CS3334 Data Structures (2022 Sem A)

Assignment: Hash Table

Name: xxx

SID: xxx

Overview

I have implemented a hash table that follows all the operations mentioned in the requirements document. Namely, for a hash table **T** it is possible to perform following operations:

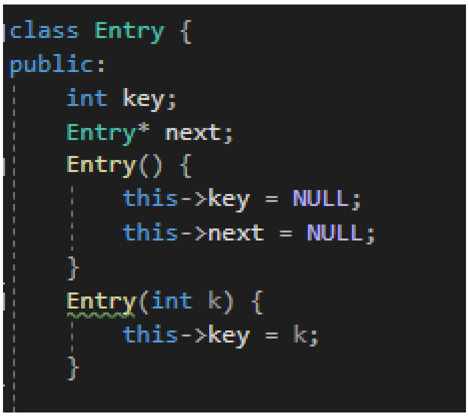
- InsertKey(x): inserts the key/entry **x** to **T**;
- DeleteKey(x): removes the key/entry **x** from **T**;
- SearchKey(x): searches for the key/entry **x** in **T**: if it is found, returns “x was found”, otherwise, it returns “x does not exist”.

Details

Having stated the general functionality of a hash table, let’s talk about the details of its implementation.

Entries

Overall, my version of hash table is basically an **Array** that contains pointers to Entries. In order to get a bigger picture of the hash table, let’s first analyze the implementation of Entry:



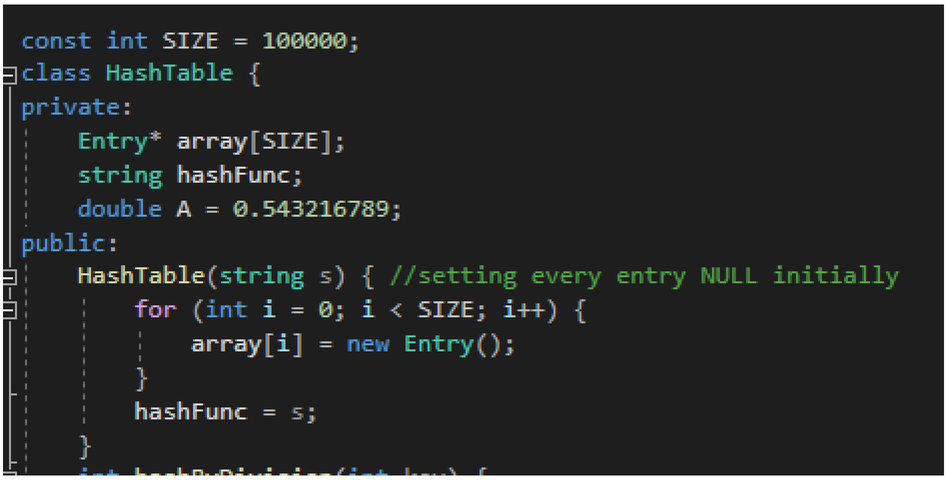
```
class Entry {
public:
    int key;
    Entry* next;
    Entry() {
        this->key = NULL;
        this->next = NULL;
    }
    Entry(int k) {
        this->key = k;
    }
```

The Entry class is implemented in a similar way as a **node** for **linked list**: it has a key and the pointer to a next Entry object. Its default constructor sets both its key and next NULL. Second constructor which asks for (int **k**) assigns **this->key**.

The reason I have made Entry class this way is because I have decided to handle collisions with separate chaining using **linked lists**.

HashTable Array

The hash table itself is an array of pointers to Entries as mentioned above:

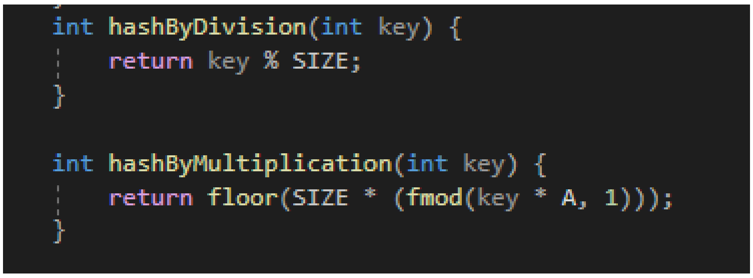


```
const int SIZE = 100000;
class HashTable {
private:
    Entry* array[SIZE];
    string hashFunc;
    double A = 0.543216789;
public:
    HashTable(string s) { //setting every entry NULL initially
        for (int i = 0; i < SIZE; i++) {
            array[i] = new Entry();
        }
        hashFunc = s;
    }
    int hashByDivision(int key) {
```

It has a fixed size of 100000 an. Its constructor takes a **string** and assigns it as a name to a hashing method **hasFunc**. So it is more convenient to choose which hashing method to try while initializing a hash table.

Hashing functions

The hashing in my hash table is implemented using **Division method** and **Multiplication method**.

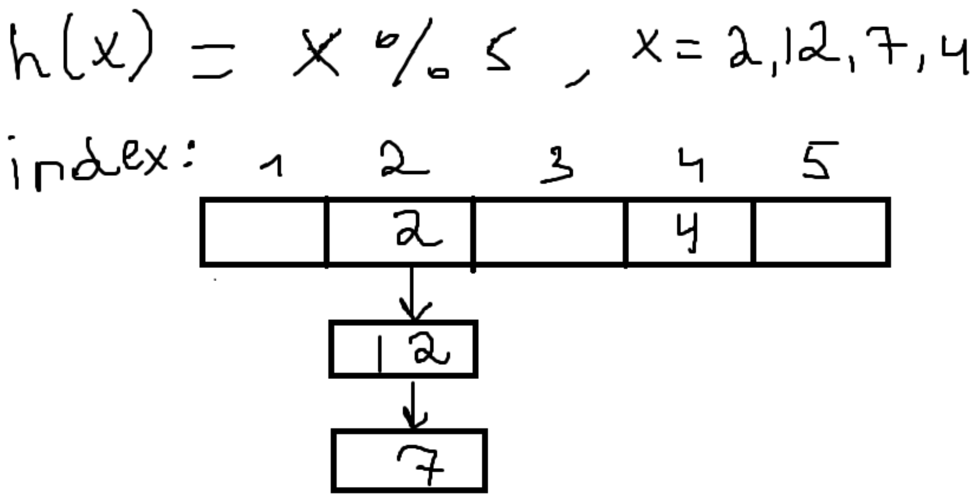


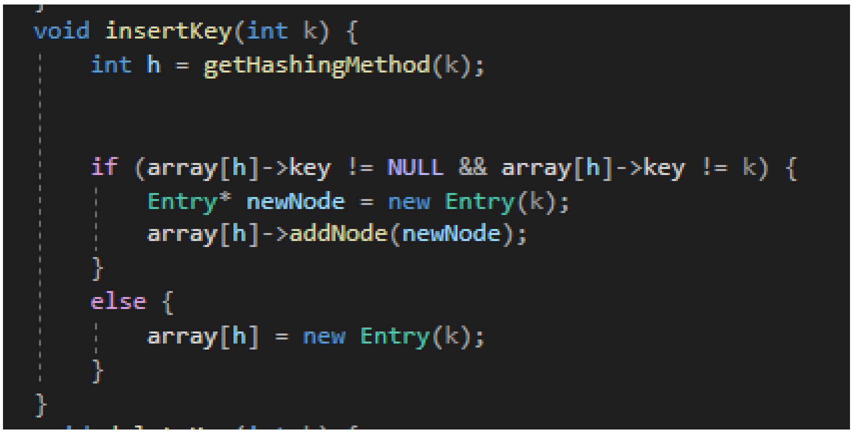
```
int hashByDivision(int key) {
    return key % SIZE;
}

int hashByMultiplication(int key) {
    return floor(SIZE * (fmod(key * A, 1)));
}
```

Note: for a Multiplication method, I have used a **floor()** function from **<cmath>**.

Insert/Delete/Search operations and Collision handling

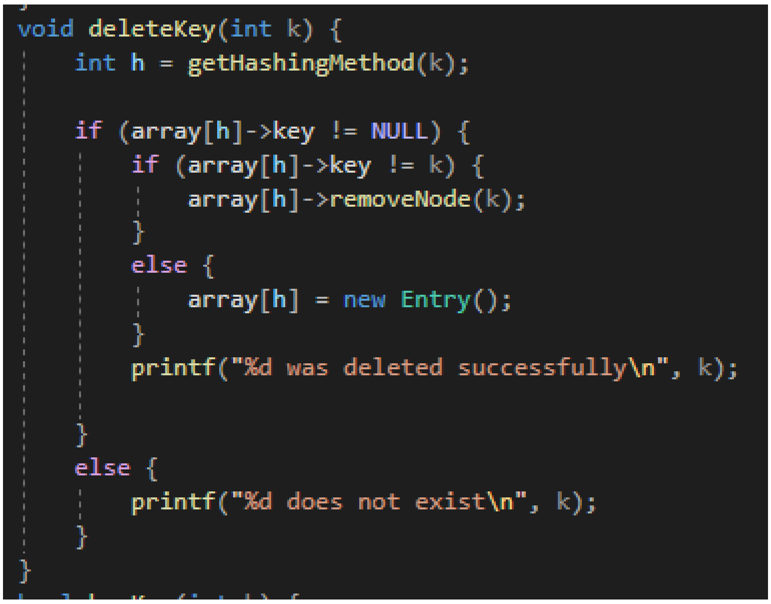
As mentioned above, I have used **linked lists** to avoid collisions, meaning for instance for a hash table of **size 5** with the following hash function, if **x** is inserted as **2** and then as **12**, then it would have a collision. This is because **2%5 = 12%5**. So, the linked list would solve our problem: every time we see a collision, we append a node to an Entry as shown below:





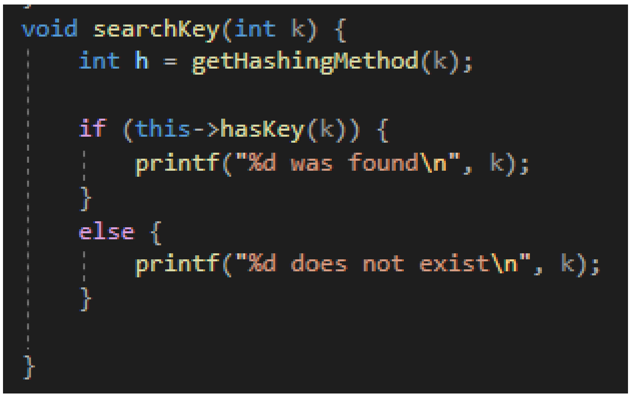
```
void insertKey(int k) {
    int h = getHashingMethod(k);


    if (array[h]->key != NULL && array[h]->key != k) {
        Entry* newNode = new Entry(k);
        array[h]->addNode(newNode);
    }
    else {
        array[h] = new Entry(k);
    }
}
```

Insert



```
void deleteKey(int k) {
    int h = getHashingMethod(k);

    if (array[h]->key != NULL) {
        if (array[h]->key != k) {
            array[h]->removeNode(k);
        }
        else {
            array[h] = new Entry();
        }
        printf("%d was deleted successfully\n", k);

    }
    else {
        printf("%d does not exist\n", k);
    }
}
```

Delete



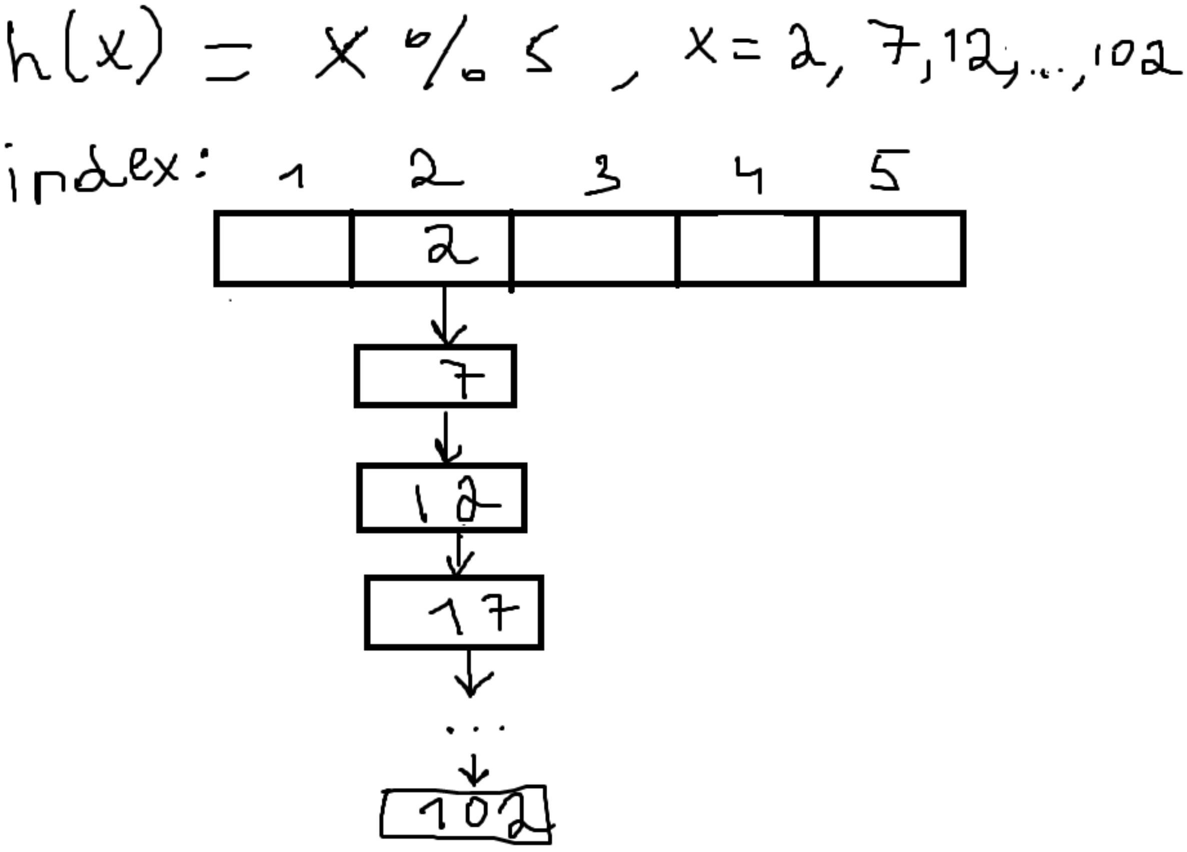
```
void searchKey(int k) {
    int h = getHashingMethod(k);

    if (this->hasKey(k)) {
        printf("%d was found\n", k);
    }
    else {
        printf("%d does not exist\n", k);
    }

}
```

Search

Time complexity analysis

The **average time complexity** for all those 3 operations is **O(1) – constant time**. This is because on average, we assume that no collision happens. Hence, looking up a key in an array takes constant time.

The **worst case scenario** for my hash table is when every insertion causes collision: meaning for instance for the previous example with **h(x)=x%5**, if the keys were **2,7,12,17,22,…,102**, the hash table might look like this:



This would mean that every insertion/deletion/search operation would take **O(n)** where **n** is the **number of keys/entries linked** together. This is because in order to search for a key, it should traverse **one-by-one** until it reaches the point. If we assume that the searched key is the **tail** of the linked list, it takes traversal of every key as well.

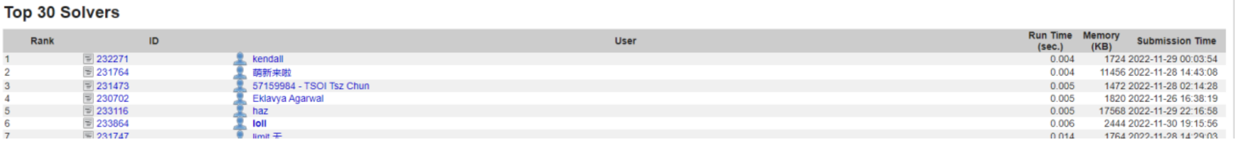
On the other hand, it is also worth noting that this method of separate chaining does not depend on the size of the array. Naturally, the type of hashing does not fully prevent collision and with linked list approach like this, we can see that it does not affect its performance. However, it should also be pointed that multiplication hashing method is more suitable for large size of array.

Testing

Here are short instructions on how to execute cpp files. Once you have downloaded the zip file, unpack it and the change the cmd directory to it using 'cd file_path' command. Then, if you type 'g++ .\sampleGenerator.cpp -o exec_file' or 'g++ .\Source.cpp -o exec_file', you will create an exec file ready for execution. The 'Source.cpp' file was for OJ system, while 'sampleGenerator' was given from the canvas, though, I have made some changes to its code (see SampleGenerator.cpp section below).

CityU CS ACM Programming Team Online Judge System

First, after the implementation of the hash table, I checked it in the OJ system. Both **Division** and **Multiplication** methods passed the testcases. The runtime for both were fairly the same. However, when I changed the hash table array size from 100000 to 10000, the difference between the two methods rocketed. In fact, the **Division** method implementation was the **6th** fastest in the OJ records:



Top 30 Solvers

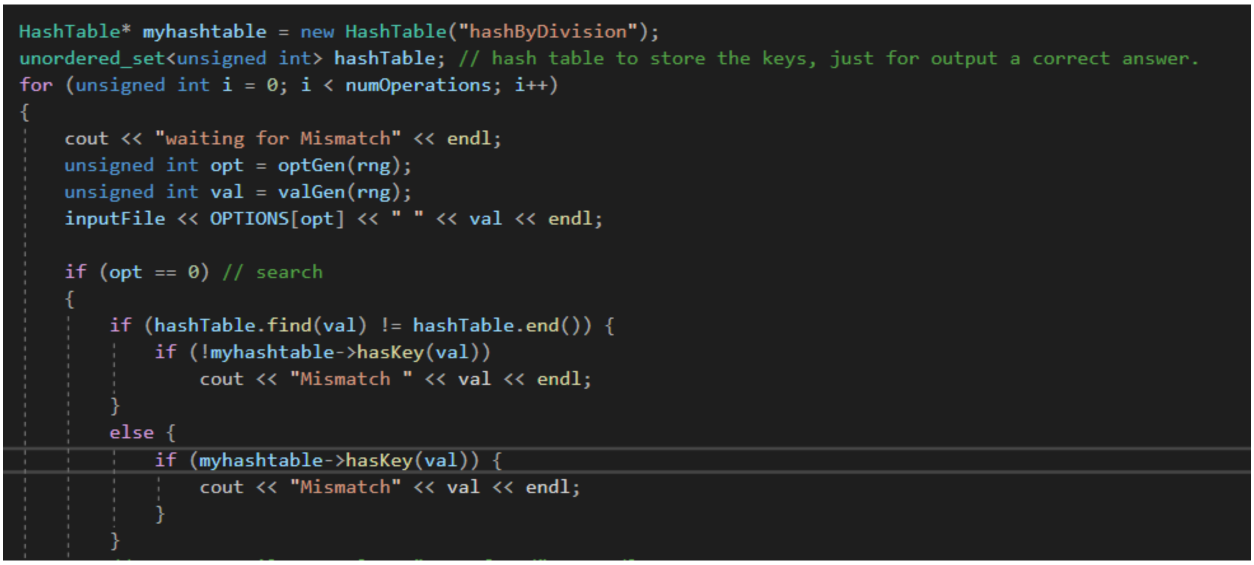
Rank	ID	User	Run Time (sec.)	Memory (KB)	Submission Time
1	232271	kendall	0.004	1724	2022-11-29 00:03:54
2	231764	萌新来啦	0.004	11456	2022-11-28 14:43:08
3	231473	57159984 - TSOI Tsz Chun	0.005	1472	2022-11-28 02:14:28
4	230702	Eklavya Agarwal	0.005	1820	2022-11-26 16:38:19
5	233116	haz	0.005	17568	2022-11-29 22:16:58
6	233864	**loll**	0.006	2444	2022-11-30 19:15:56
7	231747	limit 王	0.014	1764	2022-11-28 14:29:03

This also supports my statement that for less number of array size, **the Division** method is more suitable.

I have test my code with the C++ 11, flag: -static -std=c++0x compiler in the OJ.

SampleGenerator.cpp

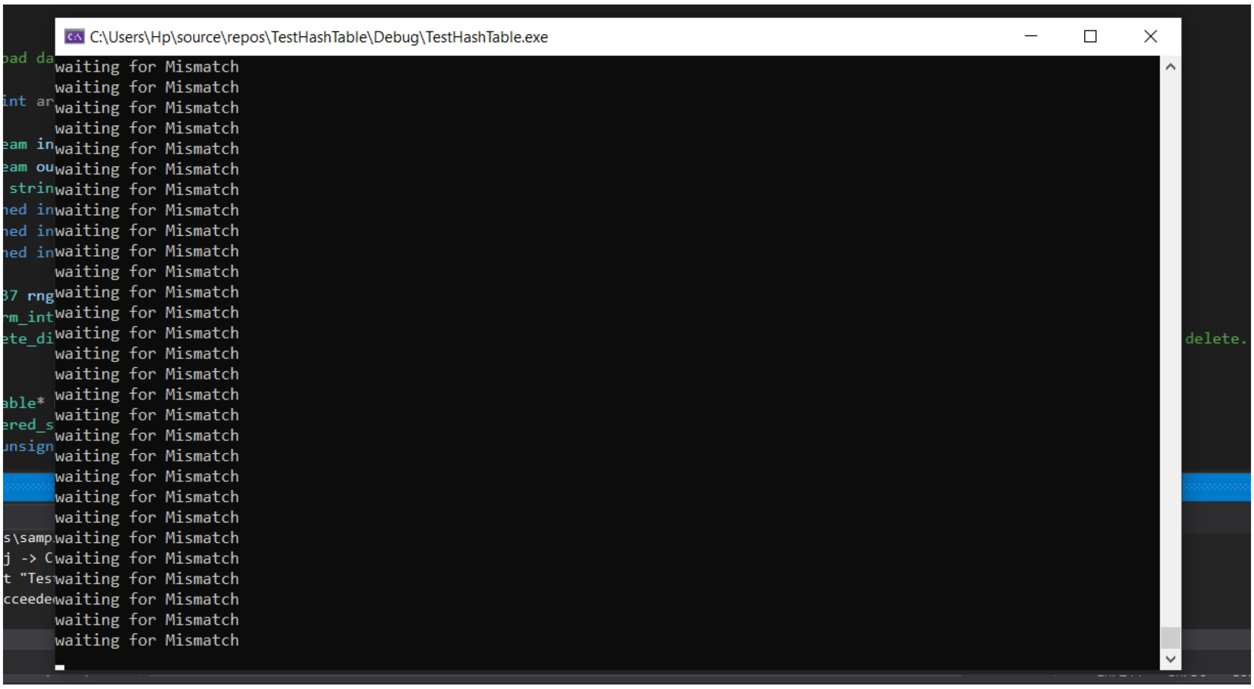
Additionally, in order check the correctness of my implementation, I have used the **sampleGenerator.cpp** source code, which uses **<unordered_set>**. I have made little changes to a code, so that whenever the unordered set searches for a key and decides that it **exists** and my hash table, on the contrary, decides that it **does not**, I printed out those records (Also, vice-versa, when the unordered says it **does not exist**, while my hash table says it **does**).



```
HashTable* myhashtable = new HashTable("hashByDivision");
unordered_set<unsigned int> hashTable; // hash table to store the keys, just for output a correct answer.
for (unsigned int i = 0; i < numOperations; i++)
{
    cout << "waiting for Mismatch" << endl;
    unsigned int opt = optGen(rng);
    unsigned int val = valGen(rng);
    inputFile << OPTIONS[opt] << " " << val << endl;

    if (opt == 0) // search
    {
        if (hashTable.find(val) != hashTable.end()) {
            if (!myhashtable->hasKey(val))
                cout << "Mismatch " << val << endl;
        }
        else {
            if (myhashtable->hasKey(val)) {
                cout << "Mismatch" << val << endl;
            }
        }
```

Here, for each iteration, I print out “waiting for Mismatch” to see if a program is running correctly. When the mismatch happens, it should print out “Mismatch”.

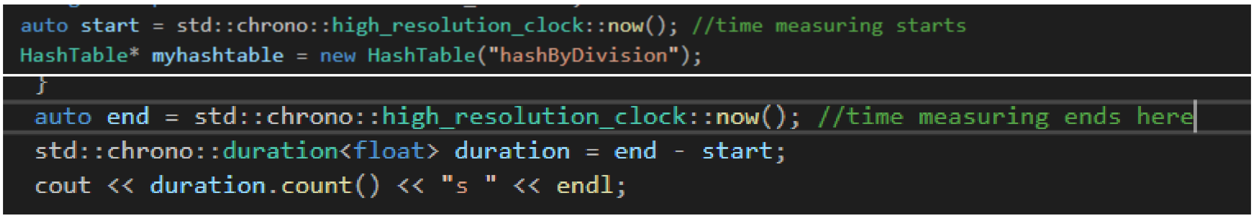
And here is the output:


C:\Users\Hp\source\repos\TestHashTable\Debug\TestHashTable.exe

```
waiting for Mismatch
waiting for Mismatch
waiting for Mismatch
waiting for Mismatch
waiting for Mismatch
waiting for Mismatch
waiting for Mismatch
waiting for Mismatch
waiting for Mismatch
waiting for Mismatch
waiting for Mismatch
waiting for Mismatch
waiting for Mismatch
waiting for Mismatch
waiting for Mismatch
waiting for Mismatch
waiting for Mismatch
waiting for Mismatch
waiting for Mismatch
waiting for Mismatch
waiting for Mismatch
waiting for Mismatch
waiting for Mismatch
waiting for Mismatch
waiting for Mismatch
waiting for Mismatch
waiting for Mismatch
waiting for Mismatch
waiting for Mismatch
```

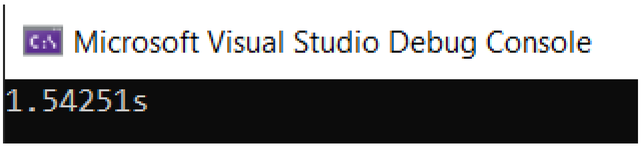
It goes on like this for quite a bit and never prints out the message that Mismatch happened, which means that every single record matches and my hash table works correctly.

Furthermore, I have used **<chrono>** to measure the running time to compare the performance of two hashing functions:



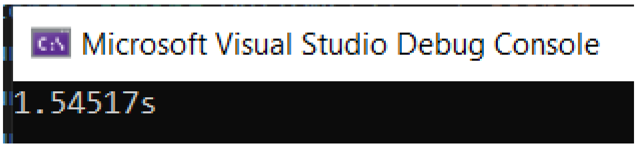
```
auto start = std::chrono::high_resolution_clock::now(); //time measuring starts
HashTable* myhashtable = new HashTable("hashByDivision");
}
auto end = std::chrono::high_resolution_clock::now(); //time measuring ends here
std::chrono::duration<float> duration = end - start;
cout << duration.count() << "s " << endl;
```

When I set size of an array to **100000**, the results were like this:



```
1.54251s
```

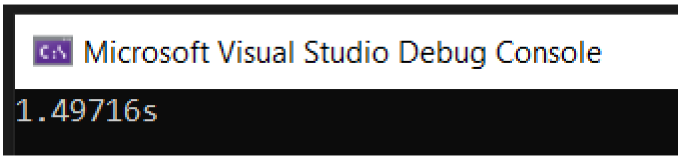
for Division method (SIZE=10^5)



```
1.54517s
```

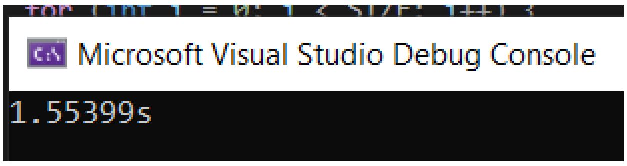
for Multiplication method (SIZE=10^5).

Here, we can see that both have the same performance time. However, if we set size of an array to **10000** now



```
1.49716s
```

for Division method (SIZE=10^4)



```
1.55399s
```

for Multiplication method (SIZE=10^4).

Now we can conclude that on practicality, the Division methods perfoms much faster than the Multiplication if the size of an array is less, which further confirms my statement mentioned above.