# 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，广度优先遍历可以使用队列实现**。

# 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。

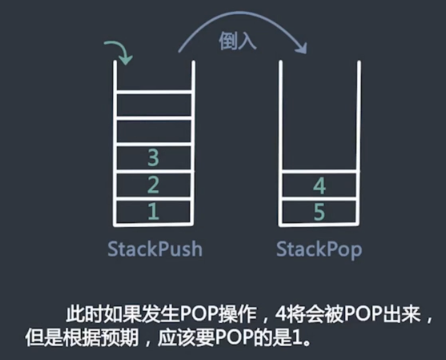
注：入栈出栈应用于生产者-消费者模型。

**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

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

}

}

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;

\*top = temp;

}

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。

**测试：**



# 应用

**直接运用：**

确保算式中的符号能够匹配

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

**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

## 函数调用

## 中断

## 内存分配

## 缓冲处理

## 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：**



**测试：**



## 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

## 进制转换

### 二进制->十进制

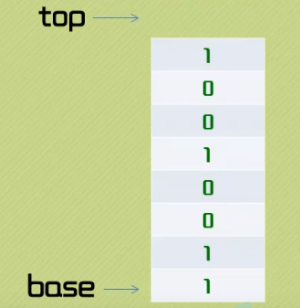
题目：利用栈的数据结构特点，将二进制转换为十进制数。

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：







## 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。



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：**



## 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。

**分析：**

**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)

{

stack<char> st;

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++)

{

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

st.push(str[i]);

else if(str[i]==']')

{

tmp = st.top();

if(tmp != '[')

return false;

else

st.pop();

}

else if(str[i] ==')')

{

tmp = st.top();

if(tmp != '(')

return false;

else

st.pop();

}

else if(str[i]=='}')

{

tmp = st.top();

if(tmp != '{')

return false;

else

st.pop();

}

else

return false;

}

if(st.empty())

return true;

else

return false;

}

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。

## 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;

}

## 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。

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;

}

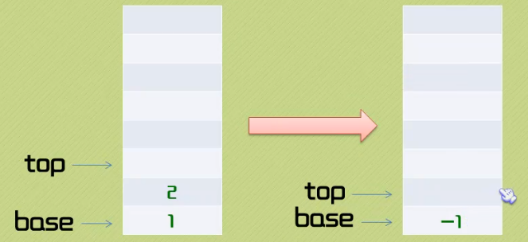
## 逆波兰表达式/求解后缀表达式

逆波兰表达式（RPN）即一种不需要括号的后缀表达式。

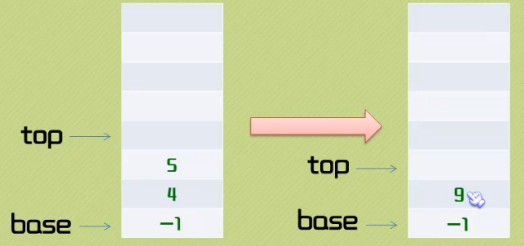
对于(1-2)\*(4+5)，如果用逆波兰表示法应该是：1 2 – 4 5 + \*。虽然这种方式在人看来难以理解，但是对于计算机而言却十分容易理解和处理。因为只需要利用栈的特点，就可以将这种后缀表达式的性能发挥到极致。

其基本原理如下：

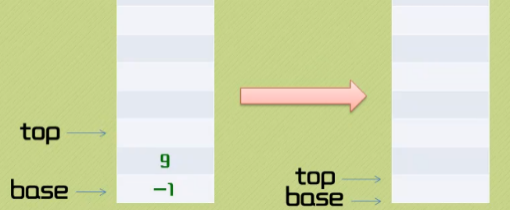
1. 数字1和2进栈，遇到减号运算符弹出两个元素进行运算并把结果入栈：



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;

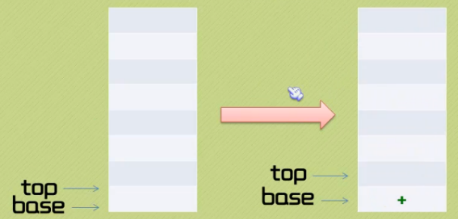
}

## 中缀表达式->后缀表达式

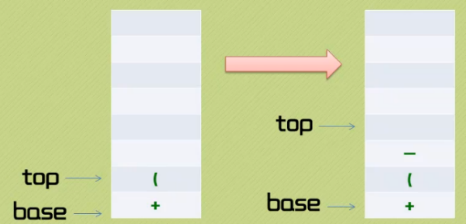
题目：将中缀表达式“(1-2)\*(4+5) ”转化为“1 2 – 4 5 + \*”

分析：利用栈的“记忆”，符号全都推入栈即可。

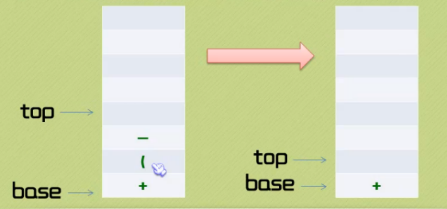
1. 首先遇到第一个输入是数字1，数字在后缀表达式中都是直接输出，接着是符号“+”，入栈：



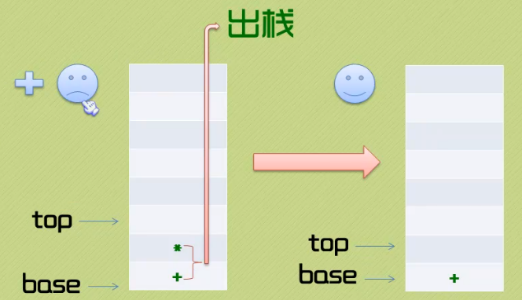
1. 第三个字符是“(”，依然是符号，入栈，接着是数字2，输出，然后是符号“-”，入栈：



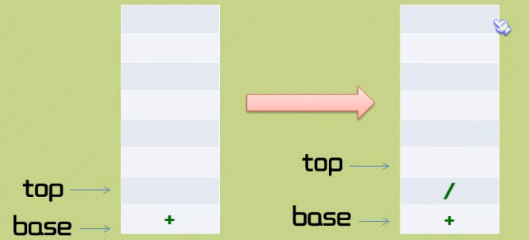
1. 接下来是数字3，输出，紧跟着是“)”，此时，我们需要去匹配栈里的“(”，然后再匹配前将栈顶数据依次出栈（这就好比括号里先执行的道理）：



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。

## 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