Hash Tables - Part 1

Robin Dawes

February 19, 2021

Attempts to provide a solid introduction to Hash Tables without making an absolute hash of it.

.

Can We Do Better Than the Best Possible?

We have seen that with a balanced binary tree (such as a Red-Black Tree) we can guarantee $O(\log n)$ worst-case time for insert, search and delete operations. We also know that these operations are in $\Omega(\log n)$ for comparison-based algorithms - which means that balanced binary trees give the optimum performance for these operations. Our challenge now is to try to improve on that optimum. It turns out we can ... if we change the rules.

I love the smell of a paradox in the morning.

Suppose we have data objects that consist of pairs - each pair having a key value that is unique to the object, and some satellite data that may or may not be be unique. For example, the data might consist of "phone number, name" pairs in which each phone number must be unique (the key), but there might be two people named "Cholmondeley Featherstonehaugh Marjoribanks Wriothesley" (duplicate satellite data).

I would feel sorry for both of them - even though their shared name is pronounced "Chumley Fanshaw Marchbanks Rotslee" which is pretty cool.

Assuming for the moment that the keys are integers, the simplest method to store the data is in an array that has an address for each possible key - this is called direct addressing. Then insert, search and delete all take $O(1)$ time - we can't beat that!

How does this avoid the $\Omega(\log n)$ lower bound that we proved earlier? The insert, search and delete algorithms for direct addressing never compare elements of the set so they are not comparison-based.

The problem of course is that the set of possible keys (we call this the key space) may be immense - with 10-digit phone numbers there are 10^{10} possible combinations, and most people have at most a couple of hundred phone numbers in their contact list. Even if you have every person in Canada in your contact list, creating an array with 10^{10} elements and only using 38,005,238[1] of them is not very practical.

[1] 2020 Q3 estimate from `https://en.wikipedia.org/wiki/Canada`

However the idea of using a simple array to store our data is very appealing and as we shall see, with a little care and attention we can

get good **average-case** complexity for our three essential operations, even though we may not achieve optimal **worst-case** complexity.

Here we go, changing the rules. Who ever said we can look at average-case complexity?

Since we are introducing average-case complexity we should spend a moment looking at balanced binary trees in this way. In a complete binary tree (ie. one in which there are no missing children - all the leaves are at the bottom level) - almost precisely half the vertices are at maximum distance from the root - and that distance is basically $\log n$. This implies that the average insert/search/delete time is going to be close to $\log n$. We can make a similar argument for Red-Black Trees.

Since we are contemplating using an array that is smaller than the set of all possible keys, we clearly need some way to map key values onto array addresses. For example if our array (which we will call T) has index values 0, 1, ..., 9 and our key value is 34, we need to choose a mapping from this key value to a table address. We call this mapping a HASH FUNCTION. We call the array T a HASH TABLE.

For the rest of this unit on hash tables and hash functions we will use the following notation consistently:

m : the size of T. T has addresses $0, 1, ..., m-1$

n : the number of data objects we need to be able to store in T. If this is not known precisely, we should at least be able to put an upper limit on it.

$h(k)$: a function that takes a key value k as its argument and returns a value in the range $[0...m-1]$

We will spend some time later talking about how to choose $h(k)$, but for now we will assume the keys are non-negative integers and we will use

$$h(k) = k \bmod m$$

as our hash function. So continuing the previous example, with $m = 10$ and $k = 34$, we get $h(k) = 4$

The problem is that since the number of possible keys exceeds m, $h(k)$ is necessarily a many-to-one function. We may get COLLISIONS - two or more keys in our set that hash to the same address. To deal with collisions we need to define a collision resolution method.

Note that due to collisions, we must store the key value as well as its satellite data - otherwise we cannot distinguish between the data associated with different keys that have the same value of $h(k)$. In

almost all of the subsequent examples and discussion in these notes we will only show the keys being stored. But in a real application we need to store both keys and satellite data.

Collision Resolution Methods

A WIDE VARIETY of collision resolution methods have been proposed. We will briefly look at some simple but not very useful ideas and then explore some popular and practical solutions to this problem.

A very bad method: If we are trying to insert a value k and the address $h(k)$ is already occupied, we simply reject the new data item. This has the advantage of making insert/search/delete all $O(1)$ worst case - but it has the downside that we are frequently unable to successfully insert new values even though there may be a lot of empty space in T.

Another very bad method: If we are trying to insert a value k and the address $h(k)$ is already occupied, we overwrite $h(k)$ with the new data item. This has $O(1)$ complexity for all operations, and we are always able to insert a new value. Alas, we are likely to lose a lot of data.

Challenge: come up with a situation where you can argue that one of the above methods is useful.

A Good Method: Chaining

IN A CHAINED HASH TABLE, T is not used to store the data directly. Each element of T is a pointer to the head of a linked list of objects that have all hashed to the same location in T. If no items currently in the set have $h(k) = i$, then T$[i]$ is a nil pointer. Each data pair is implemented as an object that contains the key value, the satellite data, and a pointer variable that will be used to connect to the next object in the list, if any.

Consider the three operations: Insert, Search, Delete.

Insert: It is always possible to insert a new item with key k into the hash table. We add the new object at the head of the list attached to its hash value address $h(k)$ in T. This gives insertion $O(1)$ complexity.

```
def insert(new_object):
    hash_val = h(new_object.key)
    new_object.next = T[hash_val]
    T[hash_val] = new_object
```

Search: Search is also simple: we go to the hash value address $h(k)$ in T and search through the list:

```
def search(k):
    hash_val = h(k)
    temp = T[hash_val]
    while temp != null AND temp.key != k:
        temp = temp.next
    if temp != null:
        return "found it"
    else:
        return "not found"
```

Of course "found it" and "not found" may not be the most useful responses from the search algorithm. It's more likely that we want to do something with the item so it is better to return a pointer to the object, or a nil pointer if we didn't find it.

```
def search(k):
    hash_val = h(k)
    temp = T[hash_val]
    while temp != nil AND temp.key != k:
       temp = temp.next
    return temp
```

Delete: Deletion is similar to searching - we just need to fix up the pointers in the list if/when we find the item.

```
def delete(k):
    hash_val = h(k):
    temp = T[hash_val]
    if temp == nil:
        return
    elif temp.key == k:
        T[hash_val] = temp.next
    else:
        previous = temp
        current = temp.next
        while current != nil and current.key != k:
            previous = current
            current = current.next
        if current != nil:
            previous.next = current.next
```

Since the linked lists can be arbitrarily long there is no upper limit to the number of values that can be stored in the hash table. But the longer the chains, the longer it will take to search and/or delete.

We will adopt the UNIFORM HASHING ASSUMPTION: we assume that our hash function $h(k)$ maps key values uniformly onto addresses - that is, each key is equally likely to be hashed to each address. The validity of this assumption depends on a number of factors, including the distribution of the keys within the key space, the size of the table, and the hashing function itself - we will return to this issue later but for now we will just make the assumption. In effect, this assumption means that approximately equal number of keys will be mapped onto each address in T.

With this assumption, the expected number of data objects in each chain is $\frac{n}{m}$ From this it is possible to show that the expected number of steps in a search (either successful or unsuccessful) is in $O\left(1+\frac{n}{m}\right)$ - full details of this proof can be found in CLRS. Since a delete operation requires only $O(1)$ operations after the object has been found, the expected time for both search and delete is in $O\left(1+\frac{n}{m}\right)$.

Recall that n is the number of values stored in the table, and m is the size of the table.

Writing $O\left(1+\frac{n}{m}\right)$ instead of just $O\left(\frac{n}{m}\right)$ may seem a little odd, but here's one way to see why it is useful: if we treat both n and m as variables, then $O\left(\frac{n}{m}\right)$ can be arbitrarily close to 0. But every search operation will take at least a constant amount of time because we have to compute the hash value of the key. Including the 1 in the order reflects the fact that the complexity cannot be arbitrarily small.

Nonetheless, I'm going to be lazy and write the complexity as $O\left(\frac{n}{m}\right)$ ... for most combinations of n and m it makes no difference.

Is it reasonable to think of n and m as variables? When we are creating our data structure we can ask this question: when the expected number of data items to be stored is n , how big should m be to make our operations efficient? Or looking at it another way: for given values of n and m, what is the expected search time? In these questions both n and m are definitely variables. When building a hash table in a real-world situation, n is most likely going to be dictated by the application, and m would be computed from that ... but knowing how to choose m wisely comes from considering the relationship between n, m and efficiency (which is what we are doing now).

If you are interested in visualizations, you could do some experimentation to gather data for plotting average search time as a function of n and m for the various collision resolution methods that we discuss.

The ratio $\frac{n}{m}$ is called the LOAD FACTOR of the hash table, and we often see the symbol α used for this. If α is high then collisions are

more likely.

The downside of chaining is that indirect addressing (the pointers we use to link together the chains) is physically slower than direct addressing. The most popular alternative is to resolve a collision by finding an empty address in the table and storing the new data object there. This is called OPEN ADDRESSING.