6 A queue Abstract Data Type (ADT) has these associated operations:

- create queue
- add item to queue
- remove item from queue

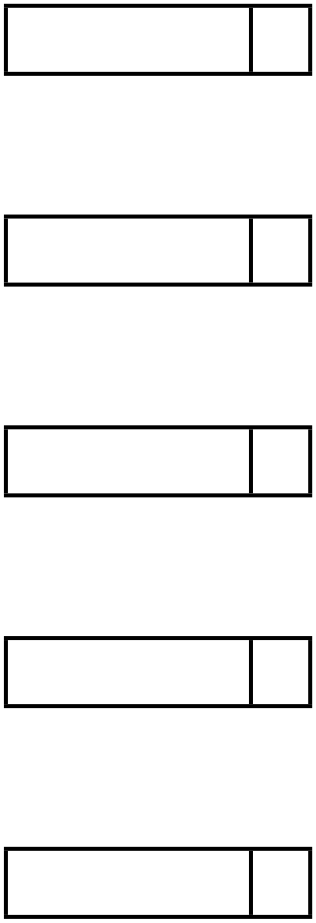
The queue ADT is to be implemented as a linked list of nodes.

Each node consists of data and a pointer to the next node.

(a) The following operations are carried out:

```
CreateQueue
AddName("Ali")
AddName("Jack")
AddName("Ben")
AddName("Ahmed")
RemoveName
AddName("Jatinder")
RemoveName
```

Add appropriate labels to the diagram to show the final state of the queue. Use the space on the left as a workspace. Show your final answer in the node shapes on the right:



[3]

(b) Using pseudocode, a record type, `Node`, is declared as follows:

```
TYPE Node
    DECLARE Name    : STRING
    DECLARE Pointer : INTEGER
ENDTYPE
```

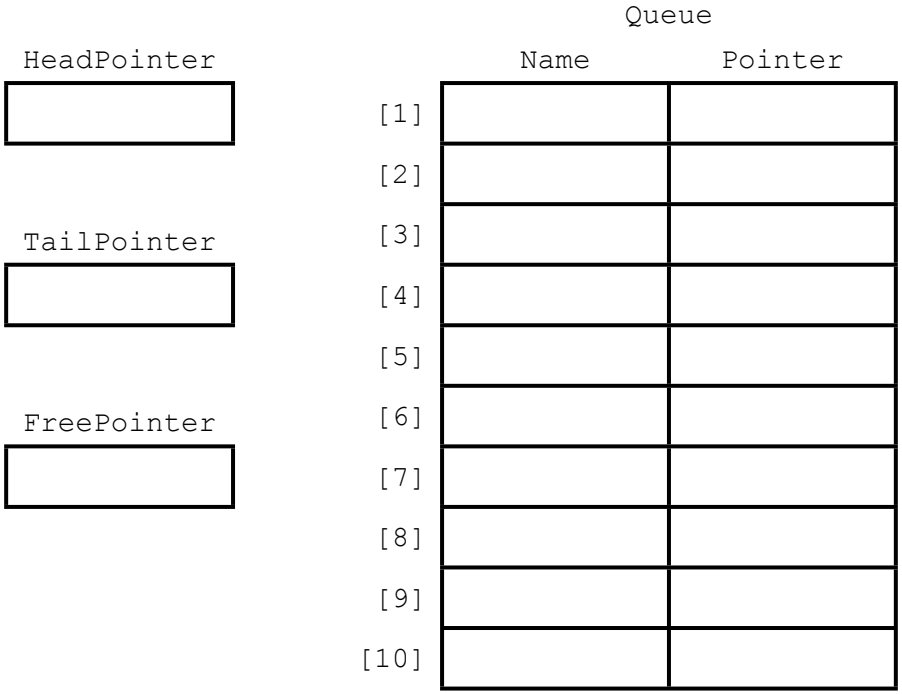
The statement

```
DECLARE Queue : ARRAY[1:10] OF Node
```

reserves space for 10 nodes in array `Queue`.

(i) The `CreateQueue` operation links all nodes and initialises the three pointers that need to be used: `HeadPointer`, `TailPointer` and `FreePointer`.

Complete the diagram to show the value of all pointers after `CreateQueue` has been executed.



[4]

(ii) The algorithm for adding a name to the queue is written, using pseudocode, as a procedure with the header:

```
PROCEDURE AddName(NewName)
```

where `NewName` is the new name to be added to the queue.

The procedure uses the variables as shown in the identifier table.

Identifier	Data type	Description
Queue	Array[1:10] OF Node	Array to store node data
NewName	STRING	Name to be added
FreePointer	INTEGER	Pointer to next free node in array
HeadPointer	INTEGER	Pointer to first node in queue
TailPointer	INTEGER	Pointer to last node in queue
CurrentPointer	INTEGER	Pointer to current node

```
PROCEDURE AddName(BYVALUE NewName : STRING)
   // Report error if no free nodes remaining
   IF FreePointer = 0
      THEN
         Report Error
   ELSE
      // new name placed in node at head of free list
      CurrentPointer ← FreePointer
      Queue[CurrentPointer].Name ← NewName
      // adjust free pointer
      FreePointer ← Queue[CurrentPointer].Pointer
      // if first name in queue then adjust head pointer
      IF HeadPointer = 0
         THEN
            HeadPointer ← CurrentPointer
      ENDIF
      // current node is new end of queue
      Queue[CurrentPointer].Pointer ← 0
      TailPointer ← CurrentPointer
   ENDIF
ENDPROCEDURE
```

Complete the **pseudocode** for the procedure `RemoveName`. Use the variables listed in the identifier table.

```
PROCEDURE RemoveName()
   // Report error if Queue is empty
.....
.....
.....
.....
   OUTPUT Queue[.....].Name
   // current node is head of queue
.....
   // update head pointer
.....
   // if only one element in queue then update tail pointer
.....
.....
.....
.....
   // link released node to free list
.....
.....
.....
ENDPROCEDURE
```

[6]

1 A linked list abstract data type (ADT) is to be used to store and organise surnames.

This will be implemented with a 1D array and a start pointer. Elements of the array consist of a user-defined type. The user-defined type consists of a data value and a link pointer.

Identifier	Data type	Description
`LinkedList`	`RECORD`	User-defined type
`Surname`	`STRING`	Surname string
`Ptr`	`INTEGER`	Link pointers for the linked list

(a) (i) Write **pseudocode** to declare the type `LinkedList`.

.....

.....

.....

.....[3]

(ii) The 1D array is implemented with an array `SurnameList` of type `LinkedList`.

Write the **pseudocode** declaration statement for `SurnameList`. The lower and upper bounds of the array are 1 and 5000 respectively.

.....[2]

(b) The following surnames are organised as a linked list with a start pointer `StartPtr`.

`StartPtr: 3`

	1	**2**	**3**	**4**	**5**	**6**	...	**5000**
Surname	Liu	Yang	Chan	Wu	Zhao	Huang	...	
Ptr	4	5	6	2	0	1	...	

State the value of the following:

(i) `SurnameList[4].Surname` .....[1]

(ii) `SurnameList[StartPtr].Ptr` .....[1]

(c) Pseudocode is to be written to search the linked list for a surname input by the user.

Identifier	Data type	Description
ThisSurname	STRING	The surname to search for
Current	INTEGER	Index to array SurnameList
StartPtr	INTEGER	Index to array SurnameList. Points to the element at the start of the linked list

(i) Study the pseudocode in **part (c)(ii)**.

Complete the table above by adding the missing identifier details. [2]

(ii) Complete the pseudocode.

```
01 Current ← .....
02 IF Current = 0
03    THEN
04       OUTPUT .....
05    ELSE
06       IsFound ← .....
07       INPUT ThisSurname
08       REPEAT
09          IF ..... = ThisSurname
10             THEN
11                IsFound ← TRUE
12                OUTPUT "Surname found at position ", Current
13             ELSE
14                // move to the next list item
15                .....
16          ENDIF
17       UNTIL IsFound = TRUE OR .....
18       IF IsFound = FALSE
19          THEN
20             OUTPUT "Not Found"
21       ENDIF
22 ENDIF
```

[6]

2 An ordered binary tree Abstract Data Type (ADT) has these associated operations:

- create tree
- add new item to tree
- traverse tree

The binary tree ADT is to be implemented as a linked list of nodes.

Each node consists of data, a left pointer and a right pointer.

(a) A null pointer is shown as Ø.

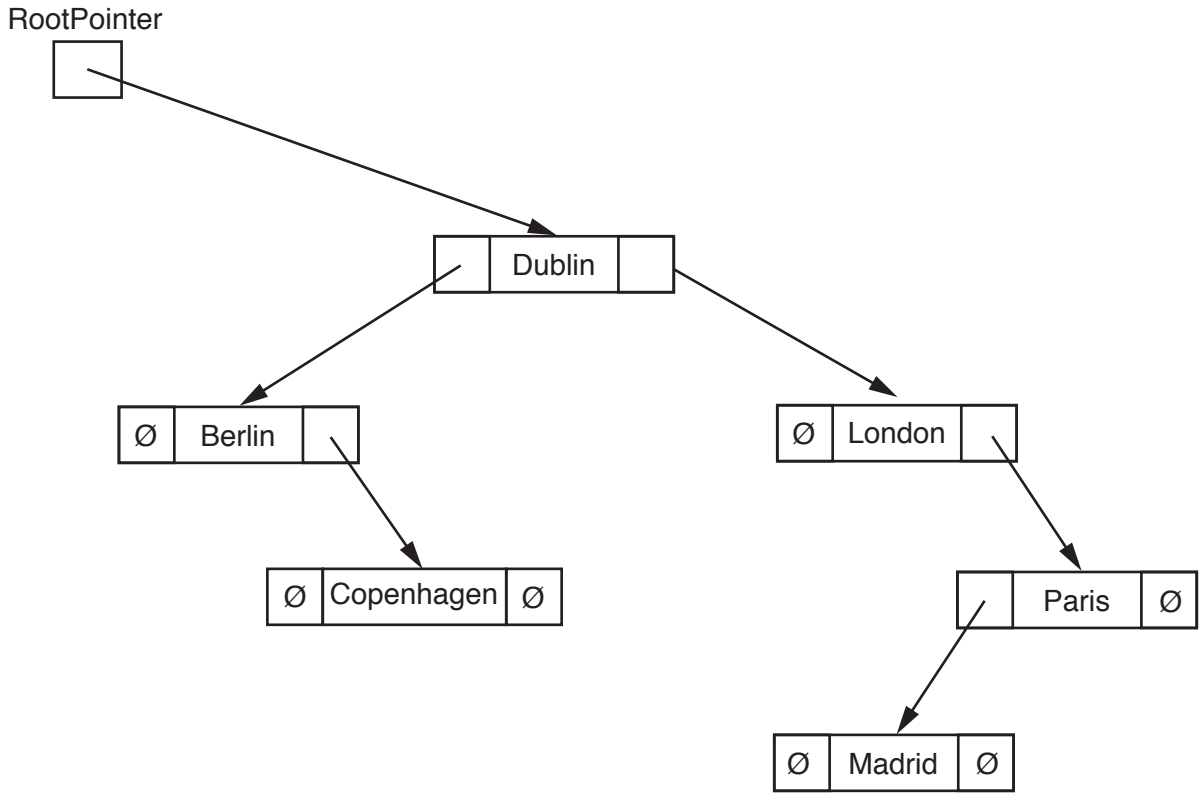
Explain the meaning of the term **null pointer**.

.....

.....[1]

(b) The following diagram shows an ordered binary tree after the following data have been added:

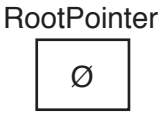
Dublin, London, Berlin, Paris, Madrid, Copenhagen



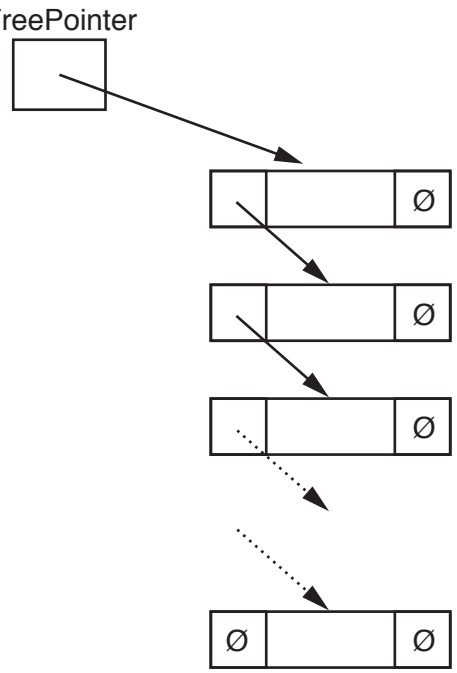
Another data item to be added is Athens.

Make the required changes to the diagram when this data item is added. [2]

(c) A tree without any nodes is represented as:



Unused nodes are linked together into a free list as shown:



The following diagram shows an array of records that stores the tree shown in **part (b)**.

(i) Add the relevant pointer values to complete the diagram.

RootPointer

0

FreePointer

	LeftPointer	Tree data	RightPointer
[0]		Dublin	
[1]		London	
[2]		Berlin	
[3]		Paris	
[4]		Madrid	
[5]		Copenhagen	
[6]		Athens	
[7]			
[8]			
[9]			

[5]

(ii) Give an appropriate numerical value to represent the null pointer for this design. Justify your answer.

.....

.....

.....

.....[2]

(d) A program is to be written to implement the tree ADT. The variables and procedures to be used are listed below:

Identifier	Data type	Description
`Node`	`RECORD`	Data structure to store node data and associated pointers.
`LeftPointer`	`INTEGER`	Stores index of start of left subtree.
`RightPointer`	`INTEGER`	Stores index of start of right subtree.
`Data`	`STRING`	Data item stored in node.
`Tree`	`ARRAY`	Array to store nodes.
`NewDataItem`	`STRING`	Stores data to be added.
`FreePointer`	`INTEGER`	Stores index of start of free list.
`RootPointer`	`INTEGER`	Stores index of root node.
`NewNodePointer`	`INTEGER`	Stores index of node to be added.
`CreateTree()`		Procedure initialises the root pointer and free pointer and links all nodes together into the free list.
`AddToTree()`		Procedure to add a new data item in the correct position in the binary tree.
`FindInsertionPoint()`		Procedure that finds the node where a new node is to be added. Procedure takes the parameter `NewDataItem` and returns two parameters: • `Index`, whose value is the index of the node where the new node is to be added • `Direction`, whose value is the direction of the pointer (“`Left`” or “`Right`”).

(i) Complete the pseudocode to create an empty tree.

```
TYPE Node
   .....
   .....
   .....
ENDTYPE
DECLARE Tree : ARRAY[0 : 9] .....
DECLARE FreePointer : INTEGER
DECLARE RootPointer : INTEGER

PROCEDURE CreateTree()
   DECLARE Index : INTEGER
   .....
   .....
   FOR Index ← 0 TO 9   // link nodes
      .....
      .....
   ENDFOR
   .....
ENDPROCEDURE
```

[7]

(ii) Complete the pseudocode to add a data item to the tree.

```
PROCEDURE AddToTree(BYVALUE NewDataItem : STRING)
// if no free node report an error
   IF FreePointer .....
      THEN
         OUTPUT("No free space left")
      ELSE // add new data item to first node in the free list
         NewNodePointer ← FreePointer
         .....
         // adjust free pointer
         FreePointer ← .....
         // clear left pointer
         Tree[NewNodePointer].LeftPointer ← .....
         // is tree currently empty ?
         IF .....
            THEN // make new node the root node
               .....
            ELSE   // find position where new node is to be added
               Index ← RootPointer
               CALL FindInsertionPoint(NewDataItem, Index, Direction)
               IF Direction = "Left"
                  THEN   // add new node on left
                     .....
                  ELSE   // add new node on right
                     .....
               ENDIF
         ENDIF
   ENDIF
ENDPROCEDURE
```

[8]

(e) The traverse tree operation outputs the data items in alphabetical order. This can be written as a recursive solution.

Complete the pseudocode for the recursive procedure `TraverseTree`.

```
PROCEDURE TraverseTree(BYVALUE Pointer : INTEGER)
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
ENDPROCEDURE
```

[5]

6 An Abstract Data Type (ADT) is used to create an unordered binary tree. The binary tree is created as an array of nodes. Each node consists of a data value and two pointers.

A record type, `Node`, is declared using pseudocode.

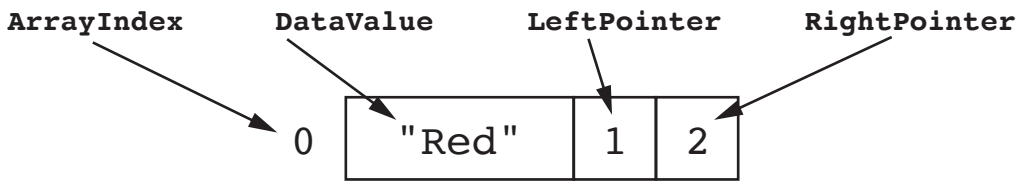
```
TYPE Node
   DECLARE DataValue : STRING
   DECLARE LeftPointer : INTEGER
   DECLARE RightPointer : INTEGER
ENDTYPE
```

The following statement declares an array `BinaryTree`.

```
   DECLARE BinaryTree : ARRAY[0:14] OF Node
```

A variable, `NextNode`, points to the next free node.

The following diagram shows a possible node.



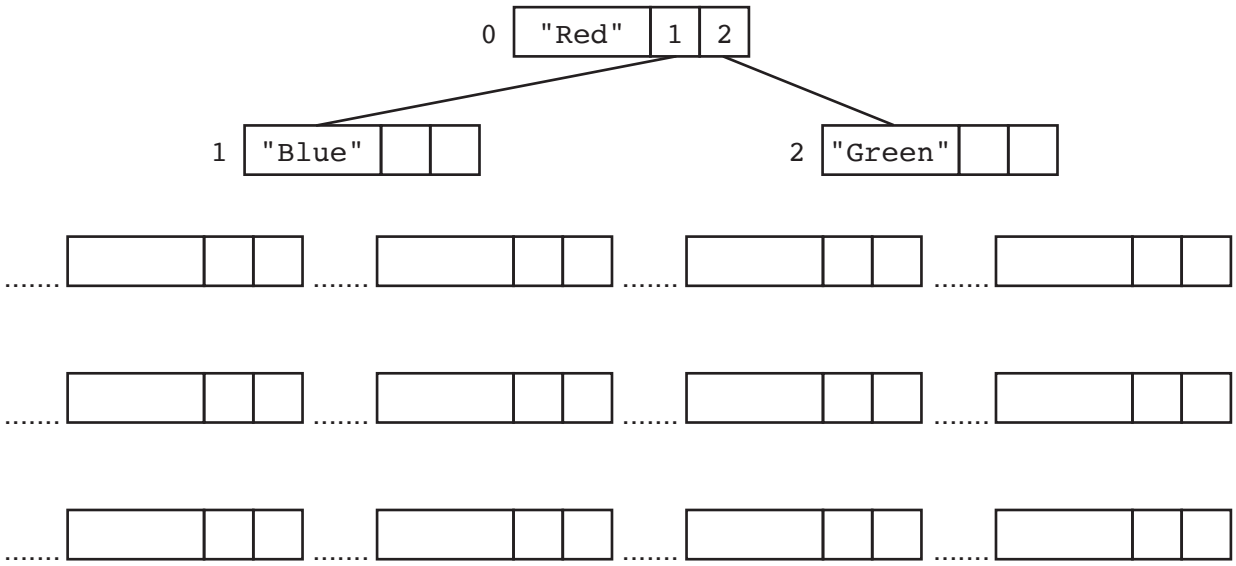
The commands in the following table create and add nodes to the binary tree.

Command	Comment
`CreateTree(NodeData)`	Sets `NextNode` to `0`. Writes `NodeData` into `DataValue` at the position `NextNode` Updates `NextNode` using `NextNode = NextNode + 1`
`AttachLeft(NodeData, ParentNode)`	Writes `NodeData` into `DataValue` of `NextNode` Sets the `LeftPointer` of node `ParentNode` to `NextNode` Updates `NextNode` using `NextNode = NextNode + 1`
`AttachRight(NodeData, ParentNode)`	Writes `NodeData` into `DataValue` of `NextNode` Sets the `RightPointer` of node `ParentNode` to `NextNode` Updates `NextNode` using `NextNode = NextNode + 1`

(a) The following commands are executed.

```
CreateTree("Red")
AttachLeft("Blue", 0)
AttachRight("Green", 0)
```

The following diagram shows the current state of the binary tree.



Write on the diagram to show the state of the binary tree after the following commands have been executed.

```
AttachRight("Black", 2)
AttachLeft("Brown", 2)
AttachLeft("Peach", 3)
AttachLeft("Yellow", 1)
AttachRight("Purple", 1)
AttachLeft("White", 6)
AttachLeft("Pink", 7)
AttachLeft("Grey", 9)
AttachRight("Orange", 9)
```

[5]

(b) A new command has been added to initialise the pointers of the binary tree to `-1` to indicate they are not in use.

A leaf is a node of the binary tree which has no children. In the case of this binary tree, a node with a `LeftPointer` of `-1` and a `RightPointer` of `-1` is a leaf.

Write a **recursive** function, in **program code**, to traverse the binary tree and output the value of `DataValue` for each leaf node.

Programming language .....

Program code

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....[8]

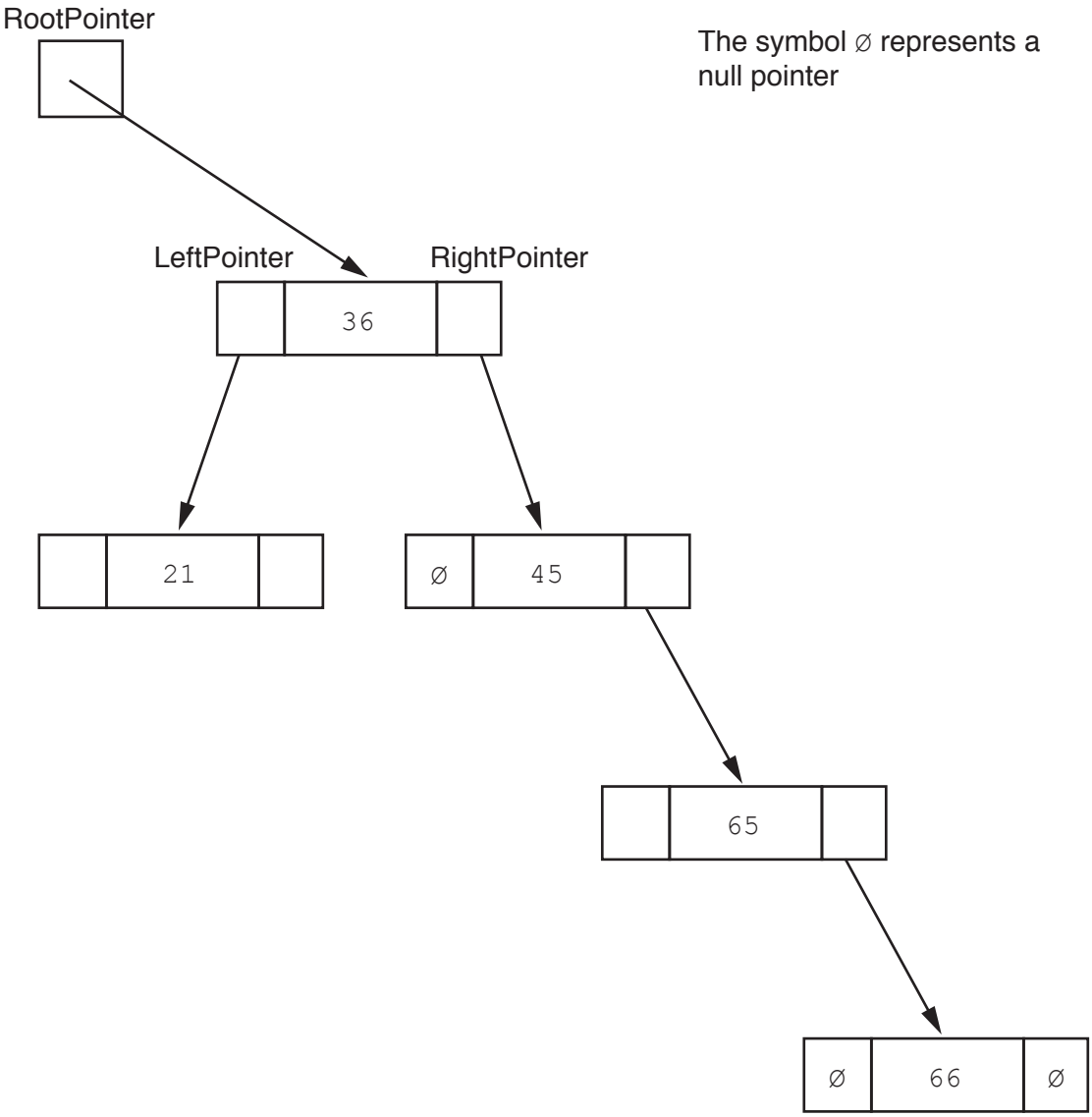
1 A company wants an online marking system for an examination.

(a) The following is a selection of data showing final marks.

```
36, 45, 21, 65, 66, 13, 54, 53, 34
```

A linked list of nodes will be used to store the data. Each node consists of the data, a left pointer and a right pointer. The linked list will be organised as a binary tree.

(i) Complete the binary tree to show how the data above will be organised.



[5]

(ii) The following diagram shows a 2D array that stores the nodes of the binary tree's linked list.

Add the correct pointer values to complete the diagram, using your answer from **part (a)(i)**.

RootPointer

0

FreePointer

Index	LeftPointer	Data	RightPointer
0		36	
1		45	
2		21	
3		65	
4		66	
5		13	
6		54	
7		53	
8		34	
9			

[6]

[Turn over

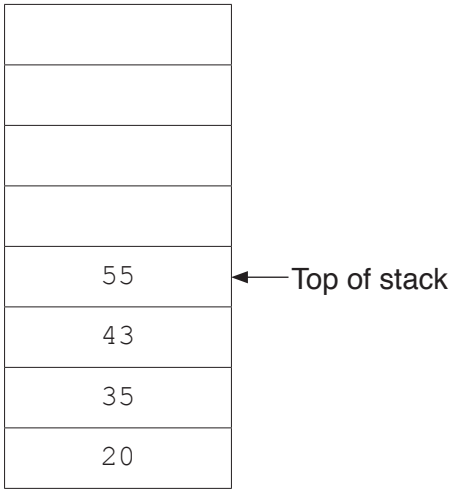
2 A stack is an Abstract Data Type (ADT).

(a) Tick (✓) **one** box to show the statement that describes a stack data structure.

Statement	Tick (✓)
Last in first out	
First in first out	
Last in last out	

[1]

(b) A stack contains the values `20`, `35`, `43`, `55`.

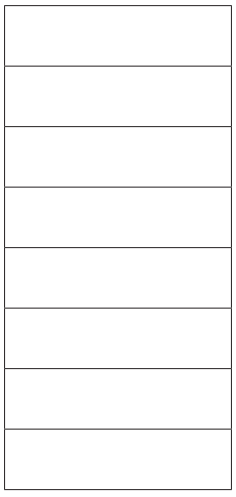


(i) Show the contents of the stack in **part (b)** after the following operations.

```
POP()

POP()

PUSH(10)
```



[1]

(ii) Show the contents of the stack from **part (b)(i)** after these further operations:

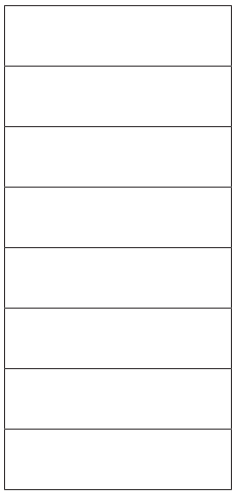
```
POP()

PUSH(50)

PUSH(55)

POP()

PUSH(65)
```



[1]

(iii) The stack is implemented as a 1D array, with eight elements, and given the identifier `ArrayStack`.

The global variable `Top` contains the index of the last element in the stack, or –1 if the stack is empty.

The function `Push()`:

- takes as a parameter an `INTEGER` value to place on the stack
- adds the value to the top of the stack and returns `TRUE` to show that the operation was successful
- returns `FALSE` if the stack is full.

Write an algorithm in **pseudocode** for the function `Push()`.

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

..... [7]

4 (a) A program has sorted some data in the array, `List`, in ascending order.

The following binary search algorithm is used to search for a value in the array.

```
ValueFound ← FALSE
UpperBound ← LengthOfList - 1
LowerBound ← 0
NotInList ← FALSE

WHILE ValueFound = FALSE AND NotInList = FALSE
   MidPoint ← ROUND((LowerBound + UpperBound) / 2)

   IF List[LowerBound] = SearchValue
      THEN
         ValueFound ← TRUE
      ELSE
         IF List[MidPoint] < SearchValue
            THEN
               UpperBound ← MidPoint + 1
            ELSE
               UpperBound ← MidPoint - 1
         ENDIF
         IF LowerBound > MidPoint
            THEN
               NotInList ← TRUE
         ENDIF
   ENDIF
ENDWHILE

IF ValueFound = FALSE
   THEN
      OUTPUT "The value is in the list"
   ELSE
      OUTPUT "The value is not found in the list"
ENDIF
```

Note:

The pseudocode function

`ROUND(Reall : REAL) RETURNS INTEGER`

rounds a number to the nearest integer value.

For example: `ROUND(4.5)` returns `5` and `ROUND(4.4)` returns `4`

(i) There are four errors in the algorithm.

Write the line of code where an error is present **and** write the correction in **pseudocode**.

Error 1 .....

Correction .....

Error 2 .....

Correction .....

Error 3 .....

Correction .....

Error 4 .....

Correction .....
[4]

(ii) A binary search is one algorithm that can be used to search an array.

Identify another searching algorithm.

..... [1]

(b) The following is an example of a sorting algorithm. It sorts the data in the array `ArrayData`.

```
01  TempValue ← ""
02  REPEAT
03     Sorted ← TRUE
04     FOR Count ← 0 TO 4
05        IF ArrayData[Count] > ArrayData[Count + 1]
06           THEN
07              TempValue ← ArrayData[Count + 1]
08              ArrayData[Count + 1] ← ArrayData[Count]
09              ArrayData[Count] ← TempValue
10              Sorted ← FALSE
11        ENDIF
12     ENDFOR
13  UNTIL Sorted = TRUE
```

(i) Complete the trace table for the algorithm given in **part (b)**, for the `ArrayData` values given in the table.

Count	**TempValue**	**Sorted**	**ArrayData**					
			0	**1**	**2**	**3**	**4**	**5**
			5	20	12	25	32	29

[4]

(ii) Rewrite lines 4 to 12 of the algorithm in **part (b)** using a `WHILE` loop instead of a `FOR` loop.

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

..... [3]

(iii) Identify the algorithm shown in **part (b)**.

..... [1]

(iv) Identify another sorting algorithm.

..... [1]

4 Zara is writing a program to simulate a circular queue.

The queue, `MyNumbers`, has 10 elements. `Enqueue()` takes a parameter value and stores it at the tail of the queue. `Dequeue()` returns the item at the head of the queue.

The current state of the circular queue is:

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Data			31	45	89	500	23	2		

`HeadIndex: 2`

`TailIndex: 8`

(a) Show the state of the queue, `HeadIndex` and `TailIndex` after the following operations:

```
Enqueue(23)

Enqueue(100)

Dequeue()

Dequeue()

Enqueue(50)
```

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Data										

`HeadIndex:` .....

`TailIndex:` .....

[3]

(b) The global array, `MyNumbers`, is used to store the positive integer numbers for the queue.

The following global variables are used:

- `HeadIndex` stores the index of the first element in the queue
- `TailIndex` stores the index of the next free space in the queue
- `NumberInQueue` stores the number of items in the queue.

(i) The function `Enqueue()` takes the value to be added to the queue as a parameter. The function returns `TRUE` if the item was added, or `FALSE` if the queue is full.

Complete the **pseudocode** for the function `Enqueue()`.

```
FUNCTION Enqueue(BYVALUE DataToInsert : INTEGER) RETURNS BOOLEAN

  IF NumberInQueue > .....

    THEN

     RETURN FALSE

    ELSE

     MyNumbers[.....] ← .....

     TailIndex ← .....

     IF TailIndex > 9

       THEN

        TailIndex ← .....

     ENDIF

     NumberInQueue ← NumberInQueue + 1

     RETURN TRUE

  ENDIF


ENDFUNCTION
```

[5]

(ii) The function `Dequeue()` returns the value at the head of the queue, or −1 if the queue is empty.

Complete the **pseudocode** for the function `Dequeue()`.

```
FUNCTION Dequeue() RETURNS INTEGER
```

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

```
ENDFUNCTION
```

[5]

(iii) to create a new instance of `FullTimeStudent` with:

- identifier: `NewStudent`
- name: A. Nyone
- date of birth: 12/11/1990
- telephone number: 099111

Programming language .....

.....

.....

.....

.....

.....

..... [3]

4 A dictionary Abstract Data Type (ADT) has these associated operations:

- Create dictionary `(CreateDictionary)`
- Add key-value pair to dictionary `(Add)`
- Delete key-value pair from dictionary `(Delete)`
- Lookup value `(Lookup)`

The dictionary ADT is to be implemented as a two-dimensional array. This stores key-value pairs.

The pseudocode statement

```
DECLARE Dictionary : Array[1:2000, 1:2] OF STRING
```

reserves space for 2000 key-value pairs in array `Dictionary`.

The `CreateDictionary` operation initialises all elements of `Dictionary` to the empty string.

(a) The hashing function `Hash` is to extract the first letter of the key and return the position of this letter in the alphabet. For example `Hash("Action")` will return the integer value 1.
(Note: The ASCII code for the letter A is 65.)

Complete the pseudocode:

```
FUNCTION Hash (.....) RETURNS .....
   DECLARE Number : INTEGER
   Number ←.....
   .....
ENDFUNCTION
```

[5]

(b) The algorithm for adding a new key-value pair to the dictionary is written, using pseudocode, as a procedure.

```
PROCEDURE Add(NewKey : STRING, NewValue : STRING)
   Index ← Hash(NewKey)
   Dictionary[Index, 1] ← NewKey          // store the key
   Dictionary[Index, 2] ← NewValue        // store the value
ENDPROCEDURE
```

An English-German dictionary of Computing terms is to be set up.

(i) Dry-run the following procedure calls by writing the keys and values in the correct elements of `Dictionary`.

```
Add("File", "Datei")
Add("Disk", "Platte")
Add("Error", "Fehler")
Add("Computer", "Rechner")
```

Dictionary

Index	Key	Value
1		
2		
3		
4		
5		
6		
7		
8		
:		
:		
1999		
2000		

[2]

(ii) Another procedure call is made: `Add("Drive", "Laufwerk")`

Explain the problem that occurs when this key-value pair is saved.

.....

.....

.....

..... [2]

(iii) Describe a method to handle the problem identified in **part (b)(ii)**.

.....

.....

.....

.....[2]

(iv) Write **pseudocode** to implement the method you described in **part (b) (iii)**. Choose line numbers to indicate where your pseudocode should be inserted in the given pseudocode.

```
10  PROCEDURE Add(NewKey : STRING, NewValue : STRING)
20      Index ← Hash(NewKey)
30      Dictionary[Index, 1] ← NewKey      // store the key
40      Dictionary[Index, 2] ← NewValue    // store the value
50  ENDPROCEDURE
```

.....

.....

.....

.....

.....

.....

.....

.....

.....

..... [4]

6 A linked list abstract data type (ADT) is created. This is implemented as an array of records. The records are of type `ListElement`.

An example of a record of `ListElement` is shown in the following table.

Data Item	Value
Country	"Scotland"
Pointer	1

(a) (i) Use **pseudocode** to write a definition for the record type, `ListElement`.

.....

.....

.....

.....

..... [3]

(ii) Use **pseudocode** to write an array declaration to reserve space for only 15 nodes of type `ListElement` in an array, `CountryList`. The lower bound element is `1`.

..... [2]

(b) The program stores the position of the last node in the linked list in `LastNode`. The last node always has a `Pointer` value of `-1`. The position of the node at the head of the list is stored in `ListHead`.

After some processing, the array and variables are in the following state.

ListHead
1
LastNode
3

CountryList

	Country	Pointer
1	"Wales"	2
2	"Scotland"	4
3		-1
4	"England"	5
5	"Brazil"	6
6	"Canada"	7
7	"Mexico"	8
8	"Peru"	9
9	"China"	10
10		11
11		12
12		13
13		14
14		15
15		3

A **recursive** algorithm searches the list for a value, deletes that value, and updates the required pointers. When a node value is deleted, it is set to empty "" and the node is added to the end of the list.

A node value is deleted using the pseudocode statement

```
CALL DeleteNode("England", 1, 0)
```

Complete the following **pseudocode** to implement the `DeleteNode` procedure.

```
PROCEDURE DeleteNode(NodeValue: STRING, ThisPointer : INTEGER,
                                                    PreviousPointer : INTEGER)

 IF CountryList[ThisPointer].Value = NodeValue

   THEN

     CountryList[ThisPointer].Value ← ""

     IF ListHead = ……

       THEN

         ListHead ← ……

       ELSE

         CountryList[PreviousPointer].Pointer ← CountryList[ThisPointer].Pointer

     ENDIF

     CountryList[LastNode].Pointer ← ……

     LastNode ← ThisPointer

     ……

   ELSE

     IF CountryList[ThisPointer].Pointer <> -1

       THEN

          CALL DeleteNode(NodeValue, ……,
                                                              ThisPointer)

       ELSE

          OUTPUT "DOES NOT EXIST"

     ENDIF

 ENDIF

ENDPROCEDURE
```

[5]

(d) Noona describes an example of a feature of object-oriented programming (OOP). She says:

"One method exists in the parent class but is overwritten in the child class, to behave differently."

Identify the feature Noona has described.

..... [1]

2 The number of cars that cross a bridge is recorded each hour. This number is placed in a circular queue before being processed.

(a) The queue is stored as an array, `NumberQueue`, with eight elements. The function `AddToQueue` adds a number to the queue. `EndPointer` and `StartPointer` are global variables.

Complete the following **pseudocode** algorithm for the function `AddToQueue`.

```
FUNCTION AddToQueue(Number : INTEGER) RETURNS BOOLEAN

   DECLARE TempPointer : INTEGER

   CONSTANT FirstIndex = 0

   CONSTANT LastIndex = .....

   TempPointer ← EndPointer + 1

   IF ..... > LastIndex

       THEN

          TempPointer ← .....

   ENDIF

   IF TempPointer = StartPointer

       THEN

          RETURN .....

       ELSE

          EndPointer ← TempPointer

          NumberQueue[EndPointer] ← .....

          RETURN TRUE

   ENDIF

ENDFUNCTION
```

[5]

(b) Describe how a number is removed from the circular queue to be processed.

.....

.....

.....

.....

.....

.....

.....

..... [4]

(c) A queue is one example of an Abstract Data Type (ADT).

Identify **three other** Abstract Data Types.

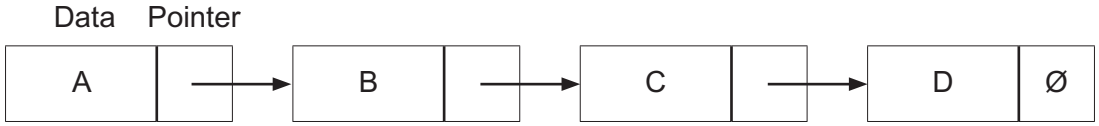
1 .....

2 .....

3 .....

[3]

6 The following diagram represents a linked list.



The symbol Ø represents a null pointer.

(a) The node with the data value C is removed from the list.

Show the new state of the linked list.

[2]

(b) State what happens to the node with the data value C when it is removed from the list.

.....

..... [1]

(c) State what is meant by a **null pointer**.

..... [1]

(d) The linked list is implemented as a 1D array, `LinkedList`. The array is declared as a record data type, with two fields, `Data` and `Pointer`.

The function `FindValue()` takes as a parameter, the value to be searched for in the linked list. The function follows the pointers in the linked list. It returns `-1` if the value is not found, or it returns the pointer to the value if it is found.

The global variable `StartPointer` points to the first element in the list.

Write **pseudocode** for the function `FindValue()`.

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

..... [8]

(e) A linked list and a record are both examples of abstract data types.

Identify **and** describe **one other** abstract data type.

Abstract data type .....

Description .....

.....

.....

.....

.....

.....

.....

[4]

4 Each node of a binary tree is a record. Each record has a left pointer, an integer data value between 0 and 100 inclusive, and a right pointer.

For example:

Item	Example data
`LeftPointer`	2
`Data`	34
`RightPointer`	3

(a) Write **pseudocode** to declare the record with the identifier `Node`.

.....

.....

.....

.....

.....

.....

..... [2]

(b) Write **pseudocode** to declare a new node, `Node100`, **and** assign 100 to its data value, 1 to the left pointer and 4 to the right pointer.

.....

.....

.....

.....

.....

.....

..... [3]

(c) The ordered binary tree is stored as a 1D global array named `BinaryTree` of type `Node`.

`RootNode` and `FreePointer` are declared as global variables.

A null pointer is represented by `-1`.

The current state of the binary tree is shown in the following table:

RootNode	0	**Index**	**LeftPointer**	**Data**	**RightPointer**
		[0]	1	23	3
FreePointer	6	[1]	-1	5	2
		[2]	-1	8	4
		[3]	5	100	-1
		[4]	-1	9	-1
		[5]	-1	88	-1
		[6]	-1	null	-1
		[7]	-1	null	-1

(i) State the purpose of the free pointer.

.....

..... [1]

(ii) Identify an appropriate integer value to represent null data.

..... [1]

(iii) Draw the current state of the binary tree.

[2]

(iv) The procedure `AddData()`:

- takes the node to be added to the tree as a parameter
- finds the location for the node to be stored
- stores the node in the next free array index
- stores `-1` in the new node's `LeftPointer` and `RightPointer`
- updates the pointers in the other nodes
- updates `FreePointer`.

Complete the pseudocode for the procedure `AddData()`.

```
PROCEDURE AddData(NewNode)
    BinaryTree[FreePointer] ← .....
    BinaryTree[FreePointer].LeftPointer ← -1
    BinaryTree[FreePointer].RightPointer ← -1
    DECLARE PositionFound : BOOLEAN
    DECLARE PointerCounter : INTEGER
    PositionFound ← .....
    PointerCounter ← RootNode
    WHILE NOT .....
        IF ..... < BinaryTree[PointerCounter].Data
            THEN
                IF BinaryTree[PointerCounter].LeftPointer = -1
                    THEN
                        BinaryTree[PointerCounter].LeftPointer ← FreePointer
                        PositionFound ← TRUE
                    ELSE
                        PointerCounter ← BinaryTree[PointerCounter].LeftPointer
                ENDIF
            ELSE
                IF BinaryTree[PointerCounter].RightPointer = -1
                    THEN
                        BinaryTree[PointerCounter].RightPointer ← FreePointer
                        PositionFound ← TRUE
                    ELSE
                        PointerCounter ← BinaryTree[PointerCounter].RightPointer
                ENDIF
        ENDIF
    ENDWHILE
    FreePointer ← FreePointer .....
ENDPROCEDURE
```

[5]

6 Details of errors generated in a program are stored in a stack.

Details of each error are stored in a record structure, `Error`.

(a) State which error will be the first retrieved from the stack.

.....

..... [1]

(b) The stack is implemented as a 1D array with the identifier `ErrorArray`.

The pointer `LastItem` stores the position of the last error in the array.

(i) The function, `AddItemToStack`, takes the next error, the array, and pointer as parameters.

If the stack is full, the function returns `FALSE`; otherwise it adds the error to the stack, changes the pointer's value and returns `TRUE`.

Complete the following pseudocode for the function `AddItemToStack`.

```
FUNCTION AddItemToStack(BYREF ErrorArray : ARRAY[0 : 99] OF Error,
                        BYREF LastItem : INTEGER,
                        BYVALUE Error1 : Error) RETURNS BOOLEAN

    IF LastItem = .....

        THEN

            RETURN .....

        ELSE

            ErrorArray[LastItem + 1] ← .....

            LastItem ← .....

            RETURN .....

    ENDIF

ENDFUNCTION
```

[4]

(ii) Explain the reasons why `ErrorArray` and `LastItem` are passed by reference, but `Error1` is passed by value.

.....

.....

.....

.....

.....

..... [3]

(iii) The function `RemoveItem` takes the next error from the stack and returns it.

If there are no errors in the stack, it returns the global record `NullError`.

Complete the pseudocode algorithm `RemoveItem`.

```
FUNCTION RemoveItem(BYREF ErrorArray : ARRAY[0 : 99] OF Error,
                    BYREF LastItem : INTEGER) RETURNS Error

    DECLARE ItemToRemove : Error

    IF ..........................................

        THEN

            RETURN ..........................................

        ELSE

            ItemToRemove ← ErrorArray[..........................................]

            LastItem ← LastItem - 1

            RETURN ..........................................

ENDFUNCTION
```

[3]

(iv) The errors that have been processed are stored in a global queue, `ErrorComplete`.

The function `Enqueue` adds a record to `ErrorComplete`:

```
Enqueue(ErrorToAdd)
```

`Enqueue()` returns `TRUE` if the record is successfully added to the queue, and returns `FALSE` if the queue is full.

The procedure `RunError()` should:

- remove a record from the stack using the function `RemoveItem()`
- output an appropriate message if there were no records in the stack
- if an error record is returned, add the record to the queue using the function `Enqueue()`
- if the record is added to the queue, output an appropriate message
- if the record is not added to the queue, output an appropriate message.

Complete the pseudocode procedure `RunError()`.

```
PROCEDURE RunError(BYREF ErrorComplete : ARRAY[0 : 99] OF Error,
                   BYREF ErrorArray : ARRAY[0 : 99] OF Error)
```

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

```
ENDPROCEDURE
```

[5]