CSE 220
Data Structures

Lecture 08:
Stacks

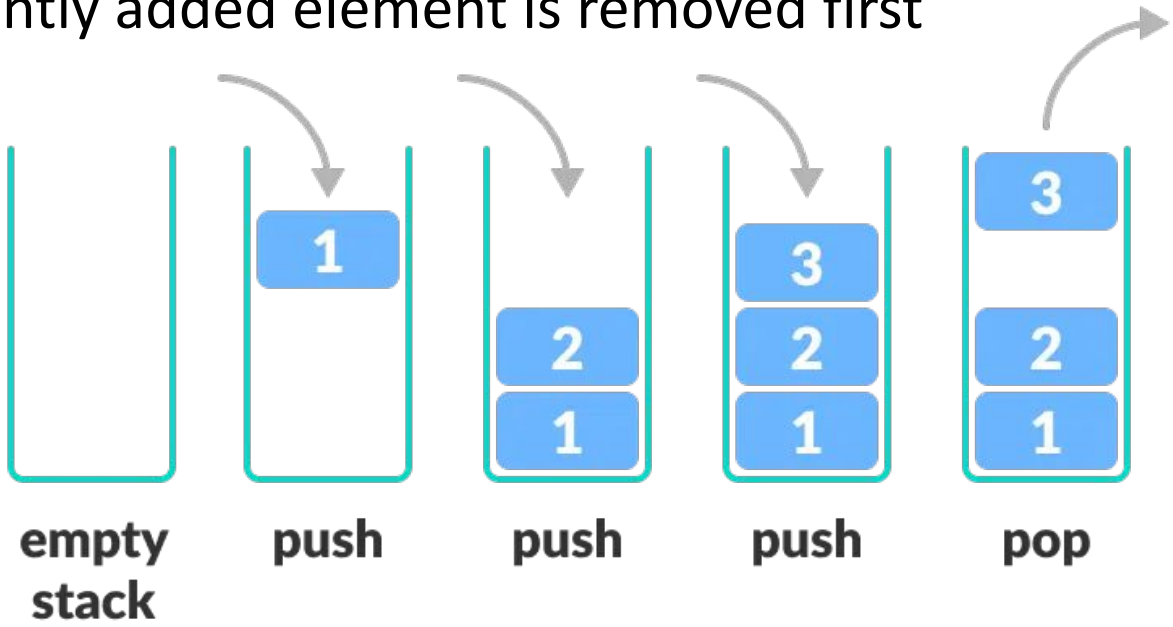
Anwarul Bashir Shuaib [AWBS]
Lecturer
Department of Computer Science and Engineering
BRAC University

Stacks

- A LIFO (Last In, First Out) data structure
- Items are added on top of each other
- When removing items, the most recently added item is removed first

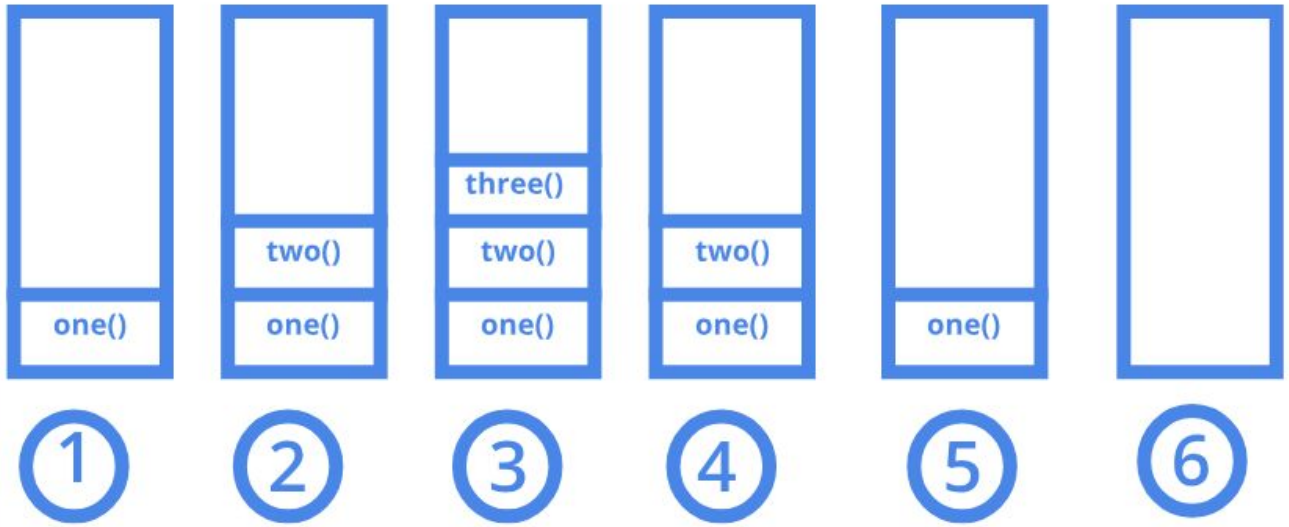
Stacks

- A LIFO (Last In, First Out) data structure
- Add elements ⇒ push
- Remove elements ⇒ pop
  - The most recently added element is removed first



Applications – Function Calls

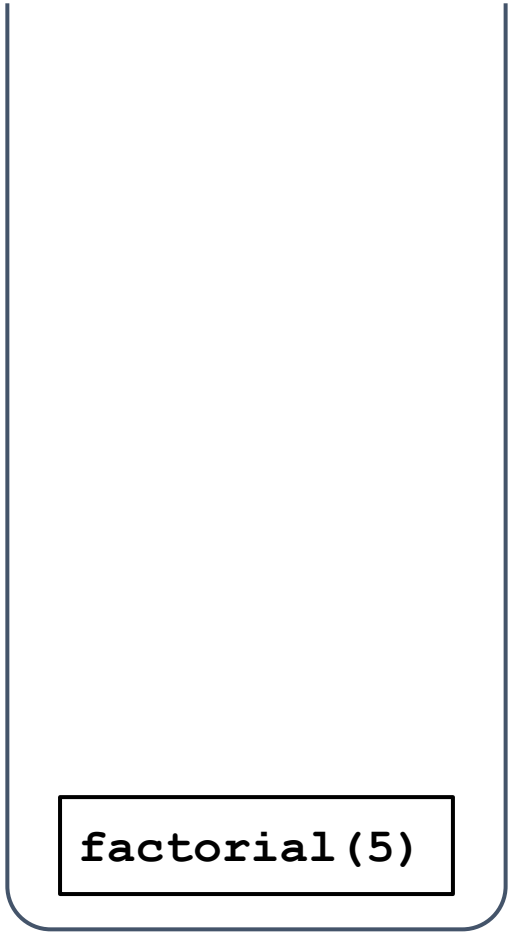
```
function one(){
  two();
}

function two() {
  three();
}

function three() {
  console.trace("Call
                Stack");
}
```



Applications – Function Calls

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

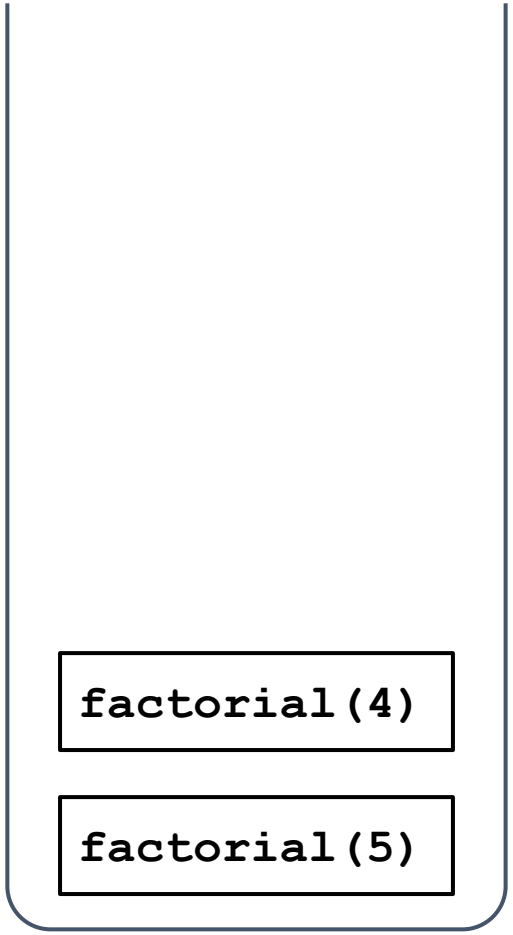
What would happen if you call factorial(5)?



Applications – Function Calls

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

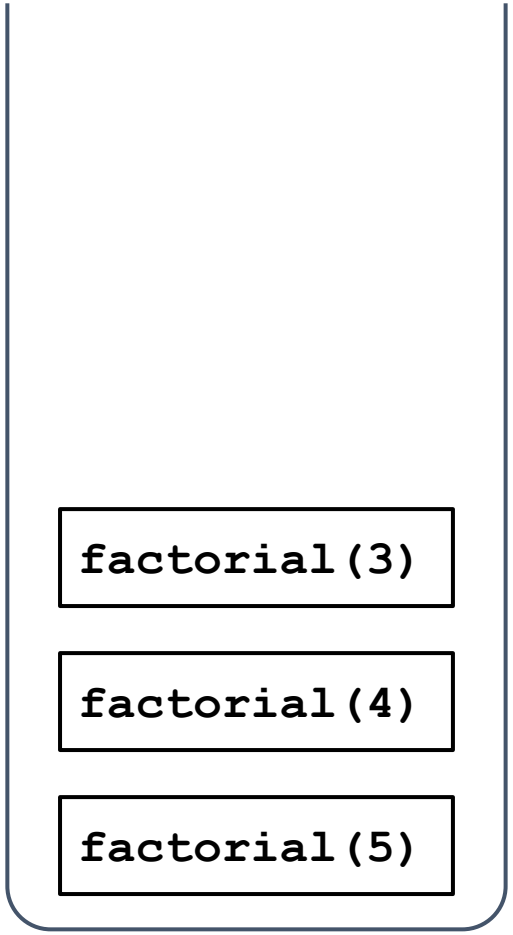
What would happen if you call factorial(5)?



Applications – Function Calls

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

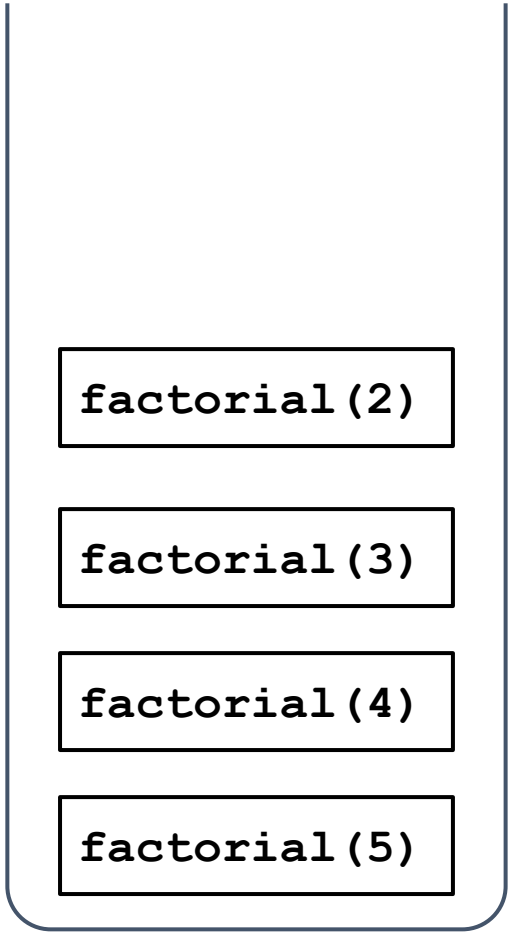
What would happen if you call factorial(5)?



Applications – Function Calls

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

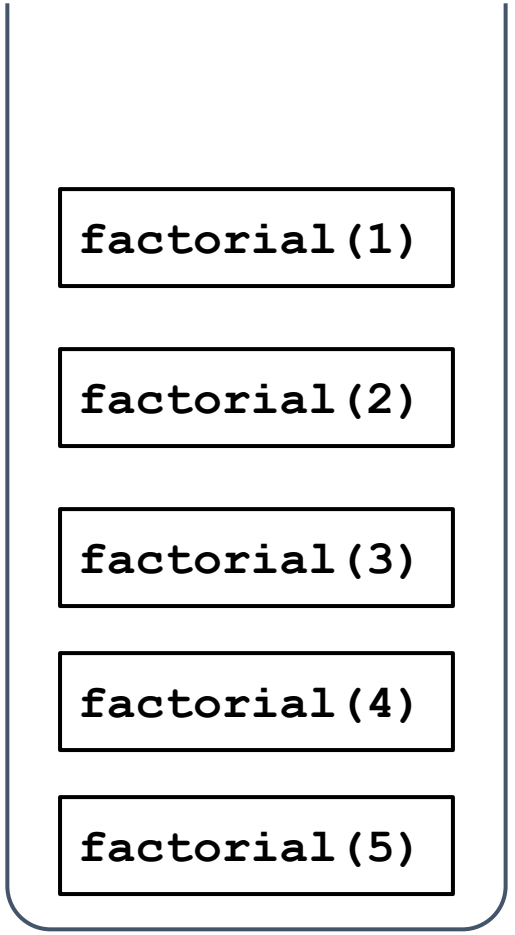
What would happen if you call factorial(5)?



Applications – Function Calls

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

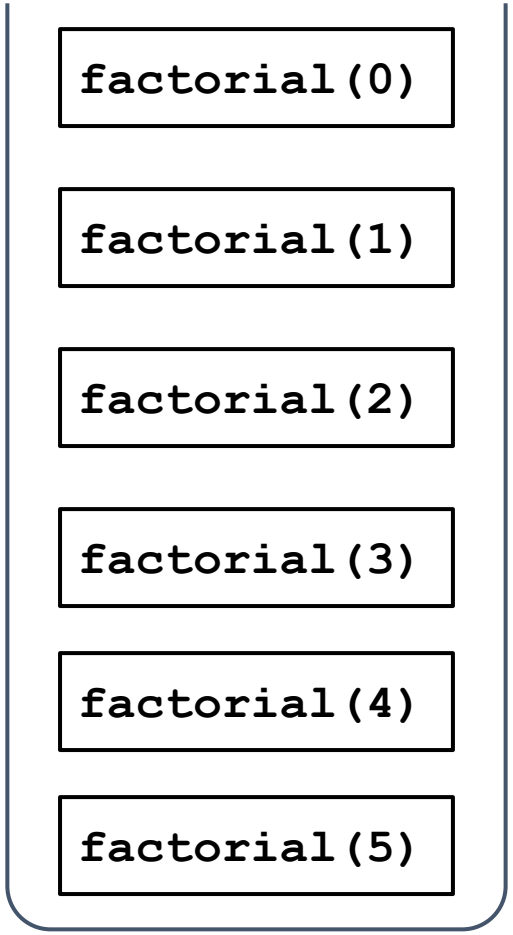
What would happen if you call factorial(5)?



Applications – Function Calls

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

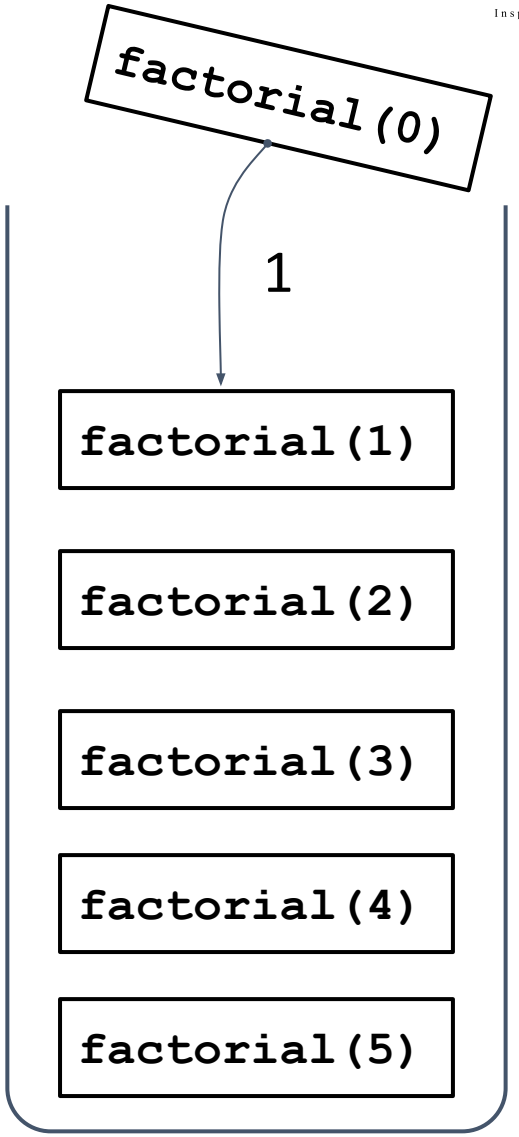
What would happen if you call factorial(5)?



Applications – Function Calls

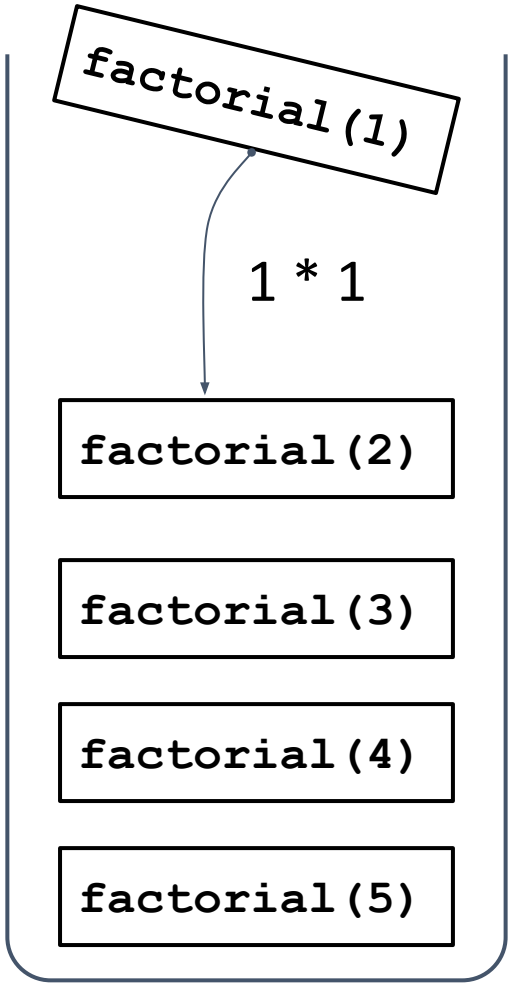
```python
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

What would happen if you call factorial(5)?



Applications – Function Calls

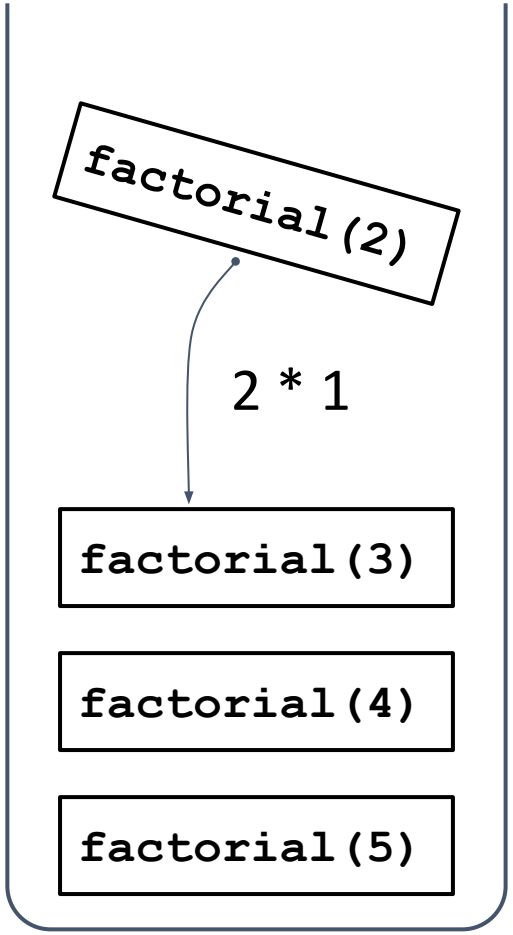
```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```



What would happen if you call factorial(5)?

Applications – Function Calls

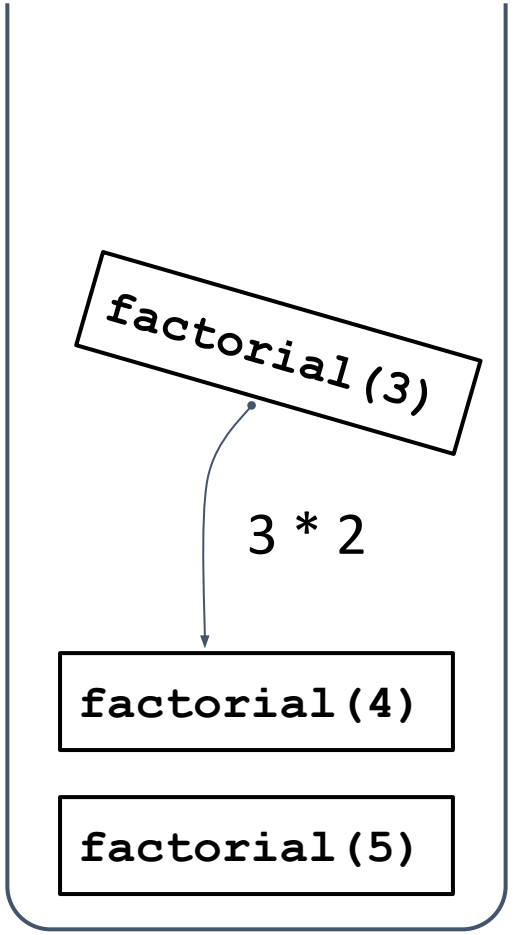
```python
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```



What would happen if you call factorial(5)?

Applications – Function Calls

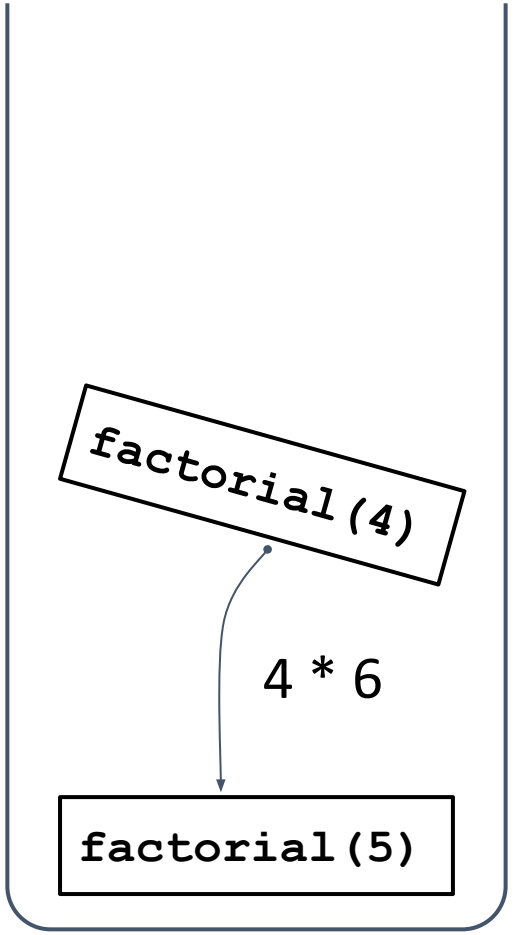
```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```



What would happen if you call factorial(5)?

Applications – Function Calls

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

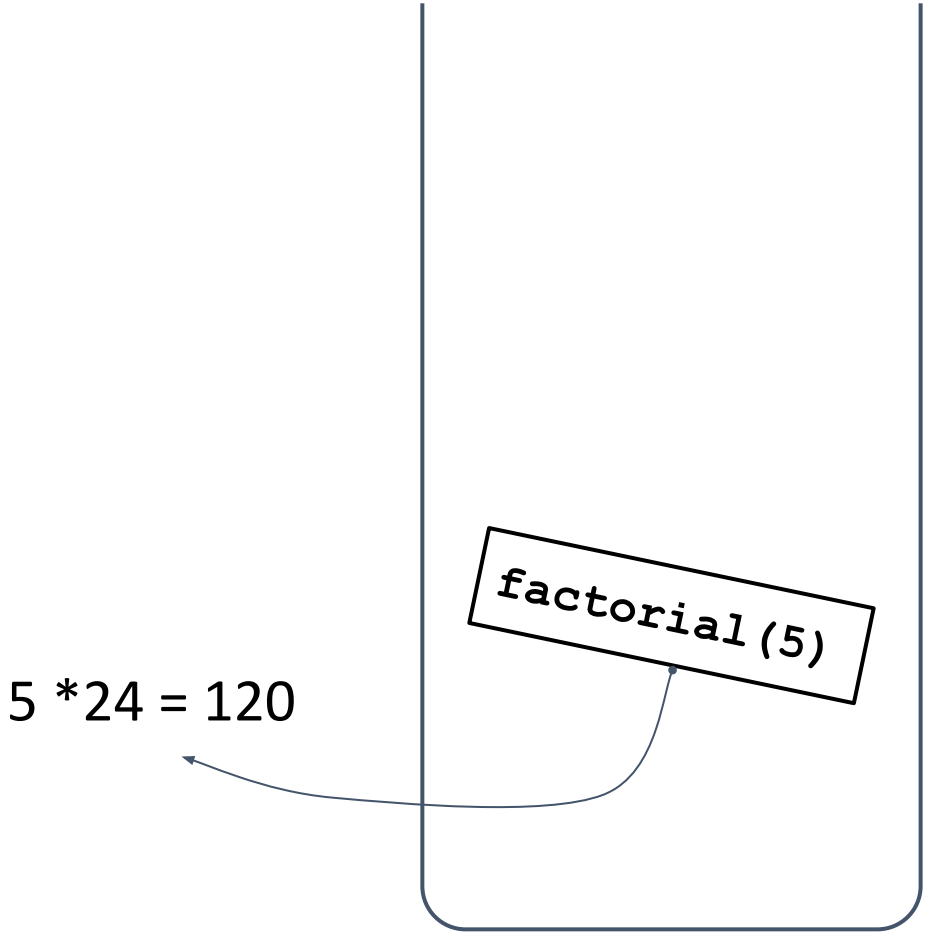


What would happen if you call factorial(5)?

Applications – Function Calls

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

What would happen if you call factorial(5)?



Stack Operations

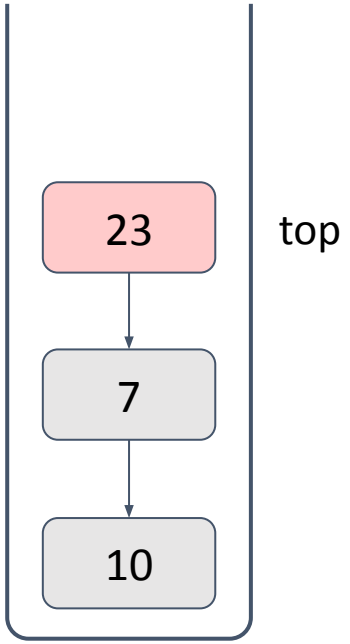
Operation	Python (`list`)	Java (`Stack<E>`)
Push	`stack.append(x)`	`stack.push(x)`
Pop	`stack.pop()`	`stack.pop()`
Peek	`stack[-1]`	`stack.peek()`
Check Empty	`len(stack) == 0`	`stack.empty()`
Size	`len(stack)`	`stack.size()`

Stack Implementation

```java
// Stack class using a singly linked list
class LinkedListStack {
    private Node top; // Points to the top element of the stack

    public LinkedListStack() {
        this.top = null; // Initialize stack as empty
    }
```

Push

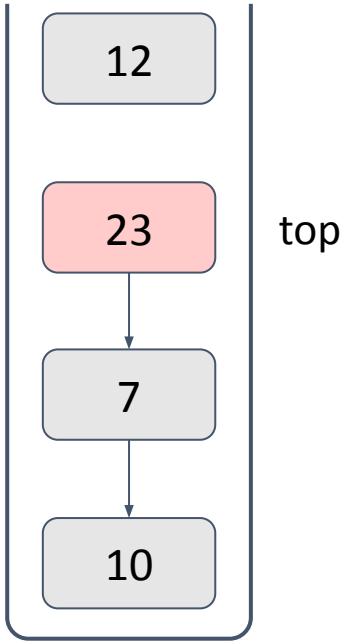
```
// Push operation: Inserts an element at the top of the stack
public void push(int value) {



}
```



push(12)

Push

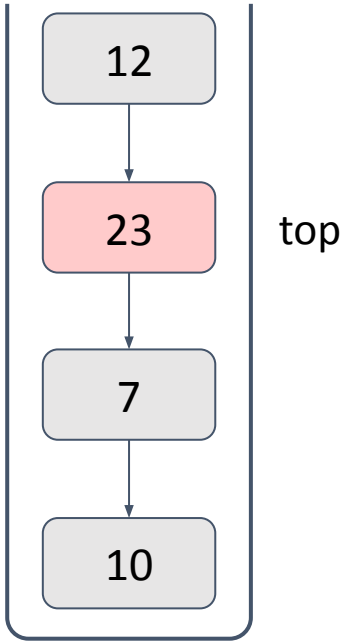
```
// Push operation: Inserts an element at the top of the stack
public void push(int value) {
    Node newNode = new Node(value);


}
```



push(12)

Push

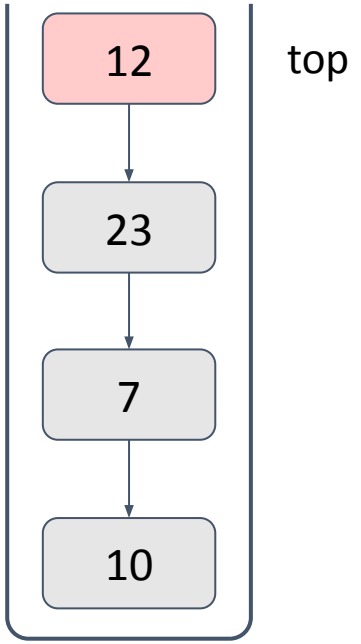
```
// Push operation: Inserts an element at the top of the stack
public void push(int value) {
    Node newNode = new Node(value);
    newNode.next = top; // New node points to the current top

}
```



push(12)

Push

```java
// Push operation: Inserts an element at the top of the stack
public void push(int value) {
    Node newNode = new Node(value);
    newNode.next = top; // New node points to the current top
    top = newNode; // Update top to the new node
}
```

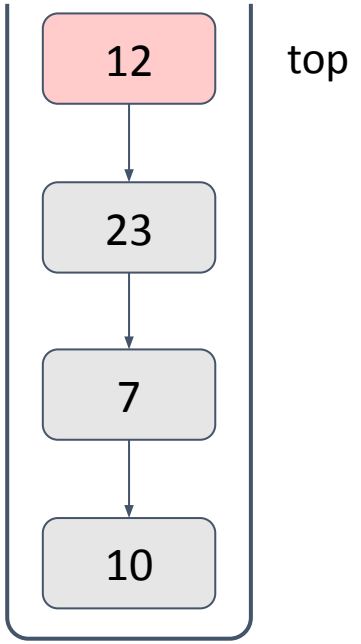


push(12)

Pop

```
// Pop operation: Removes and
// returns the top element of the stack
public int pop() {



}
```

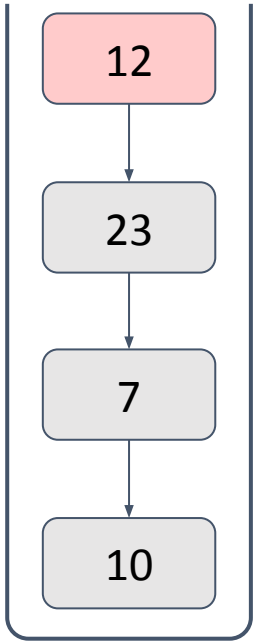
pop()



Pop

```
// Pop operation: Removes and
// returns the top element of the stack
public int pop() {



    int poppedValue = top.elem;



}
```

poppedValue = 12

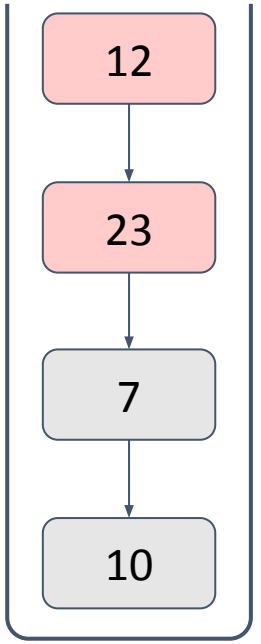


pop()

Pop

```
// Pop operation: Removes and
// returns the top element of the stack
public int pop() {



    int poppedValue = top.elem;
    top = top.next; // Move top pointer to the next node

}
```

poppedValue = 12



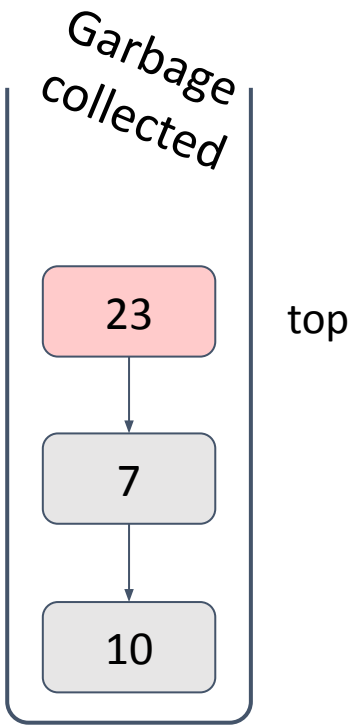
pop()

Pop

```java
// Pop operation: Removes and
// returns the top element of the stack
public int pop() {



    int poppedValue = top.elem;
    top = top.next; // Move top pointer to the next node
    return poppedValue;
}
```

pop()

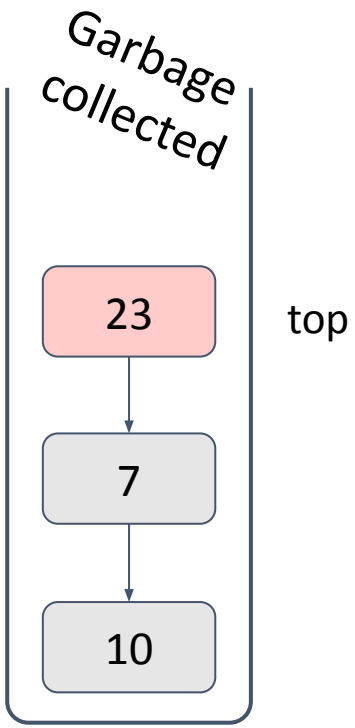
poppedValue = 12



Pop

```java
// Pop operation: Removes and
// returns the top element of the stack
public int pop() {
    if (isEmpty()) {
        return -1; // Return -1 if stack is empty
    }
    int poppedValue = top.elem;
    top = top.next; // Move top pointer to the next node
    return poppedValue;
}
```

poppedValue = 12



pop()

Peek

```
// Peek operation: Returns the top element without removing it
public int peek() {
    if (isEmpty()) {
        return -1; // Return -1 if stack is empty
    }
    return top.elem;
}
```

isEmpty

```
// isEmpty operation: Checks if the stack is empty
public boolean isEmpty() {
    return top == null;
}
```

Example: Expression Evaluation

- Expression can be written in various ways
- For this example, we will focus on postfix notation

`2 * 3 + 4` ⇒ Infix notation
`2 3 * 4 +` ⇒ Postfix notation
`+ * 2 3 4` ⇒ Prefix notation

Example: Expression Evaluation

- Imagine you have a calculator where you **first enter the numbers** and **then specify the operation**.
- Example: Instead of writing 3 + 4, you write 3 4 +. This means, "Take 3 and 4, then add them."
- No parentheses needed: (3 + 4) * 2 becomes 3 4 + 2 *

Postfix Expression

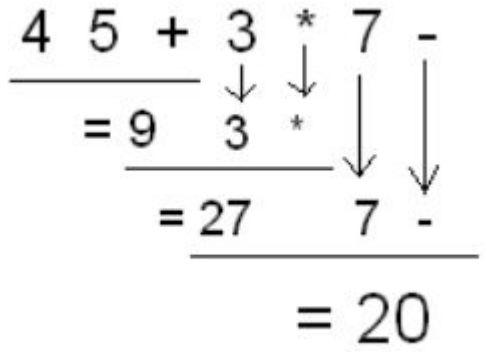
Infix	Postfix
(4 + 5) * 3 - 7	4 5 + 3 * 7 -

Postfix Expression

Infix

(4 + 5) * 3 - 7

Postfix



Walkthroughs

Expression: **3 4 +**

- Push 3 → Stack: [3]
- Push 4 → Stack: [3, 4]
- See +: Pop 4 and 3, add → 3 + 4 = 7
- Push 7 back → Stack: [7]
- Final result: **7**

Walkthroughs

Expression: **5 1 2 + 4 * + 3 -**

- Push 5 → [5]
- Push 1 → [5, 1]
- Push 2 → [5, 1, 2]
- + → Pop 2 & 1, compute 1 + 2 = 3, push 3 → [5, 3]
- Push 4 → [5, 3, 4]
- * → Pop 4 & 3, compute 3 * 4 = 12, push 12 → [5, 12]
- + → Pop 12 & 5, compute 5 + 12 = 17, push 17 → [17]
- Push 3 → [17, 3]
- - → Pop 3 & 17, compute 17 - 3 = 14, push 14 → [14]
- Final result: **14**

Postfix Evaluation

```
public static int evaluate(String expression) {
    LinkedListStack stack = new LinkedListStack();
    String[] tokens = split(expression);
    for (String token : tokens) {
        if (isNumber(token)) {  // If it's a number, push it
            stack.push(Integer.parseInt(token));
        } else {  // If it's an operator, pop two values and apply the operation
            int b = stack.pop();
            int a = stack.pop();
            switch (token) {
                case "+": stack.push( value: a + b); break;
                case "-": stack.push( value: a - b); break;
                case "*": stack.push( value: a * b); break;
                case "/": stack.push( value: a / b); break;
            }
        }
    }
    return stack.pop();  // Final result
}
```

Postfix Evaluation

```
def evaluate_postfix(expression):
    stack = []
    tokens = expression.split()

    for token in tokens:
        if token.isdigit() # Check for numbers
            stack.append(int(token))
        else:  # Operator case
            b = stack.pop()
            a = stack.pop()
            if token == '+':
                stack.append(a + b)
            elif token == '-':
                stack.append(a - b)
            elif token == '*':
                stack.append(a * b)
            elif token == '/':
                stack.append(a / b)
    return stack.pop()
```

Parenthesis Matching

- Parenthesis matching checks if an expression has balanced parentheses, meaning every opening (, {, or [has a corresponding closing), }, or] in the correct order.

Example Valid Expressions

✅ `"()"`

✅ `"( [ { } ] )"`

✅ `"{[()()]}"`

Example Invalid Expressions

× `"(]"` → Mismatched parentheses

× `"((())"` → Unmatched `(`

× `"{[)]}"` → Incorrect order

Parenthesis Matching

```java
public static boolean isValid(String expression) {
    Stack<Character> stack = new Stack<>();
    char[] characters = expression.toCharArray();

    for (char ch : characters) {
        if (ch == '(' || ch == '{' || ch == '[') {
            stack.push(ch);
        } else if (ch == ')' || ch == '}' || ch == ']') {
            if (stack.isEmpty()) return false;  // No opening bracket
            char open = stack.pop();
            if (!matches(open, ch)) return false;  // Mismatched brackets
        }
    }
    return stack.isEmpty();  // Stack should be empty if balanced
}
```

```java
private static boolean matches(char open, char close) {
    return (open == '(' && close == ')') ||
           (open == '{' && close == '}') ||
           (open == '[' && close == ']');
}
```

Parenthesis Matching

```python
def is_valid(expression):
    stack = []
    matching = {')': '(', '}': '{', ']': '['}

    for ch in expression:
        if ch in "({[":
            stack.append(ch)
        elif ch in ")}]":
            if not stack or stack.pop() != matching[ch]:
                return False  # Mismatch or missing opening bracket

      return len(stack) == 0  # Stack should be empty if balanced

# Example usage
expr = "{[()]}"
print("Valid:", is_valid(expr))  # Output: True
```