1. public StringTable(int nBuckets);

- What it does?
  - This function is a constructor of StringTable class and it creates 'nBuckets' size LinkedList and initialize of the LinkedList. It also saves 'nBuckets' value into the instance variable for future usage.
- Did it use any instance variables of the class, and if so, which ones?
  - Yes. It uses two instance variables 'buckets' and 'nBuckets'. The 'buckets' is used for saving the address of LinkedList array and 'nBuckets' is used for saving the size of LinkedList array.
- Are any helper methods called by the method (either of your own creation or in the original codebase), and if so, which ones and what do they do?
  - No. it does not call any helper methods.

2. public boolean insert(Record r);

- What it does?
  - This function adds a 'Record instance' into the StringTable. First, it finds out whether the item passed as a parameter exists. It returns 'False' if the item already exists, otherwise it appends the item and returns 'True'.
- Did it use any instance variables of the class, and if so, which ones?
  - Yes. It accesses 'buckets' instance variable to add the item into bucket. It also uses 'size' variable to increase the number of 'Record' currently stored in the StringTable.
- Are any helper methods called by the method (either of your own creation or in the original codebase), and if so, which ones and what do they do?
  - Yes. It uses three helper methods toIndex(), stringToHashCode(), getBiggerBucket(). It uses toIndex() and stringToHashCode() to find the proper index of the LinkedList array with the given parameter. And the getBiggerBucket() is used for making new bucket list when the load factor exceed certain limit.

3. public Record find(String key);

- What it does?
  - This function finds the matching item inside of the StringTable. It receives a 'key' parameter and finds the record with a key matching the input parameter. Using the 'key' parameter, try to compare with 'Record' items which are in 'buckets'. It is important to use String.equals() function when comparing String value!!
- Did it use any instance variables of the class, and if so, which ones?
  - Yes. It accesses 'buckets' instance variable to retrieve the items from the bucket.
- Are any helper methods called by the method (either of your own creation or in the original codebase), and if so, which ones and what do they do?
  - Yes. It uses two helper methods toIndex(), stringToHashCode(). It uses toIndex() and stringToHashCode() to find the proper index of the LinkedList array with the given parameter.

4. public void remove(String key)

- What it does?
  - This function finds a record in the StringTable using the given key and remove the record if it exists. After finding the proper bucket using toIndex() to the given key, the 'key' parameter is compared to all items in the bucket to find a match.
- Did it use any instance variables of the class, and if so, which ones?
  - Yes. It accesses 'buckets' instance variable to retrieve the items from the bucket. It also uses 'size' variable to decrease the number of 'Record' currently stored in the StringTable when removing the item.
- Are any helper methods called by the method (either of your own creation or in the original codebase), and if so, which ones and what do they do?
  - Yes. It uses two helper methods toIndex(), stringToHashCode(). It uses toIndex() and stringToHashCode() to find the proper index of the LinkedList array with the given parameter.

5. private int toIndex(int hashcode);

- What it does?
  - This function converts a string's hashcode to a LinkedList index. It uses 'Multiplicative hashing strategy' to get an index from a hashcode. I'll go into more detail in the next question.
- In particular, describe your computation to map hashcodes to indices in the table for the toIndex method?
  - As we saw in Prof's video lecture, I used 'Multiplicative hashing strategy'. First, multiply hashcode by A to get a fraction value. I used 'A' as a complex random value not to get a same result. According to the lecture, simple 'A' value is not good for even distribution. After that, getting a fraction value from k*A and then multiply the value by m which is bucket size.
  - After getting the index, there is one more problem. The value could be a negative value, so I need to deal with that because there is no negative value in an array. So I adds up the size of LinkedList array and apply modulo operator one more time to get a positive value.

```
private int toIndex(int hashcode)
{
    // Multiplicative Hashing!!
    int m = nBuckets;
    int k = hashcode;
    double A = 0.987654321;

    int ret = (int) (m*((k*A)%1));

    // In case of minus value, adds up size of nBucket
    // and apply modulo operator one more time
    ret += m;
    ret = ret % m;

    return ret;
}
```

- Did it use any instance variables of the class, and if so, which ones?
  - Yes. It uses 'nBuckets' to get the size of LinkedList array. Because it is used for calculating the indices.
- Are any helper methods called by the method (either of your own creation or in the original codebase), and if so, which ones and what do they do?
  - No. it does not call any helper methods.

6. In case of accessing the most recently inserted item: **The addFirst() function is better** than add() function because the addFirst() function pushes the 'Most recently inserted item' at the head of the bucket. Eventually, the items we are looking for are at the front of the bucket. Therefore, the find() function is going to find the 'Most recently inserted item' quickly.

In case of accessing the least recently inserted item: **The add() function is better** than addFirst() function because the add() function adds the item at the end of the bucket. Hence, the least recently items move up to the head of the bucket. Eventually, the items we are looking for are at the front of the bucket. Therefore, the find() function is going to find the 'Least recently inserted item' quickly.

7. There are two questions.

- Why would maintaining a fixed maximum load factor help the performance of the table?
  - We need to create more buckets to keep the fixed maximum load factor. Assuming there is no load factor, chaining occurs on specific buckets of the hash and it will greatly reduce the performance of hash functions. The more buckets we have, the better the distribution of items will be. Less chaining occurs on a single bucket, which improves the performance of the hash functions.
- What is the average-case running time of table operations if a fixed maximum load factor is used? If we have a fixed maximum load factor and then we can access items almost in a L constant time and can insert item instantly. So, the average time complexity shown below.
  - find(): $O(L)$
  - insert(): $O(1)$
  - remove(): $O(L)$

8. Sketch pseudocode for moving items from a old one to a new one.

```
// Old Array B of m buckets
// New Array B` of m` buckets

// Getting a load factor
// (number of items) / (number of buckets)
double loadFactor = size / m;

// if the load factor exceed certain limit(LOAD_FACTOR),
// And then make the bucket bigger and moves all the items into new bucket.
if( loadFactor >= LOAD_FACTOR ) {

    // make the bucket size double(Let's say making it double)
    m` = m * 2;

    // make and initailize the new bucket list
    B` = new LinkedList[m`];
    for(int i=0;i<m`;i++) {
        B`[i] = new LinkedList<Record>();
    }

    // move old bucket's items into the new bucket.
    for(a current item : Old Bucket B) {
        // Getting a proper index with the Old bucket items
        B`[toIndex(stringToHashCode(B.key))].add(a current item);
    }
}
```

9. How much time does it take (asymptotically, on average) to transfer n elements from the old to the new table, assuming simple uniform hashing? Be sure to explain where your answer comes from.

- Transferring n elements from the old to the new table will take $O(n)$.
- As we can notice from the pseudocode of previous question, we need a n-times to access the old Bucket's items. Adding an item into the new bucket only requires $O(1)$ because getting a hashcode, a proper index and adding the item requires constant amount of time. And when evaluating time complexity, there is no need to care about constant time.
- Therefore, the time complexity moving from n items from the old to the new table will need $O(n)$.

10. Describe a strategy for deciding when to allocate a new table, and what size the new table should be, so as to keep the maximum load factor ≤ L while maintaining an amortized average-case cost of insertion Θ(1).

- Let's say doubling the size of the table when the Load factor exceeds L value. Whenever factor reaches L, the table is doubled but the frequency will be decreased exponentially. So there is no need to think about the cost of resizing the table. Hence, we can assume that inserting n elements into an empty array takes O(n) time. Therefore, amortized runtime for insert operations is $O(1)$.