Algorithms

Robert Sedgewick | Kevin Wayne

http://algs4.cs.princeton.edu

1.3 ~~Bags~~, Queues, and Stacks

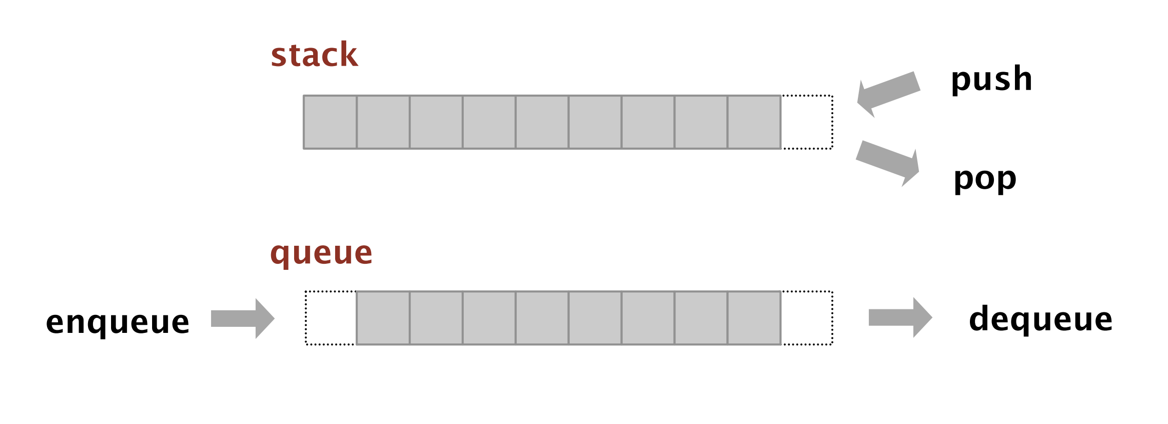
- *stacks*
- *resizing arrays*
- *queues*
- *generics*
- *iterators* ← **see precept**
- *applications*

Last updated on 2/8/16 7:22 AM

Stacks and queues

Abstract data types.

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



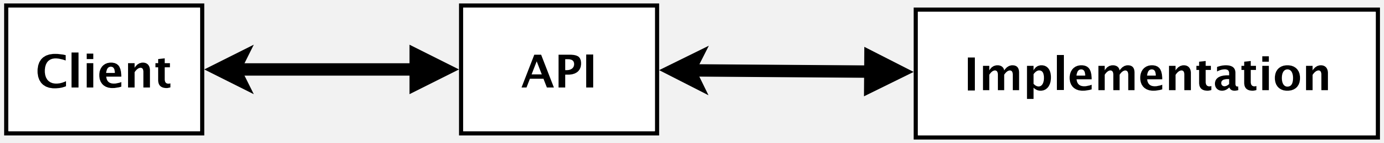
Stack. Examine the item most recently added. ⟵ LIFO = "last in first out"

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

Client, implementation, API

Separate client and implementation via API.

An Abstract Data type is one type of API.



API: description of data type, basic operations.

Client: program using operations defined in API.

Implementation: actual code implementing operations.

Benefits.

- Design: creates modular, reusable libraries.
- Performance: substitute optimized implementation when it matters.

Ex. Stack, queue, bag, priority queue, symbol table, union-find, ...

Interfaces can be ambiguous

Stacks and queues.

- Value: collection of objects.
- Operations: add, remove, iterate, test if empty.

Q. What are two ways in which the semantics of iteration can be ambiguous?

A.

- What order to iterate in: same as removal order or does client not care?
- What happens if collection is modified during iteration?

Java 1.3 bug report (June 27, 2001)

> The iterator method on java.util.Stack iterates through a Stack from the bottom up. One would think that it should iterate as if it were popping off the top of the Stack.

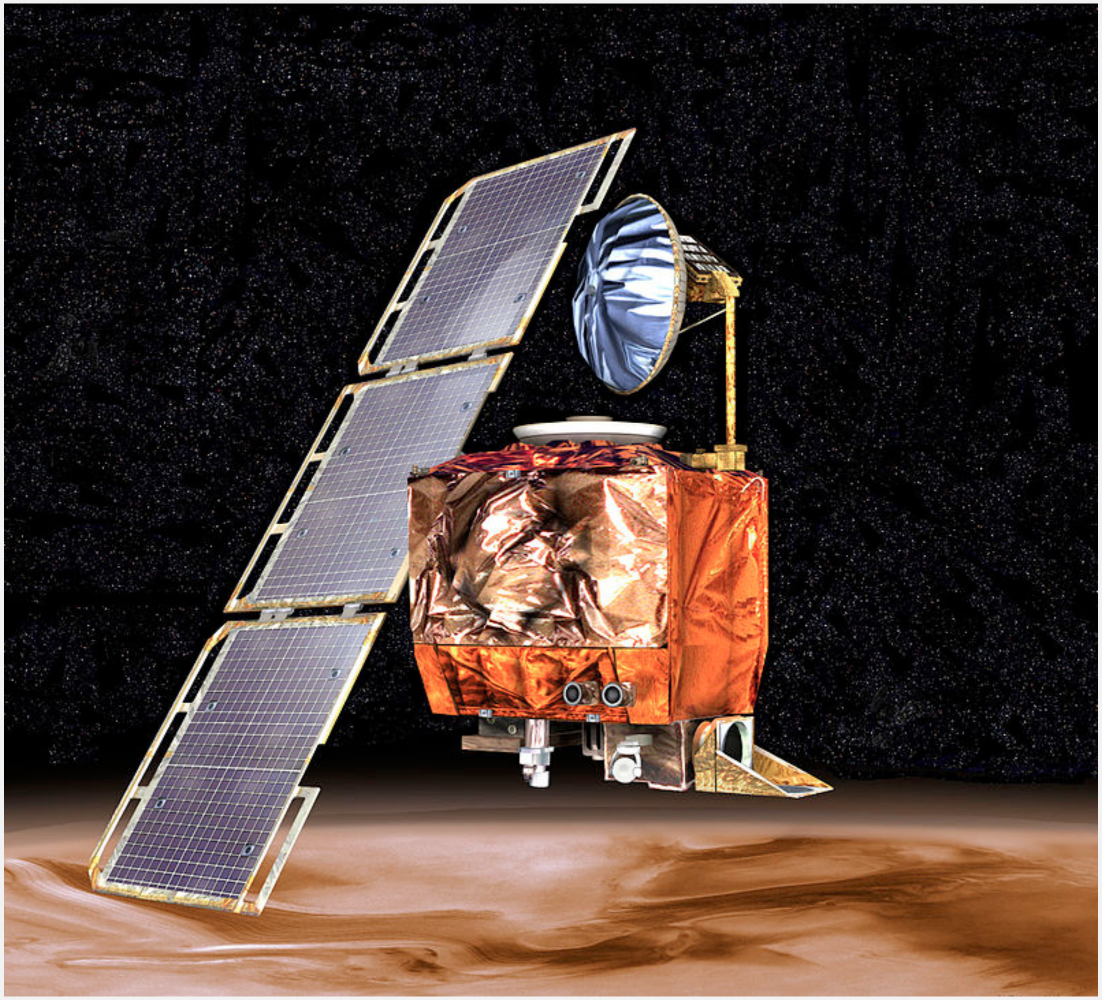
status (closed, will not fix)

> It was an incorrect design decision to have Stack extend Vector ("is-a" rather than "has-a"). We sympathize with the submitter but cannot fix this because of compatibility.

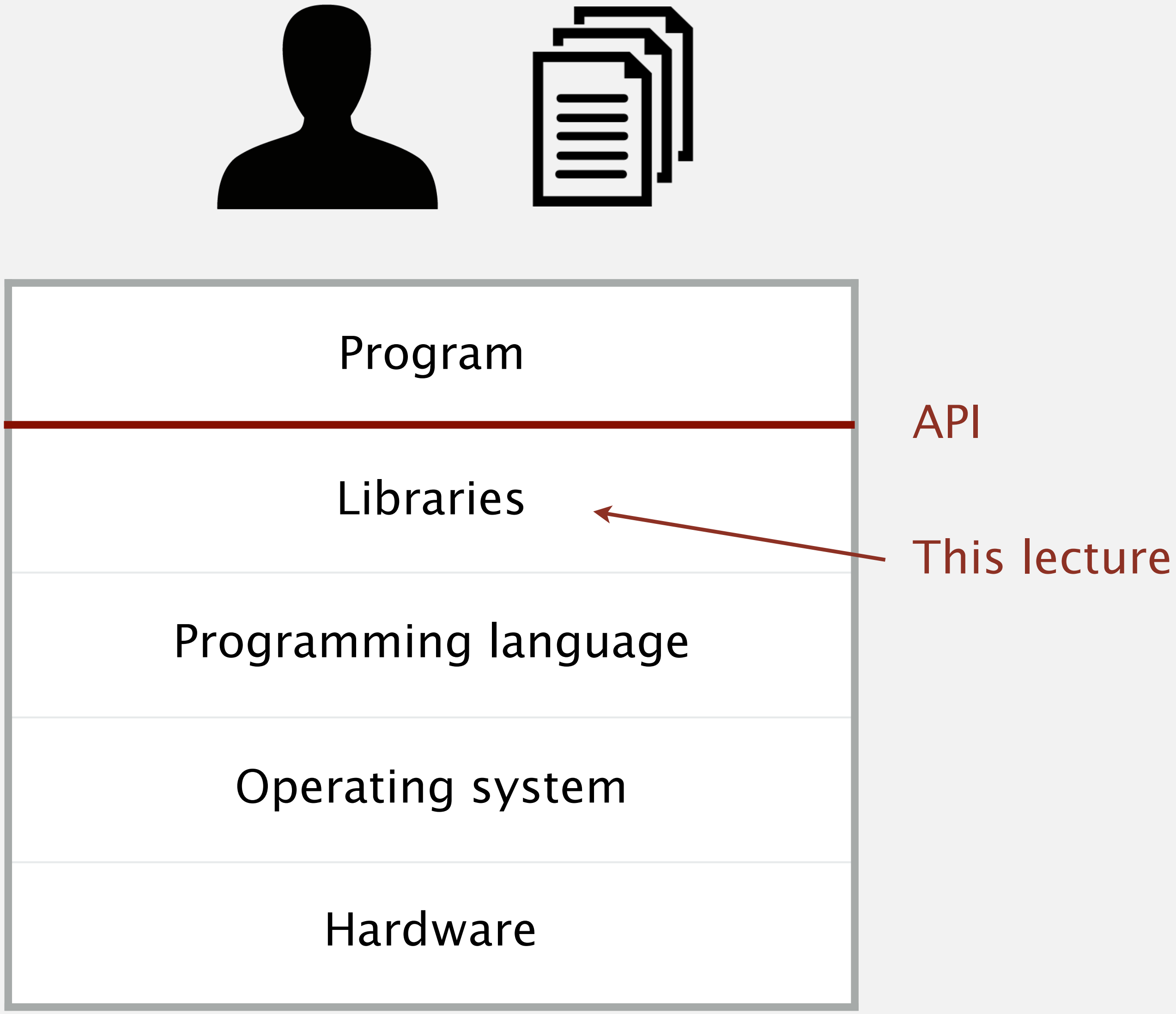
Ambiguity in the semantics of interfaces leads to bugs

Example: Mars climate orbiter.

Lost due to metric vs. imperial mishap in contract between NASA & Lockheed



Layers in a computer system



Program

API

Libraries

This lecture

Programming language

Operating system

Hardware

http://algs4.cs.princeton.edu

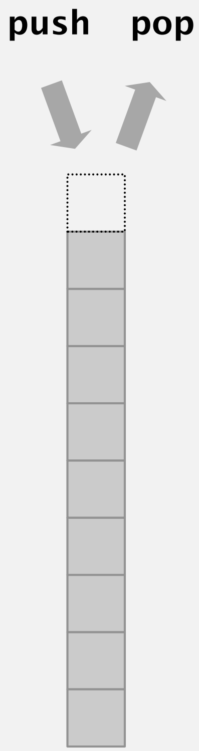
1.3 Bags, Queues, and Stacks

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

Stack API

Warmup API. Stack of strings data type.

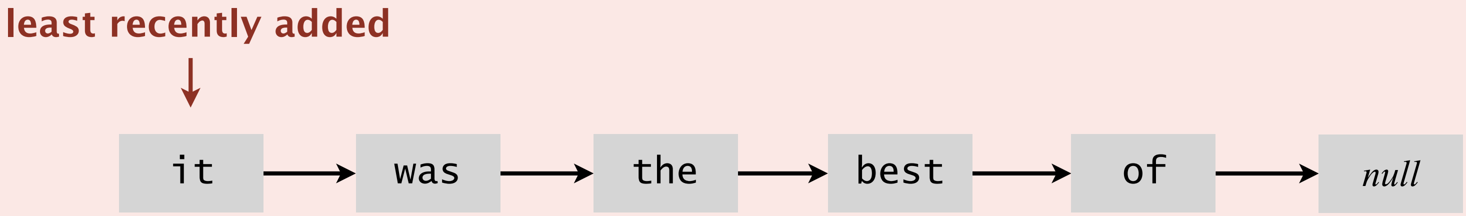
public class StackOfStrings		
	StackOfStrings()	*create an empty stack*
void	push(String item)	*add a new string to stack*
String	pop()	*remove and return the string most recently added*
boolean	isEmpty()	*is the stack empty?*
int	size()	*number of strings on the stack*



Stacks quiz 1

How to implement a stack with a singly-linked list?

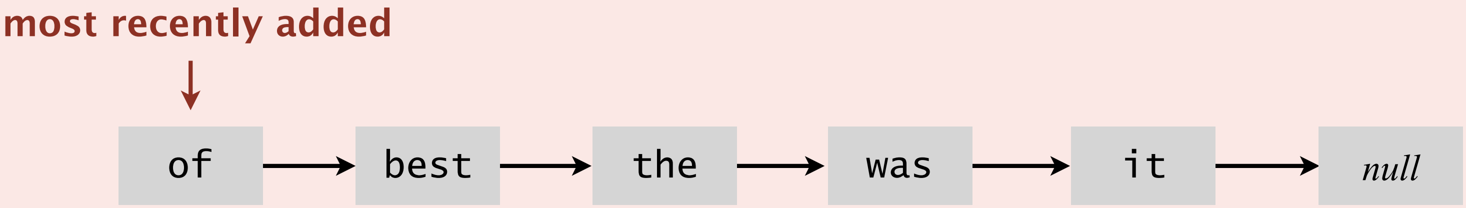
A.



least recently added

it → was → the → best → of → *null*

B.



most recently added

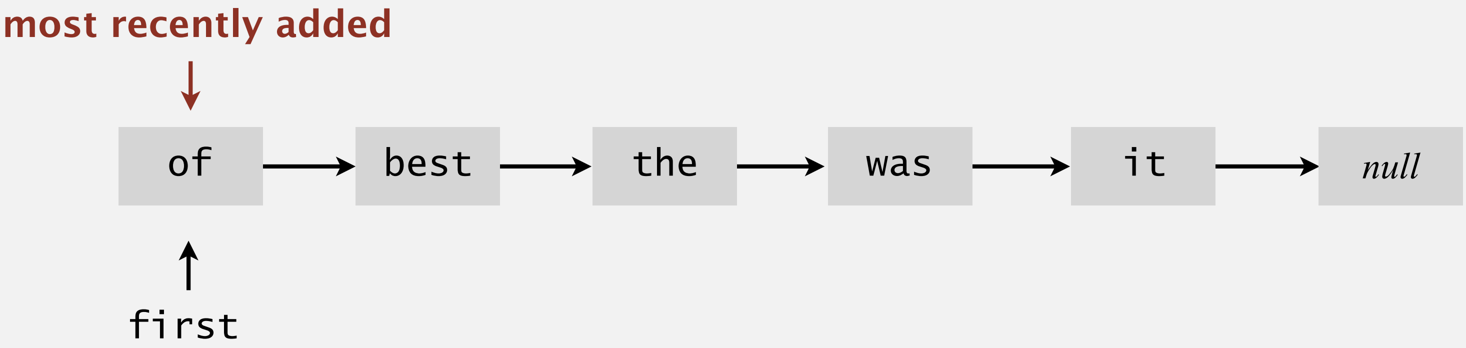
of → best → the → was → it → *null*

C. *None of the above.*

D. *I don't know.*

Stack: linked-list implementation

- Maintain pointer `first` to first node in a singly-linked list.
- Push new item before `first`.
- Pop item from `first`.



Stack pop: linked-list implementation

inner class

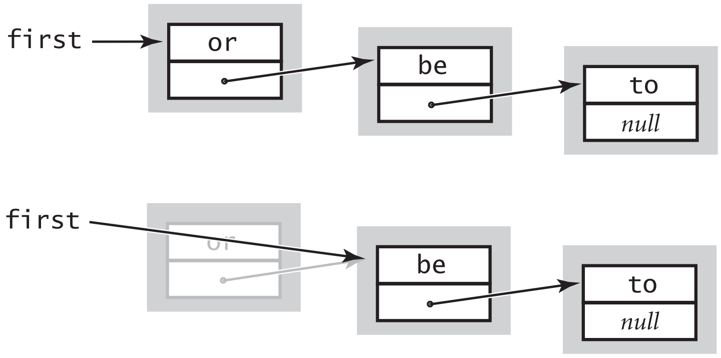
```
private class Node
{
   String item;
   Node next;
}
```

save item to return

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

delete first node

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



return saved item

```
return item;
```

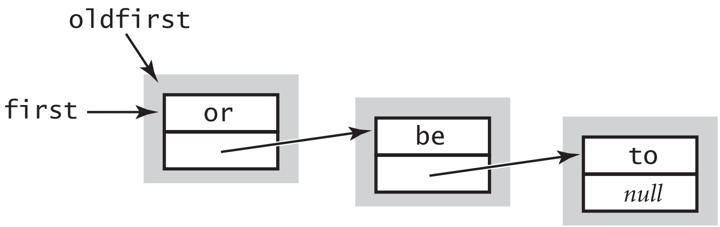
Stack push: linked-list implementation

inner class

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

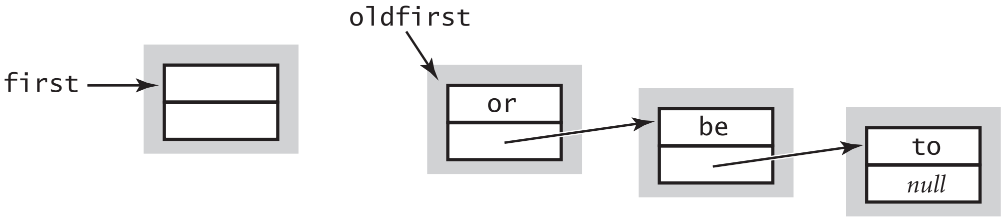
save a link to the list

```
Node oldfirst = first;
```



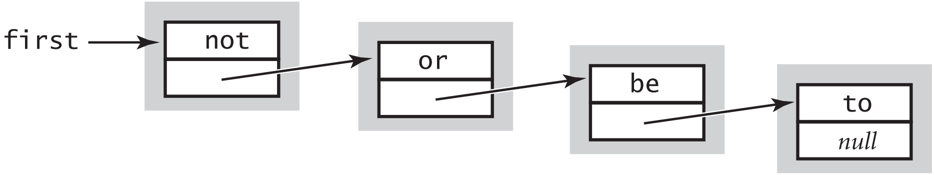
create a new node for the beginning

```
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
   {
      private String item;
      private 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 for instance
variables don't matter)

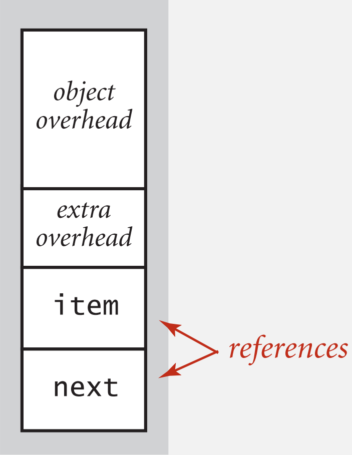
Stack: linked-list implementation performance

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

Proposition. A stack with n items uses ~ $40\ n$ 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)

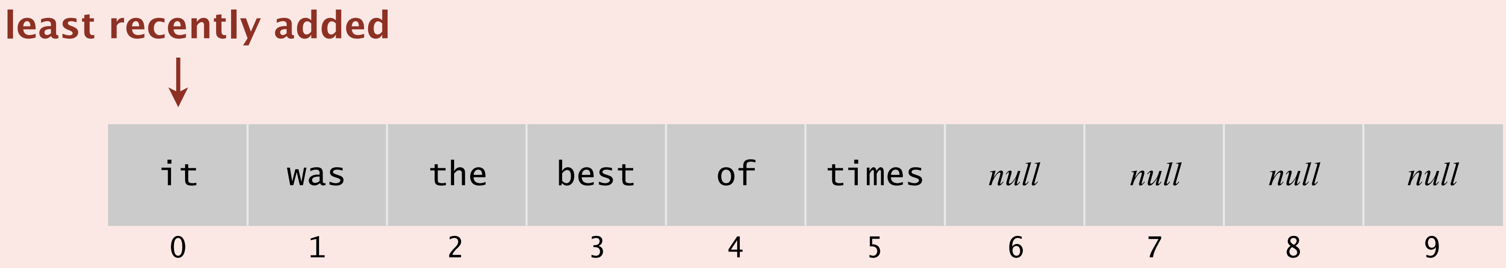
40 bytes allocated per stack node

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

Stacks quiz 2

How to implement a fixed-capacity stack with an array?

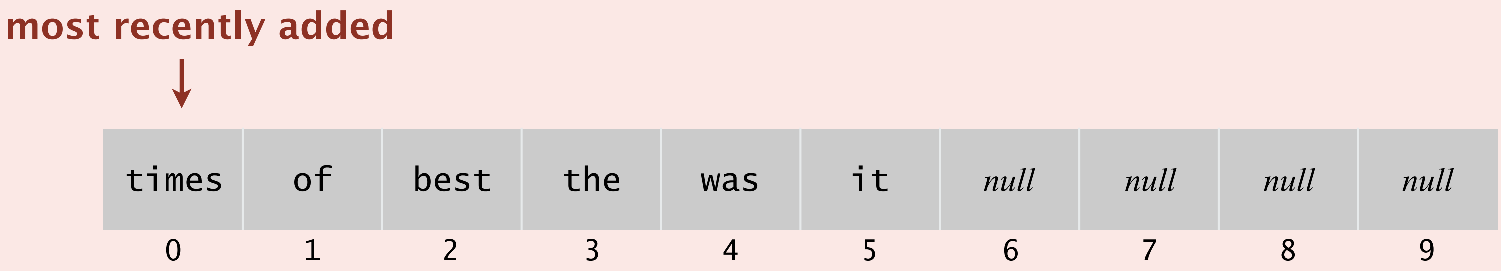
A.



least recently added

it	was	the	best	of	times	*null*	*null*	*null*	*null*
0	1	2	3	4	5	6	7	8	9

B.



most recently added

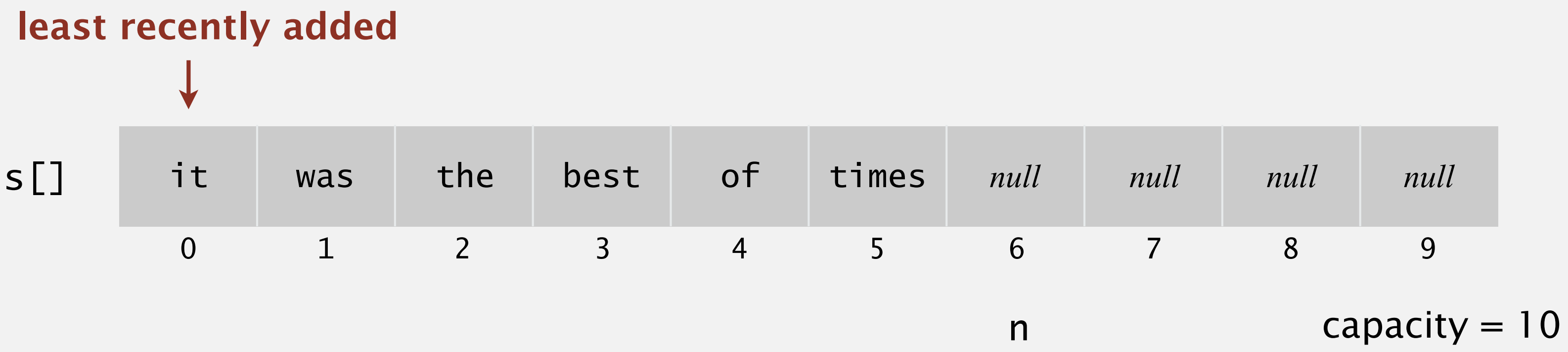
times	of	best	the	was	it	*null*	*null*	*null*	*null*
0	1	2	3	4	5	6	7	8	9

B. *None of the above.*

C. *I don't know.*

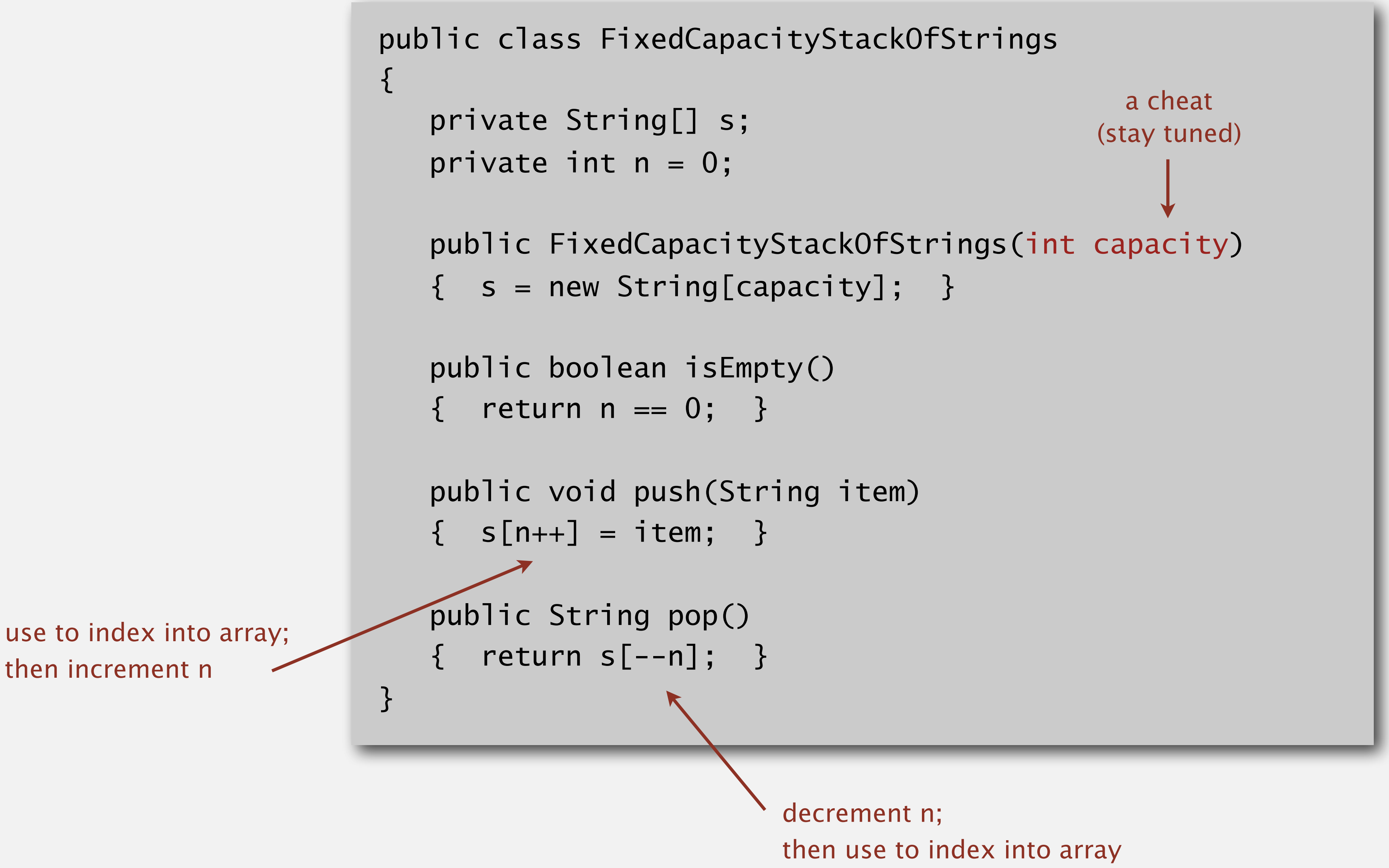
Fixed-capacity stack: array implementation

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



Defect. Stack overflows when `n` exceeds capacity. [stay tuned]

Fixed-capacity stack: array implementation



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

   public FixedCapacityStackOfStrings(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];  }
}
```

a cheat
(stay tuned)

use to index into array;
then increment 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 added.

Duplicate items. We allow an item to be added more than once.

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 for
an object only if no remaining references**

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.

infeasible for large n

- Need to copy all items to a new array, for each operation.
- Array accesses to add first n items = $n + (2 + 4 + \ldots + 2(n-1)) \sim n^2$.

1 array access per push

2(k–1) array accesses to expand to size k (ignoring cost to create new array)

Challenge. Ensure that array resizing happens infrequently.

Why isn't array resizing easier?

Q. Why no way to increment array size except by creating new one?

A. Because Java doesn't let us

Because OS doesn't let us safely write past memory bounds

Because memory is fragmented at the hardware level

Program	See Applications section
Library	Data structures (ArrayStack)
Programming language	Basic data types (Array)
Operating system	Memory allocation
Hardware	Physical memory

API / Abstract Data Type / Interface (Stack)*

Memory fragmentation

*Note: in this course Stack is a class and not an interface

Stack: resizing-array implementation

"repeated doubling"

Q. How to grow array?

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

```
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;
}
```

Array accesses to add first $n = 2^i$ items. $n + (2 + 4 + 8 + \ldots + n) \sim 3n.$

1 array access per push

k array accesses to double to size k (ignoring cost to create new array)

Stack: resizing-array implementation

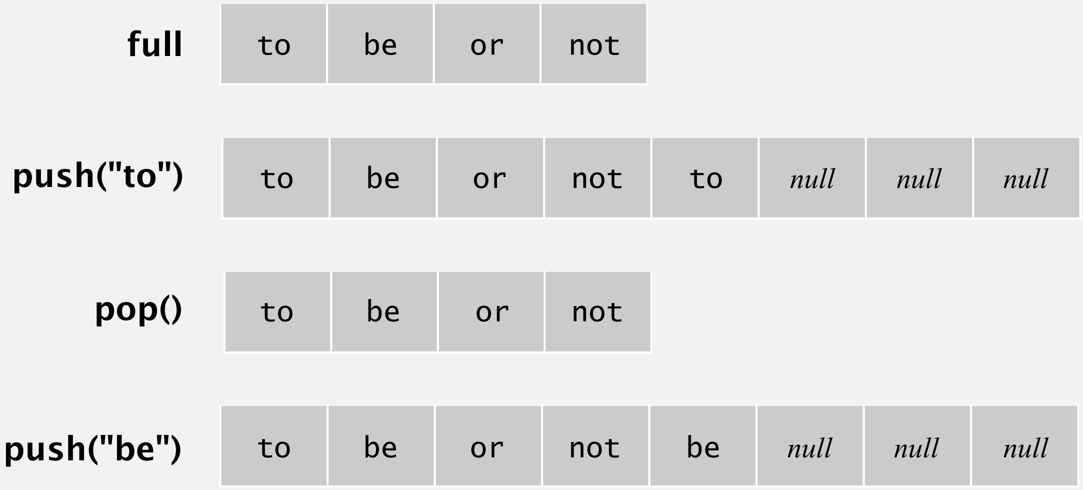
Q. How to shrink array?

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.



full	to	be	or	not				
push("to")	to	be	or	not	to	*null*	*null*	*null*
pop()	to	be	or	not				
push("be")	to	be	or	not	be	*null*	*null*	*null*

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: performance

Amortized analysis. Starting from an empty data structure, 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.

	typical	worst	amortized
construct	1	1	1
push	1	n	1
pop	1	n	1
size	1	1	1

doubling and halving operations

order of growth of running time for resizing array stack with n items

Stack resizing-array implementation: performance

Amortized analysis. Starting from an empty data structure, 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.

Proof.

- Divide the operations into batches.
- Each batch starts after a resize has just completed (or at the beginning) and ends with the next resize (or at the end).
- Claim: the cost of processing each batch is proportional to the number of operations in that batch.

Exercise: complete this argument.

Exercise: would proof hold if we'd had different constants instead of ½ and ¼ (say ⅔ & ⅓)? What are the factors to consider in picking the constants?

Stack resizing-array implementation: memory usage

Proposition. A `ResizingArrayStackOfStrings` uses $\sim 8n$ to $\sim 32n$ bytes of memory for 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 × array size

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

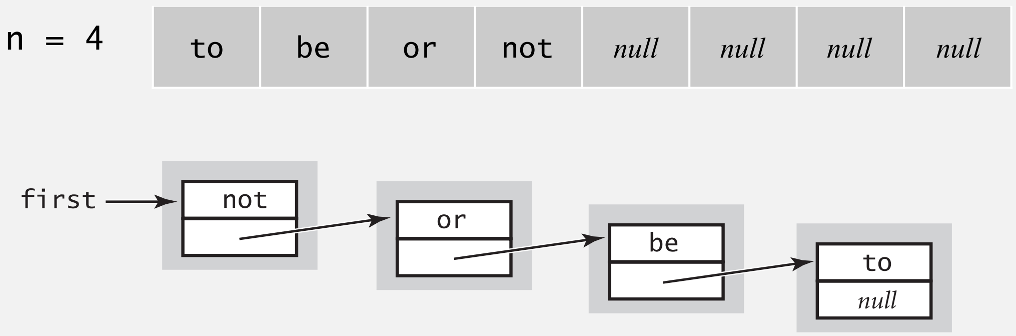
Tradeoffs. Can implement a stack with either resizing array or linked list; client can use interchangeably. Which one is better?

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.



Algorithms

Robert Sedgewick | Kevin Wayne

http://algs4.cs.princeton.edu

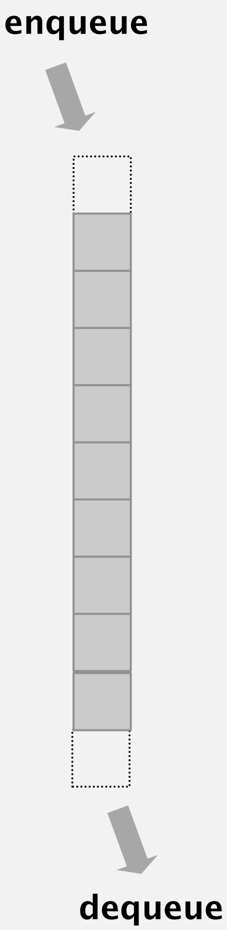
1.3 Bags, Queues, and Stacks

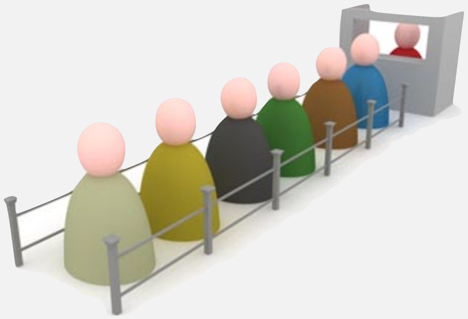
- *stacks*
- *resizing arrays*
- ***queues***
- *generics*
- *iterators*
- *applications*

Queue API

`public class QueueOfStrings`

	`QueueOfStrings()`	*create an empty queue*
`void`	`enqueue(String item)`	*add a new string to 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*

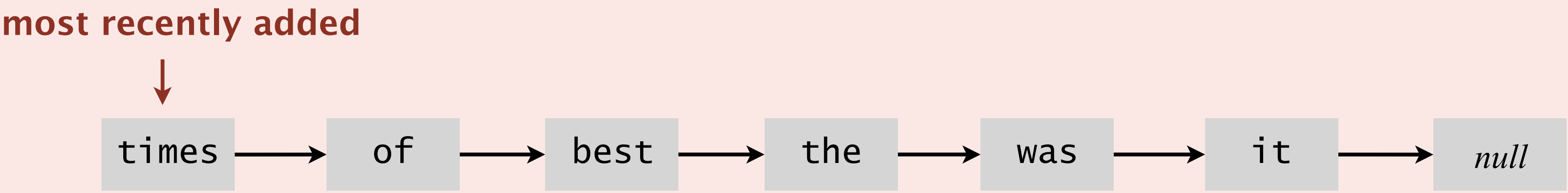




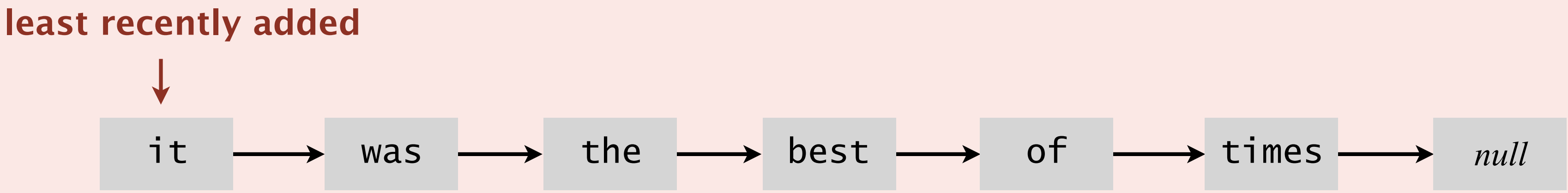
Queues quiz 1

How to implement a queue with a singly-linked linked list?

A.



B.

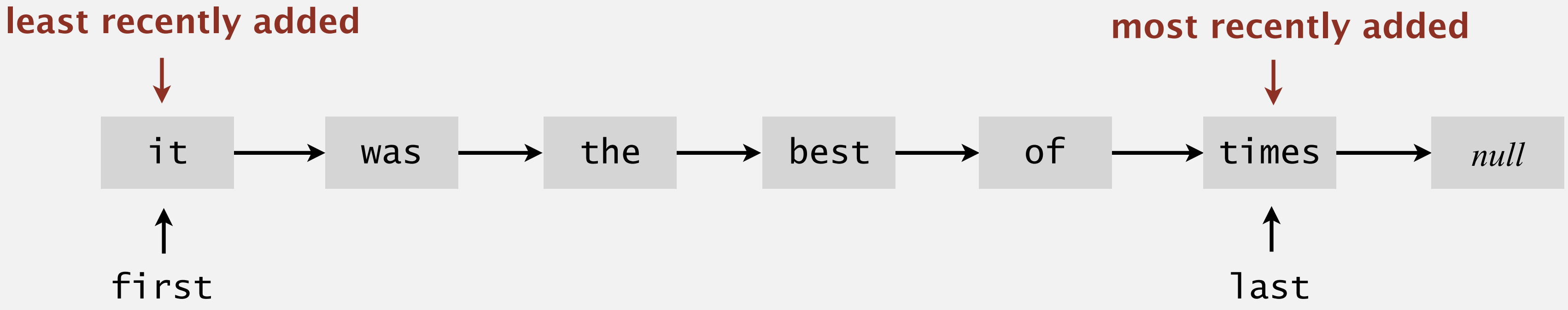


C. *None of the above.*

D. *I don't know.*

Queue: linked-list implementation

- Maintain one pointer `first` to first node in a singly-linked list.
- Maintain another pointer `last` to last node.
- Dequeue from `first`.
- Enqueue after `last`.



Queue dequeue: linked-list implementation

inner class

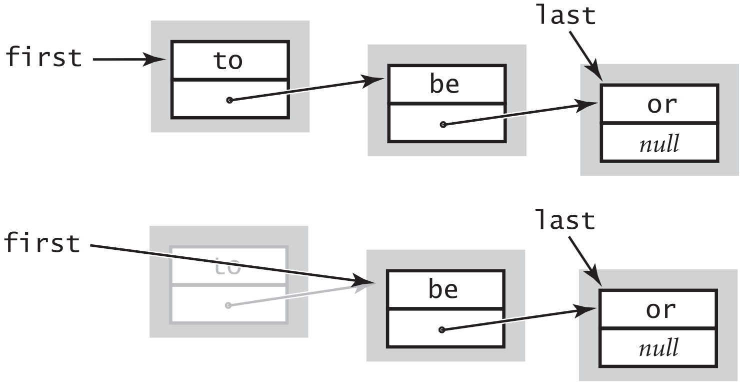
```
private class Node
{
   String item;
   Node next;
}
```

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()`.

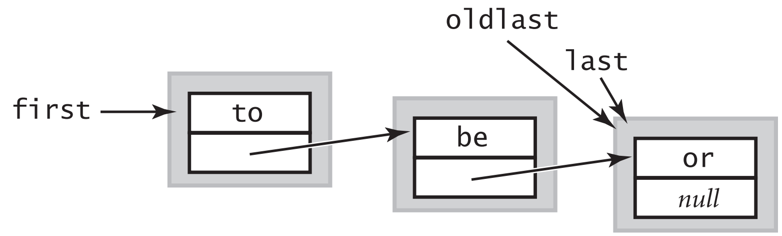
Queue enqueue: linked-list implementation

inner class

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

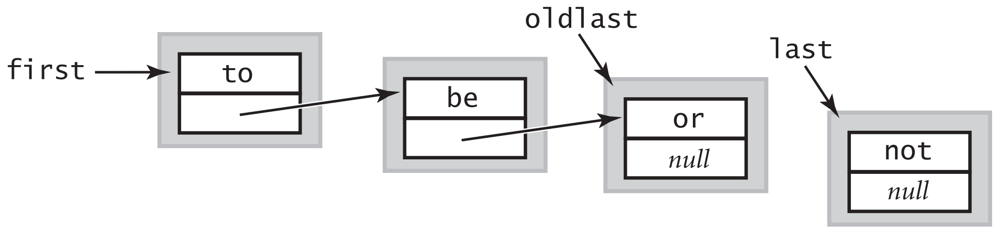
save a link to the last node

```
Node oldlast = last;
```



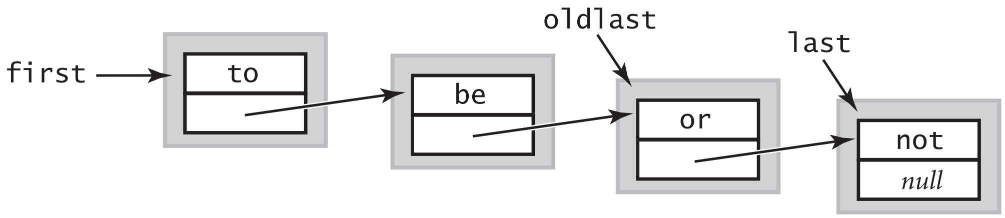
create a new node for the end

```
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

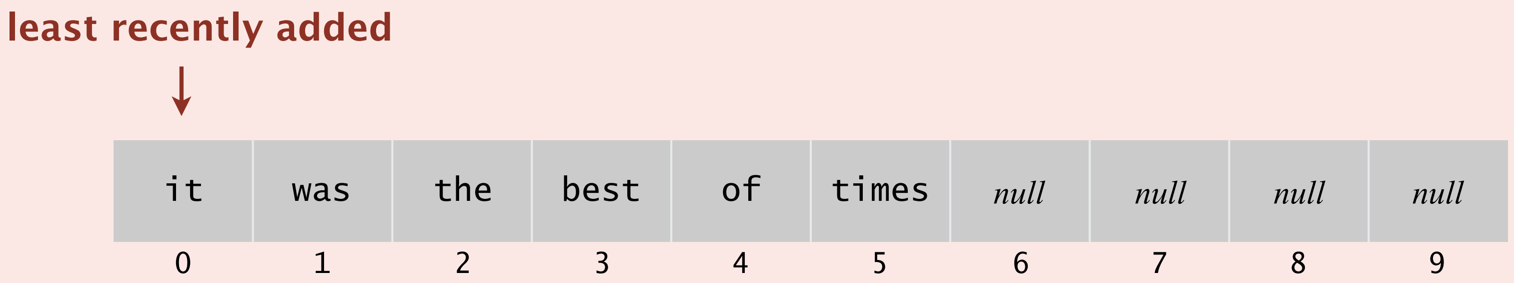
```
public class LinkedQueueOfStrings
{
   private Node first, last;

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

   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

Queues quiz 2

How to implement a fixed-capacity queue with an array?

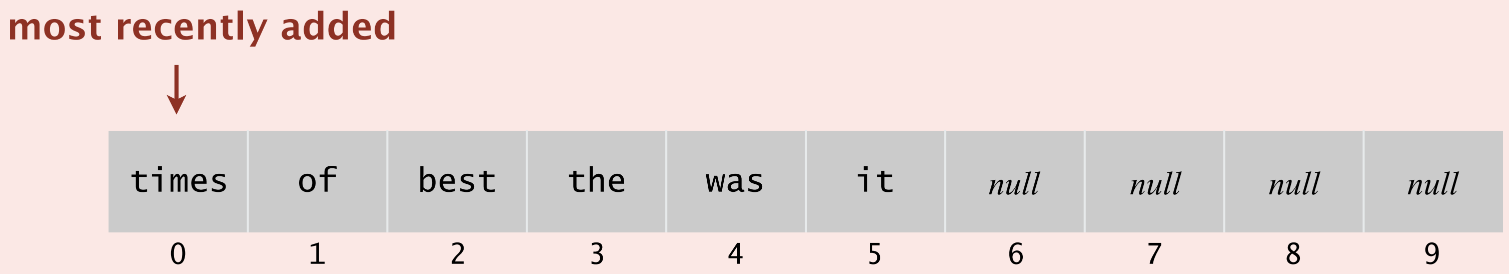
A.



least recently added

it	was	the	best	of	times	*null*	*null*	*null*	*null*
0	1	2	3	4	5	6	7	8	9

B.



most recently added

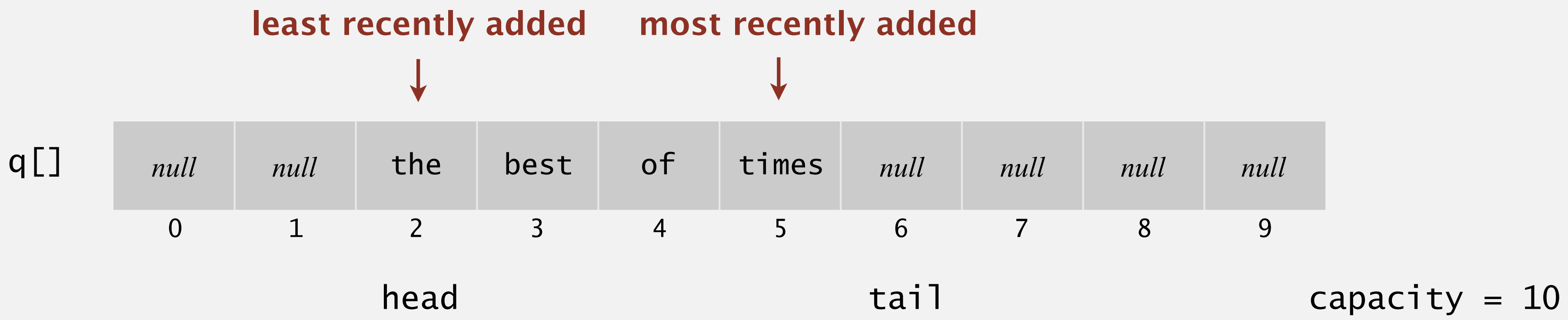
times	of	best	the	was	it	*null*	*null*	*null*	*null*
0	1	2	3	4	5	6	7	8	9

C. *None of the above.*

D. *I don't know.*

Queue: resizing-array implementation

- 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.



least recently added

most recently added

q[]	0	1	2	3	4	5	6	7	8	9
	null	*null*	the	best	of	times	*null*	*null*	*null*	*null*

head

tail

capacity = 10

Q. How to resize?

Queue with two stacks

Job interview problem. Implement a queue with two stacks so that:

- Each queue op uses a constant amortized number of stack ops.
- At most constant extra memory (besides the two stacks).

Solution. Call the two stacks `incoming` and `outgoing`.

- enqueue: push to `incoming`
- dequeue: pop from `outgoing`
  - if outgoing is empty, first "pour" `incoming` into `outgoing` (O(N)).
- `isEmpty`: check if both stacks are empty

Analysis: correctness. Left as exercise.

Analysis: efficiency. Consider the lifecycle of each item:

pushed into `incoming`, popped from `incoming`,

pushed into `outgoing`, popped from `outgoing`.

At most 4 stack operations per item. [Exercise: complete this argument.]

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`.

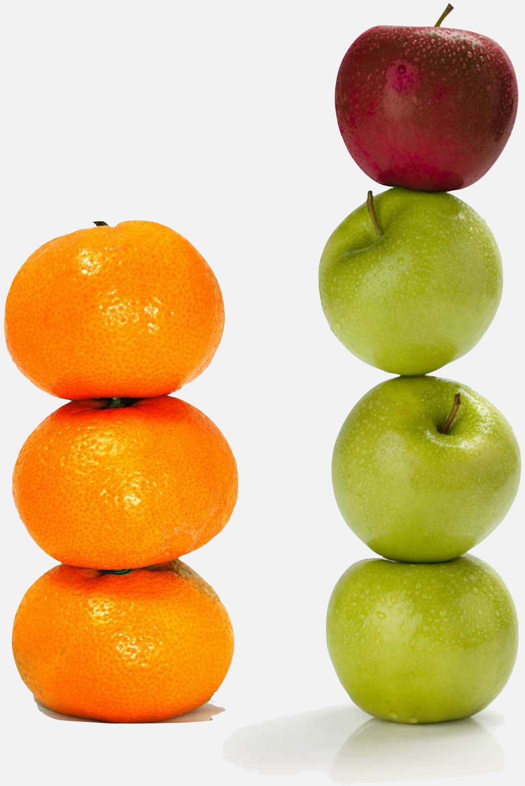
We also want: `StackOfURLs`, `StackOfInts`, `StackOfApples`, `StackOfOranges`, ...

Solution in Java: generics.

type parameter
(use both to specify type and to call constructor)

```
Stack<Apple> stack = new Stack<Apple>();
Apple apple = new Apple();
Orange orange = new Orange();
stack.push(apple);
stack.push(orange);
...
```

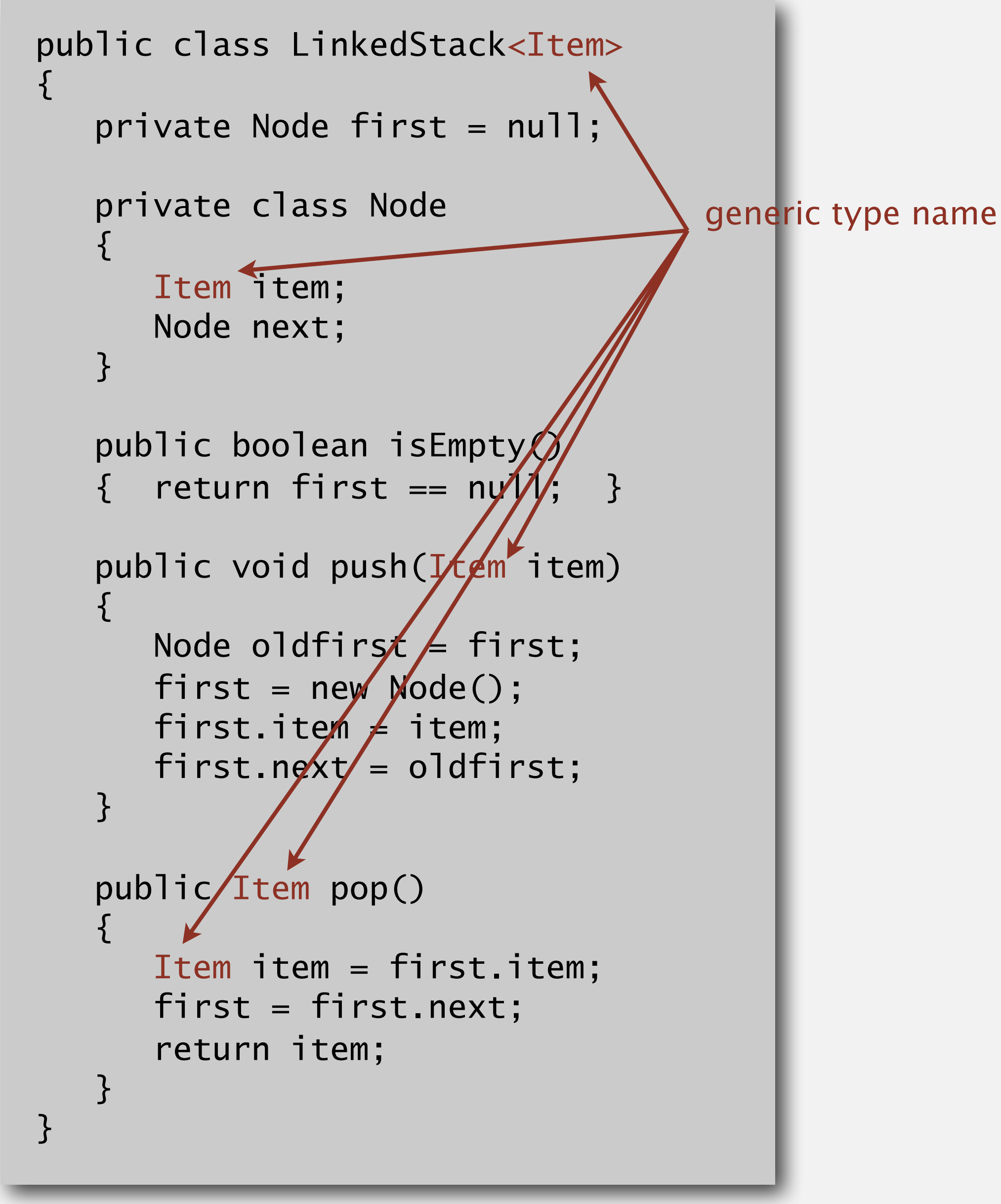
compile-time error



Guiding principle. 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 LinkedStack<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 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];  }
}
```

ugly cast
(will result in a compile-time warning, but it's not your fault)

Generic data types: autoboxing and unboxing

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 from primitive type to wrapper type.

Unboxing. Automatic cast from wrapper type to primitive type.

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

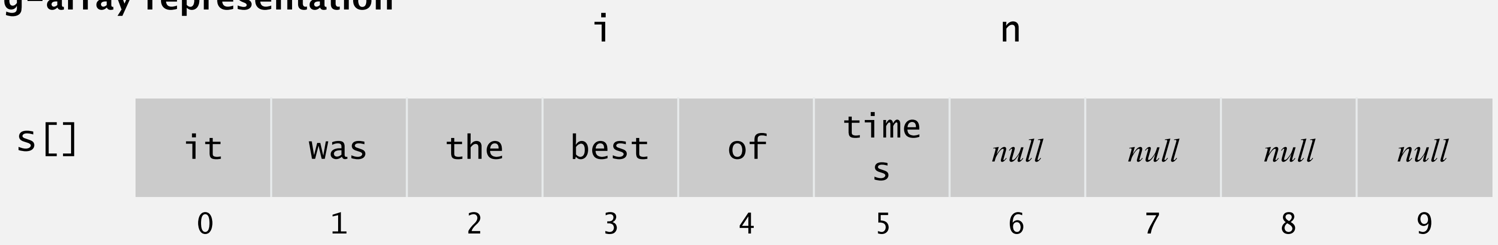
1.3 Bags, Queues, and Stacks

- stacks
- resizing arrays
- queues
- generics
- *iterators* ← **see precept**
- applications

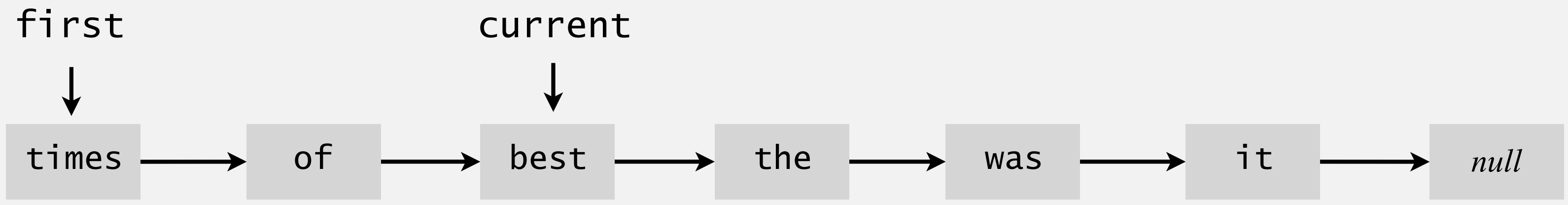
Iteration

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

resizing-array representation



linked-list representation



Java solution. Use a for-each loop.

For-each loop

Java provides elegant syntax for iteration over collections.

"for-each" loop (shorthand)

```
Stack<String> stack;
...

for (String s : stack)
    ...
```

equivalent code (longhand)

```
Stack<String> stack;
...

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

For a user-defined collection, do this to enable looping over it with for-each:

- Data type must have a method named `iterator()`.
- The `iterator()` method returns an object that has two core methods.
  - the `hasNext()` methods returns `false` when there are no more items
  - the `next()` method returns the next item in the collection

Iterators

To support for-each loops, Java provides two interfaces.

- `Iterator` interface: `next()` and `hasNext()` methods.
- `Iterable` interface: `iterator()` method that returns an `Iterator`.
- Both should be used with generics.

java.util.Iterator interface

```
public interface Iterator<Item>
{
   boolean hasNext();
   Item next();
   void remove();
}
```

optional; use at your own risk

java.lang.Iterable interface

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

Type safety.

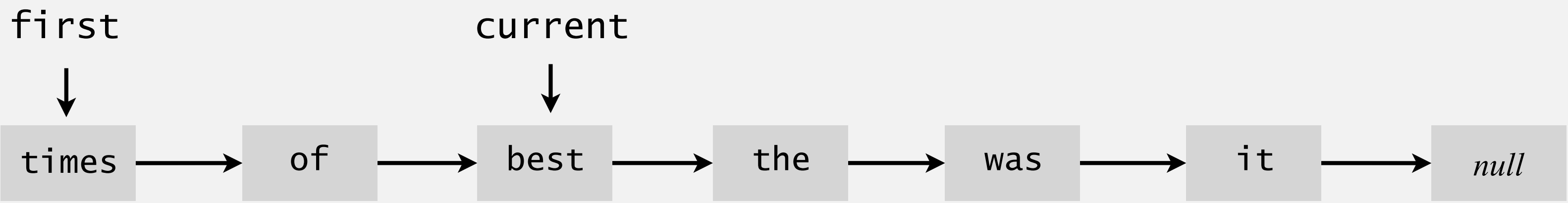
- Data type must use these interfaces to support for-each loop.
- Client program won't compile if implementation doesn't.

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;
        }
    }
}
```

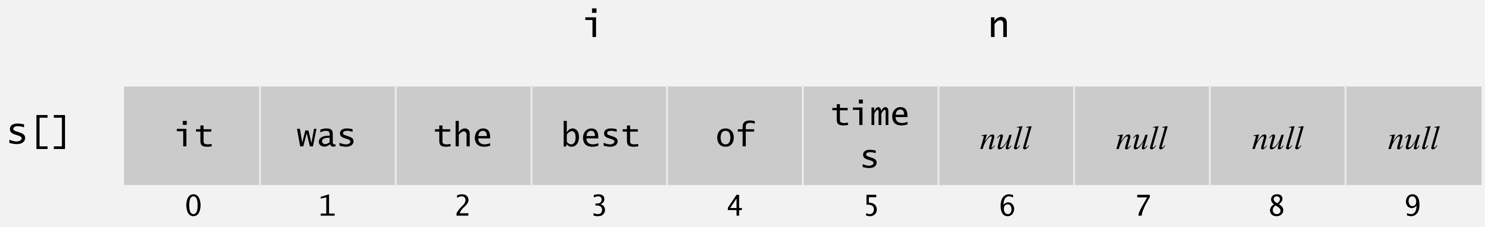
throw UnsupportedOperationException

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];          }
    }
}
```



				i			n			
s[]	it	was	the	best	of	times	*null*	*null*	*null*	*null*
	0	1	2	3	4	5	6	7	8	9

Iteration: concurrent modification

Q. What if client modifies the data structure while iterating?

A. A fail-fast iterator throws a `java.util.ConcurrentModificationException`.

concurrent modification

```
for (String s : stack)
   stack.push(s);
```

Q. How to detect concurrent modification?

A.

- Count total number of `push()` and `pop()` operations in `Stack`.
- Save counts in `*Iterator` subclass upon creation.
- If, when calling either `next()` or `hasNext()`, the current counts do not equal the saved counts, throw exception.

http://algs4.cs.princeton.edu

1.3 Bags, Queues, and Stacks

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

PSA

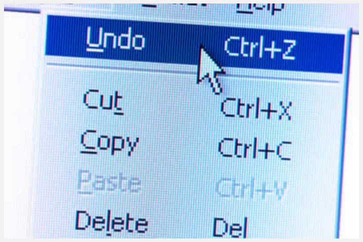
For this course, use our `Stack` and `Queue` implementations instead of Java’s.

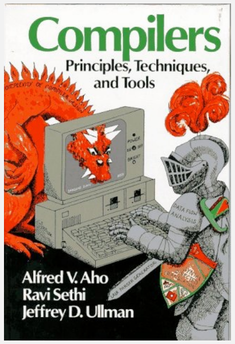
Stack applications

- 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.
- ...

Java™

Adobe® PostScript® 3™



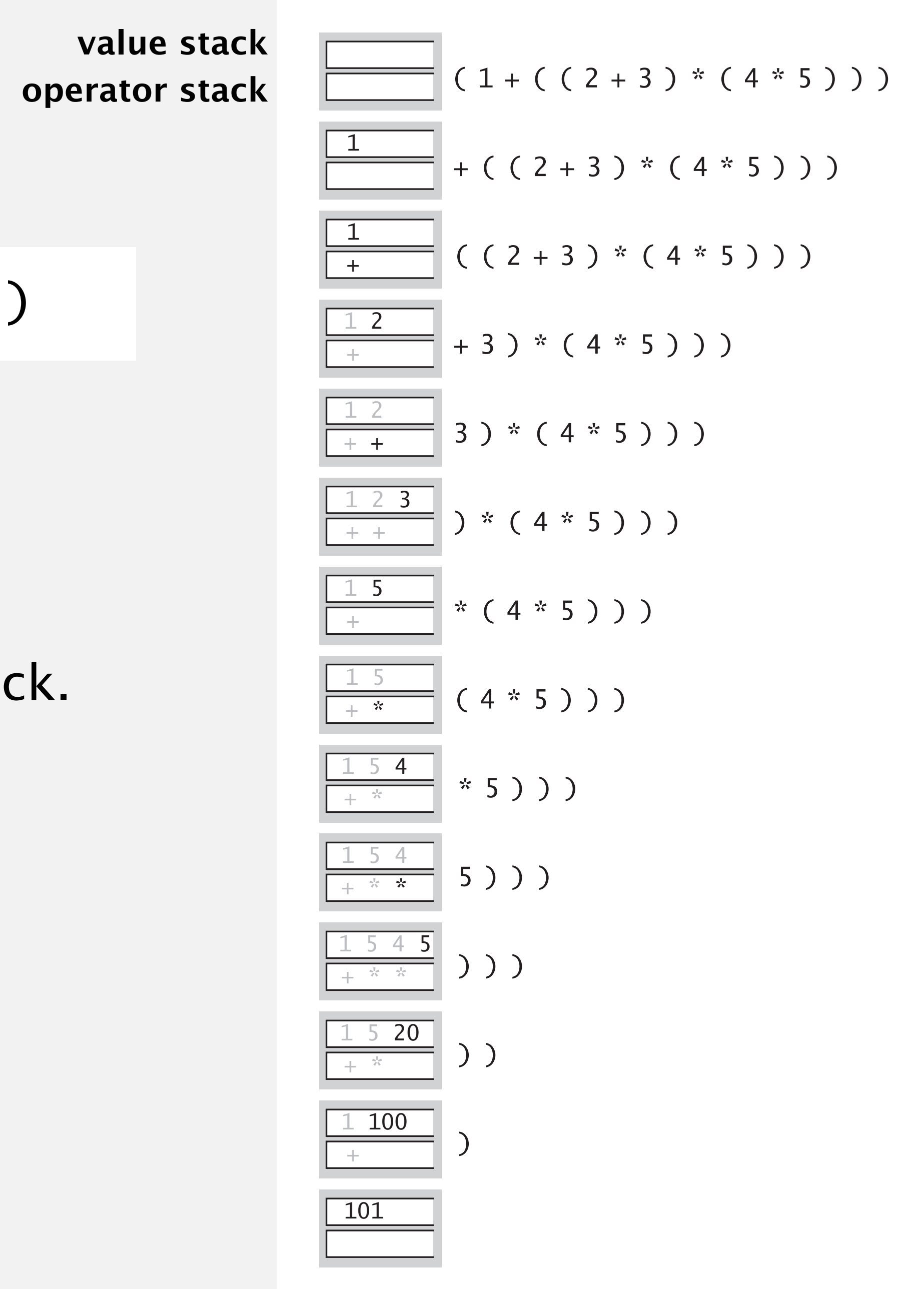

Compilers
Principles, Techniques, and Tools

Alfred V. Aho
Ravi Sethi
Jeffrey D. Ullman

Arithmetic expression evaluation

Goal. Evaluate infix expressions using two stacks.

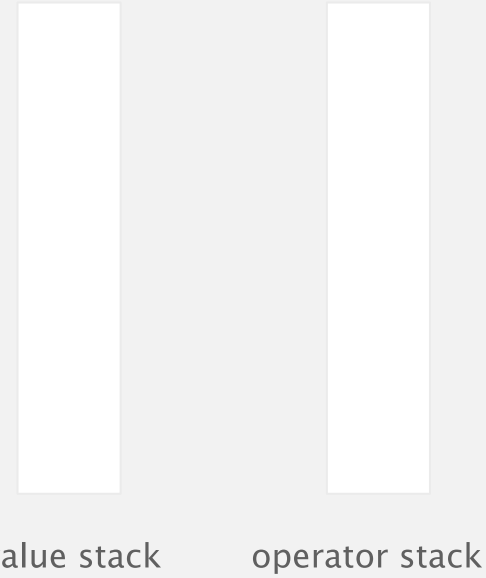
(1 + ((2 + 3) * (4 * 5)))



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
  - apply operator to the two values
  - push the result to value stack.

Dijkstra's two-stack algorithm demo



infix expression

(fully parenthesized)

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

operand

operator

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("("))    /* noop */ ;
         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 * * +
```

Jan Lukasiewicz

Bottom line. Postfix or "reverse Polish" notation.

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