10 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 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 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 that can transform a particular key K, be it a string, number, record, or the like, 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 1,000 cells is usually adequate.

But even if this table can accommodate all the variables in the program, how can we design a function h that 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 3,782 cells (for a variable K made out of 31 letters "z," $h(K) = 31 \cdot 122 = 3{,}782$). But even with this size, the function h does not return unique values. For example, h("abc") = h("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, 1,368, can be divided modulo *TSize* or, if the size of the table is 1,000, 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 that 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 = 1,566.

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. An integer in a C++ implementation for the IBM PC computer is 2 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 3,121, then $3{,}121^2 = 9{,}740{,}641$, and for the 1,000-cell table, $h(3{,}121) = 406$, which is the middle part of $3{,}121^2$. 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 1,024, then, in this example, the binary representation of $3{,}121^2$ is the bit string 100101*0101000010*1100001, 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, 1,234; the last four, 6,789; the first two combined with the last two, 1,289; 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 that 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., 1133 for the publishing company Cengage Learning). 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.

10.1.6 Universal Hash Functions

When very little is known about keys, a *universal class of hash functions* can be used; a class of functions is universal when for any sample, a randomly chosen member of that class will be expected to distribute the sample evenly, whereby members of that class guarantee low probability of collisions (Carter and Wegman 1979).

Let H be a class of functions from a set of *keys* to a hash table of *TSize*. We say that H is universal if for any two different keys x and y from *keys*, the number of hash functions h in H for which $h(x) = h(y)$ equals $|H|/TSize$. That is, H is universal if no pair of distinct keys are mapped into the same index by a randomly chosen function h with the probability equal to $1/TSize$. In other words, there is one chance in *TSize* that two keys collide when a randomly picked hash function is applied. One class of such functions is defined as follows.

For a prime number $p \geq |keys|$, and randomly chosen numbers a and b,

$$H = \{h_{a,b}(K): h_{a,b}(K) = ((aK+b) \bmod p) \bmod TSize \text{ and } 0 \leq a, b < p\}$$

Another example is a class H for keys considered to be sequences of bytes, $K = K_0K_1 \ldots K_{r-1}$. For some prime $p \geq 2^8 = 256$ and a sequence $a = a_0, a_1, \ldots, a_{r-1}$,

$$H = \{h_a(K): h_a(K) = \left(\left(\sum_{i=0}^{r-1} a_iK_i\right) \bmod p\right) \bmod TSize \text{ and } 0 \leq a_0, a_1, \ldots, a_{r-1} < p\}$$

10.2 Collision Resolution

Note that straightforward hashing is not without its problems, because 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 that 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 has 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

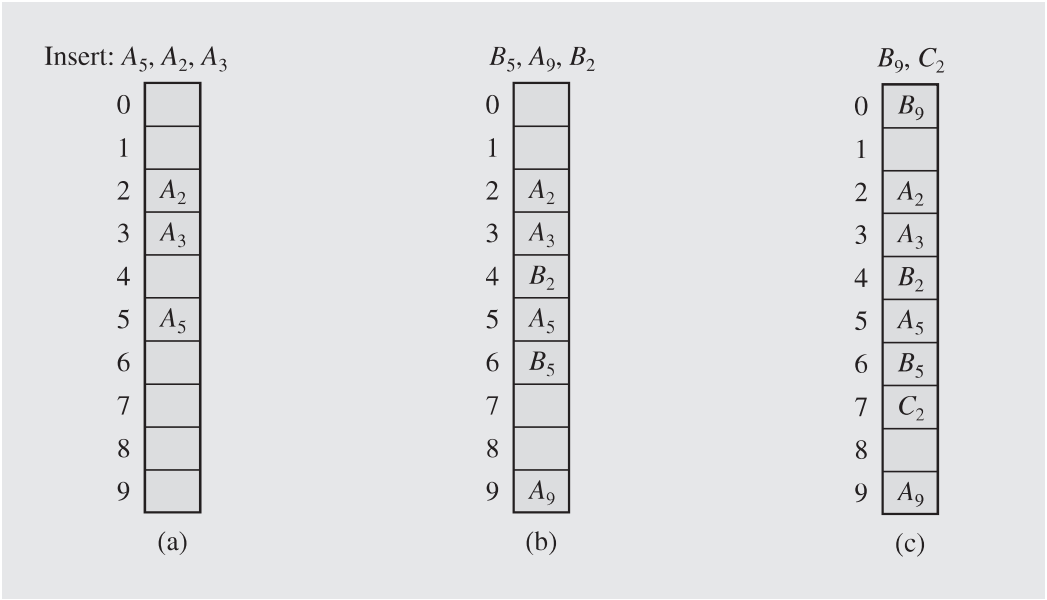
In the open addressing method, when a key collides with another key, the collision is resolved by finding an available table entry other than the position (address) to which the colliding key is originally hashed. If position $h(K)$ is occupied, then the positions in the probing sequence

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

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. Because 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 because 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.

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



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 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:

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

$$h(K) - (TSize - 1)^2/4$$

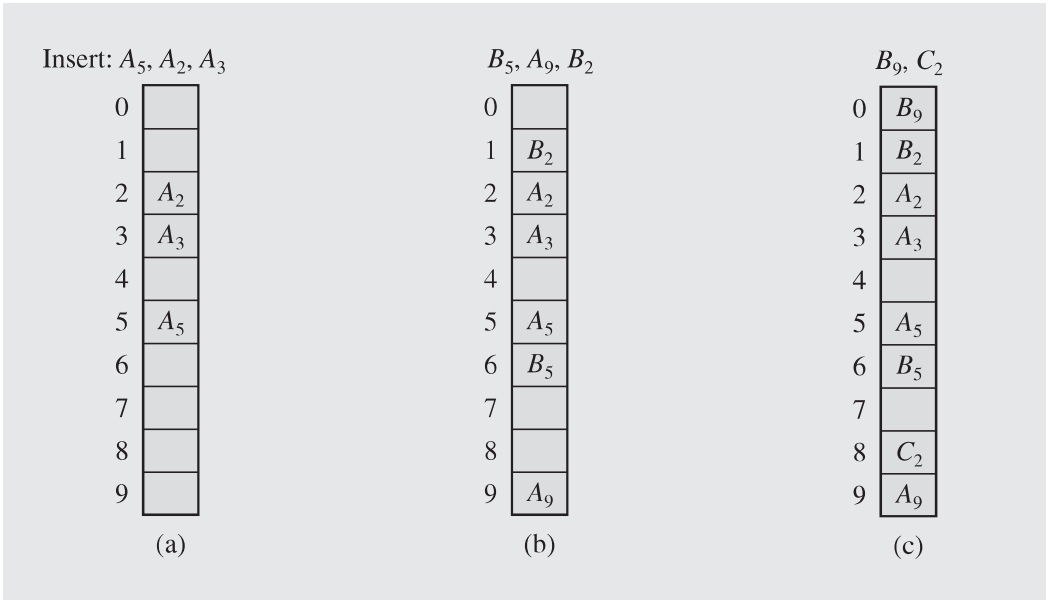
all divided modulo *TSize*. The size of the table should not be an even number, because only the even positions or only the odd positions are tried depending on the value of $h(K)$. Ideally, the table size should be a prime $4j + 3$ of an integer *j*, which

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.

FIGURE 10.2 Using quadratic probing for collision resolution.



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.

This is not an accident. The probes that 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,

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

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

Although using quadratic probing gives much better results than linear probing, the problem of cluster buildup is not avoided altogether, because 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 that 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 $h_p(K) = \texttt{strlen}(K)$, 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

$$j, 2j + 1, 5j + 2, \ldots$$

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

$$2j + 1, 4j + 3, 10j + 11, \ldots$$

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 the 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. 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 $(h(K) - 1) \bmod TSize$. Therefore, the search time increases with the number of elements in the table.

There are formulas that 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 percent 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.

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

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

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

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

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

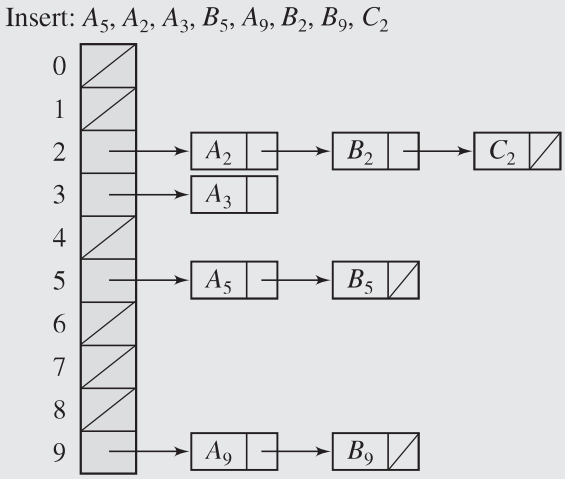
	Linear Probing		Quadratic Probing		Double Hashing	
LF	Successful	Unsuccessful	Successful	Unsuccessful	Successful	Unsuccessful
0.05	1.0	1.1	1.0	1.1	1.0	1.1
0.10	1.1	1.1	1.1	1.1	1.1	1.1
0.15	1.1	1.2	1.1	1.2	1.1	1.2
0.20	1.1	1.3	1.1	1.3	1.1	1.2
0.25	1.2	1.4	1.2	1.4	1.2	1.3
0.30	1.2	1.5	1.2	1.5	1.2	1.4
0.35	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	10.50	200.5	3.5	22.0	3.2	20.0

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 percent full, then linear probing requires, on average, fewer than two trials to find an element in the table. Because this number is usually an acceptable limit for a hash function, linear probing requires 35 percent 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 probing (25 percent) and for double hashing (20 percent), 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.

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, because the linked lists are extended only 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 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).

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



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 that 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* · *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.

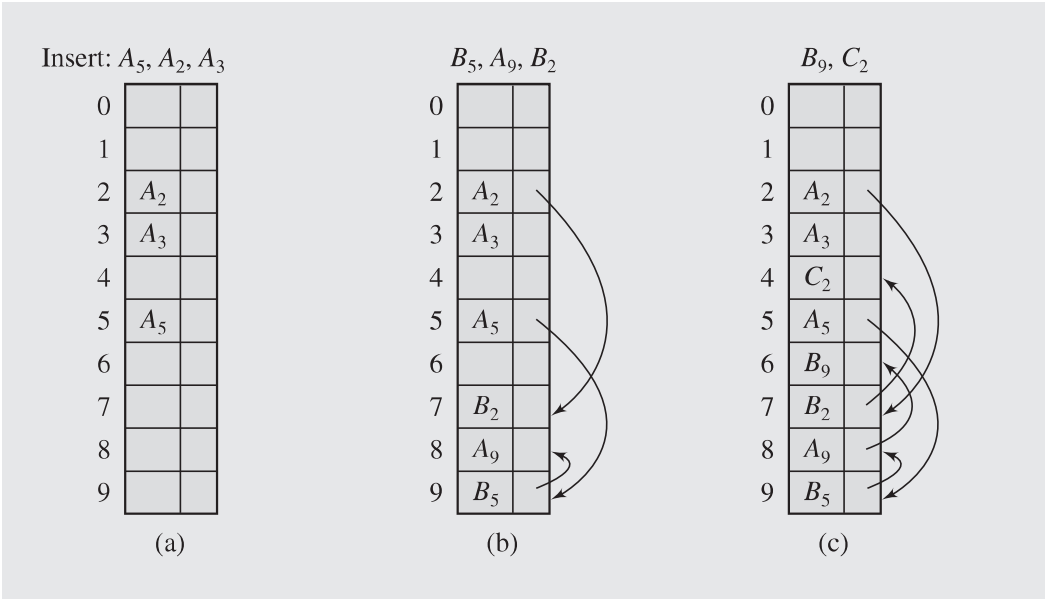
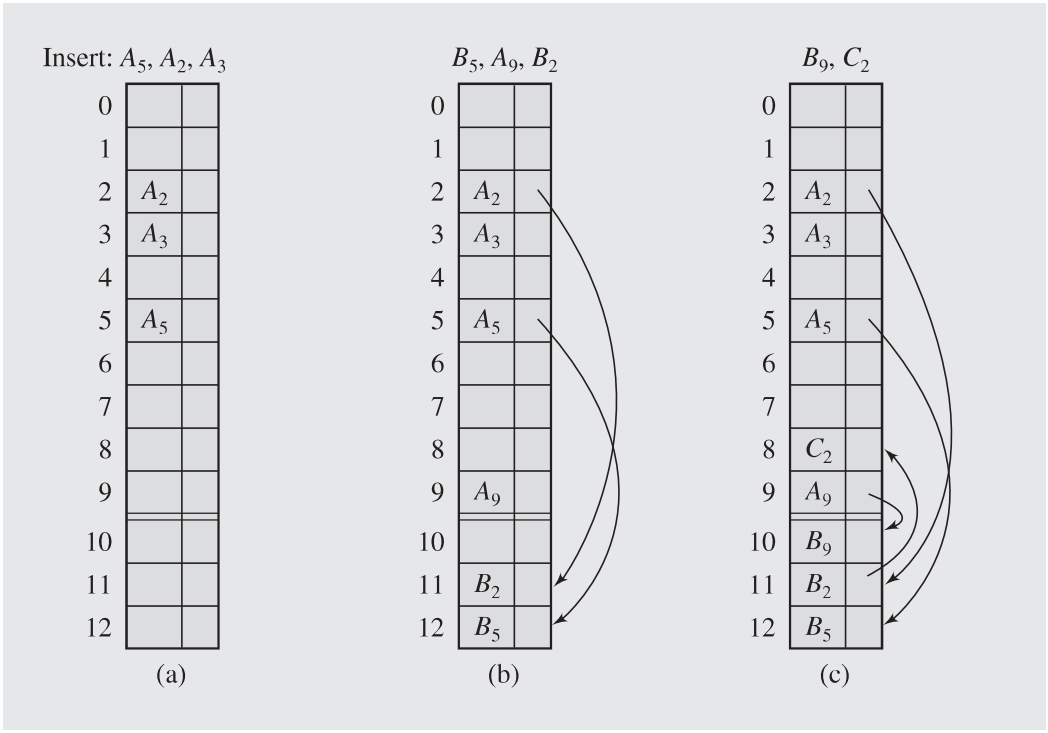


Figure 10.7 illustrates coalesced hashing that uses a cellar. Noncolliding keys are stored in their home positions, as in Figure 10.7a. Colliding keys are put in the last available slot of the cellar and added to the list starting from their home position, as in Figure 10.7b. In Figure 10.7c, the cellar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

FIGURE 10.7 Coalesced hashing that uses a cellar.



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 10.8 Collision resolution with buckets and linear probing method.

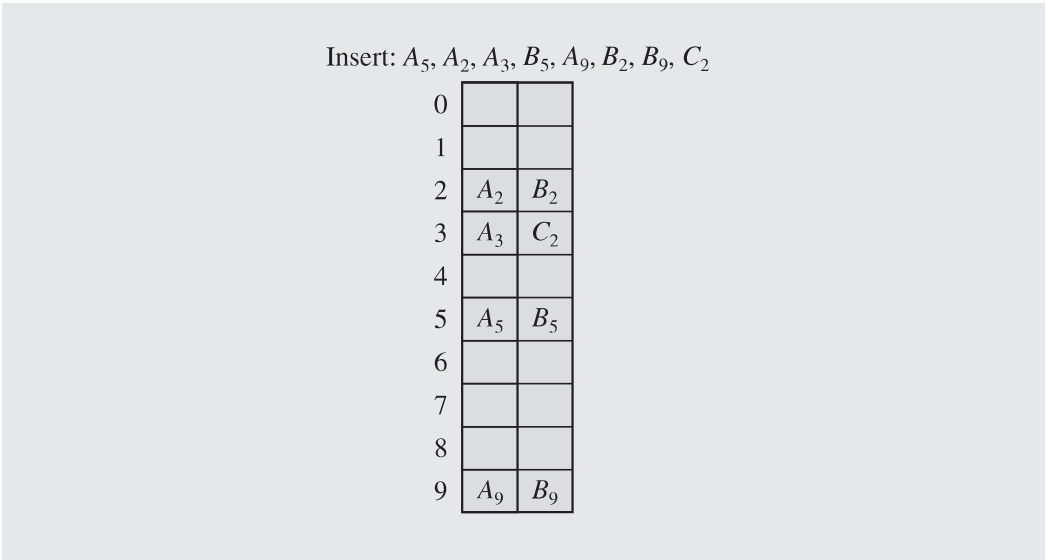
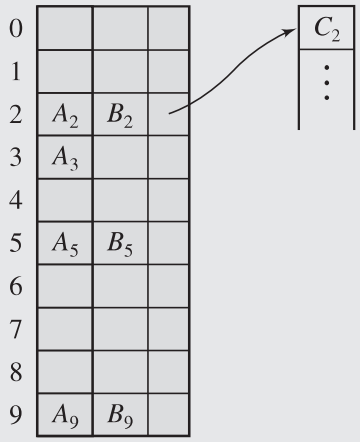


FIGURE 10.9 Collision resolution with buckets and overflow area.

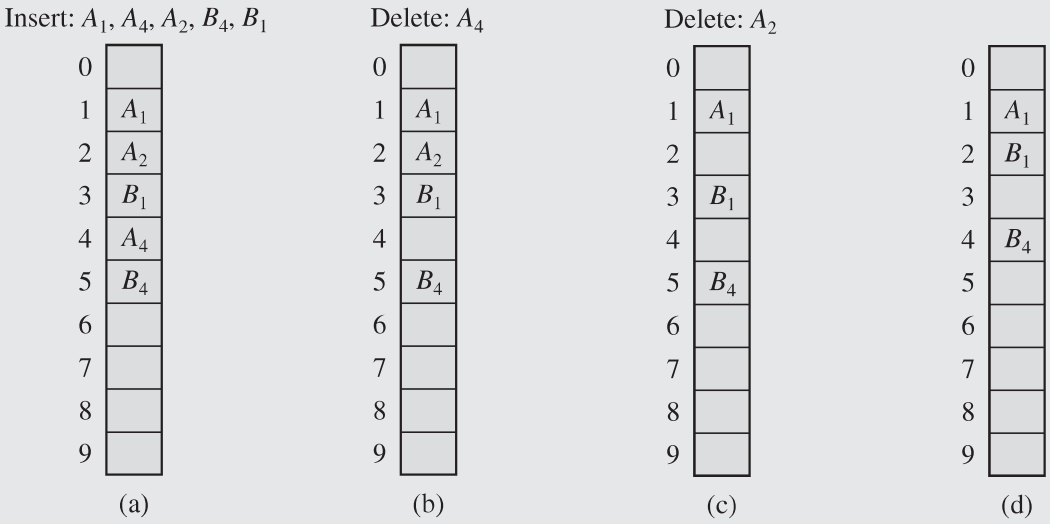


10.3 Deletion

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, because if we are using linear probing, the search terminates at position 2. The situation is the same for the other open addressing methods.

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



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 that 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 that are not overwritten by this procedure are marked as free. Figure 10.10d illustrates this situation.

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, because

sooner or later, a key would arrive that 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 letter 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);
         assign firstLetterValue and/or lastLetterValue as g-values of firstletter(word)
         and/or lastletter(word) if they are not -1;
         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 because 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

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}

function that reduced the run time of a Pascal cross-reference program by 10 percent 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

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

(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 $c_1c_2 \ldots c_m$ 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.

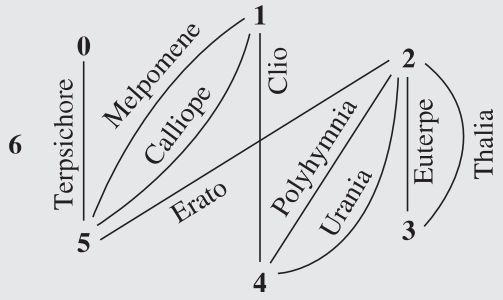
The ordering step rearranges all the vertices so that they can be partitioned into a series of levels. When a sequence $v_1, \ldots, v_t$ of vertices is established, then a level $K(v_i)$ of keys is defined as a set of all the edges that connect v_i with those v_js for which $j \leq i$. The sequence 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

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)

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, because it is equal to some v_j that 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. 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, because $h_2 = v_2$ 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, because position two is already occupied. Assume that this number is 4. As a result, $h(\text{Calliope}) = 1$ and $h(\text{Melpomene}) = 4$. 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, because 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 Rehashing

When a table becomes full, insertion of new elements becomes impossible and when it reaches a certain saturation level, hashing becomes slow, requiring many tries to locate an item. One solution is rehashing, that is, allocating a large table, modifying the hash function (at least, *TSize*), hashing all items from the old table to the new table, and discarding the old table and then using the new table for hashing with the new hash function. The new table can be a double size of the old, its size can be a prime closest to the double of current size, it can be the current size plus a predefined value or the value can be randomly chosen, etc. All the methods described thus far can use rehashing, after which they can continue processing data using the same methods of hashing and collision resolution. One method for which rehashing is particularly important is the *cuckoo hashing* (Pagh and Rodler 2004).

10.5.1 The cuckoo hashing

The cuckoo hashing uses two tables, T_1 and T_2, and two hash functions, h_1 and h_2. To insert a key K_1, table T_1 is checked with function h_1. If position $T_1[h_1(K_1)]$ is free, the key is inserted there. If the position is occupied by a key K_2, then K_2 is removed to make room for K_1 and then an attempt is made to place K_2 with the second hash function in the second table in position $T_2[h_2(K_2)]$. If this position is occupied by a

key K_3, then K_3 is moved out to make room for K_2 and then an attempt is made to place K_3 in position $T_1[h_1(K_3)]$. So the arriving key (the original key or the key being pushed out to the opposite table) has a priority over a key occupying the former's position. In theory, such a sequence of attempts can lead to an infinite loop if the very first position that was tried is tried again. Also, the sequence of tries may not be successful since both tables are full. To avoid the problem, a limit on tries is set and then if it is exceeded, rehashing is performed by creating two new and larger tables, defining two new hash functions and rehashing keys from the old tables to the new ones. If along the way the limit on the number of tries is exceeded, new rehashing takes place by creating tables larger yet with new hash functions. The algorithm can be summarized as follows:

```
insert(K)
    if K is already in T1[h1(K)] or in T2[h2(K)]
        do nothing;
    for i = 0 to maxLoop-1
        swap(K,T1[h1(K)]);
        if K is null
            return;
        swap(K,T2[h2(K)]);
        if K is null
            return;
    rehash();
    insert(K);
```

An example of application of the algorithm is given in Figure 10.13. Key 2 is inserted upon the first try in T_1 in position $T_1[h_1(2)]$ (Figure 10.13a). Because $h_1(2) = h_1(7)$, key 7 is in conflict with key 2, thus 2 is ousted to make room for 7 and then 2 is inserted in position $T_2[h_2(2)]$ (Figure 10.13b). Next 12 wants to be inserted in the position occupied by 7, so 7 is replaced by 12 (Figure 10.13c), and 7 is transferred to table T_2 to the position already given to 2, so 7 replaces 2 (Figure 10.13d) and 2 is transferred to its original position in table T_1 replacing 12 (Figure 10.13e). If $h_2(12)$ happens to be equal to $h_2(7)$, then 12 pushes 7 out, takes its place (Figure 10.13f), and 7 is hashed to where it already was, namely in position $T_1[h_1(7)]$ occupied by 2, forcing 2 out (Figure 10.13g). Key 2 finds itself in the same position in table T_2 in which it already resided, whereby 12 loses its cell (Figure 10.13h). In this way, a full circle is made and the situation that follows is the same as in Figure 10.13c so that continuation of the execution would lead to an infinite loop. The loop is broken by a timeout determined by the value of `maxLoop`, after which rehashing takes place by allocating new tables, creating new hash functions, and using these functions to hash all the keys to the new tables. If `maxLoop` happens to be set up so that looping is finished now, rehashing places 12 in the new table (Figure 10.13i), then inserting 2 leads to moving 12 to table T_2 and the subsequent insertion of 7 is done after the first try (Figure 10.13j). After key 2 is deleted (Figure 10.13k), key 13 is inserted by replacing 7; 7 replaces 12, and finally 12 ends up in the position previously freed by 2 (Figure 10.13l).

Searching for a key K in this scheme requires making only one test—to check position $T_1[h_1(K)]$, or two tests—to also check position $T_2[h_2(K)]$. Note that rehashing

may be limited to creating only new hash functions and only processing keys in the existing tables. But this would also be a global operation requiring a complete pass through both tables.

FIGURE 10.13 An example of the application of cuckoo hashing.