20 Hashing Algorithms

In the last two chapters we studied many tail bounds, including those from Markov, Chebyshev, Chernoff and Hoeffding. We also studied a tail approximation based on the Central Limit Theorem (CLT). In this chapter we will apply these bounds and approximations to an important problem in computer science: the design of hashing algorithms. In fact, hashing is closely related to the balls-and-bins problem that we recently studied in Chapter 19.

20.1 What is Hashing?

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, and GPA. You use social security numbers (SSNs) to identify students, so that not anybody can access the information. A student's SSN is called a **key**. When the student's SSN is entered, the system returns the student's academic information.

SSN	Academic Info
123456789	Mark Stein, Senior, GPA: 4.0
658372934	Tom Chen, Junior, GPA: 3.5
529842934	David Kosh, Freshman, GPA: 2.7
623498008	Divia Kana, Sophomore, GPA: 3.7
...	...

The main 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. Then search, insert, and delete all take $O(\log m)$ time on average. None of these solutions is ideal.

Question: If space were not a consideration at all, is there a solution with $O(1)$ worst-case time for search, insert, and delete?

Answer: If space is not a consideration, one could use a huge array, A, where the SSN is the index in the array. For example, if Mark's SSN 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 SSNs, 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 that uses just $O(m)$ space?

Hint: Here's an idea: Suppose we divide the students into $n = 10$ buckets according to the last digit of their SSN. Thus 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 2000 students, and our search time is 10 times faster than the single linked list. Can we take this idea further?

Answer: We can increase the number of buckets, n, to further improve the search time. For example, we can use the last *four* digits of the SSN. Then we will have 10,000 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 approximately equal, has only $\frac{20{,}000}{10{,}000} = 2$ students in expectation.

The solution is to use $n = O(m)$ buckets, which allows us to achieve $O(1)$ search time with $O(m)$ space!

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

Definition 20.1 *A* **bucket hash function** $h : U \longrightarrow B$ *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 maps the* m *keys into* n *buckets using such a hash function is called a* **hash table**.

In the above example, U is all possible nine-digit SSNs ($|U| = 10^9$), K is the set of the SSNs of the 20,000 students, and B is the 10,000 buckets. As is typical, $m = |K| \ll |U|$, 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, α.

Definition 20.2 *A hash table that stores m keys within n buckets is said to have a* **load factor** *of*

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

It is typical to aim for a load factor that is a *small constant* above 1.

In general, we assume that hash functions have two desirable properties: (1) they are *efficient to compute*, and (2) 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 $h(k) = k \bmod n$ 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 known as the *simple uniform hashing assumption*, defined next.

20.2 Simple Uniform Hashing Assumption

Definition 20.3 *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 $b \in B$, where $|B| = n$. Moreover, the hash values of different keys are independent, so for any subset of keys $k_1, k_2, \ldots, k_i \in K$, where $k_1 \neq k_2 \neq \cdots \neq k_i$ and $b_1, b_2, \ldots, b_i \in B$,*

$$\mathbf{P}\left\{h(k_1) = b_1 \;\&\; h(k_2) = b_2 \;\&\; \cdots \;\&\; h(k_i) = b_i\right\} = \frac{1}{n^i}.$$

SUHA is a lovely analytical convenience, but it may seem unattainable. 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 the 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: Let B_i denote the number of keys which map to bucket i. Assuming SUHA, and assuming a load factor of α, what is $\mathbf{E}[B_i]$?

Answer: Assume that there are n buckets and m keys, and let $\alpha = \frac{m}{n}$. 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 20.3. In Section 20.4, we will analyze a different way of storing keys that hash to the same bucket, called "bucket hashing with linear probing."

In both Sections 20.3 and 20.4, 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 (w.h.p.). In Section 20.5 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 signature hash functions," where our goal will be to prove that, w.h.p., the hashing will not expose the identity of the key.

20.3 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 20.1 shows our previous example, where the hash function maps an SSN to the last four 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

[1] See [16, p. 267] for a discussion of how number theory can be used to create a universal family of hash functions.

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.

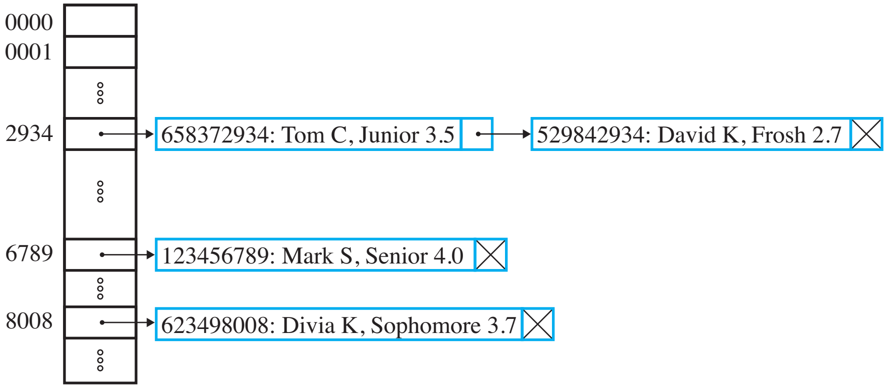
0000
0001
2934
6789
8008
658372934: Tom C, Junior 3.5
529842934: David K, Frosh 2.7
123456789: Mark S, Senior 4.0
623498008: Divia K, Sophomore 3.7

Figure 20.1 *Example of bucket hashing with separate chaining.*

We already saw that, under SUHA, 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 $O(\alpha)$. This is great because we typically imagine that α is a small constant. 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: Assume that m and n are both high, while α is still a constant. What do we know about $\mathbf{Var}(B_i)$?

Answer:

$$\mathbf{Var}(B_i) = m \cdot \frac{1}{n} \cdot \left(1 - \frac{1}{n}\right) = \alpha \cdot \left(1 - \frac{1}{n}\right) \to \alpha.$$

In the setting when m and n are high, CLT tells us that the distribution of B_i approaches that of a Normal.

Question: So, when m and n are high, what, approximately, can we say is $\mathbf{P}\left\{B_i > \alpha + 2\sqrt{\alpha}\right\}$?

Answer: This is the probability that B_i exceeds its mean by more than 2 standard deviations. As the distribution of B_i approaches a Normal, this is approximately 2%.

So the number of keys in any individual bucket is likely to be small. The mean is α and the distribution approaches Normal(α, α) when m and n are high. But what about the *worst* bucket? How many keys does it have?

Question: In the case of $\alpha = 1$, what can we say with high probability (w.h.p.) about the *fullest* bin?

Answer: When $\alpha = 1$, we have $m = n$. In Section 19.4 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. This is a w.h.p. bound on the cost of search when $\alpha = 1$.

We can imagine proving similar w.h.p. bounds on the cost of search for the case when $\alpha = 2$ or $\alpha = 3$. But what happens if α is high, say $\ln n$? One could imagine that the number of keys in the fullest bucket could be quite high now. Theorem 20.4 shows that this is not the case. Both the mean search cost and the w.h.p. search cost are $O(\alpha)$, for high α. Thus for the case where $\alpha = \ln n$, our w.h.p. bound on the cost of search is $O(\ln n)$, which is not that different than the case where $\alpha = 1$.

Theorem 20.4 *Under SUHA, for bucket hashing with separate chaining, assuming $m \geq 2n \ln n$ keys, and n buckets, then with probability $\geq 1 - \frac{1}{n}$ the largest bucket has size $< e\alpha$, where $\alpha = \frac{m}{n}$.*

Proof: Our proof follows along the same lines as that in Section 19.4. The idea will be to first prove that for any B_i,

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

(We will show below how to do this).

Once we have that result, then by the union bound,

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

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

All that remains is to prove that

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

We start by observing that since $m \geq 2n \ln n$, we know that

$$\alpha = \frac{m}{n} \geq 2 \ln n.$$

Applying The Chernoff bound from Theorem 18.6, with

- $1 + \epsilon = e$ (so $\epsilon = e - 1 > 0$), and
- $\mu = \alpha \geq 2 \ln n$,

we have:

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

20.4 Linear Probing and Open Addressing

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. More importantly, chaining is not cache friendly; the items in a given bucket list are typically scattered over the memory space. This section presents a more practical bucket hashing 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 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 when running linear probing, $n > 2m$, 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 the record for key k into $B[h(k)]$. Later, if another key, k', has the same hash value as k, that is, $h(k') = h(k)$, then we cannot place k''s record 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 k''s record 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 occupied?

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

Question: When searching for a key, k, how do we know 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. The empty cell 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, so that we don't stop our search early. Thus, cells are never cleared in linear probing. When the number of tombstones gets 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*. 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 – that is, searching for a key that is not in the array. The cost of a successful search is upper-bounded by the cost of an unsuccessful search.

Unfortunately, bucket hashing with linear probing can often lead to *clustering* (long chains of full cells). Clustering is an artifact of using a linear probe sequence for inserting key k. When inserting key k, if cell $h(k)$ is already full, we next try for $h(k) + 1$, and then $h(k) + 2$. Thus, full cells are likely to be followed by more full cells.

Question: Any idea for how to get around this clustering problem, so that the full cells can be more uniformly spread out?

Answer: We can instead make the probe sequence for key k be a uniformly selected sequence of cells (a particular randomly-chosen *permutation* of cells in the table). Specifically, we denote the probe sequence for inserting key k by:

$$\langle h(k,1), h(k,2), h(k,3), \ldots, h(k,n) \rangle.$$

If key k finds a cell full, instead of trying the next consecutive cell in the array, it now tries the next cell in its probe sequence (its permutation).

In an ideal world, the probe sequence for each key is equally likely to be assigned any one of the $n!$ permutations of $\langle 1, 2, \ldots, n \rangle$. (Obviously the probe sequence corresponding to any particular key k is fixed.) This idea is called **open addressing with uniform probe sequences**. It leads to lower search times than linear probing. While open addressing does require skipping to different locations, at least all of these locations are within the same array, which keeps the pointer cost more reasonable.

Theorem 20.5 *Assume that m keys have been inserted into a table with n cells via open addressing with uniform probe sequences. The load factor is $\alpha = \frac{m}{n} < 1$. Then the expected cost of an (unsuccessful) search is at most $\frac{1}{1-\alpha}$.*

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

$$\mathbf{P}\{X > 0\} = 1 \qquad \text{(we always need to probe at least once)}$$
$$\mathbf{P}\{X > 1\} = \mathbf{P}\{\text{First cell we look at is occupied}\} = \alpha.$$

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

$$\begin{aligned}
\mathbf{P}\{X > 2\} &= \mathbf{P}\{\text{First two cells we look at are occupied}\} \\
&= \mathbf{P}\{A_1 \cap A_2\} \\
&= \mathbf{P}\{A_1\} \cdot \mathbf{P}\{A_2 \mid A_1\} \\
&= \alpha \cdot \frac{m-1}{n-1} \qquad (m-1 \text{ keys and } n-1 \text{ cells remain}) \\
&= \alpha \cdot \frac{\alpha n - 1}{n-1} \\
&< \alpha \cdot \frac{\alpha n}{n} \\
&= \alpha^2.
\end{aligned}$$

Using the chain rule from Theorem 2.10, we have:

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

Finally, applying Theorem 4.9, we have:

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

Theorem 20.5 provides only an *upper bound* on expected search cost. Exercises 20.4 and 20.5 will provide *exact analysis*.

20.5 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, for example, 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 ID 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 ID? We could store the IDs and corresponding passwords in a database. However, if the database is hacked, then all passwords will be compromised.

For example, let's say that Mark Stein's ID is *mstein* and his password is `ILoveToHelp`.

Question: Mark's ID is public. How can we identify Mark via his ID and password without ever storing his password?

Answer: The solution is to use a cryptographic hash function to hash passwords to signatures, and store *signatures* in the database instead.

Using a cryptographic hash function, we hash `ILoveToHelp` to a 32-bit signature $0x1b3a4f52$, and store the entry *mstein:* $0x1b3a4f52$ into the database. Our database might look like Table 20.1.

ID	Signature of password
mstein	$0x1b3a4f52$
tchen	$0x51c2df33$
dkosh	$0xbb89e27a$
dkana	$0x2f85ad73$
...	...

Table 20.1 *Database storing signatures.*

Note: Table 20.1 is *not* a hash table. This is our database. Importantly, by looking at the database, you have no idea what passwords correspond to these IDs. Say Mark is trying to log into the course evaluations service with his ID *mstein* and password `ILoveToHelp`. To verify that Mark's password is correct, we apply a hash function to his entered password, obtaining:

$$h(\texttt{ILoveToHelp}) = 0x1b3a4f52.$$

Then we compare $0x1b3a4f52$ to the signature stored under *mstein* in the database. Since they're the same, we know that Mark (probably) entered the correct password. In this way, we can verify passwords without storing the actual passwords in the database.[2]

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

[2] In this section we will not be interested in the time to search our database, just in hiding the identity of passwords. However, if we were interested in search time, we could apply a bucket hash to the IDs in Table 20.1 to bring the search time down to $O(1)$. It is thus very reasonable to use bucket hashing and cryptographic signature hashing in conjunction.

Definition 20.6 *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 $|B| = n$. Generally,*

$$|U| \gg n \gg m,$$

because U represents a potentially infinite number of strings of any length, and we want $n \gg m$ to avoid collisions. Thus, $\alpha = \frac{m}{n} \ll 1$.

Question: Which of n or m represents the number of entries in the database in Table 20.1?

Answer: The number of entries in the database is m, which is the number of actual keys (passwords) that we're hashing and also represents the number of actual IDs. However, the database is not our hash table. There is no "hash table," but rather just a hash function that maps the m passwords to a space of n possible signatures.

For cryptographic hash functions we typically want $n \gg m$, so that there are few "collisions." Thus, $\alpha \ll 1$.

Definition 20.7 *A* **hash collision** *occurs when two different keys have the same hash value. That is, $h(k_1) = h(k_2)$, where $k_1 \neq k_2$.*

Hash collisions are undesirable. 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 ID.

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 an attacker tries m different passwords (keys). Each of the m keys is hashed, using a cryptographic hash function h, into a hash space of size $|B| = n$, where $n \gg m$. Assume SUHA 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 the birthday problem from Exercise 2.10.

Answer: In the birthday problem, 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, that is, 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

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

Thus,

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

Now, by (1.14),

$$1 - \frac{x}{n} \leq e^{-\frac{x}{n}}, \tag{20.1}$$

where this upper bound is close to exact for high n.

This yields the upper bound:

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

This result is summarized in Theorem 20.8.

Theorem 20.8 (Probability no collisions) *If we use a simple uniform hashing function to hash m keys to a hash space of size n, then the probability that there are no collisions is denoted by $p(m,n)$, where*

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

This is upper-bounded by:

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

Assuming that $n \gg m$, the upper bound is very close to exact.

Proof: The only part we have not proven yet is the tightness of the upper bound. Observe that (20.1) is close to an equality when $n \gg x$. In particular, if $n \gg m$, then the "upper bound" in Theorem 20.8 is a good approximation for each of the m terms in the product of $p(m, n)$. ■

Corollary 20.9 *Assuming* $n \gg m$, $\mathbf{P}\{\textit{no collisions}\} \approx e^{-\frac{m^2}{2n}}$.

Corollary 20.9 is interesting because it tells us that we need $m = \Theta(\sqrt{n})$ to ensure that the probability of no collisions is high. In fact, in Exercise 20.3, we'll derive formally that the expected number of keys that we can insert before we get a collision is $\approx 1 + \sqrt{\frac{\pi n}{2}}$.

We now use Corollary 20.9 to evaluate the effectiveness of the SHA-256 cryptographic hashing algorithm.[3] 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, $m = 10^{10}$, so $m^2 = 10^{20}$, and $n = |B| = 2^{256} = 10^{77}$. Since $n \gg m$, we can use Corollary 20.9. Thus,

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

This is very close to 1, as desired.

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

Answer: Let

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

Then, $\ln p \approx \frac{-m^2}{2n}$, so $m \approx \sqrt{-2n \ln p}$.

Thus, setting $p = 99\%$, we see that, after hashing

$$m = \sqrt{-2 \cdot 2^{256} \ln 0.99} \approx 5 \cdot 10^{37}$$

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

Question: Suppose a supercomputer can calculate 10^{10} hashes a second, and we have one billion such computers, and a year has about 10^7 seconds. How

[3] SHA stands for Secure Hash Algorithm.

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

$$\frac{5 \cdot 10^{37}}{10^{10} \cdot 10^{9} \cdot 10^{7}} = 5 \cdot 10^{11} = 500 \text{ billion years!}$$

So it is virtually impossible to find a pair of keys that collides in SHA-256.

20.6 Remarks

This chapter was written in collaboration with Sheng Xu. The chapter presents only the briefest discussion of hashing, and instead emphasizes the probabilistic analysis. We have spent no time discussing data structures for implementing hashing. Our discussion of bucket hashing with open addressing and uniformly distributed probe sequences allows us to get away with some very simple analysis, which will be made exact in Exercise 20.5. By contrast, the analysis of search time under bucket hashing with linear probing is far harder, but is solved exactly in Knuth's book [46, section 6.4]. Finally, there are also many more advanced hashing schemes, including Bloom filters (see Exercise 20.6), cuckoo hashing [56], consistent hashing [44], and others which we didn't have room to cover, or whose analysis is beyond the scope of the book.

20.7 Exercises

20.1 **Expected hashes until buckets are full**
You are hashing keys, one at a time, into n buckets, where each key has probability $\frac{1}{n}$ of landing in each bucket. What is the expected number of keys hashed until every bucket has at least one key?

20.2 **Inspection paradox: the key's perspective**
You are hashing 100 keys into 100 buckets. One bucket ends up with 20 keys, another bucket ends up with 10 keys, and 70 buckets end up with 1 key each. The remaining 28 buckets end up with zero keys.
(a) From the perspective of the buckets, what is the average number of keys per bucket?
(b) When I search for a random key, on average, how many total keys do I find in the same bucket as my key (including my own key)?
The difference in your answers is the inspection paradox, see Section 5.11.

20.3 **Expected hashes until collision**

You are hashing keys, one at a time, into n buckets, where each key has probability $\frac{1}{n}$ of landing in each bucket. What is the expected number of keys hashed until you get a collision? Use this asymptotic result, proved by Ramanujan [28]:

$$\sum_{k=1}^{n} \frac{n!}{n^k (n-k)!} \sim \sqrt{\frac{\pi n}{2}},$$

to show that your answer grows as $\sqrt{n}$. Notice that you can think of this problem in terms of an n-sided die, where you ask how many times you have to roll the die, in expectation, until you get a number you've seen before. [Hint: You might want to get the mean by summing the tail. This problem will resemble the birthday paradox.]

20.4 **Largest insert cost for open addressing with uniform probe sequences**

Under open addressing with a uniform probe sequence, assume that 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 expected largest insert cost among the m keys was $O(\log_2 m)$. Note that the insert cost of a key is equal to the number of cells probed by the key.

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

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

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

(b) Let X denote the length of the longest probe sequence among all m keys. Show that the $\mathbf{P}\left\{X > 2\log_2 m\right\} < \frac{1}{m}$.

(c) Show that $\mathbf{E}[X] = O(\log_2 m)$. [Hint: Condition via (b).]

20.5 **Open addressing with uniform probe sequences: exact analysis**

In Theorem 20.5, we derived an upper bound on the expected cost of an (unsuccessful) search under open addressing with a uniform probe sequence. In this problem we will derive an exact expression for the expected cost, which is not far from the upper bound in Theorem 20.5. Use the same setup as in Theorem 20.5, again assuming that m keys have been hashed into an array of size n.

(a) First prove two useful lemmas (use counting arguments):

(i) Lemma 1: $\binom{n'}{k}\binom{k}{1} = \binom{n'-1}{k-1}\binom{n'}{1}$.

(ii) Lemma 2: $\sum_{r=1}^{n} \binom{n+1-r}{m-(r-1)} = \binom{n+1}{m}$.

(b) Prove that the probability that an (unsuccessful) search requires exactly r probes is $p_r = \frac{\binom{n-r}{m-r+1}}{\binom{n}{m}}$.

(c) Let U denote the cost of an (unsuccessful) search in this array of m keys. Prove $\mathbf{E}[U] = \frac{n+1}{n-m+1}$.

20.6 **Bloom filter hashing**

[Proposed by Priyatham Bollimpalli] Priyatham is creating new software to check music for copyright violations. For each candidate song, s, if s is the same as an already existing song, the software should output "copyright violation" with 100% certainty (all violations need to be reported). On the other hand, if s is an arbitrary new song, the software should output "new song" at least 99% of the time (it is okay to have a few false alarms).

(a) To maximize efficiency, Priyatham opts for a hash table implementation, with b buckets, where every song, i, is mapped to $h(i)$, which corresponds to one of the b buckets. (Assume that h obeys SUHA, mapping each key i to a uniformly random bucket.) To fill his hash table, Priyatham scrapes all one billion songs in the Internet and maps each to a bucket. Given a candidate song, s, Priyatham's software computes $h(s)$. If $h(s)$ is an empty bucket, the software outputs "new song," otherwise it outputs "copyright violation." Approximately how many buckets b are needed to achieve the desired correctness for an arbitrary song s? Hint: It will help to recall from (1.9) that, for large b,

$$\left(1 - \frac{1}{b}\right)^b \to e^{-1}.$$

(b) After determining that the above scheme uses too much space, Priyatham considers a new approach: He chooses 10 idealized, independent hash functions $h_1, \ldots, h_{10}$ that each map songs to the numbers 1 through 10 billion. He initializes an array A of 10 billion bits, initially set to 0. For each song s that he encounters, he computes $h_1(s), h_2(s), \ldots, h_{10}(s)$, and sets the corresponding indices of A to be 1 (that is, he sets $A[h_1(s)] := 1, A[h_2(s)] := 1$, etc.). Argue that after processing the one billion unique songs, we expect $\approx e^{-1} \approx 0.37$ fraction of the array elements to be 0. [Hint: Linearity of Expectation.]

(c) Now, given a song s, to check if s already exists, Priyatham computes the 10 hashes of s and checks if $A[h_1(s)] = A[h_2(s)] = \cdots = A[h_{10}(s)] = 1$. If so, he outputs "copyright violation," otherwise he outputs "new song." Prove that, if s is actually in your set of one billion songs, you will output "copyright violation" with probability 1. Likewise, if s is *not* in your set of one billion songs, you output "new song" with probability ≈ 0.99. [Hint: Use part (b).]

(d) In the above, we've assumed that the number of buckets (array A's size) is $b = 10$ billion and the number of independent hash functions is $k = 10$. Write a general equation that relates b to k, assuming that the Internet has one billion songs and that we desire no more than 1% false positives.

Note: This space-efficient probabilistic data structure is called a **Bloom filter**. It was conceived by Burton Howard Bloom in 1970 [9].