



第二章 线性表、栈和队列

张铭

<http://db.pku.edu.cn/mzhang/DS/>

北京大学信息科学与技术学院
“数据结构与算法”教学小组

©版权所有，转载或翻印必究



资源提示

作业提交ftp更改

- 实验班作业提交
ftp://ds_honor:ds07honor@fusion.grids.cn
- 实习课作业提交
ftp://ds_proj:ds07proj@fusion.grids.cn

讲义和作业发布不变

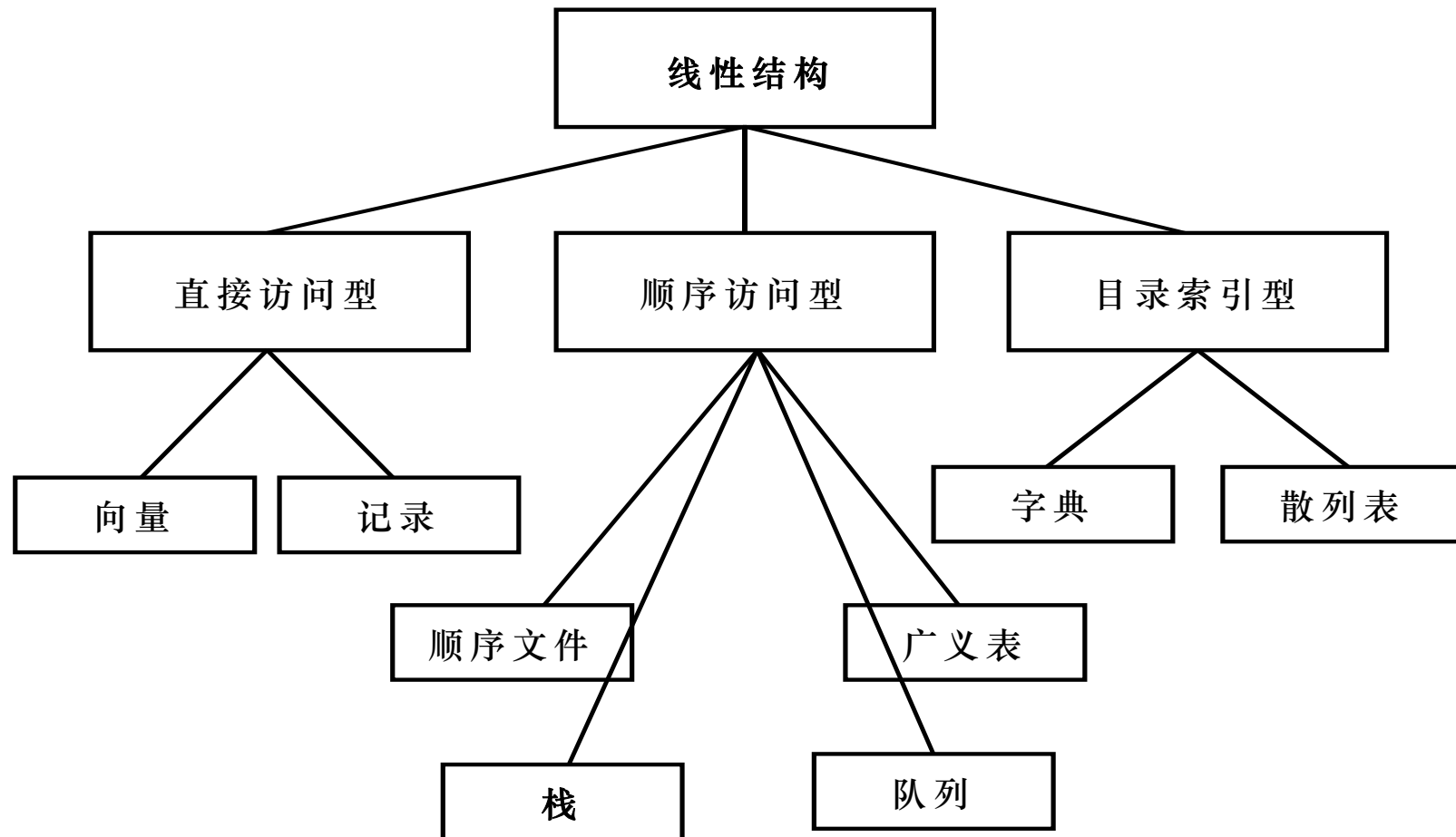
- 实验班讲义和作业发布（包括实习作业）
<http://db.pku.edu.cn/mzhang/ds/honor/>
- 实习课讲义和作业发布
<http://db.pku.edu.cn/mzhang/DS/shixi/>



大纲

- ◆ 2.1 线性表(linear list)
- ◆ 2.2 顺序表—向量(Sequential list—vector)
- ◆ 2.3 链表(Linked list)
- ◆ 2.4 线性表实现方法的比较
- ◆ 2.5 栈(Stack)
- ◆ 2.6 队列(Queue)

线性结构分类





2.1 线性表(linear list)

- 逻辑定义：由结点集 N ，以及定义在结点集 N 上的线性关系 r 所组成的线性结构
 - 结点：线性表的元素
 - 唯一的起点：没有前驱，有一个唯一的后继
 - 唯一的终点：有一个唯一的前驱而没有后继
 - 内部结点：有唯一的前驱，唯一的后继
 - 结点个数：线性表的长度
 - 线性表的关系 r ，简称前驱关系
 - 反对称性、传递性



```
template<class ELEM>
```

```
class list //线性表类模板list, 模板参数ELEM
```

```
{
```

```
    list(); // 创建一个空的新线性表
```

```
    ~list(); // 从计算机存储空间删去整个线性表
```

```
    void clear(); // 将该线性表的全部元素清除, 成为空表
```

```
    void append(ELEM value); // 尾附函数, 在表尾加新元素
```

```
    void insert(int i, ELEM value); // 插入函数, 在第i号位插入
```

```
    void remove(int i); // 删除函数, 删去第i号元素
```

```
    ELEM fetch(int i); //读取, 返回第i个元素的值
```

back

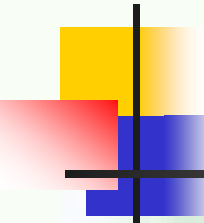
next



2.2 顺序表—向量

- 采用定长的一维数组存储结构
- 主要特性:
 - 元素的类型相同
 - 元素顺序地存储在连续存储空间中
 - 通过下标读写即可指定位置元素

顺序表类定义



```
const int Max_length = 100; // 假定最大长度为100  
class list { // 顺序表，向量  
private :  
    int msize; // 顺序表最大长度  
    int curr_len; // 顺序表当前长度  
    ELEM* nodelist; //私有变量，存储顺序表实例的向量  
public:  
    //以下列出成员函数（顺序表的函数集）  
    int curr; //当前下标，顺序表的公共变量  
    list(const int size); // constructor函数,创建新表  
    ~list(); //destructor函数，将该表实例删去  
    void clear(); //清除内容，成为空表
```




void setFirst(); // 第一个元素位置赋予当前下标**curr**

void next(); // **curr**下移一格，即**curr+1**

void prev(); //将**curr**上移一格，即**curr-1**

bool isInList(); // 若**curr**位置有值时，返回**true**

void append(const ELEM&); //在表尾增添新元素

void insert(const ELEM&); // 在当前下标**curr**插入

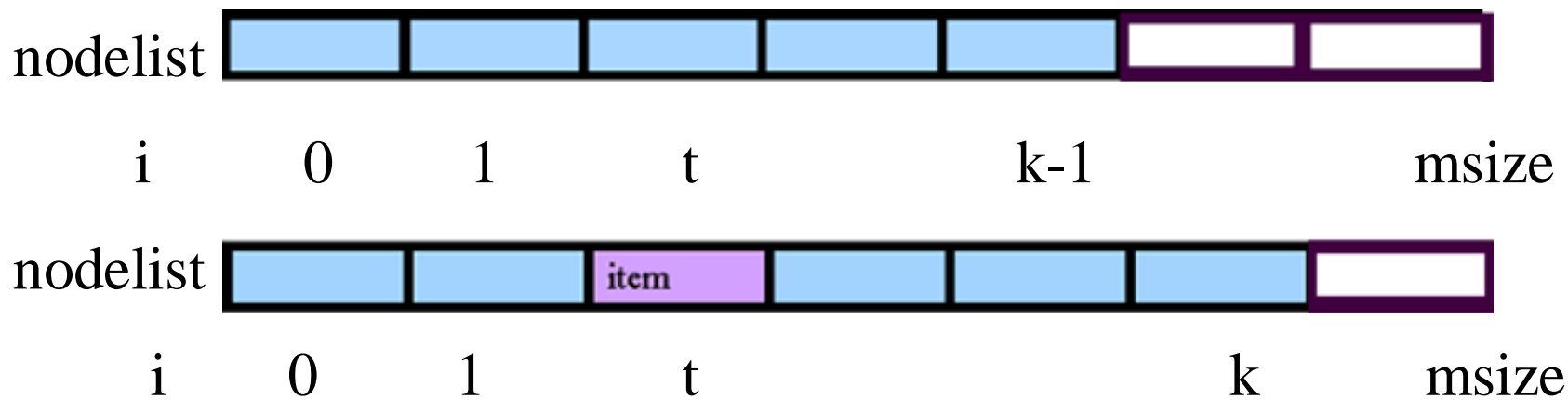
ELEM remove(); //返回**curr**的位置元素值，并删除

bool isEmpty(); //当线性表为空时，返回**true**

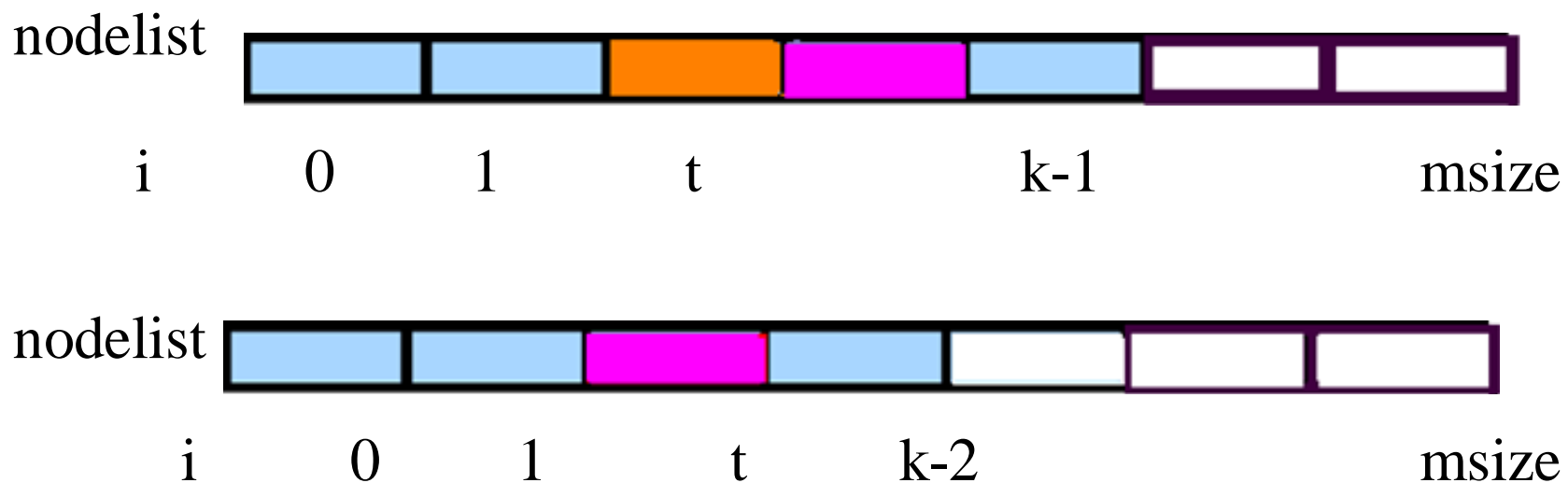
ELEM currValue(); //返回**curr**位置的元素值。

int length(); // 返回此表的当前实际长度





(a) 在 t 位置插入元素 `item`



(b) 删除 t 位置的元素



插入算法执行时间

- 元素总个数为 n ，各个位置插入的概率相等为 $p=1/n$
- 平均移动元素次数为

$$\sum_{i=0}^{n-1} 1/n \cdot (n-i) \approx \frac{n}{2}$$

- 总时间开销估计为 $O(n)$



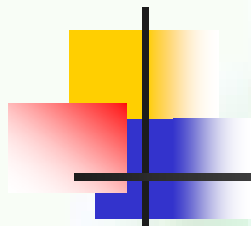
删除算法时间代价

- 与插入操作相似， $O(n)$
- 顺序表存取元素方便，时间代价为 $O(1)$
- 但插入、删除操作则付出时间代价 $O(n)$



2.3 链表(linked list)

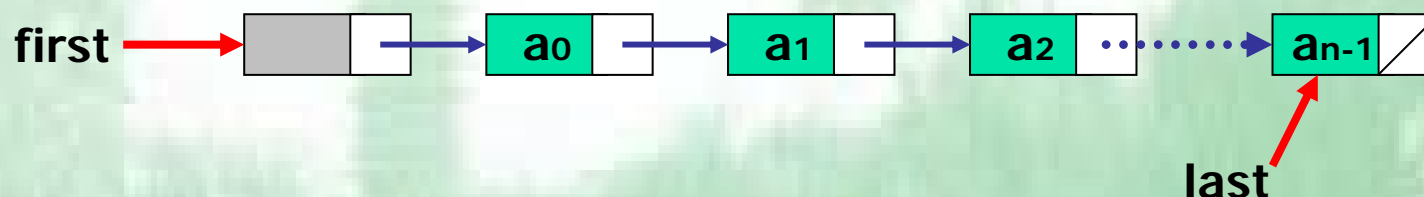
- ◆ 单链表
- ◆ 双链表
- ◆ 循环链表



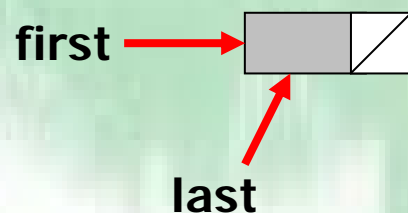
- **first**指向特殊的头结点
 - 引入头结点有利于特殊情况的处理
 - 空链表
 - 链表头
- **i**按照C/C++数组下标编号的规则
 - 从0到n-1
 - 链表的第0个结点为**first->link**

单链表的存储结构

单链表



空的单链表



data字段
link字段



单链表结点类型以及变量说明

```
struct ListNode
{
    ELEM data;
    ListNode * link;
};
typedef ListNode * ListPtr;
ListPtr first, last;
```




单链表插入算法

// 插入数据内容为value的新结点，为第i个结点。

```
ListNode * Insert(ELEM value, int i) {  
    ListNode *p, *q;  
    p = FindIndex(i-1);  
    if (p == NULL ) return NULL;  
    q = new ListNode;           // 需要时才new  
    q->link = p->link;  
    q->data = value;  
    p->link = q;  
    if(q->link == NULL )  
        last=q;  
    return q;  
}
```

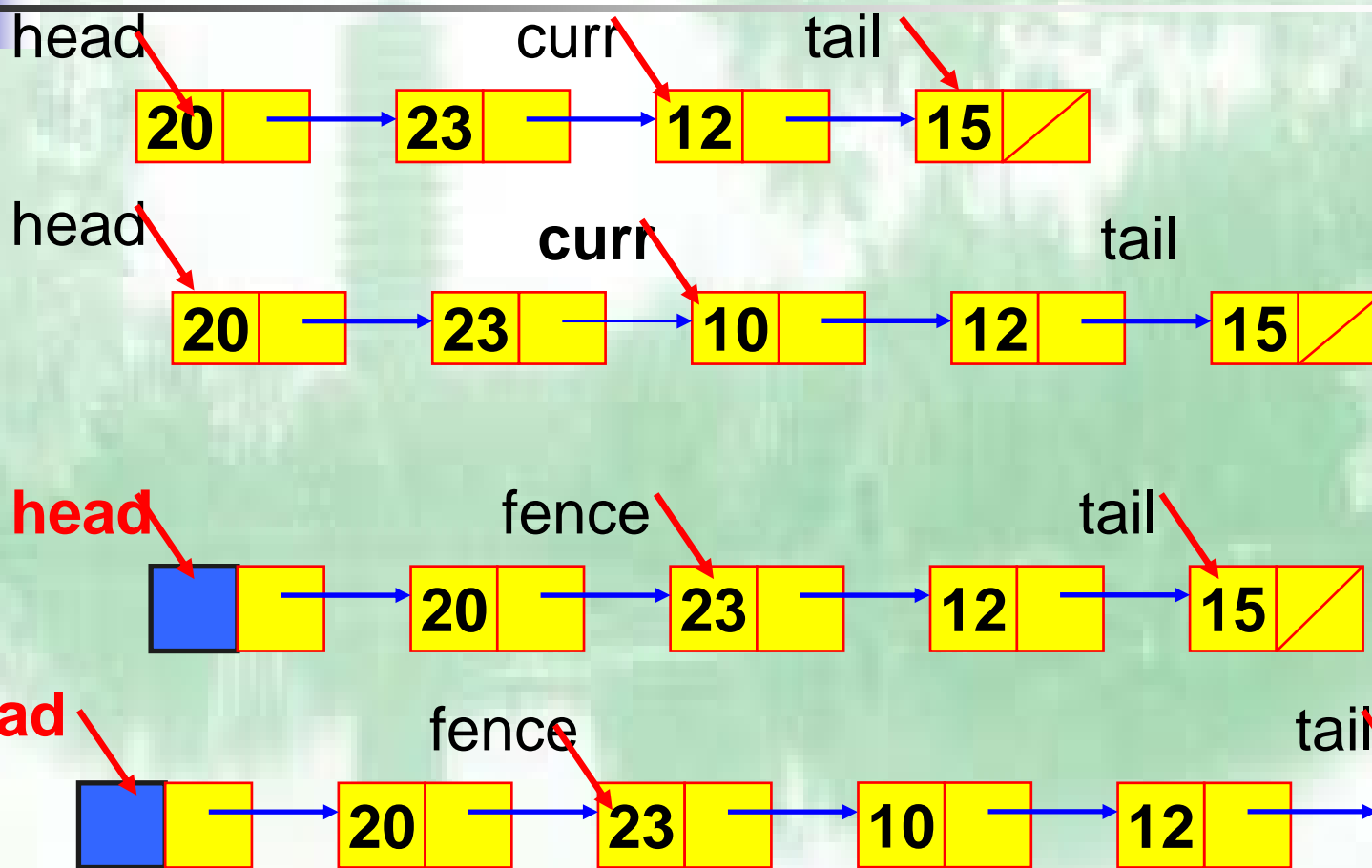
[back](#)

[next](#)

不带头结点 vs 带头结点

不带头结点

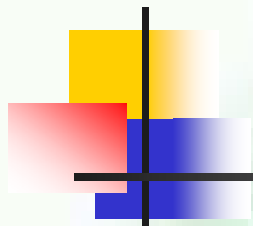
带头结点



back

next

// 不带头结点



// 带头结点

```
template <class Elem>
bool LList<Elem>::insert(const
    Elem& item) {
    fence->next = new
        Link<Elem>(item, fence->next);
    if (tail == fence)
        tail = fence->next;
    rightcnt++;
    return true;
}
```

```
template<class Elem>
bool NHList<Elem>::insert(const
    Elem& item) {
    if (head == NULL)
        head = curr = tail = new
            Link<Elem>(item, NULL);
    else {
        Link<Elem>* temp = head;
        if (temp == curr) {
            head = new
                Link<Elem>(item, head);
            curr = head;
        }
        else {
            while(temp->next != curr)
                temp = temp->next;
            temp->next = new
                Link<Elem>(item, curr);
            curr = temp->next;
        }
    }
    rightcnt++;
    return true;
}
```

back

next

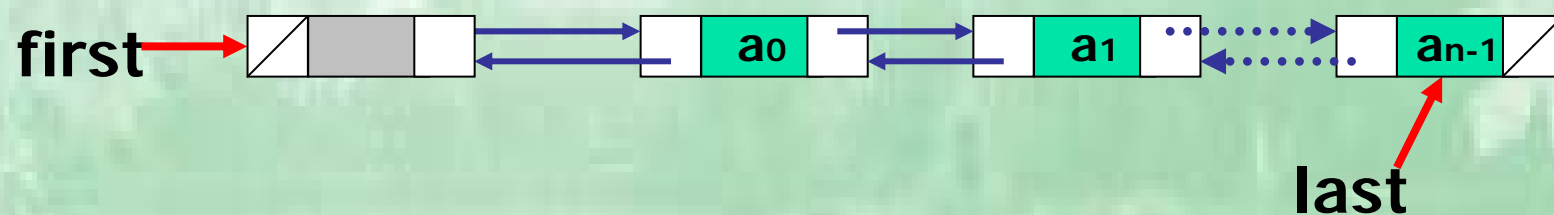


2.3.2 双链表 (double linked list)

- 单链表的主要不足之处是：
 - **link**字段仅仅指向后继结点，不能有效地找到前驱
- 双链表弥补了上述不足之处
 - 增加一个指向前驱的指针

双链表示意图

双向链表



back

next



双链表及其结点类型的说明

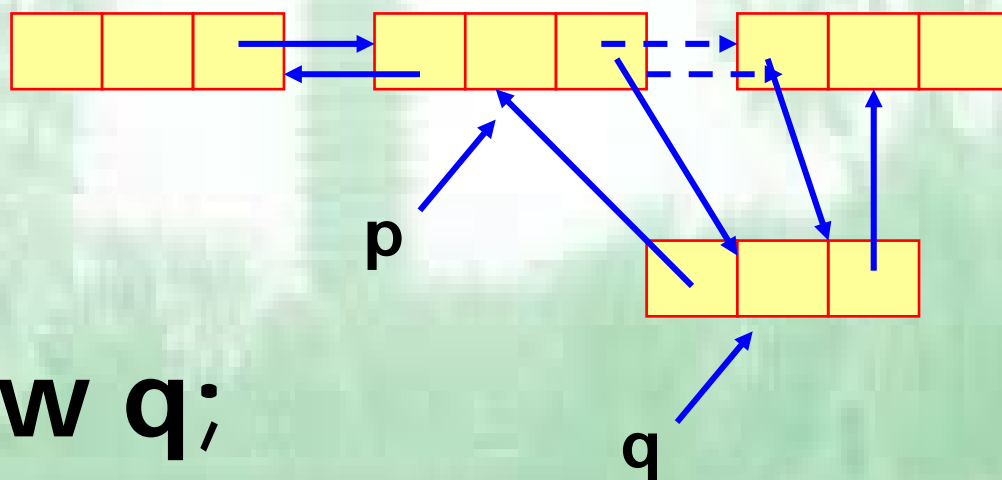
```
struct DbListNode
{
    ELEM data;
    DbListNode *rlink;
    DbListNode *llink;
};
struct DoubleList
{
    DbListNode *first, *last;
};
```

[back](#)

[next](#)

插入过程（注意顺序）

在p所指结点后面插入一个新结点



new q;

q->rlink=p->rlink

q->llink=p (注意，教材p->rlink->llink)

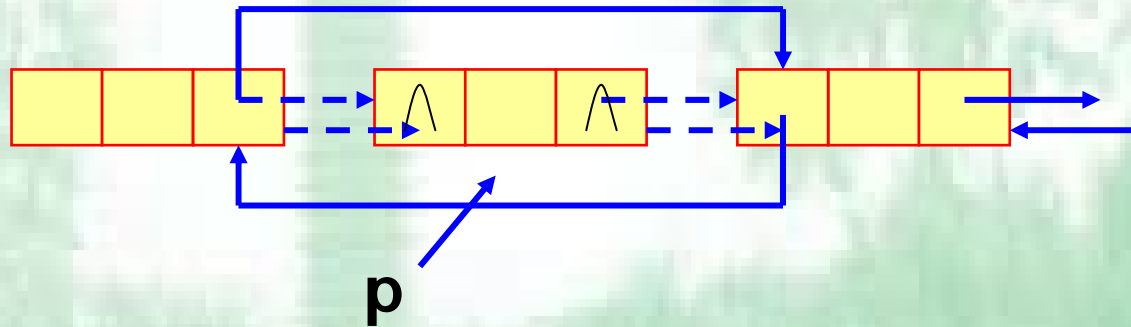
p->rlink=q

q->rlink->llink=q



删除过程

删除p所指的结点



$p \rightarrow llink \rightarrow rlink = p \rightarrow rlink$
 $p \rightarrow rlink \rightarrow llink = p \rightarrow llink$

$p \rightarrow rlink = \text{NULL}$
 $p \rightarrow llink = \text{NULL}$

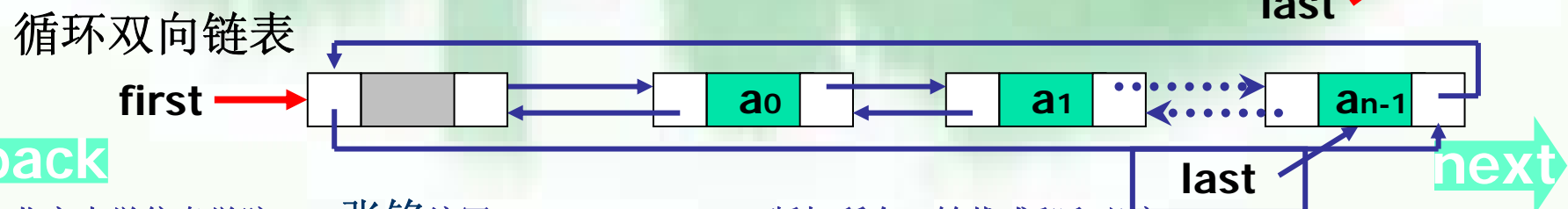
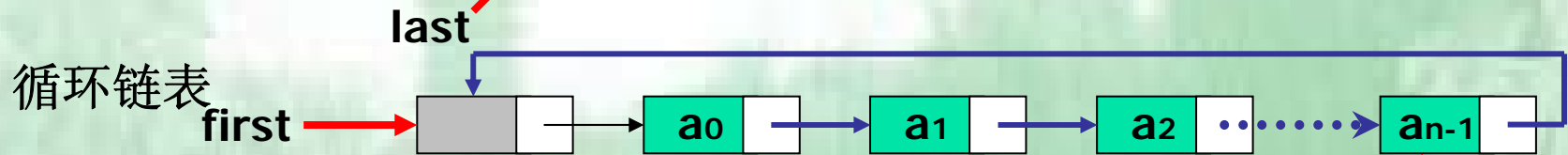
- 如果马上删除p
 - 则可以不赋空



2.3.3 循环链表 (circularly linked list)

- 单链表或双链表的头尾结点链接起来
- 给不少操作带来了方便
 - 从循环表中任一结点出发，都能访问到表中其他结点
 - 不增加额外存储花销

几种主要链表比较





注意指针变量的有效性

- 记得使用**new**
 - 使用新指针变量，或要生成一个新结点前先执行**new**操作
- 不要引用**NULL**指针
 - 使用指针前，先用**if**语句（或**while**循环条件中的语句）判断它非空
- 不用引用**Delete**了的指针



注意链表的边界处理

- 几个特殊点的处理
 - 头指针处理
 - 非循环链表尾结点的指针域保持为NULL
 - 循环链表尾结点的指针回指头结点
- 链表处理
 - 空链表的特殊处理
 - 插入或删除结点时指针勾链的顺序
 - 指针移动的正确性
 - 插入
 - 查找或遍历



2.4 线性表实现方法的比较

- 顺序表的主要优点
 - 没用使用指针，不用花费附加开销
 - 线性表元素的读访问非常简洁便利
- 链表的主要优点
 - 无需事先了解线性表的长度
 - 允许线性表的长度有很大变化
 - 能够适应经常插入删除内部元素的情况



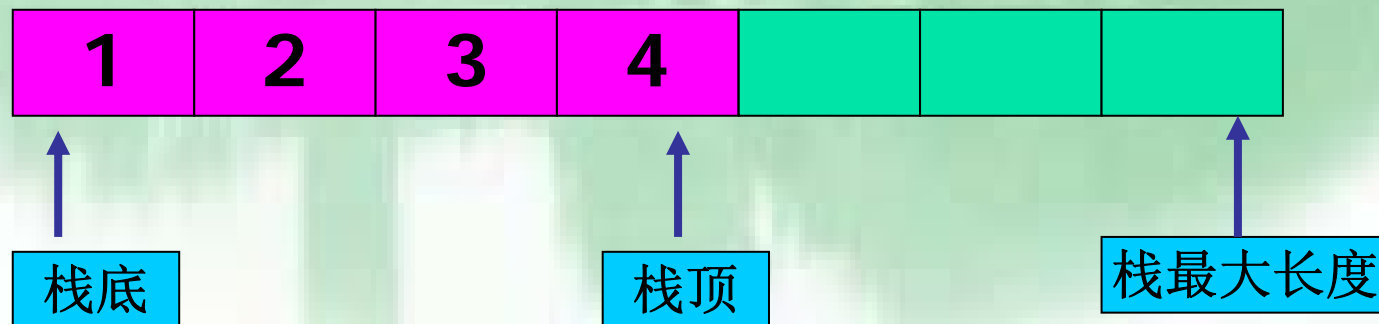
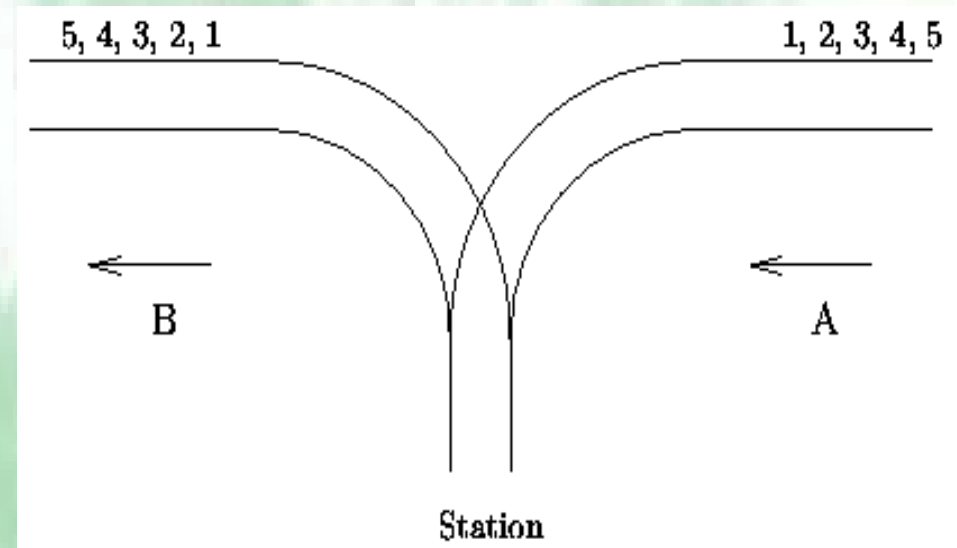
应用场合的选择

- 不要使用顺序表的场合
 - 经常插入删除时，不宜使用顺序表
 - 线性表的最大长度也是一个重要因素
- 不要使用链表的场合
 - 当读操作比插入删除操作频率大时，不应选择链表
 - 当指针的存储开销，和整个结点内容所占空间相比其比例较大时，应该慎重选择

2.5 栈(stack)

栈：限制访问端口的线性表
LIFO表

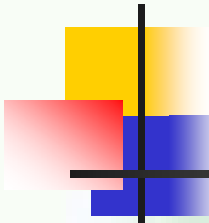
- Push: 元素插入，‘压入’
- Pop: 元素删除，‘弹出’
- Top: 表首被称为‘栈顶’



back

next

栈的抽象数据类型



```
template <class ELEM> class Stack
{ // 栈的运算集为:
    Stack(int sz);      //创建栈的实例
    ~Stack();          //该实例消亡
    void Push(ELEM item); // item压入栈顶
    ELEM Pop();        // 返回栈顶内容，并从栈顶弹出
    ELEM GetTop();      // 返回栈顶内容，但不弹出
    void ClearStack();  // 变为空栈MakeEmpty();
    bool IsEmpty();     // 若栈已空返回真
    bool IsFull();      // 若栈已满返回真，顺序栈有用
    int length();       // 当前栈长
```

 **back** };

 **next**



栈的实现

顺序栈

- 使用向量实现

链式栈

- 用单链表方式存储，其中指针的方向是从栈顶向下链接

栈的应用--计算表达式的值

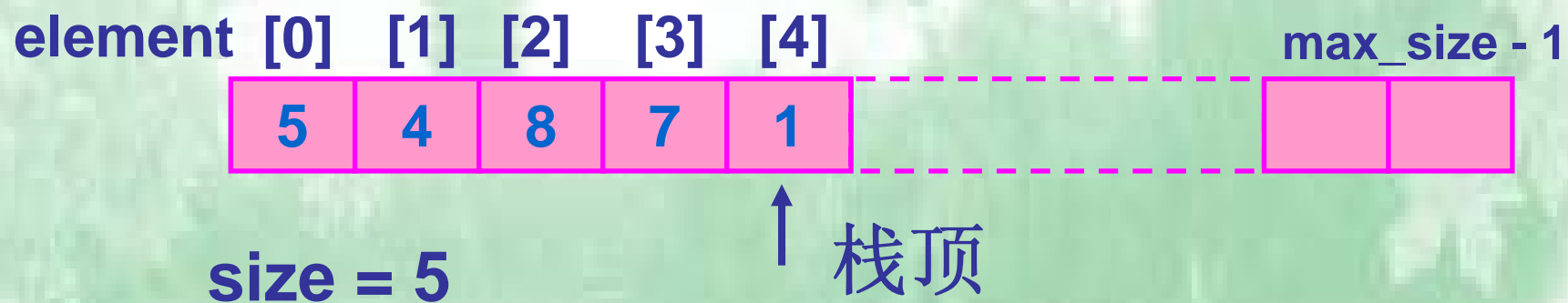
栈与递归



2.5.1 顺序栈

```
template <class ELEM>
class Stack
{
private:
    ELEM *ElmList; // 存放数据元素数组
    int top;        // 栈顶在的位置，即下标值
    int maxsize;    // 栈的最大长度
    //构建栈的实例，向量空间长度为size
Public:
    Stack(int size);
    // 其他运算同栈ADT
    ...
};
```

顺序栈





顺序栈的创建

```
//栈实例的创建，指定最大长度100。在类的内部实现
Stack(int size=100)
{
    maxsize=size;
    //开辟向量存储空间
    ElmList=new ELEM[maxsize];
    //判断new命令成功否，否则断言程序异常
    assert(ElmList!=NULL);
    top=-1;    // 表示栈空
}
```



压入栈顶

```
template <class ELEM>  
void Stack<ELEM>::Push(ELEM item)  
{  
    //判非栈满，否则栈溢出，退出运行  
    assert(!IsFull());  
    top++;    //栈顶  
    ElmList[top]=item;  
}
```



从栈顶弹出

```
template <class ELEM>  
ELEM Stack<ELEM>::Pop()  
{  
    //判栈非空，否则断言栈空异常退出运行  
    assert(!IsEmpty());  
    return ElmList[top--];  
}
```



从栈顶读取，但不弹出

```
template <class ELEM>  
ELEM Stack<ELEM>::GetTop()  
{  
    //判栈非空，否则断言栈空异常退出  
    assert(!IsEmpty());  
    return ElmList[top];  
}
```



其他函数

- // 函数在类定义之内实现, 不用“::”修饰
- **bool IsEmpty()** // 返回真, 若栈已空
 { **return top == -1;**}
- // 栈满时, 返回非零值 (真true)
 bool IsFull() {return top==maxsize-1;}
- **ClearStack() { top=-1; }** // 变空栈
- **~Stack() {delete []ElmList;}** // 栈消亡

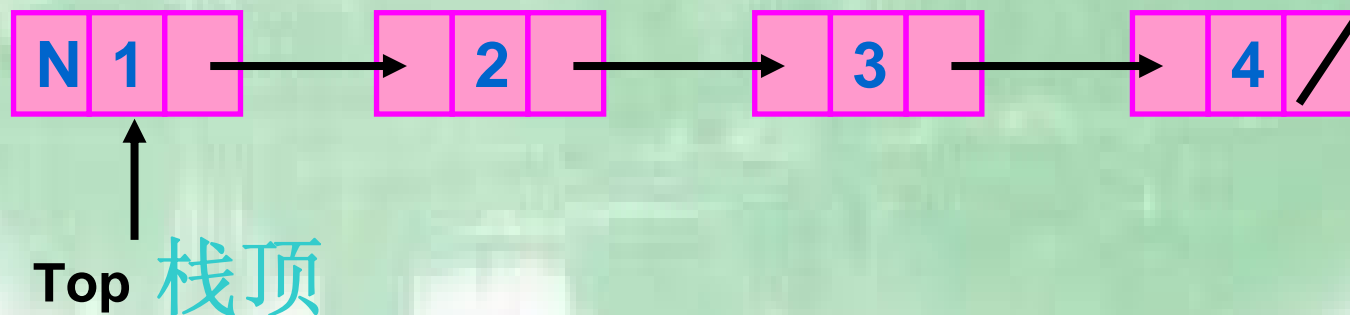


STL中关于堆栈的函数

- 其中**top**函数表示取栈顶元素，并将结果返回给用户
- **pop**函数表示将栈顶元素弹出（如果栈不空的话）
 - **pop**函数仅仅是一个操作，并不将结果返回。
 - **pointer=aStack.pop()**？错误！
- 有些库函数提供了这样的函数**ptop**？
- **STL为什么这两个操作分开？**
 - 主要是概念上显得清晰
 - 保证一个函数只确定地完成一项特定的功能
 - 使得函数之间的耦合度降低

2.5.2 链式栈

- 用单链表方式存储
- 指针的方向从栈顶向下链接

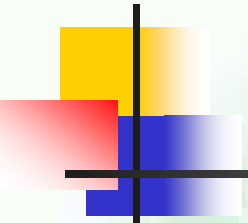




单链表的结点类型

```
// Singly-linked list node
template <class Elem> class Link {
public:
    Elem data;    // Value for this node
    Link *next;   // Pointer to next node in list
    Link(const Elem& elemval, Link* nextval =NULL)
        { element = elemval; next = nextval; }
    Link(Link* nextval =NULL) { next = nextval; }
};
```

链式栈的创建



```
template <class ELEM>
class Stack {
private:
    Link <ELEM> *top;
    int maxsize;           // 最大栈长, 链栈可以不要
    int curr_len;          // 当前栈长
public:
    Stack(int size=100) { //构建栈的实例
        //创建一个空栈, 链式栈不用指定最大长度
        top = NULL;    curr_len = 0;
        maxsize = size;
    }
    ...
};
```

back

next



压入栈顶

```
template <class ELEM>
void Stack<ELEM>::Push(ELEM item)
{
    Link <ELEM> *temp;
    temp = new Link<ELEM>;
    //若无存储空间则异常，程序退出运行
    assert(!temp==NULL);
    temp->data = item;
    temp->next = top;    //老栈顶指针
    top = temp;          //新栈顶指针
    curr_len++;
}
```

 back

next 

自单链栈弹出

```
template <class ELEM>
ELEM Stack<ELEM>::Pop() {
    //判栈非空，否则断言栈空异常，程序退出
    assert(!IsEmpty());
    ELEM result = top->data; //暂存栈顶内容
    Link <ELEM> *temptr;
    temptr = top;           //老栈顶指针
    top = top->next;        //新栈顶指针
    curr_len--;
    delete temptr;         //释放空间
    return result;         //返回的是弹出内容
}
```

back

next



小结

- 实际应用中，顺序栈比链式栈用得更为广泛些
 - 一般来说，栈不允许“读取内部元素”，只能在栈顶操作



计算机过时技能TOP10

- 1.Cobol 、 2. Nonrelational DBMS
(非关系数据库管理系统)
- 3. Non-IP networks (非IP网络)、 4. CC:Mail
- 5. ColdFusion
- **6. C 语言设计**
- 7. PowerBuilder、 8. CNE
(NetWare认证工程师)
- 9. PC网络管理员、 10. OS/2

[back](#)

[next](#)



抽象数据类型

- 抽象数据类型
 - 实质为“逻辑结构 + 运算/操作”
 - 隐蔽了存储实现细节
 - 上层算法以一致的形式调用底层数据结构
- 例如在**STL C++**中
 - **Stack.push(x)**
 - 顺序栈，链式栈
 - 上层调用语句不需要改变



C数据类型的问题

- 1. 在同一个程序中，如果栈的 **StackElementType** 不同
 - 需要定义不同的栈；
- 2. 从顺序栈改为链式栈
 - 上层调用语句一定要改变



2.5.3 栈的应用--计算表达式的值

- 栈可以应用于递归函数 (recursive function) 的实现
- 使用下推表（栈）自动进行复杂的算术表达式的递归求值



计算表达式的值

- 表达式的递归定义
 - 基本符号集: $\{0, 1, \dots, 9, +, -, *, /, (,)\}$
 - 语法成分集: $\{\langle \text{表达式} \rangle, \langle \text{项} \rangle, \langle \text{因子} \rangle, \langle \text{常数} \rangle, \langle \text{数字} \rangle\}$
 - 语法公式集
- 后缀表达式
- 后缀表达式求值



语法公式(中缀表达法)

BNF范式，可以利用Lex, Yacc等自动构造编译器

- $\langle \text{表达式} \rangle ::= \langle \text{项} \rangle + \langle \text{项} \rangle \mid \langle \text{项} \rangle - \langle \text{项} \rangle \mid \langle \text{项} \rangle$
- $\langle \text{项} \rangle ::= \langle \text{因子} \rangle * \langle \text{因子} \rangle \mid \langle \text{因子} \rangle / \langle \text{因子} \rangle \mid \langle \text{因子} \rangle$
- $\langle \text{因子} \rangle ::= \langle \text{常数} \rangle \mid (\langle \text{表达式} \rangle)$
- $\langle \text{常数} \rangle ::= \langle \text{数字} \rangle \mid \langle \text{数字} \rangle \langle \text{常数} \rangle$
- $\langle \text{数字} \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

后缀表达式

- $\langle \text{表达式} \rangle ::= \langle \text{项} \rangle \mid \langle \text{项} \rangle + \langle \text{项} \rangle \mid \langle \text{项} \rangle - \langle \text{项} \rangle$
- $\langle \text{项} \rangle ::= \langle \text{因子} \rangle \mid \langle \text{因子} \rangle * \langle \text{因子} \rangle \mid \langle \text{因子} \rangle / \langle \text{因子} \rangle$
- $\langle \text{因子} \rangle ::= \langle \text{常数} \rangle$
- $\langle \text{常数} \rangle ::= \langle \text{数字} \rangle \mid \langle \text{数字} \rangle \langle \text{常数} \rangle \mid \langle \text{数字} \rangle . \langle \text{常数} \rangle$
- $\langle \text{数字} \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

栈的应用

——后缀表达式求值

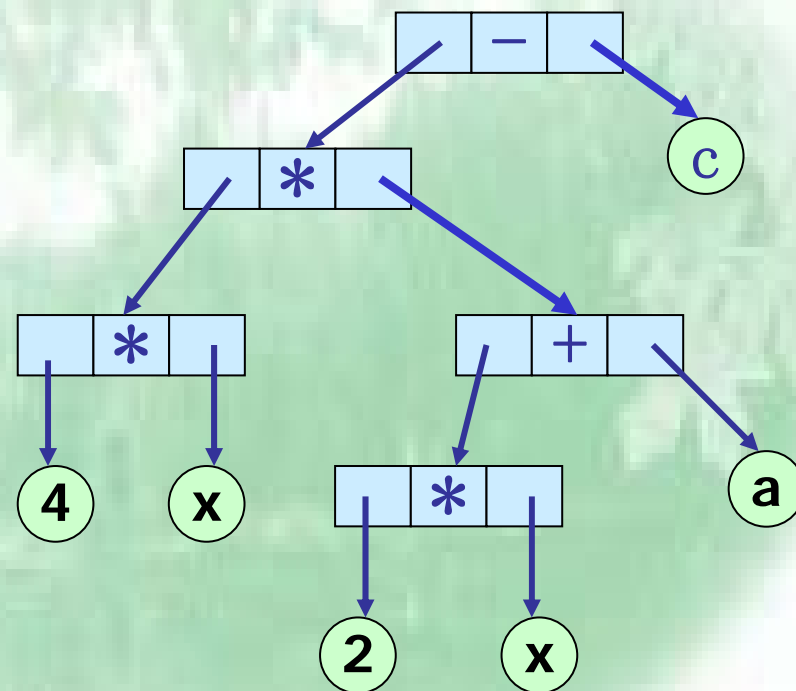
■ 中缀表达式:

- 运算符在中间
- 需要括号改变优先级
- $4 * x * (2 * x + a) - c$

■ 后缀表达式

- 运算符在后面
- 完全不需要括号

■ $4 \ x \ * \ 2 \ x \ * \ a \ + \ * \ c \ -$





中缀表达的算术表达式的计算次序

- (1) 先执行括号内的计算，后执行括号外的计算。在具有多层括号时，按层次反复地脱括号，左右括号必须配对。
- (2) 在无括号或同层括号时，先乘($*$)、除($/$)，后作加($+$)、减($-$)。
- (3) 在同一个层次，若有多个乘除($*$ 、 $/$)或加减($+$ 、 $-$)的运算，那就按自左至右顺序执行。



中缀转后缀表达式值示例

- $4 * x * (2 * x + a) - c$

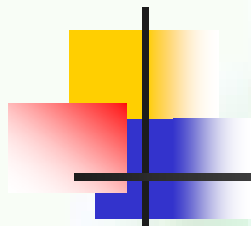
- $4\ x\ *\ 2\ x\ *\ a\ +\ *\ c\ -$

- 运算符可以作为中缀输入的自然分割
- 输出的后缀表达式，可以采用空格分割
- **cin**按照变量指定的格式输入



中缀转后缀算法

- (1) 当输入的是操作数时，直接输出到后缀表达式序列。
- (2) 当输入的是开括号时，也把它压栈。
- (3) 当输入的是闭括号时，先判断栈是否为空，若为空（括号不匹配），应该当错误异常处理，清栈退出。若非空，则把栈中的元素依次弹出，直到遇到第一个开括号为止，将弹出的元素输出到后缀表达式的序列中（弹出的开括号不放到序列中），若没有遇到开括号，说明括号也不匹配，做异常处理，清栈退出。



- (4) 当输入的是运算符时
 - (a) 循环，当（栈非空 **and** 栈顶不是开括号 **and** 栈顶运算符的优先级不低于输入的运算符的优先级）时，反复操作
将栈顶元素弹出，放到后缀表达式序列中；
 - (b) 把输入的运算符压栈。
- (5) 最后，当中缀表达式的符号序列全部读入时，若栈内仍有元素，把它们全部依次弹出，都放到后缀表达式序列尾部。若弹出的元素遇到开括号时，则说明括号不匹配，做错误异常处理，清栈退出。



后缀表达式求值

- 循环：依次顺序读用户键入的符号序列，组成并判别语法成分的类别
 - 1. 当遇到的是一个操作数，则压入栈顶；
 - 2. 当遇到的是一个运算符，就从栈中两次取出栈顶，按照运算符对这两个操作数进行计算。然后将计算结果压入栈顶。
- 如此继续，直到遇到符号=，这时栈顶的值就是输入表达式的值。

栈的应用

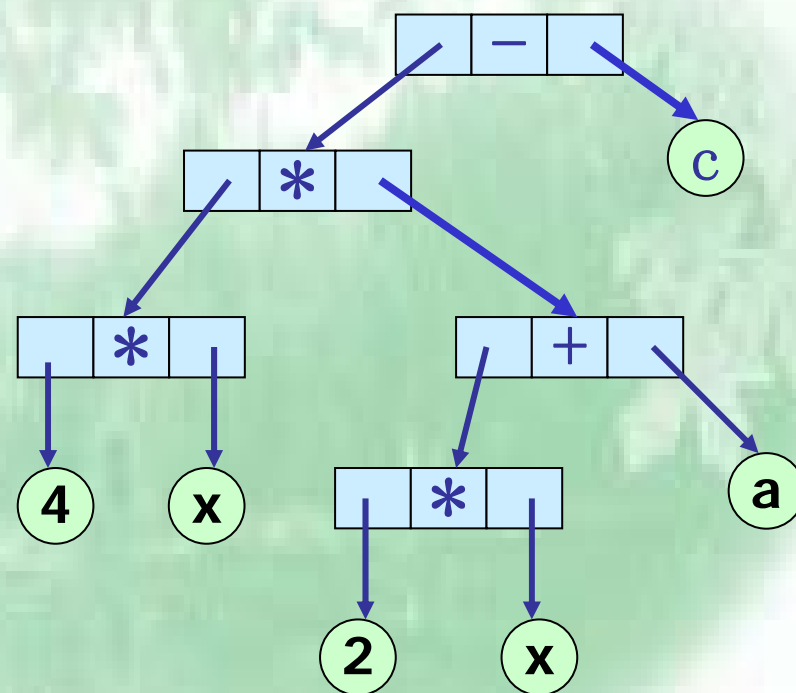
——后缀表达式求值

■ 中缀表达式:

- 运算符在中间
- 需要括号改变优先级
- $4 * x * (2 * x + a) - c$

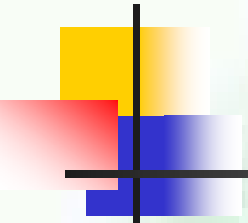
■ 后缀表达式

- 运算符在后面
- 完全不需要括号
- $4 \ x \ * \ 2 \ x \ * \ a \ + \ * \ c \ -$



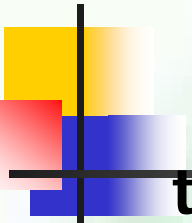
后缀计算器的类定义

```
template <class ELEM>
class Calculator
{
private:
    Stack <ELEM> S; // 这个栈用于压入保存操作数
    void Enter(ELEM num); // 把一个浮点数num压入栈
    // 从栈顶弹出两个操作数，赋值给变参opnd1和opnd2
    bool GetTwoOperands(ELEM& opnd1, ELEM& opnd2);
    // 调用GetTwoOperands，并按op运算对两个操作数进行计算
    void Compute(char op);
public:
    Calculator(void) {} ; // 创建计算器实例，开辟一个空栈
    void Run(void);        // 读入，遇到“=”时，启动求值计算
    void Clear(void);      // 清除，为随后的下一次计算做准备
};
```

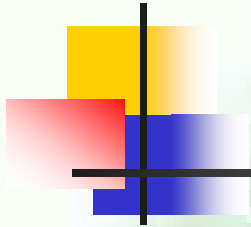


```
template <class ELEM>
bool Calculator<ELEM>::GetTwoOperands(ELEM&
    opnd1,ELEM& opnd2)
```

```
{
    if(S.IsEmpty())
    {
        cerr << "Missing operand!" <<endl;
        return false;
    }
    opnd1=S.Pop();//右操作数
    if (S.IsEmpty())
    {
        cerr << "Missing operand!" <<endl;
        return false;
    }
    opnd2=S.Pop();//左操作数
    return true;
}
```



```
template <class ELEM>
void Calculator<ELEM>::Compute(char op)
{
    bool result;
    ELEM operand1,operand2;
    result=GetTwoOperands(operand1,operand2);
    if (result==true)
        switch(op) {
            case '+' : S.Push(operand2 + operand1);
                       break;
            case '-' : S.Push(operand2 - operand1);
                       break;
```

```
case '*' : S.Push(operand2 * operand1);  
          break;  
case '/' : if (operand1 == 0.0) {  
            cerr << "Divide by 0!" << endl;  
            S.ClearStack();  
          }  
          else  
            S.Push(operand2 / operand1);  
          break;  
        }  
      else  
        S.ClearStack();  
    }  
  }
```



```

template <class ELEM>
void Calculator<ELEM>::Run(void) {
    char c;    ELEM newoperand;
    while (cin >> c, c!= '=') {
        switch(c) {
            case '+':    case '-':    case '*':    case '/':
                Compute(c);    break;
            default :
                cin.putback(c);
                cin >> newoperand;
                Enter(newoperand);
                break;
        }
    }
    if(!S.IsEmpty())
        cout << S.Pop() << endl;//印出求值的最后结果
}

```

back

next

2.5.4 栈与递归 (recursion with stack)

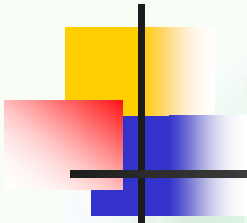
- 函数的递归定义
- 主程序和子程序的参数传递
- 栈在实现函数递归调用中所发挥的作用



递归定义阶乘 $n!$ 函数

- 阶乘 $n!$ 的递归定义如下:

$$factiorial(n) = \begin{cases} 1 & \text{当 } n \leq 0 \\ n \times factiorial(n-1) & \text{当 } n > 0 \end{cases}$$



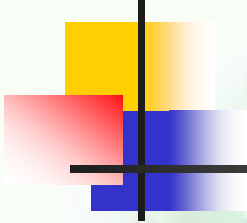
计算阶乘 $n!$ 的两个程序

//使用循环迭代方法，计算阶乘 $n!$ 的程序

```
long factorial(long n)
{
    int m = 1;
    int i;
    if (n>0)
        for ( i = 1; i <= n; i++ )
            m = m * i;
    return m;
}
```

[back](#)

[next](#)



//递归定义的计算阶乘 $n!$ 的函数

long factorial(long n)

{

if (n==0)

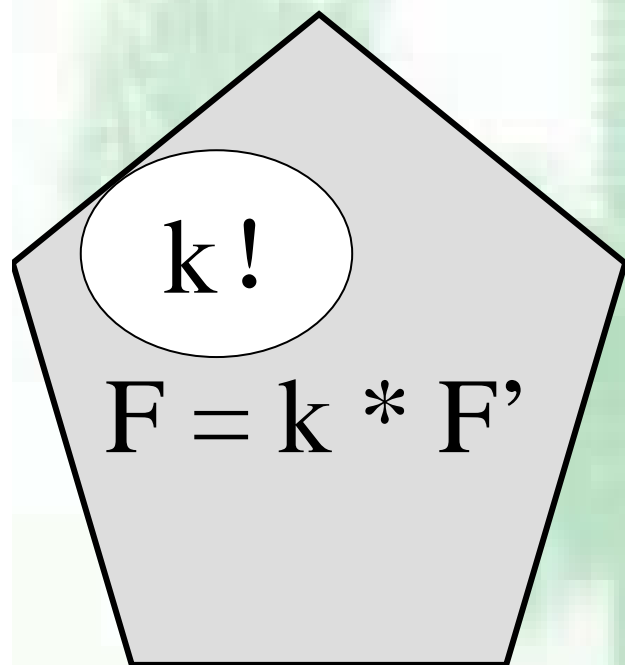
return 1;

else

return n * factorial(n-1) ; //递归调用

}

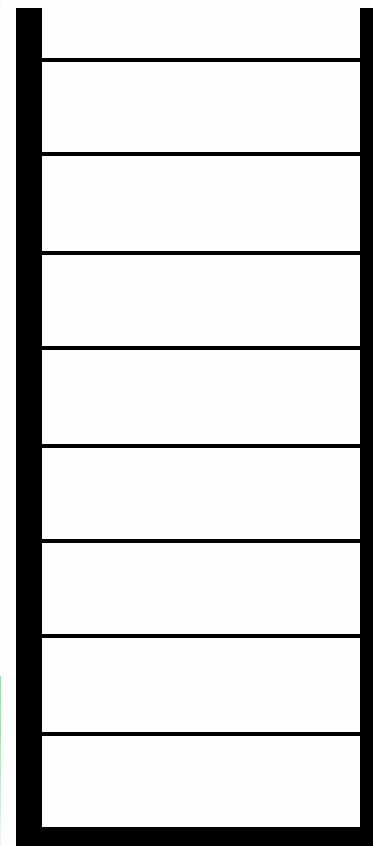
用5个F机器来模拟 $4!$ 的计算



F机器

计算乘积

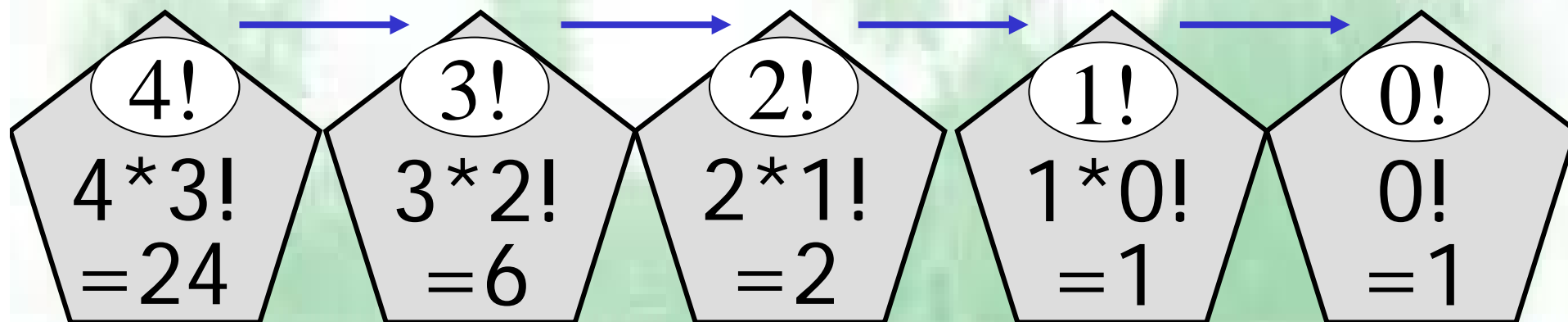
通过内部栈与其他
F机器交换信息



back

next

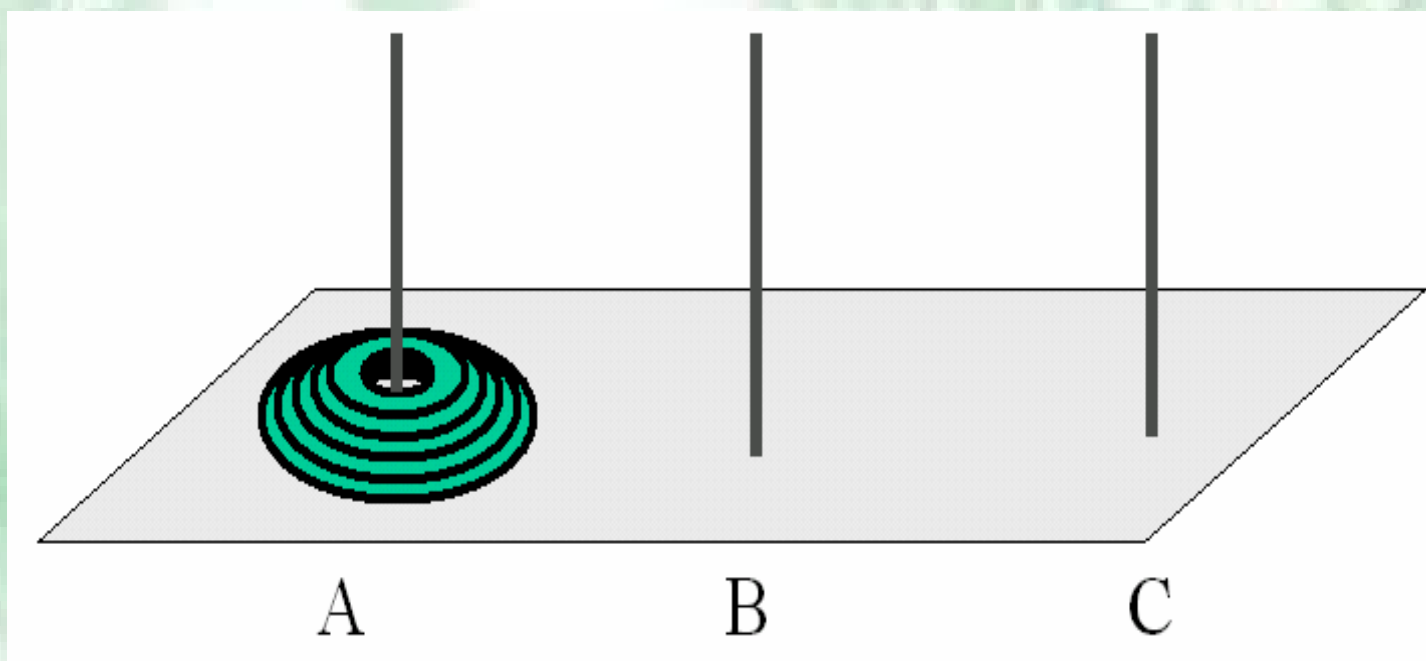
递归调 递归调 递归调 递归调
子程序 子程序 子程序 子程序



结果返回 结果返回 结果返回 结果返回

$\text{factorial}(4) = 24$

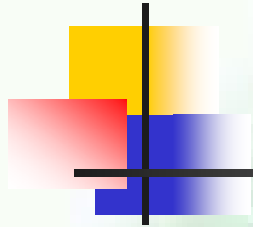
Hanoi塔问题的递归求解





河内塔问题的递归求解程序

- **hanoi(n,X,Y,Z)**
 - 移动n个槃环
 - X柱出发，将槃环移动到Z 柱
 - X、Y、Z都可以暂存
 - 大盘不能压小盘
- 例如， **hanoi(2, 'B', 'C', 'A')**
 - B柱上部的2个环槃移动 到A 柱



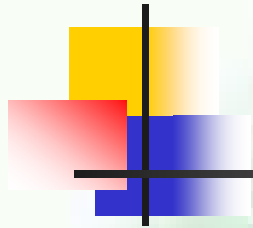
**//move(char X, char Y)子程序，表示移动一步，
//输出打印，把柱X的顶部环槃移到柱Y**

void move(char X, char Y)

{

**cout << " move " << X << "to" << Y <<
endl;**

}



```
void hanoi(int n, char X, char Y, char Z)
```

```
{
```

```
    if (n <= 1)
```

```
        move(X,Z);
```

```
    else
```

```
    {
```

```
        // 最大的环槃在X上不动，把X上的n-1个环槃移到Y
```

```
        hanoi(n-1,X,Z,Y);
```

```
        move(X,Z);           //移动最大环槃到Z，放好
```

```
        hanoi(n-1,Y,X,Z);    //把 Y上的n-1个环槃移到Z
```

```
    }
```

```
}
```



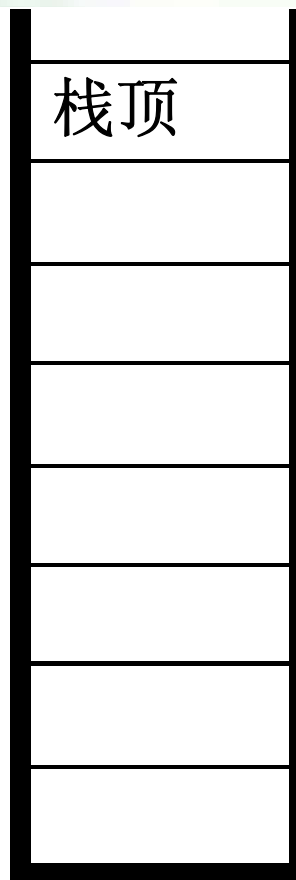
Hanoi递归子程序的运行示意图

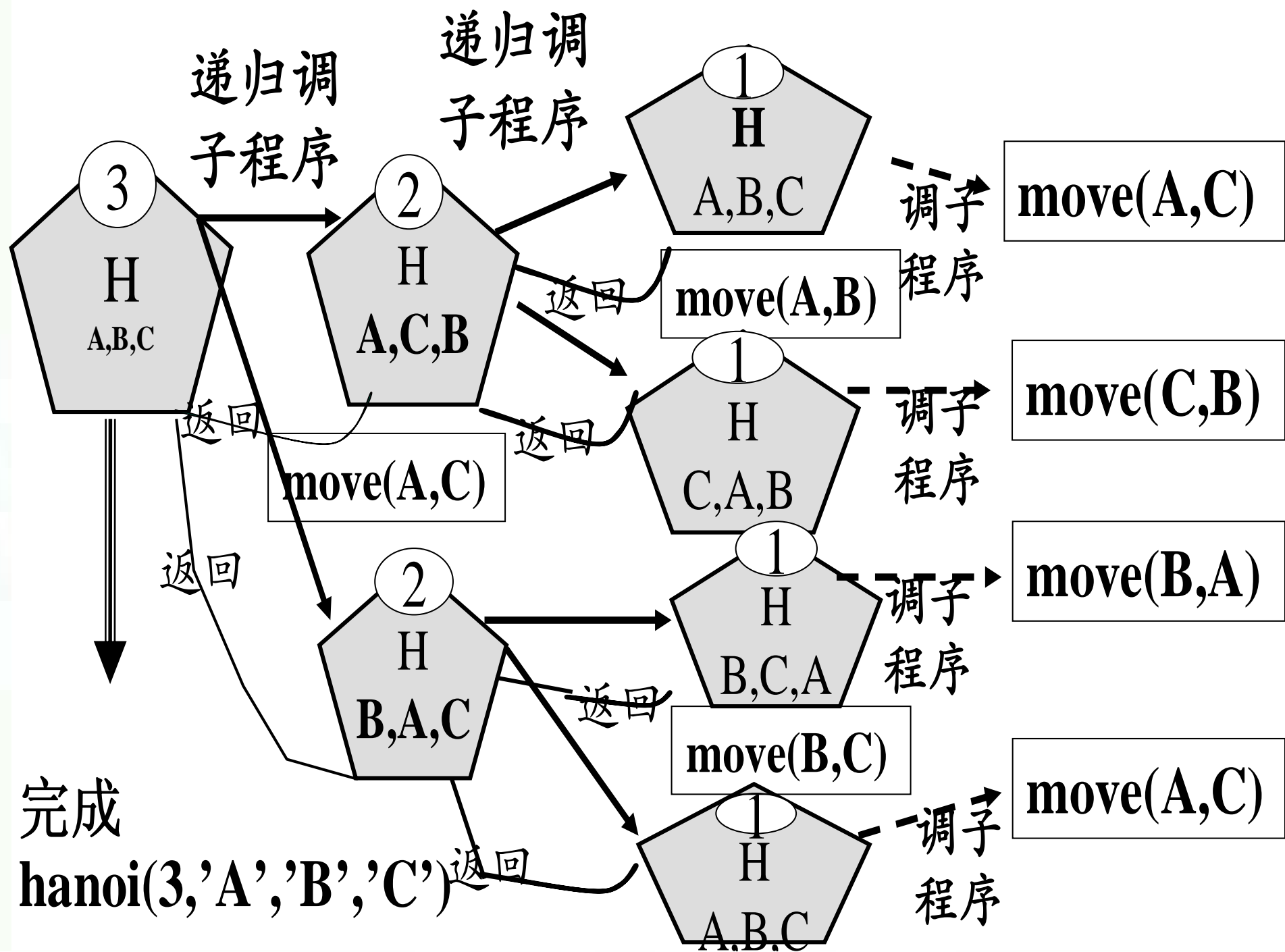


H机器

执行hanoi程序的
指令流

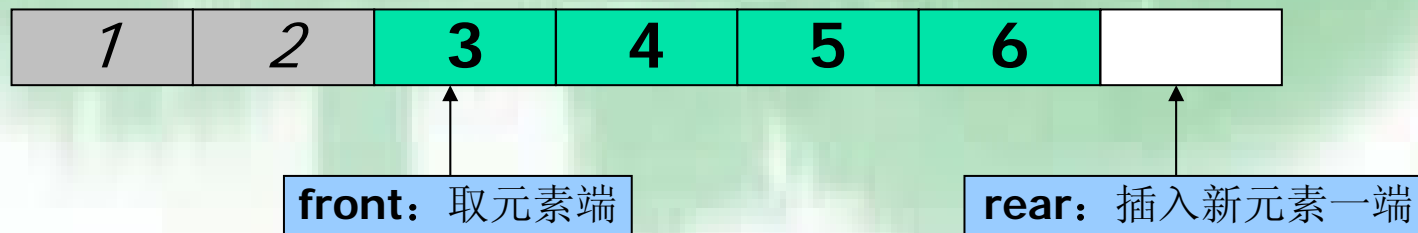
通过内部栈和子程
序交换信息





2.6 队列(queue)

- 限制访问点的线性表
 - 队列的尾端
 - ‘加入’新元素的一端
 - 队列的前端
 - ‘取出’元素的另一端
- “先进先出表”(FIFO, First In First Out)





队列的应用

- 消息缓冲器
- 邮件缓冲器
- 计算机的硬设备之间的通信
- 操作系统
- 广度优先搜索



队列的抽象数据类型

```
template <class ELEM> class Queue
{ //队列的运算集为:
    Queue(int s);           //创建队列实例, 最大长度为s
    ~ Queue();              //该实例消亡, 释放全部空间
    void EnQueue(ELEM item); // item进入队列尾
    ELEM DeQueue();          // 队列头出队列
    ELEM GetFirst();         // 读取队列头, 不删
    void ClearQueue();       // 变为空队列
    int IsEmpty();           // 队列空返回真
    int IsFull();            // 队列满返回真
    int length();            // 返回队列长度
};
```

back

next

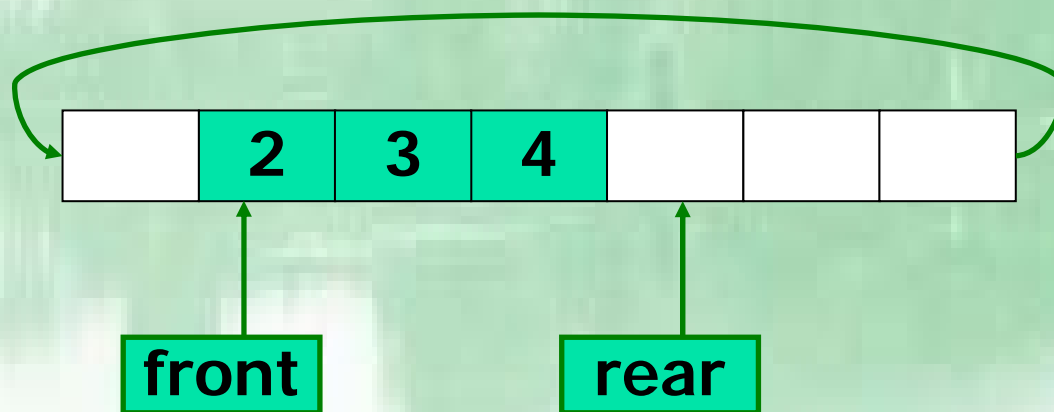


队 列

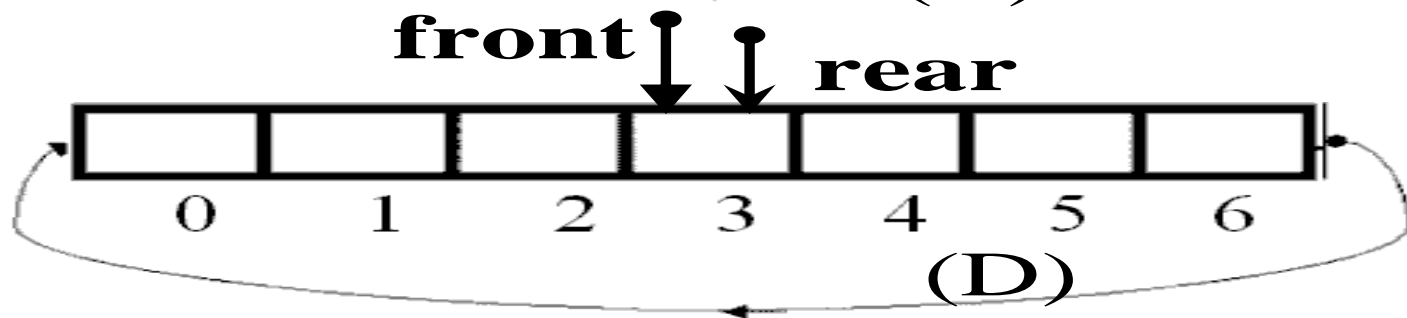
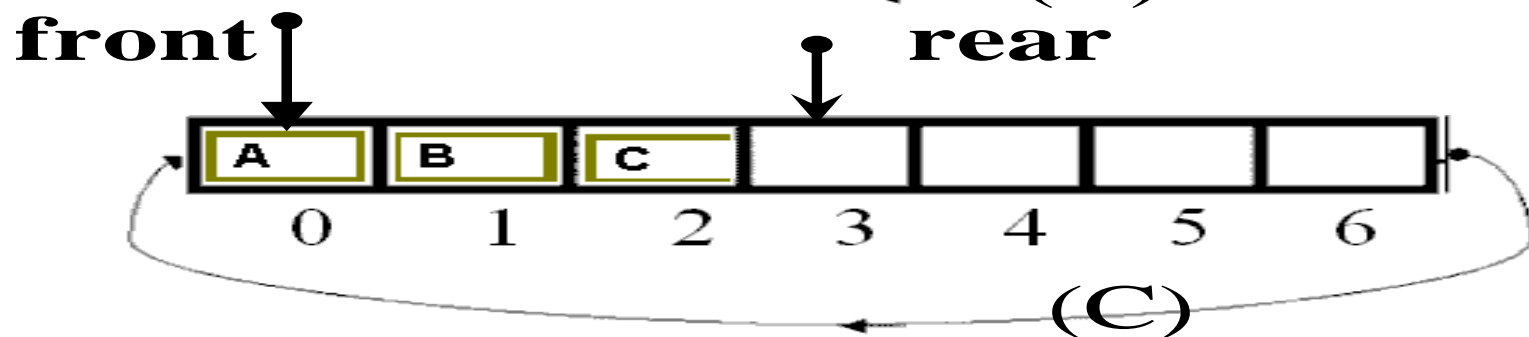
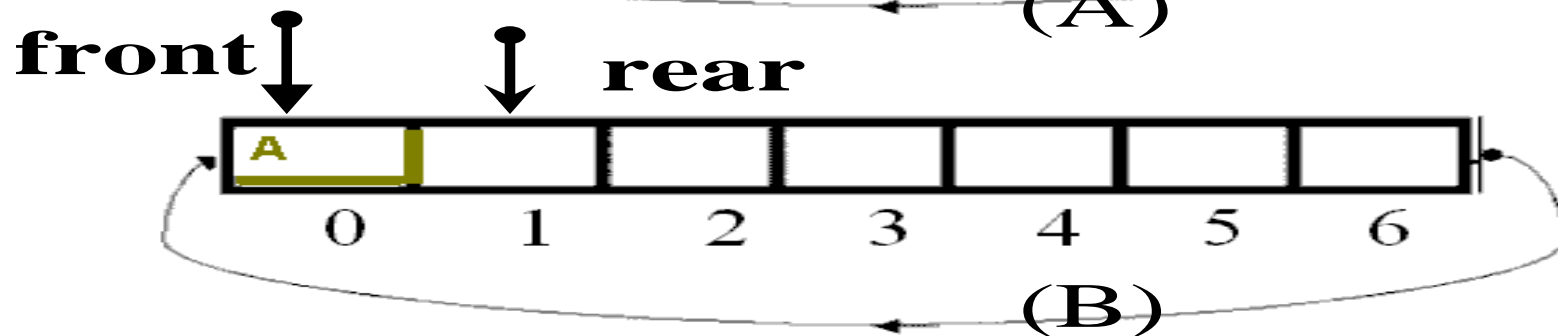
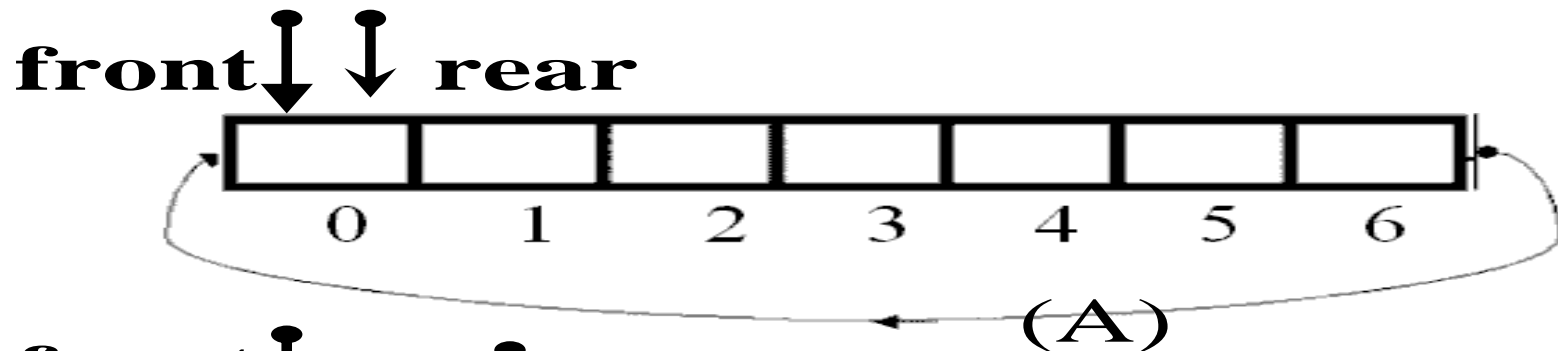
- ◆ 顺序队列
- ◆ 链式队列
- ◆ 顺序队列与链式队列的比较

2.6.1 顺序队列

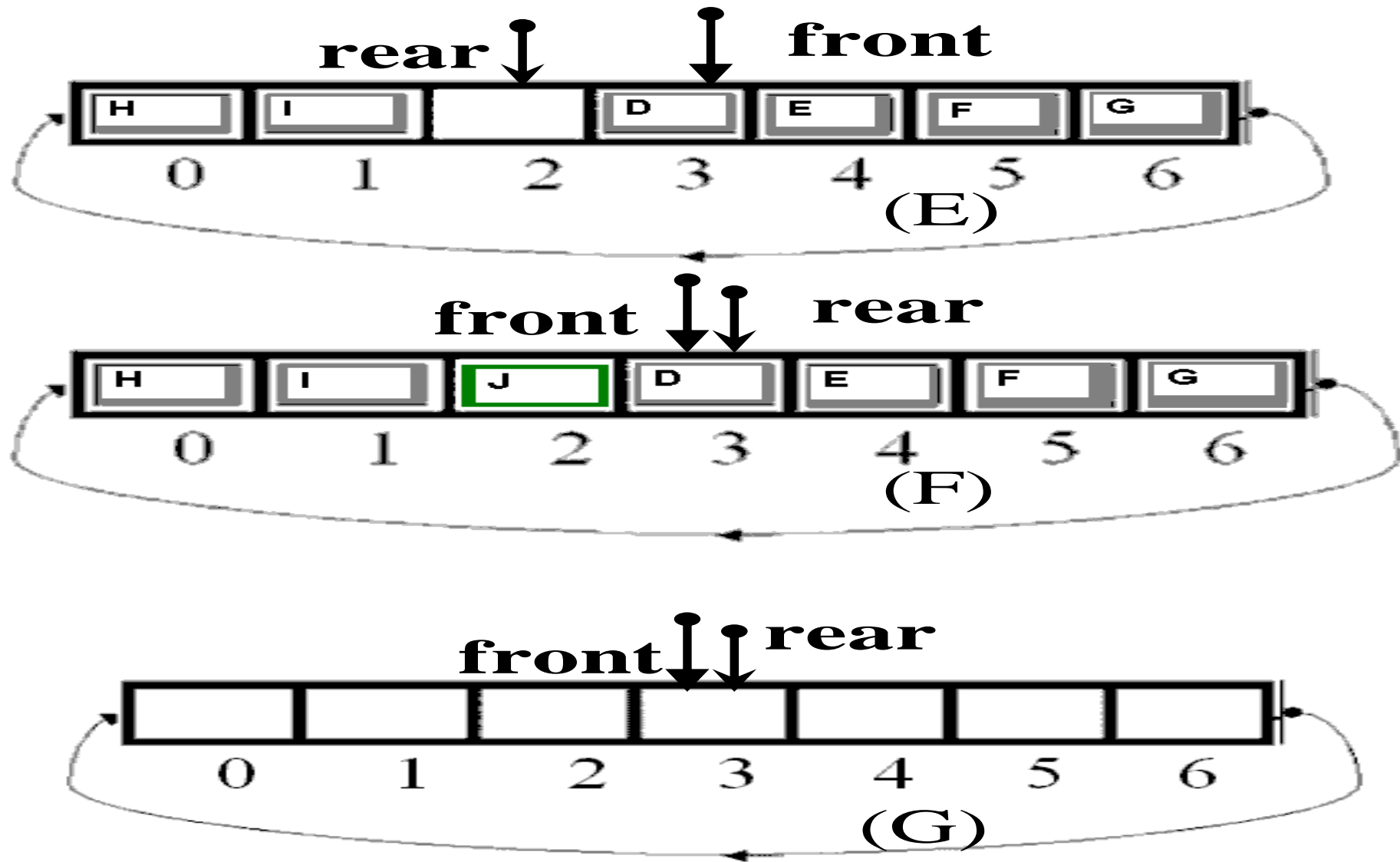
- 使用顺序表来实现队列。用向量存储队列元素，并用两个变量分别指向队列的前端和尾端



Rear向前虚指的队列运行示意图



队列的运行示意图





顺序队列的类定义

```
template <class ELEM> class Queue
{ //队列的运算集为:
private:
    ELEM *Qlist; //存放数据元素的向量
    int front, rear; //队列前端和尾端向量的下标值
    int maxsize; //队列最大长度
    int curr_len; //队列当前长度
public:
    Queue(int size=100) ;
    ~Queue() {delete [] Qlist;}
    ..... // 与ADT相同
};
```





队列的创建

```
Queue(int size=100) { //创建队列实例
    maxsize=size;
    Qlist = new ELEM[maxsize]; //动态空间
    assert(Qlist!=NULL); // 断言异常则退出
    front=rear=curr_len=0; // 让队列为空
}
```



压入队列顶

```
template <class ELEM>
void Queue<ELEM>::EnQueue(ELEM item)
{
    //判队列满，否则队列溢出异常，退出运行
    assert(!IsFull());
    curr_len ++;
    Qlist[rear] = item;           //在队列尾端加入队列
    rear = (rear + 1) % maxsize; //尾指针移动
}
```




算法2-15 从队列前端取出

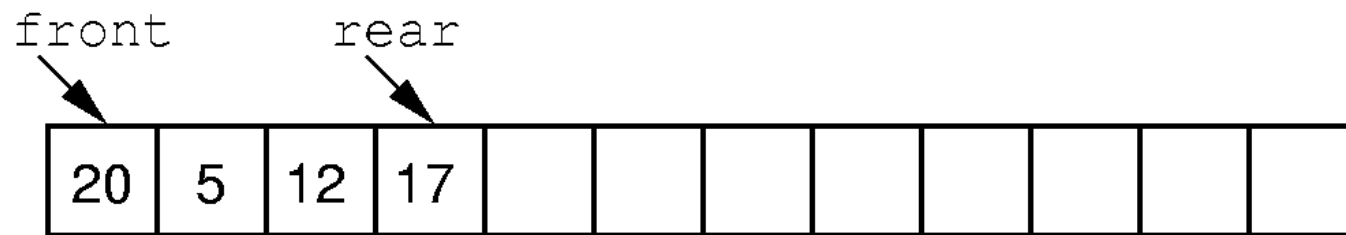
```
template <class ELEM>
ELEM Queue<ELEM>::DeQueue() {
    ELEM temp;
    //判队列非空，否则队列已空，异常退出运行
    assert(!IsEmpty());
    temp = Qlist[front];
    curr_len--;
    front = (front+1) % maxsize;
    return temp;
```

←back

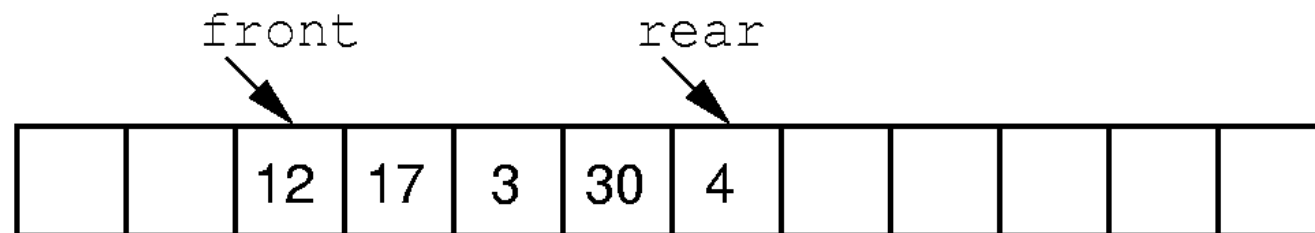
next→

顺序队列

■ front和rear都实指



(a)



(b)

// Array-based queue implementation

```
template <class Elem> class AQueue: public Queue<Elem> {
```

```
private:
```

```
int size; // Maximum size of queue
```

```
int front; // Index of front element
```

```
int rear; // Index of rear element
```

```
Elem *listArray; // Array holding queue elements
```

```
public:
```

```
AQueue(int sz =DefaultListSize) { // Constructor
```

```
// Make list array one position larger for empty slot
```

```
size = sz + 1; //size数组长, sz队列最大长度
```

```
rear = 0; front = 1; // 也可以rear=-1; front=0
```

```
listArray = new Elem[size];
```

```
}
```

```
~AQueue() { delete [] listArray; } // Destructor
```

```
void clear() { front = rear + 1; }
```





```
bool enqueue(const Elem& it) {  
    if (((rear+2) % size) == front)  
        return false;           // 还剩一个空位，就要报满  
    rear = (rear+1) % size;      // 因为实指，需要先移动到下一个空位  
    listArray[rear] = it;  
    return true;  
}  
  
bool dequeue(Elem& it) {  
    if (length() == 0) return false; // Empty  
    it = listArray[front];           // 先出队，再移动front下标  
    front = (front+1) % size;        // Circular increment  
    return true;  
}  
  
bool frontValue(Elem& it) const {  
    if (length() == 0) return false; // Empty  
    it = listArray[front]; return true; }  
  
int length() const  
{ return (size + (rear - front + 1)) % size; }  
  
back};
```

next



2.6.2 链式队列

- 单链表队列
- 链接指针的方向是从队列的前端向尾端链接

链式队列的类定义

```
template <class ELEM>
class Queue {
private:
    Link <ELEM> *front, *rear;
    int maxsize;           // 最大队列
    int curr_len;          // 当前队列长
public:
    Queue(int sz=100)      // 链式可以不设长度
        // 其他运算同队列ADT
    ...
};
```



将元素加入队列前端

```
template <class ELEM>
void Queue<ELEM>::EnQueue(ELEM item) {
    Link <ELEM> *temp;
    temp = new Link<ELEM>;
    assert(!temp==NULL); //若无存储空间则异常
    temp->data = item;
    temp->next = NULL;
    if (curr_len != 0) {
        rear->next = temp;
        rear = temp;           //新队列尾端指针
    }
    else front = rear = temp; //原空，首尾同指新结点
    curr_len++;
}
```





自单链队列前端取出

```
template <class ELEM>
ELEM Queue<ELEM>::DeQueue() {
    Link <ELEM> *temp; ELEM result;
    assert(curr_len != 0);           // 队空异常退出
    result = front->data;            // 暂存队列顶内容
    temp = front;                   // 老前端指针
    front = front->next;             // 新前端指针
    delete temp;
    curr_len--;
    if (curr_len == 0)
        rear = front = NULL;
    return result;
}
```




2.6.3 顺序队列与链式队列的比较

- 顺序队列
 - 固定的存储空间
- 链式队列
 - 可以满足浪涌大小无法估计的情况
 - 访问队列内部元素不方便



补充：递归到非递归的转换

- 递归函数调用原理
- 机械的递归转换
- 优化后的非递归函数



一个递归数学公式

$$fu(n) = \left\{ \begin{array}{ll} n+1 & \text{当 } n < 2 \text{ 时} \\ fu(\lfloor n/2 \rfloor) * fu(\lfloor n/4 \rfloor) & n \geq 2 \text{ 时} \end{array} \right\}$$



递归函数示例

```
void exmp(int n, int& f) {  
    int u1, u2;  
    if (n<2)  
        f = n+1;  
    else {  
        exmp((int) (n/2), u1);  
        exmp((int) (n/4), u2);  
        f = u1*u2;  
    }  
}
```

[back](#)

[next](#)



函数调用及返回的步骤

■ 调用

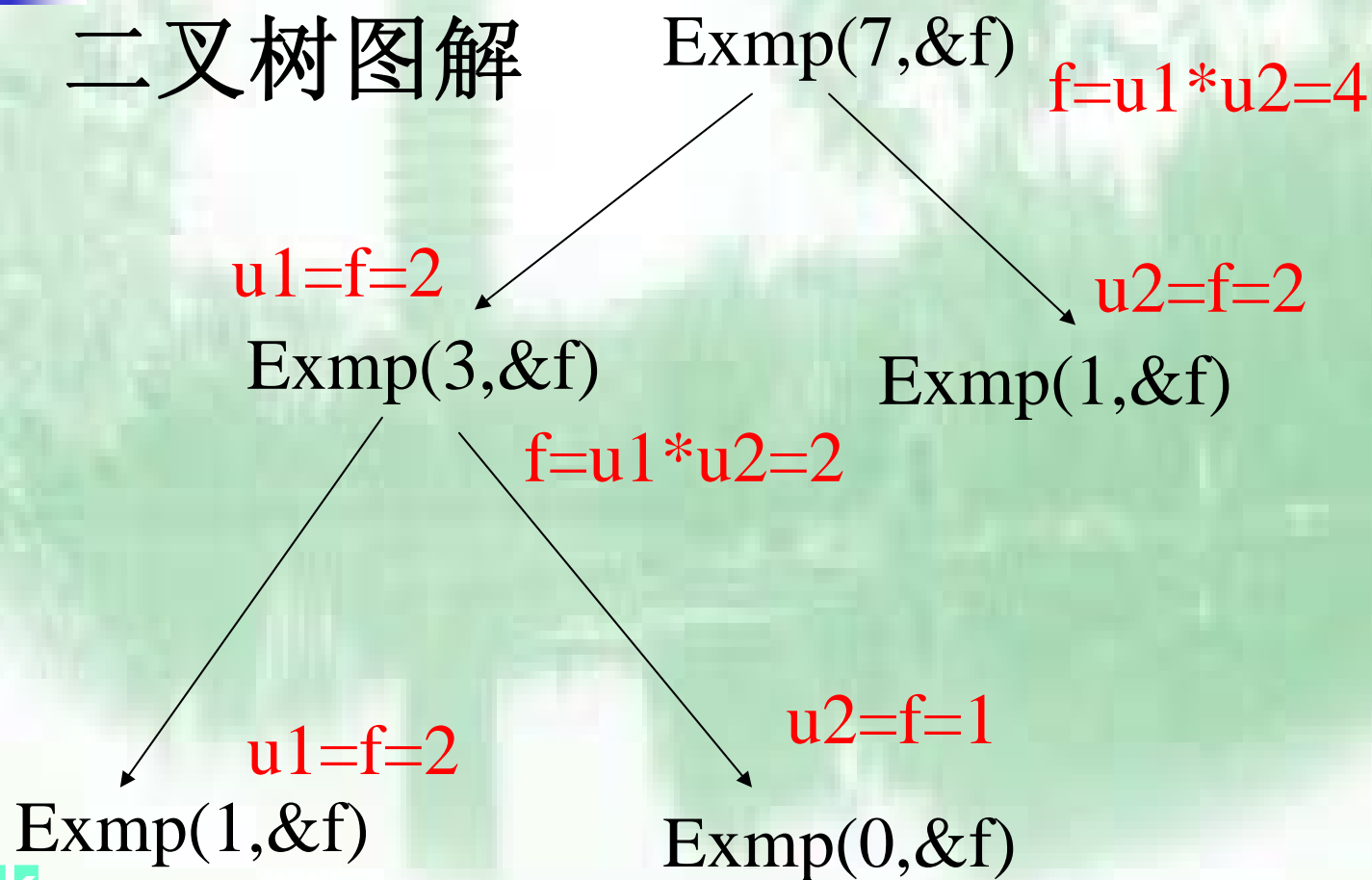
- 保存调用信息（参数，返回地址）
- 分配数据区（局部变量）
- 控制转移给被调函数的入口

■ 返回

- 保存返回信息
- 释放数据区
- 控制转移到上级函数（主调用函数）

函数执行过程图解

二叉树图解



back

next

用栈模拟递归调用过程

- 后调用，先返回(LIFO)，所以用栈

rd=3: n=7 f=? **u1=2 u2=2**

back

next



递归过程的模拟

- 假设 `void recfunc($p_1, p_2, p_3, \dots, p_k, p_{k+1}, \dots, p_m$)` 是一个递归函数
 - `void`是无返回值型的函数，如果有返回值，我们可以把返回值转换为一个引用型参数
 - 其中参数 $p_1, p_2, p_3, \dots, p_k$ 是值传递，参数 p_{k+1}, \dots, p_m 是引用传递。
 - 并设函数中有 n 个局部变量 q_1, \dots, q_n ，以及 t 个递归调用本函数的语句。

1. 设置一工作栈当前工作记录

- 在函数中出现的所有参数和局部变量都必须用栈中相应的数据成员代替

- 返回语句标号域 (t+2个数值)
- 函数参数 (值参、引用型)
- 局部变量

```
■ typedef struct elem { // 栈数据元素类型
    int rd; // 返回语句的标号
    Datatypeofpl pl; // 函数参数
    ... Datatypeofpm pm;
    Datatypeofql ql; // 局部变量
    ... Datatypeofqn qn; } ELEM;
```



2. 设置 $t+2$ 个语句标号:

- $label\ 0$: 第一个可执行语句
- $label\ t+1$: 设在函数体结束处
- $label\ i$: 第 i 个递归返回处
($1 \leq i \leq t$)



3. 增加非递归入口

- // 入栈
S.push(t+1, p1, ..., pm,
q1, ..., qn);

4. 第 i ($i=1, \dots, t$)个递归: 用以下语句替换:

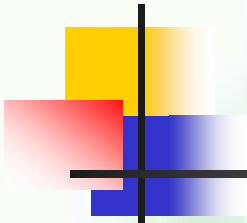
- 假设函数体中第 i ($i=1, \dots, t$)个递归调用语句为: `recfunc(a1, a2, ..., am);`
- 则用以下语句替换:

```
S.push(i, a1, ..., am); // 实参进栈
goto label0;
labeli: x = S.pop();
/* 退栈, 然后根据需要, 将x中某些值
   赋给栈顶的工作记录S.topValue() —— 相当于
   把引用型参数的值回传给局部变量 */
```



5. 所有递归出口处增加语句

- `goto label $t+1$;`



6. 标号为 $t+1$ 的语句


```
switch((x=S.topValue()).rd) {  
    case 0 : goto label0;  
            break;  
    case 1 : goto label1;  
            break;  
    ...  
    case t+1 : item=S.pop()  
              // 返回处理  
              break;  
    default : break;  
}
```

[back](#)

[next](#)





7. 改写循环和嵌套中的递归

 对于循环中的递归，改写成等价的goto型循环

 对于嵌套递归调用

如 `recfunc(... recfunc())`
改为: `exmp1 = recfunc();`
`exmp2 = recfunc(exmp1);`
`...`
`exmpk = recfunc(exmpk-1)`

  然后应用规则4解决



8. 优化处理

- 经过上述变换所得到的的是一个带goto语句的非递归程序。我们可以进一步优化，
 - 去掉冗余进栈，
 - 根据流程图找出相应的循环结构，从而消去goto语句



数据结构定义

```
typedef struct elem
{int rd, pn, pf, q1, q2;} ELEM;
class nonrec {
private:
    Stack S;
    void Enter(ELEM& x) { S.push(x); }
public:
    nonrec(void) { } // constructor
    void replacel(int n, int& f);
};
```



递归入口

```
void nonrec::replacel(int n, int& f) {  
    ELEM x, tmp  
    x.rd = 3;      x.pn = n;  
    Enter(x);  
label10: if ((x = S.topValue()).pn < 2) {  
        S.pop( );  
        x.pf = x.pn + 1;  
        Enter(x);  
        goto label13;  
    }  
}
```

back

next



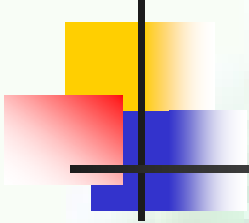
第一个递归语句

```
x.rd = 1; // The first rec  
x.pn = (int) (x.pn/2);  
Enter(x);  
goto label0;  
label1: tmp = S.pop();  
x = S.pop();  
x.q1 = tmp.pf; // modify u1=pf  
Enter(x);
```



第二个递归语句

```
x.pn = (int) (x.pn/4);  
x.rd = 2;  
Enter(x);  
goto label0;  
label2: tmp = S.pop();  
x = S.pop();  
x.q2 = tmp.pf;  
x.pf = x.q1 * x.q2;  
Enter(x);
```

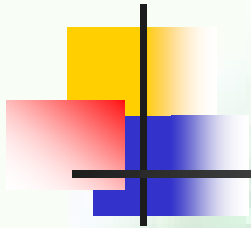


```
label3: x = S.topValue();  
       switch(x.rd) {  
           case 1 : goto label1;  
                   break;  
           case 2 : goto label2;  
                   break;  
           case 3 : tmp = S.pop();  
                   f = tmp.pf; //计算结束  
                   break;  
           default : cerr << "error label  
number in stack";  
                   break;  
       }
```

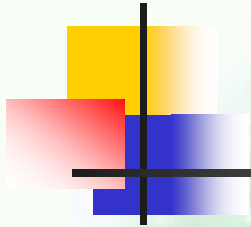


优化后的非递归算法

```
void nonrec::replace2(int n, int& f) {  
    ELEM x, tmp;  
    // 入口信息  
    x.rd = 3;    x.pn = n;    Enter(x);  
    do {  
        // 沿左边入栈  
        while ((x=S.topValue()).pn >= 2) {  
            x.rd = 1;  
            x.pn = (int)(x.pn/2);  
            Enter(x);  
        }  
    }  
}
```



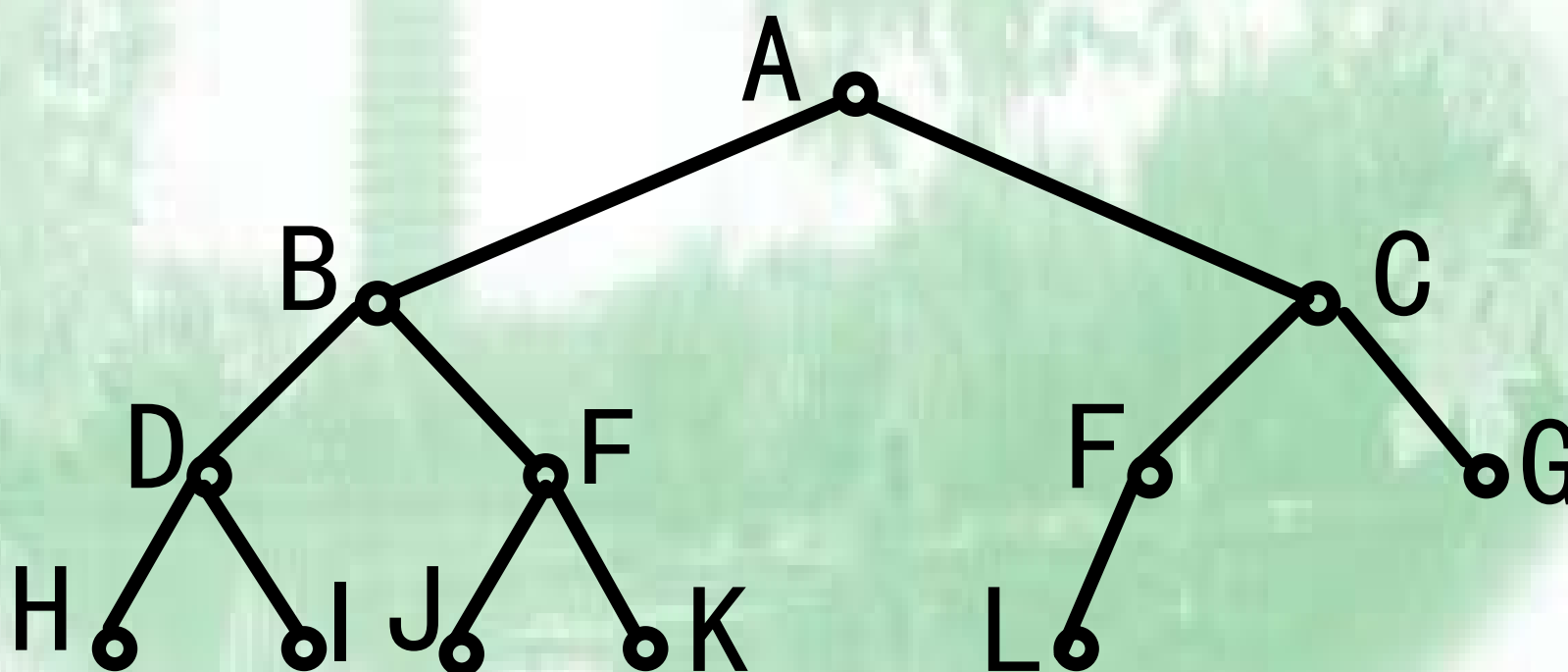
```
        x = S.pop();    // 原出口, n <= 2
    x.pf = x.pn + 1;
    Enter(x);
    // 如果是从第二个递归返回, 则上升
    while ((x = S.topValue()).rd==2) {
        tmp = S.pop();
        x = S.pop();
        x.pf = x.q * tmp.pf;
        Enter(x);
    }
```



```
if ((x = S.topValue()).rd == 1) {  
    tmp = S.pop();    x = S.pop();  
    x.q = tmp.pf;    Enter(x);  
    tmp.rd = 2;    // 进入第二个递归  
    tmp.pn = (int)(x.pn/4);  
    Enter(tmp);  
}  
} while ((x = S.topValue()).rd != 3);  
x = S.pop();  
f = x.pf;  
}
```




应用：二叉树非递归周游





小结

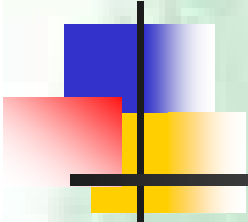
- 2.1 线性表(linear list)
- 2.2 顺序表—向量(sequential list—vector)
- 2.3 链表(linked list)
- 2.4 线性表实现方法的比较
- 2.5 栈——表达式求值，栈与递归
- 2.6 队列



变种的栈或队列结构

- 双端队列
- 双栈
- 超队列
- 超栈

谢谢!



祝大家学习进步!

<http://db.pku.edu.cn/mzhang/DS/>

张铭: mzhang@db.pku.edu.cn