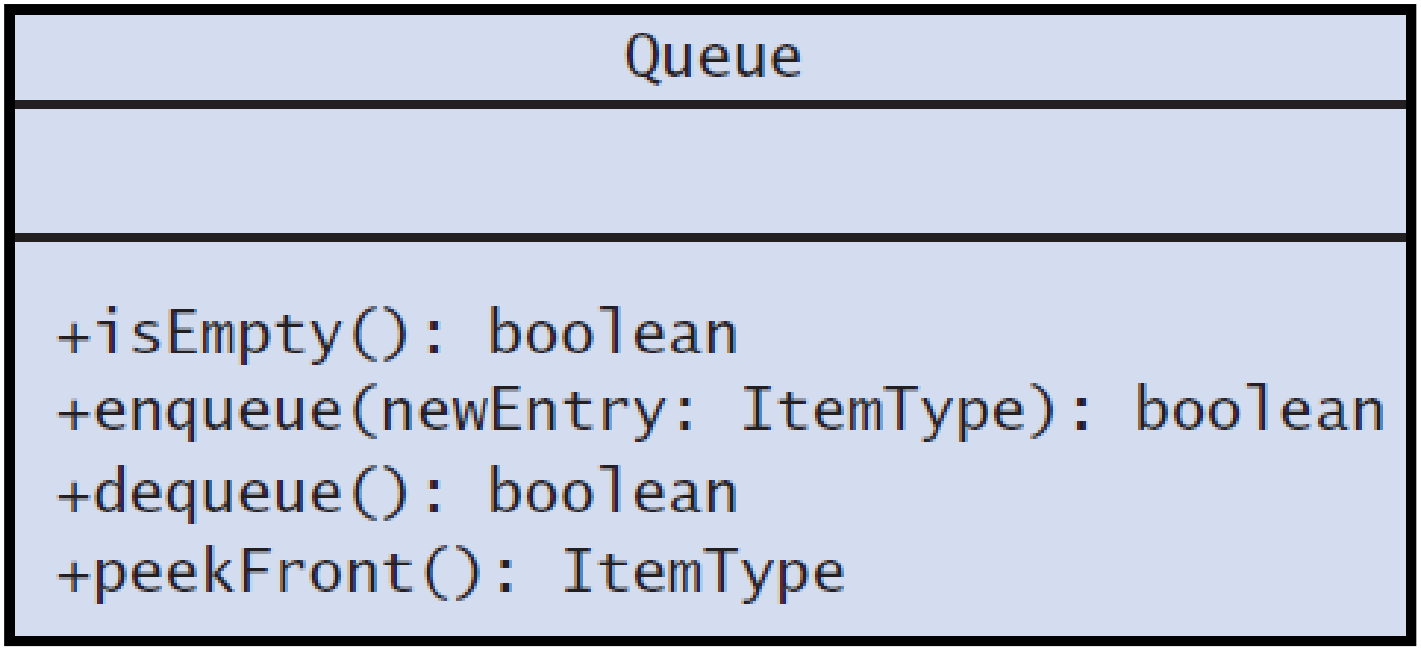
EECE 2560:
Fundamentals of Engineering Algorithms

Queues

The ADT Queue

- Like a line of people
  - First person in line is first person served
  - New elements of queue enter at its back
  - Items leave the queue from its front
- Called FIFO behavior
  - **F**irst **I**n **F**irst **O**ut
- Position-oriented ADTs
  - Stack, list, queue
- Value-oriented ADTs
  - Sorted list

Queue UML diagram

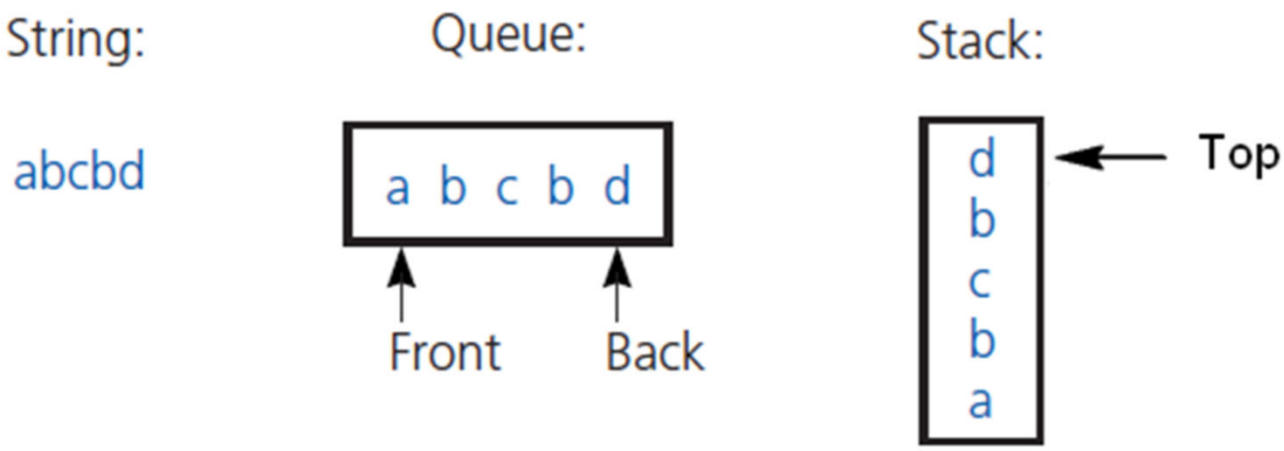


Queue Sample Operations

Operation	Queue after operation (Front →)
aQueue = *an empty queue*	
aQueue.enqueue(5)	5
aQueue.enqueue(2)	5 2
aQueue.enqueue(7)	5 2 7
aQueue.peekFront()	5 2 7 (Returns 5)
aQueue.dequeue()	2 7
aQueue.dequeue()	7

Queue Application: String Palindrome

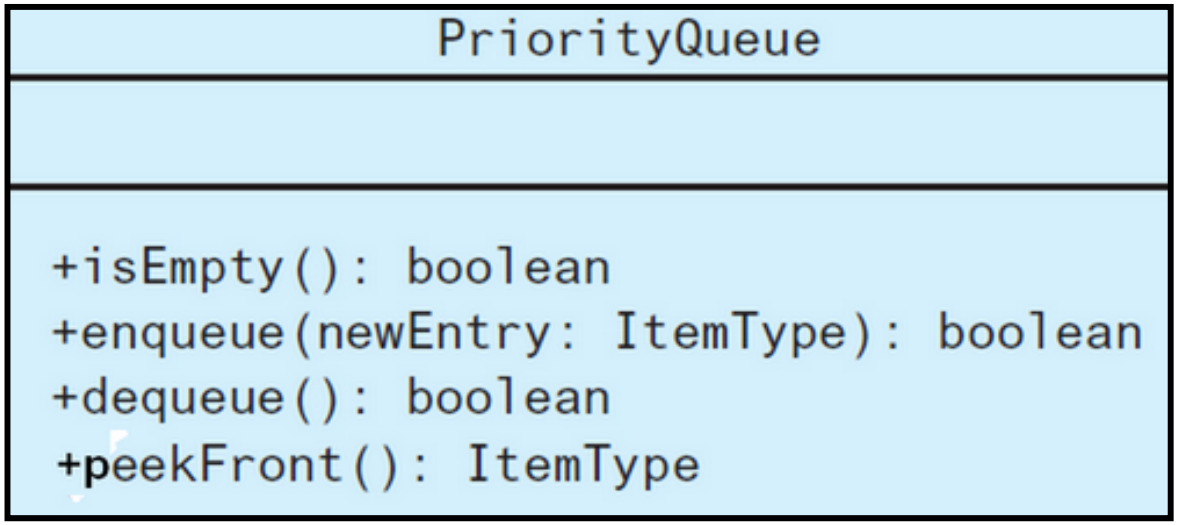
- String palindrome example: **abcdcba**
- *Example:* To solve the problem for string **abcbd**, insert characters a, b, c, b, d into both a queue and a stack
- Remove characters from front of queue, top of stack
- Compare each pair removed
- If all pairs match, string is a palindrome
- Do we need to insert the whole string in both the queue and the stack?



The ADT Priority Queue

- Organize data by priorities
  - Example: organize traffic in the Internet routers based on their priorities (Voice over IP, video streaming, emails, file transfer).
- Priority value
  - We will say high value ⇨ high priority
- Priority queue operations:
  - Test for empty
  - Add to queue in sorted position
  - Remove/get entry with highest priority

Priority Queue UML diagram



- Priority Queue can be implemented as a *sorted list* that maintains the entries in sorted order based on the priority values of the entries.

- It can also be implemented by a data structure that keeps track only of the element with the highest priority (e.g., maximum value) regardless of the order of the remaining elements in the queue.

Queue Application: Simulation

- Simulation models behavior of systems
- Example Problems
  - Calculate the approximate average time bank customer must wait for service from a teller
  - Does the customer wait time decrease with each new teller added?
- Simulation Types:
  - Time-driven simulation
    - Simulates the ticking of a clock
  - Event-driven simulation considers
    - Only the times of certain events (e.g., arrivals and departures of bank customers)
    - Event list contains all future arrival and departure events.

Bank Line Simulation Example (1 of 5)

Sample arrival and transaction times:

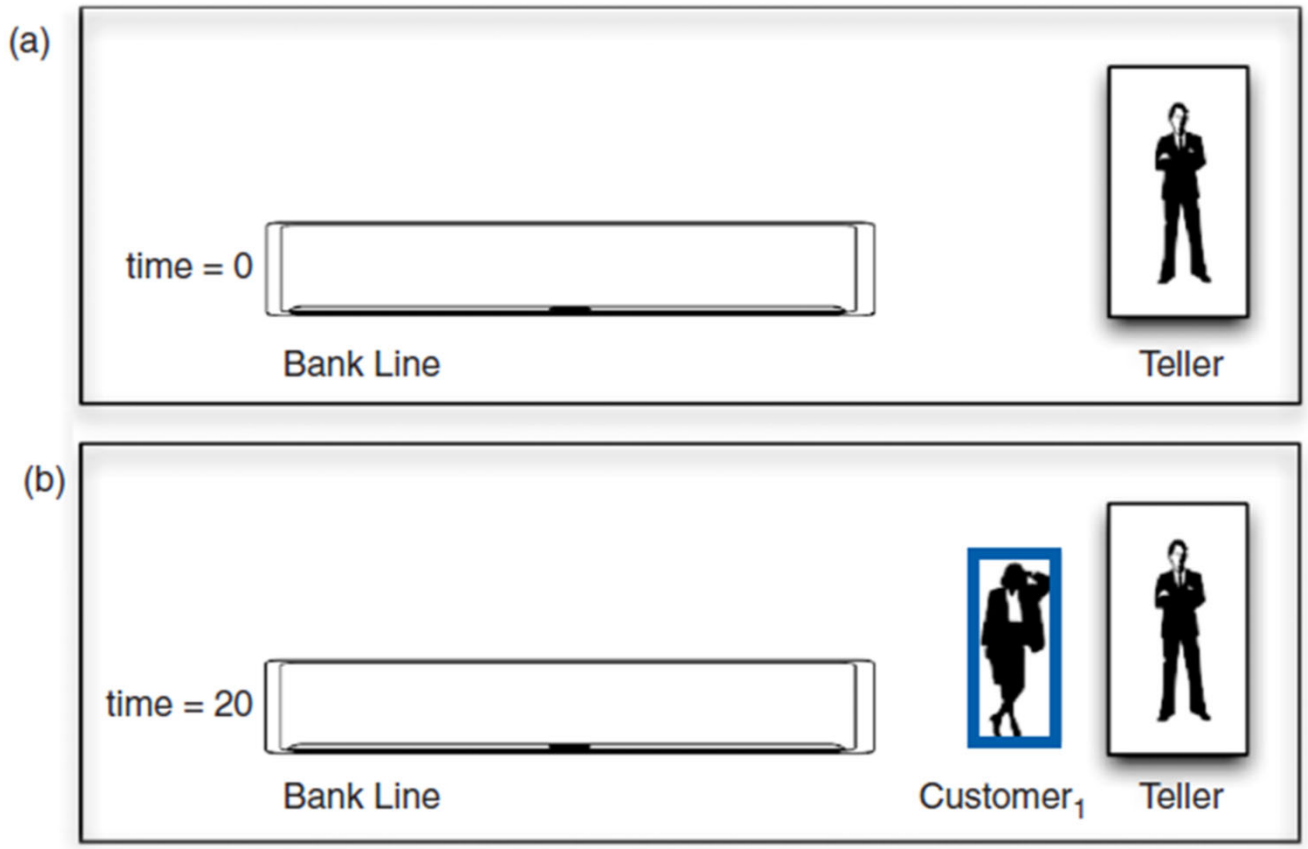
Arrival time	Transaction length
20	6
22	4
23	2
30	3

A typical instance of events:

	Type	Time	Length
(a) Arrival event	A	20	6

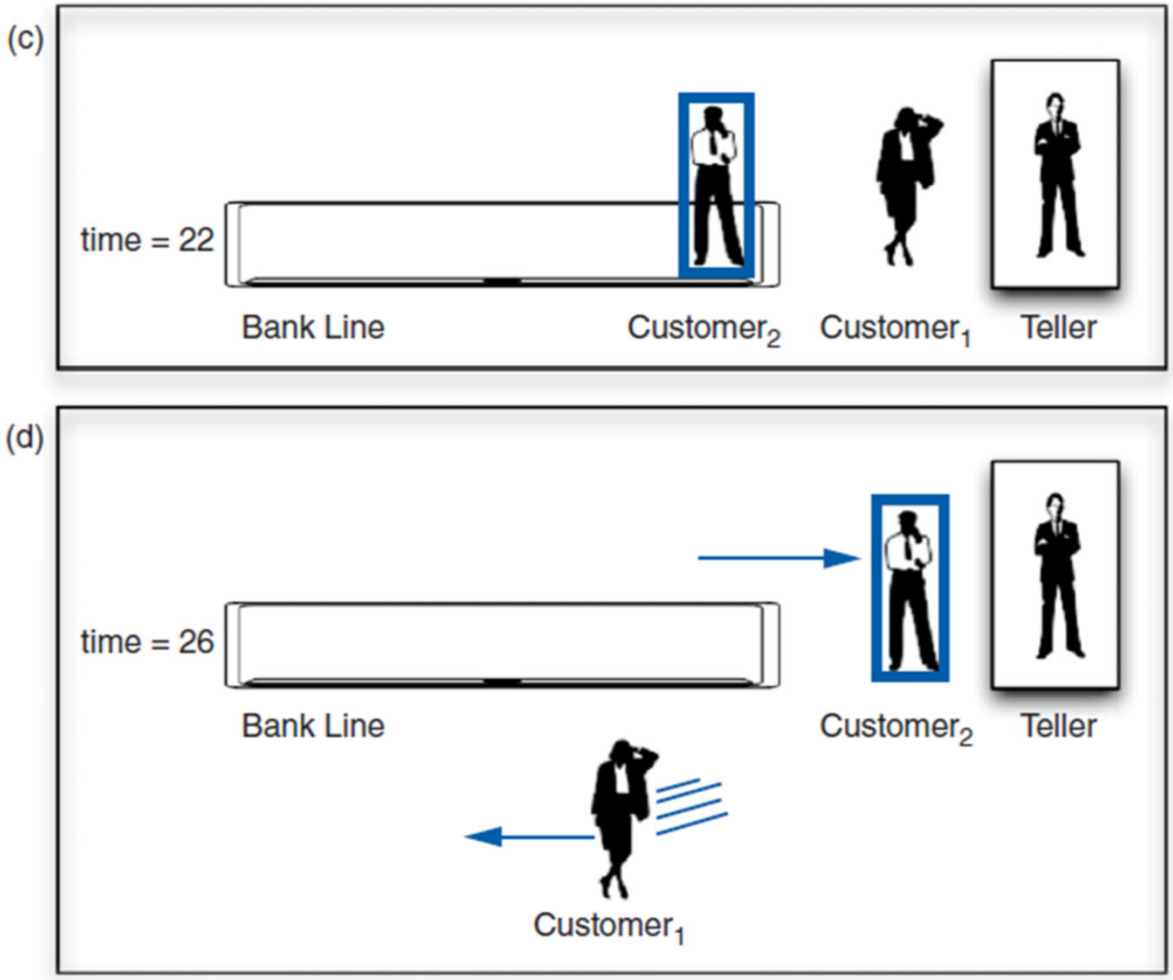
	Type	Time	Length
(b) Departure event	D	26	–

Bank Line Simulation Example (2 of 5)



Arrival Time	Transaction Length
20	6
22	4
23	2
30	3

Bank Line Simulation Example (3 of 5)



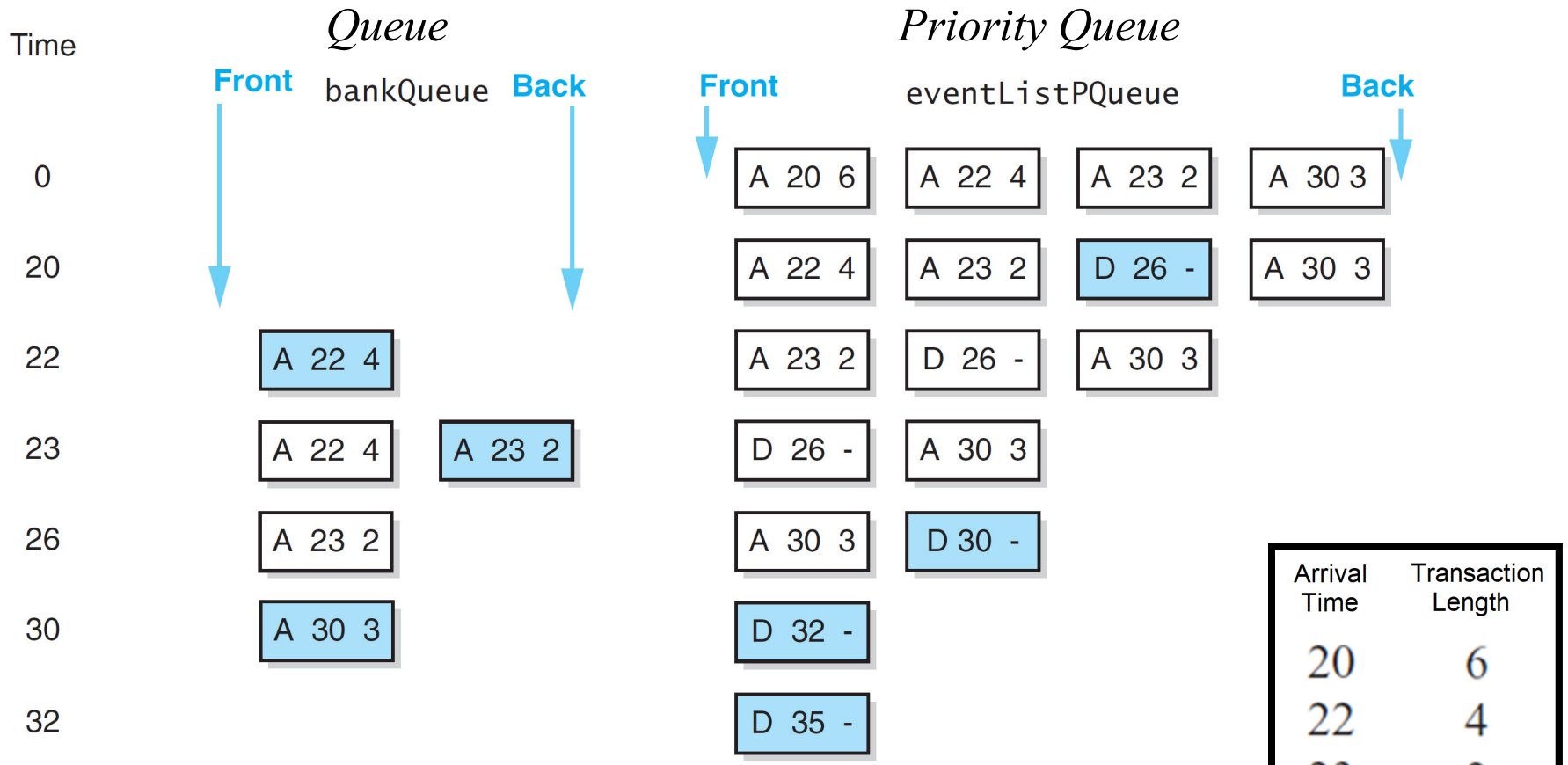
Arrival Time	Transaction Length
20	6
22	4
23	2
30	3

Bank Line Simulation Example (4 of 5)

Time	Event
20	Customer 1 enters bank and begins transaction
	Determine customer 1 departure event is at time 26
22	Customer 2 enters bank and stands at end of line
23	Customer 3 enters bank and stands at end of line
26	Customer 1 departs; customer 2 begins transaction
	Determine customer 2 departure event is at time 30
30	Customer 2 departs; customer 3 begins transaction
	Determine customer 3 departure event is at time 32
30	Customer 4 enters bank and stands at end of line
32	Customer 3 departs; customer 4 begins transaction
	Determine customer 4 departure event is at time 35
35	Customer 4 departs

Arrival Time	Transaction Length
20	6
22	4
23	2
30	3

Bank Line Simulation Example (5 of 5)



Arrival Time	Transaction Length
20	6
22	4
23	2
30	3

Time-Driven Simulation Algorithm

```
// Initialize
currentTime = 0
Initialize the line to “no customers”
while (currentTime <= time of the final event )
{
  if ( an arrival event occurs at time currentTime)
       Process the arrival event
  if ( a departure event occurs at time currentTime)
       Process the departure event
  // When an arrival event and departure event occur at the
  // same time, arbitrarily process the arrival event first
  currentTime++
}
```

- Video games use this approach, since events can occur or need to be processed in almost every unit of time

Event-Driven Simulation Algorithm

```
Initialize the line to "no customers"
while ( events remain to be processed )
{
   currentTime = time of next event
   if ( event is an arrival event )
        Process the arrival event
   else
        Process the departure event
   //When an arrival event and a departure event occur at the
   //same time, arbitrarily process the arrival event first
}
```

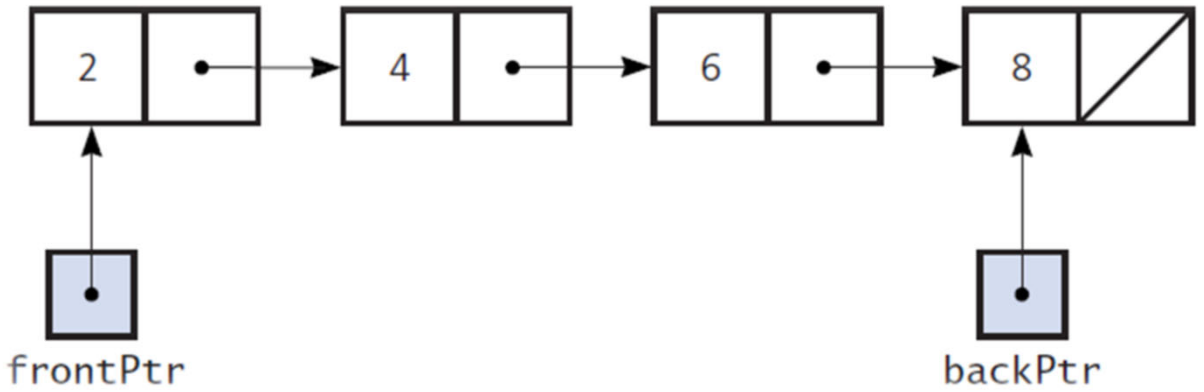
- Here we are interested only in those times at which arrival and departure events occur, as no action is required between events.

ADT Queue Implementations

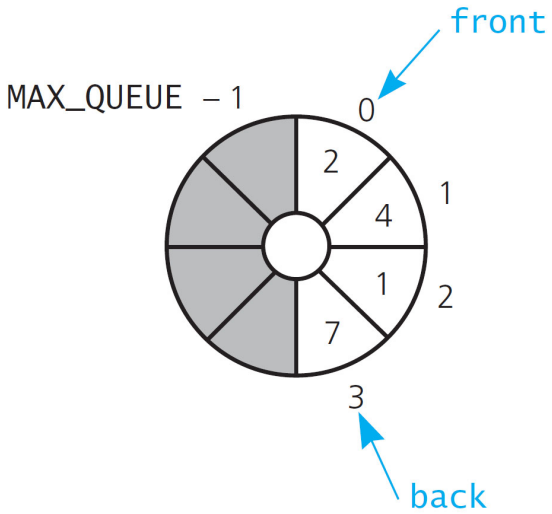
- Like stacks, queues can have
  - Array-based or
  - Link-based implementation.

A Link-Based Implementation

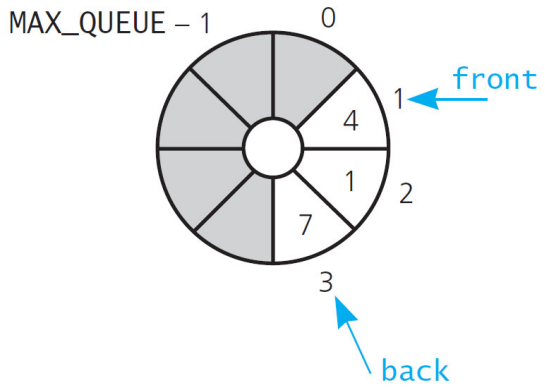
- Similar to other link-based implementation
- One difference … must be able to access entries
  - From front
  - From back
- Requires a pointer to chain's last node
  - Called the "tail pointer"



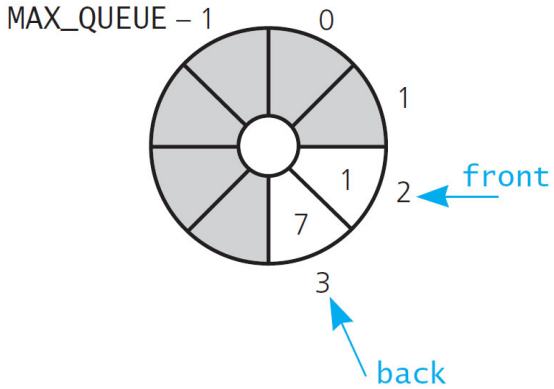
An Array-Based Implementation



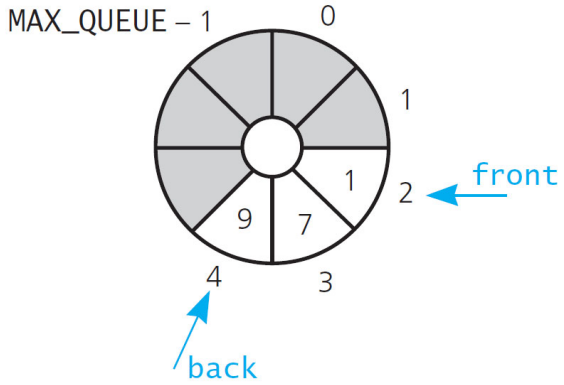
dequeue();



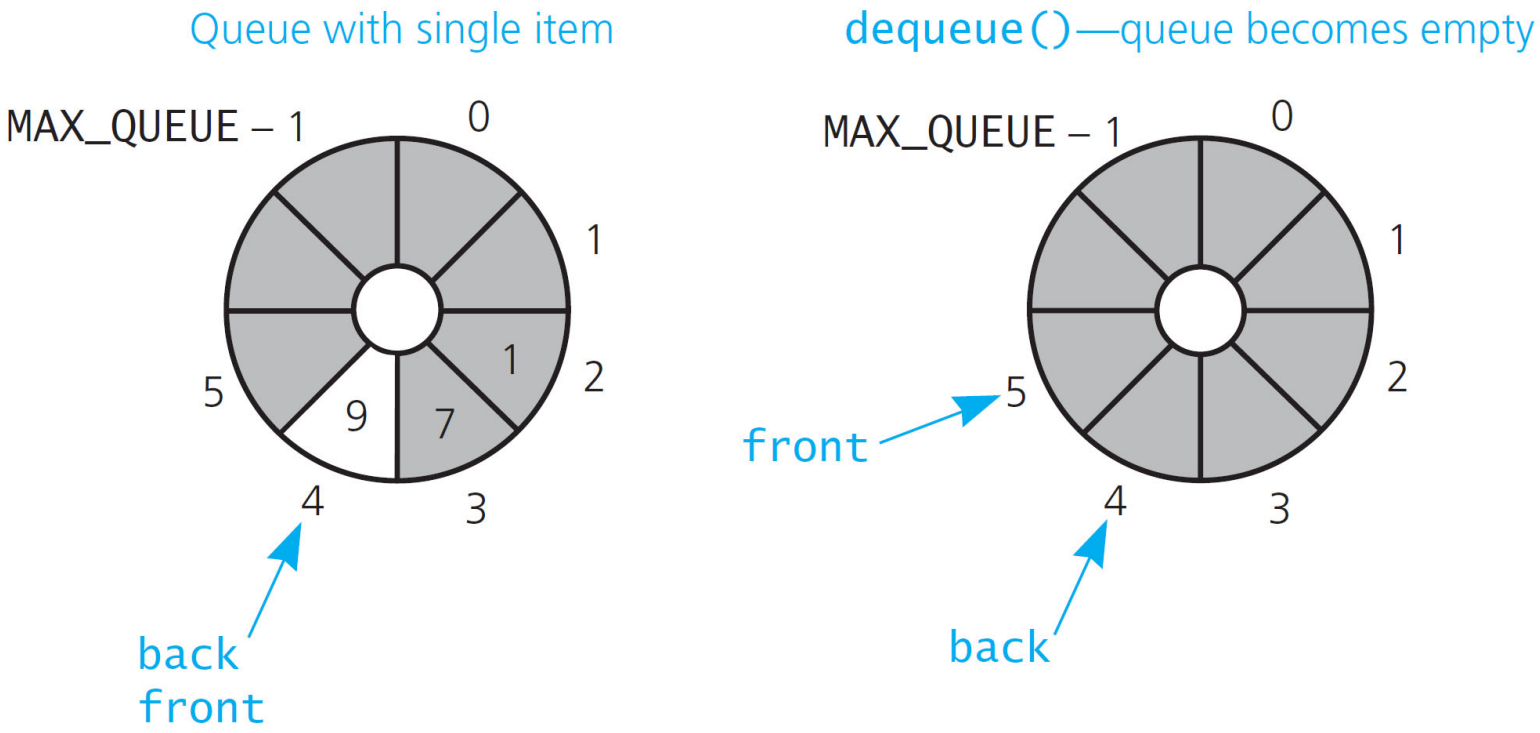
dequeue();



enqueue(9);

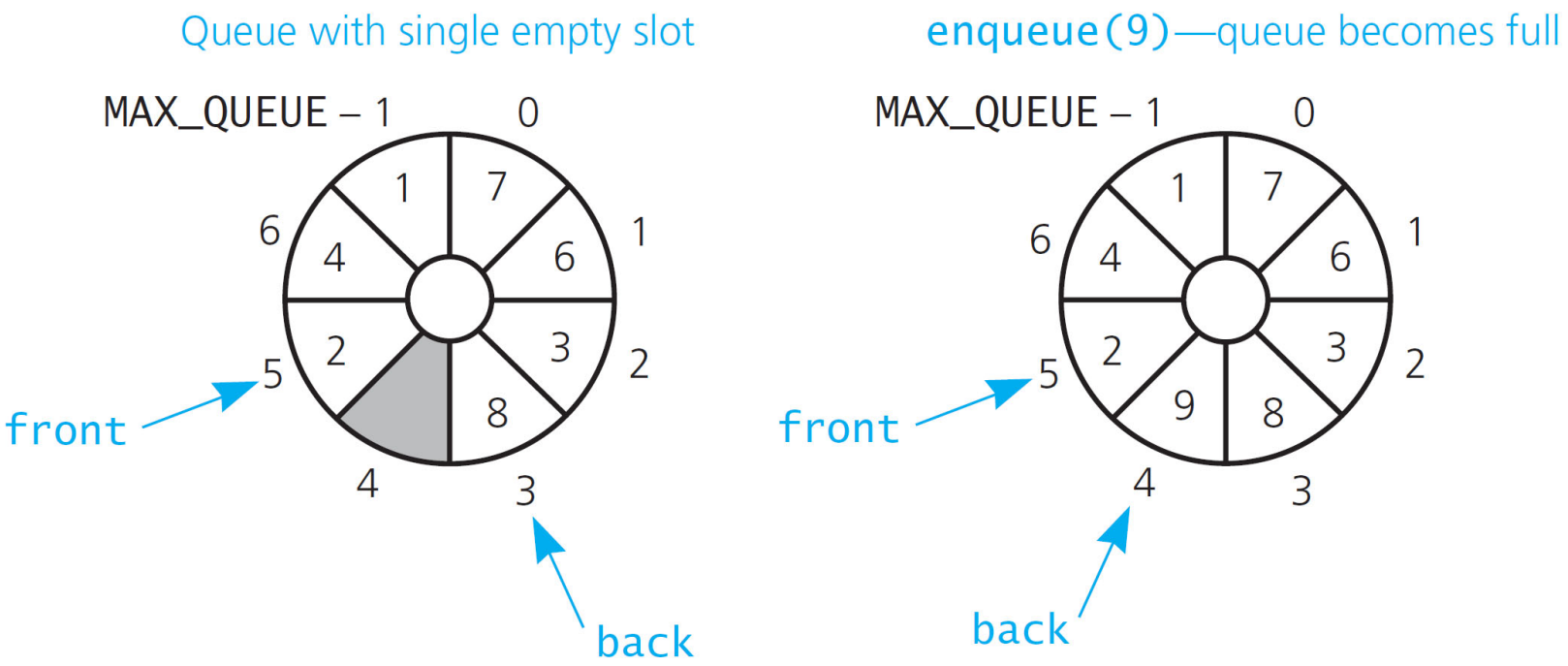


Empty Array Queue



front passes back when the queue becomes empty

Full Array Queue



back catches up to front when the queue becomes full

Array Queue Implementation (1 of 3)

```
template<class ItemType>
class ArrayQueue {
private:
   static const int DEFAULT_CAPACITY = 50;
   ItemType items[DEFAULT_CAPACITY]; // Array of queue items
   int front;       // Index to front of queue
   int back;        // Index to back of queue
   int count;       // Number of items currently in the queue

public:
   ArrayQueue();
   bool isEmpty() const;
   bool enqueue(const ItemType& newEntry);
   bool dequeue();

   ItemType peekFront() const;
}; // end ArrayQueue
```

Array Queue Implementation (2 of 3)

```
template<class ItemType>
ArrayQueue<ItemType>::ArrayQueue() : front(0),
back(DEFAULT_CAPACITY - 1), count(0) {} //end default constructor

template<class ItemType>
bool ArrayQueue<ItemType>::isEmpty() const {
 return count == 0; } // end isEmpty

template<class ItemType>
bool ArrayQueue<ItemType>::enqueue(const ItemType& newEntry) {
   bool result = false;
   if (count < DEFAULT_CAPACITY) { //Queue has room for another item
      back = (back + 1) % DEFAULT_CAPACITY;
      items[back] = newEntry;
      count++;
      result = true;
   } // end if
   return result;
} // end enqueue
```

Array Queue Implementation (3 of 3)

```
template<class ItemType>
bool ArrayQueue<ItemType>::dequeue() {
   bool result = false;
   if (!isEmpty())  {
      front = (front + 1) % DEFAULT_CAPACITY;
      count--;
      result = true;
   } // end if
   return result;
} // end dequeue

template<class ItemType>
ItemType ArrayQueue<ItemType>::peekFront() const {
 assert(!isEmpty()); // If empty, the program is terminated and display an error message
 // Queue is not empty; return front
   return items[front];
} // end peekFront
```