

# 第一节 栈接口与实现

2016年8月31日 16:51

## 6.1.1 栈

线性序列，任何时候都只能访问栈中某一特定元素（其中某一端）  
该端称为top，另一端为bottom  
汉诺塔问题

## 6.1.2 实例

Stack() empty() T/F push(5) pop() size()n top()i  
First in, last out.

## 6.1.3 实现

❖ 栈既然属于序列的特例，故可直接基于向量或列表派生

```
❖ template <typename T> class Stack public Vector<T> { //由向量派生
public: //size(), empty()以及其它开放接口均可直接沿用
    void push(T const &e) { insert(size(), e); } //入栈
    → T pop() { return remove(size() - 1); } //出栈
    → T & top() { return (*this)[size() - 1]; } //取顶
}; //以向量首/末端为栈底/顶——颠倒过来呢？
```



## 第二节 栈应用：进制转换

2016年8月31日 17:33

### 6.2.1 应用

逆序输出


### 6.2.2 算法

进制转化短除法中的栈.

### 6.2.3 实现

**算法实现**

```
❖ void convert( Stack<char> &S, __int64 n, int base ) {  
    static char digit[] = //新进制下的数位符号, 可视base取值范围适当扩充  
    → { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F' };  
    while ( n > 0 ) { //由低到高, 逐一计算出新进制下的各数位  
        S.push( digit[n % base] ); //余数 ( 对应的数位 ) 入栈  
        n /= base; //n更新为其对base的除商  
    }  
}  
  
❖ main() {  
    Stack<char> S; convert(S, n, base); //用栈记录转换得到的各数位  
    while ( !S.empty() ) printf( "%c", S.pop() ); //逆序输出  
}
```



Data Structures & Algorithms (Fall 2015), Tsinghua University

## 第三节 栈应用：括号匹配

2016年8月31日 17:45

### 6.3.1 实例

**递归嵌套**

- stack permutation + parenthesis
- 具有自相似性的问题可递归描述，但分支位置和嵌套深度不固定

### 6.3.2 尝试

**尝试**

0) 平凡：无括号的表达式是匹配的  $\emptyset$  ✓

1) 减治？  $\cancel{\times}$   $\text{E}$  匹配 仅当  $( \text{E} )$  匹配 *only if*

2) 分治？  $\cancel{\times}$   $\text{E}$  和  $\text{F}$  均匹配，仅当  $\text{E}$  和  $\text{F}$  匹配

✧ 然而，根据以上性质，却不易直接应用已知的策略

✧ 究其根源在于，1) 和 2) 均为必要性，比如反例：

$( ( ) ( ) ( ) ) = ( ( ( ) ( ) ) ) \times$

$( ( ) ( ) ( ) ) = ( ( ) ) ( ( ) ) ( )$

图中，减治与分治的方法，均为必要性而非充分性，所以尝试失败。

### 6.3.3 构思

遇（则进栈，遇）则出栈。

### 6.3.4 实现

**实现**

```
bool paren(const char exp[], int lo, int hi) { //exp[lo, hi)
    Stack<char> S; //使用栈记录已发现但尚未匹配的左括号
    for (int i = lo; i < hi; i++) //逐一检查当前字符
        if ( '(' == exp[i] ) S.push( exp[i] ); //遇左括号：则进栈
        else if ( !S.empty() ) S.pop(); //遇右括号：若占非空，则弹出左括号
}
```

### 实现

```
❖ bool paren(const char exp[], int lo, int hi) { //exp[lo, hi)

    Stack<char> S; //使用栈记录已发现但尚未匹配的左括号

    for (int i = lo; i < hi; i++) //逐一检查当前字符

        if ( '(' == exp[i] ) S.push( exp[i] ); //遇左括号：则进栈

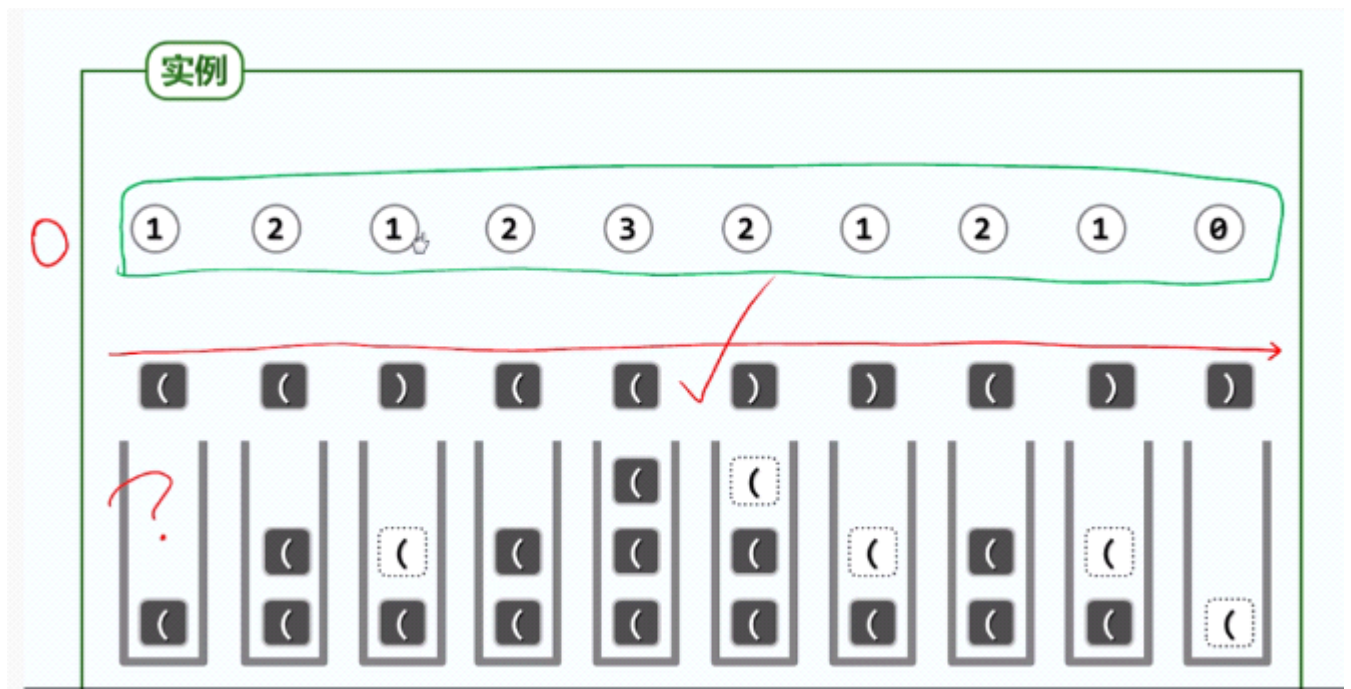
        else if ( !S.empty() ) S.pop(); //遇右括号：若占非空，则弹出左括号

        else return false; //否则（遇右括号时栈已空），必不匹配

    return S.empty(); //最终，栈空当且仅当匹配

}
```

#### 6.3.5 反思



能否使用计数器（也就是栈中的size）代替栈呢？

#### 6.3.6 拓展

计数器的操作并不能，反映出【与（、{的区别，所以并不能完整实现括号匹配的功能。

## 第四节 栈应用：栈混洗

2016年8月31日 18:25

### 6.4.1 混洗

S为中转栈，A为起始栈，B为目的栈

**栈混洗**

- ❖ 考查栈  $A = \langle a_1, a_2, \dots, a_n \rangle$ 、 $B = S = \emptyset$
- ❖ 只允许 将A的顶元素弹出并压入S，或  
将S的顶元素弹出并压入B
- ❖ 若经过一系列以上操作后，A中元素全部转入B中  
 $B = \langle a_{k1}, \dots, a_{kn} \rangle$   
则称之为A的一个栈混洗 (stack permutation)

//左端为栈顶  
//S.push(A.pop())  
//B.push(S.pop())  
//右端为栈顶

因此我们在记录

规模为n的栈混洗，所得栈的数目？

### 6.4.2 计数

**计数**

$SP(n) = ?$

$\sum_{k=1}^n SP(k-1) \times SP(n-k)$

可能的栈混洗的数目为：Catalan数  $\frac{(2n)!}{(n+1)!n!}$

### 6.4.3 甄别



**甄别**

❖ 输入序列  $\langle 1, 2, 3, \dots, n \rangle$  的任一排列  $[p_1, p_2, p_3, \dots, p_n]$  是否为栈混洗？

❖ 简单情况： $\langle 1, 2, 3 \rangle, n = 3$   
 栈混洗共  $6! / 4! / 3! = 5$  种  
 全排列共  $3! = 6$  种

❖  $[3, 1, 2]$

❖ 观察：任意三个元素能否按某相对次序出现于混洗中，与其它元素无关 // 故可推而广之...

❖ 对于任何  $1 \leq i < j < k \leq n$ ,  $[ \dots, \overset{3}{k}, \dots, \overset{1}{i}, \dots, \overset{2}{j}, \dots ]$  必非栈混洗

❖ 反过来，不存在“312”模式的序列，一定是栈混洗吗？

**禁形**

Diagram illustrating the stack permutation process for  $[3, 1, 2]$ . It shows input sequence A  $[1, 2, 3]$ , output sequence B  $[3, 2, 1]$ , and a stack S containing  $[3, 2, 1]$ . Arrows indicate the flow of elements from A to S and then to B.

// 少了一种...?  
 // 为什么是它?

#### 6.4.4 算法

“312”模式为充要条件

**甄别**

❖ 充要性：A permutation is a stack permutation iff (Knuth, 1968) it does NOT involve the permutation 312 // 习题[4-3]

❖ 如此，可得一个  $O(n^3)$  的甄别算法  $(i, j, k)$  // 进一步地...

❖  $[p_1, p_2, p_3, \dots, p_n]$  是  $\langle 1, 2, 3, \dots, n \rangle$  的栈混洗 当且仅当 对于任意  $i < j$  不含模式  $[ \dots, j+1, \dots, i, \dots, j, \dots ]$

❖ 如此，可得一个  $O(n^2)$  的甄别算法 // 再进一步地...

❖  $O(n)$  算法：直接借助栈 A、B 和 S，模拟混洗过程 // 为何可行？  
 每次 S.pop() 之前，检测 S 是否已空；或需弹出的元素在 S 中，却非顶元素

#### 6.4.5 括号

栈混洗其实与括号匹配一一对应

## 第五节 栈应用：中缀表达式求值

2016年8月31日 20:19

### 6.5.1 把玩



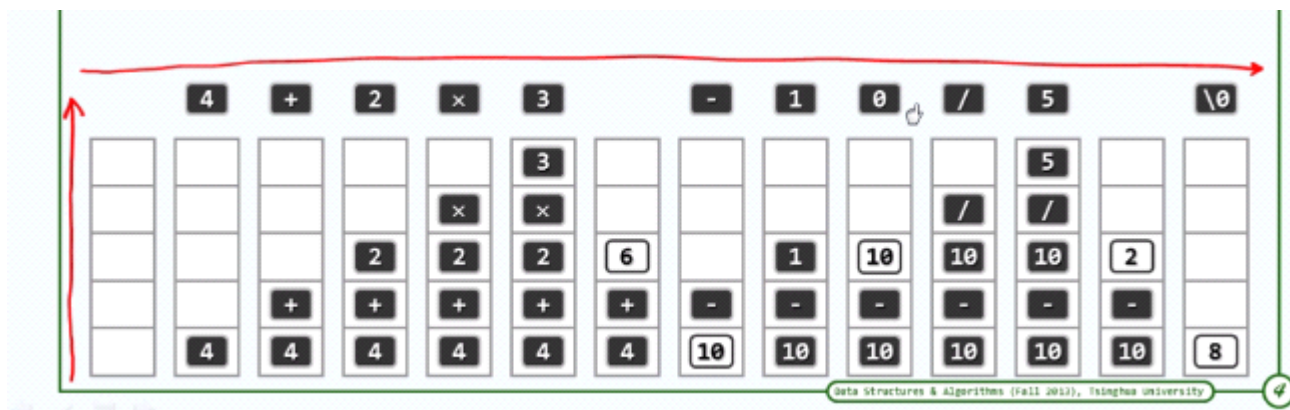
calc中用y代替^

### 6.5.2 构思

按照优先级，依次取出运算符，并进行运算代入。

对表达式进行扫描，对已扫描部分中局部优先级足够高的部分进行运算，其余部分存入栈中，继续扫描。

### 6.5.3 实例



两位数的处理

### 6.5.4 算法框架

## 实现：主算法

```

❖ float evaluate( char* S, char* & RPN ) { //中缀表达式求值
    Stack<float> opnd; Stack<char> optr; //运算数栈、运算符栈
    optr.push('\0'); //尾哨兵'\0'也作为头哨兵首先入栈
    while ( !optr.empty() ) { //逐个处理各字符，直至运算符栈空
        ✓ if ( isdigit( *S ) ) //若当前字符为操作数，则
            readNumber( S, opnd ); //读入 (可能多位的) 操作数
        ✓ else //若当前字符为运算符，则视其与栈顶运算符之间优先级的高低
            switch( orderBetween( optr.top(), *S ) ) { /* 分别处理 */ }
    } //while
    return opnd.pop(); //弹出并返回最后的计算结果
}

```

优先级二维表格

## 实现：优先级表

```

const char pri[N_OPTR][N_OPTR] = { //运算符优先等级 [栈顶][当前]
    //      |-----当前运算符-----|
    //      + - * / ^ ! ( ) \0
    /* -- + */ '>', '>', '<', '<', '<', '<', '<', '>', '>',
    /* | - */ '>', '>', '<', '<', '<', '<', '<', '>', '>',
    /* 栈 * */ '>', '>', '>', '>', '<', '<', '<', '>', '>',
    /* 顶 / */ '>', '>', '>', '>', '<', '<', '<', '>', '>',
    /* 运 ^ */ '>', '>', '>', '>', '>', '<', '<', '>', '>',
    /* 算 ! */ '>', '>', '>', '>', '>', '>', '>', '>', '>',
    /* 符 ( */ '<', '<', '<', '<', '<', '<', '<', '=', '=',
    /* | ) */ ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',
    /* -- \0 */ '<', '<', '<', '<', '<', '<', '<', ' ', '='
}

```

空字符？

## 6.5.5 算法细节



### 实现：不同优先级处理方法

```

❖ switch( orderBetween( optr.top(), *S ) ) {
    case '<': //栈顶运算符优先级更低
        optr.push( *S ); S++; break; //计算推迟，当前运算符进栈
    case '=': //优先级相等（当前运算符为右括号，或尾部哨兵'\0'）
        optr.pop(); S++; break; //脱括号并接收下一个字符
    case '>': { //栈顶运算符优先级更高，实施相应的计算，结果入栈
        char op = optr.pop(); //栈顶运算符出栈，执行对应的运算
        if ( '!' == op ) opnd.push( calcul( op, opnd.pop() ) ); //一元运算符
        else { float pOpnd2 = opnd.pop(), pOpnd1 = opnd.pop(); //二元运算符
            opnd.push( calcul( pOpnd1, op, pOpnd2 ) ); //实施计算，结果入栈
        } //为何不直接：opnd.push( calcul( opnd.pop(), op, opnd.pop() ) )?
        break;
    } //case '>'
}

```

从运算符优先级二维表格中发现，仅有（）以及‘\0\0’对应为=号的处理，由于计算的逐步进行，其实括号的存在可以忽略了。

注意某些运算符的不可交换性

#### 6.5.6 实例

pri['-'][')'] = '>'

( 1 + 2 ^ 3 ! - 4 ) \* ( 5 ! - ( 6 - ( 7 - ( 8 9 - 0 ! ) ) ) ) \$

\$ \* ( - ( - ( - ( -

61 120 6 7 89 1

## 第六节 栈应用：逆波兰表达式

2016年8月31日 21:27

### 6.6.1 简化

逆波兰表达式是一种对后缀表达式简化的逻辑谓词表达式

### 6.6.2 体验

奇怪又高效的RPN表达式

### 6.6.3 手工

**infix到postfix：手工转换**

❖ 例如： $(0! + 1)^{(2 * 3! + 4 - 5)}$

假设：事先未就运算符之间的优先级关系做过任何约定

- 1) 用括号显式地表示优先级  
 $\{ ([0!] + 1) ^ ([ (2 * [3!]) + 4 ] - 5) \}$
- 2) 将运算符移到对应的右括号后  
 $\{ ([0] ] ! 1) + ([ (2 [3] ] ! ) * 4 ] + 5 ) - \} ^$
- 3) 抹去所有括号  
 $0 ] [ ! 1 ] + [ 2 ] [ 3 ] [ ! ] * [ 4 ] + [ 5 ] - [ ] ^$
- 4) 稍事整理，即得  
 $0 ! 1 + 2 3 ! * 4 + 5 - ^$

RPN

运算符先后次序可能发生变化，运算数次序不会发生变化。

### 6.6.4 算法

**infix到postfix：转换算法**

```
❖ float evaluate( char* S, char* & RPN ) { //RPN转换
/* ..... */
while ( !optr.empty() ) { //逐个处理各字符，直至运算符栈空
    if ( isdigit(*S) ) //若当前字符为操作数，则直接将其
    { readNumber( S, opnd ); append( RPN, opnd.top() ); } //接入RPN
    else //若当前字符为运算符
    switch( orderBetween( optr.top(), *S ) ) {
        /* ..... */
        case '>' { //且可立即执行，则在执行相应计算的同时将其
            char op = optr.pop(); append( RPN, op ); //接入RPN
```

```
case '>': { //且可立即执行, 则在执行相应计算的同时将其  
    char op = optr.pop(); append(RPN op ); //接入RPN  
    /* ..... */  
} //case '>'
```

## 第七节 栈应用：队列接口与实现

2016年8月31日 21:53

### 6.7.1 接口

#### 操作与接口

❖ **队列 (queue)** 也是**受限**的序列

只能在**队尾**插入 (查询) : enqueue() + rear()

只能在**队头**删除 (查询) : dequeue() + front()

### 6.7.2 实例

操作实例								
操作	输出	队列 (右侧为队头)	操作	输出	队列 (右侧为队头)			
⇒ Queue()			enqueue(11)		11	3	7	3
empty()	true		⇒ size()	4	11	3	7	3
✓ enqueue(5)		5	⇒ enqueue(6)		6	11	3	7
✓ enqueue(3)		3 5	empty()	false	6	11	3	7
⇒ dequeue()	5	3	enqueue(7)		7	6	11	3
⇒ enqueue(7)		7 3	dequeue()	3	7	6	11	3
⇒ enqueue(3)		3 7 3	dequeue()	7	7	6	11	3
✓ front()	3	3 7 3	front()	3	7	6	11	3
⇒ empty()	false	3 7 3	size()	4	7	6	11	3

### 6.7.3 实现

#### 模板类

❖ 队列既然属于序列的特例，故亦可直接基于**向量或列表**派生

❖ `template <typename T> class Queue: public List<T> { //由列表`  
`public: //size()与empty()直接沿用`

⇒ `void enqueue(T const & e) { insertAsLast(e); } //入队`



## 模板类

❖ 队列既然属于序列的特例，故亦可直接基于向量或列表派生

❖ `template <typename T> class Queue: public List<T> { //由列表派生  
public: //size()与empty()直接沿用`

`⇒ void enqueue(T const & e) { insertAsLast(e); } //入队`

`T dequeue() { return remove(first()); } //出队`

`T & front() { return first()->data; } //队首`

`}; //以列表首/末端为队列头/尾——颠倒过来呢？`

队列的实现选择继承自列表  
栈的实现则继承自向量