Part 4. Queues

CS 200 Algorithms and Data Structures

Outline

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

2

"Grill the Buffs" event 9/16 2011

4

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)

5

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

6

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**()

7

Outline

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

8

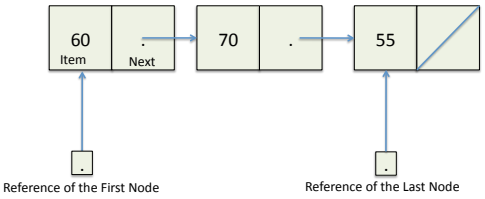
Implementations of Queue

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

9

Reference-based Implementation(1)

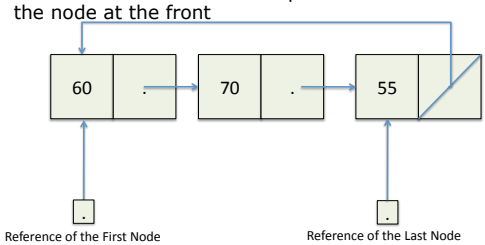
- Needs
  - **Nodes** with the *item* and a *reference* to the next item
  - **two external references** pointing to the *first node* and the *last node*.



10

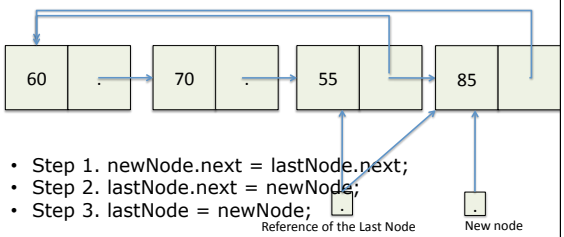
Reference-based Implementation(2)

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



11

Inserting an item into a nonempty queue



- Step 1. newNode.next = lastNode.next;
- Step 2. lastNode.next = newNode;
- Step 3. lastNode = newNode;

12

isEmpty()

```
public class QueuerReferenceBased implements
  QueueInterface {

    private Node lastNode;

    public QueueReferenceBased(){
        lastNode = null;
    }

    public boolean isEmpty(){
        return lastNode == null;
    }
}
```

13

Insert new item into the queue

```
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;
}
```

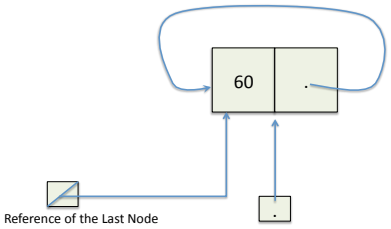
A. Empty queue

B. More than 1 item

14

Inserting a New Item

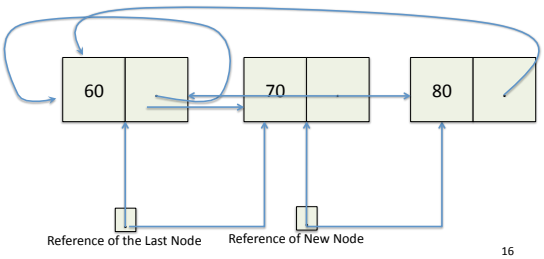
- Insert a ***new item*** into the **empty queue**



15

Inserting a New Item

- Insert a **new item** into a queue that has ***more than 1 items.***



16

Removing an item from queue

```
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..
    }
}
```

Why?

17

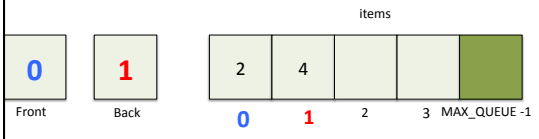
Peek?

```
public Object peek() throws QueueException {

   if (!isEmpty()){
      Node firstNode = lastNode.next;
      return firstNode.item;

   }else {
      throw new QueueException(your_message);
   }
}
```

18

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



19

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

20

Implementing queue with Array

- **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.

21

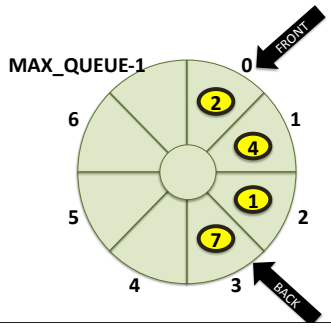
Solving Rightward Drift Problem (1)

- **Shifting** array elements to **the left**. Therefore front is always pointing to `items[0]`.
- **Cost** of implementation

22

Solving Rightward Drift Problem (2)

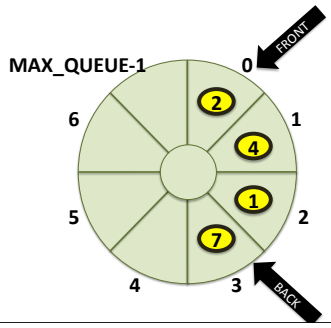
- **Circular implementation** of a queue



23

Solving Rightward Drift Problem (2)

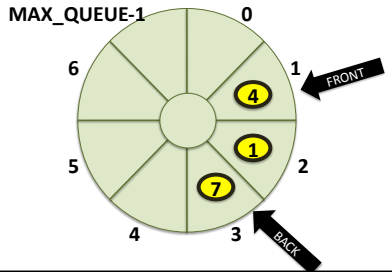
- **Delete**



24

Solving Rightward Drift Problem (2)

- **Delete**

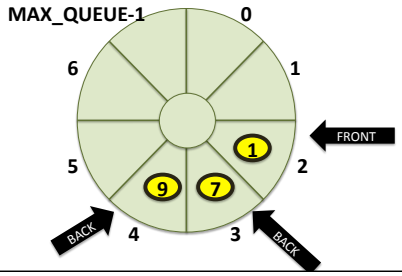


25

Solving Rightward Drift Problem (2)

- **Insert 9**

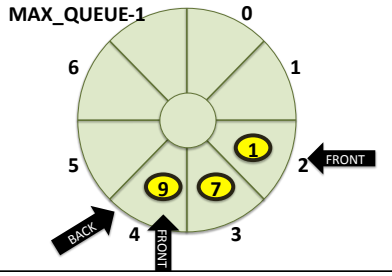
When either front or back advances past MAX_QUEUE-1, it wraps around 0



26

Queue with Single Item

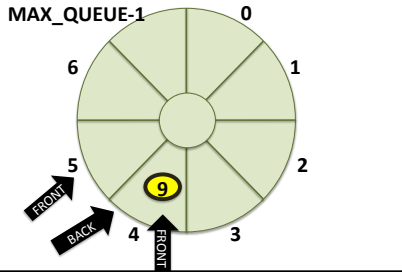
- Queue contains **only one item**.
- *back* and *front* are pointing at the same slot.



27

Queue with Single Item

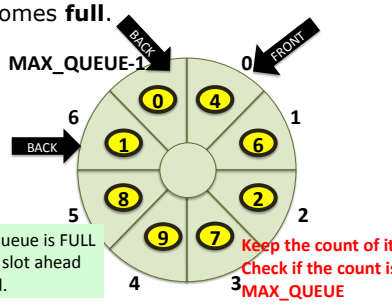
- The last item is removed. Queue is empty.
  - *front* passed *back*.



28

Insert the last item

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



When the queue is FULL *front* is one slot ahead *back* as well.

Keep the count of items and Check if the count is equal to MAX_QUEUE

29

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;
```

30

enqueue with Array

```
public void enqueue(Object newItem) throws
  QueueException{

   if (!isFull()){
      back = (back+1) % (MAX_QUEUE);
      items[back] = newItem;
      ++count;

   }else {
      throw QueueException(your_message);
   }
}
```

31

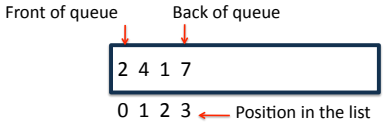
dequeue()

```
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);
   }
}
```

32

Implementation with List

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



33

Outline

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

34

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**.

35

Summary of Queue Operations (FIFO)

ADT Queue	JCF Queue	JCF Deque
`enqueue(e)`	`add(e)`	`addLast(e)`
	`offer(e)`	`offerLast(e)`
`dequeue()`	`remove()`	`removeFirst()`
	`poll()`	`pollFirst()`
`peek()`	`peek()`	`peekFirst()`
	`element()`	`getFirst()`

36

Summary of LIFO operations

ADT Stack	JCF Stack	JCF Deque
push(e)	push(e)	addFirst(e)
pop()	pop()	removeFirst()
peek()	peek()	peekFirst()

37

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`

38

Next Reading

- Section 9. Advanced Java Topics

39