



数据结构与算法

陈越编写

高等教育出版社出版

讲解人：张家琦

日期：2022.09

- ❖ 从不同的应用中抽象出共性的数据组织与操作方法？
- ❖ 还是为每个具体应用都编一个程序？

[例2.1] 在日常数据处理中经常碰到的问题是需要对一组数据进行基本的统计分析。比如，分析一个课程班学生的平均成绩、最高成绩、最低成绩、中位数、标准差等。同样的统计要求也可能发生在其他领域。

➤ **类型名称：**数据集合的基本统计

➤ **数据对象集：**集合 $S = \{ x_1, x_2, \dots, x_N \}$

➤ **操作集：**

1. **ElementType Average(S)：**求S中元素的平均值；
2. **ElementType Max(S)：**求S中元素的最大值；
3. **ElementType Min(S)：**求S中元素的最小值；
4. **ElementType Median(S)：**求S中元素的中位数。

❖ 如何利用程序设计语言实现上述抽象类型？

1. 数据存储

- C语言（包括其他高级语言）提供了数组、结构、链表等。
- 数据结构的存储实现跟所需要的操作密切相关。
- 在数据结构里，利用数组和链表方式来实现的，包括很复杂的数据结构，如图、树等。

2. 操作实现

- 流程控制语句，即分支控制语句（如if-else、switch语句）、循环控制语句（如for、while、do-while语句）。
- 此外，还有模块化的程序设计方法——函数

```
ElementType Average(ElementType S[], int N)
{ /* 求集合元素的平均值。集合元素存放在数组S中，数组大小为N */
    int i;
    ElementType Sum=0;
    for(i = 0; i<N; i++)
        Sum += S[i];          /* 将数组元素累加到Sum中 */
    return Sum/N;
}
```

❖ 求中位数Median(S)

[方法1] 基于排序。首先将集合S从大到小排序，第 $\lceil N/2 \rceil$ （大于等于 $N/2$ 的最小整数）个元素就是中位数。

[方法2] 基于问题分解

比较慢!

- 相近的另一个问题是：求集合中的第K大整数。
当 $K = \lceil N/2 \rceil$ 时，集合的第K大整数就是中位数。
- 求解集合第K大整数问题的一种递归思路是：

```
ElementType FindKthLargest ( ElementType S[], int K)
{
    选取S中的第一个元素e;
    根据e将集合S（不包含e）分解为大于等于e的元素集合S1和小于
    e的元素集合S2;
    if ( |S1| >= K )      return FindKthLargest( S1, K );
    else if ( |S1| < K-1 ) return FindKthLargest( S2, K-|S1|-1 );
    else return e;
}
```

[例2.2] 求集合{ 6 5 9 8 2 1 7 3 4 } 的中位数。

【分析】 由于该集合有9个元素，所以中位数应该是集合从大到小排序后的第 $\lceil 9/2 \rceil = 5$ 个元素。

➤ 首先，选取集合的第一个元素6，根据这个元素从集合中分解出 $S1=\{6, 9, 8, 7\}$ ， $S2=\{5, 2, 1, 3, 4\}$ 。

➤ 由于 $|S1|=4<5$ ，所以该中位数应该在集合S2中，且是S2中第 $(5 - 4 = 1)$ 大整数。

➤ 继续选取S2中的第一个整数5，将S2分解出两个集合 $S1'=\{5\}$ ， $S2'=\{2,1,3,4\}$ 。

➤ 由于 $|S1'|=1$ ，所以5就是S2集合的第1大整数，也就是集合{6 5 9 8 2 1 7 3 4}的中位数。

- ❖ 变量是数据存储的基本单位。变量的类型决定了存储和操作。
- ❖ 几种基本的数据类型：整型、实型（浮点型）、字符型等。
- ❖ 提供了构造数据类型：数组、结构、指针等。

1. 数组

数组是最基本的构造类型，它是一组相同类型数据的有序集合。数组中的元素在内存中连续存放，用数组名和下标可以唯一地确定数组元素。

[例2.3] 求集合元素的最大值。集合元素存放在数组S中，数组大小为N。

```
ElementType Max( ElementType S[], int N )
{
    int i;
    ElementType CurMax = S[0];
    for ( i=1; i<N; i++ )
        if ( S[i] > CurMax ) /* 若S[i]比当前最大值还要大 */
            CurMax = S[i]; /* 则更新当前最大值 */
    return CurMax;
}
```

2. 类型定义 `typedef`

除了使用C语言提供的标准类型和自己定义的一些结构体、枚举等类型外，还可以用 `typedef` 语句来建立已经定义好的数据类型的别名。

`typedef` 原有类型名 新类型名

`typedef int ElementType;`

这样在调用 `ElementType Max(ElementType S[], int N)` 处理 `int` 型数组时，就不需要把每个 `ElementType` 替换成 `int` 了。

3. 指针

指针是C语言中一个非常重要的概念。使用指针可以对**复杂数据**进行处理，能对计算机的**内存进行分配**控制，在函数调用中使用指针还可以**返回多个值**。

(1) 指针与数组

数组名是数组中第1个元素（下标为0）的地址，可以看作是**常量指针**，不能改变指针常量（数组名）的值。

(2) 用指针实现内存动态分配

① 分配函数 **void *malloc(unsigned size)** 。

② 释放函数 **void free(void *ptr)** 。

4. 结构

【定义】结构类型把一些可以是不同类型的数据分量聚合成一个整体。同时，结构又是一个变量的集合，可以单独使用其变量成员。

结构类型定义的一般形式为：

```
struct 结构名{  
    类型名 结构成员名1;  
    类型名 结构成员名2;  
    .....  
    类型名 结构成员名n;  
};
```

① 结构变量的使用

使用结构变量就是对其成员进行操作。格式为：结构变量名.结构成员名。此外，结构变量不仅可以作为函数参数，也可以作为函数的返回值。

4. 结构

② 结构数组：结构与数组的结合

对结构数组元素成员的引用是通过使用数组下标与结构成员操作符“.”相结合的方式完成的，其一般格式为：

结构数组名[下标].结构成员名

③ 结构指针：指向结构的指针

(1) 用*方式访问，形式：（*结构指针变量名）.结构成员名

(2) 用指向运算符“->”访问指针指向的结构成员，形式：

结构指针变量名->结构成员名

④ 共用体

【定义】共用体类型是指将不同的数据项组织成一个整体，它们在内存中占用同一段存储单元。

共用体类型定义的一般形式为：

```
union 共用体名{  
    类型名 成员名1;  
    类型名 成员名2;  
    .....  
    类型名 成员名n;  
};
```

- ❖ 各个成员变量在内存中都使用同一段存储空间，因此共用体变量的长度等于最长的成员的长度。
- ❖ 共用体的访问方式同结构体类似。

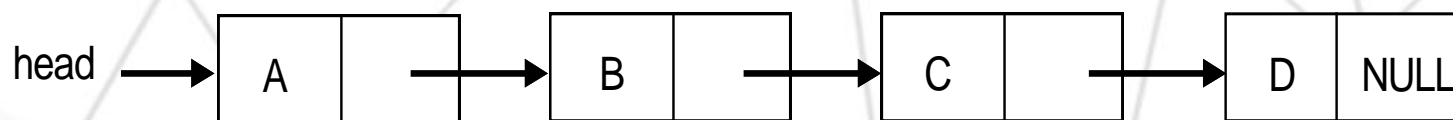
```
int main()  
{  
    union key {  
        int k;  
        char ch[2];  
    } u;  
    u.k = 258;  
    printf(“%d %d\n”, u.ch[0],u.ch[1]);  
    return 0;  
}
```

5. 链表

链表是一种重要的基础数据结构，也是实现**复杂数据结构**的重要手段。它不按照线性的顺序存储数据，而是由若干个同一结构类型的“**结点**”依次串接而成的，即每一个结点里保存着**下一个结点的地址**（指针）。

链表又分**单向链表**，**双向链表**以及**循环链表**等

① 单向链表的结构

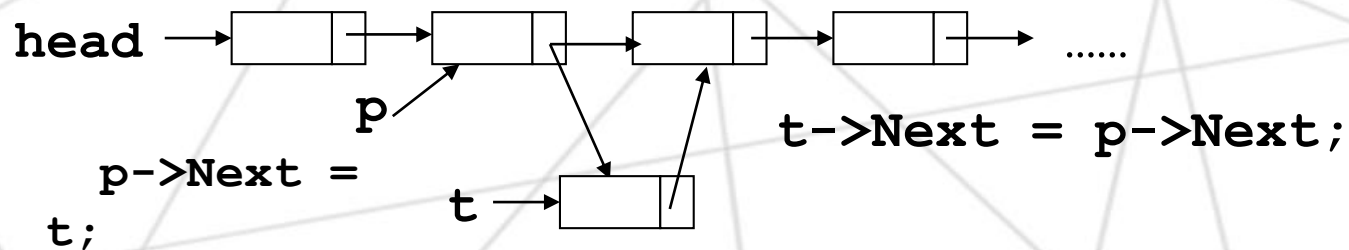


使用结构的**嵌套**来定义**单向链表结点**的数据类型。如：

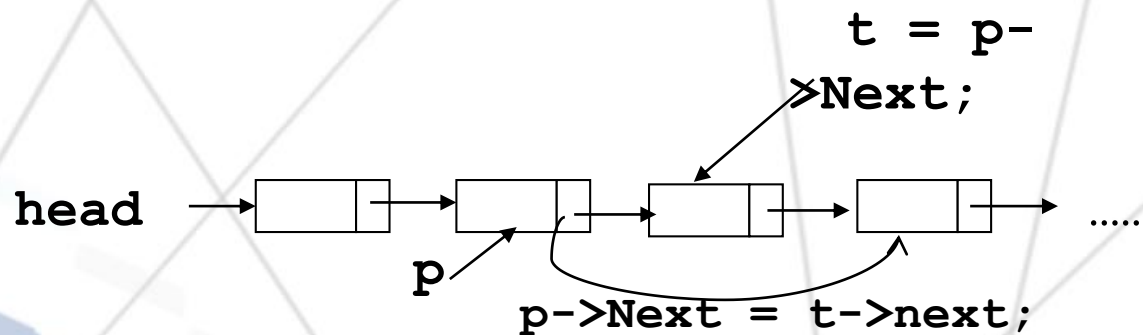
```
typedef struct Node *PtrToNode;   PtrToNode
struct Node {                     p = (PtrToNode)malloc(sizeof(struct Node));
    ElementType Data;
    PtrToNode  Next;
};                                typedef PtrToNode List; /* 定义单链表类型 */
```

② 单向链表的常见操作

(1) 插入结点 (p之后插入新结点t)



(2) 删除结点



(3) 单向链表的遍历

```
p = head;  
while (p!=NULL) {  
    .....  
    处理p所指的结点信息;  
    .....  
    p = p->Next;  
}
```

(4) 链表的建立

有两种常见的插入结点方式:

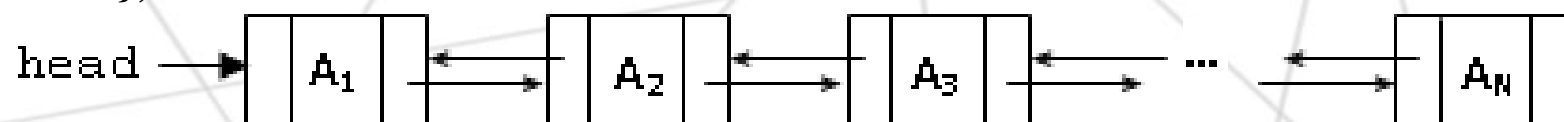
(1) 在链表的**头上**不断插入新结点;

(2) 在链表的**尾部**不断插入新结点。

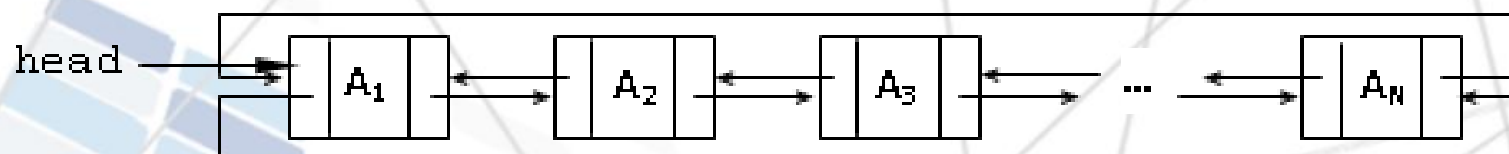
如果是后者，一般需要有一个**临时的结点指针**一直指向当前链表的最后一个结点，以方便新结点的插入。

③ 双向链表

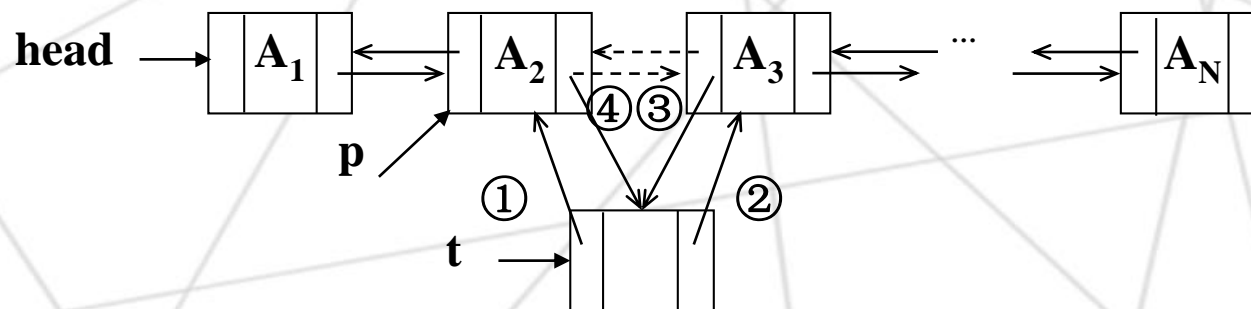
```
typedef struct DNode *PtrToDNode;
struct DNode {
    ElementType Data; /* 存储结点数据 */
    PtrToDNode Next; /* 指向下一个结点的指针 */
    PtrToDNode Previous; /* 指向前一个结点的指针 */
};
```



如果将双向链表最后一个单元的Next指针指向链表的第一个单元，而第一个单元的Previous指针指向链表的最后一个单元，这样构成的链表称为双向循环链表。



❖ 双向链表的插入、删除和遍历基本思路与单向链表相同，但需要同时考虑前后两个指针。



PtrToDNode p, t;

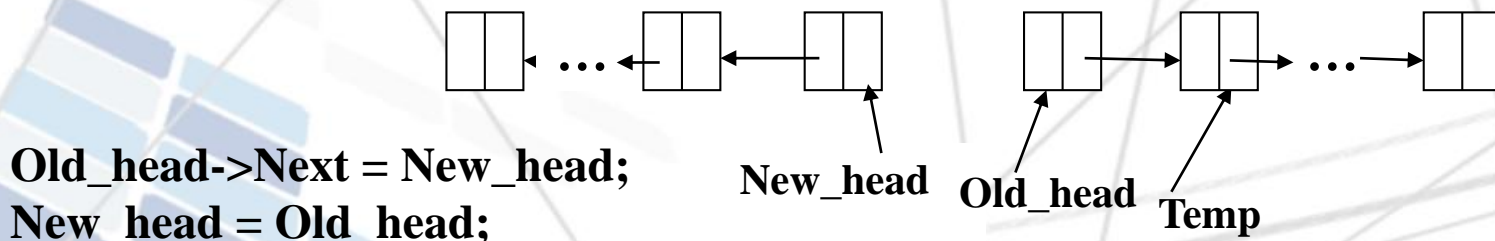
指针操作顺序:

- ① **$t \rightarrow \text{Previous} = p;$**
- ② **$t \rightarrow \text{Next} = p \rightarrow \text{Next};$**
- ③ **$p \rightarrow \text{Next} \rightarrow \text{Previous} = t;$**
- ④ **$p \rightarrow \text{Next} = t;$**

[例2.4] 给定一个单链表L，请设计函数**Reverse**将链表L就地逆转，即不需要申请新的结点，将链表的第一个元素转为最后一个元素，第二个元素转为倒数第二个元素，……。

【分析】 基本思路是：

- 利用**循环**，从链表头开始逐个处理。
- 如何把握住**循环不变式**。（循环不变式表示一种在循环过程进行时不变的性质，**不依赖于前面所执行过程的重复次数的断言**。）
- 在每轮循环开始前我们都面临两个序列，其中**Old_head**是一个待逆转的序列，而**New_head**是一个已经逆转好的序列，如下图。
- 每轮循环的目的是把**Old_head**中的第一个元素插入到**New_head**的头，使这轮循环执行好后，**Old_head**和**New_head**还是分别指向新的待逆转序列和已经逆转好的序列。



```
List Reverse( List L )
{   PtrToNode Old_head, New_head, Temp;
    Old_head = L;  /* 初始化当前旧表头为L */
    New_head = NULL; /* 初始化逆转后新表头为空 */
    while ( Old_head ) { /* 当旧表不为空时 */
        Temp = Old_head->Next;
        Old_head->Next = New_head;
        New_head = Old_head; /* 将当前旧表头逆转为新表头 */
        Old_head = Temp; /* 更新旧表头 */
    }
    L = New_head; /* 更新L */
    return L;
}
```


❖ 三种基本的控制结构是顺序、分支和循环。

- 顺序结构是一种自然的控制结构，通过安排语句或模块的顺序就能实现。
- C语言为分支控制提供了if-else和switch两类语句，
- 为循环控制提供了for、while和do-while三类语句。

- 函数定义
- 函数调用
- 函数递归



语句级控制

单位级控制



3. 函数与递归

【定义】函数是一个完成特定工作的独立程序模块。

- 只需定义一次，就可以多次调用。
- 函数包括库函数和自定义函数两种。例如，scanf、printf等库函数由C语言系统提供定义，编程时只要直接调用即可。
- 在程序设计中，往往根据模块化程序设计的需要，用户可以自己定义函数，属于自定义函数。

比如：C语言提供了实数和整数的加法运算符“+”来完成运算；但是“+”不能对复数做加法运算；可以写一个函数来实现这个功能。

先定义复数类型 `ImgType`，以约定何为复数：

```
struct Image { double r; double i; };  
typedef struct Image ImgType;
```

再定义复数的加法函数：

```
ImgType ImgAdd(ImgType a, ImgType b)  
{  
    ImgType c;  
    c.r = a.r + b.r;    c.i = a.i + b.i;  
    return c;  
}
```

有了这个函数，以后可以在任何需要计算复数加法的地方调用它！

❖ 在设计函数时，注意掌握以下原则：

(1) 函数**功能的设计原则**：结合模块的独立性原则，函数的**功能要单一**，不要设计多用途的函数，否则会降低模块的聚合度；

(2) 函数**规模的设计原则**：函数的**规模要小**，尽量控制在50行代码以内，这样可以使得函数更易于维护；

(3) 函数**接口的设计原则**：结合模块的独立性原则，函数的接口包括函数的参数（入口）和返回值（出口），**不要设计过于复杂的接口**，合理选择、设置并控制参数的数量，尽量不要使用全局变量，否则会增加模块的耦合度。

❖ 递归函数

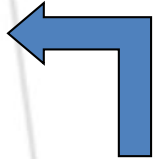
【定义】一个函数除了可以调用其他函数外，C语言还支持函数直接或间接调用自己。这种函数自己调用自己的形式称为函数的递归调用，带有递归调用的函数也称为递归函数。

❖ 两个关键点：

- (1) 递归出口：即递归的结束条件，到何时不再递归调用下去；
- (2) 递归式子：当前函数结果与准备调用的函数结果之间的关系，如求阶乘函数的递归式子：

$$Factorial(n) = n * Factorial(n-1)。$$

```
long int Factorial( int n )  
{  
    if( n == 0 ) return 1;  
    else      return n * Factorial(n-1);  
}
```



递归调用

[例2.8] 设计函数求n!

```
long int Factorial( int n )  
{  
    if( n == 0 ) return 1;  
    else return n * Factorial(n-1);  
}
```

递归出口

递归式子

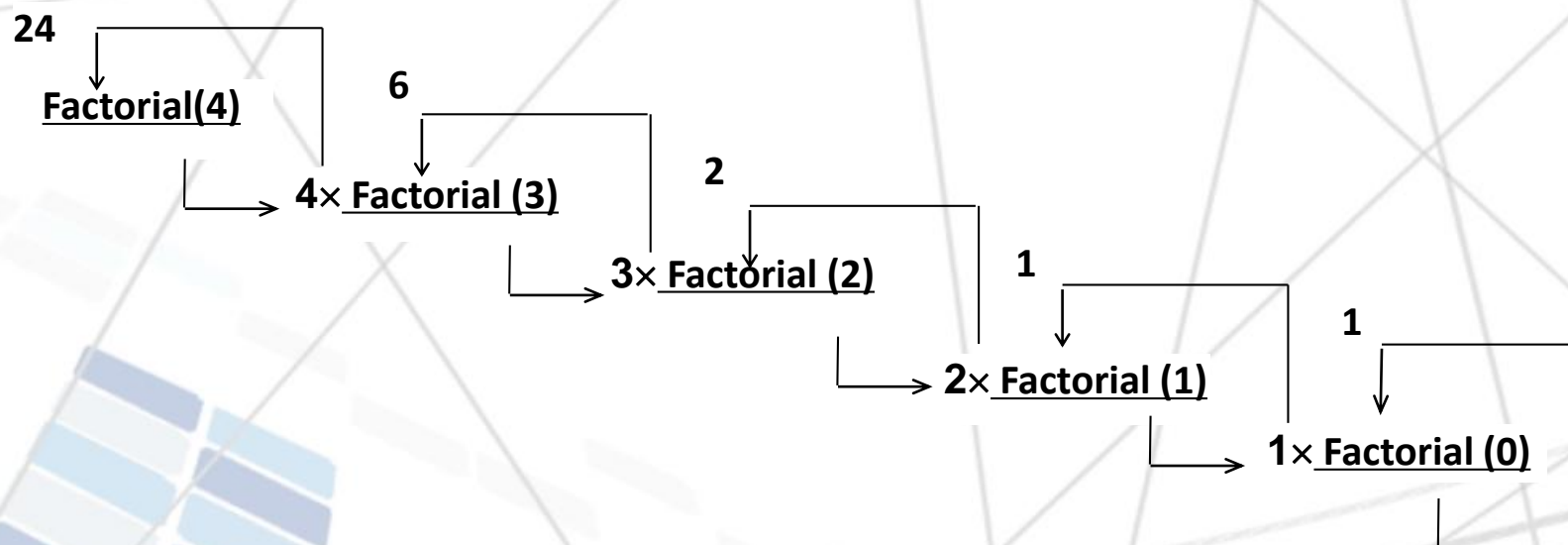
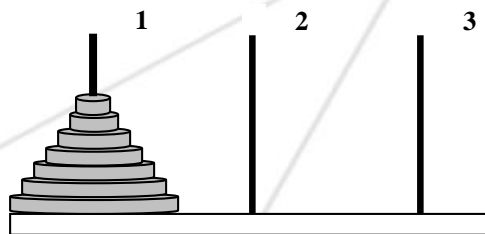
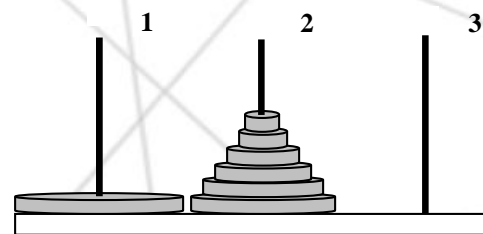


图2.7 递归求解4!的过程

[例2.9] 汉诺塔（Tower of Hanoi）问题



(a) 初始状态

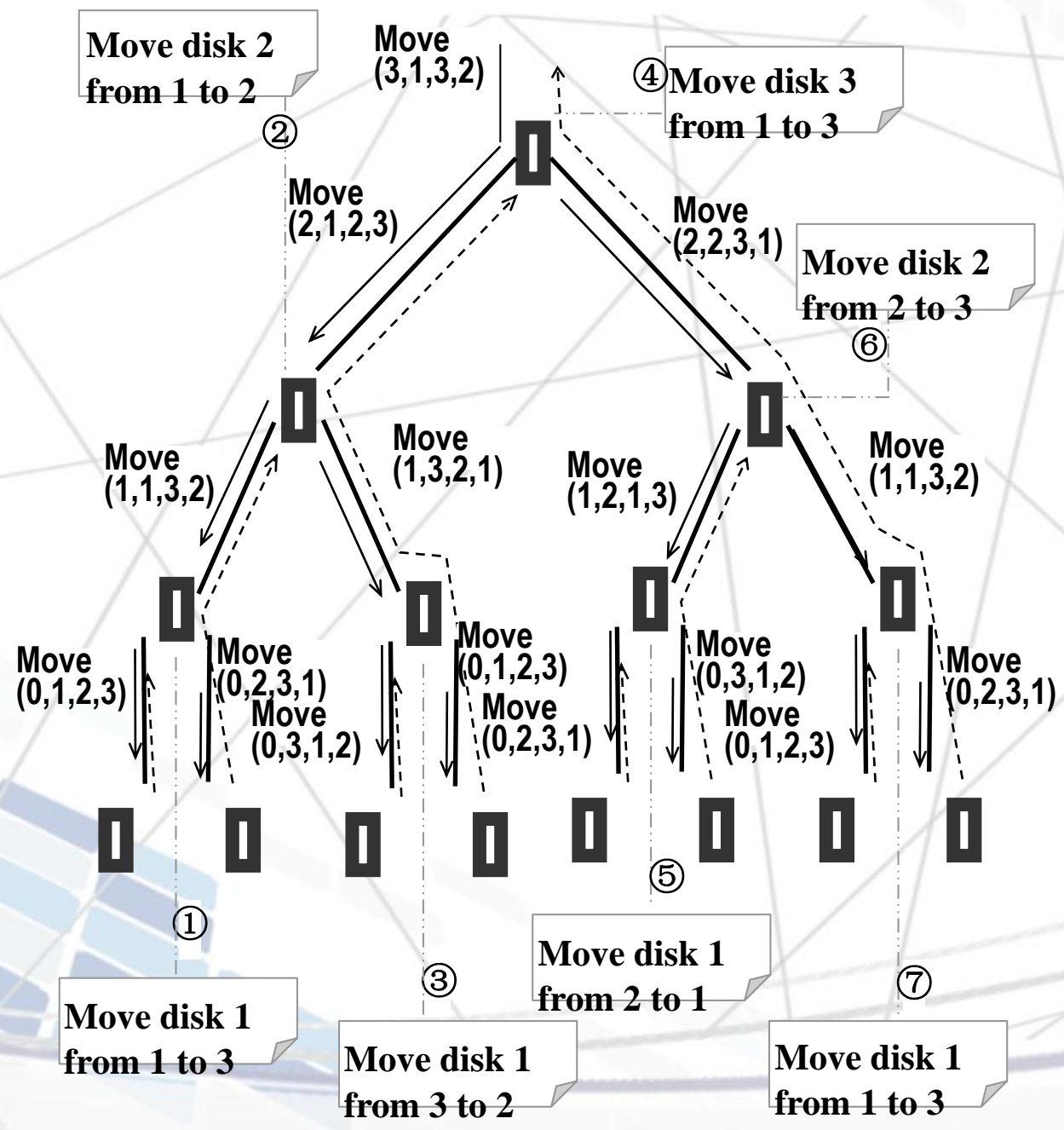


(b) 中间状态

【分析】可以用递归方法来求解汉诺塔问题，也就是将 n 个金片的移动问题转换为2个 $n-1$ 个金片的移动问题。当 $n=1$ 时，就不需要再递归了。

```
void Move(int n, int start, int goal, int temp)
{
    if( n == 0 ) return;
    Move(n-1, start, temp, goal);
    printf("Move disk %d from %d to %d.\n", n, start,
goal);
    Move(n-1, temp, goal, start);
}
```

递归调用



[例2.10] 用递归方法求集合的中位数。

❖ 根据前面求解集合第K大整数问题的递归算法思路，还需要解决以下两个关键问题：

(1) 如何根据元素e将集合S分解为S1和S2两个集合？

(2) 如何设计递归函数的参数？

```
ElementType Median( ElementType S[], int N )  
{  
    return FindKthLargest(S, (N+1)/2, 0, N-1);  
}
```

```

ElementType FindKthLargest ( ElementType S[], int K, int Left, int Right )
{ /* 在S[Left]...S[Right]中找第K大元素 */
    ElementType e = S[Left]; /* 简单取首元素为基准 */
    int L = Left, R = Right;

    while (1) { /* 将序列中比基准大的移到基准左边，小的移到右边 */
        while ( (Left<=Right)&&(e <= S[Left]) ) Left++;
        while ( (Left<Right)&&(e > S[Right]) ) Right--;
        if ( Left < Right )
            Swap ( &S[Left], &S[Right] );
        else break;
    }
    Swap ( &S[Left-1], &S[L] ); /* 将基准换到两集合之间 */
    if ( (Left-L-1) >= K ) /* (Left-L-1)代表了集合S1的大小 */
        return FindKthLargest(S, K, L, Left-2); /* 在集合S1中找 */
    else if ( (Left-L-1) < K-1 )
        return FindKthLargest(S, K-(Left-L-1)-1, Left, R); /* 在集合S2中找 */
    else
        return e; /* 找到，返回 */
}

```