CSE 101
Introduction to Data Structures and Algorithms
Programming Assignment 6

In this project you will create a new, and somewhat different integer List ADT, this time in C++. You will use this List to perform shuffling operations, and determine how many shuffles are necessary to bring a List back into its original order. Begin by reading the handout *ADTs in C++*, and by carefully reviewing the Queue and Stack examples posted on the webpage in Examples/C++. These examples establish our norms and conventions for building ADTs in the C++ language. The header file List.h has also been posted at Examples/pa6, along with a test client, some output files and a Makefile for this project.

The Perfect Shuffle
A perfect shuffle, also called a *riffle shuffle*, is one in which a deck of cards is split evenly, then merged into a new deck by alternately inserting cards from each half into the new deck. For instance, if our deck contains 7 cards, labeled 0-6, we would perform the following steps.

Deck:	0 1 2 3 4 5 6
Split:	0 1 2 | 3 4 5 6
Prepare to Merge:	3 4 5 6 0 1 2
Merge:	3 0 4 1 5 2 6

Performing the same perfect shuffle operation on the new list, we get: 1 3 5 0 2 4 6. Repeating the shuffle once again gives the original arrangement: 0 1 2 3 4 5 6. We say that the *order* of this re-arrangement (or permutation) is 3, since applying it to any deck 3 times returns the deck to its original state.

We will represent a deck of n cards by a list of length n, consisting of the integers $(0, 1, 2, \ldots, n-1)$. If n is even, then we can split the list into two equal halves, each of length $n/2$.

$$\left(0, 1, \ldots, \frac{n}{2} - 1\right) \qquad \left(\frac{n}{2}, \frac{n}{2} + 1, \ldots, n-1\right)$$

If n is odd, we place the extra card in the right half. The left half then contains $\lfloor n/2 \rfloor$ cards, and the right half $\lceil n/2 \rceil$ cards.

$$\left(0, 1, \ldots, \left\lfloor \frac{n}{2} \right\rfloor - 1\right) \qquad \left(\left\lfloor \frac{n}{2} \right\rfloor, \left\lfloor \frac{n}{2} \right\rfloor + 1, \ldots, n-1\right)$$

Observe that the latter formulas are correct in both the even and odd case. Your top level client in this project, which will be written in C++, will be called Shuffle.cpp. It will contain a function with the following prototype.

```
void shuffle(List& D);
```

Function `shuffle()` will alter its `List&` (List reference) argument `D` by performing one shuffle operation, as described above. Function `main()` will read a single command line argument, which will be a positive integer specifying the maximum number of cards in a deck. For each n in the range 1 up to this maximum, your program will perform shuffles until the list $(0, 1, 2, \ldots, n-1)$ is brought back to its original state,

counting the number of shuffles as it goes. It will print a table to standard output giving this count, for each value of n. A sample session follows.

```
$ Shuffle 16
deck size       shuffle count
------------------------------
 1               1
 2               2
 3               2
 4               4
 5               4
 6               3
 7               3
 8               6
 9               6
 10              10
 11              10
 12              12
 13              12
 14              4
 15              4
 16              8
```

As usual, the `$` sign represents the Linux prompt. If you re-direct your program output to a file, then you can verify your formatting and results by comparing to the files `out35`, `out75` and `out100` respectively, all posted on the webpage. For instance `Shuffle 35 > myout35` and then `diff myout35 out35` will verify your results up to a deck size of 35.

List ADT

The majority of your work in this project will be to build the List ADT in C++. As you would expect, the implementation will be split into two files, List.h and List.cpp. List.h is provided on the website, and will be submitted unaltered. Note that, unlike ADT implementations in C, the primary class defining the exported type is within the .h file, not the .cpp file. The file List.h therefore includes a private inner struct called Node, the field declarations for the List class, as well as prototypes for ADT operations.

The underlying data structure for this incarnation of the List ADT will be a doubly linked list of Node objects, with two dummy nodes at the front and the back. The empty state of the List will be represented by these two sentinel nodes, pointing to each other as next and prev, respectively. The value stored in the data fields of the dummy nodes can by anything you like, and will not be read from or written to. As you may be aware, dummy nodes are useful for simplifying special cases that arise in the insertion and deletion operations.

A key difference between this List ADT and the one you created in previous assignments is the cursor. Instead of a horizontal bar lying under a list element, the cursor will be envisioned (in the client view) as a vertical bar standing between two elements, or to the left of all elements, or to the right of all elements. In fact, the elements themselves are not indexed. Instead the spaces between the elements are indexed. Unlike our List in C, *the cursor will always stand in one of these in-between positions, and cannot become undefined.* A List containing n elements will therefore have exactly $n + 1$ possible cursor positions, namely 0, signifying the front of the List, through position n at the back of the List. For instance, if $n = 7$ the List has 8 available cursor positions.

```
0     1     2     3     4     5     6     7
|  a  |  b  |  c  |  d  |  e  |  f  |  g  |
```

To represent the cursor within the ADT, we will use *two* Node pointers, which are called `beforeCursor` and `afterCusor` in the .h file. These pointers will always straddle the vertical cursor, pointing to the Node objects immediately before and after the cursor position. If the cursor is at the front of the List (position 0), then `beforeCursor` will point to `frontDummy`. Likewise if the cursor is at the back of the List (position n), then `afterCursor` will point to `backDummy`.

All List operations are described in detail in the comment blocks in List.h. Some of these functions may be challenging to implement. It will be very much worth your while to study the Queue and Stack ADT implementations in C++ posted on the webpage. For instance, function `join()` in the Queue ADT is very similar to `concat()` in List. It is very common in C++ programs to overload built in operators. In this project, you will overload `operator<<()` (stream insertion), `operator==()` (compare for equality) and `operator=()` (assignment). These can be difficult to get right without some guidance. All of these operators are overloaded in the Queue and Stack examples. See the reference pages

https://en.cppreference.com/w/cpp/language/operators

and the following article

https://www.cplusplus.com/articles/y8hv0pDG/

for information on the relationship between the copy constructor and the assignment operator in C++.

What to turn in

Once the List is constructed and tested, building the client Shuffle.cpp should pose no major difficulties for most students. Submit the following 6 files to your pa6 directory on git.ucsc.edu.

`README.md`	Written by you, a catalog of submitted files and any notes to the grader
`Makefile`	Provided, alter as you see fit
`List.h`	Provided, **do not alter**
`List.cpp`	Written by you, most of the work in this assignment
`ListTest.cpp`	Written by you, a test harness for your List
`Shuffle.cpp`	Written by you, the top level client for this project

As usual, do not turn in executable files, binaries, or anything not listed above. Start early, ask plenty of questions, and submit your project by the due date. Here are some more links to important C++ topics.

Pointers vs. References Operator Overloading Standard Exception Classes

In addition, the recommended textbook *Data Abstraction & Problem Solving with C++* (6th edition) by Carrano and Henry (Pearson 2013 ISBN 10: 0-13-292372-6) contains a nice illustration of how to implement an ADT in C++ on pages 31-37, along with many other examples.

More on Riffle Shuffles

To see more sophisted methods for computing the order of a riffle shuffle, visit the On-Line Encyclopedia of Integer Sequences and search the following partial sequence: 2, 4, 3, 6, 10, 12, 4, 8, 18, 6, 11,…