METROPOLITAN TIRANA UNIVERSITY

Course: Data Structures and Algorithms

Fundamental Data Structures
Stacks and Queues

Evis Plaku

The classic Stack vs Queue struggle

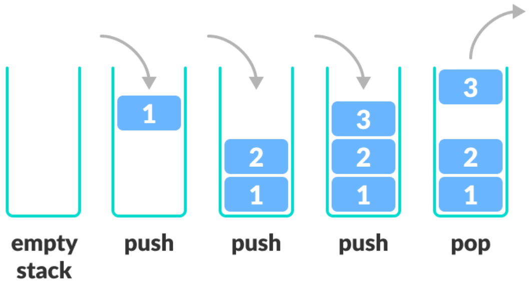
Queue: *First come, first served (FIFO)*

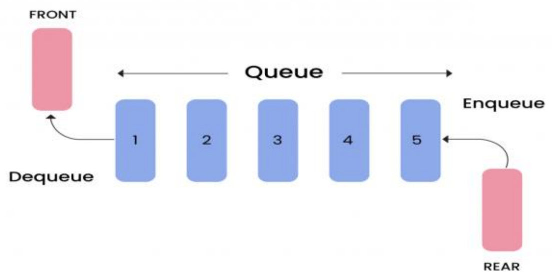
Stack: Last-minute shoppers (LIFO) be like…

What are stacks and queues

METROPOLITAN TIRANA UNIVERSITY

Linear data structures used to store and manage data efficiently





- Store elements like a stack of plates: last added, first removed
- Used in undo/redo, function calls, and expression evaluation

- Works like a line at a checkout: first added, first removed
- Used in task scheduling, buffering, and request handling

Abstract data type overview

METROPOLITAN TIRANA UNIVERSITY

An **Abstract Data Type (ADT)** defines **what operations** a data structure supports, **not how** they are implemented

Key ADTs

- List: indexed collection
- Set: unordered, unique
- Map: key – value pairs
- Graph: nodes and edges
- Stack and Queue: operations performed in particular order

Why ADTs?

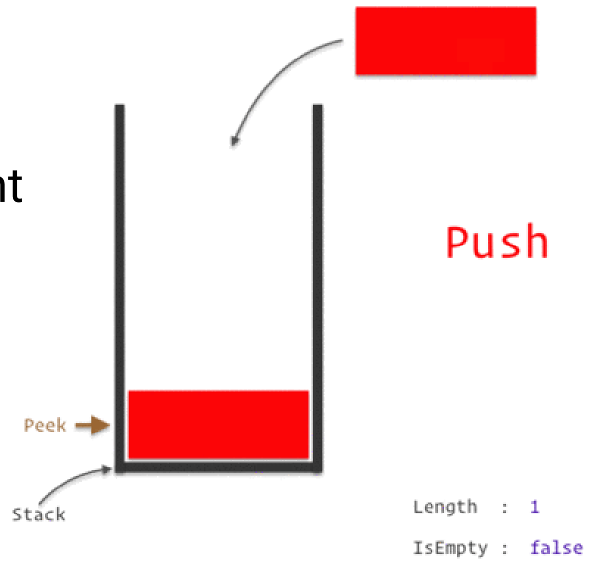
- Separates logic from implementation: focus on functionality, not internal details
- Help design flexible and reusable data structures

Stack Fundamentals

Stack ADT: key concepts and operations

A **Stack** follows **LIFO (Last In, First Out)**
the last element added is the first removed

- `push(x)`: adds x to the top
- `pop()`: removes and returns the top element
- `peek()`: returns the top element without removing it
- `isEmpty()`: check if stack is empty



Stack implementation using arrays

A fixed-size array stores elements, with a top pointer tracking the stack's end

```
class Stack {
    private int[] stack;
    private int top;

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

    public void push(int x) { /* Add element */ }
    public int pop() { /* Remove and return top */ }
    public int peek() { /* Return top without removing */ }
    public boolean iEmpty() { /* Check if empty */ }
}
```

[Click here to access complete code]

Stack implementation using linked lists

METROPOLITAN TIRANA UNIVERSITY

A linked list-based stack uses nodes, where the top points to the last added element

```
class Node {
    int data;
    Node next;
}

class Stack {
    private Node top;

    public Stack() { top = null; }

    public void push(int x) { /* Add node at top */ }
    public int pop() { /* Remove and return top node */ }
    public int peek() { /* Return top node value */ }
    public boolean isEmpty() { /* Check if top is null */ }
}
```

[Click here to access complete code]

Applications of stacks in real life

METROPOLITAN TIRANA UNIVERSITY

Undo / Redo in Editors

Function call management

Expression evaluation

Backtracking (maze, puzzles)

Browser history navigation

Parenthesis matching

Stack use case: balanced parentheses problem

Use a **stack** to ensure every opening bracket has a matching closing bracket

$\{[x \cdot (2x + y \; - 3)] + (2x \; - 7)\}$

Algorithm

- Push opening brackets onto the stack
- Pop when a matching closing bracket appears
- Mismatch or non-empty stack at end → not balanced

```
FUNCTION isBalanced(expression):
    CREATE an empty stack

    FOR each character in expression:
        IF character is an opening bracket:
            PUSH it onto the stack
        ELSE IF character is a closing bracket:
            IF stack is empty OR top does not match:
                RETURN false
            POP from stack

    RETURN true IF stack is empty, ELSE false
```

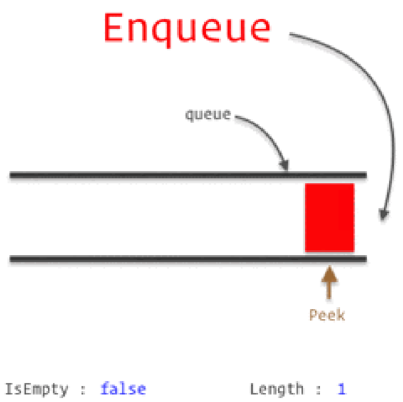
Click here to access Java implementation

Queue Fundamentals

Queue ADT: key concepts and operations

A **Queue** follows **FIFO (First In, First Out)**
Elements added in back and removed in front

- Enqueue/add: insert at the rear
- Dequeue: remove element from front
- peek(): returns the front element without removing it
- isEmpty(): check if queue is empty



Queue implementation using arrays

METROPOLITAN TIRANA UNIVERSITY

Follows **FIFO**
(First In, First Out)

Use array with front and rear pointers

```java
class Queue {
    private int[] arr;
    private int front, rear, size, capacity;

    public Queue(int capacity) {
        this.capacity = capacity;
        arr = new int[capacity];
        front = size = 0;
        rear = -1;
    }

    public void enqueue(int item) { /* Add item at rear */
}   public int dequeue() { /* Remove item from front */ }
    public int front() { /* Get front element */ }
    public boolean isEmpty() { /* Check if empty */ }
}
```

Click here to access Java implementation

Queue implementation using linked lists

Store elements in nodes with pointers to the next element.

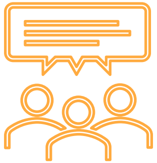
Add element at back, remove from front

```
class Queue<T> {
    private Node<T> front, rear;
    private int size;

    static class Node<T> {
        T data;
        Node<T> next;

        Node(T data) {
            this.data = data;
            this.next = null;
        }
    }

    public Queue() {
        front = rear = null;
        size = 0;
    }

    public void enqueue(T item) { /* Add item at rear */ }
    public T dequeue() { /* Remove item from front */ }
    public T front() { /* Get front element */ }
    public boolean isEmpty() { /* Check if empty */ }
}
```

[Click here to access Java implementation]

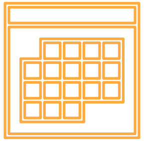
Applications of queues in real life

Customer service

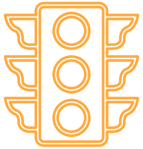


Print spooling

CPU scheduling



Traffic management



Call center systems

Breadth – first search

Queue use case: call center simulation

Queues & multi-threading are essential for simulating real-time systems like call centers

- Queues manage incoming calls and prioritize based on availability
- Agent availability is tracked to allocate customers to free agents
- Multi-threading simulates parallel call handling, improving system responsiveness

Queues ensure efficient task management and fair resource allocation

Advanced variations: circular queues & deques

Circular queue

- Overcomes limitations of regular queues by reusing empty space
- The rear pointer loops back to the front when space is available
- Efficient memory usage in applications like buffering and round-robin scheduling

Dequeue

- Allows insertion and removal of elements from both ends
- Supports both FIFO and LIFO operations
- Useful in scenarios requiring both stack and queue functionality, like task scheduling

Complexity and Performance Analysis

Complexity and performance analysis

METROPOLITAN TIRANA UNIVERSITY

Both stacks and queues offer efficient time complexities for common operations

Operation	Stack	Queue
Push / enqueue	$O(1)$	$O(1)$
Pop / dequeue	$O(1)$	$O(1)$
Peek	$O(1)$	$O(1)$
Access / search	$O(n)$	$O(n)$
Sort	$O(n \cdot \log n)$	$O(n \cdot \log n)$

Analyzing Big O for various data structures

Choose data structures based on access speed, insertion/deletion needs, and memory efficiency

Data Structure	Access	Search	Insert	Delete
Array	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Linked list	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Stack	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Queue	$O(n)$	$O(n)$	$O(1)$	$O(1)$

Key Takeaways

- Stacks follow LIFO, queues follow FIFO for ordering elements
- Stacks are ideal for recursion, backtracking, and undo operations
- Queues are used in scheduling, buffers, and handling real-time data
- Both can be implemented using arrays or linked lists with different performance

Helpful resources on arrays and linked lists

METROPOLITAN TIRANA UNIVERSITY

- 🔍 VisualAlgo: an interactive platform that visualizes data structures and algorithms

- 🎥 BroCode video to learn linked lists in a concise and interactive manner

- 📚 Programiz: beginner-friendly tutorials on data structures and algorithms with clear explanations and examples

- 💻 FreeCodeCamp course on algorithms and data structures

Quote of the Week

Stacks push you to the top,

but queues keep you in line