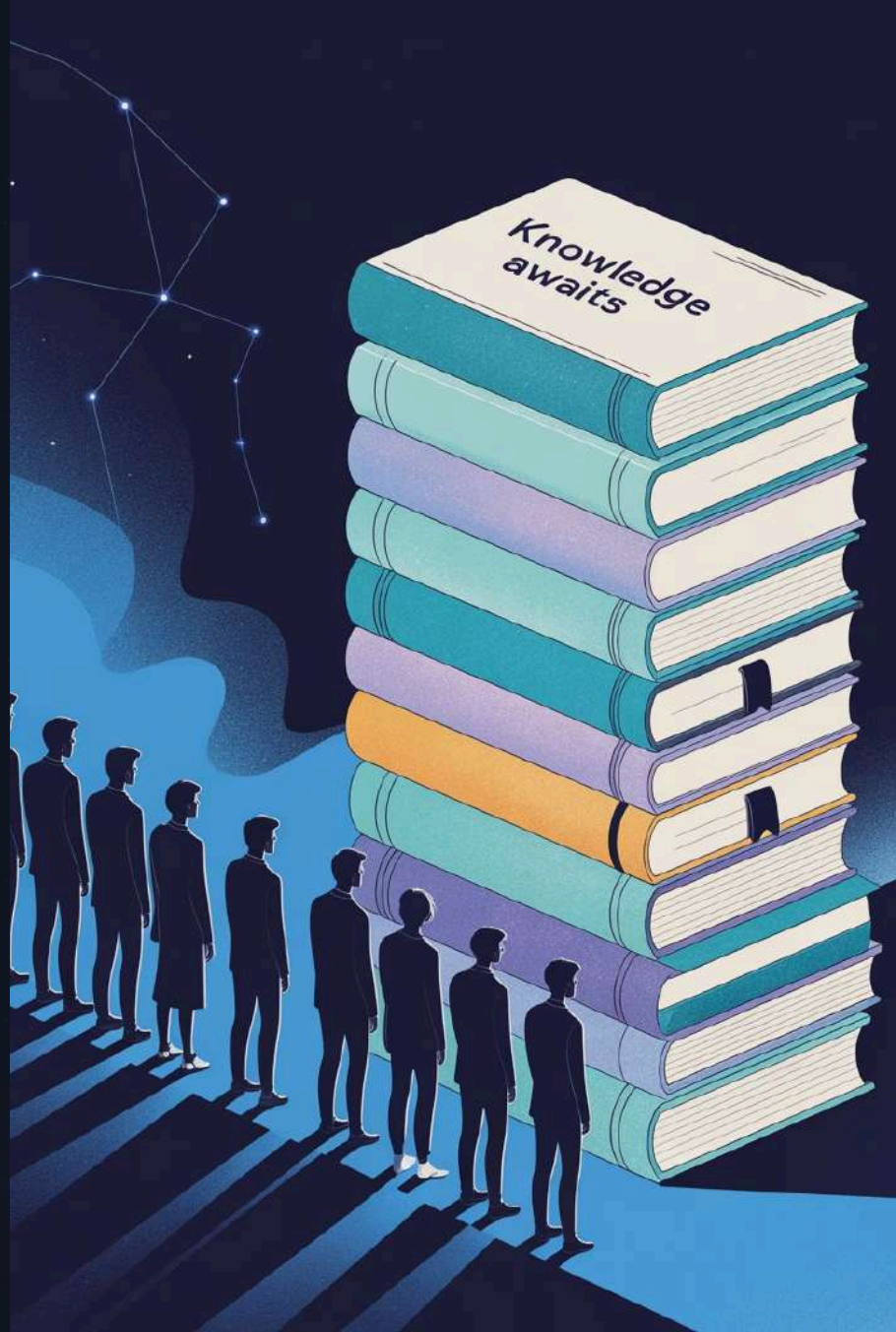
Stack and Queue: Fundamental Data Structures

Welcome to the lecture on two crucial linear data structures in programming. Today, we will explore the working principles of stacks and queues, examine their practical applications, and learn how to implement them using various methods.



What is a Stack?

LIFO Principle

Last In, First Out — the last one in, is the first one out. This is the main principle of a stack's operation, where access is only possible to the top element.

Real-world analogies:

- A stack of books on a table
- A stack of plates in a buffet
- The "Undo" button in a text editor
- Browser history (the "Back" button)

Key operations:

- `push(x)` — add an element to the top
- `pop()` — remove the top element
- `top()/peek()` — view the top element
- `empty()` — check if the stack is empty

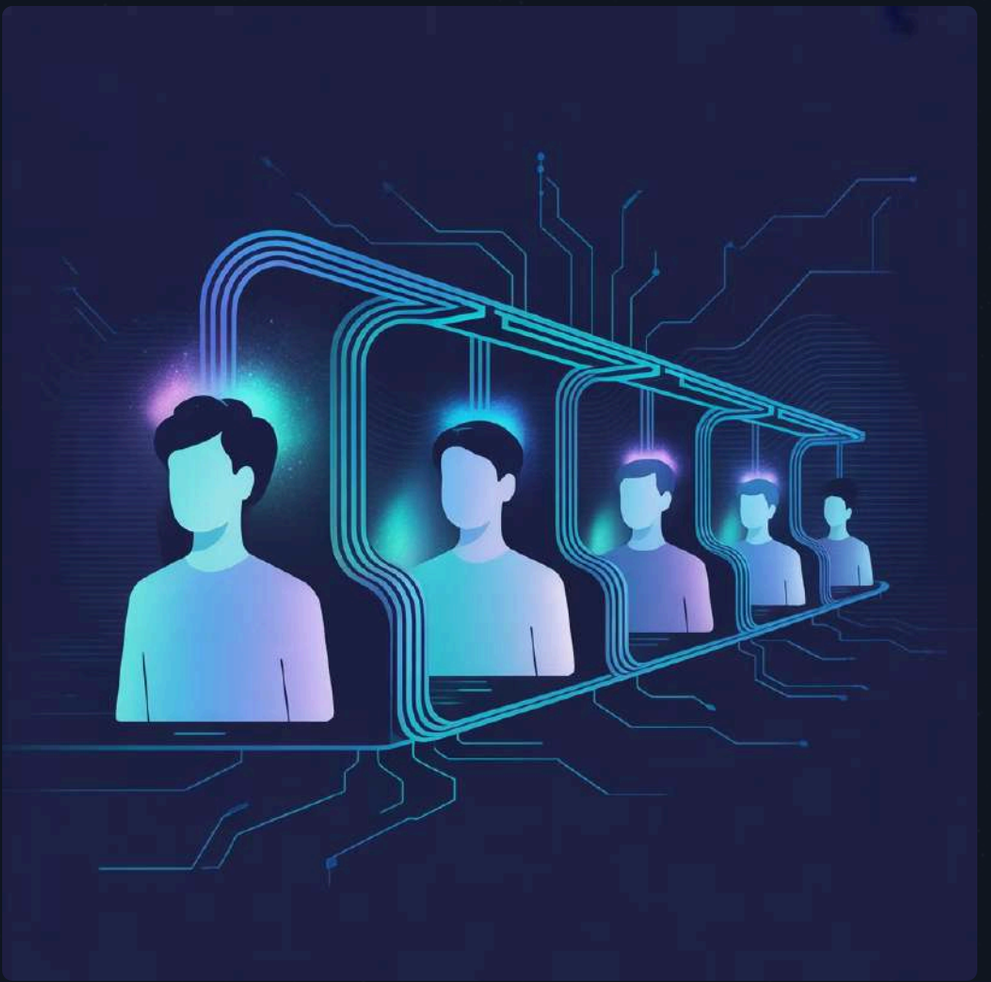
What is a Queue?

FIFO Principle

First In, First Out — the first one to arrive is the first one to leave. In a queue, elements are added to the end and removed from the beginning.

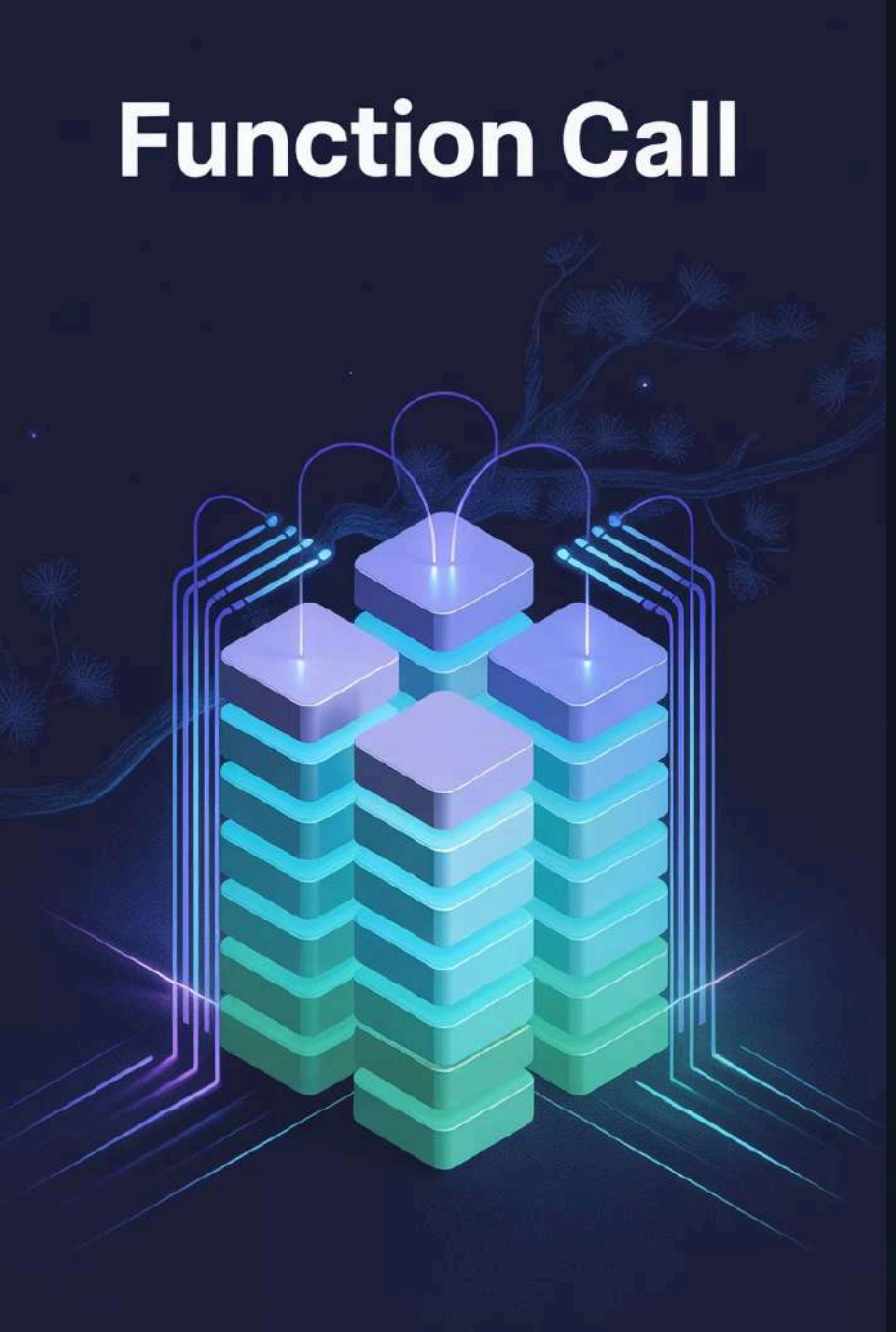
Real-world analogies:

- Queue in a store or bank
- Print queue on a printer
- Processing requests in order of arrival
- Data buffering in network protocols



Key operations:

- `enqueue(x)` — add element to the end
- `dequeue()` — remove element from the beginning
- `front()` — view the first element
- `empty()` — check if the queue is empty



Applications of Stack in Programming

Bracket Checking

The stack is ideal for checking the correctness of bracket placement in mathematical expressions and program code.

- Opening bracket → push
- Closing bracket → pop and compare

Function Call Stack

Each function call is placed onto the stack. When a function finishes, its data is removed from the stack, returning control to the previous function.

Reverse Polish Notation

Evaluating expressions in postfix notation: operands are pushed onto the stack, operators pop them for calculations.

Applications of Queues in Programming

Breadth-First Search (BFS)

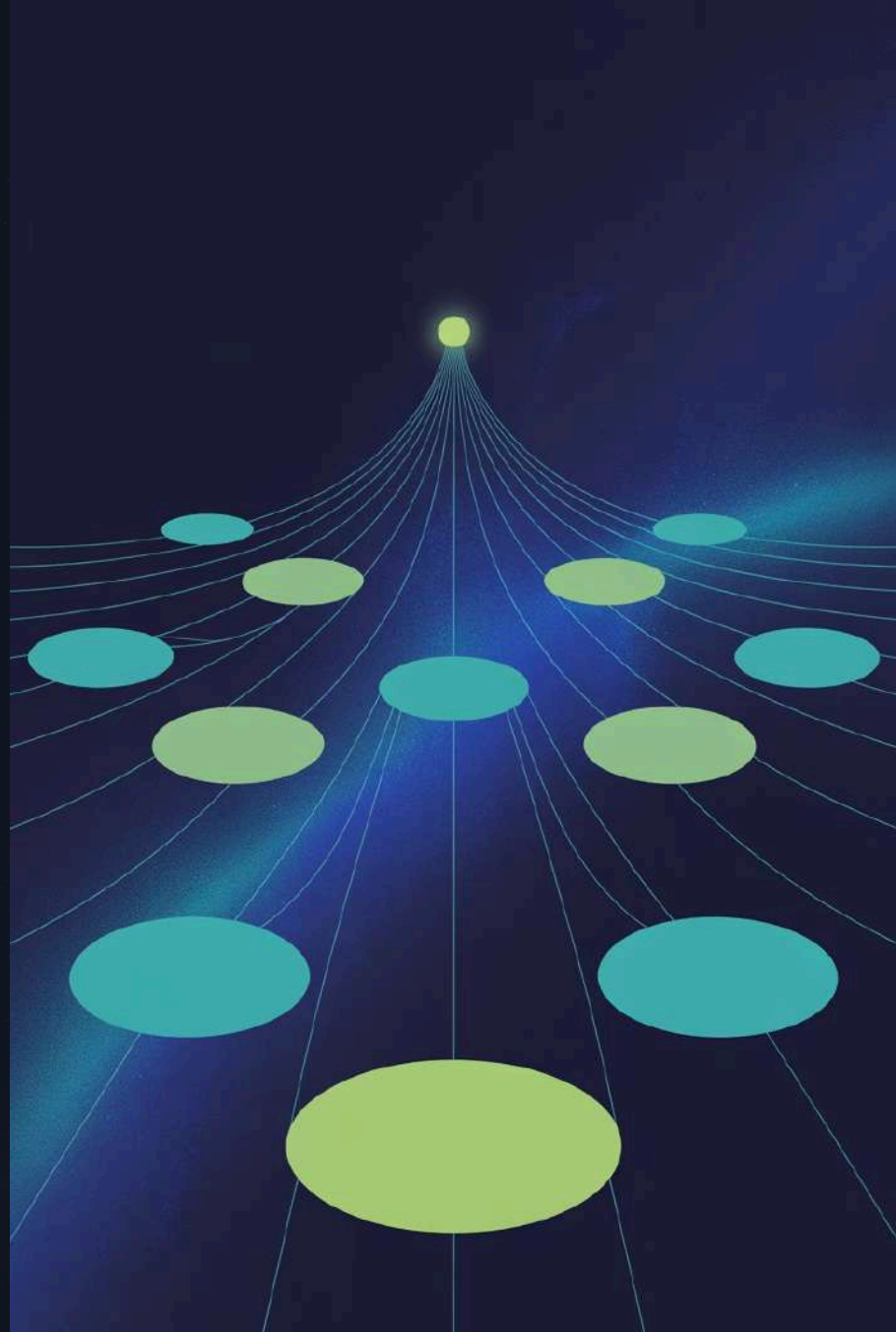
The BFS algorithm uses a queue to visit nodes level by level, ensuring the shortest path in an unweighted graph.

Task Scheduler

Operating systems use queues to manage processes and threads, ensuring fair resource allocation.

Data Buffering

Queues are used for temporary storage of data between processes with different processing speeds.



Array-based Stack Implementation

Data Structure

An array-based stack uses a static array of fixed size and a `top` variable that points to the index of the top element.

```
#include
using namespace std;

struct Stack {
    int arr[100];
    int top = -1;

    void push(int x) {
        if (top == 99) {
            cout << "Stack overflow!" << endl;
            return;
        }
        arr[++top] = x;
    }

    void pop() {
        if (top == -1) {
            cout << "Stack underflow!" << endl;
            return;
        }
        top--;
    }

    int peek() {
        if (top == -1) return -1;
        return arr[top];
    }

    bool empty() {
        return top == -1;
    }
};
```

Advantages

Simple implementation, fast O(1) access, minimal memory overhead

Disadvantages

Fixed size, potential for overflow, inefficient memory usage

Linked List Stack Implementation

Dynamic Structure

A stack based on a linked list uses nodes connected by pointers. The `head` variable points to the top of the stack.

```
struct Node {
    int data;
    Node* next;
};

struct Stack {
    Node* head = nullptr;

    void push(int x) {
        Node* newNode = new Node{x, head};
        head = newNode;
    }

    void pop() {
        if (!head) {
            cout << "Stack is empty!" << endl;
            return;
        }
        Node* temp = head;
        head = head->next;
        delete temp;
    }

    int peek() {
        if (!head) return -1;
        return head->data;
    }

    bool empty() {
        return head == nullptr;
    }
};
```

Advantages:

- Dynamic size
- Efficient memory usage
- No limitations on the number of elements

Disadvantages:

- Additional memory for pointers
- Slower due to pointer operations
- Potential for memory leaks

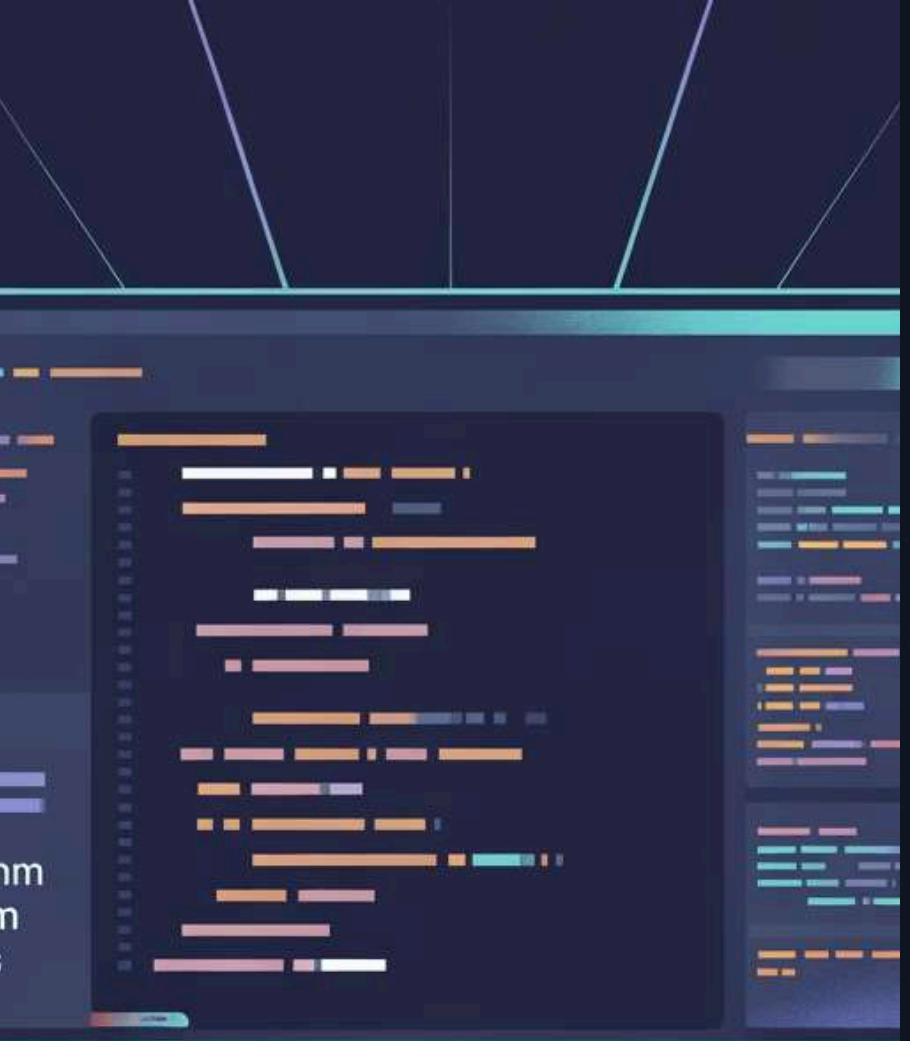
Queue Implementation using an Array

Circular Buffer

An efficient array-based queue implementation uses a circular buffer with two pointers: `front` and `rear`.

```
struct Queue {
    int arr[100];
    int front = 0, rear = 0, size = 0;

    void enqueue(int x) {
        if (size == 100) {
            cout << "Queue overflow!" << endl;
            return;
        }
        arr[rear] = x;
        rear = (rear + 1) % 100;
        size++;
    }

    void dequeue() {
        if (size == 0) {
            cout << "Queue is empty!" << endl;
            return;
        }
        front = (front + 1) % 100;
        size--;
    }

    int peek() {
        if (size == 0) return -1;
        return arr[front];
    }

    bool empty() {
        return size == 0;
    }
};
```

Important! Using the modulo (%) operation allows creating a circular buffer, efficiently utilizing the entire array without shifting elements.



Conclusion and Practical Questions

O(1)

Time Complexity

All main stack and queue operations are performed in constant time

2

Implementation Methods

Array (static) and linked list (dynamic)

100%

Practical Importance

Used in all modern programming languages

Self-Assessment Questions:

01

Principles of Operation

What is the fundamental difference between LIFO and FIFO? Give real-life examples.

02

Choice of Implementation

When to prefer an array, and when a linked list? What factors influence the choice?

03

Circular Buffer

Why is a queue more efficient to implement as a circular buffer rather than a regular array?

04

Practical Application

Name algorithms that use stacks and queues. Explain their role in each case.

Next step: Practical exercises in the seminar — implementing a stack for bracket checking and a queue for supermarket checkout modeling!