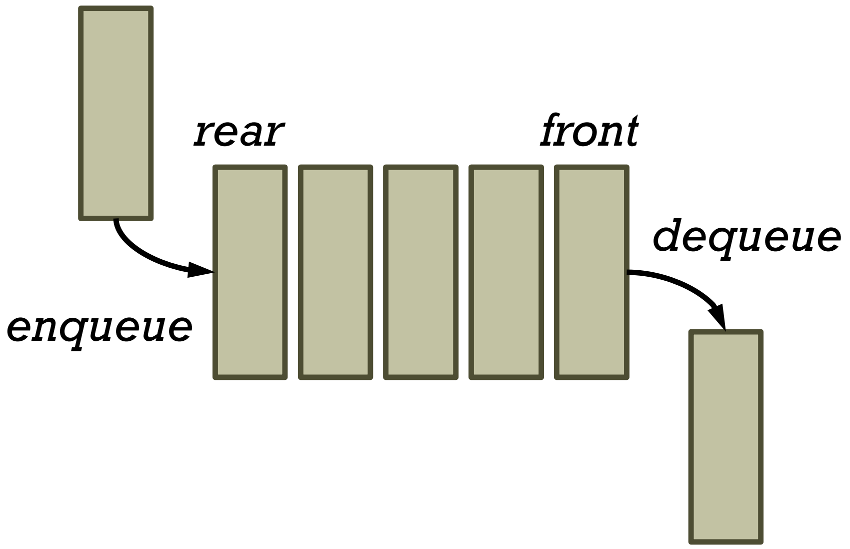
Queue

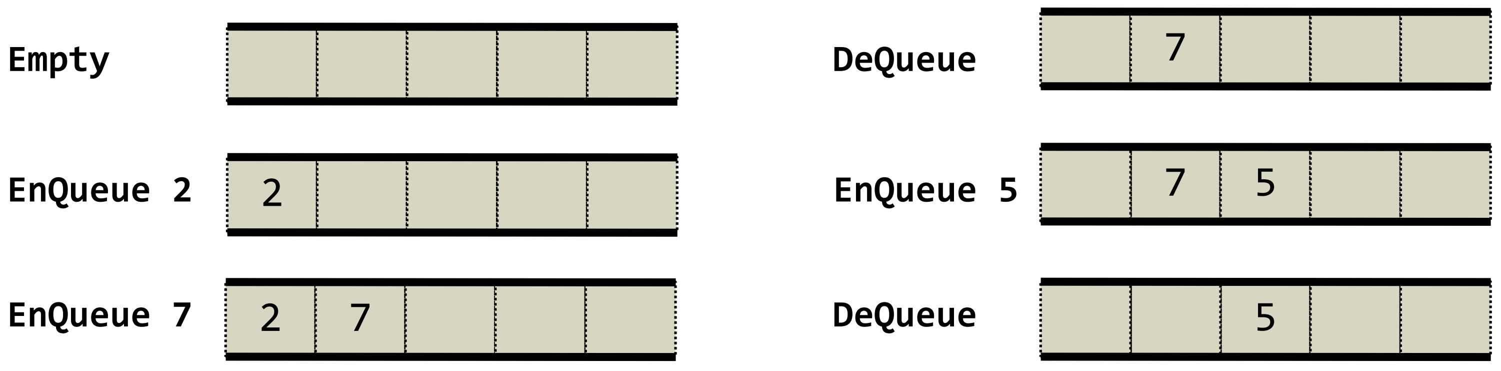
What is Queue?

- Definition
  - A collection of elements that are inserted and removed according to the **first-in first-out (FIFO) principle**.
    - The first element added to the queue will be the first one to be removed.
    - All insertions are made at one end, called **rear**.
    - All deletions are made at the other end, called **front**.



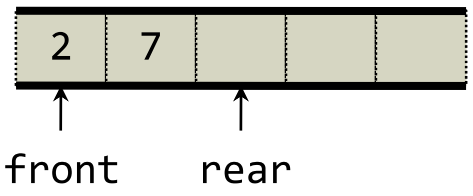
What is Queue?

- Terminology
  - **Front** : The front of queue, where deletions take place or the position of the first item
  - **Rear** : The rear of queue, where insertions take place or the next of the last item
  - **EnQueue**: Insert an item at the rear.
  - **DeQueue**: Delete the item at the front.



What is Queue?

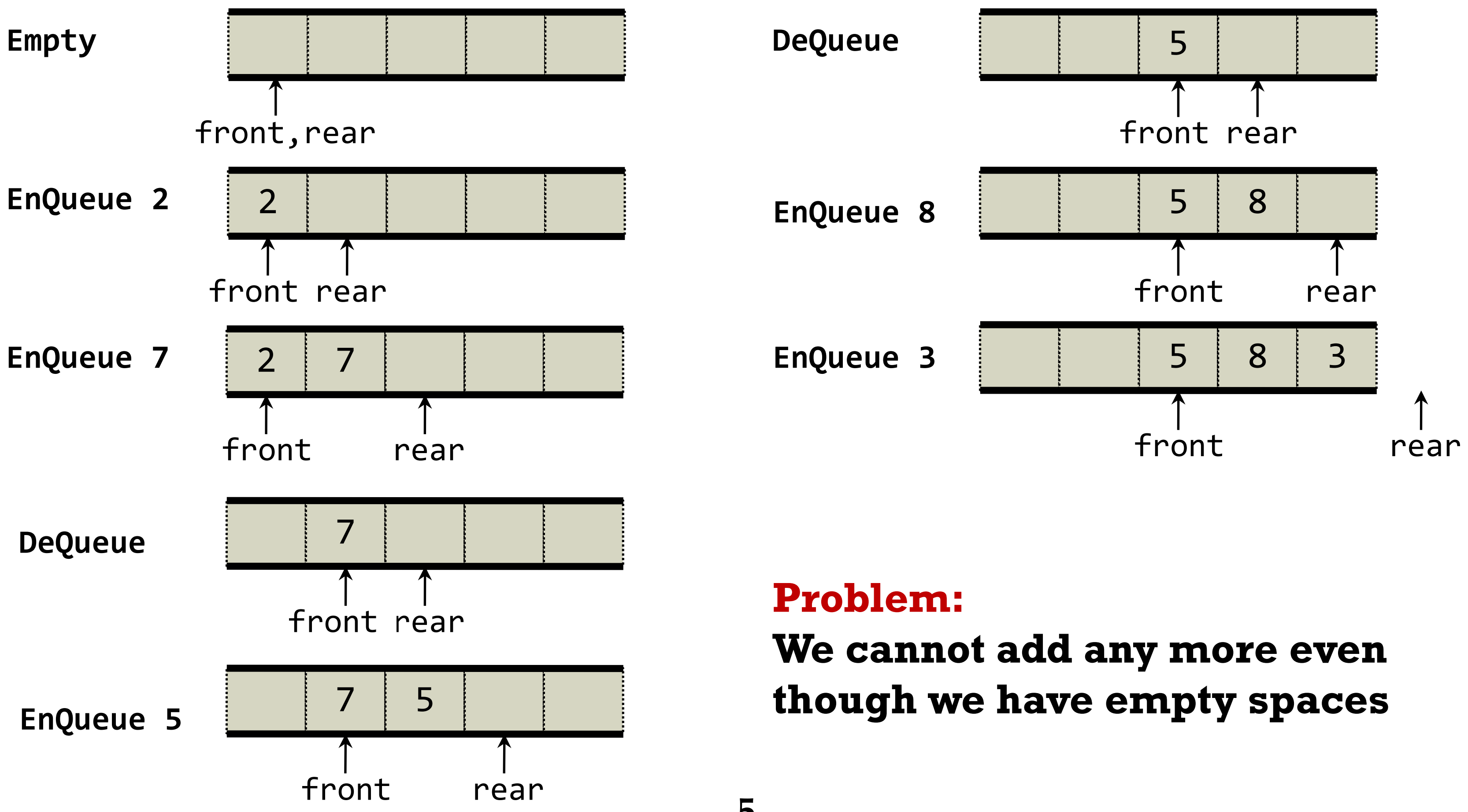
- Operations
  - **InitQueue**: Make queue empty.
  - **IsFull**: Check whether queue is full.
  - **IsEmpty**: Check whether queue is empty.
  - **Peek** : Read the item at the front.
  - **EnQueue**: Insert an item at the rear.
  - **DeQueue**: Remove an item at the front.



Q: Can we access items other than at the front?
A: By definition, No

Linear Queue

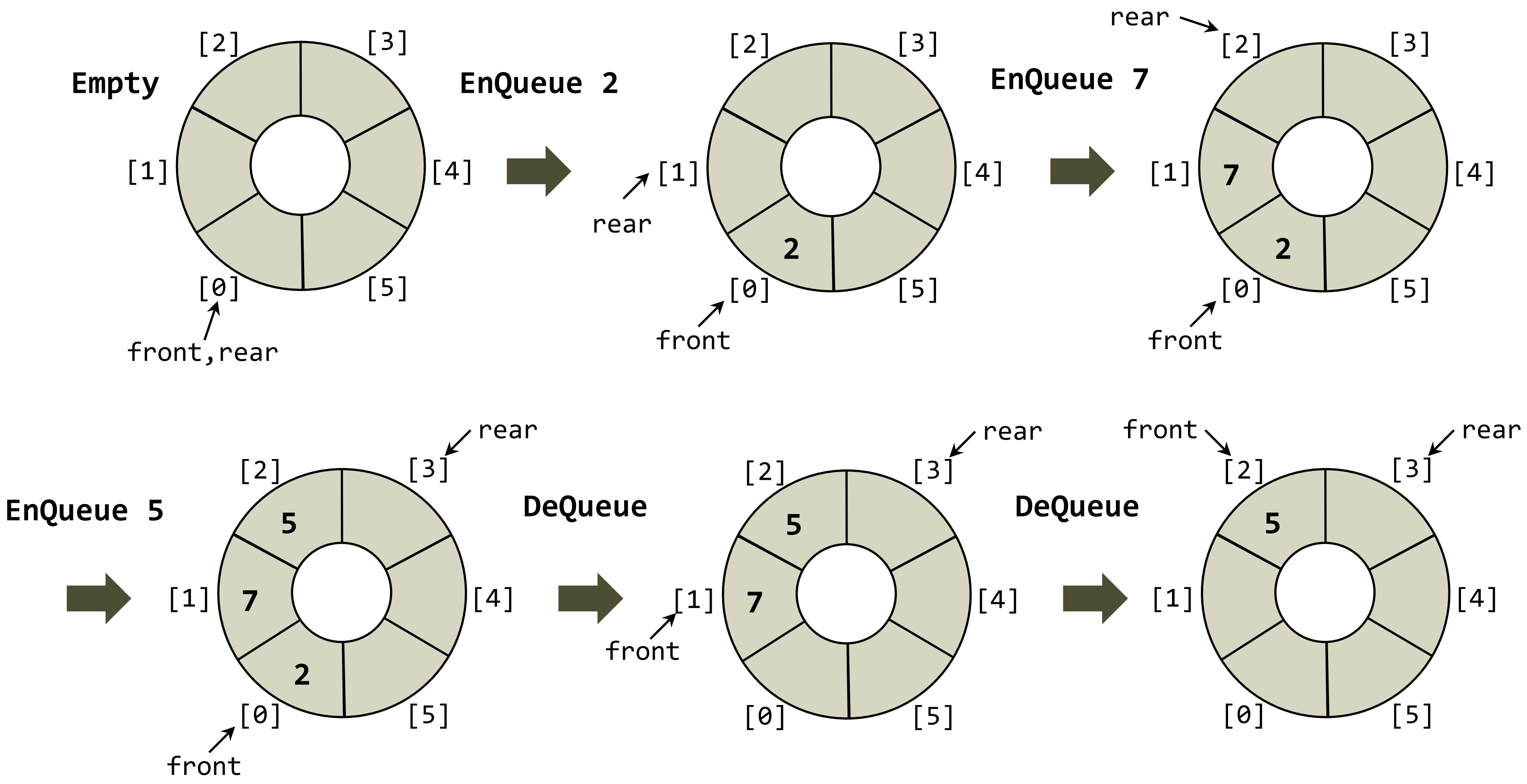
- It performs deletion at one end of the list and the insertion at the other end.



Problem:
We cannot add any more even though we have empty spaces

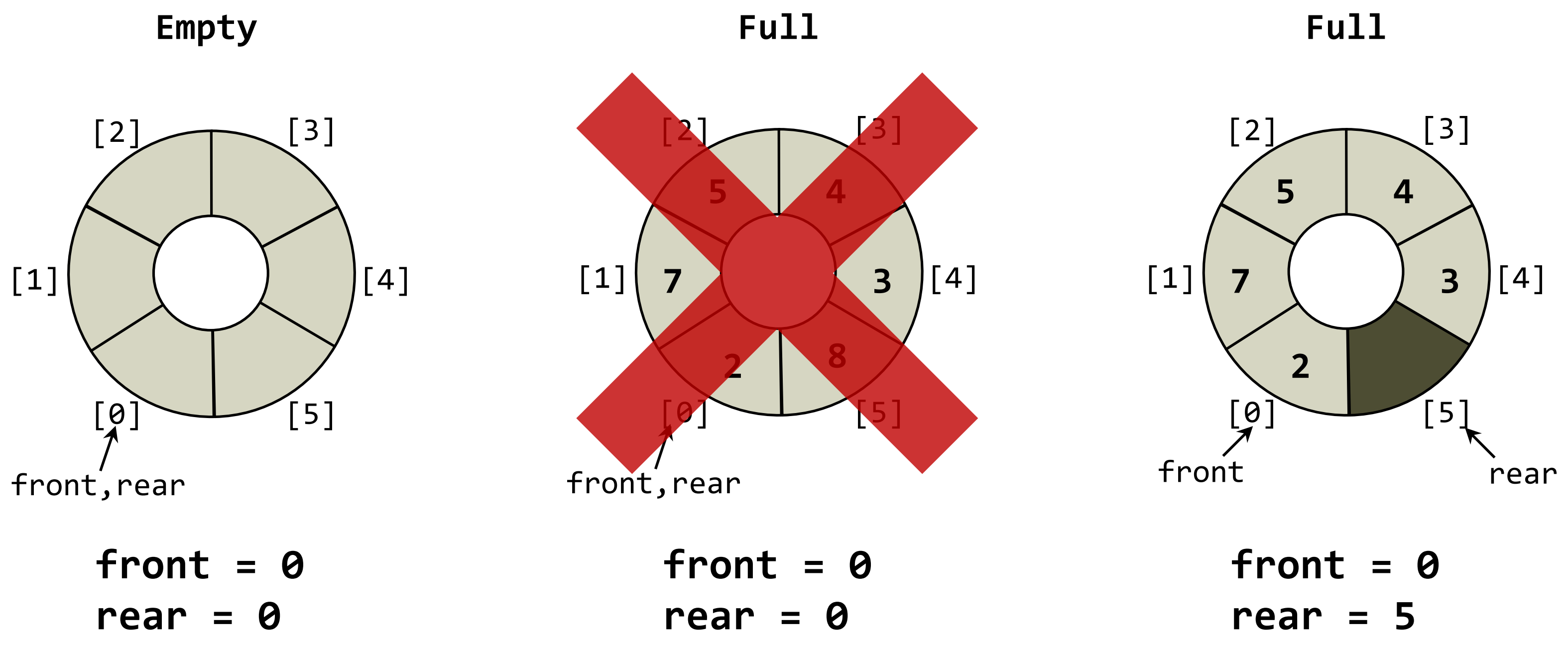
Circular Queue

- The last position is connected back to the first position to make a circle.



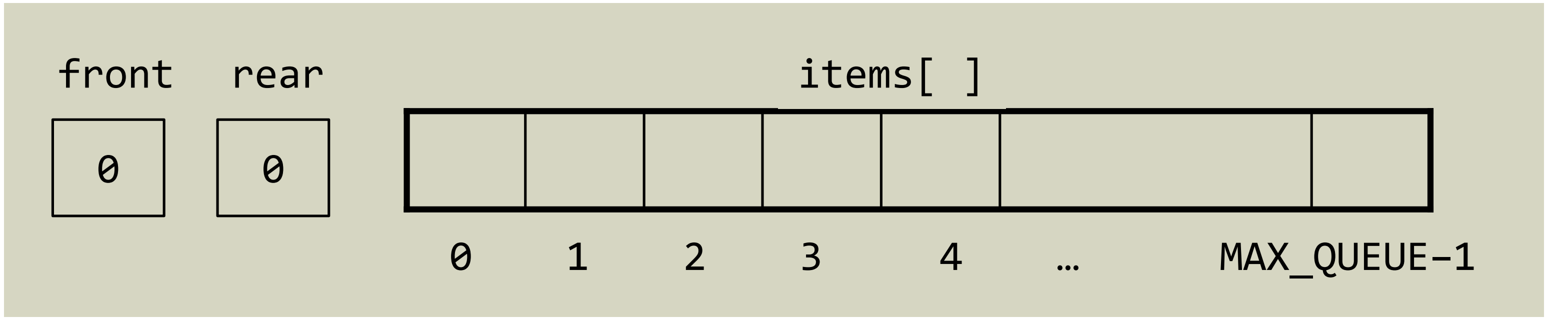
Circular Queue

- How to distinguish Empty and Full?
  - Full: check whether front is 0 and rear is MAX_SIZE – 1.



Array-based Implementation

- Queue representation



```
#define MAX_QUEUE    100

typedef enum { false, true } bool;
typedef int Data;

typedef struct {
    int front, rear;
    Data items[MAX_QUEUE];
} Queue;
```

Array-based Implementation

- Operations

```
// Make a queue empty.
void InitQueue(Queue *pqueue);
// Check whether a queue is full.
bool IsFull(Queue *pqueue);
// Check whether a queue is empty.
bool IsEmpty(Queue *pqueue);

// Read the item at the front.
Data Peek(Queue *pqueue);
// Insert an item at the rear.
void EnQueue(Queue *pqueue, Data item);
// Delete an item at the front.
void DeQueue(Queue *pqueue);
```

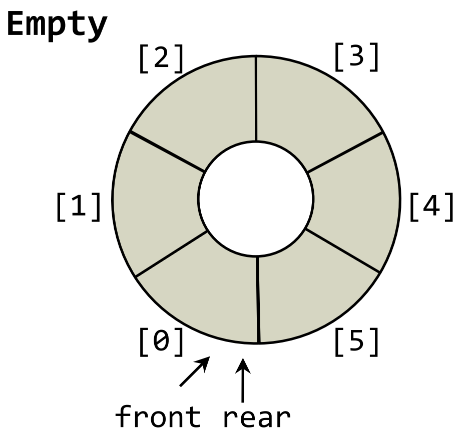
design alternative

```
Data Peek(Queue *pqueue);
// Insert an item at the rear.
void EnQueue(Queue *pqueue, Data item);
// Delete an item at the front.
void DeQueue(Queue *pqueue);
```

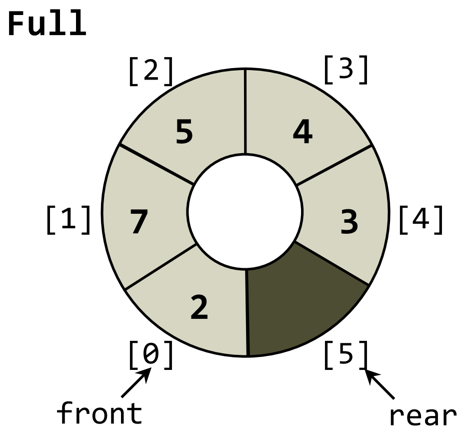
Array-based Implementation

- InitQueue and IsFull operations

```
// Make a queue empty.
void InitQueue(Queue *pqueue)
{
    pqueue->front = pqueue->rear = 0;
}
```



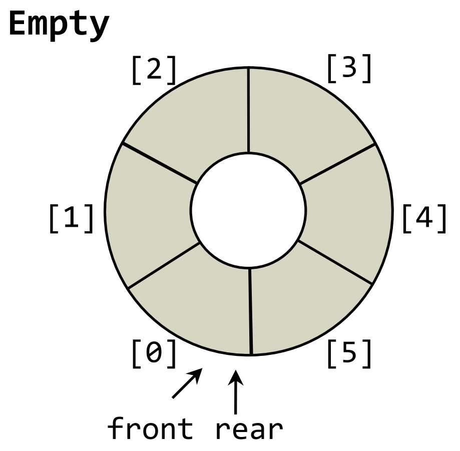
```
// Check whether a queue is full.
bool IsFull(Queue *pqueue)
{
    return pqueue->front
        == (pqueue->rear + 1) % MAX_QUEUE;
}
```



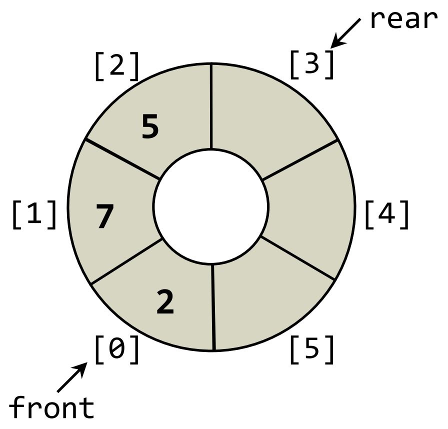
Array-based Implementation

- IsEmpty and Peek operations

```c
// Check whether a queue is empty.
bool IsEmpty(Queue *pqueue)
{
    return pqueue->front == pqueue->rear;
}
```

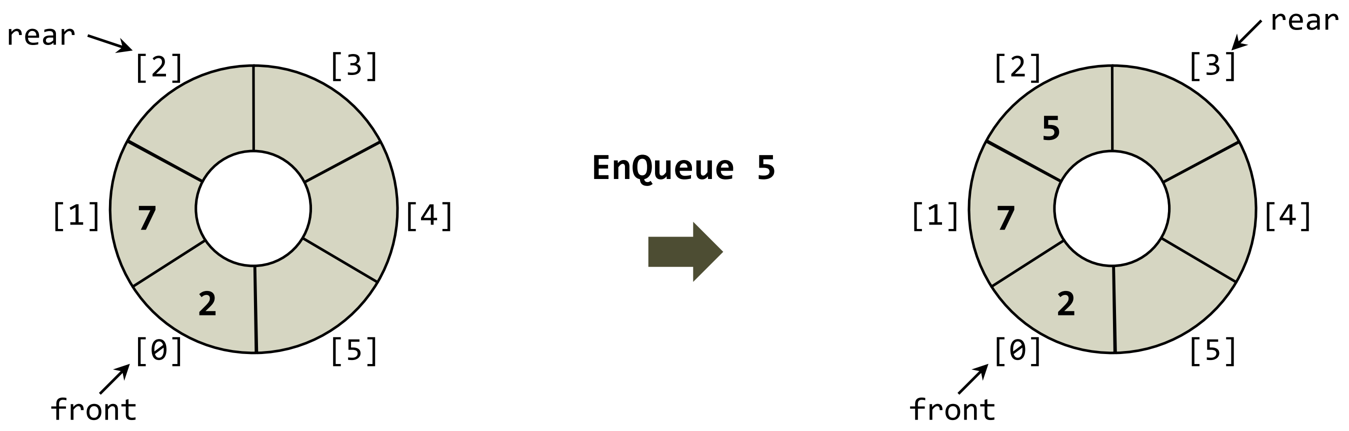


```c
// Read the item at the front.
Data Peek(Queue *pqueue)
{
    if (IsEmpty(pqueue))
        exit(1); //error: empty stack
    return pqueue->items[pqueue->front];
}
```



Array-based Implementation

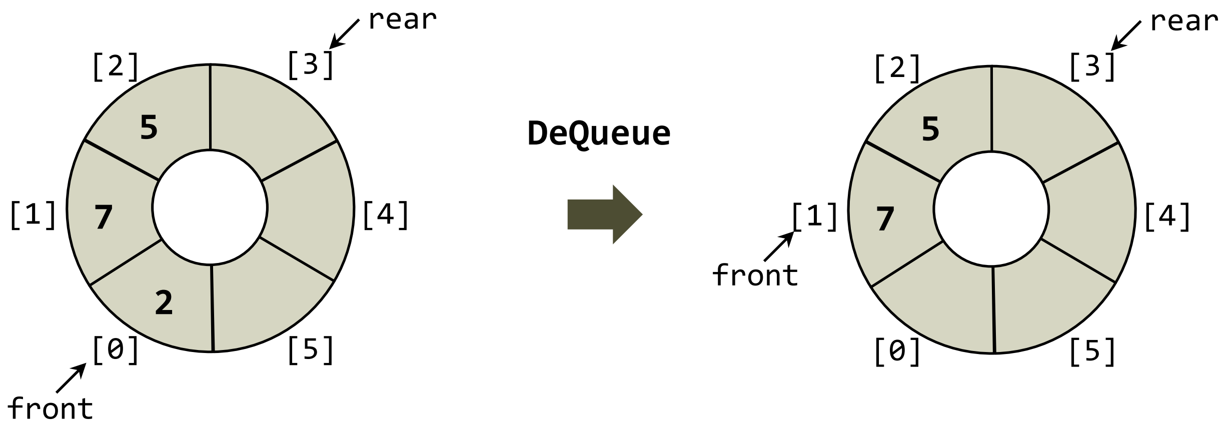
- Enqueue operation



```
// Insert an item at the rear.
void EnQueue(Queue *pqueue, Data item)
{
    if (IsFull(pqueue))
        exit(1); //error: stack full
    pqueue->items[pqueue->rear] = item;
    pqueue->rear = (pqueue->rear + 1) % MAX_QUEUE;
}
```

Array-based Implementation

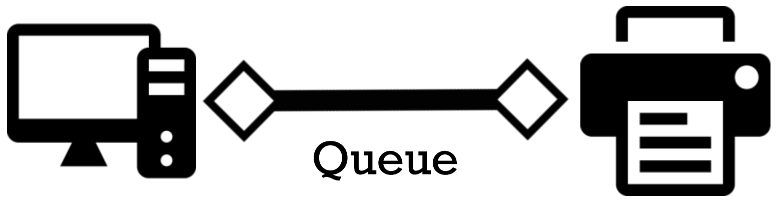
- Dequeue operation

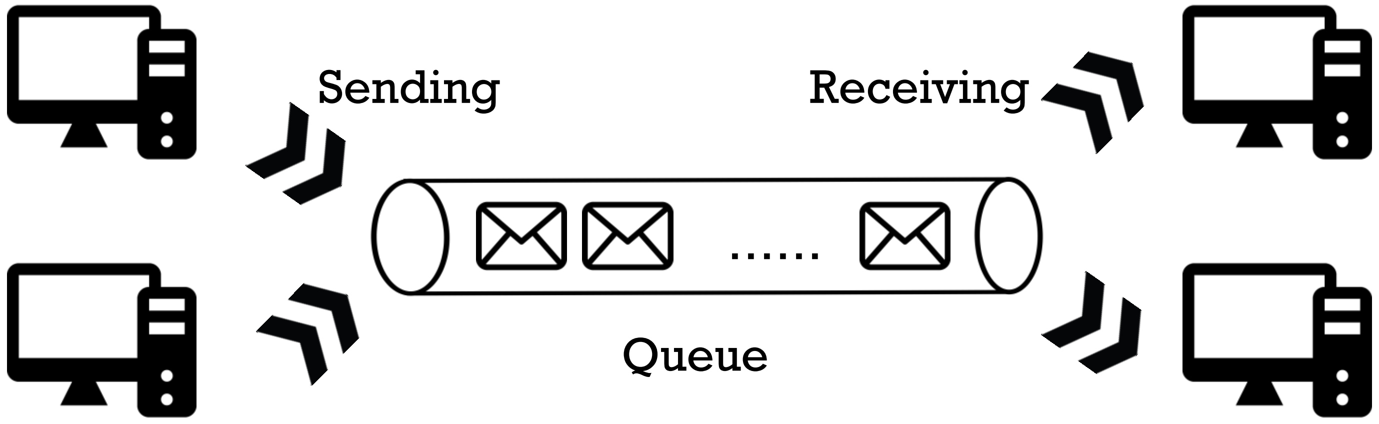


```
// Delete an item at the front.
void DeQueue(Queue *pqueue)
{
    if (IsEmpty(pqueue))
        exit(1); //error: empty stack
    pqueue->front = (pqueue->front + 1) % MAX_QUEUE;
}
```

Buffer Management

- The queue is used to implement a buffer that connects two processes.
  - A buffer is a memory storage used to temporarily store data while it is moved from one place to another.





Naïve Buffer

```c
// Insert an item at the rear.
void EnQueue(Queue *pqueue, Data item)
{
     if (IsFull(pqueue))
          exit(1); //error: queu is full
     pqueue->items[pqueue->rear] = item;
     pqueue->rear = (pqueue->rear + 1) % MAX_QUEUE;
}
```

```c
// Delete an item at the front.
void DeQueue(Queue *pqueue)
{
     if (IsEmpty(pqueue))
          exit(1); //error: empty stack
     pqueue->front = (pqueue->front + 1) % MAX_QUEUE;
}
```

Buffer Management

```
// Example of a producer process
void Producer(Queue* buffer, Data data)
{
     if (lock(buffer) == false) {
          if (!IsFull(buffer)) {
               // Append the data to the buffer.
               EnQueue(buffer, data);
          }
          unlock(buffer);
     }
}

// Example of a consumer process
void Consumer(Queue* buffer, Data data)
{
     if (lock(buffer) == false) {
          if (!IsEmpty(buffer)) {
               Data data = Peek(buffer);
               DeQueue(buffer);
               // Consume the data.
               // ....
          }
          unlock(buffer);
     }
}
```

Is this correct?

Bank Simulation

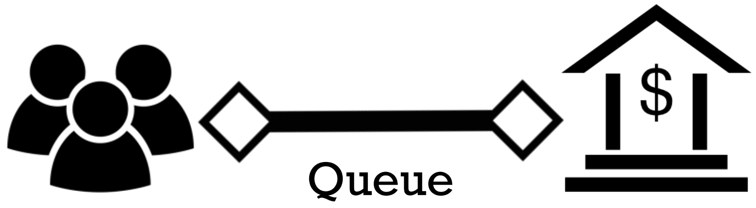
- Each customer has id, arrival time, and service time.
  - Customers have arrived sequentially.
  - Customers wait until previous customers finish their services.

```c
#include "queue.h"
#include <stdlib.h>
#include <stdio.h>

#define MAX_SERV_TIME   10

typedef struct
{
	int id;
	int arrival_time;
	int service_time;
} Customer;

int waited_time = 0;
int served_customers = 0;
int num_customers = 0;
```



Results of Bank Simulation

```
Time = 1
Time = 2
Customer  1 enters. service time: 5 mins
Customer  1:  5 mins service starts. waiting time:  0 mins
Time = 3
Customer  2 enters. service time: 10 mins
Time = 4
Time = 5
Time = 6
Customer  3 enters. service time: 3 mins
Time = 7
Customer  4 enters. service time: 6 mins
Time = 8
Customer  5 enters. service time: 2 mins
Customer  2: 10 mins service starts. waiting time:  5 mins
Time = 9
Customer  6 enters. service time: 2 mins
Time = 10

Total waiting time = 5 mins
Average waiting time per customer = 2.50 mins
Number of served customer = 2
Number of waiting customers = 4
```

Implementing Bank Simulation

```c
int main()
{
     int service_time = 0, duration = 10;
     int clock = 0, id = 1;
     Queue queue;

     InitQueue(&queue);
     while (clock < duration){
          clock++;
          printf("Time = %d\n", clock);
          if (IsCustomerArrived())
               // Insert a customer in a sequential manner.
               InsertCustomer(&queue, id++, clock);

          // Remove a customer in a sequential manner.
          if (service_time > 0) service_time--;
          else service_time = RemoveCustomer(&queue, clock);
     }

     PrintStat(); // Print statistics on customers.
     return 0;
}
```

Implementing Bank Simulation

- InsertCustomer function
  - A customer enters unless the queue is full.

```
void InsertCustomer(Queue* pqueue, int id, int clock)
{
     Customer c;
     int service_time = (int)(rand() % MAX_SERV_TIME) + 1;

     if (IsFull(pqueue))
          return;

     c.id = id;
     c.arrival_time = clock;
     c.service_time = service_time;

     EnQueue(pqueue, c);
     printf("Customer %2d enters. service time: %d mins\n", id,  service_time);
     num_customers++;
}
```

Is this correct?

Implementing Bank Simulation

- RemoveCustomer function
  - A customer's service starts. The other customers wait until the service is completed.

```
int RemoveCustomer(Queue* pqueue, int clock)
{
     Customer customer;
     int service_time = 0;

     if (IsEmpty(pqueue)) return 0;

     customer = Peek(pqueue);
     DeQueue(pqueue);

     service_time = customer.service_time;
    printf("Customer %2d: %2d mins service starts. waiting time: %2d mins\n",
     customer.id, service_time, clock - customer.arrival_time);

     served_customers++;
     waited_time += clock - customer.arrival_time;

     return service_time;
}
```

Implementing Bank Simulation

- IsCustomerArrived and PrintStat functions
  - IsCustomerArririved checks whether a customer is arrived or not.
  - PrintStat prints general statistics on bank service.

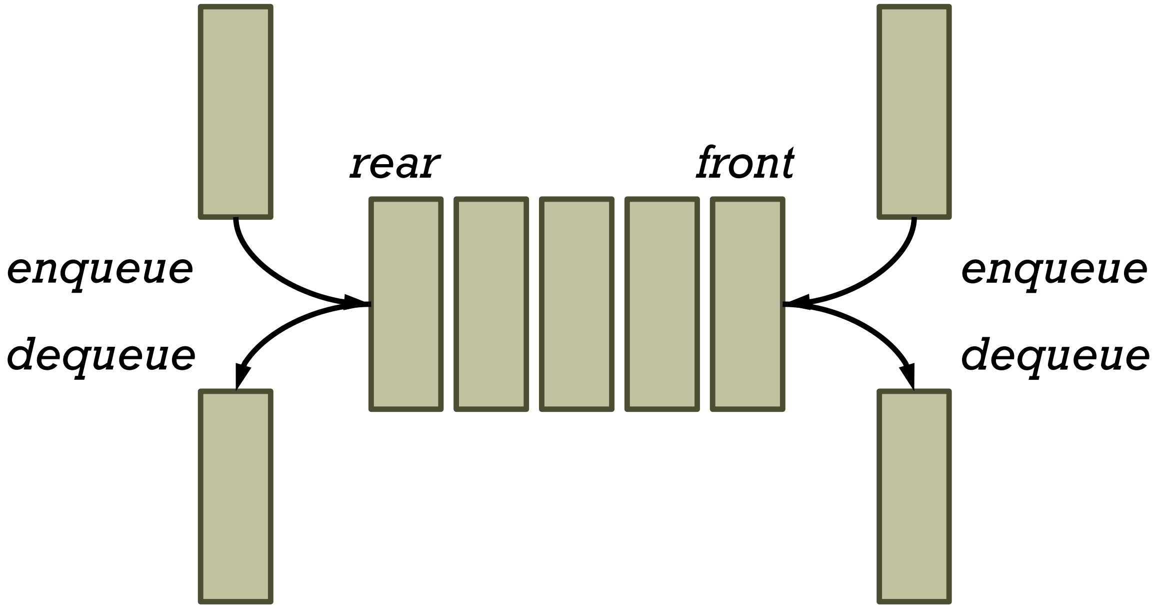
```
bool IsCustomerArrived()
{
	double prob = rand() / (double)RAND_MAX;
	if (prob >= 0.5) return true;
	else return false;
}
```

```
void PrintStat()
{
	printf("\nTotal waiting time = %d mins\n", waited_time);
	printf("Average waiting time per customer = %.2f mins\n",
			(double)waited_time / served_customers);

	printf("Number of served customer = %d\n", served_customers);
	printf("Number of waiting customers = %d\n",
			num_customers - served_customers);
}
```

What is DEQ?

- Definition
  - **Double-ended queue** that generalizes a queue
  - Elements are added to or removed from either the front or rear.
    - Enqueue for both front and rear.
    - Dequeue for both front end rear.



DEQ Operations

- Operations
  - **InitDeque** : make deque empty.
  - **IsFull** : check whether deque is full.
  - **IsEmpty** : check whether deque is empty.

  - **AddFront**: Insert an item at the front.
  - **AddRear**: Insert an item at the rear.
  - **RemoveFront**: Delete an item at the front.
  - **RemoveRear**: Delete an item at the rear.
  - **PeekFront** : Read the item at the front.
  - **PeekRear** : Read the item at the rear.

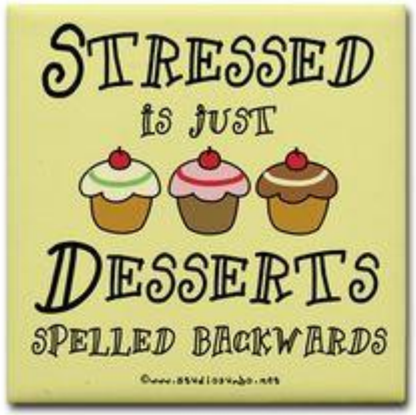
Palindrome Checker

- A palindrome is a sequence of characters that reads the same backward as forward.

```
bool checkPalindrome(char * str, int len)
{
     Deque deq;

     InitDeque(&deq);
     for (int i = 0; i < len; i++)
          AddRear(&deq, str[i]);

     while (len > 1)
     {
          if (PeekFront(&deq) == PeekRear(&deq))
          {
               RemoveFront(&deq), RemoveRear(&deq);
               len = len - 2;
          }
          else
               return false;
     }
     return true;
}
```



Palindrome Checker

- Checking a palindrome using stack and queue

```
bool checkPalindrome(char* str, int len)
{
     Stack stack;
     Queue queue;

     InitStack(&stack);
     InitQueue(&queue);
     for (int i = 0; i < len; i++) {
          Push(&stack, str[i]);
          EnQueue(&queue, str[i]);
     }

     while (!IsEmpty(&queue)) {
          if (Peek(&stack) == Peek(&queue)) {
               Pop(&stack);
               DeQueue(&queue);
          }
          else
               return false;
     }
     return true;
}
```