# 定义

栈是后进先出（LIFO=Last In First Out）的特殊的线性表，限定为只能在一端（表尾）进行插入和删除操作。

注：对于栈来说，这个表尾称为栈的栈顶（top），相应的表头称为栈底（bottom）。



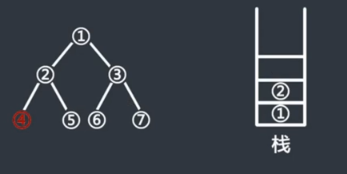
注：静态内存分配是借助栈（操作系统控制），动态内存分配是借助堆（程序员控制）。

**深度优先遍历可以使用栈实现，广度/宽度优先遍历可以使用队列实现**。

## DFS/BFS

深度优先遍历可以用栈实现：









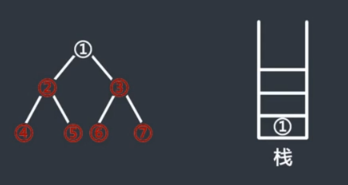






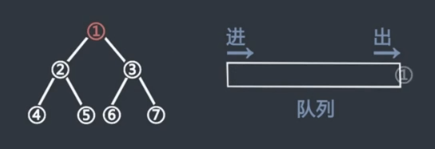




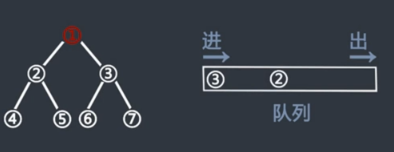


宽优先遍历可以用队列实现：

节点1进队列，出队列：



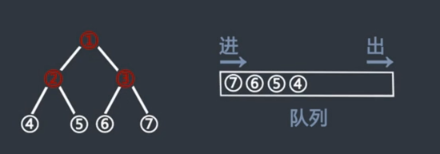
节点1的孩子2、3依次入队：



节点2出队列，并将其孩子4、5依次入队：



节点3出队列，将其孩子6、7依次入队：



4、5、6、7依次出队列：



## 栈与递归

栈与递归：平时使用的递归函数实际上用到了提供的函数系统栈，递归的过程可以看作递归函数依次进入函数栈的处理过程。所有用递归函数可以做到的过程都一定可以用非递归的方式实现。

# 单调栈

## 栈

### 定义

定义：栈是一种后进先出（LIFO）的线性数据结构，仅允许在栈顶进行插入（push）和删除（pop）操作。

### 核心特性

操作受限：仅栈顶可操作，确保数据访问顺序的严格性。

时间复杂度：入栈、出栈、取栈顶元素均为 O(1)。

结构简单：可通过数组或链表实现，支持顺序存储（顺序栈）和链式存储（链式栈）。

### 典型应用场景

函数调用堆栈：管理函数调用与返回顺序，确保最近调用的函数最先返回。

括号匹配：检测最近打开的括号是否最先闭合（如 ()、[]、{}）。

表达式求值：处理中缀表达式（如 3 + 5 \* 2）的运算符优先级和临时存储。

浏览器后退功能：通过压栈实现访问记录的逆序回退。

编辑器撤销操作：保存操作状态到栈中，支持撤销和恢复。

## 单调栈（Monotonic Stack）

### 定义

定义：单调栈是栈的扩展，通过维护栈内元素的单调性（递增或递减）来高效解决特定问题。

作用：记录遍历过哪些元素，从而便于进行其他操作。

### 分类

单调递增栈：栈底到栈顶元素递增，用于查找右侧第一个更大元素。

单调递减栈：栈底到栈顶元素递减，用于查找右侧第一个更小元素。

### 核心特性

维护单调性：新元素入栈前，需弹出所有破坏单调性的栈顶元素。

时间复杂度：遍历数组一次，时间复杂度为 O(n)，通过空间换时间优化效率。

应用场景：解决数组中元素关系查询问题，如寻找下一个更大/更小元素、柱状图最大矩形等。

### 工作过程

### 典型应用案例

#### 下一个更大元素

问题描述：给定数组，找出每个元素右侧第一个比它大的元素。

示例：数组 [5, 3, 8, 2, 7, 1] 的下一个更大元素为 [8, 8, -1, 7, -1, -1]。

解决方案：

使用单调递减栈，遍历数组时维护栈的单调性。

当前元素大于栈顶元素时，弹出栈顶并记录结果。

代码：

#include <vector>

#include <stack>

using namespace std;

vector<int> nextGreaterElements(vector<int>& nums) {

int n = nums.size();

vector<int> res(n, -1);

stack<int> stk; // 存储元素索引

// 遍历两次数组处理循环

for (int i = 0; i < 2\*n; ++i) {

int idx = i % n;

while (!stk.empty() && nums[idx] > nums[stk.top()]) {

res[stk.top()] = nums[idx];

stk.pop();

}

if (i < n) stk.push(idx); // 只在第一次遍历时压栈

}

return res;

}

/\* 示例：

输入：[1,2,1]

输出：[2,-1,2]

\*/

#### 柱状图最大矩形

问题描述：给定柱状图高度数组，求最大矩形面积。

示例：输入 [2, 1, 5, 6, 2, 3]，最大矩形面积为 10。

解决方案：

在数组首尾添加 0，维护单调递增栈。

计算每个柱子的左右边界，确定矩形宽度和高度。

代码：

#include <vector>

#include <stack>

using namespace std;

int largestRectangleArea(vector<int>& heights) {

int n = heights.size();

vector<int> h(n+2, 0); // 添加哨兵

copy(heights.begin(), heights.end(), h.begin()+1);

stack<int> stk; // 存储递增柱子的索引

int max\_area = 0;

for (int i = 0; i < n+2; ++i) {

while (!stk.empty() && h[i] < h[stk.top()]) {

int height = h[stk.top()];

stk.pop();

int width = i - stk.top() - 1;

max\_area = max(max\_area, height \* width);

}

stk.push(i);

}

return max\_area;

}

/\* 示例：

输入：[2,1,5,6,2,3]

输出：10（对应5和6构成的矩形）

\*/

#### 接雨水问题

问题描述：给定高度数组，计算能接多少雨水。

示例：输入 [0,1,0,2,1,0,1,3,2,1,2,1]，可接 6 单位雨水。

解决方案：

使用单调递减栈记录柱子索引，计算相邻柱子间的积水面积。

代码：

#include <vector>

#include <stack>

using namespace std;

int trap(vector<int>& height) {

stack<int> stk; // 存储递减柱子的索引

int res = 0;

for (int i = 0; i < height.size(); ++i) {

while (!stk.empty() && height[i] > height[stk.top()]) {

int bottom = stk.top();

stk.pop();

if (stk.empty()) break;

int left = stk.top();

int width = i - left - 1;

int h = min(height[left], height[i]) - height[bottom];

res += width \* h;

}

stk.push(i);

}

return res;

}

/\* 示例：

输入：[0,1,0,2,1,0,1,3,2,1,2,1]

输出：6（对应多个凹陷区域）

\*/

### 数学基础与正确性证明

单调性维护：

以单调递增栈为例，若当前元素大于栈顶元素，则栈顶元素的右侧更大元素已找到，可安全弹出。

此过程确保栈内元素始终递增，当前元素入栈后继续维护单调性。

时间复杂度证明：

每个元素最多入栈和出栈一次，总操作次数为 O(n)，故时间复杂度为 O(n)。

## 区别与联系

### 核心区别

表格

描述已自动生成

### 内在联系

继承关系：单调栈是栈的扩展应用，继承了栈的 LIFO 特性。

优化目标：两者均用于解决特定问题，但单调栈通过约束元素顺序，将暴力解法的 O(n²) 优化至 O(n)。

## 总结

栈是基础数据结构，强调 LIFO 特性，适用于通用数据操作场景。

单调栈通过维护元素单调性，高效解决数组中元素关系查询问题，是算法优化中的关键工具。

应用选择：简单数据操作选普通栈；需要快速查找元素关系时，优先使用单调栈。

通过理解两者的核心差异与联系，可灵活选择数据结构，提升算法效率。

# 分类

因为栈的本质是一个线性表，线性表有两种存储形式，那么栈也有顺序存储结构和链式存储结构。

按照栈的存储方式可以分为：

## 静态栈

以数组为存储单元。

## 动态栈

以链表实现的栈。

# 操作

## 出栈

栈的删除操作(Pop)，叫做出栈，也称为弹栈。

## 压栈

栈的插入操作(Push)，叫做进栈，也称为压栈，入栈。

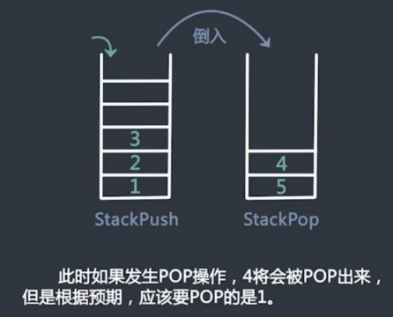
注：**入栈出栈应用于生产者-消费者模型**。

**需要注意的地方：**

1. 如果StackPush要往StackPop中倒入数据，那么必须要把StackPush中的所有数据一次性倒完；
2. 如果StackPop中有数据，则不能发生倒数据的行为。

违反注意点1，会发生错误：





违反注意点2，发生错误：







# 存储

## 顺序存储

## 链式存储

# 实现

栈这种抽象数据结构，最常见的有3种实现方式：

1. 用简单的数组实现
2. 用动态数组实现
3. 用链表实现

## 简单数组

用数组实现栈的时候，需要按照从左向右的顺序添加元素，并且要用变量记录栈顶元素所对应的数组下标。

用来存放栈元素的数组可能会占满，此时如果用户还要执行push操作，那么程序就应该抛出full stack exception。同理，数组也有可能处于不含元素的状态，此时如果用户执行pop操作，那么应该抛出stack empty exception。

### 定义

typedf struct{

int data[MAX\_SIZE];//数据域（简单数组存储，大小固定）

int top;//指针域

int stackSize; //可以不添加这个变量

}SqStack;

### 空间判断

#### 空栈

方式一：

判断栈是否为空：

int IsEmptyStack(struct SqStack \*S){

return (s->top==-1);//如果条件为true，返回1；否则返回0

}

或者：

int IsEmptyStack(struct SqStack \*S){

if(S->top == -1){

return 1;

}else{

return 0;

}

}

#### 溢出

判断栈是否溢出：

int IsFullStack(struct SqStack \*S){

return (S->top == S->capacity-1);

}

或：

int IsFullStack(struct SqStack \*S){

if(S->top == s->capacity-1){

return 1;

}else{

return 0;

}

}

### 获取元素

int SqStackGetElement(SqStack \*s, int \*element){

if(s->top == -1){

return -1;

}

\*element=s->data[s->top];

return 0;

}

### 入栈

思路：插入需要判断是不是越界，即top指针域MAX\_SIZE对比大小

代码：

int SqStackPush(Stack \*s, int \*element){

if(s->top == MAX\_SIZE-1){

return -1;

}

s->top++;//top指针先自加1，然后赋值

s->data[s->top]=\*element;

return 0;

}

### 出栈

思路：删除栈的元素需要考虑栈是否为空

代码：

int SqStackPop(SqStack \*s, int \*element){

if(s->top == -1){

return -1;

}

\*element=s->data[s->top];

s->top--;

return 0;

}

### 分析

性能：

如果把栈中的元素数量设为n，那么用这种方式实现的栈，其各项操作的复杂度如下：

|  |  |
| --- | --- |
| 空间复杂度（执行n次push操作） | O(n) |
| Push()操作的时间复杂度 | O(1) |
| Pop()操作的时间复杂度 | O(1) |
| Size()操作的时间复杂度 | O(1) |
| IsEmptyStack()操作的时间复杂度 | O(1) |
| IsFullStack()操作的时间复杂度 | O(1) |
| DeleteStack()操作的时间复杂度 | O(1) |

局限：

栈的最大容量必须提前定好（即设置好capacity），并且不能修改。

一般来说，预先分配数组大小并不是个问题。在典型的应用程序中，即使有相当多的栈操作，在任一时刻栈元素的实际个数从不会太大。声明一个数组足够大而不至于浪费太多的空间通常并没有什么困难。如果不能做到这一点，那么节省的做法是使用链表来实现。

## 动态数组

注：栈最重要的是动态数组的分配方式，因为一般使用的时候，栈不会设置过长，链表存储相对少。

### 定义

**方式一：**

struct ArrayStack{

int \*array; //数据域（内存地址是连续的，既可以执行array[i]++操作）

int top; //指针域

int capacity; //需要定义这个容量，放置溢出（链式存储不需要）

};

注：采用这种方式定义，后面使用结构体的时候需要带上struct关键字，如果不想带struc关键字，则使用typedef定义一个名称即可。

**方式二：**

typedef struct

{

ElemType \*base;

ElemType \*top;

int stackSize;

}SqStack;//（《妙趣横生的算法（C语言实现）》实现方式）

该线性表包含了三个元素：base，top，stackSize。其中base是指向栈底的指针变量，top是指向栈顶的指针变量，stackSize指示栈的当前可使用的最大容量，可以通过base和top对栈进行各种操作，通过stackSize判断栈的空间分配情况。

这里没有上述两种方式的数据域，对于栈其实不需要存储每个数据，top指针执行数据的push、pop操作，只需要一个top指针即可。

注：这里存储数据直接定义一个数组，这种方式是将大小写死了，还可以采用指针的形式定义，即int \*data。

### 初始化

方法一：

struct ArrayStack \*CreateStack(){

struct ArrayStack \*S = malloc(sizeof(struct ArrayStack)); //为结构体分配内存

if(!S)

return NULL:

S->capacity = 1; //定义数组的大小

S->top = -1;

S->array = malloc(S->capacity \* sizeof(int)); //先分配1个int的大小

if(!S->array)

return NULL;

return S;

}

方式二：

#define STACK\_INIT\_SIZE 100

initStack(sqStack \*s)

{

s->base = (ElemType \*)malloc(STACK\_INIIT\_SIZE \* sizeof(ElemType));

//内存中开辟一段连续空间作为栈空间，首地址赋给s->base

if( !s->base)

exit(0);

s->top = s->base; //最开始，栈顶就是栈底

s->stackSize = STACK\_INIT\_SIZE;

}

### 空间判断

#### 空栈

判断栈是否为空：

int IsEmptyStack(struct ArrayStack \*S){

return (S->top==-1);//如果条件为true，返回1；否则返回0

}

或者：

int IsEmptyStack(struct ArrayStack \*S){

if(S->top == -1){

return 1;

}else{

return 0;

}

}

#### 溢出

判断栈是否溢出：

int IsFullStack(struct ArrayStack \*S){

return (S->top == S->capacity-1);

}

或者：

int IsFullStack(structArrayStack \*S){

if(S->top == s->capacity-1){

return 1;

}else{

return 0;

}

}

### 容量

方式一：

int StackLen(ArrayStack s)

{

return s.capacity;

}

方式二：

int StackLen(sqStack s)

{

return (s.top – s.base);

}

### 翻倍

void DoubleStack(struct ArrayStack \*S){

S->capacity \*= 2;

S->array = realloc(S->array, S->capacity\*sizeof(int));

}

### 入栈

方式一：

void Push(struct ArrayStack \*S, int data){

if(IsFullStack(S)){

printf(“Stack overflow!”);

}else{

S->array[++S->top] = data; //top先自增，腾出一个空间放新的元素（仍然是数组形式赋值）

}

}

或：

void Push(struct ArrayStack \*S, int data){

if(IsFullStack(S)){

DoubleStack(S); //此时不报错直接翻倍

S->array[++S->top] = data;

}

}

方式二：

#define STACKINCREMENT 10

Push(sqStack \*s, ElemType e)

{

//如果栈满，追加空间

if(s->top - s->base >= s->stackSize)

{

s->base = (ElemType \*)realloc(

s->base, (s->stackSize+STACKINCREMENT)\*sizeof(ElemType));

}

if( !s->base )

exit(0);

s->top = s->base + s->stackSize; //设置栈顶指针（通过base分配内存，然后定位栈顶）

s->stackSize = s->stackSize + STACKINCREMENT; //设置栈的最大容量

\*(s->top) = e; //存储栈顶元素

s->top++;

}

### 出栈

方式一：

int Pop(struct ArrayStack \*S){

if(IsEmptyStack(S)){

printf(“Stack is Empty!”);

// return INT\_MIN;//（如果空栈，可以返回错误或者INT最小值）

}else{

return (S->array[S->top--]);

}

}

方式二：

Pop(sqStack \*s, ElemType \*e)

{

if(s->top == s->base) //栈空

return;

\*e = \*--(s->top);

}

### 清空栈

所谓清空一个栈，就是将栈中的元素全部作废，但栈本身物理空间并不发生改变（不是销毁）。

方式一：

ClearStack(sqStack \*s)

{

s->top = NULL;

s->stackSize = 0；

}

方法二（采用base和top定义栈的情况）：

ClearStack(sqStack \*s)

{

s->top = s->base;

}

注：因此，我们只要将s->top的内容赋值给s->base即可，这样s->base等于s->top，也就表名这个栈是空的了。这个原理跟高级格式化只是单纯地清空文件列表而没有覆盖硬盘的原理是一样的。

### 删除栈

方式一：

void DeleteStack(struct ArratStack \*S){

if(S){

if(S->array)

free(S->array);

free(S);

}

}

方式二：

销毁一个栈与清空一个栈不同，销毁一个栈是要释放掉该栈所占据的物理内存空间，因此不要把销毁一个栈与清空一个栈两种操作混淆。

DestroyStack(sqStack \*s)

{

int i,len;

len = s->stackSize;

for( i=0;i<len;i++)

{

free(s->base);

s->base++; //在需要销毁栈的时候，就要引入指向队首的指针base

}

s->base = s->top = NULL;

s->stackSize = 0;

}

### 分析

性能：

如果把栈中的元素数量设为n，那么用这种方式实现的栈，其各项操作的复杂度如下：

|  |  |
| --- | --- |
| 空间复杂度（执行n次push操作） | O(n) |
| CreateStack()操作的时间复杂度 | O(1) |
| Push()操作的时间复杂度 | O(1)（平均） |
| Pop()操作的时间复杂度 | O(1) |
| Top()操作的时间复杂度 | O(1) |
| IsEmptyStack()操作的时间复杂度 | O(1) |
| IsFullStack()操作的时间复杂度 | O(1) |
| DeleteStack()操作的时间复杂度 | O(1) |

局限：

翻倍操作执行得过多，可能会导致程序发生内存溢出异常（memory overflow ecxceptiion）。

## 链表

### 定义

**方式一**（这是将top作为操作函数入参使用，定义的时候没有设置）：

struct LinkStack {

int data;//数据域

struct Stack \*next;//指针域

};

LinkStack \*top;

**方式二（推荐）：**

typedef struct LinkStackNode{

int data;//数据域

stuct LinkStackNode \*next;//指针域

}LinkStackNode \*LinkStackPtr;

typedef struct LinkStack{

LinkStackPtr top;

LinkStackPtr bottom; //栈底部（可选）

int count;//栈元素计数器（可选，链表存储不需要担心溢出问题）

}LinkStack;//定义时采用指针类型

注：栈、队列等都是在链表的Node基础上增加一个指针域（这里是top）实现的。

如下：

typedef struct LinkStackNode{

int data;

struct LinkStackNode \*next;

}LinkStackNode \*LinkStackPtr;

LinkStackPtr p;//这种定义的是指针类型，注意与前面的区别

### 初始化

**方式一：**

struct LinkStack \*CreateLinkStack (){

return NULL;

}

**方式二：**

void init(LinkStackPtr \*l)

{

l->top = (LinkStackPtr)malloc(sizeof\*(LinkStackNode));

if (NULL == l->top)

{

exit(-1);

}

else

{

l->bottom = l->top;

l->top->next = NULL;

}

}

### 空间判断

**方式一：**

判断栈是否为空：

int IsEmptyLinkStack (struct LinkStack\*top){

return top ==NULL;

}

**方式二：**

通过top==bottom判断是否栈为空

### 获取元素

int LinkStackGetElement(LinkStack \*l, int \*element){

//这个是定义LinkList \*l而不是LinkList l，与前面链表的定义有区别

LinkStackPtr s=(LinkStackPtr)malloc(sizeof\*(LinkStackNode));

s=l->top;

\*element=s->data;

return 0;

}

### 入栈

方式一：

void Push(struct LinkStack \*\*top, int data){

struct LinkStack \*temp;

temp = malloc(sizeof(struct Stack));

if(!temp)

return;

temp->data = data;

temp->next = \*top; //临时节点数据域和指针域初始化

\*top = temp; //临时节点与原来栈的top建立联系

}

方式二：

int LinkStackPush(LinkStack \*l, int \*element){

LinkStackPtr s = (LinkStackPtr)malloc(sizeof(LinkStackNode));

s->data = \*element;

s->next=l->top;//调整指针

//为新节点分配内存并更新成员变量

l->top=s;//设置新节点为top

l->count++;

return 0;

}

### 出栈

方式一：

int Top(struct LinkStack\*top){

if(IsEmptyStack(top))

return INT\_MIN;

return top->next->data;

}

int Pop(struct LinkStack\*\*top)

{

int data;

struct LinkStack \*temp;

if(IsEmptyStack(\*top))

return INT\_MIN;

temp = \*top;

\*top = (\*top)temp->next;

data = temp->data;

free(temp);

return data;

}

方式二：

思路：把top指针并更为原来top节点的next，这个借助于中间变量指针s完成

代码：

int LinkStackPop(LinkStack \*l, int \*element){

LinkStackPtr s;//只是声明指针，但是不需要分配内存

if(StackEmpty(l)) //需要判空

return -1;

\*element = l->top->data;

s=l->top;//待删除节点初始化（暂存节点，放置断链）

l->top=s->top->next;

free(s);

l->count--;

return 0;

}

### 删除栈

void DeleteLinkStack (struct LinkStack \*\*top){

struct LinkStack \*temp,\*p;

p = \*top;

while(p->next){

temp = p->next; //必须使用临时变量存储，否则删除了就找不到下一个next节点了

p->next = temp->next;

free(temp);

}

free(p);//删除完next之后，删除栈顶结点

}

### 分析

性能：

如果把栈中的元素数量设为n，那么用这种方式实现的栈，其各项操作的复杂度如下：

|  |  |
| --- | --- |
| 空间复杂度（执行n次push操作） | O(n) |
| CreateStack()操作的时间复杂度 | O(1) |
| Push()操作的时间复杂度 | O(1)（平均） |
| Pop()操作的时间复杂度 | O(1) |
| Top()操作的时间复杂度 | O(1) |
| IsEmptyStack()操作的时间复杂度 | O(1) |
| DeleteStack()操作的时间复杂度 | O(1) |

## 队列

设计一个栈，支持基本的栈操作，这个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。



注：Leetcode 225

**分析：**





**代码：**



注：用队列实现栈，其实就是实现栈的top，push等操作。

class MyQueue {

private:

    std::queue<int> \_data;

public:

    /\*\* Initialize your data structure here. \*/

    MyQueue() {

    }

    /\*\* Push element x to the back of queue. \*/

    void push(int x) {

        std::queue<int> temp\_queue;

        temp\_queue.push(x);

*while(!\_data.empty()){ //note:采用empty()循环,不要采用迭代器*

            temp\_queue.push(\_data.front());

             \_data.pop();

        }

*//note:不能采用在这里\_data.clear()清空队列，因为循环中会使用\_data*

        while(!temp\_queue.empty()){

            \_data.push(temp\_queue.front());

            temp\_queue.pop();

        }

    }

    /\*\* Removes the element from in front of queue and returns that element. \*/

    int pop() {

        return \_data.front(); //note:pop()需要修改栈的元素，不能采用这样

*//note:正确的pop()操作*

*int top = \_data.front();*

*\_data.pop();*

*return top;*

    }

    /\*\* Get the front element. \*/

    int peek() {

        return \_data.front();

    }

    /\*\* Returns whether the queue is empty. \*/

    bool empty() {

        return \_data.empty();

    }

};

**测试：**



# 无锁栈

无锁栈（Lock-Free Stack）是一种不依赖互斥锁等阻塞同步机制，仅通过原子操作（如CAS，Compare-And-Swap）实现多线程安全的栈数据结构。它利用CPU提供的原子指令保证并发操作的正确性，避免了锁竞争带来的性能开销和阻塞延迟。

## 基本原理

无锁栈的核心是通过原子操作实现栈顶指针的修改，确保push（入栈）和pop（出栈）操作在并发场景下的安全性。以单向链表实现的栈为例，其关键设计和操作逻辑如下：

### 栈结构

template <typename T>

struct Node {

T data;

std::atomic<Node\*> next; // 原子指针，指向栈中的下一个节点

Node(const T& val) : data(val), next(nullptr) {}

};

template <typename T>

class LockFreeStack {

private:

std::atomic<Node<T>\*> top; // 原子指针，指向栈顶节点

};

- 栈顶指针（top）和节点的next指针均为原子类型，确保对指针的修改具有原子性和内存可见性。

### 核心操作实现

#### 入栈操作（push）

void push(const T& val) {

Node<T>\* new\_node = new Node<T>(val); // 创建新节点

do {

// 读取当前栈顶指针，作为新节点的next

new\_node->next = top.load(std::memory\_order\_relaxed);

// CAS尝试将新节点设为新栈顶：

// 若当前栈顶仍为new\_node->next（未被其他线程修改），则更新为新节点

} while (!top.compare\_exchange\_weak(

new\_node->next, // 预期值（当前栈顶）

new\_node, // 目标值（新节点）

std::memory\_order\_release, // 成功时的内存序（确保新节点状态可见）

std::memory\_order\_relaxed // 失败时的内存序（无需额外约束）

));

}

- 逻辑：新节点的`next`先指向当前栈顶，再通过CAS将栈顶更新为新节点。若期间栈顶被其他线程修改（如另一个线程同时入栈），则重试直到成功。

#### 出栈操作（pop）

bool pop(T& out) {

Node<T>\* old\_top;

do {

old\_top = top.load(std::memory\_order\_acquire); // 读取当前栈顶

if (!old\_top) return false; // 栈为空，返回失败

// 读取栈顶节点的next（即新栈顶）

} while (!top.compare\_exchange\_weak(

old\_top, // 预期值（当前栈顶）

old\_top->next.load(std::memory\_order\_relaxed), // 目标值（新栈顶）

std::memory\_order\_release, // 成功时的内存序

std::memory\_order\_acquire // 失败时的内存序

));

out = old\_top->data; // 获取出栈数据

delete old\_top; // 释放节点内存

return true;

}

- 逻辑：通过CAS将栈顶从`old\_top`更新为`old\_top->next`（下一个节点）。若期间栈顶被修改（如其他线程入栈/出栈），则重试直到成功。

#### 解决关键问题

- ABA问题：当栈顶节点A被弹出后又重新入栈，CAS可能误判栈顶未变。解决方案包括：

- 版本号机制：将指针与版本号绑定（如`std::atomic<std::pair<Node\*, int>>`），每次修改递增版本号。

- Hazard Pointers：跟踪被访问的节点，确保删除前节点未被其他线程引用。

- 内存管理：出栈时直接删除节点可能导致其他线程访问悬垂指针，需通过安全的内存回收机制（如Epoch-Based Reclamation）延迟释放。

## 优缺点

### 优点

1、高并发性能

避免锁竞争导致的线程阻塞和上下文切换，在多线程高频访问场景下（如高并发任务队列），吞吐量显著高于有锁栈。

2、低延迟

无锁操作不会导致线程无限期等待，适合对响应时间敏感的场景（如实时数据处理、高频交易）。

3、良好的多核扩展性

栈操作仅涉及栈顶指针的原子修改，多个线程可在不同CPU核心上并行操作，性能随核心数增加而提升（理想情况下）。

### 缺点

1、实现复杂度高

- 需处理CAS循环、ABA问题、内存序（如`acquire`/`release`）等细节，逻辑错误可能导致数据不一致或崩溃。

- 内存管理复杂，需确保节点删除不会影响其他线程的并发访问，通常需要额外的安全回收机制。

2、高冲突下的性能退化

当多个线程频繁竞争栈顶时，CAS操作会反复失败并重试，导致CPU资源浪费（活锁风险），性能可能反不如有锁实现。

3、功能局限性

仅支持栈的基本操作（push/pop），无法高效实现复杂功能（如查找、中间节点插入），这些操作的无锁实现逻辑过于复杂。

4、调试困难

并发问题（如偶发的数据丢失、重复出栈）难以复现和定位，依赖专门的并发调试工具（如ThreadSanitizer）。

## 适用场景

无锁栈适合高并发、操作简单（仅需push/pop）且冲突率较低的场景，典型包括：

1、多线程任务调度队列

如生产者-消费者模型中，多个线程生产任务（push）和消费任务（pop），无锁栈可减少锁竞争带来的延迟（如Disruptor框架的无锁栈变体）。

2、中断处理与内核态编程

操作系统内核中的栈（如中断栈、进程切换栈），需避免锁导致的内核阻塞，无锁实现是更安全的选择。

3、实时数据采集

如传感器数据实时入栈缓存，多个线程并发读取和处理，无锁栈可保证数据处理的低延迟。

4、高频交易系统

用于存储订单或交易事件，无锁栈的高吞吐和低延迟可避免交易机会丢失。

## 总结

无锁栈通过原子操作实现多线程安全，在高并发低冲突场景下性能优势明显，但实现复杂度高、调试困难，且高冲突时可能出现性能退化。实际开发中，应优先使用经过验证的无锁库（如C++的folly::LockFreeStack），而非手动实现。对于冲突率高或操作复杂的场景，带锁的栈（如`std::mutex`保护的栈）仍是更务实的选择。

# 工业级实现

## folly::LockFreeStack（Facebook Folly 库）

特点：无锁实现，基于 CAS 操作和 Hazard Pointers 解决内存安全问题，支持泛型类型，适合高并发场景。

优势：

优化了 ABA 问题和内存回收，避免悬垂指针。

提供批量操作接口（如pushN/popN）提升吞吐量。

适配多核 CPU，减少缓存竞争。

适用场景：高频交易、高性能服务器等对延迟敏感的场景。

仓库地址：https://github.com/facebook/folly

## abseil::flat\_hash\_map 中的栈结构（Google Abseil 库）

特点：虽然 Abseil 以哈希表闻名，但其absl::InlinedVector可作为高效栈使用，结合absl::Mutex实现线程安全。

优势：

内存紧凑（小容量时栈上存储），缓存友好。

配合 Abseil 的同步原语（如absl::Mutex），可轻松实现线程安全栈。

适用场景：中等并发的业务逻辑、内存受限的场景。

仓库地址：https://github.com/abseil/abseil-cpp

## tbb::concurrent\_stack（Intel TBB 库）

特点：Intel 线程构建块库中的并发栈，采用细粒度锁和分段技术，平衡安全性和性能。

优势：

自动适应线程数，减少锁竞争。

支持迭代器和批量操作，易用性高。

经过工业级验证，稳定性强。

适用场景：多线程任务调度、并行计算中的数据交换。

仓库地址：https://github.com/oneapi-src/oneTBB

# 应用

**直接运用：**

确保算式中的符号能够匹配（配对问题）

把中缀（infix）表示法转化成后缀（postfix）表示法

求后缀表达式（postfix expression）的值

实现（包括递归调用在内的）函数调用

寻找数值持续上升的区间（经常用于寻找股价的峰值）

保存浏览器访问过的网页（用来实现浏览器上后退按钮）

在文本编辑器中实现撤销功能，使得用户可以撤销早前所做的一系列修改

检查HTML与XML文件中的标签（tag）是否匹配

**说明：**对于查找下一个最值这种问题，栈比较实用，可以通过入栈出栈实现动态的算法。

**间接运用：**

给其他算法（例如图的遍历）充当辅助数据结构

充当其他数据结构（例如模拟队列）中的组件

## 实现栈

### 用队列实现栈

队列模拟栈，其实一个队列就够了，那么我们先说一说两个队列来实现栈的思路。

队列是先进先出的规则，把一个队列中的数据导入另一个队列中，数据的顺序并没有变，并没有变成先进后出的顺序。

所以用栈实现队列，和用队列实现栈的思路还是不一样的，这取决于这两个数据结构的性质。

但是依然还是要用两个队列来模拟栈，只不过没有输入和输出的关系，而是另一个队列完全用来备份的！

代码：

class MyStack {

public:

queue<int> que1;

queue<int> que2; // 辅助队列，用来备份

/\*\* Initialize your data structure here. \*/

MyStack() {

}

/\*\* Push element x onto stack. \*/

void push(int x) {

que1.push(x);

}

/\*\* Removes the element on top of the stack and returns that element. \*/

int pop() {

int size = que1.size();

size--;

while (size--) { // 将que1 导入que2，但要留下最后一个元素

que2.push(que1.front());

que1.pop();

}

int result = que1.front(); // 留下的最后一个元素就是要返回的值

que1.pop();

que1 = que2; // 再将que2赋值给que1

while (!que2.empty()) { // 清空que2

que2.pop();

}

return result;

}

/\*\* Get the top element.

\*\* Can not use back() direactly.

\*/

int top(){

int size = que1.size();

size--;

while (size--){

// 将que1 导入que2，但要留下最后一个元素

que2.push(que1.front());

que1.pop();

}

int result = que1.front(); // 留下的最后一个元素就是要回返的值

que2.push(que1.front()); // 获取值后将最后一个元素也加入que2中，保持原本的结构不变

que1.pop();

que1 = que2; // 再将que2赋值给que1

while (!que2.empty()){

// 清空que2

que2.pop();

}

return result;

}

/\*\* Returns whether the stack is empty. \*/

bool empty() {

return que1.empty();

}

};

### 用栈实现队列

思路：

在push数据的时候，只要数据放进输入栈就好，但在pop的时候，操作就复杂一些，输出栈如果为空，就把进栈数据全部导入进来（注意是全部导入），再从出栈弹出数据，如果输出栈不为空，则直接从出栈弹出数据就可以了。

最后如何判断队列为空呢？如果进栈和出栈都为空的话，说明模拟的队列为空了。

使用两个栈模拟队列，主要是因为栈的顺序限制。

代码：

class MyQueue {

public:

stack<int> stIn;

stack<int> stOut;

/\*\* Initialize your data structure here. \*/

MyQueue() {

}

/\*\* Push element x to the back of queue. \*/

void push(int x) {

stIn.push(x);

}

/\*\* Removes the element from in front of queue and returns that element. \*/

int pop() {

// 只有当stOut为空的时候，再从stIn里导入数据（导入stIn全部数据），否则顺序会错乱

if (stOut.empty()) {

// 从stIn导入数据直到stIn为空

while(!stIn.empty()) {

stOut.push(stIn.top());

stIn.pop();

}

}

int result = stOut.top();

stOut.pop();

return result;

}

/\*\* Get the front element. \*/

int peek() {

int res = this->pop(); // 直接使用已有的pop函数

stOut.push(res); // 因为pop函数弹出了元素res，所以再添加回去（或者直接实现pop里面的逻辑，但是不执行pop）

return res;

}

/\*\* Returns whether the queue is empty. \*/

bool empty() {

return stIn.empty() && stOut.empty();

}

};

### 最小栈/包含min函数的栈

**题目：**定义栈的数据结构，请在该类型中实现一个能够得到栈的最小元素的min函数GetMin()。在该栈中，调用min、push及pop的时间复杂度都是O(1)。

push(x)：将元素x推入栈中；

pop()：删除栈顶的元素

top()：获取栈顶元素

getMin()：检索栈中的最小元素

**分析：**

可以定义一个变量minEle存储最小值，在入栈的时候依次将入栈元素与之比较，小于该值则更新该值为入栈元素，否则继续。在获取最小元素的时候，直接返回该变量即可。但是，这种方案在pop操作的时候存在问题：比如push元素为3,2,1，minEle=1，pop出1的时候，该值应该更新为2，而不应该为1。也就是说，我们在push操作更新minEle的时候不应该将之前的旧的最小值覆盖掉，应该保存起来pop的时候使用。

采用1个变量记录最小值：



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（pop操作被覆盖）；

2、栈的每个状态，都需要有一个变量记录最小值

应该使用数组依次记录入栈时最小元素，执行push操作时（入栈顺序3->2->1->2->3）：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 3 | 2 | 1 | 2 | 3 |

入栈操作对应的最小元素数组为：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 3 | 2 | 1 | 1 | 1 |

执行pop操作的时候，两者的变化为：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 3 | 2 | 1 | 2 |  |
| 3 | 2 | 1 | 1 |  |

继续：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 3 | 2 | 1 |  |  |
| 3 | 2 | 1 |  |  |

继续：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 3 | 2 |  |  |  |
| 3 | 2 |  |  |  |

继续：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 3 |  |  |  |  |
| 3 |  |  |  |  |

图示：



注：剑指offer P132（非常重要！） Leetcode155

**代码：**



class MinStack {

private:

    stack<int> \_data;

    stack<int> \_min;

public:

    /\*\* initialize your data structure here. \*/

    MinStack() {

    }

    void push(int x) {

        \_data.push(x);

        if (\_min.empty()){

            \_min.push(x);

        }else{

            if(x > \_min.top()){

                x = \_min.top();

            }

            \_min.push(x);

        }

    }

    void pop() {

        \_data.pop();

        \_min.pop();

    }

    int top() {

        return \_data.top();

    }

    int min() {

        return \_min.top();

    }

};

**测试：**



## 入栈/出栈

### 合法的出栈序列

**题目：**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？



注：poj 1363 Rail

**分析：**











不合法：



**代码：**



**测试：**



### 栈的压入/弹出序列

**题目：**输入两个整数序列，第一个序列表示栈的压入顺序，请判断第二个序列是否为该栈的弹出顺序。假如压入栈的所有数字均不相等。例如序列1、2、3、4、5是某栈的压栈序列，序列4、5、3、2、1是该压栈序列对应的一个弹出序列，但4、3、5、1、2就不可能是该压栈序列的弹出序列。

注：剑指offer P134

## 栈翻转

/\*

使用递归的方法将栈中的数据进行翻转

思路：如果栈为空，那么直接将元素放到栈的底部即可，如果栈中有元素，那么取出栈中的元素，将原来的元素再次调用函数存放到栈底，然后将取出的元素压入栈即可。

\*/

//当某一个元素放入栈底部

void Push\_Bottom(stack<int>& st,int value)

{

int tmp;

//如果栈为空，那么直接将当前元素压入栈

if(st.size()==0)

st.push(value);

else //如果栈中有元素，那么取出其中的元素，然后再将元素压入

{

tmp = st.top();

st.pop();

Push\_Bottom(st,value);

st.push(tmp);

}

}

/\*

翻转栈

\*/

void Reverse\_st(stack<int>& st)

{

int tmp;

if(st.size()<=1)

return;

else

{

//取出栈顶元素

tmp = st.top();

st.pop();

//递归调用，翻转剩余的元素

Reverse\_st(st);

//将取出的元素放入栈底部

Push\_Bottom(st,tmp);

}

}

int main()

{

string str("{()[]{}}");

cout<<isValidParenthess(str);

stack<int> st;

int i;

int tmp;

for(i=0;i<5;i++)

st.push(i);

Reverse\_st(st);

// Push\_Bottom(st,5);

while(!st.empty())

{

tmp = st.top();

cout<<tmp<<endl;

st.pop();

}

return 0;

}

## 匹配问题

### 有效的括号/括号匹配检测

**题目：**一个只包含括号的字符串，判断该字符串所包含的括号匹配是否合法。

注：LeetCode 20

**分析：**

1、首先对空字符串和仅含有一个字符的字符串做判断

2、如果字符串中出现左侧括号"{"，"("，"["入栈

3、如果字符串中出现右侧括号"}"，")"，"]"则与该字符前面的字符判断(即栈顶元素)，看是否能够配对，如果可以配对则将栈顶元素弹出，否则返回false

4、整个字符串判断完成后，如果最后栈为空，则表示全部配对，否则返回false

**代码：**

#include <iostream>

#include <vector>

#include <string>

#include <stack>

using namespace std;

/\*

判断一个括号字符串组成的序列是否合法

使用栈的思想 ,最后判断栈是否为空

\*/

bool isValidParenthess(string& str)

{

//首先对空等特殊情况判断

if(str.empty()) //note:需要先对输入判空，如果为空根据需求判断为true

return true;

if(1 == str.size()) //note：输入大小为1的情况需要特殊判断一下，这样平均执行时间大大提升

return false;

stack<char> st; //note:定义的是栈stack而不是vector

int i;

char tmp;

//开始借助栈这个数据结构的特性来遍历整个字符串

for(i=0;i<str.length();i++) //note:使用str.size()和str.length()都可以

{

if(str[i]=='(' || str[i]=='[' || str[i]=='{')

st.push(str[i]); //note:不是push\_back，是push

else if(str[i]==']' && !st.empty()) //note：判空不可少，否则进入函数core

{

tmp = st.top(); //现阶段判断的就是跟前一个元素比较，即栈顶元素

if(tmp != '[')

return false;

else

st.pop();

}

else if(str[i] ==')' && !st.empty())

{

tmp = st.top();

if(tmp != '(')

return false;

else

st.pop();

}

else if(str[i]=='}' && !st.empty())

{

tmp = st.top();

if(tmp != '{')

return false;

else

st.pop();

}

else

return false; //note:这一步不能少，对于”]”这样的输入判断一下false

}

if(st.empty())

return true;

else

return false;

}

或：

class Solution {

public:

bool isValid(string s) {

if (s.size() % 2 != 0) return false; // 如果s的长度为奇数，一定不符合要求

stack<char> st;

for (int i = 0; i < s.size(); i++) {

if (s[i] == '(') st.push(')');

else if (s[i] == '{') st.push('}');

else if (s[i] == '[') st.push(']');

// 第三种情况：遍历字符串匹配的过程中，栈已经为空了，没有匹配的字符了，说明右括号没有找到对应的左括号 return false

// 第二种情况：遍历字符串匹配的过程中，发现栈里没有我们要匹配的字符。所以return false

else if (st.empty() || st.top() != s[i]) return false;

else st.pop(); // st.top() 与 s[i]相等，栈弹出元素

}

// 第一种情况：此时我们已经遍历完了字符串，但是栈不为空，说明有相应的左括号没有右括号来匹配，所以return false，否则就return true

return st.empty();

}

};

注：Leetcode20、921类似

P21：给定一个只包括“（”，“）”，“{”，“}”，“[”，“]”的字符串，判断字符串是否有效。

有效字符串需满足：

1. 左括号必须用相同类型的右括号闭合。
2. 左括号必须以正确的顺序闭合。

注意：空字符串可以被认为是有效字符串。

分析：如果字符串中的字符是左括号（(/[/{），则入栈，如果是右括号，则判断是否与栈顶元素相等，如果相等则栈顶元素出栈，表示存在一对匹配的括号，如果不相等则不合法。

这里是匹配多种括号，如果只匹配一种，则不需要使用栈（空间复杂度为O(n)），而使用一个变量标识是否匹配（这样空间复杂度为O(1)）。

Java代码实现：

public class Stack{

public boolean isValid(String s) {

char[] stack = new char[s.;ength()];

int size = 0;

for(int i=0;i<s.length();i++} {

char c = s.charAt(i);

if(c=='(' || c=='{' || c=='[') { //如果是左括号则入栈

stack[size++] = c;

}else if(size>0 && c-stack[size-1]<2 && c-stack[size-1]>0) {

//如果是右括号则判断是否与栈顶元素匹配(注意空栈)

//查看ASCII码表可知码值相差1、2为左右括号

size--;

}

}

}

}

L921：使括号有效的最少添加

给定一个由“(”和“)”括号组成的字符串S，我们需要添加最少的括号(“(”或是“)”，可以在任何位置，以使得到的括号字符串有效。

从形式上讲，只有满足下面几点之一，括号字符串才是有效的：

1. 它是一个空字符串或者
2. 它可以被写成AB（A与B连接），其中A和B都是有效字符串，或者
3. 它可以被写成（A），其中A是有效字符串

给定一个括号字符串，返回为使结果字符串有效而必须添加的最少括号数。

分析：遇到左括号，则放入栈，遇到右括号，则检查是否与栈顶元素匹配，如果匹配则出栈，不匹配继续。最后，栈的长度就是最后额外需要匹配的括号数。

### 有效的括号字符串

给你一个只包含三种字符的字符串，支持的字符类型分别是 '('、')' 和 '\*'。请你检验这个字符串是否为有效字符串，如果是有效字符串返回 true 。

有效字符串符合如下规则：

任何左括号 '(' 必须有相应的右括号 ')'。

任何右括号 ')' 必须有相应的左括号 '(' 。

左括号 '(' 必须在对应的右括号之前 ')'。

'\*' 可以被视为单个右括号 ')' ，或单个左括号 '(' ，或一个空字符串。

一个空字符串也被视为有效字符串。

注：LeetCode 678

### 判断一个括号字符串是否有效

一个括号字符串是只由'(' 和 ')'组成的非空字符串。如果一个字符串满足下面任意一个条件，那么它就是有效的：

* 字符串为().
* 它可以表示为AB（A与B连接），其中A和B都是有效括号字符串。
* 它可以表示为(A)，其中 A 是一个有效括号字符串。

给你一个括号字符串s和一个字符串locked，两者长度都为n。locked是一个二进制字符串，只包含'0'和'1'。对于locked中每一个下标i：

* 如果locked[i]是'1' ，你不能改变s[i]。
* 如果locked[i]是'0'，你可以将s[i]变为'('或者')'。

如果你可以将s变为有效括号字符串，请你返回true，否则返回false。

注：LeetCode 2116

分析：

### 有效括号的嵌套深度

注：LeetCode 1111

### 使括号有效的最少添加

注：LeetCode 921

### 平衡括号字符串的最少插入次数

注：LeetCode 1541

### 回文链表

给你一个单链表的头节点 head ，请你判断该链表是否为

回文链表。如果是，返回 true ；否则，返回 false 。

注：LeetCode 234

分析：

采用双指针模拟实现或者采用栈实现。这类回文问题，采用快慢指针会更合适。

### 最长合法括号匹配长度

题目要求：有一个由括号组成的字符串，在该字符串中找到最长的符合括号匹配规则的子串的长度。

注：LeetCode 32

分析：

代码：

#include <iostream>

#include <string>

#include <stack>

using namespace std;

/\*

最长合法括号匹配长度

\*/

/\*

使用一个bool数组来标记已经匹配过的字符扎到最长的连续标记的长度就是所求的结果，只需要两次遍历数组，时间复杂度为O(n)

\*/

int LongestValidParentheses(string s) {

bool \*a = new bool[s.length()];

memset(a, false, s.length());

stack<int> st;

for (int i = 0; i < s.length(); ++i) {

if (s[i] == '(') {

st.push(i);

} else if (s[i] == ')' && !st.empty()) {

a[i] = true;

a[st.top()] = true;

st.pop();

}

}

int max\_len = 0, cur\_len = 0;

for (int i = 0; i < s.length(); ++i) {

if (a[i])

cur\_len++;

else

cur\_len = 0;

max\_len = max(max\_len, cur\_len);

}

return max\_len;

}

/\*

如果我们使用栈记录某一个符号在字符串中的位置，假如对于当前字符和栈顶位置的

字符可以匹配，那么可以根据栈是否为空来判断当前最长匹配子序列然后和目标进行匹配

\*/

int LongestValidParentheses2(string s) {

stack<int> st;

int pos;

int maxlen = 0;

for(int i = 0 ; i < s.size() ; i++)

{

if(s[i] == '(') st.push(i);

else // ')'

{

if(!st.empty() && s[st.top()]=='(')

{

st.pop();

if(st.empty())

pos = i+1;

else

pos = i-st.top();

if(maxlen < pos)

maxlen = pos;

}

else

st.push(i);

}

}

return maxlen;

}

int main()

{

string str=")()())";

cout<<LongestValidParentheses(str);

cout<<endl<<Longest\_dp(str);

// cout<<str<<endl;

return 0;

}

### 比较含退格的字符串

给定 s 和 t 两个字符串，当它们分别被输入到空白的文本编辑器后，如果两者相等，返回 true 。# 代表退格字符。

注意：如果对空文本输入退格字符，文本继续为空。

注：LeetCode 844

分析：

用两个栈，不断添加字符，遇上#就删除尾部字符，最后对比两个栈是否相同。

### 检查替换后的词是否有效

注：LeetCode 1003

## 删除问题

### 整理字符串

给你一个由大小写英文字母组成的字符串 s 。

一个整理好的字符串中，两个相邻字符 s[i] 和 s[i+1]，其中 0<= i <= s.length-2 ，要满足如下条件:

若 s[i] 是小写字符，则 s[i+1] 不可以是相同的大写字符。

若 s[i] 是大写字符，则 s[i+1] 不可以是相同的小写字符。

请你将字符串整理好，每次你都可以从字符串中选出满足上述条件的 两个相邻 字符并删除，直到字符串整理好为止。

请返回整理好的 字符串 。题目保证在给出的约束条件下，测试样例对应的答案是唯一的。

注意：空字符串也属于整理好的字符串，尽管其中没有任何字符。

注：LeetCode 1544

类似题目：LeetCode 1047

分析：使用栈模拟。

### 删除字符串中的所有相邻重复项

给出由小写字母组成的字符串 S，重复项删除操作会选择两个相邻且相同的字母，并删除它们。

在 S 上反复执行重复项删除操作，直到无法继续删除。

在完成所有重复项删除操作后返回最终的字符串。答案保证唯一。

示例：

输入："abbaca"

输出："ca"

解释：

例如，在 "abbaca" 中，我们可以删除 "bb" 由于两字母相邻且相同，这是此时唯一可以执行删除操作的重复项。之后我们得到字符串 "aaca"，其中又只有 "aa" 可以执行重复项删除操作，所以最后的字符串为 "ca"。

提示：

1 <= S.length <= 20000

S 仅由小写英文字母组成。

注：LeetCode 1047

类似题目：LeetCode 1544

分析：

使用栈模拟。

本题要删除相邻相同元素，相对于20. 有效的括号 (opens new window)来说其实也是匹配问题，20. 有效的括号 是匹配左右括号，本题是匹配相邻元素，最后都是做消除的操作。

本题也是用栈来解决的经典题目。

那么栈里应该放的是什么元素呢？

我们在删除相邻重复项的时候，其实就是要知道当前遍历的这个元素，我们在前一位是不是遍历过一样数值的元素，那么如何记录前面遍历过的元素呢？

所以就是用栈来存放，那么栈的目的，就是存放遍历过的元素，当遍历当前的这个元素的时候，去栈里看一下我们是不是遍历过相同数值的相邻元素。

然后再去做对应的消除操作。

代码：

class Solution {

public:

string removeDuplicates(string S) {

stack<char> st;

for (char s : S) {

if (st.empty() || s != st.top()) {

st.push(s);

} else {

st.pop(); // s 与 st.top()相等的情况

}

}

string result = "";

while (!st.empty()) { // 将栈中元素放到result字符串汇总

result += st.top();

st.pop();

}

reverse (result.begin(), result.end()); // 此时字符串需要反转一下

return result;

}

};

当然可以拿字符串直接作为栈，这样省去了栈还要转为字符串的操作。

代码如下：

class Solution {

public:

string removeDuplicates(string S) {

string result;

for(char s : S) {

if(result.empty() || result.back() != s) {

result.push\_back(s);

}

else {

result.pop\_back();

}

}

return result;

}

};

### 去除重复字母

给你一个字符串 s ，请你去除字符串中重复的字母，使得每个字母只出现一次。需保证 返回结果的字典序最小（要求不能打乱其他字符的相对位置）。

注：LeetCode 316

类似题目：LeetCode 1081

### 不同字符的最小子序列

返回 s 字典序最小的子序列，该子序列包含 s 的所有不同字符，且只包含一次。

注：LeetCode 1081

类似题目：LeetCode 316

### 移除无效的括号

注：LeetCode 1249

### 从字符串中移除星号

注：LeetCode 2390

分析：

直接使用string模拟或者使用栈stack<char>。

### 删除最外层的括号

有效括号字符串为空("")、"(" + A + ")"或A + B，其中A和B都是有效的括号字符串，+代表字符串的连接。例如，""，"()"，"(())()"和"(()(()))"都是有效的括号字符串。

如果有效字符串S非空，且不存在将其拆分为S = A+B的方法，我们称其为原语（primitive），其中A 和B都是非空有效括号字符串。

给出一个非空有效字符串S，考虑将其进行原语化分解，使得：S = P\_1 + P\_2 + ... + P\_k，其中P\_i是有效括号字符串原语。

对S进行原语化分解，删除分解中每个原语字符串的最外层括号，返回S。

**示例 1：**

输入："(()())(())"

输出："()()()"

解释：

输入字符串为 "(()())(())"，原语化分解得到 "(()())" + "(())"，删除每个部分中的最外层括号后得到 "()()" + "()" = "()()()"。

**示例 2：**

输入："(()())(())(()(()))"

输出："()()()()(())"

解释：

输入字符串为 "(()())(())(()(()))"，原语化分解得到 "(()())" + "(())" + "(()(()))"，

删除每个部分中的最外层括号后得到 "()()" + "()" + "()(())" = "()()()()(())"。

**示例 3：**

输入："()()"

输出：""

解释：

输入字符串为 "()()"，原语化分解得到 "()" + "()"，

删除每个部分中的最外层括号后得到 "" + "" = ""。

**提示：**

S.length <= 10000

S[i]为"(" 或")"

S是一个有效括号字符串

注：LeetCode 1021

分析：使用栈模拟。

### 删除子串后的字符串最小长度

给你一个仅由 大写 英文字符组成的字符串 s 。

你可以对此字符串执行一些操作，在每一步操作中，你可以从 s 中删除 任一个 "AB" 或 "CD" 子字符串。

通过执行操作，删除所有 "AB" 和 "CD" 子串，返回可获得的最终字符串的最小可能长度。

注意，删除子串后，重新连接出的字符串可能会产生新的 "AB" 或 "CD" 子串。

注：LeetCode 2696

分析：

使用栈模拟。

### 删除子字符串的最大得分

注：LeetCode 1717

### 删除最短的子数组使剩余数组有序

注：LeetCode 1574

### 使字符串平衡的最少删除次数

注：LeetCode 1653

## 进制转换

### 二进制->十进制

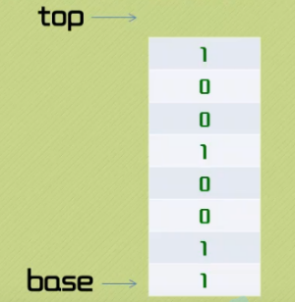
题目：利用栈的数据结构特点，将二进制转换为十进制数。

分析：二进制数是计算机数据的存储形式，它是由一串0和1组成的，每个二进制数转换成相应的十进制数方法如下：

(XnXn-1……X3X2X1)2 = X1\*2^0 + X2\*2^1 + …+ Xn\*2^\*(n-1)

一个二进制数要转换为相应的十进制数，就是从最低位起用每一位去乘以对应位的积，也就是说用第n位乘以2^(n-1)，然后全部加起来。

由于栈具有后进先出的特性，例如我们输入11001001这样的二进制数，如图：



由此可以看出，输出的二进制顺序，在计算十进制数的时候是先弹出栈顶元素。

代码：

printf("输入二进制数(#表示结束)：\n");

scanf("%c",&c);

while(c!='#')

{

Push(&s,c); //入栈

scanf("%c",&c);

}

getchar(); //把'\n'从缓冲区去掉

len = StackLen(s);

for(int i=0;i<len;i++)

{

Pop(&s,&c);

sum = sum + (c-48) \* pow(2,i);

}

printf("转化后十进制数：%d\n",sum);

### 二进制->八进制

一个字节（8bit）刚好可以用两个十六进制数表示完整，也大大的节省了显示空间。

因为早期的计算机系统都是三的倍数，所以用八进制比较方便。

在进行二进制到八进制的转换时，要将二进制数的每三位转换成一个二进制数来表示，然后按照顺序输出即可。如下图所示：







## 数学应用

### 简单的计算器

**题目：**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。

图片包含 图形用户界面

AI 生成的内容可能不正确。

注：Leetcode224

**分析：**

栈内元素计算：



使用栈处理优先级：



计算思路：







字符串转换为数字：



计算函数：



字符串处理思路：



**代码：**



**测试：**



### 逆波兰表达式/求解后缀表达式

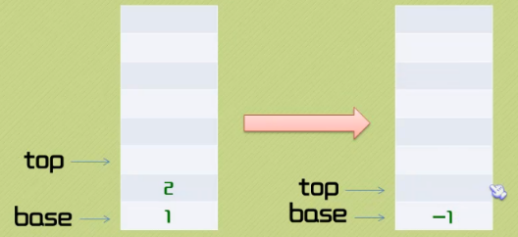
逆波兰表达式（RPN）即一种不需要括号的后缀表达式。

对于(1-2)\*(4+5)，如果用逆波兰表示法应该是：1 2 – 4 5 + \*。虽然这种方式在人看来难以理解，但是对于计算机而言却十分容易理解和处理。因为只需要利用栈的特点，就可以将这种后缀表达式的性能发挥到极致。

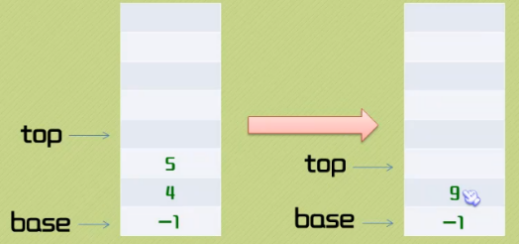
注：LeetCode 150

其基本原理如下：

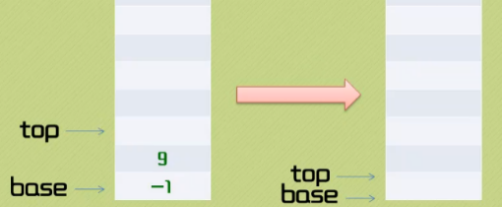
1. 数字1和2进栈，遇到减号运算符弹出两个元素进行运算并把结果入栈：



1. 4和5入栈，遇到加号运算符，4和5弹出栈，相加后结果9入栈。



1. 然后又遇到乘法运算符，将9和-1弹出栈进行乘法计算，此时栈空并无数据压栈，-9位最终运算结果，



题目要求：求一个由字符串构成的后缀表达式的值。

代码：

#include <iostream>

#include <vector>

#include <string>

#include <stack>

#include <algorithm>

using namespace std;

/\*

表达式求解，借助栈这种数据结构，需要注意字符串和整数之间的转化

\*/

//将字符串转化为整型

int strtoi(string& s)

{

if(s.length() == 0)

return 0;

int i=0,flag=1,result=0;

if(s[0]=='-')

{

flag =-1;

i++;

}

if(s[0]=='+')

{

i++;

}

for(;i<s.length();i++)

{

result \*= 10;

result += (s[i]-'0');

}

return result\*flag;

}

//将整型转化为字符串

string itostr(int num)

{

int flag =1,count = num;

if(num <0)

{

flag = 0;

count = -count;

}

string result;

while(count)

{

result += (count%10 +'0');

count = count/10;

}

if(flag == 0)

result += '-';

reverse(result.begin(),result.end());

return result;

}

/\*

表达式求解主函数

\*/

int evalRPN(vector<string> &tokens)

{

stack<string> sk;

int result =0,i,temp;

string first,second;

//开始处理字符串向量

for(i=0;i<tokens.size();i++)

{

//如果当前的字符串为符号，需要从栈中取出操作数

if(tokens[i]=="+"||tokens[i]=="-"||tokens[i]=="\*"||tokens[i]=="/")

{

first = sk.top();

sk.pop();

second =sk.top();

sk.pop();

//开始计算

if(tokens[i] == "+")

temp = strtoi(first)+strtoi(second);

else if(tokens[i]=="-")

temp = strtoi(first)-strtoi(second);

else if(tokens[i]=="\*")

temp = strtoi(first)\*strtoi(second);

else

temp = strtoi(second)/strtoi(first);

//将结果继续压入栈

first = itostr(temp);

sk.push(first);

}

else // 其他情况都只需要将当前字符串压入栈内即可

sk.push(tokens[i]);

}

first = sk.top();

result = strtoi(first);

return result;

}

int main()

{

string array[]={"4", "13", "5", "/", "+"};

vector<string> tokens(array,array+sizeof(array)/sizeof(array[0]));

cout<<evalRPN(tokens)<<endl;

return 0;

}

### 中缀表达式->后缀表达式

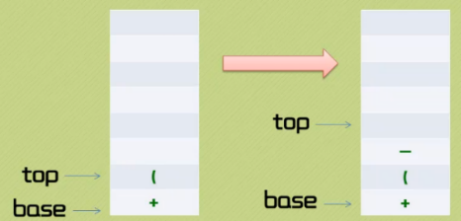
题目：将中缀表达式“(1-2)\*(4+5) ”转化为“1 2 – 4 5 + \*”

分析：利用栈的“记忆”，符号全都推入栈即可。

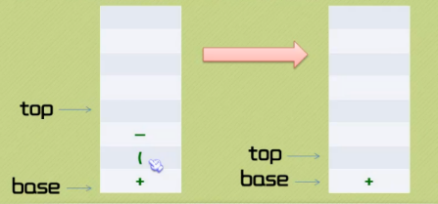
1. 首先遇到第一个输入是数字1，数字在后缀表达式中都是直接输出，接着是符号“+”，入栈：



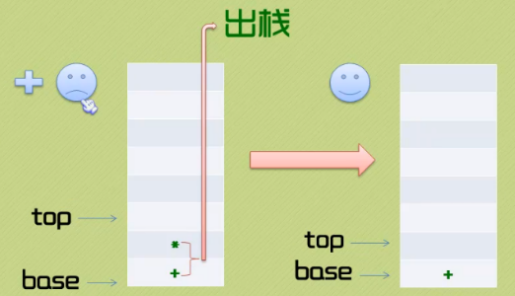
1. 第三个字符是“(”，依然是符号，入栈，接着是数字2，输出，然后是符号“-”，入栈：



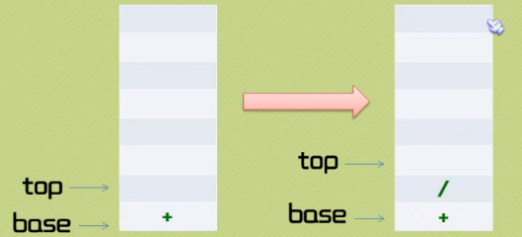
1. 接下来是数字3，输出，紧跟着是“)”，此时，我们需要去匹配栈里的“(”，然后再匹配前将栈顶数据依次出栈（这就好比括号里先执行的道理）：



1. 紧接着是符号“\*”，直接入栈
2. 遇到数字4，输出，之后是符号“+”，此时栈顶元素是符号“\*”，按照先乘除后加减原理，此时栈顶的乘号优先级比即将入栈的加号要高，所以出栈
3. 栈中第二个元素是加号，按照先进先出原则，加号入栈：



1. 紧接着数字10，输出，最后是符号“/”，进栈



1. 最后一个数字5，输出，所有的输入处理完毕，但是栈中仍然有数据，所以将栈中符号依次出栈。

总结：从左到右遍历中缀表达式的每个数组和符号，若是数字则直接输出，若是符号，则判断其栈顶符号的优先级，是右括号或者优先级低于栈顶符号，则栈顶元素依次出栈并输出，直到遇到左括号或栈空才将符号入栈。

## 操作系统

### 函数调用

### 中断

### 内存分配

### 缓冲处理

## 二叉树遍历

### 二叉树的中序遍历

### 二叉树的前序遍历

### 二叉树的后序遍历

### 验证二叉树的前序序列化

注：LeetCode 331

### N叉树的前序遍历

### N叉树的后续遍历

### 递增顺序搜索树

### 二叉树展开为链表

给定一个二叉树，原地将它展开为一个单链表。

例如，给定二叉树

1

/ \

2 5

/ \ \

3 4 6

将其展开为：

1

\

2

\

3

\

4

\

5

\

6

注：可以修改一下原题目，按照层次转换为链表，原理相同，只是采用层次遍历然后设置链表的next指针。

注：LeetCode 114

分析：

前序遍历（栈）模拟。

## 更大值

这个其实与配对问题类似，只是这里的配对改为了“下一个元素的某种特性”。

### 下一个更大元素

nums1中数字x的下一个更大元素是指x在nums2中对应位置右侧的第一个比x大的元素。

给你两个没有重复元素的数组nums1和nums2，下标从0开始计数，其中nums1是nums2的子集。

对于每个0 <= i < nums1.length，找出满足nums1[i] == nums2[j]的下标j，并且在nums2确定nums2[j]的下一个更大元素。如果不存在下一个更大元素，那么本次查询的答案是-1。

返回一个长度为nums1.length的数组ans作为答案，满足ans[i]是如上所述的 下一个更大元素。

注：LeetCode 496

分析：

采用栈模拟。

代码：

// 版本一

class Solution {

public:

vector<int> nextGreaterElement(vector<int>& nums1, vector<int>& nums2) {

stack<int> st;

vector<int> result(nums1.size(), -1);

if (nums1.size() == 0) return result;

unordered\_map<int, int> umap; // key:下标元素，value：下标（两个数组之间的关系）

for (int i = 0; i < nums1.size(); i++) {

umap[nums1[i]] = i;

}

st.push(0);

for (int i = 1; i < nums2.size(); i++) {

if (nums2[i] < nums2[st.top()]) { // 情况一

st.push(i);

} else if (nums2[i] == nums2[st.top()]) { // 情况二

st.push(i);

} else { // 情况三

while (!st.empty() && nums2[i] > nums2[st.top()]) {

if (umap.count(nums2[st.top()]) > 0) { // 看map里是否存在这个元素

int index = umap[nums2[st.top()]]; // 根据map找到nums2[st.top()] 在 nums1中的下标

result[index] = nums2[i];

}

st.pop();

}

st.push(i);

}

}

return result;

}

};

### 下一个更大元素II

给定一个循环数组（最后一个元素的下一个元素是数组的第一个元素），输出每个元素的下一个更大元素。数字 x 的下一个更大的元素是按数组遍历顺序，这个数字之后的第一个比它更大的数，这意味着你应该循环地搜索它的下一个更大的数。如果不存在，则输出 -1。

**示例 1:**

输入: [1,2,1]

输出: [2,-1,2]

**解释：**

第一个 1 的下一个更大的数是 2；

数字 2 找不到下一个更大的数；

第二个 1 的下一个最大的数需要循环搜索，结果也是 2。

**注意：**

输入数组的长度不会超过 10000。

注：LeetCode 503

类似题目：

496. 下一个更大元素 I

1856. 子数组最小乘积的最大值

分析：单调栈模拟

代码：

// 版本一

class Solution {

public:

vector<int> nextGreaterElements(vector<int>& nums) {

// 拼接一个新的nums

vector<int> nums1(nums.begin(), nums.end());

nums.insert(nums.end(), nums1.begin(), nums1.end());

// 用新的nums大小来初始化result

vector<int> result(nums.size(), -1);

if (nums.size() == 0) return result;

// 开始单调栈

stack<int> st;

st.push(0);

for (int i = 1; i < nums.size(); i++) {

if (nums[i] < nums[st.top()]) st.push(i);

else if (nums[i] == nums[st.top()]) st.push(i);

else {

while (!st.empty() && nums[i] > nums[st.top()]) {

result[st.top()] = nums[i];

st.pop();

}

st.push(i);

}

}

// 最后再把结果集即result数组resize到原数组大小

result.resize(nums.size() / 2);

return result;

}

};

### 链表中的下一个更大节点

给定一个长度为 n 的链表 head

对于列表中的每个节点，查找下一个 更大节点 的值。也就是说，对于每个节点，找到它旁边的第一个节点的值，这个节点的值 严格大于 它的值。

返回一个整数数组 answer ，其中 answer[i] 是第 i 个节点( 从1开始 )的下一个更大的节点的值。如果第 i 个节点没有下一个更大的节点，设置 answer[i] = 0 。

注：LeetCode 1019

### 子数组范围和

给你一个整数数组 nums 。nums 中，子数组的 范围 是子数组中最大元素和最小元素的差值。

返回 nums 中 所有 子数组范围的 和 。

子数组是数组中一个连续 非空 的元素序列。

注：LeetCode 2104

分析：

### 子数组最小乘积的最大值

一个数组的最小乘积定义为这个数组中最小值乘以数组的和。

比方说，数组[3,2,5]（最小值是2）的最小乘积为2 \* (3+2+5) = 2 \* 10 = 20。

给你一个正整数数组nums，请你返回nums任意非空子数组的最小乘积的 最大值。由于答案可能很大，请你返回答案对109 + 7取余的结果。

请注意，最小乘积的最大值考虑的是取余操作之前的结果。题目保证最小乘积的最大值在不取余的情况下可以用64位有符号整数保存。

子数组定义为一个数组的连续部分。

注：LeetCode 1856

分析：

## 每日温度

请根据每日气温列表，重新生成一个列表。对应位置的输出为：要想观测到更高的气温，至少需要等待的天数。如果气温在这之后都不会升高，请在该位置用0来代替。

例如，给定一个列表 temperatures = [73, 74, 75, 71, 69, 72, 76, 73]，你的输出应该是[1, 1, 4, 2, 1, 1, 0, 0]。

提示：气温列表长度的范围是[1, 30000]。每个气温的值的均为华氏度，都是在[30, 100]范围内的整数。

思路：

代码：

// 版本一

class Solution {

public:

vector<int> dailyTemperatures(vector<int>& T) {

// 递增栈

stack<int> st;

vector<int> result(T.size(), 0);

st.push(0);

for (int i = 1; i < T.size(); i++) {

if (T[i] < T[st.top()]) { // 情况一

st.push(i);

} else if (T[i] == T[st.top()]) { // 情况二

st.push(i);

} else {

while (!st.empty() && T[i] > T[st.top()]) { // 情况三

result[st.top()] = i - st.top();

st.pop();

}

st.push(i); // 栈清空后再push

}

}

return result;

}

};

## 接雨水

给定n 个非负整数表示每个宽度为 1 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。

注：LeetCode 42

分析：

可以使用双指针（推荐），单调栈和动态规划实现。

代码（单调栈）：

class Solution {

public:

int trap(vector<int>& height) {

if (height.size() <= 2) return 0; // 可以不加

stack<int> st; // 存着下标，计算的时候用下标对应的柱子高度

st.push(0);

int sum = 0;

for (int i = 1; i < height.size(); i++) {

if (height[i] < height[st.top()]) { // 情况一

st.push(i);

} if (height[i] == height[st.top()]) { // 情况二

st.pop(); // 其实这一句可以不加，效果是一样的，但处理相同的情况的思路却变了。

st.push(i);

} else { // 情况三

while (!st.empty() && height[i] > height[st.top()]) { // 注意这里是while

int mid = st.top();

st.pop();

if (!st.empty()) {

int h = min(height[st.top()], height[i]) - height[mid];

int w = i - st.top() - 1; // 注意减一，只求中间宽度

sum += h \* w;

}

}

st.push(i);

}

}

return sum;

}

};

## 柱状图中最大矩形

## 简化目录路径

**题目：**给定一个字符串，是Linux操作系统下的绝对目录路径，将该字符串所代表的目录路径最简化。

代码：

#include <iostream>

#include <string>

#include <vector>

#include <stack>

using namespace std;

/\*

一个表达式所表述的路径，进行最简化

\*/

string SimplifyPath(string& str)

{

string result;

stack<char> st;

if(str.length() == 0 || str[0] !='/')

return result;

int i;

char tmp;

st.push(str[0]);

for(i=1;i<str.length();i++)

{

tmp = st.top();

if(isalpha(str[i])) //当期字符是字母

st.push(str[i]);

if(str[i]=='/' && i!=str.length()-1)

{

if(tmp != '/' && tmp != '.') // 斜线

{

st.push(str[i]);

}

if(tmp == '.')

{

st.pop();

}

}

if(str[i]=='.') //逗点

{

if(tmp == '.')

{

st.pop();

st.pop();

if(!st.empty())

tmp = st.top();

while(!st.empty() && tmp != '/')

{

st.pop();

tmp = st.top();

}

if(st.empty())

st.push('/');

}

else

st.push(str[i]);

}

}

//处理结果

result.append(st.size(),'c');

i=st.size()-1;

while(!st.empty())

{

tmp = st.top();

st.pop();

result[i--] = tmp;

}

return result;

}

int main()

{

string str("/a/./b/../../c/");

//string str("/../");

cout<<SimplifyPath(str);

return 0;

}

## 迷宫