QUESTION 1.

7 A system is monitored using sensors. The sensors output binary values corresp... conditions, as shown in the table:

Parameter	Description of parameter	Binary value	Description of condition
P	oil pressure	1	pressure >= 3 bar
		0	pressure < 3 bar
T	temperature	1	temperature >= 200°C
		0	temperature < 200°C
R	rotation	1	rotation <= 1000 revs per minute (rpm)
		0	rotation > 1000 revs per minute (rpm)

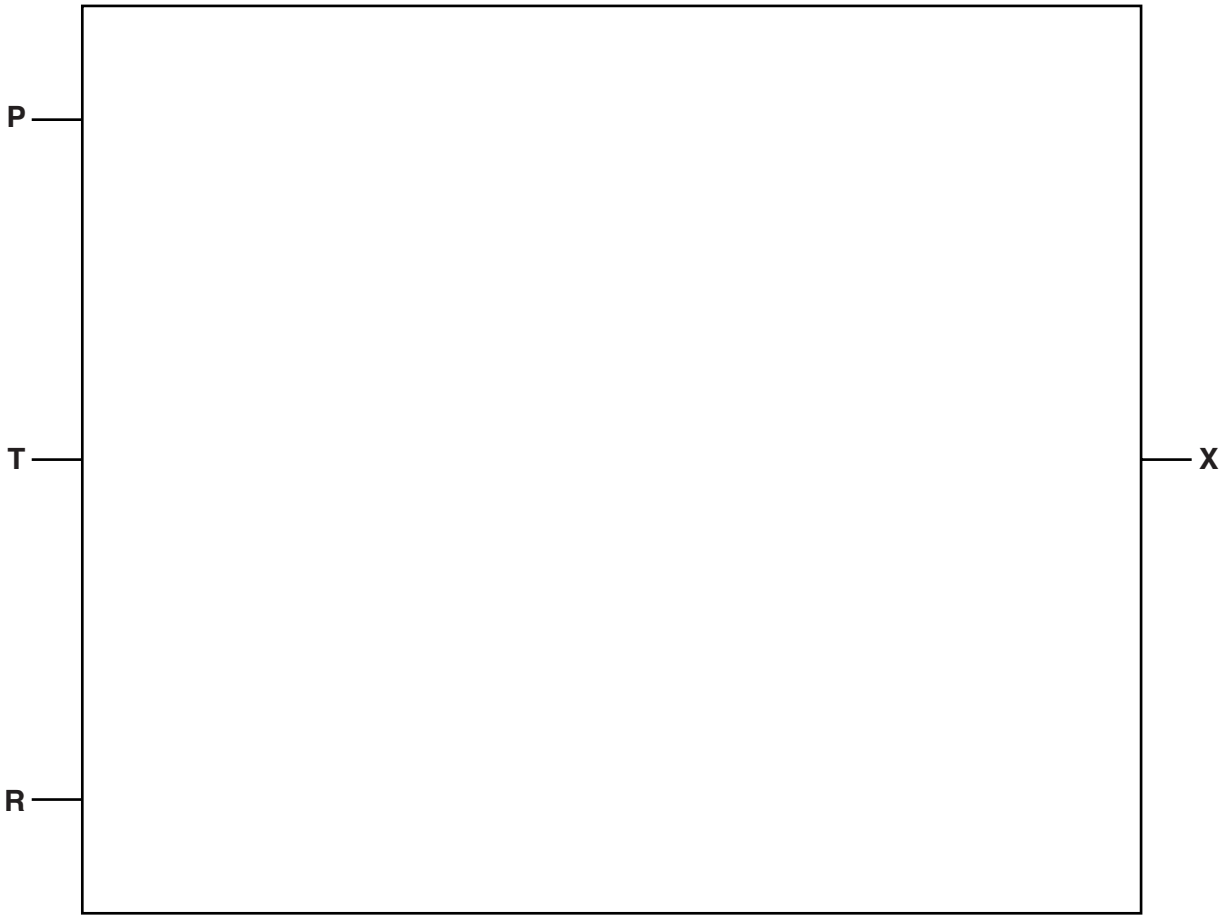
The outputs of the sensors form the inputs to a logic circuit. The output from the circuit, X, is 1 if any of the following three conditions occur:

either oil pressure >= 3 bar **and** temperature >= 200°C

or oil pressure < 3 bar **and** rotation > 1000 rpm

or temperature >= 200°C **and** rotation > 1000 rpm

(a) Draw a logic circuit to represent the above system.



[5]

(b) Complete the truth table for this system.

P	T	R	Workspace	X
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

[4]

QUESTION 2.

2 Assemblers translate from assembly language to machine code. Some assemblers read the assembly language program twice; these are referred to as two-pass assemblers.

The following table shows five activities performed by two-pass assemblers.

Write 1 **or** 2 to indicate whether the activity is carried out during the first pass or during the second pass.

Activity	First pass or second pass
any symbolic address is replaced by an absolute address	
any directives are acted upon	
any symbolic address is added to the symbolic address table	
data items are converted into their binary equivalent	
forward references are resolved	

[5]

QUESTION 1.

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

- create the tree (`CreateTree`)
- add an item to tree (`Add`)
- output items in ascending order (`TraverseTree`)

(a) Show the final state of the binary tree after the following operations are carried out.

```
CreateTree
Add("Dodi")
Add("Farai")
Add("Elli")
Add("George")
Add("Ben")
Add("Celine")
Add("Ali")
```

[4]

(b) The binary tree ADT is to be implemented as an array of nodes. Each node and two pointers.

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

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

The statement

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

reserves space for 10 nodes in array `Tree`.

The `CreateTree` operation links all nodes into a linked list of free nodes. It also initialises the `RootPointer` and `FreePointer`.

Show the contents of the `Tree` array and the values of the two pointers, `RootPointer` and `FreePointer`, after the operations given in **part (a)** have been carried out.

`RootPointer`

`FreePointer`

`Tree`

	Name	LeftPointer	RightPointer
[1]			
[2]			
[3]			
[4]			
[5]			
[6]			
[7]			
[8]			
[9]			
[10]			

[7]

(c) A programmer needs an algorithm for outputting items in ascending order. To
programmer writes a recursive procedure in pseudocode.

(i) Complete the pseudocode:

```
01 PROCEDURE TraverseTree(BYVALUE Root: INTEGER)
02     IF Tree[Root].LeftPointer ..........
03         THEN
04             TraverseTree(..........)
05     ENDIF
06     OUTPUT ..........Name
07     IF ..........<> 0
08         THEN
09             TraverseTree(..........)
10     ENDIF
11 ENDPROCEDURE
```

[5]

(ii) Explain what is meant by a recursive procedure. Give a line number from the code above that shows procedure `TraverseTree` is recursive.

.....

.....

.....

Line number .....[2]

(iii) Write the pseudocode call required to output all names stored in `Tree`.

.....

.....[1]

Question 5 begins on page 14.

QUESTION 2.

3 The following table shows part of the instruction set for a processor which has one register, the Accumulator (ACC), and an index register (IX).

Instruction		Explanation
Op code	**Operand**	
LDM	#n	Immediate addressing. Load the number n to ACC.
LDD	<address>	Direct addressing. Load the contents of the given address to ACC.
LDX	<address>	Indexed addressing. Form the address from <address> + the contents of the index register. Copy the contents of this calculated address to ACC.
LDR	#n	Immediate addressing. Load the number n into IX.
STO	<address>	Store the contents of ACC at the given address.
INC	<register>	Add 1 to the contents of the register (ACC or IX).
DEC	<register>	Subtract 1 from the contents of the register (ACC or IX).
CMP	<address>	Compare the contents of ACC with the contents of <address>.
CMP	#n	Compare the contents of ACC with number n.
JPE	<address>	Following a compare instruction, jump to <address> if the compare was True.
JPN	<address>	Following a compare instruction, jump to <address> if the compare was False.
OUT		Output to the screen the character whose ASCII value is stored in ACC.
END		Return control to the operating system.

A programmer is writing a program that outputs a string, first in its original order and then in reverse order.

The program will use locations starting at address NAME to store the characters in the string. The location with address MAX stores the number of characters that make up the string.

The programmer has started to write the program in the table opposite. The Comment column contains descriptions for the missing program instructions.

Complete the program using op codes from the given instruction set.

Label	Op code	Operand	Comment
START:			// initialise index register to zero
			// initialise COUNT to zero
LOOP1:			// load character from indexed address NAME
			// output character to screen
			// increment index register
			// increment COUNT starts here
			// is COUNT = MAX ?
			// if FALSE, jump to LOOP1
REVERSE:			// decrement index register
			// set ACC to zero
			// store in COUNT
LOOP2:			// load character from indexed address NAME
			// output character to screen
			// decrement index register
			// increment COUNT starts here
			// is COUNT = MAX ?
			// if FALSE, jump to LOOP2
			// end of program
COUNT:			
MAX:	4		
NAME:	B01000110		// ASCII code in binary for 'F'
	B01010010		// ASCII code in binary for 'R'
	B01000101		// ASCII code in binary for 'E'
	B01000100		// ASCII code in binary for 'D'

[15]

QUESTION 3.

3 **(a)** The numerical difference between the ASCII code of an upper case letter and the ASCII code of its lower case equivalent is 32 denary (32_{10}).

For example, `'F'` has ASCII code 70 and `'f'` has ASCII code 102.

	Bit number							
	7	6	5	4	3	2	1	0
ASCII code	ASCII code in binary							
70	0	1	0	0	0	1	1	0
102	0	1	1	0	0	1	1	0

The bit patterns differ only at bit number 5. This bit is 1 if the letter is lower case and 0 if the letter is upper case.

(i) A program needs a mask to ensure that a letter is in **upper case.**

Write the binary pattern of the mask in the space provided in the table below.

	Bit number							
	7	6	5	4	3	2	1	0
ASCII code	ASCII code in binary							
70	0	1	0	0	0	1	1	0
102	0	1	1	0	0	1	1	0
Mask								

Give the bit-wise operation that needs to be performed using the mask and the ASCII code.

.....[2]

(ii) A program needs a mask to ensure that a letter is in **lower case**.

Write the binary pattern of the mask in the space provided in the table below.

	Bit number							
	7	6	5	4	3	2	1	0
ASCII code	ASCII code in binary							
70	0	1	0	0	0	1	1	0
102	0	1	1	0	0	1	1	0
Mask								

Give the bit-wise operation that needs to be performed using the mask and the ASCII code.

.....[2]

The following table shows part of the instruction set for a processor which
purpose register, the Accumulator (ACC), and an index register (IX).

Instruction		Explanation
Op code	**Operand**	
LDM	#n	Immediate addressing. Load the number n to ACC.
LDD	<address>	Direct addressing. Load the contents of the given address to ACC.
LDX	<address>	Indexed addressing. Form the address from <address> + the contents of the index register. Copy the contents of this calculated address to ACC.
LDR	#n	Immediate addressing. Load the number n into IX.
STO	<address>	Store the contents of ACC at the given address.
INC	<register>	Add 1 to the contents of the register (ACC or IX).
CMP	<address>	Compare the contents of ACC with the contents of <address>.
CMP	#n	Compare the contents of ACC with number n.
JPE	<address>	Following a compare instruction, jump to <address> if the compare was True.
JPN	<address>	Following a compare instruction, jump to <address> if the compare was False.
AND	#n	Bitwise AND operation of the contents of ACC with the operand.
AND	<address>	Bitwise AND operation of the contents of ACC with the contents of <address>.
XOR	#n	Bitwise XOR operation of the contents of ACC with the operand.
XOR	<address>	Bitwise XOR operation of the contents of ACC with the contents of <address>.
OR	#n	Bitwise OR operation of the contents of ACC with the operand.
OR	<address>	Bitwise OR operation of the contents of ACC with the contents of <address>.
OUT		Output to the screen the character whose ASCII value is stored in ACC.
END		Return control to the operating system.

A programmer is writing a program that will output the first character of a string in upper case and the remaining characters of the string in lower case.

The program will use locations from address `WORD` onwards to store the characters in the string. The location with address `LENGTH` stores the number of characters that make up the string.

The programmer has started to write the program in the following table. The c
contains descriptions for the missing program instructions.

(b) Complete the program using op codes from the given instruction set.

Label	Op code	Operand	Comment
START:			// initialise index register to zero
			// get first character of WORD
			// ensure it is in upper case using MASK1
			// output character to screen
			// increment index register
			// load 1 into ACC
			// store in COUNT
LOOP:			// load next character from indexed address WORD
			// make lower case using MASK2
			// output character to screen
			// increment COUNT starts here
			// is COUNT = LENGTH ?
			// if FALSE, jump to LOOP
			// end of program
COUNT:			
MASK1:			// bit pattern for upper case
MASK2:			// bit pattern for lower case
LENGTH:		4	
WORD:		B01100110	// ASCII code in binary for 'f'
		B01110010	// ASCII code in binary for 'r'
		B01000101	// ASCII code in binary for 'E'
		B01000100	// ASCII code in binary for 'D'

[12]

Question 4 begins on page 15.

QUESTION 4.

3 `NameList` is a 1D array that stores a sorted list of names. A programmer decl… pseudocode as follows:

```
NameList : Array[0 : 100] OF STRING
```

The programmer wants to search the list using a binary search algorithm.

The programmer decides to write the search algorithm as a recursive function. The function, `Find`, takes three parameters:

- `Name`, the string to be searched for
- `Start`, the index of the first item in the list to be searched
- `Finish`, the index of the last item in the list to be searched

The function will return the position of the name in the list, or –1 if the name is not found.

Complete the **pseudocode** for the recursive function.

```
FUNCTION Find(BYVALUE Name : STRING, BYVALUE Start : INTEGER,
                                BYVALUE Finish : INTEGER) RETURNS INTEGER

   // base case

   IF ..........

      THEN

         RETURN -1

      ELSE

         Middle ← ..........

         IF ..........

            THEN

               RETURN ..........

            ELSE    // general case

               IF SearchItem > ..........

                  THEN

                     ..........

                  ELSE

                     ..........

               ENDIF

         ENDIF

   ENDIF

ENDFUNCTION
```

[7]

QUESTION 5.

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

- 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.

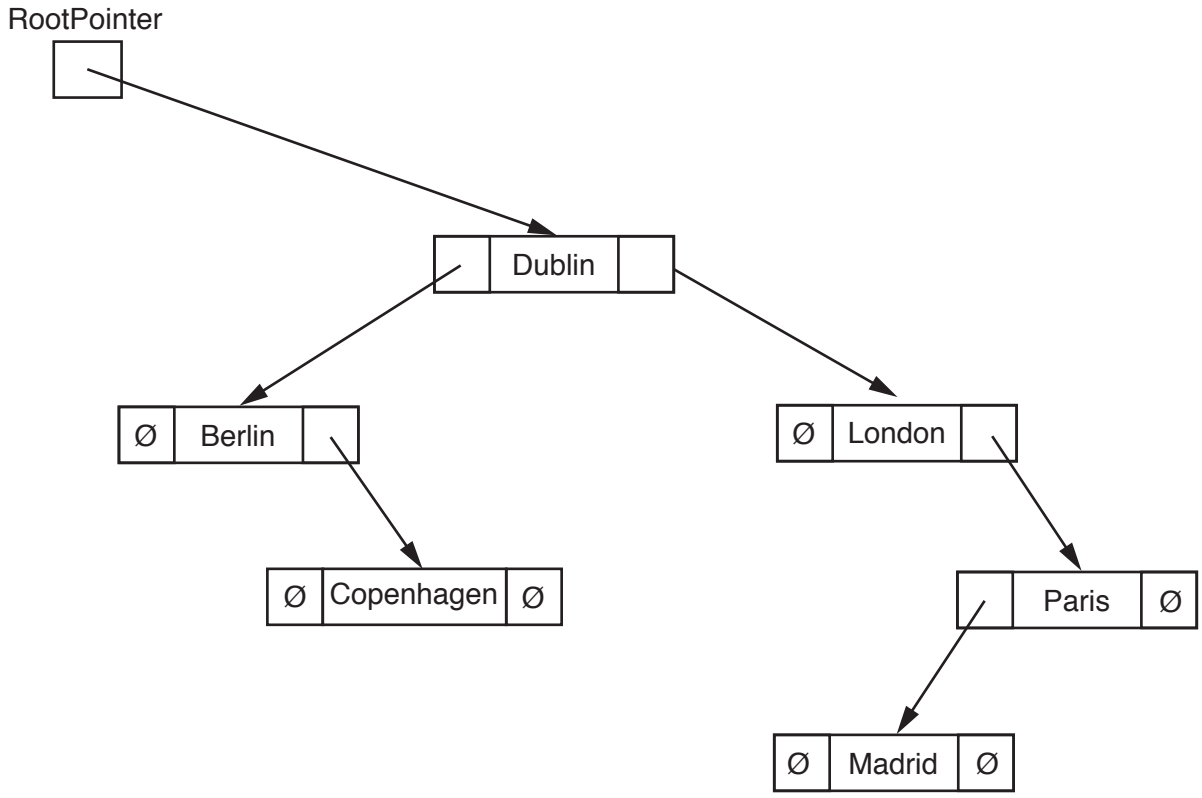
Explain the meaning of the term **null pointer**.

.....

.....[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.

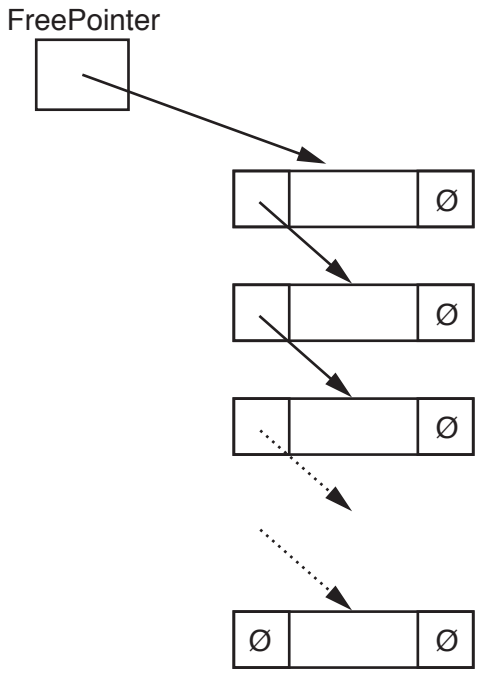
Make the required changes to the diagram when this data item is added. [2]

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

RootPointer

Ø

Unused nodes are linked together 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 th
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. Th
as a recursive solution.

Complete the pseudocode for the recursive procedure `TraverseTree`.

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

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

```
ENDPROCEDURE
```

[5]

QUESTION 6.

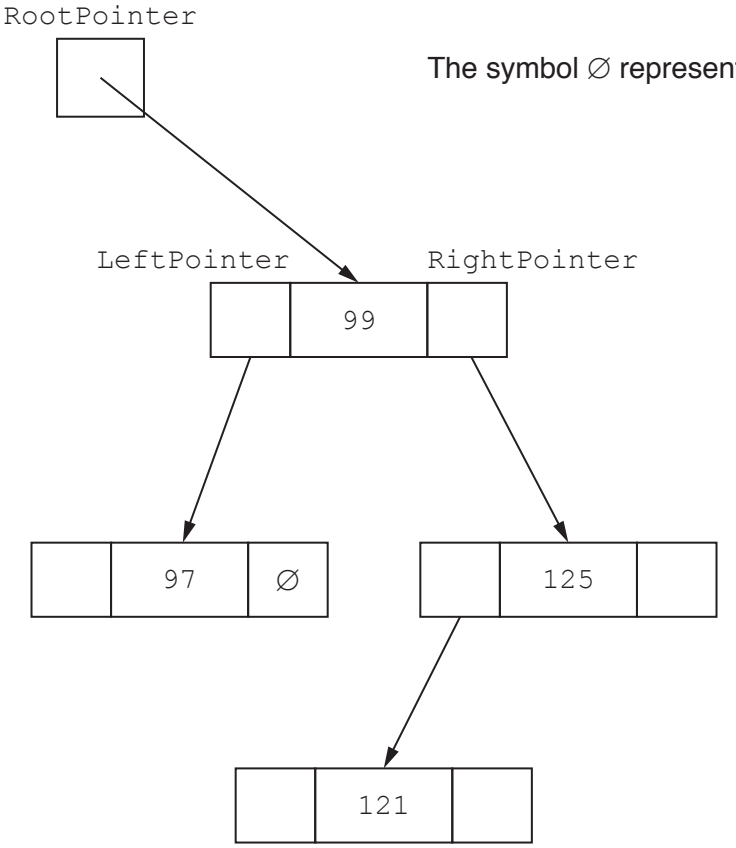
2 A computer games club wants to run a competition. The club needs a system to achieved in the competition.

A selection of score data is as follows:

99, 125, 121, 97, 109, 95, 135, 149

(a) 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 score data above will be organised.



The symbol ∅ represents a null pointer.

[5]

(ii) The following diagram shows a 2D array that stores the nodes of the bi
list.

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

RootPointer

0

FreePointer

Index	LeftPointer	Data	RightPointer
0		99	
1		125	
2		121	
3		97	
4		109	
5		95	
6		135	
7		149	
8			

[6]

(b) The club also considers storing the data in the order in which it receives linked list in a 1D array of records.

The following pseudocode algorithm searches for an element in the linked list.

Complete the **six** missing sections in the algorithm.

```
FUNCTION FindElement(Item : INTEGER) RETURNS ..........

    .......... ← RootPointer

    WHILE CurrentPointer .......... NullPointer

        IF List[CurrentPointer].Data <> ..........

            THEN

                CurrentPointer ← List[..........].Pointer

            ELSE

                RETURN CurrentPointer

        ENDIF

    ENDWHILE

    CurrentPointer ← NullPointer

    .......... CurrentPointer

ENDFUNCTION
```

[6]

(c) The games club is looking at two programming paradigms: imperative and
programming paradigms.

Describe what is meant by the **imperative programming paradigm** and the **object-**
programming paradigm.

(i) Imperative .....

.....

.....

.....

.....

..... [3]

(ii) Object-oriented .....

.....

.....

.....

.....

..... [3]

(d) Players complete one game to place them into a category for the competition. wants to implement a program to place players into the correct category. The has decided to use object-oriented programming (OOP).

The highest score that can be achieved in the game is 150. Any score less than 50 w qualify for the competition. Players will be placed in a category based on their score.

The following diagram shows the design for the class `Player`. This includes the properties and methods.

Player

```
Score    : INTEGER // initialised to 0
Category : STRING  // "Beginner", "Intermediate",
                   // "Advanced" or "Not Qualified", initialised
                   // to "Not Qualified"
PlayerID : STRING  // initialised with the parameter InputPlayerID
```

```
Create()          // method to create and initialise an object using
                  // language-appropriate constructor
SetScore()        // checks that the Score parameter has a valid value
                  // if so, assigns it to Score
SetCategory()     // sets Category based on player's Score
SetPlayerID()     // allows a player to change their PlayerID
                  // validates the new PlayerID
GetScore()        // returns Score
GetCategory()     // returns Category
GetPlayerID()     // returns PlayerID
```

(i) The constructor receives the parameter `InputPlayerID` to create a
Other properties are initialised as instructed in the class diagram.

Write **program code** for the `Create()` constructor method.

Programming language .....

Program code

.....

.....

.....

.....

.....

.....

.....

.....

.....

..... [5]

(ii) Write **program code** for the following **three** get methods.

Programming language .....

`GetScore()`

Program code

.....

.....

.....

.....

`GetCategory()`

Program code

.....

.....

.....

.....

`GetPlayerID()`

Program code

.....

.....

.....

.....

[4]

(iii) The method `SetPlayerID()` asks the user to input the new player ID
value.

It checks that the length of the `PlayerID` is less than or equal to 15 charac
greater than or equal to 4 characters. If the input is valid, it sets this as the `Play`
otherwise it loops until the player inputs a valid `PlayerID`.

Use suitable input and output messages.

Write **program code** for `SetPlayerID()`.

Programming language .....

Program code

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

..... [4]

(iv) The method `SetScore()` checks that its `INTEGER` parameter `ScoreI` it is valid, it is then set as `Score`. A valid `ScoreInput` is greater than or e less than or equal to 150.

If the `ScoreInput` is valid, the method sets `Score` and returns `TRUE`.

If the `ScoreInput` is not valid, the method does not set `Score`, displays an erro message, and it returns `FALSE`.

Write **program code** for `SetScore(ScoreInput : INTEGER)`.

Programming language .....

Program code

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

..... [5]

(v) Write **program code** for the method `SetCategory()`. Use the properti
in the original class definition.

Players will be placed in one of the following categories.

Category	Criteria
Advanced	Score is greater than 120
Intermediate	Score is greater than 80 and less than or equal to 120
Beginner	Score is greater than or equal to 50 and less than or equal to 80
Not Qualified	Score is less than 50

Programming language .....

Program code

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

..... [4]

(vi) Joanne has played the first game to place her in a category for the com

The procedure `CreatePlayer()` performs the following tasks.

- allows the player ID and score to be input with suitable prompts
- creates an instance of `Player` with the identifier `JoannePlayer`
- sets the score for the object
- sets the category for the object
- outputs the category for the object

Write **program code** for the `CreatePlayer()` procedure.

Programming language .....

Program code

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

..... [8]

(e) The programmer wants to test that the correct category is set for a player's s

As stated in **part (d)(v)**, players will be placed in one of the following categories.

Category	Criteria
Advanced	Score is greater than 120
Intermediate	Score is greater than 80 and less than or equal to 120
Beginner	Score is greater than or equal to 50 and less than or equal to 80
Not Qualified	Score is less than 50

Complete the table to provide test data for each category.

Category	Type of test data	Example test data
Beginner	Normal	
	Abnormal	
	Boundary	
Intermediate	Normal	
	Abnormal	
	Boundary	
Advanced	Normal	
	Abnormal	
	Boundary	

[3]

(f) In **part (b)**, the club stored scores in a 1D array. This allows the club to sort t

The following is a sorting algorithm in pseudocode.

```
NumberOfScores ← 5
FOR Item ← 1 TO NumberOfScores - 1
   InsertScore ← ArrayData[Item]
   Index ← Item - 1
   WHILE (ArrayData[Index] > InsertScore) AND (Index >= 0)
      ArrayData[Index + 1] ← ArrayData[Index]
      Index ← Index - 1
   ENDWHILE
   ArrayData[Index + 1] ← InsertScore
ENDFOR
```

(i) Give the name of this algorithm.

..... [1]

(ii) State the name of **one** other sorting algorithm.

..... [1]

(iii) Complete a dry run of the algorithm using the following trace table.

Item	NumberOfScores	InsertScore	Index	ArrayData				
				0	1	2	3	
				99	125	121	109	115

[7]

QUESTION 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 **ps**

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 ar

```
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 i
loop.

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

..... [3]

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

..... [1]

(iv) Identify another sorting algorithm.

..... [1]

QUESTION 8.

5 The following table shows part of the instruction set for a processor which has one register, the Accumulator (ACC) and an Index Register (IX).

Instruction		Explanation
Op code	**Operand**	
LDM	#n	Immediate addressing. Load the number n to ACC.
LDD	<address>	Direct addressing. Load the contents of the location at the given address to ACC.
LDI	<address>	Indirect addressing. The address to be used is at the given address. Load the contents of this second address to ACC.
LDX	<address>	Indexed addressing. Form the address from <address> + the contents of the Index Register. Copy the contents of this calculated address to ACC.
LDR	#n	Immediate addressing. Load the number n to IX.
STO	<address>	Store the contents of ACC at the given address.
STX	<address>	Indexed addressing. Form the address from <address> + the contents of the Index Register. Copy the contents from ACC to this calculated address.
ADD	<address>	Add the contents of the given address to the ACC.
INC	<register>	Add 1 to the contents of the register (ACC or IX).
DEC	<register>	Subtract 1 from the contents of the register (ACC or IX).
JMP	<address>	Jump to the given address.
CMP	<address>	Compare the contents of ACC with the contents of <address>.
CMP	#n	Compare the contents of ACC with number n.
JPE	<address>	Following a compare instruction, jump to <address> if the compare was True.
JPN	<address>	Following a compare instruction, jump to <address> if the compare was False.
AND	#n	Bitwise AND operation of the contents of ACC with the operand.
AND	<address>	Bitwise AND operation of the contents of ACC with the contents of <address>.
XOR	#n	Bitwise XOR operation of the contents of ACC with the operand.
XOR	<address>	Bitwise XOR operation of the contents of ACC with the contents of <address>.
OR	#n	Bitwise OR operation of the contents of ACC with the operand.
OR	<address>	Bitwise OR operation of the contents of ACC with the contents of <address>. <address> can be an absolute address or a symbolic address.
LSL	#n	Bits in ACC are shifted n places to the left. Zeros are introduced on the right hand end.
LSR	#n	Bits in ACC are shifted n places to the right. Zeros are introduced on the left hand end.
IN		Key in a character and store its ASCII value in ACC.
OUT		Output to the screen the character whose ASCII value is stored in ACC.
END		Return control to the operating system.

(a) A programmer needs a program that multiplies a binary number by 4.

The programmer has started to write the program in the following table. The comm
contains explanations for the missing program instructions.

Write the program using the given instruction set.

Label	Instruction		Comment
	Op code	**Operand**	
			`// load contents of NUMBER`
			`// perform shift to multiply by 4`
			`// store contents of ACC in NUMBER`
			`// end program`
`NUMBER:`	`B00110110`		

[5]

Note:

`#` denotes immediate addressing
`B` denotes a binary number, e.g. `B01001010`
`&` denotes a hexadecimal number, e.g. `&4A`

(b) A programmer needs a program that counts the number of lower case letters

The programmer has started to write the program in the following table. The comm
contains explanations for the missing program instructions.

Complete the program using the given instruction set. A copy of the instruction set is provi
on the opposite page.

Label	Instruction		Comment
	Op code	Operand	
	LDR	#0	// initialise Index Register to 0
START:			// load the next value from the STRING
			// perform bitwise AND operation with MASK
			// check if result is equal to MASK
			// if FALSE, jump to UPPER
			// increment COUNT
UPPER:	INC	IX	// increment the Index Register
			// decrement LENGTH
			// is LENGTH = 0 ?
			// if FALSE, jump to START
	END		// end program
MASK:	B00100000		// if bit 5 is 1, letter is lower case
COUNT:	0		
LENGTH:	5		
STRING:	B01001000		// ASCII code for 'H'
	B01100001		// ASCII code for 'a'
	B01110000		// ASCII code for 'p'
	B01110000		// ASCII code for 'p'
	B01011001		// ASCII code for 'Y'

[8]

Instruction		Explanation
Op code	**Operand**	
LDM	#n	Immediate addressing. Load the number n to ACC.
LDD	<address>	Direct addressing. Load the contents of the location at the given addres ACC.
LDI	<address>	Indirect addressing. The address to be used is at the given address. Load the contents of this second address to ACC.
LDX	<address>	Indexed addressing. Form the address from <address> + the contents of the Index Register. Copy the contents of this calculated address to ACC.
LDR	#n	Immediate addressing. Load the number n to IX.
STO	<address>	Store the contents of ACC at the given address.
STX	<address>	Indexed addressing. Form the address from <address> + the contents of the Index Register. Copy the contents from ACC to this calculated address.
ADD	<address>	Add the contents of the given address to the ACC.
INC	<register>	Add 1 to the contents of the register (ACC or IX).
DEC	<register>	Subtract 1 from the contents of the register (ACC or IX).
JMP	<address>	Jump to the given address.
CMP	<address>	Compare the contents of ACC with the contents of <address>.
CMP	#n	Compare the contents of ACC with number n.
JPE	<address>	Following a compare instruction, jump to <address> if the compare was True.
JPN	<address>	Following a compare instruction, jump to <address> if the compare was False.
AND	#n	Bitwise AND operation of the contents of ACC with the operand.
AND	<address>	Bitwise AND operation of the contents of ACC with the contents of <address>.
XOR	#n	Bitwise XOR operation of the contents of ACC with the operand.
XOR	<address>	Bitwise XOR operation of the contents of ACC with the contents of <address>.
OR	#n	Bitwise OR operation of the contents of ACC with the operand.
OR	<address>	Bitwise OR operation of the contents of ACC with the contents of <address>. <address> can be an absolute address or a symbolic address.
LSL	#n	Bits in ACC are shifted n places to the left. Zeros are introduced on the right hand end.
LSR	#n	Bits in ACC are shifted n places to the right. Zeros are introduced on the left hand end.
IN		Key in a character and store its ASCII value in ACC.
OUT		Output to the screen the character whose ASCII value is stored in ACC.
END		Return control to the operating system.

BLANK PAGE

BLANK PAGE