Problem Set 8

part I due by 10:00 p.m. on Tuesday, July 29
part II due by 10:00 p.m. on Thursday, July 31

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 the course account (libs111 @ fas . harvard . edu – removing the spaces).

Part I: Short-Answer Problems (40-50 points total)
All of Part I is pair-optional. See the syllabus for a reminder of the rules for working with a partner.

Put your answers to Part I in a plain-text file called ps8_partI.txt, and put your name and email address at the top of the file.

1. Initializing a doubly linked list (10 points)
Suppose you have a doubly linked list (implemented using the DNode class described in the last written problem of Problem Set 7) in which the next references have the correct values but the prev references have not been initialized. Write a method that takes one parameter, a reference to the first node of the linked list, and traverses the entire linked list filling in the prev references. 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 DNode class.

2. Comparing two algorithms (10 points)
Suppose that you want to add an LList constructor that takes a reference to an ArrayList as its only parameter and constructs an LList object that represents the same list as the ArrayList – i.e., that has the same items in the same positions in the list. You have two versions to choose from, both of which are shown below. Their key differences are highlighted using bold type. Both algorithms construct the list correctly, but one is more efficient than the other. Compare the time efficiency of these two algorithms, making use of big-O notation. Make sure to explain briefly how you came up with the big-O expressions that you use, and to say explicitly which algorithm is more efficient.
Algorithm A:
public LLList(ArrayList aList) {
    // initialize an empty list
    head = new Node(null, null);  // dummy head node
    length = 0;

    // add the items from aList to this list
    for (int i = 0; i < aList.length(); i++) {
        Object item = aList.getItem(i);
        addItem(item, i);
    }
}

Algorithm B:
public LLList(ArrayList aList) {
    // initialize an empty list
    head = new Node(null, null);  // dummy head node
    length = 0;

    // add the items from aList to this list
    for (int i = aList.length() - 1; i >= 0; i--) {
        Object item = aList.getItem(i);
        addItem(item, 0);
    }
}

3. Removing the smallest element in a linked list (10 points)
Suppose that you have an unsorted 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 includes a dummy head node that is not used to store an actual data value; it is similar to the dummy head node found in the LLList class discussed in lecture. The dummy head node in the linked list of integers is also of type IntNode, and its val field always has a value of 0. The diagram below shows an example of such a linked list.

![Diagram](image-url)

Write a method named `removeSmallest()` that takes a reference to the dummy head node in such a linked list and (1) removes the node containing the smallest
value and (2) returns the smallest value. If there are multiple occurrences of the smallest value, the method should remove the node containing the first occurrence. The method should not remove the dummy head node.

For example, given the linked list shown above, `removeSmallest(head)` should return the value 5 and change the linked list to look like the one shown below.

![Linked List Diagram]

*The method must not perform more than one traversal of the listed list.* 
*Hint:* the method will need several reference variables to accomplish this task.

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.

4. **Implementing a stack using two queues** (10 points)

In lecture, we discussed two implementations of the stack ADT: one using an array and one using a linked list. Devise a third implementation of the stack ADT that uses two queues, Q1 and Q2. Use pseudocode to describe how you would use Q1 and Q2 to implement the stack operations push, pop, and peek, and give the running time of each operation using big-O notation. Explain briefly how you came up with the big-O expressions that you use.

Your algorithms for these methods may use Q1, Q2, and a constant number of additional variables (either instance variables of the stack or local variables of the method). It may not use an array, linked list, or other data structure. You may assume that the queues support the operations in our `Queue<T>` interface, and that they can store an arbitrary number of objects of any type.

5. **Reversing a doubly linked list** (10 points; *required of grad-credit students; "partial" extra credit for others*)

Write an iterative method named `reverse()` that reverses a doubly linked list (implemented using the `DNode` class described in the last written problem of Problem Set 7). Your method should take a reference to the first node in the linked list, reverse the order of the nodes in the linked list, and return a reference to the new first node.

Your method should use no more than a constant amount of storage space beyond that needed for the linked list itself (i.e., no more than a few extra variables).
You can accomplish this by using the same nodes as the ones in the original linked list and simply adjusting references so that the linked list goes in the opposite direction.

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 DNode class.

II. Programming Problems (60 points total)

1. From recursion to iteration (30 points total; pair-optional)
   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 will be covered in section. 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.
   b. Next, rewrite getNode(), which is the easiest of the methods that return a reference to a StringNode.
   c. Next, rewrite removeAllSpaces().
d. Next, rewrite \texttt{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 \texttt{concat()} and \texttt{replace()}. You may be able to make use of one or more of the other methods in your iterative versions of these methods.

f. The remaining methods either don't use recursion in the first place \texttt{(charAt(), convert(), deleteChar(), insertChar(), insertSorted(), toString() and toUpperCase()}) or will be handled in section \texttt{(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!

\textit{A general hint:} Diagrams are a great help in the design process!

2. \textbf{More practice with recursion} (10 points total; 5 pts each part; pair-optional)

Now that we've created some iterative methods, let's return to recursion! Add the methods described below to the \texttt{StringNode} class. \textbf{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, \textit{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. \texttt{public static void printEveryOther(StringNode str)}

This method should use recursion to print every other character in the string represented by \texttt{str}. For example, let's say that we have used the \texttt{convert} method in the \texttt{StringNode} class to create a linked list for the string "method" as follows:

\begin{verbatim}
StringNode str = convert("method");
\end{verbatim}

Given this value of \texttt{str}, the call \texttt{printEveryOther(str)} should print the following:

\begin{verbatim}
  mto
\end{verbatim}

The method should print a blank line if \texttt{null} (representing an empty string) is passed in as the parameter. \textit{(Note: In Problem Set 4, you wrote a version of this method that processed a Java \texttt{String} object. In this problem, you will write a method that processes a string that is stored in a linked list!)}

\textit{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 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.

3. Implementing the Bag ADT using a linked list (20 points; individual-only)
   Earlier in the course, we worked with the ArrayBag class, which implements the
   Bag interface using an array. In a file named LLBag.java, write a class that
   implements the Bag interface using a linked list instead of an array. You may
   use a linked list with or without a dummy head node. In addition to the methods
   in the Bag interface, your LLBag class should also have a toString() method
   that overrides the default toString() method inherited from the Object class.
   (Consult the ArrayBag toString() method to see what the string returned by
   this method should look like.) Copy over the main() method from the ArrayBag
class, and make whatever modifications are necessary to allow it to test your
LLBag class.

   One of the benefits of using a linked list is that there is no need to specify a
   maximum size—the bag can effectively grow without limit. Therefore, you will
   not need to provide a constructor that specifies a maximum size, and your add() method can always return true. Because the order of the items in a bag doesn't
   matter, you can add items to either end of the linked list; however, make sure
   that your method adds items in $O(1)$ time. In general, you should make your
   methods as efficient as possible. For example, when implementing the remove() method, you should make sure that you don't traverse the list more than once.

In designing your implementation, you may find it helpful to compare LList, our linked-list implementation of the List ADT, to ArrayList, our array-based implementation of that same ADT. In the same way that LList uses a linked list instead of an array to implement a list, your LLBag class should use a linked list instead of an array to implement a bag. Like the LList class, your LLBag class should use a private inner class called Node for the nodes in the linked list.
Submitting Your Work
You should use the ps8 folder in the homework submissions dropbox to submit the following files:

- your ps8_partI.txt file containing your part I answers
- your modified StringNode.java file
- your LLBag.java file

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 ps8.
- Upload each of your files into this folder.
- If you worked on one or more pair-optional problems with a partner, you should click on the Comment link for the relevant files and include a comment that specifies that the name of your partner and the problems that you worked on together.
- In addition, you should 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 libs111 @ fas . harvard . edu (with the spaces removed).