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?
  - If we can guarantee $\log n$ in worst case, why might this be preferable to hashing? Properties of search tree?

- **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 more 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 (Adel’son-Velskii and Landis), 1962**
  - Nodes are “height-balanced”, subtree heights differ by 1
  - Rebalancing requires per-node bookkeeping of height
  - [http://www.seanet.com/users/arsen/avltree.html](http://www.seanet.com/users/arsen/avltree.html)

- **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
  - STI in C++ uses red-black tree for map and set classes
  - Standard `java.util.TreeMap/TreeSet` use red-black

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 then make sure:
  - On path from root to inserted/deleted node
  - Rebalance at this unbalanced point only

- **Are these trees height-balanced?**

- **Are these trees height-balanced?**

- **Are these trees height-balanced?**

- **Are these trees height-balanced?**
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->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

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”
  - doLeftChildRotate?

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

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 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

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, not found
    - Check word flag at end
- To print all nodes
  - Visit every node, build string as nodes traversed
- What about union and intersection?
  - Indicates word ends here
Boggle: Tries, backtracking, structure

Find words on 4x4 grid
- Adjacent letters: H E S I W L B D A B I M Z E V S
- No re-use in same word

Two approaches to find all words
- Try to form every word on board
  - Look up prefix as you go
    - Trie is useful for prefixes
  - Look up every word in dictionary
    - For each word: on board?
- ZEAL and SMILES

Search board for word: trieBoggle.cpp

```cpp
void search(int row, int col, TrieNode * t, string soFar)
// pre: row, col are on board, soFar is valid prefix, 
// string constructed on board during current search 
// t represents the path in the trie of soFar 
{ if (!legal(row,col) || isVisited(row,col)) return;
  char ch = myBoard[row][col]; // check if still a prefix
  Node * child = t->links[ch]; // NOT a prefix, stop
  if (child == 0) return;      // still prefix, continue
  markVisited(row,col);        // don't revisit
  string newPrefix = word + ch;
  if (child->isWord) cout << newPrefix << endl;
  doFind(row-1,col-1,child,newPrefix); // up-left
  doFind(row-1,col,  child,newPrefix); // straight up
  doFind(row-1,col+1,child,newPrefix); // still 5 more calls
  unVisit(row,col); // now ok to revisit
}
```

Search for all words: boggle.cpp

```cpp
bool wordFound(const string& s, const Point& p)
// pre: s is suffix of word searched for, prefix so far 
// is found and last letter of found prefix at p[row, col]
// if (s.length() == 0) return true; // no more suffix, done
{ tvector<Point> points = myPointsFor(s[0]);
  for(int k=0; k < points.size(); k++) {
    Point nextP = points[k];
    if (IsAdjacent(p,nextP) && ! isVisited(nextP)) {
      markVisited(nextP);     // don't visit again
      if (wordFound(s.substr(1,s.length()-1),nextP)) {
        return true;
      }
      unVisit(nextP);         // ok to visit again
    }
  }
  return false; // tried to find s, failed in all attempts
}
```

Danny Hillis

- The third culture consists of those scientists and other thinkers in the empirical world who, through their work and expository writing, are taking the place of the traditional intellectual in rendering visible the deeper meanings of our lives, redefining who and what we are.

(Wired 1998) And now we are beginning to depend on computers to help us evolve new computers that let us produce things of much greater complexity. Yet we don’t quite understand the process - it’s getting ahead of us. We’re now using programs to make much faster computers so the process can run much faster.

That’s what’s so confusing – technologies are feeding back on themselves; we’re taking off. We’re at that point analogous to when single-celled organisms were turning into multicelled organisms. We are amoebas and we can’t figure out what the hell this thing is that we’re creating.