# Problem Set 3

*Due before the start of lecture on October 31, 2018.*
*See below for a summary of the policies regarding late submissions.*



## Preliminaries

Homework is due prior to the start of lecture. If it is submitted more than 10 minutes after the start of lecture, it will be considered a full day late. There will be a 10% deduction for late submissions that are made by 11:59 p.m. on the Sunday after the deadline, and a 20% deduction for submissions that are made after that Sunday and before the start of the next lecture. ***We will not accept any homework that is more than 7 days late.*** Plan your time carefully, and don't wait until the last minute to begin an assignment. Starting early will give you ample time to ask questions and obtain assistance.

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 attend office hours, post them on Piazza, or email `cscie22@fas.harvard.edu`.

## Part I

*40 points total*

Submit your answers to Part I in a plain-text file called `ps3_partI.txt`.

### Problem 1: Initializing a doubly linked list

*8 points*

Suppose you have a doubly linked list (implemented using the `DNode` class described in Problem 6 of Problem Set 2) in which the `prev` references have the correct values but the `next` references have not yet been initialized. Write a static

method called `initNextFields()` that:

- takes one parameter, a reference to the **last** node of the linked list
- traverses the linked list from back to front, filling in the `next` references
- returns a reference to the first node of the linked list.

You may assume that there is at least one node in the list.

You do *not* need to code up this method as part of a class; simply include it in the same file as your other problems for Part I. You may assume that the method is part of the `DNode` class.

## Problem 2: Choosing an appropriate list implementation

*12 pts. total; 4 pts. each part*

In lecture, we considered two different implementations of the list ADT: `ArrayList` and `LLList`. For each of the following applications, decide which list implementation would be better for that particular application. Your should consider both time efficiency and space efficiency, and you should assume that both implementations have an associated iterator like the `LLListIterator` that we discussed in lecture. Explain your answers.

1. You need to store information about the gardening tools that you sell in your online store. To do so, you maintain a list of product records, each of which contains information about a single type of product, including how many copies of it you have in stock. You don't tend to change the types of tools that you sell, and thus items are seldom added or removed from the list. However, you need to update a product's record whenever your inventory of that product changes (e.g., because someone purchases one!). A product's position in the list is also its product ID, and therefore updating a record involves using the product ID to get the corresponding record and modifying its contents.
2. You need to maintain a list of tweets that are made by government officials in the current week. The number of tweets can vary widely from week to week in a way that is hard to predict. You start with an empty list at the start of the week, and as tweets are made you add them to the list. You need to frequently display the tweets in reverse chronological order – from most recent to least recent.
3. You are responsible for maintaining a monthly list of events for an online arts calendar. The number of events is fairly consistent from month to month. Events are added to the list in chronological order, and you need to display them in that same order.

## Problem 3: Reversing a linked list

*12 points total; 4 points each part*

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.

1. 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.

2. 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.

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

### Problem 4: Using a queue to search a stack

*8 points*

*We will cover the material needed for this problem on October 17.*

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()` method and later return them using the `push()` method.

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.

## Part II

*60-70 points total*

### Preparing for Part II

Begin by downloading the following zip file: ps3.zip

Unzip this archive, and you should find a folder named `ps3`, and within it the files you will need for Part II. **Make sure that you keep all of the files in the same folder.**

**Important:** When you compile the code in this folder, you may see one or more warnings labeled "unchecked cast". You can safely ignore them.

### Problem 5: Rewriting linked-list methods

*30 points*

***If you haven't already done so, complete the steps above to prepare for this and the remaining problems in Part II.***

In lecture, we've been looking at linked lists of characters that are composed of objects from the `StringNode` class. The class includes a variety of methods for manipulating these linked lists, and many of these methods provide functionality that

is comparable to the methods found in Java `String` objects.

Some of the existing `StringNode` methods use recursion, while others use iteration (i.e., a loop!). In this problem, you will rewrite several of the methods so that they use the alternative approach.

**Guidelines**

- 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.
- Global variables (variables declared outside of the method) are not allowed.
- Make sure to read the comments accompanying the methods to see how they should behave.
- 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. **However, you may *not* use the `toString()` method as part of any of your solutions.**
- You should **not** use the existing `getNode()` or `charAt()` methods in any of your solutions, because we want you to practice writing your own code for traversing a linked list.
- **Make your methods as efficient as possible.** For example, you should avoid performing multiple traversals of the linked list if your task could be performed using a single traversal.
- Another useful method for testing is the `convert()` method, which converts a Java `String` object into the corresponding linked-list string.
- There is existing test code in the `main()` method. Leave that code intact, and use it to test your new versions of the methods. You are also welcome to add extra test code to this method, although doing so is not required.
- You can also test your revised methods from the Interactions Pane in DrJava. For example:

```
> StringNode s1 = StringNode.convert("hello");
> StringNode.length(s1)
5
```

- *A general hint:* Drawing diagrams will be a great help as you design your revised methods.

0. Before you get started, we recommend that you put a copy of the original `StringNode` class in a different folder, so that you can compare the behavior of the original methods to the behavior of your revised methods.
1. Rewrite the `length()` method. Remove the existing recursive implementation of the method, and replace it with one that uses iteration instead.
2. Rewrite the `toUpperCase()` method. Remove the existing iterative implementation of the method, and replace it with one that uses recursion instead. No loops are allowed.
3. Rewrite the `printReverse()` method so that it uses iteration. 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 linked list before printing it*: This approach is the trickier of the two, but we recommend it because it uses less memory, and it will give you more practice with manipulating linked lists. Remember that you may *not* use

the `toString()` method for this or any other method. In addition, you should *not* use the `print()` method. Finally, don't forget to restore the linked list to its original form!

- *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.

4. Rewrite the `removeFirst()` method. Remove the existing iterative implementation of the method, and replace it with one that uses recursion instead. No loops are allowed.

5. Rewrite the `copy()` method. Remove the existing recursive implementation of the method, and replace it with one that uses iteration instead.

***Remember: Drawing diagrams will be a great help as you design your revised methods!***

## Problem 6: More practice with recursion

*10 points total; 5 points each part; required of grad-credit students; can be completed for "partial" extra credit by others.*

***For full credit, these methods must be fully recursive***: they may not use any type of loop, and they must call themselves recursively. Global variables (variables declared outside of the method) are not allowed.

1. `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:

   ```
   > StringNode s1 = StringNode.convert("hello");
   > StringNode.substring(s1, 1, 3)
   el
   ```

   The method should *not* change the original linked list, and it should *not* use any of the existing `StringNode` methods.

   *Special cases:*

   - 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`.

2. `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 `s1` created in the example given above for the `substring` method:

   - `nthIndexOf(s1, 1, 'l')` should return `2`, because the first occurrence of `'l'` in `"hello"` has an index of `2`.
   - `nthIndexOf(s1, 2, 'l')` should return `3`, because the second occurrence of `'l'` in `"hello"` has an index of `3`.

- `nthIndexOf(s1, 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`. It should *not* use any of the existing `StringNode` methods.

## Problem 7: Removing all occurrences from a list

*16 points*

Assume that we want list objects to support the following method:

```
boolean removeAll(Object item)
```

This method takes an item, and it should removes ***all*** occurrences of that item from the list. If one or more occurrences of the item are removed, the method should return `true`. If there were no occurrences of the item to begin with, it should return `false`.

Create two implementations of this method: one as part of the `ArrayList` class, and one as part of the `LLList` class. Both classes can be found in the `ps3` folder.

***Important:*** For full credit, both methods should:

- run in *O*(*n*) time, where *n* is the number of items in the list
- use no more than a constant (*O*(1)) amount of additional memory.

***In addition, you should make sure that your `ArrayList` version of the method doesn't leave any extraneous references to objects in the `items` array.*** For example, let's say that you have 9 items in the `ArrayList`. If a call to `removeAll()` removes 3 of them, you should end up with an array in which the first 6 elements hold references to actual items, and all of the remaining elements are `null`.

To facilitate testing, we have added a constructor to both classes that takes an array of type `Object` containing the initial contents of the list. Given these constructors, you can test your methods using examples that look like this:

```
> String[] letters = {"a", "b", "c", "a", "c", "d", "e", "a"};
> ArrayList list1 = new ArrayList(letters);
> list1
{a, b, c, a, c, d, e, a}
> list1.removeAll("a")
true
> list1
{b, c, c, d, e}
> list1.removeAll("c")
true
> list1
{b, d, e}
> list1.removeAll("x")
false
> list1
{b, d, e}
> LLList list2 = new LLList(letters);
> list2
{a, b, c, a, c, d, e, a}
> list2.removeAll("a")
true
> list2
```

```
{b, c, c, d, e}
> list2.removeAll("x")
false
```

### Problem 8: Palindrome tester

*14 points*

*We will cover the material needed for this problem on October 17.*

A palindrome is a string like `"radar"`, `"racecar"`, and `"abba"` that reads the same in either direction. To enable longer palindromes, we can ignore spaces, punctuation, and the cases of the letters. For example:

```
"A man, a plan, a canal, Panama!"
```

is a palindrome, because if we ignore spaces and punctuation and convert everything to lowercase we get

```
"amanaplanacanalpanama"
```

which is a palindrome.

In the file `Problem8.java` that we've included in the `ps3` folder, add a static method called `isPal()` that takes a `String` object as a parameter and determines if it is a palindrome, returning `true` if it is and `false` if it is not.

A string of length 1 and an empty string should both be considered palindromes. Throw an exception for `null` values.

Although this problem could be solved using recursion or an appropriately constructed loop that manipulates the `String` object directly, ***your method* must *use an instance of one or more of the following collection classes from the* `ps3` *folder:***

- `ArrayStack`
- `LLStack`
- `ArrayQueue`
- `LLQueue`

***You must* not *use any other data structure, including arrays or linked lists other than the ones that are "inside" instances of the above collections.*** Rather, you should put individual characters from the original string into an instance of one or more of the above collections, and use those collection object(s) to determine if the string is a palindrome.

For full credit, you should:

- Write your method so that spaces, punctuation, and the cases of the letters don't prevent a string from being a palindrome. To put it another way, make sure that your method only considers characters in the string that are letters of the alphabet and that it ignores the cases of the letters. See our example above.
- Make your method as efficient as possible. In particular:
  - ***You should perform only one iteration over the original `String` object.*** After that one iteration, any additional manipulations of the characters should be done using the collection object(s) that you have chosen to use.
  - Because we want to avoid unnecessary scanning, ***you may* not *use any of the built-in* `String` *methods except* `charAt()` *and* `length()`.**

*Hints:*

- When constructing the collection object(s) that you decide to use, you will need to specify the appropriate data type for the items in the collection. Because `char` values are primitives, you should use the corresponding "wrapper" class, which is called `Character`.
- You may also find it helpful to use the `Character.toLowerCase()` or `Character.toUpperCase()` method.
- Remember that `char` values are essentially integers, so you can compare them just as you would compare integers.

## 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 `ArrayList.java` and `LLList.java` files
  - your modified `Problem8.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:

1. Go to the page for submitting assignments (logging in as needed)
2. Click on the appropriate link: either `ps3, part I` or `ps3, part II`.
3. 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.)
4. Use the *Choose File* button to select a file to be submitted. If you have multiple files to submit, click *Add Another File* as needed, and repeat the process for each file.
5. Once you have chosen all of the files that you need to submit, click on the *Submit Assignment* button.
6. 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 from the part of the assignment 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.**

#### Warning

We will not accept any files after the fact, so please check your submission carefully following the instructions in Step 6.

#### 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.

- If you re-submit a file in Canvas, it will append a version number to the file name. You do *not* need to worry about this. Our grading scripts will remove the version number before we attempt to grade your work.
- There is a *Comments* box that accompanies each submission, but we do **not** read anything that you write in that space. If you need to inform us of something about your submission, please email `cscie22@fas.harvard.edu`.
- 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`.