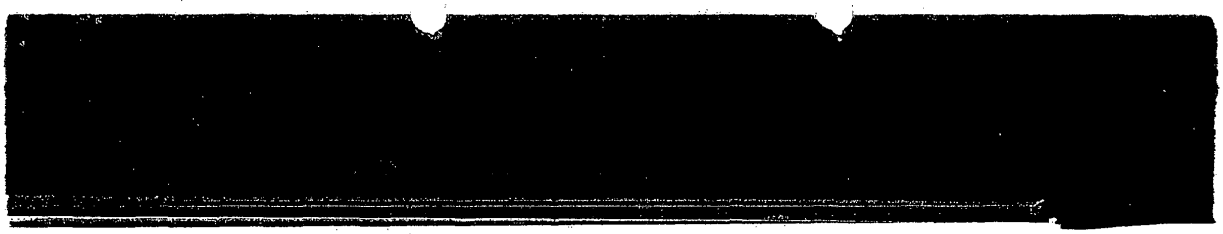
Exhibit 9

Data Structures
with Abstract Data Types and Pascal

Daniel F. Stubbs and Neil W. Webre
California Polytechnic State University, San Luis Obispo

Brooks/Cole Publishing Company
Monterey, California



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 Sreenkin*
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*

Apple is a registered trademark of Apple Computer, Inc.
DEC is a registered trademark of Digital Equipment Corporation.
IBM is a registered trademark of International Business Machines, Inc.
Pascal/MT+™ is a trademark of Digital Research, Inc.

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,

$$\text{memory address of record} = f(\text{key})$$

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:

$$\text{home address} = H(\text{key}_i)$$

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 some

memory or s
be thought o
the key as a s
that key.

One of t
ments. There
analogous. Th
among eleme
among consto
this chapter
discussion of

One of t
probes. The f
on n in ever
for a linked l
a sorted list,
fewest probe
effective sear
hash table, it
All of these
technique fo
not changed

It is con
to let the has
than to calcul
is computed,
an actual me
Figure 7.12.

const tabl
type pos

var table

Figure 7.

Suppose

var table

and that the

$H(\text{key})$

Notice that t
0 and 6, whic

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 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 sc

$$\underline{\times}$$
$$r_1r_2r_3r_4$$

The result is
selection on
example, r_3r

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 = a

and the prog
hardware div
form a hash f

H(key) =

The result w

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

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

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
  08642
 :62963
 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:

```
    0987
    8765
+   4321
------
   13095
```

fold(key) = 13095

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

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

• 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:

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

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

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

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

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

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

key = 'jy'

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;
```

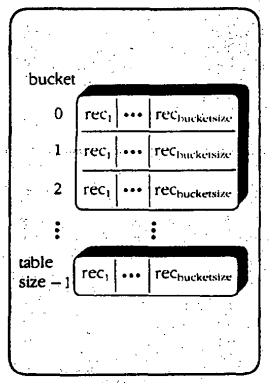


Figure 7.18
Hash table of buckets.

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].

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.

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

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

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.

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.

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*.

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

Figure 7.20
Linear rehashing.

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.

```
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
rehashing pa
1 in Figure 7
any key that l
hashed to 1 l
call this phe
Problem
position 1 m
two rehash p
clustering.

Consid
difference in
Only new k
position 0. K
tion 5.

The exp
can be calcul

Original position
0
1
2
3
4
5
6

Figure 7
hash tabl

The exp
and ***unsucc***
of performan
general way i
that the perf
noted—prim

You ma
other than 1
7.3 would be

k := (k

where 1 <=
tablesize are
tern will cov

Problem 1. Any key that hashes to a position, say *b*, will follow the same rehashing pattern as all other keys that hash to *b*. 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 $4k + 3$.

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 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

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

we define a related step size function

$$c(\text{key}) = [\text{key mod } (tablesize - 2)] + 1$$

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

$$c(421) = 421 \bmod 5 + 1 = 2$$

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,

$$c(624) = 624 \bmod 5 + 1 = 5$$

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

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

Figure 7.25

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.

• 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.

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):

```
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.

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.

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(key) = 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 ha***
perfect hash
hash table ha
collisions, we
that has a giv
that such fun
Perfect
One such co
applications
programmin
procedure,
program's st
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 t
functions th
1973b). Thus
the number
perfect hashi
There a
has propose
suggested so
the times to
fect function
Let us l
are for keys t
of Pascal (se

H(key)

where

L = len

The function
is the intege
integer assoc
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.

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^{13} (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 = 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.

Pascal Reserved Words

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

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

2. The division hash function

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

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 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 $O(tablesize)$ 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.

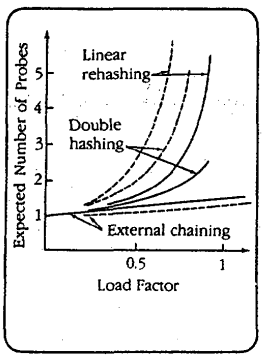


Figure 7.40
Number of probes required for successful and unsuccessful searches in a hash table. —, successful, -------, unsuccessful.

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, 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

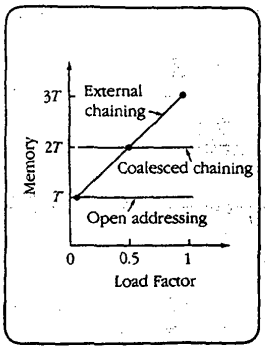


Figure 7.41
Memory requirements when an *element occupies the same* amount of memory as a pointer.

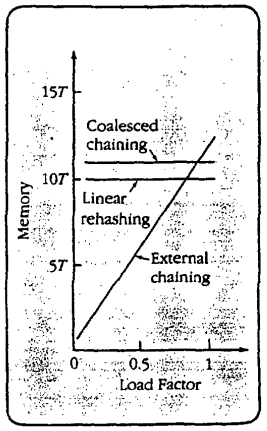


Figure 7.42
Memory requirements when an element occupies 10 times the amount of memory as a pointer.

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:

$$d = d_1 d_2$$

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 :
by

$$I(d)$$

where $I(d$
Figure 7.4
The I

$$\Pi(d) =$$

where *tab*
The frequ
= 300. Fi
from von
Figu
egies and
predicted

Digraph
aa
ab
ac
⋮
zz

Figure 7.4
Values of *I*

Reca
values to I
and the n
Figure 7.4
Figu
four hashi
for compa
addressin
Direct ad
address t
plifies the
is one. Th
ing shoul
elements
digraphs i

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) **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.

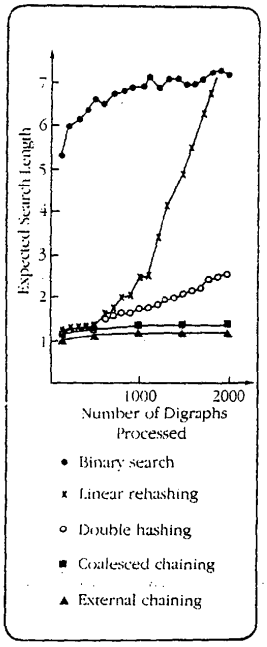


Figure 7.46
Frequency analysis of digraphs. Expected search length.

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.

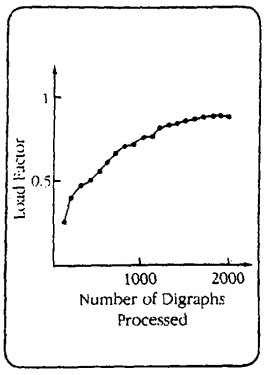


Figure 7.47
Frequency analysis of digraphs. Load factor.

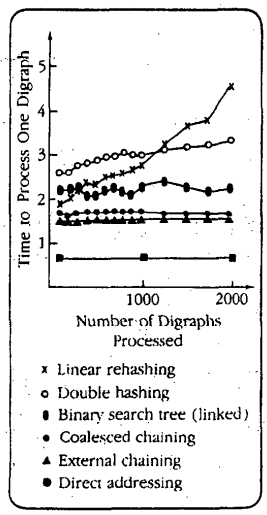


Figure 7.48
Frequency analyses of digraphs. Hashed representations. Average measured time per digraph processed.

The results shown in Figure 7.48 are those expected. Direct addressing, with one probe per digraph, gives the best performance. External chaining and coalesced chaining give essentially the same performance. Using a binary search tree gives better performance than either linear or double hashing, despite requiring more probes during a search. This is because searching a hash table requires more overhead: evaluating the hash function and the array mapping function. If the data structure were bigger, the difference in number of probes required would be larger and the hash techniques would provide better performance than would a binary search tree.

Note that initially, linear probing is faster than double hashing. This is because clusters have little effect on performance, whereas collisions require an extra computation (the step size) for double hashing. As the hash table fills and clusters form, the performance of linear probing deteriorates.

Another consideration for hashing is worst-case performance. Table 7.2 shows the maximum number of probes required to find a digraph as a function of the number of digraphs processed.

Observe that for coalesced and external chaining the maximum number of probes is little different from the average. For linear rehashing and, to a lesser extent, double hashing the maximum number of probes becomes quite large as the table fills. In the worst case, every element hashes to the same home address, and the performance can be disastrous. A general discussion of the worst-case performance of hash files, which compares linear searching, random hashing, and external chaining, is given in Larson (1982).

TABLE 7.2 Comparison of hashing techniques, including the maximum number of probes for a successful search. The table size = 300 and the cellar size for coalesced chaining is 43.

			Linear search		Double hashing		Coalesced chaining		External chaining	
Digraphs processed	Distinct digraphs	Load factor	Avg probes	Max probes	Avg probes	Max probes	Avg probes	Max probes	Avg probes	Max probes
500	165	0.55	1.33	6	1.41	7	1.15	2	1.15	2
1000	225	0.75	2.44	57	1.72	7	1.27	3	1.22	3
1500	256	0.85	5.44	103	2.12	23	1.30	3	1.23	3

7.6.2 *Frequency Analysis Summary*

Let us now review and compare all the approaches we have used for our frequency analysis problem. We have used three data structures: lists, binary trees, and sets. The analysis with lists involved two representations—array and linked—and four different orders—chronological, sorted, frequency ordered, and self-organizing. The analysis with binary trees used linked representations of both binary search trees and AVL trees. The analysis using sets was based

on an array representation of a hash table and considered four collision-resolution policies—linear rehashing, double hashing, coalesced chaining, and external chaining.

Figure 7.49 shows the expected search lengths for five representative data structures—sequential search of a list in chronological order, binary search of a sorted list, binary search tree, hashing with linear rehashing, and hashing with coalesced chaining. A semilog plot is used because of the range of search lengths.

Figure 7.50 compares the performance of the data structures considered in Figure 7.49. A semilog plot is used so that differences can be clearly seen. A comparative analysis of data structures includes performance (both average and worst case), memory requirements, algorithm complexity, and particular strengths and weaknesses. Figures 7.49 and 7.50 show the results of empirical studies of performance. Memory requirements are discussed in the chapters in which we introduced the data structures. All of the data structures in this text are (relatively) simple. We can, however, see one impact of complexity by comparing Figures 7.49 and 7.50. Sequential search of a list requires many more probes than does binary search of a sorted list. However, because one probe during a binary search is more complex than one probe during a sequential search, the number of elements must be large before a binary search pays off. The meaning of "large" varies with the processor and software used, and for our example it appears that large means 150 or more elements.

An example of a data structure weakness is the order of hashed data. Since the elements are stored in random order, it would be difficult to find all elements whose key values are in a given interval. It would be easy to do this for data in a sorted list, and that is an advantage of that data structure. As we suggest here, selection of a data structure involves matching application requirements with data structure strengths. We have seen in the case of a simple problem, frequency analysis of digraphs, that there are many trade-offs to consider.

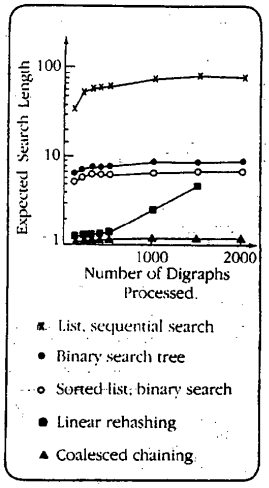


Figure 7.49
Frequency analysis of digraphs. Comparison of methods—log plot. Expected search length.

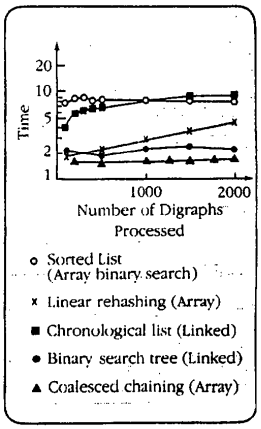


Figure 7.50
Frequency analysis of digraphs. Comparison of methods—log plot. Average measured time per digraph processed.

7.7 Summary

In this chapter we have studied data structures whose structure is setlike. The only relationship of interest among the elements is that they are members of the set. We have distinguished between two kinds of sets—those whose elements are atomic and those whose elements are structured.

We have seen that a particularly important implementation technique for sets of structured elements is hashing. We studied several hashing functions and several collision-resolution (or rehashing) strategies. The rehashing strategies upon which we concentrated were linear rehashing and double hashing (both open address methods) and external and coalesced chaining. Performance analyses and comparisons were made in Sections 7.5 and 7.6. They clearly showed that hashing is an excellent method in terms of the execution times of its algorithms.

One question that we have not yet addressed is a comparison of the

complexity of hashing algorithms with those of lists and trees. We have given no complete package for the hashed implementation of sets, but rather we have presented several of the key algorithms. In previous chapters we closed by giving tables containing at least crude measures of the complexity of the various packages presented in the chapter. As a substitute here, Table 7.3 compares the complexity of a single operation—*findkey*—for various data structures and their implementations. We see that in terms of the simple measure we are using, complexity of the hashed (open address) implementations of *findkey* are comparable with those of the other data structures.

TABLE 7.3 Complexity of *findkey* for various data structures.

Data type	Representation	Lines of code for *findkey*
List	Array	9
List	Linked	14
Sorted list	Array (binary search)	17
Binary search tree	Linked	12
Set	Hashed, linear rehash	13
Set	Hashed, double hashing	15

The applied problems section at the end of the book suggests additional applications of the data structures discussed in this chapter. Problems related to this chapter's topics are 2, 3, 4, 5, and 10.