Candidate Name: _____ CT Group: _____

Index no. _____

PIONEER JUNIOR COLLEGE
JC 2 PRELIMINARY EXAMINATION

COMPUTING H2 **9597/01**

Paper 1 **15 Sep 2015**

Time: 1400 - 1715 **3 hours 15 min**

Additional Materials: Removable storage device
Electronic version of `Rainfall_mth.csv` data file
Electronic version of `Rainfall_day.csv` data file
Electronic version of `Number.txt` data file
Electronic version of `Story.txt` data file
Electronic version of `Infix.txt` data file
Electronic version of `EVIDENCE.docx` file

READ THESE INSTRUCTIONS FIRST

Type in the `EVIDENCE.docx` document the following:

- Candidate details
- Programming language used

Answer **all** questions.

All tasks must be done in the computer laboratory. You are not allowed to bring in or take out any pieces of work or materials on paper or electronic media or in any other form.

All tasks and required evidence are numbered.

The number of marks is given in brackets [ ] at the end of each task.

Copy and paste required evidence of program listing and screen shots into the `EVIDENCE.docx` document.

This question paper consists of **10** printed pages (inclusive of this page).

1. `Rainfall_mth.csv` is a file which contains monthly total rainfall (in millimetres), for years 1984 to 2014. The format of the record is "***[Year] M [Month] , [rainfall in millimetres]***". Three sample records are:

- "`1984M01, 251.2`", which means total rainfall for January 1984 is 251.2;
- "`1999M07, 225.4`", which means total rainfall for July 1999 is 225.4;
- "`2014M11, 250.8`", which means total rainfall for November 2014 is 250.8.

The total annual rainfall for a particular year can be calculated by adding the monthly total rainfall for the twelve months.

Task 1.1
Write program code to find total annual rainfall from 1984 to 2014 and display in a table with a heading and borders as follows:

Year	Total Annual Rainfall (millimetres in 1 d.p.)
1984	2686.7
1985	1483.9
1986	2536.1
...	... ...
...	... ...
...	... ...
2014	1538.1

Evidence 1: The program code. **[7]**

Evidence 2: Screenshot to display total annual rainfall. **[1]**

`Rainfall_day.csv` is another file which contains number of rainy days in a month, for years 2009 to 2014. The format of the record is "***[Year] M [Month] , [number of days]***". Three sample record are:

- "`2009M03, 19`", which means there are 19 rainy days in March 2009;
- "`2011M05, 15`", which means there are 15 rainy days in May 2011;
- "`2014M09, 9`", which means there are 9 rainy days in September 2014.

The average rainfall for a rainy day for a particular year can be calculated by dividing the total annual rainfall by the total number of rainy days in a year.

Task 1.2
Amend your program code using the following specifications:

- Allow user to input a year from 2009 to 2014
- Output error message if input is outside these years – 2009 to 2014
- Calculate the average rainfall for a rainy day for that year
- Output the result

Evidence 3: Your amended program code. **[5]**

Evidence 4: Screenshots that show 2 test cases from running Task 1.2. **[2]**

2. Quicksort is a sorting algorithm that employs a divide-and-conquer strategy.

Here is a high-level description of Quicksort applied to an array A[0 : n – 1]:

1. Select an element from A[0 : n – 1] to be the pivot.
2. Rearrange the elements of A to partition A into a left subarray and a right subarray, such that no element in the left subarray is larger than the pivot and no element in the right subarray is smaller than the pivot.
3. Recursively sort the left and the right subarrays.

Task 2.1

Study the identifier table and incomplete quicksort algorithm. The missing parts of the algorithm are labelled A, B and C.

Variable	Data Type	Description
`ThisArray`	ARRAY OF INTEGER	Array containing the dataset
`First`	INTEGER	First index of array
`Last`	INTEGER	Last index of array
`Temp`	INTEGER	Temporary variable
`Low`	INTEGER	Index of array
`High`	INTEGER	Index of array
`Pivot`	INTEGER	Reference value in array

```
FUNCTION QuickSort(ThisArray, First, Last) RETURNS NULL
     DECLARE Temp:INTEGER, Low:INTEGER, High:INTEGER, Pivot:INTEGER
     Low ← First
     High ← Last
     .........A.........                    //Assign reference value

     WHILE Low <= High
          WHILE(ThisArray[Low] < Pivot)     //Scan left
               .........B.........
          ENDWHILE
          WHILE(ThisArray[High] > Pivot)    //Scan right
               High ← High – 1
          ENDWHILE
          IF Low <= High                    //Swapping
               Temp  ← ThisArray[Low]
               ThisArray[Low] ← ThisArray[High]
               ThisArray[High] ← Temp;
               Low ← Low + 1                //Shift right by 1 element
               High ← High - 1              //Shift left by 1 element
          ENDIF
     ENDWHILE

     IF First < High
          QuickSort(ThisArray, First, High)
     ENDIF
     If Low < Last
          .........C.........
     ENDIF
ENDFUNCTION
```

Evidence 5: What are the three missing lines of this pseudocode? **[3]**

Task 2.2

Write a program to implement the quicksort.

The program will:

- Call procedure `InitialiseList`.
- Use the function `QuickSort` to sort an array of integer `[435,646,344,54,23,98,789,212,847,201,733]`. Copy and paste this array from the file `Number.txt` into your program.
- Output the array before and after the quicksort algorithm is applied.

Evidence 6: Program code for Task 2.2. **[7]**

Evidence 7: Screenshot to show running of program code in Task 2.2. **[1]**

Task 2.3

Amend the program as follows:

The program must also output the number of function calls carried out.

Evidence 8: The amended program code. **[3]**

Task 2.4

By selecting different reference values (pivot) and input datasets, and making use of the number of function calls, evaluate the efficiency of the algorithm.

Evidence 9: Evaluation of efficiency of quicksort algorithm with accompanying screenshots (showing runs of function) for different reference values and input datasets. **[4]**

3. A program is to be written to find all the words in a piece of text and to print them in alphabetical order, together with the number of times each word occurs. The data structure used to hold this information will be a linked list, with each node holding a word, the number of occurrence of that word, and a pointer to the node containing the next word in alphabetical order.

The program will use nodes implemented as instances of the class **`ListNode`**. The class **`ListNode`** has the following properties:

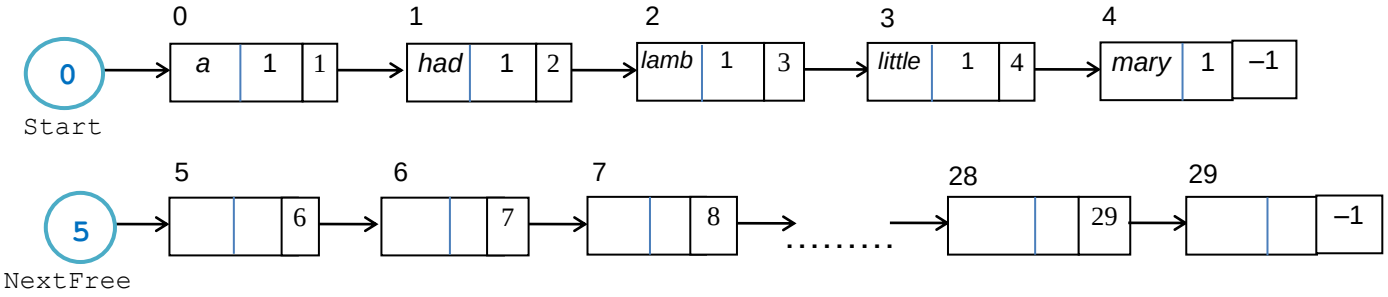
Class: `ListNode`		
Properties		
Identifier	**Data Type**	**Description**
`Word`	`STRING`	The node's value for a word from the text
`Count`	`INTEGER`	The node's value for number of occurrences of the word
`Pointer`	`INTEGER`	The pointer for the node

A linked list is implemented as an instance of the class `LinkedList`. The class `LinkedList` has the following properties and methods:

Class: `LinkedList`		
Properties		
Identifier	**Data Type**	**Description**
`Node`	`ARRAY[30] of ListNode`	The linked list data structure – data values (Word & Count) and pointers. Array index starts at 0. For testing purposes, the dataset has a maximum of 30 nodes.
`Start`	`INTEGER`	Index position of the node at the start of the linked list
`NextFree`	`INTEGER`	Index position of the next unused node
Methods		
`Initialise`	`PROCEDURE`	Sets all node data values to empty string (for Word) and 0 (for Count). Set pointers to indicate all nodes are unused and linked. Initialise values for `Start` and `NextFree`.
`Update`	`PROCEDURE`	Updates the linked list with a word read from the text
`Display`	`PROCEDURE`	Display the current state of array content and pointers in table form.
`IsEmpty`	`FUNCTION RETURNS BOOLEAN`	Test for empty linked list.
`IsFull`	`FUNCTION RETURNS BOOLEAN`	Test for no unused nodes.

The diagram shows the linked list with:

- the text "`mary had a little lamb`" added
- the unused nodes linked together.



Task 3.1

Write the program code for the classes `ListNode` and `LinkedList`, including the `Initialise`, `Display`, `IsEmpty` and `IsFull` method. The code should follow the specification given. Do not write the `Update` procedure yet.

Evidence 10: Your program code for the `ListNode` and `LinkedList` classes. **[12]**

Task 3.2

Write code to create a `LinkedList` object and run `Display` procedure to show the content of the object.

Evidence 11: Screenshot confirming all values after initialisation of the `LinkedList`. **[3]**

Task 3.3

Write code to implement for the `LinkedList` class the `Update` method that will update the linked list with a word read from the text.

Evidence 12: Program code for `Update` procedure. **[12]**

Task 3.4

Write code to use the `Update` procedure by reading in the text from the file `Story.txt`.

Evidence 13: Screenshot of state of array content and pointers by running `Display` procedure. **[4]**

Task 3.5

Write a method `Query` inside `LinkedList` class that:

- takes a word input by user,
- check if the word exists in the linked list,
- output appropriate message with the number of occurrences, if word exists.

Evidence 14: Program code for `Query` method. **[4]**

Evidence 15: Screenshot from running the `Query` method for a word that exists and another word that does not exist in the linked list. **[2]**

4. Implement a stack class using array using the following properties and methods:

Class: `Stack`		
Properties		
Identifier	**Data Type**	**Description**
`Data`	`ARRAY[x] of STRING`	x is the limit, which must be supplied when the object is called
`Limit`	`INTEGER`	The maximum number of elements the stack can hold
Methods		
Identifier	**Data Type**	**Description**
`IsEmpty`	`FUNCTION RETURNS BOOLEAN`	Indicates whether any elements are stored in stack or not
`IsFull`	`FUNCTION RETURNS BOOLEAN`	Indicates whether stack is full or not
`Push`	`PROCEDURE`	Inserts data onto stack
`Pop`	`PROCEDURE`	Removes and returns the last inserted element from the stack
`Peek`	`PROCEDURE`	Returns the last inserted element without removing it
`Size`	`PROCEDURE`	Returns the number of elements stored in stack
`Display`	`PROCEDURE`	Displays the content of the stack with top of stack clearly indicated

Task 4.1

Write program code for the stack class with all the properties and methods above.

Evidence 16: Your program code. **[12]**

A stack can be used to evaluate an arithmetic expression. An arithmetic expression can first be converted from infix notation to postfix notation, then the postfix notation can be evaluated to get the value of the infix notation.

For example, the infix notation **`5 * (6 + 2) - 12 / 4`** can first be converted to postfix notation **`5 6 2 + * 12 4 / -`**, and then evaluated to **`37`** using a stack.

The following is an algorithm for converting infix notation to postfix notation:

1. Create an empty stack called `opStack` for keeping operators.
2. Scan the token list from left to right.
    - If the token is an operand, append it to the end of the output list.
    - If the token is a left parenthesis, push it on the `opStack`.
    - If the token is an operator, *, /, +, or -, push it on the `opStack`. However, first remove any operators already on the `opStack` that have higher or equal precedence and append them to the output list.
    - If the token is a right parenthesis, pop the `opStack` until the corresponding left parenthesis is removed. Append each operator to the end of the output list.

When the input expression has been completely processed, check the `opStack`. Any operators still on the stack can be removed and appended to the end of the output list.

Task 4.2

Write program code for the algorithm to convert infix notation to postfix notation using the following specification:

```
FUNCTION infixToPostfix(infixexpression:STRING):STRING
```

Evidence 17: Your program code. **[7]**

The following algorithm can be used to evaluate postfix notation:

1. Create an empty stack called `operandStack`.
2. Scan the token list from left to right.
   - If the token is an operand, convert it from a string to an integer and push the value onto the `operandStack`.
   - If the token is an operator, *, /, +, or -, it will need two operands. Pop the `operandStack` twice. The first pop is the second operand and the second pop is the first operand. Perform the arithmetic operation. Push the result back on the `operandStack`.

When the input expression has been completely processed, the result is on the stack. Pop the `operandStack` and return the value.

Task 4.3

Write program code for the algorithm to evaluate postfix notation using the following specification:

```
FUNCTION postfixEval(postfixexpression:STRING):FLOAT
```

Evidence 18: Your program code. **[7]**

Task 4.4

Write program code to read the infix expressions from the file `Infix.txt` and output its postfix expressions and its evaluation.

Evidence 19: Your program code. **[2]**

Evidence 20: Screenshot of running your program code in Task 4.4. **[2]**

~~~ END OF PAPER ~~~