Assignment 3 Part 1

clo4

Total marks 150

Q1) We want to design a small search engine that can help its users to retrieve words related to the search query from a file. The unique terms can be stored in an unordered array/linked list. The search operation is therefore O(n) in the worst case. If we replace the array/list with a balanced BST, then we can search these terms in O(lgn) time in the worst case. Also if we replace the array/linked list with a hash table then we can search these terms in O(1) on the average. You task is to use Hash Tables with chaining as the collision resolution strategy. You can use hash function of your choice that you think is most suitable for this application.
[150 Marks]
You must make a program to count the words inside of a text file, using a hash table of your own creation.
Count the frequencies of every word in the file.

- Output the total number of distinct words it found.
- Offer the user a prompt, to query the exact count of a particular word. 🎬 If the user enters a word prepended with a -, it will delete the word.
- Continue until the user just hits enter, at which point it will terminate

Run of your program might appear like this:

```
This text contains 9852 distinct words.
Please enter a word to get its frequency, or hit enter to leave.
> france
"france" appears 59 times.
> guillotine
"guillotine" appears 26 times.
> -guillotine
"guillotine" has been deleted.
> guillotine
"guillotine" does not appear. >
Goodbye!
```

Words will be counted in a case-insensitive manner (That is, "Cat" and "cat" are the same.) Also, all non-word characters (any character that's not a letter, number, or underscore) should be considered to be a word boundary, except for an apostrophe "'". An apostrophe should be removed if it's at the beginning or end of a word (e.g. "'tis"), but not when it's internal (e.g. "isn't").Be careful with repeated non-word characters. For example, the file k.txt contains words separated by dashes, such as "yet--oh". Your program should be smart enough to see that this is just two words. The hash table may use either separate chaining and double hashing both separately.

IMPLEMENTATION

The class must contain the following methods:

- **A constructor**
- **put(key, value)** which maps key to value in the table. Its time should be roughly constant, except when rehashing is done.
- **get(key),** which returns the value to which key maps. It should be roughly constant time.
- **contains(key),** which returns whether or not key is a valid key in the table, in average constant time.
- **delete(key),** which removes key and its value from the table. It should usually be average constant, though it might be linear if you use chaining.
- **size(),** which returns the number of elements the hash table contains in constant time.

The hash table's initial capacity should be near 10. However, you must make sure that your load factor α does not get above 0.75. If it does, you must immediately rehash to a new prime number roughly twice the current one. Thus, you will likely have a private method called rehash(), and a static array containing useful prime numbers. One such list of primes is the following: 11, 19, 41, 79, 163, 317, 641, 1279, 2557, 5119, 10243, 20479, 40961, 81919, 163841, 327673

You do not need to worry about shrinking the table's capacity when enough key-value pairs are deleted. Also, you do not need to rehash when your table's size is greater than 200,000.

What is a Load factor:

On an average, if there are n entries and b is the size of the array there would be n/b entries on each index. This value n/b is called the load factor that represents the load that is there on our map.

This Load Factor needs to be kept low, so that number of entries at one index is less and so is the complexity almost constant, i.e., O(1).

The purpose of the load factor is to give an idea of how likely (on average) it is that you will need collision resolution if a new element is added to the table. A collision happens when a new element is assigned a bucket that already has an element. The chance that a given bucket already has an element depends on how many elements are in the container.

```
load factor = # of elements / # of buckets
```

The default load factor is 0.75f (75% of the map size).

That means if we have a HashTable with an array size of 100, then whenever we have 75 elements stored, we will increase the size of the array to double of its previous size i.e. to 200 now, in this case.

EXAMPLE:
Let's understand the load factor through an example. **If**
we have the initial capacity of HashTable = 16.
We insert the first element, now check if we need to increase the size of the HashTable capacity or not.
It can be determined by the formula:

Size of hashmap (m) / number of buckets (n)

In this case, the size of the hashmap is 1, and the bucket size is 16. So, 1/16=0.0625. Now compare this value with the default load factor.

0.0625<0.75

So, no need to increase the hashmap size. We do not need to increase the size of hashmap up to 12th element, because

12/16=0.75
As soon as we insert the 13th element in the hashmap, the size of hashmap is increased because:

13/16=0.8125
Which is greater than the default hashmap size.

0.8125>0.75

Now we need to increase the hashmap size