Chapter 13

Hashing Algorithms

[This chapter is co-written by Jacky Xu.]

In the last chapter we studied many concentration inequalities, from weak bounds like Markov's Inequality, to stronger bounds like the Chernoff bound. In this chapter we will apply these bounds to an important problem in computer science: the design of hashing algorithms.

What exactly is hashing? Let's start with a simple example. Suppose you are the CMU student dean, in charge of maintaining a system that stores academic information on each student, such as the student's name, major, GPA and so on. You decide to use SSN numbers to identify students, so that not anybody can access the information. When a student enters his/her SSN number, the system will search for and return his academic information.

SSN	Academic Info
123456789	Mark Stehlik, Senior, GPA: 4.0
658372934	Tom Cortina, Junior, GPA: 3.5
529842934	David Kosbie, Freshman, GPA: 3.0
623498008	Dilsun Kaynar, Sophomore, GPA: 3.7
...	...

The key feature of the system is that search needs to be fast. Additionally, when new freshmen arrive, you need to insert their information into the system, and when seniors graduate, you need to delete their information from the system.

Suppose there are $m = 20,000$ students. How would you store this collection of student info? One solution is to use a linked list or unsorted array. Then insert is fast, but search and delete need to linearly scan the whole list, which takes $O(m)$ time. A better solution is to use a sorted data structure, such as a binary search tree that sorts student info by SSN number. Then search, insert and delete all take $O(\log m)$ time. None of these solutions are ideal.

Question: Is there a solution with $O(1)$ worst-case time for search, insert, and delete? Suppose, for now, that space is not a consideration.

Answer: If space is not a consideration, one could use a huge array, A, where the SSN is the index

into the array. For example, if Mark's SSN number is 123456789, then his information will be stored in $A[123456789]$. The time for search, insert, and delete is $O(1)$. However, since there are 10^9 possible SSN numbers, the size of A needs to be 10^9. This is a waste of space for storing the info of only $20,000$ students.

Question: Suppose that we're willing to give up on worst-case guarantees. Is there a solution with $O(1)$ *average time* for search, insert, and delete, which uses just $O(m)$ space?

Hint: Here's an idea: Suppose we divide the students into buckets. For example, we could divide students into 10 buckets according to the last digit of their SSN, so all students with SSN ending with 0 go into bucket 0, all students with SSN ending with 1 go into bucket 1, and so on. Then if we want to search for Mark, we know that his SSN belongs to bucket 9, so we need only look within bucket 9. Assuming all bucket sizes are approximately equal, each bucket has about $\frac{m}{10}$ students, and our search time is 10 times faster than the linear solution ... but it's still $O(m)$. Can we take this idea further?

Answer: We can increase the number of buckets, n, to further improve the search time. For example, instead of using just the last digit, we can use the last four digits of the SSN. Then we will have 10000 buckets, with ending digits 0000 to 9999. So, to search for Mark, we need only look within bucket 6789, which, assuming all bucket sizes are equal, has only $\frac{20000}{10000} = 2$ students in expectation. Using $O(m)$ buckets allows us to achieve $O(1)$ search time with $O(m)$ space!

This method is called **bucket hashing**. Hashing makes searching, insertion, and deletion fast in expectation, because we need only search within a single small bucket.

Definition 13.1 *A* **bucket hash function** $h : U \to B$ *maps keys to buckets. For a key k, we call $h(k)$ the* **hash** *of k. The domain of h is U, the universe of all possible keys. The range of h is B, which is a subset of the non-negative integers, denoting the buckets. $K \subseteq U$ is the actual set of keys that we are hashing, where typically $|K| \ll |U|$. Let $|K| = m$ and $|B| = n$. We use r.v. B_i to denote the number of keys that hash to bucket i, also called the "size" of bucket i. The data structure which stores the m keys into n buckets using such a hash function is called a* **hash table**.

In the above example, U is all possible 9-digit SSN's ($|U| = 10^9$), K is the set of the SSN's of the 20,000 students, and B is the 10000 buckets. As is typical, $|K| \ll |U|$, which allows us to get away with a small hash table. When we adjusted the number of buckets above, we were trading off between space and search time. The ratio of keys to buckets is called the load factor.

Definition 13.2 *A hash table that stores m keys within n buckets is said to have a* **load factor** *of*

$$\alpha = \frac{\textit{Number keys}}{\textit{Number buckets}} = \frac{m}{n}.$$

In general, we assume that hash functions have two desirable properties: (i) they are *efficient to compute*, and (ii) they are *balanced* in that the keys are uniformly distributed between the buckets. If we're lucky and the keys are themselves uniformly distributed numbers, then a simple hash function like $h(k) = k \mod n$ can work well. However if the keys come from a more skewed distribution, it can be much harder to find a "balanced" hash function. Finding balanced and efficient hash functions is usually scenario-specific, so we won't dwell on this. For the purposes of analysis we will simply assume that our hash function is efficient and has a "balanced" property knows as the *simple uniform hashing assumption*, defined next.

Definition 13.3 *A bucket hash function h satisfies the* **Simple Uniform Hashing Assumption (SUHA)** *if each key k has probability $\frac{1}{n}$ of mapping to any bucket $b \in B$. Moreover, the hash values of different keys are independent, so for any subset of keys $k_1, k_2, \ldots, k_i \in K$, $k_1 \neq k_2 \neq \cdots \neq k_i$ and $b_1, b_2, \ldots, b_i \in B$, $\mathbf{P}\{h(k_1) = b_1$ and $h(k_2) = b_2$ and $\cdots$ and $h(k_i) = b_i\} = \frac{1}{n^i}$.*

The SUHA assumptions are a lovely analytical convenience, but they may not make much sense. Given that $h(k)$, the hash value of key k, is deterministic, how can we say that $h(k) = b$ with probability $\frac{1}{n}$? This is achieved by using a *universal family* of hash functions $h_1, h_2, \ldots, h_n$. The hash function to be used for a particular hash table is drawn, uniformly at random, from this universal family. Once a hash function, h_i is picked, then that same hash function is used for all they keys of the table. In this way, the hash function is deterministic, but has appropriate random properties. We ignore questions on how to create universal families[1] and instead show how SUHA is used.

Question: Assuming simple uniform hashing, and assuming a load factor of α, what is $\mathbf{E}[B_i]$, the expected number of keys which map to bucket i?

Answer: Assume that there are n buckets and m keys, where $\alpha = \frac{m}{n}$. Let I_k be the indicator random variable that key k maps to bucket i. Then, by Linearity of Expectation,

$$\mathbf{E}[B_i] = \sum_{k=1}^{m} \mathbf{E}[I_k] = \sum_{k=1}^{m} \frac{1}{n} = \frac{m}{n} = \alpha\,.$$

So all buckets have the same size, α, in expectation.

Searching for a student involves hashing their SSN to some bucket i, and then searching through all the keys that mapped to that bucket. Traditionally, the keys that map to a single bucket are stored in a linked list at that bucket. This is called "Bucket Hashing with Separate Chaining," and will be the topic of Section 13.1. In Section 13.2, we will analyze a different way of storing keys that hash to the same bucket, called "Bucket Hashing with Linear Probing."

In both Sections 13.1 and 13.2, the goal is to use hashing to store information in a way that allows for fast search, insert, and delete, both on average and with high probability. In Section 13.3, we will look at an entirely different use of hashing: how to verify the identity of a key without exposing the key (think here of the "key" as being a password that you want to ensure is correct without exposing it to an adversary). This will involve "Cryptographic Hash Functions," where our goal will be to prove that, with high probability, the hashing will not expose the identity of the key. The analysis in all of these sections will rely on concentration inequalities.

13.1 Bucket Hashing with Separate Chaining

In bucket hashing with separate chaining, the hash table is an array of buckets, where each bucket maintains a linked list of keys. Figure 13.1 shows our previous example, where the hash function maps an SSN to the last 4 digits of the SSN. To search for a key within a bucket, we traverse the linked list. To insert a key to a bucket, we first search within the linked list, and if the key does not exist, we append it to the linked list. To delete a key from a bucket, we first search for it within the linked list, and delete it from the linked list if we find it. Thus the time complexity for all operations is dominated by the time complexity for search.

[1] See [15][p.267] for a discussion of how number theory can be used to create a universal family of hash functions.

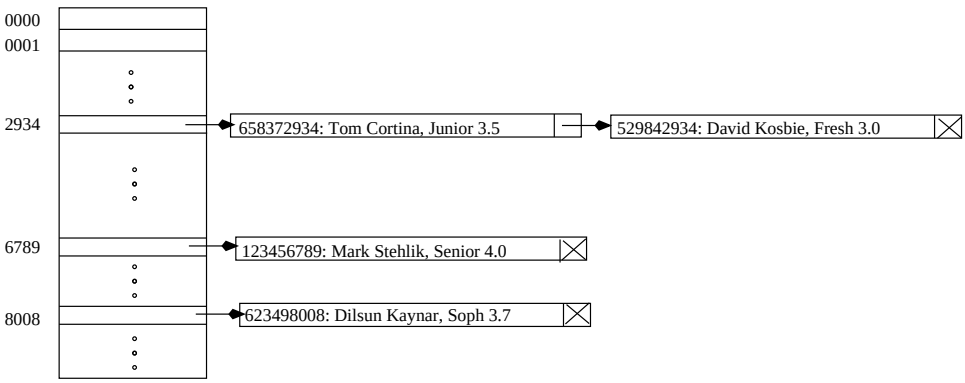

Figure 13.1: *Example of bucket hashing with separate chaining.*

We already saw that, under the simple uniform hashing assumption, and assuming a load factor of α, each bucket has α keys in expectation. Thus the *expected* search time under bucket hashing with separate chaining is $O(\alpha)$. However an individual bucket might have way more than α keys.

Question: What is the distribution on B_i, the number of keys in the ith bucket?

Hint: Remember that we're distributing m keys into n buckets, uniformly at random.

Answer:

$$B_i \sim \text{Binomial}\left(m, \frac{1}{n}\right)$$

Question: When m and n are high, what, approximately, can we say is $\mathbf{P}\left\{B_i > \alpha + 3\sqrt{\alpha}\right\}$?

Answer: Recall from Exercise 4.8, that if m is high and $p = \frac{1}{n}$ is low, then Binomial(n, p) is well approximated by Poisson(np), which in this case says that approximately

$$B_i \sim \text{Poisson}(\alpha) \ .$$

Thus, not only is the mean of B_i equal to α, but its variance is also (approximately) α and its standard deviation is $\sqrt{\alpha}$. We also know by the Central Limit Theorem that B_i approximately follows a Normal(α, α) distribution. Thus,

$$\mathbf{P}\left\{B_i > \alpha + 3\sqrt{\alpha}\right\} \approx 2\% \ .$$

In the remainder of this section, we attempt to derive a high-probability bound on the cost of search. We would like to say that, with high probability, the cost of search is not too high. Since we can't know which bucket we will land in, we would like to say that with high probability, $1 - \frac{1}{n}$, *all* buckets will be small.

Question: I claim that we already derived such a with-high-probability (w.h.p.) bound in the case where $n = m$. What was that?

Answer: In Section 12.4.2 we showed that if you throw n balls into n bins, uniformly at random, then w.h.p. the fullest bin will have $O\left(\frac{\ln n}{\ln \ln n}\right)$ balls. Thus w.h.p. the cost of search is no more than $O\left(\frac{\ln n}{\ln \ln n}\right)$.

In general, however, we are hashing more than n items into n buckets. Theorem 13.4 below tells us that whenever we are hashing at least $\Omega(n \ln n)$ items into n buckets, then not only is the search cost $O(\alpha)$ in

expectation, but it is also $O(\alpha)$ with high probability. This may seem surprising, given that we are not able to show this w.h.p. $O(\alpha)$ result when $m = n$, but we need to keep in mind that our α is also higher now, which gives us more flexibility.

Theorem 13.4 *Under simple uniform bucket hashing with separate chaining, given $m \geq 2n \ln n$ keys, and given n buckets, with probability $\geq 1 - \frac{1}{n}$, the largest bucket has size $< e\alpha$.*

Proof:

Our proof follows along the same lines as that in Section 12.4.2.

We will first prove that for any B_i,

$$\mathbf{P}\{B_i \geq e\alpha\} \leq \frac{1}{n^2}.$$

Then by the union bound,

$$\mathbf{P}\{\text{Some bucket has } \geq e\alpha \text{ balls}\} \leq \sum_{i=1}^{n} \frac{1}{n^2} = \frac{1}{n}$$

So $\mathbf{P}\{$ largest bucket has size $< e\alpha\} > 1 - \frac{1}{n}$ as desired.

Note that since $m > 2n \ln n$, $\alpha = \frac{m}{n} > 2 \ln n > 1$.

To prove $\mathbf{P}\{B_i \geq e\alpha\} \leq \frac{1}{n^2}$, we use the Chernoff Bound from Theorem 12.5 as opposed to Theorem 12.3, because α here is on the order of $\ln n$ not $\Theta(n)$, so it is unlikely that Theorem 12.3 will give a great bound.

Observe that letting $1 + \delta = e$ (so $\delta = e - 1 > 0$) and $\mu = \alpha > 1$ in Theorem 12.5 we have:

$$\begin{aligned}
\mathbf{P}\{B_i \geq e\alpha\} &= \mathbf{P}\{B_i \geq (1+\delta)\mu\} \\
&\leq \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\mu} \\
&= \left(\frac{e^{e-1}}{e^{e}}\right)^{\alpha} \\
&= (e^{-1})^{\alpha} \\
&< (e^{-1})^{2\ln n} \\
&= (e^{\ln n})^{-2} \\
&= \frac{1}{n^2}
\end{aligned}$$

■

Many questions that we considered earlier in the book are also applicable to the analysis of bucket hashing algorithms. For example, empty buckets are undesirable, since they waste space. Thus we might ask: What is the expected number of keys we need to insert until no buckets are empty? Alternatively, what are the expected number of empty buckets, given that we insert n keys into n buckets?

Question: What do such questions remind you of?

Answer: The coupon collector problem!

13.2 Bucket Hashing with Linear Probing

In the previous section, we studied bucket hashing with separate chaining, where each of the n buckets has a linked list ("chain") of keys that have mapped to that bucket. While chaining is easy to explain, it has some practical disadvantages. First storing all those pointers is memory-intensive. Also chaining is not cache friendly; the items in a given list are typically scattered over the memory space. This section presents a more practical bucket hasing solution, called "bucket hashing with linear probing," that doesn't require pointers and is more cache friendly.

The high-level idea behind linear probing is that we store only *one* key in each cell of array B. If multiple keys have the same hash value, they are stored in the first-available cell of the array B. In this way, when searching for a key, one is always reading consecutive cells of an array, which are typically in the same cache line.

Here are the specifics: First, linear probing relies on using an array, B, with size $n > m$, where m is the number of objects stored. Typically $n > 2m$, meaning that $\alpha < 0.5$, where α represents the load factor; this is in contrast with bucket hashing with separate chaining, where in general $\alpha > 1$. When we hash key k, if cell $h(k)$ of B is empty, then we place k into $B[h(k)]$. Later, if another key, k', has the same hash value as k, i.e., $h(k') = h(k)$, then we cannot place k' into $B[h(k)]$. We instead search cell by cell, starting with cell $h(k) + 1$, then cell $h(k) + 2$, and so on, until we find the *first available empty cell*. We then insert key k' into this first available cell. The process of probing consecutive cells to check if they're empty is called **linear probing**.

Question: What do you think happens if we get to the last cell of B and it is full?

Answer: The linear probing wraps around to the first cell. So when we talk about looking at cell $h(k) + 1$, $h(k) + 2$, etc., we're really looking at cell $h(k) + 1 \mod n$, $h(k) + 2 \mod n$, and so on. We will leave off the "$\mod n$" in our discussion to minimize notation.

Question: When searching for a key, k, how do we know that k is not in the table?

Answer: We start by looking at cell $h(k)$, then $h(k) + 1$, and so on, until we come to an empty cell. That is our signal that k is not in the table.

Question: But what if the empty cell was created by a deletion?

Answer: When a key is deleted, we mark its cell with a special character, called a tombstone. The tombstone lets us know that the cell used to be full. Thus cells are never cleared in linear probing. When the number of tombstones gets to be too high, we simply recreate the table from scratch.

For the remainder of this section, we'll be interested in analyzing the *expected cost of search*. Observe that the cost of insert and delete can be bounded by the cost of search. Note that when we say "cost of search" we are referring to the cost of an unsuccessful search – meaning searching for a key that turns out not to be in the array. The cost of a successful search is upper bounded by the cost of an unsuccessful search.

The cost of an (unsuccessful) search for a key, k, is the length of the interval of full cells, starting with $h(k)$. Consider an interval, I, of length ℓ, spanning cells $h(k)$ to $h(k) + \ell - 1$, all of which are full.

Question: Ask yourself: How did interval I become full?

Answer: One possibility is that there were ℓ keys, the first of which hashed to cell $h(k)$, the second of which hashed to cell $h(k)+1$, the third of which hashed to cell $h(k)+2$ and so on.

Another possibility is that there were ℓ keys that all hashed to cell $h(k)$. The first ended up at location $h(k)$. The next at location $h(k)+1$, and so on, with the ℓth ending up at location $h(k)+\ell-1$.

Yet another possibility is that there were ℓ keys, where 2 keys hashed to location $h(k)$, and 2 keys hashed to location $h(k)+2$, and 2 keys that hashed to location $h(k)+4$, and so on.

Question: So far we've only considered possibilities for I becoming full due to keys that hashed into interval I. Can I be affected by keys that hash outside of I as well?

Answer: Yes! I can "inherit" keys from other cells outside I. For example, if two keys hash to cell $h(k)-1$, then one of those keys will end up being stored within I.

Definition 13.5 differentiates between the two possible types of arrivals into an interval, I.

Definition 13.5 *Consider an interval I of length ℓ within array B.*
A key k such that $h(k) \in I$ is said to be a **direct arrival** *to I.*
A key k such that $h(k) \notin I$, but where k ends up getting stored in I is said to be an **indirect arrival** *to I.*
We say that interval I is **full** *if all of its cells are occupied, either by direct or indirect arrivals.*

Let's return to the cost of a search for key k. The cost of a search is $\geq \ell$ if the interval, I, of length ℓ, started by cell $h(k)$ and ending in cell $h(k)+\ell-1$ is full. So it seems that, to understand search cost, we need to understand of the likelihood that an interval of length ℓ is full.

Unfortunately, it is very difficult to understand the probability that an interval I of length ℓ is full, because I might be getting both direct arrivals and indirect arrivals. Analyzing direct arrivals is easy via SUHA, but analyzing indirect arrivals is difficult.

Question: Let D denote the number of direct arrivals into an interval I of length ℓ, assuming that there are m keys being hashed. What is the distribution of D?

Answer:

$$D \sim \text{Binomial}\left(m, \frac{\ell}{n}\right)$$

Fortunately, we know a lot about the Binomial distribution, both its mean and its tail.

Definition 13.6 *We say that an interval I of length ℓ is* **directly loaded** *if it receives $\geq \ell$ direct arrivals.*

Question: Observe that an interval I being full does not imply that it is directly loaded, and the converse is also not true. Explain why.

Answer: An interval I can be full without receiving any direct arrivals, just from inherited keys from indirect arrivals that map to cells before interval I and are moved down to interval I. Likewise, an interval I can be directly loaded by getting ℓ direct arrivals within the second half of I only, leaving the first half of I empty.

It may seem that we're doomed, since we want to understand the probability that an interval I is full, but all we can understand is the probability that I is directly loaded, which is unrelated. However the following theorem will allow us to immediately relate *search cost* to counting *intervals that are directly loaded*; obviating the need to ever worry about the probability that an interval is full.

Theorem 13.7

Search cost for key $k \leq$ Total number of directly loaded intervals containing $h(k)+1$

Proof: Assume that the cost of searching for key k is $s = \ell + 1$. This means that the interval, I, consisting of the following cells $[h(k), h(k)+1, h(k)+2, \ldots, h(k)+\ell-1]$ is occupied, with cell $h(k)+\ell$ being empty; so we have to walk through ℓ occupied cells and 1 empty one to know that k is not in the table.

Let N denote the total number of directly loaded intervals containing $h(k)$. We will show that $N \geq \ell$. Thus $s \leq N+1$ and the theorem is proven.

Figure 13.2 shows a series of scenarios for what our hash table B might look like to generate a search cost of s. The shaded entries denote occupied cells. What's constant in all these scenarios is that the interval I is full, followed by an empty cell. What's different in the scenarios is the number of consecutive cells prior to cell $h(k)$ which are full. In scenario 1, cell $h(k)-1$ is empty. In scenario 2, cell $h(k)-2$ is the first empty cell prior to $h(k)$. In scenario 3, cell $h(k)-3$ is the first empty cell prior to $h(k)$, and so on.

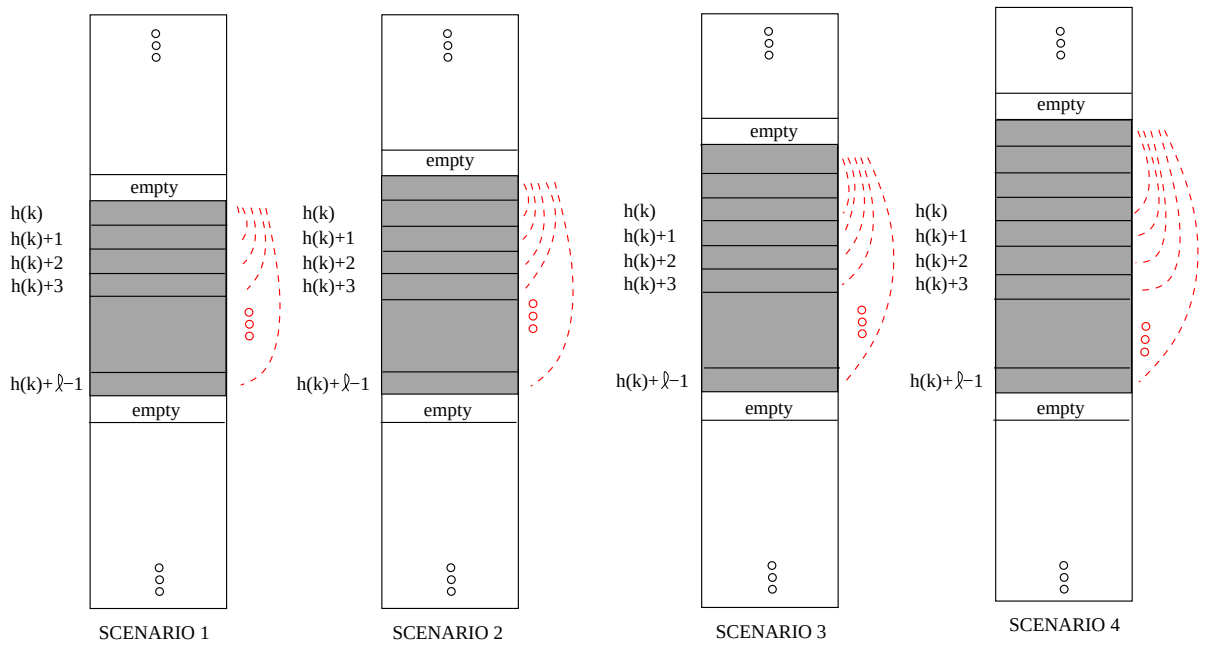

Figure 13.2: *Illustration of different scenarios where the search cost for key k is ℓ. For each scenario, via red dashed curves, we show ℓ full intervals, which contain cell $h(k)$ and which are all directly loaded.*

For each scenario in Figure 13.2, we will argue that $N \geq \ell$.

First consider scenario 1 shown in Figure 13.2, and look at the ℓ red intervals indicated. All of these intervals are full and clearly contain $h(k)$. We now argue that they are all directly loaded. Consider the

first red interval $[h(k), h(k)]$. There must have been a direct arrival to $h(k)$, because the fact that $h(k) - 1$ is empty means that $h(k)$ could not inherit an indirect arrival. Next consider full interval $[h(k), h(k) + 1]$. There must have been ≥ 2 direct arrivals into this interval, because no indirect arrival could have been inherited via $h(k) - 1$. Likewise, looking at interval $[h(k), h(k) + 1, h(k) + 2]$, we can see that there must have been ≥ 3 direct arrivals into this interval. The argument can be continued all the way to interval $[h(k), \ldots, h(k) + \ell - 1]$, which must have received at least ℓ direct arrivals.

Now consider scenario 2. By the same argument as we saw in scenario 1, each successive interval starting with cell $h(k) - 1$ is directly loaded. However, since we only want to count those intervals that contain $h(k)$, we don't bother with the interval $[h(k) - 1, h(k) - 1]$. Again there are ℓ intervals, each of which are directly loaded.

Scenario 3 is exactly the same. Now each successive interval starting with cell $h(k) - 2$ is directly loaded. Again, counting only the intervals that contain $h(k)$ we see that there are ℓ intervals which are directly loaded.

We can continue in this way to scenario 4 and all other possible scenarios involving a full $I = [h(k), h(k) + 1, \ldots, h(k) + \ell - 1]$. In every scenario involving a full I, the same argument holds, and we can see that $N \geq \ell$. ■

Question: In the proof of Theorem 13.7, for each such scenario, we pointed out ℓ intervals which contain $h(k)$ and are directly loaded. Could there be more than ℓ intervals which are directly loaded and contain $h(k)$?

Answer: Yes, there can be more than ℓ intervals. For example, looking at scenario 2, we can consider the interval $[h(k) - 3, h(k) - 2, h(k) - 1, h(k)]$. Imagine that this interval has a direct arrival to cell $h(k) - 3$ and 3 direct arrivals to cell $h(k) - 1$. Then the interval $[h(k) - 3, h(k) - 2, h(k) - 1, h(k)]$ is directly loaded, since it had ≥ 4 direct arrivals. We have not counted all these additional arrivals, which is why Theorem 13.7 represents an upper bound, not an equality.

Let

$$N = \text{Total number of directly loaded intervals containing cell } h(k)$$

Theorem 13.7 says that

$$\mathbf{E}\left[\text{Search cost for key } k\right] \quad \leq \quad \mathbf{E}\left[N\right] + 1 \tag{13.1}$$

But what is $\mathbf{E}[N]$? Let's break up N into N_1, N_2, …, N_m, where N_ℓ denotes the number of length ℓ directly loaded intervals that contain cell $h(k)$. Then

$$N = N_1 + N_2 + \cdots + N_m,$$

and, by linearity of expectation,

$$\mathbf{E}\left[N\right] = \mathbf{E}\left[N_1\right] + \mathbf{E}\left[N_2\right] + \cdots + \mathbf{E}\left[N_m\right],$$

even though these intervals are clearly *not* independent.

Question: What is $\mathbf{E}[N_\ell]$? Can we upper-bound it?

Hint: How many intervals of length ℓ are there that contain cell $h(k)$?

Answer: There are ℓ intervals of length ℓ that contain cell $h(k)$. These are the interval starting at cell $h(k) - \ell + 1$, the interval starting at cell $h(k) - \ell + 2$, the interval starting at cell $h(k) - \ell + 3$, and so on, up to the interval starting at cell $h(k)$. By the simple uniform hashing assumption, each of these ℓ intervals has equal probability of being directly loaded. Thus we can express $\mathbf{E}[N_\ell]$ as

$$\mathbf{E}[N_\ell] = \ell \cdot \mathbf{P}\{\text{length } \ell \text{ interval is directly loaded}\} \ .$$

Recall that the number of keys that hash into a length ℓ interval under simple uniform hashing is

$$D \sim \text{Binomial}\left(m, \frac{\ell}{n}\right),$$

where $\mathbf{E}[D] = m\frac{\ell}{n} = \alpha\ell$. This allows us to apply the Chernoff Bound from Theorem 12.5 to bound $\mathbf{E}[N_\ell]$, as follows[2]:

$$\begin{aligned}
\mathbf{E}[N_\ell] &= (\text{Number length } \ell \text{ intervals}) \cdot \mathbf{P}\{\text{length } \ell \text{ interval is directly loaded}\} \\
&= \ell \cdot \mathbf{P}\{\text{length } \ell \text{ interval is directly loaded}\} \\
&= \ell \cdot \mathbf{P}\{D \geq \ell\} \\
&= \ell \cdot \mathbf{P}\left\{D \geq \frac{1}{\alpha} \cdot \alpha\ell\right\} \\
&= \ell \cdot \mathbf{P}\left\{D \geq \frac{1}{\alpha}\mathbf{E}[D]\right\} \\
&= \ell \cdot \mathbf{P}\left\{D \geq \left(1 + \left(\frac{1}{\alpha} - 1\right)\right)\mathbf{E}[D]\right\} \\
&< \ell \cdot \left(\frac{e^{\frac{1}{\alpha}-1}}{(\frac{1}{\alpha})^{\frac{1}{\alpha}}}\right)^{\alpha\ell} \quad \text{by Theorem 12.5} \\
&= \ell \cdot \left(\frac{e^{1-\alpha}}{(\frac{1}{\alpha})}\right)^{\ell} \\
&= \ell \cdot \left(\alpha e^{1-\alpha}\right)^{\ell}
\end{aligned}$$

[2]We tried using Theorem 12.3 instead, but that bound included an n term which was undesirable since we want the final result to be in terms of α only.

Returning to equation (13.1), we have:

$$\begin{aligned}
\mathbf{E}\left[\text{search cost for } x\right] &\leq \mathbf{E}\left[N\right]+1 \\
&= \sum_{\ell=1}^{m}\mathbf{E}\left[N_\ell\right]+1 \\
&< \sum_{\ell=1}^{m}\ell\cdot\left(\alpha e^{1-\alpha}\right)^\ell+1 \\
&= \alpha e^{1-\alpha}\sum_{\ell=1}^{\infty}\ell\cdot\left(\alpha e^{1-\alpha}\right)^{\ell-1}+1 \\
&< \alpha e^{1-\alpha}\frac{1}{(1-\alpha e^{1-\alpha})^2}+1 \\
&= \frac{\alpha e^{1-\alpha}}{(1-\alpha e^{1-\alpha})^2}+1
\end{aligned}$$

To summarize, we have proved the following theorem:

Theorem 13.8 *Under the simple uniform hashing assumption, the expected cost of (unsuccessful) search in linear probing is at most* $\frac{\alpha e^{1-\alpha}}{(1-\alpha e^{1-\alpha})^2}+1$.

What's important in Theorem 13.8 is that the expected search cost of linear probing is upper bounded by an expression which only depends on α! For example, if we are asked to hash m keys, and we can choose a linear probing hash table with size $n=2m$, where $\alpha=\frac{1}{2}$, then the expected search cost is <27 by Theorem 13.8.

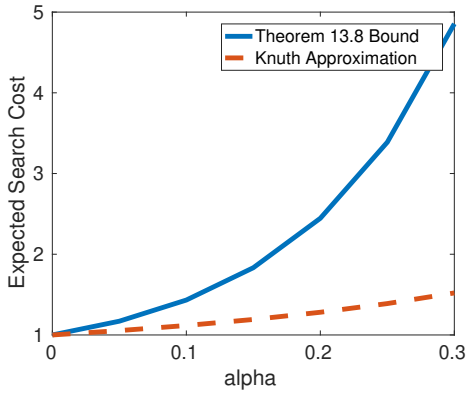

Figure 13.3: *Expected cost of (unsuccessful) search under linear probing, for low load factor* α. *When* α *gets high, we typically resize the table and rehash everything.*

While we prove an *upper bound* on the expected search cost (Theorem 13.8), Knuth [36](Section 6.4, p.537) derives an *approximation* on the expected search cost. Knuth's methods are beyond the scope of this book, but his approximation is quite good:

$$\mathbf{E}\left[\text{Search Cost}\right]\approx\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right).$$

Figure 13.3 illustrates both our upper bound on the expected search cost under linear probing from Theorem 13.8 and also via Knuth's approximation.

13.3 Cryptographic Signature Hashing

Up to now we have only talked about bucket hash functions, whose purpose is to support fast search speed. In this section we will talk about **cryptographic hash functions**. Their purpose has nothing to do with search speed, but rather they are used to encrypt (hide) information, typically passwords.

Suppose you are again the CMU student dean, but this time you are managing services that only CMU students should be able to access. For example a service might be course evaluations at CMU. To access the service, the CMU student enters her andrewID and password, and then the service becomes available. How do you design a system that allows you to check if a student's password is correct for her andrewID? We could store the andrewIDs and corresponding passwords into a database. However, if the database is hacked, then all passwords will be compromised. The solution is to use a cryptographic hash function to hash passwords to signatures, and store signatures in the database instead.

For example, say Mark Stehlik's andrewID is *mstehlik* and his password is 123456. Using a cryptographic hash function, we hash 123456 to a 32 bit signature $0x1b3a4f52$, and store the entry *stehlik:* $0x1b3a4f52$ into the database. The database might look like:

andrewID	signature of password
mstehlik	$0x1b3a4f52$
tcortina	$0x51c2df33$
dkosbie	$0xbb89e27a$
...	...

By looking at the database, you have no idea what passwords correspond to these andrewIDs. Say Mark is trying to log into the SIO service with his andrewID *mstehlik* and password 123456. To verify that Mark's password is correct, we apply a hash function to his entered password, obtaining:

$$h(123456) = 0x1b3a4f52$$

Then we compare $0x1b3a4f52$ to the signature stored under *mstehlik* in the database. Since they're the same, we know that Mark entered the correct password. In this way, we can verify passwords without storing the actual passwords in the database.[3]

In general, when using cryptographic hash functions, we refer to the passwords whose identity we're trying to hide as the *keys*.

Definition 13.9 *A* **cryptographic hash function** $h : U \to B$ *maps* **keys** *to* **signatures**. *For a key k, we call $h(k)$ the signature of k. The domain of h is U, the universe of all possible keys. The range of h is B, denoting all the possible signatures. $K \subseteq U$ is the actual set of keys that we are hashing. Let $|K| = m$ and $|B| = n$.*

[3] In this section, we will not be interested in search time, just in hiding the identity of passwords. However, if we were interested in search time, we could apply a bucket hash to the above table entries to bring the search time down to $O(1)$. It is thus very reasonable to use bucket hashing and cryptographic signature hashing in conjunction.

While U here represents all possible passwords (potentially an infinite number of strings), the set K of actual passwords is much smaller, i.e. $m = |K| \ll |U|$. This point is similar to what we saw for bucket hash tables.

However, recall that in the case of bucket hashing, we typically had $n < m$ and a load factor of $\alpha = \frac{m}{n} > 1$. In contrast, for cryptographic hash functions we will typically use $n \gg m$ so that there are few "collisions." Thus $\alpha \ll 1$.

Definition 13.10 *A* **hash collision** *occurs when two different keys have the same hash value. That is, $h(k_1) = h(k_2)$, where $k_1 \neq k_2$.*

The advantage of storing signatures rather than passwords is that we shouldn't have to worry about our database being hacked. But what if the database is hacked *and* the hash function is discovered? It's important that our hash function is *one-way* or *non-invertible*, meaning that, while it is easy to calculate $h(k)$ from k, it is (almost) impossible to get k from $h(k)$. Consider an example of a bad hash function: $h(k) = k \mod 1000$. Now, if the database is hacked and the hash function is also leaked, then the attacker has a lot of information about the passwords, namely the attacker knows the last 3 digits of every password, making it easier to guess passwords. Even more worrisome, without knowing Mark Stehlik's password, the hacker can use the fact that the signature is, say, 456 to be able to now log into Mark's account using the andrewID *mstehlik* and the password 000456, or any other password ending in 456.

In general, it can be dangerous when multiple passwords map to the same signature because it increases the likelihood that an attacker can guess a password by trying multiple passwords with the same andrewID. We'd ideally like there to be a one-to-one mapping between keys and signatures. Of course this is not possible, even with the best hash function, because $|U| \gg n$, and thus by the Pigeon-hole principle, there exist keys with the same signature. The rest of this section is devoted to analyzing how large n needs to be to achieve a "low" probability of collision, given that m keys are being hashed.

Question: Suppose that we hash m keys using a cryptographic hash function h. Assume the hash space size $|B| = n$. Also assume simple uniform hashing so each key has probability $\frac{1}{n}$ of landing in any given bucket. What is the probability $p(m, n)$ that no collisions occur?

Hint: This should look a lot like a problem that you saw way back in Chapter 3.

Answer: This is the birthday problem from Exercise 3.6. There, we had $m = 30$ people and $n = 365$ possible birthdays, and we looked for the probability of no duplicate birthdays, a.k.a., "no collisions." Repeating that analysis, let A be the event that no collisions occur, i.e., no two keys have the same signature. We imagine that the keys are ordered, from 1 to m. Let A_i be the event that key i has a different signature from each of the first $i - 1$ keys. Now observe that

$$A = \cap_{i=1}^{m} A_i \, .$$

Thus,

$$\begin{aligned}\mathbf{P}\{A\} &= \mathbf{P}\{A_1\} \cdot \prod_{i=2}^{m} \mathbf{P}\left\{A_i \mid \cap_{j=1}^{i-1} A_j\right\} \\ &= 1 \cdot \prod_{i=2}^{m} \left(1 - \frac{i-1}{n}\right) \\ &= \prod_{i=1}^{m-1} \left(1 - \frac{i}{n}\right)\end{aligned}$$

Now

$$1 - \frac{x}{n} \leq e^{-\frac{x}{n}} \tag{13.2}$$

for high n.

This yields the upper bound:

$$\begin{aligned}\mathbf{P}\{A\} &\leq \prod_{i=1}^{m-1} e^{-\frac{i}{n}} \\ &= exp\left(-\frac{1}{n}\sum_{i=1}^{m-1} i\right) \\ &= exp\left(-\frac{m(m-1)}{2n}\right)\end{aligned}$$

Theorem 13.11 *If we use a simple uniform hashing function to hash m keys to a hash space of size n, the probability $p(m,n)$ that there are no collisions is*

$$p(m,n) = \prod_{i=1}^{m-1} \left(1 - \frac{i}{n}\right) .$$

This is upper bounded by:

$$p(m,n) < e^{\frac{-m(m-1)}{2n}} .$$

Assuming that $n > m^2$, the upper bound is very close to exact.

Proof: The only part we have not proven yet is the approximation. Observe that (13.2) is close to an equality when $n \gg x$. In particular, if $n \gg m^2$, then the "upper bound" is a good approximation for each of the m terms in the product of $p(m,n)$. ■

Theorem 13.11 says that

$$\mathbf{P}\{\text{no collisions}\} \approx e^{-\frac{m^2}{2n}} \tag{13.3}$$

Equation (13.3) is interesting because it tells us that we need $m = O(\sqrt{n})$ to ensure that the probability of no collisions is high. In fact, in Exercise 13.2, we'll derive formally that the expected number of keys that we can insert before we get a collision is $\Theta(\sqrt{n})$.

We now use approximation (13.3) to evaluate the effectiveness of the SHA-256 cryptographic hashing algorithm. All you'll need to know for the evaluation is that the hash space B of SHA-256 is all 256-bit numbers.

Question: Suppose we are hashing 10 billion keys using SHA-256. Approximately, what is the probability that there are no collisions?

Answer: Here $m = 2^{10} = 10^{20}$ and $n = |B| = 2^{256} = 10^{77}$. Since $n \gg m^2$, we can use the estimation in (13.3). Thus

$$\mathbf{P}\left\{\text{no collisions}\right\} \approx e^{\frac{-m^2}{2n}} = e^{\frac{-10^{20}}{2\cdot 10^{77}}} \approx e^{-10^{57}} .$$

This is an extremely small number!

Question: Approximately how many keys do we need to hash until the probability that there is a collision exceeds 1%?

Answer: Let

$$p = \mathbf{P}\left\{\text{no collisions}\right\} \approx e^{\frac{-m^2}{2n}}$$

Then $\ln p \approx \frac{-m^2}{2n}$, so $m \approx \sqrt{-2n \ln p}$. Thus, setting $p = 1\%$, we see that, after hashing

$$m = \sqrt{-22^{256} \ln 0.01} \approx 10^{39}$$

keys, we will have a 1% probability of collision.

Question: It turns out that our estimation is a little bit off: we already have a 1% chance of collision after only $m = 5 \cdot 10^{37}$ keys. Suppose a supercomputer can calculate 10^{10} hashes a second, and we have a billion such computers, and a year has about 10^7 seconds. How many years will it take for us to hash enough keys to produce a 1% probability of collision in SHA-256?

Answer: It will take $\frac{5\cdot 10^{37}}{10^{10}\cdot 10^{9}\cdot 10^{7}} = 5 \cdot 10^{11} = 500$ billion years! So it is virtually impossible to find a pair of keys that collides in SHA-256.

13.4 Remarks

This chapter has presented only the briefest discussion of hashing, where our emphasis has been on probabilistic analysis. We have spent no time discussing data structures for implementing hashing. For example, in our discussion of bucket hashing with chaining we assumed that the chain is a linked list, while in practice it would help to use a doubly-linked list. Our discussion of bucket hashing with linear probing is part of a much bigger area called bucket hashing with open addressing, where the probe sequence does not have to involve consecutive cells. By not inserting to consecutive cells, we can avoid a lot of the bunching that we saw in Section 13.2. If we tack on the idealized assumption that the probe sequences are uniformly distributed permutations, then open addressing will perform very well and also become easy to analyze. We therefore defer the analysis to the exercises, e.g., Exercise 13.3. Finally, there are also many more advanced hashing

the numbers we've gotten so far.

$$\begin{aligned}
\mathbf{P}\{K>0\} &= 1 \\
\mathbf{P}\{K>1\} &= 1-\frac{1}{n} \\
\mathbf{P}\{K>2\} &= \left(1-\frac{1}{n}\right)\cdot\left(1-\frac{2}{n}\right) \\
\mathbf{P}\{K>3\} &= \left(1-\frac{1}{n}\right)\cdot\left(1-\frac{2}{n}\right)\cdot\left(1-\frac{3}{n}\right) \\
\mathbf{P}\{K>k\} &= \left(1-\frac{1}{n}\right)\cdot\left(1-\frac{2}{n}\right)\cdot\left(1-\frac{3}{n}\right)\cdots\left(1-\frac{k}{n}\right) \qquad (0\le k\le n-1) \\
\mathbf{P}\{K>n\} &= 0
\end{aligned}$$

It will help to put $\mathbf{P}\{K>k\}$ into a form where we can apply the Ramanujan result:

$$\begin{aligned}
\mathbf{P}\{K>k\} &= \prod_{i=1}^{k}\left(1-\frac{i}{n}\right) \\
&= \prod_{i=1}^{k}\left(\frac{n-i}{n}\right) \\
&= \frac{(n-1)!}{n^k(n-k-1)!} \\
&= \frac{n!}{n^{k+1}(n-(k+1))!}
\end{aligned}$$

Now

$$\begin{aligned}
\mathbf{E}[N] &= 1+\mathbf{E}[K] \\
&= 1+\sum_{k=0}^{\infty}\mathbf{P}\{K>k\} \\
&= 1+\sum_{k=0}^{n-1}\mathbf{P}\{K>k\} \\
&= 1+\sum_{k=0}^{n-1}\frac{n!}{n^{k+1}(n-(k+1))!} \\
&= 1+\sum_{k=1}^{n}\frac{n!}{n^k(n-k)!} \\
&\sim 1+\sqrt{\frac{\pi n}{2}}
\end{aligned}$$

13.3 **[Open Addressing with Uniform Probe Sequences: Upper Bound]**

Under bucket hashing with linear probing, the probe sequence used to insert (and later search for) key k was the consecutive sequence of cells starting with $h(k)$. Open addressing generalizes this idea to allow the probe sequence for inserting key k to be a general sequence of cells, denoted by $\langle h(k,0), h(k,1), h(k,2), \ldots, h(k,n-1)\rangle$, where the probe sequence for key k is some particular

permutation of the n cells in the table. In the ideal world, we'd like to assume that the probe sequence for each key is equally likely to be any of the $n!$ permutations of $\langle 0, 1, 2, \ldots, n-1 \rangle$. This is called *open addressing with uniform probe sequences*. (Obviously the probe sequence corresponding to any particular key k is fixed.) Since each key is inserted via a randomly chosen probe sequence, open addressing with uniform probe sequences does not lead to the long full intervals that we saw in the case of linear probling, and thus the search cost is lower.

Assume that m keys have been inserted into a table with n cells via open addressing with uniform probe sequences. The load factor is $\alpha = \frac{m}{n}$. Prove that the expected cost of an (unsuccessful) search is at most $\frac{1}{1-\alpha}$. [Hint: Let X denote the search cost. Derive an upper bound on $\mathbf{P}\{X > i\}$ for each i, where the upper bound involves α's only. Then integrate to get $\mathbf{E}[X]$.]

Solution: Let X denote the search cost. We will try to determine the tail of X and then integrate that to get $\mathbf{E}[X]$.

$\mathbf{P}\{X > 0\} = 1$ since we always need to probe at least once.

$$\mathbf{P}\{X > 1\} = \mathbf{P}\{\text{First cell we look at is occupied}\} = \alpha$$

Let A_i denote the event that the ith cell that we look at is occupied. Then

$$\begin{aligned}
\mathbf{P}\{X > 2\} &= \mathbf{P}\{\text{First two cells we look at are occupied}\} \\
&= \mathbf{P}\{A_1 \& A_2\} \\
&= \mathbf{P}\{A_1\} \cdot \mathbf{P}\{A_2 \mid A_1\} \\
&= \frac{m}{n} \cdot \frac{m-1}{n-1} \\
&\leq \alpha^2
\end{aligned}$$

Using the chain rule from Exercise 3.5, we have:

$$\begin{aligned}
\mathbf{P}\{X > i\} &= \mathbf{P}\{\text{First } i \text{ cells we look at are occupied}\} \\
&= \mathbf{P}\{A_1 \& A_2 \& \cdots \& A_i\} \\
&= \mathbf{P}\{A_1\} \cdot \mathbf{P}\{A_2 \mid A_1\} \cdot \mathbf{P}\{A_3 \mid A_1 \& A_2\} \cdots \mathbf{P}\{A_i \mid A_1 \& A_2 \& \cdots \& A_{i-1}\} \\
&= \frac{m}{n} \cdot \frac{m-1}{n-1} \cdot \frac{m-2}{n-2} \cdots \frac{m-i+1}{n-i+1} \\
&\leq \alpha^i \\
&= 2^{-k}
\end{aligned}$$

Finally, using Exercise 4.19, we have:

$$\begin{aligned}
\mathbf{E}[X] &= \sum_{i=0}^{n-1} \mathbf{P}\{X > i\} \\
&\leq \sum_{i=0}^{n-1} \alpha^i \\
&\leq \sum_{i=0}^{\infty} \alpha^i \\
&= \frac{1}{1-\alpha}
\end{aligned}$$

13.4 **[Largest Insert Cost for Open Addressing with Uniform Probe Sequences]**

We return to the setup in Exercise 13.3, where we considered open addressing with a uniform probe sequence. Assume that under this setup we store m keys in a size n array with load factor $\alpha = 0.5$. We will prove that for the m keys that have been inserted, the largest insert cost was $O(\log m)$. Note that the insert cost of key k is equal to the length of the probe sequence of k.

(a) For all $i = 1, 2, ..., m$, let

$$p_i = \mathbf{P}\{\text{the } i\text{th insertion requires} > k \text{ probes}\}$$

Show that $p_i < 2^{-k}$.

Solution:

If $k > i$, then $p_i = 0$, because the ith insertion cannot possibly require $> k$ probes.

Let's assume that $k < i$.

At the time of the ith insertion, there are $i - 1$ keys in the table. The load factor of the array at this time is

$$\alpha' = \frac{i-1}{n} < \frac{m}{n} = \frac{1}{2}\,.$$

Then p_i is the probability that the first k cells that we look at are all occupied.

Let A_j denote the event that the jth cell that we look at is occupied. Using the chain rule from Exercise 3.5, we have:

$$\begin{aligned}
p_i &= \mathbf{P}\{\text{First } k \text{ cells we look at are occupied}\} \\
&= \mathbf{P}\{A_1 \& A_2 \& \cdots \& A_k\} \\
&= \mathbf{P}\{A_1\} \cdot \mathbf{P}\{A_2 \mid A_1\} \cdot \mathbf{P}\{A_3 \mid A_1 \& A_2\} \cdots \mathbf{P}\{A_k \mid A_1 \& A_2 \& \cdots \& A_{k-1}\} \\
&= \frac{i-1}{n} \cdot \frac{i-2}{n-1} \cdot \frac{i-3}{n-2} \cdots \frac{i-k+1}{n-k+1} \\
&\leq \left(\frac{m}{n}\right)^k \\
&= 2^{-k}
\end{aligned}$$

(b) Let X denote the length of the longest probe sequence among all m keys. Show that the $\mathbf{P}\{X > 2\log m\} < \frac{1}{m}$.

Solution:

Let X_i denote the length of the probe sequence of the ith key. Then $X = max_{1 \leq i \leq m} X_i$.
Also note $\mathbf{P}\{X_i > 2\log m\} < 2^{-2\log m} = \frac{1}{m^2}$ according to part (a).
Then applying a union bound,

$$\begin{aligned}
\mathbf{P}\{X > 2\log m\} &\leq \sum_{i=1}^{m} \mathbf{P}\{X_i > 2\log m\} \\
&< m \cdot \frac{1}{m^2} \\
&= \frac{1}{m}
\end{aligned}$$

(c) Show that $\mathbf{E}[X] = O(\log m)$.