CS202, Spring 2023

Homework 4 - Balanced search trees and hashing

Due: 15/05/2023

Before you start your homework please read the following instructions carefully:

FAILURE TO FULFIL ANY OF THE FOLLOWING REQUIREMENTS WILL RESULT IN A GRADE SCORE OF 0 (zero) WITHOUT ANY CHANCE OF REDEMPTION.

- See the course page for any late submission policies and Honor Code for Assignments.
- Upload your solutions in a single ZIP archive using the Moodle submission form. Name the file as name_surname_studentID_hw4.zip.
- Your ZIP archive should contain **only** the following files (meaning no files generated by your IDE, operating system or binary):
    - **name_surname_studentID_hw4.pdf**, the file containing the answers to Question 1(no photos or handwriting).
    - .cpp(name it/them as: **name_surname_studentID_filename.cpp** and/or **name_surname_studentID_filename.h**) file/s which contain/s the C++ source code and the **Makefile**.
    - Do not include characters such as Ç, Ğ, I, İ, Ö, Ş and Ü in your filenames and especially in your .cpp/.h. Use **ONLY ASCII** characters.
    - Do not forget to put your name, student id, and section number in all of these files. Comment your implementation well. Add a header (see below) to the beginning of each file:

```
/**
 * Title: Balanced search trees and hashing
 * Author : Name & Surname
 * ID: 12345678
 * Section : 1
 * Homework : 4
 * Description : description of your code
 */
```

    - Do not put any unnecessary files such as the auxiliary files generated from your preferred IDE.
- Your code must compile.
- Your code must be complete(if your code does not work as a whole it will not be accepted).
- Your code must run on the **dijkstra.ug.bilkent.edu.tr** server.
- You are responsible for any clarification given by the TA through any means of communication.
- For any question related to the homework contact your TA: *saeed.karimi@bilkent.edu.tr*

Question 1 (30 points)

Assume that you have the following ***balanced-searched tree*** implementations:

a) AVL tree
b) 2-3 tree
c) 2-3-4 tree

Starting with an empty balanced search tree, insert the following keys into the tree in the ***given order***:

7, 91, 3, 24, 12, 33, 45, 61, 57

After inserting, delete the following keys from the tree in this order: 3, 61, 45.

Note: while deleting an internal node, its inorder successor should be used as the substitute if needed.

Show the underlying tree for the AVL, 2-3, and 2-3-4 implementations after each insertion and deletion.

Question 2 (70 points)

For this question, you will use hash tables to implement a simple snack inventory application for a kindergarten. In this scenario, you will be helping the teacher keep track of the combination of snacks they can give to their students in a day/week.

Your program must accept a sequence of commands of the following forms as input, one command to a line:

→ ***S <snack1 > :*** Create a snack record of the specified name. You may assume that there are no duplicates of the same snack.

→ ***C <snack1> <snack2> :*** Record two snacks that can be given together.

→ ***D <snack1> <snack2> :*** Record that the two specific snacks can no longer be given to the students on the same day (ex: for health reasons).

→ ***L <snack1> :*** Print out the names of the snacks that can be given as an accompaniment to the specified snack.

→ ***Q <name> <course_name> :*** Check whether that snack1 can be given together with snack2. If so, print "Yes", if not print "No".

→ ***X :*** Terminate the program.

Example of possible input (given as example, may not necessarily be valid in real life):

Input	**Output**
S Apple	Snack 'Apple' created
S Chips	Snack 'Chips' created
S Cheese	Snack 'Cheese' created
S Crackers	Snack 'Crackers' created
C Apple Chips	
C Cheese Crackers	
C Crackers Apple	
C Crackers Chips	
L Crackers	Chips Cheese Apple
L Chips	Crackers Apple
D Crackers Cheese	
L Crackers	Chips Apple
Q Apple Chips	Yes
Q Cheese Crackers	No

X

<u>You must:</u>

- Define a `Snack` class which has at least two fields; one field for the name and one field for the linked list of snacks it can be accompanied with.
- Store the snack accompaniments of each snack in a linked list, not in an array. The list must be a list of `Snack` objects, <u>not</u> a list of their names, as strings.
- Define a `SnackHashing` class that creates a hash table which indexes each `Snack` object by using the name field as key. This hash table implementation will use separate chaining and table size could be 11 for the purposes of this assignment. The hash function will be $h(x)$ $=$ (sum of the ASCII codes of each letter in the name) mod (table size).
- Define an `Accompaniment` class which has at least three fields. First field is for the accompaniment name. The accompaniment name will be the concatenation of the names of two snacks in the group in alphabetical order. For example, the group of snacks "Apple" and "Chips" will be "AppleChips". The second field will be a pointer to the node corresponding to the Apple object in the linked list of Chips' group/accompaniments, and the third field will be a pointer to the node for Chips in the linked list of Apple's group/accompaniments.
- Define an `AccompanimentHashing` class that creates a hash table which indexes each `Accompaniment` object by using the accompaniment name field as key. This hash table implementation will use quadratic probing and the table size could be 71 for the purposes of this assignment. The hash function will be $h(x)$ $=$ (sum of the ASCII codes of each letter in the accompaniment name) mod (table size).

<u>Executing commands:</u>

<u>To execute an "S" command,</u> create a `Snack` object for the name, and save it in the `SnackHashing` hash table by using the name as key.

<u>To execute an "C" command:</u>

- Find the two given `Snack` objects in the `SnackHashing` hash table.
- Add each snack to the front of the linked list of the group/accompaniment of the other snack.
- Construct the accompaniment name by using two names.
- Create an `Accompaniment` object for the accompaniment name. Connect the first pointer to the node corresponding to the second snack in the linked list of the first snack's group/accompaniment, and connect the second pointer to the node corresponding to the first snack in the linked list of the second snack's group/accompaniments.
- Save this `Accompaniment` object to the `AccompanimentHashing` hash table by using the accompaniment/group name as key.

<u>To execute a "D" command:</u>

- Construct the accompaniment name by using two snack names.
- Find the `Accompaniment` object in the `AccompanimentHashing` hash table by using accompaniment name as key.
- Find two `Snack` objects by using the appropriate pointers and delete both snacks from each other's accompaniment/group list.
- Delete the `Accompaniment` object from `AccompanimentHashing` hash table.

<u>To execute an "L" command:</u> find the `Snack` object in the `SnackHashing` hash table, and loop through the list of accompaniments.

<u>To execute a "Q" command:</u> construct the accompaniment name by using two names and look it up in the `Accompaniment` hash table by using the accompaniment name as key.

Remember to implement these methods such that their asymptotic running times are as expected (from the involved data structure and operation). For instance, retrieval, insertion and deletion operations should take expected constant time for hash tables.

<u>Input/Output:</u>

You may assume that the input is correctly formatted. That is:

- Each line consists of a command character 'S', 'C', 'D', 'L', 'Q', or 'X' followed by a blank followed by one or two snack names separated by a blank. A snack name is a sequence of alphabetic characters.
- Any snack name mentioned in an C, D, L, or Q command has been already created by a S command.
- The sequence of commands ends with X.

What, if anything, you want to do about invalid inputs is up to you. However, the program should do the right thing under the following circumstances:

- A snack is grouped or ungrouped with itself (i.e: D Apple Apple or C Cheese Cheese). In this case, the program should do nothing; it should not add the snack to his own accompaniments/group.
- A snack not existing in an accompaniment/group is taken out from it. In that case, the program should do nothing.