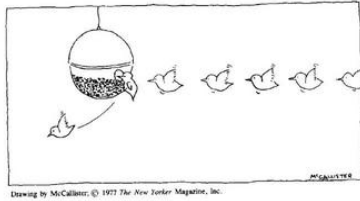


## 4.2: Stacks and Queues



Drawing by McCallister, © 1977 The New Yorker Magazine, Inc.

### Stack and Queue ADTs

#### Fundamental data type.

- Set of operations (add, remove, test if empty) on generic data.
- Intent is clear when we insert.
- Which object do we remove?

#### Stack ("last in first out" or LIFO).

- Remove the object most recently added.
- Analogy: cafeteria trays, surfing Web.

#### Queue ("first in first out" or FIFO).

- Remove the object least recently added.
- Analogy: Registrar's line.

#### MultiSet.

- Remove any object.
- Law professor calls on arbitrary student.

### Queue

#### Queue operations.

- `add` Insert a new object onto queue.
- `remove` Delete and return the object least recently added.
- `isEmpty` Is the queue empty?

```
public static void main(String[] args) {
    Queue q = new Queue();
    q.add("This");
    q.add("is");
    q.add("a");
    System.out.println(q.remove());
    q.add("test.");
    while(!q.isEmpty())
        System.out.println(q.remove());
}
```

A simple queue client



(C) 1994-2000, J. O'Reilly & Associates, Inc.

### More Applications of Queues

#### Real world applications.

- iPod playlist.
- Store last 10 sound waves to implement echo filter.
- Dispensing requests on a shared resource (printer, processor).
- Asynchronous data transfer (file IO, pipes, sockets).
- Data buffers (MP3 player, portable CD player, TiVo).
- Graph processing. **stay tuned**

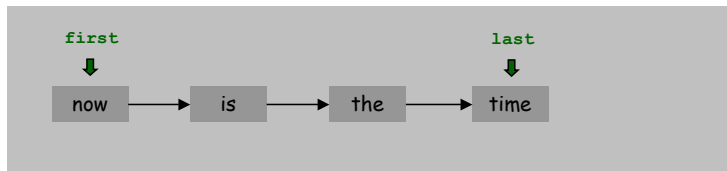
#### Simulations of the real world.

- Traffic analysis of Lincoln tunnel.
- Waiting times of customers at DMV.
- Determining number of cashiers to staff at checkout line.

## Queue: Linked List Implementation

### Linked list implementation.

- Maintain linked list of elements.
- Let `first` be reference to first node on list.
- Let `last` be reference to last node on list.



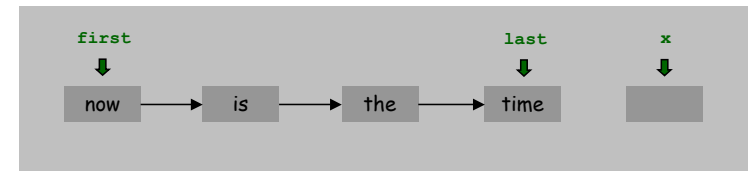
5

## Queue: Linked List Implementation

### Linked list implementation.

- Maintain linked list of elements.
- Let `first` be reference to first node on list.
- Let `last` be reference to last node on list.

### Insert.



```
x.item = "for";  
last.next = x;  
last = x;
```

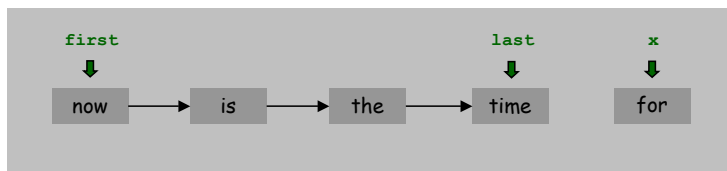
6

## Queue: Linked List Implementation

### Linked list implementation.

- Maintain linked list of elements.
- Let `first` be reference to first node on list.
- Let `last` be reference to last node on list.

### Insert.



```
List x = new List();  
last.next = x;  
last = x;
```

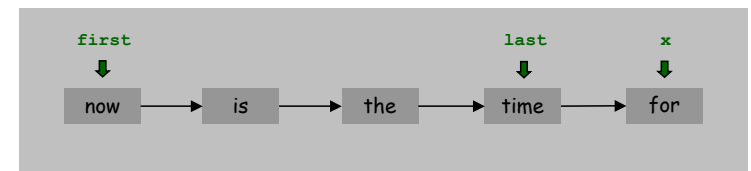
7

## Queue: Linked List Implementation

### Linked list implementation.

- Maintain linked list of elements.
- Let `first` be reference to first node on list.
- Let `last` be reference to last node on list.

### Insert.



```
List x = new List();  
x.item = "for";  
last = x;
```

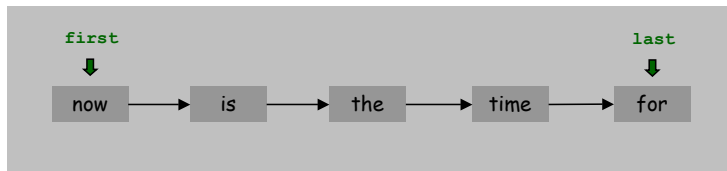
8

## Queue: Linked List Implementation

### Linked list implementation.

- Maintain linked list of elements.
- Let `first` be reference to first node on list.
- Let `last` be reference to last node on list.

### Insert.



```
List x = new List();  
x.item = "for";  
last.next = x;
```

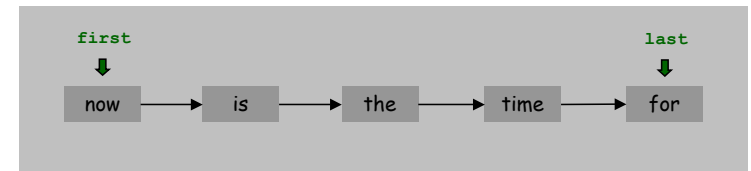
9

## Queue: Linked List Implementation

### Linked list implementation.

- Maintain linked list of elements.
- Let `first` be reference to first node on list.
- Let `last` be reference to last node on list.

### Delete.



```
val now  
first = first.next;  
return val;
```

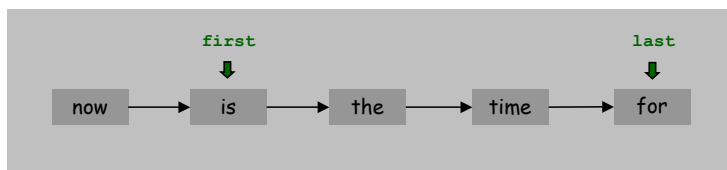
10

## Queue: Linked List Implementation

### Linked list implementation.

- Maintain linked list of elements.
- Let `first` be reference to first node on list.
- Let `last` be reference to last node on list.

### Delete.



```
val now  
Object val = first.item;  
return val;
```

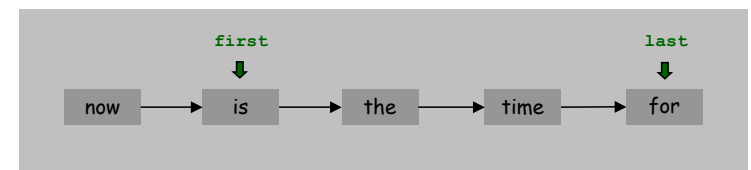
11

## Queue: Linked List Implementation

### Linked list implementation.

- Maintain linked list of elements.
- Let `first` be reference to first node on list.
- Let `last` be reference to last node on list.

### Delete.



```
val now  
Object val = first.item;  
first = first.next;
```

12

## Queue: Linked List Implementation

```

public class Queue {
    private List first;  ← reference to first element of queue
    private List last;  ← reference to last element of queue

    private class List { Object item; List next; }

    public boolean isEmpty() { return (first == null); }

    public void add(Object anItem) {
        List x = new List();  ← add to end of linked list
        x.item = anItem;
        x.next = null;
        if (isEmpty()) { first = x; last = x; }
        else { last.next = x; last = x; }
    }

    public Object remove() {
        Object val = first.item;
        first = first.next;  ← delete from front of linked list
        return val;
    }
}

```

13

## Object

### Class Object.

- All classes "inherit" from the special class `Object`.
- All objects have certain pre-defined methods.

Method	Description	Default	Example
<code>toString</code>	convert to string	memory address	"hello " + s
<code>equals</code>	are two objects equal?	are two memory addresses equal?	<code>if (s.equals(t))</code>
<code>hashCode</code>	convert to integer	memory address	<code>s.hashCode()</code>

### Consequences.

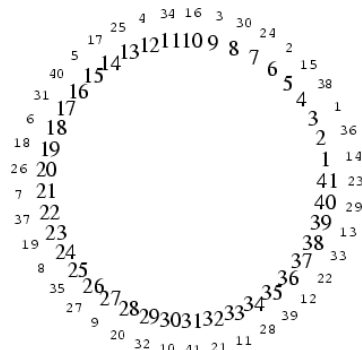
- Can have a queue of any type of object, e.g, `String` or `Wave`.
- Might need to cast the return value of `remove` to desired type.

14

## Josephus Problem

### Flavius Josephus. (first century)

- Band of 41 Jewish rebels trapped in cave by Romans.
- Preferring suicide to capture, rebels formed a circle and killed every 3rd remaining person until no one was left.
- Where should you stand to be the last survivors? 31



15

## Josephus Problem

```

public class Josephus {
    public static void main(String[] args) {
        int M = Integer.parseInt(args[0]);
        int N = Integer.parseInt(args[1]);
        IntQueue q = new IntQueue();
        for (int i = 1; i <= N; i++) q.add(i);

        while (!q.isEmpty()) {
            for (int i = 0; i < M - 1; i++)
                q.add(q.remove());
            System.out.print(q.remove() + " ");
        }
        System.out.println();
    }
}

```

N = 8, M = 3

1 2 3 4 5 6 7 8

2 3 4 5 6 7 8 1

3 4 5 6 7 8 1 2

4 5 6 7 8 1 2

5 6 7 8 1 2 4

6 7 8 1 2 4 5

7 8 1 2 4 5

...

```

% java Josephus 5 9
5 1 7 4 3 6 9 2 8

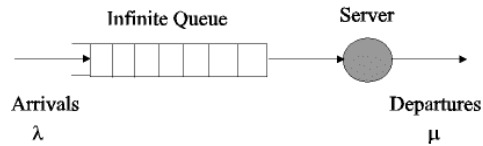
```

16

## M/M/1 Queuing Model

### M/M/1 queue.

- Customers arrive at rate of  $\lambda$  per minute.
- Customers are serviced at rate of  $\mu$  per minute.
- Use *Poisson process* to model arrivals and departures.
- How long does a customer wait in queue?



17

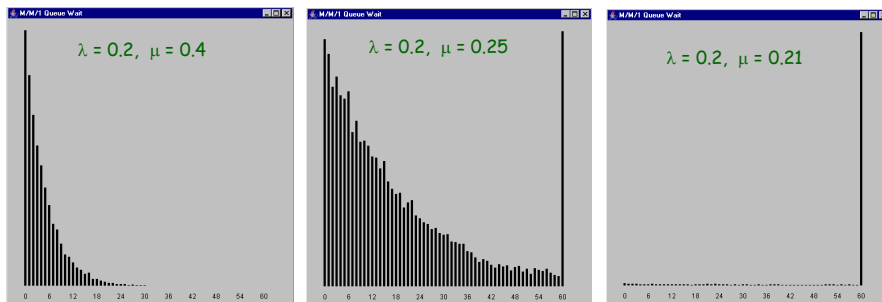
## M/M/1 Queue: Implementation

```
while(true) {  
    if (nextArrival < nextDeparture) { arrival  
        Customer c = new Customer();  
        c.arrive(nextArrival); amount of time until next arrival  
        q.add(c); ↙  
        nextArrival += exponential(lambda);  
    }  
  
    else { departure  
        if (!q.isEmpty()) {  
            Customer c = (Customer) q.remove();  
            c.depart(nextDeparture);  
            hist.addDataPoint(Math.min(60, c.length()));  
        } ↙ amount of time until next departure  
        nextDeparture += exponential(mu);  
    }  
}
```

18

## M/M/1 Queue Analysis

As service rate approaches arrival rate, service goes to  $h^{***}$ .



19

## Stack

### Stack operations.

- push            Insert a new object onto stack.
- pop             Delete and return the object most recently added.
- isEmpty        Is the stack empty?

### Applications.

- Surfing the web in a browser.
- Implementing function calls in a compiler.
- Parsing.
- PostScript language for printers.
- Reverse Polish calculators.

20

## Sample Stack Client: Balanced Parentheses

Determine if the parentheses in a String are balanced.

- Process each character, one at a time.
  - left delimiter: push
  - right delimiter: pop and check that it matches
  - other character: ignore
- Balanced if stack is empty upon termination.

String	Balanced
( ) ( ) )	true
( ( ( ) ) ) )	true
( ( ) ) ( ( )	false
[ ( [ ] ) ]	true
[ [ ( ) ] ]	false
a[2*(i+j)] = a[b[i]];	true

21

## Sample Stack Client: Balanced Parentheses

```
public class Parentheses {
    private final static char LPAREN = '(';
    private final static char RPAREN = ')';
    private final static char LBRACE = '{';
    private final static char RBRACE = '}';

    public static boolean isBalanced(String s) {
        CharStack stack = new CharStack(); // a stack of char
        for (int i = 0; i < s.length(); i++) {
            if (s.charAt(i) == LPAREN) stack.push(LPAREN);
            if (s.charAt(i) == LBRACE) stack.push(LBRACE);
            if (s.charAt(i) == RPAREN) {
                if (stack.isEmpty()) return false;
                if (stack.pop() != LPAREN) return false;
            }
            if (s.charAt(i) == RBRACE) {
                if (stack.isEmpty()) return false;
                if (stack.pop() != LBRACE) return false;
            }
        }
        return stack.isEmpty(); // balanced if stack is empty
    }
}
```

push left delimiters

check that delimiter matches

22

## Stack Implementation (Array)

Implementing a Stack with an array.

- Use array *s* to store *N* to objects on stack.
- push: add new object at *s*[*N*].
- pop: remove object from *s*[*N*-1].

How to resize array?

- Use *repeated doubling strategy*.
- If *s* not big enough, create new array of double the size and copy elements from *s* to new array



23

## Stack Implementation (Array)

```
public class Stack {
    private Object[] s = new Object[1]; // stack elements
    private int N = 0; // stack size

    public boolean isEmpty() { return N == 0; }

    public void push(Object anItem) {
        if (N >= s.length) {
            Object[] dup = new Object[2*N];
            for (int i = 0; i < N; i++) dup[i] = s[i];
            s = dup; // double size of array
        }
        s[N++] = anItem;
    }

    public Object pop() { return s[--N]; }
}
```

24

## Stack Implementation (Linked List)

```
public class Stack {
    private List first = null; // ← first node of linked list

    private class List {
        Object item;
        List next;
        List(Object item, List next) {
            this.item = item; this.next = next;
        }
    }

    public boolean isEmpty() { return (first == null); }

    public void push(Object item) {
        first = new List(item, first);
    }

    public Object pop() {
        Object val = first.item;
        first = first.next;
        return val;
    }
}
```

25

## Stack Implementations: Array vs. Linked List

We can implement a stack with either array or linked list and client can use interchangeably.

Which is better?

- Array invokes expensive re-doubling operation every once in a while.
- Linked list uses extra space and time to deal with references.

26

## Adapter Class

Class is designed to work with Objects.

- Build *adapter class* to deal with primitive types.
- Hides annoying conversions between `int` and `Integer` from client.
- Problem fixed in Java 1.5 with *generics* and *autoboxing*.

Wrapper type.

- Each primitive type has *wrapper* object type.
- Ex: `Integer` is wrapper type for `int`.

```
public class IntStack {
    private Stack s = new StackArray();

    private boolean isEmpty() { return s.isEmpty(); }
    private push(int item)    { s.push(new Integer(item)); }
    int pop()                 { return ((Integer) s.pop()).intValue(); }
}
```

27

## Postfix Evaluation

Put operator after operands in expression.

- Use stack to evaluate.
  - operand: push it onto stack
  - operator: pop operands, push result
- Systematic way to save intermediate results and avoid parentheses.



J. Lukasiewicz  
(1878-1956)

```
% java Postfix
1 2 3 4 5 * + 6 * * +
6625      infix expression: (1+((2*((3+(4*5))*6)))

% java Postfix
7 16 16 16 * * * 5 16 16 * * 3 16 * 1 + + +
30001     convert 7531 from hex to decimal

% java Postfix
7 16 * 5 + 16 * 3 + 16 * 1 +
30001     again, using Horner's method
```



28

## Postfix Evaluation

Maintain stack of integers.

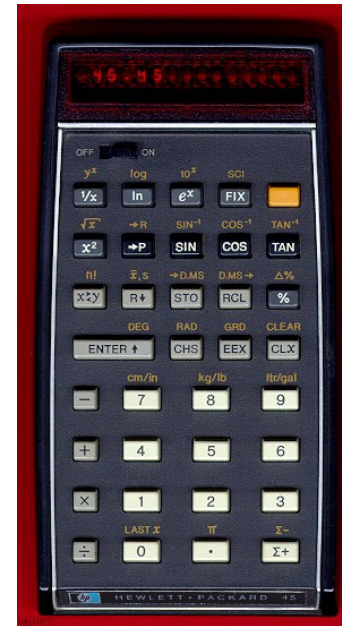
- Assume tokens separated by whitespace.
- Assume input is legal postfix expression.

```
public class Postfix {
    public static void main(String[] args) {
        IntStack stack = new IntStack();
        while (!In.isEmpty()) {
            String s = In.readString();
            if (s.equals("+"))
                stack.push(stack.pop() + stack.pop());
            else if (s.equals("*"))
                stack.push(stack.pop() * stack.pop());
            else
                stack.push(Integer.parseInt(s));
        }

        System.out.println(stack.pop());
    }
}
```

29

## An Ancient Calculator



HP 45.

Price in 1973: \$395.

- (That's \$1600 in 2002 dollars.)

What's missing from this keyboard?

30

## Infix to Postfix

Infix to postfix algorithm.

- Left parentheses: ignore.
- Right parentheses: pop and print.
- Operator: push.
- Integer: print.

```
% java Infix
( 2 + ( ( 3 + 4 ) * ( 5 * 6 ) ) )
* 2 3 4 + 5 6 * * +

% java Infix | java Postfix
( 2 + ( ( 3 + 4 ) * ( 5 * 6 ) ) )
212
```

```
public class Infix {
    public static void main(String[] args) {
        Stack stack = new StackArray();
        while (!In.isEmpty()) {
            String s = In.readString();
            if (s.equals("(")) stack.push(s);
            else if (s.equals("*")) stack.push(s);
            else if (s.equals("+")) System.out.print(stack.pop() + " ");
            else if (s.equals("(")) System.out.print("(");
            else
                System.out.print(s + " ");
        }
    }
}
```

32

## Summary

Stacks and queues are fundamental ADTs.

- Linked list implementation.
- Array implementation.
- Different performance characteristics.

Many applications.

- Calculators.
- Printers and PostScript language.
- Arithmetic expressions.
- Function implementation on compiler.
- Web browsing.
- ...

33