CS 103 Lab – The Adventure of Links

1 Introduction

In this lab you will implement a double-ended queue using a doubly-linked list.

2 What you will learn

After completing this lab you should be able to:

- Implement an efficient doubly-linked list
- Understand the APIs for different kinds of lists
- Work with pointers to objects
- Dynamically allocate memory
- Write a destructor to prevent memory leaks

3 What the deque?

There are many useful linear data structures, with different APIs. E.g.,

- A *stack* is a sequence that supports insertion at one end, and deletion at the same end (so you always delete the *newest* item)
- A *queue* is a sequence that supports insertion at one end, and deletion at the opposite end (so you always delete the *oldest* item)

The most natural generalization is a sequence where you can insert or delete from either of the two ends. The C++ standard library calls this a *double-ended queue*, or a *deque* (pronounced "deck") for short. In this lab, you will create your own double-ended queue implementation.

How can we create a deque class? *Linked lists* are a flexible kind of data structure: their contents can be spread out all over memory and re-spliced together without having to physically move the contents up/down in sequence. They have many applications. For example, in *The Human Genome Project* they were used because of their efficiency to perform essential operations. This may not be that surprising: the natural process of *meiotic recombination*, crossing two sequences, strongly resembles what would result if two linked lists exchanged pointers.

In lecture we worked through many examples of *singly-linked lists*, which are capable of simulating stacks and queues efficiently (with constant time per operation, meaning no loops are necessary). However, in order to efficiently support both insertion and deletion from both the front *and* back, it is necessary that each list item both remembers its predecessor *and* its successor. This *doubly-linked list* approach is the one you will follow in order to implement your double-ended queue. (Note: it turns out the C++ deque class actually uses a completely different implementation. Stay tuned to learn more if you take CSCI 104.)

4 Procedure

1. Download the starter files:

```
mkdir lab-linkedlists
cd lab-linkedlists
wget http://bits.usc.edu/files/cs103/lab-linkedlists.tar
tar xvf lab-linkedlists.tar
```

2. Look at `delist.h`. This contains:
   a. The definition of a `struct` called `DEItem`, which comprises the individual items in the list. Note that it contains one `int` (your linked list will only be able to contain `int` values) as well as two pointers, which will point to the adjacent `DEItem`s.
   b. The `DEList` API you must follow. In addition to member functions to insert, read, and delete from the front and back, there are utility functions to compute the size, check if it empty, and the constructor and destructor.

3. Look at `delist_test.cpp`, which is a sample client using the `DEList` class. You'll use this for testing.

4. What data members must you add to the `DEList` class definition in `delist.h`? Keep in mind that we want to be able to add or delete from either end without using a loop. Use a diagram to help reason it out.

5. Write the member functions in a file named `delist.cpp`. You may want to copy-and-paste the API and comments from `delist.h` to get started. Remember to include the header. Compile often to make sure you fix syntax errors as you go. Use lots of diagrams to help reason about your code.

6. Compile the testing program with your class definition:

```
$ compile delist.cpp delist_test.cpp -o delist_test
```

   Use `valgrind` to check for memory leaks (as well as other problems such as uninitialized variables or out-of-bounds memory access):

```
$ valgrind --tool=memcheck --leak-check=yes ./delist_test
```

Demo your program to your TA/CP and be ready to explain how any of your member functions work. Your TA/CP may pick some at random and ask you for an explanation of your code or a diagram to illustrate what your code is doing.