Program 3: Card Cipher
CS241 Fall 2014
Due on Friday Nov 7

Not accepted after Tuesday Nov 11

Announcements/Changes

- An Ace in the hearts deck does not mean that the clubs deck should be rotated a second time.

Overview

You may work on this assignment individually, or with one other person in CS 241. The goals of this assignment are to:

1. Become more familiar with using stacks and queues.
2. Learn to write your own stacks and queues
3. Give you practice in selecting the correct data structures for a problem.

Program Specifications

In this assignment you will be writing an encryption/decryption program that uses a deck of cards to encipher and decipher messages. If two people were to each have a deck of cards, with the cards in identical order, they would be able to transmit encoded messages to one another. This following algorithm draws inspiration from Neil Stephenson's *Cryptonomicon* and the WWII German encryption machine, the Enigma.

At its heart, this program will use a stream of integer keys to encrypt or decrypt an ASCII messsage. Each key is used to encipher or decipher a single ASCII character. To encipher a character, simply add the key to the character and take the modulus of 128 (to ensure the character stays in ASCII range). To decipher a character, first check to see if the character is less than the key. If it is, keep adding 128 to the character until it is larger than the key. Then subtract the key from the character. It is important to note that in C++ the *char* data type is represented by a signed byte. Adding a key to a signed byte may cause overflow that turns its value negative. This will cause your program to produce strange results. To ensure that this doesn't happen, always use the *unsigned char* data type when enciphering or deciphering an ASCII character.

KeyStream is the class that generates the stream of integers and is the heart of the encryption algorithm. Given a shuffled deck of cards, the first step of the algorithm is to divide the deck into four separate decks by suit. These decks are then placed in the following order from left to right: clubs, spades, diamonds, and hearts. The top cards of these four decks determine the current encryption key. Each card is converted to an integer by multiplying its rank (A=>1, 2=>2, …, 10=>10, J=>11, Q=>12,

K=>13) by its suit (hearts=>1, clubs=>2, diamonds=>3, spades=>4). The integer values of these cards are summed and then ten is subtracted (see if you can figure out why) to generate the key.

To generate the next key, the decks must be modified. The algorithm to modify the decks is a little complex. To start, the top card of the left most deck (the clubs deck) is moved to the bottom of the deck. This procedure is called rotating the clubs deck. If the new top card is an Ace, the deck immediately to right (the spades deck) is also rotated. If the new top card on the spades deck is an Ace, then the deck to its immediate right (the diamonds deck) is rotated and the cards on the deck to its left (the clubs) are removed until the top card matches the deck on the right (the diamonds). These removed cards are place at bottom of the deck in reverse order from how they were removed.

The general algorithm is as follows. Rotate a deck by moving its top card to the bottom. If the new top card is an Ace, rotate the deck to the right, and reverse the deck to the left until its top card matches the deck to the right of the Ace (following the procedure above). If the Ace deck has no deck to its left, don't reverse any deck. If the Ace deck has no deck to its right, use the first deck's (the clubs deck's) rank card as the rank to reverse the deck to the left to. An Ace in the hearts deck does not mean that the clubs deck should be rotated a second time. See the example below (I've removed some cards from the decks to reduce the complexity of the example):

4	J	5	9
A	A	3	6
J	2	Q	8
8	3	A	5
3	6	10	4
9	7	J	A
K	Q	8	K
C	**S**	**D**	**H**

To generate the next key we need to rotate the first deck which puts the decks in this state:

A	J	5	9
J	A	3	6
8	2	Q	8
3	3	A	5
9	6	10	4
K	7	J	A
4	Q	8	K
C	**S**	**D**	**H**

Because the deck we rotated now has an Ace on the top, we need to rotate its right neighbor:

A	A	5	9

J	2	3	6
8	3	Q	8
3	6	A	5
9	7	10	4
K	Q	J	A
4	J	8	K
C	**S**	**D**	**H**

Because the spades deck has an Ace as its top card, we now need to rotate its right neighbor:

A	A	3	9
J	2	Q	6
8	3	A	8
3	6	10	5
9	7	J	4
K	Q	8	A
4	J	5	K
C	**S**	**D**	**H**

and reverse its left neighbor to the rank its right neighbor's new top card (3D).

3	A	3	9
9	2	Q	6
K	3	A	8
4	6	10	5
8	7	J	4
J	Q	8	A
A	J	5	K
C	**S**	**D**	**H**

And we are done. These four top cards now form the next key: (3 * 2 + 1 * 4 + 3 * 3 + 9 * 1) – 10 = 18. These steps occur for every key generated.

In order for two agents to send messages, their card decks must initially start out in the same order. In a computer program we could read in a file that represents the starting deck, or we could take advantage of a pseudorandom number generator (PRNG) to create a shuffled deck where the cards are in the same order every time. A PRNG takes a seed number and then generates a stream of numbers that appear random. However, if you give the PRNG the same seed again, it will generate the exact same stream of numbers. This mechanism allows computer programs to appear random while actually being deterministic.

For this program, we will be using the PRNG approach. Our program needs a way to shuffle cards, such that, given a parameter it always produces a deck of cards in the same order. To do this we will start with our deck of cards in a specific order: A-K in each suit; with the suits ordered hearts, clubs, diamonds, and spades. The deck should then be shuffled seven times using STL's *random_shuffle* function. To ensure that two agents decks end up shuffled the exact same way, you must seed the random number generator prior to shuffling using the *srand* function.

This project requires you to write two programs, both of which will use the *KeyStream* class. The first program, *encrypt,* takes two command line arguments. The first is an integer seed for the random number generator and the second is the name of the file to encrypt. This program will print the encrypted file to the console. The second program, *decrypt,* also takes two command line arguments. The first is an integer seed for the random number generator and the second is the name of the file to decrypt. This program will print the decrypted file to the console.

A program of this size will require several data structures, including stacks and queues. As part of this program you will develop your own linked-node implementation of a stack and a circular array implementation of a queue. While developing Card-Cipher you may use STLs stack and queue, but your finished program must use *LCQueue* and *LCStack* when stacks and queues are called for. You may use STL's vector throughout this program, when appropriate.

To test this program, you may want to use some large text files. Luckily, there is a wonderful repository of text files online at the Project Gutenberg webpage (www.gutenberg.org). This website maintains an archive of public domain books in several different formats, including ASCII text. Beware, however, because it also contains texts files stored in UTF-8 format. Your program is not designed to handle this. To vouchsafe against misuse of *encrypt* or *decrypt*, these program should check each character they read to ensure that it is ASCII (ASCII characters have values between 0-127, whereas UTF-8 has characters that exceed 127). If your *encrypt* or *decrypt* programs encounters a non-ASCII character, it should print a message to standard error and exit.

I have provided starter header files for you on the web page; you may add member variables and methods but do not change the ones provided. I have also provided the ordering of cards given the seed seven, the first 30,000 keys generated by the key stream given a seed of seven, and Lewis Carroll's "The Hunting of the Snark" encrypted using seed seven. **Your output should match this exactly.** You can check the quality of the match using the program *diff*. Remember, you can redirect the output of running a program on the command-line using the > operator. Ex: *p3 7 snark.txt > my_output.txt*.

Milestones

Milestone 1 (due in class on Monday Oct 27): You must have a working circular array implementation of LCQueue.

Milestone 2 (due in lab on Monday Nov 3): You must have a working linked node implementation of LCStack.

Note, completing these milestones does not mean you are on track to finishing this program. You should be working on Card Cipher in parallel to working on *LCQueue* and *LCStack*.

How to hand in your program

Create a directory for your project with the following naming scheme: username.project (e.g., granger_s.p3). Create a compressed tarball of this directory and submit it using Moodle.

If you are working in pairs, be sure to put both of your names in the comment section of **each** file.

Grading

Your programs will be run and graded on correctness given a variety of input parameters. They will also be graded on the correct use of data structures. Furthermore, for each assignment there will be a list of proper programming techniques that you will be graded on. For each assignment you will be responsible for ensuring that you practice the new techniques as well as the techniques outlined in all previous assignments.

Proper Programming Techniques

- Do not leak any memory.
- Be sure to follow the programming technique guides from the previous assignments. Failure to do so will result in lost points.