Practice Midterm Exam III

This exam is closed-book and closed-computer. You may have a double-sided, 8.5" × 11" sheet of notes with you when you take this exam. You may not have any other notes with you during the exam. You may not use any electronic devices (laptops, cell phones, etc.) during the exam. Please write all of your solutions on this physical copy of the exam.

Unless otherwise indicated as part of the instructions for a specific problem, comments will not be required on the exam. Uncommented code that gets the job done will be sufficient for full credit on the problem. On the other hand, comments may help you get partial credit if they help us determine what you were trying to do. You do not need to worry about efficiency unless a problem states otherwise. You don't need to prototype helper functions you write, and you don't need to explicitly `#include` libraries you want to use.

This exam is "self-contained" in the sense that if you're expected to reference code from the lectures, textbook, assignments, or section handouts, we'll explicitly mention what that code is and provide sufficient context for you to use it. You should have received a C++ reference sheet before starting this exam with a refresher of common functions and types.

This practice exam is completely optional and does not contribute to your overall grade in the course. However, we think that taking the time to work through it under realistic conditions is an excellent way to prepare for the actual exam and figure out where you need to focus your studying.

You have three hours to complete this exam. There are four questions and 32 total points.

<u>You can do this.</u> Best of luck on the exam!

Problem One: Containers I (8 Points)

Suppose that you own a pet store and want to help customers find their ideal pet. To do so, you create a list of all the pets in the store, along with all of the adjectives which best describe them. For example, you might have this list of pets and their adjectives:

Ada	Happy, Loving, Fuzzy, Big, Tricksy
Babbage	Loving, Tiny, Energetic, Quiet
Lucy	Happy, Loving, Big
Larry	Happy, Fuzzy, Tiny, Tricksy
Abby	Loving, Big, Energetic
Luba	Happy, Loving, Tiny, Quiet, Tricksy
Mitzy	Fuzzy, Big, Energetic, Quiet

If a customer gives you a list of adjectives, you can then recommend to them every pet that has all of those adjectives. For example, given the adjectives "Happy" and "Tricksy," you could recommend Ada, Larry, and Luba. Given the adjective "Fuzzy," you could recommend Ada, Larry, and Mitzy. However, given the adjectives "Energetic," "Quiet," and "Fuzzy," and "Tiny," you could not recommend any pets at all.

Write a function

```
HashSet<string> petsMatching(const HashMap<string, HashSet<string>>& adjectiveMap,
                             const Vector<string>& requirements);
```

that accepts as input a `HashMap<string, HashSet<string>>` associating each pet with the adjectives that best describe it, along with a customer's requested adjectives (represented by a `Vector<string>`), then returns a `HashSet<string>` holding all of the pets that match those adjectives.

There might not be any pets that match the requirements, in which case your function should return an empty set. You can assume that all adjectives have the same capitalization, so you don't need to worry about case-sensitivity. Also, if a client does not include any adjectives at all in their requirements, you should return a set containing all the pets, since it is true that each pet matches all zero requirements. Finally, your function should not modify any of the input parameters.

Feel free to tear out this page as a reference, and write your solution on the next page.

```
HashSet<string> petsMatching(const HashMap<string, HashSet<string>>& adjectiveMap,
                             const Vector<string>& requirements) {
```

(Extra space for your answer to Problem One, if you need it.)

Problem Two: Containers II (8 Points)

Recall from lecture that a *shrinkable word* is a word that can be reduced down to a single letter by removing one letter at a time, at each step leaving a valid English word.

In lecture, we wrote a function `isShrinkableWord` that determined whether a given string was a shrinkable word. Initially, this function just returned `true` or `false`. This meant that if a word was indeed shrinkable, we would have to take on faith that the word could indeed be reduced down to a single letter.

To help explain *why* a word was shrinkable, our second version of the function additionally produced a `Stack<string>` showing the series of words one would go through while shrinking the word all the way down to a single letter. (In reality, our lecture example used a `Vector<string>` rather than a `Stack<string>`, but a `Stack<string>` is actually a better choice.) Let's call a sequence of this sort a ***shrinking sequence***.

However, let's suppose that you're *still* skeptical that the `Stack<string>` produced by the `isShrinkableWord` function actually is a legal shrinking sequence for the word. To be more thoroughly convinced that a word is shrinkable, you decide to write a function that can check whether a given `Stack<string>` is indeed a shrinking sequence for a given word.

Write a function

```
bool isShrinkingSequence(const string& word,
                         const Lexicon& english,
                         Stack<string> path);
```

that accepts as input a word, a `Lexicon` containing all words in English, and a `Stack<string>` containing an alleged shrinking sequence for that word, then returns whether the `Stack<string>` is indeed a legal shrinking sequence for that word. For example, given the word `"pirate"` and the stack

```
pirate
irate
rate
rat
at
a
```

Your function would return `true`. However, given any of these stacks:

```
pirate
irate
rate
ate
te
e
```

```
avast
vast
vat
at
a
```

```
pirate
rate
at
```

Your function would return `false` (the first stack is not a shrinking sequence because `"te"` and `"e"` are not words; the second is a legal shrinking sequence, but not for the word `"pirate"`; the third is not a shrinking sequence because it skips words of length 1, 3, and 5; and the fourth is not a shrinking sequence because it contains no words at all).

You can assume that `word` and all the words in `path` consist solely of lower-case letters.

Feel free to tear out this page as a reference, and write your solution on the next page.

```
bool isShrinkingSequence(const string& word,
                         const Lexicon& english,
                         Stack<string> path) {
```

(Extra space for your answer to Problem Two, if you need it.)

Problem Three: Recursion I (8 Points)

(This excellent problem by Eric Roberts.)

In the early part of the 20th century, there was considerable interest in both England and the United States in simplifying the rules used for spelling English words, which has always been a difficult proposition. One suggestion advanced as part of this movement was the removal of all doubled letters from words. If this were done, no one would have to remember that the name of the Stanford student union is spelled "Tresidder," even though the incorrect spelling "Tressider" occurs at least as often. If doubled letters were banned, everyone could agree on "Tresider."

Write a ***recursive*** function

```
string removeDoubledLetters(const string& str);
```

that takes a string as its argument and returns a new string with any consecutive substring consisting of repeated copies of the same letter replaced by a single copy letter of that letter. For example, if you call

```
removeDoubledLetters("tresidder")
```

your function should return the string `"tresider"`. Similarly, if you call

```
removeDoubledLetters("bookkeeper")
```

your function should return `"bokeper"`. And because your function compresses strings of multiple letters into a single copy, calling

```
removeDoubledLetters("zzz")
```

should return `"z"`.

In writing your solution, you should keep the following points in mind:

- Your function should not try to consider the case of the letters. For example, calling the function on the name `"Lloyd"` should return the argument unchanged because `'L'` and `'l'` are different letters.
- Your function must be purely recursive and may not make use of any iterative constructs such as `for` or `while`.

```
string removeDoubledLetters(const string& str) {
```

(Extra space for your answer to Problem Three, if you need it.)

Problem Four: Recursion II (8 Points)

Your job in this problem is to write a function

```
HashSet<string> balancedStringsOfLength(int n);
```

that accepts as input a nonnegative number `n`, then returns a `HashSet<string>` containing all strings of exactly `n` pairs of balanced parentheses.

As examples, here is the one string of one pair of balanced parentheses:

```
()
```

Here are the two strings of two pairs of balanced parentheses:

```
(())    ()()
```

Here are all five strings of three pairs of balanced parentheses:

```
((()))    (()())    (())()    ()(())    ()()()
```

And here are the fourteen strings of four pairs of balanced parentheses:

```
(((())))          ((()()))          ((()))()          ()((()))
((()()))          ((())))()         ()((()))          ()()()()
((())())          ((()))()          ()(()())
(()((())))        (()))(())         ()(())()
```

As a hint, you might find the following observation useful: any string of $n > 0$ pairs of balanced parentheses can be split apart as follows:

`(` *some-string-of-balanced-parentheses* `)` *another-string-of-balanced-parentheses*

You might find it useful to try splitting apart some of the above strings this way to see if you understand why this works.

Write your solution on the next page, and feel free to tear out this page as a reference.

```
HashSet<string> balancedStringsOfLength(int n) {
```

(Extra space for your answer to Problem Four, if you need it.)