

Stack (ngăn xếp)

Stack (ngăn xếp)

Tác giả: Nguyễn Hà Duy - THPT Chuyên Hà Nội - Amsterdam

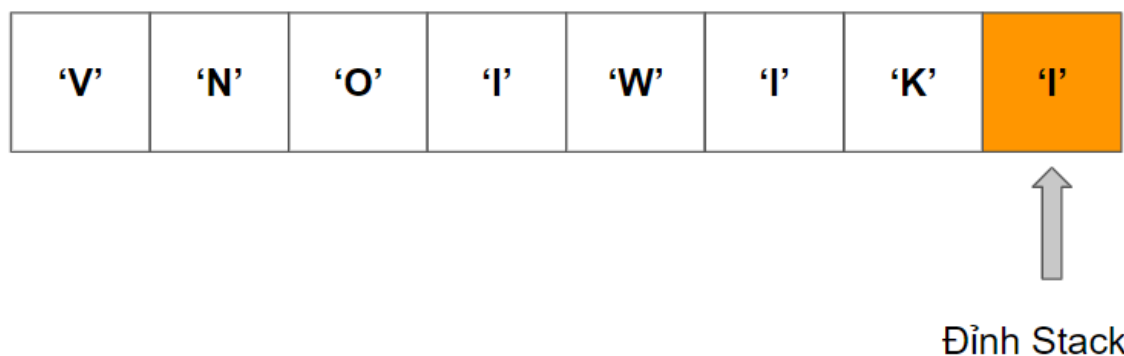
Reviewer: Hoàng Xuân Nhật

Giới thiệu

Stack là một danh sách được bổ sung 2 thao tác: **thêm một phần tử vào cuối danh sách**, và **loại bỏ một phần tử ở cuối danh sách**. Vị trí cuối của Stack được gọi là đỉnh (**top**).

Có thể hình dung Stack như một chồng sách. Việc đặt một quyển sách lên trên cùng chính là thao tác thêm phần tử, và lấy ra quyển sách ở trên đầu là thao tác loại bỏ phần tử. Như vậy, quyển sách được đặt vào sau cùng sẽ luôn được lấy ra trước tiên. Vì tính chất này, Stack còn được gọi là danh sách **LIFO** (Last In - First Out, hay vào sau - ra trước).

Hình ảnh minh họa cho Stack chứa các phần tử kiểu `char` :



Stack có khá nhiều ứng dụng trong lập trình thi đấu. Bài viết này sẽ xem xét các ứng dụng điển hình của Stack.

Cài đặt

Cài đặt thủ công

Ta có thể biểu diễn Stack bằng một mảng, cùng với biến `top` biểu diễn vị trí của phần tử nằm ở đỉnh Stack. Dưới đây là một cách cài đặt Stack chứa các phần tử thuộc kiểu `int` :

```

1  const int MAXN = 1e5 + 2;
2
3  int st[MAXN];
4  int top = 0;
5
6  void push(int x) // thêm x vào cuối Stack
7  {
8      ++top;
9      st[top] = x;
10 }
11
12 void pop() // loại bỏ phần tử ở cuối Stack
13 {
14     assert(top); // đảm bảo Stack đang chứa ít nhất 1 phần tử
15     --top;
16 }
17
18 int peek() // trả về giá trị của phần tử ở đỉnh Stack
19 {
20     return st[top];
21 }

```

Sử dụng thư viện chuẩn

Thư viện chuẩn của C++ cho phép ta sử dụng Stack qua kiểu dữ liệu `stack` trong header cùng tên. Các thao tác chính trên `stack` là:

- `push` : thêm phần tử vào cuối danh sách
- `top` : trả về giá trị phần tử ở cuối danh sách
- `pop` : loại bỏ phần tử ở cuối danh sách

Ngoài ra, `stack` cũng hỗ trợ các thao tác:

- `size` : trả về số phần tử hiện có trong stack
- `empty` : trả về trạng thái của stack (`true` nếu stack rỗng, `false` nếu stack có ít nhất 1 phần tử)

Ví dụ:

```

1  #include <iostream>
2  #include <stack>
3
4  using namespace std;
5
6

```

```

0 | int main()
7 | {
8 |     stack<int> st;
9 |     st.push(5); // thêm 5 vào stack
10 |    st.push(10); // thêm 10 vào stack
11 |    cout << st.top() << endl; // In ra 10
12 |    st.pop(); // loại bỏ phần tử ở cuối
13 |    cout << st.top() << endl; // In ra 5
14 |    return 0;
15 | }

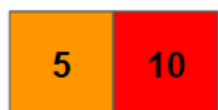
```



st.push(5)



st.push(10)



st.pop()

Chú thích*Vị trí đỉnh Stack**Vị trí của phần tử vừa bị loại bỏ*

Ngoài ra, ta có thể dùng `vector` để biểu diễn một Stack. Các hàm `push`, `top` và `pop` sẽ được thay bằng `push_back`, `pop_back` và `back` khi sử dụng `vector`.

Phân tích

Do Stack có thể được cài đặt bằng `vector` nên các thao tác trên Stack cũng có cùng độ phức tạp với `vector`.

Độ phức tạp thời gian

Các hàm `push`, `pop`, `top`, `size` và `empty` của Stack đều hoạt động trong $O(1)$. Hơn nữa, như ta đã thấy ở cách cài đặt thủ công, bản chất của Stack chính là mảng, nên tất cả các thao tác trên Stack đều hoạt động trong $O(1)$.

Độ phức tạp bộ nhớ


Độ phức tạp bộ nhớ của Stack là $O(N)$, với N là số phần tử được đưa vào Stack.

Ứng dụng

Sử dụng Stack để xử lý sâu

Bài toán 1

Cho chuỗi S chỉ gồm các số nguyên dương và các dấu $+$, $-$, \times , \div , trong S không có dấu khoảng trống. Bạn cần tính giá trị của biểu thức được biểu diễn bởi chuỗi đó.

Đây là bài toán dễ hơn của [Expression Parsing](#) .

Cách giải

Vấn đề chính của bài toán là các toán tử $+$, $-$, \times và \div không có cùng độ ưu tiên. Cụ thể, ta cần tính kết quả của các cụm dấu \times và \div trước, do nhân và chia có độ ưu tiên cao hơn cộng và trừ.

Xét bài toán đơn giản hơn: trong chuỗi S chỉ có các dấu $+$ và $-$. Rõ ràng, do $+$ và $-$ có cùng độ ưu tiên nên ta có thể xử lý bài toán bằng cách duyệt các toán hạng và toán tử từ trái sang phải và tính trực tiếp kết quả sau mỗi phép cộng hoặc trừ. Ta có thể cài đặt như sau: duy trì một danh sách chứa số (toán hạng) và một danh sách chứa toán tử. Duyệt chuỗi từ trái qua phải, nếu ký tự đang xét là chữ số thì ta đẩy nó vào danh sách chứa số. Nếu đó là toán tử, ta đẩy ký tự vào danh sách chứa toán tử. Cuối cùng, ta lần lượt tính kết quả dựa vào danh sách toán hạng và toán tử đã xây dựng.

```
// xử lý toán tử và cập nhật trực tiếp vào mảng val
void process_op(vector<int>& val, char op)
{
    // l, r là 2 toán hạng, op là dấu giữa chúng
    int r = val.back(); val.pop_back();
    int l = val.back(); val.pop_back();
    switch(op)
    {
        case '+': val.push_back(l + r); break;
        case '-': val.push_back(l - r); break;
    }
}

// tính giá trị của biểu thức biểu diễn bởi s
int evaluate(string s)
{
    vector<int> val;
    vector<char> op;
    for (int i = 0; i < (int)s.size(); ++i)
    {
        if (isdigit(s[i]))
        {
            int number = 0; // giá trị của toán hạng đang xét
            while (i < (int)s.size() && isdigit(s[i]))
            {
                number = number * 10 + s[i] - '0';
                ++i;
            }
            val.push_back(number);
            --i; // cẩn thận với index!
        }
        else
```

```

33     {
34         if (!op.empty())
35         {
36             process_op(val, op.back()); // xử lý biểu thức từ trái sang phải
37             op.pop_back();
38         }
39         op.push_back(s[i]);
40     }
41 }
42
43 if (!op.empty())
44 {
45     process_op(val, op.back());
46     op.pop_back();
47 }
48
49 return val.back();
50 }

```

Do các toán tử có cùng độ ưu tiên, ta có thể xử lý chúng lần lượt theo danh sách đã xây dựng. Tuy nhiên, nếu danh sách *op* có các toán tử khác độ ưu tiên, thì ta vẫn có thể xử lý bài toán bằng việc luôn giữ cho danh sách toán tử *op* chứa các toán tử có độ ưu tiên *tăng dần*, sau đó xử lý danh sách toán tử từ dưới lên để đảm bảo các dấu có độ ưu tiên lớn hơn luôn được xử lý trước.

Trong bài toán gốc, chuỗi *S* còn chứa ký tự \times và \div . Ta cần xử lý danh sách toán tử sao cho các phép tính có độ ưu tiên lớn hơn luôn được thực hiện trước.

Ta định nghĩa một thứ tự ưu tiên cho các dấu:

```

1 // Trả về 2 nếu là nhân hoặc chia, 1 nếu là cộng hoặc trừ
2 int priority(char op)
3 {
4     if (op == '+' || op == '-') return 1;
5     else return 2;
6 }

```

Giống với bài toán đơn giản, ta vẫn duyệt chuỗi *S* từ trái sang phải. Khi gặp toán hạng, ta vẫn đẩy chúng vào danh sách. Tuy nhiên, khi gặp toán tử, ta cần xem xét thứ tự ưu tiên của toán tử ngay trước nó. Nếu toán tử trước có độ ưu tiên lớn hơn hoặc bằng toán tử đang xét, ta cần phải xử lý nó trước, và loại bỏ toán tử đứng trước đó khỏi danh sách. Ta sẽ lặp lại việc xử lý toán tử đứng trước này cho đến khi nó có độ ưu tiên không lớn hơn toán tử đang xét. Điều này đảm bảo phép tính sẽ luôn được thực hiện theo mức độ ưu tiên.

Để ý rằng bản chất của danh sách ta sử dụng chính là Stack. Việc thêm toán tử/toán hạng vào danh sách chính là thao tác **push**; cách ta xử lý toán tử gợi đến việc **top** và **pop** do trong các bước của lời giải, ta chỉ cần quan tâm tới những phần tử ở cuối danh sách. Lời giải sử dụng **vector** để biểu diễn Stack.

```

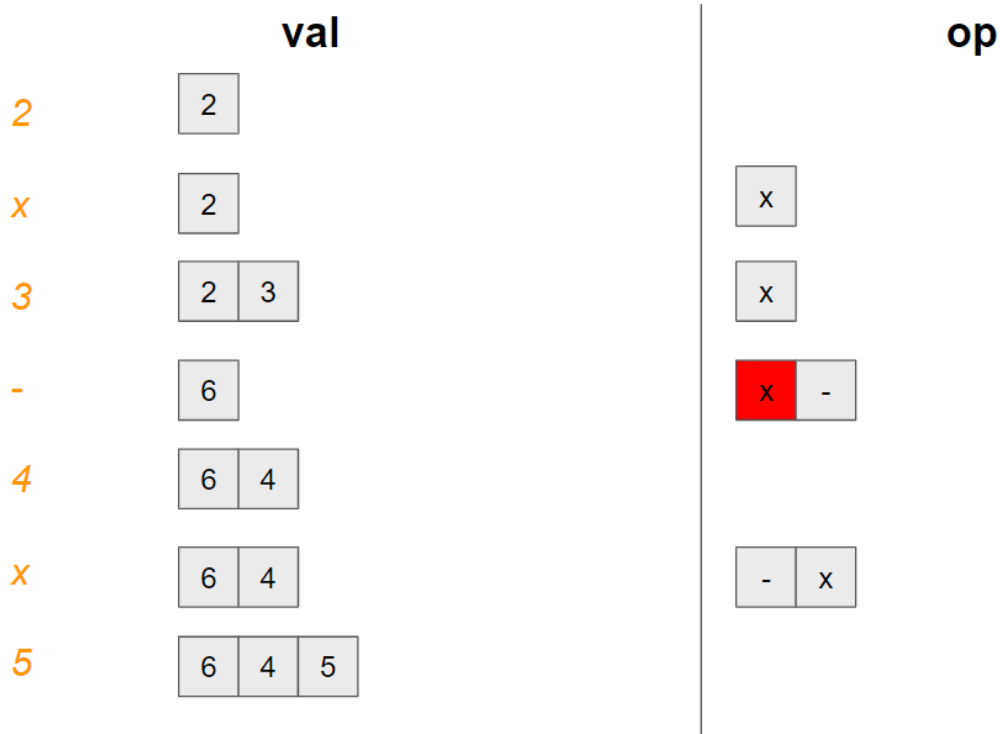
1  void process_op(vector<int>& val, char op)
2  {
3      int r = val.back(); val.pop_back();
4      int l = val.back(); val.pop_back();
5      switch(op)
6      {
7          case '+': val.push_back(l + r); break;
8          case '-': val.push_back(l - r); break;
9          case '*': val.push_back(l * r); break;
10         case '/': val.push_back(l / r); break;
11     }
12 }
13
14 int evaluate(string s)
15 {
16     // ...
17     for (int i = 0; i < (int)s.size(); ++i)
18     {
19         if (isdigit(s[i]))
20         {
21             // ...
22         }
23         else
24         {
25             char cur_op = s[i]; // toán tử hiện tại
26             while (!op.empty() && priority(op.back()) >= priority(cur_op))
27                 // xử lý các toán tử ngay trước nó có độ ưu tiên bằng hoặc lớn hơn
28                 // chú ý rằng nếu thay dấu >= bằng dấu > thì đáp án
29                 {
30                     process_op(val, op.back());
31                     op.pop_back();
32                 }
33             op.push_back(cur_op);
34         }
35     }
36     // do danh sách toán tử có độ ưu tiên tăng dần nên ta có thể xử lý
37     // lần lượt như trong bài toán chỉ có + và -
38     // chú ý là ta xử lý ngược từ đáy về đầu, nên các toán tử được xử lý
39     // sẽ tạo thành một dãy giảm theo thứ tự ưu tiên
40     while (!op.empty())
41     {
42         process_op(val, op.back());
43         op.pop_back();
44     }
45     return val.back();
46 }

```

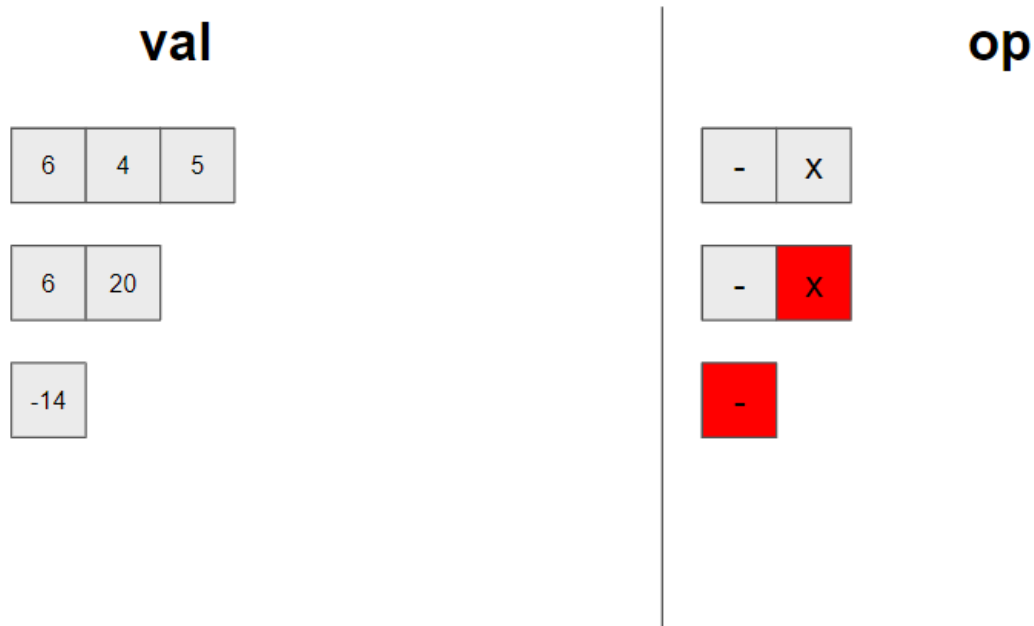
Xét ví dụ: $S = 2 \times 3 - 4 \times 5$.

Giá trị của *val* và *op* sau khi xử lý xâu *S*:

$$S = 2 \times 3 - 4 \times 5$$



Quá trình xử lý danh sách toán tử *op*:



Bài toán 2

Cho xâu *S* chỉ gồm ký tự (và). Bạn cần kiểm tra xem *S* có phải là dãy ngoặc đúng không.

Nếu *S* là dãy ngoặc đúng, với mỗi vị trí trong *S* bạn cần in ra vị trí của **dấu ngoặc tương ứng**.

Định nghĩa:

- Xâu rỗng là dãy ngoặc đúng

..... (..... (...)

..... i j ... k

Theo **tính chất 3**, dãy ngoặc từ vị trí $i + 1$ đến j cũng là dãy ngoặc đúng. Mà dãy ngoặc từ vị trí $i + 1$ đến j lại kết thúc bằng dấu mở ngoặc, trái với **tính chất 2**. Vậy giả thiết tồn tại một dấu ngoặc tương ứng với nhiều hơn một dấu ngoặc khác là sai.

Vậy ta có điều phải chứng minh.

Bổ đề 2

Dãy ngoặc đúng khi và chỉ khi số dấu ngoặc mở bằng số dấu ngoặc đóng, và trong mọi tiền tố của dãy, số dấu ngoặc mở không nhỏ hơn số dấu ngoặc đóng.

Chứng minh

- ▶ **Chiều thuận:** dãy ngoặc đúng có số dấu ngoặc mở bằng số dấu ngoặc đóng, và trong mọi tiền tố của dãy, số dấu ngoặc mở không nhỏ hơn số dấu ngoặc đóng. Theo **tính chất 2**, dãy ngoặc mở có số dấu đóng ngoặc bằng số dấu mở ngoặc. Mặt khác, nếu tồn tại một tiền tố của dãy ngoặc đúng có số dấu mở ngoặc nhỏ hơn số dấu đóng ngoặc thì rõ ràng tồn tại ít nhất 1 dấu đóng ngoặc không tương ứng với dấu mở ngoặc nào, trái với **bổ đề 1**. Vậy ta chứng minh được chiều thuận.
- ▶ **Chiều đảo:** trong mọi tiền tố của dãy, số dấu ngoặc mở không nhỏ hơn số dấu ngoặc đóng đảm bảo rằng mỗi dấu ngoặc đóng đều có dấu ngoặc mở tương ứng với nó. Hơn nữa, số dấu ngoặc mở bằng số dấu ngoặc đóng đảm bảo không có dấu ngoặc mở bị thừa ra (hay không tương ứng với dấu ngoặc nào) trong dãy ngoặc. Do đó, dãy có số dấu ngoặc mở bằng số dấu ngoặc đóng và trong mọi tiền tố của dãy, số dấu ngoặc mở không nhỏ hơn số dấu ngoặc đóng là dãy ngoặc đúng.

Hệ quả 1

Dãy ngoặc không đúng sẽ có số dấu ngoặc mở lớn hơn số ngoặc đóng, hoặc có một tiền tố mà số dấu ngoặc mở nhỏ hơn số dấu ngoặc đóng.

Cách giải

Cho một Stack chứa các phần tử kiểu `char`, đang ở trạng thái rỗng. Xét quá trình sau:

- ▶ Duyệt xâu S từ trái qua phải
- ▶ Nếu ký tự đang xét là dấu `(`, thêm nó vào Stack
- ▶ Nếu ký tự đang xét là dấu `)`, xét phần tử đang ở đỉnh Stack. Nếu đó là dấu `(`, ta tìm được một cặp dấu ngoặc tương ứng, và loại bỏ dấu ngoặc mở khỏi Stack. Nếu ngược lại, hoặc Stack rỗng, xâu S không phải dãy ngoặc đúng.
- ▶ Sau khi thực hiện quá trình, nếu Stack không rỗng thì S không phải dãy ngoặc đúng.

Trước hết, quá trình này sẽ luôn xác định xem xâu S có phải là dãy ngoặc đúng hay không. Nhắc lại, quá trình sẽ kết luận xâu S không phải dãy ngoặc đúng khi:

- ▶ Ký tự đang xét là dấu đóng ngoặc và Stack đang rỗng.
- ▶ Sau khi xử lý, Stack không rỗng.

Trường hợp 1 xảy ra khi xâu S có một tiền tố mà số dấu ngoặc đóng nhiều hơn số dấu ngoặc mở. Trường hợp 2 xảy ra khi xâu S có số dấu ngoặc mở nhiều hơn số dấu ngoặc đóng. Cả hai trường hợp trên đều là dấu hiệu của dãy ngoặc không đúng nên trong **hệ quả 1**. Vậy quá trình sẽ luôn xác định S là dãy đúng hay không.

Từ bây giờ, ta mặc định S là dãy ngoặc đúng, và chứng minh rằng việc sử dụng Stack sẽ **luôn chỉ ra các cặp dấu ngoặc tương ứng**.

Quan sát quá trình trên, ta có một nhận xét quan trọng: sau khi xử lý một dãy ngoặc đúng trên Stack st thì Stack sẽ giữ nguyên trạng thái. Điều này là đúng do mỗi dấu mở ngoặc được **push** vào Stack và **pop** khỏi Stack đúng một lần.

Theo **tính chất 4**, S có thể được tách thành tổng của các dãy ngoặc đúng cơ bản $s_1 + s_2 + \dots + s_k$. Theo **tính chất 5**, $s_i = (+ s' +)$, với s' là dãy ngoặc đúng, có thể là xâu rỗng. Vì sau khi xử lý các dãy s_1, s_2, \dots thì Stack sẽ trở về trạng thái rỗng, nên ta chỉ cần xem xét quá trình với dãy ngoặc đúng cơ bản s_i :

- Đẩy dấu ngoặc mở vào Stack
- Xử lý dãy ngoặc đúng s' để tìm các cặp dấu ngoặc tương ứng trong đó. Sau bước này, ta tìm được tất cả cặp dấu tương ứng trong s' .
- Xét dấu ngoặc đóng ở cuối s_i . Do sau khi xử lý dãy ngoặc đúng, Stack chỉ còn một phần tử là dấu ngoặc mở được thêm vào từ đầu, ta tìm được một cặp dấu ngoặc tương ứng, và xóa dấu ngoặc mở khỏi Stack
- Stack trở về trạng thái rỗng

Như vậy, quá trình sẽ xác định tất cả các cặp dấu ngoặc tương ứng.

Cài đặt

Ta có thể cài đặt bài toán đúng như mô tả của quá trình nêu trên:

```

1  stack<int> st;
2  vector<pair<int, int>> matches; // lưu các cặp dấu ngoặc tương ứng
3
4  bool solve(string s)
5  {
6      int n = (int)s.size();
7      for (int i = 0; i < n; ++i)
8      {
9          if (s[i] == '(') // chỉ đẩy dấu ngoặc mở vào Stack
10             st.push(i);
11         else
12         {
13             if (st.empty()) return false; // dãy không phải là dãy ngoặc đúng
14             matches.push_back({st.top(), i}); // tìm thấy một cặp tương ứng
15             st.pop();
16         }
17     }
18     if (!st.empty()) return false; // nếu Stack không rỗng thì dãy không đúng
19     return true;
20 }
```

Mở rộng

Bài toán 2 có thể được mở rộng thêm: dãy có thể có cả ngoặc vuông và ngoặc nhọn. Rõ ràng, ta có thể xử lý các loại ngoặc như cách ta làm với bài toán 2. Lưu ý duy nhất là ta cần phải kiểm soát thêm cả kiểu loại của dấu.

Minh họa cho quá trình với $S = "([{}])()"$:

| | | | | |
|---|---|---|--|--|
| (| | | | $S[1] = '(' \rightarrow$ đẩy '(' vào Stack |
| (| [| | | $S[2] = '[' \rightarrow$ đẩy '[' vào Stack |
| (| [| { | | $S[3] = '{' \rightarrow$ đẩy '{' vào Stack |
| (| [| } | | $S[4] = '}' \rightarrow$ do dấu ngoặc ở đỉnh Stack trùng kiểu với nó nên pop() ra khỏi Stack |
| (| [| | | $S[5] = ']' \rightarrow$ do dấu ngoặc ở đỉnh Stack trùng kiểu với nó nên pop() ra khỏi Stack |
| (| | | | $S[6] = ')' \rightarrow$ do dấu ngoặc ở đỉnh Stack trùng kiểu với nó nên pop() ra khỏi Stack |
| (| | | | $S[7] = '(' \rightarrow$ đẩy '(' vào Stack |
| (| | | | $S[8] = ')' \rightarrow$ do dấu ngoặc ở đỉnh Stack trùng kiểu với nó nên pop() ra khỏi Stack |

Sử dụng Stack để khử đệ quy

Vì tính chất **LIFO** của Stack, nó có thể được sử dụng để khử các hàm đệ quy. Trên thực tế, khi ta dùng hàm đệ quy, hệ thống sẽ tự động tạo một cấu trúc **LIFO** như Stack để chứa và thực hiện các lời gọi hàm.

Xét thuật toán DFS với cách cài đặt sử dụng đệ quy:

```
void dfs(int start)
{
    visited[start] = true; // đánh dấu đỉnh đã thăm
    for (int u : adj[start]) // với các đỉnh thuộc danh sách kề của đỉnh đang :
    {
        if (visited[u])
            continue;
        dfs(u); // đệ quy: thăm u
    }
}
```

```

    }
}

```


Ta có thể cài đặt DFS sử dụng Stack:


```

1  void dfs(int start)
2  {
3      stack<int> st;
4      st.push(start); // ta bắt đầu từ đỉnh "start"
5
6      while (!st.empty())
7      {
8          int v = st.top(); // thăm đỉnh v ở đỉnh Stack
9          st.pop(); // loại bỏ đỉnh v khỏi Stack do đã thăm
10         visited[v] = true; // đánh dấu v đã thăm
11
12         for (int u : adj[v]) // xét các đỉnh của đồ thị chung cạnh với v
13         {
14             if (visited[u]) // nếu đã thăm thì thôi
15                 continue;
16             st.push(u); // đẩy đỉnh u vào Stack thay vì gọi đệ quy đến u
17         }
18     }
19 }

```

Trong cách cài đặt DFS đệ quy, một nhánh đệ quy luôn phải được xử lý trọn vẹn trước khi ta gọi đệ quy đến nhánh khác. Nói cách khác, khi ta thăm một đỉnh v thì ta phải thăm toàn bộ những đỉnh kề với nó, và cứ tiếp tục như vậy đến khi không còn đỉnh nào thăm được nữa. Stack hoạt động trên nguyên lý tương tự: những đỉnh được đẩy vào Stack sau đều được xử lý trọn vẹn trước những đỉnh được đẩy vào sớm hơn. Do đó, lời giải trên là đúng đắn.

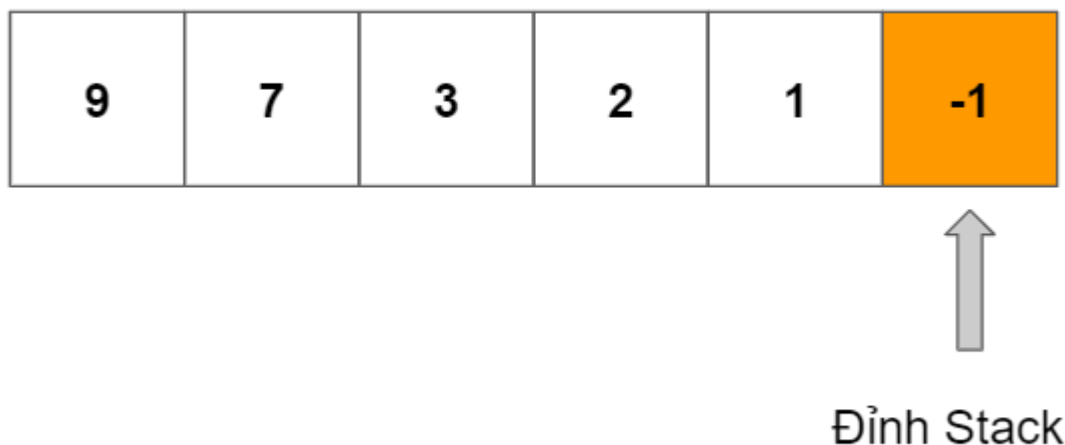
Việc dùng Stack để khử đệ quy có thể áp dụng với mọi hàm có tính chất đệ quy. Trên lý thuyết, độ phức tạp thời gian và bộ nhớ của 2 cách cài đặt dùng Stack và dùng đệ quy là như nhau, nhưng trên thực tế, cách cài đặt dùng Stack thường hiệu quả hơn về mặt bộ nhớ, do việc gọi đệ quy chịu ảnh hưởng bởi [Function Overhead](#) . Đối lại, cách cài đặt dùng Stack thường khiến code trở nên phức tạp và thiếu trực quan hơn.

Trong lập trình thi đấu, ta có thể sử dụng đệ quy thông thường trong hầu hết các trường hợp. Ta chỉ cần dùng Stack khi hàm đệ quy quá sâu và có nguy cơ bị [tràn bộ nhớ](#) . Theo kinh nghiệm của người viết, ta cần khử đệ quy khi hàm có thể đạt độ sâu khoảng 10^7 .

Stack đơn điệu

Stack đơn điệu là ngăn xếp mà các phần tử của nó xét từ đáy của Stack đến đỉnh Stack tạo thành một dãy số đơn điệu.

Hình ảnh minh họa cho một Stack đơn điệu giảm:



Bài toán

Cho mảng A có n phần tử a_1, a_2, \dots, a_n , $n \leq 10^6$. Với mỗi i từ 1 đến n ta cần tìm j sao cho $a_j > a_i$, và $|i - j|$ nhỏ nhất. Nếu không tồn tại j , in ra -1 .

Nhận xét

Ta sẽ bài toán đơn giản hơn: với mỗi i ta chỉ cần tìm j thỏa mãn điều kiện gốc, mà $j < i$. Rõ ràng, nếu ta giải được bài toán này thì bài toán gốc cũng có thể dễ dàng giải được, vì nếu $j > i$ thì ta có thể duyệt ngược lại mảng A , đưa bài toán về dạng đơn giản như đã nói.

Do $n \leq 10^6$ nên cách giải hồn nhiên: với mỗi i ta lại xét j từ 1 đến n là chưa đủ để giải quyết bài toán, do độ phức tạp thời gian lên tới $O(n^2)$.

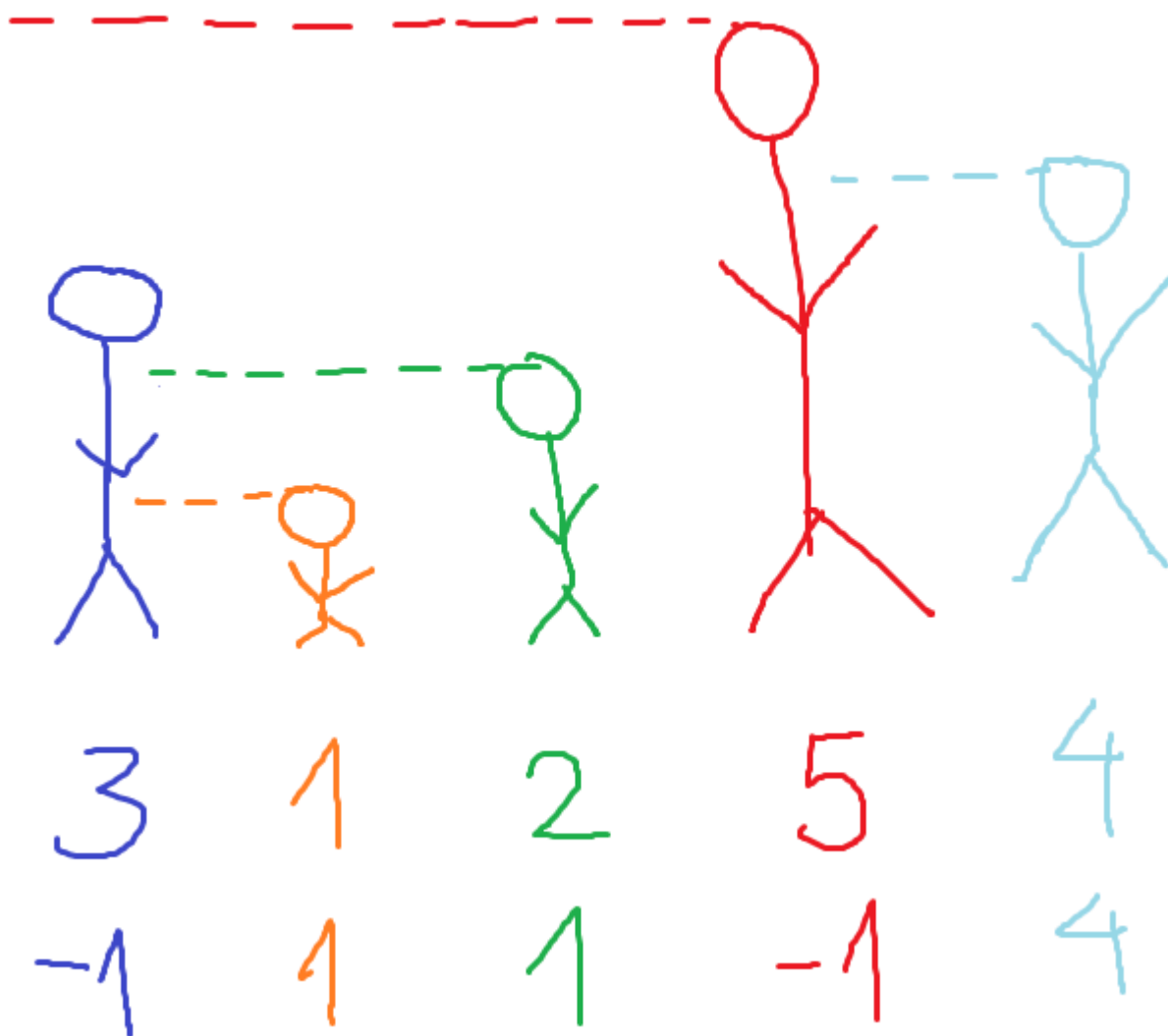
Mô hình lại bài toán

Xét mô hình sau:

- Phần tử thứ i của mảng A tượng trưng cho một người có chiều cao a_i .
- Lần lượt người từ 1 đến n xếp thành một hàng. Người thứ i sẽ *nhìn được* người thứ j nếu j đứng trước i trong hàng.
- Hàng luôn nhìn từ cuối về đầu.

Theo mô hình này, giá trị j gần i nhất ($j < i$) mà $a_i < a_j$ chính là chỉ số của người gần nhất cao hơn người i mà anh ta có thể nhìn được.

Hình ảnh minh họa, số ở hàng trên là chiều cao mỗi người, ở hàng dưới là chỉ số người gần nhất ở bên trái cao hơn họ:



Ta có thể cải tiến mô hình bằng việc chỉnh sửa cách thức xếp hàng.

Khi người thứ i xếp hàng, họ sẽ thực hiện các thao tác sau:

1. Xếp vào cuối hàng
2. Nếu họ đứng đầu hàng thì ghi lại số -1 . Mặt khác, nếu người đứng trước họ có chiều cao thấp hơn hoặc bằng với a_i , đuổi người đó ra khỏi hàng. Lặp lại quá trình đuổi người đứng trước thấp hơn cho tới khi người i đứng đầu hàng, hoặc người đứng trước có chiều cao lớn hơn. Ghi lại chỉ số của người đứng trước, hoặc -1 nếu họ đứng đầu hàng.

Ta sẽ chứng minh số mà mỗi người ghi lại chính là chỉ số của người gần nhất đứng trước mà cao hơn họ.

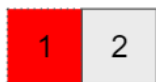
Lúc đầu, cho một người có chỉ số -1 , chiều cao $+\infty$. Việc đứng đầu hàng có thể coi như đứng ngay sau người chỉ số -1 này. Xét một người i bất kỳ ghi lại số j , $j < i$. Trước hết, a_j chắc chắn phải lớn hơn a_i . Giả sử ngược lại, j không phải là người gần i nhất mà cao hơn người i , mà k mới là người như vậy. Ta có $j < k < i$ và $a_k > a_i$. Tại thời điểm người i xếp vào hàng thì người k này *không còn trong hàng nữa* (nếu còn thì họ sẽ ghi lại người k chứ không phải j). Do đó, người k đã bị đuổi khỏi hàng bởi một người nào đó đứng sau k và trước i . Gọi chỉ số của người đó là l . Để đuổi người k thì $a_l \geq a_k$. Từ đó ta có $k < l < i$, $a_l \geq a_k > a_i$, trái với giả thiết k là người gần i nhất mà cao hơn i .

Như vậy, số mà mỗi người nhớ lại chính là **chỉ số của người gần nhất cao hơn họ** (-1 nếu không tồn tại người như vậy). Đây chính là giá trị j cần tìm với mỗi i .

Giả sử mảng $A = [1, 2, 7, 4, 3, 6]$. Các bước diễn ra như sau:



Người thứ nhất xếp vào hàng



Người thứ hai xếp vào hàng và đuổi người thứ nhất đi



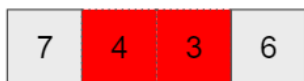
Người thứ ba xếp vào hàng và đuổi người thứ hai đi



Người thứ tư xếp vào hàng. Do người thứ ba cao hơn nên họ không đuổi ai đi



Người thứ năm xếp vào hàng, không đuổi ai đi



Người thứ sáu xếp vào hàng và đuổi người thứ tư và năm đi

Dễ thấy chiều cao của người trong hàng luôn tạo thành một dãy đơn điệu.

Cài đặt

Ta có thể biểu diễn mô hình nêu trên dưới dạng một Stack đơn điệu như sau:

- Người xếp vào hàng là phép **push**
- Đuổi người đứng trước là phép **pop**

Từ các bước nêu trong mô hình, có thể cài đặt lời giải như sau:

```

1  stack<int> st;
2
3  for (int i = 1; i <= n; ++i)
4  {
5      while (!st.empty() && a[st.top()] <= a[i])
6          st.pop();
7      int ans = -1;
8      if (!st.empty())
9          ans = st.top();
10     cout << ans << ' ';
11
12 
```

```

    st.push(i);
}

```

Đánh giá độ phức tạp

Độ phức tạp bộ nhớ của lời giải là $O(n)$ do sử dụng Stack và một mảng chứa n phần tử.

Thoạt nhìn, độ phức tạp tính toán của lời giải có vẻ là $O(n^2)$ do có vòng lặp `while` lồng trong vòng `for`. Tuy nhiên, để ý rằng mỗi phần tử a_i đều được `push()` vào Stack đúng một lần, và bị `pop()` khỏi Stack tối đa 1 lần, nên độ phức tạp tính toán vẫn là $O(n)$

Mở rộng

Bài toán gốc nêu trên có nhiều ứng dụng và mở rộng. Sau đây là một vài ví dụ.

Hình chữ nhật lớn nhất

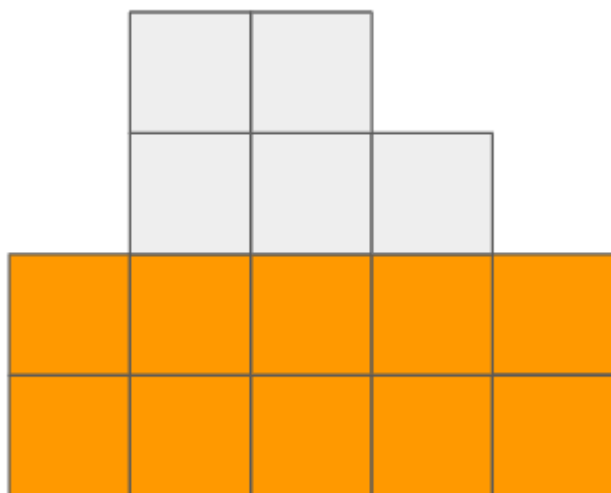
[Link SPOJ](#) .

Vẽ n cột hình chữ nhật sát nhau. Cột thứ i có chiều rộng 1 và chiều cao h_i . Tìm hình chữ nhật có diện tích lớn nhất tạo bởi các cột.

Ví dụ:

$$h = \{2, 4, 4, 3, 2\}$$

Hình chữ nhật lớn nhất có diện tích 10



Cách giải

Để ý rằng hình chữ nhật lớn nhất luôn có chiều cao bằng với chiều cao của một cột đã có.

Với mỗi cột i , ký hiệu L_i , R_i là cột gần nhất ở bên trái/phải i có chiều cao **nhỏ hơn** i . Việc tìm cột gần nhất ở bên trái/phải cột i mà thấp hơn h_i chính là bài toán gốc nêu ở mục trước. Ta chỉ cần sửa đổi thay vì tìm cột cao hơn thì phải tìm cột thấp hơn.

Khi đã xác định các mảng L và R , ta có thể xét từng cột i từ 1 đến n . Giả sử hình chữ nhật có chiều cao bằng với cột đang xét. Khi đó, chiều dài lớn nhất của hình chữ nhật là từ cột gần nhất bên trái thấp hơn cột i đến cột

gần nhất bên phải thấp hơn cột i . Diện tích của hình chữ nhật lớn nhất qua cột i là $(R_i - L_i - 1) * h_i$.

Vậy diện tích hình chữ nhật lớn nhất chính là giá trị $(R_i - L_i - 1) * h_i$ lớn nhất với mọi i từ 1 đến n .

Độ phức tạp thời gian và bộ nhớ của lời giải là $O(n)$.

Hình chữ nhật lớn nhất trong lưới ô vuông

[Link SPOJ](#) 

Đây là một mở rộng của bài toán trước. Cho lưới ô vuông $n \times m$, các ô có giá trị 0 hoặc 1. Ta cần tìm hình chữ nhật có diện tích lớn nhất có tất cả ô vuông có cùng giá trị.

Cách giải

Ta có thể chia bài toán thành hai trường hợp riêng biệt: tìm hình chữ nhật chỉ gồm các ô giá trị 0 và chỉ gồm các ô giá trị 1. Nếu giải được một trường hợp, ta cũng có thể dễ dàng giải trường hợp còn lại. Từ giờ, ta sẽ giải bài toán tìm hình chữ nhật lớn nhất *chỉ chứa giá trị 1*.





Ta xét lưới ô vuông con của lưới gốc, kích thước $k \times m$. Nói cách khác, lưới ô vuông con này chính là k hàng đầu tiên của lưới gốc. Với mỗi k từ 1 đến n , ta xét chiều cao của các cột chứa toàn số 1 dựng trên từng vị trí trong hàng. Sau đó, ta có thể áp dụng bài toán tìm hình chữ nhật lớn nhất tạo bởi các cột để giải quyết vấn đề.

```

1  for (int i = 1; i <= m; ++i) h[i] = 0;
2  for (int k = 1; k <= n; ++k)
3  {
4      for (int j = 1; j <= m; ++j)
5      {
6          if (grid[k][j] == 1) ++h[j];
7          else h[j] = 0;
8      }
9      // giải bài toán tìm hình chữ nhật lớn nhất trong các cột...
10 }
```

Độ phức tạp thời gian và bộ nhớ của lời giải là $O(n \times m)$.

Bài tập áp dụng

- [JNEXT](#) 
- [STPAR](#) 
- [280B - Codeforces](#) 
- [1132G - Codeforces](#) 
- Tham khảo thêm tại: [VNOI Problem List](#) 