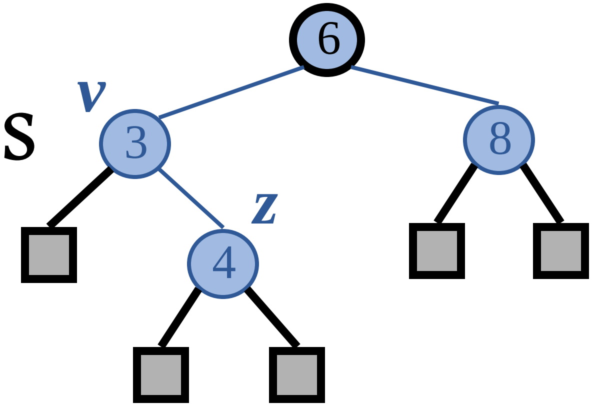
CPSC 131

Data Structures

AVL Trees



Tree Depth and Height and BST Issue

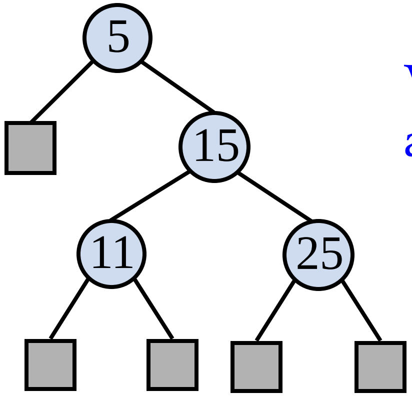
Depth

- Depth of a node: number of ancestors (between the node and the root)
  - Root has depth 0

- Depth of p's node is recursively defined:

```
if (p.isRoot()) return 0;              // root has depth 0
else return 1 + depth(p.parent());    // 1 + (depth of parent)
```

```
SearchTree<Entry<int, string>> ds;
ds.insert(5, "Monday");
ds.insert(15, "Tuesday");
ds.insert(25, "Wednesday");
ds.insert(11, "Thursday");
```



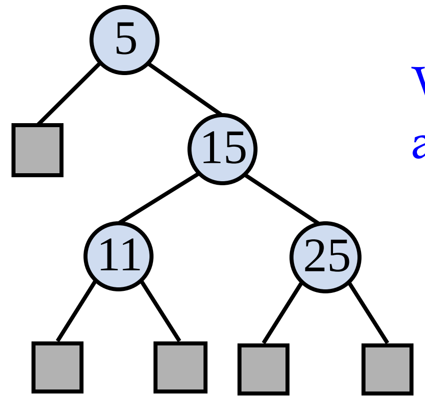
What is the depth at each node?

Height

- Height of a node p in a tree T is defined recursively:
  - If p is the external node, then the height of p is 0
  - Otherwise, the height of p is 1 + the maximum height of children of p

- Height of a tree == maximum depth of its external nodes
  - Height of a tree T is the height of the root of T

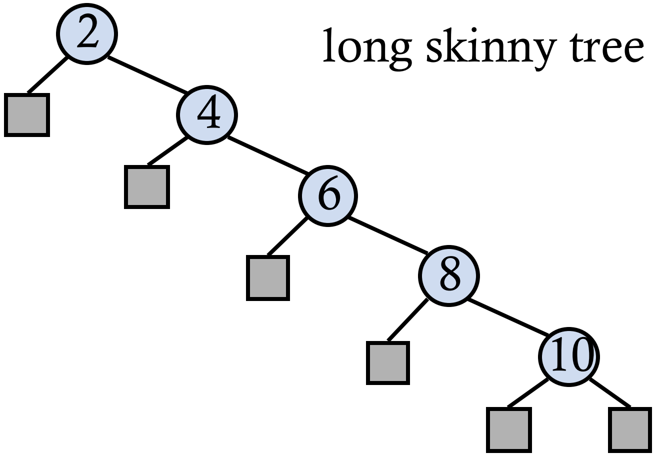
```
SearchTree<Entry<int, string>> ds;
ds.insert(5, "Monday");
ds.insert(15, "Tuesday");
ds.insert(25, "Wednesday");
ds.insert(11, "Thursday");
```



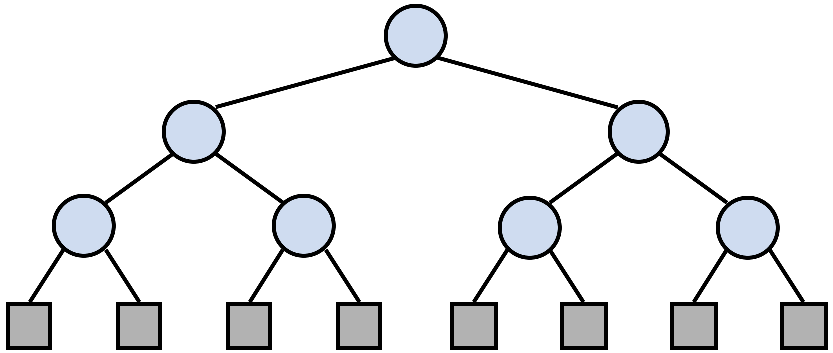
What is the height at each node?

Issue with Binary Search Tree Performance

- A binary search tree (BST) with ***n*** items
  - The space used is $O(n)$
  - Methods find, insert and erase take $O(height)$ time
  - height = n, worst case ➔ long skinny tree
  - Example: insert 2, 4, 6, 8, 10 into a BST
  - What is the issue? Lack of balance!
  - height = log n, best case ➔ balanced tree



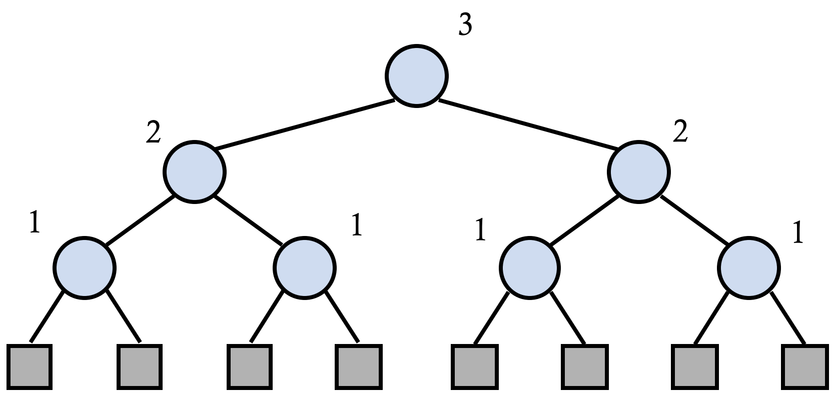
long skinny tree



balanced tree

Perfect Binary Search Tree

Height h	Number of Nodes n
1	1
2	3
3	7
4	15
h	$2^h - 1$



$$n = 2^h - 1$$
$$n + 1 = 2^h - 1 + 1$$
$$n + 1 = 2^h$$
$$\log_2(n + 1) = \log_2(2^h)$$
$$\log_2(n + 1) = h$$

➔ In a perfect tree, h is O(log n)

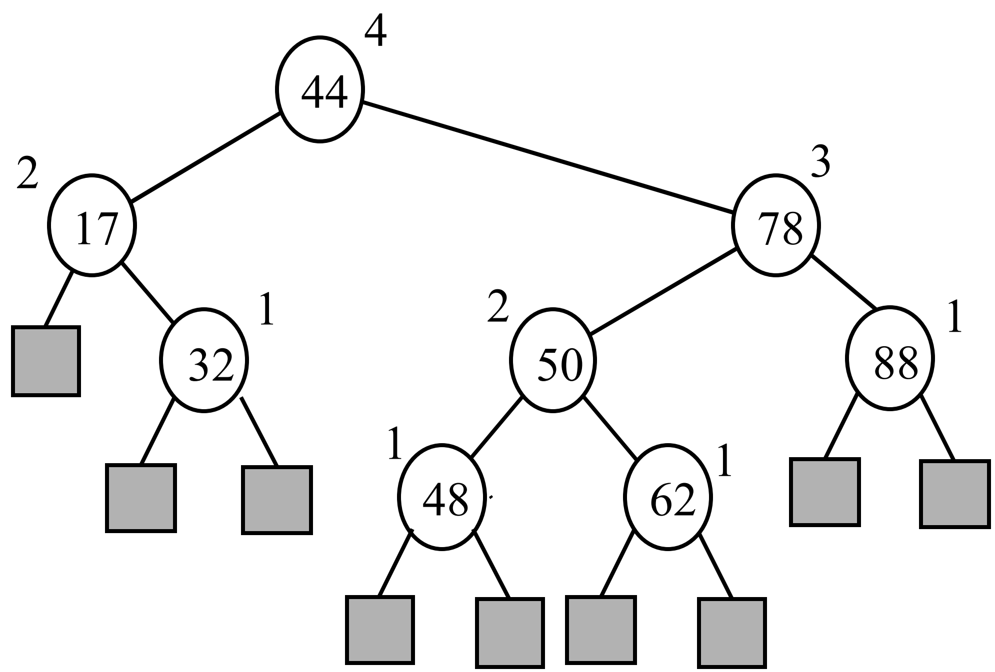
Self-Balancing Trees

- Always ensure that BST is balanced
  - IF an insert (or delete) makes the BST not balanced, then *rearrange the nodes* so that the tree stays balanced
  - Challenge: the node rearrangement should not take too long!
    - Otherwise, lost the performance benefit
  - Node rearrangement still keeps the same nodes in sorted order but in a different layout

- May types of balanced trees
  - AVL trees
    - The first self-balancing tree
    - Invented in 1962 by Russian mathematicians Georgy **A**delson-**V**elsky and Evgenii **L**andis
  - Red-Black trees
  - Splay trees
  - (2,4) trees

AVL Trees

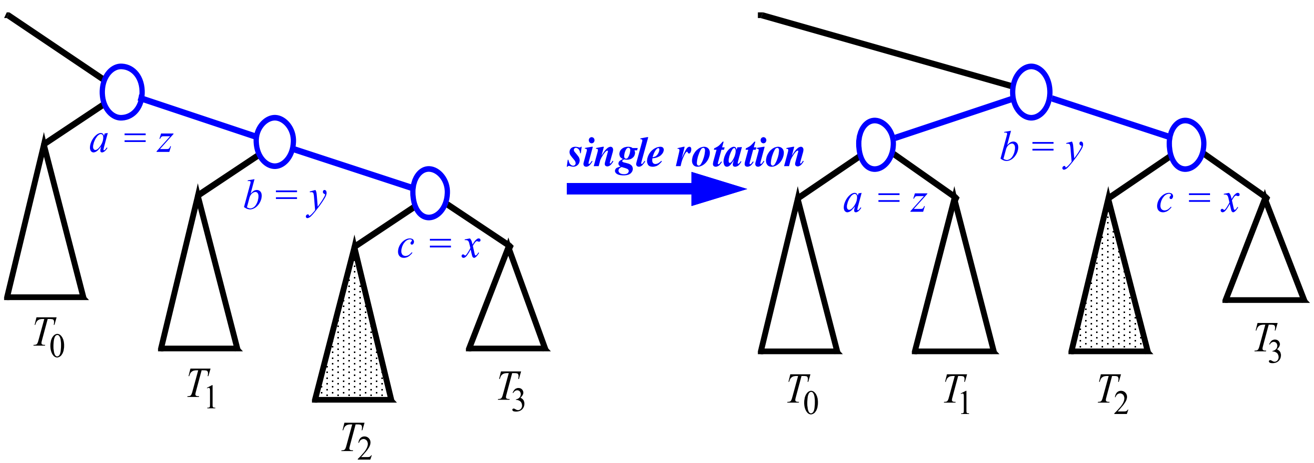
- AVL trees are height-balanced binary search trees such that for every internal node v of T, the heights of left and right subtrees of v can differ by at most 1
  - 1 + max of (height(left subtree) and height(right subtree))
- Balance factor of a node calculated at each node
  - abs(height(left subtree) - height(right subtree))



good but not perfect balanced

Trinode Restructuring - Rotations

- A single insert or delete will at most upset balance to | 2 |
- Balancing a tree after an insertion or deletion will be through rotations



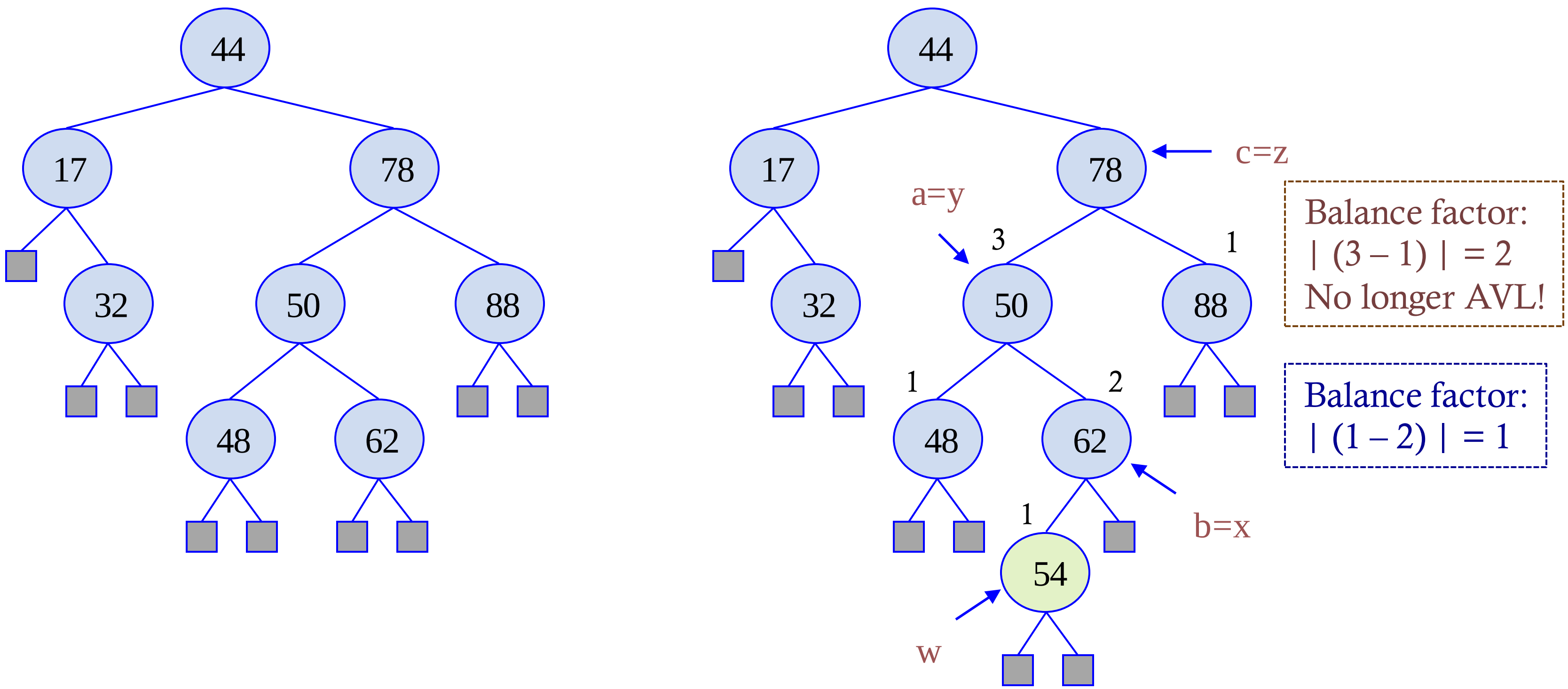
A rotations maintains the inorder of nodes, but makes for a more balanced tree

Insert and Rotation in AVL Trees

- Insert operation may cause balance factor to become an absolute value of 2 for some node
  - Only nodes on the path from insertion point to root node have possibly changed in height
  - So after the Insert, go back up to the root node by node, updating heights
  - If a new balance factor (the difference abs(h_{left}-h_{right})) is 2, adjust tree by *rotation* around the node

Insertion

- Inserting a new key-value same as in a binary search tree
  - Always done by expanding an external node.
- Example: insert key 54

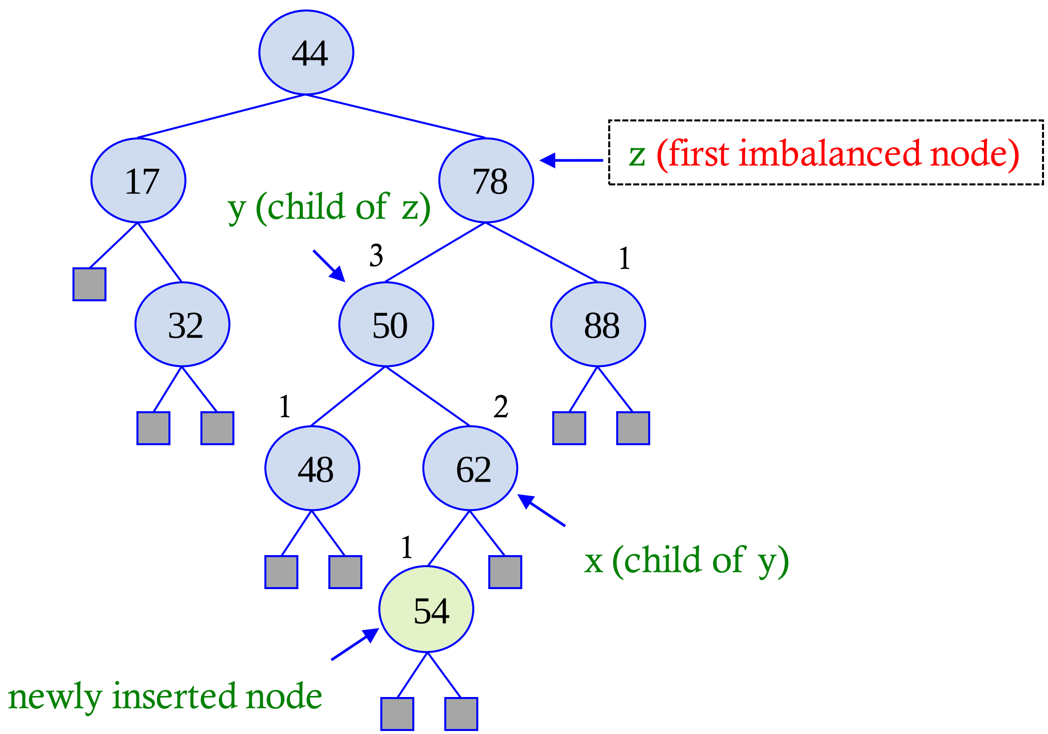


before insertion

after insertion

Trinode Restructuring

Focus on only three nodes

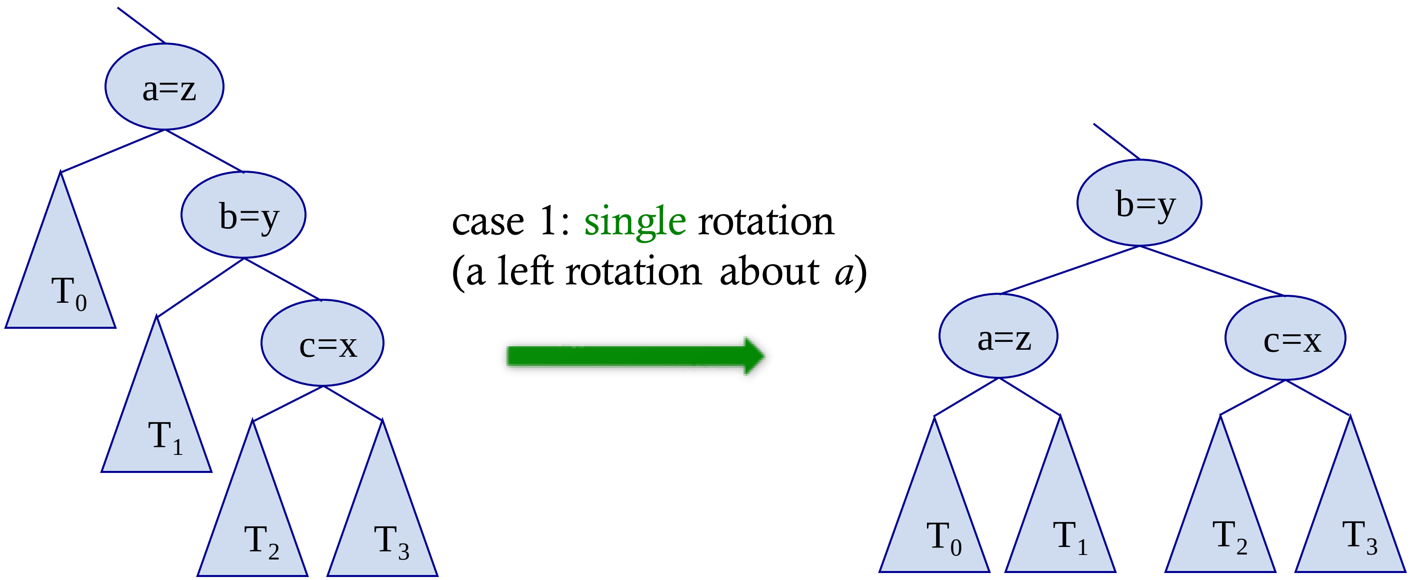


4 combinations: Y and X can be left/right child of Z

Trinode Restructuring

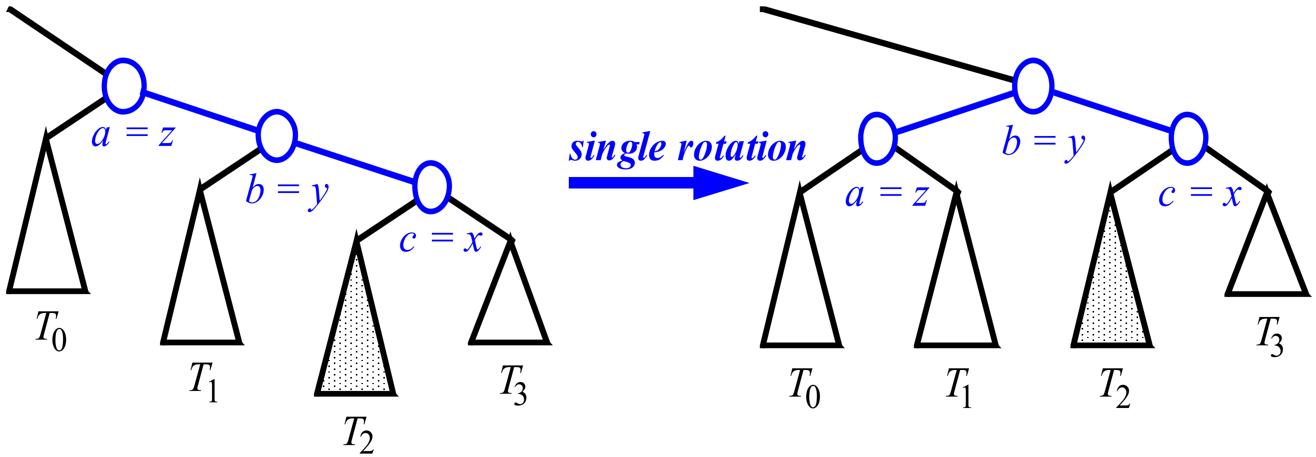
Input: A node x of a binary search tree T that has a parent y and a grandparent z
Output: Tree T after a trinode restructuring (a single or double rotation) involving nodes, x, y, and z

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

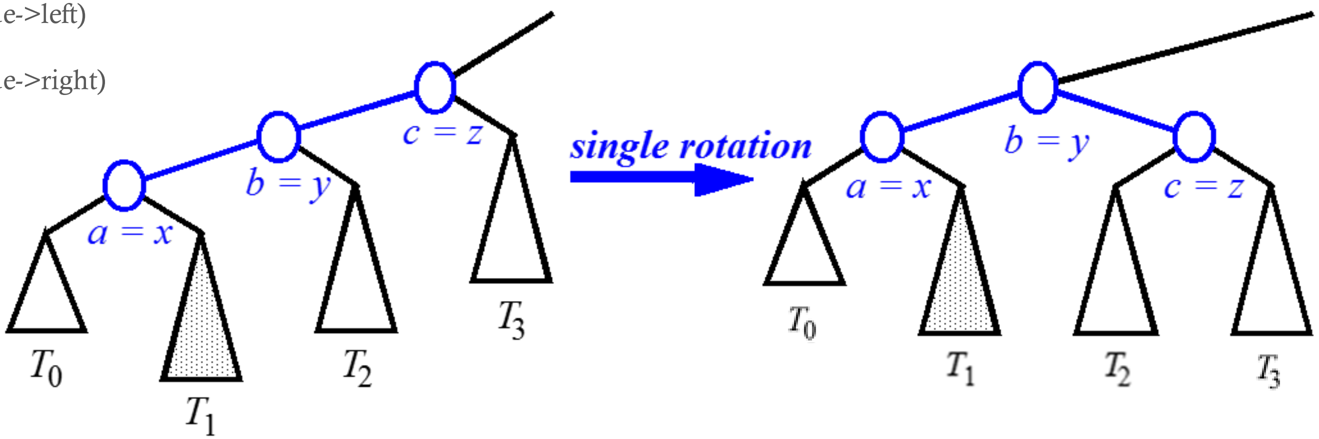


(rotating a left slanting tree is similar – see next slide)

Restructuring – as Single Rotations



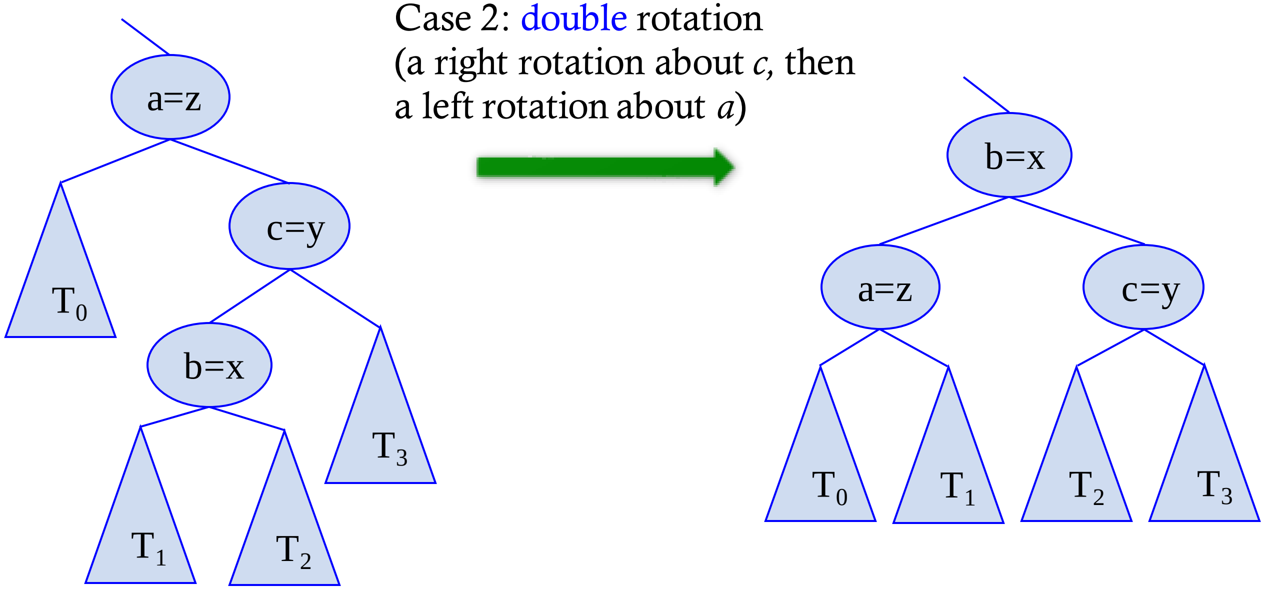

```
inorder(node):
if node == NULL
return
inorder(node->left)
visit(node)
inorder(node->right)
```



Trinode Restructuring

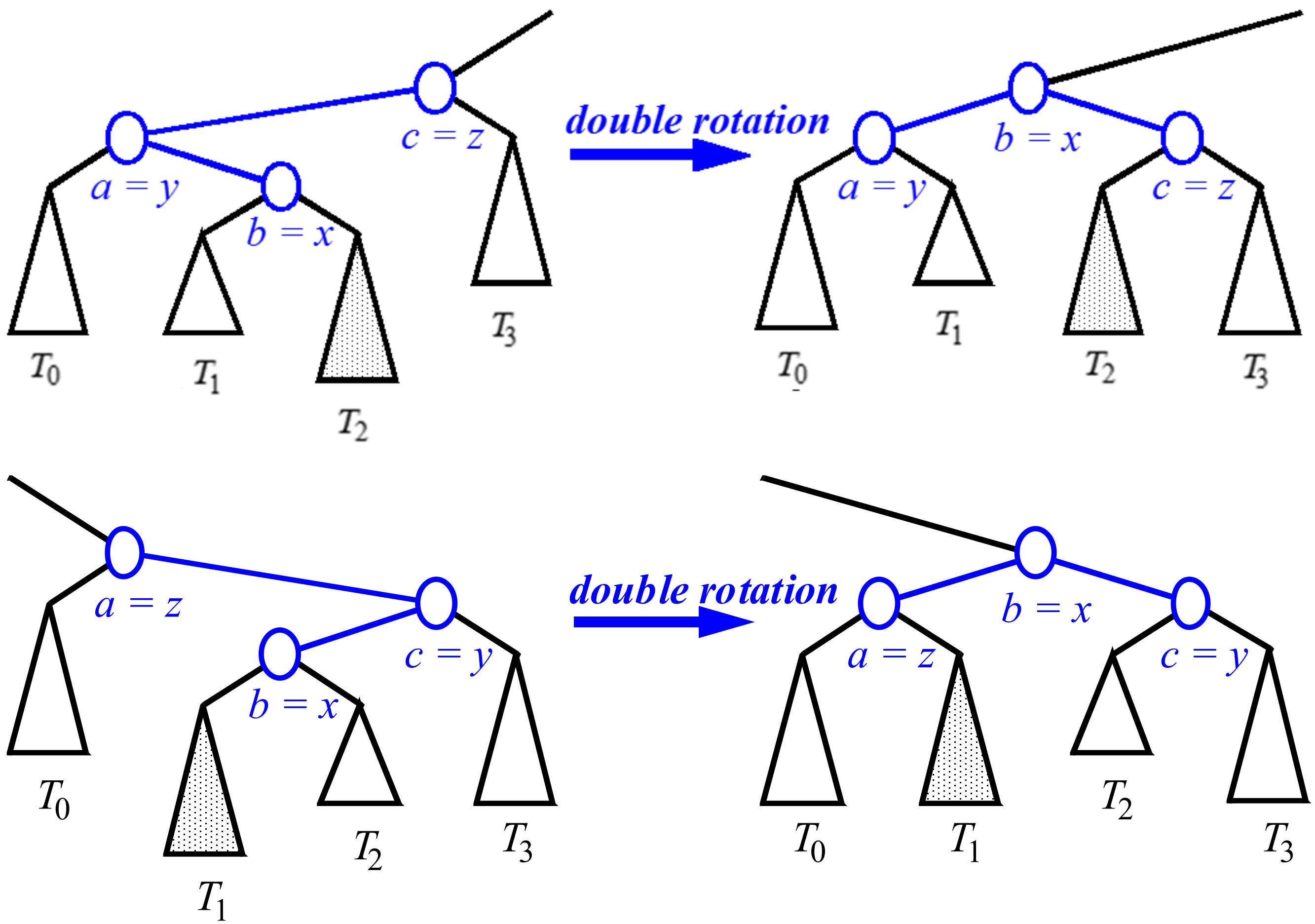
- Let (a,b,c) be an inorder listing of x, y, z
- Perform the rotations needed to make b the topmost node of the three

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

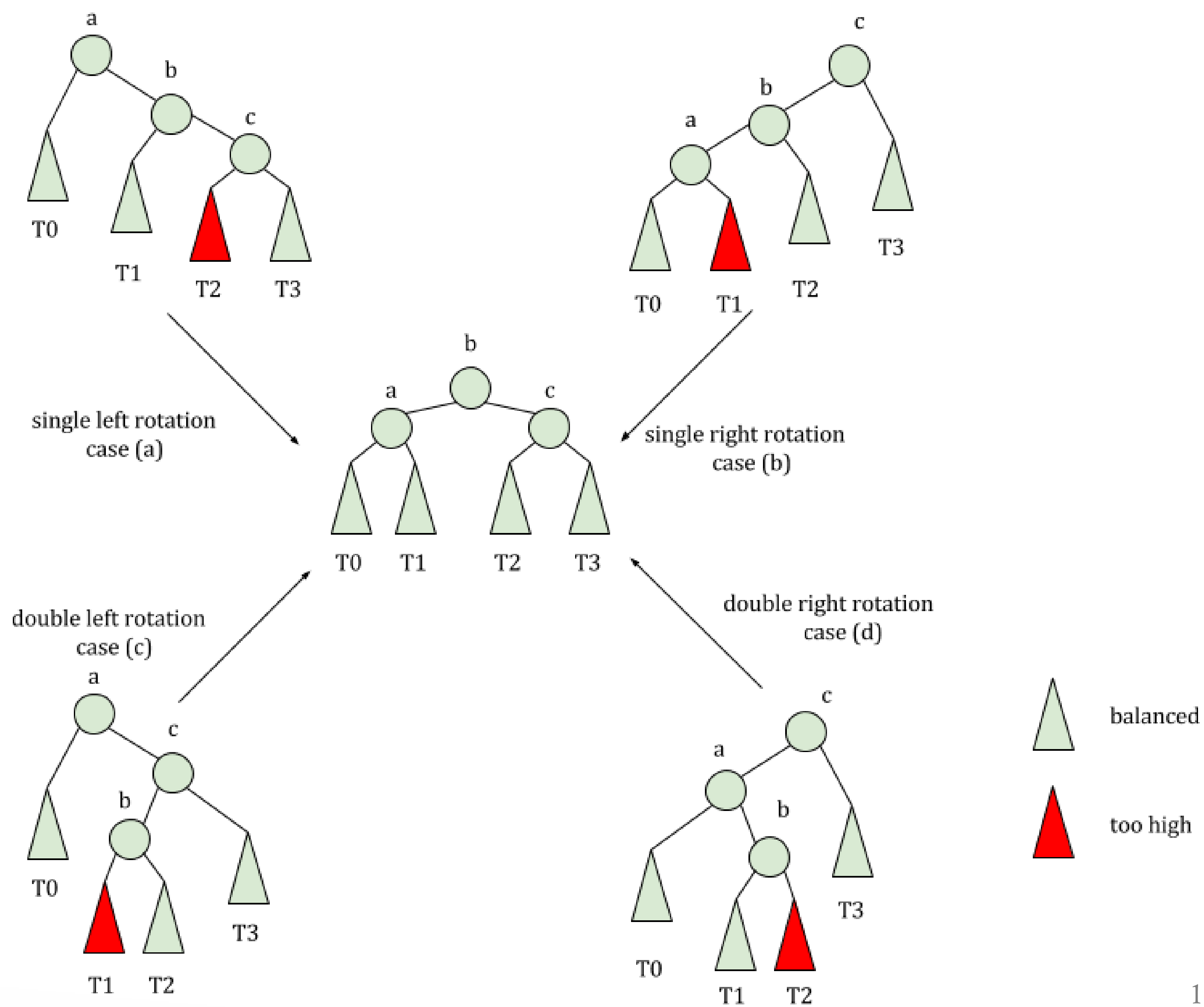


(left skewed tree is similar – next slide)

Restructuring – as Double Rotations



AVL Tree Trinode Restructuring



Trinode Restructuring Pseudocode

restructure (*x,y,z*) { // *z* is parent of *y*, *y* is parent of *x*

// Don't need 4 separate left/right child cases

Let (*a,b,c*) be inorder listing of the nodes *x,y,z*

Let T0,T1,T2,T3 be the subtrees below x,y,z from left to right

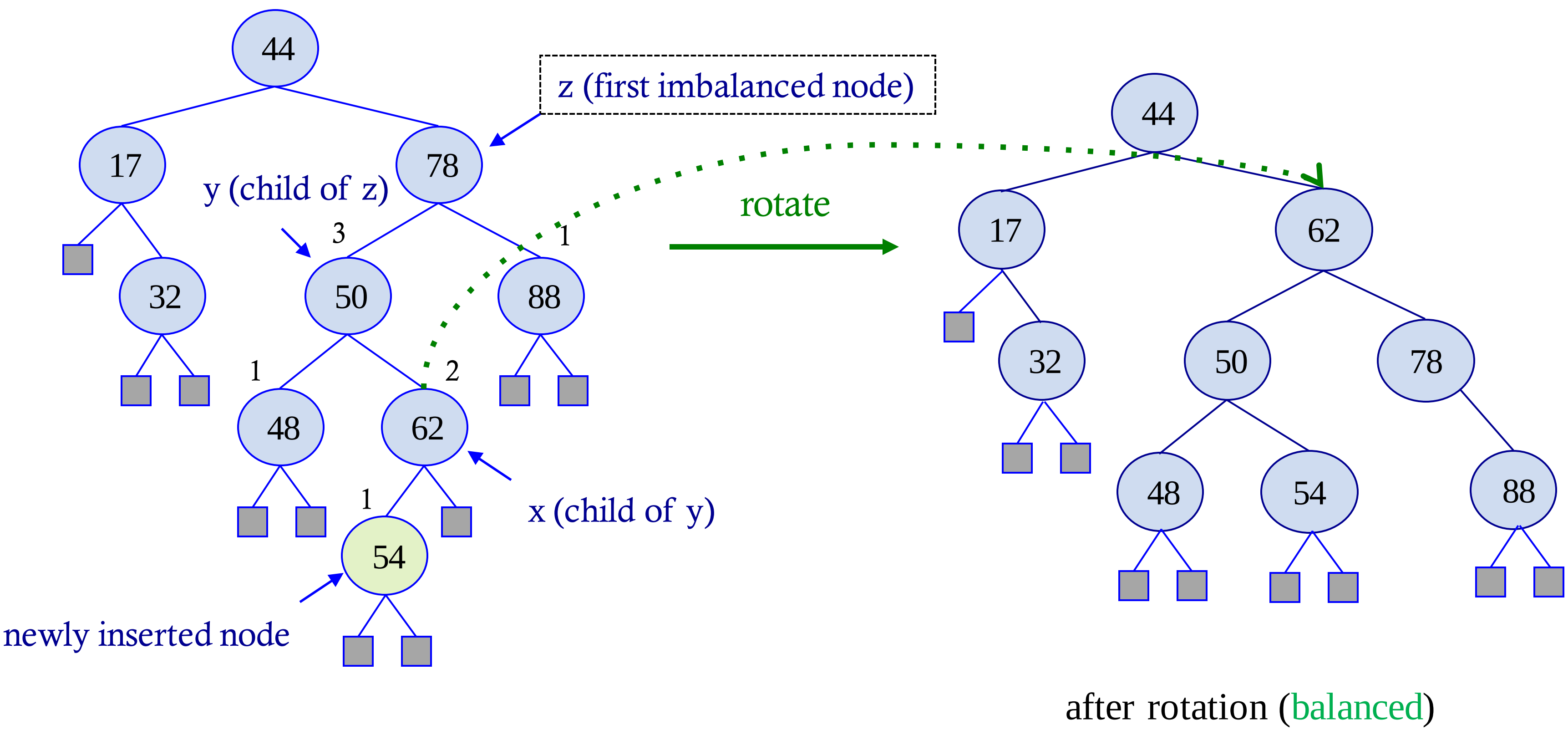
// The rotation:

1. Replace subtree at *z* with subtree at *b*
2. Make *a* left child of *b*; make T0 and T1 subtrees of *a*
3. Make *c* right child of *b*; make T2 and T3 subtrees of *c*

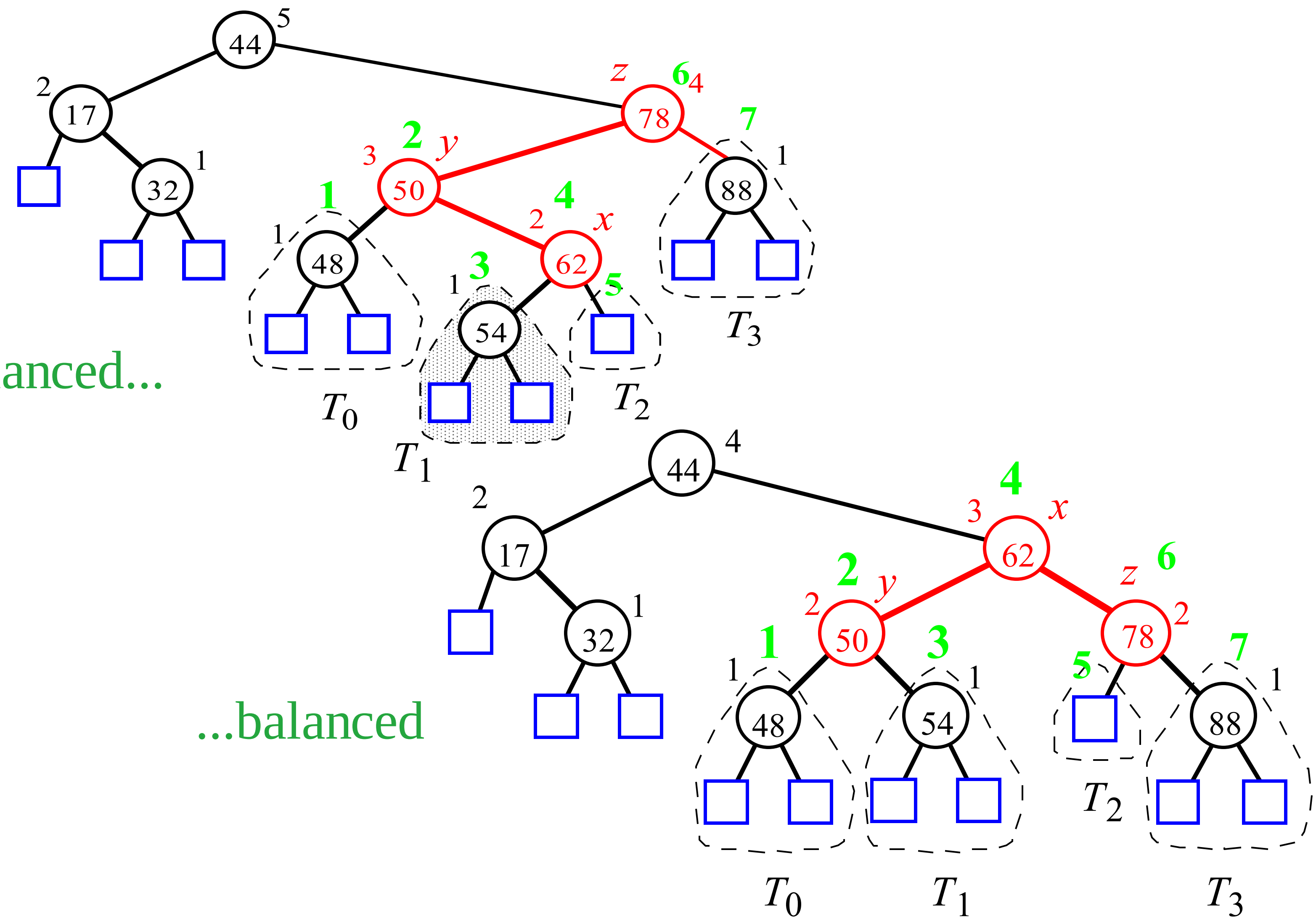
Interactive visualization ⇨ http://visualgo.net/bst

Trinode Restructuring (cont)

Focus on only three nodes



Insertion Example

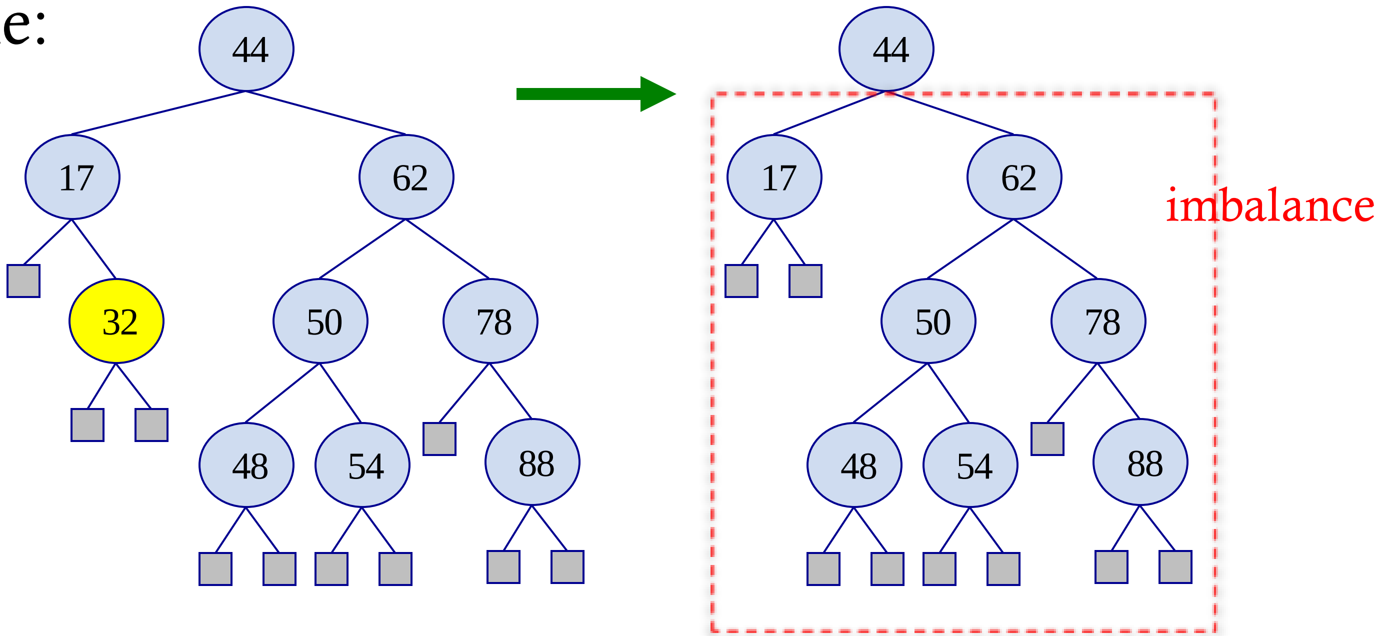


unbalanced...

...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:

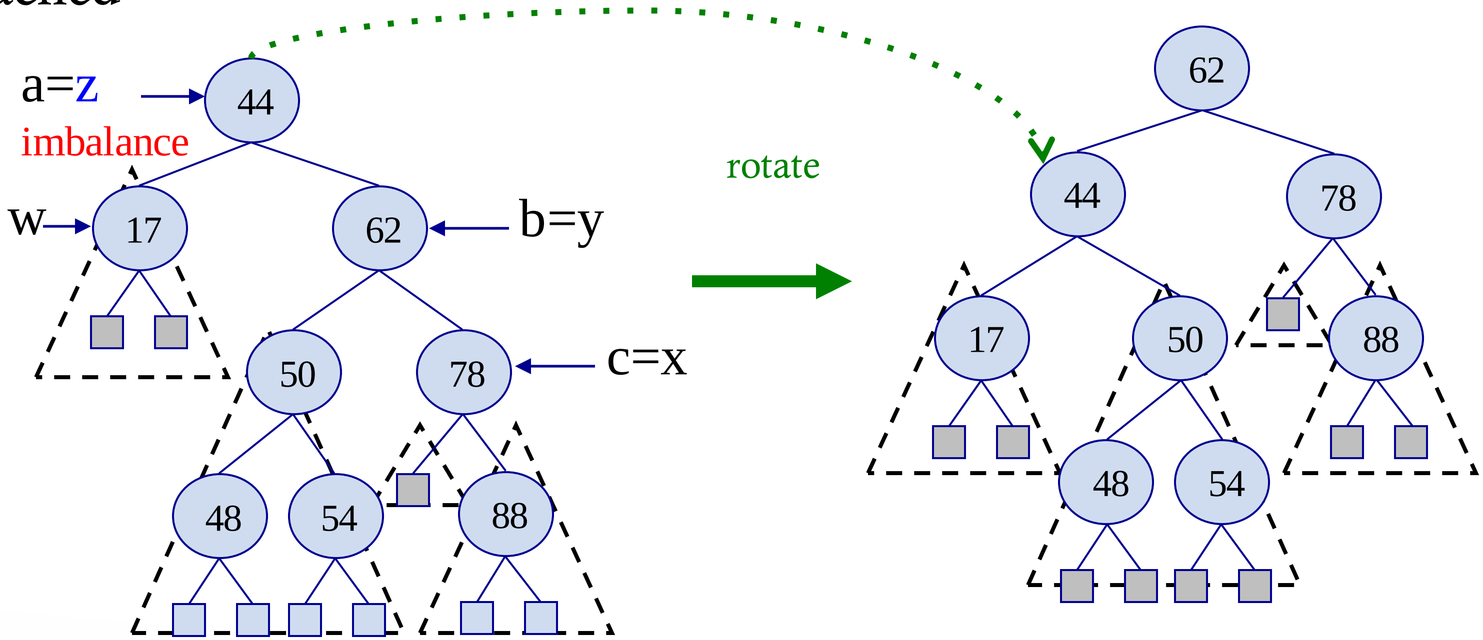


before deletion of 32

after deletion

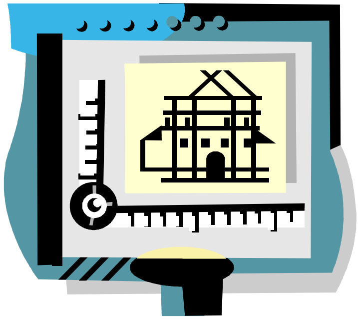
Rebalancing after a Removal

- 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 restructure(x) to restore balance at z
- If this restructuring upsets the balance of another node higher in the tree, we would continue checking for balance until the root of T is reached



AVL Tree Performance

- a single restructure takes O(1) time
  - using a linked-structure binary tree
- find takes O(log n) time
  - height of tree is O(log n), no restructures needed
- put takes O(log n) time
  - initial find is O(log n)
  - Restructuring up the tree, maintaining heights is O(log n)
- erase takes O(log n) time
  - initial find is O(log n)
  - Restructuring up the tree, maintaining heights is O(log n)



References

- Data Structures and Algorithms in C++, 2^{nd} Edition by Goodrich, Tamassia, and Mount
- Sections: 7.2, 10.2