Đại học Khoa học Tự nhiên
Đại học Quốc gia TP. HCM

Câu lạc bộ Học thuật NES
Mảng Điện tử - Kỹ thuật

Tài liệu ôn thi cuối kỳ môn Kỹ thuật lập trình

Bản thảo số 1 ngày 22/06/2021
Lưu hành nội bộ
Học kỳ 2, năm học 2020 - 2021

Mục lục

1 Đề bài

1.1 Con trỏ cơ bản

Cho biết kết quả in ra màn hình của các đoạn code dưới đây:

1.

2.

3.

4.

1.2 Con trỏ nâng cao

Hãy viết các đoạn code để cấp phát và giải phóng bộ nhớ và nhập nội dung cho các mảng động sau:

a. Mảng nguyên một chiều có n phần tử.

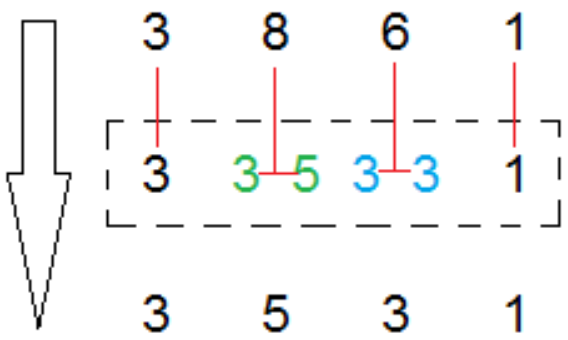
b. Ma trận thực $M \times N$.

c. Mảng có n chuỗi kí tự, mỗi chuỗi có độ dài khác nhau nhập từ bàn phím.

Lưu ý: Kích thước của các mảng trên được nhập từ bàn phím.

1.3 Danh sách liên kết

Một danh sách liên kết đơn có thành phần dữ liệu là số nguyên dương được mô tả như sau:
Hãy viết hàm
xử lý các số chẵn trong danh sách bằng cách phân tách số chẵn thành tổng 2 số lẻ có giá trị cách nhau không quá 2 đơn vị (thứ tự từ bé đến lớn), đồng thời sau đó loại bỏ các Node có dữ liệu giống nhau nằm liền nhau.



1.4 Ngăn xếp, hàng đợi

1.4.1 Ngăn xếp

Bài toán bên dưới nằm trong Interview Preparation Kit của **`hackerrank`***, nên các bạn có thể hình dung khi phỏng vấn xin việc thì những bài toán họ cho các bạn cũng nằm ở mức độ tương tự như thế này. Link: Balanced Brackets.*

Cho một chuỗi chứa các cặp dấu ngoặc `()` `[]` . Một chuỗi các dấu ngoặc được gọi là "hợp lệ" nếu như với mỗi dấu mở ngoặc thì nó sẽ có một dấu đóng ngoặc cùng loại tương ứng.

Ví dụ: các chuỗi sau là hợp lệ: `()[]`, `[()]`, `([])`, `[()]`, ...

Hãy thực hiện các yêu cầu sau:

a. Trình bày ý tưởng để biết được một chuỗi các cặp dấu ngoặc có là hợp lệ hay không?

b. Viết hàm

kiểm tra tính hợp lệ của một chuỗi như mô tả ở trên.

Gợi ý: dấu mở ngoặc nào xuất hiện trước sẽ luôn được đóng trước.

1.5 Đệ quy và giải thuật sắp xếp

Đề thi cuối kỳ KTLT lớp 19CTT4 (2020).

Cho cấu trúc danh sách liên kết như sau:

Hãy viết **hàm đệ quy** Selection Sort để sắp xếp danh sách liên kết trên tăng dần?

1.6 Quy hoạch động

Cho một dãy các mệnh giá tiền xu `int a[]` có kích thước `m` và một món hàng có giá tiền là `n` (xu). Hãy tìm ra cách lấy các đồng xu mang những mệnh giá đã cho (một mệnh giá có thể lấy một hoặc nhiều đồng xu) để trả tiền cho món hàng đó sao cho *số lượng đồng xu là nhỏ nhất*?

Ví dụ:

a	n	Output
{2, 3, 4}	1	`Khong duoc`
{2, 3, 4}	5	2 3
{2, 3, 4}	7	3 4
{2, 3, 4}	4	4
{2, 3, 4}	9	2 3 4
{2, 3, 4}	15	2 3 3 3 4
{2, 3, 5}	15	5 5 5

1.7 Bài tập tự luyện ở nhà

1.7.1 Cấp phát động

Cho cấu trúc số phức như ở bên dưới, hãy viết **hàm** cấp phát bộ nhớ động và giải phóng bộ nhớ cho một ma trận phức có kích thước $M \times N$ nhập từ bàn phím. Hãy viết thêm các **hàm** nhập, xuất dữ liệu và tính tổng các phần tử cho ma trận này.

1.7.2 Ngăn xếp

Bài toán bên dưới nằm trong Interview Preparation Kit của ***hackerrank****, nên các bạn có thể hình dung khi phỏng vấn xin việc thì những bài toán họ cho các bạn cũng nằm ở mức độ tương tự như thế này. Link: Queues: A Tale of Two Stacks.*

Sử dụng 2 stack, hãy "giả lập" ra một hàng đợi và thực hiện các yêu cầu sau:

a. Nêu ý tưởng thực hiện.

b. Tạo ra struct `QueueFS` (queue from stacks) chứa 2 stack kiểu nguyên và viết 3 hàm `enqueue`, `dequeue` và `peek` để thao tác với queue vừa tạo ra đó.

c. (Đọc thêm) Hãy xác định điểm chưa tối ưu của giải pháp vừa thực hiện và tối ưu nó.

Gợi ý: Khi ta sử dụng stack để "giả lập" một queue, vì 2 kiểu dữ liệu này ngược nhau nên khi enqueue vào queue thì phần tử đó sẽ nằm trên đỉnh stack. Tuy nhiên, lúc ta muốn dequeue thì phần tử cần dequeue lại nằm dưới đáy stack. Do đó ta cần 2 stack để giải quyết bài toán này.

1.7.3 Đệ quy

Game Minesweeper có một thuật toán rất hay đó là thuật toán vết dầu loang. Thuật toán được mô tả như sau: Có hai ma trận 2 chiều, một ma trận `int` được lưu trữ các số và boom `(-1)`, một ma trận `bool` để lưu trữ thuộc tính ô có được mở hay không. Khi ô được chọn là boom thì sẽ trả ra kết quả `BOOM!!`. Khi ô được chọn không phải boom thì nó sẽ loang ra:

- Nếu ô có boom thì nó sẽ không được mở.
- Nếu ô không có boom và không có số thì thuật toán loang sẽ tiếp tục cho ô đó.
- Nếu ô không có boom và có số thì chỉ mở ô đó.

Hãy viết thuật toán đệ quy cho hàm mở ô sử dụng thuật toán loang.

1.7.4 Quy hoạch động

Cho một số nguyên dương m ($m > 100$), có thể biểu diễn m thành tổng 3 số nguyên tố hay không? Nếu được hãy in 3 số nguyên tố đó ra.

2 Đáp án và giải thích

Các bạn có thể copy nội dung của các đoạn code trong phần này ở link sau:

2.1 Con trỏ cơ bản

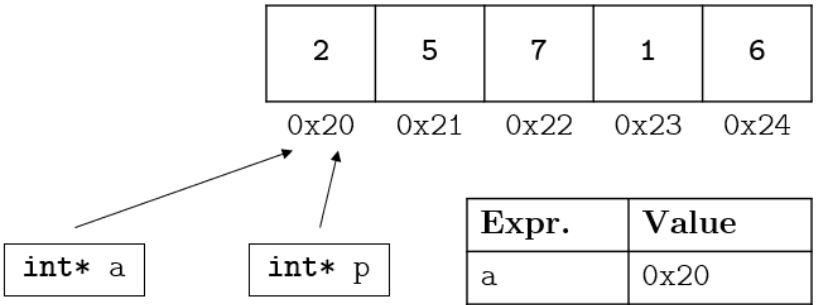
Lưu ý: các địa chỉ của các ô nhớ trong những hình dưới đây chỉ mang tính chất minh hoạ.

2.1.1 Câu a

Do `p` là một con trỏ, trỏ đến `a` nên `*p` chính là `a`. Phép gán `*p += 4` tương đương `a += 4` nên đáp án là `19.75`.

2.1.2 Câu b

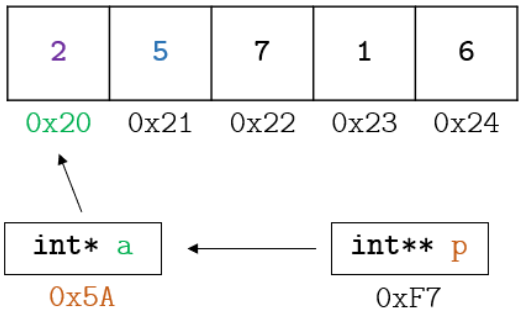
Do `p` trỏ đến cùng vùng nhớ với `a`, nên các thao tác truy xuất bộ nhớ thông qua `p` cũng hoàn toàn tương tự với các thao tác truy xuất bộ nhớ thông qua `a`.



Expr.	Value
a	0x20
a+1	0x21
*a	2
*(a+1)	5

Ta thấy `*(p+1)` trùng với `*(a+1)`, chính là `a[1]`. Còn `*a` chính là `a[0]`, có giá trị là 2.
Vậy `*(p + 1) += *a;` tương đương với phép gán `a[1] += a[0]`.
Tương tự, `cout << *(a + 1);` là `cout << a[1]`, nên kết quả in ra màn hình sẽ là 7.

2.1.3 Câu c



Expr.	Value
p	0x5A
a	0x20
*p	0x20
*a	2
*(*p)	2

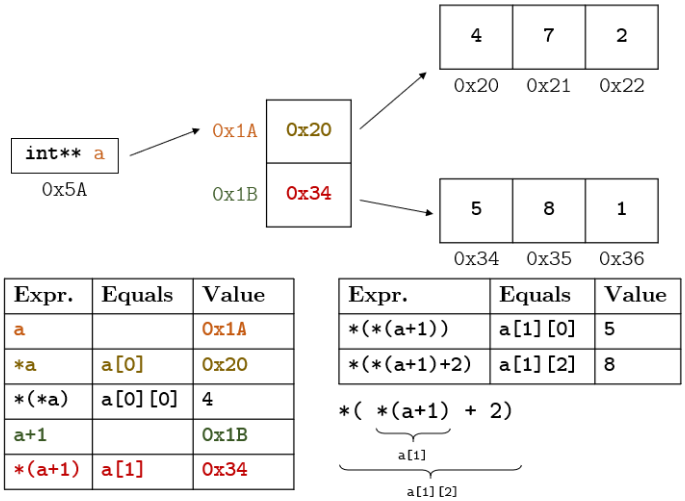
Expr.	Value
*(a+1)	5
*(*p+1)	5

Do `p` là một con trỏ, trỏ đến `a` nên `*p` chính là `a`.
`(*p)[4]` tương đương với `a[4]`, `*(*p + 1)` tương đương với `*(a + 1)`, chính là `a[1]`. Tự suy luận, ta có kết quả là 7.

2.1.4 Câu d

Ta thấy `*(a+2)` chính là `a[2]`.
Nếu ta đặt `u = *(a+2)` thì `(*(a+2)+1)` sẽ trở thành `*(u+1)`, chính là `u[1]`.
Mà vì `u = *(a+2) = a[2]` nên `u[1]` chính là `a[2][1]`.
Giá trị tại vị trí `a[2][1]` chính là `1`.

Trên đây là một cách giải thích theo phong cách "toán học". Nếu các bạn muốn một cách giải thích theo bản chất của con trỏ có thể tham khảo hình dưới đây (tự tham khảo):



Expr.	Equals	Value
a		0x1A
*a	a[0]	0x20
*(*a)	a[0][0]	4
a+1		0x1B
*(a+1)	a[1]	0x34

Expr.	Equals	Value
((a+1))	a[1][0]	5
((a+1)+2)	a[1][2]	8

*(*(a+1) + 2)
a[1]
a[1][2]

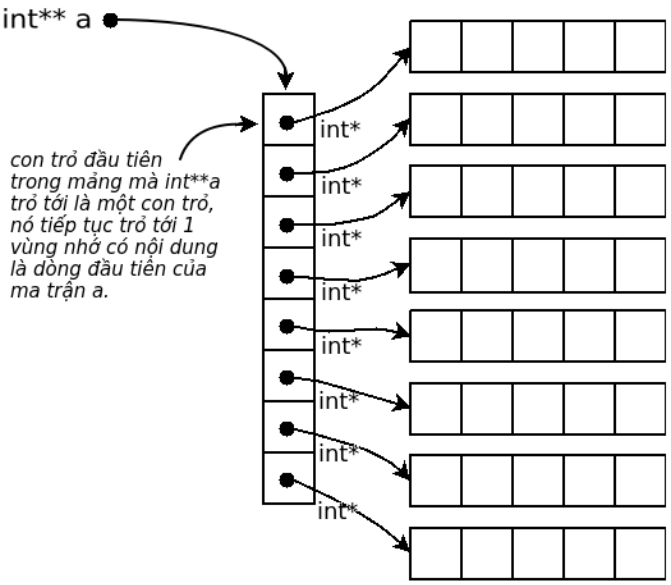
2.2 Con trỏ nâng cao

2.2.1 Câu a

```
// Cấp phát
int n;
cin >> n;
int* a = new int[n];

// Nhập
for (int i = 0; i < n; i++) {
    cin >> a[i];
}

// Giải phóng
delete[] a;
```

2.2.2 Câu b

Nhắc lại kiến thức về con trỏ 2 cấp:



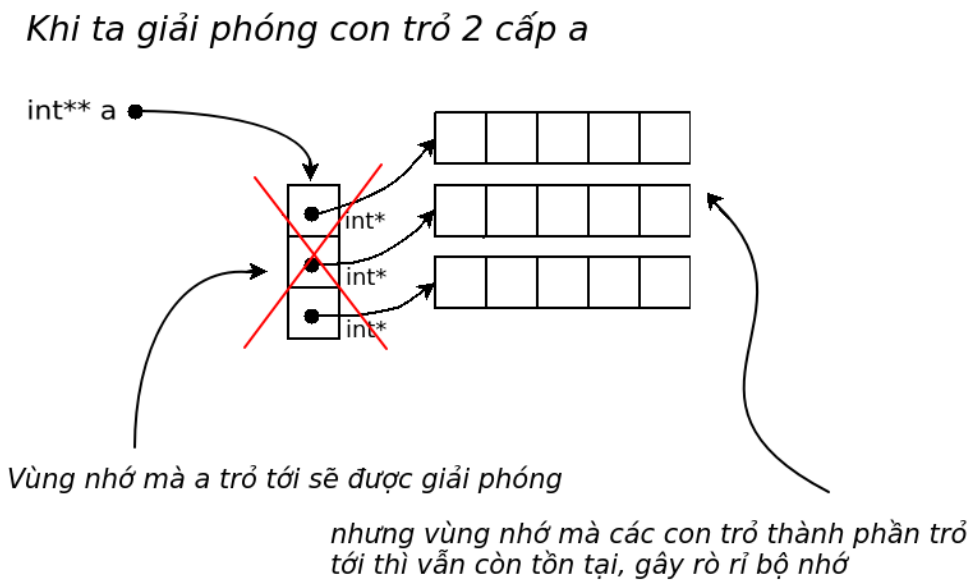
Con trỏ mảng 2 chiều là một con trỏ, trỏ đến một mảng một chiều (là danh sách các dòng của ma trận).

Trong mảng 1 chiều đó, mỗi phần tử lại là một con trỏ, trỏ đến một vùng nhớ chứa một mảng một chiều khác (chứa nội dung thực sự của dòng đó).

https://www.techiedelight.com/dynamically-allocate-memory-for-2d-array/

```cpp
// Cấp phát
int m, n;
cin >> m >> n;

// Một ma trận có m hàng
float** a = new (float*)[m];

// Mỗi hàng có n cột
for (int i = 0; i < m; i++) {
    a[i] = new float[n];
}

// Nhập
for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
        cin >> a[i][j];
    }
}

// Giải phóng
for (int i = 0; i < m; i++) {
    delete[] a[i];
}
delete[] a;
```

Lưu ý: Ta phải giải phóng các phần tử của a trước khi giải phóng a, vì bản thân a chỉ là một con trỏ 2 cấp, nó không thực sự chứa nội dung của ma trận. Khi ta giải phóng a thì các con trỏ thành phần bên trong a vẫn còn tồn tại, gây rò rỉ bộ nhớ nếu như ta không giải phóng trước.



https://www.techiedelight.com/dynamically-allocate-memory-for-2d-array/

2.2.3 Câu c

```
int n;
cin >> n;

// Mảng của các chuỗi
char** a = new (char*)[n];

// Cấp phát và nhập từng chuỗi
for (int i = 0; i < n; i++) {
    int size;
    cin >> size;

    a[i] = new char[size];
    gets(a[i]);
}

// Giải phóng
for (int i = 0; i < n; i++) {
    delete[] a[i];
}
delete[] a;
```

2.3 Danh sách liên kết

```
Node* initNode(int data) {
    Node* node = new Node;
    node->data = data;
    return node;
}
```

```
6
7  void deleteNode(Node* node) {
8      delete node;
9  }
10
11 void connectNode(Node* backNode, Node* frontNode) {
12     if (backNode == NULL) return;
13     backNode->next = frontNode;
14 }
15
16 void makeOdd(List& list) {
17     // tách số chẵn
18     int tmp;
19     Node* tmpNode;
20     for (Node* node = list.head; node != NULL; node = node->next) {
21         if (node->data % 2 != 0) continue;
22         tmp = node->data / 2;
23         if (tmp % 2 != 0) node->data = tmp;
24         else {
25             node->data = tmp - 1;
26             tmpNode = initNode(tmp + 1);
27             connectNode(tmpNode, node->next);
28             connectNode(node, tmpNode);
29         }
30     }
31
32     // loại node trùng nhau liên tiếp
33     int lastData = -1;
34     Node* prevNode = NULL;
35     Node* nextNode;
36     Node* node = list.head;
37     while (node != NULL) {
38         if (node->data == lastData) {
39             nextNode = node->next;
40             connectNode(prevNode, nextNode);
41             deleteNode(node);
42             node = nextNode;
43         }
44         else {
45             lastData = node->data;
46             prevNode = node;
47             node = node->next;
48         }
49     }
50 }
```

2.4 Ngăn xếp, hàng đợi

2.4.1 Ngăn xếp

Ta để ý là các cặp dấu ngoặc xuất hiện theo thứ tự first in first out, có nghĩa là dấu mở ngoặc nào tới sau thì nó phải được đóng trước. Do đó ta sử dụng stack để giải bài toán này. Cụ thể như sau:

- Ta duyệt qua từng kí tự của chuỗi đó.
- Nếu ta gặp một dấu mở ngoặc, ta đẩy nó vào stack.
- Nếu ta gặp một dấu đóng ngoặc, ta kiểm tra thử nó có cùng loại với dấu mở ngoặc mà ta gặp gần nhất hay không (dấu mở ngoặc gần nhất là phần tử trên đỉnh stack). Nếu có, ta tiếp tục xét kí tự tiếp theo. Nếu khác loại thì coi như chuỗi là không hợp lệ (trường hợp này là bị thiếu dấu mở ngoặc, dư dấu đóng ngoặc).
- Sau khi duyệt hết các kí tự trong chuỗi, ta kiểm tra xem trong stack còn dư phần tử nào hay không, nếu có thì ta đã rơi vào trường hợp dư dấu mở ngoặc, thiếu dấu đóng ngoặc.
- Nếu chuỗi nhập vào không vi phạm bất cứ tiêu chuẩn nào thì nó là hợp lệ.

```
bool checkBracketPairs(string s) {
    stack<char> history;

    // Ta duyệt qua từng kí tự
    for (char c : s) {
        // Nếu ta gặp dấu mở ngoặc thì push vào stack
        if (c == '(' || c == '[' || c == '{') {
            history.push(c);
        }
        // Nếu ta gặp dấu đóng ngoặc
        else if (c == ')' || c == ']' || c == '}') {
            // Nếu dấu đóng ngoặc đó không phải đồng loại
            // của dấu mở ngoặc trên đỉnh stack -> sai
            if (history.empty() || !sameType(history.top(), c) {
                return false;
            }
        }
    }

    // Nếu ta đã duyệt hết chuỗi rồi mà trong stack vẫn còn
    // dư phần tử -> sai
    if (!history.empty()) {
        return false;
    }

    // Nếu không sai thì là đúng :)
    return true;
}

```

```
30 // Hàm kiểm tra xem một cặp ngoặc có cùng loại hay không
31 bool sameType(char open, char close) {
32     if (open == '(' && close == ')') return true;
33     if (open == '[' && close == ']') return true;
34     if (open == '{' && close == '}') return true;
35     return false;
36 }
```

2.4.2 Hàng đợi

2.5 Đệ quy

Nhắc lại về thuật toán Selection Sort:

- Chọn ra phần tử nhỏ nhất trong n phần tử ban đầu.
- Hoán đổi phần tử đó lên đầu mảng.
- Bỏ qua phần tử đầu tiên, ta xem phần tử tiếp theo như là đầu mảng và tiếp tục lặp lại thuật toán cho đến khi hết mảng.

Ta có thể cài đặt một thuật toán đệ quy trên DSLK như sau:

```
bool checkBracketPairs(string s) {
    stack<char> history;

    // Ta duyệt qua từng kí tự
    for (char c : s) {
        // Nếu ta gặp dấu mở ngoặc thì push vào stack
        if (c == '(' || c == '[' || c == '{') {
            history.push(c);
        }
        // Nếu ta gặp dấu đóng ngoặc
        else if (c == ')' || c == ']' || c == '}') {
            // Nếu dấu đóng ngoặc đó không phải đồng loại
            // của dấu mở ngoặc trên đỉnh stack -> sai
            if (history.empty() || !sameType(history.top(), c) {
                return false;
            }
        }
    }

    // Nếu ta đã duyệt hết chuỗi rồi mà trong stack vẫn còn
    // dư phần tử -> sai
    if (!history.empty()) {
        return false;
    }

    // Nếu không sai thì là đúng :)
    return true;
```

```
}

// Hàm kiểm tra xem một cặp ngoặc có cùng loại hay không
bool sameType(char open, char close) {
    if (open == '(' && close == ')') return true;
    if (open == '[' && close == ']') return true;
    if (open == '{' && close == '}') return true;
    return false;
}
```

- Tìm phần tử nhỏ nhất trong danh sách liên kết.
- Hoán đổi node nhỏ nhất đó lên đầu dãy.
- Gọi hàm sắp xếp đệ quy cho dãy có `head` là `head->next`.

2.6 Quy hoạch động

Chưa soạn

2.7 Bài tập tự luyện ở nhà

Lưu ý: trong file pdf, phần giải thích này chỉ có giải thích, không có code. Các bạn vui lòng vào link được đính kèm ở đầu section để xem code.

2.7.1 Cấp phát động

Không có gì phải giải thích nhiều, thực hiện tương tự như khi thao tác với ma trận số nguyên. Cần lưu ý khi truyền một con trỏ trỏ đến kiểu dữ liệu `T` vào hàm, mà ta muốn thay đổi giá trị của con trỏ đó (như cấp phát hay xoá) thì ta cần phải truyền dưới dạng tham chiếu:

```
bool checkBracketPairs(string s) {
    stack<char> history;

    // Ta duyệt qua từng kí tự
    for (char c : s) {
        // Nếu ta gặp dấu mở ngoặc thì push vào stack
        if (c == '(' || c == '[' || c == '{') {
            history.push(c);
        }
        // Nếu ta gặp dấu đóng ngoặc
        else if (c == ')' || c == ']' || c == '}') {
            // Nếu dấu đóng ngoặc đó không phải đồng loại
            // của dấu mở ngoặc trên đỉnh stack -> sai
            if (history.empty() || !sameType(history.top(), c) {
                return false;
            }
        }
    }
```

```
    // Nếu ta đã duyệt hết chuỗi rồi mà trong stack vẫn còn
    // dư phần tử -> sai
    if (!history.empty()) {
        return false;
    }

    // Nếu không sai thì là đúng :)
    return true;
}

// Hàm kiểm tra xem một cặp ngoặc có cùng loại hay không
bool sameType(char open, char close) {
    if (open == '(' && close == ')') return true;
    if (open == '[' && close == ']') return true;
    if (open == '{' && close == '}') return true;
    return false;
}
```

Hoặc dưới dạng tham trỏ

```
bool checkBracketPairs(string s) {
    stack<char> history;

    // Ta duyệt qua từng kí tự
    for (char c : s) {
        // Nếu ta gặp dấu mở ngoặc thì push vào stack
        if (c == '(' || c == '[' || c == '{') {
            history.push(c);
        }
        // Nếu ta gặp dấu đóng ngoặc
        else if (c == ')' || c == ']' || c == '}') {
            // Nếu dấu đóng ngoặc đó không phải đồng loại
            // của dấu mở ngoặc trên đỉnh stack -> sai
            if (history.empty() || !sameType(history.top(), c) {
                return false;
            }
        }
    }

    // Nếu ta đã duyệt hết chuỗi rồi mà trong stack vẫn còn
    // dư phần tử -> sai
    if (!history.empty()) {
        return false;
    }

    // Nếu không sai thì là đúng :)
    return true;
}
```

```
// Hàm kiểm tra xem một cặp ngoặc có cùng loại hay không
bool sameType(char open, char close) {
    if (open == '(' && close == ')') return true;
    if (open == '[' && close == ']') return true;
    if (open == '{' && close == '}') return true;
    return false;
}
```

2.7.2 Ngăn xếp

Nhắc lại về 2 cấu trúc:

- Stack: phần tử nào vào sau thì ra trước (như một chồng dĩa).
- Queue: phần tử nào vào sau thì ra sau (như một hàng người đợi mua trà sữa).

Để implement một queue bằng 2 stack, ta có thể làm như sau:

- Giả sử ta có 2 stack gọi là stack (1) và stack (2).
- Khi push một phần tử vào queue, ta push vào (1).
- Khi muốn xem một phần tử ở đầu queue, vì phần tử ở đầu queue lại nằm dưới đáy của stack (1), nên ta pop tất cả phần tử từ stack (1) và chuyển qua stack (2). Lúc này, phần tử ở đầu queue đã nằm trên đỉnh của stack (2). Sau khi xem xong thì ta lại pop tất cả phần tử từ stack (2) bỏ về stack (1).
- Khi muốn xóa một phần tử khỏi queue, ta cũng chuyển tất cả phần tử từ (1) sang (2), nhưng sau đó phải bỏ phần tử trên đỉnh của stack (2) đi, rồi mới chuyển về lại stack (1).

Ảnh động minh hoạ:

- Link ảnh GIF xem đỉnh queue: https://ibb.co/64Hv1bH
- Link ảnh GIF xóa đỉnh queue: https://ibb.co/syPCBFS

Tối ưu hóa: Vì mỗi lần xem đỉnh queue ta phải pop (1) bỏ sang (2) rồi lại phải bỏ về (1), do đó ta nên tạo một biến tên là `top` bên trong struct queue của chúng ta để lưu giá trị của đỉnh queue để truy xuất thuận tiện hơn. Biến này sẽ được cập nhật giá trị khi và chỉ khi:

- Khi queue đang rỗng mà chúng ta thêm một phần tử mới vào, thì phần tử đó sẽ là đỉnh của queue (các phần tử khác sau đó nằm ở phía sau phần tử đầu).
- Khi ta pop đỉnh của queue mà trong queue vẫn còn dữ liệu thì đỉnh queue sẽ là phần tử ngay kế sau.