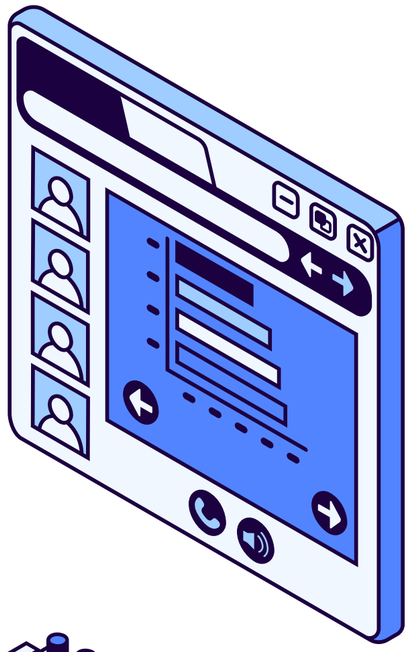
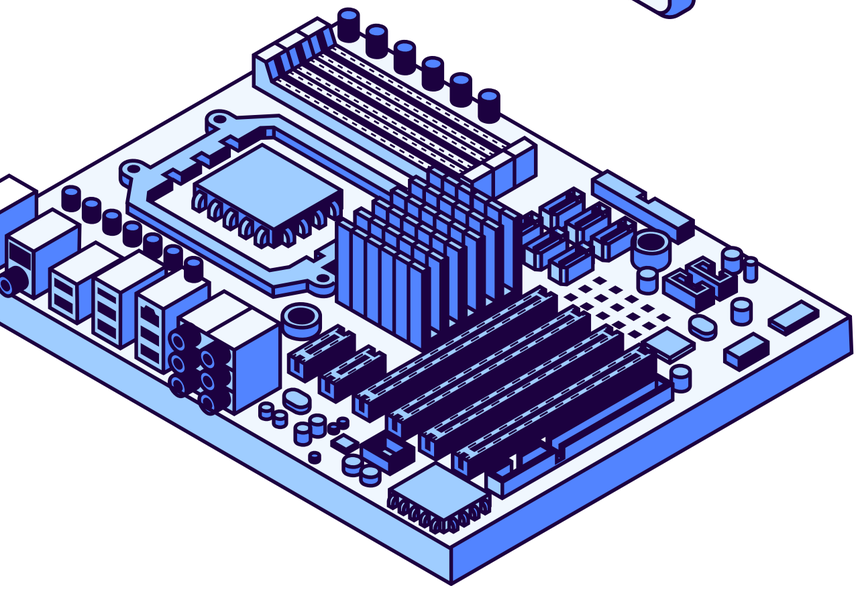




Ngăn xếp

NHÓM 9

Giáo viên HD: Th.S Vũ Đình Bảo

Sinh viên:

Nguyễn Hùng Dũng	22134002
Trần Như Hoàng	22134006
Trần Nguyên Phương Bình	24133006
Phạm Ngọc Phúc	22134010
Võ Hồng Quân	22134012

Nội dung

01 Giới thiệu về Stack

PART

02 Cơ sở lý thuyết

PART

- Các khái niệm cơ bản về Stack
- Các phương pháp cài đặt
- Phân tích độ phức tạp
- Ứng dụng thực tiễn

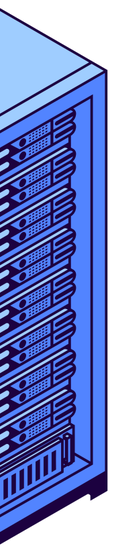
03 Kết luận

PART

- Kết quả
- Xu hướng tương lai

GIỚI THIỆU

1.1 Bối cảnh chủ đề



Lập trình

Là lĩnh vực then chốt giúp ích cho nhiều công việc khác

Cấu trúc dữ liệu và giải thuật

Là "xương sống" và "nền móng" để xây dựng phần mềm phức tạp, hiệu quả

Abstract data type

Bao gồm: List, Stack, Queue, Deque, Priority Queue, Set, Map/Dictionary, Disjoint Set.

Stack

Giữ vai trò cốt lõi trong thiết kế thuật toán và hệ thống hiện đại

1.2,1.3 Mục tiêu và giới hạn nghiên cứu

Làm Rõ Kiến Thức Nền Tảng: Stack (LIFO, thao tác cơ bản).

Phân Tích Hiệu Suất & Cài Đặt: mảng & danh sách liên kết, cài đặt mã nguồn bằng nhiều ngôn ngữ lập trình.

Minh Họa Ứng Dụng Thực Tiễn: Call Stack, biểu thức, DFS, Undo/Redo

Giới hạn: không đề cập đến CTDL khác (trừ Queue cơ bản), không nghiên cứu sâu về stack trên phần cứng

1.4 Phương Pháp Nghiên Cứu

Phương pháp nghiên cứu tài liệu

Phương pháp phân tích & tổng hợp lý thuyết

Phương pháp thực nghiệm

Phương pháp làm việc nhóm

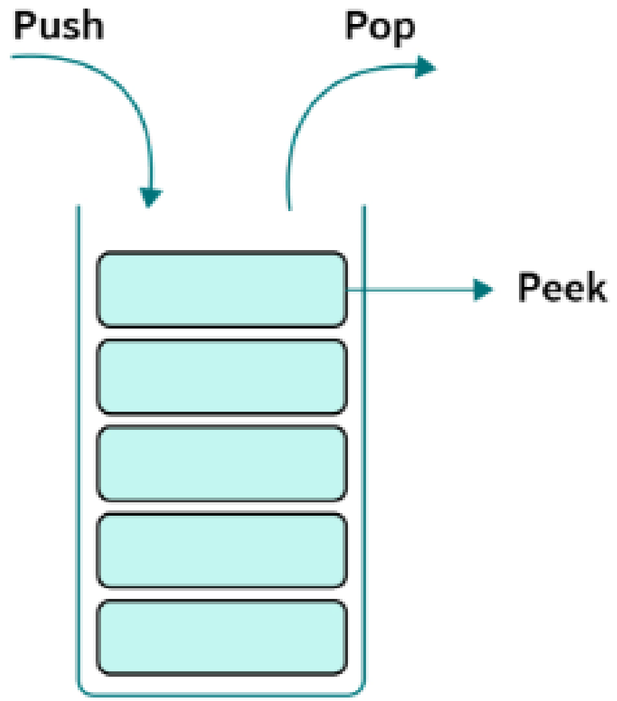
Cơ sở lý thuyết

2.1.1 Định nghĩa và các đặc trưng cơ bản của ngăn xếp (Stack)

1. Ngăn xếp (Stack) là cấu trúc dữ liệu theo nguyên tắc LIFO (Last In, First Out): vào sau, ra trước.

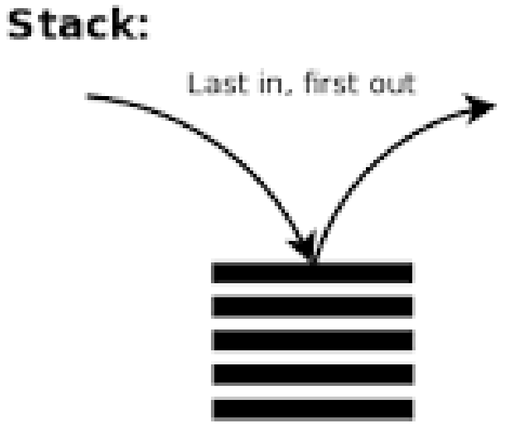
2. Các thao tác chính: push, pop, peek...

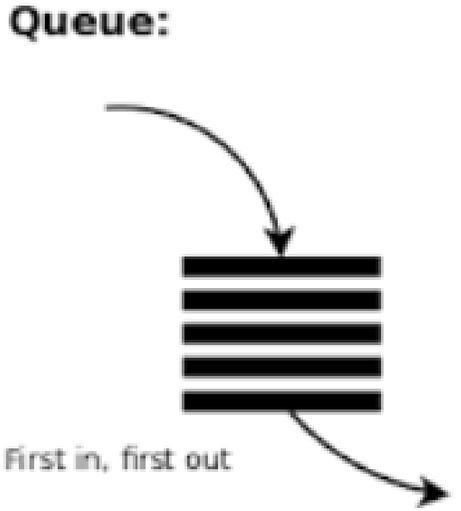
3. Phức tạp thời gian: O(1) cho các thao tác trên.

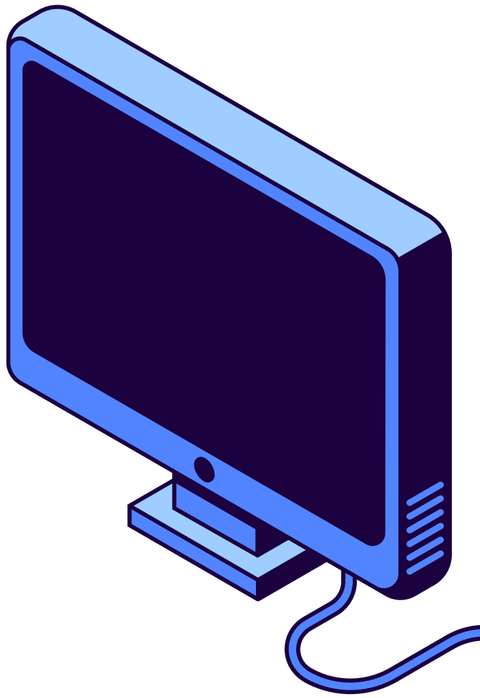


2.1.2 So sánh với hàng đợi (Queue)

Đặc điểm	Ngăn xếp	Hàng đợi
Nguyên tắc hoạt động	LIFO (Last In, First Out)	FIFO (First In, First Out)
Thao tác	push, pop, peek, isEmpty	enqueue, dequeue, front, isEmpty
Con trỏ quản lý	Một con trỏ top(đỉnh)	Hai con trỏ:front(đầu),rear(cuối)
Ứng dụng	Gọi hàm đệ quy, quay lui, undo/redo, stack frame	Lập lịch CPU, BFS, buffering I/O, producer-consumer







2.1.3 Các thao tác cơ bản với Stack

Push (Đẩy vào)

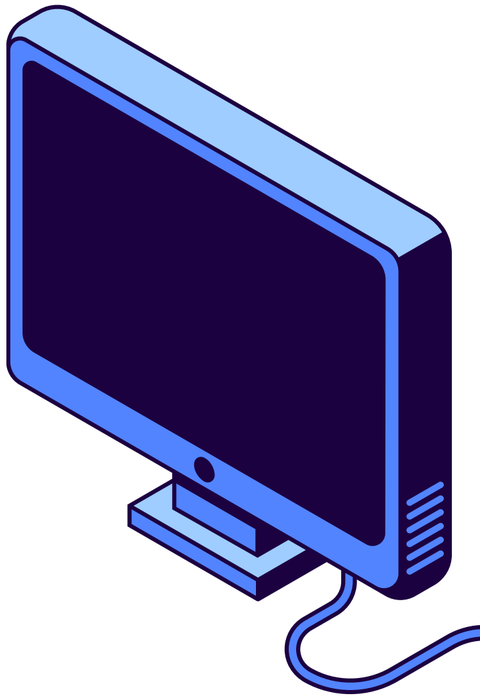
- **Mục đích:** Thêm một phần tử mới vào đỉnh Stack.
- **Hành động:** Kích thước Stack tăng lên 1.
- **Lưu ý:** Có thể gây lỗi Stack Overflow nếu Stack dùng mảng cố định và đã đầy.

Ví dụ:

Trước: [A, B] (Top = B)

Push(C)

Sau: [A, B, C] (Top = C)



2.1.3 Các thao tác cơ bản với Stack

Pop (Lấy ra)

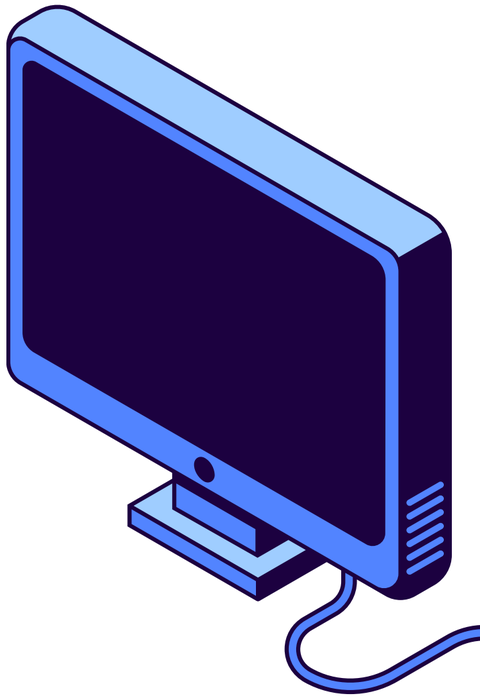
- **Mục đích:** Lấy và xóa phần tử ở đỉnh Stack.
- **Hành động:** Kích thước Stack giảm đi 1.
- **Lưu ý:** Có thể gây lỗi Stack Underflow nếu Stack rỗng.

Ví dụ:

Trước: [A, B, C] (Top = C)

Pop() → Trả về C

Sau: [A, B] (Top = B)



2.1.3 Các thao tác cơ bản với Stack

Peek / Top (Xem đỉnh)

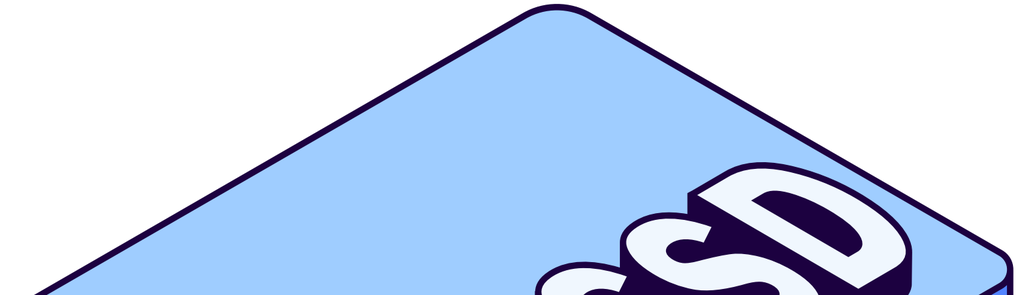
- **Mục đích:** Xem giá trị phần tử ở đỉnh mà không xóa nó.
- **Hành động:** Không làm thay đổi Stack.

Ví dụ:

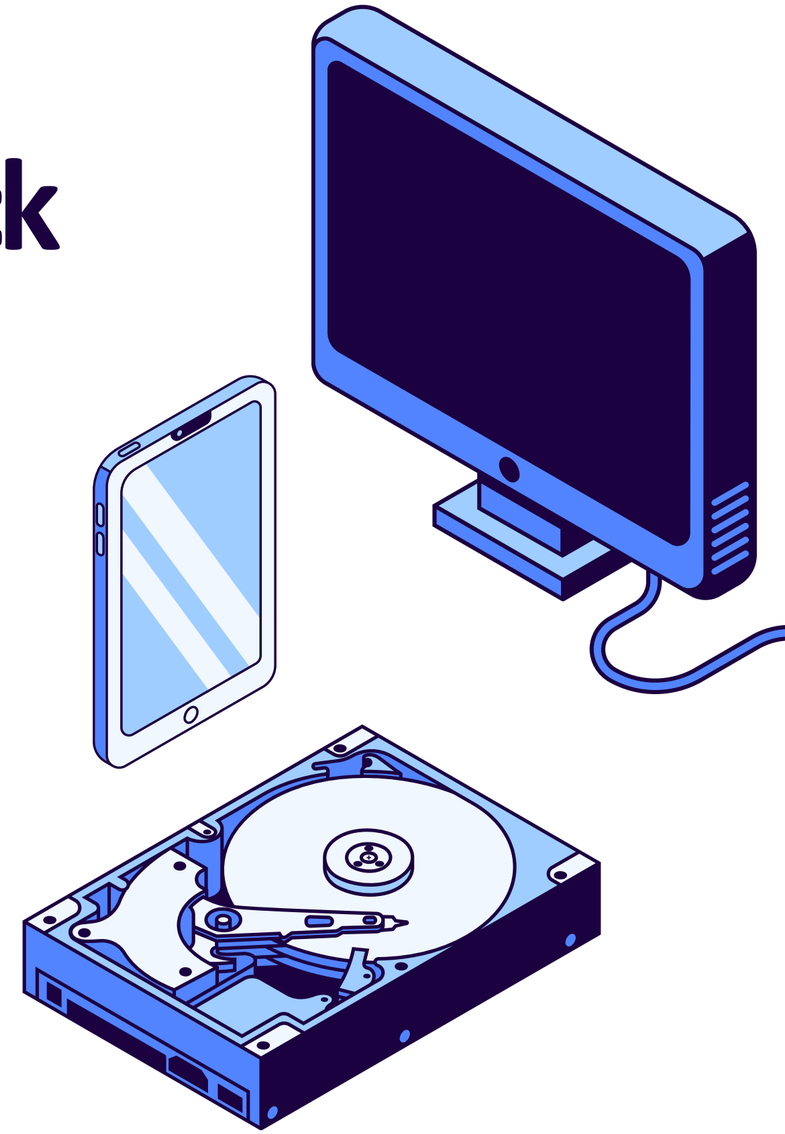
Stack: [A, B, C] (Top = C)

Peek() → Trả về C

Stack vẫn là: [A, B, C]



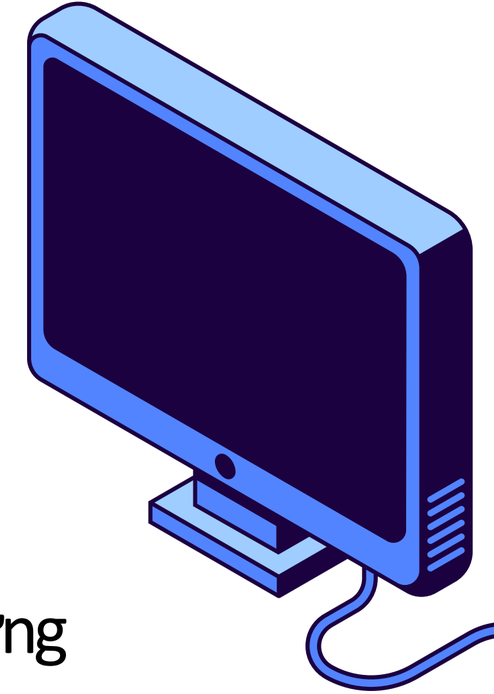
2.1.3 Các thao tác cơ bản với Stack



isEmpty (Kiểm tra rỗng)

- **Mục đích:** Kiểm tra Stack có không có phần tử nào không.
- **Ứng dụng:** Bắt buộc kiểm tra trước khi Pop() hoặc Peek() để tránh lỗi.

2.1.3 Các thao tác cơ bản với Stack

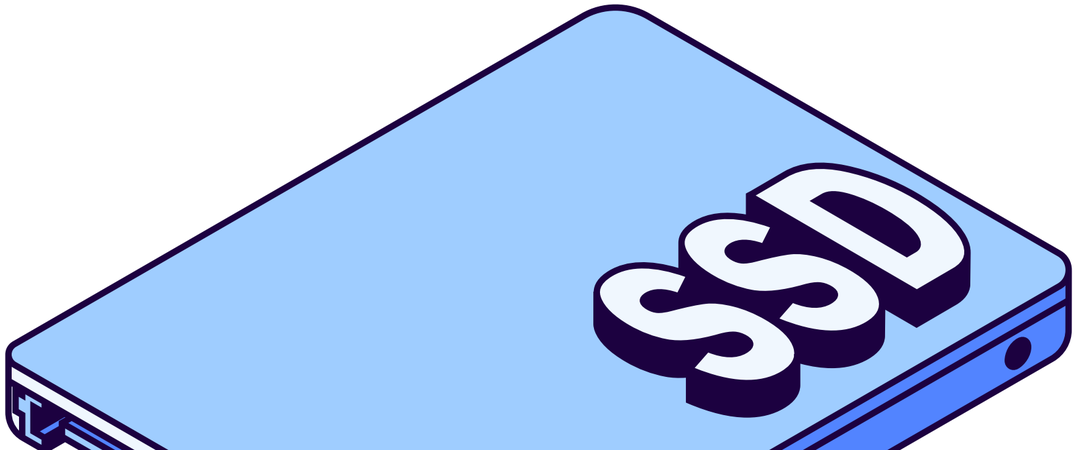


isFull (Kiểm tra đầy)

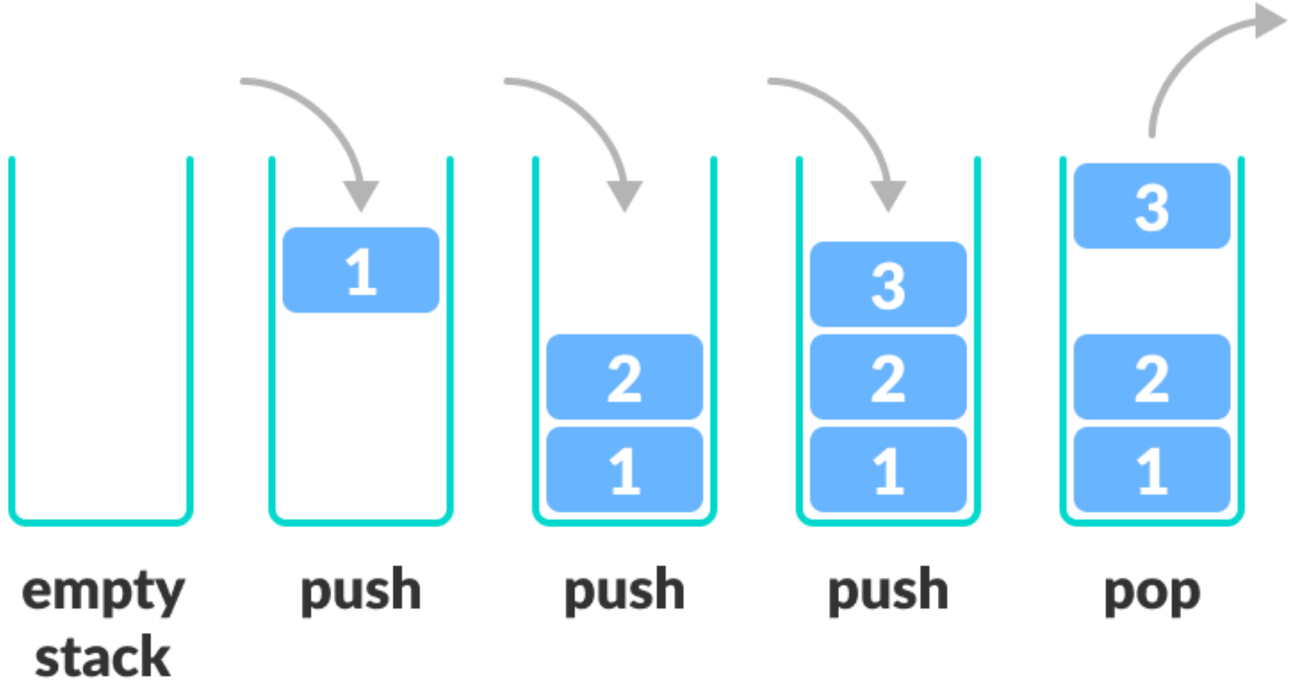
- **Mục đích:** Kiểm tra Stack (dùng mảng cố định) đã hết chỗ trống chưa.
- **Ứng dụng:** Kiểm tra trước khi Push() để tránh tràn.

size (Kích thước)

- **Mục đích:** Trả về số lượng phần tử hiện có trong Stack.



2.1.3 Các thao tác cơ bản với Stack



2.1.4 Các phương pháp cài đặt Stack

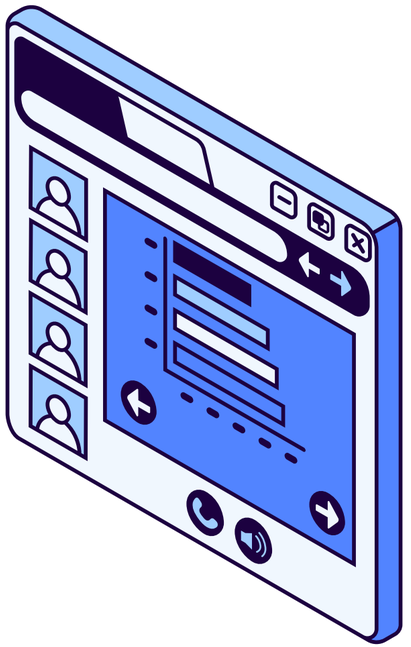
Cài đặt bằng mảng (Array-based)

Nguyên lý: Dùng mảng 1 chiều + biến top quản lý đỉnh.

- Push : Gán giá trị mới → tăng top.
- Pop : Lấy array[top] → giảm top.
- Peek : Xem phần tử array[top].

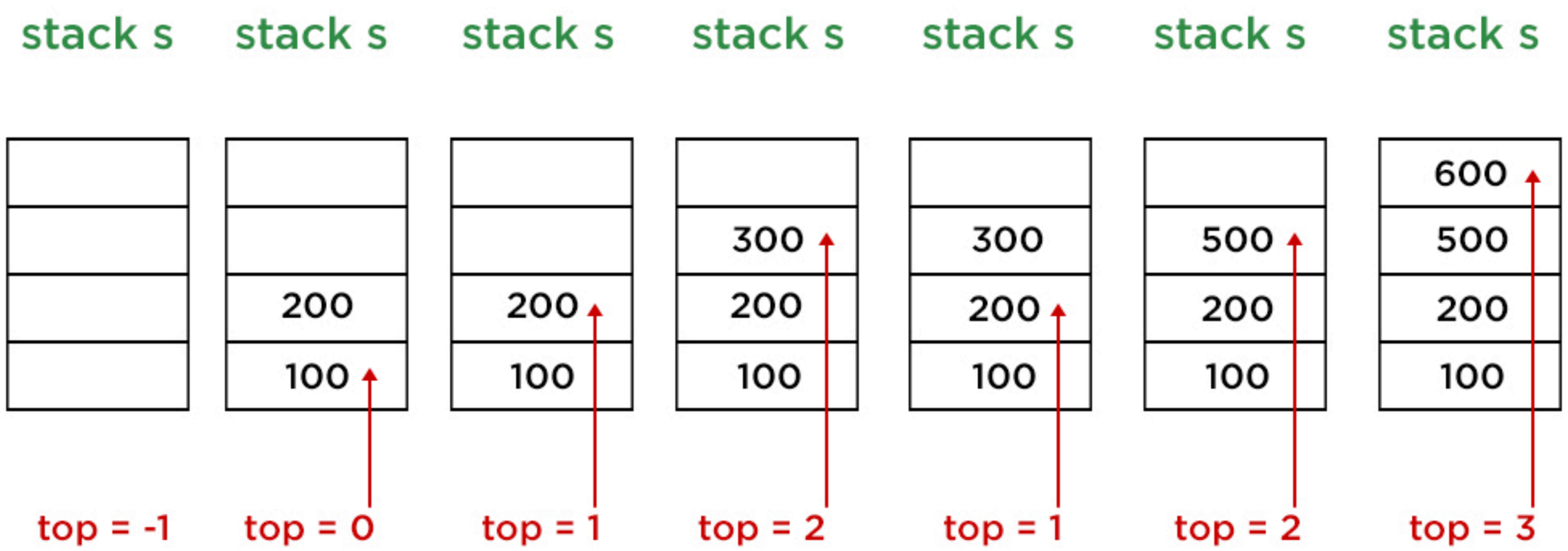
Ưu điểm : Đơn giản, nhanh.

Nhược điểm : Kích thước cố định, dễ tràn.



2.1.4 Các phương pháp cài đặt Stack

Cài đặt bằng mảng (Array-based)



2.1.4 Các phương pháp cài đặt Stack

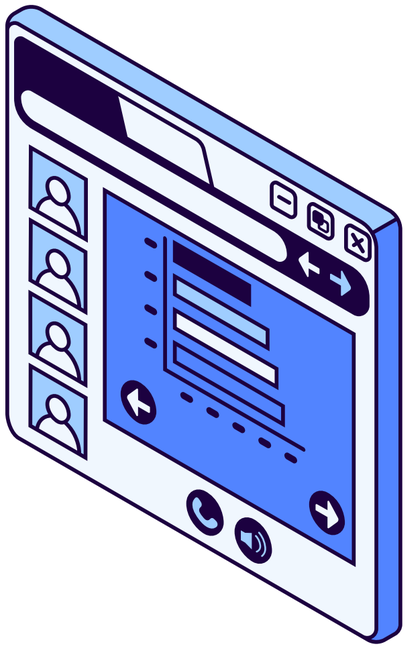
Cài đặt bằng danh sách liên kết (Linked List-based)

Nguyên lý: Mỗi phần tử là node gồm: dữ liệu + con trỏ next (top=null).

- Push: Tạo node mới → trỏ đến top → cập nhật top.
- Pop: Cập nhật top = top → next.
- Peek: Trả về dữ liệu top.

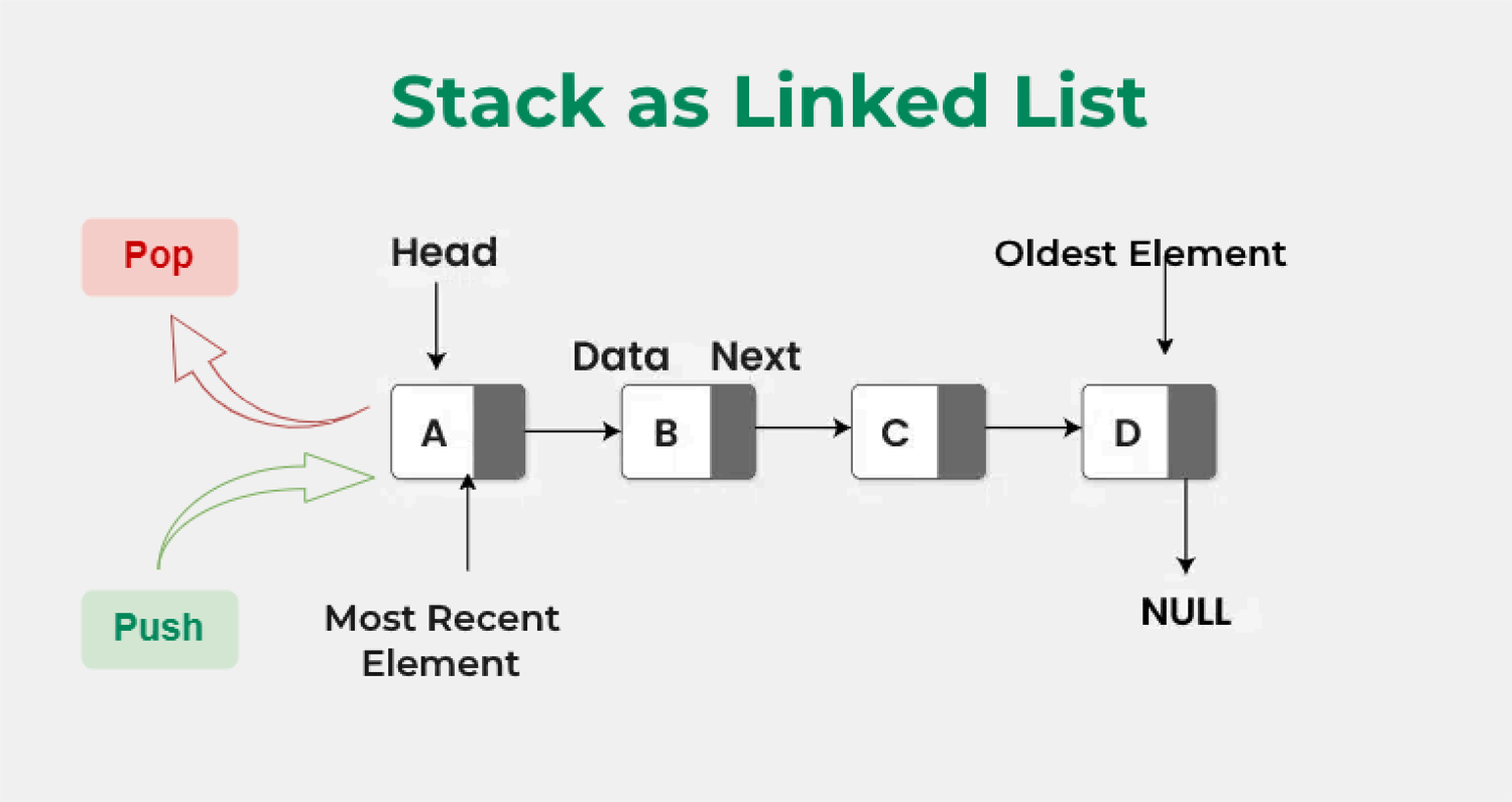
Ưu điểm : Linh hoạt, kích thước động.

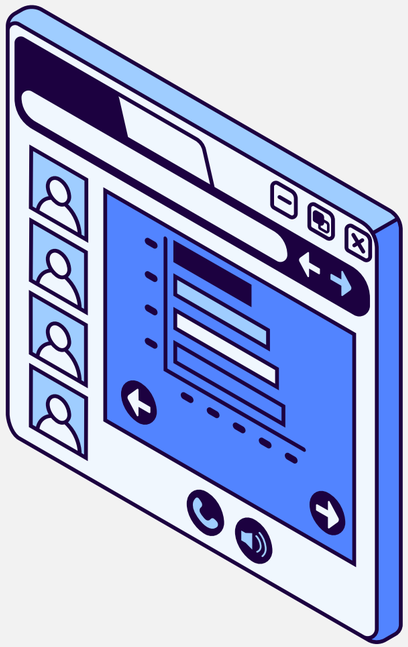
Nhược điểm : Tốn thêm bộ nhớ con trỏ, locality kém.



2.1.4 Các phương pháp cài đặt Stack

Cài đặt bằng danh sách liên kết (Linked List-based)





2.1.4.3. So sánh hai phương pháp cài đặt Stack

Tiêu chí	Array-based	Linked List-based
Ưu điểm	Cài đặt đơn giản. Truy cập đỉnh nhanh (cache locality tốt).	Kích thước động, linh hoạt. Không lo bị tràn stack (stackoverflow) trừ khi sử dụng hết bộ nhớ hệ thống.
Nhược điểm	Kích thước cố định, dễ tràn hoặc lãng phí. Resizing tốn kém.	Tốn thêm bộ nhớ cho con trỏ. Locality kém, tốc độ có thể chậm hơn.
Ứng dụng phù hợp	Khi biết trước số phần tử tối đa, cần tốc độ truy cập cao.	Khi số phần tử biến động nhiều và lớn, cần linh hoạt, chấp nhận chi phí bộ nhớ thêm.

2.1.5 Phân tích độ phức tạp

2.1.5.1 Độ phức tạp thời gian

- Các thao tác push, pop, peek, isEmpty đều O(1) (trực tiếp tại đỉnh).
- Mảng động: push thường O(1), nhưng resize → O(n) (trung bình vẫn O(1)).
- Linked List: cũng O(1), nhưng thực tế chậm hơn do overhead bộ nhớ + locality kém.

2.1.5 Phân tích độ phức tạp

Độ phức tạp thời gian

Thao tác	Mảng cố định	Mảng di động	Danh sách liên kết
Push	O(1)	O(1)/O(n)	O(1)
Pop	O(1)	O(1)	O(1)
Peek	O(1)	O(1)	O(1)
isEmpty	O(1)	O(1)	O(1)

2.1.5 Phân tích độ phức tạp

2.1.5.2 Bộ nhớ sử dụng

Đặc điểm	Mảng cố định	Mảng di động	Danh sách liên kết
Cách cấp phát	Cấp phát sẵn mảng có kích thước maxSize	Cấp phát tỷ lệ với số phần tử, có thêm "dư địa"	Mỗi phần tử là một node, gồm dữ liệu + con trỏ next
Độ phức tạp lưu trữ	Θ(maxSize) (chiếm trước toàn bộ bộ nhớ)	Θ(n), có resize khi đầy	Θ(n) dữ liệu + Θ(n) overhead con trỏ
Giới hạn dung lượng	Bị cố định bởi maxSize	Linh hoạt, không cần isFull	Không giới hạn, chỉ dừng khi hết RAM

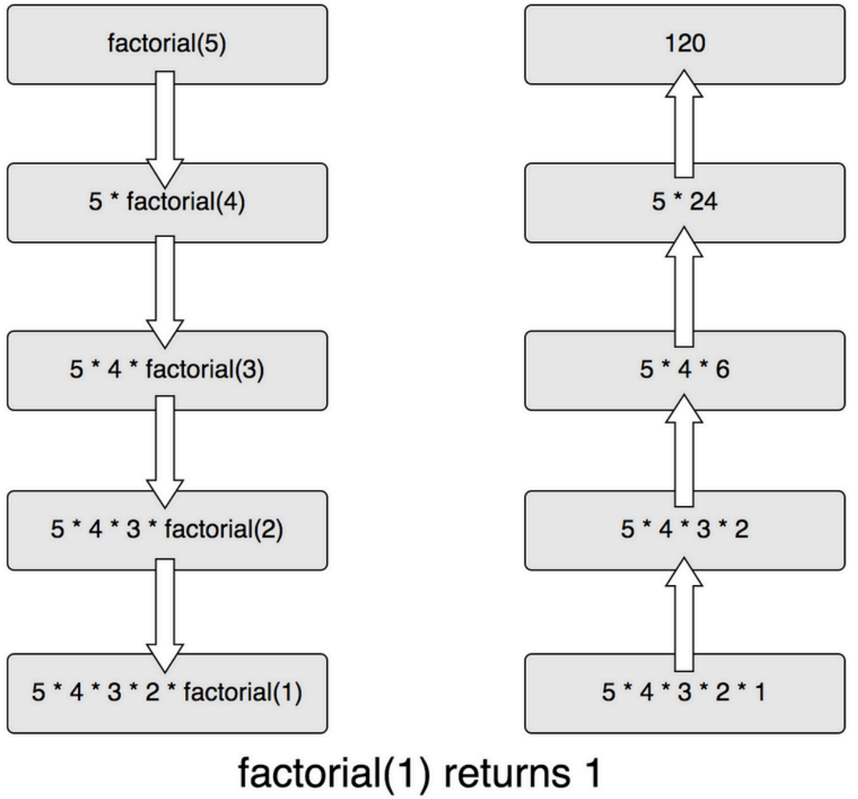
2.1.5 Phân tích độ phức tạp

2.1.5.2 Đánh giá

Phương pháp	Cách làm	Ưu điểm	Nhược điểm
Thực nghiệm	Cài đặt và chạy chương trình, đo thời gian thực thi trên các bộ dữ liệu khác nhau	Kết quả cụ thể, thực tế. Phản ánh thời gian chạy thật	Phụ thuộc phần cứng, ngôn ngữ. Khó chọn dữ liệu đầy đủ Tốn kém thời gian, chi phí
Phân tích tiệm cận	Phân tích lý thuyết, sử dụng ký hiệu Big-O để biểu diễn độ phức tạp khi kích thước đầu vào lớn	Độc lập phần cứng, ngôn ngữ. Dễ so sánh thuật toán. Không cần cài đặt. Tốt cho dữ liệu lớn	Không cho thời gian chính xác. Bỏ qua hằng số. Ít chính xác với dữ liệu nhỏ

2.2.1 Quản lý hàm đệ quy

- Khi gọi hàm → frame mới push vào Call Stack (tham số, biến, địa chỉ trả về).
- Hàm kết thúc → frame pop ra → quay lại vị trí cũ.
- Đệ quy hoạt động nhờ nhiều frame độc lập.
- Đệ quy quá sâu → Stack Overflow.



2.2.2 Chuyển đổi và đánh giá biểu thức

1. Khái niệm

Infix: Toán tử nằm giữa toán hạng.

Ví dụ: A + B * C

Postfix : Toán tử nằm sau toán hạng.

Ví dụ: A B C * +

Prefix: Toán tử nằm trước toán hạng.

Ví dụ: + A * B C

2.2.2 Chuyển đổi và đánh giá biểu thức

2. Vấn đề

Biểu thức Infix phụ thuộc vào độ ưu tiên toán tử và dấu ngoặc.

A + B * C → phải tính B * C trước.

(A + B) * C → phải tính A + B trước.

Máy tính khó hiểu Infix → cần đổi sang Postfix/Prefix để tính toán đơn giản, nhất quán.

2.2.2 Chuyển đổi và đánh giá biểu thức

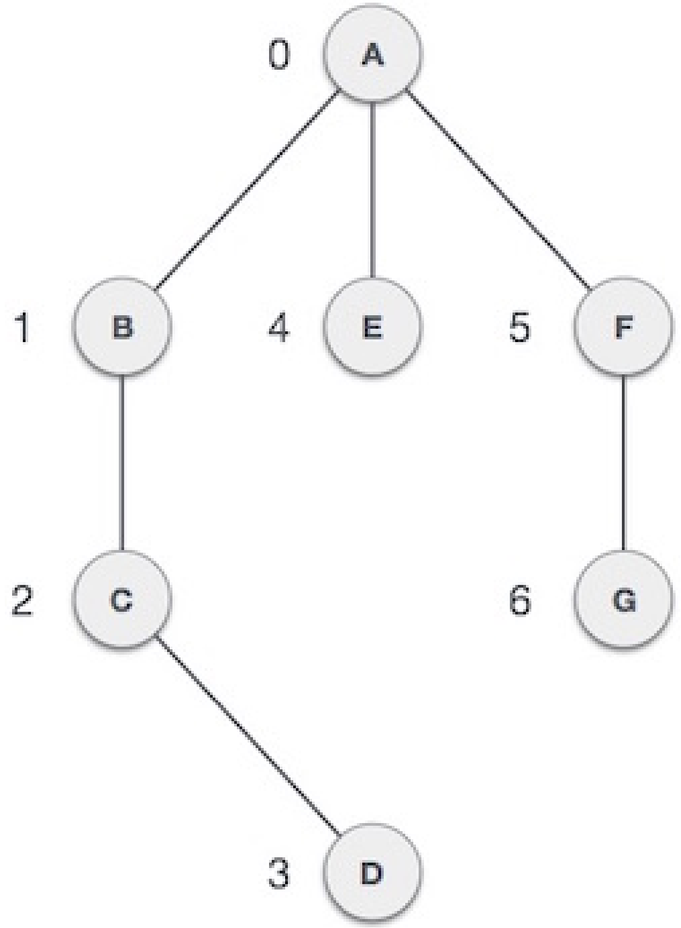
Giải pháp: Dùng thuật toán **Shunting-yard (Edsger Dijkstra)**

- Đọc biểu thức Infix từ trái sang phải.
- Nếu gặp toán hạng → đưa vào kết quả.
- Nếu gặp toán tử:
  - So sánh ưu tiên với toán tử trong Stack
  - Nếu ưu tiên cao hơn → đẩy vào Stack.
- Gặp dấu ngoặc mở → đẩy vào Stack.
- Gặp dấu ngoặc đóng → lấy toán tử ra kết quả cho đến khi gặp ngoặc mở.
- Đọc hết biểu thức → lấy toàn bộ toán tử còn lại trong Stack ra kết quả.

2.2.3 Thuật toán DFS & Backtracking

DFS (Depth-First Search):

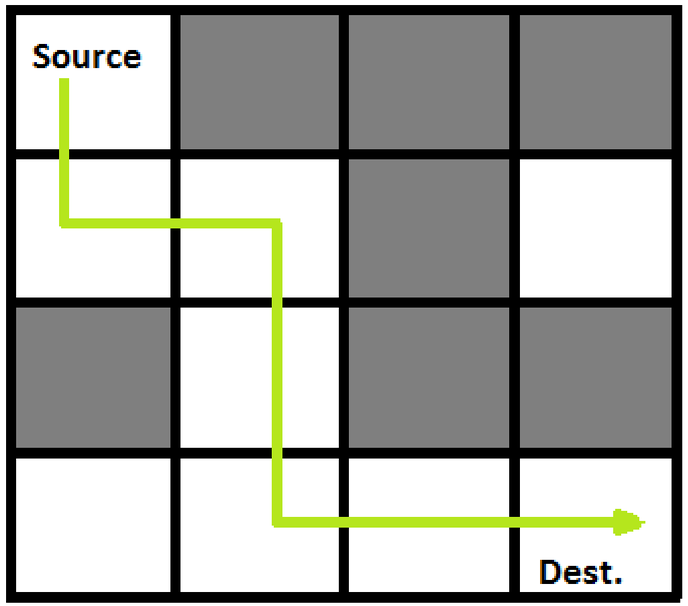
- Duyệt theo chiều sâu, đi sâu nhất rồi quay lui.
- Thực hiện bằng đệ quy hoặc Stack.
- Chủ yếu dùng để duyệt/tìm kiếm trong đồ thị, cây.



2.2.3 Thuật toán DFS & Backtracking

Backtracking:

- Dựa trên DFS nhưng có thêm bước kiểm tra ràng buộc.
- Khi đi sai hướng → quay lui thử nhánh khác.
- Thường dùng cho: Sudoku, mê cung, tổ hợp, hoán vị.

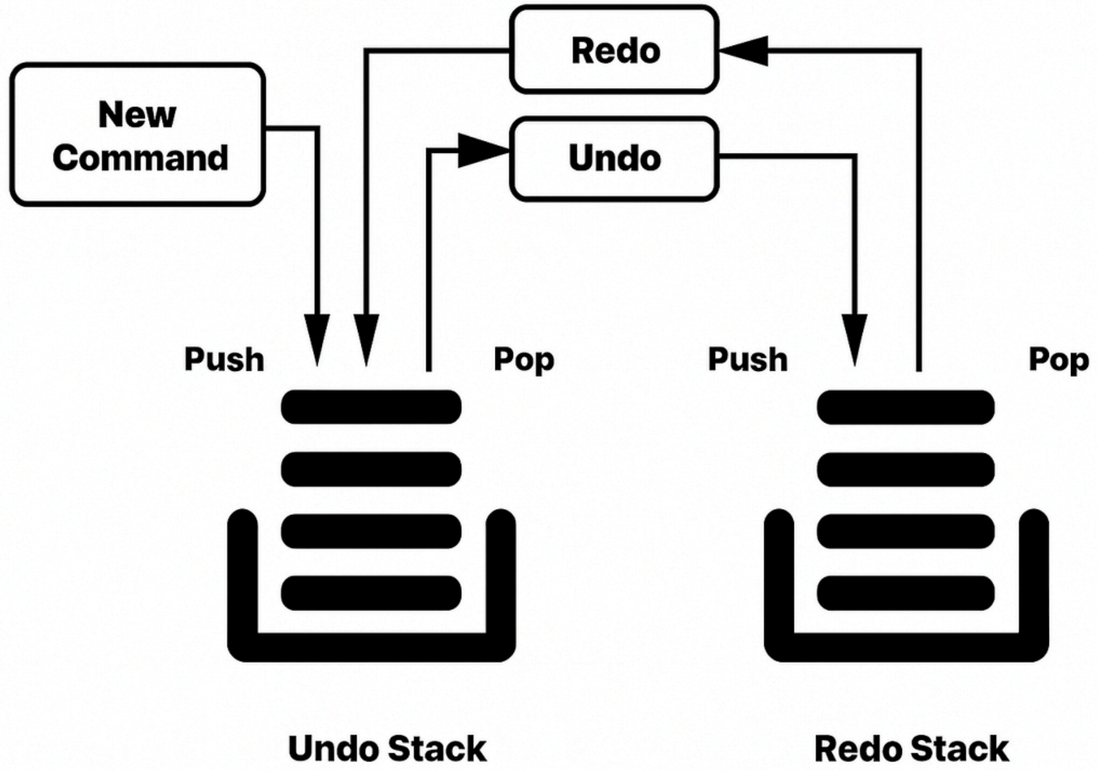


2.2.4 Ứng dụng đời sống

Undo/Redo

Trong soạn thảo văn bản, chỉnh sửa ảnh, IDE… hệ thống thường dùng 2 Stack:

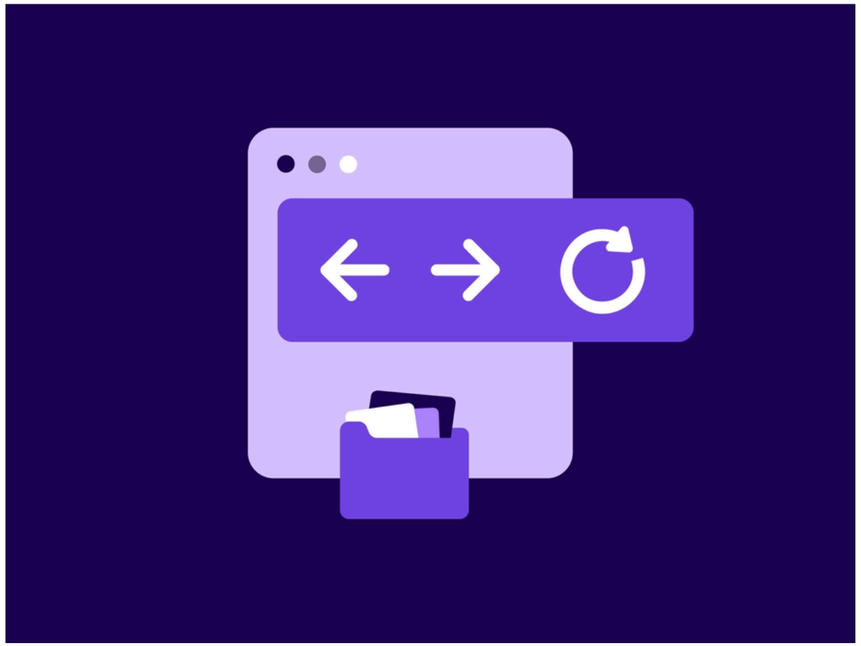
- Undo Stack: lưu các thao tác mới (mỗi hành động mới được push vào đây).
- Redo Stack: lưu các thao tác đã được hoàn tác.



2.2.4 Ứng dụng đời sống

Lịch sử duyệt web

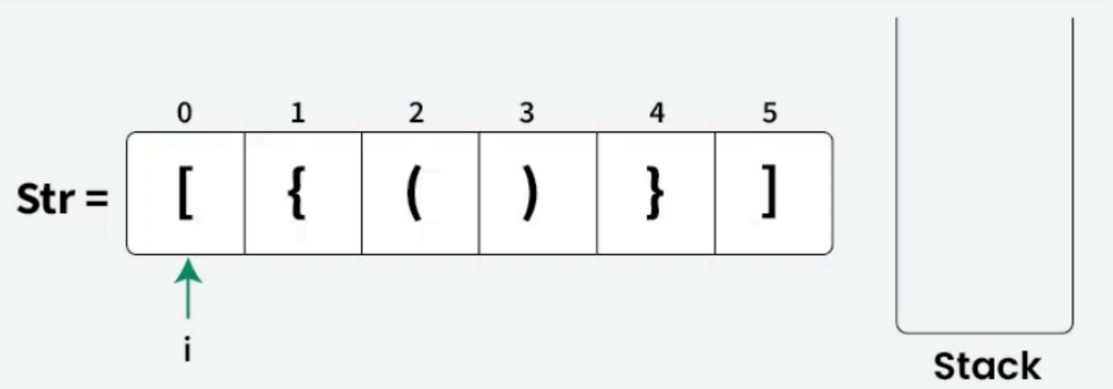
- Trình duyệt dùng 2 Stack để quản lý nút Back và Forward.
  - Back Stack: lưu các trang đã truy cập.
  - Forward Stack: lưu các trang bị quay lại từ Back.
- Cách hoạt động:
  - Khi mở một trang mới → URL hiện tại được push vào Back Stack.
  - Khi nhấn Back → URL bị pop khỏi Back Stack và push sang Forward Stack.
  - Khi nhấn Forward → URL được pop khỏi Forward Stack và mở lại.



2.2.4 Ứng dụng đời sống

Kiểm tra dấu ngoặc trong biểu thức:

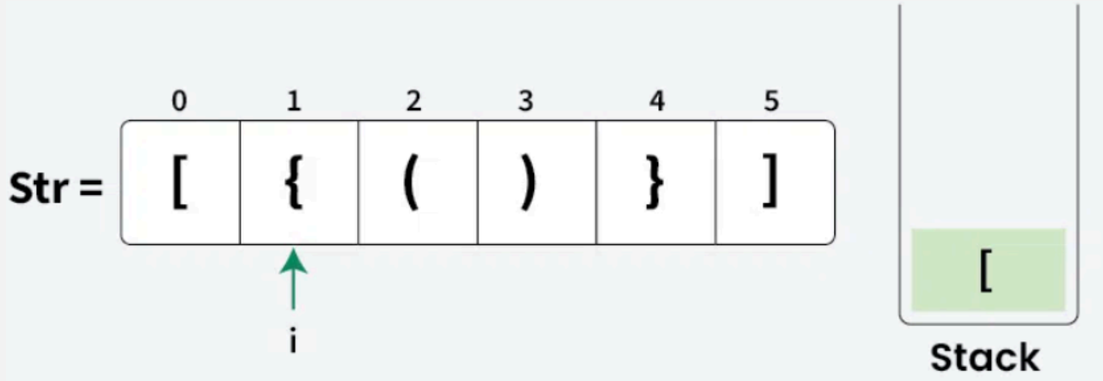
- Cách hoạt động:
  - Gặp dấu ngoặc mở (, {, [→ push vào Stack.
  - Gặp dấu ngoặc đóng), },] → pop từ Stack và kiểm tra có khớp với dấu mở gần nhất không.
  - Nếu khớp → tiếp tục, nếu không khớp hoặc Stack rỗng → biểu thức sai.



2.2.4 Ứng dụng đời sống

Kiểm tra dấu ngoặc trong biểu thức:

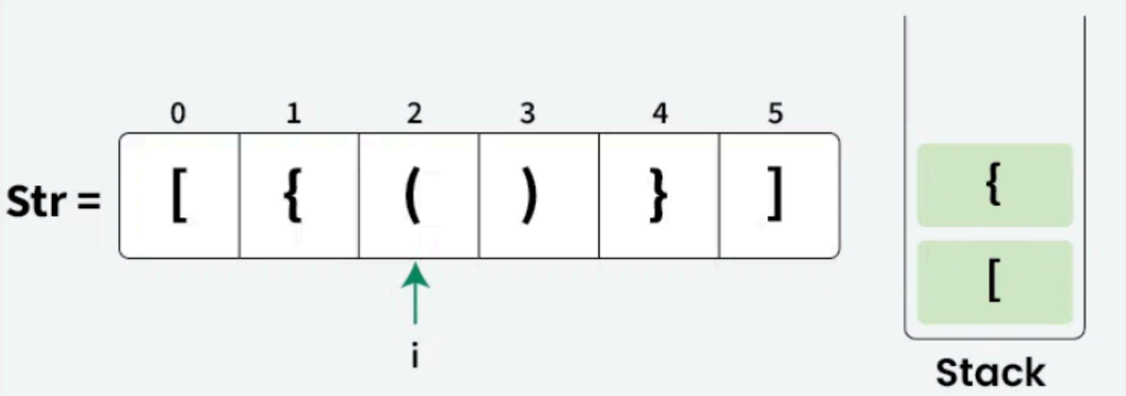
- Cách hoạt động:
  - Gặp dấu ngoặc mở (, {, [→ push vào Stack.
  - Gặp dấu ngoặc đóng), },] → pop từ Stack và kiểm tra có khớp với dấu mở gần nhất không.
  - Nếu khớp → tiếp tục, nếu không khớp hoặc Stack rỗng → biểu thức sai.



2.2.4 Ứng dụng đời sống

Kiểm tra dấu ngoặc trong biểu thức:

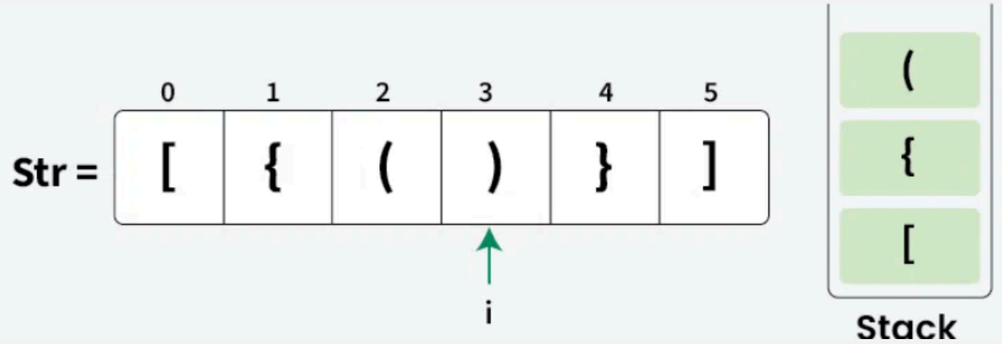
- Cách hoạt động:
  - Gặp dấu ngoặc mở (, {, [→ push vào Stack.
  - Gặp dấu ngoặc đóng), },] → pop từ Stack và kiểm tra có khớp với dấu mở gần nhất không.
  - Nếu khớp → tiếp tục, nếu không khớp hoặc Stack rỗng → biểu thức sai.



2.2.4 Ứng dụng đời sống

Kiểm tra dấu ngoặc trong biểu thức:

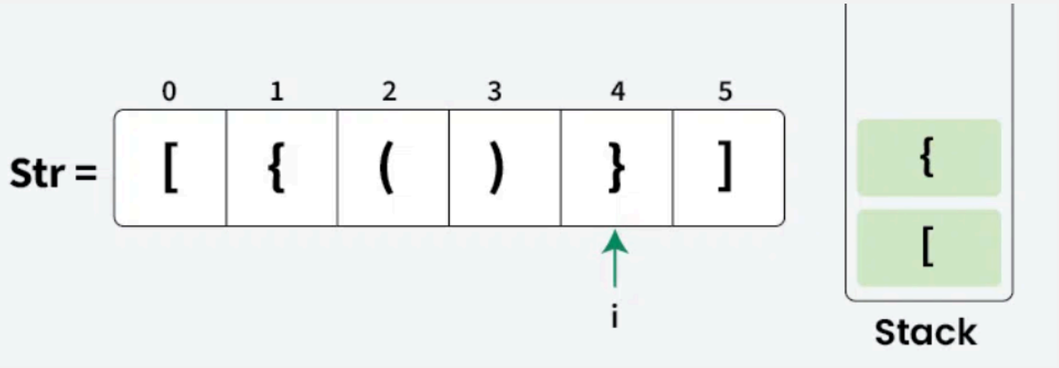
- Cách hoạt động:
  - Gặp dấu ngoặc mở (, {, [→ push vào Stack.
  - Gặp dấu ngoặc đóng), },] → pop từ Stack và kiểm tra có khớp với dấu mở gần nhất không.
  - Nếu khớp → tiếp tục, nếu không khớp hoặc Stack rỗng → biểu thức sai.



2.2.4 Ứng dụng đời sống

Kiểm tra dấu ngoặc trong biểu thức:

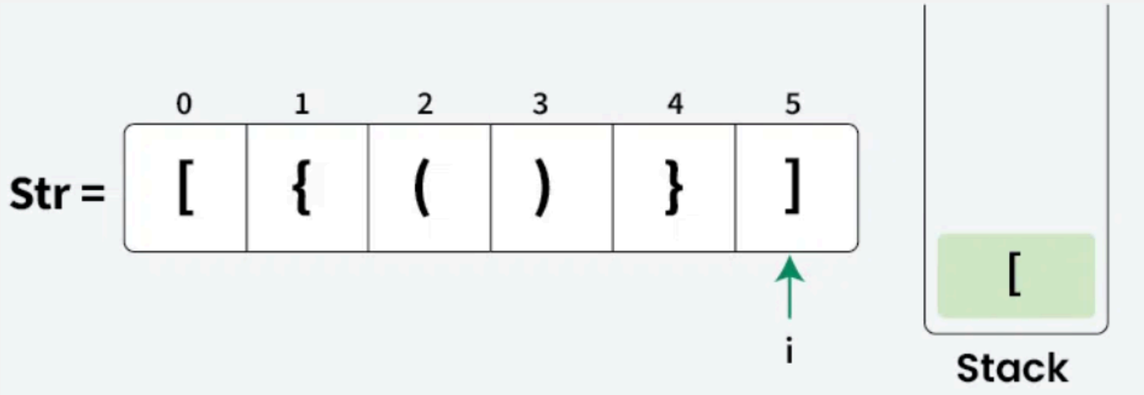
- Cách hoạt động:
  - Gặp dấu ngoặc mở (, {, [→ push vào Stack.
  - Gặp dấu ngoặc đóng), },] → pop từ Stack và kiểm tra có khớp với dấu mở gần nhất không.
  - Nếu khớp → tiếp tục, nếu không khớp hoặc Stack rỗng → biểu thức sai.



2.2.4 Ứng dụng đời sống

Kiểm tra dấu ngoặc trong biểu thức:

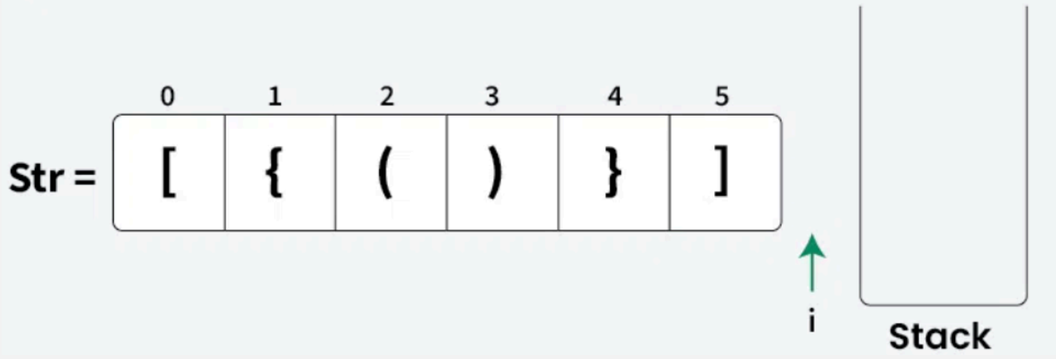
- Cách hoạt động:
  - Gặp dấu ngoặc mở (, {, [→ push vào Stack.
  - Gặp dấu ngoặc đóng), },] → pop từ Stack và kiểm tra có khớp với dấu mở gần nhất không.
  - Nếu khớp → tiếp tục, nếu không khớp hoặc Stack rỗng → biểu thức sai.



2.2.4 Ứng dụng đời sống

Kiểm tra dấu ngoặc trong biểu thức:

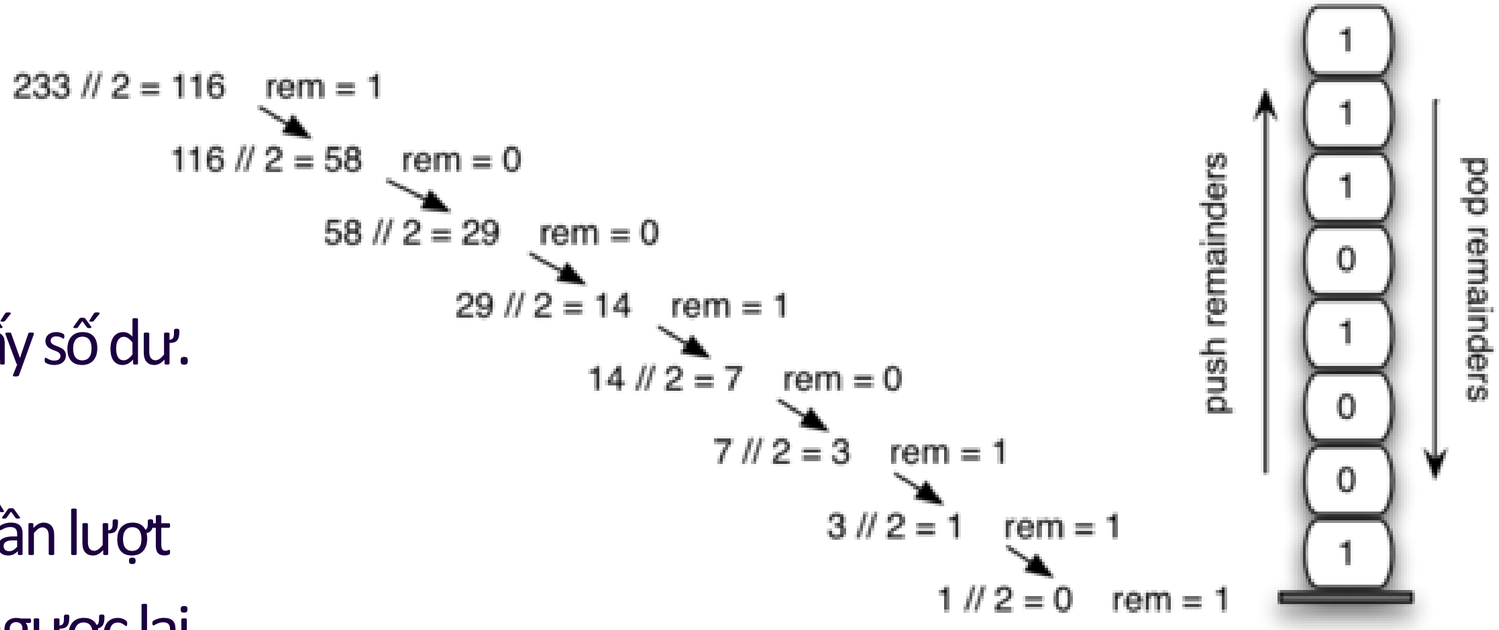
- Cách hoạt động:
  - Gặp dấu ngoặc mở (, {, [→ push vào Stack.
  - Gặp dấu ngoặc đóng), },] → pop từ Stack và kiểm tra có khớp với dấu mở gần nhất không.
  - Nếu khớp → tiếp tục, nếu không khớp hoặc Stack rỗng → biểu thức sai.



2.2.4 Ứng dụng đời sống

Chuyển đổi hệ cơ số

- Cách hoạt động:
  - Lần lượt chia số cho cơ số mới, lấy số dư.
  - Mỗi số dư được push vào Stack.
  - Sau khi chia đến khi kết quả = 0, lần lượt pop các phần tử ra theo thứ tự ngược lại để tạo số ở hệ mới.



2.2.5 Ứng dụng nâng cao

Ứng dụng trong nghiên cứu

- **Natural Language Processing(NLP)** : thuật toán Shift-reduce parsing dùng stack để xây dựng cây cú pháp.
- **Machine Learning**: giải tích ngược và backpropagation. Dùng stack để lưu trữ phép tính và duyệt ngược lại để tính gradient.
- **Machine vision**: các thuật toán flood-fill, region-growing

2.2.6 Minh hoạ bằng ngôn ngữ lập trình

C++ với mảng và thư viện std::stack

```cpp
#include <iostream>
#include <stack>

int main() {
    stack<int> s;            // Khởi tạo một ngăn xếp rỗng chứa số nguyên
    s.push(10);              // Thêm phần tử 10 vào đỉnh ngăn xếp
    s.push(20);              // Thêm phần tử 20 vào đỉnh ngăn xếp
    s.push(30);              // Thêm phần tử 30 vào đỉnh ngăn xếp

    cout << s.top() << " popped\n"; // In ra phần tử ở đỉnh (30)
    s.pop();                        // Xóa phần tử ở đỉnh (30)

    cout << "Top element is: " << s.top() << "\n"; // Xem phần tử ở đỉnh hiện tại (20) mà không xóa

    cout << "Elements present in stack: ";
    while (is.empty()) {        // Duyệt và in các phần tử còn lại trong ngăn xếp
        cout << s.top() << " "; // In phần tử ở đỉnh
        s.pop();                // Xóa phần tử vừa in
    }
    cout << "\n";
    return 0;
}
```

2.2.6 Minh hoạ bằng ngôn ngữ lập trình

Python với list

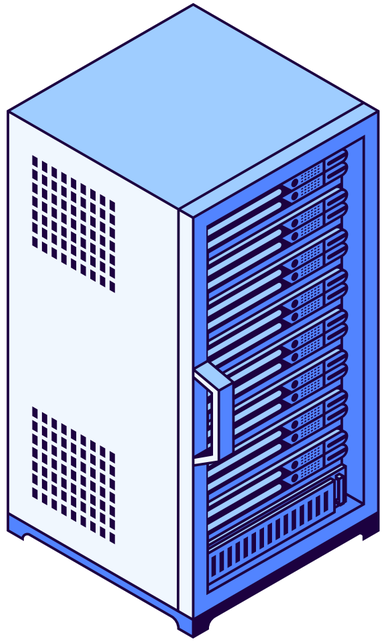
```
def create_stack():
    return []
def is_empty(stack):
    return len(stack) == 0
def push(stack, item):
    stack.append(item)
    print("pushed item:", item)
def pop(stack):
    if is_empty(stack):
        return "stack is empty"
    return stack.pop()

# Test
stack = create_stack()
push(stack, 1)
push(stack, 2)
push(stack, 3)

print("popped item:", pop(stack))
print("stack after popping:", stack)
```

2.2.6 Minh hoạ bằng ngôn ngữ lập trình

Python với collections.deque

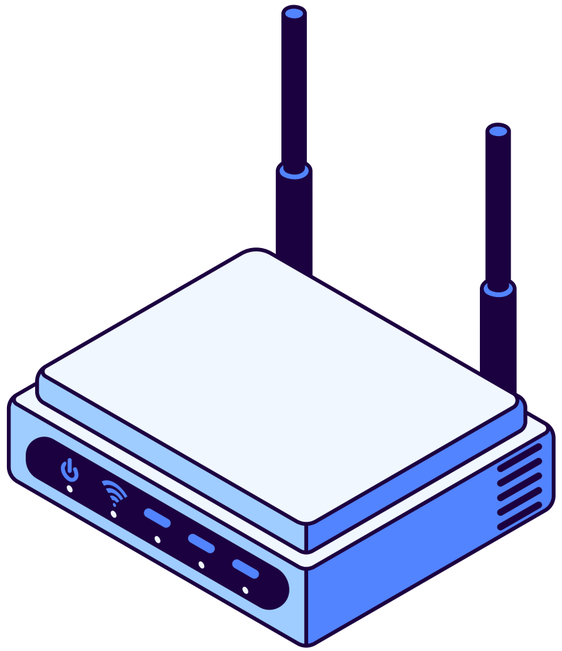
```
from collections import deque

def create_stack():
    return deque()
def is_empty(stack):
    return len(stack) == 0
def push(stack, item):
    stack.append(item)  # append to end
    print("pushed item:", item)
def pop(stack):
    if is_empty(stack):
        return "stack is empty"
    return stack.pop()  # pop from end

# Teststack = create_stack()
push(stack, 1)
push(stack, 2)
push(stack, 3)

print("popped item:", pop(stack))
print("stack after popping:", stack)
```

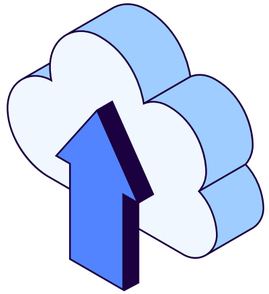
KẾT LUẬN



Các kết quả đạt được

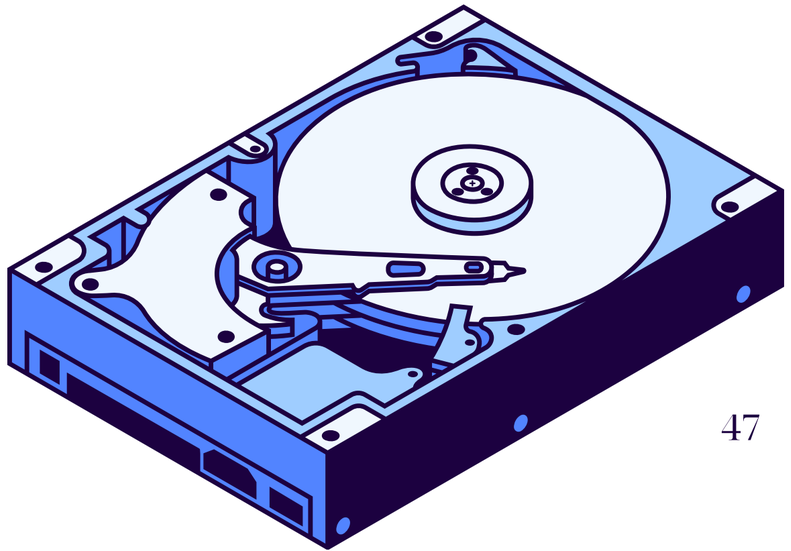
- Hiểu rõ lý thuyết về stack và nguyên lý LIFO.
- Thành thục các thao tác như là: push, pop, peek/top, IsEmpty, IsFull.





Hướng phát triển mới

- Ứng dụng trong hệ thống nhúng
- Hỗ trợ hiệu quả cho các ứng dụng trí tuệ nhân tạo, xử lý dữ liệu lớn và lập trình song song.




Thank you!