CMPSCI 187: Programming With Data Structures

Lecture 33: Hash Tables in Collections
5 December 2011

Hash Tables in Collections

- Sets and Maps
- `SetADT` in L&C and the `Set` Interface
- The `Map` Interface
- The `HashSet` Class
- The `HashMap` Class
- The Other Hash Table Classes

Sets and Maps

- If we are keeping a collection of items in a data structure, we have three basic operations on it -- **inserting** a new item, **deleting** an item that is already there, and **finding** whether a particular item is there.
- These are the basic operations of a **set**, which in mathematics is a collection with no defined order and in which no item can occur more than once.
- If we are storing a large amount of data, where items are large, the finding operation in particular is not just “is item x there” but “is any item there with these characteristics”, for example “is there an employee with this name”. A **map** is a data structure with two types, **keys** and **values**. We can think of the map as a set of key-value pairs, or as a **function** that takes a key as input and returns the value as output.
- Both binary search trees and hash tables can be used for both sets and maps.

`SetADT` in L&C and the `Set` Interface

- L&C's interface, below, has the basic `add`, `remove`, and `contains` methods and a method to remove a *random* element of the set. We'll look at implementations of these methods next lecture.

- The `Set` interface in the Collections framework extends `Collection` and has most of the same methods, but not `removeRandom`. The removing methods are optional because we may want to have a read-only set.

```
public interface SetADT<T> {
   public void add (T element);
   public T removeRandom ( );
   public T remove (T element);
   public SetADT<T> union (SetADT<T> set);
   public boolean contains (T target);
   public boolean equals (SetADT<T> set);
   public boolean isEmpty( );
   public int size( );
   public Iterator<T> iterator( );
   public String toString( );
```

The `Map` Interface

- L&C don't say much about this interface, but it is Java's primary way to arrange a database. Map<K, V> is a generic interface with *two* type variables, one for the keys and one for the values. A key may only belong to the Map once, with one value associated to it.

- The two most important methods are `public V get (Object key)`, which returns the value for that key or `null` if the key is not there, and `public V put (K key, V value)`, which assigns `value` as the new value associated to `key` and returns the old one if there was any. The `put` method is optional in case we want a read-only map.

- We can also get the Set of keys in the Map, a Collection of the values, and work with the pairs as a separate type. We also have two `contains` methods, one for keys and one for values.

The `HashSet` Class

- A `HashSet` object has a **load factor** `f` and a **capacity** `c` -- its base type `T` should have a `hashcode` method. Entry `e` would go in the **bucket** for `e.hashcode( ) % c`, and the bucket is maintained as a linked list.

- The iterator operates by running down the array of linked lists, so it returns the entries in no particular order. The table resizes if the size exceeds the load factor times the capacity.

```
public class HashSet<T> extends AbstractSet implements Set {
// constructors can take Collection, set load factor, capacity
   public boolean add (T e) { }
   public void clear ( ) { }
   public Object clone ( ) { } // makes shallow copy
   public boolean contains (Object o) { }
   public boolean isEmpty( ) { }
   public Iterator<T> iterator( ) { }
   public boolean remove (Object o) { } // returns whether there
   public boolean size( ) { }}
```

The `HashMap` Class

- This also has a load factor and capacity, and hashes using `e.hashcode( ) % c` and has a linked list for each bucket.
- `Map.Entry` is a **nested class**, which we don't want to think about -- an object of that class is a single key-value pair.

```
public class HashMap<K, V> extends AbstractMap<K, V> {
// constructor can take a Map, set capacity and load factor
   public Object clone( ) { } // shallow copy
   public boolean containsKey (Object key) { }
   public boolean containsValue (Object value) { }
   public Set<Map.Entry<K,V>> entrySet( ) { }
   public V get (Object key) { } // returns null if no value
   public Set<K> keySet( ) { }
   public V put (K key, V value) { }
   public void putAll (Map<? extends K, ? extends V> m) { }
   public V remove (Object key) { }
   public Collection<V> values( ) { }
```

The Other Hash Table Classes

- `HashTable` was the first draft of `HashMap` and is now a legacy class. It does not allow `null` values and is synchronized.

- `WeakHashMap` has **weak keys**, which are deleted by the garbage collector unless they are used outside the hash table.

- `IdentityHashMap` compares keys by == instead of `.equals`, so you could have two different keys with the same value in type `K`.

- `LinkedHashSet` and `LinkedHashMap` extend `HashSet` and `HashMap` respectively, but also keep their entries (set) or values (map) in a linked list so that their iterators return their elements in a set order such as first-added first or last-accessed first. This is good if you need to retain the order of the elements as you use them in a hash table.