CS 342 **Project #1** **Spring2023**

Generic Lists in Java

Due Date: Sunday, 02/19 @11:59pm

Description:

In this project you will implement your own versions of the HashMap 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 (like Java HashMap or LinkedList) 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 and map in the same way as most generic data structures in Java.

The Queue data structure will be a singly linked list, FIFO (first in first out). The HashMap will use your Queue data structure to handle collisions. Your implementations 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_SP2023 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 **(t**his 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(): This method will also be **abstract** since the implementation depends on the type of data structure and how the user wants to delete nodes (from the front, back,…etc.). If the list is empty, it should return null.

public ArrayList<T> dumpList(): this method stores and returns all values currently in the list into an ArrayList and returns it.

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 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 three fields:

T data

Int code

Node<T> next;

Data will contain the data being stored in the list. **Code** will be an optional field that may be used or not. It is common to see classes include data fields that may or may not be used by the eventual client. **Next** will be the reference to the next node in the list.

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

Create the class GenericQueue<T> in a separate file. It should inherit from GenericList<T>:

You should add one additional data member:

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

The constructors for this 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. This class should also implement the method **add(T data)**, **GenericQueue** will add to the back of the list. This class should also implement the **delete()** method. It will return the value of and delete the last node in the list. You should use the head and tail data members to accomplish this. This class must also keep track of the length of it's list using the **length** data field defined in the abstract class. It should also overload this with a second add method, **add(T data, int code)**. It will do the same thing as **add(T data)** but also set the **code** data member to the value passed in

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.

Once implemented, you should be able to create instances of GenericQueue 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.

Create the class MyHashMap<T> in a separate file. It should also implement the Iterable<T> interface:

This will be your implementation of a traditional, somewhat simplified, Java HashMap. You will use the Java.util data structure **ArrayList<T>** in conjunction with the **GenericQueue** class you have already created to accomplish this. Your **MyHashMap<T>** class will have a single data member (you can and will need to add more):

An **ArrayList** of **GenericQueues** called **map**

In a HashMap, it is possible that multiple keys might hash to the same index in the data structure, this is where the **GenericQueue** comes in. If there is a collision, you will simply add another node to the linked list data structure at that index.

This class will have a single constructor:

MyHashMap(String key, T value): it will initialize the ArrayList map to 10 and add the first key/value pair into the ArrayList **map** using the method **put(String key, T value)**.

Public void put(String key, T value): this method will take a key value pair and do the following:

Create a hash code and hash value using the key passed into the method.

Example of hash code and hash value:

You will use the built in String method, hashCode() to generate the hash code:

If I used the string “Cozmo” as my key with the method hashCode() it would return a value:

65303440 (this is the value you put in the nodes **code** data field.)

To use that number to find an index in the ArrayList of size 10, I would do key.hashCode()&9 which would return the value 0. That is the hash value and the index where I would put that key.

If I used the string “omzoC”, the hash code would be 105878800 and the hash value would also be 0. This would be a collision.

Notice that the hash codes are usually unique but the hash values are the same at index 0. The hash values tell you what index to look at, the hash codes are used to identify individual strings at the same index.

Use the hash value to check and see if there is a GenericQueue at that index in the ArrayList map already.

If not, create a new **GenericQueue** using the values passed into this method and add it to the ArrayList **map** at the index indicated by the hash value. Make sure to set the **code** field in the **node** to the hash code you derive from the key.

If there is already a **GenericQueue** at that index in the ArrayList map, use that instance and the **GenericQueue** method **add(T data, int code)** to append this as another node in the GenericQueue at that index.

You will also implement the following:

Public boolean contains(String key): this method will check to see if the given key exists in the HashMap and return true if yes and false if no.

Public T get(String key): this method will return the value at the given key or return null if it does not exist.

Public int size(): returns the number of key-value mappings in the map.

Public boolean isEmpty(): returns true if this map contains no key-value mappings.

Public T replace(String key, T value): replaces the entry for the specified key only if it is currently mapped to some value.

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.

Finally, you will create a third class, **HMIterator** (in its own file) which will also implement the java interface Iterator. It will also be a generic class. You will implement the two inherited methods, **public boolean hasNext()** and **public V next()**. They will work in a similar fashion to the others in **GLLIterator** and **ReverseGLLIterator** except they will know how to iterate through your hash map. This is the class that will be returned when the **iterator()** method is called in your **MYHashMap class**.

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

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 MyHashMap classes as well as all three iterators. These test cases should be split between two files: GQTest.java and HMTest.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 MyHashMap and Node classes. Also test that the descendingIterator performs as expected. You should be writing these at the same time you write methods in your project.

CS 342 **Project #1** **Spring2023**

Electronic Submission:

Zip the Maven project GLMaven_Project1_SP2023 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.