COSC 222 Data Structure

Stack ADT

Stack

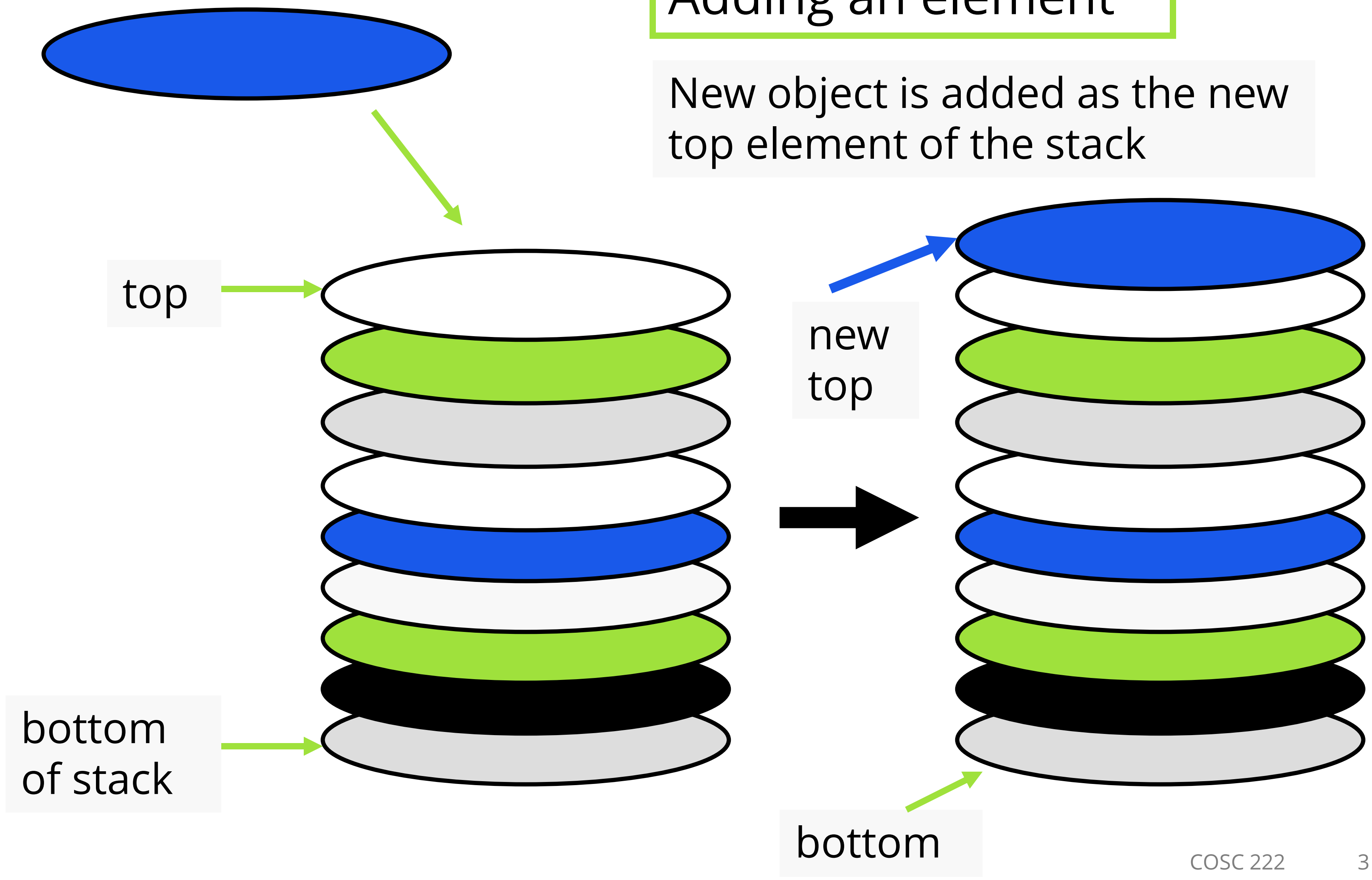
- **Stack :** a collection whose elements are added and removed from one end, called the **top** of the stack
- An element can be examined only **at one end** (the **top**).
- Stack is a **LIFO** (last in, first out) data structure
- Queue is a **FIFO** (first in, first out) data structure

- **Examples**:
  - A stack of plates

Conceptual View of a Stack

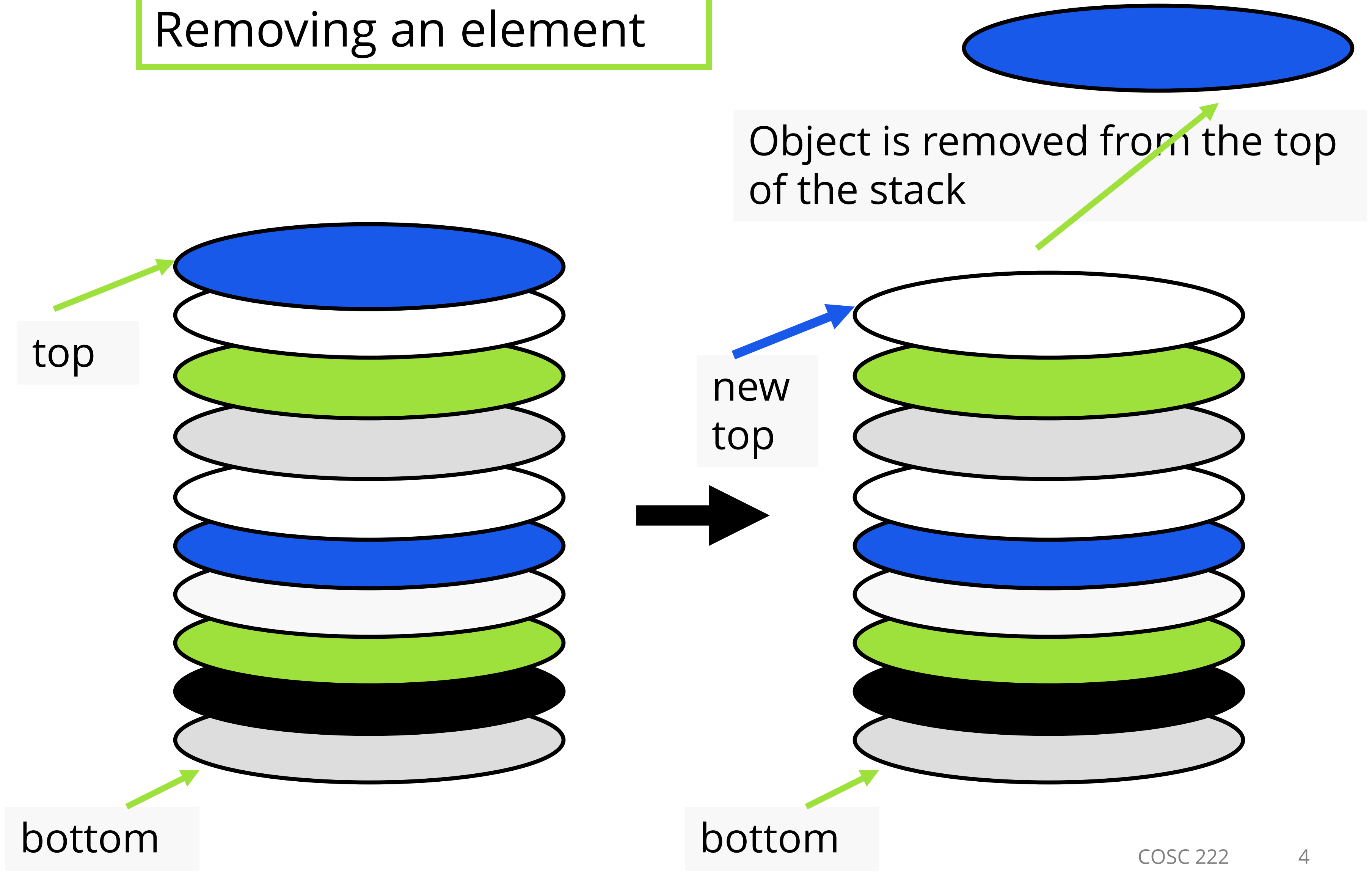
Adding an element

New object is added as the new top element of the stack



Conceptual View of a Stack

Removing an element



Uses of Stacks in Computing

- Useful for any kind of problem involving **LIFO** data
- **Browsers**
  - Back button keeps track of pages visited in a browser tab
- **Word Processors, editors**
  - To check expressions for matching parentheses / brackets
    e.g.
```
if (a == b)
          { c = (d + e) * f;
          }
```
- **Stack Calculators**
  - e.g., To convert an **infix** expression to **postfix**

```
Infix expression:       a * b + c
Postfix expression:     a b * c
```

Operations on a Collection

- Every collection has a set of operations that define how we interact with it, for example:
  - **Add** elements
  - **Remove** elements
  - Determine if the collection **is empty**
  - Determine the collection's **size**
  - ...
  - ...

Stack Operations

Operation	Description
push	Adds an element to the top of the stack
pop	Removes an element from the top of the stack
peek	Examines the element at the top of the stack.
isEmpty	Determines whether the stack is empty
size	Determines the number of elements in the stack

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:

3 4 2 * +

3 4 + 2 *

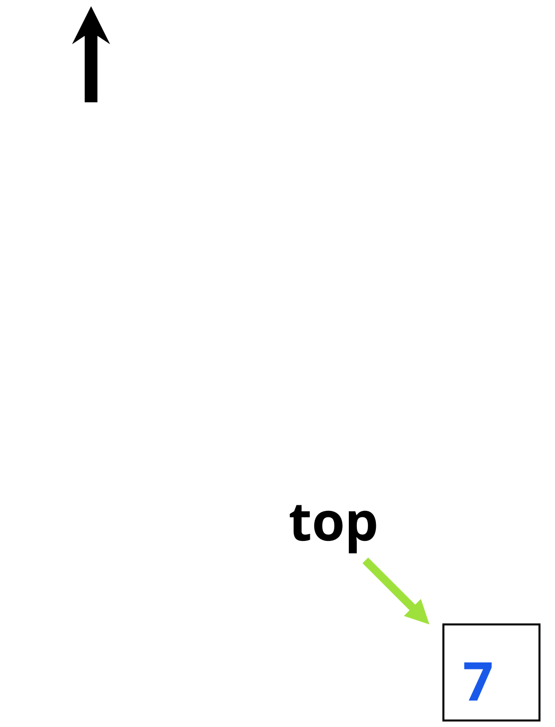
Using a Stack to Evaluate a Postfix Expression

```
stack of numbers S;
for each “token” in the expression
      // (a token is number or operator)
      if it is a number
            Push it onto S
      else (it is an operator)
            Pop S into B
            Pop S into A
            Push the result of A operator B onto S
Pop S to get the final result
```

Using a Stack to Evaluate a Postfix Expression

Evaluation of

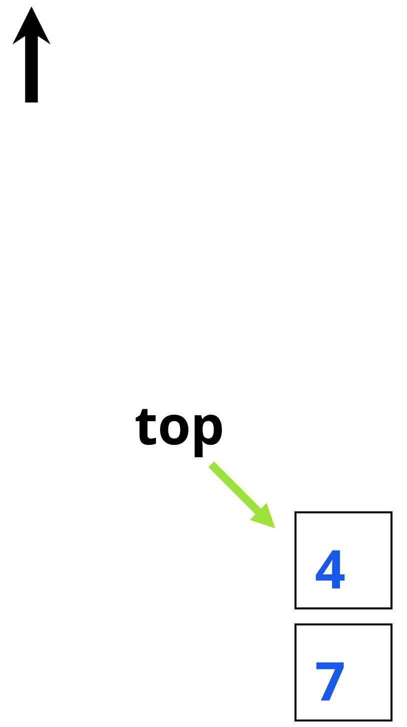
7 4 -3 * 1 5 + / *



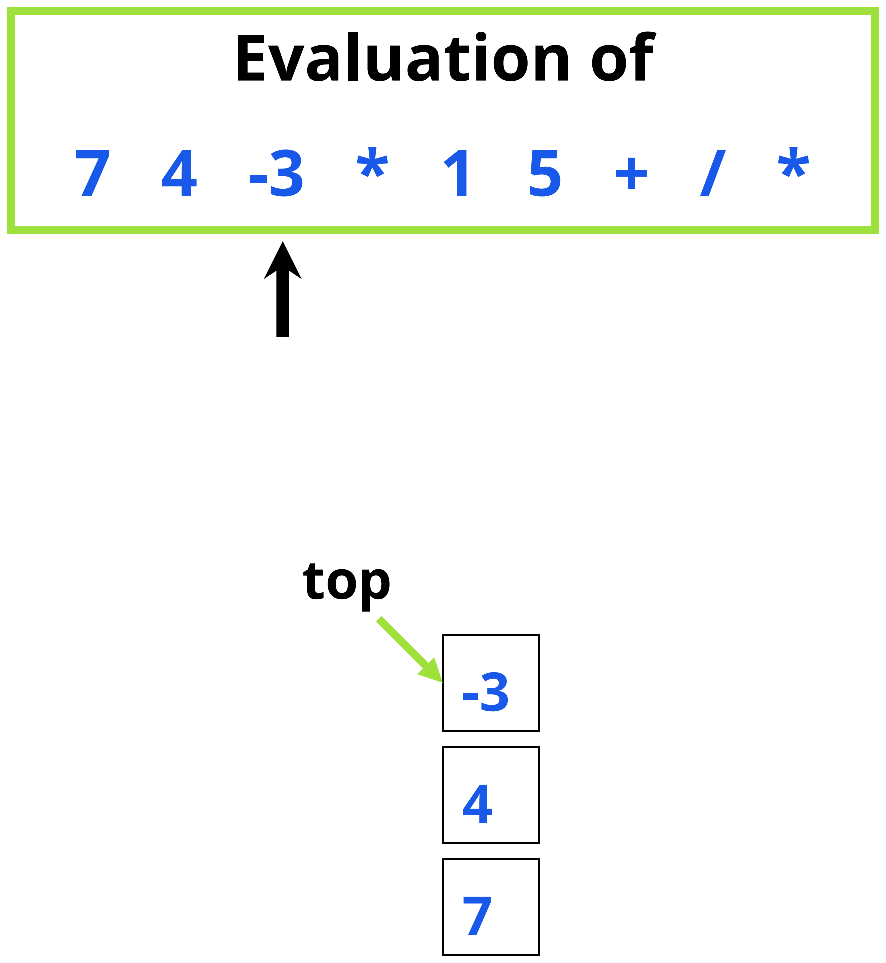
Using a Stack to Evaluate a Postfix Expression

Evaluation of

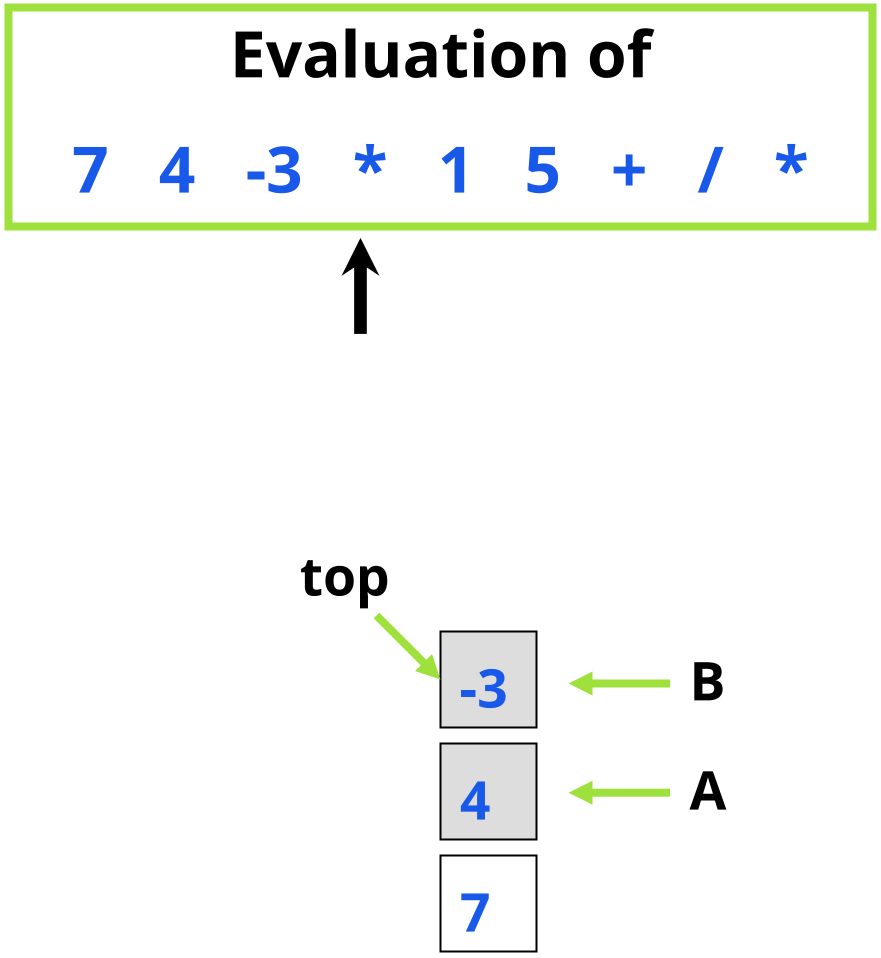
7 4 -3 * 1 5 + / *



Using a Stack to Evaluate a Postfix Expression



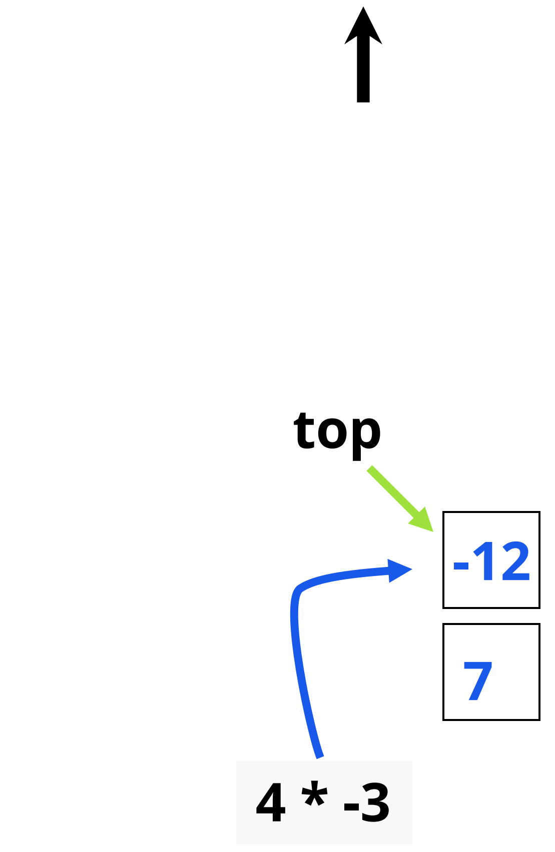
Using a Stack to Evaluate a Postfix Expression



Using a Stack to Evaluate a Postfix Expression

Evaluation of

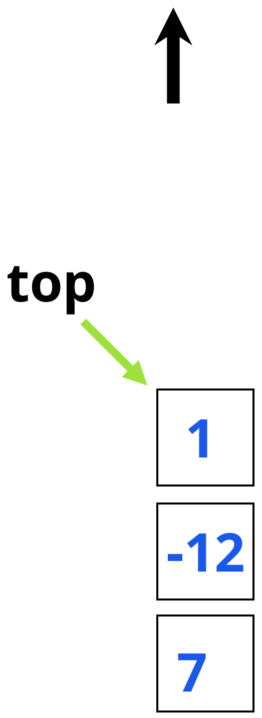
7 4 -3 * 1 5 + / *



Using a Stack to Evaluate a Postfix Expression

Evaluation of

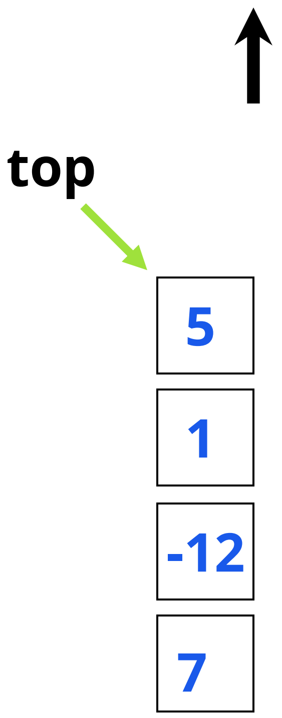
7 4 -3 * 1 5 + / *



Using a Stack to Evaluate a Postfix Expression

Evaluation of

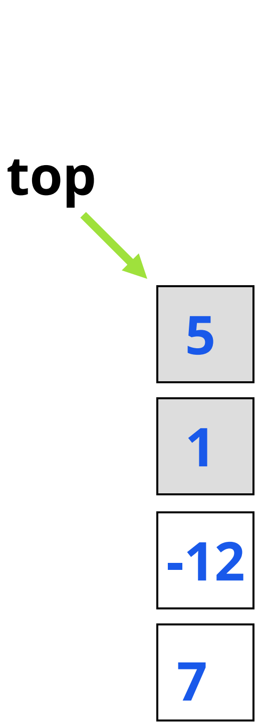
7 4 -3 * 1 5 + / *



Using a Stack to Evaluate a Postfix Expression

Evaluation of

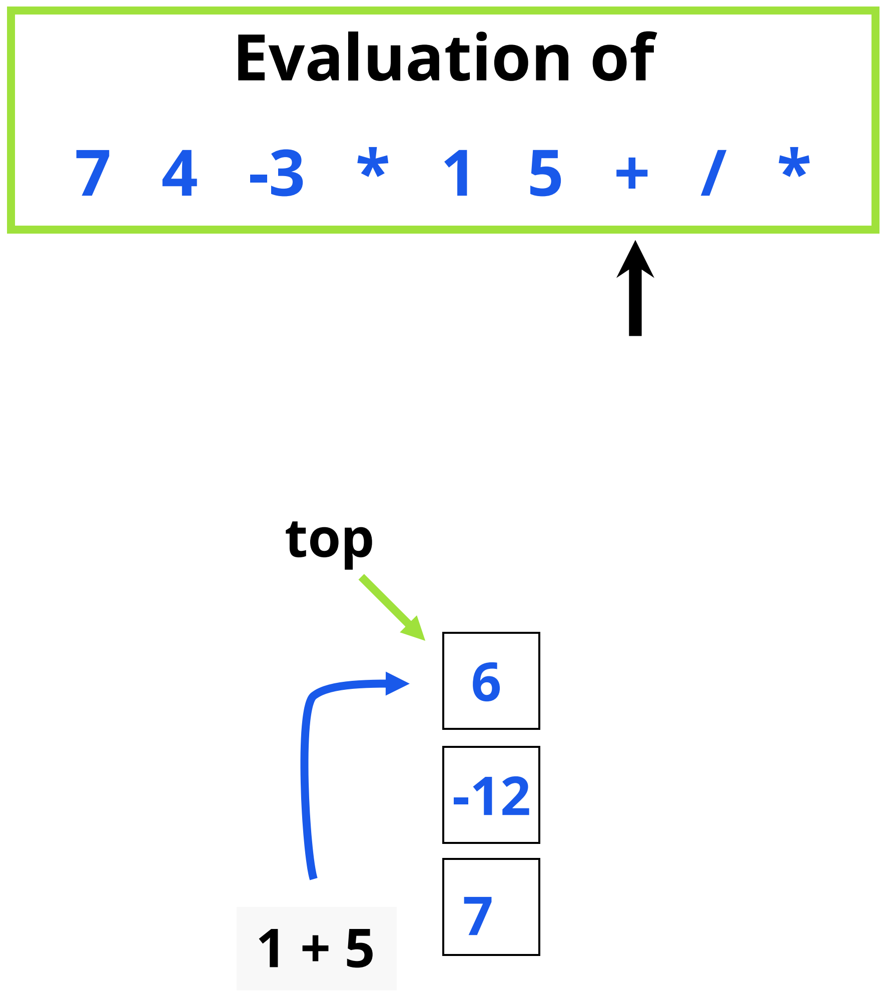
7 4 -3 * 1 5 + / *



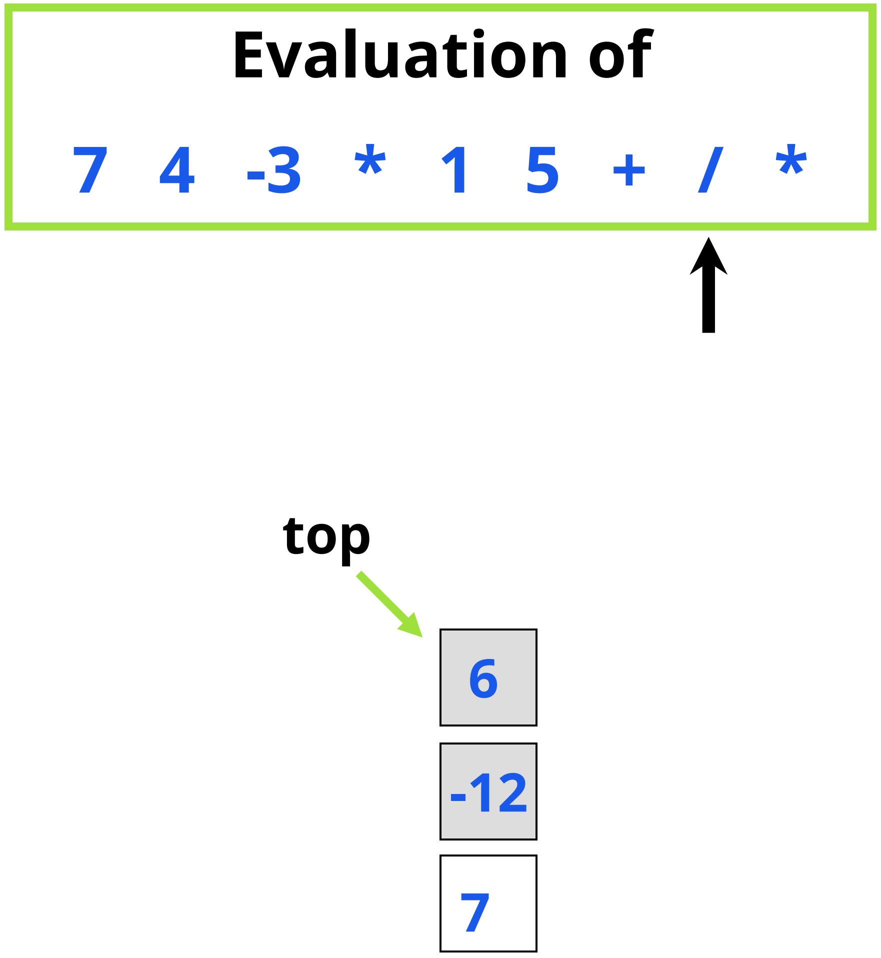
Using a Stack to Evaluate a Postfix Expression

Evaluation of

7 4 -3 * 1 5 + / *



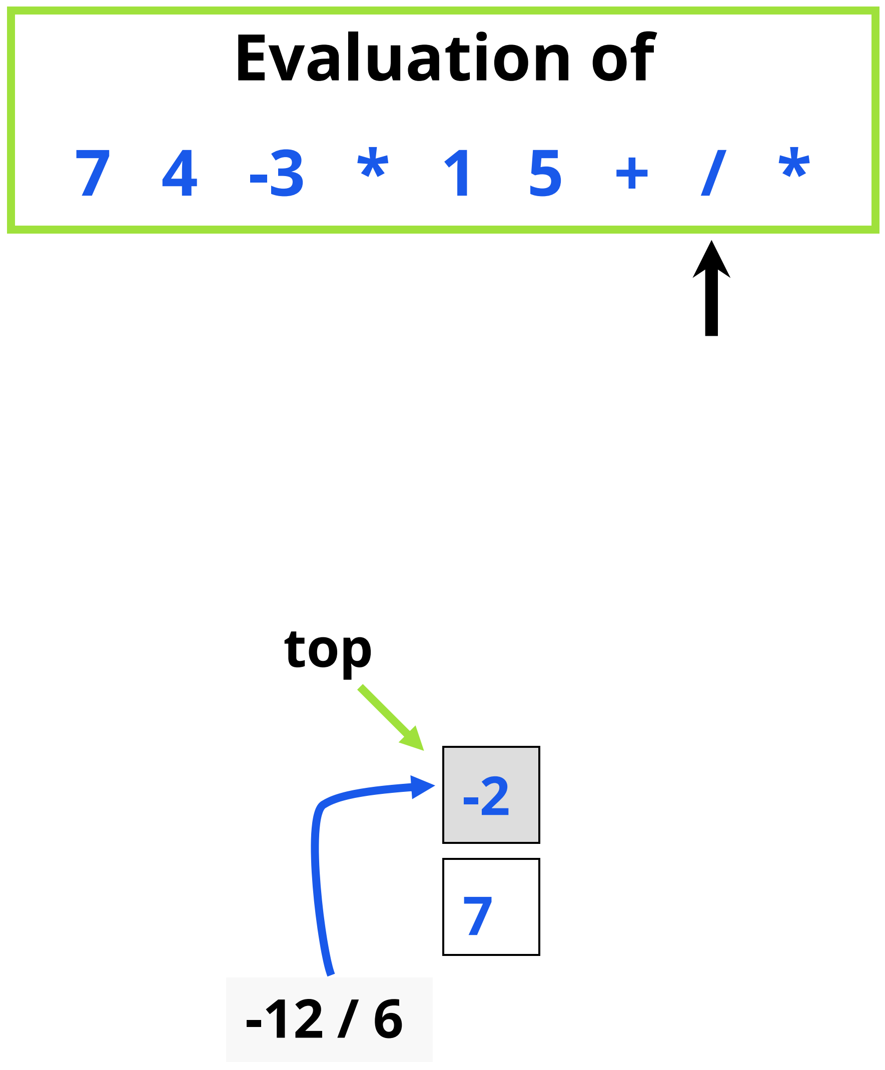
Using a Stack to Evaluate a Postfix Expression



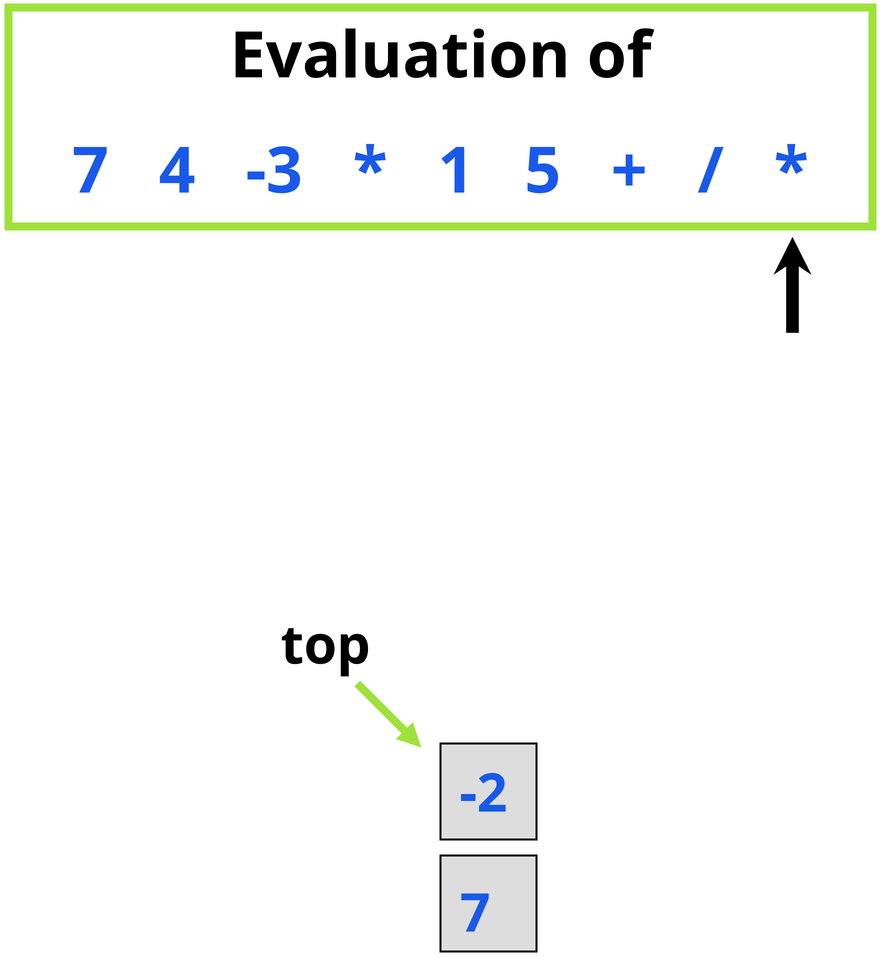
Using a Stack to Evaluate a Postfix Expression

Evaluation of

7 4 -3 * 1 5 + / *



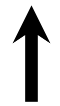
Using a Stack to Evaluate a Postfix Expression

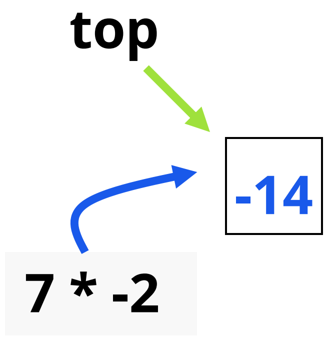


Using a Stack to Evaluate a Postfix Expression

Evaluation of

7 4 -3 * 1 5 + / *



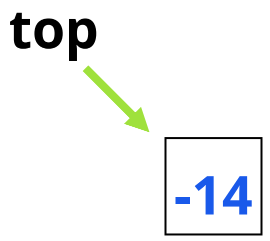


Using a Stack to Evaluate a Postfix Expression

Evaluation of

7 4 -3 * 1 5 + / *

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



Syntax Checking

- Check for matching parenthesis ( ), braces { }, brackets [ ], etc.
- Sweep through code
  - If we see an opening symbol, push onto stack
  - If we see a closing symbol, match it with the top of stack and pop

```
public void add ( int idx, AnyType x ) {for ( int
i=theSize; i > idx; i-- ) theItems [ idx ] = x;}
```

Syntax Checking

```
public void add ( int idx, AnyType x ) {for ( int
i=theSize; i > idx; i-- ) theItems [ idx ] = x;}
```

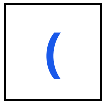
Syntax Checking

```
public void add ( int idx, AnyType x ) {for ( int
i=theSize; i > idx; i-- ) theItems [ idx ] = x;}
```

$$\boxed{(}$$

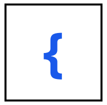
Syntax Checking

```
public void add ( int idx, AnyType x ) {for ( int
i=theSize; i > idx; i-- ) theItems [ idx ] = x;}
```



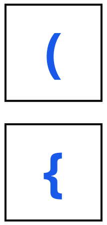
Syntax Checking

```
public void add ( int idx, AnyType x ) {for ( int
i=theSize; i > idx; i-- ) theItems [ idx ] = x;}
```



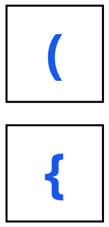
Syntax Checking

```
public void add ( int idx, AnyType x ) {for ( int
i=theSize; i > idx; i-- ) theItems [ idx ] = x;}
```



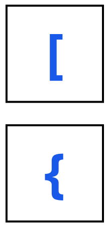
Syntax Checking

```
public void add ( int idx, AnyType x ) {for ( int
i=theSize; i > idx; i-- ) theItems [ idx ] = x;}
```



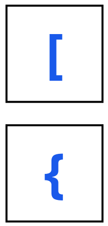
Syntax Checking

```
public void add ( int idx, AnyType x ) {for ( int
i=theSize; i > idx; i-- ) theItems [ idx ] = x;}
```



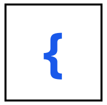
Syntax Checking

```
public void add ( int idx, AnyType x ) {for ( int
i=theSize; i > idx; i-- ) theItems [ idx ] = x;}
```



Syntax Checking

```
public void add ( int idx, AnyType x ) {for ( int
i=theSize; i > idx; i-- ) theItems [ idx ] = x;}
```

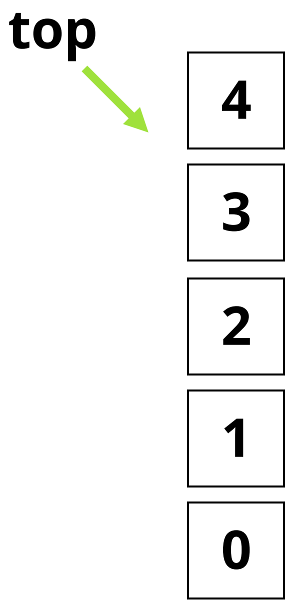


Syntax Checking

```
for each character L → R
       if opening ( or [ or {
              push onto stack
       else if closing ) or ] or }
              if ( (stack isn't empty)
                 and (it matches the top of the stack) )
                     pop the stack
              else
                     error --- mismatched
if stack not empty
       error --- missing closing ) or ] or }
```

The java.util.Stack Class

```
Stack s = new Stack();
// put stuff in stack
for(int i = 0; i < 5; i++)
      s.push( i );
// Examines the element at the top of the stack
System.out.println("Top item: " +s.peek());
// print out contents of stack while emptying it
int limit = s.size();
for(int i = 0; i < limit; i++)
      System.out.print( s.pop() + " ");
//or
//    while( !s.isEmpty() )
//          System.out.println( s.pop() );
```



Implementing a Stack

- Need an underlying collection to hold the elements of the stack
- 2 basic choices
  - **array** (native or ArrayList)
  - **linked list**

Java Interface for Stack ADT: StackADT.java

```
public interface StackADT<T>
{
   //  Adds one element to the top of this stack
   public void push (T element);
   //  Removes and returns the top element from this
 stack
   public T pop( );
   //  Returns without removing the top element of
 this stack
   public T peek( );
   //  Returns true if this stack contains no
 elements
   public boolean isEmpty( );
   //  Returns the number of elements in this stack
   public int size( );
   //  Returns a string representation of this stack
   public String toString( );
}
```

The ArrayStack Class: ArrayStack.java

- The class ArrayStack implements the StackADT interface:

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

- Attributes (instance variables):

```
private T[ ] stack;   // the container for the data
private int top;    // indicates the next open slot
```

- There is also a private constant

```
private final int DEFAULT_CAPACITY=100;
```

ArrayStack Constructors

```
//  Creates an empty stack using the default capacity.
public ArrayStack( )
{
   top = 0;
   stack = (T[ ]) (new Object[DEFAULT_CAPACITY]);
}


//  Creates an empty stack using the specified capacity.
public ArrayStack (int initialCapacity)
{
   top = 0;
   stack = (T[ ]) (new Object[initialCapacity]);
}
```

The push( ) operation

```
//  Adds the specified element to the top of the stack,
//  expanding the capacity of the stack array if
necessary

public void push (T element)
{
   if (top == stack.length)
       expandCapacity( );

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

expandCapacity()

```
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;
}
```

The pop( ) operation

```
// Removes the element at the top of the stack and returns a
// reference to it.
 public T pop( )
 {
    if (isEmpty( )) {
      System.out.println("Stack is Empty");
      return null;
    }
   top--;
   T result = stack[top];
   stack[top] = null;
   return result;
 }
```

The toString( ) operation

```
//  Returns a string representation of this stack.

public String toString( )
{
   String result = "";

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

   return result;
}
```

The size( ) operation

```
//  Returns the number of elements in the stack


public int size( )

{

   return top;

}
```

The isEmpty( ) operation

// Returns true if the stack is empty and false otherwise

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

The peek( ) operation

// Returns the top item

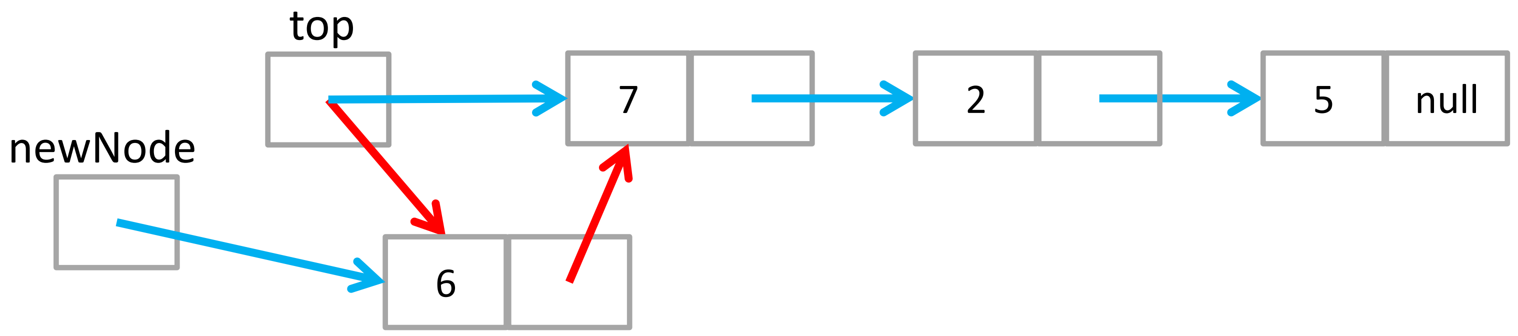
```
 public T peek( ){
    return stack[top-1];
}
```

Limitations

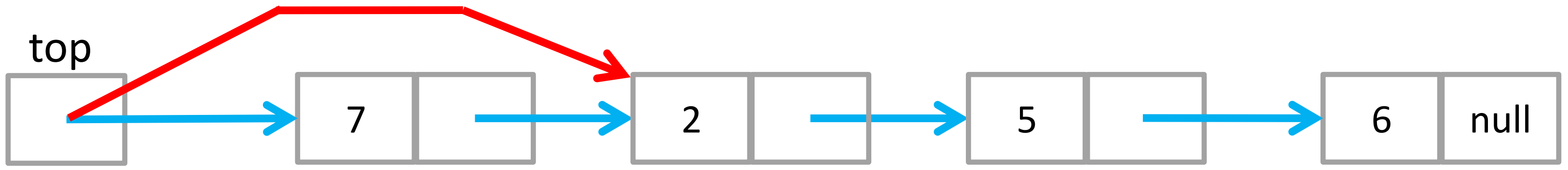
- Limitations
  - The maximum size of the stack must be defined a priori
  - Trying to push a new element into a full stack causes an implementation-specific exception

In Class Activities

- Stack with array
- Stack with Linked list
  - Check the Linked List implementation
  - push: Items were always added to the top



  - pop: To remove an item, only top pointer need to be changes



In Class Activities

- isEmpty: Check if the Top is null

- peek: Check the item that the Top is pointing

- size: Different ways to find size, you can simply use an additional counter to keep track of the size. Don't forget to update the counter when using push or pop

- toString: different ways to do that, a simple solution could be traverse the linked list –toString method that we used in the Linked list

Questions?