CS 32 – UCLA – Project 4 (Anagrams)

NULL

NULL

NULL

cat

43678

14543678

tca

43678

14543678

fsdf

43678

64543678

doge

47678

2347678

NULL

NULL

….

….

The above diagram represents the basic idea behind my solution to Project 4. It utilizes a hash table of 50,000 buckets. This hash table was implemented using a fixed size array of Node pointers, which essentially linked to custom built linked lists. Each node of the linked list has 4 properties, the value, the generic key representing the bucket number, the super key which is the unique identifier that is the same for all anagrams, and a pointer to the next node.

What my solution basically does, is within the hash function, map the character in the word to its relevant prime number and multiply it to get our super key, which is a private member variable of the DictionaryImp class. The mod of the superkey with the hash table size gives us the index (bucket number). The reason we use prime numbers is because it always ensures unique super keys for every different word but makes it such that every anagram will have the same super key. The generic key is the result we return from the hash function, but it should be noted that because of the limit of the hash table, we get loop arounds in the generic keys as shown in the figure above. So a word that is not an anagram of the word we are looking for may collide in the same bucket, which is why in our lookup we make sure that the super keys also match (ensuring no function acting on our collided element).

**PSEUDOCODE FOR NON-TRIVIAL CODE:**

int hashFunction(word)

{

remove all unnecessary characters and make word all lower case

int prime[] = contains the first 26 prime numbers

set super key to 1

for all the characters in the word

each character is mapped to its relevant prime number in the array and multiplied by the other characters and stored in super key

returns superkey % size of hash table to give the bucket number

}

void insert(word)

{

get the hash key using the hashFunction(word)

create a node pointer to the hash key index of the hash table

if the value is a null pointer

make a new node in that position with (word, superKey)

else

make a new temp node with (word, superKey)

make a node pointer pointing to the beginning of the bucket

while(pointer isnt null)

move pointer forward

make the last pointer point to the temp

}

void lookup(word, callback function)

{

if the callback function is null

return

if the word is empty

return

get the hash key using hashFunction(word)

get the superKey SK

create a pointer pointing to the bucket at the key index

while(pointer isnt null)

{

if(pointers super key is == SK)

{

callback on pointer

move pointer forward

}

else

move pointer forward

}

}

\*There are no bugs or terrible inefficiencies that are obvious from my solution. The general timing of this solution falls in the 6 millisecond range as far as my experience goes.