PAPER 4 ALGORITHM AND OOP

COMPUTER SCIENCE

Topic: Paper 4 Algorithm and OOP

INSTRUCTIONS

- Carry out every instruction in each task.
- Answer **all** questions.
- Use a black or dark blue pen.
- You may use an HB pencil for any diagram, graphs or rough working.
- **Calculator Not Allowed**.
- Show your workings if relevant.

INFORMATION

- The number of marks for each question or part question is shown in brackets [ ].

3 An ordered binary tree stores integer data in ascending numerical order.

The data for the binary tree is stored in a 2D array with the following structure:

	LeftPointer	**Data**	**RightPointer**
Index	**[0]**	**[1]**	**[2]**
[0]	1	10	2
[1]	-1	5	-1
[2]	-1	16	-1

Each row in the table represents one node on the tree.
The number `-1` represents a null pointer.

(a) The 2D array, `ArrayNodes`, is declared with space for 20 nodes.

Each node has a left pointer, data and right pointer.

The program also initialises the:

- `RootPointer` to `-1` (null); this points to the first node in the binary tree
- `FreeNode` to `0`; this points to the first empty node in the array.

Write program code to declare `ArrayNodes`, `RootPointer` and `FreeNode` in the main program.

If you are writing in Python programming language, include attribute declarations using comments.

Save your program as **question 3**.

Copy and paste the program code into **part 3(a)** in the evidence document.

[4]

(b) The procedure `AddNode()` adds a new node to the array `ArrayNodes`.

The procedure needs to:

- take the array, root pointer and free node pointer as parameters
- ask the user to enter the data and read this in
- add the node to the root pointer if the tree is empty
- otherwise, follow the pointers to find the position for the data item to be added
- store the data in the location and update all pointers.

1

There are **six** incomplete statements in the following pseudocode for the procedure `AddNode()`.

```
PROCEDURE AddNode(BYREF ArrayNodes[] : ARRAY OF INTEGER,
                  BYREF RootPointer : INTEGER, BYREF FreeNode : INTEGER)
  OUTPUT "Enter the data"
  INPUT NodeData
  IF FreeNode <= 19 THEN
    ArrayNodes[FreeNode, 0] ← -1
    ArrayNodes[FreeNode, 1] ← ..............................
    ArrayNodes[FreeNode, 2] ← -1
    IF RootPointer = .............................. THEN
      RootPointer ← 0
    ELSE
      Placed ← FALSE
      CurrentNode ← RootPointer
      WHILE Placed = FALSE
        IF NodeData < ArrayNodes[CurrentNode, 1] THEN
          IF ArrayNodes[CurrentNode, 0] = -1 THEN
            ArrayNodes[CurrentNode, 0] ← ..............................
            Placed ← TRUE
          ELSE
            .............................. ← ArrayNodes[CurrentNode, 0]
          ENDIF
        ELSE
          IF ArrayNodes[CurrentNode, 2] = -1 THEN
            ArrayNodes[CurrentNode, 2] ← FreeNode
            Placed ← ..............................
          ELSE
            CurrentNode ← ArrayNodes[CurrentNode, 2]
          ENDIF
        ENDIF
      ENDWHILE
    ENDIF
    FreeNode ← .............................. + 1
  ELSE
    OUTPUT "Tree is full"
  ENDIF
ENDPROCEDURE
```

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

Save your program.

Copy and paste the program code into **part 3(b)** in the evidence document.

[8]

(c) The procedure `PrintAll()` outputs the data in each element in `ArrayNodes`, in the order they are stored in the array.

Each element is printed in a row in the order:

```
LeftPointer  Data  RightPointer
```

For example:

```
     1        20       -1
    -1        10       -1
```

Write program code for the procedure `PrintAll()`.

Save your program.

Copy and paste the program code into **part 3(c)** in the evidence document.

[4]

(d) The main program should loop 10 times, each time calling the procedure `AddNode()`. It should then call the procedure `PrintAll()`.

(i) Edit the main program to perform the actions described.

Save your program.

Copy and paste the program code into **part 3(d)(i)** in the evidence document.

[3]

(ii) Test the program by entering the data:

10
5
15
8
12
6
20
11
9
4

Take a screenshot to show the output after the given data are entered.

Copy and paste the screenshot into **part 3(d)(ii)** in the evidence document.

[1]

(e) An in-order tree traversal visits the left node, then the root (and outputs this), then visits the right node.

(i) Write a recursive procedure, `InOrder()`, to perform an in-order traversal on the tree held in `ArrayNodes`.

Save your program.

Copy and paste the program code into **part 3(e)(i)** in the evidence document.

[7]

(ii) Test the procedure `InOrder()` with the same data entered in **part (d)(ii)**.

Take a screenshot to show the output after entering the data.

Copy and paste the screenshot into **part 3(e)(ii)** in the evidence document.

[1]

3 An ordered binary tree stores integer data in ascending numerical order.

The data for the binary tree is stored in a 2D array with the following structure:

	LeftPointer	**Data**	**RightPointer**
Index	**[0]**	**[1]**	**[2]**
[0]	1	10	2
[1]	-1	5	-1
[2]	-1	16	-1

Each row in the table represents one node on the tree.
The number `-1` represents a null pointer.

(a) The 2D array, `ArrayNodes`, is declared with space for 20 nodes.

Each node has a left pointer, data and right pointer.

The program also initialises the:

- `RootPointer` to `-1` (null); this points to the first node in the binary tree
- `FreeNode` to `0`; this points to the first empty node in the array.

Write program code to declare `ArrayNodes`, `RootPointer` and `FreeNode` in the main program.

If you are writing in Python programming language, include attribute declarations using comments.

Save your program as **question 3**.

Copy and paste the program code into **part 3(a)** in the evidence document.

[4]

(b) The procedure `AddNode()` adds a new node to the array `ArrayNodes`.

The procedure needs to:

- take the array, root pointer and free node pointer as parameters
- ask the user to enter the data and read this in
- add the node to the root pointer if the tree is empty
- otherwise, follow the pointers to find the position for the data item to be added
- store the data in the location and update all pointers.

There are **six** incomplete statements in the following pseudocode for the procedure `AddNode()`.

```
PROCEDURE AddNode(BYREF ArrayNodes[] : ARRAY OF INTEGER,
                  BYREF RootPointer : INTEGER, BYREF FreeNode : INTEGER)
  OUTPUT "Enter the data"
  INPUT NodeData
  IF FreeNode <= 19 THEN
    ArrayNodes[FreeNode, 0] ← -1
    ArrayNodes[FreeNode, 1] ← ...............................
    ArrayNodes[FreeNode, 2] ← -1
    IF RootPointer = ............................... THEN
      RootPointer ← 0
    ELSE
      Placed ← FALSE
      CurrentNode ← RootPointer
      WHILE Placed = FALSE
        IF NodeData < ArrayNodes[CurrentNode, 1] THEN
          IF ArrayNodes[CurrentNode, 0] = -1 THEN
            ArrayNodes[CurrentNode, 0] ← ...............................
            Placed ← TRUE
          ELSE
            ............................... ← ArrayNodes[CurrentNode, 0]
          ENDIF
        ELSE
          IF ArrayNodes[CurrentNode, 2] = -1 THEN
            ArrayNodes[CurrentNode, 2] ← FreeNode
            Placed ← ...............................
          ELSE
            CurrentNode ← ArrayNodes[CurrentNode, 2]
          ENDIF
        ENDIF
      ENDWHILE
    ENDIF
    FreeNode ← ............................... + 1
  ELSE
    OUTPUT("Tree is full")
  ENDIF
ENDPROCEDURE
```

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

Save your program.

Copy and paste the program code into **part 3(b)** in the evidence document.

[8]

(c) The procedure `PrintAll()` outputs the data in each element in `ArrayNodes`, in the order they are stored in the array.

Each element is printed in a row in the order:

```
LeftPointer  Data  RightPointer
```

For example:

```
     1        20      -1
    -1        10      -1
```

Write program code for the procedure `PrintAll()`.

Save your program.

Copy and paste the program code into **part 3(c)** in the evidence document.

[4]

(d) The main program should loop 10 times, each time calling the procedure `AddNode()`. It should then call the procedure `PrintAll()`.

(i) Edit the main program to perform the actions described.

Save your program.

Copy and paste the program code into **part 3(d)(i)** in the evidence document.

[3]

(ii) Test the program by entering the data:

10
5
15
8
12
6
20
11
9
4

Take a screenshot to show the output after the given data are entered.

Copy and paste the screenshot into **part 3(d)(ii)** in the evidence document.

[1]

(e) An in-order tree traversal visits the left node, then the root (and outputs this), then visits the right node.

(i) Write a recursive procedure, `InOrder()`, to perform an in-order traversal on the tree held in `ArrayNodes`.

Save your program.

Copy and paste the program code into **part 3(e)(i)** in the evidence document.

[7]

(ii) Test the procedure `InOrder()` with the same data entered in **part (d)(ii)**.

Take a screenshot to show the output after entering the data.

Copy and paste the screenshot into **part 3(e)(ii)** in the evidence document.

[1]

Open the document **evidence.doc**.

Make sure that your name, centre number and candidate number will appear on every page of this document. This document will contain your answers to each question.

Save this evidence document in your work area as:

evidence_ followed by your centre number_candidate number, for example: evidence_zz999_9999

A class declaration can be used to declare a record.

A list is an alternative to an array.

A source file is used to answer question 3. The file is called **`TreasureChestData.txt`**

1 An unordered linked list uses a 1D array to store the data.

Each item in the linked list is of a record type, `node`, with a field `data` and a field `nextNode`.

The current contents of the linked list are:

startPointer	0
emptyList	5

Index	**data**	**nextNode**
0	1	1
1	5	4
2	6	7
3	7	-1
4	2	2
5	0	6
6	0	8
7	56	3
8	0	9
9	0	-1

(a) The following is pseudocode for the record type `node`.

```
TYPE node

     DECLARE data : INTEGER

     DECLARE nextNode : INTEGER

ENDTYPE
```

Write program code to declare the record type `node`.

Save your program as **question1**.

Copy and paste the program code into **part 1(a)** in the evidence document.

[2]

(b) Write program code for the main program.

Declare a 1D array of type `node` with the identifier `linkedList`, and initialise it with the data shown in the table on page 2. Declare the pointers.

Save your program.

Copy and paste the program code into **part 1(b)** in the evidence document.

[4]

(c) The procedure `outputNodes()` takes the array and `startPointer` as parameters. The procedure outputs the data from the linked list by following the `nextNode` values.

(i) Write program code for the procedure `outputNodes()`.

Save your program.

Copy and paste the program code into **part 1(c)(i)** in the evidence document.

[6]

(ii) Edit the main program to call the procedure `outputNodes()`.

Take a screenshot to show the output of the procedure `outputNodes()`.

Save your program.

Copy and paste the screenshot into **part 1(c)(ii)** in the evidence document.

[1]

(d) The function, `addNode()`, takes the linked list and pointers as parameters, then takes as input the data to be added to the end of the `linkedList`.

The function adds the node in the next available space, updates the pointers and returns `True`. If there are no empty nodes, it returns `False`.

(i) Write program code for the function `addNode()`.

Save your program.

Copy and paste the program code into **part 1(d)(i)** in the evidence document.

[7]

(ii) Edit the main program to:

- call `addNode()`
- output an appropriate message depending on the result returned from `addNode()`
- call `outputNodes()` twice; once before calling `addNode()` and once after calling `addNode()`.

Save your program.

Copy and paste the program code into **part 1(d)(ii)** in the evidence document.

[3]

(iii) Test your program by inputting the data value `5` and take a screenshot to show the output.

Save your program.

Copy and paste the screenshot into **part 1(d)(iii)** in the evidence document.

[1]

Open the document **evidence.doc**.

Make sure that your name, centre number and candidate number will appear on every page of this document. This document will contain your answers to each question.

Save this evidence document in your work area as:

evidence_ followed by your centre number_candidate number, for example: evidence_zz999_9999

A class declaration can be used to declare a record.

A list is an alternative to an array.

A source file is used to answer question 3. The file is called **`TreasureChestData.txt`**

1 An unordered linked list uses a 1D array to store the data.

Each item in the linked list is of a record type, `node`, with a field `data` and a field `nextNode`.

The current contents of the linked list are:

startPointer	0
emptyList	5

Index	**data**	**nextNode**
0	1	1
1	5	4
2	6	7
3	7	-1
4	2	2
5	0	6
6	0	8
7	56	3
8	0	9
9	0	-1

(a) The following is pseudocode for the record type `node`.

```
TYPE node
    DECLARE data : INTEGER
    DECLARE nextNode : INTEGER
ENDTYPE
```

Write program code to declare the record type `node`.

Save your program as **question1**.

Copy and paste the program code into **part 1(a)** in the evidence document.

[2]

(b) Write program code for the main program.

Declare a 1D array of type `node` with the identifier `linkedList`, and initialise it with the data shown in the table on page 2. Declare the pointers.

Save your program.

Copy and paste the program code into **part 1(b)** in the evidence document.

[4]

(c) The procedure `outputNodes()` takes the array and `startPointer` as parameters. The procedure outputs the data from the linked list by following the `nextNode` values.

(i) Write program code for the procedure `outputNodes()`.

Save your program.

Copy and paste the program code into **part 1(c)(i)** in the evidence document.

[6]

(ii) Edit the main program to call the procedure `outputNodes()`.

Take a screenshot to show the output of the procedure `outputNodes()`.

Save your program.

Copy and paste the screenshot into **part 1(c)(ii)** in the evidence document.

[1]

(d) The function, `addNode()`, takes the linked list and pointers as parameters, then takes as input the data to be added to the end of the `linkedList`.

The function adds the node in the next available space, updates the pointers and returns `True`. If there are no empty nodes, it returns `False`.

(i) Write program code for the function `addNode()`.

Save your program.

Copy and paste the program code into **part 1(d)(i)** in the evidence document.

[7]

(ii) Edit the main program to:

- call `addNode()`
- output an appropriate message depending on the result returned from `addNode()`
- call `outputNodes()` twice; once before calling `addNode()` and once after calling `addNode()`.

Save your program.

Copy and paste the program code into **part 1(d)(ii)** in the evidence document.

[3]

(iii) Test your program by inputting the data value `5` and take a screenshot to show the output.

Save your program.

Copy and paste the screenshot into **part 1(d)(iii)** in the evidence document.

[1]

Open the document **evidence.doc**.

Make sure that your name, centre number and candidate number will appear on every page of this document. This document will contain your answers to each question.

Save this evidence document in your work area as:

evidence_ followed by your centre number_candidate number, for example: evidence_zz999_9999

A class declaration can be used to declare a record.

A list is an alternative to an array.

A source file is used to answer question 3. The file is called **`TreasureChestData.txt`**

1 An unordered linked list uses a 1D array to store the data.

Each item in the linked list is of a record type, `node`, with a field `data` and a field `nextNode`.

The current contents of the linked list are:

startPointer	0
emptyList	5

Index	**data**	**nextNode**
0	1	1
1	5	4
2	6	7
3	7	-1
4	2	2
5	0	6
6	0	8
7	56	3
8	0	9
9	0	-1

(a) The following is pseudocode for the record type `node`.

```
TYPE node
    DECLARE data : INTEGER
    DECLARE nextNode : INTEGER
ENDTYPE
```

Write program code to declare the record type `node`.

Save your program as **question1**.

Copy and paste the program code into **part 1(a)** in the evidence document.

[2]

(b) Write program code for the main program.

Declare a 1D array of type `node` with the identifier `linkedList`, and initialise it with the data shown in the table on page 2. Declare the pointers.

Save your program.

Copy and paste the program code into **part 1(b)** in the evidence document.

[4]

(c) The procedure `outputNodes()` takes the array and `startPointer` as parameters. The procedure outputs the data from the linked list by following the `nextNode` values.

(i) Write program code for the procedure `outputNodes()`.

Save your program.

Copy and paste the program code into **part 1(c)(i)** in the evidence document.

[6]

(ii) Edit the main program to call the procedure `outputNodes()`.

Take a screenshot to show the output of the procedure `outputNodes()`.

Save your program.

Copy and paste the screenshot into **part 1(c)(ii)** in the evidence document.

[1]

(d) The function, `addNode()`, takes the linked list and pointers as parameters, then takes as input the data to be added to the end of the `linkedList`.

The function adds the node in the next available space, updates the pointers and returns `True`. If there are no empty nodes, it returns `False`.

(i) Write program code for the function `addNode()`.

Save your program.

Copy and paste the program code into **part 1(d)(i)** in the evidence document.

[7]

(ii) Edit the main program to:

- call `addNode()`
- output an appropriate message depending on the result returned from `addNode()`
- call `outputNodes()` twice; once before calling `addNode()` and once after calling `addNode()`.

Save your program.

Copy and paste the program code into **part 1(d)(ii)** in the evidence document.

[3]

(iii) Test your program by inputting the data value `5` and take a screenshot to show the output.

Save your program.

Copy and paste the screenshot into **part 1(d)(iii)** in the evidence document.

[1]

3 A program uses a circular queue to store strings. The queue is created as a 1D array, `QueueArray`, with 10 string items.

The following data is stored about the queue:

- the head pointer initialised to 0
- the tail pointer initialised to 0
- the number of items in the queue initialised to 0.

(a) Declare the array, head pointer, tail pointer and number of items.

If you are writing in Python, include attribute declarations using comments.

Save your program as **Question3_J2022**.

Copy and paste the program code into **part 3(a)** in the evidence document.

[2]

(b) The function `Enqueue` is written in pseudocode. The function adds `DataToAdd` to the queue. It returns `FALSE` if the queue is full and returns `TRUE` if the item is added.

The function is incomplete, there are **five** incomplete statements.

```
FUNCTION Enqueue(BYREF QueueArray[] : STRING, BYREF HeadPointer : INTEGER,
                 BYREF TailPointer : INTEGER, NumberItems : INTEGER,
                 DataToAdd : STRING) RETURNS BOOLEAN

  IF NumberItems = ........................................ THEN

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

  ENDIF

  QueueArray[........................................] ← DataToAdd

  IF TailPointer >= 9 THEN

    TailPointer ← ........................................

  ELSE

    TailPointer ← TailPointer + 1

  ENDIF

  NumberItems ← NumberItems ........................................

  RETURN TRUE

ENDFUNCTION
```

Write program code for the function `Enqueue()`.

Save your program.

Copy and paste the program code into **part 3(b)** in the evidence document.

[7]

(c) The function `Dequeue()` returns `"FALSE"` if the queue is empty, or it returns the next data item in the queue.

Write program code for the function `Dequeue()`.

Save your program.

Copy and paste the program code into **part 3(c)** in the evidence document.

[6]

(d) **(i)** Amend the main program to:

- take as input 11 string values from the user
- use the `Enqueue()` function to add each element to the queue
- output an appropriate message to state whether each addition was successful, or not
- call `Dequeue()` function twice and output the return value each time.

Save your program.

Copy and paste the program code into **part 3(d)(i)** in the evidence document.

[5]

(ii) Test your program with the input data:

```
"A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K"
```

Take a screenshot to show the output.

Copy and paste the screenshot into **part 3(d)(ii)** in the evidence document.

[1]

Open the document **evidence.doc**

Make sure that your name, centre number and candidate number will appear on every page of this document. This document must contain your answers to each question.

Save this evidence document in your work area as:

evidence_ followed by your centre number_candidate number, for example: evidence_zz999_9999

A class declaration can be used to declare a record.
If the programming language used does not support arrays, a list can be used instead.

A source file is used to answer **Question 3**. The file is called **`CardValues.txt`**

1 A program needs to use a stack data structure. The stack can store up to 10 integer elements.

A 1D array `StackData` is used to store the stack globally. The global variable `StackPointer` points to the next available space in the stack and is initialised to 0.

(a) Write program code to declare the array and pointer as global data structures. Initialise the pointer to 0.

> Save your program as **Question1_J22**.
>
> Copy and paste the program code into **part 1(a)** in the evidence document.

[3]

(b) Write a procedure to output all 10 elements in the stack **and** the value of `StackPointer`.

> Save your program.
>
> Copy and paste the program code into **part 1(b)** in the evidence document.

[3]

(c) The function `Push()` takes an integer parameter and returns `FALSE` if the stack is full. If the stack is not full, it puts the parameter value on the stack, updates the relevant pointer and returns `TRUE`.

Write program code for the function `Push()`.

> Save your program.
>
> Copy and paste the program code into **part 1(c)** in the evidence document.

[6]

(d) **(i)** Edit the main program to test the `Push()` function. The main program needs to:

- allow the user to enter 11 numbers and attempt to add these to the stack
- output an appropriate message when a number is added to the stack
- output an appropriate message when a number is not added to the stack if it is full
- output the contents of the stack after attempting to add all 11 numbers.

Save your program.

Copy and paste the program code into **part 1(d)(i)** in the evidence document.

[5]

(ii) Test your program from **part 1(d)(i)** with the following 11 inputs:

11 12 13 14 15 16 17 18 19 20 21

Take a screenshot to show the output.

Copy and paste the screenshot into **part 1(d)(ii)** in the evidence document.

[1]

(e) The function `Pop()` returns `-1` if the stack is empty. If the stack is not empty, it returns the element at the top of the stack and updates the relevant pointer.

(i) Write program code for the function `Pop()`.

Save your program.

Copy and paste the program code into **part 1(e)(i)** in the evidence document.

[5]

(ii) After the code you wrote in the main program for **part 1(d)(i)**, add program code to:

- remove two elements from the stack using `Pop()`
- output the updated contents of the stack.

Test your program and take a screenshot to show the output.

Copy and paste the screenshot into **part 1(e)(ii)** in the evidence document.

[2]

3 A program uses a circular queue to store strings. The queue is created as a 1D array, `QueueArray`, with 10 string items.

The following data is stored about the queue:

- the head pointer initialised to 0
- the tail pointer initialised to 0
- the number of items in the queue initialised to 0.

(a) Declare the array, head pointer, tail pointer and number of items.

If you are writing in Python, include attribute declarations using comments.

Save your program as **Question3_J2022**.

Copy and paste the program code into **part 3(a)** in the evidence document.

[2]

(b) The function Enqueue is written in pseudocode. The function adds DataToAdd to the queue. It returns FALSE if the queue is full and returns TRUE if the item is added.

The function is incomplete, there are **five** incomplete statements.

```
FUNCTION Enqueue(BYREF QueueArray[] : STRING, BYREF HeadPointer : INTEGER,
                 BYREF TailPointer : INTEGER, NumberItems : INTEGER,
                 DataToAdd : STRING) RETURNS BOOLEAN

  IF NumberItems = ........................................ THEN

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

  ENDIF

  QueueArray[........................................] ← DataToAdd

  IF TailPointer >= 9 THEN

    TailPointer ← ........................................

  ELSE

    TailPointer ← TailPointer + 1

  ENDIF

  NumberItems ← NumberItems ........................................

  RETURN TRUE

ENDFUNCTION
```

Write program code for the function Enqueue().

Save your program.

Copy and paste the program code into **part 3(b)** in the evidence document.

[7]

(c) The function Dequeue() returns "FALSE" if the queue is empty, or it returns the next data item in the queue.

Write program code for the function Dequeue().

Save your program.

Copy and paste the program code into **part 3(c)** in the evidence document.

[6]

(d) **(i)** Amend the main program to:

- take as input 11 string values from the user
- use the `Enqueue()` function to add each element to the queue
- output an appropriate message to state whether each addition was successful, or not
- call `Dequeue()` function twice and output the return value each time.

Save your program.

Copy and paste the program code into **part 3(d)(i)** in the evidence document.

[5]

(ii) Test your program with the input data:

```
"A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K"
```

Take a screenshot to show the output.

Copy and paste the screenshot into **part 3(d)(ii)** in the evidence document.

[1]

3 A binary tree consists of nodes. Each node has 3 integer values: a left pointer, data and a right pointer.

The binary tree is stored using a global 2D array.

The pseudocode declaration for the array is:

```
DECLARE ArrayNodes : ARRAY[0:19, 0:2] OF INTEGER
```

For example:

- `ArrayNodes[0, 0]` stores the left pointer for the first node.
- `ArrayNodes[0, 1]` stores the data for the first node.
- `ArrayNodes[0, 2]` stores the right pointer for the first node.

`-1` indicates a null pointer, or null data.

(a) Write program code to:

- declare the global 2D array `ArrayNodes`
- initialise all 3 integer values to `-1` for each node.

Save your program as **Question3_N22**.

Copy and paste the program code into **part 3(a)** in the evidence document.

[3]

(b) The binary tree stores the following values:

Index	Left pointer	Data	Right pointer
0	1	20	5
1	2	15	–1
2	–1	3	3
3	–1	9	4
4	–1	10	–1
5	–1	58	–1
6	–1	–1	–1

`FreeNode` stores the index of the first free element in the array, initialised to 6.

`RootPointer` stores the index of the first node in the tree, initialised to 0.

Amend your program by writing program code to store the given data in `ArrayNodes` **and** initialise the free node and root node pointers.

Save your program.

Copy and paste the program code into **part 3(b)** in the evidence document.

[2]

(c) The following recursive pseudocode function searches the binary tree for a given value. If the value is found, the function must return the index of the value. If the value is not found, the function must return –1.

The function is incomplete. There are **four** incomplete statements.

```
FUNCTION SearchValue(Root : INTEGER,
                     ValueToFind : INTEGER) RETURNS INTEGER

   IF Root = -1 THEN

      RETURN -1

   ELSE

     IF ArrayNodes[Root, 1] = ValueToFind THEN

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

     ELSE

        IF ArrayNodes[Root, 1] = -1 THEN

           RETURN -1

        ENDIF

     ENDIF

   ENDIF

   IF ArrayNodes[Root, 1] ...................................... ValueToFind THEN

      RETURN SearchValue(ArrayNodes[............, 0], ValueToFind)

   ENDIF

   IF ArrayNodes[Root, ............] < ValueToFind THEN

      RETURN SearchValue(ArrayNodes[Root, 2], ValueToFind)

   ENDIF

ENDFUNCTION
```

Write program code for the function `SearchValue()`.

Save your program.

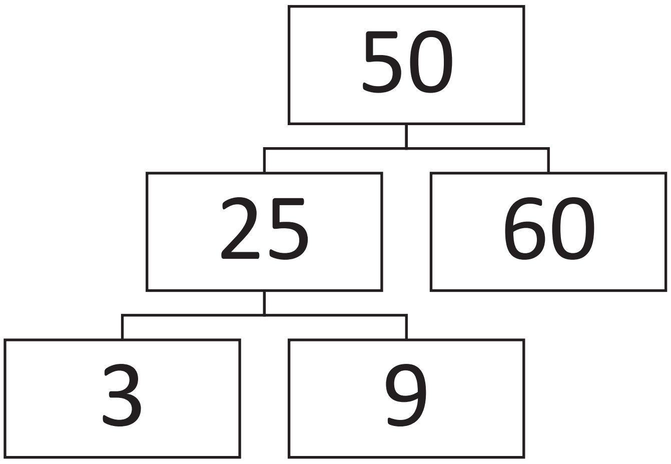
Copy and paste the program code into **part 3(c)** in the evidence document.

[5]

(d) A post order traversal performs the following operation:

- visit the left node
- visit the right node
- output the root.

For example, in the following tree, the output would be: **3 9 25 60 50**



An outline of the `PostOrder()` procedure is:

- If left node is not empty, make a recursive call with the left node as the root.
- If right node is not empty, make a recursive call with the right node as the root.
- Output the current root node.

The procedure `PostOrder()` takes the root node as a parameter.

Write program code for the procedure `PostOrder()`.

Save your program.

Copy and paste the program code into **part 3(d)** in the evidence document.

[7]

(e) **(i)** Amend the main program by writing program code to:

- call the function `SearchValue()` to find the position of the number 15 in the tree
- use the result from `SearchValue()` to output either the index of the value if found, or an appropriate message to state that the value was not found
- call the procedure `PostOrder()`.

Save your program.

Copy and paste the program code into **part 3(e)(i)** in the evidence document.

[3]

(ii) Test your program.

Take a screenshot to show the output.

Copy and paste the screenshot into **part 3(e)(ii)** in the evidence document.

[1]

3 A binary tree consists of nodes. Each node has 3 integer values: a left pointer, data and a right pointer.

The binary tree is stored using a global 2D array.

The pseudocode declaration for the array is:

```
DECLARE ArrayNodes : ARRAY[0:19, 0:2] OF INTEGER
```

For example:

- `ArrayNodes[0, 0]` stores the left pointer for the first node.
- `ArrayNodes[0, 1]` stores the data for the first node.
- `ArrayNodes[0, 2]` stores the right pointer for the first node.

`–1` indicates a null pointer, or null data.

(a) Write program code to:

- declare the global 2D array `ArrayNodes`
- initialise all 3 integer values to `–1` for each node.

Save your program as **Question3_N22**.

Copy and paste the program code into **part 3(a)** in the evidence document.

[3]

(b) The binary tree stores the following values:

Index	Left pointer	Data	Right pointer
0	1	20	5
1	2	15	–1
2	–1	3	3
3	–1	9	4
4	–1	10	–1
5	–1	58	–1
6	–1	–1	–1

`FreeNode` stores the index of the first free element in the array, initialised to 6.

`RootPointer` stores the index of the first node in the tree, initialised to 0.

Amend your program by writing program code to store the given data in `ArrayNodes` **and** initialise the free node and root node pointers.

Save your program.

Copy and paste the program code into **part 3(b)** in the evidence document.

[2]

(c) The following recursive pseudocode function searches the binary tree for a given value. If the value is found, the function must return the index of the value. If the value is not found, the function must return –1.

The function is incomplete. There are **four** incomplete statements.

```
FUNCTION SearchValue(Root : INTEGER,
                     ValueToFind : INTEGER) RETURNS INTEGER

   IF Root = −1 THEN

      RETURN −1

   ELSE

     IF ArrayNodes[Root, 1] = ValueToFind THEN

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

     ELSE

        IF ArrayNodes[Root, 1] = −1 THEN

           RETURN −1

        ENDIF

     ENDIF

   ENDIF

   IF ArrayNodes[Root, 1] ..................................... ValueToFind THEN

      RETURN SearchValue(ArrayNodes[..........., 0], ValueToFind)

   ENDIF

   IF ArrayNodes[Root, ...........] < ValueToFind THEN

      RETURN SearchValue(ArrayNodes[Root, 2], ValueToFind)

   ENDIF

ENDFUNCTION
```

Write program code for the function `SearchValue()`.

Save your program.

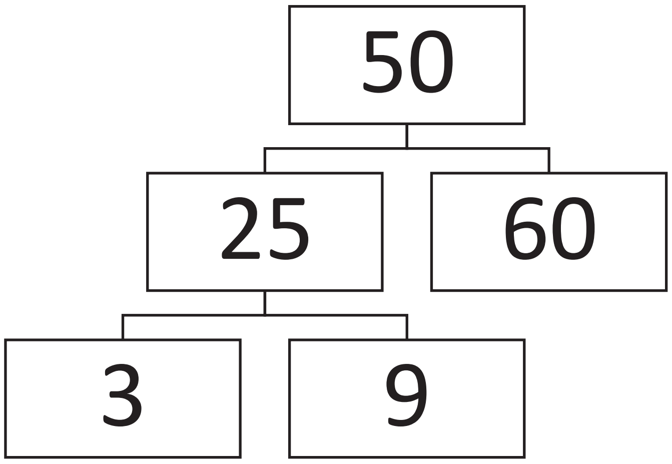
Copy and paste the program code into **part 3(c)** in the evidence document.

[5]

(d) A post order traversal performs the following operation:

- visit the left node
- visit the right node
- output the root.

For example, in the following tree, the output would be: **3 9 25 60 50**



An outline of the `PostOrder()` procedure is:

- If left node is not empty, make a recursive call with the left node as the root.
- If right node is not empty, make a recursive call with the right node as the root.
- Output the current root node.

The procedure `PostOrder()` takes the root node as a parameter.

Write program code for the procedure `PostOrder()`.

Save your program.

Copy and paste the program code into **part 3(d)** in the evidence document.

[7]

(e) **(i)** Amend the main program by writing program code to:

- call the function `SearchValue()` to find the position of the number 15 in the tree
- use the result from `SearchValue()` to output either the index of the value if found, or an appropriate message to state that the value was not found
- call the procedure `PostOrder()`.

Save your program.

Copy and paste the program code into **part 3(e)(i)** in the evidence document.

[3]

(ii) Test your program.

Take a screenshot to show the output.

Copy and paste the screenshot into **part 3(e)(ii)** in the evidence document.

[1]

2 A linear queue is implemented using the 1D array, `Queue`. The index of the first element in the array is 0.

(a) **(i)** Write program code to declare:

- `Queue` — a global array with space to store 50 IDs of type string
- `HeadPointer` — a global variable to point to the first element in the queue, initialised to −1
- `TailPointer` — a global variable to point to the next available space in the queue, initialised to 0.

Save your program as **Question2_N23**.

Copy and paste the program code into part **2(a)(i)** in the evidence document.

[2]

(ii) The procedure `Enqueue()` takes a string parameter.

If the queue is full, the procedure outputs a suitable message. If the queue is not full, the procedure inserts the parameter into the queue and updates the relevant pointer(s).

Write program code for `Enqueue()`.

Save your program.

Copy and paste the program code into part **2(a)(ii)** in the evidence document.

[4]

(iii) The function `Dequeue()` checks if the queue is empty.

If the queue is empty, the function outputs a suitable message and returns the string `"Empty"`.
If the queue is not empty, the function returns the first element in the queue and updates the relevant pointer(s).

Write program code for `Dequeue()`.

Save your program.

Copy and paste the program code into part **2(a)(iii)** in the evidence document.

[4]

(b) A shop sells computer games. Each game has a unique identifier (ID) of string data type.

The text file `QueueData.txt` contains a list of game IDs.

The procedure `ReadData()` reads the data from the text file and inserts each item of data into the array `Queue`.

Write program code for the procedure `ReadData()`.

Save your program.

Copy and paste the program code into part **2(b)** in the evidence document.

[6]

(c) Some game IDs appear in the text file more than once.

The program needs to total the number of times each game ID appears in the text file.

The record structure `RecordData` has the following fields:

- `ID` — a string to store the game ID
- `Total` — an integer to store the total number of times that game ID appears in the text file.

(i) Write program code to declare the record structure `RecordData`.

If you are writing in Python, include attribute declarations as comments.

Save your program.

Copy and paste the program code into part **2(c)(i)** in the evidence document.

[2]

(ii) The global 1D array `Records` stores up to 50 items of type `RecordData`.

The global variable `NumberRecords` stores the number of records currently in the array `Records` and is initialised to 0.

Write program code to declare `Records` and `NumberRecords`.

If you are writing in Python, include attribute declarations as comments.

Save your program.

Copy and paste the program code into part **2(c)(ii)** in the evidence document.

[2]

(iii) The pseudocode algorithm for the procedure `TotalData()`:

- uses `Dequeue()` to remove an ID from the queue
- checks whether a `RecordData` with the returned ID exists in `Records`
- increments the total for that ID in the record if the ID already exists
- creates a new record and stores it in `Records` if the ID does not exist.

```
PROCEDURE TotalData()
   DECLARE DataAccessed : STRING
   DECLARE Flag : BOOLEAN
   DataAccessed ← Dequeue()
   Flag ← FALSE
   IF NumberRecords = 0 THEN
      Records[NumberRecords].ID ← DataAccessed
      Records[NumberRecords].Total ← 1
      Flag ← TRUE
      NumberRecords ← NumberRecords + 1
    ELSE
      FOR X ← 0 TO NumberRecords - 1
         IF Records[X].ID = DataAccessed THEN
            Records[X].Total ← Records[X].Total + 1
            Flag ← TRUE
         ENDIF
       NEXT X
   ENDIF
   IF Flag = FALSE THEN
      Records[NumberRecords].ID ← DataAccessed
      Records[NumberRecords].Total ← 1
      NumberRecords ← NumberRecords + 1
   ENDIF
ENDPROCEDURE
```

Write program code for the procedure `TotalData()`.

Save your program.

Copy and paste the program code into part **2(c)(iii)** in the evidence document.

[5]

(d) The procedure `OutputRecords()` outputs the ID and total of each record in `Records` in the format:

```
ID  1234   Total   4
```

Write program code for `OutputRecords()`.

Save your program.

Copy and paste the program code into part **2(d)** in the evidence document.

[1]

(e) The main program needs to:

- call `ReadData()`
- call `TotalData()` for each element in the queue
- call `OutputRecords()`.

(i) Write program code for the main program.

Save your program.

Copy and paste the program code into part **2(e)(i)** in the evidence document.

[2]

(ii) Test your program.

Take a screenshot of the output.

Save your program.

Copy and paste the screenshot into part **2(e)(ii)** in the evidence document.

[1]

Open the evidence document, **evidence.doc**

Make sure that your name, centre number and candidate number appear on every page of this document. This document must contain your answers to each question.

Save this evidence document in your work area as:

evidence_ followed by your centre number_candidate number, for example: **evidence_zz999_9999**

A class declaration can be used to declare a record.

If the programming language used does not support arrays, a list can be used instead.

One source file is used to answer **Question 1.** The file is called `StackData.txt`

1 A program stores lower-case letters in two stacks.

One stack stores vowels (a, e, i, o, u) and one stack stores consonants (letters that are not vowels).

Each stack is implemented as a 1D array.

(a) **(i)** Write program code to declare two 1D global arrays: `StackVowel` and `StackConsonant`.

Each array needs to store up to 100 letters. The index of the first element in each array is 0.

If you are writing in Python, include declarations using comments.

Save your program as **Question1_N23**.

Copy and paste the program code into part **1(a)(i)** in the evidence document.

[2]

(ii) The global variable `VowelTop` is a pointer that stores the index of the next free space in `StackVowel`.

The global variable `ConsonantTop` is a pointer that stores the index of the next free space in `StackConsonant`.

`VowelTop` and `ConsonantTop` are both initialised to 0.

Write program code to declare and initialise the two variables.

If you are writing in Python, include declarations using comments.

Save your program.

Copy and paste the program code into part **1(a)(ii)** in the evidence document.

[1]

(b) **(i)** The procedure `PushData()` takes one letter as a parameter.

If the parameter is a vowel, it is pushed onto `StackVowel` and the relevant pointer updated.

If the stack is full, a suitable message is output.

If the parameter is a consonant, it is pushed onto `StackConsonant` and the relevant pointer updated.

If the stack is full, a suitable message is output.

You do **not** need to validate that the parameter is a letter.

Write program code for `PushData()`.

Save your program.

Copy and paste the program code into part **1(b)(i)** in the evidence document.

[6]

(ii) The file `StackData.txt` stores 100 lower-case letters.

The procedure `ReadData()` reads each letter from the file and uses `PushData()` to push each letter onto its appropriate stack.

Use appropriate exception handling if the file does not exist.

Write program code for `ReadData()`.

Save your program.

Copy and paste the program code into part **1(b)(ii)** in the evidence document.

[6]

(c) The function `PopVowel()` removes and returns the data at the top of `StackVowel` and updates the relevant pointer(s).

The function `PopConsonant()` removes and returns the data from the top of `StackConsonant` and updates the relevant pointer(s).

If either stack is empty, the string `"No data"` must be returned.

Write program code to declare `PopVowel()` and `PopConsonant()`.

Save your program.

Copy and paste the program code into part **1(c)** in the evidence document.

[5]

(d) The program first needs to call `ReadData()` and then:

1. prompt the user to input their choice of vowel or consonant
2. take, as input, the user's choice
3. depending on the user's choice, call `PopVowel()` or `PopConsonant()` and store the return value.

The three steps are repeated until 5 letters have been successfully returned and stored.

If either stack is empty at any stage, an appropriate message must be output.

Once 5 letters have been successfully returned and stored, they are output on one line, for example:

```
abyti
```

(i) Write program code for the main program.

Save your program.

Copy and paste the program code into part **1(d)(i)** in the evidence document.

[6]

(ii) Test your program with the following inputs:

```
vowel

consonant

consonant

vowel

vowel
```

Take a screenshot of the output.

Save your program.

Copy and paste the screenshot into part **1(d)(ii)** in the evidence document.

[1]

2 A linear queue is implemented using the 1D array, `Queue`. The index of the first element in the array is 0.

(a) (i) Write program code to declare:

- `Queue` — a global array with space to store 50 IDs of type string
- `HeadPointer` — a global variable to point to the first element in the queue, initialised to −1
- `TailPointer` — a global variable to point to the next available space in the queue, initialised to 0.

Save your program as **Question2_N23**.

Copy and paste the program code into part **2(a)(i)** in the evidence document.

[2]

(ii) The procedure `Enqueue()` takes a string parameter.

If the queue is full, the procedure outputs a suitable message. If the queue is not full, the procedure inserts the parameter into the queue and updates the relevant pointer(s).

Write program code for `Enqueue()`.

Save your program.

Copy and paste the program code into part **2(a)(ii)** in the evidence document.

[4]

(iii) The function `Dequeue()` checks if the queue is empty.

If the queue is empty, the function outputs a suitable message and returns the string `"Empty"`.
If the queue is not empty, the function returns the first element in the queue and updates the relevant pointer(s).

Write program code for `Dequeue()`.

Save your program.

Copy and paste the program code into part **2(a)(iii)** in the evidence document.

[4]

(b) A shop sells computer games. Each game has a unique identifier (ID) of string data type.

The text file `QueueData.txt` contains a list of game IDs.

The procedure `ReadData()` reads the data from the text file and inserts each item of data into the array `Queue`.

Write program code for the procedure `ReadData()`.

Save your program.

Copy and paste the program code into part **2(b)** in the evidence document.

[6]

(c) Some game IDs appear in the text file more than once.

The program needs to total the number of times each game ID appears in the text file.

The record structure `RecordData` has the following fields:

- `ID` — a string to store the game ID
- `Total` — an integer to store the total number of times that game ID appears in the text file.

(i) Write program code to declare the record structure `RecordData`.

If you are writing in Python, include attribute declarations as comments.

Save your program.

Copy and paste the program code into part **2(c)(i)** in the evidence document.

[2]

(ii) The global 1D array `Records` stores up to 50 items of type `RecordData`.

The global variable `NumberRecords` stores the number of records currently in the array `Records` and is initialised to 0.

Write program code to declare `Records` and `NumberRecords`.

If you are writing in Python, include attribute declarations as comments.

Save your program.

Copy and paste the program code into part **2(c)(ii)** in the evidence document.

[2]

(iii) The pseudocode algorithm for the procedure `TotalData()`:

- uses `Dequeue()` to remove an ID from the queue
- checks whether a `RecordData` with the returned ID exists in `Records`
- increments the total for that ID in the record if the ID already exists
- creates a new record and stores it in `Records` if the ID does not exist.

```
PROCEDURE TotalData()
   DECLARE DataAccessed : STRING
   DECLARE Flag : BOOLEAN
   DataAccessed ← Dequeue()
   Flag ← FALSE
   IF NumberRecords = 0 THEN
      Records[NumberRecords].ID ← DataAccessed
      Records[NumberRecords].Total ← 1
      Flag ← TRUE
      NumberRecords ← NumberRecords + 1
    ELSE
      FOR X ← 0 TO NumberRecords - 1
         IF Records[X].ID = DataAccessed THEN
            Records[X].Total ← Records[X].Total + 1
            Flag ← TRUE
         ENDIF
       NEXT X
   ENDIF
   IF Flag = FALSE THEN
      Records[NumberRecords].ID ← DataAccessed
      Records[NumberRecords].Total ← 1
      NumberRecords ← NumberRecords + 1
   ENDIF
ENDPROCEDURE
```

Write program code for the procedure `TotalData()`.

Save your program.

Copy and paste the program code into part **2(c)(iii)** in the evidence document.

[5]

(d) The procedure `OutputRecords()` outputs the ID and total of each record in `Records` in the format:

```
ID  1234   Total   4
```

Write program code for `OutputRecords()`.

Save your program.

Copy and paste the program code into part **2(d)** in the evidence document.

[1]

(e) The main program needs to:

- call `ReadData()`
- call `TotalData()` for each element in the queue
- call `OutputRecords()`.

(i) Write program code for the main program.

Save your program.

Copy and paste the program code into part **2(e)(i)** in the evidence document.

[2]

(ii) Test your program.

Take a screenshot of the output.

Save your program.

Copy and paste the screenshot into part **2(e)(ii)** in the evidence document.

[1]

3 A program reads data from the user and stores the data that is valid in a linear queue.

The queue is stored as a global 1D array, `QueueData`, of string values. The array needs space for 20 elements.

The global variable `QueueHead` stores the index of the first element in the queue.
The global variable `QueueTail` stores the index of the last element in the queue.

(a) The main program initialises all the elements in `QueueData` to a suitable null value, `QueueHead` to −1 and `QueueTail` to −1.

Write program code for the main program.

Save your program as **Question3_J24**.

Copy and paste the program code into part **3(a)** in the evidence document.

[1]

(b) The function `Enqueue()` takes the data to insert into the queue as a parameter.

If the queue is **not** full, it inserts the parameter in the queue, updates the appropriate pointer(s) and returns `TRUE`. If the queue is full, it returns `FALSE`.

Write program code for `Enqueue()`.

Save your program.

Copy and paste the program code into part **3(b)** in the evidence document.

[4]

(c) The function `Dequeue()` returns `"false"` if the queue is empty. If the queue is **not** empty, it returns the next item in the queue and updates the appropriate pointer(s).

Write program code for `Dequeue()`.

Save your program.

Copy and paste the program code into part **3(c)** in the evidence document.

[3]

(d) The string values to be stored in the queue are 7 characters long. The first 6 characters are digits and the 7th character is a check digit. The check digit is calculated from the first 6 digits using this algorithm:

- multiply the digits in position 0, position 2 and position 4 by 1
- multiply the digits in position 1, position 3 and position 5 by 3
- calculate the sum of the products (add together the results from all of the multiplications)
- divide the sum of the products by 10 and round the result down to the nearest integer to get the check digit
- if the check digit equals 10 then it is replaced with '`X`'.

Example:

Data is 954123

Character position	**0**	**1**	**2**	**3**	**4**	**5**
Digit	9	5	4	1	2	3
Multiplier	1	3	1	3	1	3
Product	9	15	4	3	2	9

Sum of products = 9 + 15 + 4 + 3 + 2 + 9 = 42

Divide sum of products by 10: 42 / 10 = 4 (rounded down)

The check digit = 4. This is inserted into character position 6.

The data including the check digit is: **9541234**

A 7-character string is valid if the 7th character matches the check digit for that data. For example, the data 9541235 is invalid because the 7th character (5) does **not** match the check digit for 954123.

(i) The subroutine `StoreItems()` takes ten 7-character strings as input from the user and uses the check digit to validate each input.

Each valid input has the check digit removed and is stored in the queue using `Enqueue()`.
An appropriate message is output if the item is inserted. An appropriate message is output if the queue is already full.

Invalid inputs are **not** stored in the queue.

The subroutine counts and outputs the number of invalid items that were entered.

`StoreItems()` can be a procedure or a function as appropriate.

Write program code for `StoreItems()`.

Save your program. Copy and paste the program code into part **3(d)(i)** in the evidence document.

[6]

(ii) Write program code to amend the main program to:

- call `StoreItems()`
- call `Dequeue()`
- output a suitable message if the queue was empty
- output the returned value if the queue was **not** empty.

Save your program.

Copy and paste the program code into part **3(d)(ii)** in the evidence document.

[1]

(iii) Test the program with the following inputs in the order given:

999999X

1251484

5500212

0033585

9845788

6666666

3258746

8111022

7568557

0012353

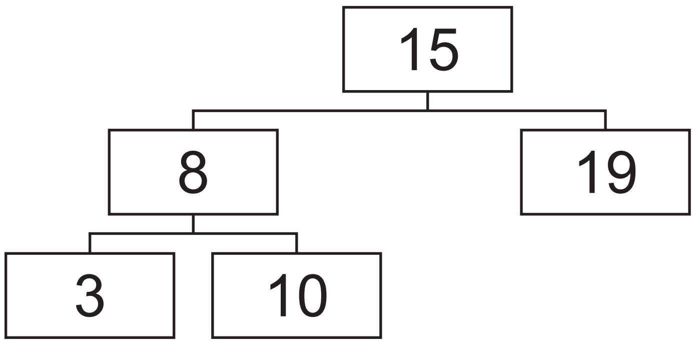
Take a screenshot of the output(s).

Save your program.

Copy and paste the screenshot into part **3(d)(iii)** in the evidence document.

[2]

2 A binary tree stores data in ascending order. For example:



A computer program stores integers in a binary tree in ascending order. The program uses Object-Oriented Programming (OOP).

The binary tree is stored as a 1D array of nodes. Each node contains a left pointer, a data value and a right pointer.

The class `Node` stores the data about a node.

`Node`	
`LeftPointer : INTEGER`	stores the index of the node to the left in the binary tree
`Data : INTEGER`	stores the node's data
`RightPointer : INTEGER`	stores the index of the node to the right in the binary tree
`Constructor()`	initialises `Data` to its parameter value initialises `LeftPointer` and `RightPointer` to −1
`GetLeft()`	returns the left pointer
`GetRight()`	returns the right pointer
`GetData()`	returns the data value
`SetLeft()`	assigns the parameter to the left pointer
`SetRight()`	assigns the parameter to the right pointer
`SetData()`	assigns the parameter to the data

(a) (i) Write program code to declare the class `Node` and its constructor.

Do **not** declare the other methods.

Use the appropriate constructor for your programming language.

If you are writing in Python, include attribute declarations using comments.

Save your program as **Question2_J24**.

Copy and paste the program code into part **2(a)(i)** in the evidence document.

[4]

(ii) The get methods `GetLeft()`, `GetRight()` and `GetData()` each return the relevant attribute.

Write program code for the **three** get methods.

Save your program.

Copy and paste the program code into part **2(a)(ii)** in the evidence document.

[3]

(iii) The set methods `SetLeft()`, `SetRight()` and `SetData()` each take a parameter and then store this in the relevant attribute.

Write program code for the **three** set methods.

Save your program.

Copy and paste the program code into part **2(a)(iii)** in the evidence document.

[3]

(b) The class `TreeClass` stores the data about the binary tree.

`TreeClass`	
`Tree[0:19] : Node`	an array of 20 elements of type `Node`
`FirstNode : INTEGER`	stores the index of the first node in the tree
`NumberNodes : INTEGER`	stores the quantity of nodes in the tree
`Constructor()`	initialises `FirstNode` to −1 and `NumberNodes` to 0 initialises each element in `Tree` to a `Node` object with the data value of −1
`InsertNode()`	takes a `Node` object as a parameter, inserts it in the array and updates the pointer for its parent node
`OutputTree()`	outputs the left pointer, data and right pointer of each node in `Tree`

Nodes cannot be deleted from this tree.

(i) Write program code to declare the class `TreeClass` and its constructor.

Do **not** declare the other methods.

Use the appropriate constructor for your programming language.

If you are writing in Python, include attribute declarations using comments.

Save your program.

Copy and paste the program code into part **2(b)(i)** in the evidence document.

[4]

(ii) The method `InsertNode()` takes a `Node` object, `NewNode`, as a parameter and inserts it into the array `Tree`.

`InsertNode()` first checks if the tree is empty.

If the tree is empty, `InsertNode()`:

- stores `NewNode` in the array `Tree` at index `NumberNodes`
- increments `NumberNodes`
- stores 0 in `FirstNode`.

If the tree is **not** empty, `InsertNode()`:

- stores `NewNode` in the array `Tree` at index `NumberNodes`
- accesses the data in the array `Tree` at index `FirstNode` and compares it to the data in `NewNode`
- repeatedly follows the pointers until the correct position for `NewNode` is found
- once the position is found, `InsertNode()` sets the left or right pointer of its parent node
- increments `NumberNodes`.

Write program code for `InsertNode()`.

Save your program.

Copy and paste the program code into part **2(b)(ii)** in the evidence document.

[6]

(iii) The method `OutputTree()` outputs the left pointer, the data and the right pointer for each node that has been inserted into the tree. The outputs are in the order they are saved in the array.

If there are no nodes in the array, the procedure outputs 'No nodes'.

Write program code for `OutputTree()`.

Save your program.

Copy and paste the program code into part **2(b)(iii)** in the evidence document.

[4]

(c) **(i)** The main program declares an instance of `TreeClass` with the identifier `TheTree`.

Write program code for the main program.

Save your program.

Copy and paste the program code into part **2(c)(i)** in the evidence document.

[1]

(ii) The main program inserts the following integers into the binary tree in the order given:

10

11

5

1

20

7

15

The main program then calls the method `OutputTree()`.

Write program code to amend the main program.

Save your program.

Copy and paste the program code into part **2(c)(ii)** in the evidence document.

[4]

(iii) Test your program.

Take a screenshot of the output(s).

Save your program.

Copy and paste the screenshot into part **2(c)(iii)** in the evidence document.

[1]

3 A program reads data from the user and stores the data that is valid in a linear queue.

The queue is stored as a global 1D array, `QueueData`, of string values. The array needs space for 20 elements.

The global variable `QueueHead` stores the index of the first element in the queue.
The global variable `QueueTail` stores the index of the last element in the queue.

(a) The main program initialises all the elements in `QueueData` to a suitable null value, `QueueHead` to −1 and `QueueTail` to −1.

Write program code for the main program.

Save your program as **Question3_J24**.

Copy and paste the program code into part **3(a)** in the evidence document.

[1]

(b) The function `Enqueue()` takes the data to insert into the queue as a parameter.

If the queue is **not** full, it inserts the parameter in the queue, updates the appropriate pointer(s) and returns `TRUE`. If the queue is full, it returns `FALSE`.

Write program code for `Enqueue()`.

Save your program.

Copy and paste the program code into part **3(b)** in the evidence document.

[4]

(c) The function `Dequeue()` returns `"false"` if the queue is empty. If the queue is **not** empty, it returns the next item in the queue and updates the appropriate pointer(s).

Write program code for `Dequeue()`.

Save your program.

Copy and paste the program code into part **3(c)** in the evidence document.

[3]

(d) The string values to be stored in the queue are 7 characters long. The first 6 characters are digits and the 7th character is a check digit. The check digit is calculated from the first 6 digits using this algorithm:

- multiply the digits in position 0, position 2 and position 4 by 1
- multiply the digits in position 1, position 3 and position 5 by 3
- calculate the sum of the products (add together the results from all of the multiplications)
- divide the sum of the products by 10 and round the result down to the nearest integer to get the check digit
- if the check digit equals 10 then it is replaced with '`X`'.

Example:

Data is 954123

Character position	**0**	**1**	**2**	**3**	**4**	**5**
Digit	9	5	4	1	2	3
Multiplier	1	3	1	3	1	3
Product	9	15	4	3	2	9

Sum of products = 9 + 15 + 4 + 3 + 2 + 9 = 42

Divide sum of products by 10: 42 / 10 = 4 (rounded down)

The check digit = 4. This is inserted into character position 6.

The data including the check digit is: **9541234**

A 7-character string is valid if the 7th character matches the check digit for that data. For example, the data 9541235 is invalid because the 7th character (5) does **not** match the check digit for 954123.

(i) The subroutine `StoreItems()` takes ten 7-character strings as input from the user and uses the check digit to validate each input.

Each valid input has the check digit removed and is stored in the queue using `Enqueue()`.
An appropriate message is output if the item is inserted. An appropriate message is output if the queue is already full.

Invalid inputs are **not** stored in the queue.

The subroutine counts and outputs the number of invalid items that were entered.

`StoreItems()` can be a procedure or a function as appropriate.

Write program code for `StoreItems()`.

Save your program.

Copy and paste the program code into part **3(d)(i)** in the evidence document.

[6]

(ii) Write program code to amend the main program to:

- call `StoreItems()`
- call `Dequeue()`
- output a suitable message if the queue was empty
- output the returned value if the queue was **not** empty.

Save your program.

Copy and paste the program code into part **3(d)(ii)** in the evidence document.

[1]

(iii) Test the program with the following inputs in the order given:

999999X

1251484

5500212

0033585

9845788

6666666

3258746

8111022

7568557

0012353

Take a screenshot of the output(s).

Save your program.

Copy and paste the screenshot into part **3(d)(iii)** in the evidence document.

[2]

3 A linked list stores positive integer data in a 2D array. The first dimension of the array stores the integer data. The second dimension of the array stores the pointer to the next node in the linked list.

A linked list node with no data is initialised with the integer `-1`. These nodes are linked together as an empty list. A pointer of `-1` identifies that node as the last node.

The linked list can store 20 nodes.

The global 2D array `LinkedList` stores the linked list.

`LinkedList` is initialised as an empty list. The data in each node is initialised to `-1`. Each node's pointer stores the index of the next node. The last node stores the pointer value `-1`, which indicates it is the last node.

The global variable `FirstEmpty` stores the index of the first element in the empty list. This is the first node in the empty linked list when it is initialised, which is index `0`.

The global variable `FirstNode` stores the index of the first element in the linked list. There is no data in the linked list when it is initialised, so `FirstNode` is initialised to `-1`.

This diagram shows the content of the initialised array.

```
FirstEmpty = 0

FirstNode = -1
```

Index	**Data**	**Pointer**
0	-1	1
1	-1	2
2	-1	3
3	-1	4
4	-1	5
⌇	⌇	⌇
19	-1	-1

(a) Write program code for the main program to declare and initialise `LinkedList`, `FirstNode` and `FirstEmpty`.

Save your program as **Question3_N24**.

Copy and paste the program code into part **3(a)** in the evidence document.

[2]

(b) The procedure `InsertData()` takes **five** positive integers as input from the user and inserts these into the linked list.

Each data item is inserted at the front of the linked list.

The table shows the steps to follow depending on the state of the linked list:

Linked list state	**Steps**
not full	insert the data in the index pointed to by `FirstEmpty` change the pointer to the index pointed to by `FirstNode` change the values of `FirstNode` and `FirstEmpty`
full	end the procedure

Any node that is at the end of the linked list has a pointer of `−1`.

Write program code for `InsertData()`.

Save your program.

Copy and paste the program code into part **3(b)** in the evidence document.

[6]

(c) The procedure `OutputLinkedList()` outputs the data in the linked list in order by following the pointers from `FirstNode`.

(i) Write program code for `OutputLinkedList()`.

Save your program.

Copy and paste the program code into part **3(c)(i)** in the evidence document.

[2]

(ii) Amend the main program to call `InsertData()` and then `OutputLinkedList()`.

Save your program.

Copy and paste the program code into part **3(c)(ii)** in the evidence document.

[1]

(iii) Test your program with the test data:

5 1 2 3 8

Take a screenshot of the output.

Save your program.

Copy and paste the screenshot into part **3(c)(iii)** in the evidence document.

[1]

(d) The procedure `RemoveData()` removes a node from the linked list.

The procedure takes the data item to be removed from the linked list as a parameter.

The procedure checks each node in the linked list, starting with the node `FirstNode`, until it finds the node to be removed. This node is added to the empty list, and pointers are changed as appropriate. The procedure only removes the first occurrence of the parameter.

Assume that the data item being removed is in the linked list.

(i) Write program code for `RemoveData()`.

Save your program.

Copy and paste the program code into part **3(d)(i)** in the evidence document.

[5]

(ii) Amend the main program to:

- call `RemoveData()` with the parameter 5
- output the word `"After"`
- call `OutputLinkedList()`.

Save your program.

Copy and paste the program code into part **3(d)(ii)** in the evidence document.

[1]

(iii) Test your program with both sets of given test data:

Test data set 1: 5 6 8 9 5

Test data set 2: 10 7 8 5 6

Take a screenshot of each output.

Save your program.

Copy and paste the screenshot(s) into part **3(d)(iii)** in the evidence document.

[1]

2 A linear queue data structure is designed using a record structure.

The record structure `Queue` has the following fields:

- `QueueArray`, a 1D array of up to 100 integer values
- `Headpointer`, a variable that stores the index of the first data item in `QueueArray`
- `Tailpointer`, a variable that stores the index of the next free location in `QueueArray`.

(a) Write program code to declare the record structure `Queue` and its fields.

If your programming language does **not** support record structures, a class can be declared instead.

If you are writing in Python, use comments to declare the appropriate data types.

Save your program as **Question2_N24**.

Copy and paste the program code into part **2(a)** in the evidence document.

[3]

(b) The main program creates a new `Queue` record with the identifier `TheQueue`. The head pointer is initialised to –1. The tail pointer is initialised to 0. Each element in the array is initialised with –1.

Write program code for the main program.

Save your program.

Copy and paste the program code into part **2(b)** in the evidence document.

[3]

(c) The pseudocode function `Enqueue()` inserts an integer value into the queue.

The function is incomplete. There are **three** incomplete statements.

```
FUNCTION Enqueue(BYREF AQueue : Queue, BYVAL TheData : INTEGER)
                  RETURNS INTEGER

   IF AQueue.Headpointer = -1 THEN

      AQueue.QueueArray[AQueue.Tailpointer] ← ......................................

      AQueue.Headpointer ← 0

      AQueue.Tailpointer ← AQueue.Tailpointer + 1

      RETURN 1

   ELSE

      IF AQueue.Tailpointer > ...................................... THEN

         RETURN -1

      ELSE

         AQueue.QueueArray[AQueue.Tailpointer] ← TheData

         AQueue.Tailpointer ← AQueue.Tailpointer ......................................

         RETURN 1

      ENDIF

   ENDIF

ENDFUNCTION
```

Write program code for `Enqueue()`.

Save your program.

Copy and paste the program code into part **2(c)** in the evidence document.

[5]

(d) The function `ReturnAllData()` accesses `TheQueue`. It concatenates all the integer values that have been inserted into the queue's array, starting from the value stored at `HeadPointer`, with a space between each integer value. The string of concatenated values is returned.

None of the integer values are removed from the queue.

Write program code for `ReturnAllData()`.

Save your program.

Copy and paste the program code into part **2(d)** in the evidence document.

[3]

(e) **(i)** The main program asks the user to enter 10 integers with values of 0 or greater. It reads each input repeatedly until a valid number is entered.

All 10 valid inputs are added to the queue, using `Enqueue()`.

If the value returned from `Enqueue()` is –1, a message is output to state that the queue is full, otherwise a message is output to state that the item has been added to the queue.

The function `ReturnAllData()` is called once all 10 integers have been entered and the return value from the function call is output.

Amend the main program to perform these actions.

Save your program.

Copy and paste the program code into part **2(e)(i)** in the evidence document.

[5]

(ii) Test your program with the following inputs in the order given:

10 9 –1 8 7 6 5 4 3 2 1

Take a screenshot of the output.

Save your program.

Copy and paste the screenshot into part **2(e)(ii)** in the evidence document.

[2]

(f) The function `Dequeue()` accesses `TheQueue`. The function returns –1 if the queue is empty. If the queue is **not** empty, the function returns the next item in the queue and updates the relevant pointer(s).

The data is **not** replaced or deleted from the queue.

Write program code for `Dequeue()`.

Save your program.

Copy and paste the program code into part **2(f)** in the evidence document.

[4]

(g) **(i)** The main program calls `Dequeue()` twice, and each time it either outputs ‘Queue empty’ if there is no data in the queue or outputs the return value.

The main program then calls `ReturnAllData()` a second time.

Amend the main program.

Save your program.

Copy and paste the program code into part **2(g)(i)** in the evidence document.

[3]

(ii) Test your program with the following inputs in the order given:

10 9 8 7 6 5 4 3 2 1

Take a screenshot of the output.

Save your program.

Copy and paste the screenshot into part **2(g)(ii)** in the evidence document.

[1]

3 A linked list stores positive integer data in a 2D array. The first dimension of the array stores the integer data. The second dimension of the array stores the pointer to the next node in the linked list.

A linked list node with no data is initialised with the integer `-1`. These nodes are linked together as an empty list. A pointer of `-1` identifies that node as the last node.

The linked list can store 20 nodes.

The global 2D array `LinkedList` stores the linked list.

`LinkedList` is initialised as an empty list. The data in each node is initialised to `-1`. Each node's pointer stores the index of the next node. The last node stores the pointer value `-1`, which indicates it is the last node.

The global variable `FirstEmpty` stores the index of the first element in the empty list. This is the first node in the empty linked list when it is initialised, which is index `0`.

The global variable `FirstNode` stores the index of the first element in the linked list. There is no data in the linked list when it is initialised, so `FirstNode` is initialised to `-1`.

This diagram shows the content of the initialised array.

```
FirstEmpty = 0

FirstNode = -1
```

Index	**Data**	**Pointer**
0	-1	1
1	-1	2
2	-1	3
3	-1	4
4	-1	5
19	-1	-1

(a) Write program code for the main program to declare and initialise `LinkedList`, `FirstNode` and `FirstEmpty`.

Save your program as **Question3_N24**.

Copy and paste the program code into part **3(a)** in the evidence document.

[2]

(b) The procedure `InsertData()` takes **five** positive integers as input from the user and inserts these into the linked list.

Each data item is inserted at the front of the linked list.

The table shows the steps to follow depending on the state of the linked list:

Linked list state	**Steps**
not full	insert the data in the index pointed to by `FirstEmpty` change the pointer to the index pointed to by `FirstNode` change the values of `FirstNode` and `FirstEmpty`
full	end the procedure

Any node that is at the end of the linked list has a pointer of `−1`.

Write program code for `InsertData()`.

Save your program.

Copy and paste the program code into part **3(b)** in the evidence document.

[6]

(c) The procedure `OutputLinkedList()` outputs the data in the linked list in order by following the pointers from `FirstNode`.

(i) Write program code for `OutputLinkedList()`.

Save your program.

Copy and paste the program code into part **3(c)(i)** in the evidence document.

[2]

(ii) Amend the main program to call `InsertData()` and then `OutputLinkedList()`.

Save your program.

Copy and paste the program code into part **3(c)(ii)** in the evidence document.

[1]

(iii) Test your program with the test data:

5 1 2 3 8

Take a screenshot of the output.

Save your program.

Copy and paste the screenshot into part **3(c)(iii)** in the evidence document.

[1]

(d) The procedure `RemoveData()` removes a node from the linked list.

The procedure takes the data item to be removed from the linked list as a parameter.

The procedure checks each node in the linked list, starting with the node `FirstNode`, until it finds the node to be removed. This node is added to the empty list, and pointers are changed as appropriate. The procedure only removes the first occurrence of the parameter.

Assume that the data item being removed is in the linked list.

(i) Write program code for `RemoveData()`.

Save your program.

Copy and paste the program code into part **3(d)(i)** in the evidence document.

[5]

(ii) Amend the main program to:

- call `RemoveData()` with the parameter 5
- output the word `"After"`
- call `OutputLinkedList()`.

Save your program.

Copy and paste the program code into part **3(d)(ii)** in the evidence document.

[1]

(iii) Test your program with both sets of given test data:

Test data set 1: 5 6 8 9 5

Test data set 2: 10 7 8 5 6

Take a screenshot of each output.

Save your program.

Copy and paste the screenshot(s) into part **3(d)(iii)** in the evidence document.

[1]