Lab 5 Stack Client

Goal

In this lab you will complete an application that uses the stack ADT to sort numbers.

Resources

- Chapter 5: Stacks

In `javadoc` directory
- *`StackInterface.html`*—Interface documentation for the interface `StackInterface`

Java Files

- *`StackInterface.java`*
- *`StackSort.java`*
- *`VectorStack.java`*

Stack Sort

Pieces of the `StackSort` class already exist and are in *`StackSort.java`*. Take a look at that code now if you have not done so already. Also before you start, make sure you are familiar with the methods available to you in the `VectorStack` class (check *`StackInterface.html`*).

Step 1. Compile the classes `StackSort` and `VectorStack`. Run the `main` method in `StackSort`.

Checkpoint: If all has gone well, the program will run. It will create arrays of various sizes and print out the result of `StackSort` method. At the end, the program will ask you to enter an integer value. It will use the value to create and then sort an array of that size. Enter any value. The sorted arrays as reported by the program should all be empty. Our first goal is to get values into a stack and then move them to the result array.

Step 2. Create a new `VectorStack<Integer>` and assign it to `lowerValues`.

Step 3. Create a new `VectorStack<Integer>` and assign it to `upperValues`.

Step 4. Using a loop, scan over the values in the argument array `data` and push them onto the `upperValues` stack.

Step 5. Using a loop, pop all the values from the `upperValues` stack and place them into the array `result`.

Checkpoint: Compile and run the program. Again it should run and ask for a size. Any value will do. This time, you should see results for each of the calls to the StackSort method. The order that values are popped off the stack should be in the reverse order that they were put on the stack. If all has gone well, you should see the values in reverse order in the results array. We will now complete the StackSort method

Step 6. Inside the loop that scans over the `data` array, we need to move the data between the two stacks before we push the value. Refer to the prelab exercise to complete the body of the loop.

Step 7. Before the loop that pops the data values from the `upperValues` stack, we need to move any data values from the `lowerValues` stack. Refer to the prelab exercise to implement this loop.

Checkpoint: Compile and run the program. The output of the StackSort method should be the original arrays in sorted order. If not, debug until the results are correct.

Post-Lab Follow-Ups

1. It should not matter into which stack the values from the data array are pushed. Change the stack that your implementation pushes the values and verify that the output is still correct.

2. Write a program that determines if an input string is a valid equation in unary representation with addition. For example:

   a. "111+1111=11111+1+1" is the statement that 3+4=5+1+1 and is valid.
   b. "11+1=1111" is not valid because 2+1 is not equal to 4.
   c. "11+11+1=11=111" is not valid because we only allow one equal sign
   d. "111-1=11" is not valid because we only allow the characters '1', '+' and '='.

   You have one small restriction. You are not allowed to use the built in arithmetic of the computer, but instead are only allowed to match characters. Furthermore, you are only allowed to scan over characters once. This can be done with a single stack. The general idea is to scan over the string and push '1's until you see the '='. After that match the '1's by poping values off of the stack. You can decide validity by tracking if the stack is empty. If the stack empties and there are still '1's in the input, it is not valid. If the input has all been used, but the stack is not empty, it is not valid.

3. Write a program that solves the previous problem extended to allow multiple occurences of '='. For example, "111+11=1111+1=11111" is a valid equation. To do this you will need three stacks. One stack is a long term memory, one stack is working memory, and the final stack allows duplication. Push '1's on the long term stack until you get an '='. Each time you read an equal, you will pop each '1' off of the long term stack and push a '1' on both of the other two stacks. Once this is done, pop all the '1's off of the duplicate stack and push them back onto the long term memory stack. Match the '1's in the input with the working memory stack.

4. Write a program that will determine if an input string is a palindrome. A palindrome is a string that reads the same forwards and backwards. For example, "Madam, I'm Adam." and "Able I was ere I saw Elba." are two classic palidromes. A stack is a natural structure for allowing us to determine if a string is a palindrome since characters come off the the stack in the reverse order that they are pushed onto the stack. Take each alpha character in the string (ignore punctuation and spaces) converted to lower case and push it on two different stacks called `reversed` and `temporary`. Pop each of the characters off of the `temporary` stack and push them on a third stack called `originalOrder`. You can now pop characters from the `reversed` and `originalOrder` stacks and compare them. If they all match, then you have palindrome.

5. Write a program which is similar to the previous program, but determines how close a string is to being a palidrome by counting the number of mismatches. A palidrome has zero mismatches.

6. Write a program that will determine if one input string is a subsequence of a second input string. (We looked for a longest common subsequence in Lab 2 on Bag clients.) In a subsequence, all the characters in the first string will also be in the second string in the same order. For example: "abc" is a subsequence of "qzabc". While the characters must be in the same order, they do not have to be consecutive. For example: "abc" is also a subsequence of "aaqbzcw". We will use two stacks `one` and `two`. Two determine if the first string is a subsequence of the second string, push each of the charcters from the first input string onto stack

`one` and then push the characters of the second input string onto stack `two`. Peek at the tops of the two stacks and if they match pop both. If they don't match pop stack `two`. If stack one becomes empty, then we have found a subsequence. If stack `two` becomes empty and there are still characters on stack `one`, it is not a subsequence.