# Fundamentals of Data Structure

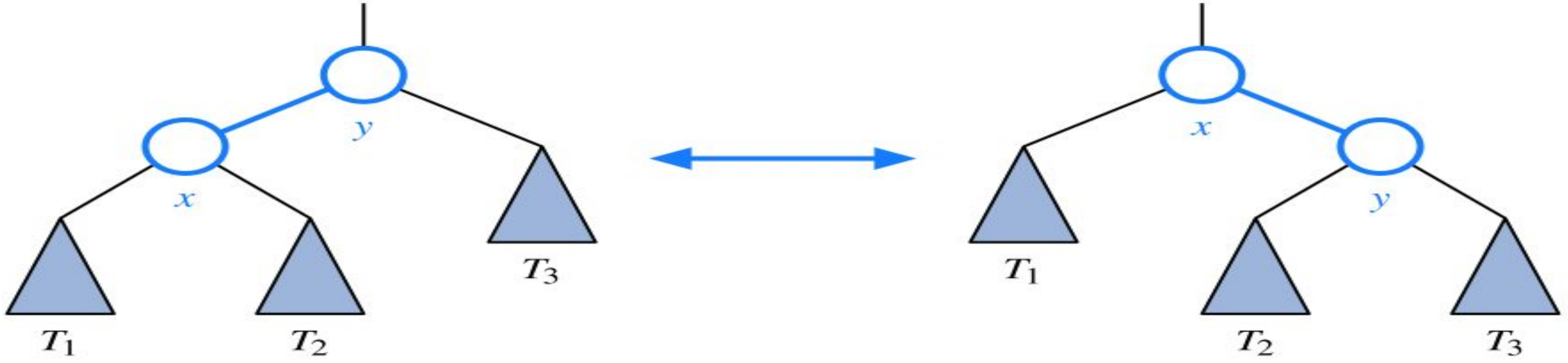
Mahesh Shirole

VJTI, Mumbai-19

Slides are prepared from
1. Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014
2.Data Structures and Algorithms in Java, by Robert Lafore, Second Edition, Sams Publishing

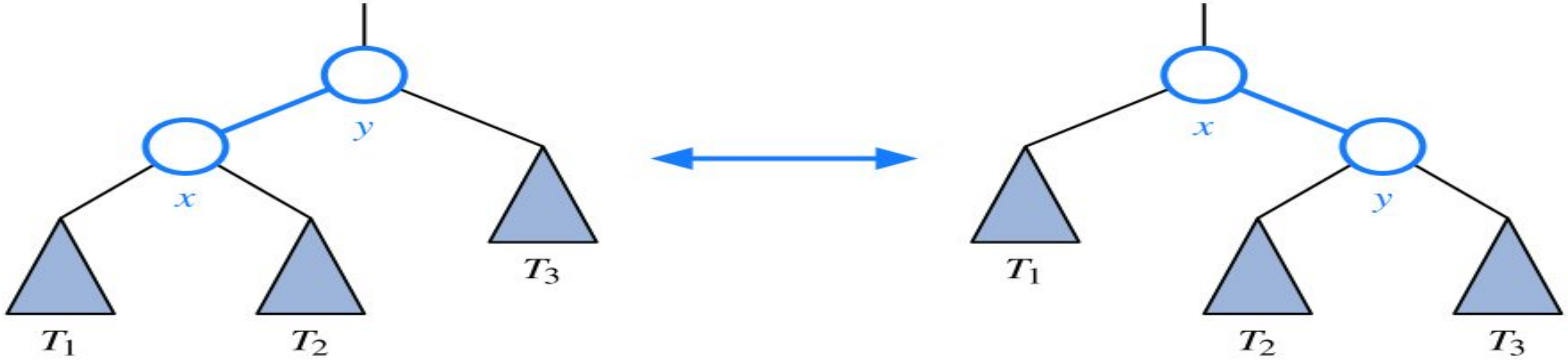
# Balanced Search Trees

- In case of binary search tree, if we could assume a random series of insertions and removals
  - The standard binary search tree supports ***O(logn)*** expected running times for the basic operations
  - However, we may only claim ***O(n)*** worst-case time, because some sequences of operations may lead to an unbalanced tree with height proportional to ***n***
- The primary operation to rebalance a binary search tree is known as a ***rotation***
- During a rotation, we "rotate" a child to be above its parent



# Balanced Search Trees - Rotation

- To maintain the binary search-tree property through a rotation, we note that *if position **x** was a left child of position **y** prior to a rotation (and therefore the key of **x** is less than the key of **y**), then **y** becomes the right child of **x** after the rotation, and vice versa*
- Furthermore, we must relink the subtree of entries with keys that lie between the keys of the two positions that are being rotated
- In Figure the subtree labeled ***T2*** represents entries with keys that are known to be greater than that of position ***x*** and less than that of position ***y***. In the first configuration of that figure, ***T2*** is the right subtree of position ***x***; in the second configuration, it is the left subtree of position ***y***
- Because a single rotation modifies a constant number of parent-child relationships, it can be implemented in ***O(1)*** time with a linked binary tree representation



# Balanced Search Trees - Rotation

- In the context of a tree-balancing algorithm, a rotation allows the shape of a tree to be modified while maintaining the search-tree property
- If used wisely, this operation can be performed to avoid highly unbalanced tree configurations
- One or more rotations can be combined to provide broader rebalancing within a tree
- One such compound operation we consider is a ***trinode restructuring***
- The goal is to restructure the subtree rooted at z in order to *reduce the overall path length* to x and its subtrees, where a position *x,* its parent *y*, and its grandparent *z*

# Balanced Search Trees - Trinode Restructuring

**Algorithm** restructure($x$):

*Input:* A position $x$ of a binary search tree $T$ that has both a parent $y$ and a grandparent $z$

*Output:* Tree $T$ after a trinode restructuring (which corresponds to a single or double rotation) involving positions $x$, $y$, and $z$

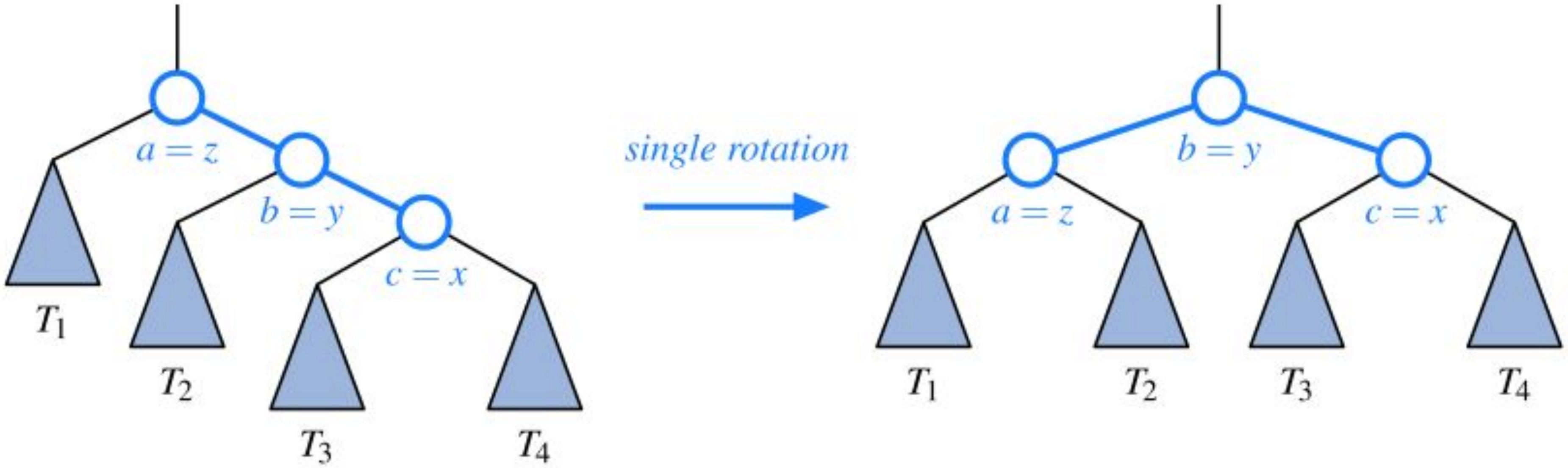
1: Let $(a, b, c)$ be a left-to-right (inorder) listing of the positions $x$, $y$, and $z$, and let $(T_1, T_2, T_3, T_4)$ be a left-to-right (inorder) listing of the four subtrees of $x$, $y$, and $z$ not rooted at $x$, $y$, or $z$.

2: Replace the subtree rooted at $z$ with a new subtree rooted at $b$.

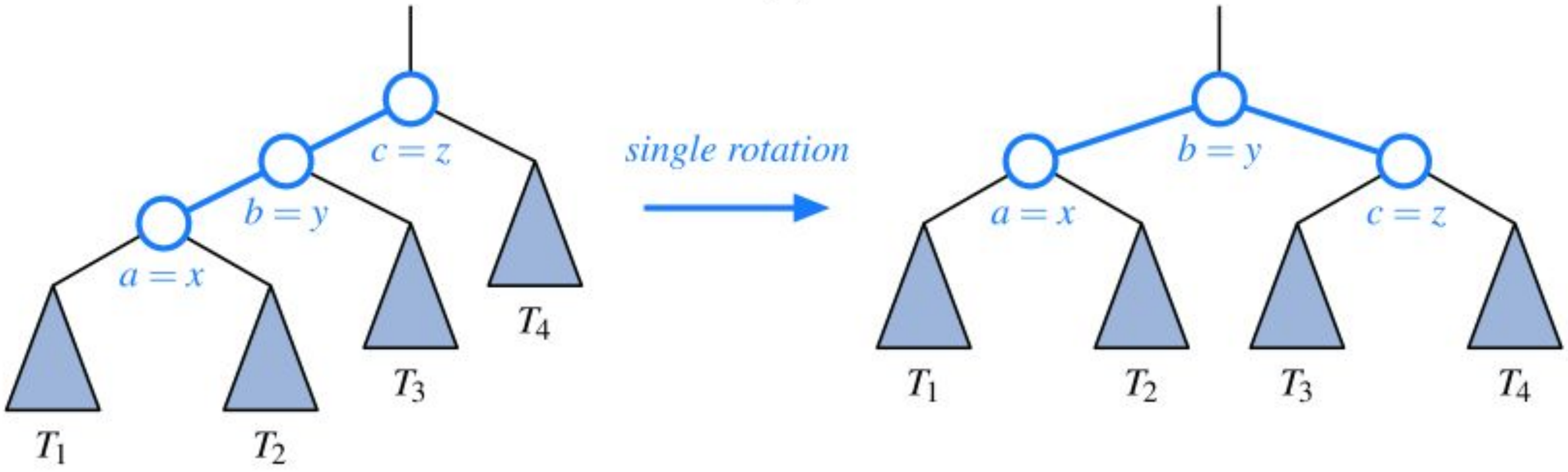
3: Let $a$ be the left child of $b$ and let $T_1$ and $T_2$ be the left and right subtrees of $a$, respectively.

4: Let $c$ be the right child of $b$ and let $T_3$ and $T_4$ be the left and right subtrees of $c$, respectively.

# Trinode Restructuring - Single Rotation

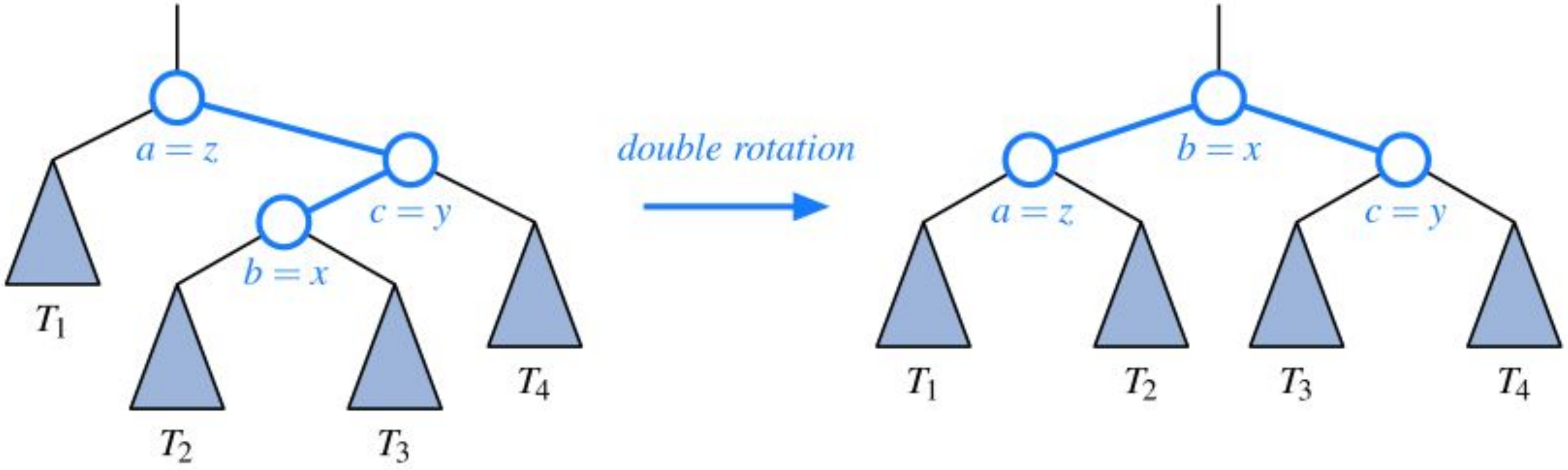



(a)

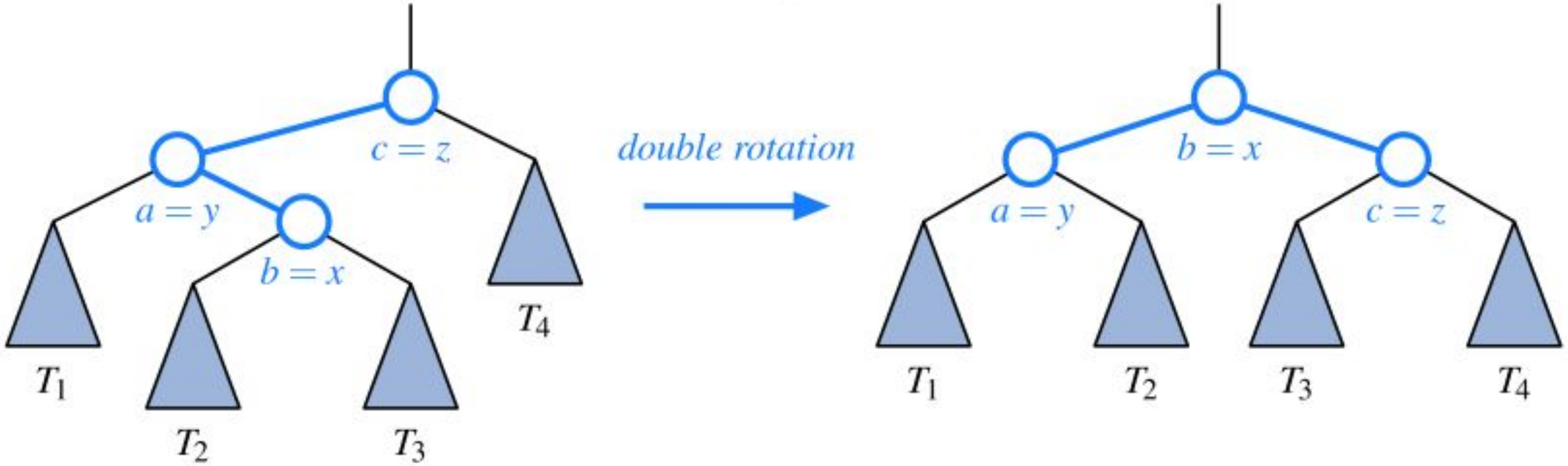



(b)

# Trinode Restructuring - Double Rotation




(c)




(d)

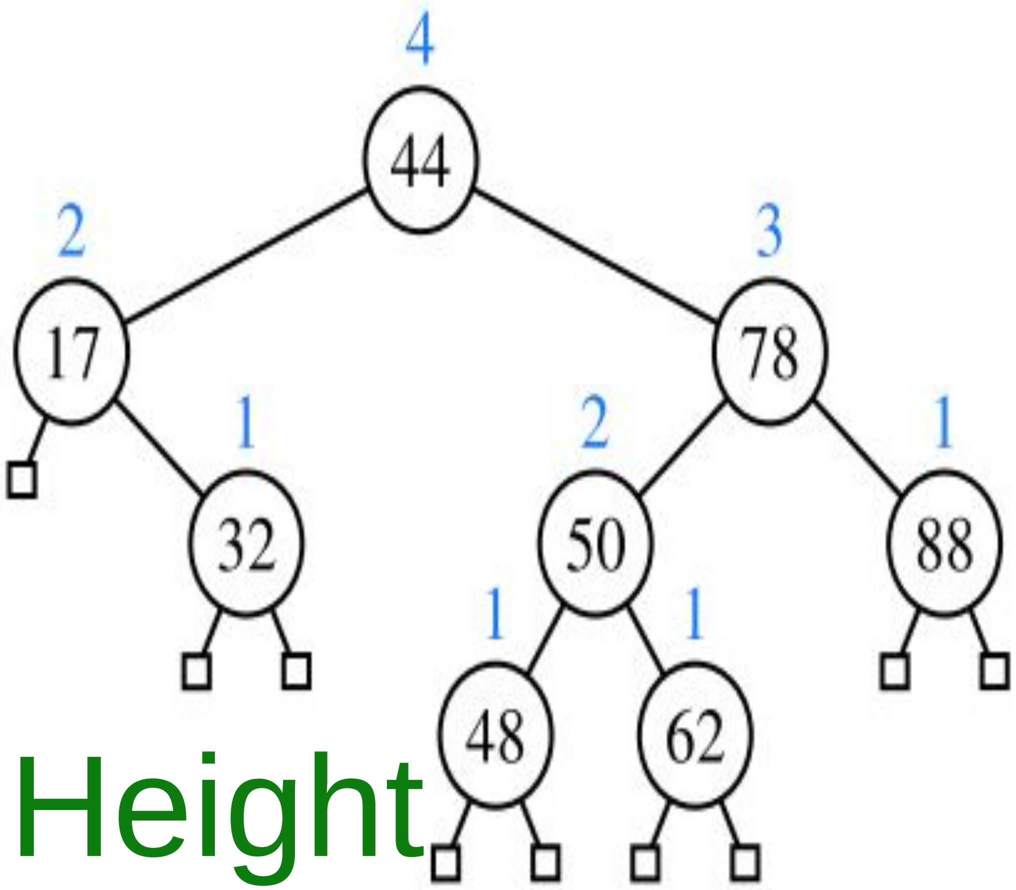
# AVL Trees

- AVL tree is a binary search-tree that will maintain a logarithmic height for the tree
- The height of a subtree rooted at position ***p*** of a tree to be the number of edges on the longest path from ***p*** to a leaf
- By this definition, a leaf position has height ***0***
- The height-balance property characterizes the structure of a binary search tree ***T*** in terms of the heights of its nodes

***Height-Balance Property:*** For every internal position $p$ of $T$, the heights of the children of $p$ differ by at most 1.

# AVL Trees

- Any binary search tree ***T*** that satisfies the height-balance property is said to be an AVL tree, named after the initials of its inventors: *Adel'son-Vel'skii and Landis*
- An immediate consequence of the height-balance property is that a ***subtree of an AVL tree is itself an AVL tree***
- The height-balance property also has the important consequence of keeping the ***height small***




Logarithmic Height

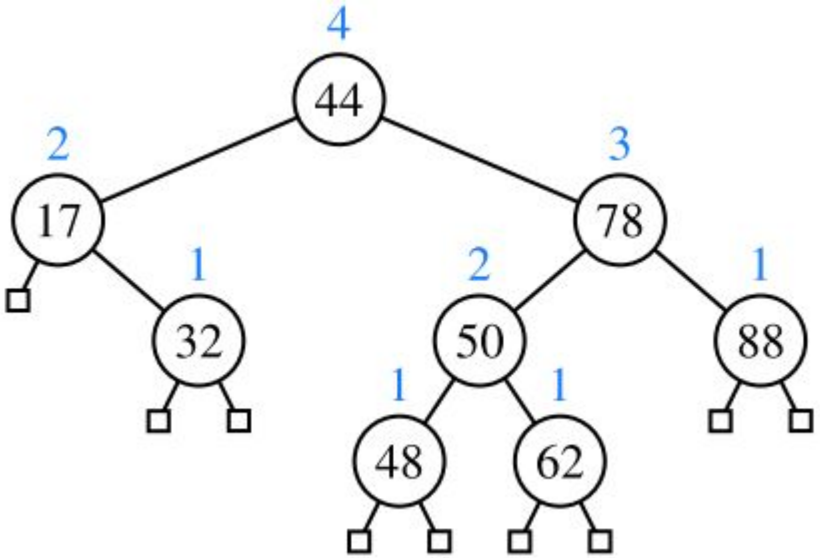
**Proposition 11.1:** *The height of an AVL tree storing $n$ entries is $O(\log n)$.*

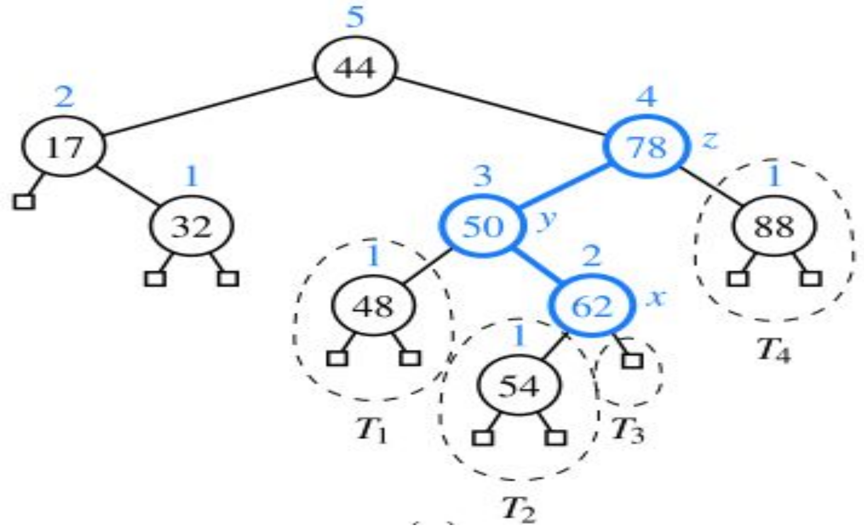
# AVL Trees

- Given a binary search tree ***T***, we say that a position is **balanced** if the absolute value of the *difference between the heights of its children is at most 1*, and we say that it is **unbalanced otherwise**
- Thus, the height-balance property characterizing AVL trees is equivalent to saying that every position is balanced
- The *insertion* and *deletion* operations for AVL trees begin similarly to the corresponding operations for (standard) binary search trees, but with post-processing for each operation to restore the balance of any portions of the tree that are adversely affected by the change

# AVL Trees - Insertion

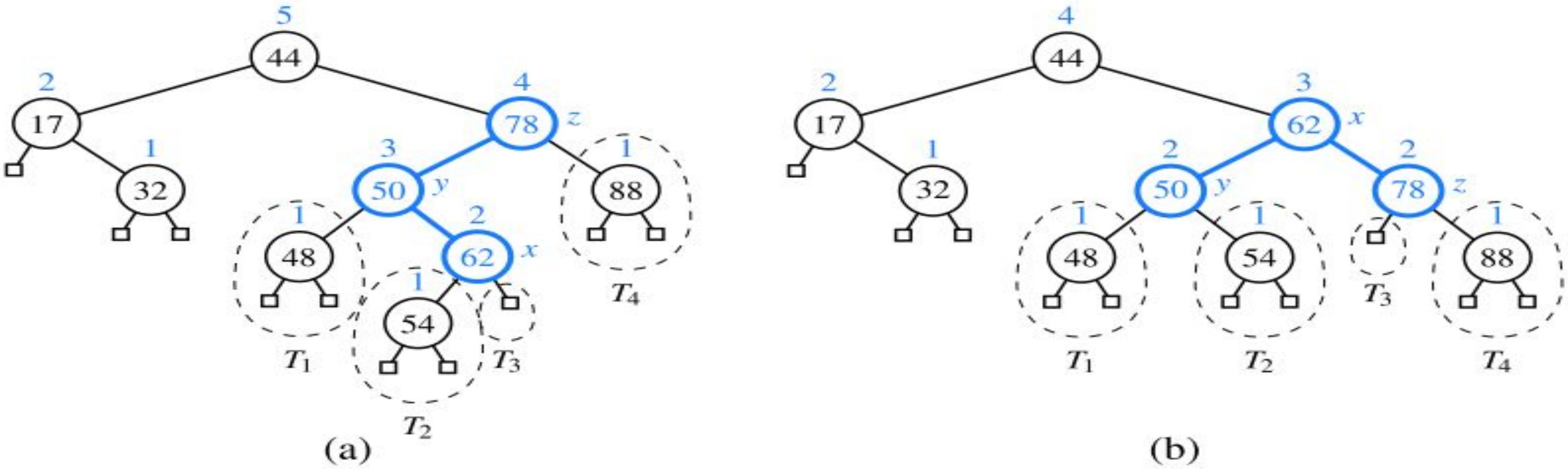
- Suppose that tree T satisfies the height-balance property, and hence is an AVL tree, prior to the insertion of a new entry
- An insertion of a new entry (***key 54***) in a binary search tree results in a leaf position p being expanded to become internal, with two new external children
- This action may violate the height-balance property yet the only positions that may become unbalanced are ancestors of ***p***, because those are the only positions whose subtrees have changed
- After adding a new node for ***key 54***, the nodes storing ***keys 78*** and ***44*** become unbalanced



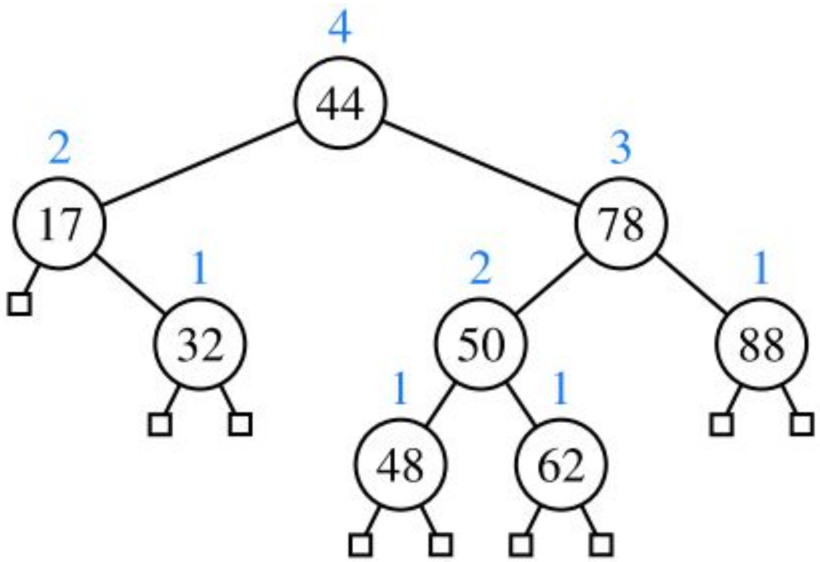
# AVL Trees - Insertion

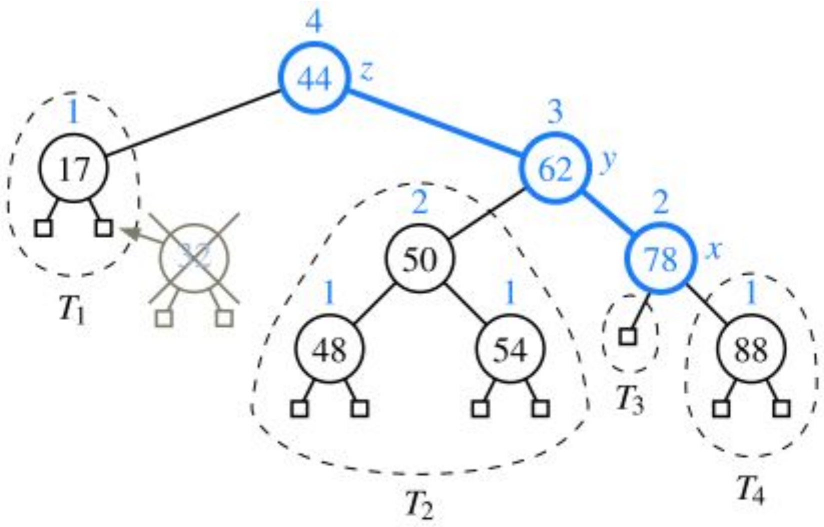
- We restore the balance of the nodes in the binary search tree T by a simple "***search-and-repair***" strategy
- In particular, let ***z*** be the first position we encounter in going up from ***p*** toward the root of ***T*** such that ***z*** is unbalanced
- Also, let ***y*** denote the child of ***z*** with greater height (and note that y must be an ancestor of ***p***)
- Finally, let ***x*** be the child of ***y*** with greater height (there cannot be a tie and position ***x*** must also be an ancestor of ***p***, possibly ***p*** itself)
- We rebalance the subtree rooted at ***z*** by calling the ***trinode restructuring method***, restructure(x)



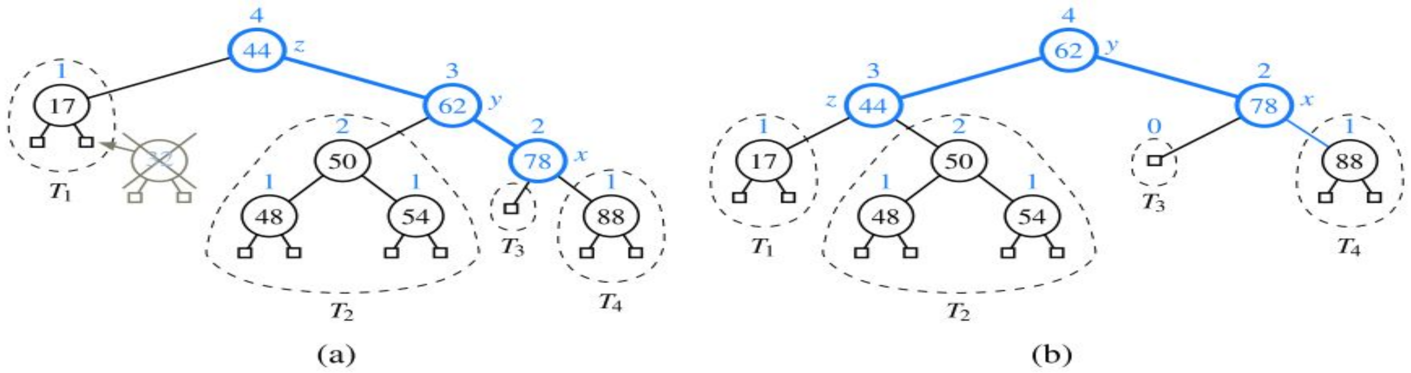
# AVL Trees - Deletion

- A deletion from a regular binary search tree results in the structural removal of a node having either zero or one internal children
- Such a change may violate the *height-balance property* in an AVL tree
- In particular, if position ***p*** (*key 32*) represents a (possibly external) child of the removed node in tree ***T***, there may be an unbalanced node on the path from ***p*** to the root of ***T***
- In fact, there can be at most one such unbalanced node



# AVL Trees - Deletion




- As with insertion, we use **trinode restructuring** to restore balance in the tree ***T***
- In particular, let ***z*** be the first unbalanced position encountered going up from ***p*** toward the root of ***T***, and let ***y*** be that child of ***z*** with greater height (***y*** will not be an ancestor of ***p***)
- Furthermore, let ***x*** be the child of ***y*** defined as follows:
  - if one of the children of ***y*** is taller than the other, let ***x*** be the taller child of ***y***;
  - else (both children of ***y*** have the same height), let ***x*** be the child of ***y*** on the same side as ***y*** (that is, if ***y*** is the left child of ***z***, let ***x*** be the left child of ***y***, else let ***x*** be the right child of ***y***)
- The restructured subtree is rooted at the middle position denoted as ***b*** in the description of the trinode restructuring operation. The height-balance property is guaranteed to be locally restored within the subtree of ***b***
- Unfortunately, this trinode restructuring may reduce the height of the subtree rooted at ***b*** by 1, which may cause an ancestor of ***b*** to become unbalanced
- So, after rebalancing ***z***, we continue walking up ***T*** looking for unbalanced positions
- If we find another, we perform a restructure operation to restore its balance, and continue marching up ***T*** looking for more, all the way to the root

# Performance of AVL Trees

| Method | Running Time |
|---|---|
| size, isEmpty | $O(1)$ |
| get, put, remove | $O(\log n)$ |
| firstEntry, lastEntry | $O(\log n)$ |
| ceilingEntry, floorEntry, lowerEntry, higherEntry | $O(\log n)$ |
| subMap | $O(s + \log n)$ |
| entrySet, keySet, values | $O(n)$ |