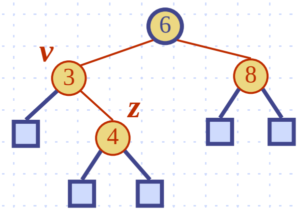
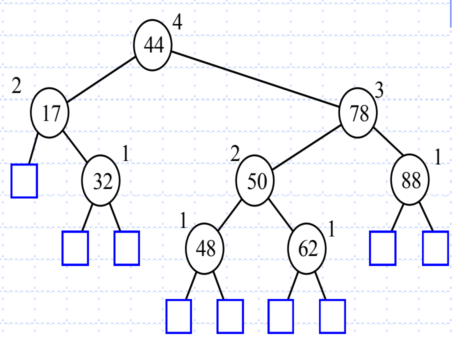
Lecture 13: AVL Trees



AVL Tree Definition

- Height of the node **v** in tree is the length of a longest path from **v** to an external node
- 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.*

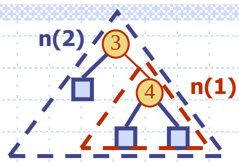


An example of an AVL tree where the heights are shown next to the nodes:

Reminder

- **Depth** of node v is the number of ancestors of v excluding the v itself
- **Height** of the tree T is equal to the maximum depth of an external node v.

Height of an AVL Tree



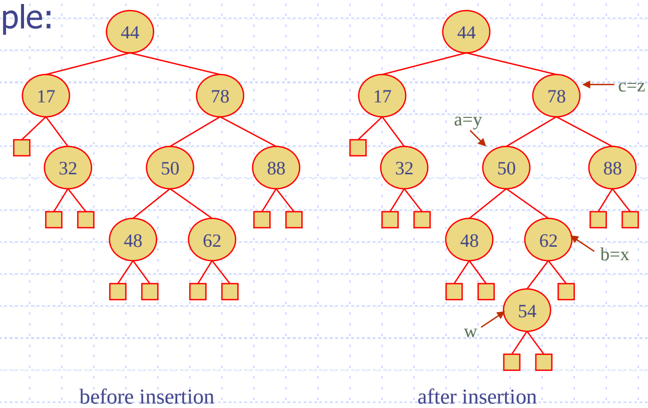
- **Fact**: The ***height*** of an AVL tree storing n keys is O(log n).
- **Proof**: Let us bound **n(h):** the minimum number of internal nodes of an AVL tree of height h.
- We easily see that n(1) = 1 and n(2) = 2
- For n > 2, an AVL tree of height h contains the root node, one AVL subtree of height h-1 and another of height h-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^in(h-2i), where h-2i≥1
- 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 storing n keys is O(log n)

Definitions:

- Node v of a tree T is **balanced** if the absolute value of the difference between the heights of the children of v is at most 1
- Node is **unbalanced** otherwise

Insertion in an AVL Tree

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



Restructuring

Algorithm ***restructure***(x)

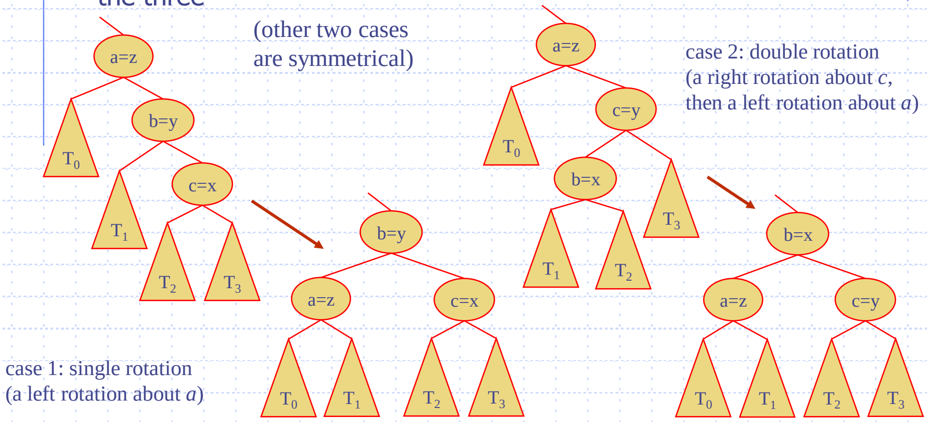
Input node ***x*** of a binary search tree T that has both a parent ***y*** and a grandparent ***z***

Output tree ***T*** after a trinode restructuring involving nodes ***x, y, z***

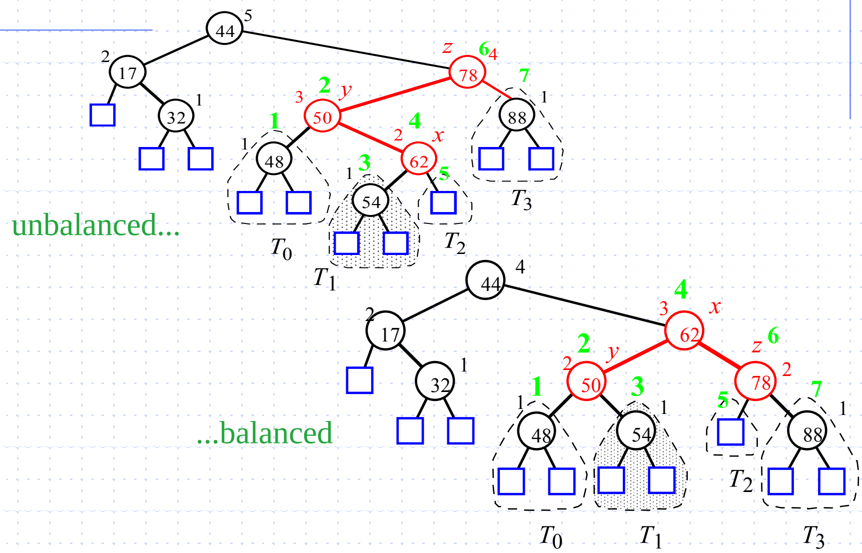
1) Let (***a,b,c***) be left to right (inorder) listing of the nodes x, y, and z. and let (T_0, T_1, T_2, T_3) be a left-to-right (inorder) listing of the four subtrees of x, y, and z not rooted at ***x, y,*** and ***z***.
2) Replace the subtree rooted at ***z*** with a new subtree rooted at ***b***
3) Let ***a*** be the left child of ***b*** and let T_0 and T_1 be the left and right subtress of a respectively.
4) Let ***c*** be the right child of **b** and let T_2 and T_3 be the left and right subtress of ***c*** respectively.

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

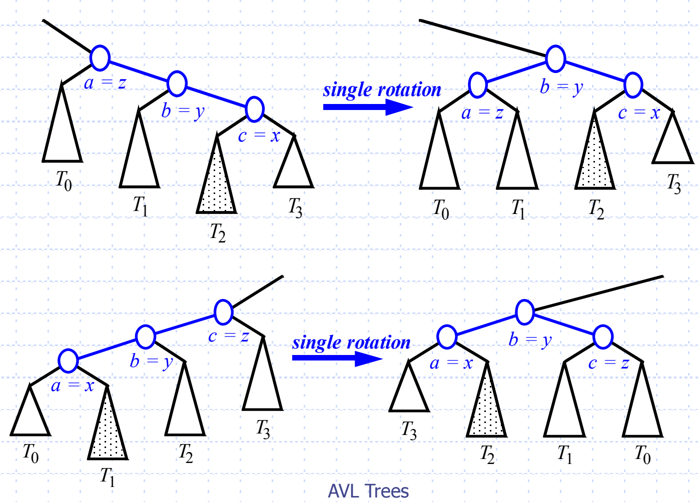


Insertion Example, continued



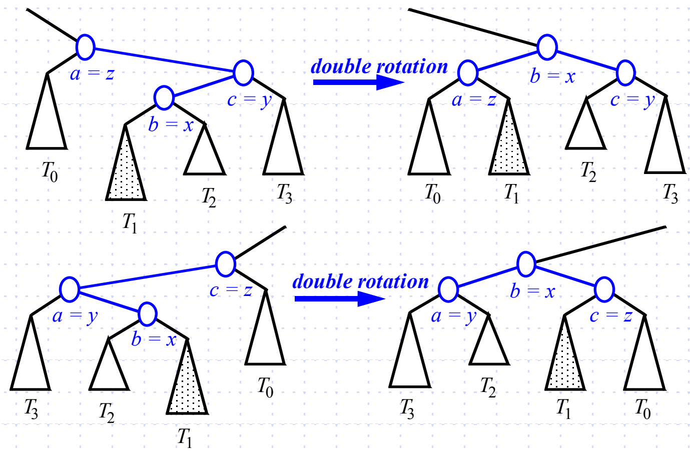
Restructuring (as Single Rotations)

- Single Rotations:



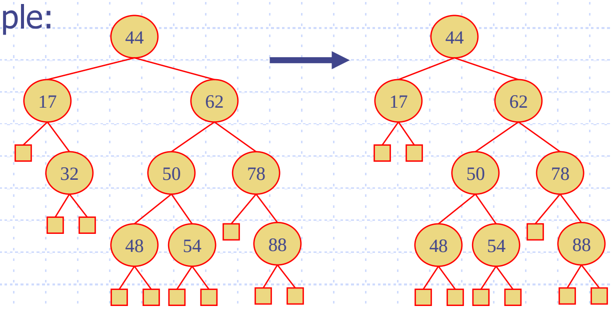
Restructuring (as Double Rotations)

- double rotations:



Removal in an AVL Tree

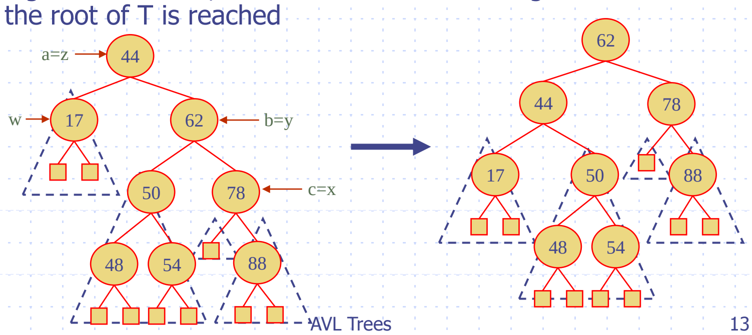
- Removal begins as in a binary search tree, which means the node removed will become an empty external node. Its parent, w, may cause an imbalance.
- Example:



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.
- 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



Running Times for AVL Trees

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