Second Programming Project
CSCI 1913: Introduction to Algorithms,
Data Structures, and Program Development
April 10, 2017

0. Introduction.

For this project, you will implement a highly efficient algorithm to sort singly linked linear lists of integers. (We'll discuss algorithms to sort arrays later in the course.) The project will give you practice working with linked data structures.

1. Theory.

To start, we need a notation for lists of integers. We'll write something like [5, 8, 4, 9, 1, 2, 3, 7, 6] to mean such a list. This list has nine nodes: its first node contains a 5, its second node contains an 8, its third node contains a 4, etc., all the way to its last node, which contains a 6. We'll write an empty list of integers as []. Java *does not* use this notation, so don't put it in your programs.

We want to sort lists like these into *nondecreasing order*. That means that the integers in the list don't decrease as we visit them from left to right. For example, if we sort [2, 3, 1, 1, 2] into nondecreasing order, then we get [1, 1, 2, 2, 3]. Similarly, if we sort [5, 8, 4, 9, 1, 2, 3, 7, 6] into nondecreasing order, then we get [1, 2, 3, 4, 5, 6, 7, 8, 9]. We'll use this list as a running example in the rest of this description.

We also want our sorting algorithm to be as efficient as possible. To explain what we mean by that, let's first consider an inefficient sorting algorithm. It might sort a list by traversing it, looking for pairs of adjacent integers that are in the wrong order, like 8 and 4 in the list shown above. Whenever it finds such a pair of integers, it exchanges their positions, so that 4 comes before 8. The algorithm might repeatedly traverse the list in this way, exchanging integers until all are in the right order. Although this algorithm is simple, it needs $O(n^2)$ comparisons to sort n integers.

The sorting algorithm we'll describe here is more complex, but much more efficient. It works without exchanging adjacent integers, and without traversing lists repeatedly, both of which can make a sorting algorithm slow. Our sorting algorithm needs only $O(n \log n)$ comparisons to sort a list of n integers. The algorithm works in four phases: first *testing*, next *halving*, then *sorting*, and finally *merging*. We'll describe each phase in detail.

> **Testing.** Suppose that the variable *unsorted* is an unsorted list of integers that we want to sort. We test if *unsorted* has length 0 or 1. If so, then the list is already sorted, so the algorithm simply stops and returns *unsorted* as its result.

> **Halving.** In this phase, *unsorted* has two or more integers. We split *unsorted* into two halves, called *left* and *right,* with approximately equal lengths. The order of integers within *left* and *right* doesn't matter. We start (step 01) with *left* and *right* as empty lists. Then, for odd numbered steps (01, 03, etc.) we delete the first integer from *unsorted* and add it to the front of *left*. For even numbered steps (02, 04, etc.), we delete the first integer from *unsorted* and add it to the front of *right* instead. We continue in this way until *unsorted* becomes empty.

STEP	*left*	*right*	*unsorted*
01	[]	[]	[5, 8, 4, 9, 1, 2, 3, 7, 6]
02	[5]	[]	[8, 4, 9, 1, 2, 3, 7, 6]
03	[5]	[8]	[4, 9, 1, 2, 3, 7, 6]
04	[4, 5]	[8]	[9, 1, 2, 3, 7, 6]
05	[4, 5]	[9, 8]	[1, 2, 3, 7, 6]
06	[1, 4, 5]	[9, 8]	[2, 3, 7, 6]
07	[1, 4, 5]	[2, 9, 8]	[3, 7, 6]

08	[3, 1, 4, 5]	[2, 9, 8]	[7, 6]
09	[3, 1, 4, 5]	[7, 2, 9, 8]	[6]
10	[6, 3, 1, 4, 5]	[7, 2, 9, 8]	[]

We did the halving phase like this because it's easy to delete an integer from the front of a linked list (as in a linked stack), and easy to add a new integer to the front of a linked list (again as in a linked stack). Also, it works without having to know the length of *unsorted*.

Sorting. Now we have two unsorted lists from the halving phase, *left* and *right*. In the example, *left* is [6, 3, 1, 4, 5], and *right* is [7, 2, 9, 8]. We sort *left* and *right* into nondecreasing order, by recursively using the sorting algorithm (the one we're describing!) on both lists. After that, *left* is [1, 3, 4, 5, 6], and *right* is [2, 7, 8, 9]. The recursion always terminates, because *left* and *right* are always shorter than the original list *unsorted*. That means *left* and *right* will eventually have exactly one integer, stopping the algorithm during the testing phase.

Merging. Here we merge the sorted *left* and *right* into one sorted list, called *merged,* which is initially empty (step 01). We look at the first integers from *left* and *right,* choosing the one that's smallest. If the integer from *left* is smallest, then we delete it and add it to the end of *merged* (as in step 01). If the integer from *right* is smallest, then we delete it and add it to the end of *merged* (as in step 02). If both integers are equal, then we do this to either one. We continue in this way until *left* or *right* become empty (as in step 07).

STEP	***merged***	***left***	***right***
01	[]	[1, 3, 4, 5, 6]	[2, 7, 8, 9]
02	[1]	[3, 4, 5, 6]	[2, 7, 8, 9]
03	[1, 2]	[3, 4, 5, 6]	[7, 8, 9]
04	[1, 2, 3]	[4, 5, 6]	[7, 8, 9]
05	[1, 2, 3, 4]	[5, 6]	[7, 8, 9]
06	[1, 2, 3, 4, 5]	[6]	[7, 8, 9]
07	[1, 2, 3, 4, 5, 6]	[]	[7, 8, 9]

At this point, either *left* and *right* may not be empty (step 07). If that happens, then we add the entire nonempty list to the end of *merged*. In the example, we get the list [1, 2, 3, 4, 5, 6, 7, 8, 9]. It's sorted into nondecreasing order, so we stop the sorting algorithm and return this list as its result.

We did the merging phase this way because it's easy to delete an integer from the front of a linked list (as in a linked stack) and also easy to add an integer to the end of a linked list (as in a linked queue). It's also easy to add *left* or *right* to the end of *merged* at the end.

For this project, you must implement this sorting algorithm in Java. The next section explains how to implement it efficiently.

2. Implementation.

The file **`SortyList.java`** is available on Moodle. It contains Java source code for the class `SortyList`, which implements a singly linked linear list of `int`'s that can be sorted. (Unlike most classes from the lectures, `SortyList` doesn't need a class parameter like `<Base>`.) You must write a sorting method for `SortyList`. The class `SortyList` has the following members.

```
private class Node
```

A nested class. The linked lists that you must sort are made from instances of `Node`. Each instance of `Node` has an `int` slot called `key`, and a `Node` slot called `next`, which points to the next `Node` in the list, or to `null`. It also has a constructor that initializes `key` and `next`.

`public SortyList()`

Constructor. It makes an empty instance of `SortyList`, with no `Node`'s.

`public SortyList(int first, int ... rest)`

Constructor. It makes an instance of `SortyList` with one or more `Node`'s. The `int` in the first `Node` is `first`. The remaining `Node`'s have `int`'s from the array `rest`. This constructor uses Java syntax that was not discussed in the lectures. Don't worry: you don't have to know how it works.

`private Node head`

A head `Node`, made by `SortyList`'s constructor.

`public SortyList sort()`

Sort the `int`'s in `this` instance of `SortyList`, by calling the private `sort` method. Return the sorted instance.

`private Node sort(Node unsorted)`

THIS IS THE ONLY METHOD YOU MUST WRITE. It takes a linear linked list called `unsorted` as its argument, sorts the list according to its `key` slots, and returns the list after it's sorted. See below for more about how `sort` must work.

`public String toString()`

Return a `string` that allows printing this instance of `SortyList`.

`public static void main(String[] args)`

Make some instances of `SortyList`, sort them, and print them. The tests demonstrate whether your `sort` method works correctly. It also has some examples of how `SortyList`'s constructor and its public `sort` method work.

As stated above, the private method `sort` is all you must write for this project. However, to make the sort algorithm more efficient, and to make the project harder and more interesting, your method `sort` must follow all of these ~~scary~~ helpful rules:

1. You must not use the Java operator `new` in any way! You are not allowed to make new `Node`'s, new instances of other classes, new arrays, etc. If you need a head node, then you must use the private variable `head`, made by `SortyList`'s constructors. That means `sort` must work in $O(1)$ space, not counting the memory space used for recursion.

2. Since you can't make new `Node`'s, your sort algorithm must work only by changing the `next` slots of existing `Node`'s. You can change the `key` slots of `Node`'s too, but you probably won't want to do that.

3. You must use the algorithm described above in the theory section, with its four phases: *testing, halving, sorting,* and *merging*. To simplify grading, you must use the same local variable names: *left, merge, right,* and *unsorted*. If you need more local variables, then they can have any names you want.

4. Your testing phase must take $O(1)$ time. That means it can't use a loop to count how many `Node`'s are in `unsorted`. Hint: recall a question on the second midterm exam.

5. Your halving phase must take $O(u)$ time, where u is the length of *unsorted*. Adding a `Node` to *left* or *right* must take $O(1)$ time. That means the halving phase can't traverse *unsorted* more than once, and it can't traverse *left* and *right* at all. It also can't count how many `Node`'s are in `unsorted`, since that would require an extra traversal. Hint: recall how linked stacks work.

6. Your sorting phase must call *sort* recursively to sort *left* and *right*.

7. Your merging phase must take $O(l + r)$ time, where l is the length of *left,* and r is the length of *right*. Adding a `Node` to *merged* must take $O(1)$ time. That means the merging phase can't traverse *left* and *right* more than once, and it can't traverse *merged* at all. Hint: recall how linked stacks and linked queues work.

8. Your method `sort` must work correctly for lists of any length, and for lists other than the ones in the examples. It also must work correctly for lists with duplicate elements.

9. You may write additional private helper methods, but they must also follow these rules. Your helper methods can have any names you want.

10. You can't change any part of `SortyList`, except its private `sort` method. But `main` is an exception to this rule: you can add more tests to show how your code works. Also, `main` is the only method that is allowed to print things.

If your code violates these rules, then you will lose a large number of points. As a result, if you have questions about whether you're following the rules correctly, then you should contact me or the TA's. Here are more hints:

- Draw box-and-arrow diagrams! It helps to use a whiteboard. If you can make the algorithm work with diagrams, then you can make it work with Java code.

- My implementation of `sort` uses about 75 lines of Java code, not using helper functions, not counting comments, and indenting as I do in the lectures. This should give you a very rough idea of how complex `sort` should be. Your code may be shorter or longer than mine, and still be correct.

3. Deliverables.

Unlike the lab assignments, you must work on this project individually, without a partner. IT MUST BE COMPLETED BY YOURSELF, ALONE! The project is worth 40 points: the testing phase is worth 5 points, the halving phase is worth 15 points, the sorting phase is worth 5 points, and the merging phase is worth 15 points. It will be due on Moodle in **two weeks,** at **11:55 PM** on **April 24, 2017.**

Unlike the laboratory assignments, the TA's will read your Java code in detail, awarding partial credit wherever possible. As a result, you must submit (1) Java source code for the class `SortyList` that includes your code for `sort`, and (2) any output produced by its `main` method. You must submit exactly one `.java` file that contains both these things. Output from `main` must appear in comments at the end of your file.