ITI 1121. Introduction to Computing II *

Marcel Turcotte
School of Electrical Engineering and Computer Science

Version of February 2, 2013

Abstract

- Abstract Data Type
  - Stack

*These lecture notes are meant to be looked at on a computer screen. Do not print them unless it is necessary.

Stacks

Software stacks are abstract data types (structures) similar to physical stacks.

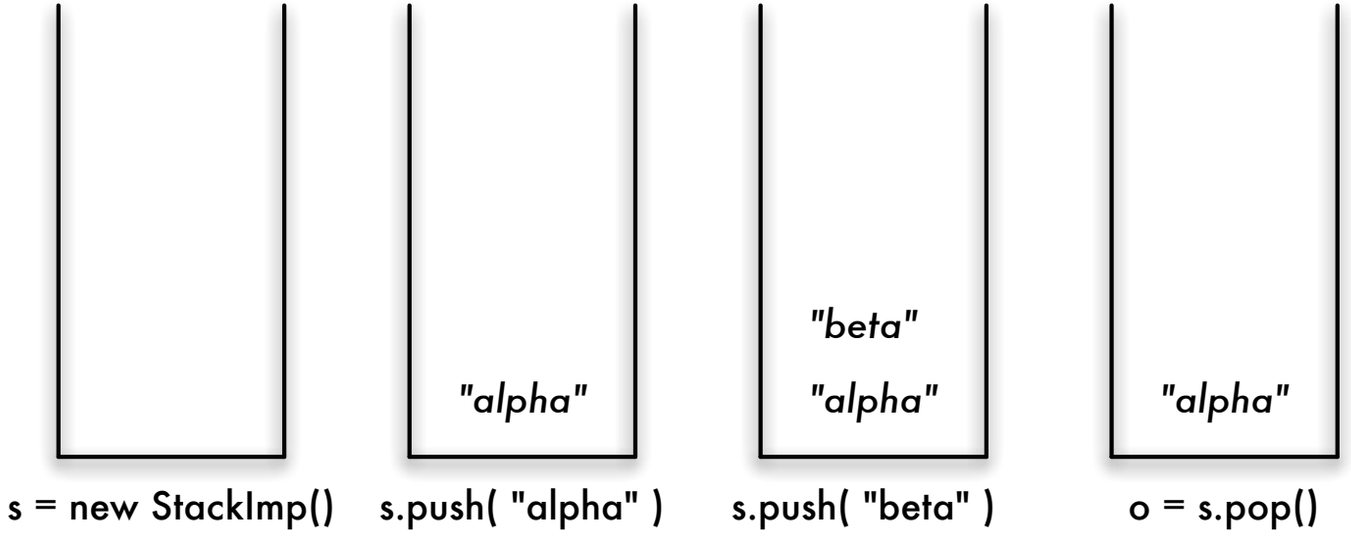
- Books
- PEZ dispenser
- Plates
- Trays

The analogy with a plate dispenser in a cafeteria is particularly interesting: 1) **for software stacks, just like physical stacks, only the top element is accessible** and 2) **the top element must be removed in order to access the remaining elements**.

Definition

A **stack** is a *linear* data structure that is always accessed from the same extremity, one element at a time, and that element is called the **top** of the stack.

Stacks are also called LIFO data structures: *last-in first-out*.



Applications

Stacks are widely used in applications and system programming:

- In compilers, and formal language analysis in general;
- To implement *backtracking* algorithms, which are used in automatic theorem provers (Prolog), games algorithms and artificial intelligence algorithms;
- For memory management during program execution, system stack, it supports the development of recursive algorithms;
- To support "undo" operations or "back" buttons inside a web browser.

Basic operations

The basic operations of a stack are:

push: add an element onto the top of the stack;

pop: removes and returns the top element;

empty: tests if the stack is empty.

Stack ADT

```
public interface Stack {
    public abstract boolean isEmpty();
    public abstract Object push( Object o );
    public abstract Object pop();
    public abstract Object peek();
}
```

Stack ADT (using Java 1.5)

```
public interface Stack<E> {
    public abstract boolean isEmpty();
    public abstract E push( E elem );
    public abstract E pop();
    public abstract E peek();
}
```

Yes, that's right, an interface can also be parametrized!

Example

```
public class Mystery {

    public static void main( String[] args ) {

        Stack<String> stack =
            new StackImplementation<String>();

        for ( int i=0; i<args.length(); i++ )
            stack.push( args[ i ] );

        while ( ! stack.empty() )
            System.out.print( stack.pop() );

    }
}
```

⇒ What does this print: "`java Mystery a b c d e`"?

Remarks

- Elements come out of a stack in reverse order;
- Frequently occurring idiom:

```
while ( ! stack.empty() ) {
    element = stack.pop();
    // ...
}
```

- Make sure to not forget the pop()!

Operations (contd)

peek: returns the object at the top of this stack without removing it;

Implementations

How would you implement this interface?

There are two popular families of implementation techniques:

- Array-based;
- Using linked-nodes.

```
Stack<Token> s;

s = new ArrayStack<Token>();
s = new DynamicArrayStack<Token>();
s = new LinkedStack<Token>();
```

Question

One of the proposed implementations will be using an array, why not using an array instead of stack in our programs since the implementation of the interface stack will be using an array anyway?

Implementing a Stack using an array: ArrayStack

What are the instance variables?

A reference to an array.

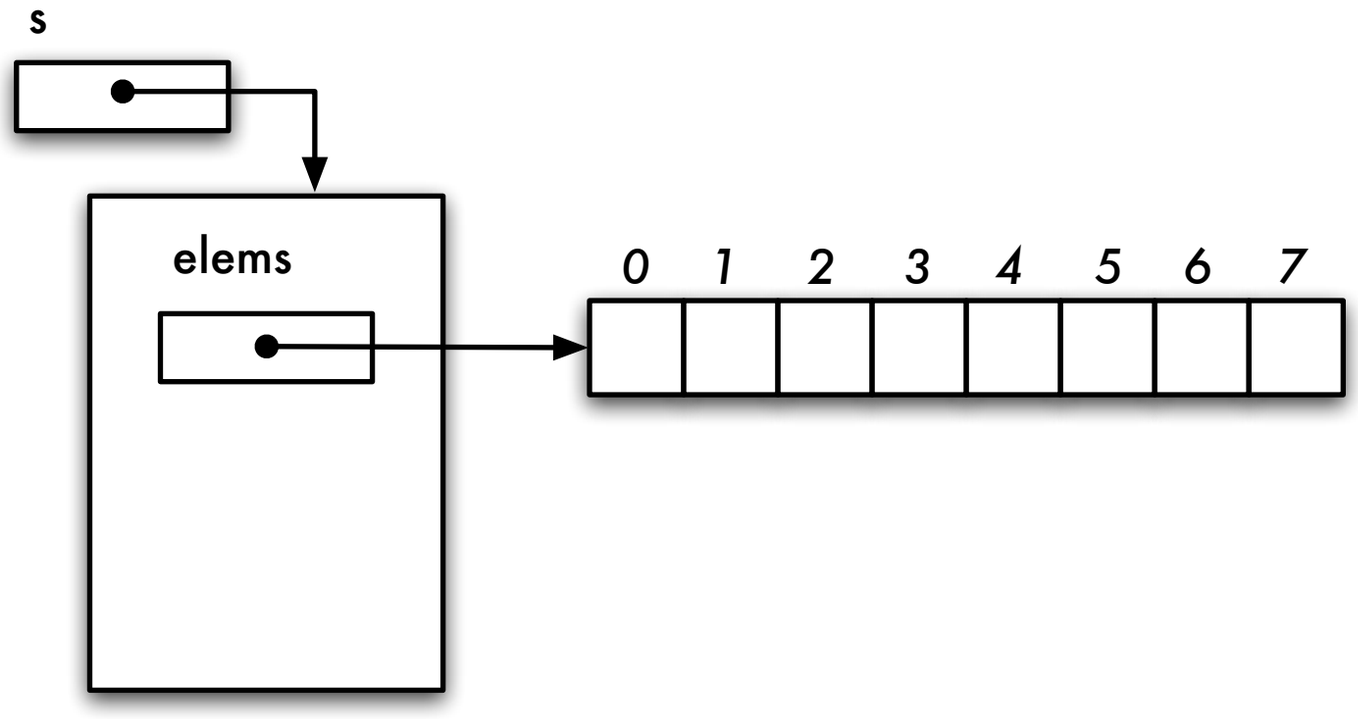
What will be the type of the references of this array?

Object!

Or some parameter if generics are used.

What will be your strategy for adding elements to the data structure?

Implementing a Stack using an array: ArrayStack



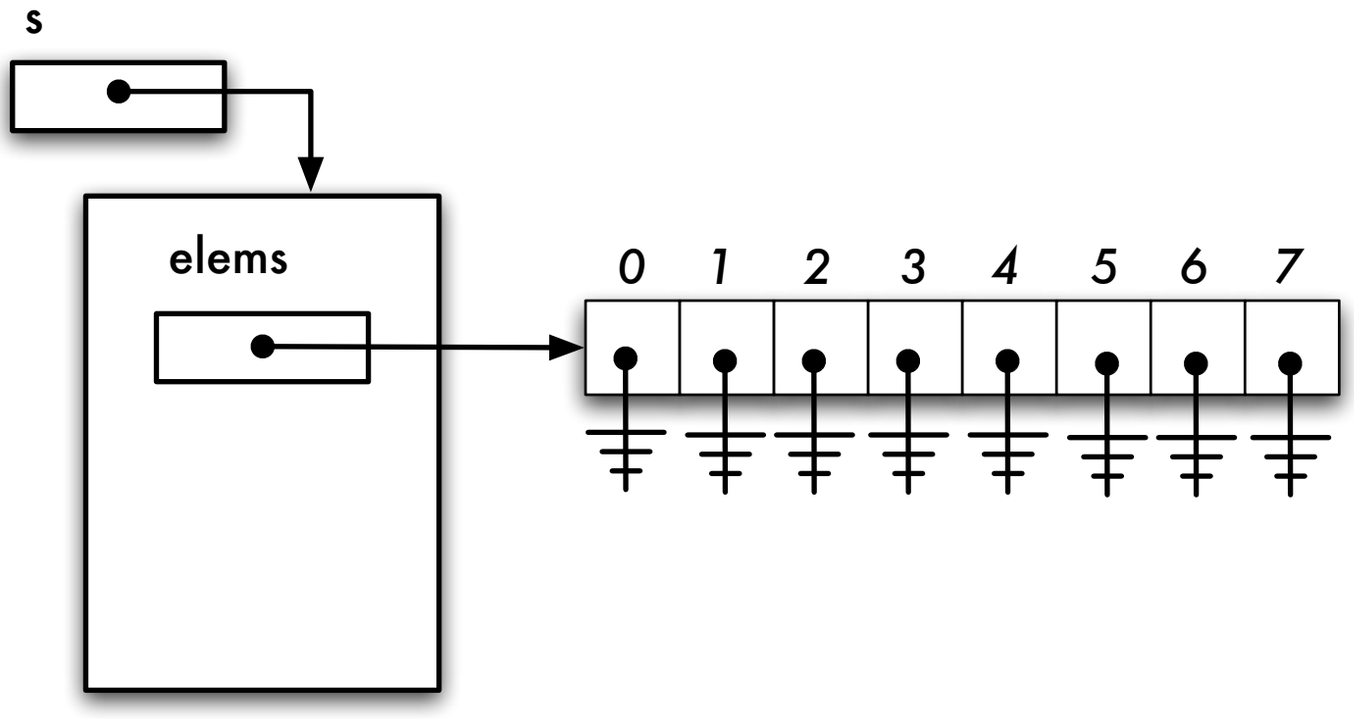
Implementing a Stack using an array: ArrayStack

Elements can be stored in the low part or high part of the array.

Let's select the low part of the array, the solution for the high part will be symmetrical.

How will the method push know where to insert the next element?

Implementing a Stack using an array: ArrayStack



Implementing a Stack using an array: ArrayStack

How will the method push know where to insert the next element?

A new instance variable is needed.

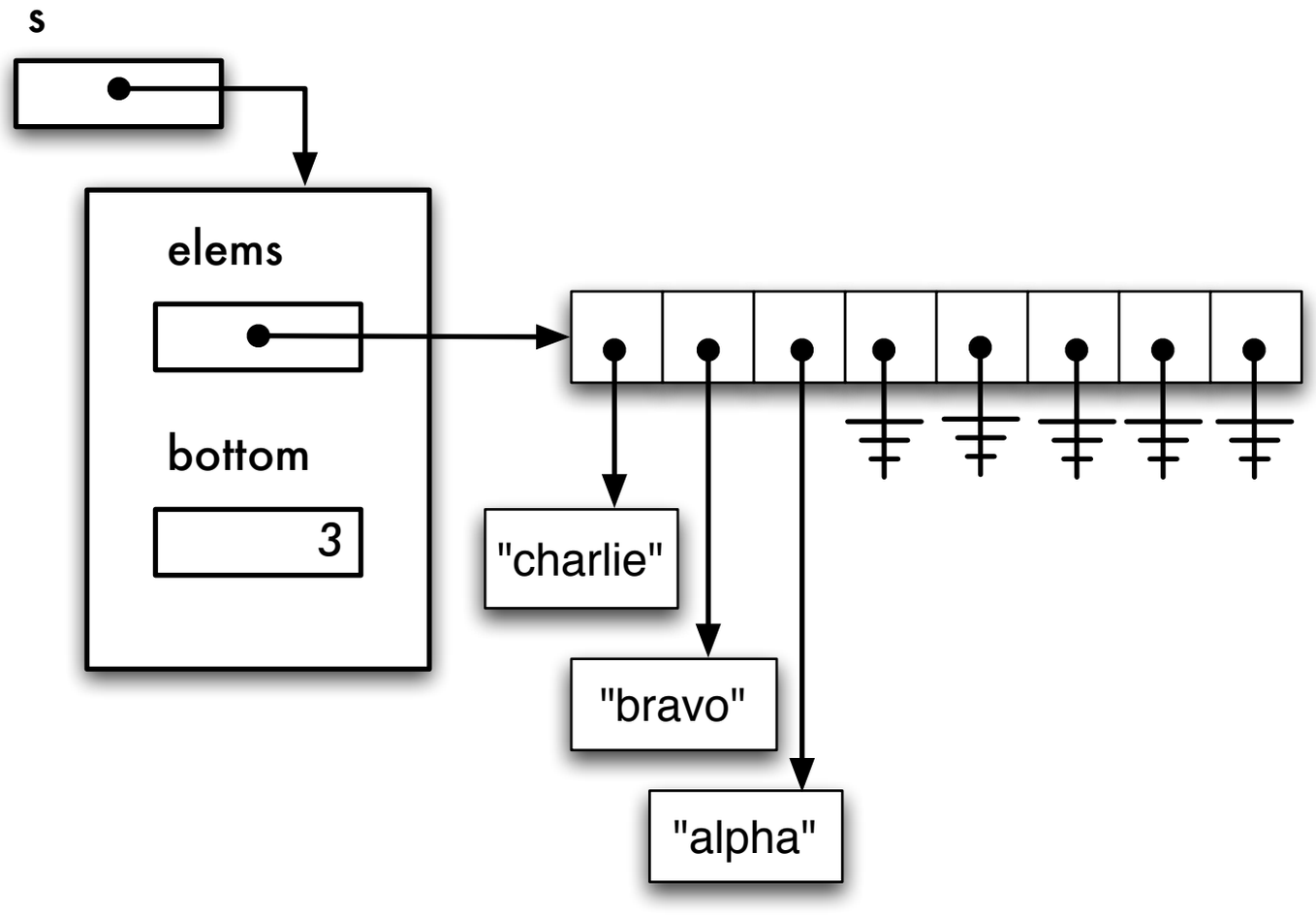
Is it top or bottom?

Is the top position fixed or is it the bottom?

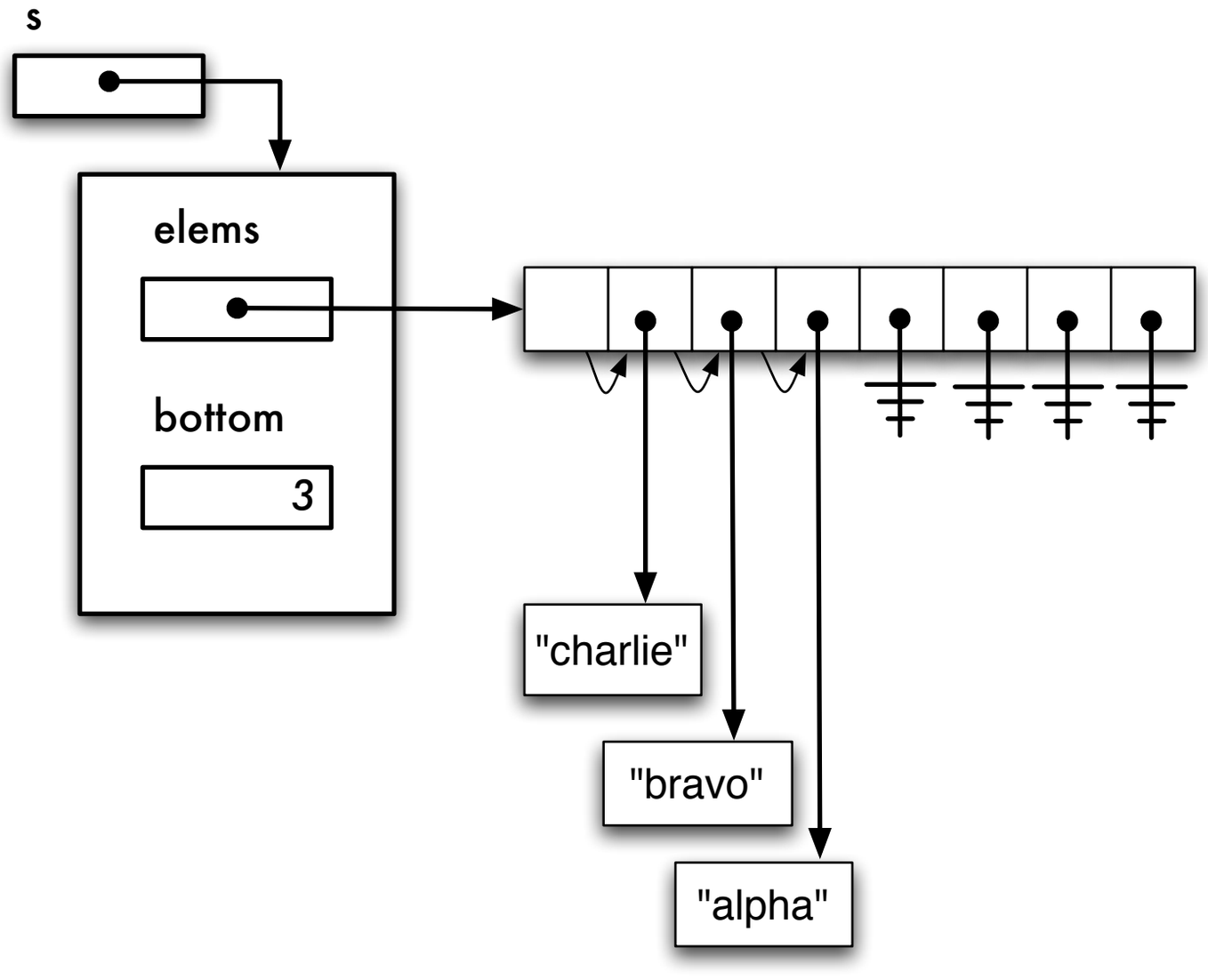
In which direction will the stack be growing?

Where will the first element, second element, third element, etc. be stored?

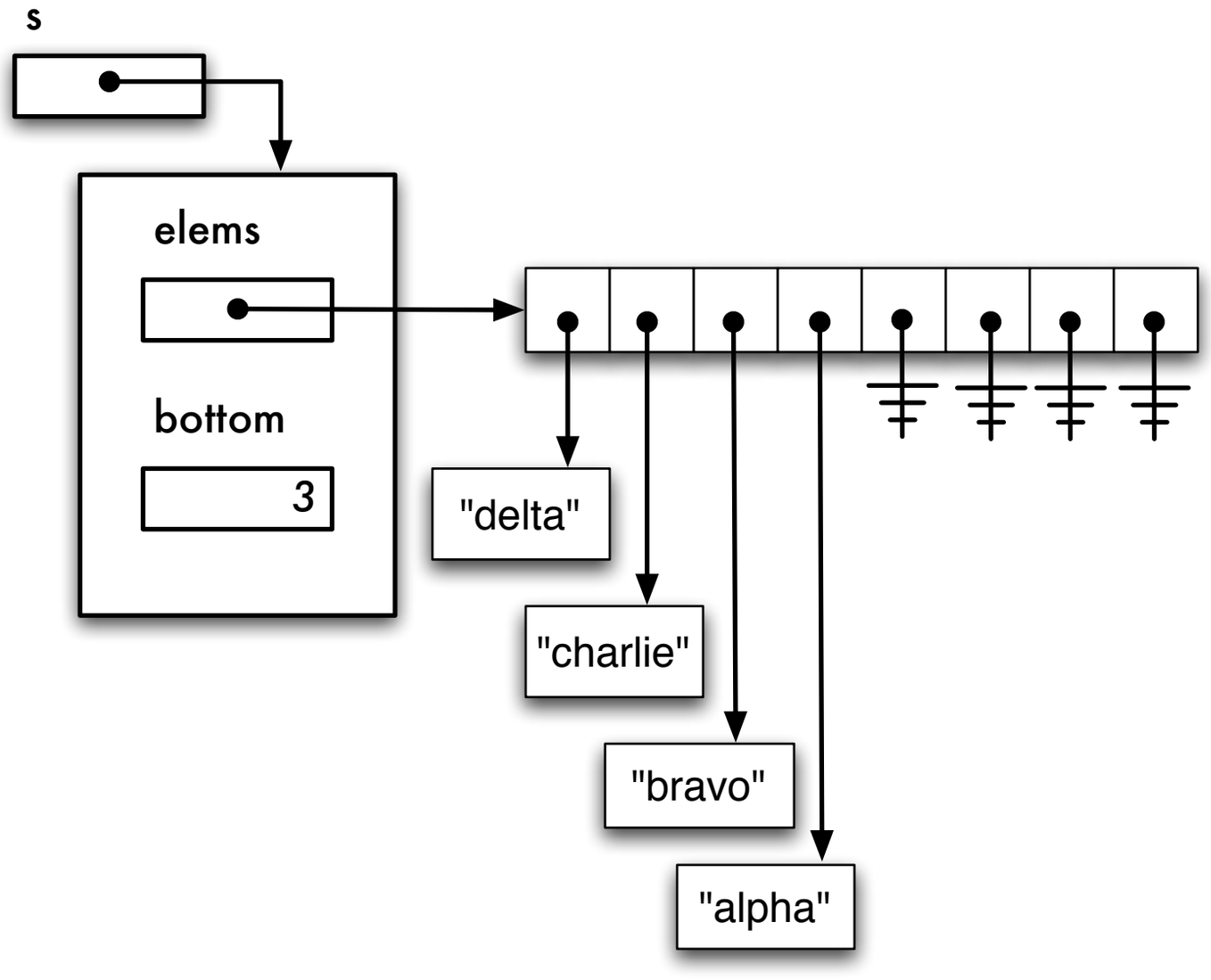
Implementing a Stack using an array: ArrayStack



Implementing a Stack using an array: ArrayStack



Implementing a Stack using an array: ArrayStack



Implementing a Stack using an array: ArrayStack

Comment the following implementation:

The elements of this stack are stored in the low part of the array, the top element is always located at position 0 (convention), an instance variable is used to indicate the position of the bottom element in the array.

For each element pushed onto the stack, all the elements must be moved one position up in the array, so that the top element is always at position 0. For each element removed, all the elements must be moved one position down in the array.

Implementing a Stack using an array: ArrayStack

Comment the following implementation:

The elements of this stack are stored in the low part of the array, the bottom element is always located at position 0 (convention), an instance variable is used to indicate the position of the top element in the array.

Each time an element is pushed onto the stack, the value of the index top is incremented by one, the new element is added at that position. Each time an element is removed from the stack, the reference of the top element is stored in a temporary variable, the position of the array designated by top is set to *null*, the index is decremented by one, the element saved is returned.

This is the preferred solution. It avoids copying the elements up or down for each insertion or removal. Copying elements would become progressively more expensive as the number of elements in the stack increases.

Implementing a Stack using an array: ArrayStack

Summary. this implementations requires a reference to an array, an array, as well as a top index.

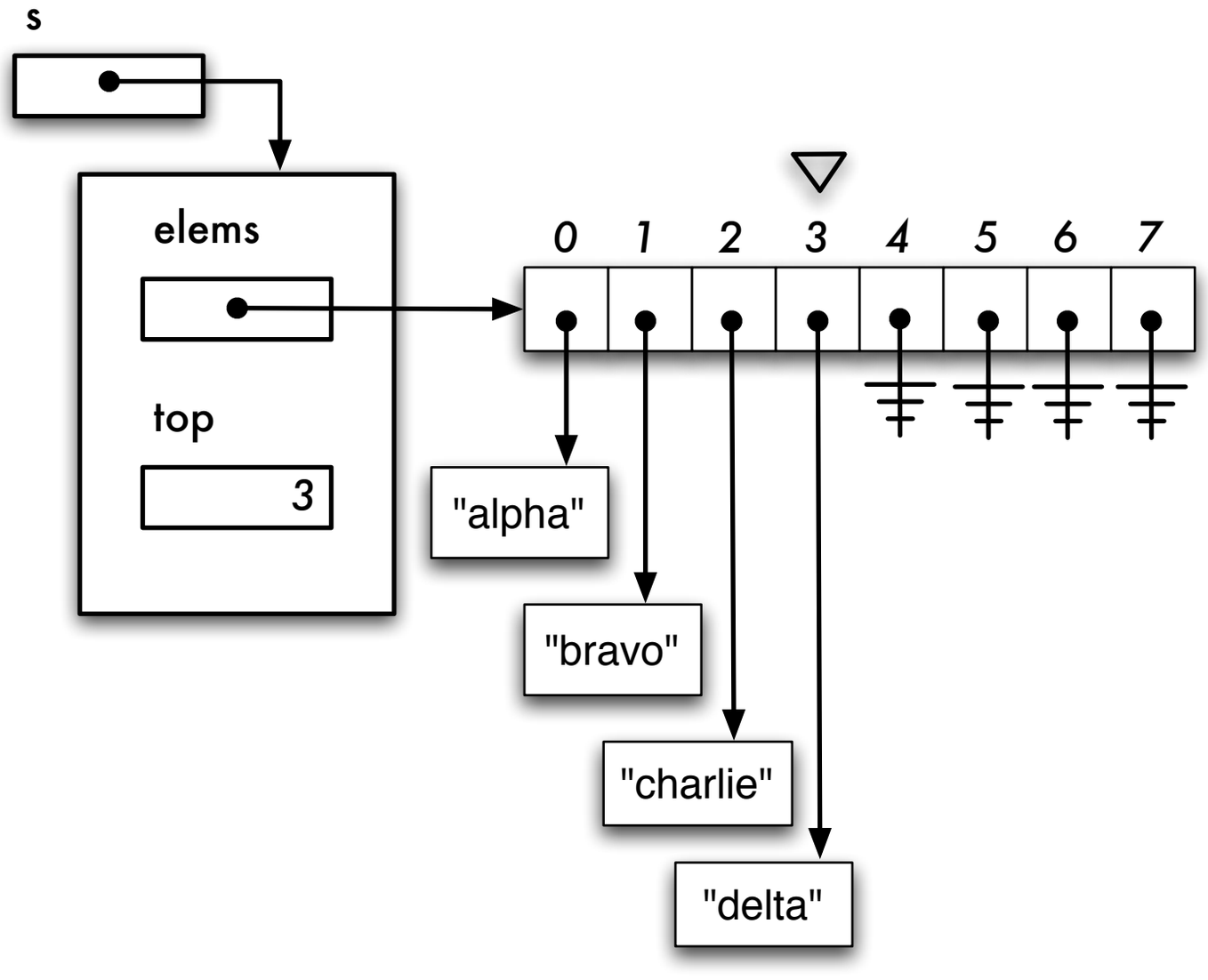
Which tasks will be carried out by the constructor?

What will be the initial value of the top index?

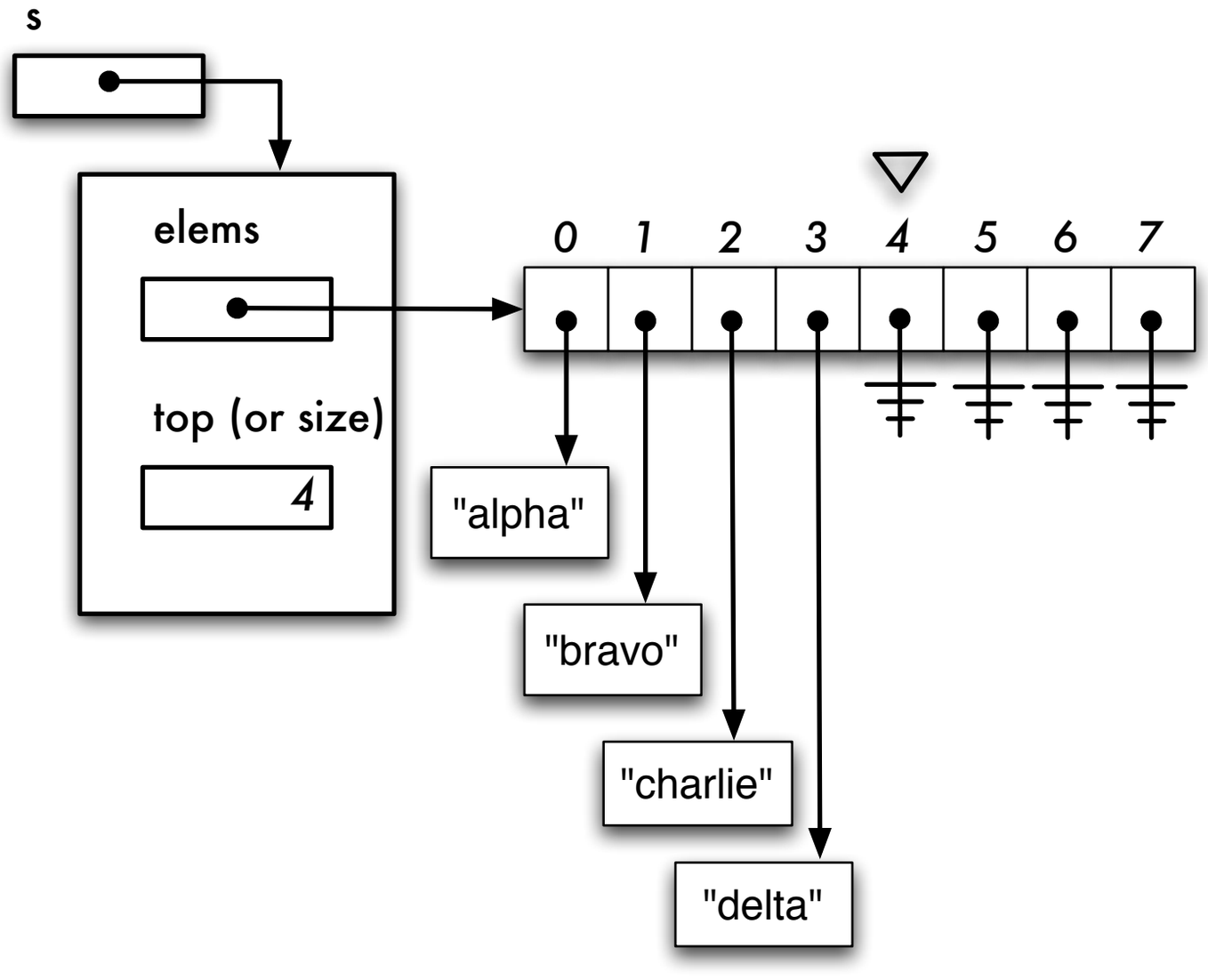
1. Let **top** designate the first empty cell of the array.

2. Let **top** designate the top element.

Implementing a Stack using an array: ArrayStack



Implementing a Stack using an array: ArrayStack



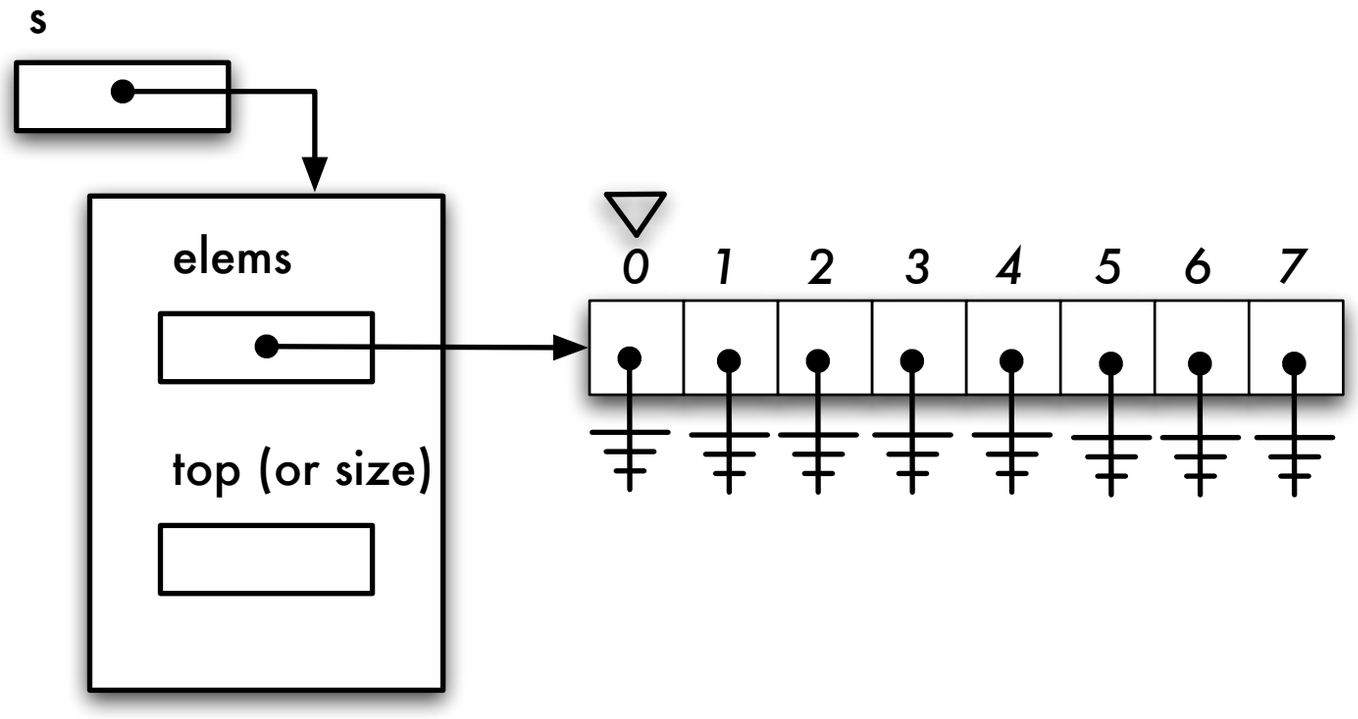
Implementing a Stack using an array: ArrayStack

1. Let **top** designate the first empty cell of the array. What should be the initial value of **top**?

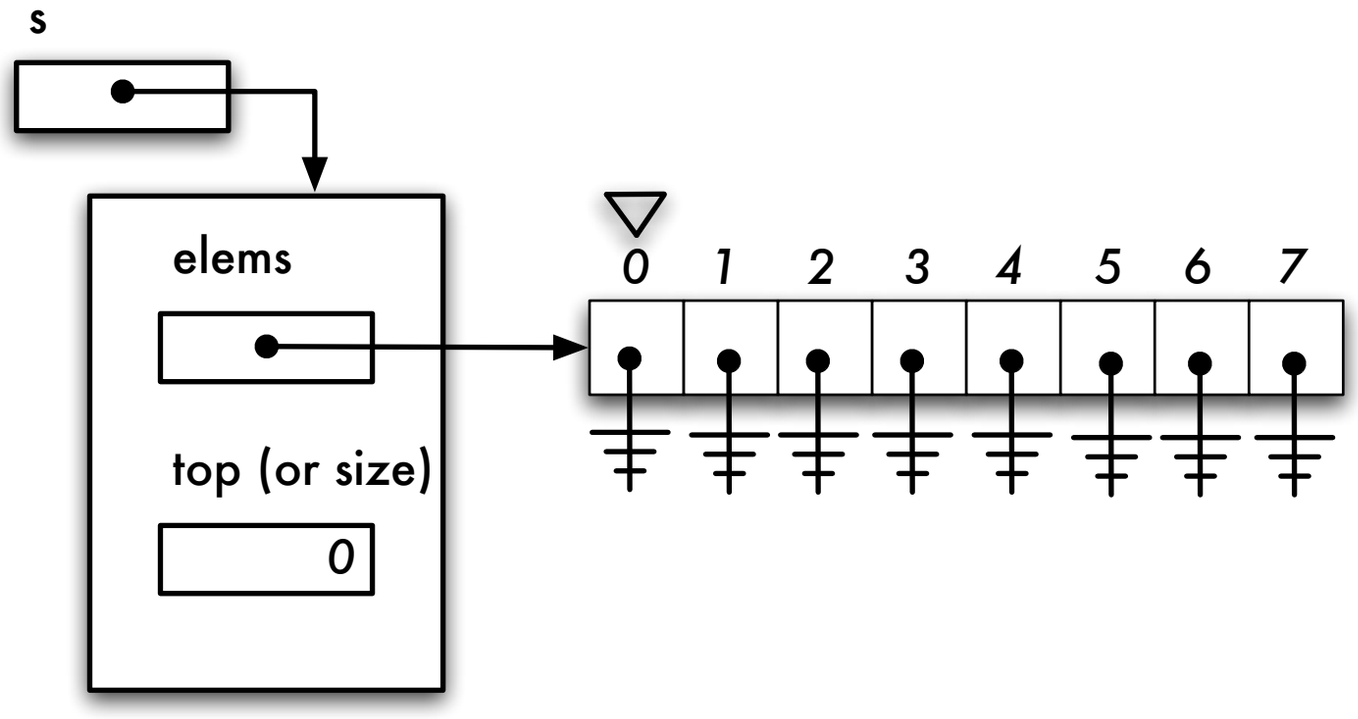
2. Let **top** designate the top element. What should be the initial value of the index **top**?

Make sure to fully understand both strategies.

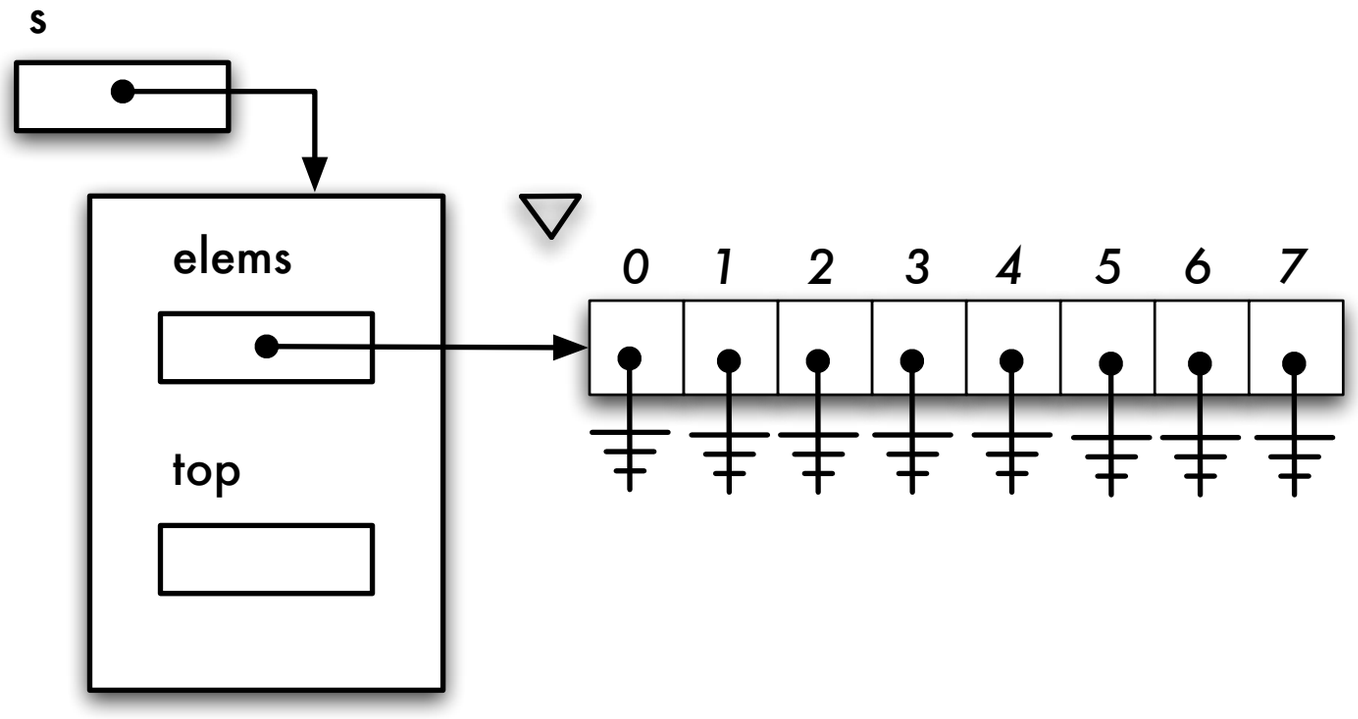
Implementing a Stack using an array: ArrayStack



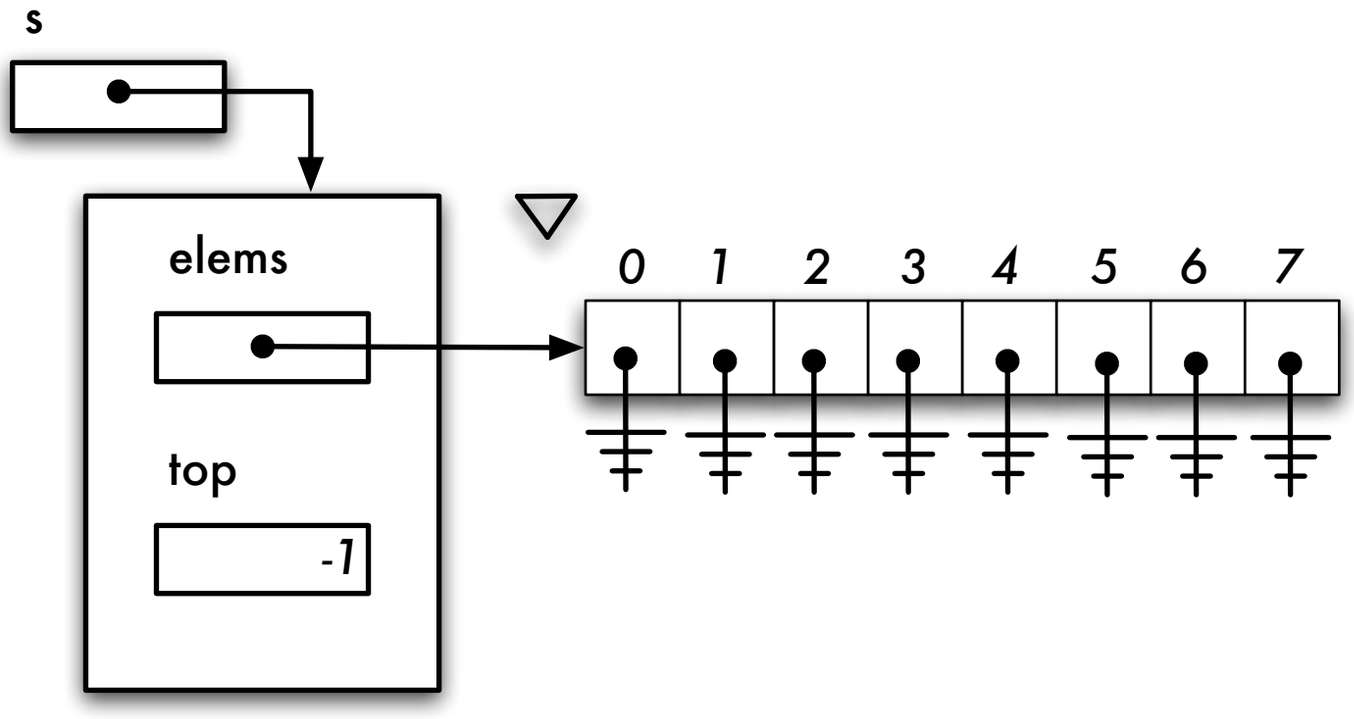
Implementing a Stack using an array: ArrayStack



Implementing a Stack using an array: ArrayStack



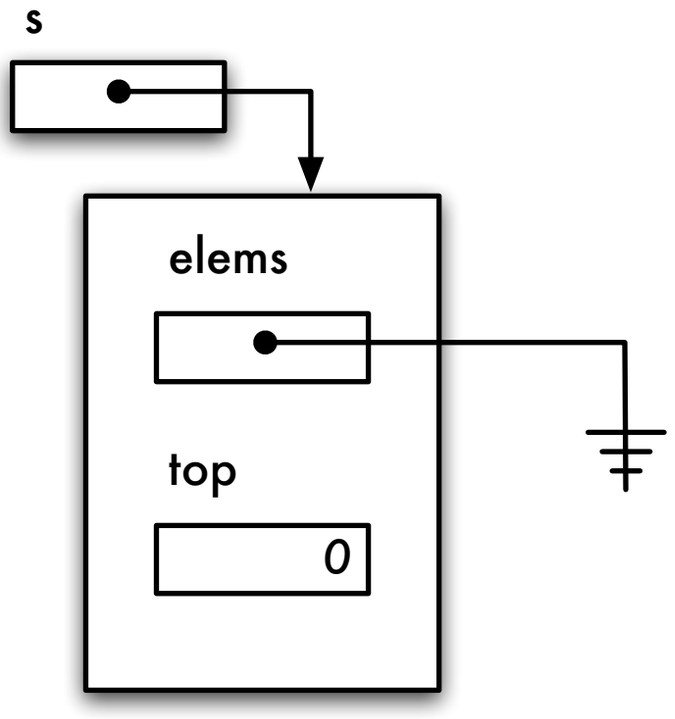
Implementing a Stack using an array: ArrayStack



Implementing a Stack using an array: ArrayStack

```
public class ArrayStack implements Stack {

    // Instance variables
    private Object[] elems;  // used to store the elements
    private int top;         // designates the first free cell!


    // Constructor

    public ArrayStack( int capacity ) {

    }
```

Implementing a Stack using an array: ArrayStack



Implementing a Stack using an array: ArrayStack

```
public class ArrayStack implements Stack {

    // Instance variables
    private Object[] elems;  // used to store the elements of this ArrayStack
    private int top;         // designates the first free cell!


    // Constructor

    public ArrayStack( int capacity ) {

        elems = new Object[ capacity ];
        top = 0;
    }

    // Returns true if this ArrayStack is empty
    public boolean isEmpty() {
        return top == 0;
    }
```

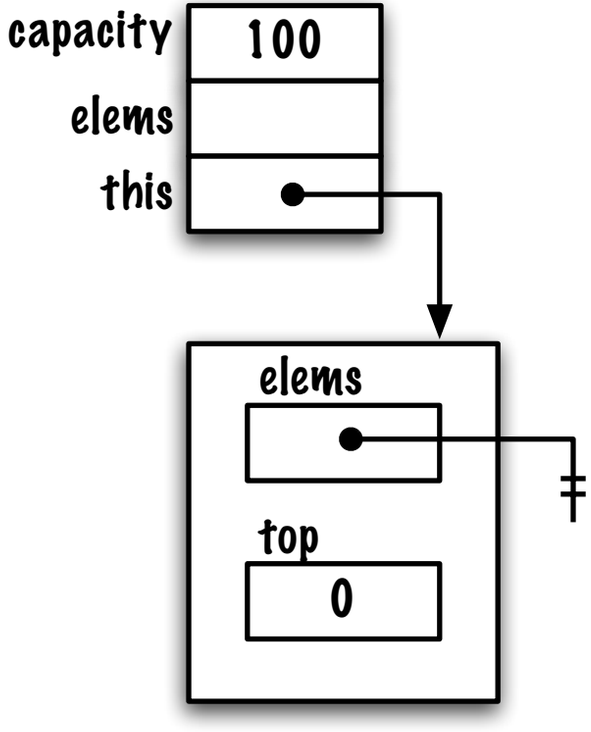
Pitfall?!

```
public class ArrayStack implements Stack {

    // Instance variables
    private Object[] elems;
    private int top;

    // Constructor
    public ArrayStack( int capacity ) {
        Object[] elems = new Object[ capacity ];
        top = 0;
    }
    // Returns true if this ArrayStack is empty
    public boolean isEmpty() {
        return top == 0;
    }
```

Activation Frame for ArrayStack

formal parameter(s)

local variable(s)

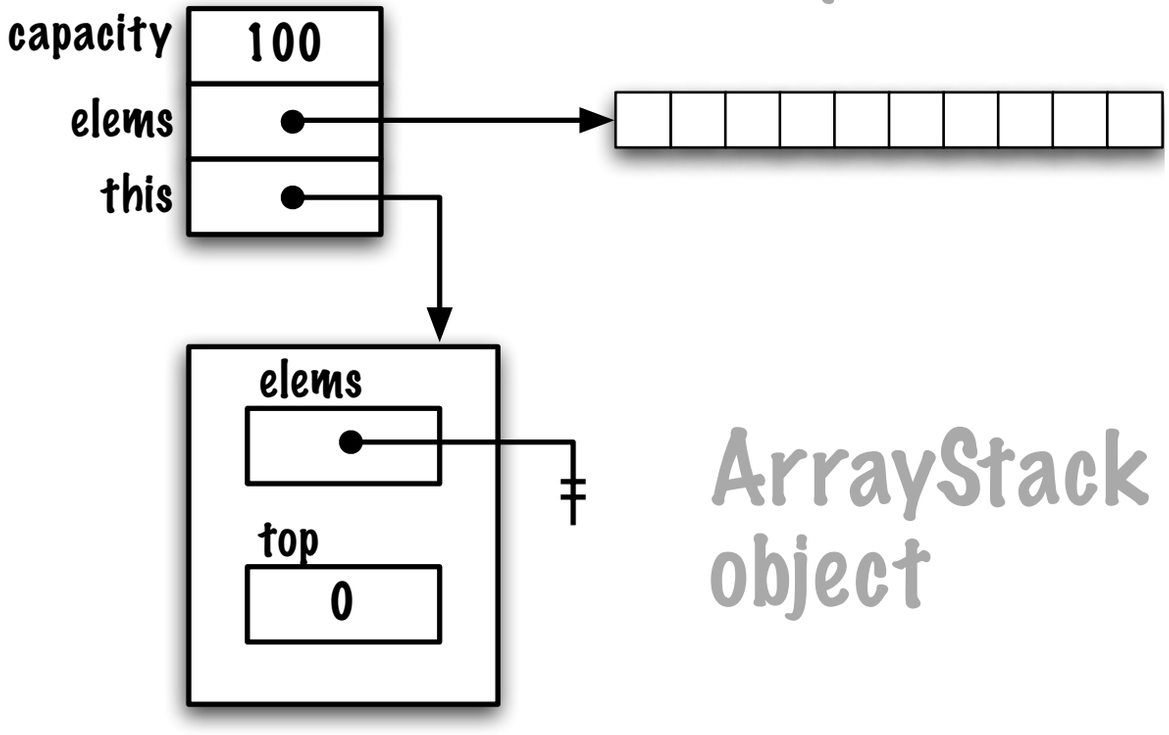


ArrayStack object

Activation Frame for ArrayStack

formal parameter(s)

local variable(s)

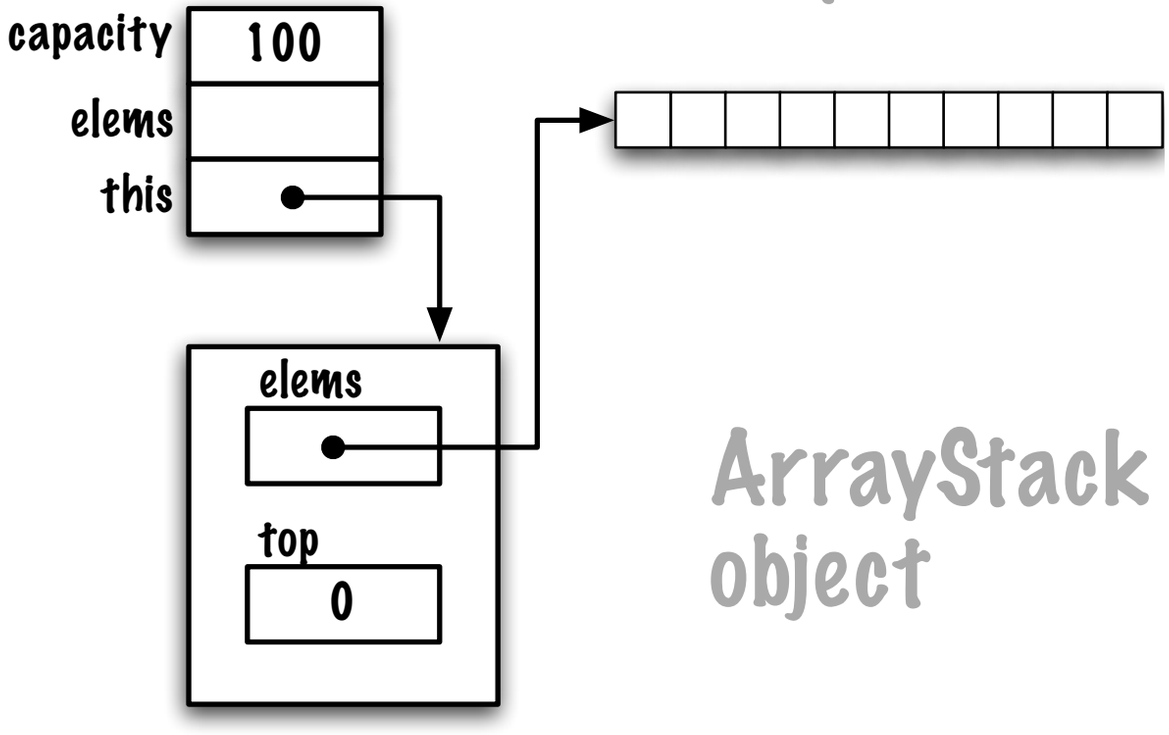


ArrayStack object

Activation Frame for ArrayStack

formal parameter(s)

local variable(s)



ArrayStack object

Implementing a Stack using an array: ArrayStack

```
// Returns the top element of this ArrayStack without removing it

public Object peek() {

    // pre-conditions: ! isEmpty()

    return elems[ top-1 ];
}
```

ArrayStack (using Java 1.5)

Do you remember the problem with the pre-Java 1.5 implementation of the class **Pair**? Well, the same problem is occurring with **ArrayStack**.

That's right, using references of type **Object** (for the instance variables and parameters) allows us to write a single implementation that can be used in a variety of contexts (to store **String**, **Time** and **Event** objects, to name a few).

However, the type of the return value is **Object**, which forces the caller to cast the type of that value.

Pair

```
public class Pair {
  private ______ first;
  private ______ second;
  public Pair( ______ first, ______ second ) {
    this.first = first;
    this.second = second;
  }
  public ______ getFirst( ) {
      return first;
  }
  public  ______ getSecond( ) {
      return second;
  }
}
```

Pair

```
public class Pair<T> {
    private T first;
    private T second;
    public Pair( T first, T second ) {
        this.first = first;
        this.second = second;
    }
    public T getFirst( ) {
        return first;
    }
    public T getSecond( ) {
        return second;
    }
}
```

Pair

```
Pair<String> name;
name = new Pair<String>( "Hilary", "Clinton" );

Pair<Time> times;
name = new Pair<Time>( new Time( 10,0,0 ), new Time( 11,30,0 ) );

String s;
s = name.getFirst();

Time t;
t = times.getFirst();
```

ArrayStack with Generics

```
Stack<String> s1;
name = new ArrayStack<String>( 100 );

Stack<Time> s2;
name = new ArrayStack<Time>( 1024 );

s1.push( "alpha" );
s2.push( new Time( 23,0,0 ) );

String a;
a = s1.pop();
```

ArrayStack with Generics

What are the required changes?

```
public class ArrayStack implements Stack { ... }
```

The header of the class becomes,

```
public class ArrayStack<E> implements Stack<E> { ... }
```

ArrayStack with Generics

What are the required changes for the instance variables?

```
// Instance variables
private Object[] elems;
private int top;
```

Becomes,

```
public class ArrayStack<E> implements Stack<E> {

    private E[] elems;
    private int top;

    // ...
```

ArrayStack with Generics

What are the required changes for the constructor?

```
public ArrayStack( int capacity ) {
    elems = new Object[ capacity ];
    top = 0;
}
```

A generic array seems appropriate,

```
public ArrayStack( int capacity ) {
    elems = new E[ capacity ];
    top = 0;
}
```

ArrayStack with Generics

```
public class ArrayStack<E> implements Stack<E> {
    private E[] elems;
    private int top;

    public ArrayStack( int capacity ) {
        elems = new E[ capacity ];
        top = 0;
    }
```

However, this causes the following compile-time error.

```
ArrayStack.java:11: generic array creation
        elems = new E[ capacity ];
                ^
1 error
```

ArrayStack with Generics

For back-compatibility reasons, the creation of generic arrays is not possible, sadly!

ArrayStack with Generics

We still have our problem to solve! Our approach will be.

```
public class ArrayStack<E> implements Stack<E> {
    private E[] elems;
    private int top;

    public ArrayStack( int capacity ) {
        elems = (E[]) new Object[ capacity ];
        top = 0;
    }
    // ...
```

which causes a compile-time warning.

```
Note: ArrayStack.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

ArrayStack with Generics

The warning can be suppressed locally.

```
public class ArrayStack<E> implements Stack<E> {
    private E[] elems;
    private int top;

    @SuppressWarnings( "unchecked" )
    public ArrayStack( int capacity ) {
        elems = (E[]) new Object[ capacity ];
        top = 0;
    }
    // ...
```

which is a better option than globally suppressing warnings.

```
> javac -Xlint:unchecked ArrayStack.java
```

Continuing with the implementation of ArrayStack

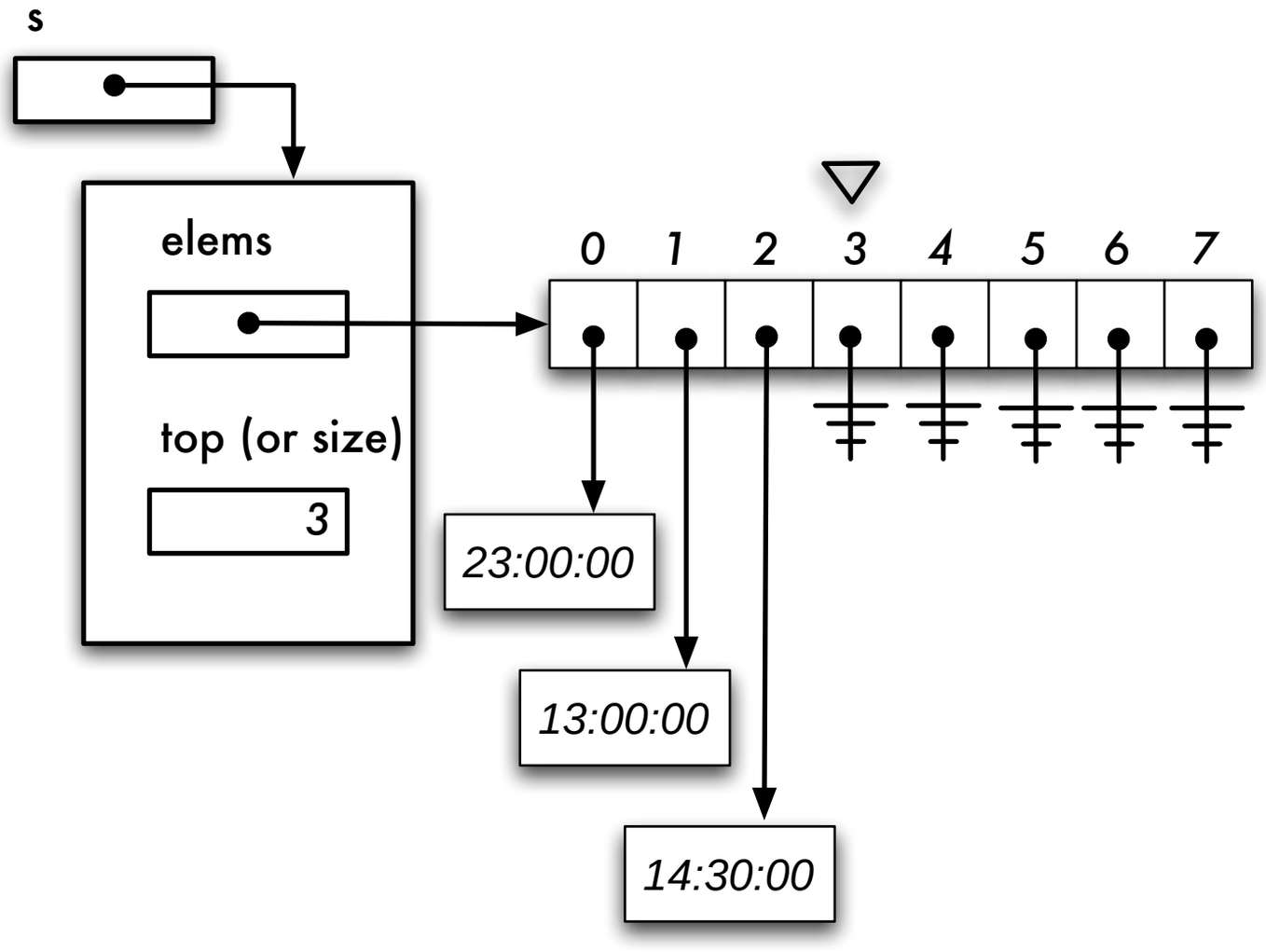
```
public void push( E element ) {
    // pre-condition: the stack is not full

    // stores the element at position top, then increments top
    elems[ top++ ] = element;
}
```

Continuing with the implementation of ArrayStack

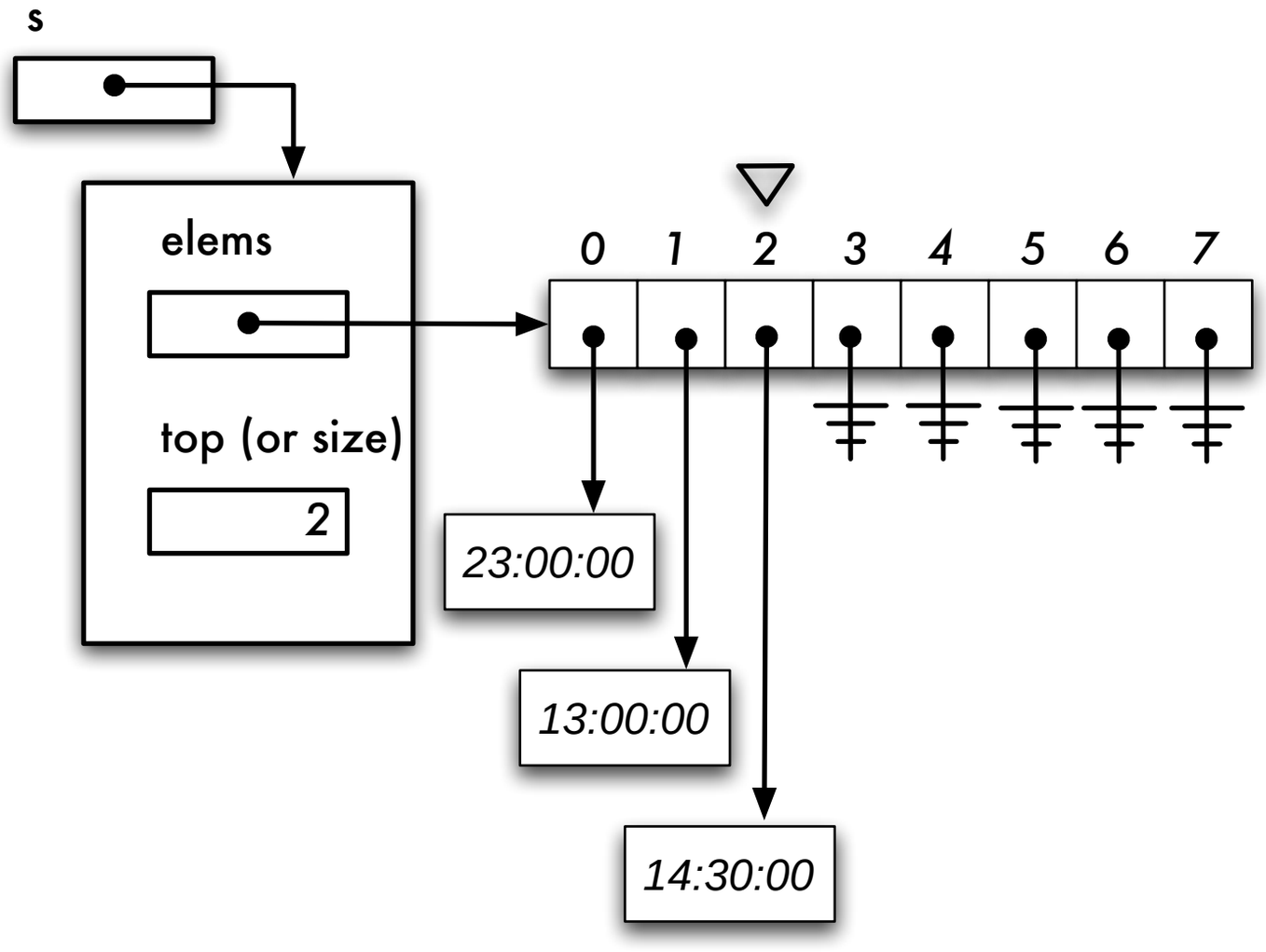
```
    // Removes and returns the top element of this stack
    public E pop() {
        // pre-conditions: ! isEmpty()

        // decrements top, then access the value
        E saved = elems[ --top ];

        return saved;
    }
}
```

This compiles and runs, but!?

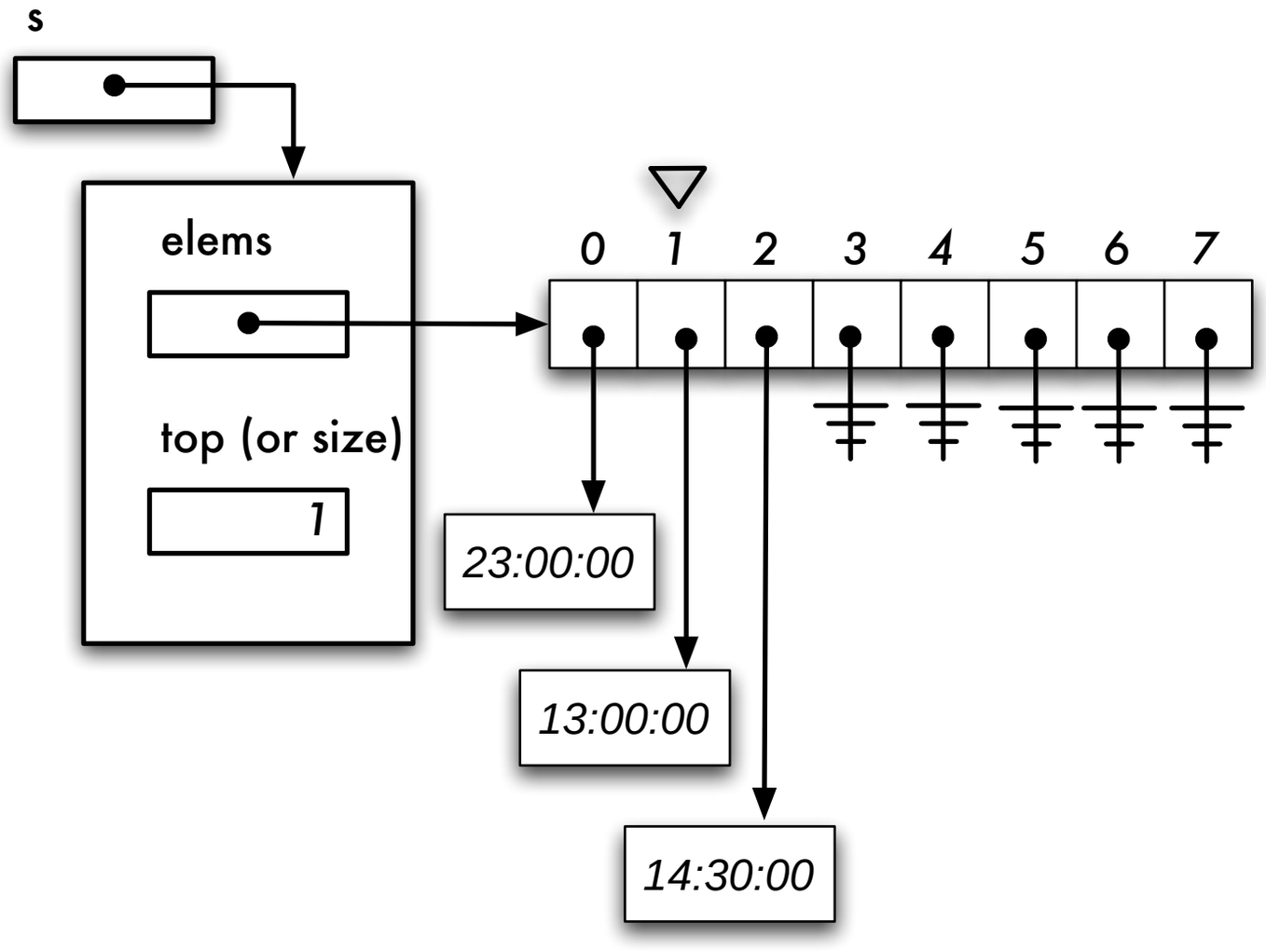
Continuing with the implementation of ArrayStack



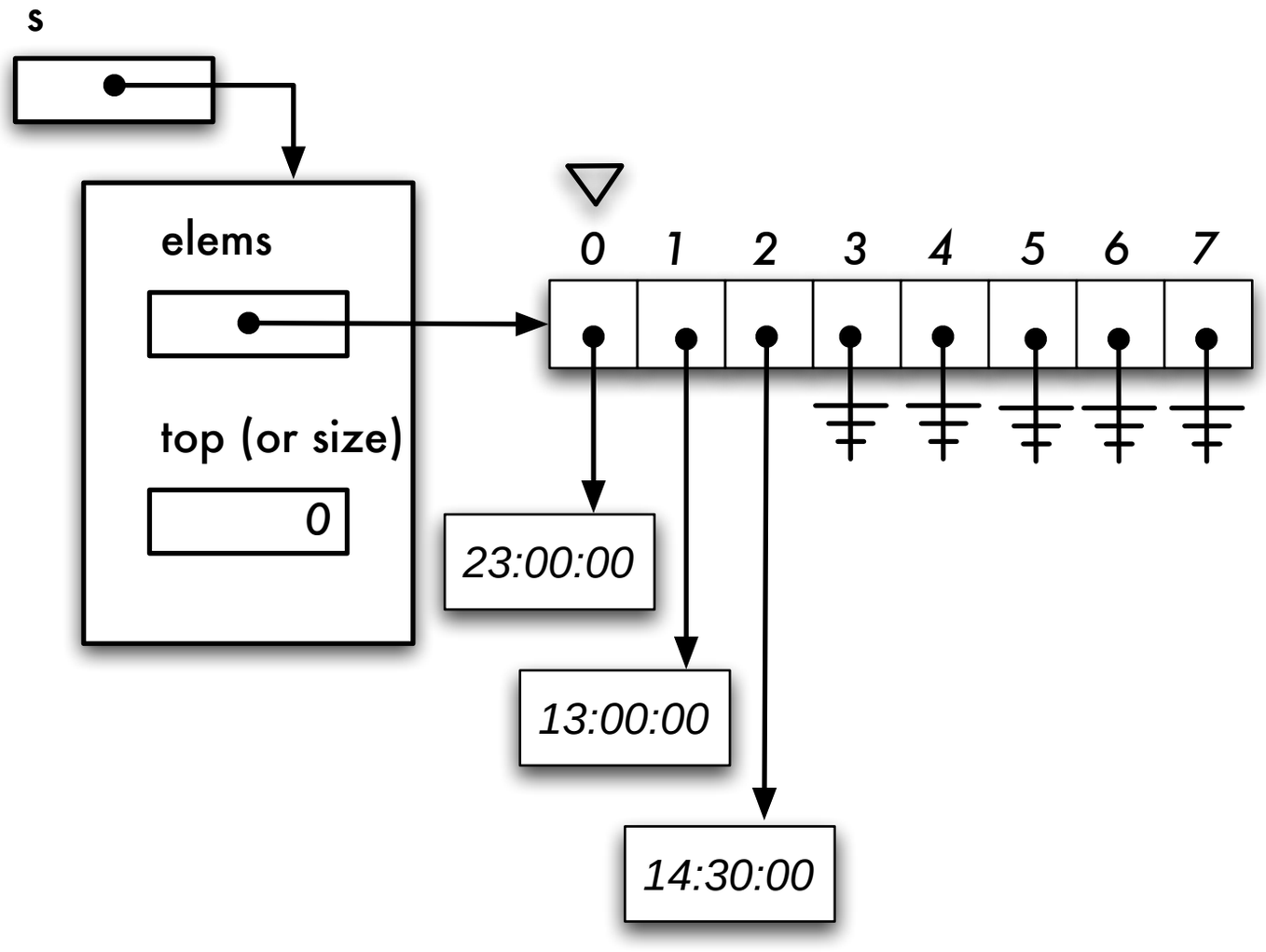
Continuing with the implementation of ArrayStack



Continuing with the implementation of ArrayStack



Continuing with the implementation of ArrayStack



Continuing with the implementation of ArrayStack

Java handles memory management tasks for us but memory leaks are possible!

```
    public E pop() {
        // pre-conditions: ! isEmpty()

        // decrements top, then access the value
        E saved = elems[ --top ];

        elems[ top ] = null; // ‘‘scrubing’’ the memory!

        return saved;
    }
}
```

Implementing a Stack using an array

The current implementation has a major limitation, its size is fixed.

Often, the number of elements to process is not known in advance.

How would you circumvent this limitation?

- Allocate an array that will be sufficiently large for most applications. What are the disadvantages of this approach? Most of the time, most of the allocated memory will not be used, the capacity can be exceeded;

- Solution: dynamic array.

Implementing a Stack using an array: dynamic array

Some programming languages are allowing you to change the size of the arrays at runtime. Those languages are simply using the technique presented below.

- When the array is full, a new bigger array is created, the elements from current array are moved (copied) into the new one, finally, the new array replaces the current one;

- The are many strategies to increase the size of the array, two of them are: let n and n' be the size of the current and new array respectively, $n' = n + c$, where $c = 1$ for instance, or, $n' = c \times n$, where $c = 2$;

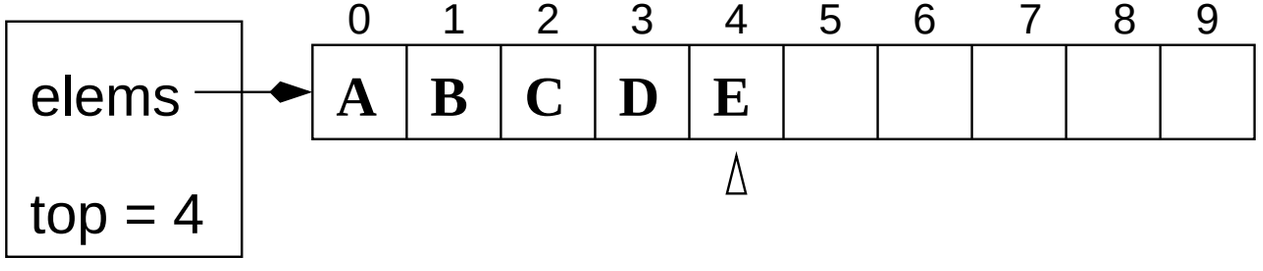
- Comment on the pros and cons of the two approaches.

Annexe

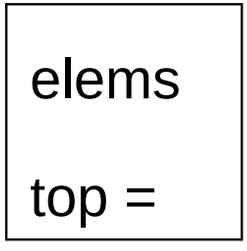
The following information is from an earlier version of the lecture notes.

Using an array

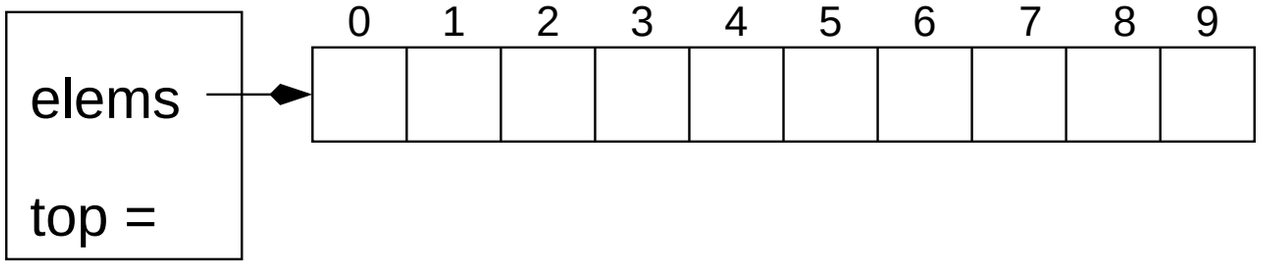
1.1 Using low memory addresses, the instance variable `top` designates the top element.



1.1 Creating a new stack

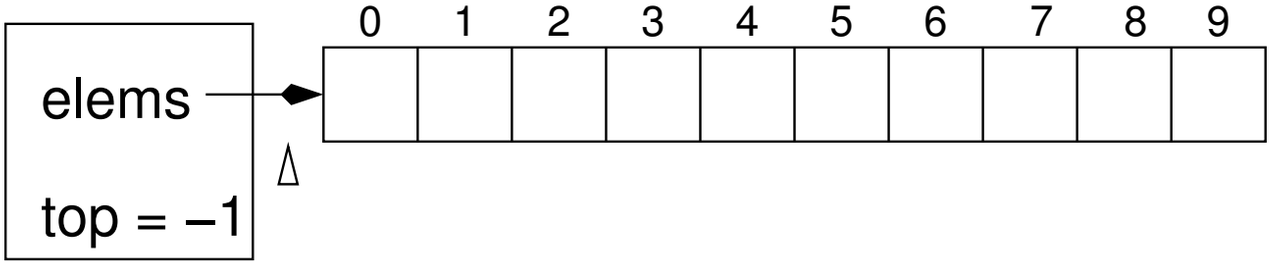


1.1 Creating a new stack (contd)



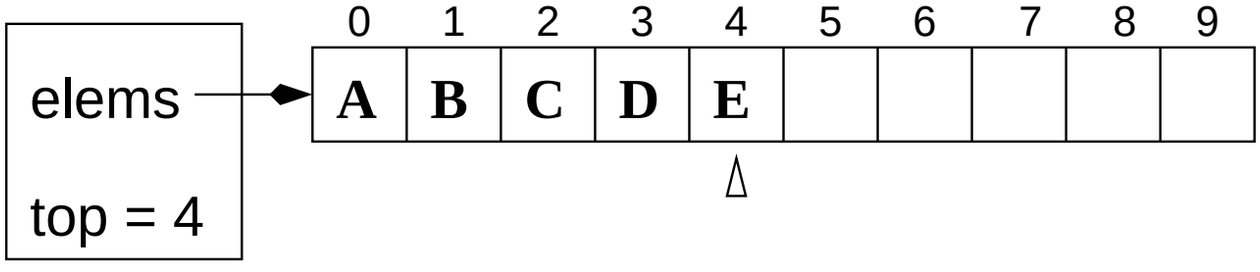
Allocating an array of size `MAX_STACK_SIZE`, here 10.

1.1 Creating a new stack (contd)



Initialize **top** to -1.

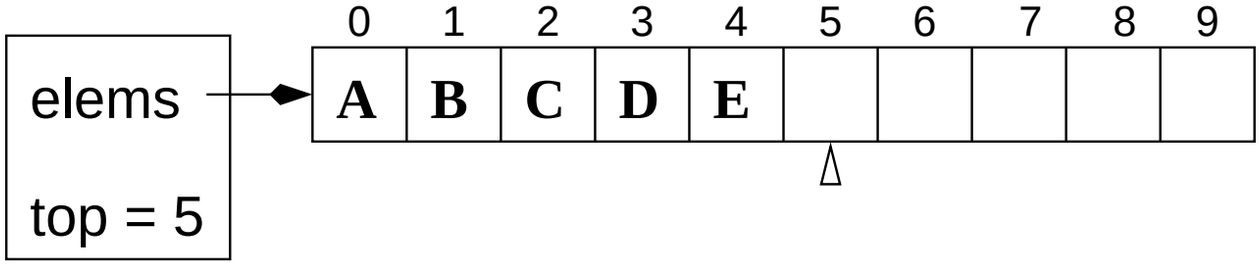
1.1 push(F)



Two steps:

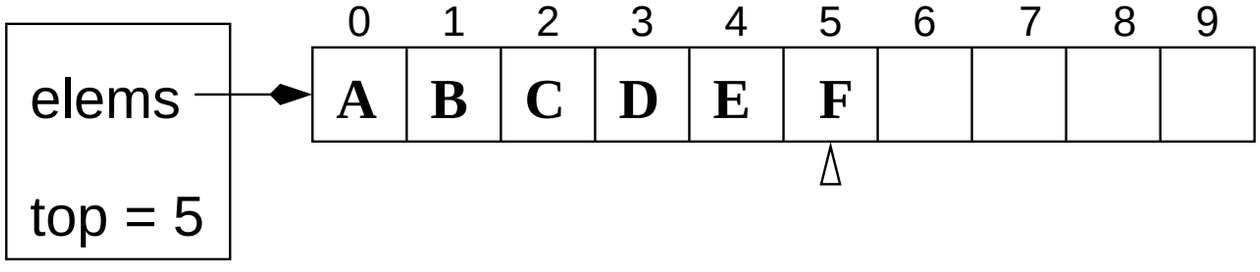
- increment `top`,
- insert the new element.

1.1 push(F)



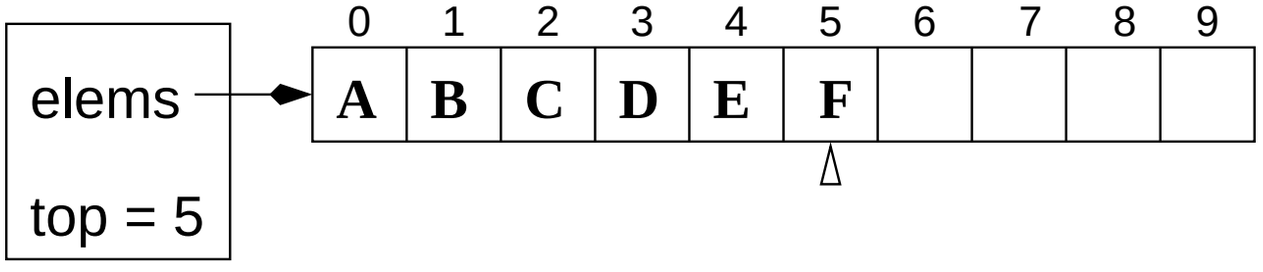
increment `top`

1.1 push(F)



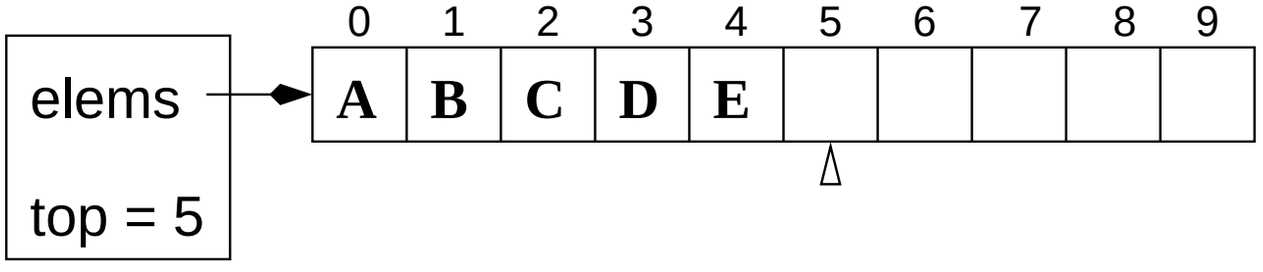
insert the new element.

1.1 pop()



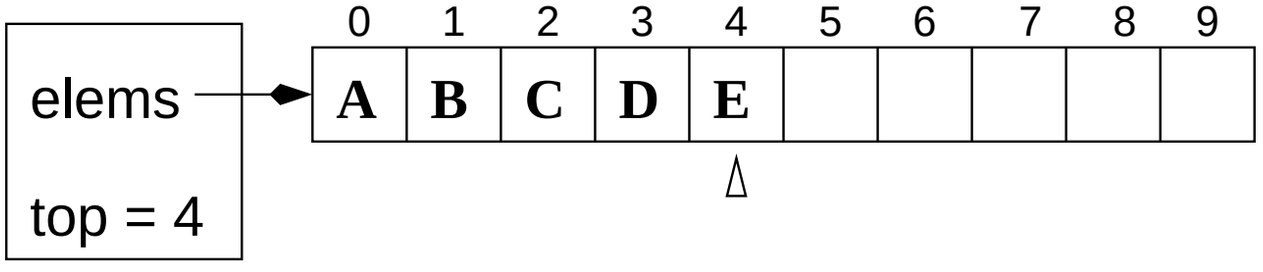
Save the content of the element on the top to a local temporary variable (say result).

1.1 pop()

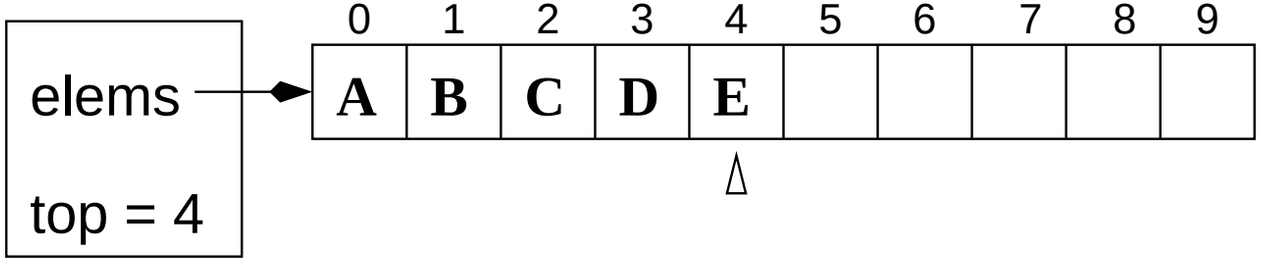


set stack[top] to its default value (0 for an int, '\u0000' for a char *null* for a reference.

1.1 pop()



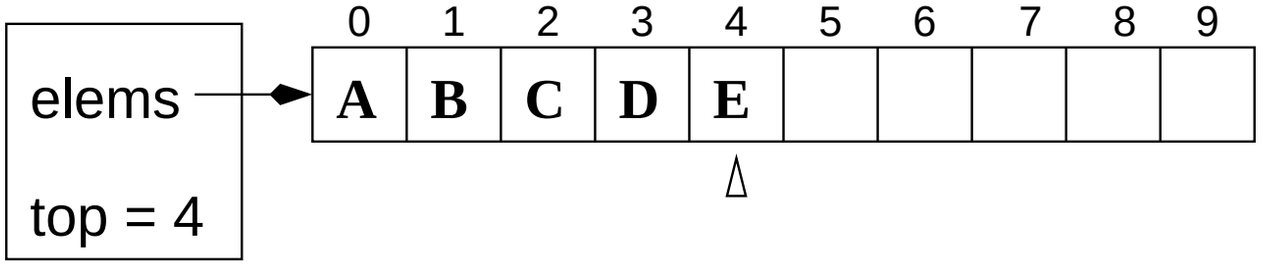
decrement top.



return the result.

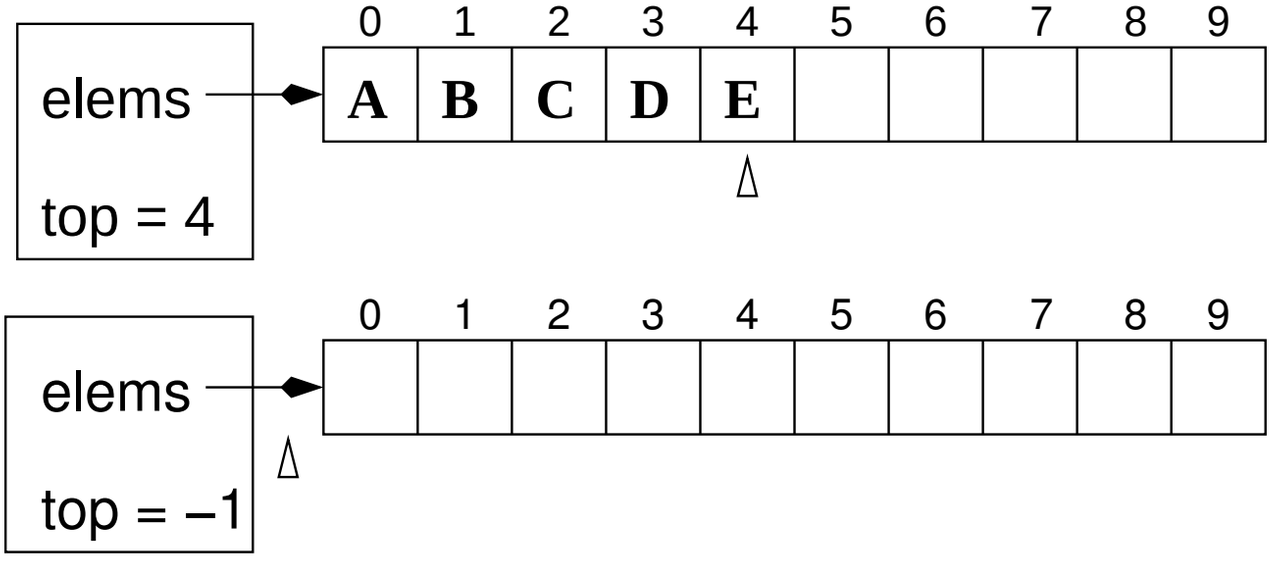
```
                top =   4, stack->[A,B,C,D,E, , , , , ]
                                             ^
E <- pop()      top =   3, stack->[A,B,C,D, , , , , , ]
                                           ^
D <- pop()      top =   2, stack->[A,B,C, , , , , , , ]
                                         ^
push(G)         top =   3, stack->[A,B,C,G, , , , , , ]
                                           ^
push(H)         top =   4, stack->[A,B,C,G,H, , , , , ]
                                             ^
push(I)         top =   5, stack->[A,B,C,G,H,I, , , , ]
                                               ^
push(J)         top =   6, stack->[A,B,C,G,H,I,J, , , ]
```

1.1 peek()



Return the element that is found at the top of the stack; without changing the state of the stack.

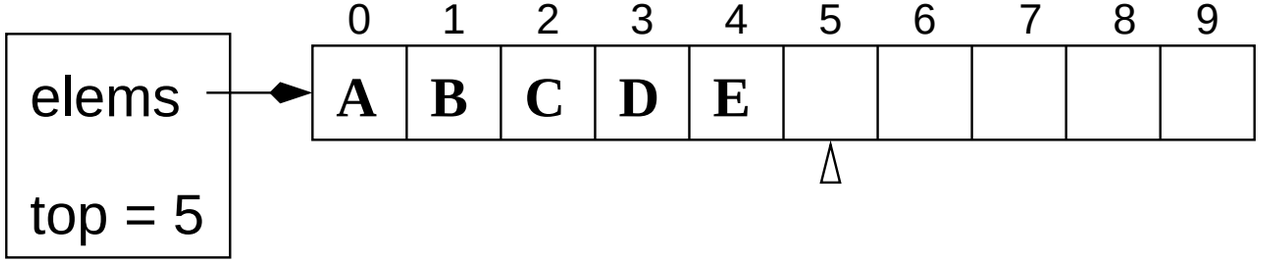
1.1 empty()



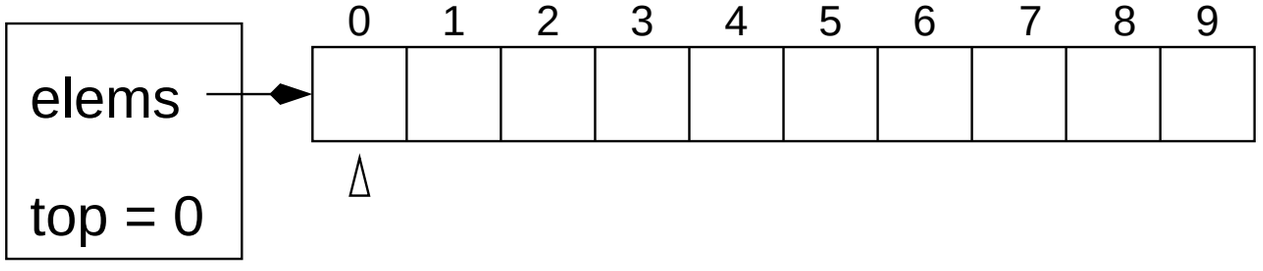
The stack is empty if the value of top is -1.

Array-based implementation -2-

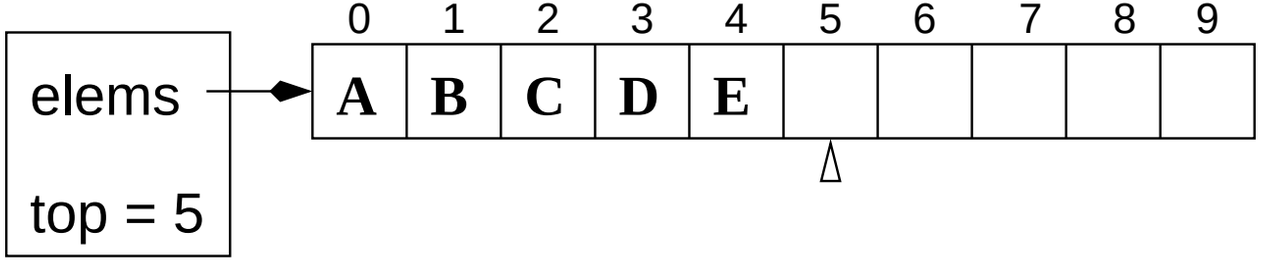
1.2 Lower part of the array, the variable top designates the first empty cell.



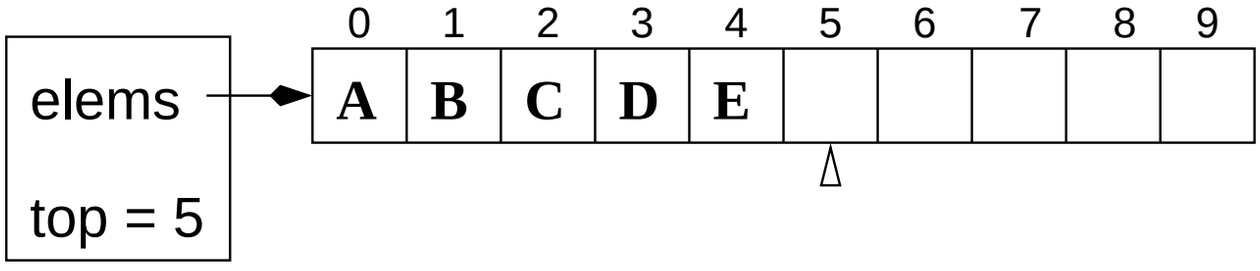
How does this affect creating a new stack?



top is initialized to 0; rather than -1.



Changes to push()? This affects the sequence of operations: the new element is inserted first at the location designated by top, then top is incremented.



How does this affect pop?

The value of top is decremented first, the value is saved into a local variable, reset the value of stack[top] and return the saved value.

How about peek?

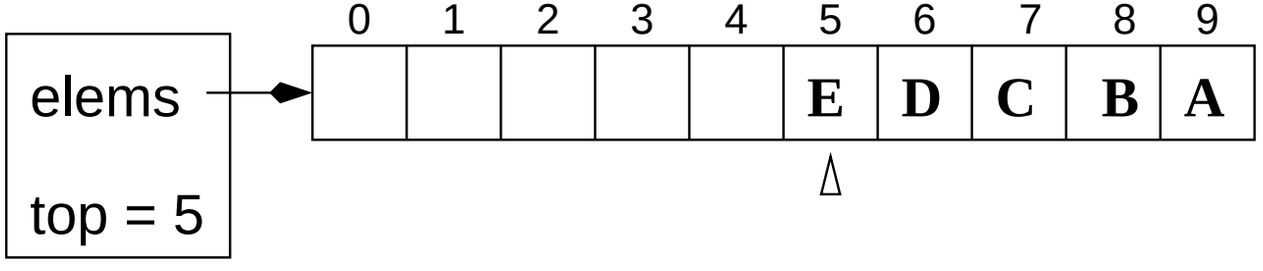
`returns stack[top-1]`

empty()?

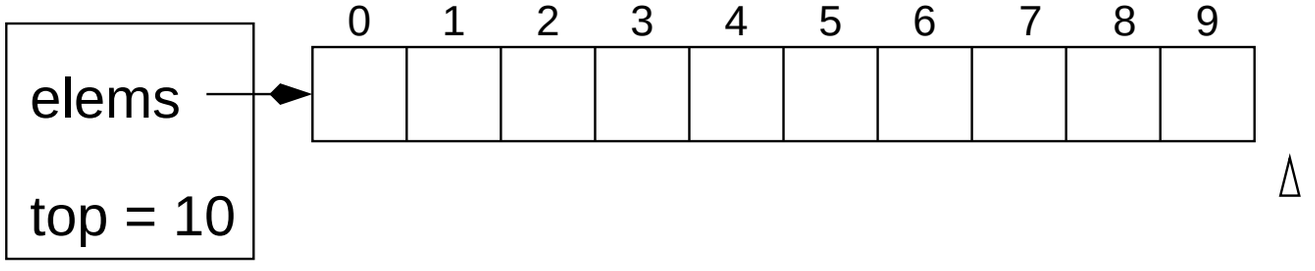
`check if top equals 0.`

Array-based implementation

2.1 Using the high part of the array, top designates the top element.

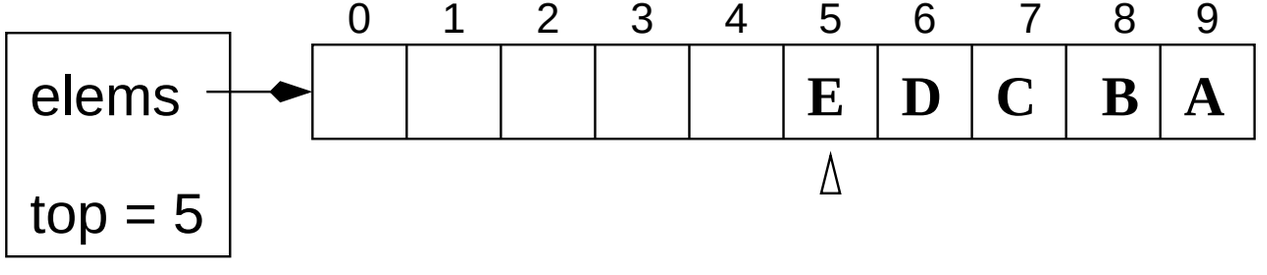


Creating a new stack.



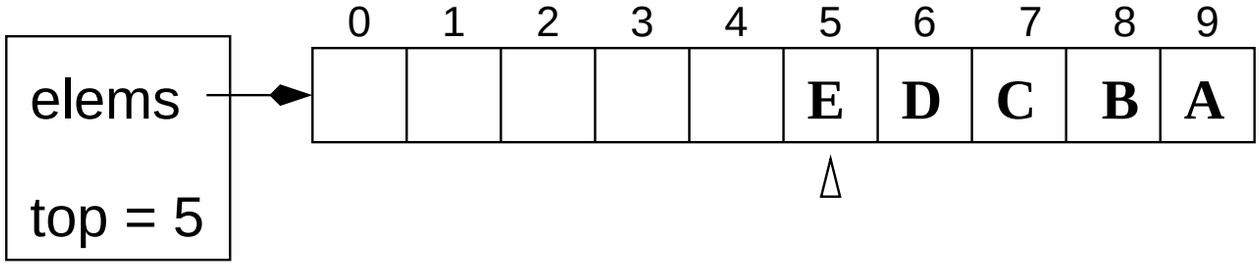
The variable top is initialized to `MAX_STACK_SIZE`; rather than -1.

push()



decrement top,
insert the new value at the location designated by top.

pop()



increment top; rather than decrementing it:

save the top value,

reset stack[top],

increment top,

return the saved value.

peek() stays the same

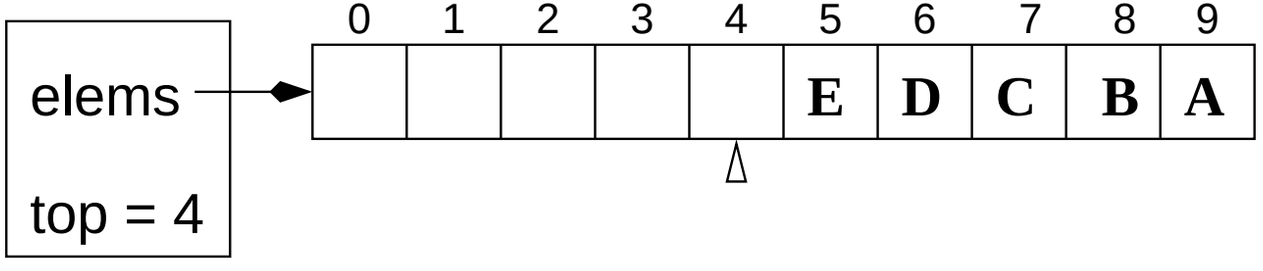
`returns stack[top]`

isEmpty()?

`top == MAX_STACK_SIZE?`

Of course the last possibility is

2.2 implementing the stack in the high part of the array, top designates the first free cell.



$\Rightarrow$ the 4 implementations are equally efficient and satisfactory, which one to choose might depend on the context.

Imagine that . . .

Can we fix the position of the top element, say top == 0 always, and we would use an instance variable to designate the bottom of the stack.

E.g.:

```
            bottom = -1   stack -> [ , , , , , , , , , ]
                                    ^
push(A)     bottom =  0   stack -> [A, , , , , , , , , ]
                                     ^
push(B)     bottom =  1   stack -> [B,A, , , , , , , , ]
                                       ^
push(C)     bottom =  2   stack -> [C,B,A, , , , , , , ]
                                         ^
C <- pop()  bottom =  1   stack -> [B,A, , , , , , , , ]
                                       ^
```

⇒ first: peek() would always return stack[0].

```
bottom =  1   stack -> [B,A, , , , , , , , ]
                          ^
```

push(C)?

```
Increment bottom

for i=bottom until 1 (decreasing loop)
  stack[i] = stack[i-1]

insert the new element, i.e. stack[0] = C

bottom =  1   stack -> [B,A, , , , , , , , ]
                          ^
bottom =  2   stack -> [B,A,A, , , , , , , ]
                            ^
bottom =  2   stack -> [B,B,A, , , , , , , ]
                            ^
bottom =  2   stack -> [C,B,A, , , , , , , ]
                            ^
```

```
            bottom =  1   stack -> [C,B,A, , , , , , , , ]
                                       ^
```

pop()?

```
returnValue = stack[0]

for i=0 until (bottom - 1) (increasing loop)
  stack[i] = stack[i+1]

Initialize stack[bottom]

Decrement bottom

return returnValue
```

```
returnValue = C
            bottom =  2   stack -> [B,B,A, , , , , , , , ]
                                       ^
            bottom =  2   stack -> [B,A,A, , , , , , , , ]
```

```
                                        ^
            bottom =  2   stack -> [B,A, , , , , , , , ]
                                          ^
            bottom =  1   stack -> [B,A, , , , , , , , ]
                                        ^
return C
```

Remarks

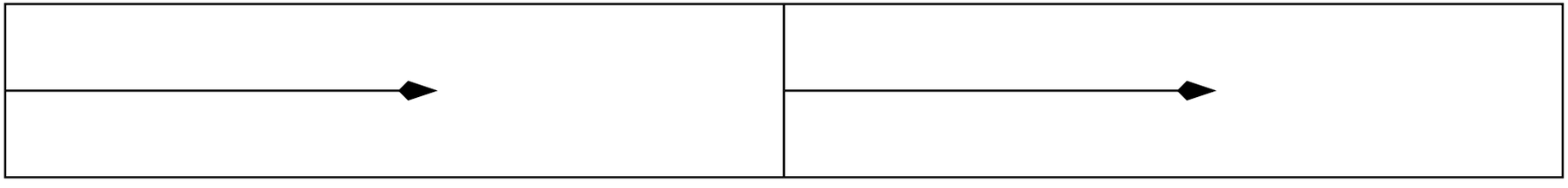
Compare the efficiency of that implementation with the previous four:

- push(): must move all the elements one position to the right,
- pop(): moves all the elements one position to the left,
- **the more elements there are in the array the more costly these operations are**.

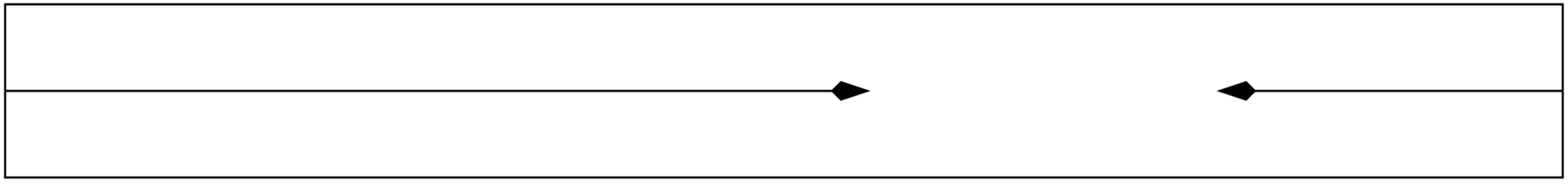
$\Rightarrow$ Which implementation do you favor, this one or 1.1?

Implementing 2 stacks in one array

and we don't mean two stacks growing in the same direction:

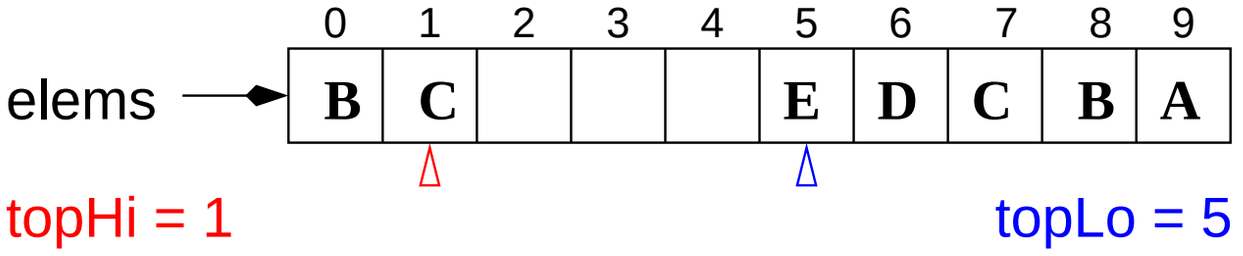


but two stacks growing in opposite directions:



How?

One array but two distinct instance variables for the two tops:

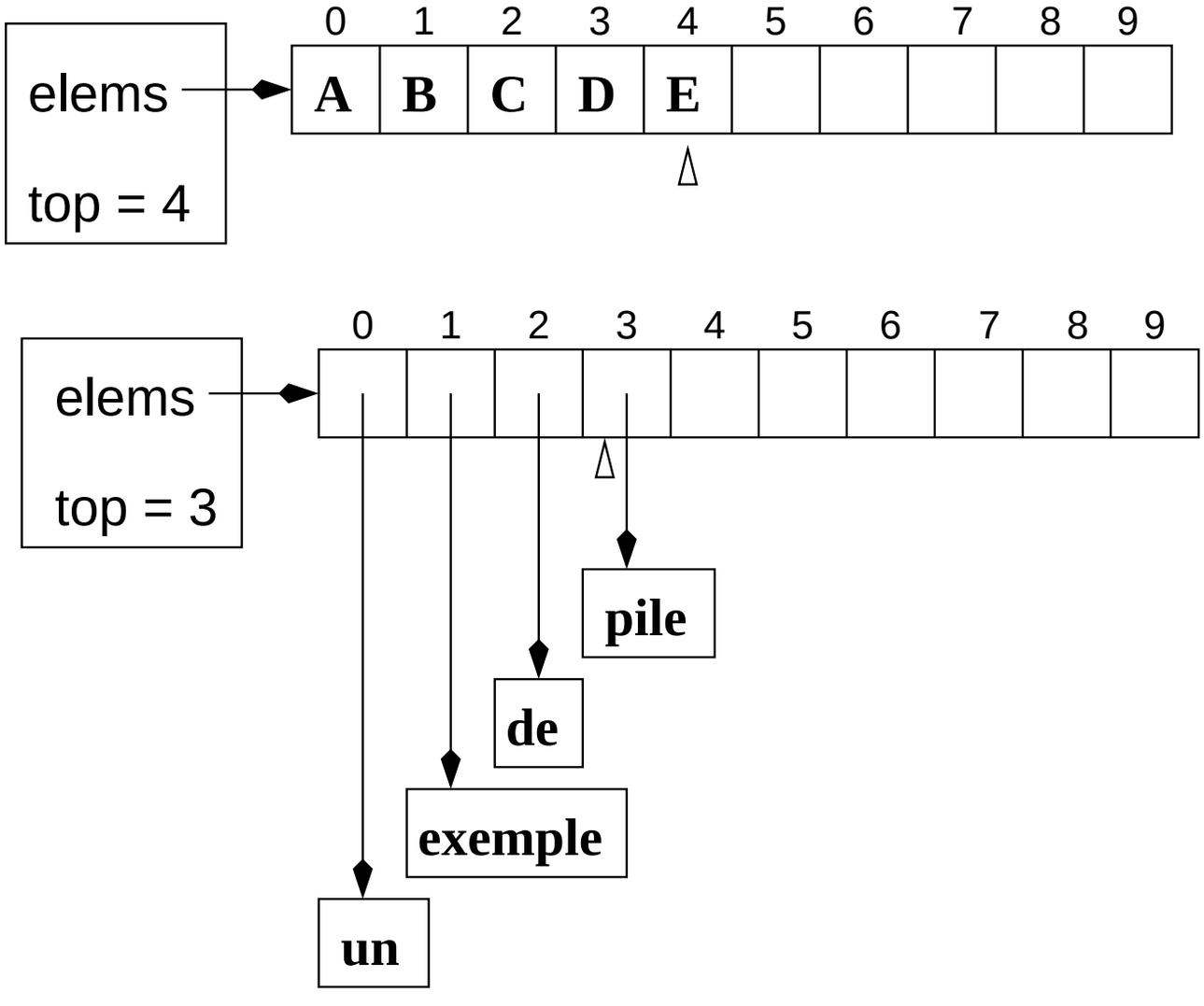


Why?

Memory management works like that.

Such implementation may reduce the amount of memory that is wasted.

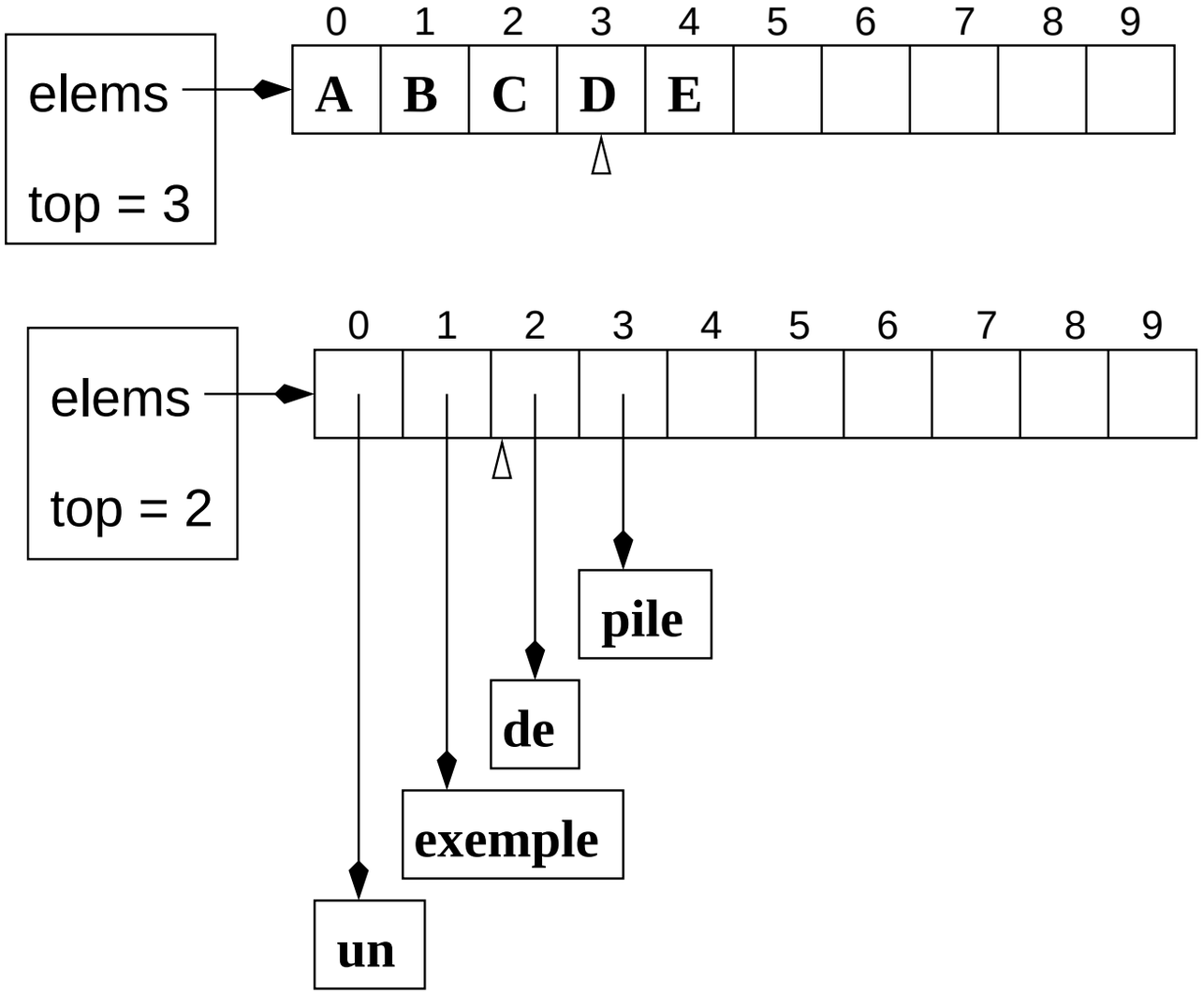
Primitive vs reference



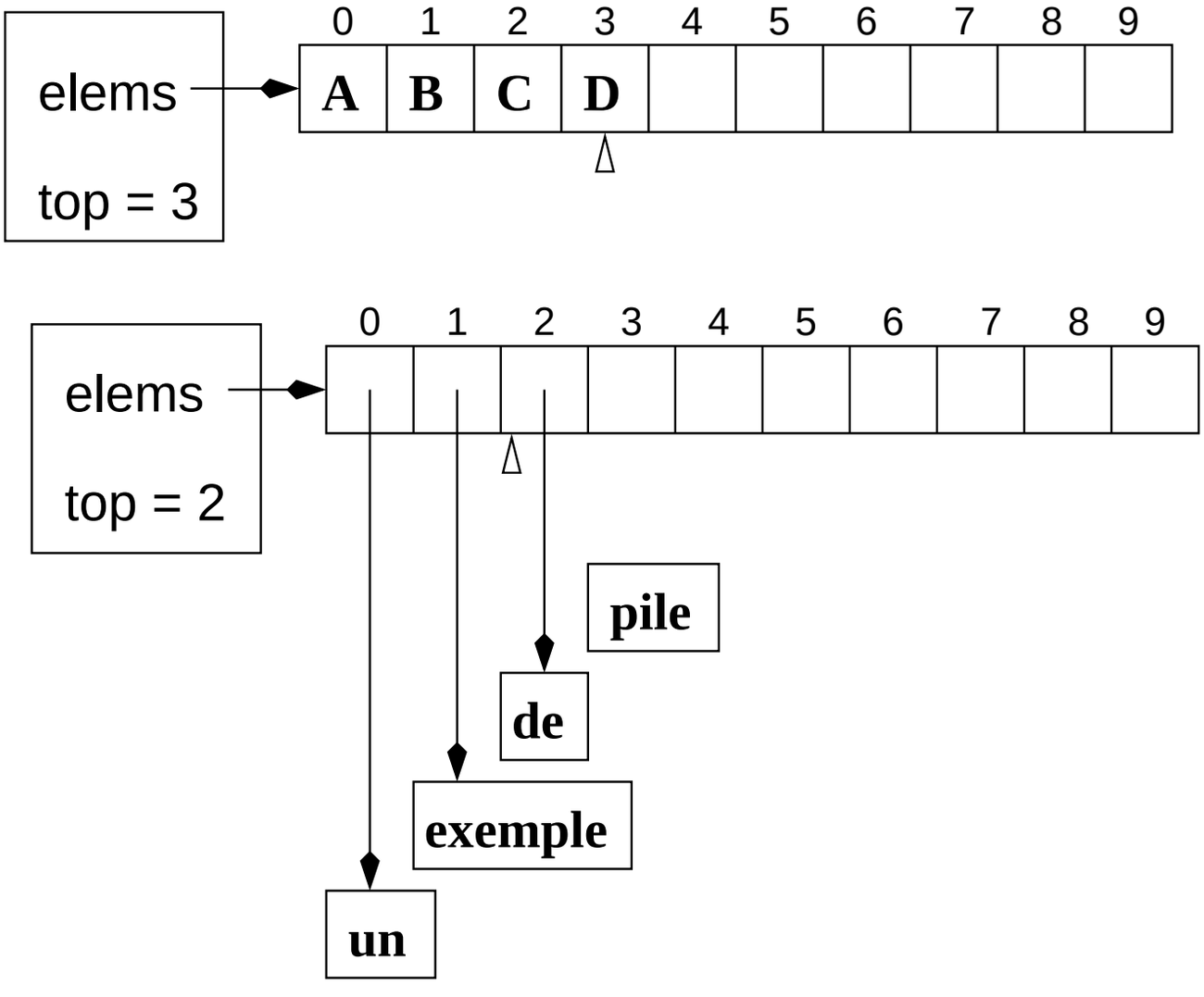
Earlier we said that pop consists of:

1. saving the current top value,
2. reset stack[top],
3. decrement top,
4. return the saved value.

Is it necessary to set the top value to null?



$\Rightarrow$ Since a reference from the "stack" to the object exists, this prevents the garbage collector from doing its job.



$\Rightarrow$ No reference to the object, therefore gc() can do its job.

Note: error condition

Pre- and post-conditions should be checked and the appropriate Exceptions should be thrown.

```
if (! s.empty())
  v = s.pop();

...

if (! s.isFull())
  s.push(v);
```

Properties of the arrays

Arrays are accessed by index position, e.g. `a[3]` designates the fourth position of the array.

Access to a position is very fast. We say that indexing is a *random access* operation in the case of the arrays, which means that the access to any element of an array always takes a constant number of steps, we say the operation necessitates **constant time**, i.e. the time to access an element is independent of:

- The size of the array;
- The number of elements that are in the array;
- The position of the element that we wish to access (first, last, middle).

Access to any element of an array is fast because the elements of an array are stored contiguously in memory.

The array starts at some address of the memory, let's call this the base address, then the first element is stored at this address and the location of the next element depends on the size of an element. All the elements of an array occupy the same amount of space and therefore the address of any element is simply,

$$\text{base address} + \text{offset}$$

where the offset is

$$\text{index} * \text{size of an element}$$

The first element ($index = 0$) is found at the base address, the second at the base address plus the size of one element, and so on.

No search involved.

Fixed size arrays

What if the size of an array is not known?

Suppose, you were asked to read positive integers from the input until a special value is read (sentinel), say -9, and the values should be stored in a array.

How large should you declare this array?[1]?

[1] Certain programming languages, such as Fortran and Pascal, require you to specify the size of the array at compile time.

Solution 1: make it large enough

A possible solution would be to create an array that would be suitable for even the largest application.

What are the consequences of such actions?

If the array is too large this wastes a lot memory.

When the array is full the program may be forced to stop.

Solution 2: variable size arrays

Create an array of a reasonable default capacity.

Increase or decrease its size according to the need.

This means that the **logical size** of the array will not correspond to its **physical size**.

Which means that it's up to the programmer to maintain information about the logical size (the instance variable `length` of an array refers to its physical size).

It's the responsibility of the programmer to access elements that are below the logical size.

Qualify the behavior of the solution as the size of the array increases.

All the elements of the array have to be copied. The more elements there are in the array the more copies are needed.

Initially, there are only few elements to be copied, but the larger the array the more copies are needed.

Insertion and resize are related to one another.

Once the logical size of the array equals its physical size every subsequent insertion necessitates resizing the array, which has a cost proportional to the number of elements in the array.

A more practical solution consists of doubling the size of the array whenever the logical size of the array equals its physical size.

What have achieved?

Not all insertions require resizing the array, hence copying its elements.

What has been lost?

Memory efficiency.

$\Rightarrow$ for some applications, the logical size of the array can also decrease, in which cases, the physical size of the array could also be decreased whenever the number of elements is below a certain threshold.

1. is the array big enough?
2. where do we start coping the elements?

$\Rightarrow$ implement remove(int pos)