通常称，栈和队列是限定**插入和删除只能**在表的“**端点**”进行的线性表。

| **线性表** | **栈** | **队列** |
| --- | --- | --- |
| Insert(L, **i**, x)<br>**1≤i≤n+1** | Insert(S, **n+1**, x) | Insert(Q, **n+1**, x) |
| Delete(L, **i**)<br>1≤i≤n | Delete(S, **n**) | Delete(Q, **1**) |

# 3.1 栈

## 3.1.1 抽象数据类型栈的定义

栈（Stack）是限定仅在表尾进行插入或删除操作的线性表。通常称其表尾为栈顶（top），表头端称为栈底（bottom）。

假设栈$S=\{ a_i \mid a_i \in ElemSet, i=1,2,...,n, n\geq 0 \}$，则称$a_1$为栈底元素，$a_n$为栈顶元素，栈中元素按$a_1,a_2,...a_n$的次序进栈，退栈的第一个元素应为栈顶元素。换句话说，栈的修改是按后进先出的原则进行的。因此，栈又称为后进先出（last in first out）的线性表（简称LIFO表）。

# 栈的抽象数据类型定义：

**ADT Stack {**

**数据对象**：

$D = \{ a_i \mid a_i \in ElemSet, i=1,2,...,n, \ n \geq 0 \}$

**数据关系**：

$R1 = \{ <a_{i-1}, a_i> \mid a_{i-1}, a_i \in D, i=2,...,n \}$

约定$a_n$ 端为栈顶，$a_1$ 端为栈底。

**基本操作：**

**} ADT Stack**

**ADT Stack {**

数据对象：D={$a_i$| ai∈ElemSet, i=1,2,…,n, n≥0}

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

约定$a_n$端为栈顶，$a_1$ 端为栈底

基本操作：

**InitStack(&S)**

操作结果：构造一个空栈 S。

**DestroyStack(&S)**

初始条件：栈 S 已存在。

操作结果：栈 S 被销毁。

**ClearStack(&S)**

初始条件：栈 S 已存在。

操作结果：将 S 清为空栈。

**StackEmpty(S)**

初始条件：栈 S 已存在。

操作结果：若栈 S 为空栈，则返TRUE，否则 FALE。

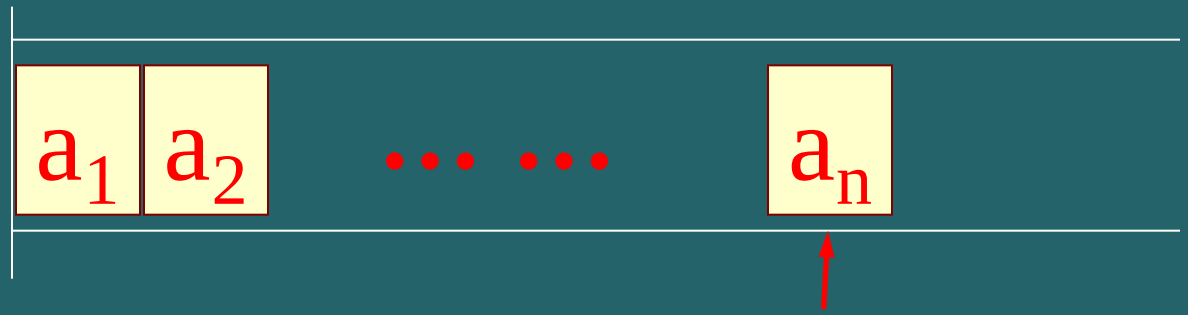
**StackLength(S)**

初始条件：栈 S 已存在。

操作结果：返回 S 的元素个数，即栈的长度。

**GetTop(S, &e)**

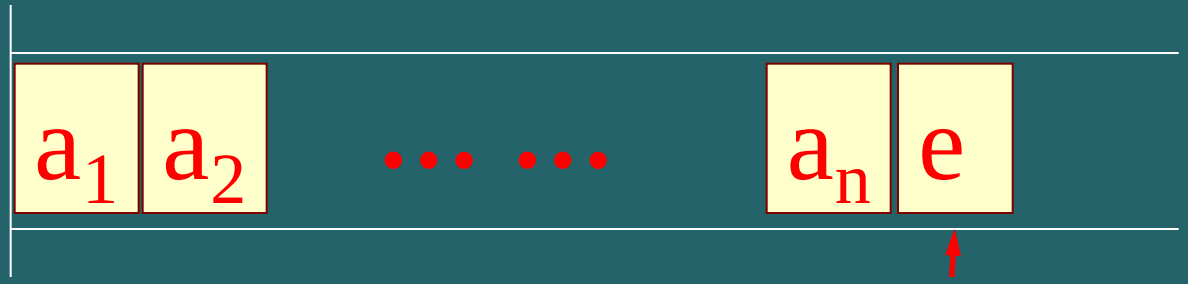
初始条件：栈 S 已存在且非空。操作结果：用 e 返回 S 的栈顶元素。




**Push(&S, e)**

初始条件：栈 S 已存在。

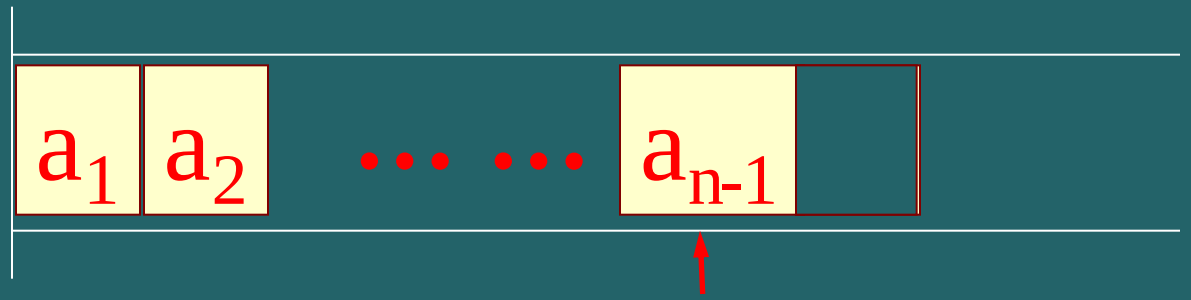
操作结果：插入元素 e 为新的栈顶元素。



**Pop(&S, &e)**

初始条件：栈 S 已存在且非空。

操作结果：删除 S 的栈顶元素，并用 e 返回其值




**StackTraverse (S, visit())**

初始条件：栈 S 已存在。

操作结果：从栈底到栈顶依次对S 的每个数据元素调用函数visit()。一旦visit()失败，则操作失败。

**} ADT Stack**

# 3.1.2 栈的表示和实现

**顺序栈的定义：**

```
typedef struct {
   SElemType      *base;  //栈底指针
   SElemType      *top;   //栈顶指针
   int     stacksize;
 } SqStack
```

其中， stacksize指示栈的当前可使用的最大容量。Top为栈顶指针，其初值指向栈底，每当插入新元素，指针top加1，删除栈顶元素时，指针top减1。非空栈中，top始终指向栈顶元素的下一个位置。

以下是顺序栈的模块说明。

```
//-----ADT Stack的表示与实现-----
//-----栈的顺序存储表示-----
#define STACK_ INIT_ SIZE 100;     //存储空间初始分配量
#define STACKINCREMENT 10;     //存储空间分配增量
typedef struct  {
   SElemType       *base;
     //在栈构造之前和销毁之后， base的值为NULL
   SElemType       *top;                  //栈顶指针
   int      stacksize;
                  //当前已分配的存储空间，以元素为单位
} SqStack
```

**-----基本操作的函数原型说明-----**

```
Status InitStack (SqStack &S );
  // 构造一个空栈S

Status DestroyStack (SqStack &S );
  // 销毁栈S，S不再存在

Status ClearStack (SqStack &S );
  // 重置S 为空栈

Status StackEmpty (SqStack S );
  //若栈为空栈，则返回TRUE，否则 返回FALSE

 int  StackLength (SqStack S );
          //返回S的元素个数，即栈的长度
```

**Status** GetTop (SqStack S , SElemType &e )

//若栈不空，则用e返回S的栈顶元素，并返回OK，

否则返回ERROR

**Status** Push (SqStack **&**S , SElemType e )

// 插入元素e为新的栈顶元素

**Status** Pop (SqStack **&**S , SElemType &e )

//若栈不空，则删除S的栈顶元素，并用e返回其值，

并返回OK，否则返回ERROR

**Status** StackTraverse (SqStack **&**S ,

**Status**(*visit)() );

//从栈底到栈顶依次对栈中每个元素调用函数visit()

。一旦visit()失败，则操作失败

```
//-----基本操作的算法描述（部分）-----

Status InitStack (SqStack &S );
    // 构造一个空栈S
    S.base = (SElemType*) malloc
    (STACK_INIT_SIZE*sizeof (ElemType));
   if (! S.base) exit(OVERFLOW);
    //存储分配失败
        S.top =S.base;
        S.stacksize =STACK_INIT_SIZE;
   return OK;
}//InitStack
```

```
Status GetTop (SqStack S , SElemType &e )
   //若栈不空，则用e返回S的栈顶元素，并
   //返回OK，否则返回ERROR
   if(S.top ==S.base)   return ERROR；
      e=*(S.top-1)；
   return OK；
}// GetTop
```

```
Status Push (SqStack &S , SElemType e )
{                    // 插入元素e为新的栈顶元素
if(S.top -S.base >= S.stacksize) {
  {                  // 当前存储空间已满，增加分配
      S.base = (ElemType *)realloc(S.base,
          (S.stacksize+STACKINCREMENT)*sizeof  (ElemType))；
    if (! S.base ) exit(OVERFLOW)；// 存储分配失败
    S.top =S.base+S.stacksize；
    S.stacksize+= STACKINCREMENT //增加存储容量
   }
  *S.top++=e；
  return OK；
}//  Push
```

```
Status Pop (SqStack &S , SElemType &e )
      //若栈不空，则删除S的栈顶元素，并
      //用e返回其值，并返回OK，否则返回
      //ERROR
     if(S.top ==S.base)   return ERROR；
         e=*--S.top；
     return OK；
}// Pop
```