Stack Data Structures

261217 Data Structures for Computer Engineers

Patiwet Wuttisarnwattana, Ph.D.

patiwet@eng.cmu.ac.th

Computer Engineering, Chiang Mai University

Arrays versus Linked-lists

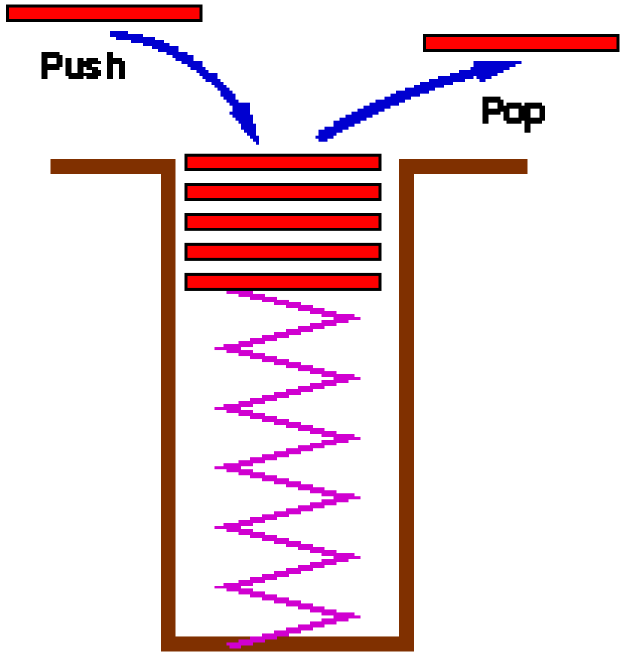
- Linear data structures
- Key Variables
  - Array: size, capacity
  - Linked-list: head
- Linear time {O(n)} to search for a key
  - **For loop** for array
    - Condition: for (int i=0; i<size; i++){ }
  - **While loop** for linked-list
    - Condition: while(node.next != null) {node.move_next;}
- Constant time {O(1)} to access the first and the last items
- How about running time to add/remove the first and last items?

Let’s apply Arrays and Linked-lists

- We will learn two more well-known data structures
  - Stacks
  - Queues
- These two data structures are *linear*
- So that means we can implement them using Arrays and Linked-lists
- At the end, you should be able to implement the new data structures from arrays and linked-lists

Stack as an ADT

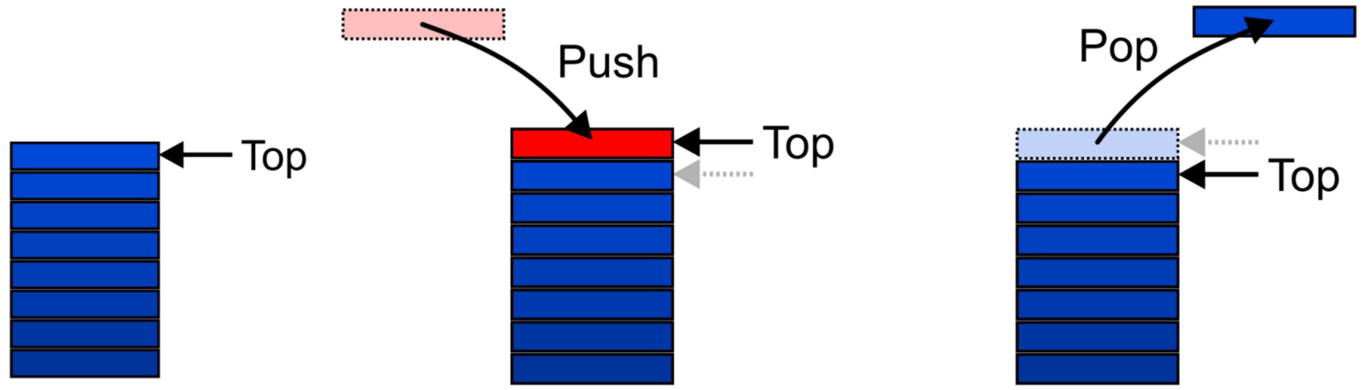
- A list for which Insert and Delete are allowed only at one end of the list (the *top*)
  - the implementation defines which end is the "top"
  - LIFO – Last in, First out
- *Push*: Insert element at the top
- *Pop*: Remove and return the top element (aka TopAndPop)
- *IsEmpty*: *test for emptiness*



a tray stack

Stack Data Structures

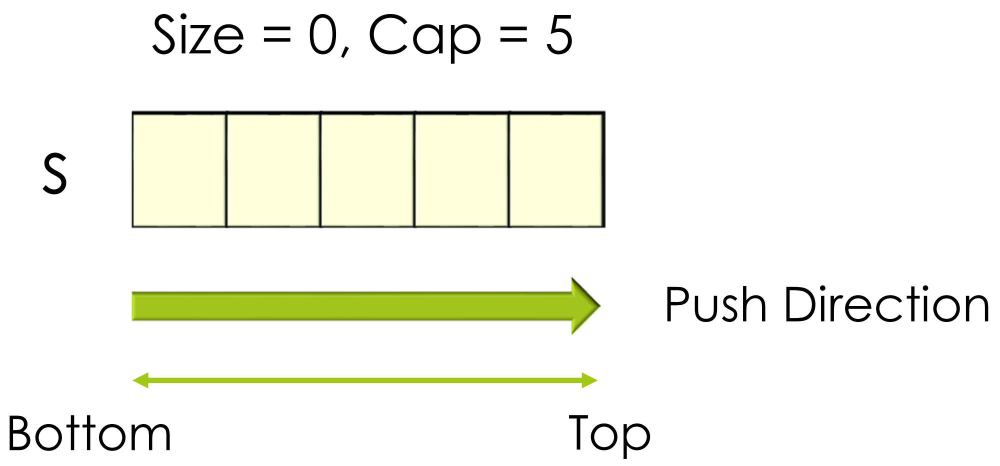
Last-in–first-out (LIFO)



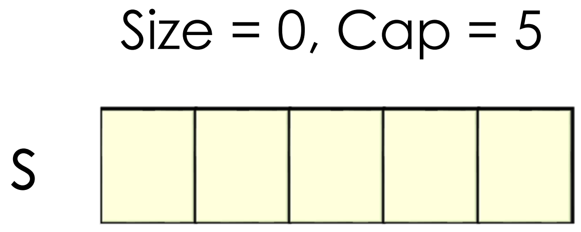
There are two exceptions associated with abstract stacks:

- It is an undefined operation to call either pop or top on an empty stack
- -> StackUnderflowException should be thrown
- If stack is full, StackOverflowException will be thrown

Stack implementation using Array



Stack implementation using Array



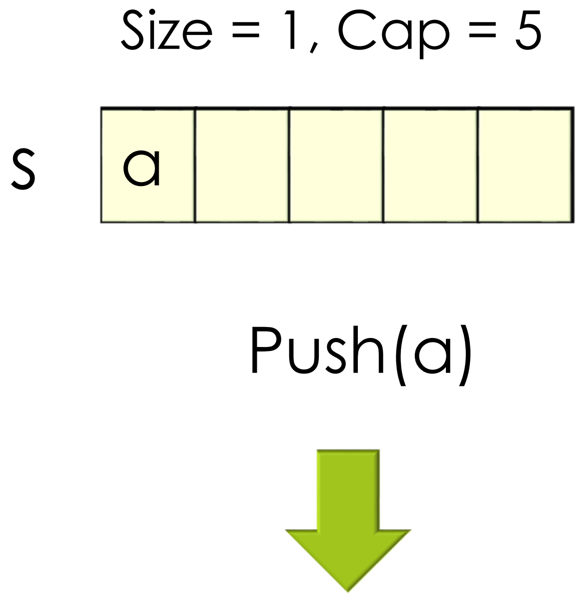
Push(a)

Array

s.pushBack(a)

What should be java code for this operation?

Stack implementation using Array



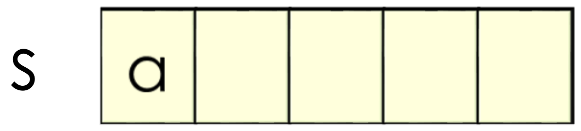
Array

s.pushBack(a)

What should be java code for this operation?

Stack implementation using Array

Size = 1, Cap = 5

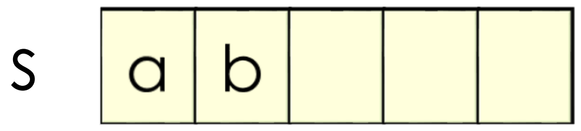


Push(b)

Array
s.pushBack(b)

Stack implementation using Array

Size = 2, Cap = 5

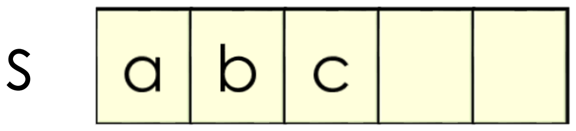


Push(b)

Array
s.pushBack(b)

Stack implementation using Array

Size = 3, Cap = 5

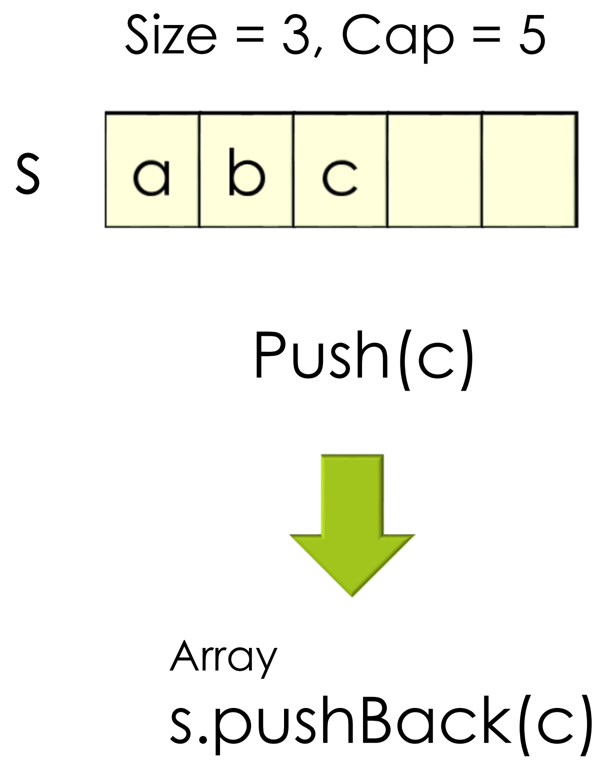


Push(c)

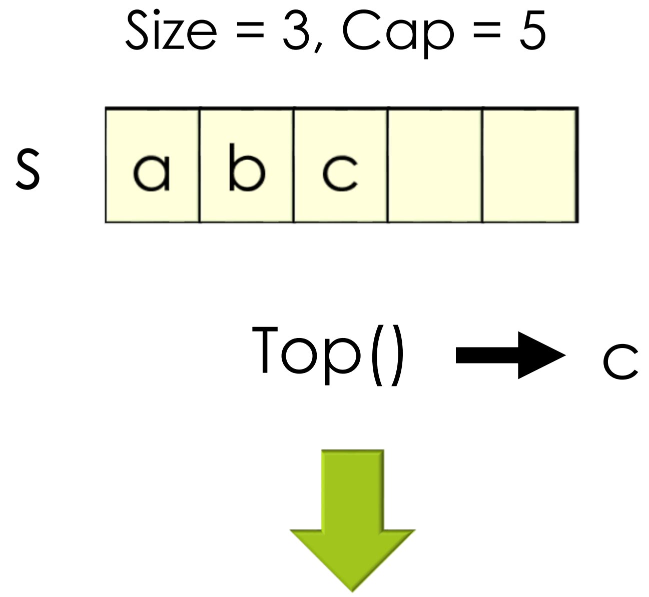
Array

s.pushBack(c)

Stack implementation using Array



Stack implementation using Array



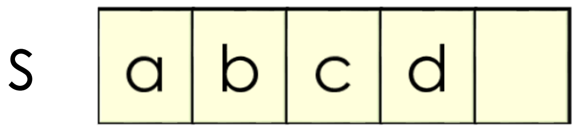
Array

s.topBack()

What should be java code for this operation?

Stack implementation using Array

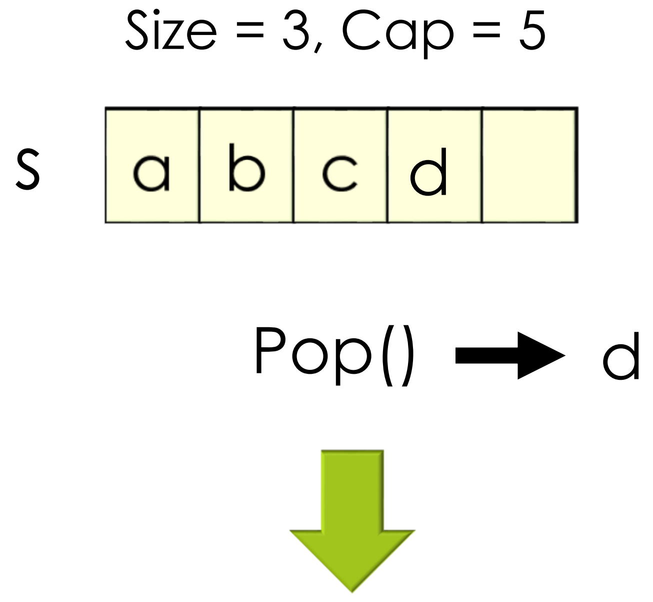
Size = 4, Cap = 5



Push(d)

Array
s.pushBack(d)

Stack implementation using Array



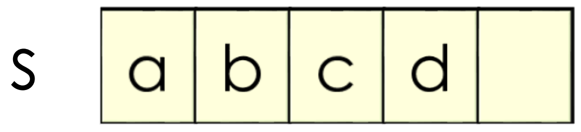
Array

s.popBack()

What should be java code for this operation?

Stack implementation using Array

Size = 4, Cap = 5



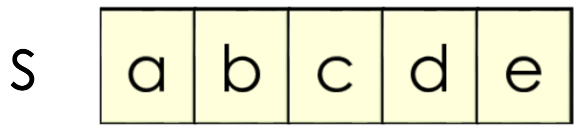
Push(d)

Array

s.pushBack(d)

Stack implementation using Array

Size = 5, Cap = 5



Push(e)

↓

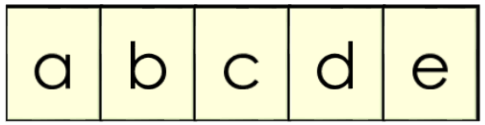
Array

s.pushBack(e)

Stack implementation using Array

Size = 5, Cap = 5

Static array **s**



Push(f)

StackOverflowException
[or other options]

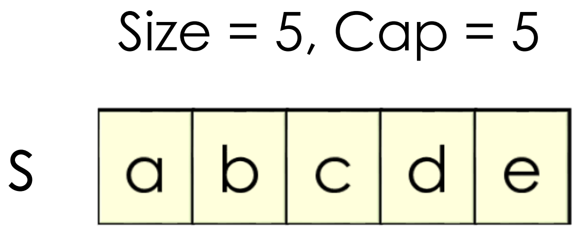
Exceptions

The case where the array is full is not an exception defined in the Stack ADT

If the array is full, we have five options:

- Increase the size of the array
- Throw an exception (StackOverflowException)
- Ignore the element being pushed
- Replace the current top of the stack
- Put the pushing process to “sleep” until something else removes the top of the stack

Stack implementation using Array

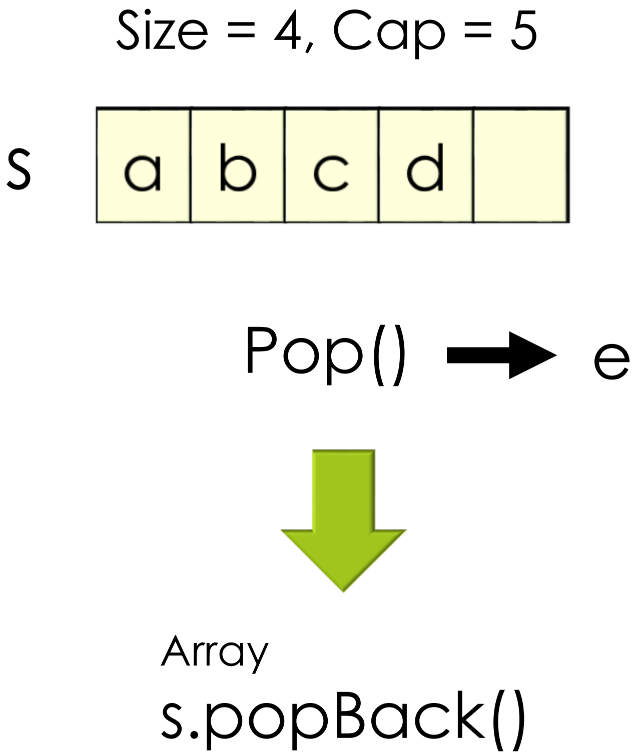


Pop()

Array

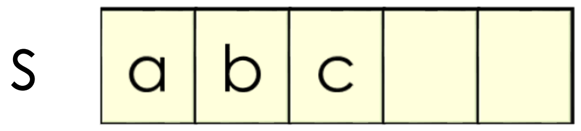
s.popBack()

Stack implementation using Array



Stack implementation using Array

Size = 3, Cap = 5



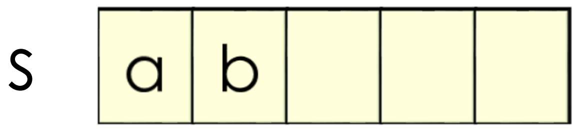
Pop() $\rightarrow$ d

Array

s.popBack()

Stack implementation using Array

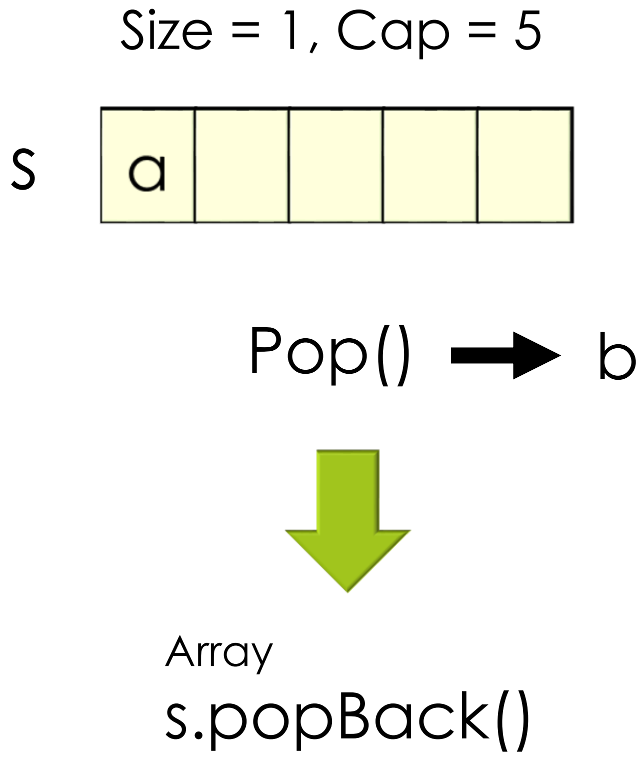
Size = 2, Cap = 5



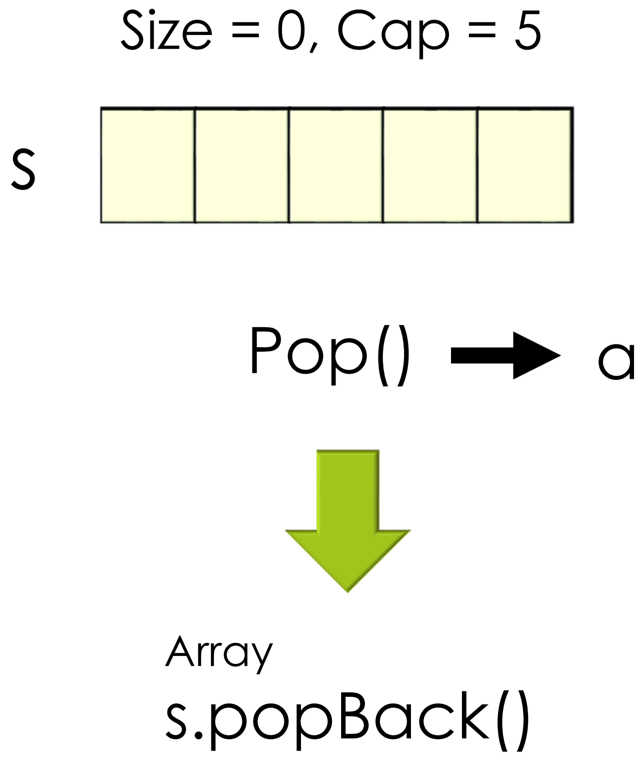
Pop() $\rightarrow$ c

Array
s.popBack()

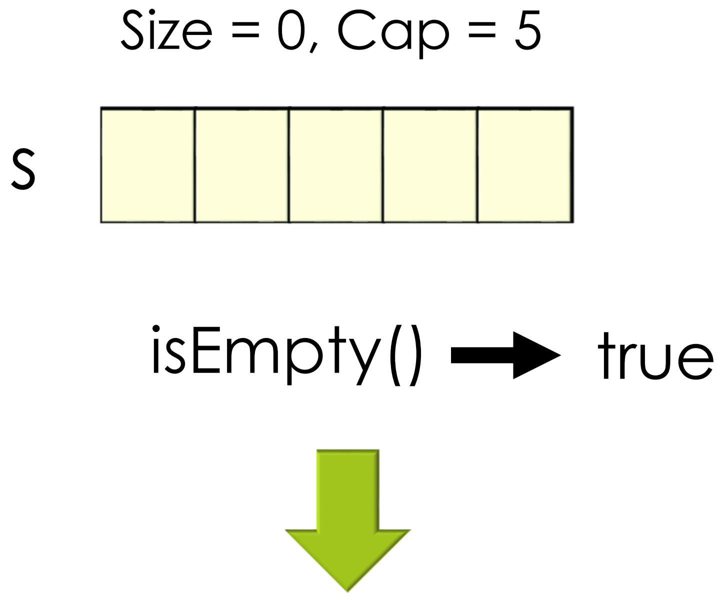
Stack implementation using Array



Stack implementation using Array



Stack implementation using Array



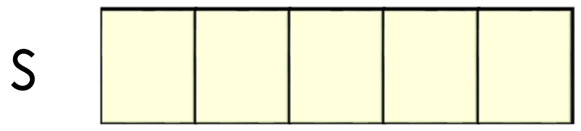
Array

s.isEmpty()

What should be java code for this?

Stack implementation using Array

Size = 0, Cap = 5

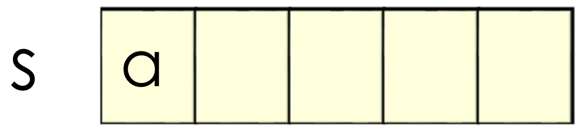


pop()

ERROR
StackUnderflowException

Stack implementation using Array

Size = 1, Cap = 5



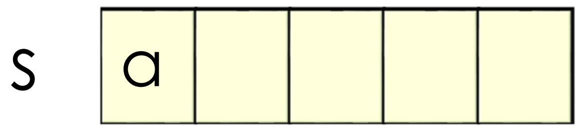
Push(a)

What is the Big O for adding an item to Stacks (implemented using Array)?

O(1)

Stack implementation using Array

Size = 1, Cap = 5

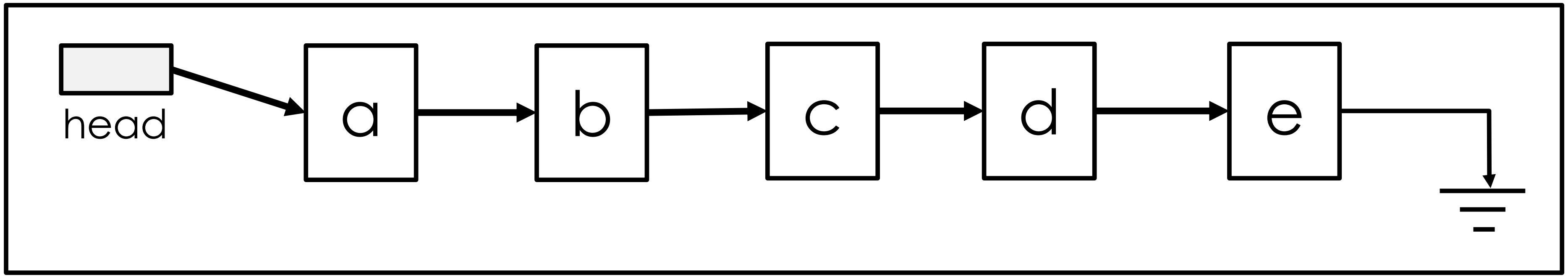


Pop() -> a

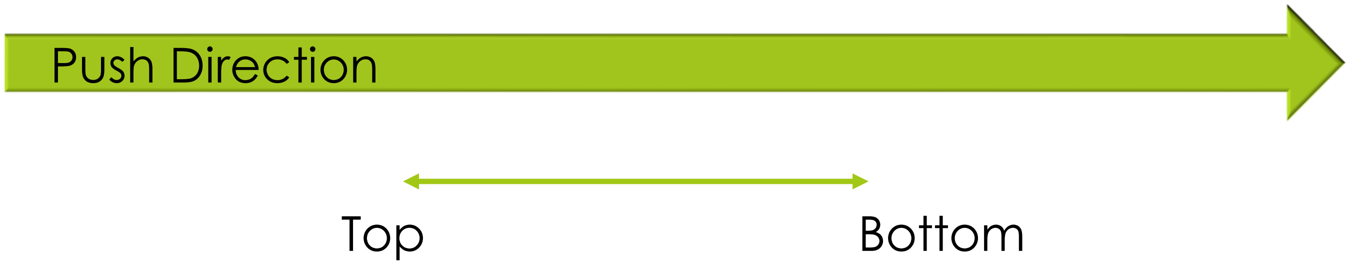
What is the Big O for removing an item from Stacks (implemented using Array)?

O(1)

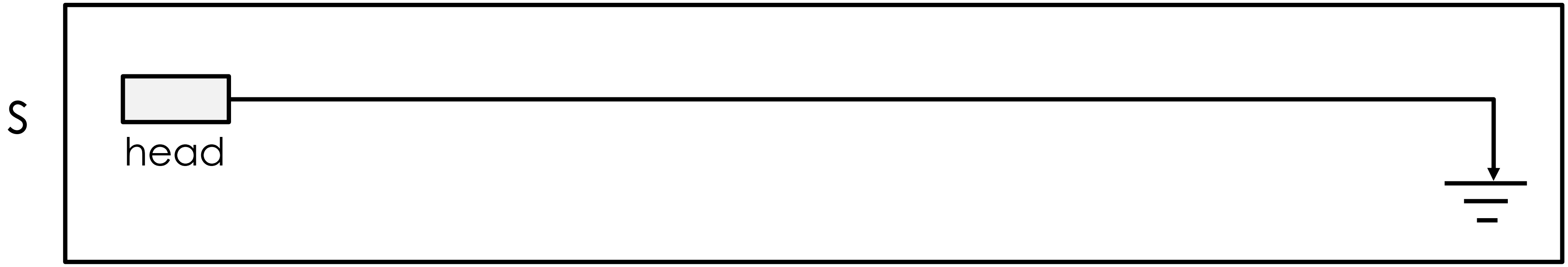
Stack implementation using Linked List



Which direction is the best for pushing an object into the stack?



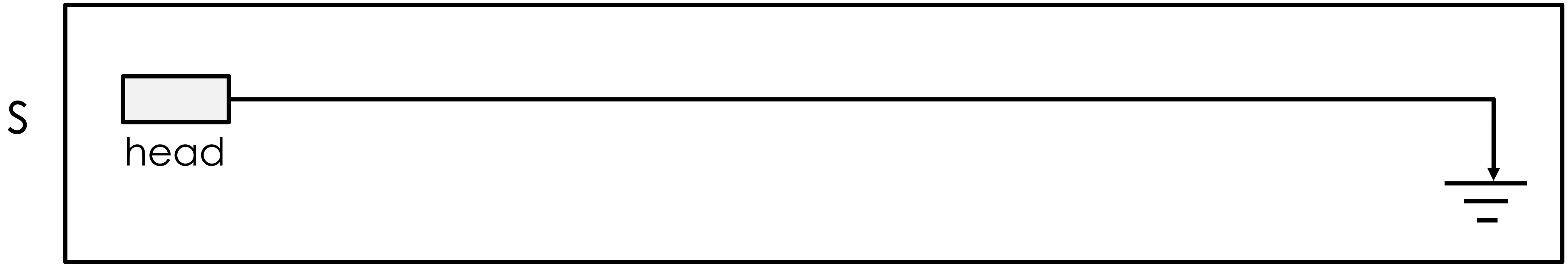
Stack implementation using Linked List



s.isEmpty()

true

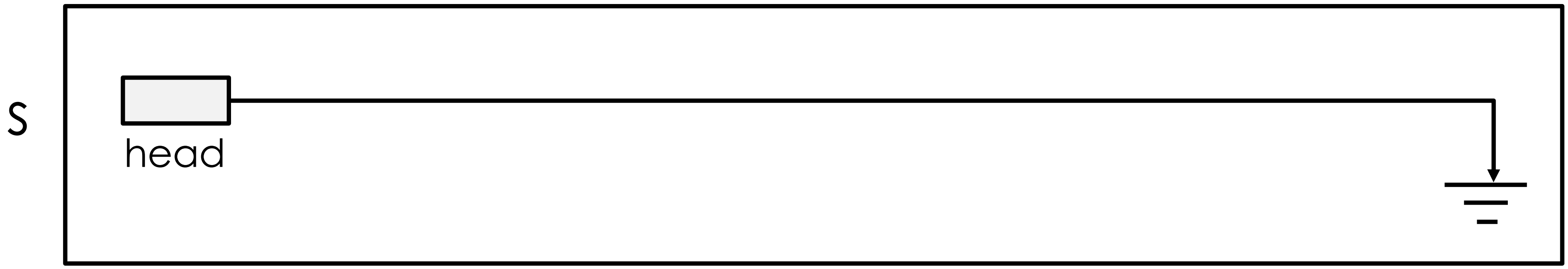
Stack implementation using Linked List



s.pop()

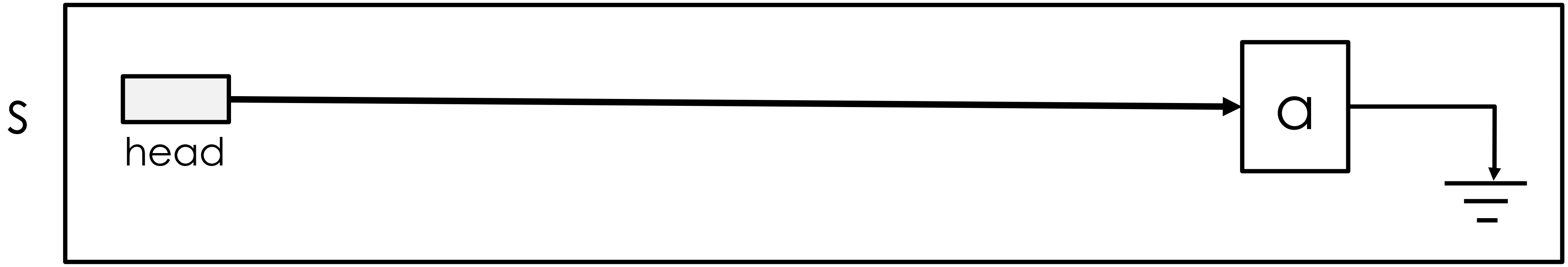
ERROR
StackUnderflowException

Stack implementation using Linked List



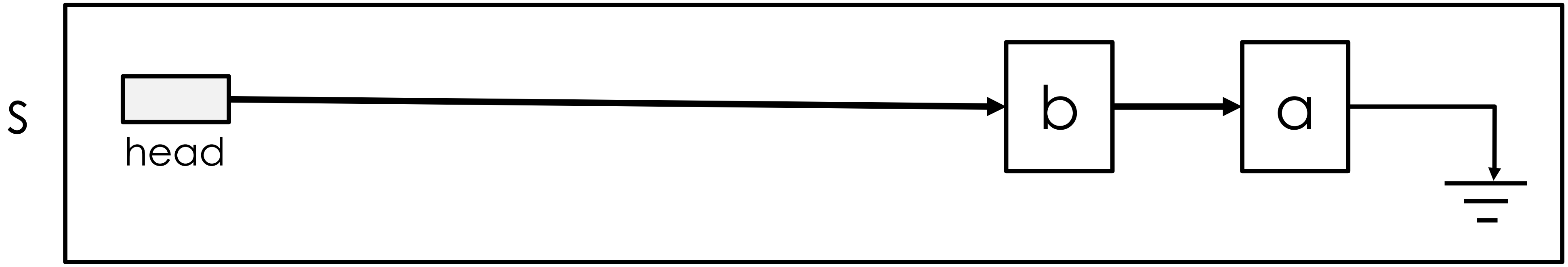
s.push(a)

Stack implementation using Linked List



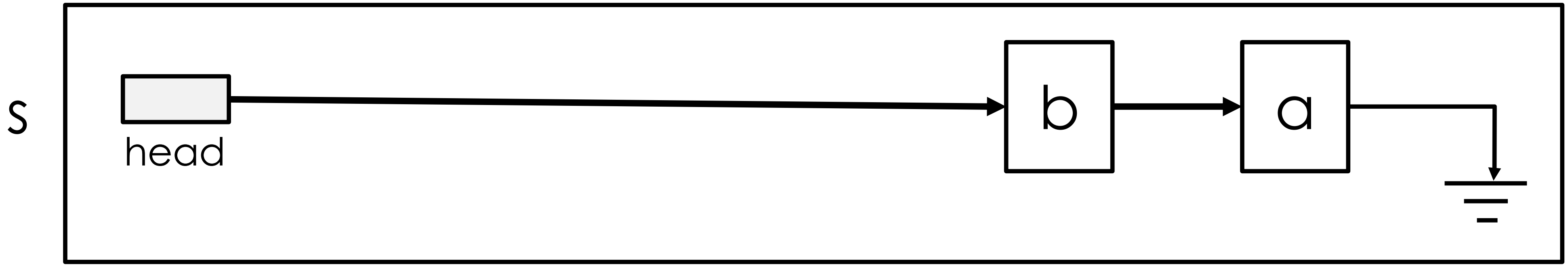
s.push(a)

Stack implementation using Linked List



s.push(b)

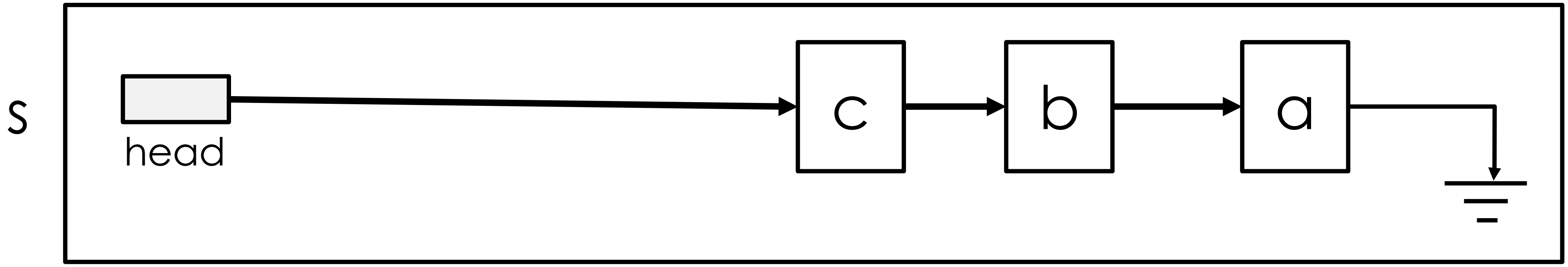
Stack implementation using Linked List



s.top()

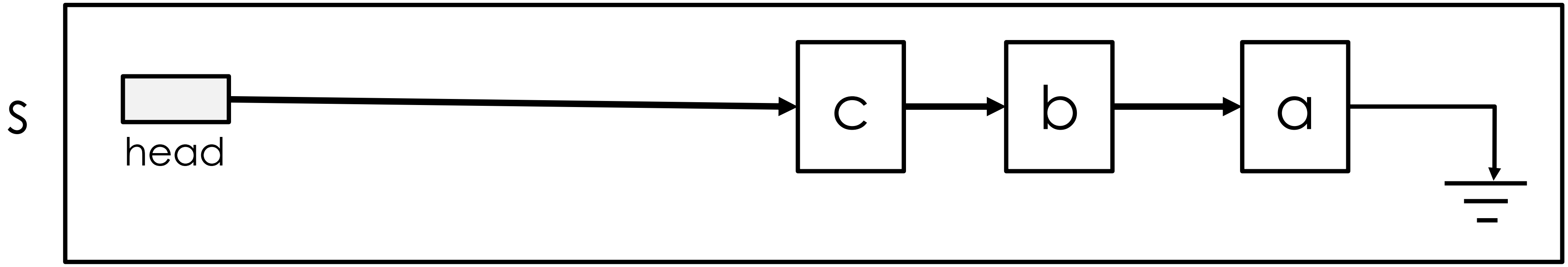
b

Stack implementation using Linked List



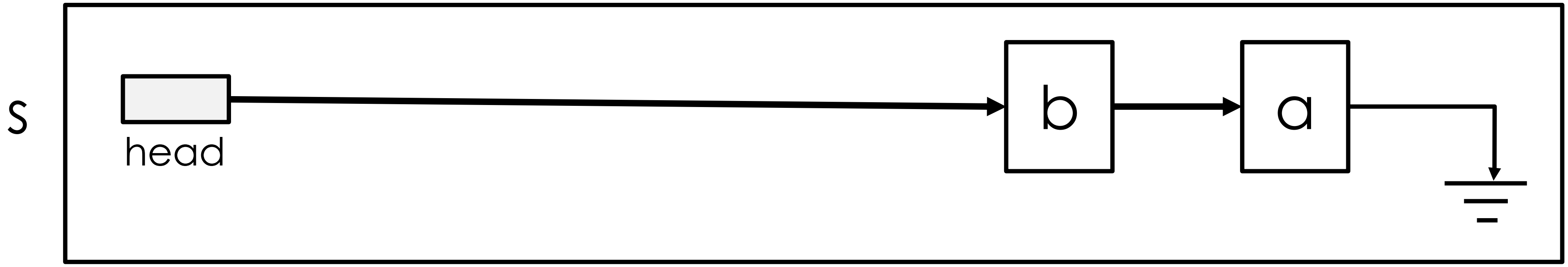
s.push(c)

Stack implementation using Linked List



s.pop()

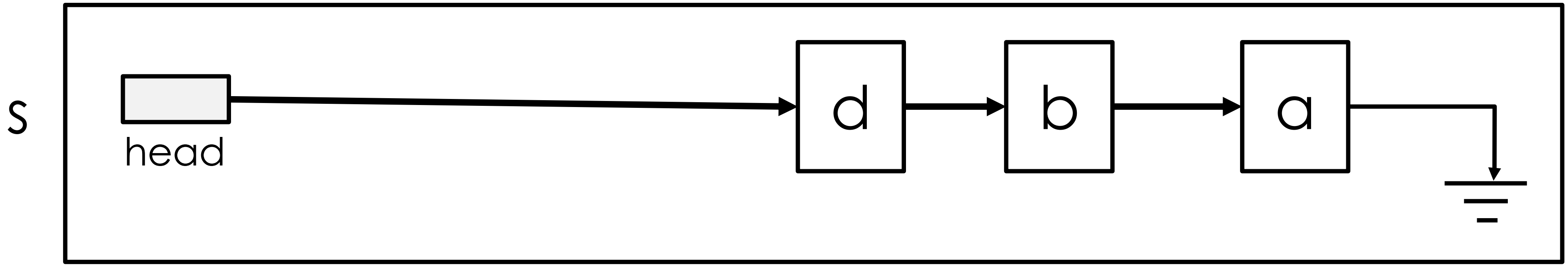
Stack implementation using Linked List



s.pop()

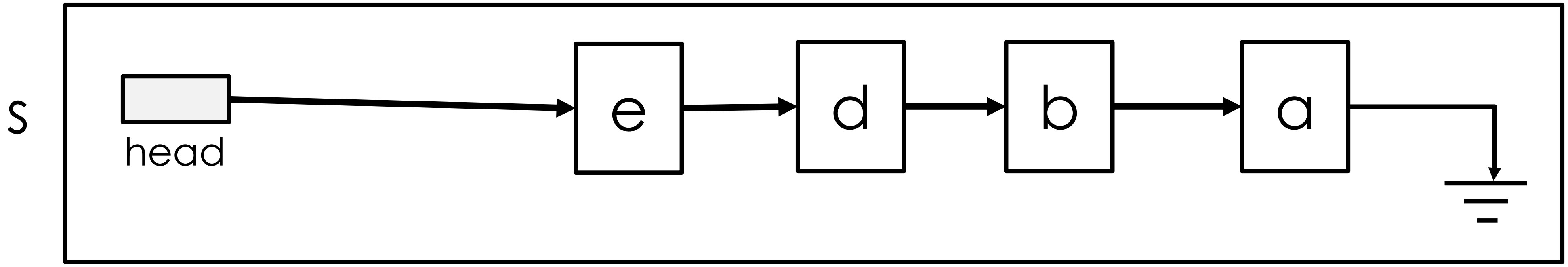
c

Stack implementation using Linked List



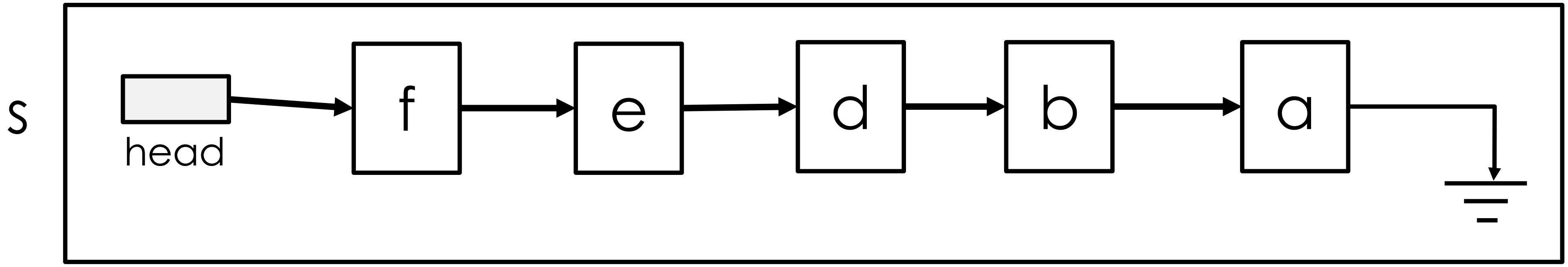
s.push(d)

Stack implementation using Linked List



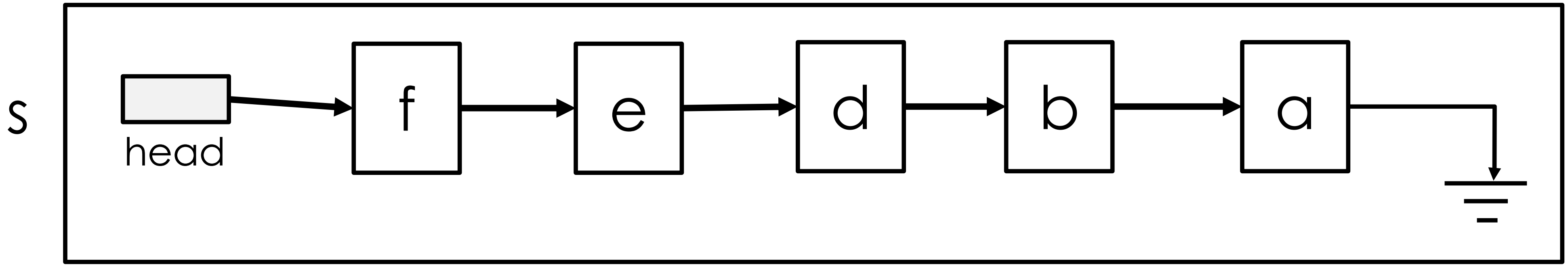
s.push(e)

Stack implementation using Linked List



s.push(f)

Stack implementation using Linked List



s.isEmpty()

false

Stack Applications

- Balance Checking
- Undo Redo Implementation
- Function Calling
- XHTML Parsing
- Reverse Polish notation

Balanced Bracket Problem

- Input:
  - A string str consisting of ‘(‘, ‘)’, ‘[‘, ‘]’ characters.
- Output:
  - Return whether or not the string’s parentheses and square brackets are balanced.

Balanced Bracket Problem

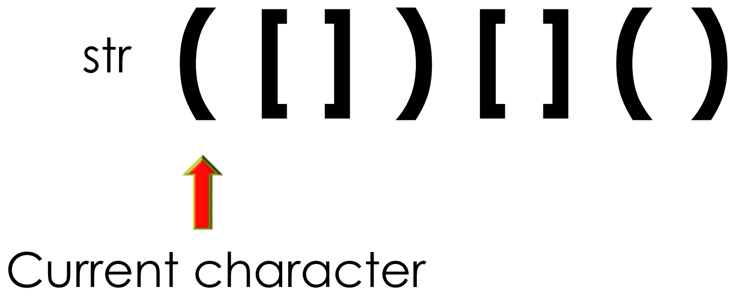
- Balanced:
  - “([ ]) [ ] ( )”
  - “((([([ ])])) ( ))”
- Unbalanced:
  - “([ ]] ( )”
  - “] [“
  - “[ )”

IsBalance Algorithm

([ ]) [ ] ( )

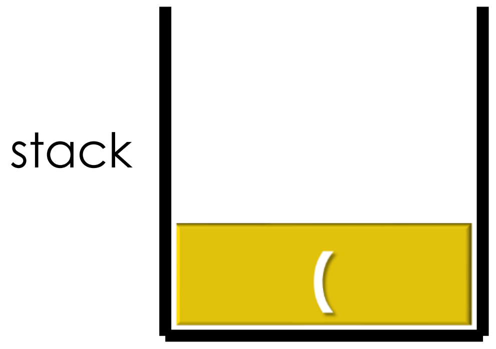
- For each character in the string (the current character)
  - If the current character is the Open's, then Push the current character into a stack and then continue.
  - If the current character is the Close's
    - If stack is empty, return false
    - If stack is not empty, then Pop the stack and check if the top matches the current character.
      - If yes, then continue
      - If no, return false
- Return true if the stack is empty otherwise return false

Boolean IsBalance(str)

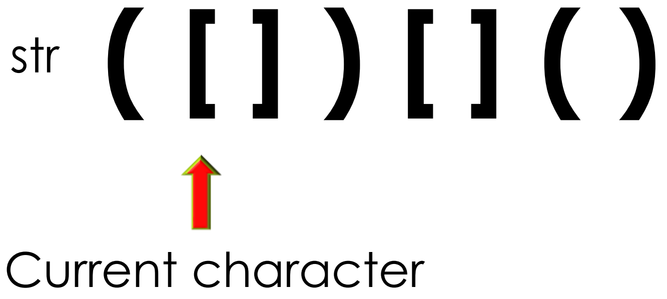


the current == the open's

Push(the current)

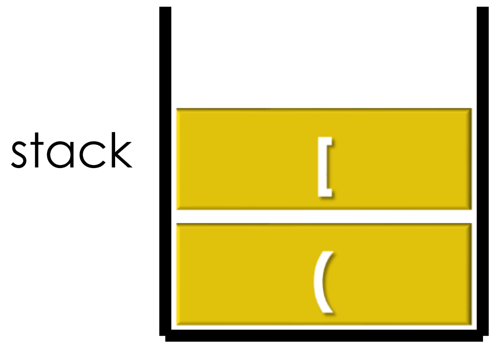


Boolean IsBalance(str)

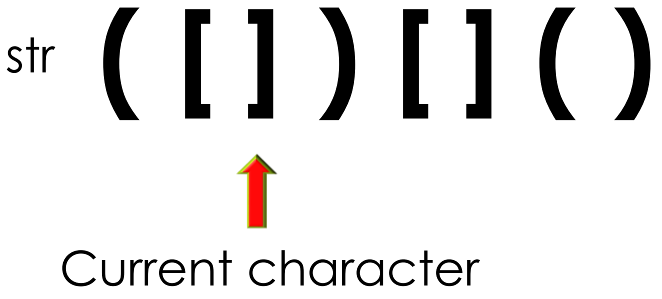


the current == the open's

Push(the current)



Boolean IsBalance(str)

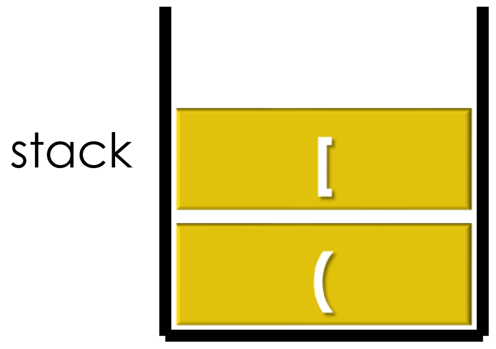


the current == the close’s

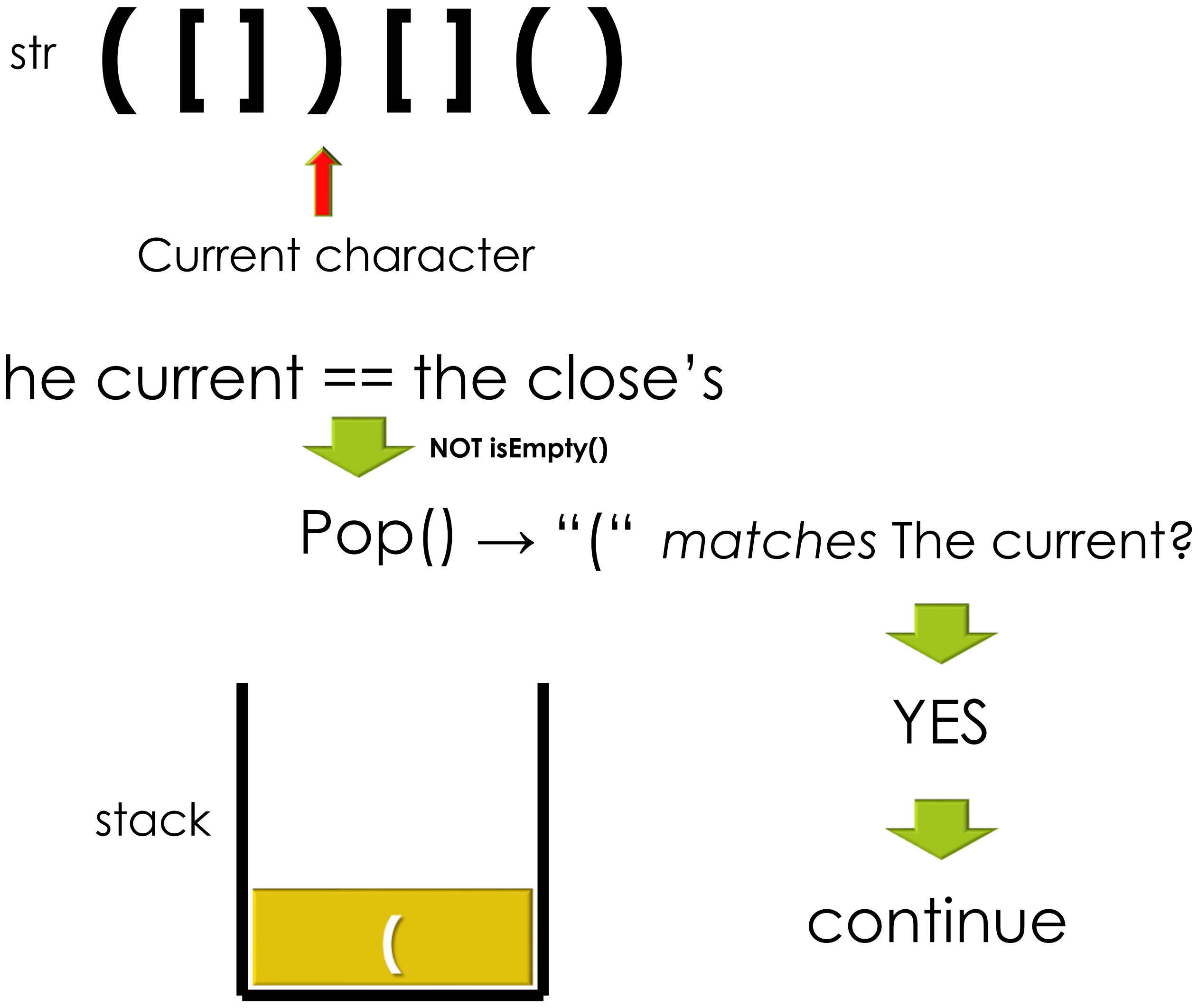
Pop() → “[“ *matches* The current?

YES

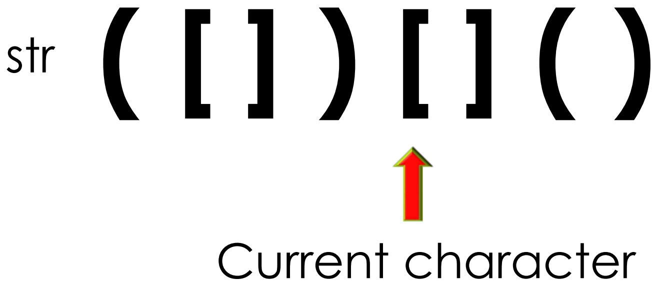
continue



Boolean IsBalance(str)

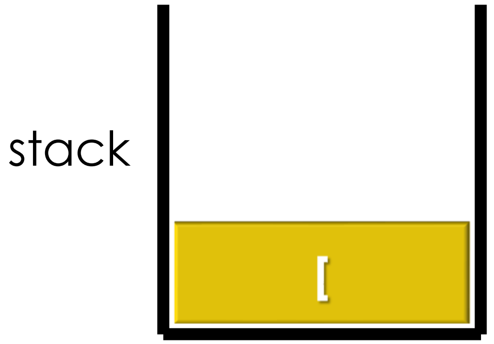


Boolean IsBalance(str)

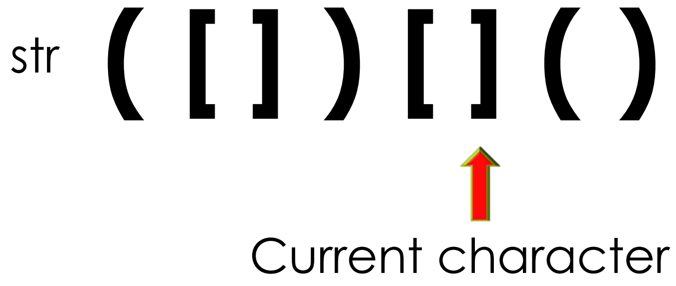


the current == the open's

Push(the current)



Boolean IsBalance(str)



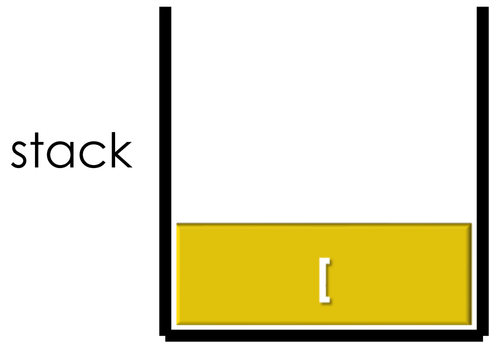
the current == the close’s

NOT isEmpty()

Pop() → “[“ *matches* The current?

YES

continue



Boolean IsBalance(str)

str **([ ]) [ ] ( )**

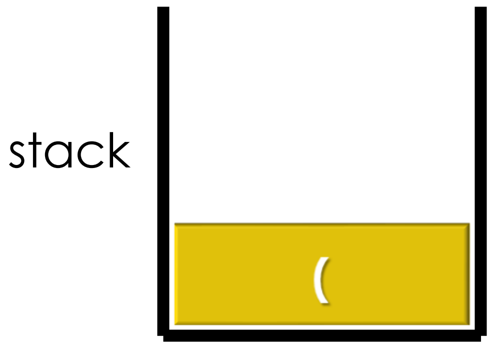
↑

Current character

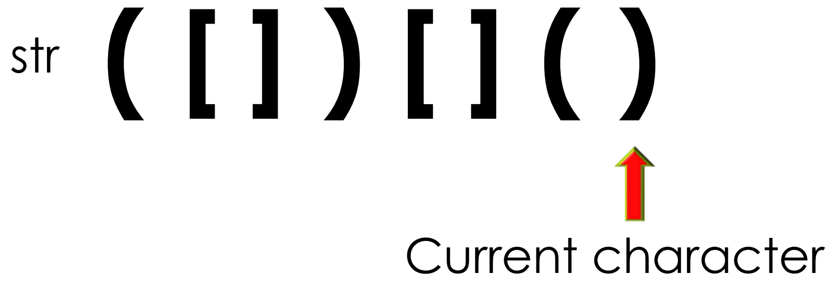
the current == the open's

⬇

Push(the current)



Boolean IsBalance(str)



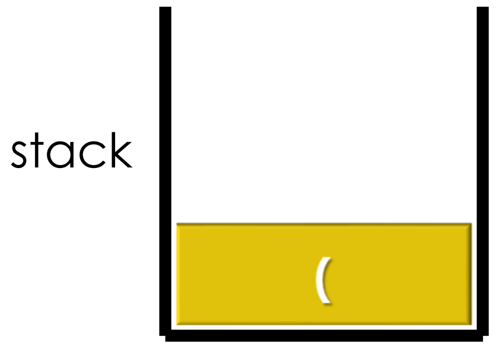
the current == the close's

NOT isEmpty()

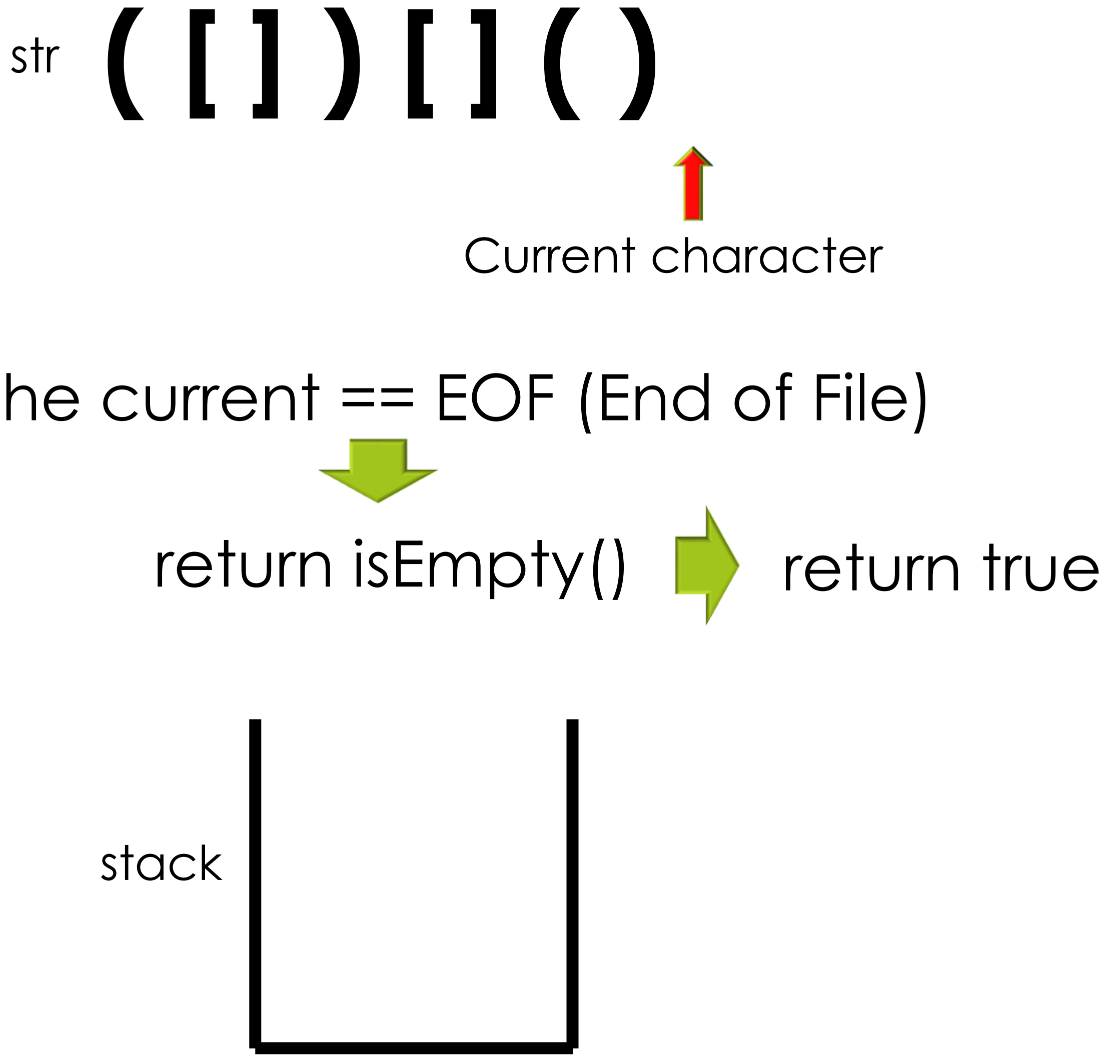
Pop() → "(" *matches* The current?

YES

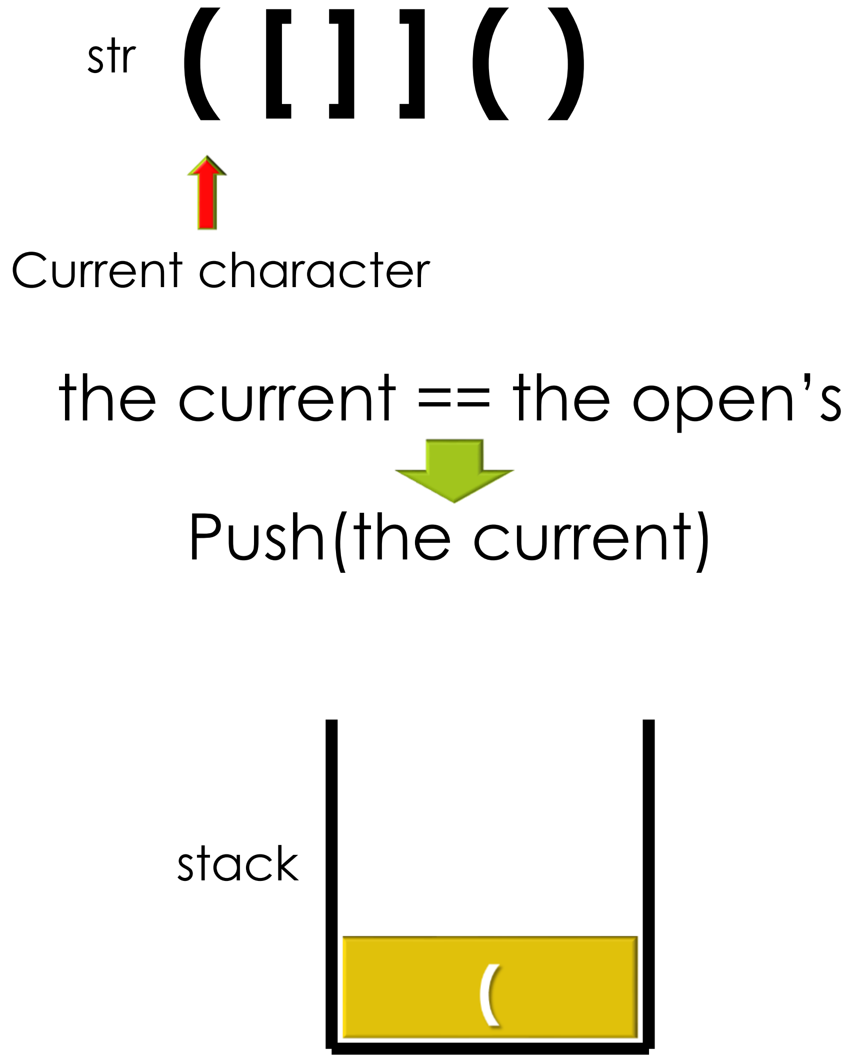
continue



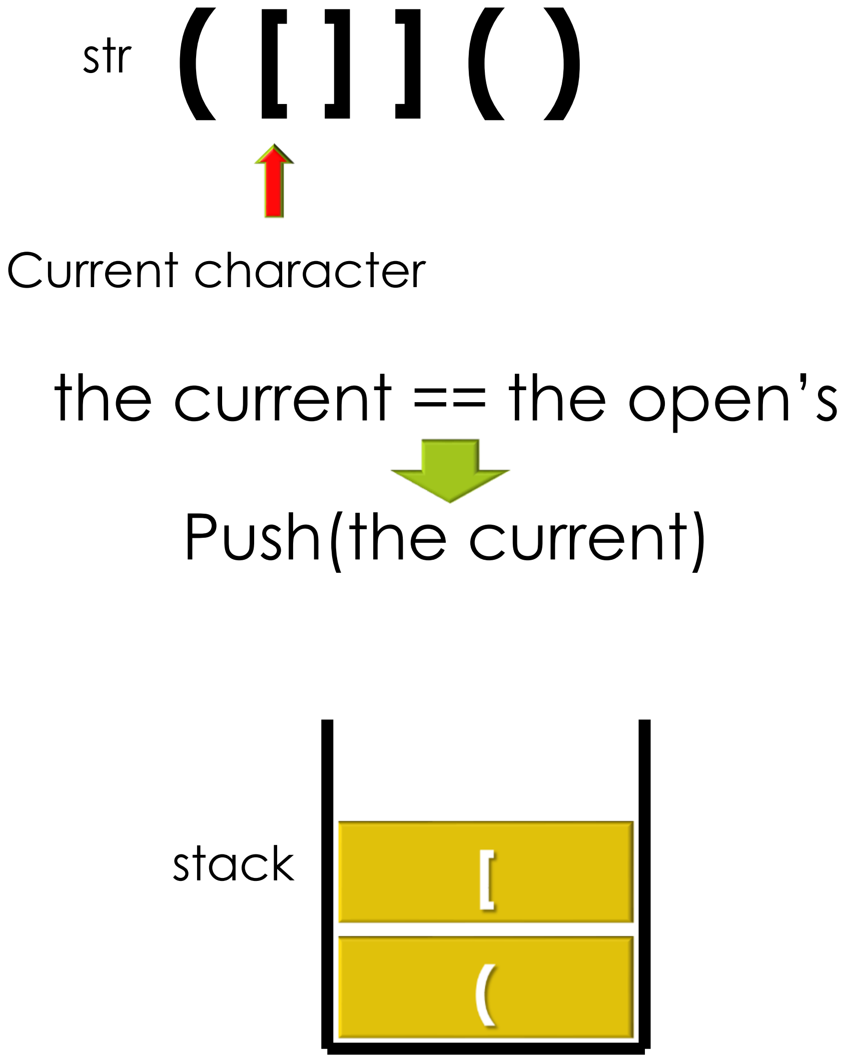
Boolean IsBalance(str)



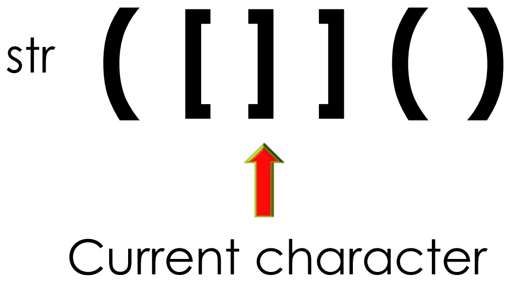
Example 2


str
([ ]] ( )
Current character
the current == the open's
Push(the current)
stack
(

Example 2


str
([ ]] ( )
Current character
the current == the open's
Push(the current)
stack
[
(

Boolean IsBalance(str)



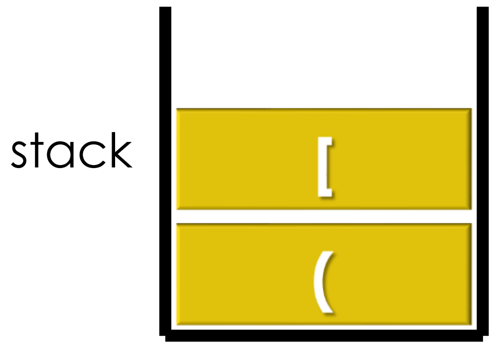
the current == the close's

NOT isEmpty()

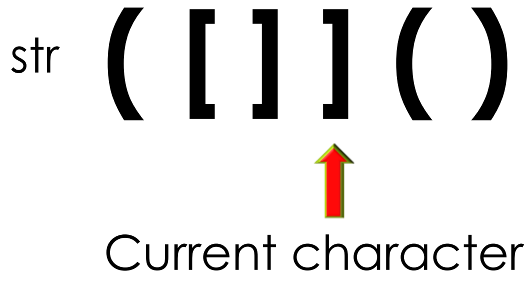
Pop() → "[" *matches* The current?

YES

continue



Boolean IsBalance(str)



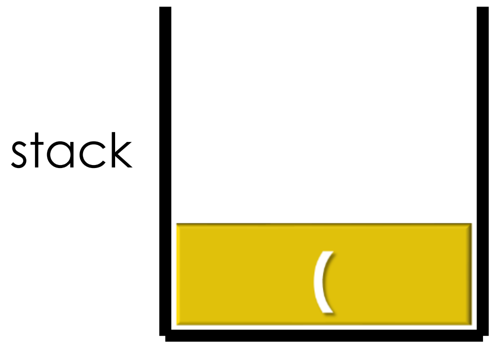
the current == the close’s

NOT isEmpty()

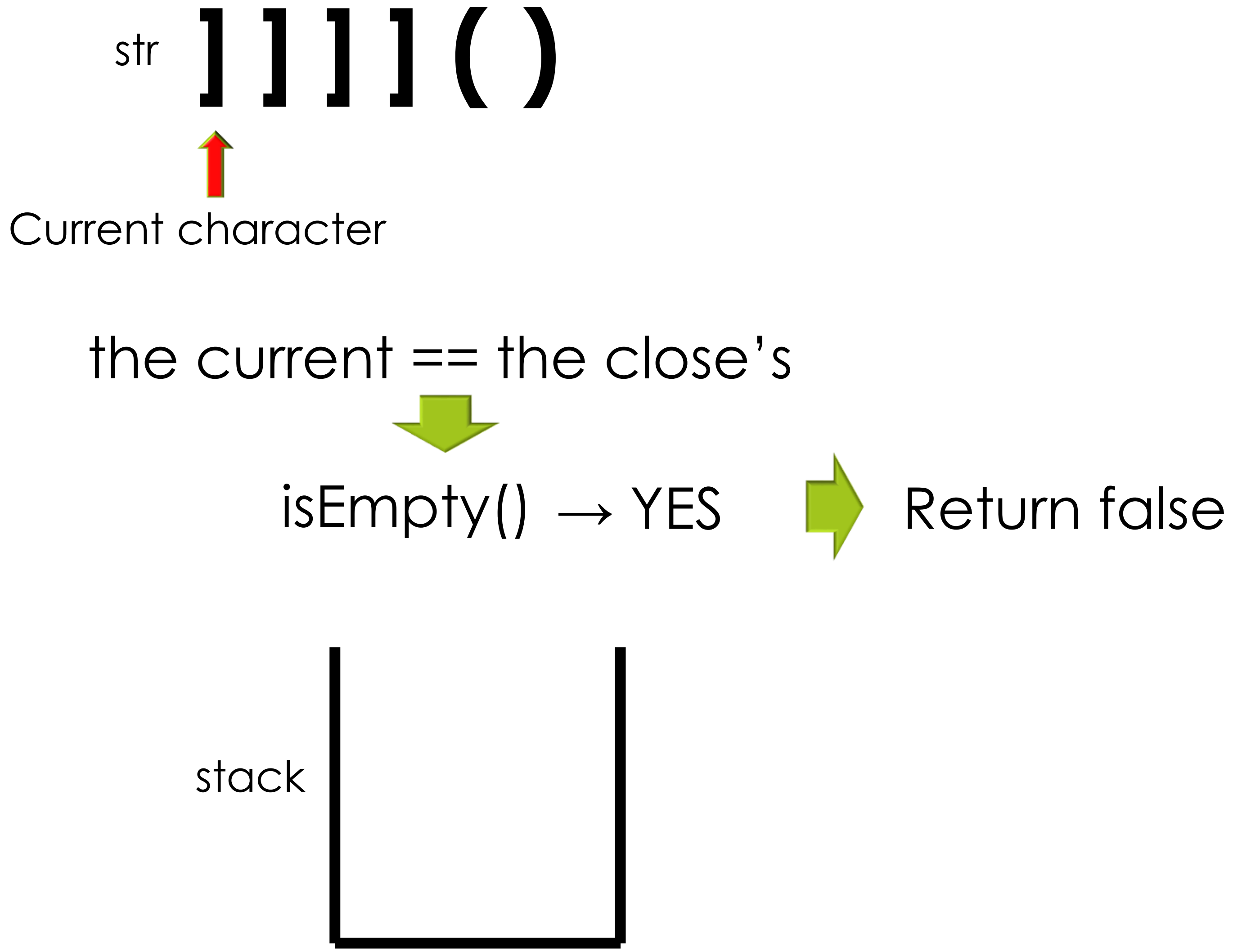
Pop() → “(“ *matches* The current?

NO

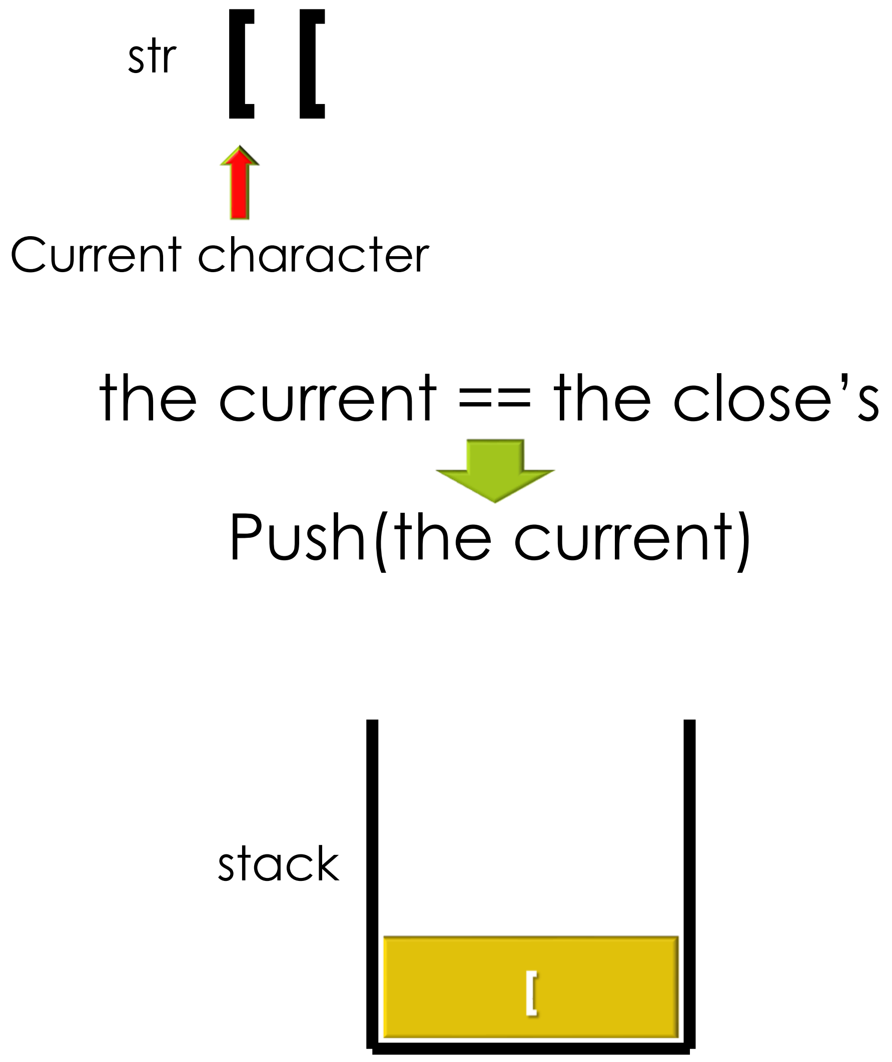
Return false



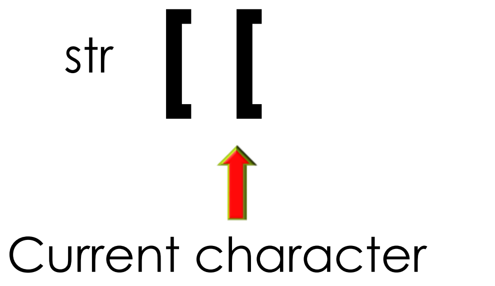
Example 3


str
]]]] ( )
Current character
the current == the close's
isEmpty() → YES
Return false
stack

Example 4

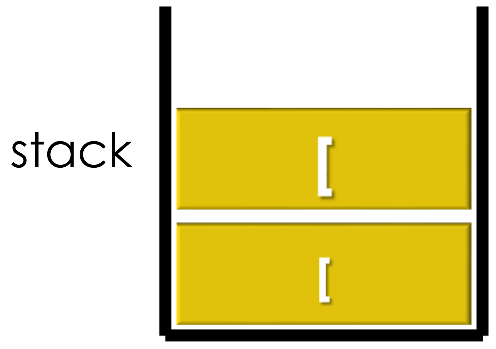

str
[[
Current character
the current == the close's
Push(the current)
stack
[

Example 4

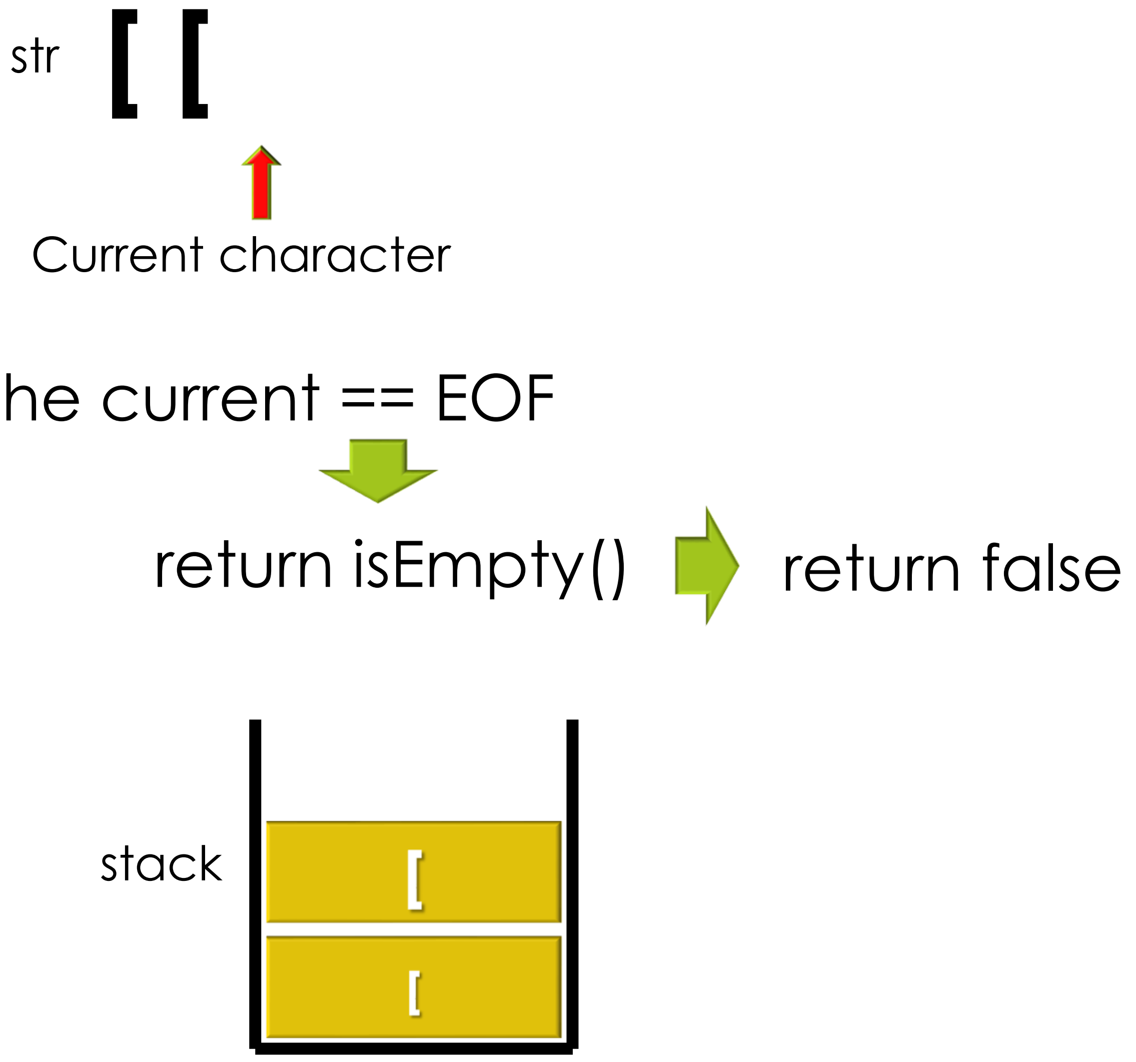


the current == the close's

Push(the current)

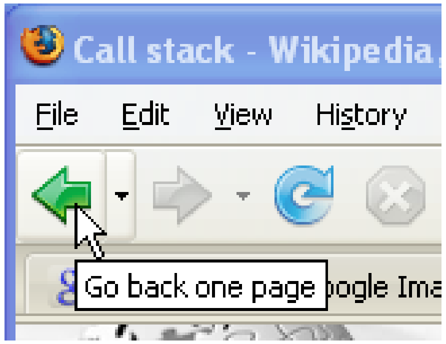


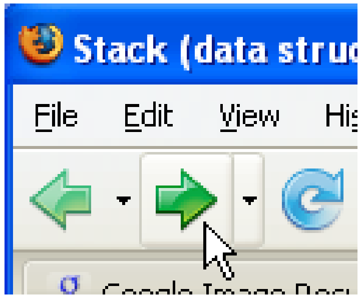
Example 4


str
[[
Current character
the current == EOF
return isEmpty()
return false
stack
[
[

Undo Redo using Stacks

- The runtime stack used by a process (running program) to keep track of methods in progress
- Undo, redo, back, forward





Undo in Word using a stack

- "Undo" can reverse the state to previous states.
- "Redo" reverses the "Undo"
- Undo/Redo in Word Processing: implementation using one stack:
  - When typing new words (ending with white space), the stack is cleared.
  - When undoing, the current word is pushed into the stack and then erased.
  - When redoing, the stack is popped, and the top word is added to the paragraph.
- Example:
  - "I" "love" "cats" [UNDO] [UNDO] [UNDO] [REDO] "like" "dogs" [UNDO] [UNDO] [REDO] "cats" "and" "dogs" [UNDO] [UNDO] "very" "much" [UNDO] [UNDO] [REDO] [REDO]

Parsing XHTML

- Stacks may be used to parse an XHTML document
- A *markup language* is a means of annotating a document to given context to the text
  - The annotations give information about the structure or presentation of the text
- The best known example is HTML, or HyperText Markup Language
  - We will look at XHTML

Parsing XHTML

XHTML is made of nested

- *opening tags*, e.g., `<some_identifier>`, and
- matching *closing tags*, e.g., `</some_identifier>`

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```

Parsing XHTML

Nesting indicates that any closing tag must match the most <u>recent</u> opening tag

Strategy for parsing XHTML:

- read though the XHTML linearly
- place the opening tags in a stack
- when a closing tag is encountered, check that it matches what is on top of the stack and

Parsing XHTML

```
<html>

  <head><title>Hello</title></head>

  <body><p>This appears in the
  <i>browser</i>.</p></body>

</html>
```

<html>			

Parsing XHTML

```
<html>

  <head><title>Hello</title></head>

  <body><p>This appears in the
  <i>browser</i>.</p></body>

</html>
```

`<html>`	`<head>`		

Parsing XHTML

```
<html>

  <head><title>Hello</title></head>

  <body><p>This appears in the
  <i>browser</i>.</p></body>

</html>
```

`<html>`	`<head>`	`<title>`	

Parsing XHTML

```
<html>

  <head><title>Hello</title></head>

  <body><p>This appears in the
  <i>browser</i>.</p></body>

</html>
```

`<html>`	`<head>`	`<title>`	

Parsing XHTML

```
<html>

  <head><title>Hello</title></head>

  <body><p>This appears in the
  <i>browser</i>.</p></body>

</html>
```

`<html>`	`<head>`		

Parsing XHTML

```
<html>

  <head><title>Hello</title></head>

  <body><p>This appears in the
  <i>browser</i>.</p></body>

</html>
```

<html>	<body>		

Parsing XHTML

```
<html>

  <head><title>Hello</title></head>

  <body><p>This appears in the
  <i>browser</i>.</p></body>

</html>
```

<html>	<body>	<p>	

Parsing XHTML

```
<html>

  <head><title>Hello</title></head>

  <body><p>This appears in the
  <i>browser</i>.</p></body>

</html>
```

<html>	<body>	<p>	<i>

Parsing XHTML

```
<html>

  <head><title>Hello</title></head>

  <body><p>This appears in the
  <i>browser</i>.</p></body>

</html>
```

`<html>`	`<body>`	`<p>`	`<i>`

Parsing XHTML

```
<html>

  <head><title>Hello</title></head>

  <body><p>This appears in the
  <i>browser</i>.</p></body>

</html>
```

`<html>`	`<body>`	`<p>`	

Parsing XHTML

```
<html>

  <head><title>Hello</title></head>

  <body><p>This appears in the
  <i>browser</i>.</p></body>

</html>
```

<html>	<body>		

Parsing XHTML

```
<html>

  <head><title>Hello</title></head>

  <body><p>This appears in the
  <i>browser</i>.</p></body>

</html>
```

<html>			

Parsing XHTML

We are finished parsing, and the stack is empty

Possible errors:

- a closing tag which does not match the opening tag on top of the stack
- a closing tag when the stack is empty
- the stack is not empty at the end of the document

Function Calls

Function calls need stack data structure:

- you write a function to solve a problem
- the function may require sub-problems to be solved, hence, it may call another function
- once a function is finished, it returns to the function which called it

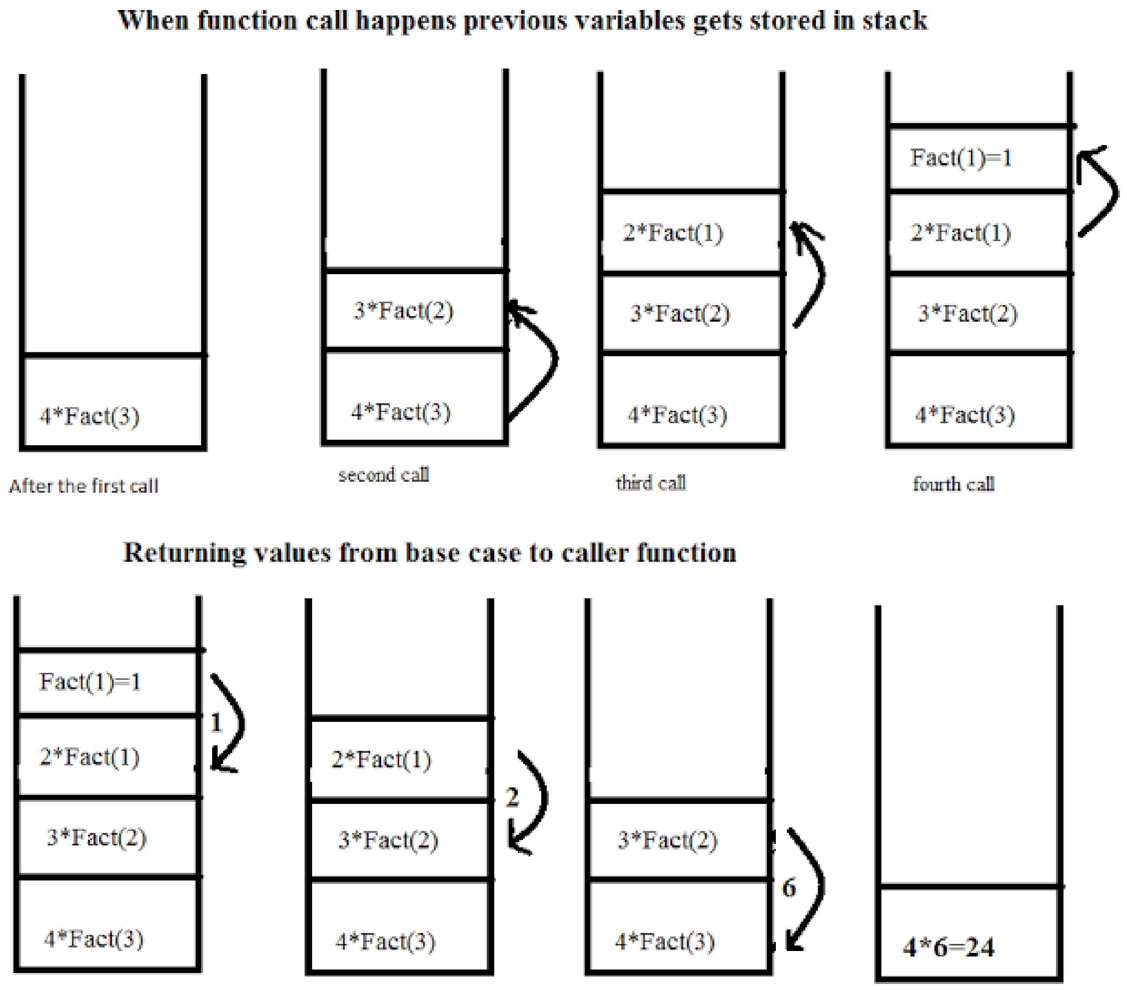
Function Calls

- You will notice that the when a function returns, execution and the return value is passed back to the last function which was called
- This is again, the last-in—first-out property

Function calls

```
System.out.println(Fact(3))
```

```
int Fact(int n){
   if n==1{
      return 1;
   else
      return n*Fact(n-1);
}
```



Reverse-Polish Notation

Normally, mathematics is written using what we call *in-fix* notation:

$$(3 + 4) \times 5 - 6$$

The operator is placed between to operands

One weakness: parentheses are required

$$(3 + 4) \times 5 - 6 = 29$$

$$3 + 4 \times 5 - 6 = 17$$

$$3 + 4 \times (5 - 6) = -1$$

$$(3 + 4) \times (5 - 6) = -7$$

Postfix (Reverse-Polish) Notation

Alternatively, we can place the operands first, followed by the operator:

$$(3 + 4) \times 5 - 6$$

$$3 \ 4 \ + \ 5 \ \times \ 6 \ -$$

Parsing reads left-to-right and performs any operation on the last two operands:

$$3 \ 4 \ + \ 5 \ \times \ 6 \ -$$

$$7 \quad 5 \ \times \ 6 \ -$$

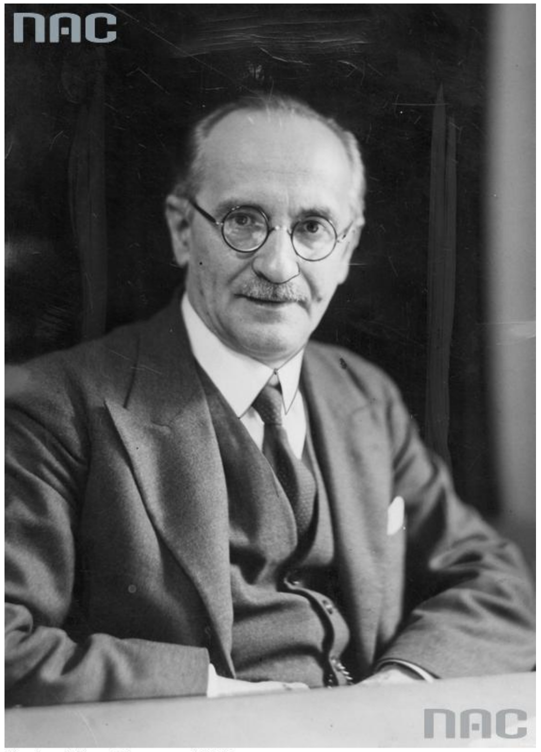
$$35 \quad 6 \ -$$

$$29$$

Postfix (Reverse-Polish) Notation

This is called *reverse-Polish* notation after the mathematician Jan Łukasiewicz

He also made significant contributions to logic and other fields



Narodowe Archiwum Cyfrowe, sygn. 1-N-358

http://www.audiovis.nac.gov.pl/

Postfix (Reverse-Polish) Notation

Other examples:

```
3  4  5  ×  +  6  –
3   20      +  6  –
    23         6  –
             17
3  4  5  6  –  ×  +
3  4   –1      ×  +
3      –4         +
       –1
```

Postfix (Reverse-Polish) Notation

Benefits:

- No ambiguity and no brackets are required
- It is the same process used by a computer to perform computations:
  - operands must be loaded into registers before operations can be performed on them
- Reverse-Polish can be processed using stacks

Postfix (Reverse-Polish) Notation

Reverse-Polish notation is used with some programming languages

- *e*.*g*., postscript, pdf, and HP calculators

Similar to the thought process required for writing assembly language code

- you cannot perform an operation until you have all of the operands loaded into registers

```
MOVE 42, D1        ; Load  42 into Register D1
MOVE 256, D2       ; Load 256 into Register D2
ADD D2, D1         ; Add D2 into D1
```

Postfix (Reverse-Polish) Notation

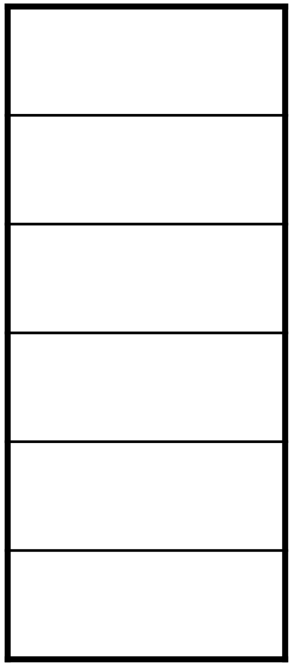
The easiest way to parse reverse-Polish notation is to use an operand stack:

- operands are processed by pushing them onto the stack
- when processing an operator:
  - pop the last two items off the operand stack,
  - perform the operation, and
  - push the result back onto the stack

Postfix (Reverse-Polish) Notation

Evaluate the following reverse-Polish expression using a stack:

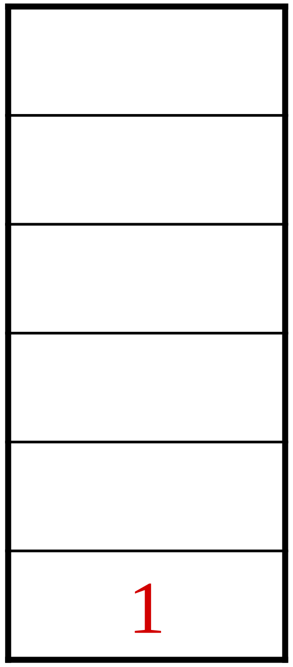
$$1\ 2\ 3\ +\ 4\ 5\ 6\ \times\ -\ 7\ \times\ +\ -\ 8\ 9\ \times\ +$$



Postfix (Reverse-Polish) Notation

Push 1 onto the stack

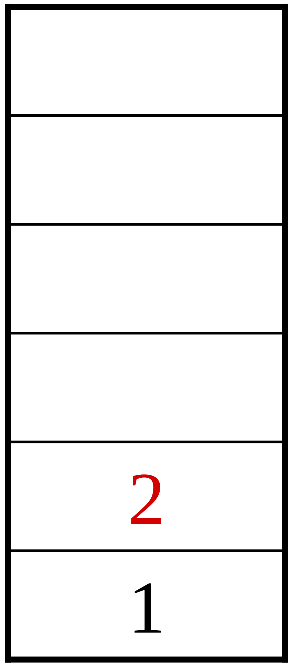
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Postfix (Reverse-Polish) Notation

Push 1 onto the stack

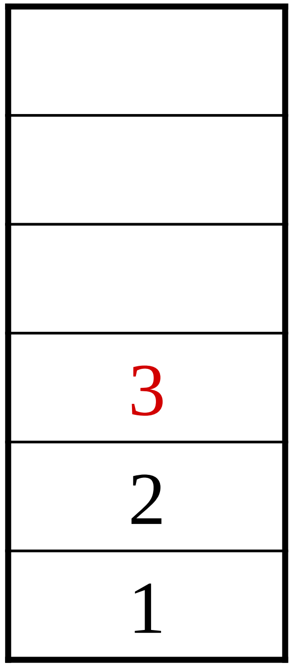
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Postfix (Reverse-Polish) Notation

Push 3 onto the stack

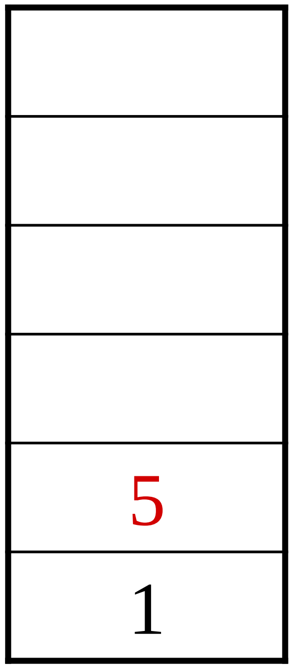
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Postfix (Reverse-Polish) Notation

Pop 3 and 2 and push 2 + 3 = 5

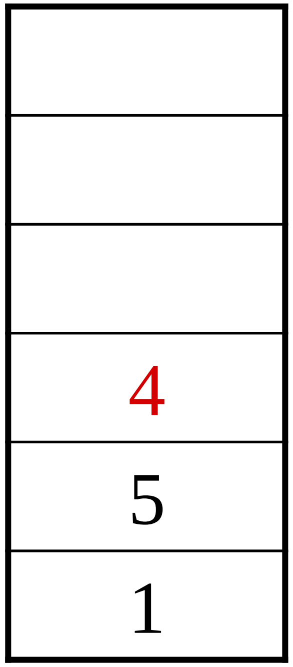
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Postfix (Reverse-Polish) Notation

Push 4 onto the stack

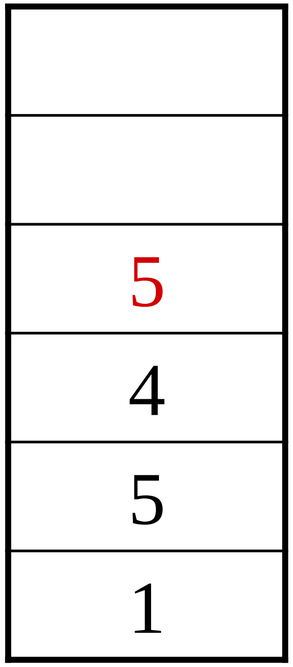
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Postfix (Reverse-Polish) Notation

Push 5 onto the stack

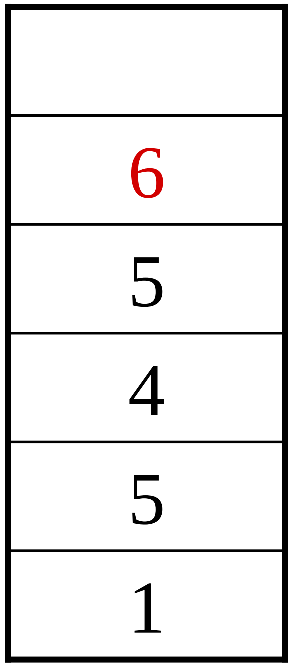
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Postfix (Reverse-Polish) Notation

Push 6 onto the stack

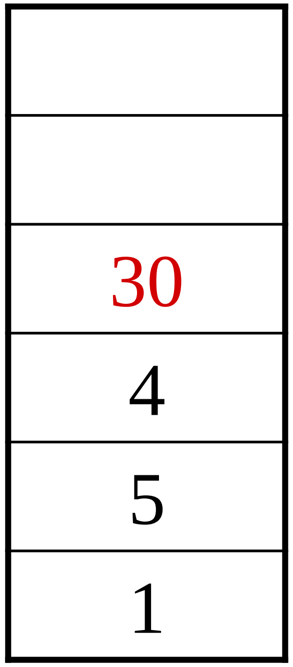
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Postfix (Reverse-Polish) Notation

Pop 6 and 5 and push 5 × 6 = 30

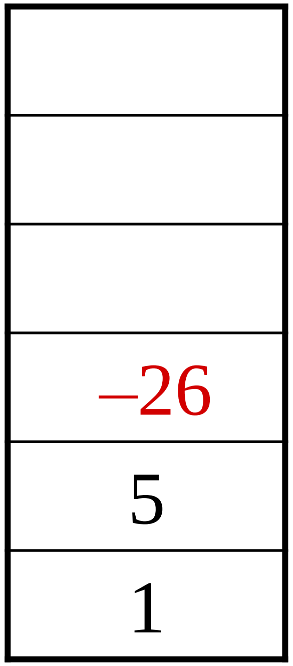
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Postfix (Reverse-Polish) Notation

Pop 30 and 4 and push 4 – 30 = –26

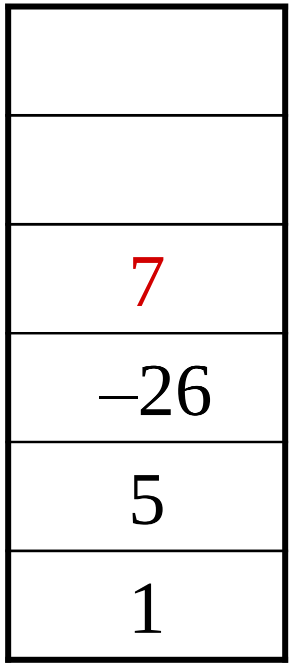
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Postfix (Reverse-Polish) Notation

Push 7 onto the stack

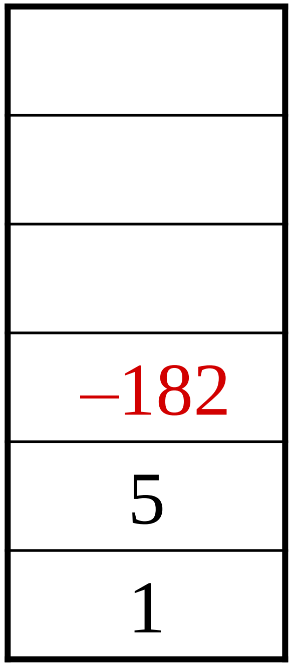
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Postfix (Reverse-Polish) Notation

Pop 7 and –26 and push –26 × 7 = –182

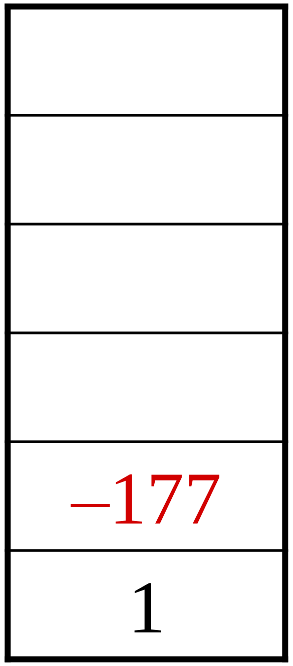
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Postfix (Reverse-Polish) Notation

Pop –182 and 5 and push –182 + 5 = –177

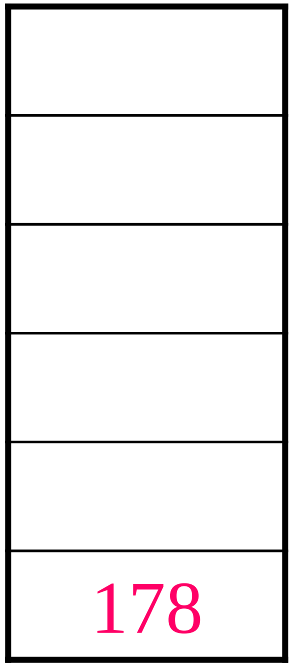
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Postfix (Reverse-Polish) Notation

Pop –177 and 1 and push 1 – (–177) = 178

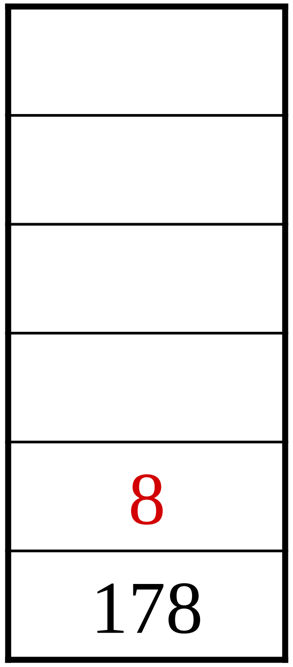
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Postfix (Reverse-Polish) Notation

Push 8 onto the stack

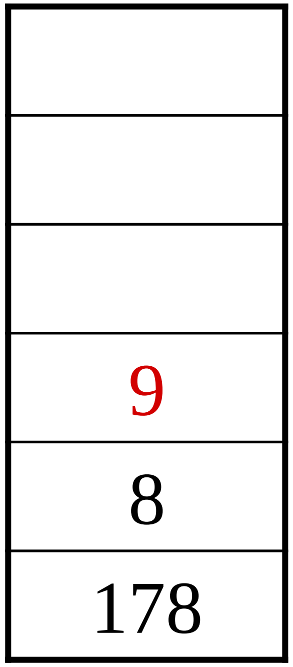
$$1\ 2\ 3\ +\ 4\ 5\ 6\ \times\ -\ 7\ \times\ +\ -\ 8\ 9\ \times\ +$$



Postfix (Reverse-Polish) Notation

Push 1 onto the stack

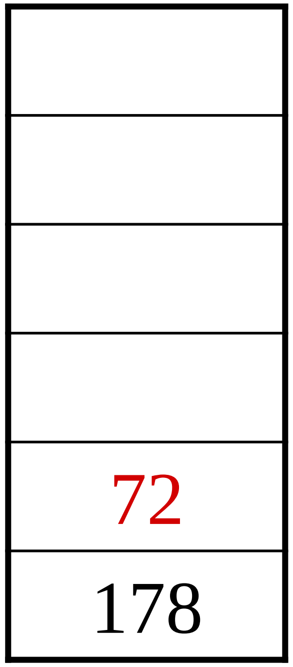
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Postfix (Reverse-Polish) Notation

Pop 9 and 8 and push $8 \times 9 = 72$

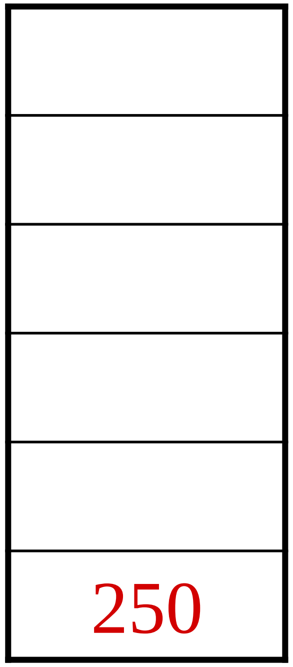
$$1\ 2\ 3\ +\ 4\ 5\ 6\ \times\ -\ 7\ \times\ +\ -\ 8\ 9\ \times\ +$$



Postfix (Reverse-Polish) Notation

Pop 72 and 178 and push 178 + 72 = 250

1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Postfix (Reverse-Polish) Notation

Thus

$$1 \ 2 \ 3 \ + \ 4 \ 5 \ 6 \ \times \ - \ 7 \ \times \ + \ - \ 8 \ 9 \ \times \ +$$

evaluates to the value on the top: 250

The equivalent in-fix notation is

$$((1 - ((2 + 3) + ((4 - (5 \times 6)) \times 7))) + (8 \times 9))$$

We reduce the parentheses using order-of-operations:

$$1 - (2 + 3 + (4 - 5 \times 6) \times 7) + 8 \times 9$$

Postfix (Reverse-Polish) Notation

Incidentally,

$$1 - 2 + 3 + 4 - 5 \times 6 \times 7 + 8 \times 9 = -132$$

which has the reverse-Polish notation of

$$1 \quad 2 \quad - \quad 3 \quad + \quad 4 \quad + \quad 5 \quad 6 \quad 7 \quad \times \quad \times \quad - \quad 8 \quad 9 \quad \times \quad +$$

For comparison, the calculated expression was

$$1 \quad 2 \quad 3 \quad + \quad 4 \quad 5 \quad 6 \quad \times \quad - \quad 7 \quad \times \quad + \quad - \quad 8 \quad 9 \quad \times \quad +$$

Summary

- Stacks can be implemented with either an array or a linked list.
- Each stack operation is O(1): Push, Pop, Top, Empty.
- Stacks are occasionally known as LIFO queues
- Stack applications such as balancing symbols, Markup language parsing, function calling, reverse Polish notation or postfix notation