Assignment 6

Name: Yinglue Chen

NetID: yxc180006

Email: yxc180006@utdallas.edu

In this assignment we're going to implement a hash table, which uses quadratic probing to handle collision.

```
public class myHashTable {
    private String[] table;
    private List<String> wordsForRehash;
    private int collisionTimes;
```

The way I detect the load is that after I insert a word, I check whether 2 * wordForRehash.size() is larger or equal to table.length. If it is, then the size of the table should be extended.

The collision number I set here is to count the collision after the program starts. If we have to expand the size of the hash table and we meet collision in the new hash table, I still add it to "collisionTimes". Also, if one word has several times during quadratic probing, I still count them in "collisionTimes". So in the output of this program you can see the collision number could be larger than it actually be in the new hash table. But in fact this can help us to choose a proper method to avoid collision as much as possible.

In my implementation of this hash table, I add five functions to add elements or print the information. Here are these functions:

- private myHashTable(int size): this is the construct method of myHashTable class. The input "size" is to set the initial value of the size.
- private void addWord(String s): this is for adding the elements into the hash table. When the percentage of load has reached 50%, it will automatically increase its size, by doubling the size and increasing itself to the next prime number.
- private void printCollision(): this is for printing out the total number of the collisions.
- private void printTableSize(): this is for printing the size of the hash table.
- private void printHashTable(): this is for printing the whole hash table, including the elements and its index. The format will be "index→elements".

Now here are my inputs, in the first screenshot. The next screenshot is the result of running this program.

```
    public static void main(String[] args){
        /**
         * myHashTable(int size): myHashTable test = new myHashTable(31);
         * addWord(String s): test.addWord(s);
         * printCollision(): test.printCollision();
         * printTableSize(): test.printTableSize();
         * printHashTable(): test.printHashTable();
         */
        myHashTable test = new myHashTable(31);
        String[] words = {
                "Hash", "table", "Pick", "word", "lengths",
                "maximum", "minimum", "Insert", "ASCII", "space",
                "probing", "size", "adequate", "Chen", "Yingue",
                "UTDallas", "ECSS", "Algo", "2018Fall", "Texas"
        };
        for (String s : words){
            test.addWord(s);
        }

        test.printTableSize();
        test.printCollision();
        test.printHashTable();
    }
}
```

```
"C:\Program Files\Java\jdk-11.0.1\bin\java.exe" "-javaagent:D
Table extension needed. Extending the table...
The size of the table: 67
The total number of the collision: 15
16->probing
20->lengths
22->Yingue
25->UTDallas
26->Insert
27->minimum
29->maximum
30->ASCII
34->ECSS
38->adequate
41->size
42->word
47->Chen
48->Texas
50->2018Fall
51->table
52->Algo
53->Hash
55->space
56->Pick

Process finished with exit code 0
```

Question: If the table size is more than 20,000 and the words to insert are 8000, is the above hashing algorithm adequate? Why?

Answer: It’s not adequate. The hash value would be in a scope of [144, 976], because the minimum case would be "000" (hash value = 48 * 3 = 144), and the maximum case would be "zzzzzzzz" (hash value = 122 * 8 = 976). Now I consider the integers in [144, 976] as "pigeonhole", so there are 976 - 143 = 883 "pigeonholes". And also there are 8,000 "pigeons", because there are 8,000 words. Therefore for one pigeonhole there will be at least 9 pigeons, which means there are at least 9 words that will have the same hash value. 883 different hash values, and each on them contains at least 9 words, there are already 883 * (9 - 1) collisions.

Moreover, let's assume I insert the words from one of these pigeon holes, say, pigeon hole A, first. Now, there are possibilities that when I use quadratic probing on pigeon hole A, the words might occupy the place which should originally belong to the words from pigeon hole B. Same thing for the words in pigeon hole B: when I insert them, the words might also occupy other places. Therefore the total number of collision will be even larger than the number I counted above.

According to these two reasons, I think for the words that have the length from 3 to 8, using this algorithm as a hashing function is not adequate, even we use quadratic probing here.