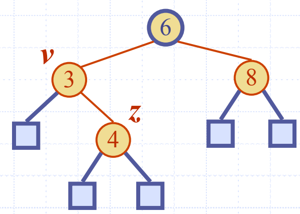
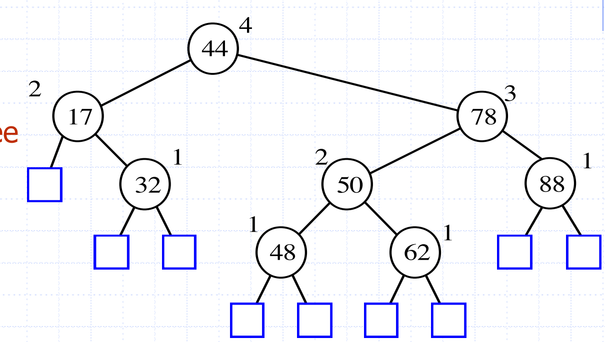
Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

AVL Trees



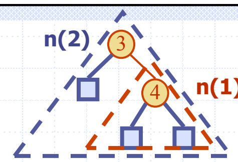
AVL Tree Definition

- 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



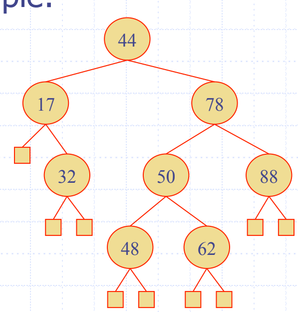
Fact: The height of an AVL tree storing n keys is O(log n).

Proof (by induction): Let us bound n(h): the minimum number of internal nodes of an AVL tree of height h.

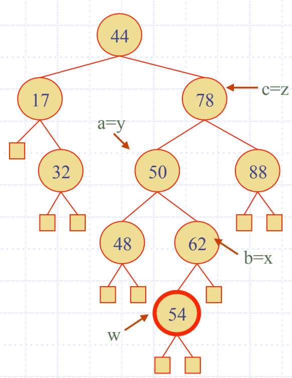
- We easily see that n(1) = 1 and n(2) = 2
- For n > 2, an AVL tree of height h contains the root node, one AVL subtree of height n-1 and another of height n-2.
- That is, n(h) = 1 + n(h-1) + n(h-2)
- Knowing n(h-1) > n(h-2), we get n(h) > 2n(h-2). So
  n(h) > 2n(h-2), n(h) > 4n(h-4), n(h) > 8n(n-6), … (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:



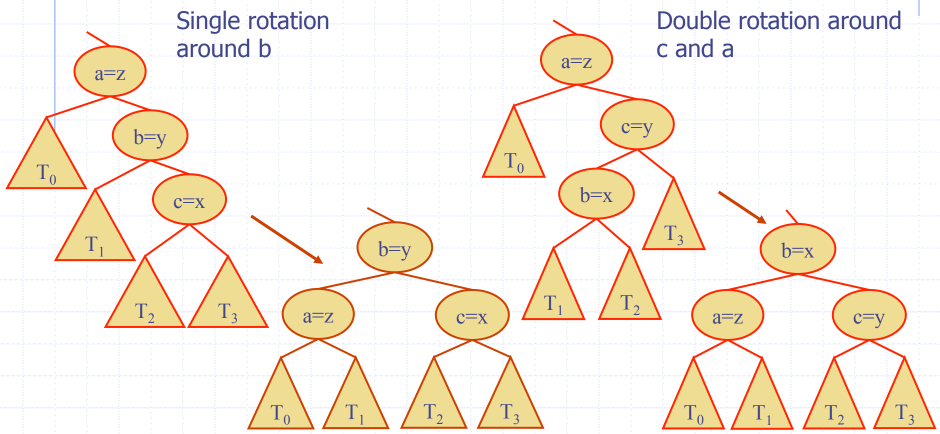
before insertion



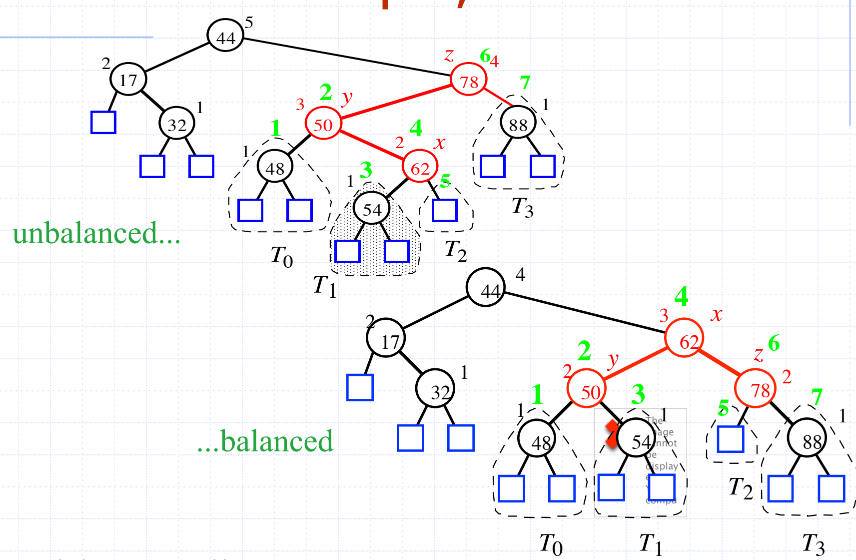
after insertion

Trinode Restructuring

- Let (*a*,*b*,*c*) be the inorder listing of *x*, *y*, *z*
- Perform the rotations needed to make *b* the topmost node of the three

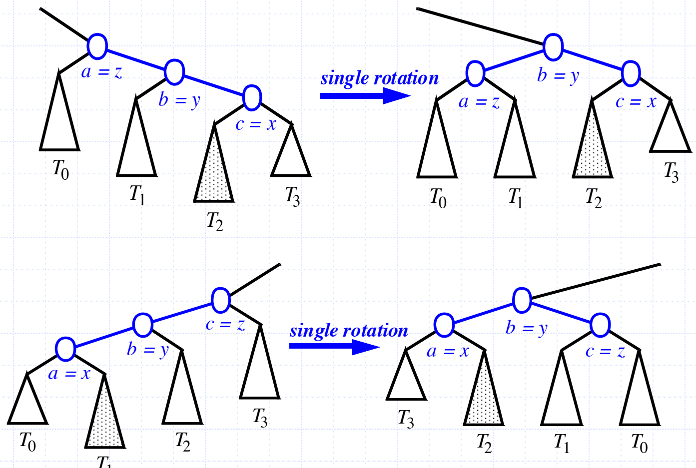


Insertion Example, continued



Restructuring (as Single Rotations)

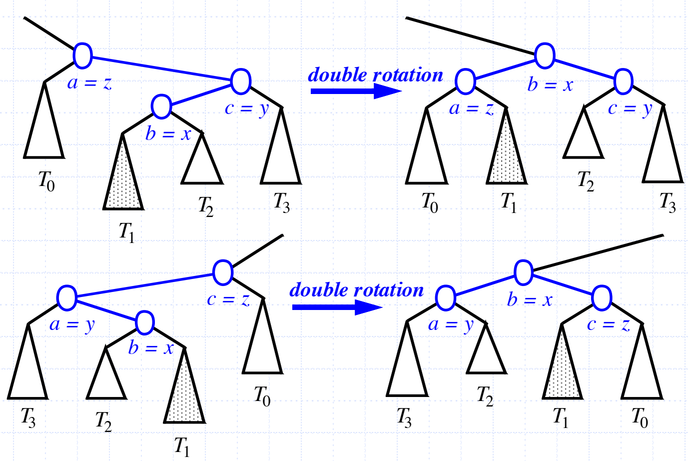
- Single Rotations:



 AVL Trees 7

Restructuring (as Double Rotations)

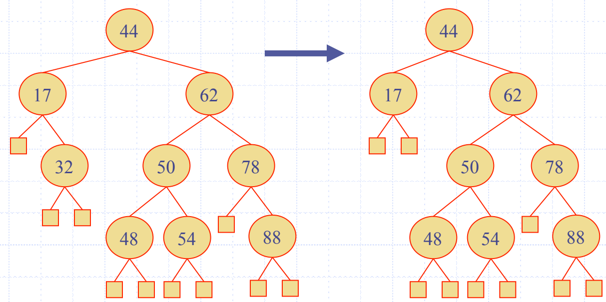
- double rotations:



 AVL Trees 8

Removal

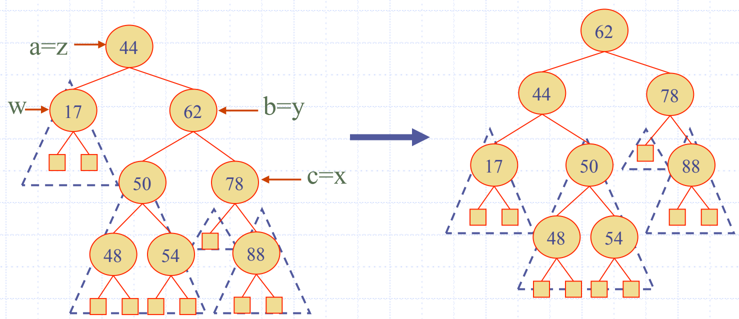
- Removal begins as in a binary search tree, which means the node removed will become an empty external node. Its parent, w, may cause an imbalance.
- Example:



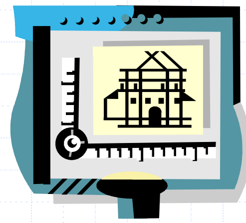
before deletion of 32 after deletion

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 a trinode restructuring 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



- AVL tree storing n items
  - The data structure uses O(n) space
  - A single restructuring 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)

Java Implementation

```
/** An implementation of a sorted map using an AVL tree. */
public class AVLTreeMap<K,V> extends TreeMap<K,V> {
  /** Constructs an empty map using the natural ordering of keys. */
  public AVLTreeMap() { super(); }
  /** Constructs an empty map using the given comparator to order keys. */
  public AVLTreeMap(Comparator<K> comp) { super(comp); }
  /** Returns the height of the given tree position. */
  protected int height(Position<Entry<K,V>> p) {
    return tree.getAux(p);
  }
  /** Recomputes the height of the given position based on its children's heights. */
  protected void recomputeHeight(Position<Entry<K,V>> p) {
    tree.setAux(p, 1 + Math.max(height(left(p)), height(right(p))));
  }
  /** Returns whether a position has balance factor between −1 and 1 inclusive. */
  protected boolean isBalanced(Position<Entry<K,V>> p) {
    return Math.abs(height(left(p)) − height(right(p))) <= 1;
  }
```


Java Implementation, 2

```java
/** Returns a child of p with height no smaller than that of the other child. */
protected Position<Entry<K,V>> tallerChild(Position<Entry<K,V>> p) {
  if (height(left(p)) > height(right(p))) return left(p);          // clear winner
  if (height(left(p)) < height(right(p))) return right(p);         // clear winner
  // equal height children; break tie while matching parent's orientation
  if (isRoot(p)) return left(p);                    // choice is irrelevant
  if (p == left(parent(p))) return left(p);         // return aligned child
  else return right(p);
}
```


Java Implementation, 3

```java
protected void rebalance(Position<Entry<K,V>> p) {
  int oldHeight, newHeight;
  do {
    oldHeight = height(p);                  // not yet recalculated if internal
    if (!isBalanced(p)) {                   // imbalance detected
      // perform trinode restructuring, setting p to resulting root,
      // and recompute new local heights after the restructuring
      p = restructure(tallerChild(tallerChild(p)));
      recomputeHeight(left(p));
      recomputeHeight(right(p));
    }
    recomputeHeight(p);
    newHeight = height(p);
    p = parent(p);
  } while (oldHeight != newHeight && p != null);
}
/** Overrides the TreeMap rebalancing hook that is called after an insertion. */
protected void rebalanceInsert(Position<Entry<K,V>> p) {
  rebalance(p);
}
/** Overrides the TreeMap rebalancing hook that is called after a deletion. */
protected void rebalanceDelete(Position<Entry<K,V>> p) {
  if (!isRoot(p))
    rebalance(parent(p));
}
}
```