Assignment No. 5: Search Operation in Hash Tables
Open Addressing with Quadratic Probing

Allocated time: 2 hours

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, in section *11.4 Open addressing*.

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

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)

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;
```

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 operation, 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”).

Evaluation

! Before you start to work on the algorithms evaluation code, make sure you have a correct algorithm! You will have to prove your algorithm(s) work on a small-sized input.

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 $\alpha \in \{0.8, 0.85, 0.9, 0.95, 0.99\}$, 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 ($m \sim 3000$), 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)

3. Output a table of the form:

Filling factor	Avg. Effort *found*	Max. Effort *found*	Avg. Effort *not-found*	Max. Effort *not-found*
0.8				
0.85				
...	...		...	

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

4. Interpret your results.

Thresholds

Grade	**Requirements**
5	Demo for the insert and search operations using the required data structure.
7	Evaluate the search operation for a single fill factor, with non-uniform selection of the elements
9	Complete evaluation for all fill factors with uniform selection of the found elements.
10	Evaluation and personal interpretations of the results