Lecture 10: More Hashing

COSC242: Algorithms and Data Structures

Brendan McCane

Department of Computer Science, University of Otago

Double hashing

Problems: Linear probing results in primary clustering. Quadratic probing results in secondary clustering. Both occur because the collision resolution strategy follows the same path from a collision point regardless of the key value.

A better idea: Double hashing.

$$H(k, i) = (h(k) + i \cdot g(k))\%m$$

where i is the number of collisions so far, h does division hashing by tablesize m, and g is a "secondary" hash function.

g is often something like $g(k) = 1 + k\%(m - 1)$.

Why don't we just make $g(k) = k\%(m - 1)$? Why is it helpful if m is prime?

Quadratic vs double hashing

Insert 12, 13, 43, 52, 72, 63, with $h(k) = k\%10$, using first quadratic probing then double hashing.

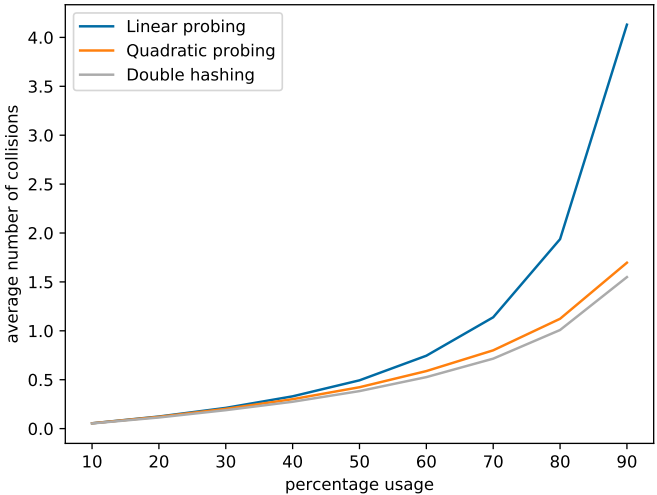
quadratic

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

double

Linear vs quadratic vs double



Full tables

What happens as a hash table gets full?

As clusters get large, gaps become fewer and the number of collisions for insertion and for search becomes larger, breaking down the $O(1)$ property.

What if you want to get rid of an item? Beware of deleting a key on a search path that has possible collisions. We have to use *lazy* deletion.

Suggestion: hash tables implemented as arrays are good if you know roughly the size of your data set before loading and it stays that size. Or, if you are willing to accept a single costly operation, you can re-hash everything into a different tablesize.

Alternative: Allow the hash table to maintain "chains" at each location. This is called "hashing with chaining" (as opposed to what we have already seen, which is called "open addressing").

Chaining

Chaining means there is a linked list hanging off each cell of your array, which makes collision resolution easy: simply put the key into the list hanging off at that location.

Note: Inserting at the head of a linked list is $O(1)$. Searching a linked list is $O(n)$, which is fine for small n but not as good as we'd hoped if n is large.

If the table has m slots and m keys, and if our hash function spreads the keys fairly evenly over the hash table, then we can expect our chains to be short. In fact, it is not unusual for the table size to be chosen to be $m = n/3$, where n is the size of the collection of data items, provided one is sure that your hash function will spread the keys evenly.

This scheme is vulnerable to any adversary who is able to select data that creates long lists, making search more time-consuming.

Universal hashing

If you use chaining, an adversary who knows your hash function could devise a set of keys that all hash to the same position, creating a list for the whole collection of data items (search and insert become $O(n)$).

Solution: choose the hash functions randomly before creating the hash table. (This assumes the hash table will not be persistent. You will want to use a different hash function on each execution.)

A *universal* set of hash functions H is a set of hash functions such that if you pick keys k and j at random, and choose a hash function h randomly from H, then the chance of $h(k) = h(j)$ is no more than $1/m$ (where m is the size of the hash table).

So how does one get a universal set of hash functions?

A universal set of hash functions

Choose a prime number p big enough that every possible key k is $< p$, and choose your table size $m < p$.

Now make $h_{a,b}(k) = ((ak + b) \mod p) \mod m$.

The parameters a and b may take on integer values up to $p - 1$, but you must choose $a > 0$.

a and b are chosen at random at program startup.

Example universal hash function

So, with $p = 17$ and $m = 6$,

$$\begin{aligned} h_{3,4}(8) &= ((3 \cdot 8 + 4)\%17)\%6 \\ &= (28\%17)\%6 \\ &= 11\%6 \\ &= 5. \end{aligned}$$

Try $h_{4,5}(8)$, and $h_{4,5}(8)$ but with $p = 19$ and $m = 7$.

Choosing a and b at random as the program starts denies an adversary the ability to choose input that generates the worst case.

Relevant parts of the textbook

Double hashing is discussed in section 11.4.

Chaining is discussed in section 11.2.

Universal hashing is discussed in section 11.3.3.