

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE



<http://algs4.cs.princeton.edu>

## 1.3 BAGS, QUEUES, AND STACKS

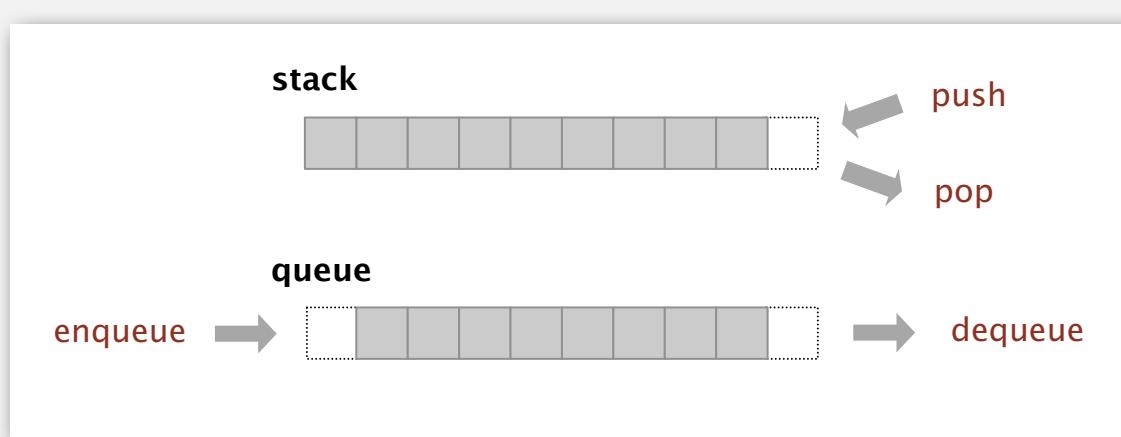
---

- ▶ *stacks*
- ▶ *resizing arrays*
- ▶ *queues*
- ▶ *generics*
- ▶ *iterators*
- ▶ *applications*

# Stacks and queues

## Fundamental data types.

- Value: collection of objects.
- Operations: **insert**, **remove**, **iterate**, test if empty.
- Intent is clear when we insert.
- Which item do we remove?



Stack. Examine the item most recently added. ← LIFO = "last in first out"

Queue. Examine the item least recently added. ← FIFO = "first in first out"

# Client, implementation, interface

---

Separate interface and implementation.

Ex: stack, queue, bag, priority queue, symbol table, union-find, ....

Benefits.

- Client can't know details of implementation ⇒ client has many implementation from which to choose.
- Implementation can't know details of client needs ⇒ many clients can re-use the same implementation.
- Design: creates modular, reusable libraries.
- Performance: use optimized implementation where it matters.

**Client:** program using operations defined in interface.

**Implementation:** actual code implementing operations.

**Interface:** description of data type, basic operations.

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 1.3 BAGS, QUEUES, AND STACKS

---

- ▶ *stacks*
- ▶ *resizing arrays*
- ▶ *queues*
- ▶ *generics*
- ▶ *iterators*
- ▶ *applications*

# Stack API

---

Warmup API. Stack of strings data type.

```
public class StackOfStrings
```

StackOfStrings()	<i>create an empty stack</i>
------------------	------------------------------

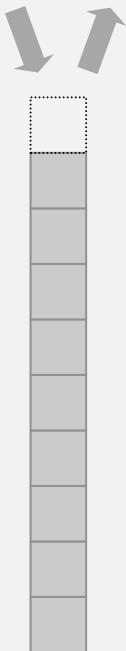
void push(String item)	<i>insert a new string onto stack</i>
------------------------	---------------------------------------

String pop()	<i>remove and return the string most recently added</i>
--------------	---

boolean isEmpty()	<i>is the stack empty?</i>
-------------------	----------------------------

int size()	<i>number of strings on the stack</i>
------------	---------------------------------------

push    pop



Warmup client. Reverse sequence of strings from standard input.

## Stack test client

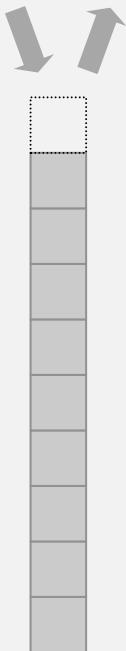
---

Read strings from standard input.

- If string equals "-", pop string from stack and print.
- Otherwise, push string onto stack.

push    pop

```
public static void main(String[] args)
{
    StackOfStrings stack = new StackOfStrings();
    while (!StdIn.isEmpty())
    {
        String s = StdIn.readString();
        if (s.equals("-")) StdOut.print(stack.pop());
        else             stack.push(s);
    }
}
```

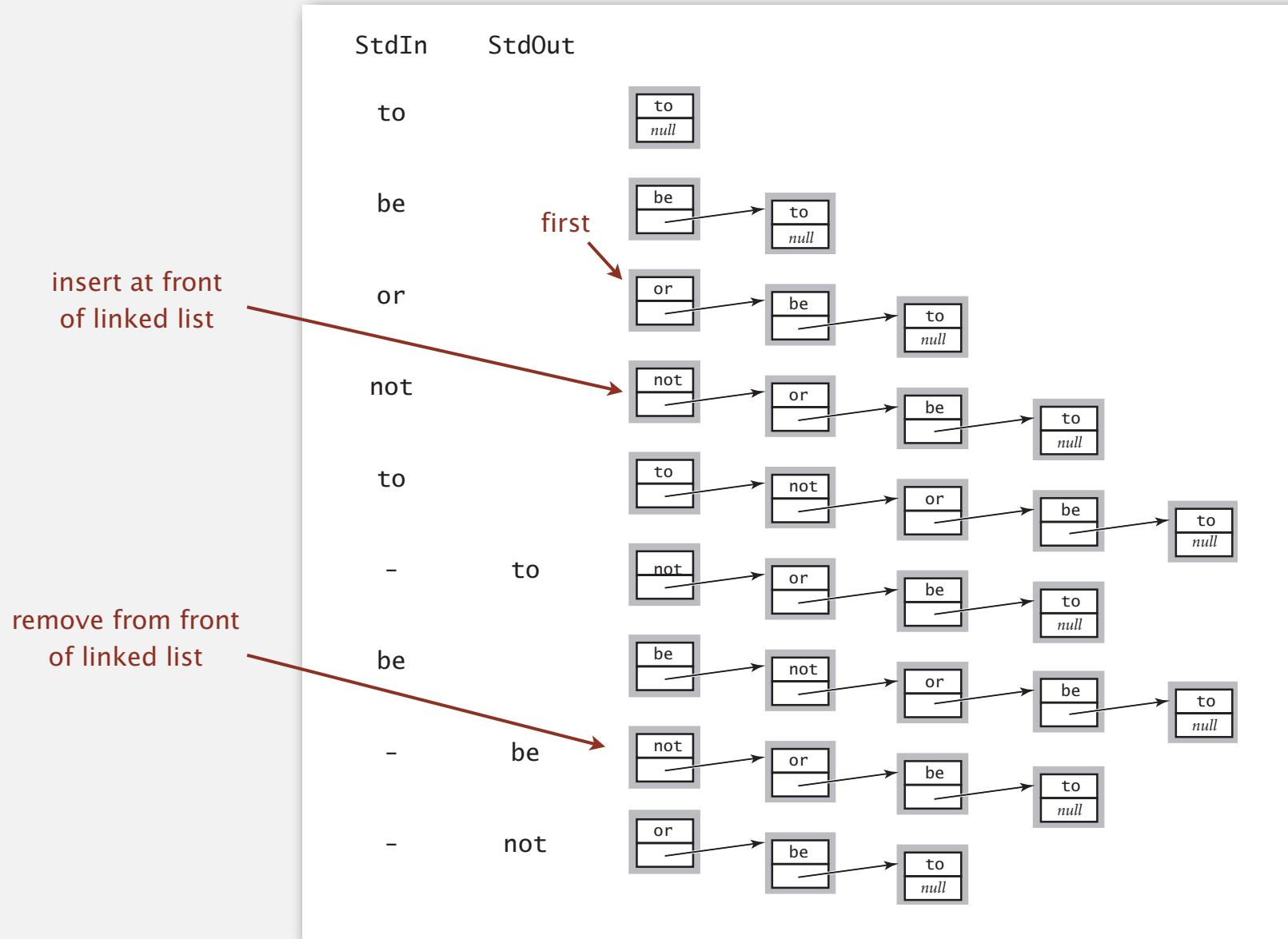


```
% more tobe.txt
to be or not to - be - - that - - - is

% java StackOfStrings < tobe.txt
to be not that or be
```

# Stack: linked-list representation

Maintain pointer to first node in a linked list; insert/remove from front.



# Stack pop: linked-list implementation

inner class

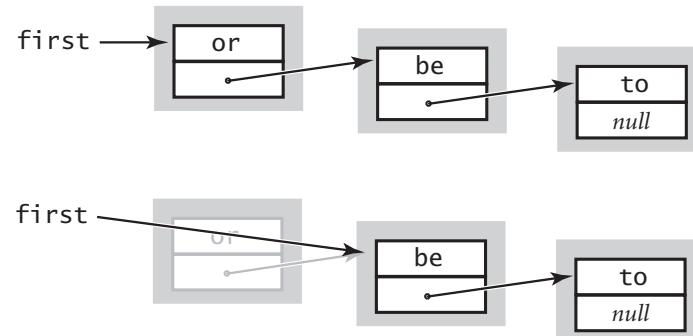
```
private class Node  
{  
    String item;  
    Node next;  
}
```

save item to return

```
String item = first.item;
```

delete first node

```
first = first.next;
```



return saved item

```
return item;
```

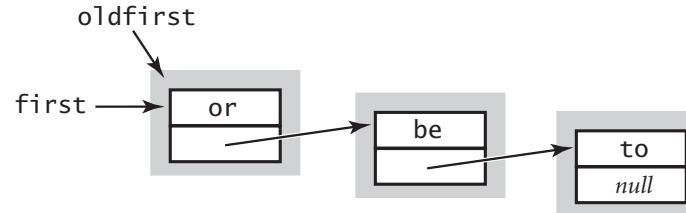
# Stack push: linked-list implementation

inner class

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

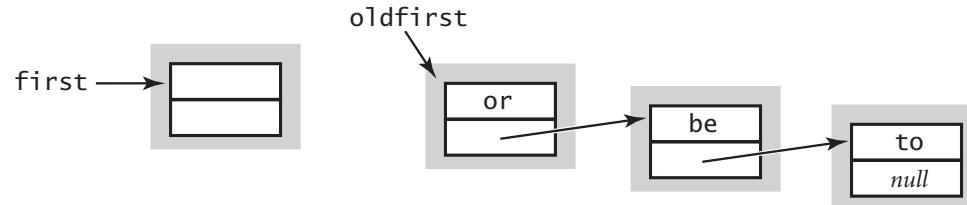
save a link to the list

```
Node oldfirst = first;
```



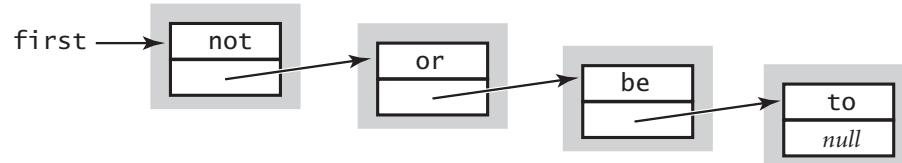
create a new node for the beginning

```
first = new Node();
```



set the instance variables in the new node

```
first.item = "not";  
first.next = oldfirst;
```



# Stack: linked-list implementation in Java

```
public class LinkedStackOfStrings
{
    private Node first = null;

    private class Node
    {
        String item;
        Node next;
    }

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

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

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

private inner class  
(access modifiers don't matter)

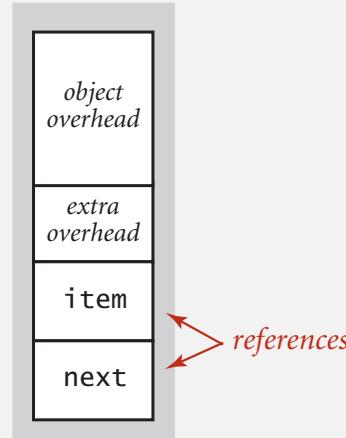


# Stack: linked-list implementation performance

**Proposition.** Every operation takes constant time in the worst case.

**Proposition.** A stack with  $N$  items uses  $\sim 40N$  bytes.

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



16 bytes (object overhead)

8 bytes (inner class extra overhead)

8 bytes (reference to String)

8 bytes (reference to Node)

---

40 bytes per stack node

**Remark.** This accounts for the memory for the stack  
(but not the memory for strings themselves, which the client owns).

# Stack: array implementation

---

## Array implementation of a stack.

- Use array  $s[]$  to store  $N$  items on stack.
- $\text{push}()$ : add new item at  $s[N]$ .
- $\text{pop}()$ : remove item from  $s[N-1]$ .

$s[]$	to	be	or	not	to	be	<i>null</i>	<i>null</i>	<i>null</i>	<i>null</i>
	0	1	2	3	4	5	6	7	8	9
							$N$			$\text{capacity} = 10$

Defect. Stack overflows when  $N$  exceeds capacity. [stay tuned]

# Stack: array implementation

```
public class FixedCapacityStackOfStrings
{
    private String[] s;
    private int N = 0;

    public FixedCapacityStackOfStrings(int capacity)
    {   s = new String[capacity]; }
```

a cheat  
(stay tuned)

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

```
public void push(String item)
{   s[N++] = item; }
```

use to index into array;  
then increment N

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

```
}
```

decrement N;  
then use to index into array

# Stack considerations

---

## Overflow and underflow.

- Underflow: throw exception if pop from an empty stack.
- Overflow: use resizing array for array implementation. [stay tuned]

Null items. We allow null items to be inserted.

Loitering. Holding a reference to an object when it is no longer needed.

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

loitering

```
public String pop()
{
    String item = s[--N];
    s[N] = null;
    return item;
}
```

this version avoids "loitering":  
garbage collector can reclaim memory  
only if no outstanding references

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 1.3 BAGS, QUEUES, AND STACKS

---

- ▶ *stacks*
- ▶ *resizing arrays*
- ▶ *queues*
- ▶ *generics*
- ▶ *iterators*
- ▶ *applications*

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 1.3 BAGS, QUEUES, AND STACKS

---

- ▶ *stacks*
- ▶ *resizing arrays*
- ▶ *queues*
- ▶ *generics*
- ▶ *iterators*
- ▶ *applications*

## Stack: resizing-array implementation

---

**Problem.** Requiring client to provide capacity does not implement API!

**Q.** How to grow and shrink array?

**First try.**

- `push()`: increase size of array `s[]` by 1.
- `pop()`: decrease size of array `s[]` by 1.

**Too expensive.**

- Need to copy all items to a new array.
- Inserting first  $N$  items takes time proportional to  $1 + 2 + \dots + N \sim N^2/2$ .

  
infeasible for large  $N$

**Challenge.** Ensure that array resizing happens infrequently.

# Stack: resizing-array implementation

Q. How to grow array?

A. If array is full, create a new array of **twice** the size, and copy items.

"repeated doubling"

```
public ResizingArrayStackOfStrings()
{   s = new String[1]; }

public void push(String item)
{
    if (N == s.length) resize(2 * s.length);
    s[N++] = item;
}

private void resize(int capacity)
{
    String[] copy = new String[capacity];
    for (int i = 0; i < N; i++)
        copy[i] = s[i];
    s = copy;
}
```

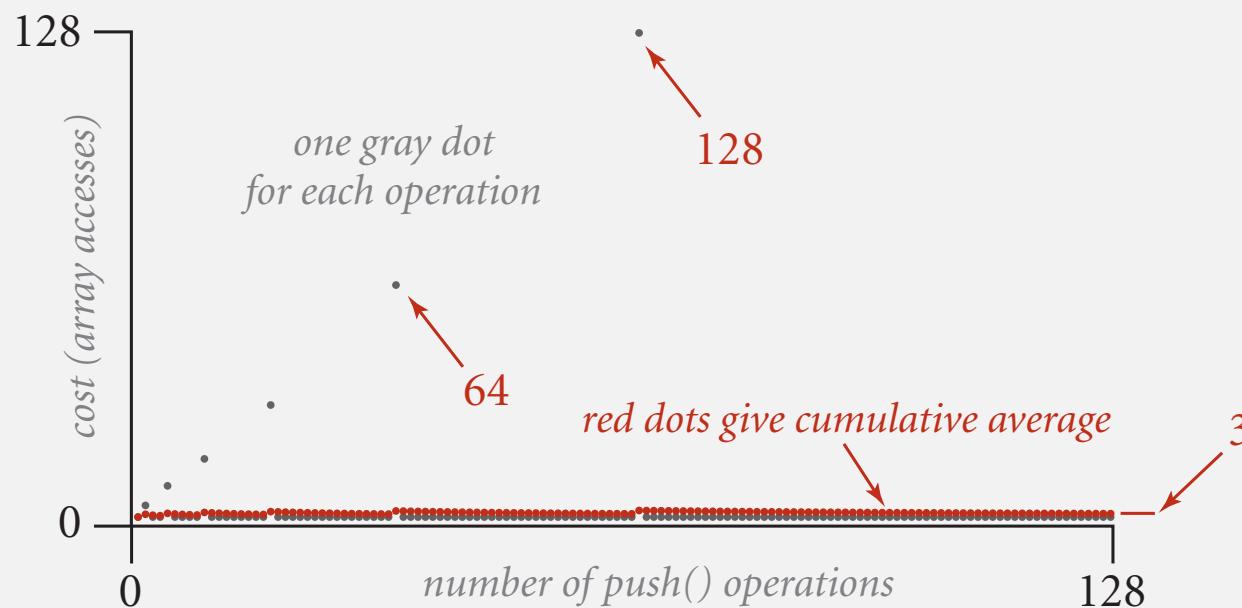
see next slide

Consequence. Inserting first  $N$  items takes time proportional to  $N$  (not  $N^2$ ).

# Stack: amortized cost of adding to a stack

Cost of inserting first  $N$  items.  $N + (2 + 4 + 8 + \dots + N) \sim 3N.$

↑  
1 array access  
per push                           ↑  
k array accesses to double to size k  
(ignoring cost to create new array)



# Stack: resizing-array implementation

---

Q. How to shrink array?

First try.

- `push()`: double size of array `s[]` when array is full.
- `pop()`: halve size of array `s[]` when array is **one-half full**.

Too expensive in worst case.

- Consider push-pop-push-pop-... sequence when array is full.
- Each operation takes time proportional to  $N$ .

$N = 5$	to	be	or	not	to	null	null	null
---------	----	----	----	-----	----	------	------	------

$N = 4$	to	be	or	not
---------	----	----	----	-----

$N = 5$	to	be	or	not	to	null	null	null
---------	----	----	----	-----	----	------	------	------

$N = 4$	to	be	or	not
---------	----	----	----	-----

# Stack: resizing-array implementation

---

Q. How to shrink array?

Efficient solution.

- `push()`: double size of array `s[]` when array is full.
- `pop()`: halve size of array `s[]` when array is one-quarter full.

```
public String pop()
{
    String item = s[--N];
    s[N] = null;
    if (N > 0 && N == s.length/4) resize(s.length/2);
    return item;
}
```

Invariant. Array is between 25% and 100% full.

# Stack: resizing-array implementation trace

---

push()	pop()	N	a.length	a[]							
				0	1	2	3	4	5	6	7
			0	1	null						
to		1	1	to							
be		2	2	to	be						
or		3	4	to	be	or		null			
not		4	4	to	be	or	not				
to		5	8	to	be	or	not	to	null	null	null
-	to	4	8	to	be	or	not	null	null	null	null
be		5	8	to	be	or	not	be	null	null	null
-	be	4	8	to	be	or	not	null	null	null	null
-	not	3	8	to	be	or	null	null	null	null	null
that		4	8	to	be	or	that	null	null	null	null
-	that	3	8	to	be	or	null	null	null	null	null
-	or	2	4	to	be	null	null				
-	be	1	2	to	null						
is		2		to	is						

Trace of array resizing during a sequence of push() and pop() operations

## Stack resizing-array implementation: performance

**Amortized analysis.** Average running time per operation over a worst-case sequence of operations.

**Proposition.** Starting from an empty stack, any sequence of  $M$  push and pop operations takes time proportional to  $M$ .

	best	worst	amortized
construct	1	1	1
push	1	$N$	1
pop	1	$N$	1
size	1	1	1

**order of growth of running time  
for resizing stack with  $N$  items**

doubling and halving operations

## Stack resizing-array implementation: memory usage

**Proposition.** Uses between  $\sim 8N$  and  $\sim 32N$  bytes to represent a stack with  $N$  items.

- $\sim 8N$  when full.
- $\sim 32N$  when one-quarter full.

```
public class ResizingArrayStackOfStrings
{
    private String[] s;
    private int N = 0;
    ...
}
```

8 bytes (reference to array)  
24 bytes (array overhead)  
8 bytes × array size  
4 bytes (int)  
4 bytes (padding)

**Remark.** This accounts for the memory for the stack  
(but not the memory for strings themselves, which the client owns).

# Stack implementations: resizing array vs. linked list

**Tradeoffs.** Can implement a stack with either resizing array or linked list; client can use interchangeably. Which one is better?

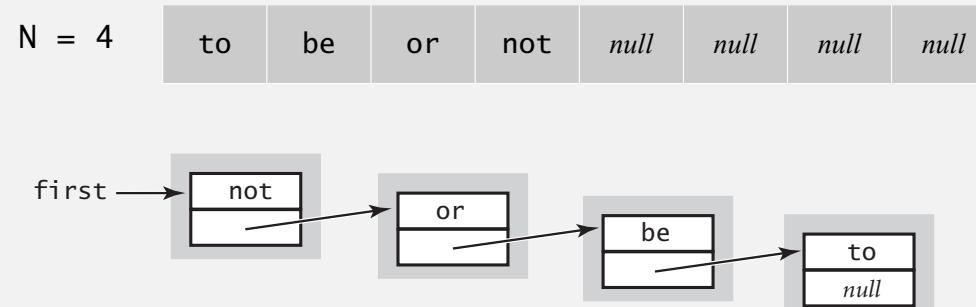
## Linked-list implementation.

- Every operation takes constant time in the **worst case**.
- Uses extra time and space to deal with the links.

## Resizing-array implementation.

- Every operation takes constant **amortized time**.
- Less wasted space.

If need to response quickly, use linked-list.  
If need to reduce the total runtime, use resizing-array



# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 1.3 BAGS, QUEUES, AND STACKS

---

- ▶ *stacks*
- ▶ *resizing arrays*
- ▶ *queues*
- ▶ *generics*
- ▶ *iterators*
- ▶ *applications*

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 1.3 BAGS, QUEUES, AND STACKS

---

- ▶ *stacks*
- ▶ *resizing arrays*
- ▶ *queues*
- ▶ *generics*
- ▶ *iterators*
- ▶ *applications*

# Queue API

```
public class QueueOfStrings
```

```
    QueueOfStrings()
```

*create an empty queue*

```
    void enqueue(String item)
```

*insert a new string onto queue*

```
    String dequeue()
```

*remove and return the string  
least recently added*

```
    boolean isEmpty()
```

*is the queue empty?*

```
    int size()
```

*number of strings on the queue*

enqueue

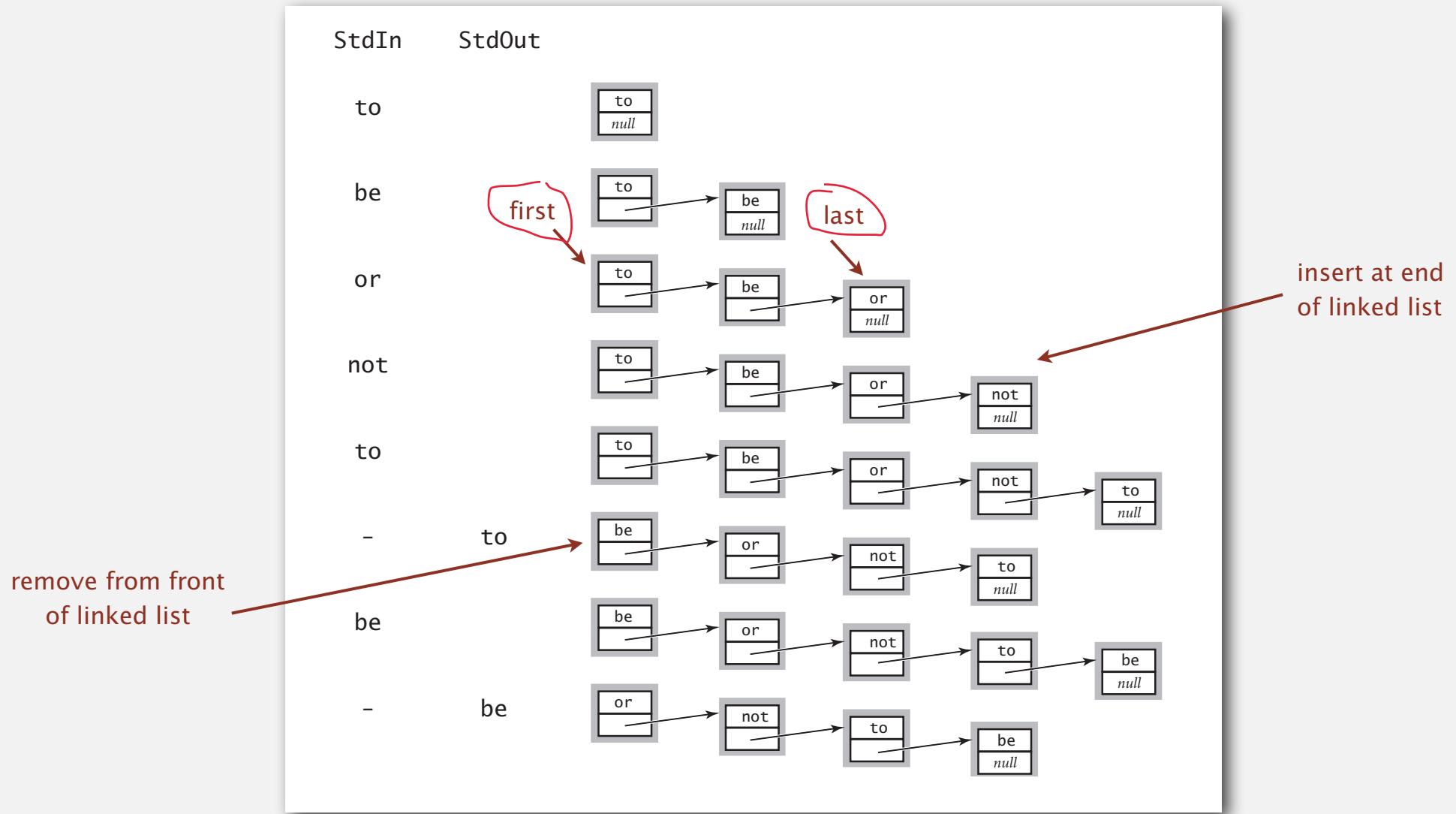


dequeue



# Queue: linked-list representation

Maintain pointer to first and last nodes in a linked list;  
insert/remove from opposite ends.



# Queue dequeue: linked-list implementation

inner class

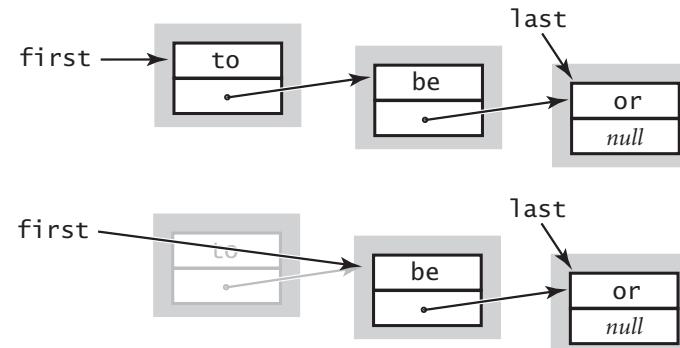
```
private class Node  
{  
    String item;  
    Node next;  
}
```

save item to return

```
String item = first.item;
```

delete first node

```
first = first.next;
```



return saved item

```
return item;
```

Remark. Identical code to linked-list stack pop().

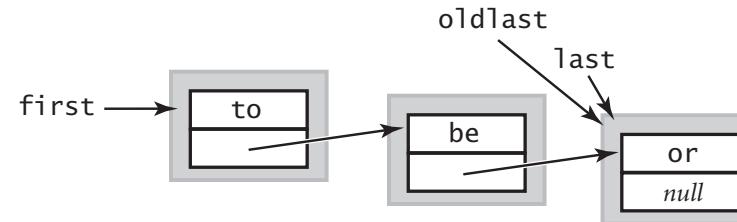
# Queue enqueue: linked-list implementation

## inner class

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

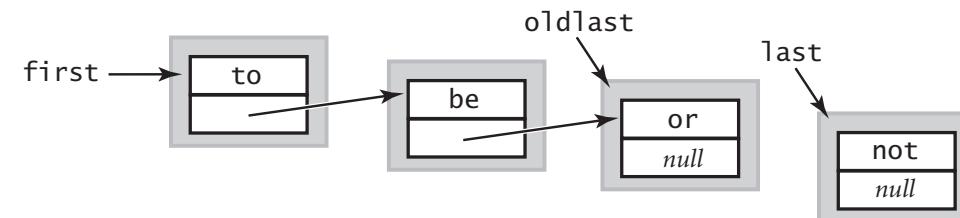
### save a link to the last node

```
Node oldlast = last;
```



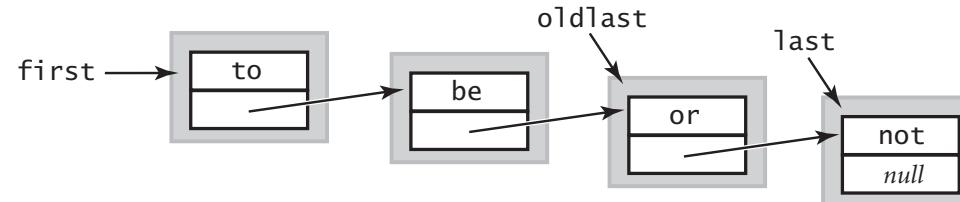
### create a new node for the end

```
last = new Node();  
last.item = "not";
```



### link the new node to the end of the list

```
oldlast.next = last;
```



# Queue: linked-list implementation in Java

```
public class LinkedQueueOfStrings
{
    private Node first, last;

    private class Node
    { /* same as in StackOfStrings */ }

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

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

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

special cases for empty queue

# Queue: resizing array implementation

---

Array implementation of a queue.

- Use array `q[]` to store items in queue.
- `enqueue()`: add new item at `q[tail]`.
- `dequeue()`: remove item from `q[head]`.
- Update head and tail modulo the capacity.
- Add resizing array.



Q. How to resize?

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 1.3 BAGS, QUEUES, AND STACKS

---

- ▶ *stacks*
- ▶ *resizing arrays*
- ▶ *queues*
- ▶ *generics*
- ▶ *iterators*
- ▶ *applications*

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 1.3 BAGS, QUEUES, AND STACKS

---

- ▶ *stacks*
- ▶ *resizing arrays*
- ▶ *queues*
- ▶ *generics*
- ▶ *iterators*
- ▶ *applications*

# Parameterized stack

---

We implemented: StackOfStrings.

We also want: StackOfURLs, StackOfInts, StackOfVans, ....

Attempt 1. 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.

@#\$\*! most reasonable approach until Java 1.5.



# Parameterized stack

---

We implemented: StackOfStrings.

We also want: StackOfURLs, StackOfInts, StackOfVans, ....

Attempt 2. Implement a stack with items of type Object.

- Casting is required in client.
- Casting is error-prone: run-time error if types mismatch.

```
StackOfObjects s = new StackOfObjects();
Apple a = new Apple();
Orange b = new Orange();
s.push(a);
s.push(b);
a = (Apple) (s.pop());
```

run-time error



## Parameterized stack

---

We implemented: StackOfStrings.

We also want: StackOfURLs, StackOfInts, StackOfVans, ....

Attempt 3. Java generics.

- Avoid casting in client.
- Discover type mismatch errors at compile-time instead of run-time.

The diagram shows a code snippet for a parameterized stack:

```
Stack<Apple> s = new Stack<Apple>();
Apple a = new Apple();
Orange b = new Orange();
s.push(a);
s.push(b);
a = s.pop();
```

Annotations:

- A red arrow points from the text "type parameter" to the generic type parameter `<Apple>` in the declaration `Stack<Apple>`.
- A red arrow points from the text "compile-time error" to the line `s.push(b);`, indicating that this assignment would result in a compile-time error due to type mismatch.

Guiding principles. Welcome compile-time errors; avoid run-time errors.

# Generic stack: linked-list implementation

```
public class LinkedStackOfStrings
{
    private Node first = null;

    private class Node
    {
        String item;
        Node next;
    }

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

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

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

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

    private class Node
    {
        Item item;
        Node next;
    }

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

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

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

generic type name

# Generic stack: array implementation

```
public class FixedCapacityStackOfStrings
{
    private String[] s;
    private int N = 0;

    public ..StackOfStrings(int capacity)
    {   s = new String[capacity];   }

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

    public void push(String item)
    {   s[N++] = item;   }

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

the way it should be

```
public class FixedCapacityStack<Item>
{
    private Item[] s;
    private int N = 0;

    public FixedCapacityStack(int capacity)
    {   s = new Item[capacity];   }

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

    public void push(Item item)
    {   s[N++] = item;   }

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

@#\$\*! generic array creation not allowed in Java

# Generic stack: array implementation

```
public class FixedCapacityStackOfStrings
{
    private String[] s;
    private int N = 0;

    public ..StackOfStrings(int capacity)
    {   s = new String[capacity];   }

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

    public void push(String item)
    {   s[N++] = item;   }

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

the way it is

```
public class FixedCapacityStack<Item>
{
    private Item[] s;
    private int N = 0;

    public FixedCapacityStack(int capacity)
    {   s = (Item[]) new Object[capacity];   }

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

    public void push(Item item)
    {   s[N++] = item;   }

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

the ugly cast

## Unchecked cast

---

```
% javac FixedCapacityStack.java
```

Note: FixedCapacityStack.java uses unchecked or unsafe operations.

Note: Recompile with -Xlint:unchecked for details.

```
% javac -Xlint:unchecked FixedCapacityStack.java
```

FixedCapacityStack.java:26: warning: [unchecked] unchecked cast

found : java.lang.Object[]

required: Item[]

```
    a = (Item[]) new Object[capacity];
```

^

1 warning

## Generic data types: autoboxing

---

Q. What to do about primitive types?

Wrapper type.

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

Autoboxing. Automatic cast between a primitive type and its wrapper.

```
Stack<Integer> s = new Stack<Integer>();
s.push(17);           // s.push(Integer.valueOf(17));
int a = s.pop();     // int a = s.pop().intValue();
```

Bottom line. Client code can use generic stack for **any** type of data.

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 1.3 BAGS, QUEUES, AND STACKS

---

- ▶ *stacks*
- ▶ *resizing arrays*
- ▶ *queues*
- ▶ *generics*
- ▶ *iterators*
- ▶ *applications*

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 1.3 BAGS, QUEUES, AND STACKS

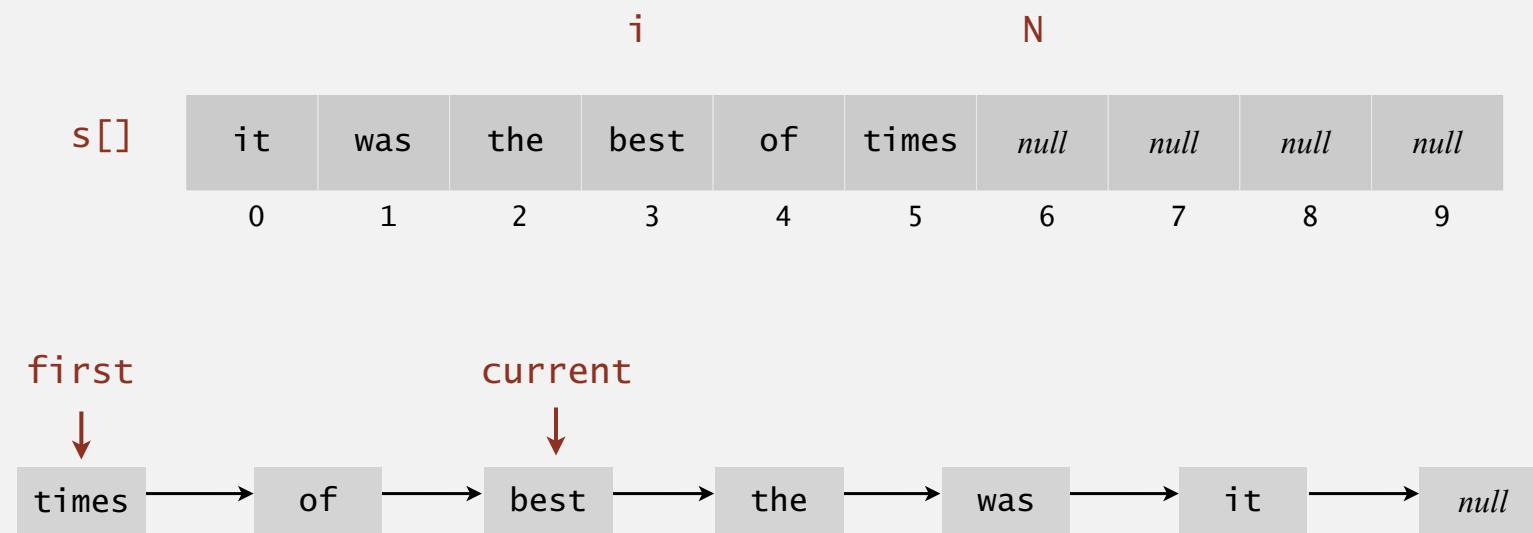
---

- ▶ *stacks*
- ▶ *resizing arrays*
- ▶ *queues*
- ▶ *generics*
- ▶ *iterators*
- ▶ *applications*

# Iteration

---

**Design challenge.** Support iteration over stack items by client, without revealing the internal representation of the stack.



**Java solution.** Make stack implement the `java.lang.Iterable` interface.

# Iterators

Q. What is an **Iterable** ?

A. Has a method that returns an **Iterator**.

## Iterable interface

```
public interface Iterable<Item>
{
    Iterator<Item> iterator();
}
```

Q. What is an **Iterator** ?

A. Has methods `hasNext()` and `next()`.

## Iterator interface

```
public interface Iterator<Item>
{
    boolean hasNext();
    Item next();
    void remove(); ← optional; use
                           at your own risk
}
```

Q. Why make data structures **Iterable** ?

A. Java supports elegant client code.

## “foreach” statement (shorthand)

```
for (String s : stack)
    StdOut.println(s);
```

## equivalent code (longhand)

```
Iterator<String> i = stack.iterator();
while (i.hasNext())
{
    String s = i.next();
    StdOut.println(s);
}
```

## Stack iterator: linked-list implementation

```
import java.util.Iterator;

public class Stack<Item> implements Iterable<Item>
{
    ...

    public Iterator<Item> iterator() { return new ListIterator(); }

    private class ListIterator implements Iterator<Item>
    {
        private Node current = first;

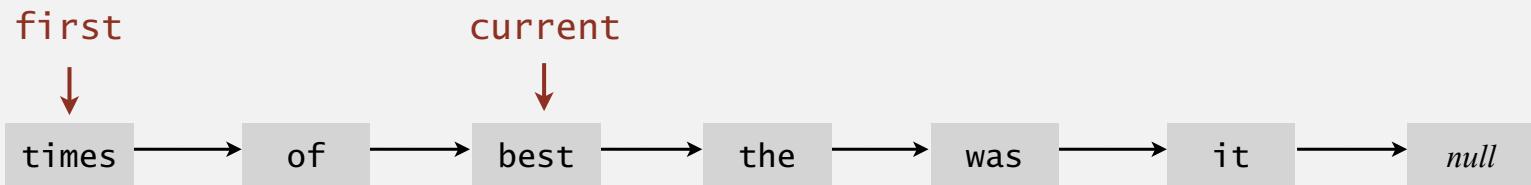
        public boolean hasNext() { return current != null; }

        public void remove() { /* not supported */ }

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

Diagram illustrating the state of the stack during iteration:

- The `first` pointer points to the head of the list.
- The `current` pointer points to the current element being iterated over.
- The list consists of nodes containing the words: `times`, `of`, `best`, `the`, `was`, `it`, and `null`.
- An arrow points from the `remove()` method to a note: "throw UnsupportedOperationException".
- An arrow points from the `next()` method to a note: "throw NoSuchElementException if no more items in iteration".



## Stack iterator: array implementation

```
import java.util.Iterator;

public class Stack<Item> implements Iterable<Item>
{
    ...

    public Iterator<Item> iterator()
    { return new ReverseArrayIterator(); }

    private class ReverseArrayIterator implements Iterator<Item>
    {
        private int i = N;

        public boolean hasNext() { return i > 0; }
        public void remove()    { /* not supported */ }
        public Item next()      { return s[--i]; }
    }
}
```



# Bag API

Main application. Adding items to a collection and iterating (when order doesn't matter).

```
public class Bag<Item> implements Iterable<Item>
```

```
    Bag()
```

*create an empty bag*

```
    void add(Item x)
```

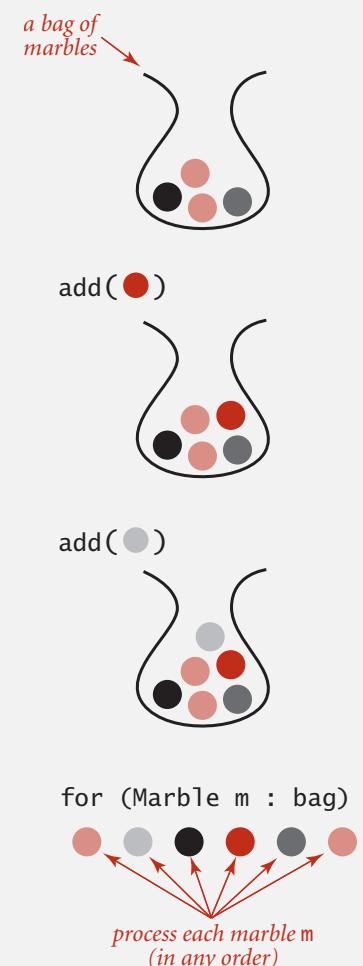
*insert a new item onto bag*

```
    int size()
```

*number of items in bag*

```
    Iterable<Item> iterator()
```

*iterator for all items in bag*



Implementation. Stack (without pop) or queue (without dequeue).

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 1.3 BAGS, QUEUES, AND STACKS

---

- ▶ *stacks*
- ▶ *resizing arrays*
- ▶ *queues*
- ▶ *generics*
- ▶ *iterators*
- ▶ *applications*

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 1.3 BAGS, QUEUES, AND STACKS

---

- ▶ *stacks*
- ▶ *resizing arrays*
- ▶ *queues*
- ▶ *generics*
- ▶ *iterators*
- ▶ *applications*

# Java collections library

List interface. `java.util.List` is API for an sequence of items.

public interface List<Item> implements Iterable<Item>	
<code>List()</code>	<i>create an empty list</i>
<code>boolean isEmpty()</code>	<i>is the list empty?</i>
<code>int size()</code>	<i>number of items</i>
<code>void add(Item item)</code>	<i>append item to the end</i>
<code>Item get(int index)</code>	<i>return item at given index</i>
<code>Item remove(int index)</code>	<i>return and delete item at given index</i>
<code>boolean contains(Item item)</code>	<i>does the list contain the given item?</i>
<code>Iterator&lt;Item&gt; iterator()</code>	<i>iterator over all items in the list</i>
<code>...</code>	

Implementations. `java.util.ArrayList` uses resizing array;

`java.util.LinkedList` uses linked list.

caveat: only some operations are efficient

# Java collections library

---

## `java.util.Stack`.

- Supports push(), pop(), and and iteration.
- Extends `java.util.Vector`, which implements `java.util.List` interface from previous slide, including, get() and remove().
- Bloated and poorly-designed API (why?)



## `java.util.Queue`. An interface, not an implementation of a queue.

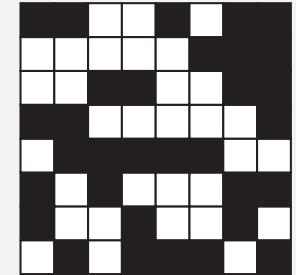
**Best practices.** Use our implementations of Stack, Queue, and Bag.

## War story (from Assignment 1)

---

Generate random open sites in an  $N$ -by- $N$  percolation system.

- Jenny: pick  $(i, j)$  at random; if already open, repeat.  
Takes  $\sim c_1 N^2$  seconds.
- Kenny: create a `java.util.ArrayList` of  $N^2$  closed sites.  
Pick an index at random and delete.  
Takes  $\sim c_2 N^4$  seconds.



Why is my program so slow?



Kenny

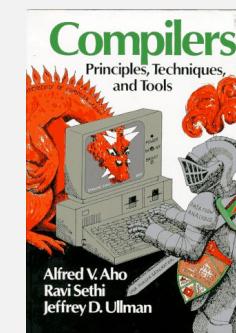
Lesson. Don't use a library until you understand its API!

This course. Can't use a library until we've implemented it in class.

# Stack 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.
- ...



# Function calls

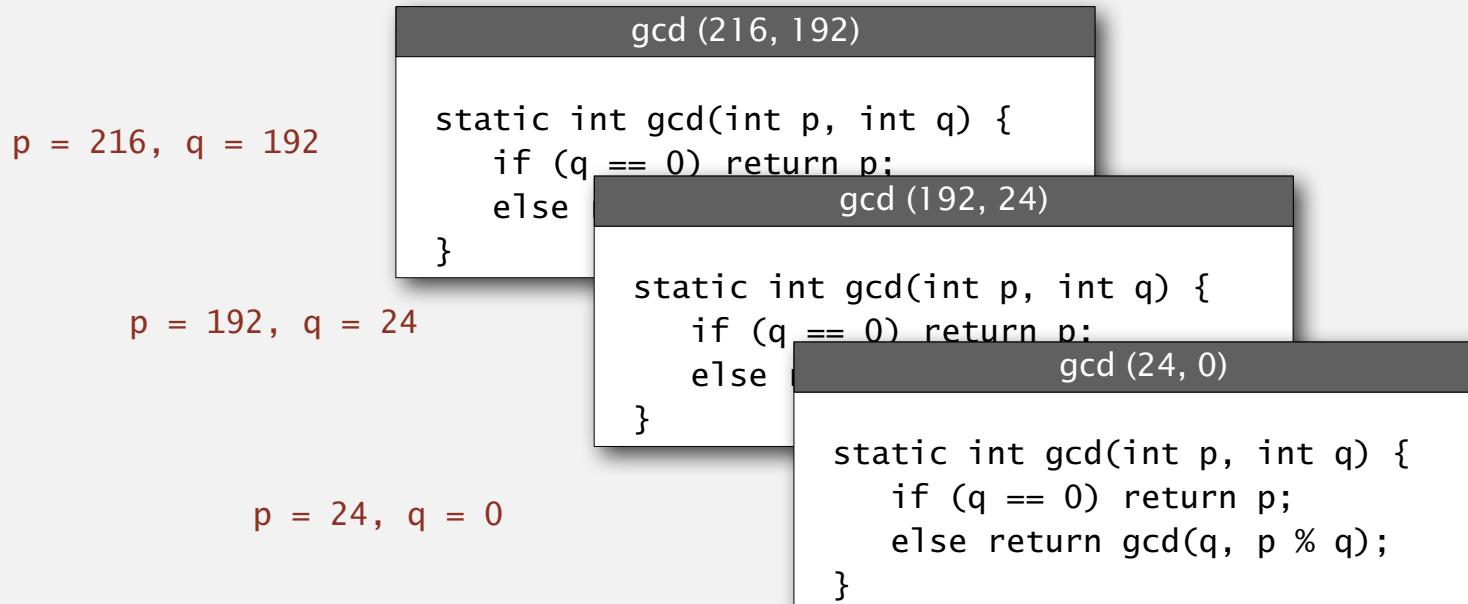
---

How a compiler implements a function.

- 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.



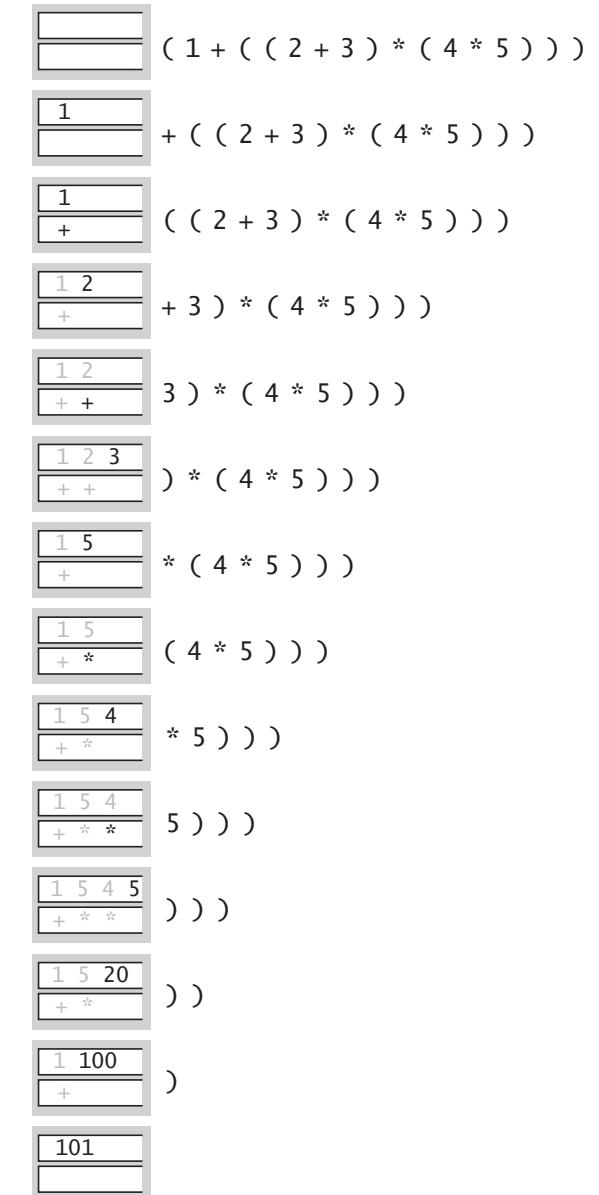
# Arithmetic expression evaluation

Goal. Evaluate infix expressions.

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

operand                    operator

value stack  
operator stack



Two-stack algorithm. [E. W. Dijkstra]

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

Context. An interpreter!

# Dijkstra's two-stack algorithm demo

---



**infix expression**  
**(fully parenthesized)**

value stack

operator stack



# 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
```

## Correctness

---

Q. Why correct?

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

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

as if the original input were:

```
( 1 + ( 5 * ( 4 * 5 ) ) )
```

Repeating the argument:

```
( 1 + ( 5 * 20 ) )  
( 1 + 100 )  
101
```

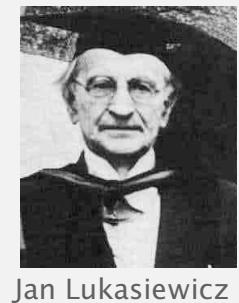
Extensions. More ops, precedence order, associativity.

# Stack-based programming languages

---

**Observation 1.** Dijkstra's two-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 * * +
```

**Bottom line.** Postfix or "reverse Polish" notation.

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

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

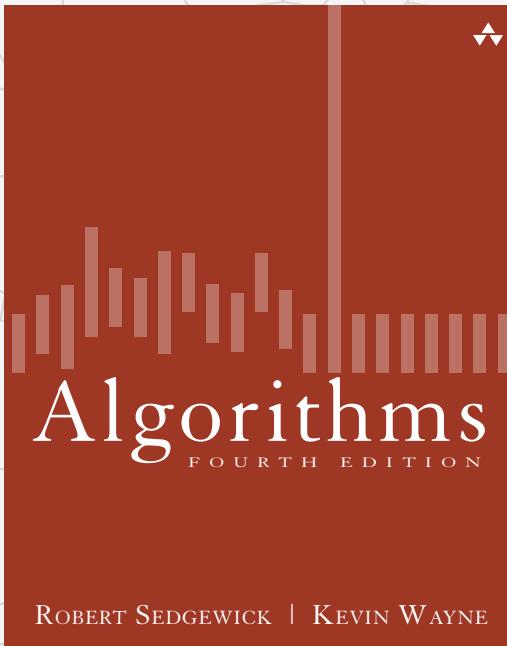
## 1.3 BAGS, QUEUES, AND STACKS

---

- ▶ *stacks*
- ▶ *resizing arrays*
- ▶ *queues*
- ▶ *generics*
- ▶ *iterators*
- ▶ *applications*

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE



<http://algs4.cs.princeton.edu>

## 1.3 BAGS, QUEUES, AND STACKS

---

- ▶ *stacks*
- ▶ *resizing arrays*
- ▶ *queues*
- ▶ *generics*
- ▶ *iterators*
- ▶ *applications*