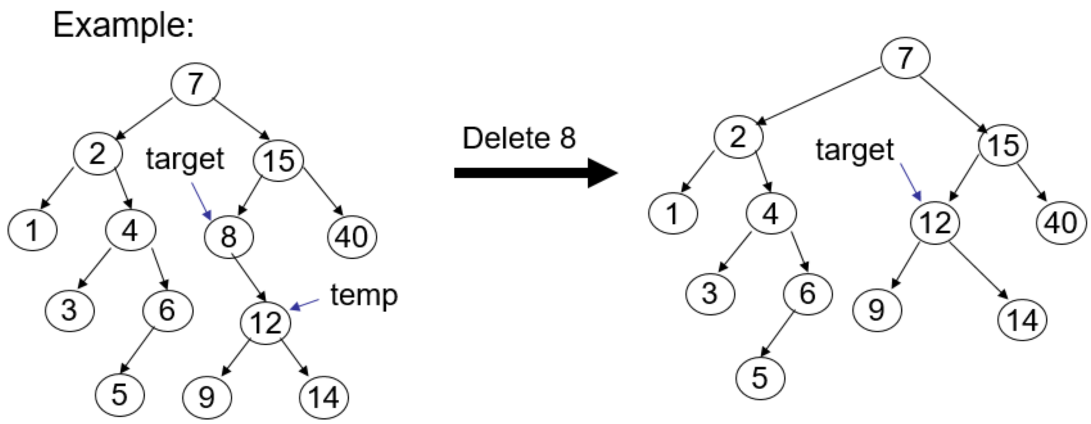
CASE 2: THE NODE TO BE DELETED HAS ONE NON-EMPTY CHILD

a. **The right subtree of the node x to be deleted is empty.**



Prepared by: Instructor, Faizan Yousuf

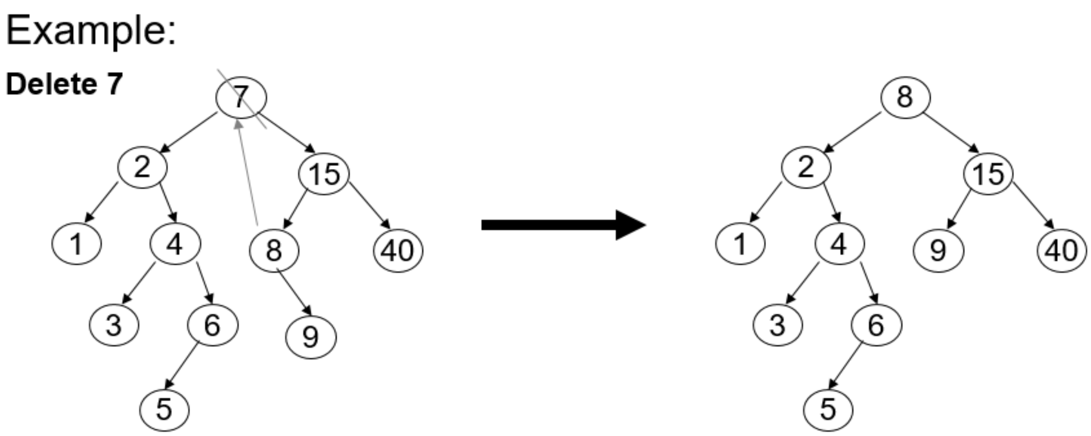
b. **The left subtree of the node x to be deleted is empty.**



CASE 3: DELETING A NODE THAT HAS TWO CHILDREN

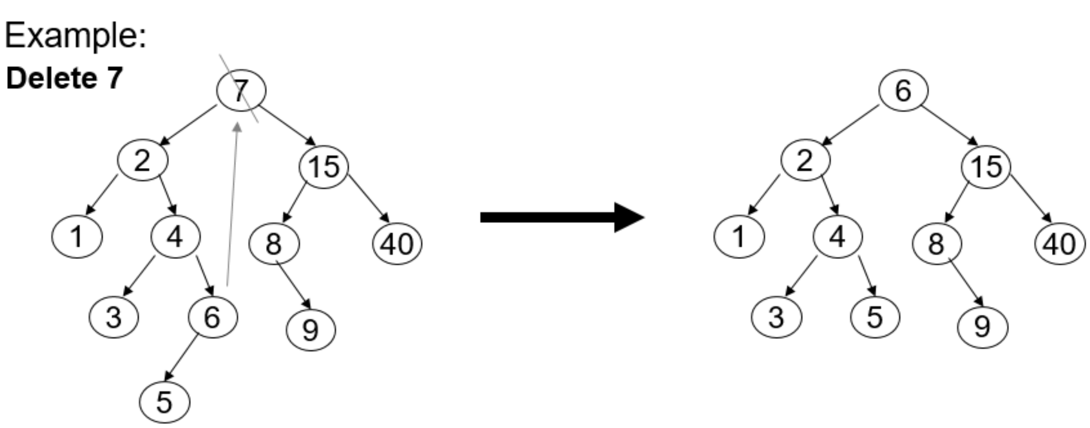
a. Method # 1: DELETION BY COPYING

Copy the minimum key in the right subtree of x to the node x, then delete the one-child or leaf-node with this minimum key.



b. Method # 2: DELETION BY COPYING:

Copy the maximum key in the left subtree of x to the node x, then delete the one-child or leaf-node with this maximum key.



Deletion by Copying Code

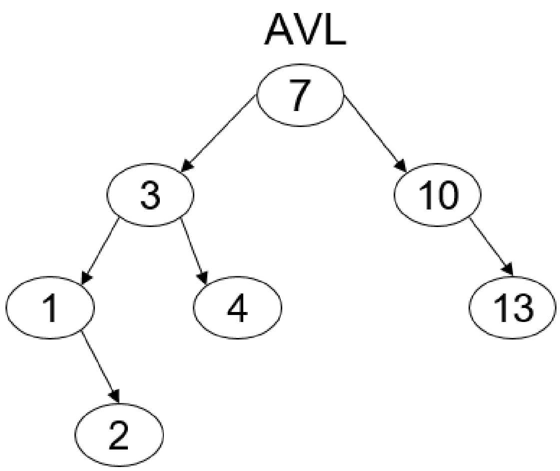
```
// find the minimum key in the right subtree of the target node
Comparable min = target.getRightBST().findMin();

// copy the minimum value to the target
target.key = min;

// delete the one-child or leaf node having the min
target.getRightBST().withdraw(min);
```

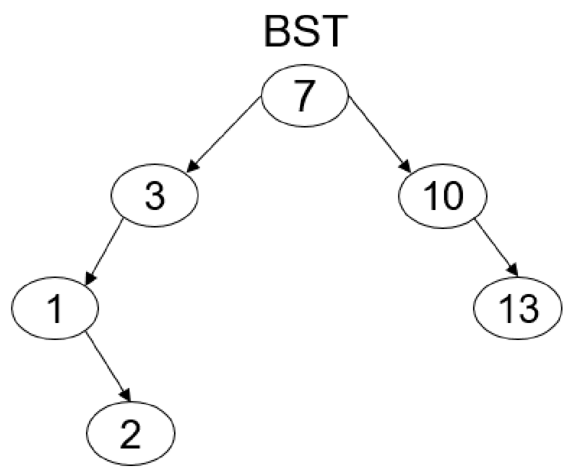
All the different cases for deleting a node are handled in the **withdraw (Comparable key)** method of **BinarySearchTree** class

AVL 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 can have a balance factor: Balance factor = height (right subtree) - height (left subtree)
An AVL tree is rebalanced after each insertion or deletion.
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).





Prepared by: Instructor, Faizan Yousuf

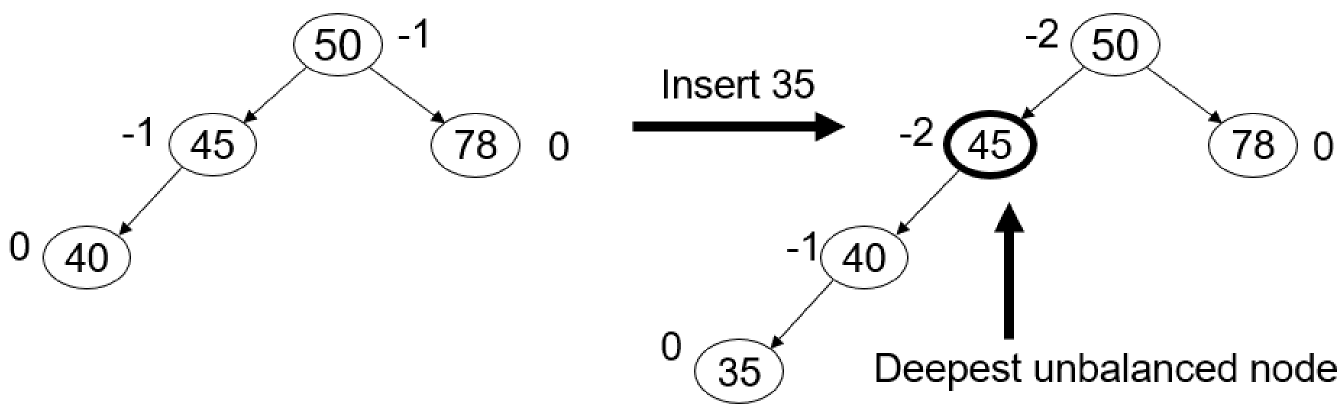
AVL TREE IMPLEMENTATION:

```
public class AVLTree extends BinarySearchTree{
   protected int height;
   public AVLTree(){ height = -1;}

   public int getHeight(){ return height } ;

   protected void adjustHeight(){
      if( isEmpty())
         height = -1;
      else
         height = 1 + Math.max( left.getHeight() , right.getHeight());
   }

   protected int getBalanceFactor(){
      if( isEmpty())
         return 0;
      else
         return right.getHeight() - left.getHeight();
   }
   // . . .
}
```

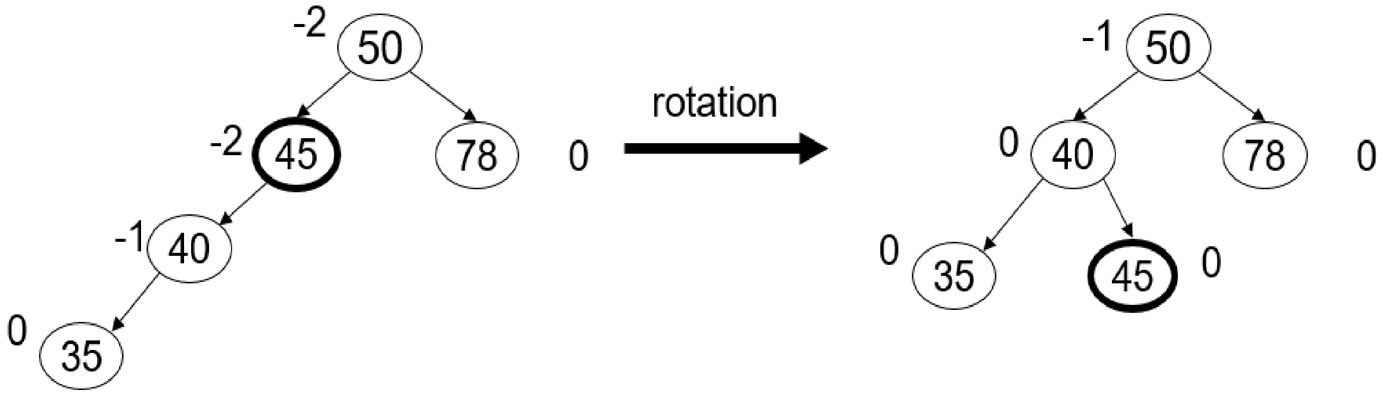
AVL TREE ROTATION:

- A rotation is a process of switching children and parents among two or three adjacent nodes to restore balance to a tree.
- **An insertion or deletion may cause an imbalance in an AVL tree.**

The deepest node, which is an ancestor of a deleted or an inserted node, and whose balance factor has changed to -2 or +2 requires rotation to rebalance the tree.



Prepared by: Instructor, Faizan Yousuf



Tree Imbalance

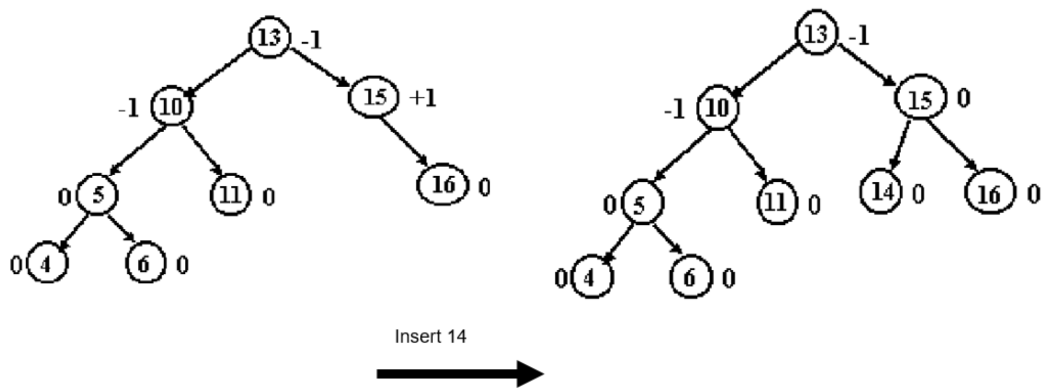
- Left Left Imbalance (Single Rotation)
- Right Right Imbalance (Single Rotation)
- Left-Right Imbalance (Double Rotation)
- Right-Left Imbalance (Double Rotation)

AVL INSERTION:

- Insert using a BST insertion algorithm.
- Rebalance the tree if an imbalance occurs.
- An imbalance occurs if a node's balance factor changes from -1 to -2 or from+1 to +2.
- Rebalancing is done at the deepest or lowest unbalanced ancestor of the inserted node.
- There are three insertion cases:
    1. Insertion that does not cause an imbalance.
    2. Same side (left-left or right-right) insertion that causes an imbalance.
        - Requires a single rotation to rebalance.
    3. Opposite side (left-right or right-left) insertion that causes an imbalance.
        - Requires a double rotation to rebalance.

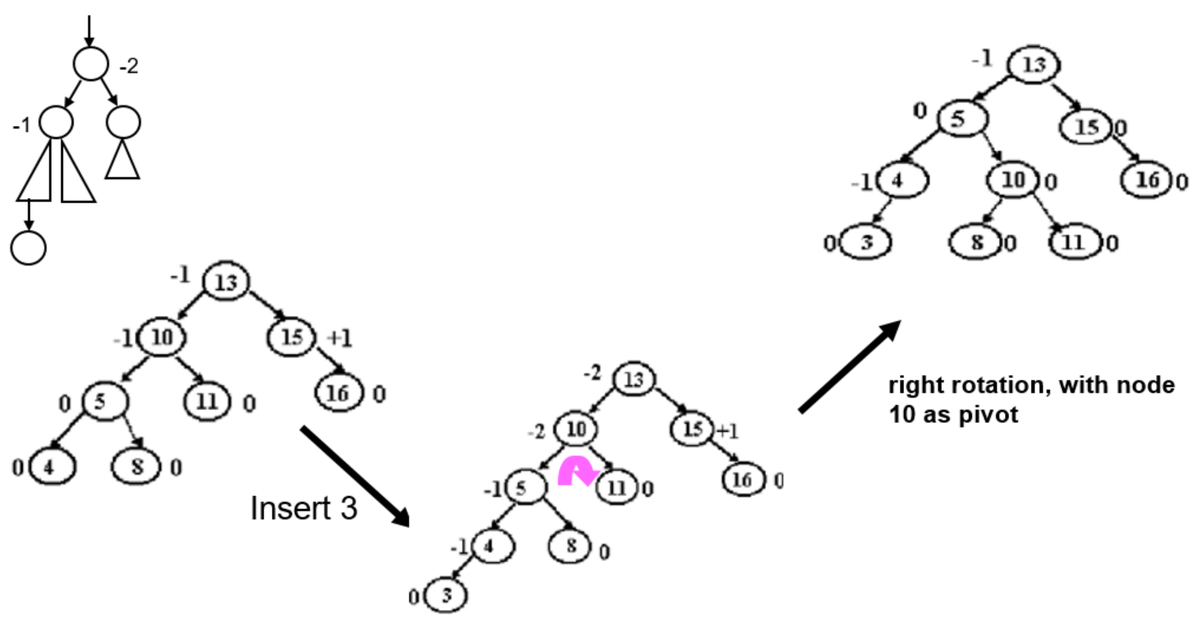
Insertion Case 1:

- Example: An insertion that does not cause an imbalance.

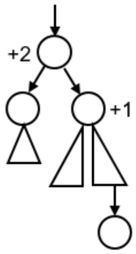


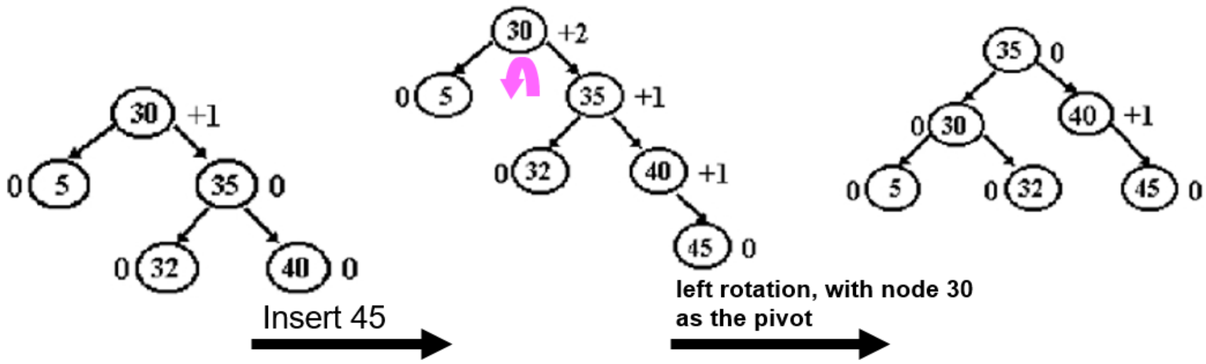
Insertion Case 2:

- Case 2a: The lowest node (with a balance factor of -2) had a taller left-subtree and the insertion was on the left-subtree of its left child.
- Requires single right rotation to rebalance.



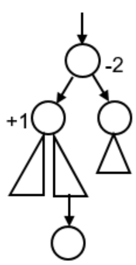
- Case 2b: The lowest node (with a balance factor of +2) had a taller right-subtree and the insertion was on the right-subtree of its right child.
- Requires single left rotation to rebalance.

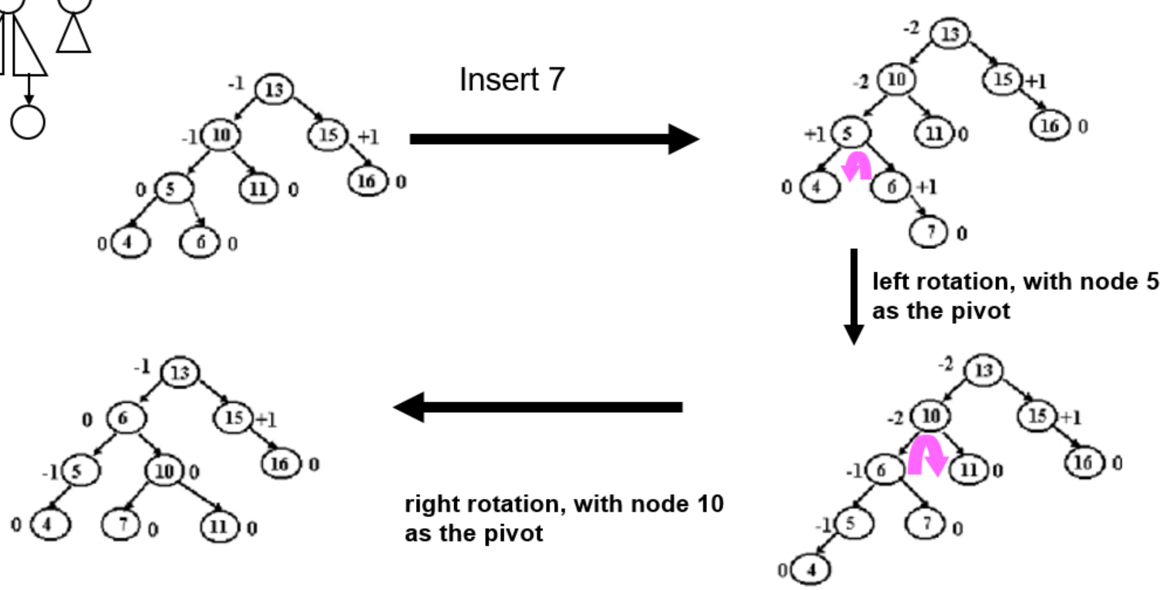




Insertion Case 3:

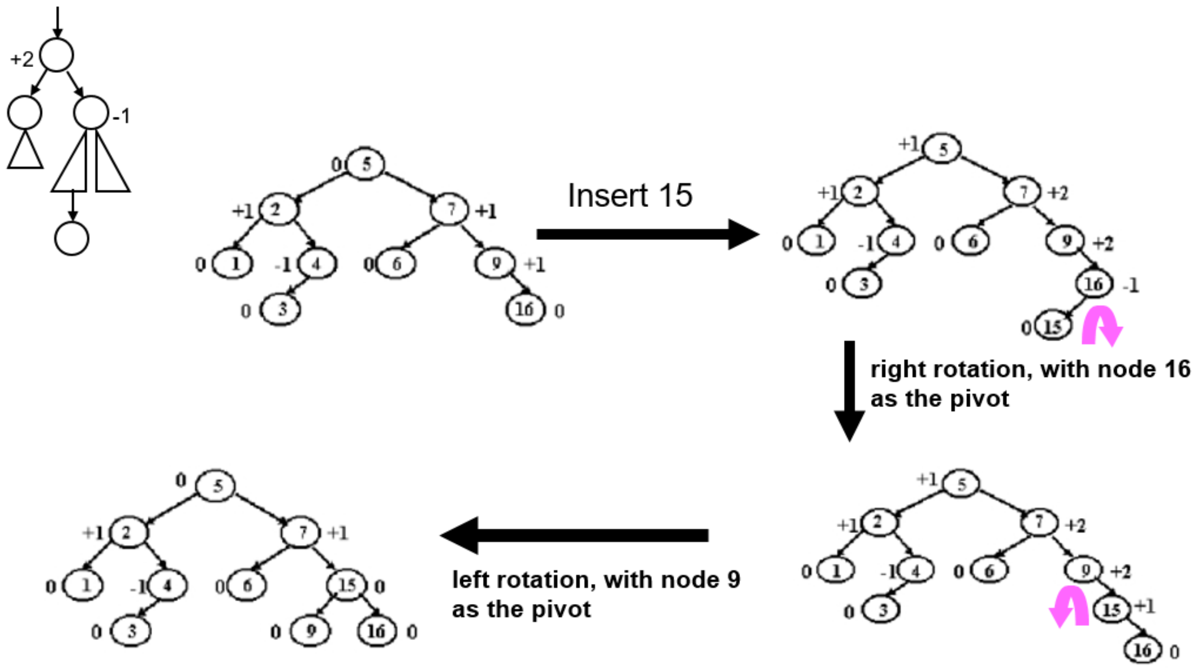
- Case 3a: The lowest node (with a balance factor of -2) had a taller left-subtree and the insertion was on the right-subtree of its left child.
- Requires a double left-right rotation to rebalance.





Prepared by: Instructor, Faizan Yousuf

- Case 3b: The lowest node (with a balance factor of +2) had a taller right-subtree and the insertion was on the left-subtree of its right child.
- Requires a double right-left rotation to rebalance.



Exercise:

Question No. 1:

Write a Java program to perform the following operations

(a) Build BST Tree
(b) Deletion in BST Tree (All Cases)

Question No. 2:

Write a Java program to perform the following operations

(a) Build AVL Tree
(b) Write code for AVL Rotations (Single and Double)
(c) Insertion in AVL Tree (All Cases)

Question No. 3:

Write a Java program to perform the following operations

(a) Build AVL Tree
(b) Insert Values
(c) Search for a value in AVL Tree using DFS and BFS
(d) Calculate time in Second for all operations.

Question No. 4:

Write a Java program that uses recursive functions.

(a) To create a binary search tree.
(b) Write a Function to find whether the tree is AVL or not
(c) Write a Function to calculate height of a Tree