CE204 Lab 2: Lists

David Richerby

Week 18, 2020–21

The lab exercises are not assessed and you are not required to complete all of them, though I recommend that you attempt them all. Feel free to work with others and talk about your answers.

Solutions will be released on Moodle, on the Friday after the labs.

1 Singly linked lists

a) Write a class `SinglyLinkedList` that implements a singly linked list, including all the operators shown on slide 56.

b) Add `addToHead()` and `addToTail()` methods.

c) Add a `toString()` method to convert your lists to a string.

If you like, you can also add iterator methods; the code will be identical to slide 63.

2 Back to stacks and queues

Use either the class `DoublyLinkedList` from the lectures or your class `SinglyLinkedList` from the previous exercise to implement a stack and a queue. Each one should only require a few lines of code.

3 Practice with recursion

In lecture 2, I presented a basic binary tree implementation in which `null` represents an empty tree, and trees with at least one node are represented by objects of the `BinaryTree` class. We can define a list class `IntList` in a similar way, with operations

- `create (int value)` (i.e., the constructor) creates a one-item list that contains the value `value`;

- `IntList cons (int value)` returns a list made by inserting `value` at the start of `this` list ("cons" is the traditional name for this operation; it's short for "construct");
- `int head()` returns the value at the front of `this` list (but does not remove it from the list);
- `IntList tail()` returns the tail of the list (i.e., the list except for the first element).

This is how lists are typically implemented in functional programming languages such as ML and Haskell. In these languages, programs typically manipulate lists using recursive functions.

a) Implement the class `IntList`. You can download an outline of the Java file from Moodle if you're stuck.

b) Write code to produce the list $1, 2, \ldots, n$ using the constructor and `cons()`.

c) Write a recursive method `int length()` that calculates the length of a list. Hint: the length of a list that has no tail is 1; the length of a list that has a tail is one more than the length of the tail.

d) Write a recursive method `String toString()` that converts the list to a string. You can now print out your list with `System.out.println (list);`.

e) Write a recursive method `Boolean contains(int i)` that returns true if the list contains `i` and false if it doesn't.

f) Write a recursive method `int sum()` that returns the sum of the numbers in the list.

g) A trickier example. Write a recursive method `IntList everyOtherValue()` that returns a list containing the first, third, fifth, ..., values from a list.

h) Using `everyOtherValue()` or otherwise, write a method `IntList otherEveryOtherValue()` that returns the second, fourth, sixth, ..., values from a list.

i) This one is surprisingly tricky to get right for *every* input (or maybe I wasn't concentrating). Write a recursive method `IntList evenValues()` that returns a list of all the even values in a list.

j) This one is difficult: don't worry if you can't do it. Write a recursive method `IntList reverse()` that reverses a list. There's a hint on the next page, if you need it.

Hint for exercise 3j:

`reverse()` itself won't be recursive, but make it call a recursive helper method `IntList reverseHelper(IntList listSoFar)` whose argument is the reversal of the list elements that have been processed so far.