Balanced Search Trees

- BST: efficient lookup, insertion, deletion
  - Average case: $O(\log n)$ for all operations since find is $O(\log n)$ [complexity of insert after find is $O(1)$, why?]
  - Worst case is bad, what's big-Oh? What's the tree shape?

Balanced Search trees

- Use rotations to maintain balance, different implementations rotate/rebalance at different times
- AVL tree is conceptually simple, bookkeeping means coefficient for big-Oh is higher than other ideas
- Red-black tree harder to code but good performance: basis for Java map classes and most C++ map classes

Balance trees we won't study

- B-trees are used when data is both in memory and on disk
  - File systems, really large data sets
  - Rebalancing guarantees good performance both asymptotically and in practice. Differences between cache, memory, disk are important

- Splay trees rebalance during insertion and during search, nodes accessed often move closer to root
  - Other nodes can move further from root, consequences?
    - Performance for some nodes gets better, for others ...
    - No guarantee running time for a single operation, but guaranteed good performance for a sequence of operations, this is good amortized cost (vector push_back)

Balanced trees we will study

- Both kinds have worst-case $O(\log n)$ time for tree operations
- AVL (Adelson-Velskii and Landis, 1962)
  - Nodes are “height-balanced”, subtree heights differ by 1
  - Rebalancing requires per-node bookkeeping of height
- 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
  - STL in C++ uses red-black tree for map and set classes
  - Standard java.util.TreeMap/TreeSet use red-black

Balanced Trees in Practice

- See useetreeset.cpp
  - Balanced and unbalanced search trees as basis for sets
  - Contrast with red-black trees in STL, see setstl.cpp

- Reasons STL code is faster than AVL code?
  - Intrinsically better algorithm, one pass vs. two pass
  - Makes better use of storage allocation? (conceivably)
  - More/better programming?

- Reading code: STL v. Java implementations
  - What does tradition tell us?
Rotations and balanced trees

- Height-balanced trees
  - For every node, left and right subtree heights differ by at most 1
  - Every operation leaves tree in a balanced state: **invariant** property of tree
- Find deepest node that’s unbalanced then make sure:
  - On path from root to inserted/deleted node
  - Rebalance at this unbalanced point only

Rotation up close (doLeft)

- Why is this called doLeft?
  - N will no longer be root, new value in left->left subtree
  - Left child becomes new root
- Rotation isn’t “to the left”, but rather “brings left child up”

Rotation to rebalance

- When a node N (root) is unbalanced height differs by 2 (must be more than one)
  - Change N->left->left
    • doLeft
  - Change N->right->left
    • doRightLeft
  - Tree * doLeft(Tree * root) → Change N->right->left
    • doRightLeft
  - Tree * newRoot = root->left.Change N->right->right
    root->left = newRoot->right; ● doRight
    newRoot->right = root;
  - First/last cases are symmetric
  - Middle cases require two rotations
    • First of the two puts tree into doLeft or 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
    • (its right child takes its place)
  - This is left child of unbalanced

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

```
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;
}
```

AVL tree practice

- Insert into AVL tree:
  - 18 10 16 12 6 3 8 13 14
  - After adding 16: doLeftRight
  - After 3, doLeft on 16

AVL practice: continued, and finished

- After adding 13, ok
- After adding 14, not ok
  - doRight at 12

Rodney Brooks

- *Flesh and Machines:* "We are machines, as are our spouses, our children, and our dogs... I believe myself and my children all to be mere machines. But this is not how I treat them. I treat them in a very special way, and I interact with them on an entirely different level. They have my unconditional love, the furthest one might be able to get from rational analysis."
- Director of MIT AI Lab