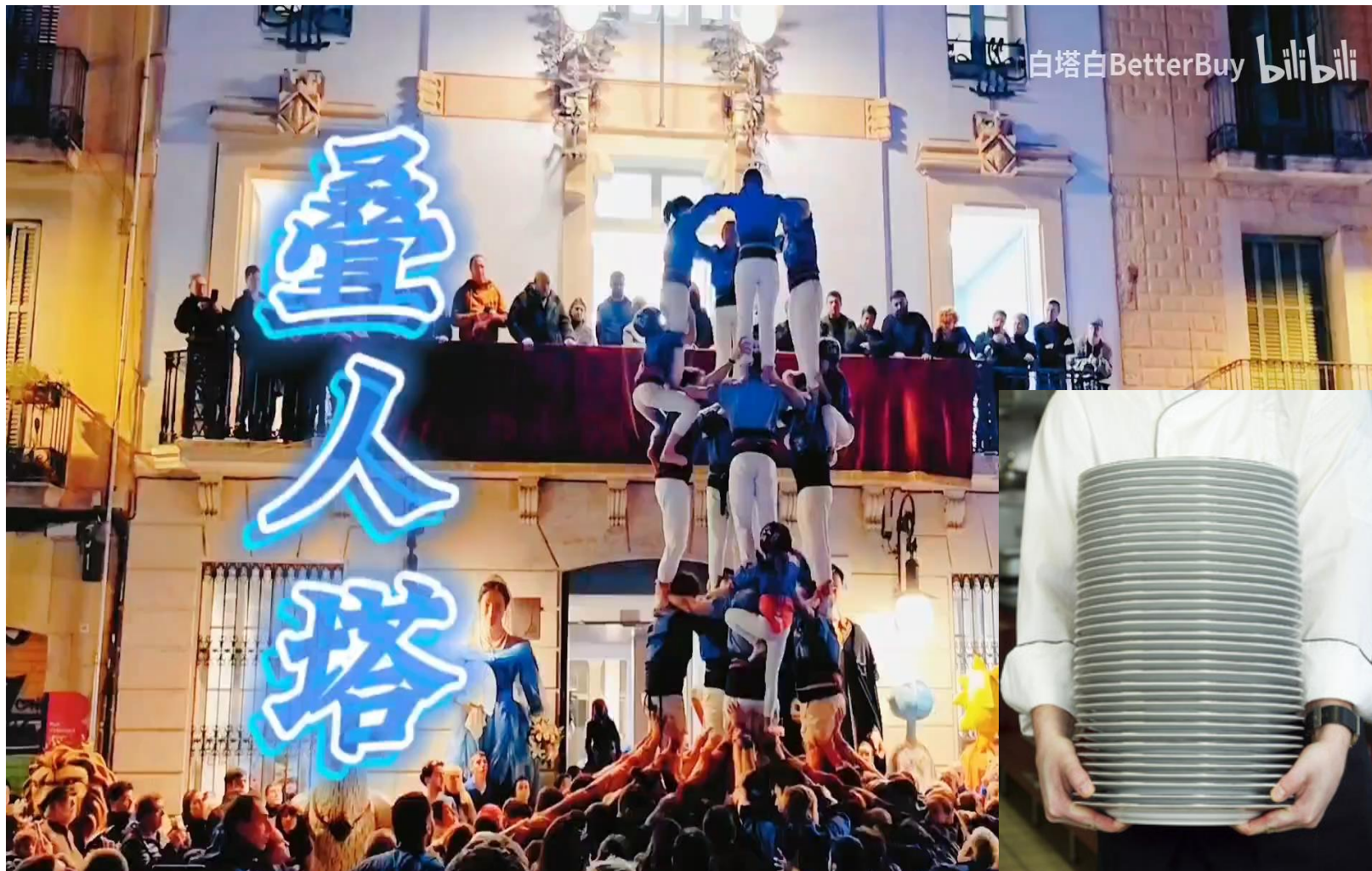


第三章 栈和队列

第三章 栈和队列

- **3.1 栈**
- **3.2 栈的实现**
- **3.3 栈的应用**
- **3.4 队列**
- **3.5 队列的实现**
- **3.6 队列的应用**

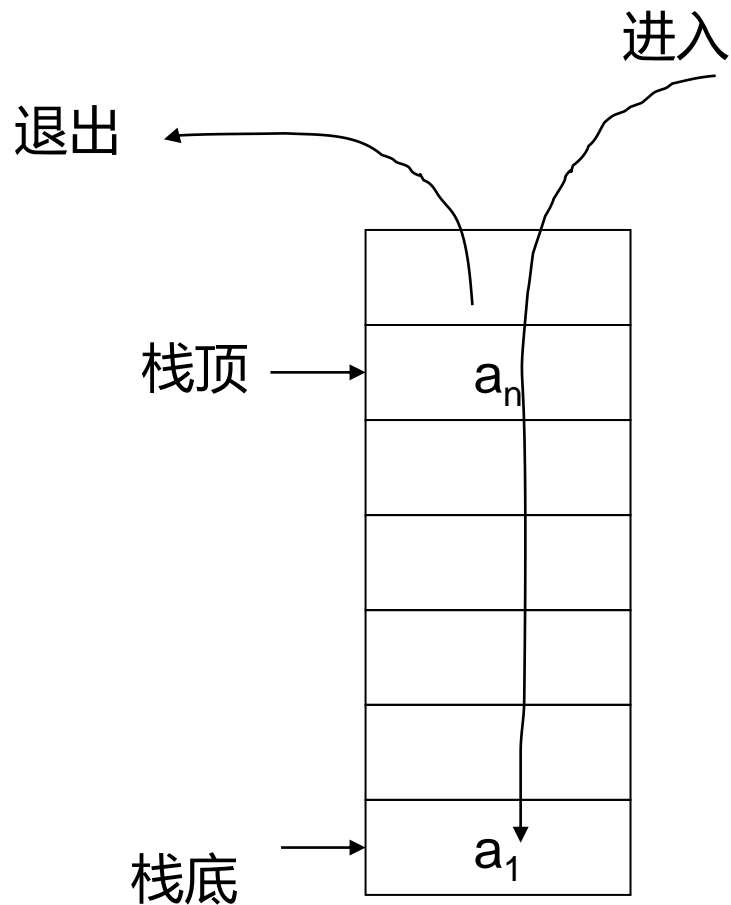
3.1 栈



3.1 栈

栈(stack)是限制线性表中元素的插入和删除**只能**在线性表的**同一端**进行的一种特殊线性表。

允许插入和删除的一端，为变化的一端，称为**栈顶**(Top)，另一端为固定的一端，称为**栈底**(Bottom)。

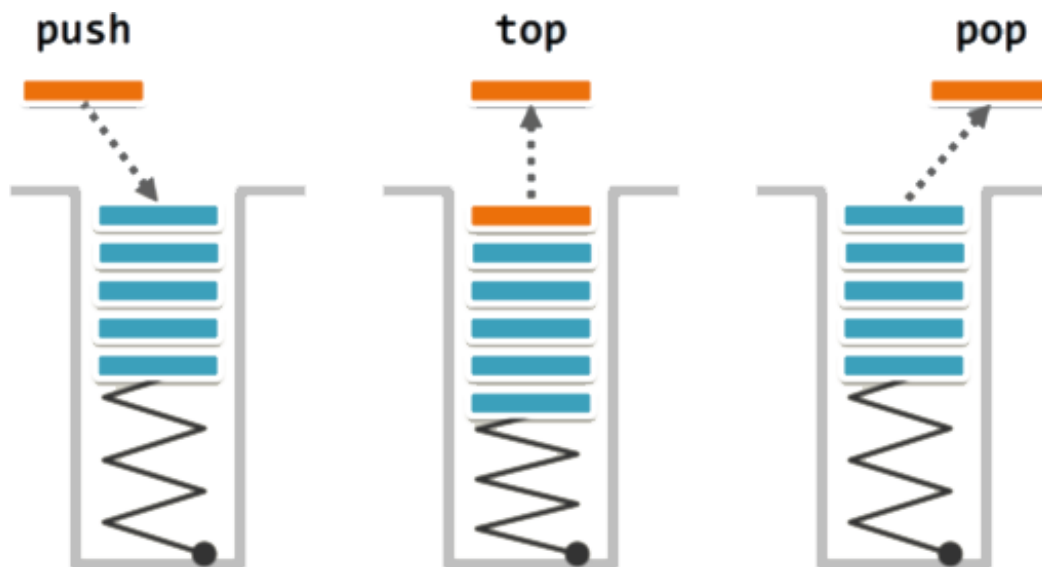


3.1 栈

最先放入栈中元素在栈底，最后放入的元素在栈顶，而删除元素刚好相反，最后放入的元素最先删除，最先放入的元素最后删除。

栈是一种后进先出(Last In First Out)的线性表，简称为LIFO表。

- ❑ 先进后出LIFO
- ❑ 后进先出FILO



3.1 栈

栈是仅在表尾进行插入、删除操作的线性表。

表尾(即 a_n 端)称为**栈顶** /top；表头(即 a_1 端)称为**栈底**/base

例如：栈 $S = (a_1, a_2, a_3, \dots, a_{n-1}, a_n)$

a_1 称为栈底元素

a_n 称为栈顶元素

强调：插入和删除都只能在表的一端（栈顶）进行！

插入元素到栈顶的操作，称为**入栈**。

从栈顶删除最后一个元素的操作，称为**出栈**。

想一想：要从栈中取出 a_1 ，应当如何操作？



3.1 栈

Q1: 栈是什么? 它与一般线性表有什么不同?

堆栈是一种特殊的线性表, 它只能在表的一端 (即栈顶) 进行插入和删除运算。

与一般线性表的区别: 仅在于运算规则不同。

一般线性表

逻辑结构: 1:1

存储结构: 顺序表、链表

运算规则: 随机存取

堆栈

逻辑结构: 1:1

存储结构: 顺序栈、链栈

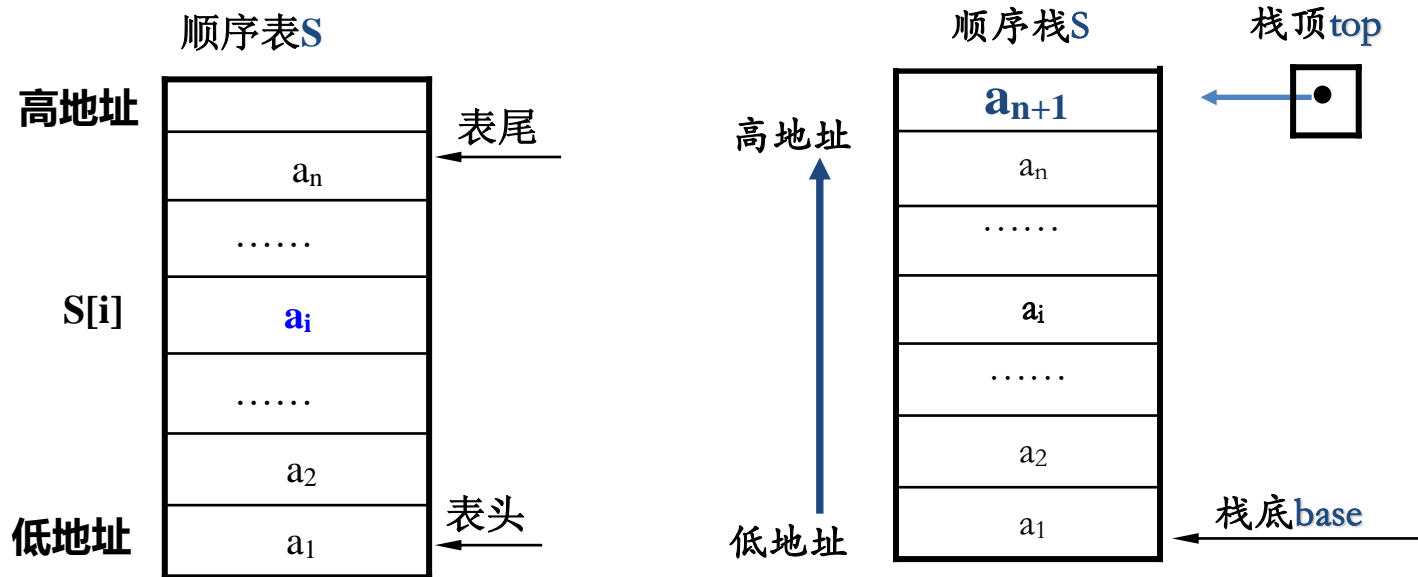
运算规则: 后进先出(LIFO)

“进” = 插入 = 压入
= PUSH (a_{n+1})

“出” = 删除 = 弹出 = POP(a_n)

3.1 栈

以线性表 $S = (a_1, a_2, \dots, a_{n-1}, a_n)$ 为例



写入: $S[i] = a_i$

读出: $e = S[i]$

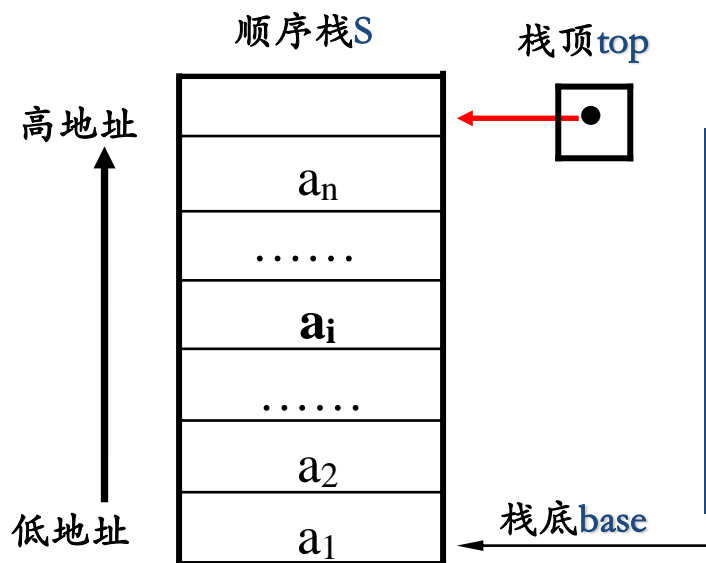
压入(PUSH): $S[top++] = a_{n+1}$

弹出(POP): $e = S[--top]$

前提: 一定要预设栈顶指针top

3.1 栈

Q2: 顺序表和顺序栈的操作有何区别?



栈不存在的条件: $\text{base}=\text{NULL};$

栈为空的条件: $\text{base}=\text{top};$

栈满的条件: $\text{top}-\text{base}=\text{stacksize};$

一般栈顶不存放元素。

入栈口诀: 堆栈指针top “先压后加”: $S[\text{top}++] = a_{n+1}$

出栈口诀: 堆栈指针top “先减后弹”: $e = S[--\text{top}]$

3.1 栈

Q3: 为什么要设计堆栈? 它有什么独特用途?

1. 调用函数或子程序非它莫属;
2. 递归运算的有力工具;
3. 用于保护现场和恢复现场;
4. 简化了程序设计的问题。

3.1 栈

例：序列 1 2 3 经过栈的操作可能的所有结果？

可以通过穷举所有可能性来求解：

① 1入1出，2入2出，3入3出，即123；

② 1入1出，2、3入，3、2出，即132；

③ 1、2入，2出，3入3出，即231；

④ 1、2入，2、1出，3入3出，即213；

⑤ 1、2、3入，3、2、1出，即321；

合计有5种可能性。



3.1 栈

例：一个栈的输入序列是12345，若在入栈的过程中允许出栈，则栈的输出序列43512可能实现吗？

43512不可能实现，主要是其中的12顺序不能实现



思考：有无通用的
判别原则？

3.1 栈

例：设依次进入一个栈的元素序列为c, a, b, d, 则可得到出栈的元素序列是：

- A) a, b, c, d B) c, d, a, b
C) b, c, d, a D) a, c, d, b

答： A) 、 D) 可以， B) 、 C) 不行。

讨论：有无通用的判别原则？

有！若输入序列是 $\cdots, P_i \cdots P_j \cdots P_k \cdots (i < j < k, \text{输入序号})$ ，一定不存在这样的输出序列 $\cdots, P_k \cdots P_i \cdots P_j \cdots$

即对于输入序列1, 2, 3, 不存在输出序列3, 1, 2

3.1 栈

例：阅读下列递归过程，说明其功能。

```
void test(int &sum){
```

```
① int x;
```

```
② scanf(x);
```

```
if(x==0)
```

```
③ sum=0;
```

```
else{
```

```
④ test(sum);
```

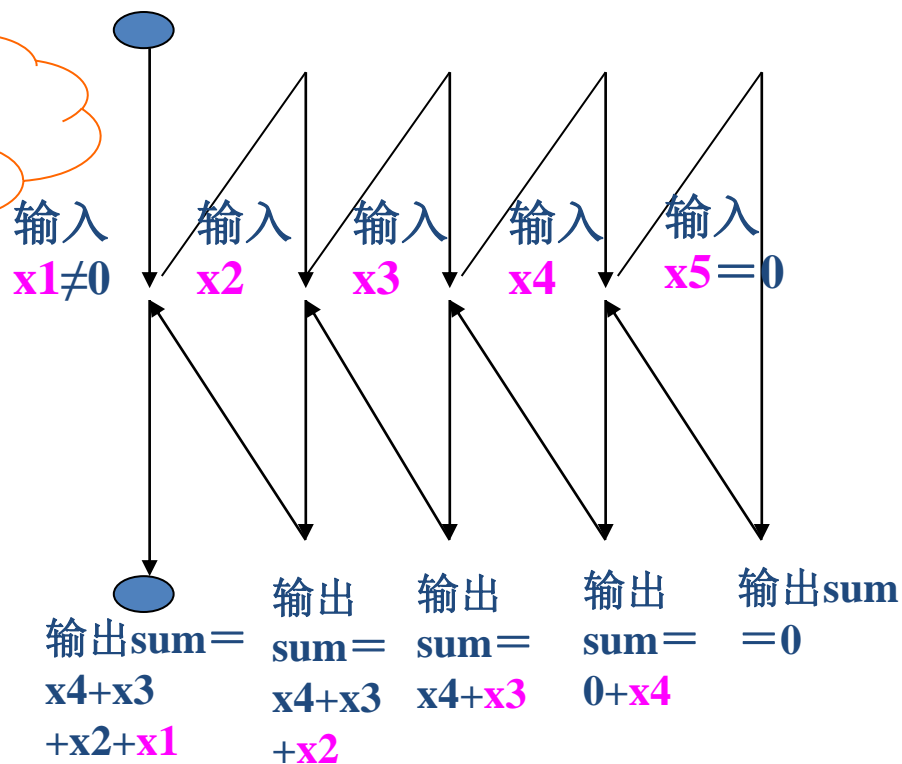
```
⑤ sum+=x;
```

```
⑥ printf(sum);
```

```
⑦ }
```

注意：最先输入的数据 x_1 最后才被累加

断点地址入栈



3.1 栈

栈的抽象数据类型定义:

(教材P44-45)

ADT Stack{

数据对象: $D = \dots\dots\dots$

数据关系: $R = \dots\dots\dots$

基本操作:

$\dots\dots\dots$

} ADT Stack

入栈、出栈、
建栈初始化、
判断栈满或栈空、
读栈顶元素值等。

3.1 栈

即栈顶

- (1) 规则 限定只能在表的一端进行插入和删除运算的线性表。
- (2) 逻辑结构 与线性表相同，仍为一对一(1:1)关系。
- (3) 存储结构 用顺序栈或链栈存储均可

3.1 栈

- (4) **运算规则** 只能在**栈顶**运算，且访问结点时依照**后进先出** (LIFO) 或**先进后出** (FILO) 的原则。
- (5) **实现方式** 关键是编写**入栈**和**出栈**函数，具体实现依顺序栈或链栈的存储结构有别而不同。
- (6) **基本操作有**：建栈、判断栈满或栈空、入栈、出栈、读栈顶元素值，等等。

第三章 栈和队列

- 3.1 栈
- 3.2 栈的实现
- 3.3 栈的应用
- 3.4 队列
- 3.5 队列的实现
- 3.6 队列的应用

3.2 栈的实现

- 顺序栈——栈的顺序存储结构
- 链栈——栈的链式存储结构
- 静态分配整型指针

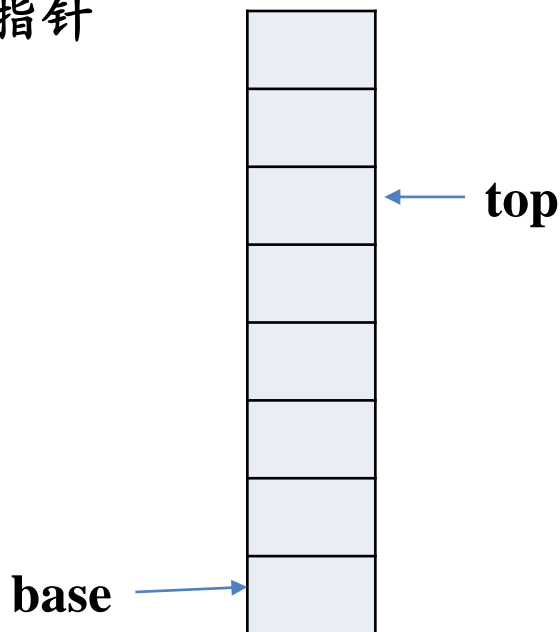
3.2 栈的实现

顺序栈的存储表示 (教材P46)

```
#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; //栈的基址即栈底指针
    SElemType    * top;  //栈顶指针
    int          stacksize; //当前分配的空间
}SqStack;
SqStack s;
```

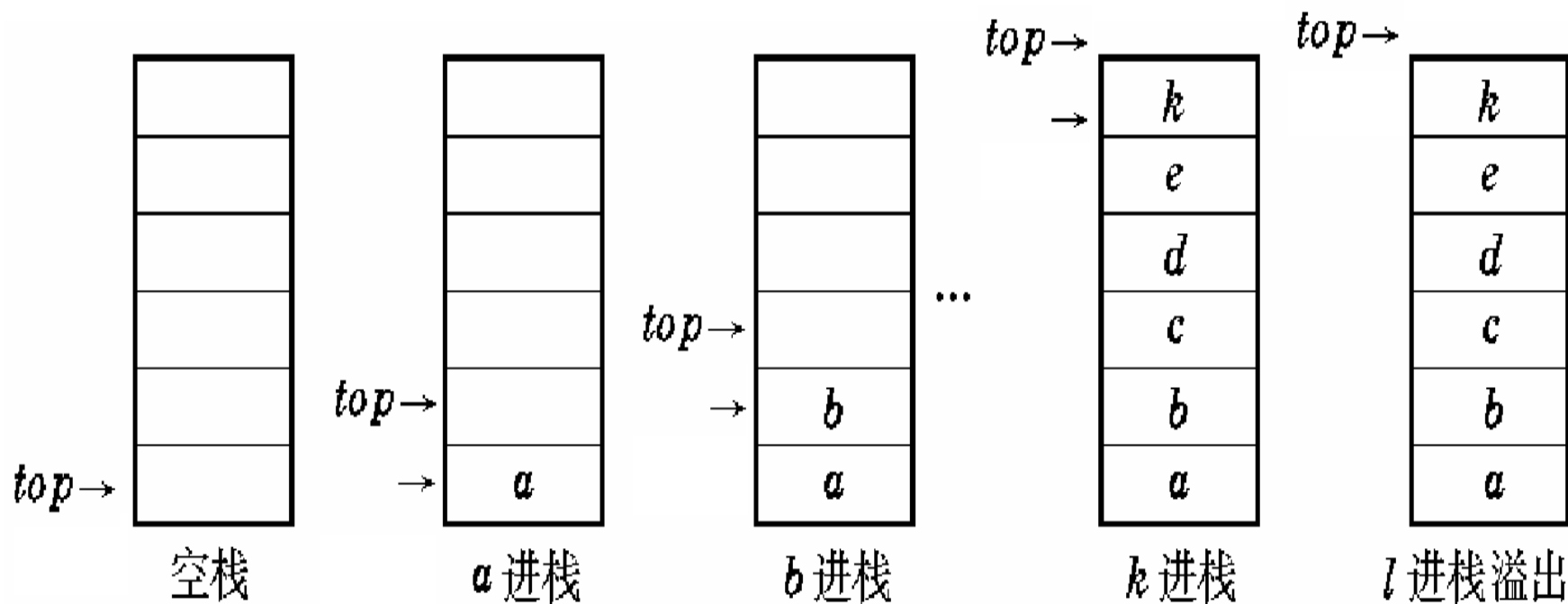
base == NULL时, 表明栈结构不存在

top==base时空栈



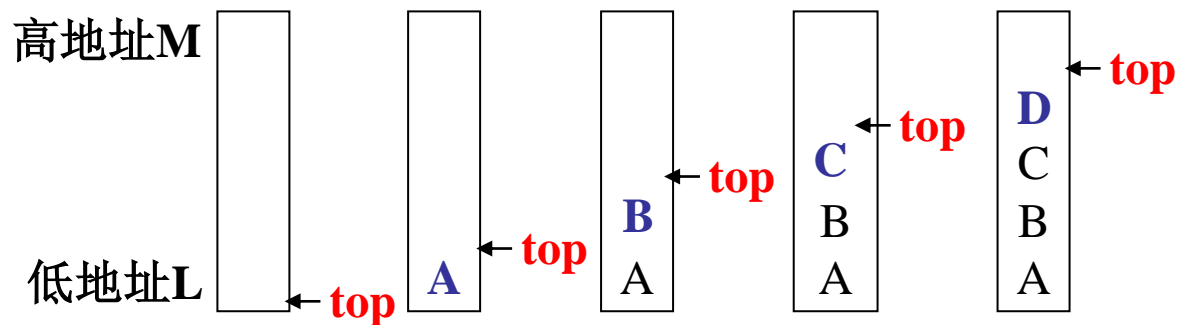
3.2 栈的实现

进栈示例



3.2 栈的实现

顺序栈的入栈操作——例如用堆栈存放 (A, B, C, D)



核心语句：

`top=L;`

`Push (A);`

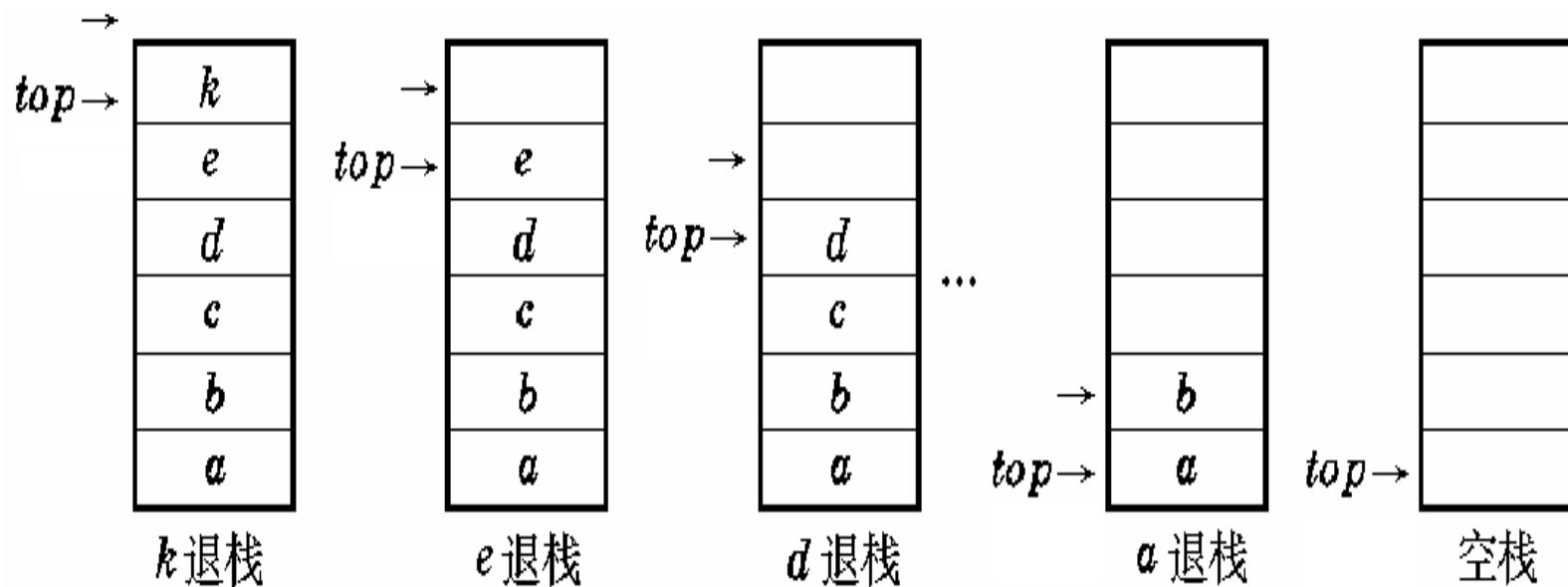
`Push (B);`

`Push (C);`

`Push (D);`

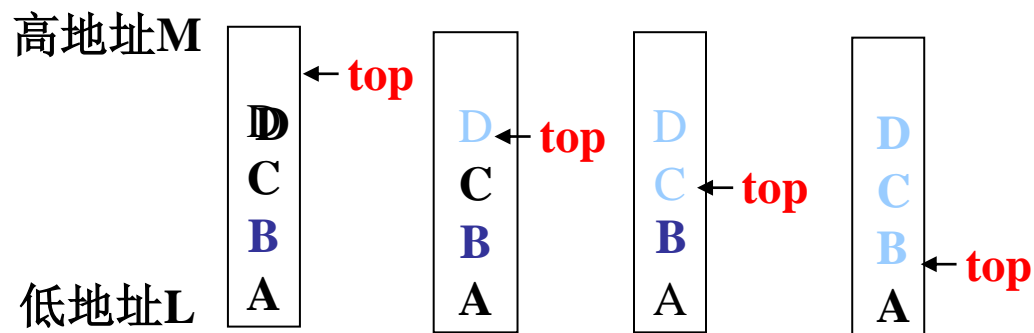
3.2 栈的实现

退栈示例



3.2 栈的实现

顺序栈出栈操作——例如从栈中取出 ‘B’



核心语句:

```
Pop ();
```

```
Pop ();
```

```
Printf( Pop () );
```


3.2 栈的实现

- 栈空条件: `s.top == s.base` 此时不能出栈
- 栈满条件: `s.top - s.base >= s.stacksize`
- 进栈操作: `*s.top++ = e;` 或 `*s.top = e; s.top++;`
- 退栈操作: `e = *--s.top;` 或 `s.top--; e = *s.top;`
- 当栈满时再做进栈运算必定产生空间溢出, 简称“上溢”;
- 当栈空时, 再做退栈运算也将产生溢出, 简称为“下溢”。

3.2 栈的实现

栈的五种基本运算

- (1) 建栈;
- (2) 判断栈满或栈空;
- (3) 入栈;
- (4) 出栈;
- (5) 读栈顶元素值。

3.2 栈的实现

(1) 建栈

```
Status InitStack (SqStack &S) {  
    S.base = (SElemType)malloc (STACK_INIT_SIZE *  
        sizeof(ElemType));  
    if (!S.base) return (OVERFLOW);  
    S.top=S.base;  
    S.stacksize = STACK_INIT_SIZE;  
    return OK;  
}
```

3.2 栈的实现

(2) 判断栈满或栈空

```
Status  IsEmptyStack(SqStack S) {  
    if(S.top==S.base) return TRUE;  
    else return  FALSE;  
}
```

```
Status  IsFullStack(SqStack S) {  
    if(s.top-s.base>=s.stacksize) return TRUE;  
    else return  FALSE;  
}
```

3.2 栈的实现

(3) 入栈

```
Status Push(SqStack &S, SElemType e) {
    if (S.top - S.base >= S.stacksize) {
        S.base = (SElemType*)realloc(S.base,
            (S.stacksize + STACKINCREMENT) * sizeof(ElemType));
        if (!S.base) return (OVERFLOW);
        S.top = S.base + S.stacksize;
        S.stacksize += STACKINCREMENT;
    }
    *S.top++ = e; /* *S.top = e; S.top = S.top+1;
return OK;
}
```

3.2 栈的实现

(4) 出栈

```
Status Pop(SqStack &S, SElemType &e)
{
    if (S.top == S.base) return ERROR;
    e=*--S.top;  // S.top=S.top-1;e=*S.top;
    return OK;
}
```

3.2 栈的实现

(5) 读栈顶元素值

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

3.2 栈的实现

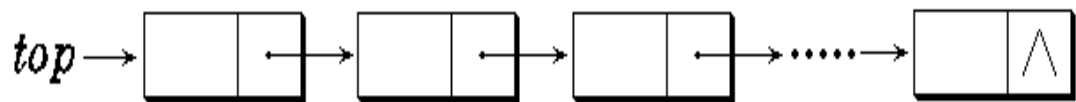
栈的链式存储结构

- 不带头结点的单链表
- 其插入和删除操作仅限制在表头位置上进行。
- 链表的头指针即栈顶指针
- 类型定义：

```
typedef struct SNode{  
    SElemType  data;  
    struct SNode  *next;  
}SNode, *LinkStack;  
LinkStack s;
```


3.2 栈的实现

□ 链栈示意图 p47 图3.3

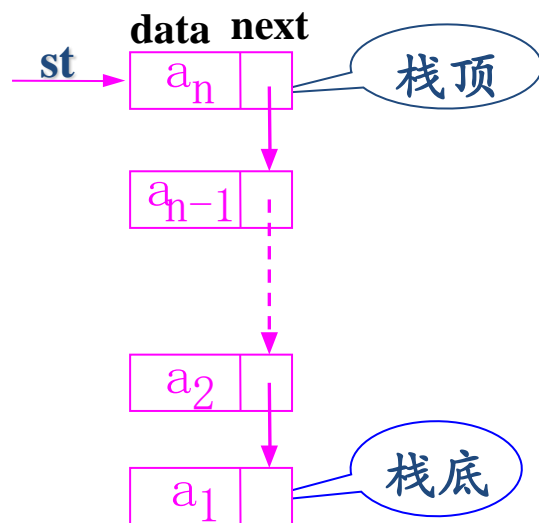


□ 栈空条件: $s == \text{NULL}$

□ 栈满条件: 无 / 无Free Memory可申请

进栈 相当于链表在 top 结点前插入

出栈 相当于链表在将 top 结点删除



3.2 栈的实现

(1) 入栈

```
Status  Push_L (LinkStack &s, SElemType e)
{
    p=(LinkStack)malloc(sizeof(SNode));
    if (!p) exit(Overflow);
    p->data = e;    p->next = s;    s=p;
    return OK;
}
```

3.2 栈的实现

(2) 出栈

```
Status Pop_L (LinkStack &s, SElemType &e)
{
    if (!s) return ERROR;
    e=s->data;  p=s;  s=s->next;
    free(p);
    return OK;
}
```

3.2 栈的实现

- 1) 链栈不必设头结点，因为栈顶（表头）操作频繁；
- 2) 链栈一般不会出现栈满情况，除非没有空间导致malloc分配失败。
- 3) 链栈的入栈、出栈操作就是栈顶的插入与删除操作，修改指针即可完成。
- 4) 采用链栈存储方式的优点是，可使多个栈共享空间；当栈中元素个数变化较大，且存在多个栈的情况下，链栈是栈的首选存储方式。

3.2 栈的实现

栈的静态分配整型指针存储

□ 定义

```
#define MAXSIZE 100
typedef struct {
    SElemType  base[MAXSIZE];
    int top;
} SqStack;
SqStack  s;
```

3.2 栈的实现

(1) 初始化

```
status InitStack(SqStack &s)
{
    s.top = 0;
    return OK;
}
```

3.2 栈的实现

(2) 入栈

```
Status Push(SqStack &s, SElemType e)
{
    if (s.top == MAXSIZE) return ERROR;
    s.base[s.top] = e;  s.top++;
    return OK;
}
```

3.2 栈的实现

(3) 出栈

```
Status Pop(SqStack &s, SElemType &e)
{
    if (s.top == 0) return ERROR;
    s.top--;
    e = s.base[s.top];
    return OK;
}
```


3.2 栈的实现

(3) 读栈顶元素值

```
Status  GetTop(SqStack S, SElemType  &e)
{
    if (S.top == 0) return ERROR;
    e = s.base[s.top-1];
    return OK;
}
```

第三章 栈和队列

- 3.1 栈
- 3.2 栈的实现
- 3.3 栈的应用
- 3.4 队列
- 3.5 队列的实现
- 3.6 队列的应用

3.3 栈的应用

(1) 数制转换

十进制N和其它进制数的转换是计算机实现

$$N = (N \text{ div } d) \times d + N \text{ mod } d$$

例如： $(1348)_{10} = (2504)_8$ ，其运算过程如下

	N	N div 8	N mod 8	
计算顺序 ↓	1348	168	4	↑ 输出顺序
	168	21	0	
	21	2	5	
	2	0	2	

第三章 栈和队列

(1) 数制转换

```
void Conversion() {  
    InitStack(s);  
    scanf("%d,%d",&N,&base);  
    N1=N;  
    while (N1){  
        Push(s,N1%base);  
        N1 = N1/base;  
    }  
    while (!(StackEmpty(s))){  
        Pop(s,e);  
        if (e>9) printf("%c",e+65);  
        else printf("%c",e+48);  
    }  
    printf("\n");  
}
```

第三章 栈和队列

(1) 数制转换

```
void conversion(int X ) {  
    if (X/8!=0)  
        conversion(X/8);  
    printf("%d", X%8);  
}
```

用递归编程时，不需要用户自行定义栈。
它是由编译程序加以设立和处理的。

第三章 栈和队列

(2) 括号匹配的检验

[({	}	[])]
1	2	3	4	5	6	7	8

检验括号是否匹配的方法可用 **“期待的急迫程度”** 这个概念来描述。

设置一个**栈**，每读入一个括号，若是右括号，则或者使置于栈顶的最急迫的期待得以消解，或者是不合法的情况；若是左括号，则作为一个新的更急迫的期待压入栈中，自然使原有的在栈中的所有未消解的期待的急迫性都降了一级

第三章 栈和队列

(2) 括号匹配的检验

```
Judgement(){/*判断表达式是否配对，表达式以 '#' 结束*/
    InitStack(sta);
    Push(&sta,'#'); /*把 '#' 放在栈底*/
    scanf("%c",&ch);
    valid = TRUE; /*valid为FALSE表示括号配对失败*/
    while (ch != '#'){ /*当读入字符为左括号时进栈*/
        if ((ch=='(')||(ch=='[')||(ch=='{')) Push(&sta,ch);
        else if ((ch==')')||(ch==']')||(ch=='}')){
            chpop = Pop(&sta);
            if (chpop == '#'){valid = FALSE;break;}/*右括号多于左括号*/
            if (!(ch=='')&&(chpop=='(') || (ch==']')&&(chpop=='[')
                || (ch=='}')&&(chpop=='{')){
                valid == FALSE; break;
            }
        }/*else*/
        scanf("%c",&ch);/*读入下一字符*/
    }
    if (Pop(&sta)!='#') valid=FALSE; /*左括号多于右括号*/
    if (valid) printf("括号配对成功 ");
    else printf("括号配对不成功 ");
}
```

第三章 栈和队列

(3) 行编辑程序

每读入一个字符

'#' —— 删除前面一个字符

'@' —— 删除前面所有字符

'*' —— 输入结束

退栈

置空栈

编辑结束

```
while( '#' != *s)
    putchar(*s++);
```



```
while (*s)
    putchar(*s++);
```

用栈来实现这种功能的文字编辑器

第三章 栈和队列

(3) 行编辑程序

```
void LineEdit( ){
    InitStack(s);
    ch=getchar();
    while(ch!=eof){
        while(ch!=eof && ch!='\n'){
            switch(ch){
                case '#' : pop(s,c);      break;
                case '@' : clearstack(s); break;
                default  : push(s,ch);     break;
            }
            ch=getchar( );
        }
        把从栈底到栈顶的栈内字符传送到调用过程的数据区;
        ClearStack(s);
        if(ch!=eof) ch=getchar( );
    }
    DestroyStack(s);
}
```

第三章 栈和队列

(4) 迷宫求解

#	#	#	#	#	#	#	#	#	#
#	→	↓	#	\$	\$	\$	#		#
#		↓	#	\$	\$	\$	#		#
#	↓	←	\$	\$	#	#			#
#	↓	#	#	#				#	#
#	→	→	↓	#				#	#
#		#	→	→	↓	#			#
#	#	#	#	#	↓	#	#		#
#					→	→	→	⊙	#
#	#	#	#	#	#	#	#	#	#

通常用的是“穷举求解”的方法

第三章 栈和队列

(4) 迷宫求解

基本思想:

- 若当前位置“可通”，则纳入路径，继续前进;
- 若当前位置“不可通”，则后退，换方向继续探索;
- 若四周“均无通路”，则将当前位置从路径中删除出去。

第三章 栈和队列

(4) 迷宫求解

设定当前位置的初值为入口位置;

do {

 若当前位置可通,

 则 {将当前位置插入栈顶;

 若该位置是出口位置, 则算法结束;

 否则切换当前位置的东邻方块为新的当前位置;

}

否则 {

 若栈不空且栈顶位置尚有其他方向未被探索,

 则设定新的当前位置为: 沿顺时针方向旋转找到的栈顶位置的下一相邻块;

 若栈不空但栈顶位置的四周均不可通,

 则 { 删去栈顶位置; // 从路径中删去该通道块

 若栈不空, 则重新测试新的栈顶位置,

 直至找到一个可通的相邻块或出栈至栈空;

 }

}

} while (栈不空) ;

第三章 栈和队列

```
bool MazePath(PosType start, PosType end) {
    PosType curpos = start; Stack S; SElemType e;
    int curstep = 1; InitStack(S);
    do {
        if (Pass(curpos)) { //当前位置不是墙和已经走过的路径
            FootPrint(curpos); //留下足迹
            e.ord = curstep; e.seat = curpos; e.di = 1;
            Push(S, e); //加入路径
            if (IsEqual(curpos, end)) return true;
            curpos = NextPos(curpos, 1); curstep++; //探索下一步
        }
        else { //当前位置是墙或者是已经走过的路径
            if (!StackEmpty(S)) {
                Pop(S, e);
                while (e.di == 4 && (!StackEmpty(S))) {
                    MarkPrint(e.seat); Pop(S, e); //标记这个路线
                }
                if (e.di < 4) {
                    e.di++; Push(S, e); //换下一个方向探索
                    curpos = NextPos(e.seat, e.di); //设定当前位置是该新方向上的相邻块
                }
            }
        }
    } while (!StackEmpty(S));
    return false;
}
```

第三章 栈和队列

(5) 表达式求值

任何一个表达式都是由**操作数** (operand)、**运算符** (operator) 和**界限符**组成的，我们称它们为**单词**。

把**运算符**和**界限符**统称为**算符**。

简单表达式的求值问题，这种表达式只含加、减、乘、除、乘方、括号等运算。

第三章 栈和队列

(5) 表达式求值

对于两个相继出现的算符 Q_1 和 Q_2 ，其优先关系：

$\theta_1 \backslash \theta_2$	+	-	*	/	^	()	#
+	>	>	<	<	<	<	>	>
-	>	>	<	<	<	<	>	>
*	>	>	>	>	<	<	>	>
/	>	>	>	>	<	<	>	>
(<	<	<	<	<	<	=	
)	>	>	>	>		>	>	
#	<	<	<	<	<	<		=

第三章 栈和队列

(5) 表达式求值

算法的基本思想

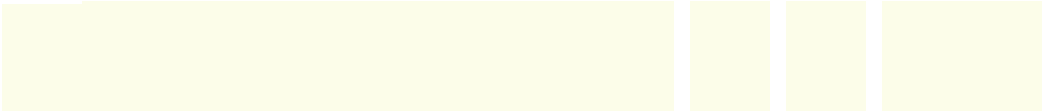
界限符 ‘#’ 优先级别最低

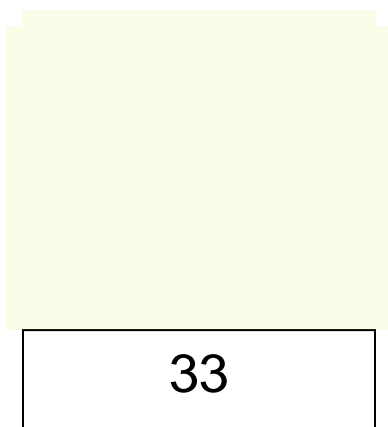
设置两个工作栈：运算符栈OPTR 操作数栈OPND

- 1) 首先置操作数栈为空栈，表达式起始符 ‘#’为运算符栈的栈底元素。
- 2) 依次读入表达式中每个字符，若是操作数则进OPND栈，若是运算符则和 OPTR 栈的栈顶运算符比较优先权后作相应操作，直至整个表达式求值完毕。

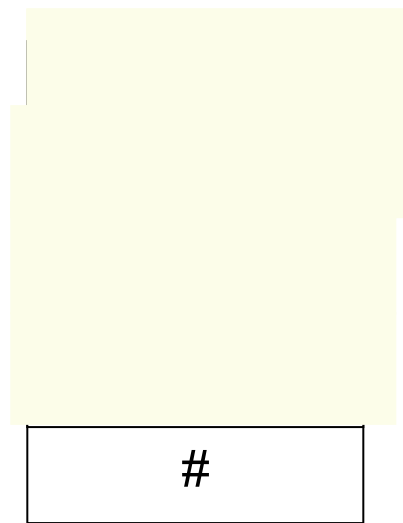
第三章 栈和队列

(5) 表达式求值

输入:  #



操作数栈



运算符栈

第三章 栈和队列

(5) 表达式求值

C是操作符吗?

```
Status EvaluateExpression( OperType &result) {  
    InitStack(OPND); InitStack(OPTR); Push(OPTR, '#'); c=getchar();  
    while((c!='#') && (GetTop(OPTR)!='#'))  
    { if (!In(c, OP)) { Push(OPND, c); c=getchar();}  
      else switch(precede(GetTop(OPTR), c))  
      { case '<': Push(OPTR, c); c=getchar(); break;  
        case '=': Pop(OPTR); c=getchar(); break;  
        case '>': Pop(OPTR, theta); Pop(OPND, b); a=Pop();  
                Push(OPND, Operate(a, theta, b)); break; ;  
      }  
    } //switch } //while  
    result=GetTop(OPND);  
} //EvaluateExpression
```

运算符与栈顶
比较并查3.1表

Operate=a θ b

第三章 栈和队列

(6) 栈与递归

递归的定义

如果一个对象部分的由自己组成，或者是按它自己定义的，则称为**递归**的。

一个递归定义必须一步比一步简单，最后是有终结的，决不能无限循环下去，终结条件就是**递归定义**的出口，简称为**递归出口**。

第三章 栈和队列

(6) 栈与递归

递归函数的特点：在函数内部可以直接或间接地调用函数自身。如阶乘函数：

$$n! = \begin{cases} 1 & , \quad n=0 \\ n*(n-1)! & , \quad n>0 \end{cases}$$

第三章 栈和队列

(6) 栈与递归

```
int fact (int n)
{   if (n == 0)
        return 1;
    else
        return(n*fact(n-1));
};
main( )
{   int n;
    scanf("%d\n",&n);
    printf("%8d%8d\n",n,fact(n));
}
```

r2

r1

第三章 栈和队列

(6) 栈与递归

递归函数到非递归函数的转换

调用前：

- (1) 为被调用函数的局部变量分配存储区；
(活动记录， 数据区)**
- (2) 将所有的实参、返回地址传递给被调用函数；
实参和形参的结合，传值。**
- (3) 将控制转移到被调用函数入口。**

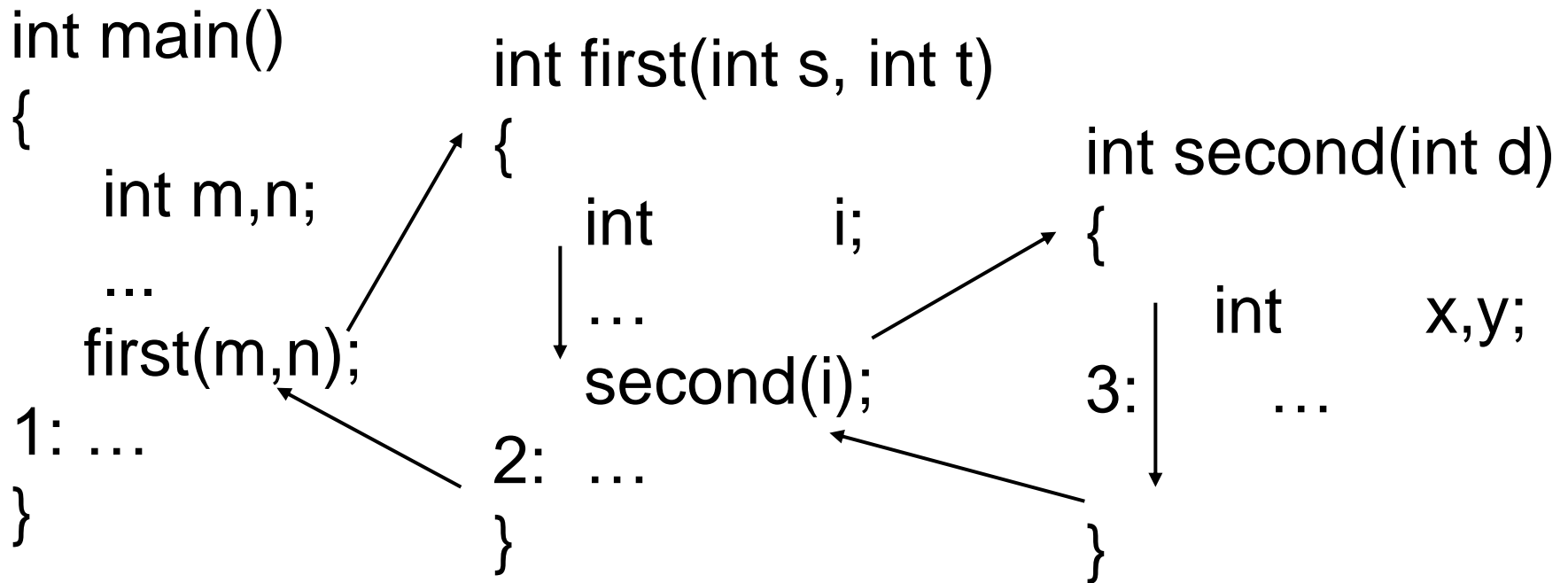
调用后返回：

- (1) 保存被调用函数的计算结果；**
- (2) 释放被调用函数的存储区；**
- (3) 依照被调用函数保存的返回地址将控制转移到调用函数。**

第三章 栈和队列

(6) 栈与递归

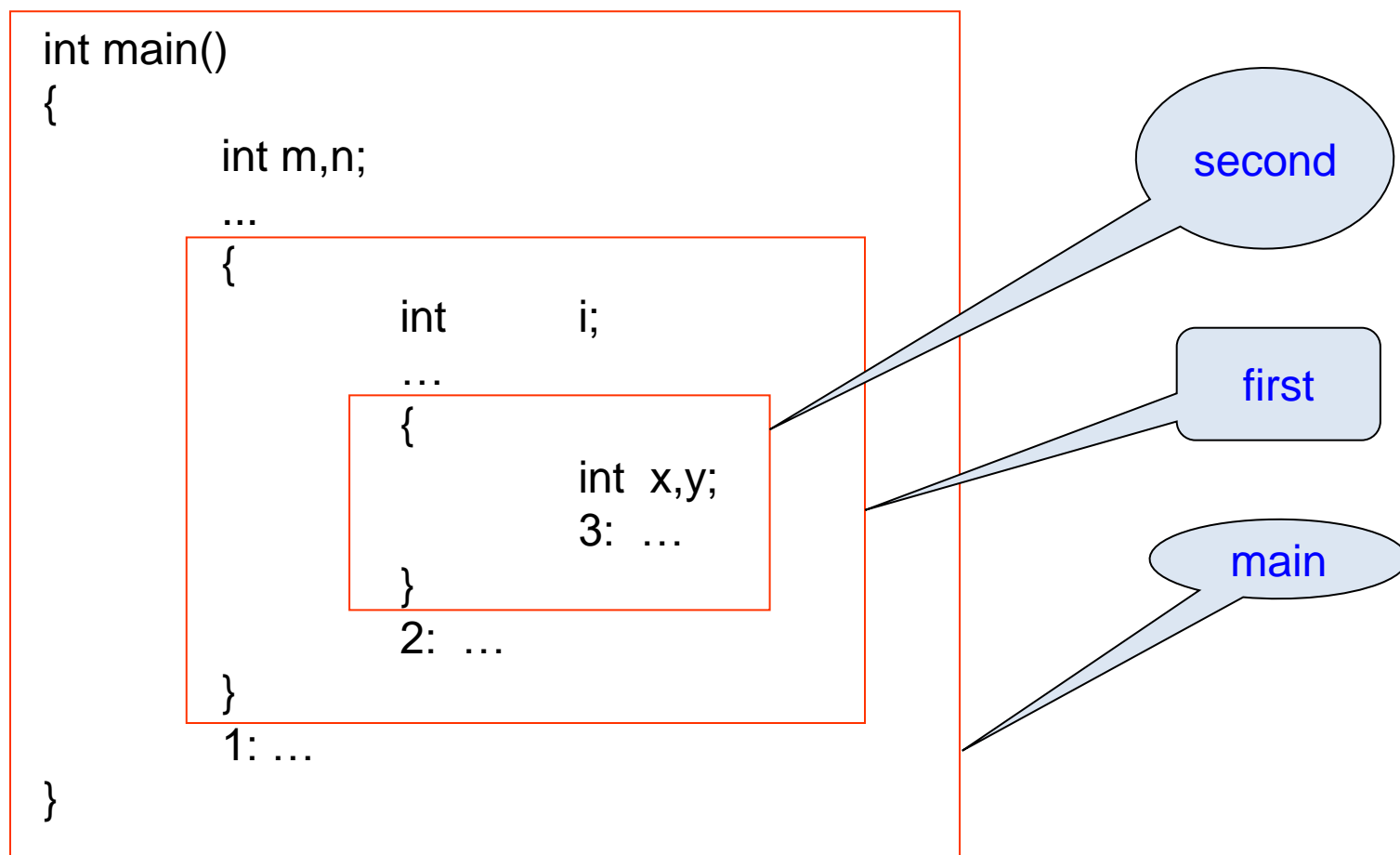
函数嵌套调用时，按照“后调用先返回”的原则进行。



第三章 栈和队列

(6) 栈与递归

函数嵌套结构



第三章 栈和队列

(6) 栈与递归

```
int  nfact(int n);
```

```
{   int res;
```

```
    SqStack st;
```

```
    InitStack(st);
```

```
    while (n>0)
```

```
    {   Push(st,n);
```

```
        n=n-1;    }
```

```
    res=1;
```

```
}
```

阶乘递归算法:

```
int fact (int n)
```

```
{   if (n == 0)
```

```
        return 1;
```

```
    else
```

```
        return(n*fact(n-1));
```

```
};
```

```
while (!StackEmpty(st)
```

```
{   res=res*GetTop(st);
```

```
    Pop(st);    }
```

```
free(st); return(res);
```

正在答疑
