
第二课 栈、队列、堆

林沐

内容概述

1.7道经典栈、队列、堆的相关题目

例1:使用队列实现栈 (easy) (栈、队列)

例2:使用栈实现队列 (easy) (栈、队列)

例3:包含min函数的栈 (easy) (栈)

例4:合法的出栈序列 (medium) (栈、队列)

例5:简单的计算器(hard) (栈)

例6:数组中第K大的数(easy) (堆)

例7:寻找中位数(hard) (堆)

2.详细讲解题目解题方法、代码实现

预备知识:STL stack (栈)

栈，先进后出的线性表。

```
#include <stdio.h>
#include <stack>
int main() {
    std::stack<int> S;
    if (1) {
        printf("S is empty!");
    }

    S.push(5);
    S.push(6);
    2

    printf("S.top = %d\n", S.top());
    S.pop();
    3

    printf("S.top = %d\n", S.top());
    printf("S.size = %d\n", S.size());
    return 0;
}
```

S.top() : 取出栈顶

S.empty() : 判断栈是否为空

S.push(x) : 将x添加至栈

S.pop() : 弹出栈顶

S.size() : 栈的存储元素个数

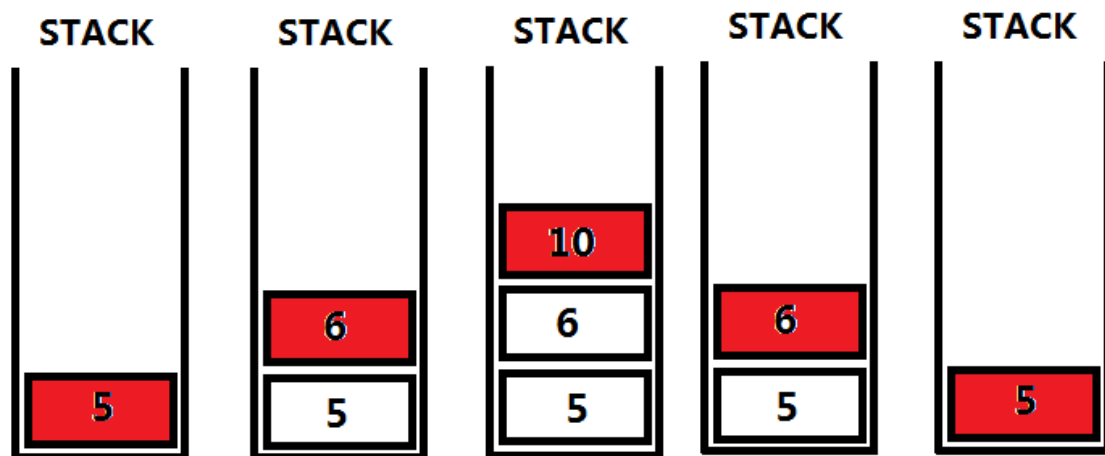
```
S is empty!
S.top = 10
S.top = 5
S.size = 1
请按任意键继续. . .
```

预备知识:STL stack (栈)

```
#include <stdio.h>
#include <stack>
int main() {
    std::stack<int> S;
    if (S.empty()) {
        printf("S is empty!");
    }
    S.push(5);
    S.push(6);
    S.push(10);

    printf("S.top = %d\n", S.top());
    S.pop();
    S.pop();




    printf("S.top = %d\n", S.top());
    printf("S.size = %d\n", S.size());
    return 0;
}
```



```
S is empty!
S.top = 10
S.top = 5
S.size = 1
请按任意键继续. . .
```

预备知识:STL queue(队列)

队列，先进先出的线性表。

```
#include <stdio.h>
#include <queue>
int main() {
    std::queue<int> Q;
    if (Q.empty()) {
        printf("Q is empty!\n");
    }
    
    Q.push(6);
    Q.push(10);
    printf("Q.front = %d\n", Q.front());
    Q.pop();
    
    printf("Q.front = %d\n", Q.front());
    
    printf("Q.back = %d\n", Q.back());
    printf("Q.size = %d\n", Q.size());
    return 0;
}
```

Q.empty(): 判断队列是否为空

Q.front(): 返回队列头部元素

Q.back(): 返回队列尾部元素

Q.pop(): 弹出队列头部元素

Q.push(x): 将x添加至队列

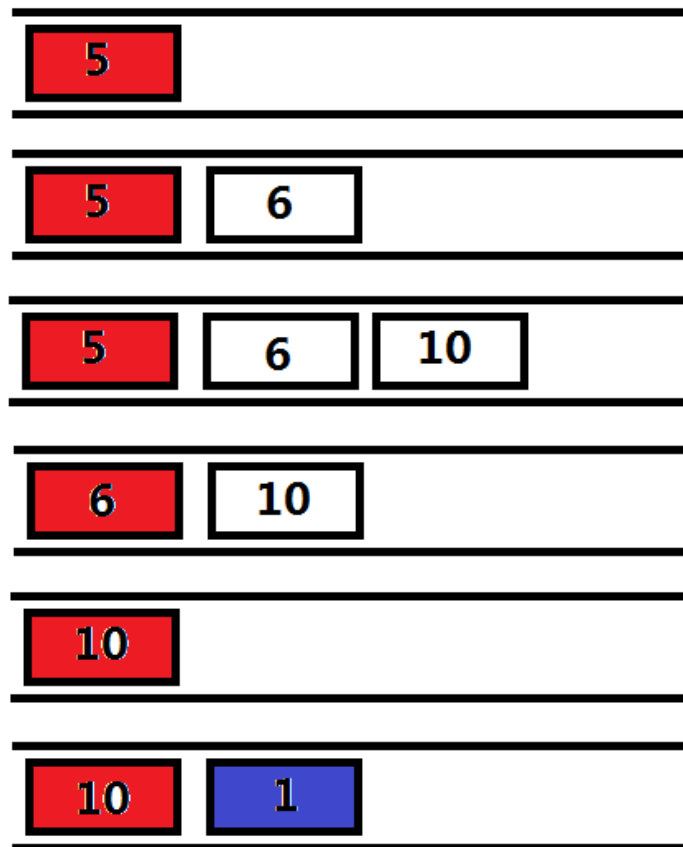
Q.size(): 返回队列的存储元素的个数

```
Q is empty!
Q.front = 5
Q.front = 10
Q.back = 1
Q.size = 2
请按任意键继续. . .
```

预备知识:STL queue(队列)

```
Q is empty?  
Q.front = 5  
Q.front = 10  
Q.back = 1  
Q.size = 2  
请按任意键继续. . .
```

```
#include <stdio.h>  
#include <queue>  
int main() {  
    std::queue<int> Q;  
    if (Q.empty()) {  
        printf("Q is empty!\n");  
    }  
    Q.push(5);  
    Q.push(6);  
    Q.push(10);  
    printf("Q.front = %d\n", Q.front());  
    Q.pop();  
    Q.pop();  
    printf("Q.front = %d\n", Q.front());  
    Q.push(1);  
    printf("Q.back = %d\n", Q.back());  
    printf("Q.size = %d\n", Q.size());  
    return 0;  
}
```



例1:使用队列实现栈

设计一个**栈**，支持如下操作，这些操作的算法复杂度需要是**常数级， $O(1)$** ，**栈**的内部存储数据的结构为**队列**，队列的方法只能包括push、peek(front)、pop、size、empty等**标准的队列方法**

- 1.**push(x)**：将元素x压入栈中
- 2.**pop()**：弹出(移除)栈顶元素
- 3.**top()**：返回栈顶元素
- 4.**empty()**：判断栈是否是空

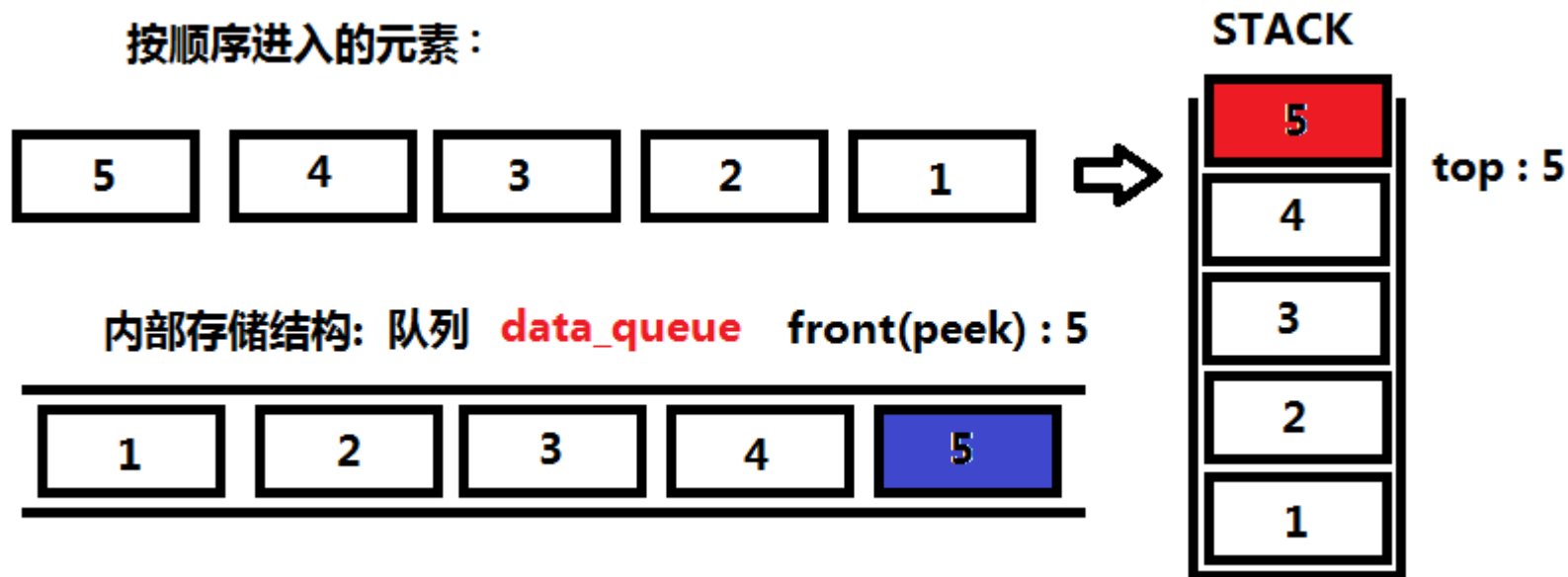
```
class MyStack {  
public:  
    MyStack() {  
    }  
    void push(int x) {  
    }  
    int pop() {  
    }  
    int top() {  
    }  
    bool empty() {  
    }  
};
```

选自 **LeetCode 225. Implement Stack using Queues**

<https://leetcode.com/problems/implement-stack-using-queues/description/>

难度:**Easy**

例1:思考



1. **push(x)** : ? ? ? (思考半分钟)
2. **pop()** : 弹出(移除)栈顶元素，即弹出(移除)队列头部元素
3. **top()** : 返回栈顶元素，即返回队列头部元素(front)
4. **empty()** : 判断队列是否是空，即判断队列是否为空

例1:思路

方案: 在STACK push元素时, 利用临时队列调换元素次序

图1: push前: data_queue

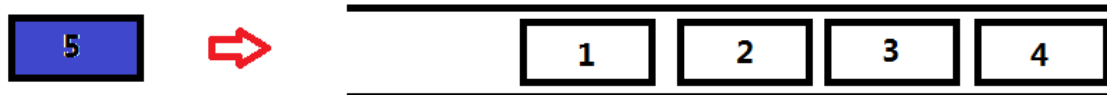


图2: 将新元素push进入 临时队列 temp_queue



图3: 将原队列内容push进入 临时队列 temp_queue

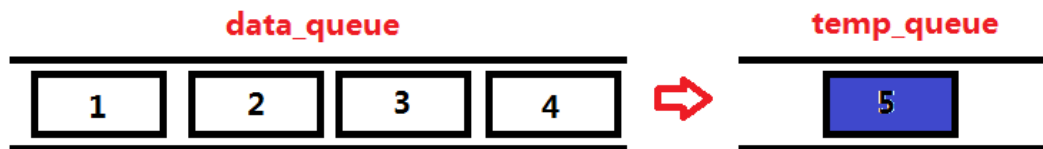


图4: 将临时队列 temp_queue 元素 push进入数据队列data_queue

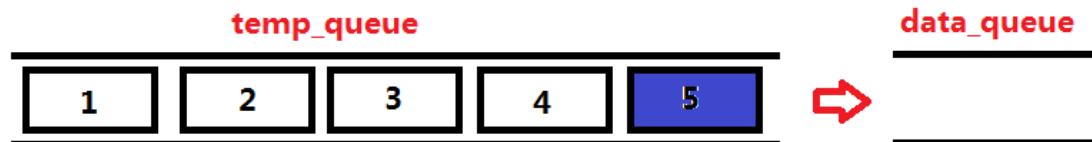
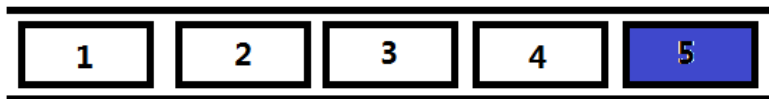
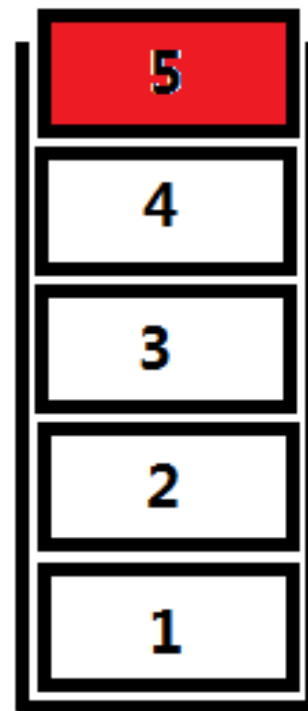


图5: 最终data queue结果:



STACK



例1:实现, 课堂练习

```
#include <queue>
class MyStack {
public:
    MyStack() {
    }
    void push(int x) {
        std::queue<int> temp_queue;


1


        while (!_data.empty()) {


2


            _data.pop();
        }
        while (!temp_queue.empty()) {


3


            temp_queue.pop();
        }
    }
    int pop() {
        int x = _data.front();
        _data.pop();
        return x;
    }
    int top() {
        return _data.front();
    }
    bool empty() {
        return _data.empty();
    }
private:
    std::queue<int> _data;
};
```

3分钟时间填写
代码,
有问题随时
提出!

例1:实现

```
#include <queue>
class MyStack {
public:
    MyStack() {
    }
    void push(int x) {
        std::queue<int> temp_queue;
        temp_queue.push(x); //先将新元素push进入temp_queue
        while (!_data.empty()) {
            temp_queue.push(_data.front()); //将数据队列元素导入临时队列
            _data.pop();
        }
        while (!temp_queue.empty()) {
            _data.push(temp_queue.front()); //将临时队列元素再导入数据队列
            temp_queue.pop();
        }
    }
    int pop() {
        int x = _data.front();
        _data.pop();
        return x;
    }
    int top() {
        return _data.front();
    }
    bool empty() {
        return _data.empty();
    }
private:
    std::queue<int> _data;
};
```

例1:测试与leetcode提交结果

```
int main() {  
    MyStack S;  
    S.push(1);  
    S.push(2);  
    S.push(3);  
    S.push(4);  
    printf("%d\n", S.top());  
    S.pop();  
    printf("%d\n", S.top());  
    S.push(5);  
    printf("%d\n", S.top());  
    return 0;  
}
```

Implement Stack using Queues

Submission Details

16 / 16 test cases passed.

Status: **Accepted**

Runtime: 0 ms

Submitted: 0 minutes ago

```
4  
3  
5  
请按任意键继续. . .
```

例2:使用栈实现队列

设计一个**队列**，**队列**支持如下操作，这些操作的算法复杂度需要是**常数级**， **$O(1)$** ，**队列**的内部存储数据的结构为**栈**，栈的方法只能包括push、top、pop、size、empty等**标准的栈方法**

1. **push(x)** : 将元素x压入队列中
2. **pop()** : 弹出(移除)队列头部元素
3. **peek()** : 返回队列头部元素(即为front)
4. **empty()** : 判断队列是否是空

选自 **LeetCode 232. Implement Queue using Stacks**

<https://leetcode.com/problems/implement-queue-using-stacks/description/>

难度:**Easy**

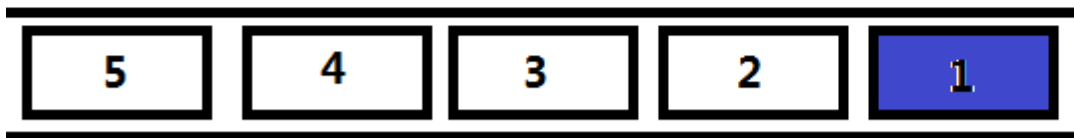
```
class MyQueue {
public:
    MyQueue() {}
    void push(int x) {}
    int pop() {}
    int peek() {}
    bool empty() {}
};
```

例2:思考

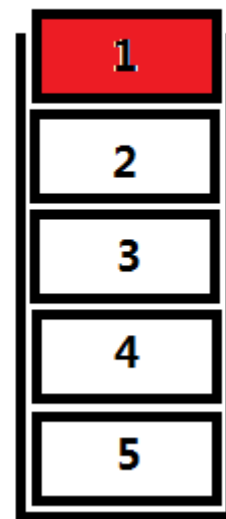
按顺序进入的元素：



front(peek) : 5



内部存储结构: **data_stack**



1. **push(x)** : ? ? ? (思考半分钟)
2. **pop()** : 弹出(移除)队列头部元素，即弹出(移除)栈头部元素
3. **peek()** : 返回队列头部元素(即为front)，即返回栈顶元素(top)
4. **empty()** : 判断队列是否是空，即判断栈是否为空

例2:思路

方案: 在 队列 push元素时, 利用**临时 栈** 调换元素次序

图1:

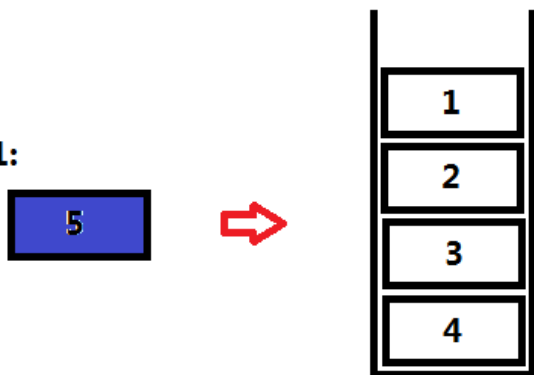


图2: 将原数据栈内容 push 进入 临时栈 temp_stack

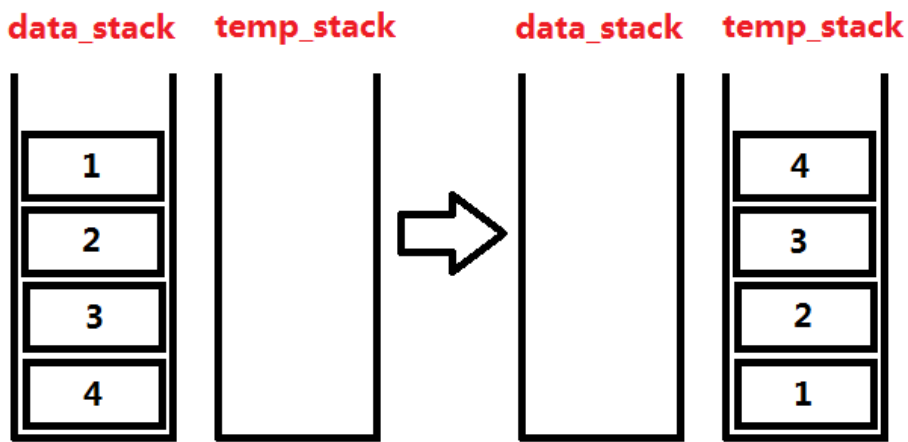


图3:将新数据 push 进入 临时栈 temp_stack

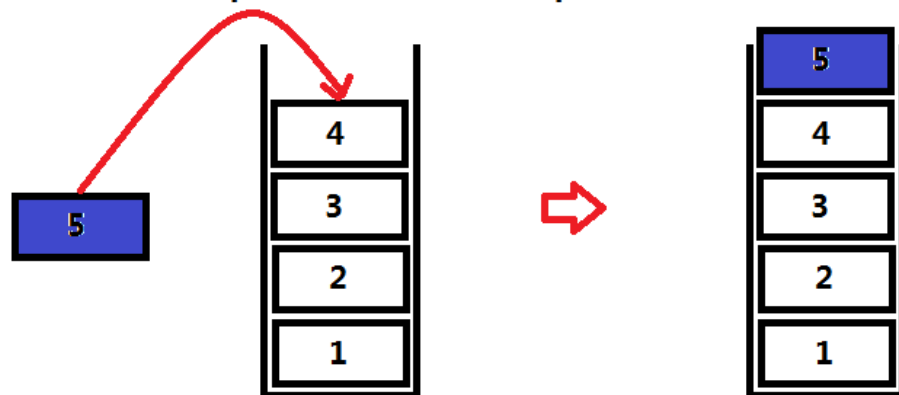
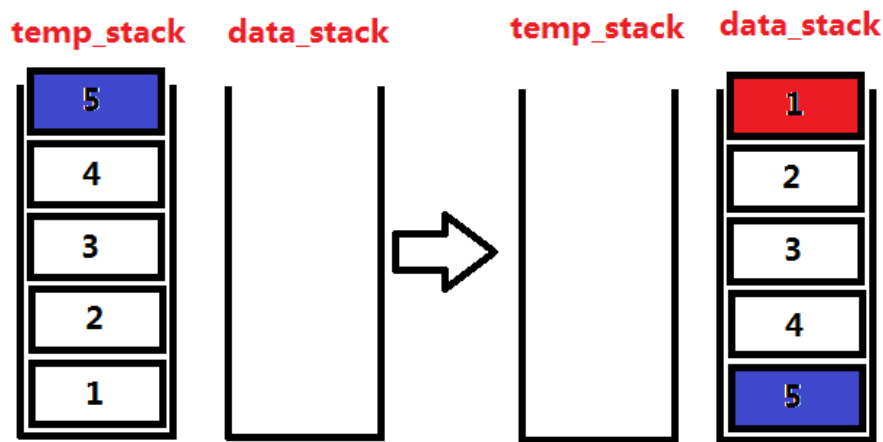


图4:将临时栈temp_stack 中的元素push进入 数据栈data_stack



例2:实现， 课堂练习

```
#include <stack>
class MyQueue {
public:
    MyQueue() {
    }
    void push(int x) {
        std::stack<int> temp_stack;
        while(!_data.empty()){
            1
            _data.pop();
        }
        2
        while(!temp_stack.empty()){
            3
            temp_stack.pop();
        }
    }
    int pop() {
        int x = _data.top();
        _data.pop();
        return x;
    }
    int peek() {
        return _data.top();
    }
    bool empty() {
        return _data.empty();
    }
private:
    std::stack<int> _data;
};
```

3分钟时间填写
代码，
有问题随时
提出！

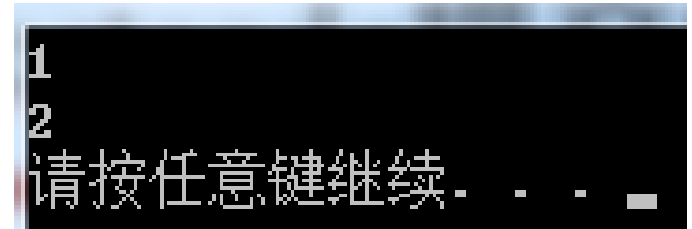
例2:实现

```
#include <stack>
class MyQueue {
public:
    MyQueue() {
    }
    void push(int x) {
        std::stack<int> temp_stack; //将数据栈中的元素push到临时栈中
        while (!_data.empty()) {
            temp_stack.push(_data.top());
            _data.pop();
        } //将新元素push到临时栈中
        temp_stack.push(x);
        while (!temp_stack.empty()) { //将临时栈中的元素push到数据栈中
            _data.push(temp_stack.top());
            temp_stack.pop();
        }
    }
    int pop() {
        int x = _data.top();
        _data.pop();
        return x;
    }
    int peek() {
        return _data.top();
    }
    bool empty() {
        return _data.empty();
    }
private:
    std::stack<int> _data;
};
```



例2:测试与leetcode提交结果

```
int main() {  
    MyQueue Q;  
    Q.push(1);  
    Q.push(2);  
    Q.push(3);  
    Q.push(4);  
    printf("%d\n", Q.peek());  
    Q.pop();  
    printf("%d\n", Q.peek());  
    return 0;  
}
```



```
1  
2  
请按任意键继续. . .
```

Implement Queue using Stacks

Submission Details

17 / 17 test cases passed.

Status: **Accepted**

Runtime: 3 ms

Submitted: 0 minutes ago

例3:包含min函数的栈

设计一个**栈**，支持如下操作，这些操作的算法复杂度需要是**常数级， $O(1)$**

- 1.**push(x)**：将元素x压入栈中
- 2.**pop()**：弹出(移除)栈顶元素
- 3.**top()**：返回栈顶元素
- 4.**getMin()**：返回栈内最小元素

选自 **LeetCode 155. Min Stack**

<https://leetcode.com/problems/min-stack/description/>

难度:**Easy**

```
class MinStack {
public:
    MinStack() {
    } //构造函数

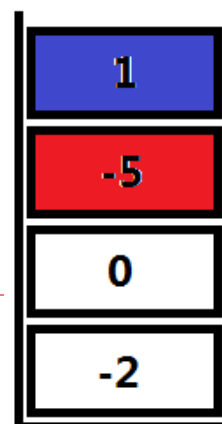
    void push(int x) {
    } //将元素x压入栈

    void pop() {
    } //将栈顶元素弹出

    int top() {
    } //返回栈顶元素

    int getMin() {
    } //返回栈内最小元素
};
```

STACK



top() 返回 1

getMin() 返回 -5

例3:思考, 1个变量记录最小值?

图1

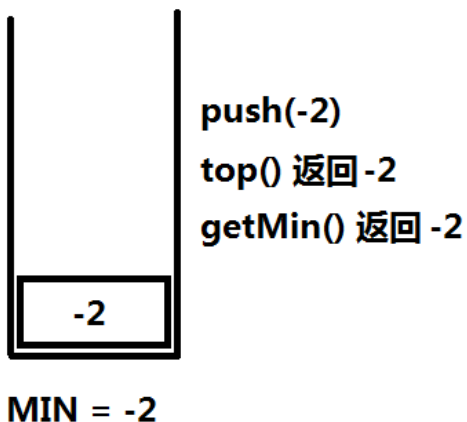


图2

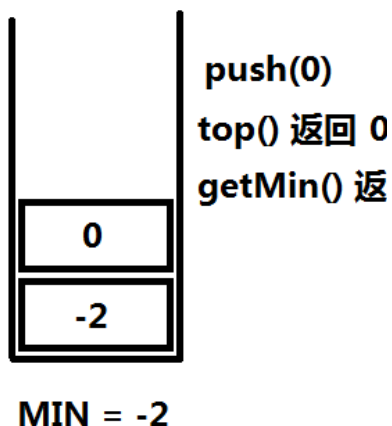


图3

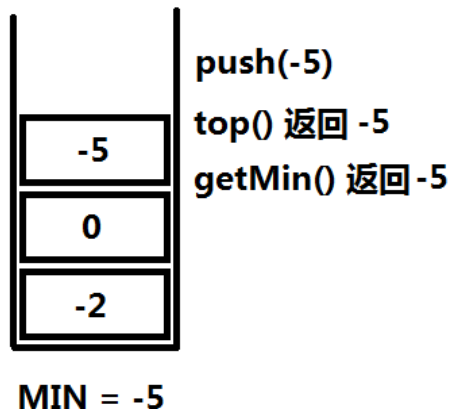
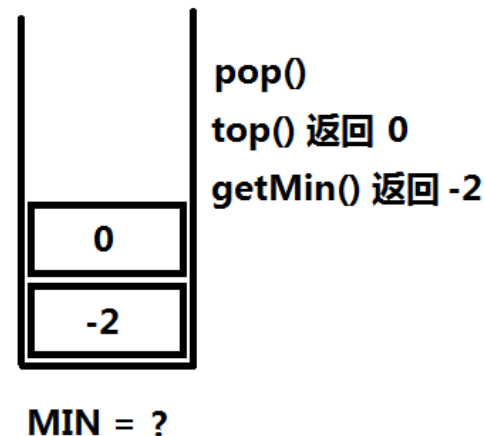


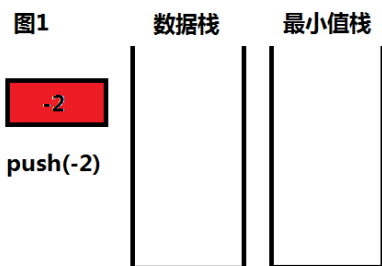
图4



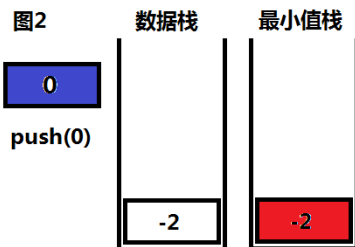
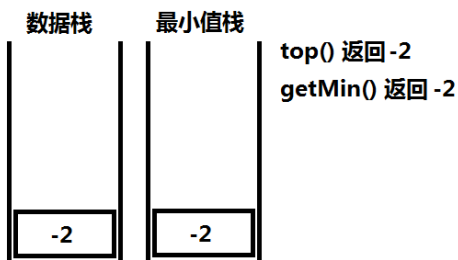
结论:

1. 1个变量MIN无法完成记录栈中**所有状态**下的最小值。
2. 栈的**每个状态**, 都需要有一个变量记录最小值。

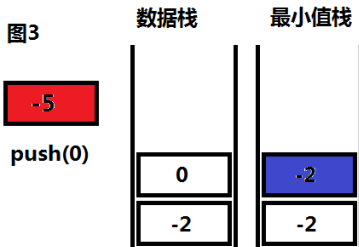
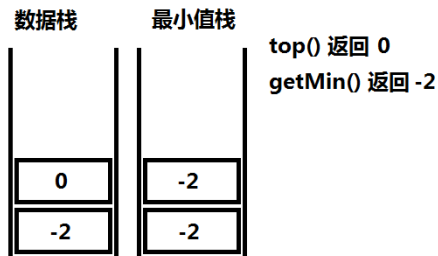
例3:用另一个栈，存储各个状态最小值



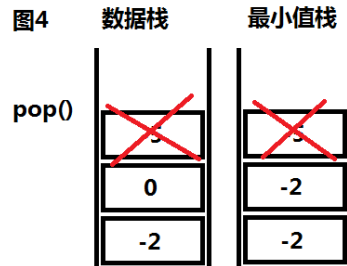
栈为空，直接入栈



0 > 最小值栈顶 -2



-5 < 最小值栈顶 -2



例3:实现, 课堂练习

```
class MinStack {
public:
    MinStack() {
    }
    void push(int x) {
        _data.push(x); //将数据压入数据栈

        if (1) {
            _min.push(x);
        } //如果最小值栈空, 直接将数据压入栈
        else{
            if (x > _min.top()) {
                2
            }
            _min.push(x);
        }
    } //比较当前数据与最小值栈顶数据大小, 选择较小的压入最小值栈
    void pop() {
        _data.pop(); //数据栈与最小值栈同时弹出
        3
    }
    int top() { //获取数据栈栈顶
        return _data.top();
    }
    int getMin() { //获取最小值栈栈顶
        return _min.top();
    }
private:
    std::stack<int> _data; //数据栈
    std::stack<int> _min; //最小值栈
};
```

3分钟时间填写
代码,
有问题随时
提出!

例3:实现

```
class MinStack {
public:
    MinStack() {
    }
    void push(int x) {
        _data.push(x); //将数据压入数据栈
        1 if (!_min.empty()) {
            _min.push(x);
        } //如果最小值栈空，直接将数据压入栈
        else{
            if (x > _min.top()) {
                2 x = _min.top();
            }
            _min.push(x);
        }
    } //比较当前数据与最小值栈顶数据大小，选择较小的压入最小值栈
    void pop() {
        _data.pop(); //数据栈与最小值栈同时弹出
        3 _min.pop();
    }
    int top() { //获取数据栈栈顶
        return _data.top();
    }
    int getMin() { //获取最小值栈栈顶
        return _min.top();
    }
private:
    std::stack<int> _data; //数据栈
    std::stack<int> _min; //最小值栈
};
```



例3:测试与leetcode提交结果

```
int main() {
    MinStack minStack;
    minStack.push(-2);
    printf("top = [%d]\n", minStack.top());
    printf("min = [%d]\n\n", minStack.getMin());
    minStack.push(0);
    printf("top = [%d]\n", minStack.top());
    printf("min = [%d]\n\n", minStack.getMin());
    minStack.push(-5);
    printf("top = [%d]\n", minStack.top());
    printf("min = [%d]\n\n", minStack.getMin());
    minStack.pop();
    printf("top = [%d]\n", minStack.top());
    printf("min = [%d]\n\n", minStack.getMin());
    return 0;
}
```

Min Stack

```
top = [-2]
min = [-2]
```

```
top = [0]
min = [-2]
```

```
top = [-5]
min = [-5]
```

```
top = [0]
min = [-2]
```

请按任意键继续. . .

Submission Details

18 / 18 test cases passed.

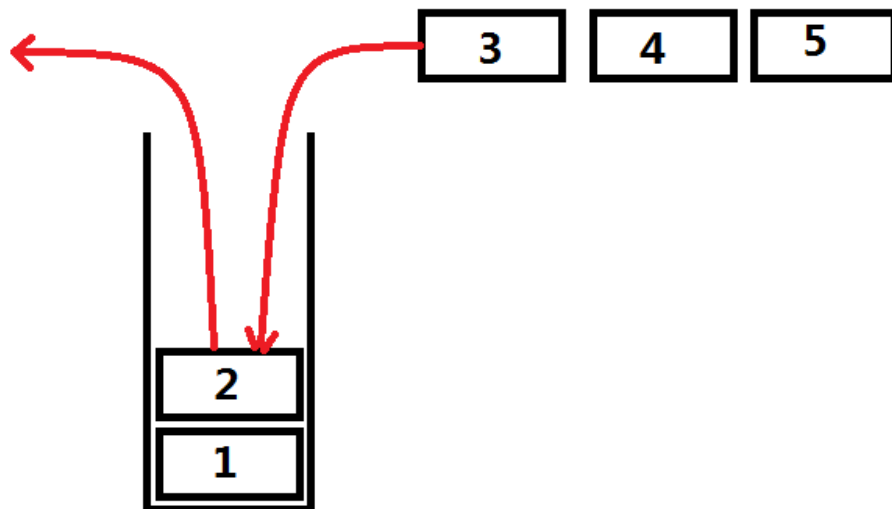
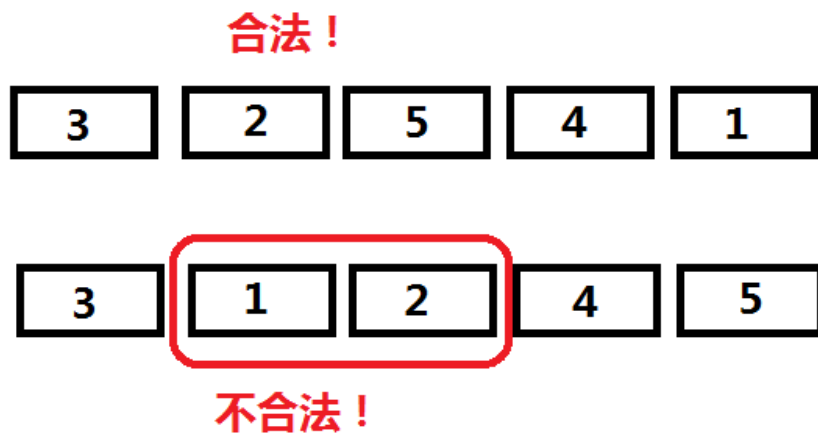
Runtime: 42 ms

Status: Accepted

Submitted: 0 minutes ago

例4:合法的出栈序列

已知从1至n的**数字序列**，**按顺序**入栈，每个数字入栈后即可出栈，也可在栈中停留，等待后面的数字入栈出栈后，该数字再出栈，求该数字序列的**出栈序列**是否合法？

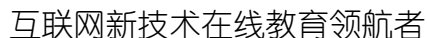


选自 **poj 1363 Rails**

<http://poj.org/problem?id=1363>

难度:**Medium**

POJ 即“**北京大学程序在线评测系统**”（Peking University Online Judge）的缩写，主要收录**ACM**国际大学生程序设计竞赛、**NOI**青少年信息学奥林匹克竞赛等各类程序设计竞赛题目，当前共有3000多道。



例4: 思路:使用栈与队列模拟入栈、出栈过程

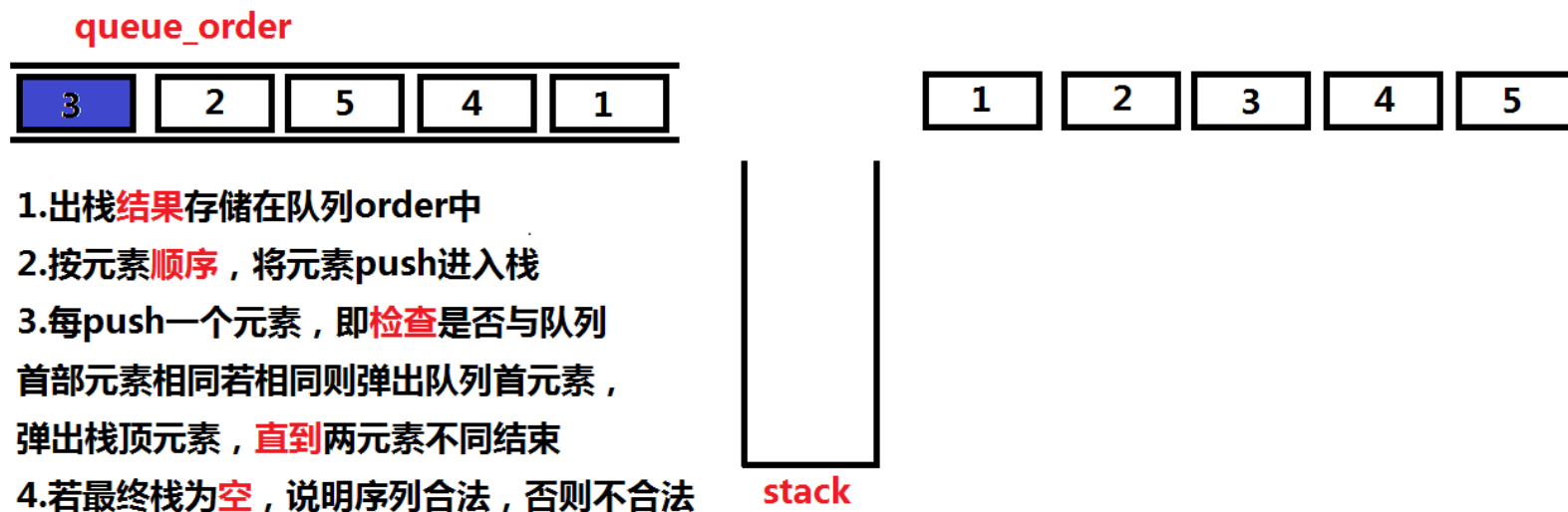
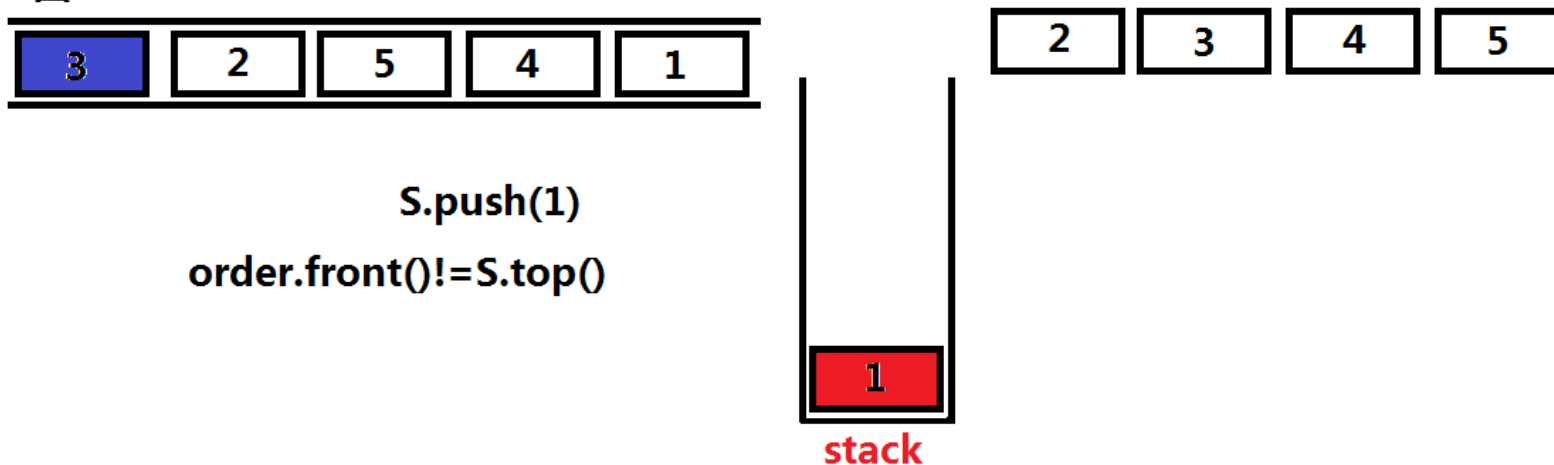


图1: queue_order



例4: 思路:使用栈与队列模拟入栈、出栈过程

图2: queue_order

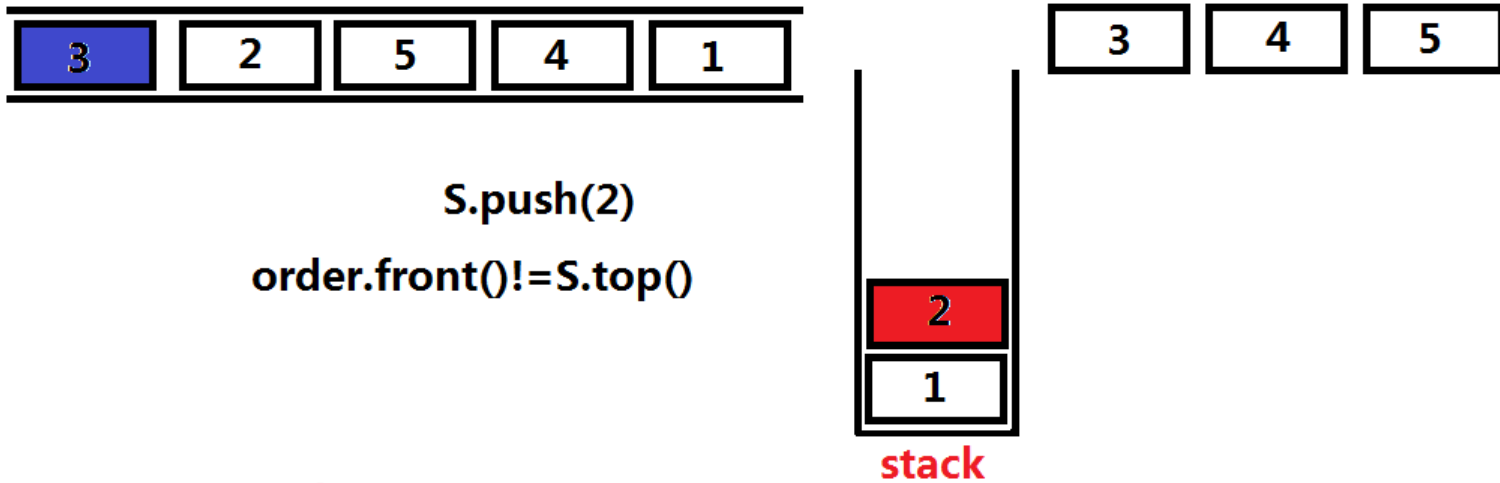
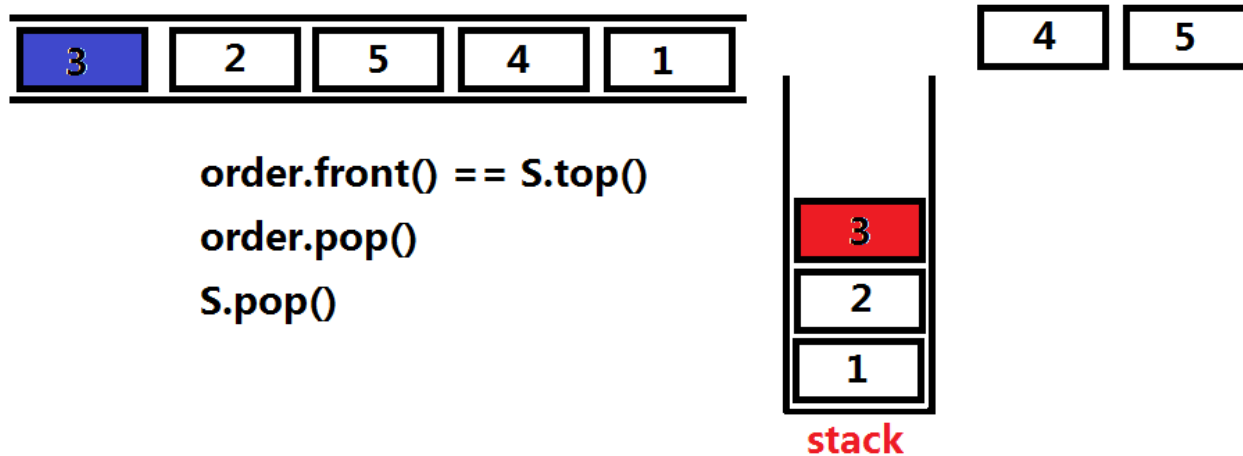


图3: queue_order



例4: 思路:使用栈与队列模拟入栈、出栈过程

图4: queue_order

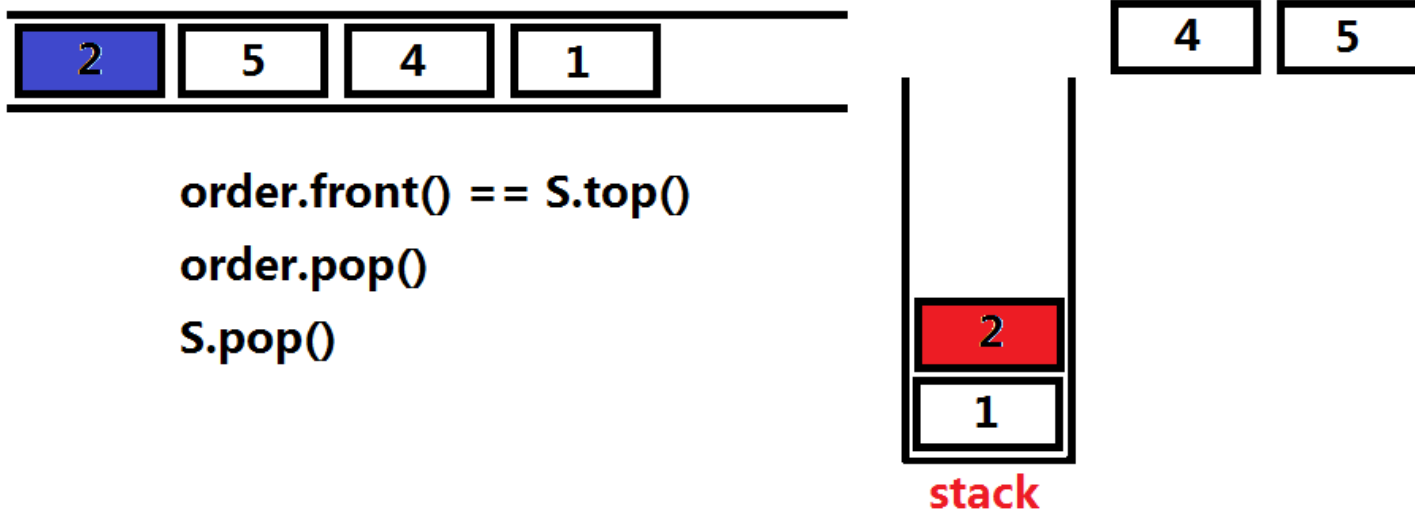
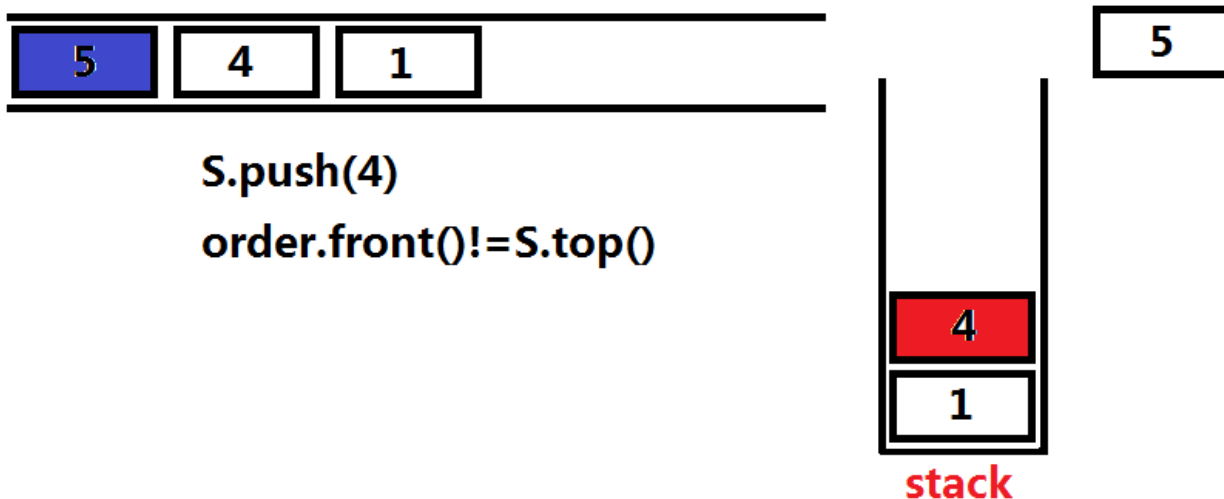


图5: queue_order

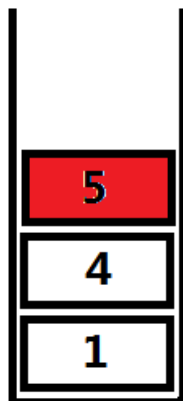


例4: 思路:使用栈与队列模拟入栈、出栈过程

图6: queue_order



`order.front() == S.top()`
`order.pop()`
`S.pop()`



stack

图8: queue_order

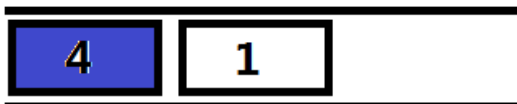


`order.front() == S.top()`
`order.pop()`
`S.pop()`

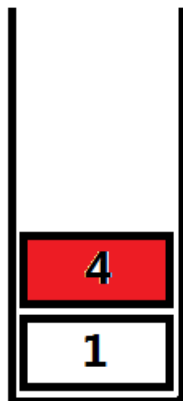


stack

图7: queue_order



`order.front() == S.top()`
`order.pop()`
`S.pop()`



stack

图9: queue_order



`S.empty()`
故:order序列合法!



stack

例4:实现， 课堂练习

```
#include <stack>
#include <queue>

//检查序列(存储在队列中)
bool check_is_valid_order(std::queue<int> &order) {
    std::stack<int> S; //S为模拟栈
    int n = order.size(); //n为序列长度,将1-n按顺序入栈
    for (int i = 1; i <= n; i++) {
        1
        while ( 2 ) {
            S.pop();
            order.pop();
        }
        if ( 3 ) {
            return false;
        }
        return true;
    }
}
```

3分钟时间填写
代码，
有问题随时
提出！

例4:实现

```
#include <stack>
#include <queue>

//检查序列(存储在队列中)
bool check_is_valid_order(std::queue<int> &order) {
    std::stack<int> S; //S为模拟栈
    int n = order.size(); //n为序列长度,将1-n按顺序入栈
    for (int i = 1; i <= n; i++) {
        S.push(i); //将i入栈
        while (!S.empty() && order.front() == S.top()) {
            S.pop();
            order.pop(); //只要S不空且队列头部与栈顶相同,即弹出元素
        }
    }
    if (!S.empty()) { //如果最终栈不空,则说明序列不合法!
        return false;
    }
    return true;
}
```


例4:poj测试与提交

Problem	Result	Memory	Time	Language	Code Length
1363	Accepted	240K	313MS	C++	741B

```
int main() {
    int n;
    int train;
    scanf("%d", &n);
    while(n) {
        scanf("%d", &train);
        while (train) {
            std::queue<int> order;
            order.push(train);
            for (int i = 1; i < n; i++) {
                scanf("%d", &train);
                order.push(train);
            }
            if (check_is_valid_order(order)) {
                printf("Yes\n");
            }
            else {
                printf("No\n");
            }
            scanf("%d", &train);
        }
        printf("\n");
        scanf("%d", &n);
    }
    return 0;
}
```

Sample Input

```
5
1 2 3 4 5
5 4 1 2 3
0
6
6 5 4 3 2 1
0
0
```

Sample Output

```
Yes
No
Yes
```

例5:简单的计算器

设计一个**计算器**，输入一个**字符串存储**的数学表达式，可以计算包括“(“、”)”、“+”、“-”四种符号的**数学表达式**，输入的数学表达式字符串**保证是合法的**。输入的数学表达式中可能存在空格字符。

如计算:

“(1 + 1)” = 2

“1+121 - (14+ (5-6)) = 109

```
class Solution {  
public:  
    int calculate(std::string s) {  
    }  
};
```

选自 **LeetCode 224. Basic Calculator**

<https://leetcode.com/problems/basic-calculator/description/>

难度:**Hard**

例5:计算思路

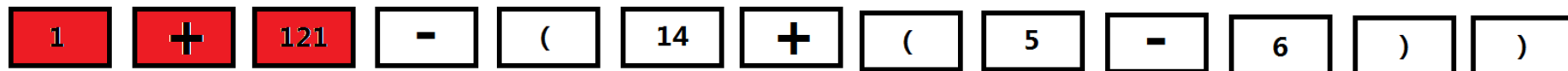


图1: compute_flag = 0

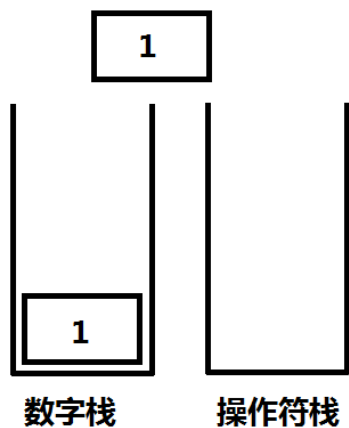


图2: compute_flag = 0

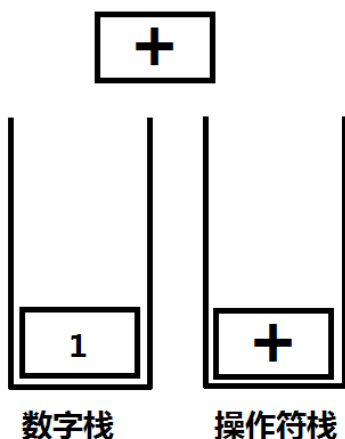


图3: compute_flag = 1

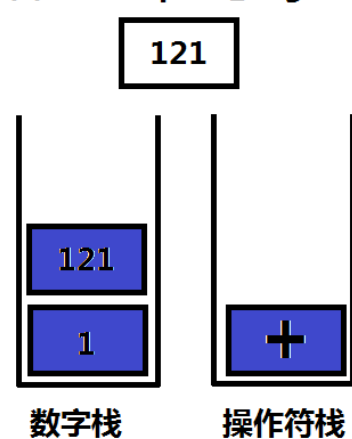
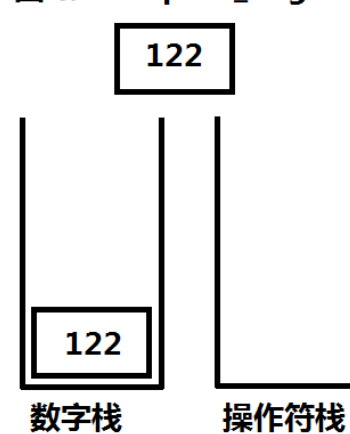


图4: compute_flag = 1



例5:计算思路



图1: compute_flag = 1

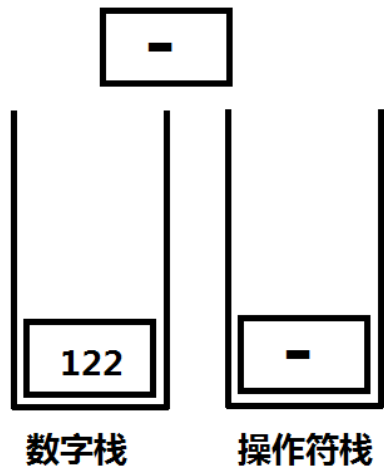


图2: compute_flag = 0

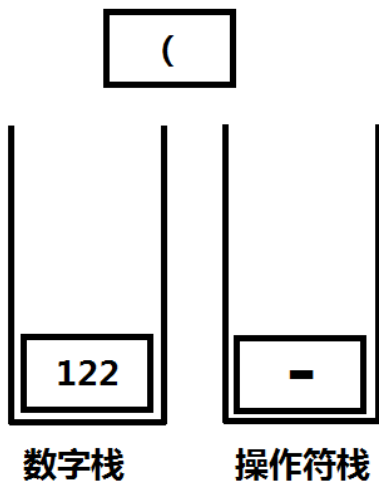


图3: compute_flag = 0

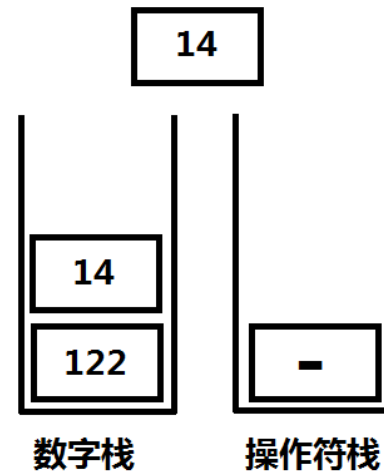
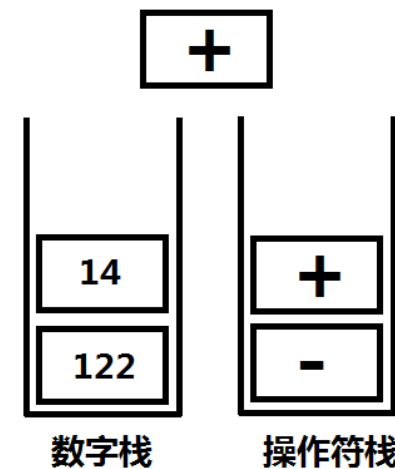


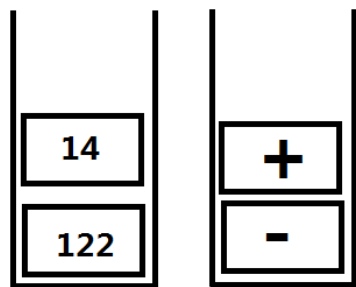
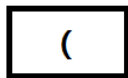
图4: compute_flag = 1



例5:计算思路



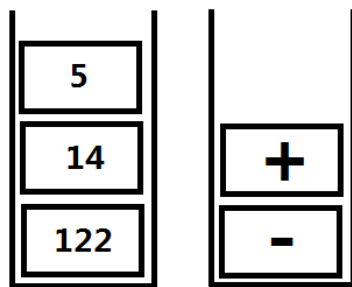
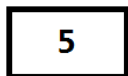
图1: compute_flag = 0



数字栈

操作符栈

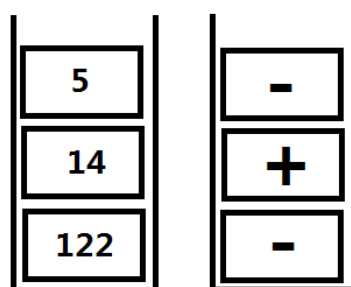
图2: compute_flag = 0



数字栈

操作符栈

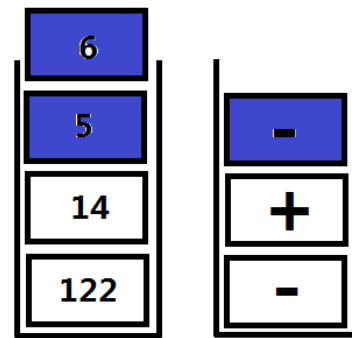
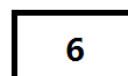
图3: compute_flag = 1



数字栈

操作符栈

图4: compute_flag = 1



数字栈

操作符栈

例5:计算思路



图1: compute_flag = 1

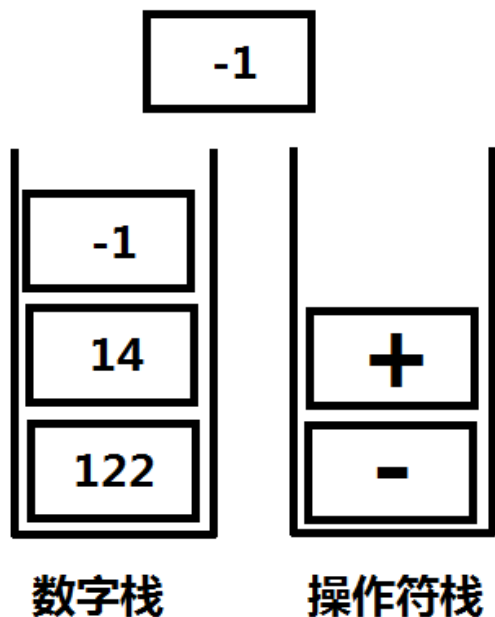


图2: compute_flag = 1

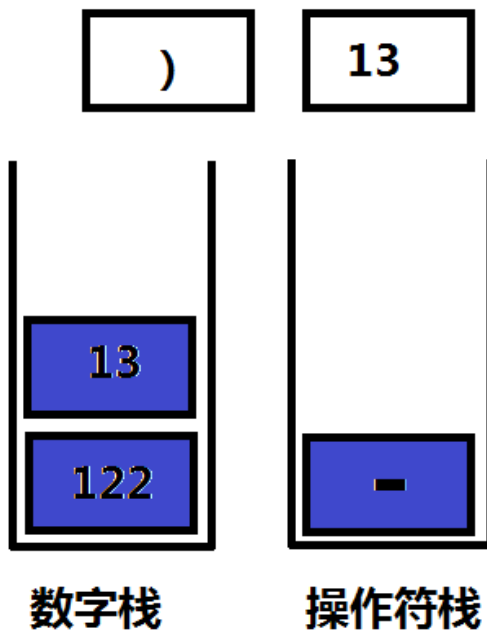
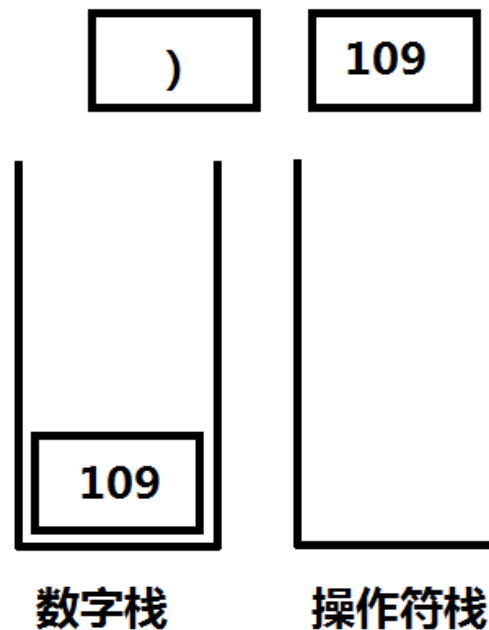
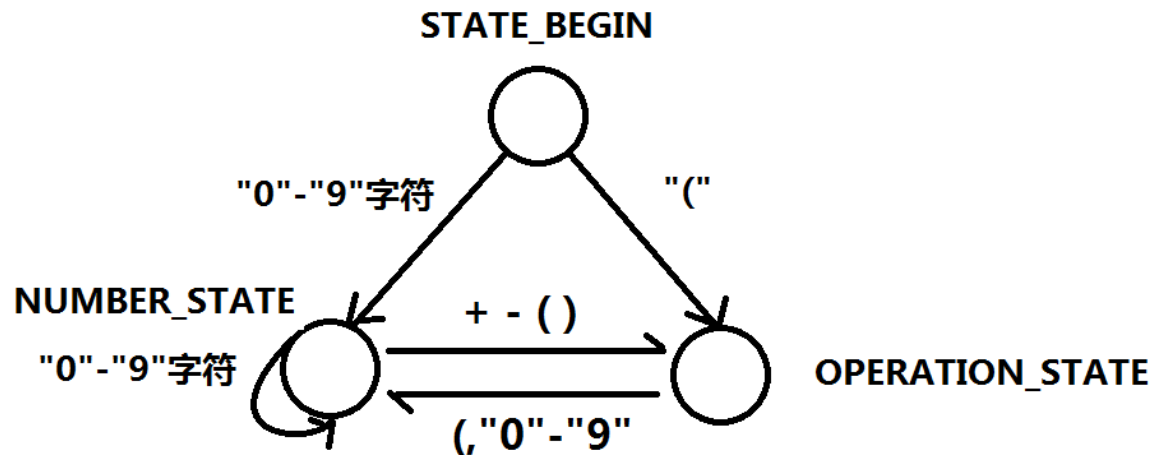


图3: compute_flag = 1



例5:字符串处理思路

$s = 1+121 - (14+(5-6))$



如果为数字字符:

$number = number * 10 + ch - '0';$

否则:

根据compute_flag进行计算

并切换至OPERATION_STATE

如果 + -

`operation_stack.push`

`compute_flag = 1`

如果 (

`compute_flag = 0`

切换至NUMBER_STATE

如果)

进行计算

如果0-9

切换至NUMBER_STATE

例5:实现，课堂练习

```
#include <string>
#include <stack>

class Solution {
public:
    int calculate(std::string s) {
        static const int STATE_BEGIN = 0;
        static const int NUMBER_STATE = 1;
        static const int OPERATION_STATE = 2;
        std::stack<int> number_stack;
        std::stack<char> operation_stack;
        int number = 0;
        int STATE = STATE_BEGIN;
        int compute_flag = 0;
        for (int i = 0; i < s.length(); i++) {
            if (s[i] == ' '){
                1
            }
            switch(STATE) {
            case STATE_BEGIN:
                if (s[i] >= '0' && s[i] <= '9'){
                    STATE = NUMBER_STATE;
                }
                else{
                    STATE = OPERATION_STATE;
                }
                2
            break;
            case NUMBER_STATE:
                if (s[i] >= '0' && s[i] <= '9'){
                    number = number * 10 + s[i] - '0';
                }
                else{
                    3
                    if (compute_flag == 1){
                        compute(number_stack, operation_stack);
                    }
                    number = 0;
                    i--;
                    STATE = OPERATION_STATE;
                }
                break;
            case OPERATION_STATE:
                if (s[i] == '+' || s[i] == '-'){
                    operation_stack.push(s[i]);
                }
                4
                else if (s[i] == '('){
                    5
                    compute_flag = 0;
                }
                else if (s[i] >= '0' && s[i] <= '9'){
                    STATE = NUMBER_STATE;
                    i--;
                }
                else if (s[i] == ')'){
                    compute(number_stack, operation_stack);
                }
                break;
            }
        }
        if (number != 0){
            number_stack.push(number);
            compute(number_stack, operation_stack);
        }
        if (number == 0 && number_stack.empty()){
            return 0;
        }
        return 6
    }
}
```

```
case NUMBER_STATE:
    if (s[i] >= '0' && s[i] <= '9'){
        number = number * 10 + s[i] - '0';
    }
    else{
        3
        if (compute_flag == 1){
            compute(number_stack, operation_stack);
        }
        number = 0;
        i--;
        STATE = OPERATION_STATE;
    }
    break;
case OPERATION_STATE:
    if (s[i] == '+' || s[i] == '-'){
        operation_stack.push(s[i]);
    }
    4
    else if (s[i] == '('){
        5
        compute_flag = 0;
    }
    else if (s[i] >= '0' && s[i] <= '9'){
        STATE = NUMBER_STATE;
        i--;
    }
    else if (s[i] == ')'){
        compute(number_stack, operation_stack);
    }
    break;
}
}
if (number != 0){
    number_stack.push(number);
    compute(number_stack, operation_stack);
}
if (number == 0 && number_stack.empty()){
    return 0;
}
return 6
}
```


例5:实现

```
#include <string>
#include <stack>

class Solution {
public:
    int calculate(std::string s) {
        static const int STATE_BEGIN = 0;
        static const int NUMBER_STATE = 1;
        static const int OPERATION_STATE = 2;
        std::stack<int> number_stack;
        std::stack<char> operation_stack;
        int number = 0;
        int STATE = STATE_BEGIN;
        int compute_flag = 0;
        for (int i = 0; i < s.length(); i++) {
            if (s[i] == ' '){
                continue;
            }
            switch(STATE){
            case STATE_BEGIN:
                if (s[i] >= '0' && s[i] <= '9'){
                    STATE = NUMBER_STATE;
                }
                else{
                    STATE = OPERATION_STATE;
                }
                i--;
                break;
            case NUMBER_STATE:
                if (s[i] >= '0' && s[i] <= '9'){
                    number = number * 10 + s[i] - '0';
                }
                else{
                    number_stack.push(number);
                    if (compute_flag == 1){
                        compute(number_stack, operation_stack);
                    }
                    number = 0;
                    i--;
                    STATE = OPERATION_STATE;
                }
                break;
            case OPERATION_STATE:
                if (s[i] == '+' || s[i] == '-'){
                    operation_stack.push(s[i]);
                    compute_flag = 1;
                }
                else if (s[i] == '('){
                    STATE = NUMBER_STATE;
                    compute_flag = 0;
                }
                else if (s[i] >= '0' && s[i] <= '9'){
                    STATE = NUMBER_STATE;
                    i--;
                }
                else if (s[i] == ')'){
                    compute(number_stack, operation_stack);
                }
                break;
            }
        }
        if (number != 0){
            number_stack.push(number);
            compute(number_stack, operation_stack);
        }
        if (number == 0 && number_stack.empty()){
            return 0;
        }
        return number_stack.top();
    }
};
```

```
case NUMBER_STATE:
    if (s[i] >= '0' && s[i] <= '9'){
        number = number * 10 + s[i] - '0';
    }
    else{
        number_stack.push(number);
        if (compute_flag == 1){
            compute(number_stack, operation_stack);
        }
        number = 0;
        i--;
        STATE = OPERATION_STATE;
    }
    break;
case OPERATION_STATE:
    if (s[i] == '+' || s[i] == '-'){
        operation_stack.push(s[i]);
        compute_flag = 1;
    }
    else if (s[i] == '('){
        STATE = NUMBER_STATE;
        compute_flag = 0;
    }
    else if (s[i] >= '0' && s[i] <= '9'){
        STATE = NUMBER_STATE;
        i--;
    }
    else if (s[i] == ')'){
        compute(number_stack, operation_stack);
    }
    break;
}
}
if (number != 0){
    number_stack.push(number);
    compute(number_stack, operation_stack);
}
if (number == 0 && number_stack.empty()){
    return 0;
}
return number_stack.top();
}
```

例5:计算函数

```
void compute(std::stack<int> &number_stack,  
             std::stack<char> &operation_stack) {  
    if (number_stack.size() < 2) {  
        return;  
    }  
    int num2 = number_stack.top();  
    number_stack.pop();  
    int num1 = number_stack.top();  
    number_stack.pop();  
    if (operation_stack.top() == '+') {  
        number_stack.push(num1 + num2);  
    }  
    else if (operation_stack.top() == '-') {  
        number_stack.push(num1 - num2);  
    }  
    operation_stack.pop();  
}
```

例5:测试与leetcode提交结果

```
int main() {  
    std::string s = "1+121 - (14+(5-6) )";  
    Solution solve;  
    printf("%d\n", solve.calculate(s));  
    return 0;  
}
```

Basic Calculator

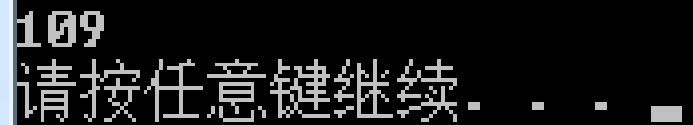
Submission Details

37 / 37 test cases passed.

Status: **Accepted**

Runtime: 16 ms

Submitted: 0 minutes ago



109
请按任意键继续. . .

预备知识:STL优先级队列(二叉堆)

二叉堆，最小(大)值先出的完全二叉树。

```
#include <stdio.h>
#include <queue>
int main() {
    std::priority_queue<int> big_heap; //默认构造是最大堆
    std::priority_queue<int, std::vector<int>, //最小堆构造方法
        std::greater<int> > small_heap;
    std::priority_queue<int, std::vector<int>, //最大堆构造方法
        std::less<int> > big_heap2;

    if (big_heap.empty()) {
        printf("big_heap is empty!\n");
    }
    int test[] = {6, 10, 1, 7, 99, 4, 33};
    for (int i = 0; i < 7; i++) {
        big_heap.push(test[i]);
    }
    printf("big_heap.top = %d\n", big_heap.top());
    1
    printf("big_heap.top = %d\n", big_heap.top());
    for (int i = 0; 2; i++) {
        3
    }
    printf("big_heap.top = %d\n", big_heap.top());
    printf("big_heap.size = %d\n", big_heap.size());
    return 0;
}
```

```
big_heap is empty!
big_heap.top = 99
big_heap.top = 1000
big_heap.top = 10
big_heap.size = 5
请按任意键继续. . .
```

big_heap.empty() : 判断堆是否为空

big_heap.pop() : 弹出堆顶元素(最大值)

big_heap.push(x) : 将元素x添加至二叉堆

big_heap.top() : 返回堆顶元素(最大值)

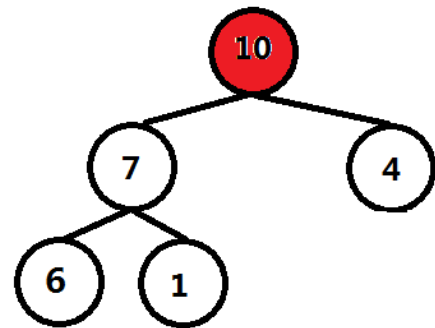
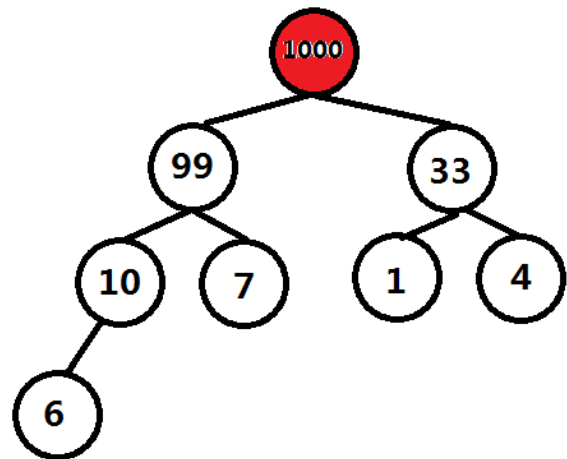
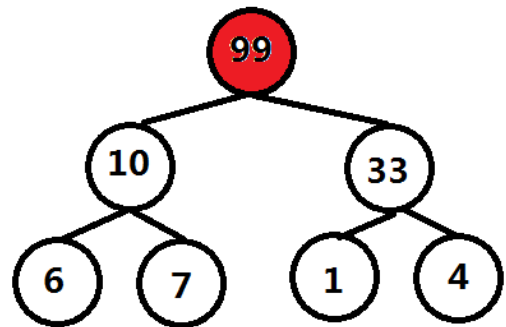
big_heap.size() : 返回堆中元素个数

预备知识:STL优先级队列

```
#include <stdio.h>
#include <queue>
int main(){
    std::priority_queue<int> big_heap; //默认构造是最大堆
    std::priority_queue<int, std::vector<int>, //最小堆构造方法
                        std::greater<int> > small_heap;
    std::priority_queue<int, std::vector<int>, //最大堆构造方法
                        std::less<int> > big_heap2;

    if (big_heap.empty()){
        printf("big_heap is empty!\n");
    }
    int test[] = {6, 10, 1, 7, 99, 4, 33};
    for (int i = 0; i < 7; i++){
        big_heap.push(test[i]);
    }
    printf("big_heap.top = %d\n", big_heap.top());
    big_heap.push(1000);
    printf("big_heap.top = %d\n", big_heap.top());
    for (int i = 0; i < 3; i++){
        big_heap.pop();
    }
    printf("big_heap.top = %d\n", big_heap.top());
    printf("big_heap.size = %d\n", big_heap.size());
    return 0;
}
```

```
big_heap is empty!
big_heap.top = 99
big_heap.top = 1000
big_heap.top = 10
big_heap.size = 5
请按任意键继续. . .
```



例6:数组中第K大的数

已知一个**未排序**的**数组**，求这个数组中**第K大**的数字。

如，array = [3,2,1,5,6,4]，k = 2，return 5

```
class Solution {  
public:  
    int findKthLargest(std::vector<int>& nums, int k) {  
    }  
};
```

选自 **LeetCode 215. Kth Largest Element in an Array**

<https://leetcode.com/problems/kth-largest-element-in-an-array/description/>

难度:**Easy**

例6:思路

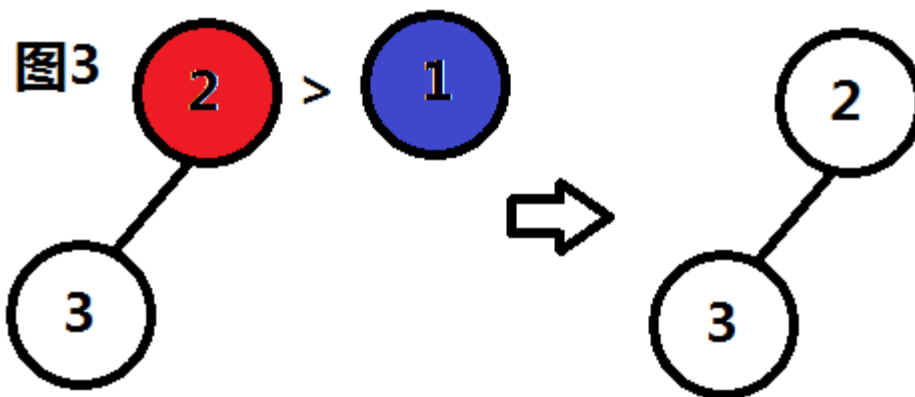
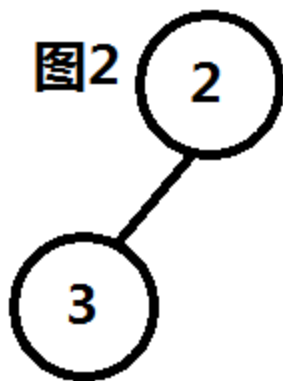
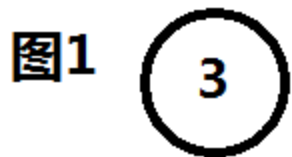
维护一个**K大小**的**最小堆**，堆中元素个数小于K时，新元素**直接**进入堆；否则，当堆顶**小于**新元素时，弹出堆顶，将**新元素加入**堆。

解释：

由于堆是**最小堆**，堆顶是堆中**最小**元素，**新元素**都会保证**比堆顶小**(否则新元素替换堆顶)，故堆中K个元素是已扫描的元素里**最大的K个**；堆顶即为**第K大**的数。

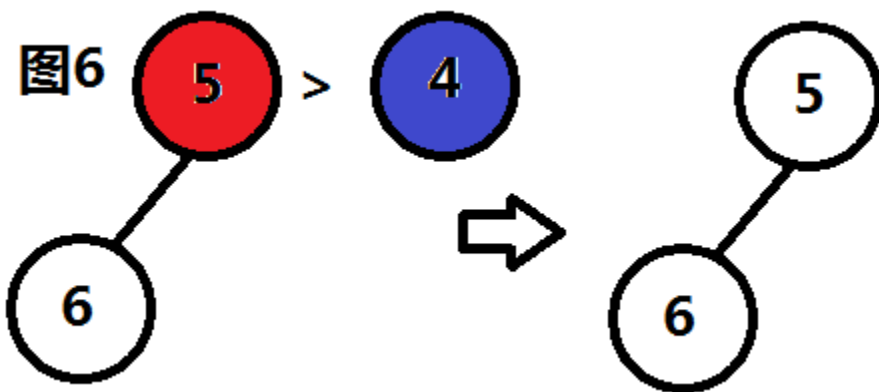
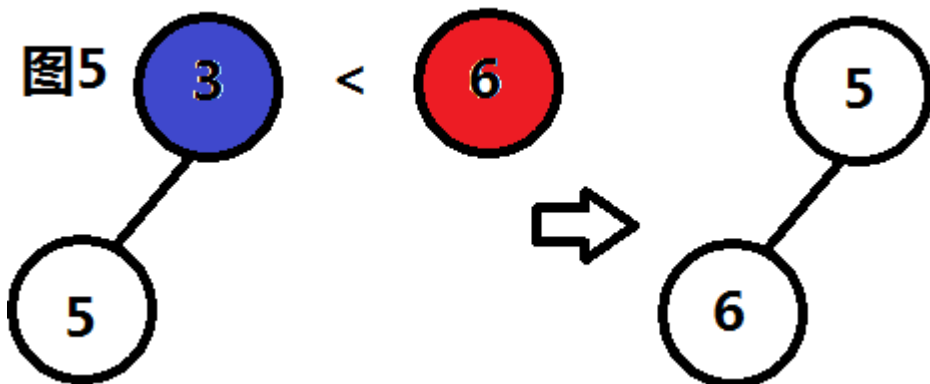
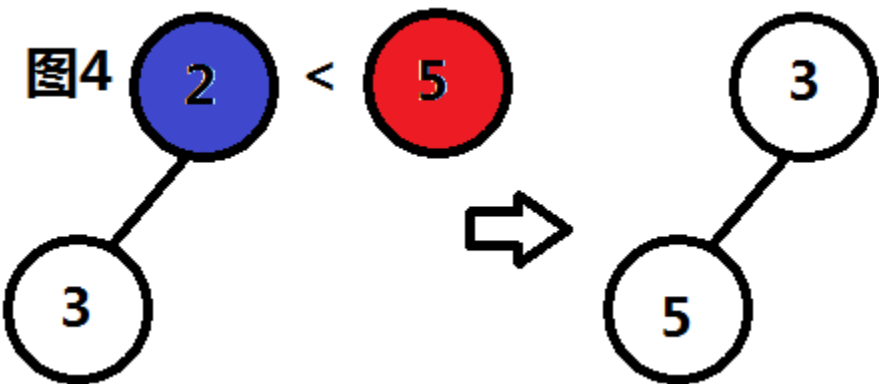
设数组长度为N，求第K大的数，时间复杂度： $N * \log K$

如，array = [3,2,1,5,6,4]：

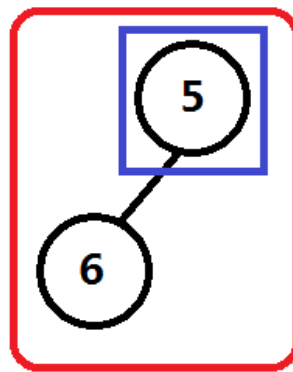


例6:思路

如, $\text{array} = [3, 2, 1, 5, 6, 4]$:



6(N)个数中最大的2(K)个



2(K)个数中最小的,
即6(N)个数第2(K)大的

例6:实现，课堂练习

```
#include <vector>
#include <queue>
class Solution {
public:
    int findKthLargest(std::vector<int>& nums, int k) { //最小堆
        std::priority_queue<int, std::vector<int>, std::greater<int>> > Q;
        for (int i = 0; i < nums.size(); i++) { //遍历nums数组
            if (1) {
                Q.push(nums[i]);
            }
            else if (2) {
                Q.pop();
                3
            }
        }
        return Q.top(); //返回堆顶
    }
};
```

3分钟时间填写
代码，
有问题随时
提出！

例6:实现

```
#include <vector>
#include <queue>
class Solution {
public:
    int findKthLargest(std::vector<int>& nums, int k) { //最小堆
        std::priority_queue<int, std::vector<int>, std::greater<int> > Q;
        for (int i = 0; i < nums.size(); i++) { //遍历nums数组
            if (Q.size() < k) {
                Q.push(nums[i]); //如果堆中元素个数小于k, 直接push进入堆
            }
            else if (Q.top() < nums[i]) { //如果堆顶比新元素小, 弹出堆顶
                Q.pop(); //push进入新元素(即替换堆顶)
                Q.push(nums[i]);
            }
        }
        return Q.top(); //返回堆顶
    }
};
```

例6:测试与leetcode提交结果

```
int main() {  
    std::vector<int> nums;  
    nums.push_back(3);  
    nums.push_back(2);  
    nums.push_back(1);  
    nums.push_back(5);  
    nums.push_back(6);  
    nums.push_back(4);  
    Solution solve;  
    printf("%d\n", solve.findKthLargest(nums, 2));  
    return 0;  
}
```

Kth Largest Element in an Array

Submission Details

31 / 31 test cases passed.

Status: **Accepted**

Runtime: 9 ms

Submitted: 0 minutes ago

5

请按任意键继续. . .

例7:寻找中位数

设计一个数据结构，该数据结构**动态**维护一组数据，且**支持**如下操作：

1. **添加**元素: void addNum(int num), 将整型num添加至数据结构中。
2. 返回数据的**中位数**: double findMedian(), 返回其维护的数据的**中位数**。

中位数定义:

1. 若数据个数为奇数，中位数是该组数排序后中间的数。[1,2,3] -> 2
2. 若数据个数为偶数，中位数是该组数排序后中间的两个数字的平均值。[1,2,3,4] -> 2.5

```
class MedianFinder {
public:
    MedianFinder() {
    } //向数据结构中添加一个整数
    void addNum(int num) {
    } //返回该数据结构中维护的数据的中位数
    double findMedian() {
    }
};

int main() {
    MedianFinder M;
    M.addNum(2);
    M.addNum(1); //返回1.5
    printf("%lf\n", M.findMedian());
    M.addNum(4); //返回2
    printf("%lf\n", M.findMedian());
    M.addNum(3); //返回2.5
    printf("%lf\n", M.findMedian());
    return 0;
}
```

选自 **LeetCode 295. Find Median from Data Stream**

<https://leetcode.com/problems/find-median-from-data-stream/description/>

难度:**Hard**

例7:思考:如何获取中位数

最**直观**的方法:

存储结构使用**数组**，每次**添加元素**或**查找中位数**时对数组**排序**，再计算结果。

时间复杂度:

- 1.若**添加元素时排序**，addNum复杂度 $O(n)$ ，findMedian复杂度 $O(1)$
- 2.若**查询中位数时排序**，addNum复杂度 $O(1)$ ，findMedian复杂度 $O(n\log n)$

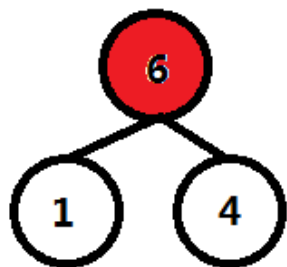
若添加元素或查询中位数是**随机**的操作，共 n 次操作，按上述思想，整体复杂度**最佳为** $O(n^2)$

是否还有**更好方法**? 思考1分钟!

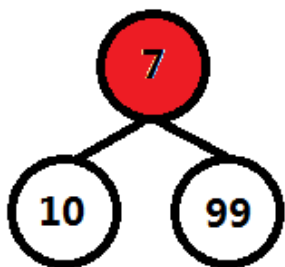
例7:思路:巧用堆的性质

动态维护一个**最大堆**与一个**最小堆**，最大堆存储一半数据，最小堆存储一般数据，**维持**最大堆的堆顶比最小堆的堆顶小。

最大堆:

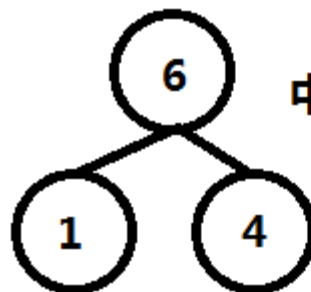


最小堆:

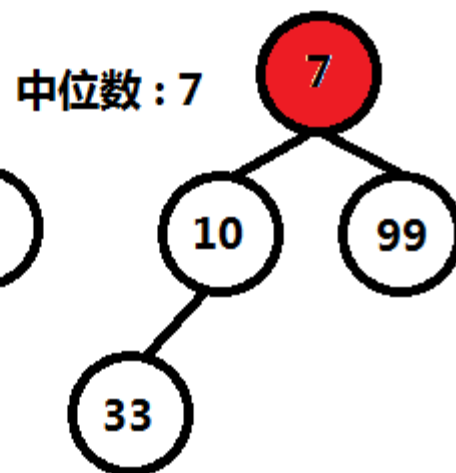


中位数: $(6 + 7)/2 = 6.5$

最大堆:

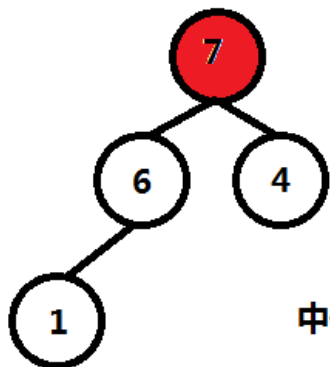


最小堆:

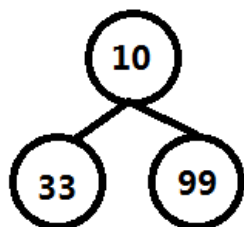


中位数: 7

最大堆:



最小堆:



中位数: 7

例7:思路:添加元素时堆调整1

情况1:

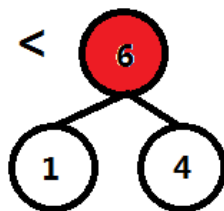
最大堆与最小堆元素个数相同时:

新元素:

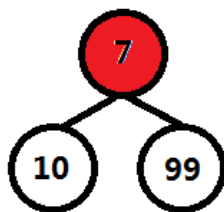


<

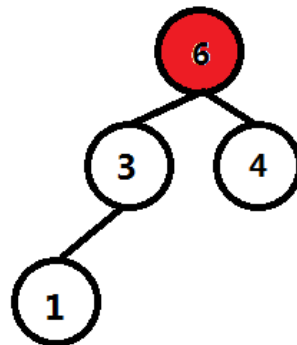
最大堆:



最小堆:



将3push进入最大堆

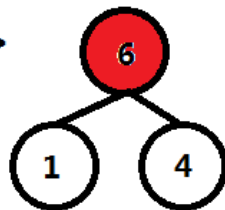


新元素:

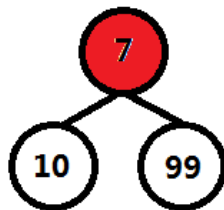


>

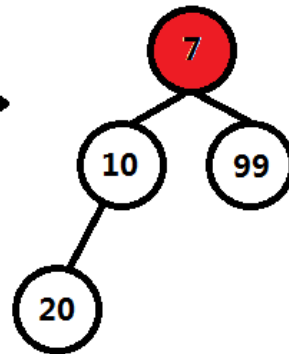
最大堆:



最小堆:



将20 push进入最小堆

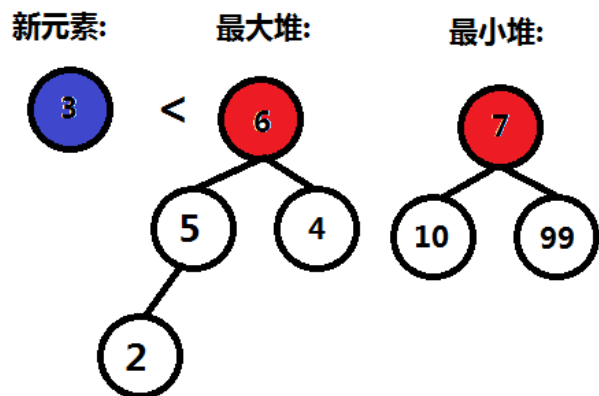


例7:思路:添加元素时堆调整2

情况2:

最大堆比最小堆多一个元素:

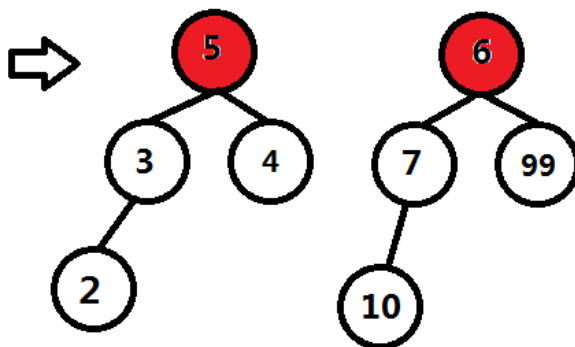
a. 如果新元素小于最大堆堆顶:



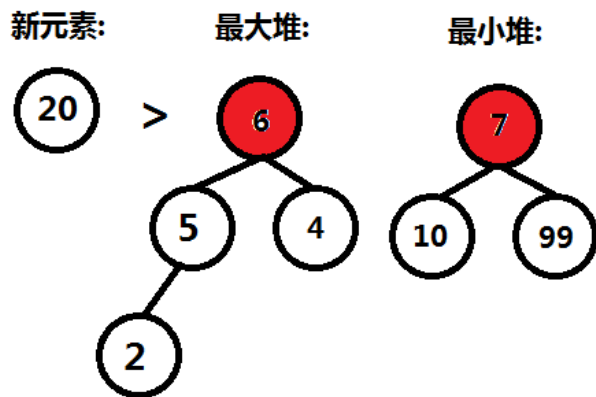
将最大堆的堆顶push进入最小堆

将最大堆的堆顶移除(pop)

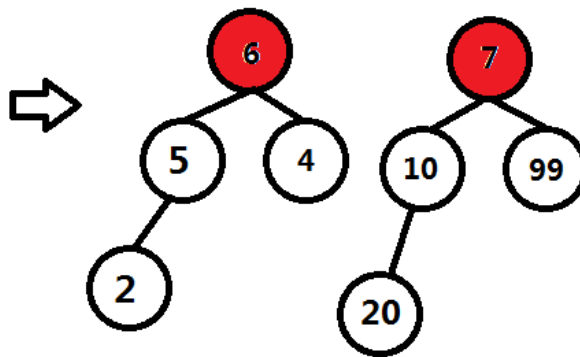
将新元素添加至最大堆



b. 如果新元素大于最大堆堆顶:



将新元素直接push进入最小堆



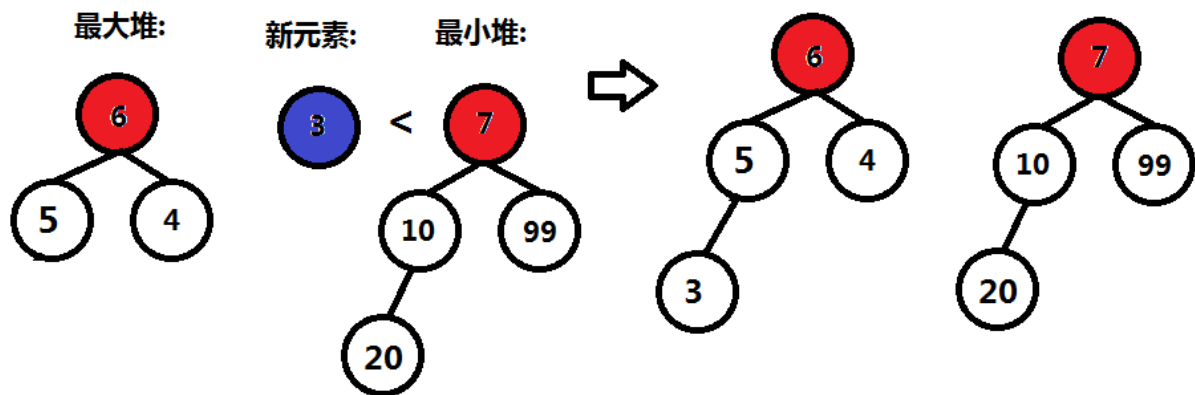
例7:思路:添加元素时堆调整3

情况3:

最大堆比最小堆少一个元素:

a. 如果新元素小于最小堆堆顶:

将新元素直接push进入最大堆

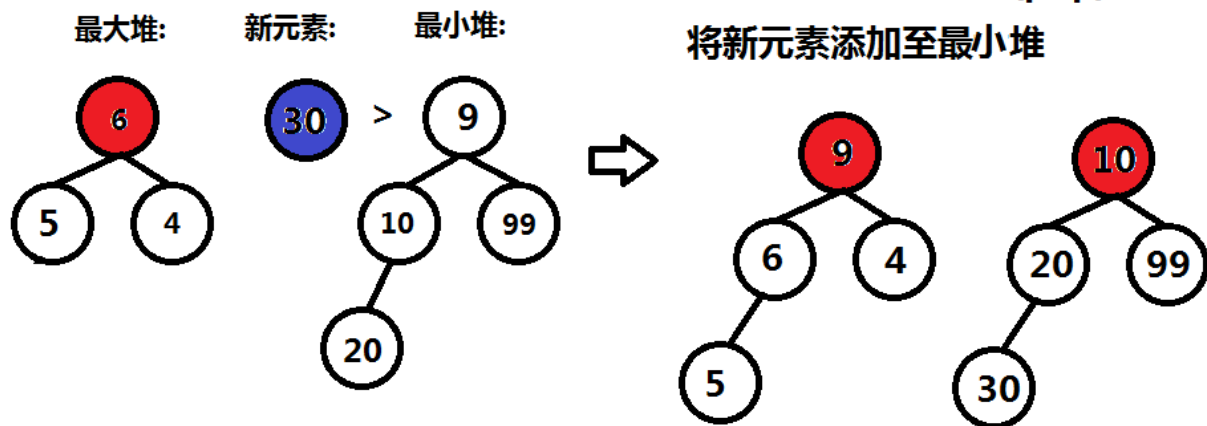


b. 如果新元素大于最小堆堆顶:

将最小堆的堆顶push进入最大堆

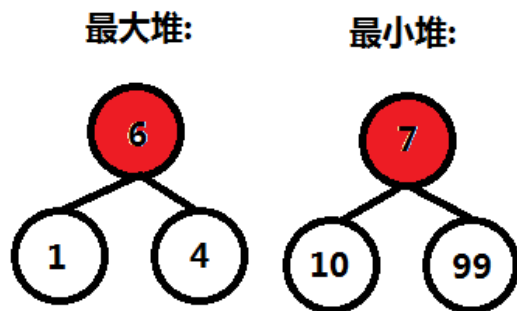
将最小堆的堆顶移除(pop)

将新元素添加至最小堆



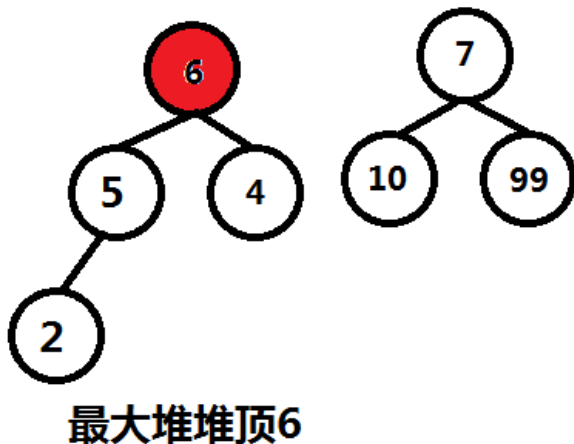
例7:思路:获取中位数

a. 最大堆最小堆中的元素个数相同时

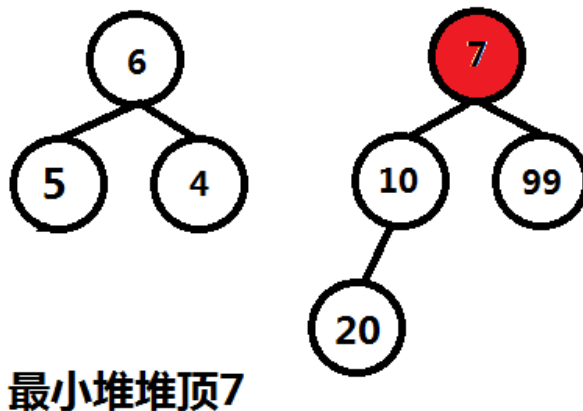


最大堆堆顶与最小堆堆顶的平均值: $(6+7)/2 = 6.5$

b. 最大堆比最小堆多一个元素:



c. 最大堆比最小堆少一个元素:



```
void addNum(int num) {  
    if (big_queue.empty()) { //big_queue最大堆  
        big_queue.push(num); //small_queue最小堆  
        return;  
    }
```

```
    if (big_queue.size() == small_queue.size()) {  
        if (1) {  
            big_queue.push(num);  
        }  
        else {  
            small_queue.push(num);  
        }  
    }  
    else if (2) {  
        if (num > big_queue.top()) {  
            small_queue.push(num);  
        }  
        else {  
            3  
            big_queue.pop();  
            big_queue.push(num);  
        }  
    }  
    else if (4) {  
        if (5) {  
            big_queue.push(num);  
        }  
        else {  
            big_queue.push(small_queue.top());  
            small_queue.pop();  
            small_queue.push(num);  
        }  
    }  
}
```

例7:实现, 课堂练习

5分钟时间填写
代码,
有问题随时
提出!

例7:实现

```
void addNum(int num) {  
    if (big_queue.empty()) { //big_queue最大堆  
        big_queue.push(num); //small_queue最小堆  
        return;  
    }  
    if (big_queue.size() == small_queue.size()) {  
        if (num < big_queue.top()) {  
            big_queue.push(num);  
        }  
        else {  
            small_queue.push(num);  
        }  
    }  
    else if (big_queue.size() > small_queue.size()) {  
        if (num > big_queue.top()) {  
            small_queue.push(num);  
        }  
        else {  
            small_queue.push(big_queue.top());  
            big_queue.pop();  
            big_queue.push(num);  
        }  
    }  
    else if (big_queue.size() < small_queue.size()) {  
        if (num < small_queue.top()) {  
            big_queue.push(num);  
        }  
        else {  
            big_queue.push(small_queue.top());  
            small_queue.pop();  
            small_queue.push(num);  
        }  
    }  
}
```

例7:实现， 课堂练习

```
double findMedian() {  
    if ( 1 ) {  
        return (big_queue.top() + small_queue.top()) / 2;  
    }  
    else if ( 2 ) {  
        return big_queue.top();  
    }  
    return 3  
}
```

例7:实现

```
double findMedian() {  
    if ( big_queue.size() == small_queue.size() ) {  
        return (big_queue.top() + small_queue.top()) / 2;  
    }  
    else if ( big_queue.size() > small_queue.size() ) {  
        return big_queue.top();  
    }  
    return small_queue.top();  
}
```

例7:测试与leetcode提交结果

```
int main() {
    MedianFinder M;
    int test[] = {6, 10, 1, 7, 99, 4, 33};
    for (int i = 0; i < 7; i++) {
        M.addNum(test[i]);
        printf("%lf\n", M.findMedian());
    }
    return 0;
}
```

Find Median from Data Stream

Submission Details

18 / 18 test cases passed.

Status: **Accepted**

Runtime: 156 ms

Submitted: 0 minutes ago

6.000000

8.000000

6.000000

6.500000

7.000000

6.500000

7.000000

请按任意键继续. . .

结束

非常感谢大家！

林沐