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

Rotation to rebalance

When a node $N$ is unbalanced height differs by 2 (must be more than one)
- Change $N->\text{left}->\text{left}$
- $\text{doLeft}$
- Change $N->\text{left}->\text{right}$
- $\text{doLeftRight}$
- Change $N->\text{right}->\text{left}$
- $\text{doRightLeft}$
- Change $N->\text{right}->\text{right}$
- $\text{doRight}$

First/last cases are symmetric
Middle cases require two rotations
- First of the two puts tree into $\text{doLeft}$ or $\text{doRight}$

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

Tree * doRight(Tree * root)

```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

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

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

- 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

Trie: efficient search of words/suffixes

- A trie (from retrieval, but pronounced “try”) supports
  - Insertion: a word into the trie (delete and look up)
  - These operations are $O(\text{size of string})$ regardless of how many strings are stored in the trie! Guaranteed!

- In some ways a trie is like a 128 (or 26 or alphabet-size) tree, one branch/edge for each character/letter
  - Node stores branches to other nodes
  - Node stores whether it ends the string from root to it

- Extremely useful in DNA/string processing
  - monkeys and typewriter simulation which is similar to some methods used in Natural Language understanding (n-gram methods)

Trie picture and code (see trie.cpp)

- To add string
  - Start at root, for each char create node as needed, go down tree, mark last node

- To find string
  - Start at root, follow links
  - If NULL/0 not contained
  - Check word flag in node

- To print all nodes
  - Visit every node, build string as nodes traversed

- What about union and intersection?
  - Indicates word ends here