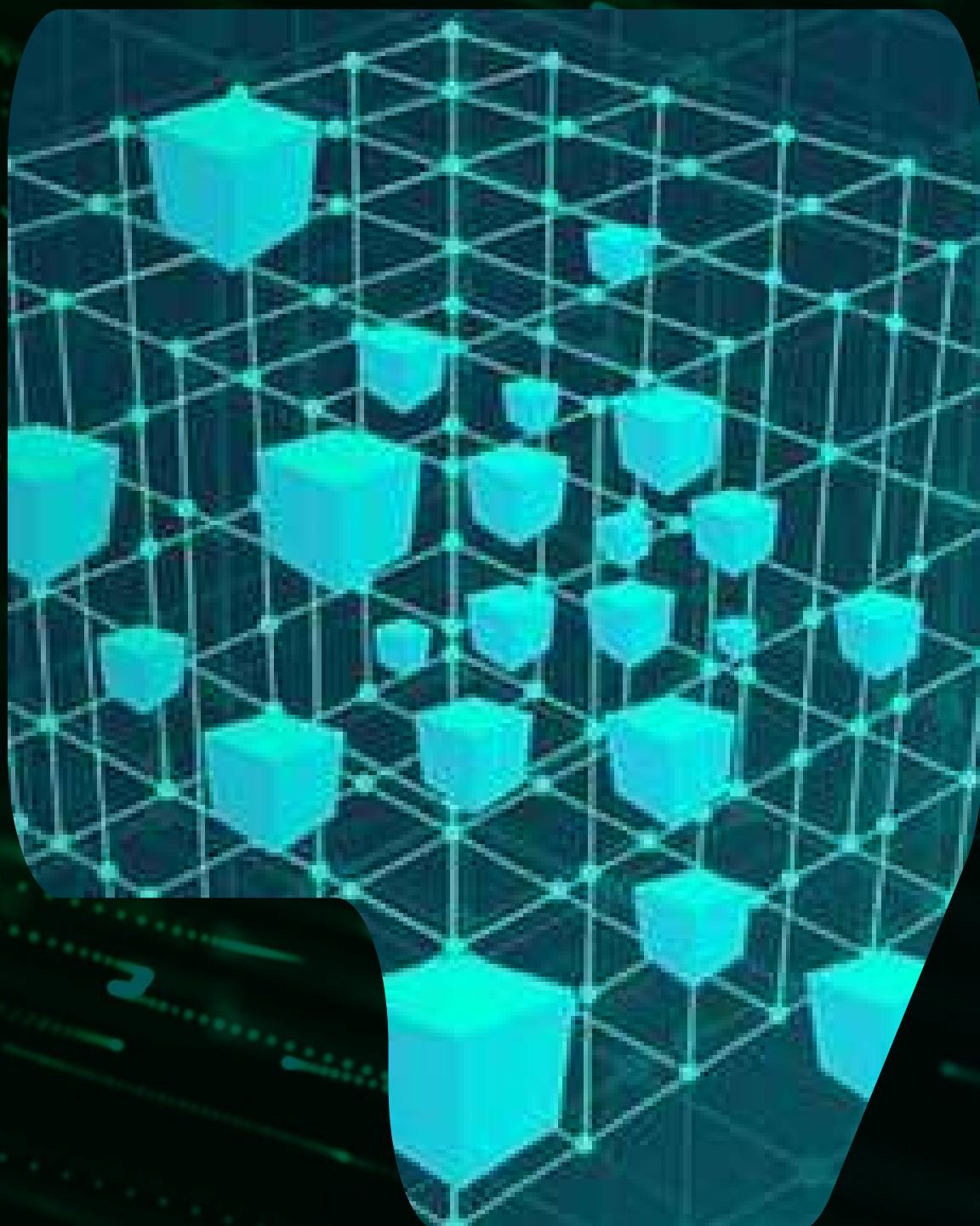


ASSIGNMENT 1

P R E S E N T A T I O N

NGUYEN TRONG THIEP - BH01619

DSA LÀ GÌ?



DSA là viết tắt của Data Structures and Algorithms (Cấu trúc Dữ liệu và Thuật toán). Đây là lĩnh vực tổ chức và xử lý dữ liệu hiệu quả, bao gồm:

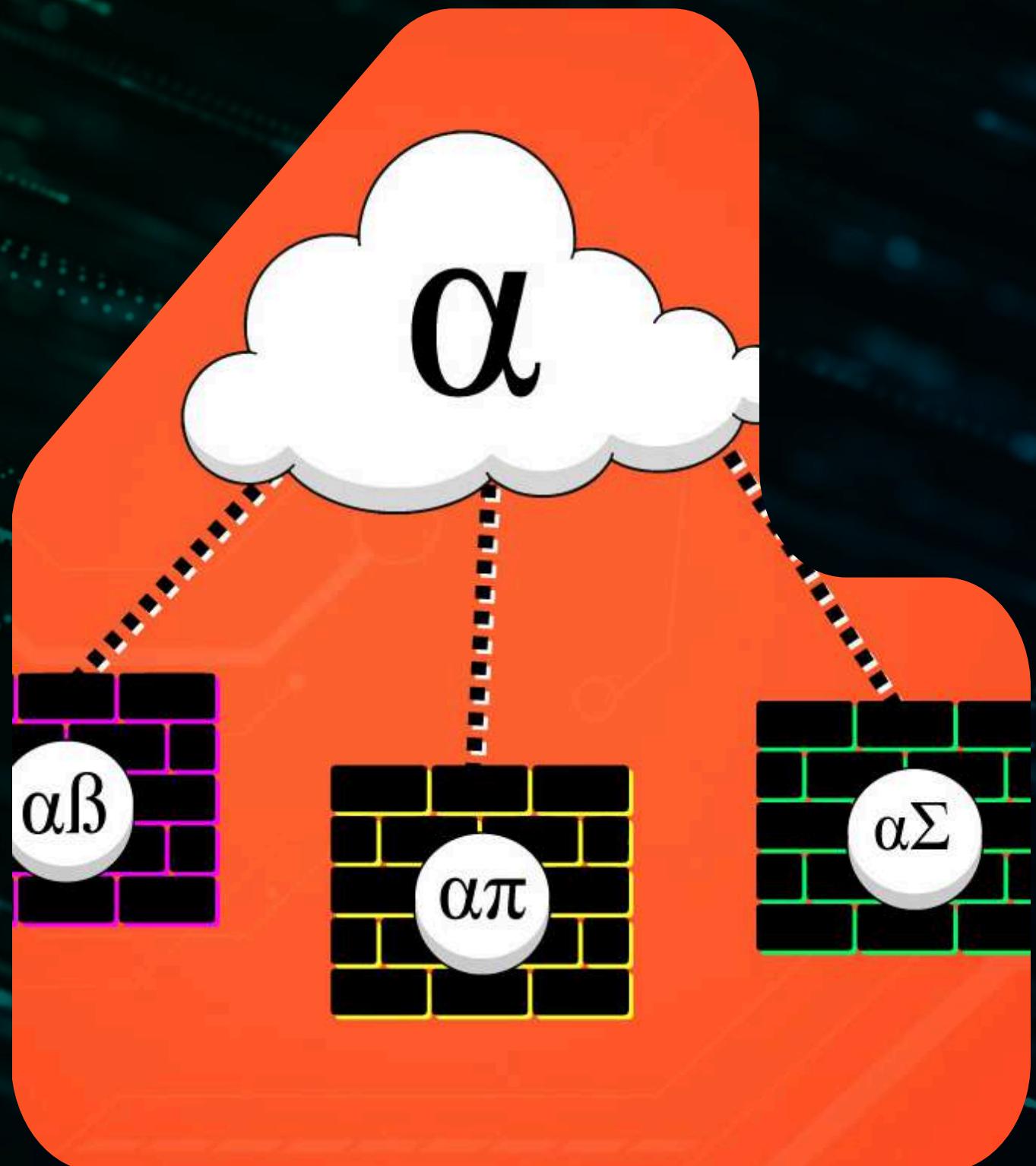
- Cấu trúc dữ liệu: Cách lưu trữ và tổ chức dữ liệu như mảng, ngăn xếp, hàng đợi, cây và đồ thị.
- Thuật toán: Các bước giải quyết vấn đề, như sắp xếp, tìm kiếm, và tìm đường đi ngắn nhất.

DSA giúp tối ưu hóa mã nguồn, nâng cao hiệu suất và giải quyết các bài toán phức tạp.

ADT LÀ GÌ?

ADT (Abstract Data Type - Kiểu dữ liệu trừu tượng)** là một mô hình lý thuyết của cấu trúc dữ liệu, chỉ định cách dữ liệu được tổ chức và các thao tác có thể thực hiện mà không quan tâm đến cách triển khai. ADT tập trung vào các hoạt động như thêm, xóa, hay truy xuất dữ liệu hơn là chi tiết thực thi.

ADT giúp trừu tượng hóa cấu trúc dữ liệu, làm cho việc thiết kế và thao tác dữ liệu dễ dàng hơn, tối ưu hóa mã nguồn, và đảm bảo tính linh hoạt khi thay đổi cách triển khai mà không ảnh hưởng đến các phần khác của chương trình.

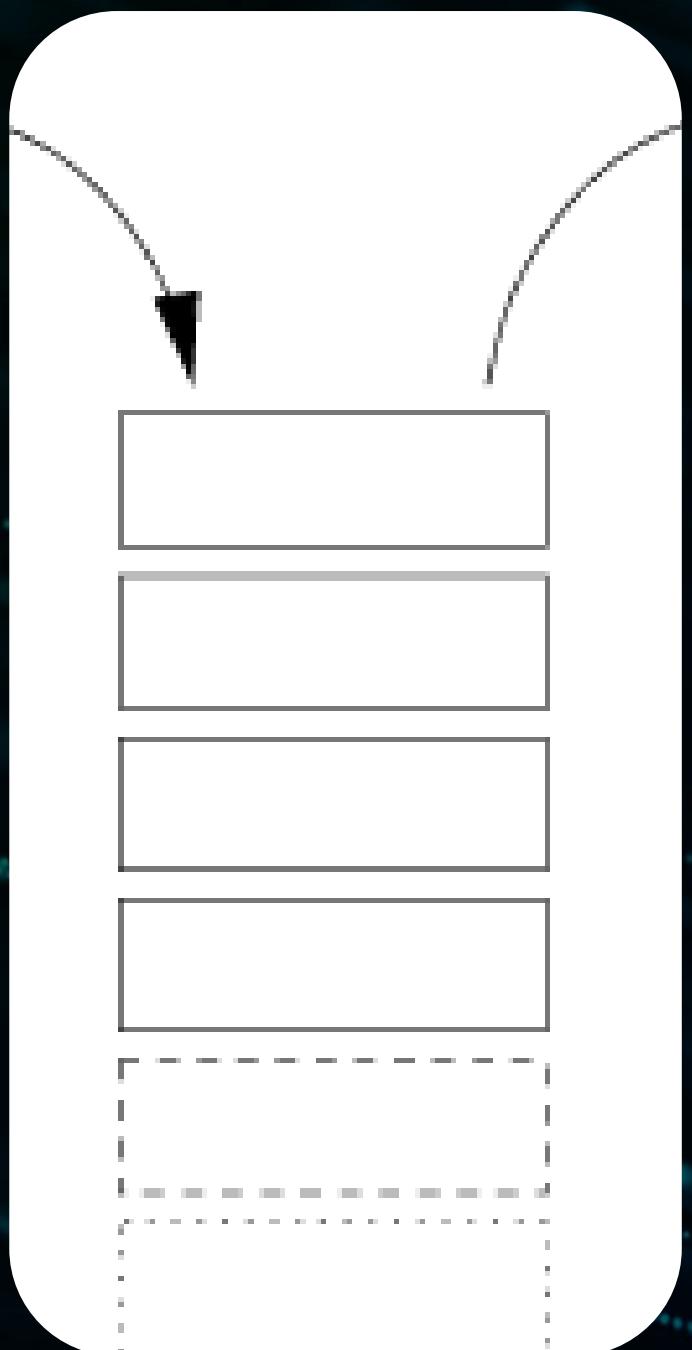


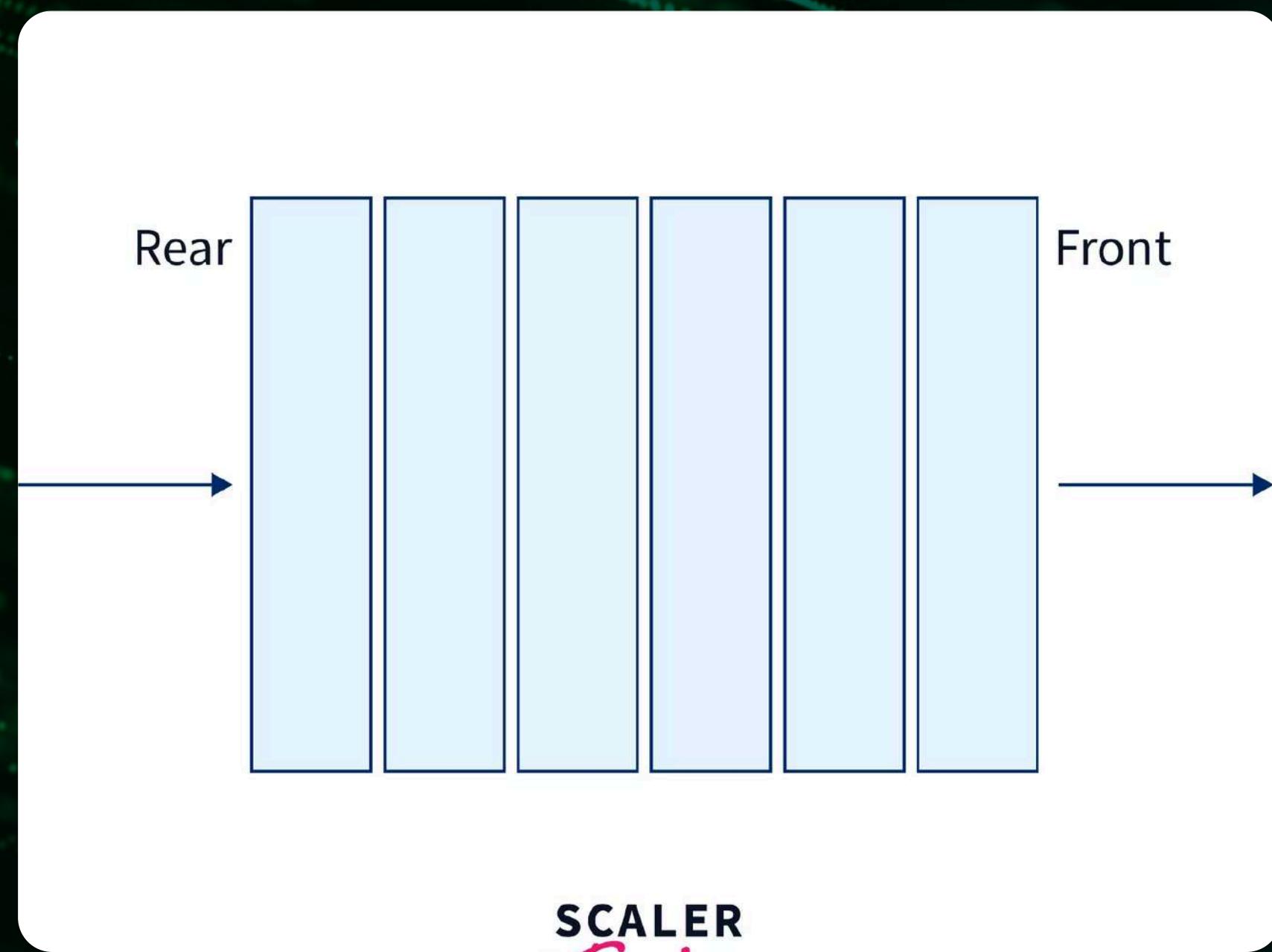


GIỚI THIỆU VỀ STACK ADT

Stack ADT là kiểu dữ liệu trùu tượng hoạt động theo cơ chế LIFO (Last In, First Out), nghĩa là phần tử cuối cùng được thêm vào sẽ là phần tử đầu tiên được lấy ra. Các thao tác cơ bản của Stack gồm **push** (thêm phần tử vào đỉnh Stack) và **pop** (lấy phần tử từ đỉnh Stack).

Ứng dụng của Stack trong lập trình bao gồm: quản lý hàm gọi (call stack) trong quá trình thực thi chương trình, tính toán biểu thức trong toán học, và xử lý tính năng undo/redo trong phần mềm.





GIỚI THIỆU VỀ QUEUE ADT

- Queue ADT là kiểu dữ liệu trừu tượng hoạt động theo cơ chế FIFO (First In, First Out), nghĩa là phần tử đầu tiên được thêm vào sẽ là phần tử đầu tiên được lấy ra. Các thao tác chính của Queue bao gồm enqueue (thêm phần tử vào cuối hàng) và dequeue (lấy phần tử từ đầu hàng).
- Ứng dụng của Queue bao gồm: quản lý hàng đợi trong hệ thống xử lý tác vụ (như lập lịch CPU), xử lý dữ liệu trong hàng đợi máy in, điều khiển hàng đợi trong mạng máy tính và mô phỏng hệ thống xếp hàng.

SO SÁNH GIỮA STACK VÀ QUEUE

Cơ chế hoạt động: Stack hoạt động theo nguyên tắc LIFO (Last In, First Out), nghĩa là phần tử cuối cùng được thêm vào sẽ được lấy ra đầu tiên.

Ngược lại, Queue hoạt động theo nguyên tắc FIFO (First In, First Out), tức là phần tử đầu tiên được thêm vào sẽ được lấy ra đầu tiên.

Ưu điểm:

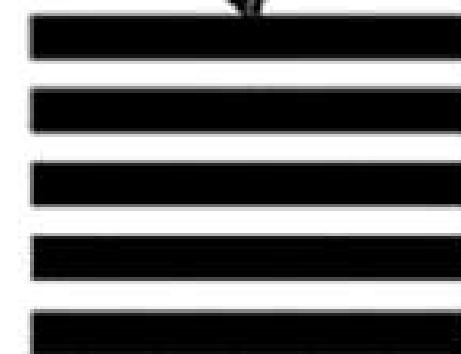
- Stack: Dễ dàng triển khai và sử dụng cho các tác vụ cần quản lý thứ tự, như duyệt cây hoặc quản lý hàm gọi.
- Queue: Thích hợp cho các tình huống cần xử lý dữ liệu theo thứ tự, như hàng đợi tác vụ và mạng.

Nhược điểm:

- Stack: Có thể gây ra tràn ngăn xếp (stack overflow) nếu số lượng phần tử vượt quá giới hạn.
- Queue: Có thể gặp vấn đề về hiệu suất nếu không được triển khai tối ưu (chẳng hạn như trong hàng đợi vòng).

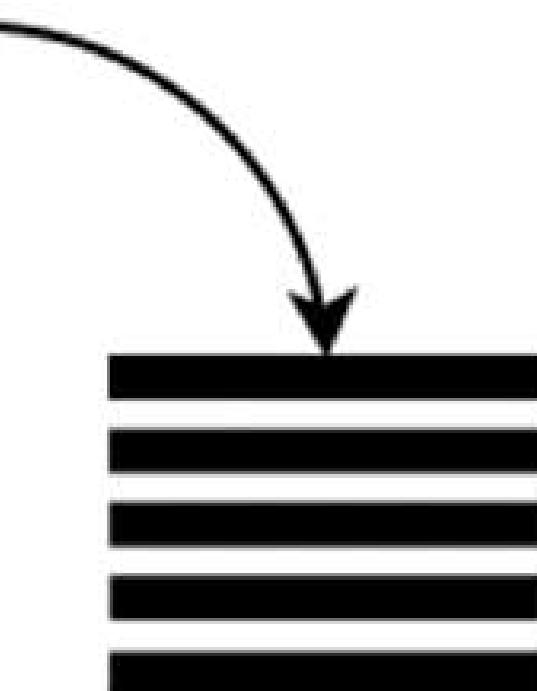
Stack:

Last in, first out



Queue:

First in, first out





PHÂN BIỆT CÁCH SỬ DỤNG STACK VÀ QUEUE

Ứng dụng của Stack:

1. Duyệt cây: Stack được sử dụng trong các thuật toán duyệt cây như duyệt theo chiều sâu (DFS) để lưu trữ các nút cần thăm.
2. Quản lý hàm gọi: Stack lưu trữ thông tin về các hàm đã được gọi và giúp quản lý việc trả về (return) từ các hàm trong chương trình.

Ứng dụng của Queue:

1. Quản lý hàng đợi: Queue được sử dụng để xử lý các tác vụ trong hàng đợi, như trong hệ thống máy in hay dịch vụ khách hàng.
2. Lập lịch CPU: Queue giúp quản lý các tiến trình đang chờ xử lý, đảm bảo rằng các tác vụ được thực hiện theo thứ tự hợp lý.

CÁC PHƯƠNG PHÁP TRIỂN KHAI STACK

Triển khai bằng mảng (Array-based):

- Sử dụng một mảng để lưu trữ các phần tử của Stack.
- Một biến chỉ số (top) được sử dụng để theo dõi vị trí của phần tử trên cùng.
- Khi thêm (push) một phần tử, ta tăng chỉ số top và gán giá trị vào mảng; khi lấy (pop) một phần tử, ta trả về giá trị tại chỉ số top và giảm chỉ số top.
- **Ưu điểm:** Dễ dàng triển khai và truy cập nhanh chóng đến các phần tử.
- **Nhược điểm:** Kích thước của Stack phải được xác định trước, có thể dẫn đến tràn ngăn xếp (stack overflow) nếu mảng đầy.

Triển khai bằng danh sách liên kết (Linked List-based):

- Sử dụng một danh sách liên kết để lưu trữ các phần tử của Stack, mỗi nút chứa giá trị và con trỏ đến nút tiếp theo.
- Thao tác push thêm một nút mới vào đầu danh sách, trong khi pop lấy nút đầu tiên và cập nhật con trỏ.
- **Ưu điểm:** Kích thước của Stack có thể thay đổi linh hoạt và không có giới hạn về số lượng phần tử.
- **Nhược điểm:** Phức tạp hơn trong triển khai và có chi phí bộ nhớ cao hơn do cần lưu trữ thêm con trỏ.

CÁC PHƯƠNG PHÁP TRIỂN KHAI QUEUE

```
#include <iostream>
using namespace std;

// Định nghĩa cấu trúc Node để lưu trữ phần tử trong hàng đợi
struct Node {
    int data;
    Node* next;
    Node(int val) : data(val), next(nullptr) {}
};

// Định nghĩa lớp Queue để quản lý hàng đợi
class Queue {
private:
    Node* front; // Con trỏ front trỏ đến phần tử đầu tiên của hàng đợi
    Node* rear; // Con trỏ rear trỏ đến phần tử cuối cùng của hàng đợi

public:
    // Constructor để khởi tạo hàng đợi rỗng
    Queue() : front(nullptr), rear(nullptr) {}

    // Phương thức enqueue để thêm một phần tử vào cuối hàng đợi
    void enqueue(int value) {
        Node* newNode = new Node(value);
        if (isEmpty()) {
            front = rear = newNode;
        } else {
            rear->next = newNode;
            rear = newNode;
        }
    }

    // Phương thức dequeue để loại bỏ phần tử đầu tiên khỏi hàng đợi và trả về giá trị của nó
    int dequeue() {
        if (isEmpty()) {
            cout << "Queue is empty" << endl;
            return -1;
        } else {
            Node* temp = front;
            int value = temp->data;
            front = front->next;
            delete temp;
            return value;
        }
    }
}
```

1. Triển khai bằng mảng (Array-based Queue):

- Sử dụng một mảng để lưu trữ các phần tử của Queue.
- Hai biến chỉ số (front và rear) theo dõi vị trí của phần tử đầu và cuối hàng.
- Khi enqueue (thêm) phần tử, chỉ số rear được tăng; khi dequeue (lấy) phần tử, chỉ số front được tăng.
- Nhược điểm: Có thể dẫn đến tràn hàng đợi (queue overflow) nếu mảng đầy và lãng phí bộ nhớ khi không sử dụng hết không gian.

2. Triển khai bằng danh sách liên kết (Linked List-based Queue):

- Sử dụng danh sách liên kết để lưu trữ các phần tử của Queue, mỗi nút chứa giá trị và con trỏ đến nút tiếp theo.
- Enqueue thêm một nút mới vào cuối danh sách; dequeue lấy nút đầu tiên.
- **Ưu điểm**: Kích thước có thể thay đổi linh hoạt mà không giới hạn.

3. Giới thiệu Hàng đợi tròn (Circular Queue):

- Triển khai hàng đợi bằng cách nối cuối mảng với đầu mảng, giúp sử dụng không gian hiệu quả hơn.
- Khi rear đạt đến cuối mảng, nó quay lại đầu mảng nếu còn chỗ trống.
- Ưu điểm: Giảm thiểu lãng phí bộ nhớ và tránh tràn hàng đợi khi có không gian trống.

CẤU TRÚC DỮ LIỆU MẢNG CHO STACK VÀ QUEUE

- Stack:
- Mảng được sử dụng để lưu trữ các phần tử, với một chỉ số (top) theo dõi vị trí của phần tử trên cùng.
- Thao tác push (thêm) tăng chỉ số top và gán giá trị vào mảng; thao tác pop (lấy) trả về giá trị tại chỉ số top và giảm chỉ số top.
- Chi phí thao tác: Cả hai thao tác push và pop đều có độ phức tạp $O(1)$.
- Queue:
- Mảng được sử dụng với hai chỉ số (front và rear) để theo dõi phần tử đầu và cuối hàng.
- Thao tác enqueue (thêm) tăng chỉ số rear và gán giá trị vào mảng; thao tác dequeue (lấy) trả về giá trị tại chỉ số front và tăng chỉ số front.
- Chi phí thao tác: Thao tác enqueue và dequeue đều có độ phức tạp $O(1)$, nhưng có thể gặp vấn đề khi không còn không gian trống nếu sử dụng hàng đợi tròn.

CẤU TRÚC LINK LIST CHO STACK VÀ QUEUE

Cách triển khai Stack bằng danh sách liên kết (Linked List):

1. Nút (Node): Định nghĩa nút bao gồm giá trị (data) và con trỏ (next) đến nút tiếp theo.
2. Stack: Tạo lớp Stack với con trỏ (top) theo dõi nút trên cùng.
3. Push: Thêm nút mới vào đầu danh sách, cập nhật con trỏ top đến nút mới.
4. Pop: Lấy nút từ đầu danh sách, cập nhật con trỏ top đến nút tiếp theo và trả về giá trị nút đã lấy.
5. Kiểm tra rỗng: Phương thức kiểm tra xem Stack có trống hay không.
6. Peek: Truy cập phần tử trên cùng mà không lấy ra.

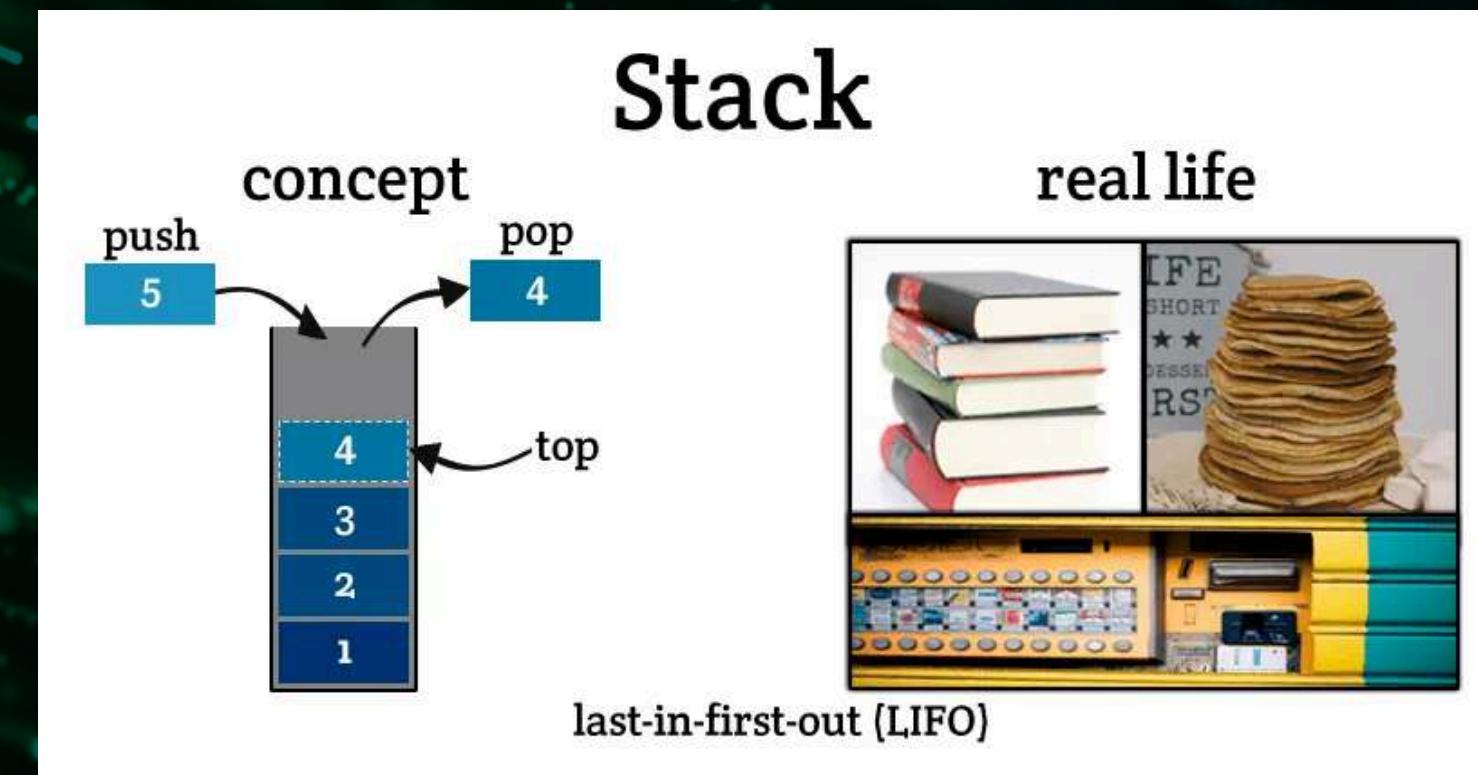
Kích thước linh hoạt, không bị giới hạn bởi mảng, và không gặp vấn đề tràn ngăn xếp.

Cách triển khai Queue bằng danh sách liên kết (Linked List):

1. Nút (Node): Định nghĩa nút bao gồm giá trị (data) và con trỏ (next) đến nút tiếp theo.
2. Queue: Tạo lớp Queue với hai con trỏ (front và rear) theo dõi phần tử đầu và cuối hàng.
3. Enqueue: Thêm nút mới vào cuối danh sách; nếu Queue rỗng, cập nhật cả front và rear.
4. Dequeue: Lấy nút từ đầu danh sách, cập nhật con trỏ front đến nút tiếp theo và trả về giá trị nút đã lấy.
5. Kiểm tra rỗng: Phương thức kiểm tra xem Queue có trống hay không.

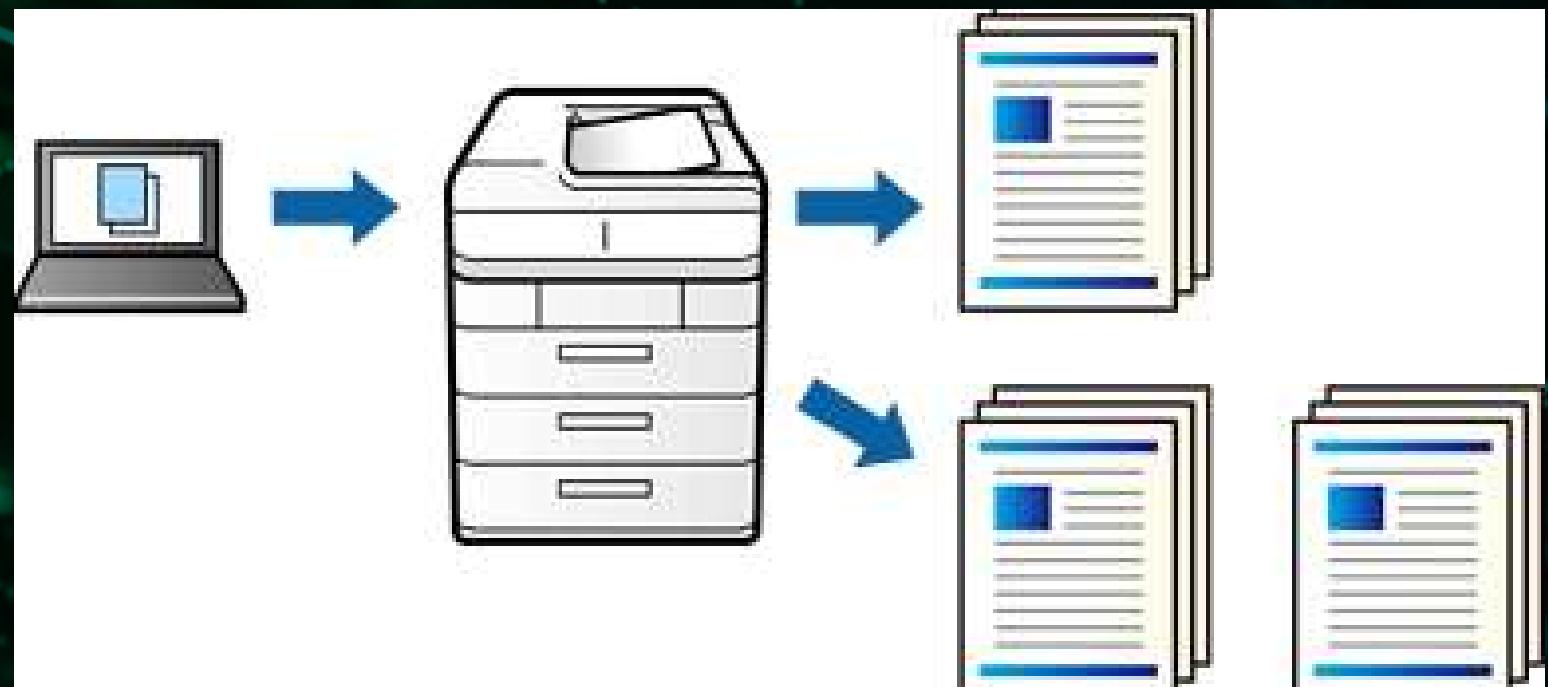
Kích thước linh hoạt, không bị giới hạn bởi mảng, giúp quản lý hàng đợi hiệu quả hơn.

CÁC TÌNH HUỐNG THỰC TẾ CHO STACK



Undo/Redo (Stack): Trong các ứng dụng chỉnh sửa văn bản hoặc đồ họa, chức năng undo/redo thường sử dụng cấu trúc dữ liệu stack. Khi người dùng thực hiện hành động, nó được đẩy vào stack. Khi cần hoàn tác, ứng dụng lấy hành động gần nhất ra khỏi stack. Tương tự, redo sẽ lấy hành động từ stack phụ lưu (nếu có).

CÁC TÌNH HUỐNG THỰC TẾ CHO QUEUE



Hệ thống hàng đợi (Queue): Hệ thống hàng đợi thường được sử dụng trong quản lý tác vụ và xử lý dữ liệu. Ví dụ, trong một máy in, tài liệu được gửi đến hàng đợi in. Các tài liệu được xử lý theo thứ tự mà chúng được nhận (first-in, first-out). Hệ thống giao thông công cộng cũng sử dụng hàng đợi để quản lý hành khách lên xe, đảm bảo mọi người được phục vụ theo thứ tự hợp lý.

THUẬT TOÁN SẮP XẾP(SORTING ALGORITHMS)

Sorting Algorithms

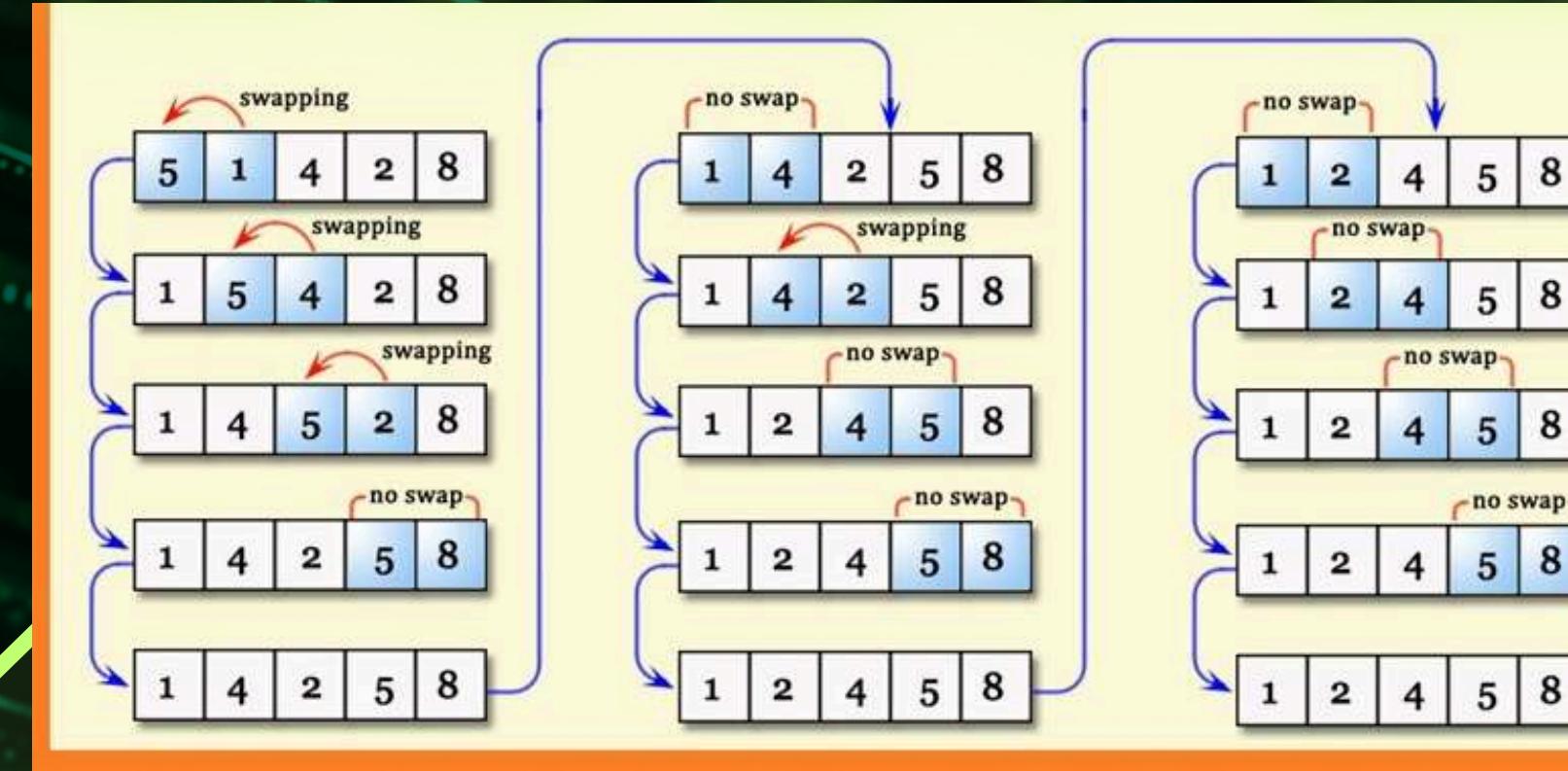


Định nghĩa: Thuật toán sắp xếp là một tập hợp các bước hoặc quy trình để tổ chức dữ liệu trong một thứ tự xác định, như tăng dần hoặc giảm dần.

Vai trò: Thuật toán sắp xếp giúp cải thiện hiệu suất tìm kiếm và quản lý dữ liệu, cho phép truy xuất và phân tích thông tin một cách nhanh chóng và hiệu quả hơn.

Tầm quan trọng: Trong DSA, việc lựa chọn thuật toán sắp xếp phù hợp có thể ảnh hưởng lớn đến hiệu suất tổng thể của ứng dụng. Thuật toán sắp xếp tối ưu giúp giảm thời gian xử lý, tiết kiệm bộ nhớ, và cải thiện trải nghiệm người dùng trong các ứng dụng như cơ sở dữ liệu, giao diện người dùng và phân tích dữ liệu.

THUẬT TOÁN SẮP XẾP 1 – BUBBLE SORT

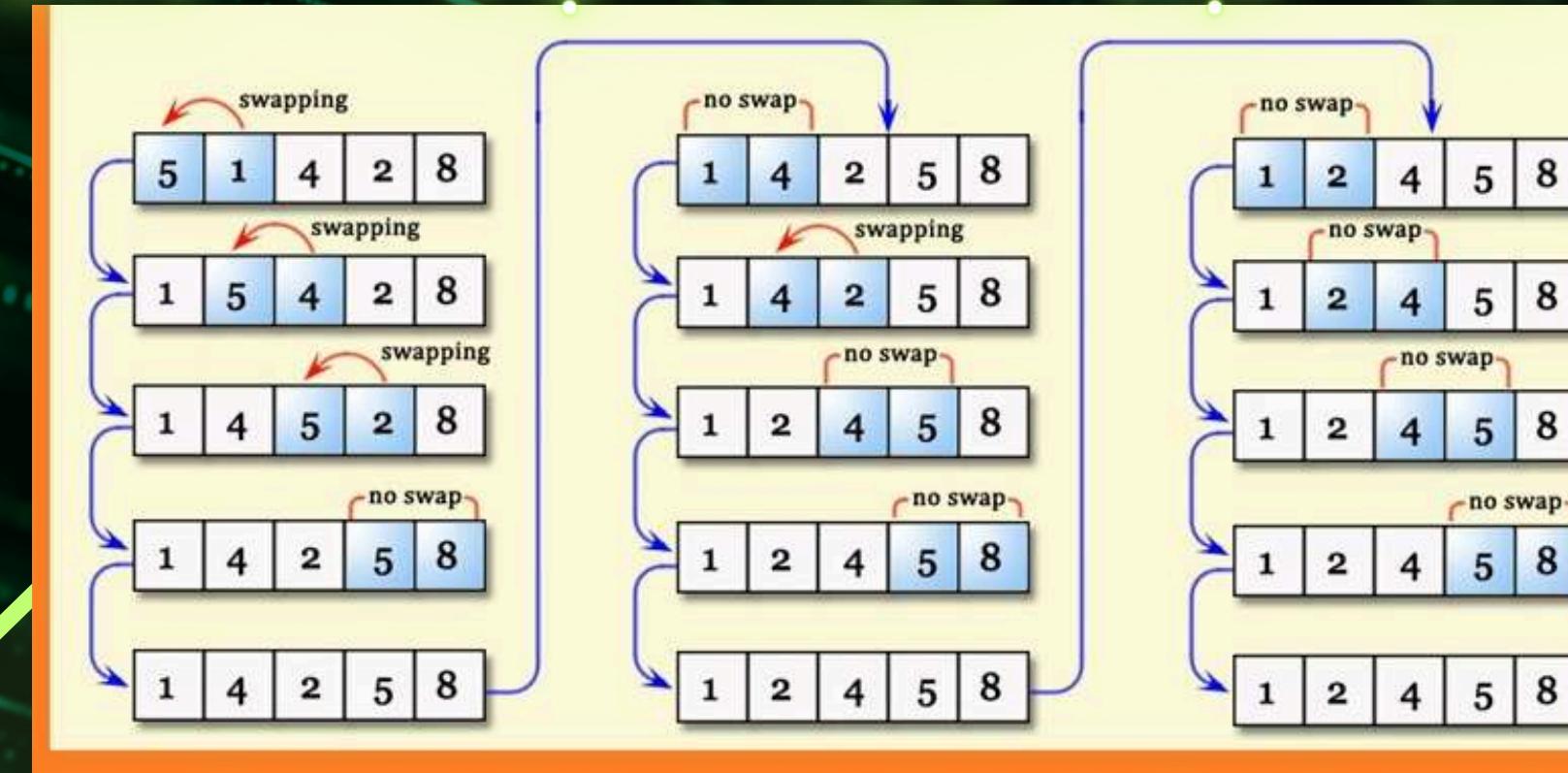


Định nghĩa Bubble Sort: Bubble Sort là thuật toán sắp xếp đơn giản hoạt động bằng cách lặp qua danh sách, so sánh từng cặp phần tử liên tiếp và hoán đổi chúng nếu chúng không theo thứ tự. Quá trình này được lặp lại cho đến khi danh sách được sắp xếp.

Các bước thực hiện Bubble Sort:

- Lặp qua mảng nhiều lần.
- Trong mỗi lượt lặp, so sánh các cặp phần tử liền kề.
- Nếu phần tử trước lớn hơn phần tử sau, hoán đổi chúng.
- Tiếp tục so sánh và hoán đổi cho đến cuối mảng.
- Lặp lại quá trình cho đến khi không có hoán đổi nào xảy ra trong một lượt lặp, tức là mảng đã được sắp xếp.

BUBBLE SORT – ĐỘ PHỨC TẠP VÀ HIỆU SUẤT



Độ phức tạp thời gian của Bubble Sort: Bubble Sort có độ phức tạp $O(n^2)$ trong trường hợp trung bình và xấu nhất, do phải lặp qua mảng nhiều lần để đảm bảo các phần tử đã được sắp xếp.

Ưu điểm:

- Đơn giản và dễ triển khai: Bubble Sort có thuật toán dễ hiểu và phù hợp cho người mới học.
- Không cần bộ nhớ phụ: Sắp xếp tại chỗ (in-place), không cần mảng phụ.

Nhược điểm:

- Hiệu suất thấp: Chậm với mảng lớn do phải thực hiện nhiều phép so sánh và hoán đổi.
- Không hiệu quả: Ít được sử dụng trong thực tế khi có nhiều thuật toán sắp xếp hiệu quả hơn như Quick Sort, Merge Sort.

THUẬT TOÁN SẮP XẾP 2 - QUICK SORT



Định nghĩa Quick Sort: Quick Sort là thuật toán sắp xếp dựa trên phương pháp chia để trị, hoạt động bằng cách chọn một phần tử làm chốt (pivot), sau đó phân chia mảng thành hai phần: các phần tử nhỏ hơn pivot ở bên trái và lớn hơn pivot ở bên phải.

Cách hoạt động:

1. Chọn một phần tử làm pivot (thường là phần tử đầu, cuối, hoặc ngẫu nhiên).
2. Phân chia: Sắp xếp lại các phần tử sao cho các phần tử nhỏ hơn pivot nằm bên trái, lớn hơn pivot nằm bên phải.
3. Đệ quy: Áp dụng Quick Sort lên hai mảng con (trái và phải) đã được chia.
4. Kết quả cuối cùng là mảng được sắp xếp khi tất cả mảng con có một phần tử.



Độ phức tạp thời gian của Quick Sort: Quick Sort có độ phức tạp trung bình là $O(n \log n)$, nhưng trong trường hợp xấu nhất là $O(n^2)$.

Trường hợp tốt:

- Xảy ra khi mảng được chia đều tại mỗi lần chọn pivot.
- Độ sâu của cây phân chia là $\log(n)$, giúp Quick Sort đạt hiệu suất $O(n \log n)$.

Trường hợp xấu:

- Xảy ra khi pivot được chọn không đều, như chọn pivot là phần tử lớn nhất hoặc nhỏ nhất trong mảng đã sắp xếp.
- Khi đó, Quick Sort phải phân chia không cân đối, dẫn đến độ phức tạp $O(n^2)$.

SO SÁNH ĐỘ PHỨC TẠP CỦA BUBBLE SORT VÀ QUICK SORT

- Bảng so sánh độ phức tạp:

Thuật toán	Trường hợp tốt	Trung bình	Trường hợp xấu
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$

- Đánh giá lựa chọn:

- Bubble Sort: Thích hợp cho mảng nhỏ hoặc khi cần thuật toán đơn giản để sắp xếp dữ liệu ít thay đổi. Không hiệu quả với mảng lớn.
- Quick Sort: Lý tưởng cho mảng lớn nhờ hiệu suất trung bình tốt hơn ($O(n \log n)$). Tuy nhiên, cần cẩn trọng với trường hợp xấu khi dữ liệu đã sắp xếp hoặc pivot không tốt.

ỨNG DỤNG THỰC TẾ CỦA BUBBLE SORT VÀ QUICK SORT

Các tình huống phù hợp để sử dụng Bubble Sort:

- Khi kích thước mảng nhỏ, đơn giản (dưới 100 phần tử) và không đòi hỏi hiệu suất cao.
- Trong các bài tập học thuật hoặc để minh họa cách hoạt động của các thuật toán sắp xếp cơ bản.
- Khi cần thuật toán dễ hiểu, dễ triển khai mà không cần tối ưu hóa hiệu suất.

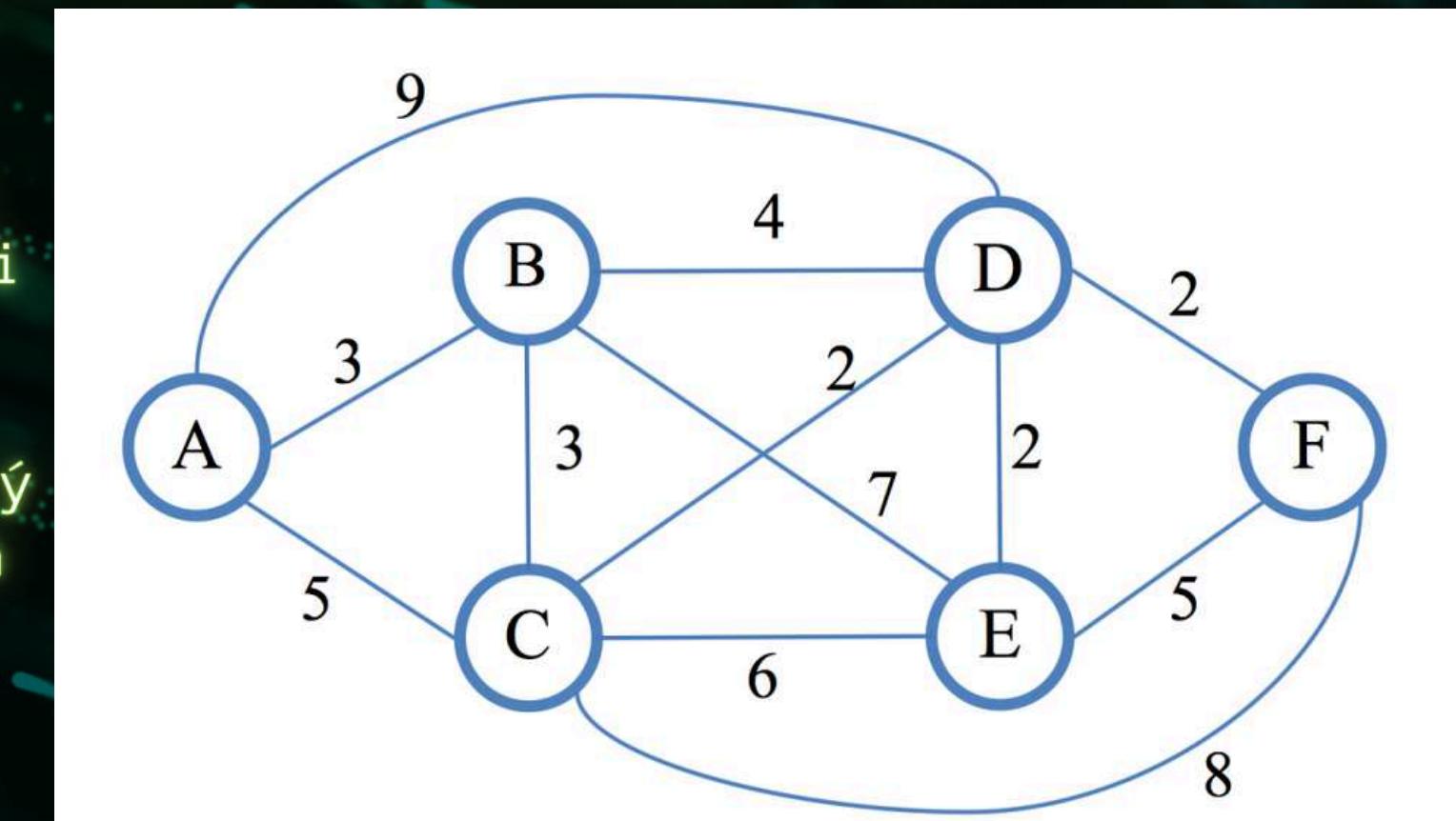
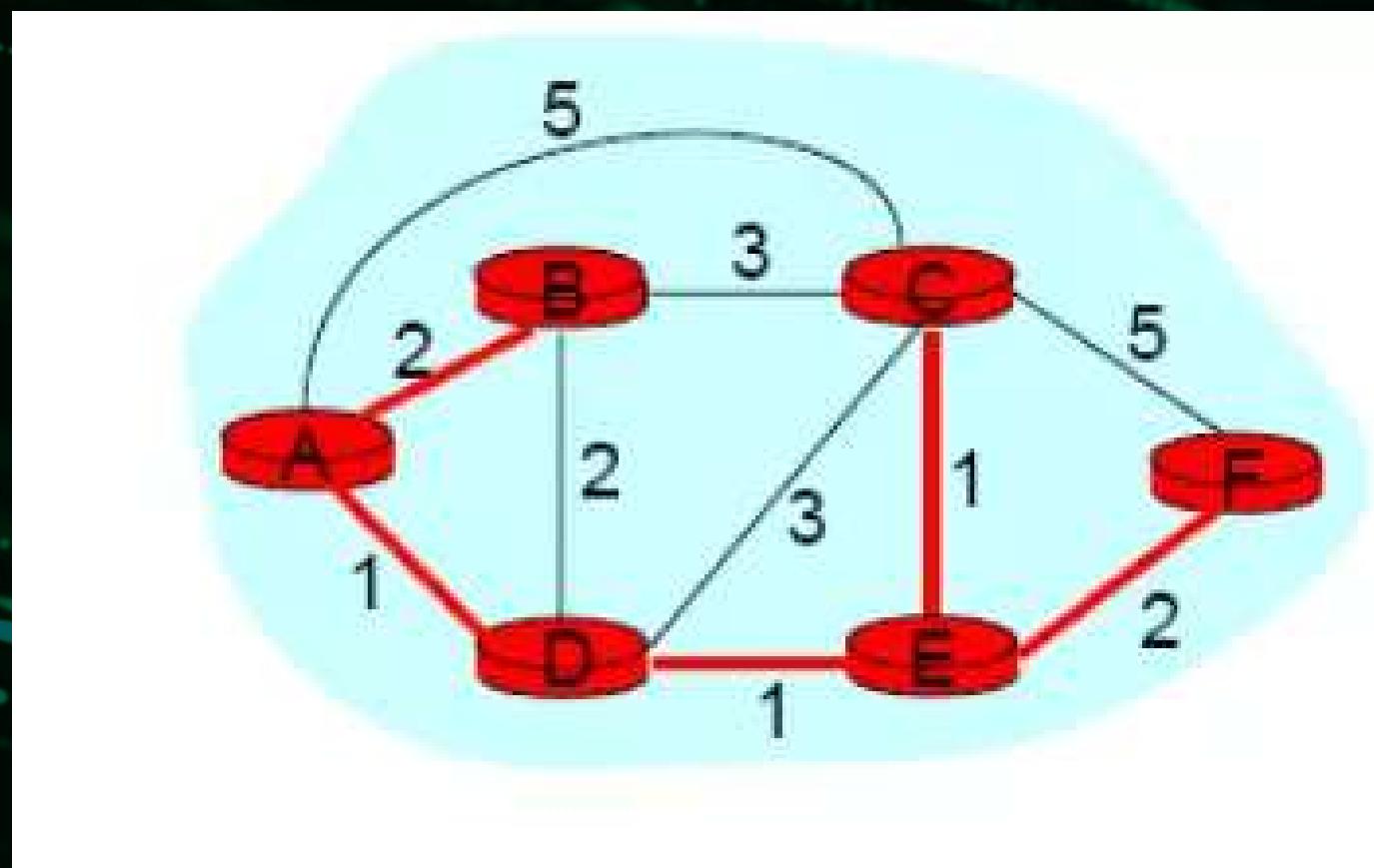
Các trường hợp sử dụng Quick Sort:

- Khi làm việc với mảng lớn, cần tốc độ sắp xếp cao, như trong các ứng dụng thực tế và hệ thống cơ sở dữ liệu.
- Khi có đủ bộ nhớ để quản lý các mảng con trong quy trình đệ quy.
- Trong các tình huống mà chọn được pivot tốt (như chọn pivot ngẫu nhiên) để tránh trường hợp xấu nhất.

GIỚI THIỆU VỀ THUẬT TOÁN TÌM ĐƯỜNG ĐI NGẮN NHẤT

Tầm quan trọng của thuật toán tìm đường trong mạng và đồ thị:

- Thuật toán tìm đường rất quan trọng trong các hệ thống định tuyến, giúp xác định đường đi tối ưu giữa các điểm, tiết kiệm thời gian và tài nguyên.
- Trong đồ thị, các thuật toán này hỗ trợ quản lý mạng lưới giao thông, đường truyền dữ liệu và tối ưu hóa luồng công việc trong hệ thống.



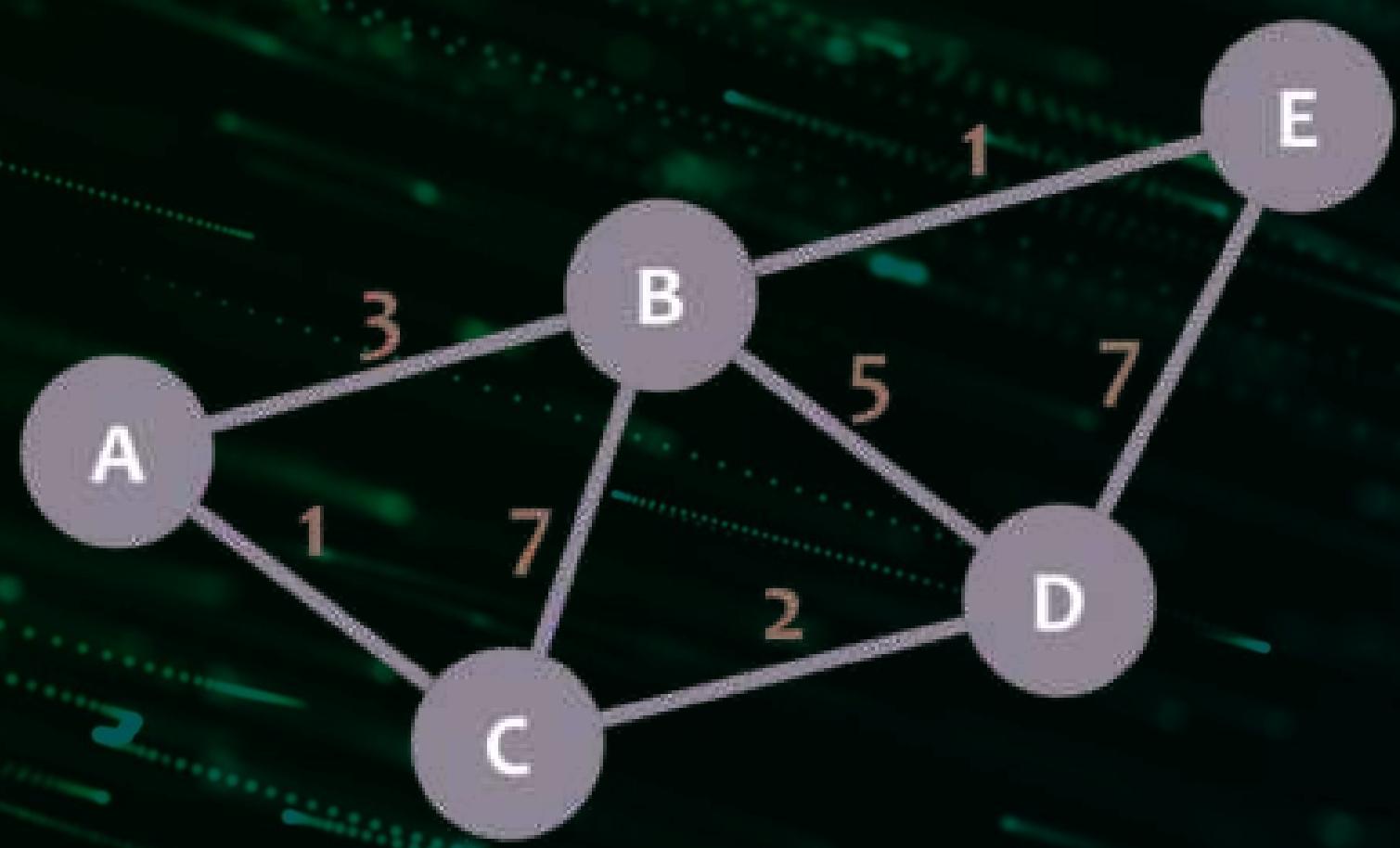
Các ứng dụng của tìm đường đi ngắn nhất:

- Định vị GPS: Tìm đường đi ngắn nhất giữa các địa điểm trong bản đồ số.
- Mạng máy tính: Tối ưu hóa đường truyền dữ liệu trong mạng Internet.
- Quản lý vận tải và logistics: Xác định tuyến đường vận chuyển hiệu quả nhất.
- Trò chơi điện tử: Tìm đường đi cho nhân vật trong môi trường trò chơi.

THUẬT TOÁN DIJKSTRA – KHÁI NIỆM

Định nghĩa và mục tiêu của thuật toán Dijkstra:

- Thuật toán Dijkstra là một thuật toán tìm đường đi ngắn nhất từ một điểm xuất phát đến các đỉnh còn lại trong đồ thị có trọng số dương.
- Mục tiêu là xác định lộ trình ngắn nhất và chi phí thấp nhất, tối ưu hóa thời gian và tài nguyên.



Các tình huống phù hợp cho thuật toán Dijkstra:

- Khi cần tìm đường ngắn nhất trong hệ thống giao thông hoặc định vị GPS.
- Tối ưu hóa đường truyền dữ liệu trong mạng máy tính với chi phí thấp nhất.
- Xác định tuyến vận tải hoặc đường đi trong các bài toán logistics và quản lý chuỗi cung ứng.
- Các bài toán định tuyến robot hoặc AI trong môi trường có đồ thị kết nối.

CÁC BƯỚC THỰC HIỆN THUẬT TOÁN DIJKSTRA

Các bước trong thuật toán Dijkstra:

1. Khởi tạo: Gán khoảng cách từ điểm xuất phát đến chính nó là 0 và đến các đỉnh khác là vô cùng (∞). Đánh dấu tất cả các đỉnh là chưa được ghé thăm.
2. Chọn đỉnh hiện tại: Chọn đỉnh chưa ghé thăm có khoảng cách nhỏ nhất từ điểm xuất phát, gọi là đỉnh hiện tại.
3. Cập nhật khoảng cách: Với mỗi đỉnh kề của đỉnh hiện tại, tính tổng khoảng cách từ điểm xuất phát đến đỉnh đó qua đỉnh hiện tại. Nếu khoảng cách này nhỏ hơn khoảng cách hiện có, cập nhật khoảng cách mới.
4. Đánh dấu đã ghé thăm: Sau khi cập nhật xong các đỉnh kề, đánh dấu đỉnh hiện tại là đã ghé thăm.
5. Lặp lại: Tiếp tục lặp lại các bước trên cho đến khi tất cả các đỉnh được ghé thăm hoặc khoảng cách ngắn nhất tới đích đã tìm được.
6. Kết quả: Truy xuất khoảng cách ngắn nhất từ điểm xuất phát đến các đỉnh khác trong đồ thị.

DIJKSTRA – VÍ DỤ MINH HỌA

Giả sử có đồ thị với các đỉnh A, B, C, D và các cạnh có trọng số:

A đến B = 4, A đến C = 2

B đến C = 1, B đến D = 5

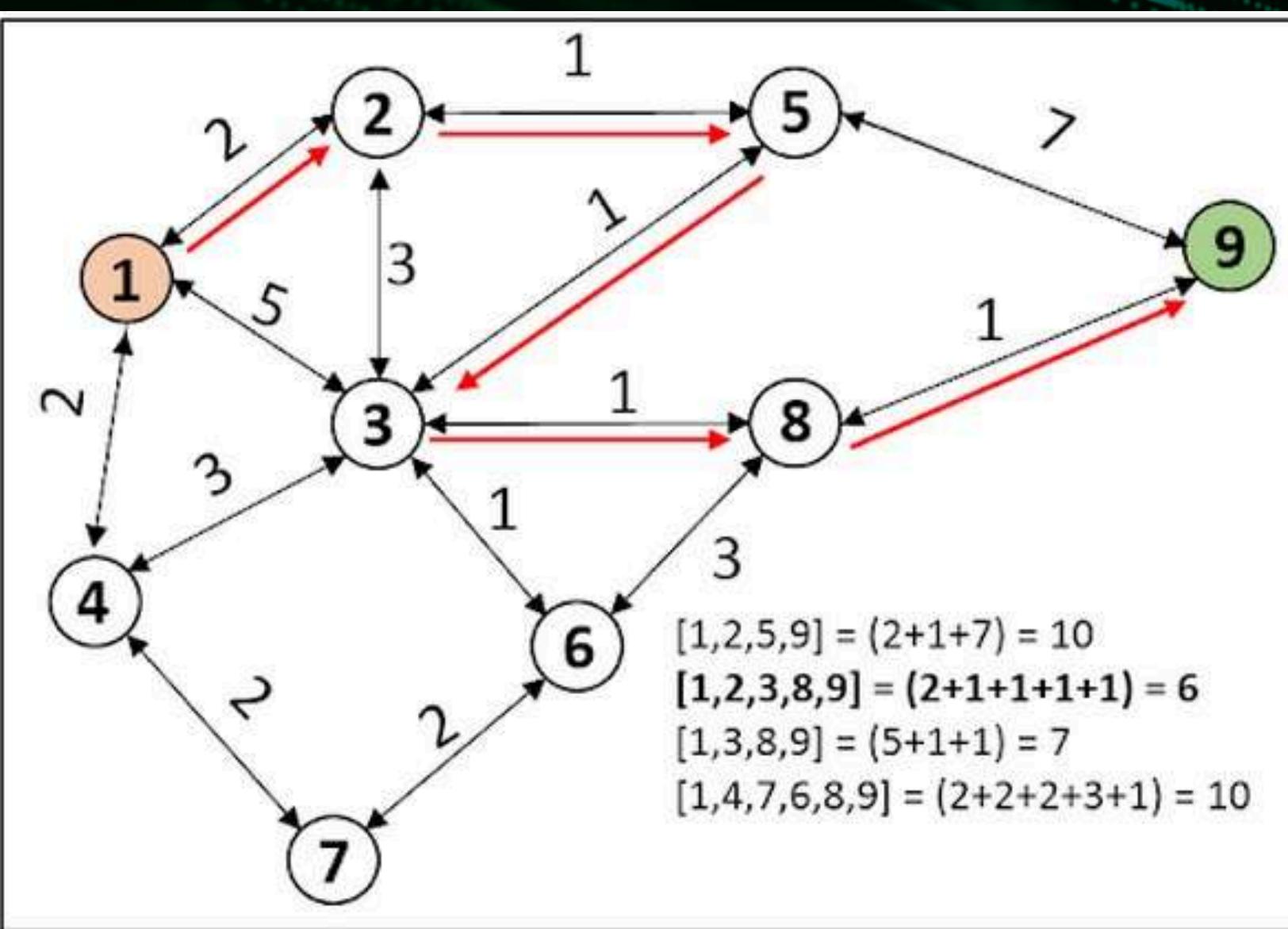
C đến D = 8, C đến B = 1

Bước thực hiện:

1. Khởi tạo: Gán khoảng cách từ A đến chính nó là 0; các đỉnh khác là ∞ .
 2. Chọn đỉnh A: Cập nhật khoảng cách từ A đến B là 4 và từ A đến C là 2.
 3. Chọn đỉnh C (có khoảng cách ngắn nhất): Cập nhật khoảng cách từ C đến B là 3 và từ C đến D là 10.
 4. Chọn đỉnh B: Cập nhật khoảng cách từ B đến D là 8.
 5. Chọn đỉnh D: Hoàn tất, khoảng cách ngắn nhất từ A đến D là 8.
- Kết quả: Khoảng cách ngắn nhất từ A đến mỗi đỉnh là A (0), B (3), C (2), D (8).

ĐỘ PHỨC TẠP CỦA DIJKSTRA

Độ phức tạp thời gian của các thuật toán đồ thị thường phụ thuộc vào cấu trúc dữ liệu được sử dụng. Đối với thuật toán Dijkstra hoặc Prim, nếu sử dụng ma trận kề, độ phức tạp là $O(V^2)$, trong khi với danh sách kề kết hợp với heap (như trong thư viện STL của C++), nó sẽ giảm xuống $O(E \log V)$.



Kích thước đồ thị, tức là số lượng đỉnh (V) và cạnh (E), ảnh hưởng lớn đến hiệu suất. Đồ thị dày đặc (nhiều cạnh) có thể làm tăng chi phí tính toán, trong khi đồ thị thưa (ít cạnh) thường có thể được xử lý nhanh hơn.Thêm vào đó, khi kích thước đồ thị tăng, bộ nhớ và thời gian xử lý cũng sẽ tăng, điều này có thể dẫn đến sự chậm trễ trong ứng dụng thực tế nếu không tối ưu hóa đúng cách.

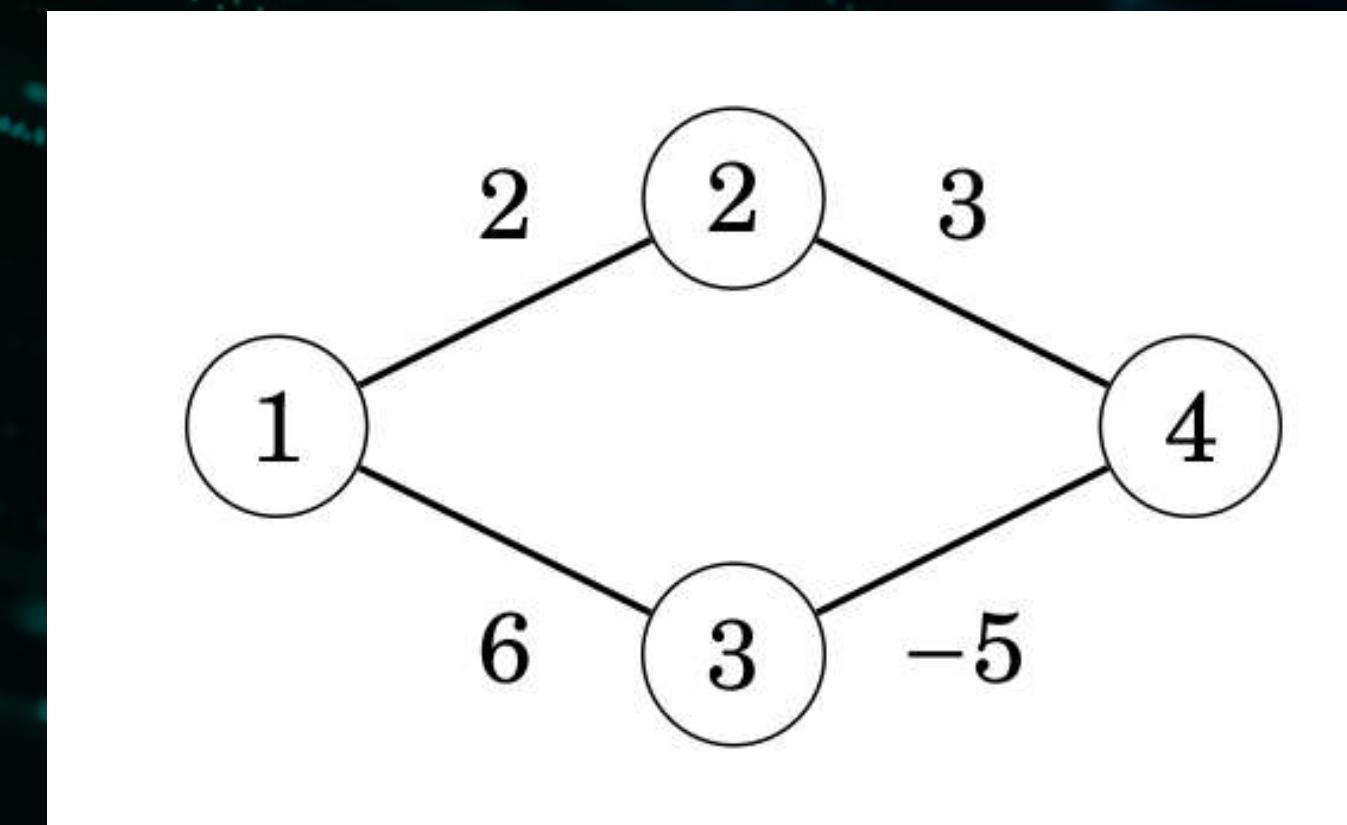
THUẬT TOÁN BELLMAN-FORD – GIỚI THIỆU

Định nghĩa và mục đích của thuật toán Bellman-Ford:

Thuật toán Bellman-Ford là một thuật toán tìm đường đi ngắn nhất từ một đỉnh nguồn đến tất cả các đỉnh khác trong đồ thị có thể có cạnh với trọng số âm. Mục đích của thuật toán này là xác định đường đi ngắn nhất trong những tình huống mà thuật toán Dijkstra không thể áp dụng, đặc biệt là khi có cạnh trọng số âm và cần kiểm tra chu trình âm trong đồ thị.

Khi nào sử dụng Bellman-Ford thay vì Dijkstra:

Bạn nên sử dụng Bellman-Ford khi đồ thị có trọng số âm, hoặc khi cần phát hiện chu trình âm. Bellman-Ford cũng là lựa chọn tốt hơn khi không có yêu cầu về hiệu suất tối ưu, vì nó xử lý mọi loại trọng số mà không gặp vấn đề như Dijkstra. Dijkstra chỉ có thể xử lý trọng số không âm, vì vậy nếu đồ thị chứa cạnh âm, Bellman-Ford sẽ là lựa chọn an toàn và chính xác hơn.



BELLMAN-FORD - CÁCH THỨC HOẠT ĐỘNG

Các bước thực hiện thuật toán Bellman-Ford để tìm đường đi ngắn nhất từ một đỉnh nguồn đến tất cả các đỉnh khác trong đồ thị như sau:

1. Khởi tạo: Đặt khoảng cách từ đỉnh nguồn đến chính nó là 0 và khoảng cách đến tất cả các đỉnh khác là vô cùng (∞).
2. Cập nhật khoảng cách: Lặp lại $V-1$ lần (V là số đỉnh trong đồ thị):
 - Duyệt qua tất cả các cạnh (u, v) trong đồ thị:
 - Nếu khoảng cách từ nguồn đến u cộng với trọng số cạnh (u, v) nhỏ hơn khoảng cách hiện tại đến v , cập nhật khoảng cách đến v .
3. Kiểm tra chu trình âm: Lặp lại một lần nữa qua tất cả các cạnh:
 - Nếu có bất kỳ khoảng cách nào có thể được cập nhật, điều đó có nghĩa là đồ thị chứa chu trình âm.
4. Kết quả: Nếu không có chu trình âm, khoảng cách đến tất cả các đỉnh là đường đi ngắn nhất từ đỉnh nguồn.

SO SÁNH DIJKSTRA VÀ BELLMAN–FORD

Độ phức tạp và ứng dụng:

- Bellman–Ford: Có độ phức tạp $O(VE)$, phù hợp cho đồ thị có cạnh trọng số âm. Ứng dụng trong các bài toán như phân tích mạng, định tuyến trong mạng máy tính, và kiểm tra chu trình âm.
- Dijkstra: Độ phức tạp $O(E \log V)$ với danh sách kề và heap. Thích hợp cho đồ thị có trọng số không âm, thường được sử dụng trong các bài toán tìm đường đi ngắn nhất trong giao thông, bản đồ, và hệ thống định vị.

Bellman–Ford:

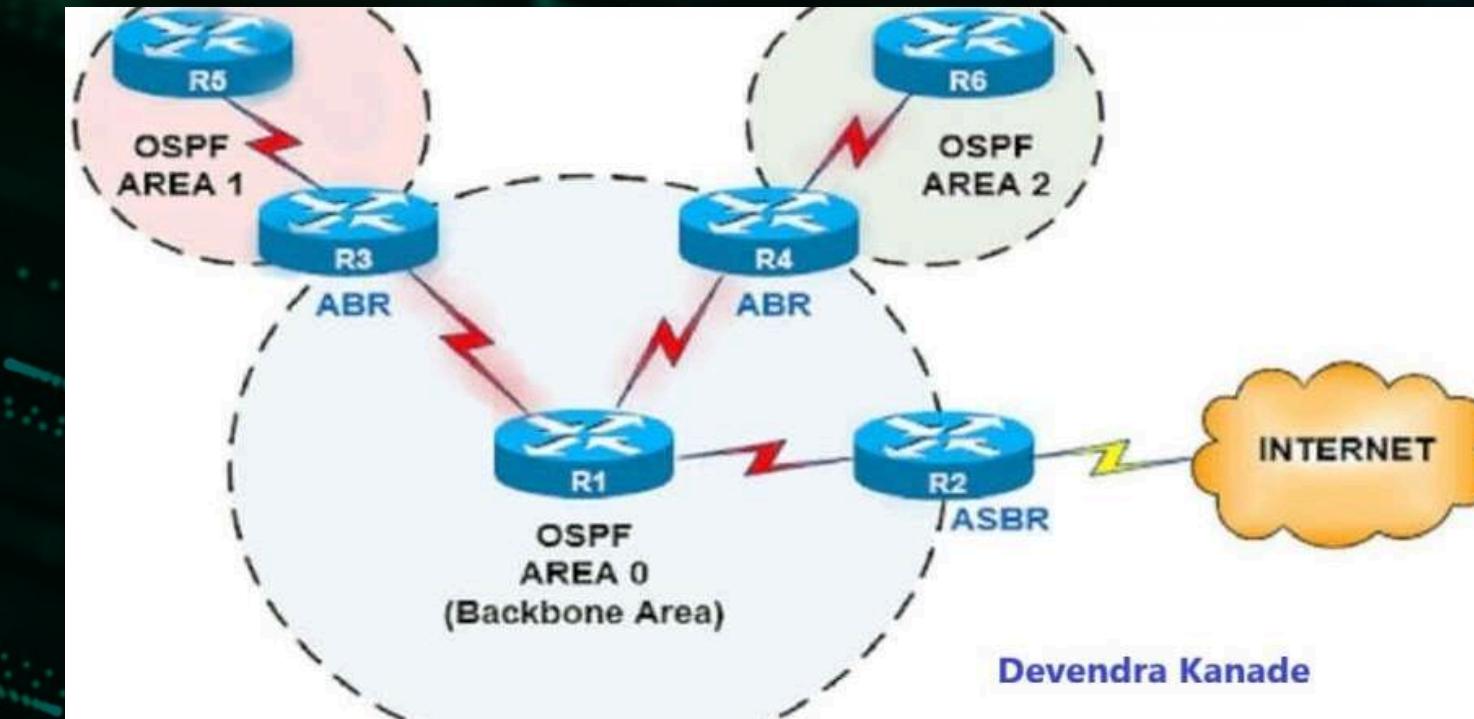
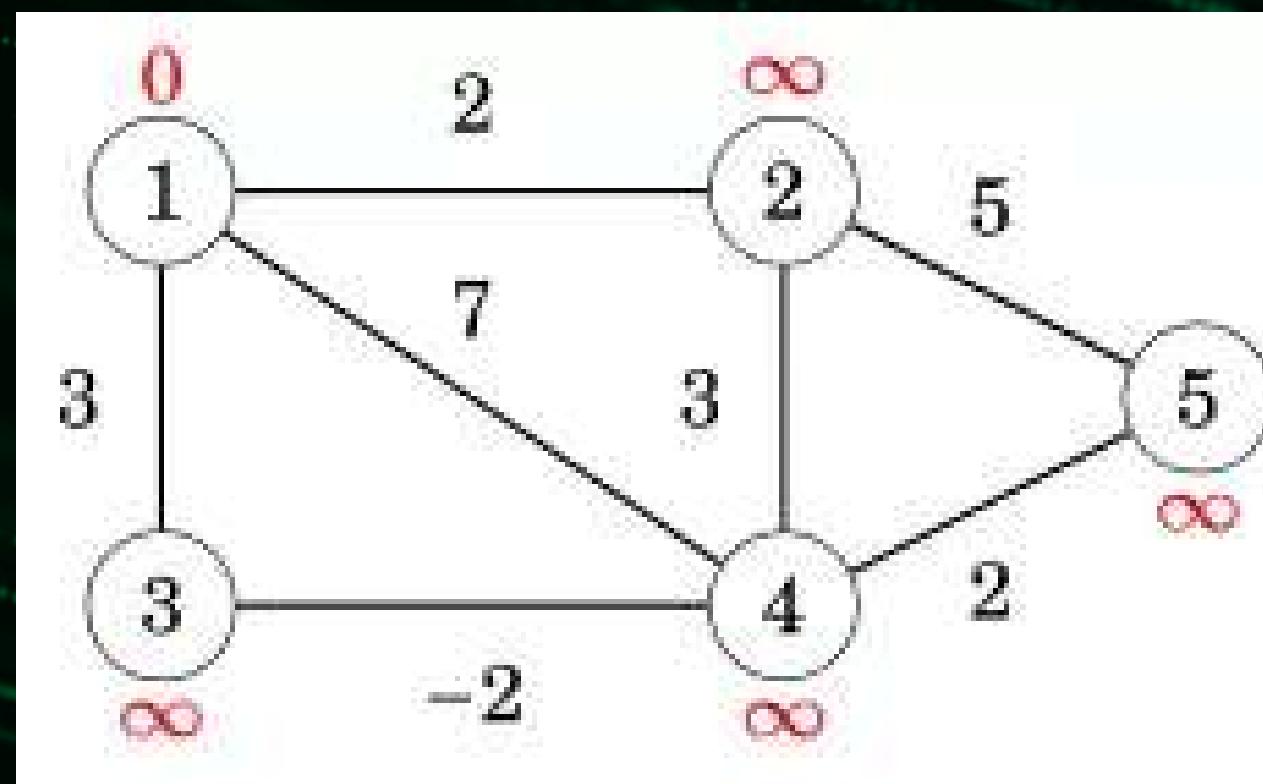
- **Ưu điểm:** Xử lý được trọng số âm, phát hiện chu trình âm.
- **Hạn chế:** Chậm hơn, đặc biệt với đồ thị lớn do độ phức tạp cao.

Dijkstra:

- **Ưu điểm:** Nhanh hơn trong trường hợp trọng số không âm, hiệu quả với đồ thị dày đặc.
- **Hạn chế:** Không xử lý được trọng số âm, có thể dẫn đến kết quả sai nếu có cạnh âm.

ỨNG DỤNG THỰC TẾ CỦA DIJKSTRA VÀ BELLMAN-FORD

Sử dụng Dijkstra trong mạng giao thông và mạng máy tính: Thuật toán Dijkstra rất hiệu quả trong mạng giao thông để tìm đường đi ngắn nhất giữa các điểm, chẳng hạn như từ một vị trí đến đích trong hệ thống định vị GPS. Nó giúp xác định lộ trình tối ưu dựa trên khoảng cách hoặc thời gian di chuyển giữa các giao lộ. Trong mạng máy tính, Dijkstra được sử dụng để tìm đường đi tối ưu giữa các nút mạng, tối ưu hóa băng thông và giảm thiểu độ trễ trong việc truyền dữ liệu.



Devendra Kanade

Bellman-Ford trong các đồ thị có cạnh trọng số âm: Thuật toán Bellman-Ford được áp dụng trong các tình huống mà đồ thị có trọng số âm, như trong các bài toán tài chính, nơi các chu trình có thể đại diện cho lãi suất hoặc chi phí tiêu cực. Nó cũng được sử dụng trong các ứng dụng phân tích mạng để phát hiện chu trình âm, giúp nhận diện các vấn đề tiềm ẩn trong hệ thống. Bellman-Ford cung cấp một phương pháp đáng tin cậy để đảm bảo rằng các đường đi ngắn nhất được xác định, ngay cả trong những điều kiện không thuận lợi.

THANK YOU