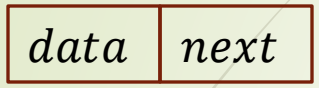
数据结构与算法

人工智能与大数据学院

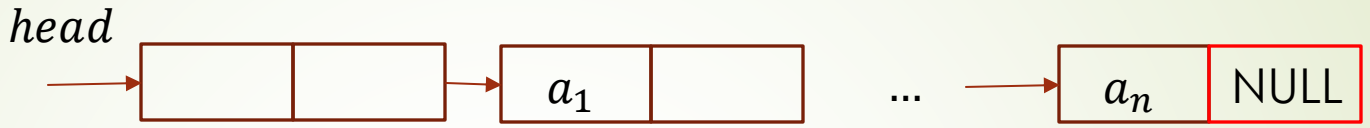
本节课主要内容

- 循环链表和双向链表
- 线性表的应用示例
- 特殊线性表——栈

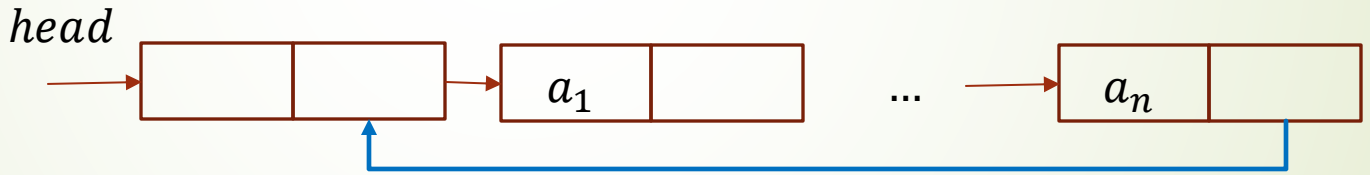
第2章 线性表——循环链表



单链表的结点结构



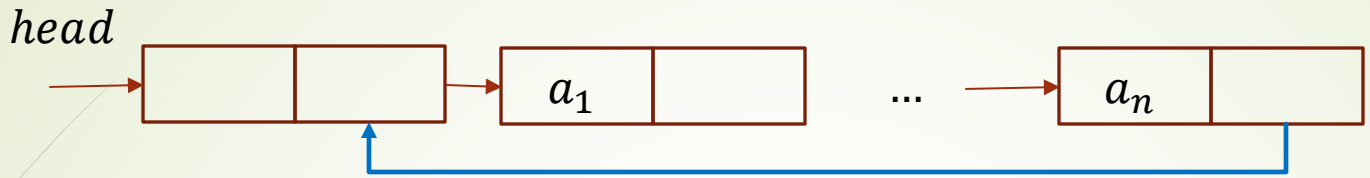
带头结点的单链表逻辑结构



带头结点的**单循环链表**逻辑结构

循环链表是一种首尾相接的链表。将单链表中最后一个结点的指针指向头结点，整个链表形成一个环，这种链式存储结构称为单循环链表。

第2章 线性表——循环链表



带头结点的**单循环链表**逻辑结构

单循环链表的特点：从表中任意结点出发均可以找到表中其他的结点。

对于循环查找/插入/删除等问题，更适合在循环链表上实现。（举例）

第2章 线性表——循环链表

例如：约瑟夫环问题（经典的循环链表问题）

题目：已知 n 个人（以编号1，2，3，…，n分别表示）围坐在一张圆桌周围，从编号为 k 的人开始顺时针报数，数到 m 的那个人出列；他的下一个人又从 1 还是顺时针开始报数，数到 m 的那个人又出列；依次重复下去，要求找到最后出列的那个人的编号。

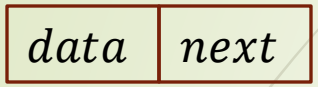
第2章 线性表——循环链表的基本操作

单循环链表上的操作实现和单链表上基本一致，但是需要修改算法中的循环条件。

```
//在单链表中查找值为x的结点
Lnode *search(Lnode *h, int x){
    Lnode *p;
    p = h->next;
    while(p && p->data!=x)
        p=p->next;
    return p;
}
```

```
//在单循环链表中查找值为x的结点
Lnode *get(Lnode *h, int x){
    Lnode *p;
    p = h->next;
    while(p!=h && p->data!=x)
        p=p->next;
    if(p==h)
        return NULL;
    return p;
}
```

第2章 线性结构——双向链表



单链表的结点结构

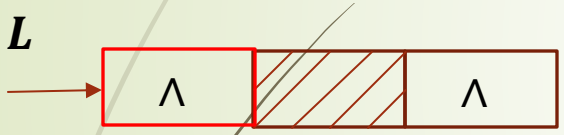
(b) 非空表

单链表的结点中只有一个指向其后继结点的指针域next，对于求指定结点的后继结点的需求，时间复杂度仅为O(1)；但是如果求前驱只能从链表的头指针开始，顺着各结点的next域进行，时间复杂度为O(n)

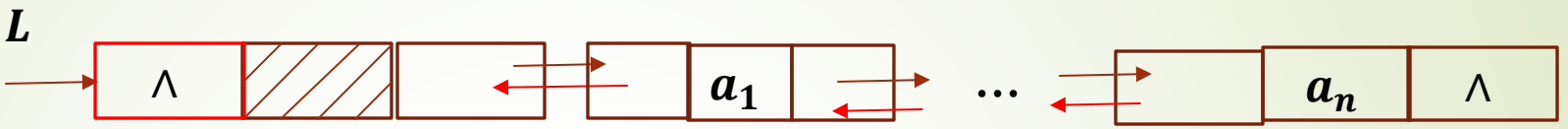
循环单链表虽然可以从表中任意结点找到其他结点，但找前驱的操作还是需要遍历整个链表，时间复杂度为O(n)

第2章 线性结构——双向链表

(a) 双向链表的结点结构



(b) 空表



(c) 非空表

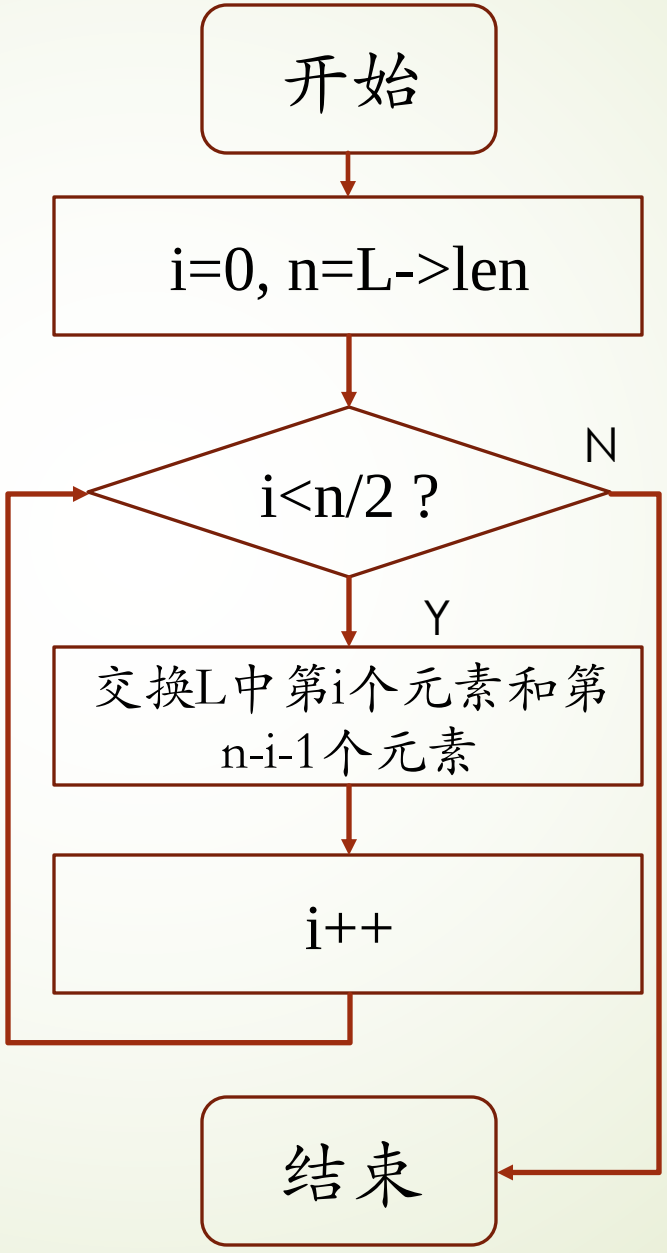
双向链表在数据结构设计方面为每个结点新加一个指向直接前驱的指针域。
与单链表相比，每个结点多占用一个指针所需的空间，但是找直接前驱的时间复杂度为O(1)，是一种用空间换时间的处理策略。

第2章 线性结构——线性表的应用示例

例1：编写一个算法将一个顺序表原地逆置（不允许新建一个顺序表）

第2章 线性结构——线性表的应用示例

例1：编写一个算法将一个顺序表原地逆置（不允许新建顺序表）



第2章 线性结构——线性表的应用示例

例1：编写一个算法将一个顺序表原地逆置（不允许新建顺序表）

```
Void inverse(sqlist * L){
    elemtype t;
    int n = L->len;
    for(int i=0; i<=(n-1)/2;i++)
    {
        t = L->data[i];
        L->data[i] = L->data[n-i-1];
        L->data[n-i-1]=t;
    }
}
```

第2章 线性结构——线性表的应用示例

例2：编写一个算法将一个带头结点的单链表逆置（不允许新建单链表）

第2章 线性结构——线性表的应用示例

例2：编写一个算法将一个带头结点的单链表逆置（不允许新建单链表）

带头结点的单链表逻辑结构

实现思路：

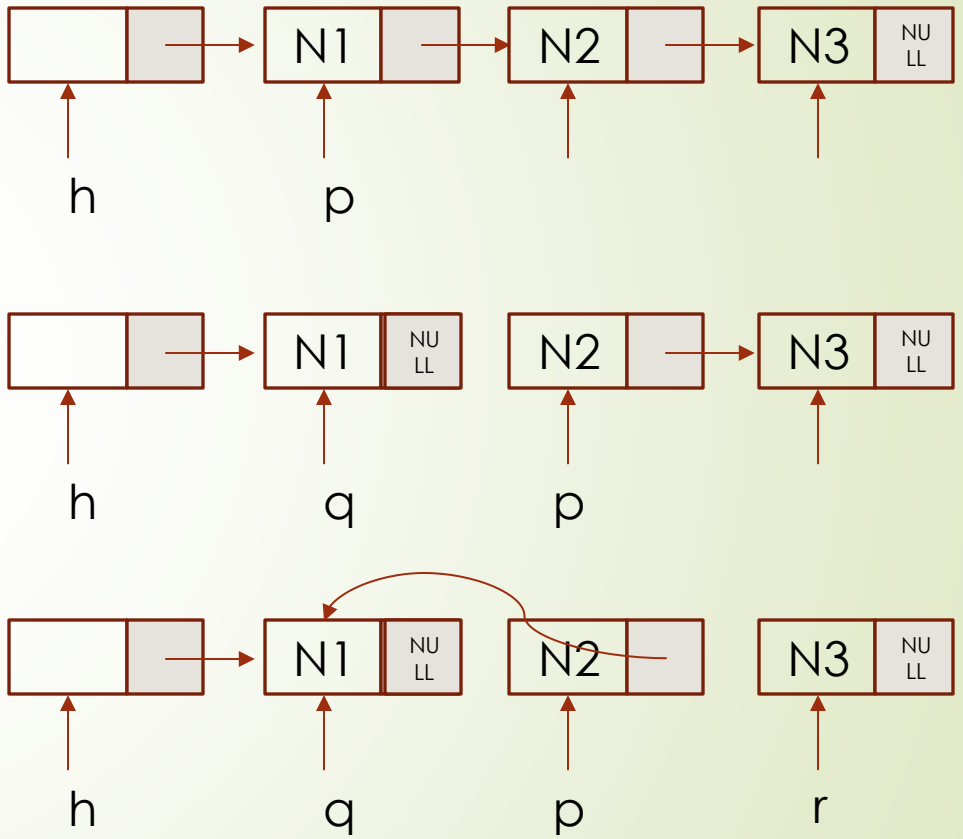
① 如果链表为空，不需要操作

② 如果单链表的长度为1，也不需要操作

③ 将首结点变为尾结点，逆转指针

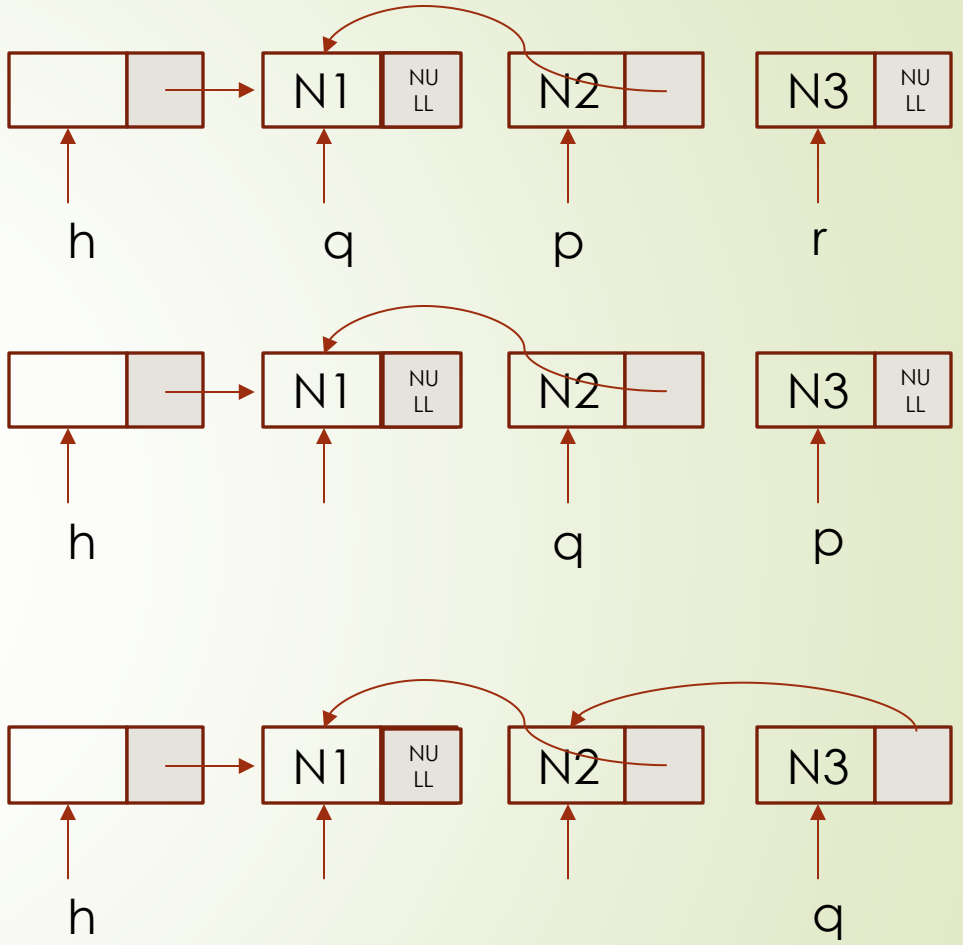
例2：编写一个算法将一个带头结点的单链表逆置（不允许新建单链表）

```
void  inverse(Lnode * h){
    Lnode *r, *q, *p;
    p = h->next;
    if(p==NULL) return 0;
    else if(p->next==NULL) return 0;
    q = p;
    p = p->next;
    q->next = NULL;
    while(p)
    {
        r=p->next;
        p->next=q;
        q=p;
        p=r;
    }
    h->next=q;
}
```



例2：编写一个算法将一个带头结点的单链表逆置（不允许新建单链表）

```
void  inverse(Lnode * h){
    Lnode *r, *q, *p;
    p = h->next;
    if(p==NULL) return 0;
    else if(p->next==NULL) return 0;
    q = p;
    p = p->next;
    q->next = NULL;
    while(p)
    {
        r=p->next;
        p->next=q;
        q=p;
        p=r;
    }
    h->next=q;
}
```



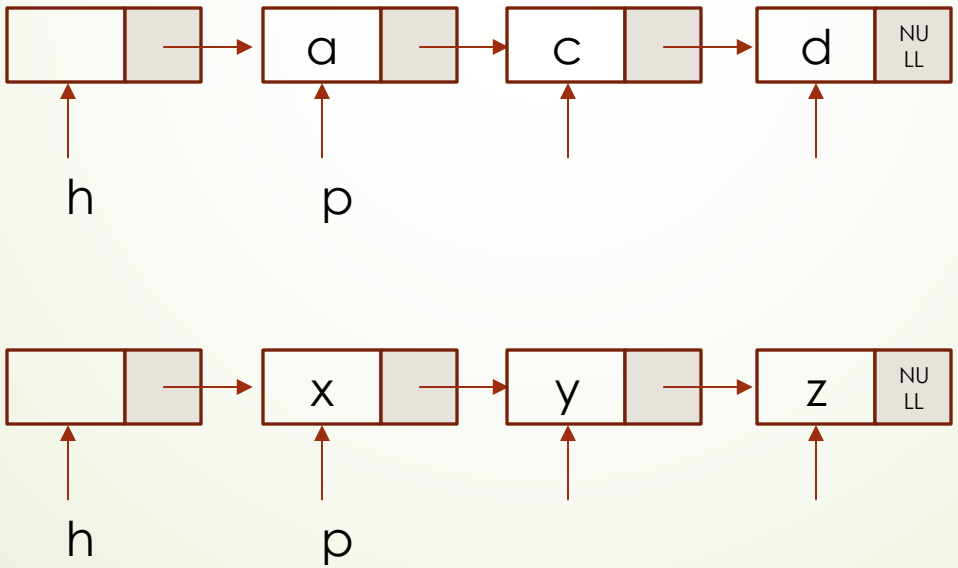
第2章 线性结构——线性表的应用示例

例3：编写一个算法，在头结点为h的单链表中，把值为b的结点插入到值为a的结点之前，若不存在a，则把结点s插入到表尾。（学习通作业）

验证：将学号采用单链表存储，在每个值为8的结点前均插入新结点（值为7）

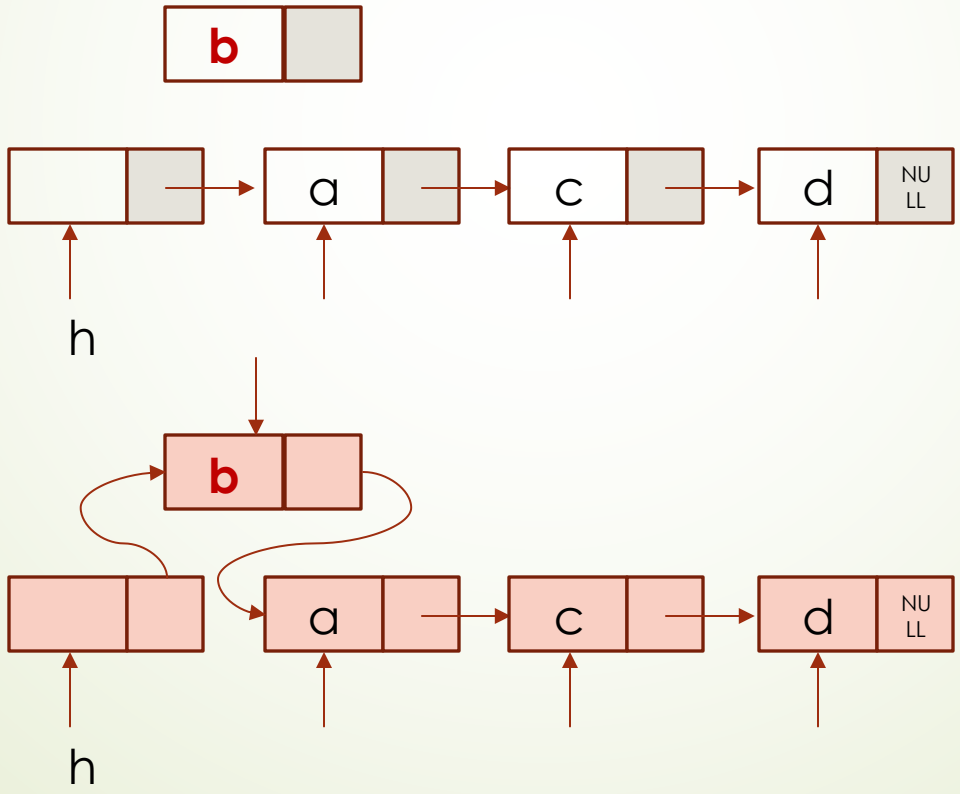
第2章 线性结构——线性表的应用示例

例3：编写一个算法，在头结点为h的单链表中，把值为b的结点插入到值为a的结点之前，若不存在a，则把结点s插入到表尾



第2章 线性结构——线性表的应用示例

例3：编写一个算法，在头结点为h的单链表中，把值为b的结点插入到值为a的结点之前，若不存在a，则把结点s插入到表尾

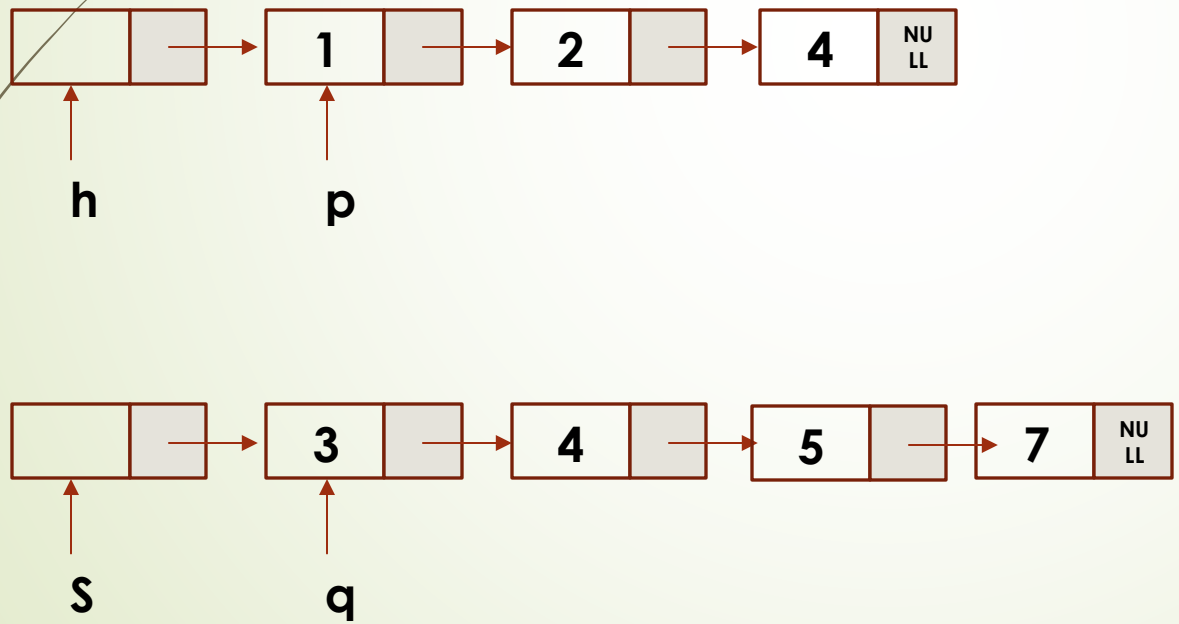


第2章 线性结构——线性表的应用示例

例4：编写一个算法将两个按元素值递增有序排列的顺序表归并成一个按元素值递增有序排列的顺序表

第2章 线性结构——线性表的应用示例

例5：编写一个算法将两个按元素值递增有序排列的单链表归并成一个按元素值递增有序排列的单链表



第3章 栈和队列

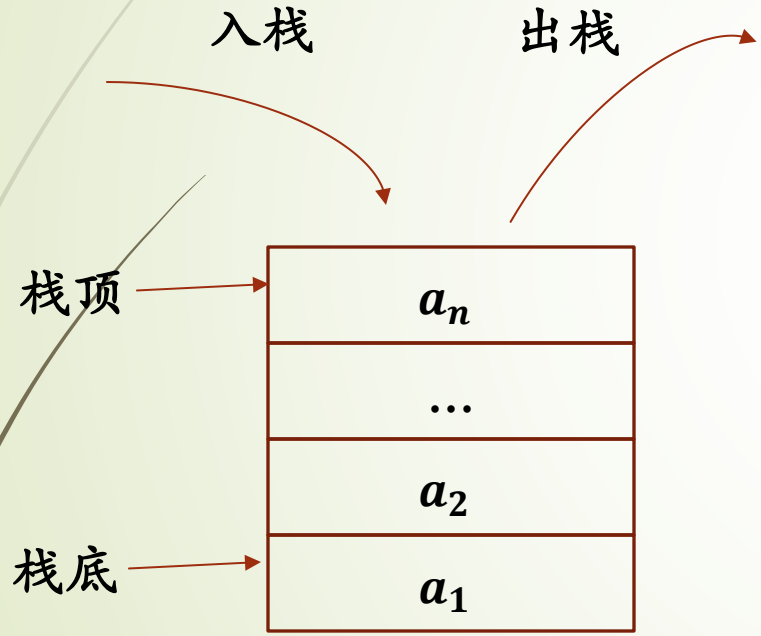
1、栈的定义

栈是一种特殊的线性表，它的逻辑结构和存储结构与线性表相同，它的特殊性在于所有操作都只能在线性表的一端进行。

进行操作的一端称为**栈顶**，用一个“栈顶指针”指示，栈顶的位置经常发生变化；而另一端是固定端，称为**栈底**。当栈中没有元素时称为空栈。向栈中插入数据元素的操作称为**入栈**，从栈中删除数据元素的操作称为**出栈**。

第3章 栈和队列

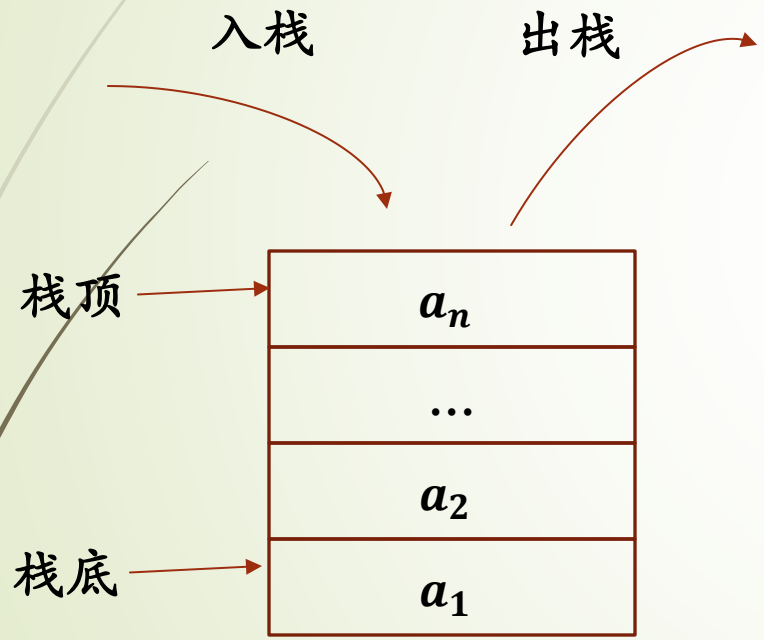
1、栈的定义



由于栈的操作都在栈顶进行，如果数据元素$a_1, a_2, \ldots, a_n$依次入栈，当a_n入栈后，其他元素都被“压”在下面。因此出栈的顺序只能是a_n，…，a_2，a_1。这表明最先入栈的数据元素在出栈时却排在最后，这个特性称为“先进后出”，也可以称为“后进先出”

第3章 栈和队列

1、栈的定义



假设一个栈的入栈序列为{a,b,c,d,e}，那么下列哪个是不可能输出的序列?

a. edcba
b. decba
c. dceab
d. abcde

2、栈的基本操作

基本操作	函数名称	操作结果
初始化	Status Stack_Init(StackPtr s)	若成功，返回success，构造一个s所指向的空栈，否则返回fatal
销毁	Void Stack_Destory(StackPtr s)	释放s所占空间，栈s不存在
清空	Void Stack_Clear(StackPtr s)	清空栈中所有元素，栈s变空
判空	Bool Stack_Empty(StackPtr s)	若栈s为空，返回true，否则返回false
判满	Bool Stack_Full(StackPtr s)	若栈s为满，返回true，否则返回false
入栈	Status Stack_Push(StackPtr s, StackEntry item)	若栈s不满，将item添加到栈顶，返回true，否则返回overflow
出栈	Status Stack_Pop(StackPtr s, StackEntry *item)	若栈s不空，将栈顶数据放入item，并删除原栈顶，返回true，否则返回underflow
取栈顶元素	Status Stack_Top(StackPtr s, StackEntry *item)	若栈s不空，将栈顶数据放入item，返回true，否则返回underflow

3、栈的顺序存储结构

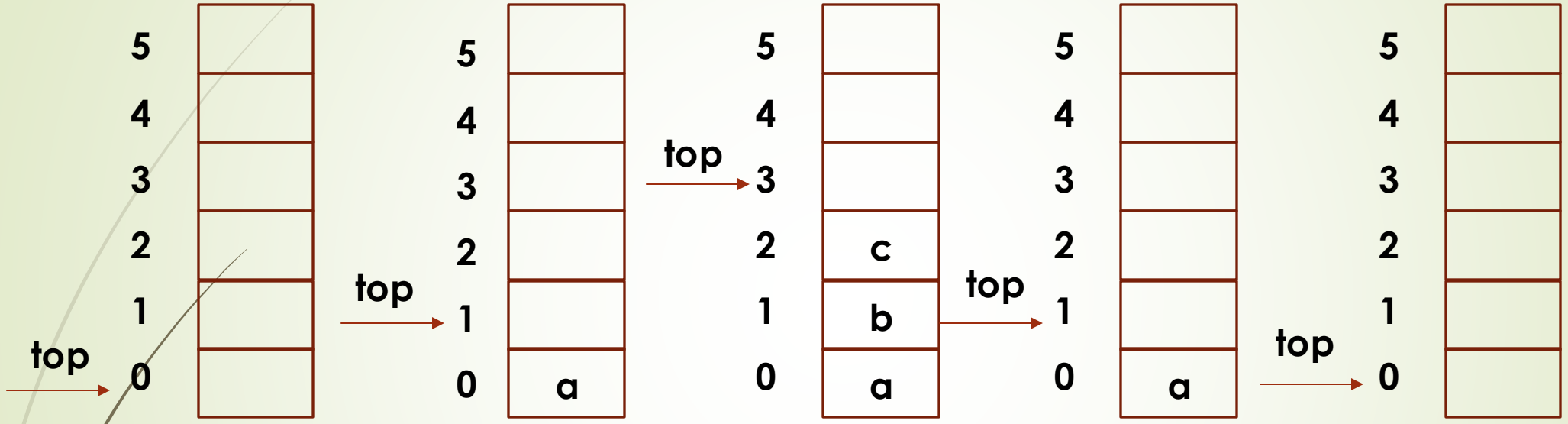
栈的顺序存储结构简称顺序栈。

顺序栈是利用一组地址连续的存储单元依次存放从栈底到栈顶的数据元素，通常是一维数组存放栈的元素，同时设“指针”top指示栈顶元素的当前位置。（空栈时top= 0）

```
#define maxsize 100
typedef struct{
   int elem[maxsize];
   int top;
}sqstacktp;
```

3、栈的顺序存储

顺序栈中入栈和出栈操作时栈顶指针的变化情况



(a) 空栈　(b) a入栈　(c) b、c入栈　(d) c、b出栈　(e) a出栈

由于顺序栈中数据元素空间的大小是预先分配的，当空间全部占满后再做入栈操作将产生溢出，称为“上溢”；类似地，当栈为空时再做出栈操作也将产生溢出，称为“下溢”

3、顺序栈实现的操作

（1）栈置空操作

因为top=0表示空栈，因此在执行出栈和读栈顶元素之前应判断栈是否为空

```
typedef struct{
    int elem[maxsize];
    int top;
}sqstacktp;


void initstack(sqstacktp *s){
    s->top = 0;
}


int main(){
    sqstacktp *s;
    s = (sqstacktp *)malloc(sizeof(sqstacktp));
    initstack(s);
    return 0;
}
```

3、顺序栈实现的操作

(2) 判空操作

```
int stackempty1(sqstacktp *s){
    if(s->top == 0)
        return 1;
    else
        return 0;
}


int stackempty2(sqstacktp *s){
    return (s->top < 1);
}
```

（3）入栈操作和出栈操作

```
void push(sqstacktp *s, int x){
    if(s->top == maxsize)
        printf("溢出\n");
    else{
        s->elem[s->top] = x;
        s->top ++;
    }
}
```

```
int pop(sqstacktp *s){
        int x;
    if(s->top == 0)
        return NULL;
    else{
                s->top--;
            x = s->elem[s->top];
        return x
    }
}
```

3、顺序栈实现的操作

（5）求栈深（栈中元素个数）操作

```
int size(sqstacktp *s){
    return (s->top);
}
```

（6）读取栈顶元素操作

```
int top(sqstacktp *s){
    if(s->top==0)
        return NULL;
    else
        return (s->elem[s->top-1]);
}
```

顺序栈：

从栈的顺序存储结构可知，顺序栈的最大缺点是：为了保证不溢出，通常预先为栈分配一个比较大的空间，很有可能造成存储空间的浪费，而且在很多时候并不能保证分配的空间足够使用。该缺点大大降低了顺序栈的可用性。

习题

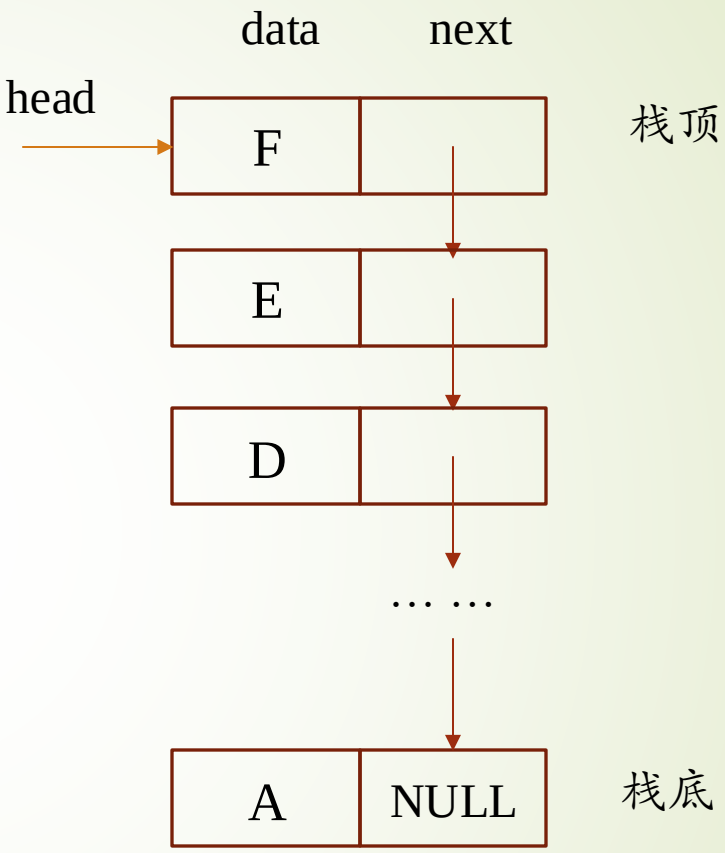
（1）设整数a，b，c，d，若执行以下操作序列，请思考出栈的序列是怎样的？push(a), pop(), push(b), push(c), pop(), push(d), pop(), pop()

（2）设整数a,b,c依次入栈，在这三个数的所有排列中，哪些序列是可以通过相应的入、出栈得到的？

4、栈的链式存储

栈的链式存储，简称链栈

它的组织形式与单链表类似，链表的尾部结点是栈底，链表的头部结点是栈顶。不过由于只在链栈的头部进行操作，没有必要设置头结点。




```
typedef struct stacknode{
    int data;
    struct stacknode *next;
}stacknode;


typedef struct{
    stacknode * top;
}LinkStack;
```

4、栈的链式存储

链栈的基本操作——初始化

```
typedef struct stacknode{
    int data;
    struct stacknode *next;
}stacknode;


typedef struct{
    stacknode * top;
}LinkStack;
```

```
void InitStack(LinkStack *ls){
    ls->top = NULL;
}
```

4、栈的链式存储

链栈的基本操作——进栈/入栈

- 生成新结点s
- 新结点s的数据域为待入栈数据元素值
- 链入新结点s
- 修改栈顶指针top

```
void Push(LinkStack *ls, int x){
    stacknode *s = NULL;
    s = (stacknode *)malloc(sizeof(stacknode));
    s ->data = x;
    s ->next = ls->top;
    ls ->top = s;
}
```

4、栈的链式存储

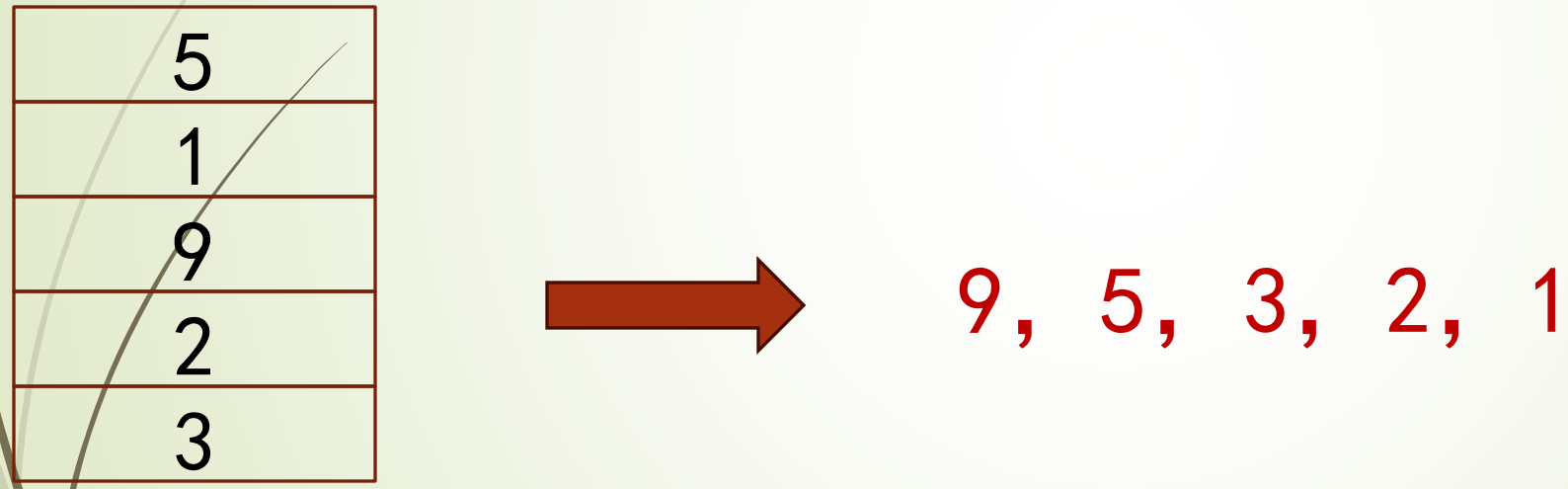
链栈的基本操作——出栈

- 判断是否是空栈
- 取栈顶元素
- 删除栈顶元素

```
int Pop(LinkStack *ls){
    stacknode *p = NULL;
    int x;
    if(ls->top == NULL)
        return NULL;
    else{
        x = (ls->top)->data;
        p = ls->top;
        ls->top = p->next;
        free(p);
        return x;
    }
}
```

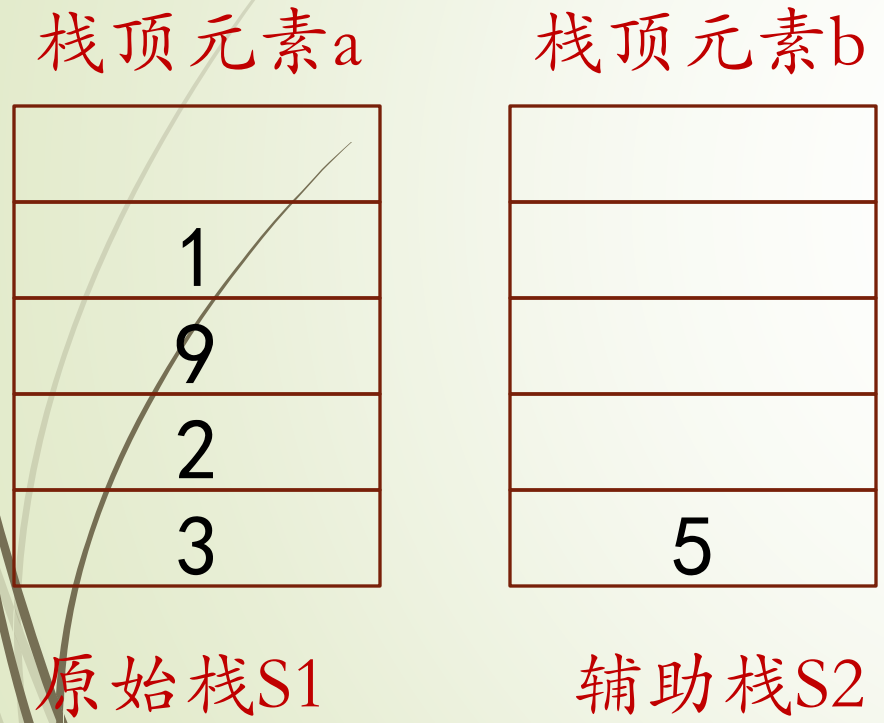
顺序栈

例1：利用栈输出降序序列



顺序栈

例1：利用栈输出降序排列（借助辅助栈）



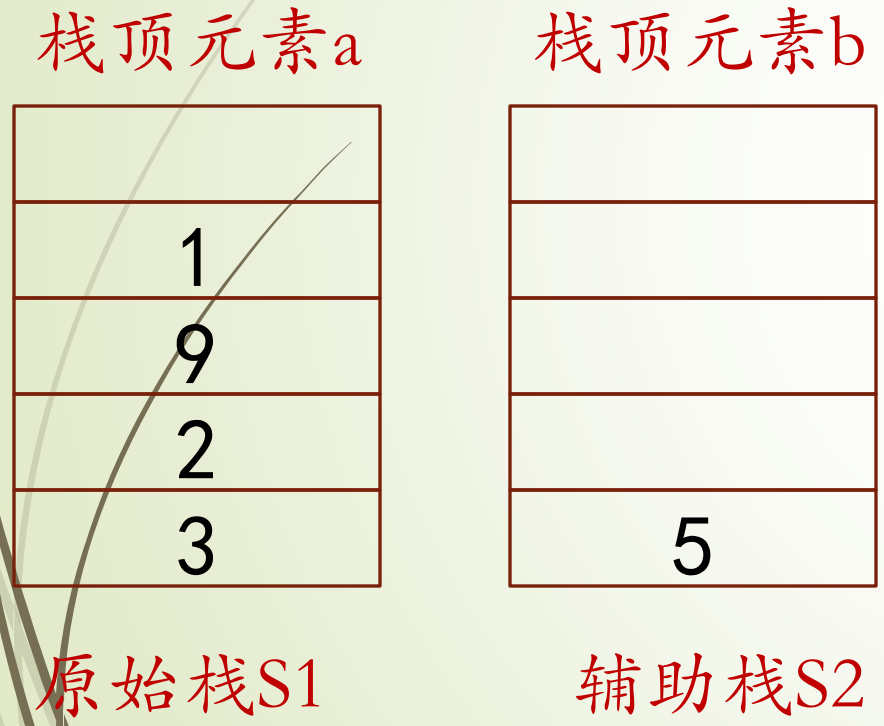
比较S1和S2的栈顶元素a（已出栈）和b，

(1)如果a<b，则将a压入S2；

(2)如果a>b，则将b弹出，压入S1，继续a和b的比较（重复执行）

顺序栈

例1：利用栈输出降序排列（借助辅助栈）

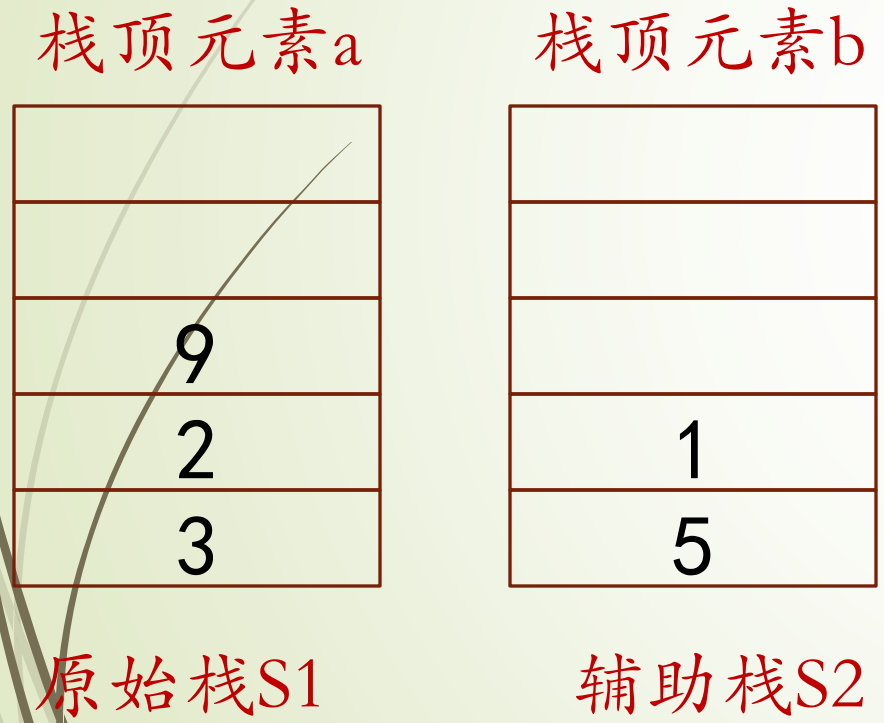


原始栈S1中的1出栈，1<5，

则将元素1压入S2

顺序栈

例1：利用栈输出降序排列（借助辅助栈）

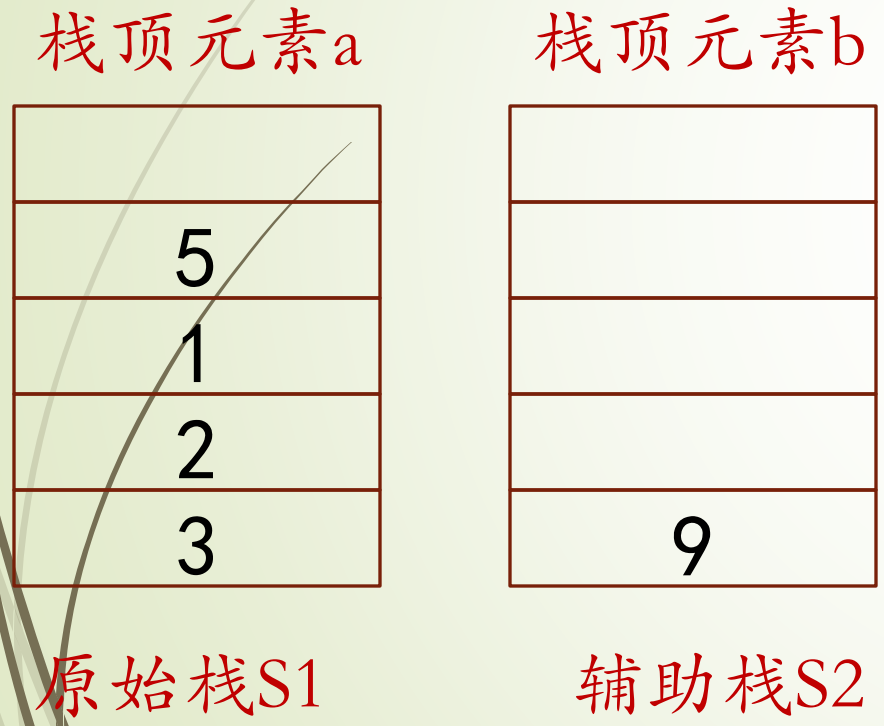


原始栈S1中的9出栈，9>1，

则将元素1出栈压入S1，比较9和5，9>5，
将元素5出栈压入S1

顺序栈

例1：利用栈输出降序排列（借助辅助栈）



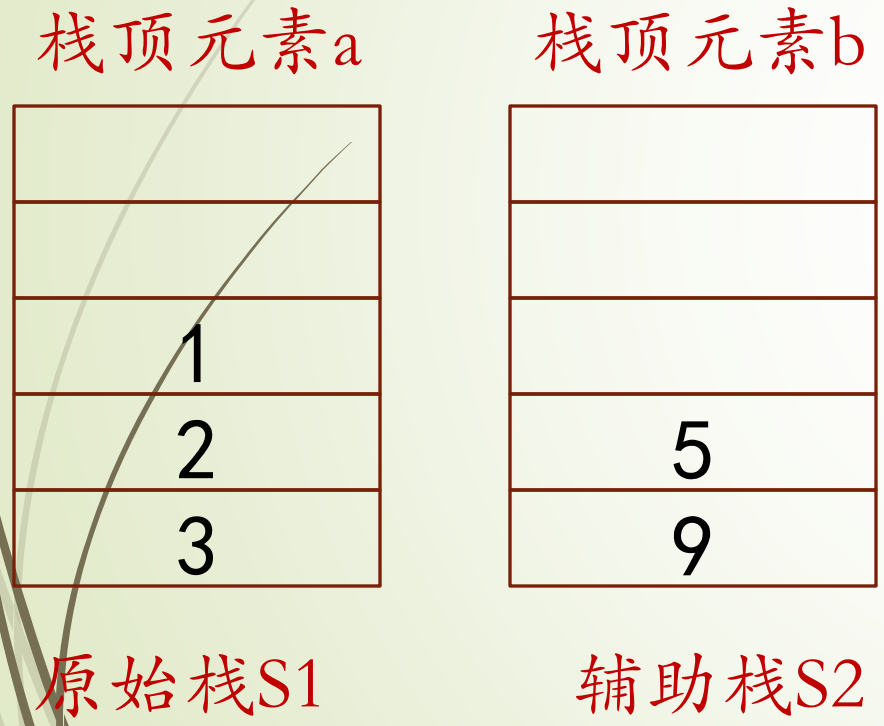
原始栈S1中的5出栈，5>1，

则将元素1出栈压入S1，比较9和5，9>5，
将元素5出栈压入S1

将元素9压入S2

顺序栈

例1：利用栈输出降序排列（借助辅助栈）

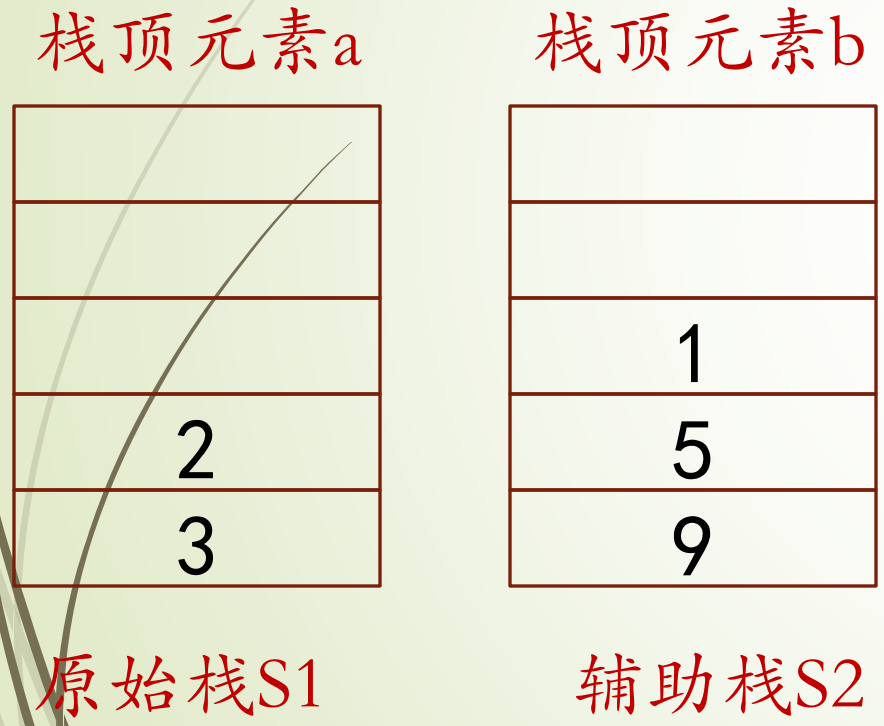


原始栈S1中的5出栈，5<9，

则将元素5压入S1

顺序栈

例1：利用栈输出降序排列（借助辅助栈）

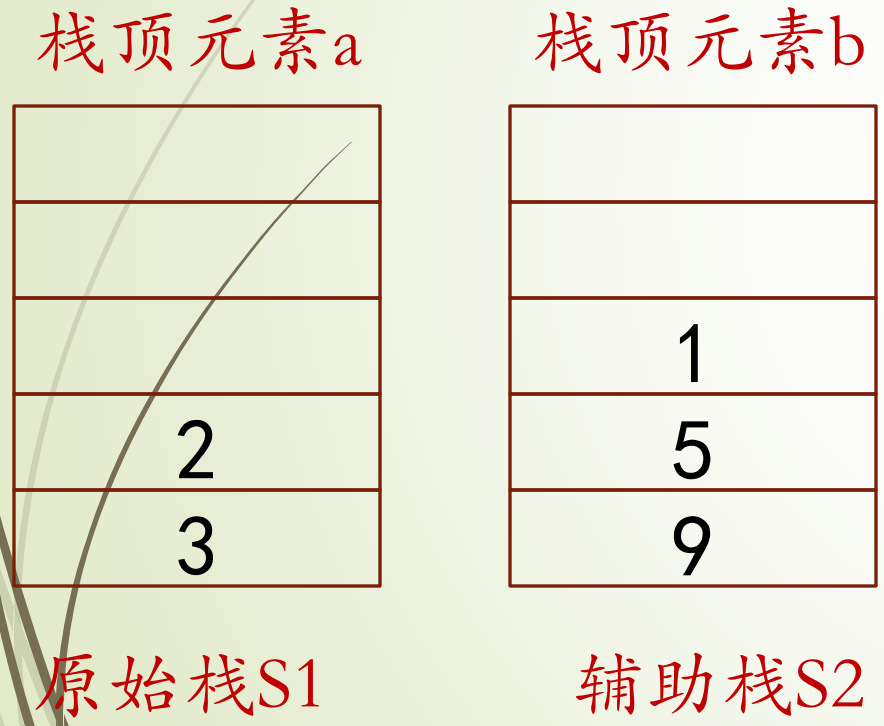


原始栈S1中的1出栈，1<5，

则将元素1压入S1

顺序栈

例1：利用栈输出降序排列（借助辅助栈）

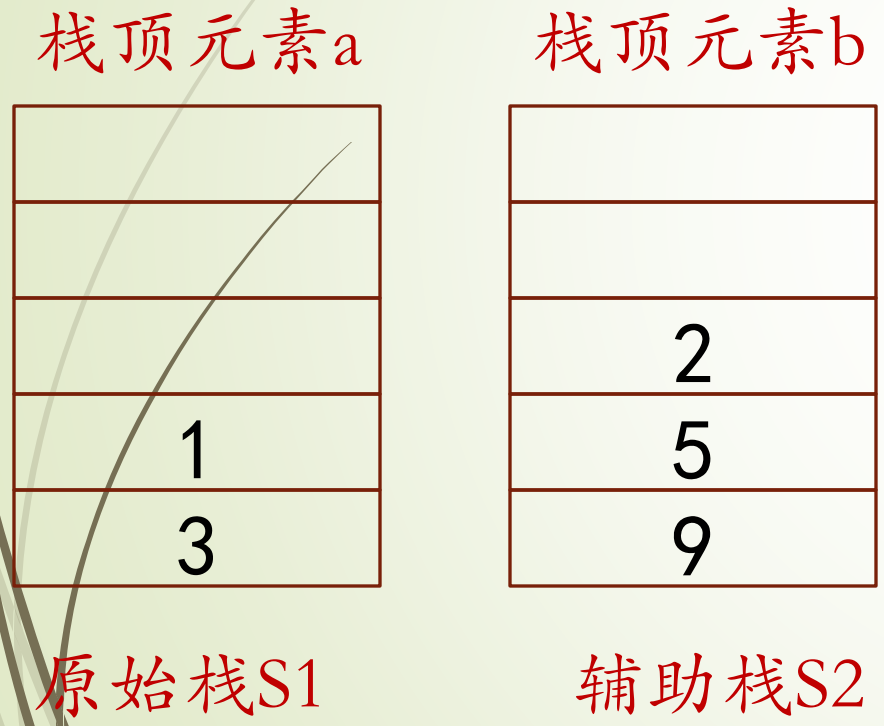


原始栈S1中的2出栈，2>1，

则将元素1压入S1，2与5比较，2<5，
将元素2压入S2

顺序栈

例1：利用栈输出降序排列（借助辅助栈）

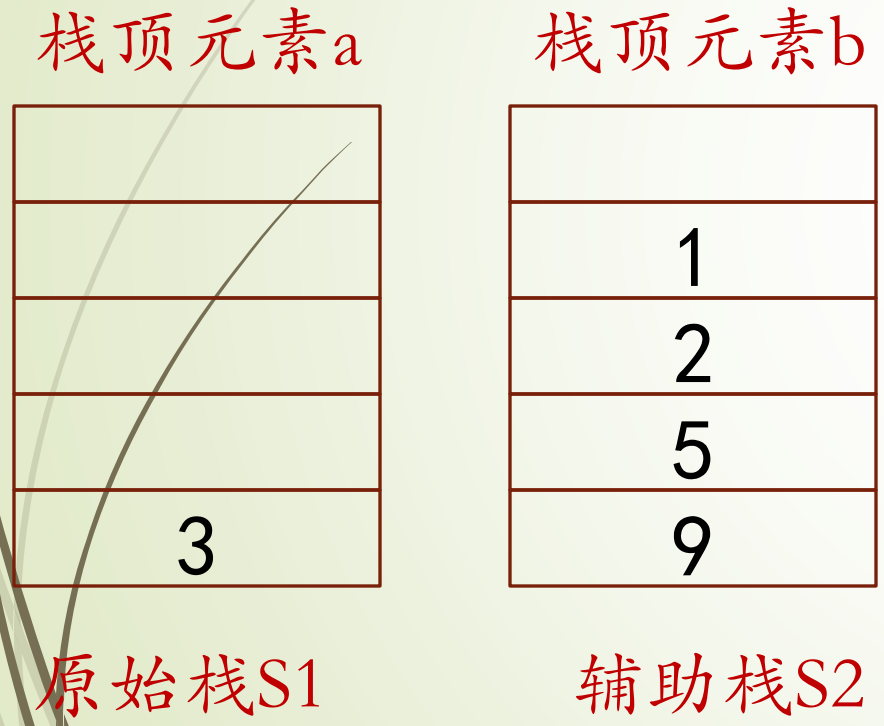


原始栈S1中的1出栈，1<2，

则将元素1压入S2

顺序栈

例1：利用栈输出降序排列（借助辅助栈）

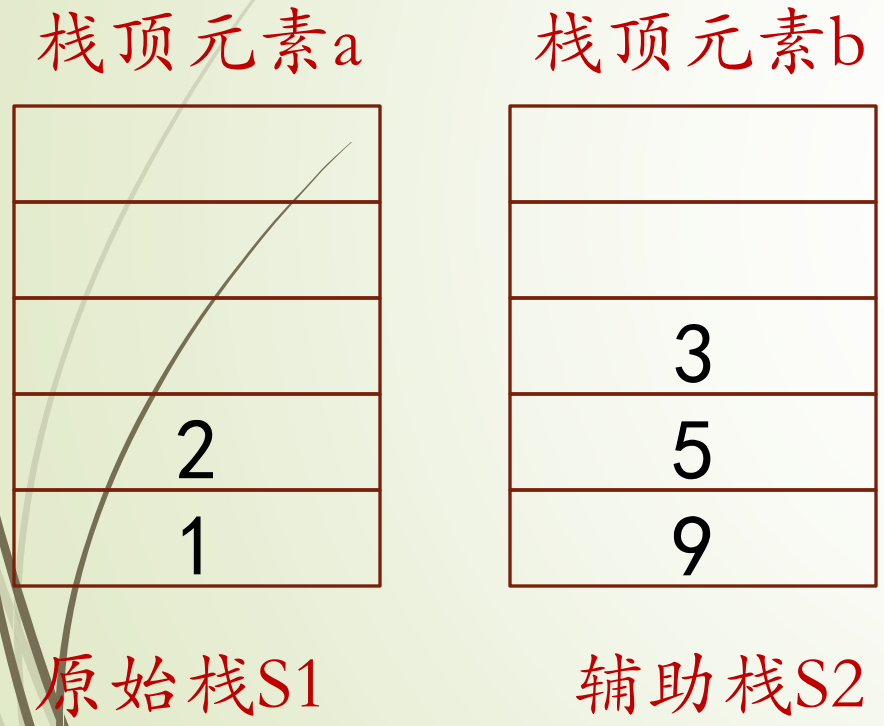


原始栈S1中的3出栈，3>1，

则将元素1出栈压入S1，3和2比较，3>2，将元素2出栈压入S1，3和5比较，3<5，将3压入S2

顺序栈

例1：利用栈输出降序排列（借助辅助栈）

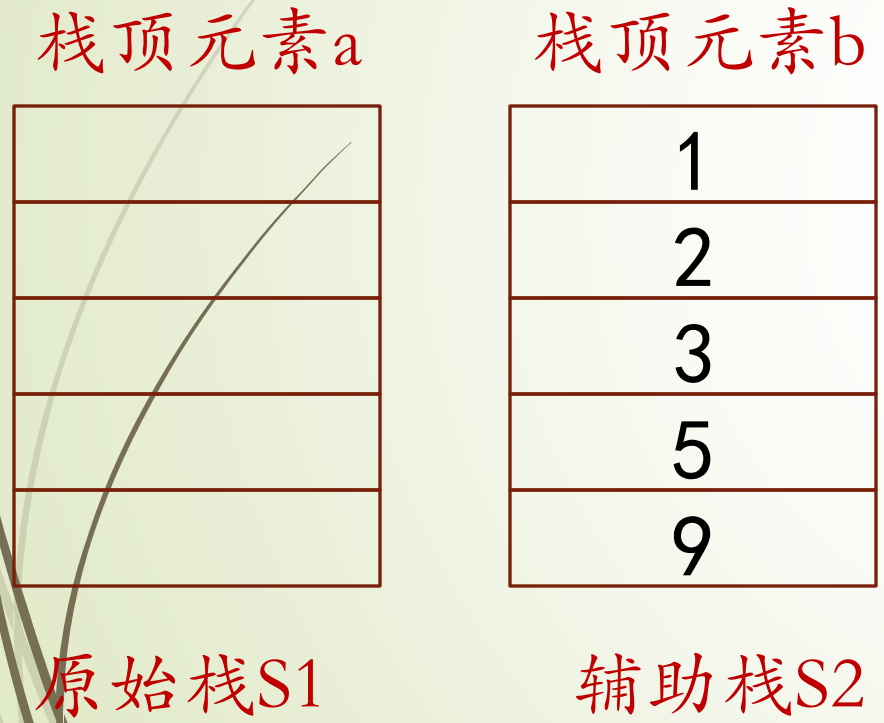


原始栈S1中的3出栈，3>1，

则将元素1出栈压入S1，3和2比较，3>2，将元素2出栈压入S1，3和5比较，3<5，将3压入S2

顺序栈

例1：利用栈输出降序排列（借助辅助栈）

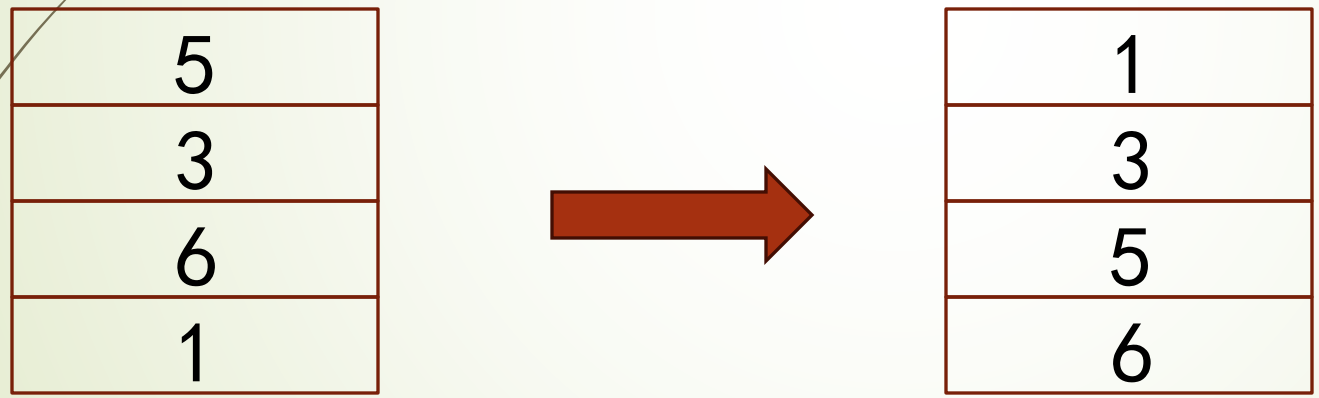


将S2中的元素依次出栈，依次入栈S1

从S1栈中出栈的序列为：9,5,3,2,1

顺序栈

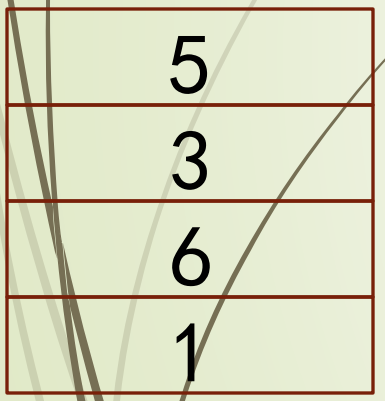
例1：实现将一个栈进行降序排列（栈顶元素最小）



顺序栈

例：实现将一个栈进行降序排列（栈顶元素最小）

思路1: 借助一个辅助栈



原始栈S1

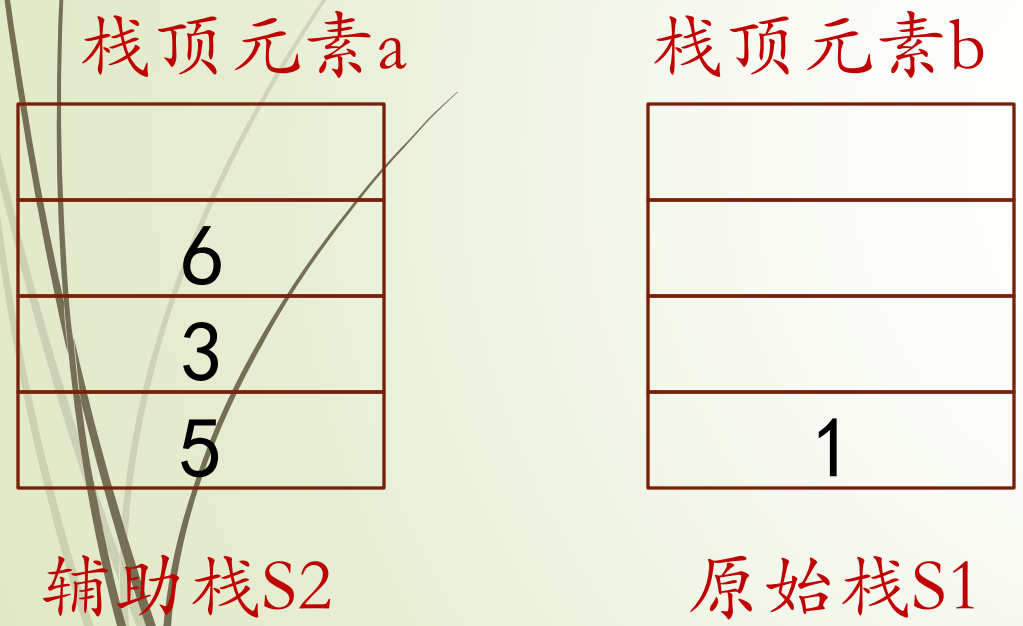
辅助栈S2

1、将S1的所有元素出栈，压入S2;
2、将S2的栈顶元素入栈S1
3、辅助栈S2的栈顶元素为a，原始栈S1的栈顶元素为b

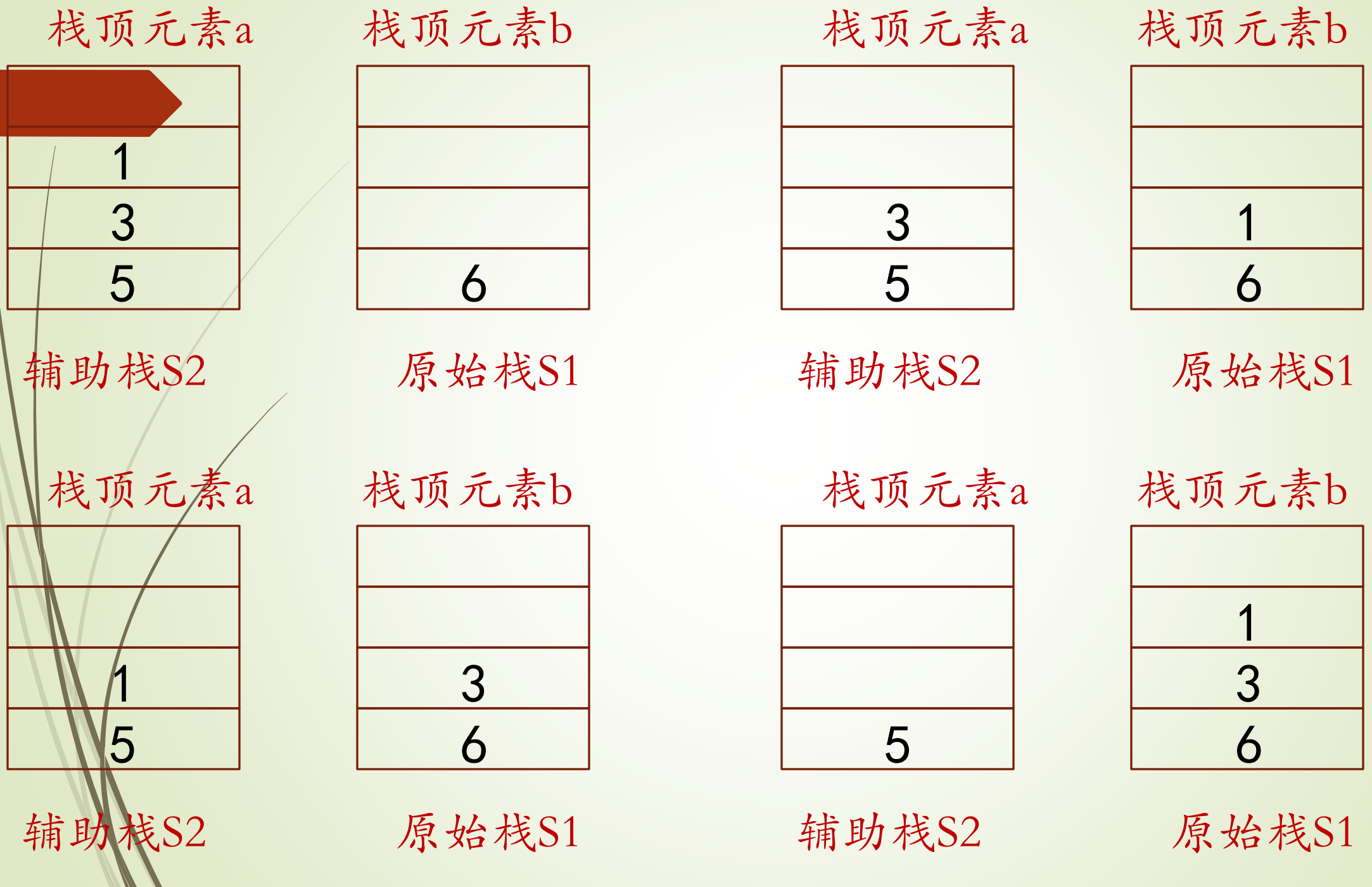
顺序栈

例：实现将一个栈进行降序排列（栈顶元素最小）

思路1: 借助一个辅助栈



参照上面例题的思路：
将a（已出栈）和b进行比较：
(1)如果a<b，则将a入栈S1
(2)如果a>b，则将b出栈压入S2，继续比较a和S1栈顶元素

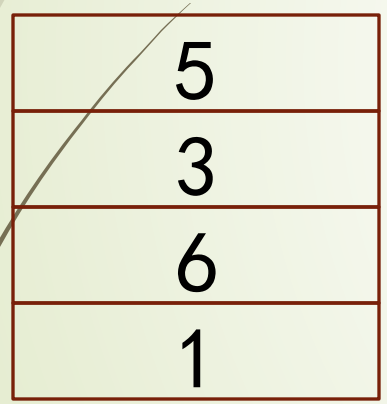

栈顶元素a
栈顶元素b
1
3
5
6
辅助栈S2
原始栈S1
栈顶元素a
栈顶元素b
3
5
1
6
辅助栈S2
原始栈S1
栈顶元素a
栈顶元素b
1
3
5
6
辅助栈S2
原始栈S1
栈顶元素a
栈顶元素b
1
3
5
6
辅助栈S2
原始栈S1


栈顶元素a
栈顶元素b
5
1
3
6
辅助栈S2
原始栈S1
栈顶元素a
栈顶元素b
5
1
3
6
辅助栈S2
原始栈S1
栈顶元素a
栈顶元素b
3
1
5
6
辅助栈S2
原始栈S1
栈顶元素a
栈顶元素b
1
3
5
6
辅助栈S2
原始栈S1

顺序栈

例：实现将一个栈进行降序排列（栈顶元素最小）

思路2: 数组



原始栈S1

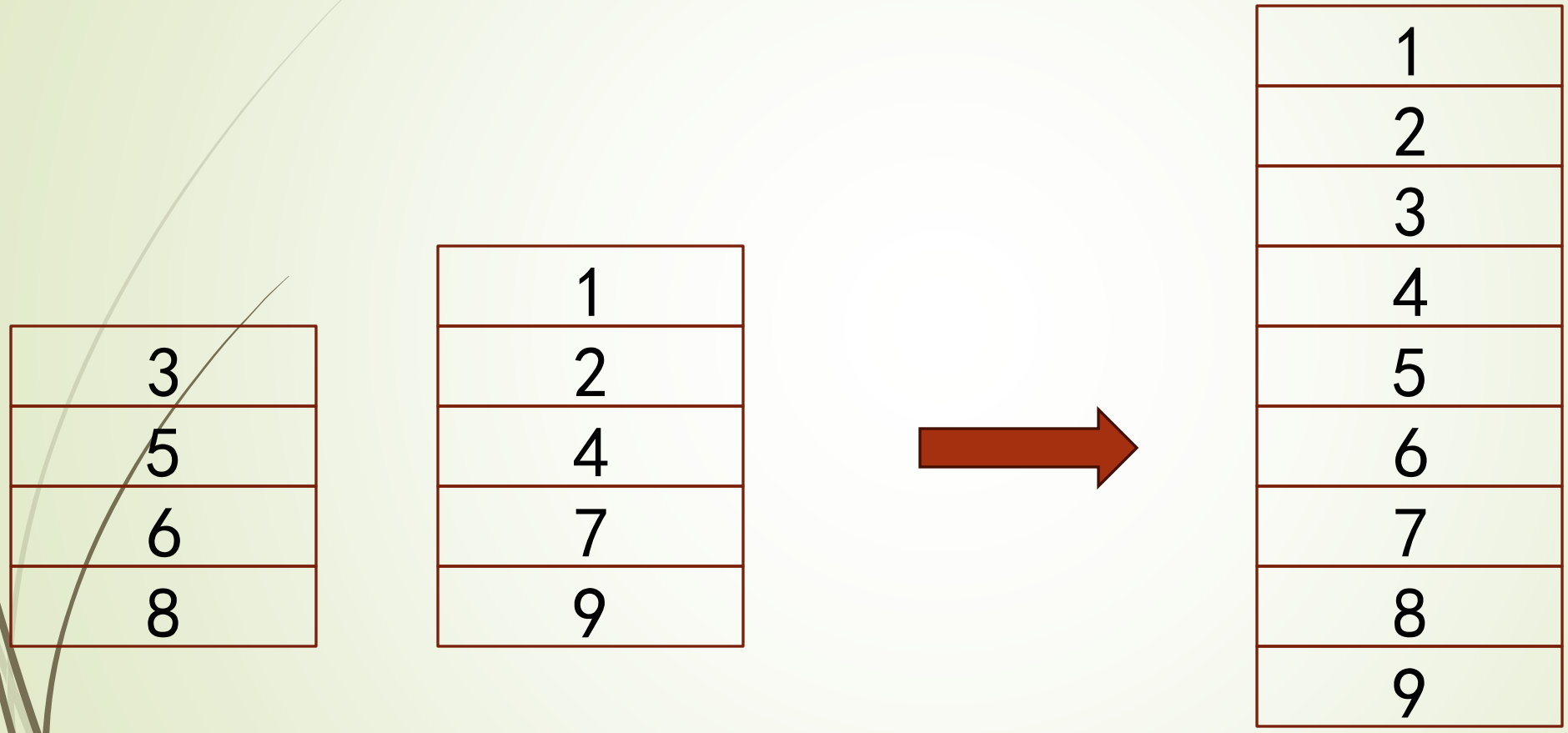
1、将原始栈S1中的数据元素全部弹出到数组中

2、采用某种排序方法将数组中的元素降序排列

3、将数组中的元素全部入栈

顺序栈

例2：将两个降序栈合并成一个降序栈（栈顶元素最小）



将两个降序栈合并成一个降序栈

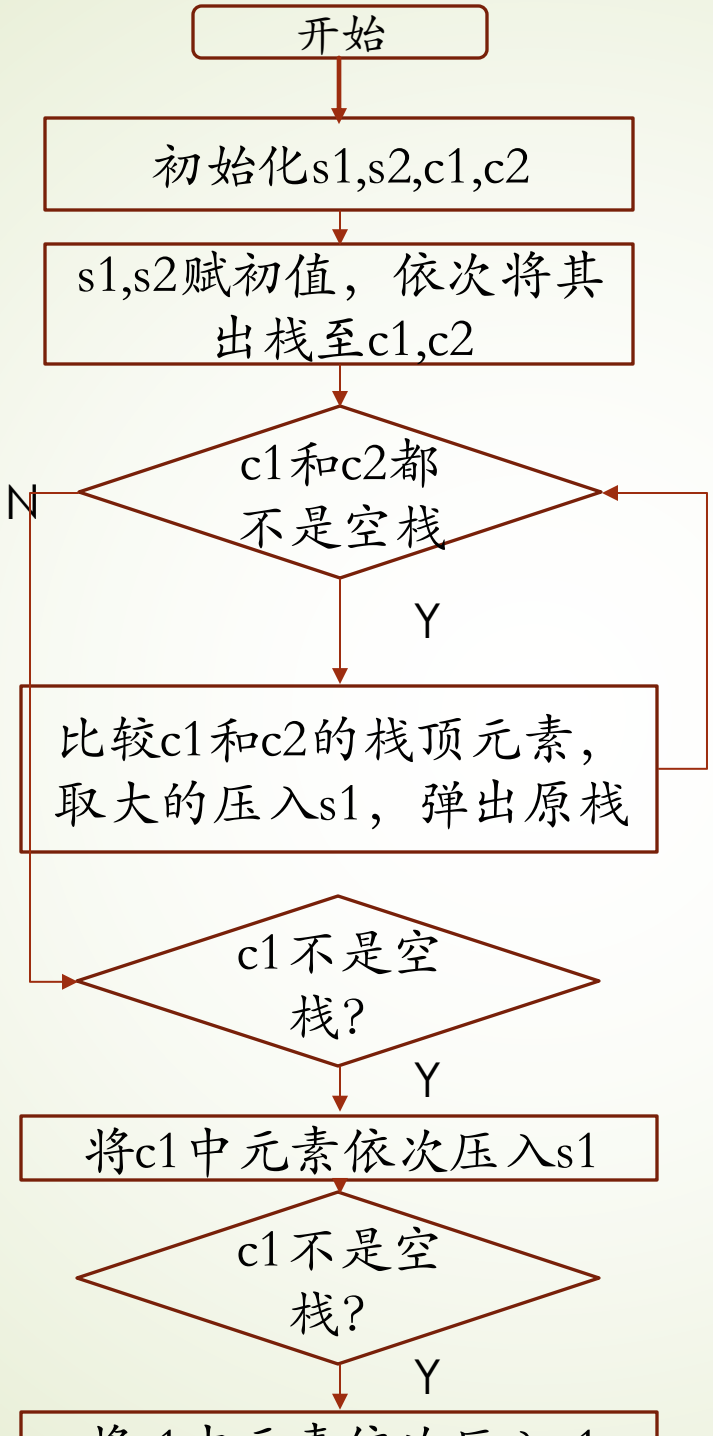
方法：借助两个辅助栈

算法思想：

① 初始栈s1和s2中自底向上是降序排列

② 辅助栈c1和c2是自底向上是升序排列

③ 依次比较c1和c2的栈顶，最大的入栈


开始
初始化s1,s2,c1,c2
s1,s2赋初值，依次将其出栈至c1,c2
c1和c2都不是空栈
N
Y
比较c1和c2的栈顶元素，取大的压入s1，弹出原栈
c1不是空栈?
Y
将c1中元素依次压入s1
c1不是空栈?
Y

将两个降序栈合并成一个降序栈

代码框架:

① 初始化降序栈s1和s2（赋初值）

② 初始化栈c1和c2（分别对应s1和s2的升序栈）

③ 依次比较c1和c2的栈顶元素，将大的元素压入s1栈，同时将其弹出原栈。直到c1或c2为空栈

④ 将c1或c2的元素依次压入s1中

初始化

```
initstack(s1);
initstack(s2);
printf("请输入堆栈s1的数据元素个数:\n");
scanf("%d", &m);
for(i=0; i<m; i++){
    scanf("%d", &input);
    push(s1, input);
}
printf("请输入堆栈s2的数据元素个数:\n");
scanf("%d", &m);
for(i=0; i<m; i++){
    scanf("%d", &input);
    push(s2, input);
}
```

初始化

```
initstack(c1);
initstack(c2);
//将s1的元素依次压入c1（c1变成升序栈）
while(!stackempty(s1))
{
    push(c1, top(s1));
    pop(s1);
}
//将s2的元素依次压入c2（c2变成升序栈）
while(!stackempty(s2))
{
    push(c2, top(s2));
    pop(s2);
}
```

筛选大元素

```
while (!stackempty(c1) && !stackempty(c2)) {
    if(top(c1) > top(c2)){
        push(s1, top(c1));
        pop(c1);
    }else{
        push(s1, top(c2));
        pop(c2);
    }
}
```

剩余元素入栈

```
while (!stackempty(c1)) {
    push(s1, top(c1));
    pop(c1);
}
while (!stackempty(c2)) {
    push(s1, top(c2));
    pop(c2);
}
while(!stackempty(s1))
{
    printf("%d, ", top(s1));
    pop(s1);
}
```