Dallas Gable

Southern New Hampshire University

CS-210

April 21, 2024

The main part of my grocery tracker program attempts to exclude external calculation and item querying functions in order to keep the main method clean. The first thing we do is create a new GroceryTracker object, defining the name of our backup file (input 1) and the name of the file we are going to load into memory (input 2). We then use an internal method in our GroceryTracker to load the data from our input file into memory.



| // create a new grocery tracker object  GroceryTracker tracker("frequency.dat", "CS210\_Project\_Three\_Input\_File.txt");   // load the file into memory  tracker.loadInputData(); |
| --- |

After loading all of the necessary data into memory, we then begin our input loop. We start by declaring an integer “choice” in which we will take input from every iteration. From there we print out the display menu, and then create a switch element to determine what action to take. The most important case (I would argue) is in the case of an integer of 1. Here is where we take an item to look up. Upon selection the switch clause defines a string for input, and requests the item to lookup. The code block then uses a public facing method called “getItemFrequency” that returns an integer value of the item frequency. If it is not found the method subsequently defaults to a zero as a return value, indicating nothing could be found.

|  |
| --- |

| int GroceryTracker::getItemFrequency(const string itemName) const {   // use a const iterator to essentially count the frequency  // if items are found, this will return an iterator pointing to the found element that is searched  // if not, it will point to the end of the map  map<string, int>::const\_iterator iterator = this->itemStorage.find(itemName);   // if an item is found  if (iterator != this->itemStorage.end()) {  return iter ator->second; // return the value of the queried item  }   // otherwise return 0  return 0;  } |
| --- |

An input of 2 will cause the code to print item frequency in a string format, 3 will print the histogram graphs, and 4 will save all of the data in memory to our backup file. The switch clause defaults to an invalid input, indicating if an input was not expected.

The main class includes our “GroceryTracker.h” file, where we define the outline of our class. We define 3 private values: itemStorage, backupFilePath, readFilePath. The latter two are defined upon constructor call.

| class GroceryTracker {   private:  std::map<std::string, int> itemStorage;  std::string backupFilePath;  std::string readFilePath;   public:   GroceryTracker(std::string backupPath, std::string readPath);   void loadInputData();  void printItemFrequency();  void printHistogram();  void saveDatabackup();   int getItemFrequency(const std::string itemName) const;    }; |
| --- |

Now, to go over the logic behind all of the child methods. The first declared method “loadInputData” is used to load data from our readFilePath variable. Upon call the method attempts to open an in file stream. If it cannot, it will indicate an error and return, ending execution. It reads each line in the file for itemName and ItemFrequency, and places them into our itemStorage map, with itemName being the key and itemFrequency being the value. It then indicates if there was an eof error, and closes the stream.

The next method is getItemFrequency. It takes an input of itemName, which is the item we are going to query in our dataset. We create an iterator object, and direct it to find the name. We then have an if clause that will only execute if the iterator isn’t at the end of itemStorage, a null place value. If it is not, return the second of the iterator(value) as we are querying for the integer known as frequency of the item. If this if statement doesn’t execute, the method is going to default to returning 0, indicating an item is not in memory.

The next method printItemFrequency is essentially just looping over every value of the map and printing. The format is first (key) + “ “ + second(value) then an endline character.

The next method printHistogram does the same thing as printItemFrequency, but in place of the integer value prints the integer in stars, creating a histogram. We achieve this effect by setting the io out stream’s width to the frequency of an item, then setting fill to the \* character.

The final method saveDatabackup works almost exactly as loadInputData, except for its inputs into our backup file rather than loading. It first opens an out file stream, going through typical validation to make sure the stream is valid. It then loops over each object in the map and saves each line with the format first(key) + “ “ second(value) followed by an endline character.