Queues and Deques

Sections 5.2 to 5.3.3
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:
  - **front**( ) returns the element at the front without removing it.
  - **size**( ): returns the number of elements stored.
  - **empty**( ): indicates whether no elements are stored.
- **Exceptions**
  - Attempting the execution of dequeue or front on an empty queue throws an `QueueEmpty` error.

© 2010 Goodrich, Tamassia
### Example

<table>
<thead>
<tr>
<th>Operation</th>
<th>Output</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>enqueue(5)</td>
<td>–</td>
<td>(5)</td>
</tr>
<tr>
<td>enqueue(3)</td>
<td>–</td>
<td>(5, 3)</td>
</tr>
<tr>
<td>dequeue()</td>
<td>–</td>
<td>(3)</td>
</tr>
<tr>
<td>enqueue(7)</td>
<td>–</td>
<td>(3, 7)</td>
</tr>
<tr>
<td>dequeue()</td>
<td>–</td>
<td>(7)</td>
</tr>
<tr>
<td>front()</td>
<td>7</td>
<td>(7)</td>
</tr>
<tr>
<td>dequeue()</td>
<td>–</td>
<td>()</td>
</tr>
<tr>
<td>dequeue()</td>
<td>“error”</td>
<td>()</td>
</tr>
<tr>
<td>empty()</td>
<td>true</td>
<td>()</td>
</tr>
<tr>
<td>enqueue(9)</td>
<td>–</td>
<td>(9)</td>
</tr>
<tr>
<td>enqueue(7)</td>
<td>–</td>
<td>(9, 7)</td>
</tr>
<tr>
<td>size()</td>
<td>2</td>
<td>(9, 7)</td>
</tr>
<tr>
<td>enqueue(3)</td>
<td>–</td>
<td>(9, 7, 3)</td>
</tr>
<tr>
<td>enqueue(5)</td>
<td>–</td>
<td>(9, 7, 3, 5)</td>
</tr>
<tr>
<td>dequeue()</td>
<td>–</td>
<td>(7, 3, 5)</td>
</tr>
</tbody>
</table>
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:
  ```cpp
  #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

![Diagram showing normal and wrapped-around configurations of an array-based queue.](image)
Queue Operations

- Use $n$ to determine size and emptiness

Algorithm \textit{size}()

\begin{algorithm}
  \textbf{return} $n$
\end{algorithm}

Algorithm \textit{empty}()

\begin{algorithm}
  \textbf{return} ($n = 0$)
\end{algorithm}
Queue Operations (cont.)

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

Algorithm `enqueue(o)`

```plaintext
if size() = N then
    throw QueueFull
else
    Q[r] ← o
    r ← (r + 1) mod N
    n ← n + 1
```

© 2010 Goodrich, Tamassia
Queue Operations (cont.)

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

Algorithm `dequeue()`

```plaintext
if empty() then
    throw QueueEmpty
else
    f ← (f + 1) mod N
    n ← n - 1
```

Diagram:

```
Q: [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
   0 1 2  f  r
```

```
Q: [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
   0 1 2  r  f
```
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. \( e = Q.\text{front}(); \) \( Q.\text{dequeue}() \)
  2. Service element \( e \)
  3. \( Q.\text{enqueue}(e) \)
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
- Auxiliary deque operations:
  - `eraseFront()`: removes the element at the front of the deque
  - `eraseBack()`: removes the element at the back of the deque

© 2019 Shermer
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:
  ```cpp
  #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;
};