Chapter 6 Storing, sorting, searching and indexing

The general term 'data structures' is given to any way of organising information handled by a program. Simple data structures that you will have met already are one-dimensional arrays and two-dimensional arrays. One-dimensional arrays facilitate random access to a list of numbers or strings. Two-dimensional arrays also allow random access and impose an organisation on the data that reflects the reality of a two-dimensional data source. For example a population map is a two-dimensional table of integers where each integer is the population of, say, a square mile zone of a geographical area. It is natural to retain this two-dimensional ordering within the program and it makes for easier and more natural programming.

The material in this chapter is concerned with building more complex data structures using combinations of arrays. The first raison d'etre of data structures is to retain a 'natural' organisation and thus ease the task of the programmer. This is very important as easy and natural programming structures make for correct programs.

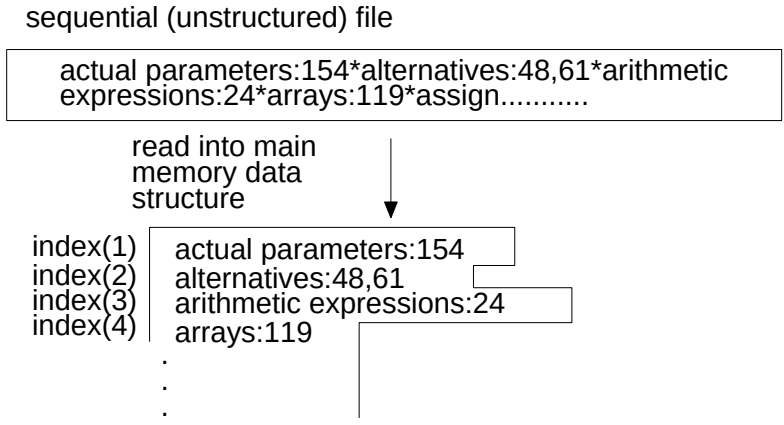
The second reason for organising data into structures is that individual 'entries' may vary considerably in size. Allocating the same space for each entry, i.e. space large enough for the largest entry, would be wasteful. Instead we may allow the entries to occupy exactly the storage space that they require, and set up a table that 'points' to the start of each entry.

The third important reason for using data structures is that if our collection of data is very large, the imposition of a data structure may to necessary to enable efficient searching. A related consideration here is the amount of memory space available. It may be that only part of the data set, held on disk say, can be brought into the memory at any time.

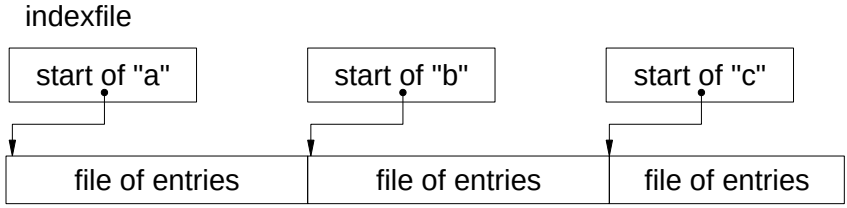
The information stored in connection with a particular application is called a data set, data base or data bank. A useful concept is that the data structure mirror-images a natural or artificial organisation of data in the real world. We may for example initialise, first thing in the morning, the main memory data structures of a stock control program and organise a sequential file into a series of departments, articles, classes etc. This data may be changed during the day, as stock levels change, for example, and be

dumped back into a sequential filre to sleep in its 'unstructured' form overnight. This in not the whole story because the disk files may themselves be subject to an organisational framework. This is certainly the case when the main store data structure is such that it can only hold a very small part of the data set. In this case the files themselves will be organised to reflect the organisational framework of the data, and the data structure in this case resides in the file structure. The general topic of structured files and data structures has come to be known as database organisation. In this chapter we will be concerned with the simpler problem of main memory data structures. These may contain information that is built up by a program while it is running, or they may be initialised from a sequential file where all of the file is taken into the data structure.

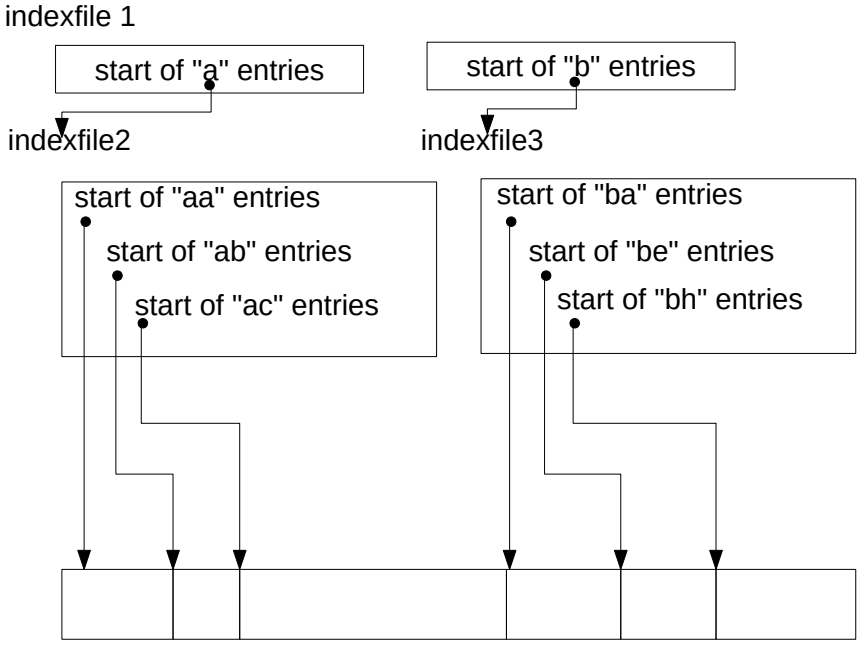
The distinguishing features of these cases can be illustrated. Consider first of all a small index for a book. This may have to undergo transformations such as addition of a new entry, sorting, checking for duplicate entries etc. The length of each entry is short and there may only be a few entries. The entire index could be held in a sequential file, and, providing each entry is de-limited, read into an array of strings. The array can then be randomly accessed and otherwise manipulated by the program. The whole of the data base is contained in the main memory data structure.



Now if we intend to store and manipulate an English dictionary the situation is completely different. The data base cannot be held in main memory and the required structures will be set up on disk. The disk file(s) will be subject to an imposed framework.

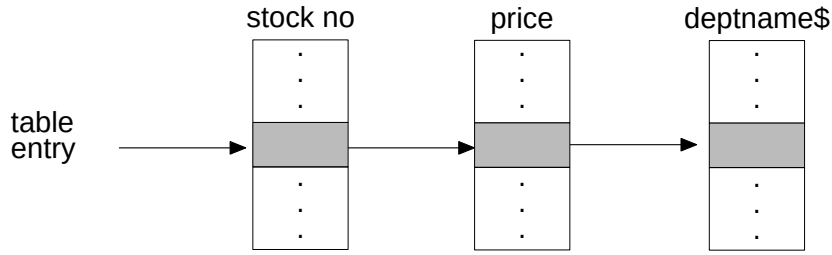


Depending on the manipulation that has to be performed by the program such a simple structure may be inadequate and the hierarchy may have to be 'deepened'



6.1 Tables

Many computer applications involve storing a table of information where each entry in the table consists of several related values. For example we may want to store a stock table, where each entry in the stock table consists of a stock number, a price and a department name. An appropriate data structure could be constructed in Basic from three parallel arrays.



Any three corresponding elements in these arrays will then be referred to as a table entry. Each element of a table entry will be referred to as a field of that entry. A program that reads a stock list from an input file into this structure, and which at the same time prints out a short list for the items sold in the food department might be:

```
10   DIM stockno(100), price(100), deptname$(100)
20   indata = OPENIN( "stockdata")
30   INPUT# indata, noofitemsinstock
40   FOR item = 1 TO noofitemsinstock
50     INPUT# indata, stockno(item),
         price(item), deptname$(item)
60     IF deptname$(item) = "food" THEN
         PRINT stocknotitem), price(item)
70   NEXT item
80   CLOSE# indata
```

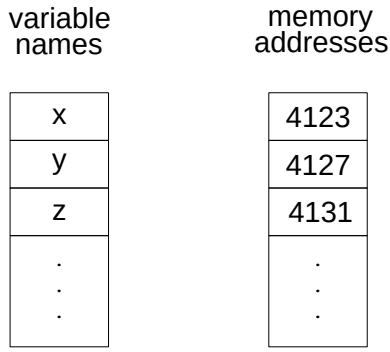
For the purpose of illustrating a technique, or in cases where a fairly small table of constant values is required by a program, the table could be read from DATA statements. Although this is convenient, it suffers from the disadvantage that storage space is allocated twice for the data; once as part of the program in the DATA statements and again when the program is run and space is allocated for the arrays.

In the following examples we will omit the details concerning the initialisation of a data structure from a file or DATA statement if this is simply a sequential transfer of information to arrays.

6.2 Searchinga table - linear search

Having seen how to set up or organise a simple table we will now look at some common operations that are performed on tables. Searching a table for a particular entry is a common problem. An example is a Basic interpreter, the program that processes and obeys your Basic programs. An interpreter program builds up a table of variable names. Each entry in the table consists of the variable name together with a

memory address that the interpreter allocates to that name the first time it encounters it.



Each time a variable name is encountered the interpreter has to access the table for a memory address. The statement:

```
x = (x + y)/(a*a + b)
```

would involve six accesses to this table. Interpreting a program may involve thousands of table accesses and the method used for the search becomes critically important in such applications. Now when we are searching tables we usually have given one field of an entry and the point of the search is to find the other field(s) associated with this 'key' field. For example in the stocklist we may be using the stock number entry as a key. The problem is then: given a particular stock number, find the associated price.

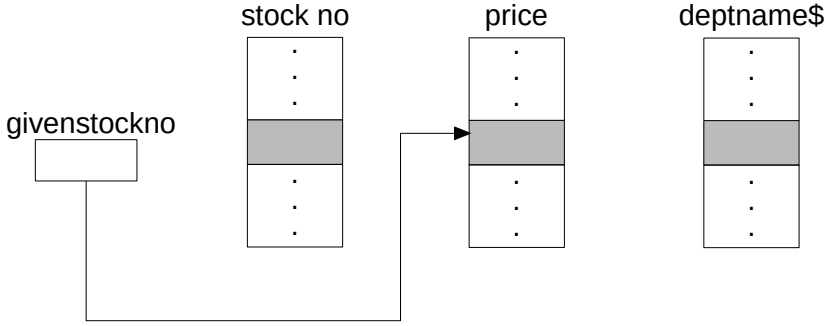


Table searching methods are concerned with organising the table in such a way that finding the field or fields associated with a key is quick and efficient. The simplest approach to organising such a table is to store the entries in the table in the order in which the information was originally presented to the program. Subsequently, when the price of an item with a given stock number is required, the technique known as linear search can be used to find the entry containing that stock number. This involves scanning through the table from location 1 onwards until the required entry is found.

```
400  DEF PROCfindprice(givenstockno)
410    LOCAL probe
420    probe = 0
430    REPEAT
440      probe = probe + 1
450    UNTIL stockno(probe) = givenstockno
460    requiredprice = price(probe)
470  ENDPROC
```

(We have here assumed that the given stock number will be found in the table.) Consider now a complete customer order consisting of a list of stock numbers for the items a customer requires. Let us assume that he requires only one of each type of item. The total number of items required is indicated at the start of the order. The cost of his order can be calculated by:

```
 10  DIM stockno(100), price(100)
 20  PROCsetupstocktable :REM from file or DATA
 30  totalcost = 0 : INPUT "No of items",items
 40  FOR item= 1 To items
 50    INPUT "Stock no:"givenstockno
 60    PROCfindprice(givenstockno)
 70    totalcost = totalcost + requiredprice
 80  NEXT item
 90  PRINT "Total cost is "; totalcost
100  END
```

In fact linear search could be carried out on the DATA statements, without the need for the stock table to be transferred to arrays. RESTORE would be used prior to each search. However, this could not be done for the more efficient search methods introduced later.

If there are n records in a table, linear search involves, on average, the examination of n/2 entries before a required record is found. For example if the stocklist is 1000 items long and each order to be processed contains on

average 350 items, then processing an order involves examning 175000 entries. In most applications, particularly where n is large, this unacceptable. In the following sections, we shall be discussing different ways of organising the information in a table, usually with a view to finding an entry containing a given key more quickly than is possible with linear search. To facilitate more efficient methods of searching we first look at sorting.

Exercises

1 Write a program that repeatedly accepts input of a French word and responds with the equivalent English word. Use a fairly small dictionary for the exercise and initialise it from DATA statements. Assume that each word has a unique translation. If a given word is not found in the dictionary, the program should report this. The program should use linear search to look for a dictionary entry.

2 Modify the previous program so that the user can specify whether he requires translation of a word from French to English or English to French.

3 Write a program that will analyse a short piece of English text and report the total number of different words used in the text and the number of times each word was used. The text should be read from a file (write a separate program to set up the file). As it reads the text, the program will have to build up a table of words encountered, together with a count of the number of occurrences of each word so far. Differences between upper and lower case letters can be ignored by converting all lower case letters into upper case.

4 Use the animation procedures developed in Chapter 4 (Section 4.1) to animate a linear search for a particular entry. Each entry should be highlighted in a different colour as it is examined and the required entry should be highlighted in a flashing colour when it has been found.

5 The membership list for a society consists of an integer n followed by a list of n entries where each entry consists of a person's name (surname first) and a membership number (an integer). Write a program that will read and store the membership list in an appropriate table. The program should then input an integer m followed by m membership numbers. For each number presented, the program should find and print out the name of the member with that number.

6.3 Ordered data - sorting

It is often more convenient or even necessary for the entries involved in an application to be stored in some specified order. This ordering is usually determined by or of the entry fields. For example, we can organise a set of entries so that a particular string field appears in alphabetic order. Alternatively we may organise them so that a numeric field appears in increasing or decreasing order We can then refer to a table as being sorted on a particular field.

As an example, we consider the problem of sorting the stock entries used previously so that the stock numbers are in increasing numerical order. We present three of the simplest sort algorithms that can be used to sort the entries stored in an array into a required order. Let us assume that initially we have a records stored in the stocklist arrays used previously.

Simple exchange sort

A simple exchange sort is perhaps the easiest algorithm to describe. It is also the least efficient. For a simple exchange sort we can describe the algorithm, using the stocklist example:

```
find the entry with the smallest stock no.
swop it with the first entry (stockno(1), price(1) etc.)
find the entry with the second smallest stock no.
  (at this stage we need only look at 2nd entry onwards)
swop it with the second entry
find the entry with the third smallest stock no.
  (at this stage we need only look at 3rd entry onwards)
  .
  .
etc.
```

In other words:

```
 10  DIM stockno(100), price(100)
 20  PROCsetupstocktable
 30  PROCsortstocktable(noofitemsinstock)
 40  PROCoutputstocktab1e
 50  END

400  DEF PROCsortstocktable(n)
410  LOCAL i
420    FOR i = I TO n-1
430      PROCfindsmallestentryfrom(i)
440      PROCswop(i, posnsmallest)
450    NEXT i
460  ENDPROC
```

```
500   DEF PROCfindsmallestantryfrom(i)
510   LOCAL next
520   posnemallest = i
530   FOR next = i+1 TO n
540     IF stockno(next) < stockno(posnsmallest) THEN
          posnemallest = next
550     NEXT next
560   ENDPROC

600   DEF PROCswop(i, j)
610     LOCAL temp
620     temp = stockno(i)
630     stockno(i) = stockno(j)
640     stockno(j) = temp
650     temp = price(i)
660     price(i) = price(j)
670     price(j) = temp
        .
        . etc. for all other fields in the entry
        .
680   ENDPROC
```

PROCsetsupstocktable will initialise the arrays from a file or DATA statements and PROCoutputstocktable will print the arrays.

Bubble sort

Depending on the state of the data a bubble sort can be considerably more efficient than an exchange sort. Sometimes data is already partially ordered and in such a context a bubble sort is preferred. Consider the following program fragment:

```
FOR i = 2 To n
  IF stockno(i) < stockno(i-1) THEN PROCswop(i,i-1)
NEXT i
```

If this is executed once, the entry with the largest stock number is picked up and carried to the end of the arrays, and in the process some of the other entries are moved closer to their correct position in the ordering. If we execute the same fragment of program again, but this time using:

```
FOR i = 2 TO n-1
```

the entry with the second largest stock no. will be carried to the second last position in the arrays.

Repeated application of this process eventually sorts the entries into order:

```
400  DEF PROCsortstocktable(n)
410  LOCAL i, last
420    FOR last = n TO 2 STEP -1
430      FOR i = 2 TO last
440        IF stockno(i) < stockno(i-1) THEN
             PROCwop(i,i-1)
450      NEXT i
460    NEXT last
470  ENDPROC
```

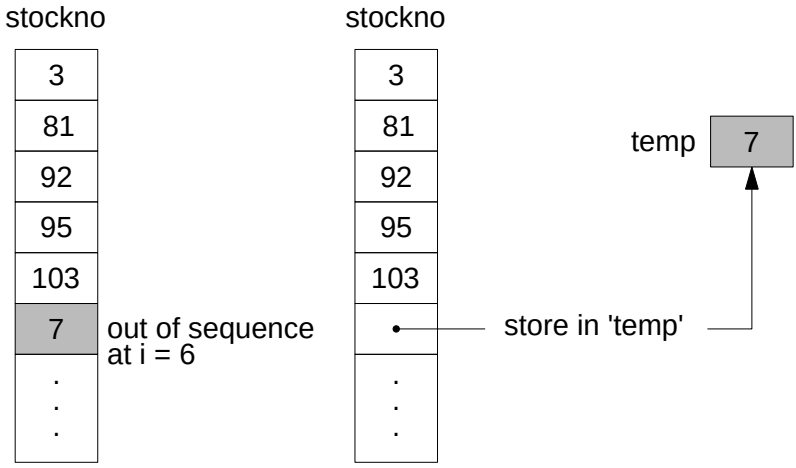
This first approximation can be considerably improved. Fact time lines 430-450 are obeyed, not only is one entry carried to its correct position, but other entries may also be moved closer to their final positions. Thus lines 430-450 may not have to be obeyed n-1 times before the entries are in order. If at some stage, obeying lines 430-450 does not move any entries then they must already be in order and the process can be terminated. This is an occurrence that that is more and more likely to happen as n increases.

Secondly, in the process of obeying lines 430-450, we may find that 'stockno(last)' is already in its correct position. This can be detected while lines 430-450 are being obeyed by keeping a note of the number of the last record moved down. Thus we have:

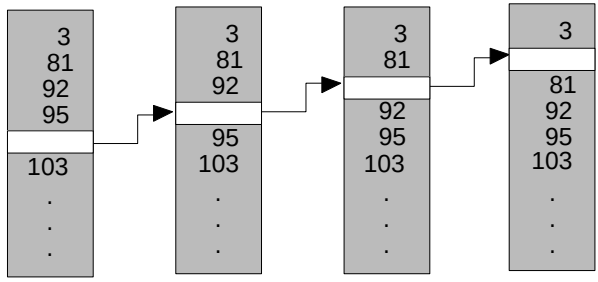
```
400  DEF PROCsortstocktable(n)
410  LOCAL i, last, lastonemoveddown
420    last = a
430    REPEAT
440      lastonernoveddown =0
450      FOR i = 2 TO last
460        IF stockno(i) < stockno(i-1) THEN
             PROCswop(i, i-1) : lastonemoveddown = i-1
470      NEXT i
480      last = lastonemoveddown
490    UNTIL last < 2
500  ENDPROC
```

Sifting sort

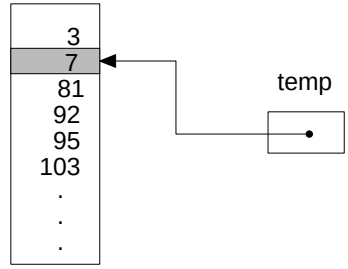
Finally we modify the central idea used in the bubble sort algorithm and introduce the algorithm known as 'sifting'. When two entries are found to be in the wrong order, instead of simply exchanging them, we move the second one back past all the entries which should came after it, moving all these entries forward one place to make rom for it. The first operation to be performed in this algorithm is to look for the first out of sequence number and store it temporarily.



The next step is to 'sift' until we find the correct position in the sequence for the contents of 'temp'.



Finally the contents of 'temp' are inserted back into 'stockno'.



The process is best imagined in the illustrations as the 'white space' made 'empty' by storing the number to be repositioned in 'temp', sifting backwards. The program is:

```
200
  400  DEF PROCsortstocktable(n)
  410  LOCAL i
  420    FOR i = 2 TO n
  430      IF stockno(i) < stockno(i-1) THEN
             PROCsiftfrom( i )
  440    NEXT i
  450  ENDPROC

  500  DEF PROCsiftfrom(i)
  510  LOCAL j, tempno, tempprice, stopsifting
  520  tempno = stockno(i) : tempprice = price(i)
         ... etc for other fields
       530 j = i-1 : stopsifting = FALSE
  540  REPEAT
  550    stockno(j+1) = stockno(j)
  560    price(j+1) = price(j)
           ... etc. for other fields
  570    IF j = 1 THEN stopsifting = TRUE
         ELSE IF tempno > stockno(j-1) THEN
           stopsifting = TRUE
         ELSE j = j-1
  580  UNTIL stopsifting
  590  stockno(j) = tempno : price(j) = tempprice ...
  610  ENDPROC
```

Exercises

1 Write a program that reads the society membership list used earlier. The list should be stored in a table and the program should use 'bubble sort' to sort these records into alphabetical order according to the member's names. The list should then be printed with the names in order.

2 The stock list for a small department store consists of a list of records, where each record contains three items:

```
department code  (1 character)
stock number     (an integer)
price            (a real number)
```

For the purposes of this exercise you can either construct a sample stocklist in DATA statements or write a short program that inputs a sample stocklist and saves it in a file. Write a program that prints the stock list in such a way that the department codes are in alphabetical order and, within each department, the stock numbers are in ascending numerical order. Two approaches are possible.

EITHER define a FNinorder(recordno1, recordno2) and use this to compare two records during sorting,

OR sort the stock twice, firstly on the stock numbers and then on the department codes, making sure that two records with the same department code remain in order determined previously by their stock numbers.

6.4 Ordered data - binary chopping

Now that we have examined techniques that sort entries into order we can return to the problem of table searching. If the information in a table is already ordered on a purticular field, or can be sorted before further processing takes place, then the technique known as 'logarithmic search' or 'binary search' can be used for finding the entry that contains a given key in that field. Use of this search algorithm requires the examination of approximately log2n entries in order to find the entry containing a given key.

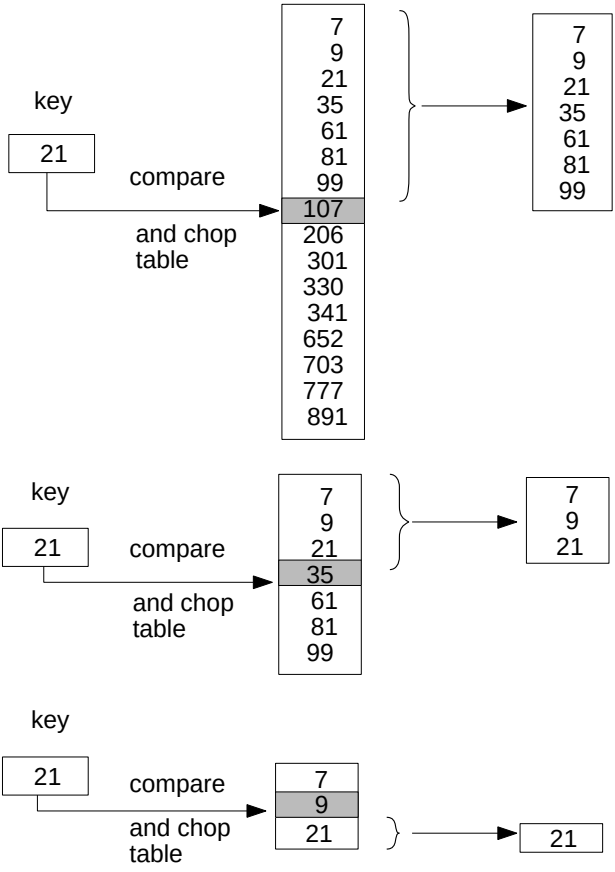
This algorithm starts by examining the entry approximately in the middle of the table. The given key is compared with the appropriate field of this middle entry and if they are the same, the search is terminated successfully. if the given key comes before the value in this field, the first half of the table must be searched, otherwise the last half of the table must be searched. The same process is repeated on the first half or the last half of the table, the area in which the required key is known to lie being repeatedly halved in this way until the key has been found.

In the program that implements this technique, the variable 'probe' has as its final value the number of the entry containing the given stock number. The entries in the table are assumed to be in ascending order of stock number.

```
 10  DIM stockno(100), price(100)
 20  PROCsetupstocktable
 30  PR0Csortstocktable(noofitemsinstock)
       .
       .
       .
700  DEF PROCfindprice(givenstockno)
710  LOCAL first,last, stopsearching,found
720    first = 1
730    last = noofitemsinstock
740    stopsearching = FALSE : found = FALSE
750    REPEAT
760      PROCchoptable
770    UNTIL stopsearching
780    IF found THEN requiredprice = price(mid)
       ELSE PRINT "Entry does not exist. " :
         requiredprice = 0
790  ENDPROC
```

```
800  DEF PROCchoptable
810    mid = (first + last) DIV 2
820    IF stockno(mid) = givenstockno THEN
         stopsearching =TRUE : found = TRUE : ENDPROC
830    IF stockno(mid) > givenstockno THEN
         last = mid — 1 ELSE first = mid + 1
840    IF first > last THEN stopsearching = TRUE
850  ENDPROC
```

The process can be illustrated for a particular arbitrary sequence :



It is interesting to compare the average number of steps required by a linear search in a table of n records with the average number of steps required by a logarithmic search, for different values of n:

number of entries	linear search averagv no. of steps is n/2	logarithmic search average no. of steps is log2n
4	2	2
16	8	4
128	64	7
1024	512	10
8192	4096	13

It must be remembered that sorting algorithms are themselves fairly time consuming and this must be balanced against the subsequent saving in look-up time. If also, new data is to be continually added to the table, it would be very inconvenient to have to keep moving the information already in the table in order to insert a new entry in its correct position. However, there are applications in which all the entries in the table are present in advance and where the table needs to be sorted for some other reason: for example, so that an ordered listing of the table can be generated for reference by a human user.

Other techniques are available for organising and accessing tables in cases where the table does not need to be ordered or where new entries are continually being added. Such techniques are described later.

Exercises

1 Write a program that sorts the French-English dictionary used earlier so that the French words are stored in alphabetical order.

2 Modify the earlier French-English word translator program so that it uses binary search to find a French word in the ordered table produced by the previous exercise.

3 Use the procedures of Chapter 4, Section 4.1, to animate a sifting sort.

6.5 Direct access

If the keys to be looked up in a table are such that we can define a function that calculates a unique value from each possible key and the values calculated by the function are In a suitably small range, then this function can be used for deciding where in a table to insert an entry associated with a given key and where to find it later.

Consider the problem of storing in a table the name of mach animal kept in a zoo together with the name of the keeper assigned to look after each animal. Let us impose the highly artificial restriction that all the animals kept have

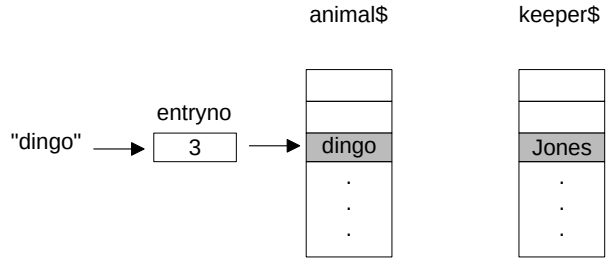
names beginning with different letters. We require a program which, given the name of the animal, prints the name of the keeper assigned to look after that animal (or an error message indicating that no animal of that description is kept). We can use the initial letter of an animal's name te determine the location in the table in which the animal's entry is to be stored.

We need two parallel arrays and it is convenient in this case to number the locations of these arrays from 0 to 25.

```
10 DIM animal$(25), keeper$(25)
```

Remember that the locations in a BASIC array are numbered from 0 upwards, but location 0 is often ignored. Here we will use slot 0 for the animal whose name begins with "a". A number in the range 0 to 25 can be easily calculated from a given 'first1etter$':

```
entryno = ASC(firstletter$) - ASC("a")
```



This table can be initialised from DATA statements:

```
200  DEF PROCsetupzootable
210  LOCAL noofanimals, i, entryno
220  READ noofanimals
230  FOR i = 1 TO nooiianimals
240      READ nextanimal$, nextkeeper$
250      firstletter$ =LEFT$(nextanimal$,1)
260      entryno = ASC(firstletter$) - ASC("a")
270      animal$(entryno) = nextanimal$
280      keeper$(entryno) = nextkeeper$
290    NEXT i
300  ENDPROC
```

After the DIM statement at line 10 has been obeyed, all array locations contain the empty string. If fewer than 26 animals are kept, then after the table has been initialised some of the animal fields will still have an empty string

stored in them. An empty string indicates that there is no animal whose name begins with the corresponding letter.

To respond to an inquiry about a particular animal we need:

```
400  DEF PROCfindkeeper(givenanimal$)
405  LOCAL entryno
410    entryno = ASC(LEFT$(givenanimal$,1)) - ASC("a")
420    IF animal$(entryno) = givenanimal$THEN
         PRINT "Keeper is "; keeper$(entryno)
       ELSE PRINT "There is no animal of that name."
430  ENDPROC
```

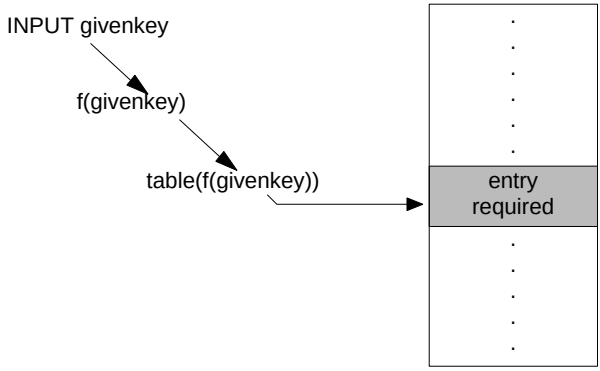
There is now no searching required to find the entry containing information about the animal with a given name, hence the term 'direct access'.

As another example of direct access, consider the stock table used previously. If the stock numbers are such that The last two digits of each number uniquely identify an item we could use these two digits to directly access the stock table. We can initialise the stock table with:

```
 10  DIM stockno(99), price(99)
200  DEF PROCsetupstocktable
     .
     .
     .
210  LOCALnoofotemsinstock, i, entryno
220  READ noofiteinsinstock
230    FOR i = 1 TO noofitemsinstock
240    READ nextstockno, nextprice
250      entryno = nextstockno MOD 100
260      stockno(entryno) = nextstockno
270      price(entryno) = nextprice
280    NEXT i
290  ENDPROC
```

Again there will be empty space in the table if we have fewer than 100 items in stock. Since numeric array locations are initialised to 0, the unused locations still contain 0 after the stock list has been read. Now for a given stock number, the location (if any) containing the associated price is 'price(givenstockno MOD 100)'.

In general, if the function f is to be used for direct access to a table of entries, we can picture f being used as follows:



In the first example the function was:

```
f(givenkey$) = ASC(LEFT$(givenkey$,1)) - ASC("a")
```

and in the second example

```
f(givenkey) = givenkey MOD 100
```

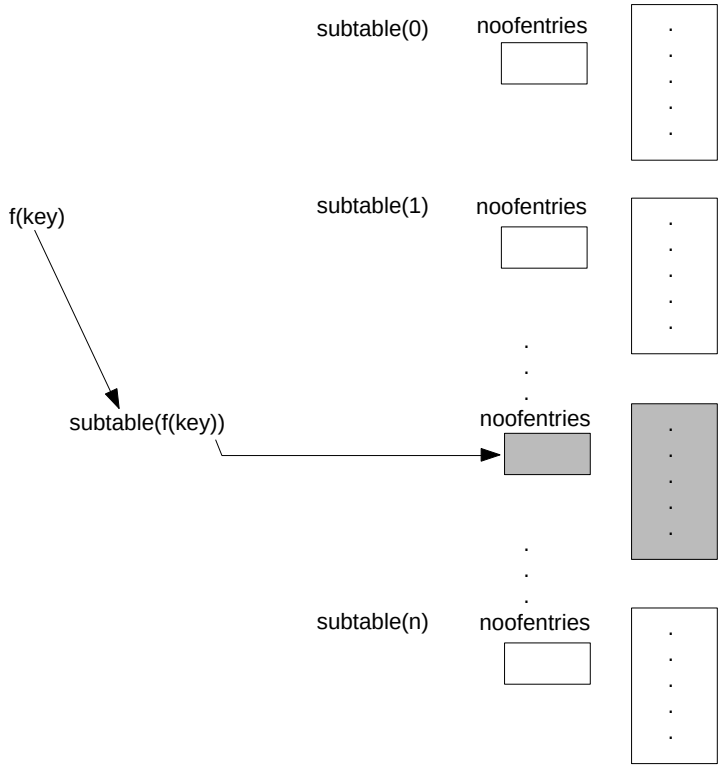
The first function has 26 possible values and the second 100, so 26 entries can be stored in the first case and 100 in the second.

6.6 Direct access to a subtable

Even if we cannot define a function that calculates a unique table subscript for each key, we can easily extend the ideas introduced in the previous section to the situation where more than one key gives rise to the same function value. One way of doing this involves allocating a subtable to hold the entries corresponding to one value of the access function. Given a particular key we evaluate f(key) and use this value to tell us in which subtable the entry with that key is stored.

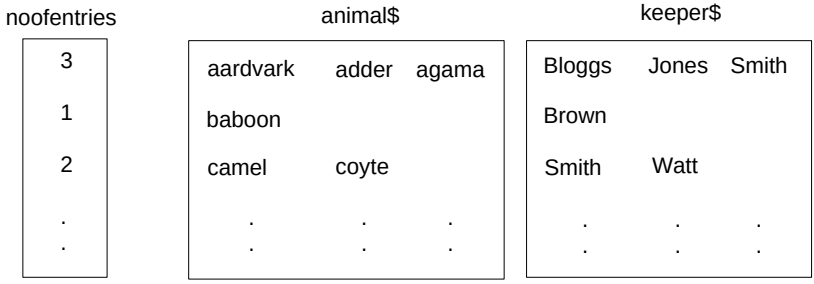
In the zoo-keeper example, we will allow for up to 10 animals whose names begin with each initial letter. We will thus allow for a maximum of 260 animals, some of which have names beginning with the same letter. Associated with each subtable is an integer value indicating the number of entries in that subtable. We can use linear search for storing and accessing entries within one such subtable. The complete table, then, will consist of 26 of these subtables, one for each initial letter.

In general, we can picture the process of accessing the table as:



We can set up this data structure for the zoo table using three arrays (or more if there were more fields associated with each animal): a one-dimensional array containing the 'noofentries' values, and two two-dimensional arrays to store the subtables:

```
10 DIM noofentries(25), animal$(25,10), keeper$(25,10)
```



We can initialise the structure from DATA statements

containing animal-keeper pairs (although initialisation from a file would be more realistic).

```
200  DEF PROCsetupzootable
210  LOCAL noofanimals, i,
         nextanimal$,nextkeeper$, entryno
220    READ noofanimals
230    FOR i = 1 TO noofanimals
240      READ nextanimal$, nextkeeper$
250      subtable = ASC(LEFT$(nextanimal$,1)) - ASC( "a")
260      noofentries(subtable)=noofentries(subtable)+1
270      entryno=noofentries(subtable)
280      animal$(subtable, entryno) = nextanimal$
290      keeper$(subtable, entryno) = nextkeeper$
300    NEXT i
310  ENDPROC
```

Additional statements would of course be needed if we wanted the program to recognise when a subtable became full.

To find a given entry in the table we could call the following procedure:

```
400  DEF PROCCfindkeeper(givenanimal$)
410  LOCALsubtable, examined,found,there
420    subtable = ASC(LEFT$(givenanimal$,1)) - ASC("a")
430    examined = 0 : found = FALSE : there = TRUE
440    REPEAT
450      examined = exanrined + 1
460    IF examined > noofentries(subtable) THEN
         there = FALSE
       ELSE
         IF animal$(subtable,examined) = givenanimal$
         THEN found = TRUE
470    UNTIL found OR NOT(there)
480    IF found THEN PRINT "Animal's keeper is ";
         keeper$(subtable, examined)
       ELSE PRINT "This animal is not in the zoo."
490  ENDPROC
```

This example illustrates a simple case of a 'hierarchical' table-access method. We use one method (direct access in this case) to find the appropriate subtable in which a given key could be found and then proceed to search for the given key in that subtable. In the above example we used linear search in the subtables, but in general any of the methods discussed could be used for organising and accessing the information within a subtable. In fact a subtable might itself be subdivided into further hierarchies of subtables.

Such hierarchical methods are extremely important when a

complete table is too large to be held main menwry. The subtables can be held on disk file, and only when a program has selected a particular subtable would that subtable be copied into main memory.

6.7 Open hash tables

The method of the previous section suffers from the disadvantage of breaking down as soon as one of the subtables is full. It is also rather uneconomical in terms of the amount of memory space used. It is surely unrealistic in the zoo-keeper example to allocate the same amount of space for animals whose names begin with X, Q or Z as for those whose names begin with B, say.

An open hash table provides a widely used method for dealing with the situation where we cannot define a function that converts each key into a unique value. The method makes more flexible use of available storage space than does the method described in the previous section.

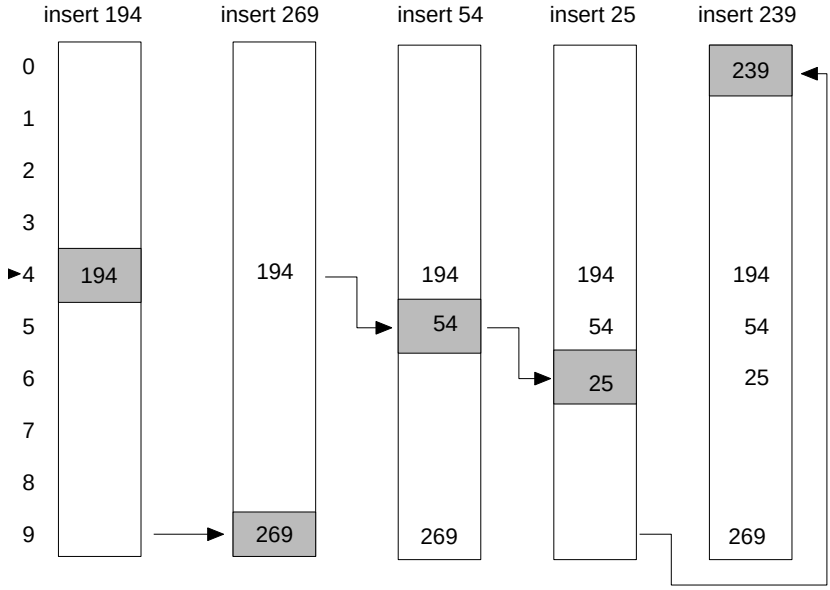
An open hash table consists of a single table similar to that used in the direct access method. The range of values produced by the access function (or 'hash function') corresponds to the range of the subscripts for the table, but we accept the possibility that several keys may result in the sane hash function value.

We describe first the simplest form of access mechanism for an open hash table. To find the location in which to insert the entry containing a given key, the hash value for that key is calculated and the entry is inserted in the first empty location from that point onwards. To find the entry in the table containing a given key, we evaluate the hash function and conduct a linear search in the table from that point onwards. The table is treated as being circular, i.e., if a search reaches the end of the table, the search carries on from the start of the table.

As a simple example, let each entry consist of a single integer and let us illustrate the process of inserting a sequence of these integers in a hash table, 'table'. We define the hash function value for a given key as:

```
h(key) = key MOD 10 (i.e. the last digit in the integer)
```

In general, the use of a hash function of the form 'key MOD n' is known as division hashing. Remember that this is an unrealistically small table used merely to introduce the technique. The point of the method is to speed up access to a large table using an access function that may map different keys into the saute value, without the need to allocate a separate table for each possible access function value. We can use the 0 values that are inserted initially in all BASIC array locations to recognise 'empty' locatione.



As a general rule, when inserting a new value in any table, we should check that it is not already there, and when searching for a given key in a table we should cater for the possibility that it is not there. These two processes are easily combined for our open hash table:

```
   10   DIM table(9)
          .
          .
          .
  110   INPUT givenkey
  120   probe = givenkey MOD 10 : there=TRUE : found=FAL8E
  130   REPEAT
  140     IF tab1e(probe) = givenkey THEN found = TRUE
          ELSE IF tab1e(probe) = 0 THEN there = FALSE
          ELSE probe = (probe + 1) MOD 10
  150   UNTIL NOT(there) OR found
  160   IF found THEN PRINT "Given key is already there."
        ELSE table(probe) = givenkey :
          PRINT "Key has been inserted in table."
```

Say, in the zoo-keeper example, we wish to allow for up to 26 animals, but we may have more than one beginning with the same letter. We can use the same table as was used for direct access:

```
   10   DIM animal$(25), keeper$(25)
```

This time we organise it as an open hash table. The same

search algorithm can be used to find an empty alot in which to insert a new animal, or to find the slot containing a given animal. The following procudurv carries out the search process required:

```
500  DEF PROCsearchfor(givenanimal$)
510  LOCAL probe, found, there
520    probe = ASC(LEFT$(givenanimal$,1)) - ASC("a")
530    found = FALSE : there =TRUE
540    REPEAT
550      IF animal$(probe) = givenanimal$ THEN
           found = TRUE
         ELSE IF animal$(probe) = "" THEN
           there = FALSE
         ELSE probe = (probe+1) MOD 26
560    UNTIL found OR NOT(there)
570  requiredslot = probe : animalfound = found
580  ENDPROC
```

Each time this procedure is called, it transmits information out to where it was called via the two variables 'requiredslot' and 'animaliound'. A sequence of animal-keeper pairs could be inserted in the table with:

```
200  DEF PROCsetupzootable
210  LOCAL noofanimals,i, nextanimal$,nextkeeper$
220    READ noofanimals
230    FOR i = 1 TO noofanimals
240      READ nextanimal$, nextkeeper$
250      PROCsearchfor(nextanimal$)
260      IF animalfound THEN
           PRINT nextanimal$; " is there already."
         ELSE animal$(requiredslot)=nextanimal$ :
           keeper$(requiredslot)=nextkeeper$
270    NEXT i
280  ENDPROC
```

To answer an inquiry about who looks after a particular animal, we could use:

```
400  DEF PROCfindkeeper(givenanimal$)
410    PROCsearchfor(givenanimal$)
420    IF animalfound THEN
         PRINT keeper$(requiredslot); " looks after ";
       ELSE PRINT "We don't keep ";
430    PRINT givenanimal$; "s."
440  ENDPROC
```

This is what the table would look like after inserting the sequence

coyote
fox
wolf
bear
warthog
antelope
ocelot
elephant
lion
zebra
yeti
buffalo
yak
cougar
bison

	animal$
0	antelope
1	bear
2	coyote
3	buffalo
4	elephant
5	fox
6	yak
7	cougar
8	bison
9	
10	
11	lion
12	
13	
14	ocelot
15	
16	
17	
18	
19	
20	
21	
22	wolf
23	warthog
24	yeti
25	zebra

The keeper information is omitted to aid clarity in the diagram.

Note the tendency for the names to bunch together in particular areas of the table. This can result in very long search lengths for some names. A simple approach like this would be even more likely to cause bunching in other application areas, for example in storing an interpreter variable table: programmers tend to invent names for their variables in some systematic way - x, y, z, a, b, c, root1, root2, root3, etc. We usually ccunbine two approaches to solving this problem.

Firstly, we should attempt to define a hash function that results in as wide a spread of values as possible over the subseripts of the table. The function need not be particularly meaningful. We could, for example, select an arbitrary subset of the bit pattern representation of a key and transform this in any way we like to give a hash value.

Secondly we can modify the simple linear search used above to find a location in the table. In the first example considered above, we calculate an initial value for probe (p say) and proceed to examine locations p, p+1, p+2,..., all values being taken MOD 10. We could, in general, examine locations p, p+d(1), p+d(2),..., where d is some

displacement function. Tho simplest such function is a linear one, d(i) = ki and in the above we used k = 1. A value of k > 1 can give us a better spread of the information in the table and shorten search lengths. With k = 7 we would examine locations p, p+7 , p+14, ..., (all MOD the size of the table).

Modifying the zoo-keeper example in this way produces:

Using k = 7 and inserting the same name$ as before

	animal$
coyote	0 antelope
fox	2 bear
wolf	3 warthog
bear	4 elephant
warthog	5 fox
antelope	6
ocelot	7
elephant	8 buffalo
lion	9 cougar
zebra	10
yeti	11 lion
buffalo	12 yak
yak	13
cougar	14 ocelot
bison	15 bison
	16
	17
	18
	19
	20
	21
	22 wolf
	23
	24 yeti
	25 zebra

Search lengths up to 9 were necessary in the previous table, but here the longest search length involves only 3 comparisons.

Note that if n is the size of the table, then k and n must ire coprime, otherwise not all the locations of the table will be available for any given hash value. Consider n=12 and k=4. For hash value 7, only locations 7, 11 and 3 are visited by the algorithm and only these locations are available for storing keys with hash value 7.

The theory tells us that for a table of size n containing m records, use of a linear displacement function results in an average search length of (1-r/2)/(1-r), where r= m/n. In order to obtain the best time/space trade-off, the table should not get more than 2/3 to 3/4 full.

Improved search times can be obtained by using non-linear

displacement functions, but we will not discuss this here.

Finally, note that in real-life implementations, greatest efficiency is achieved by using a table of size n, where a is a power of 2. Calculations MOD n can then be performed by ANDing with a computer word containing a sequence of bits all set to 1. The hash function can be calculated efficiently in a similar way. In data processing jargon, hashing is sometimes known as 'randomising' and is an example of an 'address generating' access method.

Exercises

1 Write a program that reads the stock list for the small department store described in an earlier exercise. The program should store the stock list in an open hash table using the stock numbers as keys. This table should then be used in repeatedly processing price enquiries.

2 Modify the French-English word translation program so that it uses an open hash table for storing the dictionary.

3 Use the procedures of Chapter 4, Section 4.1, to animate the process of building up an open hash table. In the strings manipulated in Chapter 4, the last three characters represented a stock number. A suitable hash function that would allow the table to fit on the screen could be calculated by:

```
VAL(RIGHT$(nextitem$,3)) MOD 20
```

This gives values in the range 0 to 19 and we would therefore use a table where locations were numbered 0 to 19. The table would be better displayed at the right of the screen, the value that is to enter the table being displayed with its hash value at the left of the screen. Use colour to highlight the locations of the table being examined during the search for an empty location.

6.8 Indexing and pointers

In this section we introduce the idea of storing and manipulating indirect references to the information being handled by a program. We describe two applications of this idea.

Indexed access to a table - introduction

Here we return to the idea of hierarchical access methods. Let us consider the problem of storing our animal records in a table which we shall divide into 26 subtables as before, one for the animals beginning with A, one for the animals beginning with B, and so on. As discussed above, this permits a two level access method, firstly to find the

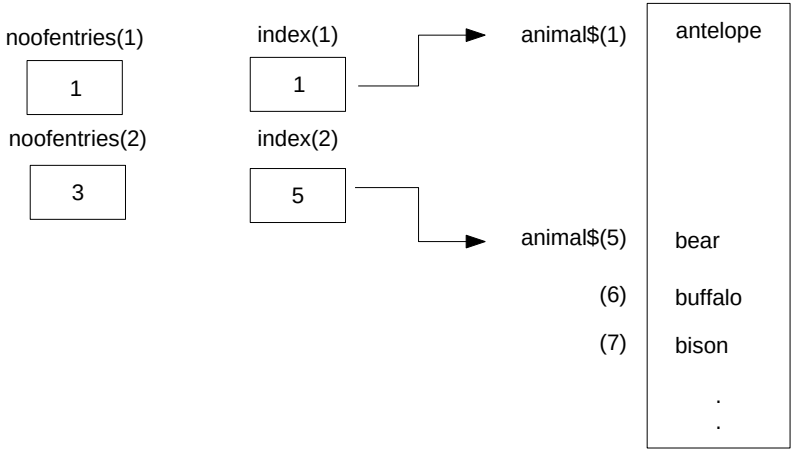
subtable in which tho required entry lies, and then to search that subtable. Rememember that this will in general require less effort than the march of a single large table. Here we shall make our division into subtables more flexible than it was before. For example, we might want to allow 4 locations for animals beginning with A, 7 locations for animals beginning with B, ... 2 locations for X, 4 for Y and 2 for Z. We allocate one large table in which the animal entries are to be stored:

```
DIM animal$(150), keeper(150)
```

We shall store information in an 'index table' which Indicates where in the, main table to find information about animals beginning with each letter.

```
DIM noofentries(25), index(25)
```

The intention is that, for example, 'noofentries(i)' and 'index(i)' should contain two integers, the first indicating how many animals beginning with the ith letter of the alphabet are stored, and the second indicating where in the main table the first animal beginning with the ith letter is stored. (Letters are numbered from 0 as before.) Entries for all animals beginning with a certain letter are to be stored in consecutive locations of the main table. For example:



We can think of the entries in the index array as 'pointers' to information in the main table. One way of indicating to the program how large to make the subtables for each letter would be to supply these 26 sizes in a DATA statement:

```
DATA 4, 7, ..., 2, 4, 2
```

The next precdure sets up the table from DATA statements. Lines 220-260 initialise an empty table and the remainder the procedure adds each animal to the appropriate subtable.

```
200  DEF PROCsetupzootab1e
210  LOCAL next,letter,size, i,entryno,location,
         nextanima1$,nextkeeper$
220    next = 1
230    FOR letter =0 TO 25
240      noofentries(letter)=0 : index(letter)=next
250      READ size : next = next + size
260    NEXT letter d
270    READ noofanimals J
280    FOR i=1 TO noofanimals
290      READ nextaninial$, nextkeeper$
300      letterno=ASC(LEFT$(nextanimal$,1))-ASC("a")
310      1ocation=index(1etterno)+noofentries(letterno)
320      noofentries(letterno) =noofentries(letterno)+1
330      animal$(1ocation) = nextanimal$+1
340      keeper$(location) = nextkeeper$
350    NEXT i
360  ENDPROC
```

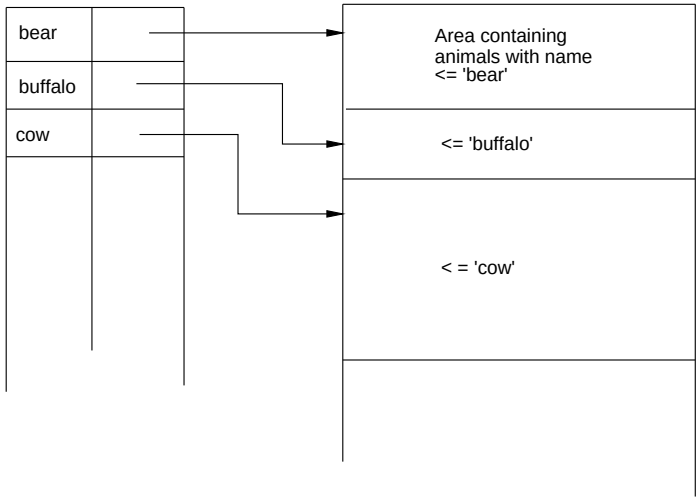
The index table is accessed directly using the first 1etter of an animal's name as a key and the subtables of the main table are accessed by linear search. To find a given animal name in the table we would use:

```
400  DEF PROC£indkeeper(givenanimal$)
410  LOCAL letterno, start,finish, found,there,probe
420    letterno = ASC(LEFT$(givenanimnal$,1)) - ASC("a")
430    start = index(letterno)
440    finish = start + noofentries(1etterno) - 1
450    found =FALSE : there =TRUE : probe = start
460    REPEAT
470    IF probe > finishTHEN there =FALSE
       ELSE IF animal$(probe) = givenanimal$ THEN
         found = TRUE
       ELSE probe = probe + 1
480  UNTIL found OR NOT(there)
490  IF found THEN PRINT "Keeper is "; keeper$(probe)
       ELSE PRINT "Given name not found."
500  ENDPROC
```

Indexed sequential access

In the above example, the keys used to access the table were the animal names, and the main table in which the keys were stored was to a certain extent ordered on these keys. Whenever the main table is ordered on the keys, access to

the main table can be easily accomplished via an index. In the above example, entries in the index table were found by direct access, but in genernl the distributton of the keys will not permit this. It would be silly to have index entries for animals beginning with A, B, C, ..., Z if we only have animals beginning with B and C, say. In such cases some of the keys theaselves could be stored in the index table. For example:



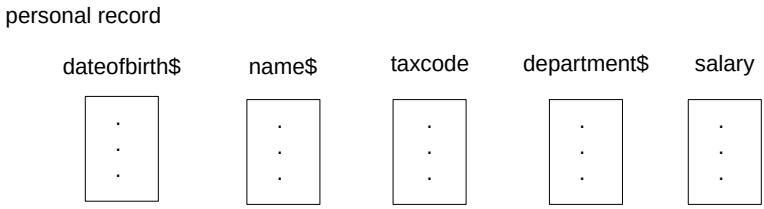
In these situations, entries in the index table could be found by linear search or binary search.

If the number of records involved is very large, resulting in a large index, an index to the index can be used, resulting in a three level access method.

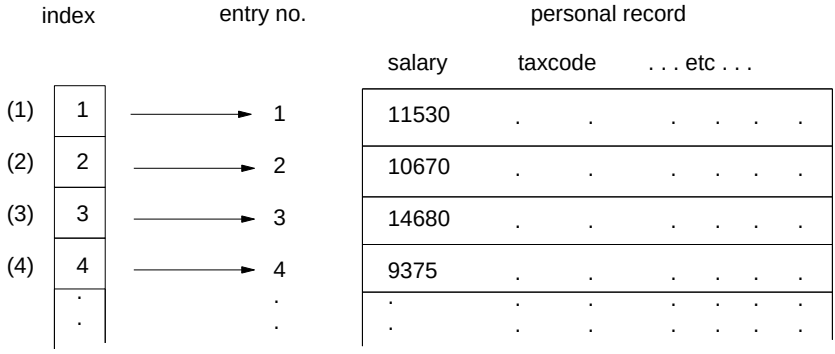
In data processing applications where the records are usually held on disk, sequential files with indexes constitute the most common form of addressing. This is because most data processing applications involve a mixture of random access to a file (e.g. find the entry for a given employee) and sequential access to the file (e.g. print out the names of all employees in alphabetical order). For random access the index is used and for sequential access the main file itself can be scanned in order. These techniques are therefore known as 'indexed sequential access' methods.

Sorting an index table

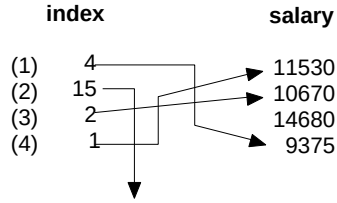
Another area in which use of an index table is advantageous is in sorting when each table entry is large, perhaps spanning a number of parallel arrays. For example:



It may be desirable to sort or otherwise re-order these entries from time to time. The amount of work involved can be considerably reduced by re-ordering the entries in an index table and leaving the entries in the main table undisturbed. Say we start with the following situation:



and we wanted to sort the entries into increasing order of salary. We can move the numbers or pointers in the index table until 'index(1)' contains the entry number for the entry with the lowest salary, 'index(2)' contains the entry number for the entry with the next lowest salary etc.



Here is a program that uses a simple exchange sort to sort the same stock list as was used in Section 6.3. In this case, we rearrange only the items in the index array to indicate the new ordering. You should refer back to the previous exchange sort and note the differences. Here, whenever we want to refer to the stock number for item i, we need to use:

stockno(index(i))

and to swap items i and j in the ordering, we need only move the contents of 'index(i)' and 'index(j)'.

```
 10  DIM stockno(100), price(100), index(100)
 20  PROCsetupstocktable
 30  PROCsortstocktable(noofitemsinstock)
 40  PROCoutputstocktab1e
 50  END

200  DEF PROCsetupstocktable
210  LOCAL i
220    READ noofitemsinstock
230    FOR i = 1 TO noofitemsinstock
240      READ stockno(i), price(i)
250      index(i) = i
260    NEXT i
270  ENDPROC

400  DEF PROCsortstocktable(n)
410  LOCAL i
420    FOR i = 1 TO n-1
430      PROCfindsmallestentryfrom(i)
440      PROCswop(i, posnsmallest)
450    NEXT i
460  ENDPROC

500  DEF PROCfindsmallestentryfrom(i)
510    LOCAL next
520    posnsmallest = i
530    FOR next = i+1 TO n
540      IF stockno(index(next))
           < stockno(index(posnsmallest))
         THEN posnsmallest = next
550    NEXT next
560  ENDPROC

600  DEF PROCswop(i, j)
610    LOCAL temp
620    temp = indext(i)
630    index(i) = index(j)
640    index(j) = temp
650  ENDPROC

700  DEF PROCoutputstocktable
710  LOCAL i
720    FOR i = 1 TO noofitemsinstock
730      PRINT stockno(index(i)), price(index(i))
740    NEXT i
750  ENDPROC
```

Exercises

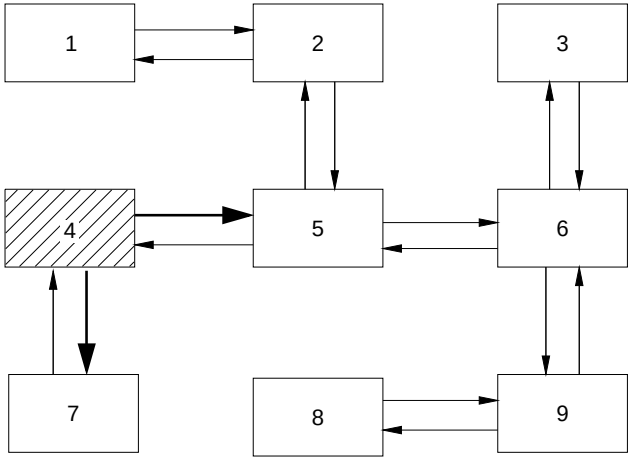
1 Set up two parallel arrays containing tho French-English dictionary used earlier. Now initialise two index tables each of which should contain pointers to all the locations in the dictionary. The entries in the first index table should be sorted so that they point to entries in alphabetical order of English words. The second table should then be sorted on the French words. Check that the program now works by using the index tables to print out the dictionary once with the English words in order and again with the French words in order.

2 Use the index tables created by the last exercise in a program that translates a given word from French into English or from English into French. The program should use binary search in the appropriate index table.

6.9 Adventure gamess - an example of the use of pointers

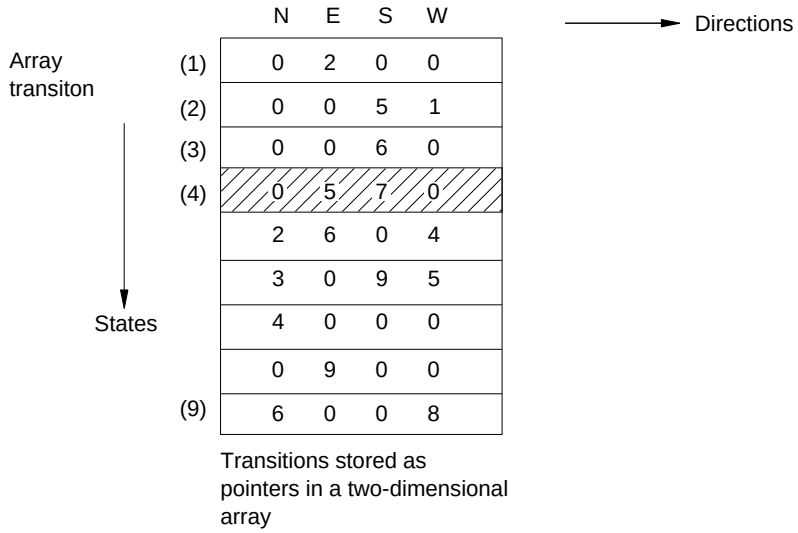
An interesting possible use of indexing techniques is an adventure game. These games were originally developed on a mainframe computer and despite the fact that they are dialogue games, using no graphics facilities, they have became very popular.

In an adventure game the player 'moves' from one geographical state to another. When in a particular state a transition to another state is only possible by moving in a predetermined direction. For example, the transitions for an extremely simple nine-state game are shown by arrows in the 'map':



'Geographical' states 1-9 and possible transitiona

Both the allowed directions and movement through the states can be controlled by a two dlmcmnional array of indices or pointers. We could represent thaw by using rows of four elements. Each item in a row represents a possible transition path in the directions N,E,S, or W, say.



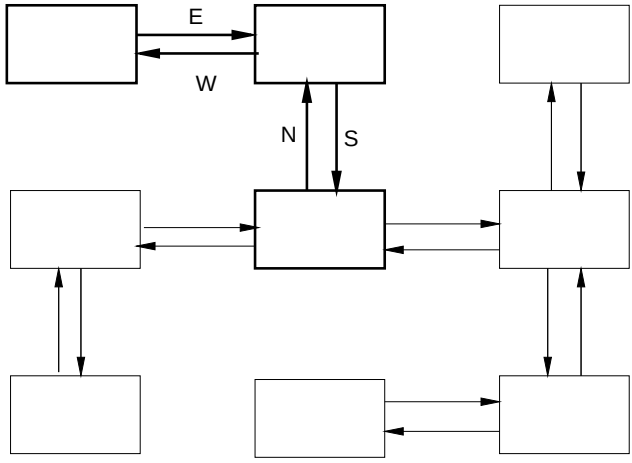
Transitions stored as pointers in a two-dimensional array

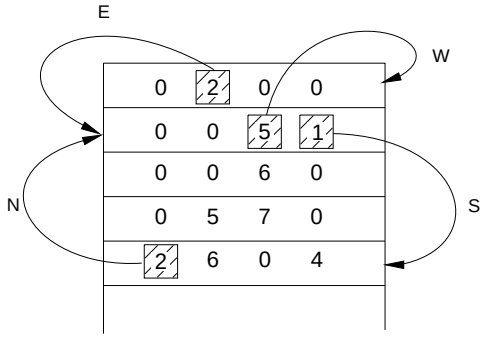
We could 'move' through these states by using the following structure. This structure or something equivalent forms the kernel of all adventure game programs.

```
10   PROCsetupdatastructures
20   position = 3
30   REPEAT
40     INPUT direction$
50     dir = INSTR("NESW", direction$)
60     IF transition(dir, position) =0 THEN
         PRINT "can't move"
       ELSE position = transition(dir, position)
70   UNTIL FALSE
```

The important statement is the one that follows the ELSE. If we started with 'position' equal to 1 and typed the sequence E,S,N, W, we would 'move' from state I to states 2 and 5 back to state 1.

The player of the game must explore and attempt to construct his own map for the game.





'Travelling through' the states in response to the sequence E, S, N, W.

Now another array is needed to store a description of each place so that the player/computer dialogue can be set in a particular environment, say:

```
description$(1)    in a shrubbery
            (2)    in a conservatory
            (3)    outside a glocunyhouse
            (4)    in a dark tunnel
            (5)    in a library
            (6)    in a dingy hallway
            (7)    in a smuggler's cave
            (8)    in a wizard' s lair
            (9)    in a dark passage
```

The kernel can then be enhanced to include indexing to this array.

```
60  IF transition(dir, position) = 0 THEN
      PRINT "can't move"
    ELSE position = transition(dir, position):
      PRINT "You are now "; description$(position)
```

Now the point of the game is to reach a particular destination or goal. The instructions telling the player which state he is in needs to be part of a procedure that checks to see if the goal state has been reached.

```
20  position = 3 : goalachieved ="FALSE
30  REPEAT
40    INPUT direction$
50    dir = INSTR("NESW", direction$)
60    IF transition(dir, position) =0 THEN
        PRINT "can 't move"
      ELSE position = transition(dir, position) :
        PROCcheckgoalreaehed
70  UNTIL goalachieved
```

So far so good, but it still isn't a very interesting game. The game is made more difficult by making a transition conditional on factors other than direction, such as possession of a particular object and avoiding various pitfalls. For example a lamp may be required to pass through a 'dark' state or a weapon may be required to deal with a state containing an enemy. Failure to cope with these pitfalls by non-possession of the requisite object or not taking the appropriate action with the object finishes the game.

```
30  REPEAT
      .
      .
      .
50  UNTIL goalachieved OR dead
60  IF goalachieved THEN PRINT "well done!"
    ELSE PRINT "hard luck"
```

The objects and the initial states that they appear in can be stored in two parallel arrays.

```
object$ (1)  lamp          objectpostn (1)  5
        (2)  emerald                   (2)  7
        (3)  sword                     (3)  6
        (4)  axe                       (4)  7
        (5)  ogre                      (5)  4
        (6)  serpent                   (6)  9
```

Additional commands might now be included for 'getting' or 'dropping' objects, post tion code 0 bel ng uaod to indicate that the player is carrying an objvet. The validity of a move might now be checked by calling a procedure such as:

```
100  DEF PROCcheckcanmove
110  IF transition(direction, dir) =0 THEN
       PRINT "can't move" : ENDPROC
120  IF position = 6 AND objectpostn(1) <>0 THEN
       PRINT "You have fallen into a bottcanless pit":
       dead = TRUE : ENDPROC
         .
         . more pitfall checks
         .
130    position = transition(dir, position)
140  ENDPROC
```

The implication of the first pitfall check is that in state 4 you must possess the lamp. Further elaborations can easily be added so that a sufficient level of detail is achieved to produce an intriguing game. For example, the lamp may be required to be lit.

The initialisation for the string arrays and the numeric arrays can originate from DATA statements or more usually froma file. Another important point is that a well-written pogram structure can be imposed on alternative data. Changing the string initialisations (and the numeric initialisation if required) produces a completely different game. The data would be a simple example of a database.

Exercises

1 Design a map for a simple adventure game and implement this as a transition table in a complete program which moves from one state to another in response to commands N, S, E, W. The main loop should repeatedly call a procedure PROCobeycommand:

```
goalachieved = FALSE : dead = FALSE
REPEAT
    INPUT command$
    PROCobeycommand
UNTIL goalachieved OR dead
```

At each step, the program should either print a message indicating that the move is not allowed or print the number of the new state reached. Do not attempt to add any frills until you are sure that the basic transition process is working and the program is moving around the map as expected. At first there will be no statements in the program for changing 'goalachieved' or 'dead' and you

will have to terminate the program with the ESCAPE key.

2 Now invent a verbal description for each state and set up an description array of strings. Use this to print a description of each state reached.

3 Make up a short list of objects, decide on their initial position and set up two parallel arrays containing a description and a position for each object. Implement a new command L (for look) which prints a list of objects that are at the current position.

4 Implement new commands G (for Get), D (for Drop) each of which is followed by an object name. The object name will have to be looked up in the object description array and the corresponding object position code changed appropriately. Implement a command I (for Inventory) which prints a list of objects currently held.

5 Decide on a goal for your game. For example the purpose could be to find one of the objects and transfer it to a particular place. PROCobeycommand can be extended to test whether the goal has been achieved and set the variable 'goalachieved' accordingly.

6 Add some pitfalls to your program and test for these in a move-making procedure setting 'dead' accordingly.