The Master Theorem

- If recurrence has the form
  \[ T(n) = \begin{cases} \Theta(1) & n = 1 \\ aT(n/b) + f(n) & n > 1 \end{cases} \]

- The **Master Theorem** tells us how to analyze it:
  - If \( f \in O(n^\log_b a - \varepsilon) \), for some constant \( \varepsilon > 0 \), then \( T \in \Theta(n^{\log_b a}) \).
  - If \( f \in \Theta(n^{\log_b a}) \), then \( T \in \Theta(n^{\log_b a \lg n}) \).
  - If \( f \in \Omega(n^{\log_b a + \varepsilon}) \), for some constant \( \varepsilon > 0 \), and if \( af(n/b) \leq cf(n) \) for some constant \( c < 1 \) and \( n > n_0 \), then \( T \in \Theta(f) \).

Some More Examples...

- Here we will do a couple examples of the master theorem
- Also I will show you a little trick (substitution) that can come in handy – especially if you have \( \sqrt{\cdot} \):

Not Fun!

- Homework 2.2-1: and **prove** that it has that form.
- Do all of 4.1
Fun!

Simple Sorting Algorithms:
- **Merge Sort:** Divide the list into smaller pieces. Sort the small pieces. Then merge together sorted lists.
- **Insertion Sort:** Insert item $j$ into $A[0\ldots j-1]$
- **Selection Sort:** Find $j$th smallest element and put it in $A[j]$
- **Bubble sort:** Start at end of array: If $A[j] < A[j-1]$, swap them

(A subset of) the Collections Interface

```java
public interface Collection<E> extends Iterable<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element); //optional
    boolean remove(Object element); //optional
    Iterator<E> iterator();

    // Array operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

The Java Collections Interfaces

- In the remainder of the class, we will be using the Java Collections Interface: [http://java.sun.com/docs/books/tutorial/collections/TOC.html](http://java.sun.com/docs/books/tutorial/collections/TOC.html)
- **Important:** Most of what I will say only works if you set the "code level" to Java 5.0 in eclipse!
- The interfaces form a hierarchy:

Traversing Collections

- Use for-each construct:
  ```java
  for (Object o : collection)
      System.out.println(o);
  ```
- Use an iterator. (Can remove() items with iterator
  ```java
  Iterator<String> i = words.iterator();
  while (i.hasNext()) {
      System.out.println(i.next());
  }
  ```
- hasNext(): returns true if the iteration has more elements,
- next(): returns the next element in the iteration.
- remove(): removes the last element that was returned by next from the underlying Collection.
Converting to Array

- Sometimes you need to convert a collection to an array:
  ```java
  Object[] a = c.toArray();
  ```

Set

- A Set is a Collection that cannot contain duplicate elements.
- It models the mathematical set abstraction.
- The Set interface contains only methods inherited from Collection and adds the restriction that duplicate elements are prohibited.
- Set is still an interface. There are 3 implementations of Set in Java:
  - HashSet
  - TreeSet
  - LinkedHashSet

Hash?

- No, Cheech. A hash table is a data structure in which we can "look up" (or search) for an element efficiently.
- The expected time to search for an element in a hash table in $O(1)$. (Worst case time in $\Theta(n)$).
- Think of a hash table as an array
- With a regular array, we find the element whose "key" is $j$ in position $j$ of the array. $j = 17$; `val = a[j];`
- This is called direct addressing and it takes $O(1)$ on your regular ol' random access computer.
- This form of direct addressing works when we can afford to have an array with one position for every possible key

More on Hash

- In a hash table the number of keys stored is small relative to the number of possible keys
- A hash table is an array. Given a key $k$, we don’t use $k$ as the index into the array – rather, we have a hash function $h$, and we use $h(k)$ as an index into the array.
- Given a “universe” of keys $K$.
  - Think of $K$ as all the words in a dictionary, for example
  - $h : K \rightarrow \{0, 1, \ldots m – 1\}$, so that $h(k)$ gets mapped to an integer between 0 and $m – 1$ for every $k \in K$
- We say that $k$ hashes to $h(k)$
Example

- This looks great. However, what happens if \( h(k_1) = h(k_2) \) for \( k_1 \neq k_2 \)?
- Two keys hash to the same value. The element collide
- This is typically handled by chaining
- Instead of storing a key \( k \) (or later key value pair \( (k, v) \)) at every position in the array, we store a linked list of keys.

A (Fairly) Obvious point

- BAD hash function. \( h(k) = 3 \).
- If all keys hash to the same value, then looking up a key takes \( \Theta(n) \). (Since it is just a list).
- We would like a hash function to be “random” in the sense that a key \( k \) is equally likely to have into any of the \( m \) slots in the hash table (array).
- If have have such a function, then we can show that the time required to search for a key is \( \Theta(1 + \frac{n}{m}) \).
- When hashing keys that are not numbers, you must convert them to numbers.

\[
\text{BEER} = -142 + 2^4 + 5^3 + 5^2 + 18^1 = 42.
\]

Back to the Java Collections

- So now you know what a Java HashSet is.
- A LinkedHashSet is a HashSet that also keeps track of the order in which elements were inserted.
- (Think of laying a linked list on top of the Hash Table)
- A TreeSet stores its elements in a red-black tree.
- In order to understand red-black trees, we must know about binary search trees.
- Hash table is “good” at \texttt{insert()}, \texttt{search()}, \texttt{delete()}.
  But what if you also want to support (efficiently) \texttt{minimum()}, \texttt{maximum()}

Binary Search Tree

- A \textit{binary search tree} is a data structure that is conceptualized as a binary tree. (Have you read Appendix B-4 yet?)
- Each node in the tree contains:
  - key \( k \). (Or maybe (key, value): \( (k, v) \))
  - \texttt{left} \( l \): Points to the left child
  - \texttt{right} \( r \): Points to the right child
  - parent \( p \): Points to the parent

Binary Search Tree Property

\[\text{If } y \text{ is in the left subtree of } x, \text{ then } k(y) \leq k(x)\]
Binary Search Trees

- There are lots of binary trees that can satisfy this property.
- It is obvious that the number of binary tree on $n$ nodes $b_n$ is

$$b_n = \frac{1}{n+1} \binom{2n}{n}$$

$$b_n = \frac{4^n}{\sqrt{\pi n^{3/2}}} (1 + O(1/n))$$

- And not all of these (exponentially many) are created equal.
- In fact, we would like to keep our binary search trees “short”, because most of the operations we would like to support are a function of the height $h$ of the tree.

Short Is Beautiful

- SEARCH() takes $O(h)$
- MINIMUM(), MAXIMUM() also take $O(h)$
- Slightly less obvious is that INSERT(), DELETE() also take $O(h)$
- Thus we would like to keep out binary search trees “short” ($h$ is small).

red-black Trees

- red-black trees are simply a way to keep binary search trees short. (Or balanced)
- Balanced here means that no path on the tree is more than twice as long as another path.
- An implication of this is that its maximum height is $2 \lg(n+1)$
- SEARCH(), MINIMUM(), MAXIMUM(), all take $O(\lg n)$
- It’s implementation is complicated, so we won’t cover it
- INSERT(): also runs in $O \lg(n)$
- DELETE(): runs in $O \lg(n)$
  - (but it is more complicated to maintain the “red-black” property)

Next Time?

- More on data structures and Java collections
- The greatest lab ever

Small News

Let’s have a little quiz on 2/7