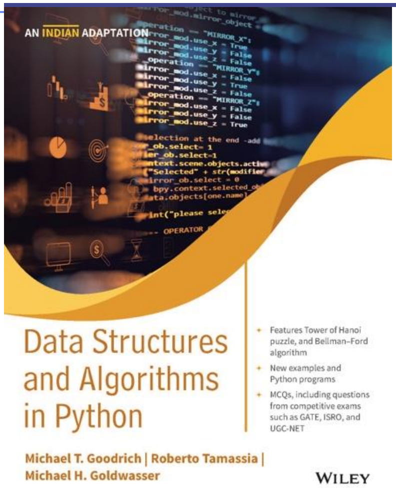


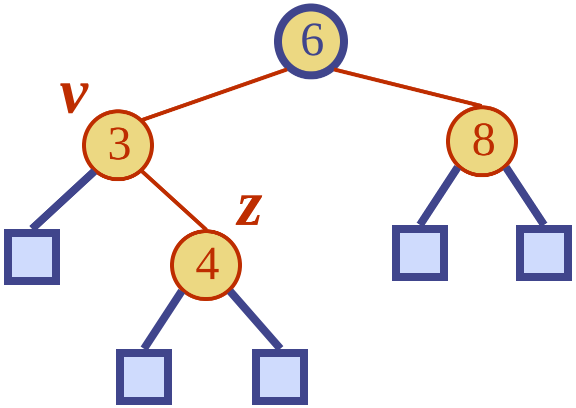
Data Structures and Algorithms in Python

Michael T. Goodrich/ Roberto Tamassia/ Michael H. Goldwasser

Adapted by Subhrakanta Panda

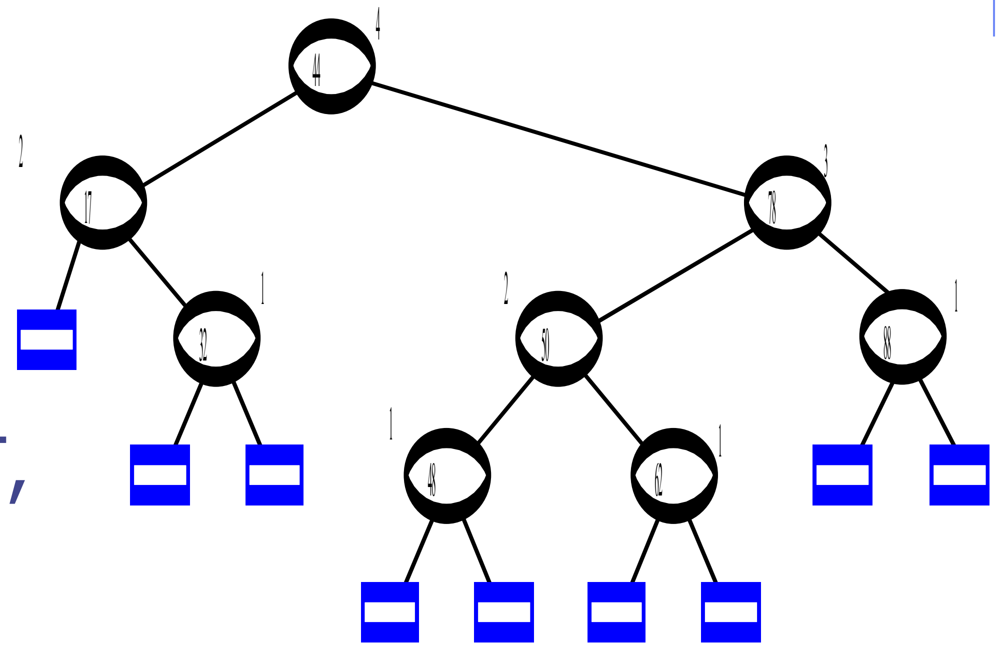
Chapter 12
Search Trees

AVL Trees



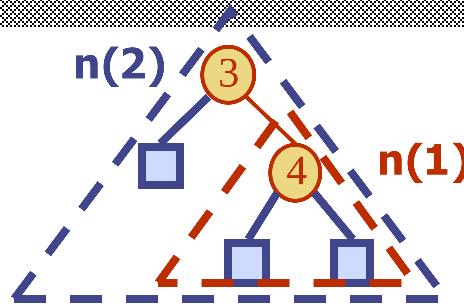
AVL Tree Definition

- 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



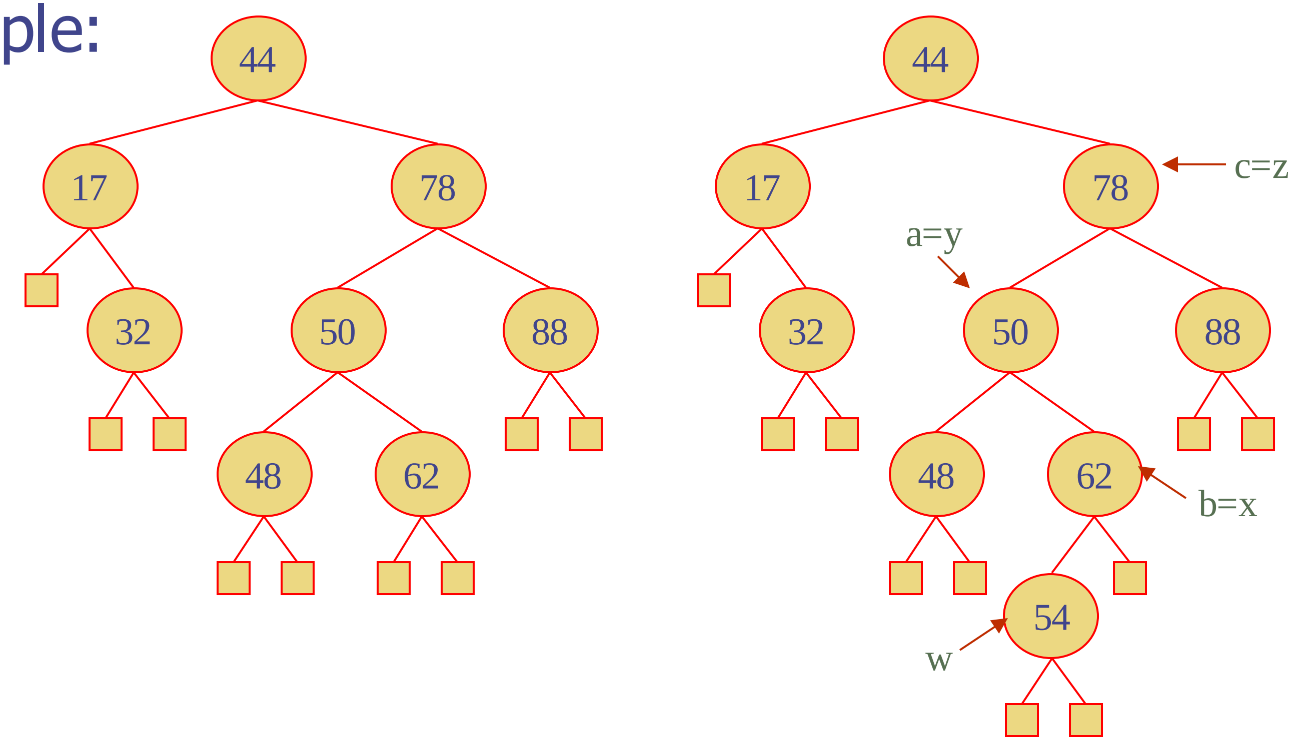
- Fact: The height of an AVL tree storing n keys is O(log n).
- Proof: Let us bound n(h): the minimum number of internal nodes of an AVL tree of height h.
- We easily see that n(1) = 1 and n(2) = 2
- For n > 2, an AVL tree of height h contains the root node, one AVL subtree of height h-1 and another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

  $n(h) > 2^i n(h-2i)$
- 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:

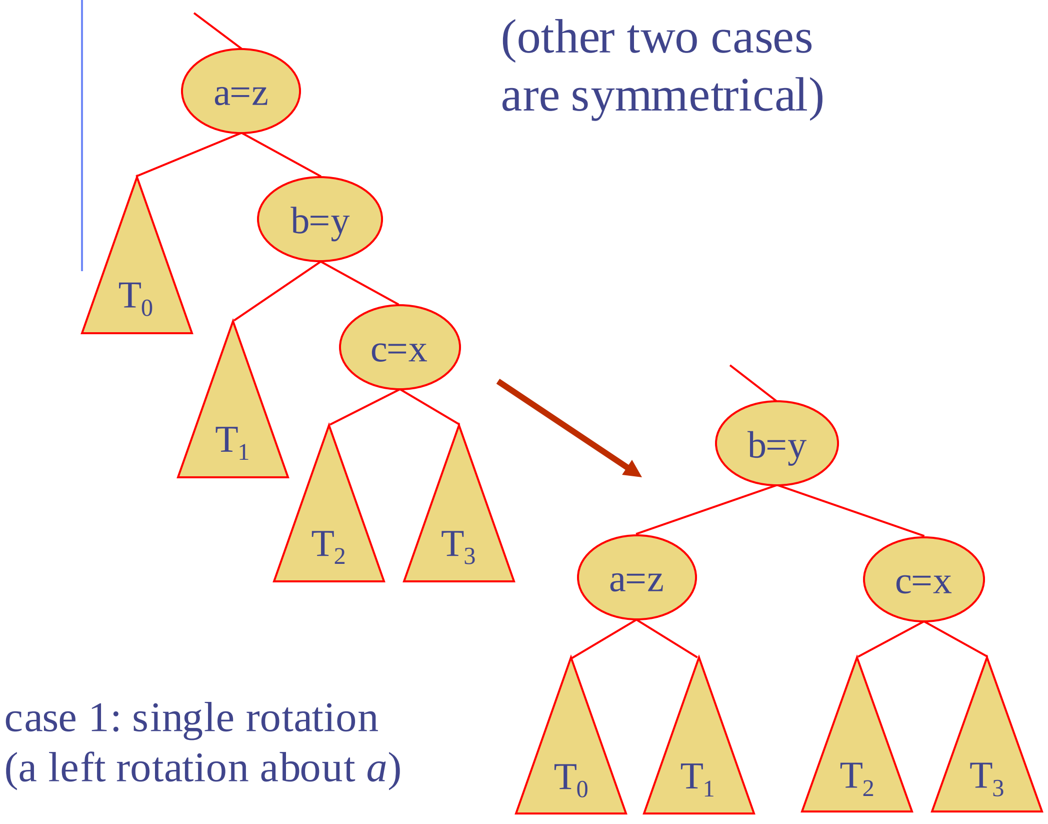


before insertion

after insertion

Trinode Restructuring

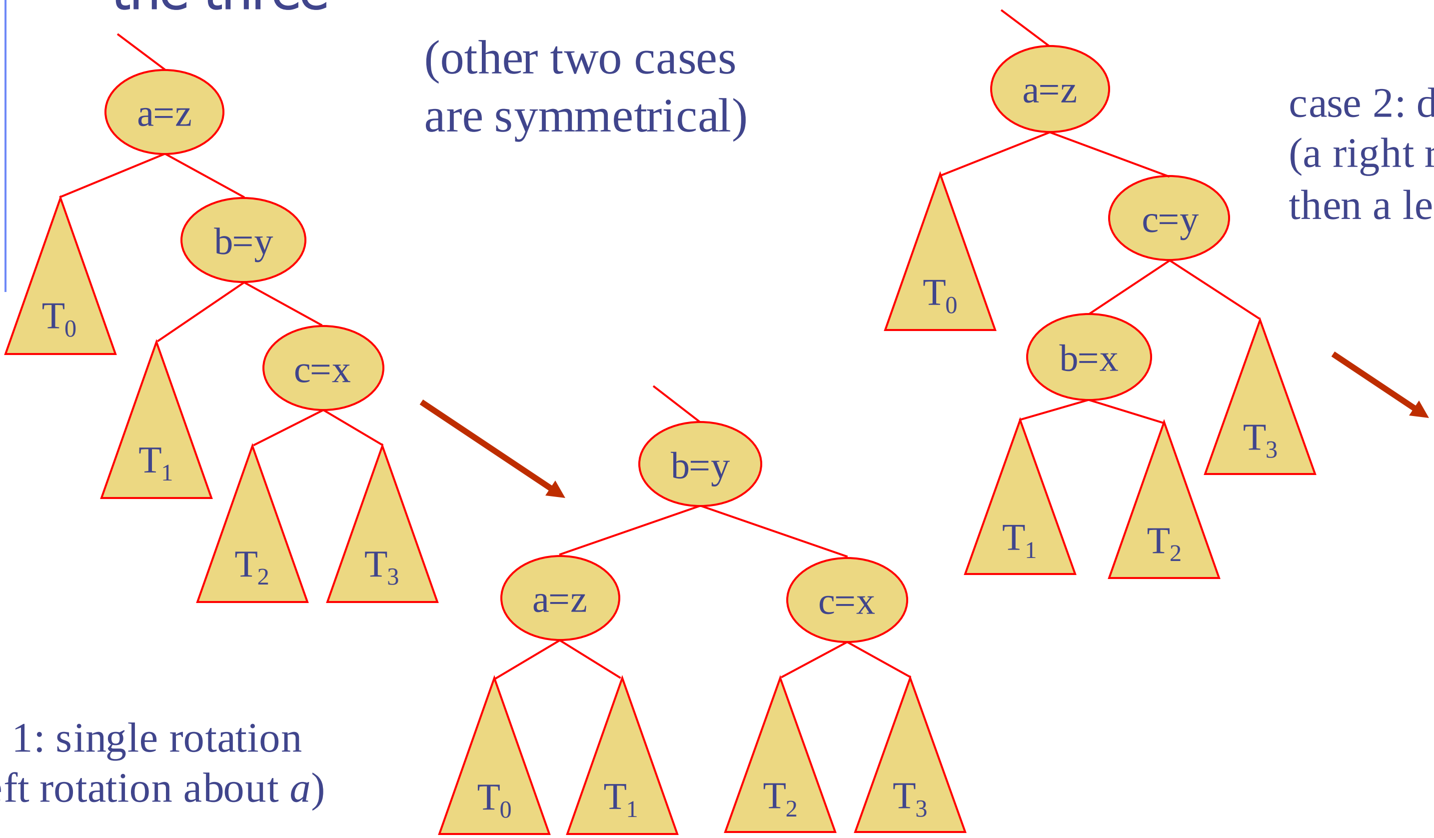
- let (a,b,c) be an inorder listing of x, y, z
- perform the rotations needed to make b the topmost node of the three



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

(other two cases are symmetrical)



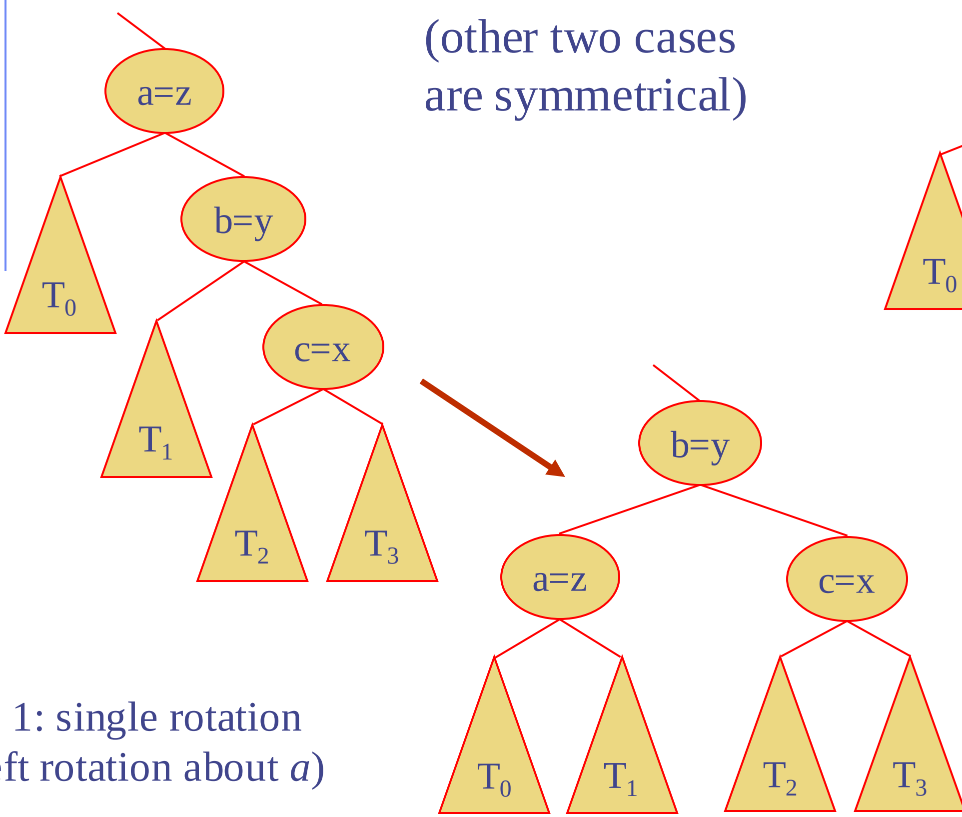
case 2: double rotation (a right rotation about *c*, then a left rotation about *a*)

case 1: single rotation (a left rotation about *a*)

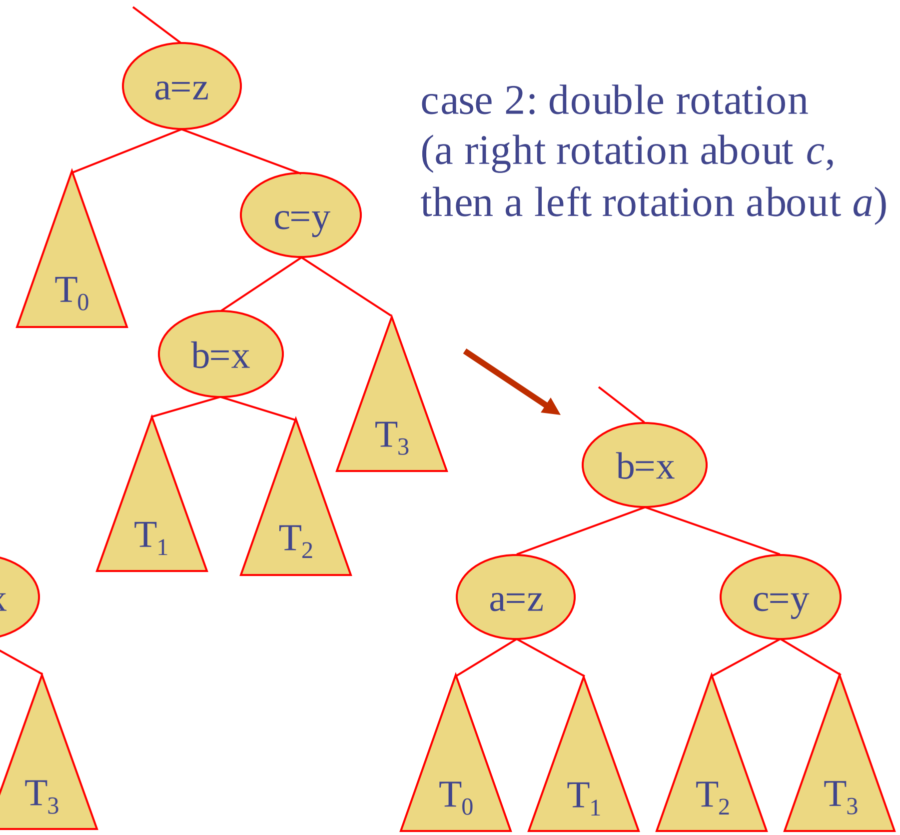
Trinode Restructuring

- 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

(other two cases are symmetrical)

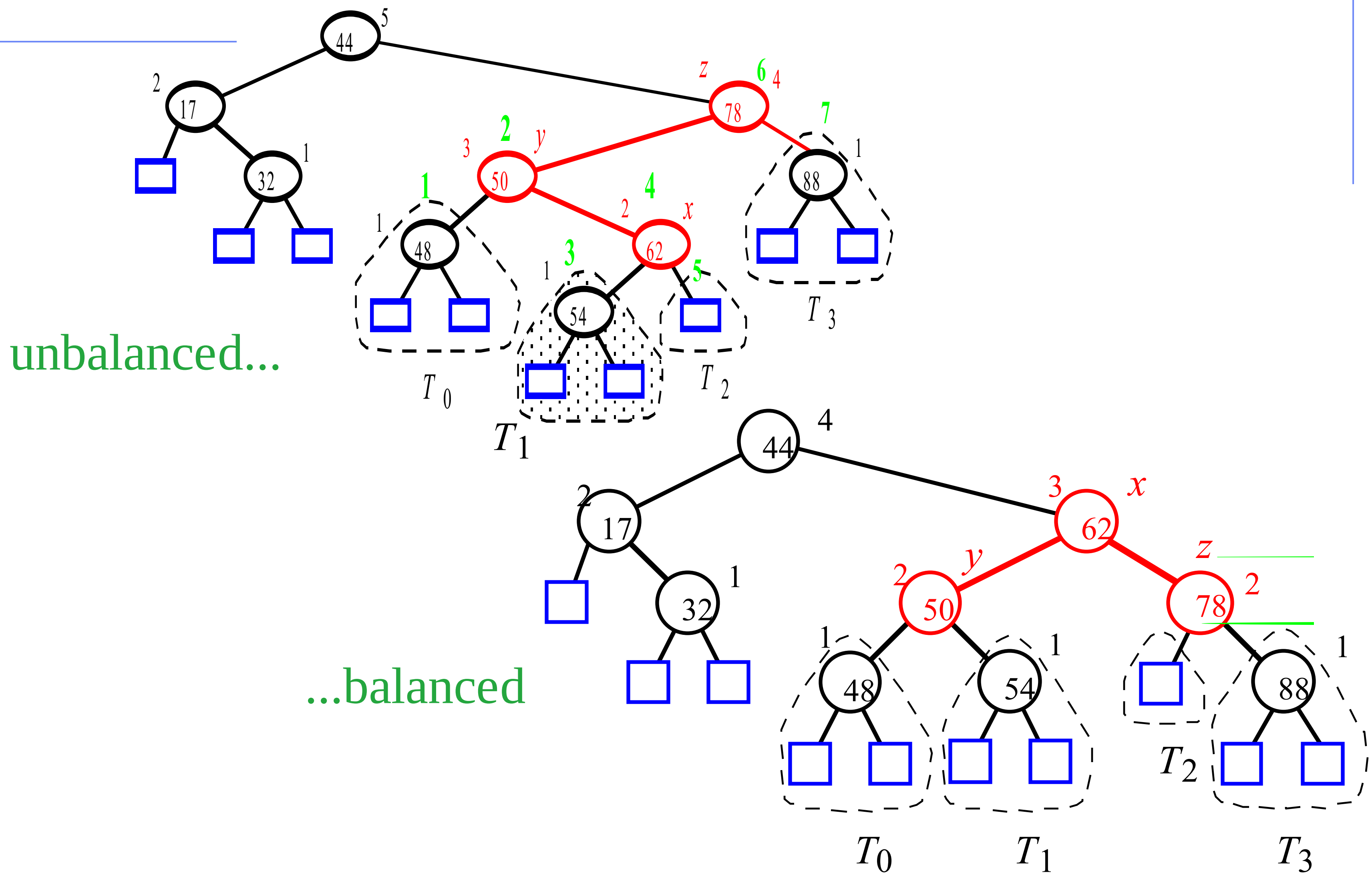


case 1: single rotation
(a left rotation about a)



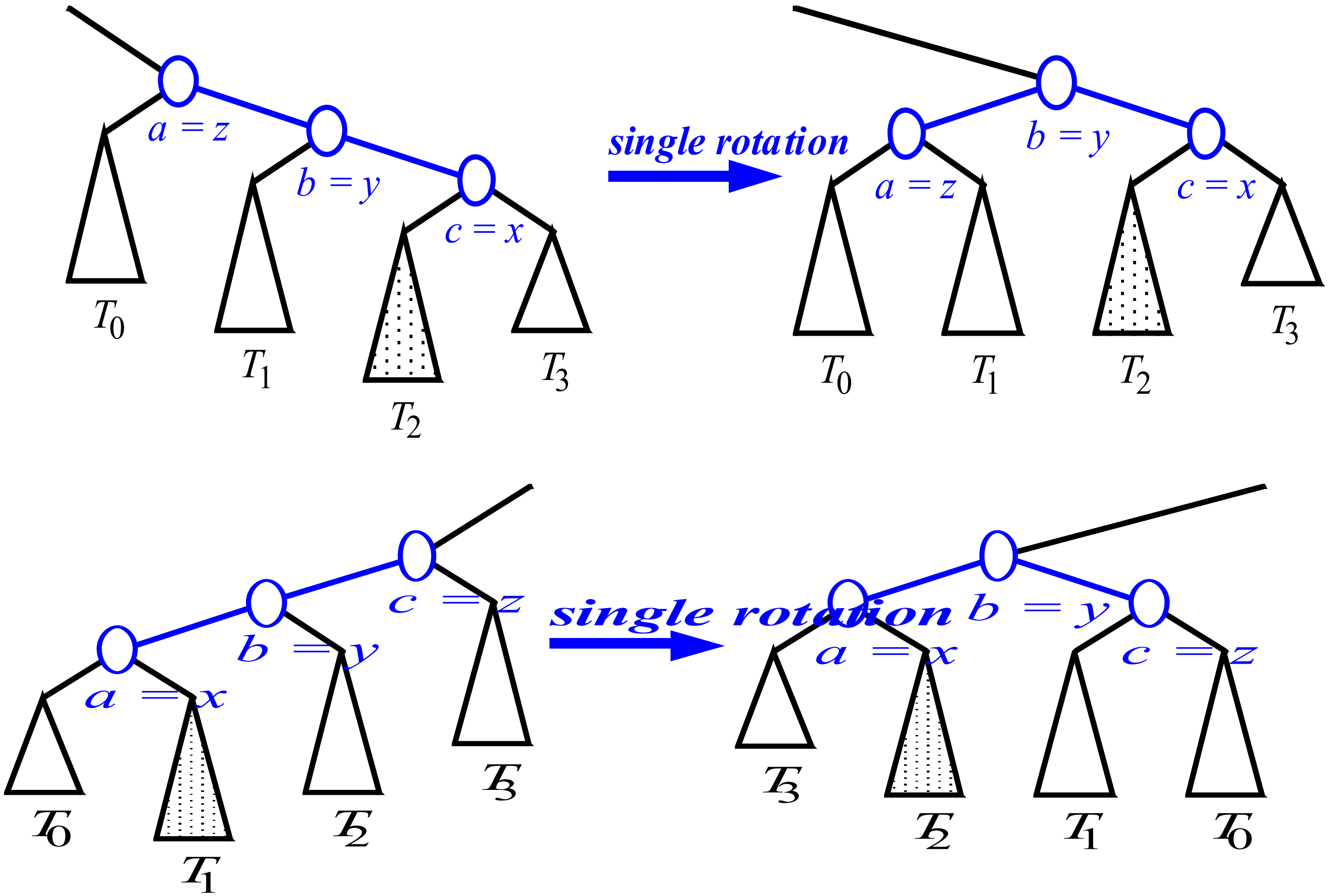
case 2: double rotation
(a right rotation about c, then a left rotation about a)

Insertion Example, continued



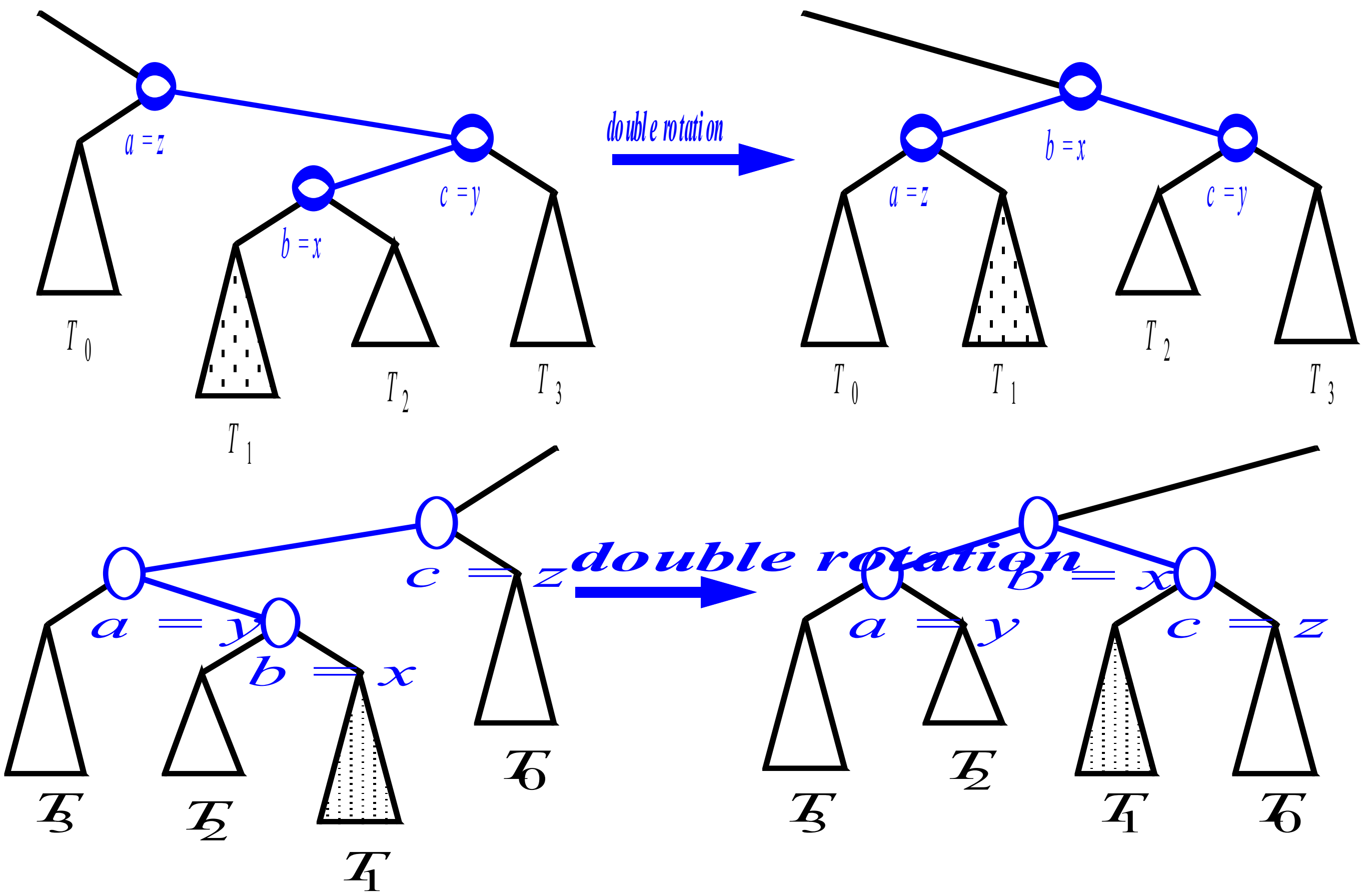
Restructuring (as Single Rotations)

- Single Rotations:



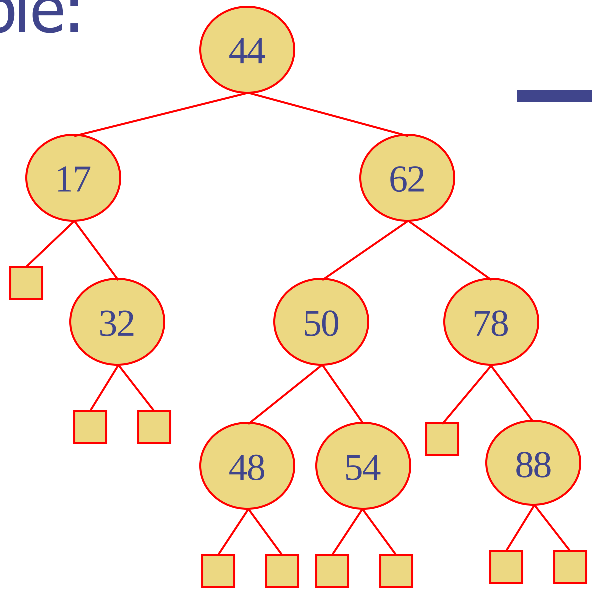
Restructuring (as Double Rotations)

- double rotations:

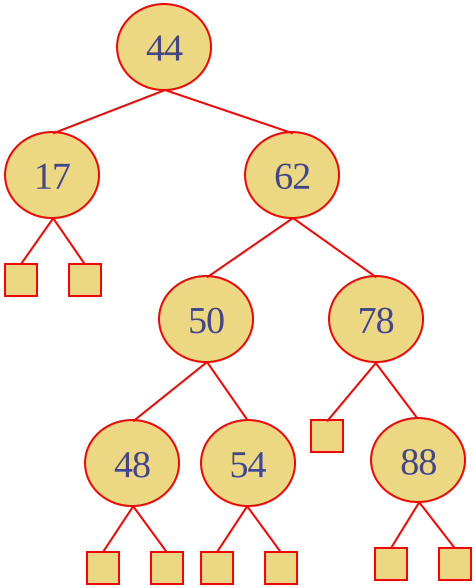


Removal

- 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:



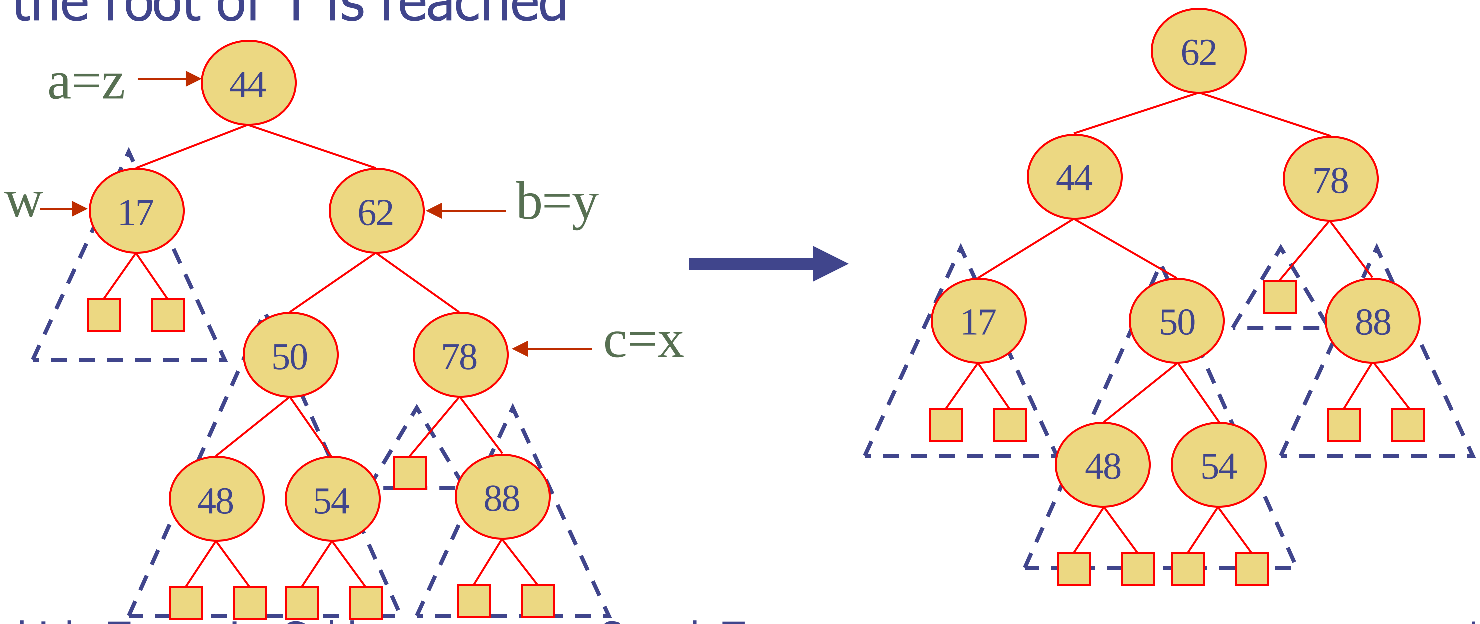
before deletion of 32



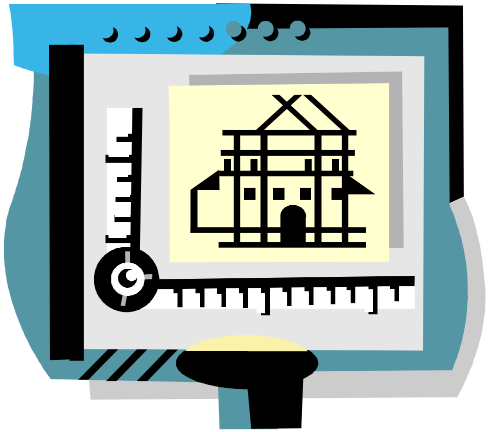
after deletion

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



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

```
class AVLTreeMap(TreeMap):
  """Sorted map implementation using an AVL tree."""

  #-------------------------- nested _Node class --------------------------
  class _Node(TreeMap._Node):
    """Node class for AVL maintains height value for balancing."""
    __slots__ = '_height'         # additional data member to store height

    def __init__(self, element, parent=None, left=None, right=None):
      super().__init__(element, parent, left, right)
      self._height = 0            # will be recomputed during balancing

    def left_height(self):
      return self._left._height if self._left is not None else 0

    def right_height(self):
      return self._right._height if self._right is not None else 0
```

Python Implementation, Part 2

```
#------------------------ positional-based utility methods ------------------------
def _recompute_height(self, p):
  p._node._height = 1 + max(p._node.left_height(), p._node.right_height())

def _isbalanced(self, p):
  return abs(p._node.left_height( ) - p._node.right_height()) <= 1

def _tall_child(self, p, favorleft=False): # parameter controls tiebreaker
  if p._node.left_height( ) + (1 if favorleft else 0) > p._node.right_height():
    return self.left(p)
  else:
    return self.right(p)

def _tall_grandchild(self, p):
  child = self._tall_child(p)
  # if child is on left, favor left grandchild; else favor right grandchild
  alignment = (child == self.left(p))
  return self._tall_child(child, alignment)
```

Python Implementation, end

```
def _rebalance(self, p):
  while p is not None:
    old_height = p._node._height             # trivially 0 if new node
    if not self._isbalanced(p):              # imbalance detected!
      # perform trinode restructuring, setting p to resulting root,
      # and recompute new local heights after the restructuring
      p = self._restructure(self._tall_grandchild(p))
      self._recompute_height(self.left(p))
      self._recompute_height(self.right(p))
    self._recompute_height(p)                # adjust for recent changes
    if p._node._height == old_height:        # has height changed?
      p = None                               # no further changes needed
    else:
      p = self.parent(p)                     # repeat with parent

#------------------------- override balancing hooks -------------------------
def _rebalance_insert(self, p):
  self._rebalance(p)

def _rebalance_delete(self, p):
  self._rebalance(p)
```