Queues and Stacks

Queue: An Abstract Data Type

- an abstract data type for maintaining a sequence of objects
- three operations — enter, leave, and next
- enter adds an object to the queue
- leave removes the next object of the queue, returning it
- next returns the next object without altering the queue
- a FIFO queue is "first in, first out" — objects enter the queue at the back, and leave from the front

```
public interface Queue {
    public boolean isEmpty();
    public boolean isFull();
    public Object next();
    public boolean enter(Object o); // true if successful
    public Object leave();
    public int length();
}
```

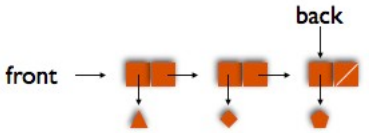
Implementing FIFO Queues with Mutable Lists*

front → NIL back → NIL

empty queue

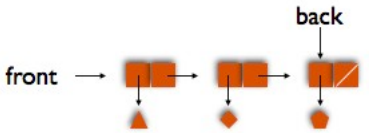
*Mutable Lists are like Lists except
data and next are not declared final and
methods are written for setHead and setTail

Implementing FIFO Queues with Mutable Lists



queue with three objects

Implementing FIFO Queues with Mutable Lists



queue with three objects

enter makes a new node and adds it to the end
by setting the tail of back and then back itself to the
new node

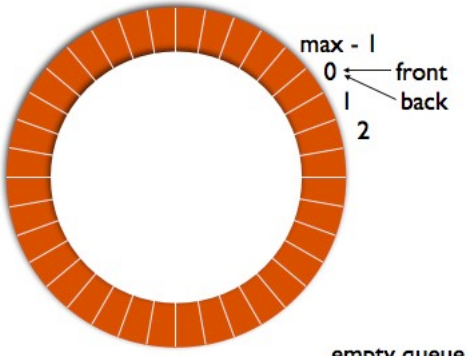
leave returns the head of front and changes front to
its tail

```
public class ListQueue implements Queue {
    private MutableList front;
    private MutableList back;

    public ListQueue () {
        this.front = MutableList.NIL;
        this.back = MutableList.NIL;
    }

    public boolean isEmpty () {
        return this.front.isEmpty();
    }

    public boolean isFull () {
        return false;
    }

    public int length () {
        return front.length();
    }
.
.
.
}
```

```
public class ListQueue implements Queue {
.
.
.
    public Object next () {
        return !this.isEmpty() ?
               this.front.head() :
               null;
    }

    public Object leave () {
        Object obj = this.next();
        if (!this.isEmpty())
            this.front = this.front.tail();
        return obj;
    }
.
.
.
}
```

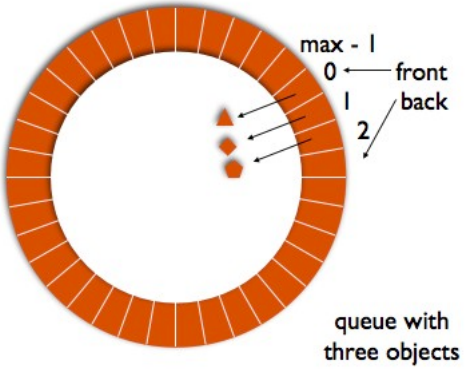
```
public class ListQueue implements Queue {
.
.
.
    public boolean enter (Object d) {
        try {
            MutableList p = MutableList.list(d);
            if (this.isEmpty()) {
                this.front = p;
                this.back = p;
            } else {
                this.back.setTail(p);
                this.back = p;
            }
            return true;
        } catch (OutOfMemoryError e) { return false; }
    }
.
.
.
}
```

Implementing FIFO Queues with Circular Arrays



empty queue

Implementing FIFO Queues with Circular Arrays



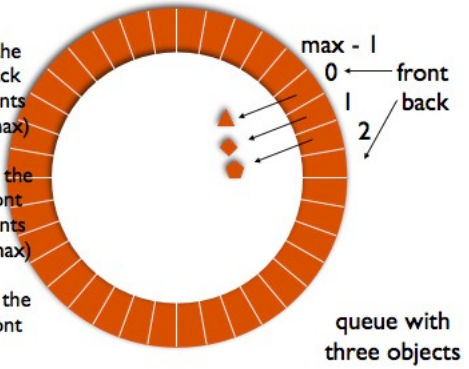
queue with three objects

Implementing FIFO Queues with Circular Arrays

enter puts the object at back and increments back (mod max)

leave returns the object at front and increments front (mod max)

next returns the object at front



queue with three objects

```
public class ArrayQueue implements Queue {
    private Object[] values;
    private int max;
    private int front;
    private int back;

    public ArrayQueue (int max) {
        this.values = new Object[max];
        this.max = max;
        this.front = 0;
        this.back = 0;
    }

    private int addl (int i) {
        return (i + 1) % this.max;
    }
.
.
.
}
```

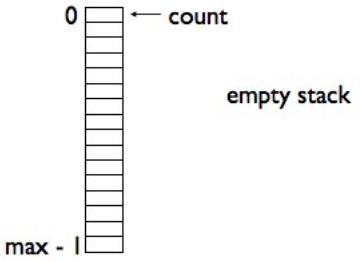
```
public class ArrayQueue implements Queue {
.
.
.
    public boolean isEmpty () {
        return this.front == this.back;
    }

    public boolean isFull () {
        return this.front == this.addl(this.back);
    }

    public int length () {
        if (this.isEmpty()) return 0;
        if (this.back < this.front)
            return this.back + this.max - this.front;
        else return this.back - this.front;
    }
.
.
.
}
```

```
public class ArrayQueue implements Queue {
.
.
.
    public Object next () {
        if (this.isEmpty()) return null;
        return this.values[this.front];
    }

    public boolean enter (Object d) {
        if (this.isFull()) return false;
        this.values[this.back] = d;
        this.back = this.addl(this.back);
        return true;
    }

    public Object leave () {
        Object result = this.next();
        if (!this.isEmpty())
            this.front = this.addl(this.front);
        return result;
    }
.
.
```

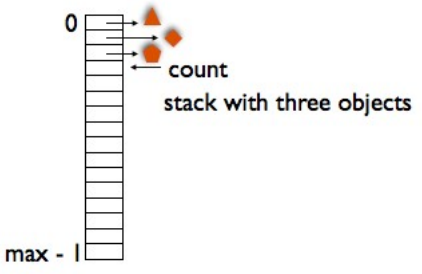
Stack: An Abstract Data Type

- a last-in, first-out (LIFO) abstract data type for maintaining a sequence of objects
- three operations — often referred to as push, pop, and top, but we will implement stacks as LIFO queues
- enter (push) adds an object to the top of the stack
- leave (pop) removes the top object of the stack, returning it
- next (top) returns the top object without altering the stack

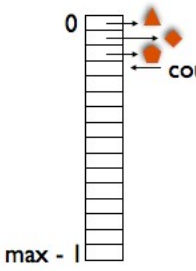
Implementing Stacks with Arrays



Implementing Stacks with Arrays



Implementing Stacks with Arrays



enter (push) adds object at position count and increments count

leave (pop) decrements count and then returns object at that position

next (top) returns object at position count - 1

Implementing Stacks with Lists

values → NIL

empty stack

Implementing Stacks with Lists

stack with three objects

Implementing Stacks with Lists

stack with three objects

enter (push) adds a node at the front of the list and changes values to point to it

leave (pop) returns the head of values and changes values to point to its tail

next (top) returns the head of values

```
public class ListStack implements Queue {
    private List values;

    public ListStack () {
        this.values = List.NIL;
    }

    public boolean isEmpty () {
        return this.values.isEmpty();
    }

    public boolean isFull () {
        return false;
    }

    public int length () {
        return this.values.length();
    }
.
.
.
}
```

```
public class ListStack implements Queue {
.
.
    public boolean enter (Object d) {
        try { this.values = this.values.push(d); }
        catch (OutOfMemoryError e) { return false; }
        return true;
    }

    public Object next () {
        return this.values.isEmpty() ?
            null :
            this.values.head();
    }

    public Object leave () {
        Object result = this.next();
        if (!this.values.isEmpty())
            this.values = this.values.tail();
        return result;
    }
.
.
}
```

```
public class ListStack implements Queue {
.
.
.
    public String toString () {
        String s = this.values.toString();
        return "Stack:" + s.substring(1,s.length() - 1) + ":";
    }
.
.
.
}
```

A Simple Postfix Calculator

- to calculate (5 + 3) * 7 - 2 in postfix, 3 5 + 7 * 2 -
- using a stack, if a value is entered, push it; if an operator is entered, pop two values, apply the operator, push the result

Calculating 3 5 + 7 * 2 -

the stack

Calculating 3 5 + 7 * 2 -

push 3

the stack

Calculating 3 5 + 7 * 2 -

the stack

Calculating 3 5 + 7 * 2 -

push 5

the stack

Calculating 3 5 + 7 * 2 -

the stack

Calculating 3 5 + 7 * 2 -

+

the stack

Calculating 3 5 + 7 * 2 -

pop twice

the stack

Calculating 3 5 + 7 * 2 -

5+3=8

the stack

Calculating 3 5 + 7 * 2 -

push 8

the stack

Calculating 3 5 + 7 * 2 -

the stack

Calculating 3 5 + 7 * 2 -

push 7

the stack

Calculating 3 5 + 7 * 2 -

the stack

Calculating 3 5 + 7 * 2 -

*

the stack

Calculating 3 5 + 7 * 2 -

pop twice

the stack

Calculating 3 5 + 7 * 2 -

8*7=56

the stack

Calculating 3 5 + 7 * 2 -

push 56

the stack

Calculating 3 5 + 7 * 2 -

the stack

Calculating 3 5 + 7 * 2 -

push 2

the stack

Calculating 3 5 + 7 * 2 -

the stack

Calculating 3 5 + 7 * 2 -

-

the stack

Calculating 3 5 + 7 * 2 -

pop twice

the stack

Calculating 3 5 + 7 * 2 -

56-2=54

the stack

Calculating 3 5 + 7 * 2 -

push 54

the stack

Calculating 3 5 + 7 * 2 -

the stack

```
public class Calc {
  public static void main (String[] args) {
      String input; Queue s = new ListStack();
      int val, arg1 = 0, arg2 = 0; char op;
      while ((op = (input = IO.prompt("? ")).charAt(0)) != 'q')
        try {
          if (op == '+' || op == '-' || op == '*' || op == '/') {
          arg2 = (Integer) s.leave(); arg1 = (Integer) s.leave();
          }
          switch (op) {
          case '+': val = arg1 + arg2; IO.stdout.println(val);
                    s.enter(val); break;
          case '-': val = arg1 - arg2; IO.stdout.println(val);
                    s.enter(val); break;
          case '*': val = arg1 * arg2; IO.stdout.println(val);
                    s.enter(val); break;
          case '/': val = arg1 / arg2; IO.stdout.println(val);
                    s.enter(val); break;
          default:  s.enter(Integer.parseInt(input));
          }
        }
        catch (Exception e) { IO.stdout.println("Huh?"); }
  }
}
```