8.1 Queue abstract data type (ADT)

Queue abstract data type

A ***queue*** is an ADT in which items are inserted at the end of the queue and removed from the front of the queue. The queue ***enqueue*** operation inserts an item at the end of the queue. The queue ***dequeue*** operation removes and returns the item at the front of the queue. Ex: After the operations "Enqueue 7", "Enqueue 14", and "Enqueue 9", "Dequeue" returns 7. A second "Dequeue" returns 14. A queue is referred to as a ***first-in first-out*** ADT. A queue can be implemented using a linked list or an array.

A queue ADT is similar to waiting in line at the grocery store. A person enters at the end of the line and exits at the front. British English actually uses the word "queue" in everyday vernacular where American English uses the word "line".

PARTICIPATION ACTIVITY 8.1.1: Queue ADT.

Animation content:

undefined

Animation captions:

1. A new queue named "wQueue" is created. Items are enqueued to the end of the queue.
2. Items are dequeued from the front of the queue.

PARTICIPATION ACTIVITY 8.1.2: Queue ADT.

1) Given numQueue: 5, 9, 1 (front is 5)
What are the queue contents after the following enqueue operation? Type the queue as: 1, 2, 3

Enqueue(numQueue, 4)

Check Show answer

2) Given numQueue: 11, 22 (the front is 11)
What are the queue contents after the following enqueue operations? Type the queue as: 1, 2, 3

Enqueue(numQueue, 28)
Enqueue(numQueue, 72)

Check Show answer

3) Given numQueue: 49, 3, 8
What is returned by the following dequeue operation?

Dequeue(numQueue)

Check Show answer

4) Given numQueue: 4, 8, 7, 1, 3
What is returned by the second dequeue operation?

Dequeue(numQueue)
Dequeue(numQueue)

Check Show answer

5) Given numQueue: 15, 91, 11
What is the queue after the following dequeue operation?
Type the queue as: 1, 2, 3

Dequeue(numQueue)

Check Show answer

6) Given numQueue: 87, 21, 43
What are the queue's contents after the following operations?
Type the queue as: 1, 2, 3

Dequeue(numQueue)
Enqueue(numQueue, 6)
Enqueue(numQueue, 50)
Dequeue(numQueue)

Check Show answer

Common queue ADT operations

Table 8.1.1: Some common operations for a queue ADT.

Operation	Description	Example starting with queue: 43, 12, 77 (front is 43)
Enqueue(queue, x)	Inserts x at end of the queue	Enqueue(queue, 56). Queue: 43, 12, 77, 56
Dequeue(queue)	Returns and removes item at front of queue	Dequeue(queue) returns: 43. Queue: 12, 77
Peek(queue)	Returns but does not remove item at the front of the queue	Peek(queue) return 43. Queue: 43, 12, 77
IsEmpty(queue)	Returns true if queue has no items	IsEmpty(queue) returns false.
GetLength(queue)	Returns the number of items in the queue	GetLength(queue) returns 3.

Note: Dequeue and Peek operations should not be applied to an empty queue; the resulting behavior may be undefined.

PARTICIPATION ACTIVITY 8.1.3: Common queue ADT operations.

1) Given rosterQueue: 400, 313, 270, 514, 119, what does GetLength(rosterQueue) return?

- ○ 400
- ○ 5

2) Which operation determines if the queue contains no items?

- ○ IsEmpty
- ○ Peek

3) Given parkingQueue: 1, 8, 3, what are the queue contents after Peek(parkingQueue)?

- ○ 1, 8, 3
- ○ 8, 3

4) Given parkingQueue: 2, 9, 4, what are the contents of the queue after Dequeue(parkingQueue)?

- ○ 9, 4
- ○ 2, 9, 4

5) Given that parkingQueue has no items (i.e., is empty), what does GetLength(parkingQueue) return?

- ○ -1
- ○ 0
- ○ Undefined

CHALLENGE ACTIVITY 8.1.1: Queue ADT.

422352.2723990.qx3zqy7

Start

Given numQueue: 53, 72, 75
What are the queue's contents after the following operations?

```
Dequeue(numQueue)
Enqueue(numQueue, 38)
```

Ex: 1, 2, 3

After the above operations, what does GetLength(numQueue) return?

Ex: 8

1 2 3 4

Check Next

8.2 Queues using linked lists

A queue is often implemented using a linked list, with the list's head node representing the queue's front, and the list's tail node representing the queue's end. Enqueueing an item is performed by creating a new list node, assigning the node's data with the item, and appending the node to the list. Dequeuing is performed by assigning a local variable with the head node's data, removing the head node from the list, and returning the local variable.

PARTICIPATION ACTIVITY 8.2.1: Queue implemented using a linked list.

Animation content:

undefined

Animation captions:

1. Enqueueing an item puts the item in a list node and appends the node to the list.
2. A dequeue stores the head node's data in a local variable, removes the list's head node, and returns the local variable.

PARTICIPATION ACTIVITY 8.2.2: Queue push and pop operations with a linked list.

Assume the queue is implemented using a linked list.

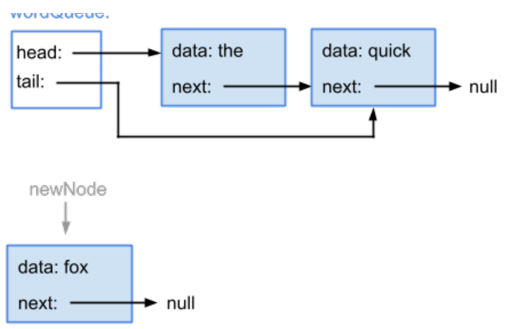
1) If the head pointer is null, the queue _____.
   - is empty
   - is full
   - has at least one item

2) For the operation QueueDequeue(queue), what is the second parameter passed to ListRemoveAfter?
   - The list's head node
   - The list's tail node
   - null

3) For the operation QueueDequeue(queue), headData is assigned with the list _____ node's data.
   - head
   - tail

4) For QueueEnqueue(wordQueue, "fox"), which pointer is updated to point to the node?

wordQueue:



- ○ wordQueue's head pointer
- ○ The head node's next pointer
- ○ The tail node's next pointer

CHALLENGE ACTIVITY 8.2.1: Queues using linked lists.

422352.2723990.qx3zqy7

Start

Given an empty queue numQueue, what does the list head pointer point to? If the pointer is null, enter null.

Ex: 5 or null

What does the list tail pointer point to?

After the following operations:

```
QueueEnqueue(numQueue, 35)
QueueEnqueue(numQueue, 95)
QueueEnqueue(numQueue, 84)
QueueDequeue(numQueue)
```

What does the list head pointer point to?

What does the list tail pointer point to?

1 2

Check Next

8.3 Deque abstract data type (ADT)

Deque abstract data type

A ***deque*** (pronounced "deck" and short for double-ended queue) is an ADT in which items can be inserted and removed at both the front and back. The deque push-front operation inserts an item at the front of the deque, and the push-back operation inserts at the back of the deque. The pop-front operation removes and returns the item at the front of the deque, and the pop-back operation removes and returns the item at the back of the deque. Ex: After the operations "push-back 7", "push-front 14", "push-front 9", and "push-back 5", "pop-back" returns 5. A subsequent "pop-front" returns 9. A deque can be implemented using a linked list or an array.

PARTICIPATION ACTIVITY 8.3.1: Deque ADT.

Animation captions:

1. The "push-front 34" operation followed by "push-front 51" produces a deque with contents 51, 34.
2. The "push-back 19" operation pushes 19 to the back of the deque, yielding 51, 34, 19. "Pop-front" then removes and returns 51.
3. Items can also be removed from the back of the deque. The "pop-back" operation removes and returns 19.

PARTICIPATION ACTIVITY 8.3.2: Deque ADT.

Determine the deque contents after the following operations.

If unable to drag and drop, refresh the page.

push-front 71, push-front 68, push-front 97, pop-back, push-front 45	**push-front 97, push-back 71, pop-front, push-front 45, push-back 68**	**push-back 45, push-back 71, push-front 97, push-front 68, pop-back**

45, 97, 68

45, 71, 68

68, 97, 45

Reset

Common deque ADT operations

In addition to pushing or popping at the front or back, a deque typically supports peeking at the front and back of the deck and determining the length. A ***peek*** operation returns an item in the deque without removing the item.

Table 8.3.1: Common deque ADT operations.

Operation	Description	Example starting with deque: 59, 63, 19 (front is 59)
PushFront(deque, x)	Inserts x at the front of the deque	PushFront(deque, 41). Deque: 41, 59, 63, 19
PushBack(deque, x)	Inserts x at the back of the deque	PushBack(deque, 41). Deque: 59, 63, 19, 41
PopFront(deque)	Returns and removes item at front of deque	PopFront(deque) returns 59. Deque: 63, 19
PopBack(deque)	Returns and removes item at back of deque	PopBack(deque) returns 19. Deque: 59, 63
PeekFront(deque)	Returns but does not remove the item at the front of deque	PeekFront(deque) returns 59. Deque is still: 59, 63, 19
PeekBack(deque)	Returns but does not remove the item at the back of deque	PeekBack(deque) returns 19. Deque is still: 59, 63, 19
IsEmpty(deque)	Returns true if the deque is empty	IsEmpty(deque) returns false.
GetLength(deque)	Returns the number of items in the deque	GetLength(deque) returns 3.

PARTICIPATION ACTIVITY 8.3.3: Common deque ADT operations.

1) Given rosterDeque: 351, 814, 216, 636, 484, 102, what does GetLength(rosterDeque) return?

- 351
- 102
- 6

2) Which operation determines if the deque contains no items?

○ IsEmpty

○ PeekFront

3) Given jobsDeque: 4, 7, 5, what are the deque contents after PeekBack(jobsDeque)?

○ 4, 7, 5

○ 4, 7

4) Given jobsDeque: 3, 6, 1, 7, what are the contents of the deque after PopFront(jobsDeque)?

○ 6, 1, 7

○ 3, 6, 1, 7

5) Given that jobsDeque is empty, what does GetLength(jobsDeque) return?

○ -1

○ 0

○ Undefined

CHALLENGE ACTIVITY | 8.3.1: Deque ADT.

422352.2723990.qx3zqy7

Start

Given an empty deque numDeque, what are the deque's contents after the following opera

PushFront(numDeque, 32)
PushBack(numDeque, 57)
PushBack(numDeque, 29)

Ex: 1, 2, 3

After the above operations, what does PeekFront(numDeque) return?

Ex: 5

After the above operations, what does PeekBack(numDeque) return?

Ex: 5

After the above operations, what does GetLength(numDeque) return?

1 2 3

Check Next

8.4 Queue interface

Queue interface

The ***Queue*** interface defined within the Java Collections Framework defines a Collection of ordered elements that supports element insertion at the tail and element retrieval from the head.

A LinkedList is one of several types that implements the Queue interface. A LinkedList implementation of a Queue can be declared and created as `Queue<T> queue = new LinkedList<T>();` where T represents the element's type, such as Integer or String. Java supports automatic conversion of an object, e.g., LinkedList, to a reference variable of an interface type, e.g., Queue, as long as the object implements the interface.

The statements `import java.util.LinkedList;` and `import java.util.Queue;` enable use of a LinkedList and Queue within a program.

A Queue's ***add()*** method adds an element to the tail of the queue and increases the queue's size by one. A Queue's ***remove()*** method removes and returns the element at the head of the queue. If the queue is empty, remove() throws an exception.

PARTICIPATION ACTIVITY 8.4.1: Queue: add() method adds an element to the tail of the queue, and remove() method returns and removes the element at the head of the queue.

Animation captions:

1. The add() method adds an element, such as a String, to the tail of the queue.
2. The remove() method returns and removes the element at the head of the queue.

PARTICIPATION ACTIVITY 8.4.2: Use Queue's add() and remove() methods to insert and retrieve elements.

Answer the questions given the following code that creates and initializes a Queue.

```
Queue<Integer> ordersQueue = new LinkedList<Integer>();

ordersQueue.add(351);
ordersQueue.add(352);
ordersQueue.add(353);
```

1) Complete the statement to add the value 354 to the tail of the queue.

```
ordersQueue.
____________;
```

Check Show answer

2) Complete the statement to print the element at the head of the queue.

```
System.out.println(ordersQueue.
____________);
```

Check Show answer

3) Given the original queue initialization above, what is the size of the queue *after* three calls to ordersQueue.remove()?

____

Check Show answer

Common Queue methods

The Queue interface has several methods for inserting and removing elements and examining the contents of a queue.

Table 8.4.1: Common Queue methods.

add()	`add(newElement)`
	
	Adds newElement element to the tail of the queue. The queue's size increases by one.
		`// Assume exQueue is: "down" "right" "A"`
		`exQueue.add("B");`
		`// exQueue is now: "down" "right" "A" "B"`

remove()	`remove()`
	
	Removes and returns the element at the head of the queue. Throws an exception if the queue is empty.
		`// Assume exQueue is: "down" "right" "A"`
		`exQueue.remove(); // Returns "down"`
		`// exQueue is now: "right" "A"`

poll()	`poll()`
	
	Removes and returns the element at the head of the queue if the queue is not empty. Otherwise, returns null.
		`// Assume exQueue is: "down"`
		`exQueue.poll(); // Returns "down"`
		`// exQueue is now empty`
		`exQueue.poll(); // Returns null`

element()	`element()`
	
	Returns, but does not remove, the element at the head of the queue. Throws an exception if the queue is empty.
		`// Assume exQueue is: "down" "right" "A"`
		`exQueue.element(); // Returns "down"`
		`// exQueue is still: "down" "right" "A"`

	`peek()` Returns, but does	

peek()	not remove, the element at the head of the queue if the queue is not empty. Otherwise, returns null.	`// Assume exQueue is: "down"`
		`exQueue.peek(); // Returns "down"`
		`// exQueue is still: "down"`

The programmer should select the Queue implementation that is appropriate for the intended task. In this material, we use the LinkedList class, but the examples can be modified to use other Queue implementations, including PriorityQueue, LinkedBlockingQueue, and ArrayBlockingQueue, among others.

CHALLENGE ACTIVITY 8.4.1: Enter the output for queue interface.

422352.2723990.qx3zqy7

Start

Type the program's output

```
import java.util.Queue;
import java.util.LinkedList;

public class ItemsList {
   public static void main(String[] args) {
      Queue<String> flowers = new LinkedList<String>();

      flowers.add("poppy");
      flowers.add("lily");
      flowers.add("lilac");

      System.out.println(flowers.remove());
   }
}
```

```
poppy
```

1 2

Check Next

Exploring further:

- Queue from Oracle's Java documentation.
- LinkedList from Oracle's Java documentation.

- PriorityQueue from Oracle's Java documentation.
- LinkedBlockingQueue from Oracle's Java documentation.
- ArrayBlockingQueue from Oracle's Java documentation.
- Java Collections Framework Overview from Oracle's Java documentation.

8.5 Deque interface

This section has been set as optional by your instructor.

Deque interface

The ***Deque*** (pronounced "deck") interface defined within the Java Collections Framework defines a Collection of ordered elements that supports element insertion and removal at both ends (i.e., at the head and tail of the deque).

A LinkedList is one of several types that implements the Deque interface. A LinkedList implementation of a Deque can be declared and created as `Deque<T> deque = new LinkedList<T>();` where T represents the element's type, such as Integer or String. Java supports automatic conversion of an object, e.g., LinkedList, to a reference variable of an interface type, e.g., Deque, as long as the object implements the interface.

The statements `import java.util.LinkedList;` and `import java.util.Deque;` enable use of a LinkedList and Deque within a program.

Deque's addFirst() and removeFirst() methods allow a Deque to be used as a stack. A ***stack*** is an ADT in which elements are only added or removed from the top of a stack. Deque's addFirst() method adds an element at the head of the deque and increases the deque's size by one. The addFirst() method shifts elements in the deque to make room for the new element. The removeFirst() method removes and returns the element at the head of the deque. If the deque is empty, removeFirst() throws an exception.

PARTICIPATION ACTIVITY 8.5.1: Deque: addFirst() method adds an element at the head, and removeFirst() method returns and removes the element at the head.

Animation captions:

1. The addFirst() method adds an element, such as a String, at the head of the deque.
2. The removeFirst() method returns and removes the element at the head of the deque.

PARTICIPATION ACTIVITY | 8.5.2: Use Deque's addFirst() and removeFirst() methods to insert and retrieve elements.

Given the following code that creates and initializes a Deque.

```
Deque<String> jobsDeque = new LinkedList<String>();

jobsDeque.addFirst("Filter");
jobsDeque.addFirst("Download");
jobsDeque.addFirst("Process");
```

1) What is the value of the element at the head of the deque?

Check Show answer

2) What is the value of the element at the tail of the deque?

Check Show answer

3) Complete the statement to add the value "Draw" at the head of the deque.

```
jobsDeque.
;
```

Check Show answer

4) Complete the statement to remove and print the element at the head of the deque.

```
System.out.println(jobsDeque.
);
```

Check Show answer

Common Deque methods

The Deque interface has methods for inserting and removing elements at both ends of the deque and examining the contents of the deque.

Table 8.5.1: Common Deque methods.

addFirst()	`addFirst(newElement)`
	
	Adds newElement element at the head of the deque. The deque's size increases by one.
		`// Assume exDeque is: 3 5 6`
		`exDeque.addFirst(1);`
		`// exDeque is now: 1 3 5 6`

addLast()	`addLast(newElement)`
	
	Adds newElement element at the tail of the deque. The deque's size increases by one.
		`// Assume exDeque is: 3 5 6`
		`exDeque.addLast(7);`
		`// exDeque is now:    3 5 6 7`

removeFirst()	`removeFirst()`
	
	Removes and returns the element at the head of the deque. Throws an exception if the deque is empty.
		`// Assume exDeque is: 3 5 6`
		`exDeque.removeFirst(); // Returns 3`
		`// exDeque is now:    5 6`

removeLast()	`removeLast()`
	
	Removes and returns the element at the tail of the deque. Throws an exception if the deque is empty.
		`// Assume exDeque is: 3 5 6`
		`exDeque.removeLast(); // Returns 6`
		`// exDeque is now:    3 5`

pollFirst()	`pollFirst()`
	
	Removes and returns the element at the head of the deque if the deque is not empty. Otherwise, returns null.
		`// Assume exDeque is: 3 5 6`
		`exDeque.pollFirst(); // Returns 3`
		`// exDeque is now:    5 6`
		
		`exDeque.pollFirst(); // Returns 5`
		`exDeque.pollFirst(); // Returns 6`
		`// exDeque is now empty`
		
		`exDeque.pollFirst(); // Returns null`

pollLast()	`pollLast()`
	
	Removes and returns the element at the tail of the deque if the deque is not empty. Otherwise, returns null.
		`// Assume exDeque is: 3 5 6`
		`exDeque.pollLast(); // Returns 6`
		`// exDeque is now:    3 5`
		
		`exDeque.pollLast(); // Returns 5`
		`exDeque.pollLast(); // Returns 3`
		`// exDeque is now empty`
		
		`exDeque.pollLast(); // Returns null`

getFirst()	`getFirst()`
	
	Returns, but does not remove, the element at the head of the deque. Throws an exception if the deque is empty.
		`// Assume exDeque is: 3 5 6`
		`exDeque.getFirst(); // Returns 3`
		`// exDeque is still:  3 5 6`

getLast()	`getLast()`
	
	Returns, but does not remove, the element at the tail of the deque. Throws an exception if the deque is empty.
		`// Assume exDeque is: 3 5 6`
		`exDeque.getLast(); // Returns 6`
		`// exDeque is still:  3 5 6`

peekFirst()	`peekFirst()`
	
	Returns, but does not remove, the element at the head of the deque if the deque is not empty. Otherwise, returns null.
		`// Assume exDeque is: 3 5 6`
		`exDeque.peekFirst(); // Returns 3`
		`// exDeque is still:  3 5 6`

peekLast()	`peekLast()`
	
	Returns, but does not remove, the element at the tail of the deque if the deque is not empty. Otherwise, returns null.
		`// Assume exDeque is: 3 5 6`
		`exDeque.peekLast(); // Returns 6`
		`// exDeque is still:  3 5 6`

The programmer should select the Deque implementation that is appropriate for the intended task. In this material, we use the LinkedList class, but the examples can be modified to use other Deque implementations, including ArrayDeque, LinkedBlockingDeque, and ConcurrentLinkedDeque.

CHALLENGE ACTIVITY 8.5.1: Deque interface.

422352.2723990.qx3zqy7

Start

Type the program's output

```
import java.util.LinkedList;
import java.util.Deque;

public class ToInvite {
   public static void main (String[] args) {
      Deque<String> invites = new LinkedList<String>();

      invites.addFirst("Bill");
      invites.removeFirst();
      invites.addFirst("Liz");
      invites.addFirst("Sue");

      System.out.println("1. " + invites.removeFirst());
   }
}
```

```
1. Sue
```

1 2 3

Check Next

Exploring further:

- Deque from Oracle's Java documentation.
- LinkedList from Oracle's Java documentation.
- ArrayDeque from Oracle's Java documentation.
- LinkedBlockingDeque from Oracle's Java documentation.
- ConcurrentLinkedDeque from Oracle's Java documentation.
- Java Collections Framework Overview from Oracle's Java documentation.

8.6 LAB: Ticketing service (Queue)

Given main(), complete the program to add people to a queue. The program should read in a list of people's names including "You" (ending with -1), adding each person to the **peopleInQueue** queue. Then, remove each person from the queue until "You" is at the head of the queue. Include print statements as shown in the example below.

Ex: If the input is:

```
Zadie Smith
Tom Sawyer
You
Louisa Alcott
-1
```

the output is:

```
Welcome to the ticketing service...
You are number 3 in the queue.
Zadie Smith has purchased a ticket.
You are now number 2
Tom Sawyer has purchased a ticket.
You are now number 1
You can now purchase your ticket!
```

422352.2723990.qx3zqy7

LAB ACTIVITY | 8.6.1: LAB: Ticketing service (Queue) 0 / 10

TicketingService.java Load default template...

```java
import java.util.Scanner;
import java.util.LinkedList;
import java.util.Queue;

public class TicketingService {

   public static void main (String[] args) {
      Scanner scnr = new Scanner(System.in);
      String personName = "";
      int counter = 0;
      int youPosition;

      Queue<String> peopleInQueue = new LinkedList<String>();
```

Develop mode Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → **TicketingService.java** (Your program)

Program output displayed here

Coding trail of your work What is this?

```
History of your effort will appear here once you begin
working on this zyLab.
```

8.7 Lab 14 - Queue Implementation with LinkedList

Module 7: Lab 14 - Queue Implementation with LinkedList

This lab includes the following .java files:

```
L13/
└── LinkedQueue.java
└── MyQueue.java
```

LinkedQueue is the *main* in zyBooks, and MyQueue is the interface you are implementing in LinkedQueue. Here is the starter jar with if you would like to code in a different environment: L13.jar.

A Queue is a FIFO (First in First out) data structure meaning that the first data to be inserted is the first to be removed so we can maintain a certain priority. Queues in Computer Science are much like queues/lines in real life. For example, at an ice cream booth people line up and the first person in line is the first to get ice cream and the last person in line is served last. We could maintain chronological order.

Implementing a Queue

We are going to be implementing a Queue using a linked list because as long as we keep references to the head and tail of the list we can have very fast removals and inserts to our queue.

The comments in the source file will guide you through exactly what each function should do.

Exceptions

Your stack may run into errors, such as trying to remove an element from an empty stack or add to a full one. We ask you to use exceptions to deal with such errors in this lab. Recall that exceptions are a way to signal errors, and you can use a `throw` statement to tell Java that something wrong has happened:

```
throw new NoSuchElementException();
```

Remember that you don't have to *catch* these exceptions. Whoever calls our method has to deal with the exception with a try/catch or a throws declaration

Incremental Development

Remember to test your code as you go. The **main** has individual tests written that will each give you output like the following if you implement it correctly:

```
Testing Add:
Add Looks Good
```

Completing the Code

IMPORTANT NOTE:

Remember that methods like poll/remove, add/offer and element/peek() are the SAME except one of them returns a value and one throws an exception. For example, once you have written poll then simply call poll from remove and if it returned null then throw the exception. Also, do **not** modify simulateBooth(). You will need that method for testing.

- Finish the constructor
    - Make sure to change maxCount to the correct value
    - initialize size
- Finish the size method
- Finish the contains method
- Finish add/offer
    - check if the queue is full
    - Create a new Node with name
    - if list is empty then create head and tail
    - if list is not empty insert the node at the tail
    - increment size
- Finish remove/poll
    - check if queue is empty
    - remove the node at the head
    - return the name of the node that was removed
    - decrement size
- Finish peek/element:
    - return the name of the item at the head of the list

Submission

In Submit mode, select "Submit for grading" when you are ready to turn in your assignment.

422352.2723990.qx3zqy7

LAB ACTIVITY | 8.7.1: Lab 14 - Queue Implementation with LinkedList | 7 / 7

Downloadable files

`LinkedQueue.java` and `MyQueue.java` **Download**

Current file: LinkedQueue.java Load default template...

```
import java.util.*;




public class LinkedQueue implements MyQueue{



    public class Node {
        String name;
```

Develop mode Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → **LinkedQueue.java** (Your program) →

Program output displayed here

Coding trail of your work What is this?

```
10/18 T----- W-- R------ M------------------ T--------------
------ W----------- R4 F--7 min:191
```

8.8 Lab 14 - Circular Array Queue

Module 7: Lab 14 - Circular Array Queue

This lab includes the following .java files:

L14/
└── AQueue.java
└── ArrayQueue.java (*This is the only class you need to update.)
└── IQueue.java
└── QueueTestProgram.java

└── TrainCar.java
└── TestMain.java **
Note: TestMain.java is the **main in zyBooks and the only class that will run. However, ArrayQueue does have a main of its own that you can use for testing in another environment.

Download the lab materials here (not including TestMain.java):

L14.jar

Overview

A Queue is a FIFO (First in First out) data structure meaning that the first data to be inserted is the first to be removed so we can maintain a certain priority. Queues in Computer Science are much like queues/lines in real life. For example, at an ice cream booth people line up and the first person in line is the first to get ice cream and the last person in line is served last. We could maintain chronological order.

Instead of using a linked list we are going to be using an array implemented with a circular queue. This class will also be generic unlike the last lab which implemented a queue that held strings.

Generics

The oracle documentation states that generics enable types (classes and interfaces) to be parameters when defining classes, interfaces, and methods.

Before generics were introduced into the language, the compiler had a much harder time identifying errors due to type restrictions and you had to cast your objects when pulling them from data structures such as `Lists`. An example of this:

without generics, needs cast:

```
List list = new ArrayList();
list.add(“hello”);
String s = (String) list.get(0);
```

with generics:

```
List<String> list = new ArrayList<String>();
list.add(“hello”);
String s = list.get(0);
```

Because our underlying data structures are generic you might try to instantiate a generic array like this:

```
E[] arr = new E[size];
```

This is not possible.

This is because when the code runs, the type `E` is "not known" (this is called type erasure). This idea works in many other languages, but it doesn't work in `Java`. Other languages will compile each generic use into a custom class. For example, in `C++` a `vector` of `int`, a `vector` of `char`, and `vector` of `Shape` would each end up having a custom class.

Instead, when instantiating a new generic array, you should use the following code:

```
@SuppressWarnings("unchecked")
E[] arr = (E[]) new Object[size];
```

NOTE	we add the `@SuppressWarnings("unchecked")` because this code will cause a warning. Unchecked casting is normally dangerous. Do you think there is a situation where this code could be dangerous?

Writing code that generates warnings can desensitize you to the bad idea of writing code that generates warnings. Don't leave in warning-generating code unless you understand exactly why that code is there and why it is unavoidable. Generic code that uses arrays is one such justifiable circumstance; you are unlikely to encounter many others.

To learn more about generics, which look at the Oracle Java Tutorials here.

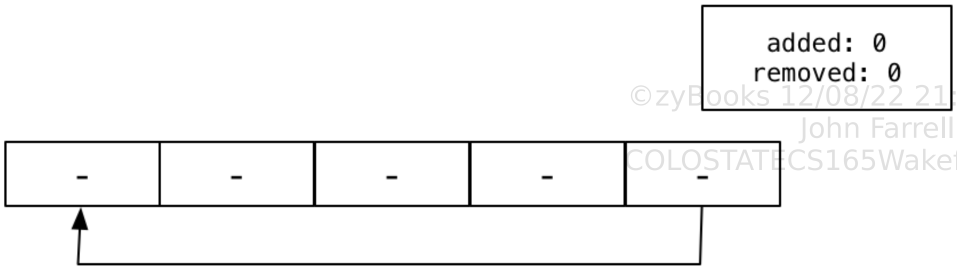
Circular Array Queues

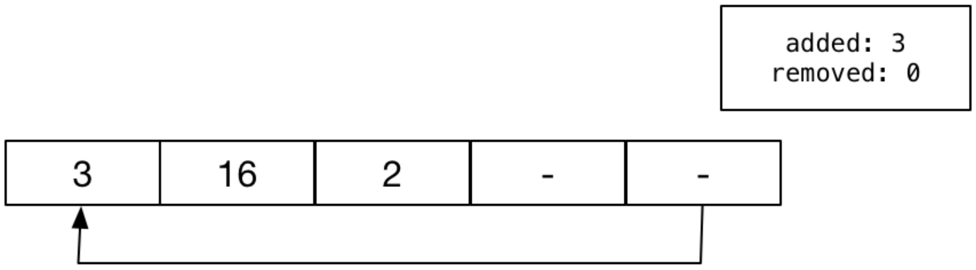
During this lab we will be creating a circular array queue with a maximum capacity. This will give us an idea of how queues are implemented outside of academia. The capacity is a very realistic constraint.

Notice the notes in the aid below about using the modulo operator to find the next position to insert and remove, this will be very helpful.

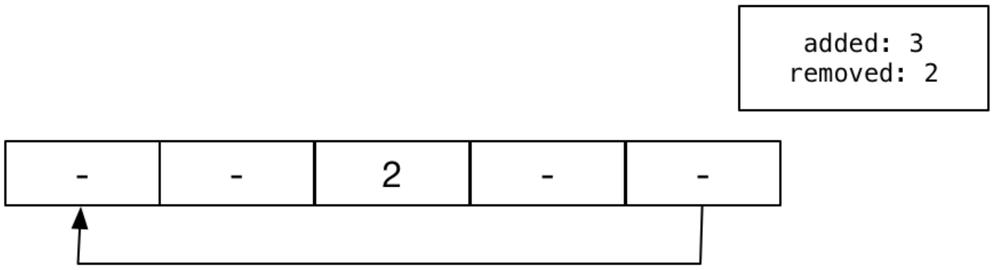
The following image is a visual aid as to how circular array queues work:

A circular buffer is a useful property to have when implementing a queue. It makes it so that if one element is removed, the rest of the elements in the queue do not need to be shuffled around. The following visual aid should be used to gain a better understanding of how to implement the `add()`, `offer()`, `remove()`, and `poll()` methods.

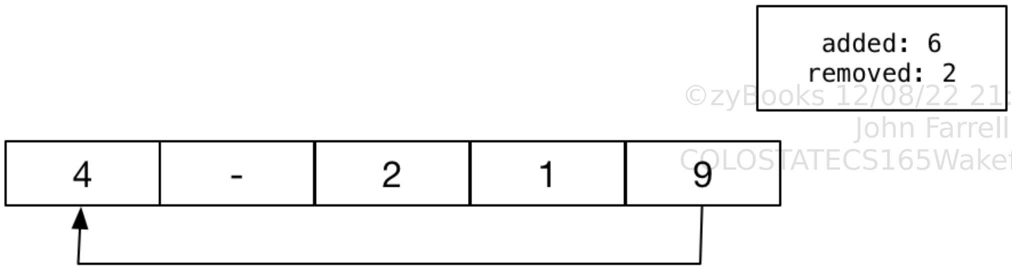




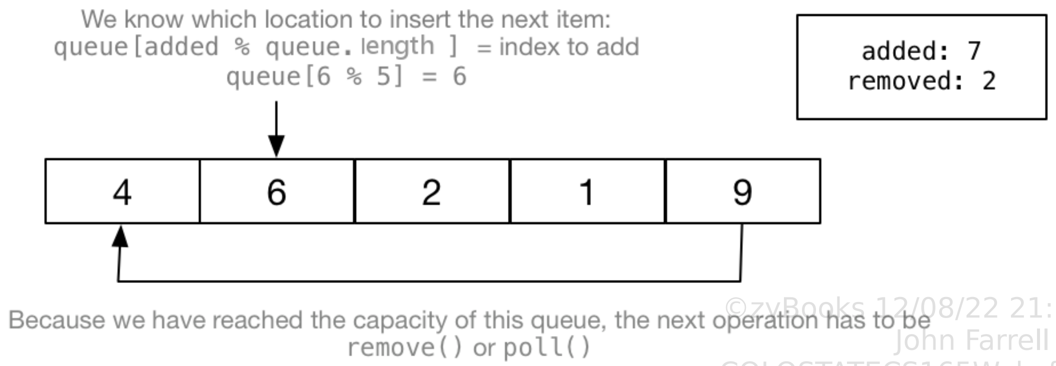
The added variable keeps track of the total number of elements added to the queue. The only time it's reset to `0` is when the `clear()` method is called on the queue.

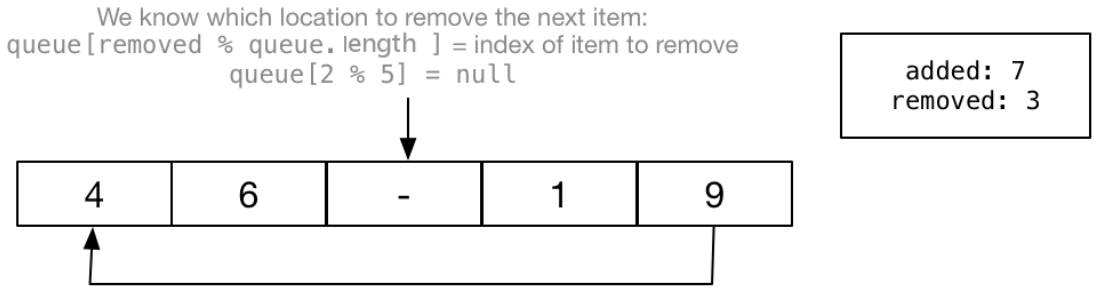


Similarly removed keeps track of the total number of elements removed from the queue.



When we have reached the end of the queue, so long as the size does not equal the capacity, in other words -> (`added−removed != queue.length`), a circular buffer will store the next element at the start of the queue.


We know which location to insert the next item:
queue[added % queue.length] = index to add
queue[6 % 5] = 6
added: 7
removed: 2
4
6
2
1
9
Because we have reached the capacity of this queue, the next operation has to be
remove() or poll()


We know which location to remove the next item:
queue[removed % queue.length] = index of item to remove
queue[2 % 5] = null
added: 7
removed: 3
4
6
-
1
9

IMPORTANT OBSERVATION:

Notice that when there were items in the queue and some were removed from the front we did not have to shift all the elements down which could become quite annoying if the queue was large. Instead we simply keep track of where to insert the next item and where to remove the next item.

Implementing the Code:

You may implement the methods in whatever order makes sense to you but make sure to at least do the constructor first. The comments above each method provide ample documentation about how to implement them.
Another Note:
The IQueue interface and AQueue abstract class are completed for you. IQueue describes the methods a Queue must implement and AQueue deals with the duplicate methods like add/offer, remove/poll, peek/element() and throws the appropriate exceptions instead of failing silently. The TrainCar class is provided for testing purposes.

Testing:

In the main method in the ArrayQueue class there are a few basic tests at the beginning just to sanity check your code is working. Again, it's a good idea to run this code in an IDE so that you can take advantage of these tests and write your own as well. The expected output would be:

```
Offering to Queue:
     Offer(1) -> true
     Offer(2) -> true
     Offer(3) -> true
     size() should be 3 -> 3
     contains(2) should be true -> true
Adding more elements using add():
     add(4);  add(5); add(6); add(7); add(8); add(9); add(10);
     size() should be 10 -> 10
Trying to offer to a full queue:
     offer(11) should return false -> false
     size() should still be 10 -> 10
Polling and removing some elements:
     poll() should be 1 -> 1
     remove() should be 2 -> 2
Testing clear:
     size() should be 0 -> 0
Trying to poll from an empty queue:
     poll() should return null -> null
     size() should still be 0 -> 0
Testing contains() with TrainCar objects:
     contains(red Caboose: 12345) should be true -> true
```

Once you have completed this uncomment the lines at the bottom of the code that say:

```
QueueTestProgram.printFailedTests(thousand,
                                  ArrayBlockingQueue::new,
                                  ArrayQueue::new);
```

and run it if everything is working your code will not have any failed tests and will tell you all tests passed. If there is a problem you will get some output that looks something like this:

```
RuntimeException ArrayBlockingQueue.contains(22) => true, but..
ArrayQueue.contains(22) => false Method calls that expose bug:
[offer(29), offer(22), add(7), contains(22)]
```

which tells you that a test with code something like:

```
ArrayQueue<Integer> test = new ArrayQueue<Integer>(10);
test.offer(22);
test.offer(29);
test.add(7);
test.contains(22);
```

was run and produced the error. The part that says ArrayQueue is your output and the part that says ArrayBlockingQueue is the correct answer.

Submission

In Submit mode, select "Submit for grading" when you are ready to turn in your assignment.

422352.2723990.qx3zqy7

LAB ACTIVITY | 8.8.1: Lab 14 - Circular Array Queue | 0 / 7

Downloadable files

AQueue.java , ArrayQueue.java , IQueue.java , TrainCar.java , and QueueTestProgram.java

Download

File is marked as read only | Current file: AQueue.java

```
import java.util.NoSuchElementException;

/**
 * AQueue provides default implementations for several no-brainer methods in the IQueue in
 * <p>
 * Some methods in IQueue are pairs that do the same job, but report failure differently.
 * Implement the fail-by-throwing methods here, in this abstract class,
 * by calling the paired method and checking the return value, then throwing as appropriate
 * Why is defining these methods here desirable? What are some alternatives?
 * <p>
 * created by {@code cspfrederick} and {@code garethhalladay} Fall17 <br>
 * inspired by Chris Wilcox
 */
public abstract class AQueue<E> implements IQueue<E> {
    // The add method is similar to the offer method except for handling failure.
    // Instead of returning false, it throws an IllegalStateException.
    @Override
```

Develop mode | **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → AQueue.java (Your program) → O

Program output displayed here

Coding trail of your work What is this?

```
10/20 R-- R- min:15
```

8.9 LAB: Palindrome (Deque)

This section's content is not available for print.

8.10 LAB: Unique random integers (HashSet)

This section's content is not available for print.