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

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

$$100*128^2 + 111*128 + 103 = 1652711.$$