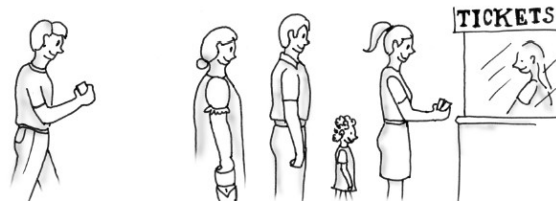




Queues

- A **queue** is another name for a **waiting line**:



- Used within operating systems and to simulate real-world events.
 - Come into play whenever processes or events must wait
- Entries organized **first-in, first-out**.

Terminology

- Item added first, or earliest, is at the front of the queue
- Item added most recently is at the back of the queue
- Additions to a software queue must occur at its back.
- Client can look at or remove only the entry at the front of the queue



The ADT Queue

DATA

- A collection of objects in chronological order and having the same data type

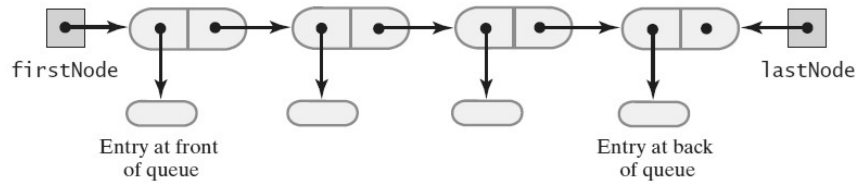
OPERATIONS

PSEUDOCODE	UML	DESCRIPTION
enqueue(newEntry)	+enqueue(newEntry: integer): void	Task: Adds a new entry to the back of the queue.
dequeue()	+dequeue(): T	Task: Removes and returns the entry at the front of the queue.
getFront()	+getFront(): T	Task: Retrieves the queue's front entry without changing the queue in any way.
isEmpty()	+isEmpty(): boolean	Task: Detects whether the queue is empty.
clear()	+clear(): void	Task: Removes all entries from the queue.





Linked-list Representation of a Queue



```

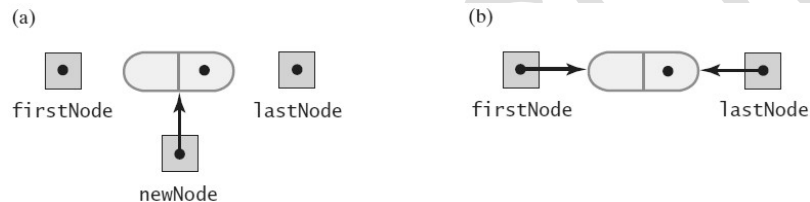
public class linkedQueue <T extends Comparable<T>> {
    private Node<T> first;
    private Node<T> last;

    public boolean isEmpty(){ return (first==null) && (last==null); }
    public void clear(){
        first = null;
        last = null;
    }
}

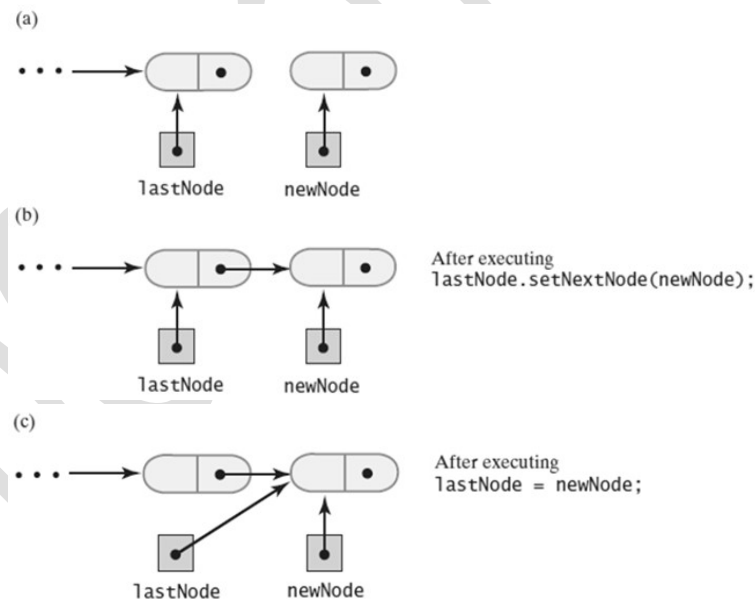
```

- The definition of **enqueue** Performance is **O(1)**:

- Adding a new node to an empty chain



- Adding a new node to the end of a nonempty chain that has a **tail** reference



```

public void enqueue(T data){
    Node<T> newNode = new Node<T>(data);
    if(isEmpty())
        first=newNode;
    else
        last.next = newNode;
    last = newNode;
}

```



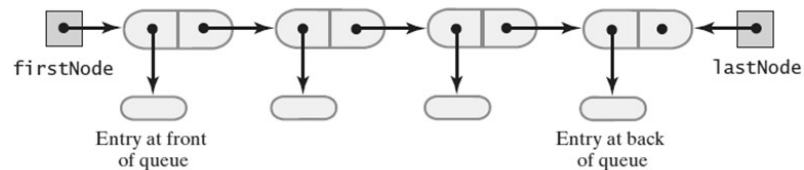


- Retrieving the front entry:

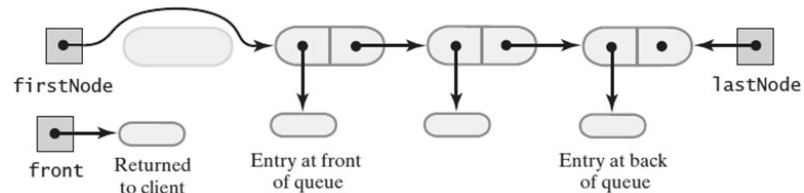
```
public T getFront(){
    if(!isEmpty())
        return first.data;
    return null;
}
```

- Removing the front entry (**dequeue**):
 - A queue of more than one entry:

(a)

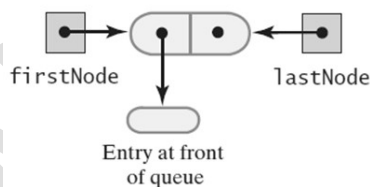


(b)

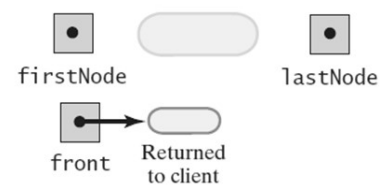


- A queue of one entry:

(a)



(b)



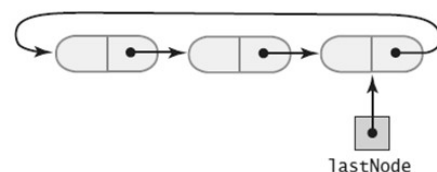
```
public T dequeue(){
    T front = getFront();
    if(!isEmpty())
        first = first.next;
    if(first==null)
        last = null;
    return front;
}
```

Circular Linked Implementations of a Queue

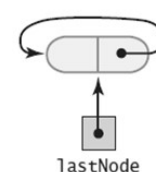
A circular linked chain with an external reference to its last node that

- a) has more than one node; b) has one node; c) is empty

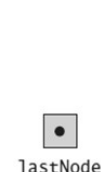
(a)



(b)

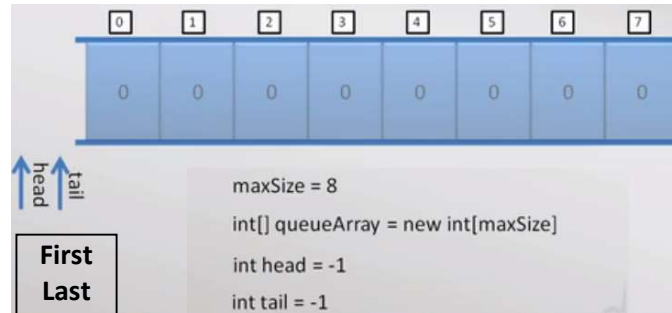


(c)



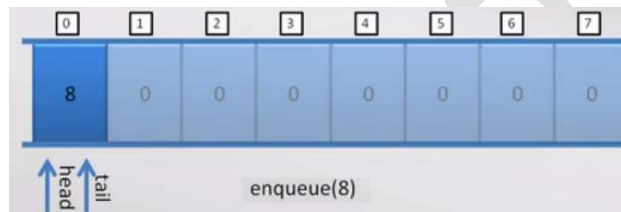


Array implementation of a Queue



- **enqueue()**: add new item at after **last (tail)**.
- **dequeue()**: remove item from **first (head)**.

enqueue(8)



enqueue (12)

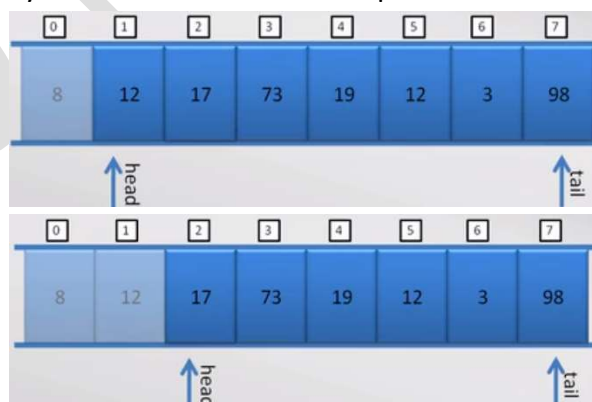


After a number of enqueues:

dequeue(): returns the item pointed by **head** and advances **head** pointer



dequeue()

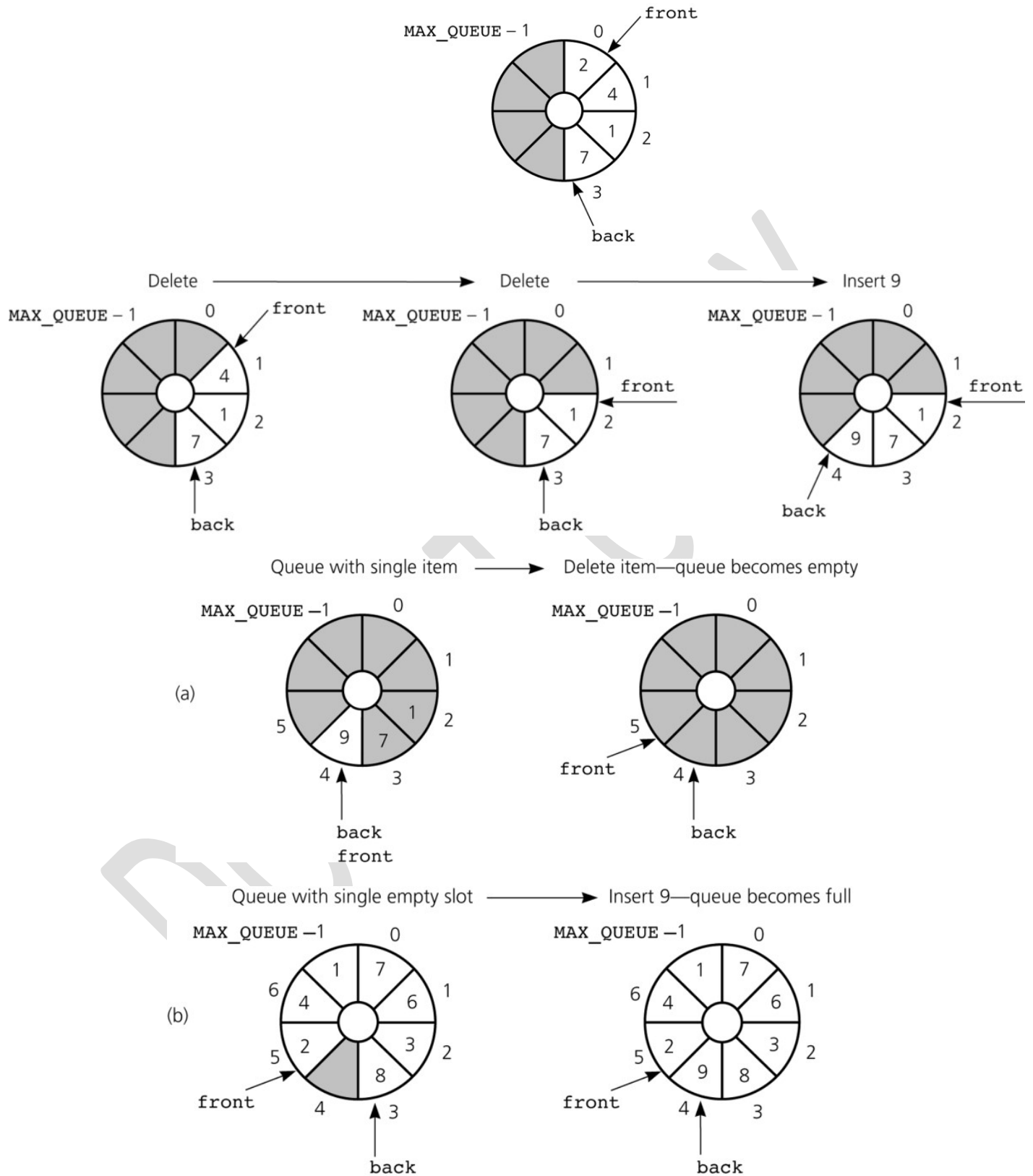


enqueue (27) ?? How to advance **tail**?? We have space at the beginning?? Shift??





Circular Queue





- **To detect circular queue-full and queue-empty conditions**
 - Keep a **count** of the queue items
- **To initialize the circular queue, set:**
 - front to -1
 - back to -1
 - count to 0
- **Inserting into a circular queue:**

```
If(count < MAX_QUEUE) // free
    back = (++back) % MAX_QUEUE;
    items[back] = newItem;
    ++count;
If(count == 1) // first item
    front = back;
```
- **Deleting from a circular queue:**

```
If(count > 0) // not empty
    front = (++front) % MAX_QUEUE;
    --count;
If(count == 0) // empty
    front = back = -1
```

HW: Queue implementations using linked List and Arrays.

DE Queue (Double Ended Queue)

Allows add/remove elements from both head/tail.

