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

Name: Göktürk GÖK

Student ID: 150110029

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)**

Hash tables look for the hashed key slots instead of looking for the whole array from beginning to the end. This makes hash table more quick and guidable to access to key. The key which is searched is quickly accessed by its hash function value which is its original slot so that it takes O(1) time to access while normal array while the normal array takes O(n). This gains Hash table data structures more advantages than normal. Especially if a dictionary implementation is required then using hash tables for its alphabetic structure is the best way. There are two ways to get rid of collisions which are open addressing(probing) and chaining(linked list). Using linked list makes running time O(n) to be accessed because of traversing nodes.

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	
23	2	
1	5	
0	3	
15	1	2, 3, 4
31	0	
4	0	1, 2, 3, 4, 5, 6
7	8	
11	9	
3	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)

Program evaluates the **EMPTY** slots as **“”** and **DELETED** slots as **“~~~~0~~~~”.**

- MyHash Class
  - **insert(string s) :** First takes a word to be inserted then takes its original home slot with hashFunc(s). Then starts to look for an EMPTY(“”) or DELETED(“~~~~0~~~~”) slots to insert the words by traversing from its original slot to itself until it finds an available slot to insert. If it cannot inserted at its original home slot then collision occurs. Traversing the next slot collision is increased as 1 till it is inserted at an available slot. If whole table is full then collision equals its table size and gives proper messages to be displayed.
  - **retrieve(string s):** Firstly takes the original home slot of given string s by using hashFunc(s) function. Then traverse the whole table starting its original home slot. If it finds then return **true** , if not then returns **false**.
  - **hashfunc(string s):** Calculate the slot index value in table of given string s by multiplying ascii values of each character in given string and takes its mod with the table size. In my implementation it takes the mod in every step to avoid overloading issue.
  - **remove(string s):** This function takes string to remove then search the string s using search(s) function if it exist in the hash table. If it exist then this function makes its value as “~~~~0~~~~” (lazy deletion) and returns **true**, if not then gives the proper messages and returns **false.**

```
class myHash {
public:
    int _tableSize;
    int collision=0;

    myHash(int tableSize); // Creates a hash table that can store up to tableSize entries.(Constructor)

    void insert(string s); // Inserts w into the hash table.
    int hashFunc(string s); // Computes the hash value of w.

    bool retrieve(string s); // Finds index of a given word in the hash table, if it isn't inthe table, starts for spell checking
        and suggests similiar words if any found.
    bool remove(string s); // Removes given word from the dictionary.

    // ************ Additional Methods *****************
    bool spell_checker(string s);
    int search(string s);
    void readParseEvaluateFile(ifstream &inputFile);
    void writeOutputFile(ofstream &outputFile );
    // *************************************************

    ~myHash();
private:
    string *myHashTable; // Stores words.
    int totalEntries=0; // Stores the current number of entries in the table.

};
```

4 extra methods are added in given myHash class here .

- **spell_checker(string s)** function which takes a string, produce and search alternative words to be found in hash table per changing each of its characters.
- **search(string s)** function which takes wanted string to find in table. It starts to search from its home slot which equals the hashFunc() value. If it cannot find then continue to search by traversing the table by itself. If it has found the key then returns its index , if it cannot find then returns -1. while searching if there is an "EMPTY" slot after its original home slot (

```
if(myHashTable[counter] == ""){
    isFound = true;
    return -1;
    break;
}
```

=hashFunc(word)). Then the word can not be found because if the word would have been in the table slot after its original home slot , word could have been inserted there (next empty slot).

- **readParseEvaluateFile(ifstream &inFile)** which takes an input file. It reads the input file given as argument (.txt), parses the line as 'command' and 'word', Evaluate the operations according to taken command and finally prints the total collisions.

- **writeOutputFile(ofstream & outputFile)** which writes whole the myHashTable array into text file .