BLG335E, Analysis of Algorithms I, Fall 2016 Project Report 4

Name: Halit Uyanık

Student ID: 150140138

Part A. Questions on Hash Tables (20 points)

1) Why do we use Hash Tables as a data structure in our problems? Please explain briefly. **(5 points)**

In order to have an efficient dictionary for our data. Its worst case is O(n) for search and expected search time is O(1). It is widely used in applications that require fast search and insert operations.

2) Consider a hash table consisting of M = 11 slots, and suppose nonnegative integer key values are hashed into the table using the hash function `h1()` :

```
int h1 (int key) {
int x = (key + 7) * (key + 7);
x = x / 16;
x = x + key;
x = x % 11;
return x;
}
```

Suppose that collisions are resolved by using linear probing. The integer key values listed below are to be inserted, in the order given. Show the home slot (the slot to which the key hashes, before any probing), the probe sequence (if any) for each key, and the final contents of the hash table after the following key values have been inserted in the given order: **(10+5 points)**

Key Value	Home Slot	Probe Sequence
43	1	1
23	2	2
1	5	5
0	3	3
15	1	1->2->3->4
31	0	0
4	0	0->1->2->3->4->5->6
7	8	8
11	9	9
3	9	9->10

Final Hash Table:

Slot	0	1	2	3	4	5	6	7	8	9	10
Contents	31	43	23	0	15	1	4	-	7	11	3

Part B. Implementation and Report (80 points)

Main Class – myHash:

```
class myHash {
public:
	int _tablesize;
	int collision = 0;
	myHash(int tablesize);
	void insert(string s);
	bool isSame(long long int, string);
	long long int hashFunc(string s);
	bool retrieve(string s);
	bool spell_checker(string s);
	bool remove(string s);
	void printCollision(long long int);
	void printOutput();
	void readFile(char *);
private:
	string *myHashTable;
	int totalEntries = 0;
};
```

Definitions:

Local variables:

1)_tablesize : holds the user defined size for the hash table.

2)Collision : holds the total collision count.

Methods:

1)myHash:

*params : int tablesize

*functionality: creates a new myHash object.

2)İnsert:

*params: string s

functionality: inserts the given string to the hash table. First calculates the initial index via hashFunc, then starting from that point checks until it finds a place to put the number in. The emptiness is defined as whether the place is empty or equal to '' character.

3)isSame:

*params: long long int location, string s

*functionality: checks whether the given string is equal to the one in the given location.

4)hashFunc:

*params: string s

*functionality: Calculates the initial location of a string, since it takes the mod operation after each multiplication there is no overflow problem.

```
long long int result = 1;
for (int i = 0; i < s.size(); i++) {
	result *= (s[i] % this->_tablesize);
	result = (result % this->_tablesize);
}
return result;
```

5)retrieve:

*params: string s

*functionality: Tries to locate the given string in the hash table. It searches till it finds the string, reaches itself, or sees an empty location. After searching if the item doesnt exists, then it calls the spell_checker.

6)spell_checker:

*params: string s

*functionality: Called in case retrieve fails. Gives suggestions for the given string. It uses the ASCII decimal values to change each of the bits of a string one by one and processes search operation through the hash table.

7)remove:

*params: string s

functionality: searches the hash table for the given string. If it is found, it is replaced with '' character.

8)readFile:

*params: char *

*functionality: Reads the file given by user. It reads character by character each line, and puts them to either operand or the word. Then calls the respective methods from the hash class.

Rest of the methods are mandatory print functions.