Chapter 8

Stacks and Queues

A *stack* is a finite sequence of elements in which the only element that can be removed is the element that was most recently inserted.

That is, the element most recently inserted is the next element to be removed.

Last-In, First-Out (LIFO)

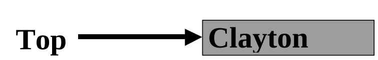
Top **– The most recently inserted element**

Push **– To insert onto the top of a stack**

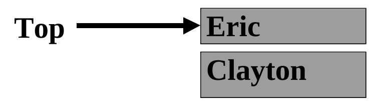
Pop **– To remove the top element in a stack**

Start with an empty stack.

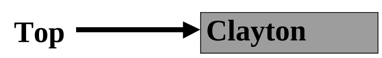
Push “Clayton.”



Push “Eric.”



Pop.



The PureStack Interface

```
public interface PureStack<E>
{
        /**
         *  Determines the number of elements in this
         *  PureStack object.
         *
         *  @return the number of elements in this
         *                 PureStack object.
         *
         */
        int size();
```

```java
/**
 *  Determines if this PureStack object has no elements.
 *
 *  @return true – if this PureStack object has no
 *                    elements; otherwise, return false.
 *
 */
boolean isEmpty();


/**
 *  Inserts a specified element on the top of this
 *  PureStack object.
 *  The averageTime(n) is constant and worstTime(n) is
 *  O(n).
 *
 *  @param element – the element to be pushed.
 *
 */
void push (E element);
```

```java
/**
 *  Removes the top element from this PureStack object.
 *
 *  @return – the element removed.
 *  @throws NoSuchElementException – if this PureStack
 *               object is empty.
 */
 E pop();

/**
 *  Returns the top element on this PureStack object.
 *
 *  @return – the element returned.
 *  @throws NoSuchElementException – if this PureStack
 *               object is empty.
 */
 E peek();

} // interface PureStack
```

There is an implementation in java.util.

```java
public class Stack extends Vector {
…
```

Vector **is virtually identical to** ArrayList**.**

The push**,** pop **and** peek **methods are easily defined. For example:**

```java
public E push(E item) {
    addElement(item);

    return item;
}
```

But NO Vector **methods are overridden. So it is possible to invoke methods that violate the definition of a stack!**

For example,

```java
myStack.remove (7);
```

Alternative implementations:

1. **Inherit from** ArrayList **or** LinkedList**? Ugh! Too many overrides.**
2. **Use an array?**

```
protected E[ ] data;
protected int top;
```

Then top **would be at the back of the array. Why?**

3. **Use an** ArrayList **or** LinkedList**? Yes, and all method definitions are one-liners.**

```
public class LinkedListPureStack<E>
{
    protected LinkedList<E> list;

    public LinkedListPureStack (
                LinkedListPureStack<E> otherStack)
    {
        list = new LinkedList<E> (otherStack.list);
    } // copy constructor

    public void push (E element)
    {
        list.add (element);
    } // method push
    …
} // class LinkedListPureStack
```

Determine the output from the following:

```
LinkedListPureStack<Integer> myStack =
                new LinkedListPureStack<Integer>( );

for (int i = 0; i < 10; i++)
   myStack.push (i * i);

while (!myStack.isEmpty( ))
   System.out.println (myStack.pop( ));
```

Stack Application 1

How Compilers Implement Recursion

Whenever a method is called, information is saved to prevent overlaying of that info in case the method is recursive. This information is collectively referred to as an *activation record* or *stack frame*.

Each activation record contains:

1. **A variable that contains the return address in the calling method;**
2. **For each parameter in the called method, a variable that contains a copy of the corresponding argument;**
3. **For each variable declared in the called method's block, a variable that contains a copy of that declared variable.**

There is a run-time stack to handle these activation records.

Push: When method is called

Pop: When execution of method is completed

An activation record is similar to an execution frame, except that an activation record has variables only, no code.

You can replace recursion with iteration by creating your own stack.

Recall from Chapter 5:

Decimal to Binary:

```
public static String getBinary (int n)
{
    if (n < 0)
        throw new IllegalArgumentException();
    if (n <= 1)
        return Integer.toString (n);
    return getBinary(n / 2) + Integer.toString(n % 2);// RA2
} // method getBinary
```

The following method maintains its own stack:

```
public static String getBinary (int n)
{
    ArrayStack<Integer> myStack =
                            new ArrayStack<Integer>();

    String binary = new String();

    if (n < 0)
            throw new IllegalArgumentException( );
    myStack.push (n % 2);
    while (n > 1)
    {
            n /= 2;
            myStack.push (n % 2);
    } // pushing
    while (!myStack.isEmpty())
            binary += myStack.pop();
    return binary + "\n\n";
} // method getBinary
```

Notice that we save n % 2 **on the stack, but there is no need to save the return address because this version of** getBinary **is not recursive.**

Exercise: Trace the execution of the above method after an initial call of

`getBinary (20);`

show the contents of `myStack`**.**

Stack Application 2

Converting from Infix to Postfix

In *infix* notation, an operator is placed between its operands.

a + b

c – d + (e * f – g * h) / i

Old compilers:

Infix → Machine language

This gets messy because of parentheses.

Newer compilers:

Infix → Postfix → Machine language

In *postfix* notation, an operator is placed immediately after its operands.

Infix	Postfix
a + b	ab+
a + b * c	abc*+
a * b + c	ab*c+
(a + b) * c	ab+c*

Parentheses are not needed and not used, in postfix.

Let’s convert an infix string to a postfix string.

x – y * z

Postfix preserves the order of operands, so an operand can be appended to postfix as soon as that operand is encountered in infix.

Infix	Postfix
x – y * z	x

Infix	Postfix
x – y * z	x

The operands for ‘-’ are not yet in postfix, so ‘-’ must be temporarily saved somewhere.

Infix	Postfix
x – y * z	xy

Infix	Postfix
x – y * z	xy

The operands for ‘*’ are not yet in postfix, so ‘*’ must be temporarily saved somewhere, ***and restored before*** **‘-’.**

Infix	Postfix
x – y * z	xyz

Infix	Postfix
x – y * z	xyz* –

Suppose, instead, we started with x*y-z. After moving ‘x’ to postfix, ‘*’ is temporarily saved, and then ‘y’ is appended to postfix. What happens when ‘-’ is accessed?

Infix	Postfix
x * y – z	xy

The ‘*’ must be moved to postfix now, because both of the operands for ‘*’ are on postfix. Then the ‘-’ must be saved temporarily. After ‘z’ is moved to postfix, ‘-’ is moved to postfix, and we are done.

Infix	Postfix
x * y – z	xy*z–

The temporary storage facility is a stack.

Here is the strategy for maintaining the stack:

For each operator in infix:
- **Loop until operator pushed:**
  - **If operator stack is empty,**
    - **Push**
  - **Else if *infix* operator has *greater* precedence than top operator on stack,**
    - ***Push***
  - **Else**
    - **Pop and append to postfix**

Infix Greater, Push

Convert from infix to postfix:

Infix	Postfix
a + b * c / d - e	

Infix	Postfix
a + b * c / d – e	abc*d/+e –

-
/
*
+

Operator stack

What about parentheses?

Left parenthesis: Push, but with lowest precedence.

Right parenthesis: keep popping and appending to postfix until '(' popped; pitch '(' and proceed.

Convert to postfix:

x * (y + z)

Infix	Posstfix
x * (y + z)	xyz+*

+
(
*

Operator stack

Infix **Postfix**

x * (y + z – (a / b + c) * d) / e

Operator stack

To decide what action to take in converting from infix to postfix, all we need to know is the current character in infix and the top character on operator stack.

The following transition matrix specifies the transition from infix notation to postfix notation:

Top Character on Stack

Infix Char	(	+,-	*,/	empty
identifier	Append to Postfix	Append to Postfix	Append to Postfix	Append to Postfix
)	Pop; Pitch '('	Pop to Postfix	Pop to Postfix	Error
(	Push	Push	Push	Push
+,-	Push	Pop to Postfix	Pop to Postfix	Push
*,/	Push	Push	Pop to Postfix	Push
empty	Error	Pop to Postfix	Pop to Postfix	Done

Tokens

A *token* is the smallest meaningful unit in a program.

Each token has two parts:

A generic part, for the category of the token;

A specific part, to access the characters in the token.

For example:

ADD_OP	+, -

IDENTIFIER	35

// index 35 in symbol table

Infix-to-postfix applet

http://www.cs.lafayette.edu/~collinsw/infix/infix.html

In *prefix* notation, an operator immediately precedes its operands.

Infix	Prefix
a + b	+ab
a * (b + c)	*a+bc
a * b + c	+*abc

In prefix notation, as in postfix, there are no parentheses.

Two stacks are used:

Operator stack: Same rules as for postfix stack

Operand stack: to hold the operands

Whenever opt is popped from operator stack, opd1 and then opd2 are popped from operand stack. The string opt + opd2 + opd1 is pushed onto operand stack.

Note: opd2 was pushed before opd1.

Convert from infix to prefix:

Infix
a + (b * c – d) / e

Infix	Prefix
a + (b * c – d) / e	+a/– *bcde

Operand stack	Operator stack
+a/– *bcde	
/– *bcde	
e	
–*bcd	
d	/
*bc	–
c	*
b	(
a	+

Exercise: Convert to Prefix

a – b + c * (d / e – (f + g))

A *queue* is a finite sequence of elements in which:

- Insertion occurs only at the back;
- Deletion occurs only at the front.

Enqueue – To inset an element at the back

Dequeue – To delete the front element

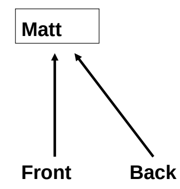
Front – To return a reference to the front element

In a queue, the first element inserted will be the first element deleted: FIFO (First-In, First-Out)

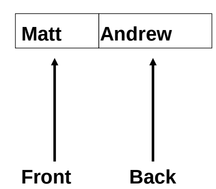
Compare to a stack: LIFO

(Last-In-First-Out)

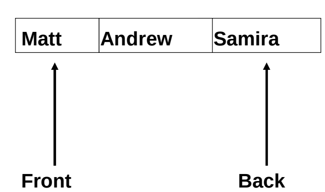
Enqueue “Matt”



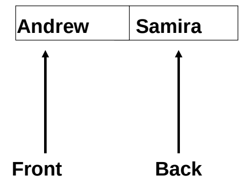
Enqueue “Andrew”



Enqueue “Samira”



Dequeue



The PureQueue interface

```
public interface PureQueue<E>
{
    // Returns the number of elements in this PureQueue
    // object.
    int size( );


    // Returns true if this PureQueue object has no
    // elements.
    boolean isEmpty( );
```

```
/**
 *  Inserts a specified element at the back of this
 *  PureQueue object.  The averageTime(n) is
 *  constant and worstTime(n) is O(n).
 *
 *  @param element – the element to be appended.
 */
void enqueue (E element);


/**
 *  Removes the front element from this PureQueue
 *  object.
 *
 *  @return – the element removed.
 *  @throws NoSuchElementException – if this
 *              PureQueue object is empty.
 */
E dequeue();
```

```
    /**
     *  Returns the front element in this PureQueue
     *  object.
     *
     *  @return – the element returned.
     *
     *  @throws NoSuchElementException – if
     *              PureQueue object is empty.
     *
     */
     E front();

} // interface PureQueue
```

For the dequeue method, what is

worstTime (n)?

For the sake of code re-use, the implementation will work with an existing class.

ArrayList?

LinkedList?

Inheritance:
The implementation of PureQueue **is-a** LinkedList

or

Aggregation:
The implementation of PureQueue **has-a** LinkedList

Inheritance Tax: 32 Overrides

```
public E get (int index) {

    throw new UnsupportedOperationException( );

}
```

So we'll use aggregation:

```
public class LinkedListPureQueue<E>
                    implements PureQueue<E>
{

    protected LinkedList<E> list;
```

```
public LinkedListPureQueue()
{
    list = new LinkedList<E>();
} // default constructor

public void enqueue (E element)
{
    list.addLast (element); // same as list.add (element);
} // method enqueue

public E dequeue()
{
    return list.removeFirst();
} // method dequeue
```

Determine the output from the following:

```
LinkedListPureQueue<Integer> myQueue =
            new LinkedListPureQueue<Integer>();

for (int i = 0; i < 10; i++)
   myQueue.enqueue (i * i);

while (!myQueue.isEmpty( ))
   System.out.println (myQueue.dequeue( ));
```

Computer Simulation

A *system* is a collection of interacting parts.

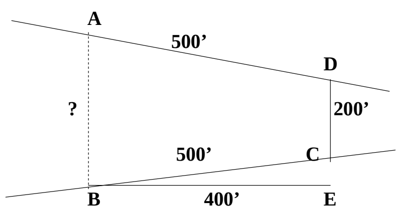
A *model* is a simplification of a system.

The purpose of building a model is to study the underlying system.

***Physical model*: Differs from the system only in scale or intensity.**

Examples: War games, pre-season

***Mathematical model*: A set of equations, variables, and assumptions**

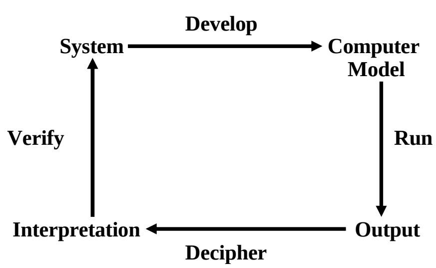


Assumptions: Angle ADC = angle BCD
BEC forms a right triangle
DCE forms a straight line
Line segment AB parallel to DC

Distance from A to B?

If it is infeasible to solve the math model by hand, a program is developed.

***Computer simulation*: The development of computer programs to solve math models**



If the interpretation does not correspond to the behavior of the system, change the model!

***Feedback:* A process in which the factors that produce a result are themselves affected by that result**

Here, the model is affected by its output.

Queue Application

A Simulated Car Wash

Analysis:

One wash station

10 minutes for each car to get washed

At any time, at most 5 cars waiting to be washed; any others turned away and not counted

Average waiting time = sum of waiting times / number of cars

In a given minute, a departure is processed before an arrival.

If a car arrives when no car is being washed (then no car is waiting), the car immediately enters the wash station.

A car stops waiting when it enters the wash station.

Sentinel is 999.

System test 1:
8
11
11
13
14
16
16
20
999

Time	Event	Waiting Time
8	Arrival	
11	Arrival	
11	Arrival	
13	Arrival	
14	Arrival	
16	Arrival	
16	Arrival (Overflow)	
18	Departure	0
20	Arrival	
28	Departure	7
38	Departure	17
48	Departure	25
58	Departure	34
68	Departure	42
78	Departure	48

Average waiting time

= 173.0 minutes / 7 cars

= 24.7 minutes per car

Car Wash Applet

http://www.cs.lafayette.edu/~collinsw/carwash/car.html

Exercise:

Given the following arrival times, determine the average waiting time:

4, 8, 12, 16, 23, 999 (the sentinel)

Design of CarWash **class**

```
/**
 *  Initializes this CarWash object.
 *
 */
public CarWash()
```

```
/**
 *  The next arrival at the specified time has been
 *   processed.
 *
 *  @param nextArrivalTime – the time when the
 *               next arrival will occur.
 *
 *  @throws IllegalArgumentException – if
 *               nextArrivalTime is less than the
 *               current time.
 *
 */
public void process (int nextArrivalTime)
```

```
/**
 *  Washes all cars that are still unwashed after last arrival.
 *
 */
public void finishUp()


/*
 *  Returns the history of this CarWash object's arrivals and
 *  departures, and the  average waiting time.
 *
 *  @return the history of the simulation, including the
 *               average waiting time.
 *
 */
public LinkedList<String> getResults()
```

Fields?

First, we'll decide what variables will be needed, and then choose the fields from them.

```
PureQueue<Car> carQueue;
```

Each element in carQueue **will be of type** Car**. What information about a car do we need?**

In the Car **class:**

```
// @return the arrival time of the car just dequeued.
public int getArrivalTime( )
```

We have a Car **class for the sake of later modifications to the problem. For example, the cost of a wash might depend on the number of axles.**

To calculate the average waiting time:

```
int numberOfCars,
    sumOfWaitingTimes;
```

To get the sum of the waiting times:

```
int currentTime,
    waitingTime;

waitingTime = currentTime – car.getArrivalTime();
```

Calculated just before a car enters the wash

The simulation will be *event-based*: Is the next event an arrival or a departure?

```
int nextArrivalTime,
    nextDepartureTime;  // = 10000 if no car being washed
                        // (so next event will be an arrival)
```

Finally,

```
LinkedList<String> results;  // to hold the chart of arrivals,
                   //  departures, and averageWaitingTime
```

A rule of thumb is that a field should be needed in most of the class's public methods.

Fields:

```
PureQueue<Car> carQueue;

LinkedList<String> results;

int currentTime,
     waitingTime,
     sumOfWaitingTimes,
     numberOfCars,
    nextDepartureTime;  // = 10000 if no car being washed
```

```
public CarWash()
{
      carQueue<Car> =
                 new LinkedListPureQueue<Car>();
      results = new LinkedList<String>();
      results.add ("Time    Event   Waiting Time");
      currentTime = 0;
      waitingTime = 0;
      numberOfCars = 0;
      sumOfWaitingTimes = 0;
      nextDepartureTime = 10000;
} // constructor
```

```
public void process (int nextArrivalTime)
{
        if (nextArrivalTime < currentTime)
                throw new IllegalArgumentException();
        while (nextArrivalTime >= nextDepartureTime)
                processDeparture();
        processArrival (nextArrivalTime);
} // process
```

```
protected void processArrival (int nextArrivalTime)
{
    currentTime = nextArrivalTime;
    results.add (Integer.toString (currentTime) + "\tArrival");
    if (carQueue.size() == 5)
        results.add (" (Overflow)\n");
    else
    {
        numberOfCars++;
        if (nextDepartureTime == 10000)
            nextDepartureTime = currentTime + 10;
        else
            carQueue.enqueue (new Car (nextArrivalTime));
        results.add ("\n");
    } // not an overflow
} // method processArrival
```

```
protected void processDeparture()
{
    currentTime = nextDepartureTime;
    results.add (Integer.toString (currentTime) + "\tDeparture\t\t" +
                        Integer.toString (waitingTime) + "\n");
    if (!carQueue.isEmpty())
    {
        Car car = carQueue.dequeue();
        waitingTime = currentTime – car.getArrivalTime();
        sumOfWaitingTimes += waitingTime;
        nextDepartureTime = currentTime + 10;
    } // carQueue was not empty
    else
    {
        waitingTime = 0;
        nextDepartureTime = 10000;
    } // carQueue was empty
} // method processDeparture
```

If the next arrival times are read in, the results are not generalizable. Instead, we will read in

```
int meanArrivalTime;   // the average time between arrivals
```

Then, using

```
double randomDouble = random.nextDouble( );
```

We calculate

```
int timeTillNext = (int)Math.round (-meanArrivalTime *
                            Math.log (1 - randomDouble));
```

Exercise: Assume that meanArrivalTime **is 3. If** Math.log (1 – randomDouble) = **–0.6 and** currentTime = **25, When will the next arrival occur?**

```
timeTillNext = (int)Math.round (-meanArrivalTime *
                     Math.log (1 - randomDouble));
```