Lab 5 Stack Client

Goal

In this lab you will complete an application that uses the Abstract Data Type (ADT) stack.

Resources

- Chapter 5: Stacks

In `javadoc` directory

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

Java Files

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

Introduction

In computer science, one of the important basic structures is the stack. It is of both theoretical and practical use. In its simplest form it has three operations: push, pop, and empty. Push places a value on the top of the stack. Pop removes the top value from the stack. Empty is a test to determine if the stack has any values in it. Some specifications give a fourth operation called peek (or top). Peek will return the top value on the stack but leaves the number of items unchanged. Strictly speaking, peek is unnecessary because a pop followed by a push will mimic its operation. Before continuing the lab you should review the material in Chapter 5. In particular, review the documentation of the interface *`StackInterface.java`*.

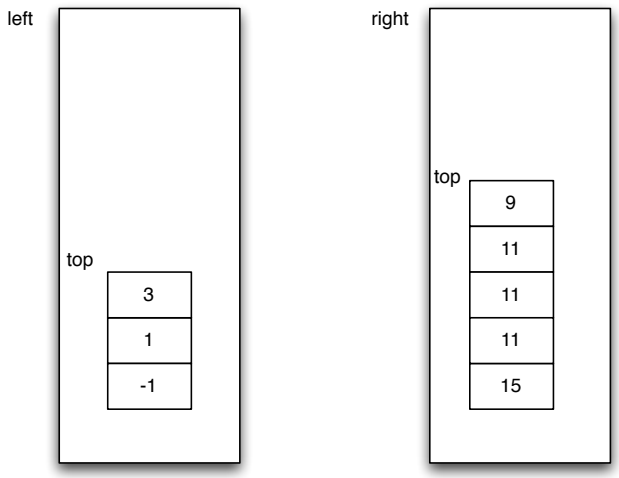
In theoretical computer science, one of the problems of interest is recognizing words in a language. In this context, a language is a set of words that follow some pattern. For example, one language is all the words from the alphabet {0, 1} that have equal numbers of zeros and ones. The word 001011 is in the language, but the word 00111 is not. There are a number of primitive models of computation that have different abilities. One kind of model is a machine called a pushdown automata (PDA). It has a finite control (program) and a single stack that it can use for memory. While fairly powerful, a PDA does have some surprising limits. For example, while a PDA can recognize words of the form 0^n1^n, it cannot recognize $0^n1^n0^n$. Modern computer languages are often recursively defined by a grammar, which can be recognized by a PDA. A couple of the problems in the follow up section ask you to implement a solution for a language recognition problem.

The application you will complete implements a sorting algorithm. Sorting is a general problem where given a collection of items, you arrange them in order from smallest to largest. We will restrict ourselves to a collection of integer values in an array. For example, if given the integers 8, 2, 9, 1, 1, 3; their sorted order is 1, 1, 2, 3, 8, 9. You will examine a number of different sorting techniques in Chapters 8 and 9. While it is not obvious, the sort that we will be doing in this lab is equivalent to the insertion sort from Chapter 8 and has the same performance. As with any of the sorts from Chapter 8, the stack sort should not be used in general applications.

Pre-Lab Visualization

Stack Sort

In order to sort values we will use two stacks which will be called the left and right stacks. The values in the stacks will be sorted and the values in the left stack will all be less than or equal to the values in the right stack. The following example illustrates a possible state for our two stacks. Notice that the values in the left stack are sorted so that the smallest value is at the bottom of the stack. The values in the right stact are sorted so that the smallest value is at the top of the stack. If we read the values up the left stack and then down the right stack, we get -1, 1, 3, 9, 11, 11, 11, 15, which is in sorted order.



Suppose that we have a new value that we want to put into our sorted collection. We will want to put it on the top of one of the two stacks, but we may have to first move values around.

No moves required:

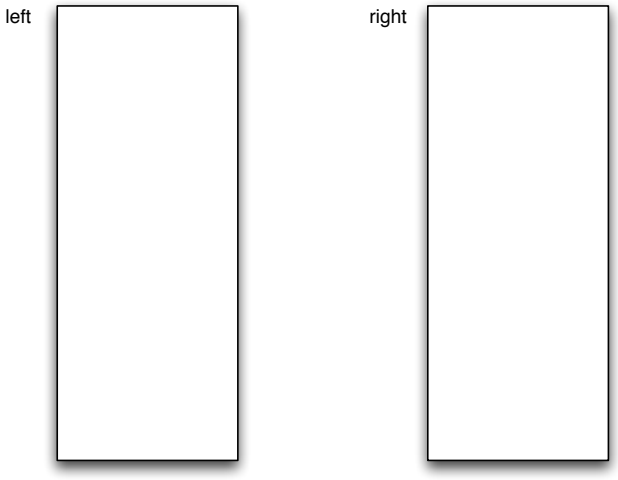
Consider adding the value 5 to the example shown above. We do not have to move any values and can place the 5 on the top of either stack and still have a sorted collection.

Which values would not require that the contents of the stacks be changed?

Moves from left to right required:

Consider adding the value 0 to the example shown above. We must move values from the left stack to the right stack.

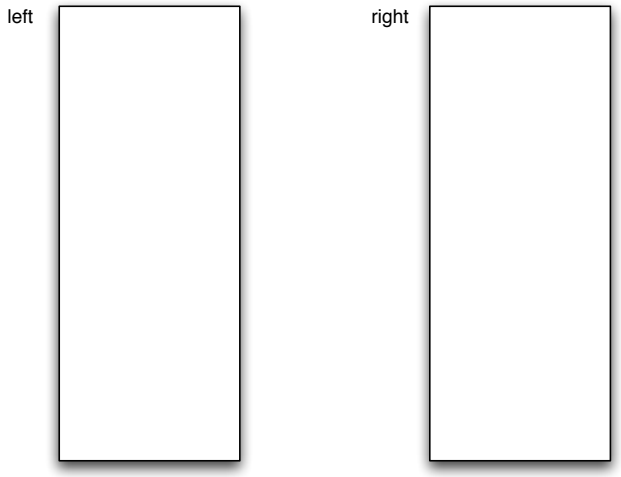
How many values must be moved and what is the state of the two stacks before we add the value 0?



What condition should we use to determine if enough values have been moved?

Consider adding the value -2 to the example shown above. Again must move values from the left stack to the right stack.

How many values must be moved and what is the state of the two stacks before we add the value -2?



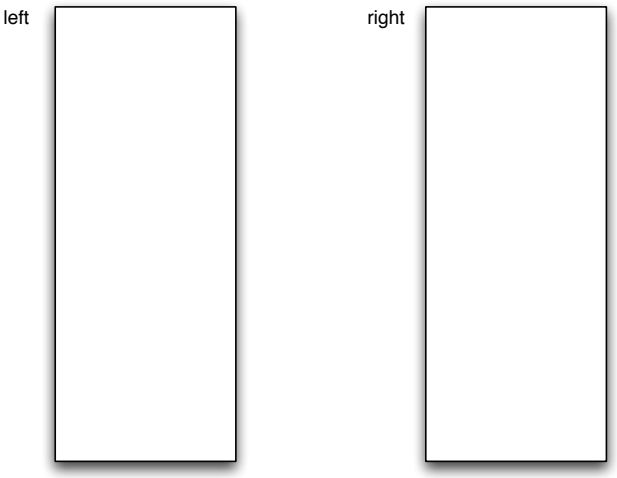
What condition should we use to determine if enough values have been moved?

Write code using iteration that will move values from the left to the right stack as required.

Moves from right to left required:

Consider adding the value 11 to the example shown above. We must move values from the right stack to the left stack.

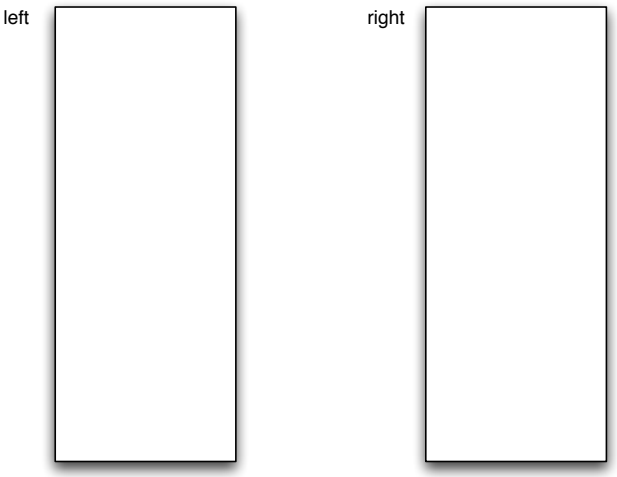
How many values must be moved and what is the state of the two stacks before we add the value 11?



What condition should we use to determine if enough values have been moved?

Consider adding the value 20 to the example shown above. Again we must move values from the right stack to the left stack.

How many values must be moved and what is the state of the two stacks before we add the value 20?



What condition should we use to determine if enough values have been moved?

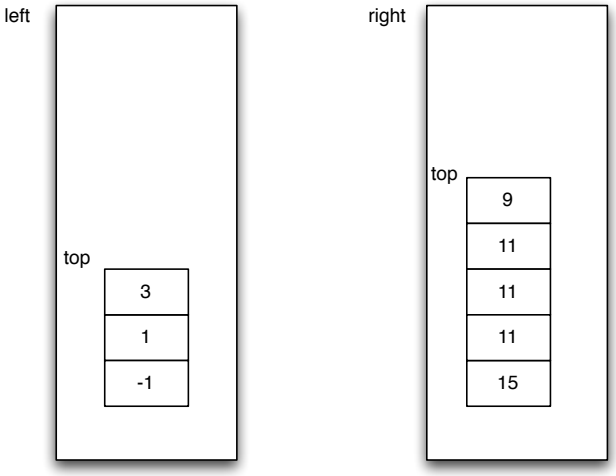
Write code using iteration that will move values from the right to the left stack as required.

Adding all the values from an array into the two stacks:

We can add the values from the array one at a time to the stacks. Putting together the pieces from the previous questions, write an algorithm for this task.

Putting the values into a new array:

For our particular sorting algorithm, we are going to create a second array with the values from the original array in sorted order. Therefore, the final task we need to do before we return is to put the values into the `result` array. Consider again our example.



Suppose we pop the values off of the left stack one at a time. What order do we get?

Suppose we pop the values off of the right stack one at a time. What order do we get?

This suggests that if we move the values from the left stack to the right stack, we can then directly pop them off of the right stack into the `result` array. Write an algorithm that accomplishes this task.

Directed Lab Work

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.