We considered this modified version of `Quicksort` to simplify the analysis. Coming back to the original `Quicksort`, our intuition suggests that the expected running time is no worse than in the modified algorithm, as accepting the noncentral splitters helps a bit with sorting, even if it does not help as much as when a central splitter is chosen. As mentioned earlier, one can in fact make this intuition precise, leading to an $O(n \log n)$ expected time bound for the original `Quicksort` algorithm; we will not go into the details of this here.

13.6 Hashing: A Randomized Implementation of Dictionaries

Randomization has also proved to be a powerful technique in the design of data structures. Here we discuss perhaps the most fundamental use of randomization in this setting, a technique called *hashing* that can be used to maintain a dynamically changing set of elements. In the next section, we will show how an application of this technique yields a very simple algorithm for a problem that we saw in Chapter 5—the problem of finding the closest pair of points in the plane.

The Problem

One of the most basic applications of data structures is to simply maintain a set of elements that changes over time. For example, such applications could include a large company maintaining the set of its current employees and contractors, a news indexing service recording the first paragraphs of news articles it has seen coming across the newswire, or a search algorithm keeping track of the small part of an exponentially large search space that it has already explored.

In all these examples, there is a *universe* U of possible elements that is extremely large: the set of all possible people, all possible paragraphs (say, up to some character length limit), or all possible solutions to a computationally hard problem. The data structure is trying to keep track of a set $S \subseteq U$ whose size is generally a negligible fraction of U, and the goal is to be able to insert and delete elements from S and quickly determine whether a given element belongs to S.

We will call a data structure that accomplishes this a *dictionary*. More precisely, a dictionary is a data structure that supports the following operations.

- `MakeDictionary`. This operation initializes a fresh dictionary that can maintain a subset S of U; the dictionary starts out empty.
- `Insert`(u) adds element $u \in U$ to the set S. In many applications, there may be some additional information that we want to associate with u

(for example, u may be the name or ID number of an employee, and we want to also store some personal information about this employee), and we will simply imagine this being stored in the dictionary as part of a record together with u. (So, in general, when we talk about the element u, we really mean u and any additional information stored with u.)

- `Delete`(u) removes element u from the set S, if it is currently present.
- `Lookup`(u) determines whether u currently belongs to S; if it does, it also retrieves any additional information stored with u.

Many of the implementations we've discussed earlier in the book involve (most of) these operations: For example, in the implementation of the BFS and DFS graph traversal algorithms, we needed to maintain the set S of nodes already visited. But there is a fundamental difference between those problems and the present setting, and that is the size of U. The universe U in BFS or DFS is the set of nodes V, which is already given explicitly as part of the input. Thus it is completely feasible in those cases to maintain a set $S \subseteq U$ as we did there: defining an array with $|U|$ positions, one for each possible element, and setting the array position for u equal to 1 if $u \in S$, and equal to 0 if $u \notin S$. This allows for insertion, deletion, and lookup of elements in constant time per operation, by simply accessing the desired array entry.

Here, by contrast, we are considering the setting in which the universe U is enormous. So we are not going to be able to use an array whose size is anywhere near that of U. The fundamental question is whether, in this case, we can still implement a dictionary to support the basic operations almost as quickly as when U was relatively small.

We now describe a randomized technique called *hashing* that addresses this question. While we will not be able to do quite as well as the case in which it is feasible to define an array over all of U, hashing will allow us to come quite close.

Designing the Data Structure

As a motivating example, let's think a bit more about the problem faced by an automated service that processes breaking news. Suppose you're receiving a steady stream of short articles from various wire services, weblog postings, and so forth, and you're storing the lead paragraph of each article (truncated to at most 1,000 characters). Because you're using many sources for the sake of full coverage, there's a lot of redundancy: the same article can show up many times.

When a new article shows up, you'd like to quickly check whether you've seen the lead paragraph before. So a dictionary is exactly what you want for this problem: The universe U is the set of all strings of length at most 1,000 (or of

length exactly 1,000, if we pad them out with blanks), and we're maintaining a set $S \subseteq U$ consisting of strings (i.e., lead paragraphs) that we've seen before.

One solution would be to keep a linked list of all paragraphs, and scan this list each time a new one arrives. But a `Lookup` operation in this case takes time proportional to $|S|$. How can we get back to something that looks like an array-based solution?

Hash Functions The basic idea of hashing is to work with an array of size $|S|$, rather than one comparable to the (astronomical) size of U.

Suppose we want to be able to store a set S of size up to n. We will set up an array H of size n to store the information, and use a function $h : U \to \{0, 1, \ldots, n-1\}$ that maps elements of U to array positions. We call such a function h a *hash function*, and the array H a *hash table*. Now, if we want to add an element u to the set S, we simply place u in position $h(u)$ of the array H. In the case of storing paragraphs of text, we can think of $h(\cdot)$ as computing some kind of numerical signature or "check-sum" of the paragraph u, and this tells us the array position at which to store u.

This would work extremely well if, for all distinct u and v in our set S, it happened to be the case that $h(u) \neq h(v)$. In such a case, we could look up u in constant time: when we check array position $H[h(u)]$, it would either be empty or would contain just u.

In general, though, we cannot expect to be this lucky: there can be distinct elements $u, v \in S$ for which $h(u) = h(v)$. We will say that these two elements *collide*, since they are mapped to the same place in H. There are a number of ways to deal with collisions. Here we will assume that each position $H[i]$ of the hash table stores a linked list of all elements $u \in S$ with $h(u) = i$. The operation `Lookup`(u) would now work as follows.

- Compute the hash function $h(u)$.
- Scan the linked list at position $H[h(u)]$ to see if u is present in this list.

Hence the time required for `Lookup`(u) is proportional to the time to compute $h(u)$, plus the length of the linked list at $H[h(u)]$. And this latter quantity, in turn, is just the number of elements in S that collide with u. The `Insert` and `Delete` operations work similarly: `Insert` adds u to the linked list at position $H[h(u)]$, and `Delete` scans this list and removes u if it is present.

So now the goal is clear: We'd like to find a hash function that "spreads out" the elements being added, so that no one entry of the hash table H contains too many elements. This is not a problem for which worst-case analysis is very informative. Indeed, suppose that $|U| \geq n^2$ (we're imagining applications where it's much larger than this). Then, for any hash function h that we choose, there will be some set S of n elements that all map to the same

position. In the worst case, we will insert all the elements of this set, and then our `Lookup` operations will consist of scanning a linked list of length n.

Our main goal here is to show that randomization can help significantly for this problem. As usual, we won't make any assumptions about the set of elements S being random; we will simply exploit randomization in the design of the hash function. In doing this, we won't be able to completely avoid collisions, but can make them relatively rare enough, and so the lists will be quite short.

Choosing a Good Hash Function We've seen that the efficiency of the dictionary is based on the choice of the hash function h. Typically, we will think of U as a large set of numbers, and then use an easily computable function h that maps each number $u \in U$ to some value in the smaller range of integers $\{0, 1, \ldots, n-1\}$. There are many simple ways to do this: we could use the first or last few digits of u, or simply take u modulo n. While these simple choices may work well in many situations, it is also possible to get large numbers of collisions. Indeed, a fixed choice of hash function may run into problems because of the types of elements u encountered in the application: Maybe the particular digits we use to define the hash function encode some property of u, and hence maybe only a few options are possible. Taking u modulo n can have the same problem, especially if n is a power of 2. To take a concrete example, suppose we used a hash function that took an English paragraph, used a standard character encoding scheme like ASCII to map it to a sequence of bits, and then kept only the first few bits in this sequence. We'd expect a huge number of collisions at the array entries corresponding to the bit strings that encoded common English words like *The*, while vast portions of the array can be occupied only by paragraphs that begin with strings like *qxf*, and hence will be empty.

A slightly better choice in practice is to take (u mod p) for a prime number p that is approximately equal to n. While in some applications this may yield a good hashing function, it may not work well in all applications, and some primes may work much better than others (for example, primes very close to powers of 2 may not work so well).

Since hashing has been widely used in practice for a long time, there is a lot of experience with what makes for a good hash function, and many hash functions have been proposed that tend to work well empirically. Here we would like to develop a hashing scheme where we can prove that it results in efficient dictionary operations with high probability.

The basic idea, as suggested earlier, is to use randomization in the construction of h. First let's consider an extreme version of this: for every element $u \in U$, when we go to insert u into S, we select a value $h(u)$ uniformly at

random in the set $\{0, 1, \ldots, n-1\}$, independently of all previous choices. In this case, the probability that two randomly selected values $h(u)$ and $h(v)$ are equal (and hence cause a collision) is quite small.

(13.22) *With this uniform random hashing scheme, the probability that two randomly selected values $h(u)$ and $h(v)$ collide—that is, that $h(u) = h(v)$—is exactly $1/n$.*

Proof. Of the n^2 possible choices for the pair of values $(h(u), h(v))$, all are equally likely, and exactly n of these choices results in a collision. ■

However, it will not work to use a hash function with independently random chosen values. To see why, suppose we inserted u into S, and then later want to perform either `Delete`(u) or `Lookup`(u). We immediately run into the "Where did I put it?" problem: We will need to know the random value $h(u)$ that we used, so we will need to have stored the value $h(u)$ in some form where we can quickly look it up. But this is exactly the same problem we were trying to solve in the first place.

There are two things that we can learn from (13.22). First, it provides a concrete basis for the intuition from practice that hash functions that spread things around in a "random" way can be effective at reducing collisions. Second, and more crucial for our goals here, we will be able to show how a more controlled use of randomization achieves performance as good as suggested in (13.22), but in a way that leads to an efficient dictionary implementation.

Universal Classes of Hash Functions The key idea is to choose a hash function at random not from the collection of all possible functions into $[0, n-1]$, but from a carefully selected class of functions. Each function h in our class of functions $\mathcal{H}$ will map the universe U into the set $\{0, 1, \ldots, n-1\}$, and we will design it so that it has two properties. First, we'd like it to come with the guarantee from (13.22):

- For any pair of elements $u, v \in U$, the probability that a randomly chosen $h \in \mathcal{H}$ satisfies $h(u) = h(v)$ is at most $1/n$.

We say that a class $\mathcal{H}$ of functions is *universal* if it satisfies this first property. Thus (13.22) can be viewed as saying that the class of all possible functions from U into $\{0, 1, \ldots, n-1\}$ is universal.

However, we also need $\mathcal{H}$ to satisfy a second property. We will state this slightly informally for now and make it more precise later.

- Each $h \in \mathcal{H}$ can be compactly represented and, for a given $h \in \mathcal{H}$ and $u \in U$, we can compute the value $h(u)$ efficiently.

The class of all possible functions failed to have this property: Essentially, the only way to represent an arbitrary function from U into $\{0, 1, \ldots, n-1\}$ is to write down the value it takes on every single element of U.

In the remainder of this section, we will show the surprising fact that there exist classes $\mathcal{H}$ that satisfy both of these properties. Before we do this, we first make precise the basic property we need from a universal class of hash functions. We argue that if a function h is selected at random from a universal class of hash functions, then in any set $S \subset U$ of size at most n, and any $u \in U$, the expected number of items in S that collide with u is a constant.

(13.23) *Let $\mathcal{H}$ be a universal class of hash functions mapping a universe U to the set $\{0, 1, \ldots, n-1\}$, let S be an arbitrary subset of U of size at most n, and let u be any element in U. We define X to be a random variable equal to the number of elements $s \in S$ for which $h(s) = h(u)$, for a random choice of hash function $h \in \mathcal{H}$. (Here S and u are fixed, and the randomness is in the choice of $h \in \mathcal{H}$.) Then $E[X] \leq 1$.*

Proof. For an element $s \in S$, we define a random variable X_s that is equal to 1 if $h(s) = h(u)$, and equal to 0 otherwise. We have $E\left[X_s\right] = \Pr\left[X_s = 1\right] \leq 1/n$, since the class of functions is universal.

Now $X = \sum_{s \in S} X_s$, and so, by linearity of expectation, we have

$$E[X] = \sum_{s \in S} E\left[X_s\right] \leq |S| \cdot \frac{1}{n} \leq 1. \quad \blacksquare$$

Designing a Universal Class of Hash Functions Next we will design a universal class of hash functions. We will use a prime number $p \approx n$ as the size of the hash table H. To be able to use integer arithmetic in designing our hash functions, we will identify the universe with vectors of the form $x = (x_1, x_2, \ldots x_r)$ for some integer r, where $0 \leq x_i < p$ for each i. For example, we can first identify U with integers in the range $[0, N-1]$ for some N, and then use consecutive blocks of $\lfloor \log p \rfloor$ bits of u to define the corresponding coordinates x_i. If $U \subseteq [0, N-1]$, then we will need a number of coordinates $r \approx \log N / \log n$.

Let $\mathcal{A}$ be the set of all vectors of the form $a = (a_1, \ldots, a_r)$, where a_i is an integer in the range $[0, p-1]$ for each $i = 1, \ldots, r$. For each $a \in \mathcal{A}$, we define the linear function

$$h_a(x) = \left(\sum_{i=1}^{r} a_i x_i \right) \bmod p.$$

This now completes our random implementation of dictionaries. We define the family of hash functions to be $\mathcal{H} = \{h_a : a \in \mathcal{A}\}$. To execute `MakeDictionary`, we choose a random hash function from $\mathcal{H}$; in other words, we choose a random vector from $\mathcal{A}$ (by choosing each coordinate uniformly at random), and form the function h_a. Note that in order to define $\mathcal{A}$, we need to find a prime number $p \geq n$. There are methods for generating prime numbers quickly, which we will not go into here. (In practice, this can also be accomplished using a table of known prime numbers, even for relatively large n.)

We then use this as the hash function with which to implement `Insert`, `Delete`, and `Lookup`. The family $\mathcal{H} = \{h_a : a \in \mathcal{A}\}$ satisfies a formal version of the second property we were seeking: It has a compact representation, since by simply choosing and remembering a random $a \in \mathcal{A}$, we can compute $h_a(u)$ for all elements $u \in U$. Thus, to show that $\mathcal{H}$ leads to an efficient, hashing-based implementation of dictionaries, we just need to establish that $\mathcal{H}$ is a universal family of hash functions.

Analyzing the Data Structure

If we are using a hash function h_a from the class $\mathcal{H}$ that we've defined, then a collision $h_a(x) = h_a(y)$ defines a linear equation modulo the prime number p. In order to analyze such equations, it's useful to have the following "cancellation law."

(13.24) *For any prime p and any integer $z \neq 0 \bmod p$, and any two integers α, β, if $\alpha z = \beta z \bmod p$, then $\alpha = \beta \bmod p$.*

Proof. Suppose $\alpha z = \beta z \bmod p$. Then, by rearranging terms, we get $z(\alpha - \beta) = 0 \bmod p$, and hence $z(\alpha - \beta)$ is divisible by p. But $z \neq 0 \bmod p$, so z is not divisible by p. Since p is prime, it follows that $\alpha - \beta$ must be divisible by p; that is, $\alpha = \beta \bmod p$ as claimed. ■

We now use this to prove the main result in our analysis.

(13.25) *The class of linear functions $\mathcal{H}$ defined above is universal.*

Proof. Let $x = (x_1, x_2, \ldots x_r)$ and $y = (y_1, y_2, \ldots y_r)$ be two distinct elements of U. We need to show that the probability of $h_a(x) = h_a(y)$, for a randomly chosen $a \in A$, is at most $1/p$.

Since $x \neq y$, then there must be an index j such that $x_j \neq y_j$. We now consider the following way of choosing the random vector $a \in \mathcal{A}$. We first choose all the coordinates a_i where $i \neq j$. Then, finally, we choose coordinate a_j. We will show that regardless of how all the other coordinates a_i were

chosen, the probability of $h_a(x) = h_a(y)$, taken over the final choice of a_j, is exactly $1/p$. It will follow that the probability of $h_a(x) = h_a(y)$ over the random choice of the full vector a must be $1/p$ as well.

This conclusion is intuitively clear: If the probability is $1/p$ regardless of how we choose all other a_i, then it is $1/p$ overall. There is also a direct proof of this using conditional probabilities. Let $\mathcal{E}$ be the event that $h_a(x) = h_a(y)$, and let $\mathcal{F}_b$ be the event that all coordinates a_i (for $i \neq j$) receive a sequence of values b. We will show, below, that $\Pr\left[\mathcal{E} \mid \mathcal{F}_b\right] = 1/p$ for all b. It then follows that $\Pr[\mathcal{E}] = \sum_b \Pr\left[\mathcal{E} \mid \mathcal{F}_b\right] \cdot \Pr\left[\mathcal{F}_b\right] = (1/p) \sum_b \Pr\left[\mathcal{F}_b\right] = 1/p$.

So, to conclude the proof, we assume that values have been chosen arbitrarily for all other coordinates a_i, and we consider the probability of selecting a_j so that $h_a(x) = h_a(y)$. By rearranging terms, we see that $h_a(x) = h_a(y)$ if and only if

$$a_j(y_j - x_j) = \sum_{i \neq j} a_i(x_i - y_i) \bmod p.$$

Since the choices for all a_i $(i \neq j)$ have been fixed, we can view the right-hand side as some fixed quantity m. Also, let us define $z = y_j - x_j$.

Now it is enough to show that there is exactly one value $0 \leq a_j < p$ that satisfies $a_j z = m \bmod p$; indeed, if this is the case, then there is a probability of exactly $1/p$ of choosing this value for a_j. So suppose there were two such values, a_j and a'_j. Then we would have $a_j z = a'_j z \bmod p$, and so by (13.24) we would have $a_j = a'_j \bmod p$. But we assumed that $a_j, a'_j < p$, and so in fact a_j and a'_j would be the same. It follows that there is only one a_j in this range that satisfies $a_j z = m \bmod p$.

Tracing back through the implications, this means that the probability of choosing a_j so that $h_a(x) = h_a(y)$ is $1/p$, however we set the other coordinates a_i in a; thus the probability that x and y collide is $1/p$. Thus we have shown that $\mathcal{H}$ is a universal class of hash functions. ■

13.7 Finding the Closest Pair of Points: A Randomized Approach

In Chapter 5, we used the divide-and-conquer technique to develop an $O(n \log n)$ time algorithm for the problem of finding the closest pair of points in the plane. Here we will show how to use randomization to develop a different algorithm for this problem, using an underlying dictionary data structure. We will show that this algorithm runs in $O(n)$ expected time, plus $O(n)$ expected dictionary operations.

There are several related reasons why it is useful to express the running time of our algorithm in this way, accounting for the dictionary operations