NATIONAL JUNIOR COLLEGE
Mathematics Department

General Certificate of Education Advanced Level
Higher 2

COMPUTING 9569/2

Paper 2 (Lab-based)

15 Aug 2023
3 hours

Additional Materials:	Electronic version of `ARRIVALS.TXT` data file
	Electronic version of `WORDS.TXT` data file
	Electronic version of `TASK4.db`
	Electronic version of `ADDITIONS.CSV`
	Electronic version of `style.css`
	Insert Quick Reference Guide

READ THESE INSTRUCTIONS FIRST

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

Approved calculators are allowed.

Save each task as it is completed.

The use of built-in functions, where appropriate, is allowed for this paper unless stated otherwise.

Note that up to 5 marks out of 100 will be awarded for the use of common coding standards for programming style.

The number of marks is given in the brackets [ ] at the end of each question or part question.
The total number of marks for this paper is 100.

This document consists of **12** printed pages and 0 blank pages.

Instructions to candidates:

Copy the folder from the thumb drive to the PC's desktop and rename the folder on the desktop to `<your name>.`(For example, TanKengHan). All the resource files are found in the folder and you should work on the folder in the desktop.
Your program code and output for each of Task 1 to 3 should be saved in a single `.ipynb` file. For example, your program code and output for Task 1 should be saved as `Task_1_<your name>.ipynb.`
You should have a total of **three** `.ipynb` files to submit at the end of the paper.
At the end of the exam, copy the working folder on your desktop to the thumb drive.

1 Name your Jupyter Notebook as `Task1_<your name>.ipynb.`

This task uses an Object Oriented Programming (OOP) and layered approach to implement linked list, stack and queue data structures. Each data structure will be implemented by using the abstractions (attributes and operations) provided by a previously implemented data structure.

For each sub-task, add a comment statement, at the beginning of the code using the hash symbol "#", to indicate the sub-task the program code belongs to, for example:

```
In [1]:   #Task 1.1
          Program code
          Output:
```

Task 1.1

A linked list is a dynamic data structure where the storage for new data is allocated on demand and the data items are accessed by using pointers to traverse the linked list. Implement a linked list class and a node class, using Python code.

The node class will be used to encapsulate and store the data item and will have the following attributes:

- `data`, stores the data item.
- `next,` points to the next node in the linked list.

The linked list class will have the following attributes and operations:

- `start` , this attribute points to the first node in the linked list. A `None` value indicates an empty linked list
- `is_empty()`,returns `True` if the linked list is empty and `False` otherwise.

- `insert_front()`, inserts a data item at the front of the linked list, so that it is accessible by the `start` attribute.
- `remove_front()`, removes and returns the first item from the linked list.
- `__repr__()`, returns a string representation of the linked list as follows:
    - `[ <item>, <item>,… ]`, where `<item>` is the data item in the linked list starting from the first item pointed by the `start` attribute. [8]

Task 1.2

A stack is a data structure with the following attribute and operations:

- `top`, pointer that points to a linked list data structure.
- `push(item: object)` , inserts an item into the top of the stack.
- `pop(): object` , returns and removes the item at the top of the stack. If the stack is empty `None` is returned.
- `peek(): object`, returns the item at the top of the stack without removing it. If the stack is empty `None` is returned.
- `constructor()`, creates and initialises attributes used by the stack object.
- `is_empty()`, returns `True` if stack is empty, else returns `False`.

Implement the stack class by using the linked list implemented in Task 1.1. [4]

Task 1.3

Write test cases to test the stack implementation in Task 1.2. [1]

Task 1.4

A queue is a data structure with the following operations:

- `enqueue(item: object)` , inserts an item into the end of the queue.
- `dequeue(): object` , returns and removes the item at the front of the queue. If the queue is empty `None` is returned.
- `constructor(),` creates the queue object.

Implement the queue class by using **ONLY** the stack data structure implemented in Task 1.2 and you are allowed to use only the operations and attributes provided by the stack data structure. You may need to include additional attributes and operations in the queue. [6]

Task 1.5

Write test cases to test the queue implementation in Task 1.4. [1]

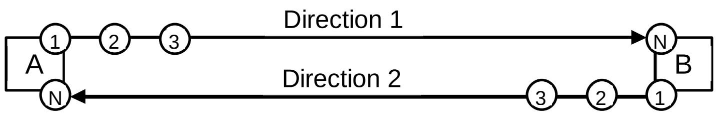
Task 1.6

A new data structure named `MaxQueue` is to be implemented by inheriting from the queue class implemented in Task 1.4. The `MaxQueue` has a new operation `peek_max()` which will return the data item that has the highest value among all the data items currently in the queue. You can assume that the data item's value can be compare using the default `<` and `>` operators. [10]

Save your Jupyter Notebook for Task 1.

2 Name your Jupyter Notebook as `Task2_<your name>.ipynb.`

The journeys taken by two buses were recorded in the file `ARRIVALS.TXT` and depicted in the diagram below:



The first bus started its journey from bus interchange A and ended its journey at bus interchange B. The second bus started its journey from bus interchange B and ended its journey at bus interchange A. The numbers, 1,2,3, ... N in the diagram represents the sequence of stops made by each bus in Direction 1 and 2.

The data in the file has the following fields, `StopSequence`, `Direction`, `BusStopCode`, `Distance`, `ArrivalTime` described below:

Each bus stop (including the bus interchanges) has a unique bus stop code, `BusStopCode`. The sequence of the bus stops visited by each bus is recorded as a `StopSequence` number for each `Direction`. The `ArrivalTime` is recorded in 24-hour format and the `Distance` is the cumulative distance travelled by the bus in km. You can assume that the two routes taken by the buses in Direction 1 and Direction 2 are on the same straight line and have the same distance. The outputs shown may not be the correct results.

For each sub-task, add a comment statement, at the beginning of the code using the hash symbol "#", to indicate the sub-task the program code belongs to, for example:

```
In [1]:  #Task 2.1
         Program code
        Output:
```

Task 2.1

Write Python code to determine the bus stop code of the two interchanges A and B where the two buses start and end their journeys. The bus interchanges have the same bus stop codes in both directions. Print the two bus stop codes. [3]

Task 2.2

Write Python code to determine the start time, end time and total time it takes for the two buses to travel in Direction 1 and Direction 2 respectively. The output should look like this:

```
Direction Start End  Total
    1     0500  0627 1H,27m
    2     0600  0730 1H,30m
```

[5]

Task 2.3

For both directions, write Python code to determine all the adjacent bus stops that required more than two minutes to travel between them. Print the bus stop codes of the adjacent bus stops and the time taken in minutes.The output should look like this:

```
Direction 1
76069 to 96289: 4m
80219 to 80169: 3m
02119 to 03019: 3m
03217 to 05649: 3m
Direction 2
14089 to 14069: 3m
80161 to 80211: 4m
96281 to 76061: 4m
```

[4]

Task 2.4

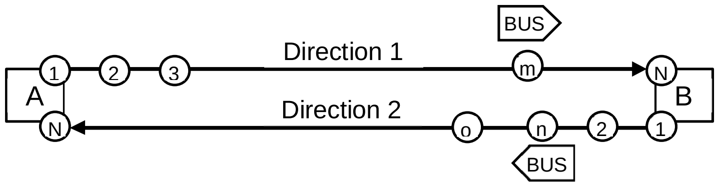
For both directions, write Python code to determine the longest distance that the two buses need to travel between two adjacent bus stops.

```
Direction 1
longest distance between two adjacent bus stops is 1.40 km
Direction 2
longest distance between two adjacent bus stops is 1.60 km
```

[3]

Task 2.5

Write Python code to determine the time when the two buses are at bus stops that are closest to each other, as they travel in Direction 1 and 2 respectively, as depicted by the diagram below:



Your computation will take into account of only the time and distance recorded by the buses at the bus stops. You cannot assume that the two buses travel with a uniform speed throughtout the journey. Print out the time and the bus stop codes that the buses were at. The output should look like this:

```
The time in which the two buses is closest to each other is at
0617 at bus stops <m> and <n>.
```
[6]

Save your Jupyter Notebook for Task 2.

3 Name your Jupyter Notebook as `Task3_<your name>.ipynb.`

The file, `WORDS.TXT`, contains more than eighty thousand unique words. This task requires you to sort the words in ascending lexicographical order such that an efficient search operation can be perform to look up a word to locate its line number in the file `WORDS.TXT.` Line numbering should start at 1.

For each sub-task, add a comment statement, at the beginning of the code using the hash symbol "#", to indicate the sub-task the program code belongs to, for example:

```
In [1]:   #Task 1.1
          Program code
         Output:
```

Task 3.1

Write Python code to implement a **recursive** function,

`FUNCTION qsort(words: LIST of TUPLE) RETURNS LIST of TUPLE` ,

that uses the quick sort algorithm to sort a Python list of tuples in ascending lexicographical order of the first element of the tuple. For example,

`qsort([("Orange",23), ("Apple",121), ("Mango",2),("Durian",4)])`

should return

`[('Apple',121),('Durian',4),('Mango',2),('Orange',23)]` [5]

Task 3.2

Implementing a recursive solution in Python is limited by the maximum call stack size of the Python interpreter, which by default is set to 3000. In the worst case, the call stack limit will be reached when you attempt to use the `qsort()` implemented in Task 3.1 t o sort a list containing 3000 tuples or more.

Write three test cases to validate the correctness of the `qsort()` function, including one invalid test case that will cause a run time error to occur as a result of exceeding the maximum call stack size. In Python, the run time error looks like this: [3]

`RecursionError: maximum recursion depth exceeded …`

You may use built-in Python functions in your test cases.

Task 3.3

In this task, you are required to read all the words and their corresponding line numbers in a file, `WORDS.TXT`, into a Python list and then sort them in ascending lexicographical order of the words. The file `WORDS.TXT` contains more than 3000 words. In the worst case, the call stack limit will be reached when you attempt to use the `qsort()` implemented in Task 3.1, to sort the entire file. In order to sort all the words in the file, you will need to perform the sorting in batches as follows:

- Read the words to be sorted from the file `WORDS.TXT` and append the word and its line number in the file as a tuple into a list, `words.`For example if the word `'abandoning'` is in line 2971 of the file, it should be appended into the list as `('abandoning',2971).`
- While the list `words` is not empty, extract no more than 3000 items from the list `words` and use `qsort()`implemented in Task 3.1 to sort the items into a list `sorted_tmp`, append the list `sorted_tmp` into a list `sorted_partial.` Repeat Step 2. `sorted_partial` should contain a list of lists.
- Merge all the lists in the list `sorted_partial` into a single sorted list, `sorted_all`. You have to provide your own algorithm to perform the merge.

Write Python code to perform the steps described above. Name the sorted list as `sorted_all` and print its contents. [10]

Task 3.4

The list `sorted_all` created in Task 3.2 should contain elements that are sorted in ascending lexicographical order of the first element of the tuple as follows:

`[('abandoning',2971),('abas',9338),…('zoos',6146)]`

where each element contains a tuple representing a word and its corresponding line number in the file `WORDS.TXT.` Implement a binary search algorithm as a Python function `bin_search(sorted_all:LIST, word:STRING)`, to search for a word in the file `WORDS.TXT` by passing in the list `sorted_all` and the word to be searched as arguments. If the word is found, return the word and its line number otherwise return `None.` [3]

Task 3.5

Implement a Python function :

```
FUNCTION prefix_search(sorted_all:LIST, prefix:STRING)
RETURNS LIST,
```

to perform a binary search for all words in the file `WORDS.TXT,` with a given prefix. For example,

`prefix_search(sorted_all, "arson")` should return the list,

[('**arson**ist',42704), ('**arson**',36888), ('**arson**ists',33575)] [4]

Task 3.6

Implement a Python function :

```
FUNCTION suffix_search(sorted_all:LIST, suffix:STRING)
RETURNS LIST,
```

to perform a search for all words in the file `WORDS.TXT,` with a given suffix. For example,

`suffix_search(sorted_all, "rusty")` should return the list,

[('c**rusty**',19915),('**rusty**',73656),('t**rusty**',52448)] [3]

Save your Jupyter Notebook for Task 3.

4 The database file TASK4.db contains a partial data set for an event ticket booking system.

An event has a unique ID number, event name, artist and event type. Each event has one or more performances. Each performance will contain the date, time and venue. The two tables in the database are as follows:

- `Event( id, name, artist, type )`
- `TimeTable( id, event_id*, date, time, venue )`

```
Underline : Primary key
* : Foreign key
```

The tables have been pre-loaded with data.

Task 4.1

Write a Python program and the necessary files to to create a web application. The landing or home page of the web application should display all the event records in the database as follows (output 1):

Event Name	Artist	Type of Event	
GARNiDELiA stellacage 2023	GARNiDELiA	concert	See performance dates
JACKY CHEUNG 60+ CONCERT TOUR	Jacky Cheung	concert	See performance dates
Legend: Heaven and Earth	Sulwyn Lok	concert	See performance dates
Taylor Swift | The Eras Tour	Taylor Swift	concert	See performance dates

A stylesheet file, `style.css` has been created for you, You should use the style and formatting instruction in this file in all your html pages. This landing page should provide a user interface for the user to select an event to view all the performance dates for the the selected event. After a particular event has been selected by the user, the web application should then display the performance details as follows (output 2):

Taylor Swift | The Eras Tour

Date	Time	Venue
02/03/24	2000	National Stadium
03/03/24	2000	National Stadium
04/03/24	2000	National Stadium

Save your program code as `Task4_1_<your_name>.py` and any additional files/subfolders as needed in a folder named `Task4.`

Run the web application and save the two outputs of the program as `Task4_1a.html` and `Task4_1b.html` respectively. [10]

Task 4.2

The file `ADDITIONS.CSV` contains details of additional performances by the artist after the database has been populated with data. Write Python code to read the contents from the file and update the database with the additional performances.

Save your program code as `Task4_2_<your_name>.py` [6]

END OF PAPER