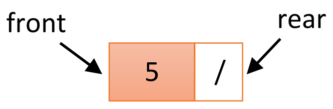
Question 2:

Data Structures and Algorithms

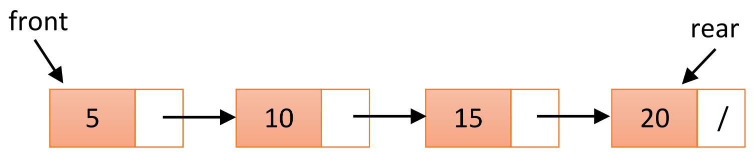
The objective of this exam is to implement a new data structure for called ULQueue. An ULQueue is a linked list based implementation of a queue.

Initially: front = null and rear = null

After adding one element:



After adding some more elements:



The queue contains integers and supports the usual *enqueue* and *dequeue* operations, plus a new *undo* operation. The undo operation allows us to change the queue back to a previous version, by reversing the last *enqueue* or *dequeue* operation. We can call undo n times to reverse the last n operations. If there is no previous *enqueue* or *dequeue* operations (because the list is new, or because all operations has been undone), undo does nothing.

As an example, the following shows the sequence of operations executed and the queue after each operation

1. enqueue(5)	//<front> 5 <rear>	
2. enqueue(10)	//<front> 5 , 10 <rear>	
3. enqueue(15)	//<front> 5 , 10 , 15 <rear>	
4. enqueue(20)	//<front> 5 , 10 , 15 , 20 <rear>	
5. dequeue()	//<front> 10 , 15 , 20 <rear>	
6. dequeue()	//<front> 15 , 20 <rear>	
7. undo()	//<front> 10 , 15 , 20 <rear>	10 is reinserted
8. undo()	//<front> 5 , 10 , 15 , 20 <rear>	5 is reinserted
9. undo()	//<front> 5 , 10 , 15 <rear>	enqueue(20) is cancelled
10. enqueue(30)	//<front> 5 10 15 30 <rear>	
11. enqueue(40)	//<front> 5 10 15 30 40 <rear>	
12. enqueue(50)	//<front> 5 10 15 30 40 50 <rear>	
13. undo()	//<front> 5 10 15 30 40 <rear>	enqueue(50) is cancelled
14. dequeue()	//<front> 10 15 30 40 <rear>	
15. undo()	//<front> 5 10 15 30 40 <rear>	5 is reinserted

The most suitable data structure for storing undo states is a *stack*. Whenever an action is performed, a pair of action and the element is pushed onto the undo stack. When undo is invoked, pop the pair from the stack, and reverse the action in the queue.

Question continues on next page --->

To indicate the type of the action, we can use integers, such as: 1 for enqueue and -1 for dequeuer.

For example,

element action

1.	enqueue(5)	//<front> 5 <rear>	Stack TOP -> 5 , 1
2.	enqueue(10)	//<front> 5 , 10 <rear>	Stack TOP -> 10 , 1 , 5 , 1
3.	enqueue(15)	//<front> 5 , 10 , 15 <rear>	Stack TOP -> 15 , 1 , 10 , 1 , 5 , 1
4.	enqueue(20)	//<front> 5 , 10 , 15 , 20 <rear>	Stack TOP -> 20 , 1 , 15 , 1 , 10 , 1 , 5 , 1
5.	dequeue()	//<front> 10 , 15 , 20 <rear>	Stack TOP -> 5 , -1 , 20 , 1 , 15 , 1 , 10 , 1 , 5 , 1
6.	dequeue()	//<front> 15 , 20 <rear>	Stack TOP -> 10 , -1 , 5 , -1 , 20 , 1 , 15 , 1 , 10 , 1 , 5 , 1
7.	undo()	//<front> 10 , 15 , 20 <rear>	pop out the pair and do the action accordingly
8.	undo()	//<front> 5 , 10 , 15 , 20 <rear>	pop out the pair and do the action accordingly
9.	undo()	//<front> 5 , 10 , 15 <rear>	pop out the pair and do the action accordingly

Following three classes have been given to you:

1. QNode: Implements a node in ULQueue. Each node contains an int value, and a next reference.
2. ULQueue: Implements an UNDOable List based Queue. Some methods are incomplete. You are required to complete all the incomplete methods. A method toString() is given to you. You can call toString() to convert your queue into a string for printing and debugging.
3. testULQueue: This class defines the main() method used to test your implementation.

YOUR TASK:

You are required to complete the three methods in class ULQueue listed below as described previously:

- **void enqueue(int element)**: insert the element at the rear of the queue
- **Integer dequeue()**: remove and return the front element
- **void undo()**: Revert the list back to the previous version by canceling the last edit operation. If no previous version exists (either we have reverted to the oldest version available or the list is new), it has no effect on the list.

For your convenience, the Java's stack class is used to define the undostack and it is ready to use. You may use the following predefined associated operations as necessary:

- **undostack.empty()**: The empty() method of the Stack class check the stack is empty or not. If the stack is empty, it returns true, else returns false.
- **undostack.push(value)**: The method inserts an item onto the top of the stack.
- **undostack.pop()**: The method removes an element from the top of the stack and returns it.

Additional marks for REDO operation!

After completing the above three queue operations, try to perform REDO operation to get additional marks.

Good Luck!!

*** End of Questions ***