Queues

- FIFO queue ADT
- Examples using queues
  - reading character string in order
  - recognize palindromes
- Queue implementations
  - LL pointer based
  - List ADT based
  - array based
  - tradeoffs

The Abstract Data Type Queue

- Another common linear data structure similar to the stack
- Queue is an ADT with following properties
  - elements are kept in their order of arrival
  - new items enter at the back, or rear, of the queue
  - items leave from the front of the queue
- Thus queue has first-in, first-out (FIFO) property
  - nicely models several real-world processes
    - line to buy movie tickets, or queue jobs and print requests

The Abstract Data Type Queue

- ADT queue operations
  - Create an empty queue
  - Destroy a queue
  - Determine whether a queue is empty
  - Add a new item to the queue
  - Remove the item that was added earliest
  - Retrieve the item that was added earliest

The Abstract Data Type Queue

- Operation Contract for the ADT Queue
  - isEmpty():boolean {query}
  - enqueue(in newItem:QueueItemType) throw QueueException
  - dequeue() throw QueueException
  - dequeue(out queueFront:QueueItemType) throw QueueException
  - getFront(out queueFront:QueueItemType) {query} throw QueueException
The Abstract Data Type Queue

**Example 1: Ordering Character String**

- A queue can retain characters in the order in which they are typed
  ```cpp
  aQueue.createQueue()  
  while (not end of line)  
    {  Read a new character ch 
      aQueue.enqueue(ch)  
    }  // end while
  ```
- Once the characters are in a queue, the system can process them as necessary

**Example 2: Recognizing Palindromes**

- A palindrome is a string of characters that reads the same backwards and forwards
  - RADAR, MADAM, EYE, etc.
- Observations
  - stack reverses the order of occurrences
  - queue preserves the order of occurrences
- A palindrome stored in both stack and queue will display a match when retrieved

**Example 2: Recognizing Palindromes**

- A nonrecursive recognition algorithm for palindromes
  - traverse character string from left to right
  - insert each character into both a queue and a stack
  - compare the characters at the front of the queue and the top of the stack

---

**Figure 7-2** Some queue operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Queue after operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>aQueue.createQueue()</td>
<td></td>
</tr>
<tr>
<td>aQueue.enqueue(5)</td>
<td>5</td>
</tr>
<tr>
<td>aQueue.enqueue(2)</td>
<td>5 2</td>
</tr>
<tr>
<td>aQueue.enqueue(7)</td>
<td>5 2 7</td>
</tr>
<tr>
<td>aQueue.getFront(queueFront)</td>
<td>2 7 (queueFront is 5)</td>
</tr>
<tr>
<td>aQueue.dequeue(queueFront)</td>
<td>2</td>
</tr>
<tr>
<td>aQueue.dequeue(queueFront)</td>
<td>7</td>
</tr>
<tr>
<td>aQueue.dequeue(queueFront)</td>
<td>7</td>
</tr>
</tbody>
</table>

---

see C7-palin.cpp
Implementations of the ADT Queue

- Linked list based queue implementation
  - can maintain pointers to front and back of Queue
  - circular linked list with one external reference also possible
- Using ADT List class to implement queue
  - possible less efficient, but simple
- An array-based queue implementation
  - problem of rightward-drift

Operations in LL Implementation

List Based Queue Implementation

- Queue operations map well to ADT List operations
  - enqueue(item) → insert(getLength()+1, item)
  - dequeue() → remove(1)
  - getFront(qfront) → retrieve(1, qfront)
- We can built the queue ADT as a wrapper over the List ADT

Linked List Implementations

Figure 7-4 A pointer-based implementation of a queue: (a) a linear linked list with two external pointers; (b) a circular linear linked list with one external pointer

Figure 7-5 Inserting an item into a nonempty queue

Figure 7-6 Inserting an item into an empty queue:
(a) before insertion;
(b) after insertion

Figure 7-7 Deleting an item from a queue of more than one item

see C7-QueueP.cpp
An Array-Based Implementation

• Using arrays is slightly more complex
  — naïve implementation causes rightward drift
  — queue appears full even when array does not hold MAX_QUEUE-1 elements
• Solutions to rightward drift
  — always copy array elements to left – expensive
  — maintain circular array – how to detect queue full/empty?

Circular Array Implementation

• Problem:
  — front == (back + 1) is true for both queue full & empty
• Solution:
  — use integer counter to hold size of queue
  — update on each enqueue/dequeue

Array Implementation Variations

• Use a flag isFull to distinguish between the full and empty conditions
• Declare MAX_QUEUE + 1 locations for the array items, but use only MAX_QUEUE of them for queue items

Initialize the queue,
front = 0, back = MAX_QUEUE - 1,
count = 0

Inserting into a queue
back = (back + 1) % MAX_QUEUE;
items[back] = newItem;
++count;

Deleting from a queue
front = (front + 1) % MAX_QUEUE;
--count;

see C7-QueueA.cpp
Comparing Implementations

- Static arrays Vs. dynamically allocated LLs
  - enqueue operation cannot add item if array is full
  - no size restriction with LL (unless memory full)

- LL Vs List bases array implementations
  - LL-based implementation is more efficient
  - ADT list approach reuses already implemented class
    - much simpler to write
    - saves programming time

A Summary of Position-Oriented ADTs

- Position-oriented ADTs
  - List
  - Stack
  - Queue

- Stacks and queues
  - Only the end positions can be accessed

- Lists
  - All positions can be accessed

Summary

- ADT queue has first-in, first-out (FIFO) behavior
- Circular array eliminates the problem of rightward drift in array-based implementation
- To distinguish between the queue-full and queue-empty conditions in a circular array
  - count the number of items in the queue
  - use an isFull flag
  - leave one array location empty

- LL and List ADT based implementations possible