7. Hashing and Integer Search Structures

This chapter continues our discussion of *dictionary* data structures from the last chapter, only now in the RAM model of computation. Using a *hash table*, we will see how to support the basic dictionary operations *insert*, *delete*, and *find* in only constant time (possibly amortized or in expectation). This improves on the logarithmic time bounds of the comparison-based structures from the last chapter[1], and lets us build extremely fast *set* and *map* data structures, which have wide-ranging applicability. Hash tables, and the underlying technique of *hashing* on which they are based, are absolutely fundamental concepts in computing, with tremendous influence both in theory and practice[2]. Proficiency in hashing is a strong indicator of a well-trained computing professional.

We begin this chapter with a detailed discussion of hash tables, focusing particularly on *universal* hashing, the concept most hash tables have to thank for their strong theoretical performance guarantees. Since hash tables generally only support the basic dictionary operations *insert*, *delete*, and *find*, we next turn our attention to more general RAM-based "search structures" that also support many of the extended operations provided by balanced binary search trees and their relatives from the last chapter (e.g., *find-min / find-max* and *pred / succ*). If our keys are integers in the range $0 \ldots C-1$, we will see how a *radix tree* can implement *insert*, *delete*, and *find*, as well as many extended operations in $O(\log C)$ time, and we will see how to improve this to $O(\log \log C)$ (possibly amortized or in expectation) with the *Y-fast tree* and its equivalent relative, the *van Emde Boas (vEB) structure*. As a consequence, we can now sort in the RAM model in $O(n \log \log C)$ expected time and build a RAM priority queue whose operations take $O(\log \log C)$ time (possibly amortized and/or in expectation).

At the end of the chapter, we show how the more general idea of *hashing* (mapping a complicated object down to a simpler representation, often a single integer) has numerous applications beyond just data structures, in domains like security, distributed algorithms and data management, and the analysis of massive data sets.

[1] Although to be fair, this is a lopsided comparison, since the RAM is a stronger model of computation. Comparison-based structures like binary search trees are essentially the best one can do within the confines of the comparison-based model, where lower bounds on sorting and other related problems pose fundamental obstacles to achieving fast running times.

[2] Hashing has even entered popular culture with the advent of #hashtags, used as dictionary keys behind the scenes to help group together related social media postings.

7.1 Hash Tables

Suppose we want to design a RAM dictionary capable of storing integer keys in the range $0 \ldots C-1$. Since integer keys can be used to index into an array, one solution is to use a *direct access* table $A[0 \ldots C-1]$, where an element with key k is stored in $A[k]$, or if there is no such element, then $A[k]$ would be set to some designated "uninitialized" value. Although this structure is trivial to implement and supports *insert*, *delete*, and *find* all in $O(1)$ worst-case time, it has a critical drawback: *memory usage*. Since C is usually very large (often larger than our entire memory), the direct access table is usually not feasible due to space considerations[3]. Ideally, a dictionary storing n keys should occupy only $\Theta(n)$ space.

Rather than storing our n elements in a direct access table of size C, we instead use a much smaller *hash table* $A[0 \ldots m-1]$ where typically $m = \Theta(n)$. Each element is mapped to a location in the table using a *hash function*, $h(k)$, taking a key $k \in \{0, \ldots, C-1\}$ and mapping it to a table index $h(k) \in \{0, \ldots, m-1\}$. To find an element with key k, we would therefore inspect $A[h(k)]$. We usually assume that $h(k)$ can be evaluated in $O(1)$ time. However, we can also design hash functions that work with larger items and hence take longer to evaluate — say, if we want to store or index a collection of text strings or even large files. We discuss methods for hashing large objects later in the chapter, in Section 7.5.5. For now, we continue to assume that our keys are just integers in $0 \ldots C-1$.

Hashing works quite well except for the case when two elements are mapped to the same location, known as a *collision*. Collisions are unavoidable in any function mapping a large range down to a smaller range, so we should always expect collisions no matter what hash function we use. Most of the difficulty with hash tables lies in dealing with collisions gracefully — although collisions don't cause a properly-designed hash table to malfunction, they usually do slow it down. To minimize collisions in the first place, a good hash function spreads elements across the m locations in our table as haphazardly as possible. The less predictable our function, the better it generally performs. This explains the origin of the term "hashing", since the word "hash", when used as a verb, can mean to jumble or mix up. Hash functions that feel "random" are generally good, and as we will see shortly, randomization is indeed often used to build effective hash functions.

7.1.1 Collision Resolution

There are several good ways to deal with collisions. We discuss perhaps the two simplest here, both of which happen to sound like rather unpleasant forms of torture: *probing* and *chaining*. Both work well and are popular methods in practice, though chaining tends to be easier to analyze mathematically.

Resolving Collisions with Probing. With probing (also called *open addressing*), the hash table is a single array of size $m \geq n$. As shown in Figure 7.1, our n elements of data are each stored directly in this array, with the key of each element stored alongside it. Unused table entries are marked as such during initialization. To

[3] One might also worry about the time required to initially set all the entries to "uninitialized", although in theory we could use virtual initialization (problem 2) to reduce this from $\Theta(C)$ down to $O(1)$ at the expense of even more memory.

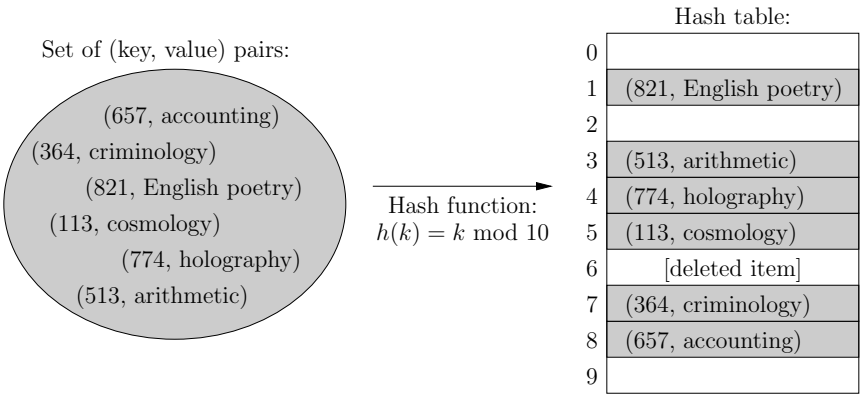


FIGURE 7.1: An example of a hash table where linear probing is used to resolve collisions. Our elements of data are (key, value) pairs representing subject categories from the Dewey decimal system. The element formerly in position 6 of the hash table has been deleted, and we can deduce that this must have occurred *after* the element (364, criminology) was inserted, since otherwise this element would have filled the slot left open by the deleted element. If next we were to insert (874, Latin lyric poetry), then we would probe locations 4 and 5 before finally placing this new element in position 6.

insert a new element e with key k, we start at index $h(k)$. If unused, we store e in this position. Otherwise, we scan forward sequentially until we find an unused location, wrapping around back to the beginning if we scan past the end of the table. To find an element with key k, we start scanning from index $h(k)$ until we locate an element with our desired key, or until we reach an unused entry, in which case we conclude k is not present in the table. Deletion is straightforward except for one small subtlety: we cannot simply delete an element by marking its array entry as "unused", as this might break subsequent *find* operations, causing them to stop scanning too early. Instead, we mark the entry of our dearly-departed element with a so-called *tombstone* annotation, telling *find* operations to keep scanning, but instructing *insert* operations to stop and re-use the entry.

Probing works well for a sparsely-filled table. However, once n grows almost as large as m, large "clumps" of consecutive elements can appear, leading to increasingly slow performance, since larger clumps attract larger numbers of collisions. To alleviate this problem, it is sometimes recommended to use fancier "probing patterns". For example, the approach above uses *linear probing*, where we examine array entries $(h(k) + i) \bmod m$ for $i = 0, 1, 2, \ldots$ in sequence. Instead, we may consider probing locations $(h(k) + g(i)) \bmod m$, where g is a more complicated function; for example, *quadratic* probing uses $g(i) = ai^2 + bi$, where a and b are constants. Alternatively (see problem 109), we could probe locations $h_1(k), h_2(k), \ldots$ according to a sequence of different hash functions that are either implicitly defined or explicitly generated and stored on demand. By making the probe sequence skip around in a more haphazard fashion, we may be less prone to generate large clumps. However, this can also lead to rather poor cache performance, so it not at all a foregone conclusion that there will be any improvement in practice. Empirical testing is advised.

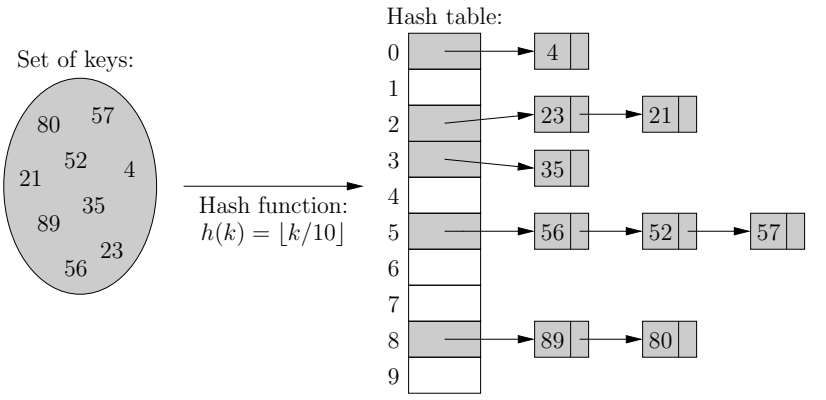


FIGURE 7.2: An example of chaining used to resolve collisions. Each table entry points to a linked list of elements hashing to that entry.

Problem 107 (Batch Construction with Linear Probing.). Suppose we wish to insert n elements into a hash table of size $m = \Theta(n)$ ($m \geq n$) using linear probing. Analyzing the time required for n successive calls to *insert* is tricky and also highly dependent on our choice of hash function. However, please show how we can build the hash table in $\Theta(n)$ time and space in a "batch" fashion, irrespective of our hash function. [Solution]

Depending on our choice of hash function and probing strategy, mathematical running time analysis can be somewhat challenging. In a few pages, we will have the tools to create hashing schemes based on probing where *insert* and *find* both run in $O(1)$ expected time (these two running times always match, since inserting an element requires the same amount of probing as it takes to subsequently find it). Deletion of an element takes only $O(1)$ worst case time, assuming that we adopt our usual convention of defining *delete* so it takes a pointer directly to an element in the table, to separate its functionality from *find*.

Resolving Collisions with Chaining. In a hash table using *chaining*, each entry points to a linked list containing all of the elements hashing to that entry, as shown in Figure 7.2. To insert an element, we hash its key to an index in the table, then *prepend* (not *append*) the element to its linked list. This takes $O(1)$ worst-case time as long as we can evaluate our hash function in $O(1)$ time. Deletion takes $O(1)$ worst-case time given a pointer directly to an element, since we can just unlink the element from its enclosing list (here, we would need to use doubly-linked lists or end our lists with dummy sentinel elements).

To *find* an element with key k, we search the linked list attached to the entry $h(k)$ to see if we find an element with key k. Assuming we can hash in constant time, this takes $O(1 + L)$ total time if we end up searching a list of length L. In a perfect world, our hash function would distribute our n elements uniformly throughout our size-m table, so every linked list would have length at most $\lceil n/m \rceil$, and *find* would therefore run in $O(1 + n/m)$ time. In reality, this is too much to hope for in the worst case, but when we study universal hashing shortly, we will see how to use

randomization to guarantee an $O(1 + n/m)$ *expected* running time for *find*, which becomes $O(1)$ expected time if we ensure that $m = \Theta(n)$.

7.1.2 Guidelines for Designing a Good Hash Function

The first line of defense against collisions is our choice of hash function. In the next section, we will design hash functions that give strong *provable* performance guarantees for *insert*, *delete*, and *find*, allowing us to design hash tables that perform well both in theory and in practice. However, one also finds "ad hoc" hash functions used quite commonly in practice, and despite their lack of provable guarantees, these usually perform reasonably well as long as they follow a few general guidelines.

A good hash function should fully utilize "all the bits" of a key. For example, the natural function $h(k) = \lfloor mk/C \rfloor$ that just linearly rescales $0 \ldots C-1$ down to $0 \ldots m-1$ places too much emphasis on the "higher-order" part of a key. Keys close together in value will tend to collide, which is quite undesirable. We generally want $h(k)$ and $h(k+1)$ to differ substantially and unpredictably. Moreover, C and m are often both powers of 2, in which case this function completely ignores the lower-order bits of k. Another natural function, $h(k) = k \bmod m$, places too much emphasis on the "lower-order" part of a key. Keys spaced out by multiples of m will collide, and this can be particularly bad if m is a power of 2, since then the hash function completely ignores the higher-order bits in a key. A close relative prone to the same issue is $h(k) = (ak) \bmod m$ for some parameter a. In fact, this variant can be dangerous for another reason — as we will see in Chapter 25, if a and m share common factors, then $(ak) \bmod m$ can only output a subset of the indices $\{0, \ldots, m-1\}$ in our table. For example, if a and m are both even, then $(ak) \bmod m$ must also be even.

The "universal" hash functions we will introduce shortly give several nice examples of simple functions that circumvent the pitfalls above.

7.1.3 Choosing the Size of a Hash Table

Since we usually do not know in advance the number of elements n we will end up storing, we typically use amortized table expansion and contraction (Section 4.4) to maintain the size of our table always at $m = \Theta(n)$. This ensures that *find* runs in $O(1)$ expected time with both chaining (if we use universal hashing) and probing (if we use appropriate hash functions and keep m at least a small constant factor larger than n). For example, if n grows too large, we might re-hash the contents of our table into a newly-allocated table of size, say, $2m$. Note that rehashing is usually necessary when m changes, since most hash functions depend on m, so changing m might change where every element is mapped[4].

Rebuilding a table takes only linear time[5], so by our standard amortized rebuilding analysis, it contributes only $O(1)$ additional amortized time for each call to *insert* or *delete*. We must therefore add "amortized" qualifiers to these operations.

[4] Hash functions that do not require substantial rehashing when resizing a table are sometimes known as *consistent hash functions*. We discuss these later in the chapter.

[5] This is immediate with chaining since *insert* only takes $O(1)$ time, and with probing, we get this by using the result of problem 107.

7.2 Universal Hashing

If we want to prove any sort of reasonable performance bounds for hashing, we generally cannot use worst-case analysis. Assume $C > m(n-1)$, which is typically true, since C is usually much larger than both m and n. In this case, the "pigeonhole principle" tells us that for any hash function, we can always find a bad set of n keys that all hash to the same location. Otherwise, if at most $n-1$ key values hash to each of the m table locations, then the total number of possible keys C can be at most $m(n-1)$. Using this set of n bad keys will turn our hash table into essentially a glorified linked list, with $\Theta(n)$ worst-case performance for *find*. To prove good performance bounds for hashing, we therefore need to leave worst-case analysis behind and consider instead the expected performance of hash functions that involve randomly-chosen parameters.

The Infeasibility of Fully-Random Hashing. At the beginning of our algorithm's execution, suppose we select a hash function uniformly at random from all possible functions mapping $\{0, \ldots, C-1\}$ down to $\{0, \ldots, m-1\}$. Since this function maps n/m expected elements to each table entry[6], *find* now runs in $O(1+n/m)$ expected time when we resolve collisions with chaining, which is good news. The bad news is that it takes $\Theta(C)$ space to represent a truly random hash function, to remember where each of the C possible input keys should be mapped. With this amount of space usage, we might as well use a direct access table!

Partially-Random Hashing. To overcome the memory issues above, we choose a partially-random hash function specified by a small number of random parameters. Such a function is often "random enough" for our analytical needs if it satisfies *strong universality*, one of the most important concepts in hashing:

> A randomized hash function h is *strongly universal* if for any pair of keys $k_1 \neq k_2$ and any two table locations y_1, y_2, we have
>
> $$\mathbf{Pr}[h(k_1) = y_1 \text{ and } h(k_2) = y_2] = O(1/m^2),$$
>
> where the probability is over the selection of random parameters built into the function.

For example, if we randomly choose integers a and b from $\{0, 1, \ldots, mC-1\}$, then

$$h(k) = \left\lfloor \frac{(ak+b) \bmod (mC)}{C} \right\rfloor$$

is strongly universal, since $\mathbf{Pr}[h(k_1) = y_1 \text{ and } h(k_2) = y_2] \leq 1.25/m^2$ for any two keys $k_1 \neq k_2$ and any two table locations y_1, y_2. [Proof]

A fully-random hash function would map a given key k_1 to each table location with probability $1/m$, and it would map a pair of keys $k_1 \neq k_2$ to any two specific table locations with probability $1/m^2$. Strongly universal hashing provides this same guarantee of *pairwise independence*, only asymptotically, since we allow a

[6] For a randomized hash function to work, we need to ensure that the keys being inserted into our hash table have no dependence on our random choice of hash function. That is, we cannot have a malicious adversary that peers inside the state of our algorithm *after* it has chosen its hash function, and then chooses a bad set of keys for it.

probability bound of $O(1/m^2)$ (some definitions of strong universality are more strict, using $1/m^2$). Hence, strongly universal hashing behaves like fully random hashing as long as we only look at pairs of keys at a time. If we add a third key to the picture, its mapping might exhibit some dependency[7] on the mappings of the first two keys.

By taking a union bound over all m pairs of locations $y_1 = y_2$, strong universality implies a slightly weaker property known simply as *universality*:

> A randomized hash function h is *universal* if for any pair of different keys $k_1 \neq k_2$, we have
>
> $$\mathbf{Pr}[h(k_1) = h(k_2)] = O(1/m),$$
>
> where the probability is over the selection of random parameters built into the function.

Now for the punchline: universal hashing (and therefore also strongly universal hashing) guarantees that when we resolve collisions in a hash table with chaining, *find* runs in $O(1 + n/m)$ expected time. This is easy to prove using linearity of expectation [Proof]. If we use amortized table resizing to maintain $m = \Theta(n)$, a chained universal hash table therefore supports *insert* and *delete* both in $O(1)$ amortized time, and *find* in $O(1)$ expected time.

7.2.1 Acing Your Next Job Interview

Now that we know the fundamentals of universal hashing, it is worth stepping back for a moment to look at some simple yet common problems we can now solve easily and efficiently. These sorts of problems tend to appear quite frequently in job interviews for computing positions, since job interviewers need to differentiate (e.g., "hash") candidates according to their computational problem solving skills, and knowledge of hashing is an excellent metric to use in this regard.

For example, suppose you are asked during an interview: "given an array containing n integers, how many distinct numbers are present if we remove duplicates?" You could answer this concisely by saying you would insert the numbers one by one into a universal hash table, skipping over the numbers that already exist in the table. The number of elements stored in the hash table at the end gives the desired answer. In terms of running time, we make at most n calls to *insert*, each running in $O(1)$ amortized time, and we make n calls to *find*, each running in $O(1)$ expected time. The total running time is therefore $\Theta(n)$ in expectation. To ensure the job is yours, you could mention that using a balanced binary search tree instead of a hash table would give a running time of $O(n \log n)$, which is optimal in the comparison-based model since any faster algorithm would allow you to circumvent the $\Omega(n \log n)$ worst-case lower bound on element uniqueness. If you want to ensure landing the

[7] Many strongly universal hash functions are specified by two random parameters (like a and b above), so if we knew the locations y_1 and y_2 where two keys k_1 and k_2 are mapped, we could then solve the two equations $h(k_1) = y_1$ and $h(k_2) = y_2$ to determine these parameters, which in turn would determine where a third element would be mapped. Soon, we will see how to build hash functions with r-wise independence by incorporating r random parameters.

job and a high salary offer, you could also mention the technique we will discuss later in Section 7.5.1 for how to estimate the answer to this problem when the hash table above would be too large to fit in memory.

There are many common problems taking $\Omega(n \log n)$ worst-case time in the comparison or real RAM models that we can now solve in linear expected time with universal hashing. Examples include finding the intersection, difference, or union of two sets (problem 42), detecting whether two arrays are *anagrams* (having the same count of each distinct element; problem 54), and re-arranging the contents of an array so as to group equal elements together (Section 3.3.3). Many of these are favorites for job interviewers as well. You may want to pause for a moment to make sure you can now formulate and analyze the hashing-based solutions to these problems with ease.

7.2.2 Further Examples of Universal Hash Functions

Several universal families of hash functions have been studied in the literature. The one we mentioned earlier is particularly simple and fast to evaluate, especially if m and C are powers of two (as is typical), since the division and remainder operations can be implemented with right shifts and bitwise ANDs.

Multiplicative Hashing. If m and C are powers of two, another option is to choose a random odd integer $a \in \{1, 3, 5, \ldots, C-1\}$, after which

$$h(k) = \left\lfloor \frac{ak}{C/m} \right\rfloor \mod m$$

is universal (but not strongly universal), since it satisfies $\mathbf{Pr}[h(k_1) = h(k_2)] \leq 2/m$ for any $k_1 \neq k_2$ [Proof]. This is sometimes called *multiplicative* hashing in the literature, and it can also be evaluated very quickly, using right shifts for division and a bitwise AND for the remainder operation.

Tabulated Hashing. Continuing to assume m and C are powers of two, let us regard any key k as a binary string of $b = \log C$ bits $k_1 k_2 \cdots k_b$. Suppose for each digit $i \in \{1, \ldots, b\}$ that we independently construct a 2-element table H_i whose entries $H_i[0]$ and $H_i[1]$ are randomly chosen from $\{0, \ldots, m-1\}$. Then

$$h(k) = H_1[k_1] \oplus H_2[k_2] \oplus \ldots \oplus H_b[k_b],$$

where $\oplus$ denotes the XOR operation, is strongly universal. Otherwise stated, we hash each of the bits of our key independently in a completely random fashion, and XOR the results together. This is known as *tabulated* hashing, due to its use of small lookup tables, and it works equally well if we break our key into larger chunks rather than individual bits. For example, a very practical way to hash a 32-bit integer k is to regard it as a sequence of four bytes $k_1 k_2 k_3 k_4$ (each in the range $0 \ldots 255$). We can then use the hash function

$$h(k) = H_1[k_1] \oplus H_2[k_2] \oplus H_3[k_3] \oplus H_4[k_4],$$

where each of the H_i's is a length-256 array containing random integers independently chosen during preprocessing from $\{0, \ldots, m-1\}$. This function is also strongly universal. [Proof that tabulated hashing is strongly universal]

Linear Hashing. The earliest universal hash functions described in the literature were also quite simple, although they require selection of an arbitrary (not necessarily random) prime number $p \geq C$ during initialization (we will see how to generate prime numbers in Chapter 25). If we now choose a random integer $a \in \{1, 2, \ldots, p-1\}$, then

$$h(k) = (ak \bmod p) \bmod m,$$

is universal (but not strongly universal), with $\mathbf{Pr}[h(k_1) = h(k_2)] \leq 2/m$ for any two keys $k_1 \neq k_2$. The related function

$$h(k) = ((ak + b) \bmod p) \bmod m,$$

with b randomly chosen from $\{0, \ldots, p-1\}$, is strongly universal; we will prove shortly that $\mathbf{Pr}[h(k_1) = h(k_2)] \leq 1/m^2$ for any two keys $k_1 \neq k_2$. We call this the *linear* hashing method. Note that as opposed to multiplicative and tabulated hashing, linear hashing makes no assumptions about m and C being powers of two.

Problem 108 (Universal Versus Fully-Random Hashing). Recall from problem 19 that if we randomly map n elements to a size-n table, then with high probability each entry receives $O(\log n)$ elements ($O(\log n / \log \log n)$, with a more careful analysis). Strongly universal hashing gives us a much weaker guarantee.

(a) Please design a strongly-universal hash function for mapping n elements into a size-n table that always sends $\Omega(\sqrt{n})$ elements to some table entry. [Solution]

(b) Please show that with probability at least $1/2$, any universal hash function mapping n elements into a size-n table sends $O(\sqrt{n})$ elements to every entry. Hint: what is the expected total number of collisions? [Solution]

7.2.3 Higher Degrees of Independence

A hash function is strongly r-universal if given any set of r different keys $k_1 \ldots k_r$ and any r table locations $y_1 \ldots y_r$,

$$\mathbf{Pr}[h(k_i) = y_i \text{ for all } i \in \{1, \ldots, r\}] = O(1/m^r).$$

While strong universality (the special case where $r = 2$) gives a hash function with (asymptotic) pairwise independence, strong r-universal hashing gives (asymptotic) r-wise independence. We can easily generalize the linear hashing method above to create a strongly r-universal function by using a higher-degree polynomial:

$$h(k) = ((a_{r-1}k^{r-1} + \ldots + a_1 k + a_0) \bmod p) \bmod m,$$

where $p \geq C$ is an arbitrary prime number, and each a_i is chosen independently at random from $\{0, \ldots, p-1\}$. [Proof of strong universality, also for the special case of linear hashing]

We will see another use of high-degree polynomial hash functions in a few pages when we use them to hash large objects.

Probing Revisited. It was only recently shown that 5-wise independence (e.g., using a degree-4 polynomial hash function) suffices to obtain $O(1)$ expected running

times for *insert* and *find* when we use linear probing, as long as we keep m at least some small constant factor larger than n [Somewhat complicated details]. This level of independence is necessary since some 4-wise independent hash functions can lead to $\Omega(\log n)$ expected time guarantees.

Problem 109 (Probing with Successive Universal Hash Functions). Another probing approach that behaves well in theory uses multiple simple hash functions instead of one 5-wise independent function. Consider probing locations $h_1(k), h_2(k), \ldots$ where the h_i's are universal hash functions generated and stored on demand. Please show that this leads to an $O(1)$ expected running time for *insert*, and hence also for *find* (as long as we keep m at least some small constant factor larger than n). Please also show that we only generate $O(\log n)$ hash functions with high probability. [Solution]

7.3 Collision-Free (Perfect) Hashing

With universal hashing, the probability of two different keys colliding is at most $\frac{c}{m}$, for some constant c. In a set of n keys, a union bound over all $\binom{n}{2}$ pairs of keys gives

$$\mathbf{Pr}[\text{any pair of keys collides}] \leq \frac{c}{m}\binom{n}{2} \leq \frac{cn^2}{2m}.$$

Setting the table size to $m = cn^2 = \Theta(n^2)$, we therefore see zero collisions with probability at least $1/2$.

A hash table supporting *find* in $O(1)$ *worst-case* time is called a *perfect* hash table. According to the analysis above, we can build a *static* perfect hash table on n keys (not supporting further insertions after it is built) if we are willing to tolerate $\Theta(n^2)$ space. To do this, we universally hash n elements into a table of size $m = cn^2$, stopping and repeating the entire process with a different randomly-chosen universal hash function if we find even a single collision. Since the probability of success is at least $1/2$, we expect at most two such attempts.

The analysis above is essentially the same analysis we conducted for the famous "birthday paradox" (problem 17), which states that if everyone is randomly assigned one of m possible birthdays, then $n \approx \sqrt{m}$ is the rough threshold for the size of a group of people where we start to see shared birthdays. In the context of (universal) hashing, this result says that for a hash table of size m, we expect to start seeing our first collisions after inserting roughly $n \approx \sqrt{m}$ elements.

Achieving Linear Space. A clever two-level hashing approach can reduce the space used by our static perfect hash table from $\Theta(n^2)$ to $\Theta(n)$. We first universally hash n elements into a size-n table. There will almost certainly be collisions, which we resolve by chaining for the moment. For each index i in our table (say, containing b_i elements), we then use the approach above to build a collision-free second-level universal hash table of size $\Theta(b_i^2)$ on its elements in $\Theta(b_i^2)$ expected time. As shown in Figure 7.3, we can now *find* an element in in $O(1)$ worst-case time with two hash lookups, one at the top level and the other in a second-level table.

It takes $\Theta(n)$ space and time to build the top-level table, and $\Theta(\sum_i b_i^2)$ space and expected time to build the second-level tables. Universality of the top-level hash table implies that $\mathbf{E}[\sum_i b_i^2] = \Theta(n)$ [Simple proof], so the entire process takes linear

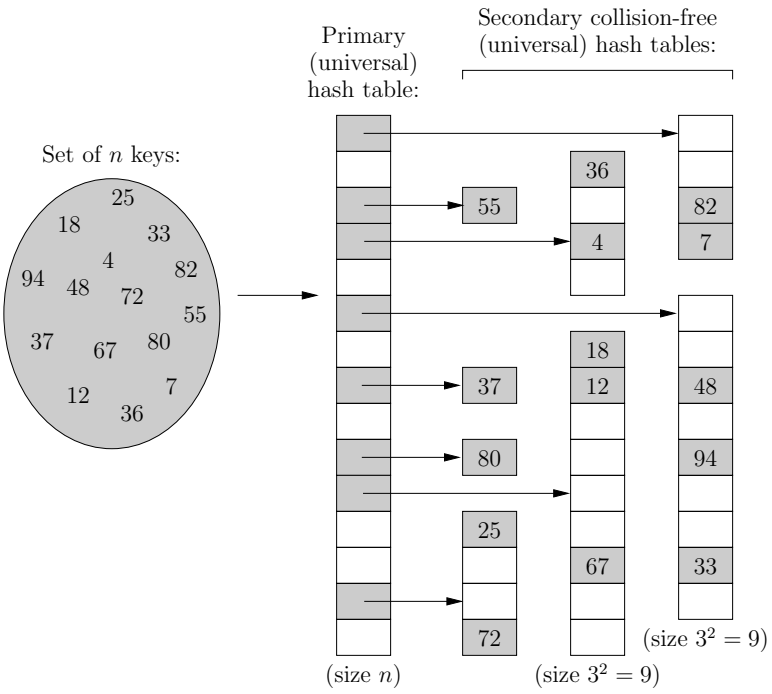


FIGURE 7.3: A static perfect hash table. Collisions in the primary universal hash function are resolved by hashing into collision-free secondary tables. If cell i contains b_i elements, then the second-level table attached to cell i has size $cb_i^2 = \Theta(b_i^2)$; we assume $c = 1$ above.

expected space and time. It is easy to convert the space bound to $\Theta(n)$ in the worst case, preserving linear expected construction time, by re-selecting the parameters in our top-level function until $\sum_i b_i^2$ turns out sufficiently small. [Details]

Problem 110 (Bucket Sort). Suppose we are sorting n integers $A[1 \ldots n]$, each drawn independently from the uniform distribution $[0, C-1]$, where C is a multiple of n (this is just for simplicity of analysis; it is not a fundamental restriction). Let us sort A as follows: first create an array of n buckets $B[0 \ldots n-1]$, and then "hash" each element $A[i]$ to the bucket $B[\lfloor nA[i]/C \rfloor]$ (so the first $1/n$ fraction of the values of $[0, C-1]$ end up in bucket $B[0]$, the next $1/n$ fraction in $B[1]$, etc.). We then insertion sort the contents of each bucket, and enumerate the sorted contents of $B[0], B[1], \ldots, B[n-1]$ in order to produce the sorted ordering of A. Using what you know about the analysis of static perfect hashing, please show that this *bucket sort* algorithm runs in $O(n)$ expected time. [Solution]

Problem 111 (Hashing Trees). The two-level hashing concept above generalizes nicely, leading to a "tree" of hash tables. Suppose we hash n elements into a size n table, resolving collisions with chaining for now. This table is the root of our tree. For each entry containing $k > 1$ elements, we then build a second-level hash table of size k. Collisions in second-level tables are resolved with third-level tables, and so on, until each leaf of the tree is a hash table with no collisions. Based on the results of problems 8 and 108, please prove that with strongly universal hashing, the depth of such a tree can be $\Omega(\log \log n)$,

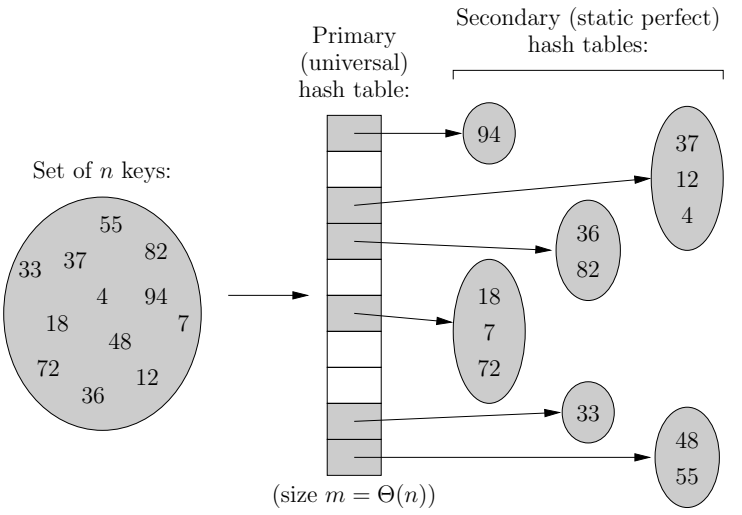


FIGURE 7.4: A dynamic perfect hash table. Collisions in the primary universal hash table are resolved by lookups in secondary static perfect hash tables.

but that the expected depth is $O(\log \log n)$ under universal hashing. As a hint, use the randomized reduction lemma. How does this compare to the expected depth if we were to hypothetically use fully-random hashing? [Solution]

Double Displacement. There are several other nice ways to build a static perfect hash table on n keys in $\Theta(n)$ expected time and $\Theta(n)$ space. One such method, called *double displacement*, has an elegant geometric interpretation where we place our keys on a grid, then shuffle the rows and columns of the grid so that each key ends up in a unique column (which is its hash). In addition to having a relatively simple analysis, this method can also be derandomized to obtain an $O(n \log n)$ *worst-case* construction time. [Full details]

7.3.1 Dynamic Perfect Hashing

So far, our perfect hash tables have all been static, not supporting the ability to add new elements after construction while maintaining an $O(1)$ worst-case running time for *find*. However, using these as black boxes, we can now build a linear-space *dynamic* perfect hashing structure supporting *insert* and *delete* in $O(1)$ expected amortized time[8] and *find* in $O(1)$ worst-case time.

Suppose we maintain n elements in a universal hash table of size $m = \Theta(n)$. As shown in Figure 7.4, sets of colliding elements are stored in second-level static perfect hash tables, which are rebuilt from scratch any time they are changed by an insertion or deletion. Since each static perfect hash structure requires linear

[8]It can be somewhat confusing initially to see a running time that is both "expected" and "amortized". To clarify, an expected amortized time of $O(f(n))$ means that any sequence of k operations should run in $O(kf(n))$ expected time.

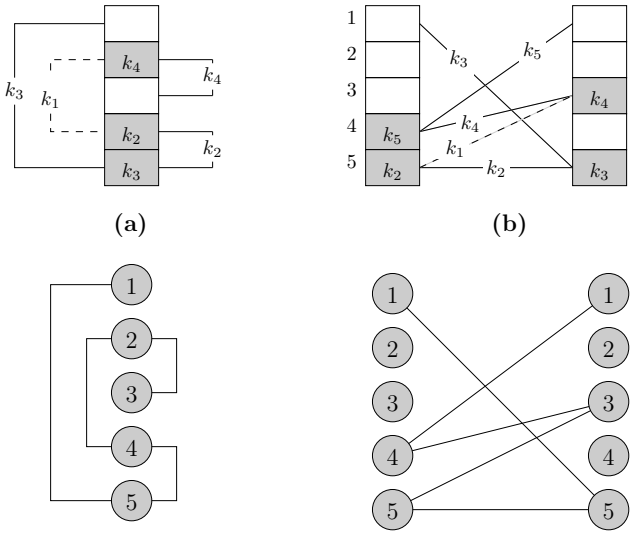


FIGURE 7.5: Examples of cuckoo hash tables. The single-table variant and its associated graph are shown in (a), and the two-table variant and its associated bipartite graph are shown in (b). An insertion of key k_1 in both cases could displace k_2 which in turn will displace k_3.

space, the entire two-level structure occupies $\Theta(n)$ total space. *Find* clearly takes $O(1)$ worst-case time. For *insert* and *delete*, we use the same analysis as with *find* from universal hashing: if k elements collide in some entry of a chained universal hash table, then it takes $O(k)$ time to search them during a *find* operation, and universality implies that $k = O(1)$ in expectation for a generic *find* operation. In our new structure, if k elements collide in some entry of our top-level universal hash table, then it takes $O(k)$ expected time to rebuild them into a second-level static perfect hash table; this is again $O(1)$ expected time for a generic *insert* or *delete* operation due to universality. Finally, we need to add an "amortized" quantifier to *insert* and *delete* since we periodically resize the top-level hash table to ensure $m = \Theta(n)$, just as we have done in the past.

7.3.2 Cuckoo Hashing

We can implement *Cuckoo hashing* — a particularly simple method for dynamic perfect hashing — with either one large table T of size $\Theta(n)$ (usually just larger than $2n$) or two tables T_1 and T_2 both of size larger than n. We use amortized rebuilding to ensure the tables remain sufficiently large. Instead of a single hash function, we use two functions h_1 and h_2. In the single-table variant, an element of key k will be found at one of two locations, either $T[h_1(k)]$ or $T[h_2(k)]$. In the two-table variant, the two locations are $T_1[h_1(k)]$ and $T_2[h_2(k)]$. It is now trivial to *find* an element in $O(1)$ worst-case time, since there are only two places to look for it. A nice way to visualize a Cuckoo hash table is as a graph (Figure 7.5), where each key corresponds to an edge between its two possible locations.

To *insert* an element with key k, we place the element at either of its two valid locations if one is empty. Otherwise, we claim one of these location (chosen arbitrarily), kicking out whatever element resides there. This explains the name "Cuckoo hashing", since the European Cuckoo is a bird that takes over the nest of another bird. Since every element can live in one of two possible locations[9], the newly-evicted element now relocates itself to its alternate place of residence. This may in turn evict another element, and so on, until we finally reach an empty table entry or until we have evicted a chain of $L = \Theta(\log n)$ elements, in which case we give up and rebuild the entire structure from scratch with a new random choice of hash functions h_1 and h_2. Rebuilding make take several attempts, but only a constant number in expectation. When we study graphs in Chapter 15, we will show in problem 268 how to build a Cuckoo hash table on n elements in $\Theta(n)$ time or deduce that this is impossible, given our choice of h_1 and h_2.

Cuckoo Hashing Caveats. One drawback of Cuckoo hashing is that its standard analysis requires the use of L-universal hash functions (i.e., $O(\log n)$-universal hash functions) to ensure the desired $O(1)$ expected amortized guarantees for *insert* and *delete*. While we showed earlier how to construct such functions, that particular construction involves a polynomial of degree $\Theta(\log n)$, and therefore takes $\Theta(\log n)$ time to evaluate. The only known constructions of L-universal hash functions that can be evaluated in $O(1)$ time are impractically complicated, and require more than linear space to represent. In practice, we therefore often use simpler hash functions, but we must do so with care. For example, it has recently been shown that the linear and multiplicative universal hash functions we defined earlier, as well as even some classes of functions with even higher levels of independence, can in some situations fail to allow any valid assignment of keys to table slots *with high probability*. Tabulation-based hashing has, however, been shown to work well both in theory and practice, so that may be the safest approach to use. Further elaboration on this and several other surprising properties of tabulated hashing can be found in the endnotes. [Standard analysis of Cuckoo hashing]

7.4 Radix Trees

Hash tables are extremely fast, but only support the fundamental dictionary operations *insert*, *delete*, and *find*. If we need any of the extended operations (e.g., *pred*, *succ*, *find-min*, *find-max*, *select*, and *rank*) supported by binary search trees and their relatives, we need to consider other types of integer search structures. As a trade-off for supporting a more robust set of operations, these will all have input-sensitive running times depending on C. We begin with the *radix tree*, introduced back in Section 5.5.1 as a fast way to implement a RAM priority queue. Radix trees also work well as general RAM dictionaries, supporting *insert*, *delete*, *find* and all of the extended operations above in $O(\log C)$ time.

As shown in Figure 7.6(a), the (binary) radix tree is a binary tree of height $\log C$. Elements are stored in leaves, where the root-to-leaf path to an element encodes

[9] With the single-table variant, we must choose our hash functions carefully to ensure that $h_1(k) \neq h_2(k)$ for every key k in the table. We could achieve this by rebuilding the table with a new random choice of h_1 and h_2 if we ever try to insert a key k for which $h_1(k) = h_2(k)$, or also by setting $h_2(k) = (h_1(k) + g(k)) \bmod m$, where $g(k)$ hashes to the range $1 \ldots m-1$.

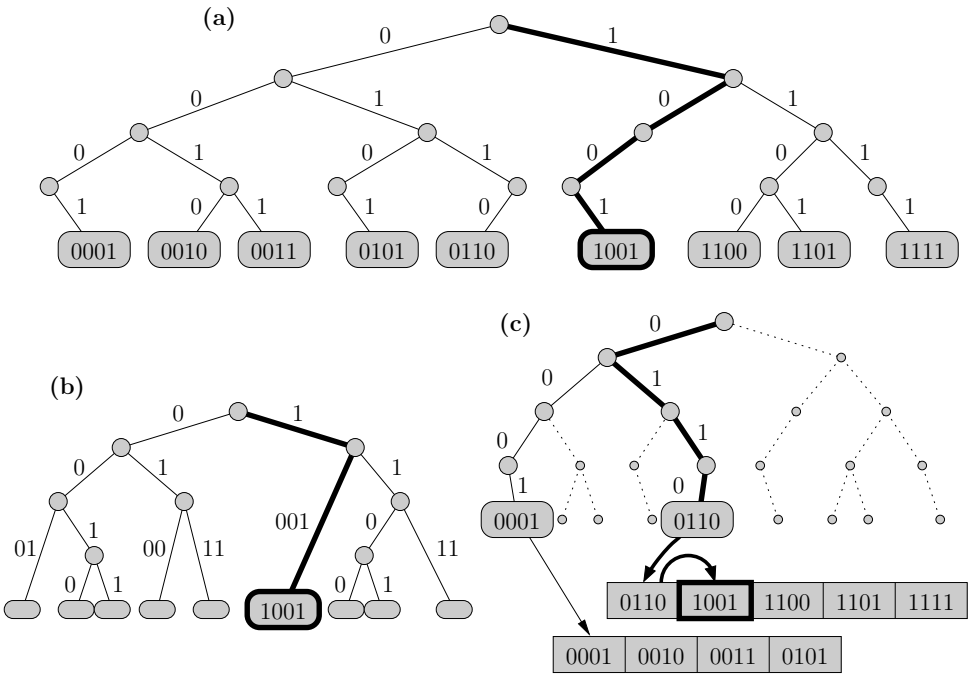


FIGURE 7.6: Reducing the space required to store a radix tree in (a), using (b) path compression and (c) indirection. The search path to the element of key 1001 is shown in each case. In (c), the small nodes and dotted edges are not actually part of the tree; they are merely shown for reference to see what we have succeeded in removing from the tree.

the binary representation of its key — left edges are 0s and right edges are 1s. Operations on a radix tree are straightforward and quite similar to those in binary search trees, so we leave their details to the reader to fill in. We focus here on binary radix trees, although B-ary radix trees are also easy to construct; for example, see Section 5.5.1. We typically assume when dealing with binary radix trees that C is a power of two, so $\log C$ is integer-valued.

Radix trees have many applications in practice. As an example, routers on the Internet often maintain a routing table containing entries like "All packets destined for IP addresses of the form 192.168.0.0/16 should be sent to output port X". The notation 192.168.0.0/16 means all 32-bit IP addresses starting with the 2 bytes (16 bits) 192 and 168. In a radix tree of height 32 (if we are using 32-bit IP addresses), we would attach this particular routing entry to the interior node at depth 16 corresponding to the 16-bit representation of the two bytes 192 and 168, since it applies to all 32-bit addresses in the subtree rooted at that node. When a new packet arrives with destination IP address A, we then walk from the leaf corresponding to A up to the root, using the first, and therefore most specific, rule we encounter along the way (for example, a rule for 192.168.0.0/16 should take precedence over a less-specific rule for 192.0.0.0/8).

7.4.1 Reducing Space: Path Contraction and Indirection

An n-element radix tree occupies $O(n \log C)$ space since each element e requires the storage of $\log C$ internal nodes along the path from the root down to e. We can reduce this down to just $\Theta(n)$ using two different techniques, *path contraction* and *indirection*, both of which are quite powerful and widely-applicable in the study of data structures.

Path Contraction. Suppose we remove all non-branching nodes, as shown in Figure 7.6(b), thereby contracting every path in our tree into a single "long" edge. Every such edge is now labeled with multi-digit binary identifier (stored in a single machine word[10] for the path it represents. Every internal node is now a branching node, and any such binary tree with n leaves has exactly $n-1$ internal nodes (since each internal branching point adds one to the leaf count). Compressed trees of this form are sometimes called *PATRICIA* trees in the literature, where the acronym stands for "Practical Algorithm to Retrieve Information Coded in Alphanumeric", the name of a 1968 paper on this subject by Donald R. Morrison. It is easy to modify the implementation of all radix tree operations so they work natively on a compressed tree with no performance penalty.

Indirection. Rather than storing individual elements in a data structure, suppose we instead store blocks of elements, with each block stored its own separate data structure. This technique — with a coarse-grained "high level" structure storing blocks of data encoded by "low level" structures — is known as *indirection*. Since a radix tree occupies $O(n \log C)$ space, we can reduce this to $\Theta(n)$ if we only store $O(n/\log C)$ actual elements in the tree. To do this, we use indirection and store at the leaves of our radix tree blocks of between $\log C$ and $2 \log C$ consecutive elements, each stored in a separate array, as shown in Figure 7.6(c). The minimum element in each block serves as a "canonical" element representing the entire block in the top-level radix tree. Since only the canonical elements are stored in the tree, the total space usage is now just $\Theta(n)$ for the tree plus the blocks[11].

Operations on our two-level structure now consist of two parts, one that interacts with the high-level radix tree, and the other interacting with a low-level block of elements. For example, *find*(k) calls *pred*(k) in the top-level radix tree to locate the block to which our element should belong, after which we search for k within this block. Both parts take only $O(\log C)$ time. The *insert* and *delete* operations may occasionally need to split or join blocks, or to rotate elements between adjacent blocks (just like in a B-tree) to maintain the $\log C \ldots 2 \log C$ size constraint on the blocks, but this still only requires $O(\log C)$ time, since splitting, joining, and rotation all cause at most two operations in the high-level radix tree.

[10] When we store a collection of variable-length binary numbers, it can be hard to tell the difference between numbers like 0110 and 110, since both of them end up stored as the number 6 in a machine word. For this reason, we typically also store the number of digits in the number, or alternatively we might prepend a dummy 1 bit to the front of each number so leading zero bits are no longer a problem (note that this encoding scheme labels the nodes of a radix tree the same way we index the nodes in a binary heap).

[11] If $n < \log C$, we need to be slightly careful since to store anything at all in the leaves of a radix tree requires $\Theta(\log C)$ space, which is worse than $\Theta(n)$ space. However, in this case our elements fit into a single block, so there is no need for the radix tree yet. Only once we reach $n \geq \log C$ will we bring the full radix tree into the picture.

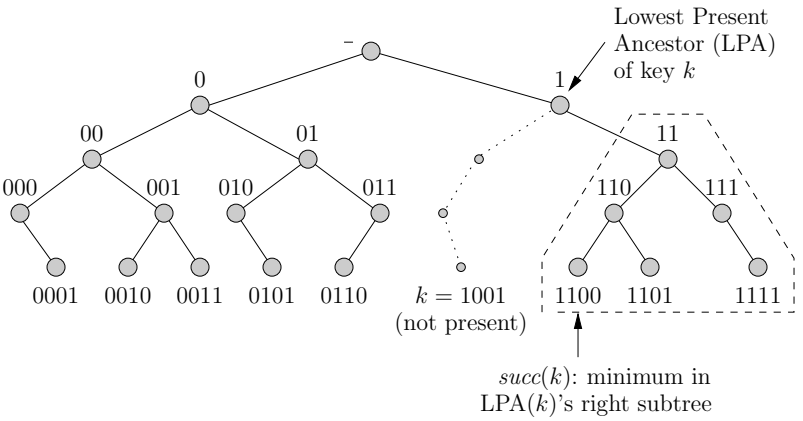


FIGURE 7.7: The Y-fast tree. The binary ID of each node is stored in a universal hash table, which is used to locate the lowest present ancestor of key k. The small nodes and dotted edges are not actually present in the tree.

7.4.2 The Y-Fast Tree

The standard radix tree does not win on performance against a balanced binary search tree, since $C \geq n$ if all our keys are distinct (a typical assumption), and therefore $O(\log C)$ is no better than the $O(\log n)$ running time per operation provided by the balanced binary search tree. However, we can do several things to speed up the radix tree:

- We can store each element simultaneously in a universal hash table to make *find* run in $O(1)$ expected time.
- We can augment each tree node with a direct pointer to its subtree minimum and maximum, improving *find-min* and *find-max* to run in $O(1)$ time.
- For the operations *pred* and *succ*, recall these have two variants: *pred*(e) and *succ*(e) find the predecessor and successor of an element e, and *pred*(k) and *succ*(k) find the next-smallest or next-largest element given a key value k. We can speed up the first variant to run in $O(1)$ time by maintaining a pointer directly from each element to its predecessor and successor.

The augmentations above can easily be maintained without sacrificing the asymptotic $O(\log C)$ running time of *insert* or *delete*.

The only remaining "slow" operations taking $O(\log C)$ time are *insert*, *delete*, *pred*(k), *succ*(k), *select*, and *rank*. We now speed these up (except for *select* and *rank*) to run in only $O(\log \log C)$ time. This gives us a RAM dictionary whose operations run in $O(\log \log C)$ time (except for *select* and *rank*), a range query structure capable of finding all k elements in some range $[a, b]$ in $O(k + \log \log C)$ time, a RAM priority queue whose operations run in $O(\log \log C)$ time, and a RAM sorting algorithm running in $O(n \log \log C)$ time. Depending on implementation, some of these bounds will be amortized and/or in expectation.

The structure we now describe is called a *Y-fast* tree (its name comes from a sequence of historical data structures called P-fast trees, Q-fast trees, and X-fast trees). Following this, we discuss an alternative data structure called a *stratified tree* or a *van Emde Boas (vEB) structure*, which is actually equivalent to the Y-fast tree but provides another useful perspective.

The Y-fast tree is a radix tree that uses hashing to quickly implement *pred*(k) and *succ*(k). We begin by augmenting our tree so that every node maintains pointers to the minimum and maximum elements (leaves) in its subtree, and every element (leaf) is augmented with a pointer to its predecessor and successor. Additionally, we build a universal hash table containing the IDs of all nodes in the tree. As shown in Figure 7.7, a node ID is the binary string (stored as an integer in a single machine word) corresponding to the path from the root down to that node. Recall that we may need to prepend a dummy 1 bit to each node ID or store also the number of bits in each node ID, since otherwise we might have trouble distinguishing IDs like 011 and 0011 with leading zeros.

Consider now the *pred*(k) and *succ*(k) operations. If k is present in the data structure (which we can determine in $O(1)$ expected time by calling *find*(k)), we are done. Otherwise, we quickly locate its *lowest present ancestor* (LPA), shown in Figure 7.7. If LPA(k) has a right child, then *succ*(k) is the minimum in LPA(k)'s right subtree (from which we can follow a predecessor link to determine the predecessor of k). Similarly, if LPA(k) has a left child, then *pred*(k) is the maximum in LPA(k)'s left subtree. The only challenge remaining is to compute LPA(k), whose node ID is the longest prefix of k's $\log C$-bit binary representation that exists in our node ID hash table. We can therefore binary search over prefix lengths in $O(\log\log C)$ expected time. That is, we initially check if the first half of k's binary representation is the ID of some node present in our hash table. If yes, we try a prefix of 3/4 of k's binary length. If not, we try a prefix length of 1/4 of k's binary length, and so on.

We can improve the running time of *insert* and *delete* to $O(\log\log C)$ amortized by using indirection. Suppose we group consecutive elements stored in the leaves of a radix tree into blocks each containing $\Theta(\log C)$ elements as before. If we store each block in a small balanced binary search tree, then operations within a block take only $O(\log\log C)$ time. Insertions and deletions in the top-level radix tree still take $O(\log C)$ time, but these only need to happen when blocks are subject to splits, joins, or rotations, which happen infrequently enough to give us our desired amortized bounds. As an added benefit, indirection also reduces the space required to store the Y-fast tree to $\Theta(n)$. [Complete details]

7.4.3 Stratified / van Emde Boas (vEB) Trees

The final data structure we discuss in this section is an elegant result due to Peter van Emde Boas, sometimes called a *stratified tree*, that "vertically" decomposes a tree in a recursive fashion. A radix tree of height $h = \log C$ has room for $C = 2^h$ elements at its leaves, so a radix tree of height $h/2$ has at most $\sqrt{C} = 2^{h/2}$ leaves. Therefore, by splitting a radix tree in half height-wise (Figure 7.8(a)), we get a top-level radix tree with up to $\sqrt{C}$ leaves, and each leaf in this structure is the root of a low-level radix tree having up to $\sqrt{C}$ leaves. After dividing into half-height trees, we then recursively subdivide these into quarter-height trees, and so on.

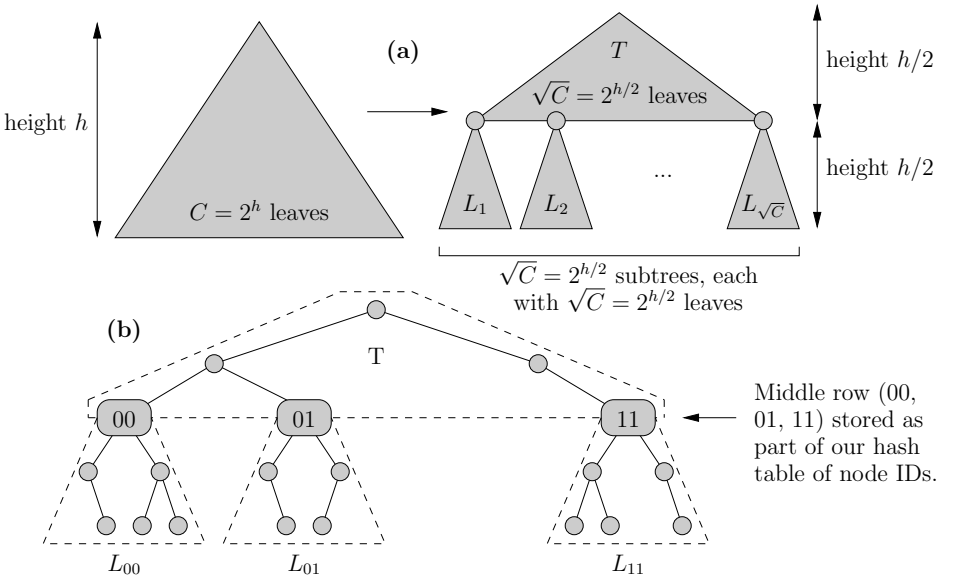


FIGURE 7.8: Illustration of (a) one level of the recursive decomposition of a stratified tree into a top-level tree T and low-level trees L_i, and (b) the same recursive structure overlaid on a radix tree to build a vEB structure.

Although the original van emde Boas (vEB) structure is not described in terms of radix trees, we will describe it in a way that highlights its equivalence with the Y-fast tree. We start with a radix tree, augmented as before so that every node maintains a pointer to the minimum and maximum element (leaf) in its subtree, and every element (leaf) maintains a pointer to its predecessor and successor. We also maintain a universal hash table of node IDs. Recall that we can answer $pred(k)$ and $succ(k)$ queries in this structure in $O(1)$ time once we have located $\text{LPA}(k)$. In the Y-fast tree, we do this in $O(\log \log C)$ time by binary searching over prefixes of the $\log C$-bit binary representation of k. The same process has a nice interpretation in terms of our stratified tree. Let us write the binary representation of k as $k_h k_l$, where k_h contains the "high" half of k's bits, and k_l contains the "low" half. For example, if $k = 10111100$ in binary, then $k_h = 1011$ and $k_l = 1100$. We now perform a single hash lookup to check in $O(1)$ expected time whether a node with ID k_h is present in the "middle row" of our radix tree, as shown in Figure 7.8(b). If so, we recursively search the low-level tree rooted at this node for $\text{LPA}(k_l)$ — note that all nodes in this tree all agree in the high-order half of their IDs, so we are effectively now dealing with just the low-order halves of all node IDs. Otherwise, we recursively search the top-level tree for $\text{LPA}(k_h)$ — in this case, we know that the low-order bits of $\text{LPA}(k)$ are all zero, and we are effectively only dealing with the high-order halves of all node IDs. Each step narrows our search to a tree of half its original height (or equivalently, to a binary string of half its original length), so the entire process locates $\text{LPA}(k)$ in only $O(\log \log C)$ expected time. Moreover, it is easy to see the equivalence between this recursive search in the stratified tree and the binary search over prefixes in the Y-fast tree.

Problem 112 (Cache-Oblivious BST Layout). The stratified tree decomposition has other useful applications other than speeding up a radix tree. Suppose we want to store a large static n-element dictionary in an external memory that supports block reads of size-S blocks. Here, $\Theta(\log_S n)$ block reads are necessary in the worst case to find an element based on its key, and a B-tree with $B = \Theta(S)$, achieves this asymptotic bound. Hardware parameters are often hard to know, however, so we might like to design a *cache-oblivious* data structure (see Section 1.7.5) for which the number of block reads is always $O(\log_S n)$ in the worst case even if we do not know S. Let us take a B-tree with $B = \sqrt{n}$. We then store the contents of each node ($\Theta(\sqrt{n})$ elements for each non-root node) recursively in another B-tree, this time with $B = \sqrt{\sqrt{n}}$, and so on. Please show that this is essentially performing the stratified decomposition above, and also that this structure does indeed require at most $O(\log_S n)$ block reads for *find*, irrespective of the block S. This structure is in some sense the cache-oblivious way to perform binary search. [Solution]

7.5 Further Hashing Applications

Hashing goes well beyond just data structures. Its central idea of mapping a complex object down to a simpler representation has far-ranging application in many computing subfields. For example, in machine learning, we often represent complicated objects using low-dimensional *feature vectors*. After "hashing" a person down to just the vector (height, weight), for instance, we might still be able to predict gender with reasonable accuracy. In this example and others, some might argue that the term "hashing" is less appropriate, since our mapping lacks the intentionally haphazard nature we often strive to achieve with most hash functions. Nonetheless, to "hash" is sometimes spoken more generally to describe any mapping from large and complex to small and simple. In this section, we highlight other prominent instances of this general idea that have substantial algorithmic impact.

7.5.1 Pseudorandom Number Generation

Hash functions are usually designed to behave "randomly", so they are ideal for generating pseudorandom numbers. To generate C pseudorandom numbers in the range $\{0, \ldots, m-1\}$, we evaluate $h(0) \ldots h(C-1)$. For pseudorandom numbers that are pairwise independent or higher, we can use a hash function satisfying this condition. For example, linear strongly universal hashing satisfies pairwise independence. Sometimes this is viewed as "randomness amplification", using a small number of truly-random parameters baked into a hash function to produce a much larger sequence of numbers having a more limited degree of independence.

As a nice example, consider the problem of estimating the number of distinct elements, k, appearing in a massive data stream of length n. We would normally solve this problem by storing the distinct stream elements in a hash table of size $\Theta(k)$, except here k can be much larger than the size of our memory. Suppose the distinct elements in our stream are themselves integers distributed uniformly at random. In this case, roughly half will be multiples of two (ending with 0 in binary), one quarter will be multiples of four (ending with 00 in binary), and so on. If we therefore take the maximum number of 0s we see at the end of any number (when written in

binary), this gives a reasonable estimate of $\log k$. Unfortunately, stream elements may not be randomly distributed, but we can achieve the same effect by looking at hashes of the elements. That is, we map the elements in our stream through a hash function, effectively making the stream look "random" but preserving the relative frequencies of its elements. By aggregating several independent estimates (see, e.g., problem 32) made in parallel as the stream goes by, we can ultimately refine this technique to produce a final estimate that with high probability is accurate to within constant relative error, using only $\Theta(\log n)$ total space. [Full details]

7.5.2 Load Balancing and Consistent Hashing

Large websites often operate from a bank of servers, allowing them to handle high load and also provide fault tolerance if servers fail. To assign incoming packets to servers in a balanced fashion, we could use "round robin" assignment (cycling through the list of servers), or even simpler, we could just assign each packet to a random server. However, these approaches do not provide *consistency* of assignment. Packets from the same source IP address might be mapped to different servers over time, and this is undesirable since servers often maintain some amount of state with respect to each of their active connections (e.g., a "shopping cart", for an on-line store). By hashing packets to servers based on source IP address, the random nature of our hash function gives the same load balancing we would expect from a random assignment, with the added benefit of consistency.

With most common hash functions, changing the size of a hash table can dramatically change where most elements are mapped. This can be problematic here if servers are removed or added — say, if a server becomes unresponsive or unreachable for a short period of time. When a server goes offline, we should ideally re-distribute only its load temporarily among the other servers, leaving the assignments of all other packets unchanged. Again, we want a hashing scheme that provides a measure of *consistency* in response to changes in hash table size. We can achieve consistent hashing of this sort in several ways. Often, instead of hashing packets to servers, we instead think of hashing *both* packets and servers to a third space, where proximity between packets and servers determines our assignment[12]. For example, in Figure 7.9(a), we have mapped packets and servers to an address space that logically wraps around in a circle, where each packet is assigned to the next server following it in clockwise order. As stated, this is not an ideal solution, since removal of a server causes all of its traffic to fail over to the next server in sequence, potentially overloading it. We can do better by considering each server to be a collection of virtual servers, all mapped to different locations on the circle. Temporary removal of all of these virtual instances for a failing server causes that server's load to be spread more uniformly across the other servers.

7.5.3 Distributed Hash Tables

It is now common to find data sets far larger than can fit within the memory of a single machine, making it necessary to distribute the data across a network of

[12]The idea of mapping a server to "network coordinates" in a virtual geometric space is common in the networking literature, as it can be useful for problems such as routing.

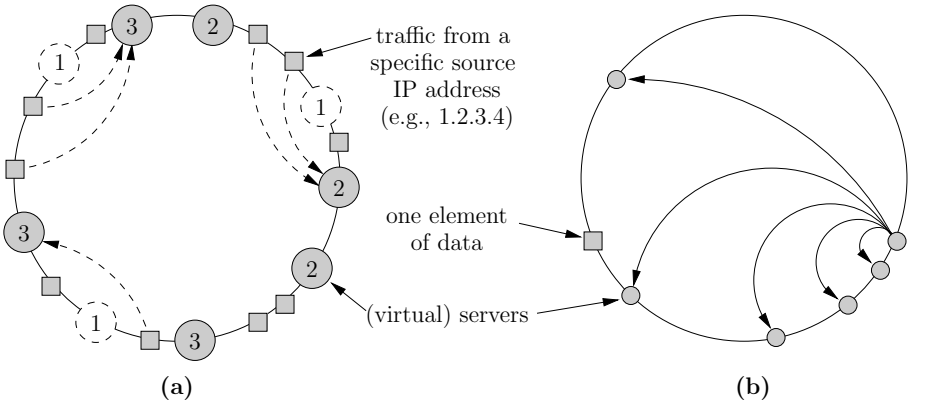


FIGURE 7.9: Consistent hashing around a circle is shown in (a), where removal of all instances of server 1 causes its traffic to be temporarily reassigned to the other servers in a uniform fashion. The long-distance links from a single server in the "Chord" distributed hash table are shown in (b).

servers. A *distributed hash table* (DHT) is a distributed data structure behaving like a large virtual hash table, where calls to *insert*, *delete*, and *find* can be issued from any participating server. Well-designed DHTs gracefully handle server additions and removals, they should have no single point of failure (say, a special "root" server that must be consulted for every query), and they should not require servers to have global knowledge of the entire network, since this would scale poorly.

The circular hashing scheme described above for load balancing leads to one popular approach for implementing a DHT, known as the "Chord" DHT, since servers maintain links across the circle that resemble chords in geometry. Servers and elements of data are all hashed to a common circular address space, with each element stored in a hash table on the server immediately following it in clockwise order. As before, each server is often split into several virtual servers so that its load can be more uniformly redistributed in the event of failure. Similarly, each element may also be hashed to several locations on the circle (and hence stored on several machines), so that isolated server failures don't destroy all existing copies of the element.

The main challenge here is routing. No server knows the global list of all other servers, so it must relay *insert*, *delete*, and *find* requests to the small handful of other servers it does know, so these ultimately reach their intended destinations. By maintaining a short list of servers immediately preceding and following it on the circle, each server can forward requests around the circle, but the number of hops involved could be quite large. To improve performance, we borrow inspiration from the skip list: as shown in Figure 7.9(b), each server also maintains a list of servers at exponentially-increasing distances around the circle (e.g., roughly 1, 2, 4, 8, etc., hops away), thereby reducing the number of hops from linear to logarithmic until a request reaches its destination. Through periodic communication, each server keeps its links up-to-date in response to server additions and removals.

7.5.4 Problem Decomposition, MapReduce

Taking "divide and conquer" to the next level, problems can sometimes be broken down (i.e., "hashed") into very large numbers of subproblems that can be processed in a nearly independent fashion. For example, if we run an quadratic-time algorithm on m subproblems of uniform size n/m, this gives a total running time of $m \times O((n/m)^2) = O(n^2/m)$, a speedup of m compared to running the algorithm without partitioning. Bucket sort (problem 110) is an excellent example of precisely this sort of speedup. As another simple example, suppose we are searching for words in a dictionary that are *anagrams* of each-other — containing the same count of each letter, or equivalently, words that become identical when their letters are sorted. If we apply a hash function to the sorted contents of each word (e.g., using polynomial hashing from the next section), we ensure that anagrams will collide. We can now afford to apply a slower (say, quadratic time) method for partitioning into anagrams within each set of colliding words, being far smaller than the original dictionary.

Problem 113 (Finding Near Neighbors). The approach above is common in geometric algorithms, where we "spatially hash" a set of objects in order to partition them by location, after which we further process each set of nearby objects. To give a nice example, recall that we have already seen how to solve the element uniqueness problem (determine if any two elements in an array $A[1 \ldots n]$ are equal) in $\Theta(n)$ expected time using hashing. Consider now asking whether two array elements have values within some specified distance k of each-other.

(a) Please solve this problem in $\Theta(n)$ expected time and $\Theta(n)$ space. [Solution]

(b) Given n points in the plane $(x_1, y_1) \ldots (x_n, y_n)$ with integer coordinates, show how to find two points within distance k of each-other (or determine that no such pair exists) in $\Theta(n)$ expected time and $\Theta(n)$ space. [Solution]

Problem 114 (Sampling from a Discrete Distribution). Following the same theme of decomposing a problem into smaller pieces that are much more efficient to process, consider the following problem: given a probability distribution $p_1 \ldots p_n$ over n elements (where $\sum p_i = 1$), we would like to process the distribution in $\Theta(n)$ time so that we can sample an element (according to its associated probability) in $O(1)$ time. As a hint, map the elements into "buckets" each representing $1/n$ of probability space, so that after a bucket is chosen with uniform probability, it only takes $O(1)$ additional time to select an element from that bucket. If you like, you can assume the probabilities have been rescaled to integers, in order to avoid real-valued arithmetic. [Solution]

Large-scale problem decomposition can be particularly effective when applied to truly massive data sets. In this case, we often spread the work of problem decomposition across a distributed network of processors, each responsible for decomposing a small part of the input. Similarly, we often distribute the work of solving the resulting subproblems, with each processor being assigned only a small range of subproblems. As part of an increasingly popular framework called *MapReduce*, the first stage is sometimes known as *mapping*, and the second as *reducing*. MapReduce is now built into several widespread programming environments, its popularity stemming from the way it simplifies the implementation of distributed algorithms in the style above. All one needs to do is implement two functions, one for mapping and one for reducing. The framework handles the remaining technical details for instantiating these in parallel, and for routing the output of the mappers to the

appropriate reducers — a task perfectly suited for a distributed hash table, where each parcel of data generated by a mapper is keyed based on the subproblem to which it belongs (i.e., the reducer to which it should be routed).

As a simple example, we could use the anagram detection problem above. Suppose we have a large collection of text phrases stored on a distributed network of machines, in which we would like to identify anagrams. Each mapping task would take a collection of phrases and hash them in a way that ensures collision among anagrams. Each reducing task would then process all the phrases hashing to a particular value, looking for anagrams. We can tune the number of reducing tasks by changing the range of output values for our hash function.

7.5.5 Fingerprinting Large Objects

Many applications involve hashing a large object (e.g., a web page or file on disk) into a single integer, often called a *fingerprint*. We can test equality between two objects very quickly by just comparing their fingerprints. If the fingerprints differ, the original objects must differ. If the fingerprints match, then we strongly suspect the objects are the same. It is possible we could be seeing a hash collision between two different objects, but for an appropriate choice of hash function mapping to a sufficiently large range of fingerprints, we can decrease the probability of such false positives so they are essentially negligible. As a prototypical application, a web search engine might want to remove duplicate pages from its database. After fingerprinting, this problem reduces to removing duplicates from a large set of integers, a problem easily solved using a standard hash table.

We can regard any large object as just an integer array $A[0 \ldots n-1]$, with each $A[i] \in \{0, \ldots, C-1\}$. Perhaps the simplest and most common way to hash A down to a single integer small enough to fit in a machine word is with a *polynomial hash function*, where we take the array elements to be coefficients of a polynomial

$$A(x) = A[0] + A[1]x + A[2]x^2 + \ldots + A[n-1]x^{n-1},$$

and then set $h(A) = A(x) \bmod p$, where $p \geq C$ is an arbitrary prime (say, small enough to fit into a machine word), and x is randomly chosen from $\{0, \ldots, p-1\}$ during initialization. As in other hashing schemes, we use arithmetic modulo p to keep our numbers from growing too large.

If A and A' are two different length-n arrays, then $\mathbf{Pr}[h(A) = h(A')] \leq \frac{n-1}{p}$, so we can reduce this collision probability to a miniscule level by making p sufficiently large. For example, we avoid collisions with high probability by choosing $p \geq n^c$ with c constant. The $\frac{n-1}{p}$ collision probability follows from the fact that two different polynomials $A(x)$ and $A'(x)$ of degree $n-1$ can satisfy $A(x) \equiv A'(x) \pmod p$ for at most $n-1$ of the p possible choices of x. Such a value of x would be a root of the difference polynomial $A(x) - A'(x)$, and a nonzero polynomial of degree $n-1$ like this can have at most $n-1$ roots, even modulo a prime p.

We can easily compute $h(A)$ in only $O(n)$ time using Horner's rule (Section 23.1.3): start with $A[n-1]$, multiply by x, then add $A[n-2]$, then multiply by x again, then add $A[n-3]$, and so on, ending with the addition of $A[0]$. All the while, we keep reducing the result modulo p so it always fits within a machine word. Moreover,

after precomputing a table of $x^i \bmod p$ for $i \in \{-1, \ldots, n-1\}$[13], we can update $h(A)$ in only $O(1)$ time after changing A by modifying one of its elements, or by adding or removing an element at one of its endpoints. This will be particularly useful in Chapter 9 when we use hashing to solve string matching problems.

Problem 115 (Alternative Methods for Hashing Large Objects). This problem highlights two other common ways to hash an array $A[0 \ldots n-1]$, with each $A[i] \in \{0, \ldots, C-1\}$, down to a small integer in $O(n)$ time.

(a) **Dot Product with a Random Vector.** Regard A as a length-n vector, and let $h(A) = A \cdot X \bmod p = (A[0]X[0] + \ldots + A[n-1]X[n-1]) \bmod p$, where $p \geq C$ is a prime number, and $X[0] \ldots X[n-1]$ are each chosen independently from $\{0, \ldots, p-1\}$ during preprocessing. Please show that for any two length-n arrays $A \neq A'$, we have $\mathbf{Pr}[h(A) = h(A')] = 1/p$, so we again avoid collisions with high probability as long as p is sufficiently large. For a hint at the solution, refer to problem 38, which uses essentially the same underlying mathematics. [Solution]

(b) **Reduce Modulo a Random Prime.** Regard A as a large integer whose digits are $A[n-1] \ldots A[0]$ when written in base C. In terms of the polynomial $A(x)$, this number is precisely $A(C)$. Let $h(A) = A(C) \bmod p$, where p is a prime number chosen randomly from a set of r different alternatives during preprocessing. Please show that for any two arrays $A \neq A'$, we have $\mathbf{Pr}[h(A) = h(A')] \leq \frac{n}{r} \log C$, so yet again we avoid collisions with high probability as long as r is sufficiently large. In Chapter 25, we will learn how to generate random prime numbers. [Solution]

Problem 116 (Detecting Infinite Loops). If we are watching the execution of a computer program, we know it has entered an infinite loop if the contents of its memory ever becomes identical to the memory contents at some point in the past. Using the trick from problem 1, please show how to instrument any computer program so that we can detect infinite looping (with high probability), using asymptotically the same time and space as the original program. Assume that the number of steps we run our program is sufficiently small so as to fit into a single machine word. [Solution]

7.5.6 Security and Data Integrity

Hashing has extensive applications in the domain of computer security. A system administrator may keep a record of the fingerprints of important files on a machine, which can be used later to detect tampering. Fingerprints are also used to certify the integrity of important digital records, often in combination with other cryptographic methods such as digital signatures (discussed in Section 25.4.3). For example, by including the hash of the contents of a packet as a short "checksum" before sending it across a network, we can detect corruption during transit. We can also vastly improve the security of a computer system by storing passwords as fingerprints rather than "in the clear". It is still easy to verify a user's password in this setting.

Hash functions are often hard to "invert" (i.e., it can be difficult to find a key that hashes to some specific given value), and those designed for cryptographic or security purposes are specifically tailored to prevent attempts at inversion, to make it hard to falsify data or reverse engineer passwords from their hashes. The standardized *Secure Hash Algorithm* (SHA-1 and SHA-2) family of hash functions are the two

[13] In Chapter 25, we will see how to compute the multiplicative inverse x^{-1} modulo p efficiently using Euclid's algorithm.

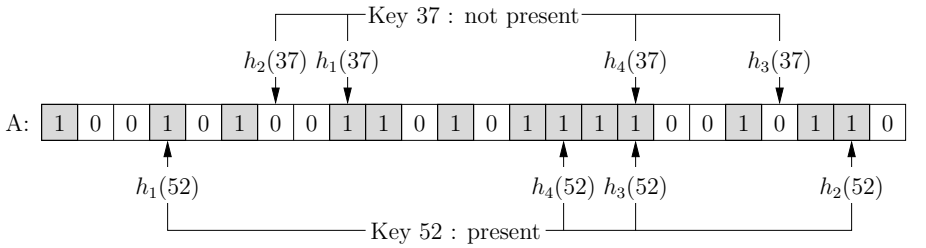


FIGURE 7.10: Diagram of a Bloom filter with four hash functions $h_1 \ldots h_4$. A particular key k is taken to be present in the structure if $A[h_1(k)] \ldots A[h_4(k)]$ are all set to 1.

most popular such functions in practice, producing as output fingerprints ranging from 160 bits (for SHA-1) up through 512 bits (for the longest variant of SHA-2). To find a collision by brute force for, say, SHA-1, one would need to check roughly 2^{80} different keys according to the birthday paradox[14]. To date, no colliding pairs of keys for any of these hash functions have ever been discovered.

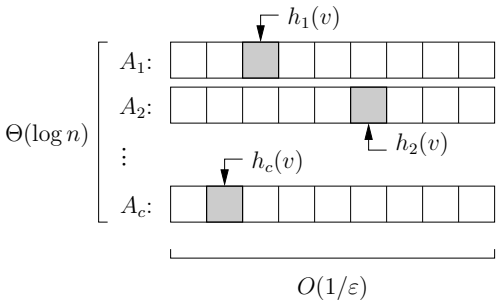
7.5.7 Bloom Filters and Sketching

The *Bloom filter*, named after its creator, Burton Bloom, provides a very compact representation of a set that trades space for accuracy of membership queries. Calling *find*(k) simply returns "yes" or "no", and additional satellite data cannot be associated with keys. There is a chance *find*(k) might return "yes" even if k is not present, although the probability of these false positives can be set arbitrarily low at the expense of extra query time or space. The structure supports insertion but not deletion of keys, at least not in its most basic incarnation.

A Bloom filter starts as a zero-filled binary array $A[1 \ldots m]$ where $m > n$, along with c independent hash functions $h_1 \ldots h_c$. Insertion of a key k sets $A[h_1(k)] \ldots A[h_c(k)]$ all to 1, and *find*(k) returns true only if $A[h_1(k)] \ldots A[h_c(k)]$ are all set to 1, as shown in Figure 7.10. Both operations run in $O(c)$ time. Clearly, *find*(k) always succeeds if k has been inserted into the structure. If k has not been inserted, then *find*(k) can produce a false positive result if $A[h_1(k)] \ldots A[h_c(k)]$ all happen to be 1, but we can reduce the probability of this by making c larger (sacrificing speed) or making m larger (sacrificing space). Making c too large, however, can cause the false positive rate to climb again, since this fills the structure with too many 1s. [Analysis of Bloom filters]

An adequately functioning Bloom filter requires linear space: $m > n$ bits for storing n elements. Its underlying idea, however, motivates the design of several related

[14]One might actually worry about using 2^{80} memory even more than taking 2^{80} units of time. That is, it may seem necessary to store each hash we check in a hash table so the hash of every new random key we consider can be quickly compared to those generated earlier. However, if we use our hash function to generate a pseudorandom sequence of keys $x_0, x_1, x_2, \ldots$, where $x_i = h(x_{i-1})$, then this sequence behaves like an implicitly-defined linked list, ending with a loop whose initial element tells us the hash code generated by $h(x_i)$ and $h(x_j)$ for $x_i \neq x_j$. We can find this using linear time but only constant memory with the result of problem 1.



Count-Min Sketch

Update(v): For $i = 1 \ldots c$,
Increase $A_i[h_i(v)]$ by $+1$

Query(v): Return $\min_{i=1\ldots c} A_i[h_i(v)]$

Count Sketch

Update(v): For $i = 1 \ldots c$,
Increase $A_i[h_i(v)]$ by $g_i(v)$

Query(v): Return $\underset{i=1\ldots c}{\text{median}}\, A_i[h_i(v)]\, g_i(v)$

FIGURE 7.11: The count sketch and count-min sketch structures.

randomized "sketch" data structures that can summarize the contents of a massive data stream using only *logarithmic* space. For example, we showed earlier how to estimate the number of distinct elements appearing in a length-n stream with only logarithmic space. Here, we discuss two elegant structures for estimating frequencies (occurrence counts) of values in a length-n stream, also using just logarithmic space. These are often used to identify elements occurring with high frequency, sometimes known as "heavy hitters". As a common example, a router might want to inspect a stream of data packets to learn which IP source or destination addresses account for the most traffic.

As shown in Figure 7.11, both the *count-min sketch* and *count sketch* involve $c = \Theta(\log n)$ arrays $A_1 \ldots A_c$ of counters, each array having length $O(1/\varepsilon)$. We create hash functions $h_1 \ldots h_c$ indexing into each array. For every successive value v we observe in the data stream as it passes by, we call *update*(v), which modifies the counter at index $h_i(v)$ in each array A_i. For count-min sketch, the update is just an increment. For count sketch, we randomly choose between an increment or a decrement, but using hashing to do this in a consistent way: we define hash functions $g_1 \ldots g_c$ mapping stream values to $\{-1, +1\}$, and add $g_i(v)$ to $A_i[h_i(v)]$.

An estimate of the frequency f_v of any value v can be obtained at any time by calling *query*(v). For count-min sketch, this returns the minimum of the values at $A_i[h_i(v)]$ across all arrays $i = 1 \ldots c$. This will always be an over-estimate, since f_v has already been added into each of these counters. It is likely, however, that at least one of these counters has not been "contaminated" by too much additional weight from other values, so the minimum gives an estimate in the range $[f_v, f_v + \varepsilon n]$ with high probability [Detailed analysis]. With count sketch, observe that value v contributes either $+f_v$ to the counter $A_i[h_i(v)]$ (if $g_i(v) = +1$) or $-f_v$ (if $g_i(v) = -1$). Hence, $A_i[h_i(v)]\, g_i(v)$ is a good estimator for f_v. Indeed, $\mathbf{E}[A_i[h_i(v)]\, g_i(v)] = f_v$, since value v contributes f_v and all other values contribute zero in expectation, with their mappings in g being equally likely to be positive or negative. We improve our final estimate by taking the median of these estimates across all arrays $i = 1 \ldots c$. Count sketch gives an estimate in $[f_v - \varepsilon n, f_v + \varepsilon n]$ with high probability. Otherwise written, this error bound is $\pm\ \varepsilon ||f||_1$, where f is the vector of frequencies for all values (since $||f||_1 = n$). An alternate bound we can show – stronger for more uniform frequency distributions – is $\pm\ \varepsilon ||f||_2$, although this requires each counter array to have $O(1/\varepsilon^2)$ size. [Detailed analysis]

With the count-min sketch, if all we want to do is estimate whether v appears in the data stream ($f_v \geq 1$), it suffices to just set the counters $A_i[h_i(v)]$ to 1 instead of incrementing them during an update. Observe that this leads to what is essentially a multi-array variant of the Bloom filter, so we might think of the Bloom filter as a special case of the count-min sketch.

Problem 117 (Deterministic Algorithms to Find Frequent Elements). A majority element in a data stream $x_1 \dots x_n$ is an element with frequency strictly more than $n/2$. If you are told that a stream contains a majority element, it can be identified with a simple algorithm taking linear time and just constant space, making only a single pass through the stream. The algorithm keeps track of two values, a value v (initialized to null) and a count c (initialized to zero). For each stream element x_i it examines in sequence, it increments c if $x_i = v$, and decrements c if $x_i \neq v$; if c drops to zero, we replace v with x_i and set $c = 1$. If you are unsure if the stream contains a majority element, we can run two passes, the first producing some output x, and the second checking if x is indeed a majority element.

(a) Please show that if there is a majority element, then this element will reside in v at termination. [Solution]

(b) A straightforward generalization of the algorithm above can find elements occurring with frequency exceeding n/k using only $\Theta(k)$ space (so the approach above is the special case where $k = 2$). We maintain $k - 1$ pairs $(v_1, c_1) \dots (v_{k-1}, c_{k-1})$ similar to the (value, counter) pair above. For each stream element x_i we inspect, if there is some pair (v_j, c_j) with $v_j = x_i$, we increment c_j. Otherwise, we decrement *all* of the counters $c_1 \dots c_{k-1}$, and if one of these drops to zero, we select any such pair and replace it with $(x_i, 1)$. Please argue that any stream element with frequency exceeding n/k must end up as one of the v_i's at termination (so in particular, we could verify whether these $k - 1$ elements are indeed high-frequency with a second pass over the stream). [Solution]

7.5.8 Compressive Sensing and its Relatives

The process of mapping a large data stream $x_1 \dots x_n$ to a shorter sketch $A[1 \dots m]$ inherently loses information, and hence we can usually only extract limited information about x from the sketch. However, in some cases — for instance if we know most of the mass in x is concentrated in only a few of its elements — we can surprisingly "invert" A to obtain a good estimate of x. Here, we usually consider linear sketches, where each element of the sketch $A[j]$ is a linear combination of elements in x. This process is known as *compressive sensing* (also *compressed sensing*) since it seeks to recover a large signal from a compressed representation. It has many potential applications, since real-world signals (e.g., audio, image data) are often quite sparse when viewed in the right "basis". For example, as we will see in Chapter 23, a photograph may contain only a few significant frequency components when we look at its Fourier transform. Given a signal that is sparse in some basis, we would like to approximately reconstruct it from a small number of samples from its linear projection into some other basis (i.e., our sketch) in which its sparsity may not be so apparent. Hashing is one of several techniques that can be used for "sparse recovery" problems of this sort.

For a simple example, suppose each $A[j]$ is the sum of a random subset of elements in x, given by a hash function g_j mapping $i \in \{1, \dots, n\}$ to $\{0, 1\}$, so $A[j] = \sum_i x_i g_j(i)$.

Here, a sketch of size $m = O(k \log n)$ is all we need to exactly reconstruct with high probability any length-n stream x that is *k-sparse* — having $||x||_0 \leq k$, where $||x||_0$ is the number of nonzero entries[15] in x. The intuition here is similar to the count-min sketch: each of the k nonzero entries in x likely appears in at least one location of the sketch uncontaminated by other entries in x. [Full details]

Problem 118 (Problems Related to Sparse Recovery). Here, we investigate two other related problems that can be approached with techniques very similar to those used in the example above.

(a) **Group Testing.** Suppose up to k of a set of n individuals has a particular disease. By pooling blood samples, you can test any group of individuals in a single step, but if the test comes back positive, all you know is that one or more members of the group have the disease. We would like to identify the diseased individuals with a minimum number of tests. In problem 41, we showed how to use only $O(k \log n)$ tests in an "adaptive" fashion, where each test can depend on the results of previous tests. We can achieve the same bound in a non-adaptive setting, however. Let the $m = O(k \log n)$ groups being tested be determined by m random subsets as above. In this case, our sketch $A[1 \dots m]$ is defined identically, except $A[j] = \max_i x_i g_j(i)$ (here, $x_i = 1$ denotes an individual with the disease, and $x_i = 0$ is a healthy individual). Please show how to reconstruct x from A with high probability. [Solution]

(b) **Checksums and Error Correcting Codes.** Consider a length-n binary message $x_1 \dots x_n$ in which up to k bits might be corrupted. We would like to identify these bits by comparing the corrupted version of x with an m-bit checksum $A[1 \dots m]$ precomputed on the original x. As above, suppose each bit in the checksum corresponds to a random set of bits in x, where we define $A[j] = \bigoplus_i x_i g_j(i)$ ($\oplus$ denotes the XOR operation). Show that we can determine which bits were corrupted[16] with high probability with a checksum of length $m = O(k \log n)$. [Solution]

L_1 Minimization and Linear Programming. Since each $A[j]$ is defined by a linear equation in $x_1 \dots x_n$, a linear sketch is a linear system of n variables in m equations, which in this context has no single unique solution for x since n is much larger than m. If we believe x should have most of its mass concentrated in a few components, we can attempt to recover x by minimizing $||x||_0$ subject to our system of linear constraints — that is, we want to find a *sparsest* solution to an under-determined linear system. Although this is unfortunately NP-hard, a sparse solution often "magically" materializes if we instead minimize $||x||_1 = \sum_i |x_i|$, an objective we can minimize in polynomial time over a system of linear constraints by solving a linear program (see Chapter 12). For those well-versed in linear algebra, we include a [brief discussion] of why this approach tends to work.

Due to the propensity of the $||x||_1$ objective to produce sparse solutions, it is often used as a "regularizing" term added as a penalty to other objective functions when sparse solutions are desired. For example, instead of just minimizing $f(x)$, we might minimize instead $f(x)+\lambda||x||_1$ for some appropriate value of λ to encourage solutions

[15] Recall that in general, we define the L_p norm of x as $||x||_p = \left(\sum_i |x_i|^p\right)^{1/p}$. In the limit as $p \to 0$, this does indeed converge to the number of non-zero components in x.

[16] A complicating aspect we do not consider here, but that is important in most constructions of error-correcting codes, is that bits in the checksum itself might also be corrupted by the same process that corrupted x (e.g., a noisy communication channel).

that are sparse. Further comments on regularization in optimization appear in Chapter 14.

7.5.9 Locality-Sensitive Hashing

Collisions in hashing are often undesirable. However, in *locality-sensitive hashing*, they are actually the most useful aspect of a hashing scheme! Since similar objects often collide when hashed, we carry this idea one step further and try to design a randomized hash function for which collision probability directly reflects object similarity. If $\text{sim}(x, y)$ is the similarity between two complicated objects x and y, our goal is to build a hash function h for which $\mathbf{Pr}[h(x) = h(y)] = \text{sim}(x, y)$, where the probability is taken over the random parameters baked into the hash function[17]. Here are three prominent examples:

- **Negated L_1 Distance.** Let x and y be two points in $[0, 1]^d$, the d-dimensional unit cube. We often use distance to measure dissimilarity, so we can convert this into a similarity measurement by negation:

  $$\text{sim}_D(X, Y) = 1 - \frac{1}{d}||x - y||_1.$$

  Here, we use L_1 distance, which between two points in a d-dimensional unit cube can be at most d. By dividing by d and then subtracting from one, we therefore obtain a measurement of similarity in $[0, 1]$.

- **Correlation.** If x and y are vectors in high-dimensional space, we often measure similarity by taking their *correlation*, $\hat{x} \cdot \hat{y}$, where $\hat{x} = x/||x||$ denotes x normalized to have unit Euclidean length. Correlation measures the cosine of the angle between x and y, ranging from -1 for vectors pointing in opposite directions to $+1$ for vectors pointing in the same direction. Since we prefer measurements of similarity in the range $[0, 1]$, we use a re-scaled version:

  $$\text{sim}_C(x, y) = \frac{1}{2}\left(1 + \hat{x} \cdot \hat{y}\right).$$

- **Jaccard Similarity.** If X and Y are two sets, it is natural to measure their similarity in terms of relative overlap,

  $$\text{sim}_J(X, Y) = \frac{|X \cap Y|}{|X \cup Y|},$$

  This is known as their *Jaccard* similarity, and it always lies in the range $[0, 1]$.

In each of these three settings, we can design a simple hash function for which collision probability reflects similarity.

For negated L_1 distance, we select a random coordinate $i \in \{1, \ldots, d\}$ and a random threshold t, and hash point x to 1 if $x_i \geq t$, and 0 otherwise. It is easy to show that $\mathbf{Pr}[h(x) = h(y)] = \text{sim}_D(x, y)$. [Details]

[17] A common alternate definition of a locality-sensitive hash function is a function for which $\mathbf{Pr}[h(x) = h(y)] \geq p$ if $\text{sim}(x, y)$ is above some threshold t, but $\mathbf{Pr}[h(x) = h(y)] \leq p'$ (with $p' < p$) if $\text{sim}(x, y)$ is below some smaller threshold $t' < t$. Hence, collision probabilities still allow us to differentiate between similar and less-similar objects.

For correlation, we pick a random hyperplane through the origin described by its normal vector $\hat{v}$. We hash point x to 1 if $x \cdot \hat{v} \geq 0$, otherwise we hash x to 0. It is easy to show that $\mathbf{Pr}[h(x) = h(y)] \approx \text{sim}_C(x, y)$. [Details]

For Jaccard similarity, we use a *min-wise* hash function, defined for a set X as $h(X) = \min\{f(e) : e \in X\}$, where f assigns a unique random number to every element[18] in our universe U. Since $h(X) = h(Y)$ only if the minimum element in $X \cup Y$ comes from $X \cap Y$, we have $\mathbf{Pr}[h(X) = h(Y)] = \frac{|X \cap Y|}{|X \cup Y|} = \text{sim}_J(X, Y)$.

Hashing to a Single Bit. Several of the example hash functions above map a complicated object down to just a single bit. In fact, we can achieve this with a simple modification to *any* locality-sensitive hash function. Starting with a function h satisfying $\mathbf{Pr}[h(x) = h(y)] = \text{sim}(x, y)$, consider sending its output through a secondary hash function g mapping to $\{0, 1\}$. Using an appropriate universal hash function for g, we have

$$\mathbf{Pr}[g(x) = g(y)] = \begin{cases} 1 & \text{if } x = y \\ 1/2 & \text{if } x \neq y \end{cases}.$$

If we now expand out $\mathbf{Pr}[g(h(x)) = g(h(y))]$ by conditioning on whether or not $h(x) = h(y)$, we get

$$\underbrace{\mathbf{Pr}[g(h(x)) = g(h(y)) \mid h(x) = h(y)]}_{1} \underbrace{\mathbf{Pr}[h(x) = h(y)]}_{\text{sim}(x,y)}$$
$$+ \underbrace{\mathbf{Pr}[g(h(x)) = g(h(y)) \mid h(x) \neq h(y)]}_{1/2} \underbrace{\mathbf{Pr}[h(x) \neq h(y)]}_{1-\text{sim}(x,y)},$$

so $\mathbf{Pr}[g(h(x)) = g(h(y))] = (\text{sim}(x, y)+1)/2$. The binary function $g(h(x))$ therefore still gives a collision probability reflecting object similarity, albeit slightly re-scaled.

Hashing to a Binary String. We can now hash an object down to a single bit so that collision probability estimates object similarity. To obtain a more accurate estimate, we can repeat this with d independent hash functions, thereby hashing an object x to a d-bit binary string s_x. Object similarity between x and y is now given by the average number of bits that agree between s_x and s_y, which we often write as $1 - H(s_x, s_y)/d$, where $H(s_x, s_y)$ is the number of bits that differ between s_x and s_y, known as their *Hamming distance* (note that this is actually the same as our formula for negated L_1 distance, since Hamming distance is the same thing as L_1 distance, only in the context of binary strings). Locality-sensitive hashing therefore reduces similarity computation of potentially large, complex objects, down to the conceptually simpler problem of Hamming distance computation in binary strings. A nice application of this is shown in the following section.

7.5.10 High-Dimensional Nearest Neighbor Search

In the next chapter, we will learn several classical data structures for finding *near neighbors* in a geometric point set, but these unfortunately only work well in very

[18] Ideal min-wise hash functions are generally not possible to construct due to space considerations, since we would need to store where every element in our universe U (a very large set) is mapped under f. We therefore usually try to build *approximate* min-wise hash functions described by limited amounts of randomness (much the same idea as with universal hashing), where $\mathbf{Pr}[h(X) = h(Y)] \approx \text{sim}_J(X, Y)$. See the endnotes for further references to results in this area.

low-dimensional spaces, having time and/or space bounds that scale exponentially in terms of dimension (this phenomenon is widespread in computational geometry, and is sometimes known as the "curse of dimensionality"). It is quite common in many applications, however, to encounter near neighbor problems in extremely high-dimensional spaces — say, with thousands of dimensions. For example, in machine learning, we might represent objects by high-dimensional *feature vectors*, where we try to infer the classification of a query object by looking at its nearest neighbors in a training set of pre-classified objects.

Locality-sensitive hashing helps to simplify proximity questions like near neighbor queries by reducing them to equivalent proximity questions over points in $\{0,1\}^d$ (i.e., length-d binary strings) using the Hamming distance metric. Note that the dimension d of this new "binary" problem is somewhat unrelated to the dimension of our original data before applying locality-sensitive hashing, but that d is typically also quite large. Even so, the simpler structure of the new problem allows us to use hashing to build *approximate* near neighbor structures that perform well even if d is quite large. Given a distance threshold Δ and desired approximation factor $c > 1$, our goal is build a static data structure on n data points in $\{0,1\}^d$ such that given any query point q:

- If there exists a point p in our data set with $H(p,q) \leq \Delta$, the query returns an approximate near neighbor — a point p' with $H(p',q) < c\Delta$.
- If $H(p,q) \geq c\Delta$ for every point p in our data set, the query returns "No near neighbor".

By querying multiple instances of this structure built for distance thresholds $\Delta = 1, c, c^2, c^3, \ldots$, we can find a c^2-approximate *nearest neighbor* of q, since if the distance to the true nearest neighbor of q lies in the range $[c^x, c^{x+1}]$, then the structure will return a point at distance at most c^{x+2} when run at the threshold $\Delta = c^{x+1}$.

To find approximate near neighbors, the main idea is to hash each data point p to a lower-dimensional point by projecting it onto a random subset of coordinates chosen during preprocessing, with each coordinate $i = 1 \ldots d$ retained independently with some probability α. The lower we set α, the more likely distant points will collide when hashed. For example, points $p_i = 0\underline{1}\underline{1}\underline{1}0\underline{1}0$ and $p_j = 1\underline{1}\underline{1}\underline{1}0\underline{0}0\underline{0}$ collide if we retain their even coordinates (underlined).

Consider a query point q for which a nearby point p with $H(p,q) \leq \Delta$ exists in our dataset. Our hash table makes it easy to enumerate all the points p' in our dataset colliding with q. If we have chosen α to be low enough, then it is likely that we will find in this set a "good" point p' (with $H(p',q) < c\Delta$). However, if we chose α to be too low, we will end up also searching through a large number of "bad" points (with $H(p',q) \geq c\Delta$). By analogy, if you are trying to catch a rare breed of fish, you need to cast a large enough net to ensure catching at least one specimen of the rare breed, but if you cast too large a net, you will waste time sorting through the many other fish you catch as collateral damage.

By choosing α appropriately and also querying several independent copies of our data structure, we can answer c-approximate nearest neighbors queries correctly with high probability in time $O(dn^{1/c})$, and with $O(nd + n^{1+1/c})$ space usage (ignoring extra logarithmic factors for simplicity, to highlight the dominant terms

in each bound). Observe that these bounds are respectively sublinear and subquadratic in n, with no exponential dependence on dimension d. For large values of n, the query time bound above can be much better than the naïve approach that finds a nearest neighbor in $\Theta(dn)$ time by simply checking every point in the data set. [Full details]

