Description

Problem 5.3 asks to write a program to compute the number of collisions required in a long random sequence of insertions using linear probing, quadratic probing, and double hashing.

I approached this by doing everything in the main class. I made a file of random numbers named "input.txt". I then scanned through this file and added all the numbers into an array list.

I then set the size as the first prime larger than double the amount of values, to prevent infinite collisions during quadratic probing. I then created the hash table for linear probing.

The program loops through all the values in the linked list. It finds the hash code with the function value % tableSize, and if that position in the table is already taken, it goes into a while loop where 1 is added to the hashcode and then again modularly divided by the tableSize, until an empty spot is found. Also inside the while loop is an increment of the count of collisions. Then, the found spot is updated to hold the value.

The quadratic probing algorithm works very similarly, except for the fact that when the position is already taken, instead of repeatedly adding 1, the square of the loop number is added instead, and this is kept track of with a variable i which is incremented until an empty spot is found.

The double hashing algorithm also works very similarly, except that when dealing with a collision, this algorithm adds a multiple of a second hashing function. This second function is defined as the largest prime number less than the table size minus (the value modularly divided by the same prime number). Similar to quadratic probing, the multiple of this value is kept track of with a variable i which is incremented until an empty spot is found.

Finally, all the results are printed.

Some other methods also exist in the program, and they include the nextPrime function, previousPrime function, and isPrime function, which all exist to help the above algorithms function properly.

Complexity Analysis

All three algorithms have the same average case and worst case time complexities, and the same reason for having them.

The time complexity of all three algorithms for inserting n elements into a hash table with m slots is O(n) in the average and best case, and $O(n^2)$ in the worst case.

In the average and best case, the algorithms have a small constant number of probes per insertion, which means that the time complexity is linear in the number of elements being inserted.

However, in the worst case, the algorithms may result in a sequence of probes that visits every slot in the hash table before finding an empty slot. This can happen if the hash function produces a sequence of values that all map to the same slot. In this case, the time complexity would be $O(n^2)$.

It's worth noting that the worst-case scenario is unlikely to occur with a well-designed hash function and a suitably sized hash table, which I have implemented. Therefore, these three algorithms should always take O(n) time in my implementation.