Binary Trees

- Linked lists: efficient insertion/deletion, inefficient search
  - ArrayList: search can be efficient, insertion/deletion not

- Binary trees: efficient insertion, deletion, and search
  - trees used in many contexts, not just for searching, e.g., expression trees
  - search in $O(\log n)$ like sorted array
  - insertion/deletion $O(1)$ like list, only location found!
  - binary trees are inherently recursive, difficult to process non-recursively, but possible
    - recursion never required, often makes coding simpler

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, $N_1, N_2, \ldots, N_k$
    - $N_i$ 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

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?

A TreeNode by any other name...

- What does this look like?
  - What does the picture look like?

```java
public class TreeNode {
    TreeNode left;
    TreeNode right;
    String info;
    TreeNode(String s, TreeNode llink, TreeNode rlink) {
        info = s;
        left = llink;
        right = rlink;
    }
}
```
Printing a search tree in order

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

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

- Inorder traversal

```
llama
giraffe
pig
```

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 contains look like?

```java
public boolean contains(E element) {
    return myList.indexOf(element) >= 0;
}
```

```java
public boolean contains(E element){
    returns contains(myHead, element);
}
```

```java
private boolean contains(Node list, E element) {
    if (list == null) return false;
    if (list.info.equals(element)) return true;
    return contains(list.next, element);
}
```

- Why is there a private, helper method?
  - What will be different about Tree?
**What does contains look like?**

```java
public boolean contains(E element) {
    return contains(myRoot, element);
}
private boolean contains(TreeNode root, E element) {
    if (root == null) return false;
    if ((list.info.equals(element))) return true;
    if (element.compareTo(root.info) <= 0) {
        return contains(root.left, element);
    } else {
        return contains(root.right, element);
    }
}
```

- **What is recurrence? Complexity?**
  - When good trees go bad, how can this happen?

**What does insertion look like?**

```java
public boolean contains(TreeNode root, E element) {
    return contains(root, element);
}
private boolean contains(TreeNode root, E element) {
    if (root == null) return false;
    if (element.compareTo(root.info) <= 0) {
        return contains(root.left, element);
    } else {
        return contains(root.right, element);
    }
}
```

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

```java
public TreeNode insert(TreeNode t, String s) {
    if (t == null) t = new TreeNode(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 = treeMethod(t, ...)` be used?

**Removal from tree?**

- **For insertion we can use iteration (see BSTSet)**
  - Look below, either left or right
    - If null, stop and add
    - Otherwise go left when <=, else go right when >

- **Removal is tricky, depends on number of children**
  - Straightforward when zero or one child
  - Complicated when two children, find successor
    - See set code for complete cases
    - If right child, straightforward
    - Otherwise find node that’s left child of its parent (why?)

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

- Java implementation, similar to doubly-linked list
**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?

**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

**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;
}
```

**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
Searching, Maps, Tries (hashing)

- Searching is a fundamentally important operation
  - We want to search quickly, very very quickly
  - Consider searching using Google, ACES, issues?
  - In general we want to search in a collection for a key

- We've searched using trees and arrays
  - Tree implementation was quick: $O(\log n)$ worst/average?
  - Arrays: access is $O(1)$, search is slower

- If we compare keys, $\log n$ is best for searching n elements
  - Lower bound is $\Omega(\log n)$, provable
  - Hashing is $O(1)$ on average, not a contradiction, why?
  - Tries are $O(1)$ worst-case!! (ignoring length of key)

From Google to Maps

- If we wanted to write a search engine we'd need to access lots of pages and keep lots of data
  - Given a word, on what pages does it appear?
  - This is a map of words->web pages

- In general a map associates a key with a value
  - Look up the key in the map, get the value
  - Google: key is word/words, value is list of web pages
  - Anagram: key is string, value is words that are anagrams

- Interface issues
  - Lookup a key, return boolean: in map or value: associated with the key (what if key not in map?)
  - Insert a key/value pair into the map

Replacing Counter with Integer

- Key is a string, Value is # occurrences
  - Code below shows how Map interface/classes work

```java
while (it.hasNext()) {
    String s = it.next();
    Counter c = map.get(s);
    if (c != null) c.increment();
    else map.put(s, new Counter());
}
```

- What clues are there for prototype of map.get and map.put?
  - What if a key is not in map, what value returned?
  - What kind of objects can be put in a map?
Getting keys and values from a map

- Access every key in the map, then get the corresponding value
  - Get an iterator of the set of keys: `keySet().iterator()`
  - For each key returned by this iterator call `map.get(key)` ...

- Get an iterator over (key,value) pairs, there's a nested class called `Map.Entry` that the iterator returns, accessing the key and the value separately is then possible
  - To see all the pairs use `entrySet().iterator()`

External Iterator with generics

- Avoid `Object`, we know what we have a map of
  - Is the syntax worth it?

```java
Iterator<String> it = map.keySet().iterator();
while (it.hasNext()){    
    Counter value = it.next();
}
```

```java
Iterator<Map.Entry<String,Counter>> it2 = map.entrySet().iterator();
while (it2.hasNext()){   
    Map.Entry<String,Counter> me = it.next();
    Counter value = me.getValue();
}
```

External Iterator without generics

- The `Iterator` interface access elements
  - Source of iterator makes a difference: cast required?

```java
Iterator it = map.keySet().iterator();
while (it.hasNext()){    
    Object value = map.get(it.next());
}
```

```java
Iterator it2 = map.entrySet().iterator();
while (it2.hasNext()){   
    Map.Entry me = (Map.Entry) it.next();
    Object value = me.getValue();
}
```

Hashing: Log (10^{100}) is a big number

- Comparison based searches are too slow for lots of data
  - How many comparisons needed for a billion elements?
  - What if one billion web-pages indexed?

- Hashing is a search method: average case O(1) search
  - Worst case is very bad, but in practice hashing is good
  - Associate a number with every key, use the number to store the key
    - Like catalog in library, given book title, find the book

- A hash function generates the number from the key
  - Goal: Efficient to calculate
  - Goal: Distributes keys evenly in hash table
Hashing problems

- Linear probing, $hash(x) = x \mod \text{tablesize}$
  - Insert 24, 12, 45, 14, delete 24, insert 23 (where?)
  - Same numbers, use quadratic probing (clustering better?)
- What about chaining, what happens?

Chaining with hashing

- With n buckets each bucket stores linked list
  - Compute hash value $h$, look up key in linked list at $h$
  - Hopefully linked lists are short, searching is fast
  - Unsuccessful searches often faster than successful
    - Empty linked lists searched more quickly than non-empty
  - Potential problems?
- Hash table details
  - Size of hash table should be a prime number
  - Keep load factor small: number of keys/size of table
  - On average, with reasonable load factor, search is $O(1)$
  - What if load factor gets too high? Rehash or other method

What about hash functions

- Hashing often done on strings, consider two alternatives
  - Consider $total = (k+1)\times s.charAt(k)$, why might this be better?
  - Other functions used, always mod result by table size
- What about hashing other objects?
  - Need conversion of key to index, not always simple
  - Ever object has method $hashCode()$!
**Trie: efficient search words/suffixes**

- A trie (from retrieval, but pronounced “try”) supports
  - Insertion: put string into 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
  - Very useful matching suffixes: suffix tree/suffix array

**RSS Aggregator**

- **What is RSS (Really Simple Syndication)?**
  - What problem does it try to solve?
- **We want to retrieve and index news articles**
  - How will we search
- **What does an item have in it?**

```xml
<item>
  <title>Medicare Change Will Limit Access to Claim Hearing</title>
  <description>Medicare beneficiaries must now show special circumstances to appear in person before a judge when their claims are denied.</description>
  <author>By ROBERT PEAR</author>
  <pubDate>Sun, 24 Apr 2005 00:00:00 EDT</pubDate>
</item>
```

**What about union and intersection, iteration?**

- Indicates word ends here

**What does RSS look like?**

```xml
<?xml version="1.0" encoding="iso-8859-1"?>
<rss version="2.0">
  <channel>
    <title>NYT &gt; Home Page</title>
    <description>NYT: Breaking News</description>
    ...<item>
      <item node 1
    </item>
    ...</channel>
  </rss>
```