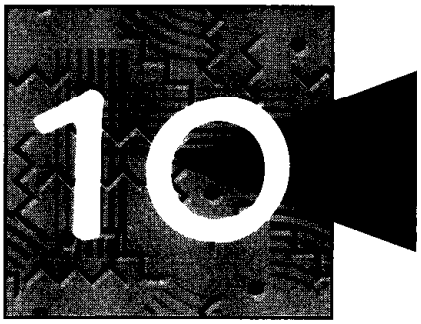


Hashing

The main operation used by the searching methods described in the preceding chapters was comparison of keys. In a sequential search, the table that stores the elements is searched successively to determine which cell of the table to check, and the key comparison determines whether or not an element has been found. In a binary search, the table that stores the elements is divided successively into halves to determine which cell of the table to check, and again, the key comparison determines whether or not an element has been found. Similarly, the decision to continue the search in a binary search tree in a particular direction is accomplished by comparing keys.

A different approach to searching calculates the position of the key in the table based on the value of the key. The value of the key is the only indication of the position. When the key is known, the position in the table can be accessed directly, without making any other preliminary tests, as required in a binary search or when searching a tree. This means that the search time is reduced from $O(n)$, as in a sequential search, or from $O(\lg n)$, as in a binary search, to 1 or at least $O(1)$; regardless of the number of elements being searched, the run time is always the same. But this is just an ideal, and in real applications, this ideal can only be approximated.

We need to find a function h which can transform a particular key K, be it a string, number, record, etc., into an index in the table used for storing items of the same type as K. The function h is called a *hash function*. If h transforms different keys into different numbers, it is called a *perfect hash function*. To create a perfect hash function, which is always the goal, the table has to contain at least the same number of positions as the number of elements being hashed. But the number of elements is not always known ahead of time. For example, a compiler keeps all variables used in a program in a symbol table. Real programs use only a fraction of the vast number of possible variable names, so a table size of 1000 cells is usually adequate.

But even if this table can accommodate all the variables in the program, how can we design a function h which allows the compiler to immediately access the position associated with each variable? All the letters of the variable name can be added together and the sum can be used as an index. In this case, the table needs 3782 cells (for a variable K made out of 31 letters "z," $h(K) = 31 \cdot 122 = 3782$). But even with this size, the function h does not return unique values. For example, $h(\text{"abc"}) = h(\text{"acb"})$. This problem is called *collision* and is discussed later. The worth of a hash function depends on how well it avoids collisions. Avoiding collisions can be achieved by making the function more sophisticated, but this sophistication should not go too far because the computational cost in determining $h(K)$ can be prohibitive, and less sophisticated methods may be faster.

10.1 Hash Functions

The number of hash functions that can be used to assign positions to n items in a table of m positions (for $n \leq m$) is equal to m^n. The number of perfect hash functions is the same as the number of different placements of these items in the table and is equal to $\frac{m!}{(m-n)!}$. For example, for 50 elements and a 100-cell array, there are $100^{50} = 10^{100}$ hash functions out of which "only" 10^{94} (one in a million) are perfect. Most of these functions are too unwieldy for practical applications and cannot be represented by a concise formula. However, even among functions that can be expressed with a formula, the number of possibilities is vast. This section discusses some specific types of hash functions.

10.1.1 Division

A hash function must guarantee that the number it returns is a valid index to one of the table cells. The simplest way to accomplish this is to use division modulo *TSize* = *sizeof(table)*, as in $h(K) = K \bmod TSize$, if K is a number. It is best if *TSize* is a prime number. Otherwise, $h(K) = (K \bmod p) \bmod TSize$ for some prime $p > TSize$ can be used. However, nonprime divisors may work equally well as prime divisors provided they do not have prime factors less than 20 (Lum et al., 1971). The division method is usually the preferred choice for the hash function if very little is known about the keys.

10.1.2 Folding

In this method, the key is divided into several parts (which conveys the true meaning of the word *hash*). These parts are combined or folded together and are often transformed in a certain way to create the target address. There are two types of folding: *shift folding* and *boundary folding*.

The key is divided into several parts and these parts are then processed using a simple operation such as addition to combine them in a certain way. In shift folding, they are put underneath one another and then processed. For example, a social security number (SSN) 123–45–6789 can be divided into three parts, 123, 456, 789, and

then these parts can be added. The resulting number, 1368, can be divided modulo *TSize* or, if the size of the table is 1000, the first three digits can be used for the address. To be sure, the division can be done in many different ways. Another possibility is to divide the same number 123-45-6789 into five parts (say, 12, 34, 56, 78, and 9), add them, and divide the result modulo *TSize*.

With boundary folding, the key is seen as being written on a piece of paper which is folded on the borders between different parts of the key. In this way, every other part will be put in the reverse order. Consider the same three parts of the SSN: 123, 456, and 789. The first part, 123, is taken in the same order, then the piece of paper with the second part is folded underneath it so that 123 is aligned with 654, which is the second part, 456, in reverse order. When the folding continues, 789 is aligned with the two previous parts. The result is 123+654+789=1566.

In both versions, the key is usually divided into even parts of some fixed size plus some remainder and then added. This process is simple and fast, especially when bit patterns are used instead of numerical values. A bit-oriented version of shift folding is obtained by applying the exclusive or operation, ^.

In the case of strings, one approach processes all characters of the string by "xor'ing" them together and using the result for the address. For example, for the string "abcd," *h*("abcd") = "a"^"b"^"c"^"d." However, this simple method results in addresses between the numbers 0 and 127. For better result, chunks of strings are "xor'ed" together rather than single characters. These chunks are composed of the number of characters equal to the number of bytes in an integer. Since an integer in a C++ implementation for the IBM PC computer is two bytes long, *h*("abcd") = "ab" xor "cd" (most likely divided modulo *TSize*). Such a function is used in the case study in this chapter.

10.1.3 Mid-Square Function

In the mid-square method, the key is *squared* and the middle or *mid* part of the result is used as the address. If the key is a string, it has to be preprocessed to produce a number by using, for instance, folding. In a mid-square hash function, the entire key participates in generating the address so that there is a better chance that different addresses are generated for different keys. For example, if the key is 3121, then 3121^2 = 9740641, and for the 1000-cell table, *h*(3121) = 406, which is the middle part of 3121^2. In practice, it is more efficient to choose a power of 2 for the size of the table and extract the middle part of the bit representation of the square of a key. If we assume that the size of the table is 1024, then, in this example, the binary representation of 3121^2 is the bit string 100101001010000101100001 with the middle part shown in italics. This middle part, the binary number 0101000010, is equal to 322. This part can easily be extracted by using a mask and a shift operation.

10.1.4 Extraction

In the extraction method, only a part of the key is used to compute the address. For the social security number 123–45–6789, this method might use the first four digits, 1234, the last four, 6789, the first two combined with the last two, 1289, or some other combination. Each time, only a portion of the key is used. If this portion is carefully

chosen, it can be sufficient for hashing provided the omitted portion distinguishes the keys only in an insignificant way. For example, in some university settings, all international students' ID numbers start with 999. Therefore, the first three digits can be safely omitted in a hash function which uses student IDs for computing table positions. Similarly, the starting digits of the ISBN code are the same for all books published by the same publisher (e.g., 0534 for Brooks/Cole Publishing Company). Therefore, they should be excluded from the computation of addresses if a data table contains only books from one publisher.

10.1.5 Radix Transformation

Using the radix transformation, the key K is transformed into another number base; K is expressed in a numerical system using a different radix. If K is the decimal number 345, then its value in base 9 (nonal) is 423. This value is then divided modulo *TSize*, and the resulting number is used as the address of the location to which K should be hashed. Collisions, however, cannot be avoided. If *TSize* = 100, then although 345 and 245 (decimal) are not hashed to the same location, 345 and 264 are because 264 decimal is 323 in the nonal system, and both 423 and 323 return 23 when divided modulo 100.

10.2 Collision Resolution

Note that straightforward hashing is not without its problems, since for almost all hash functions, more than one key can be assigned to the same position. For example, if the hash function h_1 applied to names returns the ASCII value of the first letter of each name, i.e., $h_1(name) = name[0]$, then all names starting with the same letter are hashed to the same position. This problem can be solved by finding a function which distributes names more uniformly in the table. For example, the function h_2 could add the first two letters, i.e., $h_2(name) = name[0] + name[1]$, which is better than h_1. But even if all the letters are considered, i.e., $h_3(name) = name[0] + \cdots + name[\texttt{strlen}(name) - 1]$, the possibility of hashing different names to the same location still exists. The function h_3 is the best of the three because it distributes the names most uniformly for the three defined functions, but it also tacitly assumes that the size of the table has been increased. If the table had only 26 positions, which is the number of different values returned by h_1, there is no improvement using h_3 instead of h_1. Therefore, one more factor can contribute to avoiding conflicts between hashed keys, namely, the size of the table. Increasing this size may lead to better hashing, but not always! These two factors—hash function and table size—may minimize the number of collisions, but they cannot completely eliminate them. The problem of collision has to be dealt with in a way that always guarantees a solution.

There are scores of strategies that attempt to avoid hashing multiple keys to the same location. Only a handful of these methods are discussed in this chapter.

10.2.1 Open Addressing

In the open addressing method, when a key collides with another key, the collision is resolved by finding an available table entry other than the position (address) to which the colliding key is originally hashed. If position $h(K)$ is occupied, then the positions in the probing sequence

$$norm(h(K) + p(1)), norm(h(K) + p(2)), \ldots, norm(h(K) + p(i)), \ldots$$

are tried until either an available cell is found or the same positions are tried repeatedly or the table is full. Function p is a *probing function*, i is a *probe*, and *norm* is a *normalization function*, most likely, division modulo the size of the table.

The simplest method is *linear probing*, for which $p(i) = i$, and for the ith probe, the position to be tried is $(h(K) + i)$ mod *TSize*. In linear probing, the position in which a key can be stored is found by sequentially searching all positions starting from the position calculated by the hash function until an empty cell is found. If the end of the table is reached and no empty cell has been found, the search is continued from the beginning of the table and stops—in the extreme case—in the cell preceding the one from which the search started. Linear probing, however, has a tendency to create clusters in the table. Figure 10.1 contains an example where a key K_i is hashed to the position i. In Figure 10.1a, three keys—A_5, A_2, and A_3—have been hashed to their home positions. Then B_5 arrives (Figure 10.1b), whose home position is occupied by A_5. Since the next position is available, B_5 is stored there. Next, A_9 is stored with no problem, but B_2 is stored in position 4, two positions from its home address. A large cluster has already been formed. Next, B_9 arrives. Position 9 is not available, and since it is the last cell of the table, the search starts from the beginning of the table, whose first slot can now host B_9. The next key, C_2, ends up in position 7, five positions from its home address.

In this example, the empty cells following clusters have a much greater chance to be filled than other positions. This probability is equal to $(sizeof(cluster) + 1)/TSize$. Other empty cells have only $1/TSize$ chance of being filled. If a cluster is created, it has a tendency to grow, and the larger a cluster becomes, the larger is the likelihood that it will become even larger. This fact undermines the performance of the hash table for storing and retrieving data. The problem at hand is how to avoid cluster buildup. An answer can be found in a more careful choice of the probing function p.

One such choice is a quadratic function so that the resulting formula is

$$p(i) = h(K) + (-1)^{i-1}((i + 1)/2)^2 \text{ for } i = 1, 2, \ldots, TSize - 1$$

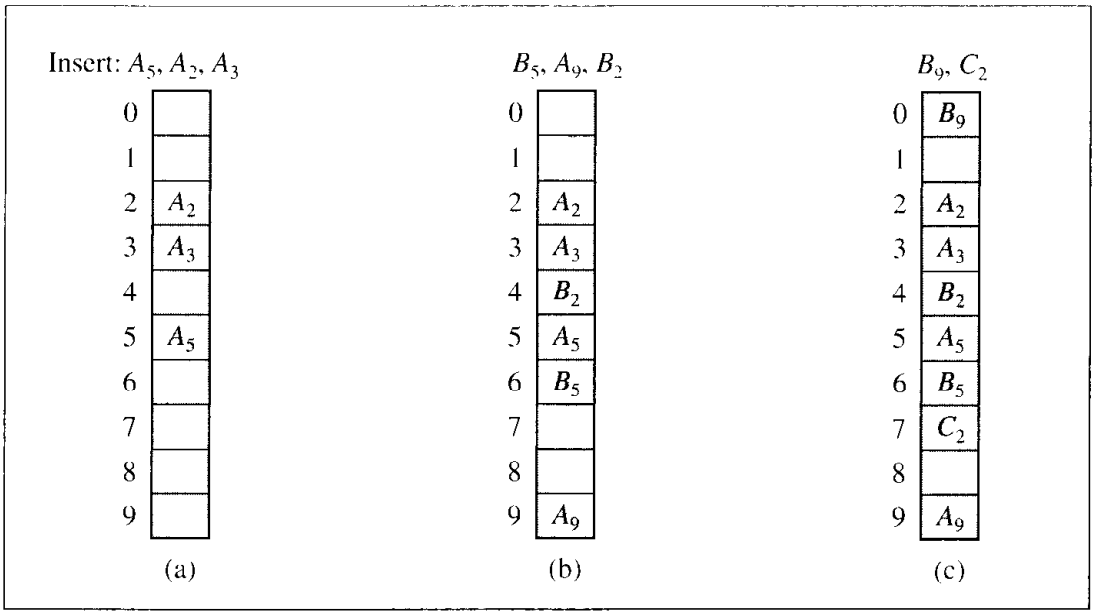
This rather cumbersome formula can be expressed in a simpler form as a sequence of probes:

$$h(K) + i^2, h(K) - i^2 \text{ for } i = 1, 2, \ldots, (TSize - 1)/2$$

Including the first attempt to hash K, this results in the sequence:

$$h(K), h(K) + 1, h(K) - 1, h(K) + 4, h(K) - 4, \ldots, h(K) + (TSize - 1)^2/4,$$
$$h(K) - (TSize - 1)^2/4$$

FIGURE **10.1** Resolving collisions with the linear probing method. Subscripts indicate the home positions of the keys being hashed.



all divided modulo *TSize*. The size of the table should not be an even number, since only the even positions or only the odd positions are tried depending on the value of $h(K)$. Ideally, the table size should be a prime $4j + 3$ of an integer j, which guarantees the inclusion of all positions in the probing sequence (Radke 1970). For example, if $j = 4$, then $TSize = 19$, and assuming that $h(K) = 9$ for some K, the resulting sequence of probes is[1]

$$9, 10, 8, 13, 5, 18, 0, 6, 12, 15, 3, 7, 11, 1, 17, 16, 2, 14, 4$$

The table from Figure 10.1 would have the same keys in a different configuration, as in Figure 10.2. It still takes two probes to locate B_2 in some location, but for C_2, only four probes are required, not five.

Note that the formula determining the sequence of probes chosen for quadratic probing is not $h(K) + i^2$, for $i = 1, 2, \ldots, TSize - 1$, because the first half of the sequence

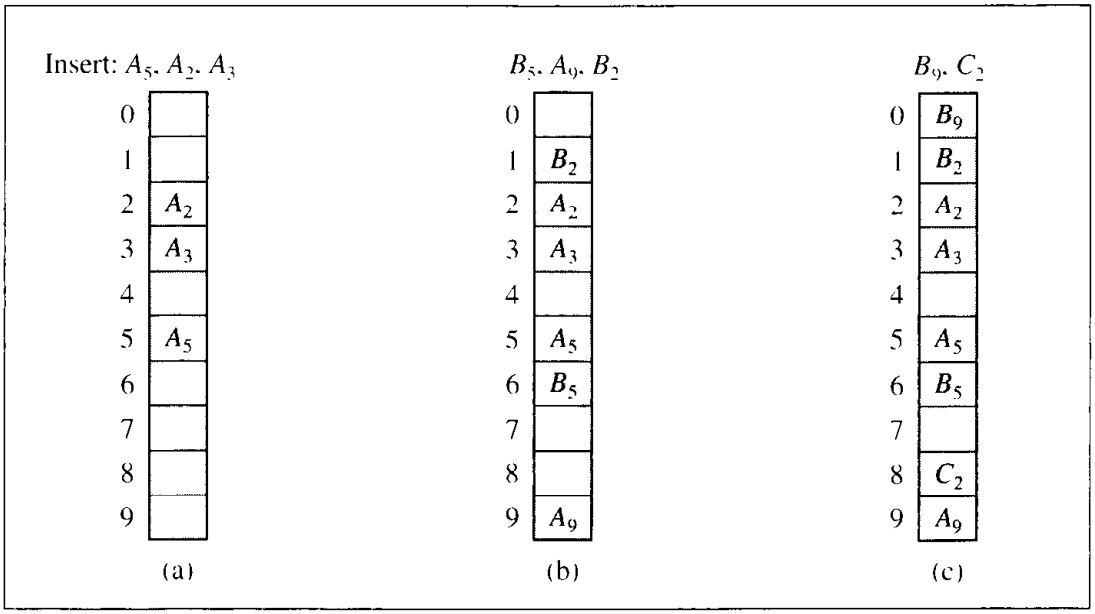
$$h(K) + 1, h(K) + 4, h(K) + 9, \ldots, h(K) + (TSize - 1)^2$$

covers only half of the table, and the second half of the sequence repeats the first half in the reverse order. For example, if $TSize = 19$, and $h(K) = 9$, then the sequence is

$$9, 10, 13, 18, 6, 15, 7, 1, 16, 14, 14, 16, 1, 7, 15, 6, 18, 13, 10$$

[1]Special care should be taken for negative numbers. When implementing these formulas, the operator % means division modulo a modulus. However, this operator is usually implemented as the *remainder* of division. For example, –6 % 23 is equal to –6, and not to 17, as expected. Therefore, when using the operator % for the implementation of division modulo, the modulus (the right operand of %) should be added to the result when the result is negative. Therefore, (–6 % 23) + 23 returns 17.

FIGURE **10.2** Using quadratic probing for collision resolution.



This is not an accident. The probes which render the same address are of the form

$$i = TSize/2 + 1 \text{ and } j = TSize/2 - 1$$

and they are probes for which

$$i^2 \bmod TSize = j^2 \bmod TSize$$

that is,

$$(i^2 - j^2) \bmod TSize$$

In this case,

$$\begin{aligned}(i^2 - j^2) &= (TSize/2 + 1)^2 - (TSize/2 - 1)^2 \\ &= (TSize^2/4 + TSize + 1 - TSize^2/4 + TSize - 1) \\ &= 2\,TSize\end{aligned}$$

and to be sure, $2\,TSize \bmod TSize = 0$.

Although using quadratic search gives much better results than linear probing, the problem of cluster buildup is not avoided altogether, since for keys hashed to the same location, the same probe sequence is used. Such clusters are called *secondary clusters.* These secondary clusters, however, are less harmful than primary clusters.

Another possibility is to have p be a random number generator (Morris 1968), which eliminates the need to take special care about the table size. This approach prevents the formation of secondary clusters, but it causes a problem with repeating the same probing sequence for the same keys. If the random number generator is initialized at the first invocation, then different probing sequences are generated for the

same key K. Consequently, K is hashed more than once to the table and even then it might not be found when searched. Therefore, the random number generator should be initialized to the same seed for the same key before beginning the generation of the probing sequence. This can be achieved in C++ by using the `srand()` function with a parameter that depends on the key; for example, $p(i)$ = `srand(`*sizeof*(K)`)` $\cdot\ i$ or `srand(`K`[0])` $+\ i$. To avoid relying on `srand()`, a random number generator can be written which assures that each invocation generates a unique number between 0 and $TSize - 1$. The following algorithm was developed by Robert Morris for tables with $TSize = 2^n$ for some integer n:

```
generateNumber()
   static int r = 1;
   r = 5*r;
   r = mask out n + 2 low-order bits of r;
   return r/4;
```

The problem of secondary clustering is best addressed with *double hashing*. This method utilizes two hash functions, one for accessing the primary position of a key, h, and a second function, h_p, for resolving conflicts. The probing sequence becomes

$$h(K), h(K) + h_p(K), \ldots, h(K) + i \cdot h_p(K), \ldots$$

(all divided modulo $TSize$). The table size should be a prime number so that each position in the table can be included in the sequence. Experiments indicate that secondary clustering is generally eliminated because the sequence depends on the values of h_p which, in turn, depend on the key. Therefore, if the key K_1 is hashed to the position j, the probing sequence is

$$j, j + h_p(K_1), j + 2 \cdot h_p(K_1), \ldots$$

(all divided modulo $TSize$). If another key K_2 is hashed to $j + h_p(K_1)$, then the next position tried is $j + h_p(K_1) + h_p(K_2)$, not $j + 2 \cdot h_p(K_1)$, which avoids secondary clustering if h_p is carefully chosen. Also, even if K_1 and K_2 are hashed primarily to the same position j, the probing sequences can be different for each. This, however, depends on the choice of the second hash function, h_p, which may render the same sequences for both keys. This is the case for function $h_p(K)$ = `strlen(`K`)`, when both keys are of the same length.

Using two hash functions can be time-consuming, especially for sophisticated functions. Therefore, the second hash function can be defined in terms of the first, as in $h_p(K) = i \cdot h(K) + 1$. The probing sequence for K_1 is

$$j, 2j + 1, 3j + 1, \ldots$$

(modulo $TSize$). If K_2 is hashed to $2j + 1$, then the probing sequence for K_2 is

$$2j + 1, 4j + 3, 6j + 4, \ldots$$

which does not conflict with the former sequence. Thus, it does not lead to cluster buildup.

How efficient are all these methods? Obviously, it depends on the size of table and on the number of elements already in the table. The inefficiency of these methods is especially evident for *unsuccessful searches*, searching for elements not in the table.

FIGURE **10.3** Formulas approximating, for different hashing methods, the average numbers of trials for successful and unsuccessful searches (Knuth 1998).

	linear probing	quadratic search[a]	double hashing
successful search	$\frac{1}{2}\left(1+\frac{1}{1-LF}\right)$	$1-\ln(1-LF)-\frac{LF}{2}$	$\frac{1}{LF}\ln\frac{1}{1-LF}$
unsuccessful search	$\frac{1}{2}\left(1+\frac{1}{(1-LF)^2}\right)$	$\frac{1}{1-LF}-LF-\ln(1-LF)$	$\frac{1}{1-LF}$

Loading factor $LF = \frac{\text{number of elements in the table}}{\text{table size}}$

[a] the formulas given in this column approximate any open addressing method which causes secondary clusters to arise, and quadratic probing is only one of them.

The more elements in the table, the more likely it is that clusters will form (primary or secondary) and the more likely it is that these clusters are large.

Consider the case when linear probing is used for collision resolution. If K is not in the table, then starting from the position $h(K)$, all consecutively occupied cells are checked; the longer the cluster, the longer it takes to determine that K, in fact, is not in the table. In the extreme case, when the table is full, we have to check all the cells starting from $h(K)$ and ending with $(h(K) - 1)$ mod *TSize*. Therefore, the search time increases with the number of elements in the table.

There are formulas which approximate the number of times for successful and unsuccessful searches for different hashing methods. These formulas were developed by Donald Knuth and are considered by Thomas Standish to be "among the prettiest in computer science." Figure 10.3 contains these formulas. Figure 10.4 contains a table showing the number of searches for different percentages of occupied cells. This table indicates that the formulas from Figure 10.3 provide only approximations of the number of searches. This is particularly evident for the higher percentages. For example, if 90% of the cells are occupied, then linear probing requires 50 trials to determine that the key being searched for is not in the table. However, for the full table of 10 cells, this number is 10, not 50.

For the lower percentages, the approximations computed by these formulas are closer to the real values. The table in Figure 10.4 indicates that if the table is 65% full, then linear probing requires, on average, fewer than two trials to find an element in the table. Since this number is usually an acceptable limit for a hash function, linear probing requires 35% of the spaces in the table to be unoccupied to keep performance at an acceptable level. This may be considered too wasteful, especially for very large tables or files. This percentage is lower for a quadratic search (25%) and for double hashing (20%), but it may still be considered large. Double hashing requires one cell out of five to be empty, which is a relatively high fraction. But all these problems can be solved by allowing more than one item to be stored in a given position or in an area associated with one position.

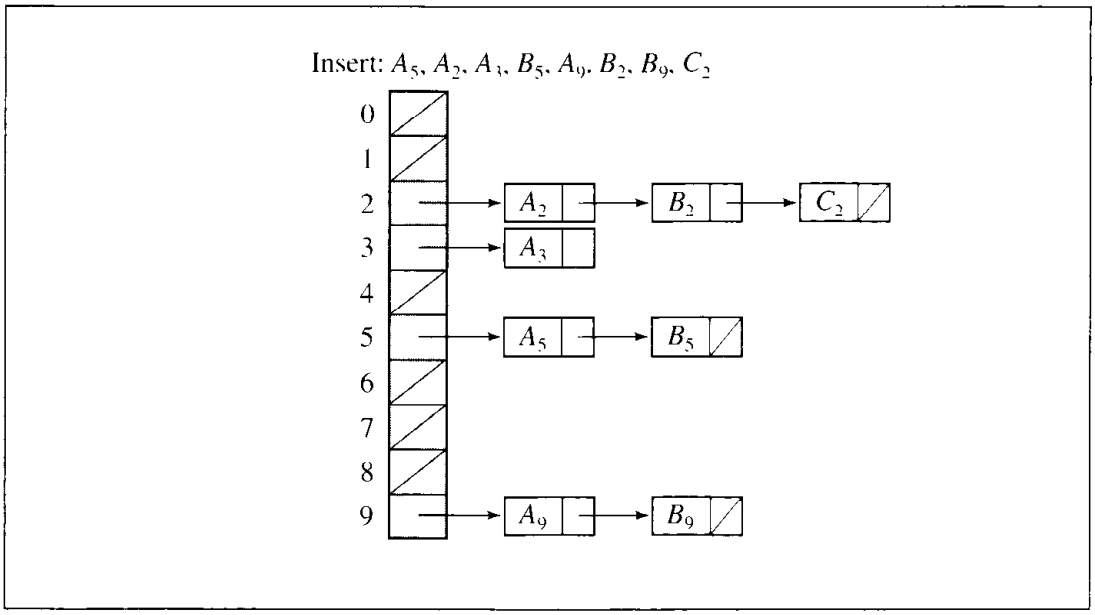
FIGURE **10.4** The average numbers of successful searches and unsuccessful searches for different collision resolution methods.

	linear probing		quadratic search		double hashing	
LF	Successful	unsuccessful	successful	unsuccessful	successful	unsuccessful
0.05	1.0	1.1	1.0	1.1	1.0	1.1
0.10	1.1	1.1	1.1	1.1	1.1	1.1
0.15	1.1	1.2	1.1	1.2	1.1	1.2
0.20	1.1	1.3	1.1	1.3	1.1	1.2
0.25	1.2	1.4	1.2	1.4	1.2	1.3
0.30	1.2	1.5	1.2	1.5	1.2	1.4
0.35	1.3	1.7	1.3	1.6	1.2	1.5
0.40	1.3	1.9	1.3	1.8	1.3	1.7
0.45	1.4	2.2	1.4	2.0	1.3	1.8
0.50	1.5	2.5	1.4	2.2	1.4	2.0
0.55	1.6	3.0	1.5	2.5	1.5	2.2
0.60	1.8	3.6	1.6	2.8	1.5	2.5
0.65	1.9	4.6	1.7	3.3	1.6	2.9
0.70	2.2	6.1	1.9	3.8	1.7	3.3
0.75	2.5	8.5	2.0	4.6	1.8	4.0
0.80	3.0	13.0	2.2	5.8	2.0	5.0
0.85	3.8	22.7	2.5	7.7	2.2	6.7
0.90	5.5	50.5	2.9	11.4	2.6	10.0
0.95	10.5	200.5	3.5	22.0	3.2	20.0

10.2.2 Chaining

Keys do not have to be stored in the table itself. In *chaining*, each position of the table is associated with a linked list or *chain* of structures whose `info` fields store keys or references to keys. This method is called *separate chaining*, and a table of references (pointers) is called a *scatter table*. In this method, the table can never overflow, since the linked lists are only extended upon the arrival of new keys, as illustrated in Figure 10.5. For short linked lists, this is a very fast method, but increasing the length of these lists can significantly degrade retrieval performance. Performance can be improved by

FIGURE **10.5** In chaining, colliding keys are put on the same linked list.



maintaining an order on all these lists so that, for unsuccessful searches, an exhaustive search is not required in most cases or by using self-organizing linked lists (Pagli 1985).

This method requires additional space for maintaining pointers. The table stores only pointers and each node requires one pointer field. Therefore, for n keys, $n + TSize$ pointers are needed, which for large n can be a very demanding requirement.

A version of chaining called *coalesced hashing* (or *coalesced chaining*) combines linear probing with chaining. In this method, the first available position is found for a key colliding with another key, and the index of this position is stored with the key already in the table. In this way, a sequential search down the table can be avoided by directly accessing the next element on the linked list. Each position *pos* of the table stores an object with two members: `info` for a key and `next` with the index of the next key which is hashed to *pos*. Available positions can be marked by, say, –2 in `next`; –1 can be used to indicate the end of a chain. This method requires $TSize \cdot sizeof($`next`$)$ more space for the table in addition to the space required for the keys. This is less than for chaining, but the table size limits the number of keys that can be hashed into the table.

An overflow area known as a *cellar* can be allocated to store keys for which there is no room in the table. This area should be located dynamically if implemented as a list of arrays.

Figure 10.6 illustrates an example where coalesced hashing puts a colliding key in the last position of the table. In Figure 10.6a, no collision occurs. In Figure 10.6b, B_5 is put in the last cell of the table, which is found occupied by A_9 when it arrives. Hence, A_9 is attached to the list accessible from position 9. In Figure 10.6c, two new colliding keys are added to the corresponding lists.

FIGURE **10.6** Coalesced hashing puts a colliding key in the last available position of the table.

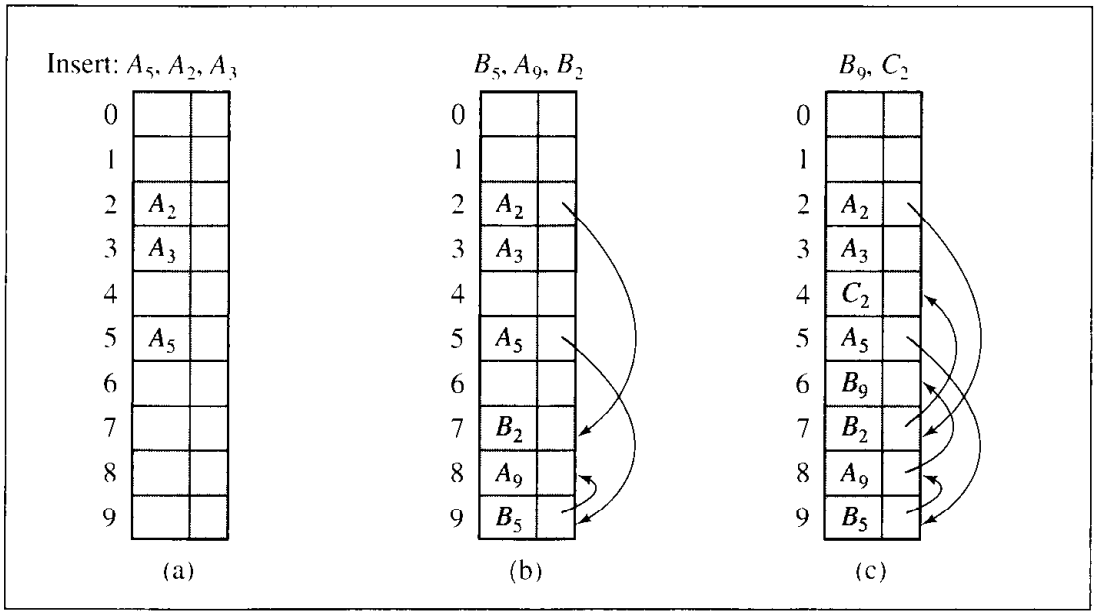


Figure 10.7 illustrates coalesced hashing which uses a cellar. Noncolliding keys are stored in their home positions, as in Figure 10.7a. Colliding keys are put in the last available slot of the cellar and added to the list starting from their home position, as in Figure 10.7b. In Figure 10.7c, the cellar is full, so an available cell is taken from the table when C_2 arrives.

10.2.3 Bucket Addressing

Another solution to the collision problem is to store colliding elements in the same position in the table. This can be achieved by associating a *bucket* with each address. A bucket is a block of space large enough to store multiple items.

By using buckets, the problem of collisions is not totally avoided. If a bucket is already full, then an item hashed to it has to be stored somewhere else. By incorporating the open addressing approach, the colliding item can be stored in the next bucket if it has an available slot when using linear probing, as illustrated in Figure 10.8, or it can be stored in some other bucket when, say, quadratic probing is used.

The colliding items can also be stored in an overflow area. In this case, each bucket includes a field that indicates whether the search should be continued in this area or not. It can be simply a yes/no marker. In conjunction with chaining, this marker can be the number indicating the position in which the beginning of the linked list associated with this bucket can be found in the overflow area (see Figure 10.9).

FIGURE **10.7** Coalesced hashing which uses a cellar.

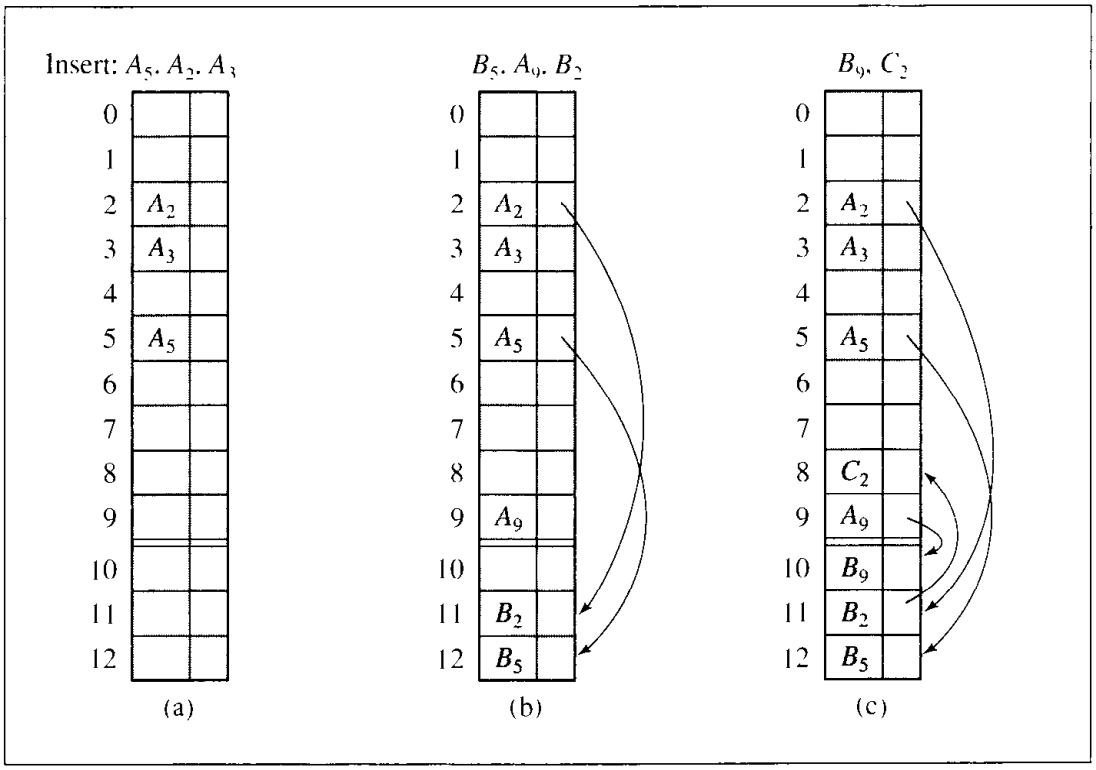


FIGURE **10.8** Collision resolution with buckets and linear probing method.

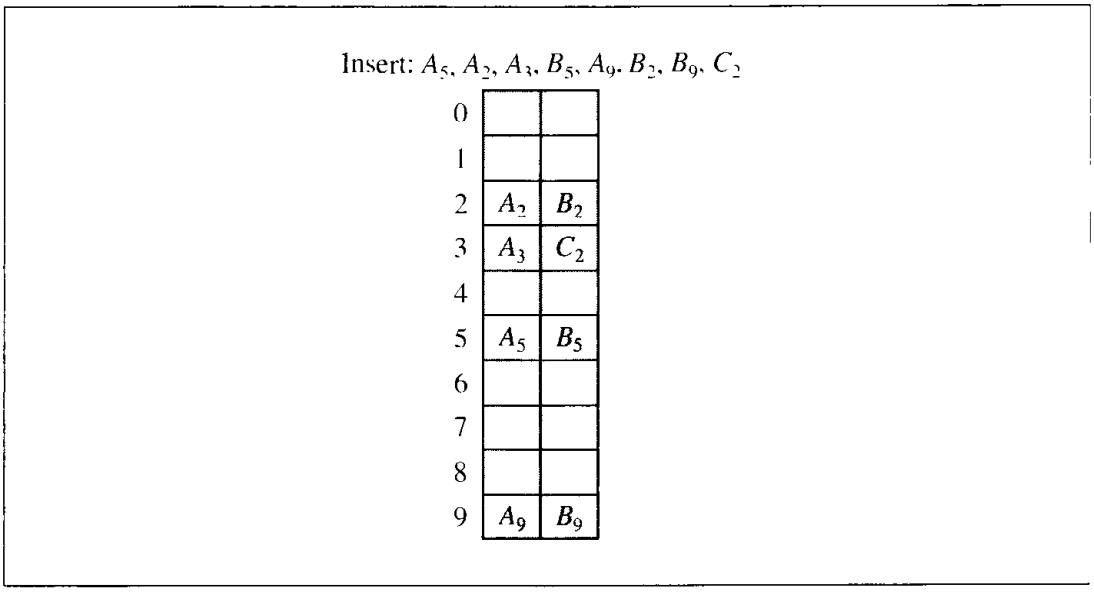
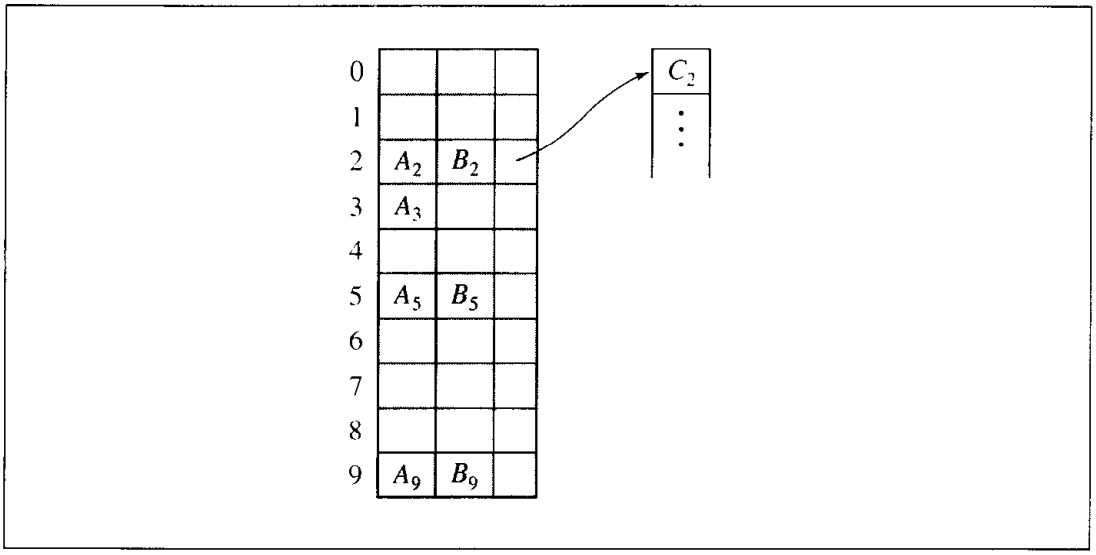


FIGURE 10.9 Collision resolution with buckets and overflow area.



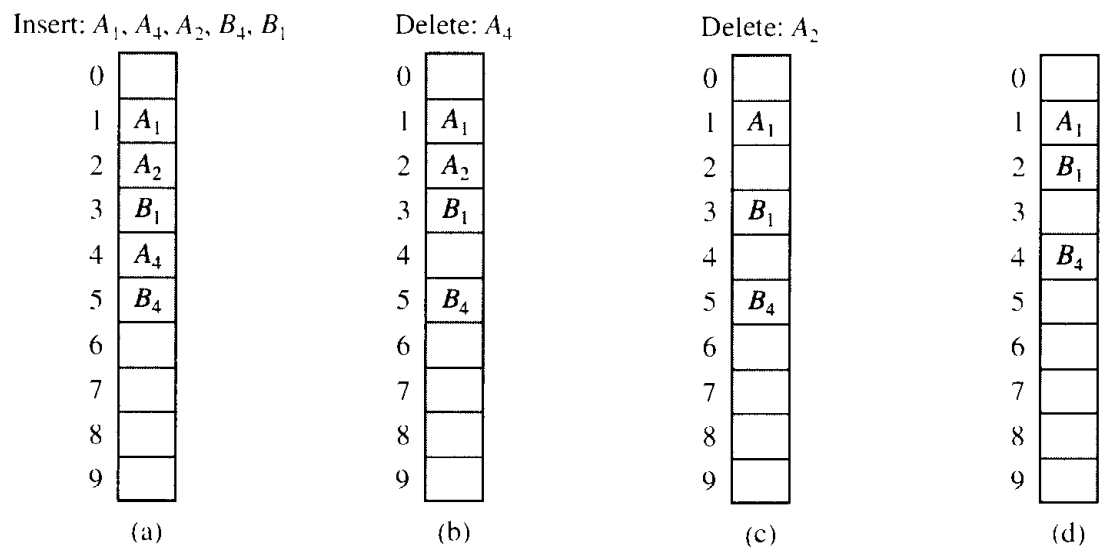
10.3 DELETION

How can we remove data from a hash table? With a chaining method, deleting an element leads to the deletion of a node from a linked list holding the element. For other methods, a deletion operation may require a more careful treatment of collision resolution, except for the rare occurrence when a perfect hash function is used.

Consider the table in Figure 10.10a in which the keys are stored using linear probing. The keys have been entered in the following order: A_1, A_4, A_2, B_4, B_1. After A_4 is deleted and position 4 is freed (Figure 10.10b), we try to find B_4 by first checking position 4. But this position is now empty, so we may conclude that B_4 is not in the table. The same result occurs after deleting A_2 and marking cell 2 as empty (Figure 10.10c). Then, the search for B_1 is unsuccessful, since if we are using linear probing, the search terminates at position 2. The situation is the same for the other open addressing methods.

If we leave deleted keys in the table with markers indicating that they are not valid elements of the table, any subsequent search for an element does not terminate prematurely. When a new key is inserted, it overwrites a key which is only a space filler. However, for a large number of deletions and a small number of additional insertions, the table becomes overloaded with deleted records, which increases the search time because the open addressing methods require testing the deleted elements. Therefore, the table should be purged after a certain number of deletions by moving undeleted elements to the cells occupied by deleted elements. Cells with deleted elements which are not overwritten by this procedure are marked as free. Figure 10.10d illustrates this situation.

FIGURE **10.10** Linear search in the situation where both insertion and deletion of keys are permitted.



10.4 PERFECT HASH FUNCTIONS

All the cases discussed so far assume that the body of data is not precisely known. Therefore, the hash function only rarely turned out to be an ideal hash function in the sense that it immediately hashed a key to its proper position and avoided any collisions. In most cases, some collision resolution technique had to be included, since sooner or later, a key would arrive which conflicted with another key in the table. Also the number of keys is rarely known in advance, so the table had to be large enough to accommodate all the arriving data. Moreover, the table size contributed to the number of collisions: A larger table has a smaller number of collisions (provided the hash function took table size into consideration). All this was caused by the fact that the body of data to be hashed in the table was not precisely known ahead of time. Therefore, a hash function was first devised and then the data were processed.

In many situations, however, the body of data is fixed, and a hash function can be devised after the data are known. Such a function may really be a perfect hash function if it hashes items on the first attempt. In addition, if such a function requires only as many cells in the table as the number of data so that no empty cell remains after hashing is completed, it is called a *minimal perfect hash function*. Wasting time for collision resolution and wasting space for unused table cells are avoided in a minimal perfect hash function.

Processing a fixed body of data is not an uncommon situation. Consider the following examples: a table of reserved words used by assemblers or compilers, files on unerasable optical disks, dictionaries, and lexical databases.

Algorithms for choosing a perfect hash function usually require tedious work due to the fact that perfect hash functions are rare. As already indicated for 50 elements and a 100-cell array, only one in 1 million is perfect. Other functions lead to collisions.

10.4.1 Cichelli's Method

One algorithm to construct a minimal perfect hash function was developed by Richard J. Cichelli. It is used to hash a relatively small number of reserved words. The function is of the form

$$h(word) = (length(word) + g(firstletter(word)) + g(lastletter(word))) \bmod TSize$$

where g is the function to be constructed. The function g assigns values to letters so that the resulting function h returns unique hash values for all words in a predefined set of words. The values assigned by g to particular letters do not have to be unique. The algorithm has three parts: computation of the letter occurrences, ordering the words, and searching. The last step is the heart of this algorithm and uses an auxiliary function `try()`. Cichelli's algorithm for constructing g and h is as follows:

```
choose a value for Max;
compute the number of occurrences of each first and last letter in the set of all words;
order all words in accordance to the frequency of occurrence of the first and the last letters;
search(wordList)
if wordList is empty
   halt;
word = first word from wordList;
wordList = wordList with the first word detached;
if the first and the last letters of word are assigned g-values
      try(word,-1,-1); // -1 signifies 'value already assigned'
      if success
         search (wordList);
      put word at the beginning of wordList and detach its hash value;
else if neither the first nor the last letters has a g-value
   for each n,m in {0,...,Max}
      try(word,n,m);
      if success
         search (wordList);
         put word at the beginning of wordList and detach its hash value;
else if either the first or the last letter has a g-value
   for each n in {0,...,Max}
      try(word,-1,n) or try(word,n,-1);
      if success
        search (wordList);
         put word at the beginning of wordList and detach its hash value;

try(word,firstLetterValue,lastLetterValue)
   if h(word) has not been claimed
         reserve h(word);
```

```
    if not -1 (i.e., not reserved)
       assign firstLetterValue and/or lastLetterValue as g-values of firstletter(word)
       and/or
       lastletter(word)
    return success;
return failure;
```

We can use this algorithm to build a hash function for the names of the nine Muses: Calliope, Clio, Erato, Euterpe, Melpomene, Polyhymnia, Terpsichore, Thalia, and Urania. A simple count of the letters renders the number of times a given letter occurs as a first and last letter (case sensitivity is disregarded): E (6), A (3), C (2), O (2), T (2), M (1), P (1), and U (1). According to these frequencies, the words can be put in the following order: Euterpe (E occurs six times as the first and the last letter), Calliope, Erato, Terpsichore, Melpomene, Thalia, Clio, Polyhymnia, and Urania.

Now the procedure `search()` is applied. Figure 10.11 contains a summary of its execution, in which Max = 4. First, the word Euterpe is tried. E is assigned the *g*-value of 0, whereby h(Euterpe) = 7, which is put on the list of reserved hash values. Everything goes well until Urania is tried. All five possible *g*-values for U result in an already reserved hash value. The procedure backtracks to the preceding step, when Polyhymnia was tried. Its current hash value is detached from the list, and the *g*-value of 1 is tried for P, which causes a failure, but 2 for P gives 3 for the hash value, so the algorithm can continue. Urania is tried again five times, then the fifth attempt is successful. All the names have been assigned unique hash values and the search process is finished. If the *g*-values for each letter are A = C = E = O = M = T = 0, P = 2, and U = 4, then h is the minimal perfect hash function for the nine Muses.

The searching process in Cichelli's algorithm is exponential since it uses an exhaustive search, and thus, it is inapplicable to a large number of words. Also, it does not guarantee that a perfect hash function can be found. For a small number of words, however, it usually gives good results. This program often needs to be run only once, and the resulting hash function can be incorporated into another program. Cichelli applied his method to the Pascal reserved words. The result was a hash function that reduced the run time of a Pascal cross-reference program by 10% after it replaced the binary search used previously.

There have been many successful attempts to extend Cichelli's technique and overcome its shortcomings. One technique modified the terms involved in the definition of the hash function. For example, other terms, the alphabetical positions of the second to last letter in the word, are added to the function definition (Sebesta and Taylor 1986), or the following definition is used (Haggard and Karplus 1986):

$$h(word) = length(word) + g_1(firstletter(word)) + \cdots + g_{length(word)}(lastletter(word))$$

Cichelli's method can also be modified by partitioning the body of data into separate buckets for which minimal perfect hash functions are found. The partitioning is performed by a grouping function, *gr*, which for each word indicates the bucket to which it belongs. Then a general hash function is generated whose form is

$$h(word) = bucket_{gr(word)} + h_{gr(word)}(word)$$

FIGURE **10.11** Subsequent invocations of the searching procedure with *Max* = 4 in Cichelli's algorithm assign the indicated values to letters and to the list of reserved hash values. The asterisks indicate failures.

		reserved hash values
Euterpe	E = 0 h = 7	{7}
Calliope	C = 0 h = 8	{7 8}
Erato	O = 0 h = 5	{5 7 8}
Terpsichore	T = 0 h = 2	{2 5 7 8}
Melpomene	M = 0 h = 0	{0 2 5 7 8}
Thalia	A = 0 h = 6	{0 2 5 6 7 8}
Clio	h = 4	{0 2 4 5 6 7 8}
Polyhymnia	P = 0 h = 1	{0 1 2 4 5 6 7 8}
Urania	U = 0 h = 6 *	{0 1 2 4 5 6 7 8}
Urania	U = 1 h = 7 *	{0 1 2 4 5 6 7 8}
Urania	U = 2 h = 8 *	{0 1 2 4 5 6 7 8}
Urania	U = 3 h = 0 *	{0 1 2 4 5 6 7 8}
Urania	U = 4 h = 1 *	{0 1 2 4 5 6 7 8}
Polyhymnia	P = 1 h = 2 *	{0 2 4 5 6 7 8}
Polyhymnia	P = 2 h = 3	{0 2 3 4 5 6 7 8}
Urania	U = 0 h = 6 *	{0 2 3 4 5 6 7 8}
Urania	U = 1 h = 7 *	{0 2 3 4 5 6 7 8}
Urania	U = 2 h = 8 *	{0 2 3 4 5 6 7 8}
Urania	U = 3 h = 0 *	{0 2 3 4 5 6 7 8}
Urania	U = 4 h = 1	{0 1 2 3 4 5 6 7 8}

(e.g., Lewis and Cook 1986). The problem with this approach is that it is difficult to find a generally applicable grouping function tuned to finding minimal perfect hash functions.

Both these ways—modifying hash function and partitioning—are not entirely successful if the same Cichelli's algorithm is used. Although Cichelli ends his paper with the adage: "When all else fails, try brute force," the attempts to modify his approach included devising a more efficient searching algorithm to circumvent the need for brute force. One such approach is incorporated in the FHCD algorithm.

10.4.2 The FHCD Algorithm

An extension of Cichelli's approach was proposed by Thomas Sager. The FHCD algorithm is an extension of Sager's method and it is discussed in this section. The FHCD algorithm (Fox et al. 1992) searches for a minimal perfect hash function of the form

$$h(word) = h_0(word) + g(h_1(word)) + g(h_2(word))$$

(modulo *TSize*), where g is the function to be determined by the algorithm. To define the functions h_i, three tables—T_0, T_1, and T_2—of random numbers are defined, one for each function h_i. Each word is equal to a string of characters $c_1 c_2 \ldots c_m$ corresponding

to a triple $(h_0(word), h_1(word), h_2(word))$ whose elements are calculated according to the formulas

$$h_0 = (T_0(c_1) + \cdots + T_0(c_m)) \bmod n$$
$$h_1 = (T_1(c_1) + \cdots + T_1(c_m)) \bmod r$$
$$h_2 = ((T_2(c_1) + \cdots + T_2(c_m)) \bmod r) + r$$

where n is the number of all words in the body of data, r is a parameter usually equal to $n/2$ or less, and $T_i(c_j)$ is the number generated in table T_i for c_j. The function g is found in three steps: *mapping*, *ordering*, and *searching*.

In the mapping step, n triples $(h_0(word), h_1(word), h_2(word))$ are created. The randomness of functions h_i usually guarantees the uniqueness of these triples; should they not be unique, new tables T_i are generated. Next, a *dependency graph* is built. It is a bipartite graph with half of its vertices corresponding to the h_1 values and labeled 0 through $r-1$ and the other half to the h_2 values and labeled r through $2r-1$. Each word corresponds to an edge of the graph between the vertices $h_1(word)$ and $h_2(word)$. The mapping step is expected to take $O(n)$ time.

As an example, we again use the set of names of the nine Muses. To generate three tables T_i, the standard function `rand()` is used, and with these tables, a set of nine triples is computed, as shown in Figure 10.12a. Figure 10.12b contains a corresponding dependency graph with $r = 3$. Note that some vertices cannot be connected to any other vertices, and some pairs of vertices can be connected with more than one arc.

The ordering step rearranges all the vertices so that they can be partitioned into a series of levels. When a sequence $v_1, \ldots, v_i$ of vertices is established, then a level $K(v_i)$ of keys is defined as a set of all the edges which connect v_i with those v_js for which $j \leq i$. The sequence is initiated with a vertex of maximum degree. Then, for each successive position i of the sequence, a vertex v_i is selected from among the vertices having at least one connection to the vertices $v_1, \ldots, v_{i-1}$ which has maximal degree. When no such vertex can be found, any vertex of maximal degree is chosen from among the unselected vertices. Figure 10.12c contains an example.

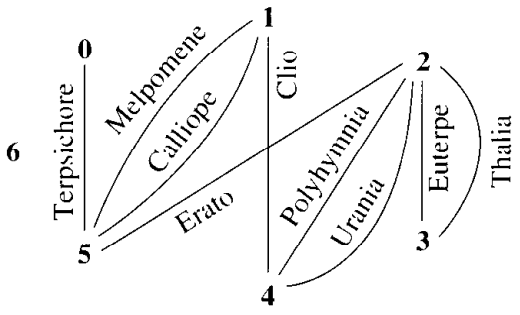
In the last step, searching, hash values are assigned to keys level by level. The g-value for the first vertex is chosen randomly among the numbers $0, \ldots, n-1$. For the other vertices, because of their construction and ordering, we have the following relation: If $v_i < r$, then $v_i = h_1$. Thus, each word in $K(v_i)$ has the same value $g(h_1(word)) = g(v_i)$. Also, $g(h_2(word))$ has already been defined, since it is equal to some v_j which has already been processed. Analogical reasoning can be applied to the case when $v_i > r$ and then $v_i = h_2$. For each word, either $g(h_1(word))$ or $g(h_2(word))$ is known. The second g-value is found randomly for each level so that the values obtained from the formula of the minimal perfect hash function h indicate the positions in the hash table that are available. Because the first choice of a random number will not always fit all words on a given level to the hash table, both random numbers may need to be tried.

The searching step for the nine Muses starts with randomly choosing $g(v_1)$. Let $g(2) = 2$, where $v_1 = 2$. The next vertex is $v_2 = 5$ so that $K(v_2) = \{\text{Erato}\}$. According to Figure 10.12a, $h_0(\text{Erato}) = 3$, and because the edge *Erato* connects v_1 and v_2, either $h_1(\text{Erato})$ or $h_2(\text{Erato})$ must be equal to v_1. We can see that $h_1(\text{Erato}) = 2 = v_1$; hence, $g(h_1(\text{Erato})) = g(v_1) = 2$. A value for $g(v_2) = g(h_2(\text{Erato})) = 6$ is chosen randomly.

FIGURE **10.12** Applying the FHCD algorithm to the names of the nine Muses.

Value of:	h_0	h_1	h_2
Calliope	(0	1	5)
Clio	(7	1	4)
Erato	(3	2	5)
Euterpe	(6	2	3)
Melpomene	(3	1	5)
Polyhymnia	(8	2	4)
Terpsichore	(8	0	5)
Thalia	(8	2	3)
Urania	(0	2	4)

(a)



(b)

Level	Node	Arcs
0	2	
1	5	Erato
2	1	Calliope, Melpomene
3	4	Clio, Polyhymnia, Urania
4	3	Euterpe, Thalia
5	0	Terpsichore

(c)

Level	Vertex	g-value		
0	2	2		
1	5	6	h(Erato)	$= (3 + 2 + 6) \% 9 = 2$
2	1	4	h(Calliope)	$= (0 + 4 + 6) \% 9 = 1$
2	1	4	h(Melpomene)	$= (3 + 4 + 6) \% 9 = 4$
3	4	2	h(Clio)	$= (7 + 6 + 2) \% 9 = 6$
3	4	2	h(Polyhymnia)	$= (8 + 6 + 2) \% 9 = 7$
3	4	2	h(Urania)	$= (0 + 6 + 2) \% 9 = 8$
4	3	4	h(Euterpe)	$= (6 + 2 + 4) \% 9 = 3$
4	3	4	h(Terpsichore)	$= (8 + 2 + 4) \% 9 = 5$
4	3	4	h(Thalia)	$= (8 + 4 + 6) \% 9 = 0$

(d)

Function g	
0	4
1	4
2	2
3	4
4	2
5	6

(e)

From this, $h(\text{Erato}) = (h_0(\text{Erato}) + g(h_1(\text{Erato})) + g(h_2(\text{Erato}))) \bmod TSize = (3 + 2 + 6) \bmod 9 = 2$. This means that position 2 of the hash table is no longer available. The new g-value, $g(5) = 6$, is retained for later use.

Now, $v_3 = 1$ is tried, with $K(v_3) = \{\text{Calliope, Melpomene}\}$. The h_0-values for both words are retrieved from the table of triples, and the $g(h_2)$-values are equal to 6 for both words, since $h_2 = v_2$ for both of them. Now we must find a random $g(h_1)$-value

such that the hash function h computed for both words renders two numbers different from 2, since position two is already occupied. Assume that this number is 4. As a result, h(Calliope) = 1 and h(Melpomene) = 4. Figure 10.12d contains a summary of all the steps. Figure 10.12e shows the values of the function g. Through these values of g, the function h becomes a minimal perfect hash function. However, since g is given in tabular form and not with a neat formula, it has to be stored as a table to be used every time function h is needed, which may not be a trivial task. The function $g: \{0, \ldots, 2r-1\} \rightarrow \{0, \ldots, n-1\}$, and the size of g's domain increases with r. The parameter r is approximately $n/2$, which for large databases means that the table storing all values for g is not of a negligible size. This table has to be kept in main memory to make computations of the hash function efficient.

10.5 Hash Functions for Extendible Files

All the methods discussed so far work on tables of fixed sizes. This is a reasonable assumption for arrays, but for files it may be too restrictive. After all, file sizes change dynamically by adding new elements or deleting old ones. Some hashing techniques can be used in this situation, such as coalesced hashing or hashing with chaining, but some of them may be inadequate. New techniques have been developed which specifically take into account the variable size of the table or file. We can distinguish two classes of such techniques: directory and directoryless.

In the directory schemes, key access is mediated by the access to a directory or an index of keys in the structure. There are several techniques and modifications to those techniques in the category of the directory schemes. We mention only a few: *expandable hashing* (Knott 1971), *dynamic hashing* (Larson 1978), and *extendible hashing* (Fagin et al. 1979). All three methods distribute keys among buckets in a similar fashion. The main difference is the structure of the index (directory). In expandable hashing and dynamic hashing, a binary tree is used as an index of buckets. On the other hand, in extendible hashing, a directory of records is kept in a table.

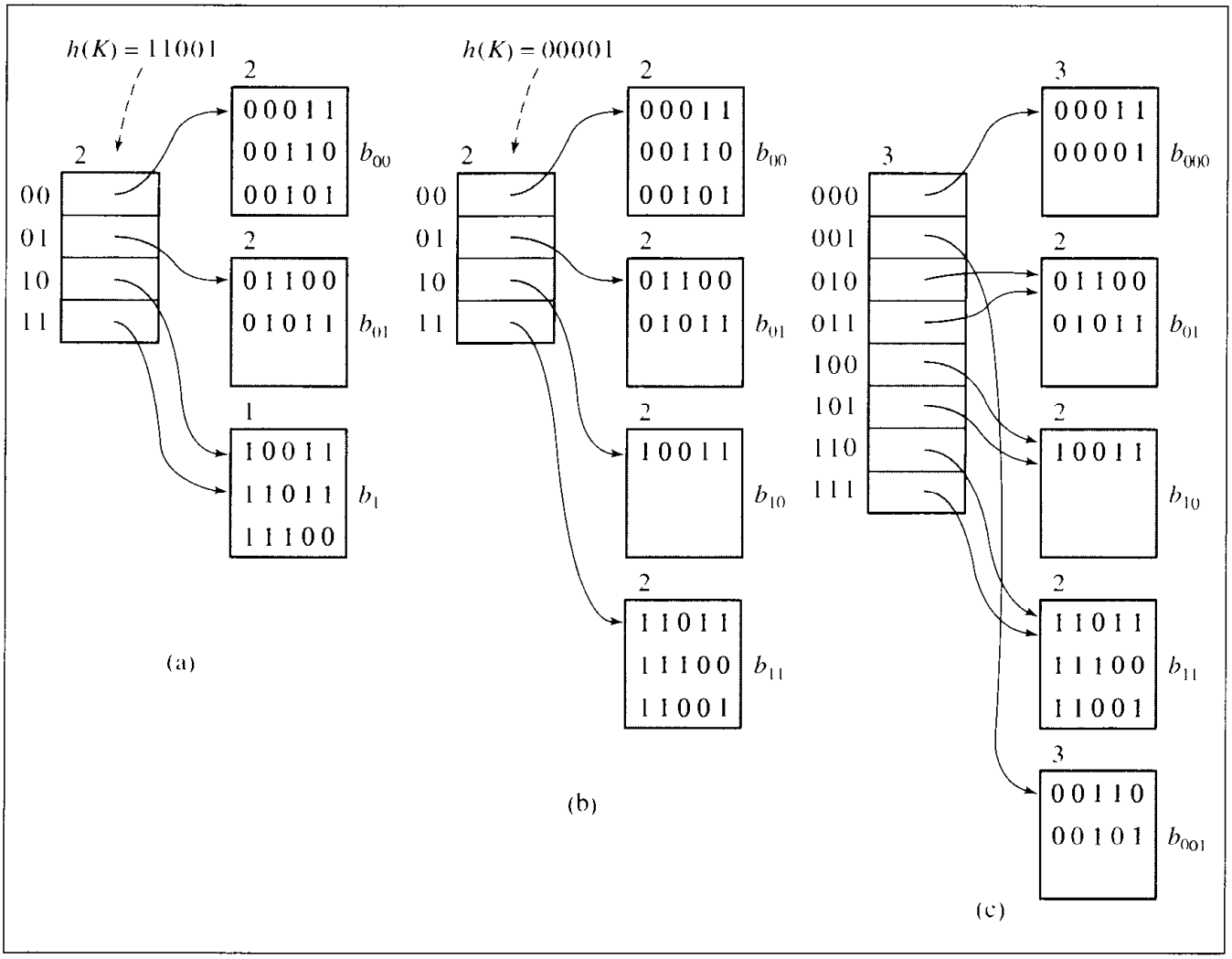
One directoryless technique is *virtual hashing*, defined as "any hashing which may dynamically change its hashing function" (Litwin 1978). This change of hashing function compensates for the lack of a directory. An example of this approach is linear hashing (Litwin 1980). In the following pages, one method from each category is discussed.

10.5.1 Extendible Hashing

Assume that a hashing technique is applied to a dynamically changing file composed of buckets, and each bucket can hold only a fixed number of items. Extendible hashing accesses the data stored in buckets indirectly through an index that is dynamically adjusted to reflect changes in the file. The characteristic feature of extendible hashing is the organization of the index, which is an expandable table.

A hash function applied to a certain key indicates a position in the index and not in the file (or table of keys). Values returned by such a hash function are called *pseudokeys*. In this way, the file requires no reorganization when data are added to it or

FIGURE **10.13** An example of extendible hashing.



deleted from it, since these changes are indicated in the index. Only one hash function h can be used, but depending on the size of the index, only a portion of the address $h(K)$ is utilized. A simple way to achieve this effect is by looking at the address $h(K)$ as a string of bits from which only the i leftmost bits can be used. The number i is called the *depth* of the directory. In Figure 10.13a, the depth is equal to two.

As an example, assume that the hash function h generates patterns of five bits. If this pattern is the string 01011 and the depth is two, then the two leftmost bits, 01, are considered to be the position in the directory containing the pointer to a bucket in which the key can be found or into which it is to be inserted. In Figure 10.13, the values of h are shown in the buckets, but these values only represent the keys that are actually stored in these buckets.

Each bucket has a *local depth* associated with it that indicates the number of leftmost bits in $h(K)$. The leftmost bits are the same for all keys in the bucket. In Figure 10.13, the local depths are shown on top of each bucket. For example, the bucket b_{00}

holds all keys for which $h(K)$ starts with 00. More important, the local depth indicates whether the bucket can be accessed from only one location in the directory or from at least two. In the first case, when the local depth is equal to the depth of directory, it is necessary to change the size of the directory after the bucket is split in the case of overflow. When the local depth is smaller than the directory depth, splitting the bucket only requires changing half of the pointers pointing to this bucket so that they point to the newly created one. Figure 10.13b illustrates this case. After a key with h-value 11001 arrives, its two first bits (since depth = 2) direct it to the fourth position of the directory, from which it is sent to the bucket b_1, which contains keys whose h-value starts with 1. An overflow occurs, and b_1 is split into b_{10} (the new name for the old bucket) and b_{11}. The local depths of these two buckets are set to two. The pointer from position 11 points now to b_{11}, and the keys from b_1 are redistributed between b_{10} and b_{11}.

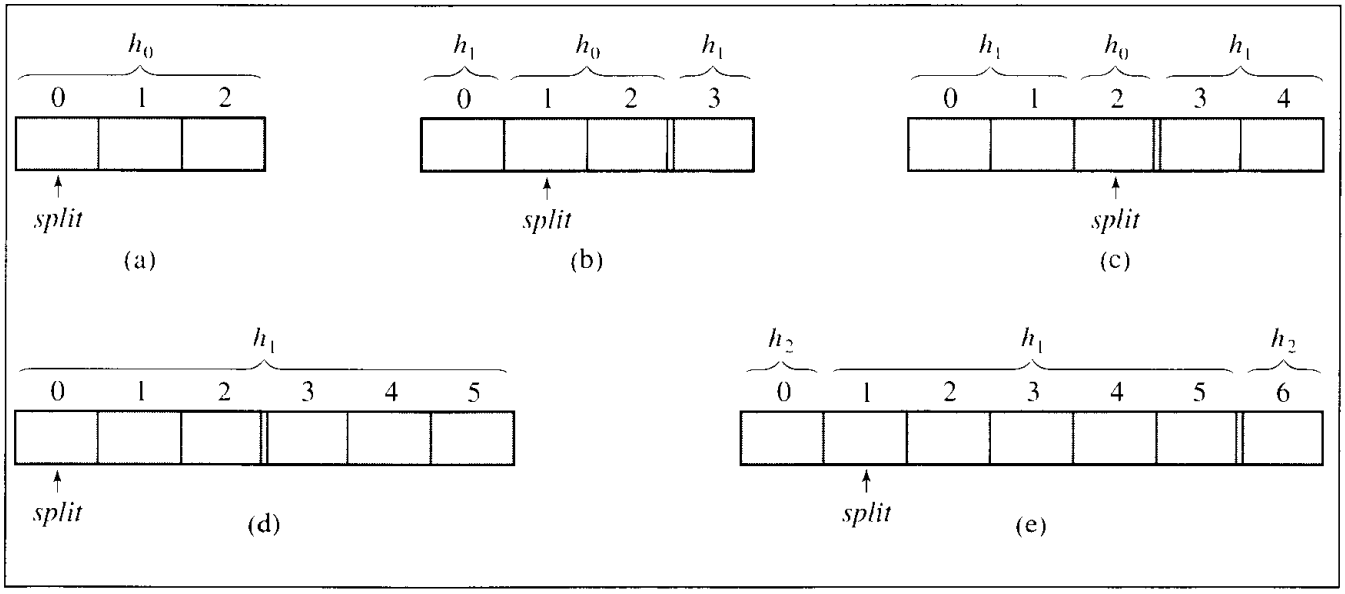
The situation is more complex if overflow occurs in a bucket with a local depth equal to the depth of the directory. For example, consider the case when a key with h-value 00001 arrives at the table in Figure 10.13b and is hashed through position 00 (its first two bits) to bucket b_{00}. A split occurs, but the directory has no room for the pointer to the new bucket. As a result, the directory is doubled in size so that its depth is now equal to three, b_{00} becomes b_{000} with an increased local depth, and the new bucket is b_{001}. All the keys from b_{00} are divided between the new buckets: Those whose h-value starts with 000 become elements of b_{000}; the remaining keys, those with prefix 001, are put in b_{001}, as in Figure 10.13c. Also, all the slots of the new directory have to be set to their proper values by having *newdirectory*$[2 \cdot i]$ = *olddirectory*$[i]$ and *newdirectory*$[2 \cdot i + 1]$ = *olddirectory*$[i]$ for i's ranging over positions of the *olddirectory*, except for the position referring to the bucket which just has been split.

The following algorithm inserts a record into a file using extendible hashing.

```
extendibleHashingInsert(K)
   bitPattern = h(K);
   p = directory[depth(directory) leftmost bits of bitPattern];
   if space is available in bucket bd pointed to by p
      place K in the bucket;
   else split bucket bd into bd0 and bd1;
      set local depth of bd0 and bd1 to depth(bd) + 1;
      distribute records from bd between bd0 and bd1;
      if depth(bd) < depth(directory)
         update the half of the pointers which pointed to bd to point to bd1;
      else double the directory and increment its depth;
         set directory entries to proper pointers;
```

An important advantage of using extendible hashing is that it avoids a reorganization of the file if the directory overflows. Only the directory is affected. Because the directory in most cases is kept in main memory, the cost of expanding and updating it is very small. However, for large files of small buckets, the size of the directory can become so large that it may be put in virtual memory or explicitly in a file, which may slow down the process of using the directory. Also, the size of the directory does not grow uniformly, since it is doubled if a bucket with a local depth equal to the depth of

FIGURE **10.14** Splitting buckets in the linear hashing technique.

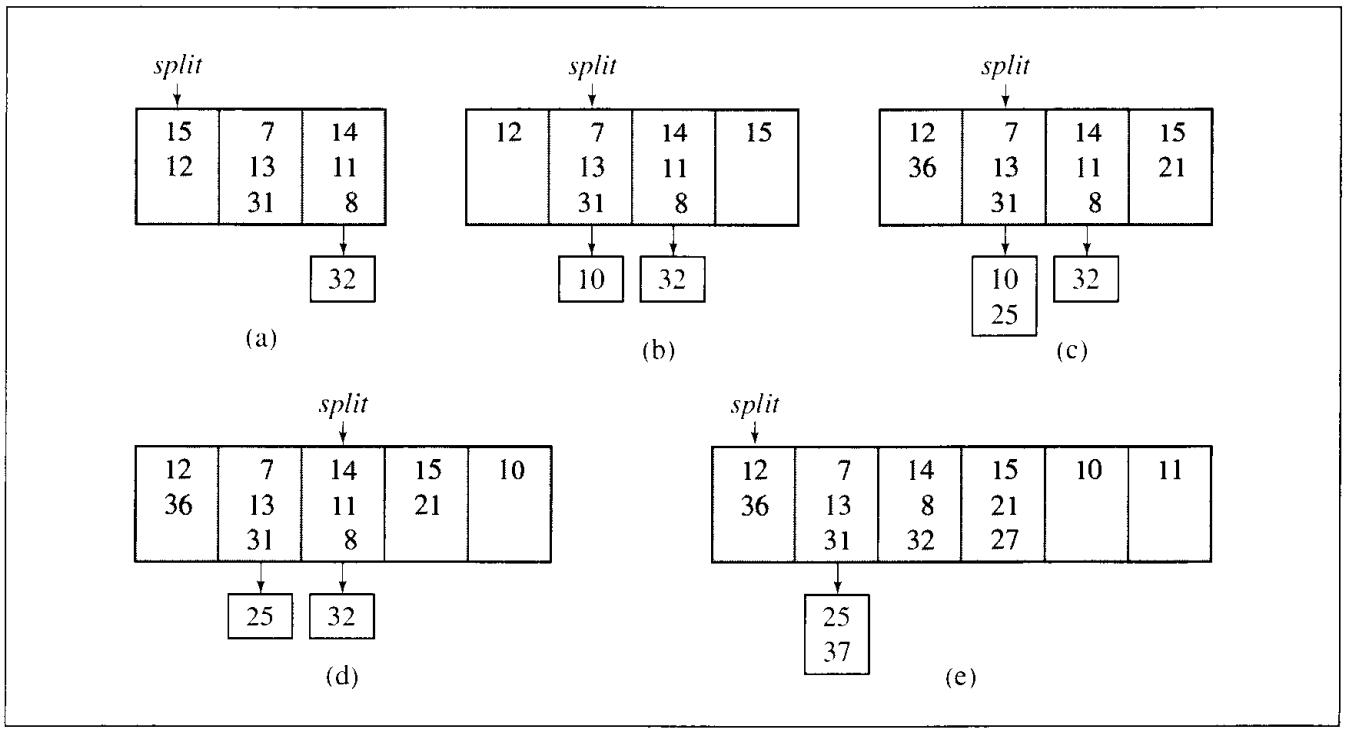


the directory is split. This means that for large directories there will be many redundant entries in the directory. To rectify the problem of an overgrown directory, David Lomet proposed using extendible hashing until the directory becomes too large to fit into main memory. Afterward, the buckets are doubled instead of the directory, and the bits in the bit pattern $h(K)$ that come after the first *depth* bits are used to distinguish between different parts of the bucket. For example, if *depth* = 3 and a bucket b_{10} has been quadrupled, its parts are distinguished with bit strings 00, 01, 10, and 11. Now, if $h(K) = 101\mathit{01}101$, the key K is searched for in the second portion, 01, of b_{101}.

10.5.2 Linear Hashing

Extendible hashing allows the file to expand without reorganizing it, but it requires storage space for an index. In the method developed by Witold Litwin, no index is necessary because new buckets generated by splitting existing buckets are always added in the same linear way, so there is no need to retain indexes. To this end, a pointer *split* indicates which bucket is to be split next. After the bucket pointed to by *split* is divided, the keys in this bucket are distributed between this bucket and the newly created bucket which is added to the end of the table. Figure 10.14 contains a sequence of initial splits in which *TSize* = 3. Initially, the pointer *split* is zero. If the loading factor exceeds a certain level, a new bucket is created, keys from bucket zero are distributed between bucket zero and bucket three, and *split* is incremented. How is this distribution performed? If only one hash function is used, then it hashes keys from bucket zero to bucket zero before and after splitting. This means that one function is not sufficient.

FIGURE **10.15** Inserting keys to buckets and overflow areas with the linear hashing technique.



At each level of splitting, linear hashing maintains two hash functions, h_{level} and $h_{level+1}$, such that $h_{level}(K) = K \bmod (TSize \cdot 2^{level})$. The first hash function, h_{level}, hashes keys to buckets which have not yet been split on the current level. The second function, $h_{level+1}$ is used for hashing keys to already split buckets. The algorithm for linear hashing is as follows:

initialize: `split = 0; level = 0;`

```
linearHashingInsert(K)
   if h_level(K) < split      // bucket h_level(K) has been split
      hashAddress = h_level+1(K);
   else hashAddress = h_level(K);
   insert K in a corresponding bucket or an overflow area if possible;
   while the loading factor is high or K not inserted
      create a new bucket with index split + TSize * 2^level;
      redistribute keys from bucket split between buckets split and split + TSize * 2^level;
      split++;
      if split == TSize * 2^level // all buckets on the current
                                  // level have been split;
         level++;                 // proceed to the next level
         split = 0;
      try to insert K if not inserted yet;
```

It may still be unclear when to split a bucket. Most likely, as the algorithm assumes, a threshold value of the loading factor is used to decide whether or not to split a bucket. This threshold has to be known in advance, and its magnitude is chosen by the program designer. To illustrate, assume that keys can be hashed to buckets in a file. If a bucket is full, the overflowing keys can be put on a linked list in an overflow area. Consider the situation in Figure 10.15a. In this figure, $TSize = 3$, $h_0(K) = K \bmod TSize$, $h_1(K) = K \bmod 2 \cdot TSize$. Let the size of the overflow area $OSize = 3$, and let the highest acceptable loading factor which equals the number of elements divided by the number of slots in the file and in the overflow area be 80%. The current loading factor in Figure 10.15a is 75%. If the key 10 arrives, it is hashed to location 1, but the loading factor increases to 83%. The first bucket is split and the keys are redistributed using function h_1, as in Figure 10.15b. Note that the first bucket had the lowest load of all three buckets, and yet it was the bucket that was split.

Assume that 21 and 36 have been hashed to the table (Figure 10.15c), and now 25 arrives. This causes the loading factor to increase to 87%, resulting in another split, this time the split of the second bucket, which results in the configuration shown in Figure 10.15d. After hashing 27 and 37, another split occurs, and Figure 10.15e illustrates the new situation. Because *split* reached the last value allowed on this level, it is assigned the value of zero, and the hash function to be used in subsequent hashing is h_1, the same as before, and a new function, h_2, is defined as $K \bmod 4 \cdot TSize$. All of these steps are summarized in the following table:

K	$h(K)$	Number of Items	Number of Cells	Loading Factor	*Split*	Hash Functions	
		9	9 + 3	9/12 = 75%	0	K mod 3	K mod 6
10	1	10	9 + 3	10/12 = 83%	0	K mod 3	K mod 6
		10	9 + 3	10/15 = 67%	1	K mod 3	K mod 6
21	3	11	12 + 3	11/15 = 73%	1	K mod 3	K mod 6
36	0	12	12 + 3	12/15 = 80%	1	K mod 3	K mod 6
25	1	13	12 + 3	13/15 = 87%	1	K mod 3	K mod 6
		13	12 + 3	13/18 = 72%	2	K mod 3	K mod 6
27	3	14	15 + 3	14/18 = 78%	2	K mod 3	K mod 6
37	1	15	15 + 3	15/18 = 83%	2	K mod 3	K mod 6
		15	18 + 3	15/21 = 71%	0	K mod 6	K mod 12

Note that linear hashing requires the use of some overflow area because the order of splitting is predetermined. In the case of files, this may mean more than one file access. This area can be explicit and different from buckets, but it can be introduced somewhat in the spirit of coalesced hashing by utilizing empty space in the buckets (Mullin 1981). In a directory scheme, on the other hand, an overflow area is not necessary, although it can be used.

As in a directory scheme, linear hashing increases the address space by splitting a bucket. It also redistributes the keys of the split bucket between the buckets that result

from the split. Because no indexes are maintained in linear hashing, this method is faster and requires less space than previous methods. The increase in efficiency is particularly noticeable for large files.

10.6 Case Study: Hashing with Buckets

The most serious problem to be solved in programs which rely on a hash function to insert and retrieve items from an undetermined body of data is resolving collision. Depending on the technique, allowing deletion of items from the table can significantly increase the complexity of the program. In this case study, a program is developed that allows the user to insert and delete elements from the file `names` interactively. This file contains names and phone numbers and is initially ordered alphabetically. At the end of the session, the file is ordered with all updates included. To that end, the `outfile` is used throughout the execution of the program. `outfile` is the file of buckets initialized as empty. Elements that cannot be hashed to the corresponding bucket in this file are stored in the file `overflow`. At the end of the session both files are combined and sorted to replace the contents of the original file `names`.

The `outfile` is used here as the hash table. First, this file is prepared by filling it with `tableSize * bucketSize` empty records (one record is simply a certain number of bytes). Next, all entries of `names` are transferred to `outfile` to buckets indicated by the hash function. This transfer is performed by the function `insertion()`, which includes the hashed item in the bucket indicated by the hash function or in `overflow` if the bucket is full. In the latter case, `overflow` is searched from the beginning, and if a position occupied by a deleted record is found, the overflow item replaces it. If the end of `overflow` is reached, the item is put at the end of this file.

After initializing `outfile`, a menu is displayed and the user chooses to insert a new record, delete an old one, or exit. For insertion, the same function is used as before. No duplicates are allowed. When the user wants to delete an item, the hash function is used to access the corresponding bucket, and the linear search of positions in the bucket is performed until the item is found, in which case the deletion marker "#" is written over the first character of the item in the bucket. However, if the item is not found and the end of the bucket is reached, the search continues sequentially in `overflow` until either the item is found and marked as deleted or the end of the file is encountered.

If the user chooses to exit, the undeleted entries of `overflow` are transferred to `outfile`, and all undeleted entries of `outfile` are sorted using an external sort. To that end, quicksort is applied both to `outfile` and to an array `pointers[]` which contains the addresses of entries in `outfile`. For comparison, the entries in `outfile` can be accessed, but the elements in `pointers[]` are moved, not the elements of `outfile`.

After this indirect sorting is accomplished, the data in `outfile` have to be put in alphabetical order. This is accomplished by transferring entries from `outfile` to `unsorted` using the order indicated in `pointers[]`, that is, by going down the array and retrieving the entry in `outfile` through the address stored in the currently accessed cell. After that, `names` is deleted and `unsorted` is renamed `names`.

Here is an example. If the contents of the original file are

```
Adam 123-4567        Brenda 345-5352      Brendon 983-7373
Charles 987-1122     Jeremiah 789-4563    Katherine 823-1573
Patrick 757-4532     Raymond 090-9383     Thorsten 929-6632
```

the hashing generates the `outfile`:

```
Katherine 823-1573  |******************||
Adam 123-4567       |Brenda 345-5352    ||
Raymond 090-9383    |Thorsten 929-6632  ||
```

and the file `overflow`:

```
Brendon 983-7373    |Charles 987-1122   ||
Jeremiah 789-4563   |Patrick 757-4532   ||
```

(The vertical bars are *not* included in the file; one bar divides the records in the same bucket, and two bars separate different buckets.)

After inserting `Carol 654-6543` and deleting `Brenda 345-5352` and `Jeremiah 789-4563`, the file's contents are:

```
outfile:
Katherine 823-1573  |Carol 654-6543     ||
Adam 123-4567       |#renda 345-5352    ||
Raymond 090-9383    |Thorsten 929-6632  ||
```

and `overflow`:

```
Brendon 983-7373    |Charles 987-1122   ||
#eremiah 789-4563   |Patrick 757-4532   ||
```

A subsequent insertion of `Maggie 733-0983` and deletion of `Brendon 983-7373` changes only `overflow`:

```
#rendon 983-7373    |Charles 987-1122   ||
Maggie 733-0983     |Patrick 757-4532   ||
```

After the user chooses to exit, undeleted records from `overflow` are transferred to `outfile`, which now includes:

```
Katherine 823-1573  |Carol 654-6543     ||
Adam 123-4567       |#renda 345-5352    ||
Raymond 090-9383    |Thorsten 929-6632  ||
Charles 987-1122    |Maggie 733-0983    ||
Patrick 757-4532    |
```

This file is sorted and the outcome is:

```
Adam 123-4567       |Carol 654-6543     ||
Charles 987-1122    |Katherine 823-1573 ||
Maggie 733-0983     |Patrick 757-4532   ||
Raymond 090-9383    |Thorsten 929-6632  ||
```

Figure 10.16 contains the code for this program.

FIGURE 10.16 Implementation of hashing using buckets.

```
#include <iostream.h>
#include <fstream.h>
#include <string.h>
#include <ctype.h>
#include <iomanip.h>
#include <stdio.h> // unlink(), rename();

const int bucketSize = 2, tableSize = 3, strLen = 20;
const int recordLen = strLen;

class File {
public:
    File() : empty('*'), delMarker('#') {
    }
    void processFile(char*);
private:
    const char empty, delMarker;
    long *pointers;
    fstream outfile, overflow, sorted;
    int  hash(char*);
    void swap(long& i, long& j) {
        long tmp = i; i = j; j = tmp;
    }
    void getName(char*);
    void insert(char line[]) {
        getName(line); insertion(line);
    }
    void insertion(char*);
    void remove(char*);
    void partition(int,int,int&);
    void QSort(int,int);
    void sortFile();
    void combineFiles();
};

unsigned long File::hash(char *s) {
    unsigned long xor = 0, pack;
    int i, j, slength; // exclude trailing blanks;
    for (slength = strlen(s); isspace(s[slength-1]); slength--);
    for (i = 0; i < slength; ) {
```

FIGURE **10.16** *(continued)*

```
        for (pack = 0, j = 0; ; j++, i++) {
            pack |= (unsigned long) s[i];     // include s[i] in the
                                                 rightmost
            if (j == 3 || i == slength - 1) { // byte of pack;
                i++;
                break;
            }
            pack <<= 8;
        }             // xor at one time 8 bytes from s;
        xor ^= pack;  // last iteration may put less
    }                 // than 8 bytes in pack;
    return (xor % tableSize) * bucketSize * recordLen;
}// return byte position of home bucket for s;

void File::getName(char line[]) {
    cout << "Enter name: ";
    cin.getline(line,recordLen+1);
    for (int i = strlen(line); i < recordLen; i++)
        line[i] = ' ';
    line[recordLen] = '\0';
}

void File::insertion(char line[]) {
    int address = hash(line), counter = 0;
    char name[recordLen+1];
    name[recordLen] = '\0';
    bool done = false, inserted = false;
    outfile.seekg(address,ios::beg);
    while (!done && outfile.getline(name,recordLen+1)) {
        if (name[0] == empty || name[0] == delMarker) {
            outfile.seekg(address+counter*recordLen,ios::beg);
            outfile << line << setw(strlen(line)-recordLen);
            done = inserted = true;
        }
        else if (!strcmp(name,line)) {
            cout << line << " is already in the file\n";
            return;
        }
        else counter++;
        if (counter == bucketSize)
```

Continues

FIGURE **10.16** *(continued)*

```
            done = true;
        else outfile.seekg(address+counter*recordLen,ios::beg);
    }
    if (!inserted) {
        done = false;
        counter = 0;
        overflow.clear();
        overflow.seekg(0,ios::beg);
        while (!done && overflow.getline(name,recordLen+1)) {
            if (name[0] == delMarker)
                done = true;
            else if (!strcmp(name,line)) {
                cout << line << " is already in the file\n";
                return;
            }
            else counter++;
        }
        overflow.clear();
        if (done)
            overflow.seekg(counter*recordLen,ios::beg);
        else overflow.seekg(0,ios::end);
        overflow << line << setw(strlen(line)-recordLen);
    }
}

void File::remove(char line[]) {
    getName(line);
    int address = hash(line), counter = 0;
    bool done = false, removed = false;
    char name2[recordLen+1];
    name2[recordLen] = '\0';
    outfile.clear();
    outfile.seekg(address,ios::beg);
    while (!done && outfile.getline(name2,recordLen+1)) {
        if (!strcmp(line,name2)) {
            outfile.seekg(address+counter*recordLen,ios::beg);
            outfile.put(delMarker);
            done = removed = true;
        }
        else counter++;
```

FIGURE **10.16** *(continued)*

```
        if (counter == bucketSize)
             done = true;
        else outfile.seekg(address+counter*recordLen,ios::beg);
    }
    if (!removed) {
        done = false;
        counter = 0;
        overflow.clear();
        overflow.seekg(0,ios::beg);
        while (!done && overflow.getline(name2,recordLen+1)) {
            if (!strcmp(line,name2)) {
                 overflow.clear();
                 overflow.seekg(counter*recordLen,ios::beg);
                 overflow.put(delMarker);
                 done = removed = true;
            }
            else counter++;
            overflow.seekg(counter*recordLen,ios::beg);
        }
    }
    if (!removed)
        cout << line << " is not in database\n";
}

void File::partition (int low, int high, int& pivotLoc) {
    char rec[recordLen+1], pivot[recordLen+1];
    rec[recordLen] = pivot[recordLen] = '\0';
    register int i, lastSmall;
    swap(pointers[low],pointers[(low+high)/2]);
    outfile.seekg(pointers[low]*recordLen,ios::beg);
    outfile.getline(pivot,recordLen+1);
    for (lastSmall = low, i = low+1; i <= high; i++) {
        outfile.seekg(pointers[i]*recordLen,ios::beg);
        outfile.getline(rec,recordLen+1);
        if (strcmp(rec,pivot) < 0) {
            lastSmall++;
            swap(pointers[lastSmall],pointers[i]);
        }
    }
    swap(pointers[low],pointers[lastSmall]);
```

Continues

FIGURE **10.16** *(continued)*

```
    pivotLoc = lastSmall;
}

void File::QSort(int low, int high) {
    int pivotLoc;
    if (low < high) {
        partition(low, high, pivotLoc);
        QSort(low, pivotLoc-1);
        QSort(pivotLoc+1, high);
    }
}

void File::sortFile() {
    char rec[recordLen+1];
    QSort(1,pointers[0]);   // pointers[0] contains the # of elements;
    rec[recordLen] = '\0';  // put data from outfile in sorted order
    for (int i = 1; i <= pointers[0]; i++) {       // in file sorted;
        outfile.seekg(pointers[i]*recordLen,ios::beg);
        outfile.getline(rec,recordLen+1);
        sorted << rec << setw(strlen(rec)-recordLen);
    }
}

// data from overflow file and outfile are all stored in outfile and
// prepared for external sort by loading positions of the data to an
array;

void File::combineFiles() {
    int counter = bucketSize*tableSize;
    char rec[recordLen+1];
    rec[recordLen] = '\0';
    outfile.seekg(0,ios::end);
    overflow.seekg(0,ios::beg);
    while (overflow.getline(rec,recordLen+1)) { // transfer from
        if (rec[0] != delMarker) {         // overflow to outfile only
            counter++;                    // valid (non-removed) items;
            outfile << rec << setw(strlen(rec)-recordLen);
        }
    }
    pointers = new long[counter+1];
```

FIGURE **10.16** *(continued)*

```
    outfile.seekg(0,ios::beg);    // load to array pointers positions
    int arrCnt = 1;               // of valid data stored in output file;
    for (int i = 0; i < counter; i++) {
        outfile.seekg(i*recordLen,ios::beg);
        outfile.getline(rec,recordLen+1);
        if (rec[0] != empty && rec[0] != delMarker)
            pointers[arrCnt++] = i;
    }
    pointers[0] = --arrCnt; // store the number of data in position 0;
}

void File::processFile(char *fileName) {
    ifstream fIn(fileName);
    if (fIn.fail()) {
         cerr << "Cannot open " << fileName << endl;
         return;
    }
    char command[strLen] = "\0";
    outfile.open("outfile",ios::in|ios::out|ios::trunc);
    sorted.open("sorted",ios::in|ios::out|ios::trunc);
    overflow.open("overflow",ios::in|ios::out|ios::trunc);
    for (int i = 1; i <= tableSize*bucketSize*recordLen; i++)
                                          // initialize
        outfile << empty;                 // outfile;
    char line[recordLen+1];
    line[recordLen] = '\0';
    while (fIn.getline(line,recordLen+1)) // load infile to outfile;
        insertion(line);
    printFile ("outfile",outfile);
    printFile ("overflow",overflow);
    while (strcmp(command,"exit")) {
        cout << "Enter command (insert, remove, or exit): ";
        cin.getline(command,strLen+1);
        if (!strcmp(command,"insert"))
             insert(line);
        else if (!strcmp(command,"remove"))
             remove(line);
        else if (strcmp(command,"exit"))
             cout << "Wrong command entered, please retry.\n";
    printFile ("outfile",outfile);
```

Continues

FIGURE **10.16** *(continued)*

```
    printFile ("overflow",overflow);
    }
    combineFiles();
    sortFile();
    printFile ("sorted",sorted);
    outfile.close();
    sorted.close();
    overflow.close();
    fIn.close();
              unlink(fileName);
              rename("sorted",fileName);
}

void main(int argc, char* argv[]) {
    char fileName[30];
    if (argc != 2) {
         cout << "Enter a file name: ";
         cin.getline(fileName,30);
    }
    else strcpy(fileName,argv[1]);
    File fClass;
    fClass.processFile(fileName);
}
```

The hash function used in the program appears overly complicated. The function `hash()` applies the xor function to the chunks of four characters of a string. For example, the hash value corresponding to the string "ABCDEFGHIJ" is the number "ABCD"^"EFGH"^"IJ," that is, in hexadecimal notation, 0x41424344^0x45464748^0x0000494a. It may appear that the same outcome can be generated with the function

```
unsigned long File::hash2(char *s) {
   unsigned long xor, remainder;
   for (xor = 0; strlen(s) >= 4; s += 4) {
      xor |= *reinterpret_cast<unsigned long*>(s);
      if (strlen(s) != 0) {
         strcpy(reinterpret_cast<char*>(&remainder),s);
         xor != remainder;
   }
   return (xor % tableSize) * bucketSize * recordLen;
}
```

The problem with this simpler function is that it may return different values for the same string. The outcome depends on the way numbers are stored on a particular system, which in turn depends on the "endianness" supported by the system. If a system is big-endian, it stores the most significant byte in the lowest address; that is, numbers are stored "big-end-first." In a little-endian system, the most significant byte is in the highest address. For example, number 0x12345678 is stored as 0x12345678 in the big-endian system—first the contents of the highest order byte, 12, then the contents of the lower order byte, 34, and so on. On the other hand, the same number is stored as 0x78563412 in the little-endian system—first the contents of the lowest order byte, 78, then the contents of the higher order byte, 56, and so on. Consequently, on the big-endian systems, the statement

```
xor |= *reinterpret_cast<unsigned long*>(s);
```

xors with `xor` the substring "ABCD" of string `s` = "ABCDEFGHIJ" because the cast forces the system to treat the first four characters in `s` as representing a long number without changing the order of the characters. On a little-endian system, the same four characters are read in reverse order, lowest order byte first. Therefore, to prevent this system dependence, one character of `s` is processed at a time and included in `xor`, after which the contents of `xor` are shifted to the left by eight bits to make room for another character. This is a way of simulating a big-endian reading.

10.7 Exercises

1. What is the minimum number of keys which are hashed to their home positions using the linear probing technique? Show an example using a 5-cell array.

2. Consider the following hashing algorithm (Bell and Kaman 1970). Let Q and R be the quotient and remainder obtained by dividing K by *TSize* and let the probing sequence be created by the following recurrence formula:

$$h_i(K) = \begin{cases} R & \text{if } i = 0 \\ (h_{i-1}(K) + Q) \bmod TSize & \text{otherwise} \end{cases}$$

   What is the desirable value of *TSize*? What condition should be imposed on Q?

3. Is there any advantage to using binary search trees instead of linked lists in the separate chaining method?

4. In Cichelli's method for constructing the minimal hash function, why are all words first ordered according to the occurrence of the first and the last letters? The subsequent searching algorithm does not make any reference to this order.

5. Trace the execution of the searching algorithm used in Cichelli's technique with *Max* = 3. (See the illustration of such a trace for *Max* = 4 in Figure 10.11.)

6. In which case does Cichelli's method not guarantee to generate a minimal perfect hash function?

7. Apply the FHCD algorithm to the nine Muses with $r = n/2 = 4$ and then with $r = 2$. What is the impact of the value of r on the execution of this algorithm?

8. Strictly speaking, the hash function used in extendible hashing also dynamically changes. In what sense is this true?

9. Consider an implementation of extendible hashing that allows buckets to be pointed to by only one pointer. The directory contains null pointers so that all pointers in the directory are unique except the null pointers. What keys are stored in the buckets? What are the advantages and disadvantages of this implementation?

10. How would the directory used in extendible hashing be updated after splitting if the last *depth* bits of $h(K)$ are considered an index to the directory, not the first *depth* bits?

11. List the similarities and differences between extendible hashing and B^+-trees.

12. What is the impact of the uniform distribution of keys over the buckets in extendible hashing on the frequency of splitting?

13. Apply the linear hashing method to hash numbers 12, 24, 36, 48, 60, 72, and 84 to an initially empty table with three buckets and with three cells in the overflow area. What problem can you observe? Can this problem bring the algorithm to a halt?

14. Outline an algorithm to delete a key from a table when the linear hashing method is used for inserting keys.

15. The function `hash()` applied in the case study uses the exclusive or (xor) operation to fold all the characters in a string. Would it be a good idea to replace it by bitwise-and or bitwise-or?

10.8 Programming Assignments

1. As discussed in this chapter, the linear probing technique used for collision resolution has a rapidly deteriorating performance if a relatively small percentage of the cells are available. This problem can be solved using another technique for resolving collisions, and also by finding a better hash function, ideally, a perfect hash function. Write a program that evaluates the efficiency of various hashing functions combined with the linear probing method. Have your program write a table similar to the one in Figure 10.4, which gives the averages for successful and unsuccessful trials of locating items in the table. Use functions for operating on strings and a large text file whose words will be hashed to the table. Here are some examples of such functions (all values are divided modulo *TSize*):

   a. FirstLetter(s) + SecondLetter(s) + · · · + LastLetter(s)

   b. FirstLetter(s) + LastLetter(s) + length(s) (Cichelli)

   c. `for (i = 1, index = 0; i < strlen(s); i++)`
      `index = (26 * index + s[i] - ' ');` (Ramakrishna)

2. Another way of improving the performance of hashing is to allow reorganization of the hash table during insertions. Write a program that compares the performance of linear probing with the following self-organization hashing methods:
   a. *Last-come-first-served hashing* places a new element in its home position, and in case of a collision, the element that occupies this position is inserted in another position using a regular linear probing method to make room for the arriving element (Poblete and Munro 1989).
   b. *Robin Hood hashing* checks the number of positions two colliding keys are away from their home positions and continues searching for an open position for the key closer to its home position (Celis et al. 1985).
3. Write a program that inserts records into a file and retrieves and deletes them using either extendible hashing or the linear hashing technique.
4. Extend the program presented in the case study by creating a linked list of overflowing records associated with each bucket of the intermediate file `outfile`. Note that if a bucket has no empty cells, the search continues in the overflow area. In the extreme case, it may mean that the bucket holds only deleted items and new items are inserted in the overflow area. Therefore, it may be advantageous to have a purging function that, after a certain number of deletions, is automatically invoked. This function transfers items from the overflow area to the main file which are hashed to buckets with deleted items. Write such a function.

Bibliography

Bell, James R. and Kaman, Charles H., "The Linear Quotient Hash Code," *Communications of the ACM* 13 (1970), 675–677.

Celis, P., Larson P., and Munro J. I., "Robin Hood Hashing," *Proceedings of the 26th IEEE Symposium on the Foundations of Computer Science*, Portland, OR, 1985, 281–288.

Cichelli, Richard J., "Minimal Perfect Hash Function Made Simple," *Communications of the ACM* 23 (1980), 17–19.

Czech, Zbigniew J. and Majewski, Bohdan S., "A Linear Time Algorithm for Finding Minimal Perfect Hash Functions," *Computer Journal* 36 (1993), 579–587.

Enbody, R. J. and Dy, H. C., "Dynamic Hashing Schemes," *Computing Surveys* 20 (1988), 85–113.

Fagin, Ronald, Nievergelt, Jurg, Pippenger, Nicholas, and Strong, H. Raymond, "Extendible Hashing—A Fast Access Method for Dynamic Files," *ACM Transactions on Database Systems* 4 (1979), 315–344.

Fox, Edward A., Heath, Lenwood S., Chen, Qi F., and Daoud, Amjad M., "Practical Minimal Perfect Hash Functions for Large Databases," *Communications of the ACM* 35 (1992), 105–121.

Haggard, G. and Karplus, K., "Finding Minimal Perfect Hash Functions," *SIGCSE Bulletin* 18 (1986), No. 1, 191–193.

Knott, G. D., "Expandable Open Addressing Hash Table Storage and Retrieval," *Proceedings of the ACM SIGFIDET Workshop on Data Description, Access, and Control* (1971), 186–206.

Knuth, Donald, *The Art of Computer Programming*, Vol. 3, Reading, MA: Addison-Wesley, 1998.

Larson, Per A., "Dynamic Hashing," *BIT* 18 (1978), 184–201.

Larson, Per A., "Dynamic Hash Tables," *Communications of the ACM* 31 (1988), 446–457.

Lewis, Ted G. and Cook, Curtis R., "Hashing for Dynamic and Static Internal Tables," *IEEE Computer* (October 1986), 45–56.

Litwin, Witold, "Virtual Hashing: A Dynamically Changing Hashing," *Proceedings of the Fourth Conference of Very Large Databases* (1978), 517–523.

Litwin, Witold, "Linear Hashing: A New Tool for File and Table Addressing," *Proceedings of the Sixth Conference of Very Large Databases* (1980), 212–223.

Lomet, David B., "Bounded Index Exponential Hashing," *ACM Transactions on Database Systems* 8 (1983), 136–165.

Lum, V. Y., Yuen, P. S. T., and Dood, M., "Key-to-Address Transformation Techniques: A Fundamental Performance Study on Large Existing Formatted Files," *Communications of the ACM* 14 (1971), 228–239.

Morris, Robert, "Scatter Storage Techniques," *Communications of the ACM* 11 (1968), 38–44.

Mullin, James K., "Tightly Controlled Linear Hashing Without Separate Overflow Storage," *BIT* 21 (1981), 390–400.

Pagli, L., "Self-Adjusting Hash Tables," *Information Processing Letters* 21 (1985), 23–25.

Poblete, Patricio V. and Munro, J. Ian, "Last-Come-First-Served Hashing," *Journal of Algorithms* 10 (1989), 228–248.

Radke, Charles E., "The Use of the Quadratic Search Residue," *Communications of the ACM* 13 (1970), 103–105.

Sager, Thomas J., "A Polynomial Time Generator for Minimal Perfect Hash Functions," *Communications of the ACM* 28 (1985), 523–532.

Sebesta, Robert W. and Taylor, Mark A., "Fast Identification of Ada and Modula-2 Reserved Words," *Journal of Pascal, Ada, and Modula-2* (March/April 1986), 36–39.

Tharp, Alan L., *File Organization and Processing*, New York: Wiley, 1988.

Vitter, Jeffrey S. and Chen, Wen C., *Design and Analysis of Coalesced Hashing*, New York: Oxford University Press, 1987.