UTM
UNIVERSITI TEKNOLOGI MALAYSIA
innovative ● entrepreneurial ● global
ONLINE LEARNING

Queue with Array Implementation

SCSJ2013 Data Structures & Algorithms

Nor Bahiah Hj Ahmad & Dayang Norhayati A. Jawawi

Faculty of Computing

Objectives

- Queue concepts and applications.
- Queue structure and operations
- Implement queue using array

Introduction to Queue

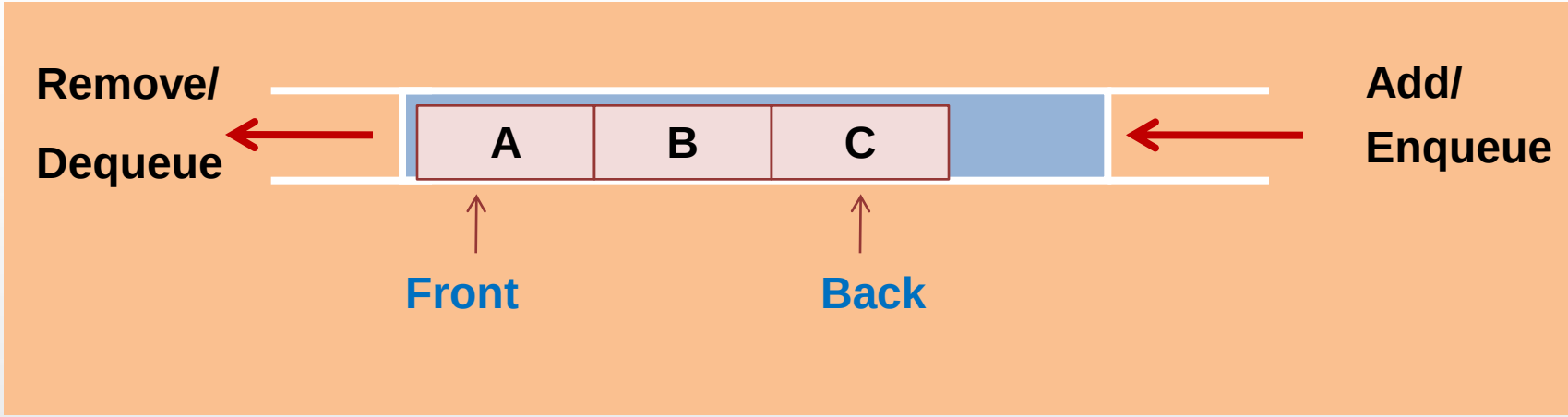
- New items enter at the back of the queue.
- Items leave from the front queue.
- Implement First-in, first-out (FIFO) property.

- Queue is important in simulation & analyzing the behavior of complex systems

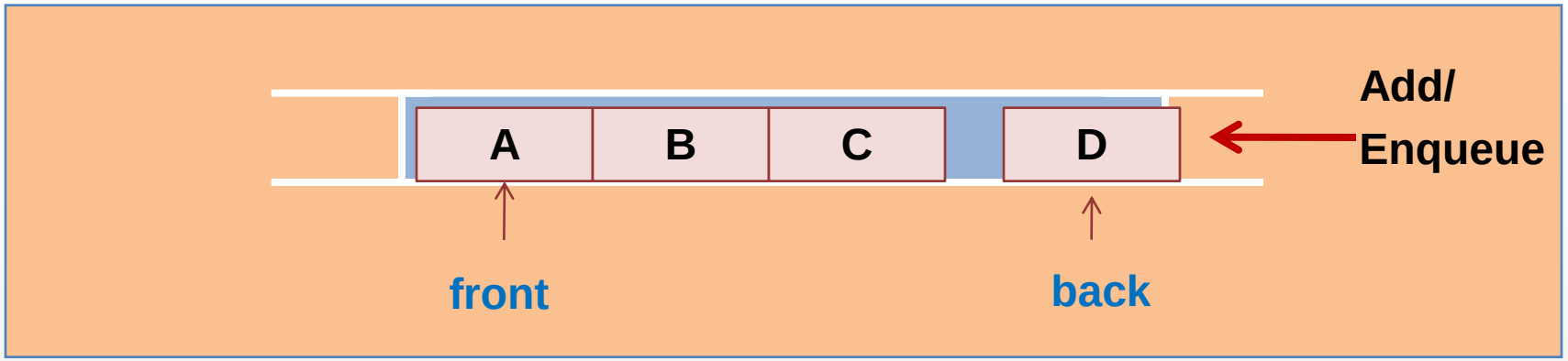
Queue Applications

Real-World Applications	Cashier lines in any store	Check out at a bookstore	Bank / ATM	Call an airline
Computer Science Applications	Print lines of a document	Printer sharing between computers	Recognizing palindromes	Shared resource usage
Simulation	To reduce the wait involved in an application			

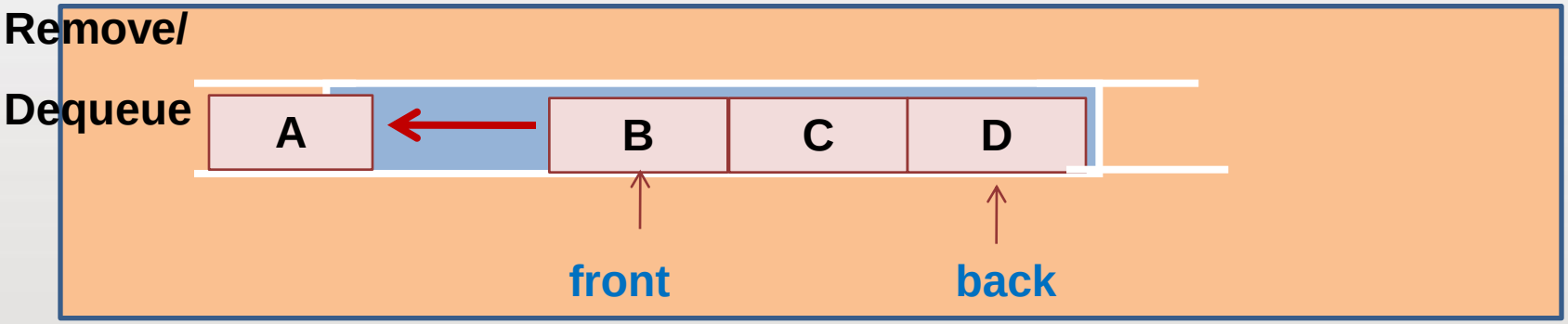
Queue Implementation



Queue Implementation

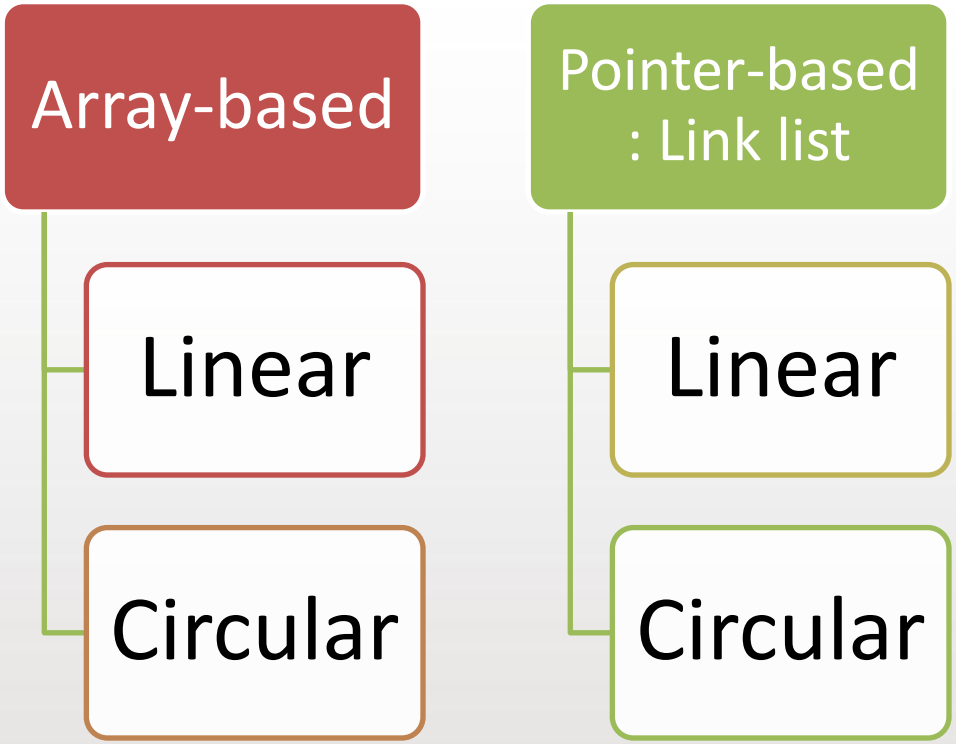


Insert D into Queue (enQueue) : D is inserted at rear



Delete from Queue (deQueue) : A is removed

Queue Implementation



Queue: Linear Array Implementation

```
class Queue
{ private:
     int front; // index at front
     int back; // index at rear queue
     char items[size]; //store item in Q
  public:
    Queue(); // Constructor - create Q
   ~Queue(); // Destructor - destroy Q
    bool isEmpty(); // check Q empty
    bool isFull(); // check Q full
    void enQueue(char); // insert into Q
    void deQueue(); // remove item from Q
    char getFront(); // get item at Front
    char getRear(); // get item at back Q
};
```

Queue Declaration

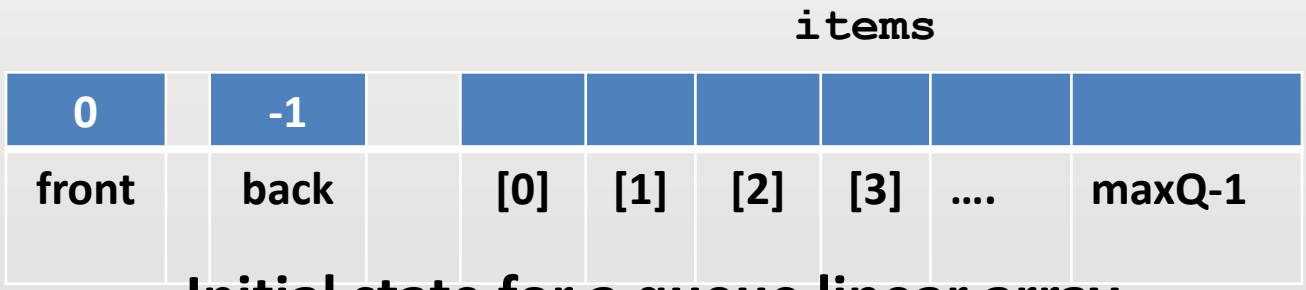
Queue
front *rear* *items*
createQueue() *destroyQueue()* *isEmpty();* *isFull();* *enQueue();* *deQueue();* *getFront();* *getRear();*

Abstract Data Type

CreateQueue() operation

- Linear Array implementation
- Constructor:
  - front and back are indexes in the array
  - Initial condition: front =0 and back = -1

```
Queue::Queue()
{ front = 0;
  back = -1;
}
```



Initial state for a queue linear array

Queue Operations

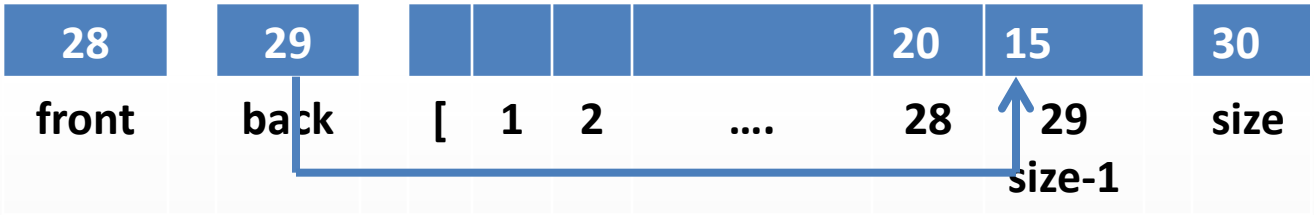
- **Destroy** Queue destructor : All elements in the queue will be disposed.

```
queue::~queue()
{ delete [ ] items; }
```

- **Check** whether a queue is **empty**
  - Queue Empty Condition : back < front

```
bool queue::isEmpty()
{ return bool(back < front); }
```

Queue Operations



- **Check** whether a queue is **Full**
  - Queue Full Condition : back = size -1

```
bool queue::isFull()
{ return bool(back == size - 1); }
```

  - Cannot insert any more item into a queue, when the queue is full.

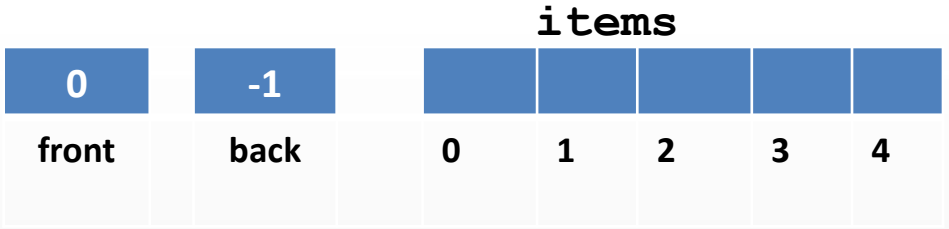
Queue Operations

- Insert into a queue (enQueue)
  - Increment back
  - Insert item in `items[ back ]`

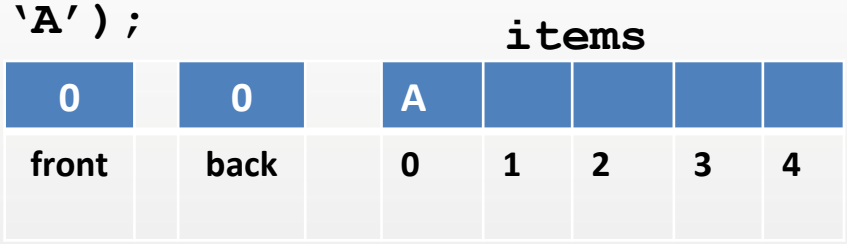
```
void queue::enQueue(char insertItem)
{ if (isFull())
     cout<< "\nCannot Insert. Queue is full!";
  else
  { //insert at back
    back++;
    items[back] = insertItem;
  } // end else if
}
```

enQueue() operations for a queue with size = 5

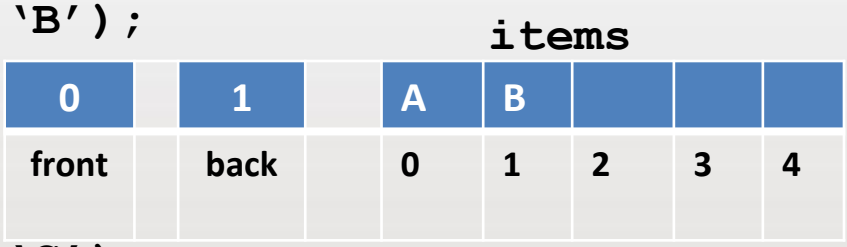
```
Queue myQueue;
```



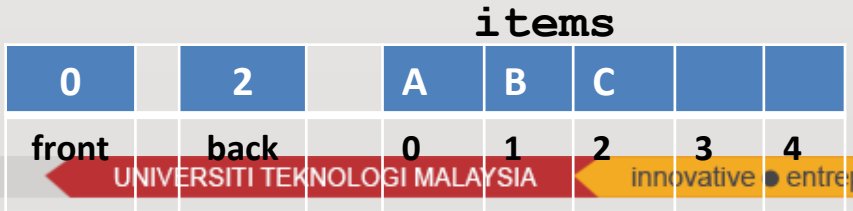

```
myQueue.enQueue('A');
```



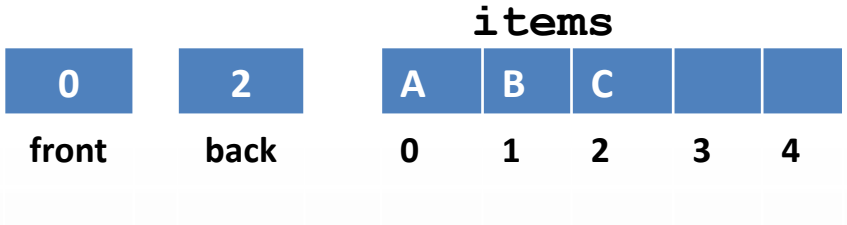

```
myQueue.enQueue('B');
```




```
myQueue.enQueue('C');
```



Queue operations



- Item at front and back can be retrieved

```
char queue:: getFront() // get item at Front
{ return items[front] ; }
```

```
char queue::getRear() // get item at Back
{ return items[back] ; }
```

```
cout << myQueue.getFront(); //output is A
cout << myQueue.getRear(); // output is C
```

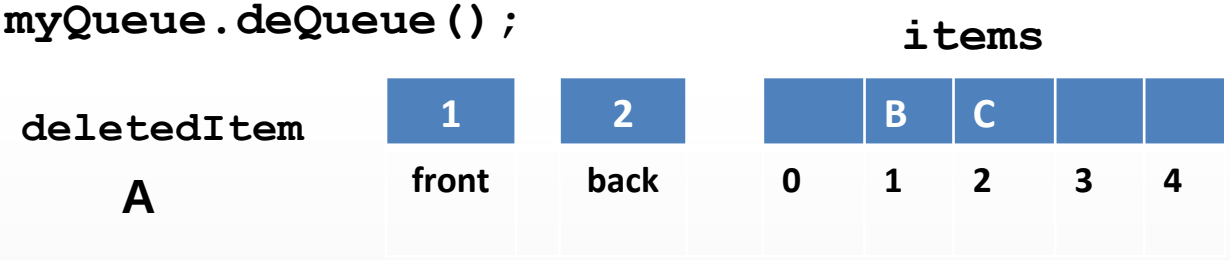
Queue operations

- Delete from a queue (deQueue)
  - Increment front

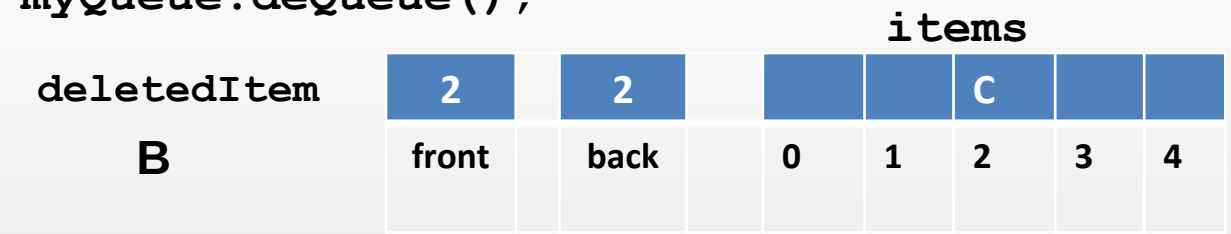
```
void queue::deQueue()
{ if (isEmpty())
    cout<< "\nCannot remove item. Empty Queue!";
  else
  { //retrieve item at front
    deletedItem = items[front];
    front++;
  } // end else if
}
```

deQueue() operations

```
myQueue.deQueue();
```




```
myQueue.deQueue();
```




```
myQueue.deQueue();
```


```
myQueue.deQueue();
```

Cannot remove item.
Queue is Empty with `back < front`

Queue operations – enQueue()

```
myQueue.enQueue('D');
```

items

3	3				D	
front	back	0	1	2	3	4

```
myQueue.enQueue('E');
```

items

3	4				D	E
front	back	0	1	2	3	4

```
myQueue.enQueue('F');
```

items

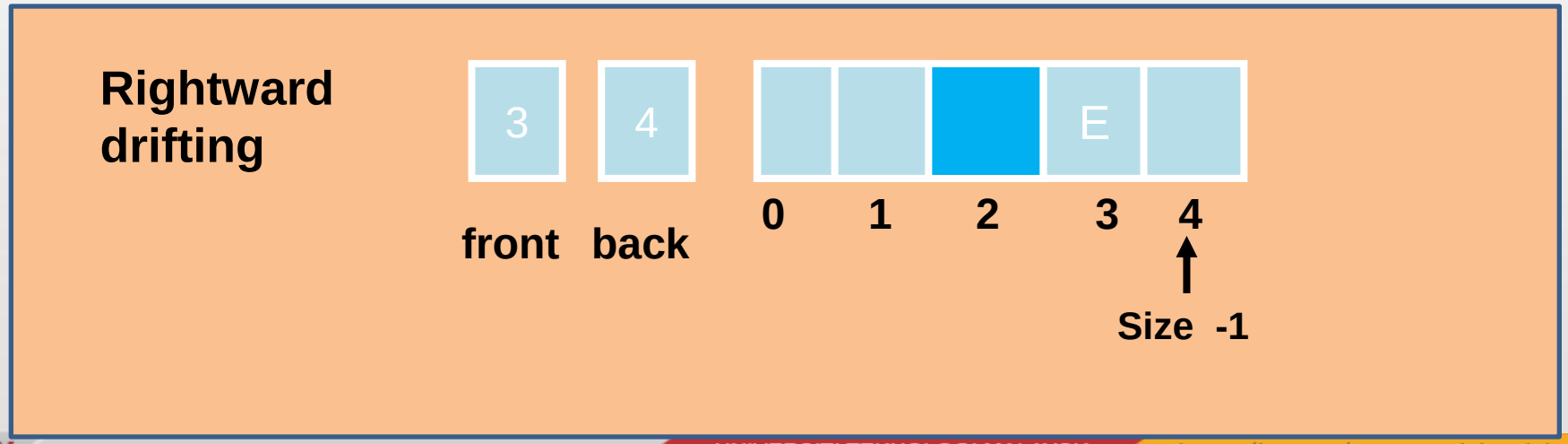
3	4				D	E
front	back	0	1	2	3	4

Cannot insert `F`, even though there are empty spaces in front of the queue array.
Currently, Queue is `FULL` with `back == size - 1`.

Linear Array Implementation - Drawback

Problem: Rightward-Drifting:

- After a sequence of additions and removals, items will drift towards the end of the array
- Even though, there are empty spaces in front of the queue array, `enQueue` operation cannot be performed on the queue, since `back = size - 1`.



Rightward Drifting Solutions

To optimize space and to solve rightward drifting:

1. Shift array elements after each deletion.

```
myQueue.deQueue();
```

deletedItem

A

items

1	2		B	C		
front	back	0	1	2	3	4

Shift array elements to front array

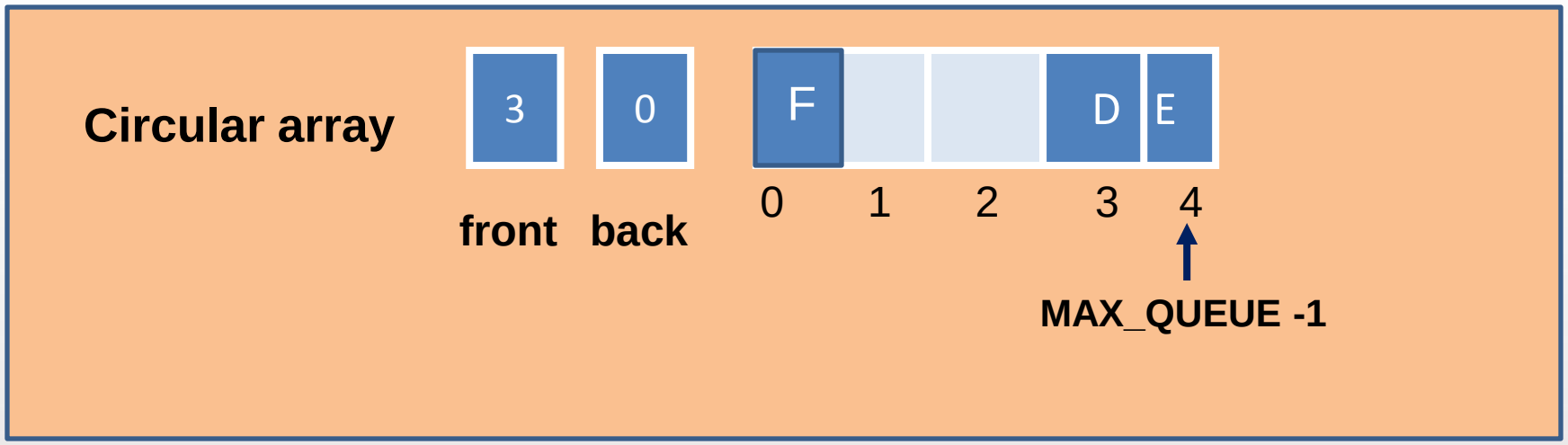
items

0	1	B	C			
front	back	0	1	2	3	4

However, shifting is not effective and dominates the cost of the implementation.

Rightward Drifting Solutions

2. Use a circular array: When front or back reach the end of the array, wrap them around to the beginning of the array.



In the figure, to insert F in the queue, F will be inserted at the front queue and restart again at index 0.

Queue Circular Array

- Problem:
  - front and back no longer can be used as condition to distinguish between queue-full and queue-empty

- Solution:
  - Use a counter, named count
  - count == 0 means empty queue
  - count == MAX_QUEUE means full queue

- Disadvantage
  - Overhead of maintaining a counter or flag

Circular Array Implementation

- Queue declarations

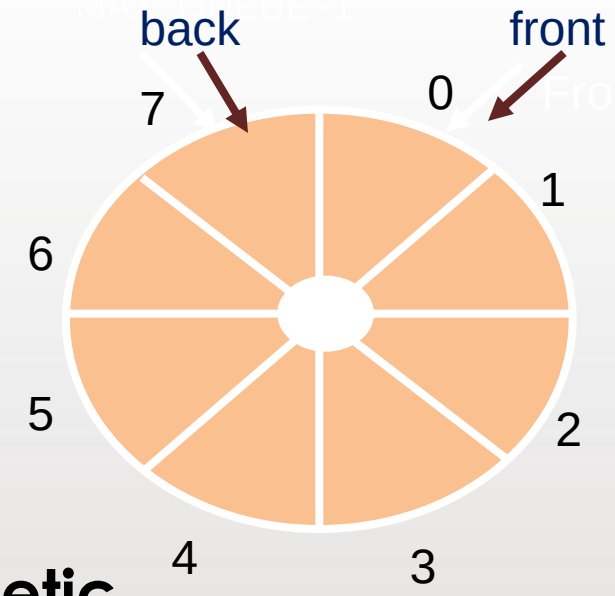
```
const int MAX_QUEUE = maximum-size-of-queue;
QueueItemType items [MAX_QUEUE];
int   front;
int   back;
int   count
```

- Initial condition:

```
count = 0, front = 0,
back = MAX_QUEUE - 1
```

- The Wrap-around effect is obtained by using modulo arithmetic (%-operator)

MAX_QUEUE = 8
count = 0

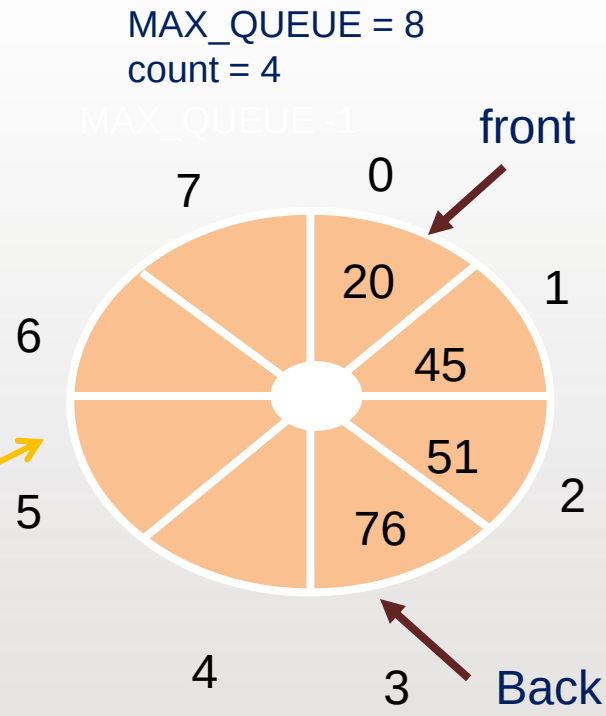


Empty circular queue.

Circular Arrays Implementation

- **Insertion**
  - Increment *back*, using modulo arithmetic
  - Insert item
  - Increment *count*

```
back = ( back + 1 ) % MAX_QUEUE;
items[back] = newItem;
++count;
```



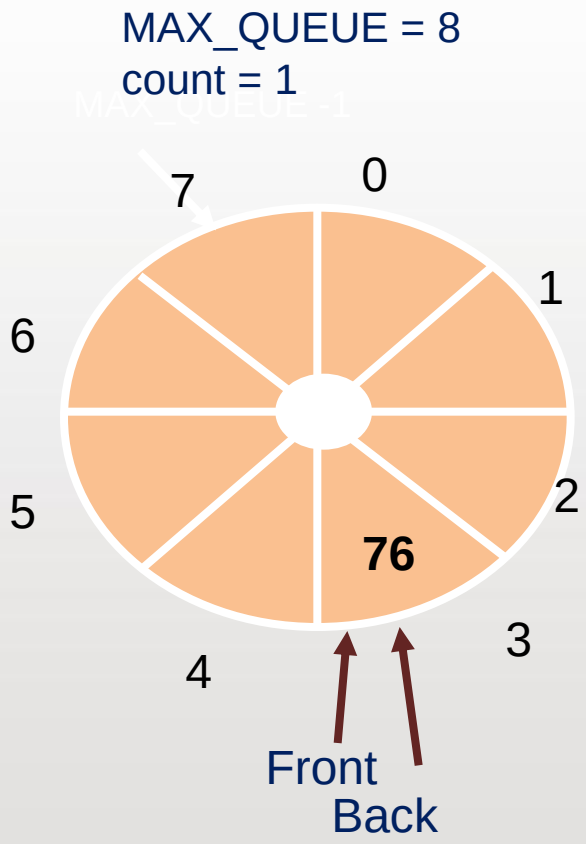
After insert 20, 45, 51 and 76 sequentially into circular queue

Circular Arrays Implementation

- **Deletion**
  - **Increment `front` using modulo arithmetic**
  - **Decrement *count***

```
front = ( front + 1 ) % MAX_QUEUE;
--count;
```

After delete 20, 45 and 51 sequentially from circular queue



Summary and Conclusion

- Queue can be implemented using linear array and circular array.
- Structure of queue linear array is the items that hold the array, front and back. Insertion happens at back, while deletion happens at front.
- Drawbacks of queue linear array is that it will lead to rightward drifting problem after sequence of deletion and insertion is performed on the queue.
- Queue circular array can be perform in order to solve the problem, whereby after front or back reach the end of the array, it will wrap around to the beginning of the array.


Thank
You