SE274 Data Structure

Lecture 8: Search Trees – Part 2

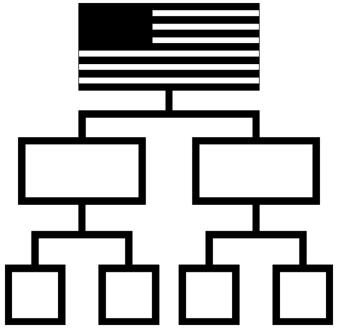
(textbook: Chapter 11)

Apr 22, 2020

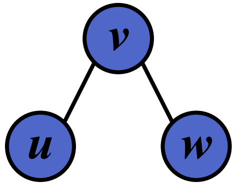
Instructor: Sunjun Kim

Information&Communication Engineering, DGIST

(Recap) Binary Search Trees

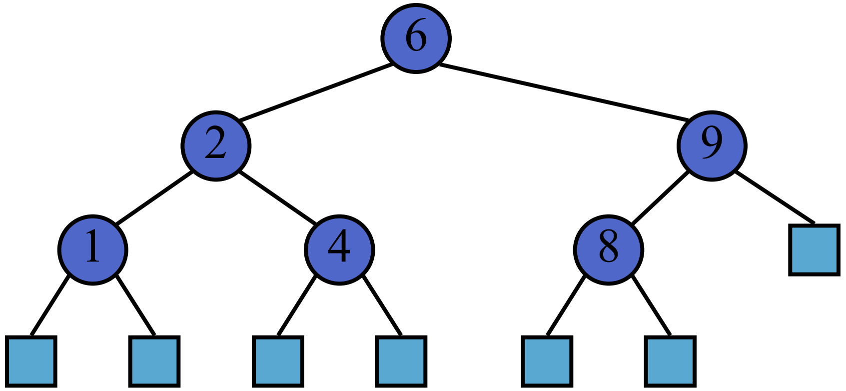


- A binary search tree is a binary tree storing keys (or key-value items) at its nodes and satisfying the following property:
  - Let $\boldsymbol{u}$, $\boldsymbol{v}$, and $\boldsymbol{w}$ be three nodes such that $\boldsymbol{u}$ is in the left subtree of $\boldsymbol{v}$ and $\boldsymbol{w}$ is in the right subtree of $\boldsymbol{v}$. We have $\boldsymbol{key}(\boldsymbol{u}) \leq \boldsymbol{key}(\boldsymbol{v}) \leq \boldsymbol{key}(\boldsymbol{w})$



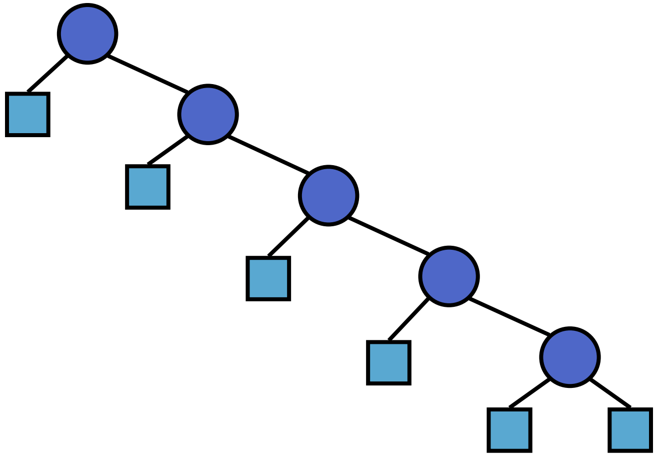
- External nodes do not store items, instead we consider them as None

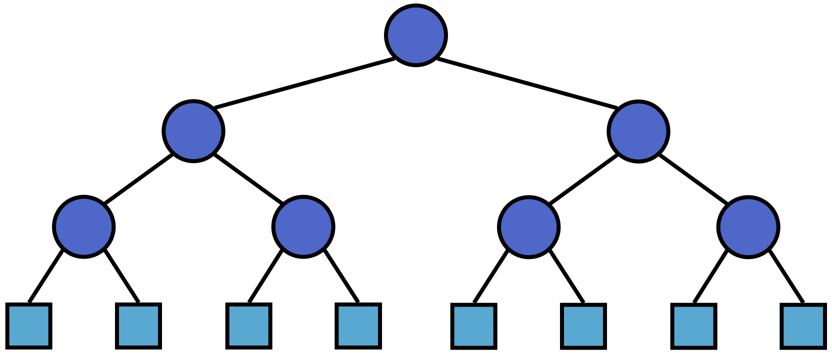
- An inorder traversal of a binary search trees visits the keys in increasing order



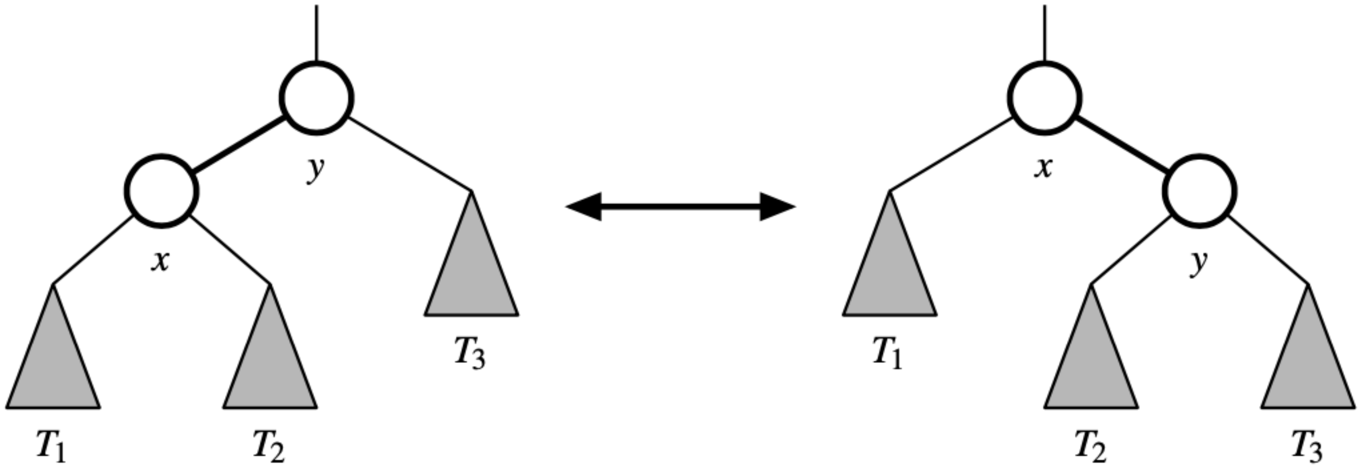
(Recap) Binary Search Tree Performance

- Consider an ordered map with n items implemented by means of a binary search tree of height h
  - the space used is $O(n)$
  - Search and update methods take $O(h)$ time
- The height h is $O(n)$ in the worst case and $O(\log n)$ in the best case



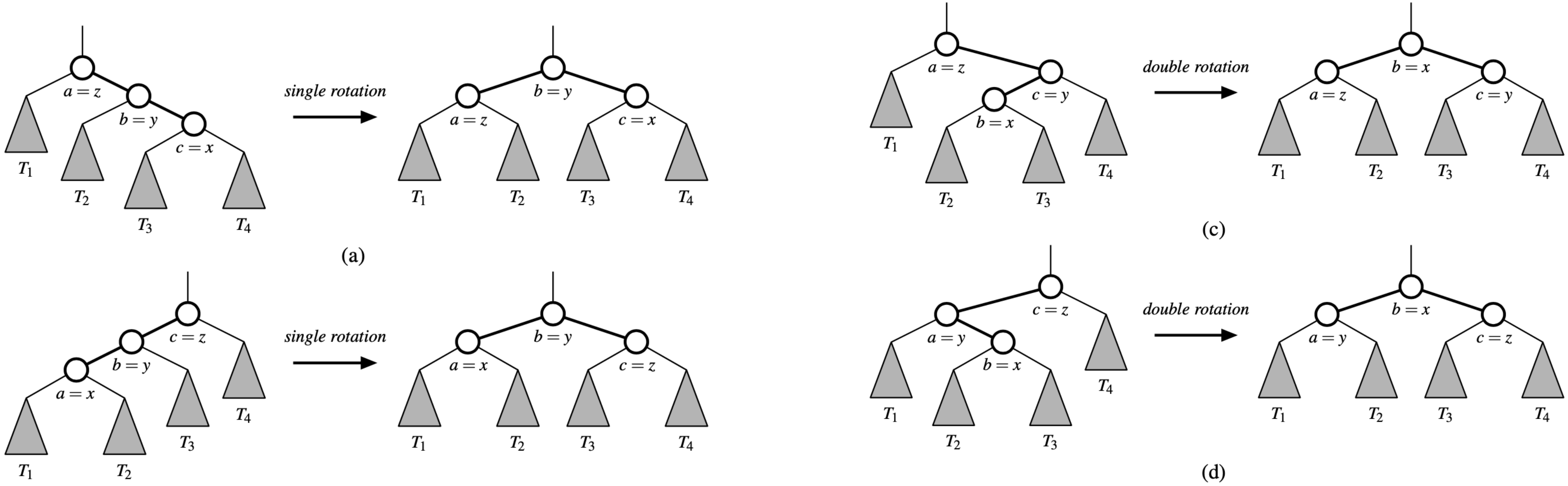


(Recap) Tree Rotation Operation



(Recap) Trinode reconstruction

- Nodes:
  (x, y, z) -> x –parent→ y –parent→ z
  (a, b, c) -> inorder listing of the three positions x, y, z
- Sub-trees: (T_1, T_2, T_3, T_4) -> inorder listing of the four subtrees



Tree reconstruct algorithm

Algorithm restructure(x):

Input: A position x of a binary search tree T that has both a parent y and a grandparent z

Output: Tree T after a trinode restructuring (which corresponds to a single or double rotation) involving positions x, y, and z

1: Let (a, b, c) be a left-to-right (inorder) listing of the positions x, y, and z, and let (T_1, T_2, T_3, T_4) be a left-to-right (inorder) listing of the four subtrees of x, y, and z not rooted at x, y, or 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_1 and T_2 be the left and right subtrees of a, respectively.

4: Let c be the right child of b and let T_3 and T_4 be the left and right subtrees of c, respectively.

Code Fragment 11.9: The trinode restructuring operation in a binary search tree.

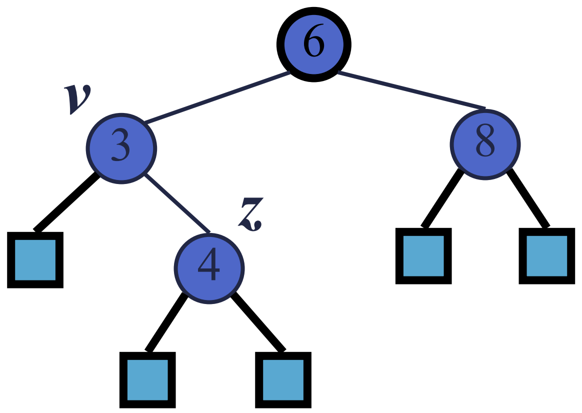
Python Implementation

```python
def _relink(self, parent, child, make_left_child):
  """Relink parent node with child node (we allow child to be None)."""
  if make_left_child:                              # make it a left child
    parent._left = child
  else:                                            # make it a right child
    parent._right = child
  if child is not None:                            # make child point to parent
    child._parent = parent
```

```python
def _rotate(self, p):
  """Rotate Position p above its parent."""
  x = p._node
  y = x._parent                              # we assume this exists
  z = y._parent                              # grandparent (possibly None)
  if z is None:
    self._root = x                           # x becomes root
    x._parent = None
  else:
    self._relink(z, x, y == z._left)         # x becomes a direct child of z
  # now rotate x and y, including transfer of middle subtree
  if x == y._left:
    self._relink(y, x._right, True)          # x._right becomes left child of y
    self._relink(x, y, False)                # y becomes right child of x
  else:
    self._relink(y, x._left, False)          # x._left becomes right child of y
    self._relink(x, y, True)                 # y becomes left child of x
```

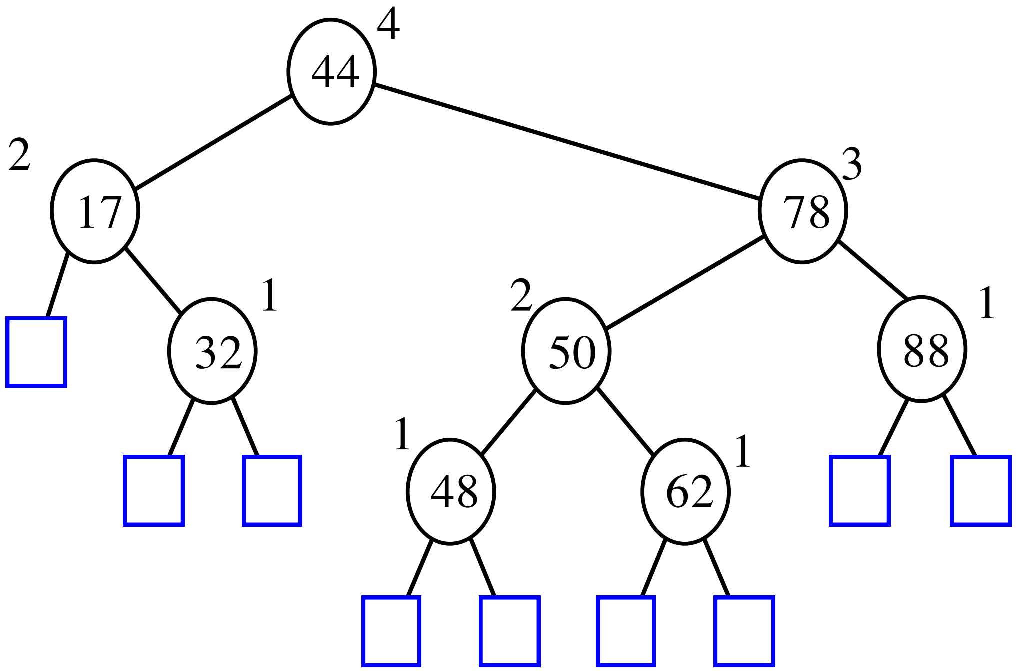
```python
def _restructure(self, x):
  """Perform trinode restructure of Position x with parent/grandparent."""
  y = self.parent(x)
  z = self.parent(y)
  if (x == self.right(y)) == (y == self.right(z)):  # matching alignments
    self._rotate(y)                                  # single rotation (of y)
    return y                                         # y is new subtree root
  else:                                              # opposite alignments
    self._rotate(x)                                  # double rotation (of x)
    self._rotate(x)
    return x                                         # x is new subtree root
```

AVL Trees



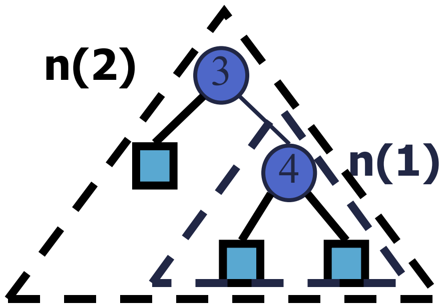
AVL Tree Definition

- 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



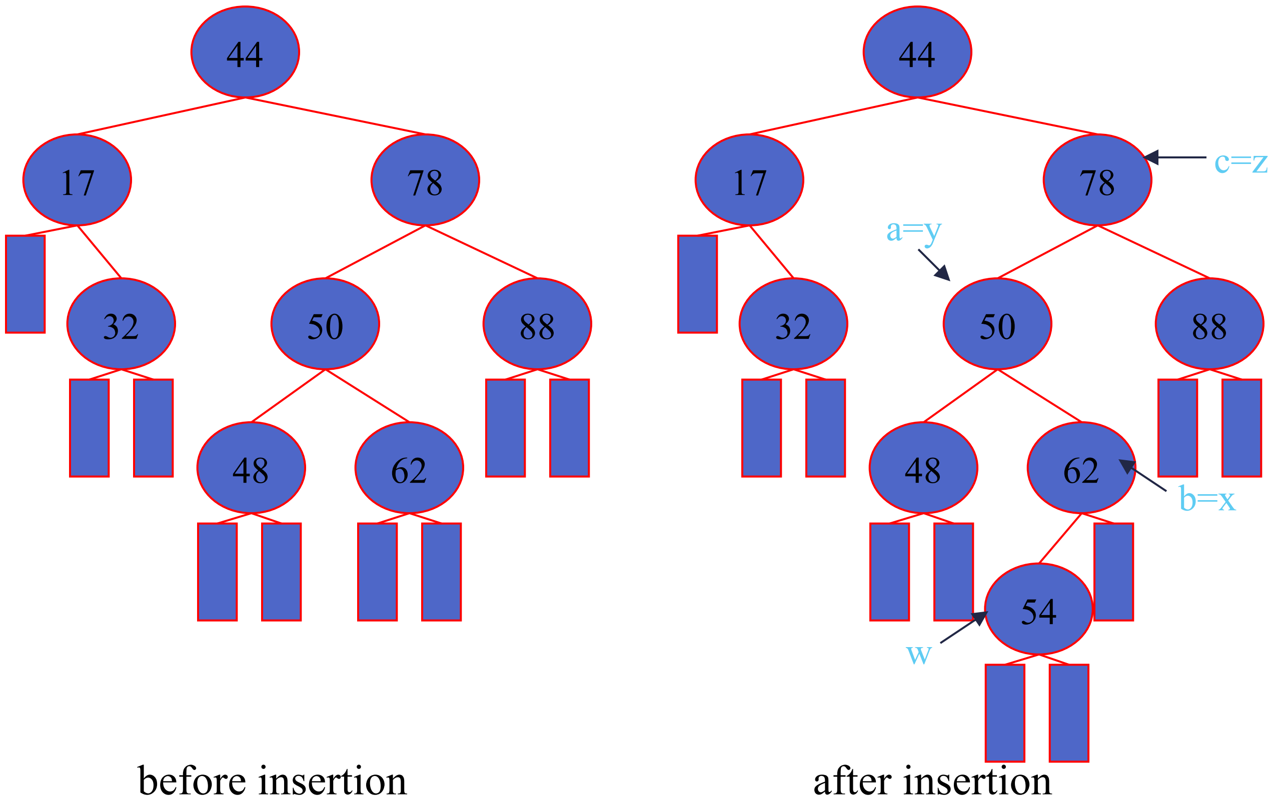
- Fact: The height of an AVL tree storing n keys is O(log n).
- Proof: Let us bound n(h): the minimum number of internal nodes of an AVL tree of height h.
- We easily see that n(1) = 1 and n(2) = 2
- For n > 2, an AVL tree of height h contains the root node, one AVL subtree of height n-1 and another 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(n-6), … (by induction),

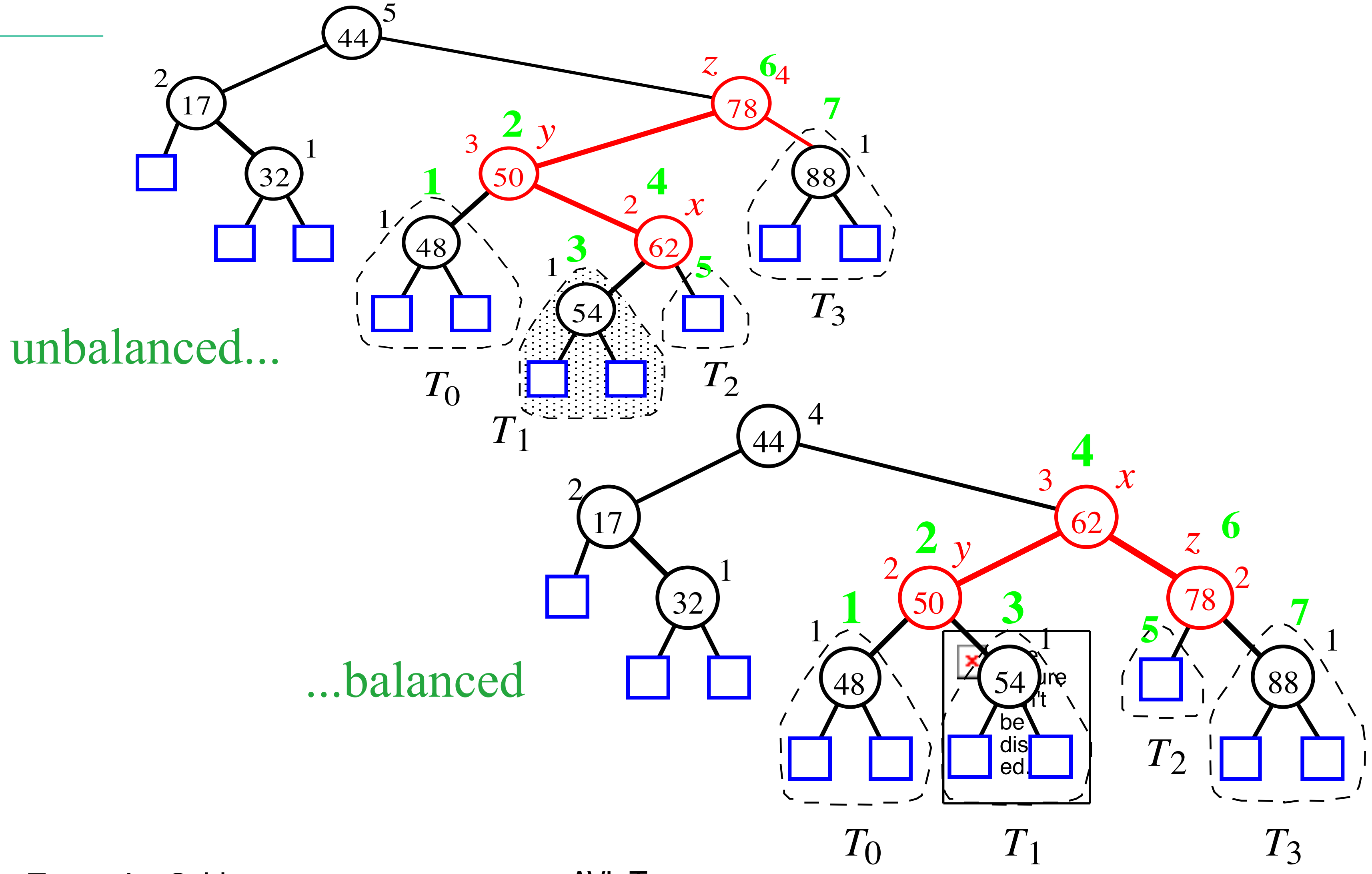
  n(h) > 2^in(h-2i)
- Solving the base case we get: n(h) > $2^{h/2-1}$
- Taking logarithms: h < 2log n(h) +2
- Thus the height of an AVL tree is O(log n)

Insertion

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

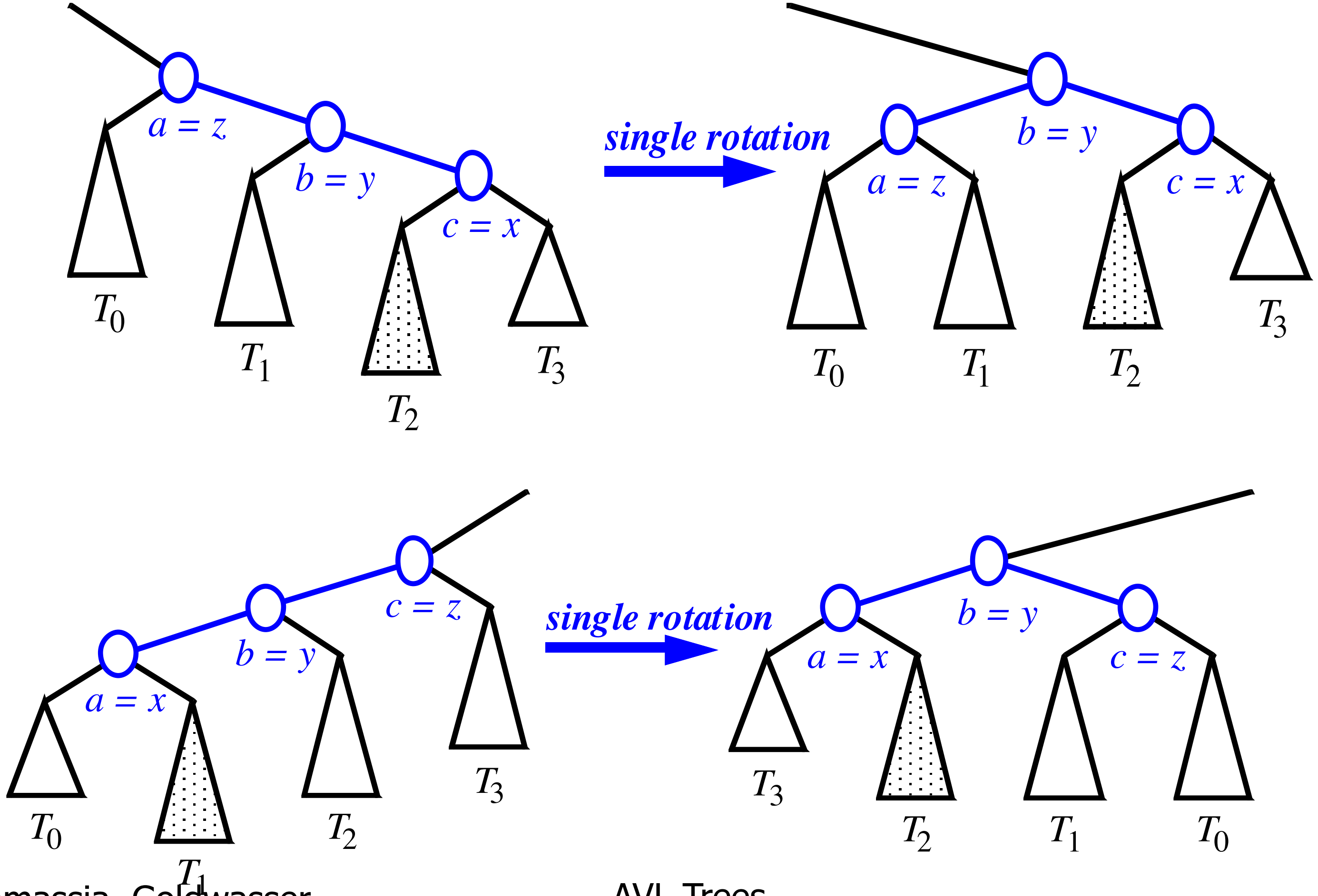


Insertion Example, continued



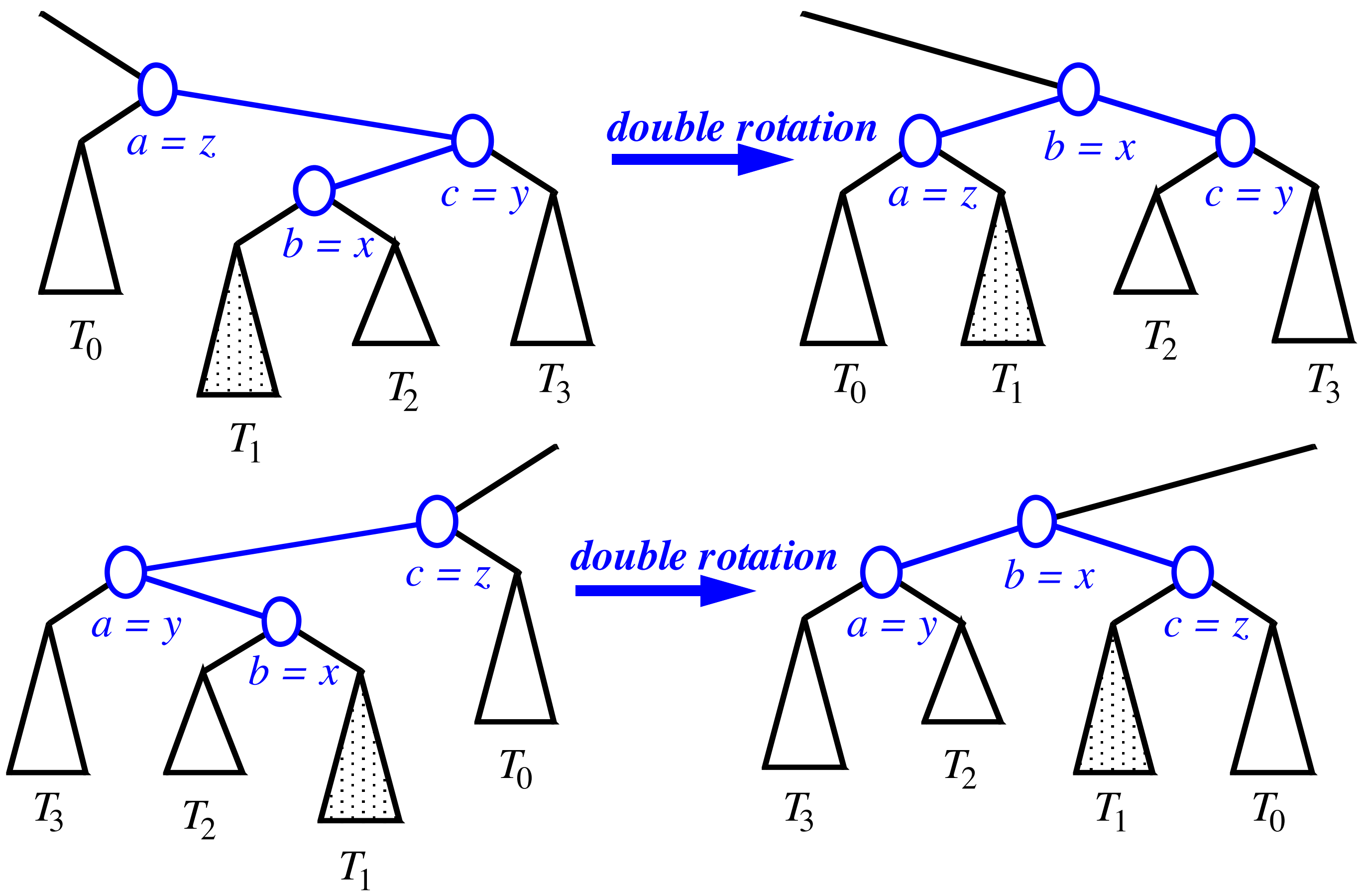
Restructuring (as Single Rotation)

- Single Rotation:



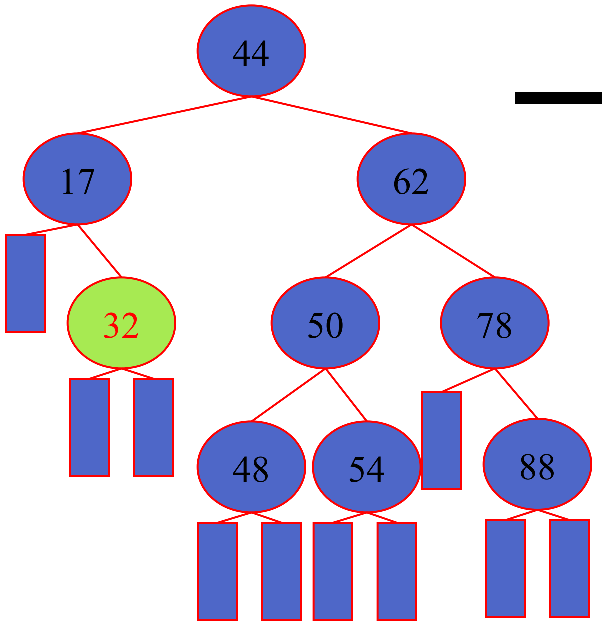
Restructuring (as Double Rotations)

- double rotations:

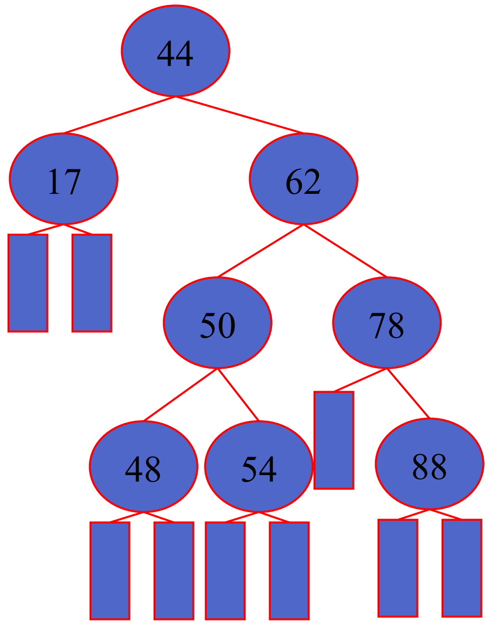


Removal

- 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:



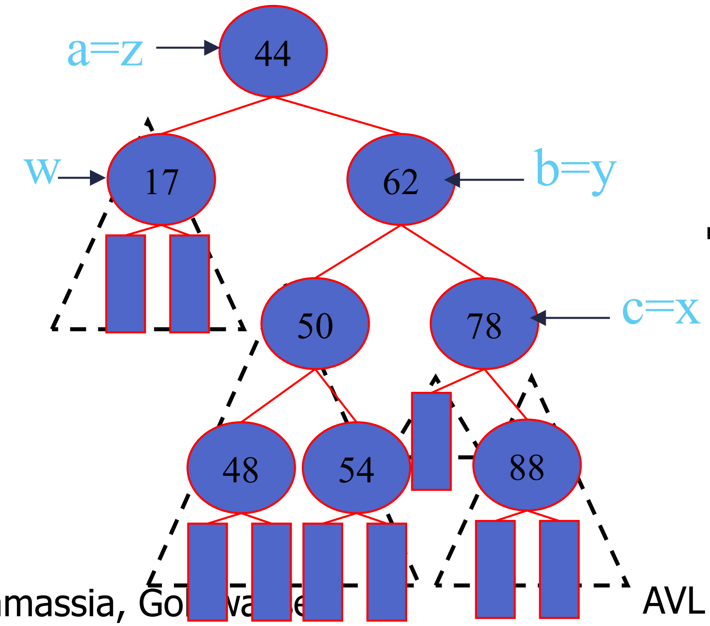
before deletion of 32

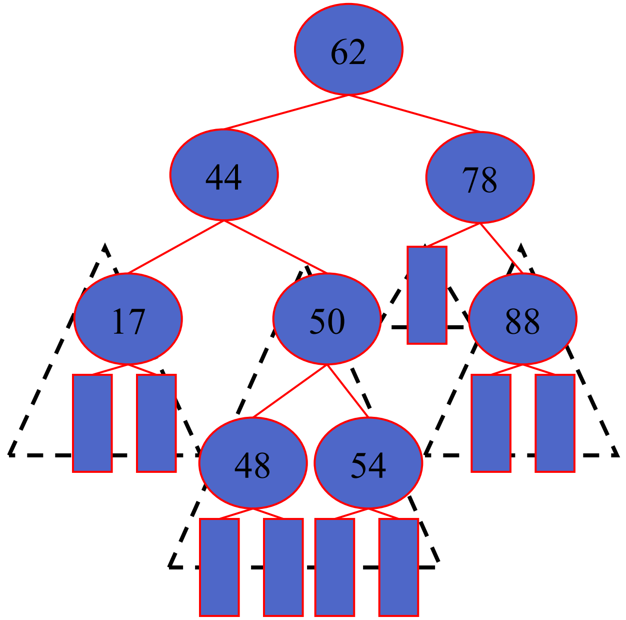


after deletion

Rebalancing after a Removal

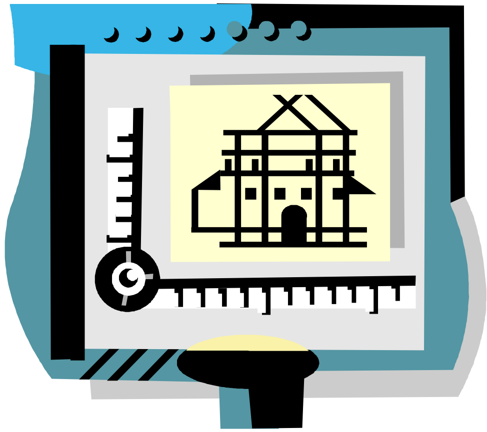
- Let z be the first unbalanced node encountered while travelling up the tree from w. Also, let y be the child of z with the larger height, and let x be the child of y with the larger height
- We perform restructure(x) to restore balance at z
- As this restructuring may upset the balance of another node higher in the tree, we must continue checking for balance until the root of T is reached





© 2013 Goodrich, Tamassia, Goldwasser

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)
```