CS 342 Project #1 Spring2022

Generic Lists in Java

Due Date: Sunday, 2/20 @11:59pm

Description:

In this project you will implement your own versions of the stack and queue data structures. Although the Java API provides built-in support for them, you will write your own to practice the constructs and language details we have seen in class. That means you are NOT allowed to use any pre existing libraries or classes for this assignment.

Essentially, you are writing your own data structure library that could be used by others in the same way that one can use ArrayList<>. Your library provides data structures as classes, the method calls expected with those data structures and the definition for an iterator so that the clients of your libraries can iterate through the lists in the same way as most generic data structures in Java.

In this simplified version, each data structure is a singly linked list. The stack is LIFO (last in first out) while the queue is FIFO (first in first out). Your implementation must be generic, like ArrayList, as to allow for different types when each data structure object is instantiated.

You will also implement the Iterator design pattern; allowing users access to multiple custom Iterators for your data structures.

Implementation Details:

You will download and use the Maven project template GLMaven_Project1_S2022 from Blackboard. You will find a file called GLProject.java in the src folder. This class contains the main method. You will create a new file, inside of src/main/java, for each outer class. In comments at the top of the file GLProject.java, please include your name and netid and university email as well as a brief description of your project.

DO NOT add any folders or change the structure of the project in anyway. DO NOT alter the pom.xml file.

You can now run main with the Maven command exec:java. You can see where this was added if you look at the POM file. Make sure to compile your project first before running.

In it's own file, create a generic abstract class called **GenericList<T>**:

It should implement the Java **Iterable<T>** interface: this will allow clients of your LL to use a forEach loop to iterate.

This class will contain only two data fields:
Node<T> head (this is the head of the list and should be **private**).
int length (the length of the list and should be **private**)

This class should include the following public methods:
print(): prints the items of the list, one value per line. If the list is empty, print "Empty List".
add(T data): adds the value to the list. This method is **abstract** since the implementation depends on what the data structure is.
public T delete(): returns the first value of the list and deletes the node. If the list is empty, return null.

public ArrayList<T> dumpList(): this method stores and returns all values currently in the list into an ArrayList and returns it. At the end of this method, your list should be empty.

public T get(int index): returns the value at the specified index or null if the index is out of bounds.

public T set(int index, T element) : replace the element at specified position in the list with the specified element and return the element previously at the specified position. Return null if index is out of bounds

getLength() setLength() getHead() setHead(), these are getters/setters for private data members head and length.

public ListIterator<T> listIterator(int index) :returns a list-iterator of the elements of this list starting at the specified position.

public Iterator<T> descendingIterator( ) :returns an iterator over the elements of the list in reverse order(tail to head)

*** you will also have to implement any abstract methods from the Iterable<T> interface. You do not need to worry about the default methods***

This class should also define a generic inner class **Node<T>**: It will include two fields:

T data and **Node<T> next**;

This inner class will also provide a single arg constructor:

public Node(T val) // sets data equal to val

This inner class is to be used to create nodes, in your linked list class

Create two more classes, each in a separate file, **GenericQueue<T>** and **GenericStack<T>**. They both should inherit from **GenericList<T>**.

Each one of these classes will have one additional data member:

Node<T> tail; This is a traditional reference to the tail of the list.

And add the following method:

public T removeTail() :retrieves and removes the tail of the list using the tail data member.

The constructors for each class will take one parameter. That parameter will be a value that will go in the first node of the list encapsulated by each instance of the class. Each constructor should initialize the linked list **head**, with the value passed in by the constructor and set the head and tail data members. Each class should also implement the method **add(T data)**, **GenericQueue** will add to the back of the list while **GenericStack** will add to the front. You should use the head and tail data members to accomplish this. Each class must also keep track of the length of it's list using the **length** data field defined in the abstract class.

GenericQueue will have the methods **enqueue(T data)** and **public T dequeue()** which will call the methods **add(T data)** and **delete()** respectively. Enqueue and dequeue merely call **add(T data)** and **delete().** The reason for this is that a user would expect these calls to be implemented in each of those data structures. You will do the same with **GenericStack**

GenericStack will have the methods **push(T data)** and **public T pop()** which will call the methods **add(T data)** and **delete()** respectively.

Once implemented, you should be able to create instances of both GenericQueue and GenericStack in main with most primitive wrapper classes. You should be able to add and delete nodes to each data structure instance as well as print out the entire list and check for the length of the list. You must follow this implementation: meaning you can not add any additional data fields or classes. You may add getters/setters as need be.

Implementing Iterator Design Pattern:

You must also create a class to contain logic for iterating through your data structure (head to tail). Call this class GLLIterator (it should be in its own file). GLLIterator should be a generic class since it provides the logic to iterate through a generic linked list.

It should implement the java interface **Iterator<E>** (java.util.Iterator). You will have to implement two inherited methods: public boolean hasNext(), checks to see if there is another value in the data structure and returns true or false, and public I next(), returns the current value in the data structure and advances to the next item. This is the class that will be returned when the iterator() method is called from the **Iterable<T>** interface.

You will create another class ReverseGLLIterator (in its own file) which will be identical to the GLLIterator class except that the hasNext() and next() methods will have logic to iterate from the list in reverse (tail to head). This is the class that will be returned when the descendingIterator() method is called in the GenericList class.

You will also need to create a class to contain the logic for the list-iterator, call it GLListIterator (in its own file). It should implement the Java interface ListIterator<E> (java.util.ListIterator). This is the class that will be returned when the listIterator(int index) method is called in the GenericList class. You will need to implement all the abstract methods inherited from the interface.

You do not need to implement optional/default operations in the ListIterator or Iterator interfaces. Those are:

In ListIterator:

add(E e)

remove()

set()

In Iterator:

remove()

forEachRemaining()

You are expected to fully comment your code and use good coding practices.

Test Cases:

You must write and include test cases for your GenericQueue and GenericStack classes as well as all three iterators. These test cases should be split between two files: GQTest.java and GSTest.java. At a minimum, you must write 1 test case per method, test that you can implement a forEach loop and fully test the constructors for the GenericQueue and GenericStack and Node classes. Also test that the descendingIterator performs as expected as well as the methods in the ListIterator. You should be writing these at the same time you write methods in your project.

Electronic Submission:

Zip the Maven project GLMaven_Project1_S2022 and name it with your netid + Project1: for example, I would have a submission called mhalle5Project1.zip, and submit it to the link on Blackboard course website.

Assignment Details:

Late work **is accepted**. You may submit your code up to 24 hours late for a 10%

penalty. Anything later than 24 hours will not be graded and result in a zero.

We will test all projects on the command line using Maven 3.6.3. You may develop in any IDE you chose but make sure your project can be run on the command line using Maven commands. Any project that does not run will result in a zero. If you are unsure about using Maven, come see your TA or Professor.

Unless stated otherwise, all work submitted for grading *must* be done individually. While we encourage you to talk to your peers and learn from them, this interaction must be superficial with regards to all work submitted for grading. This means you *cannot* work in teams, you cannot work side-by-side, you cannot submit someone else's work (partial or complete) as your own. The University's policy is available here:

https://dos.uic.edu/conductforstudents.shtml.

In particular, note that you are guilty of academic dishonesty if you extend or receive any kind of unauthorized assistance. Absolutely no transfer of program code between students is permitted (paper or electronic), and you may not solicit code from family, friends, or online forums. Other examples of academic dishonesty include emailing your program to another student, copying-pasting code from the internet, working in a group on a homework assignment, and allowing a tutor, TA, or another individual to write an answer for you. It is also considered academic dishonesty if you click someone else's iClicker with the intent of answering for that student, whether for a quiz, exam, or class participation. Academic dishonesty is unacceptable, and penalties range from a letter grade drop to expulsion from the university; cases are handled via the official student conduct process described at https://dos.uic.edu/conductforstudents.shtml.