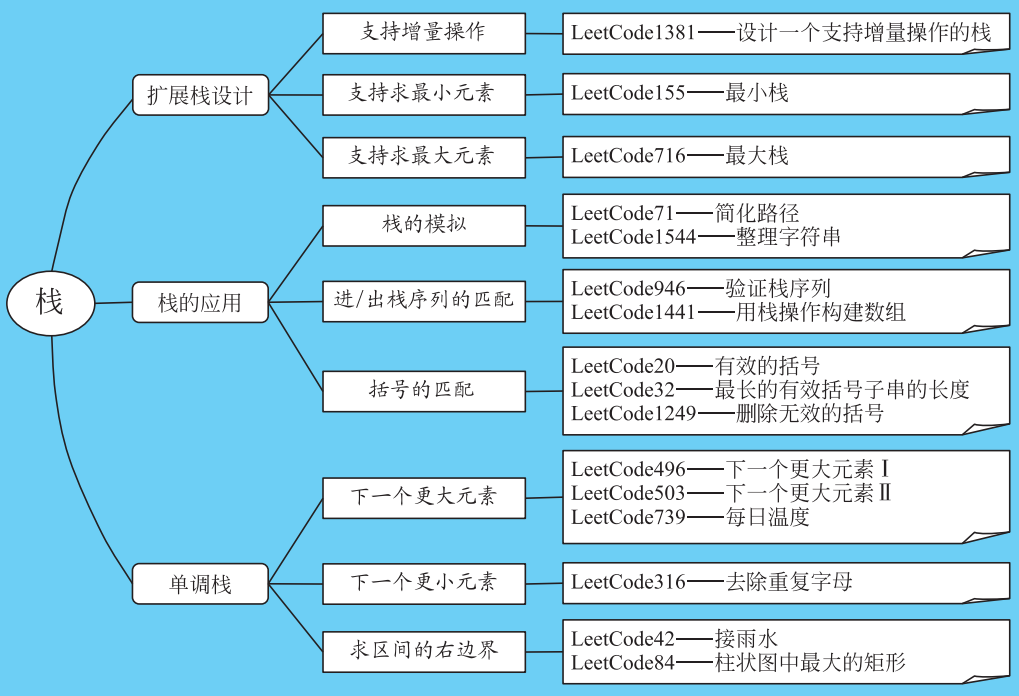
第3章 栈

知识图谱


栈
扩展栈设计
支持增量操作
LeetCode1381——设计一个支持增量操作的栈
支持求最小元素
LeetCode155——最小栈
支持求最大元素
LeetCode716——最大栈
栈的应用
栈的模拟
LeetCode71——简化路径
LeetCode1544——整理字符串
进/出栈序列的匹配
LeetCode946——验证栈序列
LeetCode1441——用栈操作构建数组
括号的匹配
LeetCode20——有效的括号
LeetCode32——最长的有效括号子串的长度
LeetCode1249——删除无效的括号
单调栈
下一个更大元素
LeetCode496——下一个更大元素 I
LeetCode503——下一个更大元素 II
LeetCode739——每日温度
下一个更小元素
LeetCode316——去除重复字母
求区间的右边界
LeetCode42——接雨水
LeetCode84——柱状图中最大的矩形

3.1 栈 概 述

3.1.1 栈的定义

栈的示意图如图 3.1 所示，栈中保存一个数据序列[a_0，a_1，…，a_{n-1}]，共有两个端点，栈底的一端（a_0 端）不动，栈顶的一端（a_{n-1} 端）是动态的，可以插入（进栈）和删除（出栈）元素。栈元素遵循“先进后出”的原则，即最先进栈的元素最后出栈。注意，不能直接对栈中的元素进行顺序遍历。

在算法设计中，栈通常用于存储临时数据。在一般情况下，若先存储的元素后处理，则使用栈数据结构存储这些元素。

n 个不同的元素经过一个栈产生不同出栈序列的个数为$\frac{1}{n+1}C_{2n}^{n}$，这称为第 n 个 Catalan（卡特兰）数。

栈可以使用数组和链表存储结构实现，使用数组实现的栈称为顺序栈，使用链表实现的栈称为链栈。

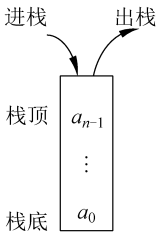


图 3.1 栈的示意图

3.1.2 栈的知识点

1. C++中的栈

在 C++ STL 中提供了 stack 类模板，实现了栈的基本运算，而且在使用时不必考虑栈满上溢出的情况，但不能顺序遍历栈中的元素。其主要的成员函数如下。

- empty()：判断栈是否为空，当栈中没有元素时返回 true，否则返回 false。
- size()：返回栈中当前元素的个数。
- top()：返回栈顶的元素。
- push(x)：将元素 x 进栈。
- pop()：出栈元素，只是删除栈顶元素，并不返回该元素。

例如，以下代码定义一个整数栈 st，依次进栈 1、2、3，再依次出栈所有元素，出栈结果是 3，2，1。

C++：

```
stack<int> st;                  //定义一个整数栈
st.push(1);                     //进栈 1
st.push(2);                     //进栈 2
st.push(3);                     //进栈 3
while(!st.empty()) {            //栈不空时循环
    printf("%d ",st.top());     //输出栈顶元素
    st.pop();                   //出栈栈顶元素
}
```

另外，由于 C++ STL 中的 vector 容器提供了 empty()、push_back()、pop_back()和 back()等成员函数，也可以将 vector 容器用作栈。

2. Python 中的栈

在 Python 中没有直接提供栈，可以将列表作为栈。当定义一个作为栈的 st 列表后，其主要的栈操作如下。

- st，len(st)==0：判断栈是否为空，当栈中没有元素时返回 True，否则返回 False。
- len(st)：返回栈中当前元素的个数。
- st[-1]：返回栈顶的元素。
- st.append(x)：将元素 x 进栈。
- st.pop()：出栈元素，只是删除栈顶元素，并不返回该元素。

说明：在 Python 中提供了双端队列 deque，可以通过限制 deque 在同一端插入和删除将其作为栈使用。

例如，以下代码用列表 st 作为一个整数栈，依次进栈 1、2、3，再依次出栈所有元素，出栈结果是 3，2，1。

Python：

```
st=[]                                  #将列表作为栈
st.append(1)                           #进栈 1
st.append(2)                           #进栈 2
st.append(3)                           #进栈 3
while st:                              #栈不空时循环，或者改为 while len(st)>0:
    print(st[-1],end=' ')              #输出栈顶元素
    st.pop()                           #出栈栈顶元素
```

3. 单调栈

将一个从栈底元素到栈顶元素有序的栈称为单调栈，如果从栈底元素到栈顶元素是递增的（栈顶元素最大），称该栈为单调递增栈或者大顶栈，例如[1，2，3]（栈底为 1，栈顶为 3）就是一个单调递增栈；如果从栈底元素到栈顶元素是递减的（栈顶元素最小），称该栈为单调递减栈或者小顶栈，例如[3，2，1]就是一个单调递减栈。

维护单调栈的操作是在向栈中插入元素时可能需要出栈元素，以保持栈中元素的单调性。这里以单调递增栈（大顶栈）为例，假设要插入的元素是 x：

（1）如果栈顶元素小于（或者等于）x，说明插入 x 后仍然保持单调性。直接进栈 x 即可。

（2）如果栈顶元素大于 x，说明插入 x 后不再满足单调性，需要出栈栈顶元素，直到栈为空或者满足（1）的条件，再将 x 进栈。

例如，以下代码中定义的单调栈 st 是一个单调递增栈，执行后 st 从栈底到栈顶的元素为[1，2，3]。

C++：

```
vector<int> a={1,5,2,4,3};
stack<int> st;
for(int i=0;i<a.size();i++) {                  //遍历 a
    while(!st.empty() && st.top()>a[i])
        st.pop();                              //将大于 a[i]的栈顶元素出栈
    st.push(a[i]);
}
```

单调栈的主要用途是寻找下/上一个更大/更小元素，例如给定一个整数数组 a，求每个元素的下一个更大元素，可以使用单调递减栈实现。由于在单调栈应用中通常每个元素仅进栈和出栈一次，所以时间复杂度为 $O(n)$。

3.2 扩展栈的算法设计

3.2.1 LeetCode1381——设计一个支持增量操作的栈★★

【问题描述】 请设计一个支持以下操作的栈。

(1) CustomStack(int maxSize)：用 maxSize 初始化对象，其中 maxSize 是栈中最多能容纳的元素的数量，栈在增长到 maxSize 之后将不支持 push 操作。

(2) void push(int x)：如果栈还未增长到 maxSize，将 x 添加到栈顶。

(3) int pop()：弹出栈顶元素，并返回栈顶的值，或栈为空时返回－1。

(4) void increment(int k, int val)：栈底的 k 个元素的值都增加 val。如果栈中元素的总数小于 k，则栈中的所有元素都增加 val。

示例：

```
CustomStack customStack=new CustomStack(3);     //栈的容量为 3,初始为空栈[]
customStack.push(1);                            //栈变为[1]
customStack.push(2);                            //栈变为[1, 2]
customStack.pop();                              //返回栈顶值 2,栈变为[1]
customStack.push(2);                            //栈变为[1, 2]
customStack.push(3);                            //栈变为[1, 2, 3]
customStack.push(4);     //栈是[1, 2, 3],不能添加其他元素使栈的大小变为 4
customStack.increment(5, 100);                  //栈变为[101, 102, 103]
customStack.increment(2, 100);                  //栈变为[201, 202, 103]
customStack.pop();                              //返回栈顶值 103,栈变为[201, 202]
customStack.pop();                              //返回栈顶值 202,栈变为[201]
customStack.pop();                              //返回栈顶值 201,栈变为[]
customStack.pop();                              //栈为空,返回－1
```

【限制】 $1 \leqslant \text{maxSize} \leqslant 1000$，$1 \leqslant x \leqslant 1000$，$1 \leqslant k \leqslant 1000$，$0 \leqslant \text{val} \leqslant 100$。increment、push、pop 操作均最多被调用 1000 次。

【解题思路】 使用顺序栈实现支持增量操作的栈。按题目要求，顺序栈中存放栈元素的 data 数组使用动态数组，栈顶指针为 top(初始为－1)，另外设置一个表示容量的 capacity 域(其值为 maxSize)，如图 3.2 所示。

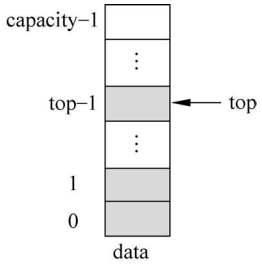


图 3.2 顺序栈结构

(1) push 和 pop 运算算法与基本顺序栈的进/出栈元素算法相同，算法的时间复杂度为 $O(1)$。

(2) increment(k,val)用于将 data 数组中下标为 0～min(top＋1,k)－1 的所有元素增加 val，算法的时间复杂度为 $O(k)$。

对应的支持增量操作的类如下。

C++：

```
1  class CustomStack {
2      int *data;                          //存放栈元素的动态数组
```

```
    int capacity;                          //data 数组的容量
    int top;                               //栈顶指针
public:
    CustomStack(int maxSize) {             //构造函数
        data=new int[maxSize];
        capacity=maxSize;
        top=-1;
    }
    void push(int x) {                     //进栈 x
        if(top+1<capacity) {               //没有出现上溢出时
            top++;data[top]=x;
        }
    }
    int pop() {                            //出栈
        if(top==-1)
            return -1;
        else {
            int e=data[top]; top--;
            return e;
        }
    }
    void increment(int k,int val) {        //增量操作
        int m=(top+1<=k?top+1:k);
        for(int i=0;i<m;i++)
            data[i]+=val;
    }
};
```

提交运行：

```
结果:通过;时间:28ms;空间:20.5MB
```

Python：

```
class CustomStack:
    def __init__(self, maxSize: int):
        self.data=[None] * maxSize          #存放栈元素的动态数组
        self.capacity=maxSize               #data 数组的容量
        self.top=-1                         #栈顶指针
    def push(self, x: int) -> None:         #进栈 x
        if self.top+1<self.capacity:        #没有出现上溢出时
            self.top=self.top+1
            self.data[self.top]=x
    def pop(self) -> int:                   #出栈
        if self.top==-1:
            return -1
        else:
            e=self.data[self.top]
            self.top=self.top-1
            return e
    def increment(self, k: int, val: int) -> None:   #增量操作
        m=k
        if self.top+1<=k:m=self.top+1
        for i in range(0,m):
            self.data[i]+=val
```

提交运行：

```
结果:通过;时间:104ms;空间:16MB
```

3.2.2 LeetCode155——最小栈★

【问题描述】 设计一个支持 push、pop、top 操作，并且能在常数时间内检索到最小元素的栈，各函数的功能如下。

(1) push(x)：将元素 x 推入栈中。

(2) pop()：删除栈顶元素。

(3) top()：获取栈顶元素。

(4) getMin()：检索栈中的最小元素。

示例：

```
MinStack minStack=new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin();          //返回-3
minStack.pop();
minStack.top();             //返回 0
minStack.getMin();          //返回-2
```

【限制】 $-2^{31} \leqslant val \leqslant 2^{31}-1$，pop、top 和 getMin 操作总是在非空栈上调用，push、pop、top 和 getMin 操作最多被调用 3×10^4 次。

扫一扫

源程序

【解法 1】 使用链栈实现最小栈。使用带头结点 h 的单链表作为最小栈，其中每个结点除了存放栈元素(val 域)外，还存放当前最小的元素(minval 域)。例如，若当前栈中从栈底到栈顶的元素为 $a_0,a_1,\cdots,a_i$ ($i\geqslant1$)，则 val 序列为 $a_0,a_1,\cdots,a_i$，minval 序列为 $b_0,b_1,\cdots,b_i$，其中 b_i 恰好为 $a_0,a_1,\cdots,a_i$ 中的最小元素。若栈非空，在进栈元素 a_i 时求 b_i 的过程如图 3.3 所示。

(1) push(val)：创建用 val 域存放 val 的结点 p，若链表 h 为空或者 val 小于或等于首结点的 minval，则置 p->minval=val，否则置 p->minval 为首结点的 minval。最后将结点 p 插入表头作为新的首结点。

(2) pop()：删除链表 h 的首结点。

(3) top()：返回链表 h 的首结点的 val 值。

(4) getMin()：返回链表 h 的首结点的 minval。

可以看出上述 4 个基本算法的时间复杂度均为 $O(1)$。

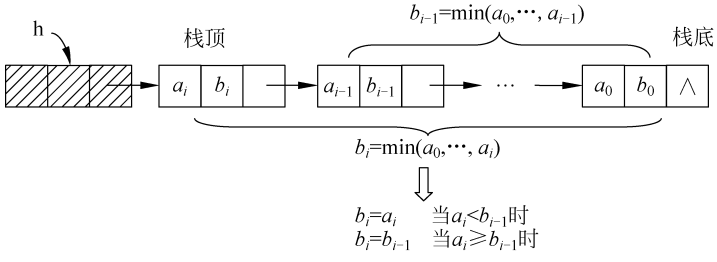


图 3.3 非空栈进栈元素 a_i 时求 b_i 的过程

扫一扫

源程序

【解法 2】 使用顺序栈实现最小栈。用两个 vector<int>容器 valst 和 minst 实现最小栈，valst 作为 val 栈(主栈)，minst 作为 min 栈(辅助栈)，后者作为存放当前最小元素的辅

助栈，如图 3.4 所示，min 栈的栈顶元素 y 表示 val 栈中从栈顶 x 到栈底的最小元素，相同的最小元素也要进入 min 栈。例如，依次进栈 3、5、3、1、3、1 后的状态如图 3.5 所示，从中看出，min 栈中元素的个数总是少于或等于 val 栈中元素的个数。

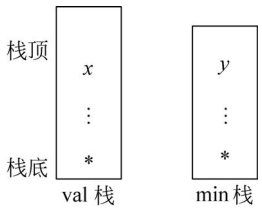


图 3.4　最小栈的结构

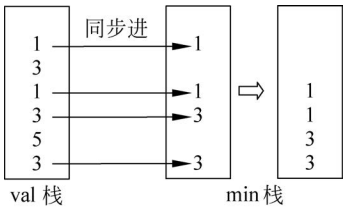


图 3.5　依次进栈 3、5、3、1、3、1 后的状态

(1) push(val)：将 val 进入 val 栈，若 min 栈为空或者 val 小于或等于 min 栈的栈顶元素，则同时将 val 进入 min 栈。

(2) pop()：从 val 栈出栈元素 e，若 min 栈的栈顶元素等于 e，则同时从 min 栈出栈元素 e。

(3) top()：返回 val 栈的栈顶元素。

(4) getMin()：返回 min 栈的栈顶元素。

同样，上述 4 个基本算法的时间复杂度均为 $O(1)$。

【解法 3】　实现辅助空间为 $O(1)$的最小栈。前面两种解法的辅助空间均为 $2n$(在解法 1 中每个结点存放两个整数，共 $2n$ 个结点，而解法 2 中使用两个栈，每个栈的空间为 n)，本解法只使用一个栈 st(初始为空)，另外设计一个存放当前最小值的变量 minval(初始为 -1)，栈 st 中仅存放进栈元素与 minval 的差，这样的辅助空间为 n。

(1) push(val)：栈 st 为空时进栈 0，置 minval=val；否则求出 val 与 minval 的差值 d，将 d 进栈，若 $d<0$，说明 val 更小，置 minval=val(或者说 st 栈的栈顶元素为 d，若 $d<0$，说明实际栈顶元素就是 minval，否则实际栈顶元素是 $d+$minval)。

(2) pop()：出栈栈顶元素 d，若 $d<0$，说明栈顶元素最小，需要更新 minval，即置 minval$-=d$，否则 minval 不变。

(3) top()：设栈顶元素为 d，若 $d<0$，返回 minval，否则返回 $d+$minval。

(4) getMin()：栈为空时返回 -1，否则返回 minval。

例如，st=[]，minval=-1，一个操作示例如下。

(1) push(-2)：将 -2 进栈，st=[0]，minval=-2。

(2) push(1)：将 1 进栈，st=[0,3]，minval=-2。

(3) push(-4)：将 -4 进栈，st=[0,3,-2]，minval=-4。

(4) push(2)：将 2 进栈，st=[0,3,-2,6]，minval=-4。

(5) top()：栈顶为 $6\geqslant 0$，则实际栈顶元素为 $6+$minval$=2$。

(6) getMin()：直接返回 minval=-4。

(7) pop()：从 st 栈中出栈 $d=6$，st=[0,3,-2]，由于 $d>0$，说明最小栈元素不变，minval=-4。

(8) top()：栈顶为 $-2<0$，说明实际栈顶元素就是 minval=-4。

由于测试数据中进栈的元素出现 2147483646 和 -2147483648 的情况，在做加减运算时超过 int 的范围，所以将 int 改为 long long 类型。对应的算法如下。

C++：

```
typedef long long LL;
class MinStack {
    stack<LL> st;
    LL minval=-1;
public:
    MinStack() {  }

    void push(LL val) {
        if(!st.empty()) {                    //st 非空
            LL d=val-minval;
            st.push(d);
            if(d<0) minval=val;
        }
        else {                               //st 为空
            st.push(0);
            minval=val;
        }
    }

    void pop() {
        LL d=st.top(); st.pop();
        if(d<0) minval-=d;
    }

    int top() {
        if(st.top()<0) return minval;
        else return st.top()+minval;
    }

    int getMin() {
        if(st.empty()) return -1;
        else return minval;
    }
};
```

提交运行：

结果：通过；时间：12ms；空间：15.8MB

Python：

```
class MinStack:
    def __init__(self):
        self.st=[]
        self.minval=-1

    def push(self, val: int) -> None:
        if len(self.st)>0:                   #st 非空
            d=val-self.minval
            self.st.append(d)
            if d<0: self.minval=val
        else:                                #st 为空
            self.st.append(0)
            self.minval=val

    def pop(self) -> None:
        d=self.st.pop()
        if d<0: self.minval=self.minval-d

    def top(self) -> int:
        if self.st[-1]<0:return self.minval
        else:return self.st[-1]+self.minval
```

```
19        def getMin(self) -> int:
20            if len(self.st)==0: return -1
21            else: return self.minval
```

提交运行：

```
结果:通过;时间:60ms;空间:18.4MB
```

3.2.3 LeetCode716——最大栈★★★

【问题描述】 设计一个最大栈数据结构，其既支持栈操作，又支持查找栈中的最大元素。

(1) MaxStack()：初始化栈对象。

(2) void push(int x)：将元素 x 压入栈中。

(3) int pop()：移除栈顶元素并返回该元素。

(4) int top()：返回栈顶元素，无须移除。

(5) int peekMax()：检索并返回栈中的最大元素，无须移除。

(6) int popMax()：检索并返回栈中的最大元素，并将其移除。如果有多个最大元素，只要移除最靠近栈顶的那一个。

示例：

```
MaxStack stk=new MaxStack();
stk.push(5);              //[5],5 既是栈顶元素,也是最大元素
stk.push(1);              //[5, 1],栈顶元素是 1,最大元素是 5
stk.push(5);              //[5, 1, 5],5 既是栈顶元素,也是最大元素
stk.top();                //返回 5,[5, 1, 5],栈没有改变
stk.popMax();             //返回 5,[5, 1],栈发生改变,栈顶元素不再是最大元素
stk.top();                //返回 1,[5, 1],栈没有改变
stk.peekMax();            //返回 5,[5, 1],栈没有改变
stk.pop();                //返回 1,[5],此操作后 5 既是栈顶元素,也是最大元素
stk.top();                //返回 5,[5],栈没有改变
```

【限制】 $-10^7 \leqslant x \leqslant 10^7$，最多调用 10^4 次 push、pop、top、peekMax 和 popMax，在调用 pop、top、peekMax 或 popMax 时栈中至少存在一个元素。

进阶：试着设计这样的解决方案，调用 top 方法的时间复杂度为 $O(1)$，调用其他方法的时间复杂度为 $O(\log_2 n)$。

扫一扫

源程序

【解法 1】 使用两个 vector 向量实现最大栈。设计两个 vector < int >向量 valst 和 maxst，其中 valst 作为 val 栈(主栈)，maxst 作为 max 栈(辅助栈)，后者作为存放当前最大元素的辅助栈，两个栈中元素的个数始终相同，若 val 栈从栈底到栈顶的元素为 $a_0, a_1, \cdots, a_i$，max 栈从栈底到栈顶的元素为 $b_0, b_1, \cdots, b_i$，则 b_i 始终为 a_0 到 a_i 中的最大元素，如图 3.6 所示。

例如，依次进栈元素 2、1、5、3、9，则 valst=[2,1,5,3,9]，而 maxst=[2,2,5,5,9](栈底到栈顶的顺序)。

(1) push(x)：将 x 进入 val 栈，若 max 栈为空，则将 x 进入 max 栈；若 max 栈不为空，则将 x 和 min 栈的栈顶元素的最大值进入 max 栈。

(2) pop()：从 val 栈出栈元素 e，同时从 max 栈出栈栈顶元素，返回 e。

(3) top()：返回 val 栈的栈顶元素。

(4) peekMax()：返回 max 栈的栈顶元素。

(5) popMax()：取 max 栈的栈顶元素 e，从 val 栈出栈元素直到 e，将出栈的元素进入

临时栈 tmp，同时 max 栈做同步出栈操作。当 val 栈遇到元素 e 时，val 栈出栈元素 e，max 栈做一次出栈操作，再出栈 tmp 中的所有元素并且调用 push 将其进栈。最后返回 e。

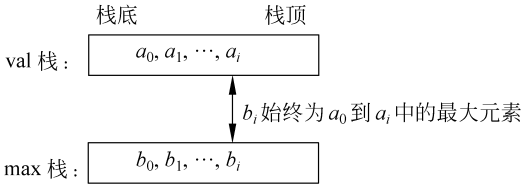


图 3.6 最大栈结构(1)

本解法的程序在提交时超时。

【解法 2】 使用一个 list 链表和一个 map 映射实现最大栈。设计一个 list < int >链表容器作为 val 栈(每个结点对应一个地址，将其尾部作为栈顶)，另外设计一个 map < int，vector < list < int > : : iterator >>映射容器 mp 作为辅助结构，其关键字为进栈的元素值 e，值为 e 对应的结点的地址(按进栈的先后顺序排列)。由于 mp 使用红黑树结构，默认按关键字递增排列，如图 3.7 所示。

(1) push(x)：将 x 进入 val 栈，将该结点的地址添加到 mp[x]的末尾。

(2) pop()：从 val 栈出栈元素 e，同时从 mp[e]中删除尾元素，最后返回 e。

(3) top()：返回 val 栈的栈顶元素。

(4) peekMax()：返回 mp 中的最大关键字。

(5) popMax()：获取最大 mp 中的最大关键字 e 及其地址 it，同时从 mp[e]中删除尾元素，再从 valst 中删除地址为 it 的结点，最后返回 e。

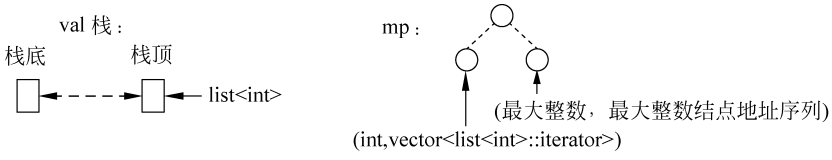


图 3.7 最大栈结构(2)

对应的 MaxStack 类如下。

C++：

```
1  class MaxStack {
2  public:
3      list < int > valst;                          //用链表作为 val 栈
4      map < int, vector < list < int > : : iterator >> mp;   //映射
5      MaxStack( ) {}                               //构造函数

6      void push(int x) {                           //进栈 x
7          valst.push_back(x);
8          mp[x].push_back(--valst.end());
9      }

10     int pop() {                                  //出栈
11         int e=valst.back();
12         mp[e].pop_back();
13         if(mp[e].empty()) mp.erase(e);  //当 mp[e]为空时删除该元素
14         valst.pop_back();
15         return e;
16     }

17     int top() {                                  //取栈顶元素
```

```
        return valst.back();
    }

    int peekMax() {                          //取栈中的最大元素
        return mp.rbegin()-> first;
    }

    int popMax() {                           //出栈最大元素
        int e=mp.rbegin()-> first;
        auto it=mp[e].back();
        mp[e].pop_back();
        if(mp[e].empty()) mp.erase(e);       //当 mp[e]为空时删除该元素
        valst.erase(it);
        return e;
    }
};
```

提交运行：

```
结果:通过;时间:368ms;空间:145.8MB
```

说明：在上述结构中 popMax()算法的时间复杂度为 $O(\log_2 n)$，因此没有超时。

Python：

```
from sortedcontainers import SortedList
class MaxStack:
    def __init__(self):
        self.idx=0                           #栈中元素的个数
        self.valst=dict()                    #作为 val 栈,元素为(序号,值)
        self.sl=SortedList()                 #有序序列,元素为(值,序号),按值递增排列

    def push(self, x: int) -> None:          #进栈 x
        self.valst[self.idx]=x
        self.sl.add((x, self.idx))
        self.idx+=1

    def pop(self) -> int:                    #出栈
        i, x = self.valst.popitem()
        self.sl.remove((x, i))
        return x

    def top(self) -> int:                    #取栈顶元素
        return next(reversed(self.valst.values()))

    def peekMax(self) -> int:                #取栈中的最大元素
        return self.sl[-1][0]

    def popMax(self) -> int:                 #出栈最大元素
        x, i = self.sl.pop()
        self.valst.pop(i)
        return x
```

提交运行：

```
结果:通过;时间:572ms;空间:57.7MB
```

3.3 栈应用的算法设计

3.3.1 LeetCode1544——整理字符串★★

【问题描述】 给定一个由大/小写英文字母组成的字符串 s，整理该字符串。一个整理

好的字符串中的两个相邻字符 s[i]和 s[i+1](0≤i≤s.length−2)要满足以下条件：

(1) 若 s[i]是小写字母，则 s[i+1]不可以是相同的大写字母。

(2) 若 s[i]是大写字母，则 s[i+1]不可以是相同的小写字母。

每次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，直到将字符串整理好为止。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。

例如，s="abBAcC"，一种整理过程是"a**bB**AcC"→"**aA**cC"→"cC"→""，答案为""。

【限制】 1≤s.length≤100，s 只包含小写英文字母和大写英文字母。

【解题思路】 使用栈模拟求解。设计一个字符栈 st，用 i 遍历 s，若栈不为空，当 s[i]与栈顶字符满足题目中给定的任意一个条件时出栈栈顶字符(相当于删除 s[i]和栈顶一对字符)，否则将 s[i]进栈。s 遍历完毕，将栈中的所有字符按从栈顶到栈底的顺序合并起来得到答案。对应的算法如下。

C++：

```
class Solution {
public:
    string makeGood(string s) {
        int n=s.size();
        stack<char> st;
        for(int i=0;i<n;i++) {
            if(!st.empty() && tolower(st.top())==tolower(s[i]) && st.top()!=s[i])
                st.pop();
            else
                st.push(s[i]);
        }
        string ans="";
        while(!st.empty()) {
            ans=st.top()+ans;
            st.pop();
        }
        return ans;
    }
};
```

提交运行：

```
结果:通过;时间:4ms;空间:6.9MB
```

另外，也可以直接用字符串 ans 模拟栈操作，这样在 s 遍历完毕，ans 中剩下的字符串就是答案。对应的算法如下。

C++：

```
class Solution {
public:
    string makeGood(string s) {
        int n=s.size();
        string ans="";
        for(int i=0;i<n;i++) {
            if(!ans.empty() && tolower(ans.back())==tolower(s[i]) && ans.back()!=s[i])
                ans.pop_back();
            else
                ans.push_back(s[i]);
        }
```

```
            return ans;
        }
    };
```

提交运行：

```
结果:通过;时间:0ms;空间:6MB
```

采用后一种方法对应的 Python 算法如下。

Python：

```
class Solution:
    def makeGood(self, s: str) -> str:
        ans=[]
        for i in range(0,len(s)):
            if len(ans)>0 and ans[-1].lower()==s[i].lower() and ans[-1]!=s[i]:
                ans.pop()
            else:
                ans.append(s[i])
        return ''.join(ans)
```

提交运行：

```
结果:通过;时间:40ms;空间:15.1MB
```

3.3.2 LeetCode71——简化路径★★

【问题描述】 给定一个字符串 path，表示指向某一文件或目录的 UNIX 风格绝对路径（以 '/' 开头），请将其转换为更加简洁的规范路径并返回得到的规范路径。在 UNIX 风格的文件系统中，一个点（.）表示当前目录本身；两个点（..）表示将目录切换到上一级（指向父目录）；两者都可以是复杂相对路径的组成部分。任意多个连续的斜线（即'//'）都被视为单个斜线 '/'。对于此问题，任何其他格式的点（例如，'...'）均被视为文件/目录名称。注意，返回的规范路径必须遵循以下格式：

（1）始终以斜线'/'开头。

（2）两个目录名之间只有一个斜线'/'。

（3）最后一个目录名（如果存在）不能以 '/' 结尾。

（4）路径仅包含从根目录到目标文件或目录的路径上的目录（即不含 '.' 或 '..'）。

扫一扫

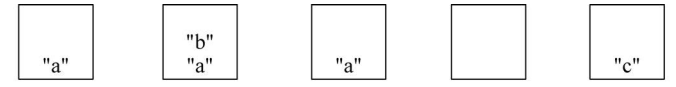
源程序

例如，path="/../"，答案为"/"，从根目录向上一级是不可行的，因为根目录是可以到达的最高级。path="/a/./b/../../c/"，答案为"/c"。

【限制】 1≤path.length≤3000，path 由英文字母、数字、'.'、'/'或'_'组成，并且是一个有效的 UNIX 风格绝对路径。

【解题思路】 用栈模拟进入下一级目录（前进）和上一级目录（回退）操作。先将 path 按'/'分隔符提取出对应的字符串序列，例如，path="/a/./b/../../c/"时对应的字符串序列为""，"a"，"."，"b"，".."，".."，"c"。然后定义一个字符串栈 st，用 word 遍历所有字符串，若 word="."，则跳过；若 word=".."，则回退，即出栈一次；若 word 为其他非空字符串，则将其进栈。遍历前面字符串序列的结果如图 3.8 所示。

遍历完毕将栈中的所有字符串按从栈底到栈顶的顺序加上"\"并连接起来构成 ans，这里 ans="/c"，最后返回 ans。



(a) "a"进栈　(b) "b"进栈　(c) 出栈　(d) 出栈　(e) "c"进栈

图 3.8　遍历字符串序列的结果

3.3.3　LeetCode1441——用栈操作构建数组★

【问题描述】　给定一个目标数组 target 和一个整数 n，每次迭代，需要从 list＝{1,2,3,…,n}中依次读取一个数字，请使用下述操作构建目标数组 target。

(1) Push：从 list 中读取一个新元素，并将其推入数组中。

(2) Pop：删除数组中的最后一个元素。

如果目标数组构建完成，停止读取更多元素。题目数据保证目标数组严格递增，并且只包含 1～n 的数字。请返回构建目标数组所用的操作序列。题目数据保证答案是唯一的。

例如，输入 target＝[1,3]，n＝3，输出为["Push","Push","Pop","Push"]。读取 1 并自动推入数组->[1]，读取 2 并自动推入数组，然后删除它->[1]，读取 3 并自动推入数组->[1,3]。

【限制】　1≤target.length≤100，1≤target[i]≤100，1≤n≤100。target 是严格递增的。

【解题思路】　使用栈实现两个序列的简单匹配。用 j 从 0 开始遍历 target，用 i 从 1 到 n 循环：先将 i 进栈(每个 i 总是要进栈的，这里并没有真正设计一个栈，栈操作是通过"Push"和"Pop"表示的，存放在结果数组 ans 中)，若当前进栈的元素 i 不等于 target[j]，则将 i 出栈，否则 j 加 1 继续比较，如图 3.9 所示。若 target 中的所有元素处理完毕，退出循环。最后返回 ans。

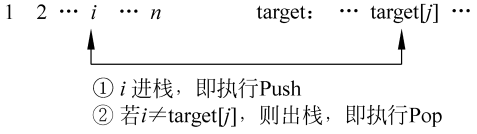


图 3.9　栈操作过程

对应的算法如下。

C++：

```
class Solution {
public:
    vector<string> buildArray(vector<int>& target, int n) {
        vector<string> ans;
        int j=0;                            //遍历 target 数组
        for(int i=1;i<=n;i++) {
            ans.push_back("Push");          //i 进栈
            if(i!=target[j])                //target 数组的当前元素不等于 i
                ans.push_back("Pop");       //出栈 i
            else                            //target 数组的当前元素等于 i
                j++;
            if(j==target.size())            //target 数组遍历完后退出循环
                break;
```

```
            }
            return ans;
        }
    };
```

提交运行：

```
结果:通过;时间:4ms;空间:7.6MB
```

Python：

```
class Solution:
    def buildArray(self, target: List[int], n: int) -> List[str]:
        ans=[]
        j=0                                  #遍历 target 数组
        for i in range(1,n+1):
            ans.append("Push")               #i 进栈
            if i!=target[j]:                 #target 数组的当前元素不等于 i
                ans.append("Pop")            #出栈 i
            else:                            #target 数组的当前元素等于 i
                j=j+1
            if j==len(target):               #target 数组遍历完后退出循环
                break;
        return ans
```

提交运行：

```
结果:通过;时间:36ms;空间:14.9MB
```

3.3.4 LeetCode946——验证栈序列★★

【问题描述】 给定 pushed 和 popped 两个序列，每个序列中的值都不重复，只有当它们可能是在最初空栈上进行的推入 push 和弹出 pop 操作序列的结果时返回 true，否则返回 false。

例如，pushed=[1,2,3,4,5]，popped=[4,5,3,2,1]，输出为 true；pushed=[1,2,3,4,5]，popped=[4,3,5,1,2]，输出为 false。

【限制】 0≤pushed.length=popped.length≤1000，0≤pushed[i]，popped[i]<1000，并且 pushed 是 popped 的一个排列。

【解题思路】 使用栈实现两个序列的简单匹配。先考虑这样的情况，假设 a 和 b 序列中均含 $n(n\geqslant 1)$ 个整数，都是 1～n 的某个排列，现在要判断 b 是否为以 a 作为输入序列的一个合法的出栈序列，即 a 序列经过一个栈是否得到了 b 的出栈序列。求解思路是先建立一个 st，用 j 遍历 b 序列(初始为 0)，i 从 0 开始遍历 a 序列：

(1) 将 $a[i]$进栈。

(2) 栈不为空并且栈顶元素与 $b[j]$相同时循环：出栈一个元素同时执行 $j++$。

在上述过程结束后，如果栈为空，返回 true 表示 b 序列是 a 序列的出栈序列，否则返回 false 表示 b 序列不是 a 序列的出栈序列。

例如 $a=\{1,2,3,4,5\}$，$b=\{3,2,4,5,1\}$，由 a 产生 b 的过程如图 3.10 所示，所以返回 true。对应的算法如下。

C++：

```
class Solution {
public:
    bool validateStackSequences(vector<int>& pushed, vector<int>& popped) {
        stack<int> st;
        int j=0;
        for(int i=0;i<pushed.size();i++) {          //输入序列没有遍历完
            st.push(pushed[i]);                     //元素 pushed[i]进栈
            while(!st.empty() && st.top()==popped[j]) {
                st.pop();                           //popped[j]与栈顶匹配时出栈
                j++;
            }
        }
        return st.empty();                          //栈为空返回 True,否则返回 False
    }
};
```

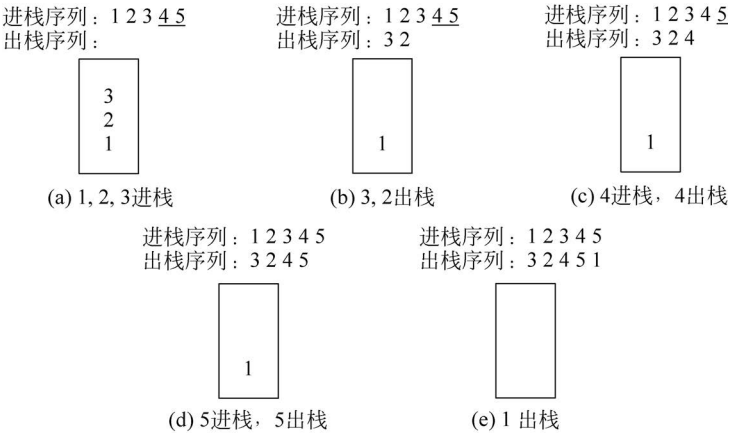


图 3.10　由 $a=\{1,2,3,4,5\}$ 产生 $b=\{3,2,4,5,1\}$ 的过程

提交运行：

```
结果:通过;时间:4ms;空间:14.9MB
```

Python：

```
class Solution:
    def validateStackSequences(self, pushed: List[int], popped: List[int]) -> bool:
        st=[]
        j=0
        for i in range(0,len(pushed)):              #输入序列没有遍历完
            st.append(pushed[i])                    #pushed[i]进栈
            while len(st)>0 and st[-1]==popped[j]:
                st.pop()                            #popped[j]与栈顶匹配时出栈
                j=j+1
        return len(st)==0                           #栈为空返回 True,否则返回 False
```

提交运行：

```
结果:通过;时间:28ms;空间:15.1MB
```

3.3.5　LeetCode20——有效的括号★

【问题描述】　给定一个只包含 '('、')'、'{'、'}'、'['、']' 的字符串 s,判断字符串是否有效。

有效字符串需满足：左括号必须用相同类型的右括号闭合，左括号必须以正确的顺序闭合，每个右括号都有一个对应的相同类型的左括号。

例如，s="()[]{}"时答案为 true，s="(]"时答案为 false。

扫一扫

源程序

【限制】 $1 \leqslant s.length \leqslant 10^4$，s 仅由'('、')'、'['、']'、'{'、'}'组成。

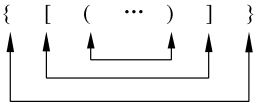


图 3.11 括号的匹配

【解题思路】 使用栈实现括号的最近匹配。字符串 s 中各种括号的匹配如图 3.11 所示，也就是说每个右括号总是与前面最接近的尚未匹配的左括号进行匹配，即满足最近匹配原则，所以用一个栈求解。

先建立仅存放左括号的栈 st，i 从 0 开始遍历 s：

(1) s[i]为任一左括号时将其进栈。

(2) 若 s[i]=')'，如果栈 st 为空，说明该')'无法匹配，返回 false；若栈顶不是'('，同样说明该')'无法匹配，返回 false；否则出栈'('继续判断。

(3) 若 s[i]=']'，如果栈 st 为空，说明该']'无法匹配，返回 false；若栈顶不是'['，同样说明该']'无法匹配，返回 false；否则出栈'['继续判断。

(4) 若 s[i]='}'，如果栈 st 为空，说明该'}'无法匹配，返回 false；若栈顶不是'{'，同样说明该'}'无法匹配，返回 false；否则出栈'{'继续判断。

s 遍历完毕，若栈为空，说明 s 是有效的，返回 true；否则说明 s 是无效的，返回 false。

3.3.6 LeetCode1249——删除无效的括号★★

【问题描述】 给定一个由'('、')'和小写字母组成的字符串 s，从该字符串中删除最少数目的'('或者')'(可以删除任意位置的括号)，使得剩下的括号字符串有效，请返回任意一个有效括号字符串。有效括号字符串应当符合以下任意一条要求：

(1) 空字符串或者只包含小写字母的字符串。

(2) 可以被写成 AB(A 连接 B)的字符串，其中 A 和 B 都是有效括号字符串。

(3) 可以被写成(A)的字符串，其中 A 是一个有效括号字符串。

例如，s="lee(t(c)o)de)"，答案为"lee(t(c)o)de"、"lee(t(co)de)"或者"lee(t(c)ode)"；s="))(("，答案为""。

【限制】 $1 \leqslant s.length \leqslant 10^5$，s[$i$]可能是'('、')'或者英文小写字母。

【解题思路】 使用栈实现括号的最近匹配。定义一个栈 st(保存所有无效的括号)，循环处理 s 中的每个字符 ch=s[i]：如果 ch 为'('，将其序号 i 进栈；如果 ch 为右括号')'，且栈顶为'('，出栈栈顶元素(说明这一对括号是匹配的)；否则将其序号 i 进栈，其他字符直接跳过。最后从栈顶到栈底处理栈中的所有序号，依次删除其相应的无效括号。例如，s="lee(t(c)o)de)"，通过栈 st 找到两对匹配的括号，遍历完毕栈中只含有最后一个右括号的下标，即 st=[12]，删除该无效括号后 s="lee(t(c)o)de"，如图 3.12 所示。

对应的算法如下。

C++：

```
1  class Solution {
2  public:
3      string minRemoveToMakeValid(string s) {
4          stack<int> st;
```

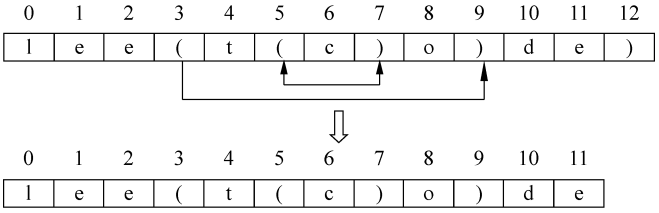


图 3.12 删除无效的括号

```
            for(int i=0;i<s.size();i++) {
                char ch=s[i];
                if(ch=='(') st.push(i);
                else if(ch==')') {
                    if(!st.empty() && s[st.top()]=='(') st.pop();
                    else st.push(i);                    //将未匹配的')'的位置进栈
                }
            }
            while(!st.empty()) {
                int i=st.top(); st.pop();
                s.erase(s.begin()+i);
            }
            return s;
        }
    };
```

提交运行：

```
结果:通过;时间:32ms;空间:10.6MB
```

Python：

```
class Solution:
    def minRemoveToMakeValid(self, s: str) -> str:
        st=[]
        for i in range(0,len(s)):
            ch=s[i]
            if ch=='(':st.append(i)
            elif ch==')':
                if len(st)>0 and s[st[-1]]=='(': st.pop()
                else: st.append(i)
        ns=list(s)
        while len(st)>0:
            i=st[-1]; st.pop()
            ns.pop(i)
        return ''.join(ns)
```

提交运行：

```
结果:通过;时间:260ms;空间:16.6MB
```

3.3.7 LeetCode32——最长的有效括号子串的长度★★★

【问题描述】 给定一个只包含'('和')'的字符串 s，找出最长有效(格式正确且连续)括号子串的长度。

例如，s="")()())"，答案为 4，其中最长有效括号子串是"()()"；s=""时答案为 0。

【限制】 0≤s.length≤3×10^4，s[i]为'('或者')'。

【解题思路】 使用栈实现括号的最近匹配。使用一个栈在遍历字符串 s 的过程中判断到目前为止扫描的子串的有效性，同时得到最长有效括号子串的长度。具体做法是始终保持栈底元素为当前最后一个没有被匹配的右括号的下标，这样主要是考虑了边界条件的处理，栈中的其他元素维护左括号的下标，用 i 从 0 开始遍历字符串 s：

(1) 若 s[i]='('，将下标 i 进栈。

(2) 若 s[i]=')'，出栈栈顶元素，表示匹配了当前的右括号。如果栈为空，说明当前的右括号为没有被匹配的右括号，将其下标进栈，以更新最后一个没有被匹配的右括号的下标；如果栈不为空，当前右括号的下标减去栈顶元素即为以该右括号为结尾的最长有效括号子串的长度，在所有这样的长度中求最大值 ans，最后返回 ans 即可。需要注意的是，如果一开始栈为空，当第一个字符为'('时会将其下标进栈，这样栈底就不满足是最后一个没有被匹配的右括号的下标，为了保持统一，初始时往栈中进栈一个值−1。

例如，s="()())"，ans=0，st=[−1]，遍历 s 如下：

(1) s[0]=')'，出栈−1，栈为空，说明该')'没有被匹配，将其下标 0 进栈，st=[0]。

(2) s[1]='('，将其下标 1 进栈，st=[0,1]。

(3) s[2]=')'，出栈 1，此时栈 st=[0]，栈不为空，说明找到匹配的子串"()"，其长度为 2−st.top()=2−0=2，ans=max(ans,2)=2。

(4) s[3]='('，将其下标 3 进栈，st=[0,3]。

(5) s[4]=')'，出栈 3，此时栈 st=[0]，栈不为空，说明找到匹配的子串"()()"，其长度为 4−st.top()=4−0=4，ans=max(ans,4)=4。

(6) s[5]=')'，出栈 0，栈为空，说明该')'没有被匹配，将其下标 5 进栈，st=[5]。

s 遍历完毕，最后答案为 ans=4。对应的算法如下。

C++：

```
class Solution {
public:
    int longestValidParentheses(string s) {
        int n=s.size();
        if(n==0) return 0;
        int ans=0;
        stack<int> st;
        st.push(-1);
        for(int i=0;i<n;i++) {
            if(s[i]=='(') st.push(i);
            else {
                st.pop();
                if(st.empty()) st.push(i);          //更新栈底为最后未匹配的右括号的下标
                else ans=max(ans,i-st.top()); //找到有效括号子串，长度=i-st.top()
            }
        }
        return ans;
    }
};
```

提交运行：

```
结果:通过;时间:0ms;空间:7.2MB
```

Python：

```
class Solution:
    def longestValidParentheses(self, s: str) -> int:
        n=len(s)
        if n==0:return 0
        ans=0
        st=[]
        st.append(-1)
        for i in range(0,n):
            if s[i]=='(': st.append(i)
            else:
                st.pop()
                if len(st)==0:st.append(i)
                else: ans=max(ans,i-st[-1])
        return ans
```

提交运行：

```
结果:通过;时间:44ms;空间:15.7MB
```

3.4 单调栈应用的算法设计

3.4.1 LeetCode503——下一个更大元素Ⅱ ★★

【问题描述】 给定一个循环数组 nums(nums[nums.length-1]的下一个元素是 nums[0])，返回 nums 中每个元素的下一个更大元素。数字 x 的下一个更大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，这意味着应该循环地搜索它的下一个更大的数。如果不存在下一个更大元素，则输出-1。

例如，nums=[1,2,3,4,3]，对应的答案为 ans=[2,3,4,-1,4]，即 nums[0]=1 的下一个循环更大的数是 2，nums[1]=2 的下一个循环更大的数是 3，以此类推。

【限制】 $1 \leqslant nums.length \leqslant 10^4$，$-10^9 \leqslant nums[i] \leqslant 10^9$。

【解题思路】 使用单调栈求下一个更大元素。先不考虑循环，对于 nums[i]，仅求 nums[$i+1..n-1$]中的下一个更大元素。使用单调递减栈保存进栈元素的下标，设计 ans 数组存放答案，即 ans[i]表示 nums[i]的下一个循环更大的数。用 i 从 0 开始遍历 nums 数组，按下标对应的数组元素进行比较，若栈顶为 j 并且 nums[j]<nums[i]，则 nums[j]的下一个更大元素即为 nums[i](因为如果有更靠前的更大元素，那么这些下标将被提前出栈)，其操作如图 3.13 所示。

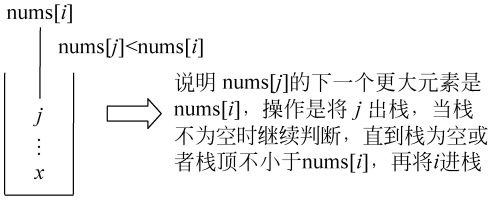


图 3.13 找 nums[j]的下一个更大元素的操作

例如，nums=[1,2,3,4,3]，小顶单调栈 st=[]，求解过程如下：

（1）nums[0]=1，栈为空，直接将序号 0 进栈，st=[0]。

（2）nums[1]=2，栈顶元素 nums[0]=1<2，说明当前元素 2 是栈顶元素的下一个循环更大的数，则出栈栈顶元素 0，置 ans[0]=nums[1]=2，再将序号 1 进栈，st=[1]。

（3）nums[2]=3，栈顶元素 nums[1]=2<3，说明当前元素 3 是栈顶元素的下一个循环更大的数，则出栈栈顶元素 1，置 ans[1]=nums[2]=3，再将序号 2 进栈，st=[2]。

（4）nums[3]=4，栈顶元素 nums[2]=3<4，说明当前元素 4 是栈顶元素的下一个循环更大的数，则出栈栈顶元素 2，置 ans[2]=nums[3]=4，再将序号 2 进栈，st=[3]。

（5）nums[4]=3，栈顶元素 nums[3]=4<3 不成立，将 nums[3]进栈，st=[3,4]。

此时遍历完毕，栈中的元素都没有下一个更大元素，即 ans[3]=ans[4]=-1，最后得到 ans=[2,3,4,-1,-1]。

本题求下一个循环更大的数，所以按上述过程遍历 nums 一次是不够的，还需要回过来考虑前面的元素，一种朴素的思路是把这个循环数组拉直，即复制该数组的前 $n-1$ 个元素拼接在原数组的后面，这样就可以将这个新数组当作普通数组来处理。在本题中实际上不需要显性地将该循环数组拉直，只需要在处理时对下标取模即可。对应的算法如下。

C++：

```
class Solution {
public:
    vector<int> nextGreaterElements(vector<int>& nums) {
        int n=nums.size();
        vector<int> ans(n,-1);
        stack<int> st;                              //单调递减栈
        for(int i=0;i<2*n-1;i++) {
            while(!st.empty() && nums[st.top()]<nums[i%n]) {
                ans[st.top()]=nums[i%n];    //栈顶元素的下一个更大元素为 nums[i%n]
                st.pop();
            }
            st.push(i%n);
        }
        return ans;
    }
};
```

提交运行：

```
结果：通过；时间：36ms；空间：23.2MB
```

Python：

```
class Solution:
    def nextGreaterElements(self, nums: List[int]) -> List[int]:
        n=len(nums)
        ans,st=[-1]*n,[]
        for i in range(0,2*n-1):
            while len(st) and nums[st[-1]]<nums[i%n]:
                ans[st[-1]]=nums[i % n]
                st.pop()
            st.append(i % n);
        return ans
```

提交运行：

结果：通过；时间：104ms；空间：16.5MB

3.4.2 LeetCode496——下一个更大元素Ⅰ★

【问题描述】 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。给定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 $0 \leqslant i <$ nums1.length，找出满足 nums1[i]=nums2[j]的下标 j，并且在 nums2 中确定 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。

例如，nums1=[4,1,2]，nums2=[1,3,4,2]。求 nums1 中每个值的下一个更大元素的过程如下：

(1) nums1[0]=4，对于 nums2[2]=4，无法在 nums2 中找到下一个更大元素，答案是−1。

(2) nums1[1]=1，对于 nums2[0]=1，在 nums2 中找到下一个更大元素为 3，答案是 3。

(3) nums1[2]=2，对于 nums2[3]=2，无法在 nums2 中找到下一个更大元素，答案是−1。

最后答案为 ans=[−1,3,−1]。

【限制】 $1 \leqslant$ nums1.length $\leqslant$ nums2.length $\leqslant 1000$，$0 \leqslant$ nums1[i]，nums2[i] $\leqslant 10^4$，nums1 和 nums2 中的所有整数互不相同，nums1 中的所有整数同样出现在 nums2 中。

【解题思路】 使用单调栈求下一个更大元素。由于 nums2 中的所有元素都不相同，为此设置一个哈希表 hmap，hmap[x]表示 nums2 中元素 x 的下一个更大元素。采用 3.4.1 节的思路，使用单调递减栈(小顶栈)求出 hmap，再遍历 nums1 数组(nums1 是 nums2 的子集)。对于 nums1[i]，若 hmap 中存在 nums1[i]为关键字的元素，说明 nums1[i]在 nums2 中的下一个更大元素为 hmap[nums1[i]]，置 ans[i]=hmap[nums1[i]]；否则说明 nums1[i]在 nums2 中没有下一个更大元素，置 ans[i]=−1。最后返回 ans 即可。对应的算法如下。

C++：

```
class Solution {
public:
    vector<int> nextGreaterElement(vector<int>& nums1, vector<int>& nums2) {
        unordered_map<int,int> hmap;                    //哈希表
        int m=nums1.size(),n=nums2.size();
        stack<int> st;                                  //小顶栈
        for(int i=0;i<n;i++) {
            while(!st.empty() && st.top()<nums2[i]) {
                hmap[st.top()]=nums2[i]; st.pop();      //小于 nums[i]的元素出栈
            }
            st.push(nums2[i]);
        }
        vector<int> ans(m,-1);                          //初始化所有元素为-1
        for(int i=0;i<m;i++) {
            if(hmap.find(nums1[i])==hmap.end()) continue;
            ans[i]=hmap[nums1[i]];
```

```
            }
            return ans;
        }
    };
```

提交运行：

```
结果:通过;时间:4ms;空间:8.5MB
```

Python：

```
class Solution:
    def nextGreaterElement(self, nums1: List[int], nums2: List[int]) -> List[int]:
        dict={}
        m,n=len(nums1),len(nums2)
        st=[]                                    #小顶栈
        for i in range(0,n):
            while len(st)>0 and st[-1]<nums2[i]:
                dict[st[-1]]=nums2[i]
                st.pop()                         #将小于 nums[i]的元素出栈
            st.append(nums2[i])
        ans=[-1]*m
        for i in range(0,m):
            if dict.get(nums1[i])==None: continue;
            ans[i]=dict.get(nums1[i])
        return ans
```

提交运行：

```
结果:通过;时间:36ms;空间:15.1MB
```

3.4.3 LeetCode739——每日温度★★

【问题描述】 给定一个整数数组 a，表示每天的温度，返回一个数组 ans，其中 ans[i]指对于第 i 天下一个更高温度出现在几天后。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，a=[73,74,75,71,69,72,76,73]，对应的答案为 ans=[1,1,4,2,1,1,0,0]。

【限制】 $1 \leqslant a.length \leqslant 10^5$，$30 \leqslant a[i] \leqslant 100$。

【解题思路】 使用单调栈求下一个更大元素。本题与 3.4.2 节类似，仅将求当前元素的下一个更大元素改为求当前元素与其下一个更大元素之间的距离。同样使用单调递减栈(小顶栈)求解，对应的算法如下。

C++：

```
class Solution {
public:
    vector<int> dailyA(vector<int>& a) {
        int n=a.size();
        vector<int> ans(n,0);
        stack<int> st;                           //小顶栈
        for(int i=0;i<n;i++) {
            while(!st.empty() && a[st.top()]<a[i]) {
                int prei=st.top();
                ans[prei]=i-prei;
                st.pop();
            }
```

```
                st.push(i);
            }
            return ans;
        }
    };
```

提交运行：

结果：通过；时间：144ms；空间：100.5MB

Python：

```
class Solution:
    def dailyA(self, a: List[int]) -> List[int]:
        n=len(a)
        ans=[0] * n
        st=[]
        for i in range(0,n):
            while st and a[st[-1]]<a[i]:
                prei=st[-1]
                ans[prei]=i-prei
                st.pop()
            st.append(i)
        return ans
```

提交运行：

结果：通过；时间：216ms；空间：23.1MB

3.4.4 LeetCode316——去除重复字母★★

【问题描述】 给定一个字符串 s，请去除字符串中重复的字母，使得每个字母只出现一次，并且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)。

例如，s="cbacdcbc"，答案为"acdb"。

【限制】 $1 \leqslant s.length \leqslant 10^4$，s 由小写英文字母组成。

【解题思路】 使用单调栈求下一个更小元素。所谓字典序最小，就是去除重复字母，并且使字典序小的字母尽可能往前，字典序大的字母尽可能往后，简单地说就是尽可能保证结果字符串中全部或者局部子串是按字典序递增的。使用单调递增栈(大顶栈)求解，先遍历字符串 s，用 cnt 数组记录每个字母出现的次数。再从前往后遍历，对于当前字母 c：

(1) 如果 c 在栈中，说明 c 已经找到了最佳位置，只需要将其计数减 1。

(2) 否则将栈顶所有字典序更大(st.top()>c，递减)且后面还有的字母(cnt[st.top()]≥1)的栈顶字母出栈(对于字典序更大但后面不出现的字母则不必弹出)。再将 c 进栈，同时将其计数减 1。

为了快速判断一个字母是否在栈中，设计一个 visited 数组，若 visited[c]=1，表示字母 c 在栈中；若 visited[c]=0，表示字母 c 不在栈中。

例如，s="cbacdcbc"，求出每个字母出现的次数为 cnt['a']=1，cnt['b']=2，cnt['c']=4，cnt['d']=1，栈 st=[]，用 c 遍历 s：

(1) c='c'，栈为空，'c'进栈(visited['c']=1)，st=['c']，其计数减 1，即 cnt['c']=3。

(2) c='b'，'b'不在栈中且栈顶字母满足'c'>'b'(递减)，出栈'c'(visited['c']=0)，st=[]，'b'进栈(visited['b']=1)，st=['b']，其计数减 1，即 cnt['b']=1。

扫一扫

源程序

（3）c='a','a'不在栈中且栈顶字母满足'b'>'a'（递减），出栈'b'（visited['b']=0），st=[]，'a'进栈（visited['a']=1），st=['a']，其计数减 1，即 cnt['a']=0。

（4）c='c','c'不在栈中且栈顶字母'a'<'c'（递增），将'c'进栈（visited['c']=1），st=['a','c']，其计数减 1，即 cnt['c']=2。

（5）c='d','d'不在栈中且栈顶字母'c'<'d'（递增），将'd'进栈（visited['d']=1），st=['a','c','d']，其计数减 1，即 cnt['d']=0。

（6）c='c','c'在栈中，其计数减 1，即 cnt['c']=1。

（7）c='b','b'不在栈中，尽管栈顶字母'd'>'b'（递减），但 cnt['d']=0，所以'd'不出栈，将'b'进栈（visited['b']=1），st=['a','c','d','b']，其计数减 1，即 cnt['b']=0。

（8）c='c','c'在栈中，其计数减 1，即 cnt['c']=0。

遍历完毕，最后将 st 栈中从栈底到栈顶的所有字母连接起来得到答案，这里为 ans="acdb"。

3.4.5 LeetCode84——柱状图中最大的矩形★★★

【问题描述】 给定 n 个非负整数的数组 a，表示柱状图中各个柱子的高度，每个柱子彼此相邻，且宽度为 1，求在该柱状图中能够勾勒出来的矩形的最大面积。

例如，对于如图 3.14 所示的柱状图，每个柱子的宽度为 1，6 个柱子的高度为[2,1,5,6,2,3]，其中最大矩形面积为 10 个单位。

【解题思路】 用单调递增栈找多个元素的共同右边界。先用穷举法求解，用 k 遍历 a，对于以 $a[k]$为高度作为矩形的高度，向左找到第一个小于 $a[k]$的高度 $a[i]$，称柱子 i 为左边界，向右找到第一个小于 $a[k]$的高度 $a[j]$，称柱子 j 为右边界，$a[k]$对应的最大矩形为 $a[i+1..j-1]$（不含柱子 i 和柱子 j），共包含 $j-i-1$ 个柱子，宽度 length=$j-i-1$，其面积为 $a[k]\times(j-i-1)$，通过比较将最大面积存放到 ans 中，a 遍历完毕返回 ans 即可。

例如，a=[1,4,3,6,4,2,5]，k=2 时对应的矩形及其面积如图 3.15 所示。

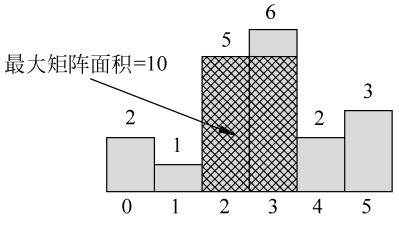


图 3.14 一个柱状图及其最大矩形

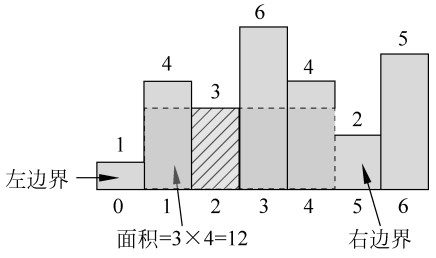


图 3.15 以 $a[2]$为高度的矩形

对应的穷举算法如下。

C++：

```
1  class Solution {
2  public:
3      int largestRectangleArea(vector<int> &a) {
4          int n=a.size();
5          int ans=0;
6          for(int k=0;k<n;k++) {
```

```
                int length=1;
                int i=k;
                while(--i>=0 && a[i]>=a[k]) length++;
                int j=k;
                while(++j<n && a[j]>=a[k]) length++;
                int area=length*a[k];
                ans=max(ans,area);
            }
            return ans;
        }
    };
```

提交运行：

```
结果:超时(98 个测试用例中通过了 86 个)
```

上述算法的时间复杂度为 $O(n^2)$，出现超时的情况。可以使用单调递增栈(大顶栈)提高性能，为了设置高度数组 a 的左、右边界，在 a 中前后添加一个 0(0 表示最小柱高度)。用 j 从 0 开始遍历 a，用栈 st 维护一个柱子高度递增序列：

(1) 若栈为空，则直接将 j 进栈。

(2) 若栈不为空且栈顶柱子 k 的高度 $a[k]$小于或等于 $a[j]$，则直接将 j 进栈。

(3) 若栈不为空且栈顶柱子 k 的高度 $a[k]$大于 $a[j]$，则柱子 k 找到了右边第一个小于其高度的柱子 j(这里大顶单调栈是为了找栈顶柱子的下一个高度更小的柱子)，也就是说柱子 k 的右边界是柱子 j。将柱子 k 出栈，新栈顶柱子 st.top()的高度肯定小于柱子 k 的高度，所以将柱子 st.top()作为柱子 k 的左边界，对应矩形的宽度 length$=j-$st.top()-1，其面积$=a[k]\times$length。然后对新栈顶柱子做同样的运算，直到不满足条件为止。再将 j 进栈。

例如，$a=[1,2,3,4,2]$，前后添加 0 后 $a=[0,1,2,3,4,2,0]$，如图 3.16 所示，用“$x[y]$”表示 $a[x]=y$，即柱子 x 的高度为 y。ans$=0$，求面积的部分过程如下。

依次将 0[0]、1[1]、2[2]、3[3]和 4[4]进栈，st$=${0[0]，1[1]，2[2]，3[3]，4[4]}。当遍历到 $a[5]=2$ 时($j=5$)：

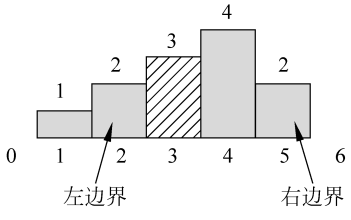


图 3.16 $a=[0,1,2,3,4,2,0]$

(1) 栈顶 4[4]的高度大于 $a[5]$，柱子 5 作为柱子 4 的右边界，出栈 4[4]，即 $k=4$，st$=${0[0]，1[1]，2[2]，3[3]}，新栈顶为 3[3]，将柱子 3 作为柱子 4 的左边界，求出 area$=a[k]\times(j-$st.top()$-1)=4\times1=4$⇨ans$=4$。

(2) 新栈顶 3[3]的高度大于 $a[5]$，柱子 5 作为柱子 3 的右边界(其中柱子 4 的高度一定大于柱子 3 的高度，见图 3.15)，出栈 3[3]，即 $k=3$，st$=${0[0]，1[1]，2[2]}，新栈顶为 2[2]，将柱子 2 作为柱子 3 的左边界，求出 area$=a[k]\times(j-$st.top()$-1)=3\times2=6$⇨ans$=6$。

(3) 新栈顶 2[2]的高度大于 $a[5]$不成立，将 $j=5$ 进栈。

从中看出通过单调栈可以一次求出多个柱子的右边界，同时又可以快速求出这些柱子的左边界。尽管在算法中使用了两重循环，实际上每个柱子最多进栈、出栈一次，所以算法的时间复杂度为 $O(n)$。对应的算法如下。

C++：

```
class Solution {
public:
    int largestRectangleArea(vector<int>& a) {
        a.insert(a.begin(),0);
        a.push_back(0);
        int n=a.size();
        stack<int> st;                          //大顶栈
        int ans=0;                              //存放最大矩形面积,初始为0
        for(int j=0;j<n;j++) {                  //遍历a
            while(!st.empty() && a[st.top()]>a[j]) {
                int k=st.top(); st.pop();       //退栈k
                int length=j-st.top()-1;
                int area=a[k] * length;
                ans=max(ans,area);
            }
            st.push(j);                         //j进栈
        }
        return ans;
    }
};
```

提交运行：

```
结果:通过;时间:136ms;空间:75.4MB
```

由于在 a 的前面插入 0 的时间为 $O(n)$，可以改为仅在 a 的后面插入 0，当遍历到 $a[j]$ 时，如果栈不为空且栈顶柱子 k 的高度 $a[k]$ 大于 $a[j]$，将柱子 k 出栈；如果栈为空，则置 length=j。对应的算法如下。

C++：

```
class Solution {
public:
    int largestRectangleArea(vector<int>& a) {
        a.push_back(0);
        int n=a.size();
        stack<int> st;                          //大顶栈
        int ans=0;                              //存放最大矩形面积,初始为0
        for(int j=0;j<n;j++) {                  //遍历a
            while(!st.empty() && a[st.top()]>a[j]) {
                int k=st.top(); st.pop();       //退栈k
                int length;
                if(st.empty()) length=j;        //栈为空时置length为j
                else length=j-st.top()-1;       //栈不为空
                int area=a[k] * length;         //求a[st.top()+1..j-1]的矩形面积
                ans=max(ans,area);              //维护ans为最大矩形面积
            }
            st.push(j);                         //j进栈
        }
        return ans;
    }
};
```

提交运行：

```
结果:通过;时间:120ms;空间:75.4MB
```

Python：

```
class Solution:
    def largestRectangleArea(self, a: List[int]) -> int:
        a.append(0)
        n,ans=len(a),0
        st=[]
        for j in range(0,n):
            while st and a[st[-1]]>a[j]:
                k=st[-1]; st.pop()      #退栈 tmp
                length=j-st[-1]-1 if st else j
                area=a[k] * length
                ans=max(ans,area)
            st.append(j)
        return ans
```

提交运行：

```
结果:通过;时间:296ms;空间:25.8MB
```

3.4.6 LeetCode42——接雨水★★★

【问题描述】 给定 n 个非负整数的数组 a，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。

例如，$a=[0,1,0,2,1,0,1,3,2,1,2,1]$，如图 3.17 所示，可以接 6 个单位的雨水，答案为 6。

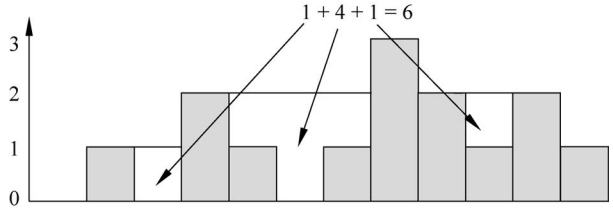


图 3.17 一个高度图

【限制】 $n=$height.length，$1\leqslant n\leqslant 2\times 10^4$，$0\leqslant$height$[i]\leqslant 10^5$。

【解题思路】 用单调递减栈找多个元素的共同右边界。先用穷举法求解，用 ans 存放答案(初始为 0)，用 k 从 1 到 $n-1$ 遍历 a(前后两个柱子不用考虑)，找到其左边最大高度 max_left 和右边最大高度 max_right，求出 max_left 和 max_right 中的最小值 minh，对于柱子 k 而言，上面接的雨水量为 minh$-a[k]$($a[k]<$mink)，对于每个柱子 k 累计这个值到 ans 中，最后返回 ans 即可。对应的算法如下。

C++：

```
class Solution {
public:
    int trap(vector<int>& a) {
        int n=a.size();
        int ans=0;
        for(int k=1;k<n-1;k++) {
            int max_left=0;          //求左边最大高度
            for(int i=k-1;i>=0;i--) {
                if(a[i]>max_left)max_left=a[i];
```

```
                }
                int max_right=0;                    //求右边最大高度
                for(int j=k+1;j<n;j++) {
                    if(a[j]> max_right) max_right=a[j];
                }
                int minh=min(max_left,max_right);
                if(minh>a[k]) ans+=(minh-a[k]);
            }
            return ans;
        }
    };
```

提交运行：

```
结果:超时(322 个测试用例中通过了 321 个)
```

上述过程是按列求接雨水量，算法的时间复杂度为 $O(n^2)$。另外也可以按层求接雨水量，例如，将图 3.17 中高度 0～1 看成第 1 层，高度 1～2 看成第 2 层，高度 2～3 看成第 3 层，求出第 1 层的接雨水量为 1+1=2，第 2 层的接雨水量为 3+1=4，第 3 层的接雨水量为 0，总的接雨水量为 2+4+0=6。这种按层求解的穷举算法的时间复杂度为 $O(m\times n)$，其中 m 为最大高度值。

可以进一步用单调栈提高性能，设置单调递减栈(小顶栈)st，ans 表示答案(初始为 0)，用 i 从 0 开始遍历 a：

(1) 若栈为空，则直接将 i 进栈。

(2) 若栈不为空且栈顶柱子 k 的高度 $a[k]$大于或等于 $a[i]$，说明柱子 i 会有积水，则直接将 i 进栈。

(3) 若栈不为空且栈顶柱子 k 的高度 $a[k]$小于 $a[i]$，说明之前的积水到这里停下，可以计算一下有多少积水。计算方式是出栈柱子 k，以新栈顶柱子 st.top()为左边界 l，柱子 i 为右边界 r，求出接雨水矩形(不含柱子 l 和柱子 r)的高度 h 为 $\min(a[r],a[l])-a[k]$，宽度为 $r-l-1$，则将接雨水量 $(r-l-1)\times h$ 累计到 ans 中。

在上述过程中每个柱子仅进栈和出栈一次，所以算法的时间复杂度为 $O(n)$。

例如，$a=[4,2,1,0,1,2,4]$，其高度图如图 3.18 所示，求其接雨水量的过程如下。

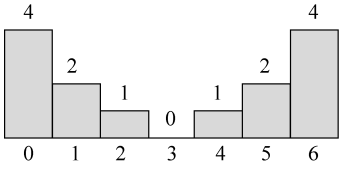


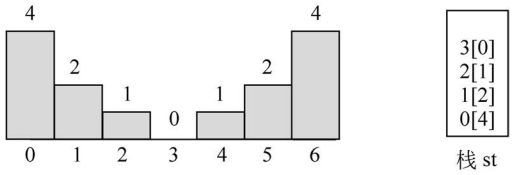
图 3.18　$a=[4,2,1,0,1,2,4]$

(1) 遍历到 $a[0]$，栈为空，将 $a[0]$进栈；遍历到 $a[1]$，$a[0]\geqslant a[1]$，将 $a[1]$进栈；遍历到 $a[2]$，$a[1]\geqslant a[2]$，将 $a[2]$进栈；遍历到 $a[3]$，$a[2]\geqslant a[3]$，将 $a[3]$进栈，如图 3.19(a)所示，st=[0[4],1[2],2[1],3[0]]。

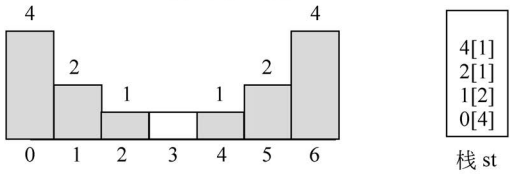
(2) 遍历到 $a[4]$，栈顶 $a[3]<a[4]$，出栈 $a[3]$，st=[0[4],1[2],2[1]]，求出以新栈顶 $a[2]$为左边界、以 $a[4]$作为右边界的接雨水量=1。新栈顶 $a[2]<a[4]$不成立，结束，再将 $a[4]$进栈，st=[0[4],1[2],2[1],4[1]]，如图 3.19(b)所示。

(3) 遍历到 $a[5]$，栈顶 $a[4]<a[5]$，出栈 $a[4]$，st=[0[4],1[2],2[1]]，新栈顶为左边界 $l=2$，右边界 $r=5$，高度 h 为 $\min(a[r],a[l])-a[4]=0$，对应的接雨水量=0。新栈顶 $a[2]<a[5]$，出栈 $a[2]$，st=[0[4],1[2]]，新栈顶为左边界 $l=1$，右边界 $r=5$，高度 h 为 $\min(a[r],a[l])-a[2]=1$，对应的接雨水量为 $(r-l-1)\times h=3\times 1=3$。再将 $a[5]$进栈，st=[0[4],1[2],5[2]]，如图 3.19(c)所示。

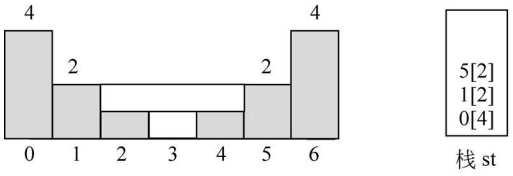
(4) 遍历到 $a[6]$，栈顶 $a[5]<a[6]$，出栈 $a[5]$，st=[0[4],1[2]]，新栈顶为左边界 $l=1$，右边界 $r=6$，高度 h 为 $\min(a[r],a[l])-a[5]=0$，对应的接雨水量=0。新栈顶 $a[1]<a[6]$，出栈 $a[1]$，st=[0[4]]，新栈顶为左边界 $l=0$，右边界 $r=6$，高度 h 为 $\min(a[r],a[l])-a[1]=2$，对应的接雨水量为 $(r-l-1)\times h=5\times 2=10$。再将 $a[6]$进栈，st=[0[4],6[4]]，如图 3.19(d)所示。



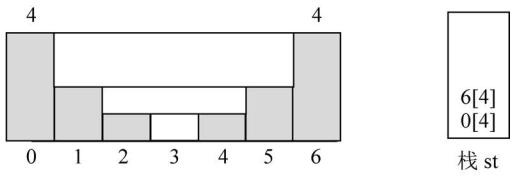
(a) $a[0]\sim a[3]$进栈



(b) $a[4]=1$，出栈 $a[3]$，求出其接雨水量=1



(c) $a[5]=2$，出栈 $a[4]$和 $a[2]$，求出其接雨水量=3



(d) $a[6]=4$，出栈 $a[5]$和 $a[1]$，求出其接雨水量=10

图 3.19 求 $a=[4,2,1,0,1,2,4]$接雨水量的过程

总的接雨水量 ans 为 1+3+10=14。

又如，$a=[1,0,2,1,0,1,3,2,1,2]$，求接雨水量的过程如图 3.20 所示。

(1) $a[2]=2$ 出现递增，触发计算出 $a[1]$的接雨水量=1。

(2) $a[5]=1$ 出现递增，触发计算出 $a[4]$的接雨水量=1。

(3) $a[6]=3$ 出现递增，触发计算出 $a[3..5]$的接雨水量=3。

(4) $a[9]=2$ 出现递增，触发计算出 $a[8]$的接雨水量=1。

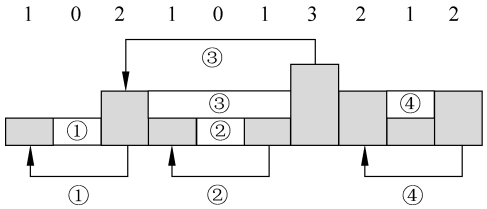


图 3.20 求 $a=[1,0,2,1,0,1,3,2,1,2]$接雨水量的过程

总的接雨水量为 1+1+3+1=6。对应的算法如下。

C++：

```
class Solution {
public:
    int trap(vector<int>& a) {
        int n=a.size();
        int ans=0;
        stack<int> st;          //小顶栈
        for(int i=0;i<n;i++) {
            while(!st.empty() && a[st.top()]<a[i]) {
                int k=st.top(); st.pop();
                if(!st.empty()) {
                    int l=st.top();
                    int r=i;
                    int h=min(a[r],a[l])-a[k];
                    ans+=(r-l-1)*h;
                }
            }
            st.push(i);
        }
        return ans;
    }
};
```

提交运行：

结果：通过；时间：16ms；空间：19.9MB

Python：

```
class Solution:
    def trap(self, a: List[int]) -> int:
        ans=0
        st=[]
        for i in range(0,len(a)):
            while st and a[st[-1]]<a[i]:
                k=st[-1]; st.pop()
                if len(st)>0:
                    l,r=st[-1],i
                    h=min(a[r],a[l])-a[k]
                    ans+=(r-l-1)*h
            st.append(i)
        return ans
```

提交运行：

结果：通过；时间：56 ms；空间：16.5MB

推荐练习题

1. LeetCode85——最大矩形★★★
2. LeetCode150——逆波兰表达式求值★★
3. LeetCode224——基本计算器★★★
4. LeetCode227——基本计算器Ⅱ★★
5. LeetCode402——移掉 k 位数字★★

6. LeetCode456——132 模式★★
7. LeetCode678——有效的括号字符串★★
8. LeetCode735——行星碰撞★★
9. LeetCode856——括号的分数★★
10. LeetCode921——使括号有效的最少添加★★
11. LeetCode1019——链表中的下一个更大结点★★
12. LeetCode1172——餐盘栈★★★
13. LeetCode1190——反转每对括号间的子串★★
14. LeetCode1209——删除字符串中所有的相邻重复项Ⅱ★★
15. LeetCode1472——设计浏览器历史记录★★
16. LeetCode1504——统计全 1 子矩形★★
17. LeetCode1598——文件夹操作日志搜集器★
18. LeetCode1653——使字符串平衡的最少删除次数★★
19. LeetCode1717——删除子字符串的最大得分★★
20. LeetCode2296——设计一个文本编辑器★★★
21. LeetCode2390——从字符串中移除星号★★
22. LeetCode2434——使用机器人打印字典序最小的字符串★★
23. LeetCode2345——寻找可见山的数量★★