CSCI 104L Lecture 18: Collision Resolution

Quadratic Probing

As before, if $h(k) = i$ and $A[i]$ is taken, try somewhere else. Except now, $A[i+1]$ is followed by an attempt at $A[i+4]$, then $A[i+9]$ and so on. As before, loop back to the beginning if you hit the end of the array.

That is, $h(k, i) = (h(k) + i^2)\%m$.

Two different items mapped to i and $i+1$ respectively will follow very different paths. We don't get *primary clumping* as we did for Linear Probing.

Secondary clumping will only occur amongst all of the items which were originally mapped to index i. Note that chaining has this form of clumping as well.

Question 1 *Using the hash function* $h(k) = k\%10$*, determine the contents of the hash table after inserting* $1, 11, 2, 21, 12, 31, 41$.

The load factor of a hash table is the number of stored elements divided by the number of indices.

When you use chaining, you generally try to keep the load factor below 1.0. If you load factor becomes too large, you should resize the array and rehash the contents (and possibly use a new hash function).

When you use probing, you **must** make sure the load factor of the hash table is reasonable. If the hash table is completely full, you simply cannot add new elements. As you approach completely full, operations take ridiculously long. Usually you'll want to keep the load factor below 0.5.

In fact, if your load factor is above 0.5, you cannot guarantee that quadratic probing will find an empty bucket, even if the hash table size is prime.

Question 2 *Using the hash function* $h(k) = k\%7$*, determine what happens after inserting* $14, 8, 21, 2, 7$.

If the table size is not prime, the problem is even worse.

Question 3 *Using the hash function* $h(k) = k\%9$*, determine what happens after inserting* $36, 27, 18, 9, 0$.

If your hash table has a prime size, then the first $\frac{m}{2}$ probes are guaranteed to go to distinct locations, meaning that you will find a location if the load factor is no more than 0.5.

Double Hashing

This avoids both primary and secondary clumping. You have a second hash function h'(k).

If $h(k) = i$ is taken, then try, in order, $i + h'(k)$, $i + 2h'(k)$, $i + 3h'(k)$, ...

That is, $h(k, i) = (h(k) + i \cdot h'(k))\%m$.

This is better because even amongst all the items mapped to i, they will follow very different paths.

A good choice of secondary hash function can ensure you always find a free slot as long as the load factor is smaller than 1 (but you still want to keep the load factor below 0.5 for performance issues).

An example of a good secondary hash functions is $h'(k) = p - (k\%p)$, where p is a prime smaller than m.

Removal

Question 4 *There is a serious problem if we want to delete an item from a hash table that uses probing. What is it?*

The "simplest" solution is to add a "deleted" flag to an item in the hash table that has been removed. When searching, if you hit a deleted item, you keep searching.

Now the load factor should consider both items and deleted items, since search will get unreasonably long otherwise.

After you have too many deleted items, you will want to rehash your entire hash table contents, so as to maintain a reasonable runtime (you don't necessarily have to resize the hash table though). This takes a long time!

In an amortized sense, it's okay, since you have to do a bunch of deletes before you have to re-hash.

Bloom Filters

Question 5 *Could we use a hashtable to implement a set?*

Question 6 *What kind of issues would arise with such an implemenation?*

Bloom Filters are a way to avoid requiring the storage of keys. They come with a tradeoff: Bloom Filters are Monte Carlo Randomized Algorithms. That is, there is a small chance you will get the wrong answer.

When implementing a set, we generally have three functions:

- add
- remove
- contains

When using a Bloom Filter, we will not implement remove (it's generally not worth it).

Your set merely needs to save which items are inside. You can add an item, and you can check whether an item is contained within it.

All we really need is an array of bits. If we want to store item k inside our set, then pass k into our hash function h(k), and set a[h(k)] = 1.

Question 7 *What's the problem with this proposed implementation?*

A false positive occurs when we say something is in the set, but it actually isn't. We will try to minimize the probability of this occuring.

Question 8 *Is a false negative possible?*

To reduce the probability of a false positive, we can use multiple hash functions $h_1, h_2, ..., h_j$. When we add an item k to the array, we set $h_1(k), h_2(k), ..., h_j(k)$ to true.

When we check whether an item is in the array, we make sure ALL of the bits $h_1(k), h_2(k), ..., h_j(k)$ are set.

Question 9 *Can a false negative occur in this new implementation?*

The probability of a false positive should have vastly decreased, because we have j independent tests as to whether the item is being stored in the set. The tradeoff is that we must increase the size of our bloom filter to compensate for the additional number of inserted bits.

If you want a false positive rate of 1%, you would want 7 hash functions, and 9.2 bits in the bloom filter for each item you plan to have inside.

If you want a false positive rate of .1%, you would want 10 hash functions, and 14 bits per key.

Most Bloom Filters will have between 2-20 hash functions.

Pre-filter Bloom Filters have become quite popular for applications that expect the majority of queries to return "no". You do a very cheap Bloom Filter operation to check whether the item is in the set or not. If the answer is "yes", we then verify it via a more expensive operation.

Suppose we have the following hash functions:

- $h_1 = (7x + 4)\%10$
- $h_2 = (2x + 1)\%10$
- $h_3 = (5x + 3)\%10$

Question 10 *What will the Bloom Filter look like after inserting* $0, 1, 2, 8$*?*

Question 11 *What will the Bloom Filter say if we lookup* $2, 3, 4, 9$*?*

Question 12 *Should we store our Bloom Filter as an array of bools?*

We will use a concept called Masking to improve our space requirements.

Declare an array of integers. Each integer has 32 bits, so the first 32 bits will be stored in a[0], the second 32 bits will be stored in a[1], etc.

- To check if the jth bit is set, use the following formula: a[j/32] & (1 << (j % 32))
- (1 << j) will take the 1, and shift it left j bits. That is, it will insert j 0's to the right of the 1.
- x & j will do a bitwise AND between x and j.
- To set the jth bit, use the following formula: a[j/32] | (1 << (j % 32))