9

Searching

Organizing and retrieving information is at the heart of most computer applications, and searching is surely the most frequently performed of all computing tasks. Search can be viewed abstractly as a process to determine if an element with a particular value is a member of a particular set. The more common view of searching is an attempt to find the record within a collection of records that has a particular key value, or those records in a collection whose key values meet some criterion such as falling within a range of values.

We can define searching formally as follows. Suppose k_1, k_2, ... k_n are distinct keys, and that we have a collection **L** of n records of the form

$$(k_1, I_1), (k_2, I_2), ..., (k_n, I_n)$$

where I_j is information associated with key k_j for $1 \leq j \leq n$. Given a particular key value *K*, the **search problem** is to locate the record (k_j, I_j) in **L** such that $k_j = K$ (if one exists). **Searching** is a systematic method for locating the record (or records) with key value $k_j = K$.

A **successful** search is one in which a record with key $k_j = K$ is found. An **unsuccessful** search is one in which no record with $k_j = K$ is found (and no such record exists).

An **exact-match query** is a search for the record whose key value matches a specified key value. A **range query** is a search for all records whose key value falls within a specified range of key values.

We can categorize search algorithms into three general approaches:

1. Sequential and list methods.
2. Direct access by key value (hashing).
3. Tree indexing methods.

This and the following chapter treat these three approaches in turn. Any of these approaches are potentially suitable for implementing the Dictionary ADT introduced in Section 4.2. However, each has different performance characteristics that make it the method of choice in particular circumstances.

The current chapter considers methods for searching data stored in lists and tables. A **table** is simply another term for an array. List in this context means any list implementation including a linked list or an array. Most of these methods are appropriate for sequences (i.e., duplicate key values are allowed), although special techniques applicable to sets are discussed in Section 9.3. The techniques from the first three sections of this chapter are most appropriate for searching a collection of records stored in RAM. Section 9.4 discusses hashing, a technique for organizing data in a table such that the location of each record within the table is a function of its key value. Hashing is appropriate when records are stored either in RAM or on disk.

Chapter 10 discusses tree-based methods for organizing information on disk, including a commonly used file structure called the B-tree. Nearly all programs that must organize large collections of records stored on disk use some variant of either hashing or the B-tree. Hashing is practical for only certain access functions (exact-match queries) and is generally appropriate only when duplicate key values are not allowed. B-trees are the method of choice for disk-based applications anytime hashing is not appropriate.

9.1 Searching Unsorted and Sorted Arrays

The simplest form of search has already been presented in Example 3.1: the sequential search algorithm. Sequential search on an unsorted list requires $\Theta(n)$ time in the worst case.

How many comparisons does linear search do on average? A major consideration is whether K is in list $\mathbf{L}$ at all. We can simplify our analysis by ignoring everything about the input except the position of K if it is found in $\mathbf{L}$. Thus, we have $n + 1$ distinct possible events: That K is in one of positions 0 to $n - 1$ in $\mathbf{L}$ (each with its own probability), or that it is not in $\mathbf{L}$ at all. We can express the probability that K is not in $\mathbf{L}$ as follows.

$$\mathbf{P}(K \notin \mathbf{L}) = 1 - \sum_{i=1}^{n} \mathbf{P}(K = \mathbf{L}[i]).$$

We can measure the average running time (in terms of the number of comparisons made) by noting that $k_i = i$ is the number of comparisons when K is in

position i of **L**. Define $k_0 = n$ to be the number of comparisons when *K* is not in **L**. Let p_i be the probability that *K* is in position i of **L**. Let p_0 be the probability that *K* is not in **L**. Then the average cost $\mathbf{T}(n)$ can be measured as follows.

$$\mathbf{T}(n) = k_0 p_0 + \sum_{i=1}^{n} k_i p_i \quad = \quad n p_0 + \sum_{i=1}^{n} i p_i.$$

What happens to the equation if we assume all the p_i's are equal (except p_0)?

$$\begin{aligned}
\mathbf{T}(n) &= p_0 n + \sum_{i=1}^{n} ip \quad = \quad p_0 n + p \sum_{i=1}^{n} i \\
&= p_0 n + p \frac{n(n+1)}{2} \\
&= p_0 n + \frac{1 - p_0}{n} \frac{n(n+1)}{2} \\
&= \frac{n + 1 + p_0(n-1)}{2}
\end{aligned}$$

Depending on the value of p_0, $\frac{n+1}{2} \leq \mathbf{T}(n) \leq n$.

For large collections of records that are searched repeatedly, sequential search is unacceptably slow. One way to reduce search time is to preprocess the records by sorting them. Given a sorted array, an obvious improvement over simple linear search is to test if the current element in **L** is greater than *K*. If it is, then we know that *K* cannot appear later in the array, and we can quit the search early. But this still does not improve the worst-case cost of the algorithm.

We can also observe that if we look at position 5 in **L** and find that *K* is bigger, then we rule out positions 0 to 4 in **L** as well. Since more is often better, what if we look at the last position in **L** and find that *K* is bigger yet? Then we know in one test that *K* is not in **L**. This is very useful to know, but what is wrong with this approach? While we learn a lot sometimes (in one comparison we might learn that *K* is not in the list), usually we learn only a little bit (that the last element is not *K*).

The question then becomes: What is the right amount to jump? This leads us to an algorithm known as **Jump Search**. For some value k, we check every k'th element in **L**, that is, we check elements $\mathbf{L}[k]$, $\mathbf{L}[2k]$, and so on. So long as *K* is greater than the values we are checking, we continue on. But when we reach a value in **L** greater than *K*, we do a linear search on the piece of length $k - 1$ that we know brackets *K* if it is in the list.

If $mk \leq n < (m+1)k$, then the total cost of this algorithm is at most $m+k-1$ 3-way comparisons. Therefore, the cost to run the algorithm on n items with a jump of size k is

$$\mathbf{T}(n, k) = m + k - 1 = \left\lfloor \frac{n}{k} \right\rfloor + k - 1.$$

What is the best value that we can pick for k? We want to minimize the cost:

$$\min_{1 \leq k \leq n} \left\{ \left\lfloor \frac{n}{k} \right\rfloor + k - 1 \right\}$$

Take the derivative and solve for $f'(x) = 0$ to find the minimum, which is $k = \sqrt{n}$. In this case, the worst case cost will be roughly $2\sqrt{n}$.

This example teaches us some lessons about algorithm design. We want to balance the work done while selecting a sublist with the work done while searching a sublist. In general, it is a good strategy to make subproblems of equal effort. This is an example of a **divide and conquer** algorithm.

What if we extend this idea to three levels? We'd jump to get a sublist, then jump to get a sub-sublist, then do sequential search. While it might make sense to do a two-level algorithm (that is, jump search jumps to find a sublist and then searches the sublist), it almost never makes sense to do a three-level algorithm. Instead, when we go beyond two levels, we nearly always generalize by using recursion. This leads us to the most commonly used search algorithm for sorted arrays, the binary search described in Section 3.5.

If we know nothing about the distribution of key values, then binary search is the best algorithm available for searching a sorted array. However, sometimes we do know something about the expected key distribution. Consider the typical behavior of a person looking up a name in a large dictionary. Most people certainly do not use sequential search! Typically, people use a modified form of binary search, at least until they get close to the word that they are looking for. The search generally does not start at the middle of the dictionary. A person looking for a word starting with 'S' generally assumes that entries beginning with 'S' start about three quarters of the way through the dictionary. Thus, they will first open the dictionary about three quarters of the way through and then make a decision based on what they find as to where to look next. In other words, people typically use some knowledge about the expected distribution of key values to "compute" where to look next. This form of "computed" binary search is called a **dictionary search**. In a dictionary search, we search $\mathbf{L}$ at a position p that is appropriate to the value of K as follows.

$$p = \frac{K - \mathbf{L}[1]}{\mathbf{L}[n] - \mathbf{L}[1]}$$

The location of a particular key within the key range is translated into the expected position for the corresponding record in the table, and this position is checked first. As with binary search, the value of the key found eliminates all records either above or below that position. The actual value of the key found can then be used to compute a new position within the remaining range of the table. The next check is made based on the new computation. This proceeds until either the desired record is found, or the table is narrowed until no records are left.

A variation on dictionary search is known as **Quadratic Binary Search**, and we will look at it in more detail since it is easier to analyze. Compute p and examine $\mathbf{L}[\lceil pn \rceil]$. If $K < \mathbf{L}[\lceil pn \rceil]$ then sequentially probe

$$\mathbf{L}[\lceil pn - i\sqrt{n} \rceil], i = 1, 2, 3, ...$$

until we reach a value less than or equal to K. Similarly for $K > \mathbf{L}[\lceil pn \rceil]$. We are now within $\sqrt{n}$ positions of K. Assume (for now) that this takes a constant number of comparisons. We then take our sublist of size $\sqrt{n}$ and repeat the process recursively.

What is the cost? Note that $\sqrt{c^n} = c^{n/2}$, and we will be repeatedly taking square roots of the current sublist size until we find the item we are looking for. Since $n = 2^{\log n}$ and we can cut $\log n$ in half only $\log \log n$ times, the cost is $\Theta(\log \log n)$ *if* the number of probes on jump search is constant.

The number of comparisons needed is

$$\sum_{i=1}^{\sqrt{n}} i\mathbf{P}(\text{need exactly } i \text{ probes}) = \sum_{i=1}^{\sqrt{n}} \mathbf{P}(\text{need at least } i \text{ probes}).$$

We require at least two probes to set the bounds, so the cost is

$$2 + \sum_{i=3}^{\sqrt{n}} \mathbf{P}(\text{need at least } i \text{ probes}).$$

We now make take advantage of a useful fact known as Čebyšev's Inequality: The probability that we need probe i times ($\mathbf{P}_i$) is

$$\mathbf{P}_i \leq \frac{p(1-p)n}{(i-2)^2 n} \leq \frac{1}{4(i-2)^2}$$

since $p(1-p) \leq 1/4$. This assumes uniformly distributed data. Thus, the expected number of probes is

$$2 + \sum_{i=3}^{\sqrt{n}} \frac{1}{4(i-2)^2} \approx 2.4112$$

Is this better than binary search? Theoretically yes, since a cost of $\mathrm{O}(\log \log n)$ grows slower than a cost of $\mathrm{O}(\log n)$. However, we have a situation here which illustrates the limits to the model of asymptotic complexity in some practical situations. Yes, $c_1 \log n$ does grow faster than $c_2 \log \log n$. In fact, its exponentially faster! But even so, for practical input sizes, the absolute cost difference is fairly small. Thus, the constant factors might play a role. Let's compare $\lg \lg n$ to $\lg n$.

n	$\lg n$	$\lg \lg n$	Diff
16	4	2	2
256	8	3	2.7
$64K$	16	4	4
2^{32}	32	5	6.4

It is not always practical to reduce an algorithm's growth rate. There is a practicality window for every problem, in that we have a practical limit to how big an input we wish to solve for. If our problem size never grows too big, it might not matter if we can reduce the cost by an extra log factor, since the constant factors in the two algorithms might overwhelm the savings.

For our two algorithms, let's look at the actual comparisons used. for Binary Search, we need about $\log n - 1$ total comparisons. Quadratic binary search requires about $2.4 \lg \lg n$ comparisons. If we incorporate this observation into our table, we get a different picture about the relative differences.

n	$\lg n - 1$	$2.4 \lg \lg n$	Diff
16	3	4.8	worse
256	7	7.2	$\approx$ same
$64K$	15	9.6	1.6
2^{32}	31	12	2.6

But we still are not done. This is only a count of comparisons! Binary search is inherently much simpler than quadratic binary search, since binary search only needs to calculating the midpoint position of the array before each comparison, while quadratic binary search must calculate an interpolation point which is more expensive. So the constant factors for quadratic binary search are even higher.

Not only are the constant factors worse on average, but quadratic binary search is far more dependent than binary search on good data distribution to perform well.

For example, imagine that you are searching a telephone directory for the name "Young." Normally you would look near the back of the book. If you found a name beginning with 'Z,' you might look just a little ways toward the front. If the next name you find also begins with 'Z,' you would look a little further toward the front. If this particular telephone directory were unusual in that half of the entries begin with 'Z,' then you would need to move toward the front many times, each time eliminating relatively few records from the search. In the extreme, the performance of interpolation search might not be much better than sequential search if the distribution of key values is badly calculated.

While it turns out that quadratic binary search is not a practical algorithm, this is not a typical situation. Fortunately, algorithm growth rates are usually well behaved, so that asymptotic algorithm analysis nearly always gives us a practical indication for which of two algorithms is better.

9.2 Self-Organizing Lists

While lists are most commonly ordered by key value, this is not the only viable option. Another approach to organizing lists for fast search is to order the records by expected frequency of access. Assume that we know, for each key k_i, the probability p_i that the record with key k_i will be requested. Assume also that the list is ordered so that the most frequently requested record is first, then the next most frequently requested record, and so on. Search in the list will be done sequentially, beginning with the first position. Over the course of many searches, the expected number of comparisons required for one search is

$$\overline{C}_n = 1p_1 + 2p_2 + ... + np_n.$$

In other words, the cost to access the first record is one (since one key value is looked at), and the probability of this occurring is p_1. The cost to access the second record is two (since we must look at the first and the second records' key values), with probability p_2, and so on. For n records, assuming that all searches are for records that actually exist, the probabilities p_1 through p_n must sum to one.

Certain probability distributions give easily computed results.

Example 9.1 Calculate the expected cost to search a list when each record has equal chance of being accessed (the classic sequential search through an unsorted list). Setting $p_i = 1/n$ yields

$$\overline{C}_n = \sum_{i=1}^{n} i/n = (n + 1)/2.$$

This result matches our expectation that half the records will be accessed on average by normal sequential search. If the records truly have equal access probabilities, then ordering records by frequency yields no benefit.

Example 9.2 The previous example did not take into account the probability of an unsuccessful search occurring. We can perform a more sophisticated analysis by accounting for this probability, which we will call p_0. Once again we assume that the desired record, if it indeed exists in the array, is equally likely to be in any position. Call the probability that the record appears in a particular slot p. Then, $p_0 + np = 1$, and $p = \frac{1-p_0}{n}$. Since the cost when the search is unsuccessful is n, and the cost when the record is found in position i is i, we can calculate as follows:

$$\begin{aligned}
\overline{C}_n &= p_0 n + \sum_{i=1}^{n} ip \\
&= p_0 n + p \sum_{i=1}^{n} i \\
&= p_0 n + p \frac{n(n+1)}{2} \\
&= p_0 n + \frac{1-p_0}{n} \frac{n(n+1)}{2} \\
&= \frac{n+1+p_0(n-1)}{2}.
\end{aligned}$$

Depending on the value of p_0 (which can range from 0 to 1), $\frac{n+1}{2} \leq \overline{C}_n \leq n$.

A geometric probability distribution yields quite different results.

Example 9.3 Calculate the expected cost for searching a list ordered by frequency when the probabilities are defined as

$$p_i = \begin{cases} 1/2^i & \text{if } 1 \leq i \leq n-1 \\ 1/2^{n-1} & \text{if } i = n. \end{cases}$$

Then,

$$\overline{C}_n \approx \sum_{i=1}^{n} (i/2^i) \approx 2.$$

For this example, the expected number of accesses is a constant. This is because the probability for accessing the first record is high, the second is much lower but still much higher than for record three, and so on. This shows that for some probability distributions, ordering the list by frequency can yield an efficient search technique.

In many search applications, real access patterns follow a rule of thumb called the **80/20 rule**. The 80/20 rule says that 80% of the record accesses are to 20% of the records. The values of 80 and 20 are only estimates; every application has its own values. However, behavior of this nature occurs surprisingly often in practice (which explains the success of caching techniques widely used by disk drive and CPU manufacturers for speeding access to data stored in slower memory; see the discussion on buffer pools in Section 8.3). When the 80/20 rule applies, we can expect reasonable search performance from a list ordered by frequency of access.

Example 9.4 The 80/20 rule is an example of a **Zipf distribution**. Naturally occurring distributions often follow a Zipf distribution. Examples include the observed frequency for the use of words in a natural language such as English, and the size of the population for cities (i.e., view the relative proportions for the populations as equivalent to the "frequency of use"). Zipf distributions are related to the Harmonic Series defined in Equation 2.10. Define the Zipf frequency for item i in the distribution for n records as $1/(i\mathcal{H}_n)$. The expected cost for the series whose members follow this Zipf distribution will be

$$\overline{C}_n = \sum_{i=1}^{n} i/i\mathcal{H}_n = n/\mathcal{H}_n \approx n/\log_e n.$$

When a frequency distribution follows the 80/20 rule, the average search looks at about one tenth of the records in a table ordered by frequency.

In many applications, we have no means of knowing in advance which records will be accessed most often. To complicate matters further, certain records might be accessed frequently for a brief period of time, and then rarely thereafter. Thus, the probability of access for records might change over time. **Self-organizing lists** seek to solve both of these problems.

Self-organizing lists modify the order of records within the list based on the actual pattern of record access. Self-organizing lists use a heuristic for deciding how to to reorder the list. These heuristics are similar to the rules for managing buffer pools (see Section 8.3). In fact, a buffer pool is a form of self-organizing list.

Ordering the buffer pool by expected frequency of access is a good strategy, since typically we must search the contents of the buffers to determine if the desired information is already in main memory. When ordered by frequency of access, the buffer at the end of the list will be the one most appropriate for reuse when a new page of information must be read. Below are three traditional heuristics for managing self-organizing lists:

1. The most obvious way to keep a list ordered by frequency would be to store a count of accesses to each record and always maintain records in this order. This method will be referred to as **count**. Count is similar to the least frequently used buffer replacement strategy. Thus, whenever a record is accessed, it might move toward the front of the list if its number of accesses becomes greater than a record preceding it. Clearly, count will store the records in the order of frequency that has actually occurred so far. Besides requiring space for the access counts, count does not react well to changing frequency of access over time. Once a record has been accessed a large number of times under the frequency count system, it will remain near the front of the list regardless of further access history.
2. Bring a record to the front of the list when it is found, pushing all the other records back one position. This is analogous to the least recently used buffer replacement strategy and is called **move-to-front**. This heuristic is easy to implement if the records are stored using a linked list. When records are stored in an array, bringing a record forward from near the end of the array will result in a large number of records changing position. Move-to-front is an efficient heuristic in that it requires at most twice the accesses required by the **optimal static ordering** for n records when at least n searches are performed. In other words, if we had known the series of (at least n) searches in advance and had stored the records in order of frequency so as to minimize the total cost for these accesses, this cost would be at least half the cost required by the move-to-front heuristic. (This will be proved using amortized analysis in Section 14.3.) Finally, move-to-front responds well to local changes in frequency of access, in that if a record is frequently accessed for a brief period of time it will be near the front of the list during that period of access.
3. Swap any record found with the record immediately preceding it in the list. This heuristic is called **transpose**. Transpose is good for list implementations based on either linked lists or arrays. Frequently used records will, over time, move to the front of the list. Records that were once frequently accessed but are no longer used will slowly drift toward the back. Thus, it appears to have

good properties with respect to changing frequency of access. Unfortunately, there are some pathological sequences of access that can make transpose perform poorly. Consider the case where the last record of the list (call it X) is accessed. This record is then swapped with the next-to-last record (call it Y), making Y the last record. If Y is now accessed, it swaps with X. A repeated series of accesses alternating between X and Y will continually search to the end of the list, since neither record will ever make progress toward the front. However, such pathological cases are unusual in practice.

Example 9.5 Assume that we have eight records, with key values A to H, and that they are initially placed in alphabetical order. Now, consider the result of applying the following access pattern:

$$F\ D\ F\ G\ E\ G\ F\ A\ D\ F\ G\ E.$$

If the list is organized by the count heuristic, the final list resulting from these accesses will be

$$F\ G\ D\ E\ A\ B\ C\ H,$$

and the total cost for the twelve accesses will be 45 comparisons. (Assume that when a record's frequency count goes up, it moves forward in the list to become the last record with that value for its frequency count. After the first two accesses, F will be the first record and D will be the second.)

If the list is organized by the move-to-front heuristic, then the final list will be

$$E\ G\ F\ D\ A\ B\ C\ H,$$

and the total number of comparisons required is 54.

Finally, if the list is organized by the transpose heuristic, then the final list will be

$$A\ B\ F\ D\ G\ E\ C\ H,$$

and the total number of comparisons required is 62.

While self-organizing lists do not generally perform as well as search trees or a sorted list, both of which require $\mathrm{O}(\log n)$ search time, there are many situations in which self-organizing lists prove a valuable tool. Obviously they have an advantage over sorted lists in that they need not be sorted. This means that the cost to insert a new record is low, which could more than make up for the higher search cost when insertions are frequent. Self-organizing lists are simpler to implement than search trees and are likely to be more efficient for small lists. Nor do they require

additional space. Finally, in the case of an application where sequential search is "almost" fast enough, changing an unsorted list to a self-organizing list might speed the application enough at a minor cost in additional code.

As an example of applying self-organizing lists, consider an algorithm for compressing and transmitting messages. The list is self-organized by the move-to-front rule. Transmission is in the form of words and numbers, by the following rules:

1. If the word has been seen before, transmit the current position of the word in the list. Move the word to the front of the list.
2. If the word is seen for the first time, transmit the word. Place the word at the front of the list.

Both the sender and the receiver keep track of the position of words in the list in the same way (using the move-to-front rule), so they agree on the meaning of the numbers that encode repeated occurrences of words. For example, consider the following example message to be transmitted (for simplicity, ignore case in letters).

The car on the left hit the car I left.

The first three words have not been seen before, so they must be sent as full words. The fourth word is the second appearance of "the," which at this point is the third word in the list. Thus, we only need to transmit the position value "3." The next two words have not yet been seen, so must be sent as full words. The seventh word is the third appearance of "the," which coincidentally is again in the third position. The eighth word is the second appearance of "car," which is now in the fifth position of the list. "I" is a new word, and the last word "left" is now in the fifth position. Thus the entire transmission would be

The car on 3 left hit 3 5 I 5.

This approach to compression is similar in spirit to Ziv-Lempel coding, which is a class of coding algorithms commonly used in file compression utilities. Ziv-Lempel coding will replace repeated occurrences of strings with a pointer to the location in the file of the first occurrence of the string. The codes are stored in a self-organizing list in order to speed up the time required to search for a string that has previously been seen.

9.3 Bit Vectors for Representing Sets

Determining whether a value is a member of a particular set is a special case of searching for keys in a sequence of records. Thus, any of the methods for searching

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	1	1	0	1	0	1	0	0	0	1	0	1	0	0

Figure 9.1 The bit table for the set of primes in the range 0 to 15. The bit at position i is set to 1 if and only if i is prime.

discussed in this book can be used to check for set membership. However, we can also take advantage of the restricted circumstances imposed by this problem to develop another representation.

In the case where the set elements fall within a limited key range, we can represent the set using a bit array with a bit position allocated for each potential member. Those members actually in the set store a value of 1 in their corresponding bit; those members not in the set store a value of 0 in their corresponding bit. For example, consider the set of primes between 0 and 15. Figure 9.1 shows the corresponding bit table. To determine if a particular value is prime, we simply check the corresponding bit. This representation scheme is called a **bit vector** or a **bitmap**. The mark array used in several of the graph algorithms of Chapter 11 is an example of such a set representation.

If the set fits within a single computer word, then set union, intersection, and difference can be performed by logical bitwise operations. The union of sets A and B is the bitwise OR function (whose symbol is `|` in **C**++). The intersection of sets A and B is the bitwise AND function (whose symbol is `&` in **C**++). For example, if we would like to compute the set of numbers between 0 and 15 that are both prime and odd numbers, we need only compute the expression

$$0011010100010100 \,\&\, 0101010101010101.$$

The set difference $A - B$ can be implemented in **C**++ using the expression `A&~B` (`~` is the symbol for bitwise negation). For larger sets that do not fit into a single computer word, the equivalent operations can be performed in turn on the words making up the entire bit vector.

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 organiza-

tion 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 tables: by direct access based on key value. The process of accessing a record by mapping a key value to a position in the table is called **hashing**. Most hashing schemes place records in the table 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 is usually 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 when hashing is to arrange things such that, for any key value K and some hash function **h**, $0 \leq \mathbf{h}(K) < M$, we have the key of $\mathbf{HT}[i]$ equal to K.

Hashing is generally appropriate only for sets; that is, hashing is generally not used for applications where multiple records with the same key value are permitted. Hashing is generally not suitable for 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 can be restricted to this query, hashing is usually the search method of choice since it is extremely efficient when implemented correctly. As you shall see in this section, however, there are many approaches to hashing and it is easy to devise an inappropriate implementation. Hashing is suitable for both in-memory and disk-based searching and is one of the two 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, introduction to the concept of hashing, consider the case where the key range is small compared to the number of records, for example, when there are n records with unique key values in the range 0 to $n-1$. In this simple case, a record with key i can be stored in $\mathbf{HT}[i]$, and the hash function is simply $\mathbf{h}(K) = K$ (in fact, we don't even need to store the key value as part of the record in this situation since it is the same as the index). To find the record with key value i, simply look in $\mathbf{HT}[i]$.

Typically, there are many more values in the key range than there are slots in the hash table. For a more realistic example, suppose the key can take any value in the range 0 to 65,535 (i.e., the key is a two-byte unsigned integer), and we expect

to store approximately 1000 records at any given time. It is probably impractical in this situation to use a hash table with 65,536 slots, leaving most of them empty. Instead, we must devise a hash function that allows us to store the records in a much smaller table. Since 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 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.

[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; 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 CD-ROM. 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-ROM.

To be practical, a database organized by hashing must store records in a high percentage of the hash table slots, typically keeping the table at least half full. Since collisions are extremely likely to occur under these conditions, 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.

In general, 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 probablility 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. Natural distributions are geometric. For example, the populations of the 100 largest cities in the United States will be clustered toward the bottom of the range, with a few outliers at the top (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. Collected data are likely to be skewed in some way. For example, samples from the field might be rounded to, say, the nearest 5 (i.e., all numbers end in 5 or 0). If the input is a collection of common English words, the beginning letter will be poorly distributed. Note that in each of these examples, 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 records 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 9.6 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. Since these bits are likely to be poorly distributed (e.g., a high percentage might be even numbers with the low order bit being zero), the result will also be poorly distributed. This example shows that the size of the table M (typically the value used as the modulus) is critical to a good hash function.

Example 9.7 One good hash function for numerical values is called the **mid-square** method. The mid-square method squares the key value, and then takes the middle r bits of the result for a table of size 2^r. This works well since most or all bits of the key value contribute to the result. For example, consider 4-digit numbers in base 10, with the goal of hashing to a table with two digits (in the range 0 to 99). If the input is the number 4567, squaring yields an 8-digit number, 20857489. The middle two digits are 57. All digits (even all bits if viewed as binary numbers) contribute to the middle two digits of the squared value. To see this, note that the square of 5567 is 30991489 (middle two digits are 91) and the square of 4568 is 20866624 (middle two digits are 66).

Example 9.8 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 the string. If M is small, it should do a good job of distributing strings evenly among the hash table slots, since it gives equal weight to all characters. This is an example of the **folding method** for designing a hash function. Note that the order of the characters in the string has no effect on the result of this hash function. 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. Since 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 good distribution results. For a hash table of size 1000, the distribution is terrible.

Example 9.9 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 long *lkey = (unsigned long *)key;
  int length = strlen(key)/4;
  unsigned long sum = 0;
  for(int i=0; i<length; i++)
    sum += lkey[i];

  // Now deal with the chars at the end
  length = strlen(key) - length*4;
  char temp[4];
  lkey = (unsigned long *)temp;
  lkey[0] = 0;
  for(int i=0; i<length; i++)
    temp[i] = key[length*4+i];
  sum += lkey[0];
  return sum % M;
}
```

This function takes a string as input. It takes the string four bytes at a time, and interprets those four ASCII bytes as a single long integer value. The four-byte quantities are added together. In the end, the resulting sum is converted to the range 0 to $M - 1$ using the modulus operator. For this to work well, it is important to pick a good value for M. A prime number is a good choice, since using a prime number for the modulus will make the result a function of all bits in the sum.

9.4.2 Open Hashing

While the goal of a hash function is to minimize collisions, collisions are normally 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**).[2] 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.

A simple 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 9.2 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 order of insertion, 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, since we know to stop searching the list once we encounter a key that is greater than the one being 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 a few of the slots, then the cost to access a record will be much higher as many elements on the linked list must be searched.

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

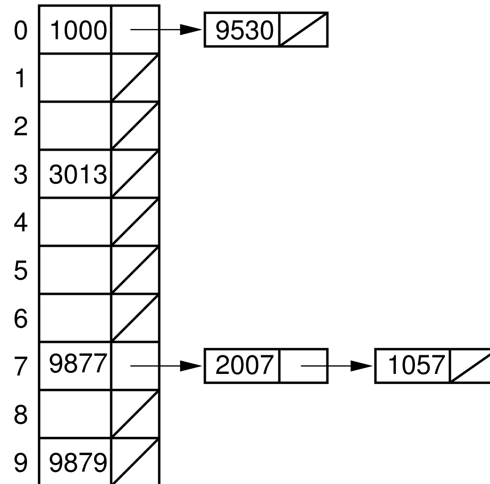

Figure 9.2 An illustration of open hashing for seven numbers stored in a ten-slot hash table using the hash function $\mathbf{h}(K) = K \bmod 10$. The numbers are inserted in the order 9877, 2007, 1000, 9530, 3013, 9879, and 1057. Two of the values hash to slot 0, one value hashes to slot 2, three of the values hash to slot 7, and one value hashes to slot 9.

Open hashing is most appropriate when the hash table is kept in main memory, implemented with a standard in-memory linked list. Storing an open hash table on disk in an efficient way is difficult, since 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 placed in a bin. Multiple records may hash to the same bin, but this initial binning should 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 i has a **home position** that is $\mathbf{h}(k_i)$, the slot computed by the hash function. If a record *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

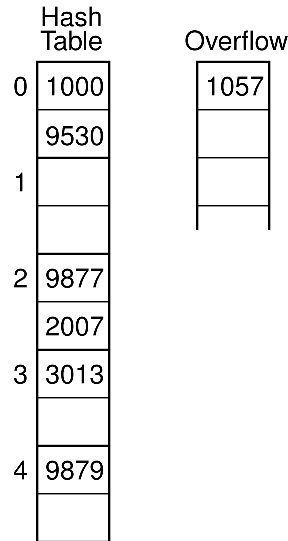

Figure 9.3 An illustration of bucket hashing for seven numbers stored in a five-bucket hash table using the hash function $\mathbf{h}(K) = K \bmod 5$. Each bucket contains two slots. The numbers are inserted in the order 9877, 2007, 1000, 9530, 3013, 9879, and 1057. Two of the values hash to bucket 0, three values hash to bucket 2, one value hashes to bucket 3, and one value hashes to bucket 4. Since bucket 2 cannot hold three values, the third one ends up in the overflow bucket.

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 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 9.3 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 bucketing first hashes the key value to its home position as though bucketing were not being used. If the home position is full, then the record is pushed down toward the end of the bucket. If the bottom of the bucket is reached, then the collision resolution routine moves to the top of the bucket looking 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. The advantage of this approach is that collisions are reduced, since any slot can be a home position rather than just the first slot in the bucket.

Bucket methods are good for implementing hash tables stored on disk, since 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. Processing an insert or search operation requires only one disk access, unless the bucket is full. If the bucket is full, 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 "classic" 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 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** generated by the collision resolution policy. Insertion works as follows:

```
// Insert e into hash table HT
template <typename Key, typename Elem, typename Comp, typename getKey>
bool hashdict<Key, Elem, Comp, getKey>::
hashInsert(const Elem& e) {
  int home;                        // Home position for e
  int pos = home = h(getKey::key(e)); // Init probe sequence
  for (int i=1; !(Comp::eq(EMPTYKEY,
                           getKey::key(HT[pos]))); i++) {
    pos = (home + p(getKey::key(e), i)) % M; // probe
    if (Comp::eq(getKey::key(e), getKey::key(HT[pos])))
      return false;                           // Duplicate
  }
  HT[pos] = e;                     // Insert e
  return true;
}
```

In this implementation, we fist compare the key for the record found in the home slot against the key value **EMPTY** to determine if the home slot is empty or not. If the home slot is occupied, then we use the **probe function**, $\mathbf{p}(R, i)$. 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 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 as follows:

```
// Search for the record with Key K
template <typename Key, typename Elem, typename Comp, typename getKey>
bool hashdict<Key, Elem, Comp, getKey>::
hashSearch(const Key& K, Elem& e) const {
  int home;              // Home position for K
  int pos = home = h(K); // Initial posit on probe sequence
  for (int i = 1;
       !Comp::eq(K, getKey::key(HT[pos])) &&
       !Comp::eq(EMPTYKEY, getKey::key(HT[pos])); i++)
    pos = (home + p(K, i)) % M;  // Next on probe sequence
  if (Comp::eq(K, getKey::key(HT[pos]))) {  // Found it
    e = HT[pos];
    return true;
  }
  else return false;      // K not in hash table
}
```

Both the insert and the 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.

	(a)
0	9050
1	1001
2	
3	
4	
5	
6	
7	9877
8	2030
9	

	(b)
0	9050
1	1001
2	
3	
4	
5	
6	
7	9877
8	2037
9	1059

Figure 9.4 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.

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

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

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.

It is important to understand that, 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 in how to do collision resolution. In fact, linear probing is one of the worst possible approaches to collision resolution. The main problem is illustrated by Figure 9.4. 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 9.4(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 (assum-

ing that every slot in the table has equal probability of being hashed to initially). In this example, 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 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 as illustrated by Figure 9.4(b), then the following record will end up in slot 2 with probability 6/10. 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 possibility is 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.

A good probe sequence will cycle through all slots in the hash table before returning to the home position. 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 key values into two sets stored in two disjoint sections of the hash table. As long as both sections 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.

Constant c must be relatively prime to M to have a linear probing sequence that visits all slots in the table. 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, since 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 insertions and searches use the same sequence of "random" numbers. The probe function would be

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

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

Example 9.10 Consider a table of size $M = 101$, with $r_1 = 2$, $r_2 = 5$, and $r_3 = 32$. Assume that we have two keys k_1 and k_2 where $\mathbf{h}(k_1) = 30$ and $\mathbf{h}(k_2)$ = 28. The probe sequence for k_1 is 30, then 32, then 35, then 62. The probe sequence for k_2 is 28, then 30, then 33, then 60. 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 technique that eliminates primary clustering is called **quadratic probing**. Here the probe function is

$$\mathbf{p}(K, i) = i^2.$$

Thus, the ith value in the probe sequence is $(\mathbf{h}(K) + i^2) \bmod M$. Under quadratic probing, two keys with different home positions will have diverging probe sequences. However, quadratic probing has the disadvantage that not all hash table slots will necessarily be on the probe sequence.

Example 9.11 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.

Both pseudo-random probing and quadratic probing eliminate primary clustering, the problem of keys sharing substantial segments of a probe sequence. If two keys hash to the same home position, however, then they will follow the same probe sequence. This is because the probe sequences generated by pseudo-random and quadratic probing are entirely a function of the home position, not the original key value. (Note that function $\mathbf{p}$ does not use its input parameter K for either pseudo-random or quadratic probing.) If the hash function causes a cluster to 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. A simple technique for doing this is to return to linear probing by a constant step size for the probe function, but to have the constant be based on 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 9.12 Again, 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.

We conclude this section by presenting Figure 9.5, which 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 project at the end of the 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 an unsuccessful search for the record (recall that a successful search for the record should generate an error since two records with the same key are not allowed).

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 in 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

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

  bool hashInsert(const Elem&);
  bool hashSearch(const Key&, Elem&) const;

public:
  hashdict(int sz, Elem e){ // e defines an EMPTY slot
    M = sz;  EMPTY = e; EMPTYKEY = getKey::key(e);
    currcnt = 0;  HT = new Elem[sz]; // Make HT of size sz
    for (int i=0; i<M; i++) HT[i] = EMPTY; // Initialize HT
  }

  ~hashdict() { delete HT; }
  void clear() { // Clear the dictionary
    for (int i=0; i<M; i++) HT[i] = EMPTY;
    currcnt = 0;
  }

  bool insert(const Elem& e) { // Insert e into dictionary
    if (currcnt == M) return false;
    if (hashInsert(e)) {
      currcnt++;
      return true;
    }
    else return false;
  }
```

Figure 9.5 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** were presented earlier in this section.

```
  bool remove(const Key& K, Elem& e) { return false; } // Not implemented

  bool removeAny(Elem& e) {  // Remove the first element
    for (int i=0; i<M; i++)
      if (!Comp::eq(EMPTYKEY, getKey::key(HT[i]))) {
        e = HT[i];  HT[i] = EMPTY;
        currcnt--;  return true;
      }
    return false;
  }

  bool find(const Key& K, Elem& e) const
    { return hashSearch(K, e); }
  int size() { return currcnt; } // Number stored in table
};
```

Figure 9.5 (continued)

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

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

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. Analysis shows that the 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 9.6 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, since 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 9.6 we see that the cost for hashing when the table is not too full is typically close to one record access, which 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. Based on this analysis, the rule of thumb is to design the system so that the hash table gets only about half full, since 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%

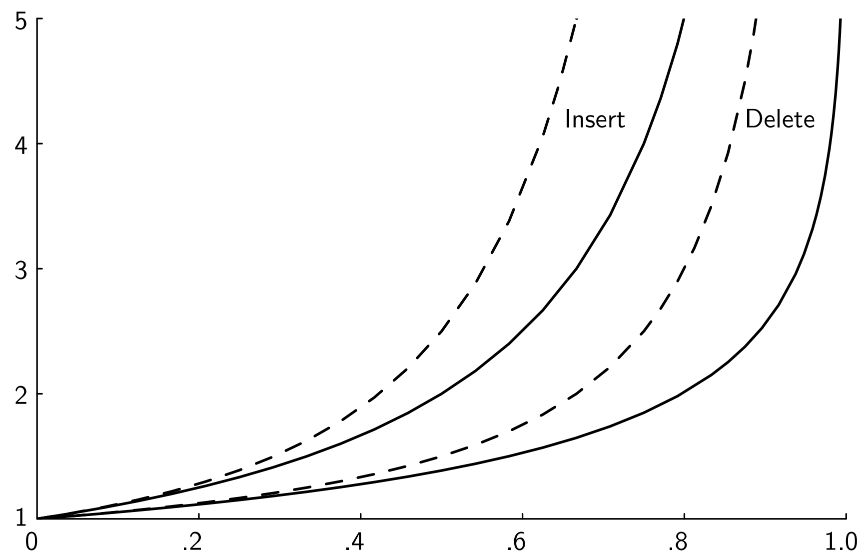

Figure 9.6 Growth of expected record accesses with α. The horizontal axis is the value for α, the vertical axis is the expected number of accesses to the hash table. Solid lines show the cost for "random" probing (a theoretical lower bound on the cost), while dashed lines show the cost for linear probing (a relatively poor collision resolution strategy). The two leftmost lines show the cost for insertion and unsuccessful search; the two rightmost lines show the cost for deletion and successful search.

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, since 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, since 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. Thus, the delete process cannot simply mark the slot as empty, since this will isolate records further down the probe sequence. For example, in Figure 9.4(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 probe 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 through a probe sequence, the search procedure is to continue 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 since 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 a 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*[Knu81].

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 [Knu81] and *Concrete Mathematics: A Foundation for Computer Science* by Graham, Knuth, and Patashnik [GKP89].

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 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 Example 9.2). What can you say qualitatively about the rate of increase in expected cost as the probability of unsuccessful search grows?

9.2 Modify the binary search routine of Section 3.5 to implement interpolation search. Assume that keys are in the range 1 to 10,000, and that all key values within the range are equally likely to occur.

9.3 Recall that binary search costs $\Theta(\log n)$ while interpolation serach costs $\Theta(\log \log n)$ in the average case.

- **(a)** Make a table showing the value of $\log n$ versus $\log \log n$ for $n = 10$, 100, 2000, 10,000, 100,000, and 1,000,000. What is the break-even point for these equations?
- **(b)** Now consider the cost of interpolation search and binary search in more detail. Since interpolation search is more complicated, it is reasonable to expect the constant factor is higher. In fact, interpolation search can be expected to do about $3 \log \log n$ searches on average, while binary search can be expected to do about $\log n - 1$ searches on average. Show a table similar to part (a) using these values. What is the break-even point these algorithms.

(c) In fact, not only does interpolation search do more searches per iteration, but since it is using a division to calcuate the next position to search, it can be expected to be even slower by a constant factor. Let us say that the additional computation accounts for a factor of two, yielding a cost of $6 \log \log n$ for interpolation search versus $\log n - 1$ for binary search. Show the table for these values and calculate the break-even point for the algorithms.

9.4 Write an algorithm to find the Kth smallest value in an unsorted array of n numbers ($K <= n$). Your algorithm should require $\Theta(n)$ time in the average case. Hint: Your algorithm shuld look similar to Quicksort.

9.5 Graph the equations $\mathbf{T}(n) = \log_2 n$ and $\mathbf{T}(n) = n/\log_e n$. Which gives the better performance, binary search on a sorted list, or sequential search on a list ordered by frequency where the frequence conforms to a Zipf distribution? Characterize the difference in running times.

9.6 Assume that the values A through H are stored in a self-organizing list, initially in ascending order. Consider the three self-organizing list heuristics: count, move-to-front, and transpose. For count, assume that the record is moved ahead in the list passing over any other record that its count is now greater than. For each, show the resulting list and the total number of comparisons required resulting from the following series of accesses:

$$D\ H\ H\ G\ H\ E\ G\ H\ G\ H\ E\ C\ E\ H\ G.$$

9.7 For each of the three self-organizing list heuristics (count, move-to-front, and transpose), describe a series of record accesses for which it would require the greatest number of comparisons of the three.

9.8 Write an algorithm to implement the frequency count self-organizing list heuristic, assuming that the list is implemented using an array. In particular, write a function **`FreqCount`** that takes as input a value to be searched for and which adjusts the list appropriately. If the value is not already in the list, add it to the end of the list with a frequency count of one.

9.9 Write an algorithm to implement the move-to-front self-organizing list heuristic, assuming that the list is implemented using an array. In particular, write a function **`MoveToFront`** that takes as input a value to be searched for and which adjusts the list appropriately. If the value is not already in the list, add it to the beginning of the list.

9.10 Write an algorithm to implement the transpose self-organizing list heuristic, assuming that the list is implemented using an array. In particular, write a function **`transpose`** that takes as input a value to be searched for and

which adjusts the list appropriately. If the value is not already in the list, add it to the end of the list.

9.11 Write functions for computing union, intersection, and set difference on arbitrarily long bit vectors used to represent set membership as described in Section 9.3. Assume that for each operation both vectors are of equal length.

9.12 Compute the probabilities for the following situations. These probabilities can be computed analytically, or you may write a computer program to generate the probabilities by simulation.

(a) Out of a group of 23 students, what is the probability that 2 students share the same birthday?
(b) Out of a group of 100 students, what is the probability that 3 students share the same birthday?
(c) How many students must be in the class for the probability to be at least 50% that there are 2 who share a birthday in the same month?

9.13 Assume that you are hashing key K to a hash table of n slots (indexed from 0 to $n-1$). For each of the following functions $\mathbf{h}(K)$, is the function acceptable as a hash function (i.e., would the hash program work correctly for both insertions and searches), and if so, is it a good hash function? Function `Random(n)` returns a random integer between 0 and $n-1$, inclusive.

(a) $\mathbf{h}(k) = k/n$ where k and n are integers.
(b) $\mathbf{h}(k) = 1$.
(c) $\mathbf{h}(k) = (k + \text{Random}(n)) \bmod n$.
(d) $\mathbf{h}(k) = k \bmod n$ where n is a prime number.

9.14 Assume that you have a seven-slot closed hash table (the slots are numbered 0 through 6). Show the final hash table that would result if you used the hash function $h(\mathbf{k}) = \mathbf{k}$ mod 7 and linear probing on this list of numbers: 3, 12, 9, 2. After inserting the record with key value 2, list for each empty slot the probability that it will be the next one filled.

9.15 Assume that you have a ten-slot closed hash table (the slots are numbered 0 through 9). Show the final hash table that would result if you used the hash function $h(\mathbf{k}) = \mathbf{k}$ mod 10 and quadratic probing on this list of numbers: 3, 12, 9, 2, 79, 46. After inserting the record with key value 46, list for each empty slot the probability that it will be the next one filled.

9.16 Assume that you have a ten-slot closed hash table (the slots are numbered 0 through 9). Show the final hash table that would result if you used the hash function $h(\mathbf{k}) = \mathbf{k}$ mod 10 and pseudo-random probing on this list of numbers: 3, 12, 9, 2, 79, 44. The permutation of offsets to be used by the

pseudo-random probing will be: 5, 9, 2, 1, 4, 8, 6, 3, 7. After inserting the record with key value 44, list for each empty slot the probability that it will be the next one filled.

9.17 What is the result of running **sfold** on the following strings? Assume a hash table size of 101 slots.

(a) HELLO WORLD
(b) NOW HEAR THIS
(c) HEAR THIS NOW

9.18 Using closed hashing, with double hashing to resolve collisions, insert the following keys into a hash table of thirteen slots (the slots are numbered 0 through 12). The hash functions to be used are H1 and H2, defined below. You should show the hash table after all eight keys have been inserted. Be sure to indicate how you are using H1 and H2 to do the hashing. Function Rev(k) reverses the decimal digits of k, for example, $\text{Rev}(37) = 73$; $\text{Rev}(7) = 7$.

H1(k) = k mod 13.
H2(k) = (Rev($k + 1$) mod 11).

Keys: 2, 8, 31, 20, 19, 18, 53, 27.

9.19 Write an algorithm for a deletion function for hash tables that replaces the record with a special value indicating a tombstone. Modify the functions **hashInsert** and **hashSearch** to work correctly with tombstones.

9.20 Consider the following permutation for the numbers 1 to 6:

2, 4, 6, 1, 3, 5.

Analyze what will happen if this permutation is used by an implementation of pseudo-random probing on a hash table of size seven. Will this permutation solve the problem of primary clustering? What does this say about selecting a permutation for use when implementing pseudo-random probing?

9.7 Projects

9.1 Implement a binary search and the quadratic binary search of Section 9.1. Run your implementations over a large range of problem sizes, timing the results for each algorithm. Graph and compare these timing results.

9.2 Implement the three self-organizing list heuristics count, move-to-front, and transpose. Compare the cost for running the three heuristics on various input data. The cost metric should be the total number of comparisons required

when searching the list. It is important to compare the heuristics using input data for which self-organizing lists are reasonable, that is, on frequency distributions that are uneven. One good approach is to read text files. The list should store individual words in the text file. Begin with an empty list, as was done for the text compression example of Section 9.2. Each time a word is encountered in the text file, search for it in the self-organizing list. If the word is found, reorder the list as appropriate. If the word is not in the list, add it to the end of the list and then reorder as appropriate.

9.3 Implement the text compression system described in Section 9.2.

9.4 Implement a system for managing document retrieval. Your system should have the ability to insert (abstract references to) documents into the system, associate keywords with a given document, and to search for documents with specified keywords.

9.5 Implement a database stored on disk using bucket hashing. Define records to be 128 bytes long with a 4-byte key and 120 bytes of data. The remaining 4 bytes are available for you to store necessary information to support the hash table. A bucket in the hash table will be 1024 bytes long, so each bucket has space for 8 records. The hash table should consist of 27 buckets (total space for 216 records with slots indexed by positions 0 to 215) followed by the overflow bucket at record position 216 in the file. The hash function for key value K should be $K \bmod 213$. (Note that this means the last three slots in the table will not be home positions for any record.) The collision resolution function should be linear probing with wrap-around within the bucket. For example, 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 a bucket is full, the record should be placed in the overflow section at the end of the file.

Your hash table should implement the dictionary ADT of Section 4.2. When you do your testing, assume that the system is meant to store about 100 or so records at a time.

9.6 Implement the dictionary ADT of Section 4.2 by means of a hash table with linear probing as the collision resolution policy. You might wish to begin with the code of Figure 9.5. Using empirical simulation, determine the cost of insert and delete as α grows (i.e., reconstruct the dashed lines of Figure 9.6). Then, repeat the experiment using quadratic probing and pseudo-random probing. What can you say about the relative performance of these three collision resolution policies?