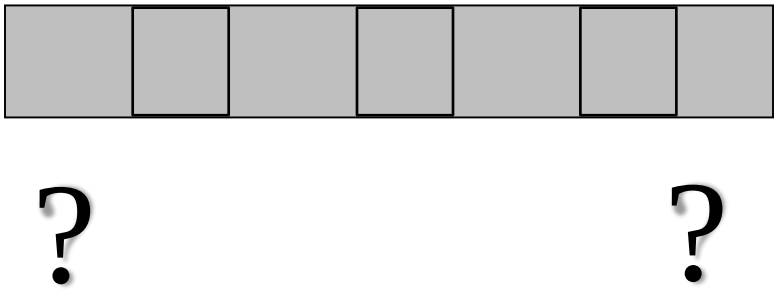
Queue using Array

Introduction and Implementation

Queue using Array

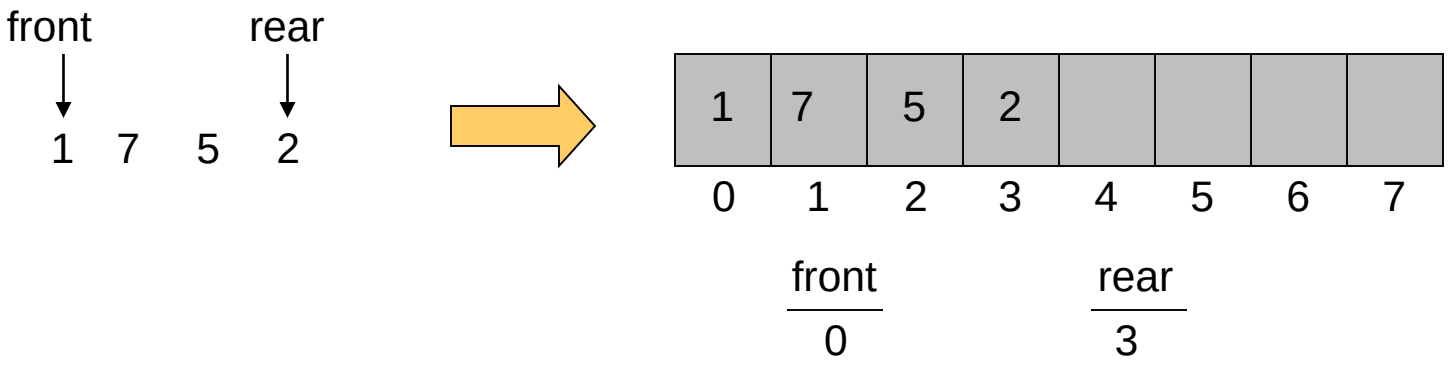
- We know that, two ends are needed for queue one for insertion and another for deletion.



Queue using Array

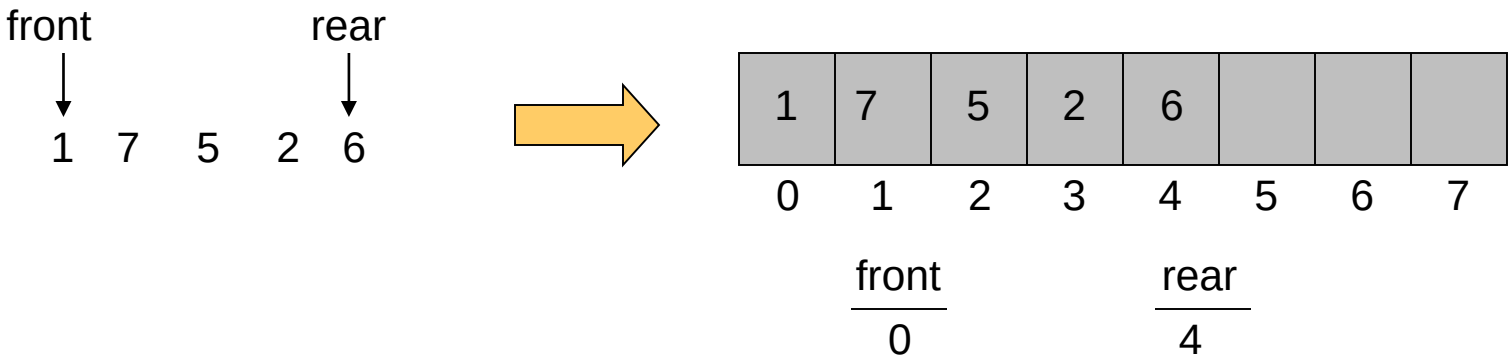
- If we use an array to hold queue elements, both insertions and removal at the front (start) of the array are expensive.
- This is because we may have to shift up to “n” elements.
- For the stack, we needed only one end; for queue we need both.
- To get around this, we will not shift upon removal of an element.

Queue using Array



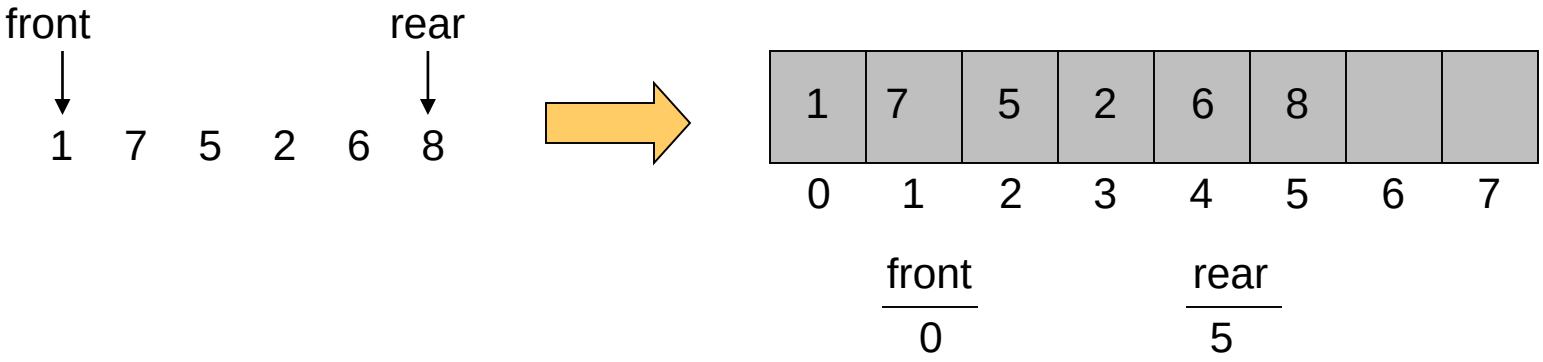
Queue using Array

enqueue(6)



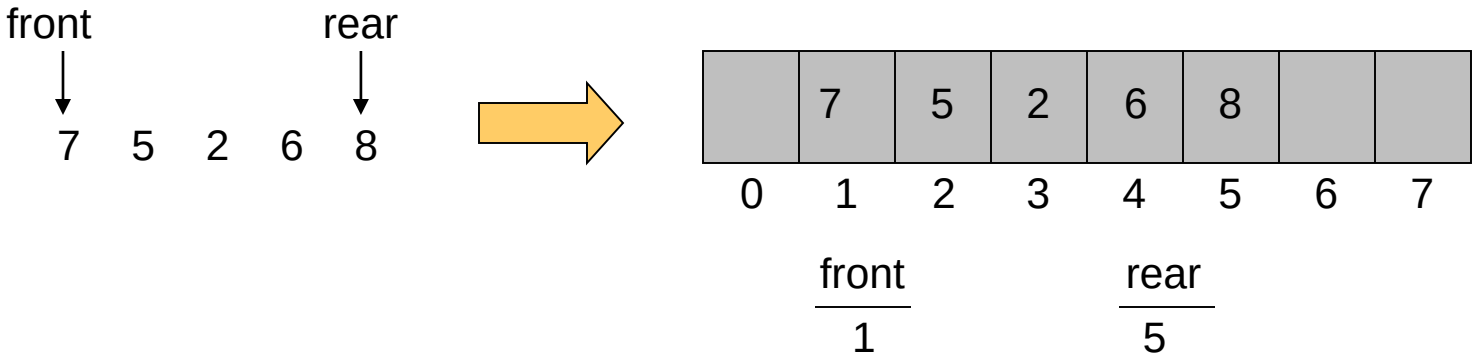
Queue using Array

enqueue(8)



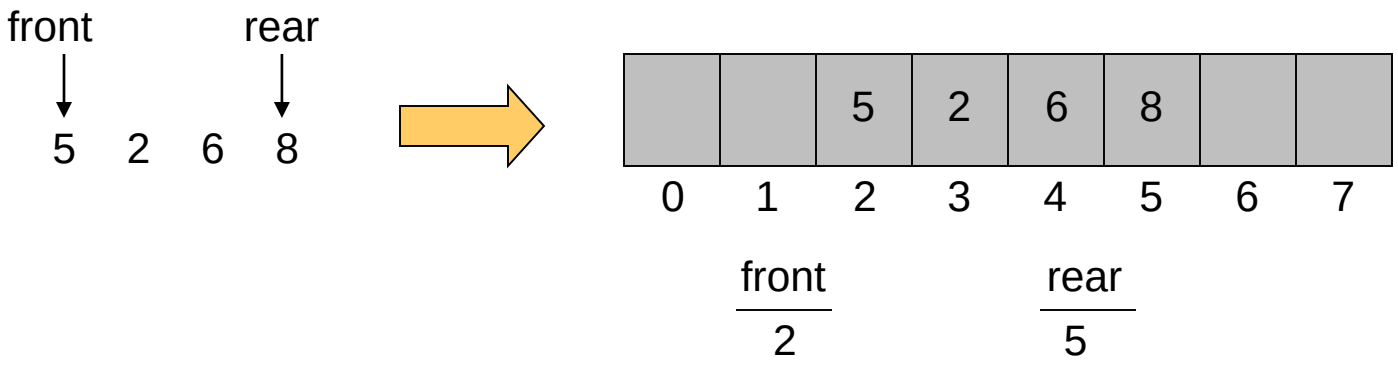
Queue using Array

dequeue()



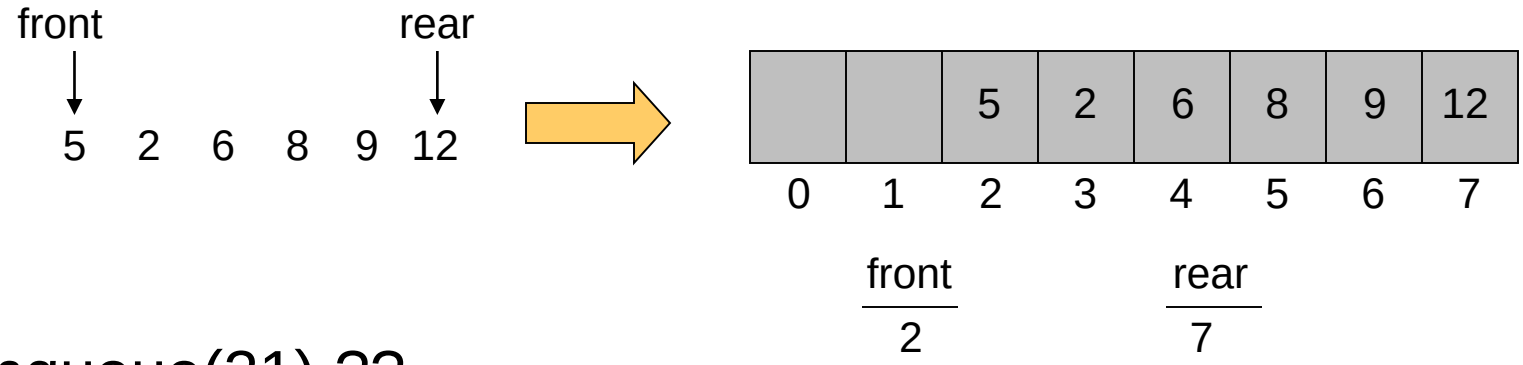
Queue using Array

dequeue()



Queue using Array

enqueue(9)
enqueue(12)



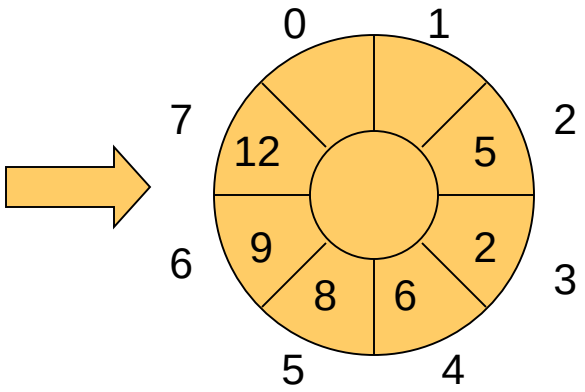
enqueue(21) ??

Queue using Array

- We have inserts and removal running in constant time but we created a new problem.
- Cannot insert new elements even though there are two places available at the start of the array.
- Solution: allow the queue to “wrap around”.
- Before Enqueue and Dequeue we have to handle overflow and underflow problem

Queue using Array (Circular Queue)

enqueue(21)



front = 2

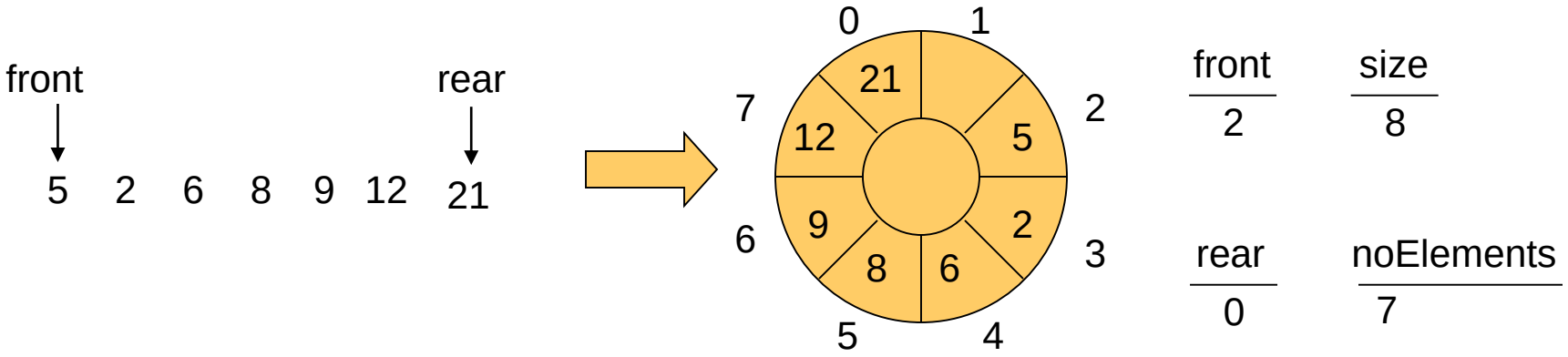
size = 8

rear = 7

noElements = 6

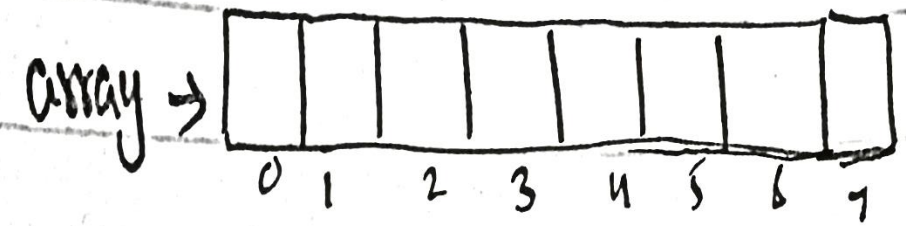
Queue using Array

enqueue(21)




```
Void enqueue(int x)
{
  rear=(rear+1) % size;
  array[rear] = x;
  noElements=noElements+1;

}
```



front	size
-1	8

rear	noElements
-1	0

Enqueue (1)

line 1 → rear = (rear+1) % size.

rear = (-1 +1) % 8

rear' = 0 % 8

rear = 0

line 2 → array[0] = 1;

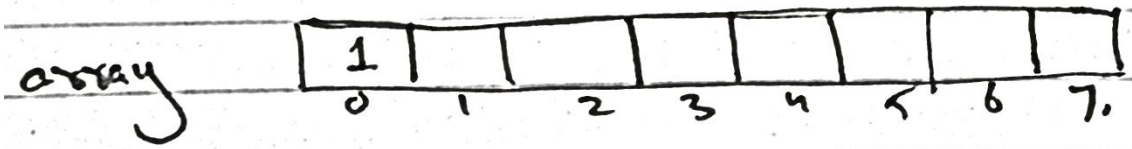
front	size
0	8

line 3 → noElements = noElement + 1;

= 0 + 1

= 1

rear	noElements
0	1

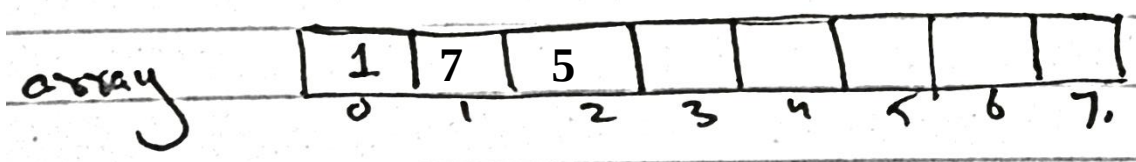


enqueue(7)

```
rear=(rear+1) % size;     // rear=(0+1)%8 => 1
array[rear] = x;          //  array[1]=7;
noElements=noElements+1;// noElements=1+1=> 2
```

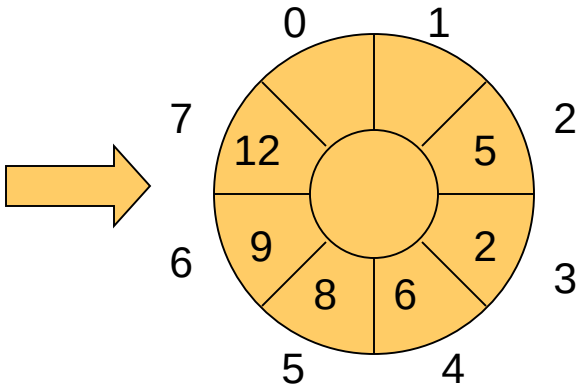
enqueue(5)

```
rear=(rear+1) % size;     // rear=(1+1)%8 => 2
array[rear] = x;          //  array[2]=5;
noElements=noElements+1;// noElements=2+1=> 3
```



Queue using Array (Circular Queue)

enqueue(21)



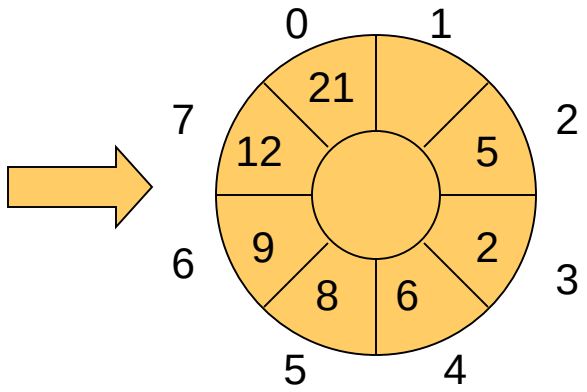
front = 2 size = 8

rear = 7 noElements = 6

```
rear=(rear+1) % size;     // rear=(7+1)%8
                                  = 8%8 = 0
array[rear] = x;          //  array[0]=21;
noElements=noElements+1;// noElements=6+1=> 7
```

Queue using Array

enqueue(21)



front	size
2	8

rear	noElements
0	7

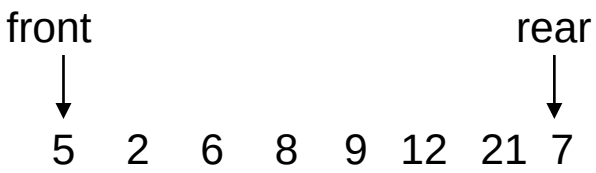
enqueue(7)

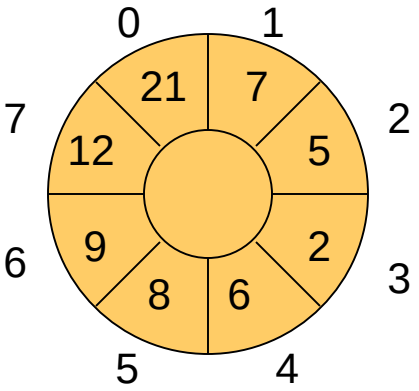
```
rear=(rear+1) % size;     // rear=(0+1)%8
                                = 1%8 = 1
array[rear] = x;          //  array[1]=7;

noElements=noElements+1;// noElements=7+1=> 8
```

Queue using Array

enqueue(7)





front = 2

size = 8

rear = 1

noElements = 8

enqueue(20) = ?

queue is full, overflow problem occur.

Queue using Array

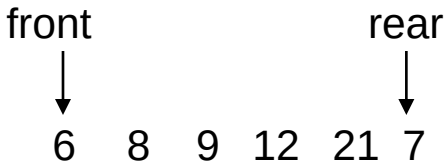
- Before Enqueue and Dequeue we have to handle overflow and underflow problem.

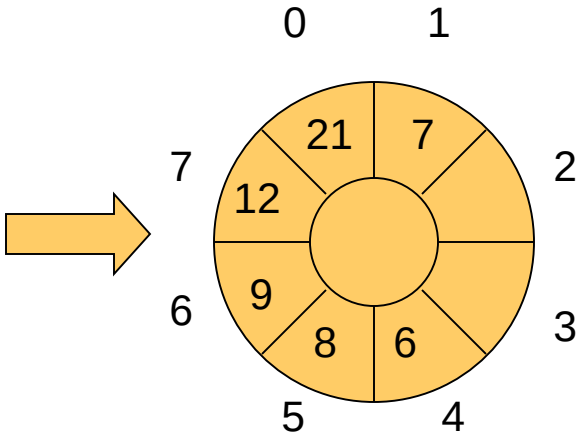
```
int isFull()
{
    return noElements == size;
}

int isEmpty()
{
    return noElements == 0;
}
```

Queue using Array

dequeue()
dequeue()





front = 4 size = 8

rear = 1 noElements = 6

```
int dequeue()
{
     int x = array[front];
     front = (front+1)%size;
     noElements = noElements-1;
     return x;
}
```

Uses of Queues

- ***I/O buffers***

  Scheduling queues in a multi-user computer system e.g. *Printer queue*:

  When files are submitted to a printer, they are placed in the printer queue. The printer software executes an algorithm something like:

```
for (;;)
{
 while (printerQueue.empty())
     sleep 1;
 printFile = printerQueue.removeQ();
 Print(printFile);}
```

Uses of Queues

Resident queue: On disk, waiting for memory

Ready queue: In memory —needs only CPU to run

Suspended queue: Waiting for I/O transfer or to be reassigned the CPU

Priority Queues

- *Priority queues* are queues in which *items are ordered by priority* rather than temporally (i.e. order in which received).

- Any queue implementation then can be taken and modified to acquire a priority queue.

- The only needed modification is the Enqueue() method.

- Instead of unconditionally adding to the back, the queue must be scanned for the correct insertion point.

Queues vs. Stacks

- **Stacks are a LIFO container => store data in the reverse of order received**
- **Queues are a FIFO container => store data in the order received**
- **Stacks then suggest applications where some sort of reversal or unwinding is desired.**
- **Queues suggest applications where service is to be rendered relative to order received.**
- **Stacks and Queues can be used in conjunction to compare different orderings of the same data set.**