

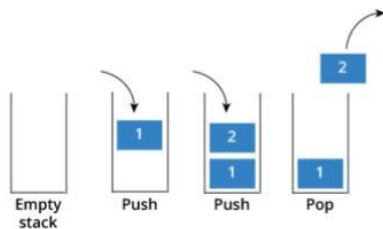


Islamic University of Gaza  
Faculty of Engineering  
Computer Engineering  
Department



## ECOM 3001 Data Structures and Algorithms Lab [Fall 2024]

### Lab "05" Topic: Stacks and Queues



**Stack**



**Queue**

**Prepared by:**  
Eng. Waseem Alfarram

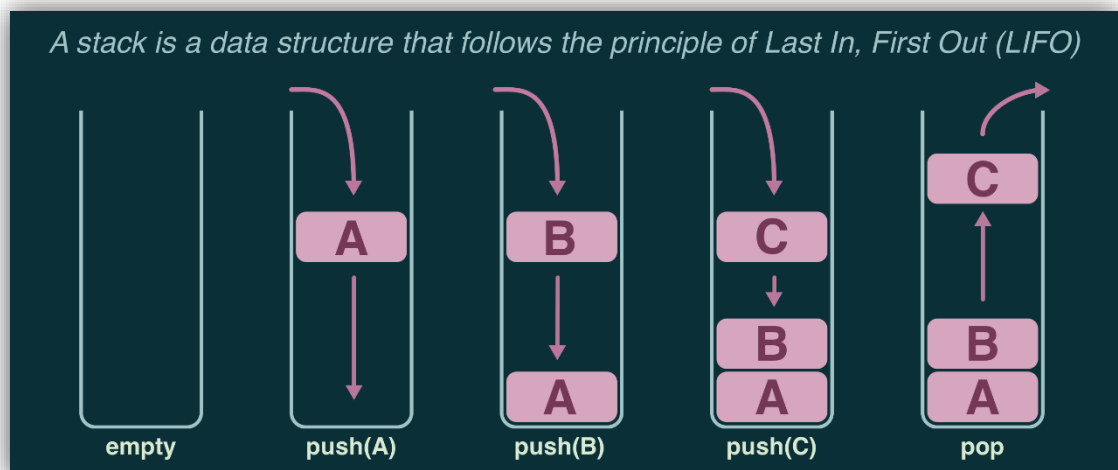
**Supervised by:**  
Dr. Ahmad Mahdi

## Lab Objectives:

- To define **Stacks** and **Queues** and differentiate between the principles they follow when interacting with their elements.
- To learn the key operations on both Stacks and Queues.
- To recognize some basic applications of both Stacks and Queues.
- To implement Stack class and Queue interface using java and to learn how to use the different operations on both of them.
- To solve some tasks about Stacks and Queues as a practice for what you learned in this lab.

## Stack Data Structure:

A **Stack** is a linear data structure that follows the Last In, First Out (LIFO) principle. This means that the last element added to the stack is the first one to be removed. It can be visualized as a pile of plates where you can only add or remove the top plate.



### ➤ Key Operations on Stack:

- **Push**: Adds an element to the top of the stack.
- **Pop**: Removes the top element from the stack.
- **Peek**: Returns the top element without removing it.
- **IsEmpty**: Checks if the stack is empty.
- **Size**: Returns the number of elements in the stack.
- **IsFull**: Checks if the stack is full (in case of fixed-size arrays).

### ➤ Applications of Stack:

- Function Calls.
- Recursion.
- Expression Evaluation and Parsing.
- Undo/Redo Operations.
- Browser History.
- Depth-First Search (DFS).

## ❖ Stack Class Implementation:

A stack data structure can be implemented by either arrays or singly linked lists. Arrays are fast and easy to use since they provide constant-time access to elements and are generally more memory efficient due to contiguous memory allocation, but they may become slow and inefficient as they grow in size. Singly linked lists, on the other hand, are more memory-efficient and have constant time complexity for adding and removing elements at the head or tail of the list.

In general, if you need to implement a stack with a fixed size, an array may be the best choice. However, if you need a dynamic stack that can grow and shrink over time, a singly linked list may be the better option.

For most applications, an **Array-Based Stack** is preferred due to its performance benefits, unless you have specific requirements that make a linked list more appropriate.

### ➤ **Array-Based Stack Class Implementation using Java:**

```
// Stack implementation in Java

class Stack {
    private int[] arr; // store elements of stack
    private int top;   // represent top of stack
    private int capacity; // total capacity of the stack

    // Creating a stack
    Stack(int size) {
        arr = new int[size]; // initialize the array
        // initialize the stack variables
        capacity = size;
        top = -1;
    }

    // push elements to the top of stack
    public void push(int x) {
        if (isFull()) {
            System.out.println("Stack OverFlow");
            System.exit(1); // terminates the program
        }
        // insert element on top of stack
        System.out.println("Inserting " + x);
        arr[++top] = x;
    }
}
```

```

// pop elements from top of stack
public int pop() {
    // if stack is empty, no element to pop
    if (isEmpty()) {
        System.out.println("STACK EMPTY");
        System.exit(1); // terminates the program
    }
    return arr[top--]; // pop element from top of stack
}

// return size of the stack
public int getSize() {
    return top + 1;
}

// check if the stack is empty
public Boolean isEmpty() {
    return top == -1;
}

// check if the stack is full
public Boolean isFull() {
    return top == capacity - 1;
}

// display elements of stack
public void printStack() {
    for (int i = 0; i <= top; i++) {
        System.out.print(arr[i] + ", ");
    }
}

public static void main(String[] args) {
    Stack stack = new Stack(5);
    stack.push(1);
    stack.push(2);
    stack.push(3);
    System.out.print("Stack: ");
    stack.printStack();

    // remove element from stack
    stack.pop();
    System.out.println("\nAfter popping out:");
    System.out.print("Stack: ");
    stack.printStack();
}
}

```

### Output:

Inserting 1  
 Inserting 2  
 Inserting 3  
 Stack: 1, 2, 3,  
 After popping out:  
 Stack: 1, 2,

➤ **Or you can easily use the Java Stack Class:**

```
import java.util.Stack;

public class StackJavaExample {
    public static void main(String[] args) {
        // Creating a stack of integers
        Stack<Integer> stack = new Stack<>();

        stack.push(1); // Pushing 1 to the stack top
        stack.push(2); // Pushing 2 to the stack top
        stack.push(3); // Pushing 3 to the stack top
        stack.push(4); // Pushing 4 to the stack top
        stack.push(5); // Pushing 5 to the stack top

        // Returns the 1-based position from the
        // top of the stack where the object is located;
        // the return value -1 indicates that the object
        // is not on the stack.
        System.out.println(stack.search(4));

        // Removing elements from the stack using the pop function
        while (!stack.empty()) {
            // Displaying the top element without removing it
            System.out.print(stack.peek() + " ");

            // Removes the top element from the stack
            stack.pop();
        }
    }
}
```

**Output:**

2  
5 4 3 2 1

➤ **Complexity Analysis:**

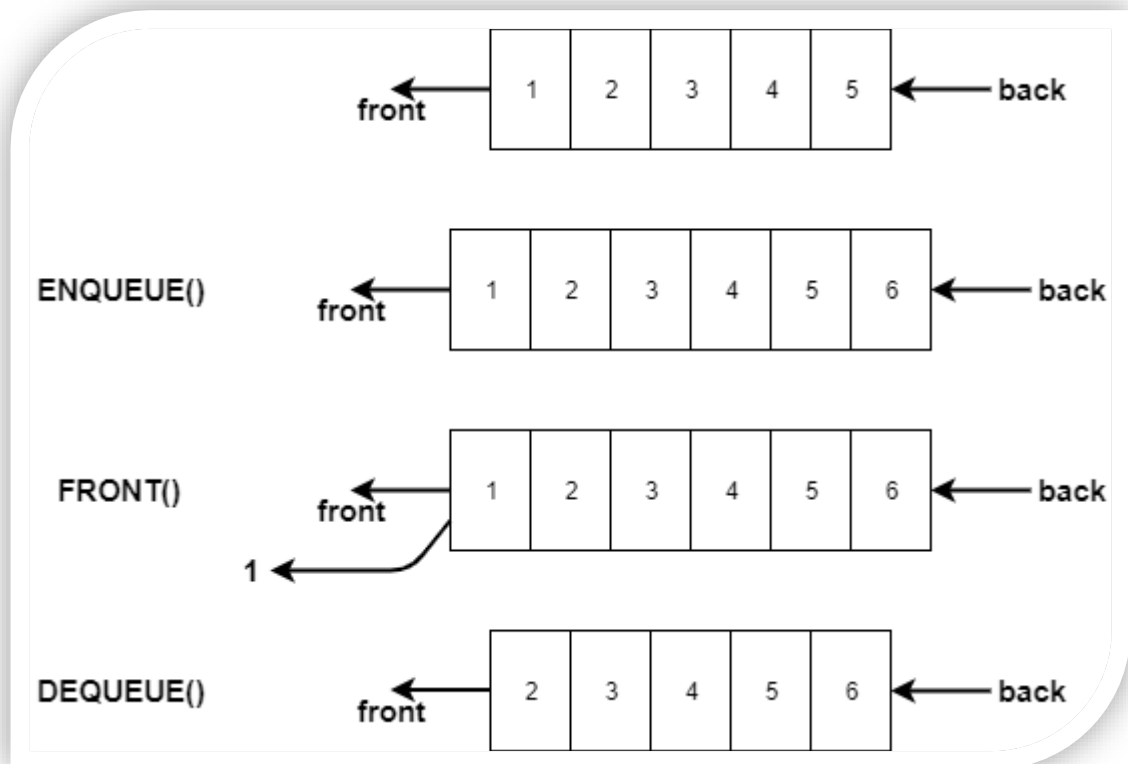
**Time Complexity:**

- push:  $O(1)$
- pop:  $O(1)$
- peek:  $O(1)$
- isEmpty:  $O(1)$
- isFull:  $O(1)$

**Space Complexity:**  $O(n)$ , where  $n$  is the number of items in the stack.

## Queue Data Structure:

A **Queue** is a linear data structure that follows the First In, First Out (FIFO) principle. This means that the first element added to the queue is the first one to be removed. It can be visualized as a line of people waiting for a service, where the first person in line is the first to be served.



### ➤ Key Operations on Queue:

- **enqueue**: Adds an element to the end (rear) of the queue.
- **dequeue**: Removes and returns the front element of the queue.
- **front (or peek)**: Returns the front element of the queue without removing it.
- **isEmpty**: Checks if the queue is empty.
- **getSize**: Returns the number of elements in the queue.
- **isFull**: Checks if the queue is full (in case of fixed-size arrays).

## ➤ Applications of Queue:

- Task scheduling in operating systems.
- Data transfer in network communication.
- Simulation of real-world systems (e.g., waiting lines).
- Priority queues for event processing queues.

## ❖ Queue Class Implementation:

A queue data structure can be implemented by either arrays or singly linked lists depends on your specific requirements and constraints. If you need a queue with dynamic size and frequent insertions and deletions, a linked list-based implementation is preferable. However, if you prioritize faster access times and have a relatively stable queue size, an array-based implementation might be more efficient.

### ➤ LinkedList-Based Queue Class Implementation using Java:

```
// Node class representing a single node in the linked list
class Node {
    int data;
    Node next;
    Node(int new_data) {
        this.data = new_data;
        this.next = null;
    }
}

// Queue implementation in Java using a linked list
class Queue {
    Node front, rear;
    int size = 0;
    Queue() {
        front = rear = null;
    }

    // Function to check if the queue is empty
    boolean isEmpty() {
        return front == null && rear == null;
    }

    // Function to add an element to the queue
    void enqueue(int new_data) {
        Node new_node = new Node(new_data);
        size++;
        if (rear == null) {
            front = rear = new_node;
            return;
        }
    }
}
```

```

        rear.next = new_node;
        rear = new_node;
    }

    // Function to remove an element from the queue
    void dequeue() {
        if (isEmpty()) {
            System.out.println("Queue Underflow");
            return;
        }
        Node temp = front;
        front = front.next;
        if (front == null) {
            rear = null;
        }
        size--;
    }

    // Function to get the number of elements in the queue
    int getSize() {
        return size;
    }

    // Function to get the front element of the queue
    int getFront() {
        if (isEmpty()) {
            System.out.println("Queue is empty");
            return Integer.MIN_VALUE;
        }
        return front.data;
    }

    // Function to get the rear element of the queue
    int getRear() {
        if (isEmpty()) {
            System.out.println("Queue is empty");
            return Integer.MIN_VALUE;
        }
        return rear.data;
    }
}

// Driver code to test the queue implementation
public class Main {
    public static void main(String[] args) {
        Queue q = new Queue();

        q.enqueue(10);
        q.enqueue(20);
        System.out.println("Queue Front: " + q.getFront());
        System.out.println("Queue Rear: " + q.getRear());
        System.out.println("Queue Size: " + q.getSize());

        q.dequeue();
        q.dequeue();
        q.enqueue(30);
    }
}

```

```

        q.enqueue(40);
        q.enqueue(50);
        q.dequeue();
        System.out.println("Queue Front: " + q.getFront());
        System.out.println("Queue Rear: " + q.getRear());
        System.out.println("Queue Size: " + q.getSize());
    }
}

```

### Output:

Queue Front: 10  
 Queue Rear: 20  
 Queue Size: 2  
 Queue Front: 40  
 Queue Rear: 50  
 Queue Size: 2

### ➤ Or you can easily use the Java Queue Interface:

```

import java.util.LinkedList;
import java.util.Queue;

public class QueueJavaExample {
    public static void main(String[] args) {
        Queue<String> queue = new LinkedList<>();

        // add(element): Adds an element to the rear of the
        // queue. If the queue is full, it throws an exception.
        // offer(element): Adds an element to the rear of the
        // queue. If the queue is full, it returns false.
        queue.offer("apple");
        queue.offer("banana");
        queue.offer("cherry");
        System.out.println("Queue: " + queue);

        // remove(): Removes and returns the element at the
        // front of the queue. If the queue is empty, it
        // throws an exception.
        // poll(): Removes and returns the element at the
        // front of the queue. If the queue is empty, it
        // returns null.
        String front = queue.poll();
        System.out.println("Removed element: " + front);
        System.out.println("Queue after removal: " + queue);

        queue.offer("date");
        // element(): Returns the element at the front of the
        // queue without removing it. If the queue is empty,
        // it throws an exception.
        // peek(): Returns the element at the front of the
        // queue without removing it. If the queue is empty,
        // it returns null.
        String peeked = queue.peek();
    }
}

```

```
        System.out.println("Peeked element: " + peeked);  
        System.out.println("Queue after peek: " + queue);  
    }  
}
```

### Output:

Queue: [apple, banana, cherry]

Removed element: apple

Queue after removal: [banana, cherry]

Peeked element: banana

Queue after peek: [banana, cherry, date]

### ➤ Complexity Analysis:

**Time Complexity:**  $O(1)$ , The time complexity of both operations enqueue() and dequeue() is  $O(1)$  as it only changes a few pointers in both operations.

**Space Complexity:**  $O(1)$ , The auxiliary Space of both operations enqueue() and dequeue() is  $O(1)$  as constant extra space is required.

## Exercises:

### ➤ Exercise 1: (Valid Parentheses)

*Leetcode Question Link:*

<https://leetcode.com/problems/valid-parentheses/>

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

An input string is valid if:

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

(Solve the question using **Stack**)

```
public boolean isValid(String s)
```

Example 1:

**Input:** *s* = "()"

**Output:** *true*

Example 2:

**Input:** *s* = "()[]{}"

**Output:** *true*

Example 3:

**Input:** *s* = "[]"

**Output:** *false*

Example 4:

**Input:** *s* = "([])"

**Output:** *true*

Constraints:

- $1 \leq s.length \leq 10^4$
- *s* consists of parentheses only '()[]{}'.

## ➤ Exercise 2: (First Unique Character in a String)

*Leetcode Question Link:*

<https://leetcode.com/problems/first-unique-character-in-a-string/>

Given a string *s*, find the first non-repeating character in it and return its index. If it does not exist, return -1.

(Solve the question using **Queue** – without using any Maps)

```
public int firstUniqChar(String s)
```

Example 1:

**Input:** *s* = "leetcode"

**Output:** 0

**Explanation:** The character 'l' at index 0 is the first character that does not occur at any other index.

Example 2:

**Input:** *s* = "loveleetcode"

**Output:** 2

Example 3:

**Input:** *s* = "aabb"

**Output:** -1

Constraints:

- $1 \leq s.length \leq 10^5$
- *s* consists of only lowercase English letters.

## Bonus Leetcode Questions:

- <https://leetcode.com/problems/backspace-string-compare/>
- <https://leetcode.com/problems/time-needed-to-buy-tickets/>

**End of Lab** 😊