Hash Tables

Using an array, we can retrieve/search for values in that array in O(lg n) time. However, inserting an item into an array quickly can be difficult. Why is this?

Also, if we store items in a binary tree, we can also retrieve/search for them in O(lg n) time as long as the tree is balanced. Insertion also takes O(lg n). These seem to be some pretty good numbers (we'll do the analysis later when we start to cover binary trees), but actually, we can do even better.

A hash table allows us to store many records and very quickly search for these records. The basic idea is as follows:

If you know you are going to deal with a total of n records, create an array of size n. Each array slot should be able to hold 1 record. Now, the crux to a hash table working is a good hash function. A hash function takes as an input the type of record being stored, and outputs a value from 0 to n-1, an integer that is a valid index into the array.

So, for example, if you were storing Strings, the hash function would have to map an arbitrary String to an integer in the range of the hash table array. Here is an example of such a hash function (this is a very poor hash function):

f(w) = ascii value of the first character of w.

One of the first things to notice about a hash function is that two different values, such as "cat" and "camera", can hash to the exact same value. (In this case, the ascii values of the first letters of both words are identical.)

For the time being, let's ignore this problem. Let's assume that our hash function works in such a way that every element we try to store in it miraculously hashes to a different location.

Now, imagine searching for an element in the hash table. All you have to do is compute its hash value and then just look in that ONE particular array slot! Thus, the running time of this operation is simply based on how long it takes to compute the hash function. Hopefully, we can come up with a hash function that works reasonably well that can be computed in O(1) time.

So now we get to the problem of collisions. A collision is when two values you are storing in your hash table hash to the exact same location. (In essence, we have that f(x) = f(y) when x ≠y.) Some ideas of how to deal with these:

1) Don't: Just replace the new value you are trying to insert with the old one stored in the hash table.

2) Linear Probing: If there is a collision, continue searching in the array in sequential order until you find the next empty location.

3) Quadratic Probing: If there is a collision, continue searching in the array by offsets of the integers square. This means you first look in array index c, the original index, followed by index c+1, then index c+4, then index c+9, etc.

4) Separate Chaining Hashing: Rather than storing the hash table as an array of elements, store it as an array of linked lists of elements. If there is a collision, just insert the new element into the linked list at that slot. Hopefully, there will only be a few collisions so that no one linked list will be too long. Although searching may not be exactly O(1) time, it will definitely be quicker than O(lg n) time.

The Hash Function

Since these are difficult to truly analyze, I won't get into much detail here. (The book does not either.) Ideally, a hash function should work well for any type of input it could receive. With that in mind, the ideal hash function simply maps an element to a random integer in the given range. Thus, the probability that a randomly chosen element maps to any one of the possible array locations should be equal.

Given this reasoning, why is the hash function I showed you earlier a poor choice?

Mainly for two reasons:

1) It's designed for an array of only size 26. (Or maybe a bit bigger if we allow non-alphabetic characters.) Usually hash tables are larger than this.

2) More words start with certain letters than others. These hash locations would be more densely filled.

Let's go over a couple ideas in the book for hash functions (when dealing with Strings):

1) Each printable character has a ascii value less than 128. Thus, a string could be a representation of a number in base 128. So, if we had the String "dog", we could hash it to the integer

$$\text{ascii('d')}*128^0 + \text{ascii('o')}*128^1 + \text{ascii('g')}*128^2 =$$

$$100*1 + 111*128 + 103*128^2 = 1701860.$$

What are the problems with this technique?

1) Relatively small Strings map to HUGE integers. It is possible that these integers are not valid indexes to our hash table.

2) Just computing this function may cause an overflow error. Remember that the int type can only store an integer up to $2^{31} - 1$. Any string of length 5 or higher would hash to a value greater than this.

How can we deal with these problems?

Simply put, the mod operator can help us deal with both of these issues. Change the function as follows:

$f(c_0c_1...c_n) =$
$(ascii(c_0)*128^0 + ascii(c_1)*128^1 +...+ ascii(c_n)*128^n)$ mod tablesize

Now, we get a guarantee that each String hashes to a valid location in the table, and it's not readily apparent that this function isn't fairly random. It may not be, but chances are it's better than the first one I showed you. However, rigorously proving this is most certainly beyond the scope of this class.

Now, if you do this calculation in one particular way, you have the possibility creating overflow. (Namely if you only do the mod at the end rather than at each point of the calculation...)

If you apply the mod at each step, then all intermediate values calculated remain low and no overflow occurs.

Also, using Horner's rule can aid in this computation. Applying the rule states that

$$(ascii(c_0)*128^0 + ascii(c_1)*128^1 + ... + ascii(c_n)*128^n) =$$

$$ascii(c_0) + 128(ascii(c_1) + 128(ascii(c_2) + ... + (128(ascii(c_{n-1}) + 128\ ascii(c_n))...))$$

In general, Horner's rule specifies how to evaluate a polynomial without calculating x^n in that polynomial directly. Here it is:

$$c_nx^n + c_{n-1}x^{n-1} + ... c_1x + c_0 = c_0 + x(c_1 + x(c_2 + ... + x(c_{n-1} + xc_n)...))$$

Notice that if you are trying to compute the value of a polynomial mod n none of your intermediate calculations get very large.

An example of a more severe pitfall with a hash function is one like the following:

$$f(c_0c_1...c_n) = (ascii(c_0) + ascii(c_1) + ... + ascii(c_n)) \bmod tablesize$$

The problem here is that if the table size is big, say just even 10000, then you find that the highest value an 8 letter string could possibly hash to is 8*127 = 1016. Then, in this situation, you would NOT be using nearly 90% of the hash locations at all. This would most definitely result in many values hashing to the same location.

Linear Probing

This is the first reasonable idea mentioned with respect to dealing with collisions. (Throwing away values from the hash table shouldn't be an option if you want any sort of retrieval accuracy.)

The idea is as follows:

Let's say that our hash table can hold up to 10 integer values and that it currently looks like this:

index	0	1	2	3	4	5	6	7	8	9
value				173	281		461			

Now imagine that the next value to store, 352, hashed to location 3.

We see that a value is stored there, so instead, we now look at location 4. Since a value is stored there also, we continue to the next location, where we have found a place to store 352:

index	0	1	2	3	4	5	6	7	8	9
value				173	281	352	461			

Now, if we want to search for 352, we'd first figure out it's hash value was 3. Since it is NOT stored there, we'd look at location 4. Since it's not stored there, we'd finally find 352 in location 5.

Incidentally, when do we know that we can stop searching for a value with this method?

When we hit an empty location is the answer...until then there is NO way to tell whether or not the values we are looking at initially hashed to 3 or not.

So the question arises - how many array elements should we expect to look at during a search for a random element.

IF, the elements in the hash table were randomly distributed, then we could use the following analysis:

(Note: If you can't follow this, don't worry about it...)

Let λ be the fraction of the elements that are filled in the hash table. This means that 1-λ fraction of elements are free in the hash table.

Thus, the probability that the first element we search is empty is 1-λ.

But, what if that's not the case. Then the probability that the first element we search is full, but the second is empty is $\lambda(1-\lambda)$.

Similarly, the probability that the first two elements are full but the third empty is $\lambda^2(1 - \lambda)$

In any event, our approximate expected number of cells searched is:

$$\sum_{i=1}^{\infty} i(1-\lambda)\lambda^{i-1} = (1-\lambda)\sum_{i=1}^{\infty} i\lambda^{i-1}$$

$$= (1-\lambda)\frac{d}{d\lambda}\sum_{i=1}^{\infty}\lambda^i$$

$$= (1-\lambda)\frac{d}{d\lambda}(\lambda\sum_{i=0}^{\infty}\lambda^i)$$

$$= (1-\lambda)\frac{d}{d\lambda}(\lambda(\frac{1}{1-\lambda}))$$

$$= (1-\lambda)(\frac{(1-\lambda)1-\lambda(-1)}{(1-\lambda)^2})$$

$$= (1-\lambda)(\frac{1-\lambda+\lambda}{(1-\lambda)^2})$$

$$= (1-\lambda)(\frac{1}{(1-\lambda)^2})$$

$$= \frac{1}{(1-\lambda)}$$

Basically, what this says is that if λ = .5, (meaning that the table is at most half full), then we would expect to have to search 2 elements in the hash table.

Unfortunately, this analysis isn't accurate because in a hash table with linear probing, the elements do NOT distribute them selves in the hash table randomly.

Rather, clustering occurs. After a few values have been placed in the table, you'll get a run of several indexes in a row in the table that are filled. Now, it's pretty likely that something will hash to that cluster. It won't "find a home" so to speak until it gets to the end of the cluster, thereby increasing the size of the cluster as well.

It turns out that if you do the analysis with clustering, which is beyond the scope of this class for sure (it's a lot uglier than what I just showed you above...) the actual number of cells examined is, on average $(1 + 1/(1 - \lambda)^2)/2$.

This isn't so bad if λ is only .5, but as λ gets close to 1, say .9, then the average number of values to search is about 50 or so, which certainly is not efficient.

Thus, if you use linear probing, it makes sense to make your hash table about twice as big as the number of elements you want to store in it.

Quadratic Probing

The idea here is to get rid of primary clustering. Now consider the same situation as before where we are trying to insert 352 into the following hash table:

index	0	1	2	3	4	5	6	7	8	9
value				173	281		461			

In quadratic probing, after we see that 352 hashes to location 3 and that location is filled, we will go to location 4 (since this is $3+1^2$), But this is taken as well. So next, we will go to location 7 (since this is $3+2^2$), and this is where we can store 352.

index	0	1	2	3	4	5	6	7	8	9
value				173	281		461	352		

Although this example wasn't a great example of the advantage of quadratic probing, hopefully you can see that if we have to try four or five locations to place an element that we aren't necessarily trying in the same cluster.

The idea is the same here. You keep on looking for an element until you find it or you hit an empty location in your search. (In our situation, if we were searching for 863, and let's say that also hashed to location 3, then we would check location 3, then location 4, then location 7, and finally location 2, since the next place to check is $(3+3^2)\%10 = 2$.

Now, there are some complications with quadratic probing. How do we know that eventually we will find a "free" location in the array, instead of looping around all filled locations?

It turns out that if the table size is prime, AND the table is at least half empty, quadratic probing will always find an empty location.

Here is the proof:

Let the table size be M, where M is an odd prime > 3. We will show that the first M/2 of the possible insertion locations are unique. Let's say that a value to store in the hash table hashes to array location H, which is already filled. Now consider two of the possible locations that the element could end up. Let these be $(H+i^2)$ mod M and $(H+j^2)$ mod M, with $0<i,j<M/2$, and i and j are distinct integers. Assume to the contrary that these two represent the same index into the hash table. Then we have:

$(H+i^2) \equiv (H+j^2) \bmod M \Rightarrow$
$i^2 \equiv j^2 \bmod M \Rightarrow$
$M \mid (i^2 - j^2) \Rightarrow$
$M \mid (i - j)(i+ j)$

Since M is prime, it follows that either M | (i-j) or M | (i+j). But neither of these can be true since i can not equal j and the maximum value of i+j is (M-1)/2, since M is odd.

This proves that if we keep our hash table at least half empty and use quadratic probing, we will ALWAYS be able to find a location for a value to hash to. (Notice that unlike linear probing where we are guaranteed to cycle through every possible location when we hash, here it is more difficult to prove such a property.)

Dynamic Table Expansion

Of course it's certainly possible that we won't know how many records we'll have to store in a hash table before we set it up. So, it would be nice if we could expand our hash table, much like we expanded our stack and queue when we used an array implementation.

But, there are more issues here to think about. How will our hash function change? Once we change that, what do we have to do for each value in the hash table?

When we want the table size to expand, it makes sense to at least double the size. Also, it turns out that prime numbers usually work best with respect to hash functions that use mod. So, our table expansion process works as follows:

1) Pick a prime number that is approximately twice as large as the current table size.

2) Use this number to change the hash function.

3) Rehash ALL the values already stored in the table.

4) Now, hash the value to currently be stored in the table.

Step one seems like it would be expensive, but probabilistic primality testing makes it a fairly quick and efficient process.

Step three is the most time consuming step. It would run in O(n) time, where there are currently n values stored in the table and the hash function can be computed in constant time.

Separate Chaining Hashing

This is last alternative mentioned, and in my opinion, the one that makes the most sense. Instead of having some sort of method to try new hash locations, stick with a hash location, but somehow find a way to store more than one value at that hash location.

Naturally, one would simply think about creating a 2D array. The problem with this is the waste of memory. A hash table is supposed to have one dimension that is very large. Also, to be safe, it would be necessary to leave plenty of room in the second dimension just in case there was one "bad" hash location. But in this situation, you'd ALSO have to allocate space for all the other rows, and this would end up being a great deal of wasted space.

Instead, we can store multiple values at one location of a hash table if each location was simply a linked list. Since we expect each linked list to be small, inserting and searching for a value should not be too time consuming. Assuming that the hash function works randomly, the worst possible time (on average) for a search operation is O(lg lg n), which is a very small amount of time. (Once again, the proof of this is beyond the scope of this class.)