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
- **Important:** Most of what I will say only works if you set the “code level” to Java 5.0 in eclipse!
- **Preferences, Java Compiler:** Set this to ≥ 5.0
- The interfaces form a hierarchy:

```
(A subset of) the Collections Interface

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);
}
```
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 is \( O(1) \). (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. \( j = 17; \text{val} = a[j]; \).
- 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
    \[ h : K \to \{0, 1, \ldots, m - 1\} \]
  - 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 look 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 elements 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.
- Example:
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 hash into any of the \( m \) slots in the hash table (array).
- If we have such a function, then we can show that the average 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, e.g.:
  \[
  \text{BEER} = -142 + 2^4 + 5^3 + 5^2 + 18^1 = 42.
  \]

Average Hash Search Time

- The number of elements to be searched is 1 more than the number of elements that appear before \( x \) in \( x’ \)’s list. Assuming we insert items into the list at the beginning, then this is the number of elements that were inserted after \( x \).
- By definition: \( \mathbb{P}(h(k_i) = h(k_j)) = \frac{1}{m} \)
- Let \( X_{ij} \) be indicator random variable that is equal to one if and only if \( h(k_i) = h(k_j) \)
- Then just compute:
  \[
  \mathbb{E} \left[ \frac{1}{n} \sum_{i=1}^{n} \left( 1 + \sum_{j=i+1}^{n} X_{ij} \right) \right].
  \]

Hash Functions

Modular Hash Function

- Let \( m \) be (roughly) the size of your hash table:
  \[ h(k) = k \mod m \]
- Good choice of \( m \): A prime number not too close to an exact power of 2

Multiplicative Hash Function

- \( h(k) = \lfloor m(kA \mod 1) \rfloor \)
- Multiply key \( k \) by \( A \), take fractional part, and multiply by \( m \)
- If \( m = 2^p \) this can be done very fast with bit shifting
- \( A \approx \phi = (\sqrt{5} - 1)/2 \) seems a good value

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 an alerted-black tree.
- In order to understand red-black trees, we must know about binary search trees.
- Hash table is “good” at insert(), search(), delete(). But what if you also want to support (efficiently) minimum(), maximum()
Trees

- A tree is a set of items organized into a hierarchical structure (think of a family tree).
- When organized in this way, we call the items nodes.
- Each node has a single designated parent and one or more children.
- There is a single designated node, called the root, with no parent.
- Any node with no children is called a leaf.
- Any node with children is called internal.
- A tree in which all nodes have 2 or fewer children is called a binary tree.
- Storing a list of items in a tree structure allows us to represent additional relationships among the items in the list.

Binary Tree Data Structures

- To store a tree of keys $k$, or maybe (key, value) pairs: $(k, v)$, we need a data structure supporting three basic operations:
  - left $l$: Points to the left child
  - right $r$: Points to the right child
  - parent $p$: Points to the parent
- This allows us to traverse the tree and perform other operations on it.
- The level of a node in the tree is the number of recursive calls to parent() needed to reach the root.
- The depth of the tree is the maximum level of any of its nodes.
- A balanced tree is one in which all leaves are at levels $k$ or $k - 1$, where $k$ is the depth of the tree.

Data Structures for Storing Trees

Array

- The root is stored in position 0.
- The children of the node in position $i$ are stored in positions $2i + 1$ and $2i + 2$.
- This determines a unique storage location for every node in the tree and makes it easy to find a node’s parent and children.
- Using an array, the basic operations can be performed very efficiently.
- If the tree is unbalanced or dynamic, a linked list may be better.

Linked List

- In a linked list, each item is stored along with explicit pointers to its parent and children.
- This allows for easy addition and deletion of nodes from the tree.
A binary search tree is a data structure that is conceptualized as a binary tree, but has one additional property:

**Binary Search Tree Property**

If $y$ is in the left subtree of $x$, then $k(y) \leq k(x)$

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

$$b_n = \frac{1}{n+1} \left( \frac{2n}{n} \right)$$

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

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).

**Operations**

- **SUCCESSOR(x)**
  - How would I know “next biggest” element?
  - If right subtree is not empty: **MINIMUM(r(x))**
  - If right subtree is empty: Walk up tree until you make the first “right” move

- **INSERT(x)**
  - Just walk down the tree and put it in. It will go “at the bottom”
DELETE()

- If 0 or 1 child, deletion is fairly easy
- If 2 children, deletion is made easier by the following fact:

**Binary Search Tree Property**

- If a node has 2 children, then
  - its successor will not have a left child
  - its predecessor will not have a right child

**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 \(2 \log(n+1)\)
- \(\text{SEARCH()}, \text{MINIMUM()}, \text{MAXIMUM}(),\) all take \(O(\log n)\)
- It’s implementation is complicated, so we won’t cover it
- \(\text{INSERT}()\): also runs in \(O(\log n)\)
- \(\text{DELETE}()\): runs in \(O(\log n)\)
  - (but it is more complicated to maintain the “red-black” property)

**Back to the Java Collections**

- red-black trees remain sorted
- You don’t really have any control over the order in which things will appear in a HashSet
- If you care about that – you should use a LinkedHashSet, which lays a linked list on top of the HashSet
- In general, Sets are not for ordered collections of items, for that, you should use a list

**Lists**

- A List is an ordered Collection (sometimes called a sequence).
- Lists may contain duplicate elements.
- In addition to the operations inherited from Collection, the List interface includes operations for the following:
  - Positional access: manipulate elements based on their numerical position in the list
  - Search: searches for a specified object in the list and returns its numerical position
Divide-And-Conquer
Data Structures
Java Collections

(Subset of) List Interface

```java
public interface List<E> extends Collection<E> {
    // Positional access
    E get(int index);
    E set(int index, E element); //optional
    boolean add(E element); //optional
    void add(int index, E element); //optional
    E remove(int index); //optional

    // Search
    int indexOf(Object o);
    int lastIndexOf(Object o);

    // Iteration
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);
}
```

Java List Implementations

### Two List Implementations

- **ArrayList:** which is usually the better-performing
- **LinkedList:** offers better performance under certain circumstances, (i.e. if lots of add/remove in the middle if the list)

Java Lists have extended iterators

```java
public interface ListIterator<E> extends Iterator<E> {
    boolean hasNext();
    E next();
    boolean hasPrevious();
    E previous();
    int nextIndex();
    int previousIndex();
    void remove(); //optional
    void set(E e); //optional
    void add(E e); //optional
}
```

List Stuff

- **ListIterator<E> listIterator():** gives iterator at beginning
- **ListIterator<E> listIterator(int index):** gives iterator at specified index
- The index refers to the element that would be returned by an initial call to `next()`
- The cursor is always between two elements:
  - the one that would be returned by a call to `previous()`
  - the one that would be returned by a call to `next()`
- The \( n + 1 \) valid index values correspond to the \( n + 1 \) gaps between elements, from the gap before the first element to the gap after the last one.
Next Time

- A bit on Java Collection Map Interface
- Move on to Heaps (Chapter 6)
- We have covered chapters 1-4, 10-11, and Appendices A and B

News

- New Homework Posted!
- Let's have a little quiz on 2/7
- Homework is due 2/5: No late homework accepted. (I need to hand out solutions and discuss in class on 2/5).