Balanced Search Trees

- Binary search trees keep keys ordered, with efficient lookup
  - Insert, Delete, Find, all are $O(\log n)$ in average case
  - Worst case is bad
  - Compared to hashing? Advantages?

- Balanced trees are guaranteed $O(\log n)$ in the worst case
  - Fundamental operation is a rotation: keep tree roughly balanced
  - AVL tree was the first one, still studied since simple conceptually
  - Red-Black tree uses rotations, harder to code, better in practice
  - B-trees are used when data is stored on disk rather than in memory
  - Splay trees rebalance “sometimes”, amortized good performance, simple (relatively) to implement
Rotations and balanced trees

- **Height-balanced trees**
  - For every node, left and right subtree heights differ by at most 1
  - After insertion/deletion need to rebalance
  - Every operation leaves tree in a balanced state: *invariant property* of tree

- **Find deepest node that’s unbalanced**
  - On path from root to inserted/deleted node
  - Rebalance at this unbalanced point only

Are these trees height-balanced?
Tree Balancing example

- **Tree with AVL property**
  - What happens if we insert a node with a value of 1?

- **Need to rotate tree to maintain AVL property**
  - What would have happened if we inserted a node with a value of 5?
  - Which nodes in the tree are unbalanced?
  - What are the four ways an insertion can unbalance a tree?
Rotation to rebalance

When a node N is unbalanced height differs by 2 (must be more than one)

- Change N->left->left • doLeft
- Change N->left->right • doLeftRight
- Change N->right->left • doRightLeft
- Change N->right->right • doRight

First/last cases are symmetric

Middle cases require two rotations

- First of the two puts tree into doLeft or doRight

Tree * doLeft(Tree * root)
{
  Tree * newRoot = root->left;
  root->left = newRoot->right;
  newRoot->right = root;
  return newRoot;
}
Rotation to rebalance

- Suppose we add a new node in right subtree of left child of root
  - Single rotation can’t fix
  - Need to rotate twice
- First stage is shown at bottom
  - Rotate blue node right
  - This is left child of unbalanced

```c
Tree * doRight(Tree * root)
{
    Tree * newRoot = root->right;
    root->right = newRoot->left;
    newRoot->left = root;
    return newRoot;
}
```
Double rotation complete

- Calculate where to rotate and what case, do the rotations

```c
Tree * doRight(Tree * root)
{
    Tree * newRoot = root->right;
    root->right = newRoot->left;
    newRoot->left = root;
    return newRoot;
}
```

```c
Tree * doLeft(Tree * root)
{
    Tree * newRoot = root->left;
    root->left = newRoot->right;
    newRoot->right = root;
    return newRoot;
}
```
Other trees

- Red-black tree uses same rotations, but can rebalance in one pass, contrast to AVL tree
  - In AVL case, insert, calculate balance factors, rebalance
  - In Red-black tree can rebalance on the way down, code is more complex, but doable
- Red-black trees used in practice, efficient, guaranteed log n
  - STL in C++ uses red-black tree for map and set classes
  - Standard java.util.TreeMap/TreeSet use red-black
- Treaps: probablistically balanced trees
  - Combines aspects of BSTs, heaps, and hashing
- B-trees, 2-3 trees, 2-3-4 trees (all variants of the same thing)
  - Data stored on disk, need higher branching factor than binary search tree
  - O(log n), but much faster in practice