Data Structures and Algorithm

Moazzam Ali Sahi

Lecture # 23

AVL Trees

Adel'son-Vel'skii and Landis

Last Lecture

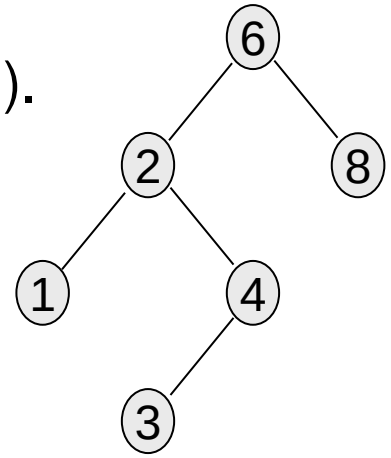
- Binary Search Tree

This Lecture

- AVL (Height-Balanced Trees)

AVL Trees

- An AVL tree is a binary search tree with a *balance* condition.
- AVL is named for its inventors: **A**del'son-**V**el'skii and **L**andis
- AVL tree *approximates* the ideal tree (completely balanced tree).
- AVL Tree maintains a height close to the minimum.

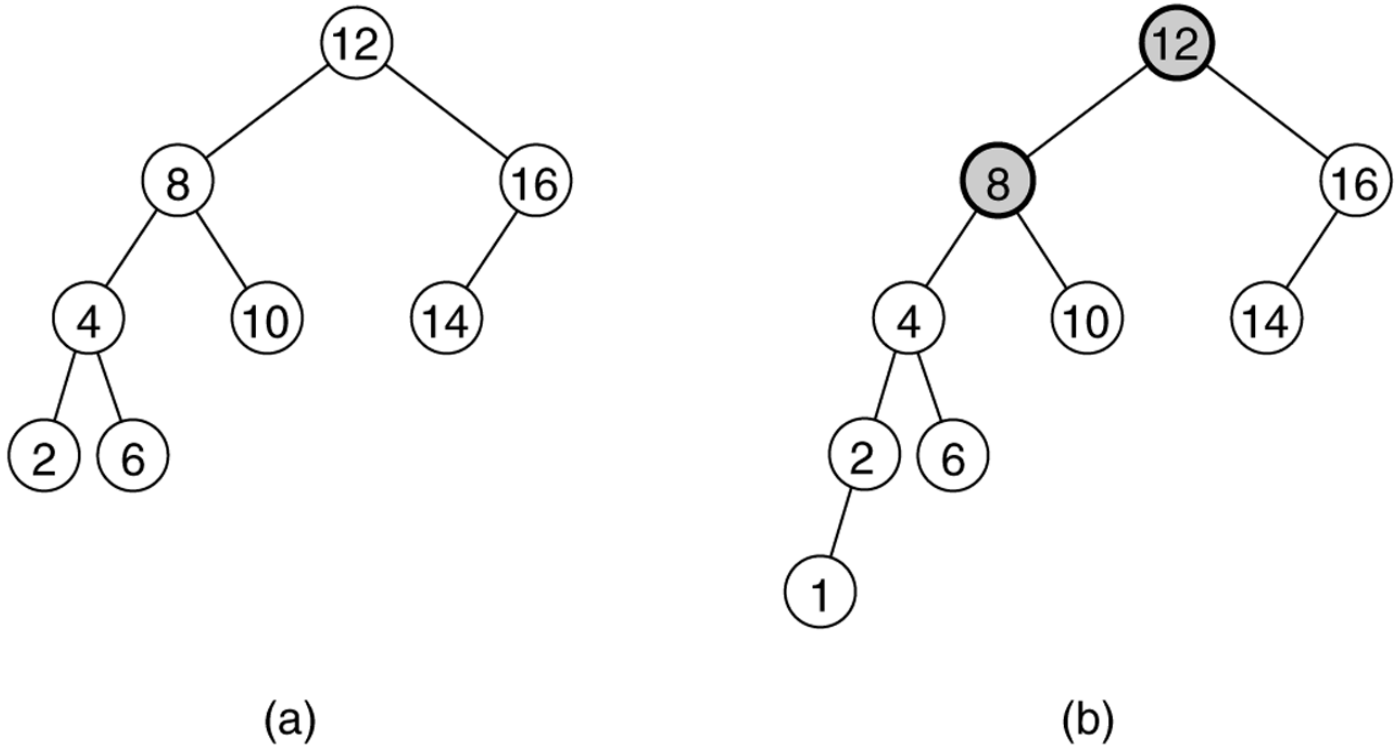


Definition:

An AVL tree is a binary search tree such that
for any node in the tree, the height of the left and
right subtrees can differ by at most 1.

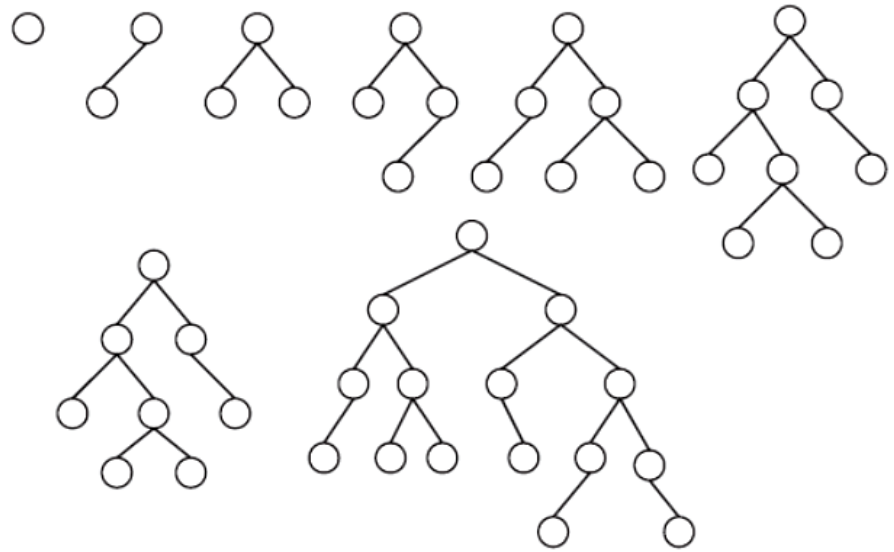
Recap: height is the length of the longest path from root to a leaf, here height of the tree is 3, subtree rooted by node 2 is 2, by 8 is 0. Tree is unbalanced.

AVL Trees

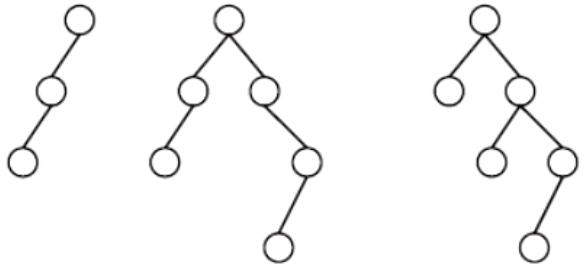


Two binary search trees: (a) an AVL tree; (b) not an AVL tree (unbalanced nodes are darkened)

AVL Trees



(a) AVL trees



(b) Non-AVL trees

Why AVL Trees?

AVL Concepts

- Let x be a node in a binary tree. Let x_l denote the height of the left subtree of x, and x_h denote the height of the right subtree of x.

Proposition: Let T be an AVL tree and x be a node in T. Then $|x_h - x_l| \leq 1$, where $|x_h - x_l|$ denotes the absolute value of $x_h - x_l$.

Let x be a node in the AVL tree T.

1. If $x_l > x_h$, we say that x is **left high**. In this case, $x_l = x_h + 1$.
2. If $x_l = x_h$, we say that x is **equal high**.
3. If $x_h > x_l$, we say that x is **right high**. In this case, $x_h = x_l + 1$.

AVL Tree (Balance Factor)

Definition: The **balance factor** of x, written $bf(x)$, is defined by $bf(x) = x_h - x_l$.

Let x be a node in the AVL tree T. Then,

1. If x is left high, $bf(x) = -1$.
2. If x is equal high, $bf(x) = 0$.
3. If x is right high, $bf(x) = 1$.

Balance Criteria

Definition: Let x be a node in a binary tree. We say that the node x **violates the balance criteria** if $|x_h - x_l| > 1$, that is, the heights of the left and right subtrees of x differ by more than 1.

```
struct AVL{
    int data;
    int bfactor;
    AVL* rlink;
    AVL* llink;
};
```

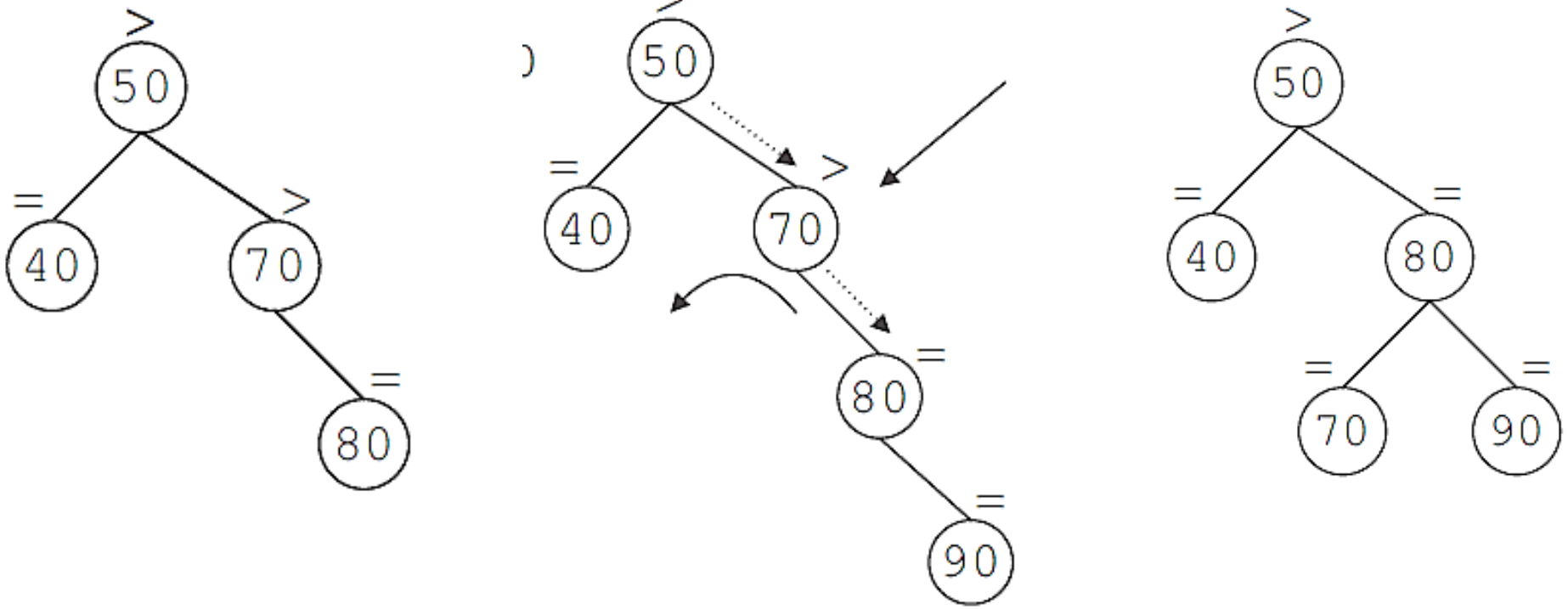
AVL Tree (Operations)

- Because an AVL tree is a binary search tree, the search algorithm for an AVL tree is the same as the search algorithm for a binary search tree.
- Other operations, such as finding the height, determining the number of nodes, checking whether the tree is empty, tree traversal, and so on, on AVL trees can be implemented exactly the same way they are implemented on binary trees.
- However, item insertion and deletion operations on AVL trees are somewhat different from the ones discussed for binary search trees.
- This is because after inserting (or deleting) a node from an AVL tree, the resulting binary tree must be an AVL tree.

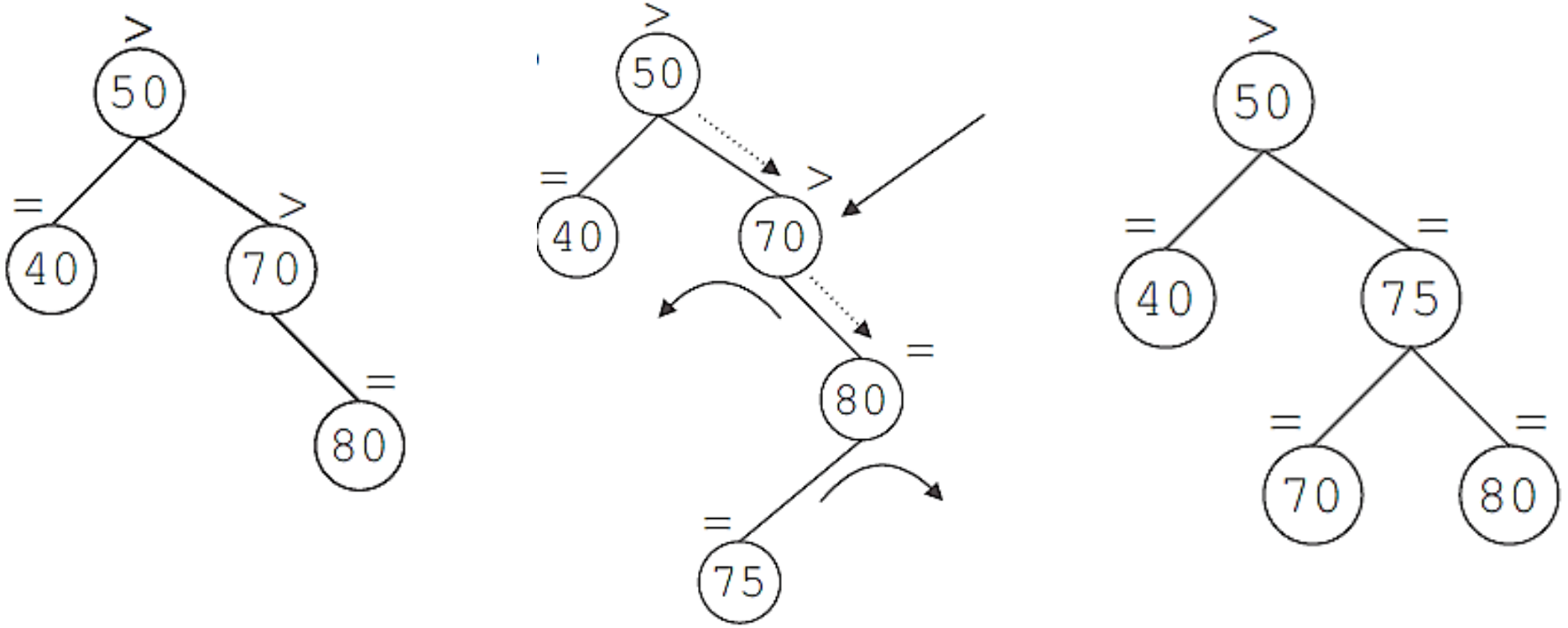
Insertion Operation

- To insert an item in an AVL tree, first we search the tree and find the place where the new item is to be inserted. Because an AVL tree is a binary search tree, to find the place for the new item we can search the AVL tree using a search algorithm similar to the search algorithm designed for binary search trees.
- If the item to be inserted is already in the tree, the search ends at a nonempty subtree. Because duplicates are not allowed, in this case we can output an appropriate error message.
- Suppose that the item to be inserted is not in the AVL tree.
- Then, the search ends at an empty subtree and we insert the item in that subtree.
- After inserting the new item in the tree, the resulting tree might not be an AVL tree.
- Thus, we must restore the tree's balance criteria.
- This is accomplished by traveling the same path, back to the root node, which was followed when the new item was inserted in the AVL tree.
- The nodes on this path (back to the root node) are visited and either their balance factor is changed, or we might have to reconstruct part of the tree.

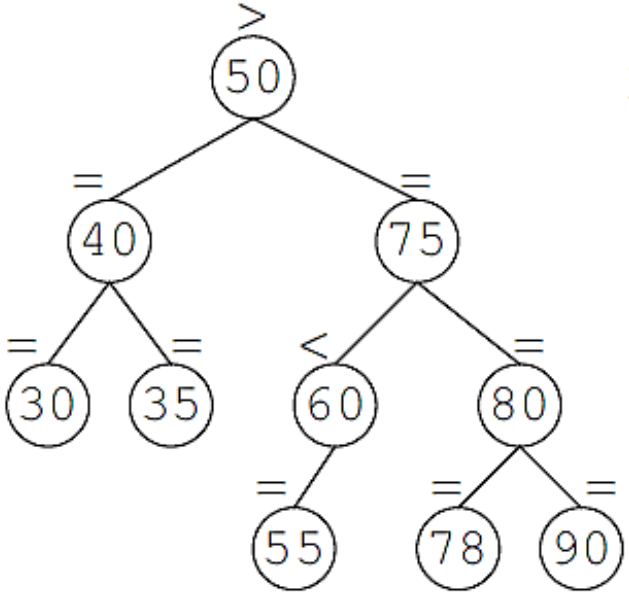
Insertion Operation [Insert 90]

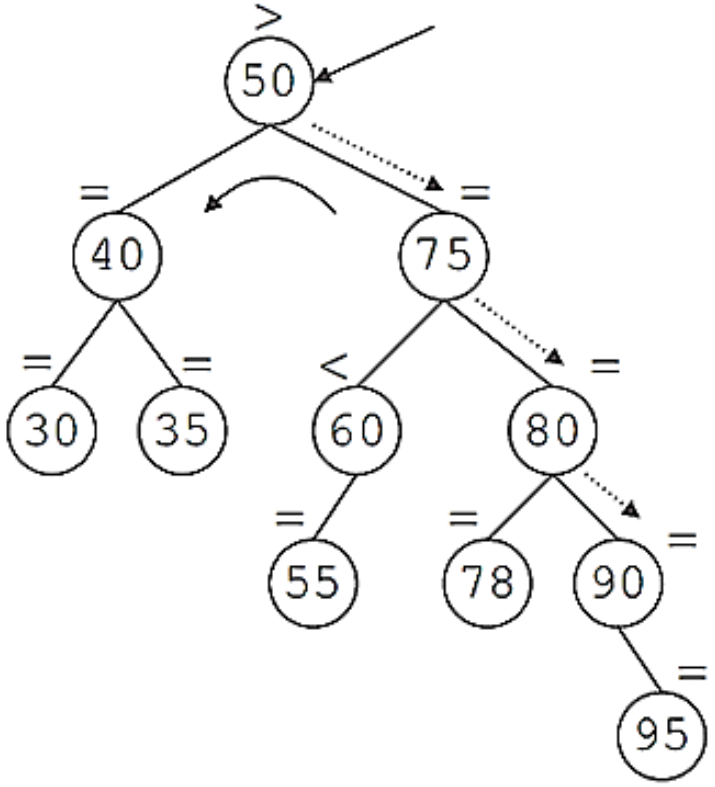


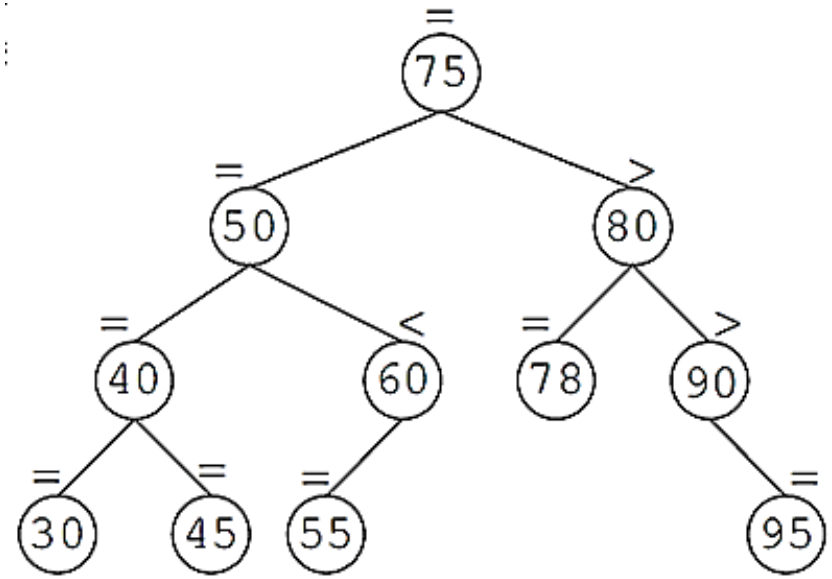
Insertion Operation [Insert 75]



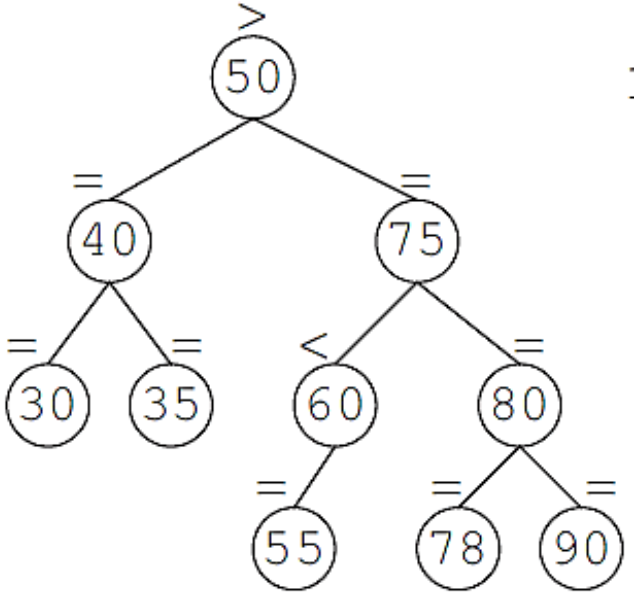
Insertion Operation [Insert 95]

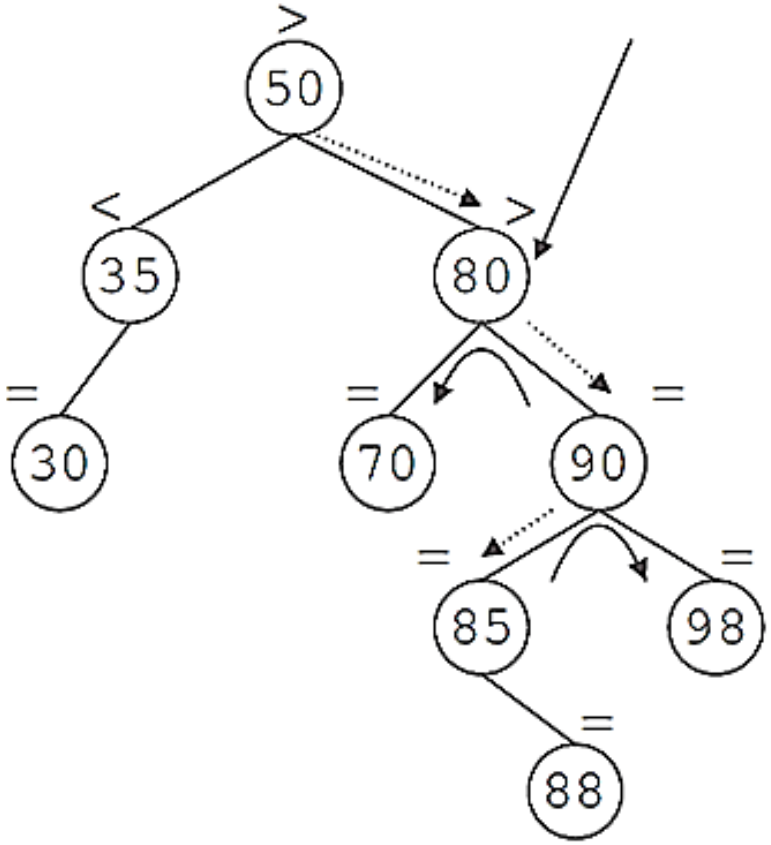


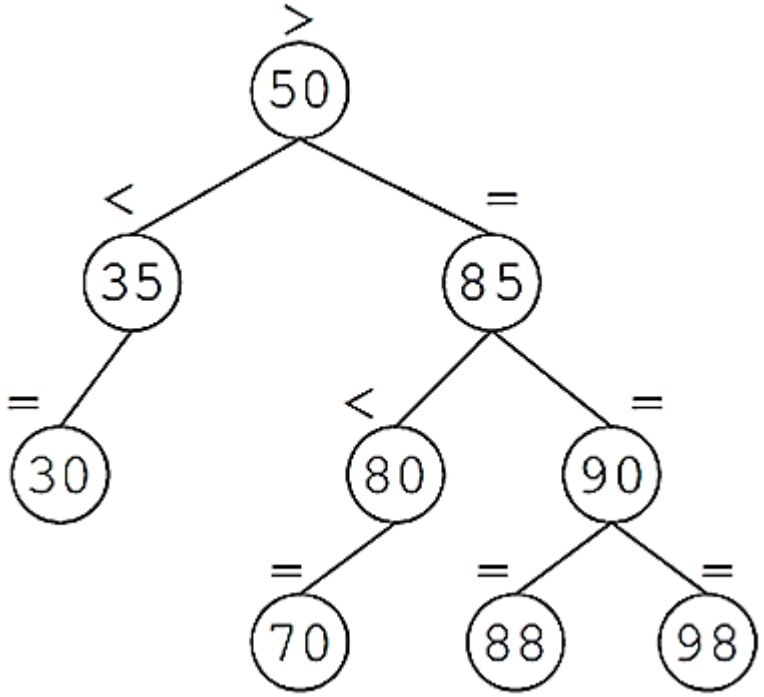




Insertion Operation [Insert 88]







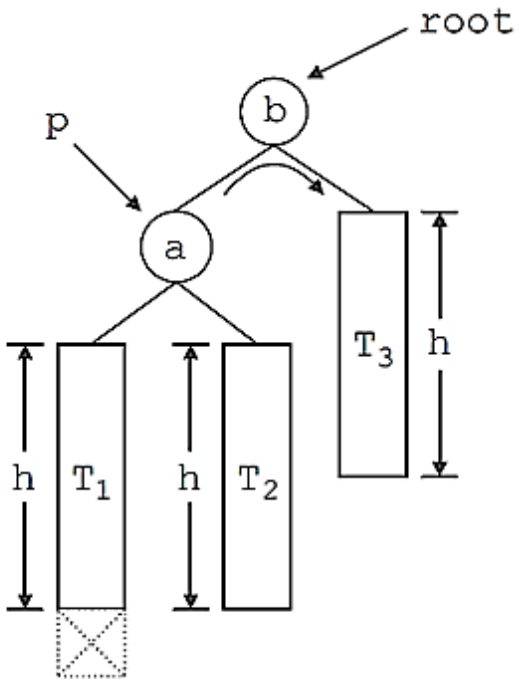
AVL Tree Rebalancing

- Suppose the node to be rebalanced is x.
- There are 4 cases that we might have to fix (two are the mirror images of the other two):
  1. An insertion in the left subtree of the left child of x,
  2. An insertion in the right subtree of the left child of x,
  3. An insertion in the left subtree of the right child of x, or
  4. An insertion in the right subtree of the right child of x.
- Balance is restored by tree *rotations*.

AVL Tree Rotations

- The reconstruction procedure, called rotating the tree.
- There are two types of rotations:
  - left rotation
  - right rotation.
- Suppose the rotation occurs at a node x.
- If it is a ***left rotation***, then
  a) Certain nodes from the right subtree of x move to its left subtree
  b) The root of the right subtree of x becomes the new root of the reconstructed subtree.
- Similarly, if it is a ***right rotation*** at x,
  a) Certain nodes from the left subtree of x move to its right subtree
  b) The root of the left subtree of x becomes the new root of the reconstructed subtree.

Case 1: Rotate right at node b

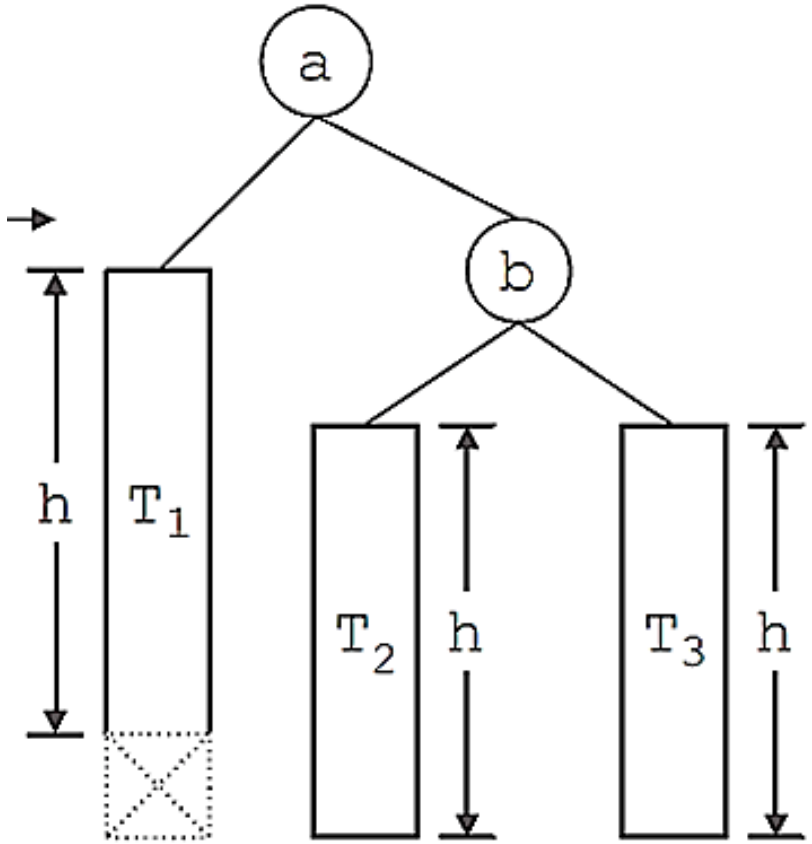


- Subtrees T1, T2, and T3 are of equal height, say h.
- The dotted rectangle shows an item insertion in T1, causing the height of the subtree T1 to increase by 1.
- The subtree at node a is still an AVL tree, but the balance criteria is violated at the root node.
- We note the following in this tree.
- Because the tree is a binary search tree,
  - Every key in subtree T1 is smaller than the key in node a.
  - Every key in subtree T2 is larger than the key in node a.
  - Every key in subtree T2 is smaller than the key in node b.

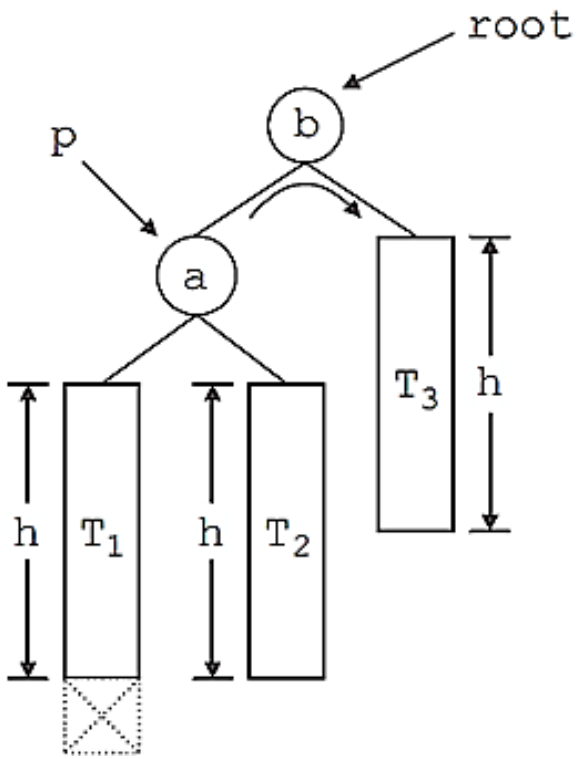
Case 1: Rotate right at node b

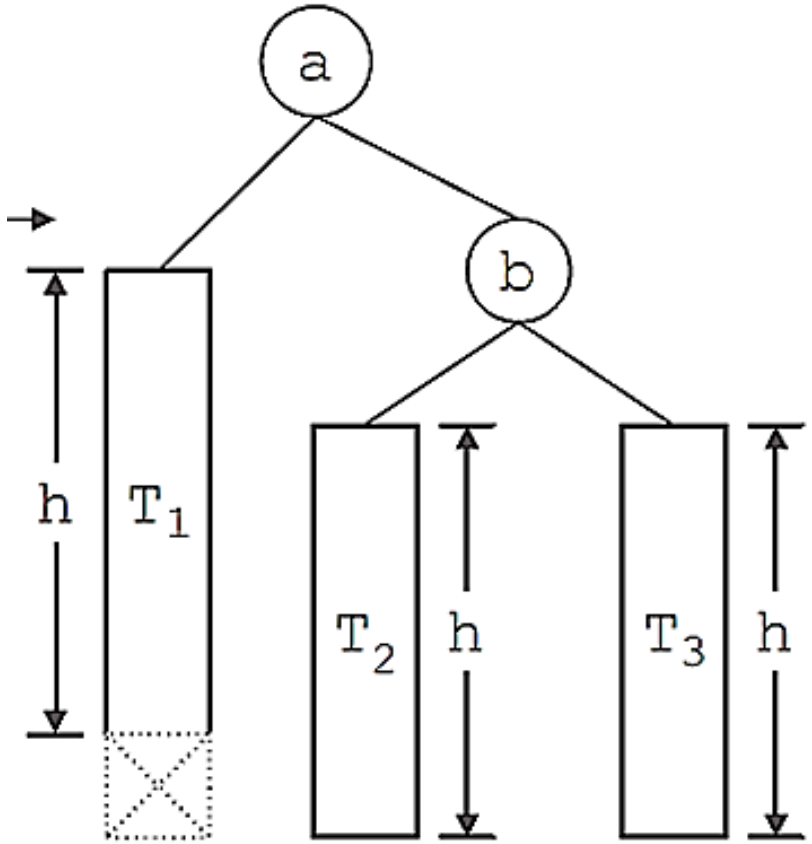
- Therefore,

1) We make T2 (the right subtree of node a) the left subtree of node b.
2) We make node b the right child of node a.
3) Node a becomes the root of the reconstructed tree

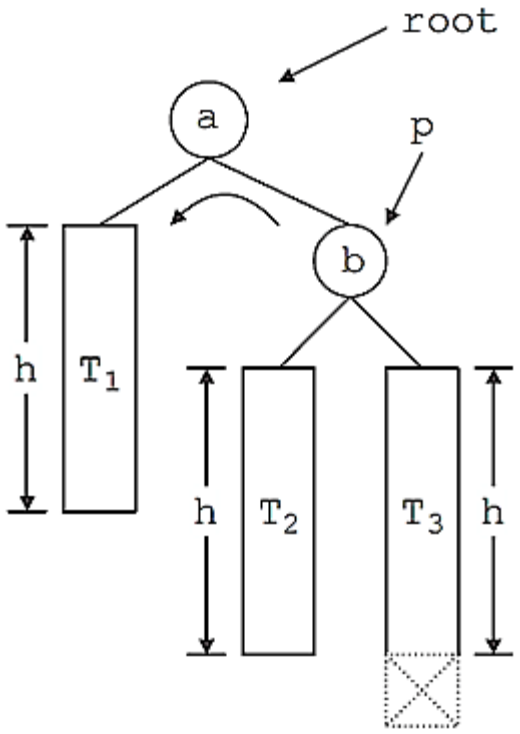


Case 1: Rotate right at node b



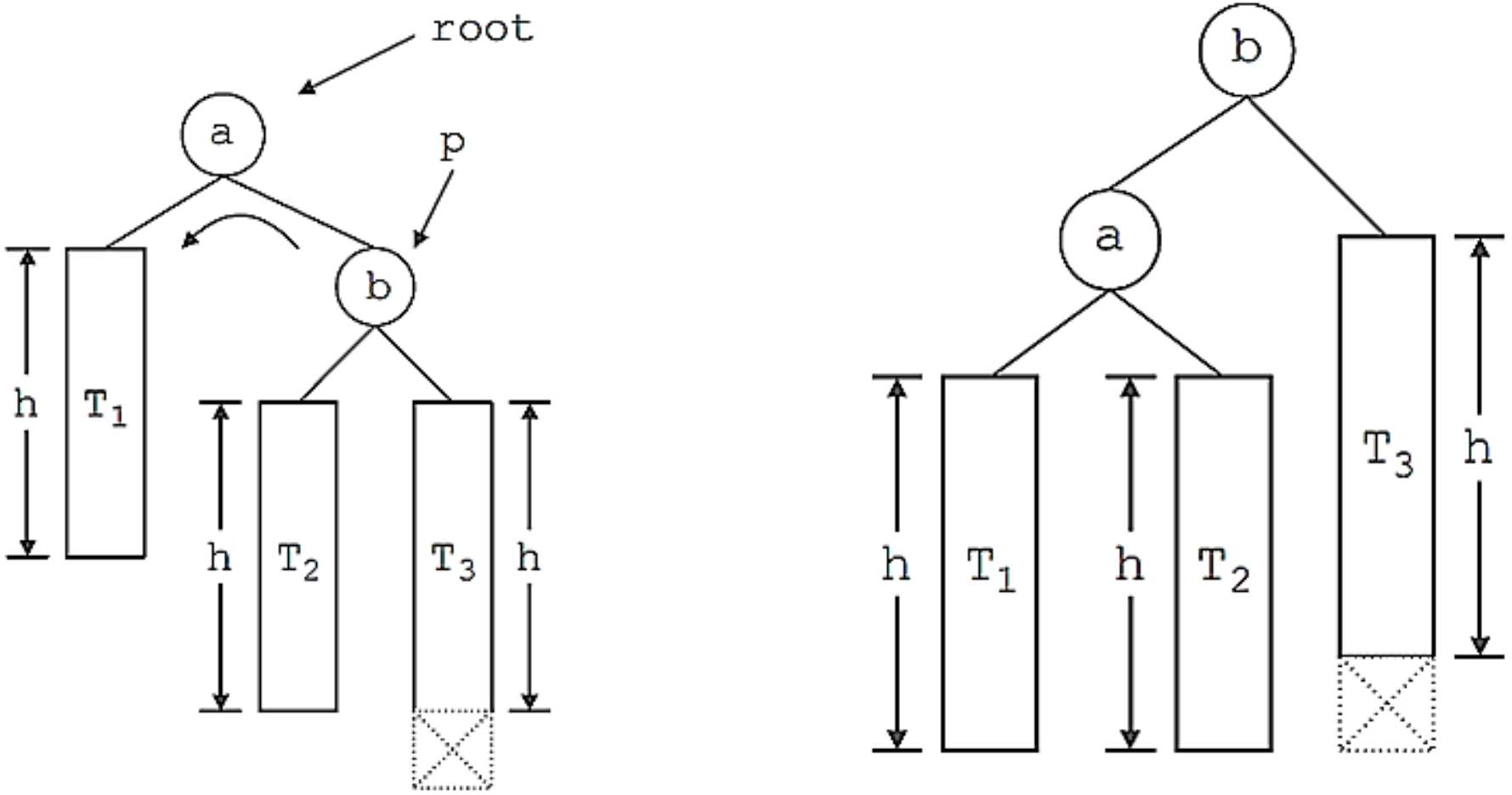


Case 2: Mirror image of Case 1

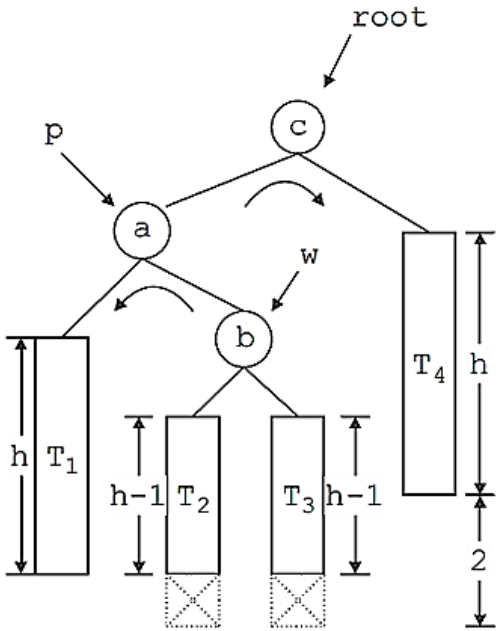


- Subtrees T1, T2, and T3 are of equal height, say h.
- The dotted rectangle shows an item insertion in T3, causing the height of the subtree T3 to increase by 1.
- The subtree at node b is still an AVL tree, but the balance criteria is violated at the root node.
- We note the following in this tree.
- Because the tree is a binary search tree,
  - Every key in subtree T1 is smaller than the key in node a.
  - Every key in subtree T3 is larger than the key in node b.
  - Every key in subtree T2 is smaller than the key in node b.

Case 2: Rotate left at node b



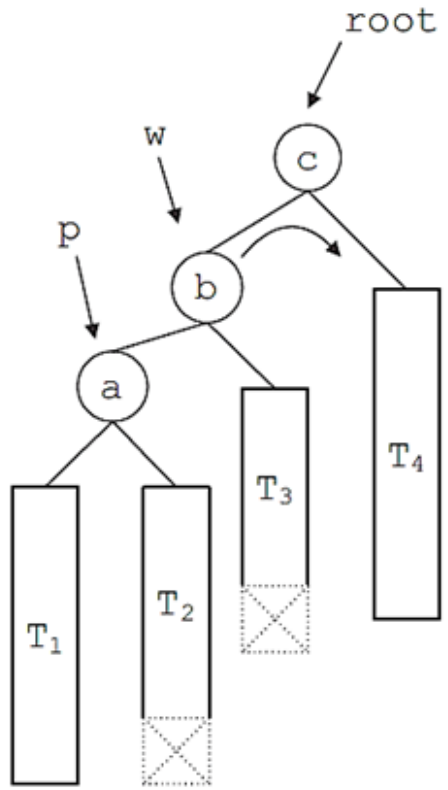
Case 3: Double Rotation



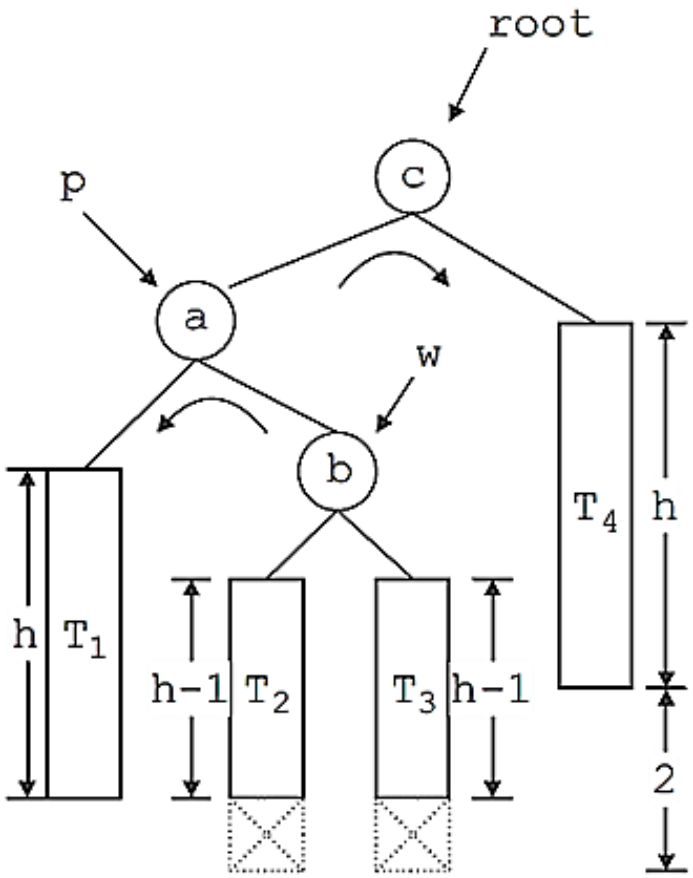
- The new item is inserted either in T2 or T3.
- We note the following (in the tree prior to reconstruction):
  - All keys in T3 are smaller than the key in node c.
  - All keys in T3 are larger than the key in node b.
  - All keys in T2 are smaller than the key in node b.
  - All keys in T2 are larger than the key in node a.
  - After insertion, the subtrees with root nodes a and b are still AVL trees.

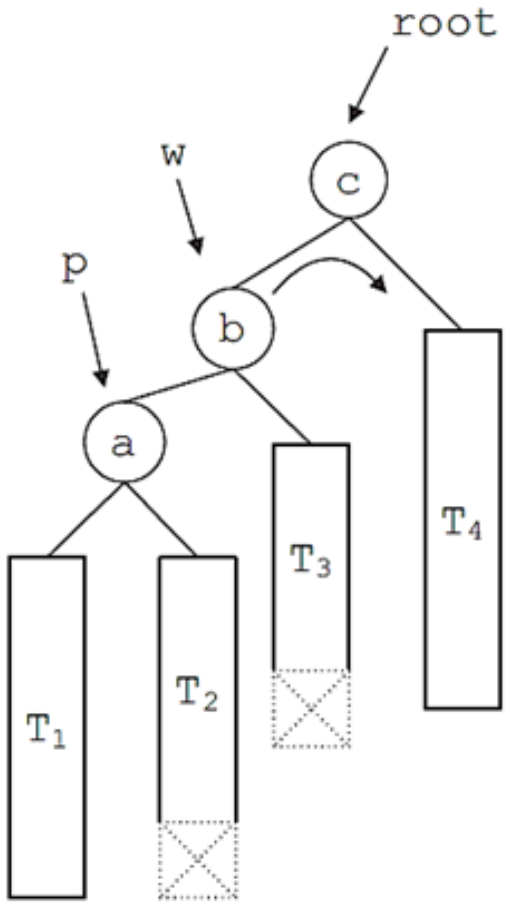
Case 3: Step1- Rotate Left at p

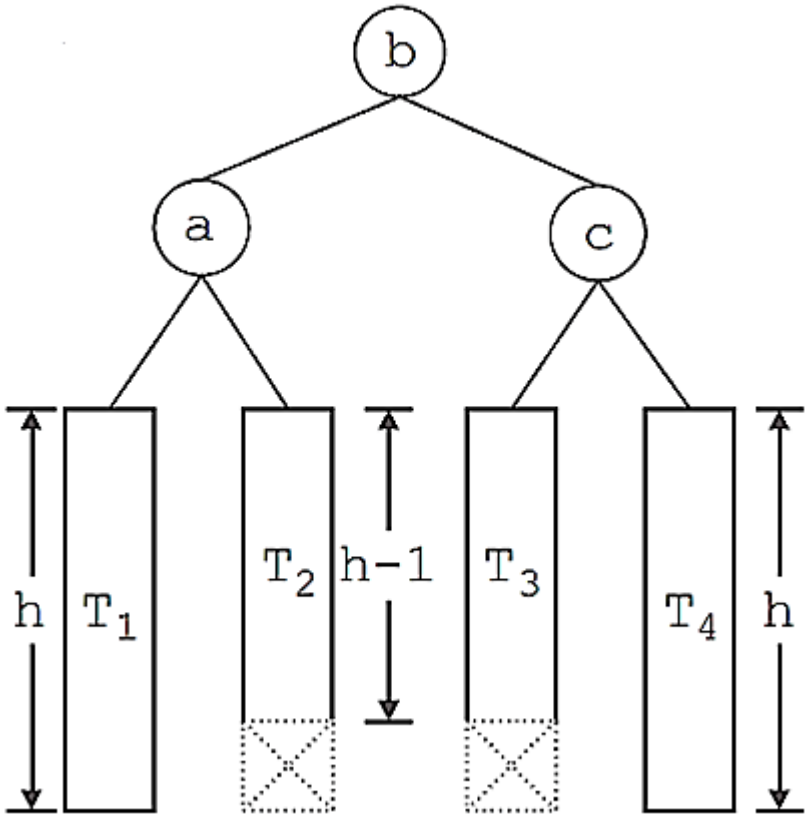
- This is an example of double rotation.
  - One rotation is required at node a, and another rotation is required at node c.
- If the balance factor of the node, where the tree is to be reconstructed, and the balance factor of the higher subtree are opposite, that node requires a ***double rotation***.
- First, we rotate the tree at node a and then at node c.
- Now the tree at node a is right high and so we make a left rotation at a.
- Next, because the tree at node c is left high, we make a right rotation at c.



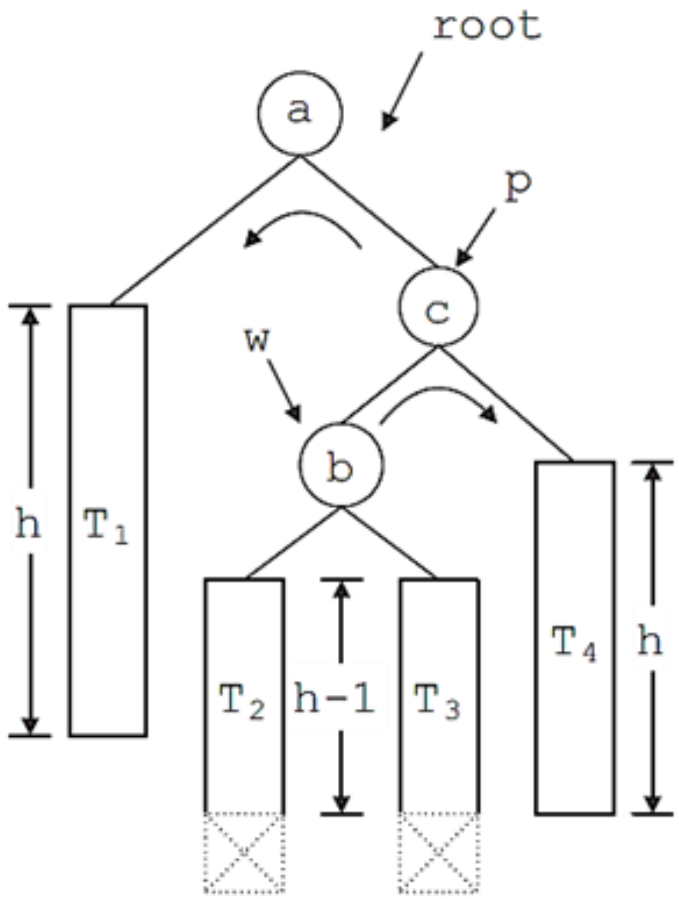
Case 3: Step-2 Rotate Right at root



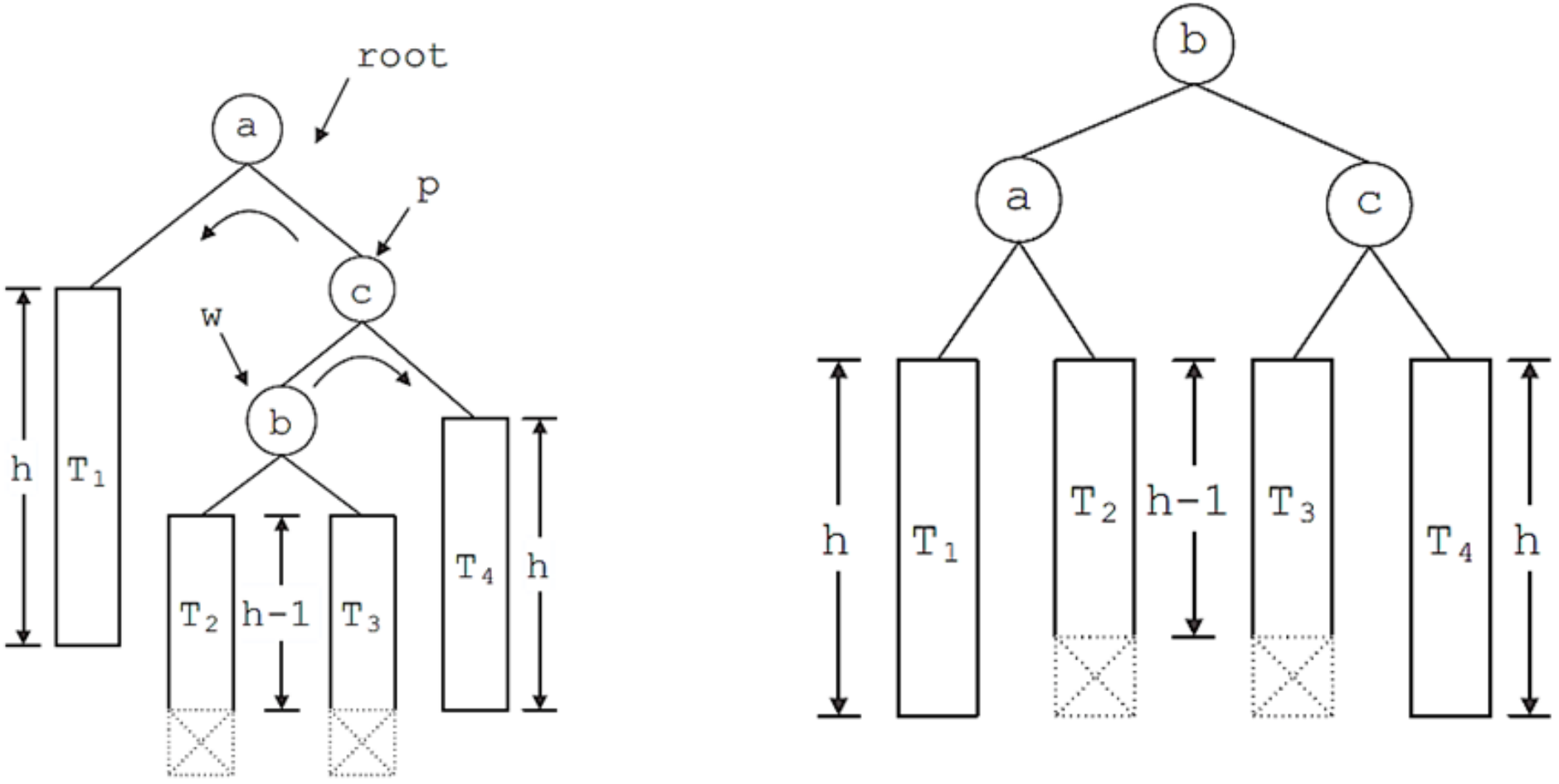




Case 4: Mirror of Case 3



Double rotation: First rotate right at c, then rotate left at a



Wrap it up!

- Suppose that the tree is to be reconstructed, by rotation, at node x.
- Then, the subtree with root node x requires either a single or a double rotation.

1) Suppose that the balance factor of the node x and the balance factor of the root node of the higher subtree of x have the same sign, that is, both positive or both negative.
   a) If these balance factors are positive, make a single left rotation at x. (Prior to insertion, the right subtree of x was higher than its left subtree. The new item was inserted in the right subtree of x, causing the height of the right subtree to increase in height, which in turn violated the balance criteria at x.)
   b) If these balance factors are negative, make a single right rotation at x. (Prior to insertion, the left right subtree of x was higher than its right subtree. The new item was inserted in the left subtree of x, causing the height of the left subtree to increase in height, which in turn violated the balance criteria at x.)

Wrap it up! Continue…

- Suppose that the balance factor of the node x and the balance factor of the higher subtree of x are opposite in sign.
- To be specific, suppose that the balance factor of node x prior to insertion was -1 and suppose that y is the root node of the left subtree of x.
- After insertion, the balance factor of node y is 1.
- That is, after insertion, the right subtree of node y grew in height.
- In this case, we require a double rotation at x.
  - First, we make a left rotation at y (because y is right high).
  - Then, we make a right rotation at x.
- The other case, which is a mirror image of this case, is handled similarly.