1. public StringTable(int nBuckets);
   The StringTable is a construtor that initializes the linked lists for each bucket. It uses all three instance variables. It keeps track of the nBuckets initialized and then uses that to create all the linkedList buckets. Then size for each bucket is started at zero. There are no helper methods for the constructor.

2. public boolean insert(Record r);
   The insert method checks to see if a key has already been added in a bucket. If it hasn't it adds the key to its specific bucket determined by its hashcode. To do this it has helper methods. First it uses stringToHashCode and toIndex to first come up with the hash code to the string then comes up with a bucket number from the hashcode. The formulation used to map the hash code to an indix was from the multiplication method where you multiple the hash code by a number between zero and one. Then mod it to get the remainer and multiply by the number of buckets to find which one it should go in. It was important to do the absolute value because when the numbers were large it could result in a negative sign added. Once this is known, the find method is called to see if the key is already in the records. If not the buckets instance variable is used to add the key and the size is increased.

3. public Record find(String key);
   The find method finds the type record that matches the keys position in one of the buckets. It does this by iterating through a bucket based on the bucket number found from using the helper methods, stringToHashCode and toIndex. Also the instance variable, buckets, is used to pick the bucket which the string is in. If the key string is equal to one of the strings in the bucket it will return the record which holds the key.

4. public void remove(String key);
   The remove method finds the key and removes it from the bucket. The find method is used as a helper method to determine if the key is even in a bucket. If it is then using the instance variable buckets the remove function is used to take the string out. Then the instance variable size is used to update the size of that bucket.

5. private int toIndex(int hashcode);

   The toIndex method turns a strings hashcode to a specific bucket. The multiplication rule was used to sort the strings into buckets more uniformily. No helper methods were used and only the instance variable nBuckets was used as part of the multiplication.

6. Java linked lists, which you used to implement each hash bucket, have both head and tail pointers. Hence, it does not matter to the cost of insertion whether you use add() or addFirst() to insert a new item.
   Assume that the find() method traverses a bucket starting from the head of its list. If the person using your hash table frequently accesses the most recently inserted item(s), which of these two insertion methods (i.e., add() or addFirst()) is likely to yield better performance, and why? What if the user frequently accesses the least recently inserted item(s)?

   If the user was accessesing the most recently inserted item then the addFirst() method would be better because then the user would always be able to check the first insertion to find what they were looking for. While if the user is frequently accessing the least recently inserted item then the add method would be better because then the first item in the list would be the least recently inserted item.

7. Our hash table is allocated with a fixed number of buckets. If we insert a bunch of values, the load factor of the table (number of items over number of buckets) can grow without bound. We'd instead like to maintain the table's load factor some constant L, no matter how big it grows. Why would

maintaining a fixed maximum load factor help the performance of the table? What is the average-case running time of table operations if a fixed maximum load factor is used?

Maintaining a fixed load factor would help performance because then it keeps the buckets from getting flooded with items. This then helps when searching through a bucket for the item of interest. The average -case running time of table operations if a fixed maximum load factor is used is $\Theta(1)$ because the amount of buckets can change to make a max amount of items in that bucket. This allows the operations to take less time.

8. One way to decrease the table's load factor is to make it bigger; that is, we create a new table with more buckets and then transfer the items from the old table to the new one. Sketch pseudocode for this operation. Assume we have an existing array B of m buckets, and we are transferring its contents to a new array B0 of m0 ¿ m buckets. Be sure to specify which value, m or m0, is used by your toIndex() function.

m0 −− > 2*m
B0 −− > new LinkedList(m0)

toIndex Mehtod
for(int i = 0; i < m ; i++)
for(Record i: B[i])
double bucketNum = Math.abs(m0*((.35789*stringToHashCode(i.key))% 1))
return (int) bucketNum
end
end

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.

Assuming SUH, the average time to transfer n elements from the old to the new table would be O(n). This is because getting an item from the first array would be of order O(1). This is true if the load factor is low. So there would be relatively few items in each bucket to look through to find the item of interest. But then putting that item into the new array would take order O(n). So the average time would take 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 (1). (Hint: remember your first couple of studios!) Justify that your answer meets these constraints.

To keep the maximum load factor less than L, I would double the size of the array. This would make the load factor L half of what it was previously. Which keeps it below L. This would allow insertion method to stay at $\Theta(1)$ because there won't be to many items in each bucket for it to check and see if the item is already in there..