John Yoo

304787909

My solution for Project utilizes a hash table of 47701 buckets. 47701 buckets were used since 47701 is a prime number, and thus, there would be more even distribution and less collision. I used an open hash table for my implementation since it is more efficient than a linear probe hash table. The hash table was implemented using an array of 47701 nodes; each bucket pointed to a linked list of values. Each node of the linked list has 4 properties, the value, the array ID of the number, the true ID of the number (the ID before it was passed through the hash function) and a pointer to the next node. An unsigned long was chosen to hold the true ID since it can hold a lot of digits (32 bits) and doesn’t store negative numbers.

The solution created a unique ID for each by mapping each character of the word to a unique prime number and then multiplying those numbers together. The unique ID was then passed through the hash function in order to determine its slot in the hash table array. Prime numbers were used because it ensured unique ID for each words and ensured that each anagram had unique IDs. The array ID is the result we return from the hash function, but it should be noted that because of the limit of the hash table, there are possible loop-arounds present. A word that is not an anagram of the word that is being searched for may collide in the same bucket, so the lookup function ensures that the unique ID matches with the words that is being searched for.

**PSUEDOCODE**

int hashFunction(word)

{

remove all nonletter characters and make the remaining letters lower case

int prime[] = contains the first 26 prime numbers.

for all characters contained in the word

each character is mapped to a prime number (each letter in the alphabet maps to a specific prime number in prime[])

all the prime numbers derived are multiplied to create a unique ID for the

word

returns uniqueID % size of hash table to give the bucket number/arrayID

}

void insert(word)

{

get the array ID using the hashFunc(word)

create a node pointer to the array ID index of the hash table

if the index contains a null ptr

create a new node in that position with (word, arrID, uniqueID)

else

create a new temp node with (word, arrID, uniqueID)

make a node pointer pointing to the index

while(pointer isnt null)

move pointer forward

make the pointer’s pointer to the next node temp

}

void lookup(word, callback function)

{

if the callback function is null

return

if the word is empty

return

remove all nonletter characters and make remaining letters lower case

get the arr ID of the word using hashFunc(word)

create a pointer pointing to the bucket at the arr ID index

while(pointer isnt null)

{

if(pointers uniqueID == node in bucket’s uniqueID)

{

callback on pointer

move pointer forward

}

else

move pointer forward

}

}