

TRINITY COLLEGE
DEPARTMENT OF COMPUTER SCIENCE

CPSC 215: Data Structures and Algorithms

Instructor: Dr. Chandranil Chakrabortii

Spring 2025

Laboratory 5

Objectives

- To get familiar with basic operations of Stack ADT using Arrays.
- To design and implement two stacks using a single Array.

Academic Honesty Policy

In working on programming assignments, you may discuss broad issues of interpretation and understanding and general approaches to a solution. However, conversion to a specific solution or to program code must be your own work. Programming assignments are expected to be the work of the individual student, designed, and coded by him or her alone. Violations are easy to identify and will be dealt with promptly according to the Academic Integrity and Intellectual Dishonesty outlined in the Student Handbook.

- Copying another person's programs or encouraging or assisting another person to commit plagiarism is cheating. In particular, the following activities are strictly prohibited:
- Giving and receiving help in the actual development of code or writing of an assignment.
- Looking at another person's code or showing your code to another person.
- Sharing a copy of all or part of your code regardless of whether that copy is on paper or in a computer file.
- Turning in the work of any other person(s) (former students, friends, textbook authors, people on the Internet, etc.) and representing it as your own work.
- Fabricating compilation or execution results.
- Use of AI ChatBots for any help regarding the assignment is **strictly prohibited**.

The penalty for cheating is a failing grade (F) for the course, and the student is asked to appear at an Academic Dishonesty Hearing. In addition, the College also places a record of the incident in the student's permanent record. It is your responsibility to protect your work from unauthorized access. Do not discard copies of your programs in public places. Do not leave computers unattended and copies of output lying around.

Getting Started

- Open Eclipse and create a Java Project (File -> New -> Java Project)
- Give a project name (For example: Lab_5_CPSC_215)
- Go to Project -> Right Click -> New Package
- Give a package name (For example: Lab_5_CPSC_215)
- Now to create the classes, go to package -> Right Click -> New Java class.
- Paste contents of the given files. Keep the first line in place (package "package_name";)
- Understand the code structure before starting to write code.

Part 1: Array based stack

First, let's get familiar with Stack ADT and its implementation using arrays.

Download the following Java files:

- **Stack.java** — *Stack ADT interface (No need to change)*
- **AStack.java** — *An array-based list implementation of Stack ADT (No need to change)*
- **StackTest.java** — *A main driver program (Update as necessary)*

Follow the same approach as described in earlier labs to setup your project, you may use one project and package for the entire Lab.

Tasks:

- Using the **StackTest** class, create a stack of Integer type elements.
- Use a loop to **push 10 random** elements between 0-99.
- Print the top value, and **pop 5 elements**.
- Print the **length**. Print the **remaining elements** of the Stack.
- Run and test program for correctness.

Sample Output:

```
[(base) nilchakraborttii@nilchakraborttii's-MacBook-Air P1 % java StackTest
Element Pushed: 79
Element Pushed: 10
Element Pushed: 15
Element Pushed: 15
Element Pushed: 10
Element Pushed: 41
Element Pushed: 72
Element Pushed: 15
Element Pushed: 50
Element Pushed: 36
The top value is: 36
The length is: 5
The remaining elements of the stack are: < 10 15 15 10 79 >
```

Part 2: Using Queues

A queue is a data structure where insertion and deletion of elements is made following the FIFO (First In First Out) principle. That is, the first element that is extracted from the queue is also the first element that is introduced into the queue. First, let's get familiar with Queue ADT and its implementation using a singly linked list. Follow the same approach as described in earlier labs to setup your project, you may use one project and package for the entire Lab.

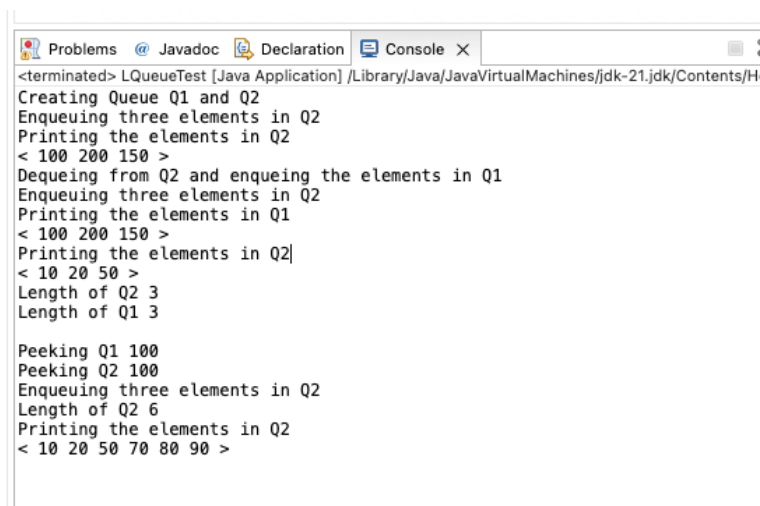
Download the following Java code files and setup your project:

- **Queue1.java:** Queue ADT interface (**No need to change**)
- **Link.java:** A singly linked list node (**No need to change**)
- **LQueue.java:** A linked list-based implementation of Queue ADT (**No need to change**)
- **LQueueTest.java:** The main driver program (**Update as necessary**)

Tasks:

- Modify LQueueTest to perform the operations described in the comments.
- Run LQueueTest.java as Java Application.
- You see an output like below.

Expected Output:



```
<terminated> LQueueTest [Java Application] /Library/Java/JavaVirtualMachines/jdk-21.jdk/Contents/H
Creating Queue Q1 and Q2
Enqueueing three elements in Q2
Printing the elements in Q2
< 100 200 150 >
Dequeing from Q2 and enqueueing the elements in Q1
Enqueueing three elements in Q2
Printing the elements in Q1
< 100 200 150 >
Printing the elements in Q2
< 10 20 50 >
Length of Q2 3
Length of Q1 3

Peeking Q1 100
Peeking Q2 100
Enqueueing three elements in Q2
Length of Q2 6
Printing the elements in Q2
< 10 20 50 70 80 90 >
```

Part 3: Implement two stacks using a single array

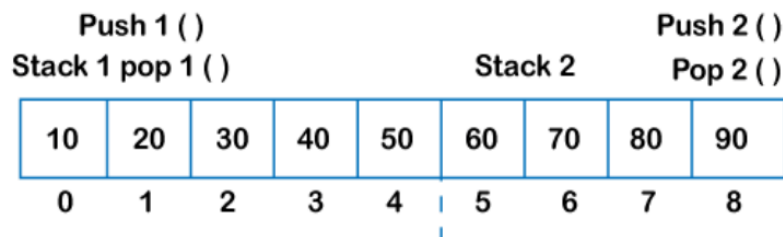
Create a new class called two stacks that implements the generic stack ADT. You will need to update the push and pop methods to allow two stacks to be stored using a single array. Follow the same approach as described in earlier labs to setup your project, you may use one project and package for the entire Lab. You will need the following code to get started:

- **twoStack.java** — Stack ADT interface (**No need to change**)
- **twoStacks.java** — Class containing methods we want to implement (**Need to change**)
- **twoStackTest.java** — A main driver program (**No need to change**)

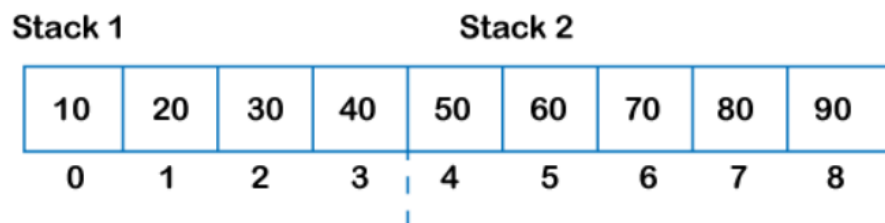
There are two approaches to implement two stacks using one array:

First Approach

First, we will divide the array into two sub-arrays. The array will be divided into two equal parts. First, the sub-array would be considered stack1 and another sub array would be considered stack2. For example, if we have an array of n equal to 8 elements. The array would be divided into two equal parts, i.e., 4 size each shown as below:



The first subarray would be stack 1 named as st1, and the second subarray would be stack 2 named as st2. On st1, we would perform push1() and pop1() operations, while in st2, we would perform push2() and pop2() operations. The stack1 would be from 0 to n/2, and stack2 would be from n/2 to n-1. If the size of the array is odd. For example, the size of an array is 9 then the left subarray would be of 4 size, and the right subarray would be of 5 size shown as below:



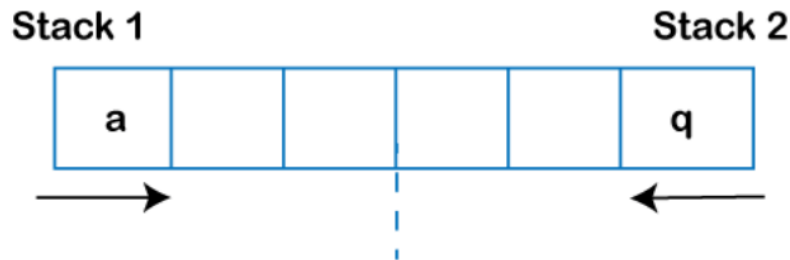
Disadvantage of using this approach

Stack overflow condition occurs even if there is a space in the array. In the above example, if we are performing push1() operation on the stack1. Once the element is inserted at the 3rd index and if we try to insert more elements, then it leads to the overflow error even there is a space left in the array.

Second Approach

In this approach, we are having a single array named as 'a'. In this case, stack1 starts from 0 while stack2 starts from n-1. Both the stacks start from the extreme corners, i.e., Stack1 starts from the leftmost corner (at index 0), and Stack2 starts from the rightmost corner (at index n-1). Stack1 extends in the right direction, and stack2 extends in the left direction, shown as below:

If we push 'a' into stack1 and 'q' into stack2 shown as below:



Therefore, we can say that this approach overcomes the problem of the first approach. In this case, the stack overflow condition occurs only when $top1 + 1 = top2$. This approach provides a space-efficient implementation means that when the array is full, then only it will show the overflow error. In contrast, the first approach shows the overflow error even if the array is not full.

You need to implement the following methods (defined in twoStack.java) interface in twoStacks.java:

- `public void push1(E it);`
- `public void push2(E it);`
- `public void clear();`
- `public E pop1();`
- `public E pop2();`
- `public E topValue1();`
- `public E topValue2();`
- `public int length1();`
- `public int length2();`

Additional Details:

clear():

- **Signature:** `void clear()`
- **Description:** Clears the contents of both stacks, resetting them to an empty state. After calling this method, both stacks should contain no elements.

push1(E it) and push2(E it):

- **Signatures:** void push1(E it) and void push2(E it)
- **Description:** Pushes an element onto the top of stack 1 and stack 2, respectively. The elements to be pushed are provided as arguments to these methods.

pop1() and pop2():

- **Signatures:** E pop1() and E pop2()
- **Description:** Removes and returns the element at the top of stack 1 and stack 2, respectively. It returns the element that was popped from the stack.

topValue1() and topValue2():

- **Signatures:** E topValue1() and E topValue2()
- **Description:** Returns the element at the top of stack 1 and stack 2, respectively, without removing it. It provides access to the element at the top of each stack without modifying the stack itself.

length1() and length2():

- **Signatures:** int length1() and int length2()
- **Description:** Returns the number of elements currently stored in stack 1 and stack 2, respectively. It provides information about the current size of each stack.

Test your Set ADT

- Review twoStacksTest.java (already completed)
- Use twoStacksTest.java to test your implemented methods.

Review your code and Add JavaDoc

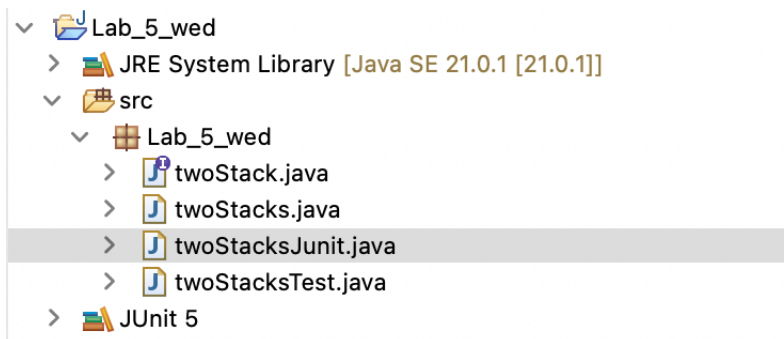
- Be sure the code is runnable, testable, and easily readable.
- Include JavaDoc to explain the purpose and functionality of each class, method, and exception as per standards.
- Reflect on the challenges faced and lessons learned during the implementation.

Sample Output:

```
Console X
<terminated> twoStacksTest (1) [Java Application] /Library/Java/JavaVirtualMachine
Creating two stacks within an array of size: 15
Pushing elements to Stack 1
Peeking top value in Stack 1: 3
Length of Stack 1: 5
Pushing elements to Stack 2
Peeking top value in Stack 2: 40
Length of Stack 2: 4
Popped element from stack 1: 3
Popped element from stack 1: 2
Popped element from stack 2: 40
Popped element from stack 2: 7
Length of Stack 1: 3
Length of Stack 2: 2
```

Part 4: JUnit Testing

- Add JUnit testfile to your project (Project -> Right Click -> New -> Other -> Wizards: JUnit Test Case -> Next)
- Give the same Package name, Enter Name: twoStacksJunit
- Remove module-info.java (if you have one)
- Implement the 5 methods (JUnit testcases) from scratch as described in twoStacksJunit.java.
- Your project structure should look similar to below:



Implement the following JUnit test cases:

1. testPushAndPop():

Description: This test case verifies the functionality of pushing elements onto both stacks and then popping them back. It ensures that the popping operation returns the elements in the correct order and from the correct stack.

Steps:

- Create an instance of `twoStacks<Integer>` with a capacity of 5.
- Push elements onto both stacks.
- Pop elements from each stack and verify that the popped elements match the expected values.

2. testTopValue():

Description: This test case verifies the functionality of retrieving the top values of both stacks without removing them. It ensures that the top values returned are indeed the top elements of the respective stacks.

Steps:

- Create an instance of `twoStacks<Integer>` with a capacity of 5.
- Push elements onto both stacks.

- Retrieve the top values of each stack and verify that they match the expected values.

3. testLength():

Description: This test case verifies the accuracy of the length1() and length2() methods, which return the number of elements in each stack. It ensures that the lengths returned are correct after pushing elements onto the stacks.

Steps:

- Create an instance of twoStacks<Integer> with a capacity of 5.
- Push elements onto both stacks.
- Retrieve the lengths of each stack and verify that they match the expected values.

4. testClear():

Description: This test case verifies the functionality of the clear() method, which clears both stacks, resetting them to an empty state. It ensures that the lengths of both stacks are 0 after the clear() operation.

Steps:

- Create an instance of twoStacks<Integer> with a capacity of 5.
- Push elements onto both stacks.
- Call the clear() method.
- Verify that the lengths of both stacks are 0.

5. testEmptyStacks():

Description: This test case verifies that both stacks are initially empty when no elements are pushed onto them. It ensures that the lengths of both stacks are 0 upon initialization.

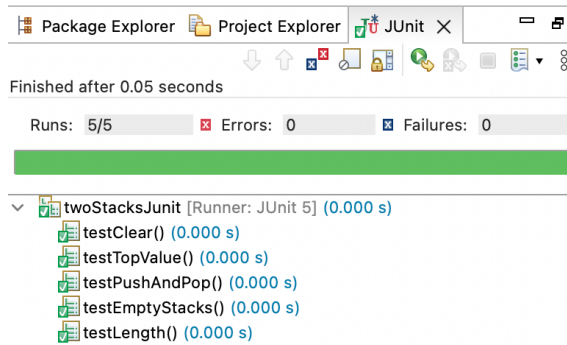
Steps:

- Create an instance of twoStacks<Integer> with a capacity of 5.
- Verify that the lengths of both stacks are 0.

These test cases collectively ensure that the twoStacks class functions correctly according to the specifications defined in the twoStack interface.

Run JUnit Test Cases:

- Run the JUnit testfile. File-> Right Click -> Run As -> JUnit Test.
- You should see your tests pass.
Runs: 5/5 Errors: 0 Failures: 0



Submission Guidelines:

- Make sure you completed all sections with “To do” statements.
 - Make sure you have added JavaDoc in your code for each class.
 - Upon the completion of your lab, upload your submission to the Moodle course website as a single Zip file.
 - See your TA and get your Assignment graded before leaving the lab. Otherwise, inform the instructor.
-