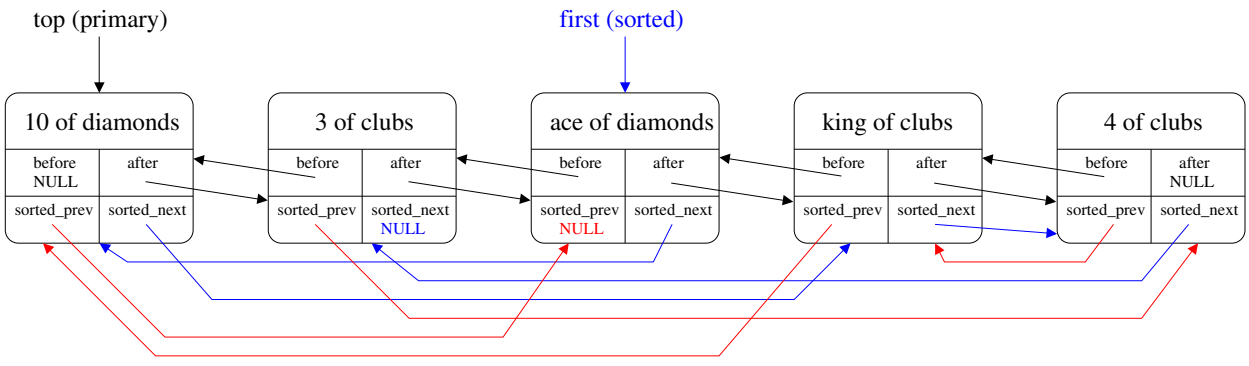
CSCI-1200 Data Structures — Fall 2013
Homework 5 — Linked Playing Cards

In this assignment you will manipulate a linked list of nodes that store playing cards of 4 different suits: clubs ♣, diamonds ♢, hearts ♡, and spades ♠; and 13 different face values: ace, 2-10, jack, queen, and king. In physical card games the deck of cards is commonly randomized by first *cutting* the deck roughly in half, and then *shuffling* the two piles into a single stack by interleaving or alternating a card or cards from one pile with each card or cards from the other pile. This cut & shuffle process is repeated several times. Once the deck is sufficiently shuffled, the cards are distributed to the other players by *dealing* the cards from the top of the stack to each player, one at a time, sequentially, until the required number of cards have been distributed. Slight imperfections in splitting the deck in half and variation in the number of cards selected as the piles are interleaved add sufficient variation into the process to achieve randomness in the resulting deck that allows games of chance to be played without bias to one or more players.

Your task for this assignment is to perform idealized versions of these core card deck operations on a linked list structure representing the ordered deck of cards.

Provided Framework

We provide a few small classes and the framework for testing the functionality of the different card deck operations. The `PlayingCard` class represents a single card (suit & face value) and includes helper functions to print and compare two cards for sorting (using a high to low ordering grouped by suit that is helpful for many card games). The `Node` class holds a single card and stores pointers/links to other cards in the deck. Interestingly, the linked structure actually maintains two different doubly-linked traversal orders. The *primary* traversal is the physical ordering of the cards in the deck (or the order they are received by each player when the cards are dealt). Note: In some card games the order that the cards are received is part of game play. A secondary ordering, the *sorted* order, is also a doubly-linked structure stored within the same set of `Node`s. This second ordering is initially disconnected (everything set to NULL), but when the `SortHand` function is called, the blue and red links below are constructed. This additional ordering represents how a card player might re-organize his/her cards, first by suit, alternating red/black suits, and then by face value from ace (high) down to 2. This data structure allows us to store and traverse both orderings. Here is a diagram showing the relationship between `Node`s and `PlayingCard`s:



First carefully study the provided code files. `main.cpp` contains the testing infrastructure. You are not to edit this file except to uncomment the test cases as you progress through the assignment and to add your own test cases in the specified function. As you read through the provided helper functions and the different test cases, and study the provided sample output, you should get a feel for the expectations of the operations on this data. You will write the 10 missing functions from these examples. (Part of your task for this homework is to deduce the exact function prototypes.)

Missing Functionality

Several of the functions you will write: `DeckSize`, `PrintDeckSorted`, `SamePrimaryOrder`, and `ReversePrimaryOrder`, are helper functions that need read-only access to the linked list structure. The latter two functions are used in inspecting the output of the perfect shuffle described below.

Two of the functions you will write: `CopyDeck` and `DeleteAll`, ask you to allocate and deallocate node objects associated with those variables. The other functions you write should not create or destroy node objects — rather, you will be re-arranging the links between nodes.

The remaining functions: `CutDeck`, `Shuffle`, `Deal`, `SortHand`, and `PrintDeckSorted`, carry out the operations necessary to model physical card shuffling, dealing, and card sorting. `CutDeck` splits a single deck into similar sized sub-decks (the top and bottom of the list):

```
deck:  A♣2♣3♣4♣5♣6♣7♣8♣
top:  A♣2♣3♣4♣
bottom:  5♣6♣7♣8♣
```

`Shuffle` takes those two smaller pieces and interleaves the cards one or a few cards at a time to form a single deck again, but with a different order than the original. For the main homework you will implement a perfect `CutDeck` and perfect one-by-one `Shuffle` function. These shuffle results are called a *perfect out-shuffle* and *perfect in-shuffle* depending on whether the top deck or the bottom deck contributes the first card to the resulting deck.

```
out shuffle:  A♣5♣2♣6♣3♣7♣4♣8♣
in shuffle:  5♣A♣6♣2♣7♣3♣8♣4♣
```

Interestingly, if we repeatedly perform perfect shuffles, we will restore the data to the original order. The number of shuffles required depends on the number of cards in the original deck, and also whether we are performing out- or in- shuffles. See also: http://en.wikipedia.org/wiki/Faro_shuffle

Next, we can `Deal` all or some of the deck to a specified number of players. We specify the number of cards each person should receive. The cards are passed to the players one-at-a-time in sequence.

```
deck:  A♣2♣3♣4♣5♣6♣7♣8♣9♣T♣J♣Q♣K♣A♢2♢3♢4♢5♢6♢7♢8♢9♢T♢J♢Q♢K♢A♡2♡3♡4♡5♡
6♡7♡8♡9♡T♡J♡Q♡K♡A♠2♠3♠4♠5♠6♠7♠8♠9♠T♠J♠Q♠K♠

west:  A♣5♣9♣K♣4♢8♢Q♢3♡7♡J♡2♠6♠T♠
north:  2♣6♣T♣A♢5♢9♢K♢4♡8♡Q♡3♠7♠J♠
east:  3♣7♣J♣2♢6♢T♢A♡5♡9♡K♡4♠8♠Q♠
south:  4♣8♣Q♣3♢7♢J♢2♡6♡T♡A♠5♠9♠K♠
```

Once a player has received their hand, the `SortHand` function can be called to initialize the node pointers for the secondary sorted ordering shown on the earlier diagram.

```
west (sorted):  T♠6♠2♠Q♢8♢4♢A♣K♣9♣5♣J♡7♡3♡
north (sorted):  J♠7♠3♠A♢K♢9♢5♢T♣6♣2♣Q♡8♡4♡
east (sorted):  Q♠8♠4♠T♢6♢2♢J♣7♣3♣A♡K♡9♡5♡
south (sorted):  A♠K♠9♠5♠J♢7♢3♢Q♣8♣4♣T♡6♡2♡
```

The provided `PlayingCard` class includes the logic for comparing two cards to determine which one comes first in the sorted order. Note: We will not use an STL sort function to construct the sorted order. Instead we recommend you implement insertion sort, a simple-to-implement, but somewhat inefficient sort routine.

Extra Credit

For extra credit, explore variants on the shuffling model to mimic the randomness that naturally occurs in physical shuffling. Specifically, the final arguments to the `CutDeck` and `Shuffle` functions indicate whether we are asking for a *perfect* or *random* method. When the deck is cut randomly, the two piles should be roughly the same size, but not necessarily exactly equal. When the shuffle is random (not perfect), the cards will sometimes be interleaved in pairs, triples, or other small clusters of cards. We recommend the *MersenneTwister* library as a robust source of randomness for C++ programs.

How many random, not-quite-perfect cut & shuffle actions are sufficient to fully randomize a deck of 52 cards? Recent theoretical research results tell us that number is about 7!

http://en.wikipedia.org/wiki/Shuffling

If we shuffle fewer than 7 times, the possible final positions for a specific card in the resulting deck or the possible neighboring cards for that element are limited or biased. Devise a scheme to measure the randomness of shuffling through simple statistics Use simulated results (over many, many trials) to support the theoretical proof. Collect data and summarize your conclusions in your `README.txt` file.

Additional Requirements, Hints and Suggestions

You will not use lists or vectors or iterators or other STL containers in this assignment. Also, you should not use the the STL sort algorithm. Instead, you will be manipulating the low-level custom `Node` objects, and the pointers that connect each `Node` to other `Node`s in the structure. You do not need worry about the efficiency of your implementation or the efficiency of your sorting algorithm. A simple insertion sort or bubble sort algorithm is quite efficient for small datasets. Some of your other methods may also seem a little inefficient. That's ok for this homework – the goal is to practice linked node pointer manipulation.

The code you write will exist as standalone functions that pass data to and from other functions. We will not implement a class structure to manage the functionality of this homework. You should work on the assignment step by step, uncommenting each provided test case, and creating your own additional test cases to make sure your code is fully debugged. You should not modify the provided code except where indicated, as we will be compiling and testing your submitted files with different portions of the solution file. To earn full credit on this homework, your code must also pass the memory error and memory leak checks of Valgrind/Dr. Memory.

A Note on the Program's Text Output

To print fancy suit characters, the provided code uses UTF, the Universal Character Set (UCS) Transformation Format. These characters do not display correctly in all web browsers, console terminals, code editors, or file viewers. The executable for this homework does not require any command line arguments and the program will, by default, output these special characters. If the default (no command line arguments) output looks like gibberish in your usual development environment, terminal console, or file viewer you can provide the optional command line argument "`no_outline_symbols`" (solid black versions of all suit symbols) *or* "`no_symbols`" (uses '`C`', '`D`', '`H`', and '`S`' for the suits).

Submission

Use good coding style when you design and implement your program. Be sure to make up new test cases and don't forget to comment your code! Please use the provided template `README.txt` file for any notes you want the grader to read. **You must do this assignment on your own, as described in the "Academic Integrity for Homework" handout. If you did discuss the problem or error messages, etc. with anyone, please list their names in your `README.txt` file.** When you are finished please zip up your files exactly as instructed for the previous assignments and submit it through the course webpage.