NATIONAL JUNIOR COLLEGE
Science Department

General Certificate of Education Advanced Level
Higher 2

COMPUTING 9597/01

Paper 1

15 August 2017

3 hour 15 minutes

Additional Materials:	Pre-printed A4 Paper Removable storage device Electronic version of `2A_SCORES.TXT` data file Electronic version of `2B_SCORES.TXT` data file Electronic version of `2C_SCORES.TXT` data file Electronic version of `MENU.TXT` data file Electronic version of `EVIDENCE.DOCX` document

READ THESE INSTRUCTIONS FIRST

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

- Candidate details
- Programming language used

They are 4 questions worth a total of 100 marks. 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 the brackets [ ] at the end of each task.

Copy and paste required evidence of program listing and screenshots into the `EVIDENCE.DOCX` document.

This document consists of **10** printed pages and **0** blank pages.

1 The head of the mathematics department would like to combine and rank the examination results from 3 classes so that a summary report can be generated. The mathematics teachers from these classes were instructed to use the following format (on each line of a text file) when collating the results:

<student index number>,<student examination score>

After the results from each class had been collated, the following was discovered. Each <student index number> was stored as a denary value. However, the mathematics teachers stored <student examination score> values using different base number systems.

The following were the base number systems utilised:

- Class 2A: binary
- Class 2B: octal
- Class 2C: hexadecimal

It should be noted that all index numbers and examination scores correspond to non-negative integer values.

Task 1.1

Write the program code for the function `base_n_to_denary(value_str, n)`, which converts the given base `n`, non-negative integer value, `value_str`, into its corresponding denary value, and then returns it.

More specifically, this function should take the following arguments.

- `value_str`: a string corresponding to a base `n` value.
- `n`: a positive integer between 2 and 16 corresponding to the base number system used to represent `value_str`.

The function should then return an integer (i.e., a denary value), whose value is equal to `value_str`. You may not use any inbuilt functionality to perform this conversion.

Evidence 1

- The program code for the function `base_n_to_denary(value_str, n)`. [3]

As mentioned above, the head of the mathematics department would like to generate a summary report of the examination.

The format of this summary report is as follows:

```
Mathematics results for classes 2A, 2B and 2C:

The highest examination score: 100.0
The average examination score: 50.0
The lowest examination score: 0.0

The top 3 students are:

  Class     Index      Mark
    2A        1        100
    2C        10        98
    2B        3         96
```

Note that the values used in the example summary report above are not based on the actual values that are found in the given files.

Task 1.2

Write the program code to perform the following.

- Read and store the contents of the files:
    - `2A_SCORES.TXT`
    - `2B_SCORES.TXT`
    - `2C_SCORES.TXT`
- When storing this data, utilise the function `base_n_to_denary(value_str, n)` to convert the scores in each file to their corresponding denary values.

Evidence 2

- The program code to perform the above task. [3]

Task 1.3

Write the program code for the function `sort_student_scores(...)`, which uses the class, index and score data (i.e., the data retrieved from the 3 files listed in **Task 1.2**), and returns the same information, but sorted in **descending order** based on score.

When considering the parameters and return value for `sort_student_scores(...)`, your decision(s) should be based upon the usage of this information to produce the summary report described earlier in this question.

Note that you are to utilise the **quicksort** algorithm to perform the required sorting. Also note that you may not utilise any inbuilt functionality to search or sort when writing this function.

Evidence 3

- The program code for the function `sort_student_scores(...)`. [5]

Task 1.4

Utilise the data from **Task 1.2**, and the function implemented in **Task 1.3** – i.e., `sort_student_scores(...)`, to produce the summary report described earlier in this question.

Note that you may not use any inbuilt functionality to find the:

- Average score
- Maximum score
- Minimum score

Evidence 4

- The program code to print the summary report. [3]

Evidence 5

- A screenshot of the summary report output. [1]

2 A Japanese restaurant would like to trial a new system for taking orders. More specifically, the owners would like to trial a prototype text-based system. You have been tasked to help with its implementation.

The menu is currently stored in the text file `MENU.TXT`, with each menu item stored on one line in the following format:

<menu item index number> <menu item name> <menu item price>

Note that the 3 pieces of information are separated by spaces. The <menu item index number> always contains 3 digits (including preceding zeros), and the <menu item price> always begins with the "`$`" sign.

Task 2.1

Write the code to display and navigate through a text-based menu with the following options:

```
1. Read menu data
2. Take order
3. Quit
```

You are to implement the following.

- Full functionality for Option 1 and Option 3.
- A stub for Option 2.

When implementing Option 1, take note of the following.

- The menu items are to be read from `MENU.TXT`.
- For each menu item, the <menu item index number>, <menu item name> and <menu item price> should be stored. You must implement an augmented **linear search** (on <menu item index number>) to ensure that insertions leave the menu items **sorted in ascending order** (i.e., by finding the index of the element just greater than the search item).
- The menu item index numbers in `MENU.TXT` need not be listed in order, and there may be missing index numbers as certain old menu items may have been omitted from the file.

Also, the selection of Option 1 must precede the selection of Option 2, though Option 1 need only be selected once for each execution of the menu program. Your implementation must incorporate these requirements.

Further, do ensure that proper exception handling is implemented for user input.

Evidence 6

- The program code for the text-based menu as specified above. [8]

Task 2.2

Write the program code for Option 2.

When Option 2 is selected (legally), the following message should be displayed:

```
Please enter a menu item index (or −1 to complete the order):
```

The message is repeated for each <menu item index number> that is input, and will repeat until the value −1 is input.

For each <menu item index number> that is input, a search should be performed to ensure that it exists (based on the menu items read from `MENU.TXT`). You are to implement the **binary search** algorithm to perform this check.

Note that for this sub-option, you need not implement any exception handling.

When an index that does not exist in `MENU.TXT` is entered, the following message is to be output.

```
Invalid menu index; that index does not exist.
```

With each order, the user inputs one or more menu items (via <menu item index>). Once all the ordered menu items have been entered, the user then inputs -1 to conclude the order, at which point, the following is output.

- A list of all the ordered items – for each item ordered, this includes: (i) <menu item index number>, and (ii) the <menu item name>.
- The total price of all the items ordered.

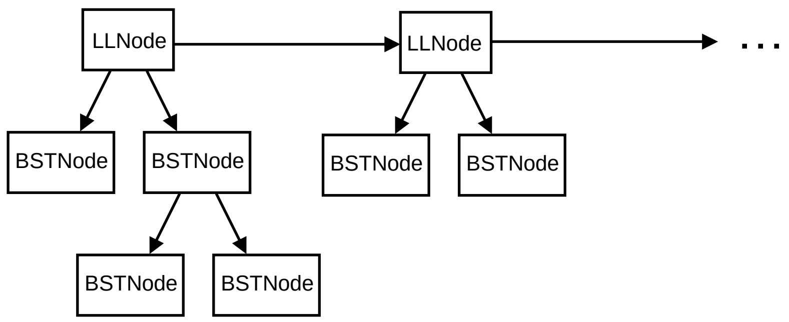
A sample of the above output would be as follows:

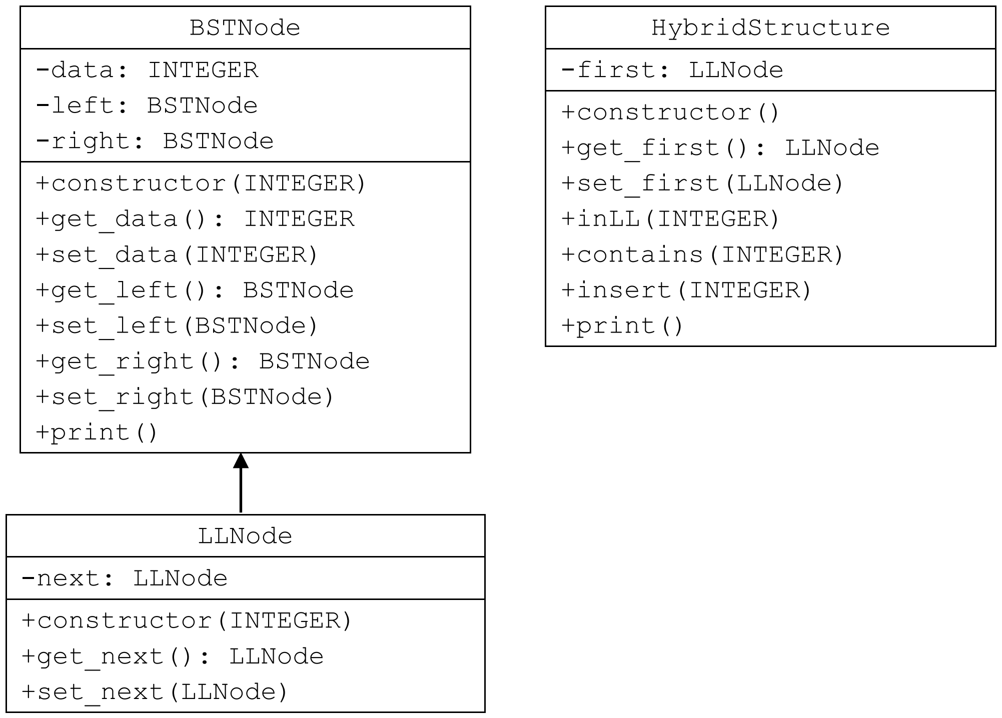
```
Index  Item
021    Ebi Curry
064    California Temaki
026    Salmon Fry Set
127    Fried Chicken Ramen (Soup)

Total price: $30.6
```

Evidence 7

- The program code for Option 2. [7]

3 Mickey is experimenting with object-oriented data structures, and would like to implement a hybrid Linked List, where each node in the Linked List is also the root of a Binary Search Tree. He intends to utilise this hybrid structure to store integers ranging from 1 to 99. His design is as follows.





Note that this Linked List is **singly-linked**, and also notice that **each node of the Linked List** is **also the root of a Binary Search Tree**.

Task 3.1

Write the object-oriented code for the `BSTNode` and `LLNode` classes described above.

Evidence 8

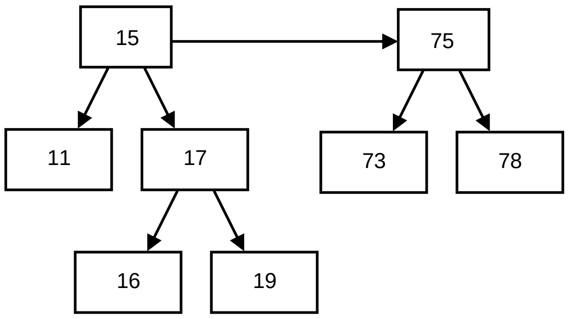
- The program code for the `BSTNode` and `LLNode` classes. [12]

Mickey's intention is to have the insert method from the `HybridStructure` class act as follows.

- When inserting integer *i*
- First search the **Linked List** to determine if there exists another integer *j* such that the tens digit in *i* and the tens digit in *j* are the same
- If the above is `True`, then insert integer *i* into the **Binary Search Tree** whose root Node holds the integer *j*, else, insert integer *i* into the **Linked List**

The following table describes the functionality of the non-standard `HybridStructure` methods.

Method	**Functionality**
`+inLL(INTEGER)`	Returns `True` if the given integer shares the same tens digit as another integer in the Linked List, else returns `False`
`+insert(INTEGER)`	Inserts a new integer based on the procedure described above
`+contains(INTEGER)`	Returns `True` if the given integer can be found in the structure, else returns `False`

`+print()`	Prints the contents of each Binary Search Tree **inorder**, leaving a blank line in between each Linked List Node (and the contents of its associated Binary Search Tree)
	
	For example, given the following `HybridStructure`:
	
	The output would be:
	```
	11
	15
	16
	17
	19

	73
	75
	78
	```

Task 3.2

Write the object-oriented code for the `HybridStructure` class described above.

Evidence 9

- The program code for the `HybridStructure` class. [25]

Task 3.3

Initialise an instance of `HybridStructure`, insert the integers `15 17 12 13 96 91 99 97` (in the order specified), and then call the method `print()` to check its contents.

Evidence 10

- The program code to for the above task. [2]

Evidence 11

- The screenshot of the output. [1]

4 Laura is always forgetting who she lends her books to and decides to implement a special pair of Hash Tables to help her keep track of her book loans.

Laura has exactly 97 books, and has no plans to buy any new books.

In her new system, a simple tuple (<Book Title>, <Person Who Borrowed the Book>) is created each time she lends someone a book.

In order for her system to work, she would like to implement two Hash Tables:

- The first Hash Table, `HT1`, utilises a Hash Function that is applied to <Book Title>.
- A checksum is applied to determine the Hash Value for each <Book Title>, where the ASCII value of each character in the title is multiplied by its position in the <Book Title> string (starting from left to right), and then summed.
- For example, given the <Book Title>: `"A Cat"`, the summed value would thus be:

  65×1 + 32×2 + 67×3 + 97×4 + 116×5.
- A **modulus operation** is then applied to determine the final Hash Value.
- The second Hash Table, `HT2`, utilises a similar Hash Function, except that it is instead applied to <Person Who Borrowed the Book>.

Task 4.1

Write the code for the function `get_hash(my_string)`, which takes in a string argument and returns its resultant Hash Value, which is defined by the checksum method described above.

Evidence 12

- The program code for the `get_hash(my_string)` function. [5]

In Laura’s data structure design, the two Hash Tables, `HT1` and `HT2`, do not themselves store the loan information tuples – i.e., (<Book Title>, <Person Who Borrowed the Book>). Instead, the loan information tuples are stored in an array, `Loans`.

The contents of `HT1` and `HT2` thus correspond to indices of `Loans`.

When a book is loaned, it is inserted into the next free index in the array `Loans` (let this index be *k*), then, the Hash Values for that particular loan are computed (based on <Book Title> for `HT1` and <Person Who Borrowed the Book> for `HT2`), and the value *k* stored in the relevant indices of `HT1` and `HT2`.

This is exemplified below.

- Assume that we begin with no loans stored.
- Suppose that we then have a new loan: `(“My Book”, “John Tan”)`
- Assume that:
    - o The Hash Value for `“My Book”` is `1`
    - o The Hash Value for `“John Tan”` is `3`
- Thus, to store this loan, we perform the following:
    - o The new loan is stored in the Loans array via: `Loans[0] = (“My Book”, “John Tan”)`
    - o Then, `0` is stored in `HT1[1]` (since the Hash Value of `“My Book”` is `1`)
    - o And, `0` is also stored in `HT2[3]` (since the Hash Value of `“John Tan”` is `3`)
- The resultant contents of `Loans`, `HT1` and `HT2` would then be as follows.

`Loans:`	`(“My Book”, “John Tan”)`	`EMPTY`	`EMPTY`	`EMPTY`

`HT1:`	`EMPTY`	`0`	`EMPTY`	`EMPTY`

`HT2:`	`EMPTY`	`EMPTY`	`EMPTY`	`0`

Laura has designed these Hash Tables to utilise the Closed Hashing method of Linear Probing to resolve any conflicts that occur.

Task 4.2

By selecting appropriate Array and Hash Table sizes to suit Laura’s needs, write the code :

- To initialise the array `Loans` and the two Hash Tables `HT1` and `HT2`.
- For the function `insert(book_title, person_borrowing)`, which will “insert” the given loan into `Loans`, `HT1` and `HT2`. Note that the function `get_hash(my_string)`, which was implemented for **Task 4.1**, should be used.

Evidence 13

- The program code to initialise `Loans`, `HT1` and `HT2`, and the code for the function `insert(book_title, person_borrowing)`. [10]

Task 4.3

Write the code for the functions:

- `find_loan_by_title(title)`, which returns the loan tuple corresponding to the `title` given, or else, will return an empty tuple.
- `find_loan_by_person(person)`, which returns the loan tuple corresponding to the `person` given, or else, will return an empty tuple.
- `remove_loan_by_title(title)`, which deletes the loan tuple corresponding to the `title` given and returns `True`, or else, returns `False` if it was not removed. This should also remove the corresponding entries in `HT1` and `HT2`.
- `remove_loan_by_person(person)`, which deletes the loan tuple corresponding to the `person` given and returns `True`, or else, returns `False` if it was not removed. This should also remove the corresponding entries in `HT1` and `HT2`.

Ensure that you practice good **modularity** and **code reuse** when implementing these functions.

Evidence 14

- The program code to for the above mentioned functions. [12]

Task 4.4

Write code to modify the `insert(book_title, person_borrowing)` function such that it prints the amount of additional probing that is required on any insertion – i.e., prints the number of iterative steps that linear probing was required to perform in order to find an insertion point.

Evidence 15

- The program code to that is required to make the above mentioned modification. [3]