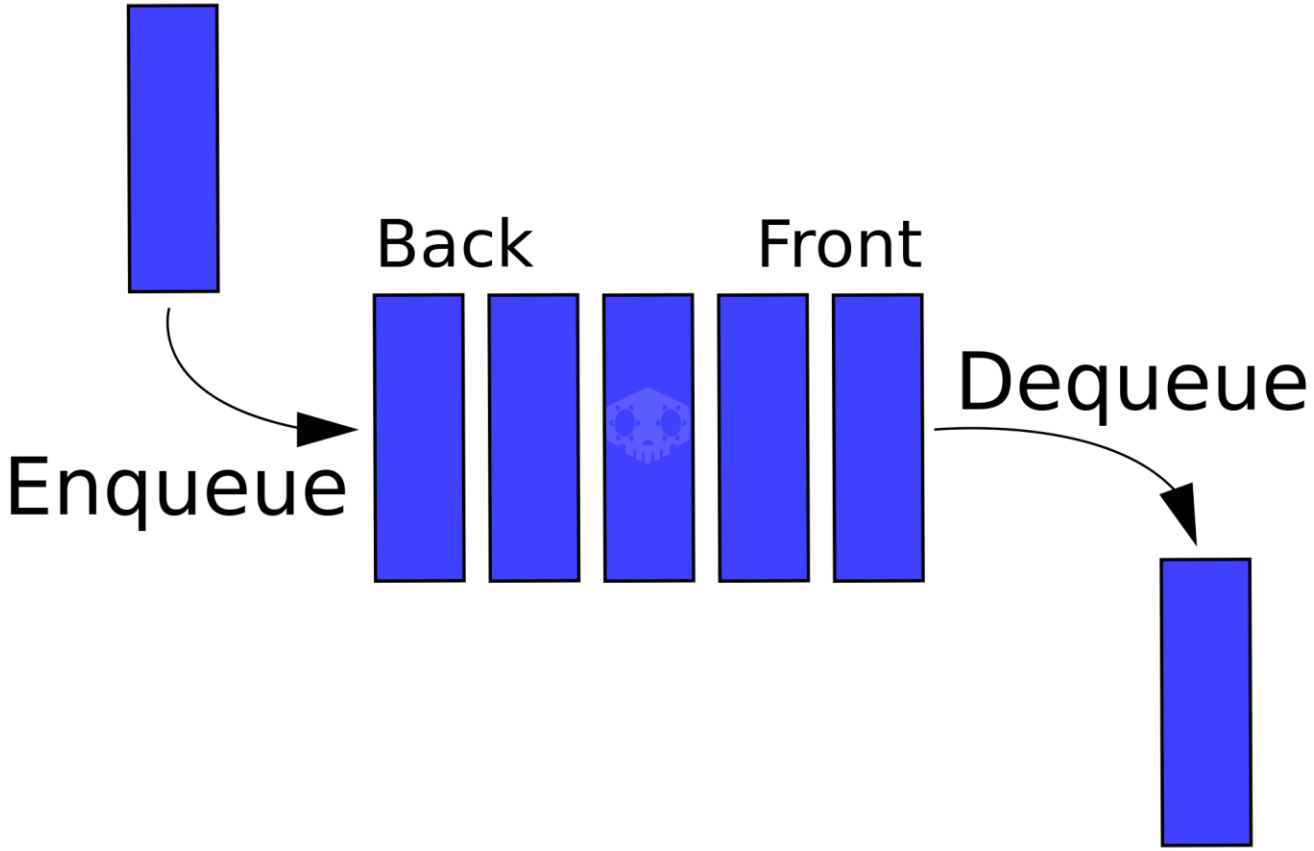
# Chapter 11:Queues



# Topics

11.1 Introduction to the Queue ADT

11.2 Dynamic Queues

11.3 The STL **`deque`** and **`queue`** Containers

11.4 Eliminating Recursion

# 11.4 Introduction to the Queue ADT

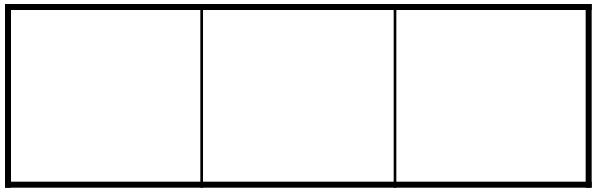
- Queue: a FIFO (first in, first out) data structure.
- Examples:
  - people waiting to use an ATM
  - cars lined up to pay and exit a parking structure
- Implementation:
  - static: fixed size, implemented as array
  - dynamic: variable size, implemented as linked list

# Queue Locations and Operations

- rear: position where elements are added
- front: position from which elements are removed
- enqueue: add an element to the rear of the queue
- dequeue: remove an element from the front of a queue

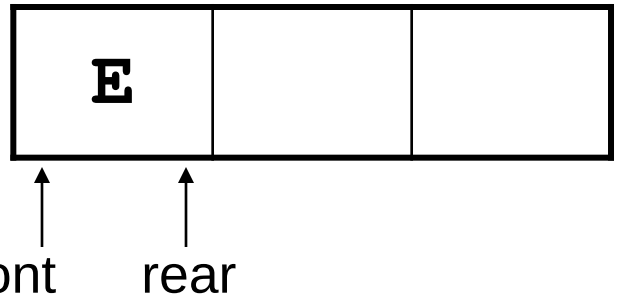
# Array Implementation of Queue

An empty queue that can hold **`char`** values:



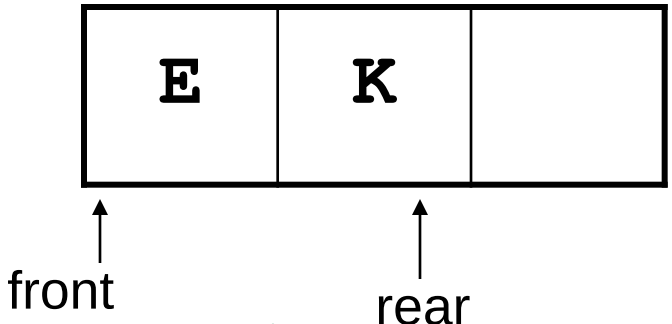
front, rear

```
enqueue('E');
```

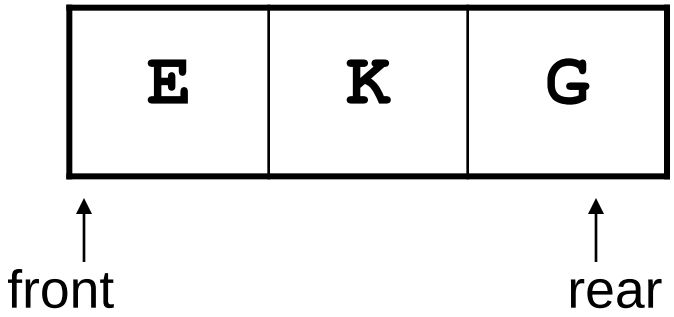


# Queue Operations - Example

```
enqueue('K');
```

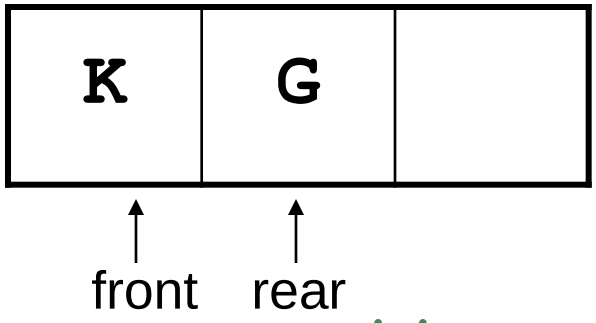



```
enqueue('G');
```

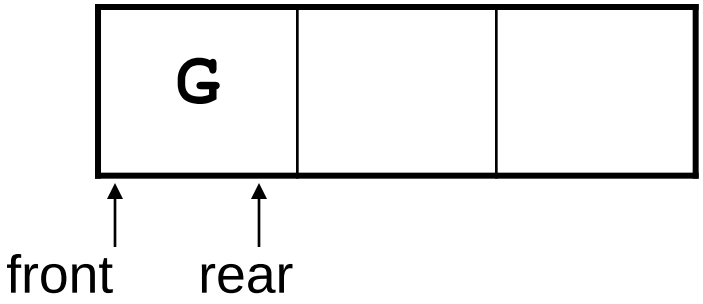


# Queue Operations - Example

```
dequeue(); // remove E
```




```
dequeue(); // remove K
```



# Array Implementation Issues

- In the preceding example, Front never moves.
- Whenever **`dequeue`** is called, all remaining queue entries move up one position. This takes time.
- Alternate approach:
  - Use a 'circular' array: `front` and `rear` both move when items are added and removed. Both can 'wrap around' from the end of the array to the front if warranted.
- Other solutions are possible

# Array Implementation Issues

- Variables needed
  - `int qSize;`
  - `char q[qSize];`
  - `int front = -1;`
  - `int rear = -1;`
  - `int number = 0; //how many in queue`
- You could make these members of a queue class, and queue operations would be member functions

# `isEmpty` Member Function

Check if queue is empty

```
bool isEmpty()
{
  if (number > 0)
    return false;
  else
    return true;
}
```

# `isFull` Member Function

Check if queue is full

```
bool isFull()
{
  if (number < qSize)
    return false;
  else
    return true;
}
```

# **enqueue** and **dequeue**

- To enqueue, we need to add an item **x** to the rear of the queue
- Queue convention says **q[rear]** is already occupied. Execute

```
if(!isFull)
{ rear = (rear + 1) % qSize;
// mod operator for wrap-around
  q[rear] = x;
   number ++;
}
```

# **enqueue** and **dequeue**

- To dequeue, we need to remove an item **x** from the front of the queue
- Queue convention says **q[front]** has already been removed. Execute

```
if(!isEmpty)
{  front = (front + 1) % qSize;
   x = q[front];
   number--;
}
```

# `enqueue` and `dequeue`

- `enqueue` moves `rear` to the right as it fills positions in the array
- `dequeue` moves `front` to the right as it empties positions in the array
- When `enqueue` gets to the end, it wraps around to the beginning to use those positions that have been emptied
- When `dequeue` gets to the end, it wraps around to the beginning use those positions that have been filled

# **enqueue** and **dequeue**

- Enqueue wraps around by executing

```
rear = (rear + 1) % qSize;
```

- Dequeue wraps around by executing

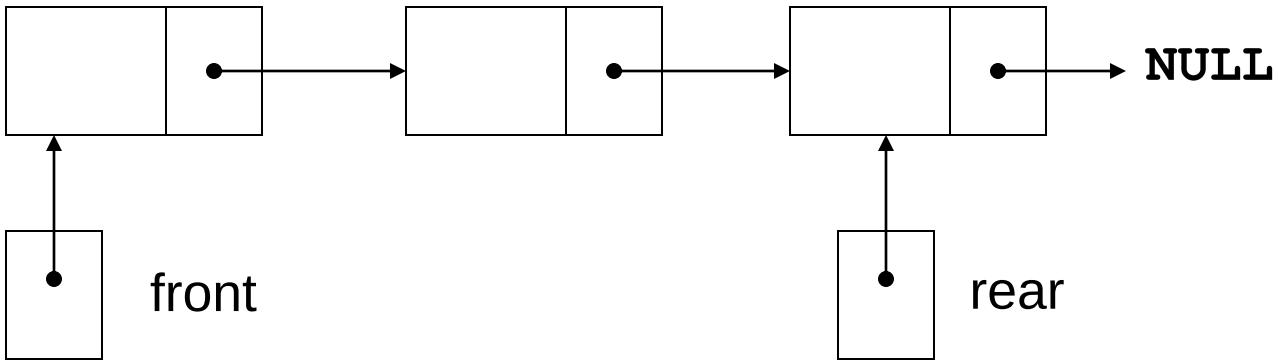
```
front = (front + 1) % qSize;
```

# Exception Handling in Static Queues

- As presented, the static queue class will encounter an error if an attempt is made to enqueue an element to a full queue, or to dequeue an element from an empty queue
- A better design is to throw an underflow or an overflow exception and allow the programmer to determine how to proceed
- Remember to throw exceptions from within a **`try`** block, and to follow the **`try`** block with a **`catch`** block

# 11.5 Dynamic Queues

- Like a stack, a queue can be implemented using a linked list
- This allows dynamic sizing and avoids the issue of wrapping indices



# Dynamic Queue Implementation Data Structures

- Define a class for the dynamic queue
- Within the dynamic queue, define a private member class for a dynamic node in the queue
- Define node pointers to the front and rear of the queue

# isEmpty Member Function

To check if queue is empty:

```
bool isEmpty()
{
  if (front == NULL)
     return true;
  else
     return false;
}
```

# enqueue Member Function Details

To add item at rear of queue

```
if (isEmpty())
     {
      front = new QNode(x);
       rear = front;
     }
  else
     {
       rear->next = new QNode(x);
       rear = rear->next;
     }
```

# dequeue Member Function

To remove item from front of queue

```
if (isEmpty())
{
     // throw exception or print
     // a message
} else {
x = front->value;
QNode *oldfront = front;
front = front->next;
delete oldfront;
}
```

# 11.6 The STL **deque** and **queue** Containers

- **deque**: a double-ended queue (DEC). Has member functions to enqueue (**push_back**) and dequeue (**pop_front**)

- **queue**: container ADT that can be used to provide a queue based on a **vector**, **list**, or **deque**. Has member functions to enqueue (**push**) and dequeue (**pop**)

# Defining a Queue

- Defining a queue of **`char`**, named cQueue, based on a **`deque`**:

```
deque<char> cQueue;
```

- Defining a **`queue`** with the default base container

```
queue<char> cQueue;
```

- Defining a queue based on a **`list`**:

```
queue<char, list<char> > cQueue;
```

- Prior to C++ 11, spaces are required between consecutive **> >** symbols to distinguish from stream extraction