CS 61B Homework 6
Due 3pm Friday, March 18, 2005

This homework will teach you about hash tables, hash codes, and compression functions. This is an individual assignment; you may not share code with other students.

Copy the Homework 6 directory by doing the following, starting from your home directory. Don't forget the "-r" switch in the cp command.

```
mkdir hw6
cd hw6
cp -r $master/hw/hw6/* .
```

Part I (6 points)

Implement a class called HashTableChained, a hash table with chaining. HashTableChained implements an interface called Dictionary, which defines the set of methods (like insert(), find() and remove()) that a dictionary needs. Both files appear in the "dict" package.

The methods you will implement are a subset of those listed on page 376 of Goodrich and Tamassia (no iterators are used in this assignment), plus a makeEmpty() method which removes every entry from a hash table. There are also two HashTableChained constructors. One lets applications specify an estimate of the number of entries that will be stored in the hash table; the other uses a default size. Both constructors should create a hash table that uses a prime number of buckets. (Several methods for identifying prime numbers were discussed early in the semester.) In the first constructor, shoot for a load factor between 0.5 and 1. In the second constructor, shoot for around 100 buckets. Descriptions of all the methods may be found in Dictionary.java and HashTableChained.java.

Do not change Dictionary.java. Do not change any prototypes in HashTableChained.java, or throw any checked exceptions. Most of your solution should appear in HashTableChained.java, but other classes are permitted. You will probably want to use a linked list code, of your choice. (Note that even though the hash table is in the "dict" package, it can still use linked list code in a separate "list" package. There's no need to move the list code into the "dict" package.)

Look up the hashCode method in the java.lang.Object API. Assume that the objects used as keys to your hash table have a hashCode() method that returns a "good" hash code between Integer.MIN_VALUE and Integer.MAX_VALUE (that is, between -2147483648 and 2147483647). Your hash table should use a compression function, as described in Section 8.2.4 of Goodrich and Tamassia, to map each key's hash code to a bucket of the table. Your compression function should be computed by the compFunction() helper method in HashTableChained.java (which has "package" protection so we can test it). insert(), find(), and remove() should all use this compFunction() method.

The methods find() and remove() should return (and in the latter case, remove) an entry whose key is equals() to the parameter "key". Reference equality is NOT required for a match.

Part II (4 points)

It is often useful to hash data structures other than strings or integers. For example, game tree search can sometimes be sped by saving game boards and their evaluation functions, so that if the same game board can be reached by several different sequences of moves, it will only have to be evaluated once. For this application each game board is a key, and the value returned by the evaluation function is the value stored alongside the key in the hash table. If we search the same game board again, we can look up its evaluation function in the dictionary, so we won't have to calculate it twice.

The class SimpleBoard represents an 8x8 checkerboard. Each position has one of three values: 0, 1, or 2. Your job is to fill in two missing methods: equals() and hashCode(). The equals() operation should be true whenever the boards have the same pieces in the same locations. The hashCode() function should satisfy the specifications described in the java.lang.Object API. In particular, if two SimpleBoards are equals(), they have the same hash code.

You will be graded on how "good" your hash code and compression function are. By "good" we mean that, regardless of the table size, the hash code and compression function evenly distribute SimpleBoards throughout the hash table. Your solution will be graded in part on how well it distributes a set of randomly constructed boards. Hence, the sum of all the cells is not a good hash code, because it does not change if cells are swapped. The product of all cells is even worse, because it's usually zero. What's better? One idea is to think of each cell as a digit of a base-3 number (with 64 digits), and convert that base-3 number to a single int. (Be careful not to use floating-point numbers for this purpose, because they roundoff the least significant digits, which is the opposite of what you want.)

Do not change any prototypes in SimpleBoard.java, or throw any checked exceptions. The file Homework6Test.java is provided to help you test your HashTableChained and your SimpleBoard together. Note that Homework6Test.java does NOT test all the methods of HashTableChained; you should write additional tests of your own. Moreover, you will need to write a test to see if your hash code is doing a good job of distributing SimpleBoards evenly through the table.

Submitting your solution

Change (cd) to your hw6 directory, which should contain SimpleBoard.java and the dict directory (and optionally a list directory). The dict directory should contain HashTableChained.java and any other .java files it uses (except those in the list package). You're not allowed to change Dictionary.java, so the 'submit' program won't take it; nor will it take Homework6Test.java (though you can change it as much as you like).

Make sure that your submission compile and runs on the _lab_ machines. From your hw6 directory, type "submit hw6". (Note that "submit" will not work if you are inside the dict directory!) After submitting, if you realize your solution is flawed, you may fix it and submit again. You may submit as often as you like. Only the last version you submit before the deadline will be graded.