Greyson Beck

805010132

Project 4 Report

**Descriptions for Data Structures and Algorithms**

For my design, I utilized an open hash table with singly linked lists, as opposed to a closed hash table with linear probing. The general idea is shown below:

struct Node

|  |  |
| --- | --- |
| string value | *holds word being inserted/looked up* |
| Node\* next | *pointer to next node in linked list* |
| int key | *holds key/index of where Node is in hash table* |
| unsigned long uniqueKey | *unique to each anagram, used as identifier* |

Index in Table

|  |
| --- |
| Node Pointer 0 |
| Node Pointer 1 |
| Node Pointer 2 |
| Node Pointer 3 |
| Node Pointer 4 |
| Node Pointer 5 |
| Node Pointer 6 |
| Node Pointer 7 |
| … |

NULL

NULL

NULL

NULL

Node2

Node2

NULL

Node1

NULL

Node1

Node1

Node1

Node1

NULL

NULL

Each pointer in the table above is mapped to by the hash key, which is the ‘key’ variable in the struct Node above. Thus as you can see, my hash table is an array of Node pointers, which point to the address of the first Node in the linked list for that particular key (if any), then that Node points to the 2nd Node, etc. Note: The hash table has an initialized/maximum size of 50000.

Algorithm #1: DictionaryImpl::DictionaryImpl()

For the DictionaryImpl class constructor, I simply used a ‘for’ loop to initialize each pointer in the hashTable array to ‘NULL.’

Algorithm #2: DictionaryImpl::~DictionaryImpl()

For the DictionaryImpl class destructor, I used a ‘for’ loop to iterator through the entire hash table. Then, I go through the linked list at each key and within a ‘while’ loop, use a temporary Node pointer to delete each Node. Once I get to NULL, or if it is NULL to start with, then the ‘while’ loop is broken and the function moves on to the next linked list in the hash table, if any.

Algorithm #3: void DictionaryImpl::insert(string word)

For this function, I call my hash function and set its return value equal to an int, hashkey. Then, I set a Node pointer AND reference (let’s call it location)equal to the pointer located at hashTable[hashkey]. If that is a nullptr, then location is set equal to a dynamically allocated Node. If location is not a nullptr, then I set a temporary pointer equal to location, both of which now point at the first Node in that particular linked list. Then, I set location equal to a dynamically allocated Node, whose ‘next’ pointer is set equal to the temporary pointer. This way, the new Node is inserted at the front, with no need to traverse the list all the way to the end every time a new word needs to be inserted into the dictionary.

Algorithm #4: void DictionaryImpl::lookup(string letters, void callback(string))

First, I check to see if the callback function is equal to nullptr, if so then I return. Otherwise, I proceed to caling ‘removeNonLetters’ on the ‘letters’ argument and then if this is empty, I return. If this is not the case, then I set a local int, hashKey, equal to the return of the hash function(letters). I then initialize a Node pointer (finder) to the hashTable[hashKey], which is the location of where the hashKey mapped ‘letters’ to. I now enter a ‘while’ loop that runs as long as finder is not equal to nullptr. The finder pointer will now search through the linked list, checking to see if any of the Node’s uniqueKey matches the unique\_value of ‘letters’ (this was calculated back when the hash function was called earlier), and if so, the ‘callback’ function is called on that Node’s value.

Algorithm #5: int DictionaryImpl::hashFunction(string& word)

First, the private member variable of the DictionaryImpl class is initialized to 1, which will be useful later in the function. Then, an int array (called primeNums) of size 26 is initialized with the values of the first 26 prime numbers. Next, removeNonLetters is called on the ‘word’ argument to remove any non-letters (as the name suggests). The function then enters a ‘for’ loop, which iterates through the entire ‘word’ string. Within the loop, unique\_value (remember, it is 1 at the start of this iteration) is multiplied by the prime numbers mapped to each char. When we map a char to a certain prime number in the array, we must first subtract ‘a’ so that a value between 0 – 25 is achieved (since we are getting the index, we can’t go past primeNums[25]. Once out of the loop, the function returns the (unique\_value) % (size of hash table), which is what gives us the key.

**Pseudocode for Non-Trivial Functions**

Algorithm #3: void DictionaryImpl::insert(string word)

Initialize int hashKey to hashFunction(word)

Initialize Node\*& ‘location’ to hashTable[hashKey]

if location is equal to nullptr

location is equal to a new Node(word, hashKey, unique\_value)

else

Initialize temp Node pointer to location

Set location equal to a new Node(word, hashKey, unique\_value)

Set location’s ‘next’ pointer equal to temp

Algorithm #4: void DictionaryImpl::lookup(string letters, void callback(string))

if callback is equal to nullptr

return

Call removeNonLetters(letters)

if letters is empty

return

Initialize int hashKey to hashFunction(letters)

Initialize Node pointer ‘finder’ to hashTable[hashKey]

while finder is not equal to nullptr

if finder’s uniqueKey is equal to unique\_value

call callback(finder’s value)

finder is equal to finder’s next

Algorithm #5: int DictionaryImpl::hashFunction(string& word)

Set unique\_value to 1

Initialize int array ‘primeNums’ to first 26 primes, w/ size of 26

Call removeNonLetters(word)

for (int i = 0; i less than word’s length; i++)

multiply unique\_value by primeNums[word[i] – ‘a’]

Initialize int ‘key’ to the (unique\_value) % (size of hash table)

Return ‘key’

**Design Challenges/Bugs**

There was one particular error that caused me a lot of grief as the time came to test my program, and it is now just a laughing matter. In my Node struct, there is a ‘uniqueKey,’ which is used to identify anagrams so when the lookup() function runs, it can easily pinpoint which Nodes in that list have values that are anagrams of the word that was looked up. Unfortunately, I forgot to initialize this variable in neither my member initialization list nor the body of Node’s constructor. Thus, when lookup() was checking for anagrams, it could not find any, including the word itself that was looked up, because all of the Node’s ‘uniqueKey’ variables were left uninitialized by me. I fixed this by simply initializing this variable in the member initialization list by setting it equal to the value passed into the ‘unique\_key’ parameter.