Chapter 7

Storage/Retrieval II: Unordered Keys

In the last chapter, we considered the problem of storage and retrieval, assuming that we also need to be able to access keys in a predefined order. In this chapter, we drop this assumption; i.e., we will be considering implementations of DICTIONARY (see Figure 6.2, p. 196) rather than ORDERED-DICTIONARY. The structures we defined in the last chapter all utilized the ordering on the keys to guide the searches. Hence, it might seem that there is nothing to be gained by neglecting to keep the keys in order. However, we will see that disorder can actually be more beneficial when it comes to locating keys quickly.

7.1 Arrays with Virtual Initialization

A simple implementation of DICTIONARY is to store all of the elements in an array indexed by keys. Though this approach is simple, it has several difficulties. The first difficulty is in using a key as an array index. For example, if our keys are strings, we must somehow be able to interpret them as natural numbers. Another difficulty is that we may have no fixed bound on the size of our keys. In this case, we would not know how large an array to construct. An expandable array would not yield a satisfactory solution because however we determine the size of the array, the next key can be so large that a new array must be constructed. Thus, we would have to expand the array each time a new key is inserted. Such an approach is clearly too expensive.

In spite of these difficulties, there is still a theoretically interesting ap-
CHAPTER 7. STORAGE/RETRIEVAL II: UNORDERED KEYS

245

approach using keys as array indices, provided we are willing to make some assumptions. First, we assume that each key is a natural number (or equivalently, each key can be treated as a natural number). Second, we assume that there is a known upper bound on the values of all of the keys. Even with these assumptions, it can still be the case that the range of the keys is much larger than the number of keys. For example, suppose our data set consists of 5,000 items keyed by 9-digit natural numbers (e.g., Social Security Numbers). An array of 1 billion elements is required to store these 5,000 items. Initializing such an array would be very expensive.

Note, however, that once an array is initialized, storage and retrieval can both be done in Θ(1) time in the worst case. What we need is a technique for initializing an array in Θ(1) time while maintaining constant-time accesses to elements. We will now present such a technique, known as virtual initialization. This technique involves keeping track of which array elements have been initialized in a way that facilitates making this determination quickly. We assume that the environment provides a facility for allocating an array in Θ(1) time without initializing its locations. We will call the resulting data structure a VArray.

In addition to an array elements[0..n − 1] to store the data, we also need an array used[0..n − 1] of Nats to keep track of which locations of elements are used to store data. We use a Nat num to keep track of how many locations of elements store data items. Thus, used[0..num − 1] will be indices at which data items are stored in elements. Finally, in order to facilitate a quick determination of whether elements[i] contains a data element, we use a third array loc[0..n − 1] such that loc[i] stores the index in used at which i is stored, if indeed i is in used[0..num − 1]. The structural invariant is that 0 ≤ num ≤ n, and for 0 ≤ i < num, loc[used[i]] = i. We interpret elements[i] as giving the data item at location i if 0 ≤ loc[i] < num and used[loc[i]] = i; otherwise, we interpret the value stored at location i as nil.

For example, Figure 7.1 shows a VArray with 10 locations, storing 35 at location 4, 17 at location 7, and nil at all other locations. Note that for i = 4 or i = 7, 0 ≤ loc[i] < num and used[loc[i]] = i. For other values of i, it is possible that loc[i] stores a natural number less than num; however, if this is the case, then used[loc[i]] is either 4 or 7, so that used[loc[i]] ≠ i.

To initialize all locations of the VArray to nil, we simply set num to 0. In this way, there is no possible value of loc[i] such that 0 ≤ loc[i] < num, so we interpret all locations as being nil. To retrieve the value at location i, we first determine whether 0 ≤ loc[i] < num and used[loc[i]] = i. Note, however, that loc[i] may not yet have been initialized, so that it may not
even refer to a Number. Therefore, we must first verify that it is a NAT. If all these tests are passed, we return elements[i]; otherwise, we return nil.

To store \textit{x} at location \textit{i} of the VARRAY, we must first determine as above whether elements[i] is currently being used to store a data item — i.e., whether $0 \leq \text{loc}[i] < \text{num}$ and used[loc[i]] = i. If so, we can simply store x in elements[i]. Otherwise, we must also update the other representation variables to reflect the fact that we are using location \textit{i} to store a data item. In particular, we must store \textit{i} in used[num], store \textit{num} in loc[i], and increment \textit{num}. As a result, we interpret elements[i] as storing \textit{x}, and the structural invariant is maintained. The entire implementation of VARRAY is shown in Figure 7.2.

It is easily seen that the constructor and all operations of VARRAY operate in $\Theta(1)$ time. However, it uses $\Theta(n)$ space, where \(n\) is the size of the range of indices, not the number of elements stored. Thus, if a VARRAY is used to implement a \textsc{Dictionary}, its space usage will be proportional to the number of possible keys. This number may be much larger than the number of keys actually stored in the \textsc{Dictionary}.

In the remainder of this chapter, we will examine techniques for improving the space usage while maintaining fast storage and retrieval. Though we cannot guarantee $\Theta(1)$ worst-case access time when we reduce the space usage, we can achieve amortized expected access time proportional to the
Figure 7.2 Implementation of VArray

**Structural Invariant:** 0 ≤ num ≤ n, and for 0 ≤ i < num, 
loc[used[i]] = i.

**Precondition:** n is a Nat.

**Postcondition:** Constructs a VARRAY with locations 0, . . . , n − 1, all initialized to nil.

VArray(n)
   elements ← new Array[0..n − 1]; used ← new Array[0..n − 1]
   loc ← new Array[0..n − 1]; num ← 0

**Precondition:** true.

**Postcondition:** Sets all locations to nil.

VArray.Clear()
   num ← 0

**Precondition:** i is a Nat less than the number of locations.

**Postcondition:** Returns the value stored at location i.

VArray.Get(i)
   if loc[i] isType Nat and loc[i] < num and used[loc[i]] = i
      return elements[i]
   else
      return nil

**Precondition:** i is a Nat less than the number of locations.

**Postcondition:** Sets the value stored at location i to x.

VArray.Put(x, i)
   if not IsNat(loc[i]) or loc[i] ≥ num or used[loc[i]] ≠ i
      loc[i] ← num; used[num] ← i; num ← num + 1
      elements[i] ← x
length of the key, even if the keys are not natural numbers and have an unbounded range. Consequently, if the keys do have a fixed range, the amortized expected access time is in $\Theta(1)$.

### 7.2 Hashing

The technique we will develop over the remainder of this chapter is known as **hashing**. The basic idea behind hashing is to convert each key $k$ to an index $h(k)$ using a hash function $h$, so that for all $k$, $0 \leq h(k) < m$ for some positive integer $m$. $h(k)$ is then used as an index into a hash table, which is an array $T[0..m-1]$. We then store the data item at that index.

Typically, the universe of keys is much larger than $m$, the size of the hash table. By choosing our array size $m$ to be close to the number of elements we need to store, we eliminate the space usage problem discussed in Section 7.1. However, because the number of possible keys will now be greater than $m$, we must deal with the problem that $h$ must map more than one potential key to the same index. When two actual keys map to the same index, it is known as a **collision**.

The potential for collisions is not just a theoretical issue unlikely to occur in practice. Suppose, for example, that we were to randomly and independently assign indices to $n$ keys, so that for any given key $k$ and index $i$, $0 \leq i < m$, the probability that $k$ is assigned $i$ is $1/m$. We can model this scenario with a discrete probability space consisting of the $m^n$ $n$-tuples of natural numbers less than $m$. Each tuple is equally likely, and so has probability $m^{-n}$. We can then define the random variable $coll$ as the number of collisions; i.e., $coll(\langle i_1, \ldots, i_n \rangle)$ is the number of ordered pairs $(i_j, i_k)$ such that $i_j = i_k$ and $j < k$.

$coll$ can be expressed as the sum of indicator random variables as follows:

$$coll(\langle i_1, \ldots, i_n \rangle) = \sum_{j=1}^{n-1} \sum_{k=j+1}^{n} I(i_j = i_k).$$
Therefore,

\[
E[\text{coll}] = E \left( \sum_{j=1}^{n-1} \sum_{k=j+1}^{n} I(i_j = i_k) \right) = \sum_{j=1}^{n-1} \sum_{k=j+1}^{n} E[I(i_j = i_k)] = \sum_{j=1}^{n-1} \sum_{k=j+1}^{n} P(i_j = i_k).
\]

For each choice of \(i, j,\) and \(i_j, i_k\) can take on \(m\) possible values, one of which is \(i_j\). Because the probabilities of all elementary events are equal, it is easily seen that \(P(i_j = i_k) = 1/m\) for \(j < k\). Hence,

\[
E[\text{coll}] = \sum_{j=1}^{n-1} \sum_{k=j+1}^{n} 1/m = \frac{1}{m} \sum_{j=1}^{n-1} (n - j) = \frac{1}{m} \sum_{j=1}^{n-1} j = \frac{n(n-1)}{2m}
\]

by (2.1).

For example, if our hash table has 500,000 locations and we have more than a thousand data elements, we should expect at least one collision, on average. In general, it requires too much space to make the table large enough so that we can reasonably expect to have no collisions.

Several solutions to the collision problem exist, but the most common is to use a linked list to store all data elements that are mapped to the same location. The approach we take here is similar, but we will use a ConsList instead of a linked list. Using a ConsList results in somewhat simpler code, and likely would not result in any significant performance degradation. This approach is illustrated in Figure 7.3.

In the remainder of this section, we will ignore the details of specific hash functions and instead focus on the other implementation details of a hash table. In order to approach the use of hash functions in a general way, we
Figure 7.3 Illustration of a hash table.

use the \texttt{HashFunction} ADT, shown in Figure 7.4. Note that because there are no operations to change the hash function, the \texttt{HashFunction} ADT specifies an immutable data type. In remaining sections of this chapter, we will consider various ways of implementing a \texttt{HashFunction}. As we will see in the next section, not all hash table sizes are appropriate for every \texttt{HashFunction} implementation. For this reason, we allow the user to select an approximate table size, but leave it up to the \texttt{HashFunction} to determine the exact table size.

Our \texttt{HashTable} representation of \texttt{Dictionary} then consists of three variables:

- \texttt{hash}: a \texttt{HashFunction} whose associated table size is some positive integer $m$;
- \texttt{table}[0..m - 1]: an array of \texttt{ConsLists}; and
- \texttt{size}: a readable \texttt{Nat}.

Our structural invariant is that:

- for $0 \leq i < \text{hash.Size}()$, \texttt{table}[i] is a \texttt{ConsList} containing only \texttt{Keyed} items;
Figure 7.4 The HashFunction ADT

**Precondition:** \( n \geq 1 \) is an Int.

**Postcondition:** Constructs a HashFunction for some table size that is at least \( n \) and strictly less than \( 3n \).

\[
\text{HashFunction}(n)
\]

**Precondition:** \( k \) refers to a Key.

**Postcondition:** Returns the index \( i \) associated with \( k \) by this HashFunction. \( i \) is a Nat strictly less than the table size.

\[
\text{HashFunction.Index}(k)
\]

**Precondition:** true.

**Postcondition:** Returns the table size for this HashFunction.

\[
\text{HashFunction.Size}()
\]

- for each Keyed item \( x \) in \( \text{table}[i] \), \( 0 \leq i < m \), hash.Index(\( x.\text{Key}() \)) = \( i \); and
- the total number of Keyed items in the ConsLists is given by size.

We interpret the Keyed items in the ConsLists to contain the elements of the data set together with their associated keys.

Accessing a data item given a key \( k \) is now straightforward — we simply compute hash.Index(\( k \)) and search the ConsList at this location in \( \text{table} \); see Figure 7.5. The worst-case time for any such operation is easily seen to be proportional to the time to compute the index plus the length of the ConsList.

Let us now consider the worst-case length of a ConsList in \( \text{table} \). Unfortunately, it can be quite bad. The following theorem shows that under reasonable assumptions, all keys can map to the same index; hence, in the worst case, the running time of a hash table access is in \( \Omega(n) \).

**Theorem 7.1** Let \( T \) be a hash table with \( m \) locations, and suppose the universe \( U \) of possible keys contains more than \( m(n - 1) \) elements. Then for any function \( h \) mapping \( U \) to natural numbers less than \( m \), there is some natural number \( i < m \) such that \( h \) maps at least \( n \) keys in \( U \) to \( i \).

The proof of the above theorem is simply the observation that if it were
Figure 7.5 The HashTable implementation of Dictionary (partial)

**Structural Invariant:** For $0 \leq i < \text{hash.Size}()$, $\text{table}[i]$ is a ConsList containing only Keyed items; for each Keyed item $x$ in $\text{table}[i]$, $0 \leq i < m$, $\text{hash.Index}(x.\text{Key}()) = i$; and the total number of Keyed items in the ConsLists is given by $\text{size}$.

**HashTable()**
\[
\text{size} \leftarrow 0; \quad \text{hash} \leftarrow \text{new HashFunction}(100) \\
\text{table} \leftarrow \text{new ARRAY}[0..\text{hash.Size}() - 1] \\
\text{for } i \leftarrow 0 \text{ to SIZEOf}(\text{table}) - 1 \\
\quad \text{table}[i] \leftarrow \text{new ConsList}()
\]

**HashTable.Get(k)**
\[
i \leftarrow \text{hash.Index}(k); \quad L \leftarrow \text{table}[i] \\
\text{while not } L.\text{isEmpty}() \\
\quad \text{if } L.\text{head}.\text{key}() = k \\
\quad \quad \text{return } L.\text{head}.\text{data}() \\
\quad L \leftarrow L.\text{tail}() \\
\text{return } \text{nil}
\]

not true — i.e., if $h$ maps at most $n - 1$ elements to each $i$ — then the size of $U$ could be at most $m(n - 1)$. Though this result looks bad, what it tells us is that we really want $h$ to produce a random distribution of the keys so that the list lengths are more evenly distributed throughout the table.

For the remainder of this section, therefore, we will assume that the key distribution is modeled by a discrete probability space $\text{hashDist}$. The elementary events in $\text{hashDist}$ are the same as those in the probability distribution defined above: all $n$-tuples of natural numbers less than $m$. Again, the $n$ positions in the tuple correspond to $n$ keys, and their values give their indices in the hash table. Regarding probabilities, however, we will make a weaker assumption, namely, the probability that any two given distinct positions are equal is at most $\epsilon$, where $0 < \epsilon < 1$. Our earlier probability space satisfies this property for $\epsilon = 1/m$, but we will see in Sections 7.4 and 7.5 that other spaces do as well.

In what follows, we will analyze the expected length of the ConsList
searched for an arbitrary key, assuming a distribution modeled by $\text{hashDist}$. In the next section we will show how to define deterministic hash functions that approximate this distribution well enough to work very well in practice. Then in Sections 7.4 and 7.5, we will show how to guarantee this behavior using randomization.

For a given search in the hash table, suppose there are a total of $n$ keys in the table together with the key for which we are searching. Thus, if the given key is in the hash table, there are $n$ keys in the hash table; otherwise, there are $n - 1$. We will use $\text{hashDist}$ to model the distribution of these $n$ keys, where the $n$th key is the one for which we are searching. Let $\text{len}$ be the discrete random variable giving the number of positions equal to position $n$ in a given element of $\text{hashDist}$. Then if the given key is in the hash table, $E[\text{len}]$ gives the expected length of the ConsList searched; otherwise, $E[\text{len}] - 1$ gives this expected length.

We can express $\text{len}$ as the sum of indicator random variables as follows:

$$\text{len} = \sum_{j=1}^{n} I(i_j = i_n),$$

where $i_j$ denotes position $j$ of an elementary event. Applying linearity of expectation, we have

$$E[\text{len}] = \sum_{j=1}^{n} E[I(i_j = i_n)] = \sum_{j=1}^{n} P(i_j = i_n).$$

Now using the fact that $P(i_j = i_n) \leq \epsilon$ when $j \neq n$, we have

$$E[\text{len}] = \sum_{j=1}^{n} P(i_j = i_n) \leq P(i_n = i_n) + \sum_{j=1}^{n-1} \epsilon = 1 + \epsilon(n - 1).$$

The above value is the expected length of the ConsList searched when the key is found in a table containing $n$ keys. If the key is not in the table, $n - 1$ gives the number of keys in the table, and $E[\text{len}]$ is one greater than
the expected length of the ConsList. Thus, if we let \( n \) denote the number of keys in the table, the length of the ConsList searched is expected to be \( n \epsilon \).

In either case, the length of the ConsList is linear in \( n \) if \( \epsilon \) is a fixed constant. However, \( \epsilon \) may depend upon \( m \). Thus, if \( \epsilon \leq c/m \) for some positive constant \( c \) and we use an expandable array for the table, we can keep the expected length bounded by a constant. Let \( \lambda = n/m \) be known as the load factor of the hash table. Using the expandable array design pattern, we can ensure that \( \lambda \leq d \), where \( d \) is a fixed positive real number of our choosing. Thus, the expected list length is bounded by

\[
1 + \epsilon n \leq 1 + cn/m \\
= 1 + c\lambda \\
\leq 1 + cd \\
\in O(1).
\]

In order to implement the expandable array pattern for a hash table, we will need to change the hash function to take advantage of the larger range of indices. In copying elements to the new table, we therefore need to apply the new hash function to each element in order to find its proper location. This technique is called rehashing. We leave it as an exercise to show that as long as the size of the table increases by at least a factor of 2 and at most a factor of 6, the amortized cost of rehashing is proportional to the cost of hashing a single key. The HashTable.Put operation, which employs rehashing, is shown in Figure 7.6.

We conclude that as long as the hash function is designed so that the probability of two arbitrary keys colliding is no more than \( c/m \), where \( m \) is the number of locations in the hash table and \( c \) is a positive real number, the amortized expected running time of a hash table access is in \( \Theta(1) \) plus the time needed to compute the hash function. We can keep the constant bounding the look-up time quite small by bounding \( \lambda \) by a small constant, provided \( c \) is not much larger than 1. A bound of \( 3/4 \) on \( \lambda \), for example, gives a nice trade-off between space and time.

### 7.3 Deterministic Hash Functions

In this section, we will consider the design of a deterministic hash function. Theorem 7.1 guarantees that any realistic deterministic hash function will result in linear-time accesses in the worst case. However, it is possible to
Figure 7.6 The Put operation for HashTable

\[
\text{HashTable.Put}(x, k) \\
i \leftarrow \text{hash.Index}(k); L \leftarrow \text{table}[i] \\
\text{while not } L.\text{isEmpty()} \\
\quad \text{if } L.\text{head().key()} = k \\
\quad \quad \text{error} \\
\quad L \leftarrow L.\text{tail()} \\
\text{table}[i] \leftarrow \text{new ConsList(new Keyed}(x, k), \text{table}[i]) \\
\quad \text{size} \leftarrow \text{size} + 1 \\
\text{if size/sizeof(table)} > \lambda \\
\quad \text{hash} \leftarrow \text{new HashFunction}(2 \cdot \text{sizeof(table)}) \\
\quad t \leftarrow \text{new Array}[0..\text{hash.size()} - 1] \\
\quad \text{for } j \leftarrow 0 \text{ to } \text{sizeof}(t) - 1 \\
\quad \quad t[j] \leftarrow \text{new ConsList()} \\
\quad \text{for } j \leftarrow 0 \text{ to } \text{sizeof}(\text{table}) - 1 \\
\quad \quad L \leftarrow \text{table}[j] \\
\quad \text{while not } L.\text{isEmpty()} \\
\quad \quad y \leftarrow L.\text{head()}; i \leftarrow \text{hash.index}(y.\text{key()}) \\
\quad \quad t[i] \leftarrow \text{new ConsList}(y, t[i]); L \leftarrow L.\text{tail()} \\
\quad \text{table} \leftarrow t
\]

construct a deterministic hash function for which such cases are very unlikely to occur in practice.

We will assume that our keys are represented as natural numbers. This assumption does not result in any loss of generality, because all data types can be viewed as sequences of bytes, or more generally, as \(w\)-bit components. We can view each component as a natural number less than \(2^w\). The sequence \(\langle k_1, \ldots, k_l \rangle\) then represents the natural number

\[
\sum_{i=1}^{l} k_i 2^{w(l-i)};
\]

thus, we view the key as encoding a natural number in radix \(2^w\). We must realize, however, that the keys may be very large, so that they do not fit in a single machine word.

The basic idea of the division method is simple. For a natural number
**Figure 7.7** The INDEX operation for the DIVISION METHOD implementation of HashFunction

```
DIVISIONMETHOD.INDEX(k)
components[1..l] ← TOARRAY(k, w); h ← 0
for i ← 1 to l
    h ← (h · 2^w + components[i]) mod size
return h
```

$k$, we define

$$h(k) = k \mod m,$$

where $m$ is the number of array locations in the hash table. Thus, $0 \leq k \mod m < m$. The table shown in Figure 7.3 uses the division method.

It is not hard to show that

$$(xy + z) \mod m = (x(y \mod m) + z) \mod m$$

(see Exercise 7.5). This relationship gives us a top-down solution to the problem of computing $h(k)$ for large $k$. If $z$ is the last $w$-bit component of $k$, we can write $k = 2^w y + z$, where $y$ is the value obtained by removing the component $z$ from $k$. We then have

$$h(k) = (2^w y + z) \mod m$$

$$= (2^w (y \mod m) + z) \mod m$$

$$= (2^w h(y) + z) \mod m.$$  

We can therefore compute $h(k)$ bottom-up by starting with the first component of $k$ and repeatedly multiplying by $2^w$, adding the next component, and taking the result mod $m$.

The division method is illustrated in Figure 7.7, where an implementation of HashFunction is presented. The representation of HashFunction is a Nat size, and the structural invariant is size > 0. We assume the existence of a function TOARRAY($x, w$), which returns an array of Nats, each strictly less than $2^w$, and which together give a representation of $x$. It is easily seen that INDEX runs in time linear in the length of the key.

One advantage of the division method is that it can be applied quickly. Because the multiplication is by a power of 2, it can be implemented by
shifting to the left by \( w \) bits. The addition then adds a \( w \)-bit number to a number whose binary representation ends in \( w \) zeros; hence, the addition can be accomplished via a bitwise or. Otherwise, there is only an application of the mod operator for each word of the key.

The effectiveness of the division method as a hash function depends on the value chosen for the table size. Knowing that we cannot prevent bad cases from ever occurring, the best we can do is to try to avoid bad behavior on cases which may be likely to occur. If data were random, our job would be much simpler, because we could take advantage of this randomness to generate a random distribution in the table. Real data sets, however, tend to contain patterns. We need our hash function to perform well in the presence of these patterns.

Suppose, for example, that the table size \( m = 255 \), and that each byte of the key is a character encoded in ASCII. From the binomial theorem ((6.15) on page 241), we can write the key as

\[
\sum_{i=1}^{l} 256^{l-i}k_i = \sum_{i=1}^{l} (255 + 1)^{l-i}k_i = \sum_{i=1}^{l} \sum_{j=0}^{l-i} \left( \begin{array}{c} l - i \\ j \end{array} \right) 255^j k_i.
\]

Each term of the inner sum such that \( j > 0 \) is divisible by 255; hence, computing the key mod 255 yields:

\[
\left( \sum_{i=1}^{l} 256^{l-i}k_i \right) \mod 255 = \sum_{i=1}^{l} k_i \mod 255.
\]

Thus, applying this hash function to the key is equivalent to applying it to the sum of the bytes in the key. Because addition is commutative, we can see that the hash function would yield the same value for any permutation of the bytes in the key; hence every permutation of the same bytes would hash to the same location. Such behavior is undesirable. Similar behavior can occur for other values very near powers of 2.

As another example, suppose that all of the keys are even numbers. If \( m \) is even, then \( k \mod m \) will always be even. As a result, only half the locations in the table are used, so we could expect the lists to be at least twice as long. More generally, if even keys are more likely than odd keys, even \( m \) will cause the division method to perform poorly. This tells us that \( m \) should not be even.
We can generalize the above arguments to conclude that \( m \) should ideally be a **prime number** — a number with no factors other than itself and 1 — and not too close to a power of two. It turns out that these restrictions are nearly always sufficient to yield good performance from the division method.

The constructor `DIVISIONMETHOD(n)` therefore needs to select a table size \( m \) that is a prime number in the range \( n \leq m < 3n \), such that \( m \) is not too close to a power of 2. Searching for such a number can be somewhat expensive.

Fortunately, we can simplify the search in practice. Consider the following sequence of prime numbers:

\[
\begin{align*}
2 & \quad 5 & \quad 11 & \quad 23 & \quad 47 & \quad 97 \\
197 & \quad 397 & \quad 797 & \quad 1597 & \quad 3203 & \quad 6421 \\
12853 & \quad 25717 & \quad 51437 & \quad 102877 & \quad 205759 & \quad 411527 \\
823117 & \quad 1646237 & \quad 3292489 & \quad 6584983 & \quad 13169977 & \quad 26339969 \\
52679969 & \quad 105359939 & \quad 210719881 & \quad 421439783 & \quad 842879579 & \quad 1685759167
\end{align*}
\]

Suppose we were to initialize an array with these values, beginning with index 1. Then for \( 2 \leq n \leq 2^{30} = 1,073,741,824 \), the value at location \( \lceil \lg n \rceil \) is at least \( n \) and strictly less than \( 3n \). Thus, by adding an extra 2 at location 0, we can easily find a prime table size in the correct range for all \( n \leq 2^{30} \), which is sufficiently large for most applications. Furthermore, except for the first three or four of these values, none are close to any power of 2. We can avoid using the first three or four sizes by setting the initial size to be sufficiently large, but even if we were to use them, rehashing guarantees that they will only be used for very small data sets.

One drawback to the `Put` operation as shown in Figure 7.6, is that when rehashing is performed, all of the hash values must be recomputed from scratch. When the division method is used on long keys, this can lead to a significant amount of computation.

An improvement is to use a combination of two hash functions. The first hash function produces an index in a range that may be too large to be used as a table size, but which is small enough to fit into a single machine word. This first hash function is called a **compression map**. The second hash function is then applied to the result of the compression map to produce an index into the actual table. Because the compression map generates an index that will fit into a single machine word, the computation of the hash table index from the compression map index can be done quickly; for example the division method would consist of a single mod operation. Thus, if we save the result of the compression map with the element we are storing, we can perform rehashing by applying a new hash function to the
The division method could be used for both of the hash functions in such a scheme. However, if the modulus is near the maximum value that can be stored in a single machine word, double-word arithmetic is required. An alternative that avoids double-word arithmetic for computing the compression map is polynomial hashing.

In order to motivate polynomial hashing, let’s consider what happens when we pack four bytes, \( k_1, k_2, k_3, \) and \( k_4 \), into a 4-byte word. If we wish to retain all of the information, we might produce the value

\[
k_1256^3 + k_2256^2 + k_3256 + k_4.
\]

Polynomial hashing generalizes this technique by producing, for a given key \( \langle k_1, \ldots, k_l \rangle \),

\[
\left( \sum_{i=1}^{l} k_ir^{l-i} \right) \mod 2^w,
\]

where \( w \) is the number of bits in a machine word, excluding any sign bit. The final “mod \( 2^w \)” describes the effect of overflow in a \( w \)-bit unsigned integer. Thus, if an unsigned integer is used, this operation need not be explicitly performed.

One reason this technique is popular is that it can be computed quickly. Note that

\[
\sum_{i=1}^{l} k_ir^{l-i} = k_l + \sum_{i=1}^{l-1} k_ir^{l-i} = k_l + r \sum_{i=1}^{l-1} k_ir^{l-1-i}.
\]

This gives us a top-down solution that can be applied bottom-up in the same way as we applied the division method directly to large keys. Specifically, we start with \( k_1 \) and repeatedly multiply by \( r \) and add the next \( k_i \). This procedure requires one multiplication and one addition for each component of the key. Furthermore, all computation can be done with single-word arithmetic.

In order for this method to work well, \( r \) must be chosen properly. We first note that 256 is a poor choice, because \( 256^i \mod 2^w = 0 \) for all \( i \geq w/8 \); thus only the first \( w/8 \) components of the key are used in computing the hash value. More generally, \( r \) should never be even, because \( (c2^j)^i \mod 2^w = 0 \)
for \( j > 0 \) and \( i \geq w/j \). Furthermore, not all odd values work well. For example, \( r = 1 \) yields \( r^i = 1 \) for all \( i \), so that the result is simply the sum of the components, mod \( 2^w \). This has the disadvantage of causing all permutations of a key to collide.

More generally, if \( r \) is odd, \( r^i \mod 2^w \) will repeat its values in a cyclic fashion. In other words, for every odd \( r \) there is a natural number \( n \) such that \( r^{n+i} \mod 2^w = r^i \) for all \( i \in \mathbb{N} \). Fortunately, there are only a few values of \( r \) (like 1) that have short cycles. In order to avoid these short cycles, we would like to choose \( r \) so that this cycle length is as large as possible. It is beyond the scope of this book to explain why, but it turns out that this cycle length is maximized whenever \( r \mod 8 \) is either 3 or 5.

We can run into other problems if \( r \) is small and the component size is smaller than \( w \). Suppose, for example that \( r = 3, w = 32 \), and each component is one byte. For any key containing fewer than 15 components, the polynomial-hash value will be less than \( 2^{31} \). We have therefore reduced the range of possible results by more than half — much more for shorter keys. As a result, more collisions than necessary are introduced. A similar phenomenon occurs if \( r \) is very close to \( 2^w \).

If we avoid these problems, polynomial hashing usually works very well as a compression map. To summarize, we should choose \( r \) so that \( r \mod 8 \) is either 3 or 5, and not too close to either 0 or \( 2^w \). This last condition can typically be satisfied if we choose an \( r \) with 5-9 bits (i.e., between 16 and 512). The division method can then be used to obtain an index into the table. Because it will be applied to a single word, its computation consists of a single mod operation.

### 7.4 Universal Hashing

Though the techniques discussed so far are widely used and work well in practice, we cannot prove much of anything useful about their performance. In this section, we consider how to use randomization to yield a hashing strategy with provably good expected behavior.

We cannot simply store data items in random array locations because we would then be unable to find them quickly. We can, however, randomly select a hash function from a set of alternatives. If the set of potential hash functions is chosen well, we can prove that the resulting hash table will be expected to have few collisions.

Let \( U \) be our universe of keys, and let \( \mathcal{H} \) be some countable set of hash functions of the form \( h : U \rightarrow M \), where \( M \) is the set of natural numbers less
than \( m \). Let us also suppose that each element of \( \mathcal{H} \) has some probability, so that \( \mathcal{H} \) is a discrete probability space. Two distinct keys \( k_1 \) and \( k_2 \) collide for \( h \in \mathcal{H} \) iff \( h(k_1) = h(k_2) \). Taking \( h(k_1) \) and \( h(k_2) \) as random variables over \( \mathcal{H} \), we see that the probability that these keys collide is \( P(h(k_1) = h(k_2)) \). If two values from \( M \) are chosen independently with uniform probability, then the probability that they are the same is \( 1/m \). We therefore say that a \( \mathcal{H} \) is a universal family of hash functions if for any two keys in \( U \), the probability that they collide is no more than \( 1/m \). As we showed in Section 7.2, this probability bound implies that for any hash table access, the expected length of the list searched is in \( \Theta(1) \).

Several universal families of hash functions have been defined, but most of them require some number theory in order to prove that they are universal families. In what follows, we present a universal family that is easier to understand at the cost of requiring a bit more computational overhead. Then in the next section, we will show how number theory can be utilized to define universal families whose hash functions can be computed more efficiently.

Suppose each key \( k \in U \) is encoded by \( l \) bits. Ideally, we would like to generate each function mapping \( U \) into \( M \) with equal probability. However, doing so is too expensive. There are \( 2^l \) keys in \( U \), and \( m \) possible values to which each could be mapped. The total number of possible hash functions is therefore \( m^{2^l} \). Uniquely identifying one of these functions therefore requires at least \( \lg \ m^{2^l} = 2^l \lg m \) bits. If, for example, each key is 32 bits and our hash table size is 256, four gigabytes of storage would be needed just to identify the hash function.

Instead, we will randomly generate a table location for each of the \( l \) bit positions. Let these locations be \( t_1, \ldots, t_l \). We will assume that \( m \) is a power of 2 so that each of these locations is encoded using \( \lg m \) bits. A given key \( k \) will select the subsequence of \( \langle t_1, \ldots, t_l \rangle \) such that \( t_i \) is included iff the \( i \)th bit of \( k \) is a 1. Thus, each key selects a unique subsequence of locations. The hash table location of \( k \) is then given by the bitwise exclusive-or of the locations in the subsequence; in other words, the binary encoding of the hash location has a 1 in position \( j \) iff the number of selected locations having a 1 in position \( j \) is odd.

**Example 7.2** Suppose our keys contain 4 bits, and we want to use a hash table with 8 locations. We then randomly generate 4 table locations, one for each of the 4 bit positions in the keys:

- \( t_1 = 3 \), or 011 in binary;
• \( t_2 = 6 \), or 110 in binary;

• \( t_3 = 0 \), or 000 in binary;

• \( t_4 = 3 \), or 011 in binary.

Note that these locations don’t need to be distinct.

Now let us compute the hash value for the keys 5 and 11, whose binary encodings are 0101 and 1011. The key 5 selects the locations \( t_2 \) and \( t_4 \), whose binary encodings are 110 and 011, respectively. The bitwise exclusive-or of these two values is 101 because the first and third bit positions each have an odd number of 1, but the second has an even number. The key 5 therefore is placed in location 5. Likewise, the key 11 selects the locations \( t_1 \), \( t_3 \), and \( t_4 \), whose binary encodings are 011, 000, and 011, respectively. The bitwise exclusive-or of these three values is 000 because each of the three bit positions contains an even number of 1s. The hash value for 11 is therefore 0.

To see why we use the exclusive-or operation, suppose we have a bit value \( x \) that is 1 with probability \( p \), \( 0 \leq p \leq 1 \). Suppose we then assign a value to bit \( y \) by flipping a fair coin; i.e., \( y \) has a value of 1 with probability 1/2, independent of the value of \( x \). The exclusive-or of \( x \) and \( y \) is 1 iff the values of \( x \) and \( y \) are different. The probability of this event is therefore

\[
\frac{p}{2} + \frac{1-p}{2} = \frac{1}{2}.
\]

Thus, the probability distribution of the exclusive-or of two independent random bits is uniform if at least one of the two has uniform probability distribution. We can easily conclude that the method outlined above for selecting a hash function results in each key mapping to any given table location with probability \( 1/m \).

However, knowing that each key maps to any given table location with probability \( 1/m \) is not sufficient to conclude that any two keys collide with probability at most \( 1/m \). Suppose, for example, that we were to select a hash function by randomly generating a natural number \( i < m \) with uniform probability. The hash function then maps all keys to \( i \). For any given key \( k \), this strategy maps \( k \) to any given location with probability \( 1/m \). However, because all keys map to the same location, the probability that two given keys collide is 1 for each pair of keys.

Before we try to prove that this family of hash functions is universal, we will define it more formally. In order to accommodate a formal definition,
we must first define a discrete probability space that will represent the set of hash functions. Let $S_{l,m}$ be the set of all $l$-tuples of bit strings of length $\lg m$, where $l$ is a positive integer and $m$ is a power of 2. Each of these $l$-tuples will represent a hash function. Note that $S_{l,m}$ has $m^l$ elements. We therefore assign each element of $S_{l,m}$ a probability of $m^{-l}$; hence, $S_{l,m}$ is a discrete probability space in which each elementary event has the same probability.

We can now formally define a hash function corresponding to each element in $S_{l,m}$. Let $\text{select}$ be the function that takes a sequence $s = \langle t_1, \ldots, t_n \rangle$ of bit strings all having the same length, together with a bit string $k_1 \cdots k_n$, and returns the subsequence of $s$ such that $t_i$ is included iff $k_i = 1$. Furthermore, let $X$ be the function that takes a sequence of bit strings each having the same length and returns their bitwise exclusive-or. Given $s = \langle t_1, \ldots, t_l \rangle \in S_{l,m}$, let $h_s : U \to M$ such that

$$h_s(k) = X(\text{select}(s, k)).$$

We now define

$$\mathcal{H}^1_{l,m} = \{ h_s \mid s \in S_{l,m} \}.$$

Each element $h \in \mathcal{H}^1_{l,m}$ corresponds to the event consisting of all sequences $s \in S_{l,m}$ such that $h = h_s$. We leave it as an exercise to show that for each $h \in \mathcal{H}^1_{l,m}$, there is exactly one such $s$; hence, there is a one-to-one correspondence between elementary events in $S_{l,m}$ and hash functions in $\mathcal{H}^1_{l,m}$. We will now show that for every distinct $k, k' \in U$, $P(h(k) = h(k')) = 1/m$, so that $\mathcal{H}^1_{l,m}$ is a universal family of hash functions. In the proof and the implementation that follows, we use $\otimes$ to denote bitwise exclusive-or.

**Theorem 7.3** Let $l$ be a positive integer and $m$ be a power of 2. Then $\mathcal{H}^1_{l,m}$ is a universal family of hash functions.

**Proof:** Suppose $k$ and $k'$ are two keys differing in bit position $i$. Without loss of generality, suppose the $i$th bit of $k$ is 0 and the $i$th bit of $k'$ is 1. Let $k''$ be the key obtained from $k'$ by changing the $i$th bit to 0. Let $t_j$ be the discrete random variable giving the value of the $j$th component of $s$ for $s \in S_{l,m}$, and let $h(x)$ be the random variable giving the hash value of $x \in U$. Then $h(k') = h(k'') \otimes t_i$. Thus, $h(k) = h(k')$ iff $h(k) = h(k'') \otimes t_i$.

Because the $i$th bits of both $k$ and $k''$ are 0, we can evaluate $h(k)$ and $h(k'')$ knowing only $t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_l$. For each choice of these values, there is exactly one value of $t_i$ for which $h(k) = h(k'') \otimes t_i$, namely $t_i =
**UniversalHash1** implementation of **HashFunction**

**Structural Invariant:** \( \text{size} = 2^i \) for some \( \text{Nat} \) \( i \), and for \( 1 \leq j \leq l \), \( \text{indices}[j] < \text{size} \).

**UniversalHash1**(\( n \))

\[
\text{size} \leftarrow 2^{\lfloor \lg n \rfloor}; \ \text{indices} \leftarrow \text{new Array}[1..l]
\]

\[
\text{for } i \leftarrow 1 \text{ to } l
\]

\[
\text{indices}[i] \leftarrow \text{Random}(\text{size})
\]

**UniversalHash1.Index**(\( k \))

\[
\text{bits}[1..l] \leftarrow \text{ToArray}(k, 1); \ h \leftarrow 0
\]

\[
\text{for } i \leftarrow 1 \text{ to } l
\]

\[
\text{if } \text{bits}[i] = 1
\]

\[
\ h \leftarrow h \otimes \text{indices}[i]
\]

\[
\text{return } h
\]

\( h(k) \otimes h(k'') \). There are then \( m^{l-1} \) hash functions for which \( k \) and \( k' \) collide. Because each hash function occurs with probability \( m^{-l} \),

\[
P(h(k) = h(k')) = m^{l-1} m^{-l} = 1/m.
\]

□

To represent an instance of this family, we use a readable \( \text{Nat} \) \( \text{size} \) and an array \( \text{indices}[1..l] \) of \( \text{Nats} \); we assume for now that \( l \), the number of bits in a key, is fixed. Our structural invariant is that \( \text{size} = 2^i \) for some natural number \( i \), and that for \( 1 \leq j \leq l \), \( \text{indices}[j] < \text{size} \). The implementation is shown in Figure 7.8. It uses a function \( \text{Random} \), which takes a positive integer \( n \) as input and returns, with uniform probability, any natural number strictly less than \( n \). It is easily seen that both the constructor and the \( \text{INDEX} \) operation run in \( \Theta(l) \) time, assuming \( \text{Random} \) runs in \( \Theta(1) \) time.

If we use this implementation of **HashFunction** with the **HashTable** implementation shown in Figures 7.5 and 7.6, the expected search time is in \( \Theta(1) \). Furthermore, it is not hard to show that the expected amortized cost of rehashing is in \( \Theta(l) \).
In many applications, the key lengths may vary, and we may not know the maximum length in advance. Such situations can be handled easily, provided we may pad keys with zeros without producing other valid keys. This padding may be done safely if the length of the key is encoded within the key, or if each key is terminated by some specific value. We can therefore consider each key as having infinite length, but containing only finitely many 1s. We can ensure that we have bit strings for indices\([1..i]\) for some \(i\). If we encounter a key with a 1 in bit position \(j > i\), we can generate bit strings for positions \(i + 1\) through \(j\) at that time. Note that neither of these strategies add any significant overhead — they simply delay the generation of the bit strings. We leave the implementation details as an exercise.

7.5 Number Theoretic Universal Hash Families

In this section, we will use some elementary number theory to obtain universal families whose hash functions resemble those of Section 7.3. The resulting functions may require less overhead than do the functions in \(\mathcal{H}_{l,m}^1\).

We first need the following fact from number theory:

**Theorem 7.4** Let \(a\), \(b\), and \(m\) be natural numbers such that \(0 < a < m\) and \(b < m\). Then the equation

\[
ai \mod m = b
\]

has a unique solution in the range \(0 \leq i < m\) iff \(a\) and \(m\) are relatively prime (i.e., 1 is the greatest common divisor of \(a\) and \(m\)).

**Proof:** Because we will only need to use this theorem in one direction, we will only prove one implication and leave the other as an exercise.

\(\Leftarrow\): Suppose \(a\) and \(m\) are relatively prime. We will show that if \(ai \mod m = aj \mod m\), where \(0 \leq i < m\) and \(0 \leq j < m\), then \(i = j\). Thus, each of the \(m\) possible values of \(i\) will result in a distinct value of \(ai \mod m\). Because only \(m\) distinct values of \(ai \mod m\) are possible, it will follow that one of the values of \(i\) must yield \(ai \mod m = b\).

Suppose \(ai \mod m = aj \mod m\). Then there exist natural numbers \(q_1\) and \(q_2\) such that

\[
a_i - q_1 m = a_j - q_2 m
\]

\[
a (i - j) = (q_1 - q_2) m,
\]
so that \( a(i - j) \) is divisible by \( m \). Because \( a \) and \( m \) are relatively prime, it must be the case that \((i - j)\) is divisible by \( m \). Given the ranges for \( i \) and \( j \), it must be the case that \(|i - j| < m\). The only multiple of \( m \) with absolute value strictly less than \( m \) is 0; hence, \( i = j \).

For our next universal family, we will interpret the keys as natural numbers and assume that there is some maximum value for a key. Let \( p \) be a prime number strictly larger than this maximum key value. Our hash functions will consist of two steps. The first step will map each key to a unique natural number less than \( p \). We will design this part so that, depending on which hash function is used, a distinct pair of keys will be mapped with uniform probability to any of the pairs of distinct natural numbers less than \( p \). The second step will apply the division method to scale the value to an appropriate range.

For the first step, let

\[
h_{p,a,b}(k) = (ak + b) \mod p
\]

for \( a \) and \( b \) strictly less than \( p \). Consider distinct keys \( k \) and \( k' \). We then have

\[
(h_{p,a,b}(k) - h_{p,a,b}(k')) \mod p = ((ak + b) \mod p - (ak' + b) \mod p) \mod p
= a(k - k') \mod p
\]

from Exercise 7.5. Because \( k - k' \neq 0 \) and \( p \) is prime, Theorem 7.4 tells us that for each natural number \( j < p \), there is a unique \( a, 0 \leq a < p \), such that

\[
j = a(k - k') \mod p
= (h_{p,a,b}(k) - h_{p,a,b}(k')) \mod p,
\]

independent of the value of \( b \). Because \( h_{p,0,b}(k) - h_{p,0,b}(k') = 0 \), each positive \( a < p \) yields a distinct positive value of \((h_{p,a,b}(k) - h_{p,a,b}(k')) \mod p\). Furthermore, for a given positive \( a \), each choice of \( b \) clearly results in a distinct value for \( h_{p,a,b}(k) \).

Each choice of \( a \) and \( b \), where \( 0 < a < p \) and \( 0 \leq b < p \), therefore results in a unique pair of distinct values \( h_{p,a,b}(k) \) and \( h_{p,a,b}(k') \). Because the number of choices of \( a \) and \( b \) is exactly the same as the number of pairs of distinct values \( h_{p,a,b}(k) \) and \( h_{p,a,b}(k') \), each of these pairs can be produced by exactly one choice of \( a \) and \( b \). We therefore have the following lemma.
Lemma 7.5 Let $p$ be a prime number, and let $k$ and $k'$ be distinct natural numbers strictly less than $p$. If $a$ and $b$ are chosen independently and uniformly such that $1 \leq a < p$ and $0 \leq b < p$, then $h_{p,a,b}(k)$ and $h_{p,a,b}(k')$ are any pair of distinct natural numbers less than $p$ with uniform probability.

To apply the second step of the hash function, let

$$f_m(i) = i \mod m,$$

where $m$ is a positive integer and $i$ is a natural number. We then define

$$\mathcal{H}^2_{p,m} = \{ f_m \circ h_{p,a,b} \mid 0 < a < p, \ 0 \leq b < p \},$$

where $\circ$ denotes function composition (i.e., $f_m \circ h_{p,a,b}(k) = f_m(h_{p,a,b}(k))$).

We define the probability of each element of $\mathcal{H}^2_{p,m}$ by selecting $a$ and $b$ independently with uniform probability. We can then show the following theorem.

Theorem 7.6 For any prime number $p$ and positive integer $m$, $\mathcal{H}^2_{p,m}$ is a universal family of hash functions.

**Proof:** Let $k$ and $k'$ be two distinct keys. As we argued above, $h_{p,a,b}(k)$ and $h_{p,a,b}(k')$ are distinct natural numbers less than $p$, and each possible pair of distinct values can be obtained by exactly one pair of values for $a$ and $b$. $f_m(h_{p,a,b}(k)) = f_m(h_{p,a,b}(k'))$ iff $h_{p,a,b}(k) \mod m = h_{p,a,b}(k') \mod m$ iff $h_{p,a,b}(k) - h_{p,a,b}(k')$ is divisible by $m$. For any natural number $i < p$, there are strictly fewer than $p/m$ natural numbers $j < p$ (other than $i$) such that $i - j$ is divisible by $m$. Because the number of these values of $j$ is an integer, it is at most $(p - 1)/m$. Because there are $p$ possible values of $h_{p,a,b}(k)$ and $p(p - 1)$ possible pairs of values for $h_{p,a,b}(k)$ and $h_{p,a,b}(k')$, each of which is equally likely, the probability that $f_m(h_{p,a,b}(k)) = f_m(h_{p,a,b}(k'))$ is at most

$$\frac{p \left( \frac{p-1}{m} \right)}{p(p-1)} = \frac{1}{m}.$$

Note that by the above theorem, $\mathcal{H}^2_{p,m}$ is universal for any positive $m$. As a result, the size of the hash table does not need to be a particular kind of number, such as a prime number or a power of 2, in order for this strategy to yield good expected performance. However, the restriction that $p$ is a
prime number larger than the value of the largest possible key places some limitations on the effectiveness of this approach. Specifically, if there is no upper bound on the length of a key, we cannot choose a \( p \) that is guaranteed to work. Furthermore, even if an upper bound is known, unless it is rather small, the sizes of \( p, a, \) and \( b \) would make the cost of computing the hash function too expensive.

Let us therefore treat keys as sequences of natural numbers strictly smaller than some value \( p \), which we presume to be not too large (e.g., small enough to fit in a single machine word). Furthermore, let us choose \( p \) to be a prime number. Let \( \langle k_1, \ldots, k_l \rangle \) be a key, and let \( s = \langle a_1, \ldots, a_l \rangle \) be a sequence of natural numbers, each of which is strictly less than \( p \). We then define

\[
    h_{p,s}(\langle k_1, \ldots, k_l \rangle) = \left( \sum_{i=1}^{l} a_i k_i \right) \mod p.
\]

We first observe that we cannot guarantee that \( h_{p,s}(k) \neq h_{p,s}(k') \) for each distinct pair of keys \( k \) and \( k' \). The reason for this is that there are potentially more keys than there are values of \( h_{p,s} \). However, suppose \( k \) and \( k' \) are distinct keys, and let \( k_i \neq k'_i \), where \( 1 \leq i \leq l \). Let us arbitrarily fix the values of all \( a_j \) such that \( j \neq i \), and let

\[
    c = \left( \sum_{j=1}^{i-1} a_j k'_j + \sum_{j=i+1}^{l} a_j k'_j - \sum_{j=1}^{i-1} a_j k_j - \sum_{j=i+1}^{l} a_j k_j \right) \mod p.
\]

Then

\[
    (h_{p,s}(k) - h_{p,s}(k')) \mod p = \left( \sum_{j=1}^{l} a_j k_j - \sum_{j=1}^{l} a_j k'_j \right) \mod p
\]

\[
    = (a_i (k_i - k'_i) + c) \mod p.
\]

Because \( 0 \leq c < p \), the above value is 0 iff

\[
    a_i (k_i - k'_i) \mod p = c.
\]

Because \( k_i \neq k'_i \) and \( p \) is prime, \( k_i - k'_i \) and \( p \) are relatively prime. From Theorem 7.4, the above equation has a unique solution for \( a_i \) such that \( 0 \leq a_i < p \). Thus, for each choice of \( a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_l \), there is exactly one choice of \( a_i \) such that \((h_{p,s}(k) - h_{p,s}(k') \mod p) = 0 \). Note that from the range of \( h_{p,s} \), the only way \((h_{p,s}(k) - h_{p,s}(k') \mod p) = 0 \) is if \( h_{p,s}(k) = h_{p,s}(k') \). We therefore have the following lemma.
Lemma 7.7 Let $p$ be a prime number and $l$ a positive integer. In addition, let $s = \langle a_1, \ldots, a_l \rangle$, where each $a_j$ is chosen independently and uniformly such that $0 \leq a_j < p$. Then the probability that $h_{p,s}(k) = h_{p,s}(k')$ for distinct keys $k$ and $k'$ is $1/p$.

We now define

$$\mathcal{H}_{p,l}^3 = \{ h_{p,s} \mid s = \langle a_1, \ldots, a_l \rangle, \ 0 \leq a_i < p \text{ for } 1 \leq i \leq l \}. \quad (7.2)$$

We define the probability of each element of $\mathcal{H}_{p,l}^3$ by selecting each $a_i$ independently with uniform probability. Note that the range of each hash function in $\mathcal{H}_{p,l}^3$ is the set of natural numbers strictly less than $p$. Therefore, based on the above discussion, we have the following theorem.

Theorem 7.8 For any prime number $p$ and positive integer $l$, $\mathcal{H}_{p,l}^3$ is a universal family of hash functions.

If we know in advance the approximate size of our data set and the maximum key length, we can select an appropriate prime value for $p$ and randomly select the appropriate hash function from $\mathcal{H}_{p,l}^3$. Because we can apply the mod operation after each addition, we are always working with values having no more than roughly twice the number of bits as $p$; hence, we can compute this hash function reasonably quickly for each key. Furthermore, even if we don’t know the maximum key length, we can generate the multipliers $a_i$ as we need them.

However, if we don’t know in advance the approximate size of the data set, we may need to use rehashing. For the sake of efficiency, we would like to avoid the need to apply a new hash function to the entire key. Furthermore, as we will see in the next section, it would be useful to have a universal family that is appropriate for large keys and for which the table size is unrestricted.

A straightforward attempt to achieve these goals is to combine $\mathcal{H}_{p,l}^3$ with $\mathcal{H}_{p,m}^2$. Specifically, we define

$$\mathcal{H}_{p,l,m}^4 = \{ h_1 \circ h_2 \mid h_1 \in \mathcal{H}_{p,m}^2, h_2 \in \mathcal{H}_{p,l}^3 \}. \quad (7.3)$$

Hash functions in this family are of the form,

$$h(k) = h_{p,a,b}(h_{p,s}(k)) \mod m$$

$$= \left( a \sum_{i=1}^{l} a_i k_i + b \right) \mod p \mod m$$

(7.3)
where \(a, b\), and each \(a_i\) are natural numbers, and \(a \neq 0\). We define the probability of each element of \(\mathcal{H}_{p,l,m}^4\) by selecting \(a, b\), and each \(a_i\) independently with uniform probability. (We leave it as an exercise to show that the same probability distribution for \(\mathcal{H}_{p,l,m}^4\) can be achieved by setting \(a = 1\) and selecting \(b\) and each \(a_i\) independently with uniform probability.)

Because \(\mathcal{H}_{p,m}^2\) is a universal family, it causes any pair of distinct keys to collide with probability at most \(1/m\). However, \(\mathcal{H}_{p,l}^3\) also causes distinct keys to collide with probability \(1/p\). When the function from \(\mathcal{H}_{p,m}^2\) is applied to equal values, it yields equal values. We must therefore be careful in analyzing the probability of collisions for \(\mathcal{H}_{p,l,m}^4\).

Let us first consider the case in which two distinct keys \(k\) and \(k'\) are mapped to distinct values by \(h_{p,s} \in \mathcal{H}_{p,l}^3\). From Lemma 7.7, the probability that this occurs is

\[
1 - \frac{1}{p} = \frac{p - 1}{p}.
\]

Furthermore, from Lemma 7.5, \(h_{p,a,b}(h_{p,s}(k))\) and \(h_{p,a,b}(h_{p,s}(k'))\) are with uniform probability any pair of distinct natural numbers less than \(p\), provided \(a\) and \(b\) are chosen independently with uniform probability such that \(1 \leq a < p\) and \(0 \leq b < p\). Because there are \(p(p-1)\) pairs of distinct natural numbers less than \(p\), this probability is

\[
\frac{1}{p(p-1)}.
\]

Therefore, given any two distinct keys \(k\) and \(k'\), and any two distinct natural numbers \(i\) and \(j\) strictly less than \(p\), the probability that \(h_{p,a,b}(h_{p,s}(k)) = i\) and \(h_{p,a,b}(h_{p,s}(k')) = j\) is

\[
\left(\frac{p - 1}{p}\right) \left(\frac{1}{p(p-1)}\right) = \frac{1}{p^2}.
\]

Now consider the case in which \(h_{p,s}(k) = h_{p,s}(k')\). From Lemma 7.7, this case occurs with probability \(1/p\). For any value of \(a\), \(1 \leq a < p\), and any value of \(i\), \(0 \leq i < p\), there is exactly one value of \(b\) such that \(0 \leq b < p\) and

\[(ah_{p,s}(k) + b) \mod p = i.
\]

Thus, each value of \(i\) is reached with probability \(1/p\). Therefore, for each natural number \(i < p\), the probability that \(h_{p,a,b}(h_{p,s}(k)) = h_{p,a,b}(h_{p,s}(k')) = i\) is \(1/p^2\).

Thus, for a hash function \(h\) chosen from \(\mathcal{H}_{p,l,m}^4\), \(h(k) = i \mod m\) and \(h(k') = j \mod m\), where \(i\) and \(j\) are natural numbers less than \(p\) chosen...
independently with uniform probability. Furthermore, \( i \mod m = j \mod m \) iff \( i - j \) is divisible by \( m \). Because \( p - (p \mod m) \) is divisible by \( m \), for any \( i \), exactly 1 of every \( m \) values \( j \) such that \( 0 \leq j < p - (p \mod m) \) is such that \( i - j \) is divisible by \( m \). Likewise, for any \( j \), exactly 1 of every \( m \) values \( i \) such that \( 0 \leq i < p - (p \mod m) \) is such that \( i - j \) is divisible by \( m \) (see Figure 7.9). Thus, of the \( p^2 - (p \mod m)^2 \) pairs in which at least one value is less than \( p - (p \mod m) \), exactly

\[
\frac{p^2 - (p \mod m)^2}{m}
\]

pairs result in collisions. Of the remaining \((p \mod m)^2\) pairs, only those in which \( i = j \) result in collisions. There are exactly \( p \mod m \) such pairs.
Thus, the probability of a collision is exactly

\[
\frac{p^2 - (p \mod m)^2}{mp^2} + \frac{p \mod m}{p^2} = \frac{p^2 + m(p \mod m) - (p \mod m)^2}{mp^2} = \frac{1}{m} + \frac{(m - (p \mod m))(p \mod m)}{mp^2}. \tag{7.4}
\]

Clearly, \(m - (p \mod m)\) is always positive and \(p \mod m\) is always non-negative. Furthermore, because \(p\) is prime, the only way \(p \mod m\) can be 0 is if \(m = 1\) or \(m = p\). A hash table of size 1 is simply a ConsList, and selecting \(m = p\) would defeat the purpose of combining \(H^3_{p,l}\) with \(H^2_{p,m}\). Thus, for all reasonable values of \(m\), the second fraction on the right-hand side of (7.4) is strictly positive; therefore, \(H^4_{p,l,m}\) is not a universal family.

However, recall from Section 7.2 that \(\Theta(1)\) amortized expected performance can be achieved using rehashing if the probability of collisions is bounded by \(c/m\) for some positive real number \(c\). We therefore define a family of hash functions to be \(c\)-universal if for each pair of distinct keys, the probability of a collision is at most \(c/m\). In what follows, we will derive a \(c\) such that \(H^4_{p,l,m}\) is \(c\)-universal whenever \(1 < m < p\).

Specifically, we need to find a real number \(c\) such that whenever \(p\) is prime and \(1 < m < p\),

\[
\frac{c}{m} \geq \frac{1}{m} + \frac{(m - (p \mod m))(p \mod m)}{mp^2} \geq 1 + \frac{(m - (p \mod m))(p \mod m)}{p^2}. \tag{7.5}
\]

Let us fix \(p\) at an arbitrary prime value and try to maximize the right-hand side of (7.5). Suppose \(m < p/2\). Then \(p - (p \mod m) > m\), and

\[
p \mod (p - p \mod m) = p \mod m.
\]

Thus, by replacing \(m\) with \(p - p \mod m\) in (7.5), we increase the value of the expression. In order to maximize its value, we therefore need \(m \geq p/2\). Because \(p\) is prime, we cannot have \(m = p/2\), so we can assume \(m > p/2\).

If \(m > p/2\), then \(p \mod m = p - m\). It therefore suffices to maximize

\[
(m - (p - m))(p - m) = (2m - p)(p - m) = -(2m^2 - 3mp + p^2),
\]

or equivalently, to minimize

\[
f(m) = 2m^2 - 3mp + p^2.
\]
There are several ways to find the minimum value of a quadratic, but one way that does not involve calculus is by the technique of completing the square. A quadratic of the form \((ax - b)^2\) is clearly nonnegative for all values of \(a, x,\) and \(b\). Furthermore, it reaches a value of 0 (its minimum) at \(x = b/a\). We can therefore minimize \(f(m)\) by finding a value \(d\) such that \(f(m) - d\) is of the form

\[
(am - b)^2 = a^2m^2 - 2abm + b^2.
\]

Because \(f(m) - d\) reaches a minimum value of 0 at \(m = b/a\), \(f(m)\) reaches a minimum value of \(d\) at the same point.

In order to make the coefficient of \(m^2\) have a value of 2, \(a\) must be \(\sqrt{2}\).

To find the coefficient \(b\), we must solve

\[
3mp = 2\sqrt{2}bm
\]

\[
b = \frac{3p}{2\sqrt{2}}.
\]

To find \(d\), we must then solve

\[
p^2 - d = \left(\frac{3p}{2\sqrt{2}}\right)^2
\]

\[
= \frac{9p^2}{8}
\]

\[
d = \frac{p^2}{8}.
\]

Thus, \(-f(m)\) — and hence the numerator of the second term in the right-hand side of (7.5) — is never more than \(p^2/8\). Furthermore, this value is achieved (assuming for the moment that \(m\) varies over the real numbers) when

\[
m = \frac{3p}{2\sqrt{2}}
\]

\[
= \frac{3p}{4}.
\]

We conclude that the right-hand side of (7.5) is bounded above by

\[
1 + \frac{p^2/8}{p^2} = 9/8.
\]

We therefore have the following theorem.
Theorem 7.9 For any prime number $p$ and positive integers $l$ and $m$ such that $1 < m < p$, $H^l_{p,l,m}$ is $9/8$-universal.

The upper bound of $9/8$ can be reached when $m = 3p/4$; however, in order for this equality to be satisfied, $p$ must be a multiple of 4, and hence cannot be prime. We can, however, come arbitrarily close to this bound by using a sufficiently large prime number $p$ and setting $m$ to either $\lceil 3p/4 \rceil$ or $\lfloor 3p/4 \rfloor$. Practically speaking, though, such values for $m$ are much too large. In practice, $m$ would be much smaller than $p$, and as a result, the actual probability of a collision would be much closer to $1/m$.

By choosing $p$ to be of an appropriate size, we can choose a single $h$ of the form

$$h(k) = \left(a \sum_{i=1}^{l} a_i k_i + b\right) \mod p,$$

and change $m$ as we need to rehash. For example, if the maximum array size on a given platform is $2^{31} - 1$, which by happy coincidence is prime, we can set $p$ to this value. We can then break the keys into 2- or 3-byte components and randomly select $a, b, a_1, \ldots, a_l$. We can select any value of $m < 2^{31} - 1$ as our table size, but a power of 2 works particularly well, because $h(k) \mod m$ is just the low-order $\lg m$ bits of $h(k)$. As we compute the hash value $h(k) \mod m$ for each key $k$, we save the value $h(k)$. Note that this value can be computed using 64-bit arithmetic and stored as a 32-bit (signed or unsigned) integer. If we need to rehash, we double the size of the table. We can then compute the new hash values for each $k$ by looking up $h(k)$ and computing $h(k) \mod 2m$.

If $p = 2^{31} - 1$ and $m$ is a power of 2, then $p \mod m = m - 1$. Substituting this value into (7.4), we see that the probability of a collision is

$$\frac{1}{m} + \frac{m - 1}{mp^2} < \frac{1}{m} + \frac{1}{(2^{31} - 1)^2} < \frac{1}{m} + 2^{-61}.$$

### 7.6 Perfect Hashing

In this section, we consider a restricted form of Dictionary for which PUT and REMOVE are not allowed; i.e., updates will never be made to the structure after it is created. In order for such a structure to be useful, we need to modify the constructor to receive as input the elements to be stored.
Figure 7.10 The ImmutableDictionary ADT

**Precondition:** elements[0..n − 1] is an array of non-nil items, keys[0..n − 1] is an array of distinct Keys, n ∈ N.

**Postcondition:** Constructs a ImmutableDictionary containing all of the items in elements[0..n − 1] using keys[i] as the key for elements[i] for 0 ≤ i < n.

```
ImmutableDictionary(elements[0..n − 1], keys[0..n − 1])
```

**Precondition:** k is a key.

**Postcondition:** Returns the item with key k, or nil if no item with key k is contained in the set.

```
ImmutableDictionary.Get(k)
```

**Precondition:** true.

**Postcondition:** Returns the number of items in the set.

```
ImmutableDictionary.Size()
```

The formal specification of the ImmutableDictionary ADT is shown in Figure 7.10.

If we expect to make a large number of accesses to an ImmutableDictionary, it might make sense to invest more time in constructing it if we can then guarantee that accesses will be fast. To achieve this goal, we use a technique called perfect hashing.

One of the drawbacks to hashing is that we can’t guarantee that there will be no collisions. In fact, we can’t even guarantee that all of keys don’t hash to the same location. Universal hashing gives us an expectation that the resulting hash table will not have too many collisions. Thus, even though we might be unlucky and choose a hash function that yields poor performance on our data set, if we randomly select several different hash functions, we can expect to find one that yields a small number of collisions.

With perfect hashing, our goal is to produce a hash table with no collisions. Unfortunately, as we saw in Section 7.2, unless the size of the hash table is much larger than the number of keys, we can expect to have at least one collision. With a reasonable table size, we would probably need to try many different hash functions before we found one that yielded no collisions.

We can avoid this difficulty, however, by employing a two-level approach
Figure 7.11 The structure of a perfect hash table

(see Figure 7.11). Instead of using a CONSList to store all of the elements that hash to a certain location, we use a secondary hash table with its own hash function. The secondary hash tables that store more than one element are much larger than the number of elements they store. As a result, we will be able to find a hash function for each secondary hash table such that no collisions occur. Furthermore, we will see that the sizes of the secondary hash tables can be chosen so that the total number of locations in all of the hash tables combined is linear in the number of elements stored.

Let us first determine an appropriate size $m$ for a secondary hash table in which we need to store $n$ distinct keys. We saw in Section 7.2 that in order for the expected number of collisions to be less than 1, if the probability that two keys collide is $1/m$, then $m$ must be nearly $n^2$. We will therefore assume that $m \geq n^2$.

Let $\mathcal{H}_m$ be a $c$-universal family of hash functions. We wish to determine an upper bound on the number of hash functions we would need to select from $\mathcal{H}_m$ before we can expect to find one that produces no collisions among
the given keys. Let \( \text{coll} \) be the discrete random variable giving the total number of collisions, as defined in Section 7.2, produced by a hash function \( h \in \mathcal{H}_m \) on distinct keys \( k_1, \ldots, k_n \). As we showed in Section 7.2,

\[
E[\text{coll}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} P(h(k_i) = h(k_j)).
\]

Because the probability that any two distinct keys collide is no more than \( c/m \leq c/n^2 \), we have

\[
E[\text{coll}] \leq \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{c}{n^2} = \frac{c}{n^2} \sum_{i=1}^{n-1} (n - i) = \frac{c}{n^2} \sum_{i=1}^{n-1} i \quad \text{(reversing the sum)}
\]

\[
= \frac{cn(n-1)}{2n^2} \quad \text{(by (2.1))}
\]

\[
< c/2.
\]

From Markov’s Inequality (5.3) on page 194, the probability that there is at least one collision is therefore less than \( c/2 \).

Suppose, for example, that \( c = 1 \), as for a universal hash family. Then the probability that a randomly chosen hash function results in no collisions is greater than \( 1/2 \). If \( c = 9/8 \), as for \( \mathcal{H}_{p,l,m}^4 \), then the probability is greater than \( 7/16 \). Suppose we repeatedly select hash functions and try storing the keys in the table. Because the probability that there are no collisions is positive whenever \( c < 2 \), we will eventually find a hash function that produces no collisions.

Let us now determine how many hash functions we would expect to try before finding one that results in no collisions. Let \( \text{reps} \) be the discrete random variable giving this number. For a given positive integer \( i \), \( P(\text{reps} \geq i) \) is the probability that \( i - 1 \) successive hash functions fail; i.e.,

\[
P(\text{reps} \geq i) < \frac{1}{2}^{i-1} = \left(\frac{2}{c}\right)^{1-i}.
\]
From Theorem 5.5,

$$E[\text{reps}] = \sum_{i=1}^{\infty} P(\text{reps} \geq i)$$

$$< \sum_{i=1}^{\infty} \frac{(2/c)^{-i}}{1}.$$  

Suppose $c < 2$. Then we can re-index the sum to begin at 0 and apply Theorem 6.7, yielding

$$E[\text{reps}] < \sum_{i=0}^{\infty} \frac{(2/c)^{-i}}{1}$$

$$= \frac{2}{(2/c) - 1}$$

$$= \frac{2}{2 - c}.$$  

Note that the above value is a fixed constant for fixed $c < 2$. Thus, the expected number of attempts at finding an appropriate secondary hash function is bounded by a fixed constant. For example, with $c = 1$, the value of this constant is less than 2, or with $c = 9/8$, the value is less than $16/7$. As a result, we would expect that the number of times a secondary hash function is applied to any key during the process placing keys in secondary hash tables is bounded by a constant.

We must now ensure that the total space used by the primary and secondary hash tables (and hence the time needed to initialize them) is linear in $n$, the total number of keys. Suppose the primary hash table has $m$ locations. Further suppose that $n_i$ keys are mapped to index $i$ in the primary hash table. We will then construct a HashFunction by passing $n_i^2$ to the constructor of an implementation providing a $c$-universal hash family. Due to the specification of the HashFunction constructor, the actual HashFunction constructed may contain up to $3n_i^2 - 1$ locations when $n_i > 0$. The size of the table constructed is therefore linear in $n_i^2$. The actual space used by all of the secondary hash tables is therefore linear in

$$\sum_{i=0}^{m-1} n_i^2.$$  

Let $\text{sumsq}$ be a discrete random variable denoting the above sum. The expected space usage of the secondary hash tables is then linear in $E[\text{sumsq}]$. 

In order to analyze $E[\text{sumsq}]$, we first observe that $n_i^2$ is closely related to the number of collisions at index $i$. The number of collisions at index $i$ is $n_i(n_i - 1)/2$, so that

$$E[\text{coll}] = E\left[\sum_{i=0}^{m-1} \frac{n_i(n_i - 1)}{2}\right]$$

$$= \frac{1}{2} \left( E\left[\sum_{i=0}^{m-1} n_i^2\right] - E\left[\sum_{i=0}^{m-1} n_i\right]\right)$$

$$= (E[\text{sumsq}] - E[n])/2$$

$$= (E[\text{sumsq}] - n)/2.$$

Rearranging terms, we have

$$E[\text{sumsq}] = 2E[\text{coll}] + n.$$

By reasoning as in Section 7.2, it is easily seen that if the probability that two keys collide is at most $c/m$, then

$$E[\text{coll}] \leq \frac{cn(n-1)}{2m}.$$

Hence,

$$E[\text{sumsq}] = 2E[\text{coll}] + n$$

$$\leq \frac{cn(n-1)}{m} + n.$$ (7.6)

Thus, if $m \in \Theta(n)$, the expected number of locations in the primary hash table and all of the secondary hash tables is in $\Theta(n)$. In particular, if $m \geq n$, then $E[\text{sumsq}] \leq (c + 1)n$. It turns out that the value of $m$ that minimizes

$$\frac{cn(n-1)}{m} + n + m$$

is roughly $n$ (see Exercise 7.17); hence, we will construct our primary hash function by passing $n$ to constructor for an appropriate implementation of HashFunction.

Of course, we could be unlucky in selecting a primary hash function, so that the number of secondary locations is much larger than what we expect. For example, if it happens that all keys hash to the same location, then a single secondary hash table with at least $n^2$ locations will be used. In order
to guarantee linear space usage in the worst case, we therefore need to select primary hash functions repeatedly until we get one that yields a reasonable total space usage. Because the space usage is linear in \( \text{sumsq} \), we don’t need to construct the actual secondary hash tables in order to determine whether the space usage is reasonable — we can instead simply compute \( \text{sumsq} \). We should therefore determine some maximum acceptable value for \( \text{sumsq} \).

In order to ensure a reasonable probability of success, we don’t want this maximum value to be too small. From Markov’s Inequality (5.3), the probability that a discrete random variable is at least twice its expected value is at most \( \frac{1}{2} \), provided its expected value is strictly positive. Based on (7.6) above, because we will be using a primary table size of at least \( n \), it makes sense to use \( 2(c + 1)n \) as the maximum allowable value for \( \text{sumsq} \). Furthermore, our derivations have assumed that \( c < 2 \); hence, we can simplify the maximum allowable value to \( 6n \). By using this maximum, we would expect to select no more than 2 primary hash functions, on average, and still guarantee linear space usage.

We represent an \texttt{ImmutableDictionary} with the following variables:

- \texttt{table[0..m-1]}: an array of (possibly \texttt{nil}) arrays of (possibly \texttt{nil}) \texttt{Keyed} elements;
- \texttt{hash}: a \texttt{HashFunction};
- \texttt{functions[0..m – 1]}: an array of (possibly \texttt{nil}) \texttt{HashFunction}s; and
- \texttt{size}: a readable \texttt{Nat}.

Our structural invariant is that:

- the size of \texttt{hash} is the number of locations \( m \) in \texttt{table};
- for \( 0 \leq i < m \), \texttt{table[i]} is \texttt{nil} iff \texttt{functions[i]} is \texttt{nil};
- if \texttt{table[i]} \neq \texttt{nil}, then the array stored there is indexed 0..\( s – 1 \), where \( s \) is the size of \texttt{functions[i]};
- if an element with key \( k \) is stored at \texttt{table[i][j]}, then \( \texttt{hash.index(k)} = i \) and \( \texttt{functions[i].index(k)} = j \); and
- \( \texttt{size} = n \), the total number of keys stored.

We interpret the \texttt{Keyed} items stored as the elements of the \texttt{ImmutableDictionary}, together with their associated keys.
The implementation of the constructor is shown in Figure 7.12. Based on the above discussion, the first repeat loop is expected to iterate no more than twice. Furthermore, it is easily seen that each iteration of this loop runs in $\Theta(nf(l) + g(n))$ time in the worst case, where $g(n)$ is the time required by the HashFunction constructor, and $f(l)$ is the time required by the INDEX operation on keys of length $l$. In order to simplify the discussion that follows, we will assume that the HashFunction constructor runs in $O(n)$ time; consequently, the expected running time of this loop is in $\Theta(nf(l))$.

The effect of this loop is to create a HashFunction hash and an array $t[0..\text{hash}\.\text{Size}() - 1]$ that is essentially an ordinary hash table for hash.

The analysis of the remainder of the algorithm is somewhat more involved, but again relies heavily on the above discussion. The outer loop constructs a secondary hash table for each $t[i]$ and places it in $\text{table}[i]$. Each iteration of the repeat loop generates a hash function and attempts to construct a secondary hash table for the elements at $t[i]$. It iterates until it has a secondary hash table with no collisions. By the above discussion, the expected number of iterations is in $\Theta(1)$. Let $n_i$ be the number of elements at $t[i]$. Then the for loop iterates $n_i^2$ times, and the while loop iterates $n_i$ times in the worst case. Therefore, the repeat loop requires $\Theta(n_i^2 + n_i f(l))$ expected time.

The expected running time of the entire for loop is then in

$$\sum_{i=0}^{m-1} \Theta(n_i^2 + n_i f(l)) = \sum_{i=0}^{m-1} \Theta(n_i^2) + \sum_{i=0}^{m-1} \Theta(n_i f(l))$$

$$= \Theta\left(\sum_{i=0}^{m-1} n_i^2\right) + \Theta\left(f(l) \sum_{i=0}^{m-1} n_i\right)$$

$$= \Theta((c + 1)n) + \Theta(nf(l))$$

$$= \Theta(nf(l)).$$

The total expected running time of the constructor is therefore in $\Theta(nf(l))$.

Thus, for $\cal{H}_{p,m}^2$, the constructor runs in $\Theta(n)$ expected time, and for $\cal{H}_{p,l,m}^4$, the constructor runs in $\Theta(nl)$ expected time. It is not hard to show that the constructor runs in $\Theta(nl)$ expected time for $\cal{H}_{l,m}^1$ as well; the details are left as an exercise.

It is easily seen that the space required by the ImmutableDictionary is in $\Theta(n)$ in the worst case. However, the hidden constants can be rather large. Specifically, the primary hash table can contain nearly $3n$ locations, and the secondary hash tables can contain a total of nearly $18n$ locations. As a result, nearly $21n$ array locations can be used to store $n$ data items.
Figure 7.12 Constructor for PerfectHash implementation of ImmutableDictionary

PerfectHash(elements[0..n − 1], keys[0..n − 1])
size ← n
repeat
  hash ← new HashFunction(n); m ← hash.Size()
  t ← new Array[0..m − 1]; count ← new Array[0..m − 1]
  for i ← 0 to m − 1
    t[i] ← new ConsList(); count[i] ← 0
  for i ← 0 to n − 1
    h ← hash.Index(keys[i])
    t[h] ← new ConsList(new Keyed(elements[i], keys[i]), t[h])
    count[h] ← count[h] + 1
  sum ← 0
  for i ← 0 to m − 1
    sum ← sum + count[i]²
  until sum ≤ 6n
  table ← new Array[0..m − 1]; functions ← new Array[0..m − 1]
  for i ← 0 to m − 1
    if t[i].isEmpty()
      table[i] = nil; functions[i] = nil
    else
      repeat
        functions[i] = new HashFunction(count[i]²)
        table[i] = new Array[0..functions[i].Size() − 1]
        for j ← 0 to SizeOf(table[i]) − 1
          table[i][j] ← nil
        L ← t[i]; coll = false
        while not L.isEmpty() and not coll
          h ← functions[i].index(L.head().key())
          if table[i][h] = nil
            table[i][h] ← L.head(); L ← L.tail()
          else
            if table[i][h].key() = L.head().key()
              error
            else
              coll ← true
        until L.isEmpty()
PerfectHash.Get(k)
    \( h \leftarrow \text{hash.Index}(k) \)
    \[
    \text{if } \text{table}[h] = \text{nil} \\
    \text{return nil} \\
    \text{else} \\
    \text{return } \text{table}[h][\text{functions}[h].\text{INDEX}(k)]
    \]

However, observe that the constructor for the family \( H_{l,m}^1 \) will return a hash function with size less than \( 2n \), and the constructors for the families \( H_{p,m}^2 \) and \( H_{p,t,m}^4 \) both return hash functions with size \( n \). Furthermore, if we were to fix a specific \( c \)-universal family of hash functions, we could reduce the bound on the first repeat loop to \( 2(c + 1)n \).

Combining the above results, we see that the worst-case total number of array locations can be reduced to:

- 10\( n \) for \( H_{l,m}^1 \);
- 5\( n \) for \( H_{p,m}^2 \); or
- 21\( n/4 \) for \( H_{p,t,m}^4 \).

Finally, we observe that because \( E[\text{sumsq}] < (c + 1)n \), the expected total number of array locations is no more than

- 6\( n \) for \( H_{l,m}^1 \);
- 3\( n \) for \( H_{p,m}^2 \); or
- 25\( n/8 \) for \( H_{p,t,m}^4 \).

These last bounds hold regardless of whether we change the bound on the first repeat loop.

The Get operation is shown in Figure 7.13. It clearly runs in \( \Theta(f(l)) \) time, where \( f(l) \) is the time needed to compute the hash function on a key of length \( l \).
7.7 Summary

If keys are natural numbers, we can implement Dictionary using a VArray and thus achieve constant-time accesses in the worst case. However, the space usage of a VArray makes it impractical. For this reason, hash tables are the preferred implementation in practice. Furthermore, hashing can be done for arbitrary types of keys.

Deterministic hashing yields data accesses that, in practice, run in amortized time proportional to the length of the key, independent of the number of data items in the set. This compares very well to the structures presented in Chapter 6, which give \( \Theta(\lg n) \) access times, where \( n \) is the number of data items in the set. In our analyses in Chapter 6, we did not consider the key length. Our analyses thus implicitly assumed that keys could be compared in constant time. Each of the structures in Chapter 6 require \( \Theta(\lg n) \) comparisons in either the worst, amortized, or expected case, depending on the structure. In the worst case, each of these comparisons requires a time proportional to the length of the key. As a result, the performance of deterministic hashing is usually significantly better in practice than those structures given in Chapter 6. The trade-off is that hash tables do not permit fast access to all of the keys in a predetermined order.

The division method, which computes the value of the key mod the table size, is the most common type of hash function. In order for this method to work well, the table size should be a prime number that is not too close to a power of 2. The division method is often combined with polynomial hashing in order to produce a single-word index, which can then be converted to locations in tables of different sizes. Polynomial hashing involves multiplying each component of the key by a radix raised to successively higher powers, retaining only those bits that will fit in a single machine word. The radix \( r \) should use 5 to 9 bits, and should be such that \( r \mod 8 \) is either 3 or 5.

Though it works very well in practice, in the worst case, deterministic hashing results in accesses having a running time in \( \Theta(n) \). We can achieve better theoretical results using universal hashing, in which a hash function is selected at random from a universal family of hash functions. When universal hashing is used, data accesses have an expected amortized running time proportional to the key length.

Perfect hashing is an application of universal hashing which produces an ImmutableDictionary. Using the inherent randomization in universal hashing, we can construct an ImmutableDictionary in expected time linear in the sum of the key lengths. Retrievals can then be performed by
computing two hash functions — no searching is required. However, if the 
keys are long, the cost of computing a second hash function may exceed the 
cost of searching for the key in an ordinary hash table.

7.8 Exercises

Exercise 7.1 Prove that VARRAY, shown in Figure 7.2, meets its specifi-
cation.

Exercise 7.2 Give an algorithm that takes as input an array $A[1..n]$ of 
natural numbers and returns an array $B[1..n]$ such that for $1 \leq i \leq n$, $B[i]$ 
gives the last location in $A$ that contains $A[i]$. Your algorithm must run in 
$O(n)$ time in the worst case, and you may make no assumptions about how 
large the elements in $A$ are. Prove the correctness and time complexity of 
your algorithm. [Hint: Use a VARRAY.]

Exercise 7.3 Complete the implementation of HashTable shown in Fig-
ures 7.5 (p. 252) and 7.6 (p. 255) by adding a REMOVE operation as specified 
in Figure 6.2 (p. 196).

Exercise 7.4 Prove that if the cost of rehashing, as implemented in Figure 
7.6 (p. 255), is amortized over all PUT and REMOVE operations, the amor-
tized cost of rehashing is proportional to the cost of computing the index 
for a single key.

Exercise 7.5 Prove the following for all integers $x$ and $y$ and all positive 
integers $m$:

a. $(x + (y \mod m)) \mod m = (x + y) \mod m$.

b. $(x(y \mod m)) \mod m = (xy) \mod m$.

c. $(-(x \mod m)) \mod m = (-x) \mod m$.

Exercise 7.6 Show the hash table that results from inserting the following 
keys in the order listed, assuming the division method is used with a table 
of size 13:

$27, 36, 14, 40, 42, 15, 25, 2$.

You may assume that no rehashing is done. How does the number of colli-
sions, as defined by the random variable $coll$ in Section 7.2, compare with 
the expected number, assuming that distinct keys collide with probability 
$1/13$?
Exercise 7.7 Give a modified version of `HashTable.Put(x, k)` (see Figure 7.6) which uses a compression map, as described in Section 7.3. Use the `HashFunction` ADT to represent both hash functions (i.e., both the compression map and the function to compute the index in the hash table). You may need to define an additional data structure in order to save results of the compression map for use in rehashing (you need to be able to retrieve these values quickly).

Exercise 7.8 Give an implementation of `HashFunction` that uses polynomial hashing, as described in Section 7.3. You may assume that the variable \( r \) contains an appropriate value to use as the radix, and that the variable \( w \) contains the number of bits in a machine word. The size should be the smallest power of 2 that is no smaller than the parameter given to the constructor. For clarity, your implementation should include the mod operation rather than relying on overflow (we don’t assume any explicit bounds on integer variables in our algorithms).

* Exercise 7.9 Prove that for each \( h \in H_{l,m}^1 \) there is exactly one \( s \in S_{l,m} \) such that \( h_s = h \). [Hint: Prove that if \( s \neq s' \), then \( h_s \neq h_{s'} \). In order to do this, it is sufficient to find a \( k \) such that \( h_s(k) \neq h_{s'}(k) \).]

* Exercise 7.10 Modify `UniversalHash1` (Figure 7.8) to handle varying-length keys of unbounded length. Use the expandable-array design pattern to store the randomly-generated indices. Show that when this implementation is used with `HashTable` (shown in Figures 7.5 and 7.6), the amortized expected running time of the `Dictionary` operations is in \( O(l) \), where \( l \) is the number of bits in the longest key in the table. You may assume that the actual running time of `Remove(k)` is proportional to the running time of `hash.Index(k)` plus the length of the `ConsList` at the resulting index.

* Exercise 7.11 Complete the proof of Theorem 7.4.

Exercise 7.12 Implement `HashFunction` to provide \( H_{p,m}^2 \). You may assume the variable \( p \) contains a prime number larger than any key. You may also assume that all values will fit into integer variables.

Exercise 7.13 Implement `HashFunction` to provide \( H_{p,l}^3 \). You may assume the variable \( p \) contains a prime number larger than \( w \) bits, where \( w \) is another variable. You may also assume that if \( a, b, \) and \( c \) are all natural numbers less than \( p \), then \( ab + c \) will fit in an integer variable; however, you
may not assume that arbitrarily many of these values added together will
fit.

* Exercise 7.14 Suppose we were to modify the definition of $H_{p,l}^3$ (7.2) so
that for each $a_i, 1 \leq a_i < p$. Show that for every $l \geq 2$ and prime number
$p$, the resulting family of hash functions is not universal. Specifically, show
that there are two distinct keys that collide with probability strictly greater
than $1/p$. [Hint: First consider $l = 2$, then generalize.]

Exercise 7.15 Implement HashFunction to provide $H_{p,l,m}^4$ using the same
assumptions as for Exercise 7.13.

Exercise 7.16 Let $p$ be a prime number and $l$ and $m$ be positive integers
such that $m < p$.

* a. Prove that for every $h \in H_{p,l,m}^4$ and every positive integer $a < p$, there
is exactly one choice of natural numbers $b, a_1, \ldots, a_l$ less than $p$ such
that

$$h(\langle k_1, \ldots, k_l \rangle) = \left( a \sum_{i=1}^{l} a_i k_i + b \right) \mod p \mod m$$

for every $l$-tuple $\langle k_1, \ldots, k_l \rangle$ of natural numbers less than $p$.

b. Consider the following two methods of randomly selecting a hash func-
tion $h$ of the form given by equation (7.3):

i. Select $a$ with uniform probability from the positive integers less
than $p$, and select $b, a_1, \ldots, a_l$ independently with uniform prob-
ability from the natural numbers less than $p$.

ii. Set $a$ to 1, and select $b, a_1, \ldots, a_l$ independently with uniform
probability from the natural numbers less than $p$.

Prove that for any $h \in H_{p,l,m}^4$, $h$ is chosen with the same probability
by both methods.

* Exercise 7.17 In terms of $c$ and $n$, find the value of $m \in \mathbb{R}^{\geq 0}$ that
minimizes

$$\frac{c n (n - 1)}{m} + n + m,$$

assuming $c \in \mathbb{R}^{> 0}, n \in \mathbb{N}$, and $n > 1$.

Exercise 7.18 Prove that the constructor for PerfectHash runs in $\Theta(n l)$
expected time if $H_{l,m}^1$ is used as the universal hash family, where $m$ is a power
of 2.
7.9 Chapter Notes

Virtual initialization was suggested by Aho, Hopcroft, and Ullman [2, Exercise 2.12].

The first description of hashing in the literature was by Dumey [31], who also introduced the division method. However, the concept appears to have been discovered a few years earlier at IBM by H. P. Luhn and independently by Gene M. Amdahl, Elaine M. Boehme, N. Rochester, and Arthur L. Samuel. Knuth [80] gives a detailed treatment of deterministic hashing.

Universal hashing was introduced by Carter and Wegman [20]. They presented the universal families $\mathcal{H}_{1,m}$ and $\mathcal{H}_{2,m}$. The notion of a $c$-universal family is closely related to the notion of an $\epsilon$-universal family defined by Cormen, et al. [25].

The perfect hashing strategy given in Section 7.6 is due to Fredman, Komlós, and Szemerédi [43].