National University of Computer & Emerging Sciences

Queues

Queues

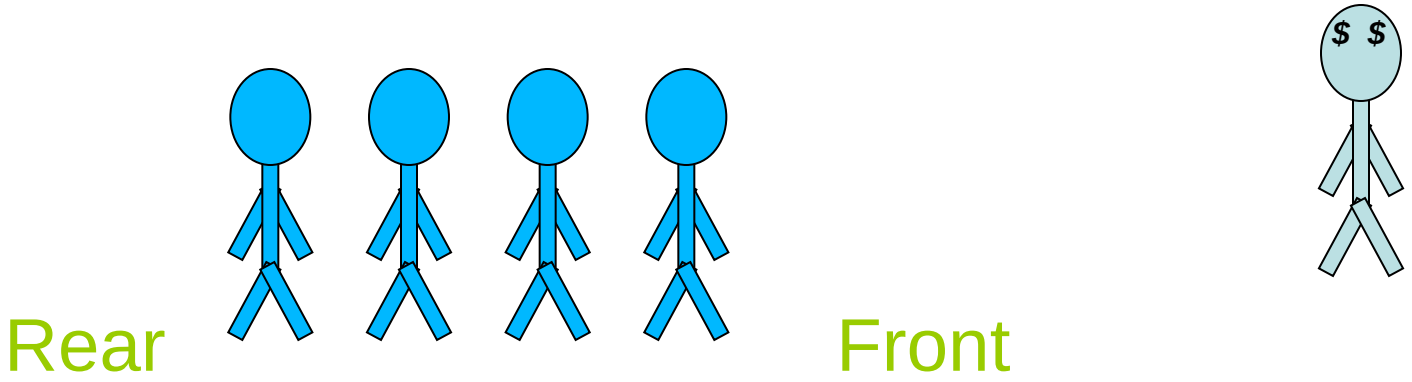
"A ***Queue*** is a special kind of list, where items are inserted at one end (***the rear***) And deleted at the other end (***the front***)"

Other Name:

- First In First Out (FIFO)

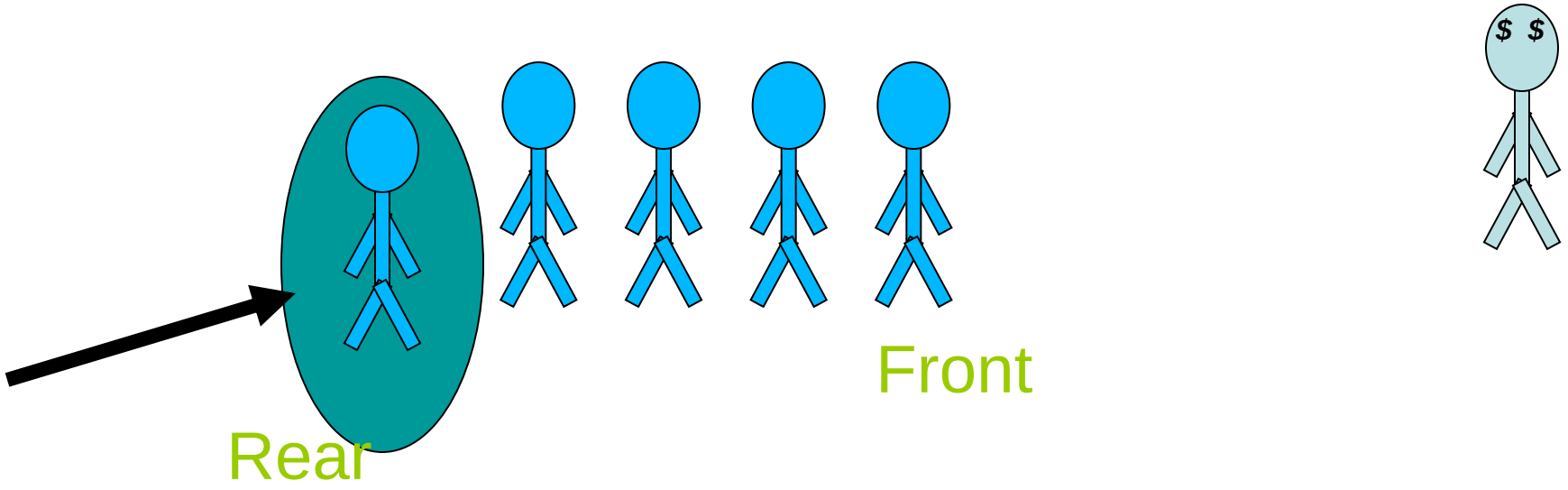
Queues

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



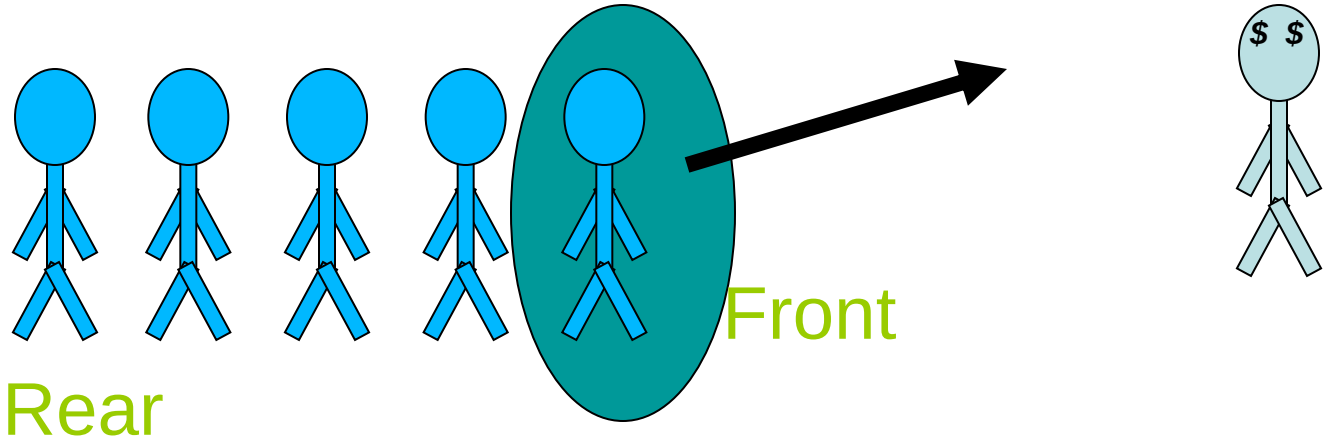
Queues

- New people must enter the queue at the rear.



Queues

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



Some examples

- Billing counter
  - Booking movie tickets
  - Queue for paying bills

- A print queue
- Vehicles on toll-tax bridge
- Luggage checking machine

- Some others?

Applications of Queues

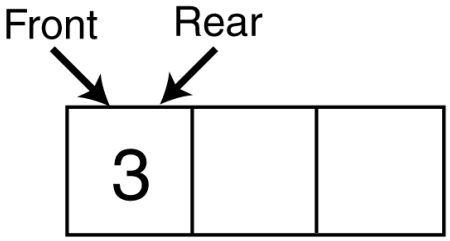
- Operating system
  - multi-user/multitasking environments, where several users or task may be requesting the same resource simultaneously.

- 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)

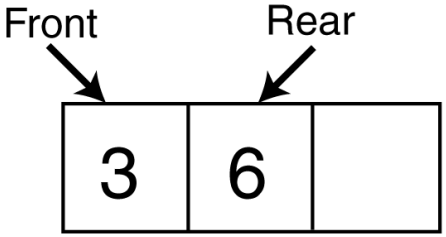
Common Operations (Queue ADT)

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.

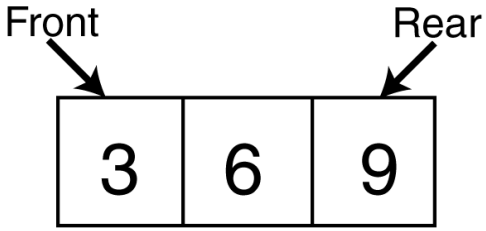
Enqueue(3);



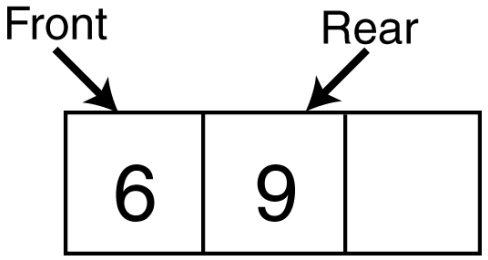
Enqueue(6);



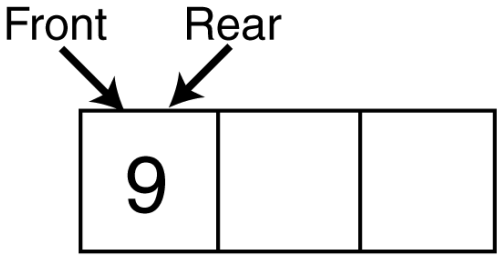
Enqueue(9);



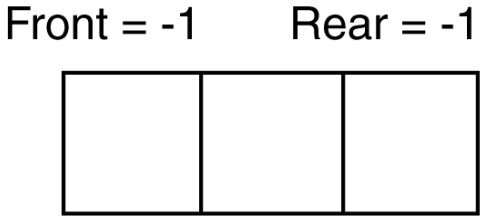
Dequeue();



Dequeue();



Dequeue();



Implementation

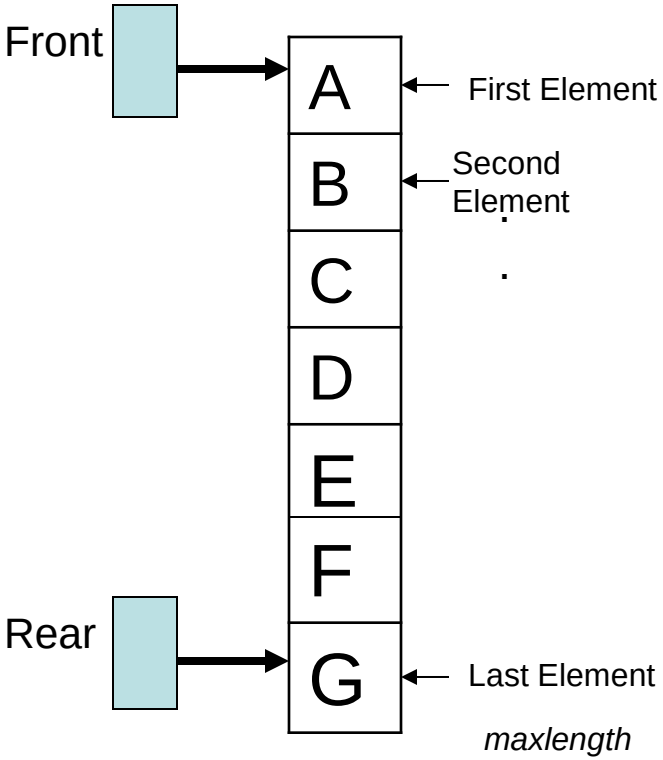
- Static
  - Queue is implemented by an array, and size of queue remains fix
- Dynamic
  - A **queue** can be **implemented** as a **linked list**, and *expand* or *shrink* with each *enqueue* or *dequeue* operation.

Array Implementation

- Signify *zero* index as front.
- Dequeue
  - Shift elements to the left
  - Expensive!
- Enqueue
  - Need to save index of last item inserted
    - On Enqueue, increment index
    - On Dequeue, decrement index

Alternative Array Implementation

- Use two counters that signify rear and front

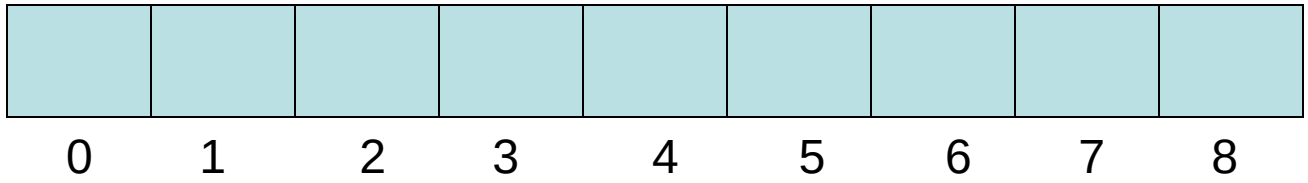


When queue is empty both front and rear are set to -1

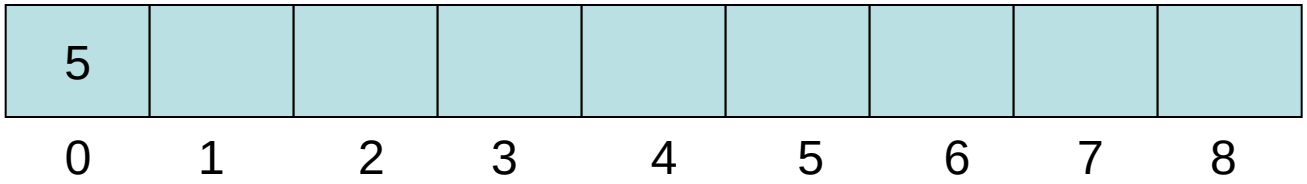
While enqueueing increment rear by 1, and while dequeueing increment front by 1

When there is only one value in the Queue, both rear and front have same index

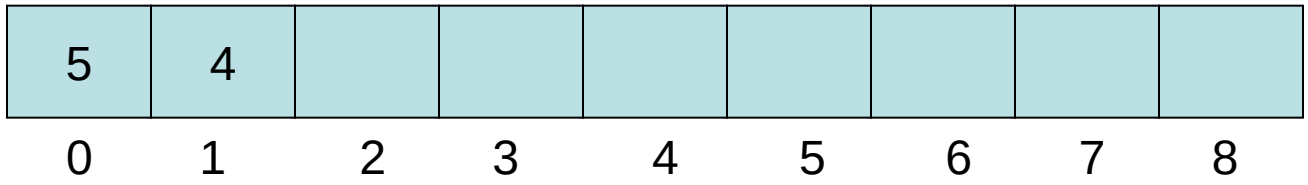
Array Implementation



Front= -1
Rear = -1

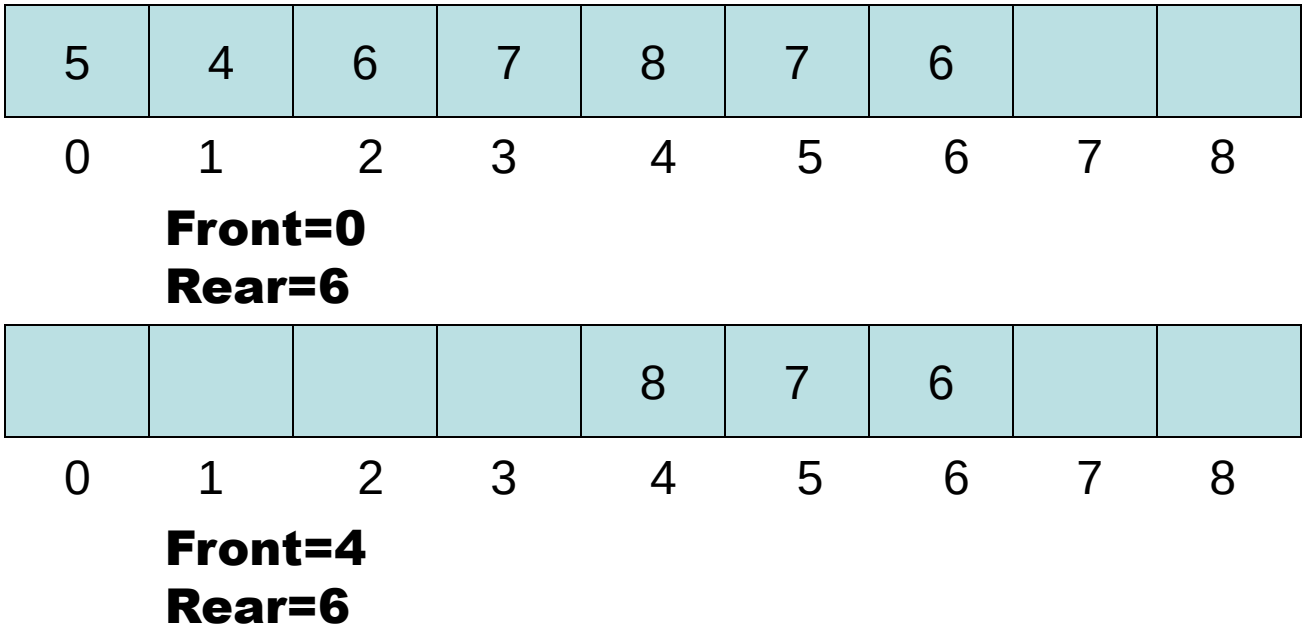


Front= 0
Rear = 0

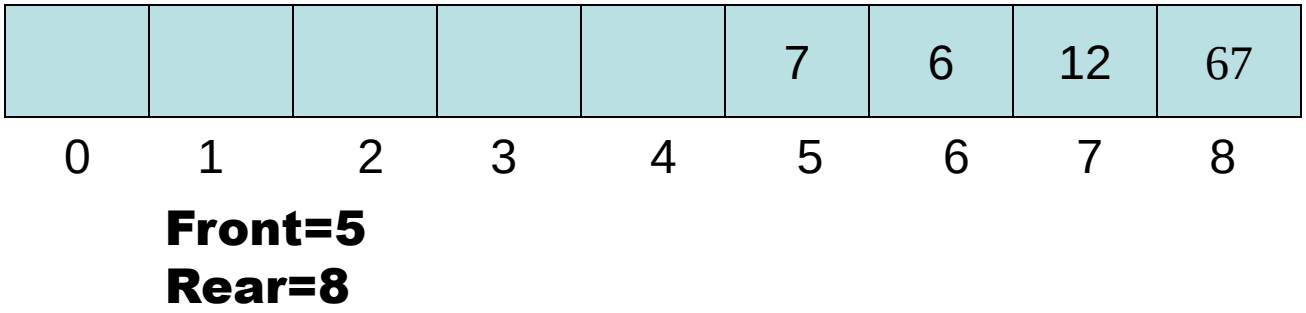


Front= 0
Rear = 1

Array Implementation



Array Implementation



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

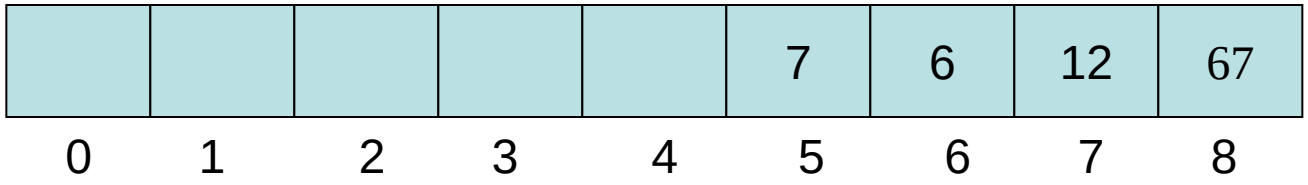
Solution: Using circular queue

- Allow rear to wrap around the array.

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

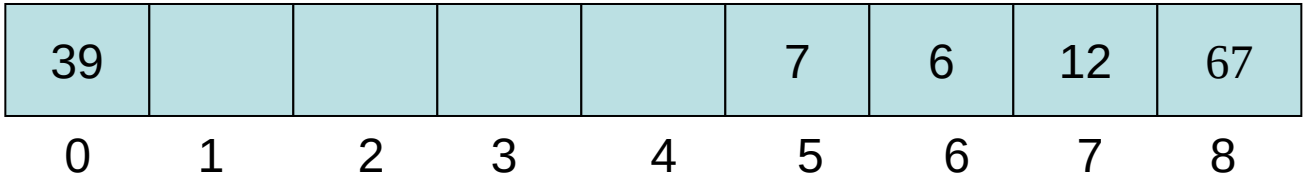
- Or use module arithmetic

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



Front=5
Rear=8

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



Front=5
Rear=0

How to determine empty and full Queues?

- It can be somewhat tricky
- Number of approaches
  - A counter indicating number of values in the queue can be used (we will use this approach)
  - Later, we will see another approach

Implementation

```
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.

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


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

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

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

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

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

```
//*****************************************
// 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++;
      }
}
```

```
//****************************************
// Function dequeue removes the value at the  *
// front of the queue, and copies it into num.*
//****************************************

bool IntQueue::dequeue(int &num)
{
        if (isEmpty())
        {
                cout << "The queue is empty.\n";
                return false;
        }

        // Move front
        front = (front + 1) % queueSize;
        // Retrieve the front item
        num = queueArray[front];
        // Update item count
        numItems--;
        return true;
}
```

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

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

//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;
                iQueue.dequeue(value);
                cout << value << endl;
        }
}
```

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
```