Data Structures and Analysis of Algorithms
CST 225-3

Queue

Have you been in a queue?

Queues

- Stores a set of elements in a particular order
- Queue principle: **FIRST IN FIRST OUT(FIFO)**
- It means the first element inserted is the first one to be removed
- Restrict access to the least recently inserted item
- Ex: A queue in a bank

  The first one in line is the first one to be served

Queues

- A queue is used in computing in much the same way as it is used in every day life: allow a sequence of items to be processed on a first-come-first-served basis.
- A queue is a data structure that is similar to a stack except that in a queue, the first item inserted is the first to be removed (FIFO), while in a stack, the last item inserted is the first to be removed (LIFO).

Queue Applications

- Queue is used when things don’t have to be processed immediately, but have to be processed in **F**irst **I**n **F**irst **O**ut order.

- Real life examples
  - People waiting in line at a bank
  - Air planes waiting to take off

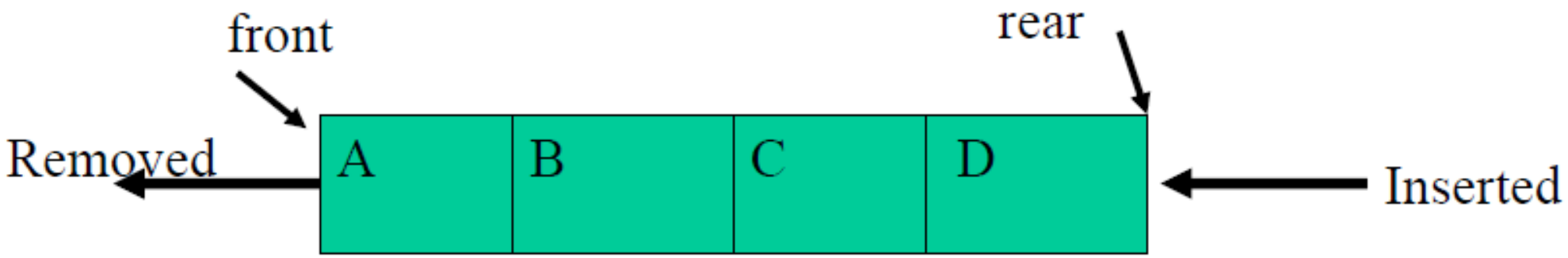
- Applications related to Computer Science
  - Data packets waiting to be transmitted over the Internet
  - Job scheduling (e.g. Round-Robin algorithm for CPU allocation)
  - Call center phone systems

Queue Applications

- When a resource is shared among multiple consumers.
- In most computer installations, for example, one printer is connected to several machines so that more than one user can submit printing jobs to the same printer, it maintains a queue.

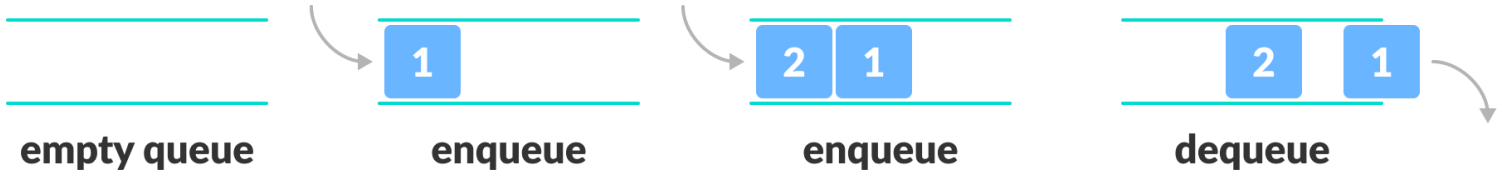
Queue Definition

- A queue is an ordered collection of items from which items may be deleted at one end (called front of the queue) and into which items may be inserted at the other end (called the rear of the queue).



Basic Queue Operations

- enqueue(e): Adds element e to the back of queue.
- dequeue( ): Removes and returns the first element from the queue (or null if the queue is empty).
- front( )/peek(): Returns the first element of the queue, without removing it (or null if the queue is empty).
- isFul( ): Returns a boolean indicating whether the queue is full.
- isEmpty( ): Returns a boolean indicating whether the queue is empty.



Series of Queue Operations : Example

Method	Return Value	first $\leftarrow Q \leftarrow$ last
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
size( )	2	(5, 3)
dequeue( )	5	(3)
isEmpty( )	false	(3)
dequeue( )	3	( )
isEmpty( )	true	( )
dequeue( )	null	( )
enqueue(7)	–	(7)
enqueue(9)	–	(7, 9)
first( )	7	(7, 9)
enqueue(4)	–	(7, 9, 4)

Implementation of Queues

- Can be done in two ways:

1. **Array based implementation** - today discussion
   - Limited in size
   - Fast

2. **Linked List based implementation** - later discussion
   - Not limited in size
   - Overhead to allocate link, unlink and deallocate

Queue Implementation with Arrays

- Three variables:

- **Front** = -1 (returns array index of the first element or front of the queue)
- **Rear** = -1 (returns array index of the last element or back of the queue)
- **Size** = 0 (Current number of elements in the array)

Queue Implementation with Arrays

- Example: To be discussed in the class

Variable Declaration

```
class Queue
{
    private int default_capacity = 10;
    int front, rear, currentSize;
    int array[];


 }
```

Constructor of Queue class

```
public Queue(int sizeOfQueue)
        {
                this. default_capacity = sizeOfQueue; currentSize = 0;
                front = -1;
                rear = -1;
                array = new int[this.default_capacity];

        }
```

Check Queue is empty : isEmpty()

```
boolean isEmpty()
{
        return (front == -1 && rear == -1);
       //or
        //return (currentSize == 0);

 }
```

Check queue is Full : isFull()

- Queue is full when size becomes equal to the capacity

```
boolean isFull()
{
        return (currentSize == default_capacity);
        //or
        // return (rear == default_capacity-1)
}
```

Add Elements: Enqueue()

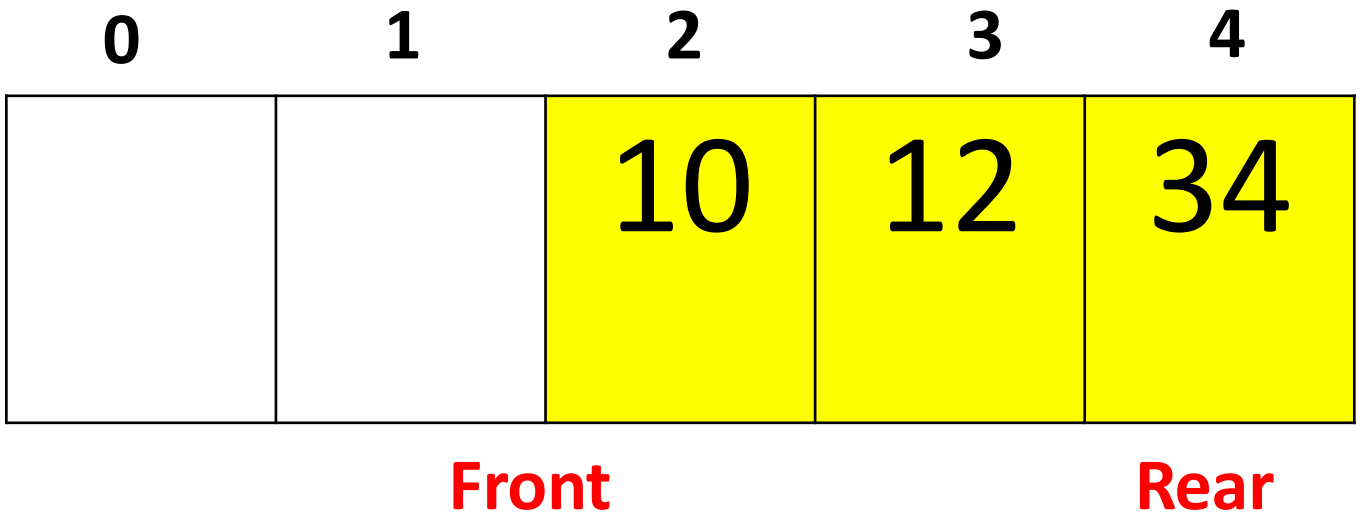
```
public void enqueue(int item) { if (isFull()) {
                System.out.println ("Queue is full!! Cannot add more elements");
        }
        else {
                rear++;
                array [rear] = item;
                System.out.println (item+ " Added to queue"); currentSize++;
                }
}
```

Remove Items from Queue – Dequeue()

```
void dequeue()
   {
         if (isEmpty ())
        System.out.println("Queue is empty!! Can not remove element");
        else if (rear == front) { rear = front = -1; } else {
                front++;
                System.out.println (item+ " Removed from queue"); currentSize--;
                }
   }
```

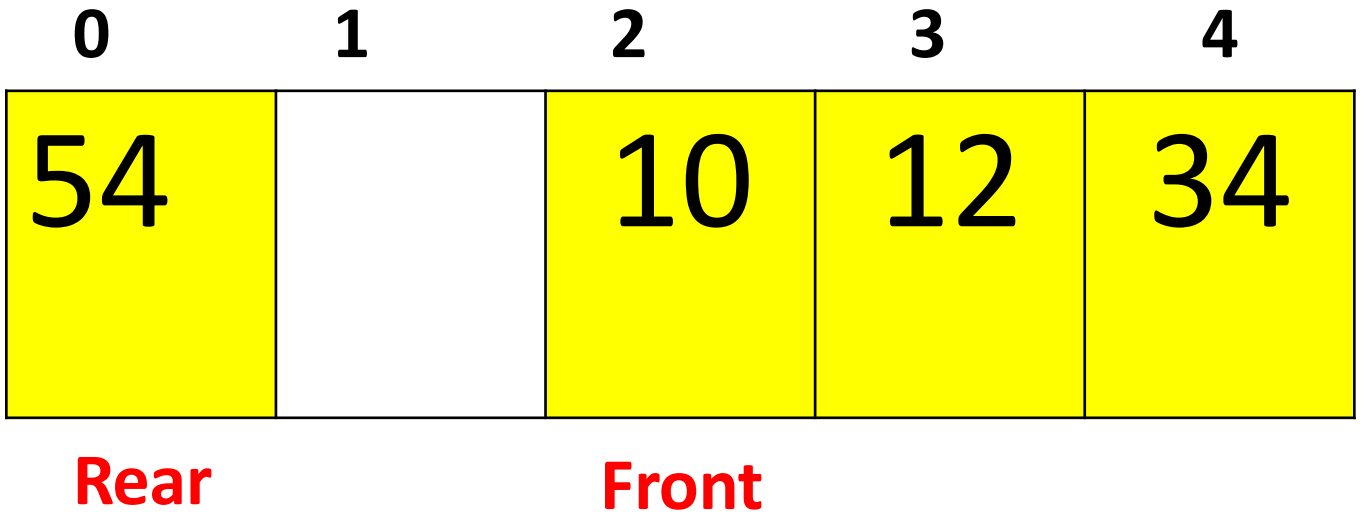
Limitation of queue

- The queue logically moves in the array from left to right.
- After several moves, rear reaches the end, leaving no space for adding new elements.



Solution : Circular Queue

- There is a free space before the front index.
- Use that space for enqueueing new items to the queue.



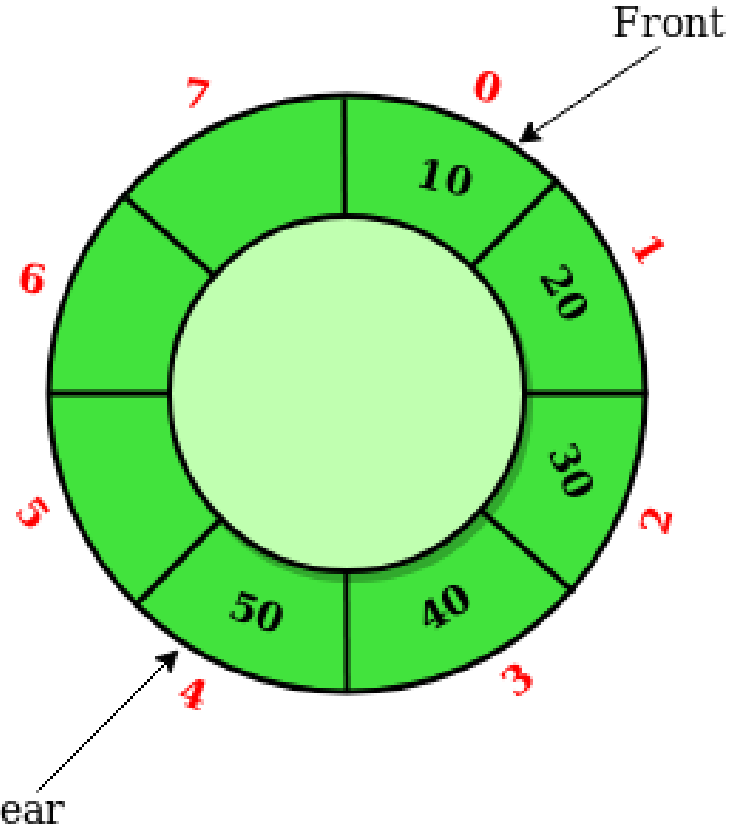
Circular Queue

- To avoid the problem of not being able to insert more items into the queue even when it’s not full, the front and rear arrows wrap around to the beginning of the array.

- There is plenty of extra space :All the positions before the front are unused and can thus be recycled. When either rear or front reaches the end of the array, we reset it to the beginning. The result is a circular queue.

Circular Queue

- It is a logical data structure in which the last position is connected back to the first position to make a circle.

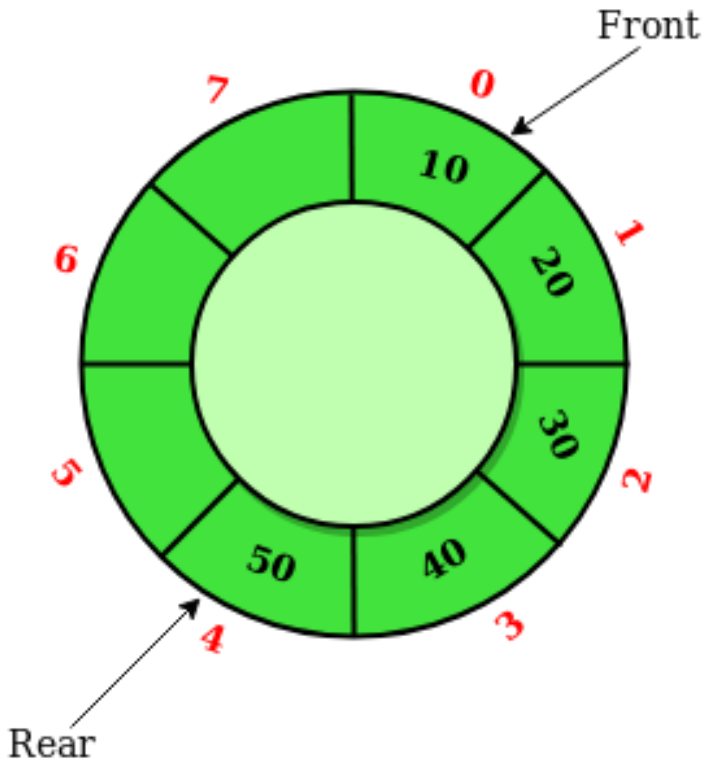


Size of array = N
Current Position = i
Next position = (i+1)%N
Previous Position = (i+N-1)%N

Enqueue() – In Circular Array

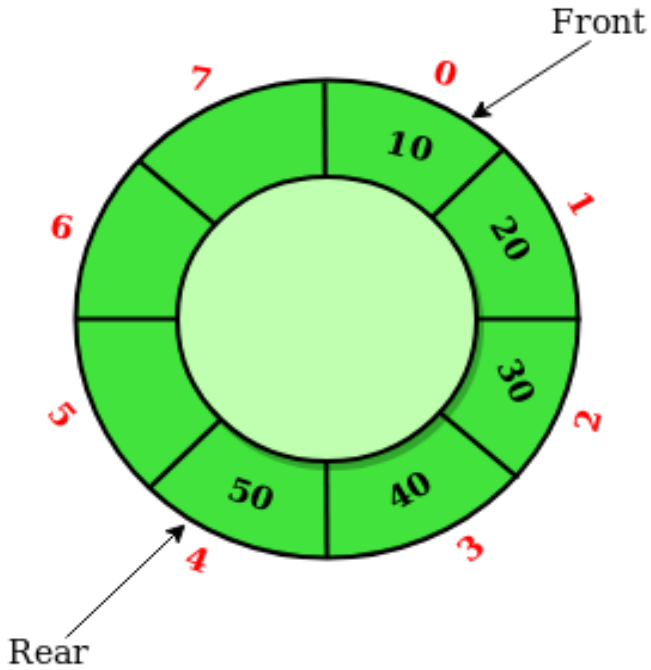
```
public void enqueue(int item) {
   if ((rear +1)%N==front) {
         System.out.println ("Queue is full!! Cannot add more
         elements");
       }else {
       rear = (rear +1)%N;
       array [rear] = item;
       System.out.println (item+ " Added to queue");
       currentSize++;
       }
}
```

Position next to rear



Dequeue() - In Circular Array

```
void dequeue()
   {
      if (isEmpty ())
          System.out.println("Queue is empty!! Can not remove
          element");
          else if (rear == front) { rear = front = -1; }
 else {
          front = (front +1)%N;
          System.out.println(item+ " Removed
          from queue"); currentSize--;
          }
}
```



front( )/peek() – Return front element

```
int Front()
        {
        if(front == -1)
        {
        System.out.println("Queue is empty!! No front element");

        } else{
                return array[front];
        }
```

Deque (Double Ended Queue)

- A deque is a linear list in which elements can be added or removed at either end but not in the middle.
- There can be methods like;
  - ✓ addFirst() / removeFirst()
  - ✓ insertLast() / removeLast()
  - ✓ First()
  - ✓ Last()
  - ✓ Size()
  - ✓ isEmpty() / isFull()

Deque (Double Ended Queue)

		Return Value
• addLast(6)		-
• addFirst(3)		-
• addFirst(5)		-
• first()		5
• removeLast()		6
• removeFirst()		5
• addFirst(2)		-
• last()		3
• isEmpty()		false

Priority Queue

- Both the stack and queue are data structures whose elements are ordered based on the sequence in which they have been inserted.

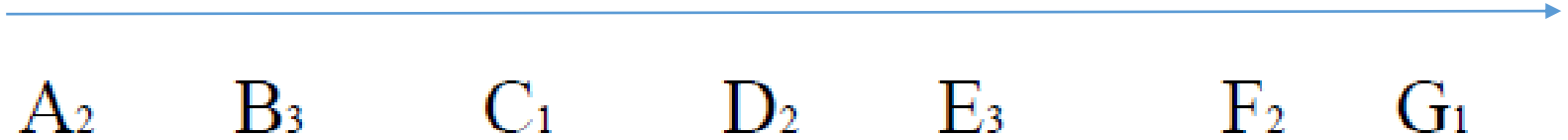
- If there is an intrinsic order among elements themselves (e.g. numeric order or alphabetic order), it is ignored in the stack or queue operations.

Priority Queue

- A priority queue in general is a collection of prioritized elements in which the next element to be removed in the queue is the element that has the highest priority of all elements.

Priority Queue

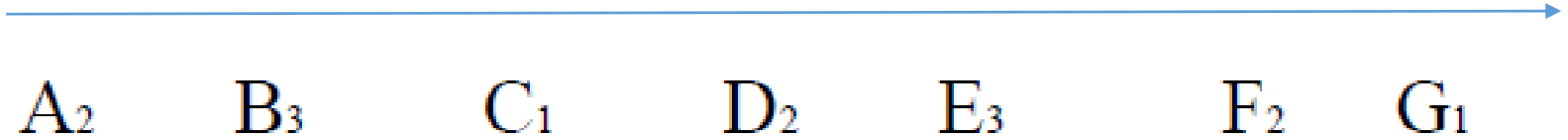
- Elements inserted



- Removal order:

Priority Queue

- Elements inserted



- Removal order: B, E, A, D, F, C, G

Priority Queue - Operations

insert(item, priority);

- Inserts an item with a priority value

max()

- Returns the item with the highest priority

removeMax()

- Removes the item with the highest priority

Priority Queue - Operations

size()

- Returns the size of the queue

isEmpty()

- Returns a Boolean value indicating whether the queue is empty

Questions?

Thank You..