ENSF 594 – Principles of Software Development II
Summer 2022

Project: module 1 handout

The objectives of this module:

1. Create a library for linear data structures
    a. Linked Lists:
        i. Singly
        ii. Doubly
        iii. Circular singly
        iv. Circular doubly
    b. Stack
    c. Queue

Step 1: pre-planning

Design and model the structure of your library for the linear data structure

- Separate base classes into a library of their own (Node classes)
  - Make sure to use inheritance and interfaces when it makes sense
- Structure your data structure class to reduce redundant coding
  - Make sure to use inheritance and interfaces when it makes sense

Step 2: implementing

Your initial implementation of the required classes does not have to be part of a package or included in a maven project just yet (although it is recommended if you know how to structure a library folder with multi levels to have sub libraries)

The initial set of classes to implement with their base functionalities are as follows:

1. singlyLL:

This is a Singly Linked List Data structure that will implement the following:

- Uses a head object of the base class Node (to be implemented as part of the base classes mentioned previously) and a tail object to keep track of the end of the list

- Has an integer member variable to keep track of the size of the List (update when necessary)

- 2 constructors:
  - Default constructor with no arguments that creates a null head object
  - Overload constructor with a Node object argument to use as head
  - You may combine both using default arguments if you prefer to

- InsertHead(node)
  - Inserts node object at head of the list

- InsertTail(node)
  - Inserts node object at the tail of the list

- Insert(node,position)
  - Inserts node object in the specified position
    - Ex. Insert(node ,5) → inserts node to 5th position in list

- SortedInsert(node)
  - Inserts node object in its proper position in a sorted list
  - Must check for list sort validity
    - If list is found to be out of order, it must call the sort function first before inserting
    - Note that you should only execute sort if the list is found to be out of order

        to avoid slowing down the insertion by executing sorting every time you insert
    - Might need to implement a helper function isSorted(), or find a creative way to know if the list is sorted

- Search(node)
  - Looks up node in the list
    - If found it returns the object
    - Otherwise returns null

- DeleteHead()
  - Delete head node

- DeleteTail()
  - Delete tail node

- Delete(node)
  - Deletes the node if found in the list

- Sort()
  - Applies insertion sort to the list
  - The insertion part will start from the head unlike the usual insertion sort algorithm
    - Instead of tracking back the list
  - Note that the sort method and SortedInsert can use each other to efficiently reduce code redundancy (not mandatory)

- Clear()
  - Deletes the whole list

- Print()
  - Prints the list information on the screen, this includes
    - List length
    - Sorted status
    - List content
    - Make sure to show information with relevant print statements to be readable by the user

2. doublyLL:

This is a doubly Linked List data structure that extends the singlyLL and will add some extra functionality. Modifications and overriding of methods from the singlyLL class might be needed, while in some cases you can use the implemented functions from super class.

NOTE: you can opt to not extend SLL class if it end up requiring rewriting a lot of functions anyway since you need to account for added previous pointer

Some of the modifications are:

- Uses a head and tail objects of the base class DNode (to be implemented as part of the base classes mentioned previously)

- The constructors will be updated
  - Default constructor needs to account for tail
  - Constructor overload with one node initializes the list with head and tail pointing to the same node

- The remaining functionalities from singlyLL will practically be modified to account for the previous reference in the DNode

3. singlyCLL:

This is a Circular Singly Linked List data structure that extends the singlyLL and will add some extra functionality. Modifications and overriding/overloading of methods from the singlyLL class might be needed, while in some cases you can use the implemented functions from super class.

NOTE: exploit the size variable that you have in the SLL class. The use of this variable for looping will minimize the requirement to rewrite codes for many functions

Some of the modifications are:

- The constructors will be updated
  - Constructor overload with one node initializes the list with head and loops it to reference itself

- The remaining functionalities from singlyLL will practically be modified to account for the closed loop / self-referencing end of the list if needed

4. doublyCLL:

This is a Circular Doubly Linked List data structure that extends the doublyLL and will add some extra functionality. Modifications and overriding/overloading of methods from the doublyLL class might be needed, while in some cases you can use the implemented functions from super class.

NOTE: since the doubly LL is expected to also have the size variable, you should do the same as you did with the circular singly linked list to maximize code reusability

Some of the modifications are:

- Uses a head and tail objects of the base class DNode (to be implemented as part of the base classes mentioned previously)

- The constructors will be updated
    - Default constructor
    - Constructor overload with one node initializes the list with head and tail pointing to the same node and self referencing

- The remaining functionalities from doublyLL will practically be modified to account for the closed loop / self-referencing end of the list if needed

5. LLStack:

This is a Stack based on Singly Linked List data structure and extends the singlyLL. Modifications and overriding/overloading of methods from the singlyLL class are needed, while in some cases you can use the implemented functions from super class.

NOTE 1: singlyLL has a lot of functionalities that are detrimental to stack behavior but will be inherited nonetheless. To avoid the miss use you need to override any method that does not apply to stacks from singlyLL with an empty body method.

- For example: InsertTail(node) needs to be redefined in this class with empty body

NOTE 2: while the functionality we are looking for is already implemented in the singlyLL, the naming of the methods is not what we expect for a stack. Make sure to define function wrappers with proper naming conventions to invoke the functionality from the super class

- For example: define push(node) which calls super.InsertHead(node)

6. LLQueue:

This is a Queue based on Singly Linked List data structure and extends the singlyLL. Modifications and overriding/overloading of methods from the singlyLL class are needed, while in some cases you can use the implemented functions from super class.

NOTE: singlyLL has a lot of functionalities that are detrimental to queue behavior but will be inherited nonetheless. To avoid the miss use you need to override any method that does not apply to stacks from singlyLL with an empty body method.

- For example: InsertHead(node) needs to be redefined in this class with empty body

NOTE 2: while the functionality we are looking for is already implemented in the singlyLL, the naming of the methods is not what we expect for a stack. Make sure to define function wrappers with proper naming conventions to invoke the functionality from the super class

- For example: define enque(node) which calls super.InsertTail(node)

Note: you are allowed to create only one Node class that can work for all linear structures (that will be the DNode) and use it for unidirectional linear structures without relying on the previous pointer

Library content after completion of this module:

- myLib
    - datastructures
        - nodes
            - SNode.java (optional)
            - DNode.java
        - Linear
            - SLL.java
            - DLL.java
            - CSLL.java → extends SLL
            - CDLL.java → extends DLL
            - StackLL.java → extends SLL
            - QueueLL.java → extends SLL
        - trees
        - heap
    - graphalgo