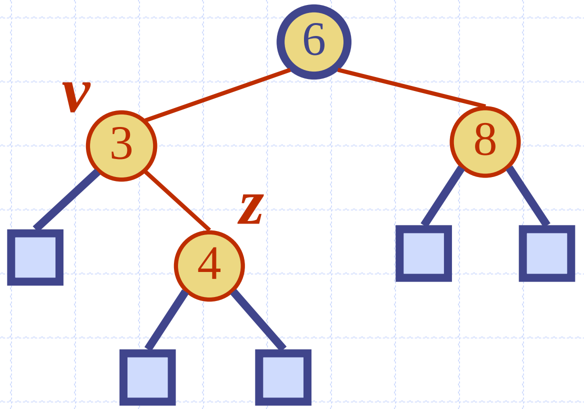
Heavily modified version of the presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

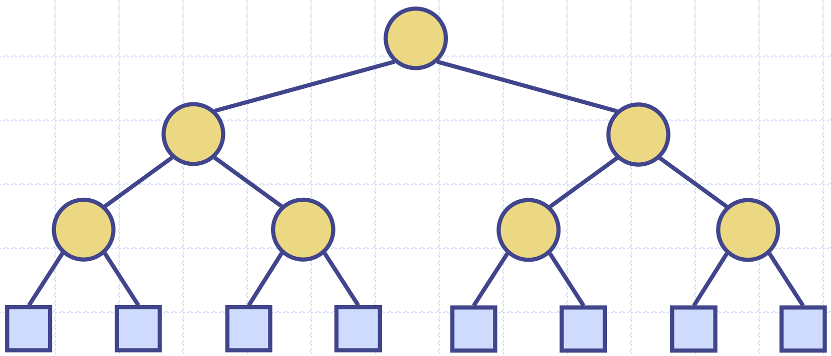
AVL Trees

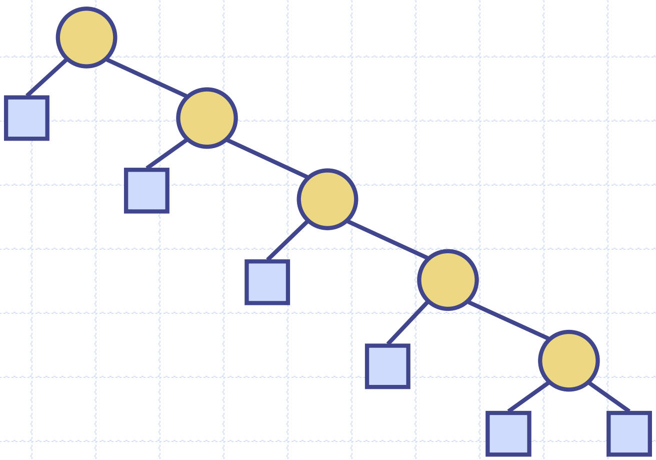


The book and the slides are poor so make sure you have my versions or use other sources

Balanced Trees

- Recall that basic operations on binary search trees are $O(h)$ where h is the height of the tree
- The height, h, itself is between $O(\log n)$ and $O(n)$





- A "balanced" tree is preferred for efficiency
- However, some sequences of insert and delete operations may leave the tree unbalanced
- We want to maintain the **logarithmic height**!

Balance of a node

- The balance factor of a node is defined by:

$$balance\_factor(x) = height(left(x)) - height(right(x))$$

- A single node is said to be **balanced** if the difference between the heights of its children is at most 1, and **unbalanced** otherwise (i.e. $\left|balance_{factor(x)}\right| < 1$)

- In a balanced tree we expect the balance factor of nodes to be small

AVL Trees

- A "self-balancing" binary search tree
  - Most likely the first such data structure
  - Named after its inventors **A**delson-**V**elsky and **L**endis
  - In AVL trees, each node is balanced and they are kept that way
- Self-balancing: Keep its height "small" in the face of arbitrary insert and delete operations

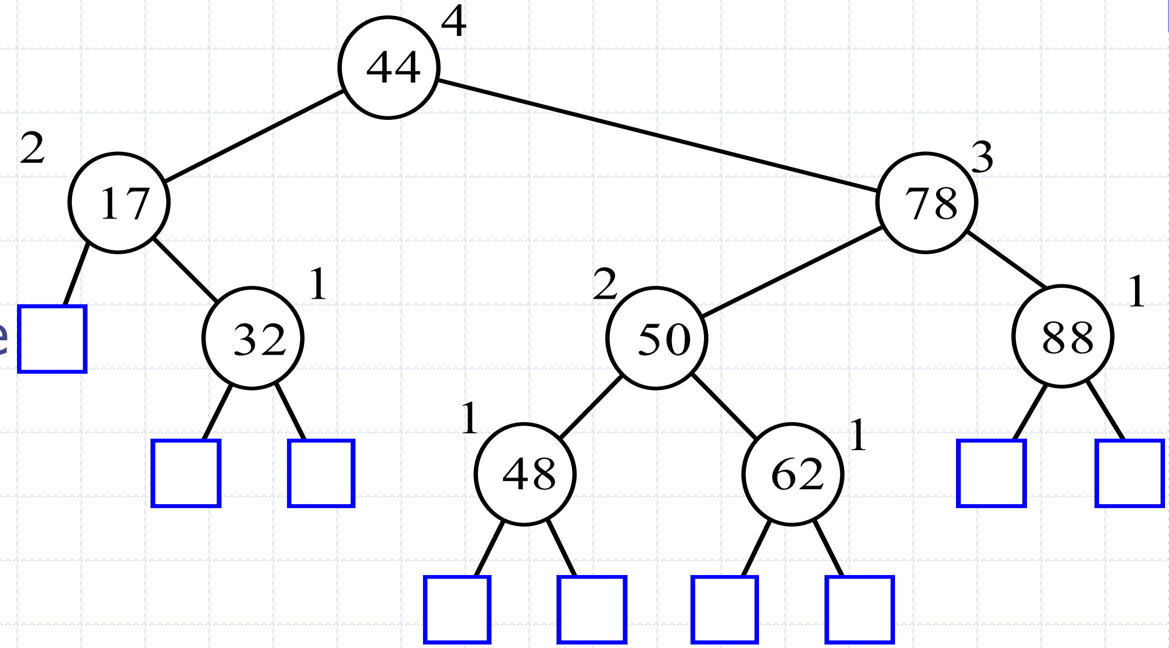
- How? – Tree Rotations!

AVL Trees

AVL Trees are balanced binary search trees that satisfy the height-balance property:

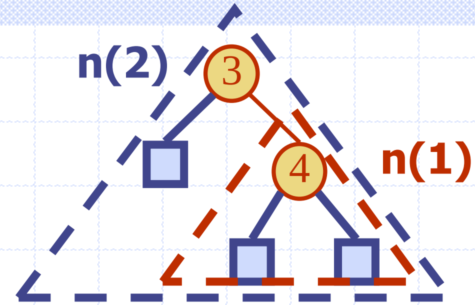
For every internal node v of T, heights of the children of v can differ by at most 1

Any subtree of an AVL tree, is itself an AVL tree



An example of an AVL tree where the heights are shown next to the nodes

Height of an AVL Tree



Fact: The height of an AVL tree storing n keys is O(log n).

Proof (by induction): 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 n-1 and another of height n-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),
  $n(h) > 2^i n(h-2i)$
- 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)

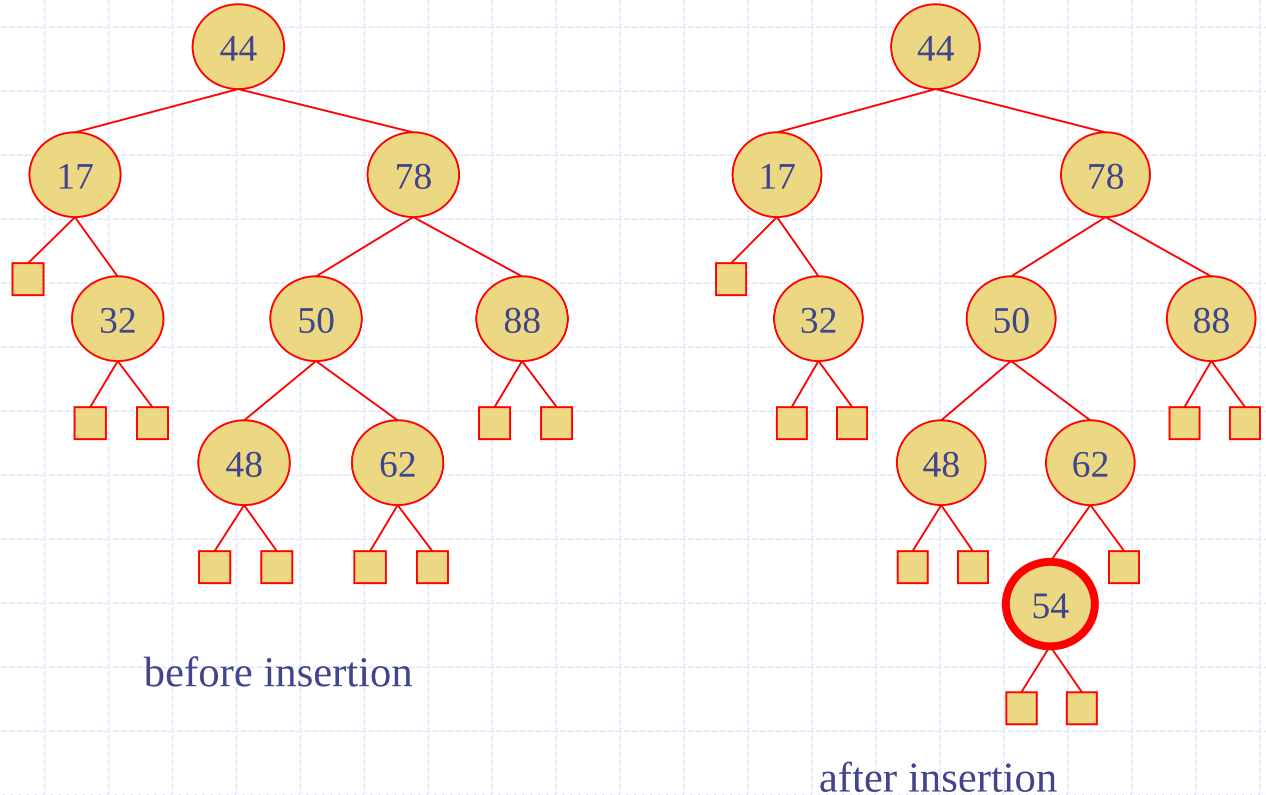
AVL Tree Operations

- Search operation is the same as the binary search tree version and it does not affect the balance

- In AVL Trees, insert and delete operations of the regular binary search trees follow a **post-processing** stage to restore balance since these operations may break the balance of an AVL Tree

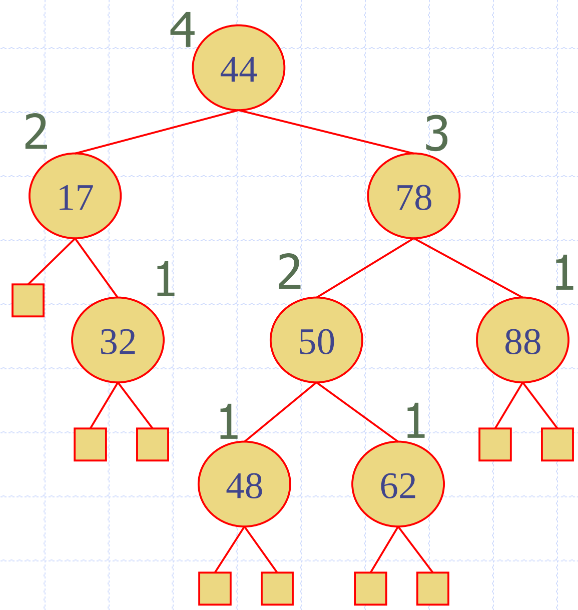
Insertion

- Insertion is as in a binary search tree
- Always done by expanding an external node.
- Example:

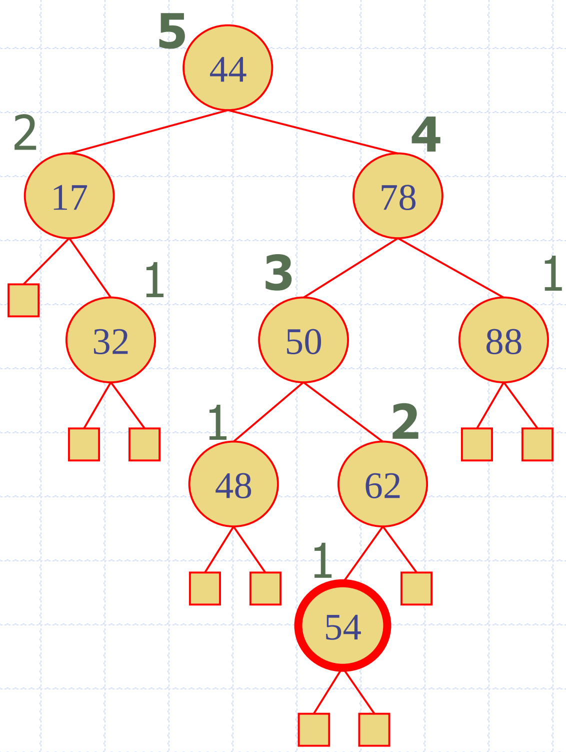


Insertion

- Observation: Only the ancestors of a node is affected by an operation (i.e. the nodes from root to the operated on node)



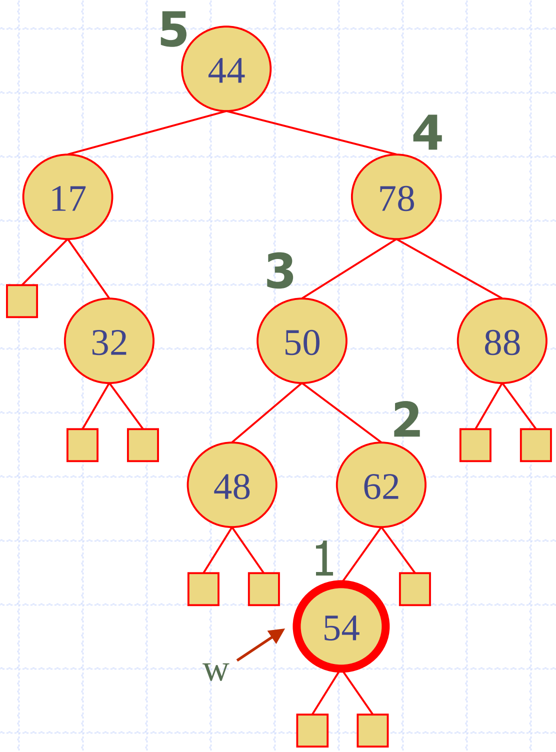
before insertion



after insertion

Insertion Approach

- Insert the node as in binary search trees
- Go up towards the root, checking whether the balance is broken or not
- When an imbalance is found, fix it by **rotating** the tree (rotation is also called **trinode restructuring** in the book)



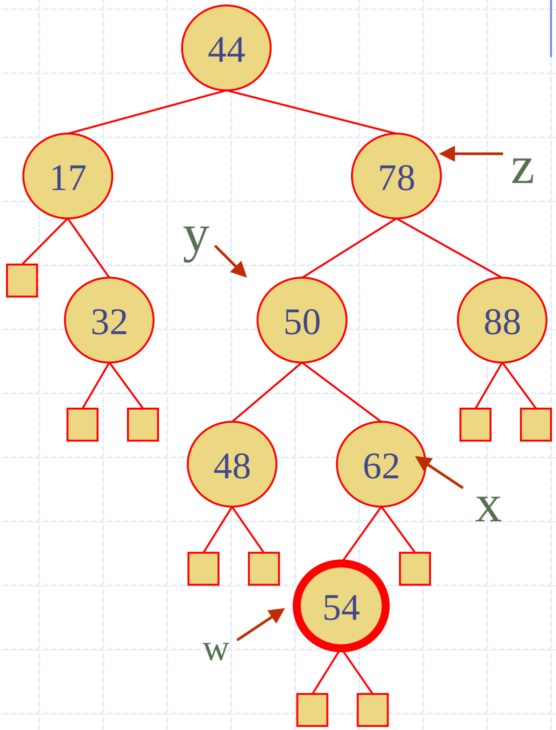
Detecting Imbalance

- Add an auxiliary element to the node structure indicating the height of the tree
- Recalculate it for the affected nodes after insertion/deletion
- Check the height difference between the children of the affected nodes

Rotations: Trinode Restructuring

- Why does the book call it **tri**node restructuring?

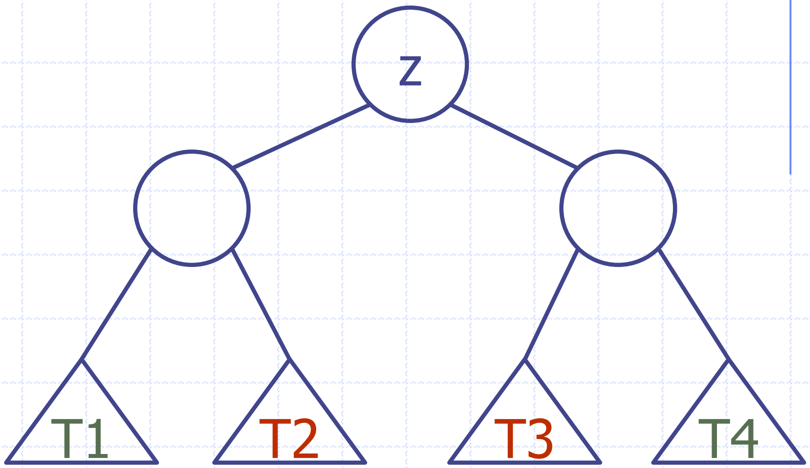
- Let z be the first node on the way to root that is imbalanced, y be the child of z with the greater height and x be the child of y with the greater height
- The idea is to get the in-order listing of these nodes (i.e. the sorted order) and to make the middle one the topmost node
- We do this by tree rotations



Insertion: 4 Cases

- Let z be the node where an imbalance occurs:

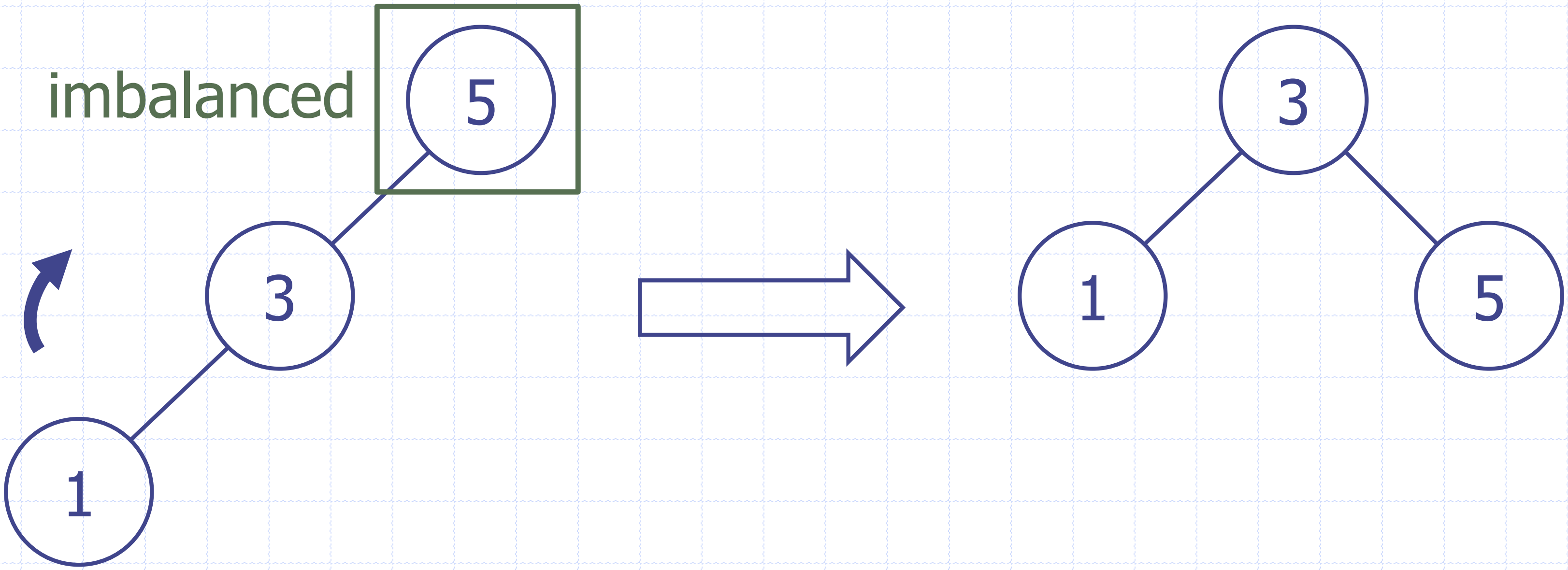
1. left subtree of left child of z
2. right subtree of left child of z
3. left subtree of right child of z
4. right subtree of right child of z



- 1&4 are solved by a **single rotation**
- 2&3 are solved by a **double rotation**

- What the book refers to as trinode restructuring includes both single and double rotations

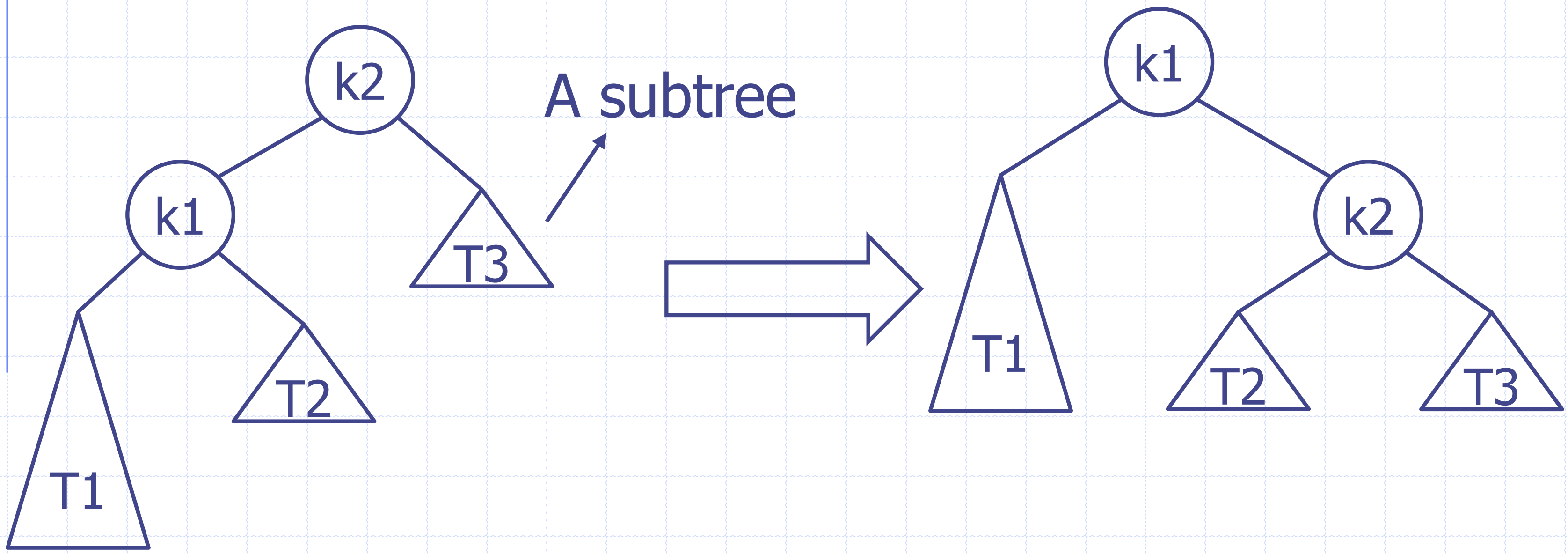
Single Rotation: A Simple Case



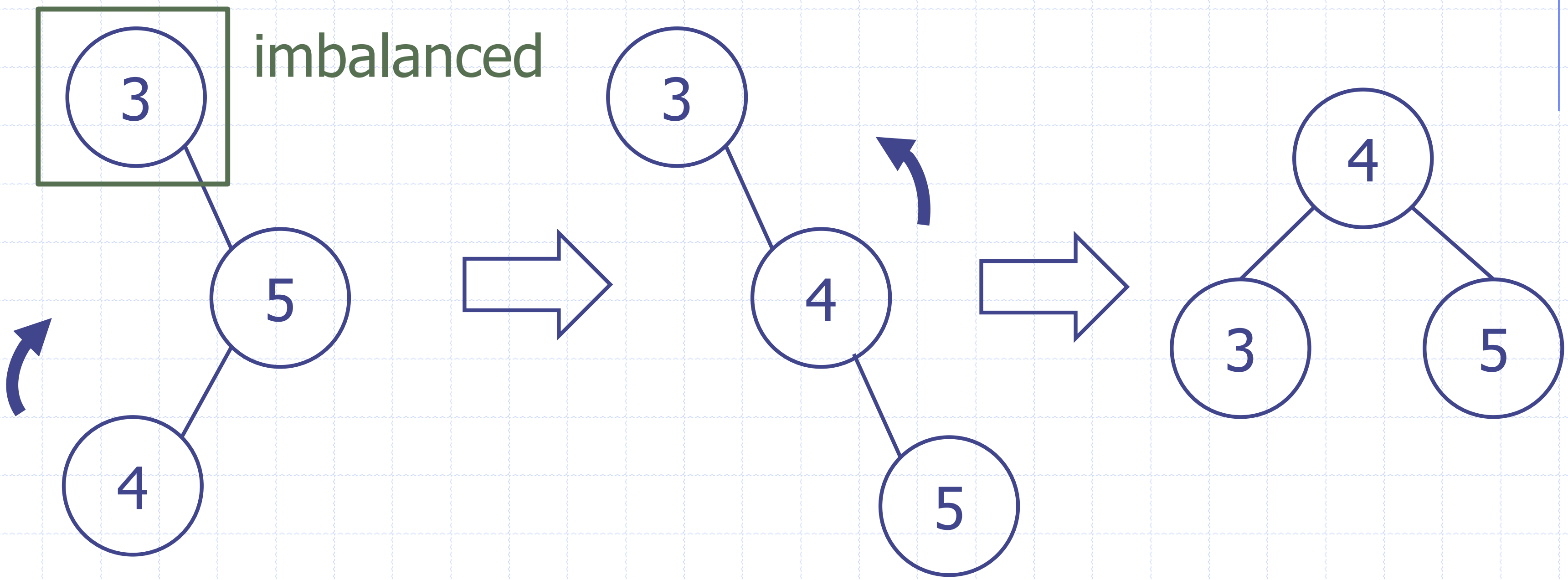
Insert 1

A single rotation:
Rotate between a
node and its child

Single Rotation: General Case

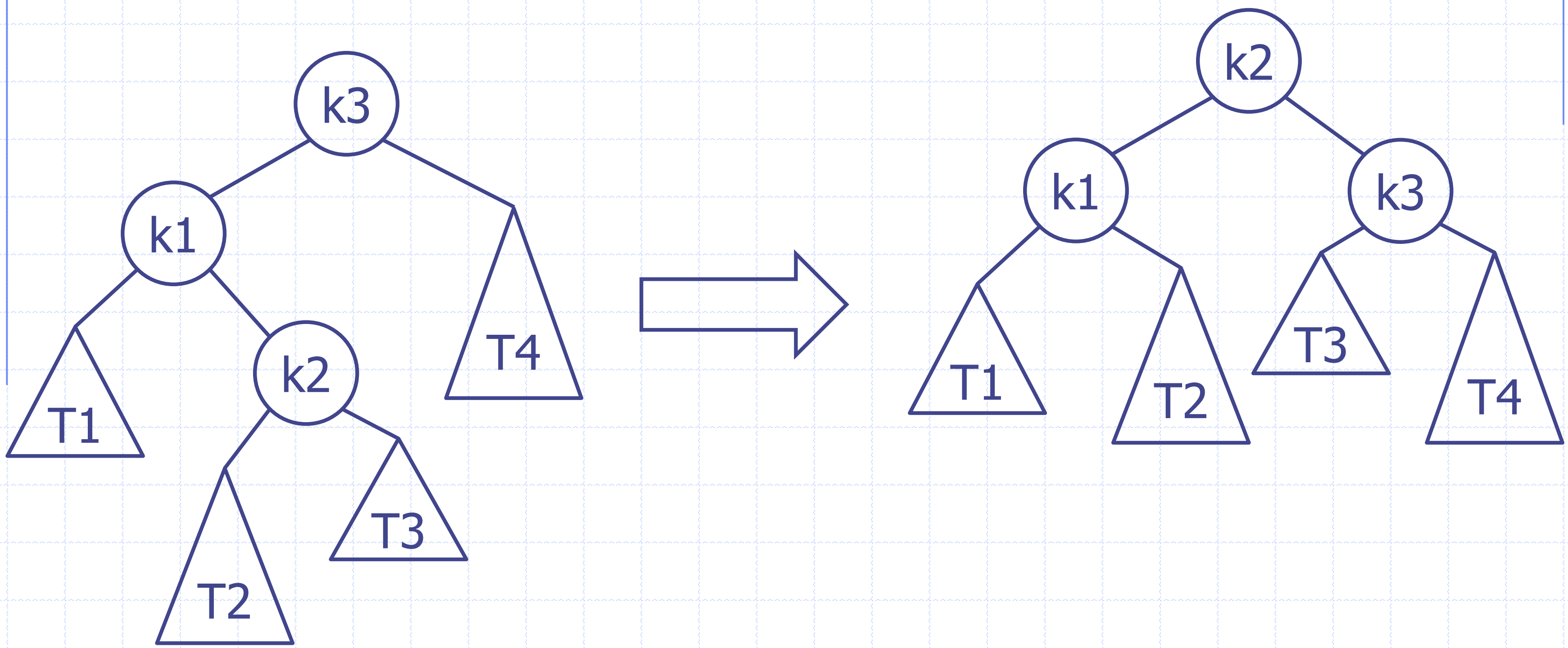


Double Rotation: A Simple Case



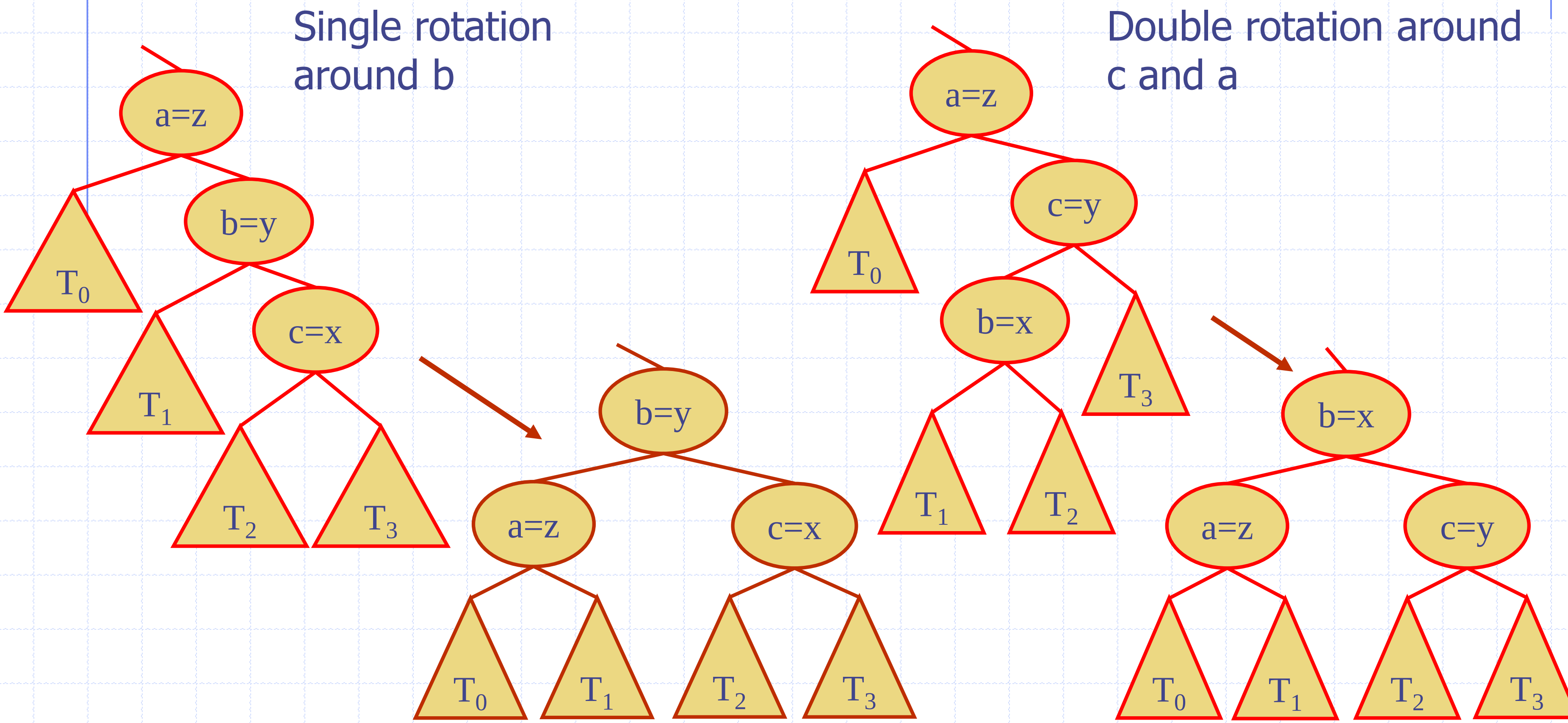
Insert 4

Duble Rotation: General Case



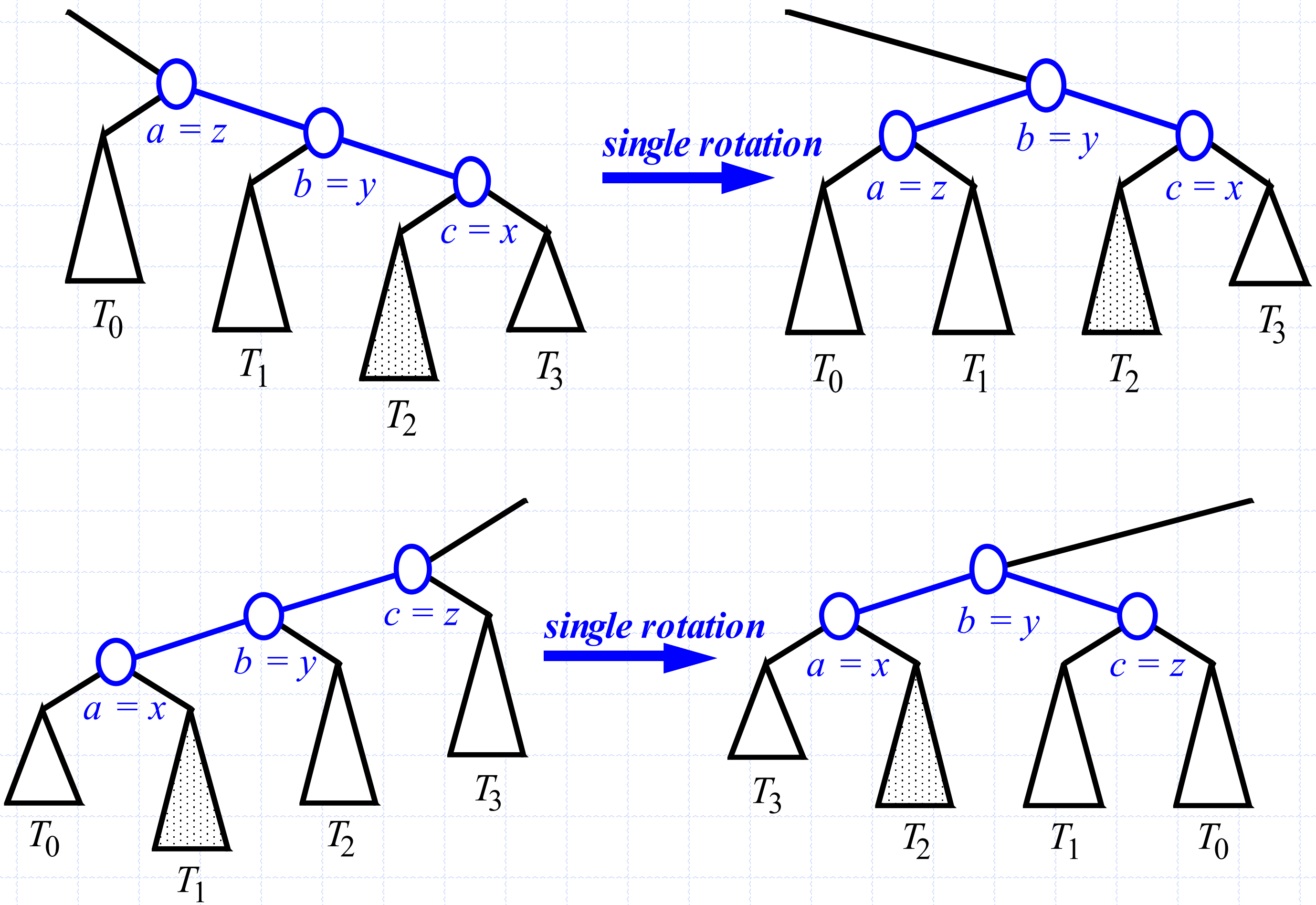
Book’s Slide: Trinode Restructuring

- Let (a,b,c) be the inorder listing of x, y, z
- Perform the rotations needed to make b the topmost node of the three



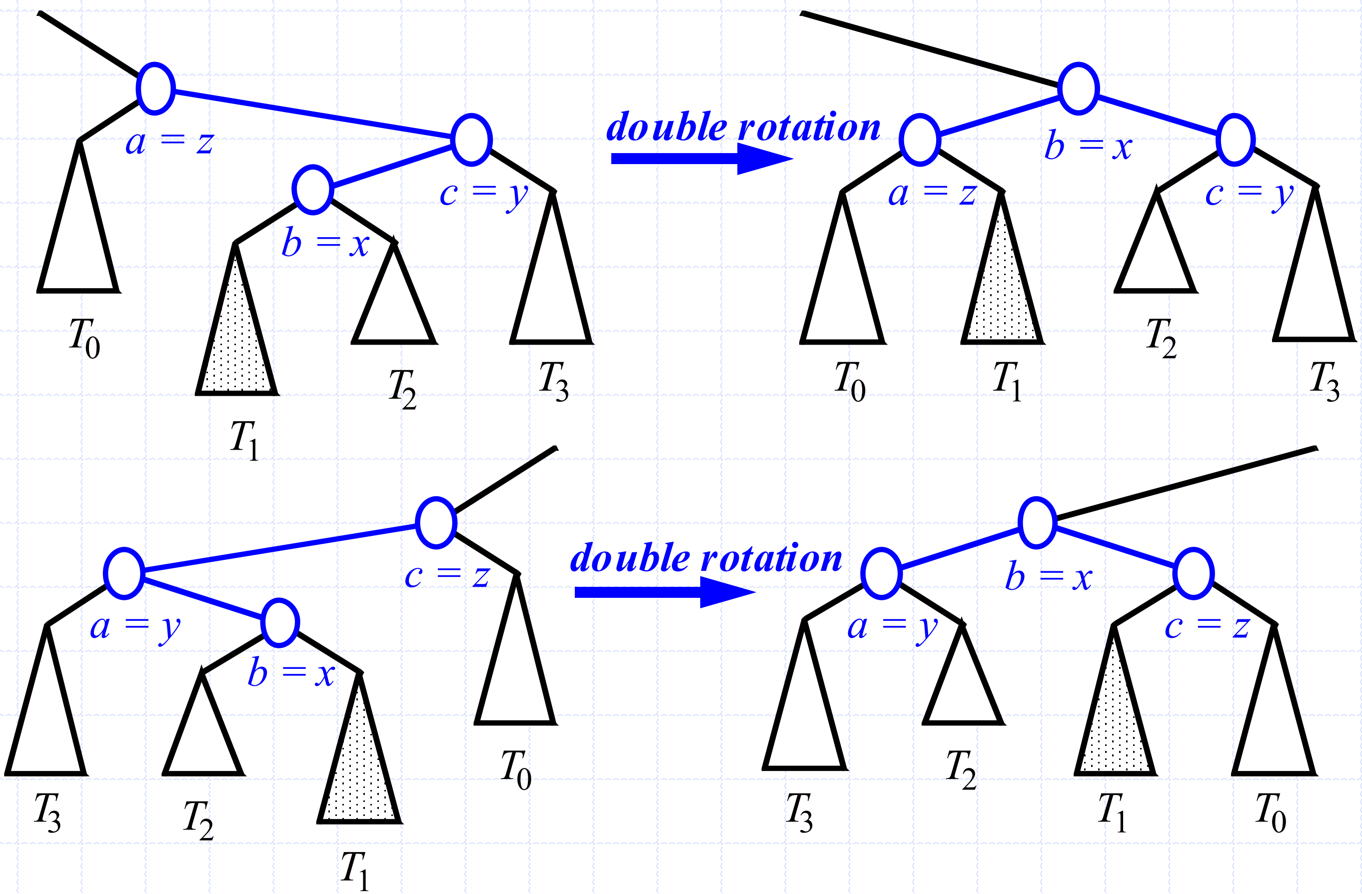
Book's Slide: Restructuring for Cases 1&4 – Single Rotations

- Single Rotations:

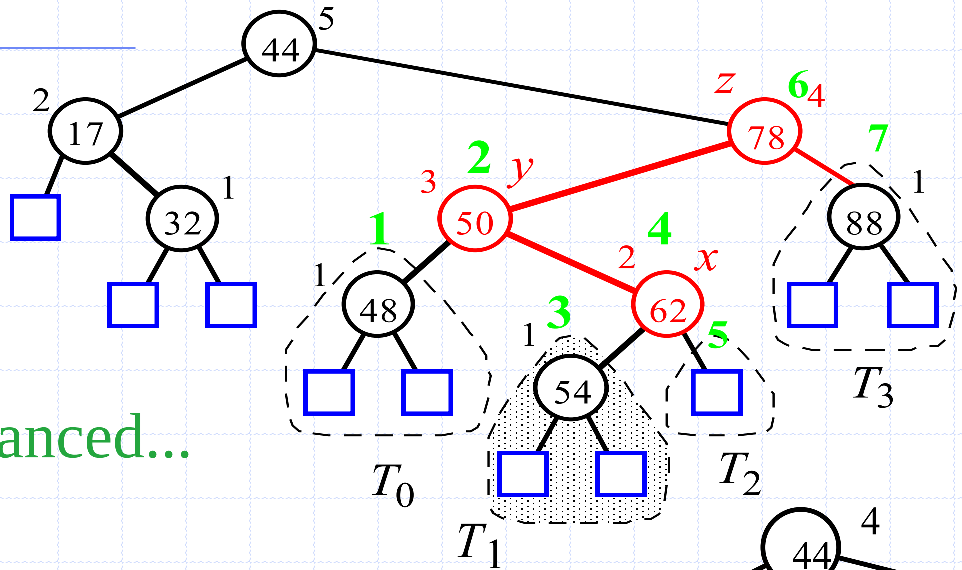


Book's Slide: Restructuring for Cases 2&3 – Double Rotations

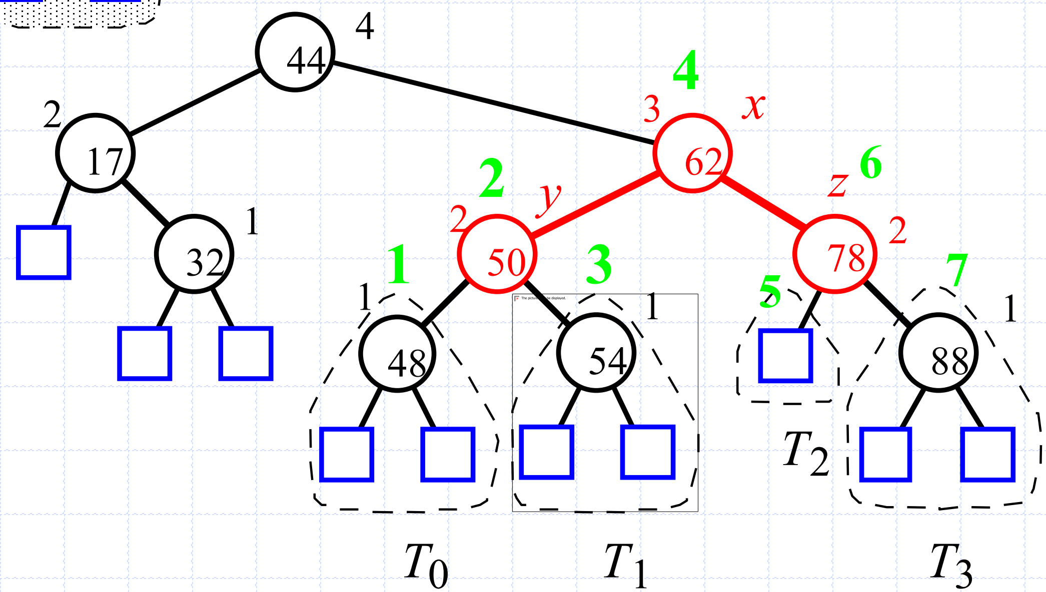
- double rotations:



Insertion Example



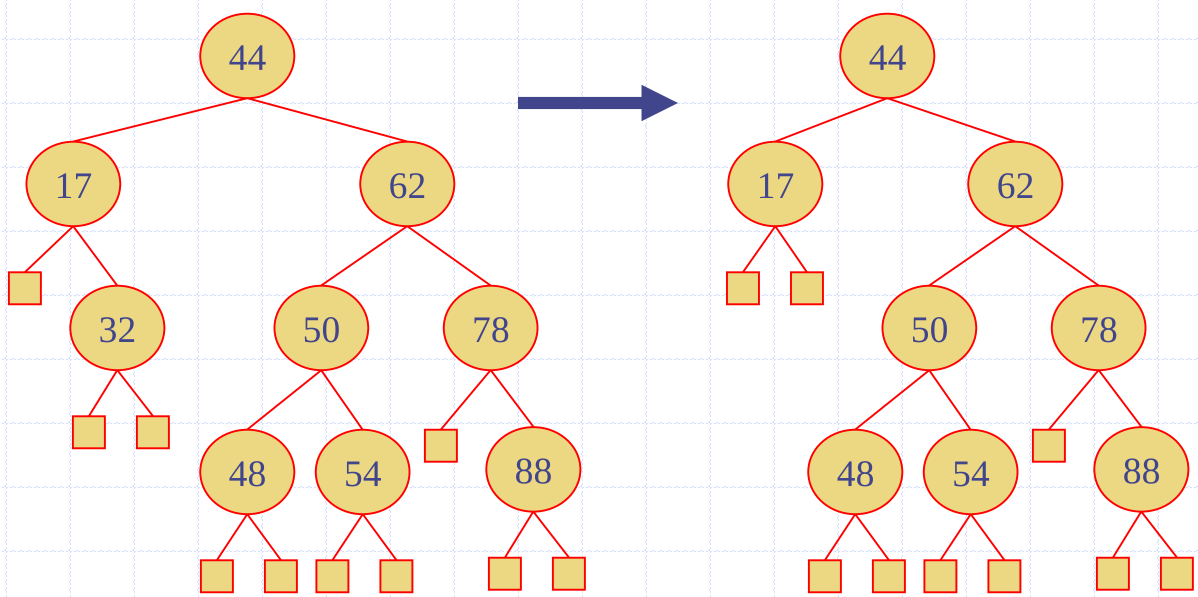
unbalanced...



...balanced

Removal

- Removal begins as in a binary search tree, which means the node removed will become an empty external node. Its parent, w, may cause an imbalance.
- Example:

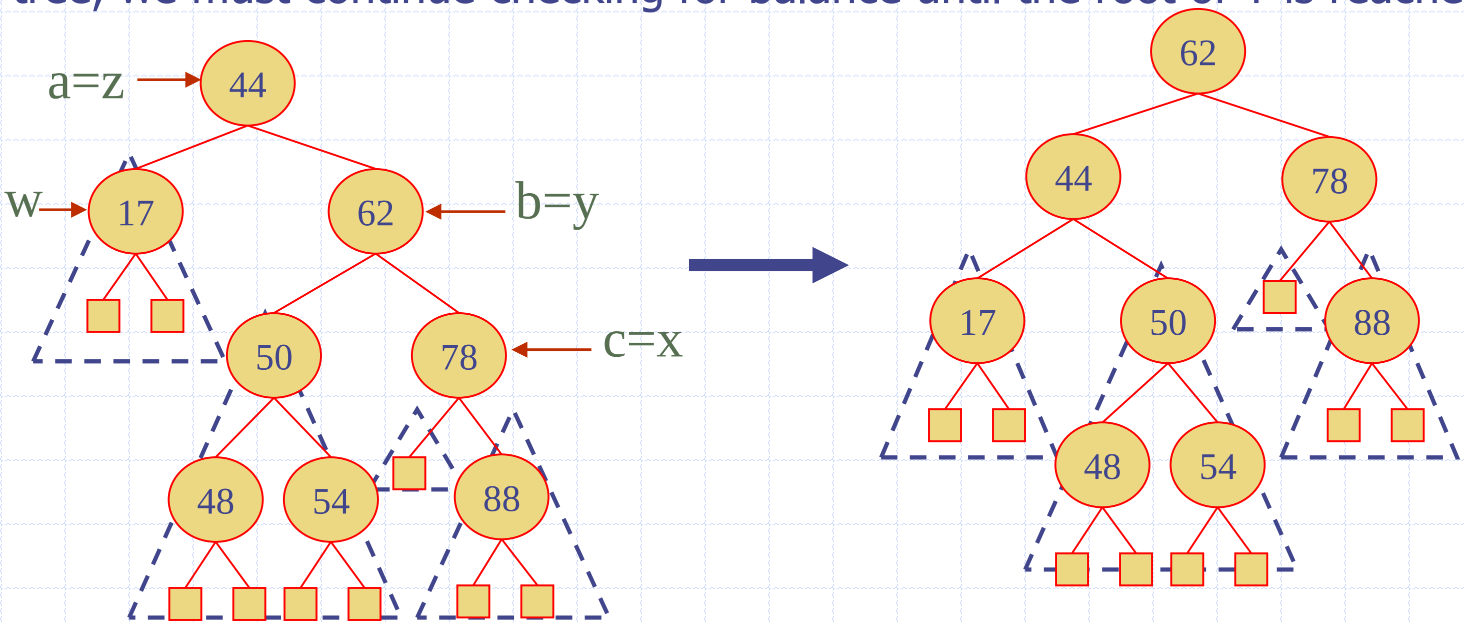


before deletion of 32

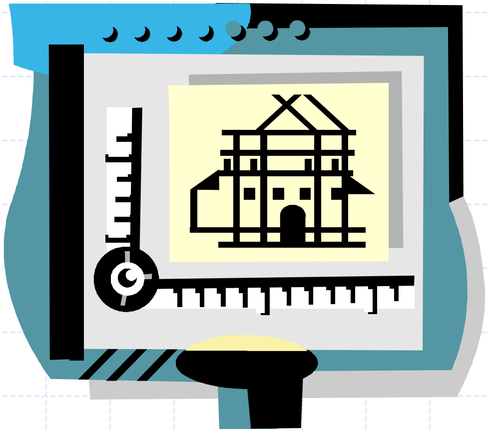
after deletion

Rebalancing after a Removal

- Let z be the first unbalanced node encountered while travelling up the tree from w. Also, let y be the child of z with the larger height, and let x be the child of y with the larger height
- We perform a trinode restructuring to restore balance at z
- As this restructuring may upset the balance of another node higher in the tree, we must continue checking for balance until the root of T is reached



AVL Tree Performance



- AVL tree storing n items
  - The data structure uses O(n) space
  - A single restructuring takes O(1) time
    - using a linked-structure binary tree
  - Searching takes O(log n) time
    - height of tree is O(log n), no restructures needed
  - Insertion takes O(log n) time
    - initial find is O(log n)
  - Removal takes O(log n) time
    - initial find is O(log n)
    - restructuring up the tree, maintaining heights is O(log n)

Home Exercise

- Play around with AVL Trees to see their behavior from this website:

http://www.cs.usfca.edu/~galles/visualization/AVLtree.html

- Go over these website for more examples:

http://www.mathcs.emory.edu/~cheung/Courses/323/Syllabus/Trees/AVL-insert.html

http://www.mathcs.emory.edu/~cheung/Courses/323/Syllabus/Trees/AVL-delete.html (this has a multiple re-structure example)

- Go over the implementation in the book (chapter 11). You are going to need to go back to previous chapters
- Alternatively, find other sources if positions are confusing or you want to see everything in one place, e.g.,

http://users.cs.fiu.edu/~weiss/dsaajava/code/DataStructures/AvlTree.java