BLG 335E

ANALYSIS OF ALGORITHMS I

CRN: 10825

REPORT OF HOMEWORK #3

Submission Date: 27.12.2013

STUDENT NAME: TUĞRUL YATAĞAN

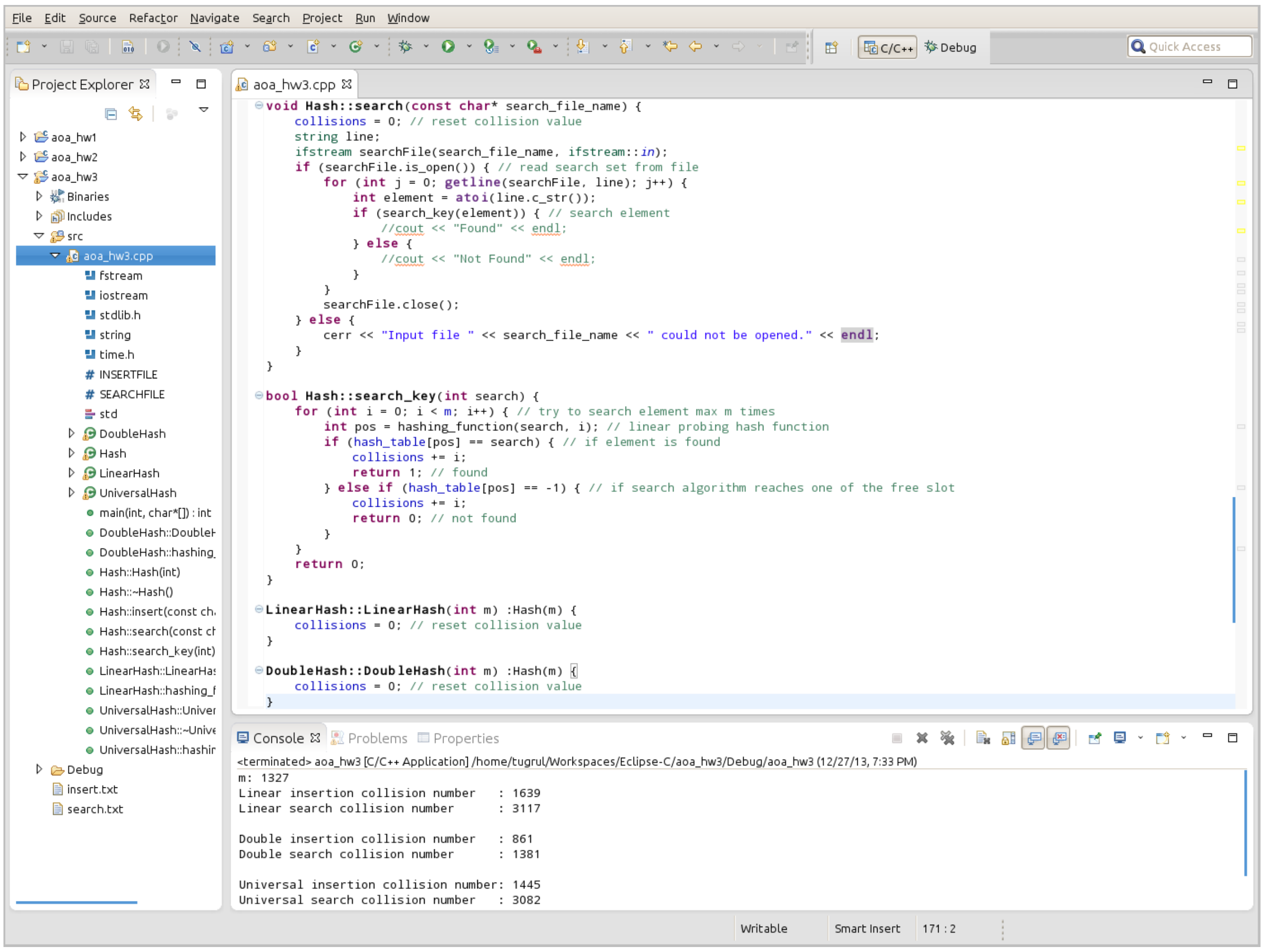
STUDENT NUMBER: 040100117

1. Introduction

In this project, we implement m-sized hash table that uses linear probing, double hashing and universal hashing strategies in order to store integer key elements by relative hash functions. Then we search integer keys in these hash tables, we calculate and compare the collision numbers among these algorithms.

2. Development and Operating Environments

Eclipse for C++ integrated development environment has been used to write the source code in Ubuntu 12.04 operation system and GNU g++ compiler has been used for compiling under Ubuntu 12.04 operation system.



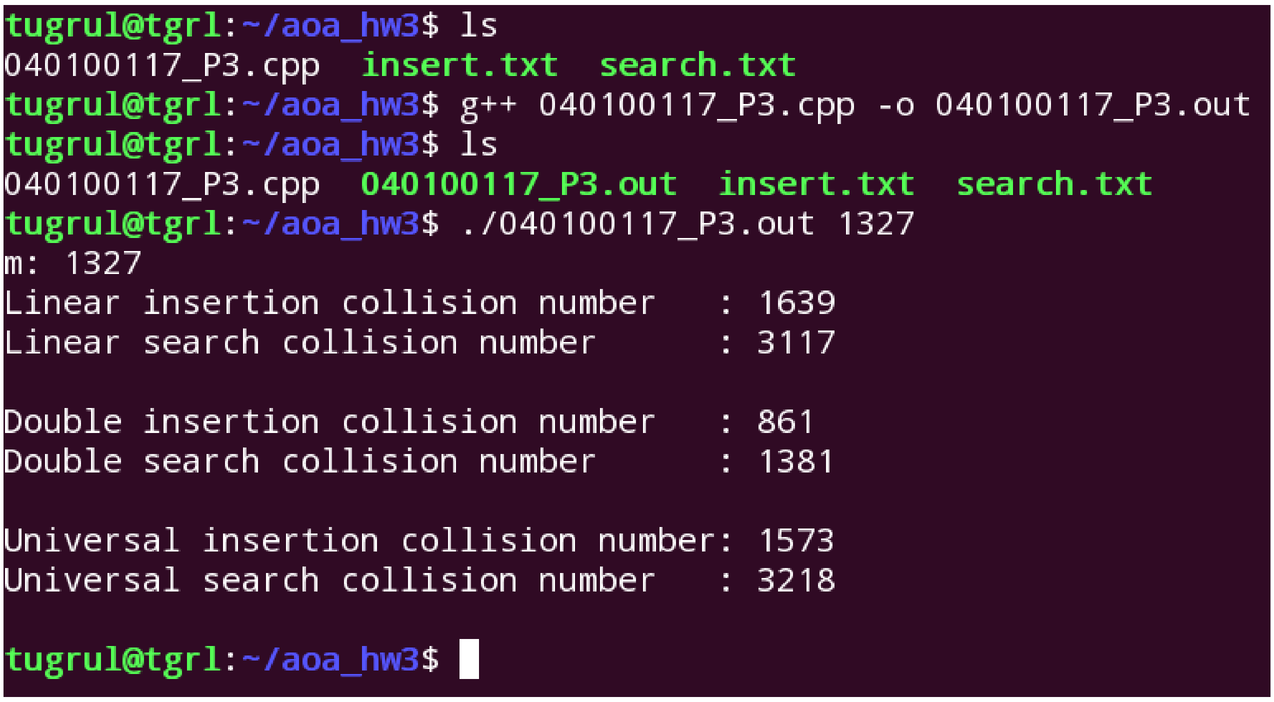
```cpp
void Hash::search(const char* search_file_name) {
    collisions = 0; // reset collision value
    string line;
    ifstream searchFile(search_file_name, ifstream::in);
    if (searchFile.is_open()) { // read search set from file
        for (int j = 0; getline(searchFile, line); j++) {
            int element = atoi(line.c_str());
            if (search_key(element)) { // search element
                //cout << "Found" << endl;
            } else {
                //cout << "Not Found" << endl;
            }
        }
        searchFile.close();
    } else {
        cerr << "Input file " << search_file_name << " could not be opened." << endl;
    }
}

bool Hash::search_key(int search) {
    for (int i = 0; i < m; i++) { // try to search element max m times
        int pos = hashing_function(search, i); // linear probing hash function
        if (hash_table[pos] == search) { // if element is found
            collisions += i;
            return 1; // found
        } else if (hash_table[pos] == -1) { // if search algorithm reaches one of the free slot
            collisions += i;
            return 0; // not found
        }
    }
    return 0;
}

LinearHash::LinearHash(int m) :Hash(m) {
    collisions = 0; // reset collision value
}

DoubleHash::DoubleHash(int m) :Hash(m) {
    collisions = 0; // reset collision value
}
```

```
<terminated> aoa_hw3 [C/C++ Application] /home/tugrul/Workspaces/Eclipse-C/aoa_hw3/Debug/aoa_hw3 (12/27/13, 7:33 PM)
m: 1327
Linear insertion collision number   : 1639
Linear search collision number      : 3117

Double insertion collision number   : 861
Double search collision number      : 1381

Universal insertion collision number: 1445
Universal search collision number   : 3082
```

The program built and compiled without any warning or error under g++ and the program executed with commands:

```
g++ 040100117_P3.cpp -o 040100117_P3.out

./040100117_P3.out 1327
```

Sample output is below:



```
tugrul@tgrl:~/aoa_hw3$ ls
040100117_P3.cpp  insert.txt  search.txt
tugrul@tgrl:~/aoa_hw3$ g++ 040100117_P3.cpp -o 040100117_P3.out
tugrul@tgrl:~/aoa_hw3$ ls
040100117_P3.cpp  040100117_P3.out  insert.txt  search.txt
tugrul@tgrl:~/aoa_hw3$ ./040100117_P3.out 1327
m: 1327
Linear insertion collision number    : 1639
Linear search collision number       : 3117

Double insertion collision number    : 861
Double search collision number       : 1381

Universal insertion collision number: 1573
Universal search collision number    : 3218

tugrul@tgrl:~/aoa_hw3$ 
```

3. Data Structures and Variables

Program takes m value as command line argument. Sample run command is:

```
./040100117_P3.out 1327
```

INSERTFILE and SEARCHFILE variables are file names for insert set and search set for hashing algorithms. These files must be under the same folder which is executable program is in.

```
#define INSERTFILE "insert.txt"
#define SEARCHFILE "search.txt"
```

- `void insert (const char*)`
  Function which inserts elements to hash table from file. Takes insert file name as an argument.

- `void search (const char*)`
  Function which searches elements from file in hash table. Takes search file name as an argument.

- `virtual int hashing_function(int, int) = 0`
  Pure virtual function which computes and returns the position according the hashing method. This function defined separately in all classes. Takes “i” and “k” values.

- `bool search_key(int)`
  Function which searches one element in hash table. Takes integer value to be search. Returns True if element is found and returns False if it is found.

- `return (k + i) % m`
  Linear probing hash function

- `return ((k % m) + i * (1 + (k % (m - 1)))) % m`
  Double hashing function

- `return (((a[0] * k[0] + a[1] * k[1] + a[2] * k[2]) % m) + i) % m`
  Universal hashing function. This function uses linear probing for open addressing with adding "`i`" to the universal function.

- `int m`
  Hash tables size

- `int collisions = 0`
  Collisions number, used for all methods

- `int *hash_table`
  Integer array which represents hash table

- `int *a`
  Array of random "a" numbers for universal hashing function

4. Analysis

The program is ran with different m parameters and collision numbers are computed by different hash size. Different insertion's and search's collision numbers are shown in these tables bellow.

	Insertion		
	Linear	**Double**	**Universal**
m = 1327	1639	861	1402
m = 2657	279	271	255

	Search		
	Linear	**Double**	**Universal**
m = 1327	3117	1381	2543
m = 2657	409	363	391

Linear probing method gives us the worst collision performance. It suffers from primary clustering, where small hash size occupied slots build up increasing the collision numbers. Moreover, the long runs of occupied slots tend to get longer.

Double hashing method almost always produces best results. Double hashing function ensures best element distribution over hash table so collision numbers are very low relatively to others.

Universal hashing method gives us average collision performance. Universal hashing method uses linear probing for open addressing so results with linear probing and universal hashing will be similar. Universal hashing methods also compels "m" to be prime. Universal hashing solves weak point of the open addressing methods with randomized decomposition.

In all methods wider hash table results smaller collision numbers. Wide hash tables increases the modular element of hash algorithm so elements will be aligned more distant from each other. This decreases the probability of collision. But when we start to decrease hash table size the increasing effect on linear hashing collision number and universal hashing collision number is higher than double hashing collision number.

5. Conclusion

During this homework, I have become more familiar with the concept of hashing algorithms and analysis of the algorithms. I had the chance to intensify my knowledge about instructing good and efficient algorithms.