Lab4 - Hash Table (Open Addressing)

Important! Read the document "Information and rules about the labs". Read through the entire lab specs before starting to write code so that you get an idea of what to do, the extent and can plan your work and time. Read each assignment carefully: it is important that you follow the instructions to get everything done correctly. If something is unclear, contact your lab assistant: Gabriele Capannini (`gabriele.capannini@mdh.se`) or Abu Naser Masud (`masud.abunaser@mdh.se`). You need to complete all the lab assignments to be eligible to attend the final lab examination.

Task 1: Hash Table

In this lab, you will write and test your own hash table which resolves collisions using open addressing.

Task 1.1: Download the project

Download and extract the file `Lab4.zip`. Create a new project with a proper name in your development environment and add all files in the zip file to the project. Make sure you can compile the project without any error messages (there may be warnings: just ignore them at the moment). The hash tables are represented as a dynamic array of "Buckets". The Bucket type can be found in `Bucket.h` and it can not be changed.

Task 1.2: Implement and test your hash table

The hash table you will implement is an array of Buckets. A bucket is a key-value pair. In this case, the key type will be a social security number (**int**) and the value type **Person**. The person type can be found in `Person.h`. The hash table will thus link the social security number to a person.

Your task now is to implement the functions of the hash table interface that you find in the `HashTable.h` file. Keep in mind that the array representing the hash table should be dynamically assigned to **createHashTable ()**.

In *task 1.3* below you should create multiple hash tables with different sizes and compare them with respect to the number of collisions upon inserting data. It is therefore important that the **linearProbe ()** and **insertElement ()** functions return the number of collisions according to the comments in the code skeleton. Count all the collisions if the index obtained from hashing (i.e. from the **linearProbe ()** function) is not empty and we have to search forward to find a free space. Every collision increases the risk of more collisions occurring.

In `HashTable.c`, there is also an empty hash function that you should implement.

The `test_HashTable.c` file contains a menu function that you can use to test your hash table as you implement its functions. In the menu function you can choose to manually enter personal data or randomly generate data.

Once you have implemented all functions in your hash table, run the **test ()** function that is in the `test_HashTable.c` file that tests all functionality. If the test function runs flawlessly, your hash table will work as it should.

Task 1.3: Analyze the hash table

Now that you have a functional implementation for a hash table (which solves open-ended collisions), compare the different sizes of the hash table with respect to the number of collisions.

Run the **comparHashTableSizes ()** function, and you will be prompted to specify how many tables you want to compare, and to specify the size of all of these tables. The same amount of data (50 random people) will be added to the hash tables of all selected sizes and the number of collisions counted. This is done 1000 times (with different random data volumes each time), the number of collisions is summed and an average for each table size is displayed on the screen.

If you want, you can change macros *SIZE* and *LOOPS*. *SIZE* specifies the number of people added to the hash tables and *LOOPS* indicates how many runs are made on selected sizes for the tables. You can, for example, Increase *SIZE* to compare table sizes to larger amounts of data than 50.

Compare a few different sizes and try to find one that gives as few collisions as possible without wasting too much memory.