UNIVERSITY of NORTH TEXAS

CSCE 3110
Data Structures & Algorithm Analysis

AVL Trees

Reading: *Chap. 4, Weiss*

Sorting with BST

- Use binary search trees for sorting
- Start with unsorted sequence
- Insert all elements in a BST
- Traverse the tree…. how ?
- Running time?

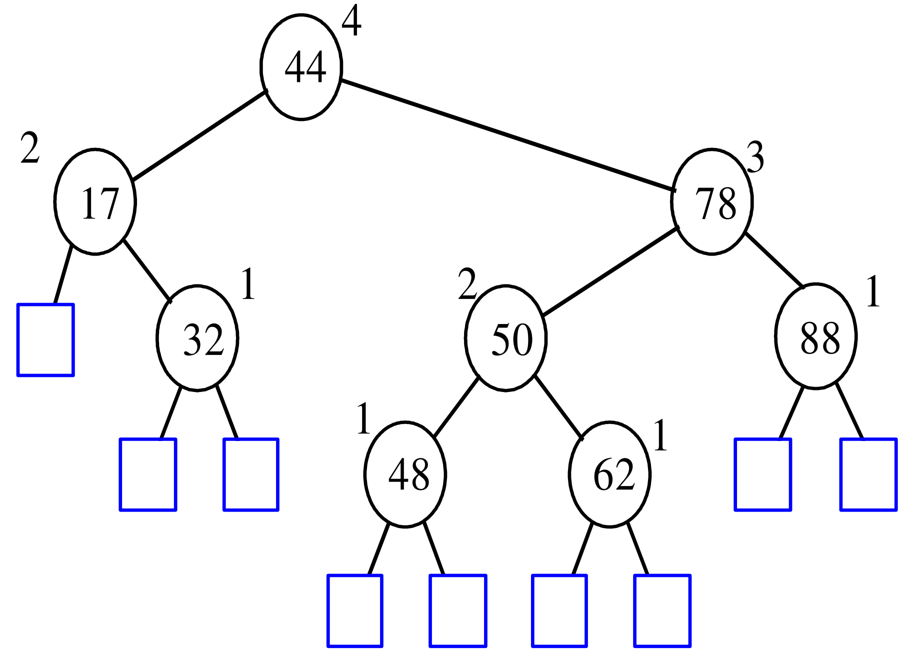
Better Binary Search Trees

Prevent the degeneration of the BST :

- A BST can be set up to maintain balance during updating operations (insertions and removals)
- Types of BST which maintain the optimal performance:
  - splay trees
  - AVL trees
  - Red-Black trees
  - B-trees

AVL Trees

- Balanced binary search trees
- An AVL Tree is a ***binary search tree*** such that for every internal node v of T, the **heights of the children of v can differ by at most 1.**



Height of an AVL Tree

- **Proposition**: The ***height*** of an AVL tree T storing n keys is O(log n).
- **Justification**: The easiest way to approach this problem is to find **n(h):** the *minimum number of nodes* of an AVL tree of height h.

- n(1) = 2 and n(2) = 4
- for n ≥ 3, an AVL tree of height h contains the root node, one AVL subtree of height n-1 and the other AVL subtree of height n-2.
- → n(h) = 1 + n(h-1) + n(h-2)
- given n(h-1) > n(h-2) → n(h) > 2n(h-2)

  n(h) > 2n(h-2)
  n(h) > 4n(h-4)
  …
  $n(h) > 2^i n(h-2i)$
- pick i = h/2 – 1 → $n(h) \geq 2^{h/2-1}$
- follow h < 2log n(h) +2
- → height of an AVL tree is O(log n)

Insertion

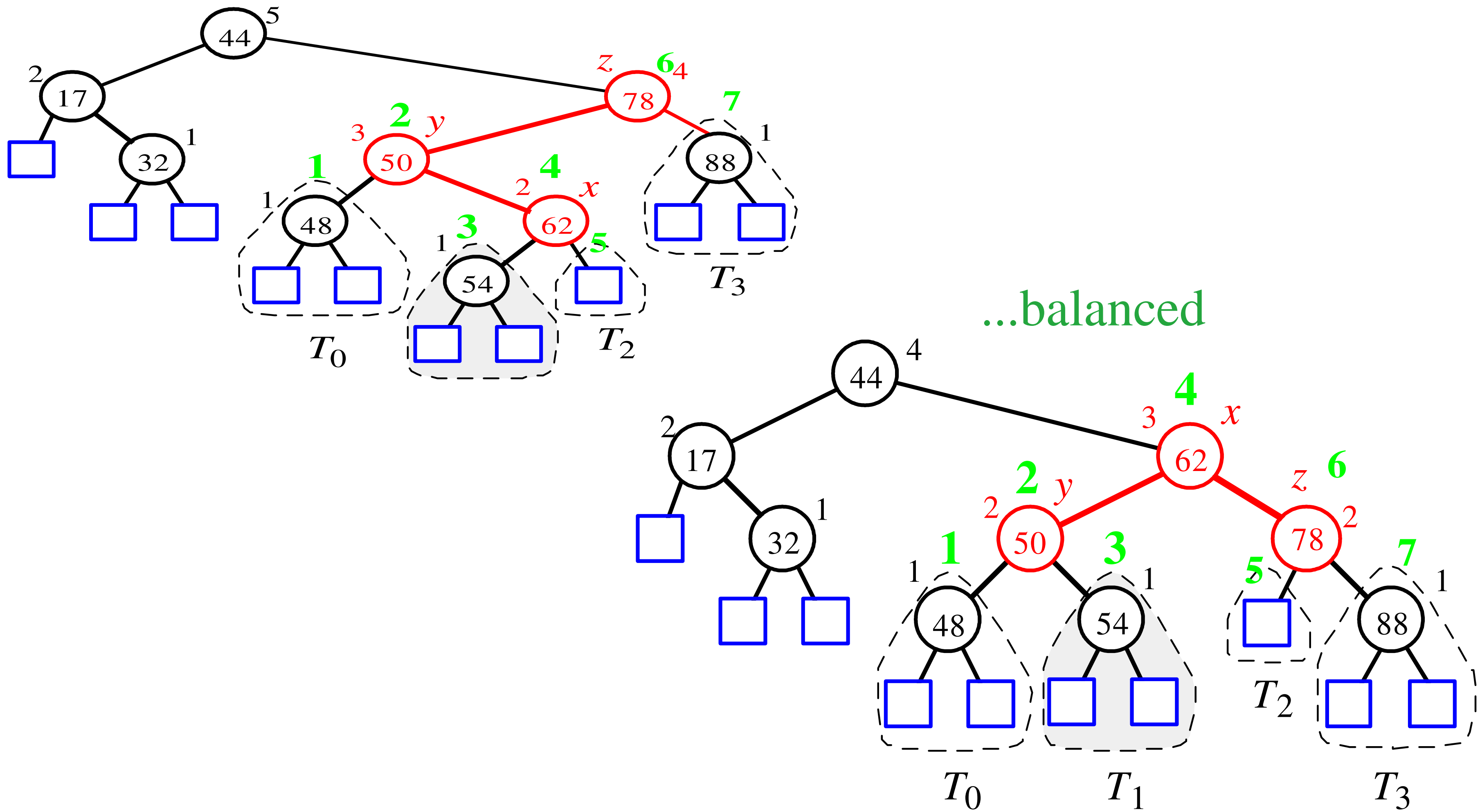
- A binary search tree T is called ***balanced*** if for every node v, the height of v's children differ by at most one.
- Inserting a node into an AVL tree involves performing an expandExternal(w) on T, which changes the heights of some of the nodes in T.
- If an insertion causes T to become unbalanced, we travel up the tree from the newly created node until we find the first node x such that its grandparent z is unbalanced node.
- Since z became unbalanced by an insertion in the subtree rooted at its child y, height(y) = height(sibling(y)) + 2
- Need to rebalance...

Insertion: Rebalancing

- To rebalance the subtree rooted at z, we must perform a **restructuring**
- we rename x, y, and z to a, b, and c based on the order of the nodes in an in-order traversal.
- z is replaced by b, whose children are now a and c whose children, in turn, consist of the four other subtrees formerly children of x, y, and z.

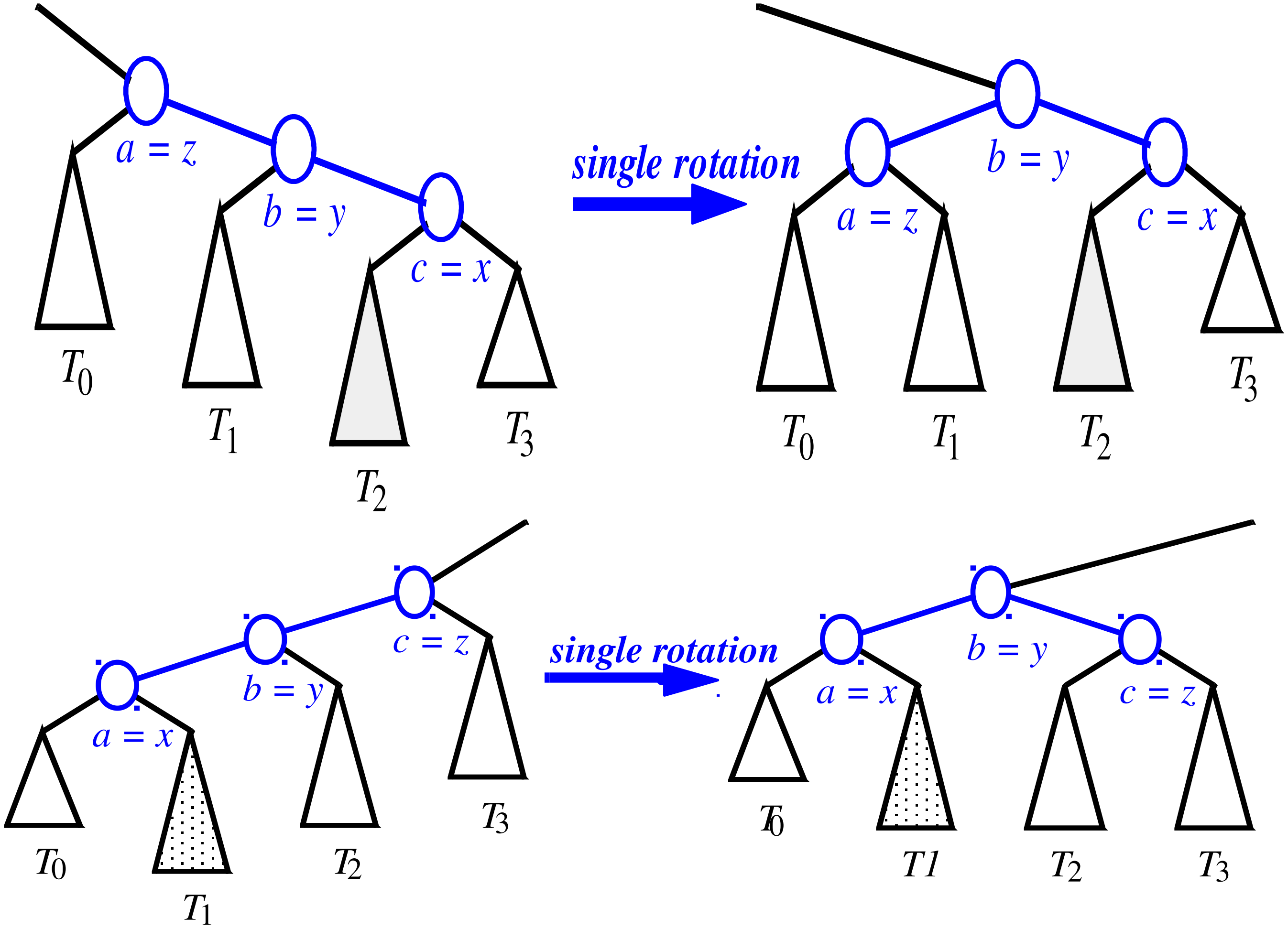
Insertion (cont'd)

unbalanced...



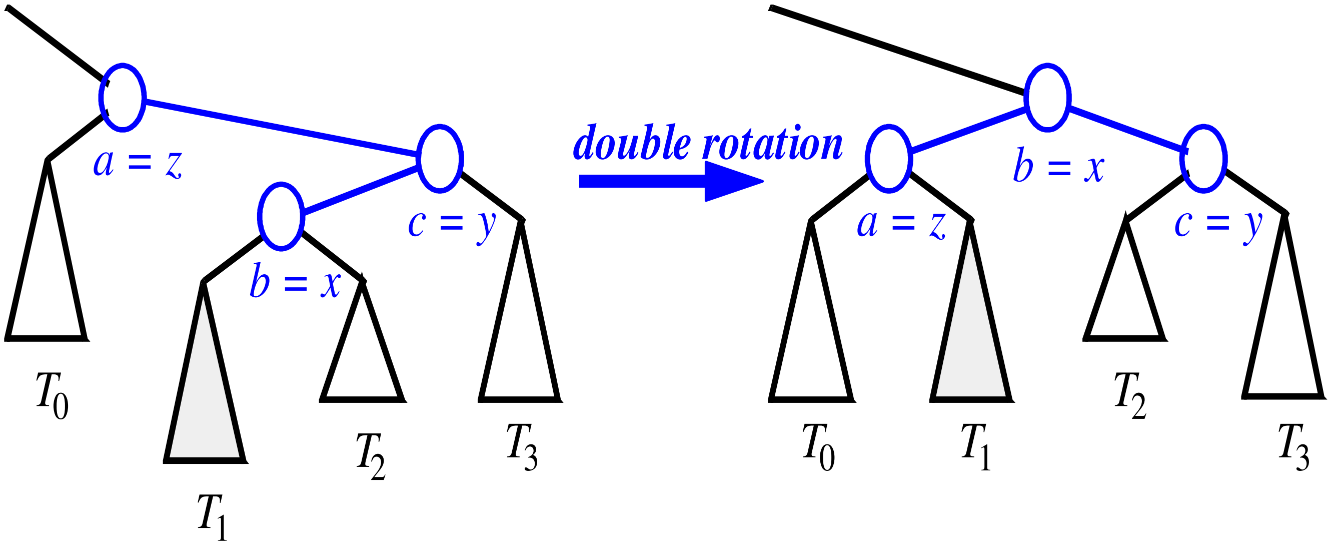
Restructuring

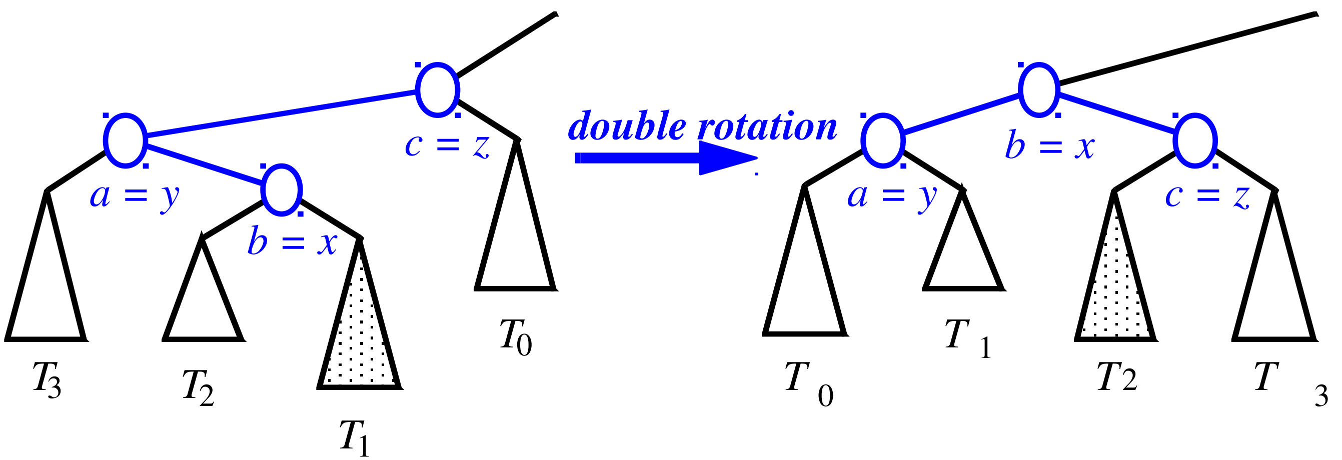
- The four ways to rotate nodes in an AVL tree, graphically represented
  -Single Rotations:



Restructuring (cont'd)

- double rotations:





Restructure Algorithm

Algorithm restructure(x):

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

Output: Tree T restructured by a rotation (either single or double) involving nodes x, y, and z.

1: Let (*a, b, c*) be an inorder listing of the nodes *x, y, and z,* and let (T0, T1, T2, T3) be an inorder listing of the the four subtrees of *x, y, and z,* rooted at *x, y, or z.*

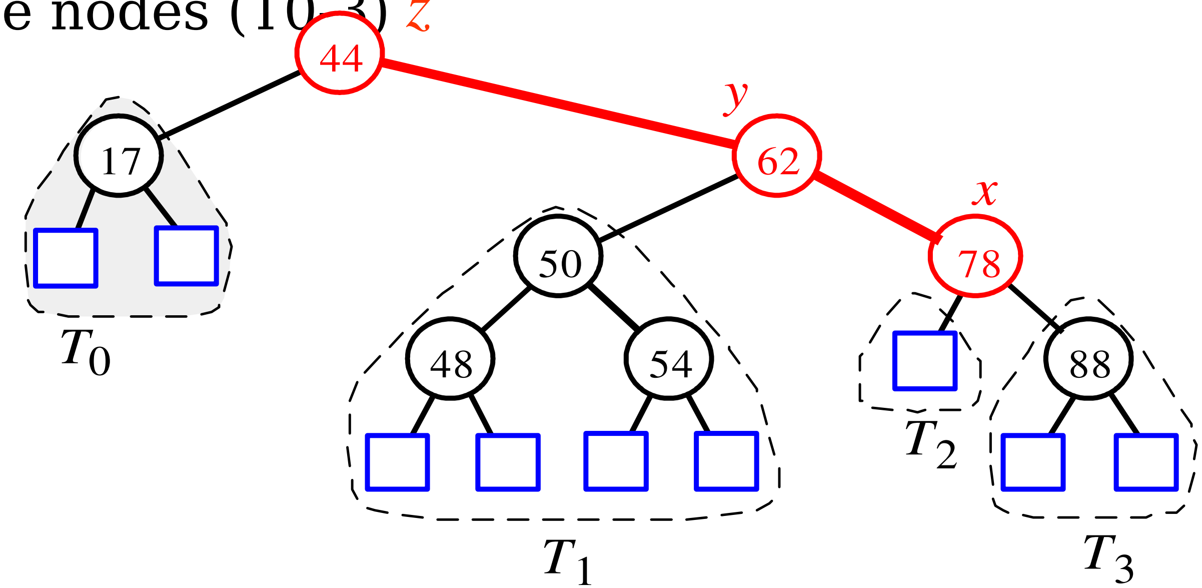
2. Replace the subtree rooted at z with a new subtree rooted at *b*

3. Let *a* be the left child of *b* and let T0, T1 be the left and right subtrees of *a*, respectively.

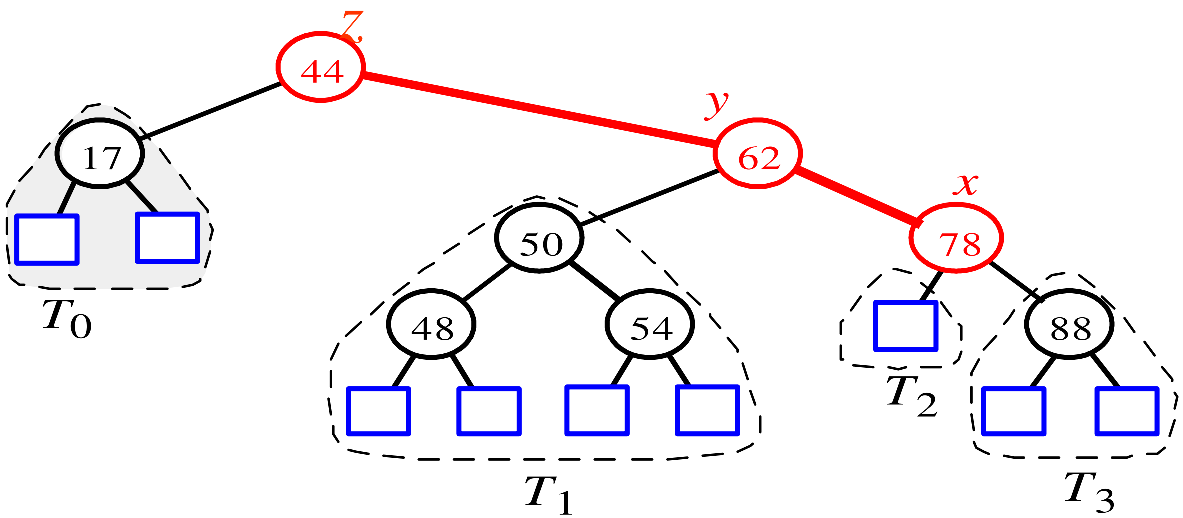
4. Let *c* be the right child of *b* and let T2, T3 be the left and right subtrees of *c*, respectively.

Cut/Link Restructure Algorithm

- Let's go into a little more detail on this algorithm…
- Any tree that needs to be balanced can be grouped into 7 parts: x, y, z, and the 4 trees anchored at the children of those nodes (T0-3)



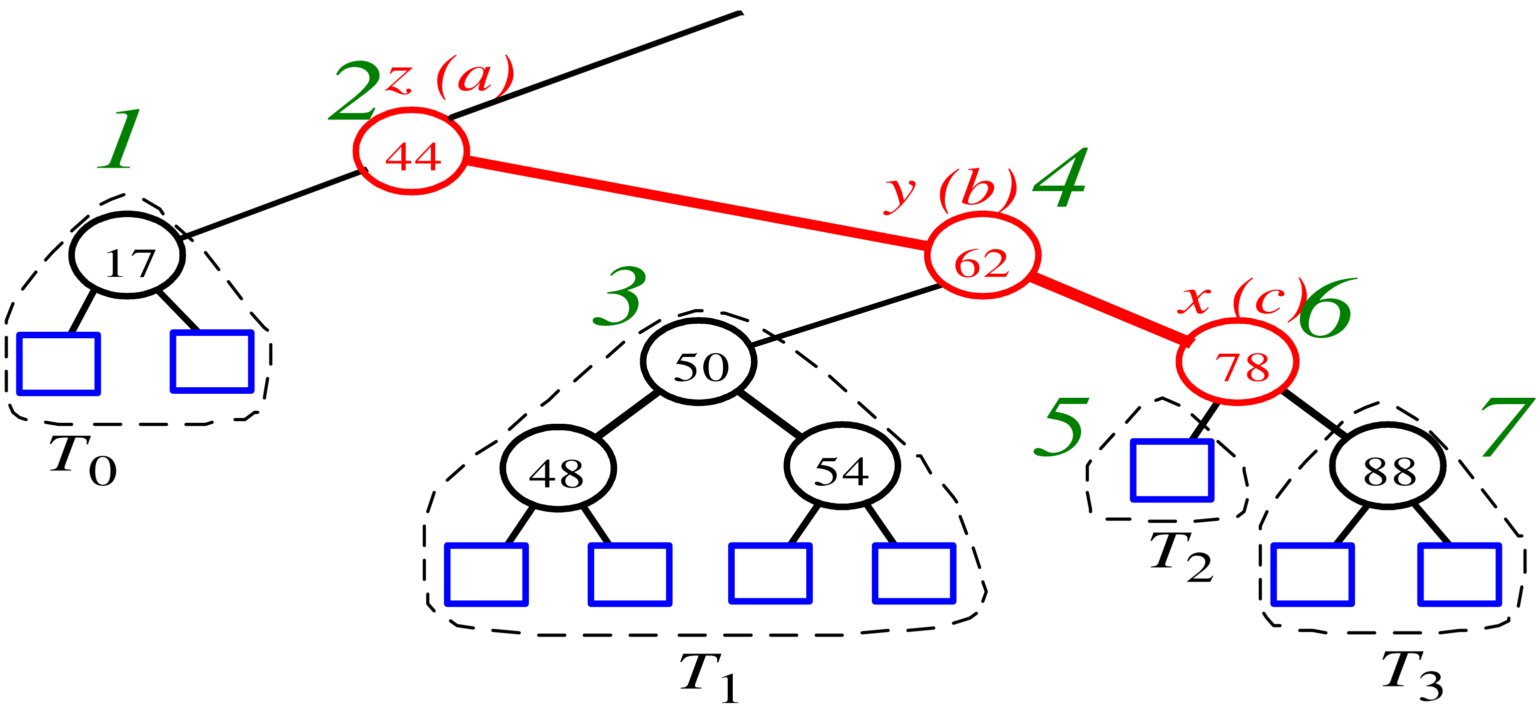
Cut/Link Restructure Algorithm



- Make a new tree which is balanced and put the 7 parts from the old tree into the new tree so that the numbering is still correct when we do an in-order-traversal of the new tree.
- This works regardless of how the tree is originally unbalanced.
- Let's see how it works!

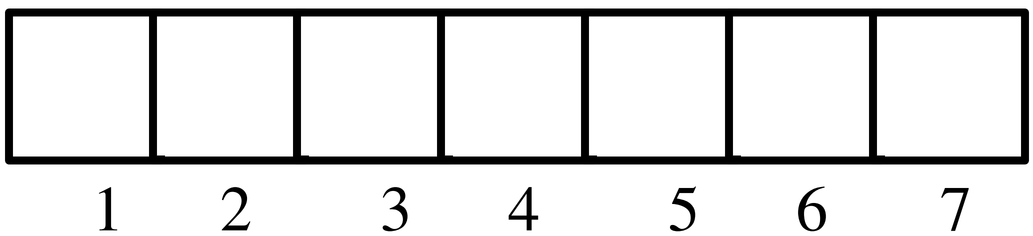
Cut/Link Restructure Algorithm

- Number the 7 parts by doing an in-order-traversal. (note that x,y, and z are now renamed based upon their order within the traversal)

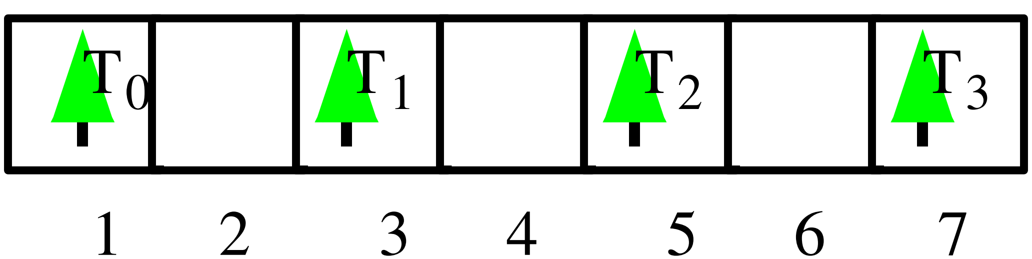


Cut/Link Restructure Algorithm

- Now create an Array, numbered 1 to 7 (the 0th element can be ignored with minimal waste of space)

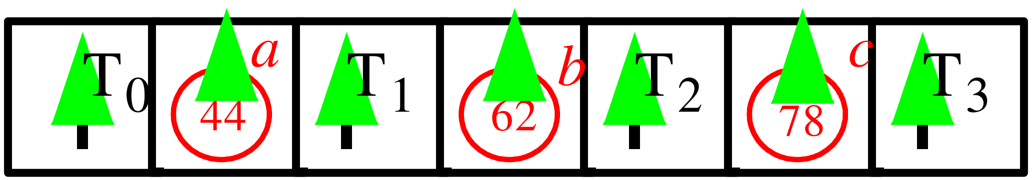


- Cut() the 4 T trees and place them in their inorder rank in the array



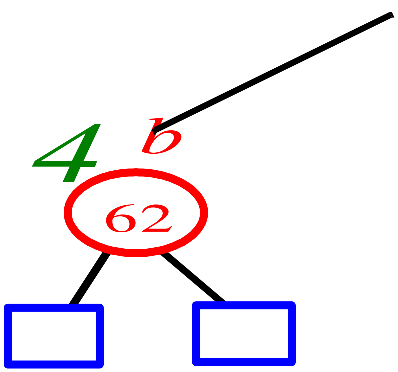
Cut/Link Restructure Algorithm

- Now cut x,y, and z in that order (child,parent,grandparent) and place them in their inorder rank in the array.



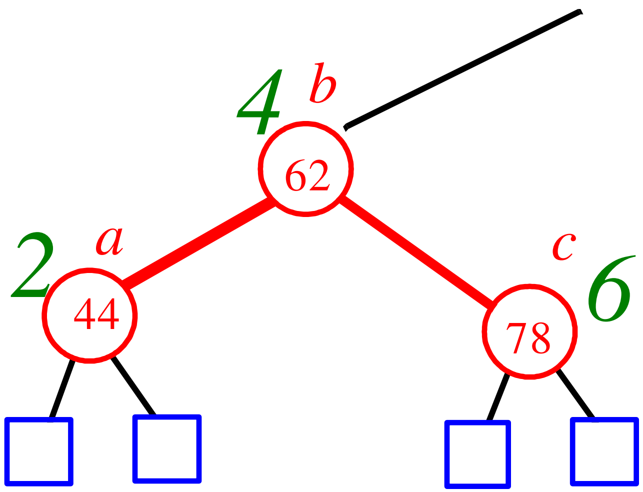
1 2 3 4 5 6 7

- Now we can re-link these subtrees to the main tree.
- Link in rank 4 (b) where the subtree's root formerly



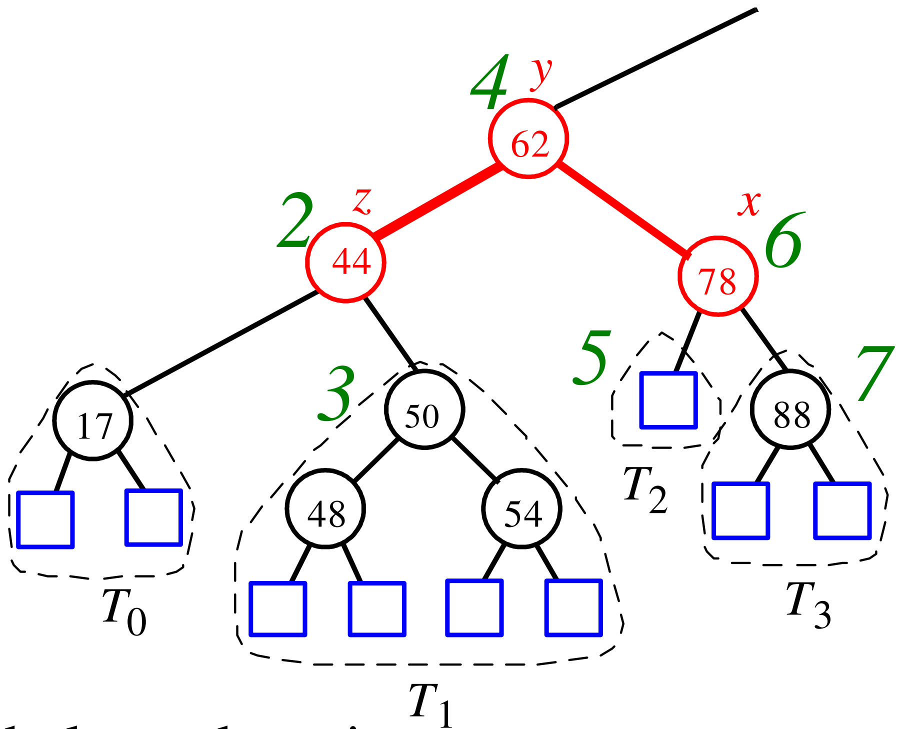
Cut/Link Restructure Algorithm

- Link in ranks 2 (a) and 6 (c) as 4’s children.



Cut/Link Restructure Algorithm

- Finally, link in ranks 1,3,5, and 7 as the children of 2 and 6.



- Now you have a balanced tree!

Cut/Link Restructure Algorithm

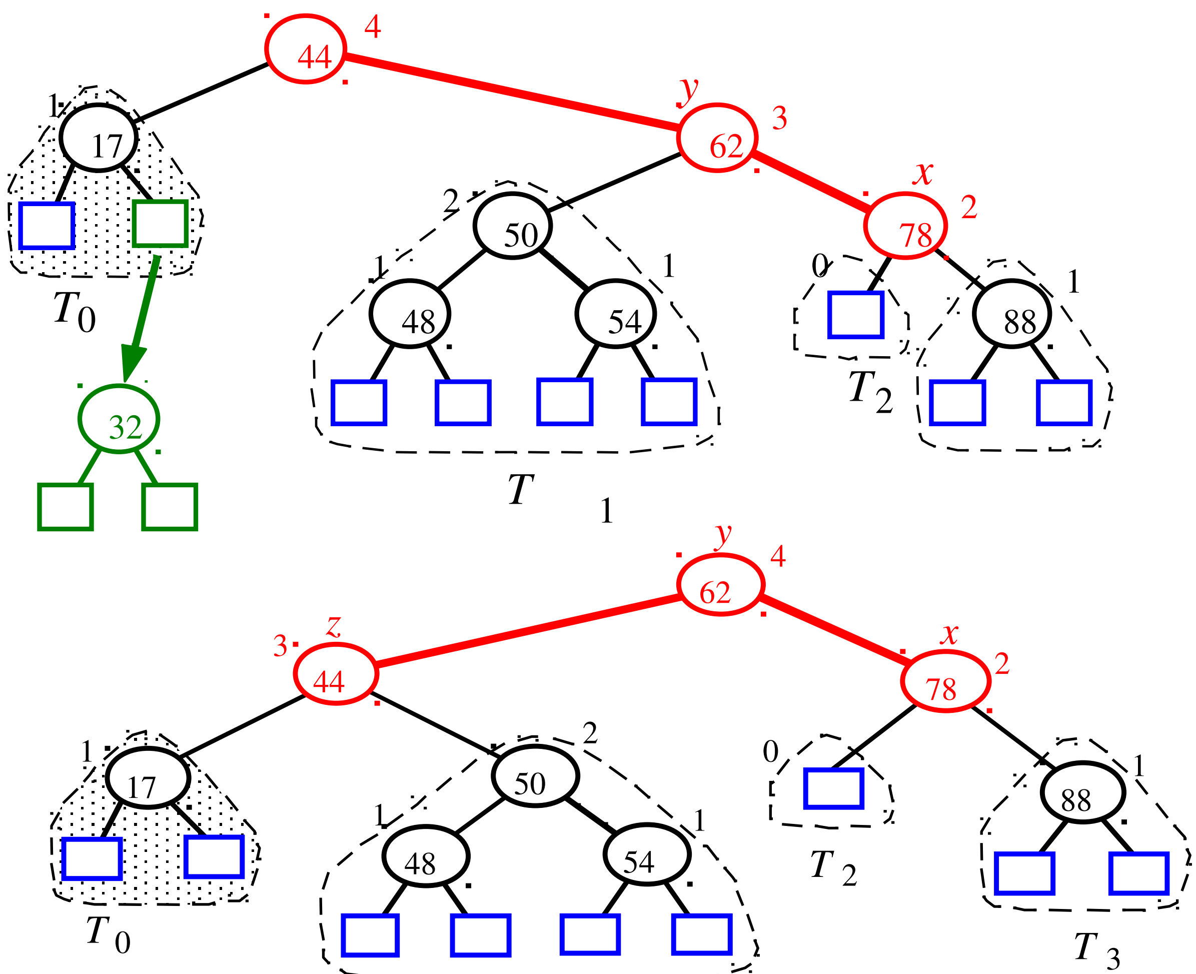
- This algorithm for restructuring has the exact same effect as using the four rotation cases discussed earlier.
- Advantages: no case analysis, more elegant
- Disadvantage: can be more code to write
- Same time complexity

Removal

- We can easily see that performing a removeAboveExternal(w) can cause T to become unbalanced.
- Let *z* be the first unbalanced node encountered while traveling 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 can perform operation restructure(x) to restore balance at the subtree rooted 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

Removal (cont'd)

- example of deletion from an AVL tree:



Removal (cont'd)

- example of deletion from an AVL tree: