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 that we have a collection $\mathbf{L}$ 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 from record j for $1 \leq j \leq n$. Given a particular key value K, the **search problem** is to locate a record (k_j, I_j) in $\mathbf{L}$ 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 4.4. 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. 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 an array such that the location of each record within the array 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 dynamic 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 position having 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

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

where $\mathbf{P}(x)$ is the probability of event x.

Let p_i be the probability that K is in position i of $\mathbf{L}$ (indexed from 0 to $n - 1$. For any position i in the list, we must look at $i + 1$ records to reach it. So we say that the cost when K is in position i is $i + 1$. When K is not in $\mathbf{L}$, sequential search will require n comparisons. Let p_n be the probability that K is not in $\mathbf{L}$. Then the average cost $\mathbf{T}(n)$ will be

$$\mathbf{T}(n) = np_n + \sum_{i=0}^{n-1}(i+1)p_i.$$

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

$$\begin{aligned}
\mathbf{T}(n) &= p_n n + \sum_{i=0}^{n-1}(i+1)p \\
&= p_n n + p\sum_{i=1}^{n} i \\
&= p_n n + p\frac{n(n+1)}{2} \\
&= p_n n + \frac{1-p_n}{n}\frac{n(n+1)}{2} \\
&= \frac{n+1+p_n(n-1)}{2}
\end{aligned}$$

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

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 first at position 1 in sorted array **L** and find that *K* is bigger, then we rule out position 0 as well as position 1. Because more is often better, what if we look at position 2 in **L** and find that *K* is bigger yet? This rules out positions 0, 1, and 2 with one comparison. What if we carry this to the extreme and look first at the last position in **L** and find that *K* is bigger? Then we know in one comparison that *K* is not in **L**. This is very useful to know, but what is wrong with the conclusion that we should always start by looking at the last position? The problem is that, 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 j, we check every j'th element in **L**, that is, we check elements **L**[j], **L**[$2j$], 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 $j-1$ that we know brackets *K* if it is in the list.

If we define m such that $mj \leq n < (m+1)j$, then the total cost of this algorithm is at most $m+j-1$ 3-way comparisons. (They are 3-way because at each comparison of *K* with some $\mathbf{L}[i]$ we need to know if *K* is less than, equal to, or greater than $\mathbf{L}[i]$.) Therefore, the cost to run the algorithm on n items with a jump of size j is

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

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

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

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

This example invokes a basic principle of 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 would first make jumps of some size j to find a sublist of size $j-1$ whose end values bracket value *K*. We would then work through this sublist by making jumps of some smaller size, say j_1. Finally, once we find a bracketed sublist of size $j_1 - 1$, we would do sequential search to complete the process.

This probably sounds convoluted to do two levels of jumping to be followed by a sequential search. While it might make sense to do a two-level algorithm (that is, jump search jumps to find a sublist and then does sequential search on the sublist), it almost never seems to make 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 (see Exercise 9.22). However, sometimes we do know something about the expected key distribution. Consider the typical behavior of a person looking up a word 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, he or

she will first open the dictionary about three quarters of the way through and then make a decision based on what is found 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** or **interpolation search**. In a dictionary search, we search $\mathbf{L}$ at a position p that is appropriate to the value of K as follows.

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

This equation is computing the position of K as a fraction of the distance between the smallest and largest key values. This will next be translated into that position which is the same fraction of the way through the array, 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 array. The next check is made based on the new computation. This proceeds until either the desired record is found, or the array is narrowed until no records are left.

A variation on dictionary search is known as **Quadratic Binary Search** (QBS), and we will analyze this in detail because its analysis is easier than that of the general dictionary search. QBS will first compute p and then examine $\mathbf{L}[\lceil pn \rceil]$. If $K < \mathbf{L}[\lceil pn \rceil]$ then QBS will sequentially probe to the left by steps of size $\sqrt{n}$, that is, we step through

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

until we reach a value less than or equal to K. Similarly for $K > \mathbf{L}[\lceil pn \rceil]$ we will step to the right by $\sqrt{n}$ until we reach a value in $\mathbf{L}$ that is greater than K. We are now within $\sqrt{n}$ positions of K. Assume (for now) that it takes a constant number of comparisons to bracket K within a sublist of size $\sqrt{n}$. We then take this sublist and repeat the process recursively. That is, at the next level we compute an interpolation to start somewhere in the subarray. We then step to the left or right (as appropriate) by steps of size $\sqrt{\sqrt{n}}$.

What is the cost for QBS? 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 that we are looking for. Because $n = 2^{\log n}$ 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.

Say that the number of comparisons needed is i, in which case the cost is i (since we have to do i comparisons). If $\mathbf{P}_i$ is the probability of needing exactly i probes, then

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

$$= 1\mathbf{P}_1 + 2\mathbf{P}_2 + 3\mathbf{P}_3 + \cdots + \sqrt{n}\mathbf{P}_{\sqrt{n}}$$

We now show that this is the same as

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

$$\begin{aligned} &= 1 + (1 - \mathbf{P}_1) + (1 - \mathbf{P}_1 - \mathbf{P}_2) + \cdots + \mathbf{P}_{\sqrt{n}} \\ &= (\mathbf{P}_1 + ... + \mathbf{P}_{\sqrt{n}}) + (\mathbf{P}_2 + ... + \mathbf{P}_{\sqrt{n}}) + \\ &\qquad (\mathbf{P}_3 + ... + \mathbf{P}_{\sqrt{n}}) + \cdots \\ &= 1\mathbf{P}_1 + 2\mathbf{P}_2 + 3\mathbf{P}_3 + \cdots + \sqrt{n}\mathbf{P}_{\sqrt{n}} \end{aligned}$$

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. Čebyšev's inequality states that $\mathbf{P}$(need exactly i probes), or $\mathbf{P}_i$, is

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

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

$$2 + \sum_{i=3}^{\sqrt{n}} \frac{1}{4(i-2)^2} < 2 + \frac{1}{4} \sum_{i=1}^{\infty} \frac{1}{i^2} = 2 + \frac{1}{4}\frac{\pi}{6} \approx 2.4112$$

Is QBS better than binary search? Theoretically yes, because $\mathrm{O}(\log \log n)$ grows slower than $\mathrm{O}(\log n)$. 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, it is exponentially faster! But even so, for practical input sizes, the absolute cost difference is fairly small. Thus, the constant factors might play a role. First we compare $\lg \lg n$ to $\lg n$.

n	$\lg n$	$\lg \lg n$	Factor Difference
16	4	2	2
256	8	3	2.7
2^{16}	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, because the constant factors in the two algorithms might differ by more than the log of the log of the input size.

For our two algorithms, let us look further and check the actual number of 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	$\lg n - 1$	$2.4 \lg \lg n$	Factor Difference
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 raw comparisons. Binary search is inherently much simpler than QBS, because binary search only needs to calculate 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 QBS are even higher.

Not only are the constant factors worse on average, but QBS 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 QBS 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 ordering of lists is most commonly done by key value, this is not the only viable option. Another approach to organizing lists to speed search is to order the

records by expected frequency of access. While the benefits might not be as great as when organized by key value, the cost to organize (at least approximately) by frequency of access can be much cheaper, and thus can speed up sequential search in some situations.

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

$$\overline{C}_n = 1p_0 + 2p_1 + ... + np_{n-1}.$$

In other words, the cost to access the record in **L**[0] is 1 (because one key value is looked at), and the probability of this occurring is p_0. The cost to access the record in **L**[1] is 2 (because we must look at the first and the second records' key values), with probability p_1, and so on. For n records, assuming that all searches are for records that actually exist, the probabilities p_0 through p_{n-1} 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. We saw in Section 9.1 the more general case where we must consider the probability (labeled p_n) that the search key does not match that for any record in the array. In that case, in accordance with our general formula, we get

$$(1-p_n)\frac{n+1}{2} + p_n n = \frac{n+1-np_n n - p_n + 2p_n}{2} = \frac{n+1+p_0(n-1)}{2}.$$

Thus, $\frac{n+1}{2} \leq \overline{C}_n \leq n$, depending on the value of p_0.

A geometric probability distribution can yield quite different results.

Example 9.2 Calculate the expected cost for searching a list ordered by frequency when the probabilities are defined as

$$p_i = \begin{cases} 1/2^i & \text{if } 0 \leq i \leq n-2 \\ 1/2^n & \text{if } i = n-1. \end{cases}$$

Then,

$$\overline{C}_n \approx \sum_{i=0}^{n-1}(i+1)/2^{i+1} = \sum_{i=1}^{n}(i/2^i) \approx 2.$$

For this example, the expected number of accesses is a constant. This is because the probability for accessing the first record is high (one half), the second is much lower (one quarter) but still much higher than for the third record, 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 data access pattern has its own values. However, behavior of this nature occurs surprisingly often in practice (which explains the success of caching techniques widely used by web browsers for speeding access to web pages, and 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 considerable improvements to search performance from a list ordered by frequency of access over standard sequential search in an unordered list.

Example 9.3 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 $1/(i\mathcal{H}_n)$ (see Exercise 9.4). 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 10-15% of the records in a list ordered by frequency.

This is potentially a useful observation that typical "real-life" distributions of record accesses, if the records were ordered by frequency, would require that we visit on average only 10-15% of the list when doing sequential search. This means that if we had an application that used sequential search, and we wanted to make it go a bit faster (by a constant amount), we could do so without a major rewrite to the system to implement something like a search tree. But that is only true if there is an easy way to (at least approximately) order the records by frequency.

In most applications, we have no means of knowing in advance the frequencies of access for the data records. 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 (in most database systems, this is to be expected). **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, because 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. 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. Thus, 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 (slightly) changing position. Move-to-front's cost is bounded in the sense that it requires at most twice the num-

ber of 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. Move-to-front does poorly when the records are processed in sequential order, especially if that sequential order is then repeated multiple times.

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, because neither record will ever make progress toward the front. However, such pathological cases are unusual in practice. A variation on transpose would be to move the accessed record forward in the list by some fixed number of steps.

Example 9.4 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:

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

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. 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.

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. 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.

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 array 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.

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 replaces 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 search methods 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 values fall within a limited 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 array. 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 bit-wise operations. The union of sets A and B is the bit-wise OR function (whose symbol is `|` in Java). The intersection of sets A and B is the bit-wise AND function (whose symbol is `&` in Java). 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 Java using the expression `A&~B` (`~` is the symbol for bit-wise negation). For larger sets that do not fit into a single computer word, the equivalent operations can be performed in turn on the series of 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 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

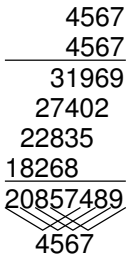

Figure 9.2 An illustration of the mid-square method, showing the details of long multiplication in the process of squaring the value 4567. The bottom of the figure indicates which digits of the answer are most influenced by each digit of the operands.

(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(String x, int M) {
  char ch[];
  ch = x.toCharArray();
  int xlength = x.length();

  int i, sum;
  for (sum=0, i=0; i<x.length(); i++)
    sum += ch[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.

```
long sfold(String s, int M) {

  int intLength = s.length() / 4;
  long sum = 0;
  for (int j = 0; j < intLength; j++) {
    char c[] = s.substring(j*4,(j*4)+4).toCharArray();
    long mult = 1;
    for (int k = 0; k < c.length; k++) {
      sum += c[k] * mult;
      mult *= 256;
    }
  }

  char c[] = s.substring(intLength * 4).toCharArray();
  long mult = 1;
  for (int k = 0; k < c.length; k++) {
    sum += c[k] * mult;
    mult *= 256;
  }

  return(Math.abs(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 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,

[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. Here, **sfold** takes the absolute value of **sum** before applying the modulus operator.

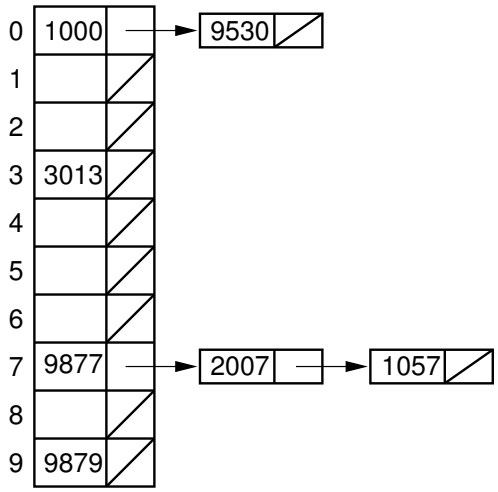

Figure 9.3 An illustration of open hashing for seven numbers stored in a ten-slot hash table using the hash function $\mathbf{h}(K) = K \bmod 10$. The numbers are inserted in the order 9877, 2007, 1000, 9530, 3013, 9879, and 1057. Two of the values hash to slot 0, one value hashes to slot 2, three of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

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.

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.

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

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 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

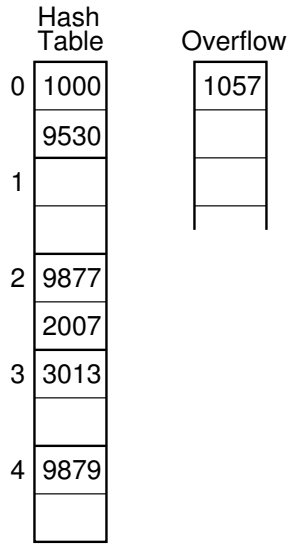

Figure 9.4 An illustration of bucket hashing for seven numbers stored in a five-bucket hash table using the hash function $\mathbf{h}(K) = K \bmod 5$. Each bucket contains two slots. The numbers are inserted in the order 9877, 2007, 1000, 9530, 3013, 9879, and 1057. Two of the values hash to bucket 0, 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.

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 9.4 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,

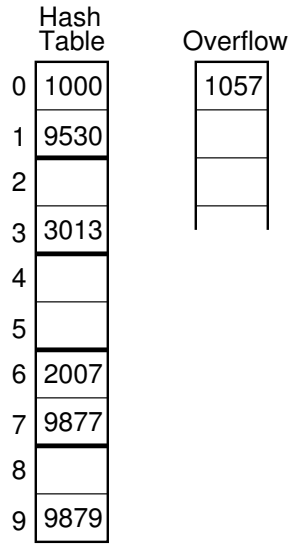

Figure 9.5 An variant of bucket hashing for seven numbers stored in a 10-slot hash table using the hash function $\mathbf{h}(K) = K \bmod 10$. Each bucket contains two slots. The numbers are inserted in the order 9877, 2007, 1000, 9530, 3013, 9879, and 1057. Value 9877 first hashes to slot 7, so when 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.

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 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.

```
/** Insert record r with key k into HT */
void hashInsert(Key k, E r) {
  int home;                          // Home position for r
  int pos = home = h(k);             // Initial position
  for (int i=1; HT[pos] != null; i++) {
    pos = (home + p(k, i)) % M;      // Next pobe slot
    assert HT[pos].key().compareTo(k) != 0 :
           "Duplicates not allowed";
  }
  HT[pos] = new KVpair<Key,E>(k, r); // Insert R
}
```

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

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 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 Java 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

```
/** Search in hash table HT for the record with key k */
E hashSearch(Key k) {
  int home;               // Home position for k
  int pos = home = h(k); // Initial position
  for (int i = 1; (HT[pos] != null) &&
                  (HT[pos].key().compareTo(k) != 0); i++)
    pos = (home + p(k, i)) % M;     // Next probe position
  if (HT[pos] == null) return null; // Key not in hash table
  else return HT[pos].value();      // Found it
}
```

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

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

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

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

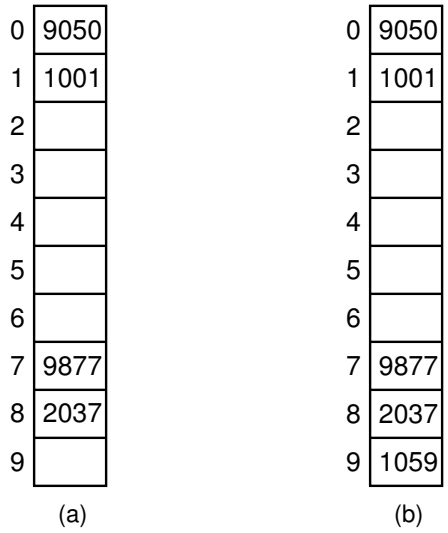

Figure 9.8 Example of problems with linear probing. (a) Four values are inserted in the order 1001, 9050, 9877, and 2037 using hash function $\mathbf{h}(K) = K \bmod 10$. (b) The value 1059 is added to the hash table.

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 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 `Perm`$[1] = 5$, `Perm`$[2] = 2$, and `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.

```
/** Dictionary implemented using hashing. */
class HashDictionary<Key extends Comparable<? super Key>, E>
                    implements Dictionary<Key, E> {
  private static final int defaultSize = 10;
  private HashTable<Key,E> T; // The hash table
  private int count;          // # of records now in table
  private int maxsize;        // Maximum size of dictionary

  HashDictionary() { this(defaultSize); }
  HashDictionary(int sz) {
    T = new HashTable<Key,E>(sz);
    count = 0;
    maxsize = sz;
  }

  public void clear() {   /** Reinitialize */
    T = new HashTable<Key,E>(maxsize);
    count = 0;
  }

  public void insert(Key k, E e) {  /** Insert an element */
    assert count < maxsize : "Hash table is full";
    T.hashInsert(k, e);
    count++;
  }

  public E remove(Key k) {  /** Remove an element */
    E temp = T.hashRemove(k);
    if (temp != null) count--;
    return temp;
  }

  public E removeAny() {  /** Remove some element. */
    if (count != 0) {
      count--;
      return T.hashRemoveAny();
    }
    else return null;
  }

  /** Find a record with key value "k" */
  public E find(Key k) { return T.hashSearch(k); }

  /** Return number of values in the hash table */
  public int size() { return count; }
}
```

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.

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 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

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

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

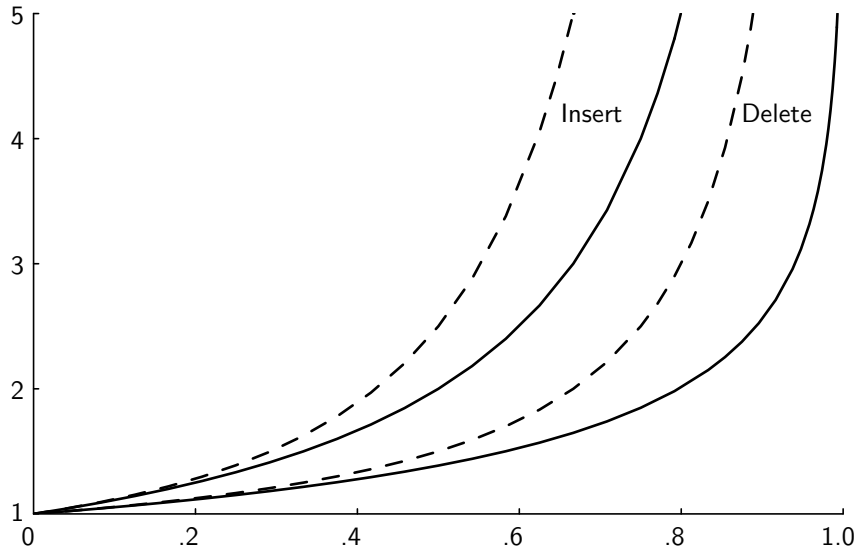

Figure 9.10 Growth of expected record accesses with α. The horizontal axis is the value for α, the vertical axis is the expected number of accesses to the hash table. Solid lines show the cost for "random" probing (a theoretical lower bound on the cost), while dashed lines show the cost for linear probing (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).

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 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 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 should look similar to Quicksort.

9.4 Example 9.9.3 discusses a distribution where the relative frequencies of the records match the harmonic series. That is, for every occurrence of the first record, the second record will appear half as often, the third will appear one third as often, the fourth one quarter as often, and so on. The actual probability for the ith record was defined to be $1/(i\mathcal{H}_n)$. Explain why this is correct.

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 frequency 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 $h(\mathbf{k}) = \mathbf{k} \bmod 7$ 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 $h(\mathbf{k}) = \mathbf{k} \bmod 10$ 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 $h(\mathbf{k}) = \mathbf{k} \bmod 10$ 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`** from Section 9.4.1 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, Rev$(37) = 73$; 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 4.4. 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 4.4 by means of a hash table with linear probing as the collision resolution policy. You might wish to begin with the code of Figure 9.9. Using empirical simulation, determine the cost of insert and delete as α grows (i.e., reconstruct the dashed lines of Figure 9.10). 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?

10

Indexing

Many large-scale computing applications are centered around data sets that are too large to fit into main memory. The classic example is a large database of records with multiple search keys, requiring the ability to insert, delete, and search for records. Hashing provides outstanding performance for such situations, but only in the limited case in which all searches are of the form "find the record with key value K." Many applications require more general search capabilities. One example is a range query search for all records whose key lies within some range. Other queries might involve visiting all records in order of their key value, or finding the record with the greatest key value. Hash tables are not organized to support any of these queries efficiently.

This chapter introduces file structures used to organize a large collection of records stored on disk. Such file structures support efficient insertion, deletion, and search operations, for exact-match queries, range queries, and largest/smallest key value searches.

Before discussing such file structures, we must become familiar with some basic file-processing terminology. An **entry-sequenced file** stores records in the order that they were added to the file. Entry-sequenced files are the disk-based equivalent to an unsorted list and so do not support efficient search. The natural solution is to sort the records by order of the search key. However, a typical database, such as a collection of employee or customer records maintained by a business, might contain multiple search keys. To answer a question about a particular customer might require a search on the name of the customer. Businesses often wish to sort and output the records by zip code order for a bulk mailing. Government paperwork might require the ability to search by Social Security number. Thus, there might not be a single "correct" order in which to store the records.

Indexing is the process of associating a key with the location of a corresponding data record. Section 8.5 discussed the concept of a key sort, in which an **index file** is created whose records consist of key/pointer pairs. Here, each key is associated with a pointer to a complete record in the main database file. The index file

could be sorted or organized using a tree structure, thereby imposing a logical order on the records without physically rearranging them. One database might have several associated index files, each supporting efficient access through a different key field.

Each record of a database normally has a unique identifier, called the **primary key**. For example, the primary key for a set of personnel records might be the Social Security number or ID number for the individual. Unfortunately, the ID number is generally an inconvenient value on which to perform a search because the searcher is unlikely to know it. Instead, the searcher might know the desired employee's name. Alternatively, the searcher might be interested in finding all employees whose salary is in a certain range. If these are typical search requests to the database, then the name and salary fields deserve separate indices. However, key values in the name and salary indices are not likely to be unique.

A key field such as salary, where a particular key value might be duplicated in multiple records, is called a **secondary key**. Most searches are performed using a secondary key. The secondary key index (or more simply, **secondary index**) will associate a secondary key value with the primary key of each record having that secondary key value. At this point, the full database might be searched directly for the record with that primary key, or there might be a primary key index (or **primary index**) that relates each primary key value with a pointer to the actual record on disk. In the latter case, only the primary index provides the location of the actual record on disk, while the secondary indices refer to the primary index.

Indexing is an important technique for organizing large databases, and many indexing methods have been developed. Direct access through hashing is discussed in Section 9.4. A simple list sorted by key value can also serve as an index to the record file. Indexing disk files by sorted lists are discussed in the following section. Unfortunately, a sorted list does not perform well for insert and delete operations.

A third approach to indexing is the tree index. Trees are typically used to organize large databases that must support record insertion, deletion, and key range searches. Section 10.2 briefly describes ISAM, a tentative step toward solving the problem of storing a large database that must support insertion and deletion of records. Its shortcomings help to illustrate the value of tree indexing techniques. Section 10.3 introduces the basic issues related to tree indexing. Section 10.4 introduces the 2-3 tree, a balanced tree structure that is a simple form of the B-tree covered in Section 10.5. B-trees are the most widely used indexing method for large disk-based databases, and for implementing file systems. Since they have such great practical importance, many variations have been invented. Section 10.5 begins with a discussion of the variant normally referred to simply as a "B-tree." Section 10.5.1 presents the most widely implemented variant, the B^+-tree.

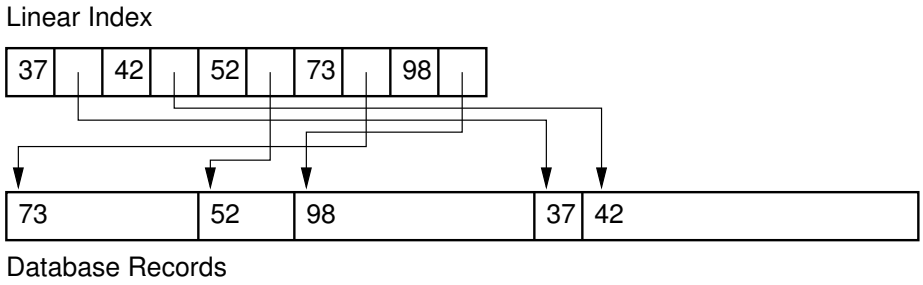

Figure 10.1 Linear indexing for variable-length records. Each record in the index file is of fixed length and contains a pointer to the beginning of the corresponding record in the database file.

10.1 Linear Indexing

A **linear index** is an index file organized as a sequence of key/pointer pairs where the keys are in sorted order and the pointers either (1) point to the position of the complete record on disk, (2) point to the position of the primary key in the primary index, or (3) are actually the value of the primary key. Depending on its size, a linear index might be stored in main memory or on disk. A linear index provides a number of advantages. It provides convenient access to variable-length database records, because each entry in the index file contains a fixed-length key field and a fixed-length pointer to the beginning of a (variable-length) record as shown in Figure 10.1. A linear index also allows for efficient search and random access to database records, because it is amenable to binary search.

If the database contains enough records, the linear index might be too large to store in main memory. This makes binary search of the index more expensive because many disk accesses would typically be required by the search process. One solution to this problem is to store a second-level linear index in main memory that indicates which disk block in the index file stores a desired key. For example, the linear index on disk might reside in a series of 1024-byte blocks. If each key/pointer pair in the linear index requires 8 bytes (a 4-byte key and a 4-byte pointer), then 128 key/pointer pairs are stored per block. The second-level index, stored in main memory, consists of a simple table storing the value of the key in the first position of each block in the linear index file. This arrangement is shown in Figure 10.2. If the linear index requires 1024 disk blocks (1MB), the second-level index contains only 1024 entries, one per disk block. To find which disk block contains a desired search key value, first search through the 1024-entry table to find the greatest value less than or equal to the search key. This directs the search to the proper block in the index file, which is then read into memory. At this point, a binary search within this block will produce a pointer to the actual record in the database. Because the

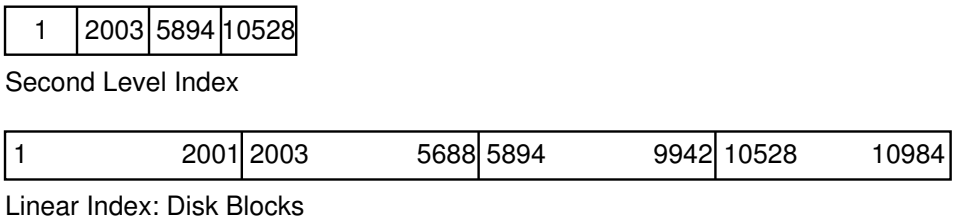

Figure 10.2 A simple two-level linear index. The linear index is stored on disk. The smaller, second-level index is stored in main memory. Each element in the second-level index stores the first key value in the corresponding disk block of the index file. In this example, the first disk block of the linear index stores keys in the range 1 to 2001, and the second disk block stores keys in the range 2003 to 5688. Thus, the first entry of the second-level index is key value 1 (the first key in the first block of the linear index), while the second entry of the second-level index is key value 2003.

second-level index is stored in main memory, accessing a record by this method requires two disk reads: one from the index file and one from the database file for the actual record.

Every time a record is inserted to or deleted from the database, all associated secondary indices must be updated. Updates to a linear index are expensive, because the entire contents of the array might be shifted. Another problem is that multiple records with the same secondary key each duplicate that key value within the index. When the secondary key field has many duplicates, such as when it has a limited range (e.g., a field to indicate job category from among a small number of possible job categories), this duplication might waste considerable space.

One improvement on the simple sorted array is a two-dimensional array where each row corresponds to a secondary key value. A row contains the primary keys whose records have the indicated secondary key value. Figure 10.3 illustrates this approach. Now there is no duplication of secondary key values, possibly yielding a considerable space savings. The cost of insertion and deletion is reduced, because only one row of the table need be adjusted. Note that a new row is added to the array when a new secondary key value is added. This might lead to moving many records, but this will happen infrequently in applications suited to using this arrangement.

A drawback to this approach is that the array must be of fixed size, which imposes an upper limit on the number of primary keys that might be associated with a particular secondary key. Furthermore, those secondary keys with fewer records than the width of the array will waste the remainder of their row. A better approach is to have a one-dimensional array of secondary key values, where each secondary key is associated with a linked list. This works well if the index is stored in main memory, but not so well when it is stored on disk because the linked list for a given key might be scattered across several disk blocks.

Jones	AA10	AB12	AB39	FF37
Smith	AX33	AX35	ZX45	
Zukowski	ZQ99			

Figure 10.3 A two-dimensional linear index. Each row lists the primary keys associated with a particular secondary key value. In this example, the secondary key is a name. The primary key is a unique four-character code.

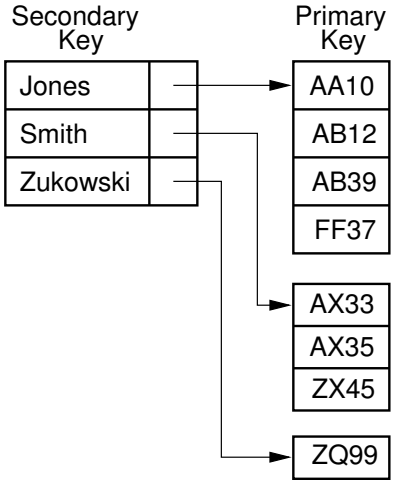

Figure 10.4 Illustration of an inverted list. Each secondary key value is stored in the secondary key list. Each secondary key value on the list has a pointer to a list of the primary keys whose associated records have that secondary key value.

Consider a large database of employee records. If the primary key is the employee's ID number and the secondary key is the employee's name, then each record in the name index associates a name with one or more ID numbers. The ID number index in turn associates an ID number with a unique pointer to the full record on disk. The secondary key index in such an organization is also known as an **inverted list** or **inverted file**. It is inverted in that searches work backwards from the secondary key to the primary key to the actual data record. It is called a list because each secondary key value has (conceptually) a list of primary keys associated with it. Figure 10.4 illustrates this arrangement. Here, we have last names as the secondary key. The primary key is a four-character unique identifier.

Figure 10.5 shows a better approach to storing inverted lists. An array of secondary key values is shown as before. Associated with each secondary key is a pointer to an array of primary keys. The primary key array uses a linked-list implementation. This approach combines the storage for all of the secondary key lists into a single array, probably saving space. Each record in this array consists of a

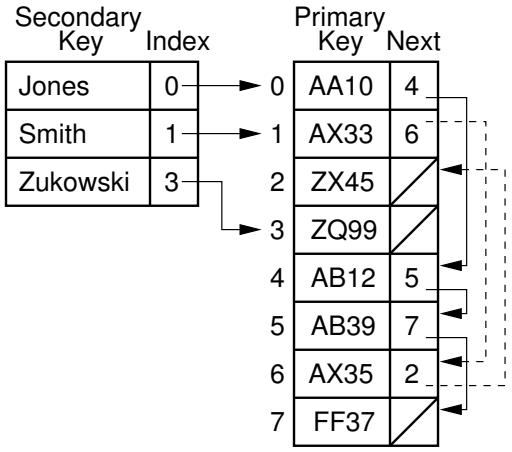

Figure 10.5 An inverted list implemented as an array of secondary keys and combined lists of primary keys. Each record in the secondary key array contains a pointer to a record in the primary key array. The **`next`** field of the primary key array indicates the next record with that secondary key value.

primary key value and a pointer to the next element on the list. It is easy to insert and delete secondary keys from this array, making this a good implementation for disk-based inverted files.

10.2 ISAM

How do we handle large databases that require frequent update? The main problem with the linear index is that it is a single, large array that does not adjust well to updates because a single update can require changing the position of every key in the index. Inverted lists reduce this problem, but they are only suitable for secondary key indices with many fewer secondary key values than records. The linear index would perform well as a primary key index if it could somehow be broken into pieces such that individual updates affect only a part of the index. This concept will be pursued throughout the rest of this chapter, eventually culminating in the B^+-tree, the most widely used indexing method today. But first, we begin by studying ISAM, an early attempt to solve the problem of large databases requiring frequent update. Its weaknesses help to illustrate why the B^+-tree works so well.

Before the invention of effective tree indexing schemes, a variety of disk-based indexing methods were in use. All were rather cumbersome, largely because no adequate method for handling updates was known. Typically, updates would cause the index to degrade in performance. ISAM is one example of such an index and was widely used by IBM prior to adoption of the B-tree.

ISAM is based on a modified form of the linear index, as illustrated by Figure 10.6. Records are stored in sorted order by primary key. The disk file is divided

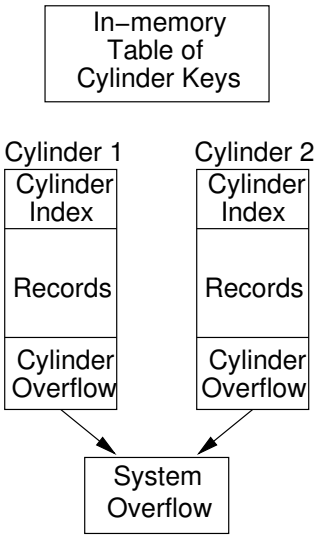

Figure 10.6 Illustration of the ISAM indexing system.

among a number of cylinders on disk.[1] Each cylinder holds a section of the list in sorted order. Initially, each cylinder is not filled to capacity, and the extra space is set aside in the **cylinder overflow**. In memory is a table listing the lowest key value stored in each cylinder of the file. Each cylinder contains a table listing the lowest key value for each block in that cylinder, called the **cylinder index**. When new records are inserted, they are placed in the correct cylinder's overflow area (in effect, a cylinder acts as a bucket). If a cylinder's overflow area fills completely, then a system-wide overflow area is used. Search proceeds by determining the proper cylinder from the system-wide table kept in main memory. The cylinder's block table is brought in from disk and consulted to determine the correct block. If the record is found in that block, then the search is complete. Otherwise, the cylinder's overflow area is searched. If that is full, and the record is not found, then the system-wide overflow is searched.

After initial construction of the database, so long as no new records are inserted or deleted, access is efficient because it requires only two disk fetches. The first disk fetch recovers the block table for the desired cylinder. The second disk fetch recovers the block that, under good conditions, contains the record. After many inserts, the overflow list becomes too long, resulting in significant search time as the cylinder overflow area fills up. Under extreme conditions, many searches might eventually lead to the system overflow area. The "solution" to this problem is to periodically reorganize the entire database. This means re-balancing the records

[1] Recall from Section 8.2.1 that a cylinder is all of the tracks readable from a particular placement of the heads on the multiple platters of a disk drive.

among the cylinders, sorting the records within each cylinder, and updating both the system index table and the within-cylinder block table. Such reorganization was typical of database systems during the 1960s and would normally be done each night or weekly.

10.3 Tree-based Indexing

Linear indexing is efficient when the database is static, that is, when records are inserted and deleted rarely or never. ISAM is adequate for a limited number of updates, but not for frequent changes. Because it has essentially two levels of indexing, ISAM will also break down for a truly large database where the number of cylinders is too great for the top-level index to fit in main memory.

In their most general form, database applications have the following characteristics:

1. Large sets of records that are frequently updated.
2. Search is by one or a combination of several keys.
3. Key range queries or min/max queries are used.

For such databases, a better organization must be found. One approach would be to use the binary search tree (BST) to store primary and secondary key indices. BSTs can store duplicate key values, they provide efficient insertion and deletion as well as efficient search, and they can perform efficient range queries. When there is enough main memory, the BST is a viable option for implementing both primary and secondary key indices.

Unfortunately, the BST can become unbalanced. Even under relatively good conditions, the depth of leaf nodes can easily vary by a factor of two. This might not be a significant concern when the tree is stored in main memory because the time required is still $\Theta(\log n)$ for search and update. When the tree is stored on disk, however, the depth of nodes in the tree becomes crucial. Every time a BST node *B* is visited, it is necessary to visit all nodes along the path from the root to *B*. Each node on this path must be retrieved from disk. Each disk access returns a block of information. If a node is on the same block as its parent, then the cost to find that node is trivial once its parent is in main memory. Thus, it is desirable to keep subtrees together on the same block. Unfortunately, many times a node is not on the same block as its parent. Thus, each access to a BST node could potentially require that another block to be read from disk. Using a buffer pool to store multiple blocks in memory can mitigate disk access problems if BST accesses display good locality of reference. But a buffer pool cannot eliminate disk I/O entirely. The problem becomes greater if the BST is unbalanced, because nodes deep in the tree have the potential of causing many disk blocks to be read. Thus, there are two significant issues that must be addressed to have efficient search from a disk-based

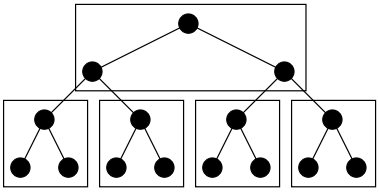

Figure 10.7 Breaking the BST into blocks. The BST is divided among disk blocks, each with space for three nodes. The path from the root to any leaf is contained on two blocks.

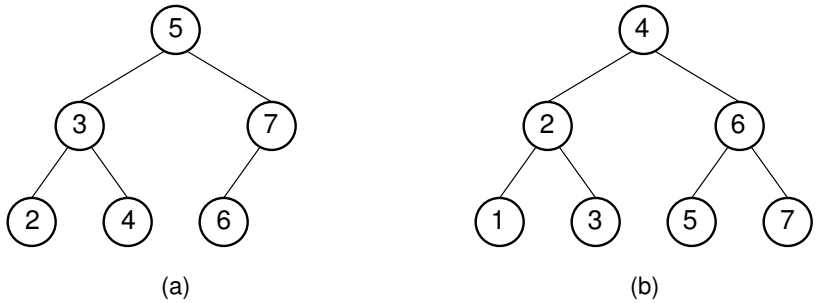

Figure 10.8 An attempt to re-balance a BST after insertion can be expensive. (a) A BST with six nodes in the shape of a complete binary tree. (b) A node with value 1 is inserted into the BST of (a). To maintain both the complete binary tree shape and the BST property, a major reorganization of the tree is required.

BST. The first is how to keep the tree balanced. The second is how to arrange the nodes on blocks so as to keep the number of blocks encountered on any path from the root to the leaves at a minimum.

We could select a scheme for balancing the BST and allocating BST nodes to blocks in a way that minimizes disk I/O, as illustrated by Figure 10.7. However, maintaining such a scheme in the face of insertions and deletions is difficult. In particular, the tree should remain balanced when an update takes place, but doing so might require much reorganization. Each update should affect only a few blocks, or its cost will be too high. As you can see from Figure 10.8, adopting a rule such as requiring the BST to be complete can cause a great deal of rearranging of data within the tree.

We can solve these problems by selecting another tree structure that automatically remains balanced after updates, and which is amenable to storing in blocks. There are a number of balanced tree data structures, and there are also techniques for keeping BSTs balanced. Examples are the AVL and splay trees discussed in Section 13.2. As an alternative, Section 10.4 presents the **2-3 tree**, which has the property that its leaves are always at the same level. The main reason for discussing the 2-3 tree here in preference to the other balanced search trees is that it naturally

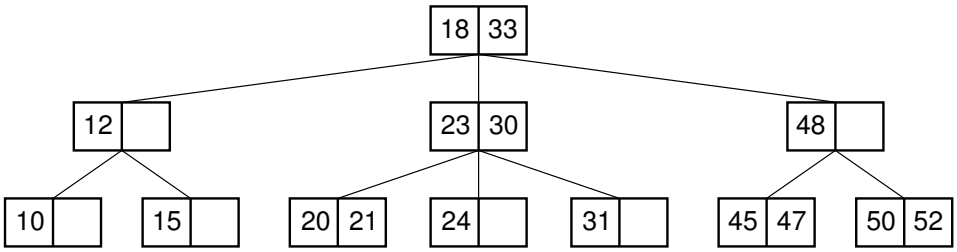

Figure 10.9 A 2-3 tree.

leads to the B-tree of Section 10.5, which is by far the most widely used indexing method today.

10.4 2-3 Trees

This section presents a data structure called the 2-3 tree. The 2-3 tree is not a binary tree, but instead its shape obeys the following definition:

1. A node contains one or two keys.
2. Every internal node has either two children (if it contains one key) or three children (if it contains two keys). Hence the name.
3. All leaves are at the same level in the tree, so the tree is always height balanced.

In addition to these shape properties, the 2-3 tree has a search tree property analogous to that of a BST. For every node, the values of all descendants in the left subtree are less than the value of the first key, while values in the center subtree are greater than or equal to the value of the first key. If there is a right subtree (equivalently, if the node stores two keys), then the values of all descendants in the center subtree are less than the value of the second key, while values in the right subtree are greater than or equal to the value of the second key. To maintain these shape and search properties requires that special action be taken when nodes are inserted and deleted. The 2-3 tree has the advantage over the BST in that the 2-3 tree can be kept height balanced at relatively low cost.

Figure 10.9 illustrates the 2-3 tree. Nodes are shown as rectangular boxes with two key fields. (These nodes actually would contain complete records or pointers to complete records, but the figures will show only the keys.) Internal nodes with only two children have an empty right key field. Leaf nodes might contain either one or two keys. Figure 10.10 is an implementation for the 2-3 tree node.

Note that this sample declaration does not distinguish between leaf and internal nodes and so is space inefficient, because leaf nodes store three pointers each. The techniques of Section 5.3.1 can be applied here to implement separate internal and leaf node types.

```
/** 2-3 tree node implementation */
class TTNode<Key extends Comparable<? super Key>,E> {
  private E lval;        // The left record
  private Key lkey;        // The node's left key
  private E rval;        // The right record
  private Key rkey;        // The node's right key
  private TTNode<Key,E> left;   // Pointer to left child
  private TTNode<Key,E> center; // Pointer to middle child
  private TTNode<Key,E> right;  // Pointer to right child

  public TTNode() { center = left = right = null; }
  public TTNode(Key lk, E lv, Key rk, E rv,
                TTNode<Key,E> p1, TTNode<Key,E> p2,
                TTNode<Key,E> p3) {
    lkey = lk; rkey = rk;
    lval = lv; rval = rv;
    left = p1; center = p2; right = p3;
  }

  public boolean isLeaf() { return left == null; }
  public TTNode<Key,E> lchild() { return left; }
  public TTNode<Key,E> rchild() { return right; }
  public TTNode<Key,E> cchild() { return center; }
  public Key lkey() { return lkey; }  // Left key
  public E lval() { return lval; }  // Left value
  public Key rkey() { return rkey; }  // Right key
  public E rval() { return rval; }  // Right value
  public void setLeft(Key k, E e) { lkey = k; lval = e; }
  public void setRight(Key k, E e) { rkey = k; rval = e; }
  public void setLeftChild(TTNode<Key,E> it) { left = it; }
  public void setCenterChild(TTNode<Key,E> it)
    { center = it; }
  public void setRightChild(TTNode<Key,E> it)
    { right = it; }
```

Figure 10.10 The 2-3 tree node implementation.

From the defining rules for 2-3 trees we can derive relationships between the number of nodes in the tree and the depth of the tree. A 2-3 tree of height k has at least 2^{k-1} leaves, because if every internal node has two children it degenerates to the shape of a complete binary tree. A 2-3 tree of height k has at most 3^{k-1} leaves, because each internal node can have at most three children.

Searching for a value in a 2-3 tree is similar to searching in a BST. Search begins at the root. If the root does not contain the search key K, then the search progresses to the only subtree that can possibly contain K. The value(s) stored in the root node determine which is the correct subtree. For example, if searching for the value 30 in the tree of Figure 10.9, we begin with the root node. Because 30 is between 18 and 33, it can only be in the middle subtree. Searching the middle child of the root node yields the desired record. If searching for 15, then the first step is

```
private E findhelp(TTNode<Key,E> root, Key k) {
  if (root == null) return null;          // val not found
  if (k.compareTo(root.lkey()) == 0) return root.lval();
  if ((root.rkey() != null) && (k.compareTo(root.rkey())
       == 0))
    return root.rval();
  if (k.compareTo(root.lkey()) < 0)       // Search left
    return findhelp(root.lchild(), k);
  else if (root.rkey() == null)           // Search center
    return findhelp(root.cchild(), k);
  else if (k.compareTo(root.rkey()) < 0)  // Search center
    return findhelp(root.cchild(), k);
  else return findhelp(root.rchild(), k); // Search right
}
```

Figure 10.11 Implementation for the 2-3 tree search method.

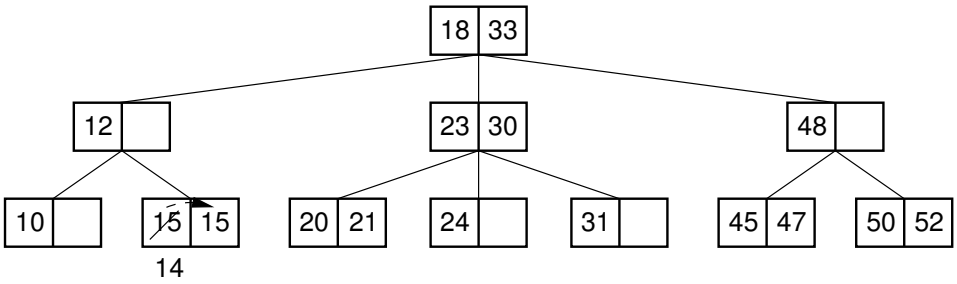

Figure 10.12 Simple insert into the 2-3 tree of Figure 10.9. The value 14 is inserted into the tree at the leaf node containing 15. Because there is room in the node for a second key, it is simply added to the left position with 15 moved to the right position.

again to search the root node. Because 15 is less than 18, the first (left) branch is taken. At the next level, we take the second branch to the leaf node containing 15. If the search key were 16, then upon encountering the leaf containing 15 we would find that the search key is not in the tree. Figure 10.11 is an implementation for the 2-3 tree search method.

Insertion into a 2-3 tree is similar to insertion into a BST to the extent that the new record is placed in the appropriate leaf node. Unlike BST insertion, a new child is not created to hold the record being inserted, that is, the 2-3 tree does not grow downward. The first step is to find the leaf node that would contain the record if it were in the tree. If this leaf node contains only one value, then the new record can be added to that node with no further modification to the tree, as illustrated in Figure 10.12. In this example, a record with key value 14 is inserted. Searching from the root, we come to the leaf node that stores 15. We add 14 as the left value (pushing the record with key 15 to the rightmost position).

If we insert the new record into a leaf node L that already contains two records, then more space must be created. Consider the two records of node L and the

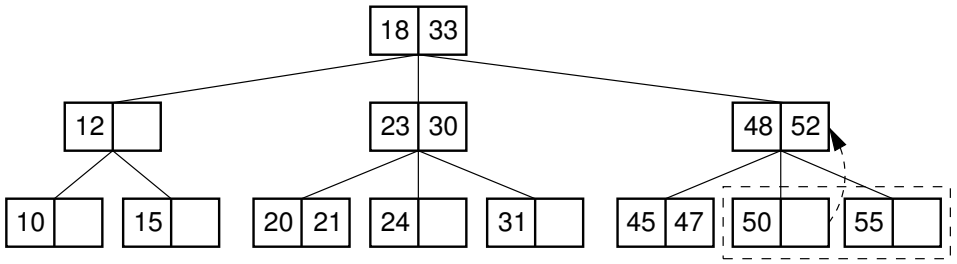

Figure 10.13 A simple node-splitting insert for a 2-3 tree. The value 55 is added to the 2-3 tree of Figure 10.9. This makes the node containing values 50 and 52 split, promoting value 52 to the parent node.

record to be inserted without further concern for which two were already in L and which is the new record. The first step is to split L into two nodes. Thus, a new node — call it L' — must be created from free store. L receives the record with the least of the three key values. L' receives the greatest of the three. The record with the middle of the three key value is passed up to the parent node along with a pointer to L'. This is called a **promotion**. The promoted key is then inserted into the parent. If the parent currently contains only one record (and thus has only two children), then the promoted record and the pointer to L' are simply added to the parent node. If the parent is full, then the split-and-promote process is repeated. Figure 10.13 illustrates a simple promotion. Figure 10.14 illustrates what happens when promotions require the root to split, adding a new level to the tree. In either case, all leaf nodes continue to have equal depth. Figures 10.15 and 10.16 present an implementation for the insertion process.

Note that **`inserthelp`** of Figure 10.15 takes three parameters. The first is a pointer to the root of the current subtree, named **`rt`**. The second is the key for the record to be inserted, and the third is the record itself. The return value for **`inserthelp`** is a pointer to a 2-3 tree node. If **`rt`** is unchanged, then a pointer to **`rt`** is returned. If **`rt`** is changed (due to the insertion causing the node to split), then a pointer to the new subtree root is returned, with the key value and record value in the leftmost fields, and a pointer to the (single) subtree in the center pointer field. This revised node will then be added to the parent, as illustrated in Figure 10.14.

When deleting a record from the 2-3 tree, there are three cases to consider. The simplest occurs when the record is to be removed from a leaf node containing two records. In this case, the record is simply removed, and no other nodes are affected. The second case occurs when the only record in a leaf node is to be removed. The third case occurs when a record is to be removed from an internal node. In both the second and the third cases, the deleted record is replaced with another that can take its place while maintaining the correct order, similar to removing a node from a BST. If the tree is sparse enough, there is no such record available that will allow all nodes to still maintain at least one record. In this situation, sibling nodes are

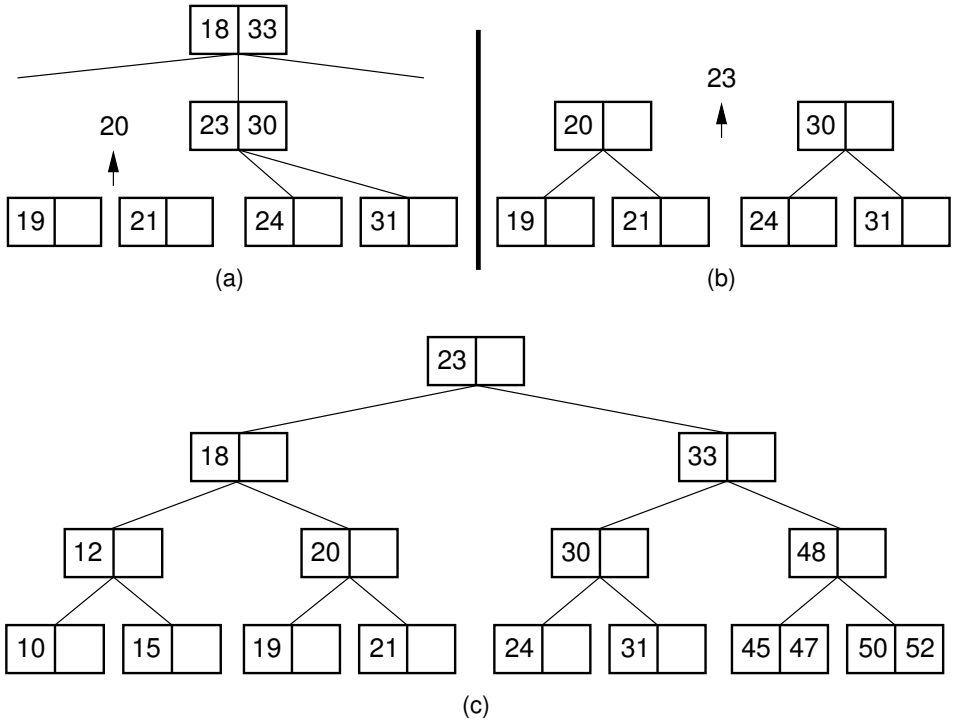

Figure 10.14 Example of inserting a record that causes the 2-3 tree root to split. (a) The value 19 is added to the 2-3 tree of Figure 10.9. This causes the node containing 20 and 21 to split, promoting 20. (b) This in turn causes the internal node containing 23 and 30 to split, promoting 23. (c) Finally, the root node splits, promoting 23 to become the left record in the new root. The result is that the tree becomes one level higher.

merged together. The delete operation for the 2-3 tree is excessively complex and will not be described further. Instead, a complete discussion of deletion will be postponed until the next section, where it can be generalized for a particular variant of the B-tree.

The 2-3 tree insert and delete routines do not add new nodes at the bottom of the tree. Instead they cause leaf nodes to split or merge, possibly causing a ripple effect moving up the tree to the root. If necessary the root will split, causing a new root node to be created and making the tree one level deeper. On deletion, if the last two children of the root merge, then the root node is removed and the tree will lose a level. In either case, all leaf nodes are always at the same level. When all leaf nodes are at the same level, we say that a tree is **height balanced**. Because the 2-3 tree is height balanced, and every internal node has at least two children, we know that the maximum depth of the tree is $\log n$. Thus, all 2-3 tree insert, find, and delete operations require $\Theta(\log n)$ time.

```
private TTNode<Key,E> inserthelp(TTNode<Key,E> rt,
                                 Key k, E e) {
  TTNode<Key,E> retval;
  if (rt == null) // Empty tree: create a leaf node for root
    return new TTNode<Key,E>(k, e, null, null,
                             null, null, null);
  if (rt.isLeaf()) // At leaf node: insert here
    return rt.add(new TTNode<Key,E>(k, e, null, null,
                                    null, null, null));
  // Add to internal node
  if (k.compareTo(rt.lkey()) < 0) { // Insert left
    retval = inserthelp(rt.lchild(), k, e);
    if (retval == rt.lchild()) return rt;
    else return rt.add(retval);
  }
  else if((rt.rkey() == null) ||
          (k.compareTo(rt.rkey()) < 0)) {
    retval = inserthelp(rt.cchild(), k, e);
    if (retval == rt.cchild()) return rt;
    else return rt.add(retval);
  }
  else { // Insert right
    retval = inserthelp(rt.rchild(), k, e);
    if (retval == rt.rchild()) return rt;
    else return rt.add(retval);
  }
}
```

Figure 10.15 The 2-3 tree insert routine.

10.5 B-Trees

This section presents the B-tree. B-trees are usually attributed to R. Bayer and E. McCreight who described the B-tree in a 1972 paper. By 1979, B-trees had replaced virtually all large-file access methods other than hashing. B-trees, or some variant of B-trees, are *the* standard file organization for applications requiring insertion, deletion, and key range searches. They are used to implement most modern file systems. B-trees address effectively all of the major problems encountered when implementing disk-based search trees:

1. B-trees are always height balanced, with all leaf nodes at the same level.
2. Update and search operations affect only a few disk blocks. The fewer the number of disk blocks affected, the less disk I/O is required.
3. B-trees keep related records (that is, records with similar key values) on the same disk block, which helps to minimize disk I/O on searches due to locality of reference.
4. B-trees guarantee that every node in the tree will be full at least to a certain minimum percentage. This improves space efficiency while reducing the typical number of disk fetches necessary during a search or update operation.

```
/** Add a new key/value pair to the node. There might be a
    subtree associated with the record being added. This
    information comes in the form of a 2-3 tree node with
    one key and a (possibly null) subtree through the
    center pointer field. */
  public TTNode<Key,E> add(TTNode<Key,E> it) {
    if (rkey == null) { // Only one key, add here
      if (lkey.compareTo(it.lkey()) < 0) {
        rkey = it.lkey(); rval = it.lval();
        right = center; center = it.cchild();
      }
      else {
        rkey = lkey; rval = lval; right = center;
        lkey = it.lkey(); lval = it.lval();
        center = it.cchild();
      }
      return this;
    }
    else if (lkey.compareTo(it.lkey()) >= 0) { // Add left
      center = new TTNode<Key,E>(rkey, rval, null, null,
                                 center, right, null);
      rkey = null; rval = null; right = null;
      it.setLeftChild(left); left = it;
      return this;
    }
    else if (rkey.compareTo(it.lkey()) < 0) { // Add center
      it.setCenterChild(new TTNode<Key,E>(rkey, rval, null,
                            null, it.cchild(), right, null));
      it.setLeftChild(this);
      rkey = null; rval = null; right = null;
      return it;
    }
    else { // Add right
      TTNode<Key,E> N1 = new TTNode<Key,E>(rkey, rval, null,
                                      null, this, it, null);
      it.setLeftChild(right);
      right = null; rkey = null; rval = null;
      return N1;
    }
  }
```

Figure 10.16 The 2-3 tree node **add** method.

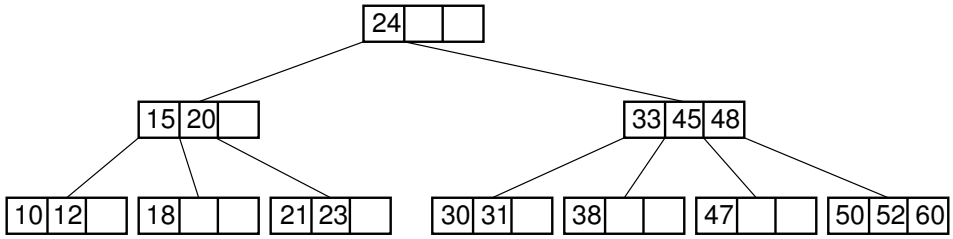

Figure 10.17 A B-tree of order four.

A B-tree of order m is defined to have the following shape properties:

- The root is either a leaf or has at least two children.
- Each internal node, except for the root, has between $\lceil m/2 \rceil$ and m children.
- All leaves are at the same level in the tree, so the tree is always height balanced.

The B-tree is a generalization of the 2-3 tree. Put another way, a 2-3 tree is a B-tree of order three. Normally, the size of a node in the B-tree is chosen to fill a disk block. A B-tree node implementation typically allows 100 or more children. Thus, a B-tree node is equivalent to a disk block, and a "pointer" value stored in the tree is actually the number of the block containing the child node (usually interpreted as an offset from the beginning of the corresponding disk file). In a typical application, the B-tree's access to the disk file will be managed using a buffer pool and a block-replacement scheme such as LRU (see Section 8.3).

Figure 10.17 shows a B-tree of order four. Each node contains up to three keys, and internal nodes have up to four children.

Search in a B-tree is a generalization of search in a 2-3 tree. It is an alternating two-step process, beginning with the root node of the B-tree.

1. Perform a binary search on the records in the current node. If a record with the search key is found, then return that record. If the current node is a leaf node and the key is not found, then report an unsuccessful search.
2. Otherwise, follow the proper branch and repeat the process.

For example, consider a search for the record with key value 47 in the tree of Figure 10.17. The root node is examined and the second (right) branch taken. After examining the node at level 1, the third branch is taken to the next level to arrive at the leaf node containing a record with key value 47.

B-tree insertion is a generalization of 2-3 tree insertion. The first step is to find the leaf node that should contain the key to be inserted, space permitting. If there is room in this node, then insert the key. If there is not, then split the node into two and promote the middle key to the parent. If the parent becomes full, then it is split in turn, and its middle key promoted.

Note that this insertion process is guaranteed to keep all nodes at least half full. For example, when we attempt to insert into a full internal node of a B-tree of order four, there will now be five children that must be dealt with. The node is split into two nodes containing two keys each, thus retaining the B-tree property. The middle of the five children is promoted to its parent.

10.5.1 B$^+$-Trees

The previous section mentioned that B-trees are universally used to implement large-scale disk-based systems. Actually, the B-tree as described in the previous section is almost never implemented, nor is the 2-3 tree as described in Section 10.4. What is most commonly implemented is a variant of the B-tree, called the B$^+$-tree. When greater efficiency is required, a more complicated variant known as the B*-tree is used.

When data are static, a linear index provides an extremely efficient way to search. The problem is how to handle those pesky inserts and deletes. We could try to keep the core idea of storing a sorted array-based list, but make it more flexible by breaking the list into manageable chunks that are more easily updated. How might we do that? First, we need to decide how big the chunks should be. Since the data are on disk, it seems reasonable to store a chunk that is the size of a disk block, or a small multiple of the disk block size. If the next record to be inserted belongs to a chunk that hasn't filled its block then we can just insert it there. The fact that this might cause other records in that chunk to move a little bit in the array is not important, since this does not cause any extra disk accesses so long as we move data within that chunk. But what if the chunk fills up the entire block that contains it? We could just split it in half. What if we want to delete a record? We could just take the deleted record out of the chunk, but we might not want a lot of near-empty chunks. So we could put adjacent chunks together if they have only a small amount of data between them. Or we could shuffle data between adjacent chunks that together contain more data. The big problem would be how to find the desired chunk when processing a record with a given key. Perhaps some sort of tree-like structure could be used to locate the appropriate chunk. These ideas are exactly what motivate the B$^+$-tree. The B$^+$-tree is essentially a mechanism for managing a sorted array-based list, where the list is broken into chunks.

The most significant difference between the B$^+$-tree and the BST or the standard B-tree is that the B$^+$-tree stores records only at the leaf nodes. Internal nodes store key values, but these are used solely as placeholders to guide the search. This means that internal nodes are significantly different in structure from leaf nodes. Internal nodes store keys to guide the search, associating each key with a pointer to a child B$^+$-tree node. Leaf nodes store actual records, or else keys and pointers to actual records in a separate disk file if the B$^+$-tree is being used purely as an index. Depending on the size of a record as compared to the size of a key, a leaf

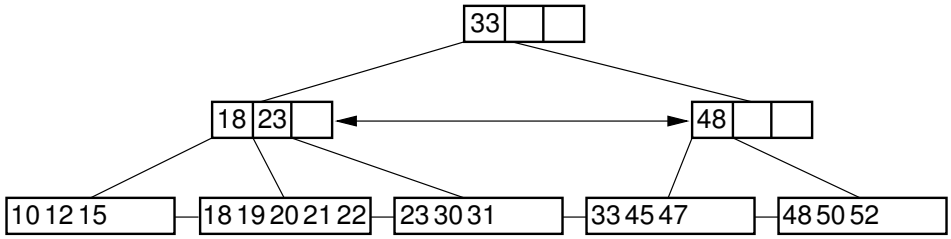

Figure 10.18 Example of a B$^+$-tree of order four. Internal nodes must store between two and four children. For this example, the record size is assumed to be such that leaf nodes store between three and five records.

node in a B$^+$-tree of order m might have enough room to store more or less than m records. The requirement is simply that the leaf nodes store enough records to remain at least half full. The leaf nodes of a B$^+$-tree are normally linked together to form a doubly linked list. Thus, the entire collection of records can be traversed in sorted order by visiting all the leaf nodes on the linked list. Here is a Java-like pseudocode representation for the B$^+$-tree node interface. Leaf node and internal node subclasses would implement this interface.

```
/** Interface for B+ Tree nodes */
public interface BPNode<Key,E> {
  public boolean isLeaf();
  public int numrecs();
  public Key[] keys();
}
```

An important implementation detail to note is that while Figure 10.17 shows internal nodes containing three keys and four pointers, class **BPNode** is slightly different in that it stores key/pointer pairs. Figure 10.17 shows the B$^+$-tree as it is traditionally drawn. To simplify implementation in practice, nodes really do associate a key with each pointer. Each internal node should be assumed to hold in the leftmost position an additional key that is less than or equal to any possible key value in the node's leftmost subtree. B$^+$-tree implementations typically store an additional dummy record in the leftmost leaf node whose key value is less than any legal key value.

B$^+$-trees are exceptionally good for range queries. Once the first record in the range has been found, the rest of the records with keys in the range can be accessed by sequential processing of the remaining records in the first node, and then continuing down the linked list of leaf nodes as far as necessary. Figure 10.18 illustrates the B$^+$-tree.

Search in a B$^+$-tree is nearly identical to search in a regular B-tree, except that the search must always continue to the proper leaf node. Even if the search-key value is found in an internal node, this is only a placeholder and does not provide

```
private E findhelp(BPNode<Key,E> rt, Key k) {
  int currec = binaryle(rt.keys(), rt.numrecs(), k);
  if (rt.isLeaf())
    if ((((BPLeaf<Key,E>)rt).keys())[currec] == k)
      return ((BPLeaf<Key,E>)rt).recs(currec);
    else return null;
  else
    return findhelp(((BPInternal<Key,E>)rt).
                        pointers(currec), k);
}
```

Figure 10.19 Implementation for the B$^+$-tree search method.

access to the actual record. To find a record with key value 33 in the B$^+$-tree of Figure 10.18, search begins at the root. The value 33 stored in the root merely serves as a placeholder, indicating that keys with values greater than or equal to 33 are found in the second subtree. From the second child of the root, the first branch is taken to reach the leaf node containing the actual record (or a pointer to the actual record) with key value 33. Figure 10.19 shows a pseudocode sketch of the B$^+$-tree search algorithm.

B$^+$-tree insertion is similar to B-tree insertion. First, the leaf L that should contain the record is found. If L is not full, then the new record is added, and no other B$^+$-tree nodes are affected. If L is already full, split it in two (dividing the records evenly among the two nodes) and promote a copy of the least-valued key in the newly formed right node. As with the 2-3 tree, promotion might cause the parent to split in turn, perhaps eventually leading to splitting the root and causing the B$^+$-tree to gain a new level. B$^+$-tree insertion keeps all leaf nodes at equal depth. Figure 10.20 illustrates the insertion process through several examples. Figure 10.21 shows a Java-like pseudocode sketch of the B$^+$-tree insert algorithm.

To delete record R from the B$^+$-tree, first locate the leaf L that contains R. If L is more than half full, then we need only remove R, leaving L still at least half full. This is demonstrated by Figure 10.22.

If deleting a record reduces the number of records in the node below the minimum threshold (called an **underflow**), then we must do something to keep the node sufficiently full. The first choice is to look at the node's adjacent siblings to determine if they have a spare record that can be used to fill the gap. If so, then enough records are transferred from the sibling so that both nodes have about the same number of records. This is done so as to delay as long as possible the next time when a delete causes this node to underflow again. This process might require that the parent node has its placeholder key value revised to reflect the true first key value in each node. Figure 10.23 illustrates the process.

If neither sibling can lend a record to the under-full node (call it N), then N must give its records to a sibling and be removed from the tree. There is certainly room to do this, because the sibling is at most half full (remember that it had no

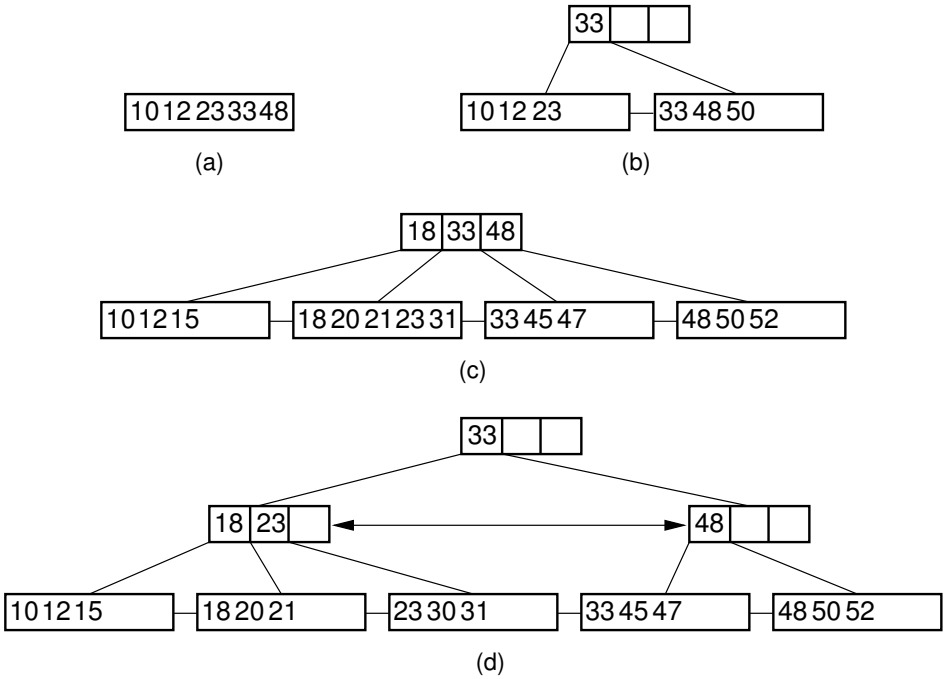

Figure 10.20 Examples of B^+-tree insertion. (a) A B^+-tree containing five records. (b) The result of inserting a record with key value 50 into the tree of (a). The leaf node splits, causing creation of the first internal node. (c) The B^+-tree of (b) after further insertions. (d) The result of inserting a record with key value 30 into the tree of (c). The second leaf node splits, which causes the internal node to split in turn, creating a new root.

```
private BPNode<Key,E> inserthelp(BPNode<Key,E> rt,
                                 Key k, E e) {
  BPNode<Key,E> retval;
  if (rt.isLeaf()) // At leaf node: insert here
    return ((BPLeaf<Key,E>)rt).add(k, e);
  // Add to internal node
  int currec = binaryle(rt.keys(), rt.numrecs(), k);
  BPNode<Key,E> temp = inserthelp(
         ((BPInternal<Key,E>)root).pointers(currec), k, e);
  if (temp != ((BPInternal<Key,E>)rt).pointers(currec))
    return ((BPInternal<Key,E>)rt).
               add((BPInternal<Key,E>)temp);
  else
    return rt;
}
```

Figure 10.21 A Java-like pseudocode sketch of the B^+-tree insert algorithm.

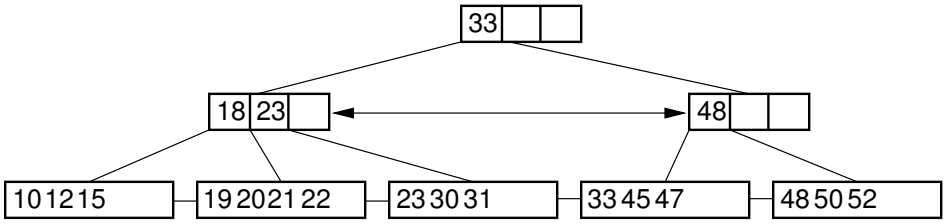

Figure 10.22 Simple deletion from a B$^+$-tree. The record with key value 18 is removed from the tree of Figure 10.18. Note that even though 18 is also a placeholder used to direct search in the parent node, that value need not be removed from internal nodes even if no record in the tree has key value 18. Thus, the leftmost node at level one in this example retains the key with value 18 after the record with key value 18 has been removed from the second leaf node.

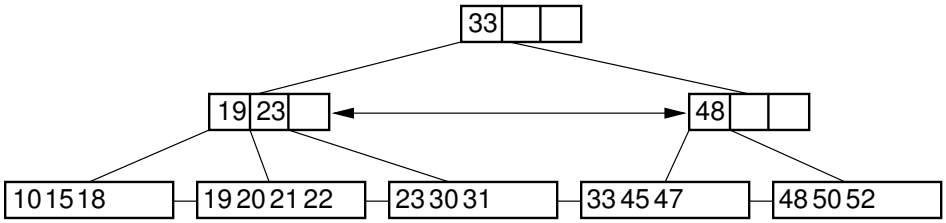

Figure 10.23 Deletion from the B$^+$-tree of Figure 10.18 via borrowing from a sibling. The key with value 12 is deleted from the leftmost leaf, causing the record with key value 18 to shift to the leftmost leaf to take its place. Note that the parent must be updated to properly indicate the key range within the subtrees. In this example, the parent node has its leftmost key value changed to 19.

records to contribute to the current node), and N has become less than half full because it is under-flowing. This merge process combines two subtrees of the parent, which might cause it to underflow in turn. If the last two children of the root merge together, then the tree loses a level. Figure 10.24 illustrates the node-merge deletion process. Figure 10.25 shows Java-like pseudocode for the B$^+$-tree delete algorithm.

The B$^+$-tree requires that all nodes be at least half full (except for the root). Thus, the storage utilization must be at least 50%. This is satisfactory for many implementations, but note that keeping nodes fuller will result both in less space required (because there is less empty space in the disk file) and in more efficient processing (fewer blocks on average will be read into memory because the amount of information in each block is greater). Because B-trees have become so popular, many algorithm designers have tried to improve B-tree performance. One method for doing so is to use the B$^+$-tree variant known as the B*-tree. The B*-tree is identical to the B$^+$-tree, except for the rules used to split and merge nodes. Instead of splitting a node in half when it overflows, the B*-tree gives some records to its

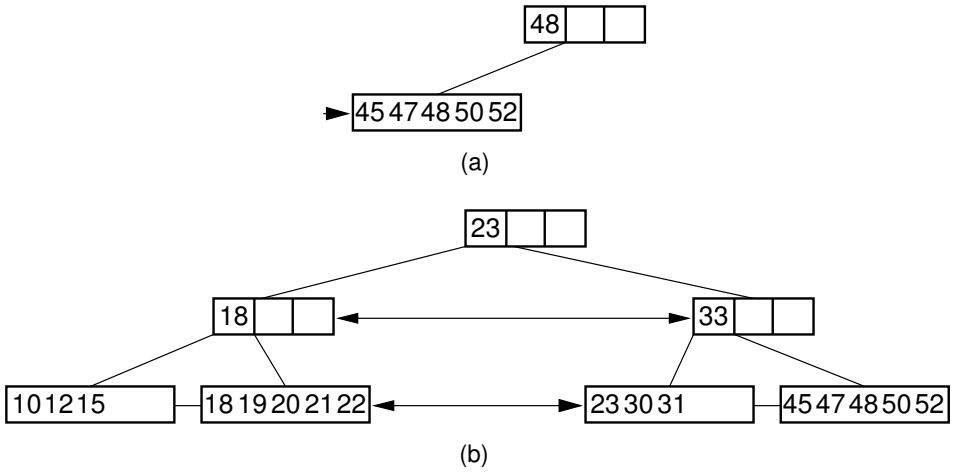

Figure 10.24 Deleting the record with key value 33 from the B$^+$-tree of Figure 10.18 via collapsing siblings. (a) The two leftmost leaf nodes merge together to form a single leaf. Unfortunately, the parent node now has only one child. (b) Because the left subtree has a spare leaf node, that node is passed to the right subtree. The placeholder values of the root and the right internal node are updated to reflect the changes. Value 23 moves to the root, and old root value 33 moves to the rightmost internal node.

```
/** Delete a record with the given key value, and
    return true if the root underflows */
private boolean removehelp(BPNode<Key,E> rt, Key k) {
  int currec = binaryle(rt.keys(), rt.numrecs(), k);
  if (rt.isLeaf())
    if (((BPLeaf<Key,E>)rt).keys()[currec] == k)
      return ((BPLeaf<Key,E>)rt).delete(currec);
    else return false;
  else // Process internal node
    if (removehelp(((BPInternal<Key,E>)rt).pointers(currec),
        k))
      // Child will merge if necessary
      return ((BPInternal<Key,E>)rt).underflow(currec);
    else return false;
}
```

Figure 10.25 Java-like pseudocode for the B$^+$-tree delete algorithm.

neighboring sibling, if possible. If the sibling is also full, then these two nodes split into three. Similarly, when a node underflows, it is combined with its two siblings, and the total reduced to two nodes. Thus, the nodes are always at least two thirds full.[2]

10.5.2 B-Tree Analysis

The asymptotic cost of search, insertion, and deletion of records from B-trees, B^+-trees, and B^*-trees is $\Theta(\log n)$ where n is the total number of records in the tree. However, the base of the log is the (average) branching factor of the tree. Typical database applications use extremely high branching factors, perhaps 100 or more. Thus, in practice the B-tree and its variants are extremely shallow.

As an illustration, consider a B^+-tree of order 100 and leaf nodes that contain up to 100 records. A B^+-tree with height one (that is, just a single leaf node) can have at most 100 records. A B^+-tree with height two (a root internal node whose children are leaves) must have at least 100 records (2 leaves with 50 records each). It has at most 10,000 records (100 leaves with 100 records each). A B^+-tree with height three must have at least 5000 records (two second-level nodes with 50 children containing 50 records each) and at most one million records (100 second-level nodes with 100 full children each). A B^+-tree with height four must have at least 250,000 records and at most 100 million records. Thus, it would require an *extremely* large database to generate a B^+-tree of more than height four.

The B^+-tree split and insert rules guarantee that every node (except perhaps the root) is at least half full. So they are on average about $3/4$ full. But the internal nodes are purely overhead, since the keys stored there are used only by the tree to direct search, rather than store actual data. Does this overhead amount to a significant use of space? No, because once again the high fan-out rate of the tree structure means that the vast majority of nodes are leaf nodes. Recall (from Section 6.4) that a full K-ary tree has approximately $1/K$ of its nodes as internal nodes. This means that while half of a full binary tree's nodes are internal nodes, in a B^+-tree of order 100 probably only about $1/75$ of its nodes are internal nodes. This means that the overhead associated with internal nodes is very low.

We can reduce the number of disk fetches required for the B-tree even more by using the following methods. First, the upper levels of the tree can be stored in main memory at all times. Because the tree branches so quickly, the top two levels (levels 0 and 1) require relatively little space. If the B-tree is only height four, then

[2] This concept can be extended further if higher space utilization is required. However, the update routines become much more complicated. I once worked on a project where we implemented 3-for-4 node split and merge routines. This gave better performance than the 2-for-3 node split and merge routines of the B^*-tree. However, the spitting and merging routines were so complicated that even their author could no longer understand them once they were completed!

at most two disk fetches (internal nodes at level two and leaves at level three) are required to reach the pointer to any given record.

A buffer pool could be used to manage nodes of the B-tree. Several nodes of the tree would typically be in main memory at one time. The most straightforward approach is to use a standard method such as LRU to do node replacement. However, sometimes it might be desirable to "lock" certain nodes such as the root into the buffer pool. In general, if the buffer pool is even of modest size (say at least twice the depth of the tree), no special techniques for node replacement will be required because the upper-level nodes will naturally be accessed frequently.

10.6 Further Reading

For an expanded discussion of the issues touched on in this chapter, see a general file processing text such as *File Structures: A Conceptual Toolkit* by Folk and Zoellick [FZ98]. In particular, Folk and Zoellick provide a good discussion of the relationship between primary and secondary indices. The most thorough discussion on various implementations for the B-tree is the survey article by Comer [Com79]. Also see [Sal88] for further details on implementing B-trees. See Shaffer and Brown [SB93] for a discussion of buffer pool management strategies for B^+-tree-like data structures.

10.7 Exercises

10.1 Assume that a computer system has disk blocks of 1024 bytes, and that you are storing records that have 4-byte keys and 4-byte data fields. The records are sorted and packed sequentially into the disk file.

(a) Assume that a linear index uses 4 bytes to store the key and 4 bytes to store the block ID for the associated records. What is the greatest number of records that can be stored in the file if a linear index of size 256KB is used?

(b) What is the greatest number of records that can be stored in the file if the linear index is also stored on disk (and thus its size is limited only by the second-level index) when using a second-level index of 1024 bytes (i.e., 256 key values) as illustrated by Figure 10.2? Each element of the second-level index references the smallest key value for a disk block of the linear index.

10.2 Assume that a computer system has disk blocks of 4096 bytes, and that you are storing records that have 4-byte keys and 64-byte data fields. The records are sorted and packed sequentially into the disk file.

(a) Assume that a linear index uses 4 bytes to store the key and 4 bytes to store the block ID for the associated records. What is the greatest

number of records that can be stored in the file if a linear index of size 2MB is used?

(b) What is the greatest number of records that can be stored in the file if the linear index is also stored on disk (and thus its size is limited only by the second-level index) when using a second-level index of 4096 bytes (i.e., 1024 key values) as illustrated by Figure 10.2? Each element of the second-level index references the smallest key value for a disk block of the linear index.

10.3 Modify the function **`binary`** of Section 3.5 so as to support variable-length records with fixed-length keys indexed by a simple linear index as illustrated by Figure 10.1.

10.4 Assume that a database stores records consisting of a 2-byte integer key and a variable-length data field consisting of a string. Show the linear index (as illustrated by Figure 10.1) for the following collection of records:

397	Hello world!
82	XYZ
1038	This string is rather long
1037	This is shorter
42	ABC
2222	Hello new world!

10.5 Each of the following series of records consists of a four-digit primary key (with no duplicates) and a four-character secondary key (with many duplicates).

3456	DEER
2398	DEER
2926	DUCK
9737	DEER
7739	GOAT
9279	DUCK
1111	FROG
8133	DEER
7183	DUCK
7186	FROG

(a) Show the inverted list (as illustrated by Figure 10.4) for this collection of records.

(b) Show the improved inverted list (as illustrated by Figure 10.5) for this collection of records.

10.6 Under what conditions will ISAM be more efficient than a B^+-tree implementation?

10.7 Prove that the number of leaf nodes in a 2-3 tree with height k is between 2^{k-1} and 3^{k-1}.

10.8 Show the result of inserting the values 55 and 46 into the 2-3 tree of Figure 10.9.

10.9 You are given a series of records whose keys are letters. The records arrive in the following order: C, S, D, T, A, M, P, I, B, W, N, G, U, R, K, E, H, O, L, J. Show the 2-3 tree that results from inserting these records.

10.10 You are given a series of records whose keys are letters. The records are inserted in the following order: C, S, D, T, A, M, P, I, B, W, N, G, U, R, K, E, H, O, L, J. Show the tree that results from inserting these records when the 2-3 tree is modified to be a 2-3$^+$ tree, that is, the internal nodes act only as placeholders. Assume that the leaf nodes are capable of holding up to two records.

10.11 Show the result of inserting the value 55 into the B-tree of Figure 10.17.

10.12 Show the result of inserting the values 1, 2, 3, 4, 5, and 6 (in that order) into the B$^+$-tree of Figure 10.18.

10.13 Show the result of deleting the values 18, 19, and 20 (in that order) from the B$^+$-tree of Figure 10.24b.

10.14 You are given a series of records whose keys are letters. The records are inserted in the following order: C, S, D, T, A, M, P, I, B, W, N, G, U, R, K, E, H, O, L, J. Show the B$^+$-tree of order four that results from inserting these records. Assume that the leaf nodes are capable of storing up to three records.

10.15 Assume that you have a B$^+$-tree whose internal nodes can store up to 100 children and whose leaf nodes can store up to 15 records. What are the minimum and maximum number of records that can be stored by the B$^+$-tree with heights 1, 2, 3, 4, and 5?

10.16 Assume that you have a B$^+$-tree whose internal nodes can store up to 50 children and whose leaf nodes can store up to 50 records. What are the minimum and maximum number of records that can be stored by the B$^+$-tree with heights 1, 2, 3, 4, and 5?

10.8 Projects

10.1 Implement a two-level linear index for variable-length records as illustrated by Figures 10.1 and 10.2. Assume that disk blocks are 1024 bytes in length. Records in the database file should typically range between 20 and 200 bytes, including a 4-byte key value. Each record of the index file should store a key value and the byte offset in the database file for the first byte of the corresponding record. The top-level index (stored in memory) should be a simple array storing the lowest key value on the corresponding block in the index file.

10.2 Implement the 2-3$^+$ tree, that is, a 2-3 tree where the internal nodes act only as placeholders. Your 2-3$^+$ tree should implement the dictionary interface of Section 4.4.

10.3 Implement the dictionary ADT of Section 4.4 for a large file stored on disk by means of the B$^+$-tree of Section 10.5. Assume that disk blocks are 1024 bytes, and thus both leaf nodes and internal nodes are also 1024 bytes. Records should store a 4-byte (`int`) key value and a 60-byte data field. Internal nodes should store key value/pointer pairs where the "pointer" is actually the block number on disk for the child node. Both internal nodes and leaf nodes will need room to store various information such as a count of the records stored on that node, and a pointer to the next node on that level. Thus, leaf nodes will store 15 records, and internal nodes will have room to store about 120 to 125 children depending on how you implement them. Use a buffer pool (Section 8.3) to manage access to the nodes stored on disk.