TCSS 342 - Data Structures
Extra Assignment 1 - Self-Organizing Lists

Time Estimate: 2-4 hours

Learning Outcomes

The purpose of this assignment is to:

- Build specialized data structures out of a linked list.
- Build move-to-front based self-organizing lists.
- Build transpose based self-organizing lists.

Overview

Self-organizing lists are lists that move their elements around in response to searches for elements. There are many heuristics that have been devised to try to keep frequently accessed elements near the front of the list. Two common ones are the move-to-front and transpose heuristics.

In the move-to-front heuristic, after an element is accessed in a search it is moved to the front of the list. In the transpose heuristic, after an element is accessed in a search it is swapped with its neighbor to move one space closer to the front of the list.

In this assignment we will build our own specialized move-to-front list and transpose list data structures. You will build these data structures out of the MyLinkedList data structure you have created already.

To complete this you will implement two public classes:

- MTFList<Type>
- TransposeList<Type>

Formal Specifications

MTFList<Type extends Comparable<Type>>
-list : MyLinkedList<Type>
+add(item : Type)
+remove(item : Type) : Type
+find(item : Type) : Type
+size() : int
+isEmpty() : boolean
+toString() : String

Field summary

- `list` - We will use a MyLinkedList as the underlying data structure. Our self-organized list will mostly serve as a wrapper around this underlying data structure.

Method summary

- `add` - Adds the `item` at the front of the list. This method should run in O(1) time.
- `remove` - Removes the `item` if it exists. The item is found using the compareTo() function and only the first occurrence is removed. This method should run in O(n) time.
- `find` - Searches the `list` for the `item` and returns the item if found (and `null` otherwise). If the item is found it is moved to the front of the list. The item is found using the compareTo() function and only the first occurrence is found and returned. This method should run in O(n) time.
- `size` - Returns the size of the list. This method should run in O(1) time.
- `isEmpty` - Returns true if the `size` is 0 and false otherwise. This method should run in O(1) time.
- `toString` - Returns a string that has the contents of the `list` separated by commas and spaces and enclosed in square brackets. This method should run in O(n) time.
    - Example: [1, 2, 3, 4]

TransposeList<Type extends Comparable<Type>>
-list : MyLinkedList<Type>
+add(item : Type) +remove(item : Type) : Type +find(item : Type) : Type +size() : int +isEmpty() : boolean +toString() : String

Field summary

- `list` - We will use a MyLinkedList as the underlying data structure. Our self-organized list will mostly serve as a wrapper around this underlying data structure.

Method summary

- `add` - Adds the `item` at the end of the list. This method should run in O(1) time.
- `remove` - Removes the `item` if it exists. The item is found using the compareTo() function and only the first occurrence is removed. This method should run in O(n) time.

- `find` - Searches the `list` for the `item` and returns the item if found (and `null` otherwise). If the item is found it is swapped with its neighbor towards the front of the list. The item is found using the compareTo() function and only the first occurrence is found and returned. This method should run in O(n) time.
- `size` - Returns the size of the list. This method should run in O(1) time.
- `isEmpty` - Returns true if the `size` is 0 and false otherwise. This method should run in O(1) time.
- `toString` - Returns a string that has the contents of the `list` separated by commas and spaces and enclosed in square brackets. This method should run in O(n) time.
    - Example: [1, 2, 3, 4]

MyLinkedList<Type>
…

…
+addToFront(item : Type)
+moveToFront()
+swapWithPrevious()

Method summary

- `addToFront` - Adds the `item` to the front of the list. This helper function is optional. This method should run in O(1) time.
- `moveToFront` - Moves the `current Node` to the front of the list. This helper function is optional. This method should run in O(1) time.
- `swapWithPrevious` - Swaps the `current Node` with the `previous Node`. This helper function is optional. Any elements after the removed element shuffle down to fill the empty position. This method should run in O(1) time.
    - **Hint**: If you are clever you don't need to move any nodes.

Note: I added these helper functions to MyLinkedList to make it easier to build the wrapper classes. These helper functions are not technically necessary so they are optional for your solution.

Testing

It is important that you test your code to ensure it works on all potential inputs. Please do this in a separate Main class, and without using additional tools (like JUnit). You will not submit your test code. Instead, your data structures will be tested by code developed by the instructor and/or grader. This code will not be provided to you, as you should test your code before submitting it. If your code fails our tests we will let you know which tests it fails so you can find and correct your errors.

Here is some example output from my Linked List test code:

```
MTF List Contents [6, 5, 4, 3, 2, 1]
find(5) 5
MTF List Contents [5, 6, 4, 3, 2, 1]
find(5) 5
MTF List Contents [5, 6, 4, 3, 2, 1]
find(4) 4
MTF List Contents [4, 5, 6, 3, 2, 1]
find(1) 1
MTF List Contents [1, 4, 5, 6, 3, 2]
find(0) null
MTF List Contents [1, 4, 5, 6, 3, 2]
remove(2) 2
MTF List Contents [1, 4, 5, 6, 3]
remove(0) null
MTF List Contents [1, 4, 5, 6, 3]
```

****Important****: You should test your data structures more than this! This is just an example to give you an idea of how to begin testing. This test does not test all the methods, nor does it test all the edge cases for possible method calls. Your testing should do both.

Submission

You will submit a .zip file containing:

- MyLinkedList.java - your linked list class.
- MTFList.java - your move-to-front list class.
- TransposeList.java - your transpose list class.

Grading Rubric

In order to count as complete your submission must:

1. Implement both the move-to-front list and transpose list efficiently and without errors.
2. Follow the API exactly so that your classes will work with any testing code.
3. Pass all tests in the testing code. If you fail any tests, you will be told which test failed.

Reminder: Incomplete assignments can always be corrected and resubmitted. If they are completed within 7 days of the due date they will count as late and after that period they will count as missed. Please review the grading matrix for the number of permitted late and missed assignments.