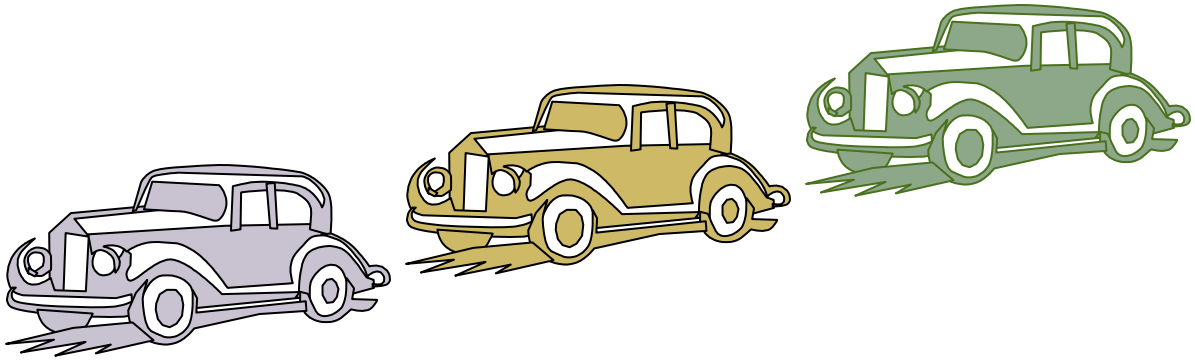
# CS-2001
# DATA STRUCTURE


**Dr. Hashim Yasin**

National University of Computer and Emerging Sciences,

Faisalabad, Pakistan.



# QUEUES

# Queues



A ***Queue*** is a special kind of list, where items are,

- inserted at one end (***the rear***) and
- deleted at the other end (***the front***)

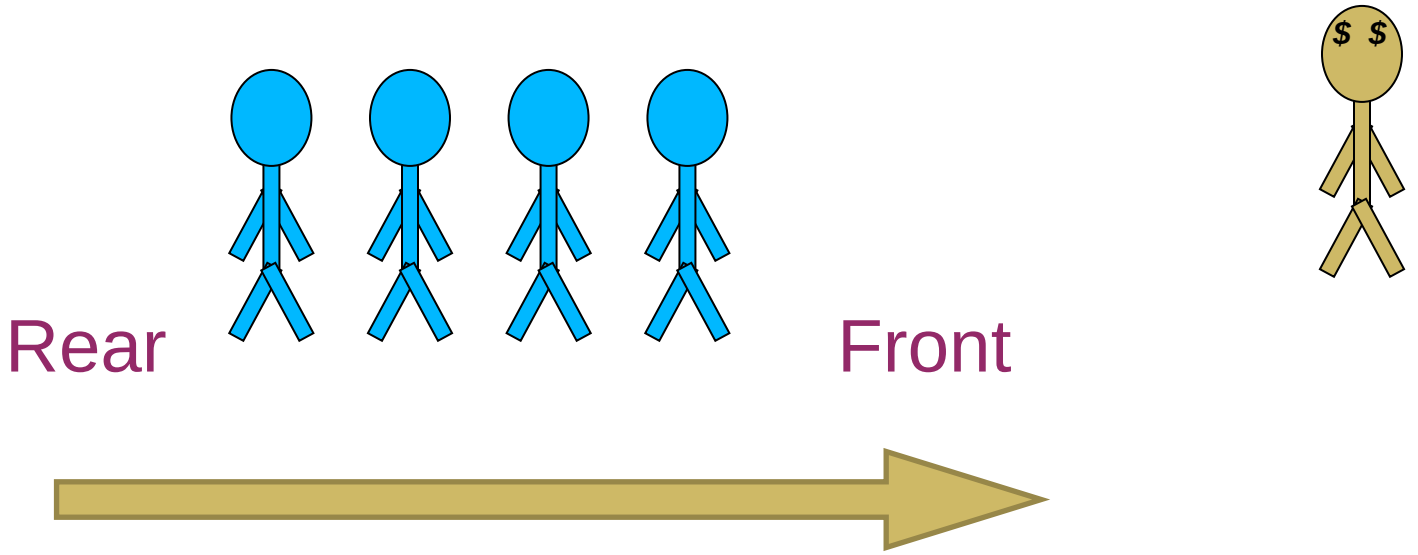
Other Name:

- First In First Out (FIFO)

# Queues



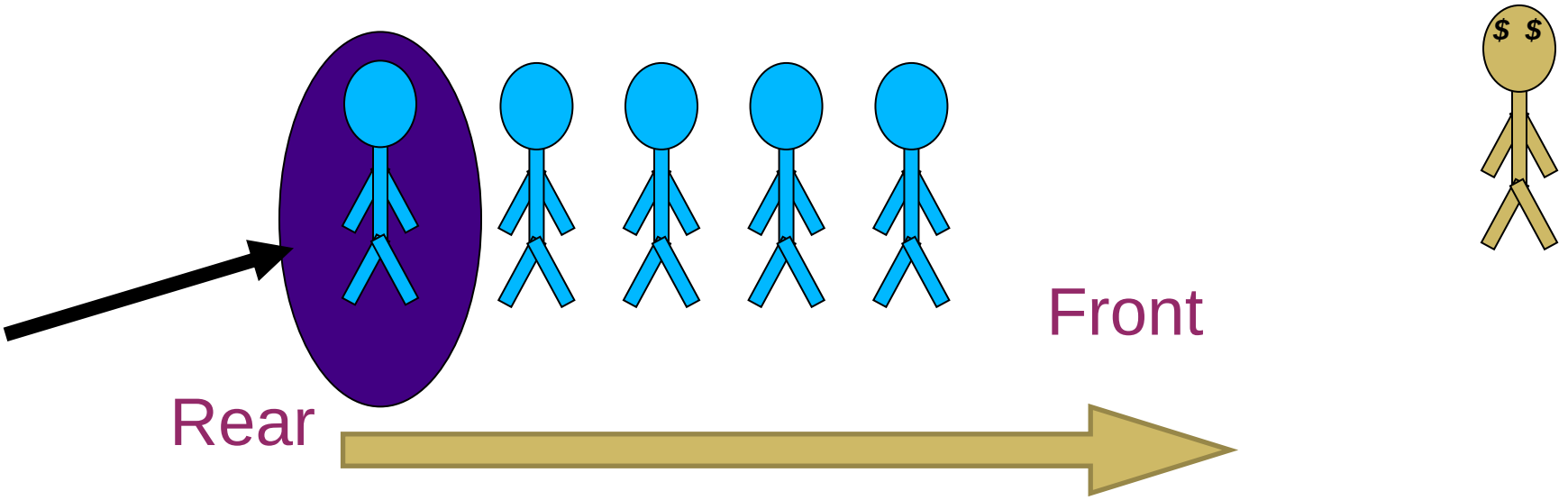
- A queue is like a line of people waiting for a bank teller.
- The queue has a **front** and a **rear**.



# Queues



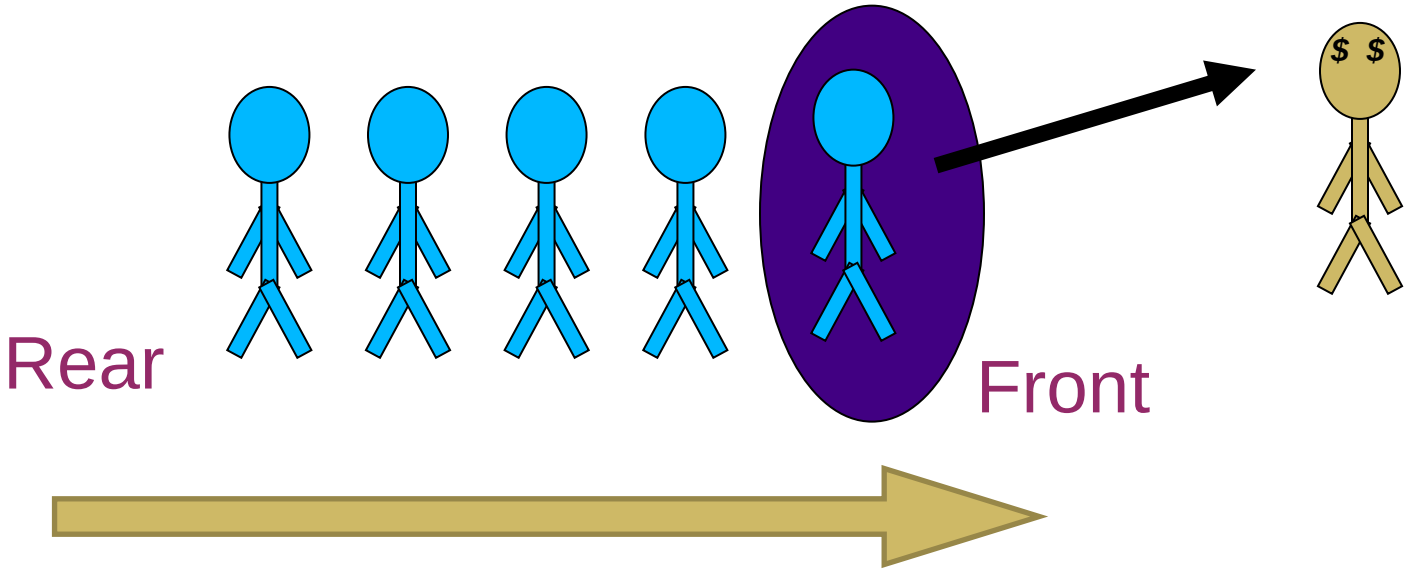
- New people must enter the queue at the **rear.**



# Queues



- When an item is taken from the queue, it always comes from the front.



# Examples



- Billing counter
  - Booking movie tickets
  - Queue for paying bills
- A print queue
- Vehicles on toll-tax bridge
- Luggage checking machine

# Applications



- **Operating system**
  - multi-user/multitasking environments, where several users or tasks may be requesting the same resource simultaneously.
- **Communication Software**
  - queues to hold *information* received over networks and dial up connections.
  - Information can be transmitted faster than it can be processed, so is placed in a queue waiting to be processed.

# Common Operations

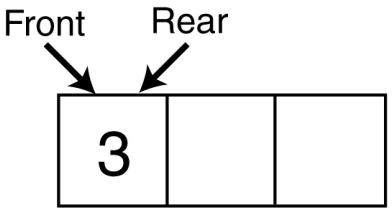


1. **MAKENULL(Q):** Makes Queue Q be an empty list.
2. **FRONT(Q):** Returns the first element on Queue Q.
3. **ENQUEUE(x, Q):** Inserts element x at the end of Queue Q.
4. **DEQUEUE(Q):** Deletes the first element of Q.
5. **EMPTY(Q):** Returns true if and only if Q is an empty queue.

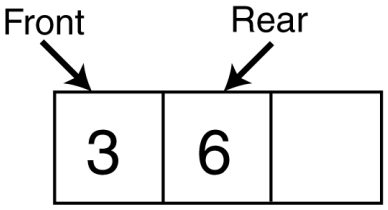
# Enqueue & Dequeue



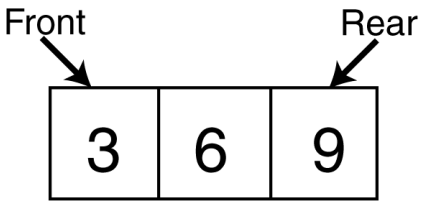
Enqueue(3);



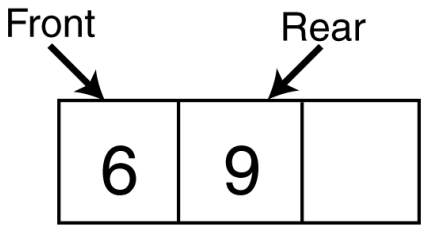

Enqueue(6);



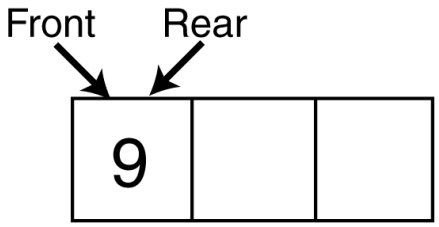

Enqueue(9);



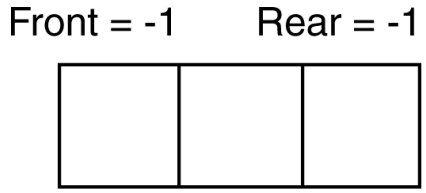

Dequeue();




Dequeue();




Dequeue();



# Enqueue Operation



- **Step 1** – Check if the queue is full.
- **Step 2** – If the queue is full, produce overflow error and exit.
- **Step 3** – If the queue is not full, increment rear pointer to point the next empty space.
- **Step 4** – Add data element to the queue location, where the rear is pointing.
- **Step 5** – Return success.

# Dequeue Operation

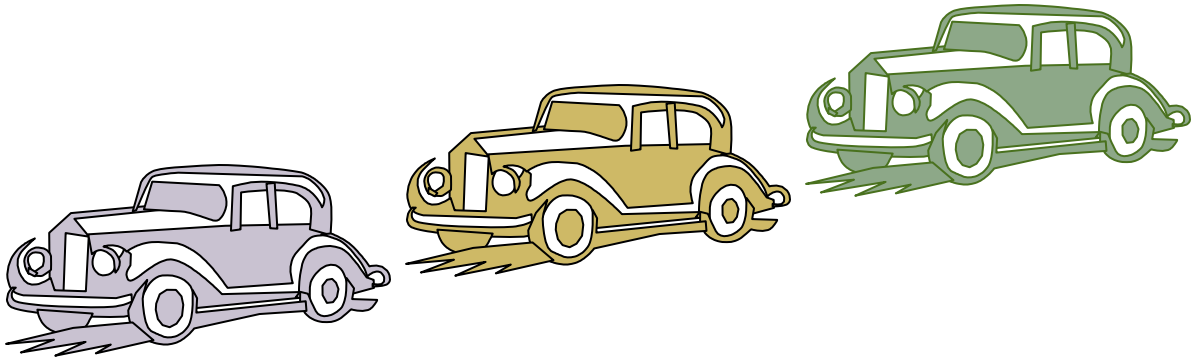


- **Step 1** – Check if the queue is empty.
- **Step 2** – If the queue is empty, produce underflow error and exit.
- **Step 3** – If the queue is not empty, access the data where front is pointing.
- **Step 4** – Increment front pointer to point to the next available data element.
- **Step 5** – Return success.

# Implementation



- Static
  - Queue is implemented by **an array**, and size of queue remains fix
- Dynamic
  - A queue can be implemented as a **linked list** and *expand* or *shrink* with each *enqueue* or *dequeue* operation.



# ARRAY IMPLEMENTATION

# Array Implementation

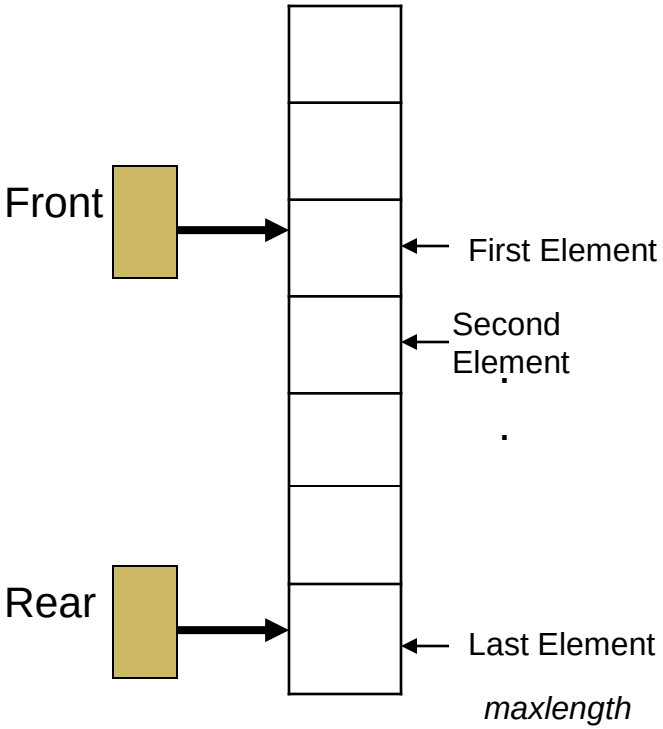


- As with Stacks, signify *zero* index as rear.
- **Enqueue**
  - Shift elements to the right
  - As expensive as with stacks
- **Dequeue**
  - Need to save index of first item inserted
- On Dequeue, decrement index
- On Enqueue, increment index

# Array Implementation



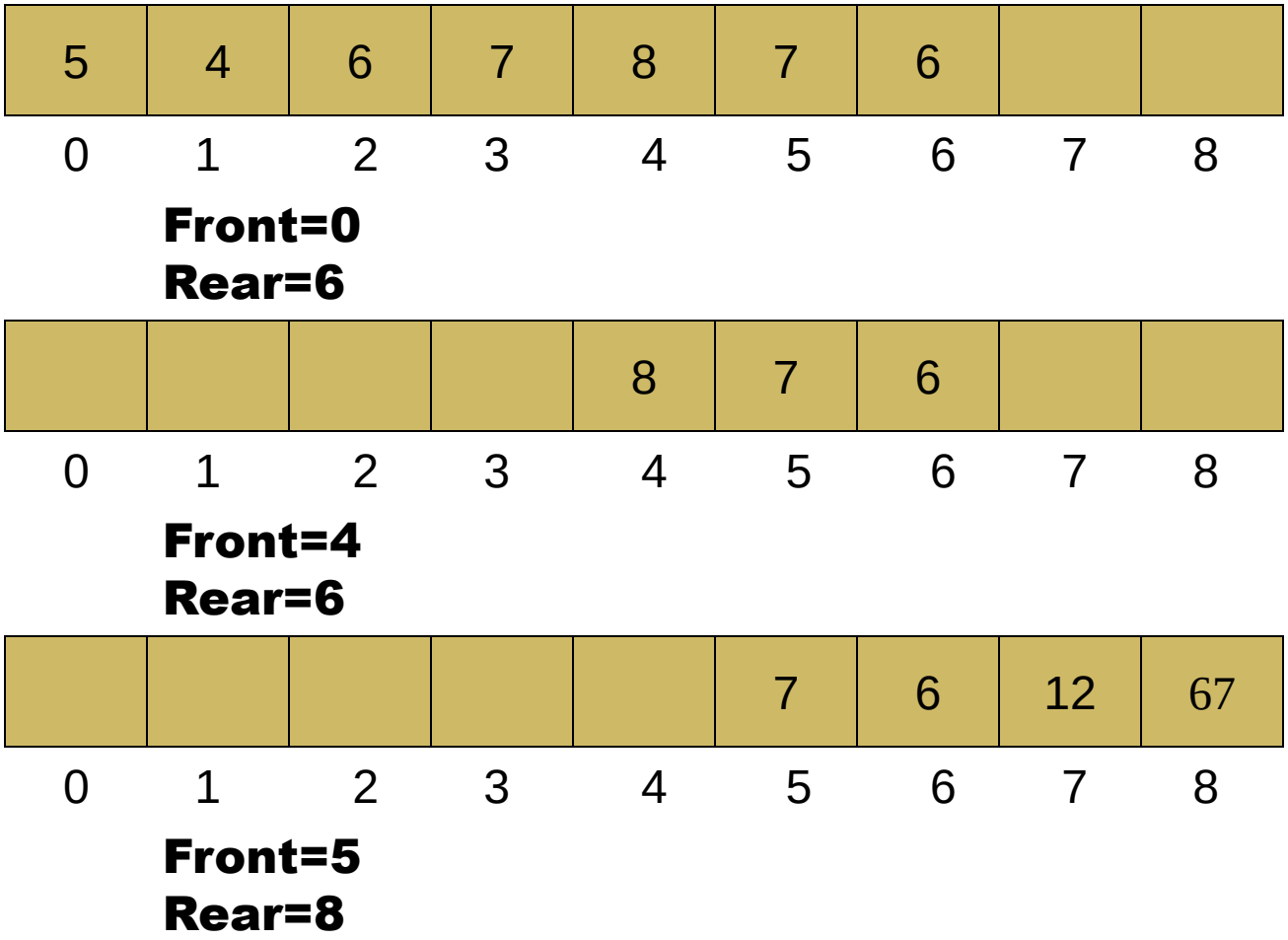
- Use two counters that signify **rear** and **front**




- When **queue is empty,** both front and rear are set to -1
- While **enqueueing** increment rear by 1, and while **dequeuing** increment front by 1
- When there is **only one value in the Queue**, both rear and front have same index

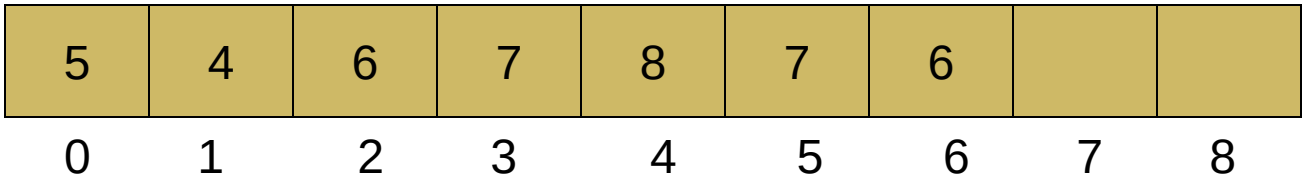
# Array Implementation



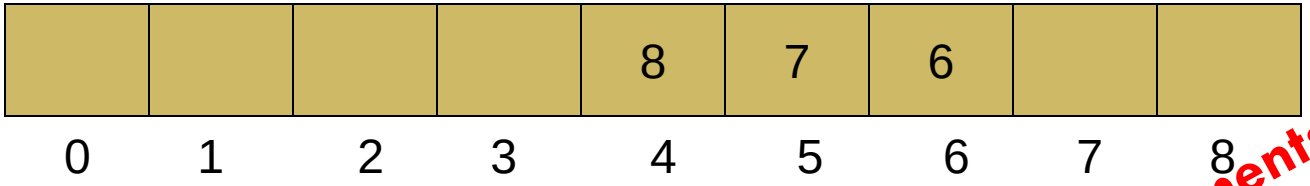


# Array Implementation






**Front=0**
**Rear=6**




**Front=4**
**Rear=6**


**Front=5**
**Rear=8**

**How can we insert more elements? Rear index can not move beyond the last element….**

# Array Implementation



**How can we insert more elements? Rear index can not move beyond the last element….**

**Solution:**

Using Circular Queue

# Circular Queue



- Allow rear to wrap around the array.

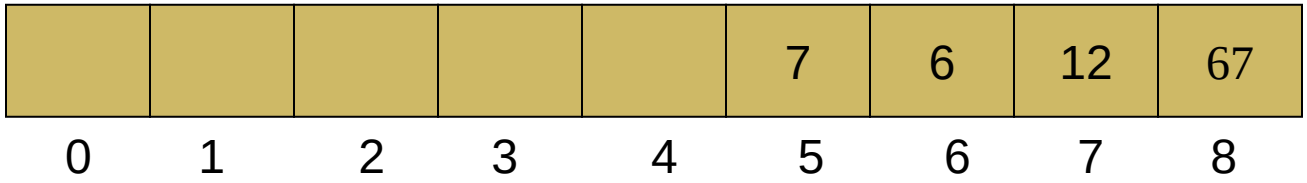
```
if(rear == queueSize-1)
        rear = 0;
else
        rear++;
```

- Or use **modular arithmetic**

```
rear = (rear + 1) % queueSize;
```

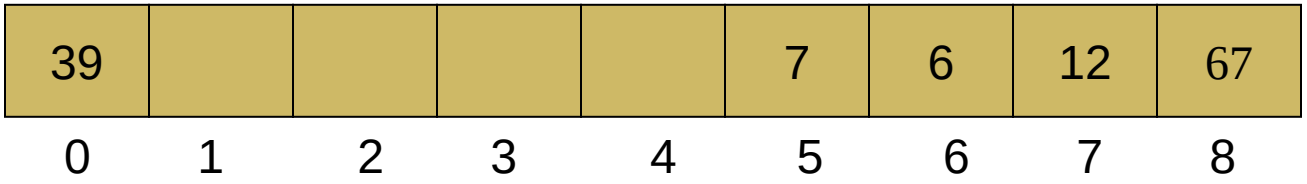
# Circular Queue






**Front=5**
**Rear=8**

**Enqueue 39, Rear = (Rear+1) mod Queue Size = (8+1) mod 9 = 0**




**Front=5**
**Rear=0**

# IMPLEMENTATION

# Implementation … Queue Class



```
class IntQueue {
private:
        int *queueArray;
        int queueSize;
        int front;
        int rear;
        int numItems;
public:
        IntQueue(int);
        ~IntQueue(void);
        void enqueue(int);
        int dequeue(void);
        bool isEmpty(void);
        bool isFull(void);
        void clear(void);
};
```

**Note, the member function clear, which clears the queue by resetting the front and rear indices and setting the numItems to 0.**

# Implementation … Queue Class



```
IntQueue::IntQueue(int s) //constructor
{
    queueArray = new int[s];
    queueSize = s;
    front = -1;
    rear = -1;
    numItems = 0;
}


IntQueue::~IntQueue(void) //destructor
{
     delete [] queueArray;
}
```

# Implementation … Enqueue Function



```
//*****************************************
// Function enqueue inserts the value in num *
// at the rear of the queue.                 *
//*****************************************
void IntQueue::enqueue(int num){
      if (isFull())
            cout << "The queue is full.\n";
      else{
            // Calculate the new rear position
            rear = (rear + 1) % queueSize;
            // Insert new item
            queueArray[rear] = num;
            // Update item count
            numItems++;
      }
}
```

# Implementation … Dequeue Function



```
//*****************************************
// Function dequeue removes the value at the   *
// front of the queue, and copies it into num. *
//*****************************************
int IntQueue::dequeue(void){
      if (isEmpty())
            cout << "The queue is empty.\n";
      else{
            // Move front
            front = (front + 1) % queueSize;
            // Retrieve the front item
            int num = queueArray[front];
            // Update item count
            numItems--;
      }
      return num;
}
```

# Implementation … isEmpty Function



```
//*****************************************
// Function isEmpty returns true if the queue *
// is empty, and false otherwise.              *
//*****************************************

bool IntQueue::isEmpty(void){
      if (numItems)
            return false;
      else
            return true;
}
```

# Implementation … isFull Function



```
//*****************************************
// Function isFull returns true if the queue *
// is full, and false otherwise.             *
//*****************************************

bool IntQueue::isFull(void){
      if (numItems < queueSize)
            return false;
      else
            return true;
}
```

# Implementation … Clear Function



```
//*****************************************
// Function clear resets the front and rear *
// indices and sets numItems to 0.          *
//*****************************************

void IntQueue::clear(void){
      front = - 1;
      rear = - 1;
      numItems = 0;
}
```

# Implementation … Demonstration



```
//Program demonstrating the IntQueue class
void main(void){
        IntQueue iQueue(5);
        cout << "Enqueuing 5 items...\n";
        // Enqueue 5 items.
        for (int x = 0; x < 5; x++)
                iQueue.enqueue(x);
        // Attempt to enqueue a 6th item.
        cout << "Now attempting to enqueue again...\n";
        iQueue.enqueue(5);

        // Deqeue and retrieve all items in the queue
        cout << "The values in the queue were:\n";
        while (!iQueue.isEmpty()){
                int value;
                value = iQueue.dequeue();
                cout << value << endl;
        }
}
```

# Implementation … Output



**Program Output:**

```
Enqueuing 5 items...
Now attempting to enqueue again...
The queue is full.
The values in the queue were:
0
1
2
3
4
```

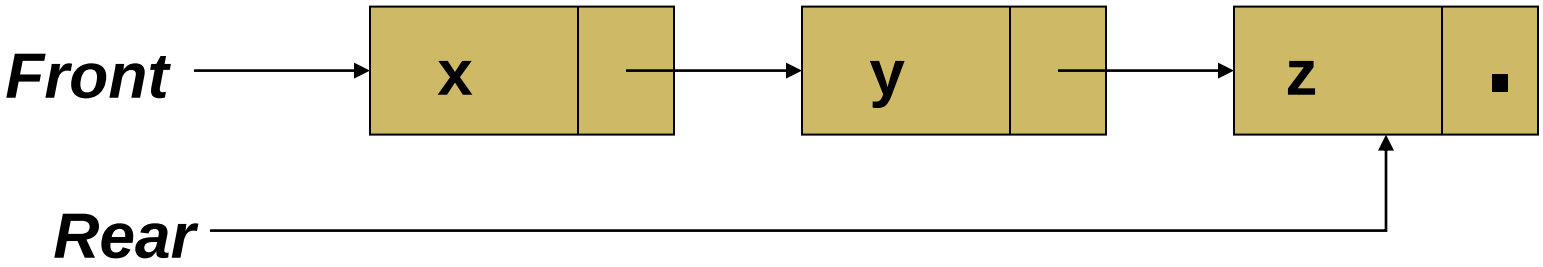
# A POINTER-BASED IMPLEMENTATION

# A Pointer based Implementation



Keep two pointers:

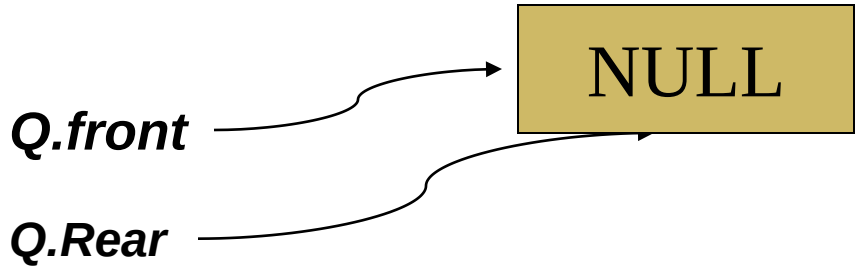
- **FRONT:** A pointer to the first element of the queue.
- **REAR:** A pointer to the last element of the queue.



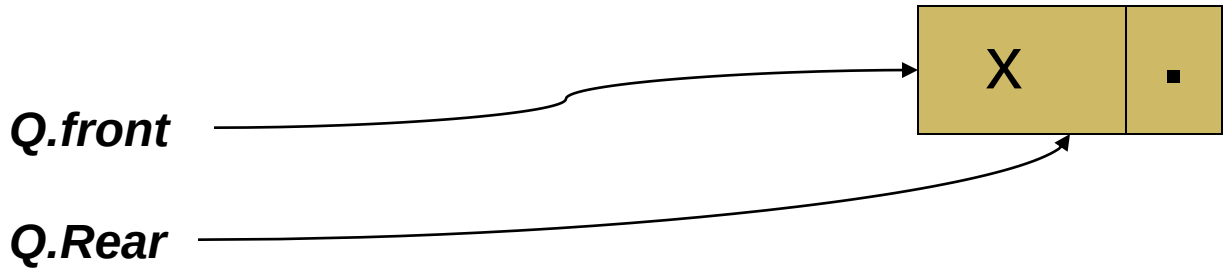
# A Pointer based Implementation



MAKENULL(Q)



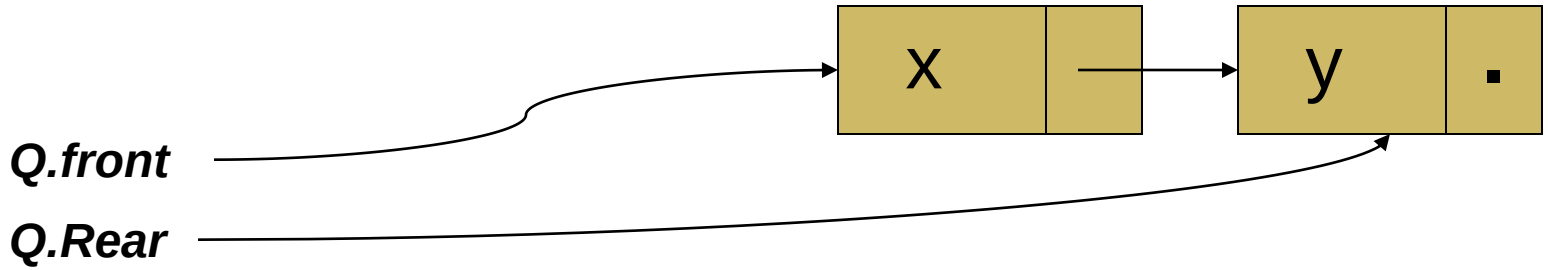

ENQUEUE(x,Q)



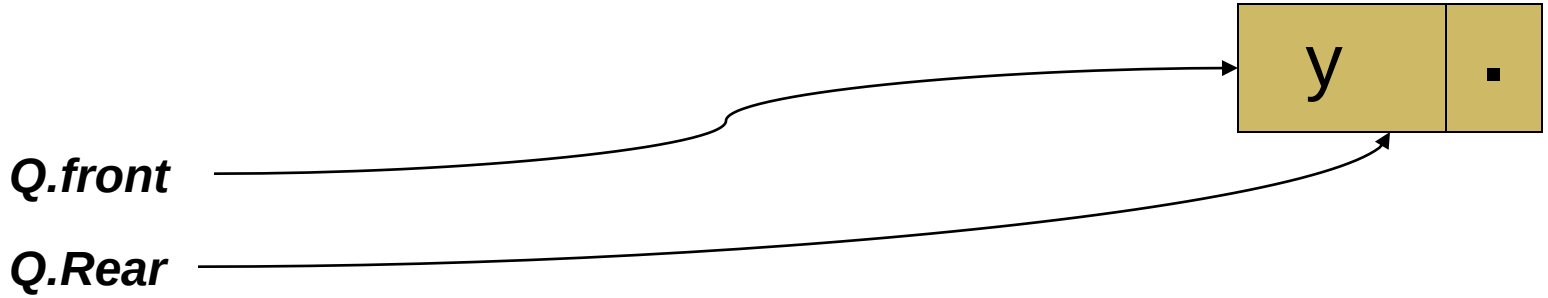
# A Pointer based Implementation



ENQUEUE(y,Q)



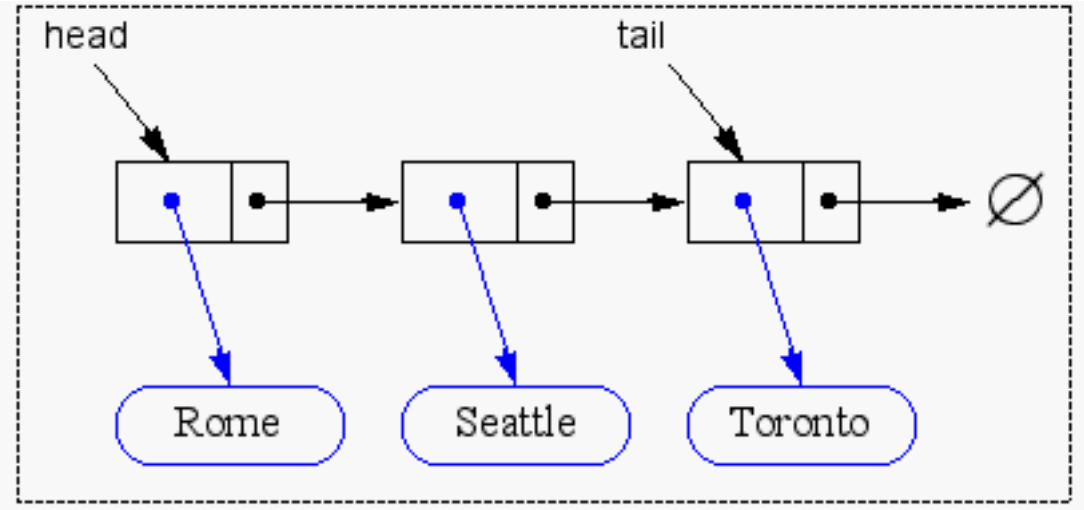

DEQUEUE(Q)



# Singly Linked List based Implementation



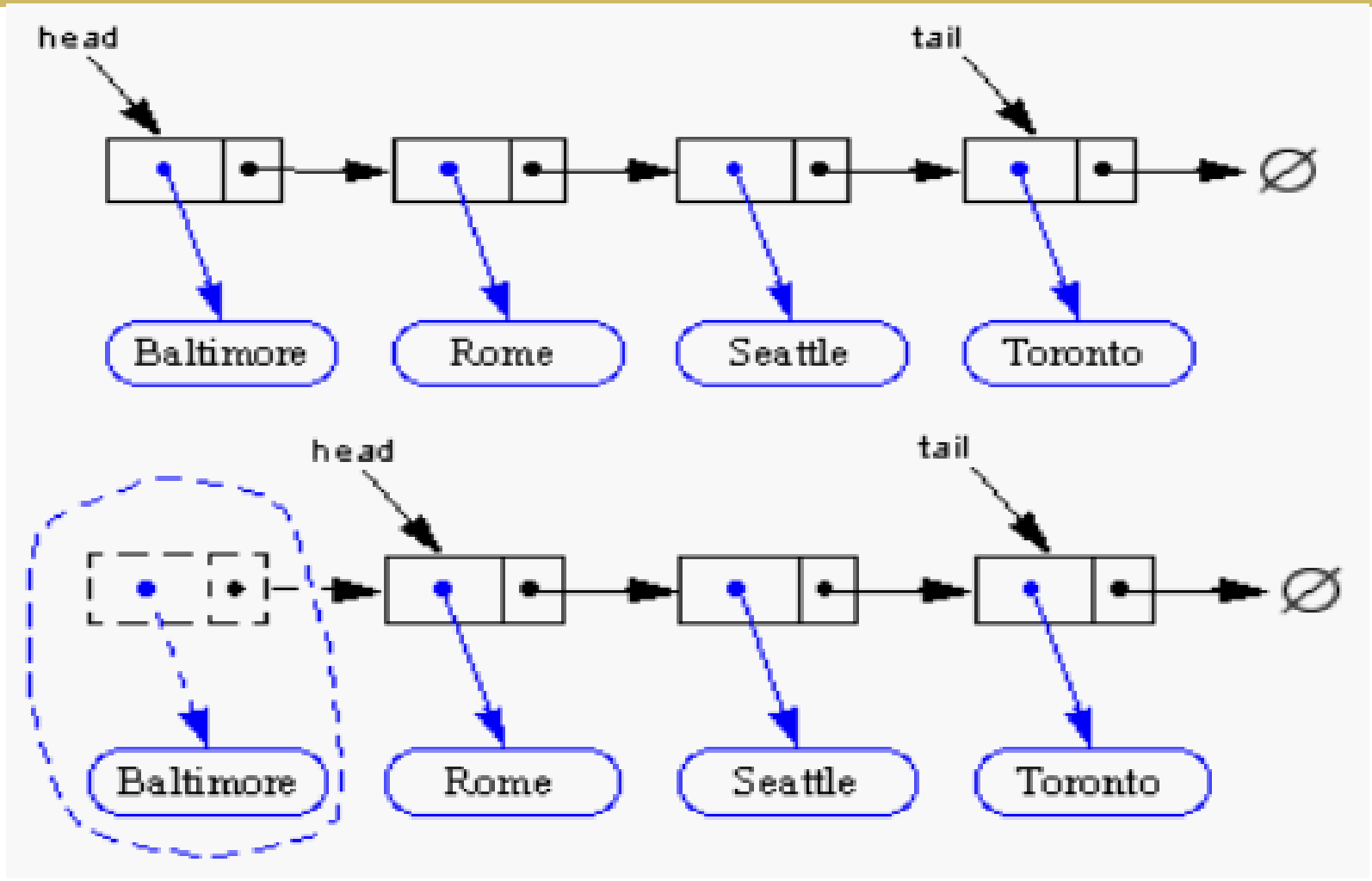
Nodes connected in a chain by links




The *head of the list is the front of the queue*, the *tail of the list is the rear of the queue*.

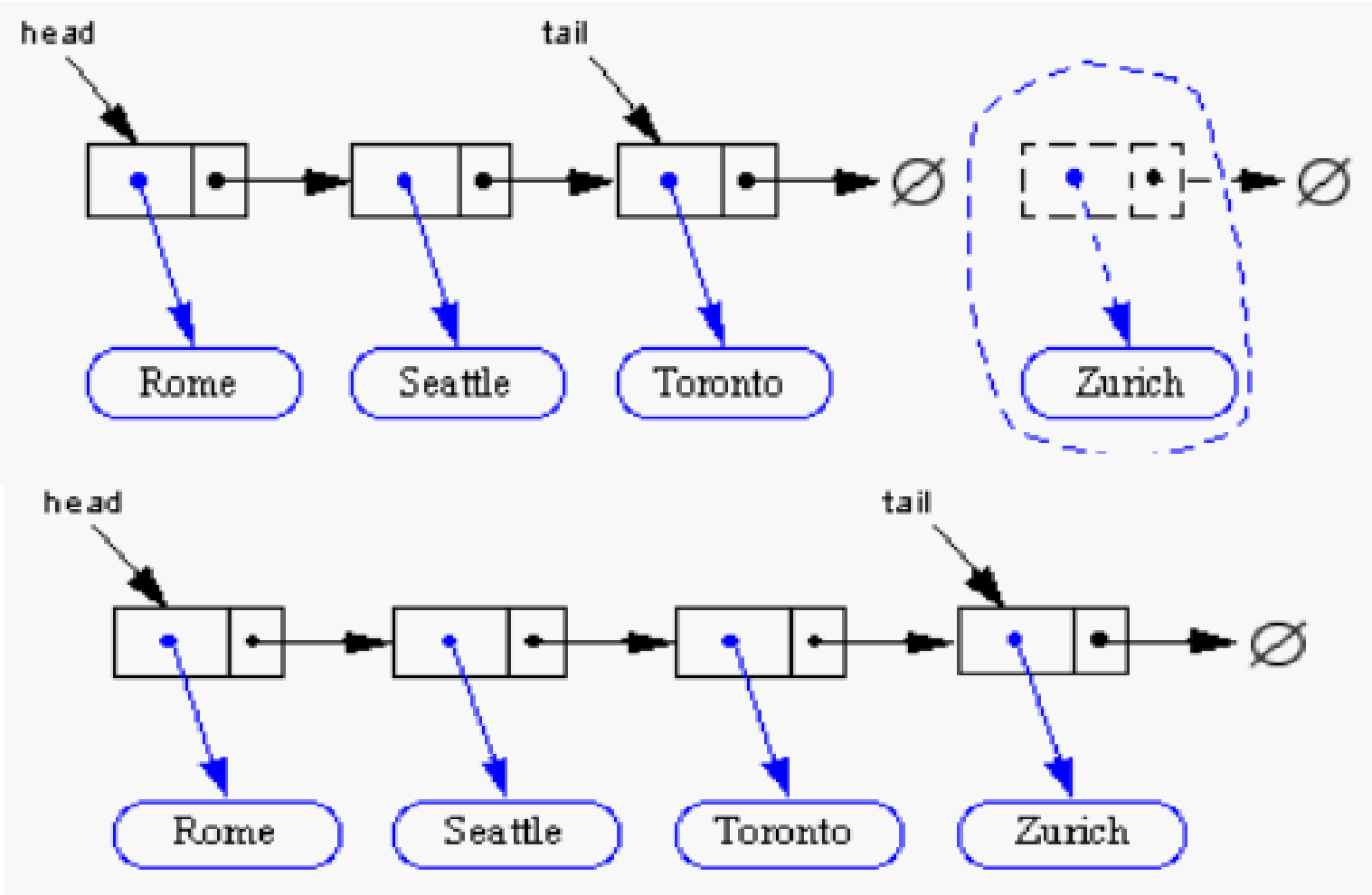
# Removing at the Head





# Inserting at the Tail





# Homework



- Implement the same program given in previous slides with the help of,
  - Singly Linked List
  - Doubly Linked List

# Reading Materials



- Chapter 8, Data Structures by Larry Nyhoff