4.1. Computational thinking and problem-solving (Pastpapers 2015 – 2018)

- 4.1.1 Abstraction
- 4.1.2 Algorithms
- 4.1.3 Abstract Data Types (ADT)

9608/41/M/J/15

Q. 1 /- A queue Abstract Data Type (ADT) has these associated operations:

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

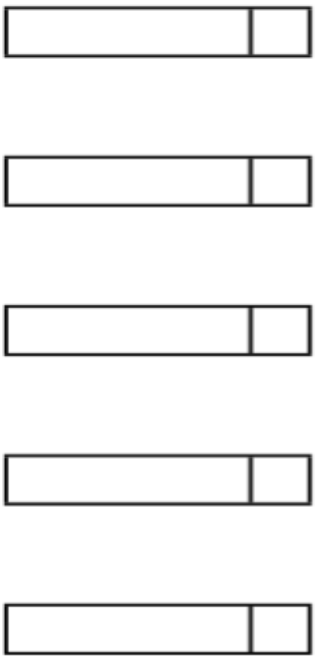
The queue ADT is to be implemented as a linked list of nodes.

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
```

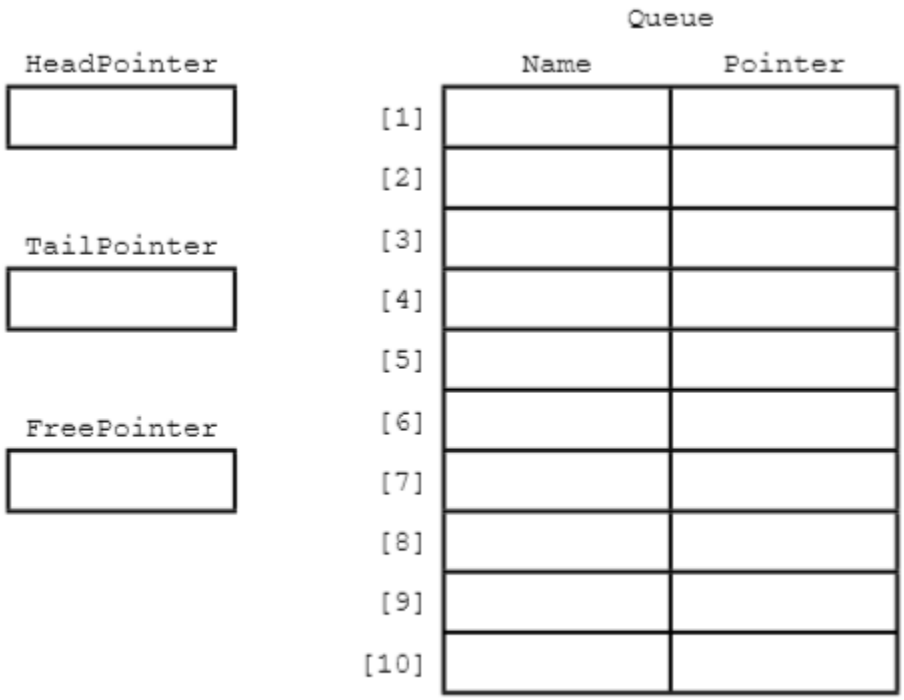
The statement

```
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]

9608/43/M/J/15

Q2/- A stack Abstract Data Type (ADT) has these associated operations:

- create stack
- add item to stack (push)
- remove item from stack (pop)

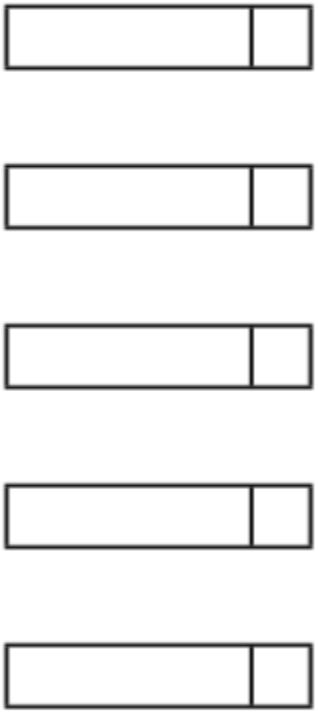
The stack ADT is to be implemented as a linked list of nodes.
Each node consists of data and a pointer to the next node.

(a) There is one pointer: the top of stack pointer, which points to the last item added to the stack.

Draw a diagram to show the final state of the stack after the following operations are carried out.

```
CreateStack
Push("Ali")
Push("Jack")
Pop
Push("Ben")
Push("Ahmed")
Pop
Push("Jatinder")
```

Add appropriate labels to the diagram to show the final state of the stack. 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
```

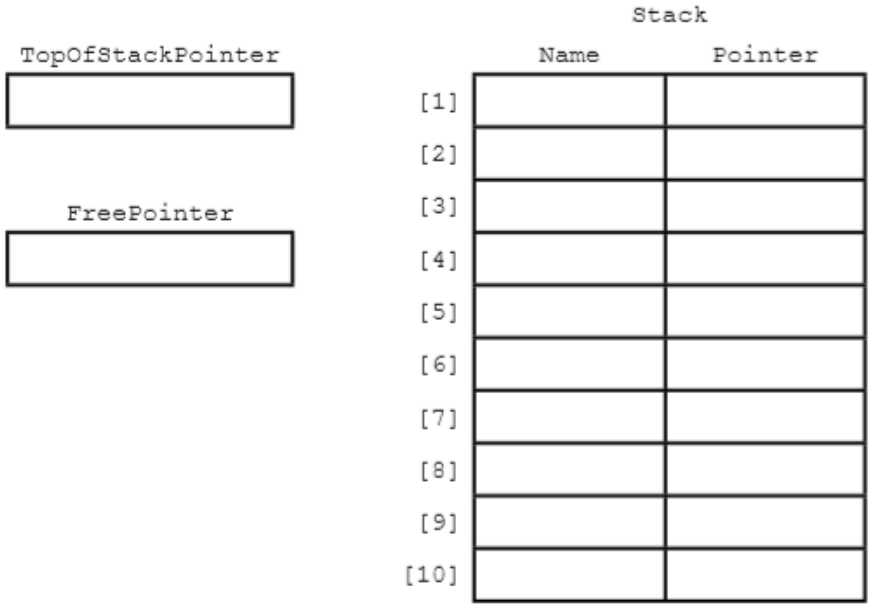
The statement

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

reserves space for 10 nodes in array `Stack`.

(i) The `CreateStack` operation links all nodes and initialises the `TopOfStackPointer` and `FreePointer`.

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



[4]

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

```
PROCEDURE Push (NewName)
```

Where NewName is the new name to be added to the stack.

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

Identifier	Data type	Description
Stack	Array[1:10] OF Node	
NewName	STRING	Name to be added
FreePointer	INTEGER	Pointer to next free node in array
TopOfStackPointer	INTEGER	Pointer to first node in stack
TempPointer	INTEGER	Temporary store for copy of FreePointer

```
PROCEDURE Push(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
          Stack[FreePointer].Name ← NewName
          // take a temporary copy and
          // then adjust free pointer
          TempPointer ← FreePointer
          FreePointer ← Stack[FreePointer].Pointer
          // link current node to previous top of stack
          Stack[TempPointer].Pointer ← TopOfStackPointer
          // adjust TopOfStackPointer to current node
          TopOfStackPointer ← TempPointer
  ENDIF
ENDPROCEDURE
```

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

```
PROCEDURE Pop()
  // Report error if Stack is empty
.....
.....
.....
.....
   OUTPUT Stack [.....].Name
   // take a copy of the current top of stack pointer
.....
   // update the top of stack pointer
.....
   // link released node to free list
.....
.....
.....
ENDPROCEDURE
```

[5]

9608/41/M/J/16

Q.3/- 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]

9608/42/M/J/17

Q4/- 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.

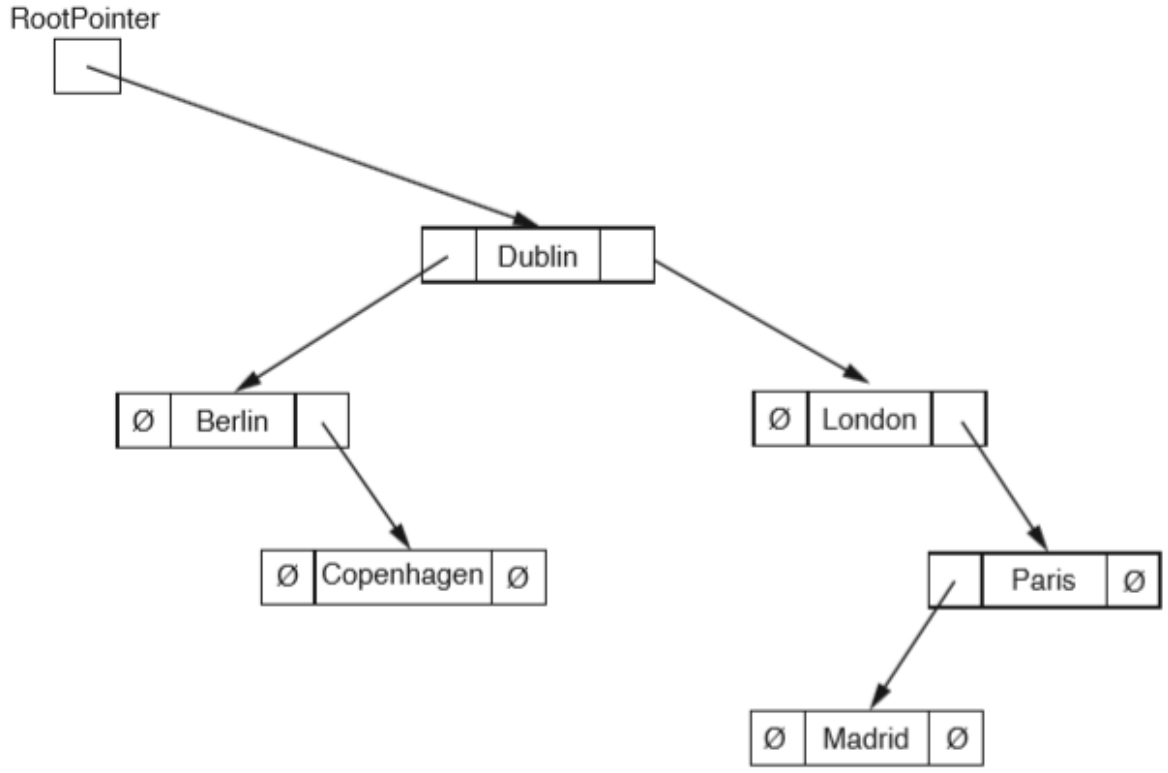
(a) A null pointer is shown as ϕ.

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

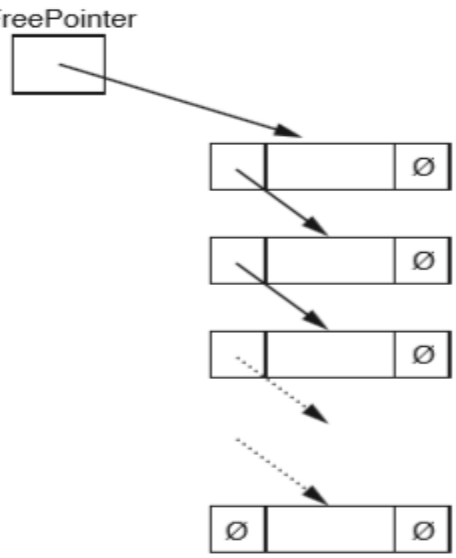


Another data item to be added is Athens.

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 lis 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]

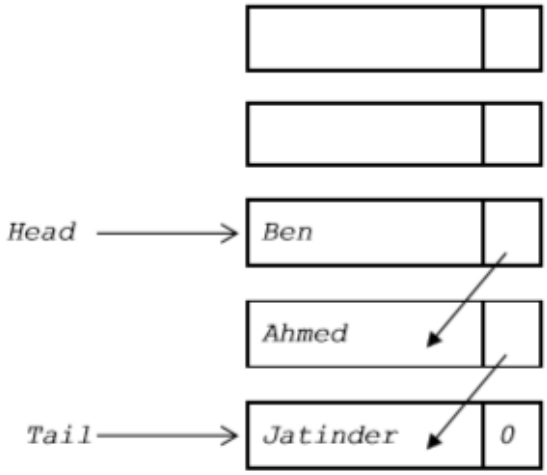
(ii) Complete the pseudocode to add a data item to the tree.

```
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]

9608/41/M/J/1
Answers:
Q1/-

(a)



1 mark for Head and Tail pointers
1 mark for 3 correct items – linked as shown
1 mark for correct order with null pointer in last nod **[3]**

(b) (i)

Queue

	HeadPointer	Name	Pointer
	0		
[1]			2
[2]			3
	TailPointer		
[3]			4
	0		
[4]			5
[5]			6
	FreePointer		
[6]			7
	1		
[7]			8
[8]			9
[9]			10
[10]			0

Mark as follows:

HeadPointer =0 & TailPointer = 0
FreePointer assigned a value
Pointers[1] to [9] links the nodes together
Pointer[10] = 'Null' **[4]**

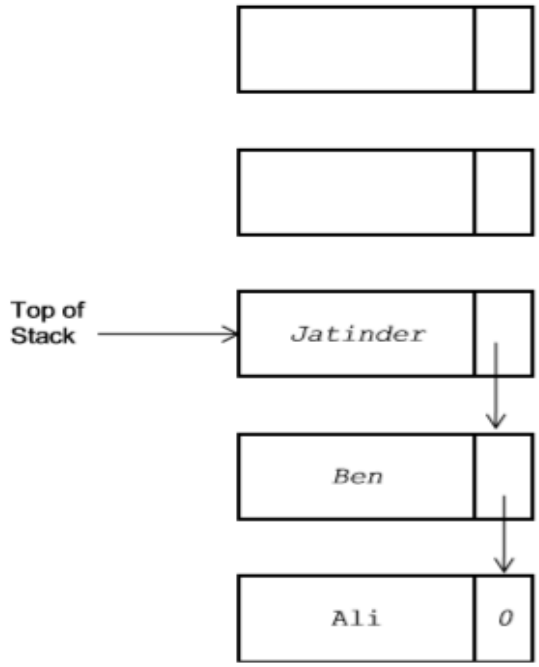
(ii)

```
PROCEDURE RemoveName()
    // Report error if Queue is empty
    IF HeadPointer = 0
        THEN
            Error
        ELSE
            OUTPUT Queue[HeadPointer].Name
            // current node is head of queue
            CurrentPointer ← HeadPointer
            // update head pointer
            HeadPointer ← Queue[CurrentPointer].Pointer
            //if only one element in queue,then update tail pointer
            IF HeadPointer = 0
                THEN
                    TailPointer ← 0
            ENDIF
                // link released node to free list
                Queue[CurrentPointer].Pointer ← FreePointer
            FreePointer ← CurrentPointer
    ENDIF
ENDPROCEDURE
```

[max 6]

9608/43/M/J/15
Q2/-

(a)



1 mark for Top of Stack pointer
1 mark for 3 correct items
1 mark for correct order with null pointer in last node

[3]

(b) (i)

Stack

TopOfStackPointer
0

FreePointer
1

	Name	Pointer
[1]		2
[2]		3
[3]		4
[4]		5
[5]		6
[6]		7
[7]		8
[8]		9
[9]		10
[10]		0

Mark as follows:
TopOfStackPointer
FreePointer
Pointers[1] to [9]
Pointer[10]

[4]

(ii)

```
PROCEDURE Pop()
    // Report error if Stack is empty
    IF TopOfStackPointer = 0
        THEN
            Error
        ELSE
            OUTPUT Stack[TopOfStackPointer].Name
            // take a copy of the current top of stack pointer
            TempPointer ← TopOfStackPointer
            // update the top of stack pointer
            TopOfStackPointer ← Stack[TempPointer].Pointer
            // link released node to free list
            Stack[TempPointer].Pointer ← FreePointer
            FreePointer ← TempPointer
    ENDIF
ENDPROCEDURE
```

1 mark for each line of code as above (first 4 lines + ENDIF for 1 mark) **[Max 5]**

9608/41/M/J/16

Q.3/-

(a) (i)

`TYPE LinkedList`	1	**3**
`(DECLARE) Surname : STRING` `(DECLARE) Ptr : INTEGER`	1	
`ENDTYPE`	1	
Accept: `LinkedList : RECORD`	1	
`Surname : STRING` `Ptr : INTEGER`	1	
`ENDRECORD`	1	
Accept: `TYPE LinkedList = RECORD`	1	
`Surname : STRING` `Ptr : INTEGER`	1	
`ENDTYPE / ENDRECORD`	1	
Accept: `STRUCTURE LinkedList`	1	
`(DECLARE) Surname : STRING` `(DECLARE) Ptr : INTEGER`	1	
`ENDSTRUCTURE`	1	
Accept `AS` / `OF` instead of `:`		

(ii)	(DECLARE) **SurnameList[1:5000]** : **LinkedList** Accept AS / OF instead of : Accept () instead of [] Accept without lower bound Index separator can be , : ...		2
(b) (i)	Wu Accept with quotes		1
(ii)	6		1
(c) (i)	IsFound + relevant description BOOLEAN	1 1	2

Question	Answer	Marks
(ii)	Accept () instead of [] 01 Current ← **StartPtr** 02 IF Current = 0 03 THEN 04 OUTPUT **"Empty List"** (or similar message) (accept without quotes) Reject "Error" 05 ELSE 06 IsFound ← **FALSE** 07 INPUT ThisSurname 08 REPEAT 09 IF **SurnameList[Current].Surname** = ThisSurname 10 THEN 11 IsFound ← TRUE 12 OUTPUT "Surname found at position ", Current 13 ELSE 14 // move to the next list item 15 **Current ← SurnameList[Current].Ptr** 16 ENDIF 17 UNTIL IsFound = TRUE OR **Current = 0** 18 IF IsFound = FALSE 19 THEN 20 OUTPUT "Not Found" 21 ENDIF 22 ENDIF	6

9608/42/M/J/17

Q4/- Answers

2(a)	• A pointer that doesn't point to another node/other data/address // indicates the end of the branch	1
2(b)	one mark per bullet • node with 'Athens' linked to left pointer of Berlin (ignore null pointer) • null pointers in left and right pointers of Athens	2
2(c)(i)	RootPointer: 0 FreePointer: 7 1 mark	5
2(c)(ii)	• –1 • It is not the number for any node.	2

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

(d)(i) — 7

```
TYPE Node
    LeftPointer : INTEGER
    RightPointer : INTEGER                                   1
    Data : STRING
ENDTYPE
DECLARE Tree : ARRAY[0 : 9] OF Node                          1
DECLARE FreePointer : INTEGER
DECLARE RootPointer : INTEGER

PROCEDURE CreateTree()
   DECLARE Index : INTEGER
    RootPointer ← -1                                         1
    FreePointer ← 0                                          1
   FOR Index ← 0 TO 9   // link nodes
       Tree[Index].LeftPointer ← Index + 1                   1
       Tree[Index].RightPointer ← -1                         1
   ENDFOR
   Tree[9].LeftPointer ← -1                                  1
ENDPROCEDURE
```