香港中文大學(深圳)
The Chinese University of Hong Kong

CSC3100 Data Structures
Lecture 9: Stack

Yixiang Fang
School of Data Science (SDS)
The Chinese University of Hong Kong, Shenzhen

Outline

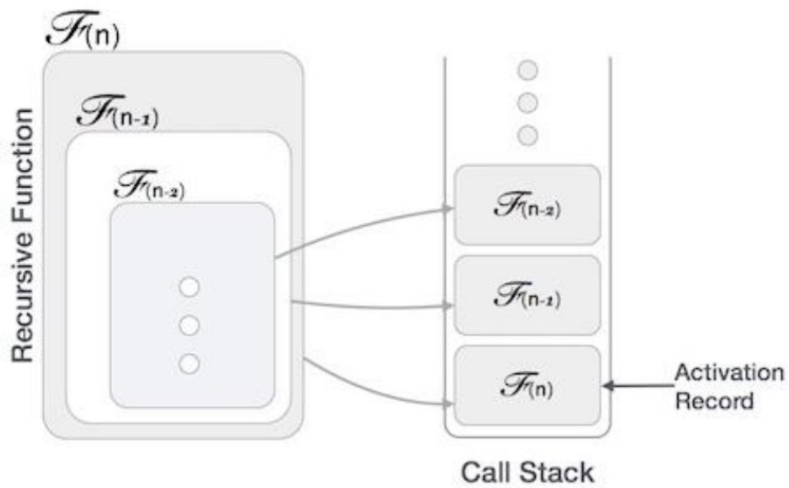
- Stack
  - Examples and definitions
  - Last-In-First-Out (LIFO) property
- Stack implementations
  - Linked list
  - Array
- Stack applications
  - Balance symbol checking
  - Evaluation of expressions

Motivating examples

- Plates and bowls

- Program codes
  - Call functions, e.g., recursive functions

```java
public class test {
    public static void main(String[] args) {
        StringBuffer sb = new StringBuffer("Hello ");
        System.out.println("before change, sb is "+sb.toString());
        change(sb);
        System.out.println("after change, sb is "+sb.toString());
    }
    public static void change(StringBuffer stringBuffer){
        stringBuffer = new StringBuffer("Hi ");
        stringBuffer.append("world !");
    }
}
```



Stack

- A stack stores a set S of elements that have two constrained updates:
  - Push(e): add a new element to S
  - Pop( ): removes the most recently added element from S
- Stack follows: Last-In-First-Out (LIFO) property
  - We can only add/remove/examine from one end (called the "top")
  - Consider the trays/dishes in canteens

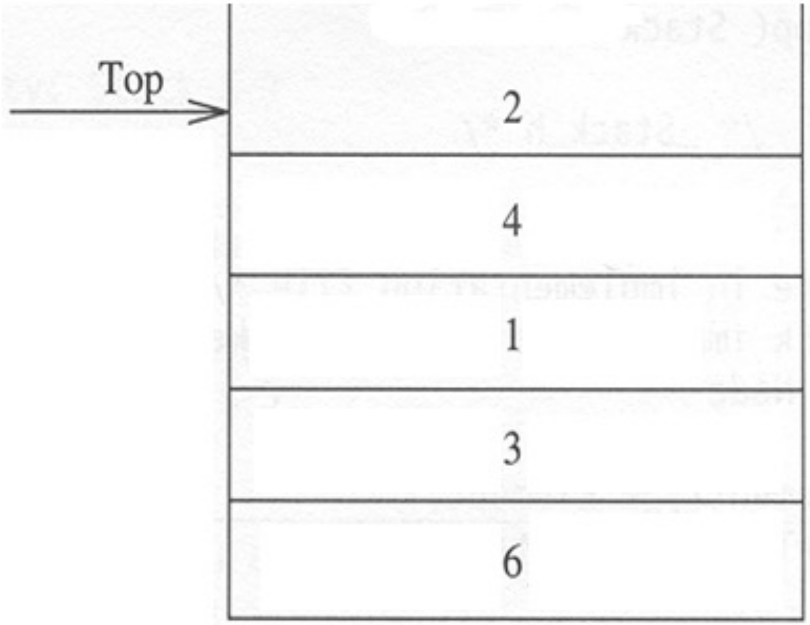
Stack

- Access from the top

- Basic operations
  - pop ()
  - push (i)

  - makeEmpty ()
  - top ()
  - isEmpty()

- Implementations
  - Linked list
  - Arrays



Implementation of Stack using linked list

```
class Node {
        Node next;
        Object element;
}

class Stack {
        Node head;
}
```

- Create an empty stack

```
public Stack() {        //constructor
        this.head = new Node();
}
```

- Push onto a stack

```
void push(Object x) {
        Node tmpNode = new Node();
        tmpNode.element = x;
        tmpNode.next = head.next;
        head.next = tmpNode;
}
```

Implementation of Stack using linked list

- Pop from a stack

```
public Object pop() {
        Node firstNode = null;
        if (isEmpty()) {
                return null;
        } else {
                firstNode = head.next;
                head.next = firstNode.next;
                return firstNode.element;
        }
}
```

- Return top element in a stack

```
public Object top( ) {
        if (!isEmpty())
                return head.next.element;
        else {
                return null;
        }
}
```

Implementation of Stack using array

- Stack class

```
class Stack {
        final static int MIN_STACK_SIZE = 5;
        int topOfStack = -1;
        Object[ ] array;
}
```

- Construction method of Stack class

```
public Stack (int maxElements) {
        int capacity = maxElements;

        if (maxElements < MIN_STACK_SIZE)
                capacity = MIN_STACK_SIZE;

        array = new Object[capacity];
}
```

Implementation of Stack using array

- Test for full stack

```
public boolean isFull() {
        return (topOfStack == array.length - 1);
}
```

- Push an element onto the stack

```
public boolean push(Object x) {
        if (isFull())
                return false;
        else
                array[++topOfStack] = x;
        return true;
}
```

Implementation of Stack using array

- Pop element from stack

```
public Object pop() {
        if (!isEmpty())
                return array[topOfStack--];
        else
                return null;
}
```

- Return top of stack

```
public Object top() {
        if (!isEmpty())
                return array[topOfStack];
        else
                return null;
}
```

Comparison of these two implementations

- Using list saves space

- Using array is faster. Why?
  - Two reasons:
    - Memory allocation
    - Continuous memory can be loaded into cache

Applications of Stack (i)

- Balanced symbol checking
  - In programing languages, there are many instances when symbols must be balanced
    - E.g., { } , [ ] , ( )
  - Stack can be used for checking if the symbols are balanced
    - Balanced
      - ( ) { [ ] }
      - ({ { } })
      - ({ [ ] })
    - Unbalanced
      - ( ]
      - ( ) { ([ ]) }]
      - ( ) [[ ] { }

C code example

```c
int sum = 0;
for(int i=0; i<n; i++){
    sum += array[i];
}
return sum;
```

Balanced symbol checking

- Observation
  - If the next symbol is the opening symbol, e.g., (, [, {
    - It will not result in unbalanced symbols
  - If the next symbol is the closing symbol, e.g.,),], }
    - It must match the last symbol
    - E.g., if the next symbol is), then the last symbol must be (

Balanced symbol checking algorithm

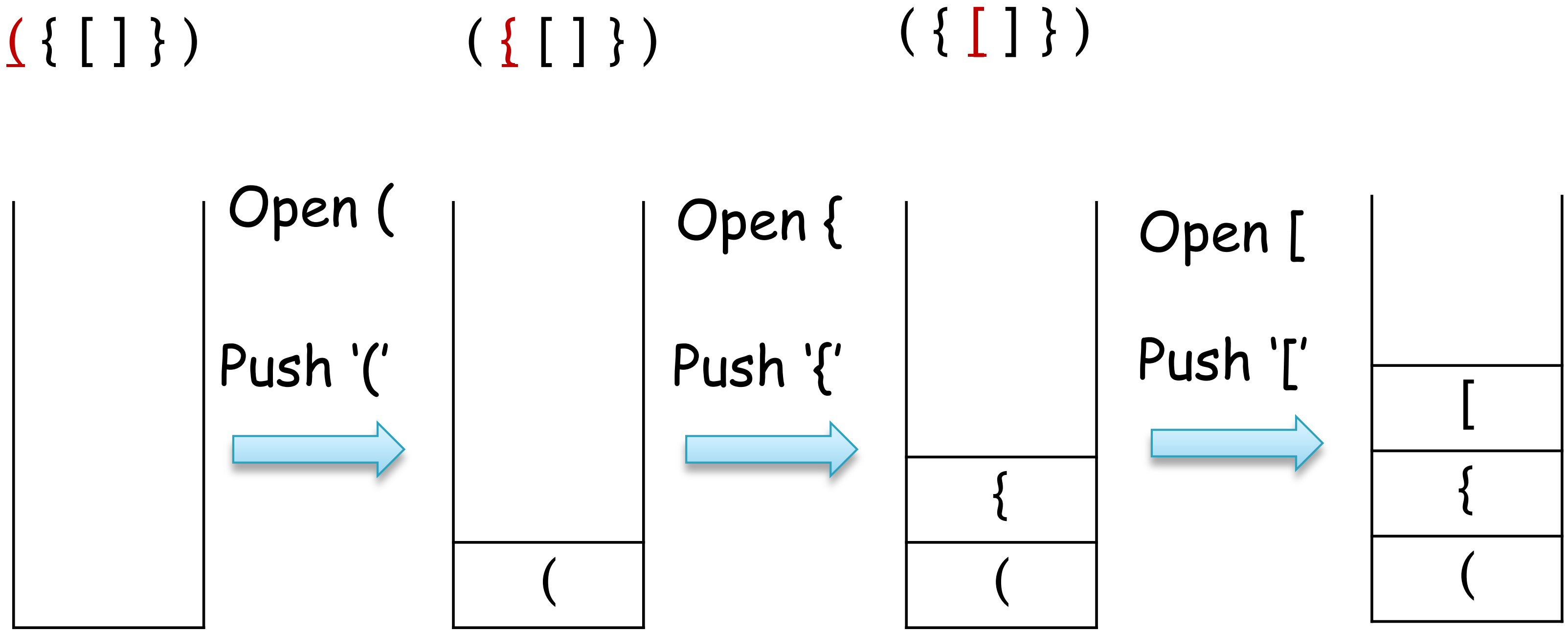
- Step 1: make an empty stack

- Step 2: read the symbols from the input text
  - If the symbol is an opening symbol, push it onto the stack
  - If it is a closing symbol
    - If the stack is empty, return false
    - Otherwise, pop from the stack; if the symbol popped does not match the closing symbol, return false

- Step 3: at the end, if the stack is not empty, return false (unbalanced), else return true (balanced)

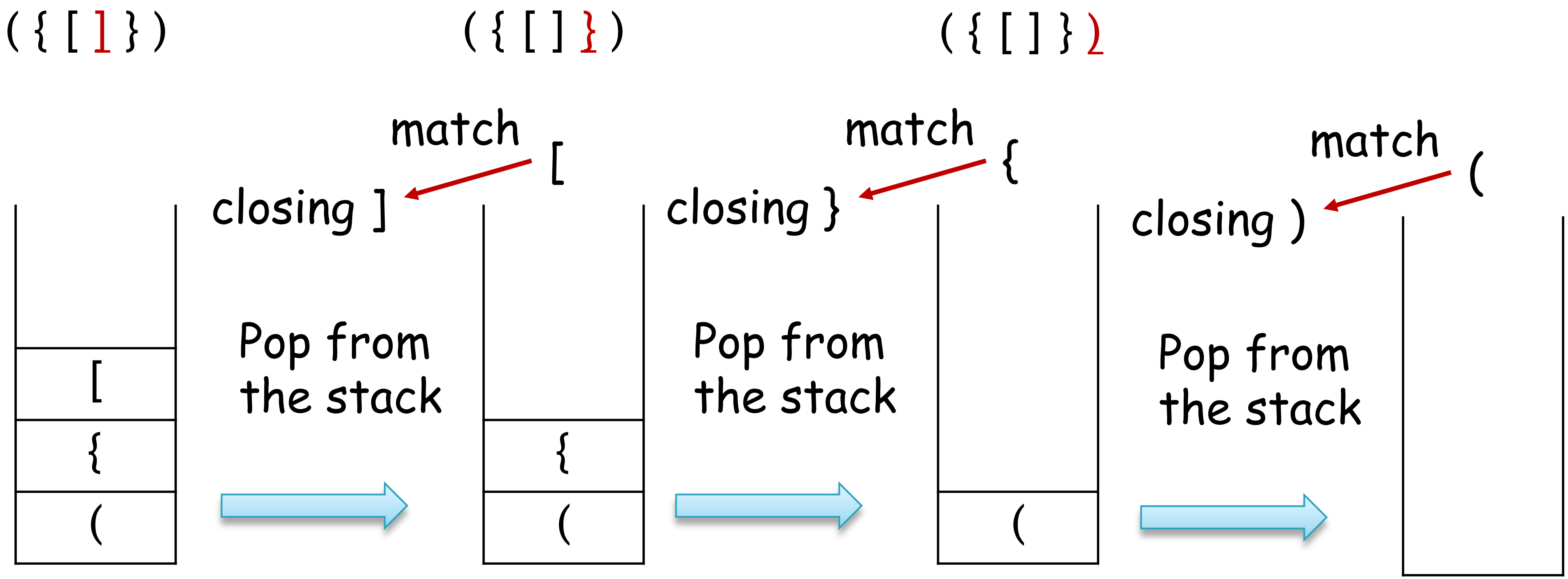
A running example

- Given an input symbol list: ({ [ ] }),
  - Check if the symbols are balanced: show the status of the stack after each symbol checking



A running example (cont.)

- Given an input symbol list: ({ [ ] }),
  - Check if the symbols are balanced: Show the status of the stack after each symbol checking



  - After checking all symbols, the stack is empty: return true

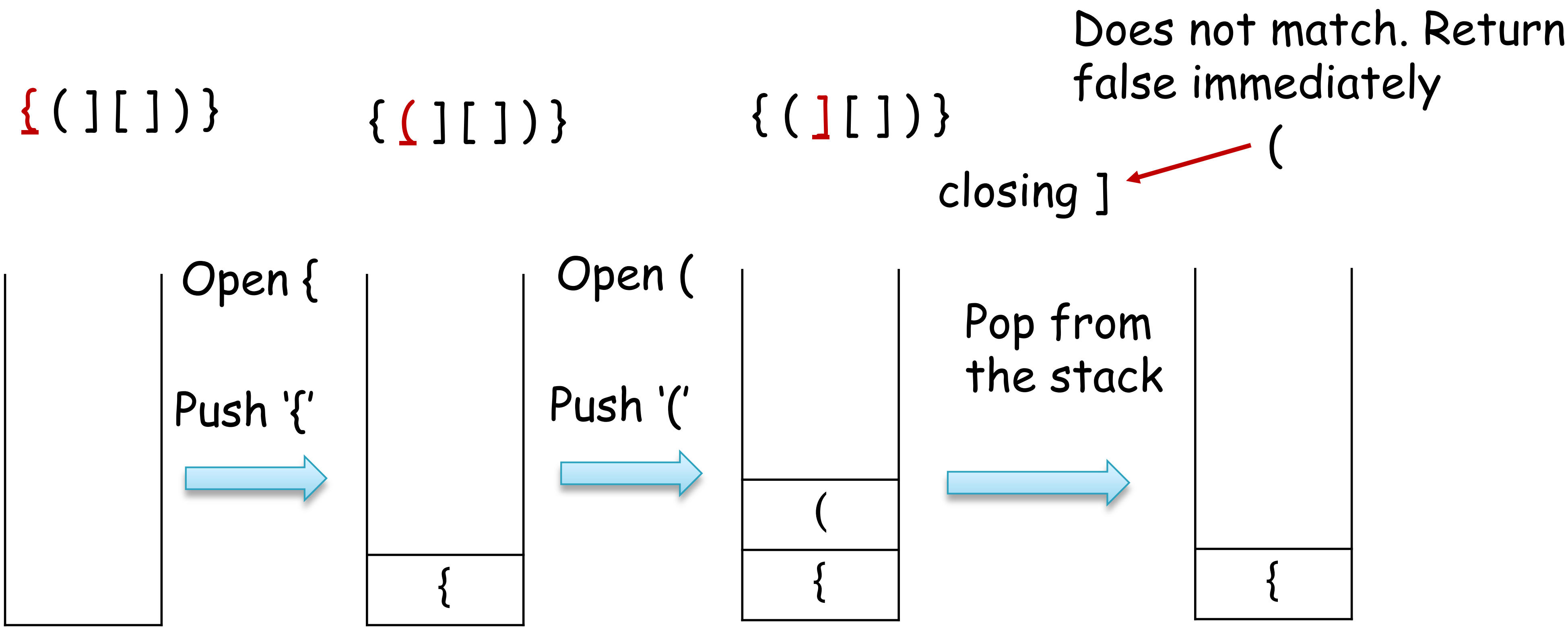
Practice

- Given an input symbol list: { ( ] [ ]) },
  - Check if the symbols are balanced
  - Show the status of the stack after each symbol checking

- Given an input symbol list: ( ) [[ ] { },
  - Check if the symbols are balanced
  - Show the status of the stack after each symbol checking

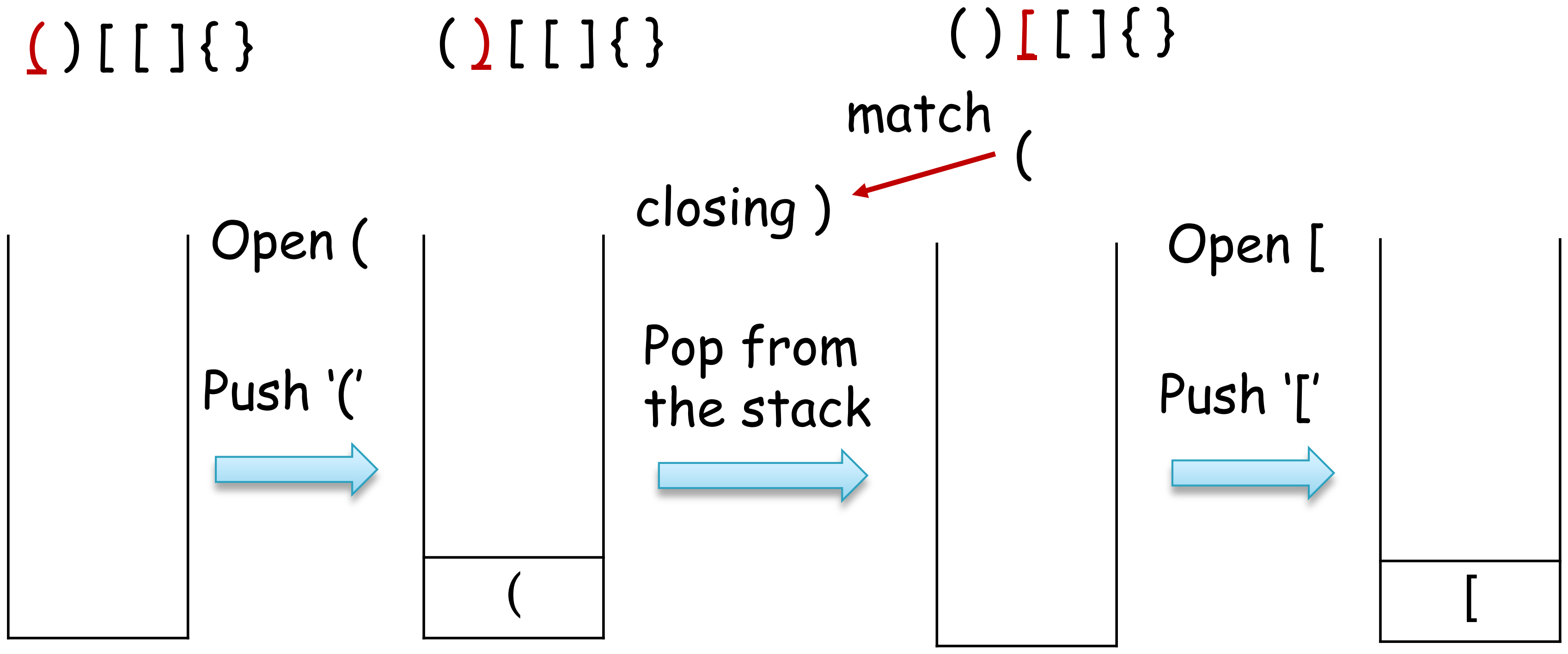
Practice

- Check if the symbol list { ( ] [ ]) } is balanced
  - Show the status of the stack after each symbol checking



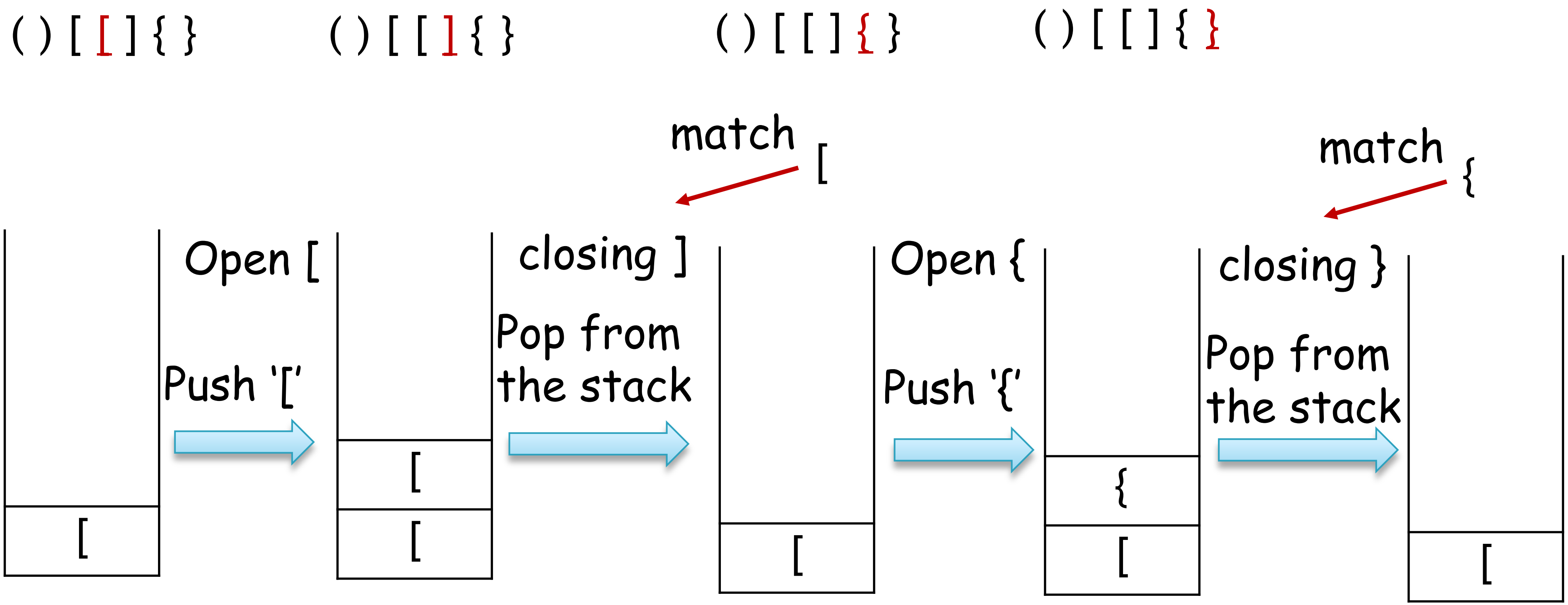
Practice

- Check if the symbol list ( ) [[ ] { } is balanced
  - Show the status of the stack after each symbol checking



Practice (Cont.)

- Check if the symbol list ( ) [[ ] { } is balanced
  - Show the status of the stack after each symbol checking



  - Finally, the stack is not empty, so return false

Application of Stack (ii)

- Evaluation of expressions
  - How we usually write expressions?
    - a/b – c+d*e - a*c
  - Examining the above expression, we see that they have:
    - Operators: +, -, *, /
    - Operands: a, b, c, d, e
  - How the expressions are interpreted?
    - Precedence rule + associative rule

Expressions: what we learnt

- Precedence of operators: the order in which the operators are performed:
  - Precedence:
    - * and / have the same precedence; + and – have the same precedence
    - * and / have higher predecence than + and -
  - (((a / b) - c) + (d * e)) - (a * c)

- Associative rule of operators:
  - +, -, * and / are left-associative (from left to right)

- Parentheses can be used to override precedence:
  - Expressions are always evaluated from the innermost parenthesized expression, e.g., a * (b + c)

Representations of expressions

- Consider the four binary operators +, -, * and /
- The standard way (of writing expressions): **Infix Expressions**
  - A binary operator is placed in-between its two operands
  - Con: need to use parentheses and precedence rules to evaluate expressions
- When a program executes an expression: **Postfix Expressions**
  - Each operator appears after its operands
  - Pro: precedence has been considered when the postfix expression is generated, so no parentheses

We leave a space here to distinguish two operands 2 and 3 and one operand 23

Infix	Postfix
2 + 3 * 4	2 3 4 * +
2 * 3 + 4	2 3 * 4 +
2 * 3 * 4	2 3 * 4 *
(2 + 3) * 4	2 3 + 4 *
a / b - c + d * e - a * c	a b / c - d e * + a c * -

How to derive the postfix?

- 7/(2+3)*4
  - According to the definition, operator should appear after operands 7/(2+3) and 4 should be put before *, so the postfix L for the expression is:
    - L: “Postfix for 7/(2+3)” 4 *
  - We got a smaller problem. What is the postfix L’ for 7/(2+3)?
    - 7 and postfix of (2+3) should appear before /
    - L’: 7 “postfix for (2+3)” /
  - What is the postfix L” for (2+3)?
    - 2 3 +
  - => Postfix for L’ is: 7 2 3 + /
  - => Postfix for L is: 7 2 3 + / 4 *

Practice

- What is the postfix expression for the following expression?
  - 2*(3+2*4)
  - Hint: operator should appear after operand. 2 and (3+2*4) are the operand of *, so they should appear before *

Answer: The operand of * is 2 and (3+2*4), let's first denote the postfix expression of (3+2*4) as x. Then, the postfix expression will be 2 x *. Now consider 3+2*4. The operand of + is 3 and 2*4.

Let's denote the postfix of 2*4 as y, and the postfix expression of 3+2*4 becomes 3 y +. Now consider the postfix expression of 2*4, we know it is 2 4 * according to its definition. So y=2 4 *. As x = 3 y +, putting y to the equation, we have x = 3 2 4 * +.

Putting back x to the postfix expression, we have the postfix expression of 2*(3+2*4) is: 2 3 2 4 * + *.

Evaluating postfix expression

- We can use the previous recursive idea to derive the postfix expression

- Given a postfix expression
  - How to evaluate the postfix expression?

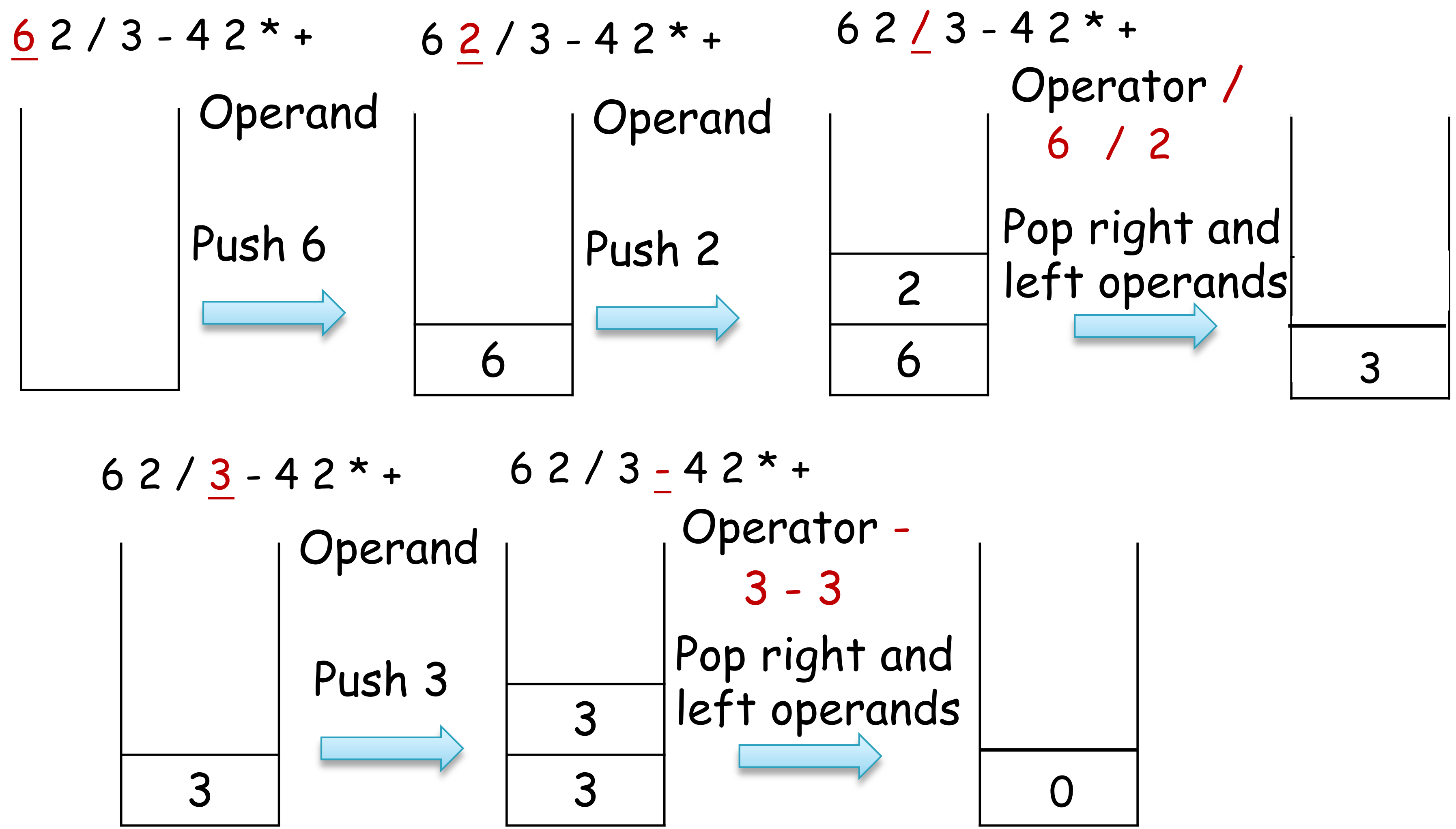
Postfix evaluation algorithm

- Algorithm steps (consider operators +, -, * and /)
  - Create an empty stack
  - Scan the postfix expression from left to right
    - If an operand is encountered, push to the stack
    - If an operator is encountered
      - Pop the stack for the right-hand operand
      - Pop the stack for the left-hand operand
      - Apply the operator to the two operands
      - Push the result onto the stack
  - When the postfix expression has been scanned, the result is kept on the top of the stack

A running example

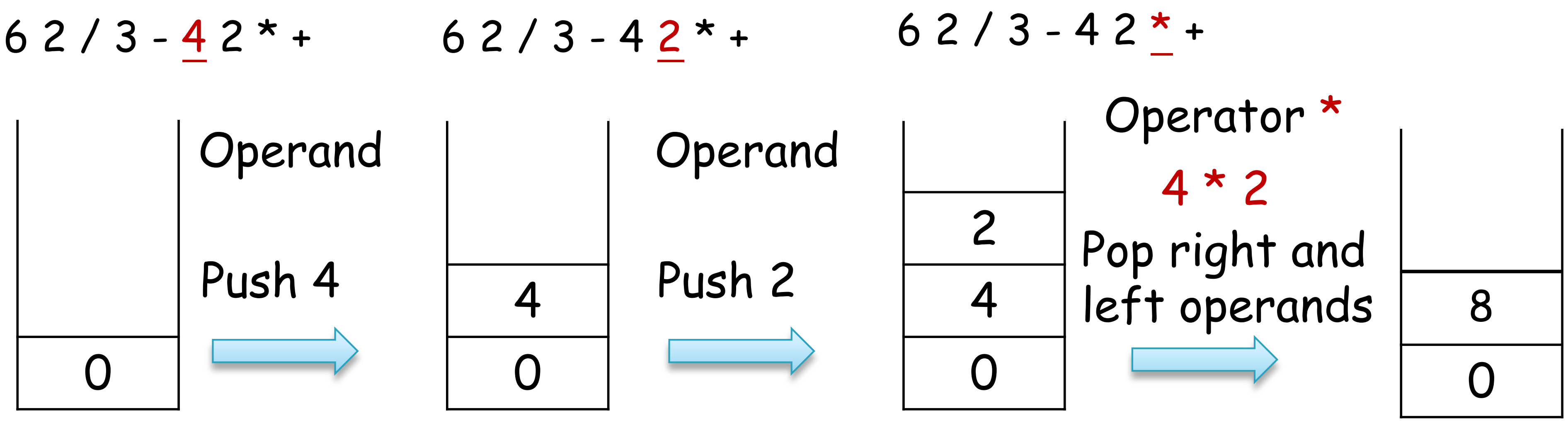
Infix expression: (6/2-3) + (4*2)

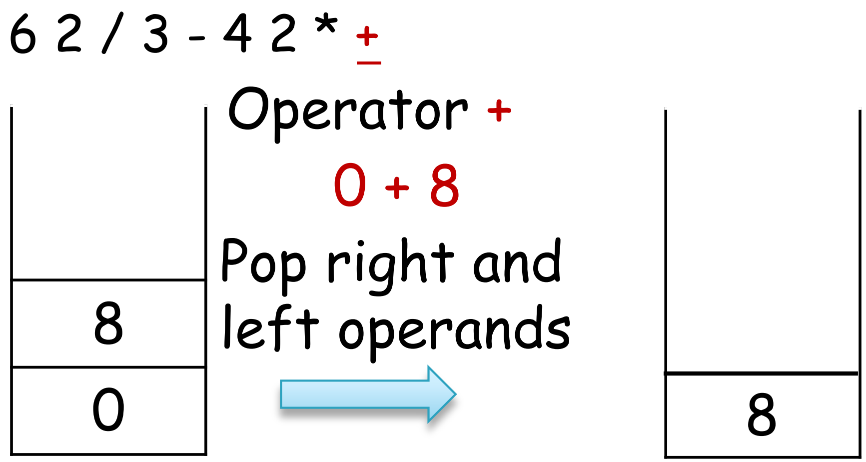
- Evaluate the postfix expression: 6 2 / 3 - 4 2 * +



A running example (cont.)

- Evaluate the postfix expression: 6 2 / 3 - 4 2 * +

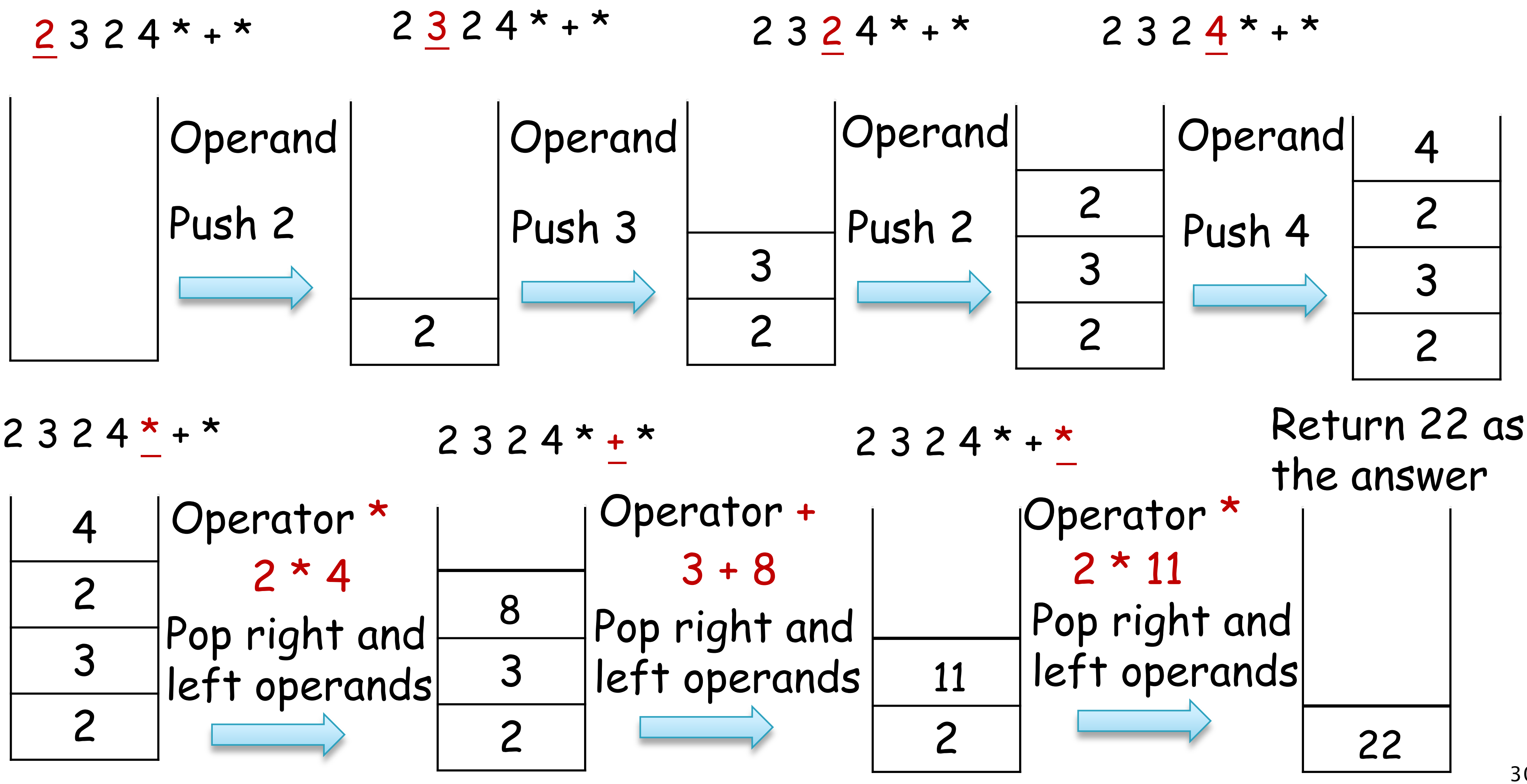




Return 8 as the answer

Practice

- Evaluate the postfix expression: 2 3 2 4 * + *



Recommended reading

- Reading
  - Chapter 10, textbook

- Next lectures
  - Queue: chapter 10, textbook