Lecture 6: Hashing Variants and Universal Hashing

Readings: CLRS section 11.1-4

Lecture Overview

1. Review: dictionaries, hashing, chaining
2. Universal hashing
3. Open addressing, linear probing, double hashing
4. Consistent hashing

Review

The Dictionary Problem

A Dictionary is an Abstract Data Type (ADT) that maintains a set of items. Each item is associated with a key. It supports the following operations:

- **insert(key, item)**: add item to the set with the given key
- **delete(key)**: remove key from the set
- **search(key)**: return item with key if it exists

We assume that items have distinct keys (or that inserting new ones clobbers old ones).

It is important, at a higher level, to differentiate between the problem, or the abstract data type (or interface, or whatever terminology works for you) and the methods used to solve it. One is basically a set of specifications ("what") and the other is the mechanism used to satisfy those specifications ("how").

Hashing

Our goal is an implementation for the dictionary problem with expected $O(1)$ time per operation and $O(n)$ space complexity where n is the number of keys stored.

Definitions:

- u = number of all possible keys (*e.g.*, all possible MIT IDs, which would be around 10^9)
- $\mathcal{U} = \{0, \ldots, u-1\}$ = universe of keys
- n = number of keys in the table (this is the actual data: # of students in 6.046, say on the order of 10^2)
- m = number of slots in the table (we want this to be on the order of 10^2, and, in general, to be not too much bigger than n)
- $h : \{0, 1, \ldots, u-1\} \to \{0, 1, \ldots, m-1\}$ is a hash function.

Clearly, any function mapping a set with u elements into a set with $m << u$ elements will have collisions. Any implementation of hashing needs to define a way of dealing with collisions. In 6.006, we discussed the idea of "hashing with chaining". See Figure 1 (Please review your notes of 6.006!). We will not discuss hashing with chaining here.

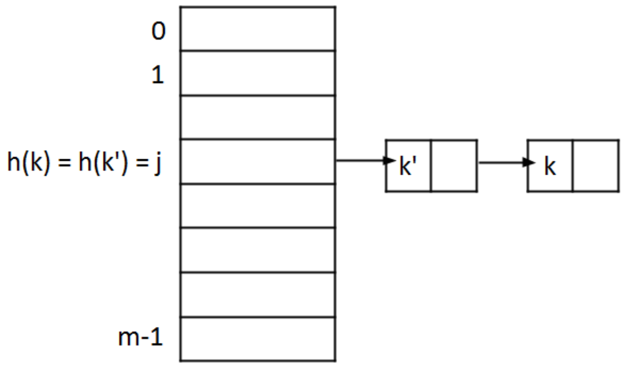


Figure 1: Hashing with chaining

Universal Hashing

An intuitive way of thinking about hash function is to view it as a "random" function. So "good" hash functions should map elements from $\mathcal{U}$ "randomly" into $\{0, \ldots, m-1\}$. Obviously, any hash function is deterministic, so the above statement should not be taken literally. Still, we will try to make this idea operational.

One way of making the idea operational is to demand that the hash function maps two random keys to the same slot with probability similar to a "random" function.

This is the so called **Simple Uniform Assumption**[1]. Practically, this assumption may be satisfactory. If m is very large, we may never see two keys that collide. However, in many applications m is not that large. In many cases, if we have n keys, m is only $O(n)$ so collisions are expected. In these applications, one may be extremely unlucky and have a set of keys which are all mapped to relatively few slots. As theoreticians we definitely don't like to depend on good fortune!

To take matters into our own hands, we use randomness! Specifically, we will use randomness to specify a hash function that works well for **arbitrary** pairs of inputs. So, rather than using one hash function, we work with a *family* of hash functions. We want a family such that a random hash from the family "behaves" like a random function in some sense. To make this idea operational, we define the notion of a **Universal Family of Hash Functions**:

Definition 1 (Universal Hash Family)**.** *A collection $\mathcal{H}$ of hash functions $h : \{0, 1, \ldots, u-1\} \to \{0, 1, \ldots, m-1\}$ is a* Universal Hash Family *if, for* ***any*** *two keys k and k',*

$$\Pr_{h\in\mathcal{H}}\{h(k) = h(k')\} \leq \frac{1}{m}$$

It is **very important** to understand the fact that the probability is taken over the random choice of hash function, and not over a random choice of inputs. Think of randomized algorithms where the probability is not over random inputs, but rather over the random choices of the algorithm.

Where did the above definition comes from? Note that, if h is truly a random function, then, for any two keys, the probability that h maps them to the same slot is exactly $\frac{1}{m}$. So, a family is universal if, in this regard, a random member behaves like a truly random function.

What is the benefit of working with UHF's? One can prove statements about their performance. For example, one can bound precisely the size of the average chain in a chaining implementation of the dictionary problem.

Claim 2. *For n arbitrary distinct keys and random $h \in \mathcal{H}$, where $\mathcal{H}$ is a universal hashing family,*

$$E[\text{ number of keys in a slot }] \leq 1 + \alpha \quad \text{where } \alpha = \frac{n}{m}$$

[1] A hash function h has the SUA if $\Pr_{k\neq k'}\{h(k) = h(k')\} \leq \frac{1}{m}$

Proof. Consider keys $k_1, k_2, ..., k_n$. Let $I_{i,j} = \begin{cases} 1 & \text{if } h(k_i) = h(k_j) \\ 0 & \text{otherwise} \end{cases}$. Then we have

$$\begin{aligned} E[I_{i,j}] &= Pr\{I_{i,j} = 1\} \\ &= Pr\{h(k_i) = h(k_j)\} \\ &\leq \frac{1}{m} \text{ for any } j \neq i \end{aligned}$$

and trivially $E[I_{i,i}] = 1$. Let's look now at a particular slot. WLOG, let's look at the slot of k_1. Then,

$$\begin{aligned} E[\#\text{ keys hashed to the same slot as } k_1] &= E[\sum_{j\geq 1} I_{1,j}] \\ &= \sum_{j\geq 1} E[I_{1,j}] \text{ (linearity of expectation)} \\ &= E[I_{1,1}] + \sum_{j\geq 2} E[I_{1,j}] \\ &\leq 1 + \frac{n-1}{m} \\ &\leq 1 + \frac{n}{m} \end{aligned}$$

□

We can now make the formal statement that, if the dictionary problem is implemented using hashing with chaining, and, if the hash function is chosen randomly from a UHF, then, in expectation, all operations take $O(1 + \alpha)$.

Later in the lecture we will discuss the construction of nice UHF. For now, let's show that such a family exists.

Claim 3. *The family of* ***all*** *function $h : \{0, 1, \ldots, u-1\} \to \{0, 1, \ldots, m-1\}$ is a UHF.*

Proof. Left as exercise. □

Exercise: Prove claim 3.

The above family is deficient. Why? In a given implementation, we choose the hash function from $\mathcal{H}$ and we need to remember which function we are working with. To specify an arbitrary function from the family above, we need to specify an arbitrary function from $\mathcal{U} = \{0, 1, \ldots, u-1\}$ to $\{0, 1, \ldots, m-1\}$. To do this, we need to specify where each element in $\mathcal{U}$ is mapped to. This takes $u \lg m$ space which is much greater than the $O(n)$ space we wanted to use in the first place. Think of the example of U being all the possible MIT IDs and n being the students registered in 6.046 (around 270). We would like to work with space $O(n)$ to store information about the students.

Open Addressing

Open addressing is a different way of dealing with collisions. Whereas in *chaining* we use a different data structure to deal with collisions, in open addressing collisions are stored in the hash table itself!

Different open addressing schemes are defined by different **probing** strategies. A probing strategy dictates which slot to try next if the particular slot being looked at is occupied with a different key. Mathematically, an open addressing hashing scheme is defined by a function (the probing scheme)

$$h : \mathcal{U} \times \{0, \ldots, m-1\} \mapsto \{0, \ldots, m-1\}$$

For a given key k, we first attempt to store k in $h(k, 0)$, if occupied, we try $h(k, 1)$, then $h(k, 2)$ and so on (see Figure 2).

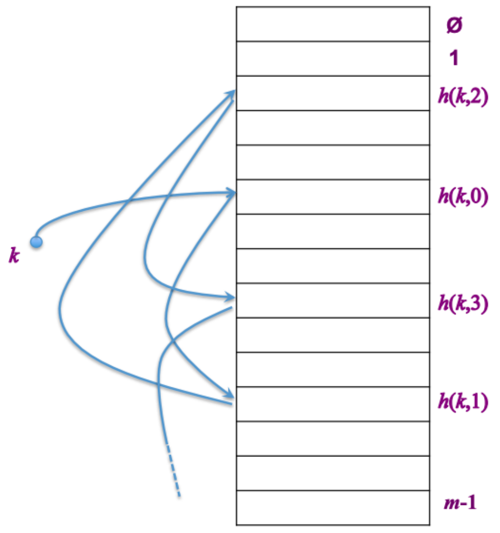


Figure 2: A probing scheme

Exercise: When using open addressing, what happens when the load factor α becomes larger than 1?

Given a fixed probing scheme h, how do we perform an insert, delete or search operation?

1. **Insert:** When inserting a key k, we just follow the probing scheme, in order, until we find an open slot. See Figure 3. In the example, we are inserting key 496.
2. **Search:** When searching for a key k, we again follow the probing scheme until we either find k, or we find an empty slot. If we find an empty slot, we report NOT FOUND.
3. **Delete:** When deleting keys, one needs to use *lazy-deleting*. If one just deletes entries, one may prevent existing keys from being found. Consider the example in Figure 3. If we delete key 586, then we wouldn't be able to find key 496! Rather, instead of deleting an item, we replace it with a special key (say -1). This special key is treated as "occupied" when searching, but as "empty" when inserting.

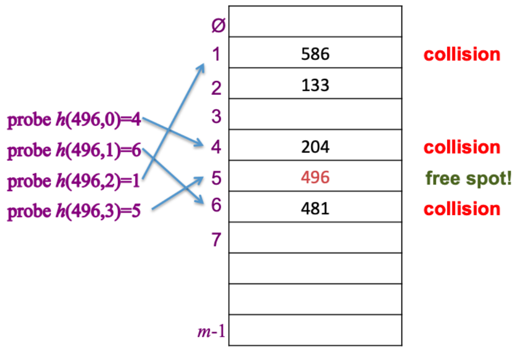


Figure 3: Inserting a key

Assuming that the probing scheme function h can be evaluated in $O(1)$ time, open addressing has similar performance to chaining, with some efficiencies especially for small load factors.

Exercise: What are the pros and cons of open addressing relative to chaining?

Probing schemes

Before discussing some of the known probing schemes, we will discuss an assumption which could help us prove formal statements. It is the analogue of the simple uniform assumption. Even though most probing schemes do not satisfy the assumption, it still gives us something to aim for.

Uniform Hashing Assumption: The probe sequence of a random key is equally likely to be any of the $m!$ permutations of $\{0, \ldots, m-1\}$.

Claim 4. *If the uniform hashing assumption holds for the probing scheme h, then the expected number of probes in an insert is at most $\frac{1}{1-\alpha}$ where α is the load factor.*

Proof. Suppose we want to insert a key k. If the UH assumption holds, then the probing sequence is a random permutation of the slots. One can choose this random permutation in a lazy manner: We pick, *when needed*, the next slot in the sequence randomly among those slots not yet picked. This makes the analysis much simpler. *This is an idea that is worth understanding as it is very powerful.* This principle if often called the *Principle of Deferred Decisions.*

We will also assume that there are already n stored keys. The argument goes as follows: The first probe will be successful with probability $p = \frac{m-n}{m}$ as there are $m-n$ empty slots out of a total of m (note that $p = 1-\alpha$). If not, the second probe will be successful with probability $\frac{m-n}{m-1} \geq p$. This is because, if the first probe was not succesful, we still have $m-n$ open slots out of the remaining $m-1$. In general, the ith probe is successful with probability $\frac{m-n}{m-i+1} \geq p$. So, we have a process that succeeds with probability at least p. The expected number of tries until we succeed is at most $\frac{1}{p} = \frac{1}{1-\alpha}$.

□

Linear probing

We will now discuss some of the standard probing schemes. The first one is the simplest to implement. It is called *linear probing* for obvious reasons.

The linear probing scheme is defined as $h(k,i) = (h'(k) + i) \mod m$ where h' is a regular hash function.

Does the linear probing scheme have the uniform hashing assumption? No! If two keys are mapped to the same initial slot by h', then they will follow the same probing sequence. So there are only m possible permutations instead of the required $m!$.

Linear probing also has the big disadvantage that clusters within the hash table are likely to arise. This is true regardless of the choice of h'. To see this, assume that h' is equally likely to map a given key k to any one of the m slots. Consider the probability that a given empty slot i gets occupied by a new **insert**. If the slot $i-1$ is empty, then the probability is $\frac{1}{m}$. On the other hand, if the j preceding slots are all occupied, then the probability is $\frac{j+1}{m}$ which gets larger and larger.

One can show that if the load factor is constant, the expected size of the longest cluster is $O(\log n)$ (Problem 11-1 in CLRS).

Exercise: Do problem 11-1 in CLRS.

Quadratic probing

A probing scheme that avoids clustering is the so called quadratic probing. For suitable constants c_1 and c_2, one defines $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \mod m$. One can use, for example, $c_1 = c_2 = \frac{1}{2}$ to guarantee that one can reach all the slots. Even though primary clustering is avoided, still, if two keys are mapped to the same slot by h', their probing sequences are identical.

Exercise: Argue that quadratic probing does not satisfy the UH assumption. How many different permutations does quadratic probing generate as the key k varies?

Double hashing

Double hashing is usually a better alternative to the preceding two schemes. We define $h(k,i) = (h_1(k) + ih_2(k)) \mod m$ where h_1 and h_2 are two hash functions. Clearly, one must ensure that no key is mapped to 0 by h_2 (why?). Also, one tries to ensure that, for all k, $h_2(k)$ is relatively prime to m. Otherwise, for that particular key k, one wouldn't reach all slots (why?). This is easily done by using prime m.

To better visualize the scheme, we can take $h_1(k) = k \mod m$ and $h_2(k) = 1 + k \mod (m-1)$ for m prime. Clearly, both requirements are satisfied.

Exercise: Argue that double hashing does not satisfy the UH assumption. How many different permutations does double hashing generate as the key k varies?

Consistent Hashing

We will now look at a multi-billion dollar application of hashing. The motivation is web caching. We would like to cache web pages in servers to facilitate quick access.

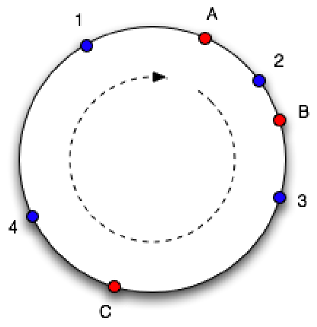
One obvious possibility is to associate servers with the set $\{0, \ldots, m-1\}$, use a hash function h and assign a web page k to the server $h(k)$. If the hash function h is "good", then we can expect the pages to be uniformly distributed over the servers. The problem is that some servers may go down, and some servers may come on-line. So, m may change. For example, initially m may be 101 and $h(k) = k \mod 101$. What happens if we decrease m (after a server goes down)? If we now use $h(k) = k \mod 100$, then possibly all pages will have to be re-assigned to their new hosts.

Consistent hashing provides a web caching solution where, when a server goes down, or a new server comes on-line, only those pages that where in the down server, or the pages that go the the new one, are re-assigned!

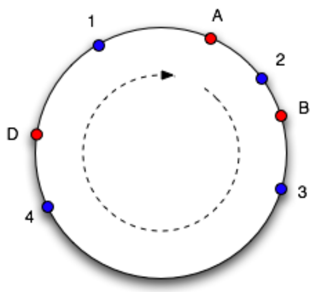
Consistent hashing was invented in 1997 by several MIT people and gave birth to Akamai (now a $16B company).

The idea is beautifully simple. Assume we have n servers. We will take m to be a large number, say $m = 2^{32}$. We will not only hash the keys (web addresses), but also the name of the servers! We will then store the web page with key k in the first server found with a hash $h(s) \geq h(k)$.

For visualization purposes, we can think of the hash table as being in a round circle (we will do $\mod m$ arithmetic). Look at Figure 4a. Dots in red are hashes of servers, dots in blue are hashes of web-pages. Each web-page is located in the server located to its right (clockwise). For example, pages 1 and 4 are located in server A.



(a) Before



(b) After

Figure 4: Consistent hashing Figure ©Tom White

What happens if server C goes down and server D is brought on-line? Now pages 3 and 4 go to the new server D (see Figure 4b). Notice that these were the only pages to be re-assigned.

So far so good. The above scheme, if implementable, and assuming a good hash function, has some good statistical properties. Intuitively, both pages and servers are hashed randomly all over the large ring. We can assume that the servers are placed first in the ring dividing the ring into n intervals. Each server is associated with the interval to its left (counter-clockwise). In expectation, a server associated with an interval of length L will get a fraction L/m of the pages. Do you see why?

Exercise: Show that the expected length of the interval associated with a given server is $\frac{m}{n}$.

As a result, in expectation, each server gets a fraction $\frac{1}{n}$ of pages. That is pretty nice. One can prove that, with high probability, no interval is too large. Unfortunately, one can also show that some intervals will be rather small.

Optional exercise: Prove that, with probability at least $1 - \frac{1}{n}$, no server owns more than a fraction $O(\frac{\log n}{n})$ of the pages. *Hint:* Divide the ring into $\frac{n}{2 \log n}$ intervals. Show that the probability that a given interval does not get a server is at most $\frac{1}{n^2}$. Use the union bound to bound the probability that each interval gets at least one server.

Optional exercise: Interpret the result from the previous exercise through the lens of the Coupon Collector's problem.

Optional exercise: Show that, with constant probability, one server owns no more than a fraction $O(1/n^2)$ of the pages. *Hint:* Divide the ring into n^2 intervals. Use the Birthday Paradox to argue that,with constant probability, some interval gets two servers.

Optional follow-up: One can further decrease the variance of the number of pages per server using the following trick. Each server is hashed l times for some $l > 1$ (say we hash the tuple (name,i) for $i = 1...l$). Each server is now associated with l intervals, each with average length $\frac{1}{ln}$. By linearity of expectation, the average fraction of the ring, call it X, associated with a given server is, as before, $\frac{1}{n}$. However, the variance of X is reduced. Intuitively, X is the sum of l pieces, some larger than the mean and some smaller than the mean, so they average out. One can show that the variance of X decreases by a factor of l relative to the base case.

How do we actually implement such a scheme. Given a key k and its hash $h(k)$, we need to be able to find the server s with the smallest hash $h(s)$ larger than $h(k)$. Ideas?

Recall Balanced Binary Search Trees from 6.006 (such as Red-Black Trees). Such trees offer a successor function such that, when searching a given key k, if the key is not in the tree, we get back the smallest key larger than k. Moreover, as the trees are balanced, operations take $O(\log n)$ time. To implement consistent hashing, we insert the hashes of the servers into a BST. Voila! Given a page with a key k, we search for $h(k)$ in the BST and find the stored key (the hash of a server) with the smallest value larger than $h(k)$.