Problem Set 3

Due before lecture on Wednesday, October 28

Preliminaries

In your work on this assignment, make sure to abide by the policies on academic conduct described in the syllabus. If you have questions while working on this assignment, please come to office hours, post them on Piazza, or email `cscie22@fas.harvard.edu`.

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. **Initializing a doubly linked list** (8 points)
   Suppose you have a doubly linked list (implemented using the `DNode` class described in the last written problem of Problem Set 2) in which the `prev` references have the correct values but the `next` references have not yet been initialized. Write a method that takes one parameter, a reference to the **last** node of the linked list, and traverses the linked list from back to front, filling in the `next` references. The method should return a reference to the first node of the linked list. You do *not* need to code up this method as part of a class – a written version of the method itself is sufficient. You may assume that the method you write is a static method of the `DNode` class.

2. **Removing odd values from a linked list** (8 points)
   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;
}
```

   The linked list does *not* use a dummy head node. Write a method `removeOdds()` that takes a reference to the first node in such a list and removes all nodes containing odd numbers. 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).

   For example, if `list` is a reference to the first node in the linked list representing the sequence {5, 12, 8, 3, 11, 16}, `removeOdds(list)` should modify the linked list to one that represents the sequence {12, 8, 16} and return a reference to the node containing the 12. If the original linked list is empty (i.e., if `null` is passed in) or if it contains only odd numbers, the method should return `null` to indicate that the modified linked list is empty.

The method must not perform more than one traversal of the listed list. *Hint:* we encourage you to use a "trailing reference" of the type mentioned near the end of the lecture notes on linked lists.

You do *not* need to code up this method as part of a class – a written version of the method itself is all that is required. You may assume that the method you write is a static method of the `IntNode` class.

3. **Representing polynomials** (12 points total; 3 points each part)
A polynomial is an algebraic expression formed by adding together terms of the form ax^b, where x is a variable and a and b are constants. a is the *coefficient* of the term, and b is the *exponent* of the term. For the sake of this problem, we will assume that all exponents are all non-negative integers. If the exponent is 0, then the term is a constant, because $x^0 = 1$.

For example, the following are all examples of polynomials of the variable x:

$6 + 5x + 8x^5$ $-2x^2$ $4x^3 + (-2)x^7 + x^{12}$

A polynomial can be evaluated for a specific value of x by plugging in the value and computing the result. For example, when x is 2, the first polynomial above has the value $6 + 5(2) + 8(2^5) = 6 + 10 + 8(32) = 272$.

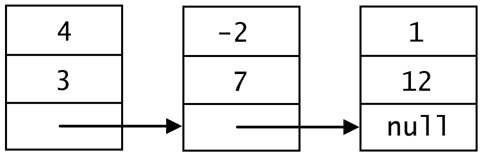
One way to represent a polynomial is to store its coefficients in an array. The exponent of a given term is used to determine the position of the corresponding coefficient in the array. For example, the polynomials given above would be represented by the following arrays:

{6, 5, 0, 0, 0, 8} {0, 0, –2} {0, 0, 0, 4, 0, 0, 0, –2, 0, 0, 0, 0, 1}

An alternative representation of a polynomial involves using a linked list in which each node in the list represents a single term of the polynomial. Here is one example of class that could be used for the nodes:

```
public class PolyNode {
    private int coeff;          // the coefficient
    private int exp;            // the exponent
    private PolyNode next;
}
```

The nodes would be sorted by their exponents, and it would not be necessary to include nodes for non-existent terms (i.e., terms with a coefficient of 0). For example, the linked list for the third polynomial above would look like this:



For each of the following questions, your answers should be big-O expressions that are functions of *t*, the number of terms in the polynomial, and/or *m*, the maximum exponent in the polynomial. (The third polynomial above has $t = 3$ terms and a maximum exponent *m* of 12.) Explain each answer briefly.

a) What is the space efficiency of each implementation?
b) What is the time efficiency of changing the coefficient of a term (e.g., changing the coefficient of the x^3 term in the third polynomial from 4 to 10) in each implementation?
c) What is the time efficiency of evaluating a polynomial in each implementtation?
d) Describe a situation in which one of the two representations would clearly be more efficient than the other one.

4. **Reversing a linked list** (10 points total)
Consider the following method, which takes a list that is an instance of our `LLList` class from lecture and creates and returns a new list that is a reverse of the original list:

```
public static LLList reverse(LLList list) {
    LLList rev = new LLList();

    for (int i = list.length() - 1; i >= 0; i--) {
         Object item = list.getItem(i);
         rev.addItem(item, rev.length());
    }

    return rev;
}
```

Note that the original list is not altered.

a) What is the worst-case running time of this algorithm? 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. (3 points)

b) Rewrite this method to improve its time efficiency by improving the ways in which the method manipulates the `LLList` objects. Your new method should *not* modify the original list in any way. Make the new method as efficient as possible. You should assume that this method is *not* part of the `LLList` class, and therefore it doesn't have direct access to the fields of the `LLList` objects. In addition, you may assume that the `LLList` objects include support for the list-iterator functionality discussed in lecture. You do *not* need to code up this method as part of a class. (4 points)

c) What is the worst-case running time of the improved algorithm? Use big-O notation, and explain your answer briefly. (3 points)

5. **Using a queue to search a stack** (7 points)
Suppose that you have a stack S, and you need to determine if it contains a given item I. Because you are limited to the operations that a stack supports, you can't just iterate over the items in S to look for I. However, you can remove items from S one at a time using the pop operation and later return them using the push operation.

Use either pseudocode or Java code to define an algorithm that uses an initially empty queue Q to search for an item I in the stack S. Your algorithm should return true if the item is found and false otherwise. In addition, your algorithm must ***restore the contents of S*** – putting the items back in their original order.

You algorithm may use S, Q, and a constant number of additional variables. It may *not* use an array, linked list, or any other data structure. You may assume that the stack S supports the operations in our `Stack<T>` interface, that the queue Q supports the operations in our `Queue<T>` interface, and that both S and Q are able to store objects of any type.

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()`, `contains()`, and `print()`. You may need to define a local variable of type `StringNode` in each case, but otherwise these should be relatively easy. Your implementation of `contains()` should *not* make use of the `numOccurrences()` method. Your `print()` method must print one character at a time – it may *not* use the `toString()` method!

b. Next, rewrite `printReverse()`. This method is easy to implement using recursion – as we have done in `StringNode.java` – but it's more difficult to implement using iteration. Here are two possible iterative approaches, either of which will receive full credit:
   - *reverse the string before printing it*: Begin by performing a traversal that reverses the string in place, switching the `next` fields so that they point to the previous node, rather than the next node. Next, make a traversal to print the reversed string (you may *not* use the `toString()` method for this!). Finally, make a traversal to reverse the string again and thereby restore it to its original form. We recommend this approach because it uses less memory, and it will also give you more practice with manipulating linked lists.
   - *use an array:* Create an array of type `char` that is large enough to hold all of the characters in the string and use it to help you with the problem. This approach is easier to implement, so we'll leave it to you to figure out the rest of the details.

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

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

e. Finally, rewrite `concat()`. 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 will be handled in section (`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 let's return to recursion! Add the methods described below to the `StringNode` class. ***For full credit, these 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.*** For parts b-d, if you can't come up with a recursive solution, you may submit an iterative one for partial credit.

a) `public static void toUpperCase()`
We've given you an iterative version of this method in `StringNode.java`. Change it so that it uses recursion instead.

b) `public static void printMirrored(StringNode str)`
This method should use recursion to print the string represented by `str` in "mirrored" form – in which the characters in `str` are followed by the same characters in reverse order. For example, let's say that we have used the `convert` method in the `StringNode` class to create a linked list for the string `"hello"` as follows:

```
StringNode str = StringNode.convert("hello");
```

Given this `str`, the call `StringNode.printMirrored(str)` should print:

```
hellloolleh
```

If an empty string (i.e., the value `null`) is passed in as the parameter, the method should not print anything (it should simply return).

Hint: This method is similar to the `print()` method that is already part of the `StringNode` class, and we recommend that you start with the code in that method and make whatever changes/additions are needed to produce mirrored output. You should not need to do much to the existing code!

c) (*required of grad-credit students; "partial" extra credit for others)*
`public static StringNode substring(StringNode str, int start, int end)`
This method should use recursion to create a new linked list that represents the substring of `str` that begins with the character at index `start` and ends with the character at index `end - 1` (the character at index `end` is *not* included, following the convention used by the `substring` method in the `String` class). For example, given the linked list `str` created in part b, the method call `StringNode.substring(str, 1, 3)` should return a reference to the first node of a new linked list representing the string `"el"`. The method should *not* change the original linked list, and it should *not* use any of the existing `StringNode` methods.

The method should throw an `IllegalArgumentException` if the indices are out of bounds (less than 0 or greater than or equal to the length of the string), or if `end` is less than `start`. If the indices are valid and `end` is equal to `start`, the method should return `null` (representing an empty string). If `end` is not equal to `start` and `str` is `null`, the method should throw an `IllegalArgumentException`. *Note:* You may not need to explicitly check for all of these conditions. In particular, you shouldn't need to check for cases in which the indices are too large; in such cases, the recursive calls should eventually get to a call in which `str` is `null`.

d) (*required of grad-credit students; "partial" extra credit for others*)
`public static int nthIndexOf(StringNode str, int n, char ch)`
This method should use recursion to find and return the index of the `n`th occurrence of the character `ch` in the string `str`, or -1 if there are fewer than `n` occurrences of `ch` in `str`.

For example, given the linked list `str` created in part b:

- `nthIndexOf(str, 1, 'l')` should return 2, because the first occurrence of 'l' in "hello" has an index of 2.
- `nthIndexOf(str, 2, 'l')` should return 3, because the second occurrence of 'l' in "hello" has an index of 3.
- `nthIndexOf(str, 3, 'l')` should return -1, because there are not three occurrences of 'l' in "hello".

The method should return `-1` if `str` is `null`, or if `n` is less than or equal to 0.

3. **Grade database** (15 points)
In the file `GradeDatabase.java`, we have given you the framework for a simple, in-memory database of information about students and their grades. The database will use objects of type `StudentRecord` to store a student's id number, last name, and first name; and it will use objects of type `GradeRecord` to store information about a given grade entry – the id number of the student in question, a description of the assignment or exam (e.g., "PS 1"), and the grade itself.

Your job is to complete the implementation of this program. Begin by adding instance variables for two "tables" of data: one containing student records, and one containing grade records. Use an `LLList` object for each table. Next, add code to the constructor so that it initializes your instance variables.

Finally, you should add code to implement each of the following methods: (1) `addStudent()`, which should add a `StudentRecord` object to the table of student records; (2) `addGrade()`, which should add a `GradeRecord` object to the table of grade records; (3) `printStudents()`, which should print the entire table of student records, (4) `printGrades()`, which should print the entire table of grade records, and (4) `printStudentsGrades()`, which should print, for each student in the student table, all of his or her grades in the grade table. Your `addStudent()` and `addGrade()` methods do *not* need to check if the specified student is already in the database.

In the skeleton code for the print methods, we have provided code that prints a header for the output produced by the method; the code you add should print one entry per line, with the fields in the order specified by the header. The entries output by `printStudentsGrades()` should be grouped together by student.

For example, let's say that you have these tables:

```
id      last      first          id        assignment     grade
-----------------------          -------------------------------
200     Smith     Amy            100       PS 2           95
100     Jones     Joe            200       PS 2           80
350     Harvard   John           100       PS 1           88
                                 200       PS 1           100
```

The output of `printStudentsGrades()` should look something like this:

```
last      first   assignment     grade
---------------------------------------------
Smith     Amy     PS 2           80
Smith     Amy     PS 1           100
Jones     Joe     PS 2           95
Jones     Joe     PS 1           88
```

In database terminology, `printStudentsGrades()` should essentially construct a *join* of the student and grade tables – matching student and grade records that have the same id number. Your implementation of this method should *not* output entries in the student table that lack corresponding entries in the grade table, or vice versa. For example, the student John Harvard from the example above (id number 350) doesn't have any entries in the grade table, and thus his name doesn't appear in the output of `printStudentsGrades()`.

Make each of the five methods as efficient as possible. Make sure to take into account the efficiency of the operations that you perform on the two `LLList` objects. In designing your methods, you may find that it is necessary to make efficiency tradeoffs – i.e., in order to make one method more efficient, it may be necessary to make another method less efficient. If this is the case, you should make choices that are informed by your understanding of where the efficiency gains would be more significant, and you should explain your reasoning in comments that accompany the methods in question.

You do *not* need to worry about "cosmetic" concerns, such as listing the entries in alphabetical order or some other order. All that we will be looking for is methods that are correct and as efficient as possible.

Submitting Your Work

You should use Canvas to submit the following files:

- for Part I: your `ps3_partI.txt` file.
- for Part II:
    - your modified `StringNode.java` file
    - your modified `GradeDatabase.java` file

Make sure to use these exact file names for your files. If you need to change the name of a Java file so that it corresponds to the name we have specified, make sure to also change the name of your class and check that it still compiles.

Here are the steps you should take to submit your work:

- Go to the page for submitting assignments (logging in as needed using the Login link in the upper-right corner, and entering your Harvard ID and PIN).
- Click on the appropriate link: either **Problem Set 3, Part I** or **Problem Set 3, Part II**.
- Click on the *Submit Assignment* link near the upper-right corner of the screen. (If you have already submitted something for this assignment, click on *Re-submit Assignment* instead.)
- Use the *Choose File* button to select a file to be submitted. If necessary, continue selecting files by clicking *Add Another File*, and repeat the process for each file. ***Important:*** You must submit all of the files for a given part of the assignment (Part I or Part II) at the same time. If you need to resubmit a file for some reason, you should also resubmit any other files from that part of the assignment.
- Once you have chosen all of the files that you need to submit, click on the *Submit Assignment* button.
- After submitting the assignment, you should check your submission carefully. In particular, you should:
    - Check to make sure that you have a green checkmark symbol labeled *Turned In!* in the upper-right corner of the submission page (where the Submit Assignment link used to be), along with the names of all of the files that you are submitting.
    - Click on the link for each file to download it, and view the downloaded file 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`