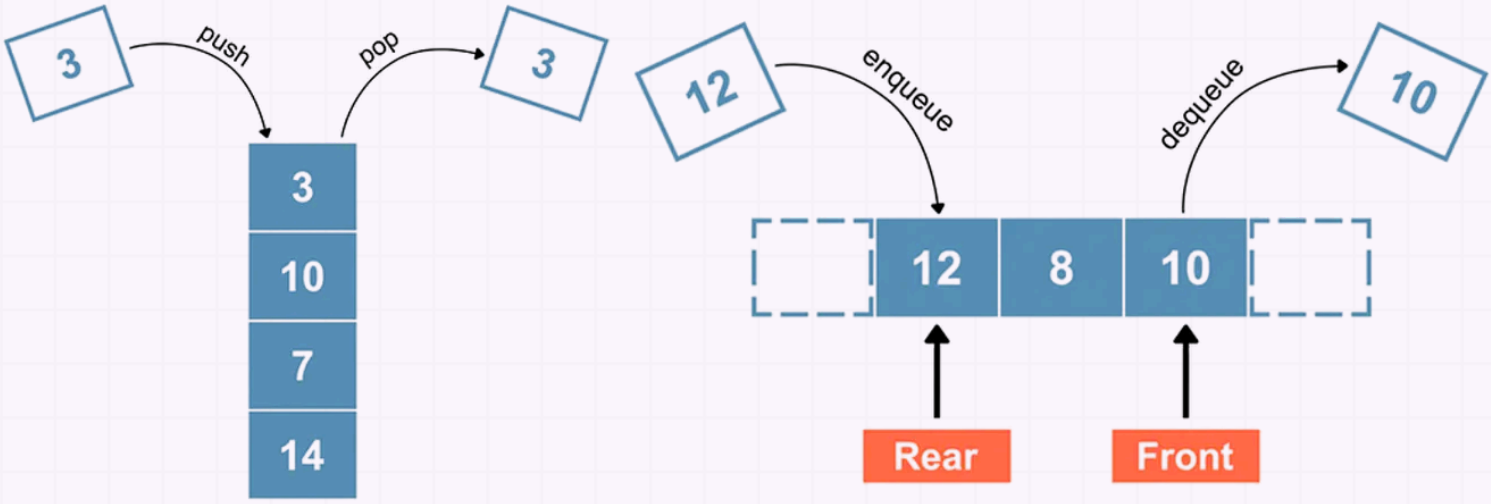
InterviewCafe
Where Learning Never Takes a Break!

Mastering Stacks and Queues

Detailed Explanations with Practice Interview Questions



Santosh Kumar Mishra
Software Engineer at Microsoft • Author • Founder of InterviewCafe

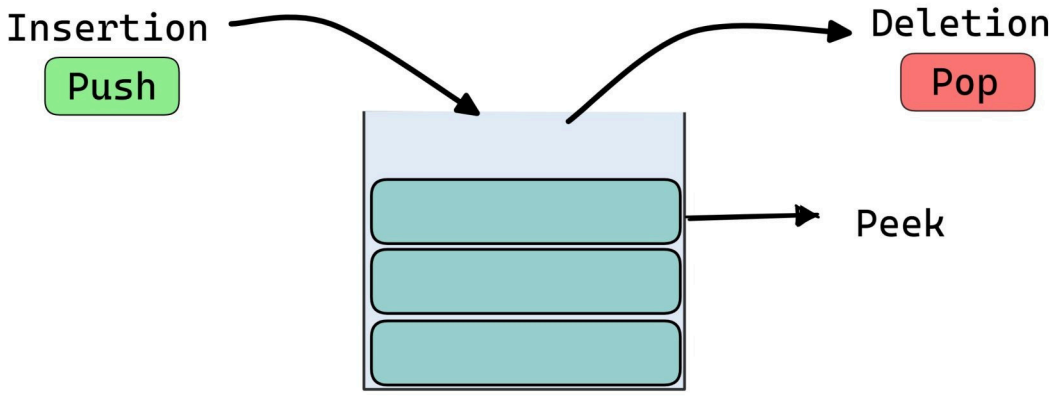
Table of Contents

Stack Introduction

→ A stack is a linear data structure that follows the principle of Last In First Out (LIFO). This means the last element inserted inside the stack is removed first.

- Stack → Linear Data Structure
  - LIFO (Last In First Out)

→ In "Stack" Insertion and Deletion happen from the same end.



• Basic Operations of Stack

→ There are some basic operations that allow us to perform different actions on a stack.

- Push ⟶ Add an element to the top of a stack
- Pop ⟶ Remove an element from the top of a stack
- IsEmpty ⟶ Check if the stack is empty

- IsFull: Check if the stack is full
- Peek: Get the value of the top element without removing it

Applications of Stack Data Structure

→ Although stack is a simple data structure to implement, it is very powerful. The most common uses of a stack are:

To reverse a word

→ Put all the letters in a stack and pop them out. Because of the LIFO order of stack, you will get the letters in reverse order.

In compilers

→ Compilers use the stack to calculate the value of expressions like 2 + 4 / 5 * (7 - 9) by converting the expression to prefix or postfix form.

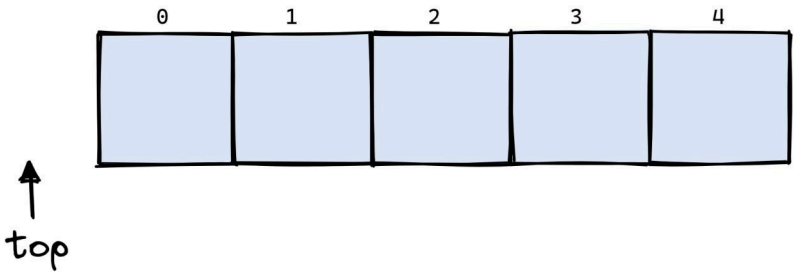
In browsers

→ The back button in a browser saves all the URLs you have visited previously in a stack. Each time you visit a new page, it is added on top of the stack. When you press the back button, the current URL is removed from the stack, and the previous URL is accessed.

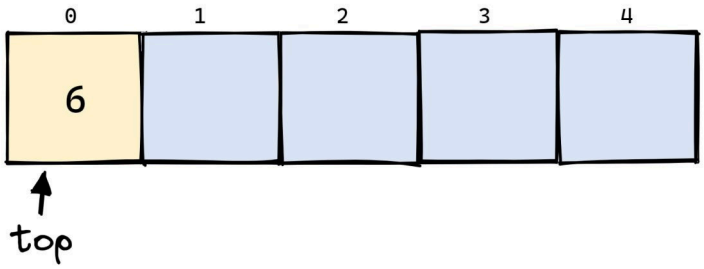
Stack Implementation using Arrays

→ In Stack implementation using arrays, it forms the stack using the arrays. All the operations regarding the stack implementation using arrays.

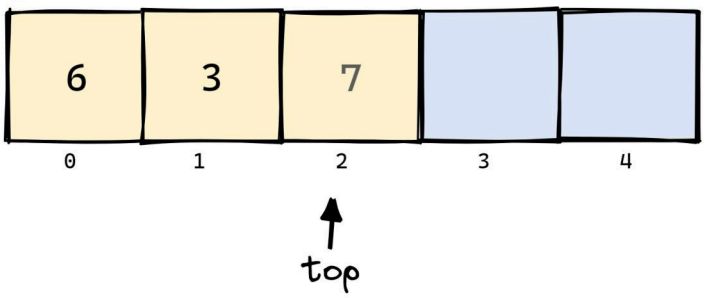
→ Let us understand implementation of stack using array problem by an example n=5



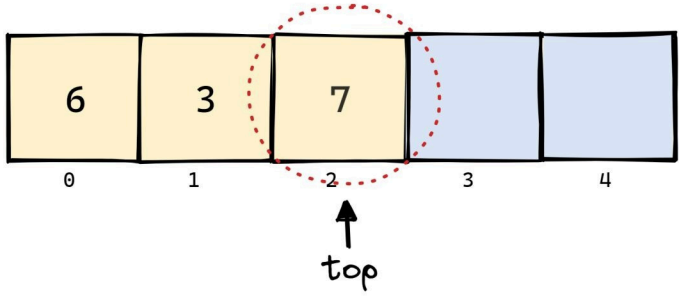
- Push(6) in an array.
  → Increase top by 1. And push that element in an array.



- Push(3),Push(7) an array.
  → Increase top at each push operation by 1.

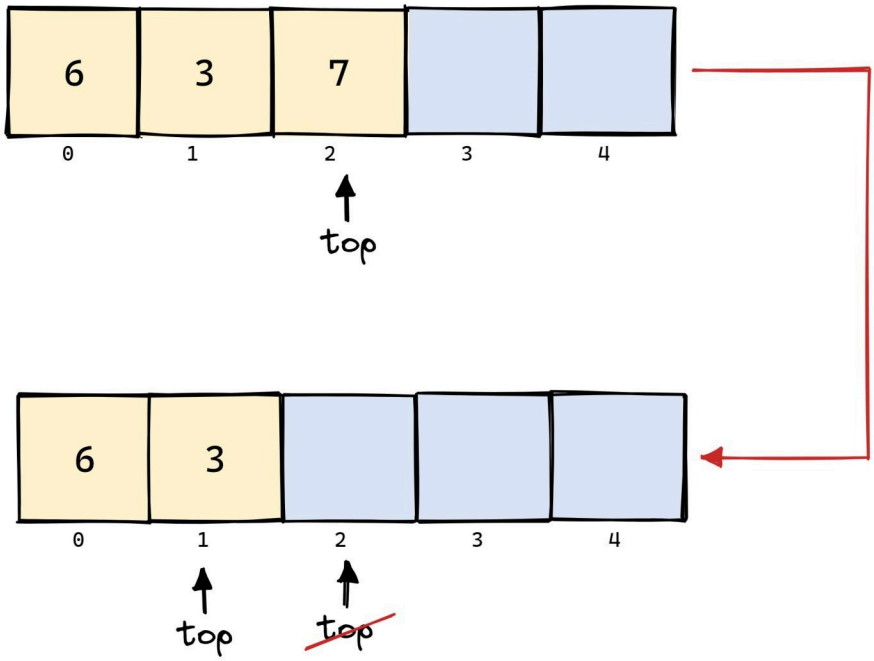


- top() ⟶ arr[top]



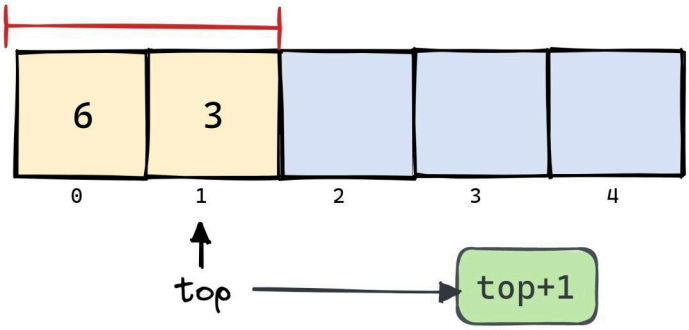
- pop()

We are going to pop 7 from array, And going to decreased top by 1.



- Size()

→ we are going to return top+1.



- Here its code implementation.

```
public class Stack {
    private int[] data;
    private int top;

    public Stack(int size) {
        data = new int[size];
        top = -1;
    }

    public void push(int value) {
        if (top == data.length - 1) {
            System.out.println("Stack is full");
            return;
        }
        top++;
        data[top] = value;
    }

    public int pop() {
        if (isEmpty()) {
            System.out.println("Stack is empty");
            return -1;
        }
        int value = data[top];
        top--;
        return value;
    }

    public int peek() {
        if (isEmpty()) {
            System.out.println("Stack is empty");
            return -1;
        }
        return data[top];
    }
    public boolean isEmpty() return top == -1;
    public boolean isFull() return top == data.length - 1;
}
```

```java
public static void main(String[] args) {
    Stack stack = new Stack(5);

    stack.push(1);
    stack.push(2);
    stack.push(3);
    stack.push(4);
    pop()
    stack.push(5);
    stack.push(6);

    // print the top element and pop it off the stack
    System.out.println("Top element: " + stack.peek());
    System.out.println("Popped element: " + stack.pop());

    stack.push(6);
}
```

Time Complexity: O(1) on average. This is because these operations only involve adding or removing an elements.

Space Complexity: O(n), where n is the maximum size of the stack. This is because we are using an array of fixed size to store the elements in the stack.

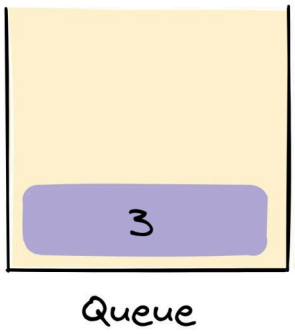
Stack Implementation using a Single Queue

→ We are given queue data structure, the task is to implement stack using only given queue data structure.

→ The basic idea behind implementing a stack using a single queue is to reverse the order of elements in the queue so that the front element of the queue becomes the top element of the stack.

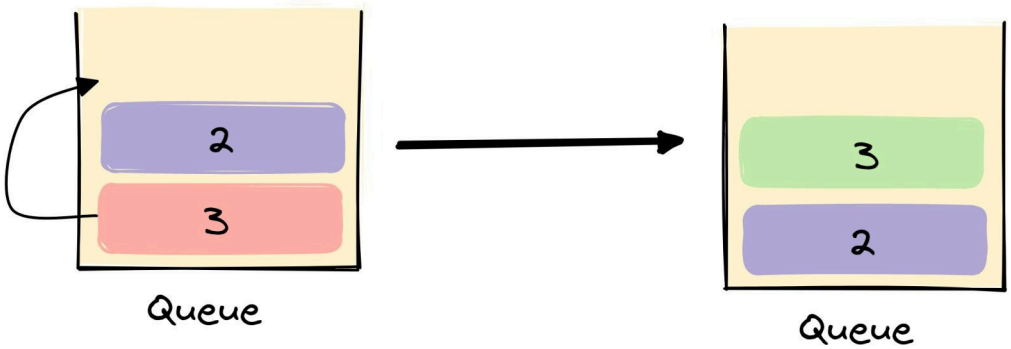
→ Let's take one example :-

- push(3) → Push 3 in our Queue. And keep a track of Queue size we are going to use its size.

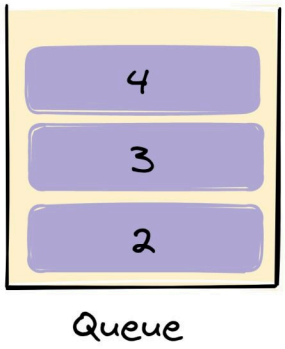


size = 1

- push(2) → Push 2 in our Queue.
  → Now size our Queue became 2.
  → we are going to perform size-1 operation in our Queue. Take top most element from Queue and push it again in our Queue till size-1 times.

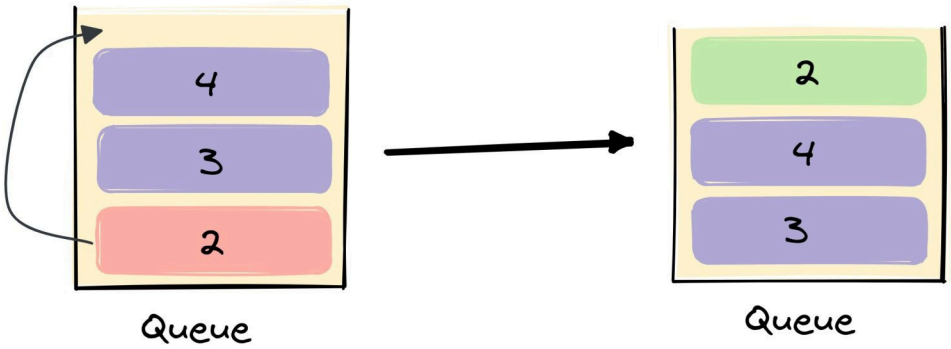


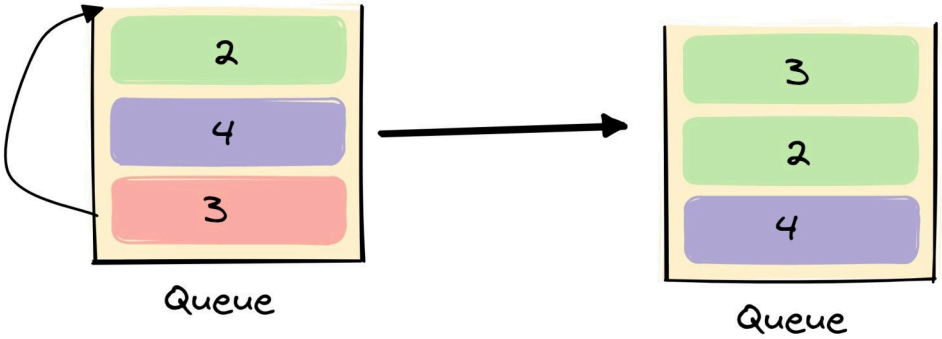
- push(4) → Push 4 in our Queue.
  → Now size our Queue became 3.
  → we are going to perform size-1 operation in our Queue. Take top most element from Queue and push it again in our Queue till size-1 times.



size = 3

- Our Queue size is 3, so we are going to perform size-1 operation in our Queue. Take top most element from Queue and push it again in our Queue till size-1 times.





- Here its code implementation.

```java
public class Stack {
    private Queue<Integer> queue;

    public Stack() {
        queue = new LinkedList<>();
    }

    public void push(int value) {
        int size = queue.size();
        queue.offer(value);
        for (int i = 0; i < size; i++) {
            int element = queue.poll();
            queue.offer(element);
        }
    }

    public int pop() {
        if (isEmpty()) {
            throw new IllegalStateException("Stack is empty");
        }
        return queue.poll();
    }

    public int peek() {
        if (isEmpty()) {
            throw new IllegalStateException("Stack is empty");
        }
        return queue.peek();
    }
    public boolean isEmpty() return queue.isEmpty();
    public int size() return queue.size();
}
```

```java
public static void main(String[] args) {
    Stack stack = new Stack();

    stack.push(10);
    stack.push(20);
    stack.push(30);

    System.out.println("Top element of stack: " + stack.peek());
    System.out.println("Stack size: " + stack.size());

    stack.pop();
    stack.pop();
    stack.pop();

    System.out.println("Stack is empty: " + stack.isEmpty());
}
```

Time Complexity: O(n), where n is the number of elements in the queue.

Space Complexity: O(n), where n is the number of elements in the queue.

Valid Parentheses

Easy

→ Given a string s containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.

→ An input string is valid if:

1. Open brackets must be closed by the same type of brackets.
2. Open brackets must be closed in the correct order.
3. Every close bracket has a corresponding open bracket of the same type.

s = "{}" ⟶ True

s = "{()}" ⟶ True

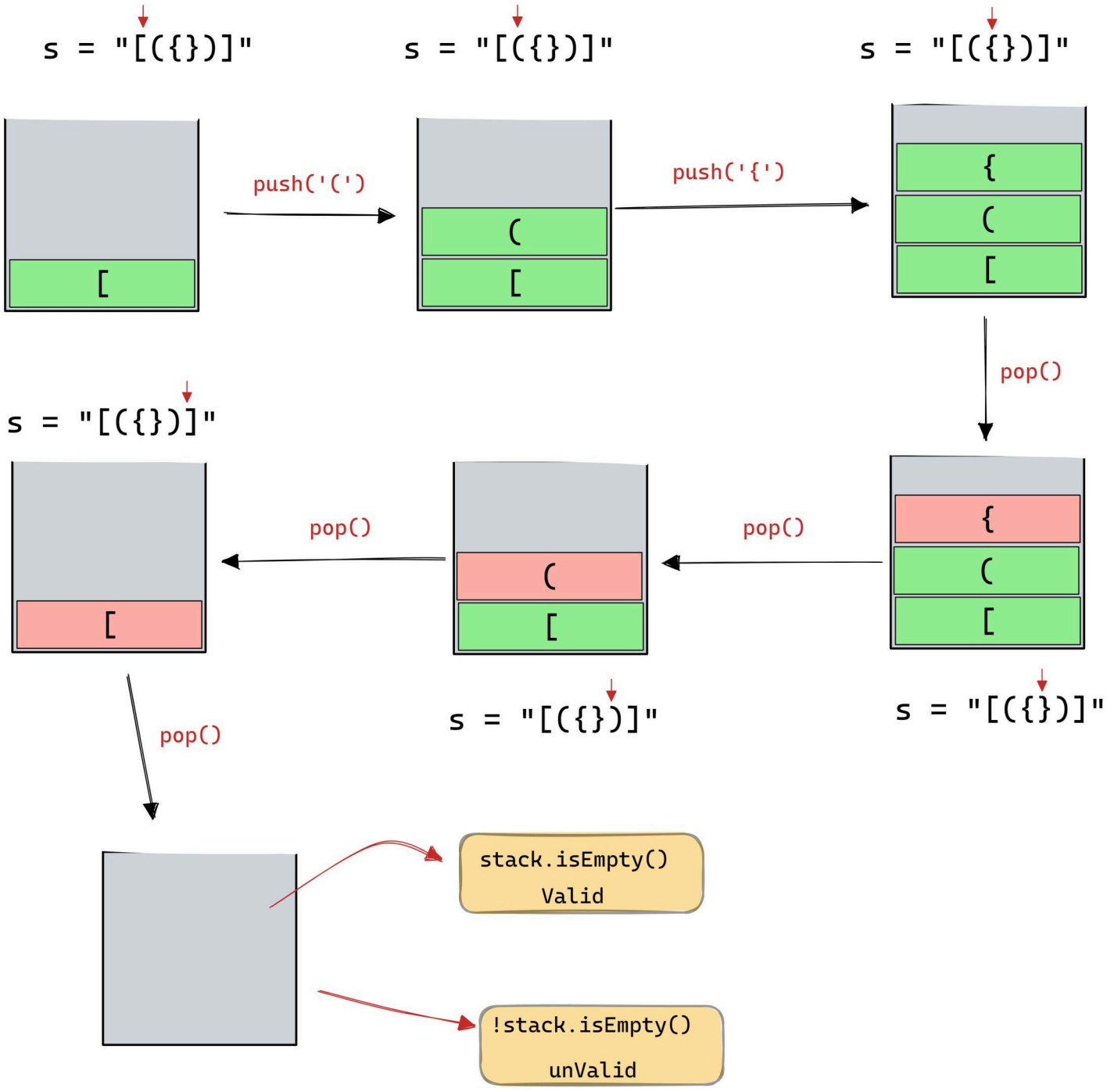
s = "}{" ⟶ False

Approach (Using Stack):-

1. Declare a character stack
2. Now traverse the string expression :-
   i. If the current character is a starting bracket ('(' or '{' or '[') then push it to stack.
   ii. If we encounter closing bracket, then we will check then pop from stack and if the popped character is the matching starting bracket then fine.
   iii. Else return false, brackets are not Valid.
4. After complete traversal, if there is some starting bracket left in stack then Not Valid, else Valid.

in Santosh Kumar Mishra

Dry-Run :-



```java
class Solution {
    public boolean isValid(String s) {
        Stack<Character> stack = new Stack<>();

        for(int i = 0;i<s.length();i++){

            Character currentChar = s.charAt(i);
            if(currentChar == '(' || currentChar == '[' || currentChar == '{'){
                stack.push(currentChar);
            }
            else{

                if(stack.isEmpty()) return false;

                if(stack.peek() == '(' && currentChar == ')'){
                    stack.pop();
                }
                else if(stack.peek() == '[' && currentChar == ']'){
                    stack.pop();
                }
                else if(stack.peek() == '{' && currentChar == '}'){
                    stack.pop();
                }
                else{
                    return false;
                }
            }
        }
        return stack.isEmpty();
    }
}
```

Time-Complexity ⟶ O(N)

Space-Complexity ⟶ O(N)

Prefix to Infix Conversion

Infix : An the operator appears in between the operands in the expression. Simply of the form (operand1 operator operand2).

infix ⟶ a + b

Prefix: if the operator appears in the expression before the operands. Simply of the form (operator operand1 operand2).

prefix ⟶ + a b

➙ Given a Prefix expression, convert it into a Infix expression.

Input : Prefix : *+AB-CD

Output : Infix : ((A+B)*(C-D))

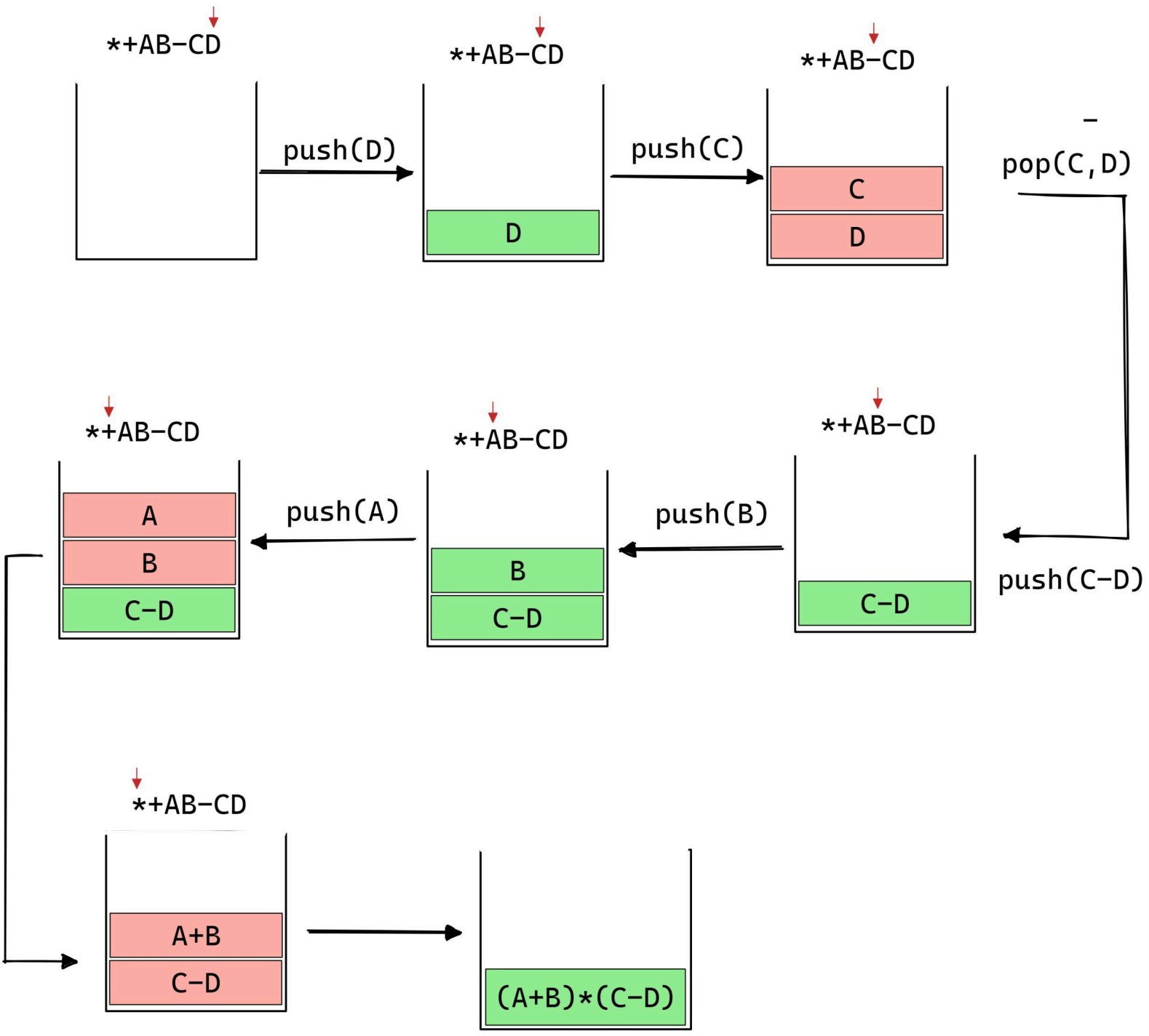
- Algorithm for Prefix to Infix:

1. Read the prefix expression from right to left.
2. If the symbol is an operand, push it onto the stack.
3. If the symbol is an operator, then pop two operands from the Stack
   Create a string by concatenating the two operands and the operator between them.
   string = (operand1 + operator + operand2).
   And push the resultant string back to Stack.
4. Repeat step 2 and 3 until the end of the prefix expression is reached.
5. The final string on the stack is the infix expression.

in Santosh Kumar Mishra

Dry-Run → prefix → *+AB-CD

infix → (A+B)*(C-D)



```java
class Solution{

   static boolean isOperator(char x){
     switch(x){
        case '+':
        case '-':
        case '*':
        case '/':
        case '^':
        case '%':
            return true;
    }
    return false;
  }

  Public static String convert(String str){
     Stack<String> stack = new Stack<>();
     int l = str.length();

     for(int i = l - 1; i >= 0; i--){
        char c = str.charAt(i);

        if (isOperator(c)){
            String op1 = stack.pop();
            String op2 = stack.pop();

            String temp = "(" + op1 + c + op2 + ")";
            stack.push(temp);
        }
        else{
            stack.push(c + "");
        }
    }
    return stack.pop();
  }
}
```

Time-Complexity ⟶ O(N)

Space-Complexity ⟶ O(N)

Prefix to Postfix Conversion

Prefix: if the operator appears in the expression before the operands.
Simply of the form (operator operand1 operand2).

prefix ⟶ + a b

Postfix: if the operator appears in the expression after the operands.
Simply of the form (operand1 operand2 operator).

postfix ⟶ a b +

➙ Given a Prefix expression, convert it into a Postfix expression.

Input : Prefix : *+AB-CD

Output : Postfix : AB+CD-*

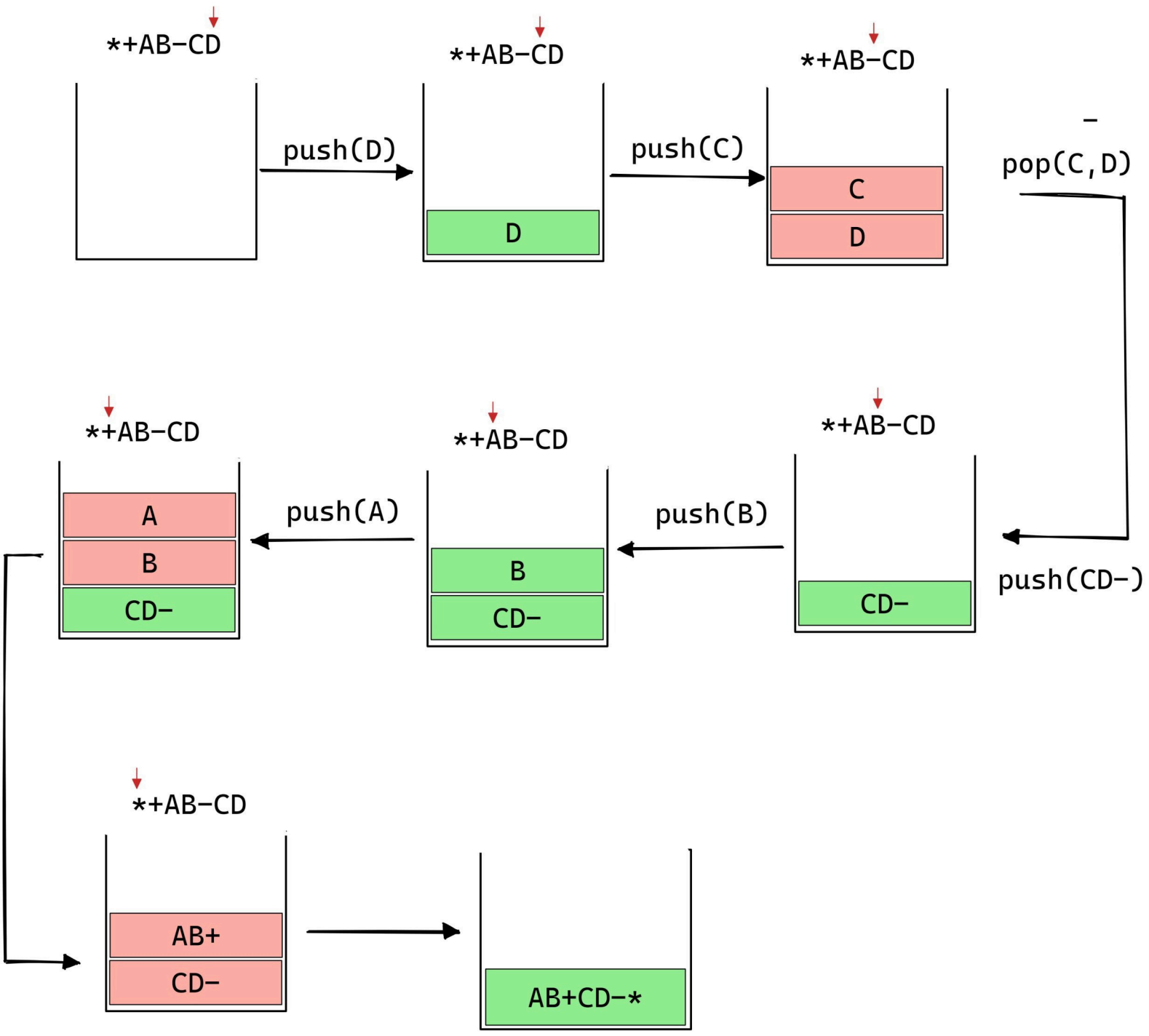
- Algorithm for Prefix to Postfix:

1. Read the prefix expression from right to left.
2. If the symbol is an operand, push it onto the stack.
3. If the symbol is an operator, then pop two operands from the Stack
   Create a string by concatenating the two operands and the operator after them.
   string = operand1 + operand2 + operator
   And push the resultant string back to Stack.
4. Repeat step 2 and 3 until the end of the prefix expression is reached.
5. The final string on the stack is the Postfix expression.

in Santosh Kumar Mishra

Dry-Run → prefix → *+AB-CD

postfix → AB+CD-*



```java
class Solution{

    static boolean isOperator(char x){
        switch (x) {
        case '+':
        case '-':
        case '/':
        case '*':
            return true;
        }
        return false;
    }

    static String preToPost(String pre_exp){

        Stack<String> s = new Stack<String>();
        int length = pre_exp.length();

        for (int i = length - 1; i >= 0; i--){

            if (isOperator(pre_exp.charAt(i))){
                String op1 = s.peek();
                s.pop();
                String op2 = s.peek();
                s.pop();

                String temp = op1 + op2 + pre_exp.charAt(i);
                s.push(temp);
            }
            else {
                s.push(pre_exp.charAt(i) + "");
            }
        }
        return s.peek();
    }
}
```

Time-Complexity ⟶ O(N)

Space-Complexity ⟶ O(N)

Postfix to Prefix Conversion

Postfix: if the operator appears in the expression after the operands.
Simply of the form (operand1 operand2 operator).

postfix ⟶ a b +

Prefix: if the operator appears in the expression before the operands.
Simply of the form (operator operand1 operand2).

prefix ⟶ + a b

➔ Given a Postfix expression, convert it into a Prefix expression.

Input : Postfix : AB+CD-*

Output : Prefix : *+AB-CD

- Algorithm for Postfix to Prefix:

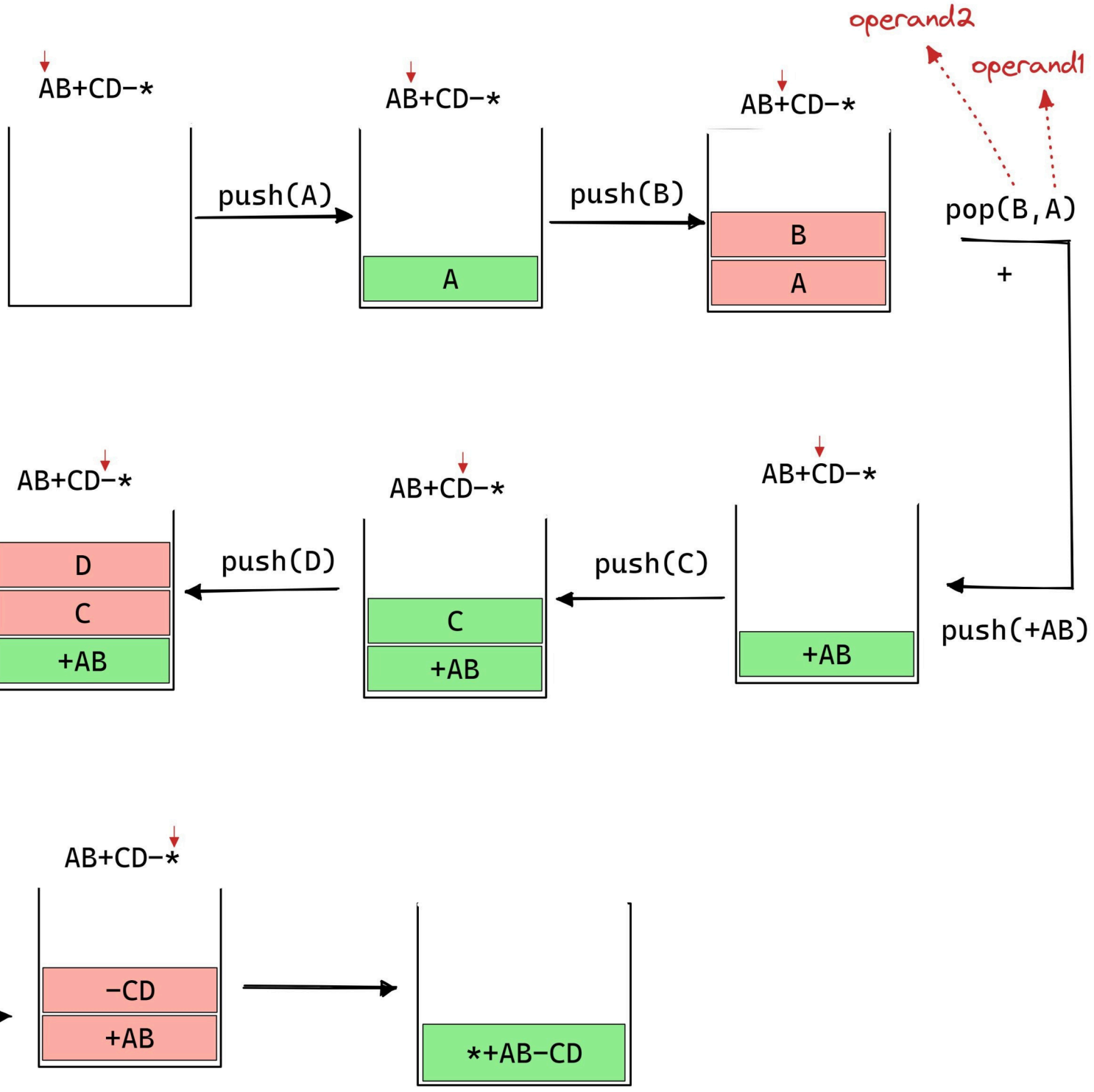
1. Read the Postfix expression from left to right.
2. If the symbol is an operand, then push it onto the Stack.
3. If the symbol is an operator, then pop two operands from the Stack
   Create a string by concatenating the two operands and the operator before them.
   string = operator + operand2 + operand1
   And push the resultant string back to Stack
4. Repeat step 2 and 3 until the end of the postfix expression is reached.
5. The final string on the stack is the Prefix expression.

Santosh Kumar Mishra

Dry-Run → Postfix : AB+CD-*

Prefix : *+AB-CD

operator + operand2 + operand1



```java
class Solution{
    static boolean isOperator(char x){
        switch (x) {
        case '+':
        case '-':
        case '/':
        case '*':
            return true;
        }
        return false;
    }

    static String postToPre(String post_exp){

        Stack<String> s = new Stack<String>();
        int length = post_exp.length();

        for (int i = 0; i < length; i++) {

            if (isOperator(post_exp.charAt(i))) {

                String op1 = s.peek();
                s.pop();
                String op2 = s.peek();
                s.pop();

                String temp = post_exp.charAt(i) + op2 + op1;
                s.push(temp);
            }
            else {
                s.push(post_exp.charAt(i) + "");
            }
        }
      return s.peek();
    }
}
```

Time-Complexity ⟶ O(N), where N is the length of the string.

Space-Complexity ⟶ O(N), where N is the stack size.

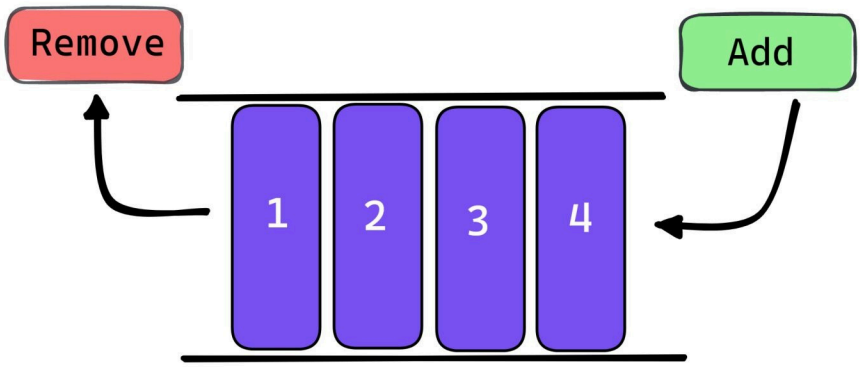
Queue Introduction

→ A queue is a data structure used in computer science and programming that allows for the orderly processing of data by following the "First In First Out" (FIFO) principle.

→ In a queue, elements are added at the rear and removed from the front, such that the first element added is always the first one to be removed.

- Queue → Linear Data Structure
  → FIFO (First In First Out)

→ Insertion and deletion in queues takes place from the opposite ends of the list.



• Basic Operations of Queue

→ There are some basic operations that allow us to perform different actions on a queue.

- Enqueue:
  This operation adds an element to the rear (or end) of the queue. It takes a data item as input and inserts it at the end of the queue.

- Dequeue:
  This operation removes the element at the front (or beginning) of the queue. It returns the data item that was removed, and updates the queue so that the next element becomes the new front of the queue.

- Peek:
  This operation returns the element at the front of the queue without removing it. It allows us to examine the next element to be dequeued, without actually dequeuing it.

- isEmpty:
  This operation checks if the queue is empty or not. It returns true if there are no elements in the queue, and false otherwise.

• Applications of Queue Data Structure in real-life

→ The queue data structure has many real-life applications, some of which include:

- Supermarket checkout lines:

→ Queues are commonly used in supermarkets to manage checkout lines. Customers join the queue at the end, and are served in the order they joined.

- Call centers:

→ Queues are used in call centers to manage incoming calls. Callers are placed in a queue and are served in the order they called.

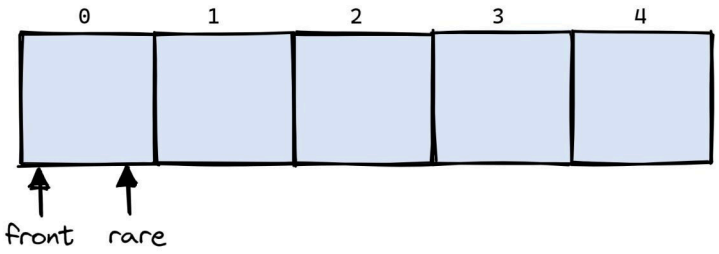
- Ticketing systems:

→ Queues are used in ticketing systems, such as for concerts or sporting events. Customers join a queue to purchase tickets, and are served in the order they joined.

Queue Implementation using Arrays

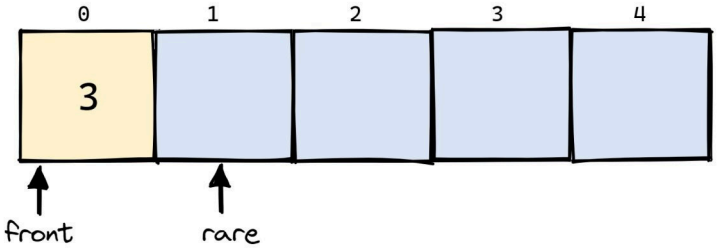
→ A queue is a linear data structure that follows the FIFO (First-In-First-Out) principle. In a queue, elements are inserted from the rear end and deleted from the front end. The simplest way to implement a queue is using an array.

→ Let us understand implementation of queue using array problem by an example n=5



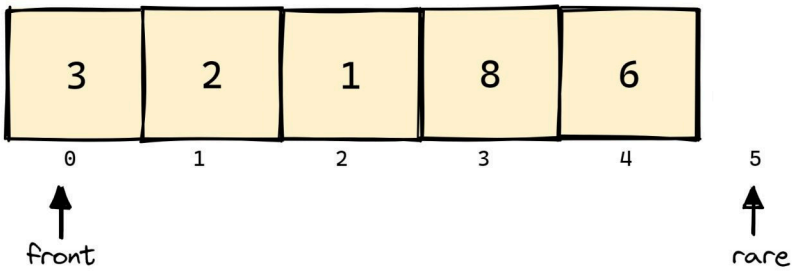
size = 0
n = 5

- Push(3) in an array.
  → Increase size by 1. And increase rare pointer.



size = 1
n = 5

- Push(2),Push(1),Push(8),Push(6),Push(7) in an array.
  → Increase size by 1. And increase rare pointer.

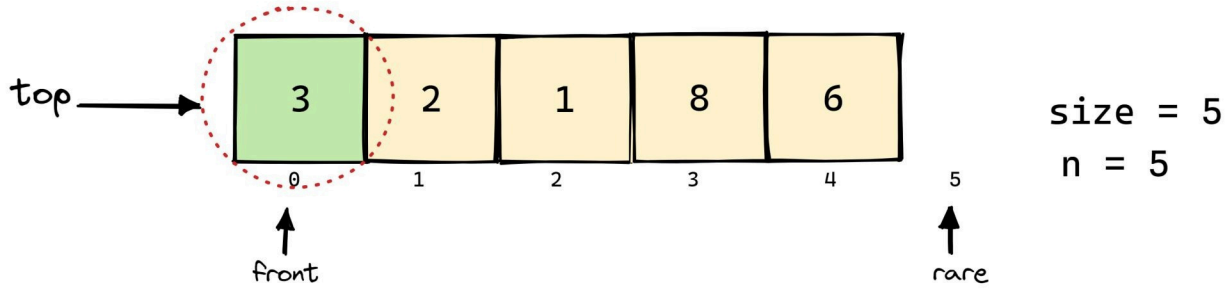


size = 5
n = 5

- Note that we can't push(7) into our array, because size of array is already full (size is equivalent to capacity of queue).

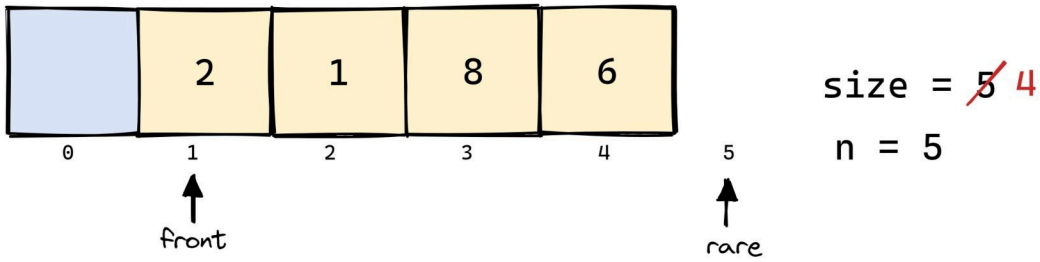
- top() ⟶ arr[front]

  Note → If front == rare that means there is no element in our array.



- pop()

  Delete the arr[front] element from array and increase front pointer by 1.

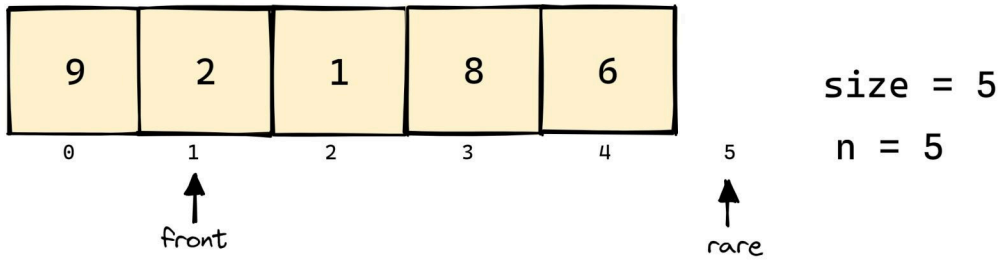


- push(9)

→ Suppose we want to push 9 into our array, but rare is pointing into some imaginare position(5) which doest not exist.

→ But note that size is lesser than the n. Hence the 9 can be inserted.

→ So what we are going to do is (rare % n) = 0th index. Then increment size by 1.



- Here its code implementation.

```
public class  Main {
    private int[] queue;
    private int front;
    private int rear;
    private int size;

    public  Main(int capacity) {
        queue = new int[capacity];
        front = 0;
        rear = 0;
        size = 0;
    }

    public boolean isEmpty() return size == 0;
    public boolean isFull() return size == queue.length;

    public void enqueue(int item) {
        if (isFull()) {
            throw new IllegalStateException("Queue is full");
        }
        queue[rear] = item;
        rear = (rear + 1) % queue.length;
        size++;
    }

    public int dequeue() {
        if (isEmpty()) {
            throw new IllegalStateException("Queue is empty");
        }
        int item = queue[front];
        front = (front + 1) % queue.length;
        size--;
        return item;
    }
```

```
    public void display() {
        if (isEmpty()) {
            System.out.println("Queue is empty");
            return;
        }
        int i = front;
        do {
            System.out.print(queue[i] + " ");
            i = (i + 1) % queue.length;
        } while (i != rear);
        System.out.println();
    }


    public static void main(String[] args) {
         Main q = new  Main(5);

          q.enqueue(3);
          q.enqueue(2);
          q.enqueue(1);
          q.enqueue(8);
          q.enqueue(6);

          q.dequeue();
          q.enqueue(9);

          q.display();
    }
}
```

Space Complexity: O(n), where n is the maximum capacity of the queue. This is because we need to allocate an array of size n to store the elements of the queue.

Time Complexity: O(1) on average. This is because these operations only involve adding or removing an element from the front or rear of the array.

Queue Implementation using Stack

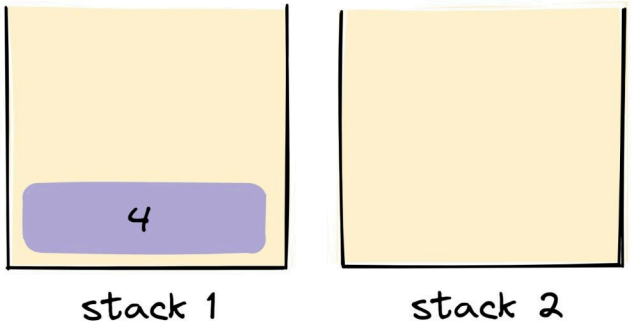
→ The problem statement here is to implement queue using stack. Queue is a linear data structure which follows First In First Out (FIFO) principle. This means that the element which is inserted first will be the one that will be removed first.

→ Stack is also a linear data structure which follows Last In First Out (LIFO) principle, means that the element inserted last in the stack will be the one to be removed first from the stack and vice versa.

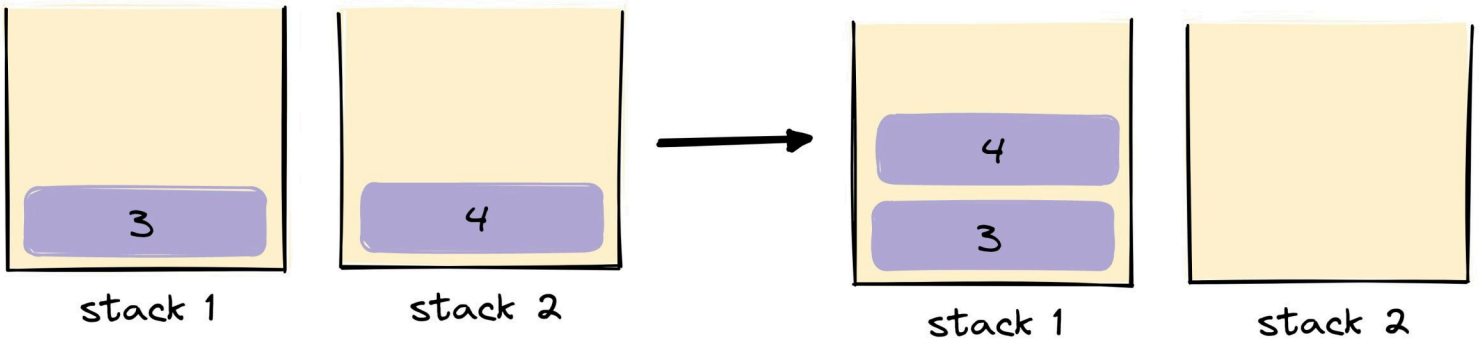
- By using 2 stacks at a time, we can simply push the element in stack 1 and will pop the first occuring element by pushing all the elements of stack 1 to stack 2 and popping out the top element of stack 2.

→ Let's take one example :-

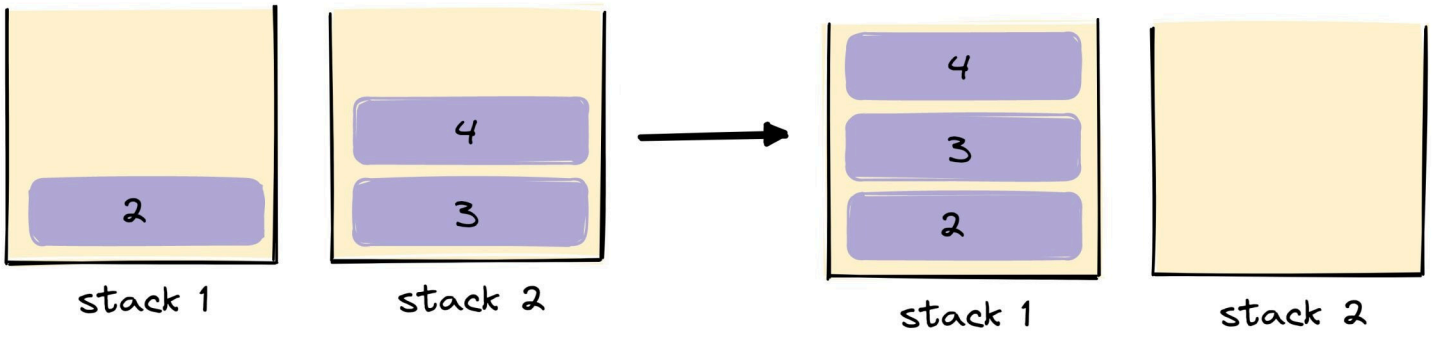
- push(4) → Base case if stack1 & stack2 is empty then push(4) into stack1.



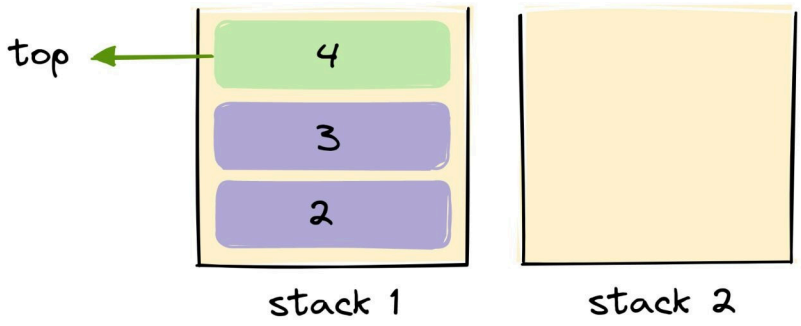
- push(3) → transfer element from stack1 to stack2.
  → Then, push(3) into stack1.
  → Now push back stack2 element to stack1.



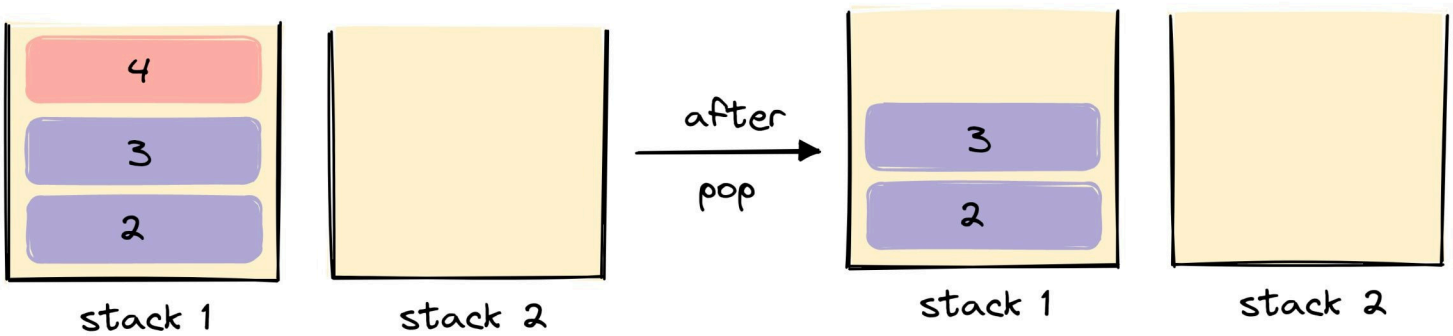
- push(2) → transfer element from stack1 to stack 2.
  → push(2) into stack1.
  → Now push back stack2 element to stack1.



- top() → stack1 peek element is its top.



- pop() → Because we are maintaining queue order in stack1. so we perform FIFO operation on stack1.



- Here its code implementation.

→ The time complexity of the dequeue operation is O(1), since it simply involves popping an element from stack1. The time complexity of the enqueue operation is O(n), where n is the number of elements in the queue, since we may need to transfer all the elements from stack1 to stack2 and back. The space complexity of the queue is O(n), where n is the number of elements in the queue.

```java
public class Queue {
    private Stack<Integer> stack1;
    private Stack<Integer> stack2;

    public Queue() {
        stack1 = new Stack<>();
        stack2 = new Stack<>();
    }

    public boolean isEmpty() {
        return stack1.isEmpty() && stack2.isEmpty();
    }

    public void enqueue(int item) {

        while (!stack1.isEmpty()) {
            stack2.push(stack1.pop());
        }

        stack1.push(item);
        while (!stack2.isEmpty()) {
            stack1.push(stack2.pop());
        }
    }
```

```
    public int dequeue() {
        if (isEmpty()) {
            throw new IllegalStateException("Queue is empty");
        }
        return stack1.pop();
    }

    public void display() {

        if (isEmpty()) {
            System.out.println("Queue is empty");
            return;
        }
        System.out.print("front ");

        for (int i = stack1.size() - 1; i >= 0; i--) {
            System.out.print(stack1.get(i) + " ");
        }
        System.out.println("rear");
    }
}
```

Space Complexity: O(n), where n is the number of elements in the queue.

Time Complexity: O(n), where n is the number of elements in the queue.

The Art Of Data Structures and Algorithms

Visual Aids and Detailed Dry Runs for Easy Understanding

The Art of Structures Algorithms

ate Quick Reference Guide

₹399 ~~₹999~~

sh Kumar Mishra

gineer at Microsoft, Author

@iamsantoshmishra

+91-9701101993

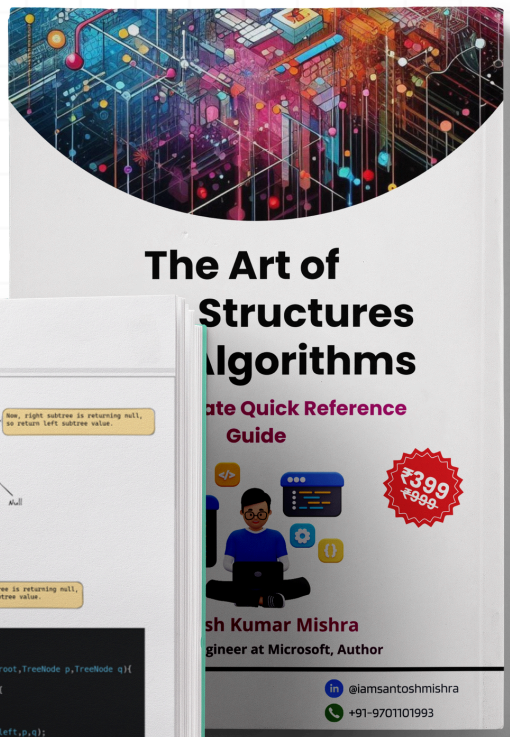
Questions Explained from Brute Force to Optimized Solutions

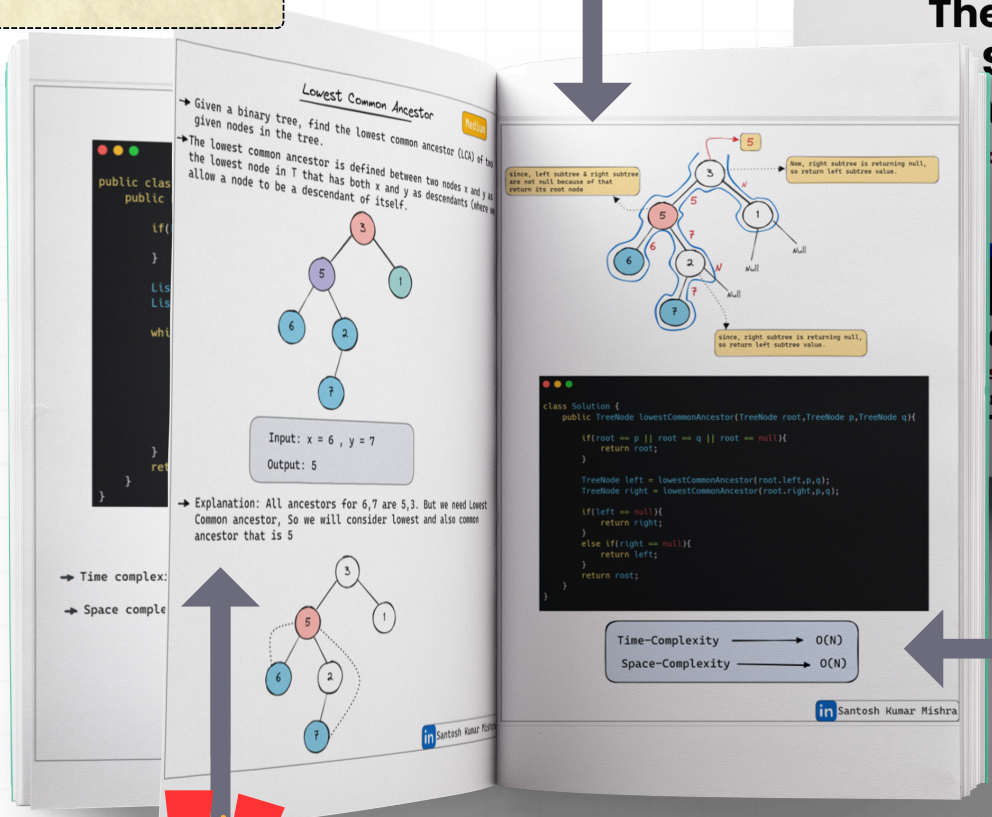
Perfect for Beginners to Advanced Learners

BUY NOW

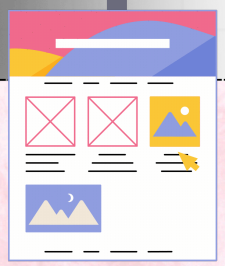
The Art Of Data Structures and Algorithms

One-Time Purchase with Lifetime Access





Comprehensive Coverage of Important DSA Questions



Expertly Designed by Industry Professionals

The Art of Data Structures and Algorithms

Readers' Reviews and Insights

Prem Kumar

Software Engineer

I've never seen a book that covers DSA questions so thoroughly! Santosh breaks down each problem from the basic brute-force solution to the most optimized version, using proper diagrams and dry runs. It's a fantastic resource if you struggle with understanding complex algorithms.

Saumya Awasthi

Software Engineer

If you're preparing for interviews, this is the book you need. It contains all the key DSA problems explained with easy-to-follow graphics. The step-by-step approach, from brute force to optimized solutions, along with well-done dry runs, makes difficult concepts very digestible.

Archy Gupta

Software Engineer

Thank you, Santosh, for sending me a copy of this book. It's truly amazing! The book contains all the important DSA questions, and each one is explained from brute-force to optimized solutions with very clear dry runs and diagrams. It's the perfect resource for mastering DSA concepts and acing coding interviews.

The Art of Data Structures and Algorithms
Readers' Reviews and Insights

Aishwarya Tripathi
Software Engineer

"This book was a game-changer for my Amazon interview preparation. The collection of important DSA problems and their step-by-step solutions from brute-force to optimized helped me build a solid understanding. The graphical dry runs made complex concepts easy to follow. I highly recommend this book if you're aiming for top tech companies!

Parth Chaturvedi
Software Engineer

"The structured explanations in this book played a huge role in my Amazon interview success. The dry runs and graphical representations provided a clear understanding of complex problems, and the progression from brute-force to optimal solutions was exactly what I needed. This book is a must-read for serious interview prep.

Namrata Tiwari
Software Engineer

"This book's strength lies in its ability to break down complex problems into simple, understandable steps. The questions are explained starting from brute-force solutions and are then optimized with clear, graphical dry runs. It's an essential guide for interview prep and mastering DSA.

And Many More Success Stories from Satisfied Readers!

The Art of Data Structures and Algorithms

The Ultimate Quick Reference Guide

BUY NOW

Buy From Gumroad **Buy From Topmate**