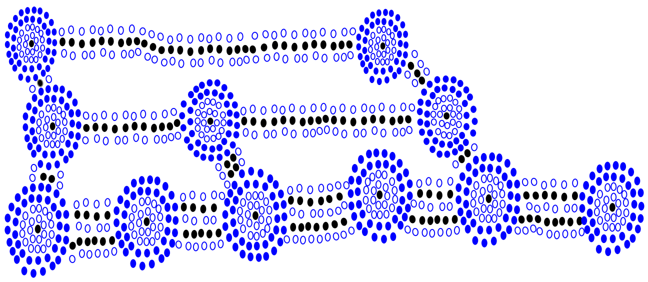
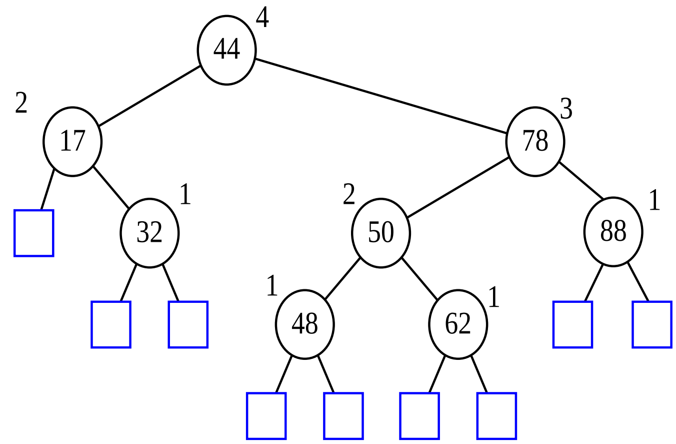
AVL TREES

- AVL Trees



AVL Tree

- **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

- **Proposition**: The height of an AVL tree *T* storing *n* keys is ***O***(log *n*).
- **Justification**: The easiest way to approach this problem is to try to find the minimum number of internal nodes of an AVL tree of height *h*: $n(h)$.
- We see that $n(1) = 1$ and $n(2) = 2$
- for $n \geq 3$, an AVL tree of height *h* with $n(h)$ minimal contains the root node, one AVL subtree of height *n*-1 and the other AVL subtree of height *n*-2.
- *i*.e. $n(h) = 1 + n(h\text{-}1) + n(h\text{-}2)$
- Knowing $n(h\text{-}1) > n(h\text{-}2)$, we get $n(h) > 2n(h\text{-}2)$
  - $n(h) > 2n(h\text{-}2)$
  - $n(h) > 4n(h\text{-}4)$

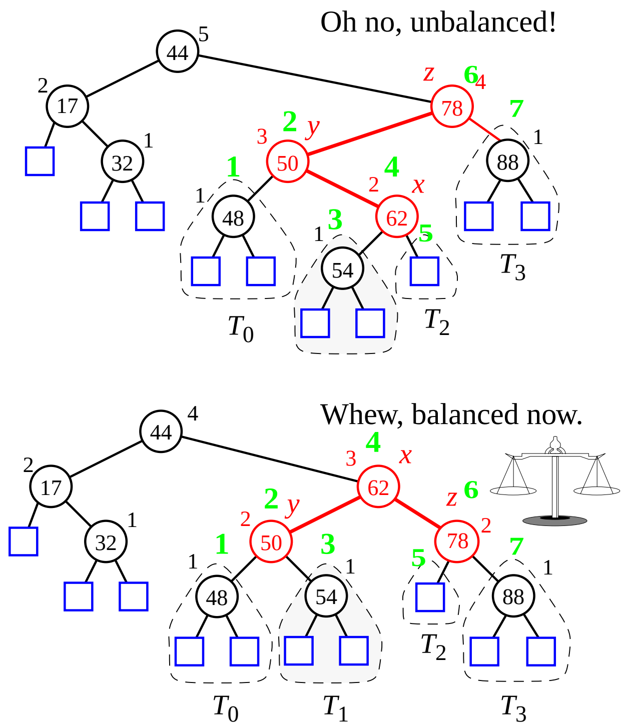
    ...
  - $n(h) > 2^i n(h\text{-}2i)$
- Solving the base case we get: $n(h) \geq 2^{h/2-1}$
- Taking logarithms: $h < 2\log n(h) + 2$
- Thus the height of an AVL tree is ***O***(log *n*)

Insertion

- A binary search tree *T* is called balanced if for every node *v*, the height of *v*'s children differ by at most one.
- Inserting a node into an AVL tree involves performing an expandExternal(*w*) on *T*, which changes the heights of some of the nodes in *T*.
- If an insertion causes *T* to become unbalanced, we travel up the tree from the newly created node until we find the first node *x* such that its grandparent *z* is unbalanced node.
- Since *z* became unbalanced by an insertion in the subtree rooted at its child *y*,
  height(*y*) = height(sibling(*y*)) + 2
- To rebalance the subtree rooted at *z*, we must perform a *restructuring*
  - we rename *x*, *y*, and *z* to *a*, *b*, and *c* based on the order of the nodes in an in-order traversal.
  - *z* is replaced by *b*, whose children are now *a* and *c* whose children, in turn, consist of the four other subtrees formerly children of *x*, *y*, and *z*.

Insertion (contd.)

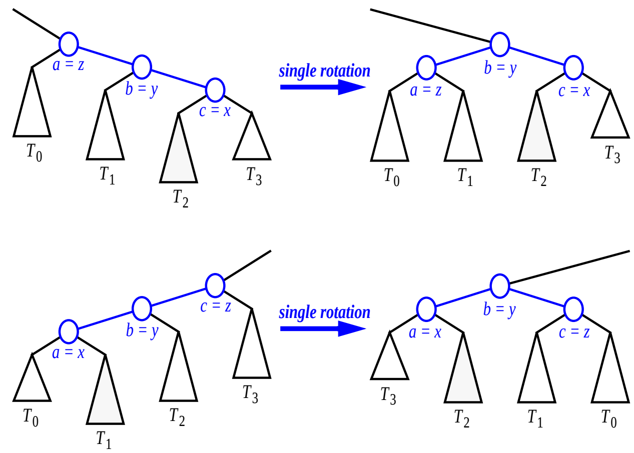
- Example of insertion into an AVL tree.



Restructuring

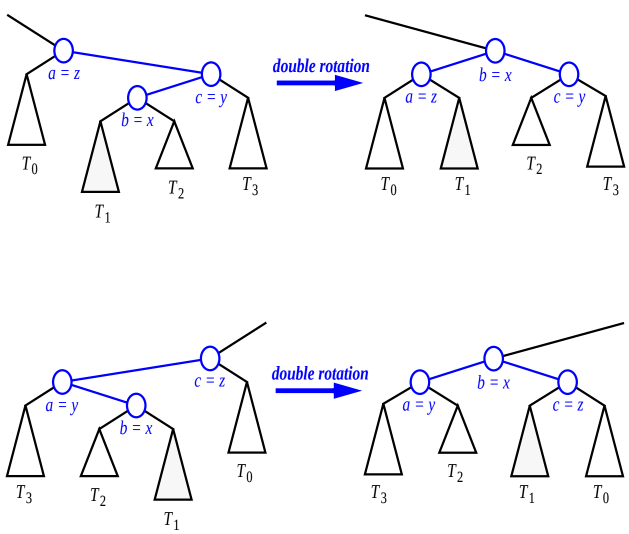
- The four ways to rotate nodes in an AVL tree, graphically represented:

  - Single Rotations:



Restructuring (contd.)

- double rotations:



Restructure Algorithm

Algorithm restructure(x):

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

Output: Tree T restructured by a rotation (either single or double) involving nodes x, y, and z.

1: Let (a, b, c) be an inorder listing of the nodes x, y, and z, and let (T_0, T_1, T_2, T_3) be an inorder listing of the 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_0, T_1 be the left and right subtrees of a, respectively.

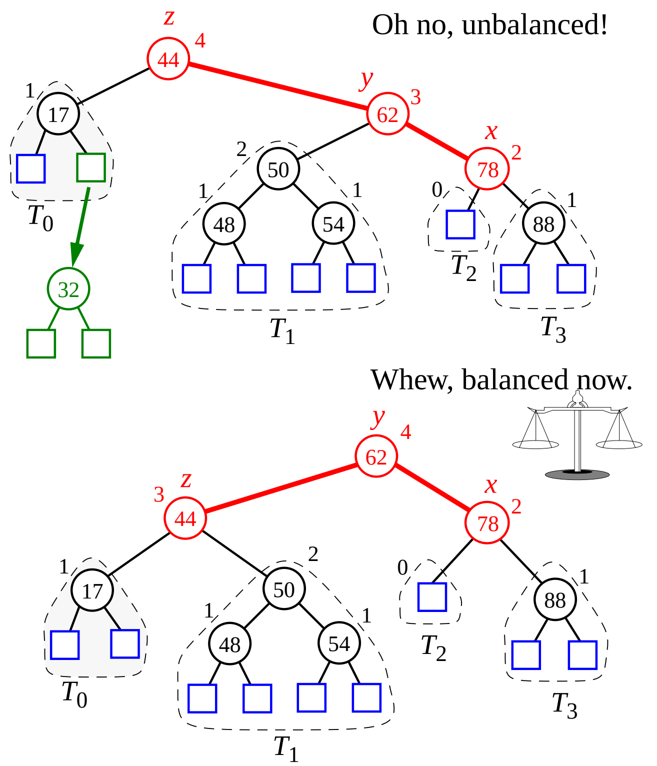
4. Let c be the right child of b and let T_2, T_3 be the left and right subtrees of c, respectively.

Removal

- We can easily see that performing a removeAboveExternal(*w*) can cause *T* to become unbalanced.
- 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 can perform operation restructure(*x*) to restore balance at the subtree rooted 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.

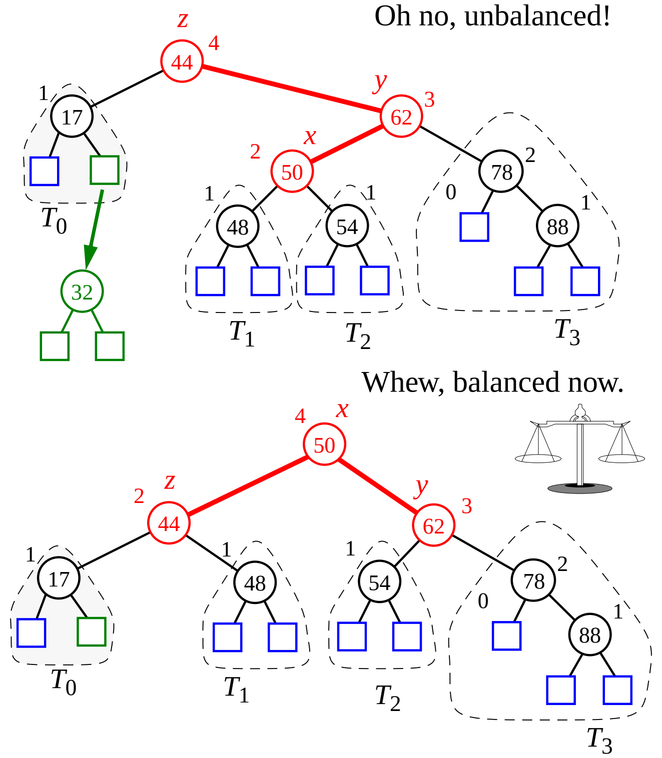
Removal (contd.)

- example of deletion from an AVL tree:



Removal (contd.)

- example of deletion from an AVL tree



Implementation

- A Java-based implementation of an AVL tree requires the following node class:

```
public class AVLItem extends Item {
  int height;

  AVLItem(Object k, Object e, int h) {
    super(k, e);
    height = h;
  }

  public int height() {
    return height;
  }

  public int setHeight(int h) {
    int oldHeight = height;
    height = h;
    return oldHeight;
  }
}
```


Implementation (contd.)

```
public class SimpleAVLTree
  extends SimpleBinarySearchTree
  implements Dictionary {

public SimpleAVLTree(Comparator c) {
    super(c);
    T = new RestructurableNodeBinaryTree();
  }
  private int height(Position p) {
    if (T.isExternal(p))
      return 0;
    else
      return ((AVLItem) p.element()).height();
  }
  private void setHeight(Position p) { // called only
                                       // if p is internal
    ((AVLItem) p.element()).setHeight
      (1 + Math.max(height(T.leftChild(p)),
                    height(T.rightChild(p))));
  }
```

Implementation (contd.)

```
private boolean isBalanced(Position p) {
    // test whether node p has balance factor
    // between -1 and 1
  int bf = height(T.leftChild(p)) - height(T.rightChild(p));
  return ((-1 <= bf) && (bf <= 1));
}


private Position tallerChild(Position p) {
    // return a child of p with height no
    // smaller than that of the other child
  if(height(T.leftChild(p)) >= height(T.rightChild(p)))
    return T.leftChild(p);
  else
    return T.rightChild(p);
}
```


Implementation (contd.)

```
private void rebalance(Position zPos) {
//traverse the path of T from zPos to the root;
//for each node encountered recompute its
//height and perform a rotation if it is
//unbalanced
  while (!T.isRoot(zPos)) {
    zPos = T.parent(zPos);
    setHeight(zPos);
    if (!isBalanced(zPos)) { // perform a rotation
      Position xPos = tallerChild(tallerChild(zPos));
      zPos = ((RestructurableNodeBinaryTree)
              T).restructure(xPos);
      setHeight(T.leftChild(zPos));
      setHeight(T.rightChild(zPos));
      setHeight(zPos);
    }
  }
}
```


Implementation (contd.)

```
public void insertItem(Object key, Object element)
    throws InvalidKeyException {
  super.insertItem(key, element);// may throw an
                                 // InvalidKeyException
  Position zPos = actionPos; // start at the
                             // insertion position
  T.replace(zPos, new AVLItem(key, element, 1));
  rebalance(zPos);
}

public Object remove(Object key)
    throws InvalidKeyException {
  Object toReturn = super.remove(key); // may throw
                                // an InvalidKeyException
  if (toReturn != NO_SUCH_KEY) {
    Position zPos = actionPos; // start at the
                               // removal position
    rebalance(zPos);
  }
  return toReturn;
}
}
```