EXHIBIT 5

PART 3 OF 6

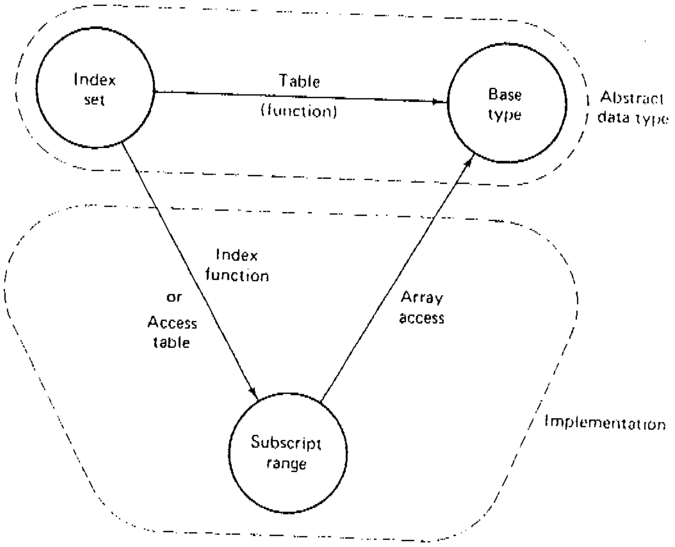

Figure 6.9. Implementation of a table

functions have no such order. (If the index set has some natural order, then sometimes this order is reflected in the table, but this is not a necessary aspect of using tables.) Hence information retrieval from a list naturally involves a search like the ones studied in the previous chapter, but information retrieval from a table requires different methods, access methods that go directly to the desired entry. The time required for searching a list generally depends on the number n of items in the list and is at least $\lg n$, but the time for accessing a table does not usually depend on the number of items in the table; that is, it is usually $O(1)$. For this reason, in many applications table access is significantly faster than list searching.

retrieval

On the other hand, traversal is a natural operation for a list but not for a table. It is generally easy to move through a list performing some operation with every item in the list. In general, it may not be nearly so easy to perform an operation on every item in a table, particularly if some special order for the items is specified in advance.

traversal

Finally, we should clarify the distinction between the terms *table* and *array*. *In general, we shall use table* as we have defined it in this section and restrict the term *array* to mean the programming feature available in Pascal and most high-level languages and used for implementing both tables and contiguous lists.

tables and arrays

6.5 HASHING

6.5.1 Sparse Tables

1. Index Functions

We can continue to exploit table lookup even in situations where the key is no longer an index that can be used directly as in array indexing. What we can do is to set up a one-to-one correspondence between the keys by which we wish to tion and indices that we can use to access an array. The index function that we produce will be somewhat more complicated than those of previous sections, since it may need to convert the key from, say, alphabetic information to an integer, but in principle it can still be done.

The only difficulty arises when the number of possible keys exceeds the amount of space available for our table. If, for example, our keys are alphabetical words of eight letters, then there are $26^8 \approx 2 \times 10^{11}$ possible keys, a number much greater than the number of positions that will be available in high-speed memory. In practice, however, only a small fraction of these keys will actually occur. That is, the table is *sparse*. Conceptually, we can regard it as indexed by a very large set, but with relatively few positions actually occupied. In Pascal, for example, we might think in terms of conceptual declarations such as

type ··· = **sparse table** [keytype] **of** item.

Even though it may not be possible to implement a declaration such as this directly, it is often helpful in problem solving to begin with such a picture, and only slowly tie down the details of how it is put into practice.

2. Hash Tables

index function not one-to-one

The idea of a *hash table* (such as the one shown in Figure 6.10) is to allow many of the different possible keys that might occur to be mapped to the same location in an array under the action of the index function. Then there will be a possibility that two records will want to be in the same place, but if the number of records that actually occur is small relative to the size of the array, then this possibility will cause little loss of time. Even when most entries in the array are occupied, hash methods can be an effective means of information retrieval.

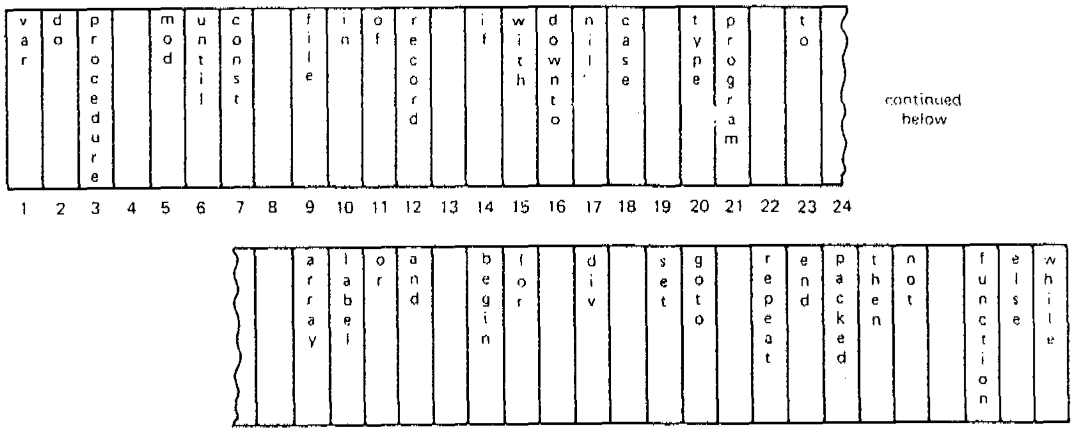

Figure 6.10. A hash table

hash function

We begin with a ***hash function*** that takes a key and maps it to some index in the array. This function will generally map several different keys to the same index.

collision

If the desired record is in the location given by the index, then our problem is solved; otherwise, we must use some method to resolve the ***collision*** that may have occurred between two records wanting to go to the same location. There are thus two questions we must answer to use hashing. First, we must find good hash functions, and, second, we must determine how to resolve collisions.

Before approaching these questions, let us pause to outline informally the steps needed to implement hashing.

3. Algorithm Outlines

keys in table

First, an array must be declared that will hold the hash table. With ordinary arrays, the keys used to locate entries are usually the indices, so there is no need to keep them within the array itself, but for a hash table, several possible keys will correspond to the same index, so one field within each record in the array must be reserved for the key itself.

initialization

Next, all locations in the array must be initialized to show that they are empty. How this is done depends on the application: often it is accomplished by setting the key fields to some value that is guaranteed never to occur as an actual key. With alphanumeric keys, for example, a key consisting of all blanks might represent an empty position.

insertion

To insert a record into the hash table, the hash function for the key is first calculated. If the corresponding location is empty, then the record can be inserted, else if the keys are equal, then insertion of the new record would not be allowed, and in the remaining case (a record with a different key is in the location), it becomes necessary to resolve the collision.

retrieval

To retrieve the record with a given key is entirely similar. First, the hash function for the key is computed. If the desired record is in the corresponding location, then the retrieval has succeeded; otherwise, while the location is nonempty and not all locations have been examined, follow the same steps used for collision resolution. If an empty position is found, or all locations have been considered, then no record with the given key is in the table, and the search is unsuccessful.

6.5.2 Choosing a Hash Function

The two principal criteria in selecting a hash function are that it should be easy and quick to compute and that it should achieve an even distribution of the keys that actually occur across the range of indices. If we know in advance exactly what keys will occur, then it is possible to construct hash functions that will be very efficient, but generally we do not know in advance what keys will occur. Therefore,

method

the usual way is for the hash function to take the key, chop it up, mix the pieces together in various ways, and thereby obtain an index that (like the pseudorandom numbers generated by computer) will be uniformly distributed over the range of indices.

It is from this process that the word *hash* comes, since the process converts the key into something that bears little resemblance. At the same time, it is hoped that any patterns or regularities that may occur in the keys will be destroyed, so that the results will be randomly distributed.

Even though the term *hash* is very descriptive, in some books the more technical terms *scatter-storage* or *key-transformation* are used in its place.

We shall consider three methods that can be put together in various ways to build a hash function.

1. Truncation

Ignore part of the key, and use the remaining part directly as the index (considering non-numeric fields as their numerical codes). If the keys, for example, are eight-digit integers and the hash table has 1000 locations, then the first, second, and fifth digits from the right might make the hash function, so that 62538194 maps to 394. Truncation is a very fast method, but it often fails to distribute the keys evenly through the table.

2. Folding

Partition the key into several parts and combine the parts in a convenient way (often using addition or multiplication) to obtain the index. For example, an eight-digit integer can be divided into groups of three, three, and two digits, the groups added together, and truncated if necessary to be in the proper range of indices. Hence 62538194 maps to 625 + 381 + 94 = 1100, which is truncated to 100. Since all information in the key can affect the value of the function, folding often achieves a better spread of indices than does truncation by itself.

3. Modular Arithmetic

Convert the key to an integer (using the above devices as desired), divide by the size of the index range, and take the remainder as the result. This amounts to using the Pascal operator **mod**. The spread achieved by taking a remainder depends very much on the modulus (in this case, the size of the hash array). If the modulus is a power of a small integer like 2 or 10, then many keys tend to map to the same index, while other indices remain unused. The best choice for modulus is a prime number, which usually has the effect of spreading the keys quite uniformly. (We shall see later that a prime modulus also improves an important method for collision resolution.) Hence, rather than choosing a hash table size of 1000, it is better to choose either 997 or 1009; 1024 = 2^{10} would usually be a poor choice. Taking the remainder is usually the best way to conclude calculating the hash function, since it can achieve a good spread at the same time that it ensures that the result is in the proper range. About the only reservation is that, on a tiny machine with no hardware division, the calculation can be slow, so other methods should be considered.

prime modulus

4. Pascal Example

As a simple example, let us write a hash function in Pascal for transforming a key consisting of eight alphanumeric characters into an integer in the range

0 . . hashsize − 1.

That is, we shall begin with the type

```
type keytype = array[1 . . 8] of char;
```

We can then write a simple hash function as follows:

sample hash function

```
function Hash(x: keytype): integer;
var
  i: 1 .. 8;
  h: integer;
begin
  h := 0;
  for i := 1 to 8 do
    h := h + ord(x[i]);
  Hash := h mod hashsize
end;
```

We have simply added the integer codes corresponding to each of the eight characters. There is no reason to believe that this method will be better (or worse), however, than any number of others. We could, for example, subtract some of the codes, multiply them in pairs, or ignore every other character. Sometimes an application will suggest that one hash function is better than another; sometimes it requires experimentation to settle on a good one.

6.5.3 Collision Resolution with Open Addressing

1. Linear Probing

The simplest method to resolve a collision is to start with the hash address (the location where the collision occurred) and do a sequential search for the desired key or an empty location. Hence this method searches in a straight line, and it is therefore called ***linear probing.*** The array should be considered circular, so that when the last location is reached, the search proceeds to the first location of the array.

2. Clustering

example of clustering

The major drawback of linear probing is that, as the table becomes about half full, there is a tendency toward ***clustering;*** that is, records start to appear in long strings of adjacent positions with gaps between the strings. Thus the sequential searches needed to find an empty position become longer and longer. For consider the example in Figure 6.11, where the occupied positions are shown in color. Suppose that there are n locations in the array and that the hash function chooses any of them with equal probability $1/n$. Begin with a fairly uniform spread, as shown in the top diagram. If a new insertion hashes to location b, then it will go there, but if it hashes to location a (which is full), then it will also go into b. Thus the probability that b will be filled has doubled to $2/n$. At the next stage, an attempted insertion into any of locations a, b, c, or d will end up in d, so the probability of filling d is $4/n$. After this, e has probability $5/n$ of being filled, and so as additional insertions are made the most likely effect is to make the string of full positions beginning at location a longer and longer, and hence the performance of the hash table starts to degenerate toward that of sequential search.

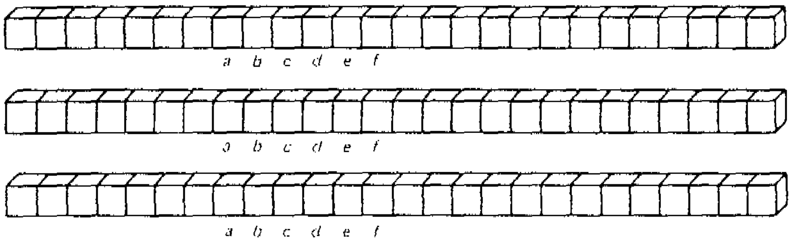

Figure 6.11. Clustering in a hash table

instability

The problem of clustering is essentially one of instability; if a few keys happen randomly to be near each other, then it becomes more and more likely that other keys will join them, and the distribution will become progressively more unbalanced.

3. Increment Functions

rehashing

If we are to avoid the problem of clustering, then we must use some more sophisticated way to select the sequence of locations to check when a collision occurs. There are many ways to do so. One, called ***rehashing***, uses a second hash function to obtain the second position to consider. If this position is filled, then some other method is needed to get the third position, and so on. But if we have a fairly good spread from the first hash function, then little is to be gained by an independent second hash function. We will do just as well to find a more sophisticated way of determining the distance to move from the first hash position and apply this method, whatever the first hash location is. Hence we wish to design an increment function that can depend on the key or on the number of probes already made and that will avoid clustering.

4. Quadratic Probing

If there is a collision at hash address h, this method probes the table at locations $h + 1$, $h + 4$, $h + 9$, . . . , that is, at locations $h + i^2$ (mod hashsize) for $i = 1, 2, \ldots$. That is, the increment function is i^2.

This method substantially reduces clustering, but it is not obvious that it will probe all locations in the table, and in fact it does not. If hashsize is a power of 2, then relatively few positions are probed. Suppose that hashsize is a prime. If we reach the same location at probe i and at probe j, then

$$h + i^2 \equiv h + j^2 \text{ (mod hashsize)}$$

so that

$$(i - j)(i + j) \equiv 0 \text{ (mod hashsize)}.$$

Since hashsize is a prime, it must divide one factor. It divides $i - j$ only when j differs from i by a multiple of hashsize, so at least hashsize probes have been made. Hashsize divides $i + j$, however, when j = hashsize $- i$, so the total number of distinct positions that will be probed is exactly

number of distinct probes

$$\text{(hashsize + 1) } \textbf{div } 2.$$

It is customary to take overflow as occurring when this number of positions has been probed, and the results are quite satisfactory.

calculation

Note that quadratic probing can be accomplished without doing multiplications: After the first probe at position x, the increment is set to 1. At each successive probe, the increment is increased by 2 after it has been added to the previous location. Since

$$1 + 3 + 5 + \cdots + (2i - 1) = i^2$$

for all $i \geq 1$ (you can prove this fact by mathematical induction), probe i will look in position

$$x + 1 + 3 + \cdots + (2i - 1) = x + i^2,$$

as desired.

5. Key-Dependent Increments

Rather than having the increment depend on the number of probes already made, we can let it be some simple function of the key itself. For example, we could truncate the key to a single character and use its code as the increment. In Pascal, we might write

```
increment := ord(k[1]).
```

A good approach, when the remainder after division is taken as the hash function, is to let the increment depend on the quotient of the same division. An optimizing compiler should specify the division only once, so the calculation will be fast, and the results generally satisfactory.

In this method, the increment, once determined, remains constant. If hashsize is a prime, it follows that the probes will step through all the entries of the array before any repetitions. Hence overflow will not be indicated until the array is completely full.

6. Random Probing

A final method is to use a pseudorandom number generator to obtain the increment. The generator used should be one that always generates the same sequence provided it starts with the same seed. The seed, then, can be specified as some function of the key. This method is excellent in avoiding clustering, but is likely to be slower than the others.

7. Pascal Algorithms

To conclude the discussion of open addressing, we continue to study the Pascal example already introduced, which used alphanumeric keys of the type

```
type keytype = array[1 . . 8] of char.
```

We set up the hash table with the declarations

rations

```
const
  hashsize = 997;                          {a prime number of appropriate size}
  hashmax = 996;                           {should be 1 less than hashsize}
type
  hashtable = array[0 .. hashmax] of item;
var
  H:           hashtable;
```

initialization

The hash table must be initialized by defining a special key called blankword that consists of eight blanks and setting the key field of each item in H to blankword.

We shall use the hash function already written in Section 6.5 2, part 4, together with quadratic probing for collision resolution. We have shown that the maximum number of probes that can be made this way is (hashsize − 1) **div** 2, and we keep a counter c to check this upper bound.

With these conventions, let us write a procedure to insert a record r, with key r.key, into the hash table H.

ion

```
procedure Insert(var H: hashtable; r: item);
var
  c,                                       {counter to be sure that table is not full}
  i,                                       {increment used for quadratic probing}
  p: integer;                              {position currently probed in H}
begin                                      {procedure Insert}
  p := Hash(r.key);
  c := 0;
  i := 1;
  while (H[p].key <> blankword)            {Is the location empty?}
    and (H[p].key <> r.key)                {Has the target key been found?}
    and (c <= hashsize div 2) do           {Has overflow occurred?}
    begin
      c := c + 1;
      p := p + i;
      i := i + 2;                          {Prepare increment for the next iteration.}
      if p > hashmax then
        p := p mod hashsize
    end;
  if H[p].key = blankword then
    H[p] := r                              {Insert the new item r.}
  else if H[p].key = r.key then
    Error                                  {The same key cannot appear twice.}
  else
    Overflow                               {Counter has reached its limit.}
end;                                       {procedure Insert}
```

quadratic probing

A procedure to retrieve the record (if any) with a given key will have a similar form and is left as an exercise.

8. Deletions

Up to now we have said nothing about deleting items from a hash table. At first glance, it may appear to be an easy task, requiring only marking the deleted location with the special key indicating that it is empty. This method will not work. The reason is that an empty location is used as the signal to stop the search for a target key. Suppose that, before the deletion, there had been a collision or two and that some item whose hash address is the now-deleted position is actually stored elsewhere in the table. If we now try to retrieve that item, then the now-empty position will stop the search, and it is impossible to find the item, even though it is still in the table.

special key

One method to remedy this difficulty is to invent another special key, to be placed in any deleted position. This special key would indicate that this position is free to receive an insertion when desired but that it should not be used to terminate the search for some other item in the table. Using this second special key will, however, make the algorithms somewhat more complicated and a bit slower. With the methods we have so far studied for hash tables, deletions are indeed awkward and should be avoided as much as possible.

6.5.4 Collision Resolution by Chaining

linked storage

Up to now we have implicitly assumed that we are using only contiguous storage while working with hash tables. Contiguous storage for the hash table itself is, in fact, the natural choice, since we wish to be able to refer quickly to random position in the table, and linked storage is not suited to random access. There is, however, no reason why linked storage should not be used for the records themselves. We can take the hash table itself as an array of pointers to the records, that is, as an array of list headers. An example appears in Figure 6.12.

It is traditional to refer to the linked lists from the hash table as *chains* and call this method collision resolution by *chaining*.

1. Advantages of Linked Storage

space saving

There are several advantages to this point of view. The first, and the most important when the records themselves are quite large, is that considerable space may be saved. Since the hash table is a contiguous array, enough space must be set aside at compilation time to avoid overflow. If the records themselves are in the hash table, then if there are many empty positions (as is desirable to help avoid the cost of collisions), these will consume considerable space that might be needed elsewhere. If, on the other hand, the hash table contains only pointers to the records, pointers that require only one word each, then the size of the hash table may be reduced by a large factor (essentially by a factor equal to the size of the records), and will become small relative to the space available for the records, or for other uses.

collision resolution

The second major advantage of keeping only pointers in the hash table is that it allows simple and efficient collision handling. We need only add a link field to each record, and organize all the records with a single hash address as a linked list. With a good hash function, few keys will give the same hash address, so the

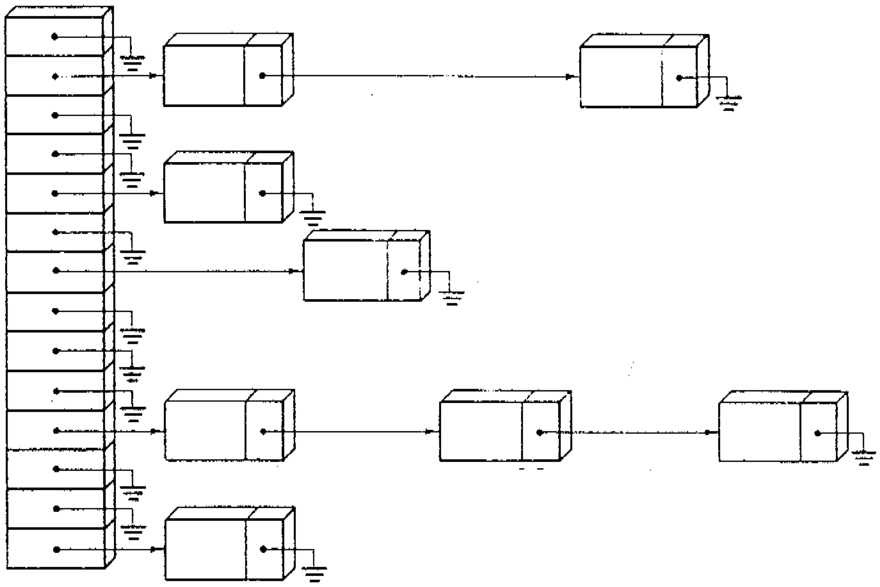

Figure 6.12. A chained hash table

linked lists will be short and can be searched quickly. Clustering is no problem at all, because keys with distinct hash addresses always go to distinct lists.

overflow

A third advantage is that it is no longer necessary that the size of the hash table exceed the number of records. If there are more records than entries in the table, it means only that some of the linked lists are now sure to contain more than one record. Even if there are several times more records than the size of the table, the average length of the linked lists will remain small, and sequential search on the appropriate list will remain efficient.

deletion

Finally, deletion becomes a quick and easy task in a chained hash table. Deletion proceeds in exactly the same way as deletion from a simple linked list.

2. Disadvantage of Linked Storage

of space

These advantages of chained hash tables are indeed powerful. Lest you believe that chaining is always superior to open addressing, however, let us point out one important disadvantage: All the links require space. If the records are large, then this space is negligible in comparison with that needed for the records themselves; but if the records are small, then it is not.

ll records

Suppose, for example, that the links take one word each and that the items themselves take only one word (which is the key alone). Such applications are quite common, where we use the hash table only to answer some yes-no question about the key. Suppose that we use chaining and make the hash table itself quite small, with the same number n of entries as the number of items. Then we shall use $3n$ words of storage altogether: n for the hash table, n for the keys, and n for the links to find the next node (if any) on each chain. Since the hash table will be nearly full, there will be many collisions, and some of the chains will have several items.

Hence searching will be a bit slow. Suppose, on the other hand, that we use open addressing. The same $3n$ words of storage put entirely into the hash table will mean that it will be only one third full, and therefore there will be relatively few collisions and the search for any given item will be faster.

3. Pascal Algorithms

A chained hash table in Pascal takes declarations like

declarations

```
type
  pointer   = ↑node;
  list      = record head: pointer end;
  hashtable = array [0 .. hashmax] of list;
```

The record type called node consists of an item, called info, and an additional field, called next, that points to the next node on a linked list.

The code needed to initialize the hash table is

initialization

```
for i := 0 to hashmax do H[i].head := nil;
```

We can even use previously written procedures to access the hash table. The hash function itself is no different from that used with open addressing; for data retrieval we can simply use the procedure SequentialSearch (linked version) from Section 5.2, as follows:

retrieval

```
procedure Retrieve(var H: hashtable; target: keytype;
                        var found: Boolean; var location: pointer);
{finds the node with key target in the hash table H, and returns with location
  pointing to that node, provided that found becomes true}
begin
  SequentialSearch(H[Hash(target)], target, found, location)
end;
```

Our procedure for inserting a new entry will assume that the key does not appear already; otherwise, only the most recent insertion with a given key will be retrievable.

insertion

```
procedure Insert(var H: hashtable; p: pointer);
{inserts node p↑ into the chained hash table H, assuming no other node with
  key p↑.info.key is in the table}
var
  i: integer;                              {used for index in hash table}
begin
  i := Hash(p↑.info.key);           {Find the index of the linked list for p↑.}
  p↑.next := H[i].head;                     {Insert p↑ at the head of the list.}
  H[i].head := p                   {Set the head of the list to the new item.}
end;
```

As you can see, both of these procedures are significantly simpler than are the versions for open addressing, since collision resolution is not a problem.

Exercises 6.5

E1. Write a Pascal procedure to insert an item into a hash table with open addressing and linear probing.

E2. Write a Pascal procedure to retrieve an item from a hash table with open addressing and (a) linear probing; (b) quadratic probing.

E3. Devise a simple, easy-to-calculate hash function for mapping three-letter words to integers between 0 and $n - 1$, inclusive. Find the values of your function on the words

PAL LAP PAM MAP PAT PET SET SAT TAT BAT

for $n = 11, 13, 17, 19$. Try for as few collisions as possible.

E4. Suppose that a hash table contains hashsize = 13 entries indexed from 0 through 12 and that the following keys are to be mapped into the table:

10 100 32 45 58 126 3 29 200 400 0.

(a) Determine the hash addresses and find how many collisions occur when these keys are reduced **mod** hashsize.

(b) Determine the hash addresses and find how many collisions occur when these keys are first folded by adding their digits together (in ordinary decimal representation) and then reducing **mod** hashsize.

perfect hash functions

(c) Find a hash function that will produce no collisions for these keys. (A hash function that has no collisions for a fixed set of keys is called ***perfect.***)

(d) Repeat the previous parts of this exercise for hashsize = 11. (A hash function that produces no collision for a fixed set of keys that completely fill the hash table is called ***minimal perfect.***)

E5. Another method for resolving collisions with open addressing is to keep a separate array called the ***overflow table,*** into which all items that collide with an occupied location are put. They can either be inserted with another hash function or simply inserted in order, with sequential search used for retrieval. Discuss the advantages and disadvantages of this method.

E6. Write an algorithm for deleting a node from a chained hash table.

E7. Write a deletion algorithm for a hash table with open addressing, using a second special key to indicate a deleted item (see part 8 of Section 6.5.3). Change the retrieval and insertion algorithms accordingly.

E8. With linear probing, it is possible to delete an item without using a second special key, as follows. Mark the deleted entry empty. Search until another empty position is found. If the search finds a key whose hash address is at or before the first empty position, then move it back there, make its previous position empty, and continue from the new empty position. Write an algorithm to implement this method. Do the retrieval and insertion algorithms need modification?

Programming Project 6.5

P1. Consider the 35 Pascal reserved words listed in Appendix C.2.1. Consider these words as strings of nine characters, where words less than nine letters long are filled with blanks on the right.

(a) Devise an integer-valued function that will produce different values when applied to all 35 reserved words. [You may find it helpful to write a short program to assist. Your program could read the words from a file, apply the function you devise, and determine what collisions occur.]

(b) Find the smallest integer hashsize such that, when the values of your function are reduced **mod** hashsize, all 35 values remain distinct.

(c) Modify your function as necessary until you can achieve hashsize = 35 in the preceding part. (You will then have discovered a *minimal perfect* hash function for the 35 Pascal reserved words.)

6.6 ANALYSIS OF HASHING

1. The Birthday Surprise

The likelihood of collisions in hashing relates to the well-known mathematical diversion: How many randomly chosen people need to be in a room before it becomes likely that two people will have the same birthday (month and day)? Since (apart from leap years) there are 365 possible birthdays, most people guess that the answer will be in the hundreds, but in fact, the answer is only 24 people.

We can determine the probabilities for this question by answering its opposite: With m randomly chosen people in a room, what is the probability that no two have the same birthday? Start with any person, and check his birthday off on a calendar. The probability that a second person has a different birthday is 364/365. Check it off. The probability that a third person has a different birthday is now 363/365. Continuing this way, we see that if the first $m - 1$ people have different birthdays, then the probability that person m has a different birthday is

$$(365 - m + 1)/365.$$

Since the birthdays of different people are independent, the probabilities multiply, and we obtain that the probability that m people all have different birthdays is

probability

$$\frac{364}{365} \times \frac{363}{365} \times \frac{362}{365} \times \cdots \times \frac{365 - m + 1}{365}.$$

This expression becomes less than 0.5 whenever $m \geq 24$.

collisions likely

In regard to hashing, the birthday surprise tells us that with any problem of reasonable size, we are almost certain to have some collisions. Our approach, therefore, should not be only to try to minimize the number of collisions, but also to handle those that occur as expeditiously as possible.

2. Counting Probes

As with other methods of information retrieval, we would like to know how many comparisons of keys occur on average during both successful and unsuccessful attempts to locate a given target key. We shall use the word *probe* for looking at one item and comparing its key with the target.

The number of probes we need clearly depends on how full the table is. Therefore (as for searching methods), we let n be the number of items in the table, and we let t (which is the same as hashsize) be the number of positions in the array. The

load factor **load factor** of the table is $\lambda = n/t$. Thus $\lambda = 0$ signifies an empty table; $\lambda = 0.5$ a table that is half full. For open addressing, λ can never exceed 1, but for chaining, there is no limit on the size of λ. We consider chaining and open addressing separately.

3. Analysis of Chaining

With a chained hash table we go directly to one of the linked lists before doing any probes. Suppose that the chain that will contain the target (if it is present) has k items.

unsuccessful retrieval If the search is unsuccessful, then the target will be compared with all k of the corresponding keys. Since the items are distributed uniformly over all t lists (equal probability of appearing on any list), the expected number of items on the one being searched is $\lambda = n/t$. Hence the average number of probes for an unsuccessful search is λ.

successful retrieval Now suppose that the search is successful. From the analysis of sequential search, we know that the average number of comparisons is $\frac{1}{2}(k + 1)$, where k is the length of the chain containing the target. But the expected length of this chain is no longer λ, since we know in advance that it must contain at least one node (the target). The $n - 1$ nodes other than the target are distributed uniformly over all t chains; hence the expected number on the chain with the target is $1 + (n - 1)/t$. Except for tables of trivially small size, we may approximate $(n - 1)/t$ by $n/t = \lambda$. Hence the average number of probes for a successful search is very nearly

$$\tfrac{1}{2}(k + 1) \approx \tfrac{1}{2}(1 + \lambda + 1) = 1 + \tfrac{1}{2}\lambda.$$

4. Analysis of Open Addressing

For our analysis of the number of probes done in open addressing, let us first ignore the problem of clustering, by assuming that not only are the first probes random,

random probes but after a collision, the next probe will be random over all remaining positions of the table. In fact, let us assume that the table is so large that all the probes can be regarded as independent events.

Let us first study an unsuccessful search. The probability that the first probe hits an occupied cell is λ, the load factor. The probability that a probe hits an empty cell is $1 - \lambda$. The probability that the unsuccessful search terminates in exactly two probes is therefore $\lambda(1 - \lambda)$, and, similarly, the probability that exactly k probes are made in an unsuccessful search is $\lambda^{k-1}(1 - \lambda)$. The expected number $U(\lambda)$ of probes in an unsuccessful search is therefore

$$U(\lambda) = \sum_{k=1}^{\infty} k\lambda^{k-1}(1-\lambda).$$

unsuccessful retrieval This sum is evaluated in Appendix A.1; we obtain thereby

$$U(\lambda) = \frac{1}{(1-\lambda)^2}(1-\lambda) = \frac{1}{1-\lambda}.$$

To count the probes needed for a successful search, we note that the number needed will be exactly one more than the number of probes in the unsuccessful search made before inserting the item. Now let us consider the table as beginning empty, with each item inserted one at a time. As these items are inserted, the load factor grows slowly from 0 to its final value, λ. It is reasonable for us to approximate this step-by-step growth by continuous growth and replace a sum with an integral. We conclude that the average number of probes in a successful search is approximately

successful retrieval

$$S(\lambda) = \frac{1}{\lambda}\int_0^\lambda U(\mu)\,d\mu = \frac{1}{\lambda}\ln\frac{1}{1-\lambda}.$$

Similar calculations may be done for open addressing with linear probing, where it is no longer reasonable to assume that successive probes are independent. The details, however, are rather more complicated, so we present only the results. For the complete derivation, consult the references at the end of the chapter. For linear probing the average number of probes for an unsuccessful search increases to

linear probing

$$\tfrac{1}{2}\left[1 + \frac{1}{(1-\lambda)^2}\right]$$

and for a successful search the number becomes

$$\tfrac{1}{2}\left[1 + \frac{1}{1-\lambda}\right].$$

5. Theoretical Comparisons

Figure 6.13 gives the values of the foregoing expressions for different values of the load factor.

Load factor	0.10	0.50	0.80	0.90	0.99	2.00
Successful search						
Chaining	1.05	1.25	1.40	1.45	1.50	2.00
Open, Random probes	1.05	1.4	2.0	2.6	4.6	—
Linear probes	1.06	1.5	3.0	5.5	50.5	—
Unsuccessful search						
Chaining	0.10	0.50	0.80	0.90	0.99	2.00
Open, Random probes	1.1	2.0	5.0	10.0	100.	—
Linear probes	1.12	2.5	13.	50.	5000.	—

Figure 6.13. Theoretical comparison of hashing methods

conclusions

We can draw several conclusions from this table. First, it is clear that chaining consistently requires fewer probes than does open addressing. On the other hand, traversal of the linked lists is usually slower than array access, which can reduce the advantage, especially if key comparisons can be done quickly. Chaining comes into its own when the records are large, and comparison of keys takes significant time. Chaining is also especially advantageous when unsuccessful searches are common, since with chaining, an empty list or very short list may be found, so that often no key comparisons at all need be done to show that a search is unsuccessful.

With open addressing and successful searches, the simpler method of linear probing is not significantly slower than more sophisticated methods, at least until the table is almost completely full. For unsuccessful searches, however, clustering quickly causes linear probing to degenerate into a long sequential search. We might conclude, therefore, that if searches are quite likely to be successful, and the load factor is moderate, then linear probing is quite satisfactory, but in other circumstances another method should be used.

6. Empirical Comparisons

It is important to remember that the computations giving Figure 6.13 are only approximate, and also that in practice nothing is completely random, so that we can always expect some differences between the theoretical results and actual computations. For sake of comparison, therefore, Figure 6.14 gives the results of one empirical study, using 900 keys that are pseudorandom numbers between 0 and 1.

Load factor	0.1	0.5	0.8	0.9	0.99	2.0
Successful search						
Chaining	1.04	1.2	1.4	1.4	1.5	2.0
Open, Quadratic probes	1.04	1.5	2.1	2.7	5.2	—
Linear probes	1.05	1.6	3.4	6.2	21.3	—
Unsuccessful search						
Chaining	0.11	0.53	0.78	0.90	0.99	2.04
Open, Quadratic probes	1.13	2.2	5.2	11.9	126.	—
Linear probes	1.13	2.7	15.4	59.8	430.	

Figure 6.14. Empirical comparison of hashing methods

conclusions

In comparison with other methods of information retrieval, the important thing to note about all these numbers is that they depend only on the load factor, not on the absolute number of items in the table. Retrieval from a hash table with 20,000 items in 40,000 possible positions is no slower, on average, than is retrieval from a table with 20 items in 40 possible positions. With sequential search, a list 1000 times the size will take 1000 times as long to search. With binary search, this ratio is reduced to 10 (more precisely, to lg 1000), but still the time needed increases with the size, which it does not with hashing.

Finally, we should emphasize the importance of devising a good hash function, one that executes quickly and maximizes the spread of keys. If the hash function is poor, the performance of hashing can degenerate to that of sequential search.

Exercises 6.6

E1. Suppose that each item (record) in a hash table occupies s words of storage (exclusive of the pointer field needed if chaining is used), and suppose that there are n items in the hash table.

(a) If the load factor is λ and open addressing is used, determine how many words of storage will be required for the hash table.

(b) If chaining is used, then each node will require $s + 1$ words, including the pointer field. How many words will be used altogether for the n nodes?

(c) If the load factor is λ and chaining is used, how many words will be used for the hash table itself? (Recall that with chaining, the hash table itself contains only pointers requiring one word each.)

(d) Add your answers to the two previous parts to find the total storage requirement for load factor λ and chaining.

(e) If s is small, then open addressing requires less total memory for a given λ, but for large s, chaining requires less space altogether. Find the break-even value for s, at which both methods use the same total storage. Your answer will depend on the load factor λ.

E2. Figures 6.13 and 6.14 are somewhat distorted in favor of chaining, because no account is taken of the space needed for links (see part 2 of Section 6.5.4). Produce tables like Figure 6.13, where the load factors are calculated for the case of chaining, and for open addressing the space required by links is added to the hash table, thereby reducing the load factor.

(a) Given n nodes in linked storage connected to a chained hash table, with s words per item (plus 1 more for the link), and with load factor λ, find the total amount of storage that will be used, including links.

(b) If this same amount of storage is used in a hash table with open addressing and n items of s words each, find the resulting load factor. This is the load factor to use for open addressing in computing the revised tables.

(c) Produce a table for the case $s = 1$.

(d) Produce another table for the case $s = 5$.

(e) What will the table look like when each item takes 100 words?

E3. One reason why the answer to the birthday problem is surprising is that it differs from the answers to apparently related questions. For the following, suppose that there are n people in the room, and disregard leap years.

(a) What is the probability that someone in the room will have a birthday on a random date drawn from a hat?

(b) What is the probability that at least two people in the room will have that same random birthday?

(c) If we choose one person and find his birthday, what is the probability that someone else in the room will share the birthday?

ordered hash table

E4. In a chained hash table, suppose that it makes sense to speak of an order for the keys, and suppose that the nodes in each chain are kept in order by key. Then a search can be terminated as soon as it passes the place where the key should be, if present. How many fewer probes will be done, on average, in an unsuccessful search? In a successful search? How many probes are needed, on average, to insert a new node in the right place? Compare your answers with the corresponding numbers derived in the text for the case of unordered chains.

E5. In our discussion of chaining, the hash table itself contained only pointers, list headers for each of the chains. One variant method is to place the first actual item of each chain in the hash table itself. (An empty position is indicated by an impossible key, as with open addressing.) With a given load factor, calculate the effect on space of this method, as a function of the number of words (except links) in each item. (A link takes one word.)

Programming Project 6.6

P1. Produce a table like Figure 6.14 for your computer, by writing and running test programs to implement the various kinds of hash tables and load factors.

6.7 CONCLUSIONS: COMPARISON OF METHODS

choice of data structures

This chapter and the previous one have together explored four quite different methods of information retrieval: sequential search, binary search, table lookup, and hashing. If we are to ask which of these is best, we must first select the criteria by which to answer, and these criteria will include both the requirements imposed by the application and other considerations that affect our choice of data structures, since the first two methods are applicable only to lists and the second two to tables. In many applications, however, we are free to choose either lists or tables for our data structures.

table lookup

In regard both to speed and convenience, ordinary lookup in contiguous tables is certainly superior, but there are many applications to which it is inapplicable, such as when a list is preferred or the set of keys is sparse. It is also inappropriate whenever insertions or deletions are frequent, since such actions in contiguous storage may require moving large amounts of information.

Which of the other three methods is best depends on other criteria, such as the form of the data.

other methods

Sequential search is certainly the most flexible of our methods. The data may be stored in any order, with either contiguous or linked representation. Binary search is much more demanding. The keys must be in order, and the data must be in random-access representation (contiguous storage). Hashing requires even more, a peculiar ordering of the keys well suited to retrieval from the hash table, but generally useless for any other purpose. If the data are to be available immediately for human inspection, then some kind of order is essential, and a hash table is inappropriate.

near miss

Finally, there is the question of the unsuccessful search. Sequential search and hashing, by themselves, say nothing except that the search was unsuccessful. Binary search can determine which data have keys closest to the target, and perhaps thereby can provide useful information.

Brooks/Cole Publishing Company
A Division of Wadsworth, Inc

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

Library of Congress Cataloging in Publication Data

Stubbs, D. F. (Dan F.), [date]
Data structures with abstract data types and Pascal.

Includes index.
1. Data structures (Computer science) 2. Abstract data types (Computer science) I. Webre, N. W. (Neil W.), [date]. II. Title.
QA76.9.D35S78 1984 001.64' 84-14925
ISBN 0-534-03819-0

Sponsoring Editors: *Michael Needham, Neil Oatley*
Editorial Assistants: *Lorraine McCloud, Gaby Bert*
Marketing Representative: *Cynthia Berg*
Production Editor: *Candyce Cameron*
Manuscript Editor: *Harriet Serenkin*
Permissions Editor: *Carline Haga*
Cover and Interior Design: *Jamie Sue Brooks*
Art Coordinators: *Rebecca Tait, Michele Judge*
Interior Illustration: *Tim Keenan, Reese Thornton, Carl Brown, David Aguero*
Typesetting: *Graphic Typesetting Service, Los Angeles, California*
Printing and Binding: *R. R. Donnelley & Sons Co., Crawfordsville, Indiana*

4. We have not included the set operations *union, intersection,* and *difference* in set Specification 7.2. Could they be included? If so, how would the specifications have to be modified to do so?

7.4 Hashed Implementations

We have studied several methods for the storage and later retrieval of keyed records. Arrays, linked lists, and several kinds of trees provide structures that allow these operations. In each of these structures, the find operation is necessarily implemented by some form of search. The key values of records in the structure are compared with the desired, or target, key until either a matching value is found or the data structure is exhausted. The pattern of probes is dependent upon the methods of organizing and relating the records of the structure. A sorted linear list implemented as an array can be probed by a binary search. The same list in linked form can only be searched sequentially.

We might ask if it is possible to create a data structure that does not require a search to implement the find operation. Is it possible, for example, to compute the location of the record that has a given key value,

memory address of record = f(key)

where f is a function that maps each distinct key value into the memory address of the record identified by that key? We shall see that the answer is a qualified yes. Such functions can be found, but they are difficult to determine and can only be constructed if all of the keys in the data set are known in advance. They are called ***perfect hashing functions*** and are further examined in Section 7.4.3.

Normally, there has to be a compromise from a strictly calculated access scheme to a hybrid scheme that involves a calculation followed by some limited searching. The function does not necessarily give the exact memory address of the target record but only gives a ***home address*** that may contain the desired record:

home address = H(key$_i$)

Functions such as H are known as ***hashing functions.*** In contrast to perfect hashing functions, these are usually easy to determine and can give excellent performance. The home address may not contain the record being sought. In that case, a search of other addresses is required, and this is known as ***rehashing.*** In Section 7.4.1, we introduce a number of hashing functions, and in Section 7.4.2 we examine several rehashing strategies. In Section 7.5 we summarize the performance of hashed implementations, and in Section 7.6 we compare its operation and performance with that of lists and trees for the frequency analysis of digraphs.

The fundamental idea behind ***hashing*** is the antithesis of sorting. A sort arranges the records in a regular pattern that makes the relatively efficient binary search possible. Hashing takes the diametrically opposite approach. The basic idea is to scatter the records completely randomly throughout so memory or storage space—the so-called ***hash table.*** The hash function can be thought of as a pseudo-random-number generator that uses the value of the key as a seed and that outputs the home address of the element containing that key.

One of the drawbacks of hashing is the random locations of stored elements. There is no notion of first, next, root, parent, or child or anything analogous. Thus, hashing is appropriate for implementing a set relationship among elements but not for implementing structures that involve relationships among constituent elements. It is for that reason that hashing is discussed in this chapter on sets. There are, however, other appropriate contexts for a discussion of hashing.

One of the virtues of hashing is that it allows us to find records with $O(1)$ probes. The *findkey* operation has required a number of probes that depend on n in every implementation of every data structure discussed so far: $O(n)$ for a linked implementation of a list, $O(\log_2 n)$ for an array implementation of a sorted list, and $O(\log_2 n)$ for a binary search tree. Since hashing requires the fewest probes to find something, it is frequently considered to be a particularly effective search technique. Also, since hashing stores elements in a table, the hash table, it is sometimes considered to be a technique for operating on tables. All of these views of hashing are correct. We choose to view hashing as a technique for implementing sets. Its other advantages and disadvantages are not changed by this point of view.

It is convenient to consider the hash table to be an array of records and to let the hash function calculate the index value of the home address rather than to calculate its memory address directly. Once the appropriate index value is computed, the array mapping function can complete the transformation into an actual memory address. The hash table is then represented as shown in Figure 7.12.

```
const tablesize = {User supplied.}
type  position  = 0..(tablesize - 1);                {Not Standard Pascal.}

var table: array[position] of stdelement;                 {The hash table.}
```

Figure 7.12 Array representation of a hash table.

Suppose that we have a hash table defined by

```
var table: array[0..6] of record
                        key:  integer;
                        data: array[1..10] of char
                      end;
```

and that the hash function H is

$H(\text{key}) = \text{key mod } 7$

Notice that the value produced by this function is always an integer between 0 and 6, which is within the range of indexes of the table.

Table address	Table contents
[0]	empty
[1]	empty
[2]	empty
[3]	empty
[4]	empty
[5]	empty
[6]	empty

Figure 7:13
Empty table.

Table address	Table contents
[0]	empty
[1]	empty
[2]	empty
[3]	374, . . . data
[4]	empty
[5]	empty
[6]	empty

Figure 7.14
First record stored at table[3].

Table address	Table contents
[0]	empty
[1]	empty
[2]	empty
[3]	374, . . . data
[4]	empty
[5]	empty
[6]	1091, . . . data

Figure 7.15
Second record stored at table[6].

Table address	Table contents
[0]	empty
[1]	911, . . . data
[2]	empty
[3]	374, . . . data
[4]	empty
[5]	empty
[6]	1091, . . . data

Figure 7.16
Third record stored at table[1].

Operation *create* will produce the empty table shown in Figure 7.13. If the first record we store has a key value of 374, then the hash function

$H(374) = 374 \bmod 7 = 3$

places the record at table[3]. This is shown in Figure 7.14. If the next record has a key value of 1091, we get

$H(1091) = 1091 \bmod 7 = 6$

and the table becomes that shown in Figure 7.15. A third record with key = 911 gives

$H(911) = 911 \bmod 7 = 1$

and the resulting table is shown in Figure 7.16.

Retrieval of any of the records already in the table is a simple matter. The target key is presented to the hash function that reproduces the same table position as it did when the record was stored. If the target key were 740, a value not in the table, the hashing function would produce

$H(740) = 740 \bmod 7 = 5$

Interrogating table[5], we find that it is empty, and we conclude that a record with key = 740 is not in the table.

The example that we have just seen was constructed to conceal a serious problem. So far, keys with different values have hashed to different locations in the table. That is not generally so and is only the case in our current example because the key values were carefully chosen. Suppose that insertion of a record with a key value of 227 is attempted. Then,

$H(227) = 227 \bmod 7 = 3$

but table[3] is already filled with another record. This is called a ***collision***—two different key values hashing to the same location. Why this happens and what to do about it are important because collisions are a fact of life when hashing.

Suppose that employee records are hashed based on Social Security number. If a firm has 500 employees it will not want to reserve a hash table with 1 billion entries (the number of possible Social Security numbers) to guarantee that each of its employee records hashes to a unique location. Even if the firm allocates 1000 slots in its hash table and uses a hash function that is a "perfect" randomizer, the probability that there will be no collisions is essentially zero. This is the ***birthday paradox*** (Feller 1950), which says that hash functions with no collisions are so rare that it is worth looking for them only in very special circumstances. These special circumstances are discussed in Section 7.4.3. In the meantime, we need to consider what to do when a collision does occur.

With careful design, strategies for handling collisions are simple. They are commonly called ***rehashing*** or ***collision-resolution strategies,*** and we will discuss them in Section 7.4.2.

We s
H(key
in the exa
thing to d

7.4.1 H

There is a
proposed
straightfor
since the s
their use. I
exotic one
Good

1. Th
2. Th

We will n

• **Digit s**

The first h
keys of th
Social Sec

key =

If the pop
the last thr
possible in

var tabl

where *pers*
keep. Notic

H(key

which sim
Care
with which
digits, *d-d*
are probab
single stat
number ar
inally issue
and cluster
state; 567,

We selected the hashing function

$H(\text{key}) = \text{key} \bmod n$

in the example we just completed. We will now see why that was a reasonable thing to do and will also look at a number of other hashing functions.

7.4.1 Hashing Functions

There is a large and diverse group of ***hashing functions*** that have been proposed since the advent of the hashing technique. Some are simple and straightforward; others are complex. Almost all are computationally simple since the speed of the computation of such functions is an important factor in their use. Lum (1971) has a good review of many, including some of the more exotic ones. We will confine our attention to simple but effective methods.

Good hashing functions have two desirable properties:

1. They compute rapidly.
2. They produce a nearly random distribution of index values.

We will now consider several hashing functions.

- **Digit selection**

The first hashing function we will discuss is ***digit selection.*** Suppose that the keys of the set of data that we are dealing with are strings of digits such as Social Security numbers (nine-digit numbers):

$\text{key} = d_1d_2d_3d_4d_5d_6d_7d_8d_9$

If the population comprising the data is randomly chosen, then the choice of the last three digits, $d_7d_8d_9$, will give a good random distribution of values. A possible implementation is the following:

```
var table: array[0..999] of person;
```

where *person* is a record type for the key and information that we wish to keep. Notice that the hashing function in this case is

$H(\text{key}) = \text{key} \bmod 1000$

which simply strips off the last three digits of the key.

Care must be taken in deciding which digits to select. If the population with which we are dealing is students at a university, for example, the last three digits, $d_7d_8d_9$, are probably a good choice, whereas the first three digits, $d_1d_2d_3$, are probably not. State universities tend to draw their student bodies from a single state or geographical region. The first three digits of the Social Security number are based on the geographical region in which the number was originally issued. Most students from California, for example, have a first digit of 5 and clustered second and third digits, indicating various subregions of the state; 567, for example, is very common. If the data were for a California university, almost all of the students' records would map into the 500–599 range of the hash table, and a large subgroup would map into position 567. The output of the function would not be uniform and random but would be loaded on certain positions of the table causing an inordinately high number of collisions. It would not be a good hashing function for that reason.

If the key population is known in advance, it is possible to analyze the distribution of values taken by each digit of the key. The digits participating in the hash address are then easy to select. Such an analysis is called ***digit analysis.*** Instead of choosing the last three digits, we would choose the three digits of the key whose digit analyses showed the most uniform distribution. If d_4, d_7, and d_9 gave the flattest distributions, the hashing function might strip out those digits from a key and put them together to form a number in the range 0–999:

$$H(d_1d_2d_3d_4d_5d_6d_7d_8d_9) = d_4d_7d_9$$

Caution is advised, since although the digits are apparently random and uniform in value, they might have dependencies among themselves. For example, certain combinations of d_7 and d_9 might tend to occur together. Then if d_9 were always 8 when d_7 is 3, $d_4$38 would be the only table position mapped to in the range $d_4$30–$d_4$39, effectively lowering the table size and increasing the chances of collision. Analysis for interdigit correlations might be necessary to bring such a situation to light.

• Division

One of the most effective hashing methods is ***division,*** which works as follows:

$$H(\text{key}) = \text{key mod } m = h \qquad 0 <= h <= m - 1$$

The bit pattern of the key, regardless of its data type, is treated as an integer, divided in the integer sense by m, and the remainder of the division is used as the table address. h is in the range from 0 to $m - 1$. Such a function is fast on computer systems that have an integer divide, since most generate the quotient in one hardware register and the remainder in another. The content of the remainder register need only be copied into the variable h, and the hash is completed.

In practice, functions of this type give very good results. Lum (1971) has an empirical study showing this to be the case. Division can, however, perform poorly in a number of cases. For example, if m were 25, then all keys that were divisible by 5 would map into positions 0, 5, 10, 15, and 20 of the table. A subset of the keys maps into a subset of the table, something that we in general wish to avoid. Of course, using the function H = key mod m maps all keys for which key mod m = 0 into table[0], all keys for which key mod m = 1 into table[1], etc., but that bias is unavoidable. What we do not want to do is to introduce any further ones.

The problem underlying the choice of 25 as the table size is that it has a factor of 5. All keys with 5 as a factor will map into a table position that also has that factor. The cure is to make sure that the key and m have no common

factors, and
factors other
time that the
However, Lu
than 20, is su

• Multiplic

A simple met
that the keys

key = (

The key is so

×

$r_1r_2r_3r$

The result is
selection on
example, r_4r

It is imp
ing the right
comes only
right most tw
the same tal
introducing
involving the
in the key is
the key is an

• Folding

The next hasl
digit key as w

key = (

and the prog
hardware div
form a hash f

H(key)

The result wo

0 <= h

and could be
(there were
the numbers

factors, and the easiest way to ensure that is to choose m so that it has no factors other than 1 and itself—a prime number. For this reason, most of the time that the division function is used the table size will be a prime number. However, Lum (1971) shows that any divisor with no small factors, say less than 20, is suitable.

• Multiplication

A simple method that is based on ***multiplication*** is sometimes used. Suppose that the keys in question are five digits in length:

$$\text{key} = d_1d_2d_3d_4d_5$$

The key is squared by

$$\begin{array}{r} d_1d_2d_3d_4d_5 \\ \times\ \underline{d_1d_2d_3d_4d_5} \\ r_1r_2r_3r_4r_5r_6r_7r_8r_9r_{10} \end{array}$$

The result is a 10-digit product. The function is completed by doing digit selection on the product. In most cases, the middle digits are chosen, for example, $r_4r_5r_6$. An example is shown in Figure 7.17.

It is important to choose the middle digits. Consider, for example, choosing the right most two digits of the product in the example—41. That value comes only from the product of 1 × 21 and 2 × 21; that is, only from the right most two digits of the original key value. All keys ending in 21 will produce the same table location—41. This is the kind of bias that we try to avoid introducing. The middle digits, on the other hand, are formed from products involving the left, middle, and right portions of the key. Changing any one digit in the key is likely to change the hash result. Information from all portions of the key is amalgamated in the calculation of the hash table subscript.

```
key = 54321

    (a)

     54321
   × 54321
   -------
     54321
    108642
   162963
  217284
 271605
 ----------
 2950771041

    (b)

  h = 077

    (c)
```

Figure 7.17
(a) Key. (b) Results of squaring the key value. (c) Digit selection of the middle digits gives the table position of the record.

• Folding

The next hash function we will discuss is **folding.** Suppose that we have a five-digit key as we had in the multiplication method:

$$\text{key} = d_1d_2d_3d_4d_5$$

and the programs are running on a simple microcomputer system that has no hardware divide or multiply but that does have an arithmetic add. One way to form a hash function is simply to add the individual digits of the key:

$$H(\text{key}) = d_1 + d_2 + d_3 + d_4 + d_5$$

The result would be in the range

$$0 <= h <= 45$$

and could be used as the index in the hash table. If a larger table were needed (there were more than 46 records), the result could be enlarged by adding the numbers as pairs of digits:

$H(\text{key}) = 0d_1 + d_2d_3 + d_4d_5$

The result would then be between 000 and 207 (09 + 99 + 99). Folding is the name given to a class of methods that involves combining portions of the key to form a smaller result. The methods for combining are usually either arithmetic addition or exclusive or's.

Folding is often used in conjunction with other methods. If the key were a Social Security number of nine digits and the program were implemented on a minicomputer that has 16 bit registers and consequently has a maximum positive integer size of 65535, then the key is intractable as it stands. It must somehow be reduced to an integer less than 65535 before it can be used. Folding can be used to do this. Suppose the key in question has a value

key = 987654321

We can break the key into four-digit groups and then add them:

$$\begin{array}{rr} & 0309 \\ & 8765 \\ & \underline{4321} \\ \text{fold(key)} = & 13095 \end{array}$$

This result would be between 0 and 20007. Now apply a second hashing function, say division, to produce a table position within the range $0 \ldots (m - 1)$ If the hash table has m positions, the composite function is

$H(\text{key}) = \text{fold(key)} \bmod m$

• Character-valued keys

All of the examples in our discussion of hashing functions assumed that the keys were some form of integer. Quite often, however, the keys are character strings, or ***character-valued keys.*** How are these handled?

Remember that all data stored in a computer memory are simply strings of bits. The ASCII code for the character 'y', for example, is

1111001_2

which can also be interpreted as the integer value 121. The *ord* function of Pascal reinterprets characters as integers in this fashion:

$\text{ord('y')} = 121_{10}$

This provides one basis for using characters in hashing functions. If the key values are single characters, division can be applied as follows:

$H(\text{key}) = \text{ord(key)} \bmod m$

In the case key = 'y' and $m = 7$,

$H(\text{'y'}) = \text{ord('y')} \bmod 7 = 2$

If the key is a character string of length 2, such as,

key = 'jy'

the bit p

110

The cor

ord(

Since 12
'j' 7 bits
the three

ord(

16384 is
beyond t
and micr
of 3.

type

funct

var i
begin
i
fold
rep
fo

i
unti
end;

Algo

Algo
the simpl
fold.

7.4.2 C

A collisio
when two
will begin
strategies.
We w
nine-digit
R =

the bit pattern for the string would be

11010101111001_2

The corresponding integer is

ord('j') * 128 + ord('y') = 13689

Since 128 = 2^7, the multiplication by 128 effectively shifts the bit pattern for 'j' 7 bits to the left. The addition effectively concatenates the 2-bit strings. For the three-character string 'djy', we get

ord('d') * 16384 + ord('j') * 128 + ord('y') = 1,652,089

16384 is 2^{14}, providing a left shift of 14 bits for 'd'. Notice that the result is beyond the capacity of a 16-bit register, the size register available on most mini- and microcomputer systems. Algorithm 7.1 folds a 21-character string in groups of 3.

```
type string21 = array[1..21] of char;

function fold := (s: string21): integer;            {Folds a character string}
                                           {of 21 characters in groups of 3.}
var i: 1..22;                                   {At least 24 bit integers are}
begin                                                {required for the result.}
  i    := 1;
  fold := 0;
  repeat
    fold := fold + ord(s[i]) * 16384
                 + ord(s[i + 1]) * 128
                 + ord(s[1 + 2]);
    i := i + 3
  until i > 21
end;
```

Algorithm 7.1 Folding a character string.

Algorithm 7.1 could be written more generally, but doing so would obscure the simple process. Division hashing can be applied to the result of function *fold*.

7.4.2 Collision-Resolution Strategies

A **collision-resolution strategy**, or **rehashing**, determines what happens when two or more elements have a collision, or hash to the same address. We will begin by defining some parameters that will be used to help describe these strategies.

We will call the number of different values that a key can assume *R*. A nine-digit integer (for example, a Social Security number) has

R = 1,000,000,000

the bit pattern for the string would be

11010101111001_2

The corresponding integer is

ord('j') * 128 + ord('y') = 13689

Since 128 = 2^7, the multiplication by 128 effectively shifts the bit pattern for 'j' 7 bits to the left. The addition effectively concatenates the 2-bit strings. For the three-character string 'djy', we get

ord('d') * 16384 + ord('j') * 128 + ord('y') = 1,652,089

16384 is 2^{14}, providing a left shift of 14 bits for 'd'. Notice that the result is beyond the capacity of a 16-bit register, the size register available on most mini- and microcomputer systems. Algorithm 7.1 folds a 21-character string in groups of 3.

```
type string21 = array[1..21] of char;

function fold :=(s: string21): integer;              {Folds a character string}
                                               {of 21 characters in groups of 3.}
var i: 1..22;                                     {At least 24 bit integers are}
begin                                               {required for the result.}
  i    := 1;
  fold := 0;
  repeat
    fold := fold + ord(s[i]) * 16384
                 + ord(s[i + 1]) * 128
                 + ord(s[1 + 2]);
    i := i + 3
  until i > 21
end;
```

Algorithm 7.1 Folding a character string.

Algorithm 7.1 could be written more generally, but doing so would obscure the simple process. Division hashing can be applied to the result of function *fold*.

7.4.2 Collision-Resolution Strategies

A **collision-resolution strategy**, or **rehashing**, determines what happens when two or more elements have a collision, or hash to the same address. We will begin by defining some parameters that will be used to help describe these strategies.

We will call the number of different values that a key can assume R. A nine-digit integer (for example, a Social Security number) has

R = 1,000,000,000

```
const bucketsize  = {User supplied.}
      tablesize   = {User supplied.}

type  bucket      = array
                    [1. .bucketsize] of
                         stdelement;

var   table       = array
                    [0..(tablesize - 1)]
                         of bucket;
```

The size of the hash table, ***tablesize***, is a second important parameter. It must be large enough to hold the number of elements we wish to store.

The number of records that is actually stored in the table varies with time and is denoted $n = n(t)$. One of the most important parameters is the fraction of the table that contains records at any time. This is called the ***load factor*** and is written

$$\alpha = \alpha(t) = n / \text{tablesize}$$

In Figure 7.16, $\alpha = 3/7$.

In summary, the keys of our data elements are chosen from R different values, and n elements are stored in the hash table that is of size *tablesize* and is $\alpha \times 100\%$ full.

A more general form of hash table is obtained by allowing each hash table position to hold more than a single record. Each of these multirecord cells is called a ***bucket*** and can hold b records. An array representation of such a hash table is shown in Figure 7.18.

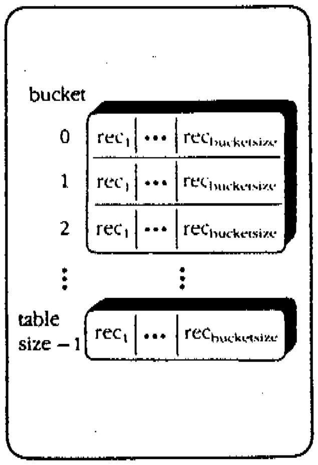

Figure 7.18
Hash table of buckets.

The concept of hash tables as collections of buckets is important for tables that are stored on direct access devices such as magnetic disks. For those devices, each bucket can be tied to a physical cell of the device, such as a track or sector. The hashing function produces a bucket number that results in the transfer of the physically related block into the random access memory (RAM). Once there, the bucket can be searched or modified at high speed.

Buckets of size greater than one are of limited use in hash tables stored in RAM. They tend to slow the average access time to records when searching. We will only discuss buckets of size one in this chapter. Bear in mind, however, that the hash table we discuss is a table of buckets of size one.

The strategies for resolving collisions will be grouped into three approaches. The first approach, ***open address methods,*** attempts to place a second and subsequent keys that hash to the one table location into some other position in the table that is unoccupied (open). The second approach, ***external chaining,*** has a linked list associated with each hash table address. Each element is added to the linked list at its home address. The third approach uses pointers to link together different buckets in the hash table. We will discuss ***coalesced chaining,*** since it is one of the better strategies that uses this technique.

• Open address methods

For all of the ***open address methods*** and their algorithms we will use the hash table represented in Figure 7.12. There are several open address methods using varying degrees of sophistication and a variety of techniques. All seek to find an open table position after a collision. Let us return to Figure 7.16, which is repeated for reference as Figure 7.19, and attempt to add the key whose value is 227. Recall that the example hashing function applied to 227 gives

Table address	Table contents
[0]	empty
[1]	911, . . . data . . .
[2]	empty
[3]	374, . . . data . . .
[4]	empty
[5]	empty
[6]	1091, . . . data . .

Figure 7.19
Three records stored at table[1], table[3], and table[6].

$$H(227) = 227 \bmod 7 = 3$$

so that 227 collides with 374.

Linear rehashing. A simple resolution to the collision called ***linear rehashing*** is to start a sequential search through the hash table at the position at which the collision occurred. The search continues until an open position is found or until the table is exhausted. A probe at position 4 reveals an open address, and the new record is stored there. The result is shown in Figure 7.20. A request to find the record with key = 227 generates the same search path used to store it.

Table address	Table contents
[0]	empty
[1]	911
[2]	empty
[3]	374
[4]	227
[5]	empty
[6]	1091

Figure 7.20
Linear rehashing.

We are now in a position to implement the operations specified in Section 7.3. The first operation is *findkey*, which is implemented by Algorithms 7.2 and 7.3.

```
procedure findkey(tkey: keytype): boolean;
var h: position;
begin
  h := H(tkey);                                    {Apply hash function.}

  if     (table[h].key <> tkey) and (table[h].key <> empty)
  then   linearrehash(tkey, h);

  if     tkey = table[h].key
  then   findkey := true
  else   findkey := false
end;
```

Algorithm 7.2 Implementation of operation *findkey* using the hash function.

```
procedure linearrehash(tkey: keytype; var h: position);
var start: position;
begin
  start := h;
  repeat
    h := (h + 1) mod tablesize
  until (table[h].key = tkey)                      {tkey found.}
    or (table[h].key = empty)                      {Open location.}
    or (k = start)                                 {Entire table searched.}
end;
```

Algorithm 7.3 Linear rehashing.

To insert an element we search, beginning at the home address, until an empty address is found or until the table is exhausted. For example, inserting an element whose key is 421 in Figure 7.20 leads to the Figure 7.21. We have added a column to our illustration of hash tables—the number of probes required to find each element stored therein. In the case of linear rehashing, it is easy to determine an element's home address from this added information. The *insert* operation can be implemented as shown in Algorithm 7.4.

Table address	Table contents	Probes
[0]	empty	
[1]	911	1
[2]	421	2
[3]	374	1
[4]	227	2
[5]	empty	
[6]	1091	1

Figure 7.21
Hash table and the number of probes required to find an element in the table.

We will assume two user-supplied values for the key of an element: *empty* and *deleted*. The use of *empty* is obvious. Let us see why we need the value *deleted*.

```
procedure insert(e: stdelement);                         {Insert an element using}
var h: position;                                                  {linear rehashing}
begin
  h := H(e.key);
  while (table[h].key <> empty) and (table[h].key <> deleted) do
    h := (h + 1) mod tablesize;
  table[h].elt := e
end;
```

Algorithm 7.4 Implementation of operation *insert* using linear rehashing.

Table address	Table contents	Probes
[0]	empty	
[1]	911	1
[2]	421	2
[3]	374	1
[4]	227	2
[5]	624	5
[6]	1091	1

Figure 7.22
The probe sequence when searching for 624 (or any other key value whose home address is 1).

Figure 7.22 shows the result of adding 624, whose home address is 1, to the hash table in Figure 7.21. The probes needed to find an empty space for 624 are also shown. A subsequent search using linear rehashing to find 624 will retrace that same path. If any of the three elements, 421, 374, or 227, were deleted and replaced by the value *empty*, subsequent searches for 624 would not work. Upon encountering a location marked *empty* the search would terminate unsuccessfully. A solution to this problem is to mark positions from which elements have been deleted with a special value. The deletion operation can be then implemented as shown in Algorithm 7.5.

```
procedure delete(tkey: keytype);          {Delete an element from the hash table.}
var h: position;
begin
  h := H(tkey);                                          {Apply hash function.}
  if    (table[h].key <> tkey) and (table[h].key <> empty)
  then linearrehash(tkey, h);
  table[h].key := deleted
end;
```

Algorithm 7.5 Implementation of operation *delete* using the hash function.

The drawback to the use of the value *deleted* is that it can clutter up the hash table thereby increasing the number of probes required to find an element. A partial solution is to reenter all legitimate elements periodically and to mark the remaining locations *empty*.

The performance of a combined hashing/rehashing strategy is measured by the number of probes it makes in searching for target key values. We will examine the performance of linear rehashing in more detail in Section 7.5, but we can get a feel for the fact that it may not perform very well by looking at the probe sequence that results when a search of Figure 7.22 is undertaken for a key value of 624. Since 624 mod 7 = 1, the search begins at position 1 in the table. The subsequent search is shown. Five probes are required to find 624. There are two problems underlying the linear probe method.

Problem 1. Any key that hashes to a position, say *h*, will follow the same rehashing pattern as all other keys that hash to *h*. Any key that hashes to position 1 in Figure 7.22 will follow the probe sequence shown. This guarantees that any key that hashes to 1 will have to collide with all of the keys that previously hashed to 1 before it is found or before an empty position is found. We will call this phenomenon ***primary clustering***.

Problem 2. Note in Figure 7.22 that the probe pattern for a rehash from position 1 merged with the probe pattern for a rehash from position 3. The two rehash patterns have merged together, a phenomenon called ***secondary clustering.***

Table address	Table contents	Probes
[0]	empty	
[1]	911	1
[2]	421	2
[3]	374	1
[4]	227	2
[5]	empty	
[6]	1091	1

Figure 7.23

Consider Figure 7.23 (which is a copy of Figure 7.21). There is a substantial difference in the probabilities of positions 0 and 5 receiving the next new key. Only new keys hashing into positions 6 and 0 will rehash (if necessary) to position 0. Keys hashing into any other position will eventually arrive at position 5.

The expected number of probes for any random key not yet in the table can be calculated as shown in Figure 7.24.

Original hash position		Number of probes	Empty position found at
0		1	0
1		5	5
2		4	5
3		3	5
4		2	5
5		1	5
6		2	0
	Total	18	

Figure 7.24 Expected number of probes for an unsuccessful search in the hash table shown in Figure 7.23. Expected number of probes = 18/7 = 2.57.

The expected number of probes for both ***successful*** (target key in table) and ***unsuccessful*** (target key not in table) ***searches*** will be our measures of performance of rehashing strategies, and we will examine them in a more general way in Section 7.5. We will confine our attention here simply to noting that the performance can be improved by eliminating the problems that we noted—primary and secondary clustering.

You may be tempted to resolve the difficulties by introducing a step size other than 1 for linear rehash. Stepping to a new table position in Algorithm 7.3 would become

```
k := (k + c) mod m
```

where 1 <= *c* <= (*tablesize* − 1). If *tablesize* is prime, or at least if *c* and *tablesize* are relatively prime (have no common factors), then the search pattern will cover the entire table probing at each position exactly once without repetition. This kind of coverage, ***nonrepetitious complete coverage,*** is highly desirable. Obviously, if a table position that was previously probed were again probed during the same rehashing sequence, the duplicate probe would be wasted and would affect performance. If the probe pattern did not cover the entire table, empty spaces that are not included in the pattern would not be discovered.

Although a value of *c* that is relatively prime to the table size does give a rehash technique that has these properties of nonrepetition and complete coverage, it does not solve or, in fact, even improve the problems of primary and secondary clustering. An approach that does solve one of these problems is described next.

Quadratic rehashing. One method of improving the performance of rehashing is to probe at

```
k := (home address ± j**2) mod tablesize
```

where *j* takes on the values 1, 2, 3,... until either the target key or an empty position is found or until the table is completely searched. This method, called ***quadratic rehashing,*** is better than linear rehashing because it solves the problem of secondary clustering (it does not solve the problem of primary clustering). Details of this method are given in Radke (1970), where it is shown that rehashing visits all table locations without repetition provided *tablesize* is a prime number of the form 4*k* + 3.

Random rehashing. Envision a rehashing strategy that, when a collision occurs, simply jumps randomly to a new table position. This method is called ***random rehashing,*** and the rehash can be considered to be a jump of a random distance from the original hash position or to be a second hash function applied to the same key. If second and subsequent collisions occur, the process is repeated until the target key or an empty position is found or until the table is determined to be full and not to contain the target key. Since each key would have its own random pattern, there would be no fixed rehashing patterns. (The random sequence would have to be determined by the key value since subsequent accesses with the same key value must follow the same pattern as the original.) Since there would be no common patterns, there would be no primary or secondary clustering. Although this approach is theoretically appealing, it appears difficult to implement. Thus we turn to schemes that are simpler and whose performances are almost as good.

Double hashing. Several methods exist that attempt to approximate the random rehashing strategy without the large overhead of calculation required by it. One of these, ***double hashing,*** is computationally efficient and simple to apply.

We ha

(i + c)
(i + 2 *
(i + 3 *

where *c* is a
The fact that
since it caus
be random l
such an app
One so
collided at p
key value so
values of *c*. l

H(key)

we define a

c(key)

Suppose that
position 1. W

c(421)

so the table

(1 + 2)
(1 + 2

If 624 had b
However, its

c(624)

and the prob

(1 + 5)
(1 + 2
(1 + 3

The reh
position orig
that hash to t
of such an e
izing step siz
of the expect
is quite clos

We have seen that the general pattern for linear probing is to probe at

(i + c)   **mod** tablesize
(i + 2 • c) **mod** tablesize
(i + 3 • c) **mod** tablesize
⋮

where *c* is a constant (*c* = 1 in our original discussion of linear rehashing). The fact that *c* is a constant is at the root of the inefficiency of linear rehashing, since it causes fixed probe patterns and clustering. Ideally we would like *c* to be random but subject to constraints on repetition. Although this is possible, such an approach leads to a computational overhead that is too high.

One solution is to compute a random jump size, *c*, for each key that has collided at position *b* and needs rehashing. Thus, *c* would be a function of the key value so that different keys hashing to the same location are given different values of *c*. For example, starting with the hashing function

H(key) = key mod *tablesize*

we define a related step size function

c(key) = [key mod (*tablesize* − 2)] + 1

Table address	Table contents
[0]	empty
[1]	911
[2]	empty
[3]	374
[4]	227
[5]	empty
[6]	1091

Figure 7.25

Suppose that 421 is to be stored in Figure 7.25. Then, 421 collides with 911 at position 1. When the collision occurs, *c* is computed as

c(421) = 421 mod 5 + 1 = 2

so the table is probed at

(1 + 2)   mod 7 = 3   *{Collision.}*
(1 + 2 • 2) mod 7 = 5   *{Empty.}*

If 624 had been the key, it would have also collided with 911 at position 1. However, its rehash pattern would have been different, that is,

c(624) = 624 mod 5 + 1 = 5

and the probes would have been at

(1 + 5)   mod 7 = 6   *{Collision.}*
(1 + 2 • 5) mod 7 = 4   *{Collision.}*
(1 + 3 • 5) mod 7 = 2   *{Empty.}*

The rehash pattern for the two keys, both of which hashed to the same position originally, is different. Although we can find pairs (or groups) of keys that hash to the same position and produce the same step size *c*, the probability of such an event is low for hash tables of reasonable size and a good randomizing step size generator. In fact, the performance of double hashing in terms of the expected number of probes for both successful and unsuccessful accesses is quite close to that of random rehashing. Since it has essentially the same performance in numbers of probes and a lower overhead in computation per probe, it has a greater overall efficiency. A rehashing algorithm for double hashing is given as Algorithm 7.6. It is comparable to Algorithm 7.3.

```
procedure doublerehash(tkey: keytype; var h: position);
var start: position;
    c:     integer;
begin
  start := h;
  c     := tkey mod (tablesize − 2) + 1;
  repeat
    h := (h + c) mod tablesize
  until (table[h].key = tkey)                    {tkey found.}
    or (table[h].key = empty)                    {Open location.}
    or (h = start)                               {Entire table searched.}
end;
```

Algorithm 7.6 Rehashing algorithm for double hashing.

Algorithm 7.6 shows only one method for computing a random step size. Any randomizing function that produces a step size that is less than n and is not based on the position of the original collision will do. However, the division algorithm that is shown is efficient and simple. In order to avoid introducing biases, *tablesize* should be a prime number. If we use this method of computing c in conjunction with the division method for the original hash, the choice of m and k as ***twin primes*** assures an exhaustive search of the table without repetition. If *tablesize* is prime, and k = *tablesize* − 2 is also prime, then m and k are twin primes.

```
const tablesize = {User supplied.}
type pointer  = ^node;
     node     = record
                  el: stdelement;
                  next: pointer
                end;
     position: 0..(tablesize − 1);

var table: array[position] of pointer;
```

Figure 7.26
Representation of a hash table for external chaining.

• External chaining

A second approach to the problem of collisions, called ***external chaining***, is to let the table position "absorb" all of the records that hash to it. Since we do not usually know how many keys will hash into any table position, a linked list is a good data structure to collect the records. A representation based on an array of pointers is shown in Figure 7.26.

As an example, let *tablesize* = 7 and suppose that operation *create* has initialized the hash table as shown in Figure 7.27.

Table address	Table contents
[0]	nil
[1]	nil
[2]	nil
[3]	nil
[4]	nil
[5]	nil
[6]	nil

Figure 7.27
Initialized hash table for external chaining.

If a division hash function is chosen, say,

$H(\text{key}) = \text{key mod } 7$

then insertion of the keys

key = 374	374 mod 7 = 3
key = 1091	1091 mod 7 = 6
key = 911	911 mod 7 = 1

produces the hash table shown in Figure 7.28. Insertion of 227 and 421 produces two collisions (the collisions are not shown in the text):

Table address	Table contents
[0]	nil
[1]	→ 911
[2]	nil
[3]	→ 374
[4]	nil
[5]	nil
[6]	→ 1091

Figure 7.28
Hash table after insertion of keys 374, 1091, 911.

key = 227 227 mod 7 = 3
key = 421 421 mod 7 = 1

and results in Figure 7.29. Subsequent insertion of 624

key = 624 624 mod 7 = 1

produces the result shown in Figure 7.30.

Each list is a linked list. The designer has all of the choices of list characteristics as he or she has for any linked list—method of termination, single or double linkage, other access pointers, and ordering of the list. If the frequencies with which the various records are accessed are quite different, it may be effective to make each list self-organizing.

Observe that the operations in this case are similar to those on lists discussed in Chapter 4. The only differences are that there are many lists instead of one and that the list in which we are interested is determined by the hash function.

External chaining has three advantages over open address methods:

1. Deletions are possible with no resulting problems.
2. The number of elements in the table can be greater than the table size; α can be greater than 1.0. Storage for the elements is dynamically allocated as the lists grow larger.
3. We shall see in Section 7.5 that the performance of external chaining in executing a *findkey* operation is better than that of open address methods and continues to be excellent as α grows beyond 1.0.

In the next technique collisions are resolved, as they are in external chaining, by adding the element to be inserted to the end of a list. The difference is in how the list is constructed.

Table address	Table contents
[0]	nil
[1]	→ 911 → 421
[2]	nil
[3]	→ 374 → 227
[4]	nil
[5]	nil
[6]	→ 1091

Figure 7.29
Hash table after insertion of keys 227 and 421.

Table address	Table contents
[0]	nil
[1]	→ 911 → 421 → 624
[2]	nil
[3]	→ 374 → 227
[4]	nil
[5]	nil
[6]	→ 1091

Figure 7.30
Hash table after insertion of key 624.

• Coalesced chaining

To illustrate ***coalesced chaining*** consider the hash table with seven buckets shown in Figure 7.31. The hash table is divided into two parts: the ***address region*** and the ***cellar.*** In our example, the first five addresses make up the address region, and the last two make up the cellar.

The hash function must map each record into the address region. The cellar is only used to store records that collided with another record at their home addresses. For our example, we will use the division hash function

$H(\text{key}) = \text{key mod } 5$

assuming that each key is an integer.

After inserting key values 27 and 29 we have Figure 7.32. If 32 is inserted next, it collides with 27 and is stored in the empty position with the largest address. In addition, it is added to a list that begins at its home address. The result is shown in Figure 7.33. To assist in visualizing the process, the empty position with the largest address, **epla**, is shown in the figures.

If key value 34 is added, it collides with 29 and is placed in address 5 (the

Table address	Table contents	
[0]	empty	↑
[1]	empty	*address*
[2]	empty	*region*
[3]	empty	
[4]	empty	↓
[5]	empty	↑
[6]	empty	*cellar*

Figure 7.31
Hash table with seven buckets initialized for coalesced chaining.

Table address	Table contents
[0]	empty
[1]	empty
[2]	27
[3]	empty
[4]	29
[5]	empty
[6]	**epla**

Figure 7.32
Hash table after inserting keys 27 and 29.

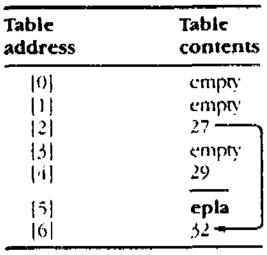

Table address	Table contents
[0]	empty
[1]	empty
[2]	27
[3]	empty
[4]	29
[5]	**epla**
[6]	32

Figure 7.33
Results after inserting key 32.

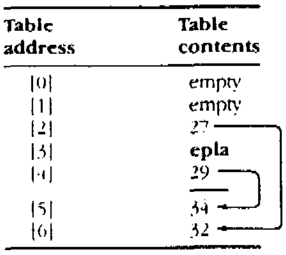

Table address	Table contents
[0]	empty
[1]	empty
[2]	27
[3]	**epla**
[4]	29
[5]	34
[6]	32

Figure 7.34
Results after inserting key 34.

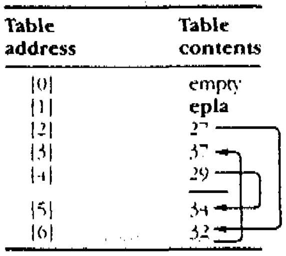

Table address	Table contents
[0]	empty
[1]	**epla**
[2]	27
[3]	37
[4]	29
[5]	34
[6]	32

Figure 7.35
Results after inserting key 37.

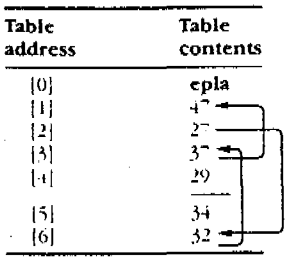

Table address	Table contents
[0]	**epla**
[1]	47
[2]	27
[3]	37
[4]	29
[5]	34
[6]	32

Figure 7.36
Results after inserting key 47.

empty position with the largest address) and is added to a list beginning at location 4. The result is shown in Figure 7.34.

Up to this point coalesced chaining has behaved exactly like external chaining—each new record is added to the end of a list that begins at its home address. The next insertion illustrates how a collision is resolved after the cellar is full.

If 37 is added it collides with 27, so it is placed in location [3] and added to the end of the list that begins at address [2]. The result is shown in Figure 7.35. The point to be made here is that once again the record being inserted was, since its home address was already occupied, placed in the empty position with the largest address. Adding 47 produces the result shown in Figure 7.36.

The term "coalesced" is used to describe this technique because, for example, if 53 were added to the hash table in Figure 7.36, it would cause the list that begins at [2] to coalesce with the list that begins at [3]. Note, however, that lists cannot coalesce until after the cellar is full.

The effectiveness of coalesced chaining depends on the choice of cellar size. Selection of cellar size is discussed in Vitter (1982, 1983) where it is shown that a cellar that contains 14% of the hash table works well under a variety of circumstances.

Because overflow records form lists, the deletion problems of open addressing schemes can be solved without resorting to marking records deleted. Any such approach is, however, more complicated than for the external chaining approach since the lists can coalesce. Details of such a deletion scheme, which essentially relinks elements in a list past the element to be deleted, are given in Vitter (1982).

This concludes our introduction to collision-resolution techniques. In Sections 7.5 and 7.6 we will compare these techniques from the point of view of performance. Before we do so, however, in Section 7.4.3 we will introduce hash functions that guarantee that collisions will not occur—perfect hashing functions.

7.4.3 Perf

A ***perfect h***
perfect hasl
hash table ha
collisions, we
that has a giv
that such fun

Perfect I
One such co
applications
programmin
procedure,
program's sta
word. Suppo
perfect hashi
reserved wo
of the specifi
same, a rese
not a reserv

Another
cerns the am
which can be
increases ex
possible fun
into a hash ta
functions th
1973b). Thus
the number
perfect hashi

There a
has propose
suggested so
the times to
fect functions

Let us l
are for keys t
of Pascal (see

H(key)

where

L = len

The function
is the intege
integer asso
ation betwee

7.4.3 Perfect Hashing Functions

A ***perfect hashing function*** is one that causes no collisions. A ***minimal perfect hashing function*** is a perfect hashing function that operates on a hash table having a load factor of 1.0. Since perfect hashing functions cause no collisions, we are assured that exactly one probe is needed to locate an element that has a given key value. This is, of course, very desirable. The problem is that such functions are not easy to construct.

Perfect hashing functions may only be found under certain conditions. One such condition is that all of the key values are known in advance. Certain applications have this quality; for example, the reserved, or key, words of a programming language. In Pascal there are 36 reserved words: **begin, end, procedure,**.... When a compiler is translating a program, as it scans the program's statements it must determine whether it has encountered a reserved word. Suppose the reserved words are stored in a hash table accessible by a perfect hashing function. Determining if a word encountered in the scan is a reserved word requires only one probe. The word is hashed, and the content of the specified table is compared with the word from the scan. If they are the same, a reserved word was found. If not, we can be certain that the word is not a reserved word.

Pascal Reserved Words

and
array
begin
case
const
div
do
downto
else
end
file
for
forward
function
goto
if
in
label
mod
nil
not
of
or
packed
procedure
program
record
repeat
set
then
to
type
until
var
while
with

Another condition for perfect hashing functions is a practical one. It concerns the amount of computation necessary to find a perfect hashing function, which can be enormous. The total amount of computation (and therefore time) increases exponentially with the number of keys in the data. The number of possible functions that map the 31 most frequently occurring English words into a hash table of size 41 is approximately 10^{50}, whereas the number of such functions that give unique (perfect) mappings is approximately 10^{43} (Knuth 1973b). Thus, only one of each 10 million functions is suitable. In practice, if the number of keys is greater than a few dozen, the amount of time to find a perfect hashing function is unacceptably long on most computers.

There are several proposals for perfect hashing functions. Sprugnoli (1977) has proposed functions that are perfect but not minimal. Cichelli (1980) has suggested some simple minimal perfect functions and has given examples and the times to compute them. Jaeschke (1981) has proposed other minimal perfect functions that avoid some problems that might arise with Cichelli's method.

Let us look briefly at Cichelli's method. The functions that he proposed are for keys that are character strings. Take, for example, the 36 reserved words of Pascal (see the list in the margin). The hashing function is

$$H(\text{key}) = L + g(\text{key}[1]) + g(\text{key}[L])$$

where

$$L = \text{length of the key}$$

The function $g(x)$ associates an integer with each character x; thus, $g(key[1])$ is the integer associated with the first letter of the key, and $g(key[L])$ is the integer associated with the last letter of the key. Figure 7.37 shows an association between letters and integers found by Cichelli.

a	=	11	q	=	0	n	=	13
f	=	15	v	=	10	s	=	6
k	=	0	c	=	1	x	=	0
p	=	15	h	=	15	e	=	0
u	=	14	m	=	15	j	=	0
z	=	0	r	=	14	o	=	0
b	=	15	w	=	6	t	=	6
g	=	3	d	=	0	y	=	13
l	=	15	i	=	13			

Figure 7.37
Cichelli's associated integer table for Pascal's reserved words.

[2]	do	[20]	record
[3]	end	[21]	packed
[4]	else	[22]	not
[5]	case	[23]	then
[6]	downto	[24]	procedure
[7]	goto	[25]	with
[8]	to	[26]	repeat
[9]	otherwise	[27]	var
[10]	type	[28]	in
[11]	while	[29]	array
[12]	const	[30]	if
[13]	div	[31]	nil
[14]	and	[32]	for
[15]	set	[33]	begin
[16]	or	[34]	until
[17]	of	[35]	label
[18]	mod	[36]	function
[19]	file	[37]	program

Figure 7.38
The hash table for Pascal reserved words.

As an example, suppose that the word "begin" were encountered by a compiler. The hashing function result would be

$$H(\text{'begin'}) = 5 + 15 + 13 = 33$$

The hashing function is simple, as it should be.

There are several problems, however. The first is that of looking up the integer associated with the two or more letters, but that can be done with reasonable efficiency. A second and more serious problem is that of determining which integer should be associated with each character. The integers are found by trial and error using a ***backtracking algorithm.*** (Of course, the associated integer table, see Figure 7.38, need be built only once.) Cichelli (1980) has a good discussion of the backtracking algorithm used for this problem.

In summary, perfect hashing functions are feasible when the keys are known in advance and the number of records is small. In that case, a perfect hashing function is determined in advance of the use of the hash table. Although its determination may be costly, it need only be done once. The resulting access to the records of the hash table requires only one probe.

Exercises 7.4

1. Explain the following terms in your own words:

    hash function
    collision
    load factor
    external chaining
    home address
    collision resolution
    linear rehash
    coalesced chaining
    perfect hashing function
    double hashing

2. The division hash function

    $$H(\text{key}) = \text{key mod } m$$

    is usually a good hash function if m has no small divisors. Explain why this restriction is placed on m.

3. Develop a hash function to convert nine-digit integers (Social Security numbers) into integers in the range 0 . . . 999. Test your hash function by applying it to 800 randomly generated keys. Determine how many of the addresses received 0, 1, 2, . . . of the hashed keys.

    Compare your experimental results with the results that would be obtained using a "perfect randomizer." The number of addresses receiving exactly k hashed values if the hash function is a perfect randomizer is approximated by

    $$e^{-\alpha}\frac{\alpha^k}{k!}$$

    where α is the load factor.

4. Develop a hash function to convert keys of the type

    keytype = **array**[1..15] **of** char;

    into integers in the range 0..999. Implement your hash function and determine

its exc
pare tl
5. Imple
its exe
6. Use th
8.
in the
var
a. Us
b. Us
c. Us
d. Us
I
F
values
e. Th
f. Th
g. Th
7. Imple
to Sp
a. Lin
b. De
c. Ext
d. Co
E
tal
and a l
chaini
produ
functi
intege

7.5 Ha

For this
groups.
hash tab
operatio
Operatio
O(*tables*

its execution time. Do the same for the hash function in Exercise 3 and compare their execution times.

5. Implement the perfect hashing function described in Section 7.4.3. Determine its execution time and compare it with the results obtained in Exercise 4.
6. Use the hash function $H(\text{key}) = \text{key mod } 11$ to store the sequence of integers

   82, 31, 28, 4, 45, 27, 59, 79, 35

   in the hash table

   ```
   var table: array[0..10] of integer;
   ```

   a. Use linear rehashing
   b. Use double hashing
   c. Use external chaining
   d. Use coalesced chaining with a cellar size of four and the hash function

   $H(\text{key}) = \text{key mod } 7$

   For each of the above collision-handling strategies determine (after all values have been placed in the table) the following:
   e. The load factor
   f. The average number of probes needed to find a value that is in the table
   g. The average number of probes needed to find a value that is not in the table
7. Implement a collection of procedures that forms a hashing package according to Specification 7.2. Use
   a. Linear rehashing
   b. Double hashing
   c. External chaining
   d. Coalesced chaining with a cellar size of 70.

   Let a hash table be given by

   ```
   table: array[0..500] of integer;
   ```

   and a hash function by $H(\text{key}) = \text{key mod } 501$. [The hash function for coalesced chaining will be $H(\text{key}) = \text{key mod } 431$.] Use a random number generator to produce a sequence of integers to store in the hash table. Determine, as a function of the load factor, the average number of probes needed to find an integer in the table.

7.5 Hashing Performance

For this discussion, the operations in Specification 7.2 are divided into two groups. The first group includes operations that do not involve searching the hash table: *full, size, create, clear,* and *traverse*. The effort to execute these operations does not depend on which collision-resolution strategy is used. Operations *full* and *size* require $O(1)$ effort. Operations *create* and *clear* require $O(\textit{tablesize})$ effort since each table position must be initialized to the value *empty*. Operation *traverse* requires probing $O(tablesize)$ table positions and processing $O(n)$ elements.

Each operation in the second group requires searching the hash table for the key value of an element. These associative searches are either successful (an element for which the target key value is found) or unsuccessful. The operations in this group are *findkey*, *insert*, *retrieve*, *update*, and *delete*. The performance of all of these operations is primarily determined by the associated search. We will therefore discuss the number of compares required for successful and unsuccessful searches. We will single out the *delete* operation for discussion later.

7.5.1 Performance

Explicit expressions that give the expected number of compares required for successful and unsuccessful searches can be developed. Results for three different collision-resolution policies are shown in Figures 7.39 and 7.40. Figure 7.39 shows the algebraic expressions [see Knuth (1973b) for their development], and Figure 7.40 shows the results of graphing the algebraic expressions. Observe that any random rehashing technique will give results very close to those for double hashing.

Expressions for coalesced chaining are given in Vitter (1982). Note that if the cellar is not full, the result for coalesced chaining is the same as for external chaining. In general, the search effort of coalesced chaining is approximately the same as that of external chaining. See Vitter (1982) in which the performance of coalesced chaining is compared with all the hashing techniques discussed in this chapter. Coalesced chaining is shown to give the best performance for the circumstances we considered.

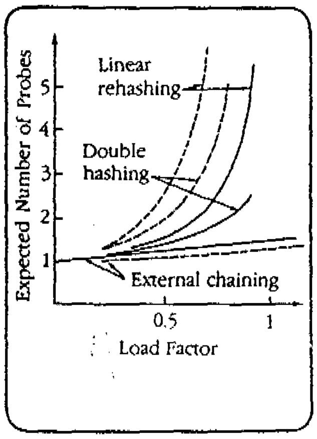

Figure 7.40
Number of probes required for successful and unsuccessful searches in a hash table. —, successful, - - - - - - -, unsuccessful.

Collision/ resolution strategy	Unsuccessful	Successful
Linear rehashing	$\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$	$\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)}\right)$
Double hashing	$\frac{1}{1-\alpha}$	$-\left(\frac{1}{\alpha}\right) \times \log(1-\alpha)$
External chaining	$\alpha + e^{-\alpha}$	$1 + \frac{1}{2}\alpha$

Figure 7.39 Algebraic expressions for the number of probes expected for successful and unsuccessful searches in a hash table.

Notice in Figures 7.39 and 7.40 that the performance curves for hashing methods are monotonically increasing functions of α, the load factor. The performance curves for lists and trees are monotonically increasing functions of n, the number of elements in the data structure. The number of elements, n, is not under the implementor's control. However, for hashing, the load

factor, α.
value of
hashing.

7.5.2 A

In additi
ments of
hash tabl
element
table cor

$T \times$

$T \times$

$T +$

The
in a hash
lesced cl
position.
position
will now

If u
itself), th
Figure 7.
table is n
as extern.
the perfo
provides

If u
External
less of a p
rules of
elements
and saves
ing provi
ments are
or nearly

Thes
elements
example,
user-defin
both large
It may be
than 1.0.

factor, α, may be made arbitrarily small by increasing the table size. For a given value of n, we can reduce the load factor and improve the performance of hashing. The price is more memory.

7.5.2 Memory Requirements

In addition to performance, it is important to compare the memory requirements of various hashing techniques. Let T be the number of buckets in the hash table; assume that a pointer occupies one word of memory and that an element occupies w words of memory. The memory requirements for a hash table containing n elements is then

$T \times w$ for any open addressing method

$T \times (w + 1)$ for coalesced chaining

$T + n(w + 1)$ for external chaining

These expressions are based on the following assumptions. Each position in a hash table for open addressing contains room for one element. For coalesced chaining the hash table contains one pointer and one element in each position. For external chaining the hash table contains one pointer in each position and one pointer and one element for each element in the table. We will now use the expressions to consider two cases.

If w is 1 (perhaps we store a pointer to an element rather than the element itself), then the memory required as a function of load factor is that shown in Figure 7.41. Open addressing always requires the least memory. When the table is nearly full, open addressing requires only one-third as much memory as external chaining. Of course, when the table is nearly full (see Figure 7.40), the performance of open addressing is poor. In this case, coalesced chaining provides good performance with a substantial saving in memory requirements.

If w is 10, then the memory requirements are as shown in Figure 7.42. External chaining is attractive over a wider range of load factors and extracts less of a penalty when the table is nearly full. This analysis leads to the following rules of thumb for constructing hash tables to be stored in RAM: For small elements and load factors, open addressing provides competitive performance and saves memory. For small elements and large load factors, coalesced chaining provides good performance with reasonable memory requirements. If elements are large, external chaining provides good performance with minimum, or nearly minimum, memory requirements.

These rules are based on the assumption that the maximum number of elements in the table can be estimated. Often that is not the case. Take, for example, the symbol table of a compiler that is used to store data about the user-defined identifiers in programs. The compiler must be able to process both large and small programs with a wide range in the numbers of identifiers. It may be possible for the table to overfill; that is, have a load factor greater than 1.0. The compiler should continue to operate smoothly. Such situations

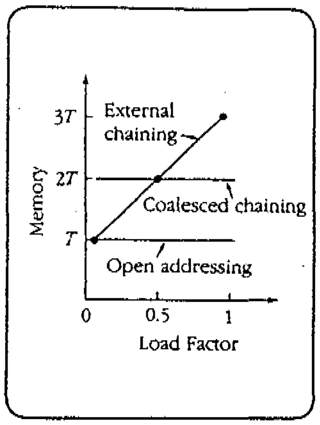

Figure 7.41
Memory requirements when an element occupies the same amount of memory as a pointer.

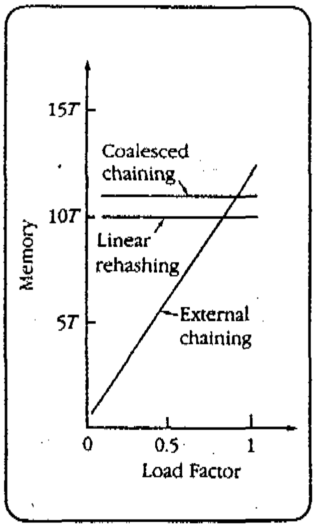

Figure 7.42
Memory requirements when an element occupies 10 times the amount of memory as a pointer.

are often handled by the use of external chaining, which continues to function for load factors greater than 1.0.

7.5.3 Deletion

We will conclude this section with a few comments about deletion. As discussed earlier, hash tables that are constructed using open addressing techniques pose problems when subjected to frequent deletions. The space previously occupied by a deleted record cannot simply be marked *empty* but must be marked *deleted*. This clutters up the hash table and hurts performance. No such problem arises if external chaining is used for collision resolution. Deletion is handled just as it is for any linked list. For coalesced chaining deletion is no problem as long as the cellar has never been full, since deletion can be handled essentially as it is for external chaining. Once the cellar is full and the possibility of coalesced lists exists, then deletion must be handled carefully. An algorithm is given in Vitter (1982). It is (slightly) more complicated and would extract a small performance penalty. When designing a hashing strategy, the frequency of deletion must be considered along with performance and memory requirements.

In Section 7.6 we will apply several hashing methods to the frequency analysis of digraphs. We will see how the theoretical results apply in a specific case.

7.6 Frequency Analysis of Digraphs

We have discussed frequency analysis of digraphs before. In Section 4.9 we used lists for the analysis, and in Section 5.7 we used binary search trees and AVL trees. In this section we will compare four hashing strategies. All four use a division hashing function, but they differ in the collision-resolution strategy: linear rehashing, double hashing, coalesced chaining, and external chaining. We will conclude with a summary of results involving all of the data structures we have used to analyze digraphs.

7.6.1 Hash Function

```
hashtable = array
    [0..tablesize] of bucket;
```

Figure 7.43
Hash table.

The hash table will be of the form shown in Figure 7.43. The hash function must map each digraph (pair of letters) into the integers between 0 and *tablesize*. We accomplish this as follows. Let d_1 and d_2 be the first and second characters of digraph d:

$$d = d_1d_2$$

Let i_1 and i_2 be computed as follows:

$$i_1 = \text{ord}(d_1) - \text{ord}(\text{'a'})$$

$$i_2 = \text{ord}(d_2) - \text{ord}(\text{'a'})$$

where i_1 and i_2 are integers between 0 and 25. Finally, let $I(d)$ be computed by

$$I(d) = 26\, i_1 + i_2$$

where $I(d)$ has values between 0 and 675. Sample values of I are shown in Figure 7.44.

The hash function for digraph analysis is

$$H(d) = I(d) \textbf{ mod } (tablesize + 1)$$

where *tablesize* is to be selected so that *tablesize* + 1 has no small divisors. The frequency analysis results reported in this section are based on *tablesize* = 300. Figure 7.45 shows the values of *H*(digraph) for the first few digraphs from von Neumann (1946).

Figure 7.46 shows the expected search lengths for the four hashing strategies and, for comparison, a binary search of a sorted array. The results are as predicted in Section 7.5.

Digraph	*I*
aa	0
ab	1
ac	2
⋮	⋮
zz	675

Figure 7.44
Values of *I* for digraph analysis.

Digraph	*H*(digraph)
pr	206
re	145
el	115
li	294
im	220
mi	19

Figure 7.45
Home address of the first few digraphs from von Neumann (1946). The table size = 300

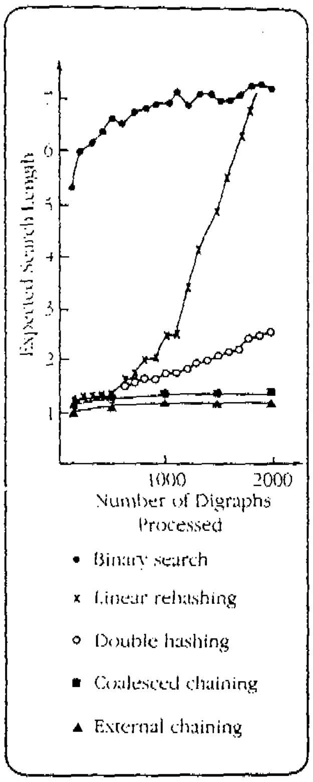

Figure 7.46
Frequency analysis of digraphs. Expected search length.

Recall (see Figure 4.40) that processing 2000 digraphs causes 270 distinct values to be entered into the hash table. The relationship between load factor and the number of digraphs processed, with *tablesize* = 300, is shown in Figure 7.47.

Figure 7.48 shows the average time required to process a digraph for the four hashing techniques and, for comparison, a binary search tree. Also included for comparison is the time required for a ***direct addressing*** scheme. Direct addressing is implemented just like hashing with, in this case, $H(d) = I(d)$. Direct addressing is possible in this case because we can assign a distinct address to each of the 676 possible digraphs. This eliminates collisions, simplifies the algorithms, and ensures that the number of probes to find a digraph is one. The price for this is the requirement for more memory. Direct addressing should not be confused with hashing. A hash function randomizes the elements stored in the hash table. Our direct addressing scheme places the digraphs in the table in alphabetical order.

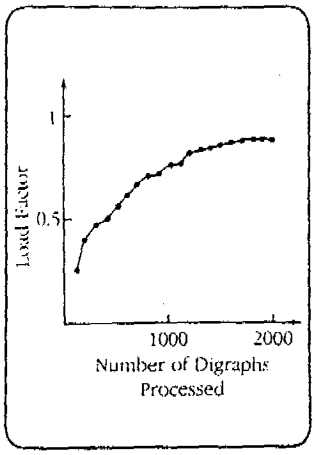

Figure 7.47
Frequency analysis of digraphs. Load factor.