UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

Department of Computer Science
Faculty of Engineering, Built Environment & IT
University of Pretoria

COS212 - Data structures and algorithms

Practical 2 Specifications: Self-Organizing Lists and Recursion

Release Date: 06-03-2023 at 06:00

Due Date: 10-03-2023 at 23:59

Total Marks: 400

Contents

1 General instructions:

- This assignment should be completed individually, no group effort is allowed.
- Be ready to upload your assignment well before the deadline as no extension will be granted.
- You may **not** import any provided Java library. Importing any library or data structure will result in a mark of zero. You may only make use of 1-dimensional native arrays where applicable. If you require additional data structures, you will have to implement them yourself.
- If your code does not compile, you will be awarded a mark of zero. Only the output of your program will be considered for marks, but your code may be inspected for the presence or absence of certain prescribed features.
- All submissions will be checked for plagiarism.
- Read the entire specification before you start coding.
- You will be afforded five upload opportunities.
- Submissions that result in a compilation failure or a run time error will also receive a mark of zero.

2 Plagiarism

The Department of Computer Science considers plagiarism as a serious offence. Disciplinary action will be taken against students who commit plagiarism. Plagiarism includes copying someone else's work without consent, copying a friend's work (even with consent) and copying material (such as text or program code) from the Internet. Copying will not be tolerated in this course. For a formal definition of plagiarism, the student is referred to http://www.library.up.ac.za/plagiarism/index.htm (from the main page

of the University of Pretoria site, follow the Library quick link, and then choose the Plagiarism option under the Services menu). **If you have any form of question regarding this, please ask one of the lecturers, to avoid any misunderstanding.** Also note that the OOP principle of code re-use does not mean that you should copy and adapt code to suit your solution.

3 Outcomes

The primary goal of this assignment is to implement a set of Self-Organizing Lists and gain experience with recursive and iterative solutions for the same problems.

4 Introduction

Complete the task below. Certain classes have been provided for you alongside this specification in the Student Files folder. Remember to test boundary cases. Submission instructions are given at the end of this document.

For this practical you will implement the following Self-Organizing lists which can be found in the textbook and lecture slides:

- Move-To-Front
- Natural Order
- Transpose
- Counting

You will also be implementing two traversal classes that will implement the same functions, one recursive and the other iterative. This class will be used to traverse the above-mentioned self-organizing lists.

5 Task:

Your task for this practical will be to implement the following class diagram as described in later sections. *Note you can ignore the «Property» tag in the UML diagram. This is part of UML standard.*

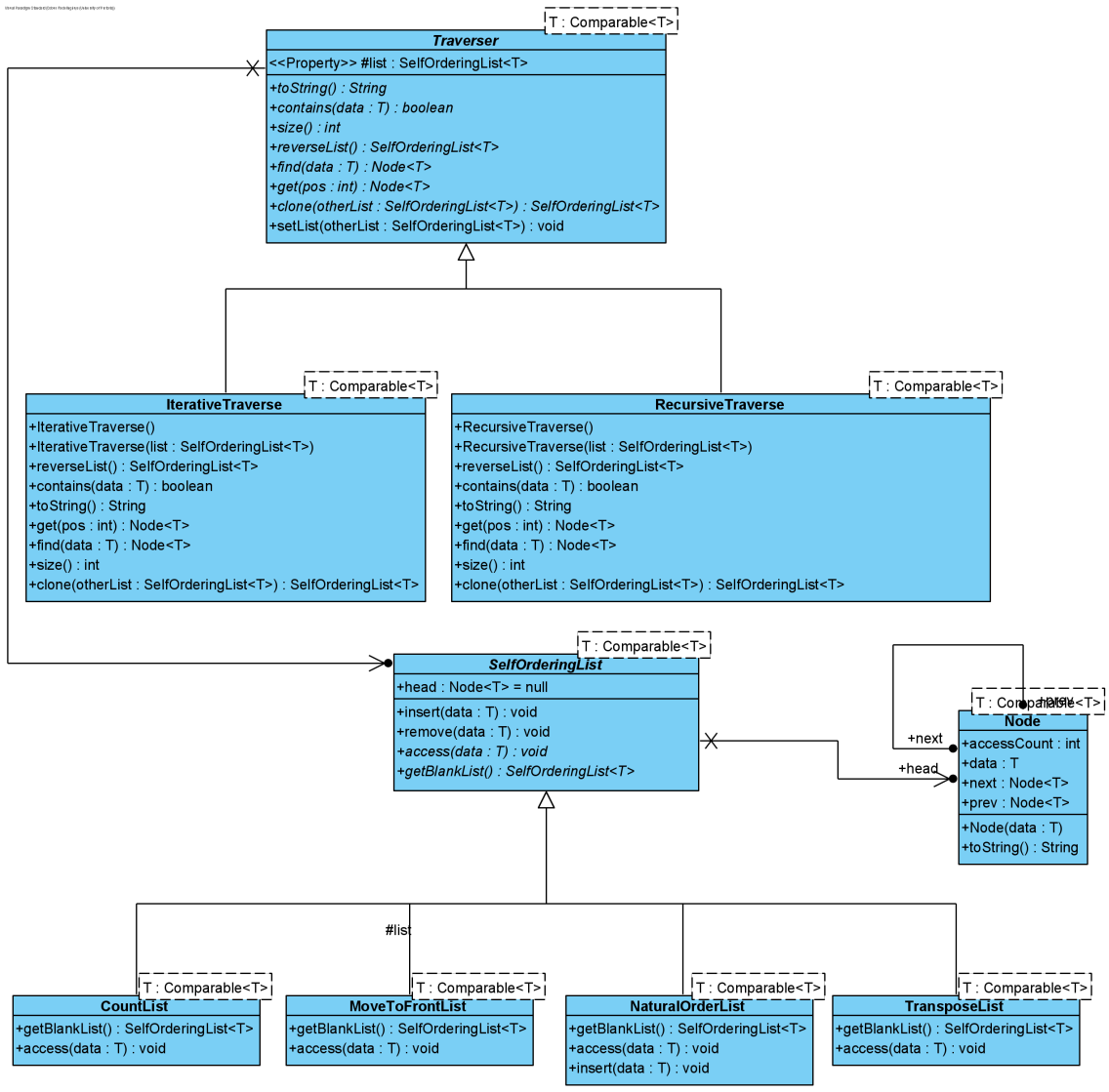

Figure 1: Class diagram

5.1 Node

This will be the class that represents the nodes in the self-organizing lists.

- Members:
  - accessCount: int
    - This is the variable that will keep track of the amount of access counts.
    - Note this variable will **only** be used in the CountingList class.
  - data: T
    - This is the data that is stored by this node.
  - next: Node<T>
    - This is the variable that points to the next node in the list.
  - prev: Node<T>
    - This is the variable that points to the previous node in the list.
- Functions:
  - Node(data: T)
    - This is the constructor for the Node class.
    - Initialize the appropriate member variable with the passed-in parameter.
    - Initialize the next and prev member variables to null.
    - Initialize the accessCount member variable to 0.
  - toString(): String
    - This function has been provided.

5.2 SelfOrderingList

This section will describe the SelfOrderingList class hierarchy. The self-organizing lists should be implemented as **doubly-linked link lists**.

Note on naming convention: Although these lists are known as Self-Organizing Lists the practical uses the term SelfOrderingList for the implementation
The lists may also contain duplicates.

5.2.1 SelfOrderingList

This is the base class for the SelfOrderingList class hierarchy.

- Members:
  - head: Node<T>
    - This is the head node for the list.
    - Should initially be initialized to null.
- insert(data: T): void
  - This function should append a new node populated with the passed-in parameter to the end of the list.
- remove(data: T): void
  - This function should remove the first node that has the same data member variable as the passed-in parameter.
  - If no node contains the passed-in parameter then the list should remain unaltered.
- access(data: T): void
  - This is an abstract function that will be implemented by the derived classes.
  - This function will be used to alter the internal structure of the self-organizing lists.
- getBlankList(): SelfOrderingList<T>
  - This is an abstract function that will be implemented by the derived classes

- This function will be used to return an instantiated object of the derived class.
- Note the list returned from this function will be empty.

5.2.2 CountList

This class inherits from SelfOrderingList and implements the CountList strategy described in the textbook and theory slides with a slight deviation.

- Functions:
  - getBlankList(): SelfOrderingList<T>
    - This function should return a new CountList object.
    - The new CountList object should not contain any nodes.
  - access(data: T): void
    - This function should find the first node that has the same data member as the passed-in parameter and increment the accessCount member variable of the node.
    - The function should then ensure that the nodes are ordered from the highest to the lowest access order (descending).
    - If multiple nodes contain the same access order then they should be ordered in the order of first accessed.
    - The order of nodes that have not been accessed yet should not change.
    - Example:
      Given the list has the following elements in this order: [0,1,2,3,4]
      If they are accessed in the following order: [2, 4, 2, 1]
      The list should be ordered as follows: [2(2), 4(1), 1(1), 0(0), 3(0)]
      where the number between () represents the accessCount value.
    - If the node is the head then the list should remain unchanged.

5.2.3 MoveToFrontList

This class inherits publically from SelfOrderingList and implements the MoveToFront strategy described in the textbook and theory slides.

- Functions:
  - getBlankList(): SelfOrderingList<T>
    - This function should return a new MoveToFrontList object.
    - The new MoveToFrontList object should not contain any nodes.
  - access(data: T): void
    - This function should find the first node with the same data member variable as the passed-in parameter and move it to the front of the list.
    - If the node is the head then the list should remain unchanged.
    - Ensure that the head member variable is updated accordingly.

5.2.4 NaturalOrderList

This class inherits from SelfOrderingList and implements the NaturalOrder strategy described in the textbook and theory slides. The order for this class will be descending (largest to smallest).

- Functions:
  - getBlankList(): SelfOrderingList<T>
    - This function should return a new NaturalOrderList object.
    - The new NaturalOrderList object should not contain any nodes.
  - access(data: T): void
    - This function should be stubbed but should do nothing.
    - Note for marking: When submitting to FitchFork you will see a [0/0] mark for the access tests for this class. This is due to the marking script template currently being used and as

the function has no implementation will not be assessed for marks.

- insert(data: T): void
  - This function should insert a new node into the list with the passed-in data.
  - This function should ensure that the list's descending (largest to smallest) ordering of elements is maintained.
  - If a node with the same data is already contained you may decide if it is inserted before or after the duplicate.

5.2.5 TransposeList

This class inherits from SelfOrderingList and implements the Transpose strategy described in the textbook and theory slides.

- Functions:
  - getBlankList(): SelfOrderingList<T>
    - This function should return a new Transpose object.
    - The new Transpose object should not contain any nodes.
  - access(data: T): void
    - This function should find the first node with the same data member variable as the passed-in parameter and swap it with its previous member.
    - If the node is the head then the list should remain unchanged.
    - Ensure that the head member variable is updated accordingly.

5.3 Traverser

The traverser hierarchy will be used to traverse through the above-mentioned lists. Note none of these traversals should use the access method as to not change the order of the list unless explicitly stated.

5.3.1 Traverser

This is the base class for the Traverser Hierarchy.

- Members:
  - list: SelfOrderingList<T>
    - This is the list that is contained in the Traverser class.
- Functions:
  - toString(): String
    - This is an abstract function.
    - This function will be used to print out the list member variable.
    - It can be assumed that the list will not be null.
    - The resulting string should have the following format:

```
->(node data[node accessCount])->(node data[node
    accessCount])
```

    - Note there should be **no** endline at the end of the resulting string.
    - Example:
      Given that the list contains the following nodes in this order with all of their accessCount values set to 0:
      [0,1,2,3,4]
      then the resulting string should be as follows:
      ->(0[0])->(1[0])->(2[0])->(3[0])->(4[0])
    - *Hint: use the node's toString function for the node's String representation.*
    - Double check that this function is indeed correct before submitting as it is the predominant method of printing out the list.

    - Note there should be no -> at the end of the string. There should however be one at the start of the string. There should also be no end line/newline character at the end of the string.

- contains(data: T): boolean
    - This is an abstract function.
    - This function will be used to determine if the passed-in parameter is contained within the list.
    - If it is contained, return true, else return false.
    - It can be assumed that the list will not be null.
- size(): int
    - This is an abstract function.
    - This function will be used to get the number of nodes in the list.
    - Note if the list contains no nodes the function should return 0.
    - It can be assumed that the list will not be null.
- reverseList(): SelfOrderingList<T>
    - This is an abstract function.
    - This function should return a new list populated with deep copies of the nodes in the member variable list.
    - Note the nodes should be inserted in the reverse order of the original list.
    - Note for the NaturalOrderList: The order of nodes will be the same as the original list as the NaturalOrderList uses insert to the order nodes in the list.
    - *Hint: use the insert function to insert elements.*
    - If the list member variable is null return null.
- find(data: T): Node<T>
    - This is an abstract function.

    - This function should return the first node that has the same data member variable as the passed-in parameter.
    - If there is no node with the passed-in parameter as a data member variable, then the function should return null.
  - get(pos: int): Node<T>
    - This is an abstract function.
    - This function should return the node at the given position assuming the first node starts at 0.
    - If the passed-in parameter is invalid then the function should return null.
  - clone(otherList: SelfOrderingList<T>): SelfOrderingList<T>
    - This is an abstract function.
    - This function should return a deep copy of the passed-in parameter.
    - If the parameter is null, the function should return null.
    - Note do not copy over the accessCount member variable.
  - setList(otherList: SelfOrderingList<T>): void
    - This function should set the list member variable to the passed-in parameter.
    - It can be assumed that the passed-in parameter will not be null.
    - Make use of shallow copy to implement this function.

5.3.2 IterativeTraverse

This class should implement all of the above-mentioned functions as specified. These functions may only use iterative loops to implement the functionality of the functions.

- Functions:
  - IterativeTraverse()

        * This is the default constructor for the IterativeTraverse class.
        * Initialize the list member variable to null.
    - IterativeTraverse(list: SelfOrderingList<T>)
        * This is the parameterized constructor for the IterativeTraverse class.
        * Initialize the list member with the passed-in parameter.
        * Make use of deep copy in your implementation.

5.3.3 RecursiveTraverse

This class should implement all of the above-mentioned functions as specified. These functions may only use recursion to implement the functionality of the functions.

- Functions:
    - RecursiveTraverse()
        * This is the default constructor for the RecursiveTraverse class.
        * Initialize the list member variable to null.
    - RecursiveTraverse(list: SelfOrderingList<T>)
        * This is the parameterized constructor for the RecursiveTraverse class.
        * Initialize the list member with the passed-in parameter.
        * Make use of deep copy in your implementation.

6 Helper functions and variables

You are allowed to add helper functions to any of the classes. You are strongly encouraged to add your own helper recursive functions to the RecursiveTraverse class.

7 Restrictions for the RecursiveTraverse Class

As a reminder, you may only use recursion for the implementation of the RecursiveTraverse class. The following keywords are thus illegal to use in this class:

- for
- while
- do

Usage of these commands in the RecursiveTraverse class will yield a mark of 0 for the entire practical.
The usage of the IterativeTraverse class is also prohibited in the RecursiveTraverse class and will also result in a mark of 0 for the entire practical.

8 Submission

You need to submit your source files on the FitchFork website (https://ff.cs.up.ac.za/). All methods need to be implemented (or at least stubbed) before submission. The following java files should at least be in a zip archive named uXXXXXXXX.zip where XXXXXXXX is your student number:

- Traverser.java
- IterativeTraverse.java
- RecursiveTraverse.java
- SelfOrderingList.java
- CountList.java
- MoveToFrontList.java
- NaturalOrderList.java

- TransposeList.java
- Node.java

You may add any other custom classes that you created. Your code should be able to be compiled with the following command:

```
make *.java
```

and run with the following command:

```
java App
```

You have 5 submissions and your best mark will be your final mark. Upload your archive to the Practical 2 slot on the FitchFork website. Submit your work before the deadline. **No late submissions will be accepted!**