VÕ TIẾN
Thảo luận kiến thức CNTT trường BK về KHMT(CScience), KTMT(CEngineering)
https://www.facebook.com/groups/khmt.ktmt.cse.bku

Cấu Trúc Dữ Liệu và Giải Thuật (DSA)

DSA1 - HK242

Stack

Thảo luận kiến thức CNTT trường BK
về KHMT(CScience), KTMT(CEngineering)
https://www.facebook.com/groups/khmt.ktmt.cse.bku

TP. HỒ CHÍ MINH, THÁNG 1/2025

Mục lục

1 Khái niệm

Ngăn xếp (Stack)

Stack là một cấu trúc dữ liệu tuân theo nguyên tắc Last In First Out *(LIFO)*. Điều này có nghĩa là phần tử cuối cùng được thêm vào *stack* sẽ là phần tử đầu tiên được xóa.

Stack thường được sử dụng trong các trường hợp:

- Lưu trữ dữ liệu tạm thời như thông tin gọi hàm hoặc các cuộc gọi đệ quy.
- Đảo ngược chuỗi hoặc từ.
- Thông dịch biểu thức hậu tố hoặc trung tố.
- Giải các bài toán của lý thuyết đồ thị như giải thuật DFS.

Dữ liệu triển khai: *Stack* thường được triển khai bằng:

- **Mảng (Array):** Đơn giản và hiệu quả với kích thước cố định.
- **Danh sách liên kết (Linked List):** Linh hoạt với kích thước động.

Dưới đây là các thao tác cơ bản với độ phức tạp $O(1)$:

- **Push:** Thêm một phần tử vào đỉnh của stack.
- **Pop:** Xóa phần tử khỏi đỉnh của stack.
- **Peek:** Trả về phần tử ở đỉnh của stack mà không xóa nó.
- **IsEmpty:** Kiểm tra xem stack có trống không.
- **Size:** Trả về số lượng phần tử trong stack.

```cpp
// Abstract class for a Stack
template <typename T>
class AbstractStack {
public:
    // Destructor
    virtual ~AbstractStack() {}

    // Add an element to the top of the stack
    virtual void push(const T& value) = 0;

    // Remove the top element from the stack
    virtual void pop() = 0;

    // Get the top element of the stack without removing it
    virtual T top() const = 0;

    // Check if the stack is empty
    virtual bool isEmpty() const = 0;

    // Get the number of elements in the stack
    virtual size_t size() const = 0;
};
```

2 Hiện thực Stack bằng Array

Trong hiện thực bằng **Array**, stack được lưu trữ trong một mảng với các thao tác thêm (**push**), xóa (**pop**), và truy cập phần tử đỉnh (**top**) được thực hiện với độ phức tạp **O(1)**.

```
template <typename T>
class ArrayStack : AbstractStack
{
private:
    T* array;  //! ngăn xếp
    int Size;  //! kích thước hiện tại
    const int MAXSIZE; //! kích thước tối đa
public:
    // Constructor
    ArrayStack(int maxSize = 100)
        : MAXSIZE(maxSize), Size(0) {
        array = new T[MAXSIZE]; // Cấp phát bộ nhớ cho mảng
    }

    // Destructor
    ~ArrayStack() {
        delete[] array; // Giải phóng bộ nhớ
    }

    // function extend AbstractStack
}
```

Giải thích

- **T* array;** Mảng động dùng để lưu trữ các phần tử trong stack.
- **int Size;** Số lượng phần tử hiện tại trong stack.
- **const int MAXSIZE;** Kích thước tối đa của stack, được khởi tạo thông qua constructor.
- **Constructor ArrayStack(int maxSize = 100)** cấp phát bộ nhớ cho mảng **array** với kích thước tối đa là **maxSize**. Khởi tạo **Size = 0**.
- **Destructor ArrayStack()** giải phóng bộ nhớ đã cấp phát cho **array**

Các hàm được thừa kế từ class AbstractStack

1. `push(const T& value)`

```
// Thêm phần tử vào đỉnh stack
void push(const T& value) override {
    if (Size >= MAXSIZE) {
        throw std::overflow_error("Stack overflow: Cannot push, stack is
        ↪ full.");
    }
    array[Size++] = value; // Thêm phần tử vào vị trí hiện tại của stack và
    ↪ tăng kích thước
}
```

- **Chức năng:** Thêm một phần tử vào đỉnh stack.
- **Cách hoạt động:**

- Kiểm tra nếu stack đầy ($Size \geq MAXSIZE$), ném ngoại lệ `std::overflow_error`.
- Nếu còn chỗ trống, đặt phần tử vào `array[Size]` và tăng `Size`.

2. `pop()`

```cpp
// Xóa phần tử ở đỉnh stack
void pop() override {
    if (isEmpty()) {
        throw std::underflow_error("Stack underflow: Cannot pop, stack is
        ↪ empty.");
    }
    Size--; // Giảm kích thước stack, phần tử trên cùng sẽ bị "bỏ qua"
}
```

- **Chức năng:** Xóa phần tử ở đỉnh stack.
- **Cách hoạt động:**
  - Kiểm tra nếu stack rỗng ($Size == 0$), ném ngoại lệ `std::underflow_error`.
  - Giảm `Size` để bỏ qua phần tử trên cùng.

3. `top() const`

```cpp
// Trả về phần tử ở đỉnh stack mà không xóa nó
T top() const override {
    if (isEmpty()) {
        throw std::underflow_error("Stack underflow: Cannot access top, stack
        ↪ is empty.");
    }
    return array[Size - 1]; // Trả về phần tử trên cùng
}
```

- **Chức năng:** Trả về phần tử ở đỉnh stack mà không xóa nó.
- **Cách hoạt động:**
  - Kiểm tra nếu stack rỗng, ném ngoại lệ `std::underflow_error`.
  - Trả về `array[Size - 1]`.

4. `isEmpty() const`

```cpp
// Kiểm tra stack có rỗng không
bool isEmpty() const override {
    return Size == 0; // Stack rỗng khi Size = 0
}
```

- **Chức năng:** Kiểm tra xem stack có rỗng không.
- **Cách hoạt động:**
  - Trả về `true` nếu `Size == 0`, ngược lại trả về `false`.

5. `size() const`

```cpp
// Trả về số lượng phần tử trong stack
size_t size() const override {
    return Size; // Trả về số phần tử hiện tại trong stack
}
```

- **Chức năng:** Trả về số lượng phần tử hiện tại trong stack.
- **Cách hoạt động:**
  - Đơn giản trả về giá trị `Size`.

3 Hiện thực Stack bằng Linked List

Trong hiện thực bằng **Linked List**, stack được lưu trữ bằng danh sách liên kết với mỗi phần tử của stack là một `Node`. Việc thêm (**push**), xóa (**pop**), và truy cập phần tử đỉnh (**top**) đều được thực hiện với độ phức tạp **O(1)**.

```cpp
template <typename T>
class LinkedListStack : public AbstractStack<T>
{
private:
    struct Node {
        T data;  //! Dữ liệu của phần tử trong stack
        Node* next; //! Con trỏ đến phần tử tiếp theo

        Node(T value, Node* nextNode = nullptr)
            : data(value), next(nextNode) {}
    };

    Node* head;  //! Con trỏ đến phần tử đầu stack
    int Size;  //! Kích thước hiện tại của stack

public:
    // Constructor
    LinkedListStack() : head(nullptr), Size(0) {}

    // Destructor
    ~LinkedListStack() {
        while (!isEmpty()) {
            pop();
        }
    }

    // function extend AbstractStack
};
```

Giải thích

- **struct Node** Cấu trúc `Node` chứa dữ liệu và con trỏ trỏ đến phần tử tiếp theo.
- **Node* head;** Con trỏ trỏ đến phần tử đầu tiên trong stack (đỉnh stack).
- **int Size;** Số lượng phần tử hiện tại trong stack.
- **Constructor LinkedListStack()** Khởi tạo stack rỗng với `head = nullptr` và `Size = 0`.

- **Destructor ~LinkedListStack()** Giải phóng toàn bộ bộ nhớ khi stack bị hủy.

Các hàm được thừa kế từ class AbstractStack

1. `push(const T& value)`

```cpp
// Thêm phần tử vào đỉnh stack
void push(const T& value) override {
    head = new Node(value, head); // Thêm phần tử vào đầu danh sách liên kết
    Size++;
}
```

   - **Chức năng:** Thêm một phần tử vào đỉnh stack.
   - **Cách hoạt động:**
     - Cấp phát một `Node` mới, chứa `value`.
     - Trỏ `next` của Node mới đến phần tử trước đó (`head`).
     - Cập nhật `head` để trỏ đến Node mới.
     - Tăng `Size`.

2. `pop()`

```cpp
// Xóa phần tử ở đỉnh stack
void pop() override {
    if (isEmpty()) {
        throw std::underflow_error("Stack underflow: Cannot pop, stack is
        ↪ empty.");
    }
    Node* temp = head; // Lưu con trỏ của phần tử hiện tại
    head = head->next; // Cập nhật head đến phần tử tiếp theo
    delete temp; // Giải phóng bộ nhớ của phần tử cũ
    Size--;
}
```

   - **Chức năng:** Xóa phần tử ở đỉnh stack.
   - **Cách hoạt động:**
     - Nếu stack rỗng, ném ngoại lệ `std::underflow_error`.
     - Lưu con trỏ của `head` hiện tại.
     - Cập nhật `head` để trỏ đến phần tử tiếp theo.
     - Giải phóng bộ nhớ của phần tử cũ.
     - Giảm `Size`.

3. `top() const`

```cpp
// Trả về phần tử ở đỉnh stack mà không xóa nó
T top() const override {
    if (isEmpty()) {
        throw std::underflow_error("Stack underflow: Cannot access top, stack
        ↪ is empty.");
    }
```

```
    return head->data; // Trả về dữ liệu của phần tử đỉnh
}
```

- **Chức năng:** Trả về phần tử ở đỉnh stack mà không xóa nó.
- **Cách hoạt động:**
  - Nếu stack rỗng, ném ngoại lệ `std::underflow_error`.
  - Trả về giá trị của `head->data`.

4. `isEmpty() const`

```
// Kiểm tra stack có rỗng không
bool isEmpty() const override {
    return head == nullptr; // Stack rỗng khi head là nullptr
}
```

- **Chức năng:** Kiểm tra xem stack có rỗng không.
- **Cách hoạt động:**
  - Trả về `true` nếu `head == nullptr`, ngược lại trả về `false`.

5. `size() const`

```
// Trả về số lượng phần tử trong stack
size_t size() const override {
    return Size; // Trả về số phần tử hiện tại trong stack
}
```

- **Chức năng:** Trả về số lượng phần tử hiện tại trong stack.
- **Cách hoạt động:**
  - Đơn giản trả về giá trị `Size`.

4 So sánh Stack bằng Linked List và Array

Tiêu chí	**ArrayStack**	**LinkedListStack**
Bộ nhớ	Cần cấp phát trước (giới hạn kích thước)	Linh hoạt, không giới hạn số phần tử
Độ phức tạp của push/pop	$O(1)$	$O(1)$
Truy cập phần tử bất kỳ	$O(1)$ (dùng chỉ số)	$O(n)$ (duyệt qua danh sách)
Tốc độ thực thi	Nhanh hơn do truy cập trực tiếp	Chậm hơn do cấp phát động

Bảng 1: *So sánh Stack bằng Array và Linked List*