Problem Set 3

Due before lecture on Wednesday, October 27

Getting Started

To get the files that you will need for this problem set, log into `nice.harvard.edu` and enter the following command:

```
gethw 3
```

This will create a directory named `ps3` in your Harvard home directory containing the files that you need for this assignment. To change into this directory, the command is:

```
cd ~/ps3
```

The final versions of your files for Problem Set 3 should ultimately be put in this directory and submitted from there, following the directions below.

Submitting Your Work

Once you are ready to submit, make sure that all of your files are in your `ps3` directory on `nice.harvard.edu`.

To check that your code compiles, enter the following command from within the `ps3` directory: `make`

To submit your assignment, enter the following command: `make submit`

To check which files were submitted, enter the following command: `make check`

Important note: the submit commands only work if you are logged onto *`nice`*`.harvard.edu`.

I. Written Problems (40-50 points total)

Once again, you may submit your solutions to these problems either in written form or electronically. See Problem Set 1 for details.

1. **Comparing two algorithms** (10 points)
   Suppose that you want to add an `LLList` constructor that takes a reference to an `ArrayList` as its only parameter and constructs an `LLList` 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);
       }
   }
   ```

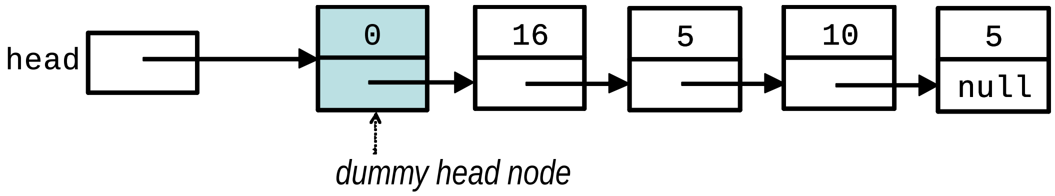
   Algorithm B:
   ```
   public LLList(ArrayList aList) {
       // initialize an empty list
       head = new Node(null, null);  // dummy head node
       length = 0;

       // 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);
       }
   }
   ```

2. **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 2) 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.

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

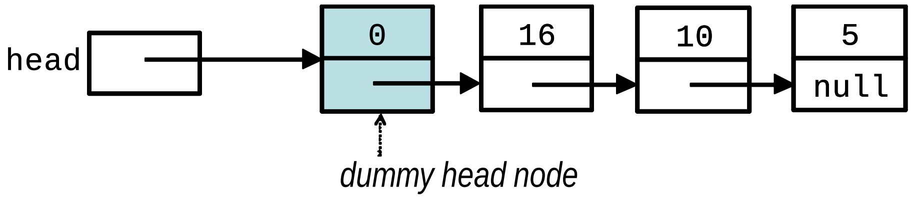
```
public class IntNode {
    private int val;
    private IntNode next;
}
```

   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.



   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.



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 2). 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)
   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 5 notes and solutions. Leave the `main()` method intact, and use it to test your new versions of the methods.

   Here is the 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 `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()` and `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 (`charAt()`, `convert()`, `deleteChar()`, `insertChar()`, `insertSorted()`, `toString()` and `toUpperCase()`) 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. **Implementing the Bag ADT using a linked list** (20 points)
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 `LLList`, our linked-list implementation of the List ADT, to `ArrayList`, our array-based implementation of that same ADT. In the same way that `LLList` 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 `LLList` class, your `LLBag` class should use a private inner class called `Node` for the nodes in the linked list.

3. **Adding iterator support to the ArrayList class** (10 points)
Add support for iterators to the `ArrayList` class. Add code to implement the `iterator()` method in this class, and define an inner class for the actual iterator objects. You may find it helpful to study the iterator support in the `LLList` class as a model for what you need to do. However, you will need to define iterator objects that work in the context of an array-based list, rather than one based on a linked list. Change the code in `IteratorTest.java` to use an `ArrayList` rather than an `LLList`, and check that it produces the same results.