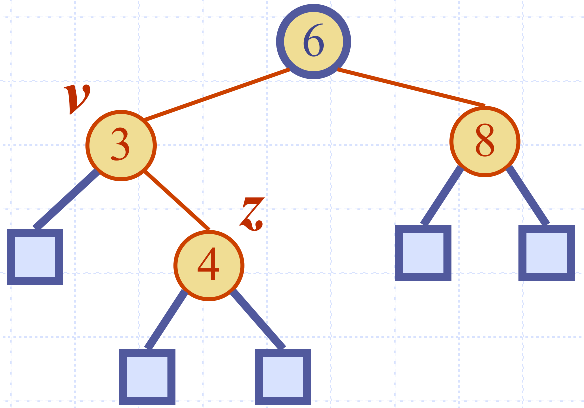
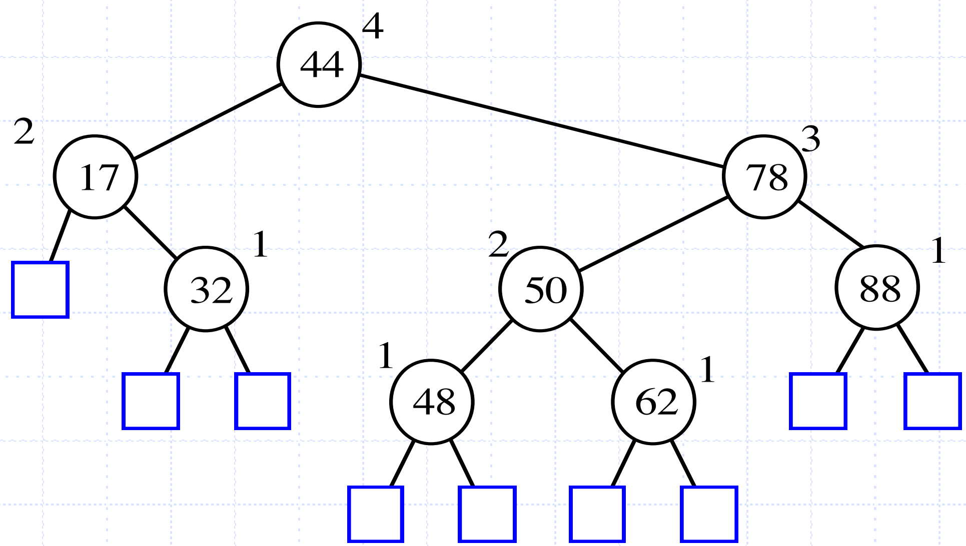
Presentation for use with the textbook Algorithm Design and Applications, by M. T. Goodrich and R. Tamassia, Wiley, 2015

AVL Trees



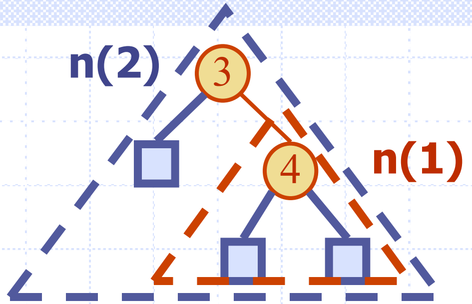
AVL Tree Definition

- AVL trees are rank-balanced trees.
- The **rank**, r(v), of each node, v, is its height.
- **Rank-balance rule**: An AVL Tree is a binary search tree such that for every internal node v of T, the heights (ranks) of the children of v can differ by at most 1.



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

Height of an AVL Tree



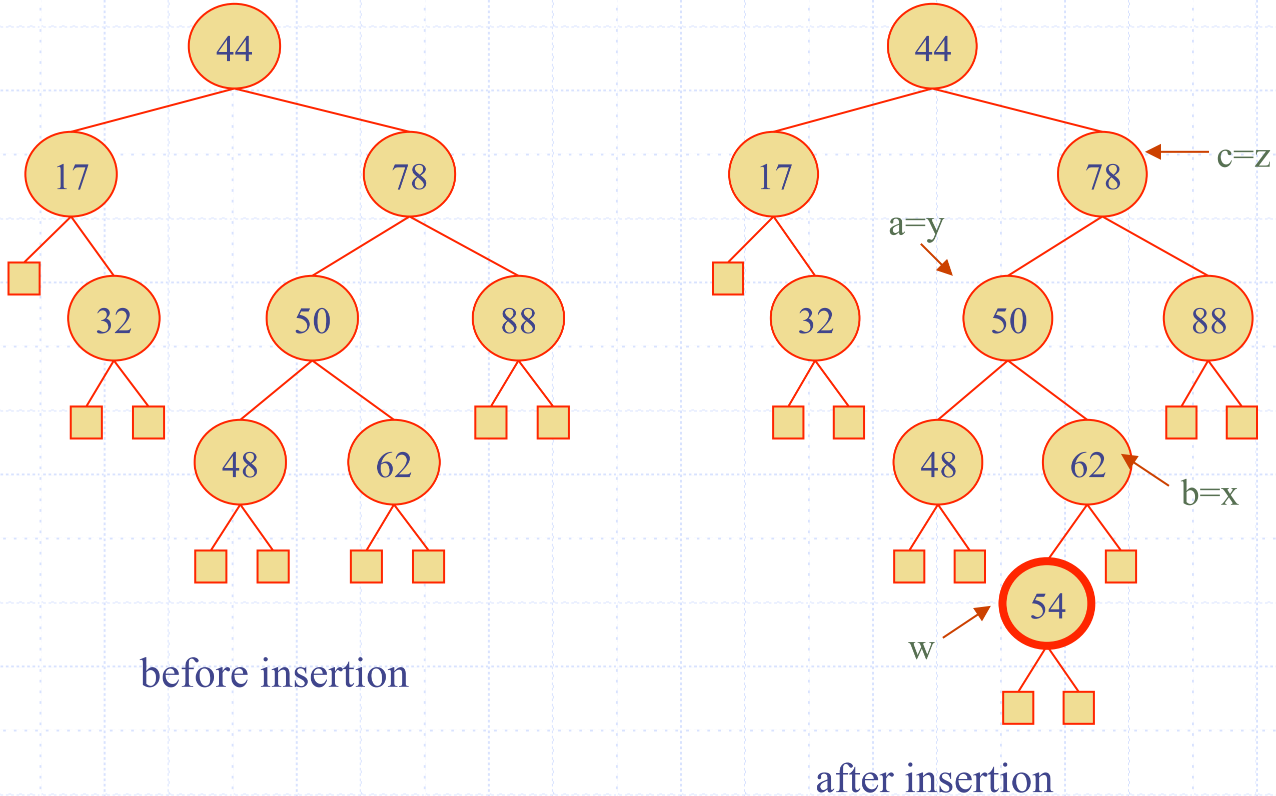
Fact: The height of an AVL tree storing n keys is O(log n).

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)

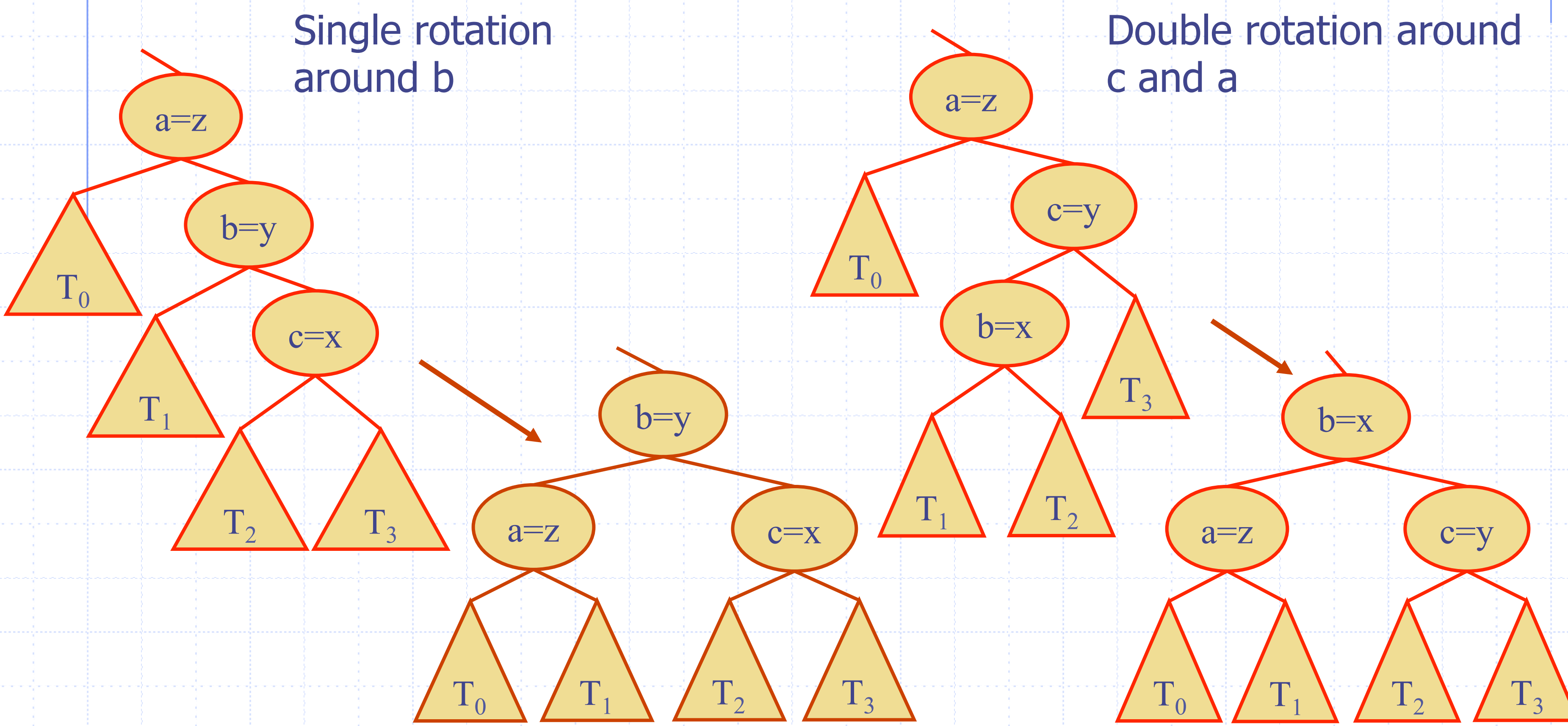
Insertion

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



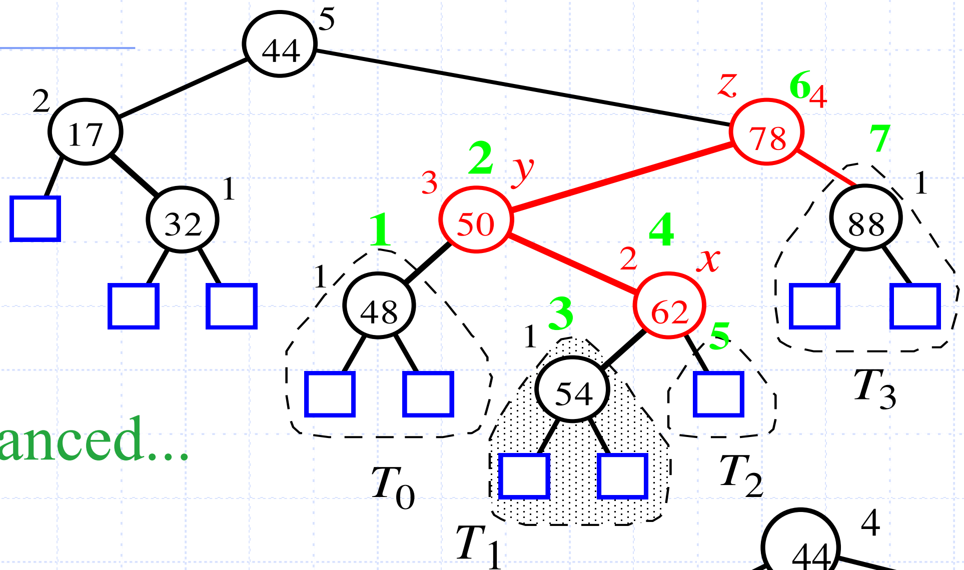
Trinode Restructuring

- 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

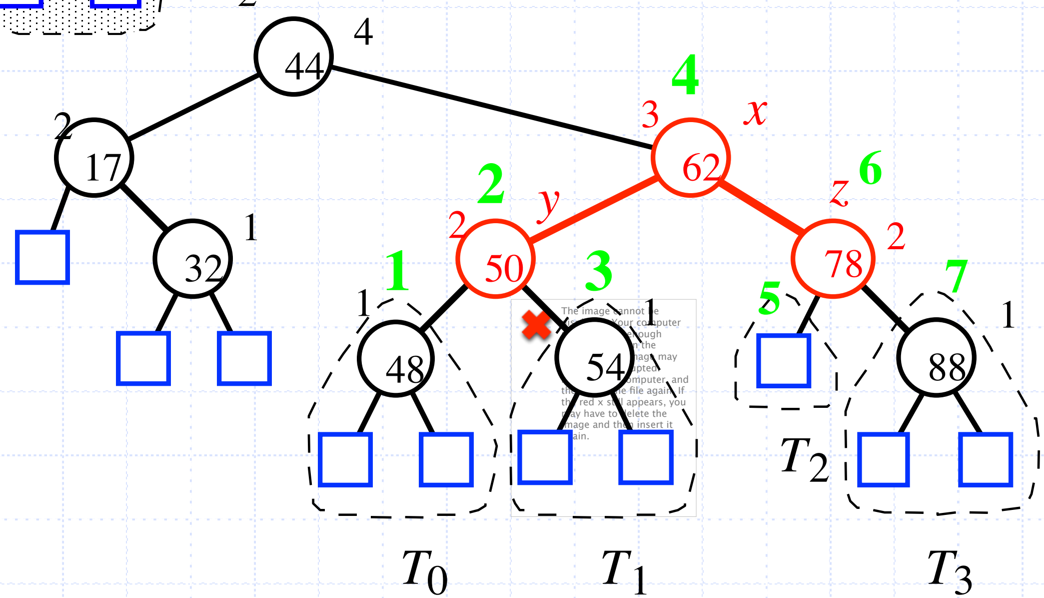


Insertion Example, continued

unbalanced...

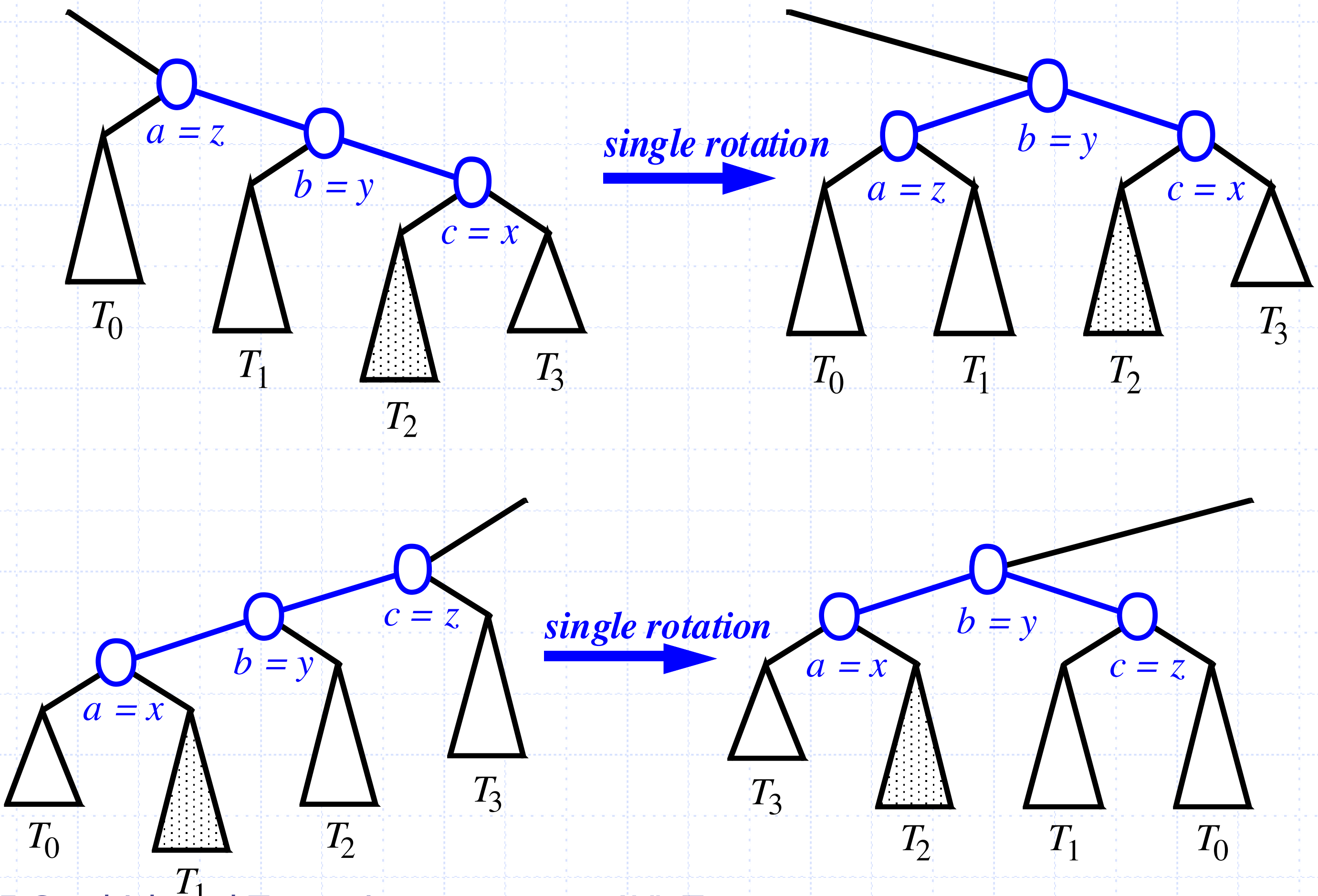


...balanced



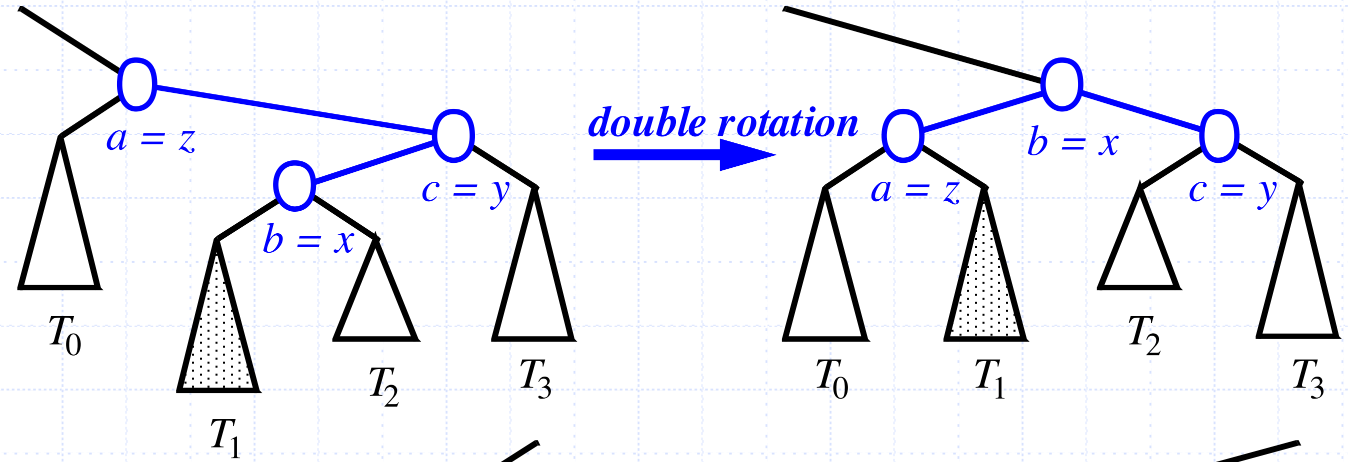
Restructuring (as Single Rotations)

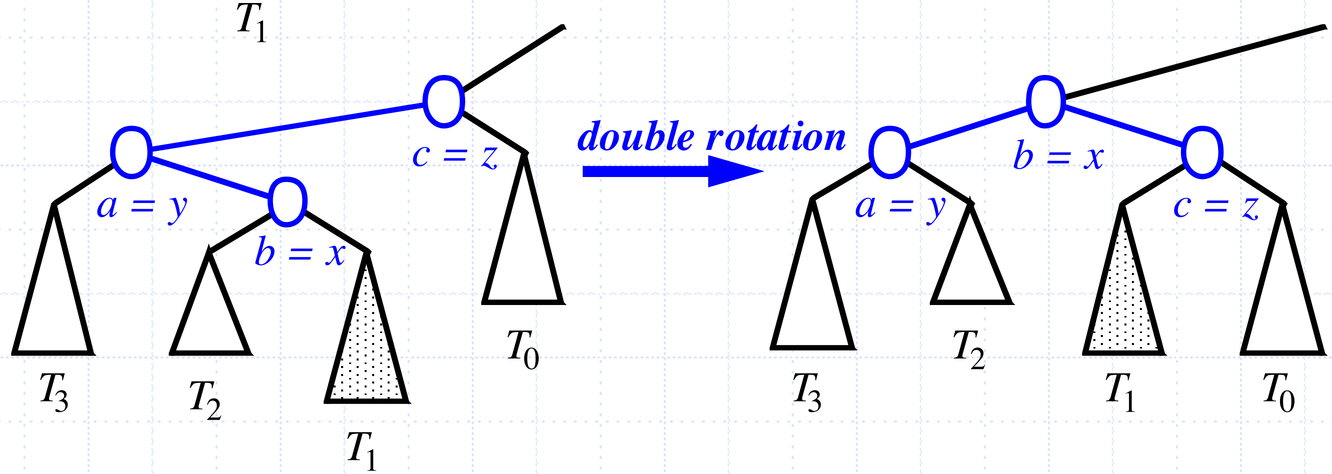
- Single Rotations:



Restructuring (as Double Rotations)

- double rotations:





Pseudo-code

- Insertion.

Algorithm insertAVL(k, e, T):
  Input: A key-element pair, (k, e), and an AVL tree, T
  Output: An update of T to now contain the item (k, e)

```
v ← IterativeTreeSearch(k, T)
if v is not an external node then
    return "An item with key k is already in T"
Expand v into an internal node with two external-node children
v.key ← k
v.element ← e
v.height ← 1
rebalanceAVL(v, T)
```

Pseudo-code

- Rebalance at a node violating the rank rule.

Algorithm rebalanceAVL(v, T):

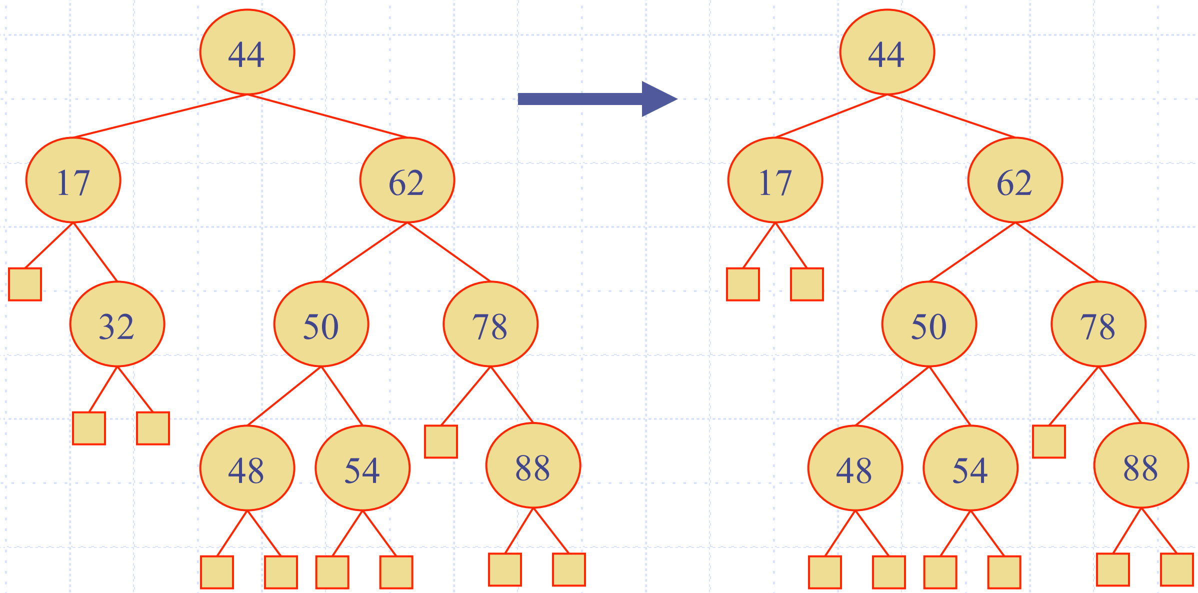
Input: A node, v, where an imbalance may have occurred in an AVL tree, T

Output: An update of T to now be balanced

```
v.height ← 1 + max{v.leftChild().height, v.rightChild().height}
while v is not the root of T do
    v ← v.parent()
    if |v.leftChild().height − v.rightChild().height| > 1 then
        Let y be the tallest child of v and let x be the tallest child of y
        v ← restructure(x)        // trinode restructure operation
    v.height ← 1 + max{v.leftChild().height, v.rightChild().height}
```

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:

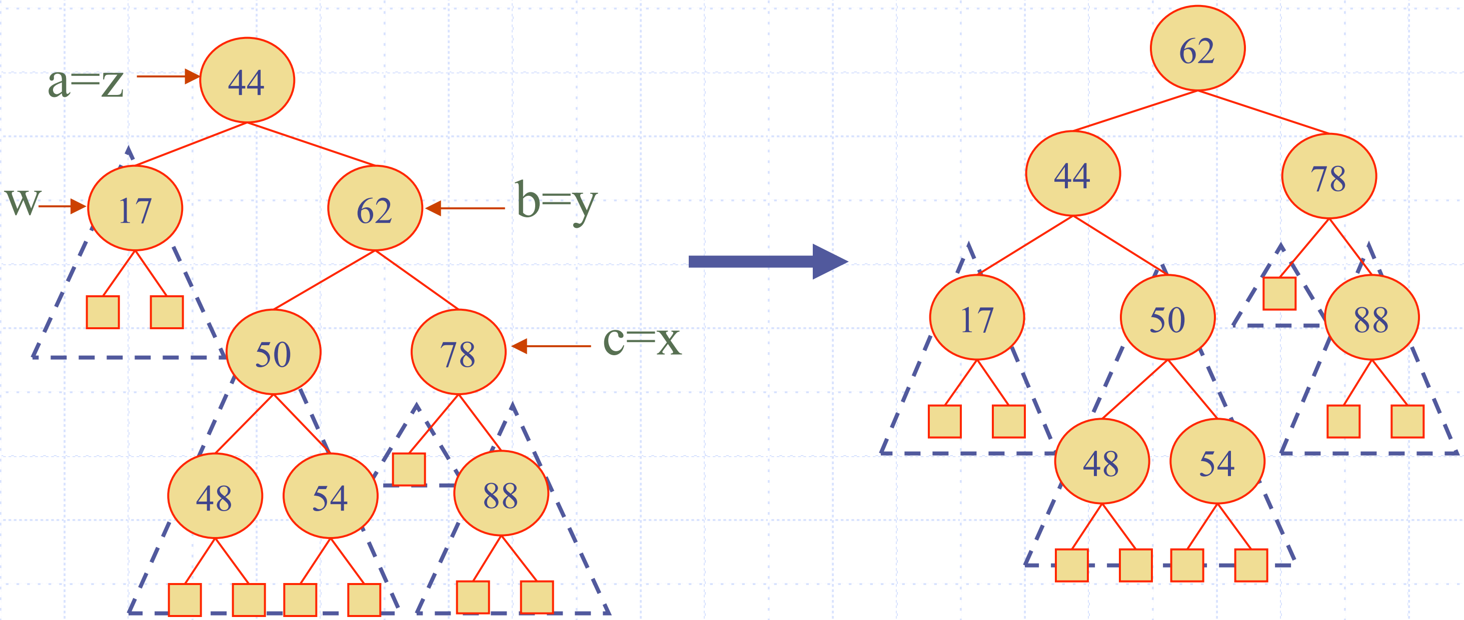


before deletion of 32

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



Pseudo-code

- Removal

Algorithm removeAVL(k, T):
  Input: A key, k, and an AVL tree, T
  Output: An update of T to now have an item (k, e) removed

  $v \leftarrow$ IterativeTreeSearch(k, T)
  if v is an external node **then**
    return "There is no item with key k in T"
  if v has no external-node child **then**
    Let u be the node in T with key nearest to k
    Move u's key-value pair to v
    $v \leftarrow u$
  Let w be v's smallest-height child
  Remove w and v from T, replacing v with w's sibling, z
  rebalanceAVL(z, T)

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)
    - restructuring up the tree, maintaining heights 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)