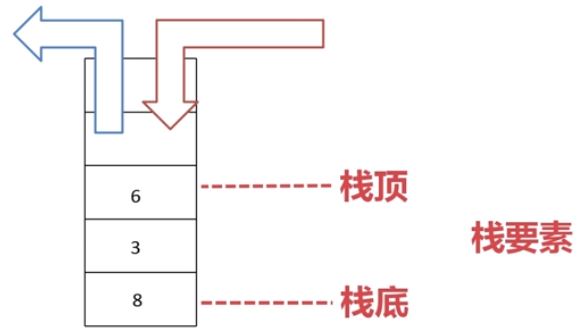
# 定义

栈是后进先出（LIFO=Last In First Out），限定为只能在一端进行插入和删除操作。



# 顺序存储

## 定义

typedf struct{

int data[MAX\_SIZE];

int top;

}SqStack;

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。

## 获取元素

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

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

return -1;

}

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

return 0;

}

## 插入元素

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;

}

## 删除元素

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;

}

# 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

}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

## 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;

}

## 插入元素

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;

}

## 删除元素

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

\*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

## 用简单的数组实现栈

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。

## 动态数组实现栈

### 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）。

## 用链表实现栈

### 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) |

# 作用

# 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

## 进制转换

## 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;

}

## 求解后缀表达式

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;

}

## 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;

}