Lab 4 Approach Document

# Self-Organizing Lists

# Assignment Objective

# Implement the three self-organizing list heuristics: Count, Move-to-Front, and Transpose

# Assignment Requirements

Your system should have the ability to

* Implement SelfOrderedListADT: Create a class that inherits from the provided SelfOrderedListADT without changes. Implement functions like find() and reorder() to perform self-organizing list operations, updating frequencies and compares.
* Implement Heuristics: Develop the three specified heuristics - Count, Move-to-front, and Transpose. Ensure that they appropriately adjust the list's order based on the number of accesses, or swapping logic, as required.
* Conduct Tests: Run two tests, one with char types and another using the provided test.txt file. In both tests, track the list's organization, the number of compares, and relevant statistics.

# Approach

* Review all files (documents, code, helps, etc.) provided as part of the assignment
* Select a data structure and implementation that could satisfy this assignment:
  + Use the SelfOrderedListADT abstract data type and the linked-list files I have provided to implement your self-ordered lists. You may incorporate the author’s linked list implementation via inheritance or composition
    - Count – Whenever a record is accessed it may move toward the front of the list if its number of accesses becomes greater than the record(s) in front of it. If the record is not in the list it is added to the back of the list and its count is set to zero.
    - Move-to-front – Whenever a record is accessed it is moved to the front of the list. This heuristic only works well with linked-lists; because, in arrays the cost of shifting all the records down one spot every time you move a record to the front is too expensive.
    - Transpose – whenever the record is accessed swap it with the record immediately in front of it provided it is already in the list. If it is not found in the list and is being added to the list for the first time, it goes to the back of the list.
    - Compare the cost of each heuristic by keeping track of the number of compares required when searching the list.
  + The first test is with char types.
    - Use the add() function to build a list in the following order: A B C D E F G H (do not add ‘I’ here).
    - After we have built that initial list we use the find function to input the following characters: F D F G E G F A D F G E H I (note that ‘I’ is not in the initial list; see what the program does when it searches for an item that is not already in the list). For each heuristic display the order of the final list and the number of compares.
  + The second test is using the test.txt file using the data type string. Do not modify the test text file; going to compare your results with his own and modifying the test file will throw your results off. For this test I want you to do the following for each heuristic:
    - Read into your program the test.txt file adding words to your list using your find() function, and then
    - Print out
      * the total number of words in the list,
      * the total number of compares, and
      * the first 10 words in the list along with their frequency.
  + SelfOrderedListADT Functions:
    - find() – finds a value in the list and, if found, increments the frequency. If not found then find() calls add() to append the value to the end of the list (initial frequency of an item added this way is 0). In either case find() calls your reorder function (see below) to reorder the list in accordance to the heuristic being used and find() increments the number of compares made (whereas add() (see below) does not).
    - add – appends the value to the end of the list without doing any compares or adjusting frequencies.
    - getCompares – returns the total number of compares done by find when searching for values in the list.
    - Size – returns the size of the list.
    - printlist – prints the list in the following format: value-## where “value” is the actual value of the node (either a char or a string) and ## is the frequency of that value. You will need a printlist() and a printlist(n) method because for your char tests you will print the entire list but for the string test I only want the first 10 nodes printed.
    - reorder – you can call this method whatever you want but what I am looking for is a method or methods that reorders your list as appropriate based on the heuristic you are using. This method is typically called by find().
* Test file:
  + Characters A-H “added()” to the list before any calls to find() are made:
    - My starting list structure for all heuristics is: A-0 B-0 C-0 D-0 E-0 F-0 G-0 H-0
  + Count Heuristic:
    - The number of compares is: 61
    - My final list structure with frequencies is: F-4 G-3 D-2 E-2 A-1 H-1 B-0 C-0 I-0
    - My list size is: 9
  + Move-to-front heuristic:
    - The number of compares is: 70
    - My final list structure with frequencies is: I-0 H-1 E-2 G-3 F-4 D-2 A-1 B-0 C-0
    - My list size is: 9
  + Transpose heuristic:
    - The number of compares is: 78
    - My final list structure with frequencies is: A-1 B-0 F-4 D-2 G-3 E-2 H-1 C-0 I-0
    - My list size is: 9

# Build Log

10/30/23 – Today I finished going over all the assignment documentation and other files and have created my approach document up to this point. I created the project folder and imported the required files from Dropbox. Planning to meet with Cameron Kauffman to work together on the lab.

11/3/23 – Met with Cameron Kauffman to get first drafts of main.cpp and SOLMTF.h finished. Worked on SOLCount.h and SOLTranspose.h as well but did not finish.

11/6/23 – Met with Cameron Kauffman to finish first drafts of SOLCount.h & SOLTranspose.h and second drafts for SOLMTF.h & main.cpp. Plan to meet with tutor to fix errors with comparisons

11/8/23 – Me & Cameron met with the tutor to fix complications with the comparisons. Afterwards, we finished the code organization and submitted the project.