第3章　栈和队列

3.1 栈

3.2 栈的应用举例

3.3 栈与递归的实现

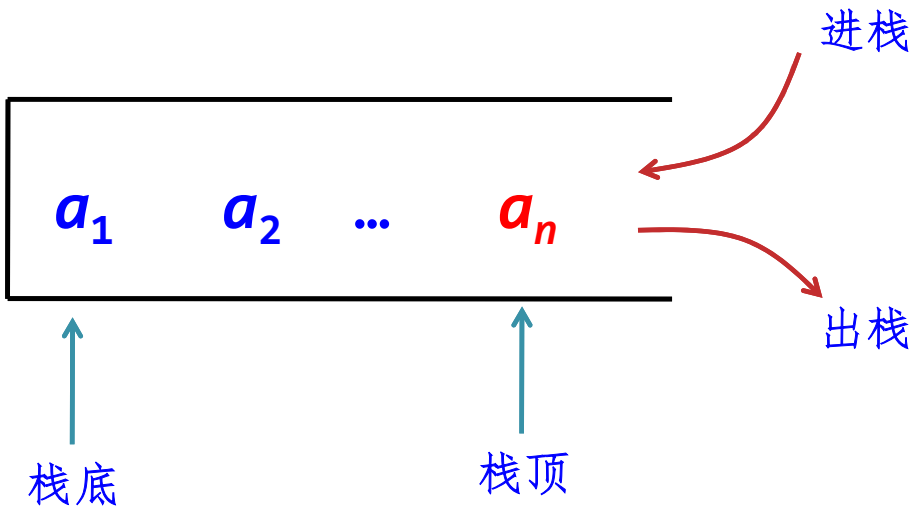
3.4 队列

3.1 栈

3.1.1 栈的基本概念

- 栈是一种特殊的线性表，其特殊性体现在元素插入和删除运算上，它的插入和删除运算仅限定在线性表的某一端进行，不能在表中间和另一端进行。
- 栈的插入操作称为进栈（入栈、Push），删除操作称为出栈（退栈、Pop）。
- 允许进行插入和删除的一端称为栈顶，另一端称为栈底。
- 处于栈顶位置的数据元素称为栈顶元素。
- 不含任何数据元素的栈称为空栈。

栈的模型示意图：



- 栈又称为后进先出LIFO (Last In First Out)或先进后出FILO (First In Last Out)线性表。

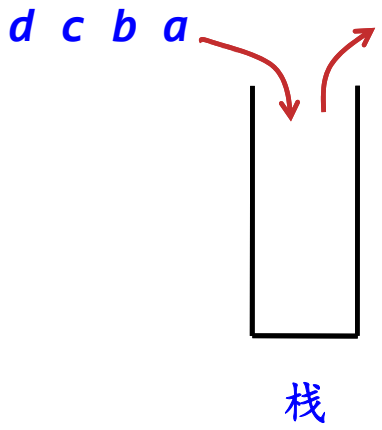
栈的基本运算主要包括以下6种：

- 初始化栈InitStack(S)。建立一个空栈S。
- 销毁栈DestroyStack(S)。释放栈S占用的内存空间。
- 进栈Push(S, *x*)。将元素*x*插入栈S中，使*x*成为栈S的栈顶元素。
- 出栈Pop(S, *x*)。当栈S不空时，将栈顶元素赋给*x*，并从栈中删除当前栈顶。
- 取栈顶元素GetTop(S)。若栈S不空，取栈顶元素*x*并返回1；否则返回0。
- 判断栈空StackEmpty(S)。判断栈S是否为空栈。

【示例-1】设一个栈的输入序列为a、b、c、d，借助一个栈所得到的输出序列不可能是（ ）。

A. a、b、c、d　　B. b、d、c、a

C. a、c、d、b　　D. d、a、b、c



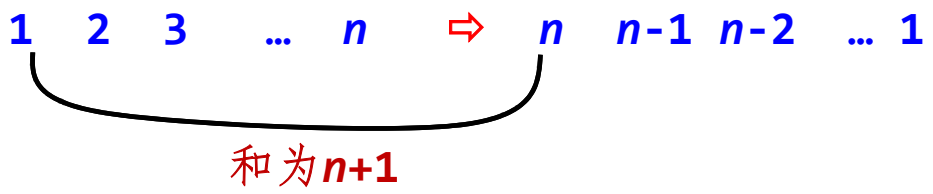
选择的答案是：D

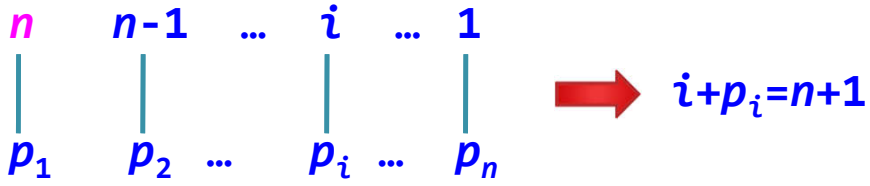
【示例-2】已知一个栈的进栈序列是1，2，3，…，n，其出栈序列是p1，p2，…，pn，若p1=n，则pi的值为（ ）。

A. i　　B. n-i

C. n-i+1　　D. 不确定

解：$p_1=n$，只有唯一的出栈序列





选择的答案是：C

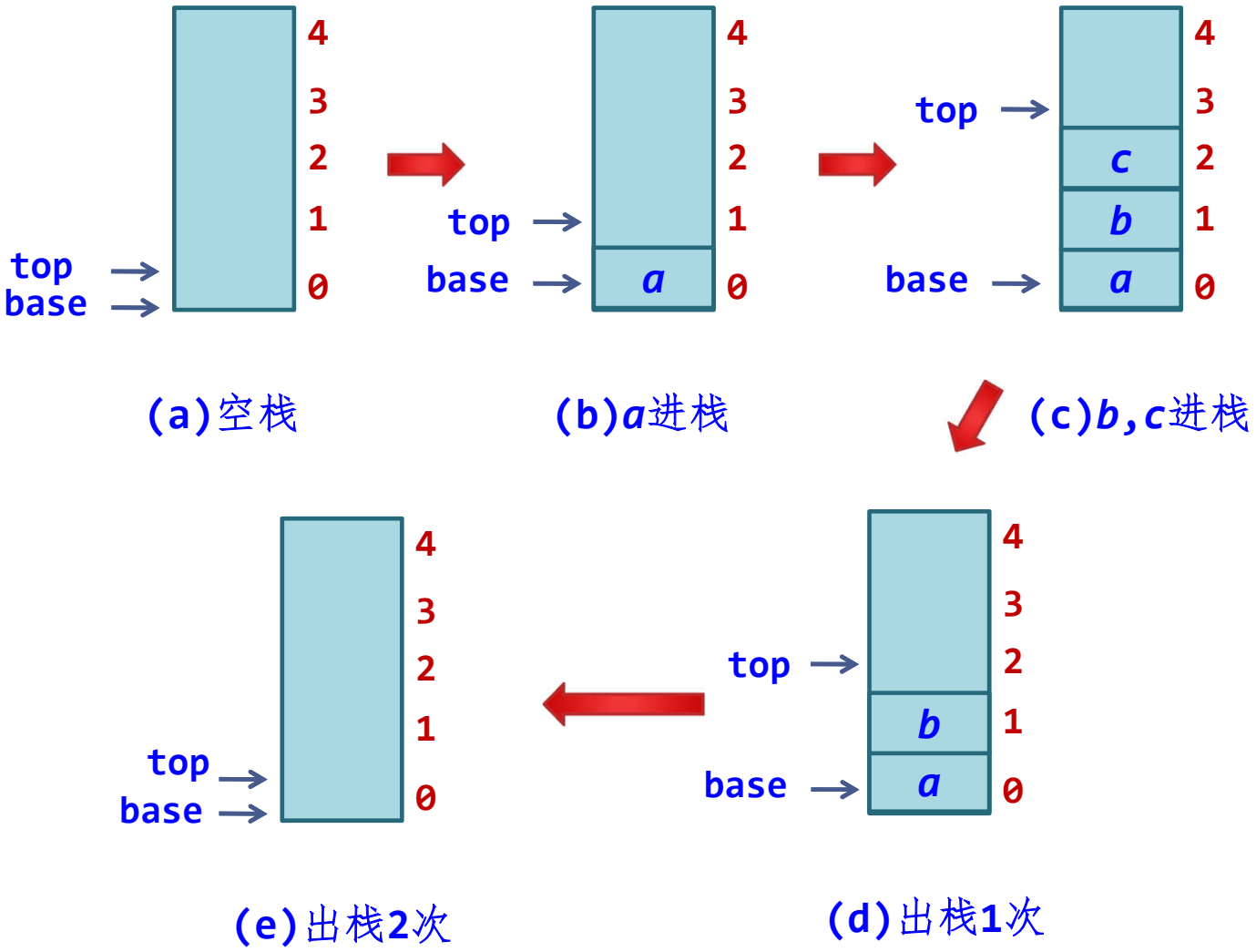
3.1.2 栈的顺序存储结构

- 栈的顺序存储结构称为顺序栈。
- 顺序栈通常由一个一维数组data和一个记录栈顶元素位置的变量top组成。
- 习惯上将栈底放在数组下标小的那端，栈顶元素由栈顶指针top所指向。

◆ 顺序栈类型声明如下：

```
// ------栈的顺序存储结构表示----------
#define STACK_INIT_SIZE 100   // 存储空间初始分配量
#define STACK_INCREMENT 10    // 存储空间分配增量
typedef char ElemType;
typedef struct {
  ElemType *base;      // 栈底指针
  ElemType *top;       // 栈顶指针
  int stacksize;       // 栈空间大小
} SqStack;
```

- 用base表示栈底指针，栈底固定不变的；栈顶则随着进栈和退栈操作而变化。
- 用top指示当前栈顶位置。
- 用top==base作为栈空的标记，每次top指向栈顶数组中的下一个存储位置。



顺序栈的几种状态

- 对于顺序栈S，可通过动态申请得到其存储空间，它有4个要素：

  - ✓ 栈空条件：S.top == S.base
  - ✓ 栈满条件：S.top-S.base >= S.stacksize
  - ✓ 元素e进栈操作：*(S.top)++ = e
  - ✓ 出栈元素e操作：e = *--S.top

顺序栈的基本算法设计

（1）初始化栈算法

主要操作：通过动态申请所需的存储空间，并且初始化相关的参数。

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

（2）销毁栈算法

由于顺序栈的内存空间是由动态申请的，在不再需要时应该主动释放空间。

```
void DestroyStack(SqStack &S)
{
    // 销毁栈S，S不再存在
    free(S.base);
    S.base = NULL;
    S.top = NULL;
    S.stacksize = 0;
}
```

（3）进栈算法

主要操作：判断是否存在“溢出”。

```
void Push(SqStack &S, ElemType e)
{
    if(S.top - S.base >= S.stacksize) { // 栈满，追加存储空间
        S.base = (ElemType *)realloc(S.base, (S.stacksize
              + STACK_INCREMENT) * sizeof(ElemType));
        if(!S.base)
            exit(OVERFLOW);           // 存储分配失败
        S.top = S.base + S.stacksize;
        S.stacksize += STACK_INCREMENT;
    }
    *(S.top)++ = e;
}
```

(4) 出栈运算算法

主要操作：判断是否为空栈。

```
Status Pop(SqStack &S, ElemType &e)
{
    // 若栈不空，则删除S的栈顶元素，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;
}
```

（5）取栈顶元素算法

主要操作：通过top指针，取出栈顶元素。需要注意的是，top指针没有发生改变。

```
Status GetTop(SqStack S, ElemType &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) {
        e = *(S.top - 1);
        return OK;
    }
    else
        return ERROR;
}
```

（6）判断栈空算法

主要操作：若栈为空（top==base）则返回值1，否则返回值0。

```
Status StackEmpty(SqStack S)
{
    // 若栈S为空栈，则返回TRUE，否则返回FALSE
    if(S.top == S.base)
        return TRUE;
    else
        return FALSE;
}
```

END