This method of computing sets from bit vectors is sometimes applied to document retrieval. Consider the problem of picking from a collection of documents those few which contain selected keywords. For each keyword, the document retrieval system stores a bit vector with one bit for each document. If the user wants to know which documents contain a certain three keywords, the corresponding three bit vectors are AND'ed together. Those bit positions resulting in a value of 1 correspond to the desired documents. Alternatively, a bit vector can be stored for each document to indicate those keywords appearing in the document. Such an organization is called a **signature file**. The signatures can be manipulated to find documents with desired combinations of keywords.

9.4 Hashing

This section presents a completely different approach to searching arrays: by direct access based on key value. The process of finding a record using some computation to map its key value to a position in the array is called **hashing**. Most hashing schemes place records in the array in whatever order satisfies the needs of the address calculation, thus the records are not ordered by value or frequency. The function that maps key values to positions is called a **hash function** and will be denoted by **h**. The array that holds the records is called the **hash table** and will be denoted by **HT**. A position in the hash table is also known as a **slot**. The number of slots in hash table **HT** will be denoted by the variable M, with slots numbered from 0 to $M-1$. The goal for a hashing system is to arrange things such that, for any key value K and some hash function **h**, $i = \mathbf{h}(K)$ is a slot in the table such that $0 \leq \mathbf{h}(K) < M$, and we have the key of the record stored at $\mathbf{HT}[i]$ equal to K.

Hashing is not good for applications where multiple records with the same key value are permitted. Hashing is not a good method for answering range searches. In other words, we cannot easily find all records (if any) whose key values fall within a certain range. Nor can we easily find the record with the minimum or maximum key value, or visit the records in key order. Hashing is most appropriate for answering the question, "What record, if any, has key value K?" For applications where access involves only exact-match queries, hashing is usually the search method of choice because it is extremely efficient when implemented correctly. As you will see in this section, however, there are many approaches to hashing and it is easy to devise an inefficient implementation. Hashing is suitable for both in-memory and disk-based searching and is one of the two most widely used methods for organizing large databases stored on disk (the other is the B-tree, which is covered in Chapter 10).

As a simple (though unrealistic) example of hashing, consider storing n records each with a unique key value in the range 0 to $n-1$. In this simple case, a record

with key k can be stored in **HT**[k], and the hash function is simply $\mathbf{h}(k) = k$. To find the record with key value k, simply look in **HT**[k].

Typically, there are many more values in the key range than there are slots in the hash table. For a more realistic example, suppose that the key can take any value in the range 0 to 65,535 (i.e., the key is a two-byte unsigned integer), and that we expect to store approximately 1000 records at any given time. It is impractical in this situation to use a hash table with 65,536 slots, because most of the slots will be left empty. Instead, we must devise a hash function that allows us to store the records in a much smaller table. Because the possible key range is larger than the size of the table, at least some of the slots must be mapped to from multiple key values. Given a hash function **h** and two keys k_1 and k_2, if $\mathbf{h}(k_1) = \beta = \mathbf{h}(k_2)$ where β is a slot in the table, then we say that k_1 and k_2 have a **collision** at slot β under hash function **h**.

Finding a record with key value K in a database organized by hashing follows a two-step procedure:

1. Compute the table location $\mathbf{h}(K)$.
2. Starting with slot $\mathbf{h}(K)$, locate the record containing key K using (if necessary) a **collision resolution policy**.

9.4.1 Hash Functions

Hashing generally takes records whose key values come from a large range and stores those records in a table with a relatively small number of slots. Collisions occur when two records hash to the same slot in the table. If we are careful—or lucky—when selecting a hash function, then the actual number of collisions will be few. Unfortunately, even under the best of circumstances, collisions are nearly unavoidable.[1] For example, consider a classroom full of students. What is the probability that some pair of students shares the same birthday (i.e., the same day of the year, not necessarily the same year)? If there are 23 students, then the odds are about even that two will share a birthday. This is despite the fact that there are 365 days in which students can have birthdays (ignoring leap years), on most of which no student in the class has a birthday. With more students, the probability of a shared birthday increases. The mapping of students to days based on their

[1]The exception to this is **perfect hashing**. Perfect hashing is a system in which records are hashed such that there are no collisions. A hash function is selected for the specific set of records being hashed, which requires that the entire collection of records be available before selecting the hash function. Perfect hashing is efficient because it always finds the record that we are looking for exactly where the hash function computes it to be, so only one access is required. Selecting a perfect hash function can be expensive, but might be worthwhile when extremely efficient search performance is required. An example is searching for data on a read-only CD. Here the database will never change, the time for each access is expensive, and the database designer can build the hash table before issuing the CD.

birthday is similar to assigning records to slots in a table (of size 365) using the birthday as a hash function. Note that this observation tells us nothing about *which* students share a birthday, or on *which* days of the year shared birthdays fall.

To be practical, a database organized by hashing must store records in a hash table that is not so large that it wastes space. Typically, this means that the hash table will be around half full. Because collisions are extremely likely to occur under these conditions (by chance, any record inserted into a table that is half full will have a collision half of the time), does this mean that we need not worry about the ability of a hash function to avoid collisions? Absolutely not. The difference between a good hash function and a bad hash function makes a big difference in practice. Technically, any function that maps all possible key values to a slot in the hash table is a hash function. In the extreme case, even a function that maps all records to the same slot is a hash function, but it does nothing to help us find records during a search operation.

We would like to pick a hash function that stores the actual records in the collection such that each slot in the hash table has equal probability of being filled. Unfortunately, we normally have no control over the key values of the actual records, so how well any particular hash function does this depends on the distribution of the keys within the allowable key range. In some cases, incoming data are well distributed across their key range. For example, if the input is a set of random numbers selected uniformly from the key range, any hash function that assigns the key range so that each slot in the hash table receives an equal share of the range will likely also distribute the input records uniformly within the table. However, in many applications the incoming records are highly clustered or otherwise poorly distributed. When input records are not well distributed throughout the key range it can be difficult to devise a hash function that does a good job of distributing the records throughout the table, especially if the input distribution is not known in advance.

There are many reasons why data values might be poorly distributed.

1. Natural frequency distributions tend to follow a common pattern where a few of the entities occur frequently while most entities occur relatively rarely. For example, consider the populations of the 100 largest cities in the United States. If you plot these populations on a number line, most of them will be clustered toward the low side, with a few outliers on the high side. This is an example of a Zipf distribution (see Section 9.2). Viewed the other way, the home town for a given person is far more likely to be a particular large city than a particular small town.
2. Collected data are likely to be skewed in some way. Field samples might be rounded to, say, the nearest 5 (i.e., all numbers end in 5 or 0).
3. If the input is a collection of common English words, the beginning letter will be poorly distributed.

Note that in examples 2 and 3, either high- or low-order bits of the key are poorly distributed.

When designing hash functions, we are generally faced with one of two situations.

1. We know nothing about the distribution of the incoming keys. In this case, we wish to select a hash function that evenly distributes the key range across the hash table, while avoiding obvious opportunities for clustering such as hash functions that are sensitive to the high- or low-order bits of the key value.
2. We know something about the distribution of the incoming keys. In this case, we should use a distribution-dependent hash function that avoids assigning clusters of related key values to the same hash table slot. For example, if hashing English words, we should *not* hash on the value of the first character because this is likely to be unevenly distributed.

Below are several examples of hash functions that illustrate these points.

Example 9.5 Consider the following hash function used to hash integers to a table of sixteen slots:

```
int h(int x) {
  return x % 16;
}
```

The value returned by this hash function depends solely on the least significant four bits of the key. Because these bits are likely to be poorly distributed (as an example, a high percentage of the keys might be even numbers, which means that the low order bit is zero), the result will also be poorly distributed. This example shows that the size of the table M can have a big effect on the performance of a hash system because this value is typically used as the modulus to ensure that the hash function produces a number in the range 0 to $M - 1$.

Example 9.6 A good hash function for numerical values comes from the **mid-square** method. The mid-square method squares the key value, and then takes the middle r bits of the result, giving a value in the range 0 to $2^r - 1$. This works well because most or all bits of the key value contribute to the result. For example, consider records whose keys are 4-digit numbers in base 10. The goal is to hash these key values to a table of size 100 (i.e., a range of 0 to 99). This range is equivalent to two digits in base 10. That is, $r = 2$. If the input is the number 4567, squaring yields an 8-digit number, 20857489. The middle two digits of this result are 57. All digits

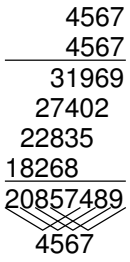

Figure 9.2 An illustration of the mid-square method, showing the details of long multiplication in the process of squaring the value 4567. The bottom of the figure indicates which digits of the answer are most influenced by each digit of the operands.

(equivalently, all bits when the number is viewed in binary) contribute to the middle two digits of the squared value. Figure 9.2 illustrates the concept. Thus, the result is not dominated by the distribution of the bottom digit or the top digit of the original key value.

Example 9.7 Here is a hash function for strings of characters:

```
int h(char* x) {
  int i, sum;
  for (sum=0, i=0; x[i] != '\0'; i++)
    sum += (int) x[i];
  return sum % M;
}
```

This function sums the ASCII values of the letters in a string. If the hash table size M is small, this hash function should do a good job of distributing strings evenly among the hash table slots, because it gives equal weight to all characters. This is an example of the **folding** approach to designing a hash function. Note that the order of the characters in the string has no effect on the result. A similar method for integers would add the digits of the key value, assuming that there are enough digits to (1) keep any one or two digits with bad distribution from skewing the results of the process and (2) generate a sum much larger than M. As with many other hash functions, the final step is to apply the modulus operator to the result, using table size M to generate a value within the table range. If the sum is not sufficiently large, then the modulus operator will yield a poor distribution. For example, because the ASCII value for "A" is 65 and "Z" is 90, `sum` will always be in the range 650 to 900 for a string of ten upper case letters. For a hash table of size 100 or less, a reasonable distribution results. For a hash table of size 1000, the distribution is terrible because only slots 650 to 900

can possibly be the home slot for some key value, and the values are not evenly distributed even within those slots.

Example 9.8 Here is a much better hash function for strings.

```
// Use folding on a string, summed 4 bytes at a time
int sfold(char* key) {
  unsigned int *lkey = (unsigned int *)key;
  int intlength = strlen(key)/4;
  unsigned int sum = 0;
  for(int i=0; i<intlength; i++)
    sum += lkey[i];

  // Now deal with the extra chars at the end
  int extra = strlen(key) - intlength*4;
  char temp[4];
  lkey = (unsigned int *)temp;
  lkey[0] = 0;
  for(int i=0; i<extra; i++)
    temp[i] = key[intlength*4+i];
  sum += lkey[0];

  return sum % M;
}
```

This function takes a string as input. It processes the string four bytes at a time, and interprets each of the four-byte chunks as a single (unsigned) long integer value. The integer values for the four-byte chunks are added together. In the end, the resulting sum is converted to the range 0 to $M - 1$ using the modulus operator.[2]

For example, if the string "aaaabbbb" is passed to **sfold**, then the first four bytes ("aaaa") will be interpreted as the integer value 1,633,771,873 and the next four bytes ("bbbb") will be interpreted as the integer value 1,650,614,882. Their sum is 3,284,386,755 (when viewed as an unsigned integer). If the table size is 101 then the modulus function will cause this key to hash to slot 75 in the table. Note that for any sufficiently long string, the sum for the integer quantities will typically cause a 32-bit integer to overflow (thus losing some of the high-order bits) because the resulting values are so large. But this causes no problems when the goal is to compute a hash function.

[2]Recall from Section 2.2 that the implementation for $n \bmod m$ on many **C**++ and Java compilers will yield a negative number if n is negative. Implementors for hash functions need to be careful that their hash function does not generate a negative number. This can be avoided either by insuring that n is positive when computing $n \bmod m$, or adding m to the result if $n \bmod m$ is negative. All computation in **sfold** is done using unsigned long values in part to protect against taking the modulus of an negative number.

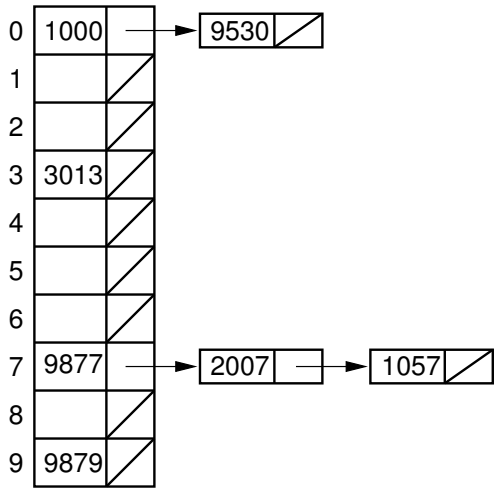

Figure 9.3 An illustration of open hashing for seven numbers stored in a ten-slot hash table using the hash function $\mathbf{h}(K) = K \bmod 10$. The numbers are inserted in the order 9877, 2007, 1000, 9530, 3013, 9879, and 1057. Two of the values hash to slot 0, one value hashes to slot 2, three of the values hash to slot 7, and one value hashes to slot 9.

9.4.2 Open Hashing

While the goal of a hash function is to minimize collisions, some collisions are unavoidable in practice. Thus, hashing implementations must include some form of collision resolution policy. Collision resolution techniques can be broken into two classes: **open hashing** (also called **separate chaining**) and **closed hashing** (also called **open addressing**).[3] The difference between the two has to do with whether collisions are stored outside the table (open hashing), or whether collisions result in storing one of the records at another slot in the table (closed hashing). Open hashing is treated in this section, and closed hashing in Section 9.4.3.

The simplest form of open hashing defines each slot in the hash table to be the head of a linked list. All records that hash to a particular slot are placed on that slot's linked list. Figure 9.3 illustrates a hash table where each slot stores one record and a link pointer to the rest of the list.

Records within a slot's list can be ordered in several ways: by insertion order, by key value order, or by frequency-of-access order. Ordering the list by key value provides an advantage in the case of an unsuccessful search, because we know to stop searching the list once we encounter a key that is greater than the one being

[3] Yes, it is confusing when "open hashing" means the opposite of "open addressing," but unfortunately, that is the way it is.

searched for. If records on the list are unordered or ordered by frequency, then an unsuccessful search will need to visit every record on the list.

Given a table of size M storing N records, the hash function will (ideally) spread the records evenly among the M positions in the table, yielding on average N/M records for each list. Assuming that the table has more slots than there are records to be stored, we can hope that few slots will contain more than one record. In the case where a list is empty or has only one record, a search requires only one access to the list. Thus, the average cost for hashing should be $\Theta(1)$. However, if clustering causes many records to hash to only a few of the slots, then the cost to access a record will be much higher because many elements on the linked list must be searched.

Open hashing is most appropriate when the hash table is kept in main memory, with the lists implemented by a standard in-memory linked list. Storing an open hash table on disk in an efficient way is difficult, because members of a given linked list might be stored on different disk blocks. This would result in multiple disk accesses when searching for a particular key value, which defeats the purpose of using hashing.

There are similarities between open hashing and Binsort. One way to view open hashing is that each record is simply placed in a bin. While multiple records may hash to the same bin, this initial binning should still greatly reduce the number of records accessed by a search operation. In a similar fashion, a simple Binsort reduces the number of records in each bin to a small number that can be sorted in some other way.

9.4.3 Closed Hashing

Closed hashing stores all records directly in the hash table. Each record *R* with key value k_R has a **home position** that is $\mathbf{h}(k_R)$, the slot computed by the hash function. If *R* is to be inserted and another record already occupies *R*'s home position, then *R* will be stored at some other slot in the table. It is the business of the collision resolution policy to determine which slot that will be. Naturally, the same policy must be followed during search as during insertion, so that any record not found in its home position can be recovered by repeating the collision resolution process.

Bucket Hashing

One implementation for closed hashing groups hash table slots into **buckets**. The M slots of the hash table are divided into B buckets, with each bucket consisting of M/B slots. The hash function assigns each record to the first slot within one of the buckets. If this slot is already occupied, then the bucket slots are searched sequentially until an open slot is found. If a bucket is entirely full, then the record is stored in an **overflow bucket** of infinite capacity at the end of the table. All

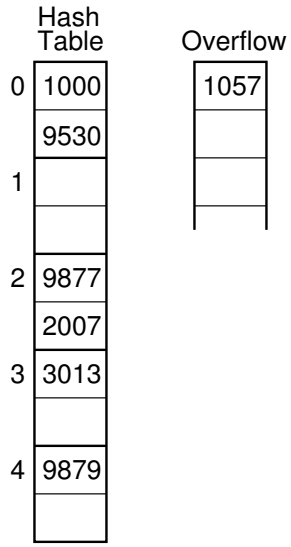

Figure 9.4 An illustration of bucket hashing for seven numbers stored in a five-bucket hash table using the hash function $\mathbf{h}(K) = K \bmod 5$. Each bucket contains two slots. The numbers are inserted in the order 9877, 2007, 1000, 9530, 3013, 9879, and 1057. Two of the values hash to bucket 0, three values hash to bucket 2, one value hashes to bucket 3, and one value hashes to bucket 4. Because bucket 2 cannot hold three values, the third one ends up in the overflow bucket.

buckets share the same overflow bucket. A good implementation will use a hash function that distributes the records evenly among the buckets so that as few records as possible go into the overflow bucket. Figure 9.4 illustrates bucket hashing.

When searching for a record, the first step is to hash the key to determine which bucket should contain the record. The records in this bucket are then searched. If the desired key value is not found and the bucket still has free slots, then the search is complete. If the bucket is full, then it is possible that the desired record is stored in the overflow bucket. In this case, the overflow bucket must be searched until the record is found or all records in the overflow bucket have been checked. If many records are in the overflow bucket, this will be an expensive process.

A simple variation on bucket hashing is to hash a key value to some slot in the hash table as though bucketing were not being used. If the home position is full, then the collision resolution process is to move down through the table toward the end of the bucket while searching for a free slot in which to store the record. If the bottom of the bucket is reached, then the collision resolution routine wraps around to the top of the bucket to continue the search for an open slot. For example, assume that buckets contain eight records, with the first bucket consisting of slots 0 through 7. If a record is hashed to slot 5, the collision resolution process will attempt to insert the record into the table in the order 5, 6, 7, 0, 1, 2, 3, and finally 4. If all slots in this bucket are full, then the record is assigned to the overflow bucket.

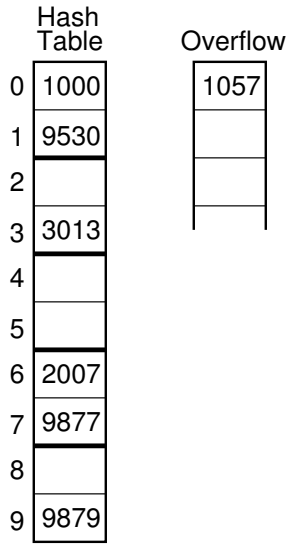

Figure 9.5 An variant of bucket hashing for seven numbers stored in a 10-slot hash table using the hash function $\mathbf{h}(K) = K \bmod 10$. Each bucket contains two slots. The numbers are inserted in the order 9877, 2007, 1000, 9530, 3013, 9879, and 1057. Value 9877 first hashes to slot 7, so when value 2007 attempts to do likewise, it is placed in the other slot associated with that bucket which is slot 6. When value 1057 is inserted, there is no longer room in the bucket and it is placed into overflow. The other collision occurs after value 1000 is inserted to slot 0, causing 9530 to be moved to slot 1.

The advantage of this approach is that initial collisions are reduced, Because any slot can be a home position rather than just the first slot in the bucket. Figure 9.5 shows another example for this form of bucket hashing.

Bucket methods are good for implementing hash tables stored on disk, because the bucket size can be set to the size of a disk block. Whenever search or insertion occurs, the entire bucket is read into memory. Because the entire bucket is then in memory, processing an insert or search operation requires only one disk access, unless the bucket is full. If the bucket is full, then the overflow bucket must be retrieved from disk as well. Naturally, overflow should be kept small to minimize unnecessary disk accesses.

Linear Probing

We now turn to the most commonly used form of hashing: closed hashing with no bucketing, and a collision resolution policy that can potentially use any slot in the hash table.

During insertion, the goal of collision resolution is to find a free slot in the hash table when the home position for the record is already occupied. We can view any collision resolution method as generating a sequence of hash table slots that can

```
// Insert e into hash table HT
template <typename Key, typename E>
void hashdict<Key, E>::
hashInsert(const Key& k, const E& e) {
  int home;                       // Home position for e
  int pos = home = h(k);          // Init probe sequence
  for (int i=1; EMPTYKEY != (HT[pos]).key(); i++) {
    pos = (home + p(k, i)) % M; // probe
    Assert(k != (HT[pos]).key(), "Duplicates not allowed");
  }
  KVpair<Key,E> temp(k, e);
  HT[pos] = temp;
}
```

Figure 9.6 Insertion method for a dictionary implemented by a hash table.

potentially hold the record. The first slot in the sequence will be the home position for the key. If the home position is occupied, then the collision resolution policy goes to the next slot in the sequence. If this is occupied as well, then another slot must be found, and so on. This sequence of slots is known as the **probe sequence**, and it is generated by some **probe function** that we will call **p**. The insert function is shown in Figure 9.6.

Method **hashInsert** first checks to see if the home slot for the key is empty. If the home slot is occupied, then we use the probe function, **p**(k, i) to locate a free slot in the table. Function **p** has two parameters, the key k and a count i for where in the probe sequence we wish to be. That is, to get the first position in the probe sequence after the home slot for key K, we call **p**(K, 1). For the next slot in the probe sequence, call **p**(K, 2). Note that the probe function returns an offset from the original home position, rather than a slot in the hash table. Thus, the **for** loop in **hashInsert** is computing positions in the table at each iteration by adding the value returned from the probe function to the home position. The ith call to **p** returns the ith offset to be used.

Searching in a hash table follows the same probe sequence that was followed when inserting records. In this way, a record not in its home position can be recovered. A C++ implementation for the search procedure is shown in Figure 9.7.

The insert and search routines assume that at least one slot on the probe sequence of every key will be empty. Otherwise, they will continue in an infinite loop on unsuccessful searches. Thus, the dictionary should keep a count of the number of records stored, and refuse to insert into a table that has only one free slot.

The discussion on bucket hashing presented a simple method of collision resolution. If the home position for the record is occupied, then move down the bucket until a free slot is found. This is an example of a technique for collision resolution known as **linear probing**. The probe function for simple linear probing is

$$\mathbf{p}(K, i) = i.$$

```
// Search for the record with Key K
template <typename Key, typename E>
E hashdict<Key, E>::
hashSearch(const Key& k) const {
  int home;                // Home position for k
  int pos = home = h(k); // Initial position is home slot
  for (int i = 1; (k != (HT[pos]).key()) &&
                  (EMPTYKEY != (HT[pos]).key()); i++)
    pos = (home + p(k, i)) % M; // Next on probe sequence
  if (k == (HT[pos]).key())     // Found it
    return (HT[pos]).value();
  else return NULL;            // k not in hash table
}
```

Figure 9.7 Search method for a dictionary implemented by a hash table.

That is, the ith offset on the probe sequence is just i, meaning that the ith step is simply to move down i slots in the table.

Once the bottom of the table is reached, the probe sequence wraps around to the beginning of the table. Linear probing has the virtue that all slots in the table will be candidates for inserting a new record before the probe sequence returns to the home position.

While linear probing is probably the first idea that comes to mind when considering collision resolution policies, it is not the only one possible. Probe function `p` allows us many options for how to do collision resolution. In fact, linear probing is one of the worst collision resolution methods. The main problem is illustrated by Figure 9.8. Here, we see a hash table of ten slots used to store four-digit numbers, with hash function $\mathbf{h}(K) = K \bmod 10$. In Figure 9.8(a), five numbers have been placed in the table, leaving five slots remaining.

The ideal behavior for a collision resolution mechanism is that each empty slot in the table will have equal probability of receiving the next record inserted (assuming that every slot in the table has equal probability of being hashed to initially). In this example, assume that the hash function gives each slot (roughly) equal probability of being the home position for the next key. However, consider what happens to the next record if its key has its home position at slot 0. Linear probing will send the record to slot 2. The same will happen to records whose home position is at slot 1. A record with home position at slot 2 will remain in slot 2. Thus, the probability is 3/10 that the next record inserted will end up in slot 2. In a similar manner, records hashing to slots 7 or 8 will end up in slot 9. However, only records hashing to slot 3 will be stored in slot 3, yielding one chance in ten of this happening. Likewise, there is only one chance in ten that the next record will be stored in slot 4, one chance in ten for slot 5, and one chance in ten for slot 6. Thus, the resulting probabilities are not equal.

To make matters worse, if the next record ends up in slot 9 (which already has a higher than normal chance of happening), then the following record will end up

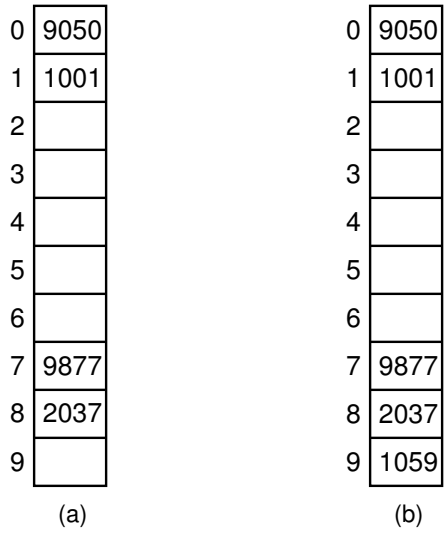

Figure 9.8 Example of problems with linear probing. (a) Four values are inserted in the order 1001, 9050, 9877, and 2037 using hash function $\mathbf{h}(K) = K \bmod 10$. (b) The value 1059 is added to the hash table.

in slot 2 with probability 6/10. This is illustrated by Figure 9.8(b). This tendency of linear probing to cluster items together is known as **primary clustering**. Small clusters tend to merge into big clusters, making the problem worse. The objection to primary clustering is that it leads to long probe sequences.

Improved Collision Resolution Methods

How can we avoid primary clustering? One possible improvement might be to use linear probing, but to skip slots by a constant c other than 1. This would make the probe function

$$\mathbf{p}(K, i) = ci,$$

and so the ith slot in the probe sequence will be $(\mathbf{h}(K) + ic) \bmod M$. In this way, records with adjacent home positions will not follow the same probe sequence. For example, if we were to skip by twos, then our offsets from the home slot would be 2, then 4, then 6, and so on.

One quality of a good probe sequence is that it will cycle through all slots in the hash table before returning to the home position. Clearly linear probing (which "skips" slots by one each time) does this. Unfortunately, not all values for c will make this happen. For example, if $c = 2$ and the table contains an even number of slots, then any key whose home position is in an even slot will have a probe sequence that cycles through only the even slots. Likewise, the probe sequence for a key whose home position is in an odd slot will cycle through the odd slots. Thus, this combination of table size and linear probing constant effectively divides

the records into two sets stored in two disjoint sections of the hash table. So long as both sections of the table contain the same number of records, this is not really important. However, just from chance it is likely that one section will become fuller than the other, leading to more collisions and poorer performance for those records. The other section would have fewer records, and thus better performance. But the overall system performance will be degraded, as the additional cost to the side that is more full outweighs the improved performance of the less-full side.

Constant c must be relatively prime to M to generate a linear probing sequence that visits all slots in the table (that is, c and M must share no factors). For a hash table of size $M = 10$, if c is any one of 1, 3, 7, or 9, then the probe sequence will visit all slots for any key. When $M = 11$, any value for c between 1 and 10 generates a probe sequence that visits all slots for every key.

Consider the situation where $c = 2$ and we wish to insert a record with key k_1 such that $\mathbf{h}(k_1) = 3$. The probe sequence for k_1 is 3, 5, 7, 9, and so on. If another key k_2 has home position at slot 5, then its probe sequence will be 5, 7, 9, and so on. The probe sequences of k_1 and k_2 are linked together in a manner that contributes to clustering. In other words, linear probing with a value of $c > 1$ does not solve the problem of primary clustering. We would like to find a probe function that does not link keys together in this way. We would prefer that the probe sequence for k_1 after the first step on the sequence should not be identical to the probe sequence of k_2. Instead, their probe sequences should diverge.

The ideal probe function would select the next position on the probe sequence at random from among the unvisited slots; that is, the probe sequence should be a random permutation of the hash table positions. Unfortunately, we cannot actually select the next position in the probe sequence at random, because then we would not be able to duplicate this same probe sequence when searching for the key. However, we can do something similar called **pseudo-random probing**. In pseudo-random probing, the ith slot in the probe sequence is $(\mathbf{h}(K) + r_i) \bmod M$ where r_i is the ith value in a random permutation of the numbers from 1 to $M - 1$. All insertion and search operations use the same random permutation. The probe function is

$$\mathbf{p}(K, i) = \texttt{Perm}[i - 1],$$

where **`Perm`** is an array of length $M - 1$ containing a random permutation of the values from 1 to $M - 1$.

Example 9.9 Consider a table of size $M = 101$, with $\texttt{Perm}[1] = 5$, $\texttt{Perm}[2] = 2$, and $\texttt{Perm}[3] = 32$. Assume that we have two keys k_1 and k_2 where $\mathbf{h}(k_1) = 30$ and $\mathbf{h}(k_2)$ = 35. The probe sequence for k_1 is 30, then 35, then 32, then 62. The probe sequence for k_2 is 35, then 40, then 37, then 67. Thus, while k_2 will probe to k_1's home position as its second choice, the two keys' probe sequences diverge immediately thereafter.

Another probe function that eliminates primary clustering is called **quadratic probing**. Here the probe function is some quadratic function

$$\mathbf{p}(K, i) = c_1 i^2 + c_2 i + c_3$$

for some choice of constants c_1, c_2, and c_3. The simplest variation is $\mathbf{p}(K, i) = i^2$ (i.e., $c_1 = 1$, $c_2 = 0$, and $c_3 = 0$. Then the ith value in the probe sequence would be $(\mathbf{h}(K) + i^2) \bmod M$. Under quadratic probing, two keys with different home positions will have diverging probe sequences.

Example 9.10 Given a hash table of size $M = 101$, assume for keys k_1 and k_2 that $\mathbf{h}(k_1) = 30$ and $\mathbf{h}(k_2) = 29$. The probe sequence for k_1 is 30, then 31, then 34, then 39. The probe sequence for k_2 is 29, then 30, then 33, then 38. Thus, while k_2 will probe to k_1's home position as its second choice, the two keys' probe sequences diverge immediately thereafter.

Unfortunately, quadratic probing has the disadvantage that typically not all hash table slots will be on the probe sequence. Using $\mathbf{p}(K, i) = i^2$ gives particularly inconsistent results. For many hash table sizes, this probe function will cycle through a relatively small number of slots. If all slots on that cycle happen to be full, then the record cannot be inserted at all! For example, if our hash table has three slots, then records that hash to slot 0 can probe only to slots 0 and 1 (that is, the probe sequence will never visit slot 2 in the table). Thus, if slots 0 and 1 are full, then the record cannot be inserted even though the table is not full. A more realistic example is a table with 105 slots. The probe sequence starting from any given slot will only visit 23 other slots in the table. If all 24 of these slots should happen to be full, even if other slots in the table are empty, then the record cannot be inserted because the probe sequence will continually hit only those same 24 slots.

Fortunately, it is possible to get good results from quadratic probing at low cost. The right combination of probe function and table size will visit many slots in the table. In particular, if the hash table size is a prime number and the probe function is $\mathbf{p}(K, i) = i^2$, then at least half the slots in the table will be visited. Thus, if the table is less than half full, we can be certain that a free slot will be found. Alternatively, if the hash table size is a power of two and the probe function is $\mathbf{p}(K, i) = (i^2 + i)/2$, then every slot in the table will be visited by the probe function.

Both pseudo-random probing and quadratic probing eliminate primary clustering, which is the problem of keys sharing substantial segments of a probe sequence. If two keys hash to the same home position, however, then they will always follow the same probe sequence for every collision resolution method that we have seen so far. The probe sequences generated by pseudo-random and quadratic probing (for example) are entirely a function of the home position, not the original key value.

This is because function $\mathbf{p}$ ignores its input parameter K for these collision resolution methods. If the hash function generates a cluster at a particular home position, then the cluster remains under pseudo-random and quadratic probing. This problem is called **secondary clustering**.

To avoid secondary clustering, we need to have the probe sequence make use of the original key value in its decision-making process. A simple technique for doing this is to return to linear probing by a constant step size for the probe function, but to have that constant be determined by a second hash function, $\mathbf{h_2}$. Thus, the probe sequence would be of the form $\mathbf{p}(K, i) = i * \mathbf{h}_2(K)$. This method is called **double hashing**.

Example 9.11 Assume a hash table has size $M = 101$, and that there are three keys k_1, k_2, and k_3 with $\mathbf{h}(k_1) = 30$, $\mathbf{h}(k_2) = 28$, $\mathbf{h}(k_3) = 30$, $\mathbf{h}_2(k_1) = 2$, $\mathbf{h}_2(k_2) = 5$, and $\mathbf{h}_2(k_3) = 5$. Then, the probe sequence for k_1 will be 30, 32, 34, 36, and so on. The probe sequence for k_2 will be 28, 33, 38, 43, and so on. The probe sequence for k_3 will be 30, 35, 40, 45, and so on. Thus, none of the keys share substantial portions of the same probe sequence. Of course, if a fourth key k_4 has $\mathbf{h}(k_4) = 28$ and $\mathbf{h}_2(k_4) = 2$, then it will follow the same probe sequence as k_1. Pseudo-random or quadratic probing can be combined with double hashing to solve this problem.

A good implementation of double hashing should ensure that all of the probe sequence constants are relatively prime to the table size M. This can be achieved easily. One way is to select M to be a prime number, and have $\mathbf{h}_2$ return a value in the range $1 \leq \mathbf{h}_2(K) \leq M - 1$. Another way is to set $M = 2^m$ for some value m and have $\mathbf{h}_2$ return an odd value between 1 and 2^m.

Figure 9.9 shows an implementation of the dictionary ADT by means of a hash table. The simplest hash function is used, with collision resolution by linear probing, as the basis for the structure of a hash table implementation. A suggested project at the end of this chapter asks you to improve the implementation with other hash functions and collision resolution policies.

9.4.4 Analysis of Closed Hashing

How efficient is hashing? We can measure hashing performance in terms of the number of record accesses required when performing an operation. The primary operations of concern are insertion, deletion, and search. It is useful to distinguish between successful and unsuccessful searches. Before a record can be deleted, it must be found. Thus, the number of accesses required to delete a record is equivalent to the number required to successfully search for it. To insert a record, an empty slot along the record's probe sequence must be found. This is equivalent to

```
// Dictionary implemented with a hash table
template <typename Key, typename E>
class hashdict : public Dictionary<Key,E> {
private:
  KVpair<Key,E>* HT;   // The hash table
  int M;         // Size of HT
  int currcnt;  // The current number of elements in HT
  Key EMPTYKEY; // User-supplied key value for an empty slot

  int p(Key K, int i) const // Probe using linear probing
    { return i; }

  int h(int x) const { return x % M; } // Poor hash function
  int h(char* x) const { // Hash function for character keys
    int i, sum;
    for (sum=0, i=0; x[i] != '\0'; i++) sum += (int) x[i];
    return sum % M;
  }

  void hashInsert(const Key&, const E&);
  E hashSearch(const Key&) const;

public:
  hashdict(int sz, Key k){ // "k" defines an empty slot
    M = sz;
    EMPTYKEY = k;
    currcnt = 0;
    HT = new KVpair<Key,E>[sz]; // Make HT of size sz
    for (int i=0; i<M; i++)
      (HT[i]).setKey(EMPTYKEY); // Initialize HT
  }

  ~hashdict() { delete HT; }

  // Find some record with key value "K"
  E find(const Key& k) const
    { return hashSearch(k); }
  int size() { return currcnt; } // Number stored in table

  // Insert element "it" with Key "k" into the dictionary.
  void insert(const Key& k, const E& it) {
    Assert(currcnt < M, "Hash table is full");
    hashInsert(k, it);
    currcnt++;
  }
```

Figure 9.9 A partial implementation for the dictionary ADT using a hash table. This uses a poor hash function and a poor collision resolution policy (linear probing), which can easily be replaced. Member functions **hashInsert** and **hashSearch** appear in Figures 9.6 and 9.7, respectively.

an unsuccessful search for the record (recall that a successful search for the record during insertion should generate an error because two records with the same key are not allowed to be stored in the table).

When the hash table is empty, the first record inserted will always find its home position free. Thus, it will require only one record access to find a free slot. If all records are stored in their home positions, then successful searches will also require only one record access. As the table begins to fill up, the probability that a record can be inserted into its home position decreases. If a record hashes to an occupied slot, then the collision resolution policy must locate another slot in which to store it. Finding records not stored in their home position also requires additional record accesses as the record is searched for along its probe sequence. As the table fills up, more and more records are likely to be located ever further from their home positions.

From this discussion, we see that the expected cost of hashing is a function of how full the table is. Define the **load factor** for the table as $\alpha = N/M$, where N is the number of records currently in the table.

An estimate of the expected cost for an insertion (or an unsuccessful search) can be derived analytically as a function of α in the case where we assume that the probe sequence follows a random permutation of the slots in the hash table. Assuming that every slot in the table has equal probability of being the home slot for the next record, the probability of finding the home position occupied is α. The probability of finding both the home position occupied and the next slot on the probe sequence occupied is $\frac{N(N-1)}{M(M-1)}$. The probability of i collisions is

$$\frac{N(N-1)\cdots(N-i+1)}{M(M-1)\cdots(M-i+1)}.$$

If N and M are large, then this is approximately $(N/M)^i$. The expected number of probes is one plus the sum over $i \geq 1$ of the probability of i collisions, which is approximately

$$1 + \sum_{i=1}^{\infty} (N/M)^i = 1/(1-\alpha).$$

The cost for a successful search (or a deletion) has the same cost as originally inserting that record. However, the expected value for the insertion cost depends on the value of α not at the time of deletion, but rather at the time of the original insertion. We can derive an estimate of this cost (essentially an average over all the insertion costs) by integrating from 0 to the current value of α, yielding a result of

$$\frac{1}{\alpha}\int_0^{\alpha} \frac{1}{1-x}dx = \frac{1}{\alpha}\log_e \frac{1}{1-\alpha}.$$

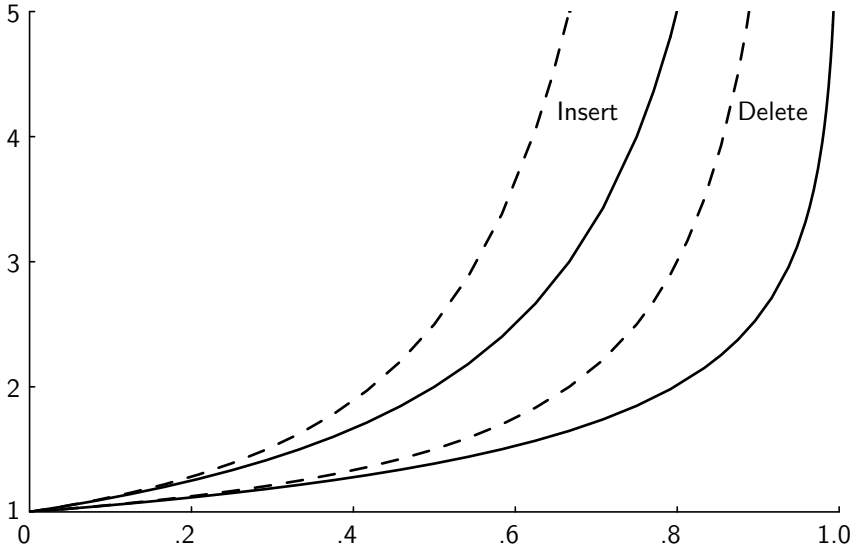

Figure 9.10 Growth of expected record accesses with α. The horizontal axis is the value for α, the vertical axis is the expected number of accesses to the hash table. Solid lines show the cost for "random" probing (a theoretical lower bound on the cost), while dashed lines show the cost for linear probing (a relatively poor collision resolution strategy). The two leftmost lines show the cost for insertion (equivalently, unsuccessful search); the two rightmost lines show the cost for deletion (equivalently, successful search).

It is important to realize that these equations represent the expected cost for operations using the unrealistic assumption that the probe sequence is based on a random permutation of the slots in the hash table (thus avoiding all expense resulting from clustering). Thus, these costs are lower-bound estimates in the average case. The true average cost under linear probing is $\frac{1}{2}(1+1/(1-\alpha)^2)$ for insertions or unsuccessful searches and $\frac{1}{2}(1+1/(1-\alpha))$ for deletions or successful searches. Proofs for these results can be found in the references cited in Section 9.5.

Figure 9.10 shows the graphs of these four equations to help you visualize the expected performance of hashing based on the load factor. The two solid lines show the costs in the case of a "random" probe sequence for (1) insertion or unsuccessful search and (2) deletion or successful search. As expected, the cost for insertion or unsuccessful search grows faster, because these operations typically search further down the probe sequence. The two dashed lines show equivalent costs for linear probing. As expected, the cost of linear probing grows faster than the cost for "random" probing.

From Figure 9.10 we see that the cost for hashing when the table is not too full is typically close to one record access. This is extraordinarily efficient, much better than binary search which requires $\log n$ record accesses. As α increases, so does

the expected cost. For small values of α, the expected cost is low. It remains below two until the hash table is about half full. When the table is nearly empty, adding a new record to the table does not increase the cost of future search operations by much. However, the additional search cost caused by each additional insertion increases rapidly once the table becomes half full. Based on this analysis, the rule of thumb is to design a hashing system so that the hash table never gets above half full. Beyond that point performance will degrade rapidly. This requires that the implementor have some idea of how many records are likely to be in the table at maximum loading, and select the table size accordingly.

You might notice that a recommendation to never let a hash table become more than half full contradicts the disk-based space/time tradeoff principle, which strives to minimize disk space to increase information density. Hashing represents an unusual situation in that there is no benefit to be expected from locality of reference. In a sense, the hashing system implementor does everything possible to eliminate the effects of locality of reference! Given the disk block containing the last record accessed, the chance of the next record access coming to the same disk block is no better than random chance in a well-designed hash system. This is because a good hashing implementation breaks up relationships between search keys. Instead of improving performance by taking advantage of locality of reference, hashing trades increased hash table space for an improved chance that the record will be in its home position. Thus, the more space available for the hash table, the more efficient hashing should be.

Depending on the pattern of record accesses, it might be possible to reduce the expected cost of access even in the face of collisions. Recall the 80/20 rule: 80% of the accesses will come to 20% of the data. In other words, some records are accessed more frequently. If two records hash to the same home position, which would be better placed in the home position, and which in a slot further down the probe sequence? The answer is that the record with higher frequency of access should be placed in the home position, because this will reduce the total number of record accesses. Ideally, records along a probe sequence will be ordered by their frequency of access.

One approach to approximating this goal is to modify the order of records along the probe sequence whenever a record is accessed. If a search is made to a record that is not in its home position, a self-organizing list heuristic can be used. For example, if the linear probing collision resolution policy is used, then whenever a record is located that is not in its home position, it can be swapped with the record preceding it in the probe sequence. That other record will now be further from its home position, but hopefully it will be accessed less frequently. Note that this approach will not work for the other collision resolution policies presented in this section, because swapping a pair of records to improve access to one might remove the other from its probe sequence.

Another approach is to keep access counts for records and periodically rehash the entire table. The records should be inserted into the hash table in frequency order, ensuring that records that were frequently accessed during the last series of requests have the best chance of being near their home positions.

9.4.5 Deletion

When deleting records from a hash table, there are two important considerations.

1. Deleting a record must not hinder later searches. In other words, the search process must still pass through the newly emptied slot to reach records whose probe sequence passed through this slot. Thus, the delete process cannot simply mark the slot as empty, because this will isolate records further down the probe sequence. For example, in Figure 9.8(a), keys 9877 and 2037 both hash to slot 7. Key 2037 is placed in slot 8 by the collision resolution policy. If 9877 is deleted from the table, a search for 2037 must still pass through Slot 7 as it probes to slot 8.
2. We do not want to make positions in the hash table unusable because of deletion. The freed slot should be available to a future insertion.

Both of these problems can be resolved by placing a special mark in place of the deleted record, called a **tombstone**. The tombstone indicates that a record once occupied the slot but does so no longer. If a tombstone is encountered when searching along a probe sequence, the search procedure continues with the search. When a tombstone is encountered during insertion, that slot can be used to store the new record. However, to avoid inserting duplicate keys, it will still be necessary for the search procedure to follow the probe sequence until a truly empty position has been found, simply to verify that a duplicate is not in the table. However, the new record would actually be inserted into the slot of the first tombstone encountered.

The use of tombstones allows searches to work correctly and allows reuse of deleted slots. However, after a series of intermixed insertion and deletion operations, some slots will contain tombstones. This will tend to lengthen the average distance from a record's home position to the record itself, beyond where it could be if the tombstones did not exist. A typical database application will first load a collection of records into the hash table and then progress to a phase of intermixed insertions and deletions. After the table is loaded with the initial collection of records, the first few deletions will lengthen the average probe sequence distance for records (it will add tombstones). Over time, the average distance will reach an equilibrium point because insertions will tend to decrease the average distance by filling in tombstone slots. For example, after initially loading records into the database, the average path distance might be 1.2 (i.e., an average of 0.2 accesses per search beyond the home position will be required). After a series of insertions and deletions, this average distance might increase to 1.6 due to tombstones. This

seems like a small increase, but it is three times longer on average beyond the home position than before deletions.

Two possible solutions to this problem are

1. Do a local reorganization upon deletion to try to shorten the average path length. For example, after deleting a key, continue to follow the probe sequence of that key and swap records further down the probe sequence into the slot of the recently deleted record (being careful not to remove any key from its probe sequence). This will not work for all collision resolution policies.
2. Periodically rehash the table by reinserting all records into a new hash table. Not only will this remove the tombstones, but it also provides an opportunity to place the most frequently accessed records into their home positions.

9.5 Further Reading

For a comparison of the efficiencies for various self-organizing techniques, see Bentley and McGeoch, "Amortized Analysis of Self-Organizing Sequential Search Heuristics" [BM85]. The text compression example of Section 9.2 comes from Bentley et al., "A Locally Adaptive Data Compression Scheme" [BSTW86]. For more on Ziv-Lempel coding, see *Data Compression: Methods and Theory* by James A. Storer [Sto88]. Knuth covers self-organizing lists and Zipf distributions in Volume 3 of *The Art of Computer Programming*[Knu98].

Introduction to Modern Information Retrieval by Salton and McGill [SM83] is an excellent source for more information about document retrieval techniques.

See the paper "Practical Minimal Perfect Hash Functions for Large Databases" by Fox et al. [FHCD92] for an introduction and a good algorithm for perfect hashing.

For further details on the analysis for various collision resolution policies, see Knuth, Volume 3 [Knu98] and *Concrete Mathematics: A Foundation for Computer Science* by Graham, Knuth, and Patashnik [GKP94].

The model of hashing presented in this chapter has been of a fixed-size hash table. A problem not addressed is what to do when the hash table gets half full and more records must be inserted. This is the domain of dynamic hashing methods. A good introduction to this topic is "Dynamic Hashing Schemes" by R.J. Enbody and H.C. Du [ED88].

9.6 Exercises

9.1 Create a graph showing expected cost versus the probability of an unsuccessful search when performing sequential search (see Section 9.1). What