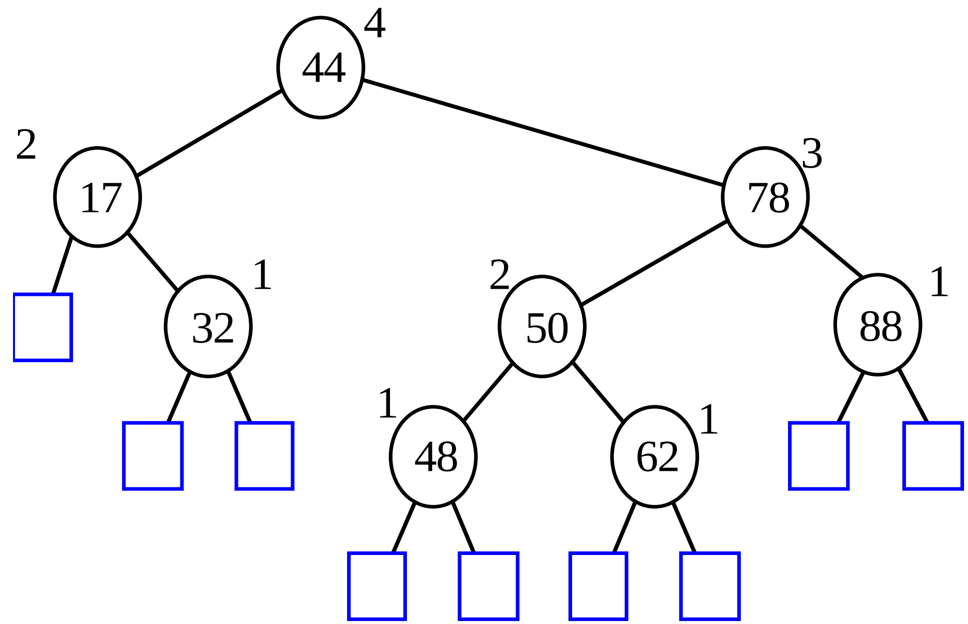
AVL Trees

cse2011

section 10.2 of textbook

AVL Trees

- AVL trees are **balanced**.
- 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***.



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

Height of an AVL Tree

- **Proposition**: The ***height*** of an AVL tree T storing n keys is O(log n).

Proof:

- Find **n(h):** the *minimum number of internal nodes* of an AVL tree of height h
- We see that n(1) = 1 and n(2) = 2
- For h ≥ 3, an AVL tree of height h contains the root node, one AVL subtree of height h–1 and the other AVL subtree of height h–2.
- i.e. $n(h) = 1 + n(h-1) + n(h-2)$

Height of an AVL Tree (2)

- Knowing n(h-1) > n(h-2), we get n(h) > 2n(h-2)

  n(h) > 2n(h-2)

  n(h) > 4n(h-4)

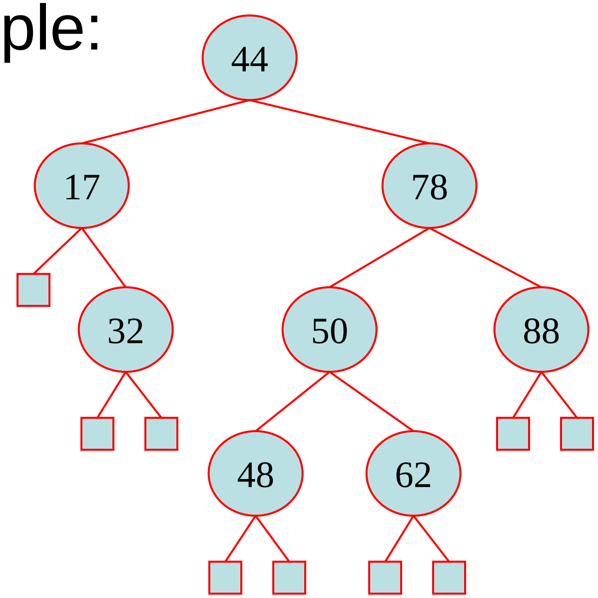
  …

  n(h) > 2^in(h-2i)

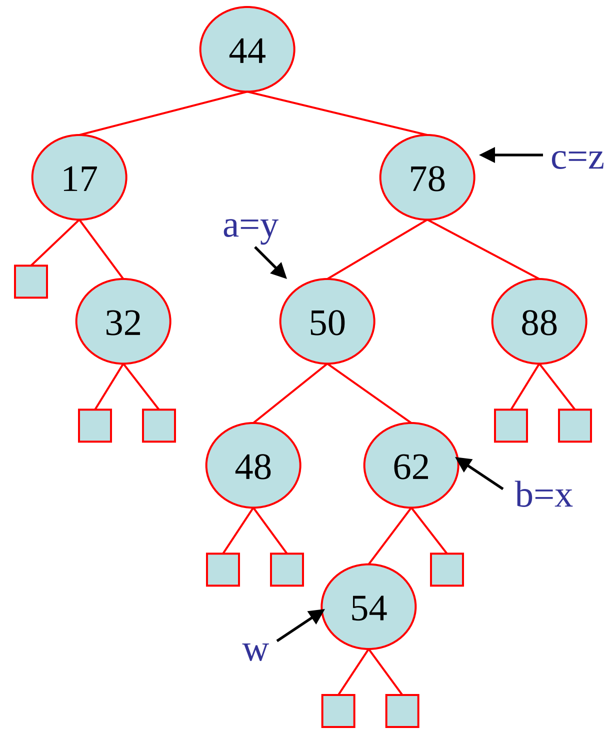
- Solving the base case we get: n(h) ≥ $2^{h/2-1}$
- Taking logarithms: h < 2log n(h) +2
- Thus the height of an AVL tree is O(log n)

Insertion in an AVL Tree

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



before insertion



after insertion

Insertion: rebalancing

- A binary search tree T is called ***balanced*** if for every node *v*, the height of *v*'s children differ by at most 1.

- Inserting a node into an AVL tree involves performing *insertAtExternal*(*w, e*) on T, which changes the heights of some of the nodes in T.

Insertion: rebalancing

- If an insertion causes T to become **unbalanced,** we travel up the tree from the newly created node w until we find the first node **z** that is unbalanced.
- y = child of **z** with higher height (Note: y = ancestor of w)
- x = child of y with higher height

  (Note: x = ancestor of w or x = w)
- Since **z** became unbalanced by an insertion in the subtree rooted at its child y, height(y) = height(sibling(y)) + 2

Insertion: restructuring

- Now to rebalance...

- To rebalance the subtree rooted at ***z***, we must perform a *restructuring*.

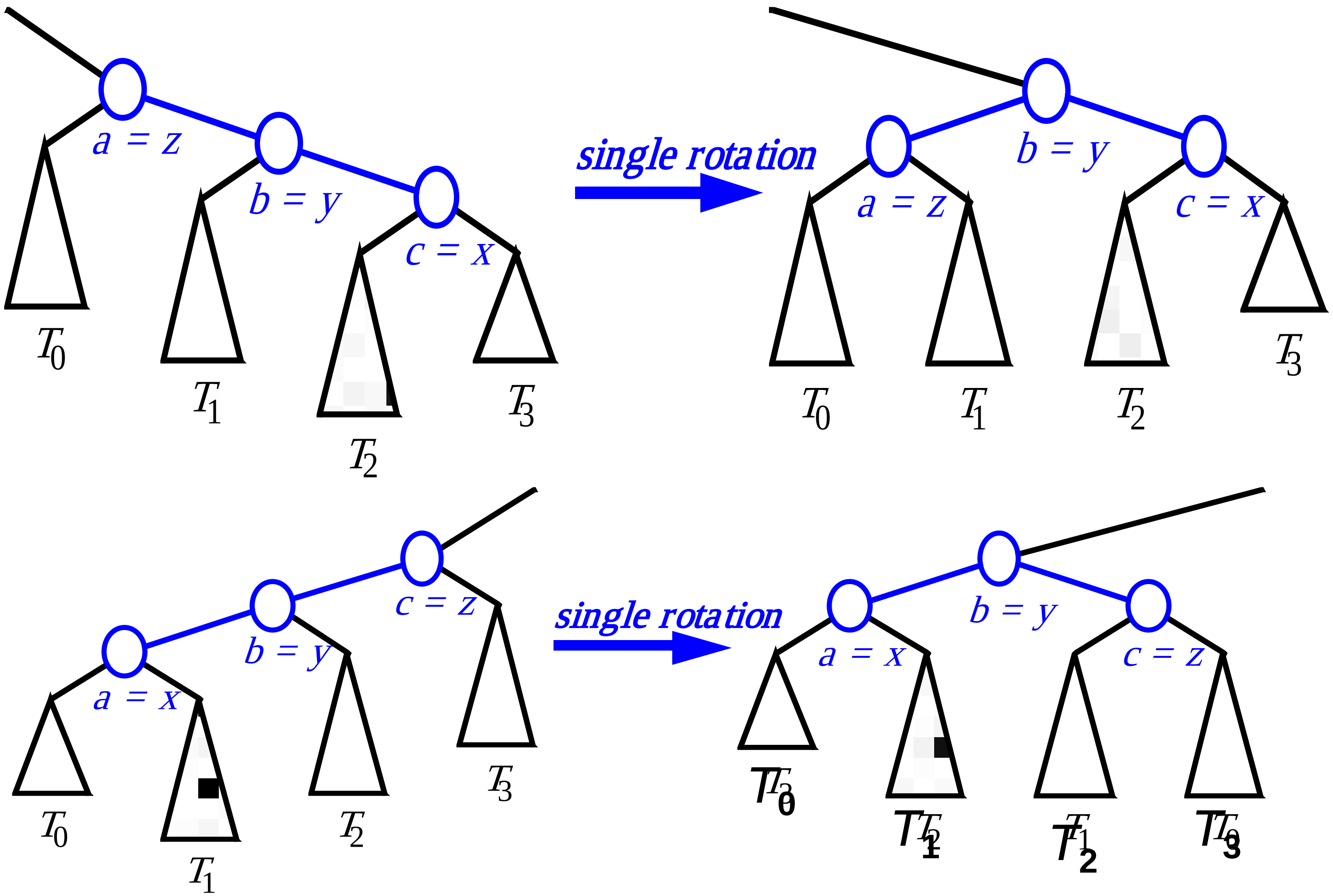
Tri-node Restructuring

- We rename x, y, and z to a, b, and c based on the order of the nodes in an **in-order traversal** (see the next slides for 4 possible mappings).

- z is replaced by b, whose children are now a and c whose children, in turn, consist of the 4 other subtrees formerly children of x, y, and z.

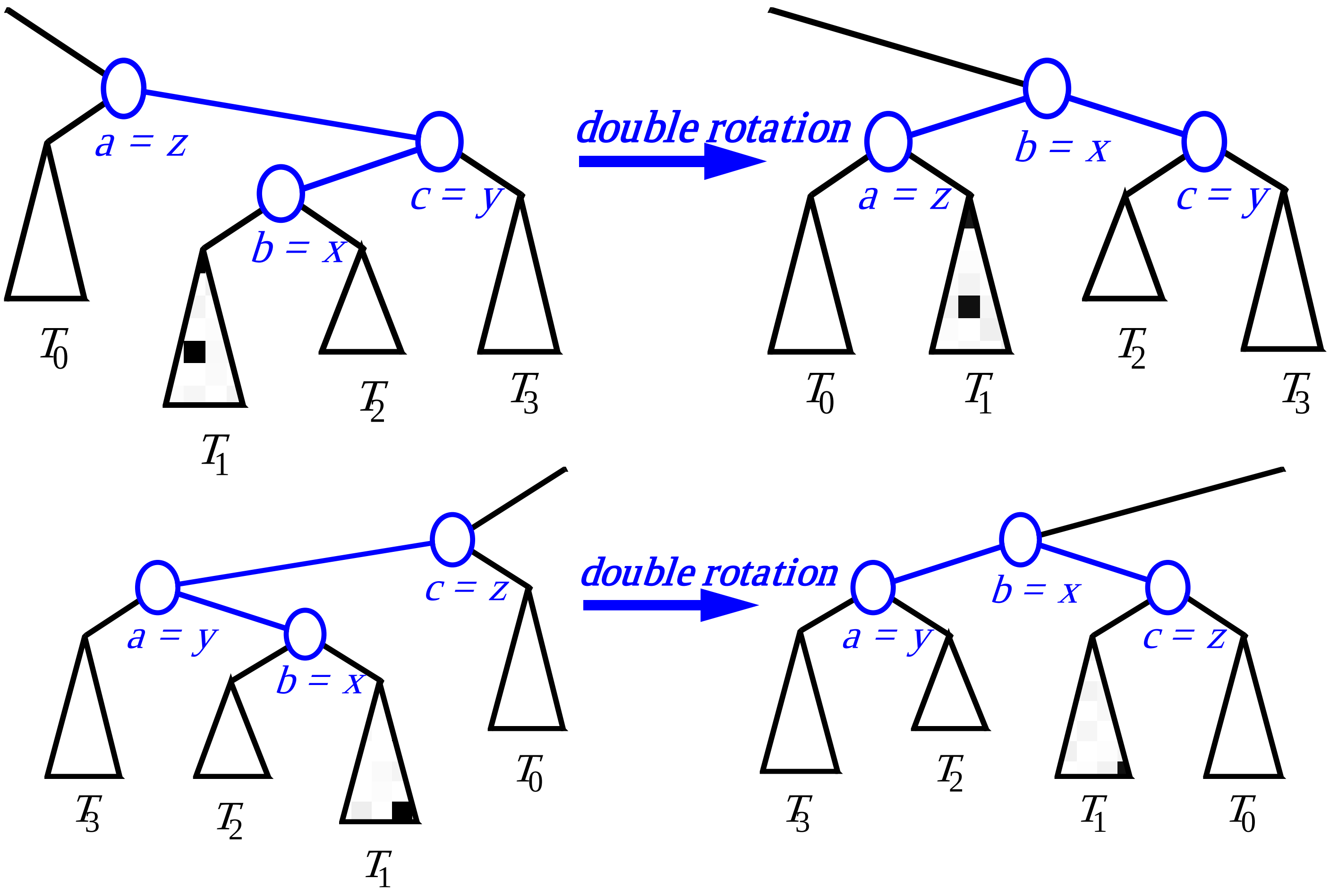
Tri-node Restructuring (2)

Single rotations



Tri-node Restructuring (3)

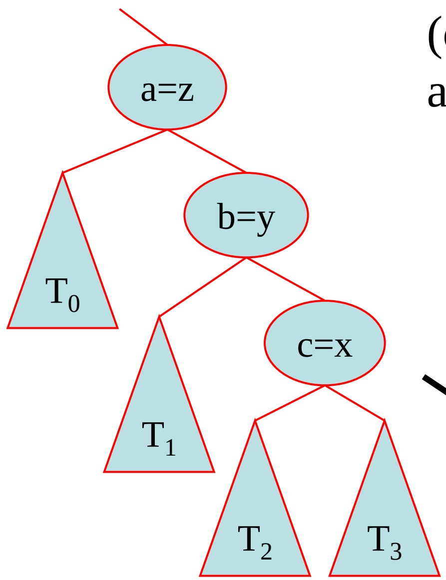
Double rotations

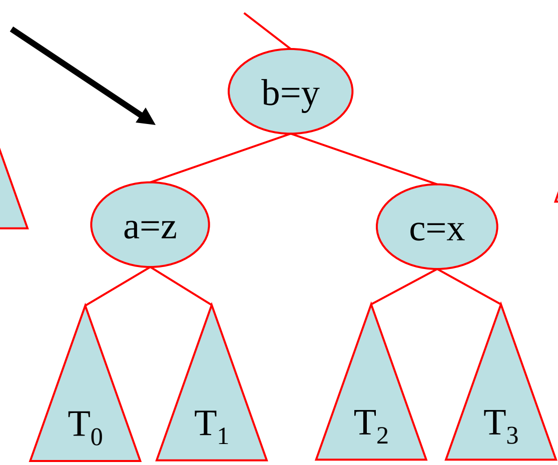


Single/Double Rotations

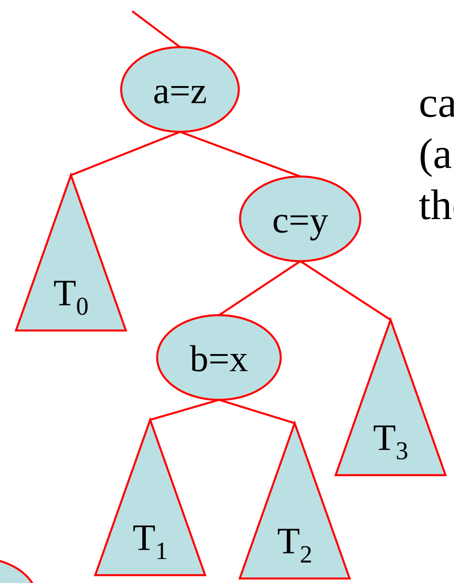
- let (*a*,*b*,*c*) be an inorder listing of *x*, *y*, *z*
- perform the rotations needed to make *b* the topmost node of the three

(other two cases are symmetrical)

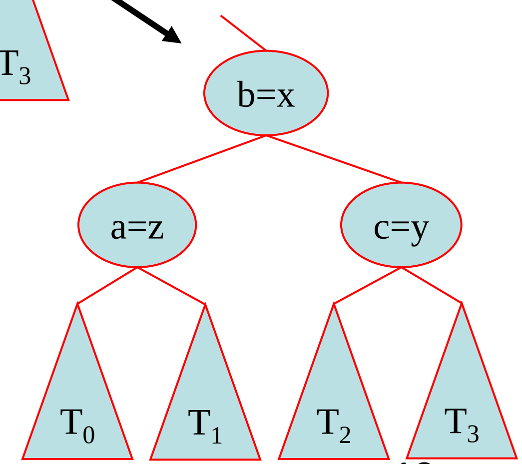




case 1: single rotation
(a left rotation about *a*)

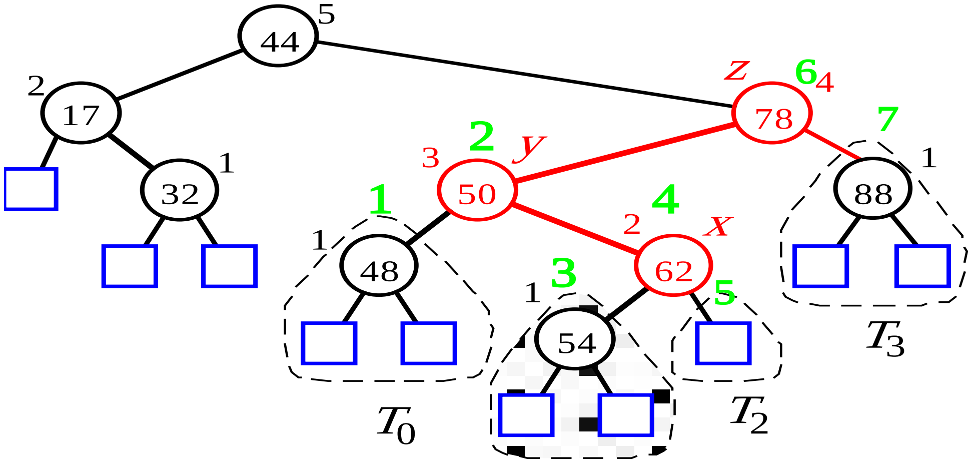


case 2: double rotation
(a right rotation about *c*, then a left rotation about *a*)

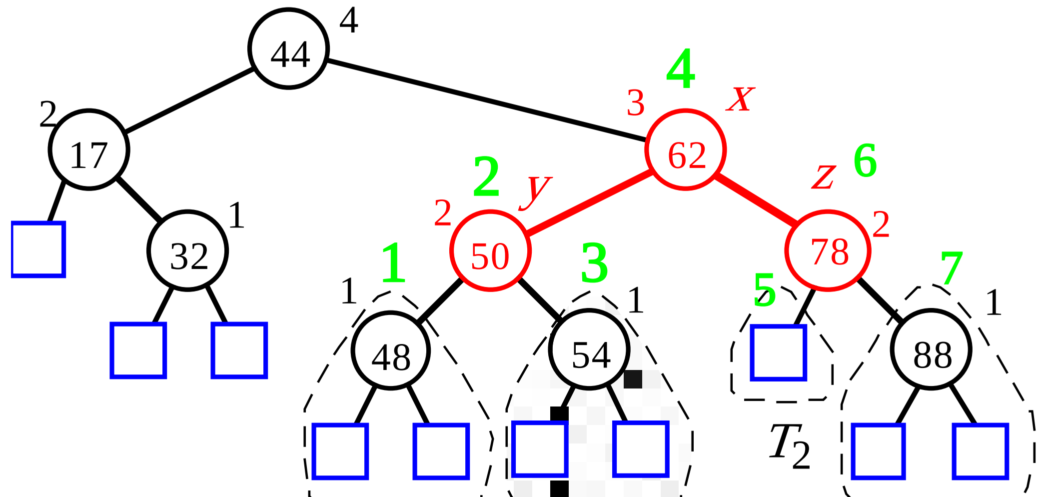


Restructuring Example

unbalanced...



...balanced



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.

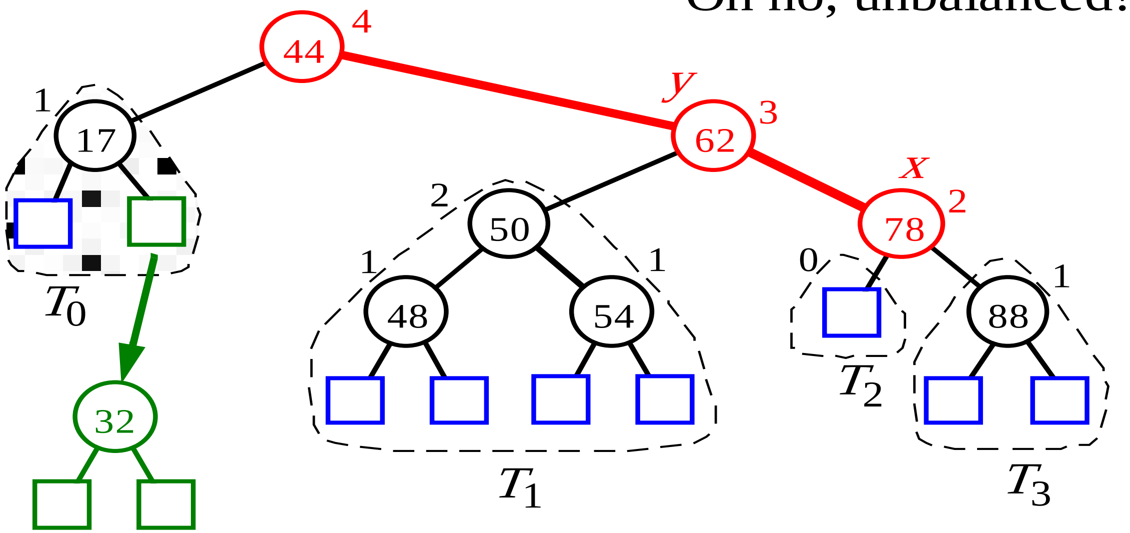
1. Let (*a, b, c*) be an inorder listing of the nodes *x, y, and z,* and let (T0, T1, T2, T3) be an inorder listing of the the four subtrees of *x, y, and z,* not rooted at *x, y, or z.*
2. Replace the subtree rooted at z with a new subtree rooted at *b*
3. Let *a* be the left child of *b* and let T0, 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.

Removal

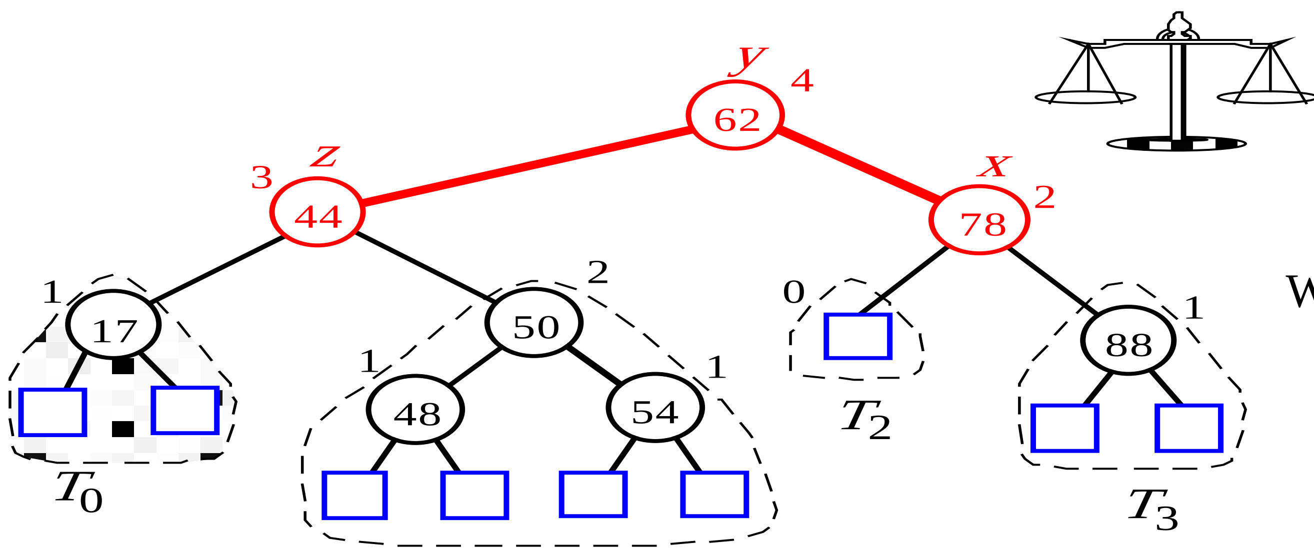
- First remove the node as in a BST.
- Performing a removeExternal(w) can cause T to become unbalanced.
- Let *z* be the first unbalanced node encountered while travelling up the tree from w.
- y = child of z with higher height (y $\neq$ ancestor of w)
- x = child of y with higher height, or either child if two children of y have the same height.
- 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 Example

Oh no, unbalanced!

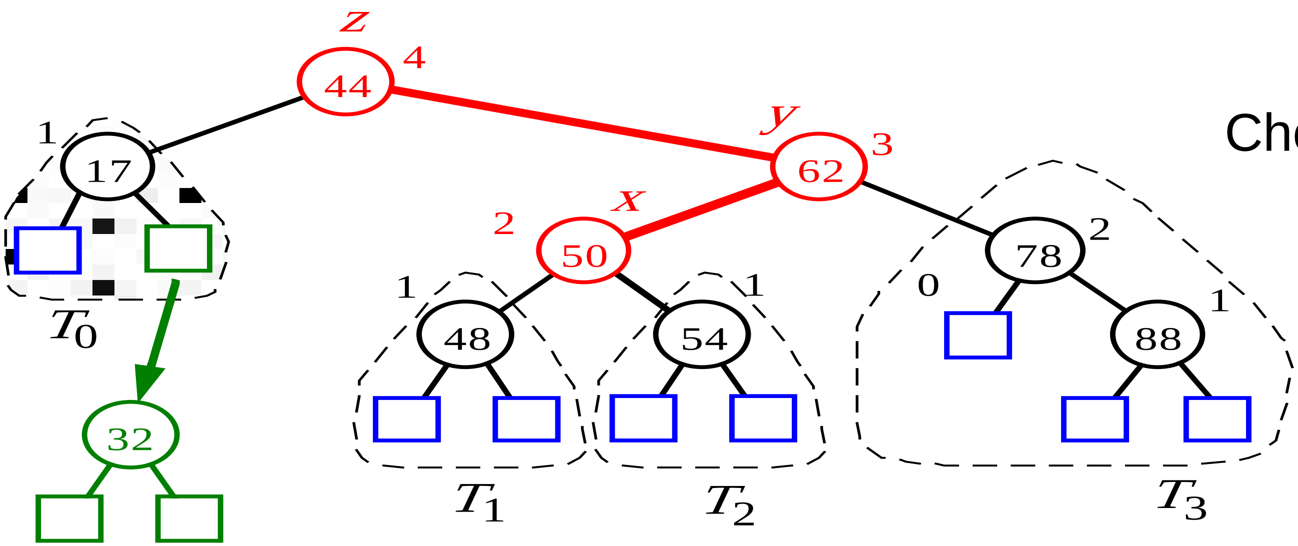


Choose either 78 or 50 as node *x*.



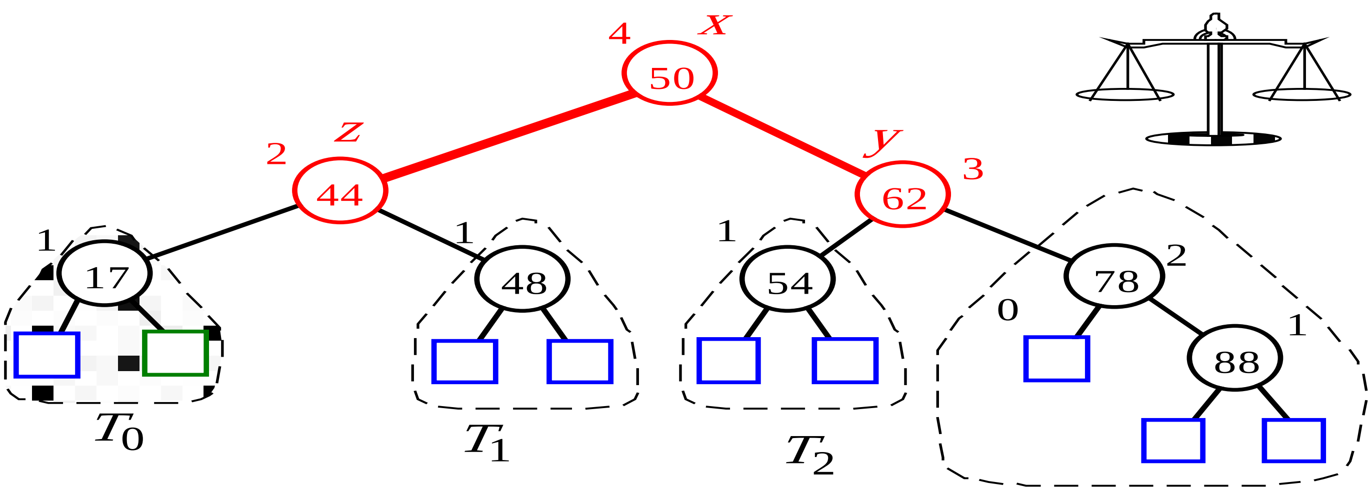
Whew, balanced!

Removal Example (2)



Choose 50 as *x*.

Oh no, unbalanced!



Whew, balanced!

Next lecture …

- Heaps (8.3)
- Midterm

- **Midterm** will be held on **Wednesday June 26th** in **CLH F**. On June 26th, we will have a lecture from 7:00pm to 8:15pm and then the exam will be from 8:30pm to 10:00pm. Please note that the lecture and the exam will be held in CLH F on June 26th.