Part 4. Queues

CS 200 Algorithms and Data Structures

Outline

• Queue?
  • Implementing Queue
  • Comparison implementations

"Grill the Buffs" event 9/16 2011

Queue

• A queue is like a line of people.
• New item enters a queue at its back.
• Items leave a queue from its front.
• First-in, first-out (FIFO) behavior
• Removing and adding are done from opposite ends of structure
• Useful for scheduling (e.g. print queue, job queue)

Operations

• Create an empty queue
• Determine whether a queue is empty
• Add a new item to the queue
• Remove item from the queue (that was added the earliest)
• Remove all items from the queue
• Retrieve item from queue that was added earliest
Queue Operations

- **enqueue**(in newItem: QueueItemType)
  - Add new item at the back of a queue
- **dequeue()**: QueueItemType
  - Retrieves and removes the item at the front of a queue
- **peek()**: queueItemType {query}
  - Retrieve item from the front of the queue. Retrieve the item that was added earliest.
- **isEmpty()**: boolean {query}
- **createQueue()**
- **dequeueAll()**

Outline

- Queue?
  - **Implementing Queue**
    - Comparison implementations

Implementations of Queue

- **Reference-Based** Implementation
- **Array-based** Implementation
- **List based** Implementation
- **java.util.queue** interface

Reference-based Implementation(1)

- Needs
  - **Nodes** with the item and a reference to the next item
  - **two external references** pointing to the first node and the last node.

Reference-based Implementation(2)

- **Single external references**
  - Circular linked list represents a queue
  - The node at the back of the queue references the node at the front

Inserting an item into a nonempty queue

- Step 1. newNode.next = lastNode.next;
- Step 2. lastNode.next = newNode;
- Step 3. lastNode = newNode;
**isEmpty()**

```java
public class QueuerReferenceBased implements QueueInterface {
    private Node lastNode;
    public QueuerReferenceBased() {
        lastNode = null;
    }
    public boolean isEmpty() {
        return lastNode == null;
    }
}
```

**Insert new item into the queue**

```java
public void enqueue(Object newItem) {
    Node newNode = new Node(newItem);
    if (isEmpty()) {
        newNode.next = newNode;
    } else {
        newNode.next = lastNode.next;
        lastNode.next = newNode;
    }
    lastNode = newNode;
}
```

**Inserting a New Item**

- Insert a new item into the **empty queue**

![Diagram of a queue with one item](image)

**Removing an item from queue**

```java
public Object dequeue() throws QueueException {
    if (isEmpty()){
        Node firstNode = lastNode.next;
        if (firstNode == lastNode) {
            lastNode = null;
        } else {
            lastNode.next = firstNode.next;
        }
        return firstNode.item;
    } else { exception handling.. }
}
```

**Peek?**

```java
public Object peek() throws QueueException {
    if (!isEmpty()) {
        Node firstNode = lastNode.next;
        return firstNode.item;
    } else {
        throw new QueueException(your_message);
    }
}
```
Implementing queue with Array

- We need,
  - An array of objects to store items
  - variables to point to the "front" and "back" index of the array

![Queue Diagram](image)

Implementing queue with Array

- Queue is **empty** if back is less than front
- Inserting an item
  - Initially front is 0, back is -1
  - Increment back
  - Place new item in items[back]

- Queue will be **full** if back equals MAX_QUEUE - 1

Solving Rightward Drift Problem (1)

- **Removing** an item
  - Remove item from items[front]
  - Increment front

- **Rightward drift**
  - After a sequence of additions and removals, items in the queue will drift toward the end of the array
  - back can reach MAX_QUEUE - 1 even when the queue contains only few items.

Solving Rightward Drift Problem (2)

- **Circular implementation** of a queue

![Circular Queue Diagram](image)

Solving Rightward Drift Problem (2)

- **Delete**
Solving Rightward Drift Problem (2)

- **Delete**

![Diagram of a queue with items](image)

- **Insert 9**

![Diagram of a queue with items](image)

When either front or back advances past MAX_QUEUE-1, it wraps around 0.

Queue with Single Item

- Queue contains **only one item**.
  - *back* and *front* are pointing at the same slot.

![Diagram of a queue with a single item](image)

- The last item is removed. Queue is empty.
  - *front* passed *back*.

![Diagram of an empty queue](image)

Insert the last item

- *back* catches up to *front* when the queue becomes full.

![Diagram of a queue with items](image)

Wrapping the values for *front* and *back*

- **Initializing**
  - Front = 0
  - Back = MAX_QUEUE-1
  - Count = 0

- **Adding**
  - back = (back+1) % MAX_QUEUE;
  - items[back] = newItem;
  - ++count;

- **Deleting**
  - deleteItem = items[front];
  - front = (front +1) % MAX_QUEUE;
  - --count;
enqueue with Array

```java
public void enqueue(Object newItem) throws QueueException{
    if (!isFull()){
        back = (back+1) % (MAX_QUEUE);
        items[back] = newItem;
        ++count;
    }else {
        throw QueueException(your_message);
    }
}
```

dequeue()

```java
public Object dequeue() throws QueueException{
    if (!isEmpty()){
        Object queueFront = items[front];
        front = (front+1) % (MAX_QUEUE);
        --count;
        return queueFront;
    }else{
        throw new QueueException (your_message);
    }
}
```

Implementation with List

- You can implement operation `dequeue()` as the list operation `remove(0)`.  
- `peek()` as `get(0)`  
- `enqueue()` as `add (size()-1, newItem)`

Outline

- Queue?  
- Implementing Queue  
  - Comparison implementations

java.util.Queue

- extends java.util.Collection  
- `add(e), remove(), element()`  
- `offer(e), poll(), peek()`  
- java.util.Deque:  
  - Subinterface of queue  
  - A linear collection that supports element insertion and removal at both ends.

Summary of Queue Operations (FIFO)

<table>
<thead>
<tr>
<th></th>
<th>Abstract</th>
<th>Interface</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>enqueue(e)</code></td>
<td><code>add(e)</code></td>
<td><code>addLast(e)</code></td>
<td></td>
</tr>
<tr>
<td><code>offer(e)</code></td>
<td><code>offer(e)</code></td>
<td><code>offerLast(e)</code></td>
<td></td>
</tr>
<tr>
<td><code>dequeue()</code></td>
<td><code>remove()</code></td>
<td><code>removeFirst()</code></td>
<td></td>
</tr>
<tr>
<td><code>poll()</code></td>
<td><code>poll()</code></td>
<td><code>pollFirst()</code></td>
<td></td>
</tr>
<tr>
<td><code>peek()</code></td>
<td><code>peek()</code></td>
<td><code>peekFirst()</code></td>
<td></td>
</tr>
<tr>
<td><code>element()</code></td>
<td><code>getFirst()</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Summary of LIFO operations

<table>
<thead>
<tr>
<th>ADT Stack</th>
<th>JCF Stack</th>
<th>JCF Deque</th>
</tr>
</thead>
<tbody>
<tr>
<td>push(e)</td>
<td>push(e)</td>
<td>addFirst(e)</td>
</tr>
<tr>
<td>pop()</td>
<td>pop()</td>
<td>removeFirst()</td>
</tr>
<tr>
<td>peek()</td>
<td>peek()</td>
<td>peekFirst()</td>
</tr>
</tbody>
</table>

Summary of Position-Oriented ADTs

- Stack, queue, and list (so far)
- createStack and createQueue
- isEmpty for stack and queue
- push and enqueue
- pop and dequeue
- Stack peek and queue peek

Next Reading

- Section 9. Advanced Java Topics