Week 2 Section

This week's section handout has practice with ADTs: Grids, Vectors, Stacks, Queues, Sets, and Maps. We'll cover Sets and Maps on Friday.
Suggested Problems: 2, 4, and 6

1. Collection Mystery - Stacks and Queues

Write the output produced by the following function when passed each of the following stacks. Note that stacks and queues are written in *front to back* order, with the oldest element on the left side of the queue/stack.

```
void collectionMystery(Stack<int>& s) {
    Queue<int> q;
    Stack<int> s2;
    while (!s.isEmpty()) {
        if (s.peek() % 2 == 0) {
            q.enqueue(s.pop());
        } else {
            s2.push(s.pop());
        }
    }
    while (!q.isEmpty()) {
        s.push(q.dequeue());
    }
    while (!s2.isEmpty()) {
        s.push(s2.pop());
    }
    cout << s << endl;
}
```

Stacks:	Output:
`{1, 2, 3, 4, 5, 6}`	______________________
`{42, 3, 12, 15, 9, 71, 88}`	______________________
`{65, 30, 10, 20, 45, 55, 6, 1}`	______________________

Thanks to CS106B and X instructors and TAs for contributing problems on this handout.

2. Mirror - Grid

Write a function **`mirror`** that accepts a reference to a `Grid` of integers as a parameter and flips the grid along its diagonal. You may assume the grid is square; in other words, that it has the same number of rows as columns. For example, the grid below at left would be altered to give it the new grid state at right:

```
{{ 6,  1,  9,  4},            {{6, -2, 14, 21},
 {-2,  5,  8, 12},             {1,  5, 39, 55},
 {14, 39, -6, 18},     -->     {9,  8, -6, 73},
 {21, 55, 73, -3}}             {4, 12, 18, -3}}
```

Bonus: How would you solve this problem if the grid were not square?

3. CrossSum - Grid

Write a function named **`crossSum`** that accepts three parameters - a reference to a `Grid` of integers, and two integers for a row and column - and returns the sum of all numbers in the row/column cross provided. For example, if a grid named `g` stores the following integers:

```
   col    0  1  2
row
 0     {{1, 2, 3},
 1      {4, 5, 6},
 2      {7, 8, 9}}
```

Then the call of `crossSum(g, 1, 1)` should return (4+5+6+2+8) or `25`. You may assume that the row and column passed are within the bounds of the grid. Do not modify the grid that is passed in.

4. RemoveConsecutiveDuplicates - Vector

Write a function named **`removeConsecutiveDuplicates`** that accepts as a parameter a reference to a `Vector` of integers, and modifies it by removing any consecutive duplicates. For example, if a vector named `v` stores `{1, 2, 2, 3, 2, 2, 3}`, the call of `removeConsecutiveDuplicates(v);` should modify it to store `{1, 2, 3, 2, 3}`.

5. Stutter - Queue

Write a function named **`stutter`** that accepts a reference to a queue of integers as a parameter and replaces every element with two copies of itself. For example, if a queue named `q` stores `{1, 2, 3}`, the call of `stutter(q);` should change it to store `{1, 1, 2, 2, 3, 3}`.

6. SplitStack - Stack and Queue

Write a method **`splitStack`** that takes a stack of integers as a parameter and splits it into negative and non-negative numbers. The numbers in the stack should be rearranged so that all negatives appear on the bottom of the stack and all the non-negatives appear on the top. In other words, if after this method is called you were to pop number off the stack, you would first get all the nonnegative numbers and then get all of the negative numbers.

Example: passing `{4, 0, -1, 5, -6, -3, 2, 7}` could change it to `{-3, -6, -1, 7, 2, 5, 0, 4}`. It does not matter what order the numbers appear in as long as all the negatives appear lower in the stack than all the non-negatives. You may use a single queue as auxiliary storage.

7. Rarest - Map (covered on Friday of Week 2)

Write a function named **`rarest`** that accepts a reference to a `Map` from strings to strings as a parameter and returns the value that occurs least frequently in the map. If there is a tie, return the value that comes earlier in ABC order. For example, if a variable called `map` contains the following elements:

```
{"Alyssa":"Harding", "Char":"Smith", "Dan":"Smith", "Jeff":"Jones",
"Kasey":"Jones", "Kim":"Smith", "Morgan":"Jones", "Ryan":"Smith",
"Stef":"Harding"}
```

Then a call of `rarest(map)` would return `"Harding"` because that value occurs 2 times, fewer than any other. Note that we are examining the values in the map, not the keys. If the map passed is empty, throw a string exception.

8. Twice - Sets (covered on Friday of week 2)
Write a function named **`twice`** that takes a vector of integers and returns a set containing all the numbers in the vector that appear exactly twice.

Example: passing {1, 3, 1, 4, 3, 7, -2, 0, 7, -2, -2, 1} returns {3, 7}.

Bonus: do the same thing, but you are not allowed to declare any kind of data structure other than sets.

9. FriendList - Maps (covered Friday of Week 2)

Write a function named **`friendList`** that takes in a file name and reads friend relationships from a file and writes them to a `Map`. **`friendList`** should return the populated `Map`. Friendships are bi-directional, so if Abby is friends with Barney, Barney is friends with Abby. The file contains one friend relationship per line, with names separated by a single space. You do not have to worry about malformed entries.

If an input file named buddies.txt looked like this:

```
Barney Abby
Abby Clyde
```

Then the call of `friendList("buddies.txt")` should return a resulting map that looks like this:

```
{"Abby":{"Barney", "Clyde"}, "Barney":{"Abby"}, "Clyde":{"Abby"}}
```

```
Map<string, Vector<string> > friendList(String filename) { ...
```