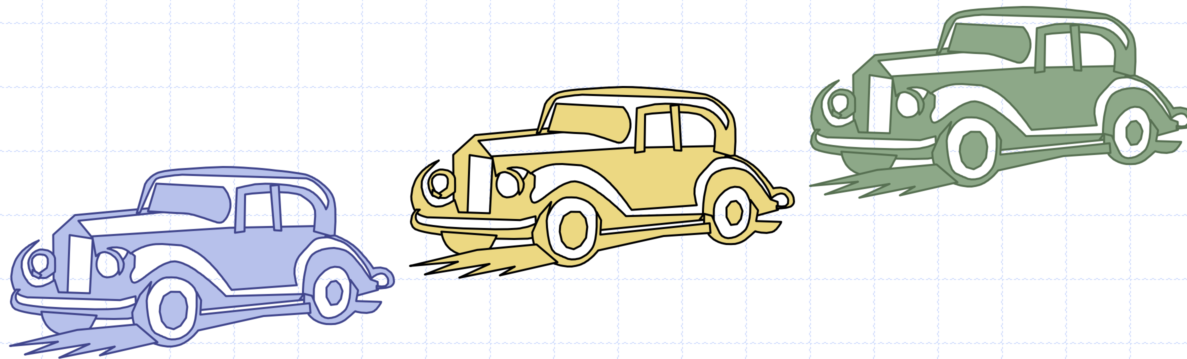
Queues and Deques

Sections 5.2 to 5.3.3



© 2019 Shermer, based on Goodrich, Tamassia, Mount

The Queue ADT

- The Queue ADT stores arbitrary objects
- Insertions and deletions follow the first-in first-out (FIFO) scheme
- Insertions are at the rear of the queue and removals are at the front of the queue
- Main queue operations:
  - enqueue(object): inserts an element at the end of the queue
  - dequeue(): removes the element at the front of the queue
- Auxiliary queue operations:
  - object front(): returns the element at the front without removing it
  - integer size(): returns the number of elements stored
  - boolean empty(): indicates whether no elements are stored
- Exceptions
  - Attempting the execution of dequeue or front on an empty queue throws an QueueEmpty

Example

Operation	*Output*	*Q*
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
dequeue()	–	(3)
enqueue(7)	–	(3, 7)
dequeue()	–	(7)
front()	*7*	(7)
dequeue()	–	()
dequeue()	*"error"*	()
empty()	*true*	()
enqueue(9)	–	(9)
enqueue(7)	–	(9, 7)
size()	*2*	(9, 7)
enqueue(3)	–	(9, 7, 3)
enqueue(5)	–	(9, 7, 3, 5)
dequeue()	–	(7, 3, 5)

Queue Interface

- Pseudo-C++ interface corresponding to our Queue ADT
- Uses an exception class QueueEmpty
- Different from the built-in C++ STL class queue

```
template <typename E>
class Queue {
public:
   int size() const;
   bool empty() const;
   const E& front() const
      throw(QueueEmpty);
   void enqueue (const E& e);
   void dequeue()
      throw(QueueEmpty);
};
```

STL queue class

- The Standard Template Library (STL) provides an implementation of a queue.
- To declare a queue of floats:

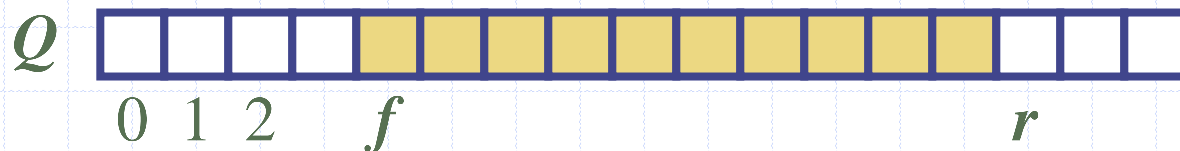
```
#include <queue>
std::queue<float> myQueue;
```

- STL's queue interface is similar to the previous one, but
  - enqueue is called push and dequeue is called pop.
  - There is an extra function back which returns the element at the back of the queue without removing it.
  - Executing pop, front, or back on an empty queue results in *undefined behavior*.

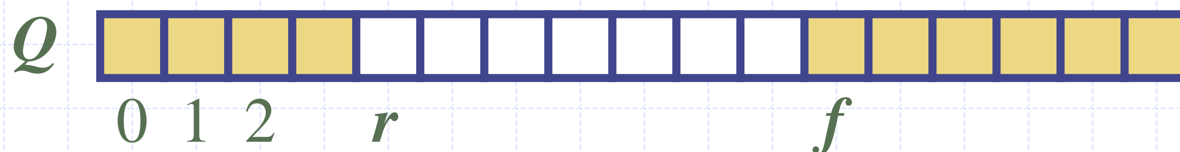
Array-based Queue

- Use an array of size N in a circular fashion
- Three variables keep track of the front and rear
  - f index of the front element
  - r index immediately past the rear element
  - n number of items in the queue

normal configuration



wrapped-around configuration

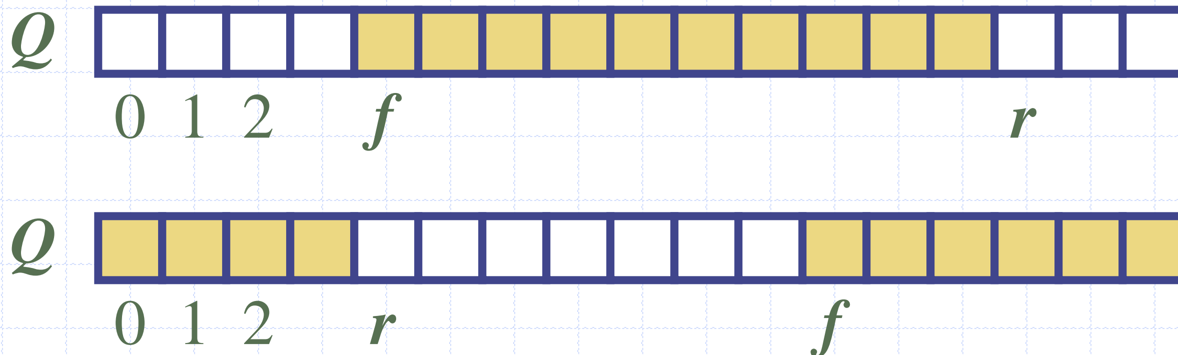


Queue Operations

- Use n to determine size and emptiness

Algorithm *size*()
  return n

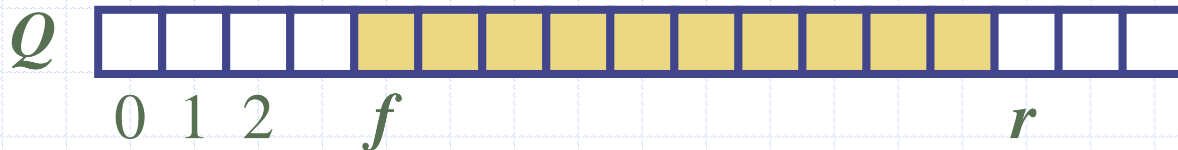
Algorithm *empty*()
  return $(n = 0)$

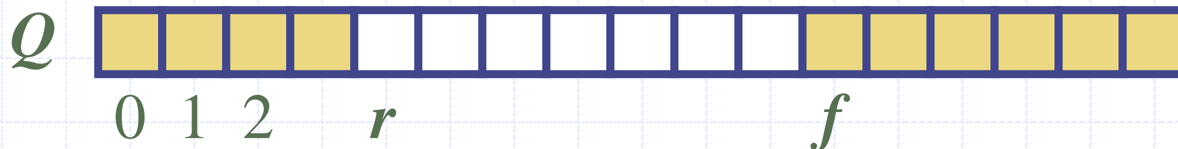


Queue Operations (cont.)

- Operation enqueue throws an exception if the array is full
- This exception is implementation-dependent

Algorithm *enqueue*(o)
  if *size*() = N **then**
    throw *QueueFull*
  else
    $Q[r] \leftarrow o$
    $r \leftarrow (r + 1) \bmod N$
    $n \leftarrow n + 1$

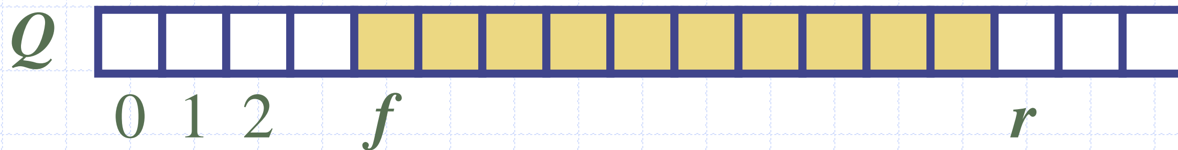


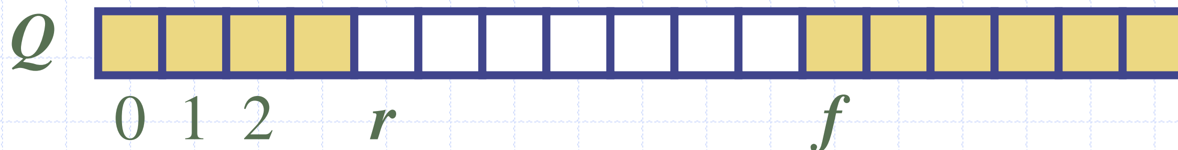


Queue Operations (cont.)

- Operation dequeue throws an exception if the queue is empty
- This exception is specified in the queue ADT

Algorithm *dequeue*()
  if *empty*() **then**
    throw *QueueEmpty*
  else
    $f \leftarrow (f + 1) \bmod N$
    $n \leftarrow n - 1$



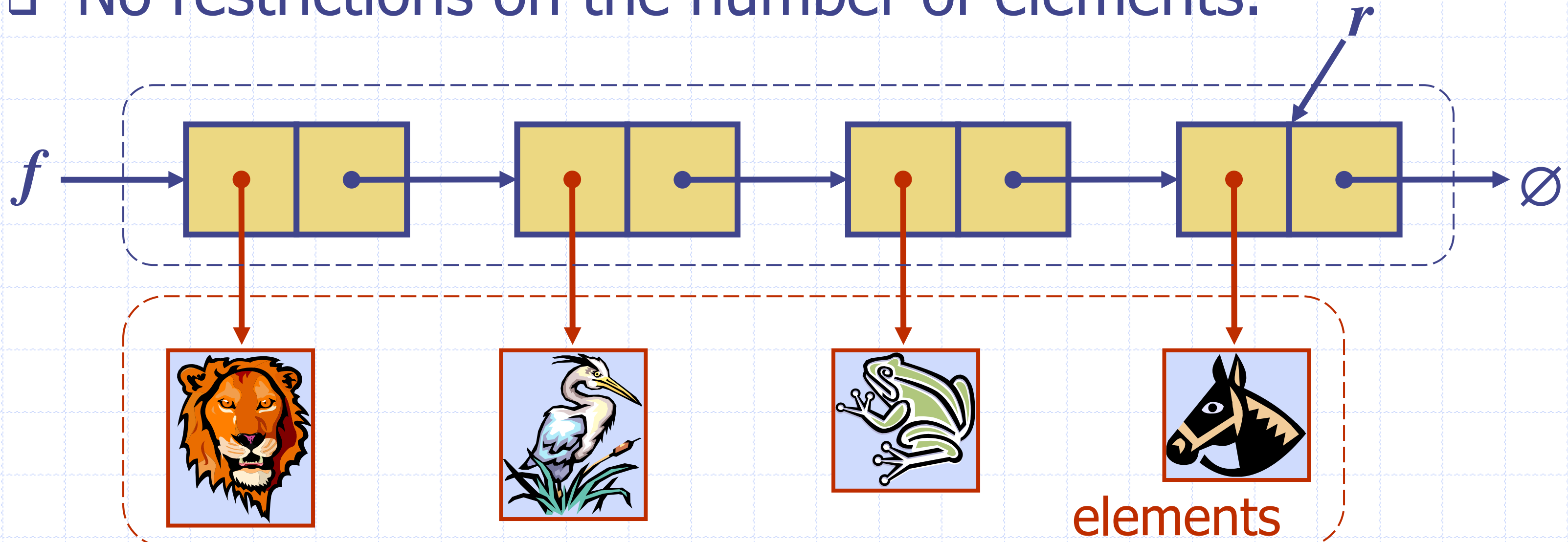


Performance and Limitations

- Performance
  - Let n be the number of elements in the queue
  - The space used is at least n
  - Each operation runs in time $O(1)$
- Limitations
  - The maximum size of the queue must be defined a priori and cannot be changed
  - Trying to enqueue an element into a full queue causes an implementation-specific exception

Linked List-based Queue

- We can implement a queue with a singly linked list
  - The front element is stored at the first node
  - The rear element is stored at the last node
- The space used is $O(n)$ and each operation of the Queue ADT takes $O(1)$ time
- No restrictions on the number of elements.

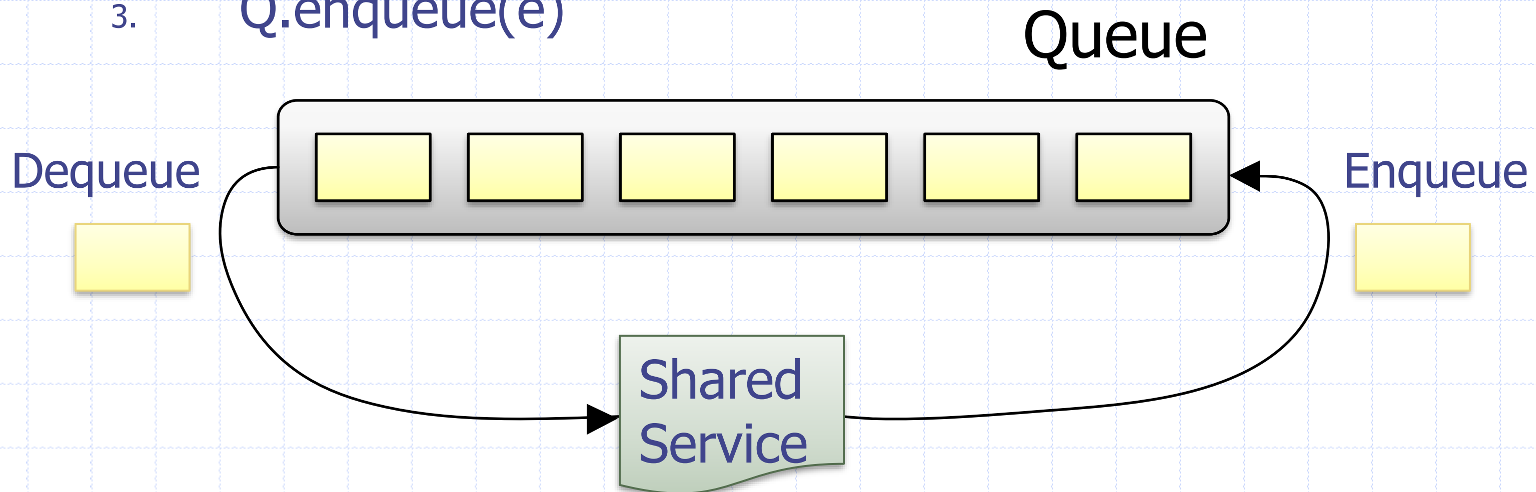


Applications of Queues

- Direct applications
  - Waiting lists, bureaucracy
  - Access to shared resources (e.g., printer)
  - Multiprogramming
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Application: Round Robin Schedulers

- We can implement a round robin scheduler using a queue Q by repeatedly performing the following steps:
  1. e = Q.front(); Q.dequeue()
  2. Service element e
  3. Q.enqueue(e)



The Deque ADT

- The Deque (double-ended queue) ADT stores arbitrary objects
- Insertions and deletions can happen at either the front or the back
- Main dequeue operations:
  - insertFront(object): inserts an element at the front of the deque
  - insertBack(object): inserts an element at the back of the deque
  - eraseFront(): removes the element at the front of the deque
  - eraseBack(): removes the element at the back of the deque
- Auxiliary deque operations:
  - object front(): returns the element at the front without removing it
  - object back(): returns the element at the back without removing it

The Deque ADT (continued)

- integer size(): returns the number of elements stored
- boolean empty(): indicates whether no elements are stored

- Exceptions
  - Attempting the execution of eraseFront, eraseBack, front, or back an empty deque throws a DequeEmpty

STL deque class

- The Standard Template Library (STL) provides an implementation of a deque.
- To declare a deque of strings:

```
#include <deque>
std::deque<string> myDeque;
```

- STL's deque interface is similar to the previous one, but
  - insertFront, insertBack, eraseFront, and eraseBack are called push_front, push_back, pop_front, and pop_back, respectively.
  - Executing pop_front, pop_back, front, or back on an empty deque results in *undefined behavior*.

Deque Interface for Doubly-Linked List Implementation

```
template <typename E>
class LinkedDeque {
public:
   LinkedDeque();
   int size() const;
   bool empty() const;
   const E& front() const
      throw(DequeEmpty);
   const E& back() const
      throw(DequeEmpty);
   void insertFront(const E& e);
   void insertBack(const E& e);
   void removeFront()
      throw(DequeEmpty);
   void removeBack()
      throw(DequeEmpty);
private:
   DLinkedList D;
   int n;
};
```