Lab6: Stacks and Queues

Goal: In this lab, we will use an abstraction to use arrays as stacks and queues. TEST YOUR CODE AS YOU GO!

Terminology:

Stack- A data structure used to store information. Information is added into the stack in order, and when the information is retrieved, it is returned on a Last In, First Out basis.

Push- The term for adding data into a stack. "Push onto the stack." The data entered maintains order.

Pop- The term for retrieving data from the stack. "Pop off the stack." When retrieving data from a stack, that most recently pushed data will be popped off first. Hence, "Last in, first out" or "First in, last out."

Queue- A data structure used to store information. Information is added into the queue in order, and when the information is retrieved, it is returned on a First In, First Out basis.

Enqueue- The term for adding data into a queue. The data entered maintains order.

Dequeue- The term for retrieving data from the queue. When retrieving data from a queue, the data that was enqueued first will be dequeued first. Hence, "First in, first out."

Task 1: Create basic stack functions

Stack overview:

A stack will be represented by a 101 element array, meaning the indexes go 0-100. The value at index 0 will represent the number of elements in the stack, starting at 0. This way, we don't have to resize the array every time.

// Sample of how code will work

```
main(){
   int[] myS = createStack();
   push(myS,5); // {1,5,0,0,0,0,...}
   push(myS,6);                    // {2,5,6,0,0,0,0,...}
   int popValue = pop(myS);        // popValue = 6, myS={1,5,6,0,0,0,0,...}
   popValue = pop(myS);            // popValue = 5, myS={0,5,6,0,0,0,0,...}
```

// Creation functions

int[] createStack(); // Create and return a stack, which is a 101 element integer array.

```
//Insert Code with comments here:
```

// Create push and pop functions.

void push(int[] stack, int data); // This function should push data onto the stack and increment the number of elements by 1.

int pop(int[] stack); // This function should pop data off of the stack and decrement the number of elements by 1.

```
//Insert Code with comments here:
```

// Accessor Functions

String prettyPrintStack(int[] stack); // Return as a string the stack using square brackets with each term separated by commas. Ex, {3,1,2,3,4} => [1,2,3] because the number of terms is 3, so the stack goes from index 1 to 3, not including 4 and above.

```
//Insert Code with comments here:
```

String dumpStack(int[] stack); // Return the entire array, including the 0th element representing the number of terms. Ex, {1,2,3,4,2,0,0,0,...} => {1,2,3,4,2,0,0,0,...}. It's not a proper stack, so use {}'s.

```
//Insert Code with comments here:
```

Task 2: (Write code that does the following and prints the results in public static void task2())
In task2(), create a stack and push the values 9,8,7,6,5,4,3 in order.

Then pop 3 times. What is the value of the 3rd pop? _____

Continuing, push 1,2,3,4.
Then pop 5 times. What is the value of the 5th pop? _____

Print the stack, what do you get? _____

```
//Insert Code with comments here:
```

Task 3: FILO stock purchase price tracking.
When you buy and sell stocks, the most recently purchased stocks are sold off first in what is known as "First In, Last Out" accounting method, or FILO. Alternatively known as "Last In, First Out" or LIFO.
Example 1: You buy 100 shares of AAPL for $200 per share. You then buy 50 shares of AAPL for $210 per share.

You then want to sell 100 shares of AAPL at $220. How much money have you made? _____

To implement this system of tracking stock purchase prices, we need to store 2 pieces of data, the quantity of shares and the price of each share for that quantity. Because our stacks only store 1 number, we'll use 2 stacks, and run those data structures side by side.

Because we can't store doubles, and we don't want to use doubles for money anyways, we'll store share price in cents. So $200 would be represented as 20000.

Example 2: How your FILO code will use 2 stacks simultaneously to represent trade data.

```
int[] sharesStack = createStack();
int[] priceStack = createStack();
// To simulate buying 100 shares for $200.00 per share:
push(staresStack,100); // {1,100,0,0,0,0, ...}
push(priceStack,20000); // {1,20000,0,0,0,0, ...}
```

// Create the following functions to be used for FILO stock purchasing:

```
// printReportFILO() should print out a nice looking summary of stock purchases.
// Ex,
// Shares        Price
// 100           $200.00
// 50            $210.00
// Do not use doubles.
// "\t" is an escape sequence for tab, use this to make your table evenly spaced.
void printReportFILO(int[] shares, int[] price);
```

```
//Insert Code with comments here:
```

// runReportFILO() is the same as printReportFILO(), except it returns a String instead of printing the report.
String runReportFILO(int[] shares, int[] price);

```
//Insert Code with comments here:
```

// buyFILO() puts the trade data (numShares and pricePerShare) into our stacks.
void buyFILO(int[] shares, int[] price, int numShares, int pricePerShare);

```
//Insert Code with comments here:
```

// sellFILO() sells the appropriate number of shares based on the trade data (numShares and pricePerShare)
// and returns how much money was made or lost in pennies total. Please remember to consider odd lots, meaning
// you may not assume you will always buy and sell a matching number of shares. Example, you can buy 100 shares,
// then immediately sell 1 share. You should have 99 shares remaining.
int sellFILO(int[] shares, int[] price, int numShares, int pricePerShare);

```
//Insert Code with comments here:
```

// averageFILO() returns the average purchase price of all shares purchased. By the end of this function, all
// information should match what was stored before.
// Round to the nearest penny, do not always round down.
int averageFILO(int[] shares, int[] price);

```
//Insert Code with comments here:
```

Task 4: Create basic queue functions

Queue overview:

A queue will be represented by a 101 element array, meaning the indexes go 0-100. The value at index 0 will represent the number of elements, starting at 0. This way, we don't have to resize the array every time.

// Creation functions

int[] createQueue(); // Create and return a queue, which is a 101 element integer array.

//Insert Code with comments here:

// Create enqueue and dequeue functions.

void enqueue(int[] queue, int data); // This function should enqueue data onto the queue and increment the number of elements by 1.

int dequeue(int[] queue); // This function should dequeue data from the queue and decrement the number of elements by 1.

//Insert Code with comments here:

// Accessor Functions

String prettyPrintQueue(int[] queue); // Return as a String the queue using square brackets with each term separated by commas. Ex, {2,1,2,3,4,0,0,0,...} => [1,2] because the number of elements is 2, so the queue goes from index 1 to 2, not including 3 and above.

//Insert Code with comments here:

String dumpQueue(int[] queue); // Return the entire array, including the 0th element representing the number of elements. Ex, {1,2,3,4,2,0,0,0,...} => {1,2,3,4,2,0,0,0,...}. It's not a proper queue, so use {}'s.

//Insert Code with comments here:

Task 5: (Write code that does the following and prints the results in public static void task5().)

In task5(), create a queue and enqueue the values 9,8,7,6,5,4,3 in order.

Then dequeue 3 times. What is the value of the 3rd dequeue? _____

Continuing, enqueue 1,2,3,4.

Then dequeue 5 times. What is the value of the 5th dequeue? _____

Print the queue, what do you get? _____

//Insert Code with comments here:

Task 6 (Graded on effort not correctness. IE, write comments, explain your thought process):
int[] merge(int[] a, int[] b); // Merge a and b. Assume a and b are sorted. If a or b don't have any elements,
// they will be passed in as null, so include null checks.
The merge() function from the Lab5: Array Functions takes 2 sorted arrays, and takes the smaller of the 2 elements off the top of both arrays. After copying an element to the resulting merged array, it is removed from consideration. For example, given array a = {1,3,5} and b = {2,4,6} the merge steps would be:

```
result = {1}               a = {~~1,~~3,5},  b = {2,4,6}
result = {1,2}             a = {~~1,~~3,5},  b = {~~2~~,4,6}
result = {1,2,3}           a = {~~1,3~~,5},  b = {~~2~~,4,6}
result = {1,2,3,4}         a = {~~1,3~~,5},  b = {~~2,4~~,6}
result = {1,2,3,4,5}       a = {~~1,3,5~~},  b = {~~2,4~~,6}
result = {1,2,3,4,5,6}     a = {~~1,3,5~~},  b = {~~2,4,6~~}
```

Step 1: Create 2 Queues called qA and qB.
Step 2: Fill in each queue from the arguments a and b using a for loop for each.
Step 3: Create a result array with the appropriate number of elements. Since we know exactly how many elements the result will have, use a for loop to fill it in.
Step 4: In the for loop from above, keep the leading terms of qA and qB in separate ints, nextA and nextB (declare these variables at the top of your code).
Step 5: Conditionally put in either nextA or nextB. Use qA[0] and qB[0] to tell whether or not you have exhausted each array to be merged.

Note: You may find it useful to use the biggest possible int value, which in JAVA is 2,147,483,647. You can also write Integer.MAX_VALUE instead, and that will evaluate to the same number.

//Insert Code with comments here:

TEST CODE:

```
// WILL ADD TO THIS TEST CODE AS WE GO.
// Test Code: Paste this in main() to test your code.
int[] myStack = createStack();
for(int i=10; i>0; i--)
    push(myStack,i);
System.out.println("You should print [10,9,8,7,6,5,4,3,2,1]");
System.out.println(prettyPrintStack(myStack));
System.out.println("This should print 1\n2\n3\n4\n5\n6\n7\n8\n9\n10");
for(int i=10; i>0; i--)
    System.out.println(pop(myStack));
System.out.println("If you add too many elements into the stack, you should print out an error.");
for(int i=0; i<110; i++)
    push(myStack, i);

//Task 3 Test Code
int[] sharesStack = createStack();
int[] priceStack = createStack();
push(staresStack,100);
push(priceStack,20000);
push(staresStack,50);
push(priceStack,21000);
System.out.println("This should print: \nShares\tPrice\n100\t\t$200.00\n50\t\t$210.00");
printReportFILO(sharesStack, priceStack);
buyFILO(sharesStack, priceStack, 20, 30050);
System.out.println("This should print -61000.”)
```

```
System.out.println(sellFILO(sharesStack, priceStack, 30, 25000));
System.out.println("This should print: \nShares\tPrice\n100\t\t$200.00\n40\t\t$210.00");
System.out.println(runReportFILO(sharesStack, priceStack));
push(staresStack,10);
push(priceStack,19999);
System.out.println("This should print 20267.”);
System.out.println(averageFILO(sharesStack, priceStack));

//Task 4 Test Code
int[] myQueue = createQueue();
for(int i=10; i>0; i--)
    enqueue(myQueue,i);
System.out.println("You should print [10,9,8,7,6,5,4,3,2,1]");
System.out.println(prettyPrintQueue(myQueue));
System.out.println("This should print \n10\n9\n8\n7\n6\n5\n4\n3\n2\n1");
for(int i=10; i>0; i--)
    System.out.println(dequeue(myQueue));
System.out.println("If you add too many elements into the queue, you should print out an error.");
for(int i=0; i<110; i++)
    enqueue(myQueue, i);

//Task 6
int[] a = createQueue();
int[] b = createQueue();
for(int i=1; i<=5; i+=2)
    enqueue(a, i);
for(int i=2; i<=6; i+=2)
    enqueue(b, i);
enqueue(a, 40000);
enqueue(a, 45000);
enqueue(b, 30000);
System.out.println("You should print [1,2,3,4,5,6,30000,40000,45000]");
System.out.println(prettyPrintQueue(merge(a, b)));
```