

CS231 Fall 2013 Lecture 37

Stephanie R Taylor

December 4, 2013

1 Administrative Topics

- Don't forget to fill out the course evaluations!

2 Hash tables/maps

On Monday, we talked about hash maps. Hash maps aim to store key/value pairs with $O(1)$ insert, removal, and look-up. They do so by “hashing” the key to an integer in an array. Whenever two keys hash to the same slot in the map, we have a collision. We talked about two strategies for handling collisions. Both involved doing some sort of linear search. And we noticed that if every key hashes to the same value, then the performance degrades severely to $O(n)$ insert, removal, and look-up.

There are two things to think about regarding performance:

1. How densely populated the map is. We need to balance the amount of empty space with the desire to avoid collisions. A good rule of thumb is that about half the slots should be filled.
2. How evenly distributed the keys are. This relies on the hash function's ability to “spread” the data.

The first item is easily taken care of with a dynamic hash map. We simply expand the capacity of the table once it becomes too full for comfort.

The second item is much more complex. Designing good hash functions is important and difficult.

(From Cliff Shaffer’s textbook:)

When designing hash functions, we are generally faced with one of two situations.

- 1. We know nothing about the distribution of the incoming keys. In this case, we wish to select a hash function that evenly distributes the key range across the hash table, while avoiding obvious opportunities for clustering such as hash functions that are sensitive to the high- or low-order bits of the key value.*
- 2. We know something about the distribution of the incoming keys. In this case, we should use a distribution-dependent hash function that avoids assigning clusters of related key values to the same hash table slot. For example, if hashing English words, we should not hash on the value of the first character because this is likely to be unevenly distributed.*

Important point: if we make a class that overrides the equals method, then we must override hashCode. If two items are equal, then their hash codes must be equal as well.

2.1 Hash functions for integers

(Functions and explanations taken from Cliff Shaffer’s textbook)

Hash function 1: The function returns the integer itself. (Then, to choose a slot in the table, its absolute value is taken and it is “modded” by the capacity of the table.)

- The function relies on the least significant digits of the number.
- Potential issues: If the the numbers are not distributed evenly (according to their least significant digits), then the hash function won’t distribute them evenly. For example, if the data has lots of numbers divisible by 10 or 5, then there will be clumping. Or if there are lots of even numbers, there will also be clumping.

Hash function 2: The function uses the mid-square method. (In this case, we use the hash map’s capacity in the hash function itself, rather than modding afterwards.)

- The mid-square method squares the key value, and then takes the middle r bits of the result, giving a value in the range 0 to $2r - 1$. r is chosen based on the capacity of the table. (And we can be loose with our definition of bits - we may simply mean “digits”).
- This works well because most or all bits of the key value contribute to the result.
- Example: consider records whose keys are 4-digit numbers in base 10. The goal is to hash these key values to a table of size 100 (i.e., a range of 0 to 99). This range is equivalent to two digits in base 10. That is, $r = 2$. If the input is the number 4567, squaring yields an 8-digit number, 20857489. The middle two digits of this result are 57. All digits (equivalently, all bits when the number is viewed in binary) contribute to the middle two digits of the squared value. Thus, the result is not dominated by the distribution of the bottom digit or the top digit of the original key value.

2.2 Hash functions for Strings

(Functions and explanations taken from Cliff Shaffer’s textbook)

Hash function 1: The function sums the ASCII values of the letters in a string. (Then, to choose a slot in the table, its absolute value is taken and it is “modded” by the capacity of the table.)

- If the hash table size M is small, this hash function should do a good job of distributing strings evenly among the hash table slots, because it gives equal weight to all characters.
- This is an example of the **folding** approach to designing a hash function. (Note that the order of the characters in the string has no effect on the result.) A similar method for integers would add the digits of the key value, assuming that there are enough digits to (1) keep any

one or two digits with bad distribution from skewing the results of the process and (2) generate a sum much larger than M.

- If the sum is not sufficiently large, then the modulus operator will yield a poor distribution. For example, because the ASCII value for “A” is 65 and “Z” is 90, the sum will always be in the range 650 to 900 for a string of ten upper case letters. For a hash table of size 100 or less, a reasonable distribution results. For a hash table of size 1000, the distribution is terrible because only slots 650 to 900 can possibly be the home slot for some key value, and the values are not evenly distributed even within those slots.

Here is the code

```
public static int simpleHash(String s) {  
    char ch[];  
    ch = s.toCharArray();  
    int i, sum;  
    for (sum=0, i=0; i<s.length(); i++)  
        sum += ch[i];  
    return sum;  
}
```

Hash function 2: The function sums the values of each 4-character substring. (Then, to choose a slot in the table, its absolute value is taken and it is “modded” by the capacity of the table.)

- It processes the string four bytes at a time, and interprets each of the four-byte chunks as a single long integer value. The integer values for the four-byte chunks are added together. In the end, the resulting sum is converted to the range 0 to $M - 1$ using the modulus operator.
- Example: For example, if the string “aaaabbbb” is hashed, then the first four bytes (aaaa) will be interpreted as the integer value 1,633,771,873 and the next four bytes (bbbb) will be interpreted as the integer value 1,650,614,882. Their sum is 3,284,386,755 (when viewed as an unsigned integer). If the table size is 101 then the modulus function will cause this key to hash to slot 75 in the table.
- Note that for any sufficiently long string, the sum for the integer quantities will typically cause a 32-bit integer to overflow (thus losing some

of the high-order bits) because the resulting values are so large. But this causes no problems when the goal is to compute a hash function.

Here is the code

```
public static long sfold(String s) {
    int intLength = s.length() / 4;
    long sum = 0;
    for (int j = 0; j < intLength; j++) {
        char c[] = s.substring(j*4,(j*4)+4).toCharArray();
        long mult = 1;
        for (int k = 0; k < c.length; k++) {
            sum += c[k] * mult;
            mult *= 256;
        }
    }
    char c[] = s.substring(intLength * 4).toCharArray();
    long mult = 1;
    for (int k = 0; k < c.length; k++) {
        sum += c[k] * mult;
        mult *= 256;
    }
    return Math.abs(sum);
}
```