Problem Set 3

Due before lecture on Wednesday, October 29

Part I: Short-Answer ("Written") Problems (45 points total)

Submit your answers to part I in a *plain-text file* *called* `ps3_partI.txt`*, and put your name and email address at the top of the file.*

1. **Printing the odd values in a list of integers** (10 pts. total; 5 pts. each part)
   Suppose that you have a linked list of integers containing nodes that are instances of the following class:

```
public class IntNode {
    private int val;
    private IntNode next;
}
```

   a. Write a method named `printOddsRecursive()` that takes a reference to the first node in a linked list of `IntNode`s and uses recursion to print the **odd values** in the list (if any), with each value printed on a separate line. If there are no odd values, the method should not do any printing.

   b. Write a method named `printOddsIterative()` that uses iteration to perform the same task.

   You do *not* need to code up these methods (or any method from Part I) as part of a class. Simply put the methods in your text file for Part I. You may assume that the methods you write are static methods of the `IntNode` class.

2. **Improving the efficiency of an algorithm** (15 pts. total; 5 pts. each part)
   Consider the following method, which takes two unsorted lists, `list1` and `list2` – both instances of our `LLList` class from lecture – and creates a third list containing the intersection of `list1` and `list2`:

```
public static LLList intersect(LLList list1, LLList list2) {
    LLList inters = new LLList();

    for (int i = 0; i < list1.length(); i++) {
         Object item1 = list1.getItem(i);
         for (int j = 0; j < list2.length(); j++) {
             Object item2 = list2.getItem(j);
             if (item2.equals(item1)) {
                 inters.addItem(item2, inters.length());
                 break;   // move onto the next item from list1
             }
         }
    }

    return inters;
}
```

The lists are unsorted, so we take a nested-loop approach in which each item in `list1` is compared to the items in `list2`; if a match is found in `list2`, the item is added to the intersection and we break out of the inner loop. (Note that this method will include duplicate values in the intersection when `list1` itself has duplicates; doing so is acceptable for the purposes of this problem.)

a. What is the worst-case running time of this algorithm as a function of the length *m* of `list1` and the length *n* of `list2`? Don't forget to take into account the time efficiency of the underlying list operations, `getItem()` and `addItem()`. Use big-O notation, and explain your answer briefly.

b. Rewrite this method to improve its time efficiency. One way to do this is to employ one or more of the iterators associated with our `LLList` class, but there may be other steps that you can take as well. Your new method should *not* modify the existing lists in any way, and it should use no more than a constant (O(1)) amount of additional memory. Make the new method as efficient time-wise as possible, given the constraints on memory usage. You should assume this method is *not* part of the `LLList` class, and therefore it doesn't have direct access to the private `LLList` members.

c. What is the worst-case running time of the improved algorithm? Use big-O notation, and explain your answer briefly.

3. **Removing a value from a doubly-linked list** (10 points)
Suppose that you have a doubly-linked list (implemented using the `DNode` class described in the last written problem of Problem Set 2). Write a method named `removeAllOccurrences()` that takes two parameters, a reference to the first node of the linked list and a character `ch`, and removes all occurrences of `ch` from the linked list. Your method will need to return a reference to the first node in the linked list, because the original first node may end up being removed (see the `StringNode insertChar` and `deleteChar` methods for examples of methods that do something similar). You may assume that the method you write is a static method of the `DNode` class.

4. **Testing for palindromes using a stack** (10 points)
A palindrome is a string like "radar" and "racecar" that reads the same in either direction. Write a static method called `isPalindrome()` that takes a `String` object as a parameter and uses a stack to determine if it is a palindrome, returning `true` if it is and `false` if it is not. Use an instance of either our `ArrayStack<T>` or `LLStack<T>` class. A string of length 1 and an empty string should both be considered palindromes. Throw an exception for `null` values. A written version of this method is all that is required. *Hints:* (1) This problem is similar to the delimiter-testing problem in the lecture notes. (2) You can't use a primitive type like `char` when creating an instance of a generic class. However, if you create a stack of type `Character` (the wrapper class for `char`), then you can store `char` values in the stack.

II. Programming Problems (55-65 points total)

1. **From recursion to iteration** (30 points total)
   In the file `StringNode.java`, rewrite the recursive methods of the `StringNode` class so that they use iteration (for, while, or do..while loops) instead of recursion. You do *not* need to rewrite the `read()` or `numOccurrences()` methods; they are included in the section notes and solutions. Leave the `main()` method intact, and use it to test your new versions of the methods.

   Notes:
   - Before you get started, we recommend that you put a copy of the original `StringNode` class in a different directory, so that you can compare the behavior of the original methods to the behavior of your revised methods.
   - The revised methods should have the same method headers as the original ones. Do not rename them or change their headers in any other way.
   - Make sure to read the comments accompanying the methods to see how they should behave.
   - Make your revised methods as efficient as possible. For example, you should *not* write a method that traverses a linked list by repeatedly calling `getNode()`. Rather, you should follow the approach discussed in lecture for iteratively traversing the nodes in a linked list.
   - Because our `StringNode` class includes a `toString()` method, you can print a StringNode `s` in order to see the portion of the linked-list string that begins with `s`. You may find this helpful for testing and debugging.
   - Another useful method for testing is the `convert()` method, which converts a Java `String` object into the corresponding linked-list string.

   Suggested approach:

   a. Begin by rewriting the following methods, all of which do not create `StringNode` objects or return references to existing `StringNode` objects: `length()`, `indexOf()`, and `print()`. You may need to define a local variable of type `StringNode` in each case, but otherwise these should be relatively easy. Your `print()` method must print one character at a time – it may not use the `toString()` method!

   b. Next, rewrite `compareAlpha()`, which also doesn't create or return a `StringNode`, but which is a bit trickier because it processes two strings.

   c. Next, rewrite `getNode()`, which is the easiest of the methods that return a reference to a `StringNode`.

   d. Next, rewrite `copy()`. The trick here is to keep one reference to the beginning of the new string and another reference to the place at which the next new character will be inserted.

   e. Finally, rewrite `concat()` and `substring()`. You may be able to make use of one or more of the other methods here, although doing so is *not* required. However, you may *not* use the `getNode()` method.

f. The remaining methods either don't use recursion in the first place or are handled in the section notes (`read() and numOccurrences()`), so you don't need to touch them.

g. Test everything. Make sure you have at least as much error detection in your new methods as in the original ones!

A general hint: diagrams are a great help in the design process!

2. **More practice with recursion** (10-20 points total; 5 points each part)
Now that we've created some iterative methods, let's return to recursion! Add the methods described below to the `StringNode` class. ***For full credit, the methods must be fully recursive***: they may not use any type of loop, and they must call themselves recursively. In addition, ***global variables (variables declared outside of the method) are not allowed.*** If you are unable to come up with a recursive solution, you may submit an iterative one for some partial credit.

   a. `public static void printEveryOther(StringNode str)`
   This method should use recursion to print every other character in the string represented by `str`. For example, let's say that we have used the `convert` method in the `StringNode` class to create a linked list for the string `"method"` as follows:

   ```
   StringNode str = convert("method");
   ```

   For this value of `str`, the call `printEveryOther(str)` should print:

   ```
   mto
   ```

   The method should print a blank line if `null` (representing an empty string) is passed in as the parameter.

   Hints: When constructing the parameter for the recursive call, you may need to take a slightly different approach than the one that we have typically used when processing linked-list strings recursively. Make sure that your method works correctly for both even-length and odd-length strings.

   b. `public static char largestChar(StringNode str)`
   This method should use recursion to find and return the character with the largest character code in the string specified by `str`. You should make use of the fact that characters can be compared using the standard relational operators; in particular, the `char ch1` is larger than the `char ch2` if `ch1 > ch2`. For example, given the linked list created in part a, `largestChar(str)` should return `'t'`, because it has the largest character code among the characters in the string `"method"`. If `null` is passed in as the parameter, the method should return the character `'\0'` (a backslash followed by a zero), which has the smallest character code of any character.

c. (*required of grad-credit students; "partial" extra credit for others)*
`public static boolean startsWith(StringNode str, StringNode prefix)`
This method should use recursion to determine if the string specified by the parameter `str` starts with the string specified by the parameter `prefix`. For example, let's say that we have used the `convert` method in the `StringNode` class to create several linked-list strings as follows:

```
StringNode str1 = convert("recursion");
StringNode str2 = convert("recur");
StringNode str3 = convert("recurse");
```

Given these lines of code, the call `startsWith(str1, str2)` should return `true`, whereas `startsWith(str1, str3)` should return `false`. If the second parameter is `null` (representing the empty string), the method should return `true`, regardless of the value of the first parameter, because any string has the empty string as a prefix. If the first parameter is `null` and the second parameter is not `null`, the method should return `false` because the empty string doesn't have any string other than itself as a prefix.

d. (*required of grad-credit students; "partial" extra credit for others)*
`public static int lastIndexOf(StringNode str, char ch)`
This method should use recursion to find and return the index of the *last* occurrence of the character `ch` in the string `str`, or -1 if `ch` does not appear in `str`. For example, given the linked-list string `str3` created above:
- `lastIndexOf(str3, 'e')` should return 6, because the last occurrence of 'e' in "recurse" has an index of 6.
- `lastIndexOf(str3, 'r')` should return 4, because the last occurrence of 'r' in "recurse" has an index of 4.
- `lastIndexOf(str3, 'l')` should return -1, because there are no occurrences of 'l' in "recurse".

Your method should *not* use any of the existing `StringNode` methods.

3. **Capital-gain calculator** (15 points)
(This problem is based on one by Michael Goodrich and Roberto Tamassia.)
The *capital gain* (or *loss*) on a single share of stock is the difference between the share's selling price and the price originally paid to buy it. If an investor purchases shares in a given stock on more than one day, computing the capital gain that results from selling shares in that stock requires that we identify which shares are actually being sold. For example, if I buy 50 shares on a day when they are selling for $10 each and 30 shares on a day when they are selling for $15 each, and I later sell 12 of my shares when they are selling for $20 each, should my capital gain be 12 * ($20 – $10) = $120, or 12 * ($20 - $15) = $60, or something involving a fraction of each set of shares?

A standard accounting practice for identifying the shares sold is to use a FIFO (first-in, first-out) protocol—the shares sold are the ones that have been held the longest. For example, suppose that we buy 100 shares at $20 each on day 1, 20 shares at $24 each on day 2, 200 shares at $36 each on day 3, and then sell 150 shares at $30 each on day 4. The FIFO protocol specifies that the 150 shares sold include all 100 of the shares bought on day 1, all 20 of the shares bought on day 2, and 150 – 100 – 20 = 30 of the shares bought on day 3. The capital gain in this case would thus be computed as follows:

	# of shares	sale price	purchase price	capital gain or loss
	100	30	20	100*(30 – 20) = 1000
	20	30	24	20*(30 – 24) = 120
	30	30	36	30*(30 – 36) = –180
total:	150			940

In the file `CapitalGainCalc.java`, we have given you a framework for a simple class for calculating capital gains. We have provided skeletons for the methods that you will need to implement, along with a `main()` method that includes all of the code needed for a simple text-based user interface. Here is a sample interaction with the user:

```
(1) purchase, (2) sale, or (3) done: 1
number of shares: 40
price: 30

(1) purchase, (2) sale, or (3) done: 1
number of shares: 80
price: 40

(1) purchase, (2) sale, or (3) done: 2
number of shares: 50
price: 35
Capital gain on sale: $150

(1) purchase, (2) sale, or (3) done: 1
number of shares: 50
price: 37

(1) purchase, (2) sale, or (3) done: 2
number of shares: 90
price: 40
Capital gain on sale: $60

(1) purchase, (2) sale, or (3) done: 2
number of shares: 50
price: 45
** You don't have 50 shares to sell. **

(1) purchase, (2) sale, or (3) done: 3
Total capital gain: $210
```

The user can enter an arbitrary sequence of purchases and sales; the program should assume that the actions happened on successive days in the order in which they are entered by the user. The only restriction is that the user cannot attempt to sell more shares than he/she currently owns; when such an attempt is made, no shares are sold and an error message is printed. The program prints the capital gain or loss from each individual sale, and, it prints the total capital gain for all of the sales before it exits.

Your job is to complete the implementation of the `CapitalGainCalc` class. You should begin by determining which fields are needed. In particular, you should select an appropriate sequence data structure (list, stack, or queue) for keeping track of the stock shares that the user owns at a given point in time. You will not implement this data structure; instead, you should use one of the sequence classes that we have given you in lecture. You may *not* modify the class that you select in any way; rather, you must use it in its existing form. Don't forget to include the appropriate interface file for the class that you select.

Within the `CapitalGainCalc` class, we have also given you an inner class called `Purchase` that encapsulates information about a single stock purchase. You will store objects of this class in the sequence data structure that you select. When only a portion of the shares bought on a given day are sold, the `Purchase` object for those shares should be updated to reflect the new number of shares, but the object itself should retain its position in the sequence. ***Reminder:*** Because `Purchase` is an inner class of `CapitalGainCalc`, your methods will have direct access to the fields of `Purchase`.

In addition to a sequence data structure, you may also need one or more other fields to keep track of relevant details about the user's stock holdings. To determine which fields you need, you may find it helpful to first sketch out your implementation of the `processPurchase()` and `processSale()` methods described below.

After adding the necessary fields, you should implement each of the following: (1) the `CapitalGainCalc` constructor; (2) `processPurchase()`, which takes the number and price of shares purchased and performs whatever bookkeeping is needed to keep track of that purchase; (2) `processSale()`, which takes the number and price of shares to be sold, applies the FIFO protocol, and returns the capital gain produced by the sale. The method should also perform whatever bookkeeping is needed to reflect the shares that have been sold. If the user doesn't have enough shares to sell, `processSale()` should throw an `IllegalArgumentException` with an error message like the one shown in the sample interaction, and it should not process the sale.

Make each of the methods as efficient as possible. In particular, the data structure that you select should allow you to perform each of the necessary insertions and removals in constant time (O(1)). Processing a partial sale of the shares from a given `Purchase` object (see above for details) should also be accomplished in O(1) time. In addition, when you apply the FIFO protocol, your data structure (and any other field(s) that you include for book-keeping purposes) should make it possible for you to access only those `Purchase` objects that are relevant to the sale, without accessing any other objects in the sequence. In particular, you should *not* need to access the sequence of purchases in order to determine if the user has enough shares to make a proposed sale. Rather, you should design your fields so that you can determine how many shares the user has at any given time without directly examining the sequence itself.

Submitting Your Work

You should use the `ps3` folder in the homework submissions dropbox to submit the following files:

- your `ps3_partI.txt` file containing your part I answers
- your modified `StringNode.java` file
- your modified `CapitalGainCalc.java` file

Make sure that these classes have the correct names, paying attention to the use of uppercase and lowercase letters in the names. If any of your classes have incorrect names, fix them, and recompile your code until you have made all of the necessary changes.

Here are the steps:

- Go to the homework submissions dropbox (logging in as needed using the Login link in the upper-right corner, and entering your Harvard ID and PIN).
- Open the folder for `ps3`.
- Upload each of your files into this folder.
- **Click on the link for each file to view it so that you can ensure that you submitted the correct file.** We will not accept any files after the fact, so please check your submission carefully.

Note: If you encounter problems submitting your files, close your browser and start again, or try again later if you still have time. If you are unable to submit and it is close to the deadline, email your homework before the deadline to `cscie22@fas.harvard.edu`.