栈（Stack）

栈是限定仅在表尾进行插入或删除操作的线性表

因此，栈又称为后进先出的（Last in first out）线性表

因此，对于栈来说，表尾端有其特殊含义，称为栈顶（top），相应地表头端称为栈底（bottom）

抽象数据类型

```
ADT Stack{
    数据对象：D = {ai|ai∈ElementSet, i = 1,2,...,n, n>=0}

    数据关系：R1 = {<ai-1,ai>|ai-1,ai∈D,i = 2,...,n}

    基本操作：

    //初始化栈
    Status InitStack(Stack);

    //清空栈
    Status ClearStack(Stack);

    //是否为空栈
    boolean isEmptyStack(Stack);

    //获取栈的长度
    int StackLength(Stack);

    //获取栈顶元素
    ElementType GetTop(Stack);

    //压栈
    Status Push(Stack, ElementType);

    //出栈
    ElementType Pop(Stack);

    //遍历栈
    void StackTraverse(Stack, visit());
}
```

栈的表示和实现

和线性表类似，栈也有两种存储表示方法；这里主要介绍顺序栈

顺序栈，即栈的顺序存储结构是利用一组地址连续的存储单元依次存放自栈底到栈顶的数据元素

同时附设指针top指示栈顶元素在顺序栈中的位置

在初始化栈时使用的大小很难估计，因此一般来说在初始化栈时不应该限定栈的容量；

应当为栈分配一个基本容量，然后在使用过程中，当栈的空间不够时再逐渐扩大

```
typedef struct{
    StackElementType    *base;      //栈底指针
    StackElementType    *top;       //栈顶指针
    int                 stackSize;  //当前栈容量
}
```

top和base值的选择

关于顺序栈 top初始值的不同有两种实现方式

第一种：top = 0

- 入栈 *(stack.top++) = value
- 出栈 value = *(--stack.top)
- 栈顶 value = *(stack.top - 1)
- 栈空 stack.top == stack.base
- 栈满 stack.top == stack.base + stack.stackSize

第二种：top = -1

- 入栈 *(++stack.top) = value
- 出栈 value = *(stack.top--)
- 栈顶 value = *(stack.top)
- 栈空 stack.top + 1 == stack.base
- 栈满 stack.top + 1 == stack.base + stack.stackSize

初始化栈

```
/**
 * 功能：初始化顺序栈
 * 参数：stack 被初始化的栈
 * 返回值：初始化结果
*/
Status initStack(SequenceStack &stack){

    //如果stack为NULL那么后面都无法操作了
    if(&stack == NULL)
        return ERROR;

    //构造一片连续的内存空间用于存储栈的元素
    ElementType *elements = (ElementType *)malloc(STACK_INIT_SIZE *
```

```
sizeof(ElementType));

    if(elements == NULL)
        exit(OVERFLOW);

    //栈顶 栈底都指向新申请的内存空间
    stack.base = stack.top = elements;

    //设置栈的初始长度
    stack.stackSize = STACK_INIT_SIZE;

    return TRUE;
}
```

获取栈顶元素

```
/**
 * 功能：获取栈顶元素但不弹出
 * 参数：stack 栈 returnElement 通过此参数返回栈顶元素
 * 返回值：获取结果
*/
Status getTop(SequenceStack stack,ElementType &returnElement){
    if(&stack == NULL)
        return ERROR;

    if(isEmpty(stack)){
        return FALSE;
    }
    else{
        returnElement = *(stack.top - 1);
        return TRUE;
    }
}
```

压栈

```
/**
 * 功能：向栈顶压入一个元素
 * 参数：stack 栈 element 要压入的元素
 * 返回值：压入结果
*/
Status push(SequenceStack &stack,ElementType element){
    if(&stack == NULL)
        return ERROR;

    //如果栈已经满了 那么需要申请新的内存空间
```

```
    if(stack.stackSize <= (stack.top - stack.base)){
        //重新申请一个新的内存空间
        //新内存空间的大小为（原来的大小 + 增量）* 一个元素的大小
        int newSize = (stack.stackSize + STACK_INCREMENT) * sizeof(ElementType);
        stack.base = (ElementType *)realloc(stack.base,newSize);

        if(stack.base == NULL)
            exit(OVERFLOW);

        //新的栈顶
        stack.top = stack.base + stack.stackSize;
        stack.stackSize += STACK_INCREMENT;
    }

    *(stack.top++) = element;

    return TRUE;
}
```

出栈

```
/**
 * 功能：弹出一个元素
 * 参数：stack 栈 returnElement 通过此参数返回弹出的栈顶元素
 * 返回值：弹出结果
*/
Status pop(SequenceStack &stack,ElementType &returnElement){
    if(&stack == NULL)
        return ERROR;


    if(isEmpty(stack))
        return FALSE;

    returnElement = *(--stack.top);
    return TRUE;
}
```