Anthony Natale

Summer 2020

Data Structures & Algorithms

# Approach

# At a High Level

We are using implementations of linked lists to add characters, strings, and file input to a self-organizing list. The purpose of this self-organizing list is to store data items and improve the efficiency of linear search as the items are accessed. As they are accessed, the list moves items closer to the beginning of the list based on one of three heuristics:

1. Count
   1. Items with a high number of accesses (recorded by a variable “count” which is incremented each access) are continually repositioned to maintain a descending order from the top of the list.
2. Move-To-Front
   1. When an item is accessed, it is moved to the front of the list and all other items above the accessed item’s previous position, maintaining their order (sans the accessed item), are pushed down one position in the list.
3. Transpose
   1. When an item is accessed, it is swapped with the preceding item, moving its position up in the list by one.

# Process

1. Add all the existing code into a Visual Studio C++ console application project.
2. Create a new private GitHub repository with which to monitor changes.
3. Create SelfOrderedList.h (implementation of the Self Ordered List ADT) and main.cpp
4. Be sure not to edit SelfOrderedListADT.h or test.txt
5. Ensure inheritance is correct:
   1. main inherits the self ordered list implementation
   2. the self ordered list ADT inherits the linked list implementation
   3. the linked list inherits both link.h and list.h
6. Begin implementation.

# Implementation

1. SelfOrderedList.h
   1. Implement all virtual functions from SelfOrderedListADT.h
      1. Constructor
      2. Destructor
      3. find()
         1. Looks through the list for the value
         2. Increments compareCount
         3. If found, increments the frequency of access (accessCount) and returns true
         4. If not found, calls add() to append the value to the end of the list
            1. And sets the new item’s initial frequency to 0
         5. Calls reorder function to reorder the list in accordance to the heuristic being used
      4. add()
         1. Appends the value to the end of the list
         2. No comparisons or frequency adjustments
      5. getCompares()
         1. Returns number of compares accomplished by find()
      6. size()
         1. Returns the size of the list
      7. printlist()
         1. Prints value-## where “value” is the actual value of the node (either a char or a string) and ## is the frequency of that value
         2. For char test
         3. Print the entire list
      8. printList(n)
         1. Prints value-## where “value” is the actual value of the node (either a char or a string) and ## is the frequency of that value
         2. For string test
         3. Only print the first 10 nodes
      9. reorder()
         1. Called by find()
         2. Reorders the list according to each heuristic
   2. Add instance variables (private) and other variables as needed.
2. Main.cpp
   1. Char type
      1. Create the char type selfOrderedList object
      2. Add the predefined characters
      3. Run find queries on another set of predefined characters, including one that isn’t in the list

# Integrity Statements

1. I have not shared the source code in my program with anyone other than my instructor’s approved human sources.
2. I have not used source code obtained from another student, or any other unauthorized source, either modified or unmodified.
3. If any source code or documentation used in my program was obtained from another source, such as a text book or course notes, that has been clearly noted with a proper citation in the comments of my program.
4. I have not knowingly designed this program in such a way as to defeat or interfere with the normal operation of any machine it is graded on or to produce apparently correct results when in fact it does not.