Lab 5 - Hash Tables with Linear and Quadratic Probing

Due Date: Lab Sessions March 28 – April 8, 2022
Assessment: 5% of the total course mark.

Description:

In this lab you will implement a hash table with linear probing and perform simulations to observe its performance. The hash table will store positive integers. The hash function to be used is $h(x) = x\%M$ (the remainder after dividing x by M), where M is the size of the hash table. The set of possible keys x is the set of positive integers representable by the `int` data type. You must write the `C++` classes `HashTableLin` and `HashTableQuad` for testing and for measuring its performance.

Specifications:

- Classes `HashTableLin` and `HashTableQuad` implement hash tables where collisions are resolved with linear and quadratic probing respectively. Each class must contain a field named `table`, of type `std::vector<int>`, which stores the keys. A empty location in the table is indicated by the integer 0. There must be other fields in each class to store the size of the table, the number of keys stored in the table and the maximum load factor allowed. All fields in the classes must be `private`. Pay attention to update these fields when performing hash table operations (when necessary).
- Classes `HashTableLin` and `HashTableQuad` must contain **at least** the following constructors:
  - `HashTableLin(int maxNum, double load)`: creates a `HashTableLin` object representing an empty hash table with maximum allowed load factor equal to `load`. The amount of memory allocated for the hash table should be large enough so that `maxNum` keys could be stored without its load factor exceeding the value of parameter `load`. The size of the table should be the smallest prime number such that the above requirement is met. For instance, if `maxNum`=5 and `load`=0.4, then the size of the table should be at least 5/0.4 = 12.5. The smallest prime number satisfying the requirement is 13.
  - `HashTableQuad(int maxNum, double load)` is specified in the same manner as in `HashTableLin`.
- Classes `HashTableLin` and `HashTableQuad` must each contain at least the following `public` methods:
  1) `void insert(int n)`: inserts the key `n` in `this` hash table if the key is not already there. Collisions are resolved using linear or quadratic probing (depending on class). If the insertion will cause the load factor to exceed the maximum limit prescribed for `this` table, then rehashing should be performed by invoking the method `rehash()`. If the insertion is impossible, do nothing.
  2) `void rehash()`: private method, allocates a bigger table of size equal to the smallest prime number at least twice larger than the old size. Then all keys

from the old table are hashed into the bigger table with the appropriate hash function and with linear or quadratic probing respectively.

3) `bool isIn(int n)`: returns `true` if key `n` is in `this` hash table. It returns `false` otherwise.
4) `void printKeys()`: prints all keys stored in `this` table, in no particular order.
5) `void printKeysAndIndexes()`: prints all keys stored in `this` table and the array index where each key is stored, in increasing order of array indexes. The purpose of this method is mainly to be invoked in your test code for verifying that insertions are correct.
6) Accessor public methods (`get` methods) to allow the user to see the values of the fields relevant to the table (max load factor, number of keys, etc.)

- Additionally, classes `HashTableLin` and `HashTableQuad` must each contain the following `public static` method:
  - `std::vector<double> simProbeSuccess()`: performs simulation to measure the average number of probes for successful search for linear or quadratic probing (depending on class). Returns a `std::vector<double>` of size 9, which contains the average number of probes for successful search in a table with load factor $\lambda = 0.1, 0.2, \cdots, 0.9$ (the first element in the return vector corresponding to $\lambda = 0.1$, etc.) and size approaching ∞.

For each $\lambda \in (0, 1)$, S_λ denotes the average number of probes for successful search in a table with load factor λ and size approaching ∞. Note that the number of probes to find a key which is in the table is the same as the number of probes performed when the key was inserted. Thus the average number of probes for a successful search in a particular table can be determined by computing the average number of probes needed when the elements were inserted (by adding all numbers of probes and dividing by the number of elements in the table).

You need to perform this measurement for load factors $\lambda = 0.1, 0.2, \cdots, 0.9$, for linear or quadratic probing cases (depending on class) and return the results. For each load factor λ allocate a hash table, insert at least $100,000$ randomly generated numbers and compute the average number of probes needed. Repeat the test 100 times for a good average (i.e., have a loop with 100 iterations and then average again over all iterations). In order to perform the test ensure that the initial hash table size is large enough so that no rehashing is needed, and after all $100,000$ keys are inserted the load factor is the required λ (to do this, when creating the test object pass the number of keys, i.e., $100,000$, and λ to the constructor). Note that the random number generator may create duplicates, and duplicates should not be inserted, so make sure that you properly count the number of inserted keys.

In order to count the number of probes performed during an insertion (i.e., the number of table slots that are examined, including the one where the key is inserted), write a `private` helper method in classes `HashTableLin` and `HashTableQuad` called `int insertCount(int n)` which inserts and returns the number of probes required.

The return of the method `simProbeSuccess()` will be compared to the theoretical values of S_λ for linear or quadratic probing (depending on class). Note that the theoretical values for linear probing are given by

$$S_\lambda = \frac{1}{2}\left(1 + \frac{1}{1-\lambda}\right)$$

and the approximate theoretical values for quadratic probing are given by

$$S_\lambda = \frac{1}{\lambda}\ln\left(\frac{1}{1-\lambda}\right)$$

- You should compute the run time and space complexity of all methods (except `simProbeSuccess()`) written and be prepared to justify them to the TA.

Notes:

- You may write additional methods in your `HashTableLin` and `HashTableQuad` classes to aid in testing, if necessary. However, you are not permitted to modify method prototypes provided in each class.
- To get credit for the lab you must demonstrate your code (i.e., classes `HashTableLin` and `HashTableQuad`) in front of a TA during your lab session. A 50% penalty will be applied for either a late demo or if the electronic submission of the code is late.

Submission Instructions:

NO REPORT IS NEEDED. Submit the source code for each of the classes `HashTableLin` and `HashTableQuad` in a separate text file. Include your student number in the name of each file. For instance, if your student number is 12345 then the files should be named as follows: HashTableLin_12345.h.txt, HashTableLin_12345.cpp.txt, etc. Submit the files in the Assignments Box on Avenue by the end of your designated lab session.

Bonus:

A bonus of 0.5% of the course mark will be awarded if the `HashTableLin` class also contains the following `public static` method:

- `std::vector<double> simProbeUnsuccess()`: performs simulation to measure the average number of probes for unsuccessful search with linear probing. Returns a `std::vector<double>` of size 9, which contains the average number of probes for unsuccessful search in a table with load factor $\lambda = 0.1, 0.2, \cdots, 0.9$ (the first element in the return vector corresponding to $\lambda = 0.1$, etc.) and size approaching ∞.

Each average U_λ should be computed over at least 100,000 searches (for randomly generated numbers) in a table with at least 100,000 keys randomly generated. Make sure that you only count the **unsuccessful** searches (i.e., when the key is not in the table). You may write additional private helper methods for this purpose. The return of the method `simProbeUnsuccess()` will be compared to the theoretical values of U_λ for linear probing, given by

$$U_\lambda = \frac{1}{2}\left(1 + \frac{1}{(1-\lambda)^2}\right)$$