

# Queues

10/12/04 19:58

Queues

1

# The Queue ADT

## ◆ Queues

- Insert in order
- Insert at end, remove from front
- Delete the item that has been in the queue the longest
- FIFO - first in, first out
- Maintain front and back pointers

10/12/04 19:58

Queues

2

# The Queue ADT

## ◆ Examples of queues

- Lines
  - ◆ Ticket line, amusement park line, etc
- Access to shared resources (e.g., printer queue)
- Phone calls to large companies
- Waiting list for adding classes

10/12/04 19:58

Queues

3

# The Queue ADT

## ◆ Main queue operations:

- **enqueue**(Object o): inserts an element o at the end of the queue
- **dequeue**(): removes and returns the element at the front of the queue

## ◆ Auxiliary queue operations:

- **front**(): returns the element at the front without removing it
- **size**(): returns the number of elements stored
- **isEmpty**(): returns a Boolean indicating whether no elements are stored

10/12/04 19:58

Queues

4

## The Queue ADT

- ◆ Exceptions
  - Attempting the execution of dequeue or front on an empty queue throws an `EmptyQueueException`
  - Attempting the execution of enqueue on a full queue throws a `FullQueueException`

10/12/04 19:58

Queues

5

## Naïve Array-based Queue

- ◆ Two variables keep track of the front and rear
  - front* index of the front element, init 0
  - rear* index immediately past the rear element, init 0
- ◆ Array location *rear* is kept empty
- ◆ If  $front = rear$ , queue is empty

10/12/04 19:58

Queues

6

## Naïve Array-based Queue

front = 4  
rear = 8



What happens on the next enqueue operation?  
What are the possible solutions?

10/12/04 19:58

Queues

7

## Circular Array-based Queue

- ◆ Best solution - use a wrapping or circular array
  - Enqueue at the beginning of the array

*front = rear! Is the queue empty now?*

front = 4  
rear = 3

Solution - keep one empty spot



10/12/04 19:58

Queues

8

# Queue Data Structure

```

class Queue {
private:
    objectType queue[MAX_QUEUE_SIZE];
    int front;
    int rear;
public:
    functions for queue manipulation
    constructor sets front and rear to 0
};
    
```

# Queue Operations

```

void Enqueue ( itemType item )
    if ((front == rear + 1) or ((rear == size - 1) and (front == 0)))
        error - queue full
    else
        Q[rear] = item
        if (rear == size - 1)
            rear = 0
        else
            rear ++
    
```

# The Mod Operator

◆ We can use the modulo operator to simplify

- $7/3 = 2$  remainder 1
- $7\%3 = 1$  (just saves the remainder)



$N = 6$

$r = (5 + 1) \% 6 = 0$

# Queue Operations - Enqueue

```

void Enqueue ( itemType item )
    if (front == (rear+1)%size)
        error - queue full
    else
        Q[rear] = item
        rear = (rear + 1)%size
    
```

## Queue Operations - Dequeue

- ◆ In class exercise - Write the code for dequeue.
  - Return dequeued item through a parameter passed by reference

10/12/04 19:58

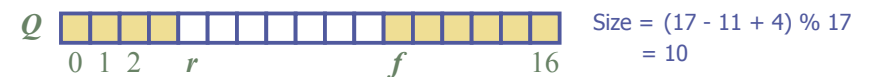
Queues

13

## Queue Operations - Size

**Algorithm *size()***  
**return**  $(N - \textit{front} + \textit{rear}) \bmod N$

**Algorithm *isEmpty()***  
**return**  $(\textit{front} == \textit{rear})$



10/12/04 19:58

Queues

14

## Growable Array-based Queue

- ◆ In an enqueue operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one
- ◆ Similar to what we did for an array-based stack
- ◆ The enqueue operation has amortized running time
  - ∨  $O(n)$  with the incremental strategy
  - ∨  $O(1)$  with the doubling strategy

10/12/04 19:58

Queues

15

## Linked List Based Queue

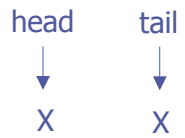
- ◆ Using a linked list can remove the size restrictions of an array
- ◆ Linked list with front and rear pointers
  - Front is the same as head, rear is the same as tail
- ◆ Front and rear initially point to null

10/12/04 19:58

Queues

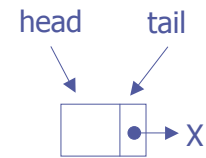
16

# Linked List Based Queue



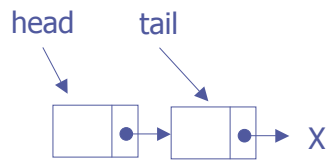
# Linked List Based Queue

## Enqueue



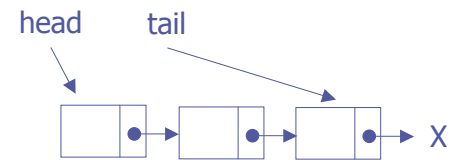
# Linked List Based Queue

## Enqueue



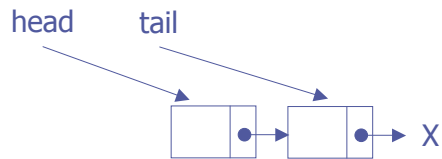
# Linked List Based Queue

## Enqueue



## Linked List Based Queue

### Dequeue



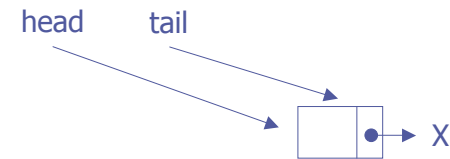
10/12/04 19:58

Queues

21

## Linked List Based Queue

### Dequeue



10/12/04 19:58

Queues

22

## Linked List Based Queue

```
class Queue {  
private:  
    Node* front;  
    Node* rear;  
    int numItems;  
public:  
    all functions to interface with queue  
};
```

10/12/04 19:58

Queues

23

## Linked List Based Queue

```
bool isEmpty ( ) {  
    return ( back == NULL );  
}  
  
void enqueue ( objectType obj ) {  
    Node* newNode = new Node;  
    newNode->item = obj;  
    if ( isEmpty ( ) )  
        front = newNode;  
    else  
        rear->next = newNode;  
    rear = newNode;  
}
```

10/12/04 19:58

Queues

24

## Linked List Based Queue

- ◆ In class exercise - write dequeue
  - Just like removing head in a list with head and tail pointers
  - Think about memory leaks
    - ◆ Do you delete the node or not?
  - Use getFront ( ) if you want to use the node
  - Use dequeue if you just want to remove the node
    - ◆ Delete the node, don't return a reference

10/12/04 19:58

Queues

25

## Linked List Based Queue

10/12/04 19:58

Queues

26

## Queue Big Oh Runtimes

- |               |                     |
|---------------|---------------------|
| ◆ Array based | ◆ Linked list based |
| ■ Enqueue     | ■ Enqueue           |
| ■ Dequeue     | ■ Dequeue           |
| ■ isEmpty     | ■ isEmpty           |
| ■ getFront    | ■ getFront          |

10/12/04 19:58

Queues

27