Middle East Technical University Department of Computer Engineering

CENG 213

Data Structures

Spring 2020-2021

Homework 3

Due date: 22nd of June 2021, Tuesday, 23:55

1 Objective

This homework aims to help you get familiar with hashing and heap data structure. In the first part, you are expected to implement a template hash table with generalized key-value pair. Then in the second part, you are asked to implement a binary heap by using a specialized form of the hash structure int the first part.

Keywords: *Hashing, double hashing, binary heap*

2 Problem Definition

As the chief of homicide bureau, you are charged to find the murderer(s) in a murder case. For this task, you are planning to use the advanced techniques that you learned in Ceng213 course. Each murder suspect is held by a unique avatar name in the system. Also, according to the statements of eyewitnesses, each suspect has been assigned a probability of committing the murder. Depending on new evidences, the probability values may need to be updated. For that reason, you need a dynamic data structure. Also, you want to make an evaluation on each suspect in the order depending on their probability values. Therefore, you decide that using a maximum heap would be very helpful and efficient. Also, you see that there will be some parts that you need to hold <*avatar, heap detail*> where *'heap detail'* can be any info that you should note during the heap operations (This info totally depends on your implementation). In order not to affect the performance of heap operations negatively, you decide that using hashing for such pairs would be the best choice.

To sum up, first you need to implement a general hash table as described in the below parts. Later, you need to implement a maximum heap to order the suspects depending on their probability values and do some operations on that heap as described in the following parts.

3 Part I - Hash Table Implementation

The hash table data structure used in this assignment is implemented as a class template `HashTable` with the template arguments `Key` and `Value`, which is used as the type of the key and the type of the value mapped to that key (it will be a probability value in the second part of the assignment), respectively. The items of the hash table are implemented as the class template `HashTableItem` with the template argument

`Key`, which is the type of the key, and `Value`, which is the type of the value stored in items. `HashTableItem` class is the basic building block of the `HashTable` class. `HashTable` class has (in its private data field) a vector of HashTableItems (namely table) which keeps the items of the hash table, and an integer (namely tableSize) which keeps the size of the hash table, and another integer (namely numberOfItems) which keeps the number of items in the hash table. The `HashTable` class has its definition in *HashTable.h* and its implementation in *HashTable.cpp* file and the `HashTableItem` class has its in *HashTableItem.h* file.

3.1 HashTableItem

`HashTableItem` class represents items that constitute hash tables. A HashTableItem has a variable of type `Key` (namely `key`) to keep the key of the item, a variable of type `Value` (namely `value`) to keep the value of the item, a boolean variable (namely `deleted`) to keep the information that the item is deleted (i.e. marked as deleted) or not, and another boolean variable (namely `active`) to keep the information that the item is active (i.e. currently in use), or not. The class has only the default constructor and it initializes the `key` and `value` variables to their default values and the `deleted` and `active` variables to false. The class also has the overloaded output operator. They are already implemented for you. You should not change anything in file *HashTableItem.h*.

3.2 HashTable

`HashTable` class implements a hash table data structure. Previously, data members of `HashTable` class have been briefly described. Their use will be elaborated in the context of utility functions discussed in the following subsections. You must provide implementations for the following functions that have been declared under indicated portions of *HashTable.cpp* file.

3.2.1 `HashTable()`

This is the default constructor. You should make necessary initializations in this function. Initial size of the hash table should be 2.

3.2.2 `bool insert(const Key & key, const Value & value)`

You should insert the given key and value pair to the hash table. Check for the current load factor before insertion.

- If the current load factor before insertion is bigger than or equal to 0.5, then you should call `rehash()` function before insertion. Details of the `rehash()` function are given in the following sections.
- Apply double hashing for collision resolution.

This function should return a boolean that indicates if the insertion was successful or not.

- If there already exists an item with given key in the hash table, yet the value for that key is different, then update the value to the new item. Return true.
- If there already exists an item with given key in the hash table and the value is also the same, then do nothing. Just return false to show the operation is unsuccessful.
- If you can not insert the key - value pair because of infinite probing, then return false to show that the operation is unsuccessful.
- Otherwise, insert the key - value pair and return true.
- You should do the rehashing for the successful operations, if the operation is unsuccessful and you did a rehashing for it, undo rehashing.

3.2.3 `bool remove(const Key & key)`

You should remove (i.e. mark as deleted) the item with given key from the hash table. This function should return true if such an item exists and it is removed successfully. Otherwise, it should return false.

3.2.4 `bool contains(const Key & key) const`

This function should return true if an item with given key exists in the hash table. If there exists no such item in the hash table, it should return false.

3.2.5 `const HashTableItem<Key, Value> * find(const Key & key) const`

You should search for the item in the hash table with given key and return a pointer to that item (i.e. a pointer to that HashTableItem object). If there exists no such item in the hash table, you should return NULL.

3.2.6 `int getTableSize() const`

This function should return an integer that is the size of the hash table.

3.2.7 `int getNumberOfItems() const`

This function should return an integer that is the number of items in the hash table.

3.2.8 `void print()`

This function is already implemented for you. It prints the whole hash table to the standard output (`stdout`) using a special output format. The output of this function includes size of the hash table, number of items in the hash table, and list of all items in the hash table by using overloaded `operator<<` of the `HashTableItem` class (which is also already implemented for you).

3.2.9 `int hashFunction(const Key & key) const`

This function should implement the first hashing formula. It gets a key and calculates the hash code (i.e. an integer) for that key. You should implement the following formula: $h_1(key) = (key[0] + key[1] + ... + key[N-1]) \ \% \ tableSize$, where N is the length of the key. You should use this hash function for implementing the hash table functions like insert, remove, contains, etc.

3.2.10 `int hashFunction2(int key) const`

This function should implement the second hashing formula (it takes the value produced by the `hashFunction()` as its argument). If there occurs a collision during the insertion, then you need to do double hashing. For that, you should implement the following formula: $h_2(key) = (h_1(key) * tableSize - 1) \ \% \ tableSize$.

3.2.11 `void rehash()`

At the beginning of *HashTable.h* file, you are given an array of prime numbers (namely `primeNumbers[]`). You should expand (resize) the hash table to the next prime number from that array and then **sequentially re-insert** all items in the hash table to the expanded hash table. Given list of prime numbers are enough for this assignment, you will not be asked to expand your hash table further.

4 Part II - Max Heap Implementation

The max heap data structure used in this assignment is implemented as the class template `ProbabilityHeap` with the template arguments `Key` and `Probability`, which is used as the type of the key and the type of the probability mapped to that key. Although this is a template implementation, in the run cases the `Key` will be a string refering to the avatar name of the suspect and `Probability` will be a float number refering to the probability of committing the murder. `ProbabilityHeap` class has nothing defined in its private data field on the contrary to `HashTable` class. It is expected you to fill the data members of this class according to your needs. The `ProbabilityHeap` class has its definition in *ProbabilityHeap.h* and its implementation in *ProbabilityHeap.cpp* file.

4.1 ProbabilityHeap

`ProbabilityHeap` implements a maximum heap data structure. It stores both key and probability for each item. The client of `ProbabilityHeap` can consume the items in order of Probability values one by one. Inside the `ProbabilityHeap`, hash table is necessary for access with key to the items on the heap and binary heap is necessary to order values for the probability query operations. Your `ProbabilityHeap` implementation should maintain the two data structures in parallel in order to support all the interface methods. The hash table(s) should keep references to the suspects on the probability heap so that the probability value of a suspect can be accessed in constant time. You should also keep references to the keys from the probability heap so that the references in the hash table are maintained after operations that update probability heap. Both structures should be updated whenever there is a change in ordering.

The const variable SUSPECT_NOT_ON_HEAP specifies the value to be returned when a key is not found on the heap. The other const variable HEAP_EMPTY is the value to be returned by the deleteMax method when the heap is empty.

You must provide implementations for the following functions that have been declared under indicated portions of *ProbabilityHeap.cpp* file.

4.1.1 `void insert(const Key & key, const Probability & probability)`

This method enables insertion of a suspect to the heap (and to the hash table in parallel). The method has no effect on the items that are already on the heap and only inserts new items. Note that depending on the current size of the heap, you may need to increase its size. You can again use the `primeNumbers` supplied at the beginning of *HashTable.h* file. Recall that with a binary heap implementation, insert method requires $O(logN)$ time where N is the number of items on the heap.

4.1.2 `const Key deleteMax()`

This method returns the key, namely avatar of the suspect which has the highest probability value. You may think that you want to remove a suspect from the suspecteds list just because you decided that s/he is the guilty one or the reverse, you want to eliminate him or her since s/he is innocent. In case of equal probability values, the order in which the items are consumed by deleteMax method is not important. Recall that with a binary heap implementation, deleteMax requires $O(logN)$ time where N is the number of items on the heap.

4.1.3 `const Probability & getProbability(const Key & key) const`

This method enables key based access to the probability values on the heap. This method is similar to the find method of the `HashTable` and contributes hash table behaviour to the probability heap. A frequent use of this method is to check if an avatar is among the suspecteds. This method provides constant time access like the find method of hash table data structure.

4.1.4 `void updateProbability(const Key & key, const Probability & newProbability)`

This is the method that enables updates to the probability values of the suspects on the heap. The method re-orders the heap if the probability updates require. The method has no effect if the suspect is not on the heap. This may require a re-ordering like deleteMax or insert methods therefore it will have $O(logN)$ worst case complexity with binary heap and hash table implementation.

5 Regulations

1. **Programming Language:** You will use C++.
2. Standard Template Library is allowed only for vector. However, you should not use it for any purpose other than storing items in `HashTable` class.
3. External libraries other than those already included are **not** allowed.
4. Those who do the operations (HashTable.insert, HashTable.remove, HashTable.contains, HashTable.find, ProbabilityHeap.insert, ProbabilityHeap.deleteMax, ProbabilityHeap.update) without utilizing the hash table and heap will receive **0 grade**.
5. Those who modify already implemented functions and those who insert other data variables or public functions and those who change the prototype of given functions will receive **0 grade**.
6. Those who use compile-time arrays or variable-size arrays (not existing in ANSI C++) will receive **0 grade**. Options used for g++ are ”-ansi -Wall -pedantic-errors -O0”. They are already included in the provided Makefile.
7. You can add private member functions whenever it is explicitly allowed.
8. **Late Submission Policy:** Each student receives 5 late days for the entire semester. You may use late days on programming assignments, and each allows you to submit up to 24 hours late without penalty. For example, if an assignment is due on Thursday at 11:30pm, you could use 2 late days to submit on Saturday by 11:30pm with no penalty. Once a student has used up all their late days, each successive day that an assignment is late will result in a loss of 5% on that assignment.

   No assignment may be submitted more than 3 days (72 hours) late without permission from the course instructor. In other words, this means there is a practical upper limit of 3 late days usable per assignment. If unusual circumstances truly beyond your control prevent you from submitting an assignment, you should discuss this with the course staff as soon as possible. If you contact us well in advance of the deadline, we may be able to show more flexibility in some cases.
9. **Cheating:** We have zero tolerance policy for cheating. In case of cheating, all parts involved (source(s) and receiver(s)) get zero. People involved in cheating will be punished according to the university regulations. Remember that students of this course are bounded to code of honor and its violation is subject to severe punishment.
10. **Newsgroup:** You must follow the newsgroup (odtuclass.metu.edu.tr) for discussions and possible updates on a daily basis.

6 Submission

- Submission will be done via CengClass (cengclass.ceng.metu.edu.tr).
- Don’t write a main function in any of your source files.

- A test environment will be ready in CengClass.
  - You can submit your source files to CengClass and test your work with a subset of evaluation inputs and outputs.
  - Additional test cases will be used for evaluation of your final grade. So, your actual grades may be different than the ones you get in CengClass.
  - Only the last submission before the deadline will be graded.