Queues

孫雅麗
台灣大學 資管所
Outline

■ What is a “queue”?
  • Concept
  • Operations – enqueue and dequeue

■ Simple applications of “queue”
Queue

- A queue is an ordered list that can be modified only by removing items from one end (the FRONT) and by adding them to the other end (the BACK).

- Queues are also referred to as FIRST-IN-FIRST-OUT (FIFO) lists.
Queue (cont’d)

- Queues are a fundamental data structure to implement process management in an operating system.

![Diagram of queue system with labels: "READY" queue, "SUSPENDED" queue, "BLOCKED" queue, CPU, I/O, and "Time-sharing"]
ADT Queue Operations (cont'd)

createQueue()
 // Creates an empty queue.

destroyQueue()
 // Destroys a queue.

isEmpty()
 // Determines whether a queue is empty.
ADT Queue Operations (cont’d)

enqueue(in newItem: QueueItemType)
// Inserts newItem at the back of a queue. Success
// indicates whether the insertion was successful.

dequeue(out queueFront: QueueItemType)
// Removes into queueFront and then removes the
// front of a queue; that is, retrieves and removes the
// item that was added earliest.
ADT Queue Operations (cont’d)

getFront (out queueFront: QueueItemType)

// Removes into queueFront the front of a queue;
// that is, retrieves the item that was added earliest.
## ADT Queue Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Queue after operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q.CreateQueue()</td>
<td></td>
</tr>
<tr>
<td>Q.QueueInsert(5, Success)</td>
<td>5</td>
</tr>
<tr>
<td>Q.QueueInsert(2, Success)</td>
<td>5 2</td>
</tr>
<tr>
<td>Q.QueueInsert(7, Success)</td>
<td>5 2 7</td>
</tr>
<tr>
<td>Q.GetQueueFront(QueueFront, Success)</td>
<td>5 2 7</td>
</tr>
<tr>
<td>Q.QueueDelete(Success)</td>
<td>2 7</td>
</tr>
<tr>
<td>Q.QueueDelete(QueueFront, Success)</td>
<td>7</td>
</tr>
</tbody>
</table>

*Figure 7-1*

Some queue operations
Application#1: Reading Characters

- When you enter characters at a keyboard, the system must retain them in the order in which you typed them.
- It could use a queue for this purpose.

```java
// read a string of characters from a single line of input into a queue
aQueue.createQueue()
while (not end of line)
{
  Read a new character ch
  aQueue.enqueue(ch)
} // end while
```
Reading a String of Characters

- A queue can retain characters in the order in which they are typed
  
aQueue.createQueue()
  
  while (not end of line) // read a string of characters from a single line of input into a queue
  
  { Read a new character ch
    aQueue.enqueue(ch)
  }
  
  } // end while

- Once the characters are in a queue, the system can process them as necessary
Application#2: Converting an "integer" string into its value

// convert digits in a queue aQueue into a decimal integer n
// get first digit, ignoring any leading blanks.
    do {  aQueue.enqueue(ch)
         } while (ch is a blank)
// Assertion: ch contains first digit
// compute n from digits in queue
n = 0; done = false;
    do {
        n = 10 * n + integer that ch represents
        if (!aQueue.isEmpty())
            aQueue.dequeue(ch);
        else  done = true
    } while (!done and ch is a digit) // n is result

• x = 247 + 968

2 4 7

front

• 247 = 10 * (10 * 2 + 4) + 7
Application #3: Recognizing Palindromes (1/3)

- Palindrome: a string of characters that reads the same from left to right as it does from right to left, e.g., dod, pop, etc.

- Use a queue and a stack
  - Compare the front of the queue and the top of the stack

Figure 7-2
The results of inserting a string into both a queue and a stack
Application #3: Recognizing Palindromes (2/3)

IsPal (string) {
    // Determines whether string is a palindrome.
    // create an empty queue and empty stack
    aQueue.createQueue()
    aStack.createStack()
    // insert each character of the string into both
    // the queue and the stack
    length = length of string
    for (i = 1 through length)
    {
        nextChar = i\textsuperscript{th} character of string
        aQueue.enqueue (nextChar)
        aStack.push (nextChar)
    } // end for
Application #3: Recognizing Palindromes (3/3)

// compare the queue characters with the stack characters
charactersAreEqual = true
while (aQueue is not empty and charactersAreEqual)
{
    aQueue.getFront(queueFront)
    aStack.getTop(stackTop)
    if (queueFront equals stackTop)
    {
        aQueue.dequeue()
        aStack.pop()
    }
    else
    {
        charactersAreEqual = false
    }
} // end while
return charactersAreEqual
QueueException

#include <exception>
#include <string>
using namespace std;
class QueueException: public exception
{
    public:
        QueueException(const string &message="")
            : exception(message.c_str())
        {
        }
}; // end QueueException
Implementation of the ADT Queue

- Pointer-based
- Array-based
- ADT list
A Linear Linked List with Two External Pointers (1/4)
Enqueue: Insertion into an empty queue (2/4)

1. `frontPtr = newPtr;` // points to the front

2. `backPtr = newPtr;` // points to the rear
Enqueue: Insertion into a nonempty queue (3/4)

1. newPtr->next = NULL;
2. backPtr->next = newPtr;
3. backPtr = newPtr;
Dequeue: deletion from a nonempty queue (4/4)

1. tempPtr = frontPtr; // points to the front
2. frontPtr = frontPtr->next; // points to the next
3. tempPtr->next = NULL;
4. Delete tempPtr;
A circular linear linked list with one external pointer (1/3)
Enqueue to an empty circular linear linked list (2/3)

---

**Figure 7-5**

Inserting an item into an empty queue: (a) before insertion; (b) after insertion

1. `newPtr->next = newPtr;` // points to the front
2. `backPtr = newPtr;` // points to the rear
Dequeue from a non-empty circular linear linked list (3/3)

1. frontPtr = backPtr -> next; //points to the front
2. backPtr -> next = frontPtr -> next; // points to the new front
3. frontPtr -> next = NULL; delete frontPtr
// ******************************************************************************************
// Header file queueP.h for the ADT queue.
// Pointer-based implementation.
// ******************************************************************************************
#include “QueueException.h”
typedef desired-type-of-queue-item queueItemType;

class Queue
{
public:
    // constructors and destructor:
    Queue();                           // default constructor
    Queue(const Queue & q);    // copy constructor
    ~Queue();                         // destructor

    // queue operations:
    bool isEmpty() const;
    // Determines whether a queue is empty.
    // Precondition: None.
    // Postcondition: Returns true if the queue is empty; otherwise returns false.
void enqueue(QueueItemType newItem) throw (QueueException);
// inserts an item at the back of a queue.
// Precondition: newItem is the item to be inserted.
// Postcondition: If insertion was successful, newItem
// is at the back of the queue.

void dequeue() throw (QueueException);
// Dequeues the front of a queue.
// Precondition; None.
// Postcondition: If the queue was not empty, the item
// that was added to the queue earliest is deleted and
void dequeue (QueueItemType& queueFront) throw (QueueException);

// Retrieves and deletes the front of a queue.
// Precondition: None.
// Postcondition: If the queue was not empty, queueFront contains the item that was added to the queue earliest, the item is deleted.
```cpp
void getFront(QueueItemType& queueFront) const throw (QueueException);

// Retrieves the item at the front of a queue.
// Precondition: None.
// Postcondition: If the queue was not empty, queueFront contains
// the item that was added to the queue earliest.

private:
    struct QueueNode
    {
        QueueItemType item;
        QueueNode *next;
    };
    QueueNode *frontPtr, *backPtr; // two pointers
}; // end class
```
// **************************************************
// Implementation file QueueP.cpp for the ADT queue.
// Pointer-based implementation.
#include "QueueP.h"   // header file
#include <cstdlib>
#include <cassert>

Queue::Queue() : backPtr(NULL), frontPtr(NULL)
{
} // end default constructor

Queue :: Queue(const Queue & Q)
{
   // Implementation left as an exercise (Exercise 4)
} // end copy constructor

Queue ::~ Queue() // Assertion: frontPtr, backPtr equal NULL
{
    while (! isEmpty())
        dequeue();
} // end destructor
Queue :: isEmpty() const
{
  return bool(backPtr == NULL);
} // end QueuesEmpty

void Queue :: enqueue(QueueItemType newItem)
{
  // create a new node
  QueueNode *newPtr = new QueueNode;
  if (newPtr == NULL) // check allocation
    throw QueueException("QueueException: enqueue cannot allocate memory");
  else
    {
      // allocation successful; set data portion of newnode
      newPtr->item = newItem; // = overloading
      newPtr -> next = NULL;
    }

// insert the new node
if (isEmpty()) // insertion into empty queue
    frontPtr = newPtr;
else // insertion into nonempty queue
    backPtr->next = newPtr;
    backPtr = newPtr; // new node is at back
} // end if
} // end enQueue
void Queue::dequeue()
{
if (isEmpty())
    throw QueueException("QueueException: queue empty, cannot dequeue");
else
{
// queue is not empty; remove front
QueueNode * tempPtr = frontPtr;
if (frontPtr == backPtr) { // after deletion, queue becomes empty
    frontPtr = NULL; // yes, one node in queue
    backPtr = NULL;
} else
    frontPtr = frontPtr ->next; // the second front node becomes front
    tempPtr->next = NULL; // defensive strategy
    delete tempPtr;
} // end if
} // end queueDelete
void Queue::dequeue (QueueItemType& queueFront)
{
    if (isEmpty())
        throw QueueException("QueueException: queue empty, cannot dequeue");
    // queue is not empty; retrieve front
    queueFront = frontPtr->item;
    dequeue();  // use the previously-defined operation
} // end if
} // end deQueue
Pointer-Based ADT Queue (10/11)

```cpp
void Queue::getFront(QueueIfrontType & queueFront) const
{
    if (isEmpty())
        throw QueueException("QueueException: empty queue, cannot getFront");

    else // queue is not empty; retrieve front
    
        queueFront = frontPtr->item;

} // end if
} // end getFront
```
Example (11/11)

#include “QueueP.h”

int main()
{
    Queue aQueue;

    aQueue.enqueue(15);

    ...

Array-Based ADT Queue (cont’d)

- Rightward drift
- Sliding (moving) window
Array-Based ADT Queue: implementation

const int MAX_QUEUE = maximum-size-of-queue;
typedef desired-type-of-queue-item QueueItemType;

QueueItemType items[MAX_QUEUE];
int front;
int back;
Circular Implementation of a Queue

• To eliminate rightward drift (fall off the array boundary)
• Advance clockwise
• Wraparound

-> When advances past MAX_QUEUE-1, wrap around to 0.
Circular Implementation of a Queue (cont’d)

- front points to the first item
- back points to the last item
- Initially, front and back point to -1 (NULL).

Figure 7-9
The effect of some operations on the queue in Figure 7-8
Circular Implementation of a Queue: How to detect queue-empty?

Situation 1: front passes back: queue is empty.

Queue with single item

Dequeue

Situation 2: back catches up to front: queue is full.

Queue with single empty slot

enqueue

Cannot distinguish queue empty from queue full by “front is one slot ahead of back”!!
How to detect queue-empty and queue-full?

- Use only $\text{MAX\_QUEUE-1}$
- $\text{front}$ points to one slot ahead of the first item (not the first item); $\text{back}$ points to the last item.
- Queue is full: $\text{front} == (\text{back}+1) \mod \text{MAX\_QUEUE}$
- Queue is empty: $\text{front} == \text{back}$
Array-Based ADT Queue (cont'd)

// ***************************************************
// Header file QueueP.h for the ADT queue.
// Array-based implementation.
// ***************************************************
const int MAX_QUEUE = maximum-size-of-queue;
typedef desired-type-of-queue-item queueItem;

class queueClass
{
    public:
        // constructors and destructor:
        Queue(); // default constructor
        // copy constructor and destructor are supplied by the compiler
Array-Based ADT Queue (cont’d)

// queue operations:
bool isEmpty() const;
void enqueue(QueueItem newItem) throw(QueueException);
void dequeue() throw(QueueException);
void dequeue(QueueItem & queueFront) throw(QueueException);
void getFront(QueueItemType & queueFront) const
  throw(QueueException);

private:
  QueueItemType items[MAX_QUEUE];
  int front; // front points to the first item
  int back; // Back points to the last item
  int count;
}; // end class

// End of header file.
Array-Based ADT Queue (cont'd)

// ****************************************************
// Implementation file QueueA.cpp for the ADT queue.
// Circular array-based implementation.
// The array has indexes to the front and back of the
// queue. A counter tracks the number of items currently in the queue.
// ****************************************************

#include "QueueA.h"  // header file

queueClass::Queue(): front(0), back(MAX_QUEUE-1), count(0)
{
}
} // end default constructor
Array-Based ADT Queue (cont’d)

bool Queue::isEmpty() const
{
    return bool(count == 0);
} // end isEmpty

void Queue::enqueue(QueueItemType newItem)
{
    if (count == MAX_QUEUE)
        throw QueueException("QueueException: queue full on enqueue");
    else { // queue is not full; insert item
        back = (back+1) % MAX_QUEUE; // Back points to the last item
        items[back] = newItem;
        ++count;
    } // end if
} // end enqueue
void Queue :: dequeue(queueItemType& queueFront) 
{ 
    if (isEmpty()) 
        throw QueueException("QueueException: queue is empty, cannot dequeue");
    else { // queue is not empty; remove front 
        queueFront = items[front]; // front points to the first item 
        front = (front+1) % MAX_QUEUE;
        --count;
    } // end if 
} // end QueueDelete
void Queue :: getFront(queueItemType& queueFront)
{
    if (isEmpty())
        throw QueueException("QueueException: queue is empty, cannot dequeue");
    else  { // queue is not empty; remove front
        queueFront = items[front];
    }
} // end GetQueueFront

// End of implementation file.
ADT List Implementation

- enqueue (newItem);
  
  insert(getLength+1, newItem);

- dequeue ();
  
  remove(1);

- getFront(queueFront);
  
  retrieve(1, &queueFront);
ADT List Implementation (cont’d)

// ***************************************************
// Header file QueueL.h for the ADT queue.
// ADT list implementation.
// ***************************************************
#include "ListP.h"  // ADT list operations
#include "QueueException.h"
typedef ListItemType QueueItemType;

class Queue
{
    public:
    // constructors and destructor:
    Queue();                          // default constructor
    Queue(const Queue& q);   // copy constructor
    ~Queue();                       // destructor
ADT List Implementation (cont’d)

bool isEmpty() const;
    void enqueue(QueueItem newItem) throw(QueueException);
    void dequeue() throw(QueueException);
    void dequeue(QueueItem & queueFront) throw(QueueException);
    void getFront(QueueItemType & queueFront) const
        throw(QueueException);

private:
    listClass L; // list of queue items
}; // end class

// End of header file.
void Queue::enqueue(QueueItemType newItem)
{
    try
    {
        aList.insert(aList.getLength()+1, newItem);
    } // end try
    catch (ListException e)
    {
        throw QueueException("QueueException: cannot enqueue");
    } // end catch
} // end enqueue
List: insert with exceptions

```cpp
void List::insert(int index, listItemTypeNewItem)
{
    if (size >= MAX_LIST)
        throw ListException("ListException: List full on insert");
    if (index >=1 && index <= size +1)
    {
        for (int pos = size; pos >= index; --pos)
            items[translate(pos+1)] = items[translate(pos)] ;
        items[translate(index)] = newItem;
        ++size;
    } else throw
        ListIndexOutOfRangeException("ListIndexOutOfRangeException: Bad index on insert");
} // end insert
```
Application: Simulation

- A technique for *modeling the behavior* of both natural and human-made systems

- **Goal**
  - Generate *statistics* that summarize the performance of an existing system
  - *Predict* the performance of a proposed system
  - Obtain in-depth understanding (close-up) of the behavior of the system

- **Example**
  - A simulation of the behavior of a bank

- “How to write a simulation program?”
The End 😊
Homework 2008

- Pages 380-383 - #10, #13, #15 and #2 (programming problem)