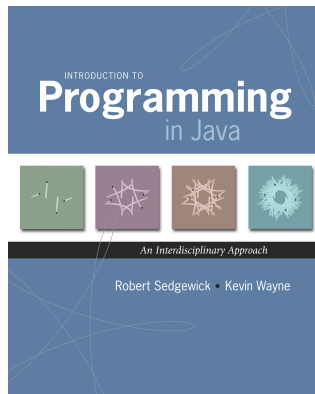


4.3 Stacks and Queues



Introduction to Programming in Java: An Interdisciplinary Approach · Robert Sedgewick and Kevin Wayne · Copyright © 2008 · October 30, 2008 11:08 AM

Fundamental data types.

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

Stack.

- Remove the item most recently added.
- Ex: cafeteria trays, Web surfing.

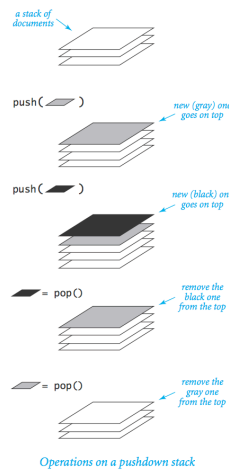
LIFO = "last in first out"

Queue.

- Remove the item least recently added.
- Ex: Registrar's line.

FIFO = "first in first out"

Stacks



3

Stack API

```
public class *StackOfStrings
    *StackOfStrings() create an empty stack
    boolean isEmpty() is the stack empty?
    void push(String item) push a string onto the stack
    String pop() pop the stack
```



```
public class Reverse {
    public static void main(String[] args) {
        StackOfStrings stack = new StackOfStrings();
        while (!StdIn.isEmpty())
            stack.push(StdIn.readString());
        while (!stack.isEmpty())
            StdOut.println(stack.pop());
    }
}
```

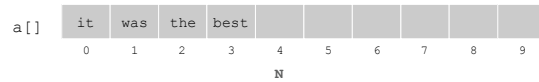
2

4

Stack: Array Implementation

Array implementation of a stack.

- Use array `a[]` to store `N` items on stack.
- `push()` add new item at `a[N]`.
- `pop()` remove item from `a[N-1]`.



```
public class ArrayStackOfStrings {
    private String[] a;
    private int N = 0;
    public ArrayStackOfStrings(int max) { a = new String[max]; }
    public boolean isEmpty() { return (N == 0); }
    public void push(String item) { a[N++] = item; }
    public String pop() { return a[--N]; }
}
```

max capacity of stack
↙

5

Array Stack: Trace

	StdIn	StdOut	N	a[]					
				0	1	2	3	4	
			0						
push	to		1	to					
	be		2	to	be				
	or		3	to	be	or			
	not		4	to	be	or	not		
	to		5	to	be	or	not	to	
pop	-	to	4	to	be	or	not	to	
	be		5	to	be	or	not	be	
	-	be	4	to	be	or	not	be	
	-	not	3	to	be	or	not	be	
	that		4	to	be	or	that	be	
	-	that	3	to	be	or	that	be	
	-	or	2	to	be	or	that	be	
	-	be	1	to	be	or	that	be	
	is		2	to	is	or	not	to	

6

Array Stack: Performance

Running time. Push and pop take constant time.

Memory. Proportional to `max`.

Challenge. Stack implementation where size is not fixed ahead of time.

Linked Lists

7

8

Sequential vs. Linked Allocation

Sequential allocation. Put object one after another.

- TOY: consecutive memory cells.
- Java: array of objects.

Linked allocation. Include in each object a **link** to the next one.

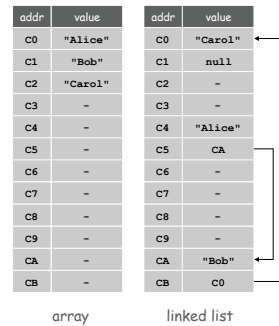
- TOY: link is memory address of next object.
- Java: link is reference to next object.

Key distinctions.

- Array: random access, fixed size.
- Linked list: sequential access, variable size.

get *i*th element

get next element



9

Linked Lists

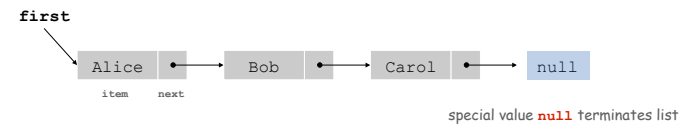
Linked list.

- A recursive data structure.
- A item plus a pointer to another linked list (or empty list).
- Unwind recursion: linked list is a sequence of items.

Node data type.

- A reference to a String.
- A reference to another Node.

```
public class Node {
    private String item;
    private Node next;
}
```



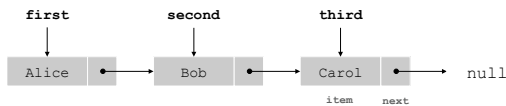
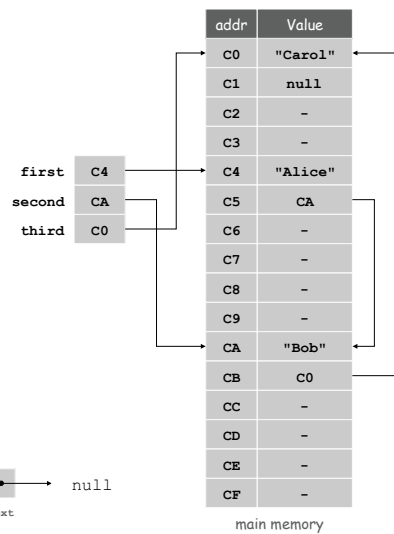
10

Building a Linked List

```
Node third = new Node();
third.item = "Carol";
third.next = null;

Node second = new Node();
second.item = "Bob";
second.next = third;

Node first = new Node();
first.item = "Alice";
first.next = second;
```

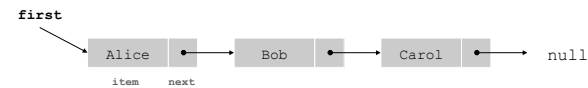


11

Traversing a Linked List

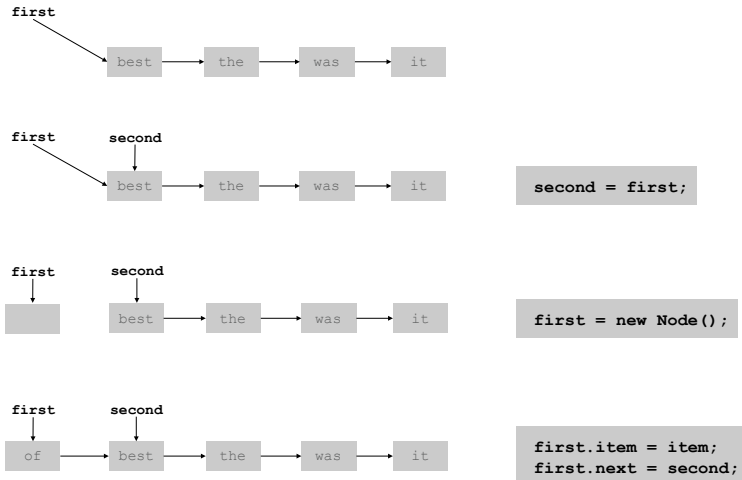
Iteration. Idiom for traversing a null-terminated linked list.

```
for (Node x = first; x != null; x = x.next) {
    StdOut.println(x.item);
}
```



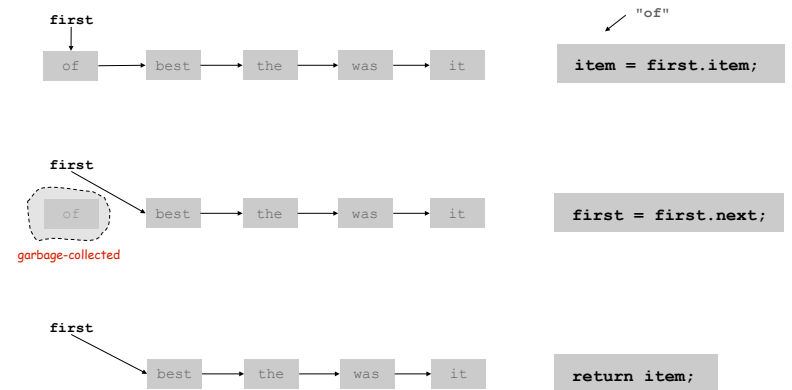
12

Stack Push: Linked List Implementation



13

Stack Pop: Linked List Implementation



14

Stack: Linked List Implementation

```

public class LinkedStackOfStrings {
    private Node first = null;

    private class Node {
        private String item;
        private Node next;
    } // "inner class"

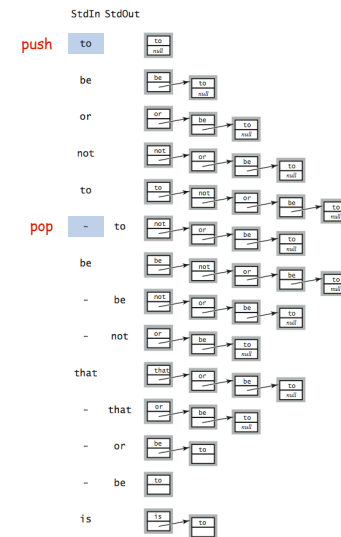
    public boolean isEmpty() { return first == null; }

    public void push(String item) {
        Node second = first;
        first = new Node();
        first.item = item;
        first.next = second;
    }

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

15

Linked List Stack: Trace



16

Stack Implementations: Tradeoffs

Array.

- Every push/pop operation take constant time.
- **But...** must fix maximum capacity of stack ahead of time.

Linked list.

- Every push/pop operation takes constant time.
- **But...** uses extra space and time to deal with references.

17

Parameterized Data Types

Parameterized Data Types

We implemented: StackOfStrings.

We also want: StackOfURLs, StackOfInts, ...

Strawman. Implement a separate stack class for each type.

- Rewriting code is tedious and **error-prone**.
- Maintaining cut-and-pasted code is tedious and **error-prone**.

19

Generics

Generics. Parameterize stack by a single type.

```
Stack<Apple> stack = new Stack<Apple>();  
Apple a = new Apple();  
Orange b = new Orange();  
stack.push(a);  
stack.push(b); // compile-time error  
a = stack.pop();
```

parameterized type

sample client

18

20

Generic Stack: Linked List Implementation

```
public class Stack<Item> {
    private Node first = null;

    private class Node {
        private Item item;
        private Node next;
    }

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

    public void push(Item item) {
        Node second = first;
        first = new Node();
        first.item = item;
        first.next = second;
    }

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

arbitrary parameterized type name

21

Stack Applications

Real world applications.

- Parsing in a compiler.
- Java virtual machine.
- Undo in a word processor.
- Back button in a Web browser.
- PostScript language for printers.
- Implementing function calls in a compiler.

23

Autoboxing

Generic stack implementation. Only permits reference types.

Wrapper type.

- Each primitive type has a **wrapper** reference type.
- Ex: Integer is wrapper type for int.

Autoboxing. Automatic cast from primitive type to wrapper type.

Autounboxing. Automatic cast from wrapper type to primitive type.

```
Stack<Integer> stack = new Stack<Integer>();
stack.push(17); // autobox (int -> Integer)
int a = stack.pop(); // autounbox (Integer -> int)
```

22

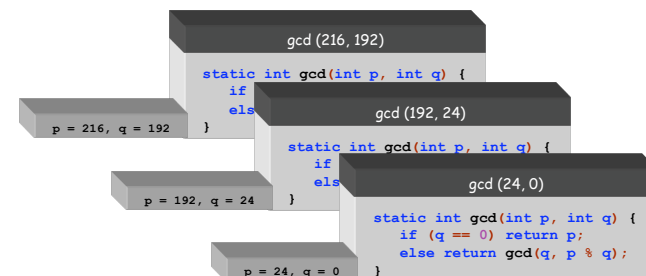
Function Calls

How a compiler implements functions.

- Function call: **push** local environment and return address.
- Return: **pop** return address and local environment.

Recursive function. Function that calls itself.

Note. Can always use an explicit stack to remove recursion.



24

Arithmetic Expression Evaluation

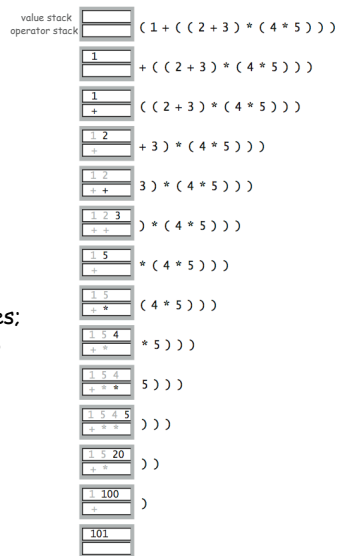
Goal. Evaluate infix expressions.

$$(1 + ((2 + 3) * (4 * 5)))$$

↙ operand
↘ operator

Two stack algorithm. [E. W. Dijkstra]

- Value: push onto the value stack.
- Operator: push onto the operator stack.
- Left parens: ignore.
- Right parens: pop operator and two values; push the result of applying that operator to those values onto the operand stack.



Context. An interpreter!

Arithmetic Expression Evaluation

```
public class Evaluate {
    public static void main(String[] args) {
        Stack<String> ops = new Stack<String>();
        Stack<Double> vals = new Stack<Double>();
        while (!StdIn.isEmpty()) {
            String s = StdIn.readString();
            if (s.equals("(")) ;
            else if (s.equals("+")) ops.push(s);
            else if (s.equals("*")) ops.push(s);
            else if (s.equals(")") {
                String op = ops.pop();
                if (op.equals("+")) vals.push(vals.pop() + vals.pop());
                else if (op.equals("*")) vals.push(vals.pop() * vals.pop());
            }
            else vals.push(Double.parseDouble(s));
        }
        StdOut.println(vals.pop());
    }
}
```

```
% java Evaluate
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
101.0
```

25

26

Correctness

Why correct? When algorithm encounters an operator surrounded by two values within parentheses, it leaves the result on the value stack.

$$(1 + ((2 + 3) * (4 * 5)))$$

So it's as if the original input were:

$$(1 + (5 * (4 * 5)))$$

Repeating the argument:

$$(1 + (5 * 20))$$

$$(1 + 100)$$

101

Extensions. More ops, precedence order, associativity, whitespace.

$$1 + (2 - 3 - 4) * 5 * \text{sqrt}(6*6 + 7*7)$$

27

Stack-Based Programming Languages

Observation 1. Remarkably, the 2-stack algorithm computes the same value if the operator occurs **after** the two values.

$$(1 ((2 3 +) (4 5 *) *) +)$$

Observation 2. All of the parentheses are redundant!

$$1 2 3 + 4 5 * * +$$



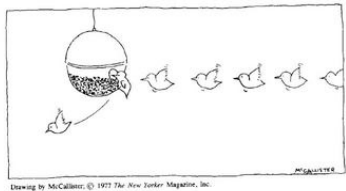
Jan Lukasiewicz

Bottom line. Postfix or "reverse Polish" notation.

Applications. Postscript, Forth, calculators, Java virtual machine, ...

28

Queues



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



(©1994-1995, 1999, 2003)

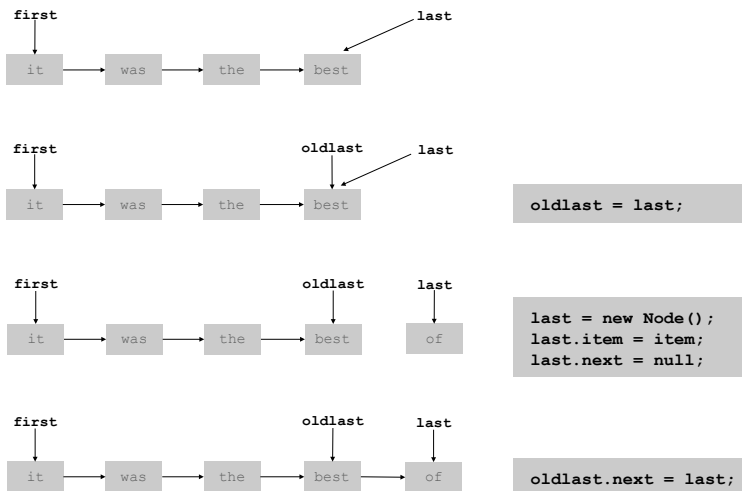
Queue API

```
public class Queue<Item>
{
    Queue<Item>()           create an empty queue
    boolean isEmpty()       is the queue empty?
    void enqueue(Item item) enqueue an item
    Item dequeue()          dequeue an item
    int length()            queue length
}
```

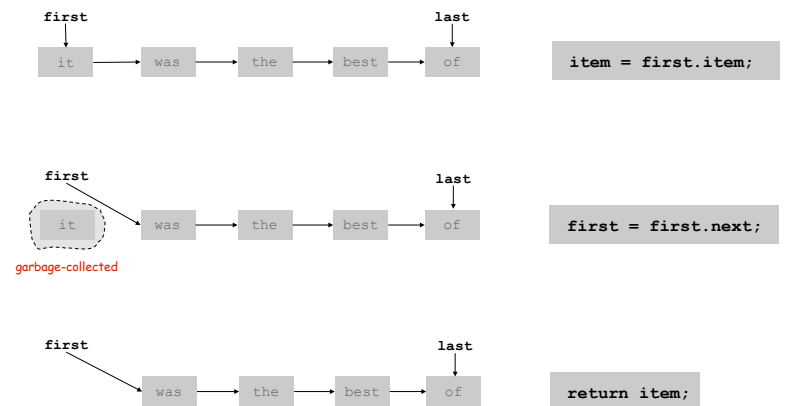


```
public static void main(String[] args) {
    Queue<String> q = new Queue<String>();
    q.enqueue("Vertigo");
    q.enqueue("Just Lose It");
    q.enqueue("Pieces of Me");
    q.enqueue("Pieces of Me");
    while (!q.isEmpty())
        StdOut.println(q.dequeue());
}
```

Enqueue: Linked List Implementation



Dequeue: Linked List Implementation



Queue: Linked List Implementation

```
public class Queue<Item> {
    private Node first, last;
    private class Node { Item item; Node next; }

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

    public void enqueue(Item item) {
        Node oldlast = last;
        last = new Node();
        last.item = item;
        last.next = null;
        if (isEmpty()) first = last;
        else oldlast.next = last;
    }

    public Item dequeue() {
        Item item = first.item;
        first = first.next;
        if (isEmpty()) last = null;
        return item;
    }
}
```

34

Summary

Stacks and queues are fundamental ADTs.

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

Many applications.

40

Queue Applications

Some applications.

- iTunes playlist.
- Data buffers (iPod, TiVo).
- Asynchronous data transfer (file IO, pipes, sockets).
- Dispensing requests on a shared resource (printer, processor).

Simulations of the real world.

- Guitar string.
- Traffic analysis.
- Waiting times of customers at call center.
- Determining number of cashiers to have at a supermarket.

35