CMSC 420: Hash Functions

Justin Wyss-Gallifent

April 21, 2025

1 Introduction

This is a brief overview of hash functions including definitions and an introduction to some of the issues.

2 The Problem

All of our data so far has used keys which are sortable in some way, whether they're integers or points, and which can be put in trees, for example, which can be quickly searched.

However what if the keys for our data are not so clean and tidy? For example keys might be strings or more abstract objects. In cases like this it's difficult to know how to begin.

We can of course use things like dictionaries in Python and assign:

```
city['justin'] = 'washington'
```

But this is obfuscating the problem of exactly how Python is storing this. After all, it's almost certainly not creating some sort of list with index `justin` and it's not even clear what that might mean.

The somewhat obvious approach would be to convert our keys into integers in a reliable manner but this presents its own problems. For example suppose for a string of characters $s = c_n...c_1$ with each $0 \leq c_i \leq 25$ (just the alphabet for now) we could create an index via the function:

$$f(s) = \sum_{i=0}^{n} c_i \cdot 26^i$$

But this rapidly gets out of control because the indices are so massive:

$$f(justin) = 9 \cdot 26^6 + 20 \cdot 26^5 + 18 \cdot 26^4 + 19 \cdot 26^3 + 8 \cdot 26^2 + 13 \cdot 26^1 = 3026434762$$

We need a better way.

3 Hash Functions and Hash Tables

Our solution is in fact to use a function to map our keys to integers as discussed above but in a more elegant way so that the range of the function is manageable.
Definition 3.0.1. Suppose we have a set of keys K and a table T (think of a simple list with m entries) called a *hash table*.

A *hash function* is a function:

$$h : K \to \mathbb{Z}_m$$

The motivating idea behind a hash function is that for a key $x \in K$ we first calculate $h(x)$, this will tell us which table entry we are dealing with, then we store x in $T[h(x)]$.

Note 3.0.1. A few things to note:

- Although T is a list we have not been specific about the type of items it can store. They could be bits, integers, or even things like pointers. Whatever they are, h will need to deal with them.
- We wrote "store" but storing might be more than just assigning $T[h(x)] = x$ as we shall see.
- In practice T should be small(ish) so that it is manageable.
- For various reasons (some of which we'll address) ideally we want our hash functions to be "as random as possible", meaning that if we don't know actually know the hash function (for example for an outside observer) then it should appear that each $h(x)$ is randomly assigned and it should appear unclear how $h(x)$ is calculated from x. Note that we don't want them to actually be random, just appear that way.
- There is no requirement that h be either one-to-one or onto. You might protest that this might lead to collisions but the hash function itself doesn't care about that; dealing with collisions is a further step.

Here are some examples which will immediately illustrate the idea although you will probably see the major problem. These examples do not appear particularly random but they're good for reference.

Example 3.1. If $K = \mathbb{Z}$ and T is a a list with 100 entries then we could define:

$$h : K \to \mathbb{Z}_{100}$$

By:

$$h(x) = x \mod 100$$

Then for example $h(7) = 7$ and $h(132343188) = 88$ and so the key $x = 7$ (and its data) would be stored in $T[7]$ and the key $x = 132343188$ (and its data) would be stored in $T[88]$.

Example 3.2. If $K = \mathbb{Z}$ and T is a a list with 100 entries then we could pick an integers a and define:

$$h : K \to \mathbb{Z}_{100}$$

By:

$$h(x) = ax \mod 100$$

Here's a sneakier one:

Example 3.3. If we have:

$$K = \{x \mid x \text{ is a finite bit string}\}$$

Then we can create a hash function by choosing some fixed positive integer j and defining $h : K \longrightarrow \mathbb{Z}_{32}$ as follows:

For any key $x = b_n...b_0$ we divide it into substrings of length j, padding if necessary, then we define $h(x)$ as the XOR of the substrings, converted to decimal.

For example if our hash function has $j = 5$ then for $x = 100111110101000101$ we do the following:

We split up into groups of 5 bits and pad the right end:

$$10011\Big|11101\Big|01000\Big|101\underbrace{00}_{\text{pad}}$$

The we line them up and XOR the columns:

$$\begin{array}{ccccc} 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ \hline 1 & 0 & 0 & 1 & 0 \end{array}$$

So $h(100111110101000101) = 10010_2 = 18$.

Thus our key $x = 100111110101000101$ would be stored in $T(18)$.

The problem with both of these is that they have *collisions*, meaning two or more inputs can produce the same output.

Example 3.4. In our first example $h(7) = h(107) = h(207) = ...$

If we have $h(x_1) = h(x_2)$ with $x_1 \neq x_2$ then what exactly do we store in $h(x_1) = h(x_2)$?

The approach here is two-fold:

1. To try to engineer the hash function so as to have as few collisions as possible. This is partly accomplished by ensuring that the hash function "acts randomly".
2. To resolve the collisions after the hash function has done its job.

In addition this should all be as fast as possible because even if it is $\mathcal{O}(1)$ we should be aware that not all $\mathcal{O}(1)$ are the same.

4 Resolving Collisions

Two common strategies for resolving collisions are:

1. *Closed addressing* aka *open hashing*: Instead of storing the keys (and data) directly in the hash table we store them in an auxiliary structure typically pointed to by the entries in the hash table. We'll look at *separate chaining* as an example.
2. *Open addressing* aka *closed hashing*: We keep the keys (and data) in the hash table and avoid collisions another way. We'll look at *probing* as an example.

For each of these strategies we'll make some comments on their performance. A full mathematical analysis of hash functions can be quite challenging in part because of the number of possibilities.

5 Open Hashing

5.1 Introduction

To reiterate, in open hashing we are "opening" the hash table up and storing our keys not in the table themselves but in auxiliary structures pointed to by the entries in the table. There are many auxiliary structures we could use to store the entries, for example linked lists, trees, and so on.

5.2 Separate Chaining

When using separate chaining the hash table doesn't contain the data but rather contains pointers to linked lists (hence the use of the word "chaining"). That is, for any key x the entry $T[h(x)]$ is a pointer to either *null* or the first node in a linked list. Each node in the linked list contains a key, its data, and a pointer to the next node.

Given a key x we calculate $h(x)$. If $T[h(x)] = null$ then there's nothing associated to $h(x)$ so we create a node with key x and associated data and we point $T[h(x)]$ to it. Later, if we have some x' with $h(x') = h(x)$ we simply add another node to the linked list.

The advantage to this is that we have solved the collision problem. The disadvantage of course is when we search for the data associated to a key x we potentially have to iterate (slowly) through the linked list pointed to by $T[h(x)]$. Moreover each linked list could potentially get rather long.

To search for the data associated with a key x we find the linked list pointed to by $T[h(x)]$ and then we step through it as we would with any linked list until we find our key k and then its associated data. If the linked list is long this can slow the process down and make us wonder why we're using a hash in the first place.

To delete a key x (and its data) we simply find it as with search and delete it as with a standard linked list.

6 Closed Hashing

To reiterate, in closed hashing we are keeping the table "closed", meaning we absolutely must store the keys in the table somewhere and somehow.

6.1 Probing

One idea behind probing is that when we have a collision we simply "probe" the table for an open spot and insert the key (and data) there.

Of course we have to do this in some reliable way which enables searching and deletion.

Here are some approaches to probing with a few comments:

1. *Linear Probing*: If $T[h(x)]$ is occupied we look at:

   $T[h(x)+1]$ and then $T[h(x)+2]$ and then $T[h(x)+3]$ and then ...

   We do this until we find an open spot. Here all sums are mod m.

   Linear probing seems pretty simple but has a major advantage in that the probing procedure is guaranteed to try every entry in T.

   One major issue with linear probing is that as soon as adjacent entries of T start to fill up, the linear probing process ends up running over these strings of adjacent entries repeatedly and the probing chains can get long, which makes for slow operations. This causes an issue known as *primary clustering*.

   The basic way to solve this is to try to use a probing function which "jumps around" in a more "random" manner.

2. *Quadratic Probing*: If $T[h(x)]$ is occupied we look at:

   $T[h(x)+1]$ and then $T[h(x)+4]$ and then $T[h(x)+9]$ and then ...

   We do this until we find an open spot. Here all sums are mod m.

   Quadratic probing can appear more "random" than linear probing which suggests it might help solve the primary clustering issue, however unlike linear probing it turns out that quadratic probing is not guaranteed to try every entry in T. This can be shown using fairly simple examples.

   However we can be somewhat careful:
   Theorem 6.1.1. If we use quadratic probing and if m is prime then the first $\lfloor m/2 \rfloor$ probes are guaranteed to be distinct.

   Proof. By way of contradiction assume that $h(x)+i^2 = h(x)+j^2 \mod m$ where m is an odd prime and $0 \leq i < j \leq \lfloor m/2 \rfloor$.

Then $m \big| (j^2 - i^2)$ so $m \big| (j-i)(j+i)$. Since m is prime it follows that $m \big| (j-i)$ or $m \big| (j+i)$. However since $0 \leq i < j \leq \lfloor m/2 \rfloor < m/2$, with this latter strict inequality because m is odd, both $j-i$ and $j+i$ are less than m and so neither of these divisibility requirements can hold and we have a contradiction. $\mathcal{QED}$

3. A generalization of the above would be to pick a function $p : \mathbb{Z} \to \mathbb{Z}_m$ where m is the size of T and then if $h(x)$ is occupied we look at:

   $T[h(x)+p(1))]$ and then $T[h(x)+p(2)]$ and then $T[h(x)+p(3)]$ and then ...

   We do this until we find an open spot. Here all sums are mod m.

   Some medium-level number theory can be used to show that judicious choices of p and m can result in the probing process trying every entry.

4. *Double-Hashing*: We define $g(x)$ to be another hash function and then if $h(x)$ is occupied we look at:

   $T[h(x)]$ and then $T[h(x)+g(x)]$ and then $T[h(x)+2g(x)]$ and then $T[h(x)+3g(x)]$ and then ...

   We do this until we find an open spot. Here all sums are mod m.

   Notice that this is different from before because our probing function $g(x)$ takes the keys as input whereas the probing function h took integers as input.

   Analysis of double-hashing is challenging but it can be shown that double-hashing is very efficient provided that the hash functions appear random.

To search for the data associated with a key k we use the same approach as insertion. That is, we apply $h(x)$ and then probe as needed until we find our target key.

Deletion is tricky business when using probing. The reason for this is that if we find the entry we wish to delete and simply empty out the table entry (to *null*) the probing function will not be able to "see past it" for any probes which need do.

The typically way to manage this is that when we delete a key (and its data) we put a special entry in the hash table which says something like *deleted*. Then when we are searching such an entry will indicate that the probing should keep going and when are probing for insertion such an entry will indicate that this table entry is available for insert.

There are of course other ways to manage deletion such as shifting other entries on the probe path.

7 Load Management

7.1 Approach

Note 7.1.1. In what follows, we'll use m to denote the size of the hash table and n to denote the number of keys which have been inserted into the hash table at any given instant.

In both cases there is the notion that a hash table could get "too full". With separate chaining we'd like to keep our linked lists short on average (with any auxiliary structure we don't want it to be too full) and with probing (or any other closed hashing approach) we may simply fill up the hash table completely.

One way to solve this issue is to monitor the situation and rebuild the hash table if needed.

We do this as follows:

1. Suppose the hash table has m entries and at any instant we have inserted n keys into it. Define $\lambda = n/m$ to be the *load factor* of the hash table. We'd like to keep λ low, ideally at most 1. Observe that if $\lambda \leq 1$ then we are averaging fewer than one key per hash table entry.
2. We set an acceptable range $[\lambda_{min}, \lambda_{max}]$ of load factors and we require that we keep:
$$0 \leq \lambda_{min} \leq \lambda \leq \lambda_{max} \leq 1$$
3. If we insert a key and find $\lambda > \lambda_{max}$ we rebuild the entire hash table from scratch with a larger m and new hash function.
4. If we delete a key and find $\lambda < \lambda_{min}$ we rebuild the entire hash table from scratch with a smaller m and new hash function.

When we rebuild the hash table from scratch we do this as follows. Assume we have n keys in the hash table.

1. Create a new hash table T' with size:
$$m' = \left\lceil \frac{2n}{\lambda_{max} + \lambda_{min}} \right\rceil$$
Why this value? In such a case the new load factor of the hash table will satisfy:
$$\lambda \approx \frac{n}{2n/(\lambda_{max} + \lambda_{min})} = \frac{n}{2n/(\lambda_{max} + \lambda_{min})} = \frac{\lambda_{max} + \lambda_{min}}{2}$$
This is essentially the average of the minimum and maximum load factors. This give us some leeway for future insertions and deletions. The value on the right will be denoted λ_0 and is called the ideal load factor.
2. Create a new hash function $h' : K \to \mathbb{Z}_{m'}$.

3. Insert all the data from the old table into the new table.

Example 7.1. Suppose our set of keys K consists of all positive integers. We start with a list T indexed from 0 to 3 (a list of length 4) and $h : K \to \mathbb{Z}_4$ by $h(x) = 3x \mod 4$ using linear probing. Note that $m = 4$.

We set $\lambda_{min} = 0.1$ and $\lambda_{max} = 0.6$ and so we must have $0.1 \leq \lambda \leq 0.6$.

We start inserting keys into the hash. For each insert we get the following result:

- Insert $x = 1$: $h(1) = 3(1) = 3$ so $T = [null, null, null, 1]$. Now $n = 1$ and so $\lambda = 1/4 = 0.25$ which is fine.
- Insert $x = 10$: $h(10) = 3(10) = 2$ so $T = [null, null, 10, 1]$. Now $n = 2$ and so $\lambda = 2/4 = 0.5$ which is fine.
- Insert $x = 42$: $h(32) = 3(42) = 2$ but $T[2] \neq null$ so we probe linearly and find $T[0] = null$ so $T = [42, null, 10, 1]$. Now $n = 3$ and so $\lambda = 3/4 = 0.75$ which is too large.

We rebuild:

We create a new hash table T' with size:

$$m' = \left\lceil \frac{2(3)}{0.2 + 0.6} = \frac{6}{0.8} \right\rceil = 8$$

Thus our new T' is indexed from 0 to 7.

We create a new hash function $h' : K \to \mathbb{Z}_8$ by $h'(x) = 5x \mod 8$ using linear probing. Note that $m = 4$.

We then re-insert our values:

- Insert $x = 1$: $h(1) = 5(1) = 5$ so $T = [null, null, null, null, null, 1, null, null]$. Now $n = 1$ and so $\lambda = 1/8 = 0.125$ which is fine.
- Insert $x = 10$: $h(10) = 5(10) = 2$ so $T = [null, null, 10, null, null, 1, null, null]$. Now $n = 2$ and so $\lambda = 2/8 = 0.25$ which is fine.
- Insert $x = 42$: $h(42) = 5(42) = 2$ but $T(2) \neq null$ so we probe linearly and find $T[3] = null$ so $T = [null, null, 10, 42, null, 1, null, null]$. Now $n = 3$ and so $\lambda = 3/8 = 0.3125$ which is fine.

Note that we still have space for more growth.

8 Time Complexity

8.1 Insertion

Theorem 8.1.1. The worst-case amortized asymptotic time complexity of insertion into a hash table is $\mathcal{O}(1)$.

Proof. We'll use the token method to analyze a single run. A single run will start after we have just reallocated, rehashed, and reinserted all the keys, continues while we hash and insert, and will end after the next time we have just reallocated, rehashed, and reinserted all the keys again.

First some notes:

- When we reallocate it's to size $\left\lceil \frac{2n}{\lambda_{min}+\lambda_{max}} \right\rceil$. Define $\lambda_0 = \frac{\lambda_{min}+\lambda_{max}}{2}$ so reallocate to size $\left\lceil \frac{n}{\lambda_0} \right\rceil$. To simplify let's just say $\frac{n}{\lambda_0}$; it's a bit more work if we leave the ceiling in.
- We rebuild when $\frac{n}{m} > \lambda_{max}$ but $\frac{n-1}{m} \leq \lambda_{max}$. We can rewrite these as $n > m\lambda_{max}$ but $n \leq m\lambda_{max} + 1$.

In addition:

- We say hashing one key costs 1 token.
- We say inserting one hashed key costs 1 token.
- Allocation cost equals the size allocated.

Now then:

- A run starts with some number n_p of items in the table. Since we just reallocated we have $m = \frac{n_p}{\lambda_0}$. Thus $n_p = m\lambda_0$.
- A run ends after the next rebuild. At that instant we have $\frac{n_c}{m} = \lambda_{max}$ with the same m as above. Thus $n_c > m\lambda_{max}$ and $n_c \leq m\lambda_{max} + 1$.

Now let's itemize the costs:

- During the run, to cause the rebuild, we hash and insert $n_c - n_p$ keys. The cost is $2(n_c - n_p)$.
- We then reallocate to size $\frac{n_c}{\lambda_0}$. The cost is $\frac{n_c}{\lambda_0}$.
- We then rehash and reinsert all n_c keys. The cost is $2n_c$.

Thus the total cost is:

$$2(n_c - n_p) + \frac{n_c}{\lambda_0} + 2n_c = 4n_c - 2n_p + \frac{n_c}{\lambda_0}$$

If we deposit β tokens per insert and there are $n_c - n_p$ inserts to get through the run, this means we need to choose the smallest possible β so that:

$$\beta(n_c - n_p) \geq 4n_c - 2n_p + \frac{n_c}{\lambda_0}$$

However observe that since $n_c > m\lambda_{max}$ we know that:

$$\beta(n_c - n_p) > \beta(m\lambda_{max} - m\lambda_0)$$

And observe that since $n_c \leq m\lambda_{max} + 1$ we know that:

$$4n_c - 2n_p + \frac{n_c}{\lambda_0} \leq 4(m\lambda_{max} + 1) - 2m\lambda_0 + \frac{m\lambda_{max} + 1}{\lambda_0}$$

But also $1 \leq m$ so that the above line becomes:

$$4n_c - 2n_p + \frac{n_c}{\lambda_0} \leq 4(m\lambda_{max} + m) - 2m\lambda_0 + \frac{m\lambda_{max} + m}{\lambda_0}$$

Thus it suffices to have:

$$\beta(m\lambda_{max} - m\lambda_0) \geq 4(m\lambda_{max} + m) - 2m\lambda_0 + \frac{m\lambda_{max} + m}{\lambda_0}$$

We can cancel all the m and solve for β:

$$\beta \geq \frac{4(\lambda_{max} + 1) - 2\lambda_0 + (\lambda_{max} + 1)/\lambda_0}{\lambda_{max} - \lambda_0}$$

Since the right side is a constant we can set β equal to the right side to get $\mathcal{O}(1)$ amortized cost.

$\mathcal{QED}$

8.2 Deletion

Theorem 8.2.1. The worst-case amortized asympotic time complexity of deletion from a hash table is $\mathcal{O}(1)$.
Note 8.2.1. This proof should be rewritten to align more closely with the previous proof, rewritten on 21 April 2025.

Proof. First some notes:

- When we reallocate it's to size $\lceil 2n/(\lambda_{min} + \lambda_{max}) \rceil$. Since $\lambda_0 = (\lambda_{min} + \lambda_{max})/2$ this means we reallocate to size $\lceil n/\lambda_0 \rceil$. Since we're only interested large n we will just say it's n/λ_0.
- We rebuild when $n/m < \lambda_{min}$. For large enough n it would only be slightly larger so for simplicity we'll say we rebuild with $n/m = \lambda_{min}$.

In addition:

- We say hashing one key costs 1 token.
- We say deleting one hashed key costs 1 token.
- Allocation cost equals the size allocated.

Now then:

- A run starts with some number n_p of items in the table. Since we just reallocated we have $m = n_p/\lambda_0$. Thus $n_p = m\lambda_0$.
- A run consists of hashing and inserting enough keys to cause a rebuild, the necessary reallocation, and the rehashing and reinserting of all the keys in the table.
- A run ends at the next rebuild. At that instant we have $n_c/m = \lambda_{min}$ with the same m as above. Thus $n_c = m\lambda_{min}$.

Now let's itemize the costs:

- During the run, to cause the rebuild, we hash and delete $n_p - n_c$ keys. The cost is $2(n_p - n_c)$.
- We then reallocate to size n_c/λ_0. The cost is n_c/λ_0.
- We then rehash and reinsert all n_c keys. The cost is $2n_c$.

If we deposit β tokens per insert and there are $n_p - n_c$ inserts to get through the run, this means we need to choose β so that:

$$\begin{aligned}
\beta(n_p - n_c) &\geq 2(n_p - n_c) + \frac{n_c}{\lambda_0} + 2n_c \\
\beta(n_p - n_c) &\geq 2n_p + \frac{n_c}{\lambda_0} \\
\beta(m\lambda_0 - m\lambda_{min}) &\geq 2m\lambda_0 + \frac{m\lambda_{min}}{\lambda_0} \\
\beta(\lambda_0 - \lambda_0) &\geq 2\lambda_0 + \frac{\lambda_{min}}{\lambda_0} \\
\beta &\geq \frac{1}{\lambda_0 - \lambda_0}\left(2\lambda_0 + \frac{\lambda_{min}}{\lambda_0}\right)
\end{aligned}$$

Since the right side is a constant which does not depend upon n we can set β equal to the right side to get $\mathcal{O}(1)$ amortized cost.

$\mathcal{QED}$

8.3 Insertion and Deletion Together

Theorem 8.3.1. The worst-case amortized asympotic time complexity of any sequence of insertion and deletion is $\mathcal{O}(1)$.

Proof. Will fill in details soon. $\mathcal{QED}$