Data Structures and Object-Oriented Design III

Spring 2014

Carola Wenk

Array-Based Stack vs. DynamicStack

```java
public class ArrayStack {
  final static int DEFAULT_CAPACITY=50;
  private int[] S;
  private int top;

  public ArrayStack(){
    this(DEFAULT_CAPACITY);
  }

  public ArrayStack(int capacity){
    S = new int[capacity];
    top=-1;
  }

  public void push(int x){
    //Needs to throw exception when stack is full
    S[++top]=x;
  }

  public int pop(){
    if(top>=0)
      return S[top--];
    else
      throw new RuntimeException("Stack is empty.");
  }
}
```

```java
public class Tester{
  public static void main(String[] args) {
    ArrayStack stack = new ArrayStack();
    stack.push(5);
    System.out.println("popped: "+stack.pop());
  }
}
```

```java
public class DynamicStack implements Stack{

  private class StackNode {
    private int data;
    private StackNode next;

    public StackNode(int d){
      data = d;
      next=null;
    }
  }

  private StackNode top = null;

  public void push(int x) {
    StackNode temp = new StackNode(x);
    temp.next = top;
    top = temp;
  }

  public int pop(){
    if (top == null)
      throw new RuntimeException("Stack empty!");

    int x = top.data;
    top = top.next;
    return x;
  }
}
```

If we change `ArrayStack` to `DynamicStack`, the code still works.

Java Interfaces

- We can specify that a Java class implements a particular kind of functionality defined as an `interface`.

```
public interface Stack {
  public int pop();
  public void push(int x);
}
```

```
public class ArrayStack implements Stack {

  ...

}
```

```
public class DynamicStack implements Stack {

  ...

}
```

```
public class Tester {
  public static void main(String[] args) {
    Stack stack = new ArrayStack();
    stack.push(5);
    System.out.println("popped: "+stack.pop());
  }
}
```

If we change `ArrayStack` to `DynamicStack`, the code still works.

Queues

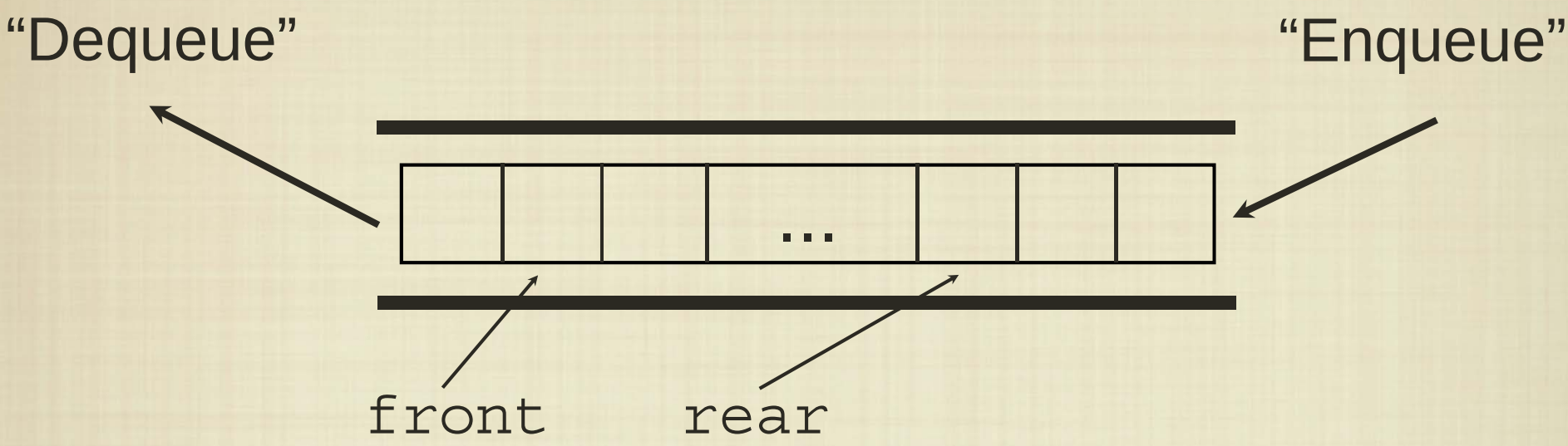
- A queue is a “first-in, first-out” data structure. How do we implement it?

“Dequeue”

“Enqueue”

```
class Queue {

public Queue() { ... }

public int dequeue() { ... }

public void enqueue(int x) { ... }

}
```

Static Implementation



- We need two indices to keep track of the front and rear of the queue. What do we know about the relationship between these two indices?

Queues

A queue can be implemented easily with an array. What limit does the size of the array impose on queue operations?

```
public class Queue {

  private int[] Q = null;
  private int front, rear;

  public Queue(int capacity) {
     Q = new int[capacity];
     front = 0; rear = -1;
  }

  public void enqueue(int x) {
     ...
  }

  public int dequeue() {
     ...
  }

  public int size() { ... }

}
```

Queues

A queue can be implemented easily with an array. What limit does the size of the array impose on queue operations?

```
public class Queue {

  private int[] Q = null;
  private int front, rear;

  public Queue(int capacity) {
     Q = new int[capacity];
     front = 0; rear = -1;
  }

  public void enqueue(int x) {
     Q[++rear] = x;
  }

  public int dequeue() {
     ...
  }

  public int size() { ... }

}
```

Queues

A queue can be implemented easily with an array. What limit does the size of the array impose on queue operations?

```
public class Queue {

  private int[] Q = null;
  private int front, rear;

  public Queue(int capacity) {
     Q = new int[capacity];
     front = 0; rear = -1;
  }

  public void enqueue(int x) {
     Q[++rear] = x;
  }

  public int dequeue() {
     return Q[front++];
  }

  public int size() { ... }

}
```

Queues

A queue can be implemented easily with an array. What limit does the size of the array impose on queue operations?

```
public class Queue {

  private int[] Q = null;
  private int front, rear;

  public Queue(int capacity) {
     Q = new int[capacity];
     front = 0; rear = -1;
  }

  public void enqueue(int x) {
     Q[++rear] = x;
  }

  public int dequeue() {
     return Q[front++];
  }

  public int size() { return rear-front+1; }

}
```

Queues

In what cases do the public methods cause runtime errors? In what cases do they simply return incorrect results?

For a queue of capacity n, how many enqueue operations will succeed before causing a runtime error? Is this reasonable?

```
public class Queue {

  private int[] Q = null;
  private int front, rear;

  public Queue(int capacity) {
     Q = new int[capacity];
     front = 0; rear = -1;
  }

  public void enqueue(int x) {
     Q[++rear] = x;
  }

  public int dequeue() {
     return Q[front++];
  }

  public int size() { return rear-front+1; }

}
```

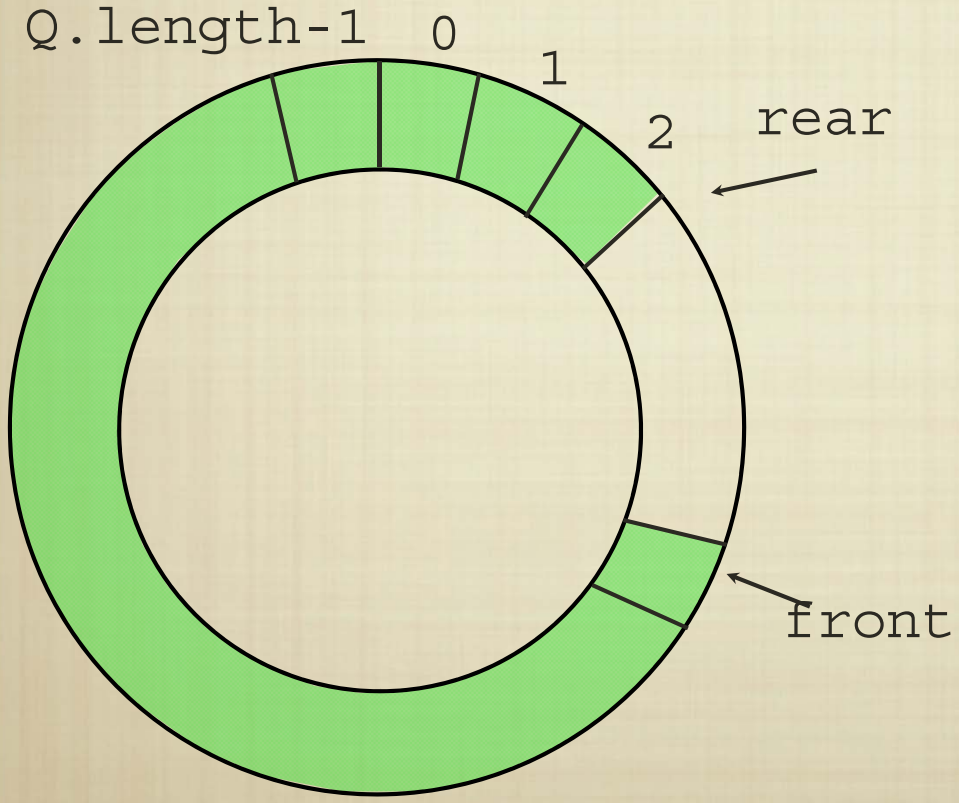
```
public class Queue {

  private int[] Q = null;
  private int front, rear;

  public Queue(int capacity) {
     Q = new int[capacity];
     front = 0; rear = -1;
  }

  public void enqueue(int x) {
     if (size() < Q.length)
        Q[++rear] = x;
     else throw new RuntimeException(“Queue full!”);
  }

  public int dequeue() {
     if (size() == 1) {
        int returnValue = Q[front];
        front = 0; rear = -1;
        return returnValue;
     } else if (size() > 0)
        return Q[front++];
     else // size() <= 0
       throw new RuntimeException(“Queue empty!”);
  }

  public int size() { return rear-front+1; }

}
```

What happens if we repeatedly enqueue/dequeue a single element?

Circular Queues

The capacity of the queue should allow a certain number of items that are in the queue at any one time - not the number of queue operations.

We can do this using modulo.

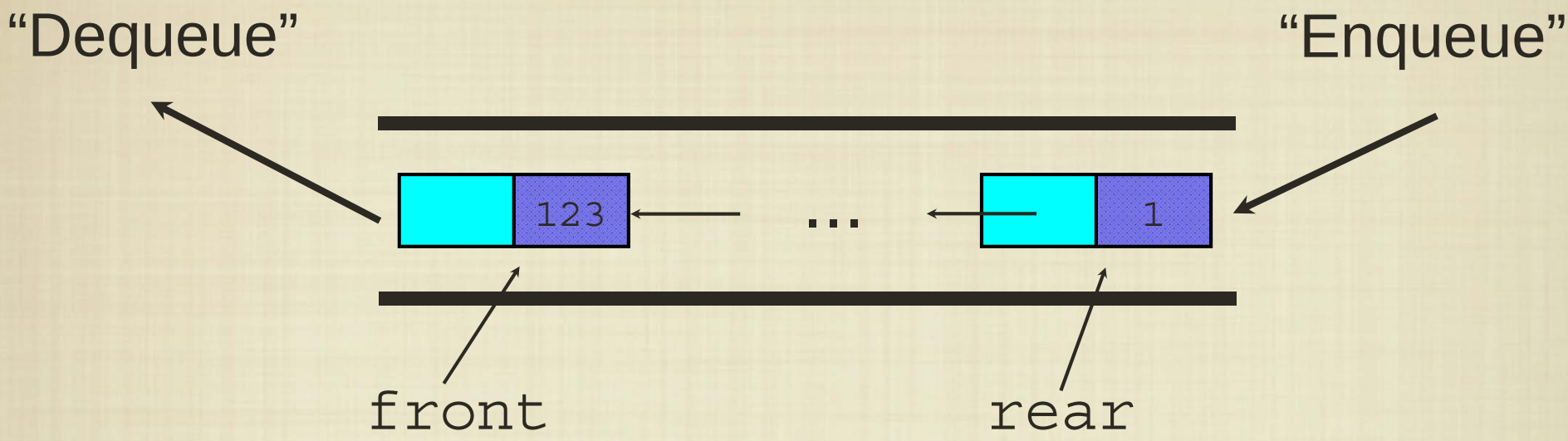



```
public class Queue {
  private int[] Q = null;
  private int front, rear, size;

  public Queue(int capacity) {
      Q = new int[capacity];
      front = 0; rear = -1; size = 0;
  }

  public void enqueue(int x) {
      if (size < Q.length) {
          rear = (rear + 1) % Q.length;
          Q[rear] = x;
          size ++;
      } else throw new RuntimeException("Queue full!");
  }

  public int dequeue() {
      if (size() > 0) {
          int returnValue = Q[front];
          front = (front + 1) % Q.length;
          size--;
          return returnValue;
      } else throw new RuntimeException("Queue empty!");
  }

  public int size() { return size; }

}
```

Instead of `Q[++rear]=x`

Instead of `return Q[front++]`

Dynamic Implementation



- We can use a linked structure to maintain the Queue elements, with a reference to the front and rear of the structure.

What correctness properties do we want from this structure?
Which special cases do we need to handle?

Dynamic Queue

```
public class DynamicQueue {

  public DynamicQueue(){

  }

  public void enqueue(int x) {

  }

  public int dequeue() {

  }

  public int size() {

  }
}
```

Dynamic Queue

```
public class DynamicQueue {

  private class QueueNode {
    public int data;
    public QueueNode next;
    public QueueNode(int d) {
      data = d;
      next = null;

    }
  }

  private QueueNode front, rear;
  private int size;

  public DynamicQueue(){
    front=rear=null;
    size=0;
  }

  public void enqueue(int x) {
    if (size == 0) {
      front = new QueueNode(x);
      rear = front;
    } else {
      rear.next = new QueueNode(x);
      rear = rear.next;
    }
    size ++;
  }
```

```
  public int dequeue() {
    if (front == null)
      throw new RuntimeException("Queue empty!");
    else{
      int returnValue = front.data;
      front = front.next;
      size--;
      return returnValue;
    }
  }

  public int size() {
    return size;
  }
}
```