CSCI-1200 Data Structures — Fall 2009
Lab 6 — Lists

Introduction

This lab gives you practice in working with iterators and the STL `list` class. *And* we'll also practice writing and debugging functions for a simple singly-linked list data structure, which will lead to our custom implementation of a list class that mimics the STL `list` class in the next lecture. Download this file (for use in Checkpoints 2 & 3) and then please turn off your internet connection.

```
http://www.cs.rpi.edu/academics/courses/fall09/ds/labs/06_lists/code.cpp
```

Checkpoint 1

Following on our exercises from last week's lab, let's reverse the contents of an STL `list` of doubles. As before, you may not use a second list (or an array or a vector, etc). Remember that the STL `list` container class does have a `size` member function that gives the number of items stored in the list. To practice using a list, you should start by writing, compiling, and testing code to create lists (the test lists) and to print the contents of a list (put this in a function). After you are satisfied that these work, proceed to the actual `reverse` function.

Write this function two different ways. In the first version use iterators and dereferencing (the `*` operator) to *change* the value in each link of the list. This will involve swapping elements similar to the reverse functions you wrote for vectors. In the second version use `insert`, `erase`, `push_front`, `pop_front`, `push_back`, and/or `pop_back` to arrive at the same final state. Write several test cases to ensure that your function works for all inputs. Draw pictures and output intermediate states to the screen to illustrate how the second version of your function works.

To complete this checkpoint and the entire lab, show a TA the two different versions of your reverse function and the test cases you used to debug your code.

Checkpoint 2

Now let's switch to the simple singly-linked list data structure we saw in the Lecture 9. Some code is provided in the file `code.cpp`. Write and test a function named `remove` which takes two arguments: a pointer to the `Node` at the start of the list, and an integer `v` that you should remove from the list. Make sure to perform memory management to avoid memory leaks! You may assume that all of the elements in the list are unique. If the element does not appear in the list, print a simple error message to the screen. Write several test cases to ensure that your function works properly no matter where the element appears in the list (front, back, middle, list of only one element, etc.)

To complete this checkpoint and the entire lab, show a TA your `remove` function and justify that your test cases ensure that your code is debugged.