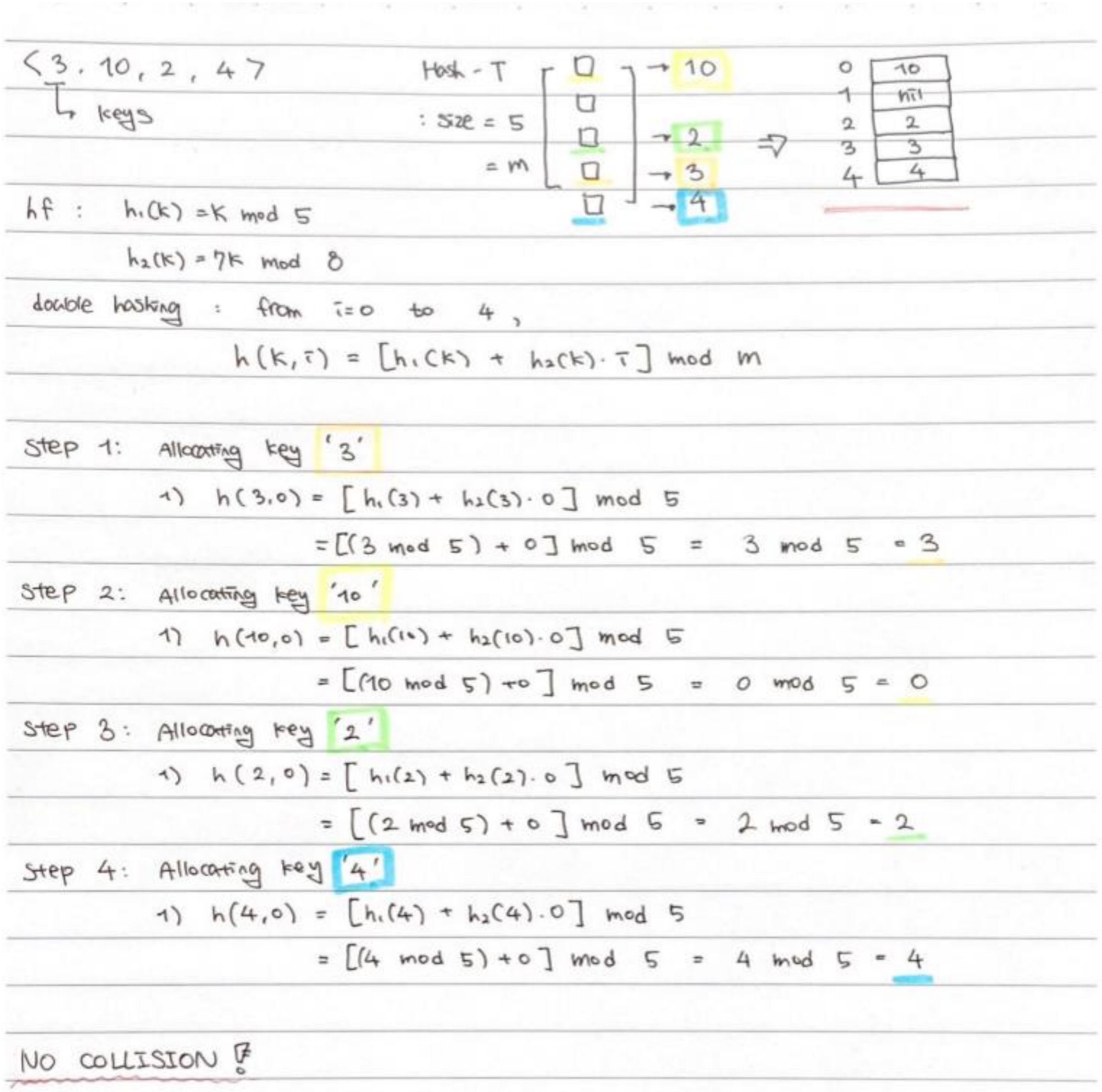
Assignment 10

Name: DONGWOOK LEE

Problem 1: *Hash Tables*

a) (4 points)



<3, 10, 2, 4>
↳ keys

Hash - T : size = 5 = m

$\begin{bmatrix} \square \\ \square \\ \square \\ \square \\ \square \end{bmatrix}$ → 10, → 2, → 3, → 4 ⇒

0	10
1	nil
2	2
3	3
4	4

hf : $h_1(k) = k \bmod 5$

$h_2(k) = 7k \bmod 8$

double hashing : from $i = 0$ to 4,

$$h(k, i) = [h_1(k) + h_2(k) \cdot i] \bmod m$$

Step 1: Allocating key '3'

1) $h(3,0) = [h_1(3) + h_2(3) \cdot 0] \bmod 5$

$= [(3 \bmod 5) + 0] \bmod 5 = 3 \bmod 5 = 3$

Step 2: Allocating key '10'

1) $h(10,0) = [h_1(10) + h_2(10) \cdot 0] \bmod 5$

$= [(10 \bmod 5) + 0] \bmod 5 = 0 \bmod 5 = 0$

Step 3: Allocating key '2'

1) $h(2,0) = [h_1(2) + h_2(2) \cdot 0] \bmod 5$

$= [(2 \bmod 5) + 0] \bmod 5 = 2 \bmod 5 = 2$

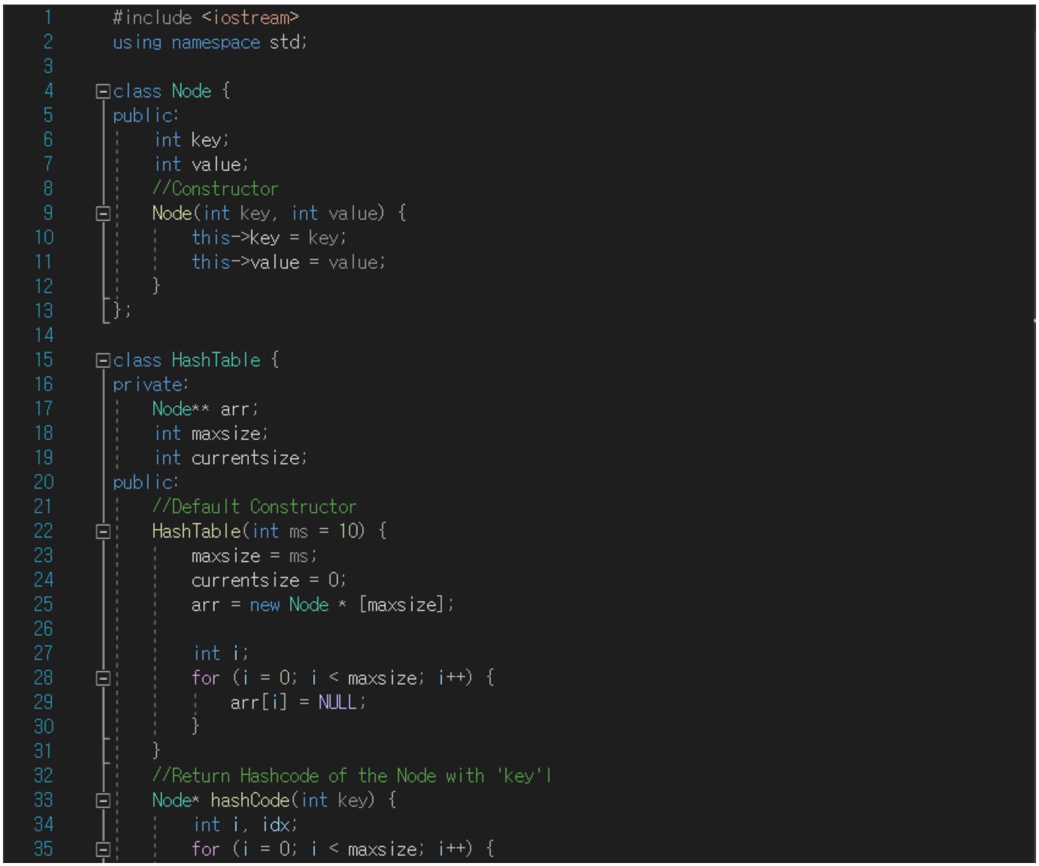
Step 4: Allocating key '4'

1) $h(4,0) = [h_1(4) + h_2(4) \cdot 0] \bmod 5$

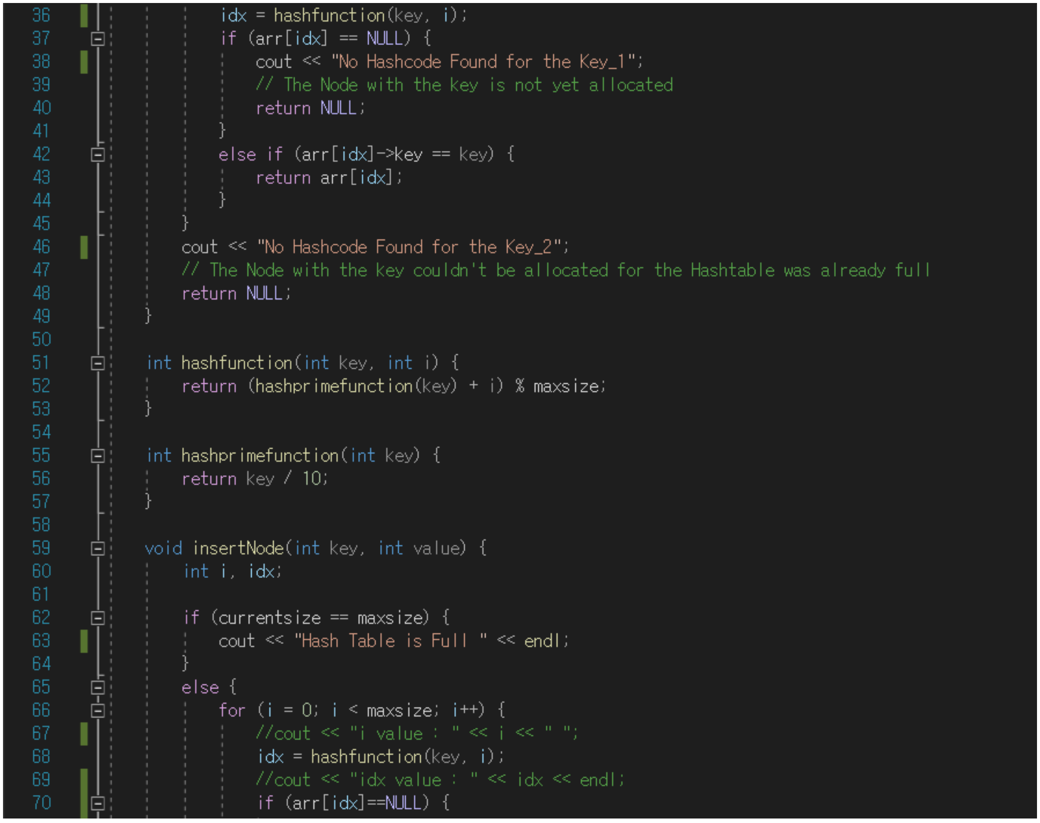
$= [(4 \bmod 5) + 0] \bmod 5 = 4 \bmod 5 = 4$

NO COLLISION !

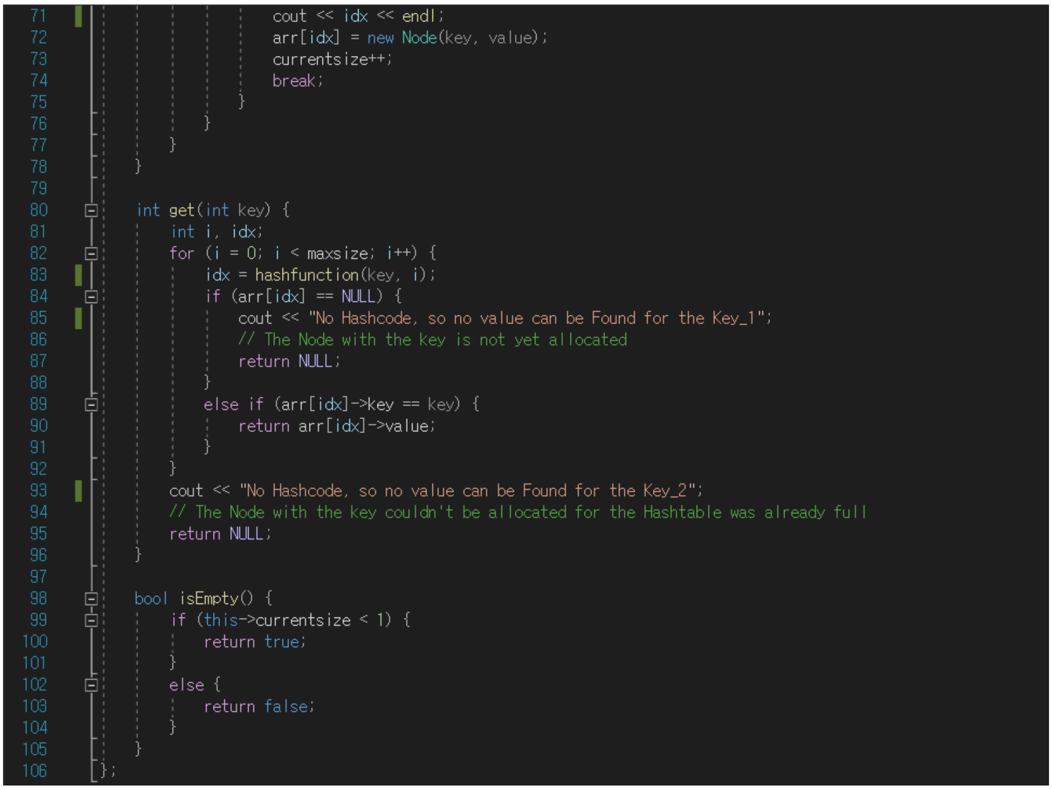
b) (7 points)



```
#include <iostream>
using namespace std;

class Node {
public:
    int key;
    int value;
    //Constructor
    Node(int key, int value) {
        this->key = key;
        this->value = value;
    }
};

class HashTable {
private:
    Node** arr;
    int maxsize;
    int currentsize;
public:
    //Default Constructor
    HashTable(int ms = 10) {
        maxsize = ms;
        currentsize = 0;
        arr = new Node * [maxsize];

        int i;
        for (i = 0; i < maxsize; i++) {
            arr[i] = NULL;
        }
    }
    //Return Hashcode of the Node with 'key'
    Node* hashCode(int key) {
        int i, idx;
        for (i = 0; i < maxsize; i++) {
```

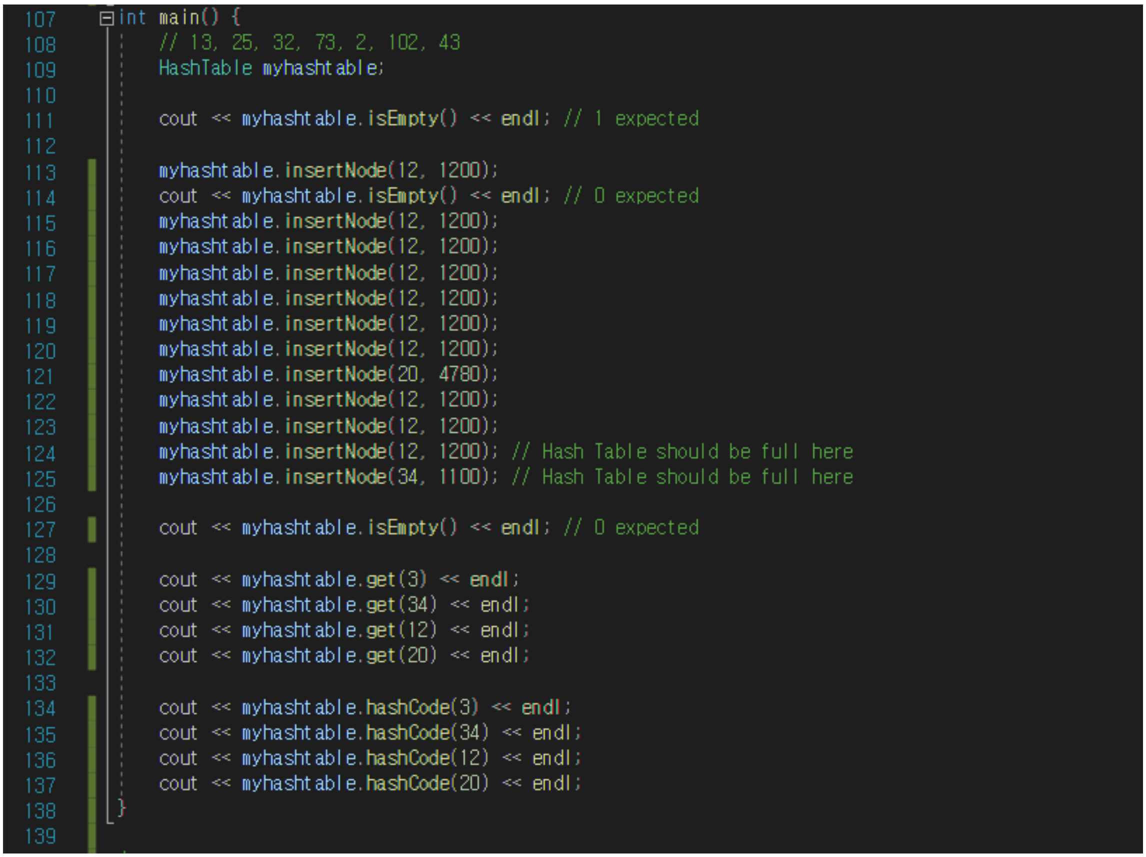


```
            idx = hashfunction(key, i);
            if (arr[idx] == NULL) {
                cout << "No Hashcode Found for the Key_1";
                // The Node with the key is not yet allocated
                return NULL;
            }
            else if (arr[idx]->key == key) {
                return arr[idx];
            }
        }
        cout << "No Hashcode Found for the Key_2";
        // The Node with the key couldn't be allocated for the Hashtable was already full
        return NULL;
    }

    int hashfunction(int key, int i) {
        return (hashprimefunction(key) + i) % maxsize;
    }

    int hashprimefunction(int key) {
        return key / 10;
    }

    void insertNode(int key, int value) {
        int i, idx;

        if (currentsize == maxsize) {
            cout << "Hash Table is Full " << endl;
        }
        else {
            for (i = 0; i < maxsize; i++) {
                //cout << "i value : " << i << " ";
                idx = hashfunction(key, i);
                //cout << "idx value : " << idx << endl;
                if (arr[idx]==NULL) {
```

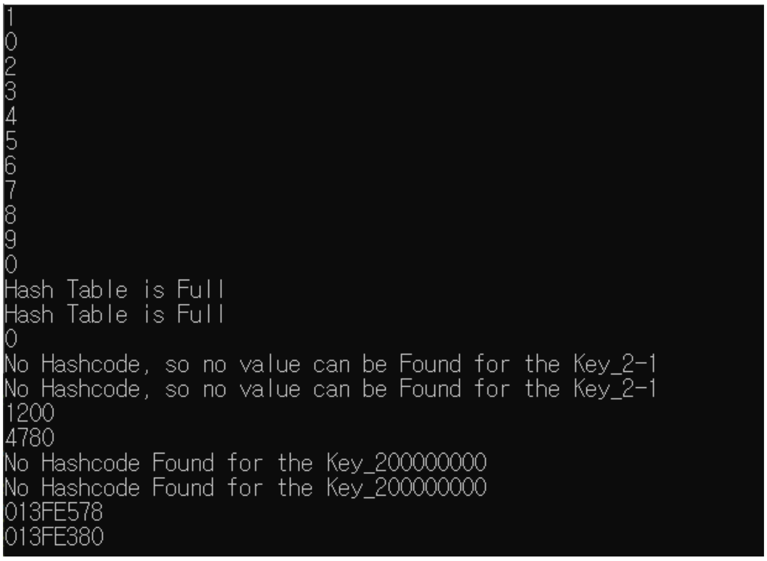


```
                    cout << idx << endl;
                    arr[idx] = new Node(key, value);
                    currentsize++;
                    break;
                }
            }
        }
    }

    int get(int key) {
        int i, idx;
        for (i = 0; i < maxsize; i++) {
            idx = hashfunction(key, i);
            if (arr[idx] == NULL) {
                cout << "No Hashcode, so no value can be Found for the Key_1";
                // The Node with the key is not yet allocated
                return NULL;
            }
            else if (arr[idx]->key == key) {
                return arr[idx]->value;
            }
        }
        cout << "No Hashcode, so no value can be Found for the Key_2";
        // The Node with the key couldn't be allocated for the Hashtable was already full
        return NULL;
    }

    bool isEmpty() {
        if (this->currentsize < 1) {
            return true;
        }
        else {
            return false;
        }
    }
};
```

EX 1)



```
int main() {
    // 13, 25, 32, 73, 2, 102, 43
    HashTable myhashtable;

    cout << myhashtable.isEmpty() << endl; // 1 expected

    myhashtable.insertNode(12, 1200);
    cout << myhashtable.isEmpty() << endl; // 0 expected
    myhashtable.insertNode(12, 1200);
    myhashtable.insertNode(12, 1200);
    myhashtable.insertNode(12, 1200);
    myhashtable.insertNode(12, 1200);
    myhashtable.insertNode(12, 1200);
    myhashtable.insertNode(12, 1200);
    myhashtable.insertNode(20, 4780);
    myhashtable.insertNode(12, 1200);
    myhashtable.insertNode(12, 1200);
    myhashtable.insertNode(12, 1200); // Hash Table should be full here
    myhashtable.insertNode(34, 1100); // Hash Table should be full here

    cout << myhashtable.isEmpty() << endl; // 0 expected

    cout << myhashtable.get(3) << endl;
    cout << myhashtable.get(34) << endl;
    cout << myhashtable.get(12) << endl;
    cout << myhashtable.get(20) << endl;

    cout << myhashtable.hashCode(3) << endl;
    cout << myhashtable.hashCode(34) << endl;
    cout << myhashtable.hashCode(12) << endl;
    cout << myhashtable.hashCode(20) << endl;
}
```



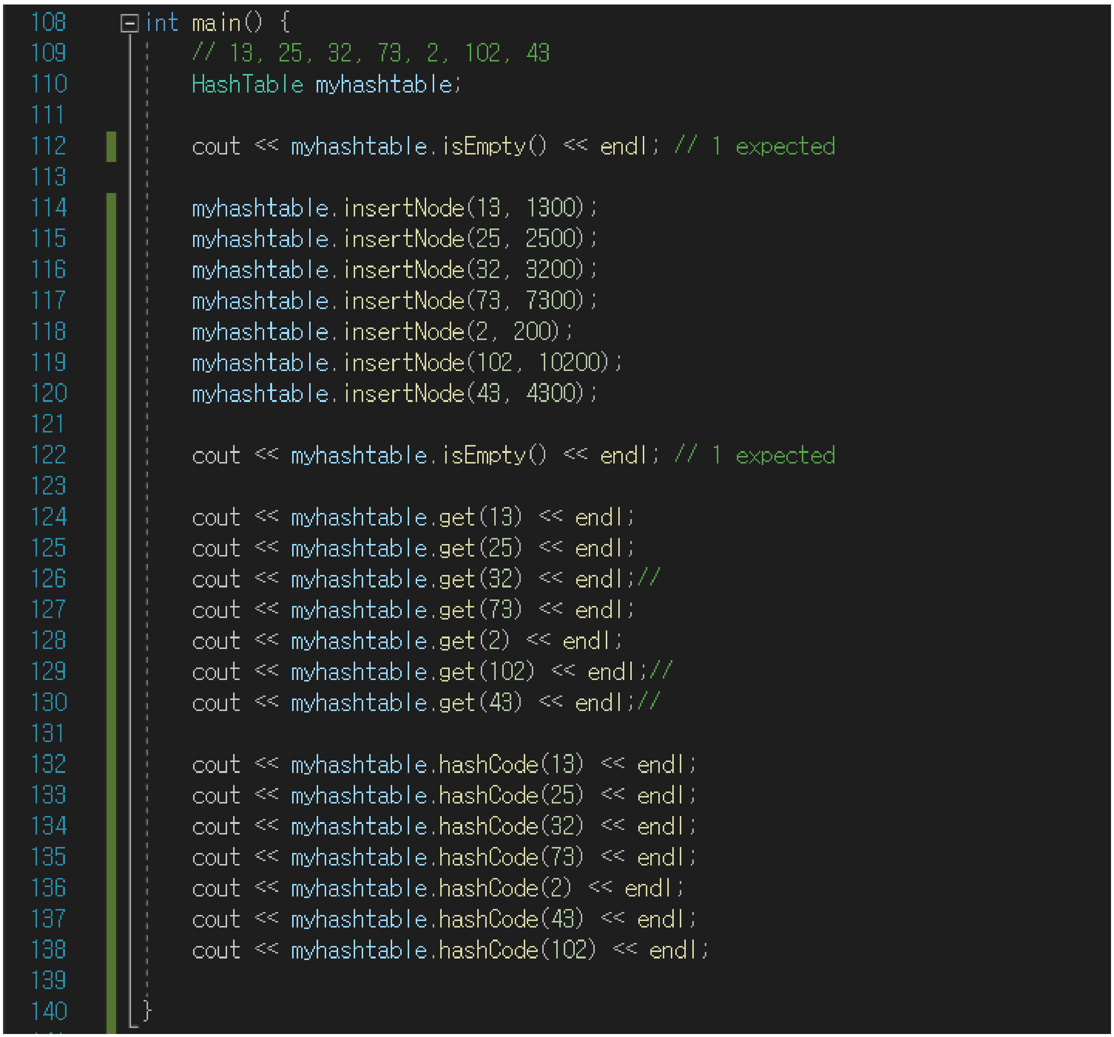
```
1
0
2
3
4
5
6
7
8
9
0
Hash Table is Full
Hash Table is Full
0
No Hashcode, so no value can be Found for the Key_2-1
No Hashcode, so no value can be Found for the Key_2-1
1200
4780
No Hashcode Found for the Key_200000000
No Hashcode Found for the Key_200000000
013FE578
013FE380
```

* It prints out 00000000 and -1 at the end of each error sentence because

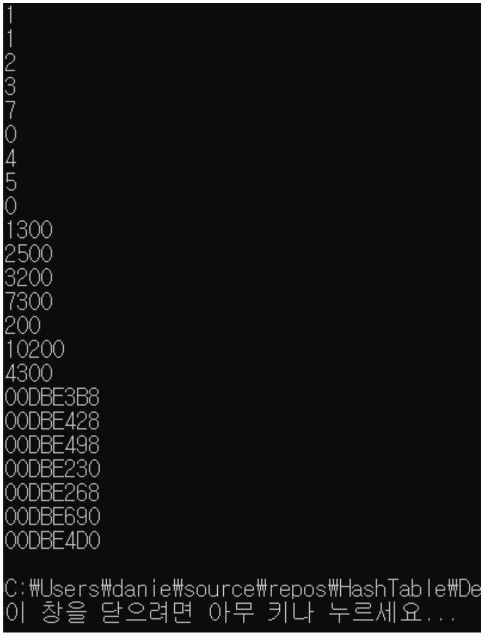
Node hashcode() function returns NULL as 8 bits

and int get() returns -1 in an error case.

EX 2)



```
int main() {
    // 13, 25, 32, 73, 2, 102, 43
    HashTable myhashtable;

    cout << myhashtable.isEmpty() << endl; // 1 expected

    myhashtable.insertNode(13, 1300);
    myhashtable.insertNode(25, 2500);
    myhashtable.insertNode(32, 3200);
    myhashtable.insertNode(73, 7300);
    myhashtable.insertNode(2, 200);
    myhashtable.insertNode(102, 10200);
    myhashtable.insertNode(43, 4300);

    cout << myhashtable.isEmpty() << endl; // 1 expected

    cout << myhashtable.get(13) << endl;
    cout << myhashtable.get(25) << endl;
    cout << myhashtable.get(32) << endl;//
    cout << myhashtable.get(73) << endl;
    cout << myhashtable.get(2) << endl;
    cout << myhashtable.get(102) << endl;//
    cout << myhashtable.get(43) << endl;//

    cout << myhashtable.hashCode(13) << endl;
    cout << myhashtable.hashCode(25) << endl;
    cout << myhashtable.hashCode(32) << endl;
    cout << myhashtable.hashCode(73) << endl;
    cout << myhashtable.hashCode(2) << endl;
    cout << myhashtable.hashCode(43) << endl;
    cout << myhashtable.hashCode(102) << endl;

}
```



```
1
1
2
3
7
0
4
5
0
1300
2500
3200
7300
200
10200
4300
00DBE3B8
00DBE428
00DBE498
00DBE230
00DBE268
00DBE690
00DBE4D0

C:₩Users₩danie₩source₩repos₩HashTable₩De
이 창을 닫으려면 아무 키나 누르세요...
```

Assume I have a test data set A(key) = {13, 25, 32, 73, 2, 102, 43}

If I create a default Hash Table using constructor, my code will create a Hash Table with size of 10.

It means, when I use Linear Probing hash function, the number in the 1's digit of each element in A will consider where to place the element in my Hash Table.

Accordingly, if there exist more than one element with same number in 1's digit, there will be a collision.

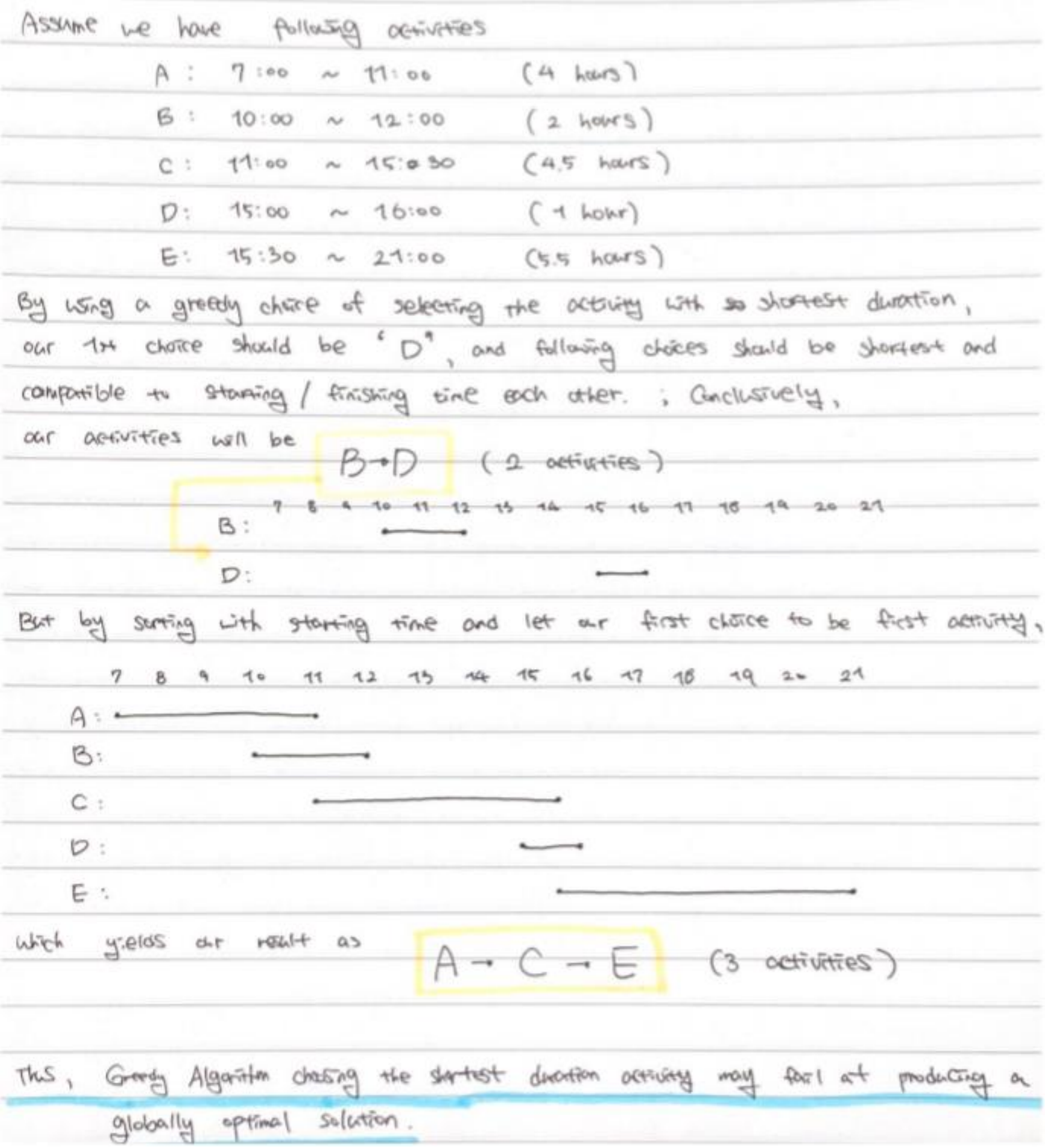
However, if I `div` each of element in set A by 10, they become {1, 2, 3, 7, 0, 10, 4}.

Now the elements will not have any collision even if we calculate `mod` of each of them.

Therefore, my h' function to be h' = key / 10 will yield efficient allocation of the elements into my Hash Table.

Problem 10.2 *Greedy Algorithms*

a) (2 points)



Assume we have following activities

A : 7:00 ~ 11:00 (4 hours)

B : 10:00 ~ 12:00 (2 hours)

C : 11:00 ~ 15:~~0~~30 (4.5 hours)

D: 15:00 ~ 16:00 (1 hour)

E: 15:30 ~ 21:00 (5.5 hours)

By using a greedy choice of selecting the activity with ~~so~~ shortest duration, our 1st choice should be 'D', and following choices should be shortest and compatible to starting / finishing time each other. ; Conclusively, our activities will be B→D (2 activities)

7 8 9 10 11 12 13 14 15 16 17 18 19 20 21

B :

D:

But by sorting with starting time and let our first choice to be first activity,

7 8 9 10 11 12 13 14 15 16 17 18 19 20 21

A :

B:

C :

D :

E :

which yields our result as A → C → E (3 activities)

Thus, Greedy Algorithm choosing the shortest duration activity may fail at producing a globally optimal solution.