第3章　栈和队列

3.1 栈

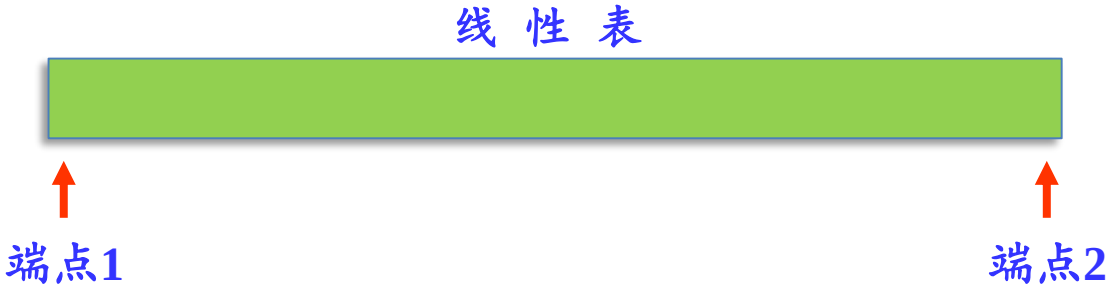
3.2 队列

3.3 栈和队列求解迷宫问题

3.1 栈

3.1.1 栈的定义

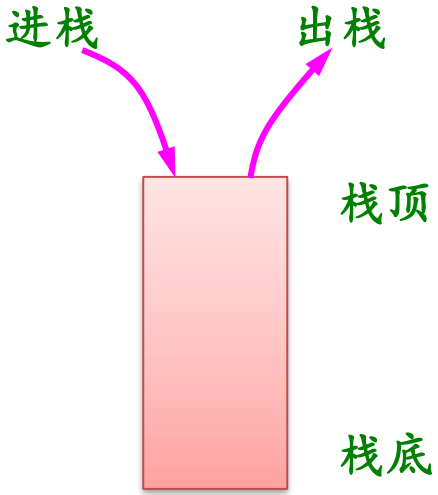
栈是一种只能在一端进行插入或删除操作的线性表。



栈只能选取同一个端点进行插入和删除操作

栈的几个概念

- 允许进行插入、删除操作的一端称为栈顶。
- 表的另一端称为栈底。
- 当栈中没有数据元素时，称为空栈。
- 栈的插入操作通常称为进栈或入栈。
- 栈的删除操作通常称为退栈或出栈。



栈示意图

栈的主要特点是“后进先出”，即后进栈的元素先出栈。栈也称为后进先出表。

例如：

假设死胡同的宽度恰好只够正一个人

走进死胡同的5人要按相反次序退出

死胡同就是一个栈！

思考题：

栈和线性表有什么不同？

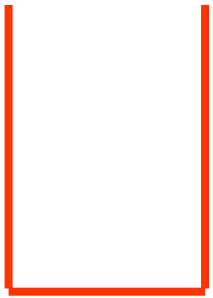
【例3-1】设一个栈的输入序列为*a,b,c,d*，则借助一个栈所得到的输出序列不可能是_____。

A. *c,d,b,a*　　B. *d,c,b,a*

C. *a,c,d,b*　　D. *d,a,b,c*

选项D是不可能的？

d　*c*　*b*　*a*



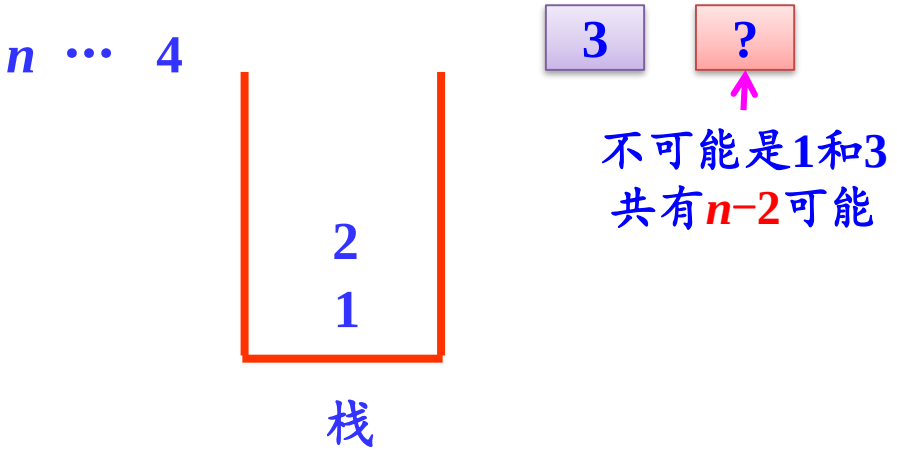
栈

下一步不可能出栈*a*

【例3-2】一个栈的入栈序列为1,2,3,…,n，其出栈序列是p_1, p_2,p_3, … ,p_n。若p_1=3，则p_2可能取值的个数是_____。

A.n−3　　B.n−2　　C.n−1　　D. 无法确定

1、2、3进栈， 3出栈的时刻：



栈抽象数据类型＝逻辑结构＋基本运算（运算描述）

栈的几种基本运算如下：

❶ InitStack(&s)：初始化栈。构造一个空栈s。

❷ DestroyStack(&s)：销毁栈。释放栈s占用的存储空间。

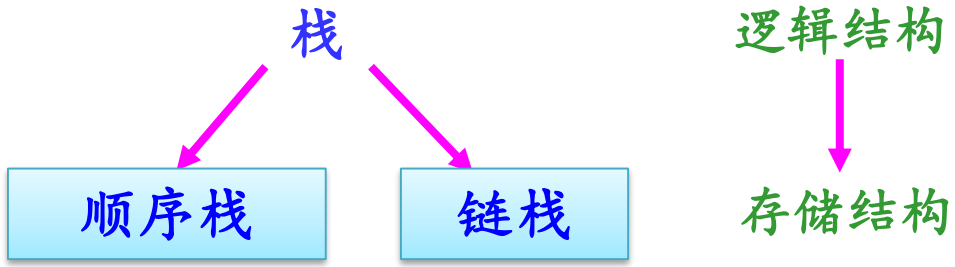
❸ StackEmpty(s)：判断栈是否为空:若栈s为空，则返回真；否则返回假。

❹ Push(&S,e)：进栈。将元素*e*插入到栈s中作为栈顶元素。

❺ Pop(&s,&e)：出栈。从栈s中退出栈顶元素，并将其值赋给*e*。

❻ GetTop(s,&e)：取栈顶元素。返回当前的栈顶元素，并将其值赋给*e*。

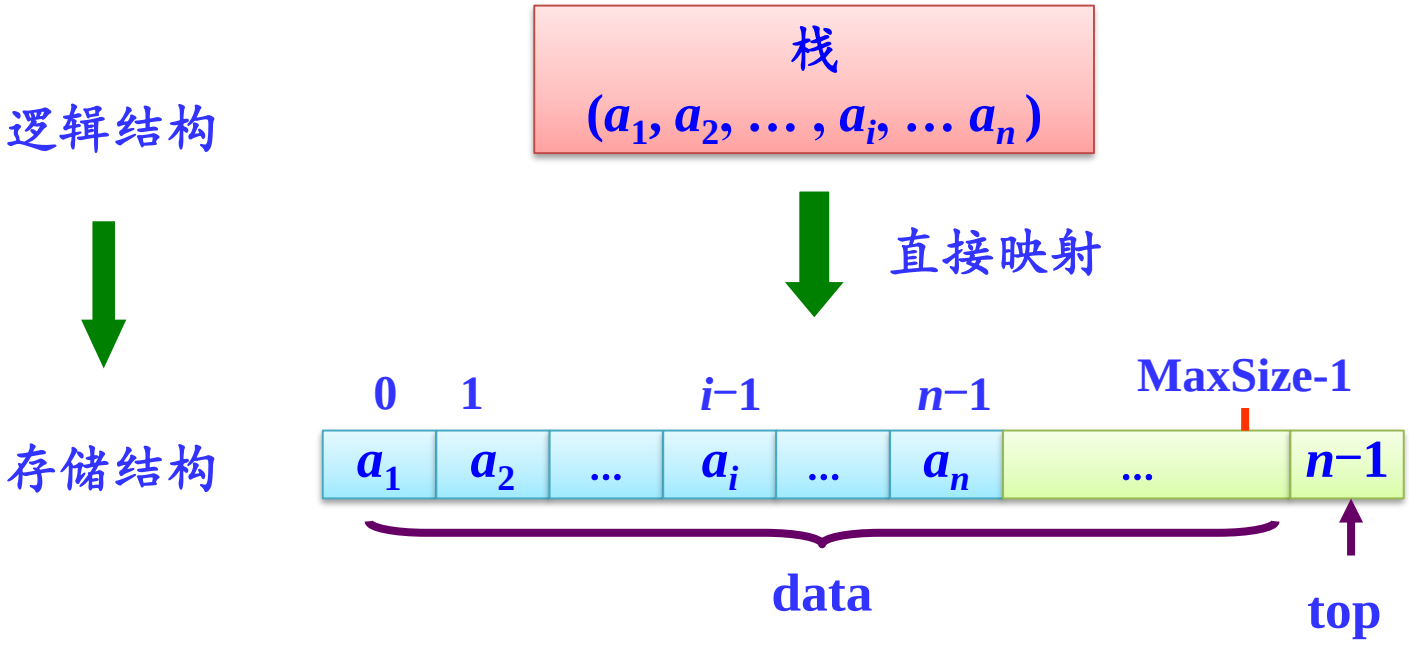
栈中元素逻辑关系与线性表的相同，栈可以采用与线性表相同的存储结构。



3.1.2 栈的顺序存储结构及其基本运算实现

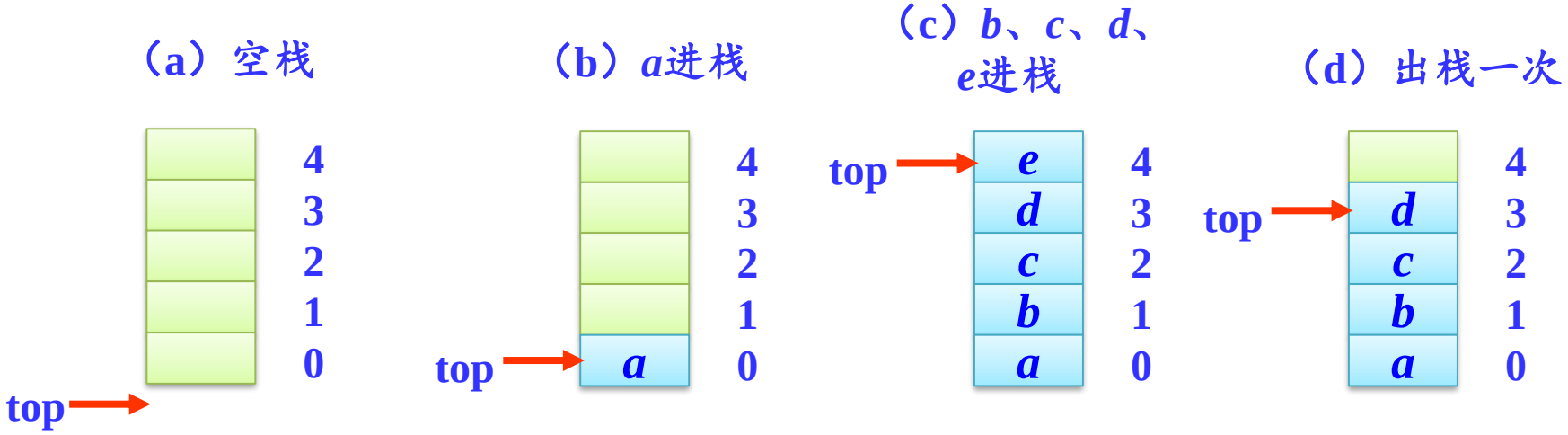
假设栈的元素个数最大不超过正整数MaxSize，所有的元素都具有同一数据类型ElemType，则可用下列方式来定义顺序栈类型SqStack:

```
typedef struct
{     ElemType data[MaxSize];
      int top;                    //栈顶指针
} SqStack;
```


栈
$(a_1, a_2, \ldots, a_i, \ldots a_n)$
逻辑结构
直接映射
存储结构
0
1
$i-1$
$n-1$
MaxSize-1
a_1
a_2
...
a_i
...
a_n
...
$n-1$
data
top

顺序栈的示意图

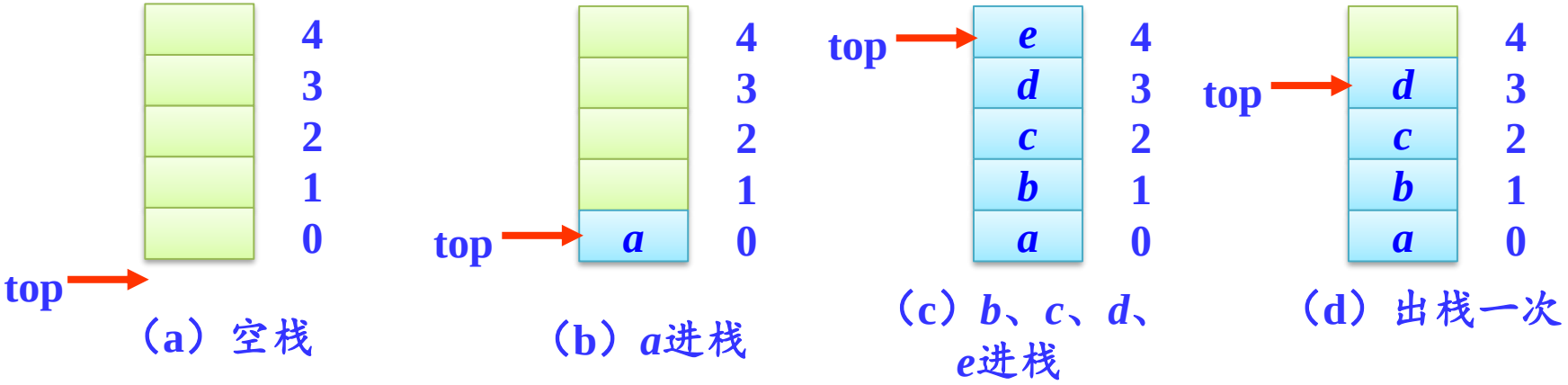
例如：MaxSize=5



总结：

- 约定top总是指向栈顶元素，初始值为-1
- 当top=MaxSize-1时不能再进栈—栈满
- 进栈时top增1，出栈时top减1

顺序栈的各种状态



（a）空栈　（b）*a*进栈　（c）*b*、*c*、*d*、*e*进栈　（d）出栈一次

顺序栈4要素：

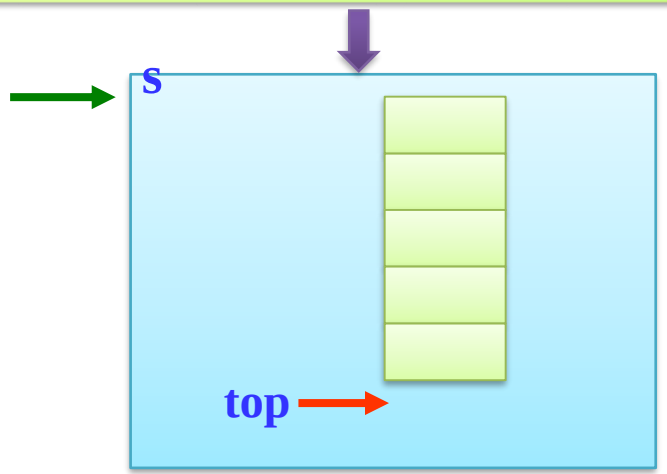
- 栈空条件：top=−1
- 栈满条件：top=MaxSize−1
- 进栈*e*操作：top++; 将*e*放在top处
- 退栈操作：从top处取出元素*e*; top−−;

在顺序栈中实现栈的基本运算算法。

（1）初始化栈InitStack(&s)

建立一个新的空栈s，实际上是将栈顶指针指向-1即可。

```
void InitStack(SqStack *&s)
{       s=(SqStack *)malloc(sizeof(SqStack));
        s->top=-1;
}
```



注意： s为栈指针，top为s所指栈的栈顶指针

（2）销毁栈DestroyStack(&s)

释放栈s占用的存储空间。

```
void  DestroyStack(SqStack *&s)
{
      free(s);
}
```

（3）判断栈是否为空StackEmpty(s)

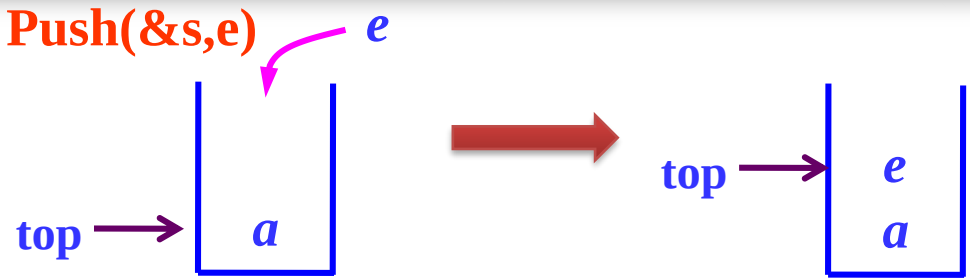
栈S为空的条件是s->top==-1。

```
bool StackEmpty(SqStack *s)
{
        return(s->top==-1);
}
```

（4）进栈Push(&s,e)

在栈不满的条件下，先将栈指针增1，然后在该位置上插入元素e。

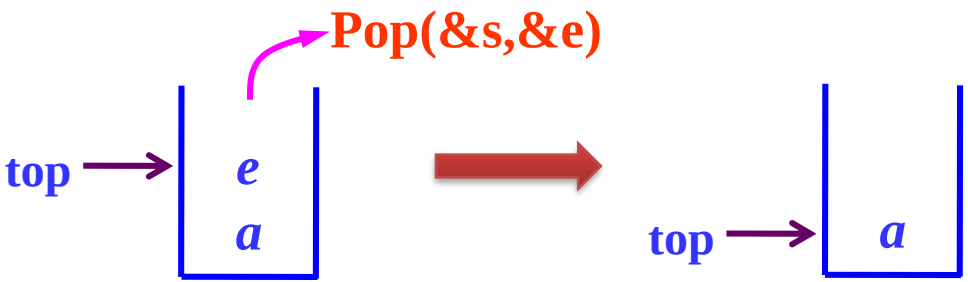
```
bool Push(SqStack *&s,ElemType e)
{     if (s->top==MaxSize-1)              //栈满的情况，即栈上溢出
            return false;
      s->top++;                           //栈顶指针增1
      s->data[s->top]=e;                  //元素e放在栈顶指针处
      return true;
}
```



（5）出栈Pop(&s,&e)

在栈不为空的条件下，先将栈顶元素赋给*e*，然后将栈指针减1。

```
bool Pop(SqStack *&s,ElemType &e)
{   if (s->top==-1)              //栈为空的情况，即栈下溢出
        return false;
    e=s->data[s->top];           //取栈顶指针元素的元素
    s->top--;                    //栈顶指针减1
    return true;
}
```



（6）取栈顶元素GetTop(s,&e)

在栈不为空的条件下，将栈顶元素赋给e。

```
bool GetTop(SqStack *s,ElemType &e)
{
    if (s->top==-1)            //栈为空的情况，即栈下溢出
        return false;
    e=s->data[s->top];         //取栈顶指针元素的元素
    return true;
}
```

【例3-3】 设计一个算法利用顺序栈判断一个字符串是否是对称串。所谓对称串是指从左向右读和从右向左读的序列相同。

算法设计思路

字符串str的所有元素依次进栈，产生的出栈序列正好与str的顺序相同 ⇨ str是对称串。

```
bool symmetry(ElemType str[])
{   int i;  ElemType e;   SqStack *st;
    InitStack(st);                          //初始化栈

    for (i=0;str[i]!='\0';i++)              //将串所有元素进栈
        Push(st,str[i]);                    //元素进栈

    for (i=0;str[i]!='\0';i++)
    {   Pop(st,e);                          //退栈元素e
        if (str[i]!=e)                      //若e与当前串元素不同则不是对称串
        {   DestroyStack(st);               //销毁栈
            return false;
        }
    }

    DestroyStack(st);                       //销毁栈
    return true;
}
```

str的所有元素依次进栈

判断正反序是否相同

——本讲完——