Naruth Kongurai (1429760)

CSE 373 AB

TA: Raquel Van Hofwegen

02/10/17

1. My favorite sentence is “embrace everywoman yourself is joyfulness headed for compose pandect and press teasing at intimacy structures”
2. 1. I decided that the starting decision would be 31. I set it to 31 because it is a prime number and so if we were to mod our hash value with 31, it will hopefully provide a better overall distribution to our table. I didn’t want to set the initial capacity to be so large or so tiny. If it’s too tiny (like 3 or 11), then resizing the table will need to occur more often and it’s such an expensive algorithm to perform. If we set too big in the beginning (like 4111), there’s a good chance that the data that the client has only takes up a tiny number of buckets in the table, thus wasting space.
   2. Because we’re dealing with separate chaining, if the load factor is 1, we will resize our table and rehash all our values
   3. The new size of the array is 2 times the original table size.
3. I use the provided WordInfo’s hashCode method and mod it by the size of the table. I was also considering using other prime numbers besides 31 or 37 but the variations in terms of efficiency/speed did not contradict dramatically. I think the hashCode function in WordInfo does an okay job in terms of hashing because it first hashes the given word with Java’s existing hash function and then multiplying the result by 31 again. Despite having two steps in one, I think it allows for good distribution nevertheless.
4. I think it is also safe to try linear probing. I do not want to implement TextAssociator with quadratic probing because there might be the case where the hash index jumps around endlessly in an infinitely loop because there’s no bucket for its element to be stored at.   
     
   With linear probing, however, there is a possibly that everything might just work without much issues. For it to work, we first need to remove the inner class WordSeparateChain because it no longer is needed. Then when it comes to insertion, a set of new logic will be implemented in TextAssociator’s addNewWord method. Before adding a WordInfo object to one of the respective indexes in the table array, we will check to see if the element and its hash index can be placed at its designated spot. If we cannot do so, we must increment our index to the next index and place the object there instead.   
     
   However, we will also now need to implement a find method so that we can look up where any of our existing elements were stored at. We cannot call the contains function like we did with the ArrayList in separate chaining. The find method simply finds the hash value of the input word and locates the index in which that element was placed.   
     
   When the find method has been implemented, we will use it for both the addNewWord method and the remove method of the TextAssociator. When removing the word, we will first call find to look up its designated spot and then set its value to “”.   
     
   It’s not that easy, however, because each word is associated with other words. For a word to be associated with other words through linear probing, I believe that some sort of data structures must be used to verify that a given word is an association of its parent word. Perhaps, each index in the array now stores not WordSeparateChain but a Map ADT in which the key is the “word” (parent word) and its value is a set of associations. Therefore, when the client calls TextAssociator’s remove method, it will hash the word that the user has given, locate the index through linear probing if necessary, and call remove on the Map ADT.