CS 61B: Lecture 22
Wednesday, March 12, 2014

Today's reading: Goodrich & Tamassia, Chapter 5.

DICTIONARIES (continued)

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 * hashVal + 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 * 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 takes 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.

STACKS

A _stack_ is a crippled list. You may manipulate only the item at the top of the stack. The main operations: you may "push" a new item onto the top of the stack; you may "pop" the top item off the stack; you may examine the "top" item of the stack. A stack can grow arbitrarily large.

```
 | |          | |            | | -size()-> 2 |d| -top()-> d     | |
 |b| -pop()-> | | -push(c)-> |c|             |c|                | | -top()--
 |a|    |     |a|            |a| -push(d)--> |a| --pop() x 3--> | |       |
 ---    v     ---            ---             ---                ---       v
        b                                                                null
```

```
public interface Stack {
  public int size();
  public boolean isEmpty();
  public void push(Object item);
  public Object pop();
  public Object top();
}
```

In any reasonable implementation, all these methods run in O(1) time. A stack is easily implemented as a singly-linked list, using just the front(), insertFront(), and removeFront() methods.

Why talk about Stacks when we already have Lists? Mainly so you can carry on discussions with other computer programmers. If somebody tells you that an algorithm uses a stack, the limitations of a stack give you a hint how the algorithm works.

Sample application: Verifying matched parentheses in a String like "{[(){[]}]()}". Scan through the String, character by character.
- When you encounter a lefty--'{', '[', or '('--push it onto the stack.
- When you encounter a righty, pop its counterpart from atop the stack, and check that they match.

If there's a mismatch or null returned, or if the stack is not empty when you reach the end of the string, the parentheses are not properly matched.

```
QUEUES
======
A _queue_ is also a crippled list.  You may read or remove only the item at the
front of the queue, and you may add an item only to the back of the queue.  The
main operations:  you may "enqueue" an item at the back of the queue; you may
"dequeue" the item at the front; you may examine the "front" item.  Don't be
fooled by the diagram; a queue can grow arbitrarily long.

 ===             ===               ===              === -front()-> b
 ab. -dequeue()-> b.. -enqueue(c)-> bc. -enqueue(d)-> bcd
 ===     |       ===               ===              === -dequeue() x 3--> ===
         v                                                                ...
         a                                               null <-front()-- ===

Sample Application:  Printer queues.  When you submit a job to be printed at
a selected printer, your job goes into a queue.  When the printer finishes
printing a job, it dequeues the next job and prints it.

public interface Queue {
  public int size();
  public boolean isEmpty();
  public void enqueue(Object item);
  public Object dequeue();
  public Object front();
}

In any reasonable implementation, all these methods run in O(1) time.  A queue
is easily implemented as a singly-linked list with a tail pointer.

DEQUES
======
A _deque_ (pronounced "deck") is a Double-Ended QUEue.  You can insert and
remove items at both ends.  You can easily build a fast deque using a
doubly-linked list.  You just have to add removeFront() and removeBack()
methods, and deny applications direct access to listnodes.  Obviously, deques
are less powerful than lists whose listnodes are accessible.
```

```
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 & 0x0fffffff) ^ ((code & 0xf0000000) >> 24);
  }

  return code;
}
```