From doubly-linked lists to binary trees

- Instead of using prev and next to point to a linear arrangement, use them to divide the universe in half
  - Similar to binary search, everything less goes left, everything greater goes right
  - How do we search?
  - How do we insert?

Implementing binary trees

- Trees can have many shapes: short/bushy, long/stringy
  - if height is $h$, number of nodes is between $h$ and $2^h - 1$
  - single node tree: height = 1, if height = 3
  - Java implementation, similar to doubly-linked list

```
public class TreeNode {
    String info;
    TreeNode left;
    TreeNode right;
    TreeNode(String s, TreeNode llink, TreeNode rlink) {
        info = s; left = llink; right = rlink;
    }
}
```

Basic tree definitions

- Binary tree is a structure:
  - empty
  - root node with left and right subtrees
- terminology: parent, children, leaf node, internal node, depth, height, path
  - link from node $N$ to $M$ then $N$ is parent of $M$
    - $M$ is child of $N$
  - leaf node has no children
    - internal node has 1 or 2 children
  - path is sequence of nodes, $N_0$, $N_1$, ..., $N_k$
    - $N_i$ is parent of $N_{i+1}$
    - sometimes edge instead of node
  - depth (level) of node: length of root-to-node path
    - level of root is 1 (measured in nodes)
  - height of node: length of longest node-to-leaf path
    - height of tree is height of root

Printing a search tree in order

- When is root printed?
  - After left subtree, before right subtree.

```
void visit(TreeNode t) {
    if (t != null) {
        visit(t.left);
        System.out.println(t.info);
        visit(t.right);
    }
}
```

- Inorder traversal
Tree traversals

- Different traversals useful in different contexts
  - Inorder prints search tree in order
    - Visit left-subtree, process root, visit right-subtree
  - Preorder useful for reading/writing trees
    - Process root, visit left-subtree, visit right-subtree
  - Postorder useful for destroying trees
    - Visit left-subtree, visit right-subtree, process root

```
if (t == null) t = new Tree(s,null,null);
```

Tree exercises

1. Build a tree
   - People standing up are nodes that are currently in the tree
   - Point at a sitting down person to make them your child
   - Is it a binary tree? Is it a BST?
   - Traversals, height, deepest leaf?
2. How many different binary search trees are there with specified elements?
   - E.g. given elements \( \{90, 13, 2, 3\} \), how many possible legal BSTs are there?
3. Convert a binary search tree to a doubly linked list in O(n) time without creating any new nodes.

Insertion and Find? Complexity?

- How do we search for a value in a tree, starting at root?
  - Can do this both iteratively and recursively, contrast to printing which is very difficult to do iteratively
  - How is insertion similar to search?

- What is complexity of print? Of insertion?
  - Is there a worst case for trees?
  - Do we use best case? Worst case? Average case?

- How do we define worst and average cases
  - For trees? For vectors? For linked lists? For arrays of linked-lists?

What does insertion look like?

- Simple recursive insertion into tree (accessed by root)
  - root = insert("foo", root);

```
public TreeNode insert(TreeNode t, String s) {
    if (t == null) t = new Tree(s,null,null);
    else if (s.compareTo(t.info) <= 0)
        t.left = insert(t.left,s);
    else  t.right = insert(t.right,s);
    return t;
}
```

- Note: in each recursive call, the parameter t in the called clone is either the left or right pointer of some node in the original tree
  - Why is this important?
  - Why must the idiom \( t = \text{treeMethod}(t,...) \) be used?

- What would removal look like?
Tree functions

- Compute height of a tree, what is complexity?
  ```java
  int height(Tree root) {
    if (root == null) return 0;
    else {
      return 1 + Math.max(height(root.left), height(root.right));
    }
  }
  ```

- Modify function to compute number of nodes in a tree, does complexity change?
  - What about computing number of leaf nodes?

Balanced Trees and Complexity

- A tree is height-balanced if
  - Left and right subtrees are height-balanced
  - Left and right heights differ by at most one
  ```java
  boolean isBalanced(Tree root) {
    if (root == null) return true;
    return isBalanced(root.left) && isBalanced(root.right) &&
    Math.abs(height(root.left) - height(root.right)) <= 1;
  }
  ```

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

What is complexity?

- Assume trees are “balanced” in analyzing complexity
  - Roughly half the nodes in each subtree
  - Leads to easier analysis

- How to develop recurrence relation?
  - What is T(n)?
  - What other work is done?

- How to solve recurrence relation
  - Plug, expand, plug, expand, find pattern
  - A real proof requires induction to verify correctness
Balanced 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
  - Standard java.util.TreeMap/TreeSet use red-black

Rotation to rebalance

```java
Node doLeft(Node root)
{
    Node newRoot = root.left;
    root.left = newRoot.right;
    newRoot.right = root;
    return newRoot;
}
```

Rotation up close (doLeft)

```java
Node doLeft(Node root)
{
    Node newRoot = root.left;
    root.left = newRoot.right;
    newRoot.right = root;
    return newRoot;
}
```

- 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 node

```java
Node doRight(Node root) {
    Node newRoot = root.right;
    root.right = newRoot.left;
    newRoot.left = root;
    return newRoot;
}
```

AVL tree practice

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

```
18
  10
    6
      3
      6
      12
```

Double rotation complete

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

```java
Node doRight(Node root) {
    Node newRoot = root.right;
    root.right = newRoot.left;
    newRoot.left = root;
    return newRoot;
}
```

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)

Scoreboard

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Insertion</th>
<th>Deletion</th>
<th>Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted Vector/array</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sorted vector/array</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linked list</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hash Maps</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary search tree</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AVL tree</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- What else might we want to do with a data structure?

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:
  - ← → ↑ ↓ ↵ ↘ ↖ ↙
- 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