("A") is 65. Also suppose that a function called "CHR" takes an integer argument and evaluates to the ASCII character as defined in Fig. 2.4. For example, CHR(65) = "A".

Evaluate these expressions:

a. ORD ("1") + ORD ("3")
b. CHR(67) & CHR(65) & CHR(84)
c. CHR(ORD("7") + 1)

Questions without answers

7. What value does the binary code 01110101 represent in each of the following cases:
   a. A two-digit BCD numeric code.
   b. A pure binary integer.
   c. A single ASCII character.

   State any assumptions that you make.

8. Evaluate these expressions:
   a. (17 **div** 5) + (17 **mod** 5 (4 * (–2))
   b. **NOT** ((5 < 2) **OR** (16 = 24))
   c. (2 ↑ 7 = 128) **AND** (2 ↑ 9 – 512 = 0) **OR** (12/4 ≤ 5)

16: Program Structures

Introduction

16.1 This chapter introduces **program structures**, which are what we may call the forms in which program components are constructed, organised and interrelated.

16.2 In order to aid a clear understanding of the material covered in this chapter we start by providing the necessary background knowledge of programming languages and the forms in which programs are written.

16.3 When considering features of programs it pays to recognise how these features relate to the operational features of the computers for which the programs are written. Therefore the chapter is organised in a way that highlights these relationships.

16.4 Particular aspects covered in the chapter are:

a. Language, its syntax and semantics.
b. Data declarations.
c. Basic operations on data.
d. Control structures.
e. Subprograms.

Programming languages

16.5 The subject of programming languages is covered fully in a number of later chapters. So, as was mentioned in the introduction, only background information needed for this chapter will be discussed here.

16.6 When learning a programming language we need to learn about two important aspects of the language:

a. its **syntax** and
b. its **semantics**.

16.7 The **syntax** of a language is the grammatical rules that govern the ways in which words, symbols, expressions and statements may be formed and combined.

16.8 The **semantics** of a language are the rules that govern its **meaning**. In the case of computer language, meaning is defined in terms of what happens when the program is executed.

16.9 **Natural languages and formal languages.** Languages used for everyday communication, such as English, French and Chinese, are called **Natural Languages**.

16.10 **Natural Languages** have complexities that make them unsuitable as computer-programming languages. Instructions written in such languages often suffer from ambiguities. For example, this simple English question has several interpretations: "Calculate 4 and 6 divided by 2".

It could mean add 4 to 6 and then divide that answer by 2, or divide 6 by 2 and then add that result to 4, or it could mean divide 4 by 2 to get one result and then divide 6 by 2 to get another result, and so on. Another example is "I want ham or eggs and chips".

16.11 **Formal Languages** differ from natural languages in that they have precise semantics. Properly defined programming languages are formal. Each instruction written in such languages has just one meaning.

16.12 In practical terms what all this boils down to is that when we learn a programming language, as we learn each new instruction, we must learn both its syntax (how to write it down correctly) and its semantics (what will happen when it is executed) so that we use it sensibly.

16.13 When we considered computer architecture in chapter 13, we saw that hardware could be viewed at different levels. The same is true of software. Some computer languages are at higher levels than others.

16.14 At machine level (13.15) the operations performed on data are very simple. Just to add together two numbers may take three of these simple operations. For each of these operations there will be one corresponding program instruction in what is called **machine language**. In the early days of computers all programs were written in machine language.

16.15 These days, programs are seldom written in machine language. Instead, programs are most frequently written in what are called **high-level languages**.

16.16 The program instructions in high-level languages are much closer to sentences in English or expressions in mathematics. This makes them easier to use than machine languages.

16.17 The price paid for using these easier and more powerful languages is that each high-level language program must first be translated into machine language because the computer can only operate using instructions in machine language. This translation is normally carried out by the computer.

16.18 Further details of programming languages at various levels, and of the translation processes involved, will be given in later chapters. In this chapter the examples will be given in forms akin to high-level languages. This should have the advantage of preventing the details of the hardware from obscuring programming concepts. This language form is a kind of **pseudocode** (See the Part introduction).

Note. Program instructions written in high-level languages or pseudocode are normally called **statements**.

The program-level view of the computer

16.19 When writing programs we need some concept of what the computer is doing when the program is used. This *program-level view of the computer* is much simpler than the true picture but is ideal for its purpose. This situation is comparable to a car driver's view of the car's controls. The driver merely thinks that when the accelerator is pressed harder the engine goes faster, without giving any thought to what actually happens under the car bonnet.

16.20 The main features of the computer are still visible in features of the language.

a. Storage.
b. Input and output.
c. Operating on data.
d. Control.

We will consider each of these in turn.

Storage

16.21 In all but the lowest level programming languages data items are identified by name rather than by their location addresses in main storage. The names that we associate with stored data values are called **identifiers** because an identifier is the name by which the data value may be identified.

16.22 **Constants and Variables.** An identifier is a **constant** if it is always associated with the same data value and it is a **variable** if its associated data value is allowed to change. Changing a variable's value implies changing what is stored.

16.23 **Literals.** When referring to letters or names in programs we must be careful to specify whether we mean them to be taken *literally* or be treated as identifiers. When names or letters are used literally we call them **literals**, and we enable them to be distinguished from identifiers by placing them within quotation marks. So the instruction PRINT "N" means print the letter N, and the instruction PRINT N means print the value associated with N. Note that PRINT 2+4 means print the value 6, but PRINT "2+2" means print the character "2" followed by the character "+" followed by the character "2".

Data declarations

16.24 The data types of constants and variables must be defined within a program so that the appropriate operations may be performed upon the data values and so that the appropriate amounts of storage space can be assigned. Such information should be given at the start of each program.

16.25 Some programming languages have special statements that might be used for this purpose. They are called **declarative statements** because they state properties of the data and are not executed. They differ in that respect from most of the statements to be discussed in the remainder of this chapter which are executed, and are called **procedural statements**.

```
constants
              pi = 3.141592654

variables
              quantity_required,
              number_of_units,
              i,
              product                  : integer
              unit_price,
              product_price,
              side_one,
              side_two,
              area,
              rate,
              charge,
              units_used,
              unit_cost                : real
              rate_code                : character
              rate_code_valid          : boolean
```

Note. Variables of the same type are presented in a list separated by commas.

Fig. 16.1 An Example of a Data Declaration.

16.26 A typical data declaration is given in Fig. 16.1. Most of the data items declared in Fig. 16.1 will be used in later examples. Two more have been added to make the example complete; the constant "`pi`", used in mathematics, and a Boolean value "`rate-code-valid`".

16.27 Some programming languages declare data types in very different ways. For example, in BASIC the use of a "$" sign at the end of an identifier declares the variable to be a string. The absence of the "$" declares the variable type to be numeric by default. In BASIC variables need not be declared in declarative statements. In the language "C" the data type name appears in front of the variable name. So, for example in C the integer variable "i" might be declared as "`int i`".

16.28 In the declaration of the real constant pi in Fig 16.1 a value has explicitly been **assigned** to the identifier "`pi`". All the other identifiers in the example have no values associated with them at the end of the declaration.

16.29 A variable name does not have an associated value until it has been assigned one. This may be done either by using an assignment statement (details follow in 16.39) or by using them as arguments in input functions (details follow in 16.33). This process of assigning values to variables before using them is called **initialisation**.

16.30 Initialisation must be done in a systematic way. Otherwise, unexpected things may happen during the program's execution if non-initialised variables are used as operands. The program may give an error message and stop, or possibly worse still, carry on and give the wrong results or cause something dangerous to happen.

Input and output

16.31 Programming languages have special functions for dealing with input and output. Common names for these functions are **input, read, get accept, output, write, print, put** and **display.**

16.32 It is not appropriate to consider all possible cases at this stage. Rather, we will take one typical example to illustrate the basic ideas.

a. PART OF THE PROGRAM (in pseudocode)	b. OUTPUT DURING PROGRAM EXECUTION (what appears on the monitor screen.)

`output ("Key in the unit price in pounds")`
`input (unit_price)`
`output ("Key in the quantity required")`
`input (quantity_required)`
	`Key in the unit price in pounds`
	`1.25`
	`Key in the quantity required`
	`20`

Note.

i. The real number 1.25 and the integer 20 are keyed in by the user and are displayed on the monitor screen as they are keyed.

ii. Outputs of variables are also possible although not shown here.

Fig. 16.2. Programming Simple Input and Output.

16.33 Suppose the data is input at a keyboard, that output appears on a monitor screen and that two numbers are to be input. Let us suppose that these two numbers represent the unit price and quantity purchased of some product from which it is intended to calculate the price for all the products. The instructions in the program dealing with the input might appear in a form similar to that shown in Fig. 16.2a. In Fig. 16.2b we see what would appear on the monitor screen when the program was executed.

16.34 Try to rewrite these instructions in the language that you are using as part of your course.

16.35 When input, the numbers shown in Fig. 16.2b will be stored in main storage and referred to by the appropriate identifier (see Fig. 16.3).

Identifier	Data Type	Associated Stored Data Value
`Unit_price`	`real`	`1.25`
`quantity_required`	`integer`	`20`

Fig. 16.3. Data Item Identifiers and Associated Stored Values.

Operations on data

16.36 The operations that may be applied to data items of various types were discussed in some detail in the previous chapter. It remains for us to examine how these operations are incorporated into programs.

16.37 The ideas are best introduced by a simple example. We will continue with the example given in 16.33.

16.38 The price for all products is calculated by multiplying the unit price by the quantity required. A program instruction for doing this might take the form:

```
product_price := unit_price * quantity_required
```

This is an example of an **assignment statement.**

16.39 The **assignment statement** has three basic components:

a. An **assignment operator**, which is represented by some appropriate symbols (in the example ":=" are used).

b. A **result**, with an identifier ("product_price") to the left of the assignment operator;

c. an **expression**, to the right of the assignment operator.

16.40 Performing this instruction involves fetching the values associated with the operands in the expression from main storage, performing the required operations (multiplication in this case) and then storing the result in the location(s) in storage associated with the identifier "product_price". Thus, the outcome will be that the value of the result has been *assigned* to the identifier "product_price".

16.41 In some cases the same identifier may appear on *both* sides of an assignment statement. This often occurs when counting or accumulating totals.

16.42 **Examples.** Suppose we wished a program to count how many inputs had been made. To do this we could place this assignment statement at the start of a program to *initialise* the counter to zero:

```
number_of_inputs := 0
```

and place this assignment statement after each input instruction to count the inputs:

```
number_of_inputs := number_of_inputs + 1
```

NB. The result of evaluating the expression on the right-hand side of the assignment statement *replaces* the value associated with the identifier number of inputs *before* the expression is evaluated.

16.43 The exact form of the assignment statement varies from language to language. The examples just given take the form used in a number of languages including Pascal, Modula and ADA. Some readers may be using the language BASIC, which has an assignment statement in the form:

```
LET P = U * Q
```

In COBOL the assignment could take the form:

```
MULTIPLY U BY Q GIVING P
```

In C the assignment could take the form:

```
number_of_inputs := number_of_inputs + 1
```

or using a C shorthand for an **increment** ie, increasing a variable by 1, it could take the form:

```
number_of_inputs++
```

Note that decreasing a variable by 1 is called **decrementing**.

Control

16.44 The order in which program instructions are performed by the computer must be carefully controlled, and programming languages contain features that allow the order of instruction execution to be controlled.

16.45 All programming problems, no matter how complex, may be reduced to combinations of controlled sequences, selections or repetitions of basic operations on data. This fact was established by two Italian computer scientists, C. Bohm and G. Jacopini in 1966. It is part of the important theory that is behind the concept of **structured programming**, a term first used by Professor E. Dijkstra in the mid-1960s and now widely used and misused!

Control structures

16.46 Program control structures are used to regulate the order in which program statements are executed. They fall into three categories:

a. **sequences**
b. **selections**
c. **repetitions** (sometimes called **iterations**).

We will consider these three categories in turn.

Sequences

16.47 In the absence of selections or repetitions program statements are executed in the **sequence** in which they appear in the program:

```
statement_1
statement_2
statement_3
    .
    .
    .
```

16.48 In the following paragraphs we will use the term **statement sequence** to mean a sequence of one or more statements.

Selections

16.49 **Selections** form part of the decision-making facilities within programming languages. They allow alternative actions to be taken according to the conditions that exist at particular stages in program executions.

16.50 Conditions are normally in the form of expressions that when evaluated give Boolean results (**true** or **false**).

16.51 **The if_then_else** statement. Versions of this simple selection statement appear in most high-level languages. A typical form of syntax for this statement is given here:

```
if
        condition
then
        statement_sequence_1
else
        statement_sequence_2
endif
```

16.52 The semantics of this statement are as follows. When the statement is executed the condition's expression is evaluated. If the result of the evaluation is **true** then `statement_sequence_1` will be executed *otherwise* `statement_sequence_2` will be executed. After *either* `statement_sequence_1` or `statement_sequence_2` has been executed the statement following "`endif`" is executed (see Fig. 16.4).

16.53 **Example.** In this simple example we consider part of a program in which two real numbers representing the two sides of a rectangle are to be used to determine the area of the rectangle and to indicate whether or not the sides form a square (see Fig. 16.5).

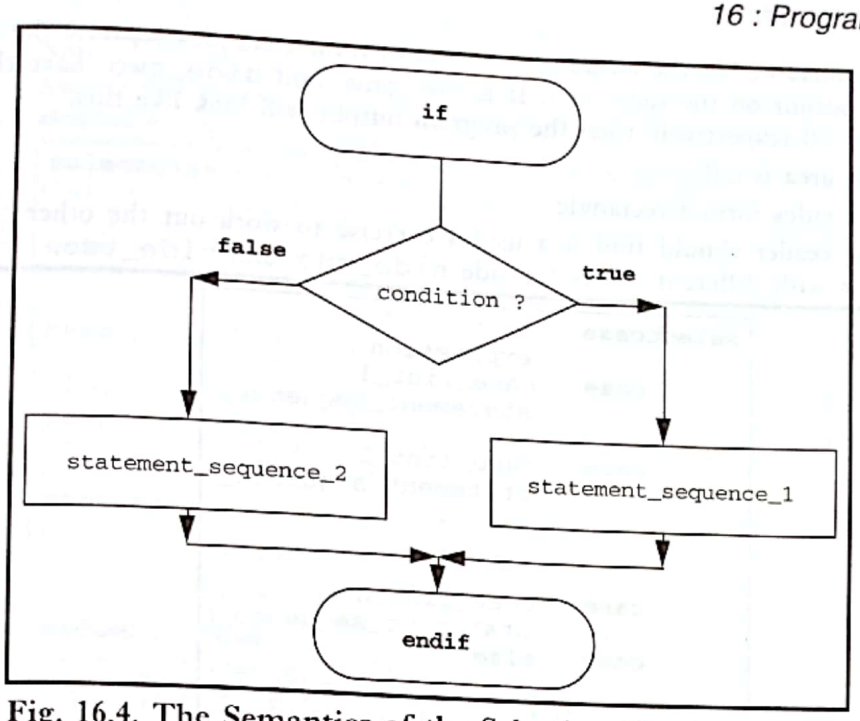


Fig. 16.4. The Semantics of the Selection **if_then_else**.

Note.

i. In this and further examples comments within programs will be enclosed between the symbols "(*" and "*)".

ii. In this example "side_1", "side_2" and "side_3" are all real numbers.

```
if
          (* The value of the sides are invalid *)
          (side_one <= 0)
          OR
          (side_two <= 0)
then
          output ("The sides do not form a rectangle.")
else
          area := side_one * side_two
          output ("The area is", area)
          if
            side_one = side_two
          then
            output ("The sides form a square.")
          else
            output ("The sides form a rectangle.")
          endif
endif
```

Fig. 16.5. Part of a Program Using Nested **if_then_elses**.

16.54 Note the following points about this example.

a. Within the first `if_then_else` statement there is another `if_then_else` statement. This is an example of "**nesting**", ie, a structure containing other structures of the same type.

b. The program is made easier to read by indenting statements progressively with each level of nesting.

c. The first condition is expressed informally as a comment before it is expressed in the syntax of the language. This makes the program easier to read.

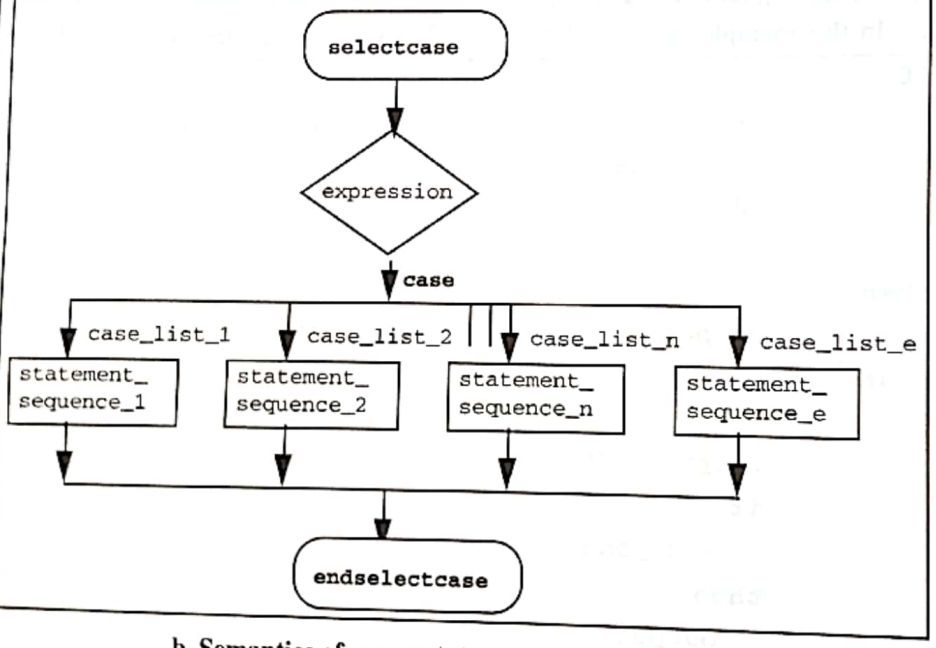
d. Commas within the output statements in these examples separate the data items to be output on the same line. If `side_one` and `side_two` have the values 2.0 and 3.0 respectively then the program output will look like this:

```
The area is 6.0
The sides form a rectangle.
```

The reader should find it a useful exercise to work out the other cases that can arise with different values for side `side_one` and `side_two`.

```
selectcase
        expression
    case  case_list_1
          statement_sequence_
    case  case_list_2
          statement_sequence_
          .
          .
          .
    case  case_list_n
          statement_sequence_
    case  else
          statement_sequence_
endselectcase
```

a. Syntax of a typical case statement.



b. Semantics of a case statement.

Fig. 16.6. A Case Statement.

16.55 A more general form of selection is the CASE statement. Again, a number of languages have their own versions of case statements. A simple case statement could have syntax like that shown in Fig. 16.6a. Fig. 16.6b explains the semantics.

16.56 **Example.** Suppose for the purpose of this example that the cost of a telephone call is to be calculated from the formula

```
charge := units_used * unit_cost * rate
```

where all four items are reals. Also assume that "`units_used`" and "`unit_cost`" have already been assigned values, but that the "`rate`" must be determined from the "`rate_code`", which is a single character. The following table summarises the relationship between "`rate_code`" and "`rate`".

The part of the program that determines the rate is given in Fig. 16.7.

Note.
Assume that the character "rate_code" has already been assigned a value.

```
selectcase
          rate_code
case      "L", "A"
          rate := 2.0
case      "B"
          rate := 2.0
case      "C"
          rate := 2.0
case else
          output ("ERROR - no such code!")
          rate := 0.0
endselectcase
```

Fig. 16.7. An Example of Using a Case Statement.

Repetitions

16.57 There are many programming problems in which the same sequence of statements needs to be performed again and again for a definite or indefinite number of times. The repeated performance of the same statements is often called looping.

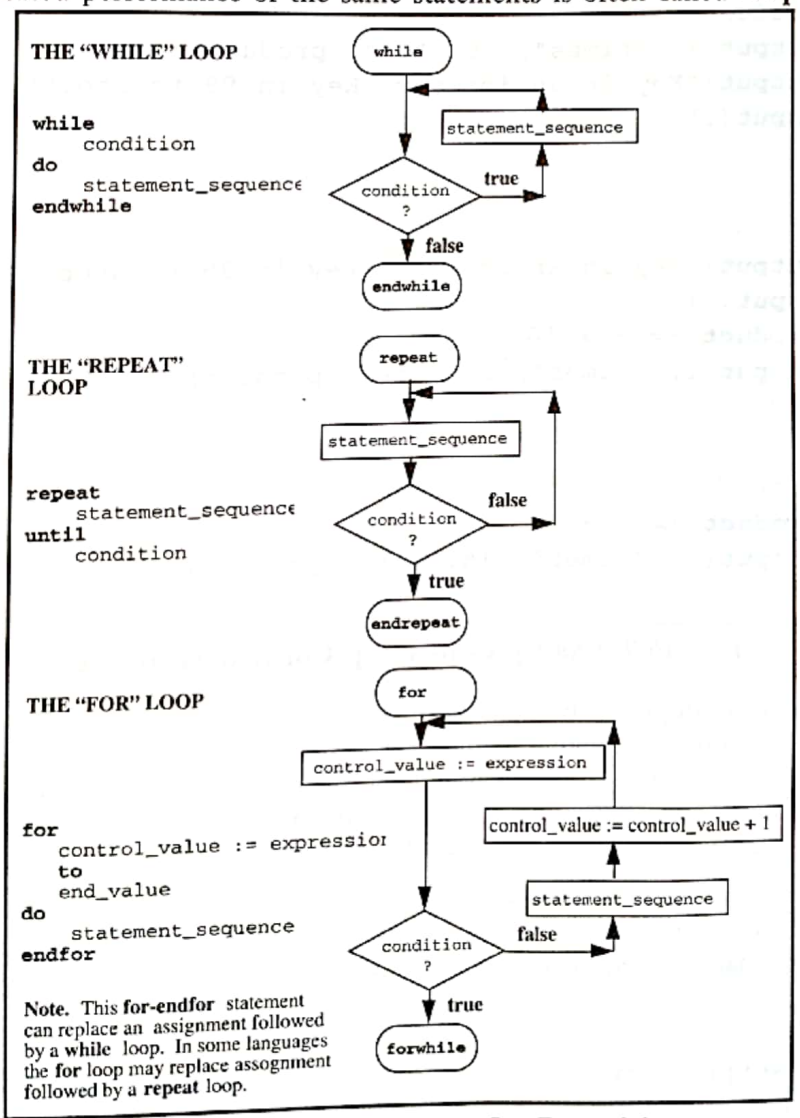


Fig. 16.8. Loop Constructs for Repetitions.

16.58 Fig. 16.8 shows three common **loop constructs** for performing repetitions. The "while" and "repeat" loops are used for **indefinite repetitions**. That is, they do not define the number of repetitions that will occur when the loop is executed. They merely give conditions for looping to stop. The "for" loop is used for **definite repetition**. It defines the number of repetitions that will occur during execution.

16.59 Examine Fig. 16.9, which gives examples of using the three loop constructs.

a. In case A, if the number 99 is keyed in first then the statement sequence within the whole loop will not be executed at all.

b. Case B uses a repeat loop. It looks simpler than case A, but even if the number 99 is keyed in first the whole statement sequence within the loop will be executed.

c. In case C the program will always output the same values. The product will be calculated and output, first for i = 1, then for i = 2 then for i = 3, then for i = 4 and finally for i = 5.

Note.

In these examples "i" and "product" are integers. Each example consists of part of a program which takes the value of "i", multiplies it by 16 and outputs the result each time the loop is repeated. The examples differ in other respects which help to highlight the differences between the loop constructs.

```
A.
output("Key in an integer. Key in 99 to stop")
Input(i)
while
          i<>99
do
          product := i * 16
          output(i, "times", 15, "is", product)
          output("Key in an integer. Key in 99 to stop")
          input(i)
endwhile
B.
repeat
          output("Key in an integer. Key in 99 to stop")
          input(i)
          product := i * 16
          output(i, "times", 15, "is", product)
until i = 99
C.
for i := 1 to 5
          product := i * 16
          output(i, "times", 15, "is", product)
endfor
```

Fig. 16.9. Examples of Loop Constructs in Use.

16.60 **Sentinels.** In example A in Fig. 16.9 it is clear that the number 99 is being used in a special way. It does not belong to the data, instead it signifies that the data has come to an end. The program must keep watch for such values, which are called **sentinels** or **rogue values**. The term sentinel is probably preferable, although less widely used, because there are situations in which sentinels occur naturally as part of the genuine data.

16.61 We have now considered all the language features listed in 16.20: Storage, Input and Output, Operations on data and Control. However, there is one aspect of program structuring that still requires attention and that is the subject of **subprograms**.

Subprograms

16.62 The term **subprogram** may be used to describe a component part of a program. Used loosely the term may merely refer to any set of statements forming part of a program

used to perform a specific task. However, a properly constructed subprogram should be self-contained, perform well-defined operations on well-defined data and have an internal structure that is independent of the program in which it is contained. When a subprogram has all these properties it is sometimes called a program **module**.

16.63 The two basic types of subprograms are:

a. functions

and

b. procedures.

Note. Here the term procedure is being used with a far more precise meaning than that used previously (see the Part introduction). Unfortunately, there is some considerable variation in the use of terms such as "function", "procedure", "module" and "subprogram". Within this text every effort has been made to use the most widely adopted terms and to use them consistently. Where general and specific meanings exist the context should make it clear which sense is intended.

Functions

16.64 Many high-level programming languages have inbuilt functions such as those mentioned in 15.65. A common example is the square root function "sqrt". The reader may remember that the square root of a number x is that number y with the property $y^2 = x$. Thus the square root of 9 is 3 because $3^2 = 9$. We will assume, for the sake of mathematical simplicity, that the square root function in the following example is only to be used for positive arguments.

16.65 Fig. 16.10 shows a simple program using an inbuilt square root function.

```
program Find_root
    (*
    This program has the name "Find_root".
    It inputs a number and outputs the
    value of the square root of the number.
    *)
variables
    number,
    root : real
Begin
    output("Key in the number whose square root you require.")
    input(number)
    if
        number < 0
    then
        output("No root is obtainable for negative numbers.")
    else
        root := sqrt(number)
        output("The square root of ",number " is ", root)
    endif
end
```

Fig. 16.10. A Simple Program Using an Inbuilt Function.

16.66 When a required function is not inbuilt it is sometimes possible for the programmer to define it for him or herself. Unfortunately, full facilities to do this are a less common feature of programming languages despite their usefulness.

16.67 Suppose we wish to have a function for calculating the area of a rectangle, as we did in Fig. 16.5. It may be argued that this operation is a little too simple to merit defining a special function but it serves as a useful example nevertheless.

16.68 The program that uses the function will contain a **function declaration**. This declaration *defines* the function. The function will also be used or **called** within the sequence of executable statements. Fig. 16.11 shows the relevant parts of the program.

16.69 The arguments used in the function declaration, ie, "first_side" and "second_side", are called **"formal arguments"**. The arguments used in the function call are called **"actual arguments"**.

Procedures

16.70 Any defined way of carrying out some actions may be called a **"procedure"**, but here we are using the term more precisely.

16.71 Programming procedures are defined operations on defined data and may be used as program components.

16.72 Procedure definitions are similar to function definitions, but procedure calls are statements whereas function calls appear in programs as expressions. These points should be clearer after the following example.

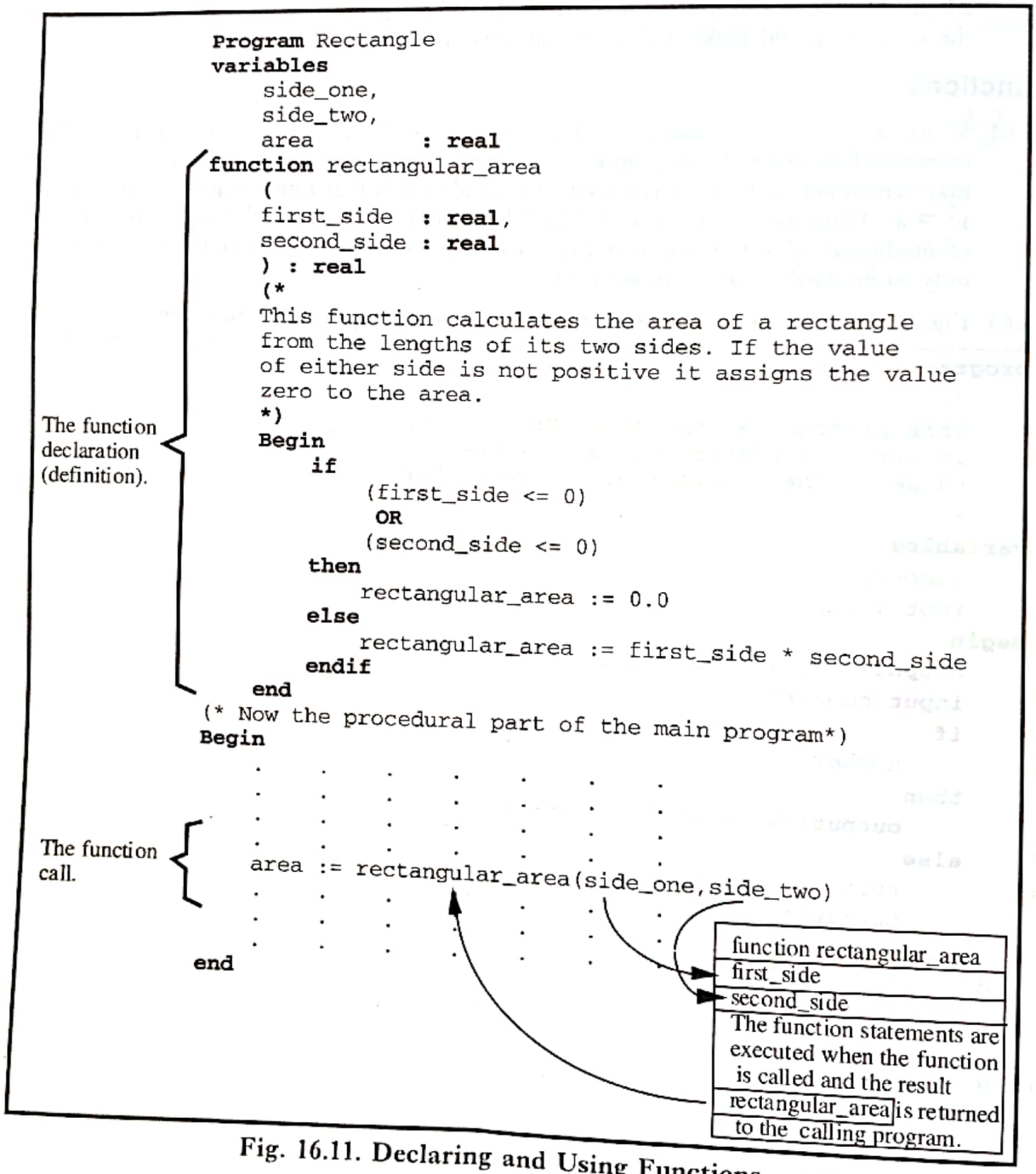


Fig. 16.11. Declaring and Using Functions.

16.73 A procedure call is a program statement. This point will be easier to understand if we compare a procedure call with a function call. The function call in Fig 16.11 had the form:

```
area := rectangular_area (side_one, side_two)
```

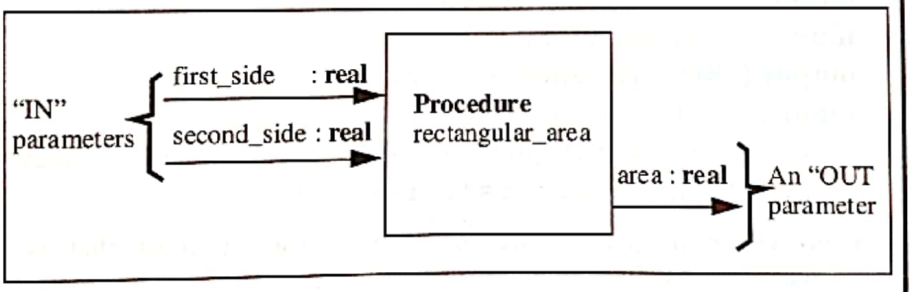
Had an equivalent procedure been used instead the procedure call could have taken the form:

```
call rectangular_area (side_one, side_two, area)
```

16.74 **Parameters.** In the procedure example just given "side_one", "side_two" and "area" are the **actual parameters** of the procedure. The **formal parameters** are "first_side" and "second_side". The reader will no doubt have noticed the similarity with the **actual arguments** and **formal arguments** of functions. However, there are some important differences. To see these, first consider Fig. 16.12.

```
procedure rectangular_area
    (
    IN    first_side  : real,
    IN    second_side : real,
    OUT   area        : real
    ) : real
    (*
    This procedure calculates the area of a rectangle
    from the lengths of its two sides. If the value
    of either side is not positive it assigns the value
    zero to the area.
    *)
    Begin
        if
            (first_side <= 0)
             OR
            (second_side <= 0)
        then
            area := 0.0
        else
            area := first_side * second_side
        endif
    end
```

a. A declaration of the procedure "rectangular_area".



b. A diagrammatic representation of the procedure "rectangular_area".

Fig. 16.12. Using a Procedure.

16.75 Note that the **IN** parameters of the procedure correspond to the arguments of a function. The **OUT** parameter of the procedure serves the same purpose as the value returned by a function.

16.76 There is a third type of parameter not present in this example, the **IN–OUT** parameter, which is used when data is to be passed into a procedure, manipulated by it in some defined way and then passed out again.

16.77 There are a number of further features of functions and procedures still to be discussed but they fit more usefully into the remaining chapters in this Part.

Summary

16.78 a. The idea of program structures was introduced.

b. The syntax, semantics and other basic properties of programming languages were introduced.

c. Data declarations were explained.
d. Identifiers, constants, variables and literals were defined.
e. Simple input and output functions were explained.
f. The assignment operation was discussed.
g. Control structures were introduced and explained including:
   i. **Statement sequences**
   ii. **Selections:** `if_then_else` and `case`
   iii. **Repetitions:** `While-loops`, `Repeat-loops` and `For-loops`.
h. Subprograms were introduced, including the basic features and uses of:
   i. **functions**
   ii. **procedures.**

Points to note

16.79 a. This chapter has placed an emphasis on the building blocks from which programs are constructed. The *method* for designing complete programs from these building blocks *will be explained in the following chapters*. Also, later chapters go into more details of how the various program components work. For example, the mechanisms for dealing with parameters are explained.

Student self-test questions

1. Explain the difference between syntax and semantics.
2. Distinguish between constants, variables and literals.
3. Using the definition of **input** and **output** given in 16.33 show what would be displayed on the monitor screen if the following statements were executed.

   NB. Assume
   i. the values input are 6 and 8.
   ii. all variables have been declared as integers.

```
output ("Key in an integer.")
input (first_number)
output ("Key in another integer.")
input (second_number)
result := (first_number + second_number) div 2
output ("The result is", result)
```

4. Write a procedure in pseudocode to calculate the discount that is allowed on purchases according to the following scheme:

   If the total value of purchases is over £100, 10% discount is given, between £20 and £100, 5% is given and below £20 no discount is given. Be careful to define all your variables.
5. Explain the differences in use between actual parameters and formal parameters.

Questions without answers

6. Write a program that uses the procedure of question 4 and that deals with several purchase totals in turn terminated by a negative purchase total.
7. Use a case statement to write the following procedure. The IN parameter is a single character. If the character is an octal digit, then the corresponding binary value is output as a literal string. For example, if the parameter is "3" the string "011" will be output. Any other character should result in "not octal" being output.

17: Data Structures

Introduction

17.1 This chapter explains what **data structures** are. It introduces the more important and common data structures and explains their uses in programming. The problems associated with sorting data and with trying to find data elements quickly and easily are also examined. The chapter has the benefit of bringing together the main topics relating to data structures which could be very useful when revising but is quite long as a result. The reader is therefore advised to read through the material in more than one session when reading the material for the first time.

17.2 A **data structure** is an organised grouping of data items treated as a unit. They may be regarded as complex data types (15.3). Data structures are sometimes called **data aggregates**.

17.3 The principal data structures considered in this chapter are:

a. Arrays.
b. Strings.
c. Records.
d. Lists.
e. Trees.
f. Queues.
g. Stacks.
h. Access tables.

17.4 The operations associated with these data structures will be explained and the most common programming methods related to these structures will be described with the aid of suitable examples.

17.5 There are so very many different types of data structure and so many related processing activities. Therefore, the discussion of some topics will be postponed until later chapters.

Arrays

17.6 In many situations, sets of data items such as the set of examination marks for a class, may be conveniently arranged into a sequence and referred to by a single identifier, eg:

MARK = (56 42 89 65 48)

Such an *arrangement* is a data structure called an **array**.

17.7 **Individual data items in the array** may be referred to separately by stating their position in the array. Thus MARK(1) refers to 56, MARK(2) refers to 42, MARK(3) refers to 89, and so on. The position numbers given in parentheses are called **subscripts**. Variables may be used as subscripts, eg, MARK(i), so when i = 2 we are referring to MARK(2), ie, 42, and when i = 4 we are referring to MARK(4), ie, 65.

Simple array handling

17.8 The basic methods used will be explained by means of examples. Suppose that individual exam marks are to be read into an array and, further, suppose that the largest mark, smallest mark, average mark, total mark and number of marks below the average are to be calculated and output.

17.9 The program to do this task will be easier to understand if taken in steps. First of all we can identify four main stages in the program:

a. Input all the data.
b. Calculate the largest mark, smallest mark, total mark and average mark.
c. Calculate the number of marks below the average.
d. Output the results.

```
Program Marks
        (*
        This program inputs a set of not more than 100 marks into an array,
        determines the lowest mark, higher mark, total mark, average mark and
        number of marks below the average. It then outputs each of the
        values it has determined.
        *)
variables
        number_of_marks,
        number_below_average : integer
        total_mark,
        smallest_mark,
        largest_mark,
        average_mark : real
        (*
        The next declaration indicates that "mark" is an array
        able to hold up to 100 real numbers.
        *)
        mark : array [1..100] of real .
Begin
        (*
        Determine how may marks are to be input and input
        them into the array.
        *)
        call input_marks (number_of_marks, mark)
        (* Calculate the required values. *)
        call calculate_results
                        (
                        number_of_marks,
                        mark,
                        total_mark,
                        smallest_mark,
                        largest_mark,
                        average_mark
        )
        number_below_average := below_average number_of_marks, mark, average_mark)
        (* Output the results *)
        output("Number of marks entered",number_of_marks)
        output("Smallest mark:",smallest_mark,"Largest mark:",largest_mark)
        output("Total mark:",total_mark,"Average mark:",average_mark)
        output("There were",number_below_average,"marks below average.")
end
```

Fig. 17.1. Outline Program for an Array-Handling Problem.

17.10 These stages suggest four corresponding stages in the program. An outline program is shown in Fig. 17.1. Note that the first two stages of the program are in the form of procedures. The two accompanying procedure declarations have been omitted from Fig. 17.1 because they are to be discussed separately in the following paragraphs. The third stage is a function that is set as a question at the end of this chapter.

17.11 First we shall consider the input of the data into the array. Examine Fig. 17.2, which shows the details of the procedure declaration.

17.12 Note the following points about the procedure in Fig. 17.2:

a. The formal parameters "`number_of_marks`" and "`marks`" are both OUT parameters. In this example they have the same identifiers as the actual parameters used, but this makes no difference to their meaning.

b. The variable "`subscript`" is a **local variable**, ie, it is used within the procedure and is only defined there. In contrast, variables that are defined throughout the program are called **global** variables. In general terms it is bad practice to *use* variables globally. They should always be passed into procedures as parameters if they are to be used there. This matter will be discussed more fully in a later chapter.

c. If the user of the program repeatedly fails to key in a valid value for "number_of_marks" the procedure repeatedly outputs an error message and asks the user to try again. A more sophisticated procedure might be designed to abort the procedure after some specific number of errors.

d. In the for_loop in the last part of the procedure the subscript is increased by one each time the loop is repeated. So, for example, if number_of_marks = 3 the loop is equivalent to:

```
output("key in a mark")
input(mark(1))
output("key in a mark")
input(mark(2))
output ("key in a mark")
input (mark(3))
```

17.13 Now for the second procedure, which is called "calculate_results" (Fig. 17.3). Note that this procedure relies upon the fact that there will be at least one mark value in the "mark" array. In this case it is a reasonable assumption because of the way "input_marks" is written. In general, however, it is a good idea to incorporate checks on IN parameters of procedures if their validity cannot be guaranteed.

```
procedure      input_marks
                        (
                        OUT number_of_marks : integer
                              mark : array [1..100] of real
                        )
               (*
               This procedure gets the user to key in the number of marks to
               be entered, which must be an integer between 1 and 100 inclusive.
               Then it gets the user to key in the numbers which it stores in
               the array.
               *)
variables
          subscript : integer
Begin
          (* Get the user to key in the number of marks. *)
          output ("Key in the number of marks to be entered.")
          output ("At least one mark and not more than 100 marks")
          output ("may be entered.")
          input (number_of_marks)
          while
             (* invalid number entered *)
             (number_of_marks < 1)
             OR
             (number_of_marks > 100)
          do
             output("ERROR - number must be between 1 and 100")
             output("Please key in the correct value.")
             input(number_of_marks)
          endwhile
          (* Get the user to key numbers into the array *)
          for subscript: = 1 to number_of_marks
             output("key in a mark")
             input(mark(subscript))
          endfor
end
```

Fig. 17.2. A Procedure for the Input of Numbers of an Array.

Other features of arrays

17.14 **Array elements.** The individual data items in an array are often called its elements. Strictly speaking, however, a data item *occupies* an element, ie, elements, rather like pigeonholes, are regarded as locations into which data items may be placed and removed.

17.15 **Main storage as an array.** The reader may have observed a close similarity between arrays and the operational characteristics of main storage. Indeed we may regard main storage as an array in which:

a. Locations correspond to elements, and

b. Location addresses correspond to subscripts.

One particular use of this fact is that skills learned in handling arrays are transferable to the handling of storage.

```
procedure      calculate_results
                    (
                    IN      number_of_marks : integer
                            mark      :array [1..100] of real
                    OUT     total_so_far,
                            smallest_so_far,
                            largest_so_far,
                            average_mark : real
                    )
               (*
               This procedure finds the smallest mark, the largest
               mark, the total of all the marks and the average of all
               the marks.
               *)
variables
               subscript : integer
Begin
               (* Initialise the total, smallest and largest values *)
               total_so_far := 0
               smallest_so_far := mark(1)
               largest_so_far := mark(1)
               (*
                    Take each mark in turn adding it to the total and
                    checking it for any change to the smallest or
                    largest value met so far.
               *)
               for  subscript := 1 to number_of_marks
                    total_so_far := total_so_far + mark(subscript)
               if
                    mark(subscript) > largest_so_far
               then
                    largest_so_far := mark(subscript)
               else
                    if
                         mark(subscript) < smallest_so_far
                    then
                         smallest_so_far := mark(subscript)
                    endif
               endif
          endfor
          average_mark : = total_so_far/number_of_marks
          (*
               There must be at least one mark so division by zero
               cannot occur.
          *)
end
```

Fig. 17.3. A Procedure for Calculating Results.

17.16 **Matrices** are arrays containing only numbers and no alphabetic data. (One **matrix**, many **matrices**.)

17.17 **Two-dimensional arrays** have elements arranged into rows and columns and are used for a variety of data tables. For example, the examination marks of a class for several subjects could be placed in a two-dimensional array thus:

Pupil or Row Number	English Column 1	Maths Column 2
1	A(1,1) = 56	A(1,2) = 44
2	A(2,1) = 42	A(2,2) = 36
3	A(3,1) = 89	A(3,2) = 73
4	A(4,1) = 65	A(4,2) = 86
5	A(5,1) = 48	A(5,2) = 51

This table can be represented in an array called $A = \begin{bmatrix} 56 & 44 \\ 42 & 36 \\ 89 & 73 \\ 65 & 86 \\ 48 & 51 \end{bmatrix}$

Individual elements can be specified by two subscripts used like map references. Rows 1st, Columns 2nd. (To remember this order think of Rhubarb and Custard.) eg, A(3,1) = 88; A(4,2) =86. Again subscripts may be variables, as for a one-dimensional array, so for A(r,c), if r = 3 and c = 2 we are referring to A(3,2) = 73.

17.18 **Array handling.** In this example the data is entered row by row. The pseudocode shown in Fig. 17.4 shows part of a procedure that reads data into the array. Assume that in the first part of the procedure valid values for the number of rows and columns in the array have been input.

```
Note. "number_of_rows" and "number_of_columns" are
      integer numbers to which valid values have been assigned
      "row_position" and "column_position" are local integers.
      "A" is an array of real numbers.

for   row_position : = 1 to number_of_rows
      for column_position : = 1 to number_of_columns
         output("Key in value for row, ",row_position,
         "column",column_position)
         input(A(row_position, column_position))
      endfor
endfor
```

Fig. 17.4. Part of a Procedure to Read Data into a 2D Array.

Sorting an array

17.19 There are many situations in which the data items within elements of an array need to be rearranged into an ascending or descending sequence. The order of the sequence is often based upon the numerical or alphabetic order of the data. For example, the data in the mark array of 17.6 could be sorted (ie, re-arranged into order) from

MARK = (56 42 89 65 48)

into *ascending sequence* to become

MARK = (42 48 56 65 89).

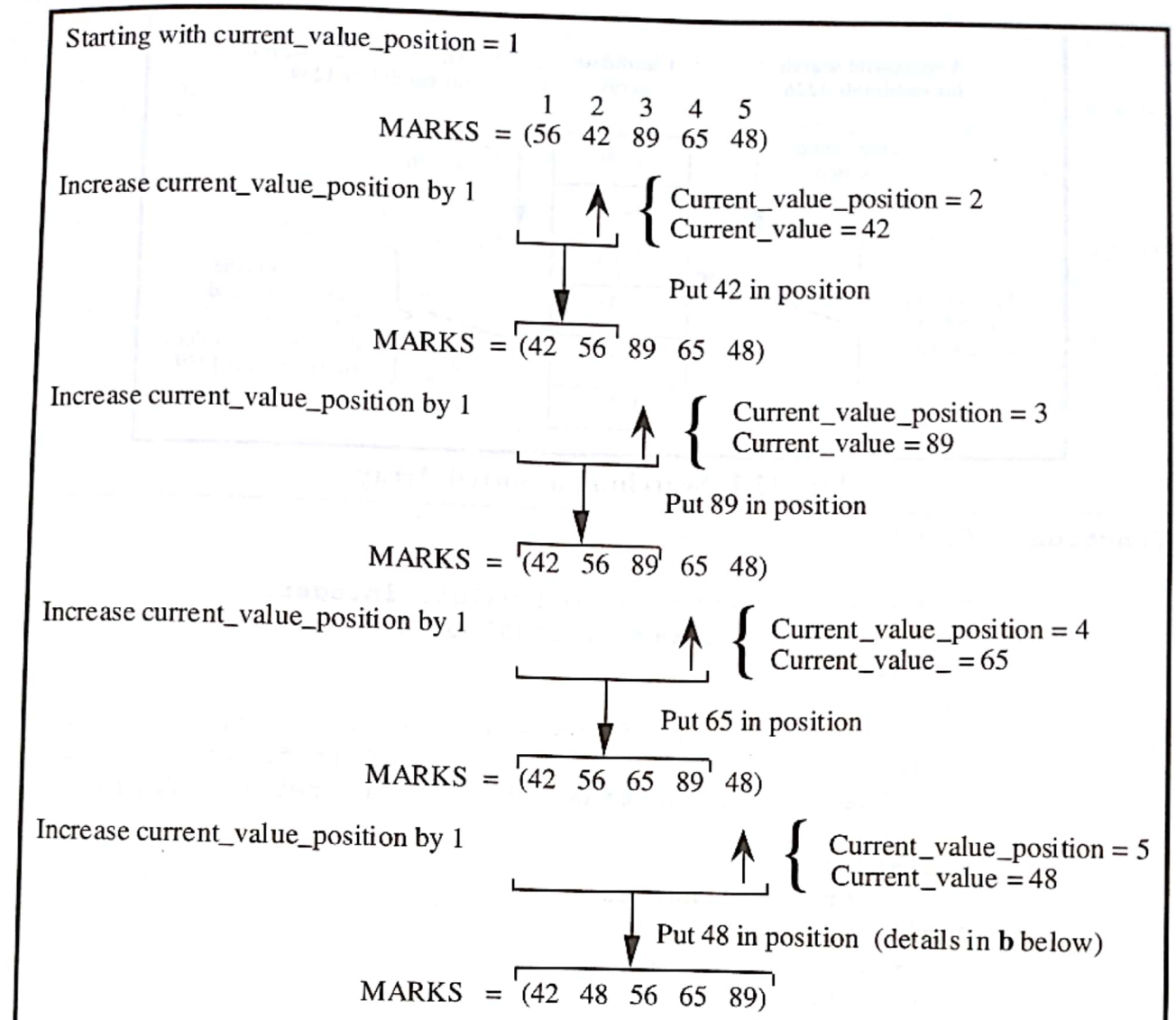
```
procedure sort_marks(IN number_of_marks : integer
                     IN-OUT mark : array [1..100] of real
          (*
          This procedure takes in the array "marks", sorts it into
          ascending numerical order using a straight insertion sort
          and returns it.
          *)
variables
          current_value      : real
          current_value_position,
          pointer_position   : integer
Begin
          (* Start at element 1 *)
          current_value_position := 1
          while
            (* some elements have not yet been reached *)
            current_value_position < number_of_marks
          do
            current_value_position := current_value_position + 1
            (* get the current value *)
            current_value := mark(current_value_position)
            (*
             put it in its correct position relative to the present
             and lower elements
            *)
              (* set a pointer to the current value position *)
              pointer_position := current_value_position
              while
                (* pointer not at insert position *)
                (pointer_position > 1 )
              AND
                (current_value < mark(pointer_position - 1))
            do
              (* copy adjacent value up to pointer position *)
              mark(pointer_position):= mark(pointer_position - 1)
              (* move pointer position down one *)
              pointer_position := pointer_position - 1
            endwhile
            (* insert value *)
            mark(pointer_position) := current_value
          endwhile
end
```

Note. The expression at the start of the inner while loop is assumed to evaluate to **false** when pointer_position = 1, ie, the left-most operand will be evaluated first so that mark(0) is not evaluated.

Fig. 17.5. A Straight Insertion Sort Procedure.

17.20 There are many methods of sorting data. The following example is based on a simple method called "straight insertion sort". This method is effective when sorting small arrays, but is rather slow when sorting larger arrays. A superior but more complicated method will be described later.

17.21 A procedure for the straight insertion sort is given in Fig. 17.5. Fig. 17.5 should be read in conjunction with the example in Fig. 17.6.



a. Outline example of the straight insertion sort.

```
MARKS = (42  56  65  89  48)
                      ^     pointer_position = 5
 (Pointer > 1) AND (48 < 89)
Copy 89 up. Move pointer down.
MARKS = (42  56  65  89  89)
                  ^     pointer–position = 4
 (Pointer > 1) AND (48 < 65)
Copy 65 up. Move pointer down.
MARKS = (42  56  65  65  89)
              ^     pointer–position = 3
 (Pointer > 1) AND (48 < 56)
Copy 56 up. Move pointer down.
MARKS = (42  56  56  65  89)
          ^     pointer–position = 2
 (Pointer > 1) BUT (48 > 42)
Place 48 into element pointed to
MARKS = (42  48  56  65  89)
```

b. More detailed example of an insertion.

Fig. 17.6. Examples to Illustrate the Procedure of 17.5.

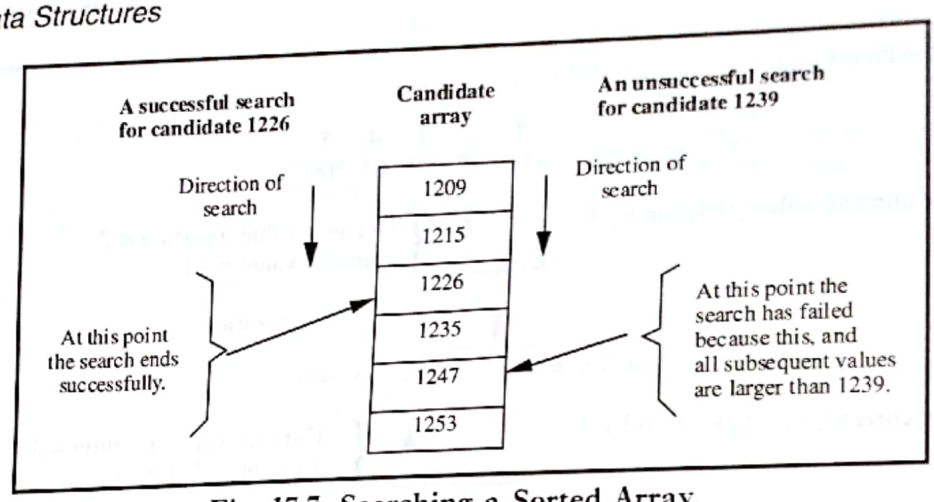


Fig. 17.7. Searching a Sorted Array.

```
function   found
           (
           number_of_elements, search_value: integer,
           search_array : array [1..100] of integer
           ) : boolean
           (*
              This function performs a linear search of the search
              array to find the search value. If it finds the search
              value it returns true otherwise it returns false .
           *)
           variables
              subscript : integer
           Begin
              (* Initialise values*)
              subscript := 0
              found := false
              (* search elements in turn *)
              repeat
              subscript := subscript + 1
              if
                 search_array(subscript) = search_value
              then
                 found := true
              endif
           until
              found
              OR
              (subscript = number_of_elements)
              OR
              (search_array(subscript) > search_value)
           end
```

Fig 17.8. A Function for Performing a Linear Search of an Array.

Searching an array

17.22 If we wish to know whether or not a particular value is present within an array we may find out by conducting a **search**. Searching is a useful and common processing activity and there are many methods that can be used.

17.23 A simple method, known as the **linear search**, is described in the next few paragraphs. There are more advanced methods of searching but they will be discussed in later chapters.

17.24 If the data in the array has already been sorted into order then the search can usually be performed faster than if the data has been left unsorted. This highlights another

advantage of sorting data. That is, in addition to helping in the presentation of the data, sorting may also make processing faster and possibly simpler too.

17.25 Suppose that we require a function that searches through an array of examination candidate numbers to see if a particular candidate's number is in the array. The function will return the Boolean value **true** if the candidate's number is found, otherwise it will return the value **false**.

17.26 The **linear search** is conducted by examining each array element in turn, from first to last, until the required value is found. Clearly, if the value has not been found by the time the last element has been examined the search has failed. If the array has been sorted into order the failure may be detected even sooner. Figures. 17.7 and 17.8 give the details.

```
Begin
A := "THE CAT SAT ON
     THE MAT".

A(6:7) := "OW"

A(20:20) := "H"

A(9:14) := "ATE UP"
end
```

The appearance of the string after each step.

Character position number	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
	T	H	E		C	A	T		S	A	T		O	N		T	H	E		M	A	T	.
	T	H	E		C	O	W		S	A	T		O	N		T	H	E		M	A	T	.
	T	H	E		C	O	W		S	A	T		O	N		T	H	E		H	A	T	.
	T	H	E		C	O	W		A	T	E		U	P		T	H	E		M	A	T	.

Fig. 17.9. Basic String-Handling Operations.

Strings

17.27 We now turn our attention to data structures that are used for text handling and related problems. As you may remember from 15.16, a sequence of characters handled as a single unit of data is called a **string**. For example, "ABC-, 3gh" and "THE CAT SAT ON THE MAT" are **literal strings**.

17.28 In other languages in which strings are used they are present in the form of one-dimensional arrays of characters. In the programming language BASIC all string identifiers have the "$" sign as a suffix. In paragraphs 17.29 to 17.35 assume that the variables A, B, C, V and F have been declared to be strings by a declaration such as

A, B, C, V, F: **string**

17.29 a. When strings are joined together they are said to be **concatenated** (15.39). So if A = "ALEX" and B= "JAMES" the assignment

C : = A & B

concatenates A and B to form C where

C = "ALEXJAMES"

b. **Part of a string** is called a **substring**, eg, A = "ALEX" and B = "JAMES" are substrings of C = "ALEXJAMES".

17.30 In the following paragraphs we are primarily concerned with strings as *data structures* composed of substrings that can be operated upon individually, whereas in chapter 15 we merely considered strings as simple indivisible data items.

String handling

17.31 **String handling notation.**

a. We number each character position in the string in sequence just like the elements in a one-dimensional array.

b. We refer to each substring by stating its *first* and *last* character position separated by a ":". The following pseudocode and table (Fig. 17.9) illustrate the method.

a.

```
(*
This is part of a procedure to output substrings of length 5 from
a string "F". The IN parameter "number_of_substrings" is of type
integer, as is the local variable "substring_number".
*)
Begin
for substring_number = 1 to number_of_substrings
 output(F(5 * substring_number - 4 : 5 * substring_number))
 endfor
end
```

b. Here are the successive values which occur during execution when F is as in 17.33a and "number_of_substrings" has the value 5.

substring_number output	5 * substring_number - 4	5 * substring_number	valu
1	1	5	JAMES
2	6	10	KATY
3	11	15	ANN
4	16	20	ALEX
5	21	25	JO

Note. Fixed substring length allows the position of each substring to be calculated.

Fig 17.10. The output of a Fixed-Length Substring.

```
(*
   These few lines of pseudocode output the first substring of V.
   The local variable "position" is of type integer.
*)
Begin
   position : = 1
   while
      V(position : position) <> "*"
   do
      output(V(position : position))
      position := position + 1
   endwhile
end
```

Note i. Further strings may be output in a similar fashion.

ii. An alternative storage strategy would be to have an integer at the start of each substring which indicated its length. eg.V = "05JAMES04KATY03ANN04ALEX02JO"

Fig. 17.11. Output from a Variable-Length String.

17.32 **Fixed-length and variable-length strings.** We return here to the concept of fixed-length and variable-length strings, which was introduced in 6.10 and 6.15.

a. **Fixed-length strings** have a fixed number of character places available for data storage.

b. **Variable-length strings** provide the data with just the number of spaces it needs.

The significance of these differences is illustrated by the following examples.

Note. These examples are important for two reasons:

i. The manipulation of fixed-length and variable-length data is often needed in programming problems.

ii. They illustrate the ways in which fixed-length and variable-length data is manipulated in main storage. The character positions in strings F and V in

these examples may be compared with location addresses in main storage, where each location is able to store one character.

17.33 **Example 1.**

a. This diagram shows five fixed-length strings concatenated into a single string called F. Each string is 5 characters long.
Details of F.

Character Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Content	J	A	M	E	S	K	A	T	Y		A	N	N			A	L	E	X		J	O			
Comment	1st String					2nd String					3rd String					4th String					5th String				

b. This diagram shows five variable-length strings concatenated into a single string called V. The same data is used but the end of each string is indicated by a "*". Note the saving in storage space.
Details of V.

Character Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Contents	J	A	M	E	S	*	K	A	T	Y	*	A	N	N	*	A	L	E	X	*	J	O	*		
Comment	1st String						2nd String					3rd String				4th String					5th String			Saved Storage	

17.34 **Handling fixed-length strings.** This example shows how the separate substrings in F can be printed in turn (see Fig. 17.10).

17.35 **Handling variable-length strings** is more difficult than handling fixed-length strings but does have the important advantage of saving space. This example shows how the first substring in V can be printed and the example in 17.36 will show how to handle all of V. Here we assume we have no prior knowledge of the string's length and so we take out each character in turn until we meet the terminator "*".

17.36 A simple example of an **access table** is an array that holds details of the locations of data values. We will consider such a case now and leave more complicated examples for later discussion. An access table called A is shown in Fig 17.12. It provides a means of accessing the strings held in V of paragraph 17.33b.

Row number (corresponds to substring i)	Position of 1st character in string i	Position of last character in string i
1	A(1,1) = 1	A(1,2) = 5
2	A(2,1) = 7	A(2,2) = 10
3	A(3,1) = 12	A(3,2) = 14
4	A(4,1) = 16	A(4,2) = 19
5	A(5,1) = 21	A(5,2) = 22

Fig. 17.12. Details of Access Table A.

```
(*
  This is part of a procedure to output the substrings of V by
  using the access table "A".
  The local variable "i" is of type integer (see Fig 17.12.)
  Other variables are as defined previously.
*)
begin
  for i := 1 to number_of_substrings
    output(V(A(i,1) : A(i,2)))
  endfor
end
```

Note. For Fig. 17.12 number_of_substrings has value 5.

Fig 17.13. Part of a Procedure Using an Access Table.

Records

17.37 Records will be easy to understand if we compare them with arrays. Records are like one-dimensional arrays in that they are comprised of a series of related data items. Records differ from arrays in that, whereas all the elements in an array have the same type, the successive data items in a record may differ in type. The following simple example should clarify these ideas.

17.38 **Example.** Suppose we wish to organise the following three items of data relating to an individual examination candidate into a single data structure:

```
candidate_number  : integer
candidate_name    : string
average_mark      : real
```

17.39 Although the three data items belong together they cannot be stored in an array because they are of different types. The answer is to store them in a record. A suitable record declaration could take the form:

```
candidate := record
               candidate_number : integer
               candidate_name   : string
               candidate_mark   : real
             endrecord
```

17.40 In the example just given a record called "candidate" was declared. The three elements of the record each had their own identifiers, unlike array elements, which are merely referred to by subscripts. The elements of a record are called **fields**.

17.41 Reference to the individual fields of a record may be made as shown in the following example.

17.42 **Example.** Suppose we wish to create a record for candidate 1209, named "Smith", who gained an average mark of 55.2. The assignments of these values to the three fields of the record called candidate may be performed in two ways:

a. Using the "dot" notation. Consider these lines of pseudocode:

```
Begin
        candidate.candidate_number    := 1209
        candidate.candidate_name      := "Smith"
        candidate.average_mark        := 55.2
end
```

Each record field is identified by the record name followed by a "dot" followed by the field name. Clearly this notation is rather clumsy when referencing several fields from the same record. In such cases the following method is preferable.

b. The "with" notation. The following pseudocode is equivalent to that given in "a".

```
Begin
  with      candidate
  do
            candidate_number  := 1209
            candidate_name    := "Smith"
            average_mark      := 55.2
  endwith
end
```

17.43 Not all high-level languages provide facilities for creating records. Pascal, C, COBOL, Modula and Ada do, but few versions of BASIC have such facilities.

Arrays of records (tables)

17.44 Suppose we wish to store the following table of data for a set of examination students.

CANDIDATE NUMBER	CANDIDATE NAME	AVERAGE MARK
1209	Smith	55.2
1215	Brown	86.4
1226	Jones	49.2
1235	Robinson	66.0
1247	Finlay	59.8
1253	Johnson	71.3

17.45 This data could be stored in an array of records declared in the following manner:

```
variables
        candidate:array [1..6] of record
                                        number :integer
                                        name   :string
                                        av_mark:real
                                endrecord
```

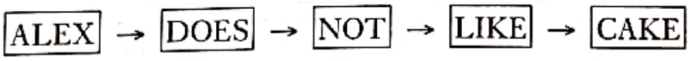
17.46 Assuming that the data had already been assigned to records in the array, and that "i" had been declared as an integer, the data could be output using the following pseudocode:

```
Begin
        for i:=1 to 6
                        with
                        candidate(i)
                        do
                        output (number, name, av_mark)
                        endwith
        endfor
end
```

Lists

17.47 Lists provide a flexible way of handling data items in order. Changes to the order can be achieved with minimal data movement and little loss of storage space.

17.48 **Example.** The sentence "Alex does not like cake" is written as a list.



We regard each word in our sentence as a data-item or **datum**, which is *linked* to the next datum, by a **pointer**. Datum plus pointer make one **element** or **node** of the list. The last pointer in the list is a **terminator**. This list may be stored in an array of records as shown in Fig. 17.14 (see page 190). Each row of the array is one element in the list. Two other pointers are useful. A **start pointer** saying where the first datum is and a **free storage pointer** saying where the next datum can go.

17.49 A data declaration of this list might take the form:

```
variables
        sentence: array [1..7] of record
                                        datum : string
                                        next  : integer
                                endrecord
```

17.50 Assuming that the start pointer "start" and row value "row" have been declared as integers then the following pseudocode could be used to output the list elements in the correct order.

```
Begin
        (* Get first datum's row *)
        row := start
        while
            (* terminator not found *)
```

```
        row <> -1
    do
        (* output next datum *)
        output(sentence(row).datum)
        (* get next row *)
        row := sentence(row).next
    endwhile
end
```

START POINTER | 1 →

FREE STORAGE POINTER | 6 →

Row Number	Datum	Pointer to Next Datum	Comment
1	"ALEX"	2	Next datum is in row 2
2	"DOES"	3	Next datum is in row 3
3	"NOT"	4	Next datum is in row 4
4	"LIKE"	5	Next datum is in row 5
5	"CAKE"	−1	Last Datum: −1 is a terminator
6		7	Empty
7		8	Empty

Note. −1 is used as a terminator in this example.

Fig. 17.14. A List Stored in an Array of Records.

17.51 **Deletions.** When elements are deleted from a list, the freed storage can be reused. In the example given here "NOT" has been deleted so the list now reads "ALEX DOES LIKE CAKE".

START POINTER | 1 →

FREE STORAGE POINTER | 3 →

Row Number	Datum	Pointer to Next Datum	Comment
1	"ALEX"	2	
2	"DOES"	4	Pointer to deleted item changed to deleted pointer
3		6	Deleted pointer replaced by free storage pointer
4	"LIKE"	5	
5	"CAKE"	−1	
6		7	
7		8	

The free storage point is replaced by the pointer to the deleted item.

17.52 **Insertions.** The free storage pointer is used for insertions. In this example the list is changed to "ALEX DOES LIKE CREAM CAKE" by inserting "CREAM".

START POINTER	1	→

FREE STORAGE POINTER	6	→

Row Number	Datum	Pointer to Next Datum	Comment
1	"ALEX"	2	
2	"DOES"	4	
3	"CREAM"	5	Old pointer given to free storage
4	"LIKE"	3	Old pointer given to new item. New pointer from free storage
5	"CAKE"	–1	
6		7	
7		8	

17.53 The pseudocode of 17.50 can be used to output the lists of 17.51 and 17.52. Extreme cases can be dealt with in these ways:

a. An empty list can be indicated by a terminator (eg, -1) in the start pointer.

b. A full list can be indicated by a terminator in the free storage pointer.

17.54 In the examples just given we have, strictly speaking, only been considering one method of **implementing** lists, namely linked lists in the form of arrays of records. A more general notation for lists is one in which the data elements are written in sequence thus:

$$< e_1, e_2, e_3, \ldots\ldots, e_n >$$

17.55 As with all data types what characterises a list is not only the set of values it takes but the operations that may be performed upon it. One would normally expect to be able to perform these operations upon a list:

a. Find the position and value of the first element.

b. Insert an element.

c. Delete an element.

d. Find the next element (sometimes the previous element too).

e. Find a specific element.

f. Output the list elements in order.

g. Create an empty list.

Trees

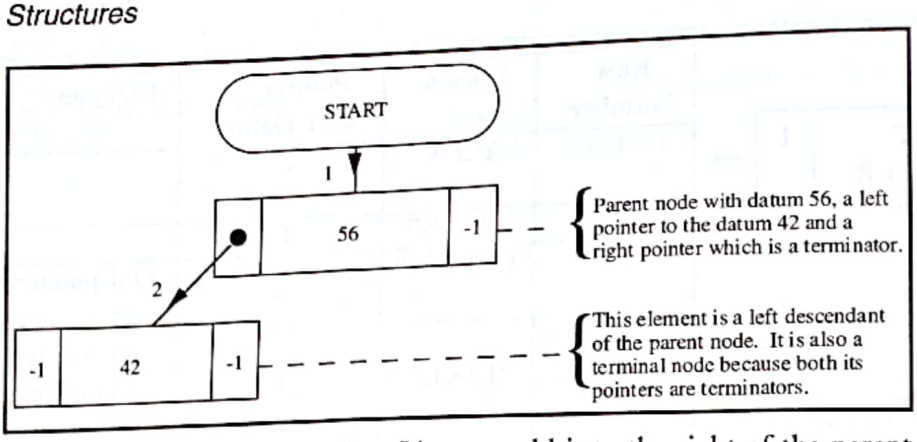
17.56 Trees are **hierarchical data structures** rather like the familiar family tree. They are constructed using a **rule of precedence** for the data items, eg, using alphabetic or numerical sequence. The **elements** of a tree are called **nodes** and each element consists of a datum and at least two pointers. In the example about to be given the examination marks used in 17.6 are entered into a tree so as to indicate their numerical order. Each element looks like this:

LEFT POINTER	DATUM	RIGHT POINTER

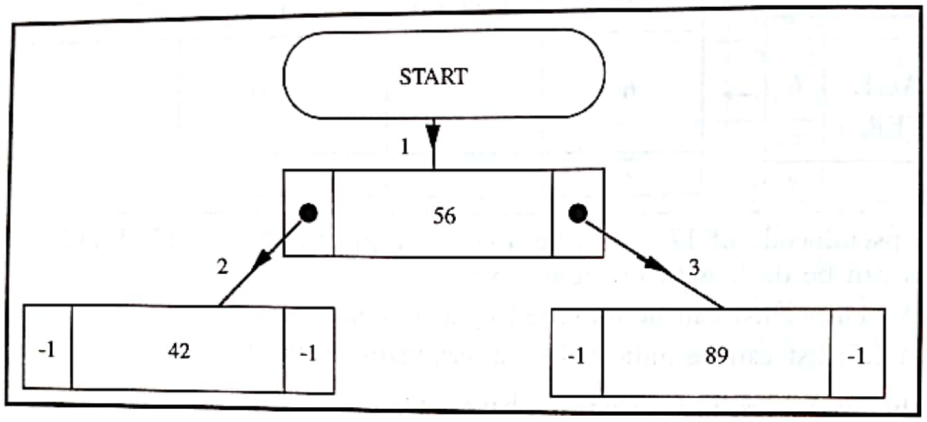
17.57 **Example.** These numbers are placed into a tree: 56, 42, 89, 65 and 48.

a. 56 is the *first* datum placed into the tree. Its node is therefore called the **parent** or **root node.**

b. We add 42 to the tree next using the precedence rule lower numbers to the left, higher numbers to the right. The tree then looks like this:

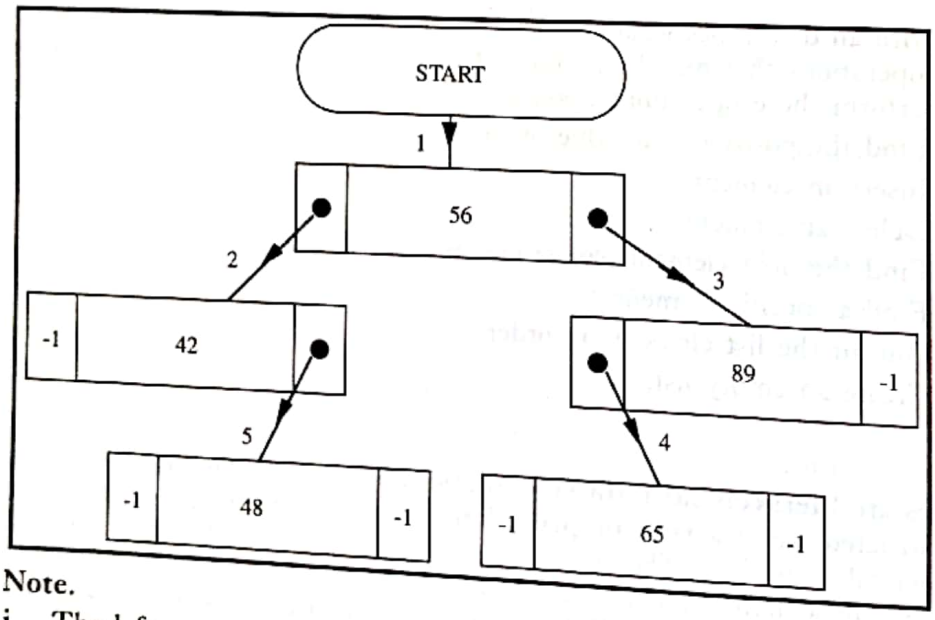


c. 89 is added next. It is larger than 56 so we add it to the right of the parent node thus:



d. 65 is added next. It is larger than 56 so we go right at the parent node but left at 89 because it is smaller.

e. 48 is added next. We go left at 56 because it is smaller and right at 42 because it is larger. Our tree finally looks like this:



Note.

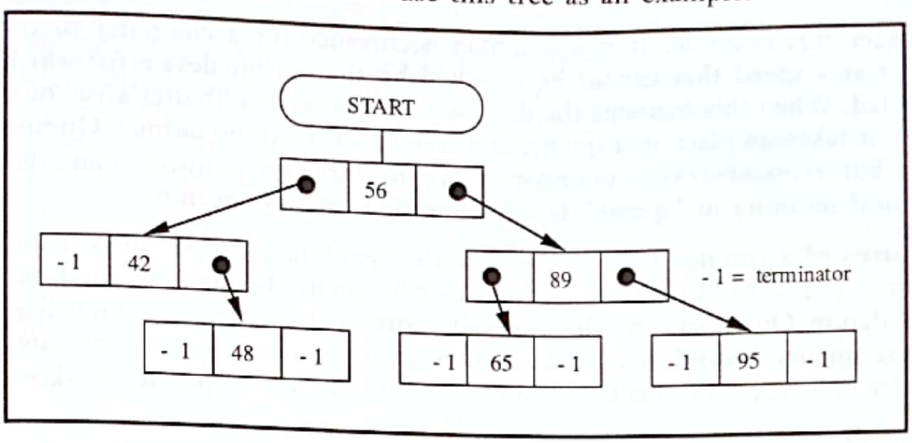
i. The leftmost node contains the smallest number.

ii. The rightmost node contains the highest number.

17.58 The tree just given in the example could be placed in an array of records as shown below.

Left Pointer	Datum	Right Pointer
2	56	3
–1	42	5
4	89	–1
–1	65	–1
–1	48	–1

17.59 Deletions from a tree. We will use this tree as an example.

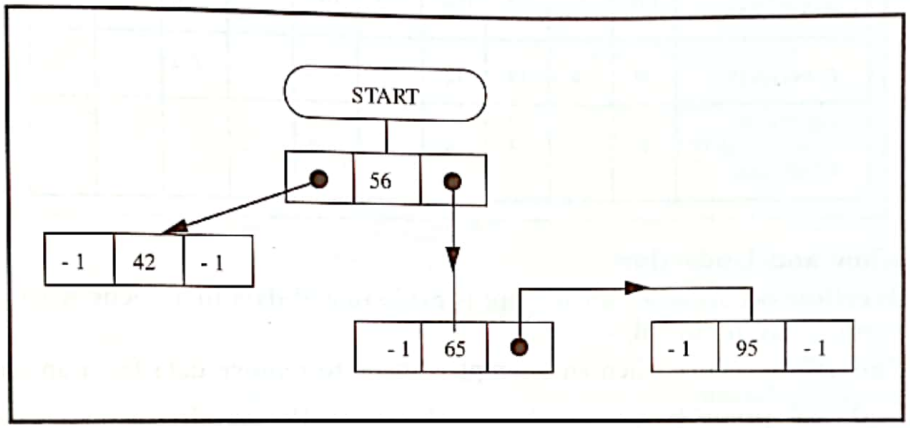


a. **Method.**

Case (i) **Deletion of a terminal node**, eg, 48. Replace the pointer to the node by a terminator.

Case (ii) **Deletions within the tree**, eg, 89. Change the rightmost node of the left-hand subtree below the node to be deleted. (If there is no left-hand subtree, change the first right-hand node below the node to be deleted.)

b. **Example** (48 has been deleted as in case (i) and 89 has been deleted as in case (ii)).



17.60 Other tree pointers.

These pointers are often used as part of the tree elements:

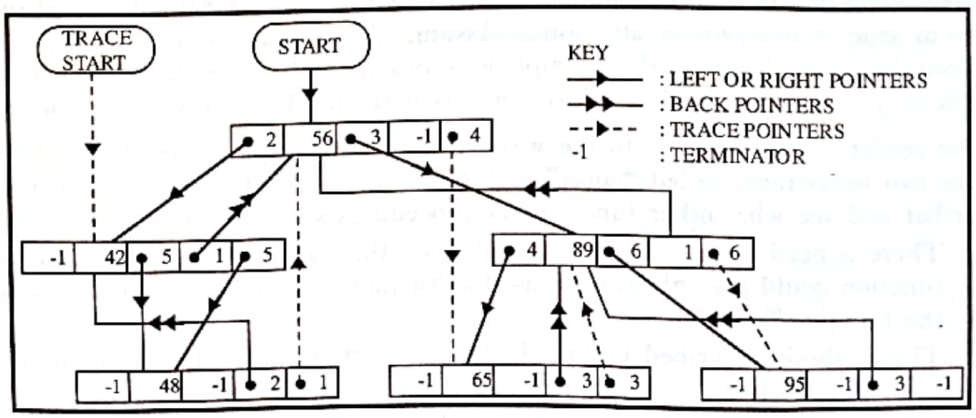
a. **Back pointers**, which give the position of the parent of each node.

b. **Trace pointers**, which point to the next node in numerical order for each node. This allows data to be read in sequence.

The complete tree element may be arranged thus:

LEFT POINTER	DATUM	RIGHT POINTER	BACK POINTER	TRACE POINTER

The complete picture looks like this.

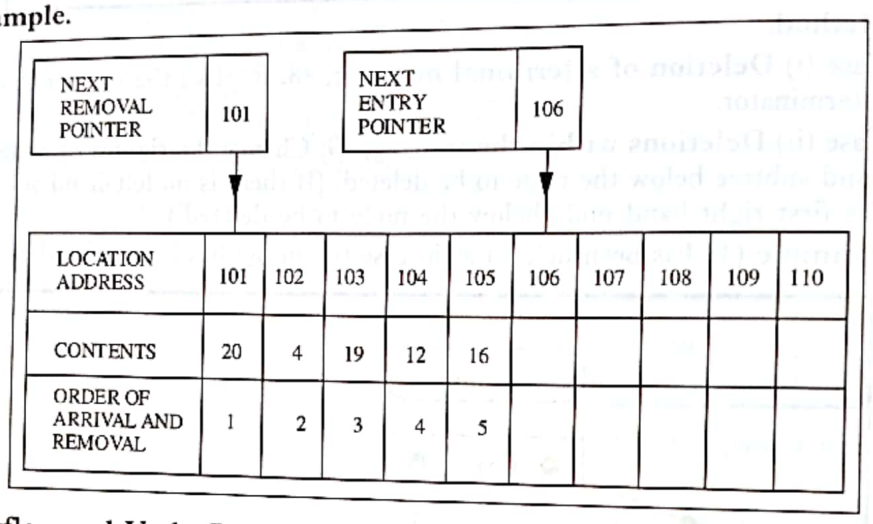


Queues

17.61 Consider this example. It is a common occurrence for a computer to create data for output at a speed that cannot be matched by the output device for which the data is intended. When this happens the data may be placed in a "buffer area" of main storage, where it takes its place in a queue, and waits its turn to be output. **Queues** have many uses, but remember this example as we now define a queue more precisely. The technical meaning of "queue" is not quite its everyday meaning.

17.62 **Features of a queue.** Data is added to the "end" of a queue but is removed from the "front". The term "**FIFO**" is used to describe queues because the First datum In is the First datum Out. Data in what we call a **queue** is *not* moved along like people in a cinema queue. Instead, each datum stays in its storage location until its turn comes, thereby reducing time spent in data movement. The use of pointers makes this possible.

17.63 **Example.**



LOCATION ADDRESS	101	102	103	104	105	106	107	108	109	110
CONTENTS	20	4	19	12	16					
ORDER OF ARRIVAL AND REMOVAL	1	2	3	4	5					

17.64 **Overflow and Underflow.**

a. **Overflow** occurs when an attempt is made to add data to a queue when all available locations are occupied.

b. **Underflow** occurs when an attempt is made to remove data from an empty queue.

17.65 **Example of queue handling.** Assume locations 101 to 110 are available to store a queue with each location storing just one number. Assume that two procedures called "place" and "fetch" are available as follows:

procedure place (**IN** pointer, number: **integer**)

(*This procedure places number in the memory location whose address value is pointer.*)

procedure fetch (**IN** pointer : **integer OUT** number : **integer**)

(* This procedure fetches number from the memory location whose address value is pointer.*)

The methods for entering and removing numbers from the queue are outlined in the pseudocode in Fig. 17.15. You should note that the Boolean variable called overflow is set to true if overflow is attempted. Assume that initially entry pointer = 101 and removal pointer = 100. In this example locations at the front of the queue are not reused once they become empty. Instead, the process is restarted once the whole queue is empty.

17.66 The reader should be able to see ways in which to adapt the methods of Fig. 17.15 into two procedures called "enter" and "remove" say. Let us pursue the idea a stage further and see what other functions or procedures should be considered.

a. There is need for a function that initialises the queue to an empty state. The same function could possibly also be used subsequently to reset the queue, so let us call the function "clear".

b. The methods described in Fig. 17.15 deal with attempts to either enter data to a full queue or to remove data from an empty queue. Provision of Boolean functions called `"q_is_full"` and `"q"_is_empty"` would help to avoid such a mistake.

```
(* Entering a number *)
(* outline method *)
variables
  entry_pointer,
  removal_pointer,
  number      : integer
  overflow : boolean
Begin
  (* assume previous initialisation *)
  if
    entry_pointer = 111
  then
    overflow := true
  else
    (* place number in location given
    by entry_pointer *)
    call place(entry_pointer, number)
    (* update pointers *)
    entry-pointer := entry_pointer + 1
    if
      removal_pointer = 100
    then
      removal_pointer := 101
    endif
  endif
endif
```

a.

```
(* Removing a number *)
(* outline method *)
variables
  entry_pointer,
  removal_pointer,
  number      : integer
  underflow : boolean
Begin
  (* assume previous initialisation *)
  if
    removal_pointer = 100
  then
    underflow := true
  else
    (* fetch number from location
    given by removal_pointer *)
  call fetch(removal_pointer, number)
  if
    (* queue is now empty *)
    removal_pointer = 110
    OR
    entry_pointer = removal_pointer + 1
  then
    (* reset queue for reuse *)
    removal_pointer := 100
    entry_pointer := 101
  else
    (* update removal_pointer *)
    removal_pointer := removal_pointer + 1
    endif
  endif
endif
```

b.

Fig. 17.15. Queue Handling Example.

NB. Use this pseudocode to create and then remove the data given in 17.63.

17.67 Some programming languages allow sets of functions and procedures associated with a data structure such as a queue to be grouped together into a single programming unit called a **module** or **package**.

17.68 The user of a queue module might have the following simple functions and procedures provided by the module:

a. **Procedure** clear. A parameterless procedure that sets the state of the queue to empty.

b. **function** q_is_empty: **Boolean**. A parameterless function that returns **true** when the queue is empty and returns **false** otherwise.

c. **function** q_is_full: **Boolean**. A parameterless function that returns **true** when the queue is full and returns **false** otherwise.

d. **Procedure** enter (**IN** number). This procedure places a number on the end of the queue. If the queue is full the procedure has no effect.

e. **Procedure** remove (**OUT** number). This procedure removes a number from the head of the queue and returns it. If the queue is empty, an undefined value is returned.

17.69 When entering and removing data these functions need to be used in conjunction with one another, eg,

a. To add a number

```
if
   q_is_full
then
   output ("queue is full")
else
   enter (number)
endif
```

b. To remove a number

```
if
   q_is_empty
then
   output ("no data in queue")
else
   remove (number)
   output ("next value is", number)
endif
```

Note the following points, which arise from these examples.

a. A data structure described in terms of the operations performed upon it, as the queue was in paragraph 17.68, may be called an **Abstract Data Type (ADT)**.

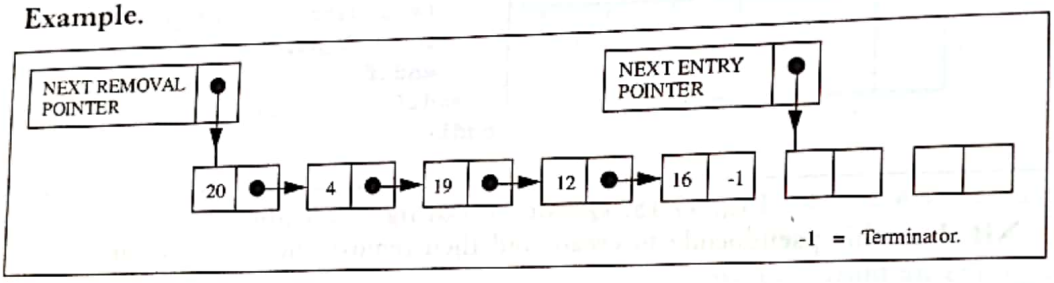
b. The functions in paragraph 17.69 do not pass either the stored form of the queue or its pointers because they are already contained within the module and are **global** to all functions and procedures within the package. They are local to the module, however, which is why this method of representing the handling of data structures is sometimes called **information hiding**.

NB. In functional decomposition these data items would be passed as parameters.

17.70 The use of abstract data types (information hiding) is important in advanced programming, but further discussion of such methods is beyond the scope of this text.

17.71 **Queue organisation in lists.** Queues may be stored in the form of lists (17.47). A list used to store a queue is called a **Push-up list.**

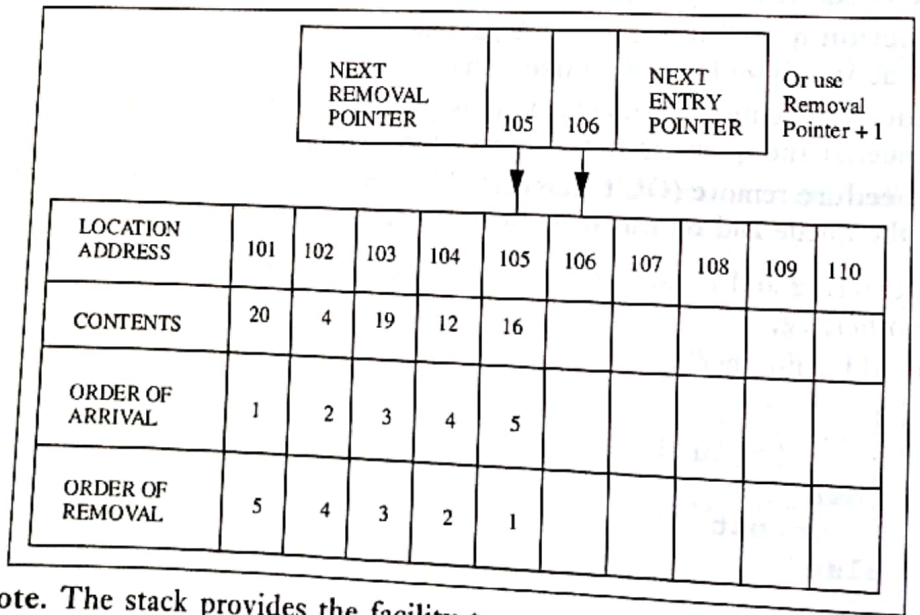
Example.



Stacks

17.72 Stacks are frequently used for temporary storage, but in a different way from queues. A stack differs from a queue in the method of data addition and removal. Data is added to the "top" of a stack and is also removed from the "top". The Last datum In is the First datum Out (**LIFO**). Compare this with a pack of cards on a table with cards either added to or removed from the top.

17.73 **a. Example.**



LOCATION ADDRESS	101	102	103	104	105	106	107	108	109	110
CONTENTS	20	4	19	12	16					
ORDER OF ARRIVAL	1	2	3	4	5					
ORDER OF REMOVAL	5	4	3	2	1					

Note. The stack provides the facility to retrace the steps of data insertion.

b. **Example of use.** One common application of stacks is for storing return addresses (link values) for closed subroutines (32.30). When entering a subroutine, the return address is placed on top of the stack. Should a second subroutine be entered from the first, then the return address will again be placed on the stack. When the time comes to return from the second subroutine to the first, the correct return address will be on top of the stack. The return address will be removed revealing the return address to the main program. The same method can be applied when many subroutines are "nested" in this manner and proves useful in other applications, which are discussed later.

17.74 **Example of stack handling** (we make the same assumptions made in 17.65 except that locations will be reused if they become vacant). The methods for entering and removing numbers from the stack are outlined in pseudocode in Fig. 17.16. The operation of placing an item on a stack is sometimes called "**push**" and the operation of removing an item from a stack is sometimes called "**pop**".

```
(* Entering a number *)
(* outline method *)
variables
  entry_pointer,
  removal_pointer,
  number : integer
  overflow    : boolean
Begin
  (* assume previous initialisation *)
  if
    entry_pointer = 111
  then
    overflow := true
  else
    (* place number in location given
    by entry_pointer *)
    call place(entry_pointer, number)
    (* update pointers *)
    removal-pointer := entry_pointer
    removal-pointer := entry_pointer
    entry-pointer := entry_pointer + 1
    then removal_pointer = 101
  endif
end
a.
```

```
(* Removing a number *)
(* outline method *)
variables
  entry_pointer,
  removal_pointer,
  number : integer
  overflow    : boolean
Begin
  (* assume previous initialisation *)
  if
    removal_pointer = 100
  then
    underflow := true
  else
    (* fetch number from location
    given by removal_pointer *)
  call fetch (removal_pointer, number)
  (* update pointers *)
  entry_pointer = removal_pointer
  removal_pointer := removal_pointer - 1
    endif
  endif
end

b.
```

Fig. 17.16. Stack Handling Example.

NB. Use this pseudocode to create and then remove the data given in 17.73.

17.75 As for the queue, the stack may also be described in terms of the operations that may be performed upon it. There are numerous ways in which this may be done. Typical functions are:

a. A **clear** procedure analogous to that described for the queue.

b. Functions called "stack_is_empty" and "stack_is_full", also analogous to those for the queue.

c. A **push** procedure, eg, **procedure** push (**IN** number), to insert numbers on to the top of a non-full stack.

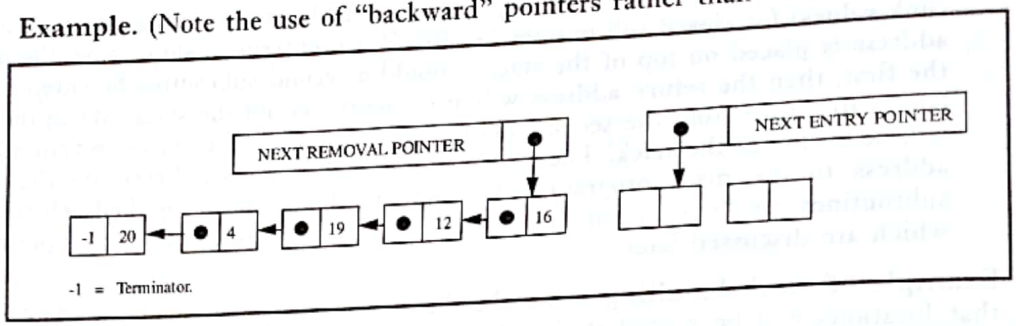
d. A **pop** procedure. The pop procedure, which involves removing the top items from the stack, is sometimes split into two operations:

   i. **pop**, which merely deletes the top item from the stack, and

   ii. **top**, which returns the stack's top item to the calling program without deleting it from the stack.

17.76 Stack organisation in lists. Lists used to store stacks are called "push-down lists".

Example. (Note the use of "backward" pointers rather than "forward" pointers.)



Search and access

17.77 **Search time and search length** (when accessing data). An important characteristic of a data structure is its search time. The **search time** is the **average time** taken to find a datum in the structure. It is often expressed in terms of the **search length**, which is the average number of elements examined in order to find a datum.

In situations where data items are required individually the data structure with the shortest **search length** will be preferable, since it will save processor time and thereby increase efficiency.

17.78 The linear search (the simplest and most common type of search). To perform a **linear search** the data is examined one element at a time in physical order until the required datum is found. (See 17.26 for the method.)

Example.

a. Suppose we had a table with 5 entries. The 5 possible search lengths are 1, 2, 3, 4, 5. For example, to find the third entry takes 3 examinations. The average search length will be

$$\frac{1 + 2 + 3 + 4 + 5}{5} = 3$$

b. For a table with N entries the average search length would be:

$$\frac{1 + 2 + 3 + 4 + \ldots\ldots + (N - 1) + N}{N}$$

This sum can be simplified if we notice that writing the series backwards can add it twice easily:

$$\begin{array}{ccccccccccc} 1 & + & 2 & + & 3 & + & 4+\ldots & + & (N-1) & + & N \\ N & + & (N-1) & + & (N-2) & + & (N-3)+\ldots & + & 2 & + & 1 \\ \hline (N+1) & + & (N+1) & + & (N+1) & + & (N+1)+\ldots & + & (N+1) & + & (N+1) \end{array}$$

There are N terms.

So twice the sum $1 + 2 + 3\ldots\ldots + N$ is $N(N + 1)$

So average search length $= \frac{N(N+1)}{2N} = \frac{(N+1)}{2}$

A linear search can be performed on all the data structures described so far. (Structures like trees are constructed so as to reduce search length.)

17.79 **Search length for a binary tree.** For a well-balanced tree the search length is approximately $\log_2 N$, where N is the number of elements in the tree (eg, if N = 32 the search length is 5, because $2^5 = 32$ or $\log_2 32 = 5$).

To justify the use of this formula we can see that:

If we had two (2^1) nodes we would find the datum after one examination.

If we had four (2^2) nodes we would find the datum after 2 examinations.

If we had eight (2^3) nodes we would find the datum after 3 examinations, etc.

17.80 **Binary search.** A sorted array can be searched in a way analogous to the method used for searching a binary tree; this is called a binary search. In a binary search the value

in the middle of the array is examined to see if it is equal to the value being searched for. If the searched-for value is not equal to the middle value, then the process is repeated with either the first half of the array or the second half of the array according to whether the middle value is larger or smaller than the searched-for value. The process is repeated with smaller and smaller subdivisions of the array until either the value is found or there are no more subdivisions to make.

NB. The binary search has a search length of $\log_2 N$.

17.81 Many attempts to reduce search times are based upon the use of tables such as the ACCESS TABLE (17.36). Various types of table used to improve the access by reducing search time are described in the following paragraphs.

Tables and their uses

17.82 Keys. Accessing most tables described in this section involves using keys. A key is a data item that is associated with the data and that can be used to locate or identify other data. It may form part of the data to be located, or it may be quite separate. A subscript of an array may be thought of as a key of this second type.

17.83 To illustrate the various types of table we will take as an example data that is the telephone numbers of various people. You may think of the telephone number as the datum item to be accessed and the person's name as the key to that number, assuming names are unique.

NAME (KEY)	TELEPHONE NUMBER (DATUM)
ALEX	3012
CHARLOTTE	7641
EMMA	5962
JAMES	4243
KATY	2126
LUCY	3562

17.84 **LOOK-UP tables.** (Types of access table used independently or in conjunction with other data structures.)

a. Suppose that each location in main memory can store 4 characters. The telephone number can be stored like this for example.

LOCATION ADDRESS	101	102	103	104	105	106
CONTENTS	3012	7641	5962	4243	2126	3562

b. Given a name, "CHARLOTTE" say, we may want to find the corresponding telephone number. The key "CHARLOTTE" is too long to fit into one location in this case, but using just the first four letters may do just as well, and that is what we use in this example (ie, CHAR).

c. This is the look-up table:

KEY	ADDRESS OF CORRESPONDING DATA
ALEX	0101
CHAR	0102
EMMA	0103
JAME	0104
KATY	0105
LUCY	0106

NB. In this simplified example the 4-character data could occupy the space given to its 4-character addresses, thus removing the need for the look-up table! *In general the data may occupy several successive locations, the first of which is specified in the look-up table.*

d. This is how the look-up table could be stored:

LOCATION ADDRESS	81	82	83	84	85	86	87	88	89	90	91	92
CONTENTS	ALEX	1011	CHAR	0102	EMMA	0103	JAME	0104	KATY	0105	LUCY	0106

e. To find the required datum we perform a **linear search** in the look-up table for the datum's key and thereby find the address of the datum.

17.85 DIRECT-ACCESS tables.

Any datum in the direct-access table can be accessed directly, ie, *without search*. This is achieved by using a *mapping function*, ie, a formula or procedure that is applied to each key to produce the location address of the corresponding datum.

17.86 Example.

a. **Mapping function.** Consider a simple example in which the first letter of each key is given a value corresponding to its position in the alphabet, ie, A = 1, B = 2, C = 3, etc. A possible mapping function is

LOCATION ADDRESS = 100 + KEY VALUE

For example, for "CHARLOTTE" the key value is "3" and the location address of the datum will be 100 + 3 = 103.

b. The data is stored and accessed as shown in this table. *Only the data is stored.*

KEY	FIRST LETTER OF KEY	KEY VALUE	LOCATION ADDRESS (=100 + KEY VALUE)	CONTENTS
ALEX	A	1	101	3012
	B	2	102	
CHARlotte	C	3	103	7642
	D	4	104	
EMMA	E	5	105	5962
	F	6	106	
	G	7	107	
	H	8	108	
	I	9	109	
JAMEs	J	10	110	4243
KATY	K	11	111	2126
LUCY	L	12	112	3562
	M	13	113	

Note

i. There is a waste of storage space; the price of direct access.

ii. This simple mapping function would not be suitable if two keys started with the same letter, so it is hardly practicable.

17.87 HASH tables. (These use methods very similar to those described in 22.34 on random files). The hash table is a compromise between the direct-access table technique and the need to reduce unused storage space. Hash table mapping functions are allowed to be ambiguous, ie, generate the same location address for two different keys. When this happens, and data is allocated to storage space that is already occupied, a procedure is used to find it alternative storage space within the table.

17.88 Example: The "open" hash table.

a. In this example

i. The datum's key is also stored with the datum, so that the datum may be identified when accessed.

ii. The mapping function is:

DATUM LOCATION ADDRESS = 100 + (2 x KEY VALUE)

KEY LOCATION ADDRESS = DATUM LOCATION ADDRESS – 1

eg, CHARLOTTE's datum will have the location address 100 + (2 x 3) = 106 and the key CHAR will have location address 106 – 1 = 105.

iii. If a datum is assigned a location that is already occupied, the datum will be placed in the next empty location instead. (This is the "open" hash method).

b. The hash table looks like this after seven entries:

KEY	FIRST LETTER OF KEY	KEY VALUE	DATUM		KEY	
			LOCATION ADDRESS [100+ 2 × (KEY VALUE)]	CONTENTS	LOCATION ADDRESS [DATUM ADDRESS –1]	CONTENTS
ALEX	A	1	102	3012	101	ALEX
BERNard	B	2	104	4251	103	BERN
CHARlotte	C	3	106	7641	105	CHAR
	D	4				
EMMA	E	5	110	5962	109	EMMA
	F	6				
	G	7				
	H	8				
	I	9				
JAMEs	J	10	120	4243	119	JAME
KATY	K	11	122	2126	121	KATY
LUCY	L	12	124	3562	123	LUCY
	M	13	126			

c. If we now try to add a datum "5482" with the key "ANN", it will be allocated location 102, which is occupied. The next available space is that normally used by "D" and so the datum "5482" will be placed in location 108 and its key "ANN" will be placed on location 107.

d. The table is **"circular"** if values not accommodated at its end are referred to the start of the table, (eg, refer "Z" to "A").

17.89 **Other features of hash tables in brief.** (A detailed understanding is not called for.)

a. Hash tables work most efficiently when the data is evenly spread out rather than clustered together. If clustering does occur then either the method of choosing the next available location must be changed, or a complete rehash must be made (ie, the table must be remade using a different mapping function).

b. An alternative to the **open hash** is the **closed hash**, in which instead of using adjacent locations in the table for surplus data a **pointer** is placed in the table indicating where the **"overflow"** data may be found.

c. The search length for a hash table is approximate.

$$\frac{2L - N}{2(L - N)}$$

where L is the length of the table and N is the number of entries. The proof of this formula is beyond the scope of this book.

17.90 **The binary search in tables.** When a direct-access table or hash table *have not been used* it is often possible to improve over a **linear search** by means of a **binary search**, provided the table is in some logical order.

Summary

17.91 a. The data structures discussed were:

i. One-dimensional **arrays**, in which data is placed in elements arranged into a sequence numbered by **subscripts**, and two-dimensional arrays, in which the elements are arranged into rows and columns.

ii. Fixed- and variable-length **strings**, which contain sequences of characters.
iii. **Access tables**, which are arrays containing details of where data values can be accessed (found).
iv. **Records**, in which the elements are a series of fields that may each be of different types.
v. Arrays of records in which each element of the array is a record.
vi. **Lists**, in which individual elements (**nodes**) containing data and **pointers** are connected together by the use of the pointers.
vii. **Trees**, in which data is organised in a hierarchical manner. Each **element** (**node**) has left and right pointers.
viii. **Queues**, in which the first item added is the first item removed (FIFO).
ix. **Stacks**, in which the last item added is the first item removed (LIFO).
x. **Look-up tables**, in which the address of a datum is found by a linear search for its key.
xii. **Direct-access tables**, in which mapping functions are used on the key of a datum to give its location directly.
xiii. **Hash tables**, in which mapping functions are also used but for which the location generated is not unique.

b. **Sorting** and **searching** were discussed.
c. The **search length** for various structures was discussed:
i. The **linear search** is a basic method that may be used in any data structure. Search length = $(N + 1)/2$.
ii. The **binary search**, which can be used on tables in logical order, usually provides quicker than a linear search. Search length = $\log_2 N$.
iii. If well balanced, the **binary tree** structure produces the same search length as a binary search.
iv. A direct-access table provides access to the data without a search, ie, search length = 1.
v. A hash table is almost as fast as a direct-access table when it only has a few entries, but otherwise still gives fast access.
Search length $\approx (2L - N)/(2L - 2N)$.

Points to note

17.92 a. One-dimensional arrays may be called **vectors** and main storage may therefore be regarded as a vector.
b. The terms **table** and **array** are synonymous most of the time, but for applications in main storage the term table is usually used. The term table is also commonly used to mean an array of records.
c. Arrays of strings may be handled in ways very similar to those used in 17.8 to 17.26.
d. Arrays are often used with subscripts starting at 0 rather than 1 .
e. Pointers may also be called **links** and lists may be called linked lists or even **threaded lists** because of their appearance.
f. The **trees** introduced in this chapter had nodes with just *two* pointers and are called *binary trees*. More complicated trees will be dealt with later.
g. As mentioned earlier the first node in a tree may be called the root or **parent**. However, the term **parent** is also used more generally to describe *any* node having nodes below it. The nodes immediately below a parent are its **child** nodes. Therefore, it is perhaps safer to use the term **root** for the first node. Nodes without child nodes are called **leaf** nodes.
h. A data type or structure considered purely in terms of the sets of values that variables of the type can take and the set of operations that may be performed upon those variables *without* considering how it is implemented is called an **abstract data type** (ADT).
i. If you are asked to compare data structures then consider these points:
i. Method of access/storage.
ii. Use and wastage of storage space.

iii. Application. (Nature and use of data.)
iv. Flexibility in handling variable data, overflow, etc.

j. Try to gain practical experience in setting up and handling various data structures.

k. **Double buffering** is an alternative to buffering. One **queue** is filled while the other is emptied.

l. Sorting within main memory, and therefore without the need to use backing storage, is called **internal sorting**. Some methods of internal sorting are much faster than others, as was indicated earlier. As a final example the quicksort method is outlined below.

m. **Quicksort.** This method is one of the most efficient methods of internal sorting but is rather more complicated. The method is recursive and is applied to smaller and smaller subdivisions of the array to be sorted. In this example we will consider a sort into ascending sequence.

Procedure.

1. Use the first value as the **pivot** value (ideally the pivot will be a middle value in terms of magnitude).

2. Permute the remaining values so that for a given subscript (the partition subscript) all values with this and lower subscripts are lower in value than the pivot and all values with higher subscripts are larger than or equal to the pivot. To permute the values perform a set of swaps in which successive high values in the lower part of the array are swapped with successive low values in the upper part.

3. Swap the pivot value with the value at the partition subscript.

4. Re-apply the procedure to the part of the array above the partition subscript and to the part below the partition subscript.

Step 1	Step 2	Step 3	Step 4	Step 5	etc.
10 ↓	32 ↓	51	51 ↓	53	53
32 ↑	10 ↑↓	32	32 ↑↓	51	51
51 raise 32 lower 10	51 ↑	10 ↓	24 ↑↓	32 ↓	43
24	24 raise 51 lower 10	24 ↑	10 ↑↓	24 ↑↓	32
53	53 and 32	53 raise 24 lower 10	53 ↓	10 ↑↓	24
43	43	43	43	43 ↑	10
36	36	36	36	36	36
37	37	37	37	37	37
47	47	47	47	47	47
21	21	21	21	21	21
15	15	15	15	15	15

Student self-test questions

1. The array B shown below contains 6 characters. Give the order in which you would select elements in order to spell the word "MATRIX".

T R X
I A M

2. Enter the letters of "MATRIX" into a tree one at a time in the order they are spelt. Use the rule "earlier in the alphabet to the left".

3. Write the function "below average" used in Fig. 17.1.

4. Write a procedure to reverse the order of elements in
   a. an array, eg, [3, 6, 2, 5] becomes [5, 2, 6, 3]
   b. a string, eg, MATRIX becomes XIRTAM.

5. Describe a suitable data structure for details of stock items numbered in the range 1 to 100. Each stock item may be held at each of 20 locations. The number of items held at each location needs to be recorded.

6. What advantages can a list have over simple strings or tables? Make reference to data storage within main store to illustrate your answer.

7. Name three types of table and compare their features.

Questions without answers

8. The linked list shown in this chapter allows us to pass through the list in one direction only, ie, given only the location of an item we do not know its predecessor. Write a record declaration for a list so that from a single element we know both the location of its predecessor and its successor. How does this differ from a tree?

9. Write a function that determines the number of words in a string such as that shown in Fig. 17.9. State any assumptions that you make.