Project instructions for Data Structure (No.1)

Introduction

We will have two projects for this course. In each project, you are requested to complete three or four questions. For questions labeled "Report needed", you are requested to complete the tasks using C language, write a brief report to explain your design, and analyze the time complexity for each algorithm required. For questions labeled "Report not needed", you are requested to complete the tasks using C language, but you do not need to write the report. For questions labeled "Bonus", you may choose whether to complete them by yourselves. If you should choose to complete them and if your answer is (partially) correct, you will be given a maximal of 1 extra point for each bonus question. This can make up for your deduction of marks for mistakes in required questions, but your total points gained for all project questions may not exceed 20.

Each project counts for a total of five points in your final score. We will decide the points you obtain based on the correctness of your codes and report. Questions are originally designed by the teaching group and are of a relatively high difficulty. You will NOT find any existing solutions on the web, but you have to think very hard and try your best efforts to complete each task. We strongly recommend you to start as soon as possible; otherwise you may have a hard time when a deadline approaches.

Policy: Each student has another total of three free late days for submissions. The calculation of the free late submission days will be independent from the statistics for homework submissions. Submissions exceeding the free late days will NOT be considered.

IMPORTANT NOTICE: Discussions among students are encouraged, but plagiarism is STRICTLY PROHIBITED. We will check and treat it VERY SERIOUSLY. Once a plagiarism is found, all students involved into the event will obtain a FAILURE GRADE for this course.

Deadlines:

Deadline for Project 1: 11:59 pm, June 5, 2016

Project 1: Lists and Trees

Question 1.1 (Report needed)

(1) Indexing.

Given a page of English text, create an indexed list of words sorted alphabetically. (If you do not know what is Alphabetical order, see here: http://en.wikipedia.org/wiki/Lexicographical_order

One test case for example:

Input:

The input is a file ’**input1.txt**’. The file only contains English letters and other legal ascii code characters.

Output.txt:

The output should be a file ‘**dictionary.txt**’, with each line keeps the following format:

```
…
Alphabet:        word1        word2        word3        ...
…
```

where Alphabet should be one of the letters from A-Z and the alphabets should be in alphabetically order from the first line to the last line. In addition, the following words should start with the leading alphabet and should also be sorted alphabetically in ascending order. In this question, **the words should be case insensitive.**

For example:

Input1.txt:

“To be, or not to be, that is the question.”

Dictionary.txt:

```
B: be
I: is
N: not
O: or
Q: question
T: that the to
```

(2) Coding

Calculate the appearance frequency of each alphabet **(case sensitive this time)** in the text. Based on your calculation, use Huffman coding tree to design a Huffman coding scheme for this text (punctuation are included for convenience).

Input:

The input is the file ’**input1.txt**’. The file only contains English letters and other legal ascii code characters.

Output:

The output should be a file '**huffman.txt**', with each line keeps the following format:

Alphabet: freq code...

where Alphabet should be one of the letters from A-Z and the alphabets should increase alphabetically from first line to the last line. *freq* is the appearance frequency of the alphabet and *code* is the Huffman code of that alphabet.

Question 1.2 (Report needed)

Consider the emergency department of a hospital. When a patient arrives, he will first receive initial check and be determined the emergency level of his/her physical condition (e.g., a patient may be judged as “critical”, “urgent”, “semi-urgent” or “non-urgent”). All patients wait for doctors in a queue, but once there is a vacancy to see the doctor, the patient with the most emergent level will be called to see the doctor first. If multiple patients are equally emergent, then the first arriver will be the first to see the doctor. During the waiting process, his/her emergency level of physical condition may change (once it changes, the patient is regarded as a new arriver in the new queue (with arrival time being the time of change level). Patients may also choose to leave the queue by themselves.

In this problem, you are required to complete the following two tasks:

We assign each patient with his/her name, ID and emergency level. A higher value of emergency level means a more serious physical condition and should be considered with a higher priority. Use two methods to implement this priority queue.

(A) Heap. The priority of a node represents the patient’s emergency level. You are required to support the following functions: (For more information about heap, please refer to the materials from the Fudan elearning website.)

enqueue (a new patient arrives),
dequeue (a patient sees the doctor),
updatePriority (the emergency level changes),
remove(key) (a patient leaves the queue before he/she sees the doctor).

(B) Multiple queues using linked list. Each queue serves for patients with the same emergency level. Only when the queue with a higher level is empty, the patients in the lower level emergency queue begin to see the doctor. You are required to maintain k queues using linked list (where k is the number of possible emergency levels), each supporting:

enqueue (a new patient arrives),
dequeue (a patient sees the doctor),
remove(key) (a patient leaves the queue before he/she sees the doctor).
switchQueue (the emergency level changes).

Input:

In each line of ‘<u>**input2.txt**</u>’, you receive a message concerning an operation of a patient or doctor. Possible cases include:

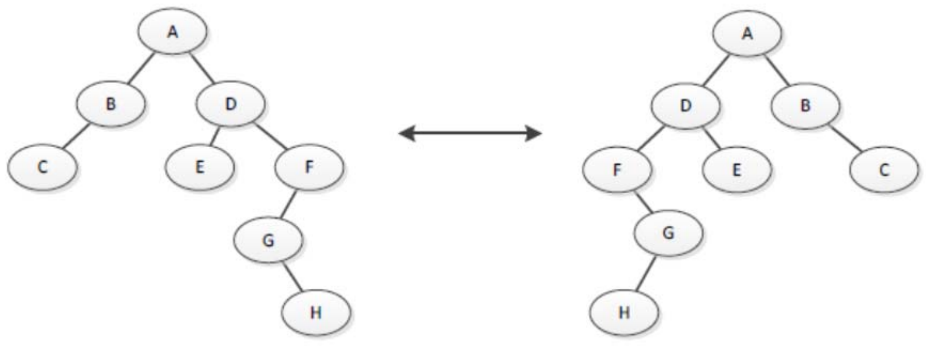
enqueue, Tom, 123, 4 (a patient named Tom with ID 123 arrives at the hospital. His emergency is 4).
quit, 123 (the patient with ID 123 quits the queue).
update, 123, 5 (the patient with ID 123 changes his/her emergency level to 5).
dequeue (a doctor is available and one selected patient should see the doctor)
print (representing a request of printing out all waiting patients in order of time to be expected to see the doctor)

Output:

You don’t need to have any output for input cases 1, 2 and 3. For input case 4, you should print the name and ID of the patient to see the doctor to file ‘**output2.txt**’. For input case 5, you should print out to the screen the sequence of patients with their priority from highest to lowest, using the heap and multiple queue implementations, respectively.

Question 1.3 (Report NOT needed)

Build the mirror of a given binary tree and calculate the number of nodes that have 2 children. The following figure shows the concept of ‘mirror’.



Test cases: Input:

Given the preorder and inorder traversal of a binary tree, you should first rebuild that tree.

Preorder: *Y A L B E C D W X M*

Inorder: *B L E A C W X D Y M*

These two orders are stored as two lines in the file “input3.txt”, and the outputs are stored in the file “output3.txt”.

Output:

Print out the postorder traversal of the mirror of the original tree and the number of nodes with 2 children in the mirror.

Question 1.4 (Report needed)

Input two decimals, or two integers, or one decimal and one integer, each number being of an arbitrary length. Denote them as *x* and *y*. You are required to use linked lists or strings to store these two decimals, each element storing one digit only (or the decimal point). You are required to implement one or several functions that supports operations plus, minus and multiplication, i.e., *x*+ *y*, *x* – *y* and *x* * *y*. If *y* is an integer, you are also requested to calculate x^y. Your outcome should archive the best accuracy possible.

Note that the multiply operation is hard. You are suggested to try your best to finish it.

Input:

Input4.txt

Each line of the file should be:
<number1> <number2>
, where <number1> and <nubmer2> are the input numbers of x and y respectively. The two numbers are separated with space.

Output:

Output4.txt

Each line of the file should be the calculation outcomes for the corresponding line in input.txt. Each calculation result should be separated with a space from the other.

On condition that the input y is an integer, you should have <x+y> <x-y> <x*y> <x^y> in the corresponding line of output.txt On condition that the input y is not an integer, you should have <x+y> <x-y> <x*y>.

For example:

Input4.txt:

```
/*1*/ 1048576 100
/*2*/ 3.141592653589793 3.141592653589793
```

Output4.txt:

```
/*1*/ 1048676 1048476 104857600
      11481306952742545242328332011776819840223177020886952004
      77642736825766261392370313856659486316506269918445964638
      98746277344711896086305533142593135616665318539129989145
      31228000068877914824004487142892699006348624478161546364
      63883639473170260404663539709049965581623988089446296056
      23311649536164221970332681344168908984458505602379484807
      91405890093477650042900271670662583052200813223628129176
      12678833172065989953964181270217798584040421598531832515
      40889433902091920554957783589672039160081957216630582755
      38042558372601552834878641943205450891527578388262517543
      5528800822842770817965453762184851149029376
/*2*/ 6.283185307179586 0 9.869604401089357120529513782849
```