Assignment0: C++ Warmup Exercise: Lookup Table

CS390 Data Networking

Create a lookup table application with the following function signature:

class LookupTable

{

List \*tableArray; // maximum of 10 slots

LookupTable(); // constructor

~LookupTable(); // destructor

Item retrieve(string key);

bool insert(string key, Item value);

bool remove(string key); // removes one instance of the key

int numberUnused(); // returns number unused positions in lookupTable

int numberUsed(); // returns number used positions in lookupTable

int minimumCollisions(); // returns smallest number of collisions in any used lookupTable

int maximumCollisions(): // returns largest number of collisions in any used lookupTable position

void display(); // displays the contents of the table at each position, plus table statistics (numberUnused, numberUsed, minimumCollisiosn, maximumCollisions)

}

class List // Keeps track of the items whose keys end up at the same table position

{

List(); // constructor

~List(); // destructor;

private: **Node \*current; // current node in list of Nodes**

int current; // current location (-1 if empty or invalid location)

void first(); // reset list position to first item

void last(); // reset list position to last item

void makecurrent(int position); // reset list position to position

void prev(); // reset list position to previous item

void next(); // reset list position to next item

Item examine(); // get item at current location

int count; // return number of items in list

void insertBefore(string key, Item value); // insert item before current position

void insertAfter(string key, Item value); // insert item after current position

void remove(); // remove current item, next node becomes current node (unless current node was last node in list in which case next node becomes previous node)

void replace(string key, Item value); // replace current item with item

bool empty(); // true if list is empty

}

struct Item

{

int consonants; // number of consonants in key

int vowels; // number of vowels in key

int count; // number of keys stored at this location

}

**struct Node**

**{**

**// You define the members of this struct**

**}**

Devise a string based function that will convert the key to position in the tableArray. The distribution should spread the keys nicely across the array (so “an” and “anna” in a list of keys that are heavily biased towards “a”s don’t end up in the same position).

It is possible that two or more keys will end up at the same position in the tableArray. When this happens, the key/value pair is added to the list of items at that position in the tableArray.

It is possible that two or more keys will be identical. When this happens, only one Item is stored in for the duplicate keys and the count member is incremented by one. For a single key, the count member reads one.

The List class should be implemented using a dynamic memory linked list. Do not use an array. The List class needs to use Nodes to implement the linked list, however, the List class HIDES this implementation mechanism from the LookupTable class by presenting methods that appear only to manipulate Items.

NOTE: You cannot alter the class signatures of the LookupTable and List classes.

Your program should not have any memory leaks. Make sure that you use the delete operator to remove dynamic memory that you are not using anymore. Make sure that you use the Destructor methods for both the LookupTable and the List classes to cleanup dynamic memory allocated to members of these classes.

NOTE: You can read about memory leak detection schemes here -> https://gcc.gnu.org/onlinedocs/libstdc++/manual/debug.html

I'll be using valgrind to check for memory leaks with this command:

valgrind --leak-check=yes assignment0

You may find the Amazon Web Services links on the course website helpful to get your project off the ground.Use the following test and main program to test your solution.

void test()

{

LookupTable table;

table.insert(“when”, createItem(“when”)}; table.display();

table.insert(“can”, createItem(“can”)); table.display();

table.insert(“sailing”, createItem(“sailing”)); table.display();

table.insert(“weather”, createItem(“weather”)); table.display();

table.insert(“weather”, createItem(“weather”)); table.display();

Item i - table.retrieve(“when”);

cout << “count for when is: “ << i.count << endl; // should be 1

i = table.retrieve(“weather”);

cout << “count for weather is: “ << i.count << endl; // should be 2

table.remove(“when”);

table.remove(“weather”);

i = table.retrieve(“weather”);

cout << “count for weather is: “ << i.count << endl; // should be 1

table.display();

}

void main()

{

LookupTable table;

// READ IN THE LIST OF KEYS HERE…

table.display();

Item i - table.retrieve(“when”);

cout << “count for when is: “ << i.count << endl; // should be 1

i = table.retrieve(“weather”);

cout << “count for weather is: “ << i.count << endl; // should be 2

table.remove(“when”);

table.remove(“weather”);

i = table.retrieve(“weather”);

cout << “count for weather is: “ << i.count << endl; // should be 1

table.display();

}

Here’s a list of keys that you should use to populate the table:

when

can

sailing

color

weather

weather

pats

sailing

past

an

anguish

the

cooking

amplifier

spins

opposite

a

silver

sailing

outcries

weather

past

a

prospective

relationship

the

pedestrian

love

blinks

behind

weather

sailing

barks

beneath

weather

Lemon  
Dark  
Light   
Black  
High   
Low  
Cellphone  
Cat  
Dog  
Penguin  
Japan  
Color  
White  
One  
Brain  
Pills  
Pencil  
Dragon  
Mint  
Chocolate  
Pink  
Green  
Brush  
Handle   
Door  
Knob  
Mask  
Knife  
Speaker  
Wood  
Orient  
Love