Emma Zajonc

CSCN 215\_003

Dr. Terri Sipantzi

April 18, 2024

Self-Organizing List Approach Document

I will first create a SelfOrderedList class that is inherited publicly from the SelfOrderedLIstADT.h file. To begin implementing the self-organizing list, I must devise a way to include the three heuristics required by the assignment. I will do this by adding a private char variable “mode” to the SelfOrderedList class. The three values this variable will assume are ‘c’, ‘m’, and ‘t’, standing for Count, Move-To-Front, and Transpose respectively. The list will by default be in Count mode, but I will also add a setMode() function so that the user can change by which heuristic the list is operating. Depending on which mode the list is in, a different reorder function will be called from find. This way, I can implement the solution using one class rather than a separate class for each heuristic. The other private variables I will add are an int variable to keep track of the number of comparisons that the program makes and a LList object of type E to hold the list itself.

There are a few other changes that I need to make to the relevant header files to provide myself with the necessary resources. In link.h, I will add a private variable to the class called freq, which tracks the frequency of each node in the list. I will also create getter and setter methods for frequency in llist.h. I will also change the insert() method in both list.h and llist.h to take a second parameter, an integer called freq, that will also initialize the frequency of a list element whenever it is inserted.

Next I will begin implementing the member functions of SelfOrderedList, starting with find().First, I will move to the start of the list. Then, I will loop through the list, using the length() function of the LList type to bound my for loop. At each node, I will check if the node’s value is equal to it. If it is, I will increment the frequency of the node, call the appropriate reorder function, and return true. I will also increment the variable tracking compares at each node. If it is not found, I will add the element either to the front or the back according to the list’s heuristic.

The reordering functions should be straightforward in that they simply must accomplish what the heuristic requires. The most complicated function will likely be count, as there are more comparisons that must be made in order to insert the list item at the appropriate location according to its frequency.

The add() function will simply call the append() function from the LList class, the getCompares() function will return the compares variable, and the size() function will return the length function of the LList class.

The final portion of the class I must implement is the print function. However, because the print functions in SelfOrderedList are constant, I will have to create entirely new print functions within the LLIst class. As in SelfOrderedList.h, one of these will take an integer parameter and one will not. I will call these print functions from the corresponding SelfOrderedList print functions.