

CS 61B: Lecture 23
Monday, March 16, 2009

Today's reading: Goodrich & Tamassia, Sections 9.1-9.3.

Hash Tables (continued)

Reminder: n is the number of keys (words) whose definitions we want to store, and we use a table of N buckets. The compression function is

$$h(\text{hashCode}) = \text{hashCode} \bmod N.$$

Several keys are hashed to the same bucket in the table if $h(\text{hashCode1}) = h(\text{hashCode2})$. 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 |.+>sucks 2BU |.+>jrs |.+>wet dirt
          |.|   |X|         |X|         |.|   |X|
          +-+   ---       ---       +-+   ---
          |           |           |           |
          v           ^           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. I'm not
    sure why, but Goodrich & Tamassia's interface also returns the Entry object.
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 ($n \gg N$), 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.

Compression Functions

Recall that we map a key to a bucket in two steps:

$$\text{key} \rightarrow \text{hash code} \rightarrow [0, N - 1].$$

I'll discuss the second half of this map, the compression function, first.

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 $\text{hashCode}(i) = i$.

Suppose we use the compression function $h(\text{hashCode}) = \text{hashCode} \bmod N$, 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! We have at least four times as many collisions as necessary.

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),

$$h(\text{hashCode}) = ((a * \text{hashCode} + b) \bmod p) \bmod N$$

is a yet better compression function. Here, a , b , and p are positive integers, p is a large prime, and $p \gg N$. 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.

Hash Codes

 Since hash codes often need to be designed specially for each new object, you're left to your own wits. Here is an example of a good hash code for Strings.

```
private static int hashCode(String key) {
    int hashVal = 0;
    for (int i = 0; i < key.length(); i++) {
        hashVal = (127 * hashVal + key.charAt(i)) % 16908799;
    }
    return hashVal;
}
```

By multiplying the hash code by 127 before adding in each new character, we make sure that each character has a different effect on the final result. The "%" operator with a prime number tends to "mix up the bits" of the hash code. The prime is chosen to be large, but not so large that $127 * \text{hashVal} + \text{key.charAt}(i)$ will ever exceed the maximum possible value of an int.

The best way to understand good hash codes is to understand why bad hash codes are bad. Here are some examples of bad hash codes on Words.

- [1] Sum up the ASCII values of the characters. Unfortunately, the sum will rarely exceed 500 or so, and most of the entries will be bunched up in a few hundred buckets. Moreover, anagrams like "pat," "tap," and "apt" will collide.
- [2] Use the first three letters of a word, in a table with 26^3 buckets. Unfortunately, words beginning with "pre" are much more common than words beginning with "xzq", and the former will be bunched up in one long list. This does not approach our uniformly distributed ideal.
- [3] Consider the "good" hashCode() function written out above. Suppose the prime modulus is 127 instead of 16908799. Then the return value is just the last character of the word, because $(127 * \text{hashVal}) \% 127 = 0$. That's why 127 and 16908799 were chosen to have no common factors.

Why is the hashCode() function presented above good? Because we can find no obvious flaws, and it seems to work well in practice. (A black art indeed.)

Resizing Hash Tables

 Sometimes we can't predict in advance how many entries we'll need to store. If the load factor n/N (entries per bucket) gets too large, we are in danger of losing constant-time performance.

One option is to enlarge the hash table when the load factor becomes too large (typically larger than 0.75). Allocate a new array (typically at least twice as long as the old), then walk through all the entries in the old array and rehash them into the new.

Take note: you CANNOT just copy the linked lists to the same buckets in the new array, because the compression functions of the two arrays will certainly be incompatible. You have to rehash each entry individually.

You can also shrink hash tables (e.g., when $n/N < 0.25$) to free memory, if you think the memory will benefit something else. (In practice, it's only sometimes worth the effort.)

Obviously, an operation that causes a hash table to resize itself will take more than $O(1)$ time; nevertheless, the average over the long run is still $O(1)$ time per operation.

Transposition Tables: Using a Dictionary to Speed Game Trees

 An inefficiency of unadorned game tree search is that some grids can be reached through many different sequences of moves, and so the same grid might be evaluated many times. To reduce this expense, maintain a hash table that records previously encountered grids. This dictionary is called a transposition table. Each time you compute a grid's score, insert into the dictionary an entry whose key is the grid and whose value is the grid's score. Each time the minimax algorithm considers a grid, it should first check whether the grid is in the transposition table; if so, its score is returned immediately. Otherwise, its score is evaluated recursively and stored in the transposition table.

Transposition tables will only help you with your project if you can search to a depth of at least three ply (within the five second time limit). It takes three ply to reach the same grid two different ways.

After each move is taken, the transposition table should be emptied, because you will want to search grids to a greater depth than you did during the previous move.

Postscript: A Faster Hash Code (not examinable)

 Here's another hash code for Strings, attributed to one P. J. Weinberger, which has been thoroughly tested and performs well in practice. It is faster than the one above, because it relies on bit operations (which are very fast) rather than the % operator (which is slow by comparison). You will learn about bit operations in CS 61C. Please don't ask me to explain them to you.

```
static int hashCode(String key) {
    int code = 0;

    for (int i = 0; i < key.length(); i++) {
        code = (code << 4) + key.charAt(i);
        code = (code & 0xffff) ^ ((code & 0xf0000000) >> 24);
    }

    return code;
}
```