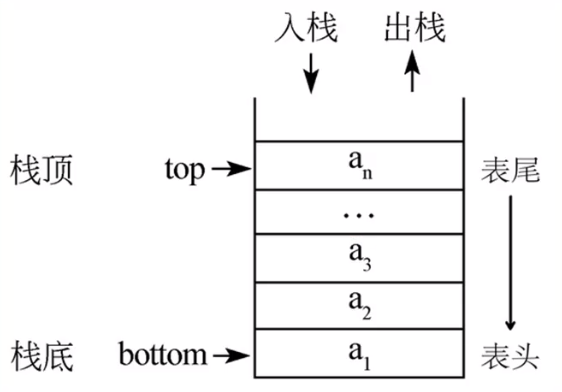
# 定义

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）。

# 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。

# 存储

## 顺序存储

### 定义

typedf struct{

int data[MAX\_SIZE];

int top;

int stackSize; //可以不添加这个变量

}SqStack;

或

typedef struct

{

ElemType \*base;

ElemType \*top;

int stackSize;

}SqStack;//不推荐这种定义方式

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。

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

#define STACK\_INIT\_SIZE 100

initStack(sqStack \*s)

{

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

if( !s->top)

exit(0);

s->stackSize = STACK\_INIT\_SIZE; //可有可无

}

或

#define STACK\_INIT\_SIZE 100

initStack(sqStack \*s)

{

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

if( !s->base)

exit(0);

s->top = s->base; //最开始，栈顶就是栈底

s->stackSize = STACK\_INIT\_SIZE;

}

### 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）。

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。

ClearStack(sqStack \*s)

{

s->top = NULL;

s->stackSize = 0；

}

或

ClearStack(sqStack \*s)

{

s->top = s->base;

}

### 销毁栈

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;

}

### 获取元素

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

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

return -1;

}

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

return 0;

}

### 计算容量

计算栈的当前容量也就是计算栈中元素的个数，因此只要返回s.top-s.base即可。

注：栈的最大容量是指该栈占据内存空间的大小，其值为s.stackSize，它与栈的当前容量不是一个概念。

int StackLen(sqStack s)

{

return (s.top - s.base);

}

### 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;

}

或

#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

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

\*(s->top) = e;

s->top++;

}

### 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;

}

或

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

{

if(s->top == s->base) //栈空

return;

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

}

## 链式存储

### 定义

typedef struct LinkStackNode{

int data;

stuct LinkStackNode \*next;

}LinkStackNode \*LinkStackPtr;

typedef struct LinkStack{

LinkStackPtr top;

int count;//这个需要定义count，栈元素计数器

}LinkStack;//定义时采用指针类型

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

### 销毁

### 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;

}

### 计算容量

### 入栈

思路：

代码：

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

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

s->data = \*element;

s->next=l->top;

//为新节点分配内存并更新成员变量

l->top=s;

l->count++;

return 0;

}

### 出栈

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

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

free(s);

l->count--;

return 0;

}

# 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

struct ArrayStack{

int top;

int capacity;

int \*array;

};

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。

### 初始化

struct ArrayStack \*CreateStack(){

struct ArrayStack \*S = malloc(sizeof(struct ArrayStack));

if(!S)

return NULL:

S->capacity = 1; //定义数组的大小

S->top = -1;

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

if(!S->array)

return NULL;

return S;

}

### 空间判断

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;

}

}

判断栈是否溢出：

int IsFullStack(struct ArrayStack \*S){

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

}

或者：

int IsFullStack(struct ArrayStack \*S){

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

return 1;

}else{

return 0;

}

}

### 入栈

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

}

}

### 出栈

int Pop(struct ArrayStack \*S){

if(IsEmptyStack(S)){

printf(“Stack is Empty!”);

}else{

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

}

}

### 删除栈

void DeleteStack(struct ArratStack \*S){

if(S){

if(S->array)

free(S->array);

free(S);

}

}

### 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。

## 动态数组

### 定义

struct DynArrayStack{

int top;

int capacity;

int \*array;

};

### 初始化

struct DynArrayStack \*CreateStack(){

struct DynArrayStack \*S = malloc(sizeof(struct DynArrayStack));

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;

}

### 空间判断

判断栈是否为空：

int IsEmptyStack(struct DynArrayStack \*S){

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

}

或者：

int IsEmptyStack(struct DynArrayStack \*S){

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

return 1;

}else{

return 0;

}

}

判断栈是否溢出：

int IsFullStack(struct DynArrayStack \*S){

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

}

或者：

int IsFullStack(struct DynArrayStack \*S){

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

return 1;

}else{

return 0;

}

}

### 翻倍

void DoubleStack(struct DynArrayStack \*S){

S->capacity \*= 2;

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

}

### 入栈

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

if(IsFullStack(S)){

DoubleStack(S); //此时不报错直接翻倍

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

}

}

### 出栈

int Top(struct DynArrayStack \*S)

{

if(IsEmptyStack(S)){

return INT\_MIN;

}

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

}

int Pop(struct DynArrayStack \*S){

if(IsEmptyStack(S)){

return INT\_MIN;

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

}

### 删除栈

void DeleteStack(struct DynArratStack \*S){

if(S){

if(S->array)

free(S->array);

free(S);

}

}

### 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

struct Stack{

int data;

struct Stack \*next;

};

### 初始化

struct Stack \*CreateStack(){

return NULL;

}

### 空间判断

判断栈是否为空：

int IsEmptyStack(struct Stack \*top){

return top ==NULL;

}

### 入栈

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

struct Stack \*temp;

temp = malloc(sizeof(struct Stack));

if(!temp)

return;

temp->data = data;

temp->next = \*top;

\*top = temp;

}

### 出栈

int Top(struct Stack \*top){

if(IsEmptyStack(top))

return INT\_MIN;

return top->next->data;

}

int Pop(struct Stack \*\*top)

{

int data;

struct Stack \*temp;

if(IsEmptyStack(\*top))

return INT\_MIN;

temp = \*top;

\*top = (\*top)0->next;

data = temp->data;

free(temp);

return data;

}

### 删除栈

void DeleteStack(struct Stack \*\*top){

struct Stack \*temp,\*p;

s = \*top;

while(p->next){

temp = p->next;

p->next = temp->next;

free(temp);

}

free(p);

}

### 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

# 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

## 进制转换

### 二进制->十进制

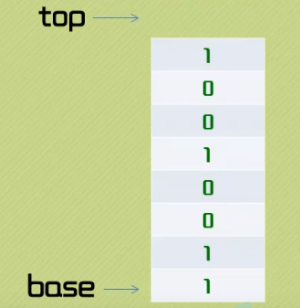
题目：利用栈的数据结构特点，将二进制转换为十进制数。

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);

}

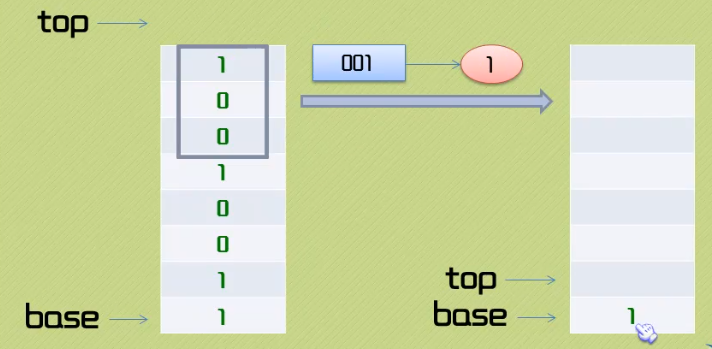
printf("转化后十进制数：%d\n",sum);

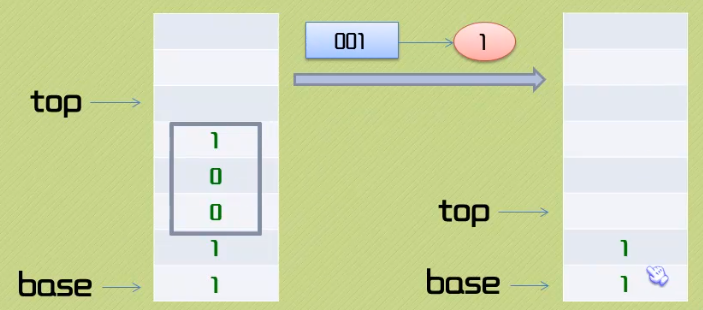
### 二进制->八进制

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：







## 括号匹配检测

题目要求：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。

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;

}

/\*

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;

}

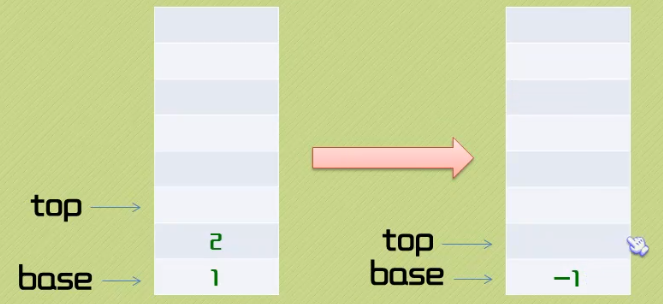
## 逆波兰表达式/求解后缀表达式

逆波兰表达式（RPN）即一种不需要括号的后缀表达式。

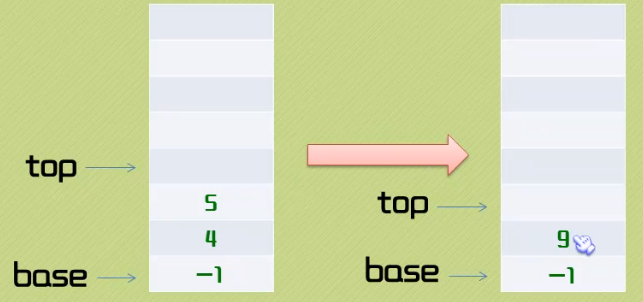
对于(1-2)\*(4+5)，如果用逆波兰表示法应该是：1 2 – 4 5 + \*。虽然这种方式在人看来难以理解，但是对于计算机而言却十分容易理解和处理。因为只需要利用栈的特点，就可以将这种后缀表达式的性能发挥到极致。

其基本原理如下：

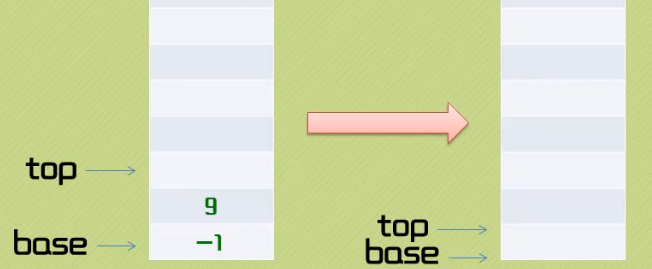
1. 数字1和2进栈，遇到减号运算符弹出两个元素进行运算并把结果入栈：



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;

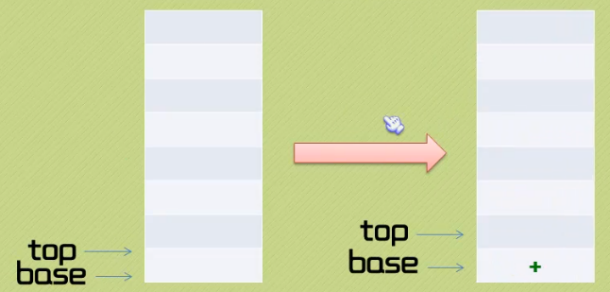
}

## 中缀表达式->后缀表达式

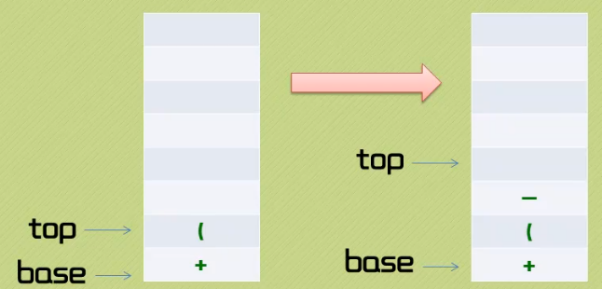
题目：将中缀表达式“(1-2)\*(4+5) ”转化为“1 2 – 4 5 + \*”

分析：利用栈的“记忆”，符号全都推入栈即可。

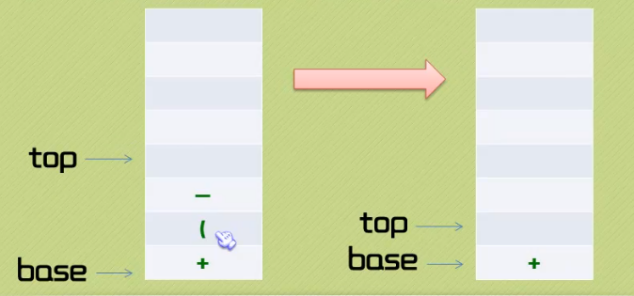
1. 首先遇到第一个输入是数字1，数字在后缀表达式中都是直接输出，接着是符号“+”，入栈：



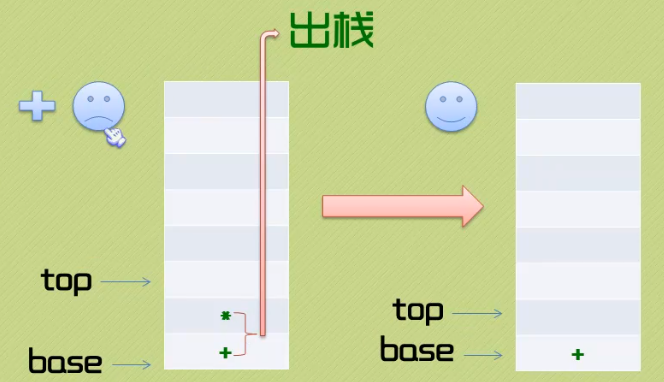
1. 第三个字符是“(”，依然是符号，入栈，接着是数字2，输出，然后是符号“-”，入栈：



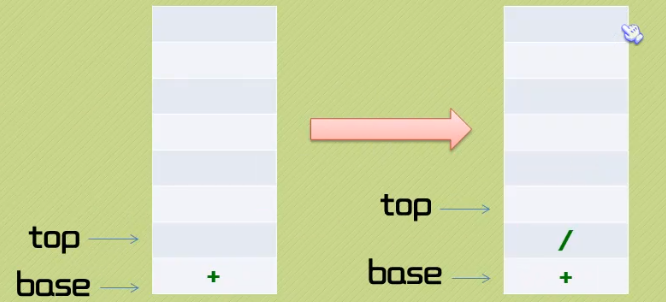
1. 接下来是数字3，输出，紧跟着是“)”，此时，我们需要去匹配栈里的“(”，然后再匹配前将栈顶数据依次出栈（这就好比括号里先执行的道理）：



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

题目要求：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