Programming implementation:
I created the base and step hash functions using multiplicative hashing, by which I set A=$\frac{\sqrt{5}-1}{2}$ for h1 and A=$\frac{\sqrt{7}-2}{5}$ for h2. This implementation of irrational number, as we learned during class, will help reduce the probability of bad hashing.
For the StringTable class, I created an array of Records to hold the different keys and their corresponding locations. I initially set the size of the table to be 4 times the input size, as we want a maximum load factor α = .25, but as for the dynamic structure, the initial table size is 2.
I would not repeat the hash function here since it is the same for all three **insert**, **find** and **delete**. One thing to notice is that if we are inserting a key, we do not need to worry about if the slot has been taken before, and we can simply insert the key at the first empty slot we find. However, as for **find** and **delete**, we would check if the slot has had some key. When deleting a key, I would set the slot to a new Record with key "". Therefore, when we check the slot, we would know if the Record is null or had some key before.
Additionally, to compare the keys we need to use .equals() instead of == because they are strings.
Finally, there are actually some same keys during the inserting process because each pattern string has some same segments within them. I set the program to return true if the same key is found in the table, meaning that this key is already inserted.

Compare hash keys instead of original strings:
To achieve this, I created another array of integers that would store the hash key of key when we insert the keys into the table. By this way, we are no comparing long strings but simple integers, which would make the program faster in **insert**, **find** and **delete** because we need to compare the keys in all the three functions. After we find two keys with same hash key, we would proceed to check if their strings are the same, because there does not exist a toHashKey function that would give every string a unique hashkey.

Dynamic Structure:
Firstly, StringTable will generate a table of size 2 no matter what the input size is. Then, before each insertion, my algorithm will check if the number of keys added divided by the size of the table exceeds .25, which is the maximum load factor we want. If the load factor goes above .25, we would call a rehash function that would create a temporary StringTable with all the capacities doubled, such as hashKeys array and table size. Then, we would go through each of the old slots and insert the non-empty ones whose key is not "" into the new one by hashing. We do not need to worry about changing the hash functions because their values automatically change with the table size. After we rehash all the keys into the new table, we replace it with the old one, and therefore maintain a dynamic growing hash table. The idea we use here is that we only need to check if the load factor is too large every time a new key is inserted.