AVL Trees

Dr. Aiman Hanna

Department of Computer Science & Software Engineering

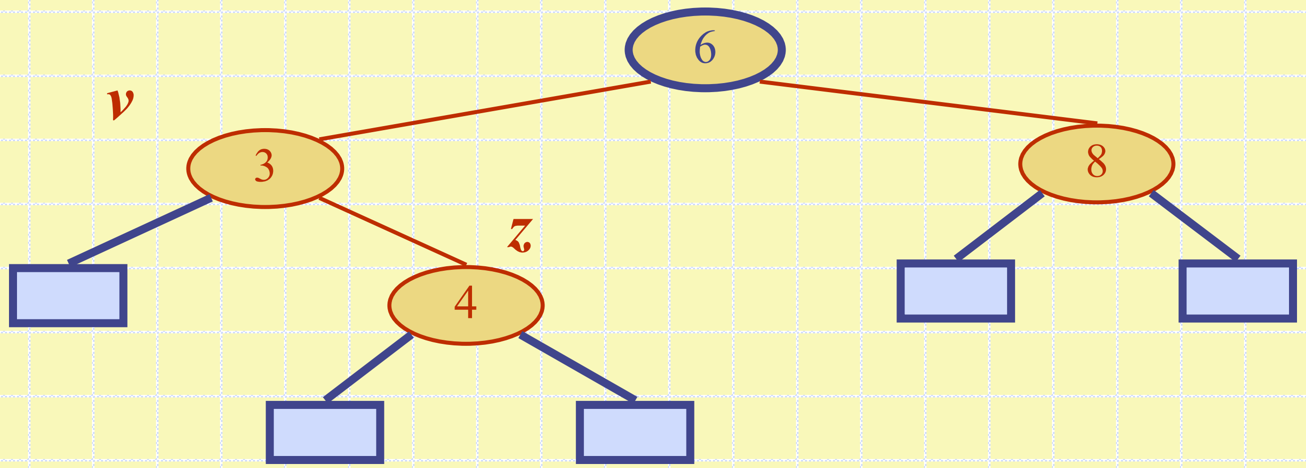
Concordia University, Montreal, Canada

These slides have been extracted, modified and updated from original slides of :
Data Structures and Algorithms in Java, 5th edition. John Wiley& Sons, 2010. ISBN 978-0-470-38326-1.
Data Structures and the Java Collections Framework by William J. Collins, 3rdedition, ISBN 978-0-470-48267-4.
Both books are published by Wiley.

Copyright © 2010-2011 Wiley
Copyright © 2010 Michael T. Goodrich, Roberto Tamassia
Copyright © 2011 William J. Collins
Copyright © 2011-2021 Aiman Hanna
All rights reserved

Coverage

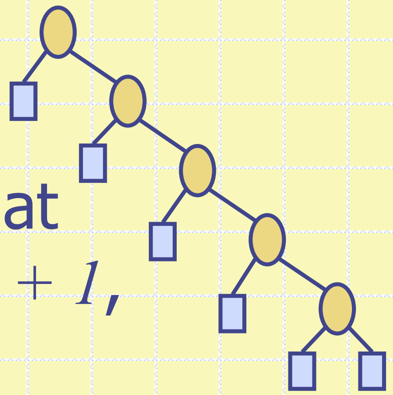
- Section 10.2: AVL Trees



AVL trees are named after its their Soviet inventors, G. M. Adelson-Velskii and E. M. Landis, in 1962. AVL Trees were the first invented self-balancing tress.

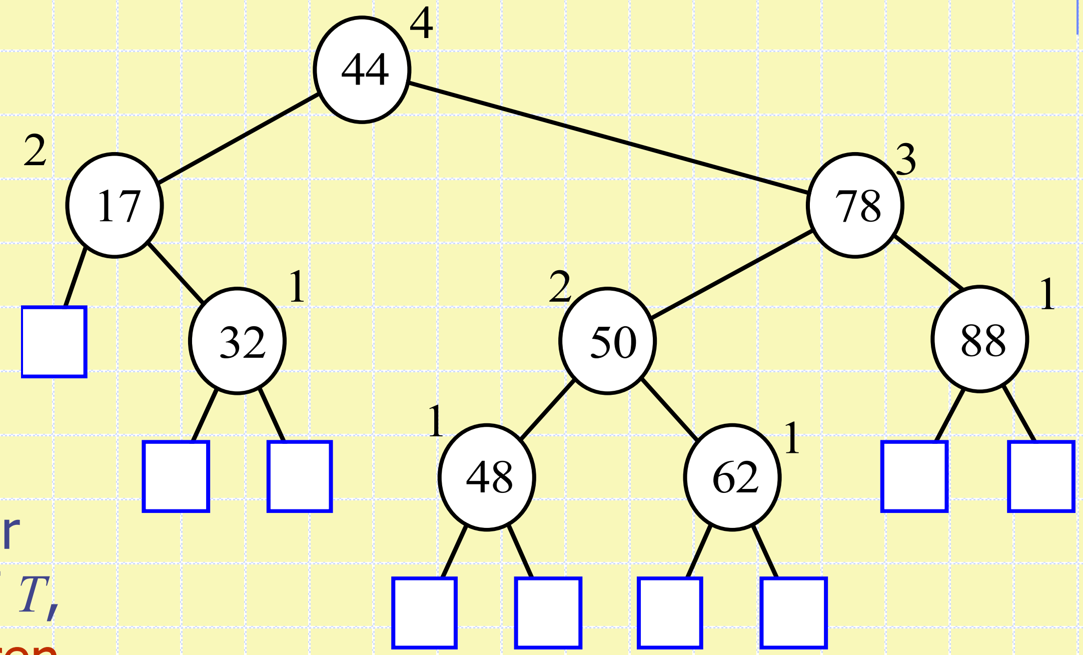
Performance of Binary Search Trees

- A binary search tree should be an efficient data structure for map implementation.
- However, as seen, binary search trees may have $O(n)$ in the worst case, which is not any better than list-based or array-based map implementation.
- That problem is caused by the possibility that the nodes may be arranged such that $n = h + 1$, where h is the height of the tree.



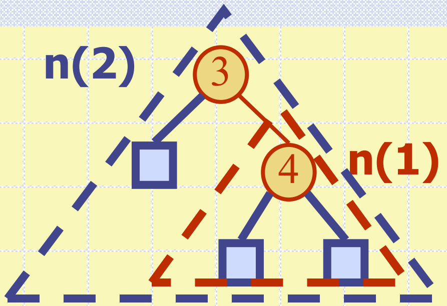
AVL Tree Definition

- The performance problem of binary search trees can be corrected using AVL trees.
- AVL Trees are balanced 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; this is referred to as the ***Height-Balance Property***.

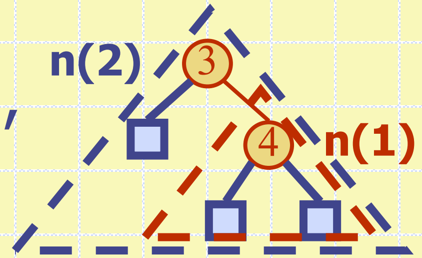


An example of an AVL tree where the heights are shown next to the nodes

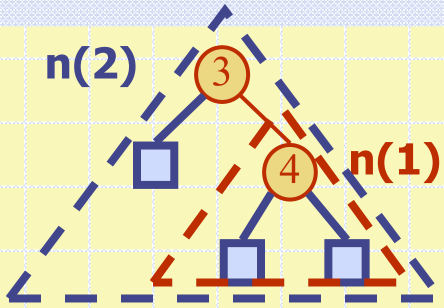
Height of an AVL Tree



- Any binary search tree that satisfies the height-balance property is said to be an AVL tree.
- AVL trees are named after their inventors (**A**del'son **V**el'skii and **L**andis).
- Fact: The height of an AVL tree storing n keys is $O(\log n)$.
- Proof: Let us bound $n(h)$: the minimum number of internal nodes of an AVL tree of height h.
- We can easily see that $n(1) = 1$ and $n(2) = 2$, since an AVL tree of height 1 must have at least 1 internal node and an AVL of height 2 must have at least 2 internal nodes.



Height of an AVL Tree



- Now, for $n >= 3$, an AVL tree of height h contains the root node, and at minimum, one AVL subtree of height $h-1$ and another of height $h-2$.

- That is, for $n >= 3$:

$$n(h) = 1 + n(h-1) + n(h-2)$$

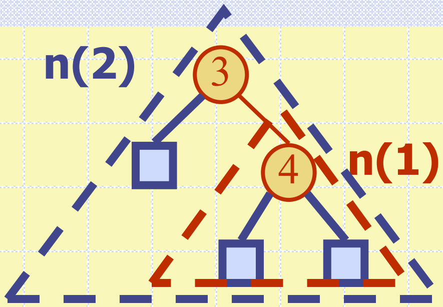
- Knowing that $n(h-1) > n(h-2)$, we get $n(h) > 2n(h-2)$, which indicates that: $n(h)$ at least doubles each time h increases by 2.

- So:

$$n(h) > 2n(h-2),\ n(h) > 4n(h-4),\ n(h) > 8n(h-6),\ \ldots \text{ (by induction)},$$

$$\rightarrow n(h) > 2^i n(h-2i)$$

Height of an AVL Tree



➔ $n(h) > 2^i n(h-2i)$, for any integer i such that $h-2i >= 1$.

- Recall that we already know $n(1)$ and $n(2)$.
- So, let us pick up i such that $h - 2i$ is equal to either 1 or 2:

$$i = ceil(h/2) - 1$$

➔ $n(h) > 2^i n(h-2i)$ ➔ $n(h) > 2^{ceil(h/2) - 1}\ n(h- (2ceil(h/2)) +2)$

➔ $n(h) > 2^{ceil(h/2) - 1}\ n(h- (2ceil(h/2)) +2)$

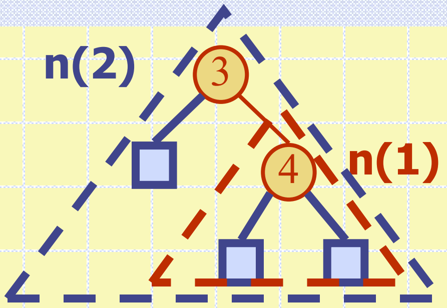
➔ $n(h) > 2^{ceil(h/2) - 1}\ n(h- (ceil(h/2) + ceil(h/2)) +2)$

➔ $n(h) > 2^{ceil(h/2) - 1}\ n(h- (ceil(h/2) -1 + ceil(h/2) -1))$

➔ $n(h) > 2^{ceil(h/2) - 1}\ n(h- (2i))$ ➔ $n(h) >= 2^{ceil(h/2) - 1}\ n(1)$

➔ $n(h) >= 2^{ceil(h/2) - 1}$

Height of an AVL Tree



➔ $n(h) >= 2^{ceil(h/2) - 1}$ ➔ $n(h) >= 2^{h/2 - 1}$

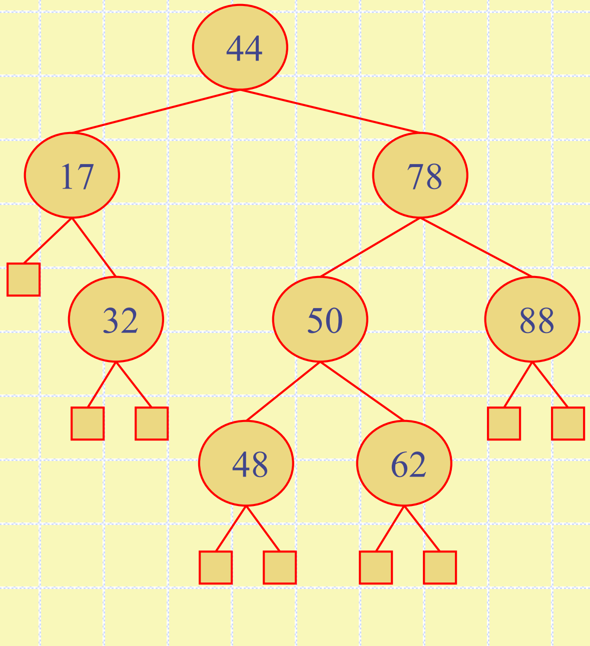
➔ $\log n(h) > (h/2) - 1$

➔ $h < 2 \log n(h) + 2$

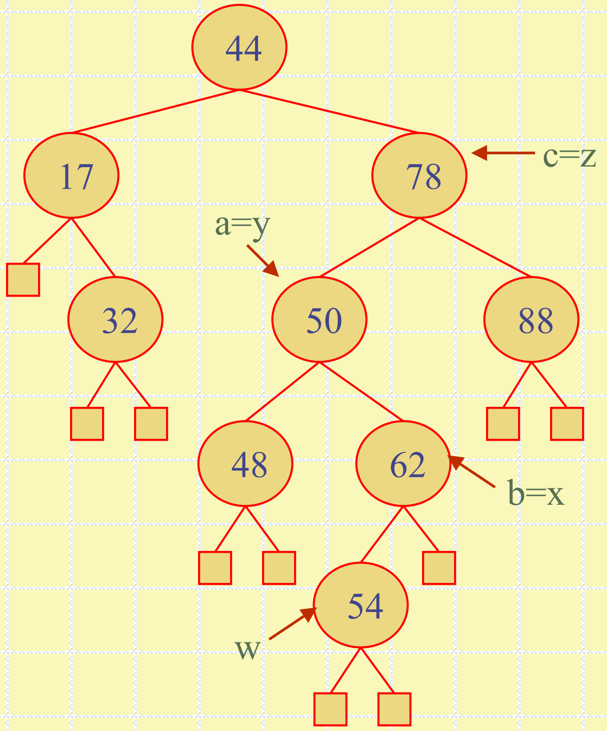
- This implies that the height of an AVL tree storing n entries can, at most, be $2 \log n + 2$.
- Thus the height of an AVL tree is $O(\log n)$.
- Consequently, a map implemented with an AVL tree runs in $O(\log n)$.

Insertion

- Insertion is as in a binary search tree (Always done by expanding an external node), but additional computations are needed afterwards.
- Example: put(54)



before insertion



after insertion

Insertion

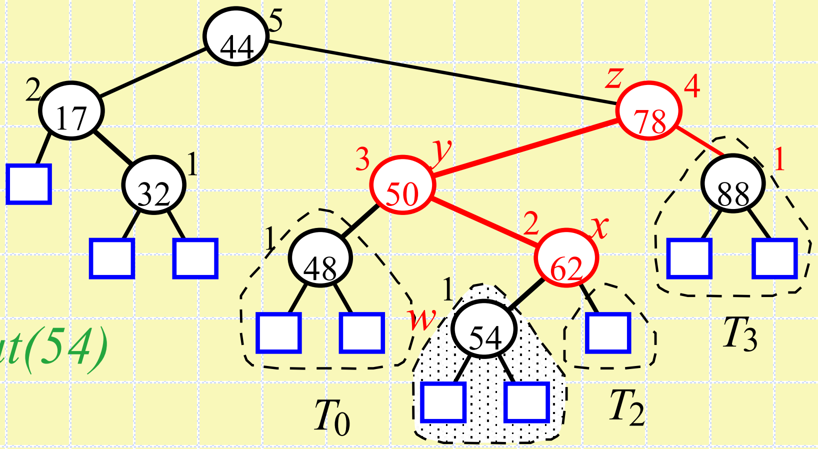
- Insertion adds the new node at a previously external node then creates two external nodes to this new added node.
- This would hence increases the height of that part of the tree which may violates the height-balance property, resulting in the tree being *unbalanced*.
- Consequently, the tree may have to be "**restructured**" to restore the height-balance.
- A *balanced* tree would have the difference (absolute value) between the heights of its left and right children being at most 1.
- NOTE: If a tree is balanced, then each of its internal nodes (subtrees at these nodes) must be balanced as well.

after insertion

Trinode Restructuring

- If the tree becomes unbalanced after the addition, then the nodes in the path from this new node to the root have to be the only involved nodes where the unbalance has occurred.

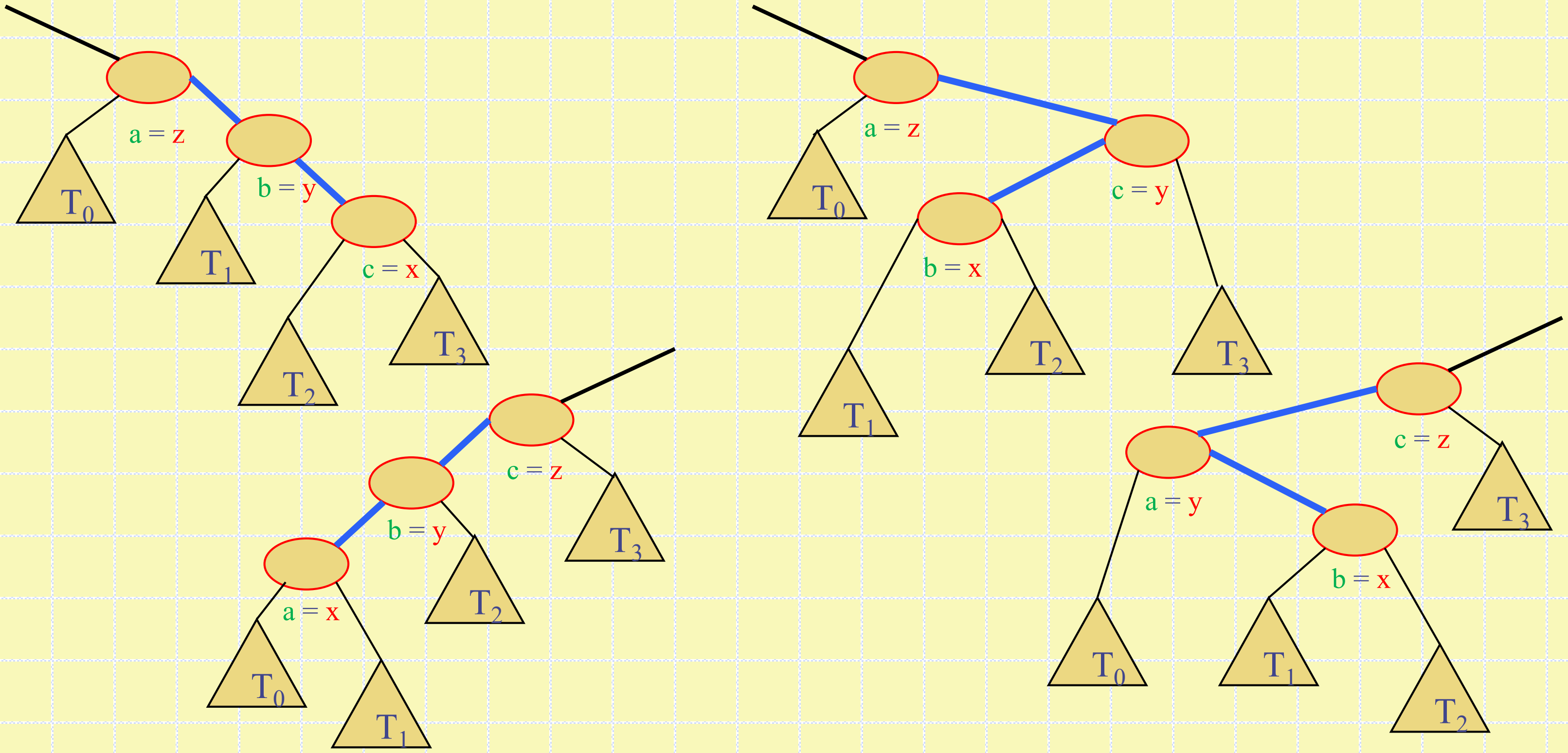
- Let w be the new added node causing the unbalance to occur.
- Let z be the first encountered node from w up towards the root when the unbalance started to occur.
- Let y be the child of z with the higher height.
- Finally, let x be the child of y with the higher height.



Unbalanced, after *put(54)*

Trinode Restructuring

- We now need to rebalance the tree rooted at z. The rebalancing method is referred to as *trinode restructuring*.
- Trinode restructuring starts by temporarily renaming x, y and z to a, b and c, where a precedes b and b precedes c in an **inorder traversal** (see illustrative example).
- There are actually 4 possible ways for mapping x, y and z to a, b and c.



Trinode Restructuring

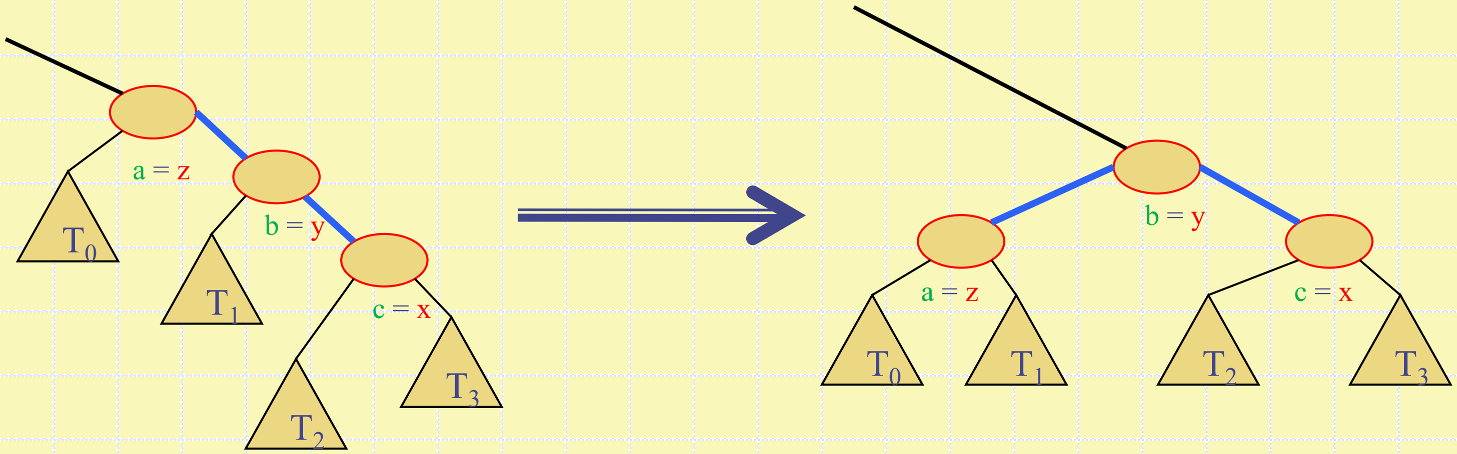
- Trinode restructuring rebalances the tree as follows:

  1) Replace *z* with the node called *b* (that is, direct the edge heading to z from its parent to *b* instead of *z*).

  2) Take *a* and *c* and make them children of *b*.

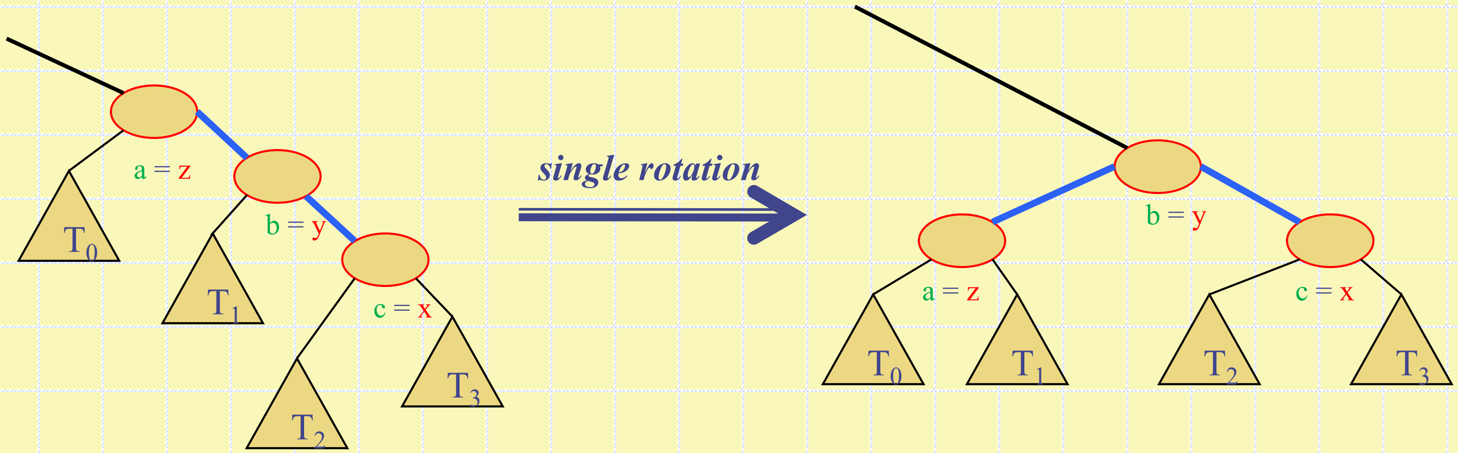
  3) Take the 4 subtrees (T_0, T_1, T_2, and T_3, which were the previous children of *x*, *y* and *z*) and make them children of *a* and *c*.
     - Maintain the inorder relationship between these 4 subtrees when placing them as the children of *a* and *c*.

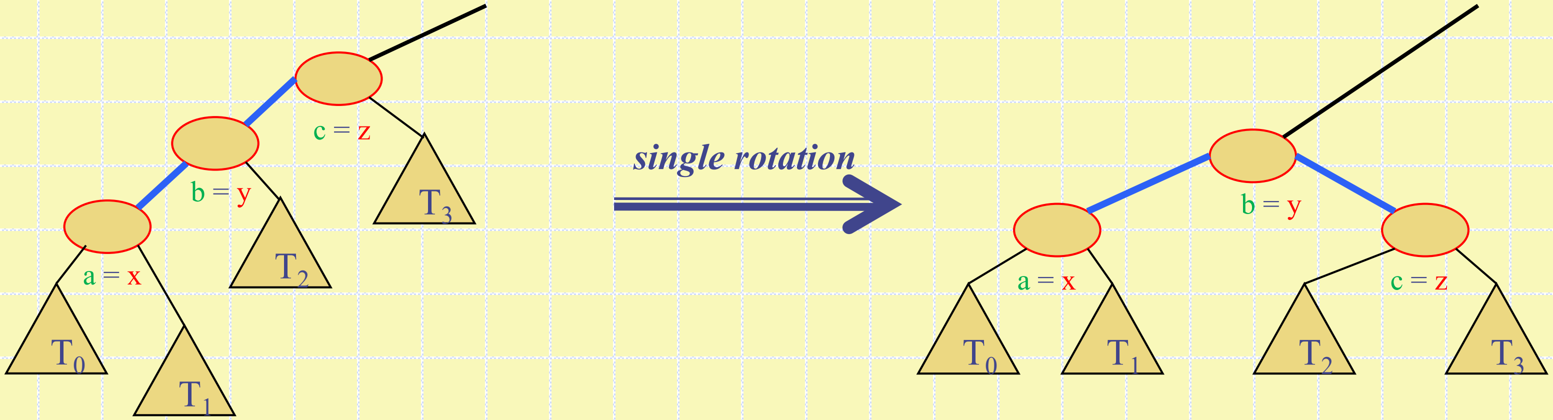
  ➔ The trinode restructuring adjustments ***locally*** the balance of the originally unbalanced subtree at the older node *z*, which actually results in the entire tree, ***globally***, becoming balanced again.



Trinode Restructuring

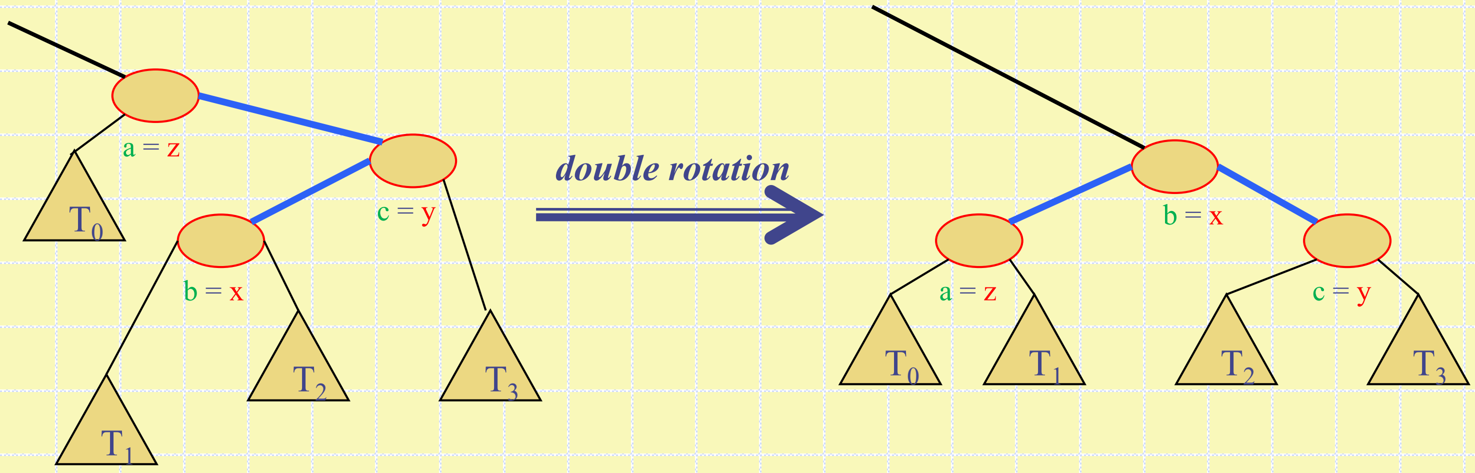
- The restructuring of the first two cases are referred to as ***single rotation*** (since geometrically can be visualized as rotating y over z.

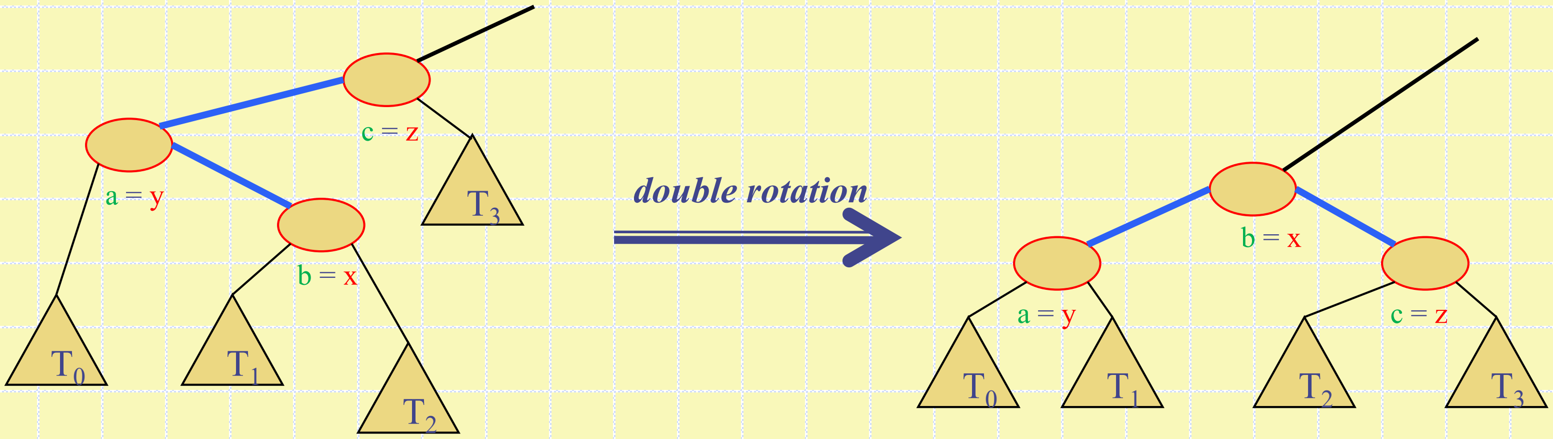




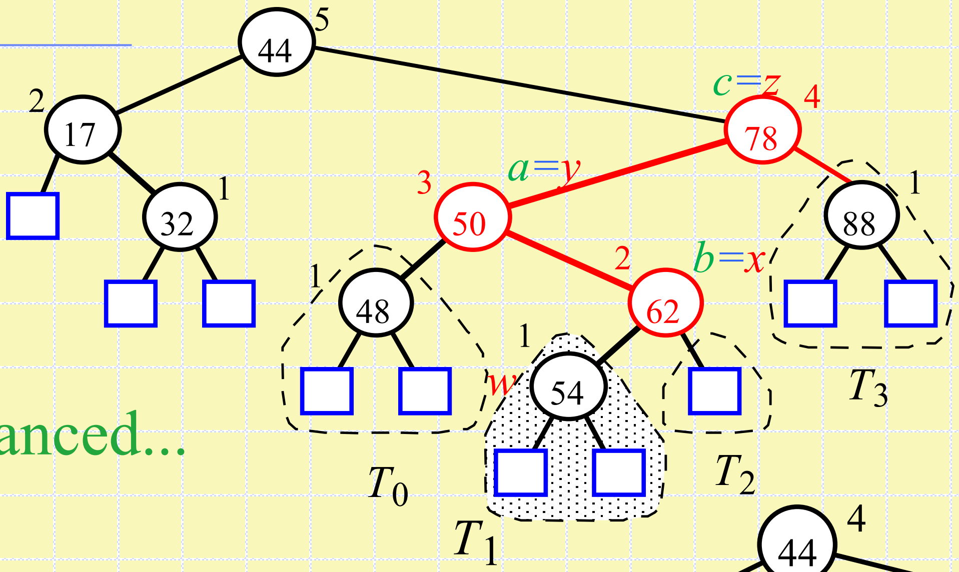
Trinode Restructuring

- The restructuring of the other two cases are referred to as ***double rotation*** (since geometrically can be visualized as rotating x over y then rotating x over z.

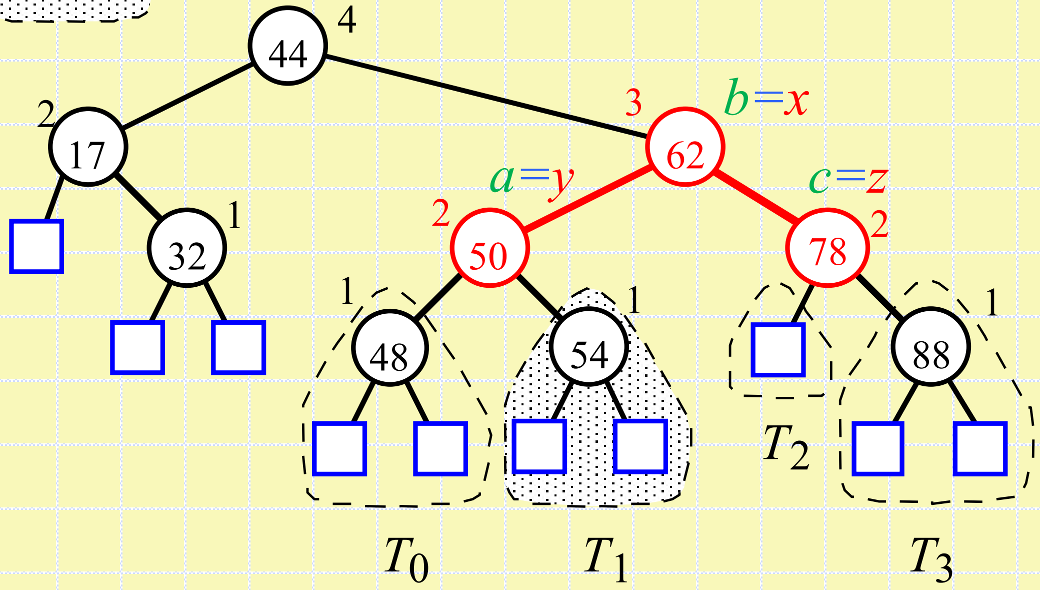




Insertion Example



unbalanced...

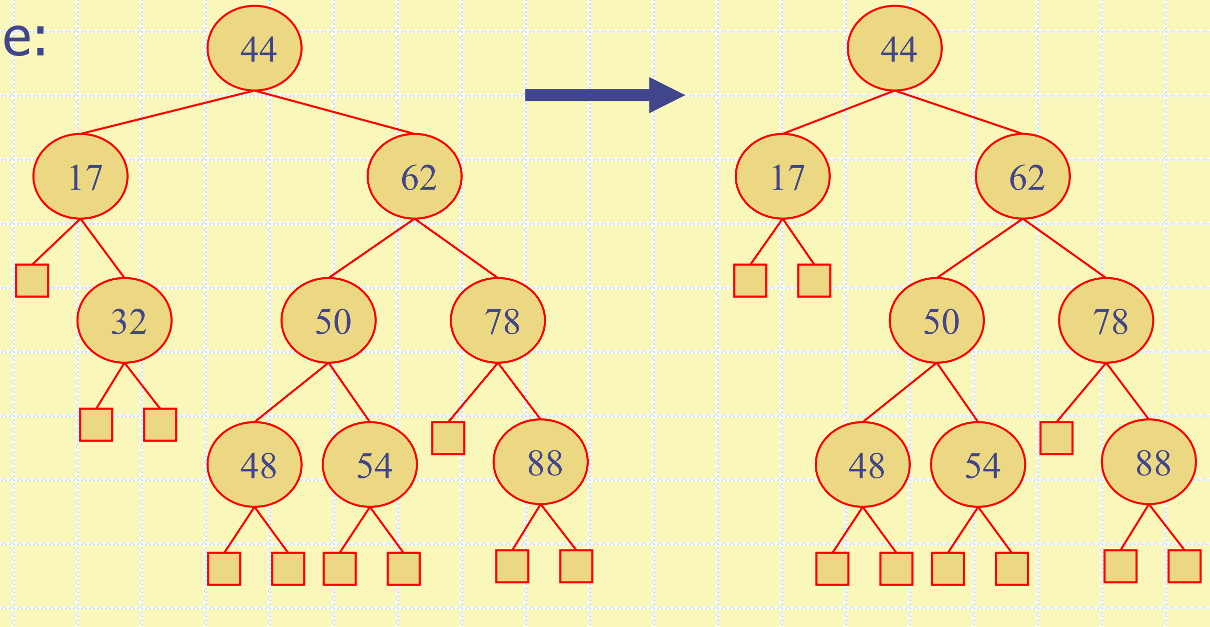


...balanced

Removal

- Removal begins as in a binary search tree, which means that the removed node will become an empty external node. Consequently, however, its parent, *w*, may cause an imbalance.

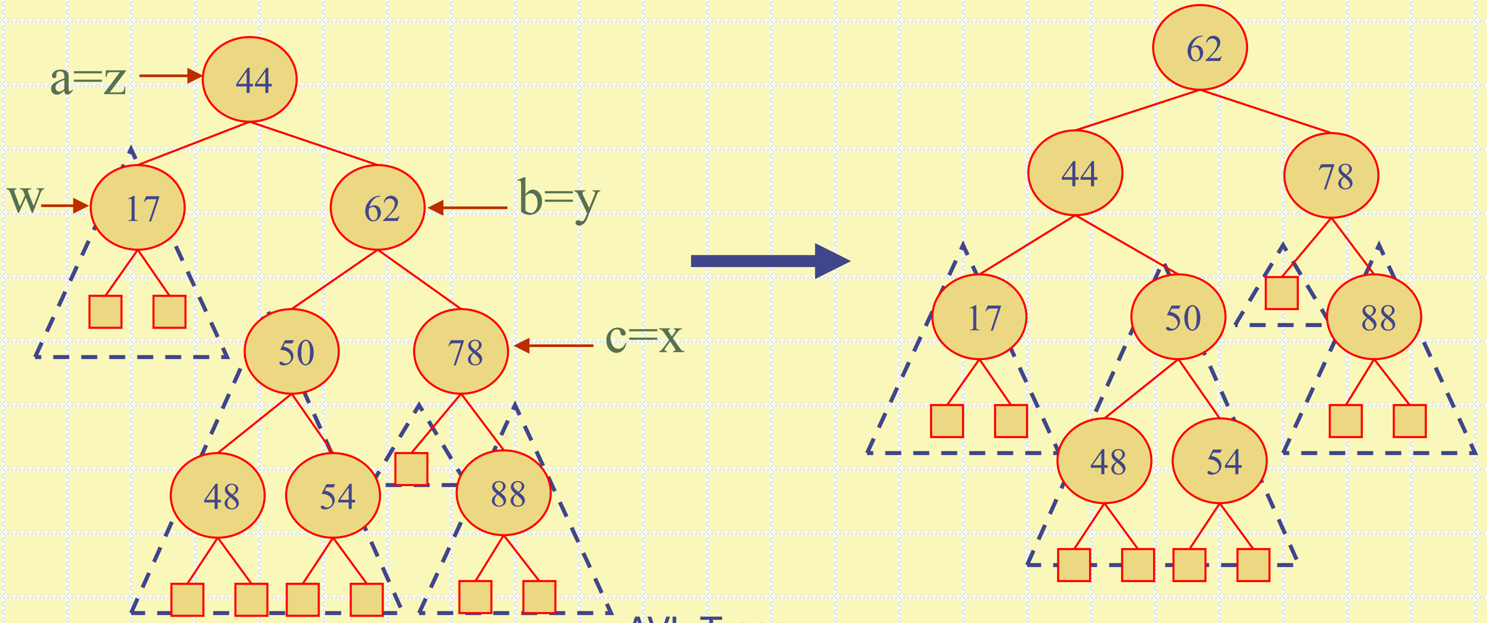
- Example:



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



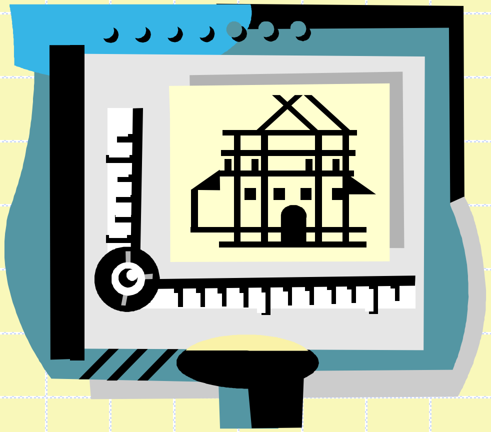
Rebalancing after a Removal

- Unfortunately, this restructuring may upset the balance of another node higher in the tree. That is, ***locally*** adjusting the tree does not guarantee the tree being adjusted ***globally***.

- For instance, assume a node has height 6 with its children having heights of 5 and 4, and the one with height 4 has children with heights of 3 and 2. If the removal is taken from the child with height 2, then that tree height becomes 1, which is unbalanced with its sibling (having 3). The restructuring will turn these two children to height 2, which is balanced, but this will consequently change the height of their parent from 4 to 3, causing further unbalance between this node (now having height 3) and its sibling, which has height 5.

- Thus, we must continue checking for balance until the root of *T* is reached, and readjust along the way if needed (which may result in $\log n$ constructions in the worst case).

AVL Tree Performance



- a single restructure takes O(1) time
  - using a linked-structure binary tree
- get 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(1)
  - However, we still need O(log n) to find out if restructuring is needed
- remove takes O(log n) time
  - initial find is O(log n)
  - Restructuring up the tree, maintaining heights is O(log n)