Zane Koch

Proj 4 Report

Cs32

6/6/18

Description of algorithms and data structures and why I made the choices I did:

The general structure of my solution to this problem is an open hash table. This was built in a class called hashTable. The structure of my hash table specifically is a array of 50,000 buckets, because that is the max number of buckets we were allowed, each of which contains a list of strings. In the class this manifests itself as a data member: list<string> \*\*chain.

When the user inputs a word it must be converted to a hashable quantity, a long long int, between 0 and 50,000. My algorithm to do this is to first removes special characters and then sorts the word alphabetically (ie. Cat would go to act, dog would go to dgo). This ensures that all anagrams of each other will end up in the same bucket. Then the hash function is applied to the word.

The hash function I used is a fairly typical one that iterates over each letters in the word and multiplies the hash so far times some appropriately sized number, I used 131, and then adds the ASCII value of the character. This is then modulated be the number of buckets, 50,000. This creates a nice even distribution with not too many collision.

My other two alorithms are hashTable’s insert and search function. Insert works by removing the nonletters, sorting the word, applying the hash function to the word, and the inserting the original word, less its non-letters, to the index indicated by the hash function. Search works by taking the non-letters out of the input word, sorting it, putting this into the hash function to find the correct index in the hash table and then iterating through each item in the list in that index checking if the string inside the list node is equal to the input string less the non-letters. It then calls back any item this is true for.

Psuedo code of non trivial algorithms:

Insert:

1. Remove the non letters from the input word
2. Return if this means there is nothing left in the word
3. Otherwise sort the word alphabetically
4. Apply the hash function to the word
5. Insert the word that was created after step 1, into the hash table at the index corresponding to the number created in the previous step

Search:

1. Remove the non letters from the input word
2. Sort the word alphabetically
3. Apply the hash function to the word
4. If the bucket in the hash table corresponding to the number to hash function returns is empty, return false
5. If the bucket is not empty check every node in that bucket, and if the word inside of the node, sorted, is equal to the sorted input word, callback the non-sorted word inside the node and return true

Hash function:

1. Take the unput word and iterate over each of its letters
2. For each letters increment an integer by the product of the integer and an appropriately sized, meaningless number and add the ASCII value of the letter
3. Return this value, once the word is over, modulated by the number of buckets.

Bugs, problems:

The biggest problem I had was making my implementation fast enough to get the mull credit for going under 20 milliseconds. The breakthrough that I had to fix this was sorting the words before they were hashed. This forces all of the anagrams of a word to be put into the same bucket which makes it so the inefficient generateNextPermutation function never needs to be called.

Another problem I ran into was that I was forgetting to remove non-letters when the dictionary was being built because I did not realize this was necessary.

My implementation does not have any known bugs currently.