CS 202, Fall 2021

Homework #4 – Hashing

Due Date: Dec 28, 2021

Important Notes

Please do not start the assignment before reading these notes.

- Before 23:55, Dec 28, 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 your report.
  - C++ source codes (files HashTable.h and HashTable.cpp as well as the file that contains your main function as described below), 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

```
/**
 *  Author :  Name  Surname
 *  ID:  XXXXXXXX
 *  Section :  X
 *  Assignment :  4
 */
```

  - 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 (for example to measure the time).

- 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

Attention: For this assignment, you are allowed to use the codes given in our text-book 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 – 80 points

You are asked to implement a class named HashTable that uses a hash table with open addressing to store the items in a data collection. To simplify the implementation, we will consider only integer items where the items themselves are used as the key values.

The items should be stored in an array whose size is given as tableSize when the HashTable object is constructed. You should use the mod operation as the primary hash function:

$$hash(key) = key \bmod tableSize$$

The HashTable class should support insert, remove, and search operations, and should use a separate function named f as the collision resolution strategy:

$$h_i(key) = hash(key) + f(i) \bmod tableSize$$

where h_i is the array index used and i = 0,1,2,… is the iteration number for finding alternative cells in the array. Your implementation should resolve the collisions using linear probing, quadratic probing, and double hashing as follows:

- Linear probing: f(i) = i
- Quadratic probing: $f(i) = i^2$

- Double hashing: f(i) = i $hash_2$(key) where $hash_2$(key) = reverse(key) is the secondary hash function that reverses the digits of the key (e.g., reverse(1234) = 4321).

Your implementation should be in the files named HashTable.h and HashTable.cpp. You should define the following enumeration to indicate which collision resolution strategy is used:

```
enum CollisionStrategy { LINEAR, QUADRATIC, DOUBLE };
```

Your implementation of the HashTable class should have the following functions:

- HashTable::HashTable(const int tableSize, const CollisionStrategy option);
  Constructor that initializes the hash table with the given size. The collision resolution strategy is given as an option that will be used in the insert, remove, and search operations.

- HashTable::~HashTable();
  Destructor that cleans up any dynamic memory used.

- bool HashTable::insert(const int item);
  Inserts the given item (also used as the key value) into the hash table, and returns true if insertion is successful, and false otherwise.

- bool HashTable::remove(const int item);
  Removes the given item from the hash table, and returns true if removal is successful, and false otherwise.

- bool HashTable::search(const int item, int& numProbes);
  Searches the given item in the hash table, and returns true if search is successful (i.e., the item is found), and false otherwise. This function also returns the number of probes used during the search for this item.

- void HashTable::display();
  Displays the contents of the hash table in the following format:

```
0:
1:  39
2:
3:  22
4:  60
5:
...
```

(In each line, the first value indicates the array index, followed by colon, followed by the item value stored at that index (no item value is shown if the cell is empty).)

- void HashTable::analyze(int& numSuccProbes, int& numUnsuccProbes); Analyzes the current hash table in terms of the average number of probes for successful and unsuccessful searches, and returns these two values in the variables passed by reference. For analyzing the performance for successful searches, you can use the item values currently stored in the table for searching. For analyzing the performance of unsuccessful searches, you can initiate a search that starts at each array location (index), and count the number of probes needed to reach an empty location in the array for each search.
  Note: You are expected to implement the analysis for both successful and unsuccessful searches for both linear and quadratic probing. For double hashing, you are expected to implement the analysis only for successful searches. You can return -1 for numUnsuccProbes if the collision resolution strategy is selected as DOUBLE. (Exhaustively specifying unsuccessful searches becomes difficult for double hashing because of the use of the keys in the second hash function.)

You can define additional functions if necessary.

Question 2 – 20 points

Part 1: Describe briefly your design of the HashTable class and how you implement the collision resolution strategies (e.g., how to decide when to stop probing).
Note: You should carefully design the stopping conditions to avoid infinite loops where probing can cycle through the same sequence of array indices even though the hash table is not completely full. You should think about this before actually implementing the methods.

Part 2: Write a driver function (main function) that uses the HashTable class described above and simulates the table operations given in an input text file. The input file should specify table operations in separate lines in the following format:

Operation	Meaning
I 1234	Inserts item 1234 into the hash table. Your driver function should display "1234 inserted" if insertion is successful, and "1234 not inserted" if insertion is unsuccessful.
R 1234	Removes item 1234 from the hash table. Your driver function should dis-play "1234 removed" if removal is successful, and "1234 not removed" if removal is unsuccessful.
S 1234	Searches item 1234 in the hash table. Your driver function should display "1234 found after 5 probes" if search is successful, and "1234 not found after 8 probes" if search is unsuccessful (numbers just given as examples).

Then, prepare an example input text file, and test your HashTable class using a mixed sequence of inputs (insert, remove, search operations) given in this file. Your driver function should call the display function and the analyze function at the end as well. Include the content of your input text file and the corresponding output from the driver function in your report. You should also report the table size used.

Part 3: Compare the empirical performance values (average number of probes for successful and unsuccessful searches as given by the analyze function) and the theoretical average number of probes that can be obtained using the formulas given in the course slides (according to the collision resolution scheme selected and the load factor for the resulting hash table after executing all insert and remove operations specified in the input text file). Briefly discuss your observations.