# Queues

Computer Science Department
University of Central Florida

*COP 3502 – Computer Science I*

# Queues – An Overview

- Queues:
  - Like stacks, Queues are an Abstract Data Type
    - They are NOT built into C
  - We must define them and their behaviors
  - So what is a queue?
    - A data structure that stores information in the form of a typical waiting line
    - New items are added at the **end** of the queue
    - Elements are removed from the **front** of the queue
  - So unlike a stack
    - A queue is accessible from both ends (front and end)

# Queues – An Overview

- Queues:
  - Access Policy:
    - The first element that is inserted into the queue is the first element that will leave the queue
      - Therefore, in order for the last element to leave the queue, it must wait until all elements preceding it are removed
    - Known as the “First in, First out” access policy
      - FIFO for short
    - Real life example:  waiting in line to be served
      - When a customer arrives, they enter the line at the back
      - They wait their turn
      - Finally, they get to the front, are served, and exit the line

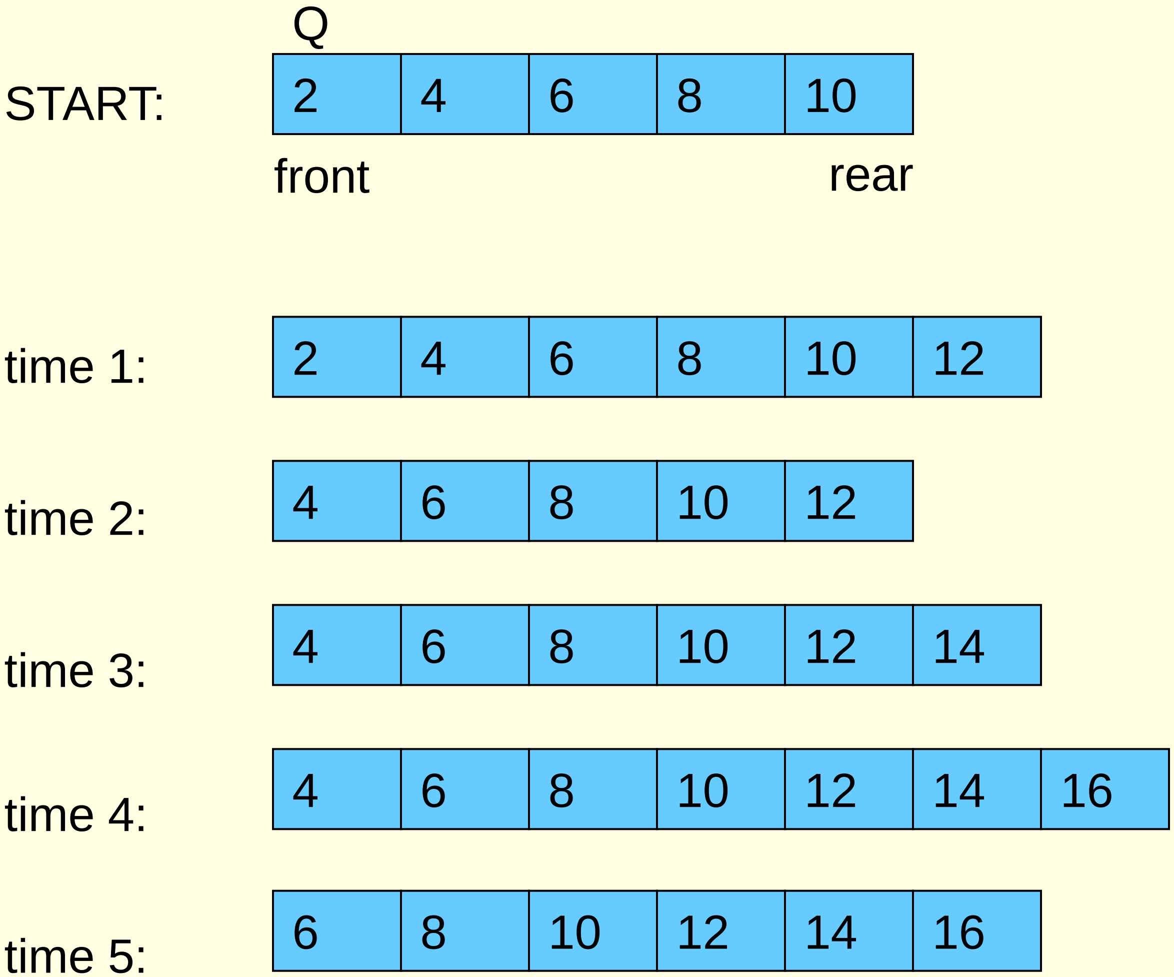
# Queues – An Overview

- Queues:
  - Basic Operations:
    - enqueue:
      - Inserts and element at the rear of the queue
      - O(1) time
    - dequeue:
      - Removes the element at the front of the queue
      - O(1) time
    - peek:
      - Looks at the element at the front of the queue without actually removing it
      - O(1) time

# Queues – An Overview

- Queues:
  - Basic Operations:
    - isEmpty:
      - Checks to see if the queue is empty
      - O(1) time
    - isFull:
      - Checks to see if the queue is full
      - O(1) time
    - clear:
      - Clears the contents of the queue
        - In “queue” order
        - From front to back
      - O(n) time

# FIFO Nature of a Queue



Q

START: 2 | 4 | 6 | 8 | 10

front rear

time 1: 2 | 4 | 6 | 8 | 10 | 12

time 2: 4 | 6 | 8 | 10 | 12

time 3: 4 | 6 | 8 | 10 | 12 | 14

time 4: 4 | 6 | 8 | 10 | 12 | 14 | 16

time 5: 6 | 8 | 10 | 12 | 14 | 16

Sequence of operations

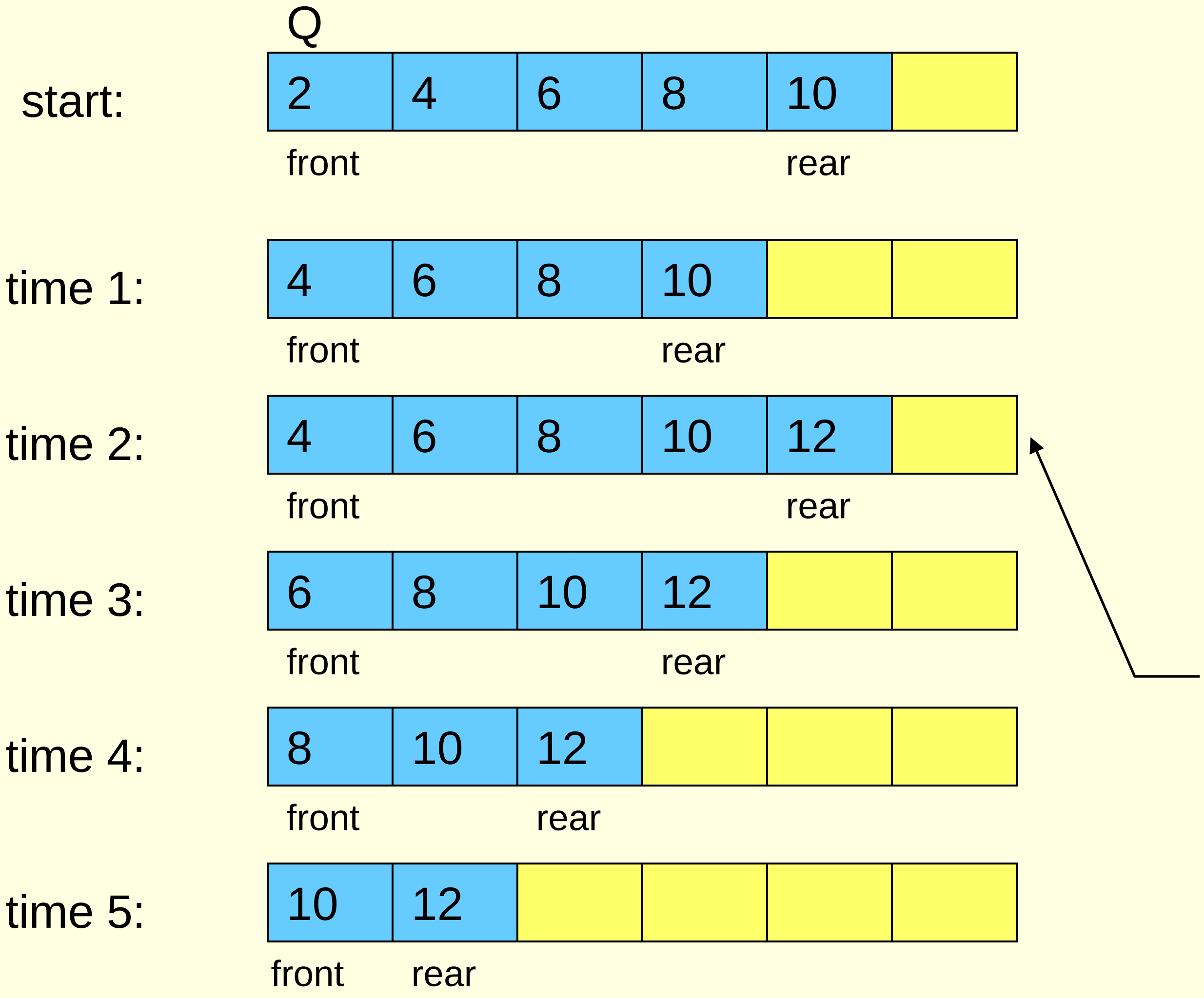
| Time | Operation |
|---|---|
| 1 | insert 12 |
| 2 | remove |
| 3 | insert 14 |
| 4 | insert 16 |
| 5 | remove |

# Queues:  Array Implementation

- Queues:
  - Array Implementation:
    - How would you implement a queue using an array?
    - Think of what "stuff" you would need…
    - Other than the actual array, what else do you need?
    - Remember, you need to enqueue and dequeue
      - Meaning:
      - You need to ALWAYS know where the front and back of the queue are.

“brute force” method

# Queues: Array Implementation



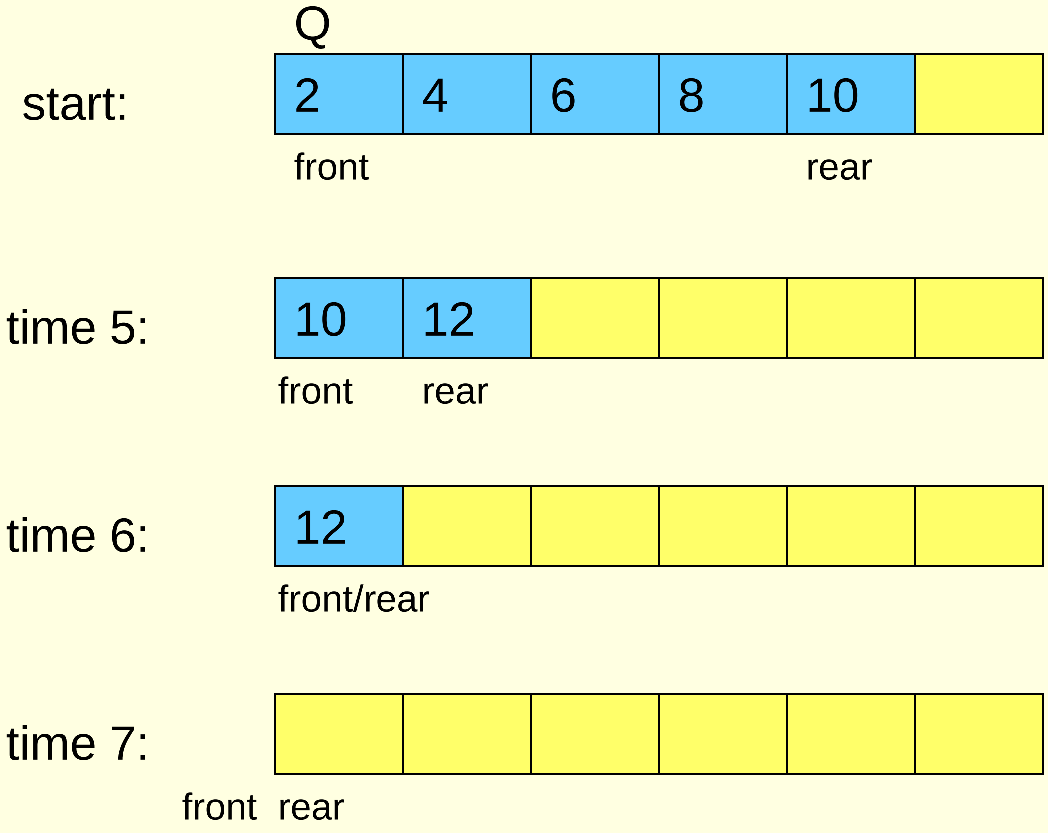
Sequence of operations

| Time | Operation |
|---|---|
| 1 | remove |
| 2 | insert 12 |
| 3 | remove |
| 4 | remove |
| 5 | remove |
| 6 | remove |
| 7 | remove |

Notice that the array now has room to add elements to the queue.

"brute force" method

# Queues: Array Implementation



Sequence of operations

| Time | Operation |
|---|---|
| 1 | remove |
| 2 | insert 12 |
| 3 | remove |
| 4 | remove |
| 5 | remove |
| 6 | remove |
| 7 | remove |

"brute force" method

# Queues: Array Implementation

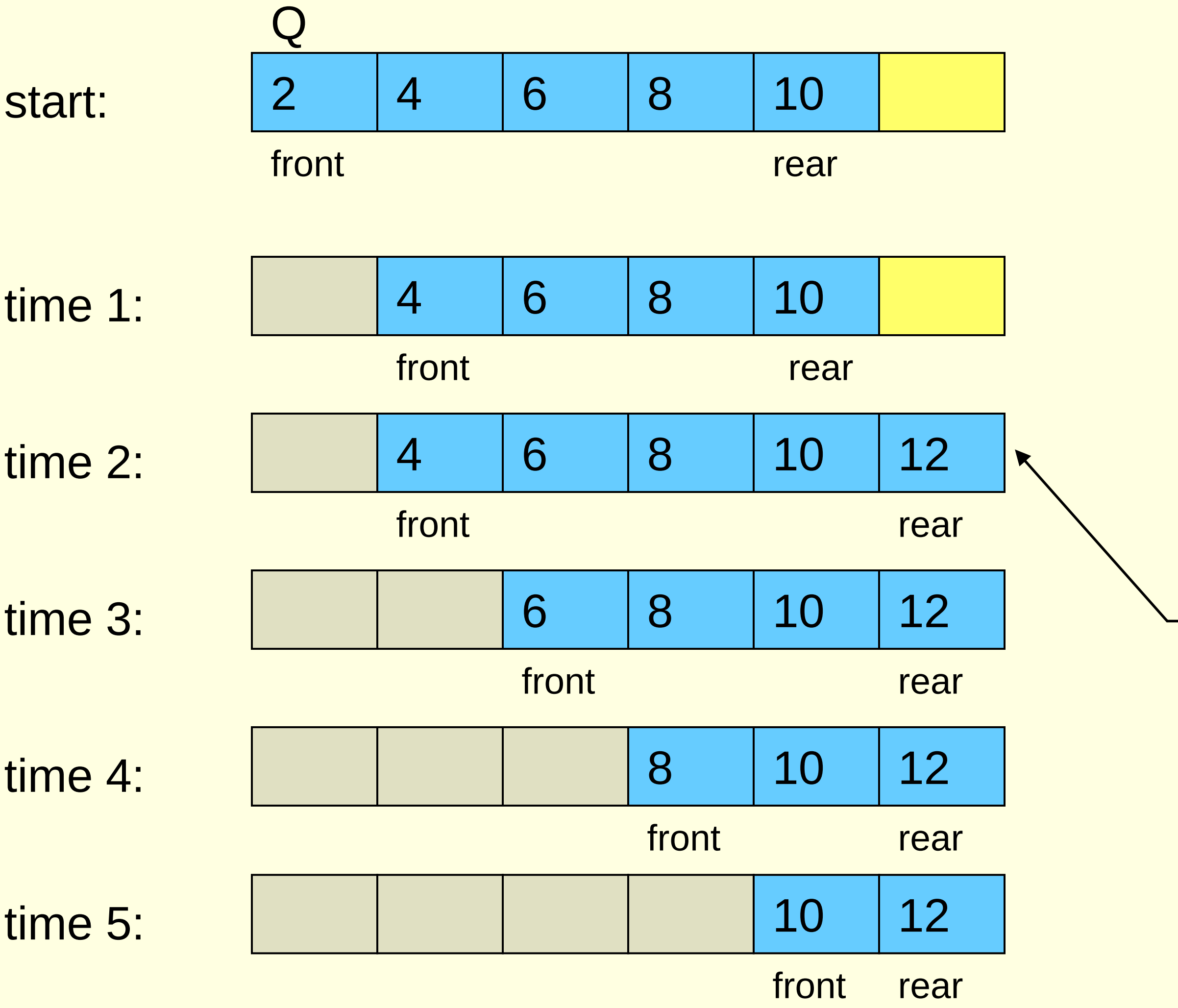
- Queues:
  - What is wrong with the last example?
    - enqueues run in O(1) time.
      - This is a GOOD thing!
    - But look at the dequeue
    - How long does a dequeue take?
      - The dequeue itself takes O(1) time
      - However, after the first node is removed, ALL nodes, that remain in the queue after the dequeue, must be moved forward one position in the array
      - Possibly n elements have to move after one dequeue
      - This is O(n) time per deletion!
      - And we know, conceptually, a dequeue should be O(1)

# Queues: Array Implementation (2)

- Queues:
  - How can we do better?
  - Well, we want to **avoid** moving all items when a dequeue occurs
  - But if we don't move the individual elements…
  - That means we MUST move the front and back "pointers" to those items

  - An example makes this clear…

# Queues:  Array Implementation (2)



Q

start:

| 2 | 4 | 6 | 8 | 10 | |
|---|---|---|---|---|---|
| front | | | | rear | |

time 1:

| | 4 | 6 | 8 | 10 | |
|---|---|---|---|---|---|
| | front | | | rear | |

time 2:

| | 4 | 6 | 8 | 10 | 12 |
|---|---|---|---|---|---|
| | front | | | | rear |

time 3:

| | | 6 | 8 | 10 | 12 |
|---|---|---|---|---|---|
| | | front | | | rear |

time 4:

| | | | 8 | 10 | 12 |
|---|---|---|---|---|---|
| | | | front | | rear |

time 5:

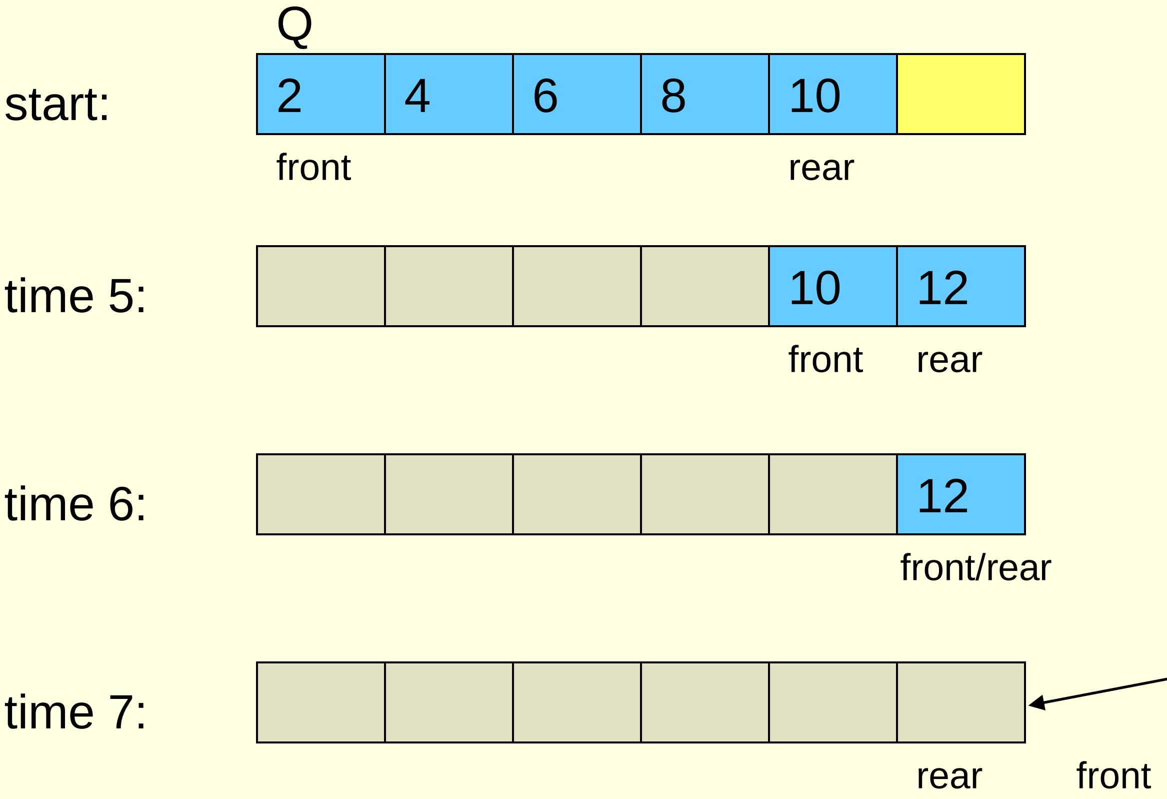
| | | | | 10 | 12 |
|---|---|---|---|---|---|
| | | | | front | rear |

Sequence of operations

| Time | Operation |
|---|---|
| 1 | remove |
| 2 | insert 12 |
| 3 | remove |
| 4 | remove |
| 5 | remove |
| 6 | remove |
| 7 | remove |

Notice that the queue now appears to be full even though there are locations available in the array!

# Queues: Array Implementation (2)



Q

start:

| | | | | | |
|---|---|---|---|---|---|
| 2 | 4 | 6 | 8 | 10 | |
| front | | | | rear | |

time 5:

| | | | | | |
|---|---|---|---|---|---|
| | | | | 10 | 12 |
| | | | | front | rear |

time 6:

| | | | | | |
|---|---|---|---|---|---|
| | | | | | 12 |
| | | | | | front/rear |

time 7:

| | | | | | | |
|---|---|---|---|---|---|---|
| | | | | | | |
| | | | | | rear | front |

Sequence of operations

| Time | Operation |
|---|---|
| 1 | remove |
| 2 | insert 12 |
| 3 | remove |
| 4 | remove |
| 5 | remove |
| 6 | remove |
| 7 | remove |

Even Worse:
The queue now appears full even though there are NO items in the array!

# Queues:  Array Implementation (2)

- Queues:
  - What is wrong with the last example?
    - The problem:  we end up with wasted cells
    - As the front moves towards the rear (when dequeues occur), we have empty, useless cells in the array
    - So we avoided the n moves when we dequeue
    - We did so by simply moving the reference to front
    - But in the process we have wasted space
  - How can we do better?
    - We view the array as if it were circular

# Queues: Array Implementation (3)

- Queues:
  - Circular Array Implementation
    - Circular arrays are a very common way of implementing an array-based queue
    - What is a circular array?
      - It is a regular array
      - We simply "view" it as being circular
    - In a circular implementation, the queue is considered to be full whenever the front of the queue immediately **precedes** the rear of the queue in the counterclockwise direction.
    - The examples on the following pages should help you to visualize a "circular" array.

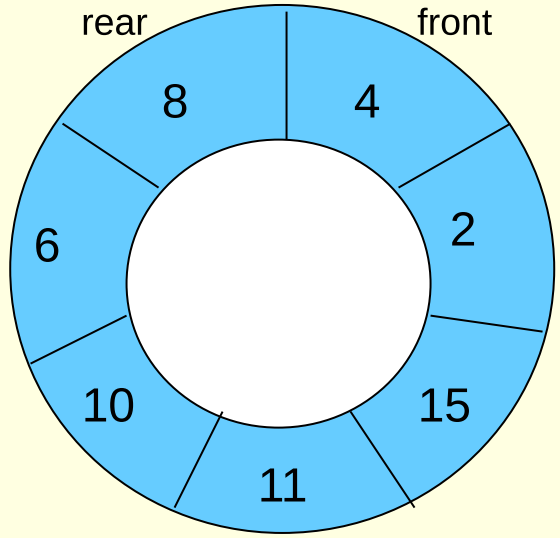
# Queues: Array Implementation (3)

| | | | | | | |
|---|---|---|---|---|---|---|
| 4 | 2 | 15 | 11 | 10 | 6 | 8 |
| front | | | | | | rear |

normal array implementation: queue is full

| | | | | | | |
|---|---|---|---|---|---|---|
| 10 | 6 | 8 | 4 | 2 | 15 | 11 |
| | | rear | front | | | |

circular array implementation: queue is full



visualization of a queue implemented as a circular array

# Queues:  Array Implementation (3)

- Queues:
  - Circular Array Implementation
    - This implementation allows us to keep the elements in their respective “cells” of the array
      - We don’t need to move n elements during dequeues
      - AND we also don’t have wasted space!
    - The circular “view” of the array allows us to “wrap” around the array
      - Assume the length of the array is SIZE
      - It is NOT the case that the rear most stop at index[SIZE-1]
        - meaning, the last element
      - Rather, since the array wraps around, the front could be at a greater index than the index of the rear!

# Queues: Circular Array Implementation

- Queues:
  - Circular Array Implementation
    - The next several slides illustrate the operation of a circular array based implementation of a queue.
    - The normal implementation (brute-force) is also shown for comparative purposes.
    - However, remember that the brute force method is extremely inefficient due to the amount of data movement required by dequeue operations.

# Queues: Circular Array Implementation

- Queues:
  - Circular Array Implementation
    - The scenario begins at some point in time before which other enqueue and dequeue operations have occurred on the queue.
    - Our scenario begins with some elements already in the queue.
      - As you can see on the next page, these elements were enqueued in the order of: 2, 4, and 8.
    - The scenario continues by enqueuing 6, enqueuing 10, dequeue, enqueuing 18, dequeue, dequeue, dequeue, enqueuing 9, dequeue, dequeue, and finally one last dequeue which empties the queue at this point.

# Queues: Circular Array Implementation

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 4 | 8 | | | | | before |
| front | | rear | | | | | |

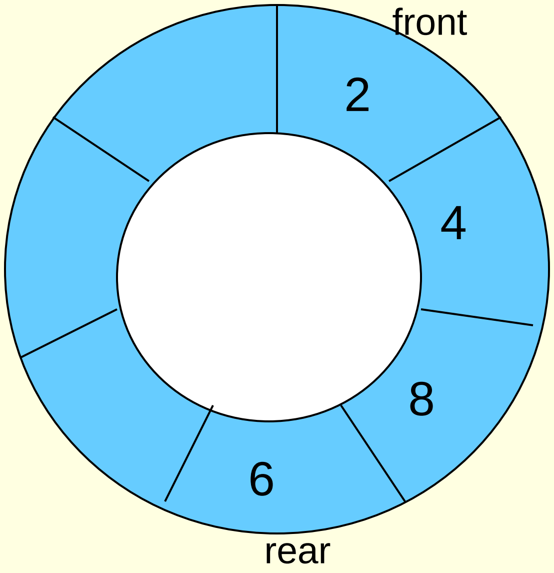
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 4 | 8 | 6 | | | | after |
| front | | | rear | | | | |

normal array implementation

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | 2 | 4 | 8 | | before |
| | | | front | | rear | | |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | 2 | 4 | 8 | 6 | after |
| | | | front | | | rear | |

circular array implementation



visualization of a queue implemented as a circular array after insertion of element 6

**Enqueue element 6**

# Queues: Circular Array Implementation

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 4 | 8 | 6 | | | | before |
| front | | | rear | | | | |

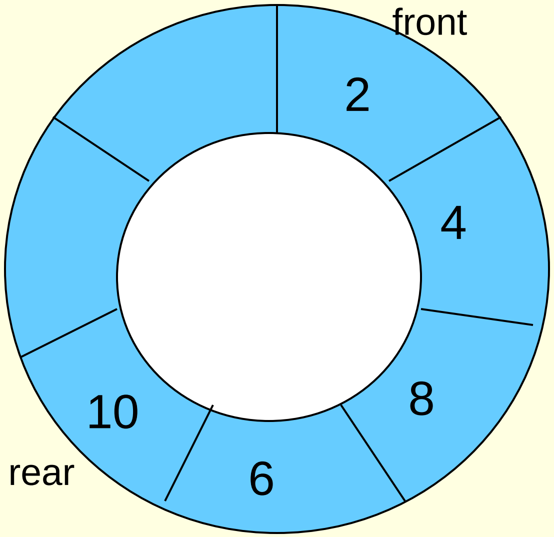
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 4 | 8 | 6 | 10 | | | after |
| front | | | | rear | | | |

normal array implementation

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | 2 | 4 | 8 | 6 | before |
| | | | front | | | rear | |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 10 | | | 2 | 4 | 8 | 6 | after |
| rear | | | front | | | | |

circular array implementation



visualization of a queue implemented as a circular array after insertion of element 10

**Enqueue element 10**

# Queues: Circular Array Implementation

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 4 | 8 | 6 | 10 | | | before |
| front | | | | rear | | | |

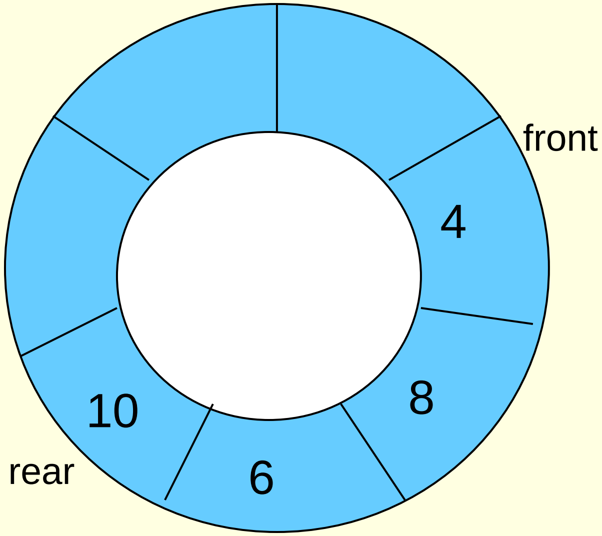
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 4 | 8 | 6 | 10 | | | | after |
| front | | | rear | | | | |

normal array implementation

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 10 | | | 2 | 4 | 8 | 6 | before |
| rear | | | front | | | | |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 10 | | | | 4 | 8 | 6 | after |
| rear | | | | front | | | |

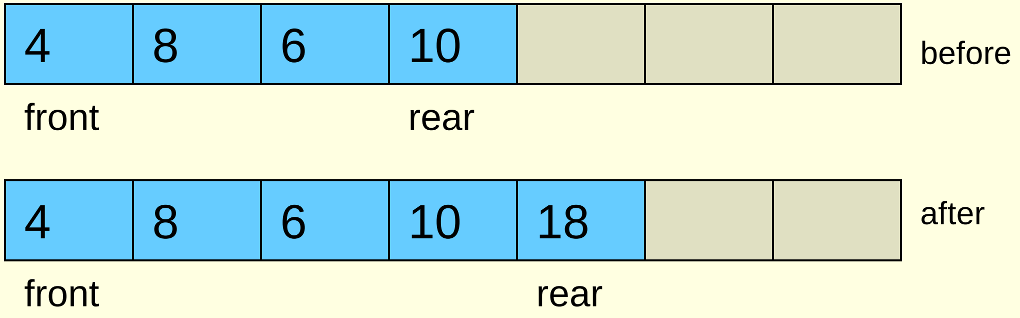
circular array implementation



visualization of a queue implemented as a circular array after dequeue operation

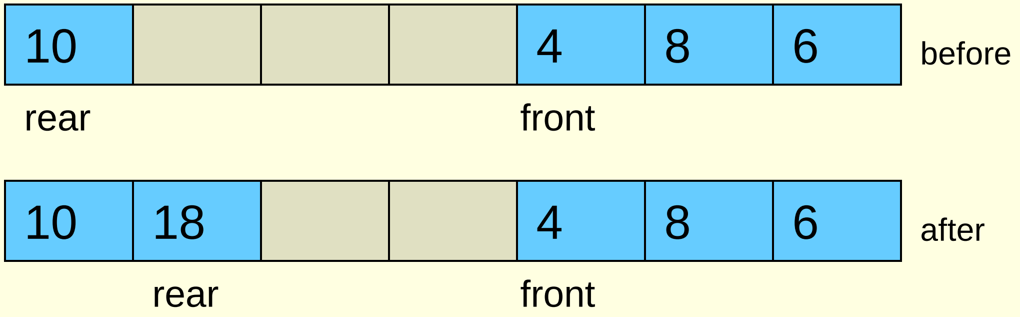
**dequeue**

# Queues: Circular Array Implementation



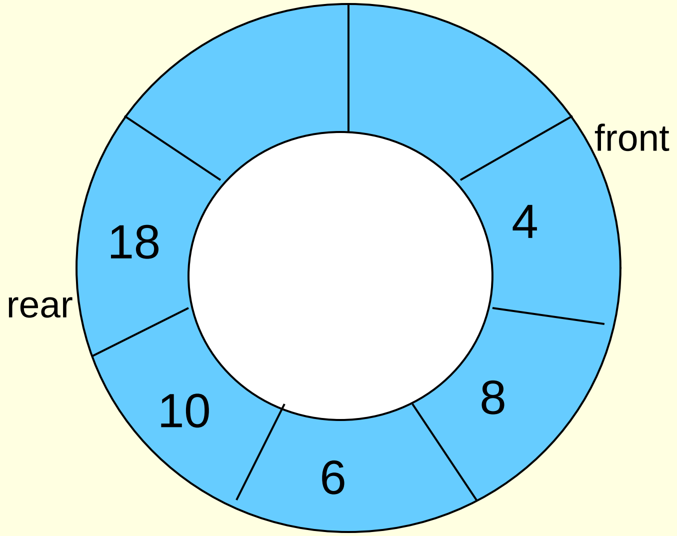
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 4 | 8 | 6 | 10 | | | | before |
| front | | | rear | | | | |
| 4 | 8 | 6 | 10 | 18 | | | after |
| front | | | | rear | | | |

normal array implementation



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 10 | | | | 4 | 8 | 6 | before |
| rear | | | | front | | | |
| 10 | 18 | | | 4 | 8 | 6 | after |
| | rear | | | front | | | |

circular array implementation



visualization of a queue implemented as a circular array after insertion of element 18

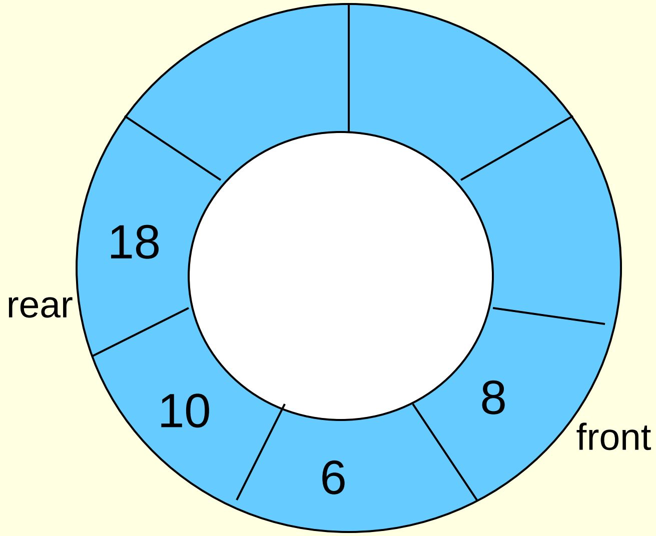
**Enqueue element 18**

# Queues: Circular Array Implementation

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 4 | 8 | 6 | 10 | 18 | | | before |
| front | | | | rear | | | |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 8 | 6 | 10 | 18 | | | | after |
| front | | | rear | | | | |

normal array implementation



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 10 | 18 | | | 4 | 8 | 6 | before |
| | rear | | | front | | | |

visualization of a queue implemented as a circular array after dequeue operation

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 10 | 18 | | | | 8 | 6 | after |
| | rear | | | | front | | |

circular array implementation

**dequeue**

# Queues: Circular Array Implementation

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 8 | 6 | 10 | 18 | | | | before |
| front | | | rear | | | | |

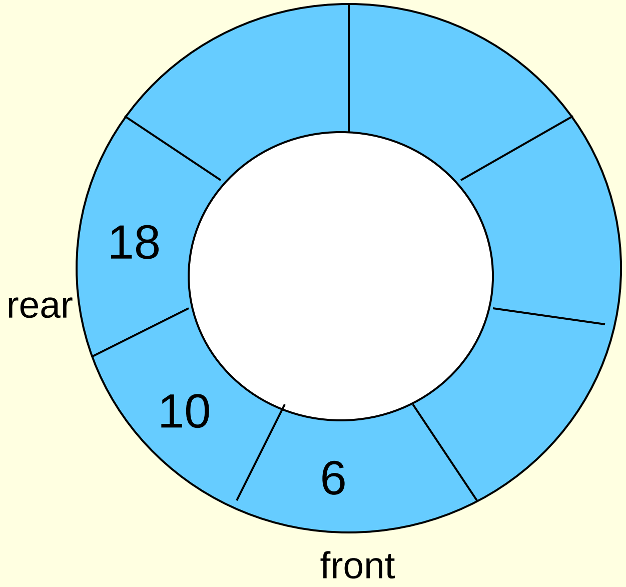
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | 10 | 18 | | | | | after |
| front | | rear | | | | | |

normal array implementation

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 10 | 18 | | | | 8 | 6 | before |
| | rear | | | | front | | |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 10 | 18 | | | | | 6 | after |
| | rear | | | | | front | |

circular array implementation



visualization of a queue implemented as a circular array after dequeue operation

**dequeue**

# Queues: Circular Array Implementation

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | 10 | 18 | | | | | before |
| front | | rear | | | | | |

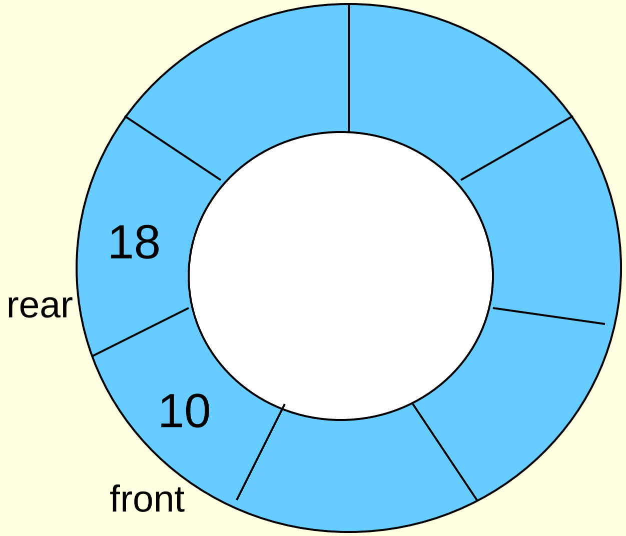
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 10 | 18 | | | | | | after |
| front | rear | | | | | | |

normal array implementation

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 10 | 18 | | | | | 6 | before |
| | rear | | | | | front | |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 10 | 18 | | | | | | after |
| front | rear | | | | | | |

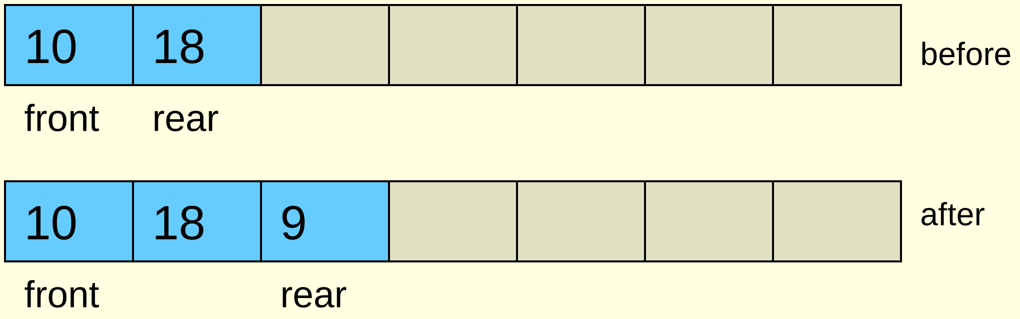
circular array implementation



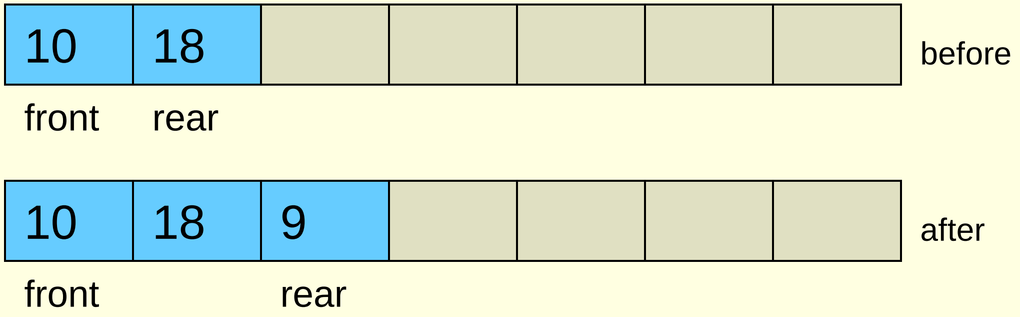
visualization of a queue implemented as a circular array after dequeue operation

**dequeue**

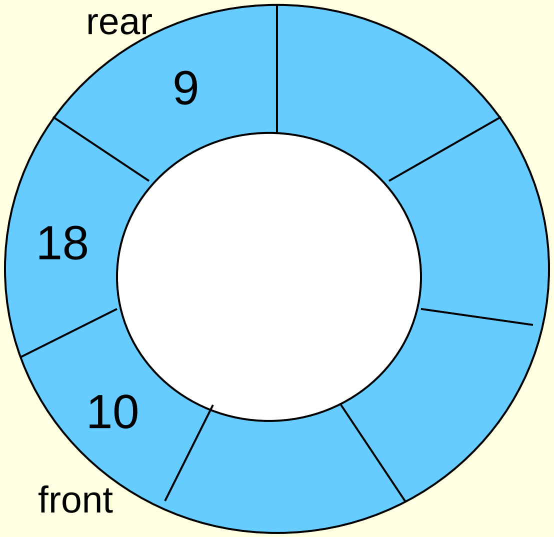
# Queues: Circular Array Implementation



normal array implementation



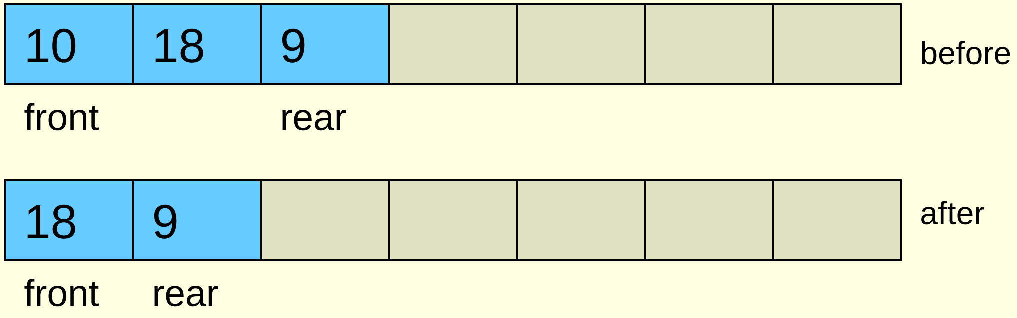
circular array implementation



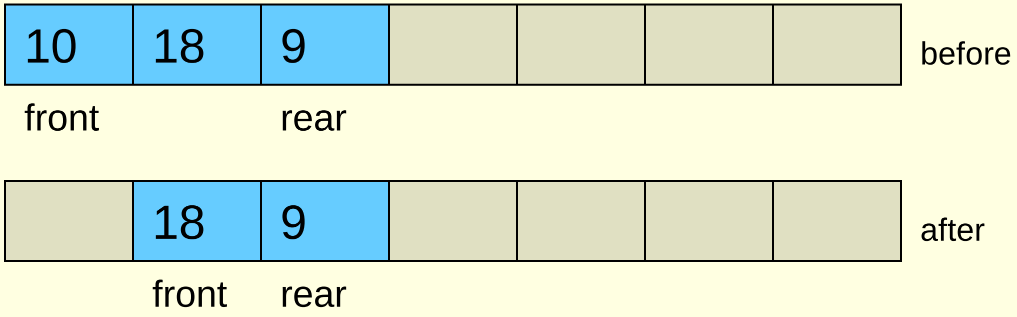
visualization of a queue implemented as a circular array after insertion of element 9

**Enqueue element 9**

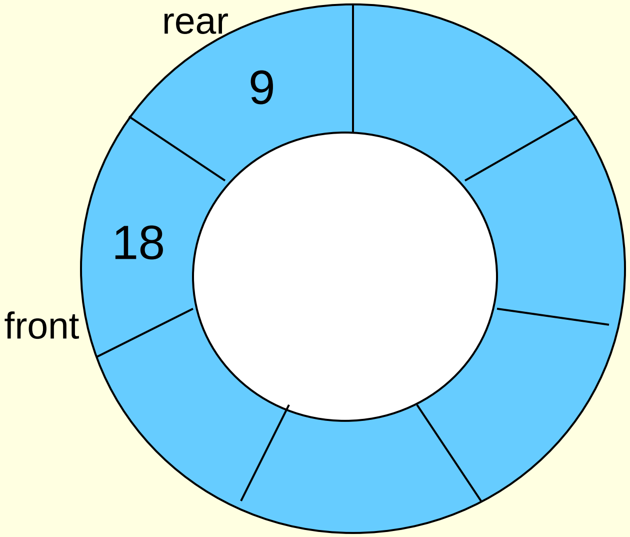
# Queues: Circular Array Implementation



normal array implementation



circular array implementation



visualization of a queue implemented as a circular array after dequeue operation

**dequeue**

# Queues: Circular Array Implementation

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 18 | 9 | | | | | | before |
| front | rear | | | | | | |

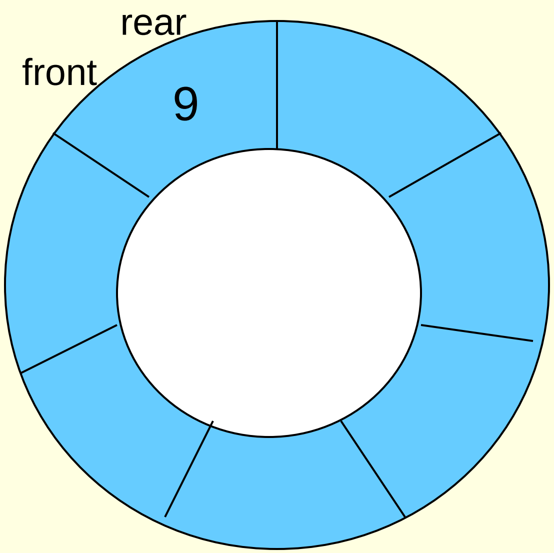
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 9 | | | | | | | after |
| front/rear | | | | | | | |

normal array implementation

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | 18 | 9 | | | | | before |
| | front | rear | | | | | |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | 9 | | | | | after |
| | | front/rear | | | | | |

circular array implementation



visualization of a queue implemented as a circular array after dequeue operation

**dequeue**

# Queues: Circular Array Implementation

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 9 | | | | | | | before |

front/rear

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | | | after |

front/rear

normal array implementation

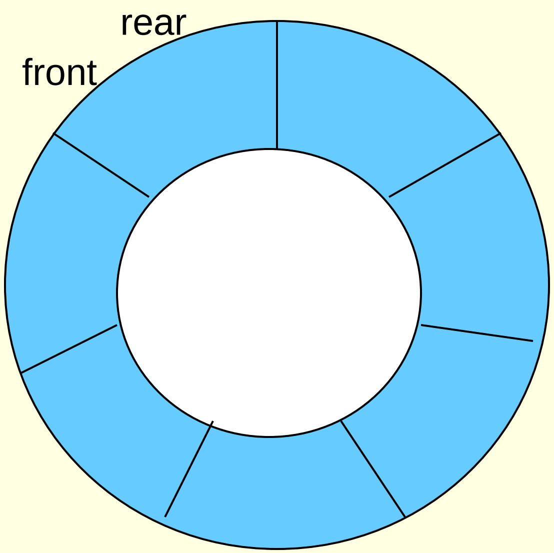
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | 9 | | | | | before |

front/rear

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | | | after |

front/rear

circular array implementation



visualization of a queue implemented as a circular array after dequeue operation

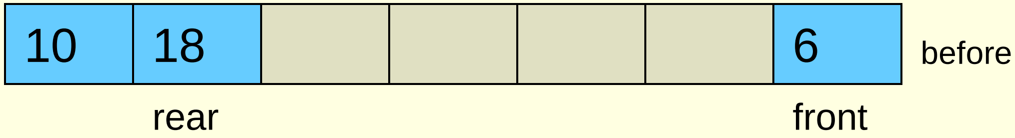
**dequeue – queue empties!**

# Queues: Circular Array Implementation

- Queues:
  - Circular Array Implementation
    - So this works great in pictures
    - But think about something…
      - How did we modify the position (index) of front and rear?
      - Did we just increment the front/rear indices as needed?
        - Meaning, did we simply increment the index of rear every time an enqueue occurs?
        - And did we simply increment the index of front every time a dequeue occurs?
        - In a normal array, this is fine.
      - However, this is a circular array, and we must pay attention!
      - Simply incrementing will not cut it!

# Queues: Circular Array Implementation

- Queues:
  - Circular Array Implementation
    - How did we modify the position (index) of front and rear?
      - We find (and then modify) the index of front and rear using modulo arithmetic.
        - This implements the circular nature of this array
      - Ex: suppose we have the situation below (also from page 26)

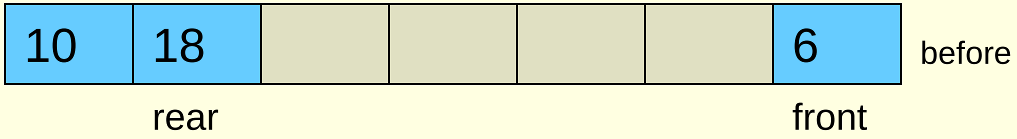


| 10 | 18 |  |  |  |  | 6 | before |
|---|---|---|---|---|---|---|---|
|  | rear |  |  |  |  | front |  |

      - If we dequeue, the front will need to refer to the ‘10’ in index 0!
      - So how do we make this happen?
      - Can we simply increment front?

# Queues: Circular Array Implementation

- Queues:
  - Circular Array Implementation
    - But think about something…
      - Ex: suppose we have the situation below (also from page 26)



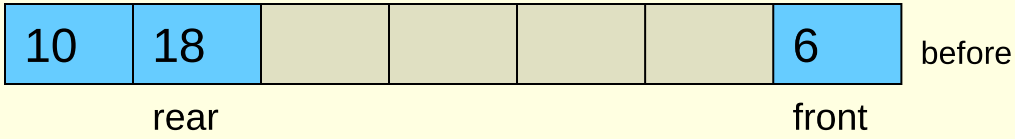
| 10 | 18 |  |  |  |  | 6 | before |
|---|---|---|---|---|---|---|---|
|  | rear |  |  |  |  | front |  |

      - If we dequeue, we usually simply increment the front
        - But if we did so, this would make front refer to index 7
        - BUT this is out of bounds!!!
      - However, (front + 1) mod 7 = 0
        - This is PRECISELY the index we want!

The ‘7’ here refers to the size of the array

# Queues: Circular Array Implementation

- Queues:
  - Circular Array Implementation
    - But think about something…
      - Ex: suppose we have the situation below (also from page 26)



|  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|
| 10 | 18 |  |  |  |  | 6 | before |
|  | rear |  |  |  |  | front |  |

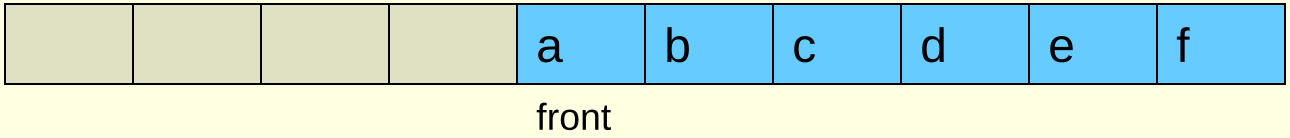
      - So how do we get front to “point” to index 0?
      - We need to use mod!
      - We increment front and then mod it by the queue size
      - front = (front + 1) mod 7
        - So now the new front refers to index 0.
        - This is PRECISELY the index we wanted!

# Queues: Circular Array Implementation

- Circular Array Implementation
  - Another method:
    - We don’t need to save the index for the rear.
    - Why?
    - Because if we know the index to the front
    - AND if we know the number of elements
    - we can quickly determine the new enqueue position
      - Ex: let’s say front was at index 7 and there are 2 elements
        - This means the rear would be at index 8
        - And the NEW enqueue position would be index 9
      - So we see that the NEW enqueue position is found by simply <u>adding the index of the front and the number of elements</u>
    - But we need to take care of wraparound…

# Queues: Circular Array Implementation

- Circular Array Implementation
  - Another method:
    - Assume we have a queue of size 10
      - From index 0 to index 9
    - The front is currently index 4
    - And there are 6 elements already in the queue

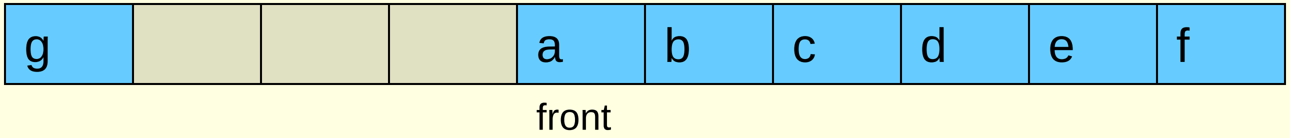


| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | a | b | c | d | e | f |
| | | | | front | | | | | |

    - The name of this array is myQueue
    - And the next operation is enqueue(g)
      - Remember, enqueue is a function that we write in the program
      - So this ‘g’ is sent over to the enqueue function as “char val”

# Queues: Circular Array Implementation

- Circular Array Implementation
  - Another method:
    - Assume we have a queue of size 10
      - From index 0 to index 9
    - The front is currently index 4
    - And there are 6 elements already in the queue



| g | | | | a | b | c | d | e | f |
|---|---|---|---|---|---|---|---|---|---|

- We know that the next enqueue will go at index 0
- But how do we do this in code (using mod)?
- myQueue[(front + numElements)%SIZE] = val
- myQueue[(4 + 6) % 10] = val
- myQueue[0] = val

# Queues: Circular Array Implementation

- Queues:
  - Circular Array Implementation
    - Using Dynamically Allocated Arrays
      - Before we get to the code, there is one other important point to make
      - If we use dynamically allocated arrays for queues, that is fine
      - Remember the steps needed when the array is full:
      1) Allocated a new, larger array (double the size)
      2) Copy the elements from the old array to the new one
      3) Deallocate the space for the old array
      4) Point to the new array appropriately

      - Step 2 now becomes a bit complicated…

# Queues: Circular Array Implementation

- Queues:
  - Circular Array Implementation
    - Using Dynamically Allocated Arrays
      2) Copy the elements from the old array to the new one
      - We can no longer loop through the elements, one by one, and copy them into the corresponding array element in the new array
      - Why?
      - Because of the wraparound issue
      - Consider the following scenario:

| | | | |
|---|---|---|---|
| 6 | 3 | 4 | 5 |
| | | front | |

      - The array is full and we want to enqueue(12)

# Queues: Circular Array Implementation

- Queues:
  - Circular Array Implementation
    - Using Dynamically Allocated Arrays
      2) Copy the elements from the old array to the new one
      - Consider the following scenario:

|   |   |   |   |
|---|---|---|---|
| 6 | 3 | 4 | 5 |

front

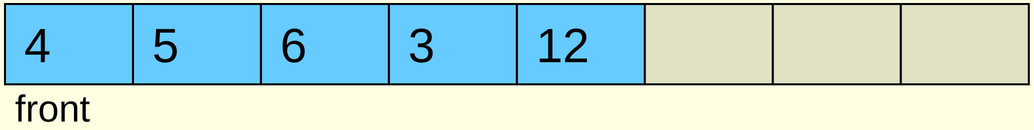
      - The array is full and we want to enqueue(12)
      - If we simply copy the contents, we come up with:

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 6 | 3 | 4 | 5 |   |   |   |   |

      - But where do ‘front’ and ‘rear’ go?
      - Where should 6 really be in this array???

# Queues:  Circular Array Implementation

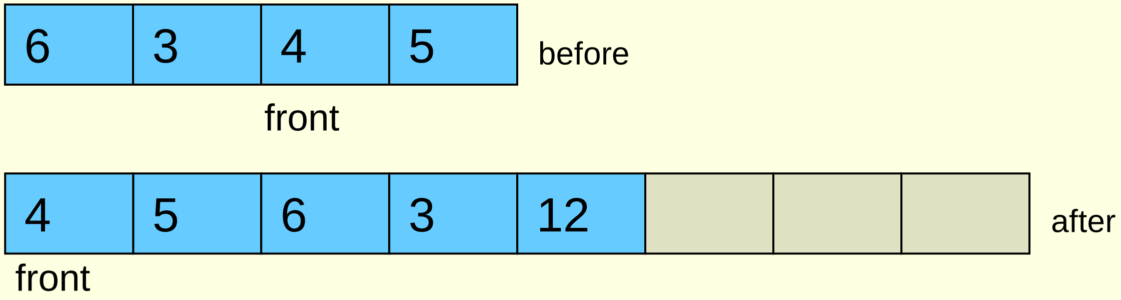
- Queues:
  - Circular Array Implementation
    - Using Dynamically Allocated Arrays
      2) Copy the elements from the old array to the new one
      - So what is the problem:
      - We see that the indices for the wraparound are only accurate for one array size!
        - They don't work when copied to larger array sizes.
      - What we need to do is reset front to 0
      - Then copy the elements into the array accordingly



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 4 | 5 | 6 | 3 | 12 | | | |
| front | | | | | | | |

# Queues: Circular Array Implementation

- Queues:
  - Circular Array Implementation
    - Using Dynamically Allocated Arrays
      2) Copy the elements from the old array to the new one



6 3 4 5 before

front

4 5 6 3 12 after

front

- Here’s how we do this in code:

```
for (i=front, j=0; i<ARRAY_SIZE; i++, j++)
      temp[j] = values[i];
for (i=0; i<front; i++, j++)
      temp[j] = values[i];
```

# Brief Interlude:  Human Stupidity

# Queues: Circular Array Implementation

- Circular Array Code:
  - Here is our queue struct:

```
struct queue {
        int* elements;
        int front;
        int numElements;
        int queueSize;
};
```

  - Contents:
    - An array for the elements of the queue
    - An integer for the index into the front of the queue
    - An integer for the number of elements in the queue
    - An integer representing the current size of the queue

# Queues: Circular Array Implementation

- Circular Array Code:
  - Here are the functions used in the code:
    - `void init(struct queue* qPtr);`
    - `int enqueue(struct queue* qPtr, int val);`
    - `int dequeue(struct queue* qPtr);`
    - `int empty(struct queue* qPtr);`
    - `int peek(struct queue* qPtr);`
  - In main, we make the queue using a pointer of type struct queue
    - We then allocate the space accordingly and call ‘`init`’

# Queues: Circular Array Implementation

- Circular Array Code:
  - init:

```
void init(struct queue* qPtr) {
        // The front index is 0, as is the number of elements.
        qPtr->elements = (int*)malloc(sizeof(int)*INIT_SIZE);
        qPtr->front = 0;
        qPtr->numElements = 0;
        qPtr->queueSize = INIT_SIZE;
}
```

  - Notes:
    - This function is straightforward
    - We must allocate the space for the actual array of elements
    - Then initialize all other struct members

# Queues: Circular Array Implementation

- Circular Array Code:
  - enqueue:
    - Here is the function header:

```
int enqueue(struct queue* qPtr, int val) {
        // body of function
}
```

    - So we send over two things:
      - The pointer to the queue
      - and the new value to enter into the rear of the queue
    - The function then tries to insert “val” into the appropriate spot of the queue

# Queues: Circular Array Implementation

- Circular Array Code:
  - enqueue:
    - Two scenarios:
      1) IF the queue is NOT full…meaning there is room
         - We simply insert "`val`" to the correct spot
         - NOTE:
           - We must use mod to take care of wraparound
           - We reference the new location with:
           - (front + numElements) % queueSize
      2) ELSE, if the queue is full
         - We need to realloc
         - Copy the values correctly
         - And then insert "`val`" correctly taking care of wraparound

# Queues: Circular Array Implementation

- Circular Array Code:
  - enqueue:
    - Two scenarios:
      1) IF the queue is NOT full…meaning there is room

```
int enqueue(struct queue* qPtr, int val) {
        int i;
        if (qPtr->numElements != qPtr->queueSize) {
                qPtr->elements[(qPtr->front+qPtr->numElements)%qPtr->queueSize] = val;
                (qPtr->numElements)++;
                return 1;
        }

        else {
                //...more code here
        }
}
```

# Queues:  Circular Array Implementation

- Circular Array Code:
  - enqueue:
    - Two scenarios:
      2) ELSE, if the queue is full

```
int enqueue(struct queue* qPtr, int val) {
        else {
                realloc(qPtr->elements, (qPtr->queueSize)*sizeof(int)*2);
                for (i=0; i<=qPtr->front-1; i++)
                        qPtr->elements[i+qPtr->queueSize] = qPtr->elements[i];

                qPtr->elements[i+qPtr->queueSize] = val;
                (qPtr->queueSize) *= 2;
                (qPtr->numElements)++;
                return 1;
        }
}
```

# Queues: Circular Array Implementation

- Circular Array Code:
  - dequeue:
    - This one is a bit easier
    - If the queue is empty, we immediately return
      - Can’t dequeue from an empty queue!
    - ELSE
      - We store the value that we want to return
      - We adjust the index to the front of the queue accordingly
      - We adjust the numElements struct member
        - Make it one fewer since we are dequeuing
      - Finally, we return the dequeued value

# Queues: Circular Array Implementation

- Circular Array Code:
  - dequeue:

```
int dequeue(struct queue* qPtr) {
        int retval;

        // Empty case.
        if (qPtr->numElements == 0)
                return EMPTY;

        retval = qPtr->elements[qPtr->front];

        qPtr->front = (qPtr->front + 1)% qPtr->queueSize;

        (qPtr->numElements)--;

        return retval;
}
```

# Queues: Circular Array Implementation

- Circular Array Code:
  - empty:

```
int empty(struct queue* qPtr) {
        return qPtr->numElements == 0;
}
```

  - Notes:
    - This function is straightforward
    - Simply returns a 1 if the queue is empty
      - If numElements is equal to 0

# Queues: Circular Array Implementation

- Circular Array Code:
  - peek:

```
int peek(struct queue* qPtr) {
        if (qPtr->numElements != 0)
                return qPtr->elements[qPtr->front];
        else
                return EMPTY;
}
```

  - Notes:
    - If there are elements in the queue
      - The front element is returned (but not dequeued)
    - Else if the queue is empty
      - We simply return accordingly

# Queues: Linked Lists Implementation

- Queues:
  - Linked Lists Implementation:
    - What would be the problem with a typical linked list implementation?
    - Either the enqueue or dequeue operation would take O(n) time
    - Why?
    - Because we need access to BOTH ends of the queue
    - And a linked lists starts at the front (or some end)
    - So if we use linked lists:
      - We MUST maintain pointers for both the front AND the rear (last node) of the list

# Queues:  Linked Lists Implementation

- Queues:
  - Linked Lists Implementation:
    - Consider the following operations:
    - **enqueue**

1) Create a new node and store the inserted value into it.
2) Link the back node's next pointer to this new node.
3) Move the back node to point to the newly added node.

- **dequeue**

1) Store a temporary pointer to the beginning of the list
2) Move the front pointer to the next node in the list
3) Free the memory pointed to by the temporary pointer.

# Queues: Linked Lists Implementation

- Queues:
  - Linked Lists Implementation:
    - Consider the following operatios:
    - **front**

      1) Directly access the data stored in the first node through the front pointer to the list.

    - **empty**

      1) Check if both pointers (front, back) are null.

    - Code for both array and linked list implementations are on the website under sample programs.

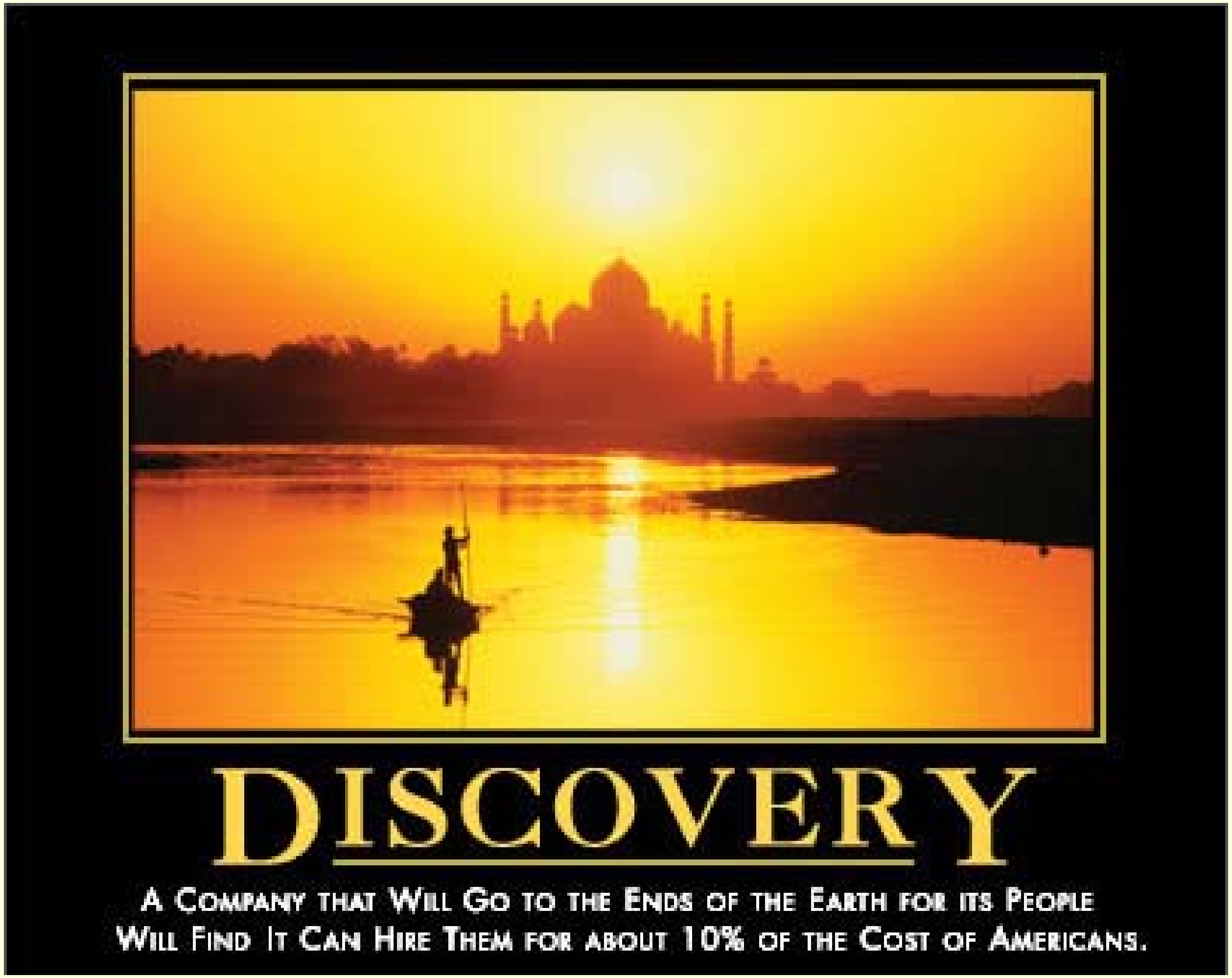
# Queues

WASN'T

THAT

SUPERB!

# Daily Demotivator


DISCOVERY
A Company that Will Go to the Ends of the Earth for its People
Will Find It Can Hire Them for about 10% of the Cost of Americans.

# Queues

Computer Science Department
University of Central Florida

*COP 3502 – Computer Science I*