Queues

Queue overview

- A queue is a FIFO list.
- Insertion is only allowed from one end of the queue called the ***rear*** of the queue.
- Removal is only allowed from one end of the queue called the ***front*** of the queue.

Specification of the ADT queue

AbstractDataType *Queue*
{
instances

ordered list of elements; one accessible end is called the *front* and the other accessible end is called the *rear.*

operations

isEmpty ():	returns *true* if the queue is empty; returns *false* otherwise.
peek():	returns the front element of the queue.
enqueue(x):	adds element *x* at the rear of the queue.
dequeue():	removes the front element from the queue & returns it.

}

The Interface Queue

- The ADT queue is encoded as follows:

```
public interface Queue
{
   public boolean isEmpty();
   public Object peek();
   public void enqueue(Object theObject);
   public Object dequeue();
}
```

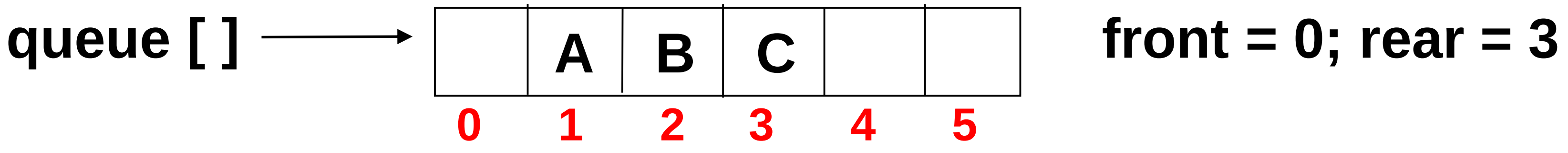
Introduction to the data structures

- We shall show two standard encodings: one using an array and the other a linked list.
- Firstly, the array, which uses a one-dimensional array with two pointers, front and rear defined as follows:
  - ➢ front is one position "in front of" the first element;
  - ➢ rear gives the position of the last element inserted.
- When we initialise a queue, we set

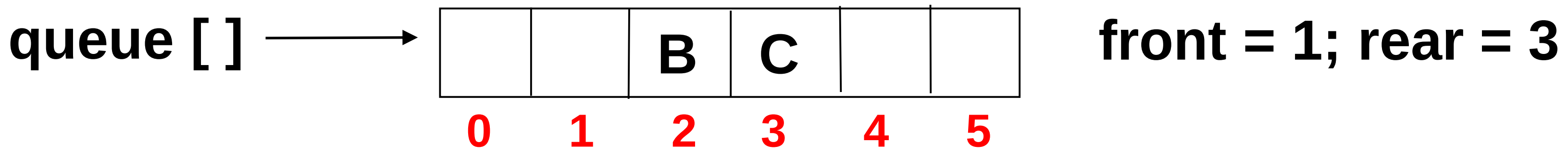
```
front = rear = 0;
```

Linear array queue example

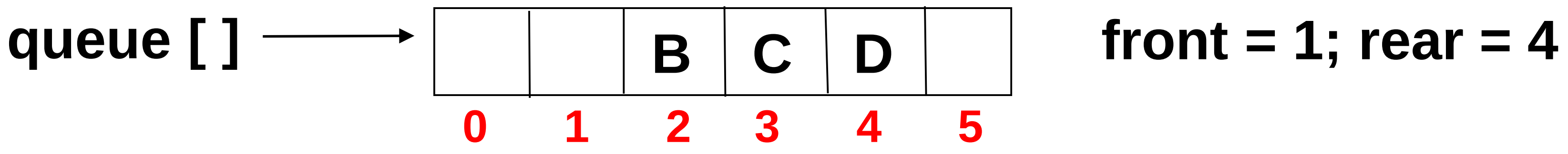
- Suppose, for example, we have "at some stage":



- Remove A and obtain:

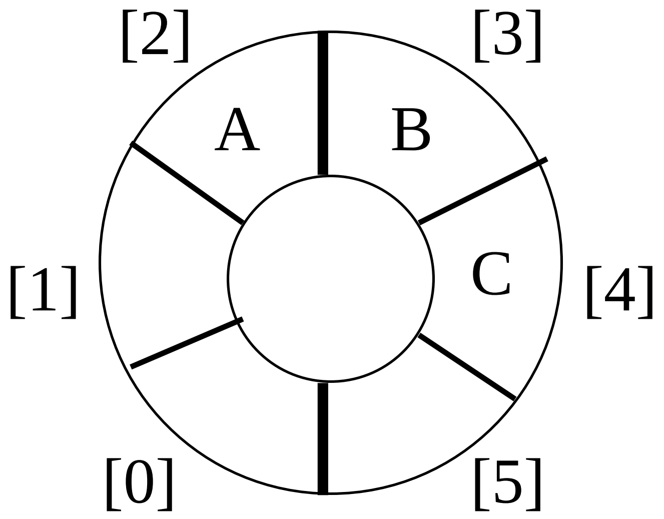


- Adding D to the queue:



Circular array example

- We often use a circular one-dimensional array and an example configuration with 3 elements is shown below:

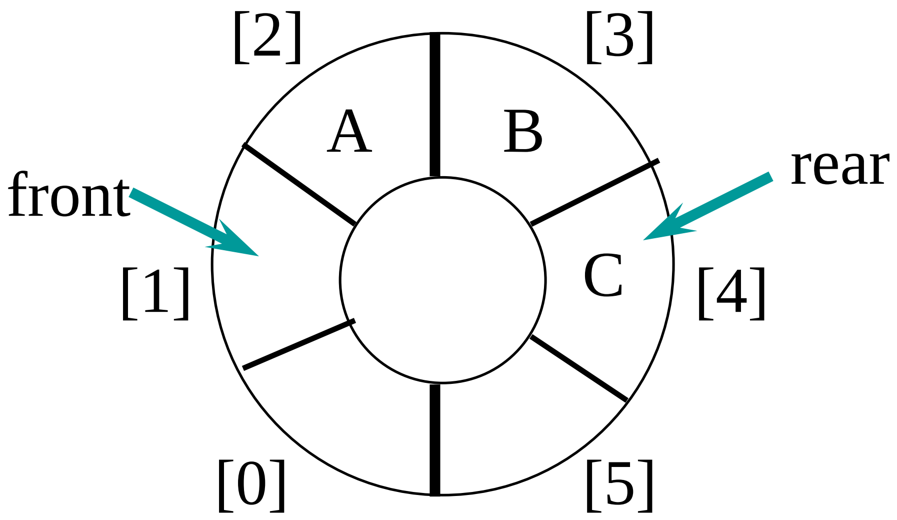


- Note again how the front of the queue (A) has moved from its starting position (presumably index zero).

Circular array : details of `front` & `rear`

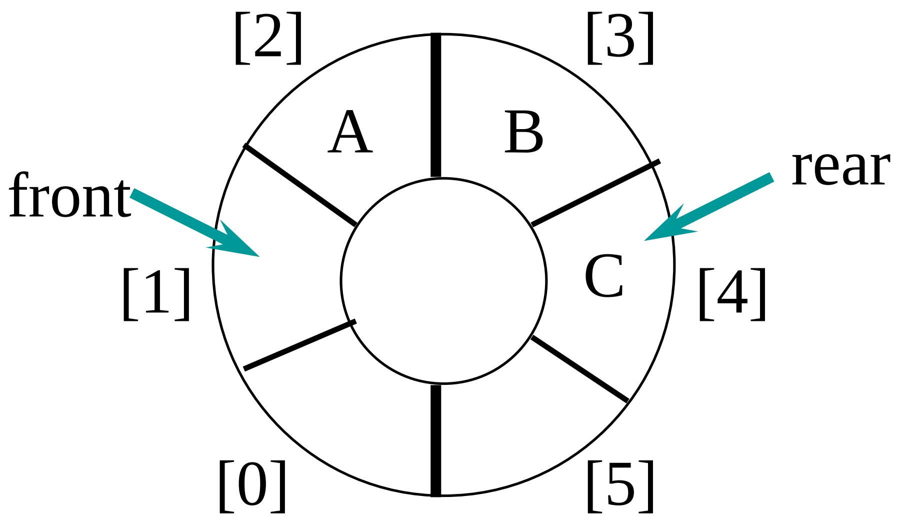
- Here **`front`** and **`rear`** are defined as:
  - ➢ **`front`**: one position anti-clockwise from the first element;
  - ➢ **`rear`**: the position of the last element inserted.

  i.e.

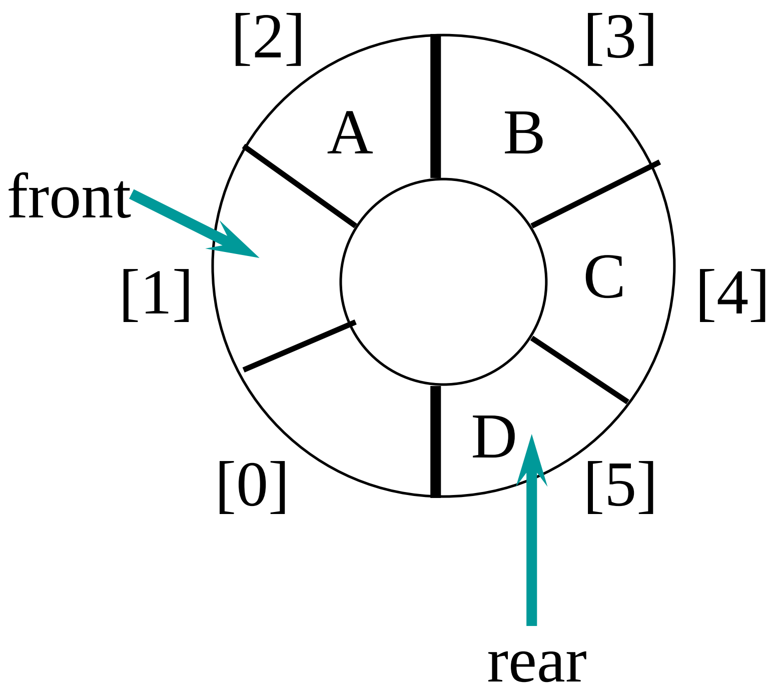


Circular array: adding an element

- Before insertion:



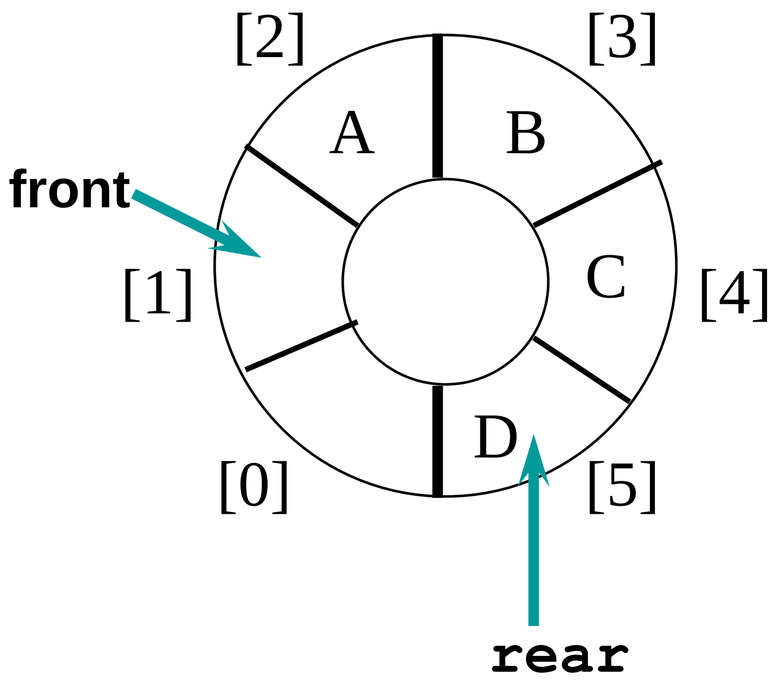
- After insertion:



i.e. move **`rear`** one place clockwise (by adding 1 to its value) and *then* insert using **`queue[rear]`**.

Circular array: a special case of insertion

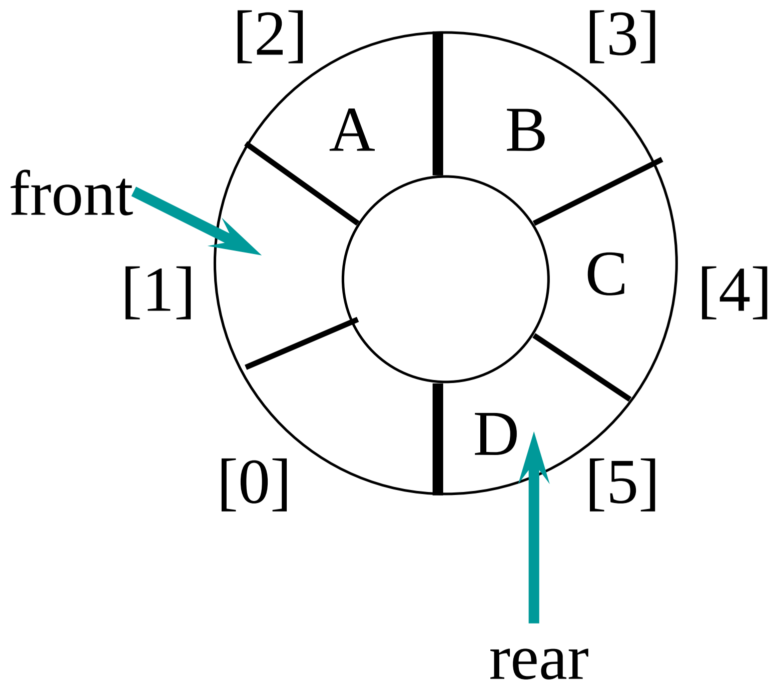
- Clearly, "at some stage" when inserting, the value of rear will become equal to n-1 (where n is the size of the array).
- Continuing with the previous example:



- If we add another element now then we need **`rear`** to point to 0 (not 6).
- To do this, we must use the modulo operator %.

Circular array: a special case of insertion cont./

- The modulo operator calculates the remainder of integer division. So, whilst 3 % 6 gives 3, 6 % 6 gives 0 and, in our example below, moving **`rear`** clockwise (by adding 1 to it) and then using the modulo operator gives the next, **correct** position of **`rear`**.

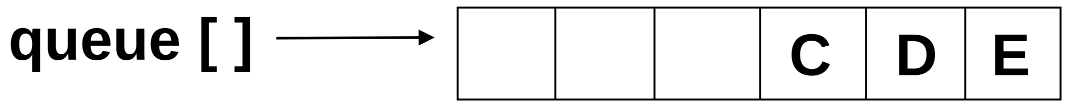


We code this as

```
rear =(rear+1)%queue.length;
```

Linear array: dealing with a "full" queue

- With a linear queue, the queue moves "up" the array.
- Suppose, for example, we have "at some stage":

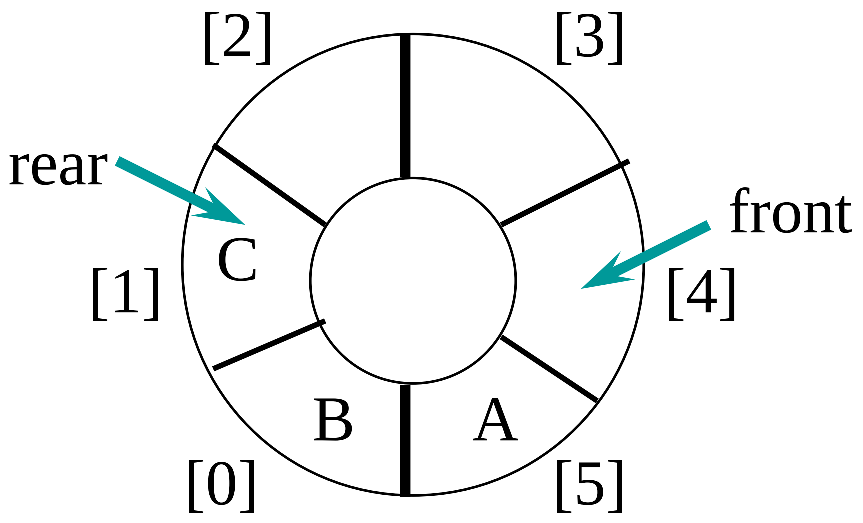


front = 2; rear = 5

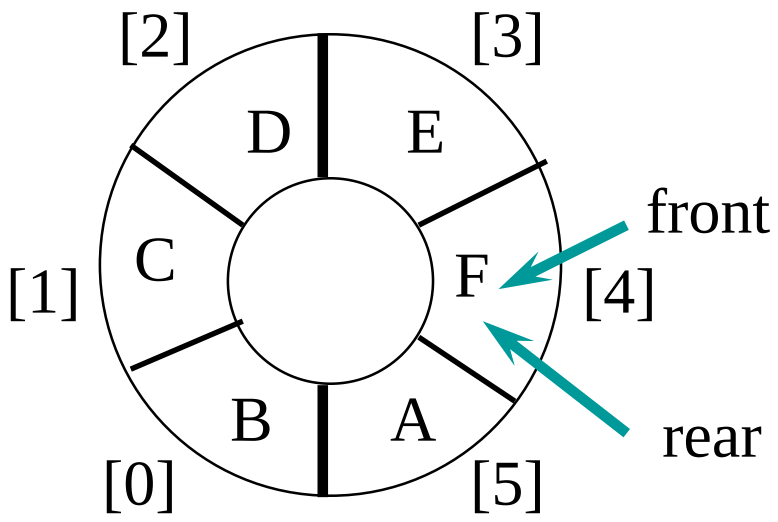
- Is this a "full" queue?
- Clearly not and yet we cannot insert any more elements.
- So, in general what is the condition for a full queue?
  - When the number of elements is one less than the size of the array.
- In the example above, we need to move the queue back to the LHS of the array – if we want to add any more elements.

Circular array: dealing with a "full" queue

- Of course, our circular queue could also become full. For example, suppose that we have:



- Adding D then E then F now gives …



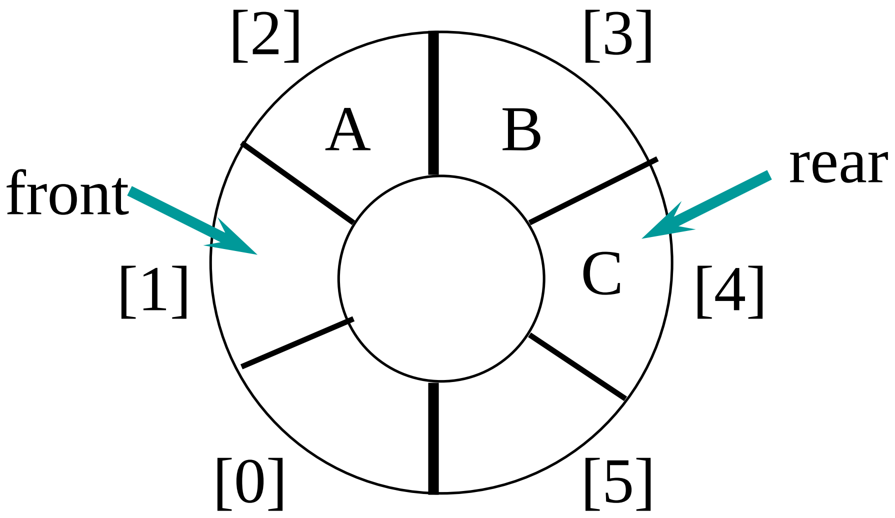
Circular array: dealing with a "full" queue cont./

- So when this queue is "full", **`front`** is equal to **`rear`**.
- But this was the condition for an empty queue!

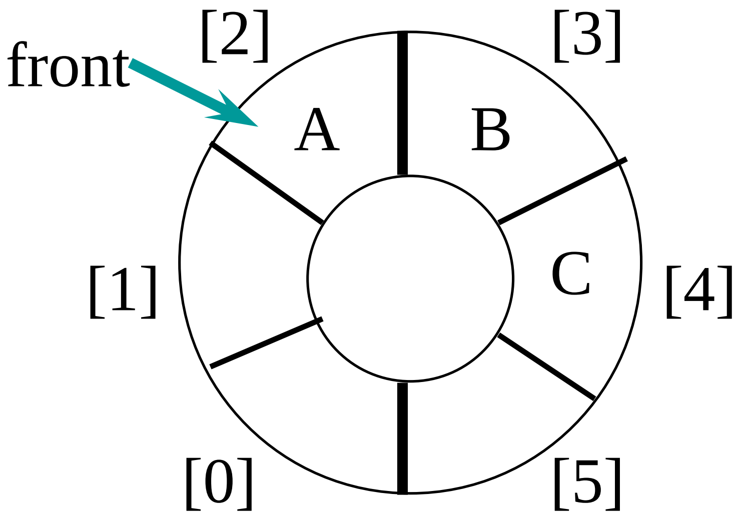
  which means that we cannot distinguish between a full queue and an empty queue!
- The usual solution to this problem is to never let the queue get full i.e. when the addition of an element will cause the queue to be full, to increase the array size.

Circular array: removing an element

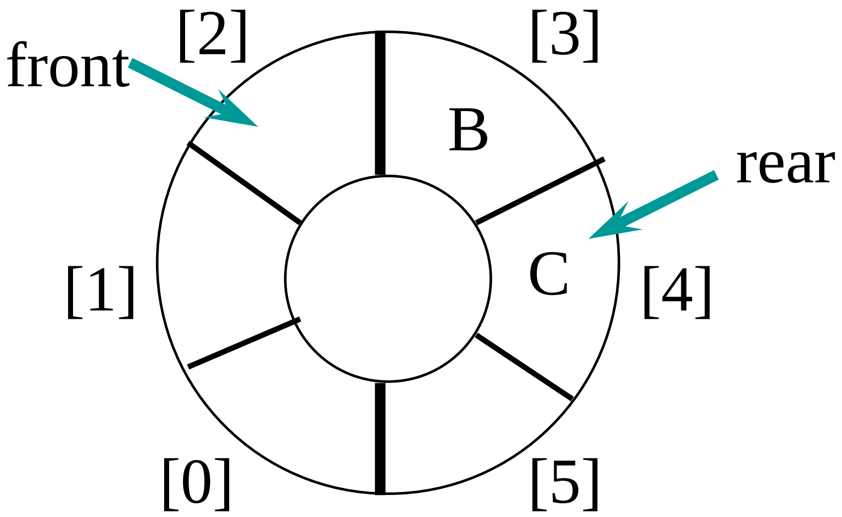
- Before removal:



- During removal:



- After removal:



The class ArrayQueue

- Below is the code/pseudocode for a circular array implementation of **`ArrayQueue`**. The linear array implementation is homework.

```
public class ArrayQueue implements Queue
{
      // data members
      int front, rear;
      Object[ ] queue;

      // constructors come here

      // queue interface methods come here

}
```

Two constructors for ArrayQueue

Two constructors for **`ArrayQueue`** are:

```
public ArrayQueue(int initialCapacity)
{
    if (initialCapacity < 1)
         throw new IllegalArgumentException
           ("initialCapacity must be >= 1");
    queue = new Object [initialCapacity + 1];
    front = rear = 0;
}

public ArrayQueue()
{
    this(10);
}
```

The four methods of the interface Queue

```
 // @return true if queue is empty
 public boolean isEmpty()
 {
    return front == rear;
 }

/** @ return front element of queue
 *  @ return null if queue is empty
 */
public Object peek()
{
    if (isEmpty())
          return null;
    else
          return queue[(front+1) % queue.length];
}
```

The four methods of the interface Queue cont./

- **`enqueue`** pseudocode

```
enqueue
    if queue is full
            double array size & allocate a new array newQueue
            copy elements into new array
            switch to newQueue and set front and rear pointers

    put the new element at the rear of the queue
```

The four methods of the interface Queue cont./

- **`dequeue`** `pseudocode`

```
dequeue
    if queue is empty
        return null
    else
        remove and return the element at the front
```

Linked implementation of ADT queue

```
public class LinkedQueue implements Queue
{
    protected LinkNode front, rear;

    public LinkedQueue()
    {
          front = rear = null;
    }
}
```

The four methods of the interface Queue

```
// @return true if list is empty
public boolean isEmpty()
{
   return front == null;
}

//  returns the element from the front of the queue
public Object peek()
{
    if (isEmpty())
         return null;
    else
         return front.theObject;
}
```

The four methods of the interface Queue cont./

```
public void enqueue(Object theObject)
{
    // creates a node for the new element
    LinkNode p = new LinkNode(theObject, null);
    // append p to the queue
    if (front == null)
        front = p;          // empty queue
    else
        rear.next = p;      // non-empty queue
    rear = p;
}
```

The four methods of the interface Queue cont./

```
public Object dequeue()
{
  if (isEmpty())
      return null;
  Object frontObject = front.theObject;
  front = front.next;
  if (isEmpty())
      rear = null;           // to allow garbage collection
  return frontObject;
}
```