Name:		Index Number:		Class:	

DUNMAN HIGH SCHOOL
Preliminary Examination
Year 6

COMPUTING **9597**

(Higher 2)
Paper 1

1 September 2014
3 hours 15 minutes

Additional Materials: Data files and `EVIDENCE.docx`

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

Copy and paste required evidence of program code and screen shots into `EVIDENCE.docx`.

At the end of the examination, print out and submit your `EVIDENCE.docx`.

<u>Data files</u>
Q1 – `RACE.txt`
Q2 – `PASTRY.txt`
Q3 – `ITEMS.dat, LOAN.dat`
Q4 – `IPV6_LONG.txt, IPV6_SHORT.txt`

1. Many large-scale races feature a chip time technology in which participants run with a computer chip containing identification information attached to their sports attire or shoes. Sensors at the start and finish lines pick up the signal from the chip as a runner passes through them. Chip time is the time it takes a runner to cross the start and finish lines.

The file `RACE.txt` contains recorded information of a 21-km half marathon race. Each entry has the following format:

```
<Runner ID><Start Time><End Time>
```

For example, the entry

```
A002318:02:0319:33:58
```

means that runner `A0023` crossed the start line at 18:02:03 and the finish line at `19:33:58`, giving a chip time of 1 hour 30 minutes 55 seconds.

Note: You may not use the built-in `sort()`, `min()` and `max()` functions.

Task 1.1
Write program code to find and output the fastest runner id and its corresponding chip time. Use the file `RACE.txt` to test your program.

Sample output:

```
Fastest runner id: A0123    Chip time: 1 h 09 m 23 s
```

Evidence 1:
Program code. [5]

- convert time string to seconds and compute chip time
- loop to find and update fastest (minimum) time
- get corresponding fastest runner id
- convert fastest time to hour, minute and second format
- exception handling to open and read input file, close file and error message

Note: Some candidates performed sorting to get the fastest result and runner. Note that sorting efficiency (linearithmic or quadratic) is not as efficient as linear time processing.

Evidence 2:
Screenshot of output. [1]

- correct runner id and chip time with specified output formatting

Task 1.2
Amend your program code to display the top 3 ranks and the runner ids and their corresponding chip times. If multiple runners record the same chip time, they will have the same rank. Use the file `RACE.txt` to test your program.

Sample output:

```
 1. A0123    Chip time: 1 h 09 m 23 s
```

```
2. A4385    Chip time: 1 h 09 m 25 s
2. A9846    Chip time: 1 h 09 m 25 s
3. A5951    Chip time: 1 h 10 m 09 s
```

Evidence 3:
Program code. [8]

- appropriate data structure (Python list/array or string) to store results for top 3 ranks
- appropriate sorting method to arrange results by ascending time [2]
- get top 3 rank timings from sorted results
- get top 3 rank runner ids
- correct condition for tie
- correct update of rank counter (account for tie)
- loop to output formatted results

Evidence 4:
Screenshot of output. [1]

- correct output of top 3 results with tie

2. The task is to manage a small confectionary's pastry stock and sales.

Information about the pastry is stored in the text file `PASTRY.txt` which you should copy and paste to your program code.

Task 2.1
Write program code to determine if a particular pastry is available or sold out.

Sample execution:
```
Enter pastry name: Spicy Floss
Available.
Enter pastry name: Coconut Kaya
Sold out!
```

Evidence 5:
Program code for Task 2.1. [4]

- initialise and parse records into appropriate data structure (Python list/array) (compensate for 1-based index/subscript)
- linear search code
- conditions to check stock level
- status message for each condition

Evidence 6:
Screenshot of output. [2]

- output for available
- output for sold out

Task 2.2
A customer wishes to maximise value for money *and* sample as many pastry types as possible given a budget of $x, subject to pastry availability. Write program code to determine the possible combination(s) of pastry choices and the amount spent. You should include as comments your strategy for determining the optimal combination(s).

Sample execution:
```
Enter budget($): 1
Peanut Butter: 1
Amount spent($): 0.90

Enter budget($): 3
Pumpkin Toast: 1
Peanut Butter: 1
Sweet Bean: 1
Amount spent($): 3.00
```

Evidence 7:
Program code for Task 2.2. [8]

- variable to accumulate amount spent / update amount remaining
- condition to compare amount spent with budget
- condition to check if purchase allowed (quantity > 0)
- update pastry purchased
- update pastry quantity
- output items and amount spent summary
- comments to determine optimal combination [2]

Evidence 8:
Screenshot of output for x = 20. [1]

- correct output

3. A new small library provides loan services to its members

- books with quota of 10 items and loan period of 21 days
- electronic resources with quota of 2 items and loan period of 14 days

Each library item can be renewed at most once.

Charges will be imposed for overdue items. For books, the overdue fine is $0.20 per item per day. Electronic resources incur an overdue fine of $0.50 per item per day.

The text file `ITEMS.dat` contains information about the library collection and has the following structure and sample records.

```
<Item ID>,<Type>,<Title>,<Status>
1,B,The 'A'rt of Computing,U
2,E,The 'A'rt of Computing,U
3,E,How to Save the World 2013 Edition,A
```

- `Item ID` is in the range 1 to 99999
- `Type` can be `B` (book) or `E` (electronic resource)
- `Title` has a 50-character length limit
- `Status` can be available (`A`) or on loan (`U`)

As a new small library, you may assume that it holds only a single copy of each item.

Task 3.1
For fast retrieval of item information, it is proposed to store the `LOAN.dat` records in a linked binary search tree using `Item ID` as the key field. Each item will be stored using a user-defined `Node` type with the following fields:

- `LeftP` - left pointer for the node
- `Data` - item information
- `RightP` - right pointer for the node

Using object-oriented programming techniques, write appropriate classes and methods to store the book information to a linked binary search tree data structure.

Evidence 9:
Program code for class definition, initialisation and insert methods. [9]

- node class declaration
- bst initialisation method code (appropriate root value)
- bst insert method code [3]
- parse text file records into appropriate data structure (Python list/array)
- sort records by item id (ensure balanced bst) [2]
- recursively insert middle record into bst [2]

Note: If you still do not know how to implement a binary search tree with and without the Node class, refer to https://github.com/limahseng/binary_tree

Task 3.2
Write a search method to process a query for information about an item given its `Item ID`.

Sample execution:
```
Enter item id: 1
Unavailable: The 'A'rt of Computing [Book]

Enter item id: 3
Available: How to Save the World 2013 Edition [Electronic]
```

Evidence 10:
Program code for Task 3.2. [3]

- bst binary search method code with appropriate parameter list
- terminating cases (found, not found)
- recursive cases (<, >)

Evidence 11:
Screenshots for annotated test cases. [2]

- case for available
- case for not available
- case for non-existent item id

Task 3.3
Write program code to display all on loan electronic resources in ascending `Item ID` order.

Sample execution:
```
2. The 'A'rt of Computing
```

Evidence 12:
Program code for Task 3.3. [3]

- correct inorder traversal code
- condition to check item type and status
- action to display items information

Evidence 13:
Screenshot. [1]

- correct sorted output for on loan electronic resources

The text file `TRANSACTION.dat` contains loan, return and renew records made by members over a period of time, and has the following structure and sample records:

```
<Action>,<Loan date in DDMMYYYY>,<Member ID>,<Item ID>
L,20140813,M007,3
R,20140813,M007,456
N,20140813,M123,8952
```

- `Action` can be `L` (loan), `R` (return) or `N` (renew)
- `Member ID` is in the range M001 to M999

Task 3.4
Write a function `query_member()` to determine if a member has any overdue loan(s) on a given date. If yes, output the overdue item'(s)' information, subtotal and total overdue fines payable.
Test your function with the following test data:

```
Member ID   Date
M123        24082014
M008        28082014
```

Sample execution:

```
Report for M123 as at 2014-08-24:
No overdue loan

Report for M008 as at 2014-08-28:
[B] Alibaba likes to eat bananas  : $0.80
[E] Mr Hong, The Pool Salon CEO   : $1.50
Total overdue payable: $2.30
```

Evidence 14:
Program code for Task 3.4. [7]

- loop through all records in text file to filter out those with given member id (append to Python list/array)
- for each filtered record, get item id to search though bst to determine type (book or electronic resource)
- if loan transaction, set return date to record date + loan period (21 for book or 14 for electronic resource)
- if renew transaction, compare given date with existing return date and compute overdue payable (if any), update return date to new return date (existing return date + new loan period)
- if return transaction, compare given date with existing return date and compute overdue payable (if any)
- loop through all filtered records to output overdue items info (type, title, overdue subtotal)
- compute and output total overdue amount payable

Evidence 15:
Screenshots for annotated test cases. [2]

- case for no overdue loan
- case for overdue loan with correct computed overdue amount payable

Task 3.5
It is proposed to store the loan information inside a queue data structure using an array.

Write program code to insert the `LOAN.dat` records into an object of the `Queue` class. Display the contents of your queue object. Assume maximum size of the array is 30.

Evidence 16:
Program code for Task 3.5. [4]

- class code for queue
- main program create queue object
- calls enqueue method by looping through records in text file
- calls display method by looping from front to rear of queue

Evidence 17:
Screenshot of output. [1]

- correct queue contents

Task 3.6
Write program code to process all loan records in the queue object and display summary information at the end of each transaction. Your program should also update the `Status` field in the item information stored in the binary search tree.

Evidence 18:
Program code for Task 3.6. [6]

- loop through and dequeue all queue items
- if loan transaction, traverse bst to update status to unavailable ('U') and output item loaned message
- if renew transaction, output item renewed message
- if return transaction, traverse bst to update status to available ('A') and output item returned message
- for renew and return transactions, adapt code for Task 3.4 to compute overdue fine payable
- inorder traversal to output item id, type, title and updated status

Evidence 19:
Screenshots for
- transaction summary
- verifying updated item status

[2]

- correct transaction summary
- call bst display method to verify updated item status

4. The 128 bits of an IPv6 address are represented in 8 groups of 16 bits each. Each group is written as 4 hexadecimal digits and the groups are separated by colons (`:`). An example of this representation is the address `2001:0db8:0000:0000:0000:ff00:0042:8329`.

For convenience, an IPv6 address may be abbreviated to shorter notations by applying the following rules, where possible:

- One or more leading zeroes from any groups of hexadecimal digits are removed. For example, the group `0042` is converted to `42`.
- Consecutive sections of zeroes are replaced with a double colon (`::`). The double colon may only be used once in an address, as multiple use would render the address indeterminate.

An example of application of these rules:
Initial address: `2001:0db8:0000:0000:0000:ff00:0042:8329`
After removing all leading zeroes: `2001:db8:0:0:0:ff00:42:8329`
After omitting consecutive sections of zeroes: `2001:db8::ff00:42:8329`

The loopback address, `0000:0000:0000:0000:0000:0000:0000:0001`, may be abbreviated to `::1` by using both rules.

Task 4.1
Write a function `abbreviate_ipv6(long_ipv6)` to convert an IPv6 address to its abbreviated form. Test your function with the data in `IPV6_LONG.txt`.

Sample execution:
```
2001:db8::ff00:42:8329
::1
```

Evidence 20:
Function code for `abbreviate_ipv6(long_ipv6).` [6]

- parse section tokens (slicing or delimiter)
- condition to check for leading zero(s)
- remove leading zero(s)
- condition to check for zero section(s)
- replace appropriate number of zeros sections by ::
- combine intermediate results to final abbreviated address

Evidence 21:
Screenshot of output. [2]

- correct output for all cases

Task 4.2

Write a function `expand_ipv6(short_ipv6)` to convert an abbreviated IPv6 address to its original form. Test your function with the data in `IPV6_SHORT.txt`.

Sample execution:

```
2001:0db8:0000:0000:0000:ff00:0042:8329
0000:0000:0000:0000:0000:0000:0000:0001
```

Evidence 22:
Function code for `expand_ipv6(short_ipv6).` [6]

- correct loop handling across sections
- compute number of leading zeros to insert
- prepend correct number of leading zeros
- compute number of zero sections to insert
- insert correct number of zero sections
- concatenate results

Evidence 23:
Screenshot of output. [2]

- correct output for all cases in text file

Task 4.3
Write a recursive function `ipv6_hex2dec(short_ipv6)` to convert an abbreviated IPv6 address to decimal.
Note: You may not use the built-in `int(s, 16)` function which converts a hexadecimal string `s` to an integer.

Sample execution:

```
>>> ipv6_hex2dec(2001:db8::ff00:42:8329)
8193:3512:0:0:0:65280:66:33577
```

Evidence 24:
Function code for `ipv6_hex2dec(short_ipv6)`. Test your function with the IPv6 address `2607:f0d0:1002:51::4.` [5]

- make use of `expand_ipv6(short_ipv6)`
- correct terminating case condition
- correct terminatin case action
- correct recursive case condition
- correct recursive case action

Evidence 25:
Screenshot of output. [1]

- correct output in decimal

Task 4.4
Errors may occur during data communication. Devise validation tests for one group (16 bits = 4 hexadecimal digits) of an IPv6 address given its *decimal* representation.

Sample execution:

```
Enter group: 8193
Ok
```

Evidence 26:
Validation code. [5]

- boolean variable to determine valid decimal for IPv6 group
- loop to validate
- presence check
- data type check
- range check (0 <= x <= 65535)

Evidence 27:
Screenshots showing annotated test cases. [3]

- normal data
- boundary data (0, 65535)
- erroneous data (negative, more than largest range, non-numeric data type)

### END OF PAPER 1 ###