FSM FATİH SULTAN MEHMET VAKIF ÜNİVERSİTESİ	Mühendislik Fakültesi Bilgisayar Mühendisliği Programı **BLM307E Algorithm Analysis & Design Lab Work**	Adı Soyadı: Notu:

Lab 8: Hash functions and Hash Table

Question 1: Implement the double hash function. You should implement only **put** function in the HashTable.java file.

```
DOUBLE_HASH (key, value)
1.  j = 0
2.  while j < N:
3.      h_k = hashFunction_1(key) % 13
4.      d_k = hashFunction_2(key) % 13
5.      probe = h_k+ j * d_k
6.      item = table[probe]
7.      if item == ∅ then
8.          insert new Entry < key, value>
9.      else if item.key == key then
10.         table[idx].value = value
11.
12.     j++;
```

```
// Double Hashing
    public void put(int key, String value) {
        // YOUR CODE HERE
        int j = 0;
        int N = 13;
        while (j < N) {
            int h_k = hashFunction_1(key) % 13;
            int d_k = hashFunction_2(key) % 13;
            int probe = h_k + j * d_k;
//          HashEntry <Integer,String> item = table[probe];
            if (item == null) {
//              table[key] = new HashEntry<Integer, String>;
            } else if(item.key == key){
//              table[idx].value = value;
            }
             j++;
        }

    }
```

Question 2: There are many functions used for collision handling in hashing. Explain one of them using the example below (Except the double hashing). Show all steps (**N = 13**).

Input: [18, 41, 22, 44, 59, 32, 31, 73]

```
H(k, i) = (H(k) + i^2)%N
```

We can use quadratic hashing for handling collision quadratic hashing:

Steps: 1. set counter j=0

2. get the hash value for the key k as, h(k)=(k+i^2) mod tablesize

3. If the slot h(k) % N is full, then we try (h(k) + 1*1) % N

4. If (hash(x) + 1*1) % N is also full, then we try (h(k) + 2*2) % N.

	Mühendislik Fakültesi Bilgisayar Mühendisliği Programı **BLM307E Algorithm Analysis & Design Lab Work**	Adı Soyadı: Notu:

5. If (hash(x) + 2*2) % N is also full, then we try (h(k) + 3*3) % N.
   6. This process is repeated for all the values of i until an empty slot is found.

Question 3: What is a good Hash function? Which factors affect the performance of the hash function? Explain each factor.

A good hash function should have the following properties:
Efficiently computable.
Should uniformly distribute the keys (Each table position equally likely for each key)
For example: For phone numbers, a bad hash function is to take the first three digits. A better function is considered the last three digits. Please note that this may not be the best hash function. There may be better ways. In practice, we can often employ heuristic techniques to create a hash function that performs well. Qualitative information about the distribution of the keys may be useful in this design process. In general, a hash function should depend on every single bit of the key, so that two keys that differ in only one bit or one group of bits (regardless of whether the group is at the beginning, end, or middle of the key or present throughout the key) hash into different values. Thus, a hash function that simply extracts a portion of a key is not suitable. Similarly, if two keys are simply digited or character permutations of each other (such as 139 and 319), they should also hash into different values.

Question 4: Which implementation is better? HashTable LinkedList implementation or Array implementation? Evaluate implementation for search, remove and insert operations.

	search	**insert**	**remove**
Array	O(n)	O(1)	O(n)
LinkedList	*O(n)*	O(n)	O(n)

Question 5: Fill in the following table with the worst-case efficiency classes using Θ notation.

	search	**insert**	**remove**
AVL Tree	Θ(logn)	Θ(logn)	Θ(logn)
Hash Table	Θ(1)	Θ(1)	Θ(1)
Binary Search Tree	Θ(n)	Θ(n)	Θ(n)