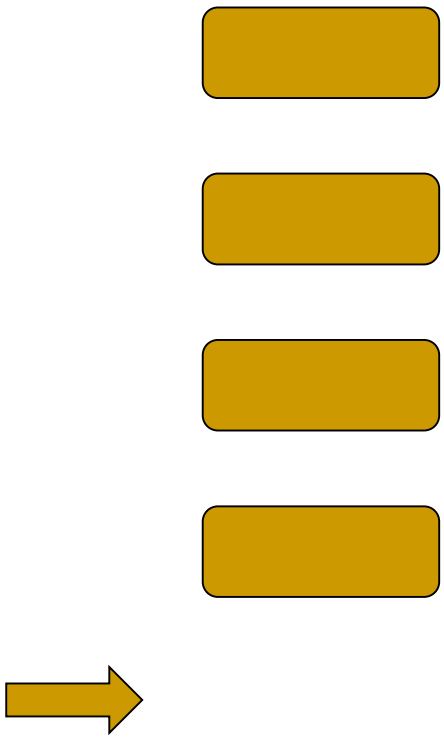
Queue

Joy Mukherjee

Queue

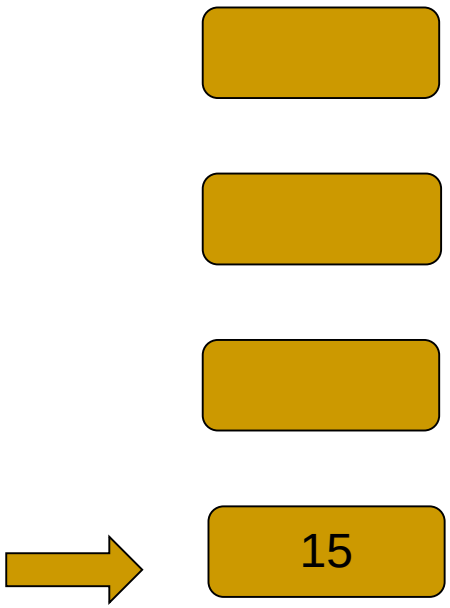
- Queue is a linear data structure
- FIFO(First In First Out)
  - The first one inserted is the first one deleted
  - The last one inserted is the last one deleted

Operations: Initially Queue is Empty



Front = -1
Rear = -1

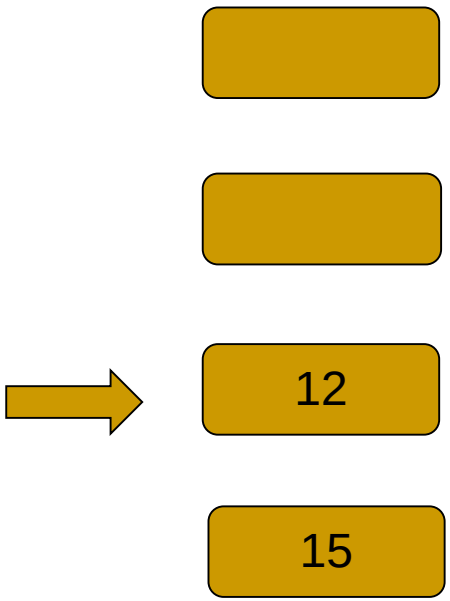
Operations: Enqueue 15



Rear = 0

Front = -1

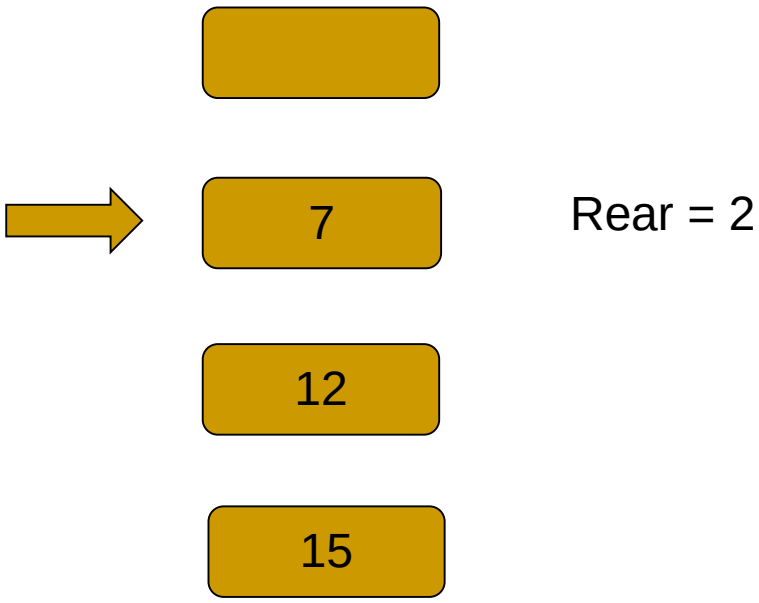
Operations: Enqueue 12



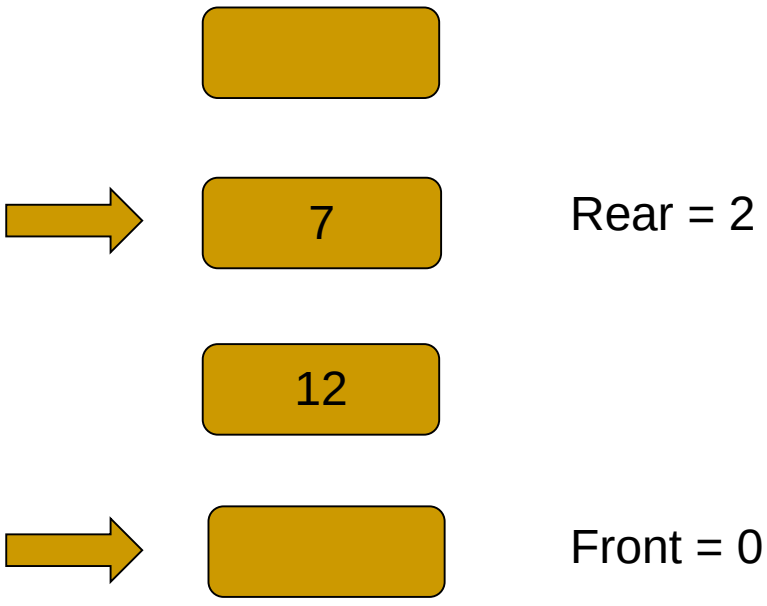
Rear = 1

Front = -1

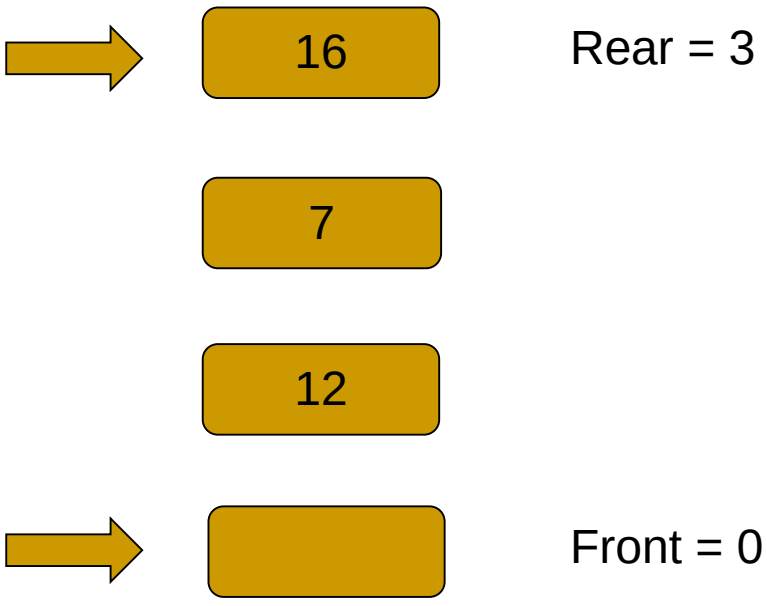
Operations: Enqueue 7



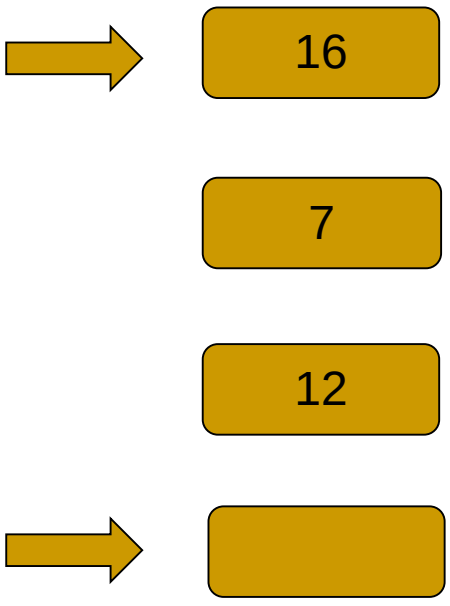
Operations: Dequeue

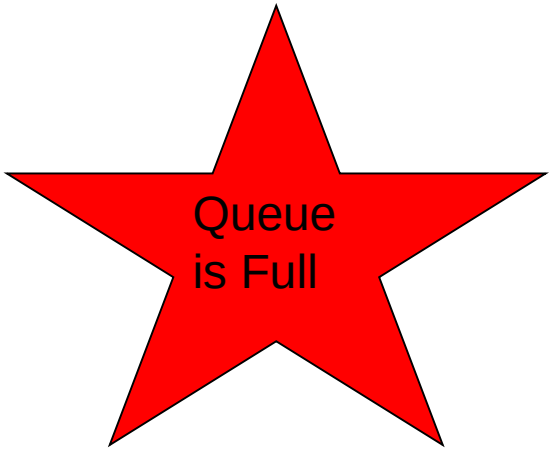


Operations: Enqueue 16

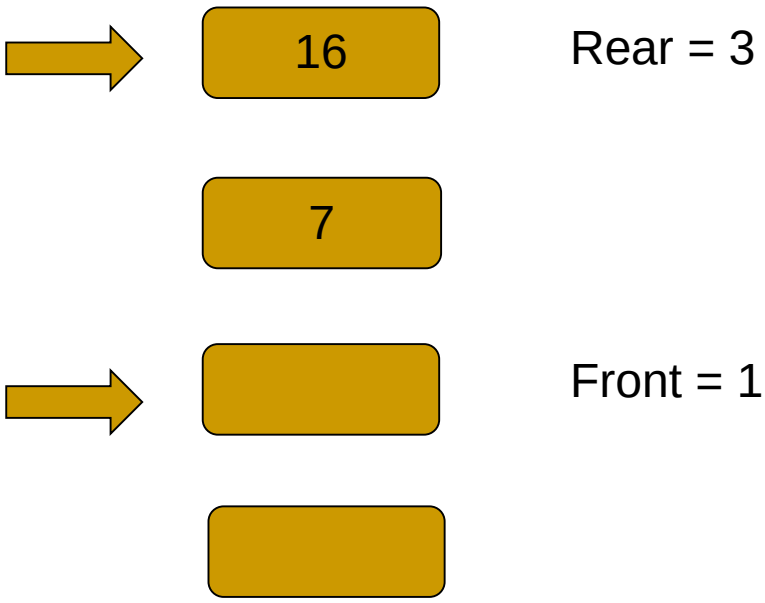


Operations: Enqueue 57

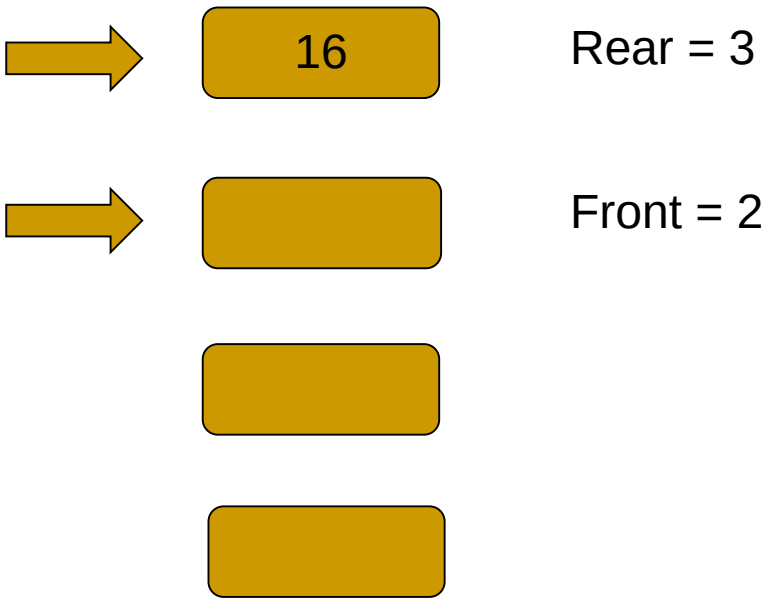




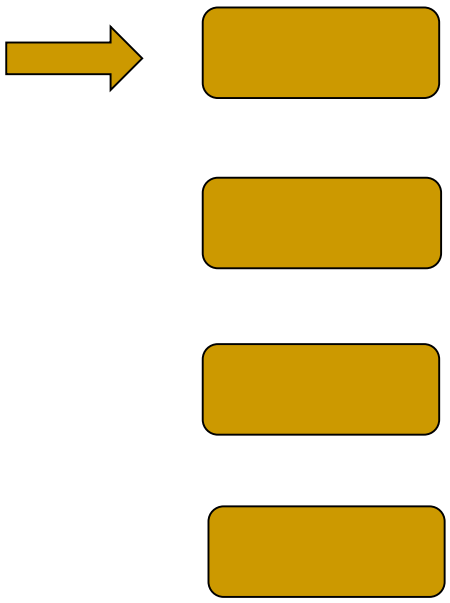
Operations: Dequeue



Operations: Dequeue

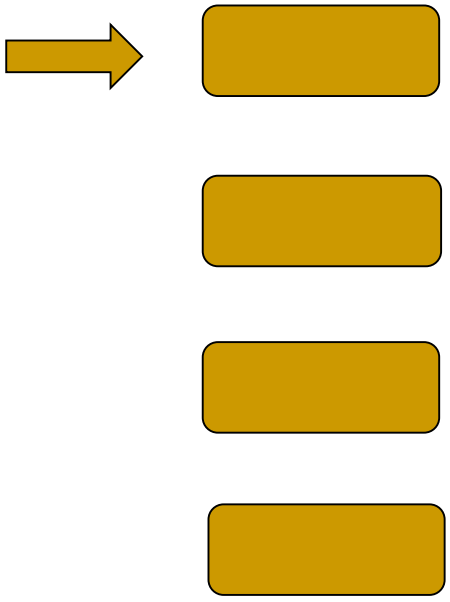


Operations: Dequeue

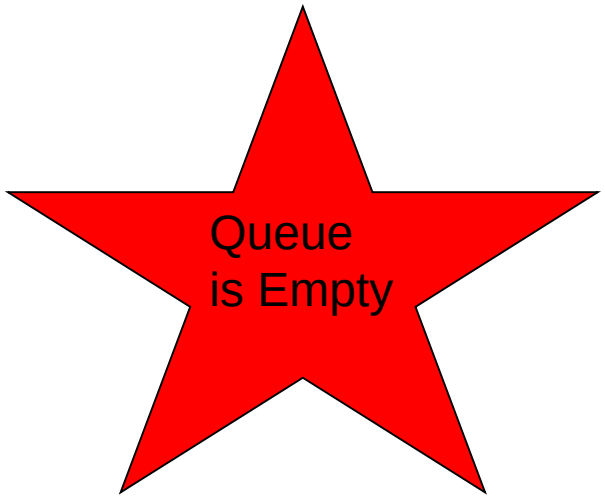


Rear = 3
Front = 3

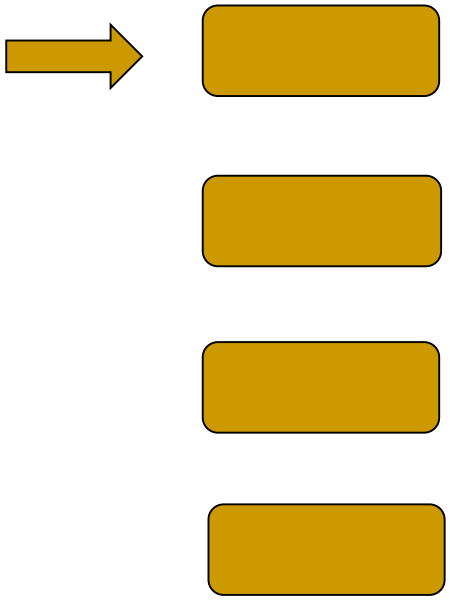
Operations: Dequeue



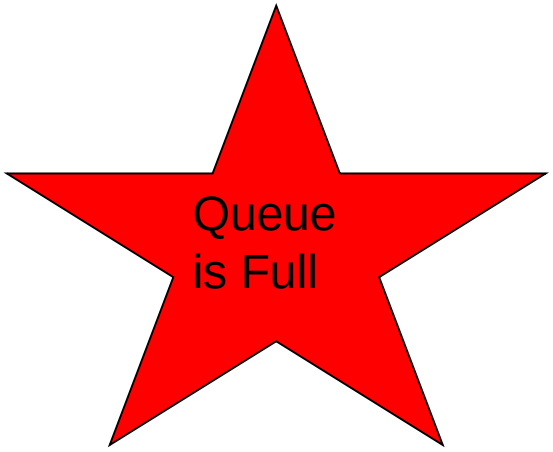
Rear = 3
Front = 3



Operations: Enqueue 57



Rear = 3
Front = 3



Queue

```
#define  MAXLEN   4

struct qu{
        int A[MAXLEN];
        int rear;
        int front;
}; // Create a data type struct qu

typedef struct qu queue;
```

Queue Operations

```
queue init ()
{
        queue Q;
        Q.front = -1;
        Q.rear  = -1;
        return Q;
}
```

```
int isEmpty (queue Q)
{
   if(Q.front == Q.rear)
        return 1;
    return 0;
}
```

```
int isFull (queue Q)
{
   if (Q.Rear == MAXLEN - 1)
        return 1;
   return 0;
}
```

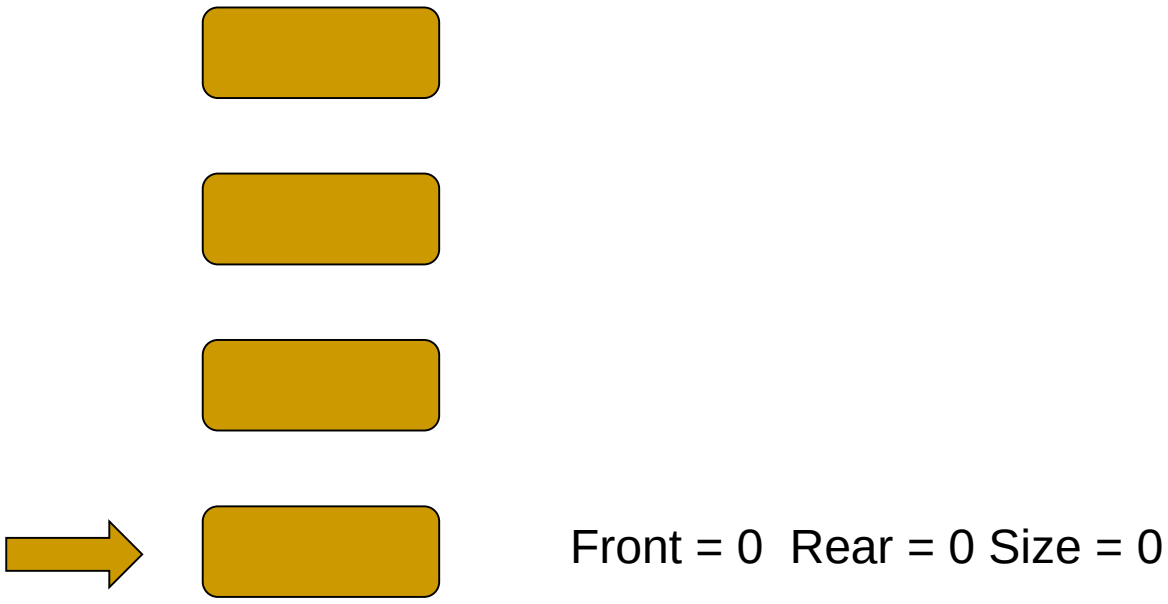
Queue Operations

```
queue enqueue (queue Q, int data)
{
  if (isFull(Q)) {
    printf("Queue is Full\n");
    return Q;
   }
   ++Q.rear;
   Q.A[Q.rear] = data;
   return Q;
}
```

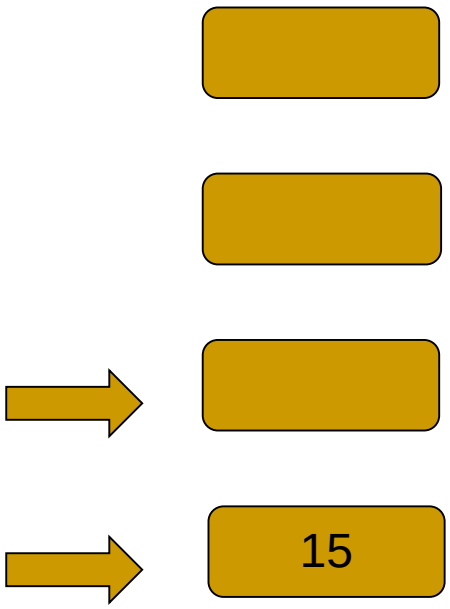
```
queue dequeue (queue Q)
{
  if (isEmpty(Q)) {
    printf("Queue is Empty\n");
    return Q;
  }
  ++Q.front;
  return Q;
}
```

Circular Queue

Operations: Initially Queue is Empty



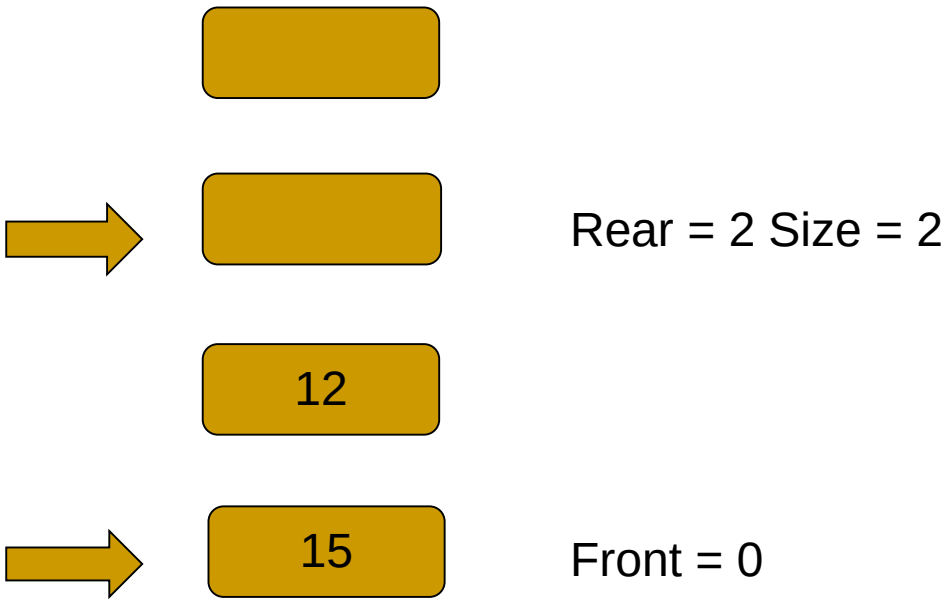
Operations: Enqueue 15



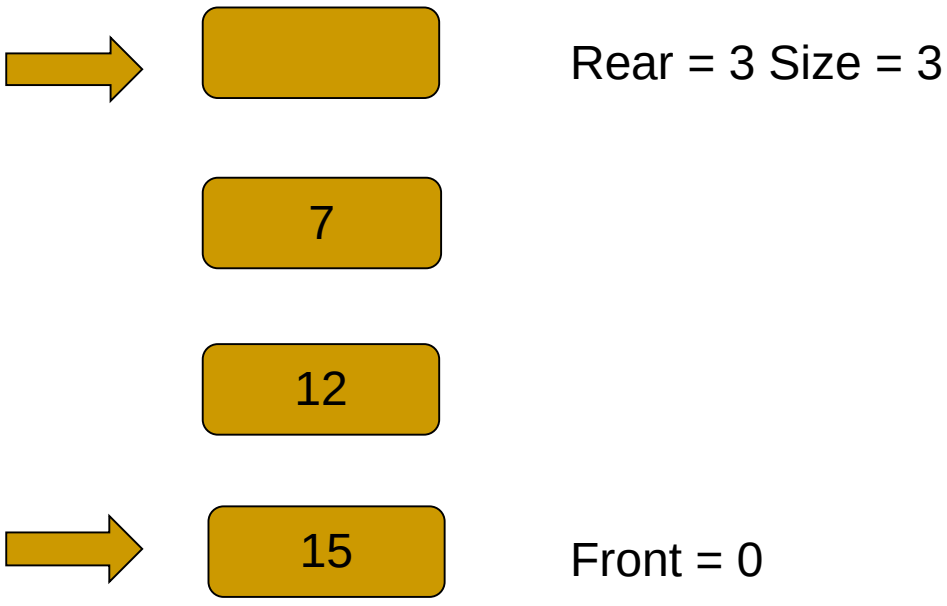
Rear = 1 Size = 1

Front = 0

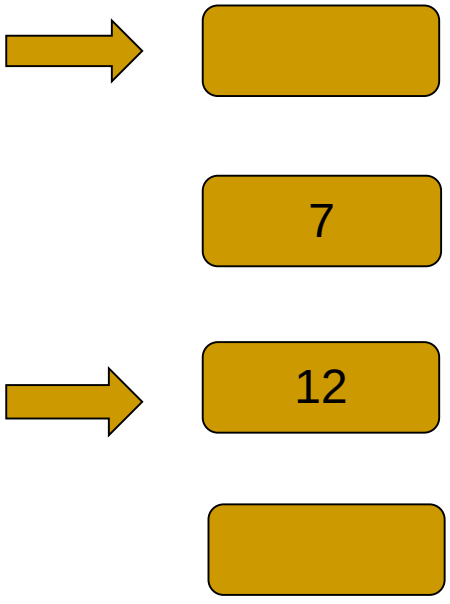
Operations: Enqueue 12



Operations: Enqueue 7



Operations: Dequeue

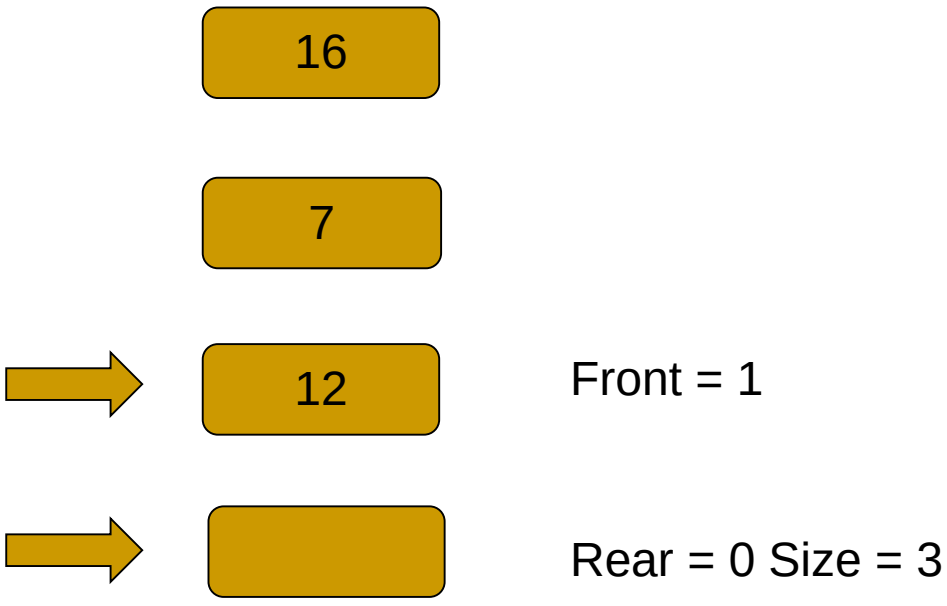


Rear = 3 Size = 2

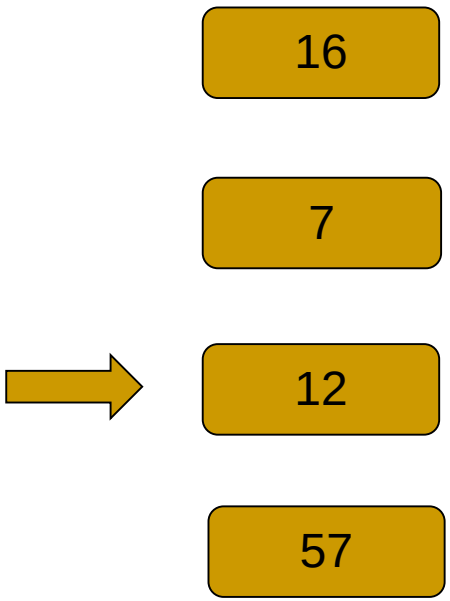
Q.Rear = (Q.rear + 1)%MAXLEN

Front = 1

Operations: Enqueue 16

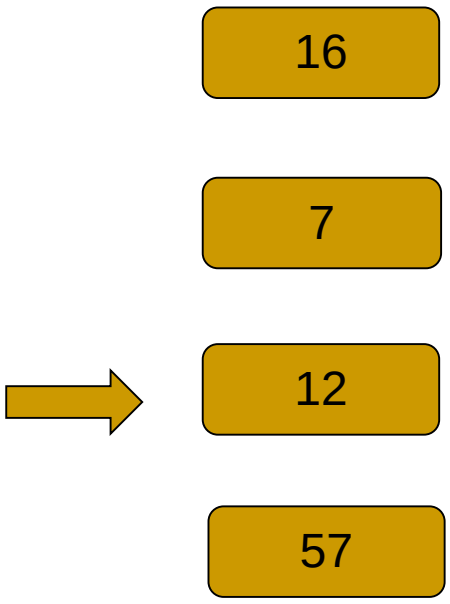


Operations: Enqueue 57



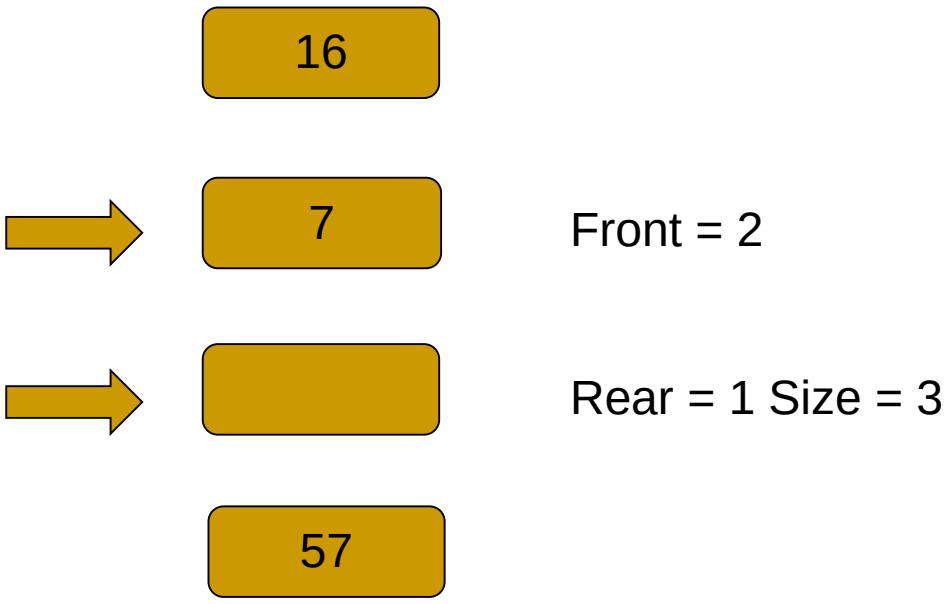
Front = 1 Rear = 1 Size = 4

Operations: Enqueue 99

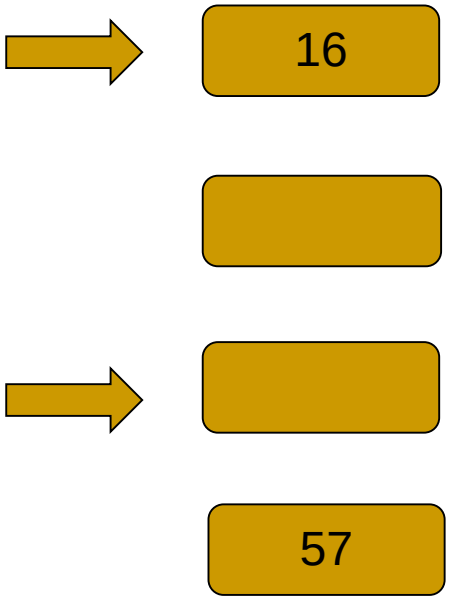


Front = 1 Rear = 1 Size = 4

Operations: Dequeue



Operations: Dequeue

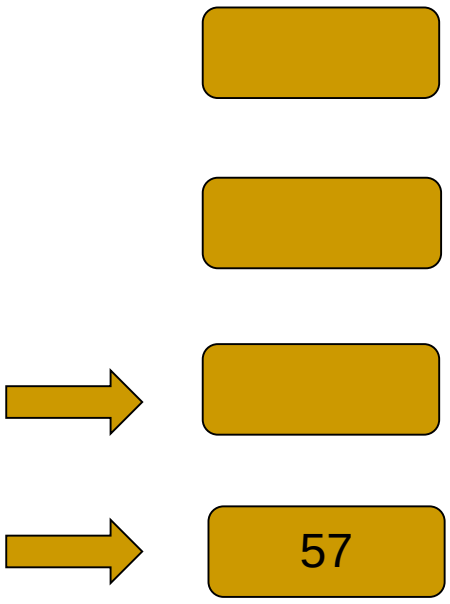


Front = 3

Q.front = (Q.front + 1)%MAXLEN

Rear = 1 Size = 2

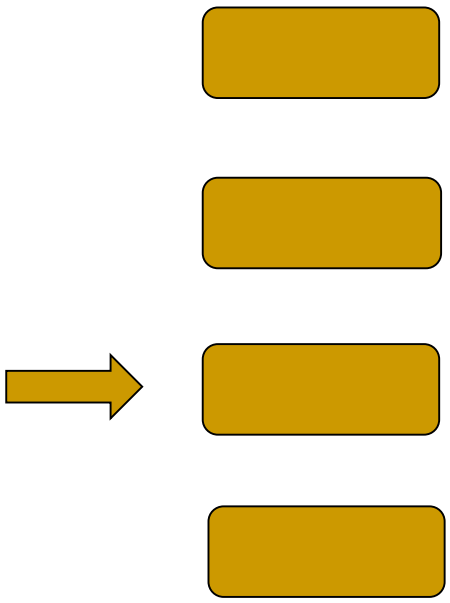
Operations: Dequeue



Rear = 1 Size = 1

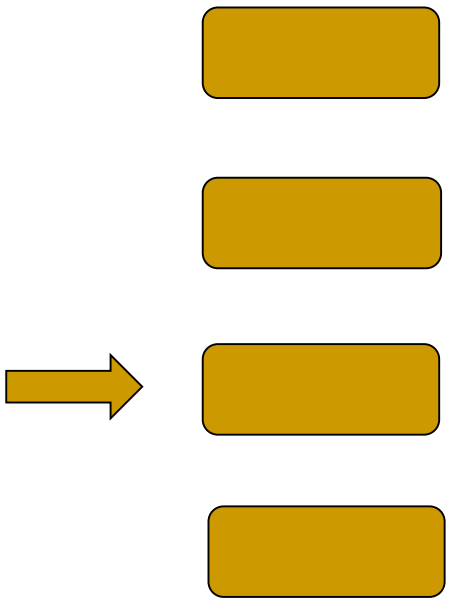
Front = 0

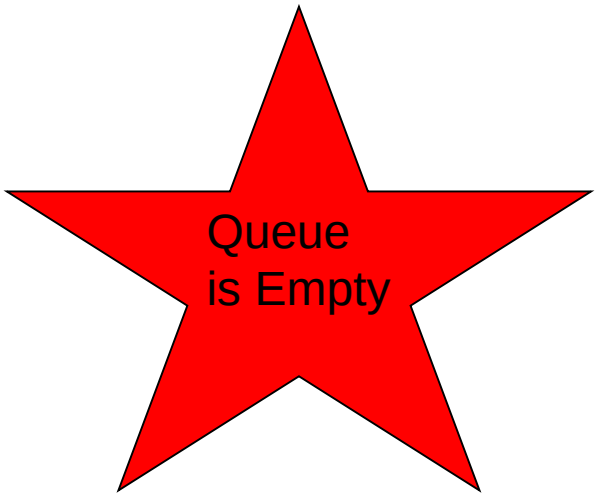
Operations: Dequeue



Front = 1 Rear = 1 Size = 0

Operations: Dequeue





Front = 1 Rear = 1 Size = 0

Circular Queue

```
#define   MAXLEN   4

struct cqu{
        int A[MAXLEN];
        int front;
        int rear;
        int size;
};

typedef  struct  cqu  cqueue;
```

Circular Queue Operations

```
cqueue init ()
{
        cqueue Q;
        Q.front = 0;
        Q.rear  = 0;
        Q.size  = 0;
        return Q;
}
```

```
int isEmpty (cqueue Q)
{
   if(Q.size == 0)
        return 1;
    return 0;
}
```

```
int isFull (cqueue Q)
{
   if (Q.size == MAXLEN)
        return 1;
   return 0;
}
```

Circular Queue Operations

```
cqueue enqueue (cqueue Q, int data)
{
  if (isFull(Q)) {
    printf("Queue is Full\n");
    return Q;
  }
  Q.A[Q.rear] = data;
  Q.size++;
  Q.rear = (Q.rear + 1) % MAXLEN;
  return Q;
}
```

```
cqueue dequeue (cqueue Q)
{
  if (isEmpty(Q)) {
    printf("Queue is Empty\n");
    return Q;
  }
  Q.front = (Q.front + 1) % MAXLEN;
  Q.size--;
  return Q;
}
```

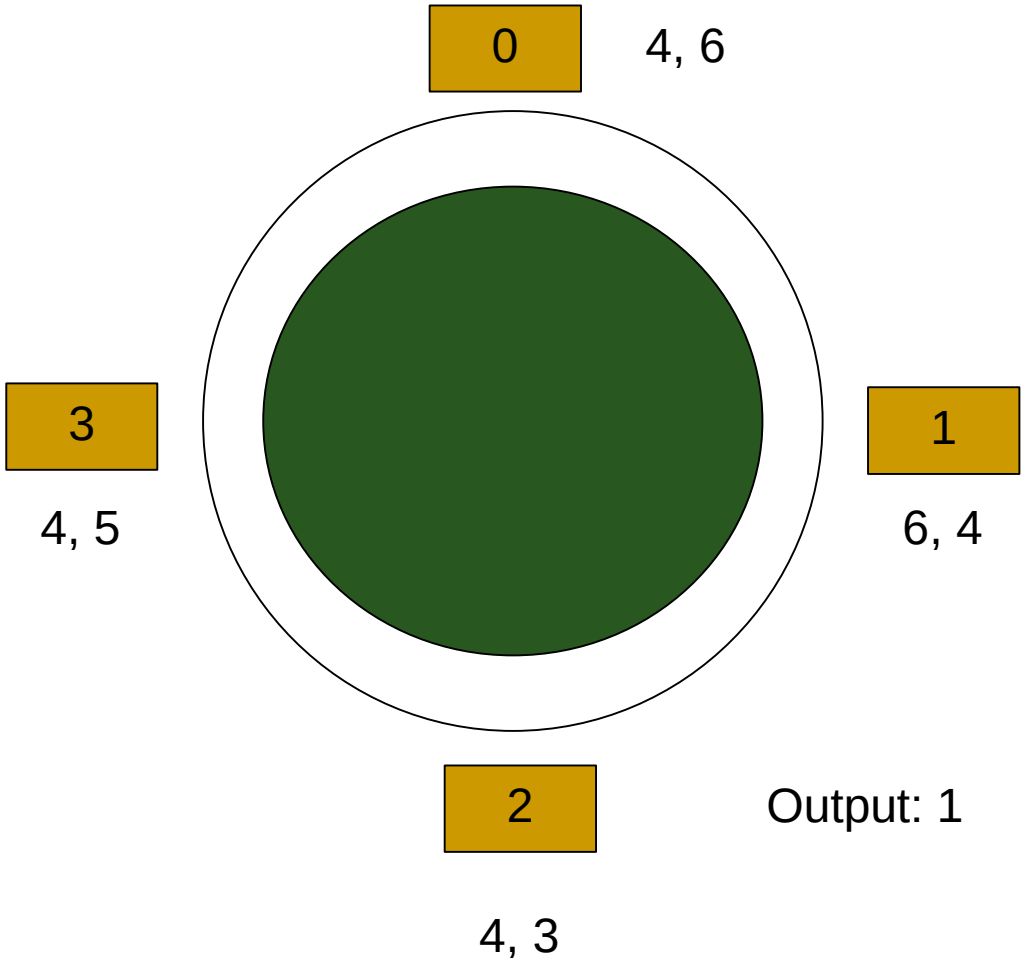
Circular Queue Operations

```
void print (cqueue Q)
{
        int i;
        if (isEmpty(Q)) {
                printf("Queue is Empty ");
                return ;
        }
        for (i = 0; i < Q.size; i++)
                printf("%4d", Q.A[(i + Q.front) % MAXLEN]);
}
```

Application: Queue

- Graph Algorithms: Breadth First Traversal (Linear Queue), Minimum spanning tree (Prim's Algorithm: Priority Queue), Shortest Path (Dijkstra's Algorithm: Priority Queue)
- Circular Tour: Circular Queue
- Window based Algorithms: Deque

Application of Circular Queue



Input:

1. Amount of petrol
2. Distance to the next petrol pump in clockwise direction

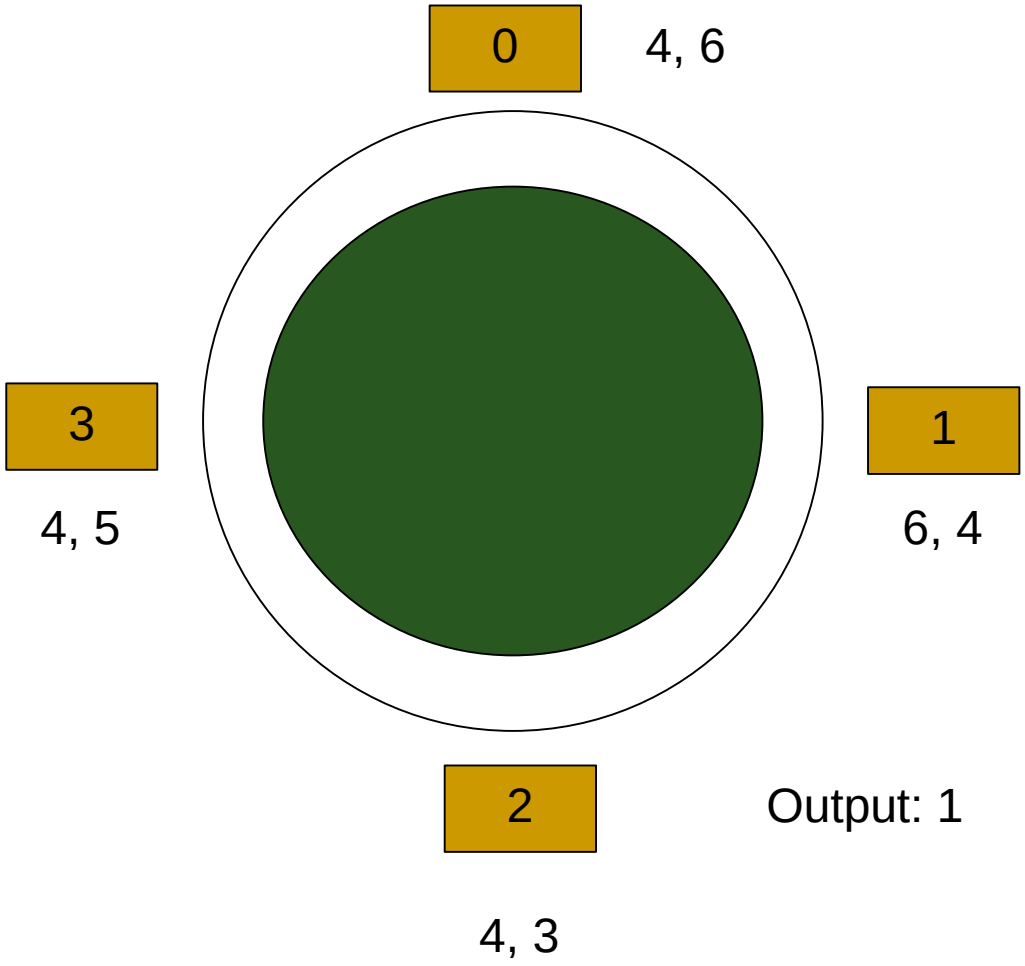
Constraints:

1. Infinite capacity of the petrol tank of the car
2. With each unit of petrol, the car can move a unit distance

Output:

The ID of the first petrol pump, from which the car can start the tour, while visiting all the petrol pumps before returning back to the first one. If no such petrol pump exists, print -1.

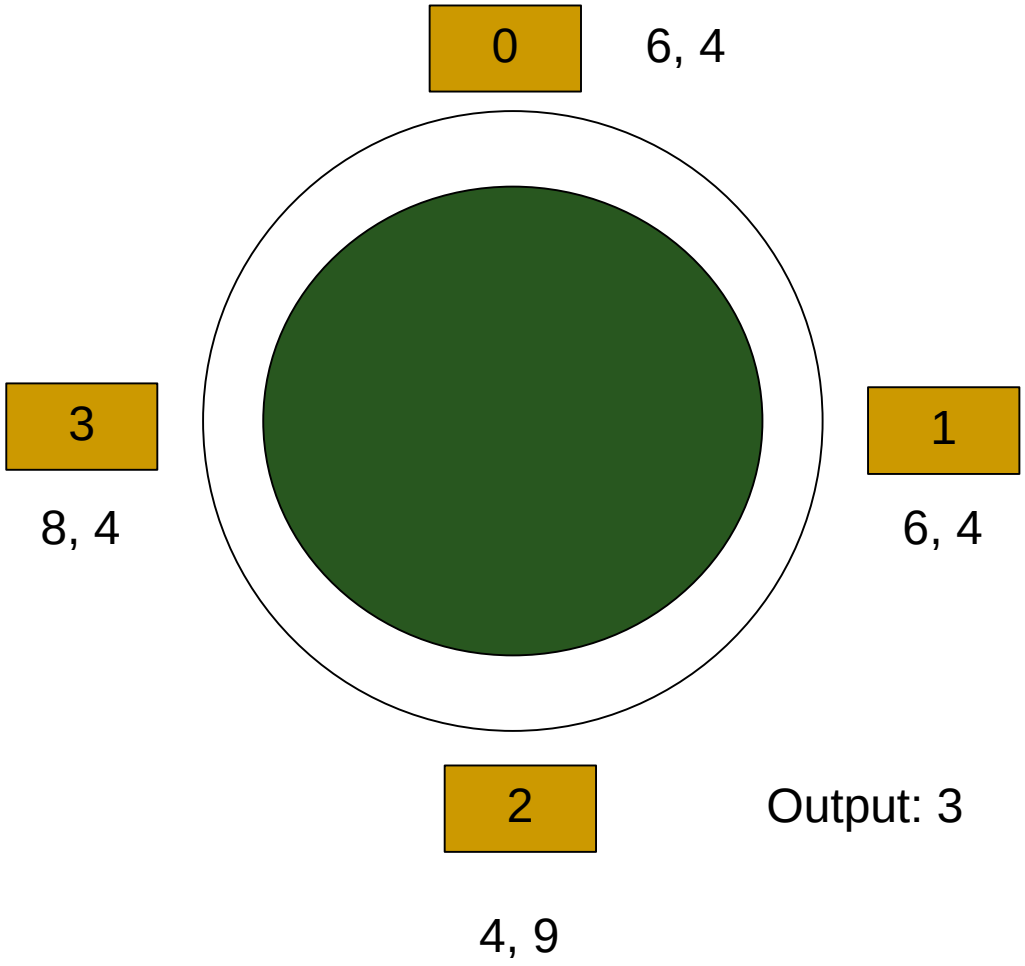
Application of Circular Queue



Output: 1

```
start = 0 end = 1 petrol = 0
petrol = P[start] – D[start]
while(start != end || petrol < 0) {
if(petrol < 0 && start != end) {
    petrol -= P[start] – D[start]
    start = (start+1)%n
    If(start == 0) return -1;
}
petrol += P[end] – D[end]
end = (end+1)%n
}
return start
// start == end && petrol >= 0
```

Application of Circular Queue



Output: 3

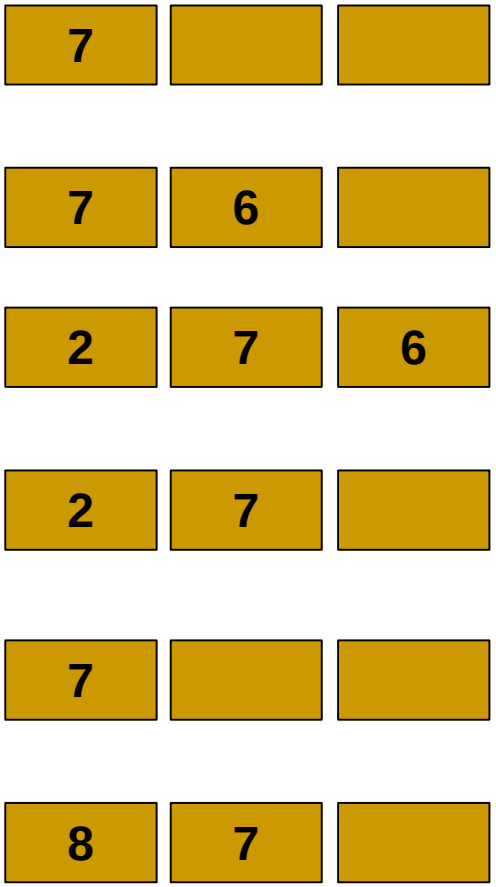
```
start = 0 end = 1 petrol = 0
petrol = P[start] – D[start]
while(start != end || petrol < 0) {
if(petrol < 0 && start != end) {
    petrol -= P[start] – D[start]
    start = (start+1)%n
    If(start == 0) return -1;
}
petrol += P[end] – D[end]
end = (end+1)%n
}
return start
// start == end && petrol >= 0
```

Double Ended Queue

Double Ended Queue (Deque)

- The enqueue and dequeue operations can be done at both ends.
  - EnqueueAtRear
  - DequeueFromRear
  - EnqueueAtFront
  - DequeueFromFront
- The operations are implemented using linked list

Deque Operations



EnqueueAtRear(7)

EnqueueAtRear(6)

EnqueueAtFront(2)

DequeueFromRear()

DequeueFromFront()

EnqueueAtFront(8)

Application of Deque

- Given an array A of size n, and an integer k (≤ n), write a program that prints the maximum element for each continuous subarray of size k of the array.

0	1	2	3	4	5	6
8	9	4	7	6	5	8

n= 7 A [] = {8, 9, 4, 7, 6, 5, 8} k = 3

9 9 7 7 8

Running time of Naïve Algorithm: O(k*(n-k+1)) = O(nk)

Sliding Window Maximum: Approach

- Scan the array from left to right (0 to n-1)
- For each element,
  - Maintain the window (current contiguous subarray of size k)
  - Once the window is fixed,
  - The current integer < The integer at rear
    - EnqueueAtRear(current integer) [The integer may be a candidate for maximum in subsequent subarrays]
  - The current integer >= The integer at rear
    - DequeueFromRear [The integer can not be a candidate for maximum in subsequent subarrays]

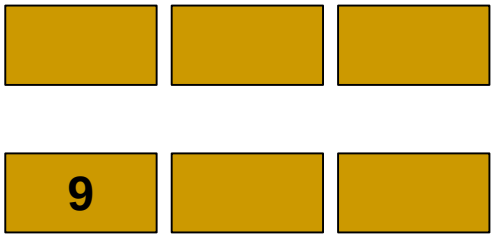
Sliding Window Maximum

0	1	2	3	4	5	6
8	9	4	7	6	5	8

EnqueueAtRear(8)

Sliding Window Maximum

0	1	2	3	4	5	6
8	9	4	7	6	5	8



DequeueFromRear()

EnqueueAtRear(9)

Sliding Window Maximum

0	1	2	3	4	5	6
8	9	4	7	6	5	8

9	4	

EnqueueAtRear(4)

PrintAtFront() prints 9

Sliding Window Maximum

0	1	2	3	4	5	6
8	9	4	7	6	5	8

DequeueFromRear()

EnqueueAtRear(7)

PrintAtFront() prints 9

Sliding Window Maximum

0	1	2	3	4	5	6
8	9	4	7	6	5	8

DequeueFromFront()

EnqueueAtRear(6)

PrintAtFront() prints 7

Sliding Window Maximum

0	1	2	3	4	5	6
8	9	4	7	6	5	8

7	6	5

EnqueueAtRear(8)

PrintAtFront() prints 7

Sliding Window Maximum

0	1	2	3	4	5	6
8	9	4	7	6	5	8

DequeueFromFront()

DequeueFromRear()

DequeueFromRear()

EnqueueAtRear(8)

PrintAtFront() prints 8

Sliding Window Maximum: Algorithm

```
Create a Double Ended Queue d of size k (contains the indices of the array)
for (i = 0; i < k; i++)    {
   while (d is not empty && a[i] >= a[d.rear()])
         d.DequeueFromRear();
   d.EnqueueAtRear(i);
}
print a[d.front()];
for (; i < n; ++i)    {
    while (d is not empty && d.front() <= i - k) // maintain the current window
          d.DequeueFromFront();
    while (d is not empty && a[i] >= a[d.rear()])
         d.DequeueFromRear();
    d.EnqueueAtRear(i);
    print a[d.front()];
}
```

Sliding Window Maximum

0	1	2	3	4	5	6
8	9	4	3	3	5	8

EnqueueAtRear(0)

Sliding Window Maximum

0	1	2	3	4	5	6
8	9	4	3	3	5	8

DequeueFromRear()

EnqueueAtRear(1)

Sliding Window Maximum

0	1	2	3	4	5	6
8	9	4	3	3	5	8

EnqueueAtRear(2)

PrintAtFront() prints a[1] = 9

Sliding Window Maximum

0	1	2	3	4	5	6
8	9	4	3	3	5	8

1	2	3

EnqueueAtRear(3)

PrintAtFront() prints a[1] = 9

Sliding Window Maximum

0	1	2	3	4	5	6
8	9	4	3	3	5	8

DequeueFromFront()

DequeueFromRear()

EnqueueAtRear(4)

PrintAtFront() prints a[2] = 4

Sliding Window Maximum

0	1	2	3	4	5	6
8	9	4	3	3	5	8

DequeueFromFront()

DequeueFromRear()

EnqueueAtRear(5)

PrintAtFront() prints a[5] = 5

Sliding Window Maximum

0	1	2	3	4	5	6
8	9	4	3	3	5	8

DequeueFromRear()

EnqueueAtRear(6)

PrintAtFront() prints a[6] = 8