1. Consider the following statements:

**Array Queue<int> queue = new Array Queue();**

**int x, y;**

Show what is output by the following segment of code:

x = 4; y = 5; queue.enqueue(x); queue. enqueue(y); x = queue.front( ); queue.dequeue( ); queue. enqueue(x + 5); queue. enqueue(16); queue. enqueue(x); queue. enqueue(y - 3); system.out.println( "Queue Elements: "); while (!queue.isEmptyQueue())

{

system.out.println(queue.front() ); queue.dequeue();

}

Queue Elements:

4

5

9

16

4

2

1. What is the output of the following program segment?

linkedQueue<int> queue = new linkedQueue(); queue.enqueue(10); queue.enqueue(20); cout << queue.front() << endl; queue.dequeue(); queue.enqueue(2 \* queue.back()); queue.enqueue(queue.front()); queue. enqueue(5); queue. enqueue(queue.back() - 2); linkedQueue<int> tempQueue = new linkedQueue() ; tempQueue = queue; while (!tempQueue.isEmptyQueue())

{

system.out.println( tempQueue.front() ); tempQueue.dequeue();

}

system.out.println( queue.front() ); system.out.println(queue.back() );

Output:

10

20

10

8

3- Consider the following statements: ArrayStack<int> stack = new ArrayStack(); ArrayQueue<int> queue = new ArrayQueue();

int x;

Suppose the input is:

14 8 14 22 64 35 19 32 7 11 13 30 -999

Show what is written by the following segment of code:

stack.push(0); queue.enqueue(0); system.out.println( x); while (x != -999)

{ switch (x % 4) { case 0: stack.push(x); break;

case 1: if (!stack.isEmptyStack())

{ system.out.println( "Stack Element = " ); system.out.println( stack.top()); stack.pop();

} else

system.out.println( "Sorry, the stack is empty." ); break; case 2: queue.enqueue(x); break; case 3: if (!queue.isEmptyQueue())

{

system.out.println( "Queue Element = " ); system.out.println( queue.front()); queue.dequeue();

} else

system.out.println( "Sorry, the queue is empty." ); break;

} //end switch system.out.println( x);

} //end while system.out.println( "Stack Elements: "); while (!stack.isEmptyStack())

{

system.out.println( stack.top() ); stack.pop(); }

system.out.println( "Queue Elements: "); while (!queue.isEmptyQueue())

{

system.out.println( queue.front() ); queue.dequeue();

}

**Output:**

14

Stack Element =

0

8

Sorry, the stack is empty.

14

Queue Element =

0

22

Stack Element =

14

64

Stack Element =

22

35

Sorry, the stack is empty.

19

Queue Element =

14

32

Stack Element =

19

7

Sorry, the stack is empty.

11

Queue Element =

32

13

Stack Element =

11

30

Queue Element =

35

Stack Elements:

13

11

Queue Elements:

30

1. Suppose that queue is a queueType object and the size of the array implementing queue is 100. Also, suppose that the value of queueFront is 50 and the value of queueRear is 99. a- What are the values of queueFront and queueRear after adding an element to queue? b- What are the values of queueFront and queueRear after removing an element from queue?

a) After adding an element to the queue:

* queueFront: Remains unchanged at 50.
* queueRear: Becomes 0 (wraps around to the beginning of the array).

Explanation:

* When an element is added to a queue, it's added at the rear.
* Since queueRear is already at the end of the array (99), it wraps around to the beginning (index 0) to accommodate the new element.
* queueFront stays the same because elements are removed from the front, not the rear.

b) After removing an element from the queue:

* queueFront: Becomes 51.
* queueRear: Remains unchanged at 0.

Explanation:

* When an element is removed from a queue, it's removed from the front.
* queueFront is incremented to point to the next element in the queue.
* queueRear stays the same because it only changes when elements are added.

Key points:

* queueFront: Points to the index of the first element in the queue.
* queueRear: Points to the index of the last element added to the queue (one position after the last element).
* When queueRear reaches the end of the array, it wraps around to the beginning to create a circular queue.

1. Suppose that queue is a queueType object and the size of the array implementing queue is 100. Also, suppose that the value of queueFront is 99 and the value of queueRear is 25. a- What are the values of queueFront and queueRear after adding an element to queue? b- What are the values of queueFront and queueRear after removing an element from queue?

Here are the answers to your questions:

a) After adding an element to the queue:

* queueFront: Remains unchanged at 99.
* queueRear: Becomes 26.

Explanation:

* Elements are added to the rear of a queue.
* Since queueRear is 25, it's incremented to 26 to accommodate the new element.
* queueFront doesn't change during enqueue operations.

b) After removing an element from the queue:

* queueFront: Becomes 0.
* queueRear: Remains unchanged at 26.

Explanation:

* Elements are removed from the front of a queue.
* Since queueFront is 99 (the end of the array), it wraps around to 0 to point to the next element.
* queueRear doesn't change during dequeue operations.

Key points:

* queueFront: Index of the first element.
* queueRear: Index of the next available position for insertion (one past the last element).
* Circular queue: queueFront and queueRear wrap around to the beginning when reaching the end of the array.

1. Suppose that queue is a queueType object and the size of the array implementing queue is 100. Also, suppose that the value of queueFront is 25 and the value of queueRear is 75. a- What are the values of queueFront and queueRear after adding an element to queue? b- What are the values of queueFront and queueRear after removing an element from queue?

Here are the answers to your questions:

a) After adding an element to the queue:

* queueFront: Remains unchanged at 25.
* queueRear: Becomes 76.

Explanation:

* Elements are added to the rear of a queue.
* Since queueRear is 75, it's incremented to 76 to accommodate the new element.
* queueFront doesn't change during enqueue operations.

b) After removing an element from the queue:

* queueFront: Becomes 26.
* queueRear: Remains unchanged at 76.

Explanation:

* Elements are removed from the front of a queue.
* queueFront is incremented to 26 to point to the next element after removal.
* queueRear doesn't change during dequeue operations.

Key points:

* queueFront: Index of the first element in the queue.
* queueRear: Index of the next available position for insertion (one past the last element).
* queueFront and queueRear only move in their respective directions (front increments for dequeue, rear increments for enqueue).

1. Suppose that queue is a queueType object and the size of the array implementing queue is 100. Also, suppose that the value of queueFront is 99 and the value of queueRear is 99. a- What are the values of queueFront and queueRear after adding an element to queue? b- What are the values of queueFront and queueRear after removing an element from queue?

a) After adding an element to the queue:

* This situation indicates a full queue. Attempting to add an element when queueFront and queueRear are both at 99 (the end of the array) will likely result in an error or exception, as there's no space for the new element.

b) After removing an element from the queue:

* queueFront: Becomes 0 (wraps around to the beginning).
* queueRear: Remains unchanged at 99.

Explanation:

* Removing an element from the front is possible even when queueFront is at the end of the array. It wraps around to the beginning, maintaining the circular queue structure.
* queueRear doesn't change during dequeue operations.

Key points:

* Full queue: When queueFront and queueRear are both at the end of the array, it usually signifies a full queue, preventing further enqueue operations.
* Circular queue: Both queueFront and queueRear can wrap around to the beginning when they reach the end of the array.
* Dequeue from a full queue: Removing elements from a full queue is still possible, as queueFront can wrap around to create space

1. Write a function, **reverseQueue**, that takes as a parameter a queue object and uses a stack object to reverse the elements of the queue.

a) After adding an element to the queue:

* This situation indicates a full queue. Attempting to add an element when queueFront and queueRear are both at 99 (the end of the array) will likely result in an error or exception, as there's no space for the new element.

b) After removing an element from the queue:

* queueFront: Becomes 0 (wraps around to the beginning).
* queueRear: Remains unchanged at 99.

Explanation:

* Removing an element from the front is possible even when queueFront is at the end of the array. It wraps around to the beginning, maintaining the circular queue structure.
* queueRear doesn't change during dequeue operations.

Key points:

* Full queue: When queueFront and queueRear are both at the end of the array, it usually signifies a full queue, preventing further enqueue operations.
* Circular queue: Both queueFront and queueRear can wrap around to the beginning when they reach the end of the array.
* Dequeue from a full queue: Removing elements from a full queue is still possible, as queueFront can wrap around to create space

1. Suppose an initially empty queue *Q* has performed a total of 32 enqueue operations, 10 first operations, and 15 dequeue operations, 5 of which returned null to indicate an empty queue. What is the current size of *Q*?

Explanation:

* Enqueue operations: Add elements to the queue, increasing its size. There have been 32 enqueue operations, so they added 32 elements.
* First operations: Only retrieve the front element without removing it, so they don't affect the size.
* Dequeue operations: Remove elements from the queue, decreasing its size. There have been 15 dequeue operations, but only 10 of them were successful (5 returned null, indicating an empty queue). So, only 10 elements were actually removed.

Calculating the size:

Starting with an empty queue:

* Add 32 elements: 0 + 32 = 32
* Remove 10 elements: 32 - 10 = 22

Therefore, the current size of Q is 22.

1. What values are returned during the following sequence of deque *(double ended queue)* ADT operations, on an initially empty deque? addFirst(3), addLast(8), addLast(9), addFirst(1), last( ), isEmpty( ), addFirst(2), removeLast( ), addLast(7), first( ), last( ), addLast(4), size( ), removeFirst( ), removeFirst( ).
2. addFirst(3): Adds 3 to the front of the deque.
3. addLast(8): Adds 8 to the back of the deque.
4. addLast(9): Adds 9 to the back of the deque.
5. addFirst(1): Adds 1 to the front of the deque.
6. last( ): Returns the last element, which is 9.
7. isEmpty( ): Returns False, as the deque is not empty.
8. addFirst(2): Adds 2 to the front of the deque.
9. removeLast( ): Removes and returns the last element, which is 9.
10. addLast(7): Adds 7 to the back of the deque.
11. first( ): Returns the first element, which is 2.
12. last( ): Returns the last element, which is 7.
13. addLast(4): Adds 4 to the back of the deque.
14. size( ): Returns the size of the deque, which is 6.
15. removeFirst( ): Removes and returns the first element, which is 2.
16. removeFirst( ): Removes and returns the new first element, which is 1.

Final state of the deque: [3, 8, 7, 4]

# Good Luck