Objective of this lab:

Hashing

Instructions:

• Make a separate project for each task.

• Indent your code properly.

• Use meaningful variable and function names. Follow the naming conventions. • Use meaningful prompt lines and labels for all input/output.

• Make sure that there are NO dangling pointers or memory leaks in your program.

Note: "YOUR PROGRAM MUST HAVE A MAIN FUNCTION THAT TEST ALL THE IMPLEMENTED FUNCTIONS"

In this lab, you are going to implement a class for Hash Table. The class definitions will look like:

```
class HashTable;
class HashItem
{
        friend class HashTable;
        private:
                int key; // index where your value will be stored
                string value; //actual value to be stored in HashTable
                int status; // 0: Empty, 1: Deleted, 2: Occupied

};
class HashTable
{
private:
        HashItem* hashArray;
        int capacity;
        int currentElements;
        int getNextCandidateIndex(int key, int i);
        void doubleCapacity()
public:
        HashTable();
        HashTable(int const capacity);
        void insert(int const key, string const value);
        bool deleteKey(int const key) const;
        string get(int const key) const;
        ~HashTable();
};
```

About HashItem class:

It represents an item (key-value pair) to be inserted in a hash table. The **status** variable can have 0, 1 or 2. 0 means empty, 1 means deleted, 2 means occupied. Status variable will be used by get and delete methods of HashTable implemented in the next questions. The default value assigned to a HashItem is 0 (empty).

About HashTable class:

As, all of the private functions' prototypes are already given for your reference. You need to implement this class in this lab. All questions (which you see below) divides this task in parts.

Question1:

Implement the two constructors (default and overloaded), and destructor of the class HashTable. 1. **HashTable():** constructor assigns a capacity of 10 to hashArray.

2. **HashTable(int const capacity):** an overloaded constructor that assigns the capacity of given capacity to hashArray. If capacity is less than 1 return error message

3. **~HashTable():** destructor

Question2:

Now, implement these functions which are given below, and are required for helping in handling collision, and also in insert and delete methods.

int getNextCandidateIndex(int key, int i): a private method that uses linear probing to return the next candidate index for storing the item containing key k. Linear probing means that it will simply add i to the hash value of key. This method does not check whether the candidate index has collision or not.

void doubleCapacity(): A private method which doubles the capacity of hash array and **rehashes** the existing items (Remember we have a dynamic hash array). Use getNextCandidateIndex method to resolve collision.

Question3:

Implement the public **insert** function

void insert(int const key, v const value):

a. The insert method inserts the value at its appropriate location. Find the first candidate index of the key using:
   *index= key **mod** capacity*
b. To resolve hash collision, it will use the function **getNextCandidateIndex**(key, i) to get the next candidate index. If the candidate index also has collision, then

getNextCandidateIndex will be called again with an **increment** in i. getNextCandidateIndex will be continued to call until we find a valid index. Initially i will be **1**.

c. If the loadFactor becomes **0.75**, then it will call the **doubleCapacity** method to double the capacity of array and rehash the existing items into the new array.

Hint: To check load factor, the overall formula used will be **currentElements >= 0.75 * capacity**

Question 4:

Implement the following functions for **deletion** of a given value from the hash table and **searching** the value in hash table:

bool deleteKey(int const key) const;

This method deletes the given key. It returns **true** if the key was found. If the key was not found it returns **false**. When the key is found, simply set the **status** of the hashItem containing the key to deleted (value of **1**). It also uses status variable to **search** for the key intelligently ��

string get(int const key) const

This method returns the value of the key. If the key is not found, it returns a message saying “not found”. It also uses **status** variable to search for the key intelligently.

Question 5:

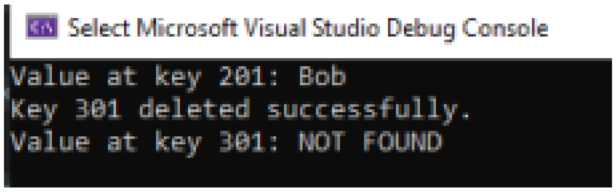
Rewrite the class in such a way that it performs **Quadratic Probing**, i.e., add the square of i to the hash value of key.

Hint: You have to change getNextCandidateIndex(int key, int i) method

Test all your functions through this main:

```
int main() {
 HashTable hashtable(5); // Create a hash map with capacity 5 (for simplicity, using
string as the value type)

 // Insert key-value pairs into the hash map
 hashtable.insert(101, "Alice");
 hashtable.insert(201, "Bob");
 hashtable.insert(301, "Charlie");
 hashtable.insert(401, "David");
 hashtable.insert(501, "Eva");

 // Test get function
 string name = hashtable.get(201);
 if (name.length()!=0) {
 cout << "Value at key 201: " << name << endl;
 }
 else {
 cout << name << endl;
 }

 // Test deleteKey function
 bool deleted = hashtable.deleteKey(301);
 if (deleted) {
 cout << "Key 301 deleted successfully." << endl;
 }
 else {
 cout << "Key 301 not found!" << endl;
 }

 // Test get after deletion
 name = hashtable.get(301);
 if (name.length()!=0) {
 cout << "Value at key 301: " << name << endl; // Should not be found after
deletion
 }
 else {
 cout << name << endl;
 }

 return 0;
}
```

Output should be like this:



```
Value at key 201: Bob
Key 301 deleted successfully.
Value at key 301: NOT FOUND
```

GOOD LUCK