Cambridge International AS & A Level

Cambridge International Examinations
Cambridge International Advanced Subsidiary and Advanced Level

CANDIDATE NAME

CENTRE NUMBER

CANDIDATE NUMBER

5010748614

COMPUTER SCIENCE **9608/41**

Paper 4 Further Problem-solving and Programming Skills **May/June 2017**

2 hours

Candidates answer on the Question Paper.

No Additional Materials are required.

No calculators allowed.

READ THESE INSTRUCTIONS FIRST

Write your Centre number, candidate number and name in the spaces at the top of this page.
Write in dark blue or black pen.
You may use an HB pencil for any diagrams, graphs or rough working.
Do not use staples, paper clips, glue or correction fluid.
DO **NOT** WRITE IN ANY BARCODES.

Answer **all** questions.
No marks will be awarded for using brand names of software packages or hardware.

At the end of the examination, fasten all your work securely together.
The number of marks is given in brackets [ ] at the end of each question or part question.

The maximum number of marks is 75.

This document consists of **15** printed pages and **1** blank page.

CAMBRIDGE International Examinations

1 The following table shows part of the instruction set for a processor which has one general 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 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.
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>.
JMP	<address>	Jump to given address.
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	<address>	Bitwise AND operation of the contents of ACC with the contents of <address>.
XOR	<address>	Bitwise XOR operation of the contents of ACC with the contents of <address>.
OR	<address>	Bitwise OR operation of the contents of ACC with the contents of <address>.
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 writes a program that:

- reads a character from the keyboard (assume it will be a capital letter)
- outputs the alphabetical sequence of characters from 'A' to the character input. For example, if the character 'G' is input, the output is:

  ABCDEFG

The programmer has started to write the program in the table on the following page. The Comment column contains descriptions for the missing instructions, labels and data.

Complete the following program. Use op codes from the given instruction set.

Label	Op code	Operand	Comment
START:			// INPUT character
			// store in CHAR
			// Initialise ACC (ASCII value for 'A' is 65)
			// OUTPUT ACC
			// compare ACC with CHAR
			// if equal jump to end of FOR loop
			// increment ACC
			// jump to LOOP
ENDFOR:	END		
CHAR:			

[8]

(b) The programmer now starts to write a program that:

- tests whether an 8-bit two's complement integer stored at address NUMBER is positive or negative
- outputs 'P' for a positive integer and 'N' for a negative integer.

Complete the following program. Use op codes from the given instruction set.
Show the required value of MASK in binary.

Label	Op code	Operand	Comment
START:			
		MASK	// set to zero all bits except sign bit
			// compare with 0
			// if not equal jump to ELSE
THEN:			// load ACC with 'P' (ASCII value 80)
	JMP	ENDIF	
ELSE:			// load ACC with 'N' (ASCII value 78)
ENDIF:			
	END		
NUMBER:	B00000101		// integer to be tested
MASK:			// value of mask in binary

[7]

2 A hash table has these associated operations:

- create hash table
- insert record
- search hash table

A hash table is to be used to store customer records.

Each record consists of a unique customer ID, the record key, and other customer data.

(a) The following pseudocode declares a customer record structure.

```
TYPE CustomerRecord
   CustomerID : INTEGER
   Data : STRING
ENDTYPE
```

The hash table is to be implemented as a 1D array `Customer` with elements indexed 0 to 199. The procedure to create a hash table will declare and initialise the array by storing 200 records with the `CustomerID` field in each record set to 0.

Complete the **pseudocode**.

```
PROCEDURE CreateHashTable()
```

.....

.....

.....

.....

```
ENDPROCEDURE
```

[4]

(b) A hashing function `Hash` exists, which takes as a parameter the customer ID and returns an integer in the range 0 to 199 inclusive.

(i) The procedure, `InsertRecord`, takes as a parameter the customer record to be inserted into the hash table.

The procedure makes use of the function `Hash`. Collisions will be managed using open hashing. This means a collision is resolved by storing the record in the next available location. The procedure will generate an error message if the hash table is full.

Complete the **pseudocode** for the procedure.

```
PROCEDURE InsertRecord(BYVALUE NewCustomer : CustomerRecord)
   TableFull ← FALSE
   // generate hash value
   Index ← .....
   Pointer ← Index   // initialise Pointer variable to hash value

   // find a free table element
   WHILE .....
      Pointer ← .....
      // wrap back to beginning of table if necessary
      IF .....
         THEN
            .....
      ENDIF
      // check if back to original index
      IF .....
         THEN
            TableFull ← TRUE
      ENDIF
   ENDWHILE

   IF .....
      THEN
         .....
      ELSE
         .....
   ENDIF
ENDPROCEDURE
```

[9]

(ii) The function `SearchHashTable` will search for a record in the hash table. The function takes as a parameter the customer ID to be searched for. The function will return the position in the hash table where the record has been saved. If the hash table does not contain the record, the function will return the value –1.

You can assume that there is at least one empty record in the hash table.

Complete the **pseudocode** for the function.

```
FUNCTION SearchHashTable(BYVALUE SearchID : INTEGER) RETURNS INTEGER
   // generate hash value
   Index ← .....
   // check each record from index until found or not there
   WHILE (.....)
      AND (.....)
      .....
   // wrap if necessary
      IF .....
         THEN
            .....
      ENDIF
   ENDWHILE
   // has customer ID been found?
   IF .....
      THEN
         .....
      ELSE
         .....
   ENDIF
ENDFUNCTION
```

[9]

(iii) A record that is no longer required is deleted.

State the problem that might be caused by this deletion.

.....

.....[1]

3 `NameList` is a 1D array that stores a sorted list of names. A programmer declares the array in 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]

4 An ordered linked list Abstract Data Type (ADT) has these associated operations:

- create list
- add item to list
- output list to console

The ADT is to be implemented using object-oriented programming as a linked list of nodes.

Each node consists of data and a pointer.

(a) There are two classes, `LinkedList` and `Node`.

(i) State the term used to describe the relationship between these classes.

.....[1]

(ii) Draw the appropriate diagram to represent this relationship. Do not list the attributes and methods of the classes.

[2]

(b) The design for the `Node` class consists of:

- attributes
  - `Data : STRING`
  - `Pointer : INTEGER`
- methods
  - `CreateNode(Data, Pointer)`
  - `SetData(Data)`
  - `SetPointer(Pointer)`
  - `GetData() RETURNS STRING`
  - `GetPointer() RETURNS INTEGER`

The constructor method sets the attributes to the initial values that are passed as parameters.

Write **program code** for:

- the `Node` class declaration
- the constructor.

Programming language used .....

Program code

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

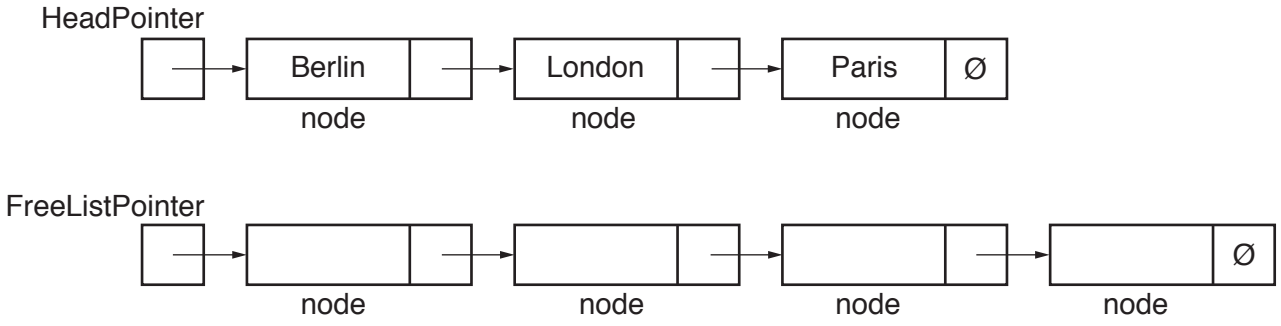
.....

.....[5]

(c) The identifier table for the `LinkedList` class is:

Identifier	Data type	Description
`HeadPointer`	`INTEGER`	Pointer to the first node in the ordered list.
`FreeListPointer`	`INTEGER`	Pointer to the first node in the free list.
`NodeArray`	`ARRAY[0 : 7] OF Node`	1D array stores the nodes that make the ordered linked list. The unused nodes are linked together into a free list.
`Constructor()`		Constructor instantiates an object of `LinkedList` class, initialises `HeadPointer` to be a null pointer and links all nodes to form the free list.
`FindInsertionPoint()`		Procedure that takes the new data item as the parameter `NewData` and returns two parameters: • `PreviousPointer`, whose value is: ◦ either pointer to node before the insertion point ◦ or the null pointer if the new node is to be inserted at the beginning of the list. • `NextPointer`, whose value is a pointer to node after the insertion point.
`AddToList(NewString)`		Procedure that takes as a parameter a unique string and links it into the correct position in the ordered list.
`OutputListToConsole()`		Procedure to output all the data from the list pointed to by `HeadPointer`.

The following diagram shows an example of a linked list object. This example list consists of three nodes, linked in alphabetical order of the data strings. The unused nodes are linked to form a free list.



The symbol Ø represents a null pointer.

(i) Explain the meaning of the term **null pointer**.

.....

.....[1]

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

....

....

....

....[2]

(iii) Write **program code** for the `LinkedList` class declaration **and** the constructor.

Programming language used ....

Program code

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....[7]

(iv) Write **program code** to instantiate a linked list object with the `contacts` identifier.

Programming language used .....

Program code

.....

.....[1]

(v) The `OutputListToConsole` method is to output all the data stored in the linked list. `HeadPointer` points to the first list node.

Write **program code** for this method.

Programming language used .....

Program code

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....[5]

Question 4 continues on page 14.

(vi) The structured English for the `AddToList(NewString)` method is as follows:

```
Make a copy of the value of free list pointer, name it NewNodePointer
Store new data item in free node pointed to by NewNodePointer
Adjust free list pointer to point to next free node
IF linked list is currently empty
   THEN
      Make this node the first node
      Set pointer of this node to null pointer
   ELSE
      Find insertion point using the FindInsertionPoint method
      // FindInsertionPoint provides
      // pointer to previous node and pointer to next node
      IF previous pointer is null pointer
         THEN
            Link this node to front of list
         ELSE
            Link this node between previous node and next node
```

The `FindInsertionPoint` method receives the new data item as the parameter `NewString`. It returns two parameters:

- `PreviousPointer`, whose value is:
  - either the pointer to the node before the insertion point
  - or the null pointer, if the new node is to be inserted at the beginning of the list.
- `NextPointer`, whose value is the pointer to the node after the insertion point.

Write **program code** for the `AddToList` method.

Programming language used ....

Program code

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....[6]

BLANK PAGE