1. kiss Near East she is entertaining in consideration of knock off maxim and foal teasing in favor of major premise structures
2.
    a. The starting capacity of textAssociator should be a prime number because the reason is to avoid clustering of values into a small number of buckets.
    b. I choose to expand my load factor when it reaches 0.75 instead of 1. The reason you want number of buckets N to be greater than number of keys n, is that no algorithm (hashvalue + compression) gives a perfect even distribution of keys though the buckets. So when n starts approaching N, the odds of collisions become higher. Which means you will have longer lists in certain buckets. Thus increasing the time to find that key.
    c. I double the size, it won't take too much space or take too long to make run time too big comparing with other method.
3. I used the String's hashCode method from Java, and mod(modulo) by the current table size. It is implement by the Java class. It is easy to use and keep the hash index in the table. I was thinking to implement it all by myself to find a way that it is almost impossible to have a same hash index after the hash function so it won't cause collision and the bucket won't be too full or empty. The hash function inside of WordInfo is a good hash function. The 31 is a large enough odd prime that the number of buckets is unlikely to be divisible by it (and in fact, modern java HashMap implementations keep the number of buckets to a power of 2). 31 is used for Strings as there is less than 31 letters in the alphabet meaning all words of up to 6 letters have a unique hash code. If you use 61 (prime less than 64) for example, up to 5 letters would produce unique codes and if you use 13 (prime less than 16) you can get collisions with two letters words.
4. I would choose double hashing. The another hash function I could use the function from WordInfo class. First I need to take off the WordInfoSeparateChain because it would be useless. Then change the table structure from WordInfoSeparateChain to a simple String array. When adding the new word implements AddNewWord method, instead of making a new chain when there is no word in the certain index, I would also check if

there is a word before insert. If there is, using another hash function to find a new index until find an empty spot. It would cause to implement another helper method for getting a hash index when there is collision happened, besides the regular hash function.