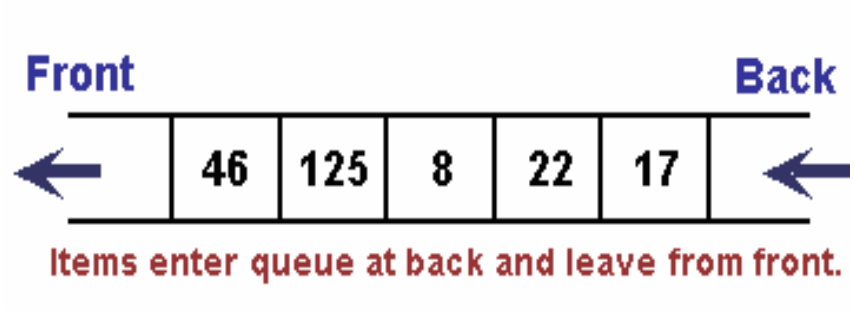
# Queues

- What is a queue?
- Queue Implementations:
  - As Array
  - As Circular Array
  - As Linked List
- Applications of Queues.
- Priority queues

# What is a queue?

- Queues are linear data structures in which we add elements to one end and remove them from the other end.




- The first item to be en-queued is the first to be de-queued. Queue is therefore called a First In First Out (FIFO) structure.

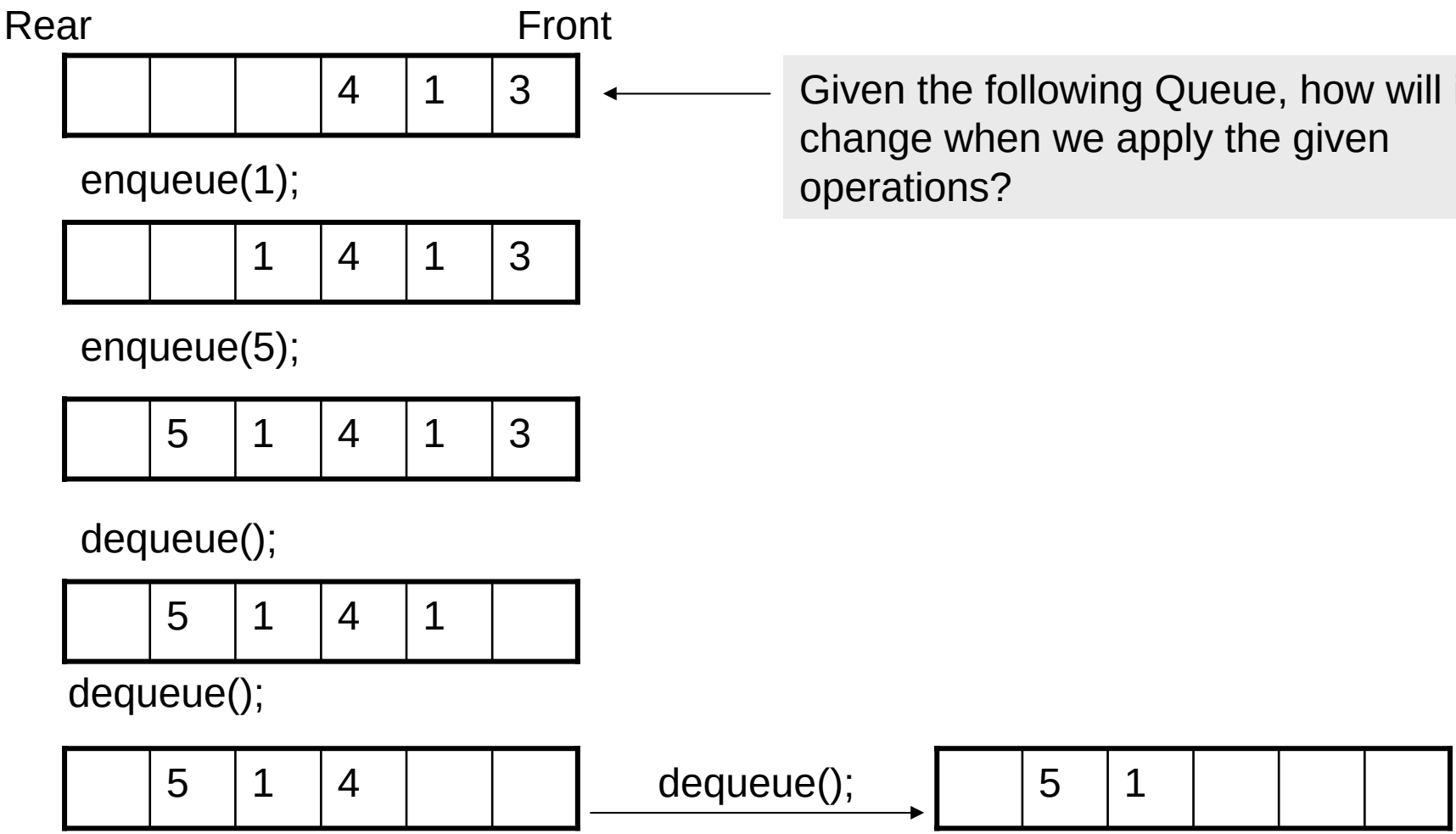
- Queue operations:

  Enqueue

  Dequeue

  GetHead

# What is a queue?



# Queue Implementation

- In our implementation, a queue is a **container** that extends the **AbstractContainer** class and implements the **Queue** interface.

```
public interface Queue extends Container{
    public abstract Object getHead();
    public abstract void enqueue(Object obj);
    public abstract Object dequeue();
}
```

- We provide three implementations for Queue
  - QueueAsArray
  - QueueAsCircularArray
  - QueueAsLinkedList

# QueueAsArray

```
public class QueueAsArray extends AbstractContainer
      implements Queue {

    protected Object[] array;
    protected int rear = 0;
    protected int size;

    public QueueAsArray(int size) {
        array = new Object[size];
        this.size = size;
    }

    public void purge(){
        int index = 0;
        while(count > 0){
            array[index] = null;
            index++;
            count--;
        }
        rear = 0;
   }
```

**Complexity is O(n)**

# QueueAsArray (Cont.)

```
public Object getHead(){
    if(count == 0)
        throw new ContainerEmptyException();
    else
        return array[0];
}

public void enqueue(Object obj){
    if(count == size){
       throw new ContainerFullException();
    }
    else{
        array[rear++] = obj;
        count++;
    }
}
```

Complexity is O(1)

Complexity is O(1)

# QueueAsArray (Cont.)

```
public Object dequeue(){
    if(count == 0)
        throw new ContainerEmptyException();
    else
    {
        Object obj = array[0];
        count--;
        for(int k = 1; k <= count; k++)
          array[k - 1] = array[k];
        rear--;
        return obj;
    }
}
```

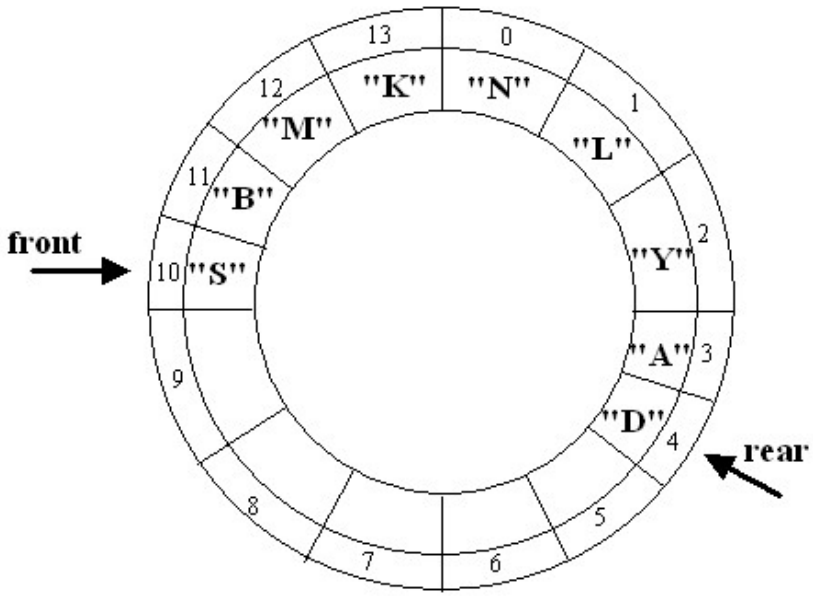
**Complexity is O(n)**

# QueueAsArray (Cont.)

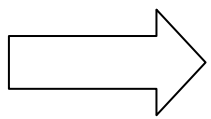
```
public Iterator iterator() {
   return new Iterator() {

      int index = 0;

      public boolean hasNext(){
         return index < count;
      }

      public Object next(){
         if(index == count)
            throw new NoSuchElementException();
         else {
            Object obj = array[index++];
            return obj;
         }
      }
   };
}
```

# QueueAsCircularArray Implementation

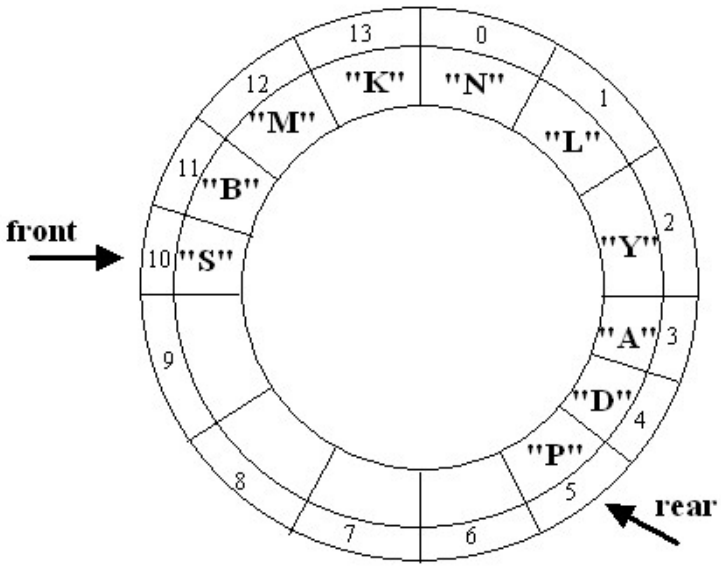
- By using modulo arithmetic for computing array indexes, we can have a queue implementation in which each of the operations enqueue, dequeue, and getHead has complexity O(1)



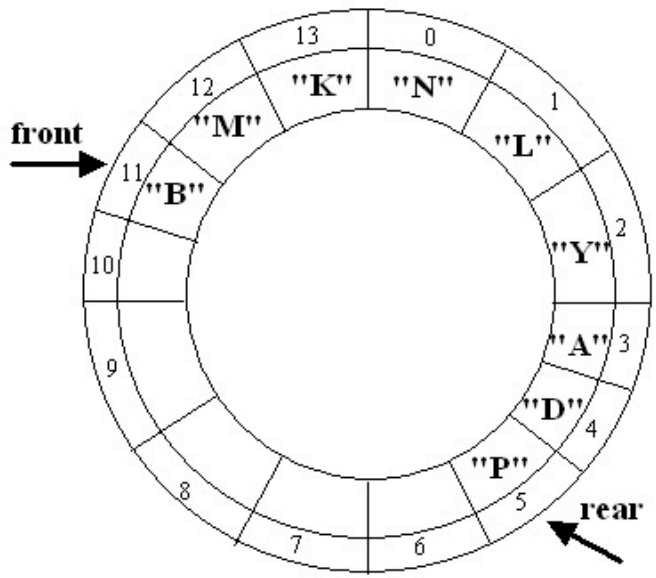

**Enqueue(“P”) will result in …**



# QueueAsCircularArray Implementation (Cont.)




**Dequeue() will result in**



# QueueAsCircularArray

```
public class QueueAsCircularArray extends AbstractContainer
      implements Queue {
   protected Object[] array;
   protected int front = 0;
   protected int rear = 0;
   protected int size;

    public QueueAsCircularArray(int size) {
        array = new Object[size]; this.size = size;
    }

    public void purge(){
        int index = front;
        while(count > 0){
             array[index] = null;
             index = (index + 1) % size;
             count--;
        }
        front = rear = 0;
   }
```

**Complexity is O(n)**

# QueueAsCircularArray

```
public Object getHead(){
    if(count == 0) throw new ContainerEmptyException();
    else return array[front];
}

public void enqueue(Object obj){
    if(count == size) throw new ContainerFullException();
    else {
           array[rear] = obj;
           rear = (rear + 1) % size;
           count++;
    }
}

public Object dequeue(){
    if(count == 0)throw new ContainerEmptyException();
    else {
        Object obj = array[front];
        front = (front + 1) % size;
        count--;
        return obj;
    }
}
```

Complexity is O(1)

Complexity is O(1)

Complexity is O(1)

# QueueAsCircularArray

```
public Iterator iterator(){
    return new Iterator() {
        int index = front;
        int counter = 0;
        public boolean hasNext(){
            return counter < count;
        }

        public Object next(){
            if(counter == count)
                throw new NoSuchElementException();
            else {
                Object obj = array[index];
                index = (index + 1) % size;
                counter++;
                return obj;
            }
        }
    };
}
```

# QueueAsLinkedList

```
public class QueueAsLinkedList extends AbstractContainer
      implements Queue {

    protected MyLinkedList list;

    public QueueAsLinkedList(){list = new MyLinkedList();}

    public void purge(){
        list.purge();
        count = 0;
    }

    public Object getHead(){
        if(count == 0)
            throw new ContainerEmptyException();
        else
            return list.getFirst();
    }
```

Complexity is O(1)

Complexity is O(1)

# QueueAsLinkedList

```
public void enqueue(Object obj){
    list.append(obj);
    count++;
}

public Object dequeue(){
    if(count == 0)
        throw new ContainerEmptyException();
    else {
        Object obj = list.getFirst();
        list.extractFirst();
        count--;
        return obj;
    }
}
```

Complexity is O(1)

Complexity is O(1)

# QueueAsLinkedList

```
public Iterator iterator() {
    return new Iterator() {

        MyLinkedList.Element position = list.getHead();

        public boolean hasNext(){
            return position != null;
        }

        public Object next(){
            if(position == null)
                throw new NoSuchElementException();
            else{
                Object obj = position.getData();
                position = position.getNext();
                return obj;
            }
        }
    };
}
```

# Application of Queues

- Direct applications
  - Waiting lines: Queues are commonly used in systems where waiting line has to be maintained for obtaining access to a resource. For example, an operating system may keep a queue of processes that are waiting to run on the CPU.
  - Access to shared resources (e.g., printer)
  - Multiprogramming

- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

# Priority Queues

- In a normal queue the enqueue operation add an item at the back of the queue, and the dequeue operation removes an item at the front of the queue.

- A priority queue is a queue in which the dequeue operation removes an item at the front of the queue; but the enqueue operation insert items according to their priorities.

- A higher priority item is always enqueued before a lower priority element.

- An element that has the same priority as one or more elements in the queue is enqueued after all the elements with that priority.