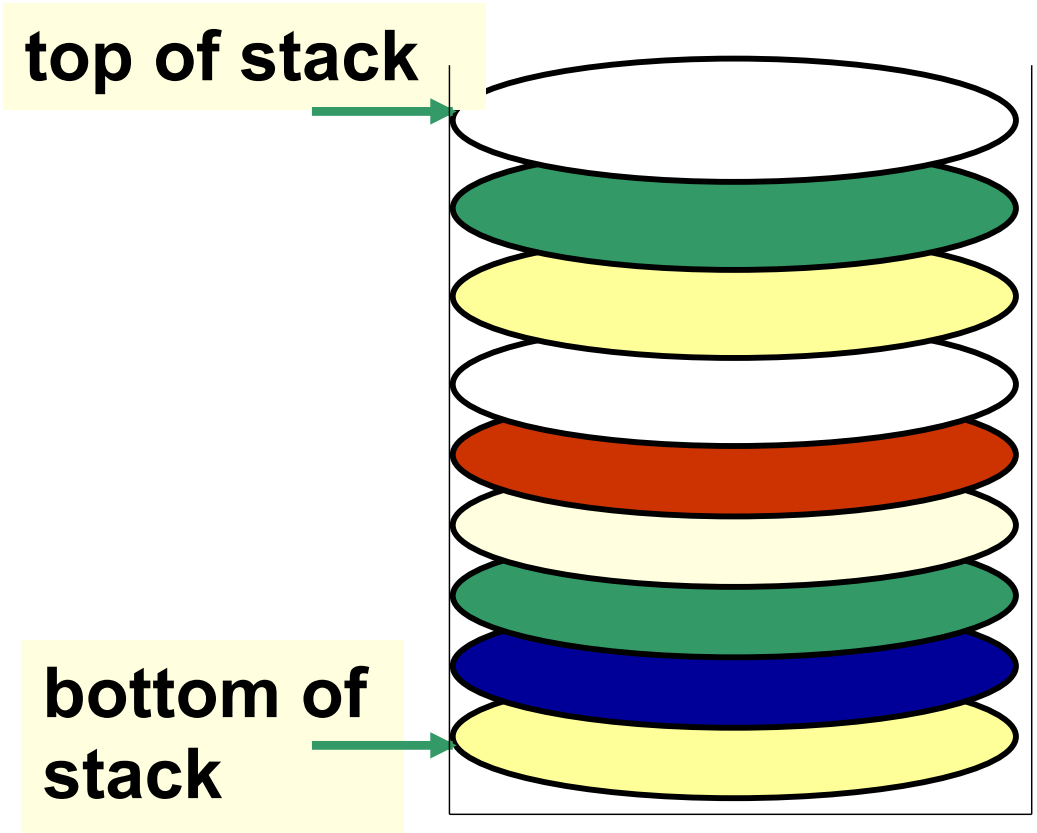
Stack ADT

Objectives

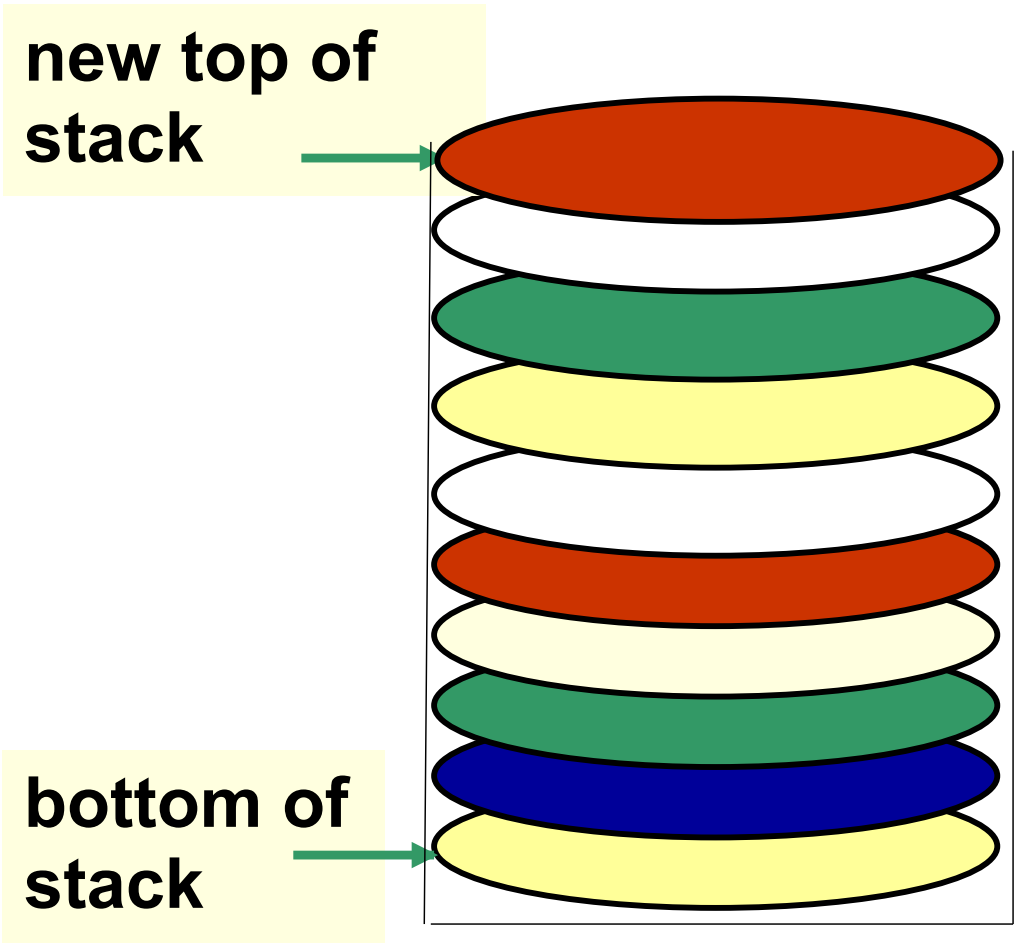
- Define the concept of a stack
- Identify the operations on the stack ADT
- Study an array implementation of stacks
- Study a linked list implementation of stacks
- Observe common uses and applications of stacks

Conceptual View of a Stack



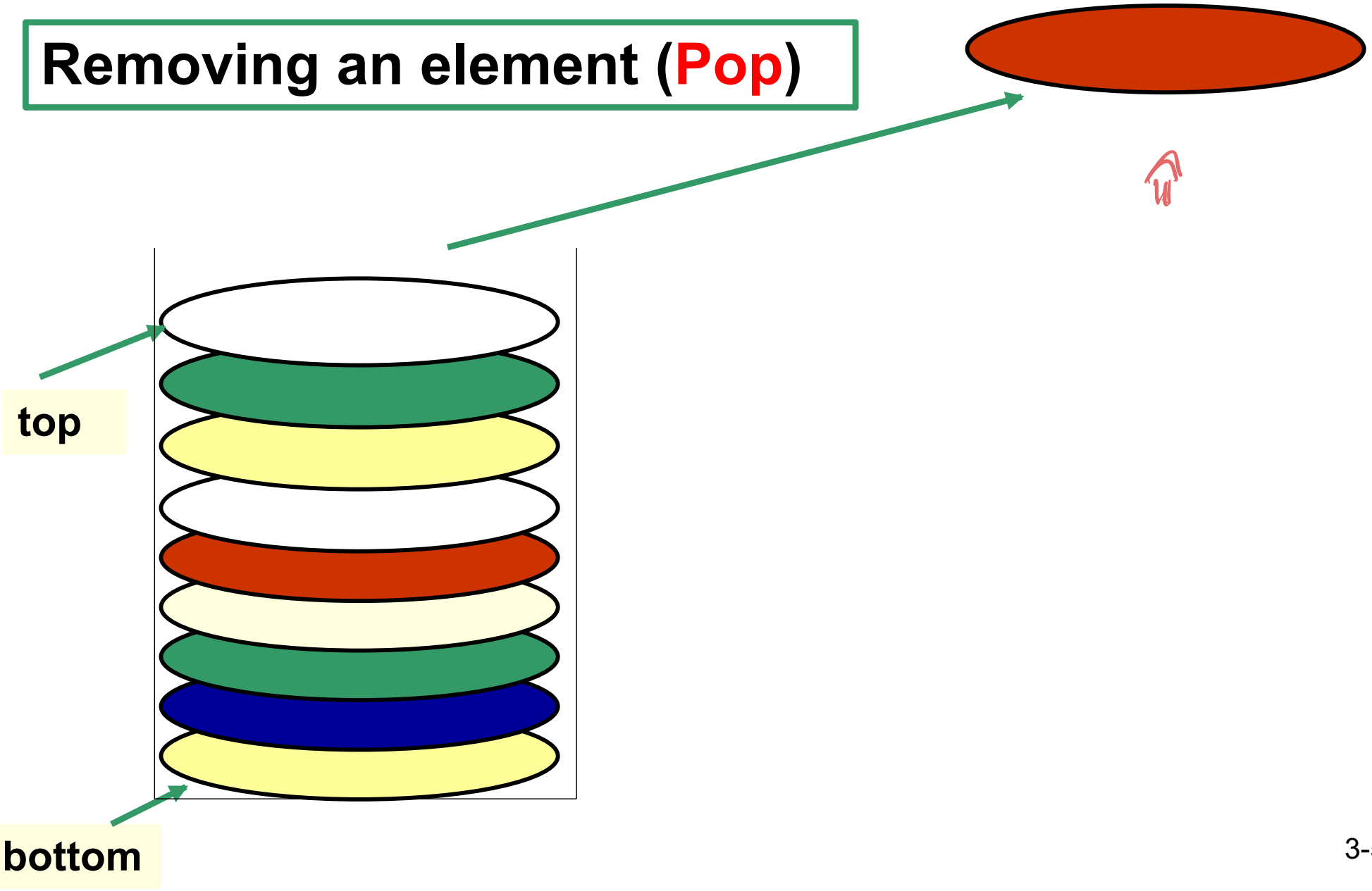
Conceptual View of a Stack

Adding an element (Push)



Conceptual View of a Stack

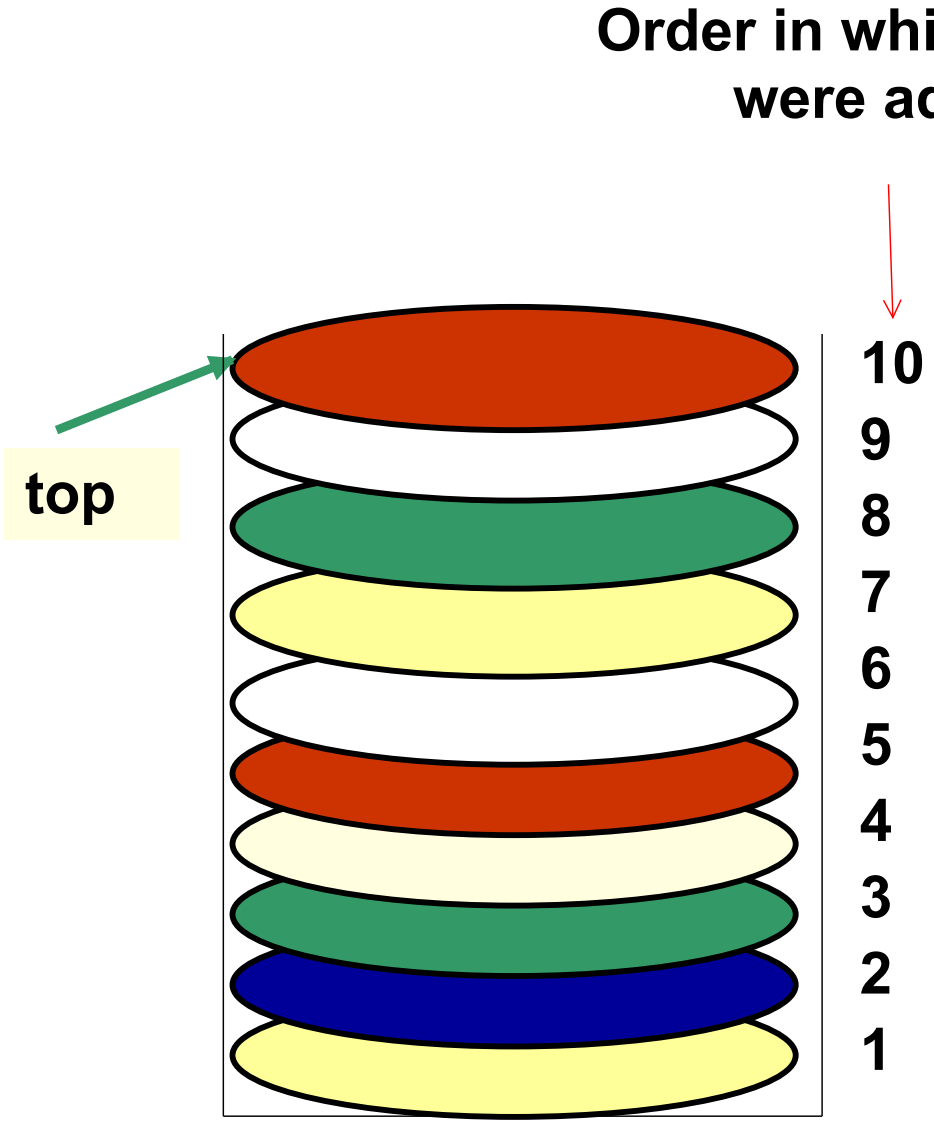
Removing an element (Pop)



Stacks

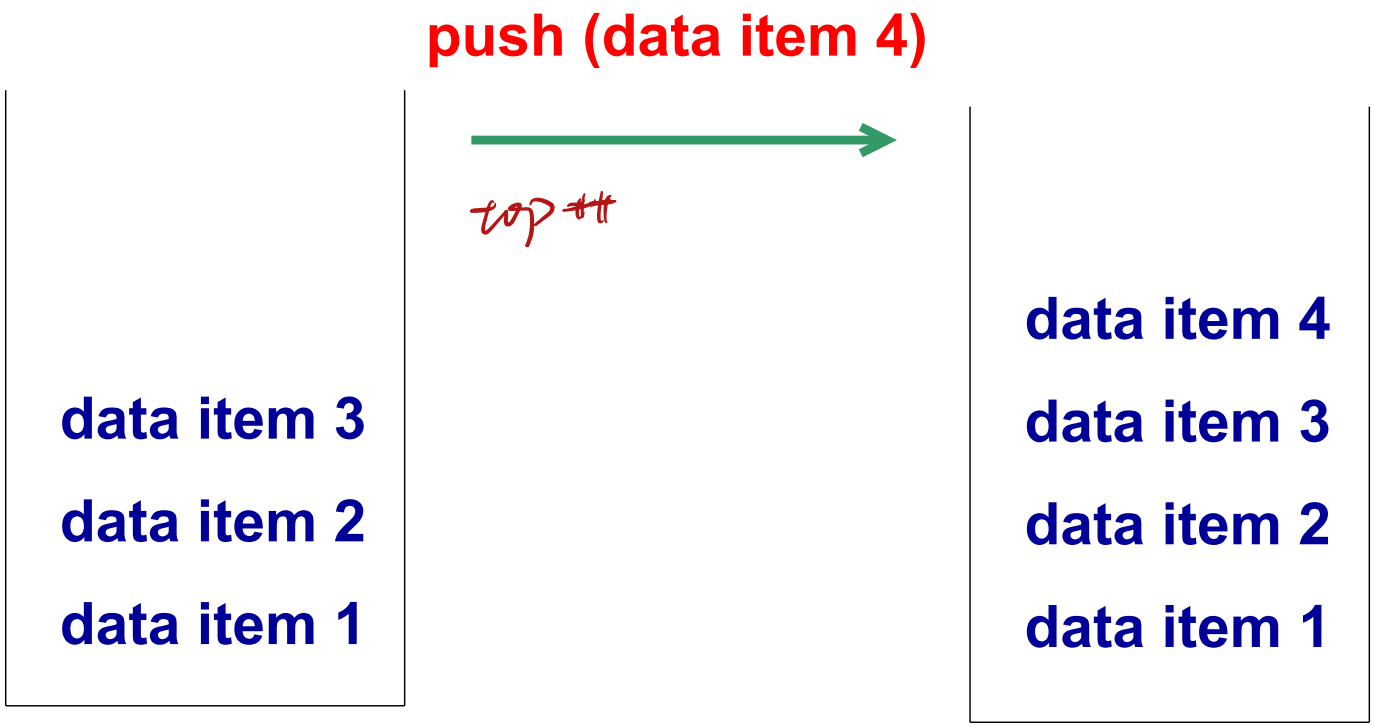
- ***Stack***: a collection whose elements are added and removed from one end, called the *top* of the stack
- Stack is a *LIFO* (**L**ast **I**n, **F**irst **O**ut) data structure

A stack is a LIFO structure



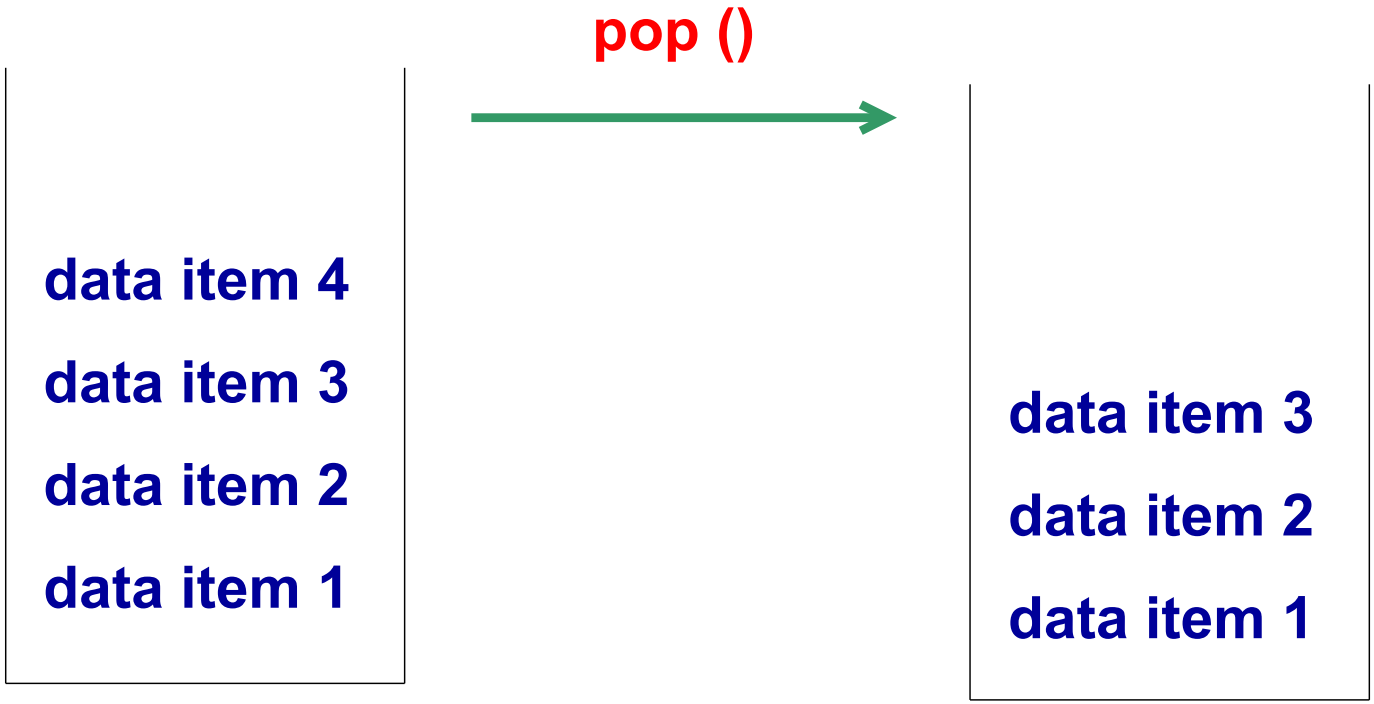
Stack Operations

- ***push***: add an element at the top of the stack



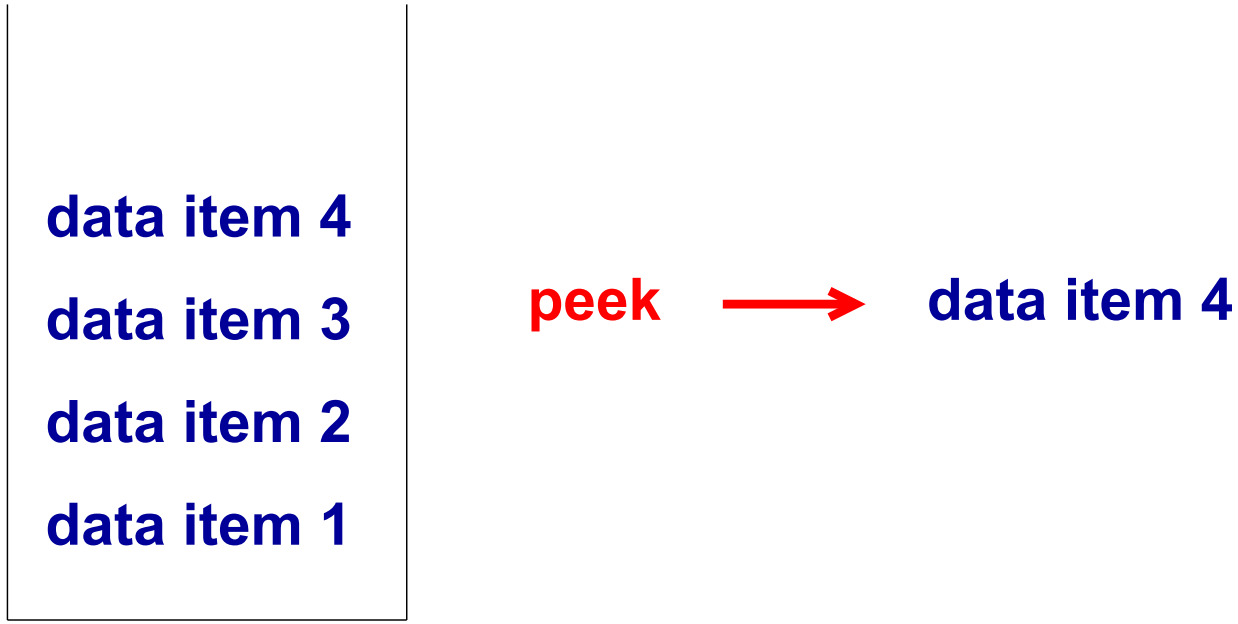
Stack Operations

- *pop*: remove the element at the top of the stack



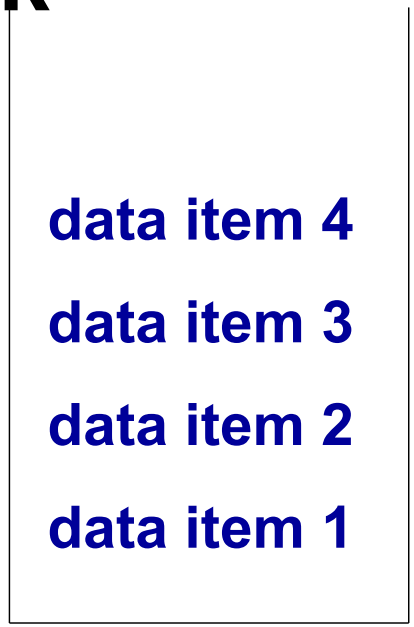
Stack Operations

- ***peek***: examine the element at the top of the stack without removing it



Stack Operations

- **size**: number of elements in the stack
- **isEmpty**: true if the stack is empty
- **toString**: string representation of stack



size → 4

isEmpty → false

toString →

"Stack:
data item 4
data item 3
data item2
data item 1"

Stack ADT

- ***Stack Abstract Data Type (Stack ADT)***

  It is a *collection* of data together with the *operations* on that data:

  - **push** - add
  - **pop** - remove
  - **peek** - look

  } top of the stack.

  - **Size**
  - **isEmpty**
  - **toString**

```
public interface StackADT<T> {
  // Adds one element to the top of this stack
  public void push (T dataItem);
  // Removes and returns the top element of this stack
  public T pop( );
  // Returns the top element of this stack
  public T peek( );
  // Returns true if this stack is empty
  public boolean isEmpty( );
  // Returns the number of elements in this stack
  public int size( );
  // Returns a string representation of this stack
  public String toString( );
}
```

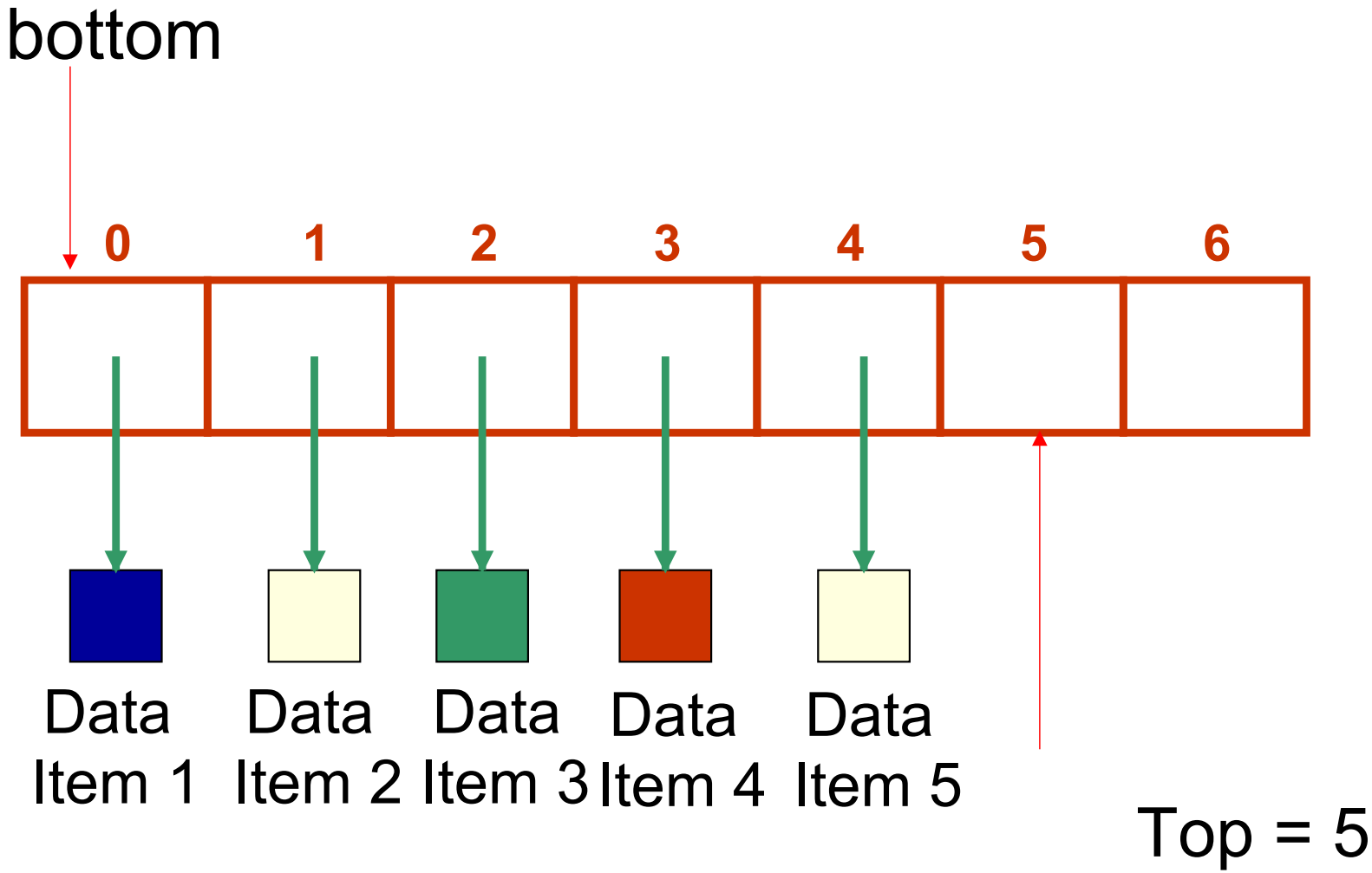
Implementing an Interface

- **Remember we cannot create an object of StackADT because it's an interface**
- **To be able to create Stack objects, we first need to create a class that *implements the interface* by providing the implementations (code) for each of the abstract methods**

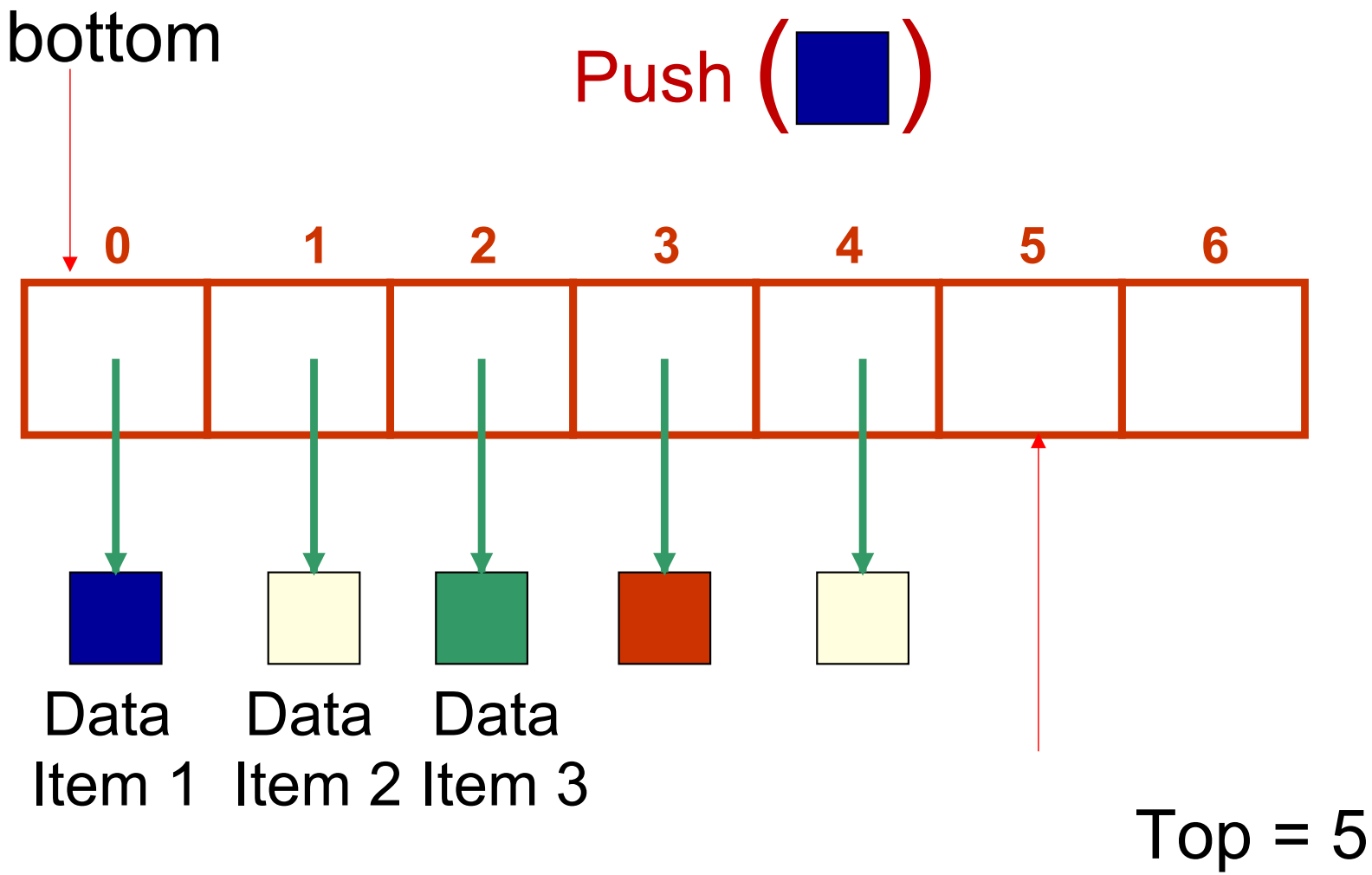
Stack Implementation Issues

- What do we need to implement a stack?
  - A data structure (*container*) to hold the data elements
  - Something to indicate the *top* and *bottom* of the stack

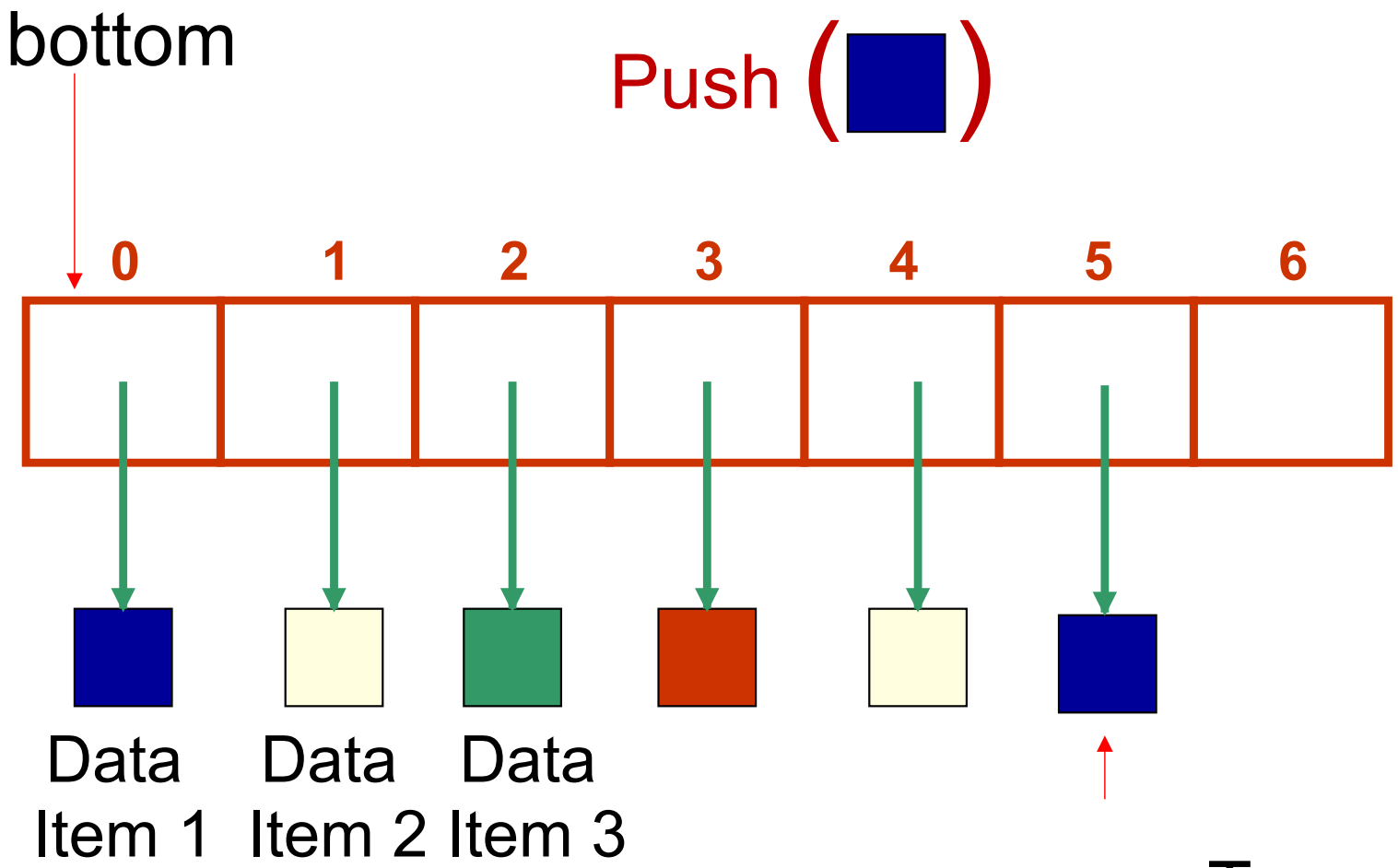
Array Implementation of a Stack



Array Implementation of a Stack

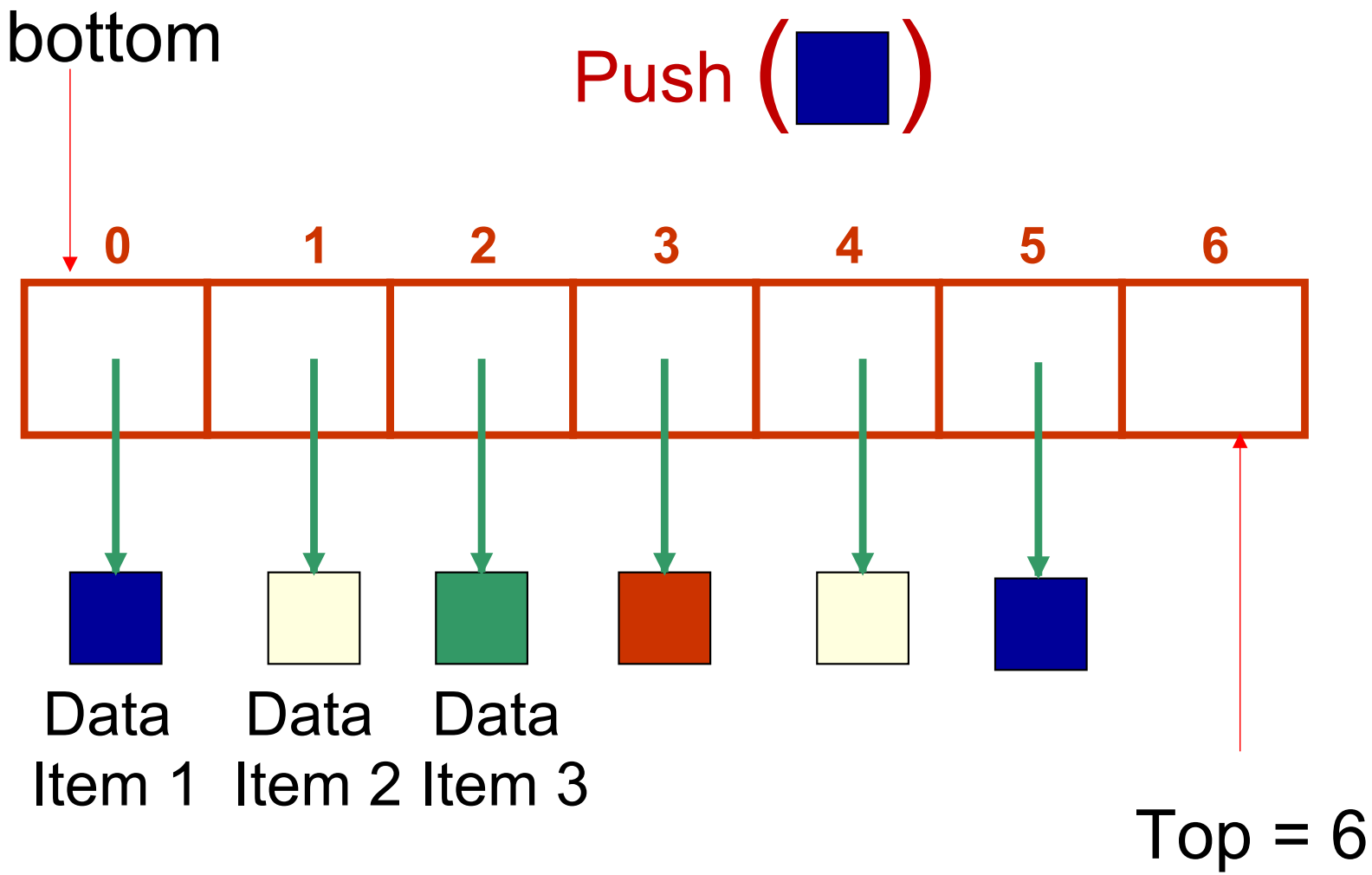


Array Implementation of a Stack



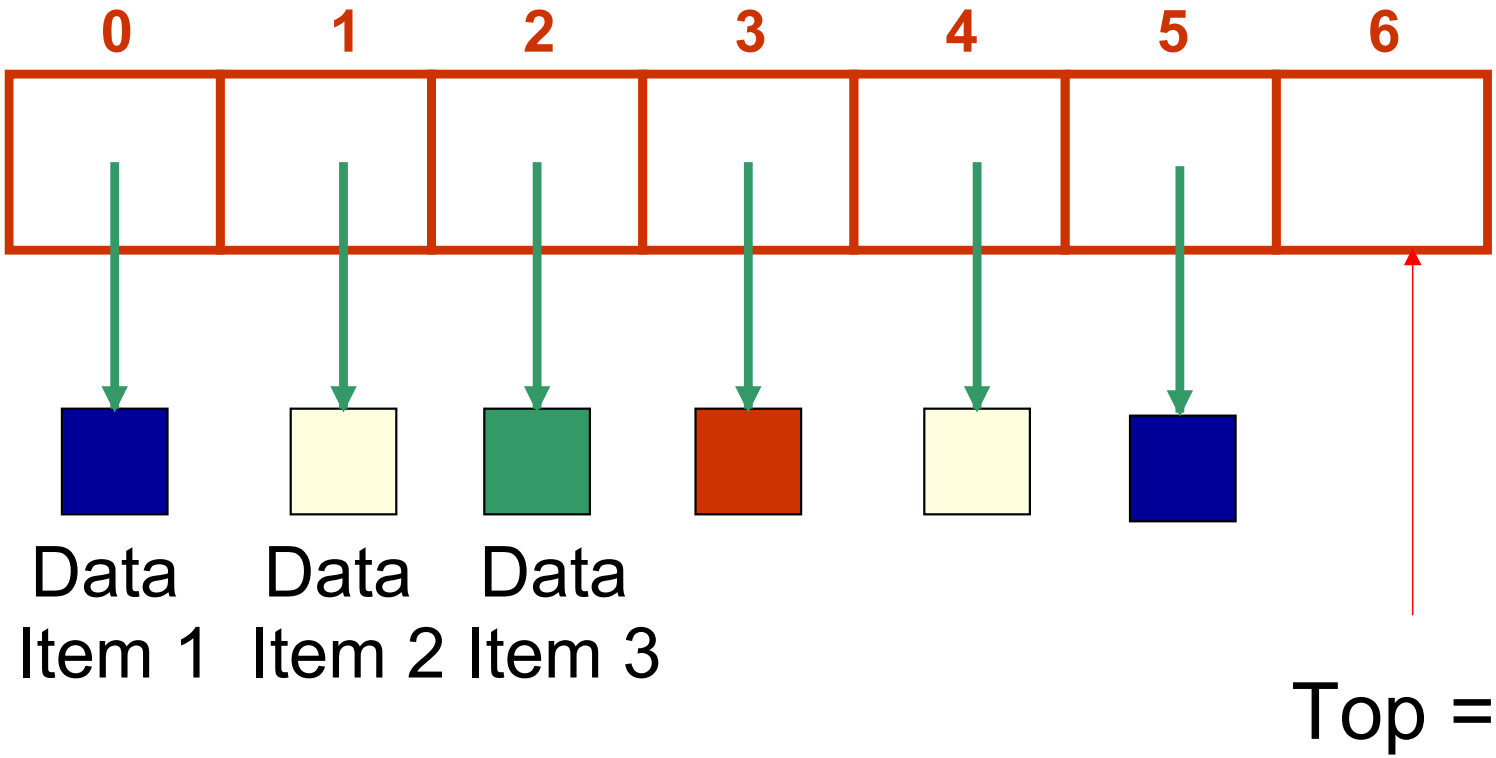
top 1. Calculating the number of items in stack
2. Show where's the next element adding.
(top - 1) is the place where the next element adding.

Array Implementation of a Stack



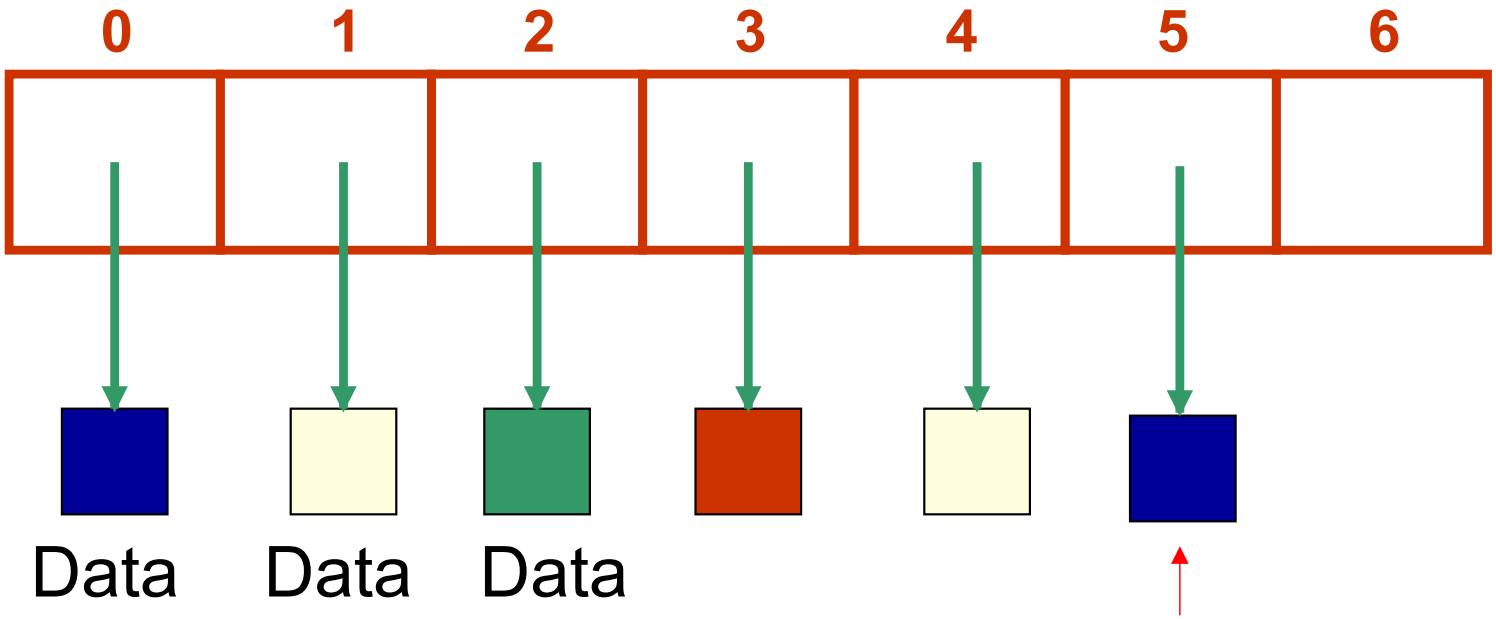
Array Implementation of a Stack

Pop ()



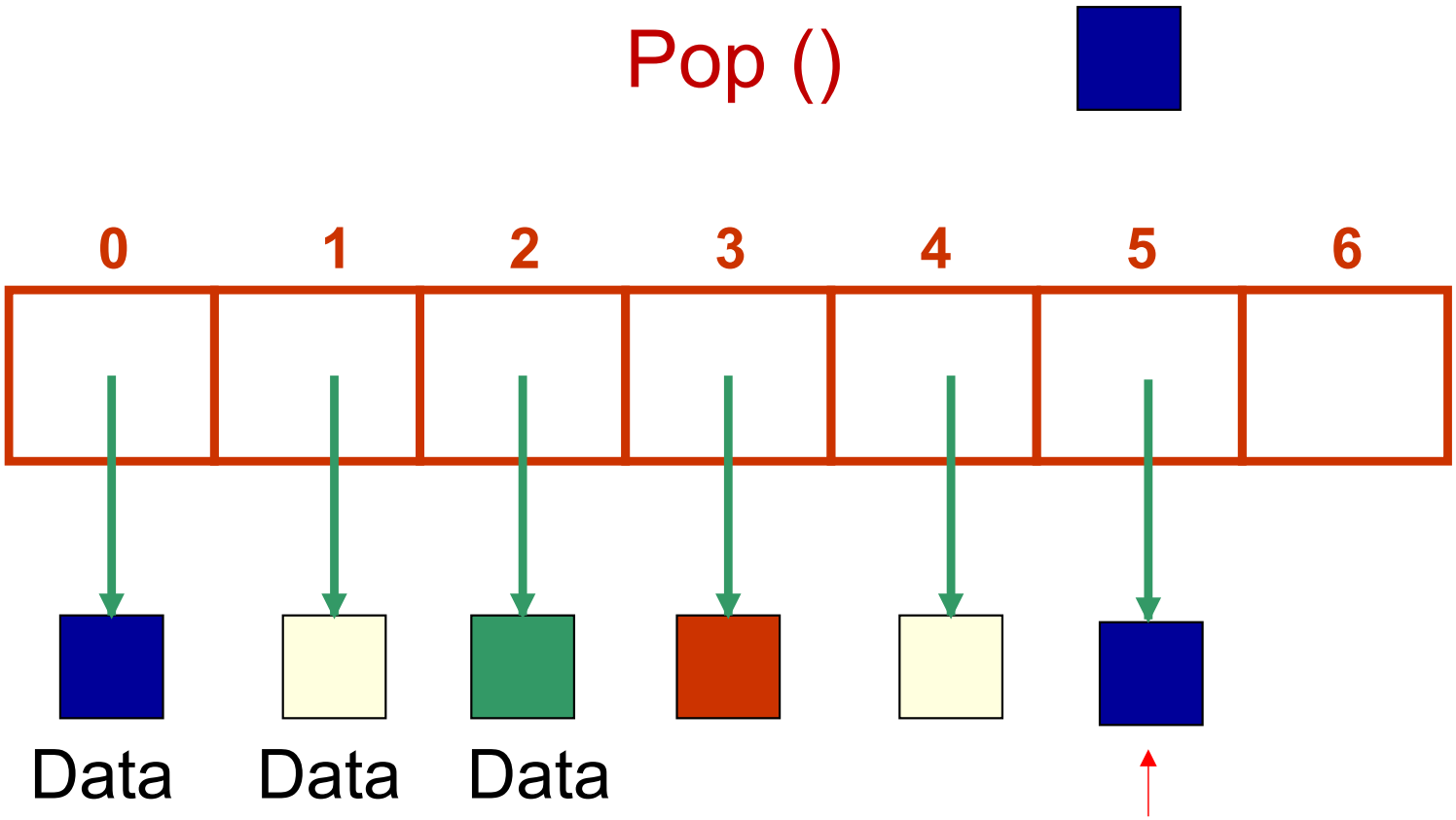
Array Implementation of a Stack

Pop ()



Array Implementation of a Stack

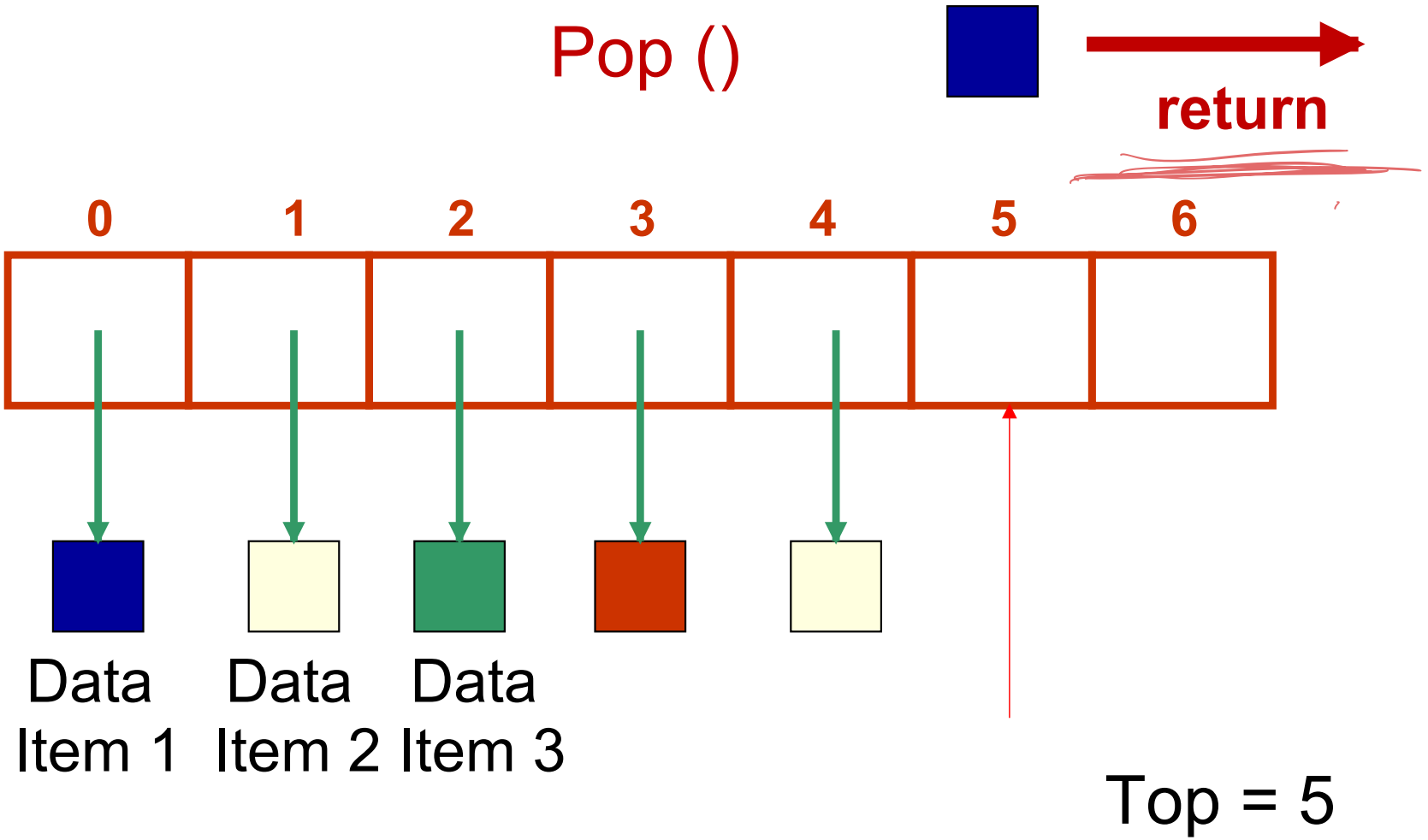
Pop ()



Top = 5

position that removing from

Array Implementation of a Stack

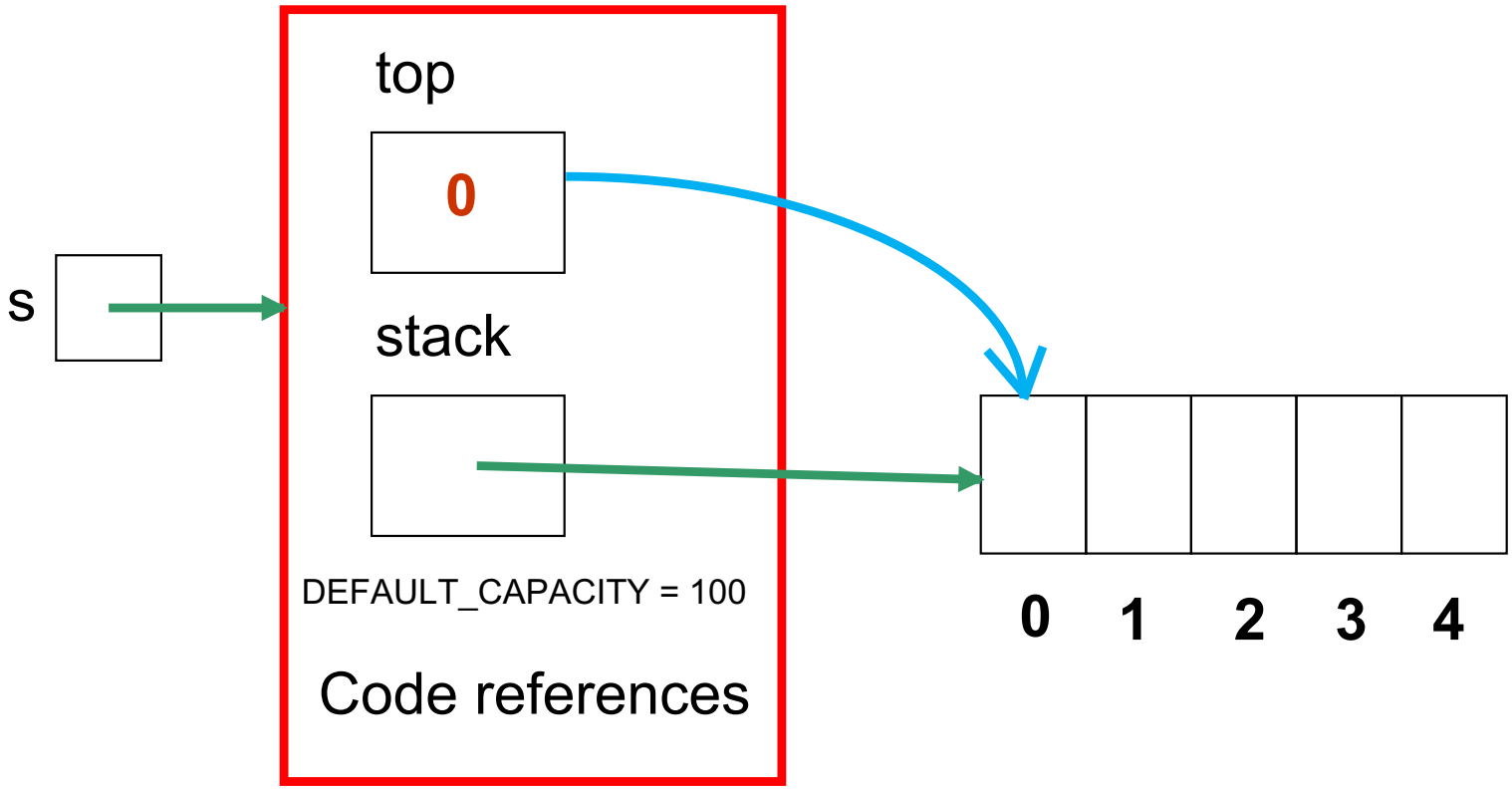


```
public interface StackADT<T> {
  //  Adds one element to the top of this stack
  public void push (T dataItem);
  //  Removes and returns the top element of this stack
  public T pop( );
  //  Returns the top element of this stack
  public T peek( );
  //  Returns true if this stack is empty
  public boolean isEmpty( );
  //  Returns the number of elements in this stack
  public int size( );
  //  Returns a string representation of this stack
  public String toString( );
}
```

```
public class ArrayStack<T> implements
   StackADT<T> {

   private T[ ] stack;   // Array for the data
    private int top;        // Top of stack
   private final int DEFAULT_CAPACITY=100;

   public ArrayStack( ) {
      top = 0;
      stack = (T[ ]) (new Object[DEFAULT_CAPACITY]);
   }

   public ArrayStack (int initialCapacity) {
      top = 0;
      stack = (T[ ]) (new Object[initialCapacity]);
   }
```

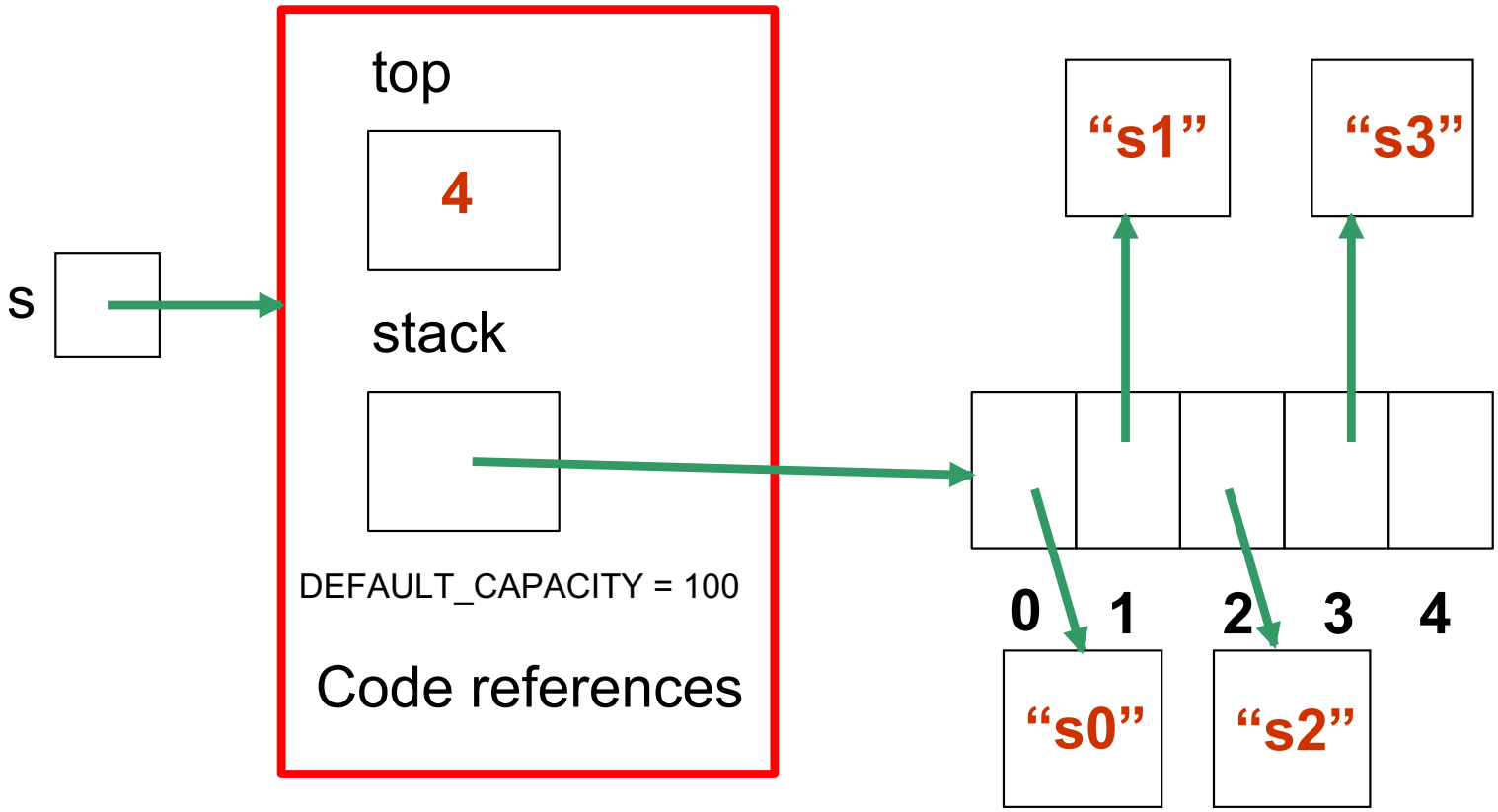
Example of using Constructor to create a Stack of Strings

```
ArrayStack<String> s =
        new ArrayStack<String>(5);
```



Example: the same **ArrayStack** object after four items have been pushed on

```
ArrayStack<String> s =
        new ArrayStack<String>(5);
```



```
//-----------------------------------------------------------
//  Adds the specified element to the top of the stack,
//  expanding the capacity of the stack array if necessary
//-----------------------------------------------------------
public void push (T dataItem) {
   if (top == stack.length)
      expandCapacity( );

   stack[top] = dataItem;
   top++;
}
```

the number of elements in the stack.

the stack is full.

create a new array with larger capacity and transfer the element

add element to the top.

increment the top by 1.

```
//  Helper method to create a new array to store the
// contents of the stack, with twice the capacity
private void expandCapacity( ) {
   T[ ] larger = (T[ ]) (new Object[stack.length*2]);

   for (int index=0; index < stack.length; index++)
      larger[index] = stack[index];

   stack = larger;
}
```

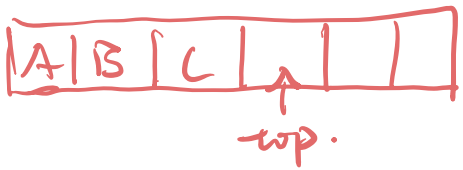
normally it is not allowed to operate the stack from bottom.

```
// Removes the element at the top of the stack and returns a
// reference to it. Throws an EmptyCollectionException if the
// stack is empty.
public T pop( ) throws EmptyCollectionException {
   if (top == 0)
      throw new EmptyCollectionException("Empty stack" );
   top--;
   T topItem = stack[top];
   stack[top] = null;
   return topItem;
}
```

=> move top to the last element position

pop out the top element.

```
// Returns the element at the top of the stack. Throws an
// EmptyCollectionException if the stack is empty.
public T peek( ) throws EmptyCollectionException {
   if (top == 0)
      throw new EmptyCollectionException("Empty stack" );
   return stack[top-1];
}
```



```
// Returns the number of elements in the stack
public int size( ) {
   return top;
}

// Returns true if the stack is empty and false otherwise
public boolean isEmpty( ) {
   return (top == 0);
}
```

```
//-----------------------------------------------------------
//  Returns a string representation of this stack.
//-----------------------------------------------------------
  public String toString( ) {
     String result = "Stack:\n";

     for (int index=0; index < top; index++)
        result = result + stack[index].toString( )
                        + "\n";

     return result;
  }
}
```

Stack:

looping stack is allowed only in expendCapacity() and toString() methods.

Another Stack Implementation

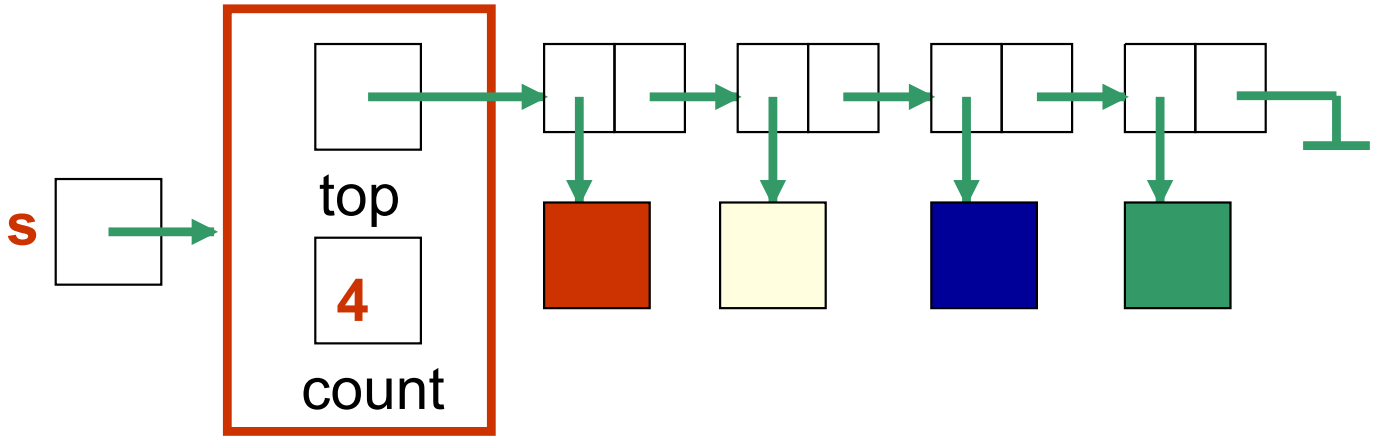
- We will now explore a *linked list implementation* of the Stack ADT
  - The data items of the stack are stored in the nodes of a linked list
- This linked list implementation will implement the same interface (Stack ADT) as the array-based implementation; only the underlying data structure changes.

Linked Implementation of a Stack

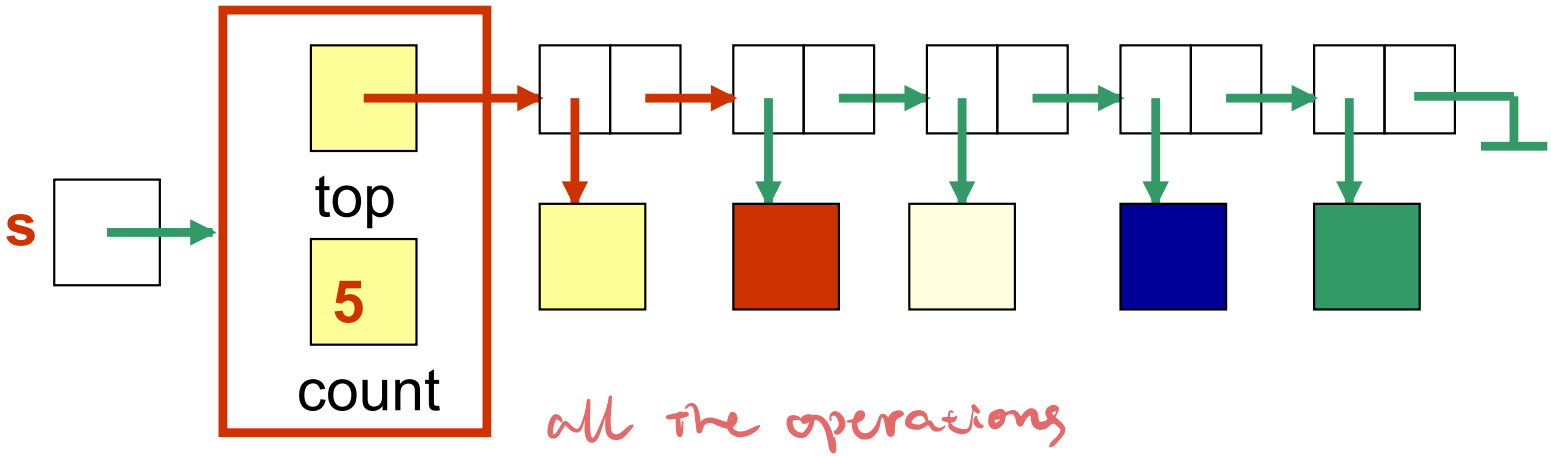
- Recall that we need a container to hold the data items and a variable to indicate the top of the stack.
- Our *container* will be a *linked list* of *nodes*, with each node containing a data item.
- The *top of the stack* will be the *first node* of the linked list.
  - So, a reference to the *first node* of the linked list (*top*) is also the reference to the whole linked list
- We will also keep track of the number of elements in the stack (*count*)

Linked Implementation of a Stack

A stack s with 4 elements



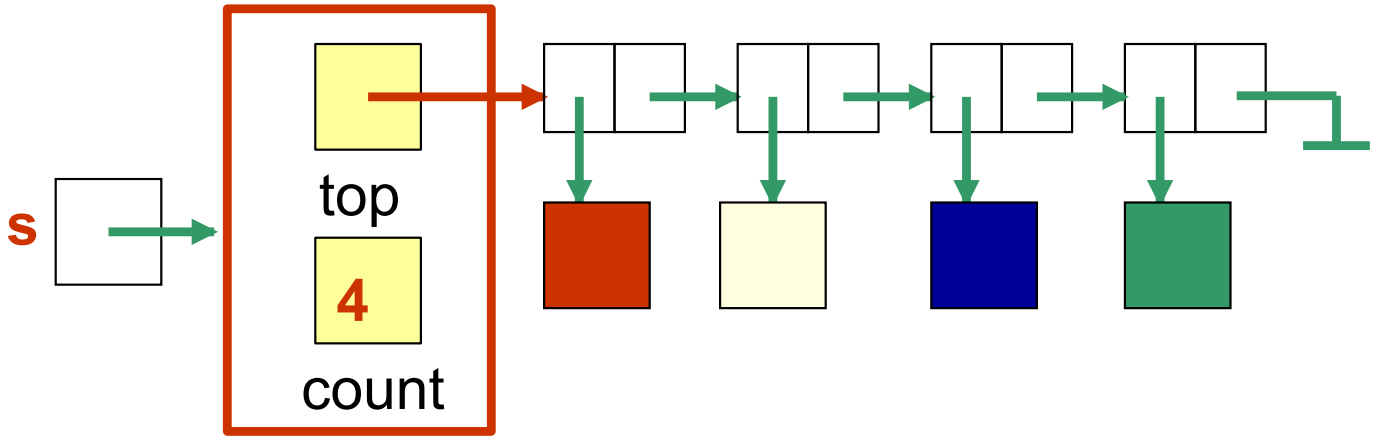
After pushing a fifth element



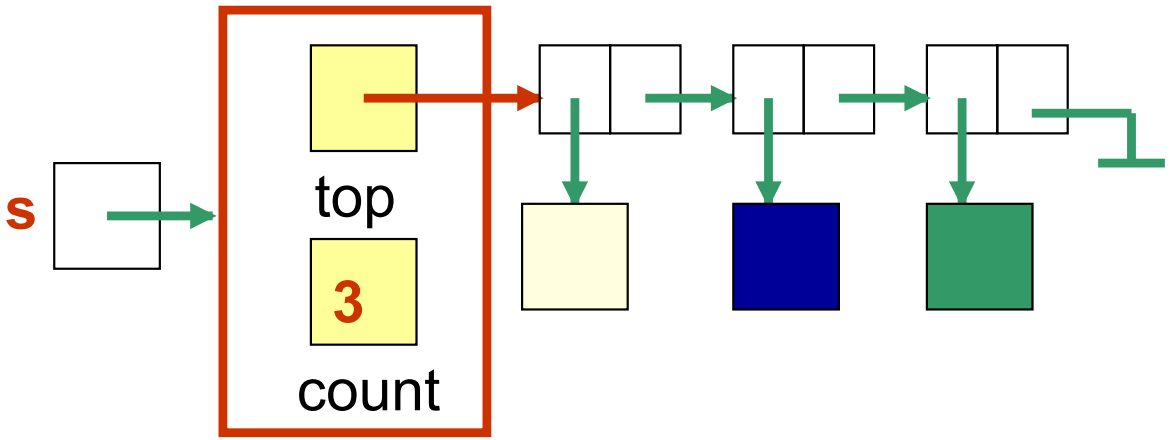
all the operations could only taken from the top.

Linked Implementation of a Stack

After popping an element



After popping another element



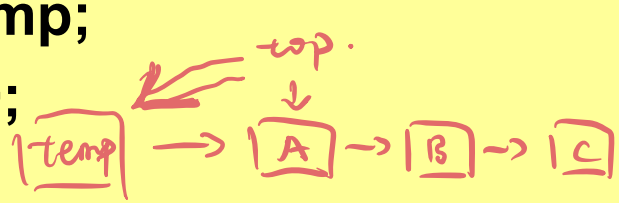
The LinkedStack Class

- Note that this class is called "LinkedStack.java" only to differentiate it s from the array implementation "ArrayStack.java"
- The nodes in the linked list are represented by the LinearNode class.
- The attributes (instance variables) are:
  - top: a reference to the first node (i.e. a reference to the linked list)
    - So it is of type LinearNode<T>
  - count: a count of the current number of data items in the stack

```
//-----------------------------------------------------------
//  Creates an empty stack.
//-----------------------------------------------------------
public LinkedStack ()
{
    top = null;
    count = 0;
}
```

The LinkedStack constructor

```
//-----------------------------------------------------------
//  Adds the specified element to the top of the stack.
//-----------------------------------------------------------
public void push (T element)
{
   LinearNode<T> temp = new LinearNode<T> (element);

   temp.setNext(top);
   top = temp;
   count++;
}
```

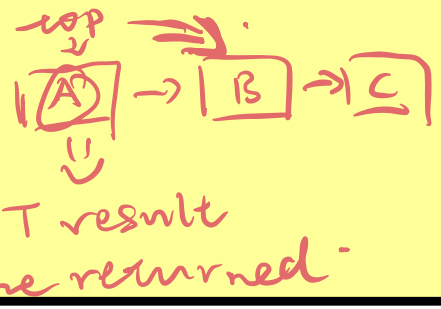


The push( ) operation

Where in the linked list is the element added?

top

```
//-----------------------------------------------------------
//  Removes the element at the top of the stack and returns
//  a reference to it. Throws an EmptyCollectionException if
//  the stack is empty.
//-----------------------------------------------------------
public T pop( ) throws EmptyCollectionException
{
   if (isEmpty( ))
      throw new EmptyCollectionException("Stack" );
   T result = top.getElement( );
   top = top.getNext( );
   count--;
   return result;
}
```



The pop( ) operation

From where in the linked list is the element removed?

From the top.

The Other Operations

- **Write the code for the methods**
  - ***peek***
  - ***isEmpty***
  - ***size***
  - ***toString***

```
if (isEmpty()) { ... }.
return top.getElement()

return (top == null);

return count.

String returnString = "Stack: \n";
Node temp = top.
While (temp.getNext() != null) {.
    returnString = returnString + temp.toString().
                   + "\n"
    temp = temp.getNext().
return returnString;
```

Discussion

- **Can the stack be empty?** Yes.
- **Can the stack be full?** Array-based => Yes.
- **How does this linked list implementation compare to the array implementation?** LinkedList-based => No

No capacity, the room is added dynamicly.

In the array case we have capacity and have to consider the case the stack is full.

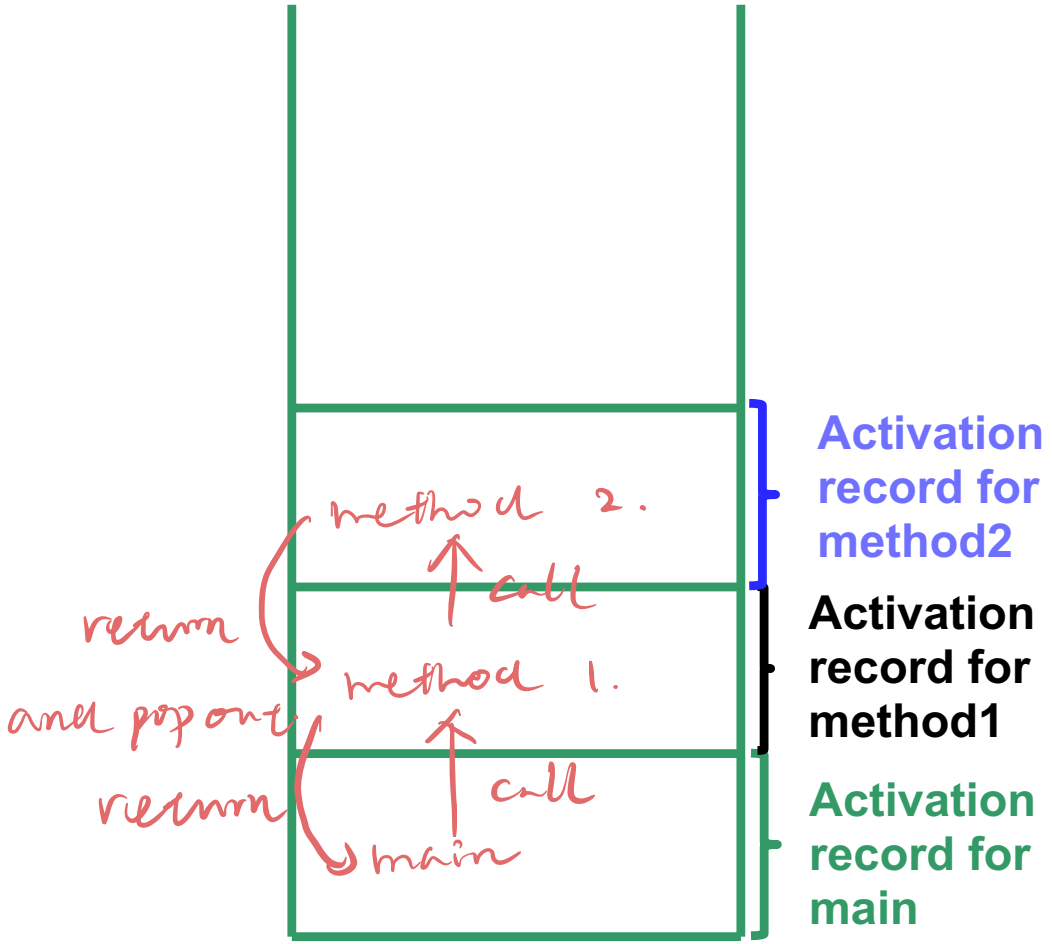
Uses of Stacks in Computing

Stacks are fundamental structures in Computer Science

- ***Execution stack (runtime or call stack)***
  - **Used by runtime system when methods are invoked**
  - **Holds "activation records" (or "frames" or "call frames") containing local variables, parameters, return address, etc.**

Execution Stack

```
public static void main (String[]
args) {

method1();

}

private void method1() {

method2(x);

}

private void method2(int x) {

}
```

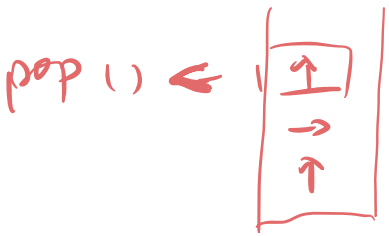


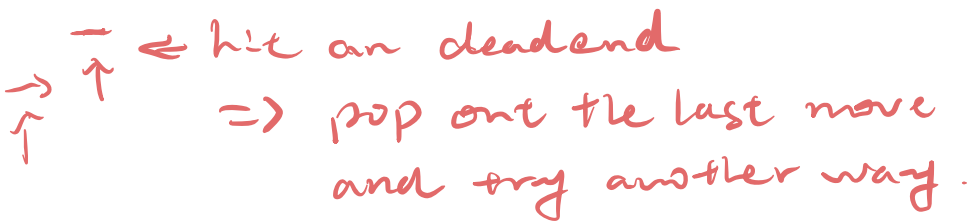
Execution stack

Uses of Stacks in Computing

Useful for any kind of problem involving *LIFO* data

- ***Backtracking*: in solving a maze or finding a path in a map**





Uses of Stacks in Computing

- *Word processors or editors*
  - To check expressions or strings of text for matching parentheses / brackets
    e.g.
```
if (a == b) {
        c = ((d + e) – f) * (d + e);
    }
```

Uses of Stacks in Computing

- *Word processors or editors*

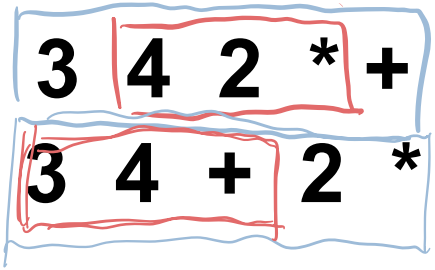
  To implement ***undo*** operations

  - Keeps track of the most recent operations

Ctrl + z

Using a Stack: Postfix Expressions

- Normally, we write expressions using *infix notation*:
  - Operators are between operands: 3 + 4 * 2
  - Parentheses force precedence: (3 + 4) * 2
- In a *postfix expression*, the operator comes *after* its two operands
  - Examples above would be written as:



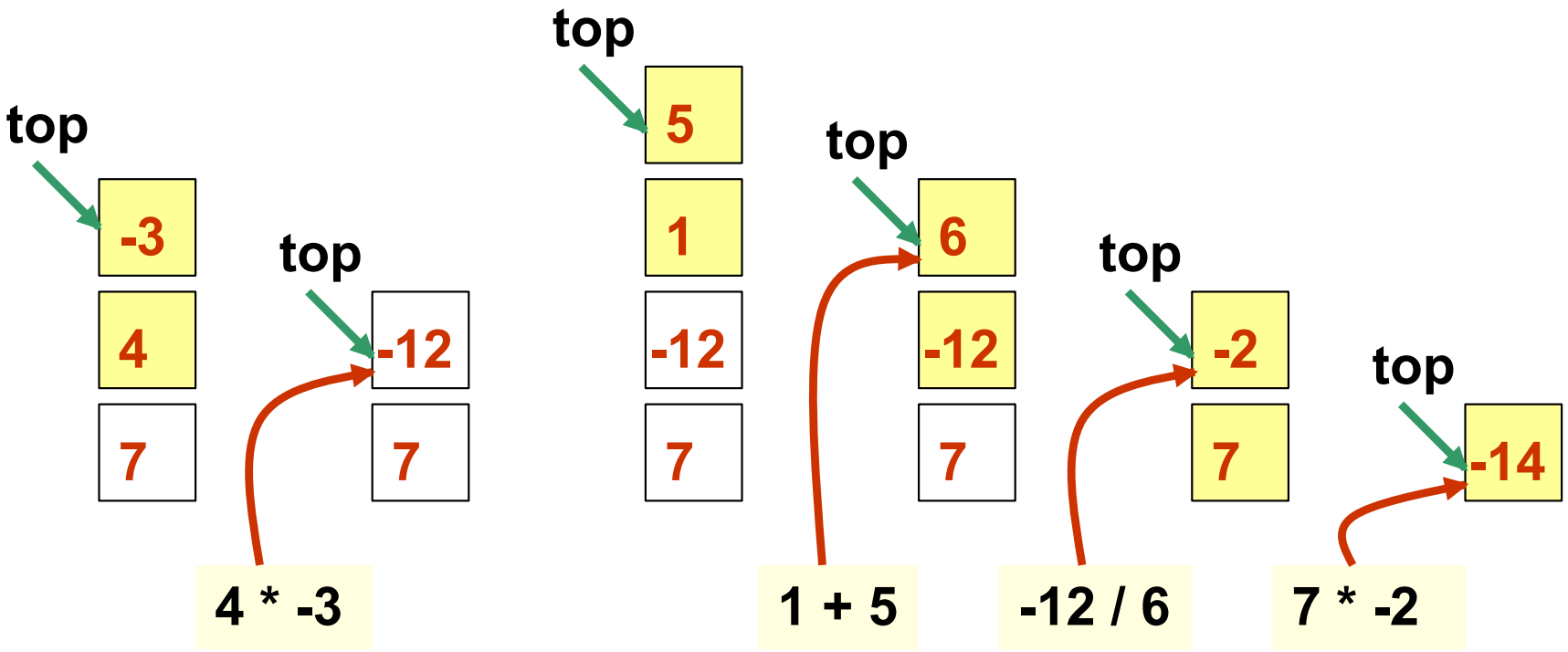
Evaluating Postfix Expressions

- *Algorithm to evaluate a postfix expression*:
  - Scan from left to right, determining if the next token is an operator or operand
  - If it is an operand, push it on the stack
  - If it is an operator, pop the stack twice to get the two operands, perform the operation, and push the result back onto the stack
- Try the algorithm on our examples …
- At the end, there will be one value in the stack – what is it?

Using a Stack to Evaluate a Postfix Expression

Evaluation of

7 4 -3 * 1 5 + / *



At end of evaluation, the result is the only item on the stack

```
for (int i = 0, i < list.length(), i++) {
    if (list[i].isNum()) stack.push(list[i])
    else { int a = stack.pop()
           int b = stack.pop()
           result = operation(a,b,list[i])
           stack.push(result)
    }
```