COSC 240: Lecture #9: Hash Functions

Introduction to Hash Functions

You have seen hash functions before so I will not spend too much time on their basics. The goal for these notes is to bring you up to speed on a couple of the more advanced (and algorithmically interesting) things you may encounter people doing with hash functions out in the real world.

Basic Dynamic Sets. Before we study hash functions let us quickly first motivate why we need them. To do so, let us define a useful data structure called a *basic dynamic set*. This data structure allows you to INSERT and DELETE elements, as well as SEARCH for elements. You can implement these data structures in many different ways. For example:

- You can store elements in a basic linked list. If the list has n elements then INSERT might only require $O(1)$ steps (i.e., if you add the new element to the head of the list). On the other hand, both SEARCH and DELETE might now require $\Omega(n)$ steps as the element you are searching for or deleting might be at the end of the list and require you to search through the entire things.
- A slightly smarter solution might be to store the elements in a binary search tree. If the tree is well balanced then INSERT, SEARCH and DELETE can all be implemented with only $O(\log n)$ steps. We have eliminated the worst-case step complexities in $\Omega(n)$ that are possible with the linked list. But if we do lots of set operations, then these $\log n$ costs might still add up.

Hash functions are going to allow us to implement basic dynamic sets in a way that on average is more efficient than the above options. There are other uses for hash functions as well. But this will be the main motivation that we will use in these notes. Let's see how they work...

Implementing Basic Dynamic Sets with Hash Tables. To make our life easier, let us assume that every element x that we might encounter when implementing our dynamic set is assigned a key from some *universe* of keys U. In other words, U is just a set of things that we call *keys*. We assume each element x has some key from U. We reference this as $key[x]$ or $x.key$ (different versions of your textbook actually use both types of notations). When we say the keys are "unique," we mean that if $x \neq y$ then $key[x] \neq key[y]$.

Let us also assume that we can take any key from U and interpret it as a natural number. (As is argued in the textbook, this is straightforward for most types of keys. Assuming the key is stored in binary—whether this binary encodes a number, string, or picture—the binary bits can be strung together and interpreted as a base-2 natural number.) This provides us a potentially smarter way to implement a dynamic set:

- Initialize an array $T[1...u_{max}]$ to be NIL in every position, where u_{max} is the largest key in U (where we consider the keys to all be natural numbers).
- To implement INSERT(x) we simply execute $T[key[x]] \leftarrow x$ and to implement DELETE(x) we simply execute $T[key[x]] \leftarrow NIL$. To implement SEARCH(k), for some key $k \in U$, we simply need to check whether or not there is an element stored in $T[k]$.
- All of these operations require just $\Theta(1)$ steps. Much better than the implementations described above!

We call the above strategy a *direct address table*. It seems like a great solution as its worst case step complexities are all constant! There, is however, a problem.

Here's the crucial question: *What are these keys in practice?* Most types of data records have some sort of unique identifier that can play the role of the key. The *key* thing to keep in mind here (ugh) is that the universe U of these keys can therefore be really, really big. Imagine, for example, that we have at most one element for each computer in a network. Therefore, we use each computer's IP address as its unique key. If we are using IPv6, each address requires only 128-bits—so it's a perfectly reasonably sized identifier to use. The *total number* of IPv6 addresses, however, is really large. In particular:

$$|U_{IPv6}| = 340,282,366,920,938,463,463,374,607,431,768,211,456$$

In this example, using the direct address table method would require us to allocate an array of size $340,282,366,920,938,463,463,374,607,431,768,211,456$. This is not a very practical thing to do. Put another way, if your boss sees that you have the following line in your source code:

$$\text{int records}[340282366920938463463374607431768211456] = \{0\};$$

you will probably lose that job.

Fortunately, we might be able to get around these size issues while preserving some of the advantages of the direct address table. The key is to introduce a *hash function*. As you probably learned in data structures, a hash function simply takes a key universe as input and returns values that come from a more manageably-sized range. In more detail, a hash function h defined for key universe U and range size $m > 0$ is of the form:

$$h : U \to \{0, 1, ..., m-1\}.$$

That is, h maps each key in U to an integer from 0 to $m-1$. This allows us to try something like the direct address table except instead of having one entry for each possible entry, we have one entry for each possible value from 0 to $m-1$. In more detail:

- Initialize an array $T[0...m-1]$ to be NIL in every position.[1]
- To implement INSERT(x) we simply execute $T[h(key[x])] \leftarrow x$ and to implement DELETE(x) we simply execute $T[h(key[x])] \leftarrow NIL$. To implement SEARCH(k), for some key $k \in \{0, 1, ..., m-1\}$, we simply need to check whether or not there is an element stored in $T[h(k)]$.
- All of these operations still require just $\Theta(1)$ steps.
- We call this implementation of a dynamic set data structure a *hash table*, as it is a table for storing values where a hash function is used to help figure out addressing.

This seems great. If m is small—say closer to the actual number of elements you expect to add to the dynamic set, and not the size of $|U|$—then the amount of memory required by our hash table is much smaller than with a direct address table.

You should, however, notice a big problem here: what if when using the data structure we encounter two different elements, x and y, such that $h(x) = h(y)$? This messes everything up...

[1] Why are we suddenly shifting our array indexes to start at 0 instead of 1? Many specific hash function implementations use the mod function (i.e., the remainder left over by a division), which return values starting with 0. So it is simpler to think of the array index starting at 0.

When two elements hash to the same location in a hash table we call this a *collision*. Figuring out how to handle these collisions so that the dynamic set operations remain correct is called *collision resolution*. There are several useful collision resolution strategies. Let us review one that is common...

Collision resolution by chaining

The simplest strategy is called *chaining*. We initialize each entry in the hash table T to a pointer that points to the head of an empty linked list (usually doubly-linked). Each table entry points to its own linked list. We can now store multiple elements per entry using the linked list. In more detail:

- To implement INSERT(x) add x to the head of the list in $T[h(key[x])]$. (In more detail, what you really add to the linked list is a node containing a pair that includes $key[x]$ and a pointer to element x elsewhere in memory.)
- To implement DELETE(x), search for x in the list stored in $T[h(key[x])]$. If you find a node pointing to x then delete that node from the list.
- To implement SEARCH(k), scan the full list in $T[h(key[x])]$, looking for a node corresponding to k.
- Notice that INSERT only requires $\Theta(1)$ work, but the worst-case step complexity for both DELETE and SEARCH is now linear in the size of the lists stored in the relevant hash table entry. In the worst case, our hash function might have hashed all n keys inserted so far into the same position in T. A subsequent DELETE or SEARCH for one of these keys might therefore require $O(n)$ steps. This is no better than just implementing the dynamic set with a simple linked list in the first place.

To get a good hash table we need a good hash function—where "good," in this context, means it does a good job of *spreading* out the keys to different values in its range. This can actually be pretty hard to accomplish—especially if we want this type of good property to hold regardless of the sequence of input keys we then encounter.

To be more detailed about our notion of "good," let n be the number of elements inserted into the hash table so far, m the range of the hash function (which is also the number of entries in the hash table), and $\alpha = n/m$ be the *load factor* for these parameters. A more formal way of defining "good" is that we would like the expected size of each linked list in the hash table to be around α. This is, in some sense, the best we can do. If we could accomplish this, then the expected time of SEARCH and DELETE operations would be $\Theta(1 + \alpha)$ (notice, we add 1 to α to capture the constant number of overhead steps required of these operations; if we just said $\Theta(\alpha)$ this becomes an inaccurate result for the case where m is much larger than n and α is a small fraction).

Failed attempt to implement a good hash function. Consider the following hash function implementation: given an input k, return a value selected with uniform randomness from $\{0, 1, ..., m - 1\}$.

It is straightforward to show that if you use this hash function implementation for n INSERT operations, the expected size of each list will be $\Theta(\alpha)$—which is what we want. So what is the problem here? Before continuing to see my explanation below, it is worth taking some time to try to work out for yourself why the above seemingly good hash function is not actually something was can use.

The problem with the above hash function is that it is not repeatable. That is, if I call $f(k)$ twice, both times return a different randomly chosen value. A key property of a hash function is that it must return the same value for $f(k)$ every time it is called with k (this is a property, more generally, of any deterministic function). Our hash table implementation really does require this property of the hash function, as it depends on it to find a given key that was previously inserted.

Your next suggestion might be that we keep track of random choices already made so that if we see the same key multiple times, we use only the random choice made the first time we saw the key. The problem with this fix is that now we have to keep track of all the keys we have seen so far. Assume, for example, that we have node n previous inserts. For each new insert operation, we have to check our list of n previous inserts which will take $O(\log n)$ steps, even if we use a clever balanced search tree data structure. We want the cost of applying the hash function to always be constant.

(An interesting aside: in analyzing cryptographic protocols, people sometimes assume the random hash function implementation described above, along with a *magic* data structure that can check if a key is new or not in constant time. Such a data structure does not exist but pretending it does allow the cryptographer to analyze how a protocol would perform with a perfectly "good" hash function. The idea is that real hash functions behave close enough to this "good" behavior that the analysis is still informative. Cryptographers call this function a *random oracle*. If you see that term, that is what it refers to.)

Hash functions that work well in practice. Here are two hash function definitions that people often use in practice (see Chapter 11.3 for more detail):

- $h(k) = k \mod m$. (Recall that $k \mod m$ returns the remainder left over after diving k by m).
- $h(k) = \lfloor m(k \cdot A \mod 1) \rfloor$, where A is some constant such that $0 < A < 1$. (Notice, the book shows how it is possible to implement these fractional operations quickly with binary words.)

Both of these hash functions can be implemented simply and fast on modern computers. They do not, however, promise to be good. For each function you can come up with a bad set of inputs that will generate lots of collisions. In practice, however, if you choose m and A (in the case of the second option) properly, these functions tend to produce a distribution of keys that is close enough to the load factor.

If we want something that provides us stronger theoretical guarantees, however, we will have to work harder. That is the topic we turn to next...

Universal Hashing

For any fixed hash function f where m is much smaller than $|U|$, an adversary can study f and come up with a selection of n input values that will all hash to the same entry in the hash table. One way to get around this problem is to fix a large collection of different hash functions. Then, after the input values are selected by the adversary, randomly choose one hash function from this collection to use to implement your hash table. If this collection of hash functions is defined properly, then you get a nice property that for any input sequence, there are only a small number of functions in the collection that behave poorly on the sequence. Therefore, if you randomly choose a function from the collection you are likely to choose one that performs well.

This strategy is called *universal hashing*. It is also used in practice and has the nice property that we can analyze its behavior and prove good properties about it. To do so, let us start by restating the above more formally:

Definition. Let $\mathcal{H}$ be a finite collection of hash functions that map some key universe U to range $\{0, 1, ..., m-1\}$. This collection is said to be **universal** if for each pair of distinct keys $k, \ell \in U$, the number of hash functions $h \in \mathcal{H}$ for which $h(k) = h(\ell)$ is at most $|\mathcal{H}|/m$.

Take a moment to stare at this definition and build an intuition for what it says. Generally speaking, a universal collection guarantees that keys collide about as often as you expect if you just randomly assigned them to values.

We now prove the key theorem of universal hashing, which establishes that if given a series of n values to insert, you first randomly select a function from a universal collection, then implement a hash table with that function, it will be, in expectation, a good table. Formally:

Theorem. Fix an arbitrary sequence of n keys to hash into a hash table T of size m that uses chaining as its contention resolution strategy. Assume you choose a hash function with uniform randomness from a universal collection to implement T. After these insertions, for each key $k \in U$:

1. if k has not been inserted in the table, then $E[n_{h(k)}] \leq \alpha$, where $n_{h(k)}$ is the length of the linked list in entry $h(k)$;
2. if k is in the table, then $E[n_{h(k)}] < 1 + \alpha$.

Proof Notes. We won't prove this whole theorem in detail, but we can summarize the key points of the argument:

- Keep in mind in the following that in calculating expectation, the only randomness is the choice of the hash function h from some universal collection.
- For each pair of distinct keys k and ℓ, define the indicator random variable $X_{k\ell}$ that evaluates to 1 if $h(k) = h(\ell)$, and otherwise evaluates to 0.
- By the definition of *universal*, the probability that $h(k) = h(\ell)$ is no more than $1/m$. Therefore, if we apply the definition of expectation to our simple $X_{k\ell}$ random variable we get $E[X_{k\ell}] \leq 1/m$.

  (Make sure you understand the above point before continuing.)
- We now define for each key k the random variable Y_k that describes the total number of keys in the table in entry $h(k)$. To give a formal definition of Y_k, let $keys(T)$ be the set of the n keys inserted into T up to this point. We now give our definition:

$$Y_k = \sum_{\ell \in keys(T), \ell \neq k} X_{k\ell}.$$

  This definition just adds up $X_{k\ell}$ for each key $\ell \neq k$ in the table. Since $X_{k\ell} = 1$ only if k and ℓ hash to the same entry (and is otherwise 0), then this sum returns exactly the count of keys in T that hash to the same value as k—which is exactly the definition of Y_k.
- Now that we have a careful definition of Y_k, we can calculate its expectation. This will then be used as the foundations for proving the two properties claimed by the above theorem statement.

$$\begin{aligned} E[Y_k] &= E\left[\sum_{\ell \in keys(T), \ell \neq k} X_{k\ell}\right] && (1) \\ &= \sum_{\ell \in keys(T), \ell \neq k} E[X_{k\ell}] && (2) \\ &\leq \sum_{\ell \in keys(T), \ell \neq k} 1/m && (3) \end{aligned}$$

There is nothing tricky in the above calculation. In step (1) we just replaced Y_k with its definition (which we established above). In step (2), we applied the *linearity of expectation* theorem we reviewed in class to move the expectation inside the sum. Finally, in step (3), we replaced $E[X_{k\ell}]$ with $1/m$, which is a fact we also established above.

- Now we are ready to prove the two properties claimed in our theorem statement.

  Let us start with the first property which concerns the case where $k \notin keys(T)$. In this case, the size of entry $h(k)$, which we can call $n_{h(k)}$ is exactly Y_k (recall, we defined Y_k to be the number of keys *other than* k that hash to $h(k)$). Therefore: $E[n_{h(k)}] = E[Y_k]$.

  As we established in the previous step, $E[Y_k] \leq \sum_{\ell \in keys(T), \ell \neq k} 1/m$. We do not want a summation in our final answer. It is easy, however, to bound this sum, as we know there are n keys in $keys(T)$ and none of them are k (as that is the case we are considering here). Therefore: $\sum_{\ell \in keys(T), \ell \neq k} 1/m \leq n/m$.

  This n/m factor should look familiar as that is the definition of α. So putting together the pieces we get $E[n_{h(k)}] \leq \alpha$, which is exactly what property one f the theorem claimed.

- Now we consider the second property which concerns the case where $k \in keys(T)$. This is similar to the first case except now we know $n_{h(k)} = Y_k + 1$, where the plus 1 adds in k to the elements stored in $h(k)$ (keep remembering that Y_k counts elements other than k).

  We can now do redo the above calculation with the plus 1. This does not change much. There are, however, two small things new to tackle. First, to handle the extra plus 1 below, remember from the *linearity of expectation* theorem that $E[X + 1] = E[X] + 1$. Second, in the above case we calculated $E[Y_k] \leq n/m$. In this case, we get $E[Y_k] \leq (n-1)/m$, because n is in $keys(T)$ so that reduces the number of times the summation is applied by one. We handle that easily below, however, by simply noting that $E[Y_k] \leq (n-1)/m < n/m$. With this in mind, we calculate as follows:

  $$E[n_{h(k)}] = E[Y_k + 1] = E[Y_k] + 1 < n/m + 1 = \alpha + 1,$$

  which is exactly what is claimed by the second property of the theorem.

Do universal collections exist? Take a moment to step back and consider what we just proved. If you can find a universal collection of hash functions, then by simply picking one at random to use every time you run your program, you get the nice theoretical bound that the expected size of each list is close to the best possible performance (i.e,. every list size near the load factor of α). What is particularly cool about universal hashing is that this property is completely independent of the input. So no matter what type of input you encounter, this strategy will behave well.

The natural follow up question, of course, is whether there actually exist easy to implement universal collections of hash functions. Fortunately, the answer is *yes*. Chapter 11.3 of your textbook contains a section that defines a specific collection of hash functions that are easy to implement and that are (relatively) easy to prove to be universal (though I will spare you that proof for now).

For completeness, let us look at the definition of the universal collection studied in the textbook. You will notice each function in this collection is pretty similar to the practical functions we summarized above.

First, some preliminaries. For some prime number $p > 1$, let $\mathbb{Z}_p = \{0, 1, ..., p-1\}$, and let $\mathbb{Z}_p^+ = \{1, 2, ..., p-1\}$.

To define the universal collection, first pick some large prime number p that is larger than key value that might be passed as input to the hash function. Next, for a given $a \in \mathbb{Z}_p^+$ and $b \in \mathbb{Z}_p$, define the following hash function:

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

Notice, the final $\mod m$ ensures that the value returned is between 0 and $m - 1$. It turns out the collection defined as follows:

$$\{h_{a,b} \mid a \in \mathbb{Z}_p^+ \text{ and } b \in \mathbb{Z}_p\},$$

is universal. That is, if you want to use universal hashing with this particular universal collection, you fix an appropriately large p, then in each execution choose a and b randomly and use the corresponding hash function.

Contention resolution with open addressing

There is one last topic related to hash tables that I want to (briefly) cover: open addressing. The proper way to think of open addressing is as a contention resolution strategy. Above, we looked at hash table implementations that used *chaining* to deal with the case where multiple elements hash to the same entry in the hash table (i.e., a *collision*). Open addressing is an alternative to chaining.

In more detail, the basic idea of of open addressing is to get rid of the linked lists used by chaining as these list create too much memory overhead. Instead, every element inserted into the table is stored directly in the table. That is, each entry in T is either an element or NIL.

This is an efficient use of space. The problem, of course, is what to do when multiple keys hash to the same entry. With open addressing, you handle collisions, when they occur, as follows:

> **Open Addressing Basics.** *When inserting an element with key k, if $T[h(k)]$ is already occupied, go search for another entry in T that is empty, and stash the element there. Do this search in a systematic way, so that if later we need to search for k, we can retrace the steps of the search to find where we stashed the corresponding element.*

The key is figuring out the right way to "search" for an open space. One way to do so is to imagine we extend our hash function so that $h(k)$ now returns a permutation of the set $\{0, 1, ..., m - 1\}$, instead of just one value from this set. This permutation of these values is sometimes called a **probe sequence**.

- Now, when we INSERT an element with key k, we apply $h(k)$ to get a probe sequence $i_1, i_2, ..., i_m$. We first try to insert the element into $T[i_1]$. If that is full, we check $T[i_2]$, and so on. If we find an empty slot, we insert the element. If the whole table is full, we can return a *table overflow* error. (This is a key property of open addressing: because there is no chaining, the table must be large enough to directly fit all elements.)
- To find an element with key k (whether it is for a SEARCH or DELETE), you search $T[i_1]$, then $T[i_2]$, and so on, looking for a table entry with an element stored with key k. Notice, you do not necessarily have to search the whole table. As soon as you encounter some $T[i_j]$ that is empty, you know that there is no element with key k in the table, as it would have been inserted into this spot (or an earlier one) in the probe sequence.

Chapter 11.4 in your text book describes three common ways to generate probe sequences: ***linear probing***, ***quadratic probing***, and ***double hashing***. It is worth reviewing these so you get a sense of what the implementations look like. I will not, however, expect you to memorize their definitions.

Performance of open address hash tables. The final question is whether we expect open addressing to work well as a hash table implementation. Here is the relevant theorem:

Theorem. Given an open address hash table with load factor $\alpha = n/m < 1$, implemented with a "good" hash function, the expected number of probes required in an unsuccessful search is at most $1/(1-\alpha)$.

A few notes about the above:

- We assume $\alpha < 1$ as the hash table requires we have more entries than elements.
- The key metric is how long it takes to find an empty spot when searching through a problem sequence. The expected time of $1/(1-\alpha)$ might not look intuitive at first but this is good. This tells us, for example, that if less than half the table is used then we expect to need only $\Theta(1)$ probes to find an empty slot.
- A conclusion is that if you know a good upper bound on m, and it is not too large, open addressing with n set to a constant factor larger than this m can actually be a really efficient hash table implementation.