```
                         CS 61B  Homework 6
                    Due 5pm Sunday, March 22, 2009
```

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.

    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 389 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 lecture, 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.

Compression functions

Besides the lecture notes, compression functions are also covered in Section
9.2.4 of Goodrich and Tamassia. Unfortunately, they make the erroneous claim
that for a hash code i and an N-bucket hash table,

   h(i) = |ai + b| mod N

is "a more sophisticated compression function" than

   h(i) = |i| mod N.

Actually, the "more sophisticated" function causes _exactly_ the same
collisions as the less sophisticated compression function; it just shuffles the
buckets to different indices. The better compression function, as I said in
lecture, is

   h(i) = ((ai + b) mod p) mod N,

where p is a large prime that's substantially bigger than N. (You can replace
the parentheses with absolute values if you like; it doesn't matter much.)

For this homework, the simplest compression function might suffice. The bottom
line is whether you have too many collisions or not in Part II. If so, you'll
need to improve your hash code or compression function or both.

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 minimax
algorithm is the value stored alongside the key in the hash table. If our
search encounters the same game board again, we can look up its value in the
dictionary, so we won't have to calculate the value 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 round off the least significant digits,
which is the opposite of what you want. Better to round off the most
significant digits, which is what happens when an int gets too big.)

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. Our autograder will do extensive tests on that.

A tutorial on collision probability

Students are always surprised when they find out how many collisions occur in
a working hash table. You might have the misimpression that there won't be
many collisions at all until the table is nearly full. Let's analyze how many
collisions you should expect to see if your hash code and compression function
are good.

If you have N buckets and a good (pseudorandom) hash function, the probability
of any two keys colliding is 1/N. So when you have i keys in the table and
insert key i + 1, the probability that the new key does NOT collide with any
old key is $(1 - 1/N)^i$. If you insert n distinct items, the expected number
that WON'T collide is

  n-1
  sum $(1 - 1/N)^i = N - N (1 - 1/N)^n$,
  i=0

so the expected number of collisions is

$n - N + N (1 - 1/N)^n$.

Now, for any n and N you test, you can just plug them into this formula and see
if the number of collisions you're getting is in the ballpark of what you
should expect to get. For example, if you have N = 100 buckets and n = 100
keys, expect about 36.6 collisions.

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 or
Entry.java, so the "submit" program won't take them; nor will it take
Homework6Test.java (which you can change as much as you like).

Make sure that your submission compiles 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 or list 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.