Computer Laboratory 11

CSCI 1913: Introduction to Algorithms,
Data Structures, and Program Development

1 Essential information

- This assignment is due Tuesday December 6th at noon and will be turned in on gradescope
- The assignment is intended to be done in pairs, although you're allowed to work on your own if you want.
- You can start the assignment before your lab period, although I recommend not getting too far – the lab TAs often have useful hints, and I don't want you wasting too much time struggling with something that the TAs might announce in-class.
- For more rules see the lab rules document on canvas.

2 Introduction

In this lab we will be building, and using, our first data structure – the stack. The stack is one of the simplest data structures to write, and often a good choice when starting to learn common data structure design patterns. Through writing this we will:

- Practice implementing a simple array-based data structure.
- Practice thinking about efficiency in your code. (We haven't done this in a while outside of quizzes.)
- Learn a bit about the Stack Abstract Data Type.
- Get a chance to use the Stack data structure in an interesting algorithm (which is hard-to code efficiently without a Stack)

3 Files

This lab will involve the following **provided** files.

- `BraceCheckerTester.java` – tests for the Brace Checker class
- `GenericStackTester.java` – test for the Generic Stack class

This lab also involves the following files **you will create**:

- `BraceChecker.java`
- `GenericStack.java`

4 Instructions

Before beginning you should:

1. Setup an IntelliJ project
2. Download the provided files and place them in the src folder. You will likely need to comment code, out before you can test anything.
3. Start with the Generic Stack class:
    - Make sure you understand what a stack is. (You will find programming the stack very hard if you don't know what a stack is!)
    - Make sure you understand how an array can *represent* a stack using the over-sized array representation.
    - Make sure you understand the formal requirements (both behavioral and runtime) One useful way to do this is to read the tests and try to understand why they are written the way they are.
    - Program the Generic Stack class
    - Test it!
4. Write the BraceChecker class
    - Understand the general algorithm design – You should be able to carry out this design with pencil/paper.
    - Write out pseudo code for this algorithm
    - Code this algorithm.
    - Test it!

5 Theory: What is a Stack

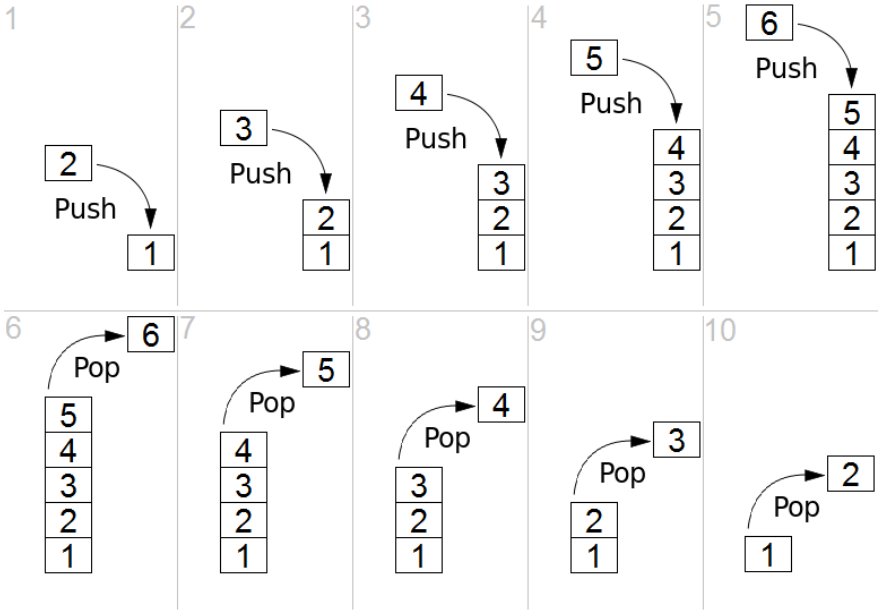
In programming theory, we say that a Stack is any data structure that supports the following four operations:

- push(elem) add something onto the top of the stack
- peek() return the current top of the stack without modifying the stack
- pop() return the current top of the stack, removing that value from the stack
- isEmpty() check if the stack is empty.

While these names are unfamiliar, the behaviors are familiar – push adds an element to the "top" of the stack (think of this like adding to the end of a list), peek gets the element at the "top" of the stack (think of this like getting the last element in a list) and pop removes the top element from the stack (and returns it) (this of this like removing the last element in a list.)

The stack is called a "stack" to help explain it's behavior – newer elements are "stacked" on top of older elements. The older elements can only be removed once the newer elements "stacked" above them are removed. Another term for this behavior is LIFO (Last In First

Out) because whenever we remove something from the stack the last element to have been put in, will always be the first element removed. Below is a visualization of this behavior with a stack of numbers:



While this looks simple, remember that things can get more interesting if we mix push and pop commands.

5.1 Using Arrays to simulate a Stack

Using an array to create Stack like behavior is relatively simple. First, we need to use the "over-large array" pattern. This is the same code design we used in lecture when building ArrayList, you simply make an array that is deliberately too large. So long as there's room in the array, we can always put a new element (on push) into the first empty space in the array. We can also pop by returning the most recently pushed element.

To do this efficiently, we will need one additional variable – an int to track where the "top" of the stack is inside the array. This int will be able to track many things for us:

- The "top" of the stack.
- The size of the Stack (note – not the size of the array)
- The index of the first empty space in the array (the beginning of the not-in-use part)
- The part of the array that is in-use (index 0 up to size-1)

(Make sure you know how to use the top int to do each of these things!)

Finally, we should have a plan for the array becoming full – this will happen. As discussed in class – the best strategy here is to create a new array (twice as large as the old one) and copy the data from the old array into the new array. We can than update our array variable to reference the new array. If we use array doubling then a series of N push functions will take $O(N)$ time. If you use other strategies, such as adding a constant extra size to the array, then the same N push functions would take $O(N^2)$ time.

6 Formal Requirement: GenericStack class

The generic stack class should have:

- A class-level generic. I will use the name `Elem` for this generic in the PDF – but you can name it whatever you want.
- A single private `Elem[]` array variable (I'll call this data)
- A single private int variable (I'll call this size)
- The class rule: `size <= data.length` at all time
- The class rule: Size is the number of in-use elements in the array.
- The class rule: The order of data in the stack (from a logical/mathematical standpoint) is equal to the order of elements in the array. (So after 3 pushes, three elements should be stored in the array – the first thing pushed should be in index 0, the second in index 1 and so-forth.)

The class should have the following public methods:

- A single constructor. This should take an int variable to serve as the initial maximum number of elements. The constructor should then initialize the data array to be a new array whose size matches the initial maximum number. Logically, any number greater than 0 should work for this parameter as the array will resize based on how the data structure is used, however, a user who enters a "good" choice for this value can avoid the array doubling step described above, which is an $O(N)$ operation. Remember – you must use a special syntax for creating this array:
  ```
  data = (Elem[]) new Object[size];
  ```
- A method `isEmpty()` which returns a boolean. This function should return true only if the stack is empty (contains no elements).
- A method `push(Elem)`. If the data array is currently full, this function should double the size of the array using the procedure outlined above. Then it should add the input element into the array and increment the size variable.
- A method `peek()` which returns an Elem. If the stack is empty, then the special value `null` should be returned. Otherwise, the current top of the stack should be returned. This can be found using the size variable and the data array. This function should not modify the data array or size int.
- A method `pop()` which returns an Elem. If the stack is empty, then the special value `null` should be returned and nothing should be changed. Otherwise, the top of the stack should be removed from the stack (you do not need to change the data array to do this, but it is common to explicitly mark that position in the array as `null`, you will need to change the size variable). The removed element should be returned.

Your class is not required to implement `equals` to `toString` although you may find `toString` personally useful when debugging the next part. Therefore I strongly advise implementing these methods.

6.1 BraceChecker

The brace checker class should have one `public static boolean` function: `checkBraces`, which takes a string parameter. The function should return true if the string parameter contains correctly nested parenthesis `()`, square-braces `[]`, and curly-braces `{}`. Any letter other than those six should be ignored. Examples of correctly nested and incorrectly nested strings can be found in the tests for this class.

This is a task that would be quite difficult without a good algorithm (and a good stack) but is quite easy with a stack. Below is pseudocode for this algorithm:

1. begin by creating a stack. This stack will store characters to help us track which open-parenthesis `(`, left-square-brace `[` and left-curly-brace `{` we've seen, but have not yet matched (an in what order).
2. Loop over each character in the string:
   (a) if the character is an open parenthesis `(`, left-square-brace `[` or left-curly-brace `{` simply push these characters onto the stack. This will remember that we've seen these characters.
   (b) if the character is a closing-parenthesis `)`, right-square-brace `]`, or right-curly-brace `}` and the stack is empty, then return false, we've found a right brace/parenthesis which is not matched by a left brace/parenthesis.
   (c) if the character is a closing-parenthesis `)`, right-square-brace `]`, or right-curly-brace `}` and the top of the stack doesn't match it, then we've found a mismatched brace/parenthesis and can return false.
   (d) Otherwise, if the character is a closing-parenthesis `)`, right-square-brace `]`, or right-curly-brace `}` and the top of the stack matches the letter, then we can pop the stack to mark that we've closed that brace/parenthesis, and continue
   (e) If the letter is not one of the 6 braces/parenthesis letters, we can ignore it.
3. Once we've processed every character, if the stack is empty, every left-parenthesis/brace was matched by a right-parenthesis/brace therefore we can return true.
4. If the stack is not empty, then there are some left-parenthesis/braces which were unmatched, and we should return false.

The idea behind this algorithm is that, as we process the string, the stack will track which braces we've seen, but have yet to match to a close-brace, as well as what order we need to see things closed. Because closing braces/parenthesis are in last-opened, first closed order, the stack is well suited to this problem.

7 Submission

For this lab you need to turn in two files:

- `BraceChecker.java`
- `GenericStack.java`

You can submit other files if you wish, but we do not promise to look at them during grading. Your submission must be entered into gradescope before the beginning of your next lab, although it can be up to 24 hours late for a 10% deduction.

8 Grading

This assignment will be partially manually graded, and partially automatically graded. 50 points will be graded automatically – based directly on the provided tests. An additional 50 points will be set aside for manual review and partial credit.

- 10 points for code style, organization, and readability. Like before we will be **requiring** a comment with your name on all files at the top, as well as a javadoc on each method. You should also use your best judgment about how to make this code make sense to others – things like good variable name choices and comments for confusing lines of code, removing large blocks of irrelevant code, etc. are all parts of doing this well.
- 10 points on your Brace Checker algorithm
- 30 points on your Generic Stack class.