Cambridge Assessment International Education

Cambridge International AS & A Level

COMPUTER SCIENCE **9618/42**

Paper 4 Practical **May/June 2024**

2 hours 30 minutes

You will need: Candidate source files (listed on page 2)
evidence.doc

INSTRUCTIONS

- Carry out every instruction in each task.
- Save your work using the file names given in the task as and when instructed.
- You must **not** have access to either the internet or any email system during this examination.
- You must save your work in the evidence document as stated in the tasks. If work is **not** saved in the evidence document, you will **not** receive marks for that task.
- You must use a high-level programming language from this list:
  - Java (console mode)
  - Python (console mode)
  - Visual Basic (console mode)
- A mark of **zero** will be awarded if a programming language other than those listed here is used.

INFORMATION

- The total mark for this paper is 75.
- The number of marks for each question or part question is shown in brackets [ ].

This document has **16** pages. Any blank pages are indicated.

Open the evidence 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.

Three source files are used to answer **Question 1**. The files are called **`Easy.txt`**, **`Medium.txt`** and **`Hard.txt`**

1 A program outputs a main word. The program asks the user to enter the different words of 3 or more letters that can be made from the letters in the main word. These are called the answers.

There are 3 files: `Easy.txt`, `Medium.txt` and `Hard.txt`. Each file has the main word on the first line. For example, the main word in `Easy.txt` is house.

The answers are stored in the file. Each answer is on a new line after the main word. For example, `Easy.txt` has 14 answers that can be made from the letters in house.

The words read from the text file are stored in the global array `WordArray`. The number of words that can be made from the letters in the main word is stored in the global variable `NumberWords`.

(a) The procedure `ReadWords()`:

- takes a file name as a parameter
- opens the file and reads in the data
- stores the main word in the first element in `WordArray`
- stores each answer in a new element in `WordArray`
- counts and stores the number of answers.

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

Save your program as **Question1_J24**. Copy and paste the program code into part **1(a)** in the evidence document.

[6]

(b) The main program asks the user to enter `"easy"`, `"medium"` or `"hard"` and calls `ReadWords()`with the filename that matches the user's input. For example, if the user enters `"easy"`, the parameter is `"Easy.txt"`.

Write program code for the main program.

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

[4]

(c) (i) The procedure `Play()`:

- outputs the main word from the array and the number of answers
- allows the user to enter words until they enter the word 'no' to indicate they want to stop
- outputs whether each word the user enters is an answer or **not** an answer
- counts the number of answers the user gets correct
- replaces each answer that the user gets correct with a null value in the array.

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

Save your program.

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

[6]

(ii) Amend the procedure `Play()`so that when the user enters the command to stop, the procedure:

- outputs the percentage of answers the user entered from the array
- outputs all the answers that the user did **not** enter.

Write program code to amend `Play()`.

Save your program.

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

[3]

(d) (i) The procedure `ReadWords()`calls `Play()`after the data in the file has been read.

Write program code to amend `ReadWords()`.

Save your program.

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

[1]

(ii) Test your program by inputting these words in the order shown:

easy

she

out

no

Take a screenshot of the output(s).

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

[1]

(iii) Test your program by inputting these words in the order shown:

hard

fine

fined

idea

no

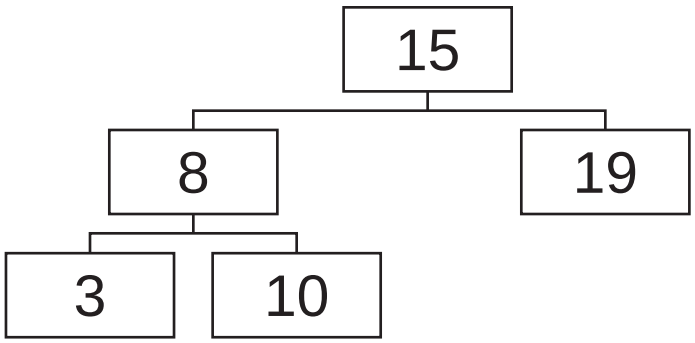
Take a screenshot of the output(s).

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

[1]

BLANK PAGE

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 sorts an array of integers and searches the array for a particular value.

(a) The array of integers, `NumberArray`, stores the following data in the order given:

100 85 644 22 15 8 1

The array is declared and initialised local to the main program.

Write program code to declare and initialise the array.

Save your program as **Question3_J24**.
Copy and paste the program code into part **3(a)** in the evidence document.

[1]

(b) (i) The following recursive pseudocode function sorts the array into ascending order using an insertion sort and returns the sorted array.

```
DECLARE LastItem : INTEGER
DECLARE CheckItem : INTEGER
DECLARE LoopAgain : BOOLEAN
FUNCTION RecursiveInsertion(IntegerArray : ARRAY[] OF INTEGER,
             NumberElements : INTEGER) RETURNS ARRAY[] OF INTEGER
   IF NumberElements <= 1 THEN
      RETURN IntegerArray
   ELSE
      CALL RecursiveInsertion(IntegerArray, NumberElements - 1)
      LastItem ← IntegerArray[NumberElements - 1]
      CheckItem ← NumberElements - 2
   ENDIF
   LoopAgain ← TRUE
   IF CheckItem < 0 THEN
      LoopAgain ← FALSE
   ELSE
      IF IntegerArray[CheckItem] < LastItem THEN
         LoopAgain ← FALSE
      ENDIF
   ENDIF
   WHILE LoopAgain
      IntegerArray[CheckItem + 1] ← IntegerArray[CheckItem]
      CheckItem ← CheckItem - 1
      IF CheckItem < 0 THEN
         LoopAgain ← FALSE
      ELSE
         IF IntegerArray[CheckItem] < LastItem THEN
            LoopAgain ← FALSE
         ENDIF
      ENDIF
   ENDWHILE
   IntegerArray[CheckItem + 1] ← LastItem
   RETURN IntegerArray
ENDFUNCTION
```

Write the program code for the pseudocode function `RecursiveInsertion()`.

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

[4]

(ii) Amend the main program to:

- call `RecursiveInsertion()` with the initialised array `NumberArray` and the number of elements as parameters
- output the word 'Recursive'
- output the content of the returned array.

Save your program.

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

[2]

(iii) Test your program.

Take a screenshot of the output(s).

Save your program.

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

[1]

(c) The function `RecursiveInsertion()` can be changed to use iteration instead of recursion.

(i) Write program code for the function `IterativeInsertion()` to perform the same processes as `RecursiveInsertion()` but using iteration instead of recursion.

Save your program.

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

[4]

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

- call `IterativeInsertion()` with the original initialised array `NumberArray`
- output the word 'iterative'
- output the content of the returned array.

Save your program.

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

[1]

(iii) Test your program.

Take a screenshot of the output(s).

Save your program.

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

[1]

(d) The recursive function `BinarySearch()` takes the parameters:

- `IntegerArray` – an array of integers
- `First` – the index of the first array element
- `Last` – the index of the last array element
- `ToFind` – an integer to search for in the array.

The function uses recursion to perform a binary search for `ToFind` in `IntegerArray`. The function returns the index where `ToFind` is stored or returns −1 if `ToFind` is **not** in the array.

(i) Write program code for the recursive function `BinarySearch()`.

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 `BinarySearch()` with the sorted array and the integer 644 as the search value
- output 'Not found' if 644 is **not** found in the array
- output the index if 644 is found in the array.

Save your program.

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

[2]

(iii) Test your program.

Take a screenshot of the output(s).

Save your program.

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

[1]

BLANK PAGE

BLANK PAGE

BLANK PAGE