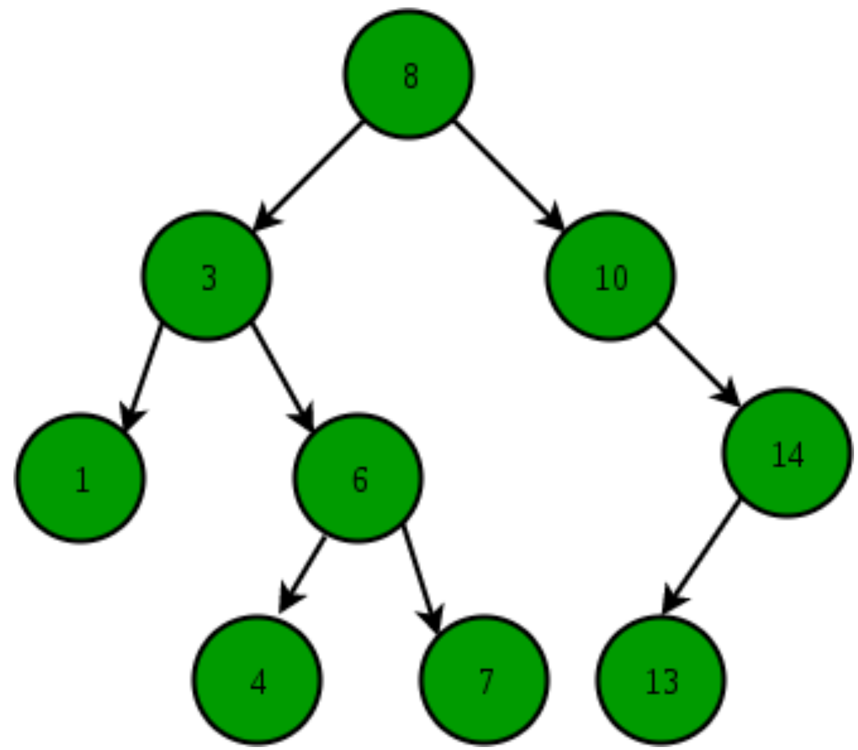
CS206

Search Trees, AVL Trees

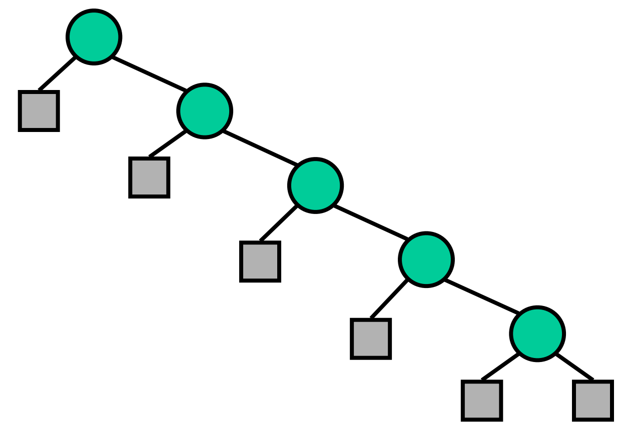
Binary Search Trees

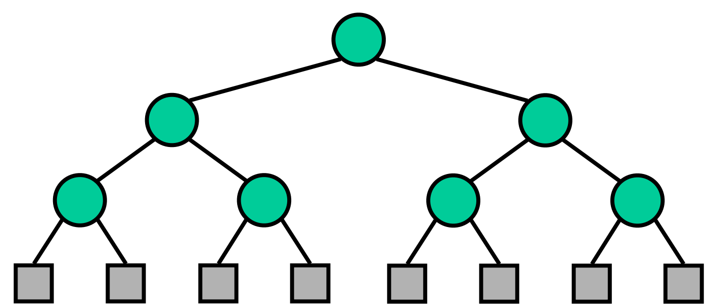
- For all nodes
  - The left node is less than parent
  - The right node is greater than parent



Binary Search Trees

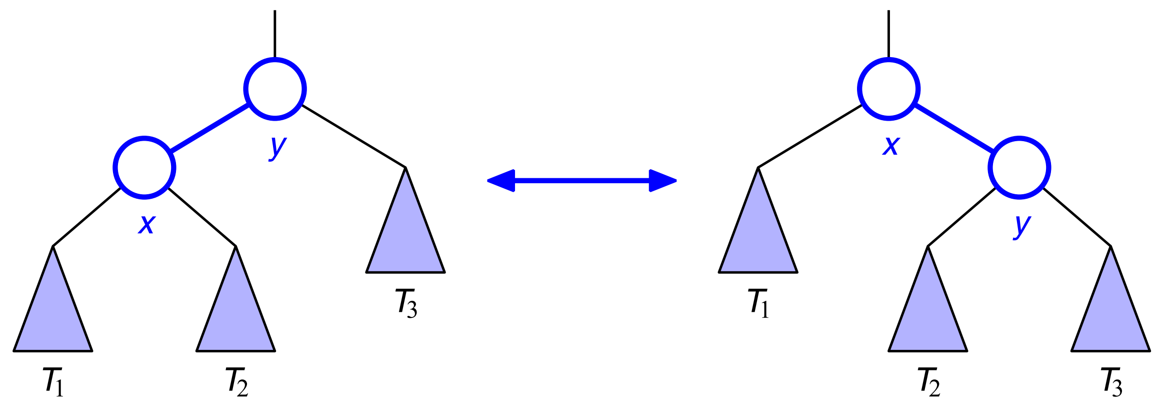
- Performance is directly affected by the height of tree
- All operations are $O(h)$
- $h = O(n)$ worst case
- $h = O(logn)$ best case
- Expected $O(logn)$ if tree is "balanced"
  - balance — generally same number of nodes in left and right subtrees





Balanced Search Trees

- A variety of algorithms that augment a standard BST with occasional operations to reshape, reduce height and maintain balance.
- General approach == Rotation: move a child to be above its parent, then relink subtrees to maintain BST order
  - $O(1)$



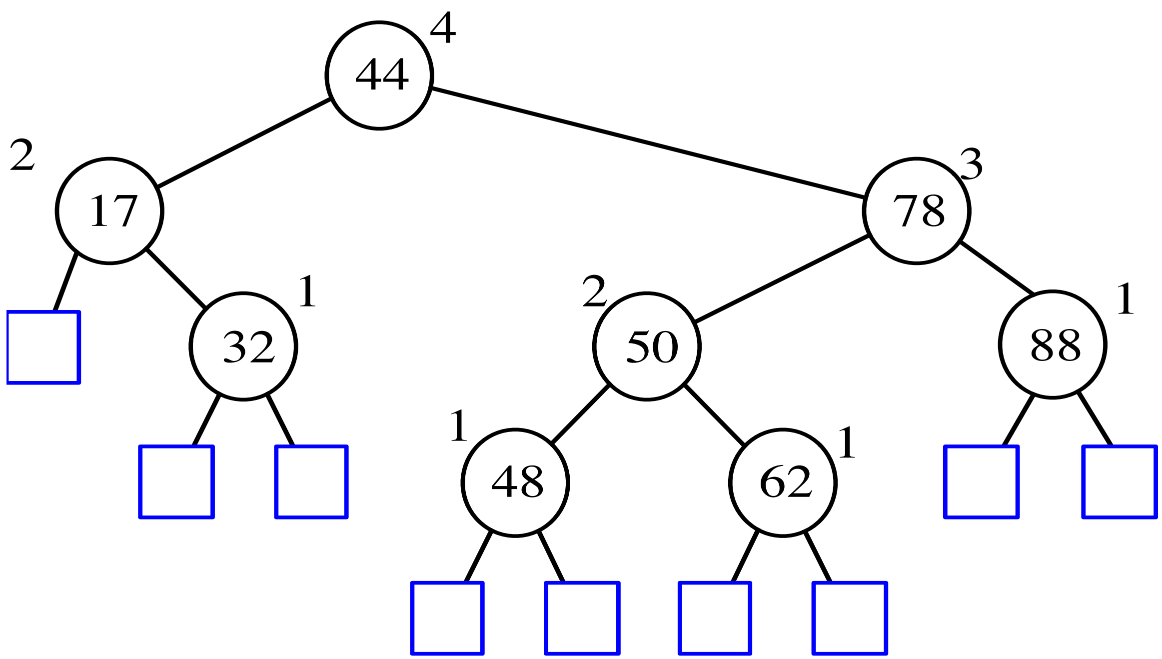
Tree Rotation

- Rotation can be to the right or left
- Rotate reduces/increases the depth of nodes in subtrees T_1 and T_3 by 1
- Rotation maintains BST order
- One or more rotations can be combined to provide broader rebalancing

AVL Tree

- Adelson-Velski and Landis (1962)
- Height-balance property
  - For every internal node, the heights of the two children differ by at most 1
- Any binary tree satisfying the height-balance property is an AVL tree
- A height-balanced tree has height O(lg n)
  - max height is provably 1.44*lg(n)
    - see book pg 481 for proof (kind of)

AVL Tree Example



Insertion

- Maintain with each node the height of its subtree.
- On insertion, first recur down through tree to insert.
- Then as you unwind recursion, update the height of each node.
- If height changes, check the height of other child
  - if not in balance then fix

Insertion code to maintain height

(the only code today!!!)

```
// assumes public insert from linked binary tree
private int iInsert(Node treepart, E toBeAdded) {
     int cmp =
treepart.element.compareTo(toBeAdded);
     if (cmp==0)
          return -11111; // the item is in the tree
     int dpth=1;
     if (cmp<0) {
          if (treepart.left==null)
           treepart.left=new Node(toBeAdded);
          else
           dpth = 1 + iInsert(treepart.left,
toBeAdded);
     }
     else { // cmp>0
          if (treepart.right==null)
           treepart.right=new Node(toBeAdded);
          else
           dpth= 1 + iInsert(treepart.right,
toBeAdded);
     }
     treepart.height=treepart.height>dpth?
treepart.height:dpth;
     return treepart.height;
    }
```

```
private class Node {
     Comparable<E> element;
     int height;
     Node right;
     Node left;

     public Node(Comparable<E> e)
{
          height = 0;
          element=e;
          right=null;
          left=null;
     }
}
```

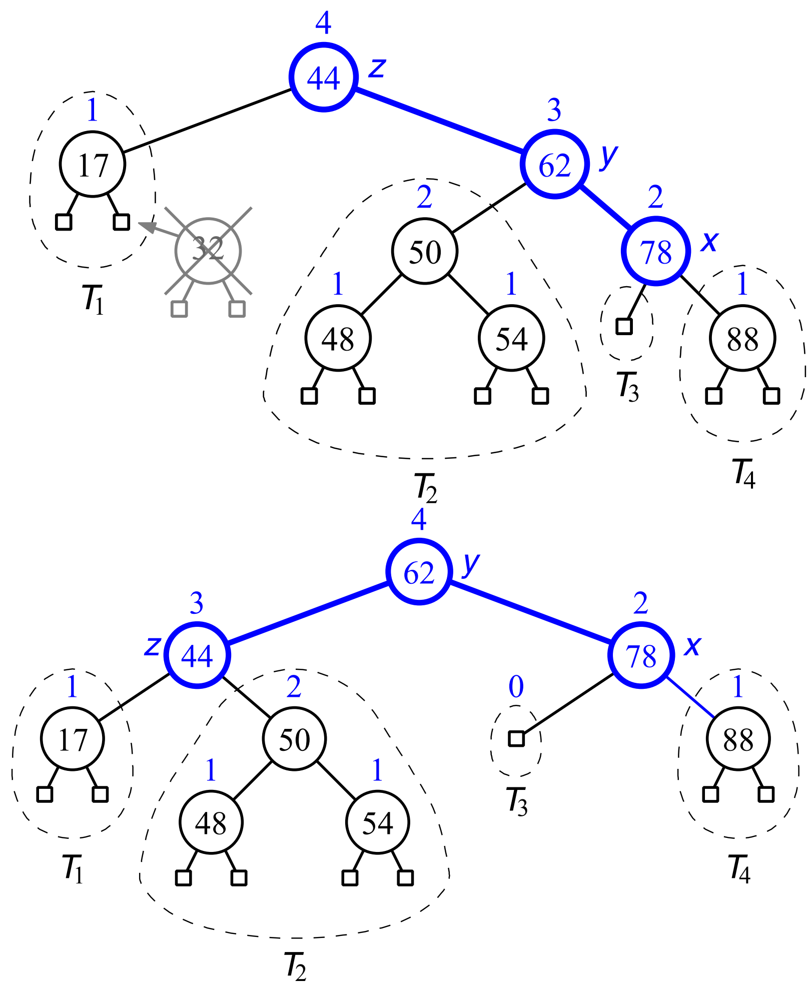
Fixing height imbalances
Rotation!!

- Two types of rotation
- Single
  - left subtree of left node causes imbalance
  - right subtree of right node causes imbalance
- Double
  - right subtree of left node causes imbalance
  - left subtree of right node causes imbalance

AVL Animation

Deletion

- Deletion removes a node with 0 or 1 child
  - recall deletion from binary tree for node with 2 children.
- Deletion may reduce the height of parent
- Rotate to rebalance just like insertion



$O(logn)$ Rotations

- Unlike insertion where rotation of the nearest unbalanced ancestor restores the balance globally
- On deletion, rotation of the nearest unbalanced ancestor only guarantees balance locally to the subtree
- Worst-case requires $O(logn)$ rotations up the tree to restore balance globally