Chapter 13:
Data types and structures

Learning objectives

By the end of this chapter you should be able to:

- select and use appropriate data types for a problem solution (INTEGER, REAL, CHAR, STRING, BOOLEAN, DATE, ARRAY, FILE)
- show understanding of the purpose of a record structure to hold a set of data of different data types under one identifier
- write pseudocode to define a record structure
- write pseudocode to read data from a record structure and save data to a record structure
- use the technical terms associated with arrays including index, upper and lower bound
- select a suitable data structure (1D or 2D array) to use for a given task
- write pseudocode to process array data including sorting using a bubble sort and searching using a linear search.
- show understanding of why files are needed
- write pseudocode to handle text files that consist of one or more lines
- show understanding that an Abstract Data Type (ADT) is a collection of data and a set of operations on those data
- show understanding that a stack, queue and linked list are examples of ADTs
- describe the key features of a stack, queue and linked list and justify their use for a given situation
- use a stack, queue and linked list to store, add, edit and delete data
- describe how a queue, stack and linked list can be implemented using arrays.

13.01 Data types

Primitive data types

In Chapter 12 we used variables to store values required by our algorithm. Look at Worked Example 12.01. The Identifier Table 12.02 lists two variable identifiers: `Miles` and `Km`. An identifier table should also show what sort of data (or data type) is going to be stored in each variable. The explanation shows that Miles will be a whole number, but that Km will be calculated using the formula Miles * 1.61. This will result in a number that may not be a whole number.

Primitive data types are those variables that can be defined simply by commands built into the programming language. Primitive data types are also known as atomic data types. In Computer Science a whole number is referred to as an INTEGER and a number with a decimal point is referred to as a REAL. Conditions are either TRUE or FALSE. These are logical values known as BOOLEAN. Sometimes we may want to store a single character; this is referred to as a CHAR.

A value that will always be a whole number should be defined to be of type INTEGER, such as when counting the iterations of a loop.

TIP

See Table 13.01 for a list of data types you should be familiar with.

See Chapter 1 (Sections 1.02 and 1.03) on how integers and characters are represented inside the computer. Chapter 16 (Section 16.03) covers the internal representation of real (single, double, float) numbers.

Further data types

If we want to store several characters; this is known as a string.

Note that there is a difference between the number 12 and the string “12”.

The string data type is known as a structured type because it is essentially a sequence of characters. A special case is the empty string: a value of data type string, but with no characters stored in it.

When we write a date, such as 3 February 2018, we can also write this as a set of three numbers: 3/2/2018. Sometimes we might wish to calculate with dates, such as taking one date away from another to find out how many days, months and years are between these dates. To make it easier to do this, DATE has been designed as a data type. To see how different programming languages implement this data type, see Chapter 14 Section 14.03.

INTEGER	A signed whole number
REAL	A signed number with a decimal point
CHAR	A single character
STRING	A sequence of zero or more characters
BOOLEAN	The logical values TRUE and FALSE
DATE	A date consisting of day, month and year, sometimes including a time in hours, minutes and seconds

Table 13.01 Summary of pseudocode data types

TASK 13.01

Look at the identifier tables in Chapter 12 (Tables 12.06 and 12.09 to 12.12). Give the data type that is appropriate for each variable listed.

13.02 The record type

Sometimes variables of different data types are a logical group, such as data about a person (name, date of birth, height, number of siblings, whether they are a full-time student).

Name is a STRING; date of birth is a DATE; height is a REAL; number of siblings is an INTEGER; whether they are a full-time student is a BOOLEAN.

We can declare a record type to suit our purposes. The record type is known as a user-defined type, because the programmer can decide which variables (fields) to include as a record.

TIP

A record type is also known as a composite type.

In pseudocode a record type is declared as:

```
TYPE <TypeIdentifier>
    DECLARE <field identifier> : <data type>
    .
    .
ENDTYPE
```

We can now declare a variable of this record type:

```
DECLARE <variable identifier> : <record type>
```

And then access an individual field using the dot notation:

```
<variable identifier>.<field identifier>
```

Using the example above we can declare a Person record type:

```
TYPE PersonType
    Name : STRING
    DateOfBirth : DATE
    Height : REAL
    NumberOfSiblings : INTEGER
    IsFullTimeStudent : BOOLEAN
ENDTYPE
```

To declare a variable of this type we write:

```
DECLARE Person : PersonType
```

And now we can assign a value to a field of this Person record:

```
Person.Name ← "Fred"
Person.NumberOfSiblings ← 3
Person.IsFullTimeStudent ← TRUE
```

To output a field of a record:

```
OUTPUT Person.Name
```

TASK 13.02

Write the declaration of a record type to store the details of a book: Title, Year of publication, Price, ISBN.

Write the statements required to assign the values “Computer Science”, 2019, £44.95, “9781108733755” to the fields respectively.

13.03 Arrays

Sometimes we want to organise data values into a list or a table / matrix. In most programming languages these structures are known as arrays. An array is an ordered set of data items, usually of the same type, grouped together using a single identifier. Individual array elements are addressed using an **array index** for each array dimension.

A list is a one-dimensional (1D) array and a table or matrix is a two-dimensional (2D) array.

TIP

When writing pseudocode, arrays need to be declared before they are used. This means choosing an identifier, the data type of the values to be stored in the array and **upper bound** and **lower bound** for each dimension.

13.04 One-dimensional arrays

When we write a list on a piece of paper and number the individual items, we would normally start the numbering with 1. You can view a 1D array like a numbered list of items. Many programming languages number array elements from 0 (the lower bound), including VB.NET, Python and Java. Depending on the problem to be solved, it might make sense to ignore element 0. The upper bound is the largest number used for numbering the elements of an array.

In pseudocode, a 1D array declaration is written as:

```
DECLARE <arrayIdentifier> : ARRAY[<lowerBound>:<upperBound>] OF <dataType>
```

Here is a pseudocode example:

```
DECLARE List1 : ARRAY[1:3] OF STRING  // 3 elements in this list
DECLARE List2 : ARRAY[0:5] OF INTEGER // 6 elements in this list
DECLARE List3 : ARRAY[1:100] OF INTEGER // 100 elements in this list
DECLARE List4 : ARRAY[0:25] OF CHAR  // 26 elements in this list
```

Accessing 1D arrays

A specific element in an array is accessed using an index value. In pseudocode, this is written as:

```
<arrayIdentifier>[x]
```

The *n*th element within the array `MyList` is referred to as `MyList[n]`.

Here is a pseudocode example:

```
NList[25] ← 0  // set 25th element to zero
AList[3] ← 'D' // set 3rd element to letter D
```

WORKED EXAMPLE 13.01

Working with a one-dimensional array

The problem to be solved: Take seven numbers as input and store them for later use.

We could use seven separate variables. However, if we wanted our algorithm to work with 70 numbers, for example, then setting up 70 variables would be complicated and waste time. Instead, we can make use of a data structure known as a 'linear list' or a 1D array.

This array is given an identifier, for example `MyList`, and each element within the array is referred to using this identifier and its position (index) within the array. For example, `MyList[4]` refers to the element at position 4 in the `MyList` array. If we are counting the element at position 0 as the first element, `MyList[4]` refers to the fifth element.

We can use a loop to access each array element in turn. If the numbers input to the pseudocode algorithm below are 25, 34, 98, 7, 41, 19 and 5 then the algorithm will produce the result in Figure 13.01.

```
FOR Index ← 0 TO 6
    INPUT MyList[Index]
NEXT Index
```

Index	MYList
[0]	25
[1]	34
[2]	98
[3]	7
[4]	41
[5]	19
[6]	5

Figure 13.01 `Mylist` array populated by a loop

TASK 13.03

Define two arrays, one for your friends' names and one for their ages as shown in Figure 13.02.

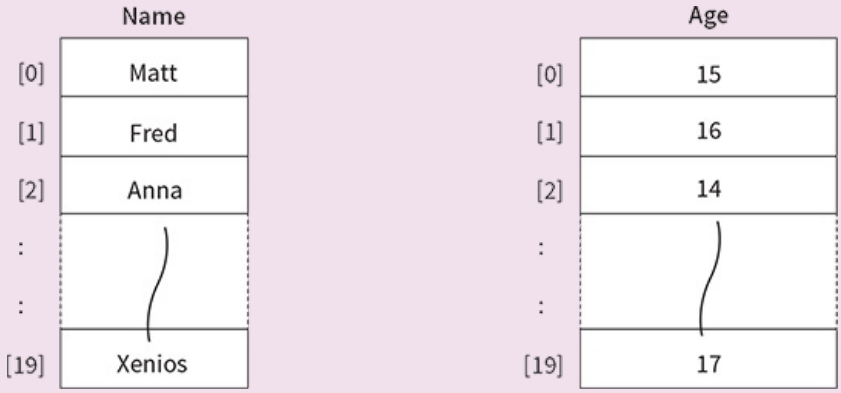


Figure 13.02 Arrays for names and ages

WORKED EXAMPLE 13.02

Searching a 1D array

The problem to be solved: Take a number as input. Search for this number in an existing 1D array of seven numbers (see Worked Example 13.01).

Start at the first element of the array and check each element in turn until the search value is found or the end of the array is reached. This method is called a **linear search**.

Identifier	Data type	Explanation
`MyList`	`ARRAY[0:6] OF INTEGER`	Data structure (1D array) to store seven numbers
`MaxIndex`	`INTEGER`	The number of elements in the array
`SearchValue`	`INTEGER`	The value to be searched for
`Found`	`BOOLEAN`	`TRUE` if the value has been found `FALSE` if the value has not been found
`Index`	`INTEGER`	Index of the array element currently being processed

Table 13.02 Identifier table for linear search algorithm

```
MaxIndex ← 6
INPUT SearchValue
Found ← FALSE
Index ← –1
REPEAT
    Index ← Index + 1
    IF MyList[Index] = SearchValue
      THEN
        Found ← TRUE
    ENDIF
UNTIL FOUND = TRUE OR Index >= MaxIndex
IF Found = TRUE
```

```
  THEN
    OUTPUT "Value found at location: " Index
  ELSE
    OUTPUT "Value not found"
ENDIF
```

The complex condition to the `REPEAT...UNTIL` loop allows us to exit the loop when the search value is found. Using the variable `Found` makes the algorithm easier to understand. `Found` is initialised (first set) to `FALSE` before entering the loop and set to `TRUE` if the value is found.

If the value is not in the array, the loop terminates when `Index` is greater than or equal to `MaxIndex`. That means we have come to the end of the array. Note that using `MaxIndex` in the logic statement to terminate the loop makes it much easier to adapt the algorithm when the array consists of a different number of elements. The algorithm only needs to be changed in the first line, where `MaxIndex` is given a value.

TASK 13.04

Use the algorithm in Worked Example 13.02 as a design pattern. Write an algorithm using the arrays from Task 13.03 to search for a friend's name and output their age.

WORKED EXAMPLE 13.03

Sorting elements in a 1D array

The simplest way to sort an unordered list of values is the following method.

1. Compare the first and second values. If the first value is larger than the second value, swap them.
2. Compare the second and third values. If the second value is larger than the third value, swap them.
3. Compare the third and fourth values. If the third value is larger than the fourth value, swap them.
4. Keep on comparing adjacent values, swapping them if necessary, until the last two values in the list have been processed.

Figure 13.03 shows what happens to the values as we work down the array, following this algorithm.

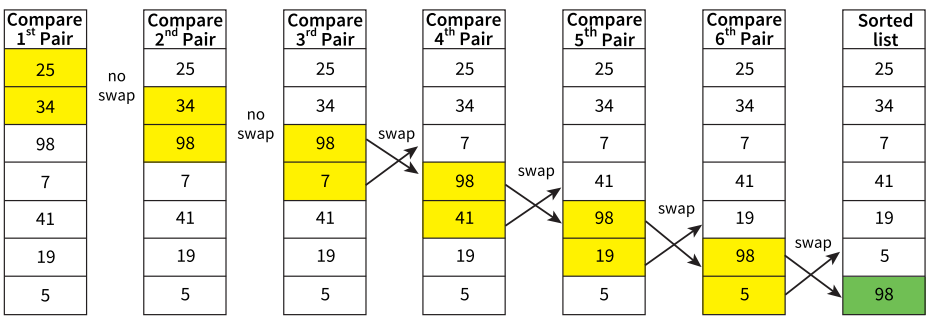


Figure 13.03 Swapping values working down the array

When we have completed the first pass through the entire array, the largest value is in the correct position at the end of the array. The other values may or may not be in the correct order.

We need to work through the array again and again. After each pass through the array the next largest value will be in its correct position, as shown in Figure 13.04.

Original list	After pass 1	After pass 2	After pass 3	After pass 4	After pass 5	After pass 6
25	25	25	7	7	7	5
34	34	7	25	19	5	7
98	7	34	19	5	19	19
7	41	19	5	25	25	25
41	19	5	34	34	34	34
19	5	41	41	41	41	41
5	98	98	98	98	98	98

Figure 13.04 States of the array after each pass

In effect we perform a loop within a loop, a nested loop. This method is known as a **bubble sort**. The name comes from the fact that smaller values slowly rise to the top, like bubbles in a liquid.

The identifiers needed for the algorithm are listed in Table 13.03.

Identifier	Data type	Explanation
`MyList`	`ARRAY[0:6] OF INTEGER`	Data structure (1D array) to store seven numbers
`MaxIndex`	`INTEGER`	The upper bound of the array
`n`	`INTEGER`	The number of pairs of elements to compare in each pass
`i`	`INTEGER`	Counter for outer loop
`j`	`INTEGER`	Counter for inner loop
`Temp`	`INTEGER`	Variable for temporary storage while swapping values

Table 13.03 Identifier table for bubble sort algorithm

The algorithm in pseudocode is:

```
n ← MaxIndex – 1
FOR i ← 0 TO MaxIndex – 1
    FOR j ← 0 TO n
        IF MyList[j] > MyList[j + 1]
          THEN
            Temp ← MyList[j]
            MyList[j] ← MyList[j + 1]
            MyList[j + 1] ← Temp
        ENDIF
    NEXT j
    n ← n – 1 // this means the next time round the inner loop, we don't
              // look at the values already in the correct positions.
NEXT i
```

The values to be sorted may already be in the correct order before the outer loop has been through all its iterations. Look at the list of values in Figure 13.05. It is only slightly different from the first list we sorted.

Original list	After pass 1	After pass 2	After pass 3	After pass 4	After pass 5	After pass 6
5	5	5	5	5	5	5
34	34	7	7	7	7	7
98	7	34	19	19	19	19

7	41	19	25	25	25	25
41	19	25	34	34	34	34
19	25	41	41	41	41	41
25	98	98	98	98	98	98

Figure 13.05 States of the list after each pass

After the third pass the values are all in the correct order but our algorithm will carry on with three further passes through the array. This means we are making comparisons when no further comparisons need to be made.

If we have gone through the whole of the inner loop (one pass) without swapping any values, we know that the array elements must be in the correct order. We can therefore replace the outer loop with a conditional loop.

We can use a variable `NoMoreSwaps` to store whether or not a swap has taken place during the current pass. We initialise the variable `NoMoreSwaps` to `TRUE`. When we swap a pair of values we set `NoMoreSwaps` to `FALSE`. At the end of the pass through the array we can check whether a swap has taken place.

The identifier table for this improved algorithm is shown in Table 13.04.

Identifier	Data type	Explanation
`MyList`	`ARRAY[0:6] OF INTEGER`	Data structure (1D array) to store seven numbers
`MaxIndex`	`INTEGER`	The upper bound of the array
`n`	`INTEGER`	The number of pairs of elements to compare in each pass
`NoMoreSwaps`	`BOOLEAN`	`TRUE` when no swaps have occurred in current pass `FALSE` when a swap has occurred
`j`	`INTEGER`	Counter for inner loop
`Temp`	`INTEGER`	Variable for temporary storage while swapping values

Table 13.04 Identifier table for improved bubble sort algorithm

This improved algorithm in pseudocode is:

```
n ← MaxIndex – 1
REPEAT
    NoMoreSwaps ← TRUE
    FOR j ← 0 TO n
        IF MyList[j] > MyList[j + 1]
          THEN
            Temp ← MyList[j]
            MyList[j] ← MyList[j + 1]
            MyList[j + 1] ← Temp
            NoMoreSwaps ← FALSE
        ENDIF
    NEXT j
    n ← n – 1
UNTIL NoMoreSwaps = TRUE
```

Discussion Point:

What happens if the array elements are already in the correct order?

TASK 13.05

Rewrite the algorithm in Worked Example 13.03 to sort the array elements into descending order.

13.05 Two-dimensional arrays

When we write a table of data (a matrix) on a piece of paper and want to refer to individual elements of the table, the convention is to give the row number first and then the column number. When declaring a 2D array, the number of rows is given first, then the number of columns. Again we have lower and upper bounds for each dimension.

In pseudocode, a 2D array declaration is written as:

```
DECLARE <identifier> : ARRAY[<lBound1>:<uBound1>,
<lBound2>:<uBound2>] OF <dataType>
```

The array elements in a 2D array can be numbered from 0. Sometimes it is more intuitive to use rows from row 1 and columns from column 1, as shown with the board game in Worked Example 13.05.

To declare a 2D array to represent a game board of six rows and seven columns, the pseudocode statement is:

```
Board : ARRAY[1:6,1:7] OF INTEGER
```

Accessing 2D arrays

A specific element in a table is accessed using an index pair. In pseudocode this is written as:

```
<arrayIdentifier>[x,y]
```

Pseudocode example:

```
Board[3,4] ← 0 // sets the element in row 3 and column 4 to zero
```

When we want to access each element of a 1D array, we use a loop to access each element in turn. When working with a 2D array, we need a loop to access each row. Within each row we need to access each column. This means we use a loop within a loop (nested loops).

In structured English our algorithm is:

```
For each row
   For each column
      Assign the initial value to the element at the current position
```

WORKED EXAMPLE 13.04

Working with two-dimensional arrays and nested loops

Using pseudocode, the algorithm to set each element of array `ThisTable` to zero is:

```
FOR Row ← 0 TO MaxRowIndex
    FOR Column ← 0 TO MaxColumnIndex
        ThisTable[Row, Column] ← 0
    NEXT Column
NEXT Row
```

We need the identifiers shown in Table 13.05.

Identifier	Data type	Explanation
`ThisTable`	`ARRAY[0:3, 0:5] OF INTEGER`	Table data structure (2D array) to store values
`MaxRowIndex`	`INTEGER`	The upper bound of the row index
`MaxColumnIndex`	`INTEGER`	The upper bound of the column index
`Row`	`INTEGER`	Counter for the row index
`Column`	`INTEGER`	Counter for the column index

Table 13.05 Identifier table for working with a table

When we want to output the contents of a 2D array, we again need nested loops. We want to output all the values in one row of the array on the same line. At the end of the row, we want to output a new line.

```
FOR Row ← 0 TO MaxRowIndex
    FOR Column ← 0 TO MaxColumnIndex
        OUTPUT ThisTable[Row, Column] // stay on same line
    NEXT Column
    OUTPUT Newline      // move to next line for next row
NEXT Row
```

TASK 13.06

1. Declare a 2D array to store the board data for the game Noughts and Crosses. The empty squares of the board are to be represented by a space. Player A's counters are to be represented by "O". Player B's counters are to be represented by "X".
2. Initialise the array to start with each square being empty.
3. Write a statement to represent player A placing their counter in the top left square.
4. Write a statement to represent player B placing their counter in the middle square.

WORKED EXAMPLE 13.05

Creating a program to play Connect 4

Connect 4 is a game played by two players. In the commercial version shown in Figure 13.06, one player uses red tokens and the other uses black. Each player has 21 tokens. The game board is a vertical grid of six rows and seven columns.

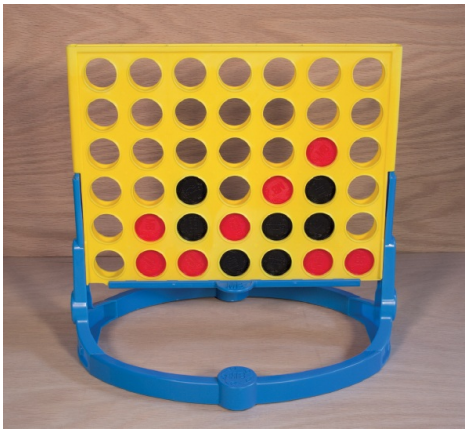


Figure 13.06 A Connect 4 board

Columns get filled with tokens from the bottom. The players take it in turns to choose a column that is not full and drop a token into this column. The token will occupy the lowest empty position in the chosen column. The winner is the player who is the first to connect four of their own tokens in a horizontal, vertical or diagonal line. If all tokens have been used and neither player has connected four tokens, the game ends in a draw.

If we want to write a program to play this game on a computer, we need to work out the steps required to 'solve the problem', that means to let players take their turn in placing tokens and checking for a winner. We will designate our players (and their tokens) by 'O' and 'X'. The game board will be represented by a 2D array. To simplify the problem, the winner is the player who is the first to connect four of their tokens horizontally or vertically.

Our first attempt in structured English is:

```
Initialise board
Set up game
Display board
While game not finished
   Player makes a move
   Display board
   Check if game finished
   If game not finished, swap player
```

The top-level pseudocode version using modules is:

```
01 CALL InitialiseBoard
02 CALL SetUpGame
03 CALL OutputBoard
04 WHILE GameFinished = FALSE DO
05     CALL PlayerMakesMove
06     CALL OutputBoard
07     CALL CheckGameFinished
08     IF GameFinished = FALSE
09       THEN
10         CALL SwapThisPlayer
11     ENDIF
12 ENDWHILE
```

Note that Steps 03 and 06 are the same. This means that we can save ourselves some effort. We only need to define this module once, but can call it from more than one place. This is one of the advantages of using modules.

The identifier table for the program is shown in Table 13.06.

Identifier	Data type	Explanation
`Board`	`ARRAY[1:6,1:7] OF CHAR`	2D array to represent the board
`InitialiseBoard`		Procedure to initialise the board to all blanks
`SetUpGame`		Procedure to set initial values for `GameFinished` and `ThisPlayer`
`GameFinished`	`BOOLEAN`	FALSE if the game is not finished TRUE if the board is full or a player has won
`ThisPlayer`	`CHAR`	`'O'` when it is Player O's turn `'X'` when it is Player X's turn
`OutputBoard`		Procedure to output the current contents of the board
`PlayerMakesMove`		Procedure to place the current player's token into the chosen board location
`CheckGameFinished`		Procedure to check if the token just placed makes the current player a winner or board is full
`SwapThisPlayer`		Procedure to change player's turn

Table 13.06 Initial identifier table for Connect 4 game

Now we can refine each procedure (module). This is likely to add some more identifiers to our identifier table. The additional entries required are shown after each procedure.

```
PROCEDURE InitialiseBoard
```

```
    FOR Row ← 1 TO 6
        FOR Column ← 1 TO 7
            Board[Row, Column] ← BLANK // use a suitable value for blank
        NEXT Column
    NEXT Row
ENDPROCEDURE
```

Identifier	Data type	Explanation
Row	INTEGER	Loop counter for the rows
Column	INTEGER	Loop counter for the columns
BLANK	CHAR	A value that represents an empty board location

Table 13.07 Additional identifiers for the `InitialiseBoard` procedure

```
PROCEDURE SetUpGame
    ThisPlayer ← 'O' // Player O always starts
    GameFinished ← FALSE
ENDPROCEDURE
PROCEDURE OutputBoard
    FOR Row ← 6 DOWNTO 1
        FOR Column ← 1 TO 7
            OUTPUT Board[Row, Column] // don't move to next line
        NEXT Column
        OUTPUT Newline // move to next line
    NEXT Row
ENDPROCEDURE
PROCEDURE PlayerMakesMove
    ValidColumn ← PlayerChoosesColumn // a module returns column number
    ValidRow ← FindFreeRow // a module returns row number
    Board[ValidRow, ValidColumn] ← ThisPlayer
ENDPROCEDURE
```

Identifier	Data type	Explanation
ValidColumn	INTEGER	The column number the player has chosen
PlayerChoosesColumn	INTEGER	Function to get the current player's valid choice of column
ValidRow	INTEGER	The row number that represents the first free location in the chosen column
FindFreeRow	INTEGER	Function to find the next free location in the chosen column

Table 13.08 Additional identifiers for the `PlayerMakesMove` procedure

```
FUNCTION PlayerChoosesColumn RETURNS INTEGER// returns a valid column number
    OUTPUT "Player ", ThisPlayer, "'s turn."
    REPEAT
        OUTPUT "Enter a valid column number: "
        INPUT ColumnNumber
    UNTIL ColumnNumberValid = TRUE // check whether the column number is valid
    RETURN ColumnNumber
ENDFUNCTION
```


Identifier	Data type	Explanation
ColumnNumber	INTEGER	The column number chosen by the current player
ColumnNumberValid	BOOLEAN	Function to check whether the chosen column is valid

Table 13.09 Additional identifiers for PlayerChoosesColumn function

Note that we need to define the function ColumnNumberValid. A column is valid if it is within the range 1 to 7 inclusive and there is still at least one empty location in that column.

```
FUNCTION ColumnNumberValid RETURNS BOOLEAN
  // returns whether or not the column number is valid
    Valid ← FALSE
    IF ColumnNumber >= 1 AND ColumnNumber <= 7
      THEN
        IF Board[6, ColumnNumber] = BLANK // at least 1 empty space in column
          THEN
            Valid ← TRUE
        ENDIF
    ENDIF
    RETURN Valid
ENDFUNCTION
```

Identifier	Data type	Explanation
Valid	BOOLEAN	FALSE if column number is not valid TRUE if column number is valid

Table 13.10 Additional identifier for the ColumnNumberValid function

```
FUNCTION FindFreeRow RETURNS INTEGER
  // returns the next free position
    ThisRow ← 1
    WHILE Board[ThisRow, ValidColumn] <> BLANK DO // find first empty cell
        ThisRow ← ThisRow + 1
    ENDWHILE
    RETURN ThisRow
ENDFUNCTION
```

Identifier	Data type	Explanation
ThisRow	INTEGER	Points to the next row to be checked

Table 13.11 Additional identifier for the FindFreeRow function

```
PROCEDURE CheckGameFinished
    WinnerFound ← FALSE
    CALL CheckIfPlayerHasWon
    IF WinnerFound = TRUE
      THEN
        GameFinished ← TRUE
        OUTPUT ThisPlayer " is the winner"
      ELSE
        CALL CheckForFullBoard
    ENDIF
ENDPROCEDURE
```

Note that the `CheckGameFinished` procedure uses two further procedures that we need to define.

Identifier	Data type	Explanation
`WinnerFound`	`BOOLEAN`	`FALSE` if no winning line `TRUE` if a winning line is found
`CheckIfPlayerHasWon`		Procedure to check if there is a winning line
`CheckVerticalLineInValidColumn`		Procedure to check if there is a winning vertical line in the column the last token was placed in
`CheckForFullBoard`		Procedure to check whether the board is full

Table 13.12 Additional identifiers for the `CheckGameFinished` procedure

```
PROCEDURE CheckIfPlayerHasWon
    WinnerFound ← False
    CALL CheckHorizontalLine
    IF WinnerFound = FALSE
      THEN
        CALL CheckVerticalLine
ENDPROCEDURE
PROCEDURE CheckHorizontalLine
    FOR i ← 1 TO 4
       IF Board[ValidRow, i] = ThisPlayer AND
         Board[ValidRow, i + 1] = ThisPlayer AND
         Board[ValidRow, i + 2] = ThisPlayer AND
         Board[ValidRow, i + 3] = ThisPlayer
         THEN
           WinnerFound ← TRUE
       ENDIF
    NEXT i
ENDPROCEDURE
PROCEDURE CheckVerticalLine
    IF ValidRow = 4 OR ValidRow = 5 OR ValidRow = 6
      THEN
        IF Board[ValidRow, ValidColumn] = ThisPlayer AND
          Board[ValidRow – 1, ValidColumn] = ThisPlayer AND
          Board[ValidRow – 2, ValidColumn] = ThisPlayer AND
          Board[ValidRow – 3, ValidColumn] = ThisPlayer
          THEN
            WinnerFound ← TRUE
        ENDIF
    ENDIF
ENDPROCEDURE
PROCEDURE CheckForFullBoard
    BlankFound ← FALSE
    ThisRow ← 0
    REPEAT
        ThisColumn ← 0
        ThisRow ← ThisRow + 1
```

```
        REPEAT
            ThisColumn ← ThisColumn + 1
            IF Board[ThisRow, ThisColumn] = BLANK
              THEN
                BlankFound ← TRUE
            ENDIF
        UNTIL ThisColumn = 7 OR BlankFound = TRUE
    UNTIL ThisRow = 6 OR BlankFound = TRUE
    IF BlankFound = FALSE
      THEN
        OUTPUT "It is a draw"
        GameFinished ← TRUE
    ENDIF
ENDPROCEDURE
```

Identifier	Data type	Explanation
BlankFound	BOOLEAN	FALSE if no blank location found on the board TRUE if a blank location found on the board
ThisRow	INTEGER	Loop counter for rows
ThisColumn	INTEGER	Loop counter for columns

Table 13.13 Additional identifiers for the CheckForFullBoard procedure

```
PROCEDURE SwapThisPlayer
    IF ThisPlayer = 'O'
      THEN
        ThisPlayer ← 'X'
      ELSE
        ThisPlayer ← 'O'
    ENDIF
ENDPROCEDURE
```

We can also use arrays of records. Using the Person record type from Section 13.02, we can declare an array of that type for 100 person records:

```
DECLARE Person : ARRAY[1:100] OF PersonType
```

We can then access an individual's data. For example the first person's name in the array is set as follows:

```
Person[1].Name ← "Fred"
OUTPUT Person[1].Name
```

This is particularly useful when we have several people's data to work with and do not want to use a separate 1D array for each field.

TASK 13.07

1. Declare an array of BookType (see Task 13.02) for 200 books.
2. Set the first book's details to the values given in Task 13.02.

13.06 Text files

Data need to be stored permanently. One approach is to use a file. For example, any data held in an array while your program is executing will be lost when the program stops. You can save the data out to a file and read it back in when your program requires it on subsequent executions.

A text file consists of a sequence of characters formatted into lines. Each line is terminated by an end-of-line marker. The text file is terminated by an end-of-file marker.

TIP

You can check the contents of a text file (or even create a text file required by a program) by using a text editor such as NotePad.

Writing to a text file

Writing to a text file usually means creating a text file.

The following pseudocode statements provide facilities for writing to a file:

```
OPENFILE <filename> FOR WRITE       // open the file for writing
WRITEFILE <filename>, <stringValue>  // write a line of text to the file
CLOSEFILE <filename>               // close file
```

Reading from a text file

An existing file can be read by a program. The following pseudocode statements provide facilities for reading from a file:

```
OPENFILE <filename> FOR READ          // open file for reading
READFILE <filename>, <stringVariable>   // read a line of text from the file
CLOSEFILE <filename>                 // close file
```

Appending to a text file

Sometimes we may wish to add data to an existing file rather than create a new file. This can be done in Append mode. It adds the new data to the end of the existing file.

The following pseudocode statements provide facilities for appending to a file:

```
OPENFILE <filename> FOR APPEND     // open file for append
WRITEFILE <filename>, <stringValue> // write a line of text to the file
CLOSEFILE <filename>              // close file
```

The end-of-file (EOF) marker

If we want to read a file from beginning to end, we can use a conditional loop. Text files contain a special marker at the end of the file that we can test for. Testing for this special end-of-file marker is a standard function in many programming languages. Every time this function is called it will test for this marker. The function will return FALSE if the end of the file is not yet reached and will return TRUE if the end-of-file marker has been reached.

In pseudocode we call this function `EOF()`. We can use the construct `REPEAT...UNTIL EOF()`. If it is possible that the file contains no data, it is better to use the construct `WHILE NOT EOF()`.

For example, the following pseudocode statements read a text file and output its contents:

```
OPENFILE "Test.txt" FOR READ
WHILE NOT EOF("Test.txt") DO
    READFILE "Test.txt", TextString
    OUTPUT TextString
ENDWHILE
CLOSEFILE "Test.txt"
```

TASK 13.08

1. Write pseudocode to save the array data from Task 13.06 to a text file.
2. Write pseudocode to read the values stored in the text file back into the board array.

13.07 Abstract Data Types (ADTs)

An **Abstract Data Type** is a collection of data and a set of associated operations:

- create a new instance of the data structure
- find an element in the data structure
- insert a new element into the data structure
- delete an element from the data structure
- access all elements stored in the data structure in a systematic manner.

The remainder of this chapter describes the following ADTs: stack, queue and linked list. It also demonstrates how they can be implemented from arrays.

In the following ADTs data items are represented as a single character, but this would normally be a set of data, possibly stored as fields in a record.

13.08 Stacks

What are the features of a stack in the real world? To make a stack, we pile items on top of each other. The item that is accessible is the one on top of the stack. If we try to find an item in the stack and take it out, we are likely to cause the pile of items to collapse.

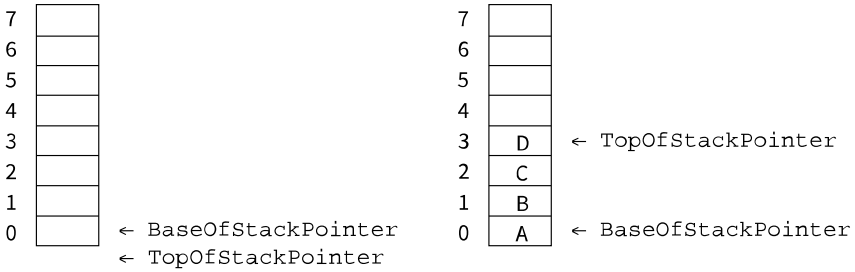


Figure 13.07 An empty stack (left) and a stack with four items pushed (right)

Figure 13.07 shows how we can represent a stack when we have added four items in this order: A, B, C, D. Note that the slots are shown numbered from the bottom as this feels more natural.

The `BaseOfStackPointer` will always point to the first slot in the stack. The `TopOfStackPointer` will point to the last element pushed (added) onto the stack. When an element is popped (removed) from the stack, the `TopOfStackPointer` will decrease to point to the element now at the top of the stack. When the stack is empty, `TopOfStackPointer` will have the value –1.

To implement this stack using a 1D array, we write:

```
DECLARE Stack:ARRAY[0 : 7] OF CHAR
```

TASK 13.09

1. Draw a diagram to show the contents of the stack shown in Figure 13.07 after “E” has been pushed onto the stack.
2. Draw a diagram to show the contents of the stack shown in Figure 13.07 after one item has been popped off the stack.

13.09 Queues

What are the features of a queue in the real world? When people form a queue, they join the queue at the end. People leave the queue from the front of the queue. If it is an orderly queue, no-one pushes in between and people don't leave the queue from any other position.

Figure 13.08 shows how we can represent a queue when five items have joined the queue in this order: A, B, C, D, E.

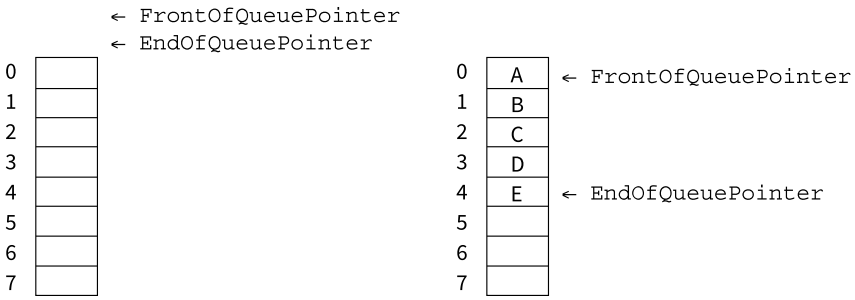


Figure 13.08 An empty queue (left) and a queue after 5 items have joined (right)

To implement a queue using an array, we can assume that the front of the queue is at position 0. When the queue is empty, the `EndOfQueuePointer` will have the value –1. When one value joins the queue, the `EndOfQueuePointer` will be incremented before adding the value to the array element where the pointer is pointing to. When the item at the front of the queue leaves, we need to move all the other items one slot forward and adjust `EndOfQueuePointer`.

TASK 13.10

1. Draw a diagram to show the contents of the queue after "F" has joined the non-empty queue shown in Figure 13.08.
2. Draw a diagram to show the contents of the queue after one item has left the non-empty queue shown in Figure 13.08.

This method involves a lot of moving of data. A more efficient way to make use of the slots is the concept of a 'circular' queue. Pointers show where the front and end of the queue are. Eventually the queue will 'wrap around' to the beginning. Figure 13.09 shows a circular queue after 11 items have joined and five items have left the queue.

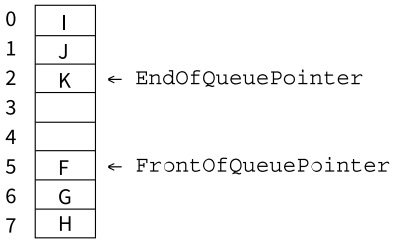


Figure 13.09 A circular queue

13.10 Linked lists

In Section 13.03 we used an array as a linear list. In a linear list, the list items are stored in consecutive locations. This is not always appropriate. Another method is to store an individual list item in whatever location is available and link the individual item into an ordered sequence using pointers.

An element of a linked list is called a node. A **node** can consist of several data items and a **pointer**, which is a variable that stores the address of the node it points to.

A pointer that does not point at anything is called a **null pointer.** It is usually represented by ⌀. A variable that stores the address of the first element is called a **start pointer.**

In Figure 13.10, the data value in the node box represents the key field of that node. There are likely to be many data items associated with each node. The arrows represent the pointers. It does not show at which address a node is stored, so the diagram does not give the value of the pointer, only where it conceptually links to.

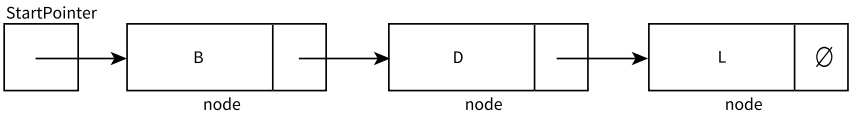


Figure 13.10 Conceptual diagram of a linked list

Figure 13.11 shows how a new node, A, is inserted at the beginning of the list. The content of `StartPointer` is copied into the new node's pointer field and `StartPointer` is set to point to the new node, A.

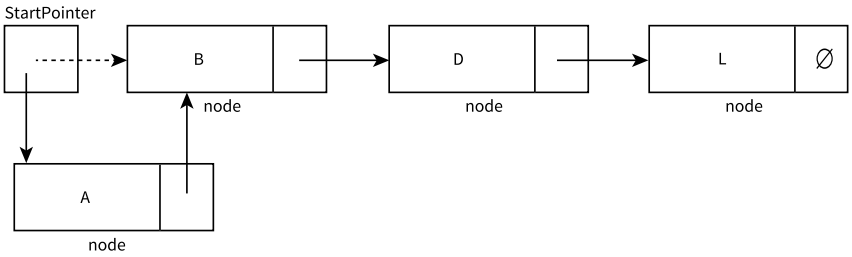


Figure 13.11 Conceptual diagram of adding a new node to the beginning of a linked list

In Figure 13.12, a new node, P, is inserted at the end of the list. The pointer field of node L points to the new node, P. The pointer field of the new node, P, contains the null pointer.

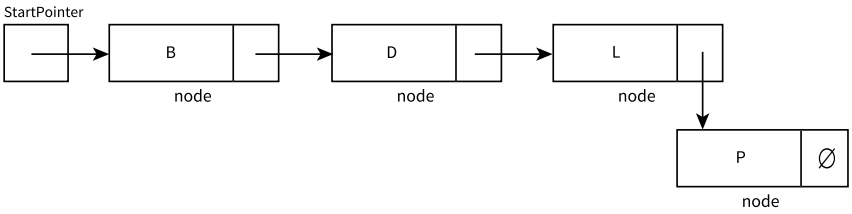


Figure 13.12 Conceptual diagram of adding a new node to the end of a linked list

To delete the first node in the list (see Figure 13.13), we copy the pointer field of the node to be deleted into `StartPointer`.

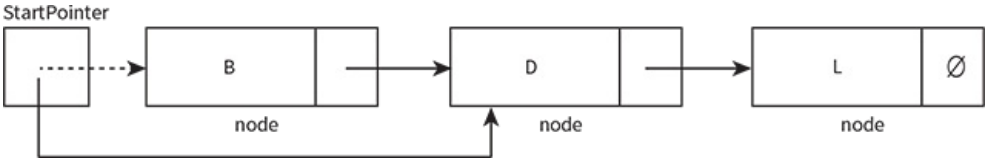


Figure 13.13 Deleting the first node in a linked list

To delete the last node in the list (see Figure 13.14), we set the pointer field for the previous node to the null pointer.

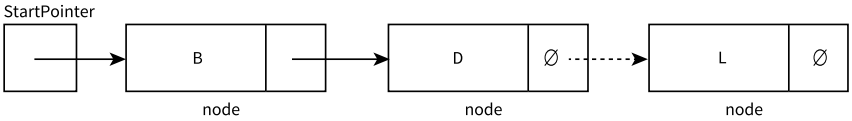


Figure 13.14 Conceptual diagram of deleting the last node of a linked list

Sometimes the nodes are linked together in order of key field value to produce an ordered linked list. This means a new node may need to be inserted or deleted from between two existing nodes.

To insert a new node, C, between existing nodes, B and D (see Figure 13.15), we copy the pointer field of node B into the pointer field of the new node, C. We change the pointer field of node B to point to the new node, C.

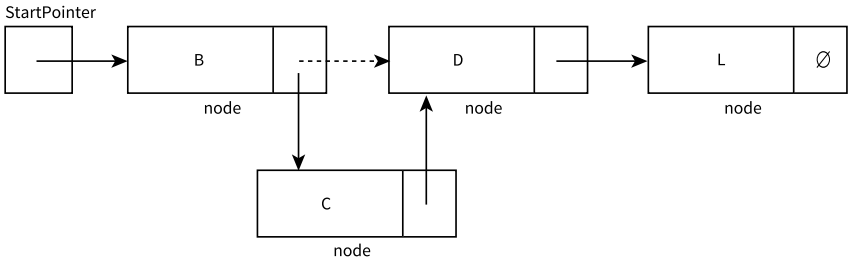


Figure 13.15 Conceptual diagram of adding a new node into a linked list

To delete a node, D, within the list (see Figure 13.16), we copy the pointer field of the node to be deleted, D, into the pointer field of node B.

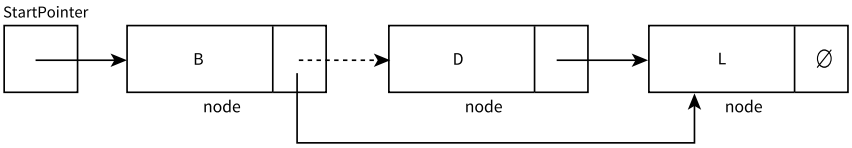


Figure 13.16 Conceptual diagram of deleting a node within a linked list

Remember that, in real applications, the data would consist of much more than a key field and one data item. This is why linked lists are preferable to linear lists. When list elements need reordering, only pointers need changing in a linked list. In a linear list, all data items would need to be moved.

Using linked lists saves time, however we need more storage space for the pointer fields.

To implement a linked list using arrays, we can use a 1D array to store the data and a 1D array to store the pointer. Reading the array values across at the same index, one row represents a node.

A value is added to the next free element of the `Data` array and pointers are adjusted to incorporate the node in the correct position within the linked list.

Figure 13.17 shows how two arrays can be used to implement the linked list from Figure 13.10.

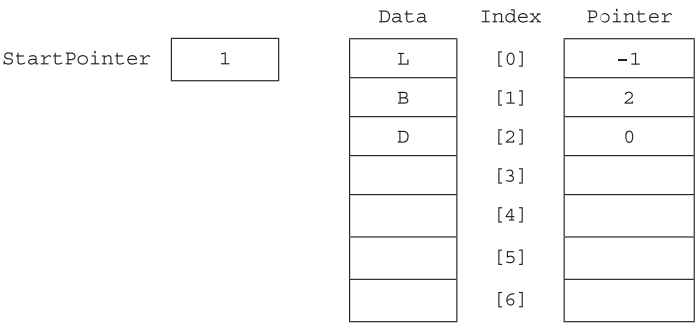


Figure 13.17 Using arrays to implement the linked list shown in Figure 13.10

Figure 13.18 shows how a new node is added to the beginning of a linked list implemented using arrays. Note that the value “A” is added at index 3 but the start pointer is adjusted to make this the new first element of the list.

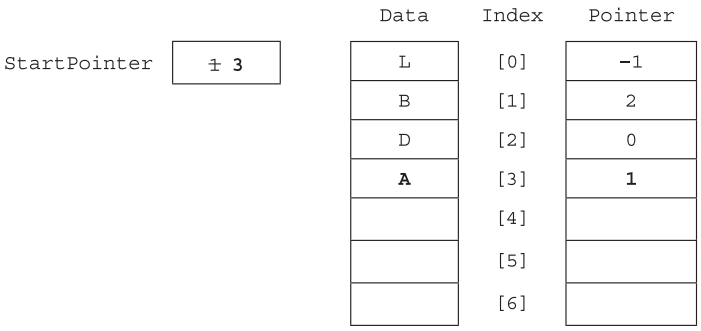


Figure 13.18 Adding a new node to the beginning of a linked list

Figure 13.19 shows how a new node is added to the end of a linked list implemented using arrays. Note that the value “P” is added at index 3. The node that previously contained the null pointer (at index 0) has its pointer adjusted to point to the new node.

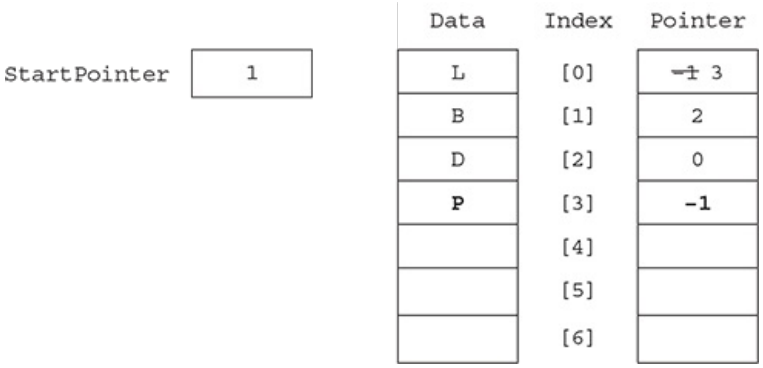


Figure 13.19 Adding a new node to the end of a linked list implemented using arrays

When deleting a node, only pointers need to be adjusted. The old data can remain in the array, but it will no longer be accessible as no pointer will point to it.

Figure 13.20 shows how the start pointer is adjusted to effectively delete the first element of the linked list. Note that the start pointer now contains the pointer value of the deleted node.

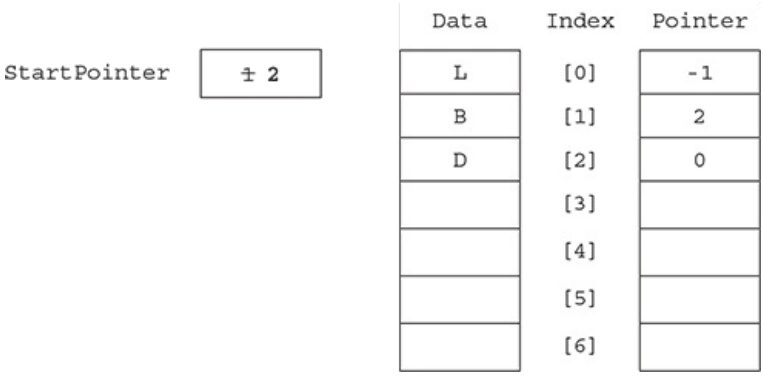


Figure 13.20 Deleting the first node in a linked list implemented using arrays

Figure 13.21 shows how the pointer value of the penultimate node of the linked list is changed to the null pointer.

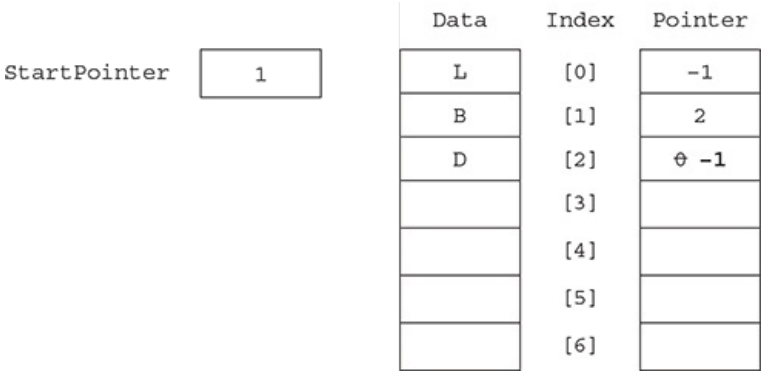


Figure 13.21 Deleting the last node of a linked list implemented using arrays

When adding a node that needs to be inserted into the list, the data is added to any free element of the Data array. The pointer of the new node is set to point to the index of the node that comes after the insertion point. Note that this is the value of the pointer of the node preceding the insertion point. The pointer of the node preceding the insertion point is set to point to the new node.

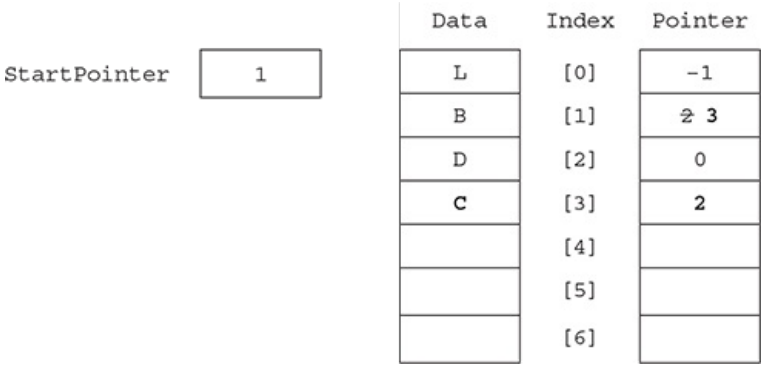


Figure 13.22 Adding a new node into a linked list implemented using arrays

Again, when deleting a node, only pointers need to be adjusted. Figure 13.23 shows how the pointer of the node to be deleted is copied into the pointer of the preceding node.

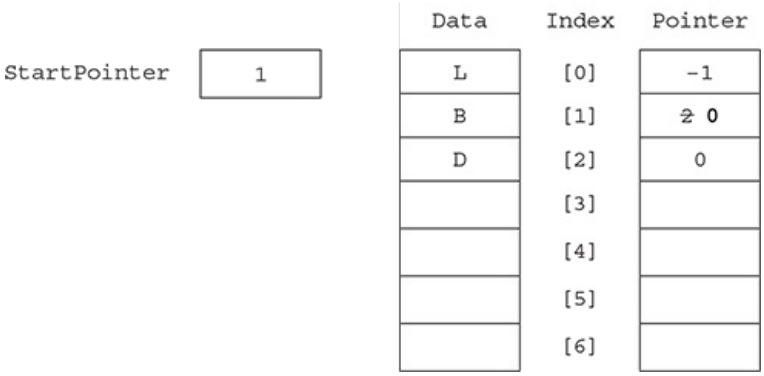


Figure 13.23 Deleting a node within a linked list implemented using arrays

Unused nodes need to be easy to find. A suitable technique is to link the unused nodes to form another linked list: the free list. Figure 13.24 shows our linked list and its free list.

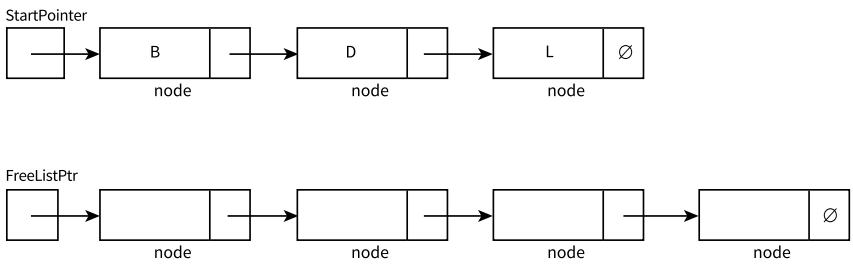


Figure 13.24 Conceptual diagram of a linked list and a free list

When an array of nodes is first initialised to work as a linked list, the linked list will be empty. So the start pointer will be the null pointer. All nodes need to be linked to form the free list. Figure 13.25 shows an example of an implementation of a linked list before any data is inserted into it.

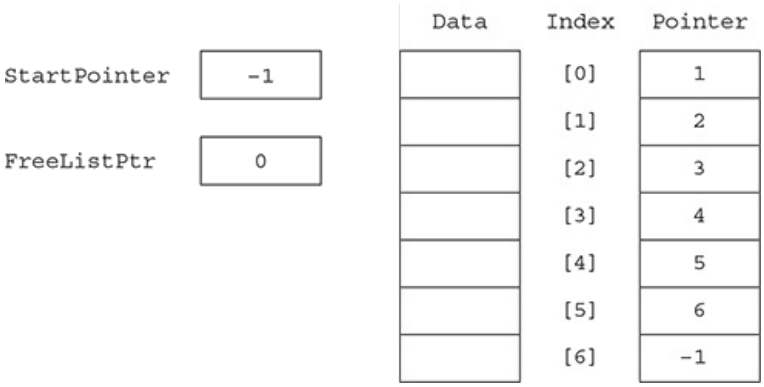


Figure 13.25 A linked list and a free list implemented using arrays

Assume “L”, “B” and “D” were added to the linked list and to be kept in alphabetical order.

Figure 13.26 shows how the values are stored in the `Data` array and the pointers of the linked list and free list adjusted.

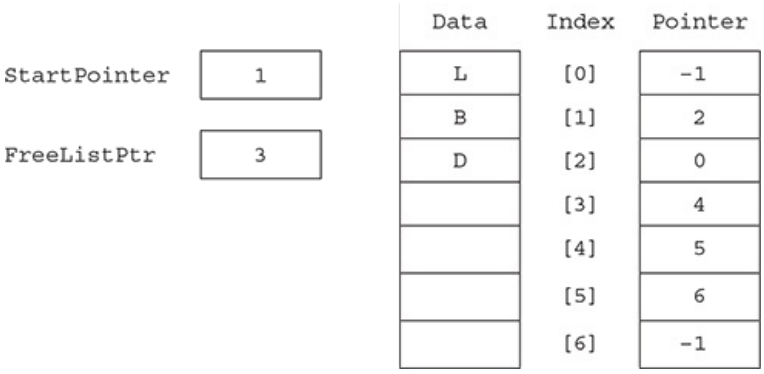


Figure 13.26 Linked list and free list implemented using arrays

If the node containing “B” is to be deleted, the array element of that node needs to be linked back into the free list. Figure 13.27 shows how this is done by adding the node to the front of the free list.

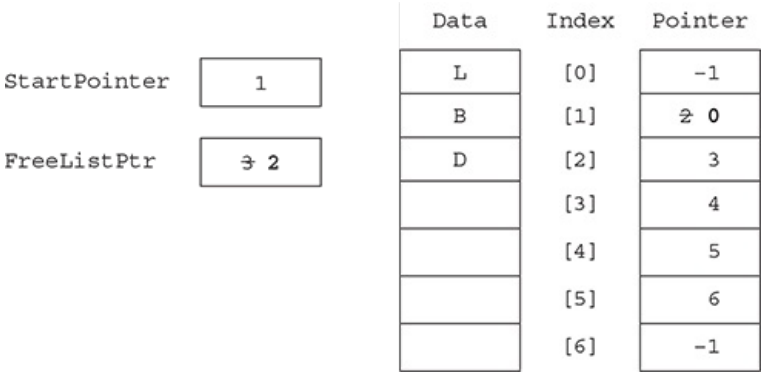
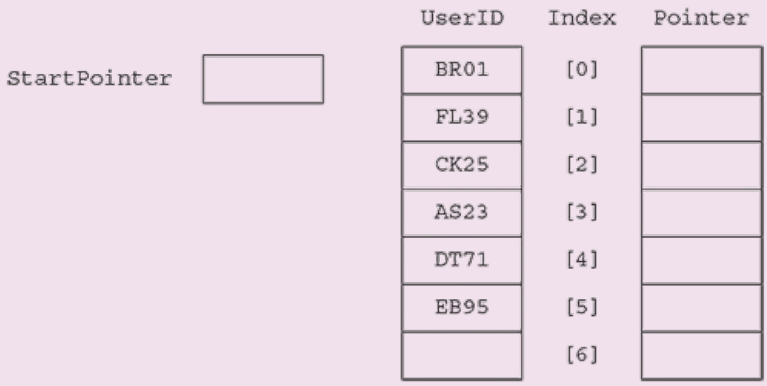


Figure 13.27 Linked list and free list implemented using arrays

TASK 13.11

A linked list is to be set up using the values in the `UserID` array shown below.



Without moving any of the contents of the `UserID` array, insert the pointer values so that the linked list is in alphabetical order.

In Section 13.02 we looked at the user-defined record type. We grouped together related data items into record data structures. To use a record variable, we first define a record type. Then we declare variables of that record type.

We can store the linked list in an array of records. One record represents a node and consists of the data and a pointer (see Figure 13.28).

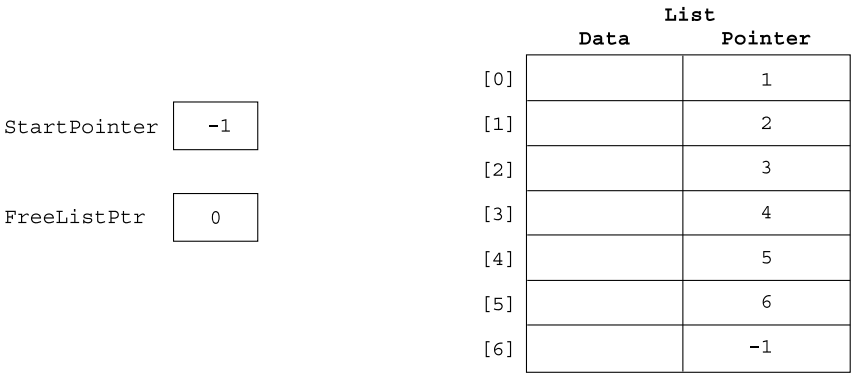


Figure 13.28 A linked list before any nodes are used

TIP

A stack can be implemented from a linked list. The start pointer is seen as the top of stack pointer and a data item is only added to the start of the linked list and a node is only removed from the start of the linked list.

TIP

A queue can be implemented from a linked list. The start pointer is seen as the front of the queue. Data items are always added to the end of the linked list and items are always removed from the start of the linked list.

Reflection Point:

What is the difference between standard data types, ADTs and user-defined data types?

Summary

- Standard data types are INTEGER, REAL, CHAR, STRING, BOOLEAN, DATE.
- A record structure holds a set of data of different data types under one identifier.
- Use the dot notation to address fields of a record.

- Arrays have dimensions with upper and lower bounds.
- Individual array elements are accessed using an index (1D arrays) or two indexes (2D arrays).
- A bubble sort algorithm compares pairs of values in a linear list and swaps them if required.
- A linear search checks each value in turn for a required value.
- Text files can be written to and read from and store data between program runs.
- Stacks, queues and linked lists are examples of Abstract Data Types (ADTs) and can be implemented using arrays.

Exam-style Questions

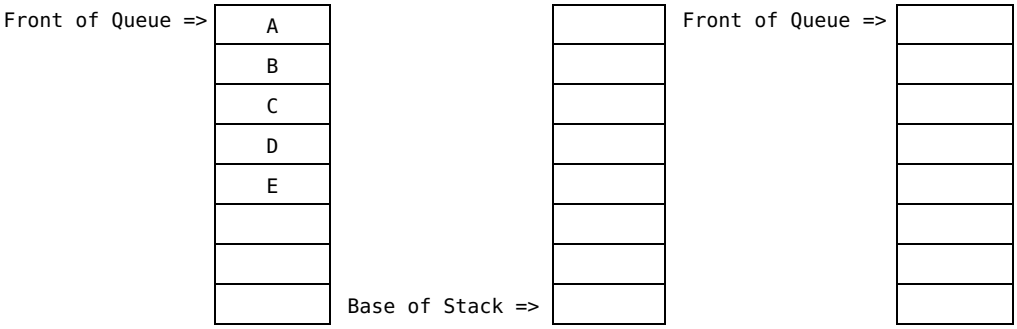
1 Complete the following variable identifier table:

Variable	Example value	Data type
`ColourCode`	***"034AB45"***	
`ProductionDate`	***2018/03/31***	
`Weight`	***67.45***	
`NumberInStock`	***98***	
`SizeCode`	***'X'***	
`Completed`	***FALSE***	

[3]

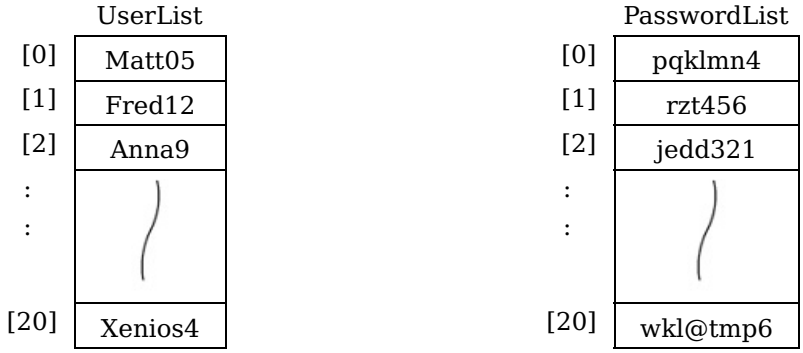
2 A stack and a queue are used to reverse the order of a set of values.

Complete the diagram:



[2]

3 Alicia uses two 1D arrays, `UserList` and `PasswordList`. For twenty users, she stores each user ID in `UserList` and the corresponding password in `PasswordList`. For example, the person with user ID `Fred12` has password `rzt456`.



Alicia wants to write an algorithm to check whether a user ID and password, entered by a user, are correct. She designs the algorithm to search `UserList` for the user ID. If the user ID is found, the password stored in `PasswordList` is to be compared to the entered password. If the passwords match, the login is successful. In all other cases, login is unsuccessful.

a Complete the identifier table. [8]

Identifier	Data type	Explanation
`UserList`		1D array to store user IDs
.....		1D array to store passwords
`MaxIndex`		Upper bound of the array
`MyUserID`		User ID entered to login

MyPassword		.....
UserIdFound		FALSE if user ID not found in UserList TRUE if .....
LoginOK		FALSE if ..... TRUE if .....
Index		Pointer to current array element

b Complete the pseudocode for Alicia's algorithm:

```
MaxIndex ← 20
INPUT MyUserID
INPUT MyPassword
UserIdFound ← FALSE
LoginOK ← ..............
Index ← –1
REPEAT
    INDEX ← ..............
    IF UserList[..............] = ..............
      THEN
        UserIdFound ← TRUE
    ENDIF
UNTIL ..............  OR ..............
IF UserIdFound = TRUE
  THEN
    IF PasswordList[..............] = ..............
      THEN
        LoginOK ← TRUE
    ..............
ENDIF
IF ..............
  THEN
    OUTPUT "Login successful"
  ELSE
    OUTPUT "User ID and/or password incorrect"
ENDIF
```

[10]

c **i** Instead of using two 1D arrays, Alicia could have used an array of records.

Write pseudocode to declare the record structure `UserRecord`. [2]

ii Write pseudocode to declare the `User` array. [2]