CSCI 1933 Lab 9

Linked Lists and Iterators

Lab Rules

You may work individually or with *one* partner in lab. We suggest that you have a TA evaluate your progress on each milestone before you move onto the next. The labs are designed to be completable by the end of lab. If you are unable to complete all of the milestones by the end of lab, you have **until the last office hours on Monday** to get those checked off by **any** of the course TAs. We suggest you get your milestones checked off as soon as you complete them since Monday office hours tend to become crowded. If you worked with a partner, both of you must be present when getting checked off to receive credit. You will only receive credit for the milestones you have checked off by a TA. There is nothing to submit to Moodle for this lab.

Attendance

Per the syllabus, students with more than 3 unexcused lab absences over the course of the semester will automatically fail the course. The TAs will take attendance during lab; it is expected that students will be actively working on the lab material. *Your physical presence in lab is not sufficient to be marked as present for the purposes of attendance.*

1 Linked List Iterator

Recall from lecture you saw how to implement an iterator for an array list. Here, you will implement an iterator for a linked list. Make sure you've downloaded `Node.java` and `LinkedList.java` from Moodle. In `LinkedList.java` we've provided you a stripped down linked list implementation with just `add` and `toString`. Take a look at how we've implemented the two methods (more specifically notice that we keep *two* `Node<T>` pointers as member variables).

To get started, implement an *inner class* iterator that implements `next()` and `hasNext()`. We've already given you the skeleton, you just need to fill in these two methods. Recall that when an iterator is at the end of the object it's iterating on and `next()` is called, a `NoSuchElementException` is thrown.

Milestone 1:
In a main method, fill out an instance of `LinkedList` with data and show your iterator working. Also show that when an iterator is at the end of the `LinkedList`, a `NoSuchElementException` is thrown by catching it in the main.

2 Using Your Iterator

Now that you have an iterator, let's use it. Implement the following two methods using your iterator from Milestone 1:

- `public static int[] xify(LinkedList<Integer> x)` – this should return an array of `n` copies of each element `n` in the list `x`.

  For example, if `x` looks like

  $$4 \to 2 \to 3 \to \texttt{null}$$

  you should return the array

  ```
  [4, 4, 4, 4, 2, 2, 3, 3, 3]
  ```

  You may assume all elements in `x` are ≥ 0.

- `public static LinkedList<Integer> countingSort(LinkedList<Integer> lst)` – this should return a sorted version of `lst` using *counting sort*. Counting sort works as follows. Consider an input list of integers that are all ≥ 0 whose maximum value is k. An array, let's call *counts*, with $k+1$ indices is allocated, and the list is iterated through so that for each element x, $counts[x]$ is incremented by one. Then, *counts* is looped over so that for each index i, $counts[i]$ copies of i are added to a new list we are returning. The pseudocode is as follows:

  ```
  function CountingSort(lst)
      k := maximum element in lst
      counts := new array of size k + 1
      for each element x in lst do
          counts[x] := counts[x] + 1
      end for
      ret := new linked list
      for i := 0 to k + 1 do
          add i to ret counts[i] times
      end for
      return ret
  end function
  ```

  You may assume all elements in `lst` are ≥ 0.

For both methods above, the input `LinkedList` *should not* be modified in any shape or form.

Milestone 2:
Show both `xify` and `countingSort` working. What is the time complexity of `countingSort`? If you're unsure, ask a TA.

3 Linked List Practice

Implement the following two methods in `LinkedList`:

- `public void reverse()` – this should reverse `this`'s elements.

  For example, if `this` looks like

  $$12 \to 4 \to 2 \to 3 \to \texttt{null}$$

  it should look like

  $$3 \to 2 \to 4 \to 12 \to \texttt{null}$$

  after a call to `reverse()`.

  > **Constraint:** You must implement `reverse` without ever using the `new` keyword. That means no arrays!

  > **Hint:** You may or may not consider creating several pointers that you update as you walk through the list.

- `public void shuffle()` – this should shuffle `this`'s elements.

  > **Hint:** You may or may not find it useful to write some helper methods.

> **Note:** Recall that `LinkedList` keeps two `Node<T>` pointers, one that's the first node and one that's the last node. You *must* update these to be correct in both of the above methods.

> **Milestone 3:**
> Show both `reverse` and `shuffle` working. Remember not to use `new` in `reverse`.