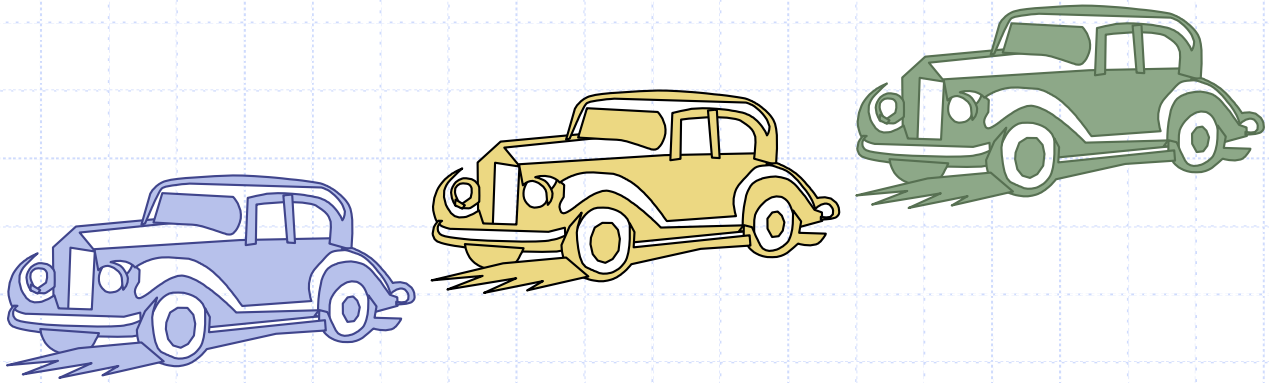
What should we learn from this class?

- Not Knowledge
  - How to use C++
  - How to use C++ STL
  - Understand the concept of stack, shortest-path algorithms, etc
  - ”I know many things” – not important
- But Design
  - Can you design something like C++ STL?
  - Are you able to develop your algorithms that are efficient?
  - Ask: what is missing in you, when you make all the concepts, methods, new algorithms in the textbook?
  - “I can design something” – Very important

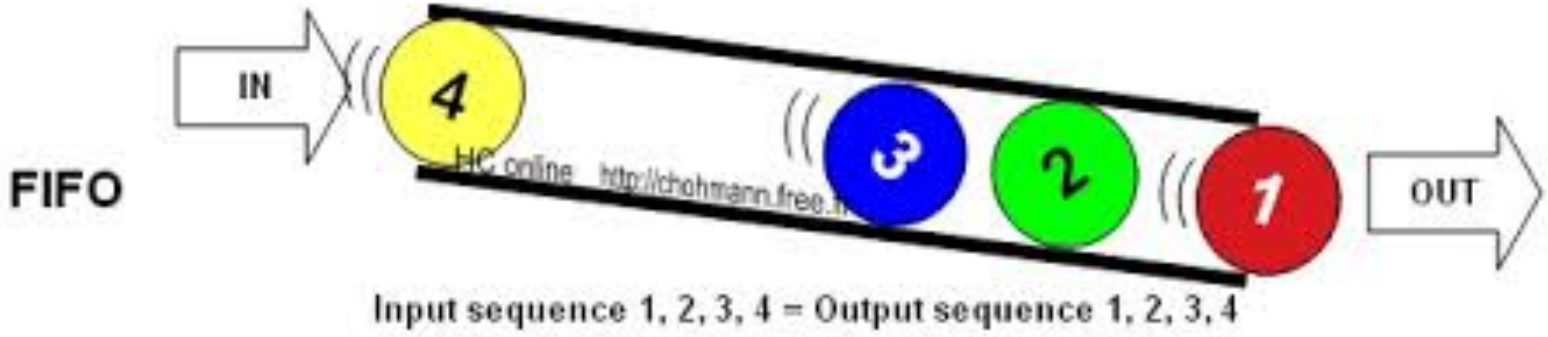
Queues



Overview and Reading

- Reading: Chapters: 5.2 and 5.3

- First-In-First-Out Data Structure



The Queue ADT (§5.2)

- The Queue ADT stores arbitrary objects
- Insertions and deletions follow the first-in first-out 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

Queue Interface in C++

- C++ interface corresponding to our Queue ADT
- Requires the definition of exception QueueEmpty
- Often dequeue returns an object

```
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);
};
```

Example

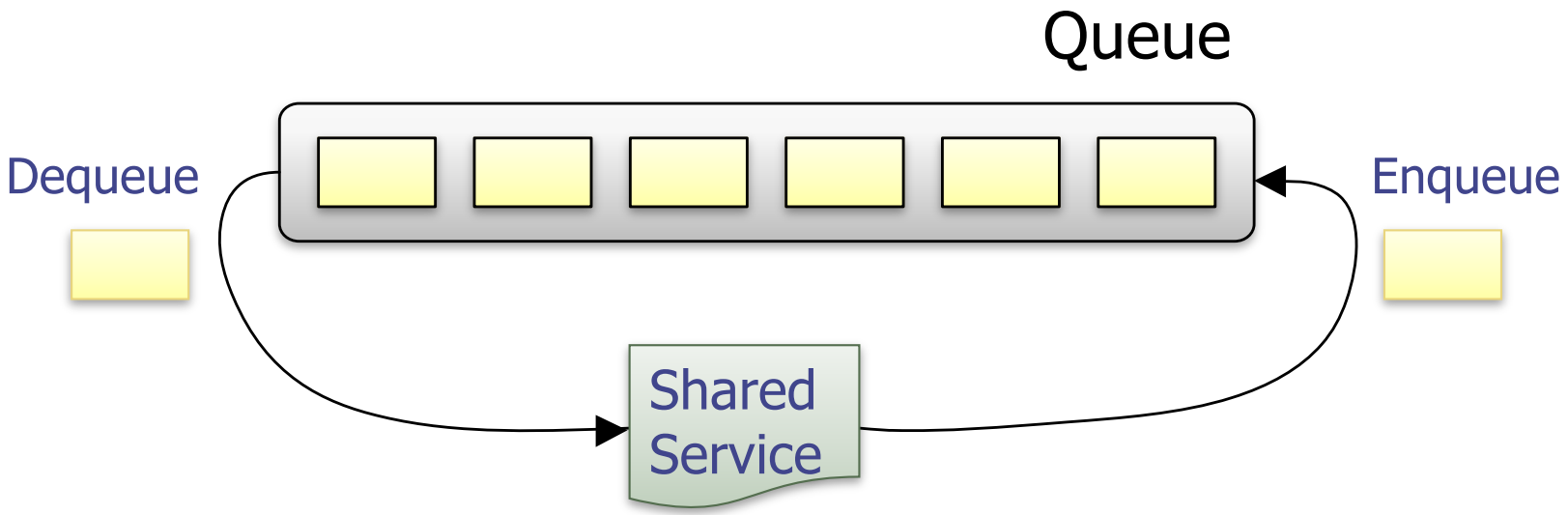
Operation	*Output*	*Q*
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
dequeue()	–	(3)
enqueue(7)	–	(3, 7)
dequeue()	–	(7)
front()	*7*	(7)
dequeue()	–	()
dequeue()	*"error"*	()
empty()	*true*	()
enqueue(9)	–	(9)
enqueue(7)	–	(9, 7)
size()	*2*	(9, 7)
enqueue(3)	–	(9, 7, 3)
enqueue(5)	–	(9, 7, 3, 5)
dequeue()	–	(7,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

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

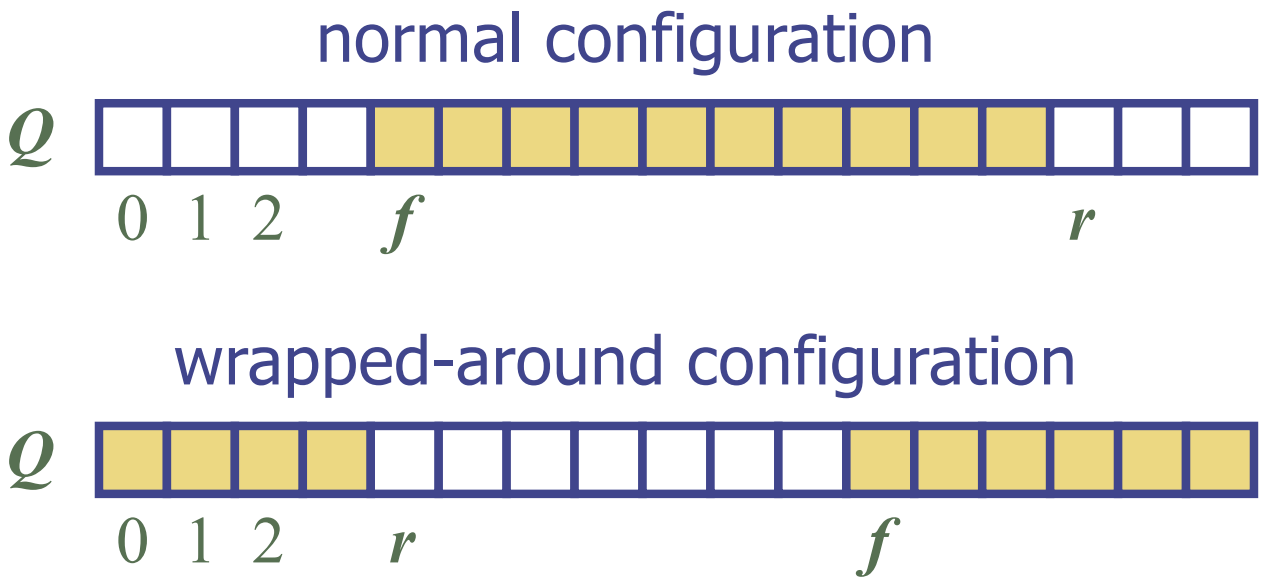
Application: Round Robin Schedulers

- 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)



Array-based Queue

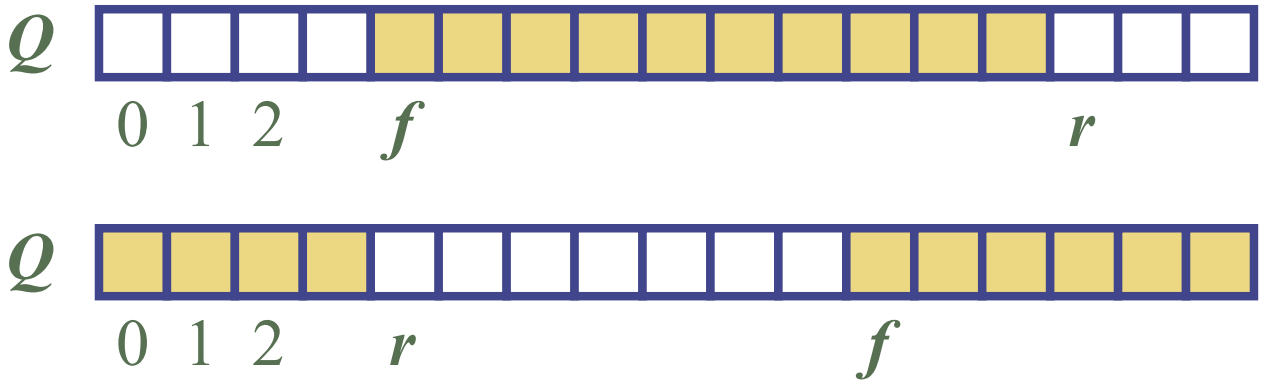
- Use an array of size N in a circular fashion
- Three variables keep track of the front and rear
  - f index of the front element
  - r index immediately past the rear element
  - n number of items in the queue



Queue Operations

- Use *n* to determine size and emptiness

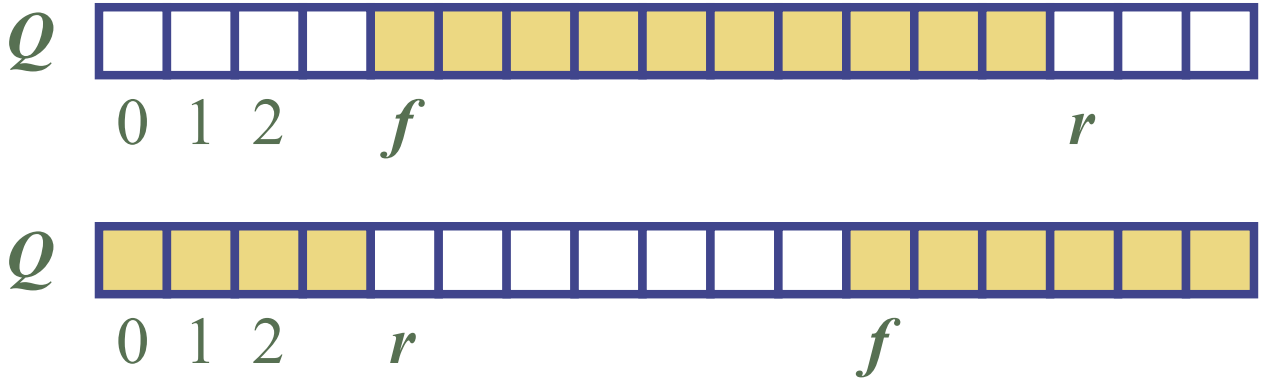
```
Algorithm size()
  return n

Algorithm empty()
  return (n = 0)
```



Queue Operations (cont.)

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

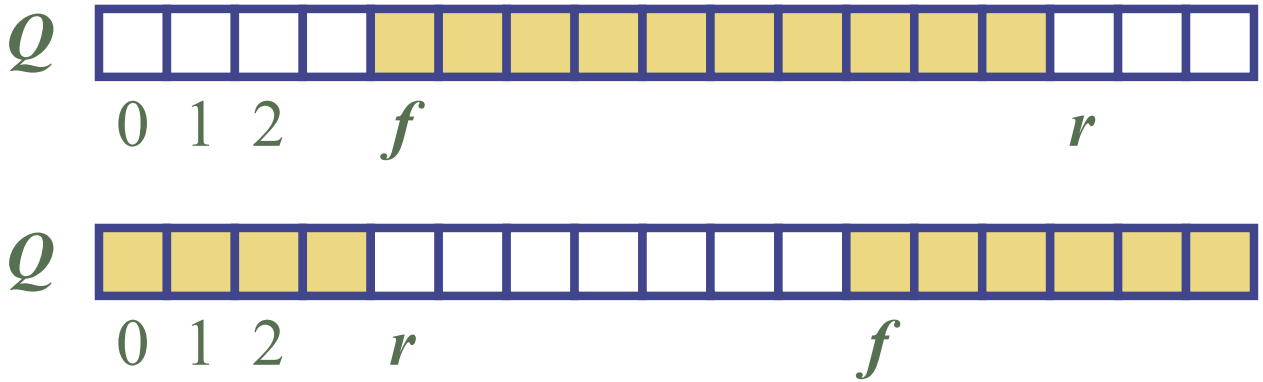
Algorithm *enqueue*(*o*)
  if *size*() = *N* – 1 **then**
    throw *QueueFull*
  else
    Q[*r*] ← *o*
    r ← (*r* + 1) mod *N*
    n ← *n* + 1



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
    $f \leftarrow (f + 1) \bmod N$
    $n \leftarrow n - 1$



Queue in C++ STL

```
#include <queue>
using std::queue;                    // make queue accessible
queue<float> myQueue;                // a queue of floats
```

size(): Return the number of elements in the queue.

empty(): Return true if the queue is empty and false otherwise.

push(e): Enqueue e at the rear of the queue.

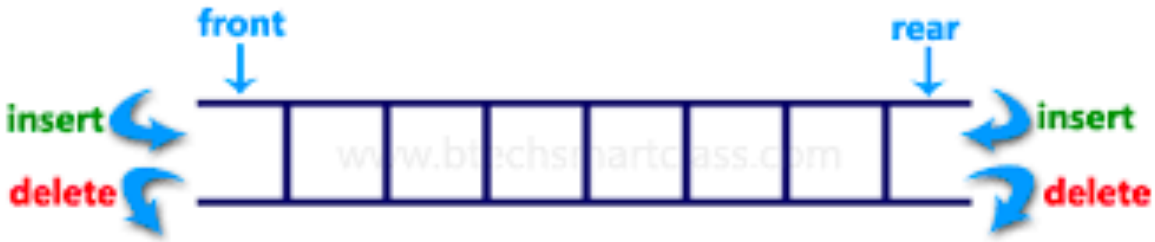
pop(): Dequeue the element at the front of the queue.

front(): Return a reference to the element at the queue's front.

back(): Return a reference to the element at the queue's rear.

Double-Ended Queues (§5.3)

- Pronounce “deck”



Operation	*Output*	*D*
insertFront(3)	–	(3)
insertFront(5)	–	(5,3)
front()	*5*	(5,3)
eraseFront()	–	(3)
insertBack(7)	–	(3,7)
back()	*7*	(3,7)
eraseFront()	–	(7)
eraseBack()	–	()

DEQUE in C++ STL

```
#include <deque>
using std::deque;                    // make deque accessible
deque<string> myDeque;               // a deque of strings
```

size(): Return the number of elements in the deque.

empty(): Return true if the deque is empty and false otherwise.

push_front(e): Insert e at the beginning the deque.

push_back(e): Insert e at the end of the deque.

pop_front(): Remove the first element of the deque.

pop_back(): Remove the last element of the deque.

front(): Return a reference to the deque's first element.

back(): Return a reference to the deque's last element.

How to implement DEQUE?

- Question
  - Which (elementary) data structure are you going to use to implement DEQUE?
    - Array, singly linked list, doubly linked list, circular linked list
  - What happens if you use others?
- Deque by a doubly linked list

Operation	*Time*
size	$O(1)$
empty	$O(1)$
front, back	$O(1)$
insertFront, insertBack	$O(1)$
eraseFront, eraseBack	$O(1)$

Questions?