1 Assignment No. 5: Search Operation in Hash Tables

Open Addressing with Quadratic Probing
Allocated time: 2 hours

1.1 Implementation

You are required to implement **correctly** and **efficiently** the *insert* and *search* operations in a hash table using *open addressing* and *quadratic probing*.

You may find relevant information and pseudo-code in your course notes, or in the book ([1]), in section *11.4 Open addressing*.

The notions of closed/open specify whether you are compelled to use a certain position or a data structure.

1.1.1 Hashing (refers to the hash table)

- Open Hashing: free to leave the hash table to hold more elements at a certain index (e.g. chaining)
- Closed Hashing: not more than one element can be stored at a certain index (e.g. linear/quadratic probing)

1.1.2 Addressing (refers to the final position of the element with respect to its initial position)

- Open Addressing: the final address is not completely determined by the hash code, it also depends on the elements which are already in the hash table (e.g. linear/quadratic probing)
- Closed Addressing: the final address is always the one initially calculated (there is no probing, e.g. chaining)

For the purpose of this assignment, the hash table will not contain integers, but a custom data structure defined as follows:

```
typedef struct {
    int  id;
    char name[30];
} Entry;
```

where the *position* of each Entry in the Hash Table will be calculated by applying the required hash function on the *id* member of the struct. The *name* member of the struct will be used only to exemplify the correctness of the search and delete operations and is not needed when evaluating the performance (i.e., the *name* member will be printed to the console if the search operation finds the *id*, otherwise print "not found").

1.2 Minimal requirements for grading

The lack of any of the minimum requirements (even partially) may result in a lower grade through penalties or refusal to accept the assignment resulting in a grade of 0.

- *Demo*: Prepare a demonstration of correctness for each algorithm implemented. The correctness of each algorithm is demonstrated through a simple example (maximum 10 values).
- The charts created must be easy to evaluate as in grouped and added through the Profiler functions as specified by the assignment requirements. The assignment will not be evaluated if it contains a plethora of ungrouped charts. For example, the comparative analysis implies the grouping of the compared algorithms.
- Interpret the chart and write your observations in the header (block comments) section at the beginning of your *main.cpp* file.
- We do not accept assignments without code indentation and with code not organized in functions (for example where the entire code is in the main function).
- *The points from the requirements correspond to a correct and complete solution, quality of interpretation from the block comment and* ***the correct answer to the questions from the teacher.***

1.3 Requirements

1.3.1 Implementation of the insert and search operations using the required data structure (5p)

Demo: You will have to prove your algorithm(s) work on a small-sized input *(ex. 10)*.

1.3.2 Evaluate the search operation for a single fill factor 95% (2p)

You are required to evaluate the *search* operation for hash tables using open addressing and quadratic probing, in the **average case** (remember to perform 5 runs for this). You will do this in the following manner:

1. Select N, the size of your hash table, as a prime number around 10000 (e.g., 9973, or 10007);
2. For each of several values for the filling factor α **=0.95** do:
    a. Insert n random elements, such that you reach the required value for α ($\alpha = n/N$)

b. Search, in each case, m random elements ($m = 3000$), such that approximately half of the searched elements will be *found* in the table, and the rest will *not* be *found* (in the table). *Make sure that you sample uniformly the elements in the found category, i.e., you should search elements which have been inserted at different moments with equal probability (there are several ways in which you could ensure this – it is up to you to figure this out)*

c. Count the operations performed by the search procedure (i.e., the number of cells accessed during the search)

d. Pay attention to the values that you search for, they should be in random order of introduction. *If you look for the first 1500 values introduced in the table, implicitly the average found effort will be 1.*

3. Output a table in the following form:

Table 1: Effort measurements at various filling factors

Filling factor	**Avg. Effort *(found)***	**Max Effort *(found)***	**Avg. Effort *(not-found)***	**Max Effort *(not-found)***
0.95	...	...	...	...

Avg. Effort = total_effort / no_elements
Max. Effort = maximum number of accesses performed by one search operation

1.3.3 Complete evaluation for all fill factors (2p)

Respecting the requirements of point 2 with $\alpha \in \{0.8, 0.85, 0.9, 0.95, 0.99\}$, output a table in the following form:

Table 2: Effort measurements at various filling factors

Filling factor	**Avg. Effort *(found)***	**Max Effort *(found)***	**Avg. Effort *(not-found)***	**Max Effort *(not-found)***
0.80				
0.85				
...	...		...	

1.3.4 Implement delete operation in a hash table, *demo (size 10)* and evaluation of the search operation after deletion of some elements (1p)

Demo: You will have to prove your algorithm(s) work on a small-sized input *(ex. 10)*.

For the evaluation of the search operation after deletion, fill the hash table until a fill factor of 0.99. Delete elements from the table until you get a filling factor of 0.8 and afterwards search m random elements (m ~ 3000) such that approximately half of the searched elements will be *found* in the table, and the rest will *not be found* (in the table). Count the operations performed by the *search* and add it in the previous table.

References

[1] Thomas H. Cormen et al. *Introduction to Algorithms*. 2nd. The MIT Press, 2001. ISBN: 0262032937.