CMSC 420: Fall 2022

Programming Assignment 0: Expanding Stack

Handed out: Fri, Sep 2. Due: **Tue, Sep 13 (11:59pm)**.

Overview: This is a start-up project designed to acquaint you with the programming/testing environment and submission process we will be using this semester. This will involve only a small bit of data structure design and implementation (but do check out the challenge problem at the end for extra credit points).

The Expanding Stack: In Lecture 2, we introduced a simple mechanism for implementing an array-based stack that automatically expands as needed by repeated size-doubling. In this assignment, you will implement a simple version of this data structure for storing string objects, called `ExpandingStack`. This data structure supports the following public functions.

`ExpandingStack(int initialCapacity)`: This creates an empty stack as an array of type `String` containing `initialCapacity` elements and sets `top` to -1. We assume that `initialCapacity` ≥ 1, and if not, it throws an `Exception` with the error message "`Invalid capacity`".

`void push(String x)`: This pushes the string `x` onto your stack. If there is not sufficient space in the current array, this allocates a new array of twice the current capacity, copies the elements of the old array into this new array, and then makes this new array the current one. In either case, you can now increment the top and a store `x` at this position in your array.

For example, suppose that your current array had capacity 8 and `top == 7`, meaning that the array is entirely filled. You would allocate an array of size 16, copy the 8 existing elements over, and make this new array the current one. You can now increment top to 8, and store `x` at this index.

`String pop()`: Assuming that `top` ≥ 0, this pops the element at index `top` off the stack, returning its value and decrementing `top`. Otherwise, it throws an `Exception` with the message "`Pop of empty stack`".

`String peek(int idx)`: Assuming that $0 \leq$ `idx` $\leq$ `top`, this returns the element at index `top` $-$ `idx` in your array. Thus, `peek(0)` returns the element at the top of the stack. If `idx` is not in this range, it throws an `Exception` with the message "`Peek index out of range`".

`int size()`: Returns the number of elements currently in the stack (or equivalently `top`$+1$).

`int capacity()`: Returns the current size of your array.

`ArrayList<String> list()`: This returns a Java `ArrayList` whose members are the elements of the stack, listed from the top of the stack down to the bottom. For example, if you started with an empty stack and performed `push("cat"); push("dog"); push("pig")`, this returns an `ArrayList` containing $\langle$`"pig", "dog", "cat"`$\rangle$.

What you need to do: We will provide you with two programs that take care of the input and output (`Part0Tester.java` and `Part0CommandHandler.java`). All you need to do is to implement the above functions. In fact, we will give you a skeleton program, `ExpandingStack.java`, with all the function prototypes, and you just need to fill them in.

```
package cmsc420_f22; // Don't change this line

import java.util.ArrayList;

public class ExpandingStack {

    public ExpandingStack(int initialCapacity) throws Exception { ... }
    public void push(String x) { ... }
    public String pop() throws Exception { ... }
    public String peek(int idx) throws Exception { ... }
    public int size() { ... }
    public int capacity() { ... }
    public ArrayList<String> list() { ... }
}
```

Sample input/output: Here is an example of what the input and output might look like. Let us assume that the initial capacity is set to 4. Notice that when `"frog"` is pushed (resulting in 5 elements in the stack), we allocate a new array with capacity $2 \cdot 4 = 8$.

Input:	**Output:**
push:ladybug	push(ladybug): successful
list	list: ladybug
pop	pop: ladybug
list	list:
push:cat	push(cat): successful
push:dog	push(dog): successful
push:pig	push(pig): successful
push:cow	push(cow): successful
list	list: cow pig dog cat
size	size: 4
capacity	capacity: 4
peek:1	peek(1): pig
peek:-1	peek(-1): Failure due to exception: "Peek index out of range"
push:frog	push(frog): successful
list	list: frog cow pig dog cat
size	size: 5
capacity	capacity: 8

What we give you: We will provide you with skeleton code to get you started on the class Projects page (`Part0-Skeleton.zip`). This code will handle the input and output and provide you with the Java template for `ExpandingStack`. All you need to do is fill in the contents of this class. Note that directory structure has been set up carefully. You should not alter it unless you know what you are doing.

Files: Our skeleton code provides the following files, which can be found in the folder "`cmsc420_f22`". Note that all must begin with the statement "`package cmsc420_f22`".

`Part0Tester.java`**:** This contains the main Java program. It reads input commands from a file (by default `tests/test01-input.txt`) and it writes the output to a file (by default `tests/test01-output.txt`). You can alter the name of the input and output files.

▷ *You should not modify this except possibly to change the input and/or output file names. The output is sent to a file in the* `tests` *directory, not to the Java console. Also note that if you use* `Eclipse`*, the contents of the* `File Explorer` *window are not automatically updated. You will need to refresh its contents to see the new output file.*

We will provide you with a few sample test input files along with the "expected" output results (e.g., `tests/test01-expected.txt`). Of course, you should do your own testing. To check your results, use a difference-checking program like "diff".

Note that the tester program does not generate output to the console (unless there are errors). The output is stored in the output file in the `tests` directory.

`Part0CommandHandler.java`**:** This provides the interface between our `Part0Tester.java` and your `ExpandingStack.java`. It invokes the functions in your `ExpandingStack` class and outputs the results. It also catches and processes any exceptions.

▷ *You should not modify this file.*

Submission Instructions: Submissions will be made through Gradescope. There is no limit to the number of submissions you can make, and only the last submission will be graded. Here is what to do:

- Log into the CMSC420 page on Gradescope, select this assignment, and select "`Submit`". A window will pop up (see Fig. 1). Drag your file `ExpandingStack.java` into the window. If you generated other files, zip them up and submit them all. Select "`Upload`".

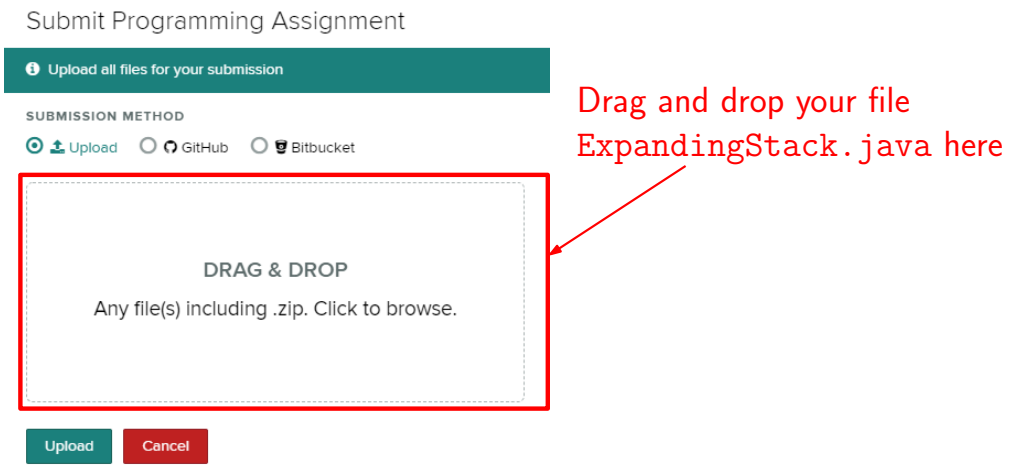

Figure 1: Gradescope submission. Drag your file `ExpandingStack.java` into the box.

- After a few minutes, Gradescope will display the results (see Fig. 2). Normally, a portion of your grade will depend on good style and efficiency, but for this initial program, only the autograder score will be used.

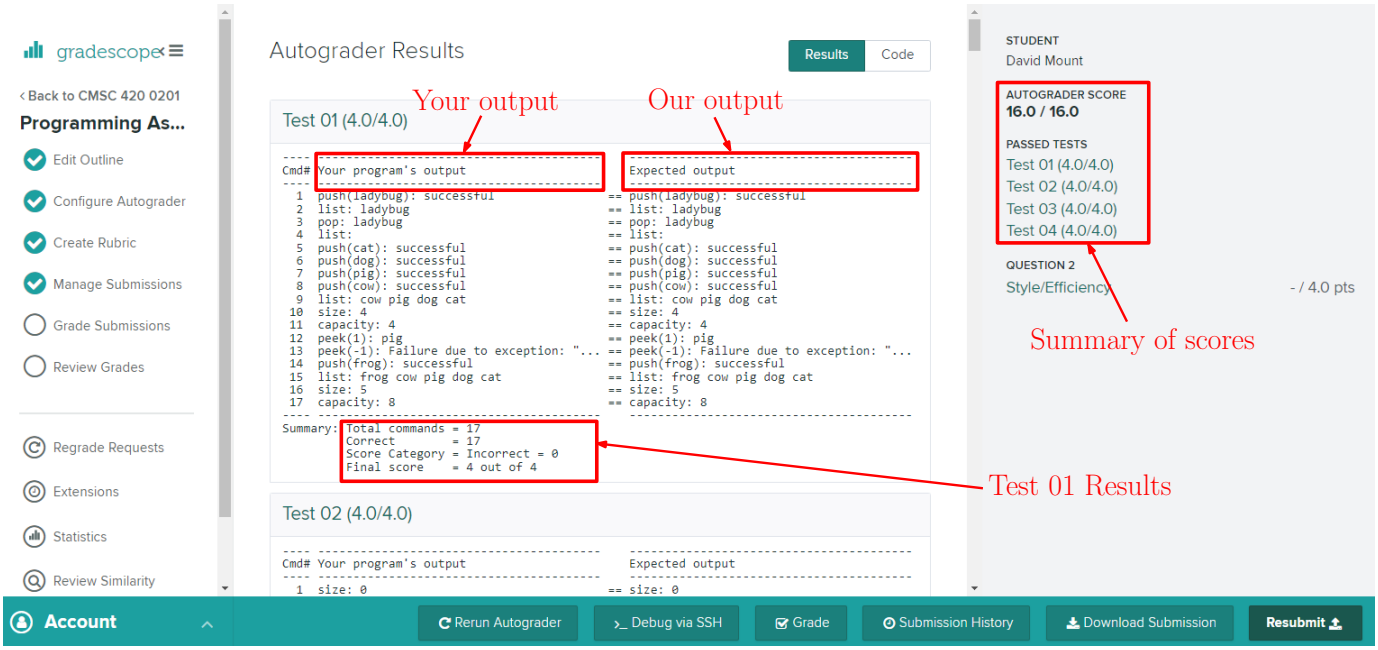

Figure 2: Gradescope autograder results (correct).

- On the top-right of the page, it shows a summary of the scores of the individual tests as generated by the autograder. (If there are compilation errors, these will be displayed on this page.) The center of the window shows a line-by-line summary, with the output generated by your program on the left and the expected output on the right. If there are mismatches, these will be highlighted (see Fig. 3).
- The final score is based on the number of commands for which your program's output differs from ours. Note that the comparison program is very primitive. It compares line by line (without considering the possibility of inserted or deleted lines) and is sensitive to changes in case and the addition of white-space.

Requirements: Because this is a very primitive data structure, your implementation should involve similarly primitive data structures. In particular, your stack contents should be stored in a simple Java array of type `String`. You should not use any additional Java data types, such as `ArrayList` or `LinkedList`. The only exception is the `list` operation (which is just there for debugging and testing purposes), which is allowed to use an `ArrayList` for storing its results.

When grading for efficiency, we require that all the operations run in constant time except for `list` and `push` (but only whenever a reallocation occurs). In both of these exceptional cases, the running time should be $O(n)$, where n is the current number of elements in the stack.

Style and Efficiency: Part of your grade (usually 5%) is based on having clean programming style

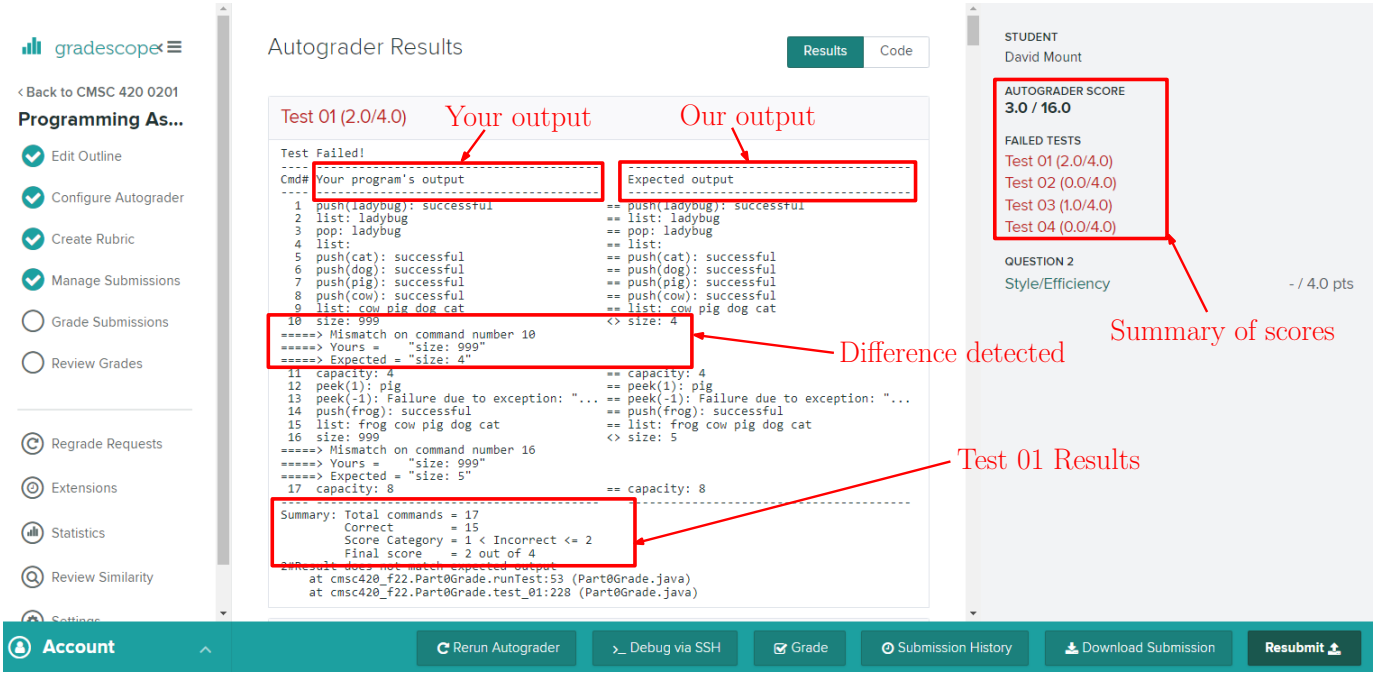

Figure 3: Gradescope autograder results (incorrect).

and implementing the operations in an efficient manner. We have no formal requirements for what constitutes good style. Mostly, it involves that your code is clear and understandable to the grader. The efficiency requirements are mentioned above.

Challenge Problem: (Challenge problems are not graded separately from the assignment. After final grades have been computed, I may "bump-up" a grade that is slightly below a cutoff threshold based on these extra points.)

Ignoring the `list` operation (which is just there for debugging and testing purposes), all the stack operations run in constant time with the exception of when a `push` operation results in the stack overflowing. This operation runs in time proportional to the number of elements in the stack, which may be arbitrarily large. However, the amortized analysis in class shows that on average, each operation requires only *constant time*.

Implement the expanding stack so that all the operations run in constant *worst-case time* (irrespective of the number of elements in the stack). Our strict way of enforcing this is that (ignoring the `list` operation), your program *may not use any looping constructs of any kind*. That is, you may not use `for` or `while` loops, and recursive function calls are not allowed. For example, the following is allowed:

```
A[0] = B[0];
A[1] = B[1];
A[2] = B[2];
```

But this is not:

```
for (int i = 0; i < 3; i++) A[i] = B[i];
```

You are **not** allowed to cheat by invoking Java's built-in functions which do copying behind the scenes (such as `ArrayList` or `java.util.Arrays.copyOf` or `System.arraycopy`.)

Note that there is no need for trickery, but some relaxation to the assignment specifications is needed. Rather than waiting until the array capacity is exceeded, you are allowed to allocate additional array storage any time you like, and you may (prematurely) copy elements into this array prior to the event when the reallocation is actually required. You cannot use any loops, recursion, or any Java functions to assist you with the copying process.

You may assume that allocating a new array of any size runs in constant time. (This is a tiny lie, because Java automatically initializes arrays to zero. But you don't need to rely on this to solve this problem.)

```
String[] A = new String[100000]; // this runs in constant time
```

Also, assigning one array to another (a so-called "shallow copy") also takes constant time. But note that this does not copy individual elements. It just results in two variables that point to the same block of memory.

```
String[] B = A; // now B and A both point to the same array (constant time)
```

If you attempt the challenge, please insert a comment at the top of your `ExpandingStack.java` file explaining that you did this and briefly (in a few sentences) how you did it:

```
// I did the challenge problem. Here's how. ...

package cmsc420_f22;
import java.util.ArrayList;

public class ExpandingStack {
...
}
```