CIS 121—Data Structures and Algorithms with Java—Spring 2018

Homework 0—Thursday, January 11

Due Monday, January 15 at 11:59 PM

4 Required Problems (60 points), and Code Review (15 points)

DO NOT modify methods or method headers that were already provided to you.

DO NOT add public or protected methods.

DO introduce helper methods that are **package-private** (i.e. do not have a privacy modifier) as you deem necessary.

Logistics

Notice that there is no package statment at the top of each file. **Do not** add a package statement. This will cause your assignment to compile on Gradescope. Instead, please use the default package.

Ensure that **ALL** of your files that you submit are within this default package.

If you're confused about how to setup your project environment, please watch the video we have posted on Piazza which covers how to set up your project.

You have unlimited submissions for this homework. Upon submission, our infrastructure will compile your code and run some basic test cases against the interface. These are meant to test that your code compiles (e.g. you didn't forget to add a class, etc.) when we run real tests. They should not be used to test your own code's correctness, since the functionality they test is very basic.

The compilation tests will be worth 1 point of the final score.

Please make sure you have Java 8 installed. Many of the projects this semester require Java 8 for some parts of the provided code and there are some neat functional programming additions to Java for those of you coming from CIS 120. It's nice to have it set up during the first assignment instead of later. See the pinned piazza posts for tips on setting up your environment correctly. A brief overview of the necessary components are outlined below.

To set up Java 8 and Eclipse, please refer to the video posted on Piazza (@12)

Another tool we will be using is code coverage, which checks what percentage of your implementation is being covered by your test cases. It may come bundled with your Eclipse download, but in case it doesn't, you can download the Eclipse extension Eclemma. Instructions for how to do this are again detailed in the post @12.

Motivation—Reviewing 120 Concepts

In order to understand recursion, you must first understand recursion...

We hope you had a great winter break! This homework is intended to help you review concepts so you can jump right into the CIS 121 material and be successful.

Don't be fooled by the homework name "Finger Exercises." This is a significant assignment that is by no means conceptually trivial.

Learning Objectives

- Ensure you have the proper setup for Java development
- Review key CIS 120 concepts:
  - Recursion
  - Basic data structures: Linked Lists and BSTs
  - Code design with interfaces, object-oriented design
  - Debugging in your IDE

Overview

CIS 121 is the combination of the mathematical analysis of CIS 160 and the programming knowledge of CIS 120. In this homework we aim to give you practice in both by walking you through some Java review of interfaces, and letting you practice developing your own algorithms.

Part Zero: Read and understand the requirements

Files to submit: None

Understanding the problem is the first step to solving it. Take time to read through this writeup and the Javadocs for the provided interfaces before starting. Familiarize yourself with Java syntax and the style conventions outlined in the course website. Go over the requirements for each class and plan things out before you start writing code.

This assignment will give you the opportunity to learn how to write proper unit tests. Be sure to read the testing guide for some pointers on what level of testing we expect from you. In addition, we have more resources on testing on Piazza.

There are a few big changes from CIS 120. First, the code you write will allow for more freedom than 120. You will be given interfaces to implement, rather than filling out stub files, which means that you have more freedom over implementation. You will have to make conscious code decisions and eventually defend the efficiency of your code in future assignments. Second, you must be confident in your unit tests. You will not get the instant feedback from tests like CIS 120 because in industry, you don't know immediately that your code will work unless you've written unit tests beforehand. Unit testing is a critical skill that allows you to be confident in the accuracy of your code, and building good practices now will carry a long way through your computer science careers.

Part One: Defining Object Behavior (10 points)

Files to submit: `StudentImpl.java`, `StudentImplTest.java`, `StudentFactoryImpl.java`

Implement the interface defined in `Student.java` with your own concrete implementation called `StudentImpl`. Students are uniquely identified by their ids. This means that two students are equal if and only if their ids are the same, regardless of what their String name is. Recall that when you change the equality behavior of a class in Java, you must also change its hash code behavior.

This means that you will be overriding the default Java Object methods `equals` and `hashCode`. As a brief recap of what a hash code is, it is an integer that determines which bucket an object is placed in when it is hashed into a hash table. The actual value that is returned is not too important: the rule is that Objects that are equal to each other must be hashed to the same bucket; that is, the integer that `hashCode` returns must be the same for the all equal objects. Objects that are not equal to each other should usually return different values. Also, you may **not** use the auto-generated versions of these methods. Try to think of which fields should be used to correctly implement these methods.

When you implement `Student`, you should ensure that your implementation of `Student` is compatible with other potential implementations of `Student`. For example, one `Student` implementation could still be equal to another type of `Student` implementation if they have the same id.

We also ask that you implement the factory interface. It is critical that you do this because we will use your factory to test your `Student` class. As you should know from 120, you cannot define constructor headers in your interface, which means that your implementing class has absolute freedom in its constructors. In order to test your code, we use this factory interface to have an established way to create an instance of your `Student` class.

Essentially, the factory class will serve a wrapper for your `Student` constructor.

Part Two: Recursive Data Structures (25 Points)

Files to submit: `LinkedListNodeImpl.java`, `LinkedListNodeImplTest.java`

Implement the provided `LinkedListNode` interface.

In this section, you will be implementing a classic recursive data structure: the LinkedList. To help you, we have provided an implementation of another recursive data structure, the binary search tree, in `BSTNodeImpl`. We have also included sample unit tests in `BSTNodeImplTest`.

BSTNode

Recall the binary search tree invariant: all nodes to the left of a node have values smaller than the node's value, and all nodes to the right of a node have values larger than the node's value (this implementation does not support duplicates). Each node is itself the root of a binary search tree.

When you read a definition that is recursive, you should immediately think of using recursion to accomplish your goals. Take time to look over our implementations of the BST methods. Similarly to LinkedList, all operations are called on the root of the tree, and an empty tree is represented by a null BSTNode.

This is also a good time to refresh your knowledge of tree traversals. An inorder traversal is the most intuitive: for a BST, it will return the nodes in sorted order, as it traverses the left subtree, then returns the parent, then traverses the right subtree. A preorder traversal considers a parent's value to be ahead of its childrens' values. A postorder traversal considers the childrens' values to be ahead of the parent's value. We have only implemented an inorder traversal (that utilizes your `LinkedListNodeImpl`!), but both preorder and postorder will be important later in the course, so it's good to know them.

Also take the time to review the unit tests we have provided in `BSTNodeImplTest`, and use them to guide your own testing.

LinkedListNode

Very Important Note: In order to pass our tests you must include a constructor in `LinkedListNodeImpl` that takes in the value of the head node as an argument. The constructor header is given in the stub file.

You will be implementing a recursive version of a LinkedList by completing an implementation of a LinkedListNode. You **must** use recursion to complete this assignment. We have also provided to you a wrapper class for the LinkedListNode called "LinkedList", which you will use on a later part of this assignment.

Note: Your LinkedListNodeImpl must be compatible with other classes that implement the LinkedListNode interface. This means that you cannot assume that other elements in your list are LinkedListNodeImpls, and thus the only operations you can perform on them is call interface-defined methods. You will be harshly penalized if you cast other LinkedListNodes to LinkedListN-odeImpls in your implementation.

You will also be implementing a LinkedListIterator. Recall that a Java Iterator is an interface with methods `next` and `hasNext`. You do not have to implement `remove` for your iterator. We use iterators because it allows for 'lazy' evaluation: that is, we don't have to figure out what is next in a sequence until we actually call `next`, which may save us time as well as space in certain applications. You must create a private inner class that implements the iterator interface. You do not have to worry about the list changing during iteration. Also, you will not receive credit if you precompute everything to iterate.

Whenever possible, your methods should be **in-place**, meaning that you shouldn't allocate new

Objects whenever possible. For example, a method like `truncate()` wouldn't need to use the keyword `new`.

Part Three: Algorithmic Interview Questions (25 points)

Files to submit: `InterviewQuestionsImpl.java`, `InterviewQuestionsImplTest.java`

In this section, you will be implementing three interview questions. We strongly encourage you to think about the questions on your own (it is also against course policy to consult other sources). They are tough problems, and not only will struggling with them provide you a sense of accomplishment once you figure them out, it will also help prepare you for job interviews!

Binary Tree Inversion

Given a binary tree, you must invert it. This means that for each node in the tree, its left subtree will now become its right subtree, and vice versa. For example, if 5's left child is 2 and its right child is 1, then in the inverted tree, 5's left child will be 1 and its right child will be 2.

The interface documentation has a good example drawn out so you can see what's required. You may test your implementation using the provided `BSTNodeImpl` class.

Hint: This problem has an elegant recursive solution.

Middle Element

Given a linked list, you must find the element that is in the 'middle' of the list; that is, if the list contains n elements, you must find the $\lceil \frac{n}{2} \rceil$ element. This problem highlights some disadvantages of linked lists versus a fixed-size array. In an array, you can easily access the middle element by indexing into the appropriate entry. However, you must traverse a linked list in order to access an arbitrary element.

For this problem, we are imposing the additional constraint that you must find the middle element without knowing the size of the LinkedList ahead of time. What does this mean? It means that you cannot call `size`, and if you are storing the size of the list as a field on your node object, you cannot use that. You must find the middle element via a single "pass" through your list; that is, if at any point you determine the last element of the list, you may not call next again (among any of your iterators, if you use more than one).

Hint: You may want to use your iterator to help you traverse through the list. Or even multiple iterators traversing the list at the same time.

Document Formatting

You are given a text document in which each word is represented by a node with String value, and each line is represented by a linked list (`LinkedList<String>`). The document as a whole is represented by a linked list of lists (`LinkedList<LinkedList<String>>`). You are given an integer `maxWords` which denotes the maximum acceptable word count on a given line.

You may modify the input document. Return a reformatted document in which no line exceeds `maxWords` words, and the following criteria are met:

- The order of the words is preserved
- The number of lines in the reformatted document is minimized
- The new document must be composed of the same word nodes as the original (you cannot create new `LinkedListNode`'s)
- Any lines you put together via `concatenate` may not exceed

  $$2 \cdot \max(\texttt{maxWords}, \text{length of longest line in original doc})$$

  (Imagine the word count of the document is extremely large, and you can't fit all the words onto one line.) We don't want you to concatenate the entire document into one really long list during any intermediate steps of the algorithm.

Example where `maxWords` = 4:

Before:
CIS 121 is the final
course
in the introductory
computer science sequence at the University of Pennsylvania

After:
CIS 121 is the
final course in the
introductory computer science sequence
at the University of
Pennsylvania

Another example, where `maxWords` = 3.

Before:
CIS
121
is
fun

After:
CIS 121 is
fun

Hint: Consider keeping a buffer of words to be added to the new document, and `truncate`-ing it as needed. You also may want to write down the algorithm in English before you try coding it.

Style & Tests (15 points)

The above parts together are worth a total of 60 points. The remaining 15 points are awarded for code style, documentation, and sensible tests.

In a typical assignment, style is worth 5 of the 15 points, and you will be graded according to the 121 style guide.

In this assignment, all the 15 points will be assigned during a face-to-face code review with your style grading TA for the assignment. This code review will be based on the style guide and your ability to communicate design decisions made during your coding process. This meeting will be an excellent opportunity to evaluate what you can improve on as you move forward through the semester when it comes to writing effective unit tests and beautiful code.