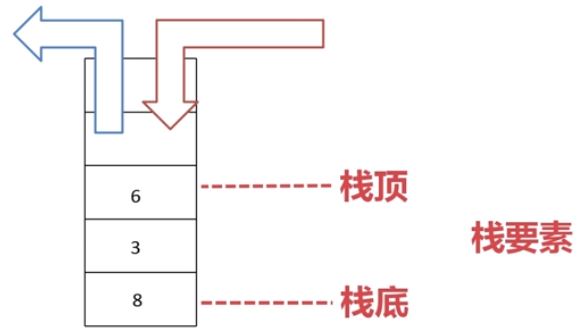
# 定义

后进先出（LIFO=Last In First Out）



# 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）。

# 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