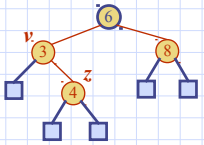
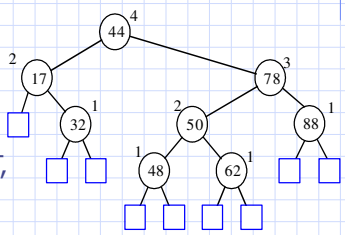
AVL Trees



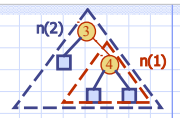
AVL Tree Definition

- 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:

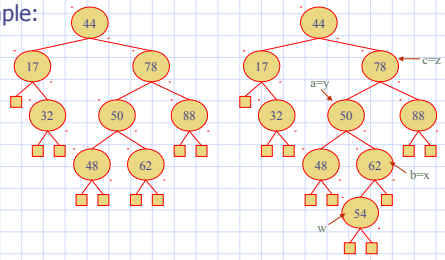
Height of an AVL Tree



- 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 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(h-6), … (by induction), n(h) > 2in(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

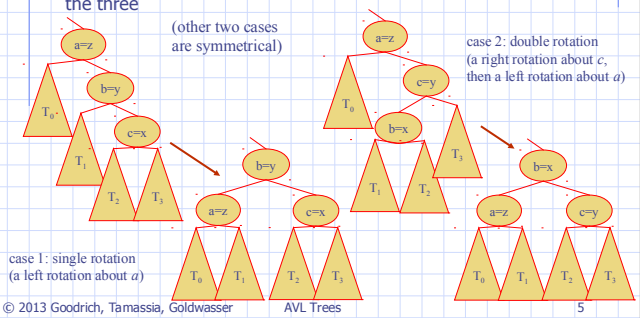
- Insertion is as in a binary search tree
- Always done by expanding an external node.
- Example:



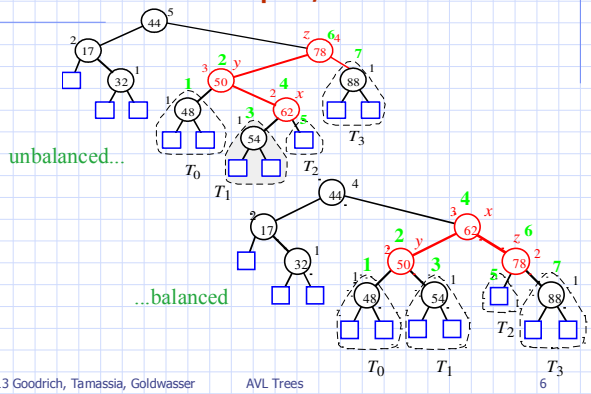
before insertion after insertion

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

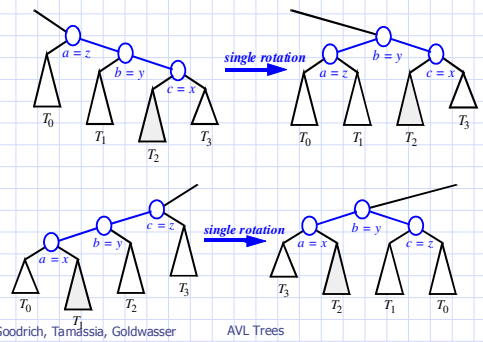


Insertion Example, continued



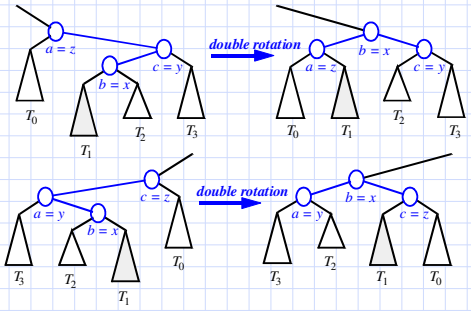
Restructuring (as Single Rotations)

- Single Rotations:



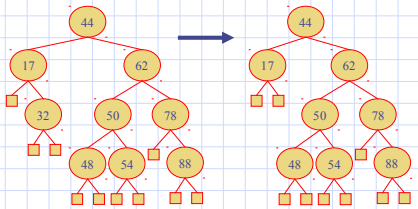
Restructuring (as Double Rotations)

- double rotations:



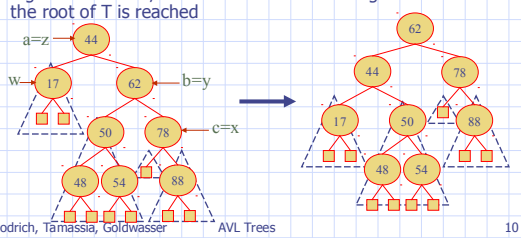
Removal

- 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:

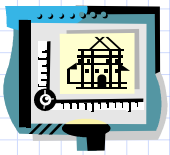


Rebalancing after a Removal

- 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



AVL Tree Performance



- a single restructure 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)

Python Implementation

```
1  class AVLTreeMap(TreeMap):
2    """Sorted map implementation using an AVL tree."""
3
4    #------------------------ nested _Node class ------------------------
5    class _Node(TreeMap._Node):
6      """Node class for AVL maintains height value for balancing."""
7      __slots__ = '_height'          # additional data member to store height
8
9      def __init__(self, element, parent=None, left=None, right=None):
10       super().__init__(element, parent, left, right)
11       self._height = 0               # will be recomputed during balancing
12
13     def _left_height(self):
14       return self._left._height if self._left is not None else 0
15
16     def _right_height(self):
17       return self._right._height if self._right is not None else 0
```

Python Implementation, Part 2

```
18   #------------------------ positional-based utility methods ------------------------
19   def _recompute_height(self, p):
20     p._node._height = 1 + max(p._node.left._height(), p._node.right._height())
21
22   def _isbalanced(self, p):
23     return abs(p._node.left._height( ) - p._node.right._height()) <= 1
24
25   def _tall_child(self, p, favorleft=False): # parameter controls tiebreaker
26     if p._node.left._height( ) + (1 if favorleft else 0) > p._node.right._height():
27       return self.left(p)
28     else:
29       return self.right(p)
30
31   def _tall_grandchild(self, p):
32     child = self._tall_child(p)
33     # if child is on left, favor left grandchild; else favor right grandchild
34     alignment = (child == self.left(p))
35     return self._tall_child(child, alignment)
36
```


Python Implementation, end

```
37   def _rebalance(self, p):
38     while p is not None:
39       old_height = p._node._height          # trivially 0 if new node
40       if not self._isbalanced(p):           # imbalance detected!
41         # perform trinode restructuring, setting p to resulting root,
42         # and recompute new local heights after the restructuring
43         p = self._restructure(self._tall_grandchild(p))
44         self._recompute_height(self.left(p))
45         self._recompute_height(self.right(p))
46       self._recompute_height(p)             # adjust for recent changes
47       if p._node._height == old_height:     # has height changed?
48         p = None                            # no further changes needed
49       else:
50         p = self.parent(p)                  # repeat with parent
51
52   #------------------------ override balancing hooks ------------------------
53   def _rebalance_insert(self, p):
54     self._rebalance(p)
55
56   def _rebalance_delete(self, p):
57     self._rebalance(p)
```