CS 202, Fall 2020

Homework #4 – Balanced Search Trees and Hashing

Due Date: December 22, 2020

Important Notes

Please do not start the assignment before reading these notes.

- Before 23:55, December 22, upload your solutions in a single **ZIP** archive using Moodle submission form. Name the file as `studentID_hw4.zip`.
- Your ZIP archive should contain the following files:
  - `hw4.pdf`, the file containing the answers to Questions 1, 2 and 3,
  - `Person.h`, `Person.cpp`, `PersonHashing.h`, `PersonHashing.cpp`, `Friendship.h`, `Friendship.cpp`, `FriendshipHashing.h`, `FriendshipHashing.cpp`, `main.cpp` files which contain the C++ source codes, and the `Makefile`.
  - Do not forget to put your name, student id, and section number in all of these files. Well comment your implementation. Add a header as in Listing 1 to the beginning of each file:

Listing 1: Header style

```
/**
 * Title: Balanced Search Trees and Hashing
 * Author: Name Surname
 * ID: 21000000
 * Section: 0
 * Assignment: 4
 * Description: description of your code
 */
```

  - Do not put any unnecessary files such as the auxiliary files generated from your favorite IDE. Be careful to avoid using any OS dependent utilities.
  - **You should prepare the answers of Questions 1, 2 and 3 using a word processor (in other words, do not submit images of handwritten answers).**

- Please use the algorithms as exactly shown in lectures.
- Although you may use any platform or any operating system to implement your algorithms and obtain your experimental results, your code should work on the dijkstra server (dijkstra.ug.bcc.bilkent.edu.tr). We will compile and test your programs on that server. Please make sure that you are aware of the homework grading policy that is explained in the rubric for homeworks.
- This homework will be graded by your TA, Hasan Balcı. Thus, please contact him directly for any homework related questions.

Attention: For this assignment, you are allowed to use the codes given in our textbook and/or our lecture slides. However, you ARE NOT ALLOWED to use any codes from other sources (including the codes given in other textbooks, found on the Internet, belonging to your classmates, etc.). Furthermore, you ARE NOT ALLOWED to use any data structure or algorithm related function from the C++ standard template library (STL).

Do not forget that plagiarism and cheating will be heavily punished. Please do the homework yourself.

Question 1 – 10 points

(a) [*5 points*]

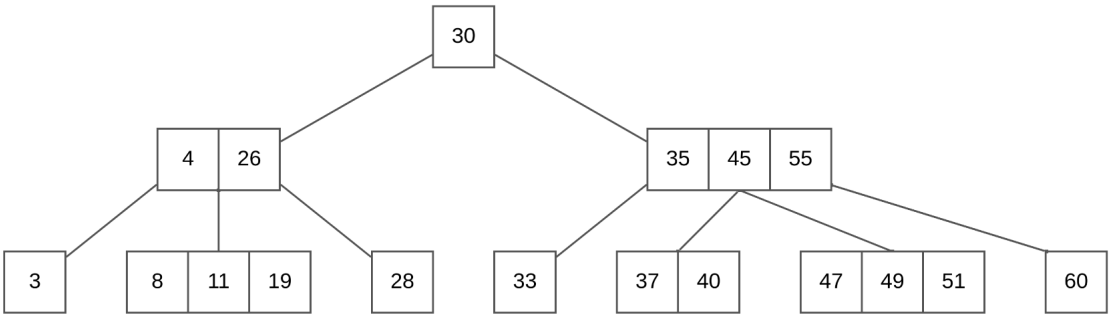

Figure 1: 2-3-4 tree

Draw an equivalent red-black tree of the 2-3-4 tree in Figure 1. Clearly indicate red and black nodes.

(b) [*5 points*] Draw the resulting 2-3-4 tree after inserting 50 into the original 2-3-4 tree in Figure 1.

Question 2 – 10 points

(a) [*3 points*] What is the maximum number of keys that a 2-3 tree of height h can hold?

(b) [*3 points*] If you start with an empty 2-3-4 tree and insert the letters in English alphabet A, B, C, D, ... in alphabetical order, the first time the tree would grow to height 2 would be after inserting D. After inserting which letter would the tree grow to height 3 for the first time? Show the 2-3-4 tree before and after inserting that letter.

(c) [*2 points*] Assuming you have a red-black tree with n elements, how fast can you sort those elements using the tree?

(d) [*2 points*] Is every subtree of a red-black tree also a red-black tree? Explain your answer with one sentence.

Question 3 – 15 points

(a) [*5 points*] Assume that you have an array of length N that consists of unique integers. Describe an algorithm in plain English (no pseudo-code is needed) to find two integers in this array such that their sum is equal to a given target integer. Your algorithm must work in expected time *O(N)*. You can assume that exactly one pair of elements in the array sums to the given target integer.

(b) [*10 points*] Insert the keys 22, 14, 30, 15, 11 and 18 into a hash table with 7 slots in the given order by using the following hash function h1():

```
int h1 (key) {
    int x = (key + 5) * (key + 5);
    x = x/20;
    x = x + key;
    x = x mod 7;  // take mod 7
    return x;
}
```

Resolve the collisions with **Linear Probing** *(5 points)* and **Quadratic Probing** *(5 points)*. When necessary, use the conventions in the related lecture slides. Show the final content of the hash tables in the following form:

Slot	0	1	2	3	4	5	6
Content							

Question 4 – 65 points

In this part, you will use hash tables to implement a simple social network application. Remember that insertion, deletion, and retrieval operations are to run in expected constant time for hash tables.

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

- P <name> – Create a person record of the specified name. You may assume that no two people have the same name.
- F <name1> <name2> — Record that the two specified people are friends.
- U <name1> <name2> — Record that the two specified people are no longer friends.
- L <name> — Print out the friends of the specified person.
- Q <name1> <name2> — Check whether the two people are friends. If so, print "Yes"; if not, print "No".
- X – terminate the program.

For instance, this is one possible input:

P Ali
P Veli
P Ahmet
P Mehmet
F Ali Veli
F Ahmet Mehmet
F Mehmet Ali
F Mehmet Veli
L Mehmet
L Veli
U Mehmet Ahmet
L Mehmet
Q Ali Veli
Q Ahmet Mehmet
X

and this is the corresponding output:

Veli Ali Ahmet

Mehmet Ali
Veli Ali
Yes
No

You **must**

- Define **Person** class which has at least two fields; one field for the name and one field for the linked list of friends.
- Store the friends of each person in a linked list, not in an array. The list must be a list of **Person** objects, not a list of their names, as strings.
- Define **PersonHashing** class that creates a hash table which indexes each **Person** object by using the name field as key. This hash table implementation will use separate chaining and table size will be 11. The hash function will be h(x) = (sum of the ASCII codes of each letter in the name) mod (table size).
- Define **Friendship** class which has at least three fields. First field is for the friendship name. The friendship name will be the concatenation of the names of two people in the friendship in alphabetical order. For example, the friendship name of "Veli" and "Ali" will be "AliVeli". The second field will be a pointer to the node corresponding to Ali in the linked list of Veli's friends, and the third field will be a pointer to the node corresponding to Veli in the linked list of Ali's friends. The goal here is to quickly locate a neighbor node in an adjacency list.
- Define **FriendshipHashing** class that creates a hash table which indexes each **Friendship** object by using the friendship name field as key. This hash table implementation will use quadratic probing and table size will be 71. The hash function will be h(x) = (sum of the ASCII codes of each letter in the friendship name) mod (table size).

Executing commands

To execute a "P" command, create a **Person** object for the name, and save it in the **PersonHashing** hash table by using the name as key.

To execute an "F" command:

– Find two **Person** objects in the **PersonHashing** hash table.

– Add each person to the front of the linked list of the friends of the other person.

– Construct the friendship name by using two names.

- Create a **Friendship** object for the friendship name. Connect the first pointer to the node corresponding to the second name in the linked list of first name's friends, and connect the second pointer to the node corresponding to the first name in the linked list of second name's friends.

- Save this **Friendship** object to the **FriendshipHashing** hash table by using the friendship name as key.

To execute a "U" command:

- Construct the friendship name by using two names.

- Find the **Friendship** object in the **FriendshipHashing** hash table by using friendship name as key.

- Find two **Person** objects by using the appropriate pointers and delete both person from each other's friend list. (***Hint***: to make the deletions in constant time, think about your linked-list structure)

- Delete the **Friendship** object from **FriendshipHashing** hash table.

To execute an "L" command", find the **Person** object in the **PersonHashing** hash table, and loop through the list of friends.

To execute a "Q" command, construct the friendship name by using two names and look it up in the **Friendship** hash table by using the friendship name as key.

Input/Output

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

- Each line consists of a command character 'P', 'F', 'U', 'L', 'Q', or 'X' followed by a blank followed by one or two names separated by a blank. A name is a sequence of alphabetic characters.

- Any name mentioned in an F, U, L, or Q command has been already created by a P 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 person friends or unfriends himself. In this case, the program should do nothing; it should not add the person to his own list of friends.

- A person unfriends someone who is not a friend. In that case, the program should do nothing.

Your `main.cpp` file must take its input from a text file named "**input.txt**" in the same directory as the program and execute each command in the text file in order.

At the end, write a basic Makefile which compiles all your code and creates an executable file named `hw4`. Check out these tutorials for writing a simple make file: tutorial 1, tutorial 2. Please make sure that your `Makefile` works properly, otherwise you will not get any points from Question 4.