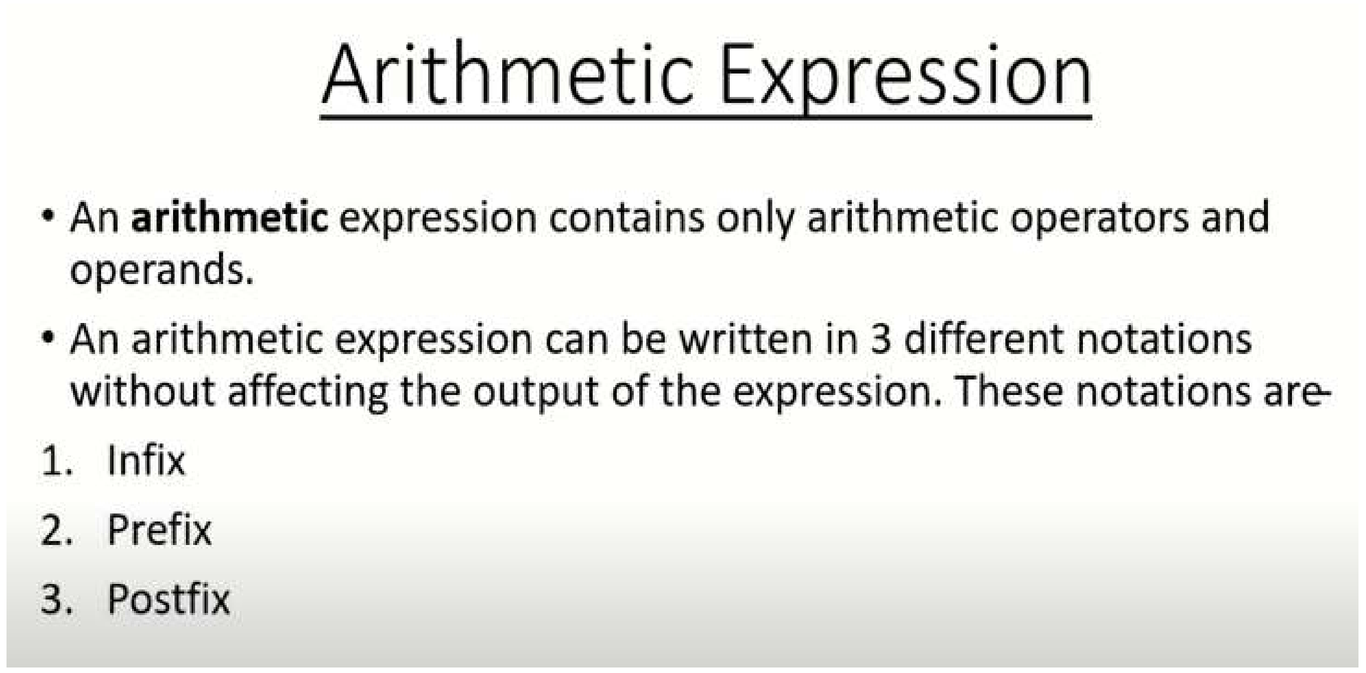
23ITT201 – DATA STRUCTURES

UNIT 2

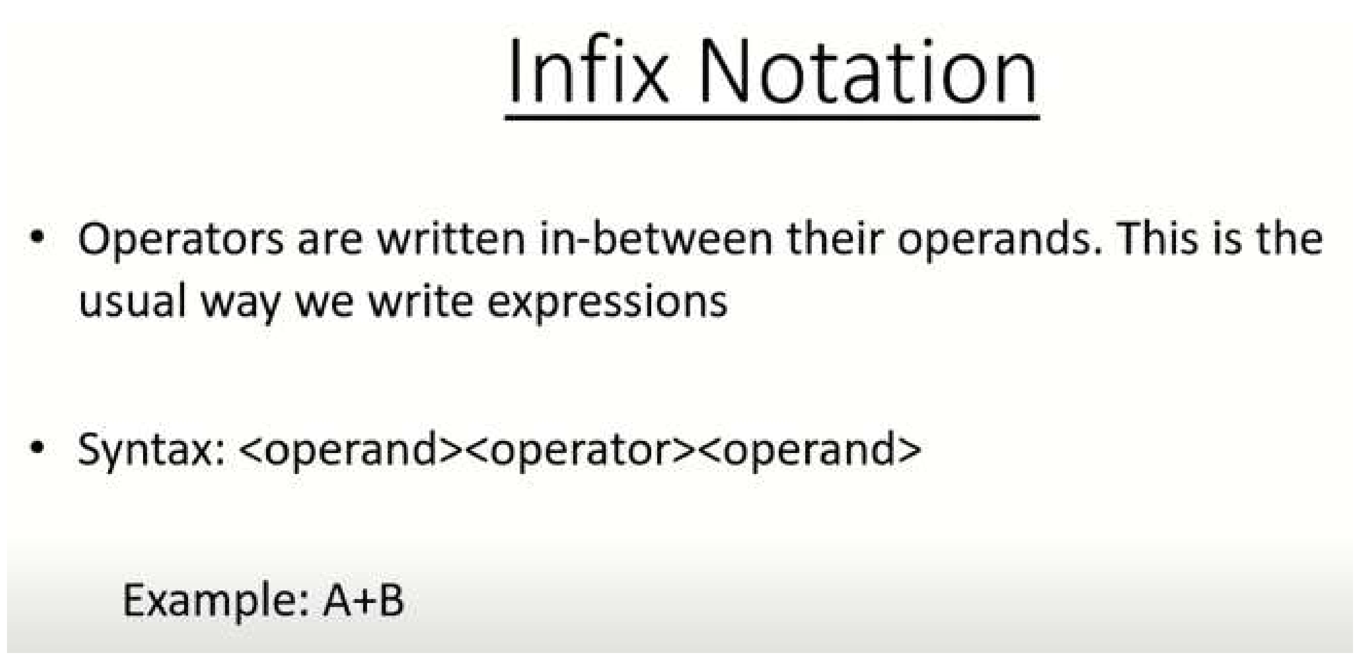
INFIX TO POSTFIX CONVERSION USING STACK



Arithmetic Expression

- An **arithmetic** expression contains only arithmetic operators and operands.
- An arithmetic expression can be written in 3 different notations without affecting the output of the expression. These notations are-

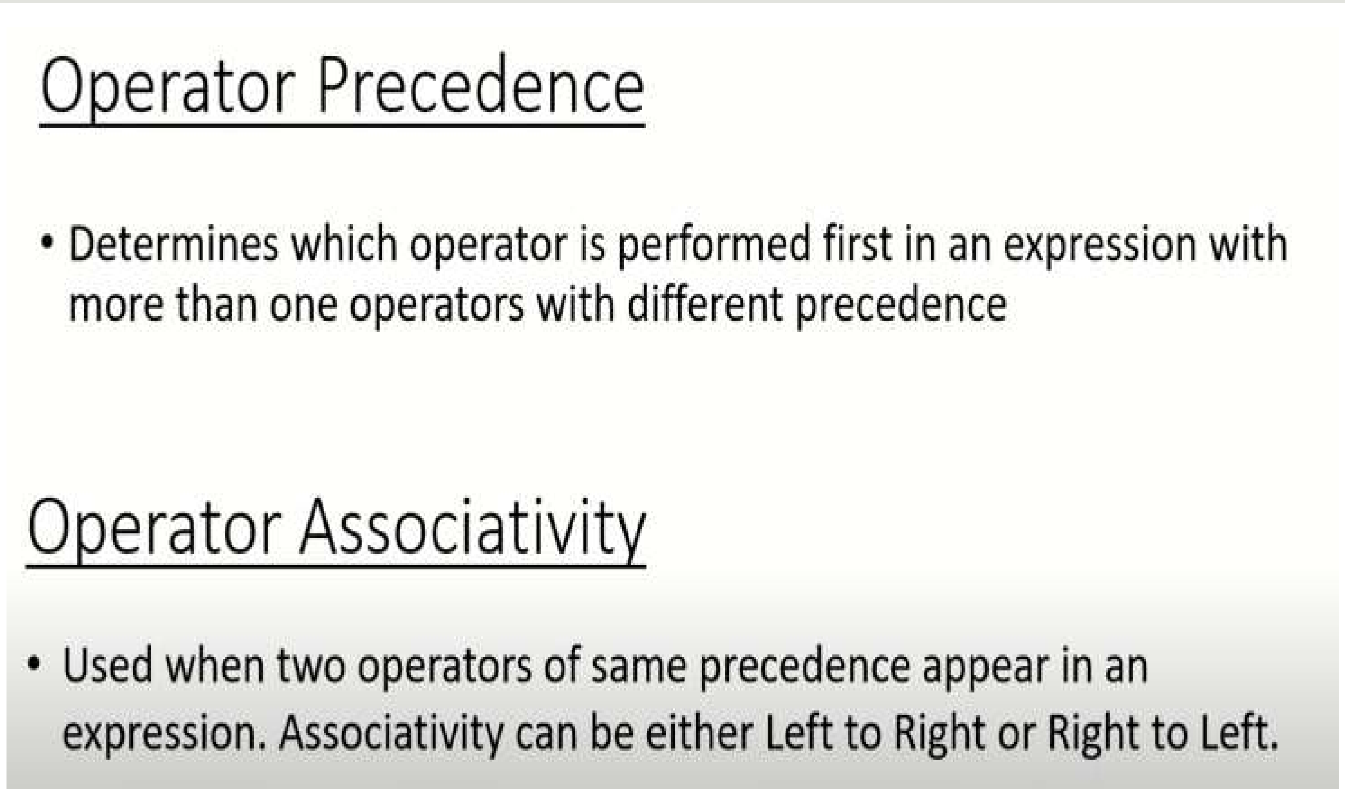
1. Infix
2. Prefix
3. Postfix



Infix Notation

- Operators are written in-between their operands. This is the usual way we write expressions
- Syntax: <operand><operator><operand>

  Example: A+B

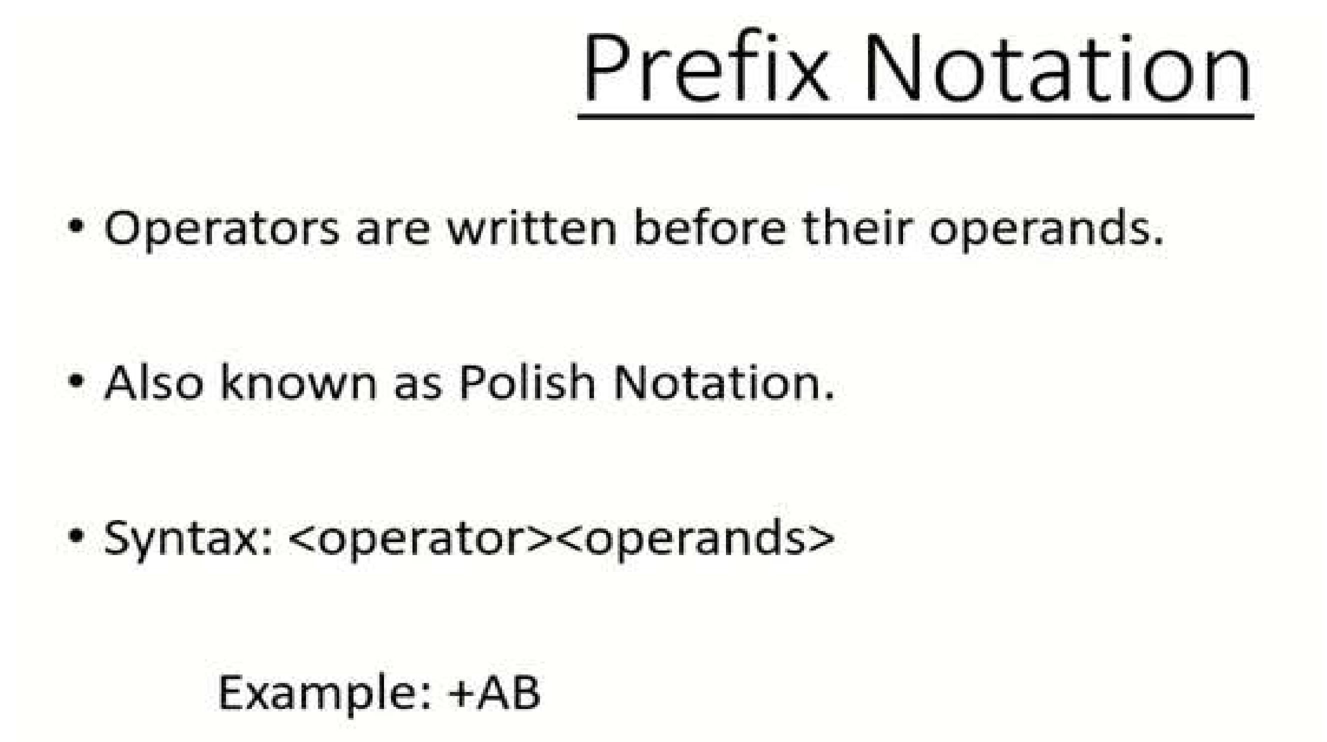


Operator Precedence

- Determines which operator is performed first in an expression with more than one operators with different precedence

Operator Associativity

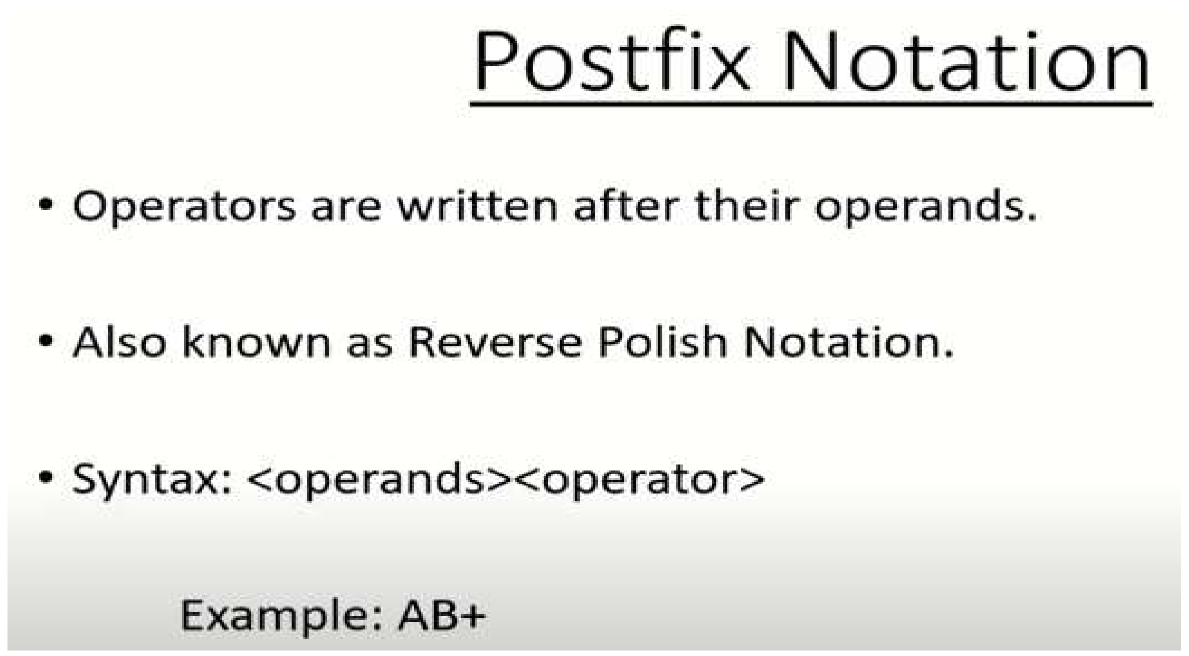
- Used when two operators of same precedence appear in an expression. Associativity can be either Left to Right or Right to Left.



Prefix Notation

- Operators are written before their operands.
- Also known as Polish Notation.
- Syntax: <operator><operands>

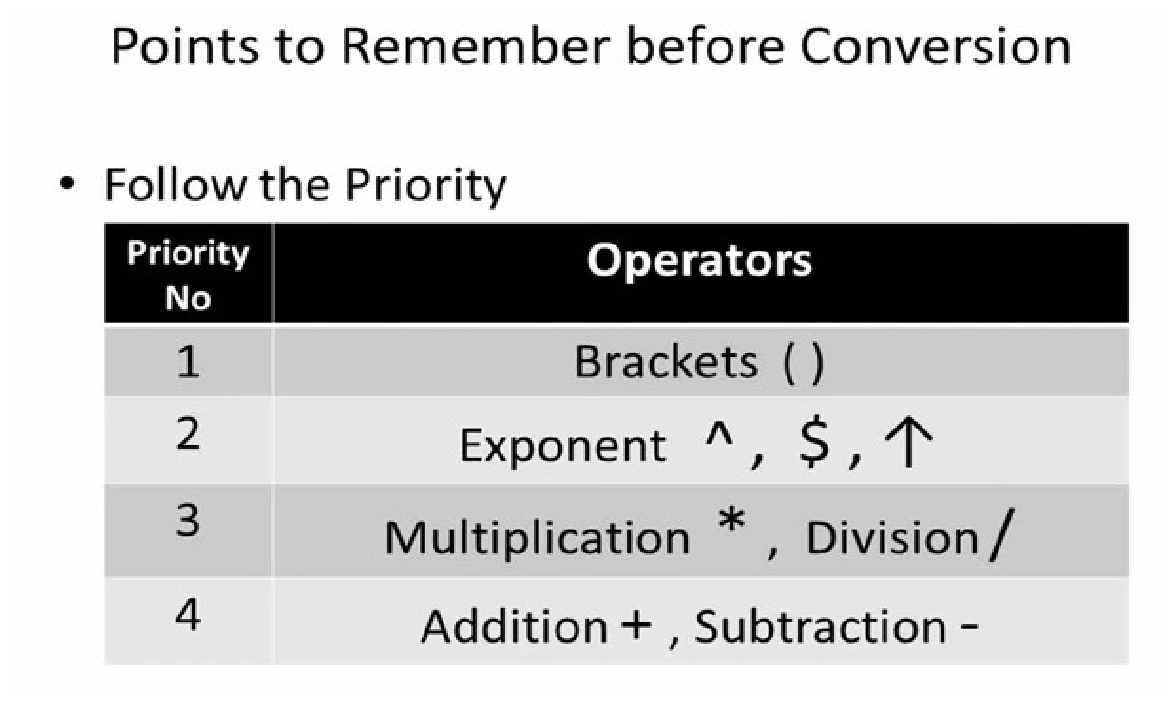
  Example: +AB



Postfix Notation

- Operators are written after their operands.
- Also known as Reverse Polish Notation.
- Syntax: <operands><operator>

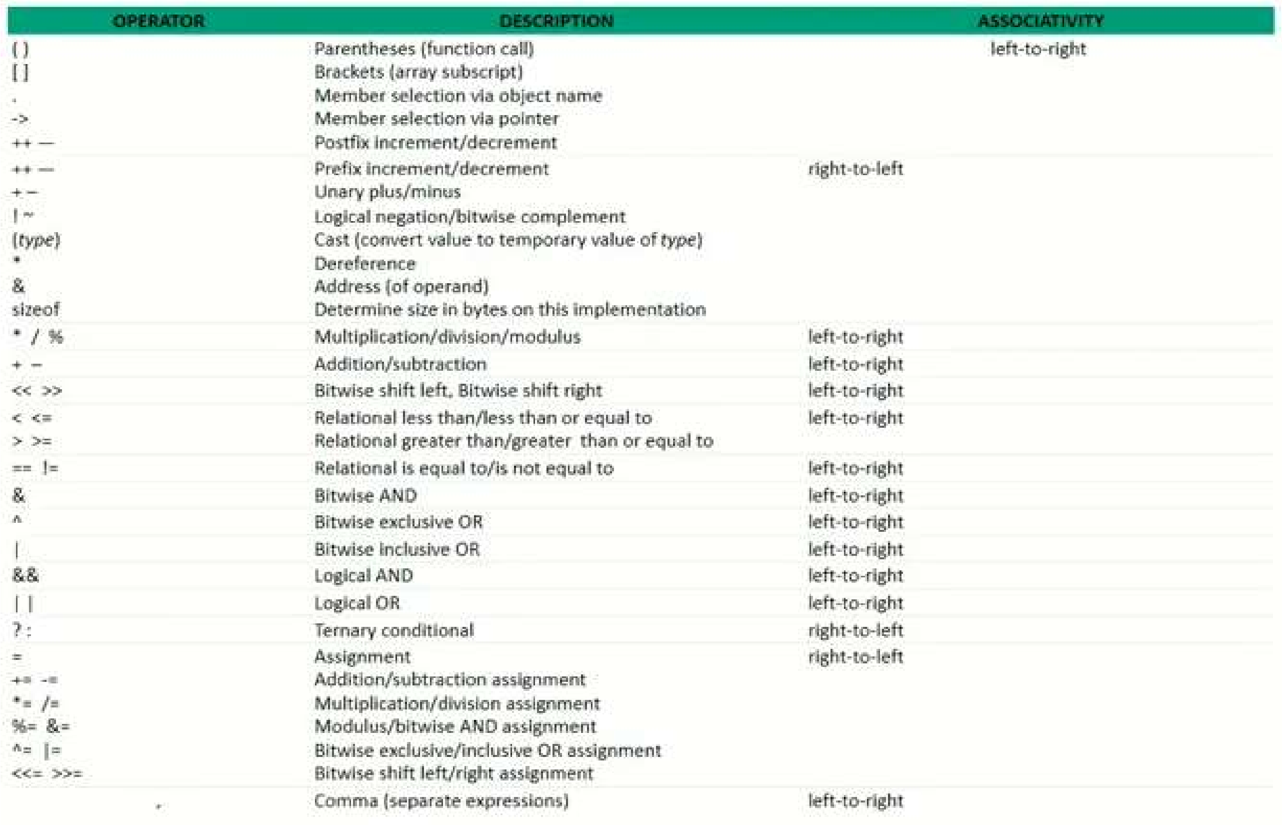
  Example: AB+



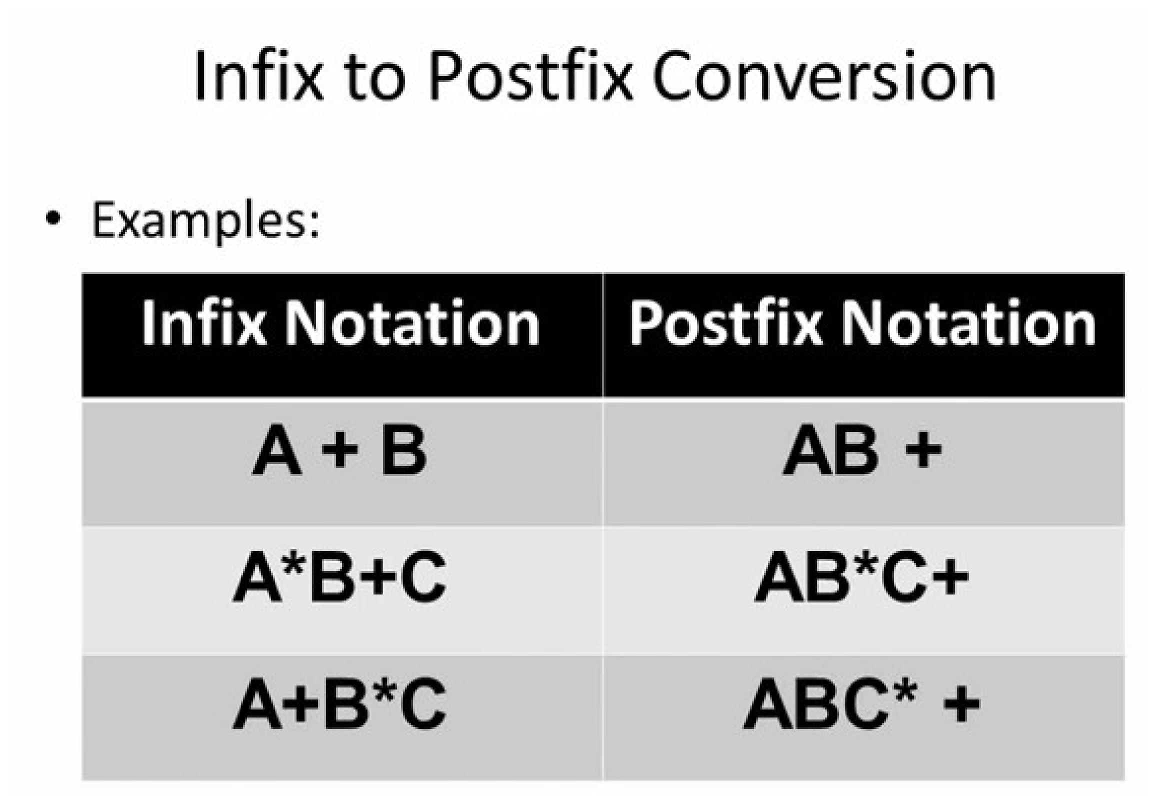
Points to Remember before Conversion

- Follow the Priority

Priority No	Operators
1	Brackets ( )
2	Exponent ^ , $, ↑
3	Multiplication * , Division /
4	Addition + , Subtraction -



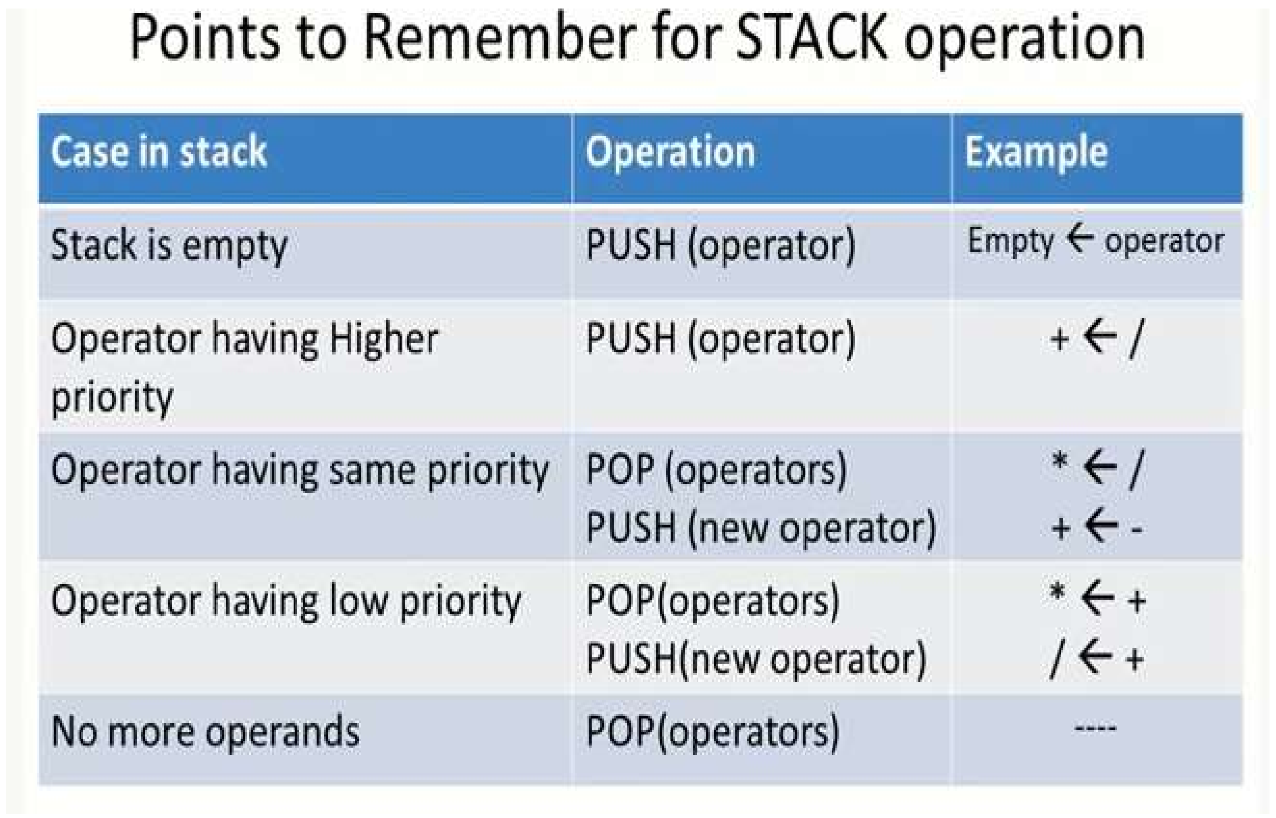
OPERATOR	DESCRIPTION	ASSOCIATIVITY
( )	Parentheses (function call)	left-to-right
[ ]	Brackets (array subscript)	
.	Member selection via object name	
->	Member selection via pointer	
++ —	Postfix increment/decrement	
++ —	Prefix increment/decrement	right-to-left
+ −	Unary plus/minus	
! ~	Logical negation/bitwise complement	
(*type*)	Cast (convert value to temporary value of *type*)	
*	Dereference	
&	Address (of operand)	
sizeof	Determine size in bytes on this implementation	
* / %	Multiplication/division/modulus	left-to-right
+ −	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <=	Relational less than/less than or equal to	left-to-right
> >=	Relational greater than/greater than or equal to	
== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
|	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
| |	Logical OR	left-to-right
? :	Ternary conditional	right-to-left
=	Assignment	right-to-left
+= -=	Addition/subtraction assignment	
*= /=	Multiplication/division assignment	
%= &=	Modulus/bitwise AND assignment	
^= |=	Bitwise exclusive/inclusive OR assignment	
<<= >>=	Bitwise shift left/right assignment	
,	Comma (separate expressions)	left-to-right



Infix to Postfix Conversion

- Examples:

Infix Notation	Postfix Notation
A + B	AB +
A*B+C	AB*C+
A+B*C	ABC* +

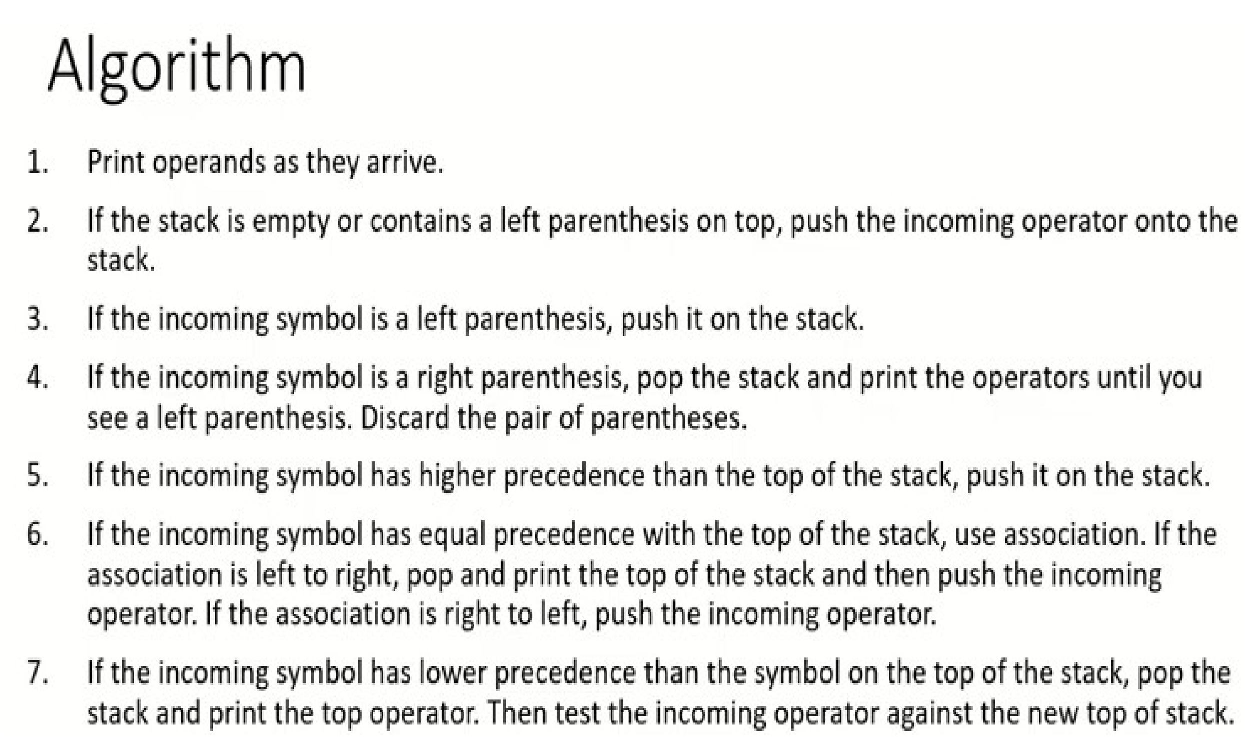


Points to Remember for STACK operation

Case in stack	Operation	Example
Stack is empty	PUSH (operator)	Empty ← operator
Operator having Higher priority	PUSH (operator)	+ ← /
Operator having same priority	POP (operators) PUSH (new operator)	* ← / + ← -

Operator having low priority	POP(operators)
	PUSH(new operator)
		* ← +
		/ ← +

No more operands	POP(operators)	----

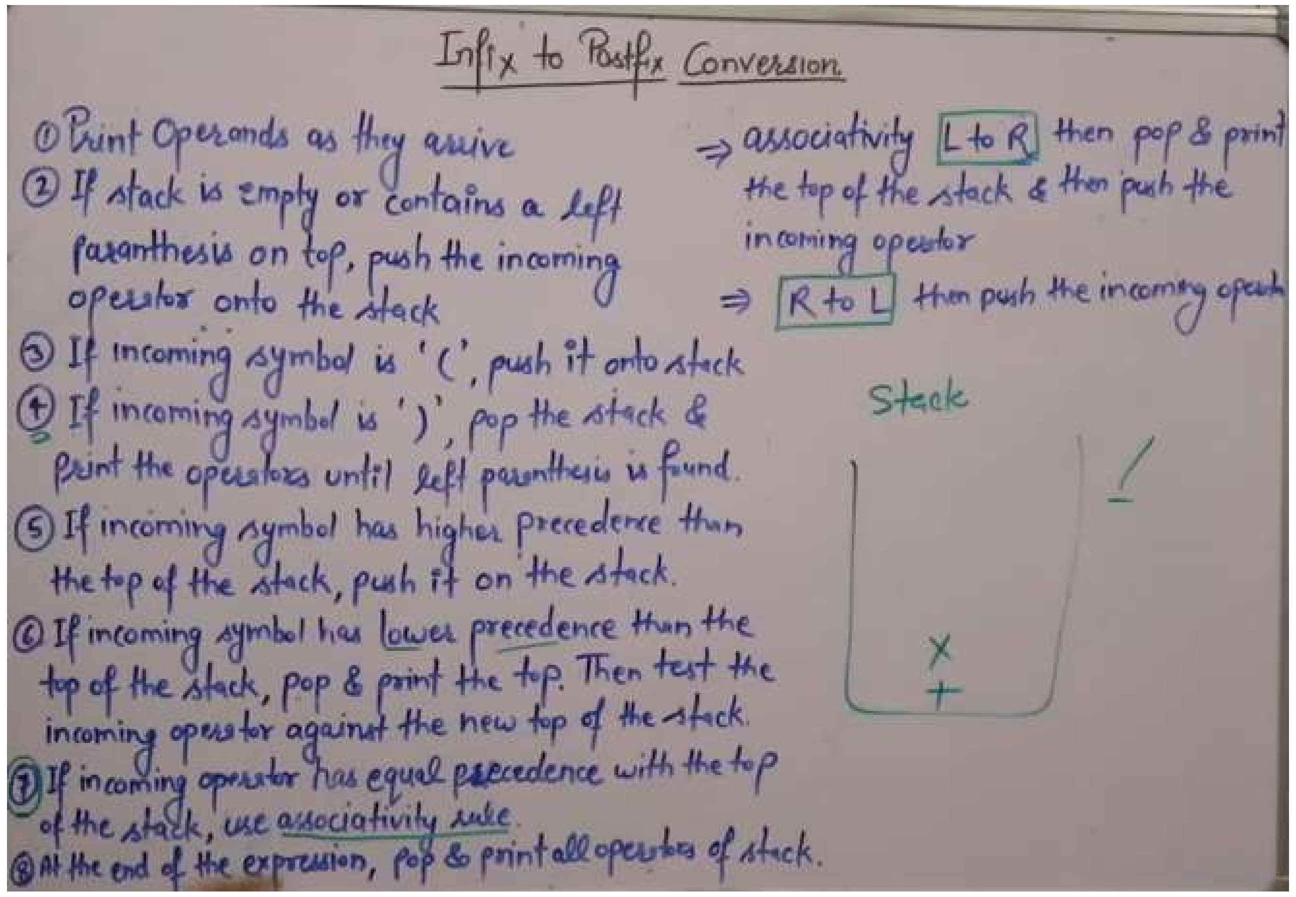


Algorithm

1. Print operands as they arrive.
2. If the stack is empty or contains a left parenthesis on top, push the incoming operator onto the stack.
3. If the incoming symbol is a left parenthesis, push it on the stack.
4. If the incoming symbol is a right parenthesis, pop the stack and print the operators until you see a left parenthesis. Discard the pair of parentheses.
5. If the incoming symbol has higher precedence than the top of the stack, push it on the stack.
6. If the incoming symbol has equal precedence with the top of the stack, use association. If the association is left to right, pop and print the top of the stack and then push the incoming operator. If the association is right to left, push the incoming operator.
7. If the incoming symbol has lower precedence than the symbol on the top of the stack, pop the stack and print the top operator. Then test the incoming operator against the new top of stack.

Advantages of Postfix over Infix

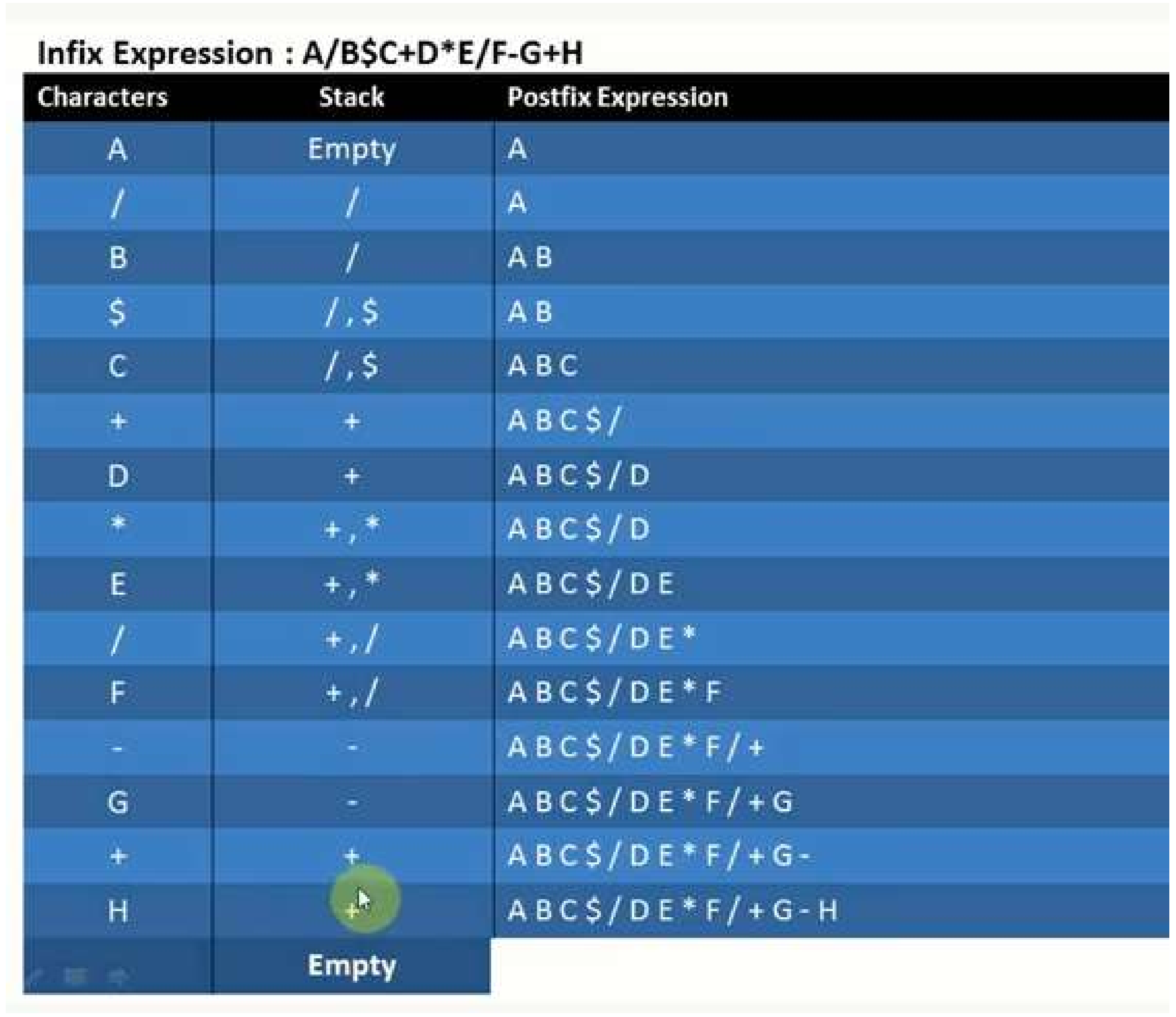
- Any formula can be expressed without advantages.
- It is very convenient for evaluating formulas on computer with stacks.
- Postfix expression doesn't has the operator precedence.
- Postfix is slightly easier to evaluate.
- It reflects the order in which operations are performed.
- You need to worry about the left and right associativity



Infix to Postfix Conversion

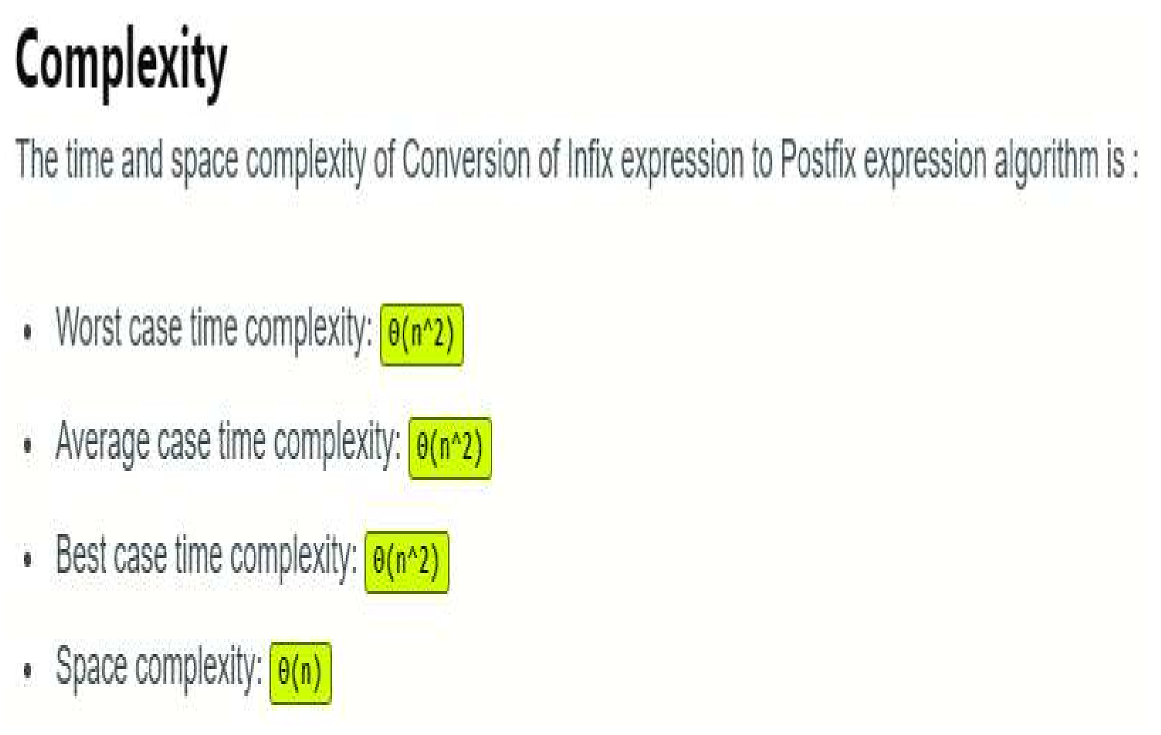
① Print Operands as they arrive
② If stack is empty or contains a left paranthesis on top, push the incoming operator onto the stack
③ If incoming symbol is '(', push it onto stack
④ If incoming symbol is ')', pop the stack & print the operators until left paranthesis is found.
⑤ If incoming symbol has higher precedence than the top of the stack, push it on the stack.
⑥ If incoming symbol has lower precedence than the top of the stack, pop & print the top. Then test the incoming operator against the new top of the stack.
⑦ If incoming operator has equal precedence with the top of the stack, use associativity rule.
⑧ At the end of the expression, pop & print all operators of stack.

⇒ associativity $\boxed{\text{L to R}}$ then pop & print the top of the stack & then push the incoming operator
⇒ $\boxed{\text{R to L}}$ then push the incoming operator



Infix Expression : A/B$C+D*E/F-G+H

Characters	Stack	Postfix Expression
A	Empty	A
/	/	A
B	/	A B
$	/ , $	A B
C	/ , $	A B C
+	+	A B C $ /
D	+	A B C $ / D
*	+ , *	A B C $ / D
E	+ , *	A B C $ / D E
/	+ , /	A B C $ / D E *
F	+ , /	A B C $ / D E * F
-	-	A B C $ / D E * F / +
G	-	A B C $ / D E * F / + G
+	+	A B C $ / D E * F / + G -
H	+	A B C $ / D E * F / + G - H
	Empty	



Complexity

The time and space complexity of Conversion of Infix expression to Postfix expression algorithm is :

- Worst case time complexity: `Θ(n^2)`
- Average case time complexity: `Θ(n^2)`
- Best case time complexity: `Θ(n^2)`
- Space complexity: `Θ(n)`