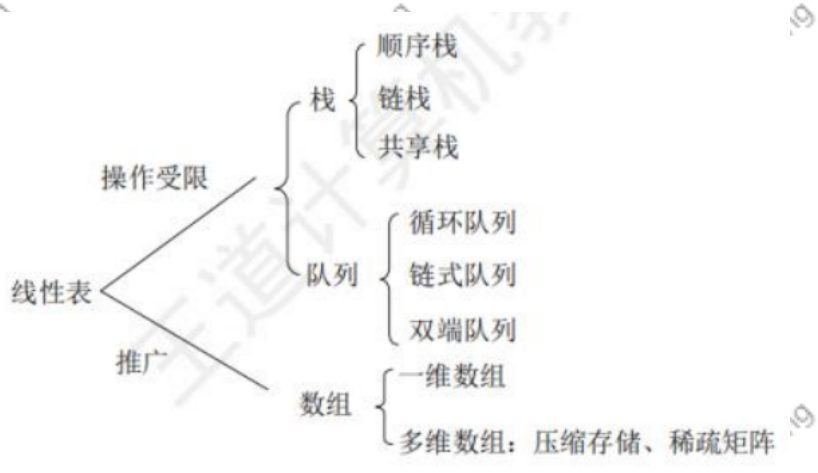
考纲内容

1. 栈和队列的基本概念
2. 栈和队列的顺序存储结构
3. 栈和队列的链式存储结构
4. 多维数组的存储
5. 特殊矩阵的压缩存储
6. 栈、队列和数组的应用

知识框架



复习提示

本章通常以选择题的形式考查，题目不算难，但命题的形式比较灵活，其中栈（出入栈的过程、出栈序列的合法性）和队列的操作及其特征是重点。因为它们均是线性表的应用和推广，所以也容易出现在算法设计题中。此外，栈和队列的顺序存储、链式存储及其特点、双端队列的特点、栈和队列的常见应用，以及数组和特殊矩阵的压缩存储都是读者必须掌握的内容。

栈

栈的基本概念

栈的定义

栈(Stack)是只允许在一端进行插入或删除操作的线性表。首先栈是一种线性表，但限定这种线性表只能在某一端进行插入和删除操作。

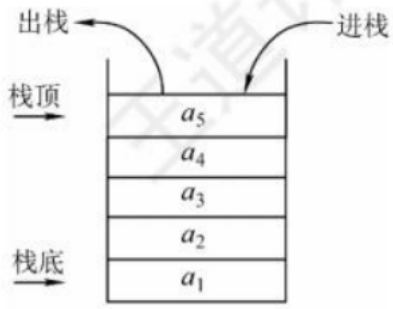


图 3.1　栈的示意图

栈顶（Top）。线性表允许进行插入删除的那一端。

栈底（Bottom）。固定的，不允许进行插入和删除的另一端。

空栈。不含任何元素的空表。

假设某个栈S=(a1,a2,a3,a4,a5)，则a1为栈底元素，a5为栈顶元素。栈只能在栈顶进行插入和删除操作，进栈次序依次为a1,a2,a3,a4,a5，而出栈次序为a5,a4,a3,a2,a1。由此可见，栈的操作特性可以明显地概括为后进先出（Last In First Out, LIFO）。

每接触一种新的数据结构，都应从其逻辑结构、存储结构和运算三个方面着手。

栈的基本操作

各种辅导书中给出的基本操作的名称不尽相同，但所表达的意思大致是一样的。这里我们以严蔚敏编写的教材为准给出栈的基本操作，希望读者能熟记下面的基本操作。

- InitStack(&S)：初始化一个空栈S。
- StackEmpty(S)：判断一个栈是否为空，若栈S为空则返回true，否则返回false。
- Push(&S,x)：进栈，若栈S未满，则将x加入使之称为新栈顶。
- Pop(&S,&x)：出栈，若栈S非空，则弹出栈顶元素，并用x返回。
- GetTop(S,&x)：读栈顶元素，但不出栈，若栈S非空，则用x返回栈顶元素。
- DestroyStack(&S)：销毁栈，并释放栈S占用的存储空间（"&"表示引用调用）。

在解答算法题时，若题干未做出限制，则也可直接使用这些基本的操作函数。

栈的数学性质：当n个不同元素进栈时，出栈元素不同排列的个数为

$$\frac{1}{n+1}C_{2n}^{n}$$

这个公式称为卡特兰数（Catalan）公式，有兴趣的读者可以参考组合数学教材。

栈的顺序存储结构

栈是一种操作受限的线性表，类似于线性表，它也有对应的两种存储方式。

顺序栈的实现

采用顺序存储的栈称为顺序栈，它利用一组地址连续的存储单元存放自栈底到栈顶的数据元素，同时附设一个指针（top）指示当前栈顶元素的位置。

栈的顺序存储类型可描述为

```
#define MaxSize 50//定义栈中元素的最大个数
typedef struct{
    Elemtype data[MaxSize];//存放栈中元素
    int top;//栈顶指针
}SqStack;
```

栈顶指针：S.top，初始时设置S.top=-1；栈顶元素：S.data[S.top]。

进栈操作：栈不满时，栈顶指针先加1，再送值到栈顶。

出栈操作：栈非空时，先取栈顶元素，再将栈顶指针减1。

栈空条件：S.top==-1；栈满条件：S.top==MaxSize-1；栈长：S.top+1。

另一种常见的方式是：初始设置栈顶指针S.top=0；进栈时先将值送到栈顶，栈顶指针再加1；出栈时，栈顶指针先减1，再取栈顶元素；栈空条件是S.top==0；栈满条件是S.top==MaxSize。

顺序栈的入栈操作受数组上界的约束，当对栈的最大使用空间估计不足时，有可能发生栈上溢，此时应及时向用户报告消息，以便及时处理，避免出错。

栈和队列的判空、判满条件，会因实际给的条件不同而变化，下面的代码实现是在栈顶指针初始化为-1的条件下的相应方法，而其他情况则需具体问题具体分析。

顺序栈的基本操作

栈操作的示意图如图所示，a是空栈，c是ABCDE共5个元素依次入栈后的结果，d是在图c之后EDC的相继出栈，此时栈中还有2个元素，或许最近出栈的元素CDE仍在原先的单元存储着，但top指针已经指向了新的栈顶，元素CDE已不在栈中，读者通过该示意图深刻理解栈顶指针的作用。

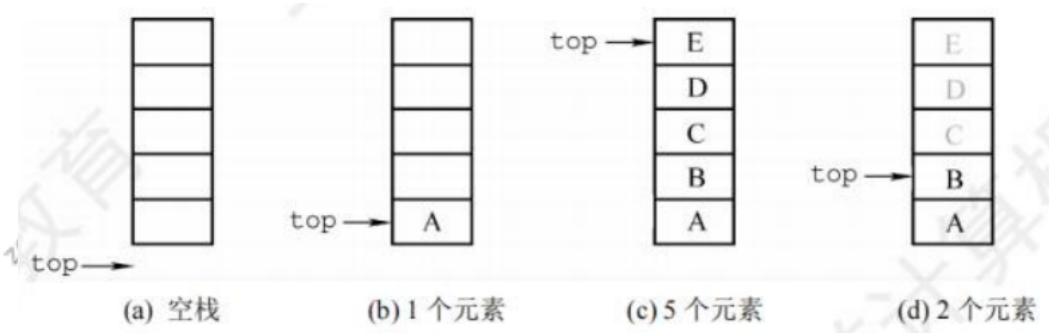


图 3.2　栈顶指针和栈中元素之间的关系

下面是顺序栈上常用的基本操作的实现。

1. 初始化

```
void InitStack(SqStack &S){
    S.top=-1;//初始化栈顶指针
}
```

2. 判栈空

```
bool StackEmpty(SqStack S){
    if(S.top==-1)//栈空
        return true;
    else//不空
        return false;
}
```

3. 进栈

```
bool Pop(SqStack &S, ElemType &x){
    if(S.top==MaxSize-1)//栈满,报错
        return false;
    S.data[++S.top]=x;//指针先加1,再入栈
    return true;
}
```

4. 出栈

```
bool Pop(SqStack &S, ElemType &x){
    if(S.top==-1)//栈空,报错
        return false;
    x=S.data[S.top--];//先出栈,指针再减1
    return true;
}
```

5. 读栈顶元素

```
bool GetTop(SqStack S, ElemType &x){
    if(S.top==-1)//栈空,报错
        return false;
    x=S.data[S.top];//x记录栈顶元素
    return true;
}
```

仅为读取栈顶元素，并没有出栈操作，因此原栈顶元素依然保留在栈中。

这里的top指的是栈顶元素。于是，进栈操作为S.data[++S.top]=x，出栈操作为x=S.data[S.top--]。若栈顶指针初始化为S.top=0，即top指向栈顶元素的下一位置，则入栈操作变为S.data[S.top++]=x；出栈操作变为x=S.data[--S.top]。相应的栈空、栈满条件也会发生变化。请读者仔细体会其中的不同之处，做题时要灵活应变。

共享栈

利用栈底位置相对不变的特性，可以让两个顺序栈共享一个一维数组空间，将两个栈的栈底分别设置在共享空间的两端，两个栈顶向共享空间的中间延申。

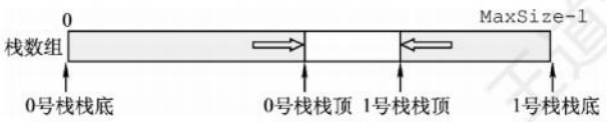


图 3.3　两个顺序栈共享存储空间

两个栈的栈顶指针都指向栈顶元素，top0=-1时0号栈为空，top1=MaxSize时1号栈为空；仅当两个栈顶指针相邻（top1-top0=1）时，判断为栈满。当0号栈进栈时top0先加1再赋值，1号栈进栈时top1先减1再赋值；出栈时则刚好相反。

共享栈是为了更有效地利用存储空间，两个栈的空间相互调节，只有在整个存储空间被栈满时才发生上溢。其存储数据的时间复杂度均为O(1)，所以对存取效率没有什么影响。

栈的链式存储结构

采用链式存储的栈称为链栈，链栈的优点是便于多个栈共享存储空间和提高其效率，且不存在栈满上溢的情况。通常采用单链表实现，并规定所有操作都是在单链表的表头进行的。这里规定链栈没有头结点，Lhead指向栈顶元素。

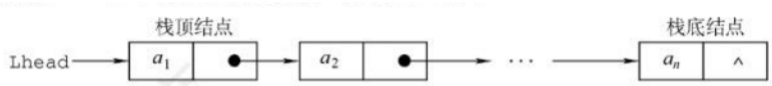


图 3.4　栈的链式存储

栈的链式存储类型可描述为

```
typedef struct Linknode{
    ElemType data;//数据域
    struct Linknode *next;//指针域
}LiStack;//栈类型定义
```

采用链式存储，便于结点的插入与删除。链栈的操作与链表类似，入栈和出栈的操作都在链表的表头进行。需要注意的是，对于带头结点和不带头结点的链栈，具体的实现会有所不同。