CS 61B: Lecture 22
Friday, March 15, 2013

Today's reading:  Goodrich & Tamassia, Sections 9.1, 9.2, 9.5-9.5.1.

DICTIONARIES
===========
Suppose you have a set of two-letter words and their definitions.  You want to
be able to look up the definition of any word, very quickly.  The two-letter
word is the _key_ that addresses the definition.

Since there are 26 English letters, there are 26 * 26 = 676 possible two-letter
words.  To implement a dictionary, we declare an array of 676 references, all
initially set to null.  To insert a Definition into the dictionary, we define
a function hashCode() that maps each two-letter word (key) to a unique integer
between 0 and 675.  We use this integer as an index into the array, and make
the corresponding bucket (array position) point to the Definition object.

```
public class Word {
  public static final int LETTERS = 26, WORDS = LETTERS * LETTERS;
  public String word;

  public int hashCode() {
    return LETTERS * (word.charAt(0) - 'a') + (word.charAt(1) - 'a');
  }
}

public class WordDictionary {
  private Definition[] defTable = new Definition[Word.WORDS];

  public void insert(Word w, Definition d) {
    defTable[w.hashCode()] = d;
  }

  Definition find(Word w) {
    return defTable[w.hashCode()];
  }
}
```

What if we want to store every English word, not just the two-letter words?
The table "defTable" must be long enough to accommodate
pneumonoultramicroscopicsilicovolcanoconiosis, 45 letters long.  Unfortunately,
declaring an array of length 26^45 is out of the question.  English has fewer
than one million words, so we should be able to do better.

Hash Tables (the most common implementation of dictionaries)
-----------
Suppose n is the number of keys (words) whose definitions we want to store, and
suppose we use a table of N buckets, where N is perhaps a bit larger than n,
but much smaller than the number of _possible_ keys.  A hash table maps a huge
set of possible keys into N buckets by applying a _compression_function_ to
each hash code.  The obvious compression function is

  h(hashCode) = hashCode mod N.

Hash codes are often negative, so remember that mod is not the same as Java's
remainder operator "%".  If you compute hashCode % N, check if the result is
negative, and add N if it is.

With this compression function, no matter how long and variegated the keys are,
we can map them into a table whose size is not much greater than the actual
number of entries we want to store.  However, we've created a new problem:
several keys are hashed to the same bucket in the table if h(hashCode1) =
h(hashCode2).  This circumstance is called a _collision_.

How do we handle collisions without losing entries?  We use a simple idea
called _chaining_.  Instead of having each bucket in the table reference one
entry, we have it reference a linked list of entries, called a _chain_.  If
several keys are mapped to the same bucket, their definitions all reside in
that bucket's linked list.

Chaining creates a second problem:  how do we know which definition corresponds
to which word?  The answer is that we must store each key in the table with its
definition.  The easiest way to do this is to have each listnode store an
_entry_ that has references to both a key (the word) and an associated value
(its definition).

```
          ---   -------------------------------------------------------
defTable |.+-->|   .   |   .   |   X   |   .   |   X   |   .   |   .   | ...
          ---   ----|-------|---------------|---------------|-------|-----
                    v       v               v               v       v
                   ---     ---             ---             ---     ---
                  |.+>pus  |.+>swirl       |.+>tough       |.+>cool|.+>mud
                  |.+>goo  |.+>vortex      |.+>too bad     |.+>jrs |.+>wet dirt
                  |.|      |X|             |X|             |.|     |X|
                  -+-      ---             ---             -+-     ---
                   |                                        |
                   v                                        v
                  ---                 ^                    ---
                  |.+>sin          < chains >             |.+>gigantic
                  |.+>have fun                            |.+>very big
                  |X|                                     |X|
                  ---                                     ---
```

Hash tables usually support at least three operations.  An Entry object
references a key and its associated value.

```
public Entry insert(key, value)
  Compute the key's hash code and compress it to determine the entry's bucket.
  Insert the entry (key and value together) into that bucket's list.
public Entry find(key)
  Hash the key to determine its bucket.  Search the list for an entry with the
  given key.  If found, return the entry; otherwise, return null.
public Entry remove(key)
  Hash the key to determine its bucket.  Search the list for an entry with the
  given key.  Remove it from the list if found.  Return the entry or null.
```

What if two entries with the same key are inserted?  There are two approaches.
Following Goodrich and Tamassia, we can insert both, and have find() or
remove() arbitrarily return/remove one.  Goodrich and Tamassia also propose
a method named findAll that returns all the entries with a given key.  The
other approach is to replace the old value with the new one, so only one entry
with a given key exists in the table.  Which approach is best?  It depends on
the application.

WARNING:  When an object is stored in a hash table, an application should never
change the object in a way that will change its hash code.  If you do so, the
object will thenceforth be in the wrong bucket.

The _load_factor_ of a hash table is n/N, where n is the number of keys in the
table and N is the number of buckets.  If the load factor stays below one (or
a small constant), and the hash code and compression function are "good," and
there are no duplicate keys, then the linked lists are all short, and each
operation takes O(1) time.  However, if the load factor grows too large
(n >> N), performance is dominated by linked list operations and degenerates to
O(n) time (albeit with a much smaller constant factor than if you replaced the
hash table with one singly-linked list).  A proper analysis requires a little
probability theory, so we'll put it off until near the end of the semester.

Hash Codes and Compression Functions
-----------------------------------
Hash codes and compression functions are a bit of a black art.  The ideal hash
code and compression function would map each key to a uniformly distributed
random bucket from zero to N - 1.  (By "random", I don't mean that the function
is different each time; a given key always hashes to the same bucket.  I mean
that two different keys, however similar, will hash to independently chosen
integers, so the probability they'll collide is 1/N.)  This ideal is tricky to
obtain.

In practice, it's easy to mess up and create far more collisions than
necessary.  Let's consider bad compression functions first.  Suppose the keys
are integers, and each integer's hash code is itself, so hashCode(i) = i.

Suppose we use the compression function h(hashCode) = hashCode mod N, and the
number N of buckets is 10,000.  Suppose for some reason that our application
only ever generates keys that are divisible by 4.  A number divisible by 4 mod
10,000 is still a number divisible by 4, so three quarters of the buckets are
never used!  Thus the average bucket has about four times as many entries as it
ought to.

The same compression function is much better if N is prime.  With N prime, even
if the hash codes are always divisible by 4, numbers larger than N often hash
to buckets not divisible by 4, so all the buckets can be used.

For reasons I won't explain (see Goodrich and Tamassia Section 9.2.4 if you're
interested),

  h(hashCode) = ((a * hashCode + b) mod p) mod N

is a yet better compression function.  Here, a, b, and p are positive integers,
p is a large prime, and p >> N.  Now, the number N of buckets doesn't need to
be prime.

I recommend always using a known good compression function like the two above.
Unfortunately, it's still possible to mess up by inventing a hash code that
creates lots of conflicts even before the compression function is used.  We'll
discuss hash codes next lecture.