Queues

Gaddis 18.4, 18.6

Molly A. O'Neil
CS 2308 :: Spring 2016
The Queue ADT

- A **queue** is an *abstract data type* that stores a collection of elements of the same type
  - The elements of a queue are accessed in FIFO (first in, first out) order
  - No random access to other elements
    - Can only add element at back, can only access element at front
- Examples: People in line at the bank; print jobs sent to a shared printer
Queue Operations

- The queue ADT supports the following operations:
  - **enqueue**: add a value onto the rear of the queue (at the end of the line)
    - If it's a size-limited queue, first make sure queue isn't full
  - **dequeue**: remove the value at the front of the queue ("Next!")
    - Depending on the implementation, sometimes dequeue returns the value as well
  - **isEmpty**: true if the queue currently contains no elements

- Optionally, a queue may also support the following operations:
  - **peek/front**: returns the value at the front of the queue, without removing it
  - **isFull**: true if the queue is currently full, i.e., has no more space to hold additional elements
Queue Illustrated

```python
q.enqueue(3)
q.enqueue(2)
q.enqueue(5)
q.enqueue(1)
x = q.dequeue()
```

3 2 5 1
Queue Illustrated

```python
q.enqueue(3)
q.enqueue(2)
q.enqueue(5)
q.enqueue(1)
x = q.dequeue()
```

![Queue Diagram]

```
3 2 5 1
```

```
x
```
Queue Illustrated

q.enqueue(3)
q.enqueue(2)
q.enqueue(5)
q.enqueue(1)
x = q.dequeue()

x

y = q.dequeue()
Queue Illustrated

```python
q.enqueue(3)
q.enqueue(2)
q.enqueue(5)
q.enqueue(1)
x = q.dequeue()
y = q.dequeue()
x = 2
y = q.dequeue()
y = 5
```
Queue Illustrated

```
q.enqueue(3)
q.enqueue(2)
q.enqueue(5)
q.enqueue(1)
x = q.dequeue()
y = q.dequeue()
q.enqueue(6)
q.enqueue(4)
z = q.dequeue()
```

![Queue Diagram]

- 5
- 1
- 6
- 4
- 2

y = 2
Queue Illustrated

```python
q.enqueue(3)
q.enqueue(2)
q.enqueue(5)
q.enqueue(1)
x = q.dequeue()
y = q.dequeue()
q.enqueue(6)
q.enqueue(4)
z = q.dequeue()
```
Queue Illustrated

```python
q.enqueue(3)
q.enqueue(2)
q.enqueue(5)
q.enqueue(1)
x = q.dequeue()
y = q.dequeue()
q.enqueue(6)
q.enqueue(4)
z = q.dequeue()
```
Queue Applications

• Queue applications often involve *multiple users or processes and a shared resource*, for example:
  • A queue is used to hold print jobs submitted by users of a shared printer, while the printer services jobs one at a time
    • Submitting a print job == enqueue
    • Printing the next job == dequeue
  • Communications software uses queues to hold data received over networks
    • When data is transmitted to a system faster than it can be processed, it is stored in a queue as it is received

• **Discrete Event Simulation**
  • Models the operation of a system as a sequence of events, each with a timestamp. The events are usually stored in a queue, often sorted/prioritized by timestamp.
Discrete Event Simulation

• For example, you could use *discrete event simulation* to study the queueing behavior of customers in line at the bank, waiting to be served by a bank teller

• Events to model:
  • Customer$_X$ arrives
  • Teller begins service to Customer$_X$

• Randomize customer interarrival times and teller service times (using appropriate probability distribution)

• Choose number of tellers to model

• Sample outputs of simulation:
  • Average wait time per customer
  • Average line length
  • Average teller busy/idle ratio
Queue Terminology

• **Queue overflow**
  • The condition resulting from trying to add an element onto the rear of a full queue

```java
if(!q.isFull())
    queue.enqueue(item);
```

• **Queue underflow**
  • The condition resulting from trying to remove an element from an empty queue

```java
if(!q.isEmpty())
    queue.dequeue();
```
Queue Implementations

• Like stacks and other ADTs, a queue is only defined in terms of the properties of the data and the operations that can be performed on that data.

• There are several data structures we could use to implement a Queue:
  
  • **Linked list** (... once again, you've already done this one!)
    • Enqueue == append node to list
    • Dequeue == remove front node from list and return its value
  
  • **Fixed-size array**
    • Keep track of front and rear indexes; queue has fixed/maximum size and can be full
  
  • **Dynamic array**
    • Keep track of front and rear indexes; re-size/re-allocate when full
A Fixed-Size Array Queue Class

class IntQueue {
  private:
    int *qArray;  // the queue array
    int qSize;    // the array size
    int front;    // the front index in the queue
    int rear;     // the last index in the queue
    int numItems; // the count of items in the queue

  public:
    IntQueue(int);               // constructor
    IntQueue(const IntQueue &);  // copy constructor
    ~IntQueue();                 // destructor
    void enqueue(int);
    int dequeue();
    int dequeue();
    bool isEmpty() const;
    bool isFull() const;
};
Queue: Constructors

```cpp
IntQueue::IntQueue(int size) {
    if(size <= 0)
        size = 1; // or error

    qArray = new int[size];
    qSize = size;
    front = 0;      // where is the front element located?
    rear = -1;      // incremented before adding next element
    numItems = 0;
}

IntQueue::IntQueue(const IntQueue &rhs) {
    qArray = new int[rhs.qSize];
    qSize = rhs.qSize;

    for(int i = 0; i < qSize; i++)
        qArray[i] = rhs.qArray[i];

    front = rhs.front;
    rear = rhs.rear;
}
```
Queue: Destructor

```cpp
IntQueue::~IntQueue() {
    delete [] qArray;
}
```
Array-Based Queue Implementation

- We know we need to track the front and rear indices of the array
  - Add at rear index
  - Remove element at front index
- When an item is dequeued from the front of the array, we **DO NOT** want to shift all elements over to fill the slot vacated by the dequeued item
  - Why not?
- Instead, shift both front and rear indices through the array....
Array-Based Queue Implementation

- Where do we put the next element?!  
  - Wrap index around to the front of the array!
- When is the queue full?  
  - `rear (wrapped) == front`
- When is the queue empty?  
  - `front (wrapped) == rear`
Index Wrapping

• To wrap the rear index back to the front of the array, you can do the following:

```c
if (rear == qSize - 1)
    rear = 0;
else
    rear = rear + 1;
```

• But this is equivalent (when $0 \leq \text{rear} < \text{qSize}$) and cleaner:

```c
rear = (rear + 1) % qSize;
```

• Do the same when advancing the front index.
void IntQueue::enqueue(int item) {
    assert(!isFull()); // queue overflow!

    // calculate the new rear position
    rear = (rear + 1) % qSize;

    // insert the new item
    qArray[rear] = item;

    // update the item count
    numItems++;
}
int IntQueue::dequeue() {
    assert(!isEmpty());    // queue underflow!

    // retrieve the front item
    int item = qArray[front];

    // move front (increment by one)
    front = (front + 1) % qSize;

    // update the item count
    numItems--;

    return item;
}
Queue: isFull & isEmpty

```cpp
bool IntQueue::isFull() {
    return (numItems == qSize);
}

bool IntQueue::isEmpty() {
    return (numItems == 0);
}
```