Assignment 1: Stacks, Queues, and Elementary Sorting

In this assignment, you are forbidden from using standard APIs that would otherwise implement Stacks, Queues, or Sorting algorithms. Any data structure used must be programmed on your own and submitted with the assignment.

You will push all of your solutions to the Assignment 1 repository through the Github Classroom. Organize your repository in such a way that it contains:

Root – Assignment 1 directory

- Problem 1 directory
  - Problem 1 code…
  - Main file
- Problem 2 directory
  - Problem 2 code…
  - Main file
- …etc. for all numbered problems in this assignment set.

The "main file" here will be a mostly empty file that just:

- Imports your code, where needed
- Implements an empty main() function that can be compiled
  - In java, you would just have public static void(String[] args)
  - In C++ you would just have imports for the headers / class files / etc, and the int main() function.
- The grader will be using your main file to execute function calls to your APIs.

There is no restriction on the programming language that you use.

For the problems, you will be implementing your code as an API through which the graders will make function calls.

Additionally, include a README.md file for the GitHub repository that has your Name as the title and a description of what is inside the repository, generally.

!! INVITATION LINK: https://classroom.github.com/a/WehBbeVG !!

For anyone new to Github, we encourage you to use **Github Desktop** as a simple GUI for interacting: https://help.github.com/desktop/guides/getting-started-with-github-desktop/

For people who want to be more advanced and up to industry standards, you can use the Git command line:

https://git-scm.com/downloads

A very barebones introduction to the command line (very readable, I recommend it):

http://rogerdudler.github.io/git-guide/

1. **ArbitraryQueue Linked List**
    a. We discussed LIFO and FIFO queues in the lecture slides, calling them **Stacks** and **Queues**, respectively. For this problem, you will implement a merged stack / queue data structure that allows for arbitrary insertion at the front or back of the data structure. We'll call this the **ArbitraryQueue** data structure. Your data structure must generalize for any type of object. It will need to support these types of functions:
        i. Constructor ArbitraryQueue(), to initialize the queue
        ii. Push(Object object), inserts at the head of the ArbitraryQueue
        iii. Enqueue(Object object), inserts at the tail of the ArbitraryQueue
        iv. Pop(), removes the Object at the head of the ArbitraryQueue and returns it
        v. Dequeue(), removes the Object at the tail of the ArbitraryQueue and returns it
        vi. Traverse(int index), which retrieves an Object at the required location and returns it
    b. Implement error handling with try/catch blocks for potential problems in the queue. Make considerations for what can potentially cause errors when a programmer interacts with your API and do the following:
        i. Use an appropriate Exception that describes the error
            1. *Example: Accessing a null value should return a NullPointerException in Java / C++ / Python / etc.*
        ii. Print out an error message describing the problem to the user.
        iii. **Important note: The program execution should not fail. We should be able to continue run-time execution regardless of a thrown exception.**
2. **ArbitraryQueue Dynamic Capacity Array**
    a. In the lecture, we discussed how arrays can be used to implement a Stack and Queue. Use an array to implement a new class, **ArbitraryQueueArray** that supports arbitrary Object insertion.
    b. You must support the same operations as 1a.
    c. You must implement programming logic to manage the array order and handle the instance when the array becomes too large. A copy function should be implemented to handle this.
3. ***Sorting.***
    a. Implement a generic Stack linked list data structure for **integers** that supports these functions:
        i. InsertionSort(), which runs an insertion sort algorithm on the stack
            1. Your insertionSort() algorithm should print to the console every time it makes a swap. Print the operation performed ("Swapped a with b"), then on the next line print the current contents of the array.
            2. Print to the console when the sort has finished ("Sort finished!")
        ii. Push(int data), which pushes values to the head of the stack
        iii. Pop();, which pops values from the head of the stack and returns it
        iv. Peek(), which returns a value from the head of the stack
    b. Implement a pastPeek() function such that it always peeks the head of the stack prior to the sort taking place. When the head of the stack is popped from the sorted stack, update peek() to move to where the next head *would* have been.

   i. Example: Insert: 10,4,14,9,3
   ii. Sort: 3,4,9,10,14
   iii. pastPeek() -> returns 3
   iv. Pop -> removes 3
   v. pastPeek() -> returns 9
   vi. Pop -> removes 4
   Make sure the data structure operation is preserved independently of InsertionSort() being called. Make all appropriate considerations for program crashes with proper error handling using exceptions. Follow the same instructions as 1b.

4. **Problem Solving.**
   a. Write a function ProblemFourA() that reverses a string. Use an appropriate data structure, implemented by yourself but not publicly available to the client calling the API.
      i. Example ProblemFourA("what the?") returns ("?eht tahw").
   b. Write a function ProblemFourB() that takes an int[][] array of arbitrary size and finds all integers that sum to an arbitrary value. ProblemFourB([1 2 ; 3 4], 5) should return (2,3) and (1,4).
   c. Design an algorithm that can be called with ProblemFourC that uses one of the data structures we've discussed so far to find the *Nth* smallest number that takes the form $9^{i}*15^{j}*7^{k}$.
      i. Example ProblemFourC(0) returns 1, ProblemFourC(1) returns 7, and so forth.