TRƯỜNG ĐẠI HỌC SƯ PHẠM KỸ THUẬT TP.HCM

KHOA CÔNG NGHỆ THÔNG TIN

BÁO CÁO VỀ NGĂN XẾP (STACK)

Course Code: DASA230179

Giáo viên HD:	Th.S Vũ Đình Bảo	
Sinh viên:	Nguyễn Hùng Dũng	MSSV: 22134002
	Trần Như Hoàng	MSSV: 22134006
	Trần Nguyễn Phương Bình	MSSV: 24133006
	Phạm Ngọc Phúc	MSSV: 22134010
	Võ Hồng Quân	MSSV: 22134012

Academic Year 2025-2026

Phân công công việc nhóm

TT	Họ và tên	Chức vụ	Nhiệm vụ được phân công
1	Võ Hồng Quân	Nhóm trưởng	Xây dựng nội dung Thuyết trình Thiết kế slide Thực hiện web demo
2	Nguyễn Hùng Dũng	Thành viên	Xây dựng nội dung Thuyết trình
3	Trần Như Hoàng	Thành viên	Xây dựng nội dung Thiết kế slide
4	Trần Nguyên Phương Bình	Thành viên	Xây dựng nội dung Thiết kế slide
5	Phạm Ngọc Phúc	Thành viên	Xây dựng nội dung Thiết kế slide

LỜI CẢM ƠN

Để hoàn thành báo cáo này, chúng em đã nhận được sự hỗ trợ vô giá từ nhiều cá nhân và tổ chức. Chúng em xin gửi lời cảm ơn chân thành đến:

- **Ban Giám hiệu Trường Đại học Sư phạm Kỹ thuật Thành phố Hồ Chí Minh** cùng tập thể giảng viên Khoa Công nghệ Thông tin, những người đã tạo điều kiện học tập và nghiên cứu trong một môi trường thuận lợi và hiệu quả.
- Người hướng dẫn của chúng em, thầy **Vũ Đình Bảo**, với sự chỉ dẫn tận tình, đã đóng vai trò quan trọng trong mọi giai đoạn của quá trình thực hiện đề tài.

Mặc dù chúng em đã nỗ lực hết sức để tiến hành làm báo cáo một cách nghiêm túc và chặt chẽ, nhưng chắc chắn vẫn còn những thiếu sót. Vì vậy, chúng em rất mong nhận được những ý kiến đóng góp và gợi ý từ thầy để có thể tiếp tục hoàn thiện công việc của mình trong tương lai.

TÓM TẮT

Báo cáo này tập trung nghiên cứu cấu trúc dữ liệu ngăn xếp (Stack), một kiểu dữ liệu trừu tượng vận hành theo nguyên lý LIFO (Last In, First Out - Vào sau, Ra trước). Nội dung trình bày các khái niệm cơ bản, thao tác điển hình, phương pháp cài đặt bằng mảng và danh sách liên kết, cũng như phân tích độ phức tạp và hiệu năng thông qua thực nghiệm. Bên cạnh đó, báo cáo còn khảo sát nhiều ứng dụng thực tiễn của Stack như quản lý lời gọi hàm đệ quy, chuyển đổi và đánh giá biểu thức, thuật toán DFS và Backtracking, cũng như các tính năng trong phần mềm như Undo/Redo, lịch sử duyệt web và xử lý biểu thức. Ngoài ra, Stack còn có vai trò quan trọng trong các lĩnh vực nâng cao như xử lý ngôn ngữ tự nhiên, học máy và thị giác máy tính. Kết quả nghiên cứu cho thấy Stack là một cấu trúc dữ liệu nền tảng, có tính ứng dụng rộng rãi trong cả lý thuyết lẫn thực hành, đồng thời mở ra hướng phát triển trong tối ưu hóa và kết hợp với các cấu trúc dữ liệu khác để giải quyết những bài toán hiện đại.

Từ khóa: **Stack, LIFO**.

Mục lục

GIỚI THIỆU

Chương này cung cấp một cái nhìn tổng quan về ngăn xếp (**Stack**), bao gồm bối cảnh, mục tiêu, giới hạn nghiên cứu và phương pháp thực hiện.

1.1 Bối cảnh chủ đề

Trong kỷ nguyên công nghệ số và cách mạng 4.0, lập trình đã trở thành một kỹ năng cốt lõi, đóng góp cho nhiều lĩnh vực khác như tài chính, y tế, giáo dục, giao thông, giải trí,... bởi hầu hết các phần mềm hệ thống và dịch vụ hiện đại đều cần đến lập trình. Tuy nhiên, để có thể lập trình được các ứng dụng phần mềm chẳng hạn như trí tuệ nhân tạo (AI), phân tích dữ liệu lớn (Big Data), blockchain hay Internet vạn vật (IoT), người học trước tiên cần có kỹ thuật tốt và nền tảng vững chắc.

Nền tảng then chốt hàng đầu mà bất kỳ lập trình viên nào cũng phải nắm vững chính là **Cấu trúc dữ liệu và Giải thuật**. Đây được xem như "xương sống", là những viên gạch nền móng để có thể xây dựng các phần mềm phức tạp và hiệu quả. Mối quan hệ giữa **Cấu trúc dữ liệu và Giải thuật** là mối quan hệ hai chiều. Một thuật toán dù tối ưu đến đâu nhưng đi kèm với **Cấu trúc dữ liệu** không thích hợp thì vẫn kém hiệu quả, khó mở rộng và tốn kém tài nguyên. Ngược lại, một **Cấu trúc dữ liệu** hợp lý nhưng gắn với **Giải thuật** không tốt có thể dẫn đến sai kết quả. Điều này cho thấy **Cấu trúc dữ liệu và Giải thuật** luôn gắn bó mật thiết, bổ trợ cho nhau. Trong đó, **Cấu trúc dữ liệu** giúp lưu trữ và truy cập dữ liệu nhanh chóng, trong khi **Giải thuật** là tập hợp các bước hoặc quy tắc nhằm giải quyết vấn đề và tối ưu hóa xử lý. Hai yếu tố này cung cấp phương pháp tổ chức, lưu trữ và quản lý dữ liệu một cách có hệ thống, từ đó nâng cao hiệu suất của chương trình. Đặc biệt, trong bối cảnh khối lượng dữ liệu khổng lồ được tạo ra mỗi giây như hiện nay, việc lựa chọn **Cấu trúc dữ liệu và Giải thuật** phù hợp trở nên vô cùng quan trọng.

Đặc biệt, khi tìm hiểu sâu hơn, lập trình viên sẽ được hiểu biết về các kiểu dữ liệu trừu tượng (ADT) cơ bản thường được sử dụng khi xây dựng chương trình trên máy tính, cũng như cách hiện thực và áp dụng chúng trong thực tế. Trong đó ADT (Abstract Data Type) là kiểu dữ liệu trừu tượng chỉ mô tả các thao tác có thể thực hiện (hành vi), chứ không quan tâm cách nó được cài đặt ra sao. Những ADT thường gặp gồm: List, **Stack**, Queue,

Deque, Priority Queue, Set và Map/Dictionary, cùng Disjoint Set. Các ADT này được hiện thực bởi nhiều cấu trúc dữ liệu như: mảng/array động và linked list (cho List), heap (cho Priority Queue), hash table hoặc cây tìm kiếm cân bằng như AVL/Red–Black/B-tree (cho Map/Set), Union–Find/DSU (cho Disjoint Set), hay đồ thị với adjacency list/matrix.

Trong số đó, ngăn xếp (**Stack**) là một ADT nền tảng vận hành theo nguyên lý LIFO và có thể hiện thực đơn giản bằng mảng hoặc danh sách liên kết. Dù xuất hiện sớm, **Stack** vẫn giữ vai trò cốt lõi trong thiết kế thuật toán và hệ thống hiện đại (call **Stack**, xử lý biểu thức, DFS, undo/redo).

1.2 Mục tiêu của báo cáo

Tiểu luận cung cấp một cái nhìn toàn diện, có cơ sở khoa học và hệ thống về đặc trưng, nguyên lý vận hành cũng như vai trò ứng dụng của **Stack** trong khoa học máy tính và phát triển phần mềm. Cụ thể, tiểu luận hướng đến việc làm rõ các khái niệm nền tảng và thao tác cơ bản, phân tích hiệu suất và phương pháp cài đặt, đồng thời minh chứng tính ứng dụng rộng rãi của **Stack** thông qua các ví dụ thực tiễn và mã nguồn minh họa đa ngôn ngữ. Thông qua đó, người đọc không chỉ nắm vững kiến thức lý thuyết, mà còn hình thành năng lực vận dụng **Stack** như một công cụ hiệu quả để giải quyết các bài toán lập trình cụ thể.

1.3 Giới hạn nghiên cứu

Tiểu luận này giới hạn phạm vi nghiên cứu trong khuôn khổ cơ bản của **Cấu trúc dữ liệu Stack**, cụ thể là tập trung vào nguyên lý LIFO cùng các thao tác cơ bản như Push, Pop, Peek và isEmpty. Đồng thời xem xét hai phương pháp cài đặt chính là mảng và danh sách liên kết. Các trường hợp ứng dụng được phân tích chọn lọc trong một số lĩnh vực tiêu biểu như Call **Stack** trong hệ thống máy tính, xử lý biểu thức, thuật toán DFS và cơ chế Undo/Redo trong giao diện người dùng. Mã minh họa được triển khai bằng năm ngôn ngữ lập trình phổ biến gồm C++, Java, Python, C# và TypeScript, với mục tiêu minh chứng tính phổ quát của **Stack** mà không đi sâu vào các kỹ thuật tối ưu hóa hay thư viện nâng cao. Ngoài ra, tiểu luận không mở rộng sang việc so sánh với các **Cấu trúc dữ liệu** khác ngoài Queue ở mức cơ bản và cũng không đi sâu vào khía cạnh phần cứng.

1.4 Phương pháp thực hiện

Để đảm bảo tiểu luận được thực hiện một cách khoa học, có căn cứ, có hệ thống và đáp ứng đầy đủ các mục tiêu đề ra, nhóm đã áp dụng các phương pháp nghiên cứu chính sau đây:

1. Phương pháp nghiên cứu tài liệu: thu thập thông tin, tài liệu từ các nguồn có uy tín. Từ đó, tập trung nghiên cứu qua các tài liệu thu thập được (giáo trình **Cấu trúc dữ liệu và Giải thuật**,...).

2. Phương pháp phân tích và tổng hợp lý thuyết: so sánh, đối chiếu các thông tin, đánh giá ưu nhược điểm để rút ra các kết luận có giá trị, tổng hợp lại các lý thuyết tìm hiểu được một cách rõ ràng, đảm bảo tham khảo thông tin đúng trọng tâm, tập trung vào lý thuyết, cách vận hành, ứng dụng **Stack**,...Đồng thời phân tích vai trò của **Stack** trong từng ứng dụng cụ thể (ví dụ: Call **Stack**, tính toán biểu thức), giải thích lý do tại sao nó phù hợp hơn các **Cấu trúc dữ liệu** khác.

3. Phương pháp thực nghiệm: mô phỏng code trên nhiều ngôn ngữ khác nhau để so sánh sự khác biệt về cú pháp và cách tiếp cận, đồng thời làm nổi bật tính phổ quát của **Cấu trúc dữ liệu**.

4. Phương pháp làm việc nhóm: để đảm bảo tiến độ và chất lượng công việc, nhóm đã xây dựng kế hoạch làm việc rõ ràng, phân công công việc dựa trên thế mạnh của từng thành viên, chia các mốc thời gian cho từng công việc và đảm bảo tuân thủ theo lộ trình để đề ra. Đồng thời, các thành viên thường xuyên trao đổi, đánh giá chéo và góp ý cho các phần việc của nhau để đảm bảo tính nhất quán và chính xác của toàn bộ đề tài.

NỘI DUNG CHÍNH

2.1 Ngăn xếp

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

Ngăn xếp (**Stack**) là một dạng danh sách được cài đặt để phục vụ cho các ứng dụng cần xử lý theo thứ tự đảo ngược. Trong cấu trúc dữ liệu ngăn xếp, tất cả các thao tác thêm hoặc xóa một phần tử đều phải thực hiện tại một đầu duy nhất của danh sách, đầu này gọi là đỉnh (top) của ngăn xếp.

Ngăn xếp là một cấu trúc dữ liệu trừu tượng có tính chất: vào sau, ra trước (Last In, First Out hay viết tắt là LIFO). Có thể hình dung ngăn xếp thông qua hình ảnh một chồng đĩa đặt trên bàn: nếu muốn thêm một chiếc đĩa, ta phải đặt nó lên trên cùng; nếu muốn lấy ra một chiếc đĩa, ta cũng phải lấy chiếc ở trên cùng.

Các thao tác cơ bản gồm **push**, **pop**, **peek** (xem phần tử đỉnh mà không loại bỏ) và **isEmpty** (kiểm tra rỗng); với cài đặt mảng cố định ta có thêm **isFull** (kiểm tra đầy).

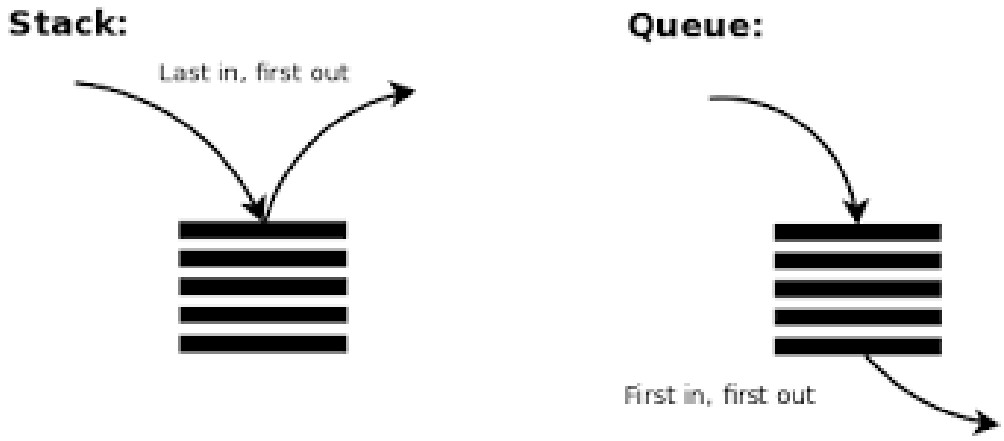
Nhờ vào việc chỉ thao tác tại đỉnh, các phép **push**/**pop**/**peek**/**isEmpty** đều có độ phức tạp thời gian $O(1)$. Với mảng động, độ phức tạp này được đảm bảo ở mức trung bình (amortized $O(1)$).

2.1.2 So sánh với hàng đợi (Queue)

Ngăn xếp (Stack) và hàng đợi (Queue) tuy cùng là cấu trúc dữ liệu tuyến tính nhưng lại tuân theo nguyên tắc hoạt động hoàn toàn trái ngược: ngăn xếp vận hành theo LIFO (Last In, First Out), trong đó phần tử được thêm vào cuối cùng sẽ là phần tử đầu tiên được loại bỏ, còn hàng đợi áp dụng FIFO (First In, First Out) – phần tử thêm vào trước sẽ được lấy ra trước. Sự khác biệt này được minh họa rõ ràng trong hình 2.1.

Về mặt cài đặt và quản lý, ngăn xếp chỉ cần một con trỏ **top** để quản lý đỉnh, cho phép thao tác push và pop luôn thực hiện tại cùng một vị trí, trong khi hàng đợi yêu cầu hai con trỏ **front** và **rear** để phân biệt vị trí thêm (enqueue) và loại bỏ (dequeue).

Ngăn xếp được sử dụng rộng rãi trong quản lý lời gọi hàm đệ quy, thuật toán quay lui, đánh giá biểu thức và chức năng undo/redo, nhờ khả năng tận dụng tốt locality of reference



Hình 2.1: So sánh cơ chế hoạt động của Ngăn xếp (LIFO) và Hàng đợi (FIFO)

và hỗ trợ tự động quản lý bộ nhớ qua stack frame. Ngược lại, hàng đợi thường được ứng dụng trong lập lịch CPU, quản lý tác vụ in, duyệt đồ thị theo chiều rộng (BFS), buffering I/O và mô hình producer–consumer, đảm bảo thứ tự xử lý công bằng và phù hợp với luồng dữ liệu liên tục.

Sự khác biệt giữa LIFO và FIFO không chỉ quyết định cách thức truy xuất dữ liệu mà còn ảnh hưởng đến hiệu suất bộ nhớ, cache và ứng dụng thực tế của từng cấu trúc trong khoa học máy tính.

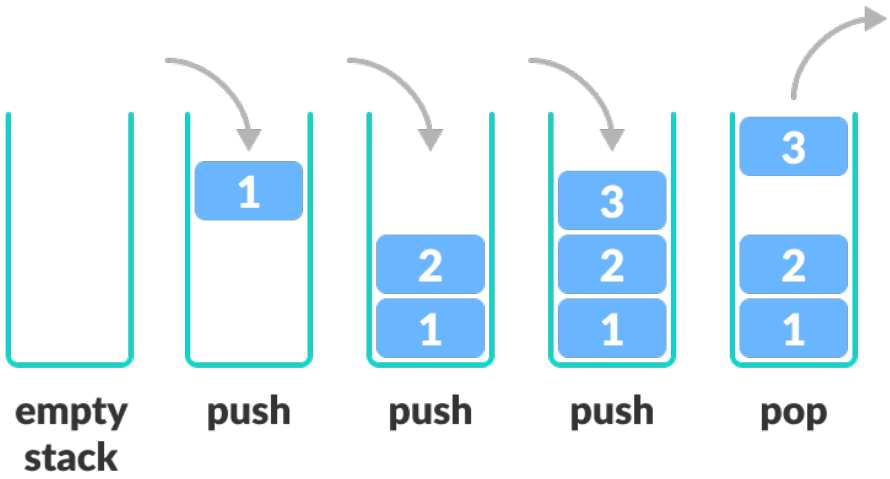
2.1.3 Các thao tác cơ bản

2.1.3.1 Push

Push được sử dụng để thêm một phần tử mới vào đỉnh ngăn xếp. Sau khi đẩy, phần tử mới sẽ trở thành đỉnh mới của ngăn xếp. Kích thước của ngăn xếp tăng thêm 1 sau mỗi lần đẩy thành công. Trong trường hợp ngăn xếp được cài đặt bằng mảng có cố định kích thước, thao tác đẩy có thể thất bại nếu ngăn xếp đã đầy (trạng thái overflow).
Ví dụ: nếu ngăn xếp đang chứa [A, B] (với B là đỉnh), sau khi thực hiện Push(C), ngăn xếp sẽ trở thành [A, B, C] (với C là đỉnh mới).

2.1.3.2 Pop

Pop được sử dụng để lấy ra và đồng thời xóa phần tử đang ở vị trí trên cùng (top) của. Phần tử ngay bên dưới (nếu có) sẽ trở thành mới. Kích thước của sẽ giảm đi 1 sau mỗi lần pop thành công.
Ví dụ: nếu stack đang là [A, B, C] (C là đỉnh), sau khi thực hiện Pop(C), stack sẽ trở thành [A, B] (B là đỉnh mới) và giá trị C sẽ được trả về.



Hình 2.2: Các thao tác cơ bản của ngăn xếp

2.1.3.3 Peek/Top

Peek (Top) cho phép chúng ta xem giá trị của phần tử đang ở đỉnh stack mà không xóa nó. Thao tác này rất hữu ích khi bạn cần biết phần tử tiếp theo sẽ được xử lý là gì mà không muốn thay đổi trạng thái hiện tại của ngăn xếp.
Ví dụ: nếu stack là [A, B, C] (C là đỉnh), thì việc thực hiện Peek() sẽ trả về giá trị C, nhưng stack vẫn giữ nguyên là [A, B, C].

2.1.3.4 isEmpty và isFull

isEmpty(): Kiểm tra ngăn xếp có rỗng hay không. Trả về true nếu ngăn xếp rỗng, và false nếu vẫn còn phần tử. Thao tác này đặc biệt quan trọng để tránh lỗi **Stack Underflow** khi thực hiện pop hoặc peek trên một ngăn xếp rỗng.

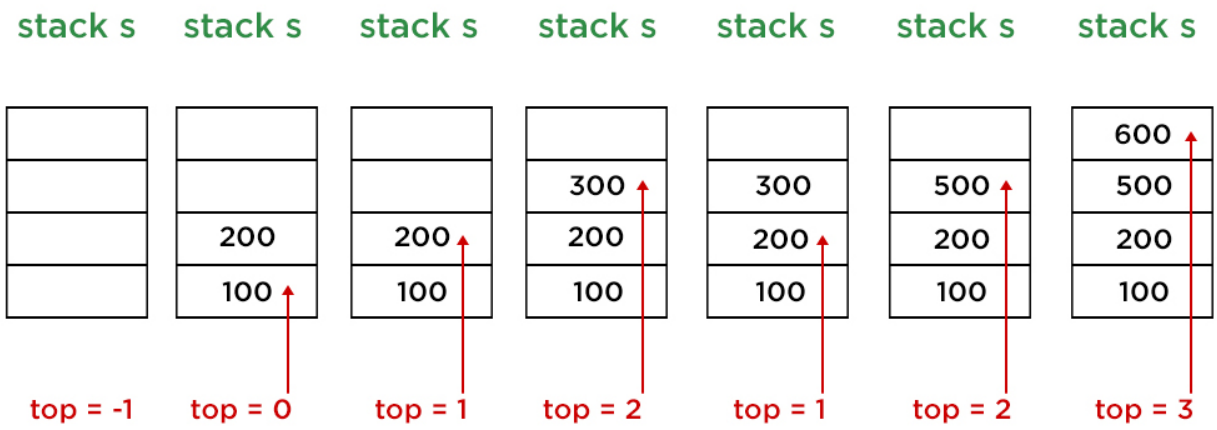
isFull(): Kiểm tra ngăn xếp có đầy hay chưa (chỉ áp dụng khi stack được cài đặt bằng mảng với kích thước cố định). Trả về true nếu ngăn xếp đã đầy, và false nếu vẫn còn chỗ trống. Thao tác này giúp ngăn chặn lỗi **Stack Overflow** khi cố gắng thực hiện push vào một ngăn xếp đã đầy.

2.1.3.5 size

Trả về số lượng phần tử hiện đang có trong ngăn xếp. Hàm này hữu ích để theo dõi dung lượng của **Stack** tại một thời điểm, ví dụ để kiểm tra xem ngăn xếp đang rỗng, đầy, hay đang chứa bao nhiêu phần tử.

2.1.4 Các phương pháp cài đặt Stack

2.1.4.1 Cài đặt bằng mảng (Array-based)



Hình 2.3: Minh họa thao tác khi cài đặt bằng mảng

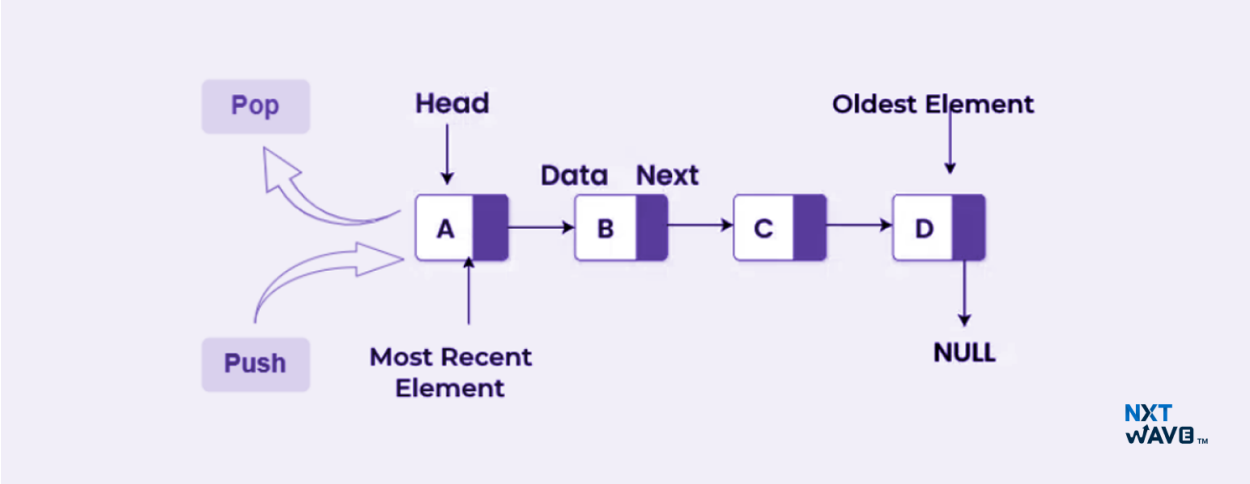
Đây là phương pháp cài đặt ngăn xếp đơn giản và phổ biến, sử dụng một mảng một chiều để lưu trữ các phần tử. Một biến đặc biệt, thường được gọi là **top**, được sử dụng để theo dõi chỉ số (index) của phần tử nằm ở đỉnh ngăn xếp. Các bước cài đặt bằng mảng bao gồm:

- **Khởi tạo:** Trước tiên, cần cấp phát một mảng với kích thước cố định được định nghĩa là **maxSize**. Đồng thời, khởi tạo biến **top** bằng **-1** để biểu thị rằng ngăn xếp hiện đang rỗng, chưa có phần tử nào được đưa vào.
- **Push(phần tử):** Khi muốn thêm một phần tử mới vào ngăn xếp, trước hết cần kiểm tra xem ngăn xếp đã đầy chưa thông qua hàm **isFull**. Nếu **top == maxSize 1**, tức là mảng đã đầy, thao tác này không thể thực hiện được vì sẽ gây lỗi tràn ngăn xếp (Stack Overflow). Nếu ngăn xếp chưa đầy, ta tăng biến **top** lên một đơn vị và gán giá trị của phần tử mới vào vị trí **array[top]**.
- **Pop():** Khi muốn lấy ra một phần tử từ ngăn xếp, trước hết cần kiểm tra xem ngăn xếp có rỗng hay không bằng hàm **isEmpty**. Nếu **top == -1**, nghĩa là ngăn xếp rỗng, thao tác này không thể thực hiện được vì sẽ gây lỗi tràn ngược (Stack Underflow). Nếu ngăn xếp có phần tử, ta lấy giá trị tại vị trí **array[top]**, trả về giá trị này, đồng thời giảm biến **top** đi một đơn vị để cập nhật đỉnh ngăn xếp mới.

- **Peek()/Top():** Thao tác này cho phép quan sát giá trị của phần tử nằm ở đỉnh ngăn xếp mà không loại bỏ nó. Tương tự như **Pop()**, trước khi thực hiện cũng cần kiểm tra xem ngăn xếp có rỗng hay không bằng **isEmpty**. Nếu ngăn xếp không rỗng, kết quả trả về là giá trị tại vị trí **array[top]**, trong khi biến **top** vẫn giữ nguyên.

Ghi chú: Khi sử dụng các cấu trúc dữ liệu *mảng động* (các mảng tự động thay đổi kích thước khi thêm phần tử vượt quá dung lượng hiện tại) như **std::vector** trong C++ hay **list** trong Python thì các thao tác **push**, **pop**, **peek** và **isEmpty** thường có độ phức tạp thời gian $O(1)$ (trong trường hợp trung bình amortized); khi đó việc kiểm tra **isFull** thường không cần thiết.

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



Hình 2.4: Minh họa thao tác khi cài đặt bằng danh sách liên kết

Trong phương pháp này, ngăn xếp không được cài đặt trên một mảng có kích thước cố định, mà được xây dựng dựa trên danh sách liên kết đơn (singly linked list). Mỗi phần tử trong ngăn xếp được biểu diễn bởi một *node*, trong đó mỗi node bao gồm hai thành phần chính: (i) dữ liệu của phần tử, và (ii) một con trỏ (trỏ **next**) trỏ đến node liền kề bên dưới nó trong ngăn xếp. Biến **top** đóng vai trò quan trọng, luôn trỏ đến node hiện đang nằm ở đỉnh của ngăn xếp. Nhờ vậy, các thao tác thêm hoặc loại bỏ phần tử đều có thể thực hiện nhanh chóng tại vị trí này.

- **Khởi tạo:** Khi bắt đầu, gán **top = NULL** để biểu thị ngăn xếp rỗng. Điều này có nghĩa là chưa có node nào được tạo, và mọi thao tác **pop** hay **peek** cần phải kiểm tra điều kiện rỗng trước khi thực hiện.

- **Push(phần tử):** Để đưa một phần tử mới vào ngăn xếp, trước tiên cần tạo một node mới và gán dữ liệu cho nó. Tiếp theo, cho con trỏ **next** của node mới trỏ đến node hiện đang ở vị trí **top**. Cuối cùng, cập nhật **top** để trỏ sang node mới này. Nhờ cách làm này, node mới sẽ trở thành đỉnh của ngăn xếp và được xử lý đầu tiên trong các thao tác tiếp theo.

- **Pop():** Khi cần loại bỏ phần tử ở đỉnh ngăn xếp, trước tiên phải kiểm tra xem ngăn xếp có rỗng không (**top == NULL**). Nếu ngăn xếp rỗng, thao tác này không hợp lệ và thường gây ra lỗi **Stack Underflow**. Nếu ngăn xếp không rỗng, tiến hành lưu node hiện tại mà **top** trỏ đến vào một biến tạm, sau đó cập nhật **top = top → next** để dịch con trỏ đỉnh xuống phần tử bên dưới. Dữ liệu trong node tạm được trả về cho người dùng, đồng thời node này được giải phóng khỏi bộ nhớ để tránh rò rỉ (memory leak).

- **Peek():** Tương tự như **Pop()**, trước tiên cần kiểm tra xem ngăn xếp có rỗng không. Nếu không rỗng, thao tác **peek** sẽ trả về dữ liệu của node mà **top** đang trỏ đến, nhưng không thay đổi vị trí của **top**. Điều này cho phép người dùng quan sát giá trị ở đỉnh ngăn xếp mà vẫn giữ nguyên trạng thái của cấu trúc dữ liệu.

Ghi chú: Phương pháp cài đặt bằng danh sách liên kết có ưu điểm nổi bật là kích thước động, ngăn xếp có thể mở rộng hoặc thu hẹp tuỳ theo số lượng phần tử thực tế, chỉ giới hạn bởi bộ nhớ hệ thống. Vì vậy, không có khái niệm "ngăn xếp đầy" như khi dùng mảng cố định. Tuy nhiên, nhược điểm là mỗi node cần thêm một vùng nhớ để lưu con trỏ **next**, và do các node không nằm liền kề trong bộ nhớ, hiệu quả truy cập bộ nhớ (cache locality) thường kém hơn so với cách cài đặt bằng mảng.

2.1.4.3 So sánh hai phương pháp cài đặt Stack

Tiêu chí	Mảng	Danh sách liên kết
Ưu điểm	Cài đặt đơn giản. Truy cập đỉnh nhanh (cache locality tốt).	Kích thước động, linh hoạt. Không lo bị tràn stack (stack overflow) 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.

Bảng 2.1: So sánh cài đặt Stack bằng mảng và danh sách liên kết

2.1.5 Phân tích độ phức tạp

2.1.5.1 Độ phức tạp thời gian của các thao tác cơ bản

Do tất cả các thao tác chính trên ngăn xếp (`push`, `pop`, `peek`, `isEmpty`) đều được thực hiện trực tiếp tại đỉnh, nên về mặt lý thuyết có độ phức tạp $O(1)$. Tuy nhiên, sự khác biệt nằm ở cách cài đặt cụ thể (mảng cố định, mảng động hay danh sách liên kết):

Thao tác	Mảng (cố định)	Mảng (động)[1]	Danh sách liên kết
`push`	$O(1)$	$O(1)$ (*Tệ nhất* $O(n)$)	$O(1)$
`pop`	$O(1)$	$O(1)$	$O(1)$
`peek/top`	$O(1)$	$O(1)$	$O(1)$
`isEmpty`	$O(1)$	$O(1)$	$O(1)$

Bảng 2.2: Độ phức tạp thời gian các thao tác chính trên Stack

Nhận xét: Đối với **mảng động**, thao tác `push` hầu hết là $O(1)$; chỉ khi phải resize mới tốn $O(n)$ vì thế tính trung bình (amortized), vẫn xem là $O(1)$. Trong khi đó, **danh sách**

[1]Ví dụ: `std::vector` trong C++, `ArrayList`/ hay `list` trong Python.

liên kết mặc dù cũng $O(1)$ cho `push/pop`, nhưng do mỗi node cần thêm bộ nhớ cho con trỏ và các node rải rác trong RAM, nên hiệu suất thực tế thường chậm hơn mảng và chỉ thực sự phù hợp trên bộ dữ liệu lớn và cần n được thêm bớt liên tục. Nếu cần thao tác `size()` chạy $O(1)$, ta chỉ cần duy trì thêm một biến đếm phần tử và cập nhật khi `push/pop`.

2.1.5.2 Không gian lưu trữ

Đặc điểm	Mảng cố định	Mảng 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)	$\Theta(n)$, có resize khi đầy	$\Theta(n)$ dữ liệu + $\Theta(n)$ 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

Bảng 2.3: So sánh độ phức tạp bộ nhớ của các phương pháp cài đặt Stack

Nhận xét: Tùy vào trường hợp ứng dụng thực tế mà ta phải lựa chon phương pháp cài đặt Stack khác nhau. Mỗi phương pháp đều có những hạn chế riêng mà ta phải cân nhắc trước khi sử dụng. Nếu sử dụng mảng cố định, việc xác định trước kích thước `maxSize` là điểm yếu lớn nhất: chọn quá nhỏ thì dễ gây tràn (overflow), còn chọn quá lớn thì dẫn đến lãng phí bộ nhớ. Với mảng động, tuy linh hoạt hơn, nhưng việc mở rộng dung lượng (resize) khi đầy sẽ phải sao chép toàn bộ dữ liệu sang vùng nhớ mới, gây chi phí thời gian $O(n)$ trong trường hợp xấu nhất; ngoài ra, kỹ thuật cấp phát dư thừa để giảm số lần resize cũng có thể làm tốn thêm một phần bộ nhớ không thực sự sử dụng. Trong khi đó, danh sách liên kết giải quyết được vấn đề cố định dung lượng, nhưng lại phát sinh overhead bộ nhớ vì mỗi node phải lưu thêm con trỏ `next`, đồng thời các node phân tán trong RAM khiến cache locality kém, nên tốc độ truy cập thực tế thường chậm hơn mảng.

2.1.5.3 Đánh giá

Có 2 phương pháp chính để đánh giá thời gian thực hiện thuật toán được so sánh trong bảng sau:

Từ bảng so sánh các phương pháp đánh giá, có thể thấy rằng trong trường hợp nghiên cứu cấu trúc dữ liệu **Stack**, cách tiếp cận bằng thực nghiệm là trực quan và dễ triển khai nhất.

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

Bảng 2.4: Bảng so sánh hai phương pháp đánh giá thuật toán

Ngôn ngữ lập trình được chọn là **Python** (cú pháp đơn giản, có sẵn `list` hỗ trợ `append`/`pop`) và **C++** (cho hiệu năng cao, dùng `std::stack` dựa trên `vector`). Thực nghiệm được tiến hành trên chip **Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz**.

Benchmark với Python

```
import time
stack = []
N_1 = 1000000
N_2 = 10000000
def benchmark(num_elements):
    # Benchmark push
    start = time.time()
    for i in range(num_elements):
        stack.append(i)
    end = time.time()
    print(f"Push {num_elements} phan tu mat: {end - start:.4f} giay"
        )
    # Benchmark pop
    start = time.time()
    for i in range(num_elements):
        stack.pop()
    end = time.time()
    print(f"Pop {num_elements} phan tu mat: {end - start:.4f} giay")
benchmark(N_1)
benchmark(N_2)
```

Kết quả chạy chương trình:

```
Push 1000000 phan tu mat: 0.0530 giay
Pop 1000000 phan tu mat: 0.0520 giay
Push 10000000 phan tu mat: 0.5318 giay
Pop 10000000 phan tu mat: 0.5179 giay
```

Nhận xét:

- Cả push và pop đều rất nhanh (xấp xỉ 0.05 giây cho 1 triệu phần tử và xấp xỉ 0.5 giây cho 10 triệu phần tử).
- Kết quả phù hợp với phân tích lý thuyết: trung bình $O(1)$ cho cả hai thao tác.
- Ưu điểm của đánh giá thực nghiệm: cung cấp số liệu cụ thể, dễ hình dung.

Benchmark với C++

```cpp
#include <iostream>
#include <stack>
#include <vector>
#include <chrono>
using namespace std;
using namespace std::chrono;
void benchmark(int N) {
    stack<int, vector<int>> s;
    // Push benchmark
    auto start = high_resolution_clock::now();
    for (int i = 0; i < N; i++) s.push(i);
    auto end = high_resolution_clock::now();
    cout << "Push " << N << " phan tu mat: "
         << duration<double>(end - start).count() << " giay\n";
    // Pop benchmark
    start = high_resolution_clock::now();
    for (int i = 0; i < N; i++) s.pop();
    end = high_resolution_clock::now();
    cout << "Pop " << N << " phan tu mat: "
         << duration<double>(end - start).count() << " giay\n";
}
```

```cpp
int main() {
    benchmark(1000000);   // 1 trieu phan tu
    benchmark(10000000);  // 10 trieu phan tu
    return 0;
}
```

Kết quả chạy chương trình:

```
Push 1000000 phan tu mat: 0.0090237 giay
Pop 1000000 phan tu mat: 0.017885 giay
Push 10000000 phan tu mat: 0.0955453 giay
Pop 10000000 phan tu mat: 0.161032 giay
```

Nhận xét:

- C++ cho kết quả nhanh hơn Python nhiều lần nhờ biên dịch tối ưu và quản lý bộ nhớ tĩnh hiệu quả (Python push 10000000 phần tử mất: 0.5318 giây trong khi C++ push 10000000 phần tử mất: 0.0955453 giây)
- Tuỳ mục tiêu (hiệu năng hoặc dễ dùng v.v) mà lựa chọn ngôn ngữ để thực hiện dự án liên quan.

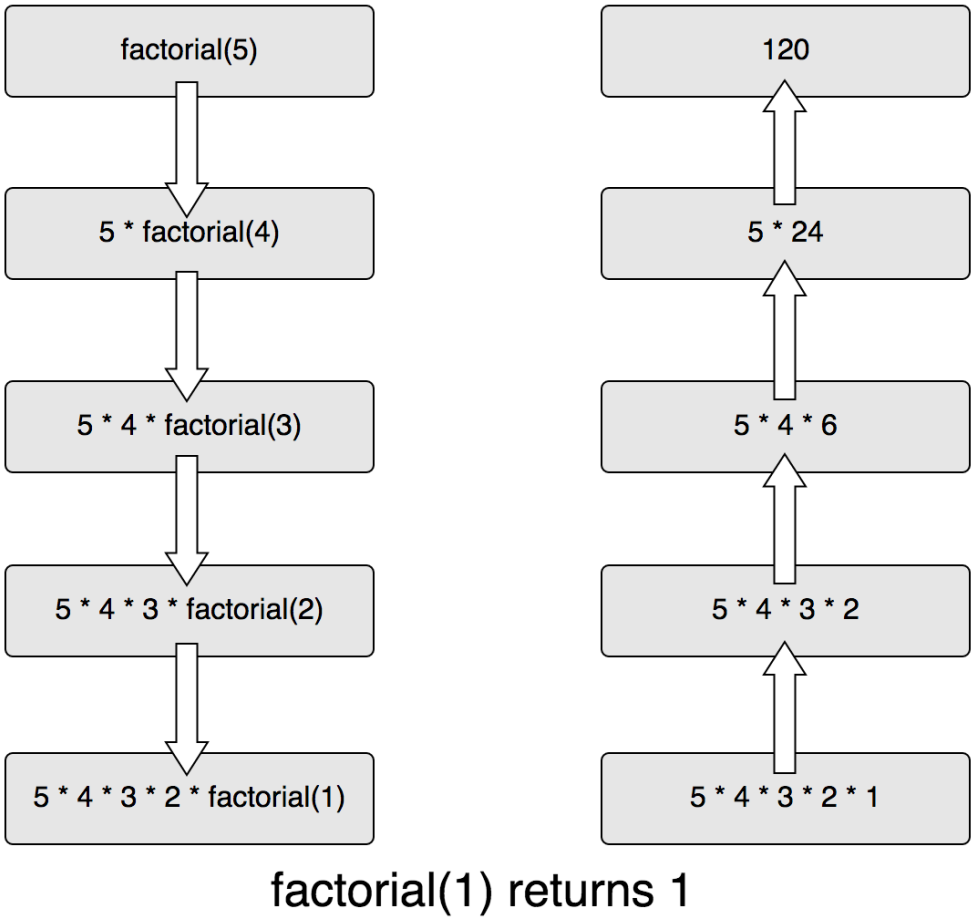
2.2 Ứng dụng thực tiễn

Mặc dù cấu trúc ngăn xếp (stack) có cách triển khai đơn giản, phạm vi ứng dụng của nó lại vô cùng đa dạng và hiệu quả nhờ nguyên lý LIFO (Last In, First Out) cùng các thao tác cơ bản như push và pop. Trong báo cáo này, nhóm em đã minh họa các ứng dụng tiêu biểu của stack thông qua một ứng dụng web demo được phát triển bằng Python. Liên kết truy trang web tại: https://demo-stack.streamlit.app/.

2.2.1 Quản lý hàm đệ quy

Trong quá trình thực thi chương trình, hệ thống runtime sử dụng cấu trúc dữ liệu ngăn xếp để quản lý trạng thái của các hàm đang được thực hiện. Cơ chế này đóng vai trò then chốt trong việc hỗ trợ các hàm đệ quy hoạt động một cách hiệu quả.

Khi một hàm được gọi, hệ thống tạo ra một *frame* (khung ngăn xếp) mới và đẩy nó vào Call Stack. Mỗi frame chứa các thông tin cần thiết như:



Hình 2.5: Tính giai thừa bằng phương pháp đệ quy

- Các tham số truyền vào,
- Các biến cục bộ của hàm,
- Địa chỉ trả về (return address) để sau khi hàm kết thúc, chương trình có thể tiếp tục tại vị trí gọi.

Sau khi hàm hoàn thành xử lý, khung tương ứng sẽ được pop khỏi Call Stack, và chương trình tiếp tục thực thi tại địa chỉ trả về đã lưu trước đó. Với cơ chế này, mỗi lời gọi đệ quy đều được lưu trữ độc lập trong một frame riêng biệt, giúp tránh tình trạng xung đột trạng thái giữa các lần gọi lồng nhau. Tuy nhiên, nếu độ sâu của đệ quy quá lớn và vượt quá giới hạn dung lượng của Call Stack, hệ thống sẽ phát sinh lỗi **Stack Overflow**, dẫn đến chương trình bị sập.

Quá trình tính toán giai thừa bằng phương pháp đệ quy được minh họa rõ ràng trong Hình 2.5. Như có thể thấy, quá trình này diễn ra theo hai giai đoạn chính: **đi xuống (descent)** và **đi lên (ascent)**. Trong giai đoạn đi xuống, các lời gọi hàm lồng nhau được thực hiện liên tiếp:

- Bắt đầu từ lời gọi `factorial(5)`, chương trình lần lượt gọi đệ quy xuống các giá trị nhỏ hơn: `factorial(4)` → `factorial(3)` → `factorial(2)` → `factorial(1)`.
- Mỗi lời gọi mới được thêm vào Call Stack, tạo thành một chuỗi các khung trạng thái (frames) chồng lên nhau để phản ánh đặc trưng LIFO của ngăn xếp.

Khi đạt đến điều kiện dừng (`factorial(1)` = 1), hệ thống bắt đầu giai đoạn đi lên. Các kết quả được trả về lần lượt từ dưới lên trên:

- Dầu tiên, `factorial(1)` trả về 1. Sau đó, kết quả này được dùng để tính tiếp: `factorial(2)` = $2 \times 1 = 2$ → `factorial(3)` = $3 \times 2 = 6$ → `factorial(4)` = $4 \times 6 = 24$.
- Cuối cùng, `factorial(5)` = $5 \times 24 = 120$.

Quá trình này cho thấy rõ vai trò của ngăn xếp trong việc lưu trữ trạng thái tạm thời và đảm bảo thứ tự trả về đúng đắn, nhờ nguyên lý LIFO. Mỗi khi một hàm hoàn thành, nó được **pop** khỏi stack và kết quả được truyền ngược lại cho hàm gọi trước đó.

2.2.2 Chuyển đổi và đánh giá biểu thức

Khái niệm

- **Biểu thức Infix**: Là cách viết thông thường, toán tử nằm *giữa* các toán hạng. Ví dụ:

$$A + B * C$$

- **Biểu thức Postfix (Reverse Polish Notation - RPN)**: Toán tử đặt *sau* các toán hạng. Ví dụ:

$$A\ B\ C * +$$

- **Biểu thức Prefix**: Toán tử đặt *trước* các toán hạng. Ví dụ:

$$+A * BC$$

Vấn đề: Trong biểu thức Infix, việc xác định thứ tự tính toán phụ thuộc vào **độ ưu tiên toán tử** và **dấu ngoặc**.

- $A + B * C \Rightarrow$ phải tính $B * C$ trước.

- $(A + B) * C \Rightarrow$ phải tính $A + B$ trước.

Máy tính không thể đọc và nhìn các biểu thức như con người, vì vậy ta cần một cách chuyển đổi biểu thức Infix sang Postfix hoặc Prefix để việc tính toán trở nên đơn giản và nhất quán.

Thuật toán Shunting-yard: Để giải quyết việc này, nhà toán học **Edsger Dijkstra** đưa ra ý tưởng chính là dùng **ngăn xếp (Stack)** để xử lý toán tử.

Nguyên tắc hoạt động:

1. Đọc biểu thức Infix từ trái sang phải.
2. Nếu gặp **toán hạng** (A, B, C, số, biến...) $\Rightarrow$ đưa vào kết quả ngay.
3. Nếu gặp **toán tử** (+, -, *, /...):
   - So sánh độ ưu tiên với toán tử trên đỉnh Stack.
   - Nếu toán tử trong Stack có ưu tiên cao hơn hoặc bằng $\Rightarrow$ lấy nó ra kết quả trước, rồi mới đẩy toán tử mới vào.
   - Nếu toán tử trong Stack có ưu tiên thấp hơn $\Rightarrow$ đẩy toán tử mới vào Stack.
4. Nếu gặp **dấu ngoặc mở "("** $\Rightarrow$ đẩy vào Stack.
5. Nếu gặp **dấu ngoặc đóng ")"** $\Rightarrow$ lấy toán tử trong Stack ra kết quả cho đến khi gặp dấu ngoặc mở.
6. Sau khi đọc hết biểu thức $\Rightarrow$ lấy toàn bộ toán tử còn lại trong Stack ra kết quả.

Ví dụ minh họa:

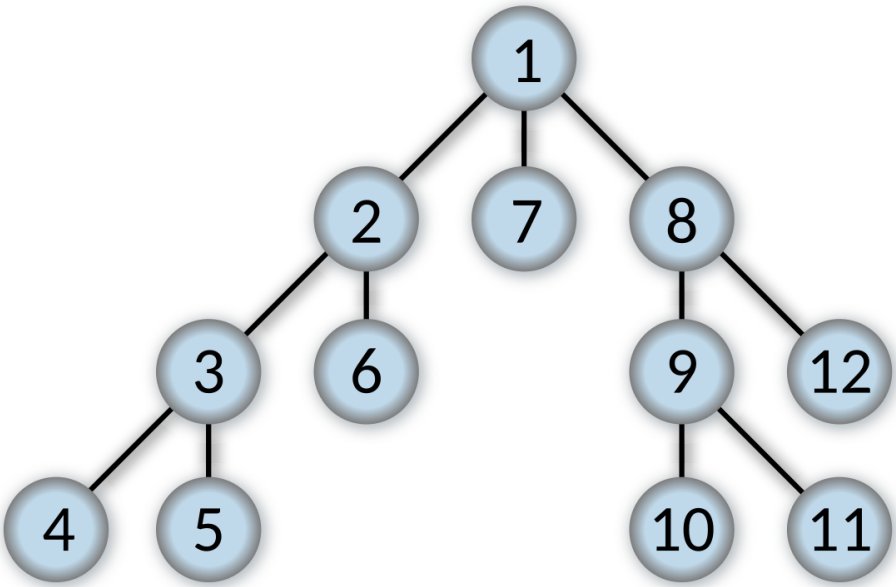
$$A + B * C$$

1. Đọc $A \Rightarrow$ kết quả: A
2. Đọc $+ \Rightarrow$ Stack: {+}
3. Đọc $B \Rightarrow$ kết quả: $A\ B$
4. Đọc $*$: vì $*$ ưu tiên cao hơn $+$, nên đẩy vào Stack $\Rightarrow$ Stack: {+, *}
5. Đọc $C \Rightarrow$ kết quả: $A\ B\ C$
6. Hết biểu thức $\Rightarrow$ lấy toàn bộ toán tử trong Stack ra: trước $*$, sau đó $+$.

→ **Kết quả Postfix:** A B C * +

→ **Kết quả Prefix:** + A * B C

2.2.3 Thuật toán DFS và Backtracking



Hình 2.6: Minh hoạ thuật toán DFS trên cây

Khái niệm:

- **DFS (Depth-First Search):** Đây là một thuật toán duyệt đồ thị hoặc cây, bắt đầu từ một đỉnh (hoặc nút) ban đầu, sau đó đi sâu nhất có thể theo từng nhánh trước khi quay lui để tiếp tục khám phá các nhánh khác.
  - DFS có thể cài đặt bằng *đệ quy*, khi đó chương trình tận dụng **Call Stack** của hệ thống. (Hoặc cài đặt bằng một *Stack tường minh* tự định nghĩa).
  - Trong mỗi bước, DFS luôn chọn đỉnh kế tiếp từ phần tử trên cùng của Stack để mở rộng.
- **Backtracking:** Đây là một kỹ thuật dùng để tìm kiếm lời giải trong không gian các cấu hình khả dĩ, bằng cách thử từng bước một, và nếu phát hiện hướng đang đi không thể dẫn đến lời giải, thì quay lại bước trước để thử hướng khác.
  - Có thể coi Backtracking là một dạng tổng quát của DFS, bởi nó cũng duyệt theo chiều sâu nhưng có thêm bước kiểm tra ràng buộc.

- Khi gặp ngõ cụt (hướng đi không hợp lệ), chương trình sẽ **pop** trạng thái gần nhất ra khỏi Stack và quay lại để thử một nhánh khác.
- Kỹ thuật này thường dùng trong các bài toán như: giải Sudoku, tìm đường trong mê cung, liệt kê tổ hợp/hoán vị thoả mãn điều kiện.

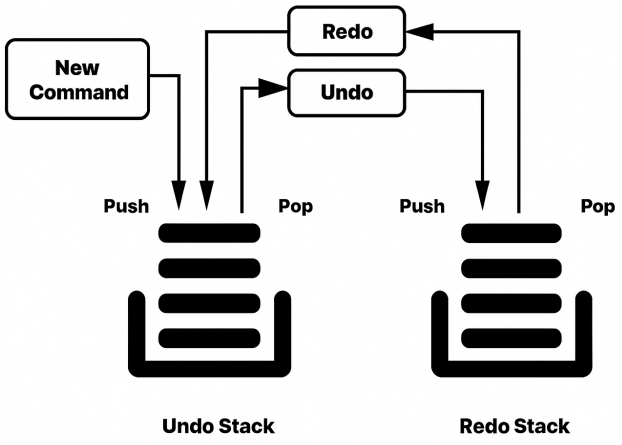
So sánh DFS và Backtracking

- **DFS:** Chủ yếu dùng để duyệt hoặc tìm kiếm trong đồ thị/cây, không nhất thiết kiểm tra ràng buộc.
- **Backtracking:** Dựa trên nguyên tắc DFS nhưng bổ sung thêm điều kiện kiểm tra và loại bỏ nhánh sai, nhờ vậy có thể giải quyết các bài toán ràng buộc phức tạp.

2.2.4 Ứng dụng trong phần mềm và đời sống

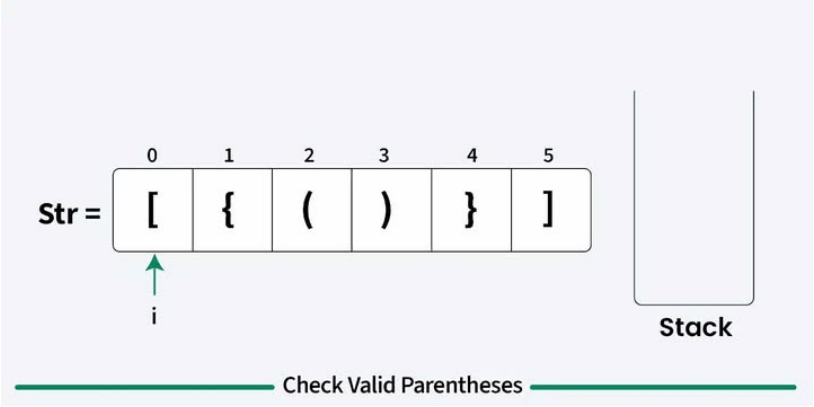
Stack không chỉ xuất hiện trong lý thuyết mà còn được ứng dụng rất nhiều trong các chương trình và công cụ quen thuộc hằng ngày:

- **Chức năng Undo/Redo:** Trong các phần mềm soạn thảo văn bản hoặc chỉnh sửa hình ảnh, hệ thống thường duy trì hai ngăn xếp.
  - Mỗi thao tác mới được `push` vào Stack **Undo**.
  - Khi người dùng chọn *Undo*, thao tác trên cùng được `pop` khỏi Undo và đồng thời `push` sang Stack **Redo**.
  - Khi chọn *Redo*, hệ thống thực hiện quá trình ngược lại.



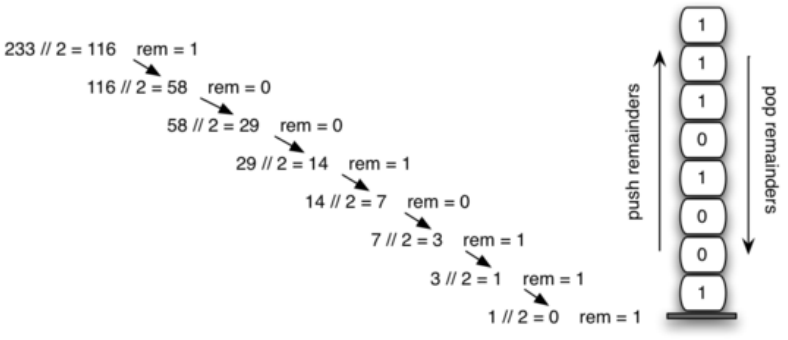
Hình 2.7: Undo / Redo bằng Stack

- **Lịch sử duyệt web:** Trình duyệt sử dụng hai Stack để quản lý nút *Back* và *Forward*.
  - Khi mở trang mới, URL hiện tại được `push` vào Stack "Back".
  - Khi nhấn *Back*, URL bị `pop` ra và đồng thời `push` sang Stack "Forward".
  - Nhờ đó, người dùng có thể quay lại hoặc đi tới dễ dàng.
- **Kiểm tra dấu ngoặc trong biểu thức:** Khi duyệt một chuỗi ký tự:
  - Gặp dấu ngoặc mở "(" thì `push` vào Stack.
  - Gặp dấu ngoặc đóng ")" thì `pop` để kiểm tra xem có khớp với dấu mở gần nhất hay không.
  - Cơ chế LIFO đảm bảo đúng thứ tự lồng nhau.



Hình 2.8: Kiểm tra ngoặc bằng Stack

- **Chuyển đổi hệ cơ số:** Khi đổi một số thập phân sang hệ cơ số khác:
  - Lần lượt chia số cho cơ số mới và `push` các số dư vào Stack.
  - Sau đó, `pop` các số dư ra theo thứ tự ngược lại để thu được kết quả cuối cùng.



Hình 2.9: Chuyển đổi cơ số 10 sang cơ số 2

2.2.5 Ứng dụng nâng cao

Ngoài các ứng dụng quen thuộc, Stack còn được sử dụng trong nhiều lĩnh vực hiện đại:

- **Xử lý ngôn ngữ tự nhiên (NLP):** Thuật toán *shift-reduce parsing* dựa trên Stack để xây dựng cây cú pháp cho câu. Một số mô hình học sâu như **Stack-LSTM**, **Neural Stack** kết hợp cơ chế Stack với mạng nơ-ron để học ngôn ngữ có cấu trúc.
- **Machine Learning:** Trong giải tích ngược (reverse-mode autodiff) và thuật toán lan truyền ngược (backpropagation), Stack (hay *tape*) được dùng để lưu các phép tính. Sau đó, ta duyệt ngược Stack để tính đạo hàm và gradient cho mô hình.
- **Thị giác máy tính:** Các thuật toán xử lý ảnh như *flood-fill*, *region-growing* hoặc tìm thành phần liên thông đều có thể được triển khai bằng DFS sử dụng Stack để mở rộng vùng ảnh.

2.2.6 Minh họa bằng các ngôn ngữ lập trình phổ biến

Các đoạn code được minh họa dưới đây bao gồm các ngôn ngữ C++, Python, C# nhằm thể hiện khả năng cài đặt và sử dụng stack của các ngôn ngữ đó.

2.2.6.1 C++

C++ với mảng và thư viện `std::stack`

```cpp
#include <iostream>
#include <stack> // stack in STL
using namespace std;
int main() {
    stack<int> s;
    s.push(10); // push elements
    s.push(20);
    s.push(30);
    cout << s.top() << " popped\n"; // show and pop top
    s.pop();
    cout << "Top element is:" << s.top() << "\n"; // peek top
    cout << "Elements present in stack: ";
    while (!s.empty()) {
        cout << s.top() << " "; // print and pop
```

```
        s.pop();
    }
    cout << "\n";
}
```

Kết quả chạy chương trình:

```
30 popped from stack
Top element is: 20
Elements present in stack: 20 10
```

Chú thích:

- Ba phần tử 10, 20, 30 được lần lượt `push` vào stack (theo thứ tự LIFO).
- Lệnh `s.top()` trả về 30, sau đó `s.pop()` loại bỏ nó.
- Khi gọi lại `s.top()`, giá trị 20 xuất hiện vì nó đang ở đỉnh.
- Vòng lặp `while (!s.empty())` duyệt qua stack: lần lượt in ra 20 rồi 10, đúng theo nguyên tắc LIFO.

2.2.6.2 Python

Python với `list`

```
# Stack implementation using 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)
```

Kết quả chạy chương trình:

```
pushed item: 1
pushed item: 2
pushed item: 3
popped item: 3
stack after popping: [1, 2]
```

Python với `collections.deque`

```
# Stack implementation using 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

# Test
stack = create_stack()
push(stack, 1)
push(stack, 2)
push(stack, 3)
print("popped item:", pop(stack))
print("stack after popping:", stack)
```

Kết quả chạy chương trình:

```
pushed item: 1
pushed item: 2
pushed item: 3
popped item: 3
stack after popping: deque([1, 2])
```

Chú thích:

- Với `list`, thao tác `append` và `pop` tương ứng với push và pop của Stack, nhưng khi số phần tử lớn có thể tốn chi phí mở rộng mảng.
- `deque` được tối ưu cho thêm/xóa ở cả hai đầu với độ phức tạp $O(1)$ ổn định, thích hợp cho Stack/Queue lớn.
- Trong thực tế, `deque` thường được khuyến nghị hơn `list` nếu hiệu năng quan trọng.

KẾT LUẬN

3.1 Các kết quả đạt được

Thông qua quá trình nghiên cứu và phân tích có hệ thống, báo cáo đã làm rõ các khái niệm cốt lõi về cấu trúc dữ liệu ngăn xếp, bao gồm nguyên lý LIFO (Last In First Out), các đặc trưng quan trọng, tập hợp phép toán cơ bản và những phương thức cài đặt điển hình. Dựa trên nền tảng lý thuyết này, nghiên cứu tiến hành phân tích độ phức tạp thuật toán và khảo sát sâu các ứng dụng thực tiễn như quản lý ngăn xếp lời gọi hàm đệ quy, thuật toán chuyển đổi cơ số, đánh giá biểu thức toán học, cùng với một số ứng dụng phổ biến trong khoa học máy tính.

Bên cạnh những đóng góp về mặt nội dung, quá trình thực hiện báo cáo còn tạo cơ hội cho các thành viên trong nhóm mở rộng kiến thức chuyên môn, phát triển kỹ năng hợp tác nghiên cứu và nâng cao chất lượng đóng góp cá nhân vào thành quả chung.

3.2 Hướng phát triển trong tương lai

Stack là một cấu trúc dữ liệu quan trọng, giữ vai trò nền tảng trong lập trình và khoa học máy tính. Với nguyên lý LIFO, stack cho phép quản lý dữ liệu theo cách mà phần tử được đưa vào sau cùng sẽ được lấy ra trước. Cơ chế này phù hợp cho việc quản lý lời gọi hàm, nhờ đó lập trình viên có thể xây dựng chương trình đệ quy, gọi hàm lồng nhau và quản lý biến cục bộ một cách hiệu quả. Bên cạnh đó, stack còn được ứng dụng rộng rãi trong xử lý biểu thức toán học, thực hiện các thuật toán duyệt dữ liệu như DFS, cũng như triển khai các tính năng tiện ích trong ứng dụng phần mềm, chẳng hạn như undo/redo. Có thể nói, stack không chỉ là một cấu trúc dữ liệu cơ bản mà còn là công cụ không thể thiếu, góp phần hình thành cơ chế vận hành của cả hệ thống máy tính lẫn các chương trình ứng dụng hiện đại.

Trong tương lai, việc nghiên cứu có thể tập trung vào tối ưu hóa stack trong các môi trường đặc thù, ví dụ như hệ thống nhúng, nơi tài nguyên bộ nhớ hạn chế. Ngoài ra, có thể mở rộng sang các biến thể của stack như deque (double-ended queue) hay stack động để giải quyết những bài toán phức tạp hơn. Một hướng quan trọng khác là kết hợp stack với các

cấu trúc dữ liệu khác (hàng đợi, cây, đồ thị) nhằm tạo ra những giải pháp linh hoạt, hỗ trợ hiệu quả cho các ứng dụng trí tuệ nhân tạo, xử lý dữ liệu lớn và lập trình song song.

TÀI LIỆU THAM KHẢO

[1] Wikibooks. (n.d.). *Data structures/Stacks and queues*. Wikimedia Foundation. https://en.wikibooks.org/wiki/Data_Structures/Stacks_and_Queues

[2] CMU 15-122 Course Staff. (n.d.). *Review: Stacks and queues* [Lecture slides]. Carnegie Mellon University. https://www.cs.cmu.edu/~15122/handouts/slides/review/09-stackqueue.pdf

[3] Quantrimang.com. (n.d.). *Cấu trúc dữ liệu ngăn xếp (Stack)*. https://quantrimang.com/cong-nghe/cau-truc-du-lieu-ngan-xep-stack-156375

[4] Vinh, L. V. (2013). *Giáo trình cấu trúc dữ liệu và giải thuật*. Đại học Quốc gia Thành phố Hồ Chí Minh.

[5] The Algorithms. (n.d.). *The Algorithms - Python* [Source code]. GitHub. https://github.com/TheAlgorithms/Python

PHỤ LỤC

Viết tắt	Giải thích
AI	Artificial Intelligence
IoT	Internet of Things
ADT	Abstract Data Type
DSU	Disjoint Set Union
LIFO	Last In, First Out
FIFO	First In, First Out
DFS	Depth First Search
BFS	Breadth First Search
CPU	Central Processing Unit
RAM	Random Access Memory
NLP	Natural Language Processing
LSTM	Long Short-Term Memory

Danh sách hình vẽ

Danh sách bảng