CS 251 Summer 2020
Project 1: Stacks and Queues
Due date:

General Guidelines

The APIs given below describe the required methods in each class that will be tested. You may need additional methods or classes that will not be directly tested but may be necessary to complete the assignment. For example, to implement a resizable Queue with an array, you will likely need a private resizing method. Keep in mind that anything that does not *need* to be public should generally be kept private (instance variables, helper methods, etc.).

Unless otherwise stated in this handout, you are welcome (and encouraged) to add to/alter the provided java files as well as create new java files as needed.

In general, you are not allowed to import any additional classes in your code without explicit permission from your instructor!

Note on using starter code: You are not required to use the starter code. However, keep in mind that (1) some of the API methods are already implemented for you and (2) if you do not use the starter code, you still need all of the defined methods below without changing any return types or names. Your code must work with the tests that are provided.

Note on academic dishonesty: Please note that it is considered academic dishonesty to read anyone else's solution code to this problem, whether it is another student's code, code from a textbook, or something you found online. You MUST do your own work!

Note on grading and provided tests: The provided tests are to help you as you implement the project. The Vocareum tests will be similar but not exactly the same. Keep in mind that the points you see when you run the local tests are only an indicator of how you are doing. The final grade will be determined by your results on Vocareum.

Part 1: The Stack (12 points)

In this section, you will implement a Stack using a linked list. Your Stack should be generic. The API you must implement is described below. Do not change the names or types of any of the methods below!

Stack<Item> API

Stack()	constructor; creates an empty Stack
void push(Item item)	adds new item to the top of the Stack
Item pop()	removes and returns top item from Stack; throws EmptyStackException if the Stack is empty
boolean isEmpty()	return true if the Stack is empty, false otherwise
int size()	return the size of the Stack (i.e. the number of elements currently in the Stack)
Item peek()	return but do not remove the top item in the Stack; throws EmptyStackException if the Stack is empty

Part 2: The Queue (18 points)
In this section, you will implement a resizable and generic Queue using an array. The API you must implement is described below. Do not change the names or types of any of the methods below!
NOTE: You should implement the Queue using the loop-around method described in the slides. If you do not, you will likely not pass the test cases!

Queue<Item> API

Queue()	constructor; creates and empty Queue with initial capacity of 1
Queue(int n)	constructor; creates an empty Queue with initial capacity of n
void enqueue(Item item)	adds new item to the back of the Queue; double the array capacity if the Queue becomes full
Item dequeue()	removes and returns front item from the Queue; throws EmptyQueueException if the Queue is empty; reduce the array capacity by half if the size of the Queue falls below ¼ full but do not make the size

	`below the initial capacity`
`boolean isEmpty()`	`return true if the Queue is empty, false otherwise`
`int size()`	`return the size of the Queue (i.e. the number of elements currently in the Queue)`
`Item peek()`	`return but do not remove the front item in the Queue; throws an EmptyQueueException if the Queue is empty`
`Item[] getArray()*`	`return the underlying array`

*This is not normally part of the Queue API, but is useful for testing purposes. If you use the skeleton code, this method is already implemented. DO NOT CHANGE OR DELETE IT!

Note: When you resize, the *front* of the queue should be reset to index 0. You should not reset it when the size goes to 0.

The Queue should use a loop-around method in order to reuse open space. Here is an example of how that works. Assume the initial capacity of the array is 5. The “F” stands for “front” and the “B” stands for “back.”

`enqueue 7`

F	B			
7	-	-	-	-

`enqueue 10`

F		B		
7	10	-	-	-

`dequeue`

	F	B		
-	10	-	-	-

`enqueue 12`

	F		B	

-	10	12	-	-

```
enqueue 4
```

	F			B
-	10	12	4	-

```
enqueue 9
```

B	F			
-	10	12	4	9

```
dequeue
```

B		F		
-	-	12	4	9

```
enqueue 3
```

	B	F		
3	-	12	4	9

```
dequeue
```

	B		F	
3	-	-	4	9

```
enqueue 10
```

		B	F	
3	10	-	4	9

Part 3: Searching a Grid (70 points)

In this section you will use your Stack and Queue implementations to solve two specific problems on a Grid of integers.

A. A helper class: *Loc.java*

The *Loc* class is provided as a helper class representing a location in the grid. The methods that are provided are explained below.

You should NOT change the existing methods in Loc.java, or you might not pass the test cases! However, you are welcome to add to the class as needed.

Loc API

`Loc(int x, int y, int val)`	constructor that creates a new Loc (*x, y, val*) where *x* is the row, *y* is the column and, and *val* is the integer value at that location
`String toString()`	returns a String representation of the Loc object in the form (*row, col*)

Note: The *row, col,* and *val* values are public constants which means (1) you can access them without calling a method and (2) you cannot change them once they are set. **Do not change this!**

B. The Grid: *Grid.java*

A Grid is a square $(N \times N)$ matrix that contains digits $(0-9)$. A basic *Grid.java* class is provided for you, as well as several text files containing grid examples. You are welcome to add to the Grid.java file, and you will need to submit it as part of your submission in case of any changes. The methods provided for you should make it easy to create new grids for testing.

You should NOT change the existing methods in Grid.java, or you might not pass the test cases! However, you are welcome to add to the class as needed.

Grid API

`Grid(int n, int min, int max)`	constructor that creates a new *n* by *n* grid and populates it with random digits in the range [min, max)
`Grid(String filename)`	constructor that creates a new Grid that is read in from a file*; note that the first line in the file is the size of the grid

`String toString()`	`creates a String representation of the Grid`
`public int size()`	`returns the size of the grid (i.e.` *n* `for an` *n* `by` *n* `grid)`
`public Loc getLoc(int i, int j)`	`returns the Loc in the grid at (i, j) or null if the indexes are outside the grid`
`private void addSeq(int x, int y, int len)`	`adds a sequence to the grid starting at (x, y) and going for length len; since the directions are chosen randomly, there is a limit to how long it will try to make the sequence before giving up`

*See the example files for reference.

C. Finding a Sequence: *Sequence.java* (35 points)
Consider the following grid.

0	2	4	6	2
1	2	8	2	3
2	4	7	8	2
3	4	5	8	9
3	4	6	0	2

A sequence here is defined as a path through the grid made of consecutive integers. Your task in this part is to use a Stack to search for a sequence where the input is a starting location (i, j) and a value v.

Example (using the Grid above):
Input: $i = 2, j = 0, v = 6$
Output: $(2,\ 0)\ (3,\ 0)\ (3,\ 1)\ (3,\ 2)\ (4,\ 2)$

The output corresponds to the following sequence starting at $(2,\ 0)$ and ending at a location with the value of 6.

0	2	4	6	2

1	2	8	2	3
2	4	7	8	2
3	4	5	8	9
3	4	6	0	2

You will write your solution in **Sequence.java**, following these specifications.

Sequence API

`Sequence (Grid grid)`	`constructor; creates a new Sequence object for the specified Grid object`
`public void reset(Grid grid)`	`resets the grid and path`
`void getSeq(int i, int j, int val)`	searches for a sequence starting at location *(i, j)* and ending at a location with the value *val*; the path, if it exists should be stored in the Stack variable called *path;* if no sequence exists, *path* should be empty
`String toString()`	`returns a String representation of the sequence`

NOTE: Neighboring locations should be searched in the following order: UP, RIGHT, DOWN, LEFT.

The following gives you a basic idea of the *getSeq* procedure. Note that this does not cover all the corner cases. *It is up to you to figure out what all the corner cases may be and handle them accordingly!*

```
G := an N X N grid
path := an empty Stack

Procedure getSeq(int i, int j, int val)
v := the value at (i, j)
until a sequence is found do:
    path.push((i, j))
    for each location (r, c) that neighbors (i, j):
        if (r, c) has not been visited and
        the value at (r, c) is equal to v + 1
        then getSeq(r, c, val)
    end for
    path.pop()
end until
end getSeq
```

Example

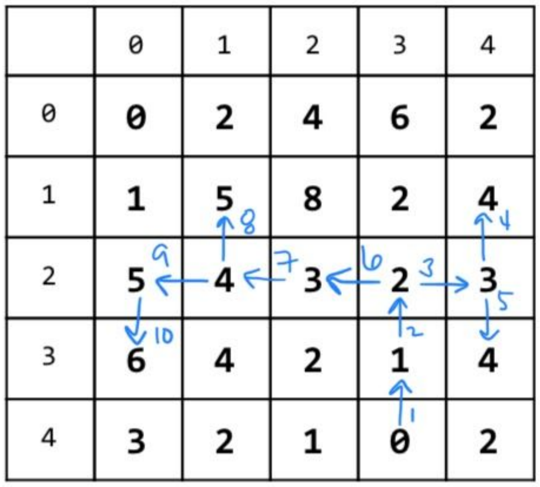
	0	1	2	3	4
0	**0**	**2**	**4**	**6**	**2**
1	**1**	**5**	**8**	**2**	**4**
2	**5**	**4**	**3**	**2**	**3**
3	**6**	**4**	**2**	**1**	**4**
4	**3**	**2**	**1**	**0**	**2**

The following is the trace of calls and the contents of *path* as it goes through the algorithm starting at (4, 3) and searching for a sequence ending with a 6.

```
getSeq(4, 3, 6)
path: (4, 3)
        getSeq(3, 3, 6)
        path: (4, 3) (3, 3)
                getSeq(2, 3, 6)
                path: (4, 3) (3, 3) (2, 3)
                        getSeq(2, 4, 6)
                        path: (4, 3) (3, 3) (2, 3) (2, 4)
                                getSeq(1, 4, 6)
                                path: (4, 3) (3, 3) (2, 3) (2, 4) (1, 4)
                                path: (4, 3) (3, 3) (2, 3) (2, 4)
                                getSeq(3, 4, 6)
                                path: (4, 3) (3, 3) (2, 3) (2, 4) (3, 4)
                                path: (4, 3) (3, 3) (2, 3) (2, 4)
                        path: (4, 3) (3, 3) (2, 3)
                        getSeq(2, 2, 6)
                        path: (4, 3) (3, 3) (2, 3) (2, 2)
                                getSeq(2, 1, 6)
                                path: (4, 3) (3, 3) (2, 3) (2, 2) (2, 1)
                                        getSeq(1, 1, 6)
                                        path: (4, 3) (3, 3) (2, 3) (2, 2) (2, 1) (1, 1)
                                        path: (4, 3) (3, 3) (2, 3) (2, 2) (2, 1)
                                        getSeq(2, 0, 6)
```

```
path: (4, 3) (3, 3) (2, 3) (2, 2) (2, 1) (2, 0)
        getSeq(3, 0, 6)
        path: (4, 3) (3, 3) (2, 3) (2, 2) (2, 1) (2, 0) (3, 0)
```

The trace corresponds to the following search through the grid. Each arrow is numbered, indicating the order in which the locations are explored.



	0	1	2	3	4
0	0	2	4	6	2
1	1	5	8	2	4
2	5	4	3	2	3
3	6	4	2	1	4
4	3	2	1	0	2

Notice that the stack (***path***) at the end contains the locations in order for the final sequence.

Hints and Guidelines

- The pseudocode above is meant as a blueprint. There are implementation details that you need to figure out for yourself. For example, what do you do when you have found the value you are looking for? How do you keep track of locations that are already visited so that you don't end up going in circles?
- Note that ***path*** is a Stack, which means that when locations are popped off, they will be in the reverse direction of that path. You should think about this when you implement the *toString* method, which expects the path to be printed in order (starting location first, ending location last.)
- Note that the algorithm is *recursive*. It may be possible to do the same thing non-recursively, but this is not recommended, as it tends to be simpler to use recursion.
- Neighboring locations should always be checked in the following order: UP, RIGHT, DOWN, LEFT.
- If a sequence does not exist, the Stack should be empty. Think about situations that should result in an empty Stack.

D. Finding a Pair: *ClosestPair.java* (35 points)
In this section, you will use a Queue to search for the closest matching value in a grid, given a starting value.

Consider the following grid and the starting location (0, 3). The closest pair would be (4, 2). In the same grid, if the input location is (2, 3), the closest pair would be (3, 3). In the case of the starting location (1, 0), there is no pair.

0	2	4	6	2
1	2	8	2	3
2	4	7	8	2
3	4	5	8	9
3	4	6	0	2

Given a Grid and a starting location, your task is to implement an algorithm for finding the closest pair for the value at the starting location. You will write your solution in **ClosestPair.java**, following these specifications.

ClosestPair API

`ClosestPair (Grid grid)`	`constructor; creates a new ClosestPair object for the specified Grid object`
`Loc search(int x, int y)`	`searches for the closest* location with a value equal to the value at` *`(i, j)`*`;` `return` *`null`* `if no match is found`

**Closest* means that it takes the fewest number of steps to reach the new location by moving only UP, RIGHT, DOWN, or LEFT.
NOTE: Neighboring locations should be searched in the following order: UP, RIGHT, DOWN, LEFT.

The following pseudocode gives you a basic idea of the *search* function in ClosestPair. It is up to you to determine specific implementation details. Since you are looking for the *closest* pair, it makes sense to search all the closest locations first before moving on to locations that are further away. This can be done using a Queue.

```
Algorithm search(Loc start)
v := the value at start
Q := a queue
Q.enqueue(start)
while Q is not empty do
    Loc curr = Q.dequeue()
    for each neighbor n of curr do
        if the value at n is equal to v
            return n
        else if n is not already in the queue and
        n has not already been checked:
            Q.enqueue(n)
    end for
end while
return null
```

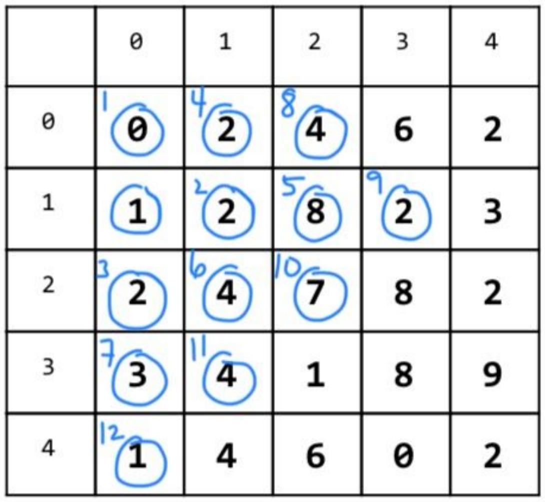
Example

	0	1	2	3	4
0	**0**	**2**	**4**	**6**	**2**
1	**1**	**2**	**8**	**2**	**3**
2	**2**	**4**	**7**	**8**	**2**
3	**3**	**4**	**1**	**8**	**9**
4	**1**	**4**	**6**	**0**	**2**

```
Q: (1, 0)
curr = (1, 0)
//check (0, 0)
//check (1, 1)
//check (2, 0)
Q: (0, 0) (1, 1) (2, 0)
curr = (0, 0)
//check (0, 1)
Q: (1, 1) (2, 0) (0, 1)
curr = (1, 1)
//check (1, 2)
//check (2, 1)
Q: (2, 0) (0, 1) (1, 2) (2, 1)
curr = (2, 0)
//check (3, 0)
Q: (0, 1) (1, 2) (2, 1) (3, 0)
curr = (0, 1)
```

```
//check (0, 2)
Q: (1, 2) (2, 1) (3, 0) (0, 2)
curr = (1, 2)
//check (1, 3)
//check (2, 2)
Q: (2, 1) (3, 0) (0, 2) (1, 3) (2, 2)
curr = (2, 1)
//check (3, 1)
Q: (3, 0) (0, 2) (1, 3) (2, 2) (3, 1)
curr = (3, 0)
//check (4, 0)--match found!
```

The following shows the grid with numberings to show the order in which locations are searched.



	0	1	2	3	4
0	^{1} 0	^{4} 2	^{8} 4	6	2
1	1	^{2} 2	^{5} 8	^{9} 2	3
2	^{3} 2	^{6} 4	^{10} 7	8	2
3	^{7} 3	^{11} 4	1	8	9
4	^{12} 1	4	6	0	2

Hints and Guidelines

- The pseudocode above is meant as a blueprint. There are implementation details that you need to figure out for yourself. For example, how do you keep track of what is already in the queue? How do you keep track of what locations you have already visited? You should try to do these things with minimal runtime.
- Neighboring locations should always be checked in the following order: UP, RIGHT, DOWN, LEFT.

Submission

Your submission should include the following files as well as any additional *.java* files that you created and are necessary for your implementation. Do NOT submit anything other than *.java* files!

- Stack.java
- Queue.java
- Loc.java
- Grid.java

- Sequence.java
- ClosestPair.java

Testing and Grading

Part 1: Stack
You are provided with the file *StackMain.java*, which tests the Stack according to the following rubric:

> **Test 1:** Tests Stack operations on String objects (6 points total)
> **Test 2:** Tests Stack operations on Integer objects (6 points total)

Part 2: Queue
You are provided with the *QueueMain.java*, which tests the Queue according to tthe following rubric:

> **Test 1:** Tests Queue operations on String objects (7 points total)
> **Test 2:** Tests Queue operations on Integer objects (7 points total)
> **Test 3**: Tests the underlying data structure (the array)--if you completely fail this

test, you fail all tests for Part 2! This test also uses input and output files which are provided. The test used for grading will use a different input and output file, but will test essentially the same thing.

Part 3.C: Sequence
You are provided with two grid files as well as some output files for specific tests as well as *SequenceMain.java* that runs the tests according to the following rubric (note that actual tests for grading will be different, but they will follow the same rubric.)

> **Tests 1-7:** Tests on basic grid, including some corner cases (26 points total)
> **Tests 8 & 9:** Tests on large grid (9 points total)

Part 3.D: ClosestPair
You are provided with two grid files and *PairMain.java*, which tests the ClosestPair implementation according to the following rubric (note that actual tests for grading will be different, but they will follow the same rubric.)

> **Tests 1-4:** Tests on basic grid (16 points total)
> **Tests 5-8:** Tests on large grid (16 points total)
> **Test 9:** Corner case (3 points)