Data Structures (English Version)

Groups B, D and I

Extraordinary Call, 2020

Rules for the Exam:

You must program solutions for each of the three exercises, test them and submit them to the automatic judge accessible at *http://exacrc.fdi.ucm.es/domjudge/team*

In the judge you will identify yourself with the username and password you received at the beginning of the exam. The username and password you have been using during the continuous assessment are **not** valid.

Write your **name and surnames** in a comment on the first line of every file you change in the templates. The link of the templates will be provided at the beginning of the exam.

Your solutions will be evaluated by the teacher regardless of the automatic judge's verdict. In doing so, the teacher will take into account exclusively the last submission you made of each exercise.

Exercise 1 (3 points). The merge operation on two increasingly ordered lists l1 and l2 is another increasingly ordered list that contains exactly all the elements of l1 and all the elements of l2. For example:

`l1:`

5	5	8	10	12	14	16	18

`l2:`

3	5	7	8	17

`Merge Operation result:`

3	5	5	5	7	8	8	10	12	14	16	17	18

You want to add a new merge operation to the List ADT implementation based on double-linked nodes. This operation has the following properties:

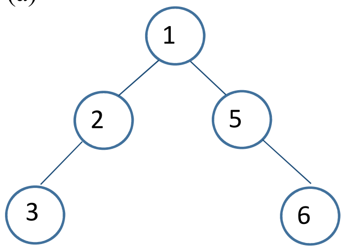
- It assumes that the l1 receiver list (that is, the list on which the operation is invoked) is sorted in increasing order.
- It receives the l2 list as a parameter, and this list must also be ordered in increasing order.
- After its execution, the l1 receiver list must contain the result of merging l1 and l2. In addition, l2 must be empty.

This operation should neither use, directly or indirectly, dynamic memory management operations (new, delete), nor make any assignment between node contents. Apart from implementing the operation, you must determine its complexity justifying it.

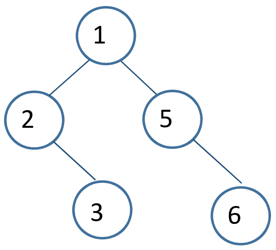
Exercise 2 (2 points). The *height* of a binary tree is: (i) 0 when the tree is empty; (ii) 1 + the maximum height of the left child and the height of the right child in another case. Moreover, the *inclination* of a binary tree is (i) -1, when the height of the left child is greater than the height of the right child; (ii) +1, when the height of the left child is less than the height of the right child; and (iii) 0 otherwise. Also, the inclination of a node is the inclination of the sub-tree of which the node is the root. Finally, the *character* of a binary tree is -1 if there are more nodes with inclination

-1 than with inclination +1, +1 if there are more nodes with inclination +1 than with inclination -1, and 0 otherwise. For example, let's consider the following trees:

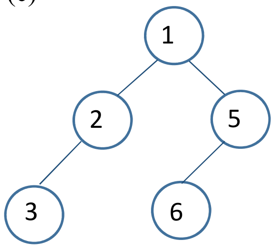
(a)



(b)



(c)



The character of the tree (a) is 0, since the inclinations of the nodes labeled by 1, 3 and 6 are 0, the inclination of the labeled by 2 is -1, and the inclination of the labeled by 5 is +1. However, the character of tree (b) is +1, since the inclinations of the nodes 1, 3 and 6 are 0, and the inclinations of the nodes 2 and 5 are +1. Finally, the character of tree (c) is -1 (the inclinations of 1, 3 and 6 are also 0, but those of 2 and 5 are -1).

Implements an algorithm that, given a tree *a*, returns its character. Apart from implementing this algorithm, you must determine its complexity with justifying it.

Exercise 3 (5 points). We have been commissioned to develop the order management system for the bar-restaurant *SpicyChips*. As part of this development, we have decided to set up an ADT *CommandManagement* in charge of managing the orders (commands) of the customers. In this system, the commands are associated to customer services, and are attended by strict order of request. Also, the same service can generate several commands (in the sense that customers to whom the service is provided can order several times during the service). The ADT will include the following operations:

- create: It creates an empty command management system..
- register_service(*s*): It registers a new service, identified by s. If there is another service in progress with the same identifier, it raises an error (*EServiceExists*).
- register_product(*p, c*): It adds the product p to the menu. The parameter c represents the price of the product. If the product already exists, it updates its price to c.
- register_commands(*s,ps*): It registers a command associated with the service. The list of products requested is given by *ps* (one product unit is requested for each item on the list. If several units of a product are desired, that product will appear in the list as many times as units ordered). If the service does not exist, or the restaurant does not serve any of the products ordered, it signals an error (*ERegisterCommand*).
- process_command(): It handles the following pending command (emphasize that the commands are processed by strict order of request: first the oldest, then the most recent). If there are no commands to attend to, it points out an error (*ENoCommands*)
- close_service(*s*)→*c*: It closes the service, returning the bill associated with that service. This bill consists of: (i) a list, ordered by product name, in which, for each product consumed by the customers, it indicates name of the product and the number of provided units of this product during the service, which is being closed; and (ii) the total amount to be paid. The operation also removes all the information about the service from the system. If the service does not exist, or there is still a command in progress for that service, it signals an error (*ECloseService*).

The implementation of all these operations must be as efficient as possible. Besides implementing them, their complexities must be justifiably determined.