

# 数据结构

## 实验报告（二）

### 栈和队列

学号：3020205015

姓名：石云天

班级：智能机器平台 2 班

日期：2020.10.25

## 目 录

一、实验内容描述 .....	3
二、实验步骤 .....	3
三、程序设计 .....	4
(一) 抽象数据类型 ADT .....	4
(二) 算法简述 .....	5
(三) 程序代码 .....	7
四、调试分析 .....	8
(一) 调试过程和主要错误 .....	8
(二) 时间复杂度 .....	9
五、程序测试 .....	9
六、实验总结 .....	11
附录 1: 程序源代码: 链式栈的实现 .....	12
附录 2: 程序源代码: 顺序循环队列的实现 .....	14
附录 3: 程序源代码: 中缀表达式求值算法 .....	16
附录 4: 程序源代码: 中缀表达式转后缀表达式算法 .....	22
附录 5: 程序源代码: 打印杨辉三角算法 .....	26

## 一、实验内容描述

本次实验主题是栈和队列，包含四个内容：链式栈的实现、顺序循环列表的实现、栈的应用（表达式求值）以及队列的应用（打印杨辉三角）。下面是详细的实验内容：

### （一）实现无表头节点的链栈 `MyStack`

- 栈初始化
- 销毁栈
- 入栈
- 出栈
- 返回栈顶元素
- 输出栈内元素

### （二）实现顺序循环队列 `MyQueue`

- 队列初始化
- 销毁队列
- 入队
- 出队
- 返回队头元素
- 输出队列内元素

（三）基于 `MyStack` 实现中缀表达式转后缀表达式和表达式求值，支持小括号、加减乘除四则运算,一位正整数(0-9)运算,不涉及多位数、负数及小数

（四）基于 `MyQueue`，实现输出杨辉三角

## 二、实验步骤

（1）根据上课所讲，回顾栈和队列的基本概念，充分认识到栈和队列是有限制的线性表，根据实验要求栈和队列分别选取链式存储结构和顺序存储结构，并总结所有操作，构造抽象数据类型。

（2）仔细阅读实验要求，考虑栈和队列各个操作的主要算法。

（3）利用 CodeBlocks 编译器，配置环境，基于 C++ 语言将算法用程序实现。

- (4) 编译运行程序，使用样例进行程序测试，观察所编程序是否实现要求的功能。
- (5) 考察算法的时间复杂度和空间复杂度，评价算法的优劣，进一步优化程序。
- (6) 撰写实验报告，进行实验总结与反思。

## 三、程序设计

### (一) 抽象数据类型 ADT

不带头节点的链式栈的抽象数据类型：

**ADT LiStack{**

**数据对象：**  $D=\{a_i | a_i \in \text{char}, i = 1, 2, 3 \dots\}$

**数据关系：**  $R=\{\langle a_{i-1}, a_i \rangle | a_i, a_{i-1} \in D\}$

**基本操作：**

**void InitStack(LiStack &S)**

操作结果：链栈的初始化函数

**void DeleteStack(LiStack &S)**

操作结果：链栈的销毁

**int StackEmpty(LiStack S)**

操作结果：判断栈空

**void Push(LiStack &S, char e)**

操作结果：入栈

**void Pop(LiStack &S, char &e)**

操作结果：出栈

**char GetTop\_Stack(LiStack S)**

操作结果：返回栈顶元素

**void PrintStack(LiStack S)**

操作结果：输出栈内元素(LIFO)

**}ADT LiStack;**

顺序循环队列的抽象数据类型：

**ADT SqQueue{**

**数据对象：**  $D=\{a_i | a_i \in \text{int}, i = 1, 2, 3 \dots\}$

**数据关系：**  $R=\{\langle a_{i-1}, a_i \rangle | a_i, a_{i-1} \in D\}$

**基本操作：**

```
int InitQueue(SqQueue &Q)
```

操作结果：队列的初始化

```
void DeleteQueue(SqQueue &Q)
```

操作结果：队列的销毁

```
int EnQueue(SqQueue &Q,int e)
```

操作结果：入队

```
int DeQueue(SqQueue &Q,int &e)
```

操作结果：出队

```
int GetTop_Queue(SqQueue &Q,int &e)
```

操作结果：返回队头元素

```
int PrintQueue(SqQueue Q)
```

操作结果：输出队列内元素(FIFO)

```
}ADT SqQueue;
```

## (二) 算法简述

因为链栈与顺序循环队列的各项基本操作的实现起来较为容易，故此处不再赘述。下面分别对中缀表达式求值、中缀表达式转换为后缀表达式以及打印杨辉三角的算法思想进行介绍。

### 一、中缀表达式求值

对于一个中缀表达式求值，其过程可分为以下几个步骤：

#### (1) 建立数据结构

本算法总体依靠链栈实现，其中需要构建一个字符栈以存储表达式中的运算符，再构建一个数字栈（int 类型）以存储表达式中的操作数，因此需要分别实现字符栈与数字栈两种数据结构。

#### (2) 创建函数比较运算符优先级

在中缀表达式中，存在 ‘+’、‘-’、‘\*’、‘/’、‘(’、‘)’、‘#’ 七种运算符，对于以上运算符，需要根据其优先级高低以确定其运算顺序，若待入栈运算符优先级高于栈顶运算符，则将其压入运算符栈中，反之则从操作数栈中取出两个操作数与栈顶运算符结合进行计算，具体优先级顺序见下表 1。表 1 的结果中有四种状态，分别是优先级高（待入栈运算符比栈顶运算符）、优先级低、无法比较（不存在）以及左右括号相互抵消。

表 1 栈内和栈外运算符优先级比较表

运算符	+	-	*	/	(	)	#
+	优先级低	优先级低	优先级高	优先级高	优先级高	优先级低	优先级低
-	优先级低	优先级低	优先级高	优先级高	优先级高	优先级低	优先级低
*	优先级低	优先级低	优先级低	优先级低	优先级高	优先级低	优先级低
/	优先级低	优先级低	优先级低	优先级低	优先级高	优先级低	优先级低
(	优先级高	优先级高	优先级高	优先级高	优先级高	相互抵消	优先级低
)	不存在	不存在	不存在	不存在	不存在	不存在	不存在
#	不存在	不存在	不存在	不存在	不存在	不存在	不存在

(3) 在.exe 文件中以 char 型数组的形式输入所要求的中缀表达式。

(4) 从左到右逐一扫扫各个字符，按照图 1 所示的算法逻辑借助操作数栈和运算符栈逐步进行运算，在读取到 char 型数组中的结束标志 ‘\0’ 后，跳出循环结束运算。

(5) 运算结束后操作数栈仅剩一个元素，取出该元素即为最终运算结果。

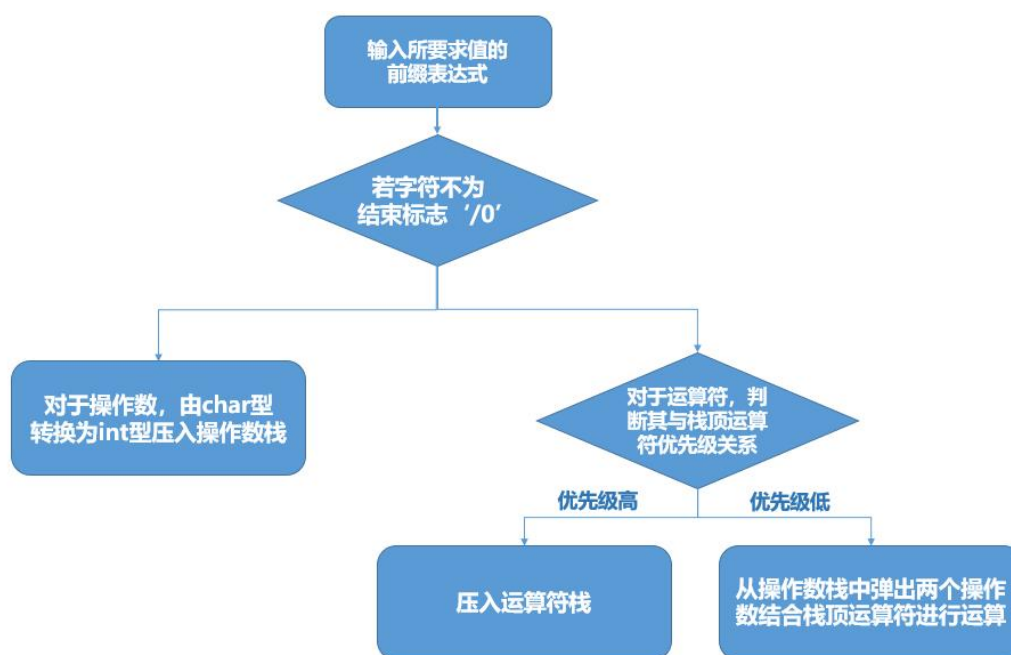


图 1 算法逻辑图示

## 二、中缀表达式转化为后缀表达式

该运算在算法上与中缀表达式求值的思想完全一致，实现起来索要使用的数据结构与操作步骤则更加简单方便。对于数据结构方面，由于该运算过程中不涉及具体值的计算，因此只需要使用运算符栈而不必使用操作数栈。对于操作步骤方面，整体上与中缀表达式求值运算的步骤相似，其中部分操作可以适当加以简化。在扫描到运算符时，若其优先级高于栈顶运算符，则直接入栈，反之则使栈顶元素出栈并打印。在扫描到操作数时，则无需加以操作，直接将

其打印便可。

### 三、打印杨辉三角

（1）整体上利用双层 while 循环语句，第一层循环语句表示需要打印的行数，嵌套的第二层循环语句表示每行要打印的元素个数。在每次循环的开始和结束过程中，要单独打印“1”，并将其入队。

（2）在循环体的内部，需要将队列中前两个元素出队，求和后打印，再将该和入队两次，因此每一行中除了首位的“1”之外都要被叠加两次。完成上述双层循环后，即可得到打印好的杨辉三角。

### （三）程序代码

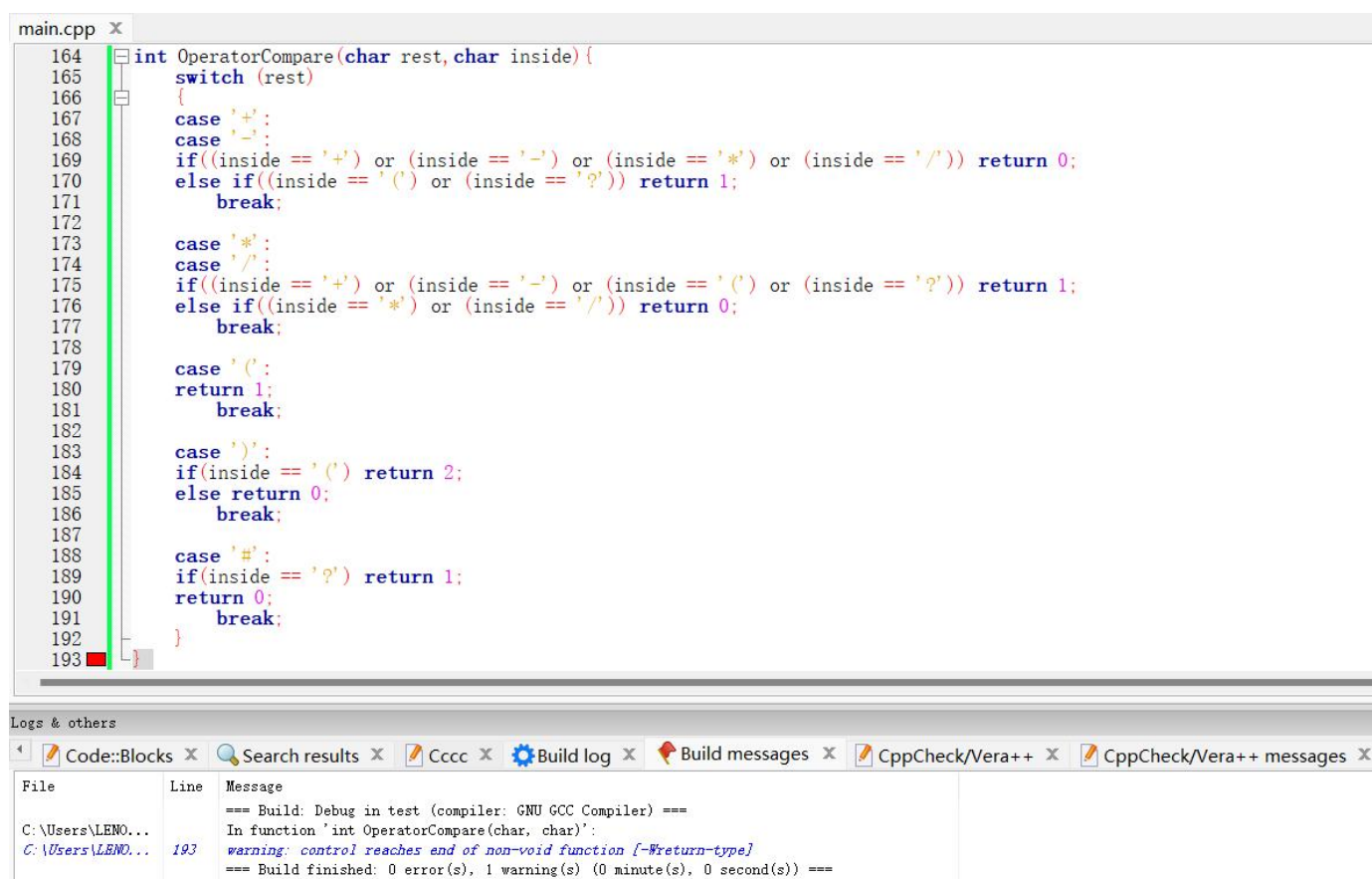
为保证实验报告的清晰和可读性，将源代码以**附录形式**附于文末。

## 四、调试分析

### （一）调试过程和主要错误

在程序编写完成后，使用 CodeBlocks 编译并运行。发现初代程序主要存在以下几方面问题：

（1）程序提示 warning: control reaches end of non-void function（见下图 2），该警告出现的原因是在 switch 选择语句中没有考虑到全部情况，存在情况没有返回值。这也导致对于有些样例测试结果不正确，对于这种情况，重新理清思路后，改为使用 if-else 语句代替 switch 语句，将所有情况纳入考虑范围内，修改后 warning 消失，同时程序测试结果正确。



The screenshot displays the CodeBlocks IDE interface. The top pane shows the source code for `main.cpp`, which defines a function `int OperatorCompare(char rest, char inside)`. The function uses a `switch` statement to handle different operators. The bottom pane shows the 'Logs & others' window with the following message:

```
==== Build: Debug in test (compiler: GNU GCC Compiler) ====
In function 'int OperatorCompare(char, char)':
C:\Users\LENO... 193 warning: control reaches end of non-void function [-Wreturn-type]
==== Build finished: 0 error(s), 1 warning(s) (0 minute(s), 0 second(s)) ====
```

图 2 程序 warning

（2）对于中缀表达式求值算法，出现了数据结构选择上的错误。在初代程序中选择使用两个字符栈分别存储运算符和操作数，最初在少量样例测试中并未发现问题，后使用较为复杂的样例进行测试，发现结果出现错误，经过思考，找到原因：在运算过程中可能会产生十位数甚至位数更多的数字，此时 `char` 型只能转换为个位数（0-9）的问题（不连续性）便会暴露出来，故需要使用新的



int 型数字栈以存储操作数，在预先编写的字符型链栈的基础上加以修改便可得到所需的 int 型数字栈（其结构见下图 3），对应修改后续程序便得到最终版本的中缀表达式求值算法程序，经过大量复杂测试样例测试后，均得到正确结果。

```
typedef struct Linknode_int{
    int data;
    struct Linknode_int *next;
}Linknode_int,*LiStack_int;
```

图 3 int 型数字栈结构

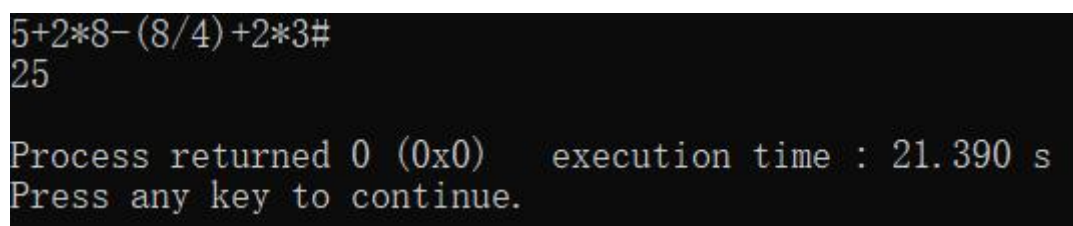
（3）对于预设的优先级比较函数 OperatorCompare，其每次被调用都需要两个运算符，即栈顶运算符与待入栈运算符，当扫描到中缀表达式中第一个运算符时，运算符栈为空，无法调用 OperatorCompare 函数。此时应在中缀表达式扫描前，预先压入一个字符 ‘?’ 以保证栈内始终非空，OperatorCompare 函数可以正常运行，随后在函数中为 ‘?’ 设置相应的优先级，保证函数运算的准确性。

## （二）时间复杂度

本次实验中的三个算法中，中缀表达式求值与中缀表达式转化为后缀表达式此二算法，需要对字符串进行一遍扫描，后压入操作数栈与运算符栈进行相应操作，时间复杂度为  $O(n)$ ，而打印杨辉三角算法需要两层 while 循环嵌套进行实现，时间复杂度为  $O(n^2)$ 。

## 五、程序测试

在完成全部程序编写后，输入测试样例进行测试，所得结果均满足要求。中缀表达式求值算法测试结果见图 4，中缀表达式转化为后缀表达式算法测试结果见图 5，打印杨辉三角算法测试结果见图 6。



```
5+2*8-(8/4)+2*3#
25

Process returned 0 (0x0)   execution time : 21.390 s
Press any key to continue.
```

```

8+(3+7/1-2*4)#
10

Process returned 0 (0x0)    execution time : 31.838 s
Press any key to continue.

```

图 4 中缀表达式求值算法测试结果

```

6+7-8+9#
6 7 + 8 - 9 + #

Process returned 0 (0x0)    execution time : 29.795 s
Press any key to continue.

```

```

3/2+5*4+(7-8)
3 2 / 5 4 * + 7 8 - #

Process returned 0 (0x0)    execution time : 33.234 s
Press any key to continue.

```

图 5 中缀表达式转化为后缀表达式算法测试结果

```

5
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1

Process returned 0 (0x0)    execution time : 1.549 s
Press any key to continue.

```

```

8
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1

Process returned 0 (0x0)    execution time : 2.137 s
Press any key to continue.

```

图 6 打印杨辉三角算法测试结果

## 六、实验总结

通过这次试验，我发现我对栈与队列这一部分的理解不够深入全面，需要不断巩固学习，加深理解。栈的数据结构与线性表基本相同，但需要关注两点：首先是 next 指针方向是相反的，其次栈顶指针永远指向数据元素的上一个节点。经过思考得出原因：普通链表的实现是带有头节点的，而栈的链式结构在实现时是不带头结点的，故会产生以上差异。在编程过程中需要完成某些特定目标时，我不能很快的想出其对应的操作，需要课下不断练习以熟能生巧，还可以多查阅一些资料以开阔自己的思路。在本次实验中，我编写并实现了链栈、顺序循环队列的各项基本操作，同时利用以上数据结构实现了中缀表达式求值算法、中缀表达式转化为后缀表达式算法、打印杨辉三角算法，在此过程中不断调试，寻找问题，并不断简化代码，提升函数执行速度。除此之外，在实验过程中，我对选择语句使用规范也有了更深的理解，对数据类型及其转换规则的掌握更加深入，能够不断改进，优化函数性能，实现预期目标与功能所需。

## 附录 1：程序源代码：链式栈的实现

```
typedef struct Linknode{
    char data;
    struct Linknode *next;
}Linknode,*LiStack;

//链栈的初始化函数
void InitStack(LiStack &S){
    S = new Linknode;
    S->next = NULL;
}

//链栈的销毁
void DeleteStack(LiStack &S){
    Linknode * p = S;
    do{
        S = S->next;
        delete(p);
        p = S;
    }while(p);
}

//判断栈空 栈空返回 0 栈非空返回 1
int StackEmpty(LiStack S){
    if(S->next == NULL) return 0;
    else return 1;
}

//入栈
void Push(LiStack &S,char x){
    Linknode * p = new Linknode;
    S->data = x;
    p->next = S;
    S = p;
}

//出栈
void Pop(LiStack &S,char &x){
    if(!StackEmpty(S)) cout<< "The stack is EMPTY!" <<endl;//判断是否为空
    栈
    else{
        Linknode * p = S;
        x = p->next->data;
    }
}
```

```

        S = S->next;
        delete(p);
    }
}

```

```

//返回栈顶元素
char GetTop_Stack(LiStack S){
    if(!StackEmpty(S)){
        cout<< "The stack is EMPTY!" <<endl;
        return '$';
    }//判断是否为空栈
    else{
        return S->next->data;
    }
}

```

```

//输出栈内元素(LIFO)
void PrintStack(LiStack S){
    if(!StackEmpty(S)){
        cout<< "The stack is EMPTY!" <<endl;
    }//判断是否为空栈
    else{
        Linknode * p = S->next;
        do{
            cout<< p->data << ' ';
            p = p->next;
        }while(p);
    }
}

```

## 附录 2：程序源代码：顺序循环队列的实现

```
typedef struct{
    int* data;
    int front;
    int rear;
}SqQueue;

int MAXSIZE=100;//全局变量设置最大长度

//队列的初始化
int InitQueue(SqQueue &Q){
    Q.data = new int[MAXSIZE-1];
    if(!Q.data) return -1;//内存分配失败
    else{
        Q.front = 0;
        Q.rear = 0;
        return 1;
    }
}

//队列的销毁
void DeleteQueue(SqQueue &Q){
    if(!Q.data){
        cout<< "There is no need to delete because it is EMPTY" <<endl;
    }//判断队列是否为空
    else{
        delete(Q.data);
    }
}

//入队
int EnQueue(SqQueue &Q,int x){
    if(((Q.rear+1)%MAXSIZE) == Q.front){
        cout<< "The queue is FULL" <<endl;
        return -1;
    }//判断队列是否已满
    else{
        Q.data[Q.rear] = x;
        Q.rear = (Q.rear+1) % MAXSIZE;
        return 1;
    }
}
```

```

//出队
int DeQueue(SqQueue &Q,int &x){
    if(Q.front == Q.rear){
        cout<< "The queue is EMPTY" <<endl;
        return -1;
    }//判断队列是否为空
    else{
        x = Q.data[Q.front];
        Q.front = (Q.front+1) % MAXSIZE;
        return 1;
    }
}

```

```

//返回队头元素
int GetTop_Queue(SqQueue &Q,int &x){
    if(Q.rear == Q.front){
        cout<< "The queue is EMPTY" <<endl;
        return -1;
    }//判断队列是否为空
    else{
        x = Q.data[Q.front];
        return 1;
    }
}

```

```

//输出队列内元素(FIFO)
int PrintQueue(SqQueue Q){
    if(Q.front == Q.rear){
        cout<< "The queue is EMPTY" <<endl;
        return -1;
    }//判断队列是否为空
    else if(Q.front < Q.rear){
        for(int i=Q.front;i<Q.rear;i++){
            cout<< Q.data[i] << ' ' ;
        }
        return 1;
    }
    else{
        for(int i=Q.front;i<Q.rear+MAXSIZE;i++){
            cout<< Q.data[i] << ' ' ;
        }
        return 1;
    }
}

```

### 附录 3：程序源代码：中缀表达式求值算法

```
#include <iostream>

using namespace std;

typedef struct Linknode{
    char data;
    struct Linknode *next;
}Linknode,*LiStack;

//链栈的初始化函数
void InitStack(LiStack &S){
    S = new Linknode;
    S->next = NULL;
}

//链栈的销毁
void DeleteStack(LiStack &S){
    Linknode * p = S;
    do{
        S = S->next;
        delete(p);
        p = S;
    }while(p);
}

//判断栈空 栈空返回 0 栈非空返回 1
int StackEmpty(LiStack S){
    if(S->next == NULL) return 0;
    else return 1;
}

//入栈
void Push(LiStack &S,char x){
    Linknode * p = new Linknode;
    S->data = x;
    p->next = S;
    S = p;
}

//出栈
void Pop(LiStack &S,char &x){
    if(!StackEmpty(S)) cout<< "The stack is EMPTY!" <<endl;//判断是否为空
```



栈

```
    else{
        Linknode * p = S;
        x = p->next->data;
        S = S->next;
        delete(p);
    }
}
```

//返回栈顶元素

```
char GetTop_Stack(LiStack S){
    if(!StackEmpty(S)){
        cout<< "The stack is EMPTY!" <<endl;
        return '$';
    }//判断是否为空栈
    else{
        return S->next->data;
    }
}
```

//输出栈内元素(LIFO)

```
void PrintStack(LiStack S){
    if(!StackEmpty(S)){
        cout<< "The stack is EMPTY!" <<endl;
    }//判断是否为空栈
    else{
        Linknode * p = S->next;
        do{
            cout<< p->data << ' ';
            p = p->next;
        }while(p);
    }
}
```

```
typedef struct Linknode_int{
    int data;
    struct Linknode_int *next;
}Linknode_int,*LiStack_int;
```

//链栈的初始化函数

```
void InitStack_int(LiStack_int &S){
    S = new Linknode_int;
    S->next = NULL;
}
```

//链栈的销毁

```
void DeleteStack_int(LiStack_int &S){
    Linknode_int *p = S;
    do{
        S = S->next;
        delete(p);
        p = S;
    }while(p);
}
```

//判断栈空 栈空返回 0 栈非空返回 1

```
int StackEmpty_int(LiStack_int S){
    if(S->next == NULL) return 0;
    else return 1;
}
```

//入栈

```
void Push_int(LiStack_int &S,int x){
    Linknode_int *p = new Linknode_int;
    S->data = x;
    p->next = S;
    S = p;
}
```

//出栈

```
void Pop_int(LiStack_int &S,int &x){
    if(!StackEmpty_int(S)) cout<< "The stack is EMPTY!" <<endl;//判断是
否为空栈
    else{
        Linknode_int * p = S;
        x = p->next->data;
        S = S->next;
        delete(p);
    }
}
```

//返回栈顶元素

```
int GetTop_Stack_int(LiStack_int S){
    if(!StackEmpty_int(S)){
        cout<< "The stack is EMPTY!" <<endl;
        return '$';
    }//判断是否为空栈
    else{
```

```

        return S->next->data;
    }
}

```

//输出栈内元素(LIFO)

```

void PrintStack_int(LiStack_int S){
    if(!StackEmpty_int(S)){
        cout<< "The stack is EMPTY!" <<endl;
    }//判断是否为空栈
    else{
        Linknode_int * p = S->next;
        do{
            cout<< p->data << ' ';
            p = p->next;
        }while(p);
    }
}

```

//编写函数比较待入栈运算符与栈顶运算符的优先级，优先级低不入栈用0表示，优先级高入栈用1表示，左右括号相互抵消用2表示

```

int OperatorCompare(char rest,char inside){
    if((rest == '+' ) or (rest == '-')){
        if((inside == '+' ) or (inside == '-') or (inside == '*') or (inside
== '/')) return 0;
        else if((inside == '(') or (inside == '?')) return 1;
        else return 0;
    }
    else if((rest == '*' ) or (rest == '/')){
        if((inside == '+' ) or (inside == '-') or (inside == '(') or (inside
== '?')) return 1;
        else if((inside == '*' ) or (inside == '/')) return 0;
        else return 0;
    }
    else if(rest == '('){
        return 1;
    }
    else if(rest == ')'){
        if (inside=='(')return 2;
        else return 0;
    }
    else if(rest == '#'){
        if (inside=='?')return 1;
        else return 0;
    }
}

```

```

        else return 0;
    }

//设置全局变量，维护符号栈和操作数栈
LiStack ope;
LiStack_int num;

int main()
{
    InitStack_int(num);
    InitStack(ope);
    Push(ope, '?'); //为保证 OperatorCompare 函数初始状态有两个操作数，预先入
    栈一个元素‘?’以维护运算符栈的正常工作
    char expression[100]; //输入的中缀表达式上限字符数为 100
    cin >> expression;
    int i = 0;
    while(expression[i] != '\0') { //字符不为结束标志‘\0’时执行循环
        if((expression[i] == '+' ) or (expression[i] == '-') or
(expression[i] == '*' ) or (expression[i] == '/') or (expression[i] ==
'(') or (expression[i] == ')') or (expression[i] == '#')){
            //判断待入栈运算符与栈顶运算符的优先级
            char top; //栈顶运算符
            int level; //优先级
            do{
                top = GetTop_Stack(ope);
                level = OperatorCompare(expression[i], top);
                //利用预设的 OperatorCompare 函数计算优先级
                if(level == 1){ //运算符入栈
                    Push(ope, expression[i]);
                    break;
                }
            }
            else if(level == 0){
                //运算符无法入栈，弹出操作数栈中两个操作数进行相应运算
                int a, b;
                Pop_int(num, b);
                Pop_int(num, a);
                int next = 0; //初步运算结果
                char next_char; //初步运算所用运算符
                Pop(ope, next_char);
                switch(next_char)
                {
                    case '+':
                        next = b + a;
                        break;

```

```

        case '-':
            next=a-b;
            break;
        case '*':
            next=b*a;
            break;
        case '/':
            next=a/b;
            break;
    }
    Push_int(num,next);
    //将初步运算结果存入操作数栈供后续运算使用
    }
    else if(level == 2){
        char brackets;
        Pop(ope,brackets);
    }
    }while(level==0);
    }
    else{//对于操作数，将其从 char 型转换为 int 型后压入操作数栈
        int p=expression[i]-'0';
        Push_int(num,p);
    }
    i++;
}
cout<<GetTop_Stack_int(num)<<endl;
//最终操作数栈中剩余的操作数即为运算结果
}

```

## 附录 4：程序源代码：中缀表达式转后缀表达式算法

```
#include <iostream>

using namespace std;

typedef struct Linknode{
    char data;
    struct Linknode *next;
}Linknode,*LiStack;

//链栈的初始化函数
void InitStack(LiStack &S){
    S = new Linknode;
    S->next = NULL;
}

//链栈的销毁
void DeleteStack(LiStack &S){
    Linknode * p = S;
    do{
        S = S->next;
        delete(p);
        p = S;
    }while(p);
}

//判断栈空 栈空返回 0 栈非空返回 1
int StackEmpty(LiStack S){
    if(S->next == NULL) return 0;
    else return 1;
}

//入栈
void Push(LiStack &S,char x){
    Linknode * p = new Linknode;
    S->data = x;
    p->next = S;
    S = p;
}

//出栈
void Pop(LiStack &S,char &x){
    if(!StackEmpty(S)) cout<< "The stack is EMPTY!" <<endl;//判断是否为空
```

栈

```
    else{
        Linknode * p = S;
        x = p->next->data;
        S = S->next;
        delete(p);
    }
}
```

//返回栈顶元素

```
char GetTop_Stack(LiStack S){
    if(!StackEmpty(S)){
        cout<< "The stack is EMPTY!" <<endl;
        return '$';
    }//判断是否为空栈
    else{
        return S->next->data;
    }
}
```

//输出栈内元素(LIFO)

```
void PrintStack(LiStack S){
    if(!StackEmpty(S)){
        cout<< "The stack is EMPTY!" <<endl;
    }//判断是否为空栈
    else{
        Linknode * p = S->next;
        do{
            cout<< p->data << ' ';
            p = p->next;
        }while(p);
    }
}
```

//编写函数比较待入栈运算符与栈顶运算符的优先级，优先级低不入栈用0表示，优先级高入栈用1表示，左右括号相互抵消用2表示

```
int OperatorCompare(char rest,char inside){
    if((rest == '+') or (rest == '-')){
        if((inside == '+') or (inside == '-') or (inside == '*') or (inside == '/')) return 0;
        else if((inside == '(') or (inside == '?')) return 1;
        else return 0;
    }
    else if((rest == '*') or (rest == '/')){
```

```

        if((inside == '+') or (inside == '-') or (inside == '(') or (inside
== '?')) return 1;
        else if((inside == '*') or (inside == '/')) return 0;
        else return 0;
    }
    else if(rest == '('){
        return 1;
    }
    else if(rest == ')'){
        if (inside=='(')return 2;
        else return 0;
    }
    else if(rest == '#'){
        if (inside=='?')return 1;
        else return 0;
    }
    else return 0;
}

```

//设置全局变量，维护符号栈

LiStack ope;

int main()

{

    InitStack(ope);

    Push(ope, '?'); //为保证 OperatorCompare 函数初始状态有两个操作数，预先入栈一个元素 '?' 以维护运算符栈的正常工作

    char expression[100]; //输入的中缀表达式上限字符数为 100

    cin>> expression;

    int i=0;

    while(expression[i]!='\0'){ //字符不为结束标志 '\0' 时执行循环

        if((expression[i] == '+') or (expression[i] == '-') or (expression[i] == '\*') or (expression[i] == '/') or (expression[i] == '(') or (expression[i] == ')') or (expression[i] == '#')){

            //判断待入栈运算符与栈顶运算符的优先级

            char top; //栈顶运算符

            int level; //优先级

            do{

                top = GetTop\_Stack(ope);

                level = OperatorCompare(expression[i], top);

                if(level == 1){ //运算符入栈

                    Push(ope, expression[i]);

                    break;

            }



```
        else if(level == 0){//运算符无法入栈，打印栈顶运算符即可
            char next_char;
            Pop(ope,next_char);
            cout<< next_char <<' ';
        }
        else if(level == 2){
            char brackets;
            Pop(ope,brackets);
        }
    }while(level==0);
}
else{//对于操作数，直接打印即可
    cout<<expression[i]<<' ';
}
i++;
}
cout<<'# '<<endl;
}
```

## 附录 5：程序源代码：打印杨辉三角算法

```
#include <iostream>

using namespace std;

typedef struct{
    int* data;
    int front;
    int rear;
}SqQueue;

int MAXSIZE=100;//全局变量设置最大长度

//队列的初始化
int InitQueue(SqQueue &Q){
    Q.data = new int[MAXSIZE-1];
    if(!Q.data) return -1;//内存分配失败
    else{
        Q.front = 0;
        Q.rear = 0;
        return 1;
    }
}

//队列的销毁
void DeleteQueue(SqQueue &Q){
    if(!Q.data){
        cout<< "There is no need to delete because it is EMPTY" <<endl;
    }//判断队列是否为空
    else{
        delete(Q.data);
    }
}

//入队
int EnQueue(SqQueue &Q,int x){
    if(((Q.rear+1)%MAXSIZE) == Q.front){
        cout<< "The queue is FULL" <<endl;
        return -1;
    }//判断队列是否已满
    else{
        Q.data[Q.rear] = x;
        Q.rear = (Q.rear+1) % MAXSIZE;
    }
}
```

```

        return 1;
    }
}

//出队
int DeQueue(SqQueue &Q,int &x){
    if(Q.front == Q.rear){
        cout<< "The queue is EMPTY" <<endl;
        return -1;
    }//判断队列是否为空
    else{
        x = Q.data[Q.front];
        Q.front = (Q.front+1) % MAXSIZE;
        return 1;
    }
}

//返回队头元素
int GetTop_Queue(SqQueue &Q,int &x){
    if(Q.rear == Q.front){
        cout<< "The queue is EMPTY" <<endl;
        return -1;
    }//判断队列是否为空
    else{
        x = Q.data[Q.front];
        return 1;
    }
}

//输出队列内元素(FIFO)
int PrintQueue(SqQueue Q){
    if(Q.front == Q.rear){
        cout<< "The queue is EMPTY" <<endl;
        return -1;
    }//判断队列是否为空
    else if(Q.front < Q.rear){
        for(int i=Q.front;i<Q.rear;i++){
            cout<< Q.data[i] << ' ' ;
        }
        return 1;
    }
    else{
        for(int i=Q.front;i<Q.rear+MAXSIZE;i++){
            cout<< Q.data[i] << ' ' ;

```

```
    }  
    return 1;  
}  
}
```

//设置全局变量，维护循环队列

```
SqQueue q;
```

```
int main()  
{  
    int n;  
    InitQueue(q);  
    cin>>n;  
    for(int i=1;i<=n;i++){  
        cout<<1<<' ';//每次循环开始时单独打印“1”  
        EnQueue(q,1);  
        for(int j=0;j<i-1;j++){  
            int x,y;  
            DeQueue(q,x);  
            DeQueue(q,y);  
            cout<<x+y<<' ';  
            EnQueue(q,x+y);  
            EnQueue(q,x+y);//将两元素和入队两次  
        }  
        cout<<1<<' '<<endl;//每次循环结束时单独打印“1”  
        EnQueue(q,1);  
    }  
}
```