



第二章 线性表、栈和队列



大纲

- **2.1 线性表(linear list)**
 - **2.1.1 线性表的抽象数据类型**
 - **2.1.2 线性表的存储结构**
 - **2.1.3 线性表运算分类**
- **2.2 顺序表—向量(sequential list—vector)**
 - **2.2.1 向量的类定义(type definition)**
 - **2.2.2 向量的运算**



大纲（续）

- **2.3 链表(linked list)**
 - **2.3.1 单链表(singly linked list)**
 - **2.3.2 双链表(double linked list)**
 - **2.3.3 循环链表(circularly linked list)**
- **2.4 线性表实现方法的比较**



■ 2.5 栈

- 2.5.1 顺序栈
- 2.5.2 链式栈
- 2.5.3 顺序栈与链式栈的比较
- 2.5.4 栈的应用——后缀表达式求值
- 2.5.4 递归的实现

■ 2.6 队列

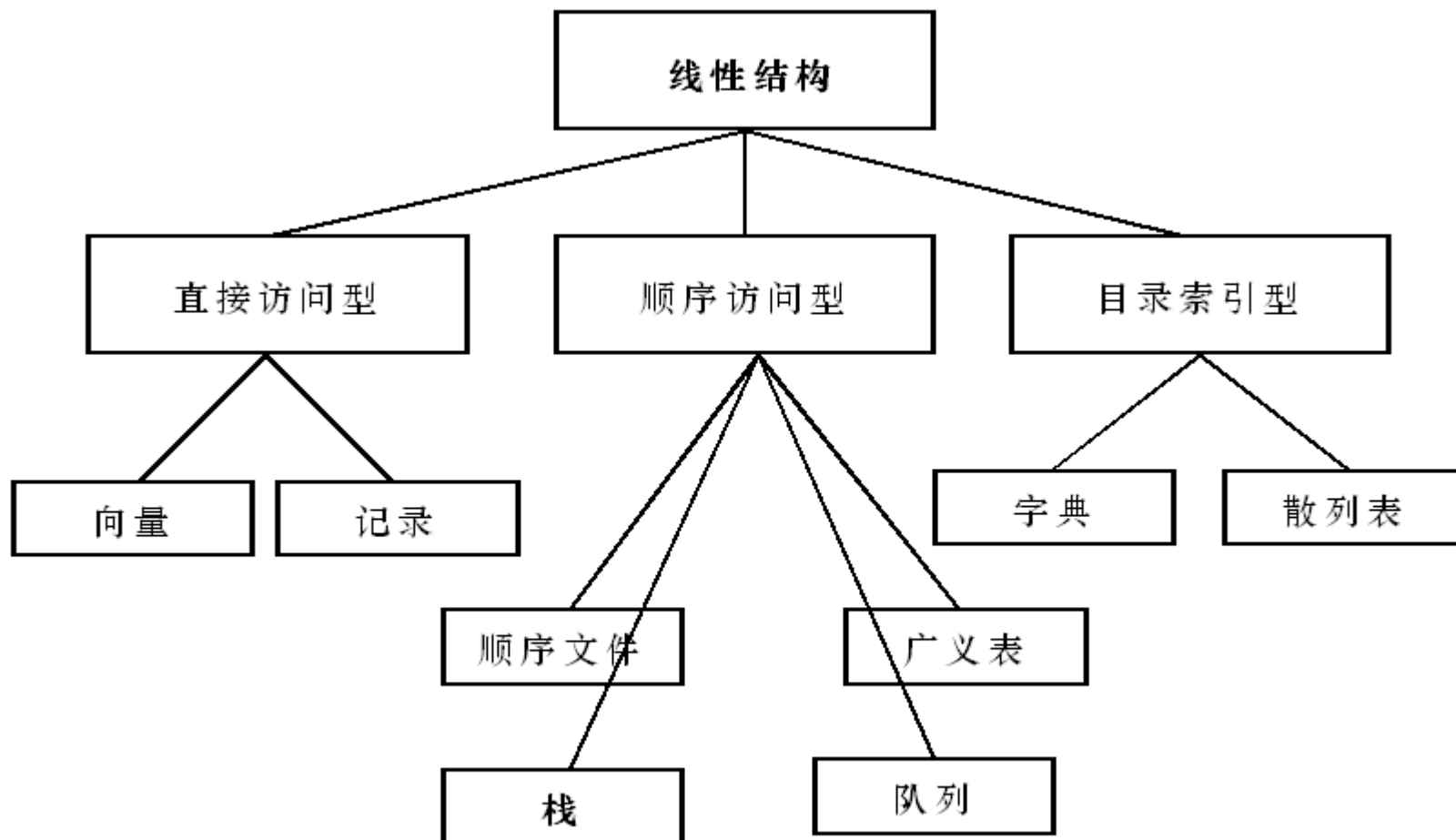
- 2.6.1 顺序队列
- 2.6.2 链式队列
- 2.2.3 顺序队列与链式队列的比较



线性结构分类

- 直接访问型(**direct access**)
- 顺序访问型(**sequential access**)
- 目录索引型(**directory access**)

线性结构分类





2.1 线性表(linear list)

- **2.1.1 线性表的抽象数据类型**
- **2.1.2 线性表的存储结构**
- **2.1.3 线性表运算分类**



线性表的抽象数据类型

- 线性表定义：
 - 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。这些结点称为线性表的元素。



线性表的性质

- 线性表 (N, r) :
 - (a) 结点集 N 中有一个唯一的开始结点，它没有前驱，但有一个唯一的后继；
 - (b) 对于有限集 N ，它存在一个唯一的终止结点，该结点有一个唯一的前驱而没有后继；
 - (c) 其它的结点皆称为内部结点，每一个内部结点既有一个唯一的前驱，也有一个唯一的后继；



线性表的性质（续）

- 线性表 (N, r) :
 - (d) 线性表所包含的结点数称为线性表的长度，它是线性表的一个重要参数；长度为0的线性表称为空表；
 - (e) 线性表的关系 r ，简称前驱关系，应具有反对称性和传递性。



线性表的抽象数据类型

- 取值空间
- 运算集



线性表类模板

```
template<class ELEM>
class list    //线性表类模板list，模板参数ELEM
{
    //1. 线性表的取值类型：
    //元素的类型为ELEM，是本list类模板的模板
    //参数ELEM。
    //本线性表用的最大长度为Max_length;
```



//2. 名字空间，使用变量访问线性表的方法：

//用curr ++或 curr--

//控制线性表游标curr的前后游走。

// 用公共变

//量curr_len指示线性表的尾部，并导出表的当

//前长度，...等。

// 3. 运算集：请参看下面的成员函数



private:

//私有变量，线性表的存储空间

//Max_length用于规定所存储线性表的最大长度

public:

int curr_len; //公共变量，该线性表的当前长度

int curr; //公共变量，该线性表的当前指针，游标

list(); // constructor算子，创建一个空的新线性表



//destructor算子,

//从计算机存储空间删去整个线性表

~list();

//将该线性表的全部元素清除，成为空表

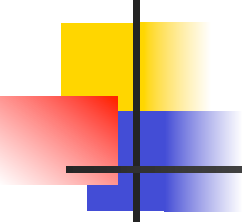
void clear() ;

// 尾附算子，在表的尾部添加一个新元素，参

//数value作为元素内容（数据类型为

//ELEM），表的长度加1

void append(ELEM value) ;



//插入算子，整数i指出第i号位置，参数value
//作为元素内容（数据类型为T），该位置上
//插入一个新结点，表的长度加1。第i号位置后
//的元素后移

void insert(int i, ELEM value) ;

//删除算子，删去第i号元素，表的长度减1，其
//后元素前移

void remove(int i);

//读取，返回第i个元素的值

ELEM fetch(int i);

}



2.1.2 线性表的存储结构

- 定长的一维数组结构
 - 又称向量型的顺序存储结构
- 变长的线性表存储结构
 - 链接式存储结构
 - 串结构、动态数组、以及顺序文件



2.1.3 线性表运算分类

- 创建线性表的一个实例list(-)
- 线性表消亡（即析构函数）
~list()
- 获取有关当前线性表的信息
- 访问线性表并改变线性表的内容或结构
- 线性表的辅助性管理操作



2.2 顺序表—向量 (sequential list—vector)

- 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，每一个元素唯一的索引值
 - 使用常数作为向量长度

2.2.1 向量的类定义(type definition)

- 数组存储
- 读写其元素很方便，通过下标即可指定位置



顺序表类定义

```
enum Boolean {False,True};  
//假定最大长度为100  
//并假定顺序表的元素类型T为ELEM  
const int Max_length = 100;  
class list {                //顺序表， 向量  
private :  
    //私有变量， 顺序表实例的最大长度  
    int msize;  
    // 私有变量， 顺序表实例的当前长度  
    int curr_len;
```



//私有变量，存储顺序表实例的向量

ELEM* nodelist;

public:

//以下列出成员函数（顺序表的算子集）

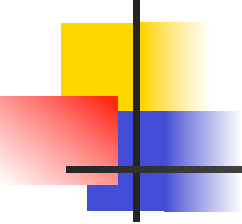
//当前下标，顺序表的公共变量

int curr;

// constructor算子,创建一个新的顺序表,

//其实参是表实例的最大长度。

list(const int size) ;



//destructor算子，用于将该表实例删去
~list();

//将顺序表存储的内容清除，成为空表

void clear();

//将当前下标curr赋值为第一个元素的位置

void setFirst();

//将当前下标curr下移一格，即curr+1

void next();

//若当前下标curr位置有值时，返回True

Boolean isInList();

//在表尾增添一个新元素，顺序表的实际长度加1

void append(const ELEM&);

//在当前下标curr位置插入元素新值。

void insert(const ELEM&);

//当前下标curr位置的元素值作为返回值，并删去该元素

ELEM remove();

Boolean isEmpty(); **//当线性表为空时，返回True**

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

int length(); **//返回此顺序表的当前实际长度**

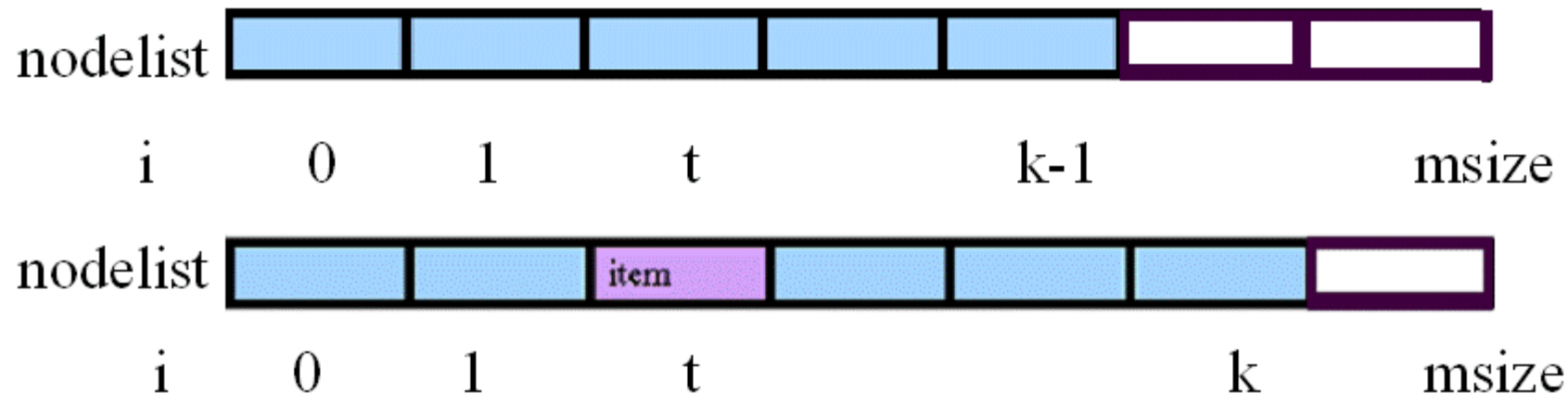
void prev(); **//将当前下标curr上移一格，即curr-1**

}

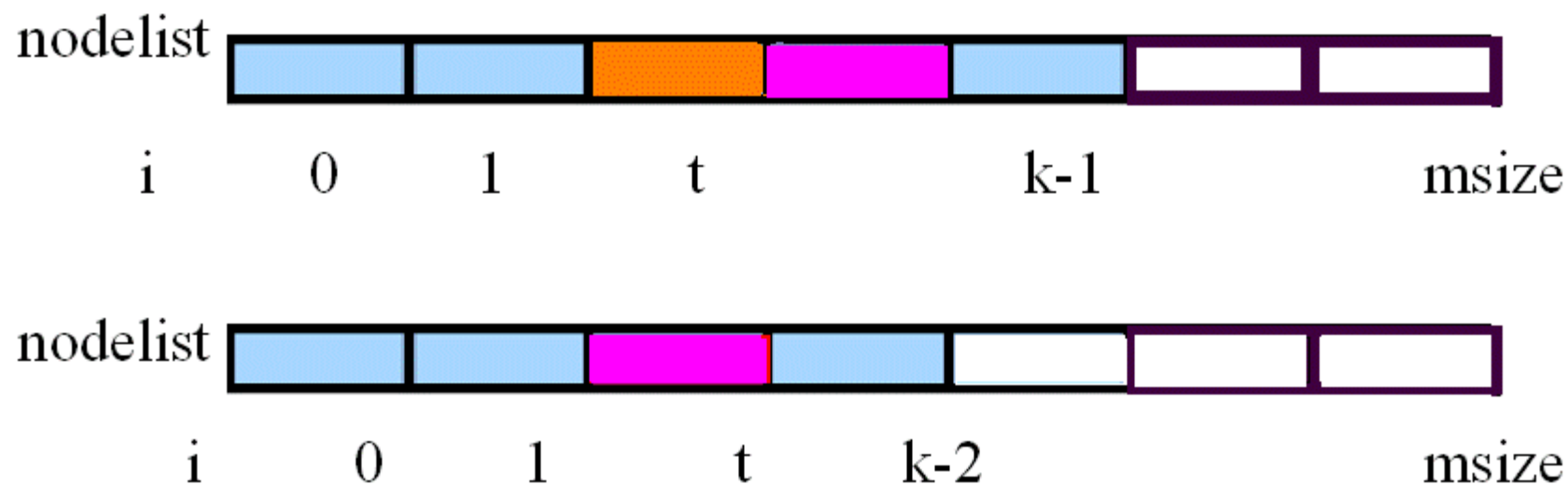


2.2.2 向量的运算

- 插入元素运算
 - **void insert(item)**
- 删除元素运算
 - **ELEM remove()**



(a) 在 t 位置插入元素 item



(b) 删除 t 位置的元素

插入算法



```
/* (设元素的类型为ELEM, nodelist是存储顺序表的  
向量, msize是此向量的最大长度, curr_len是此向  
量的当前长度, curr为此向量当前下标) */
```

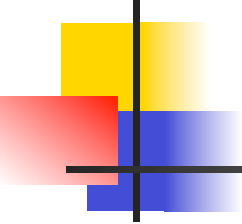
```
#include <assert.h>
```

```
viod insert(ELEM item)
```

```
{
```

```
//需要检查当前长度不能等于msize, 当前游标指针
```

```
//curr不能小于0,也不能大于当前长度
```



```
assert((curr_len < msize) && (curr >= 0)
      && (curr <= curr_len));
```

```
//此条件不满足时，程序异常，退出运行
```

```
//从表尾curr_len-1起往右移动直到curr
```

```
for(int i=curr_len; i>curr; i--)
```

```
    nodelist[i] = nodelist[i-1];
```

```
//当前指针处插入新元素
```

```
nodelist[curr] = item;
```

```
//表的实际长度curr_len加1
```

```
curr_len++;
```

```
}
```



算法执行时间

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

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

- 总时间开销估计为 **$O(k)$**



删除算法

```
/*（设元素的类型为ELEM，nodelist是存储顺序  
表的向量，msize是此向量的最大长度，  
curr_len是此向量的当前长度，curr为此向量  
当前下标）*/  
//返回curr所指的元素值，并从表中删去此元素  
ELEM remove()  
{  
//首先需要检验当前长度不能等于0，当前指针  
//curr不能小于0，不能等于curr_len
```



```
assert( (curr_len != 0) && (curr >= 0) && ( curr
```

```
    < curr_len ) );
```

```
//若上述条件为假，则程序异常，退出运行
```

```
ELEM temp = nodelist[curr];
```

```
//从指针curr到curr_len每个元素往前移一格
```

```
for(int i = curr; i < curr_len - 1; i++)
```

```
    nodelist[i] = nodelist[i + 1];
```

```
    curr_len--;                //表的实际长度curr_len减1
```

```
    return temp;                //返回值是进入时的旧值
```

```
}
```



算法时间代价

- 与插入操作相似， $O(k)$
- 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 $O(k)$



2.3 链表(linked list)

- 单链表
- 双链表
- 循环链表

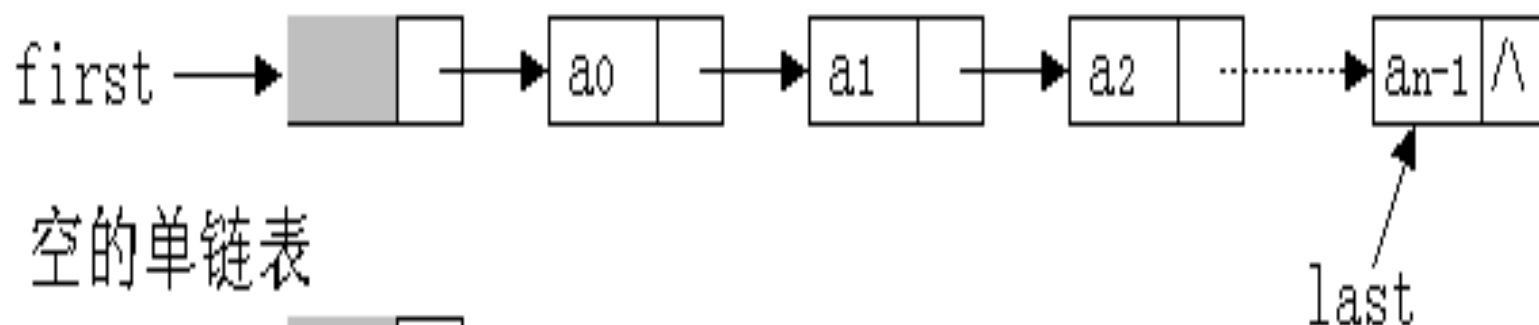


2.3.1 单链表

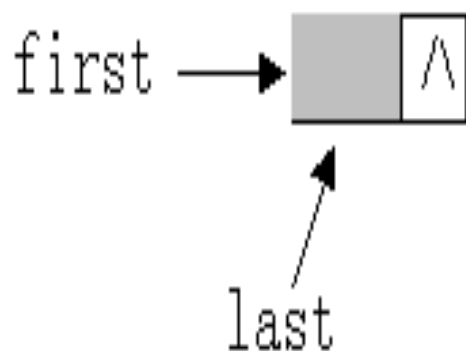
- 通过指针把它的一串存储结点链接成一个链
- 存储结点由两部分组成：
 - data字段
 - link字段

单链表的存储结构

单链表



空的单链表





单链表的结点类型以及变量 first说明

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



查找单链表中第i个结点算法

//函数返回值是找到的结点指针

ListNode * FindIndex(const int i)

{

 // **first**表首变量，可能指向空表

if(i == -1) return first;

***p=first->link; // p没有定义!**

int j=0;



```
while( p !=NULL && j < i )
```

```
{
```

```
    p=p->link;
```

```
    j++;
```

```
}
```

```
// 指向第i结点, i=0,1,..., 当链表
```

```
//中结点数小于i时返回NULL
```

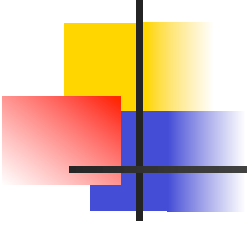
```
return p;
```



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。

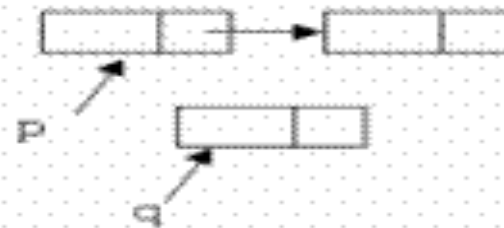
```
ListNode * Insert(ELEM value, int i)  
{  
    ListNode *p, *q;  
    q = new ListNode;  
    p = FindIndex(i-1);  
    if(p == NULL ) return NULL;
```



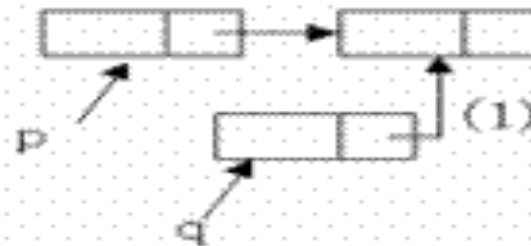
```
q->link = p->link;  
q->data = value;  
p->link = q;  
if(q->link == NULL )  
    last=q;  
return q;  
}
```


插入过程

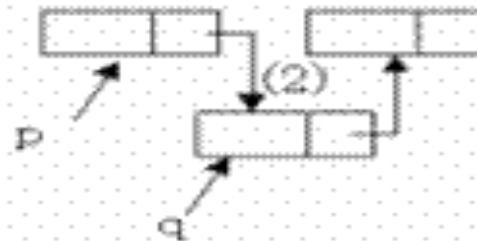
经过 $p = \text{FindIndex}(i-1);$
和 $q = \text{new ListNode};$



经过 $q \rightarrow \text{link} = p \rightarrow \text{link};$



再经过 $p \rightarrow \text{link} = q;$

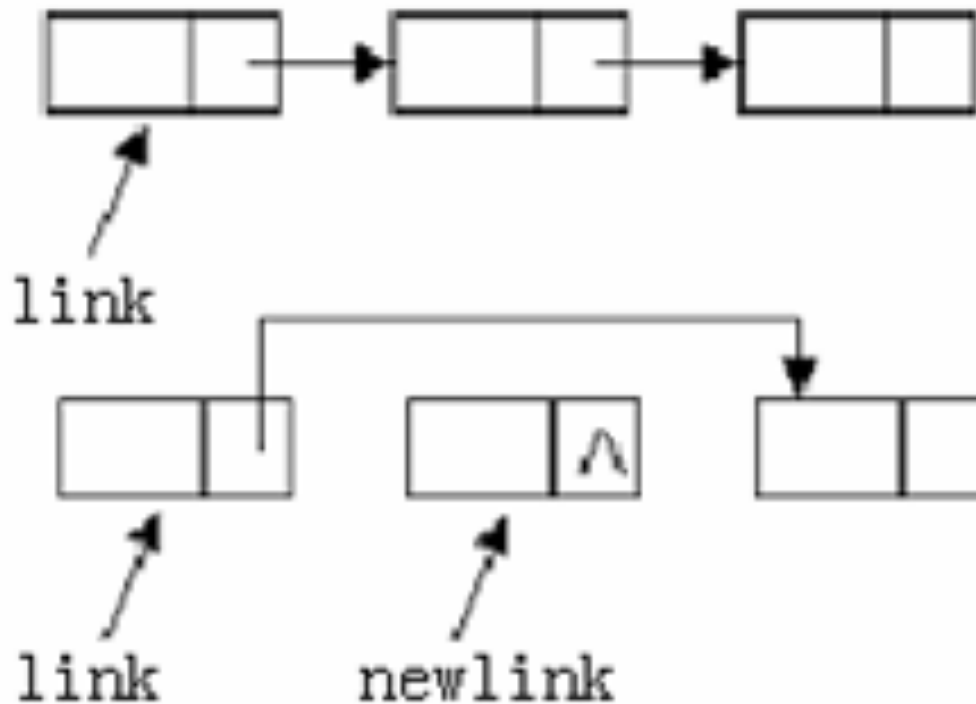




单链表删除算法

```
//删除由参数link所指定的结点  
void RemoveAfter(ListNode * link)  
{  
    ListNode *newlink=link;  
    if(link!=NULL)  
        link=link->link;  
    delete newlink;  
}
```

删除过程



求长度算法



```
int Length()
```

```
{
```

```
    ListNode *p=first->link;
```

```
    int count=0;
```

```
    while(p!=NULL)
```

```
    {
```

```
        p=p->link;
```

```
        count++;
```

```
    }
```

```
    return count;
```

```
}
```

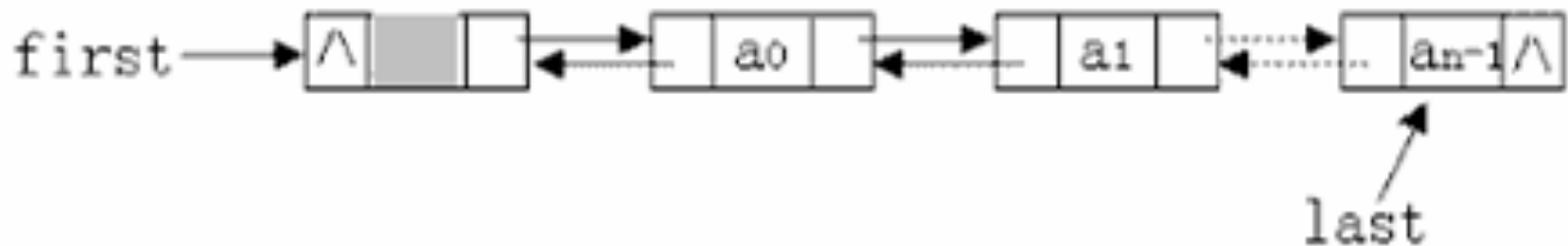


2.3.2 双链表 (double linked list)

- 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





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;
};
```



双链表的插入

如果要在p所指结点后插入一个新结点，首先执行new q开辟结点空间。然后，让该新结点的rlink填入p所指的后继地址，新结点的llink填入p所指结点的后继的llink字段，即

new q;

q->rlink=p->rlink;

q->llink=p->rlink->llink;

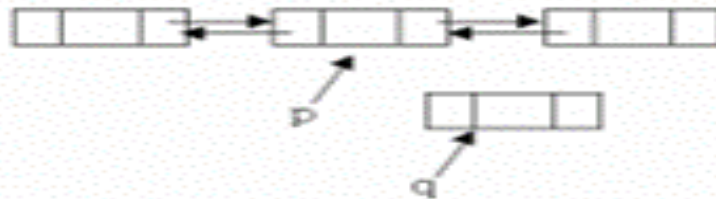
此外，要把新结点的地址填入原p所指结点的rlink字段，而且新结点后继的llink字段也应该回指新结点。

p->rlink=q;

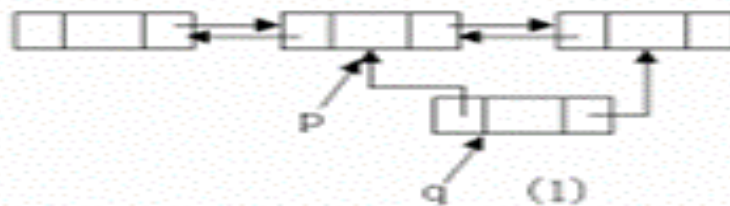
q->rlink->llink=q;

插入过程

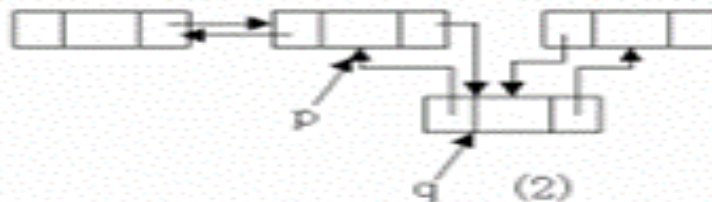
在 p 所指结点后面插入一个新结点



```
q->rlink=p->rlink;  
q->llink=p->rlink->llink;
```



```
p->rlink=q;  
q->rlink->llink=q;
```





双链表删除结点

- 如果要删除指针变量p所指的结点，只需修改该结点前驱的rlink字段和该结点后继的llink字段，即

p->llink->rlink=p->rlink;

p->rlink->llink=p->llink;

然后把变量p变空，再把p所指空间释放即可。

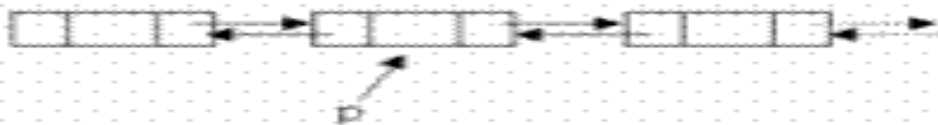
p->rlink=NULL;

p->llink=NULL;

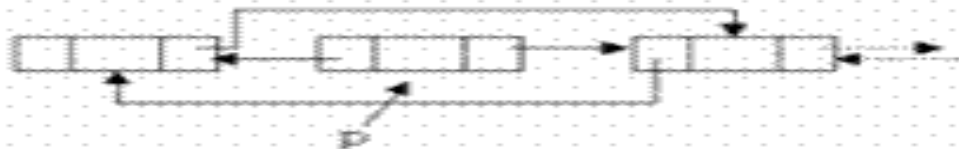
delete p;

删除过程

删除 p 所指的结点



```
p->llink->rlink=p->rlink;  
p->rlink->llink=p->llink;
```



```
p->rlink=NULL;  
p->llink=NULL
```



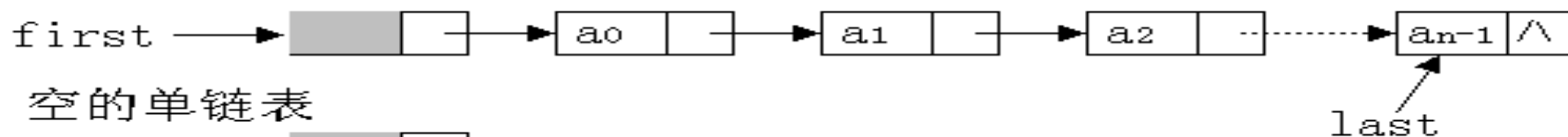


2.3.3 循环链表 (circularly linked list)

- 将单链表或者双链表的头尾结点链接起来，就是一个循环链表。
- 不增加额外存储花销，却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。

几种主要链表比较

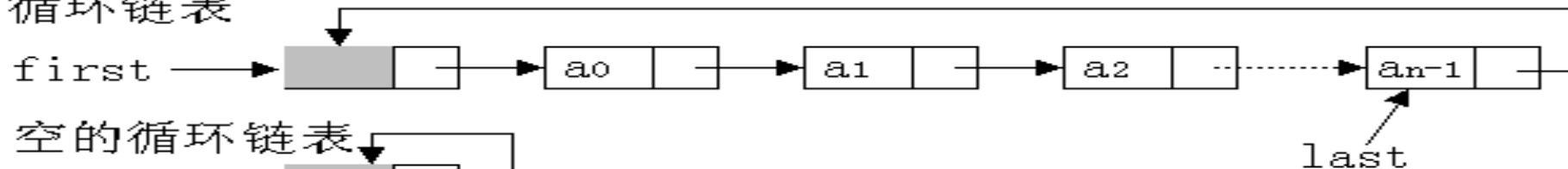
单链表



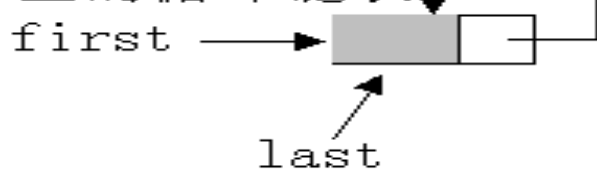
空的单链表



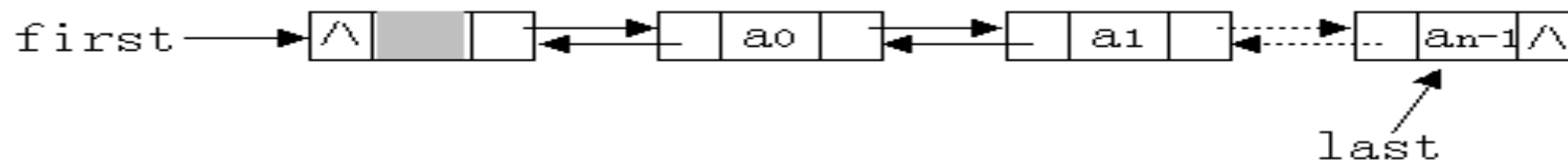
循环链表



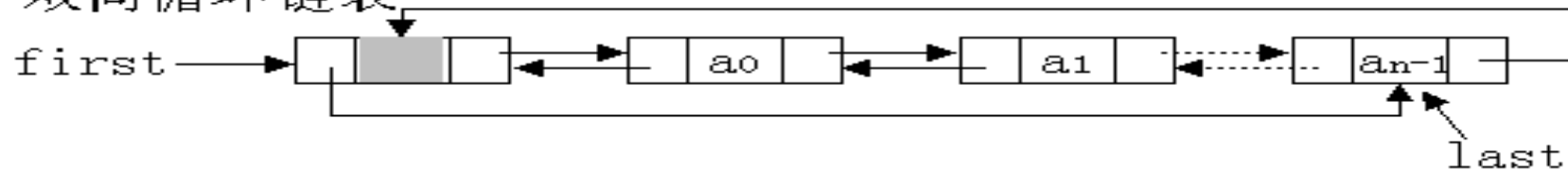
空的循环链表



双向链表



双向循环链表





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

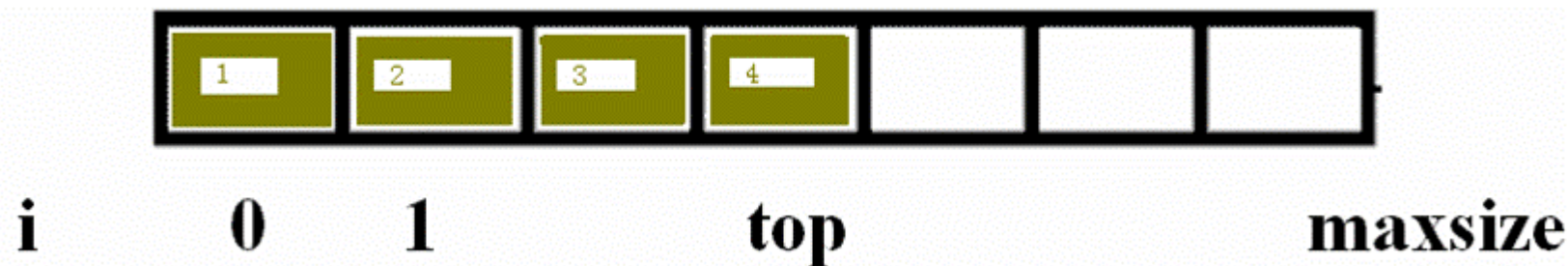


应用场合的选择

- 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)

- 一种限制访问端口的线性表，后进先出表(LIFO表，Least-In First-Out)。也称为“下推表”。
- 元素插入称为栈的‘压入’，push，删除称为栈的‘弹出’，pop
- 表首被称为‘栈顶’，而栈的另一端则叫做‘栈底’
- 每次取出（并被删除）的总是刚压进的元素，而最先压入的元素则被放在栈的底部





栈的抽象数据类型

```
enum Boolean {True,False}  
template <class ELEM> class Stack  
{ // 栈的元素类型为ELEM  
    //栈的存储：可以用顺序表或单链表存储，长  
    //度为定长  
    //栈的运算集为：  
    stack(int s); //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 MakeEmpty(); //变为空栈  
Boolean IsEmpty(); //返回真，若栈已空  
Boolean IsFull();    //返回真，若栈已满  
};
```



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



2.5.1 顺序栈

//设栈的类定义为**stack**，栈元素类型为浮点**float**类型

```
enum Boolean {True,False}
```

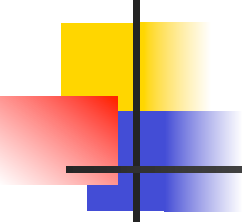
```
#include <assert.h>    //引入逻辑断言语句
```

```
class Stack
```

```
{
```

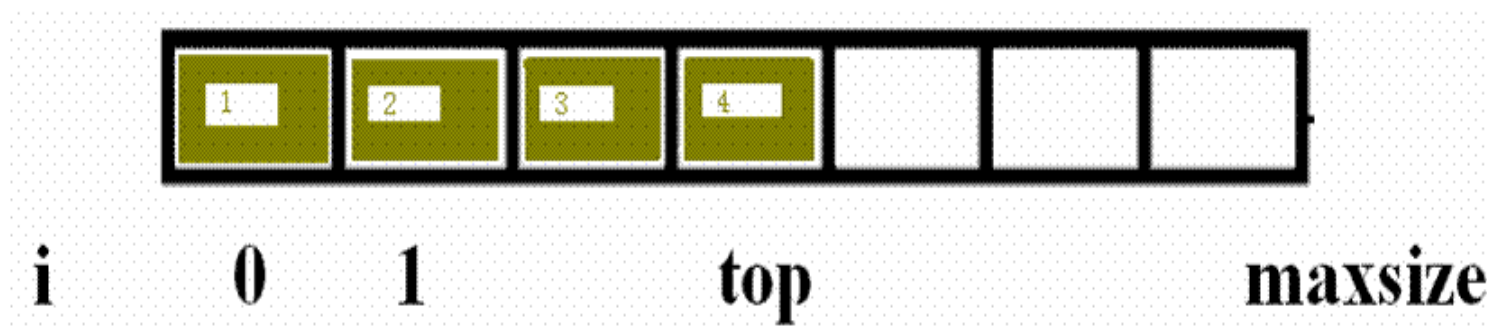
```
public:
```

```
    float *ElmList; //存放数据元素的指针变量
```



```
int top;//该变量指示栈顶在该向量的位置，下标  
    值  
//当新元素压入或栈内容弹出，top值随之增减  
int maxsize;    //栈的最大长度  
//构造函数，创建栈的实例，向量空间长度为size  
Stack(int size);  
  
...  
};
```

顺序栈



- 按压入先后次序，最先压入的元素编号为1，然后依次为2, 3, 4



顺序栈的创建

//栈实例的创建，指定最大长度10

Stack::Stack(int size=10)

{

maxsize=size;

//开辟向量存储空间

ElmList=new float[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

```
void Stack::Push(float 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

```
float Stack::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

```
float Stack::GetTop()  
{  
    //判栈非空，否则断言栈空异常，退  
    //出运行  
    assert(!IsEmpty());  
    return ElmList[top];  
}
```



其他算法

- 变空栈

```
void Stack::MakeEmpty()  
{  
    top=-1;  
}
```

- 栈消亡

```
~Stack(){delete []ElmList;}
```

- 栈满时，返回非零值（真true）

```
Boolean IsFull(){return top==maxsize-1;}
```



2.5.2 链式栈

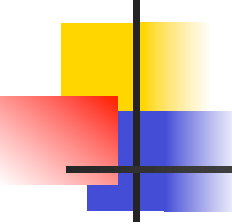
- 用单链表方式存储
- 指针的方向从栈顶向下链接



单链表的结点类型

```
struct ListNode  
{  
    ELEM data;  
    ListNode * link;  
};
```

链式栈的创建



```
#include <assert.h>
```

```
Class Stack {
```

```
// linked stack, 假定其元素类型为ListNode
```

```
private:
```

```
    ListNode *top;
```

```
public:
```

```
    Stack()
```

```
    {    //创建一个空栈, 不用指定最大长度
```

```
        top=NULL;
```

```
    };
```

```
...
```

```
};
```



压入栈顶

```
void Stack::Push(float item)  
{  
    ListNode * temp;  
    temp = new ListNode;  
    //若无存储空间则异常，程序退出运行  
    assert(!temp==NULL);  
    temp->data = item;  
    temp->link = top;    //老栈顶指针  
    top=temp;           //新栈顶指针  
}
```

北京大学软件与微电子学院 ©版权所有，转载或翻印必究



自单链栈弹出

```
ELEM Stack::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  
    ListNode * temptr;  
    temptr = top;           //老栈顶指针  
    top = top->link ;       //新栈顶指针  
    delete temptr;         //释放空间  
    return result;         //返回的是弹出内容  
}
```




小结

- 实际应用中，顺序栈比链式栈用得更广泛些
 - 顺序栈容易根据栈顶位置，进行相对位移，快速定位并读取栈的内部元素
 - 顺序栈读取内部元素的时间为 $O(1)$ ，而链式栈则需要沿着指针链游走，显然慢些，读取第 k 个元素需要时间为 $O(k)$
 - 一般来说，栈不允许“读取内部元素”，只能在栈顶操作



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)

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



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。

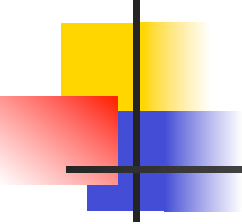


后缀表达式

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

中缀表达式转换成等价的后缀表达式

- (1) 当输入的是操作数时，直接输出到后缀表达式**PostfixExp**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**PostfixExp**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**InfixExp**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**PostfixExp**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

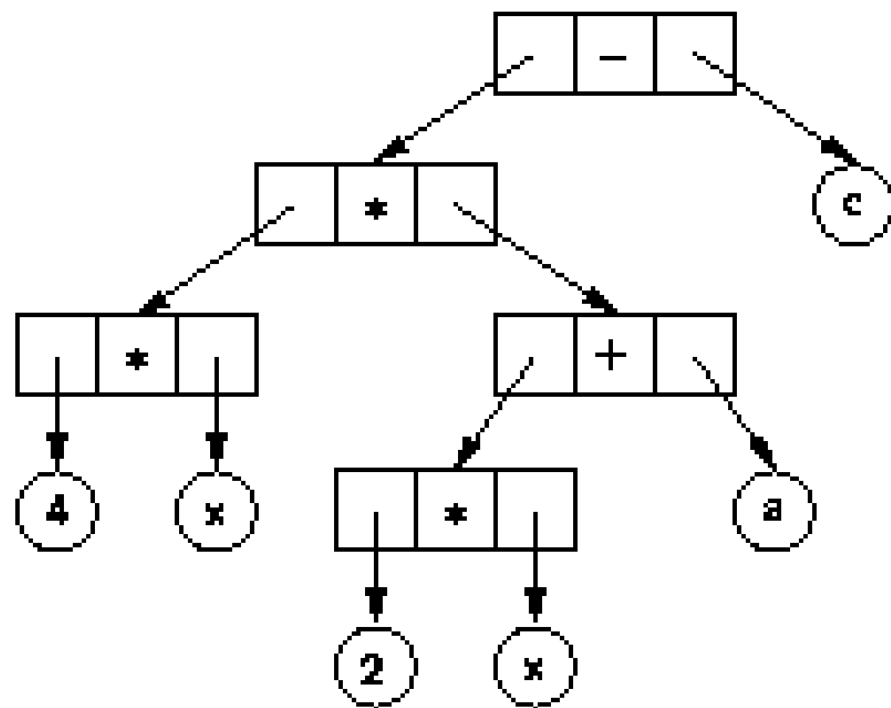
——后缀表达式求值

■ 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

//Class Declaration 类的说明

enum Boolean {False,True};

typedef double ELEM;

//文件astack.h中有栈，类stack的定义

#include "astack.h"

class Calculator{

private:

Stack S; //这个栈是用于压入保存操作数

//把一个浮点数num压入栈

void Enter(double num);



//从栈顶弹出两个操作数，赋值给变参opnd1和opnd2

Boolean GetTwoOperands(double& opnd1,double& opnd2);

//调用GetTwoOperands，并按op运算，对两个操作数

//进行计算

void Compute(char op);

public:

//创建计算器实例，开辟一个空栈

Calculator(void) { };



```
//后缀表达式的读入，在遇到=符号，
```

```
//启动求值计算
```

```
void Run (void);
```

```
//计算器的清除，为随后的下一次计算作准备
```

```
void Clear (void);
```

```
//-----//
```

```
}
```

```
//计算器类class Calculator的程序实现
```

```
void Calculator::Enter(double num)
```

```
{
```

```
    S.Push(num);
```



**Boolean Calculator::GetTwoOperands(double&
opnd1,double& opnd2)**

{

if (S.StackEmpty())

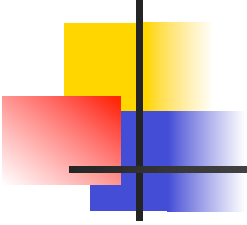
{

cerr<<"Missing operand!"<<endl;

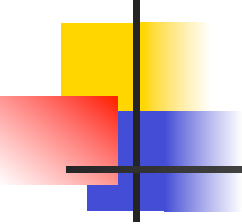
return False;

}

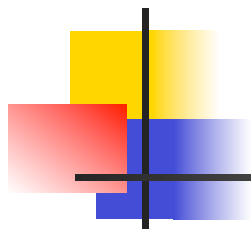
opnd1 = S.Pop();//右操作数



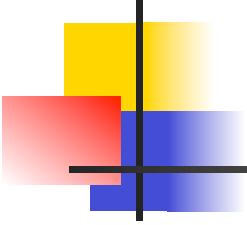
```
if (S.StackEmpty())
{
    cerr<<"Missing operand!"<<endl;
    return False;
}
opnd2 = S.Pop();//左操作数
return True;
}
```



```
void Calculator::Compute (char op)  
{  
    Boolean result;  
    double operand1, operand1;  
    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<<"Divided by 0!"<<endl;
```



```
        S.ClearStack();  
    }  
    else  
        S.Push(operand2 / operand1);  
    break;  
}  
else  
    S.ClearStack();  
}
```



```
void Calculator::Run(void)
```

```
{
```

```
    char c;
```

```
    double newoperand;
```

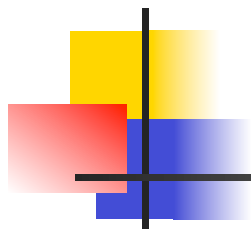
```
    while(cin>>c, c != '=')
```

```
{
```

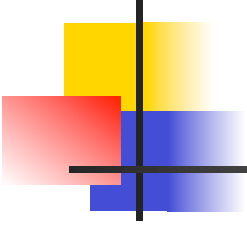
```
        case '+':
```

```
        case '-':
```

```
        case '*':
```



```
case '/':  
    Compute(c);  
    break;  
default:  
    cin.putback(c);  
    cin >> newoperand;  
    Enter(newoperand);  
    break;  
}
```



```
if (!S.StackEmpty())  
    //印出求值的最后结果  
    cout << S.top() << endl;  
}  
void Calculator::Clear(void)  
{  
    S.ClearStack();  
}
```

[后缀计算器的类定义，结束]



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:

$$\text{factorial}(k+1) = \begin{cases} 1, & \text{if } n \leq 0 \\ n * \text{factorial}(n-1), & \text{if } n > 0 \end{cases}$$



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 ;
}
```




//递归定义的阶乘n!的函数

long factorial(long n)

{

if (n==0)

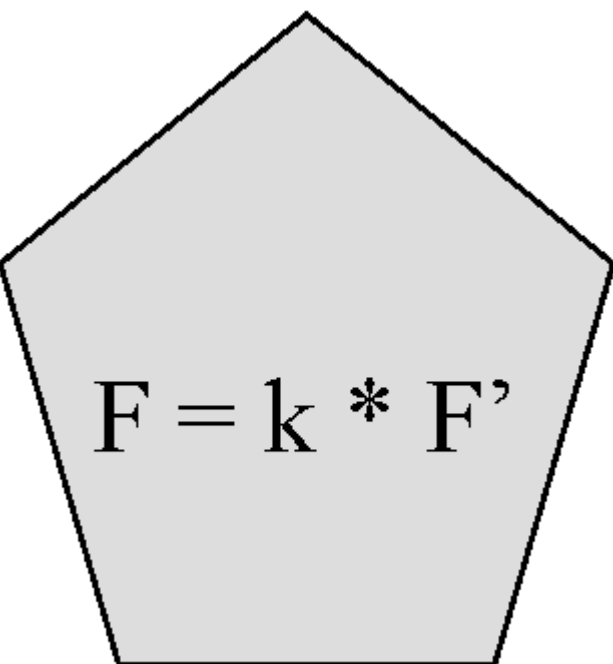
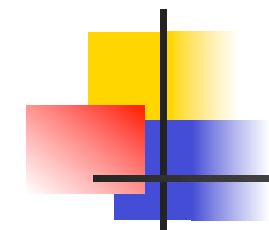
return 1;

else

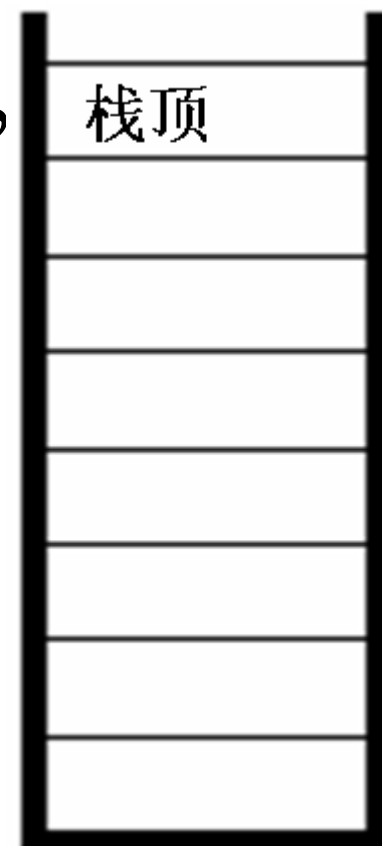
return n * factorial(n-1) ; //递归调用

}

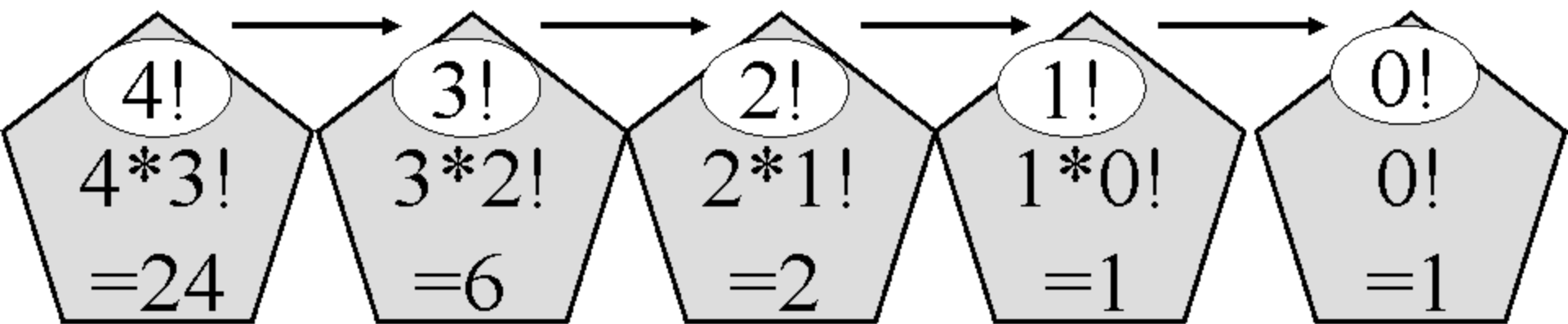
用5个F机器来模拟 4! 的计算



左边是一个 F 机器，它能够计算乘积并能够与其他机器交换信息。通过内部栈交换信息。右边是它们所使用的栈



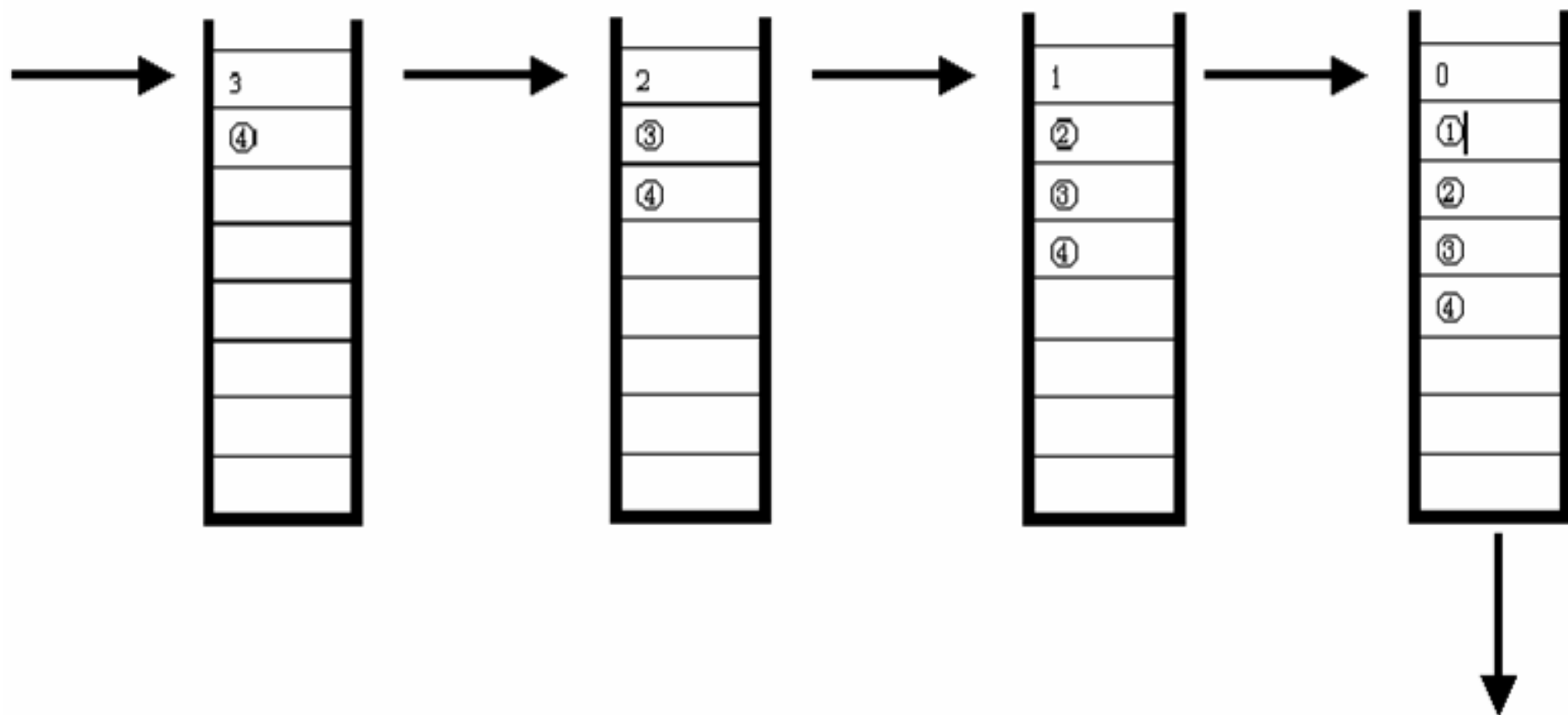
递归调 递归调 递归调 递归调
子程序 子程序 子程序 子程序

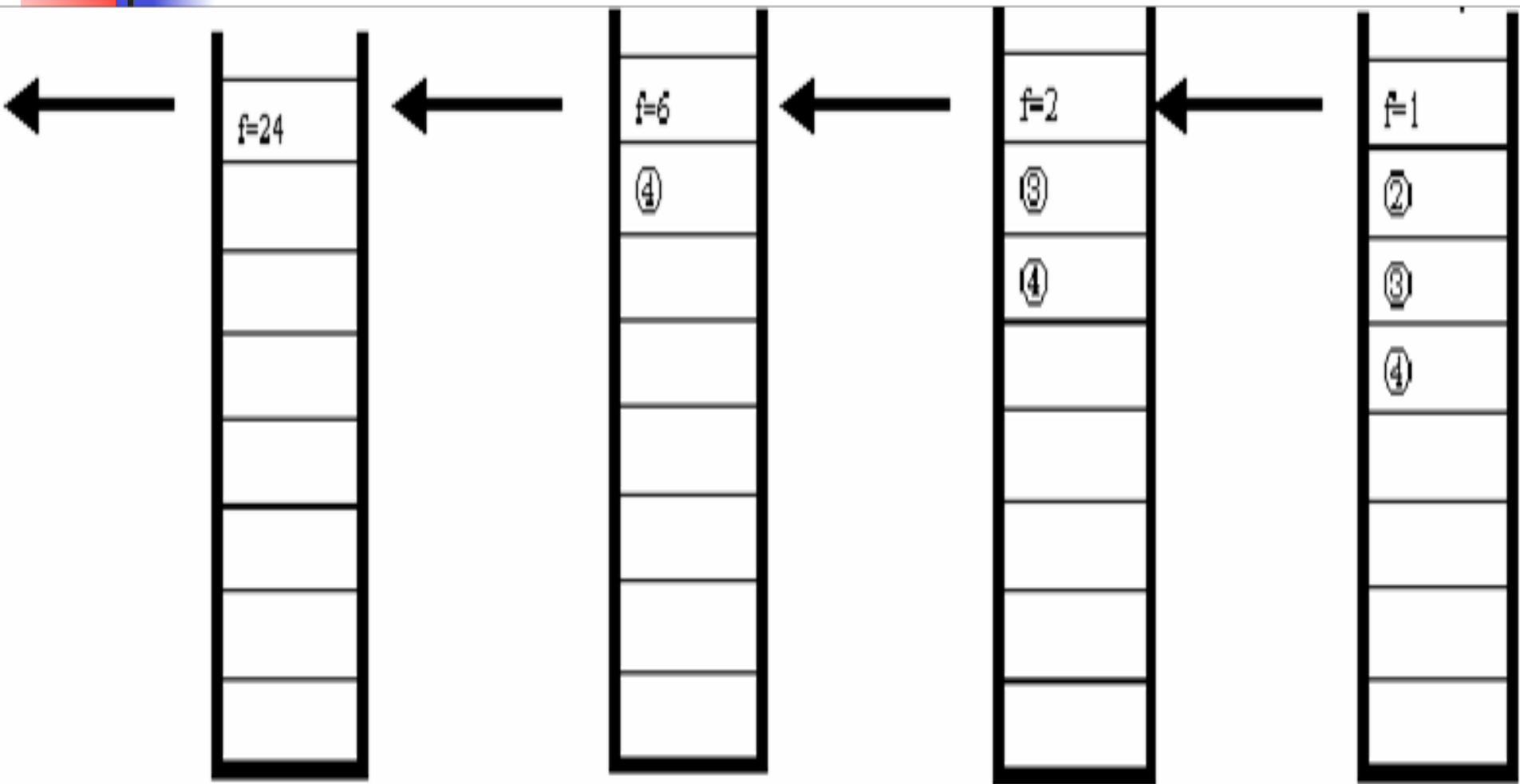
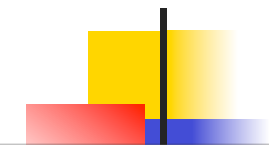


结果返回 结果返回 结果返回 结果返回

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

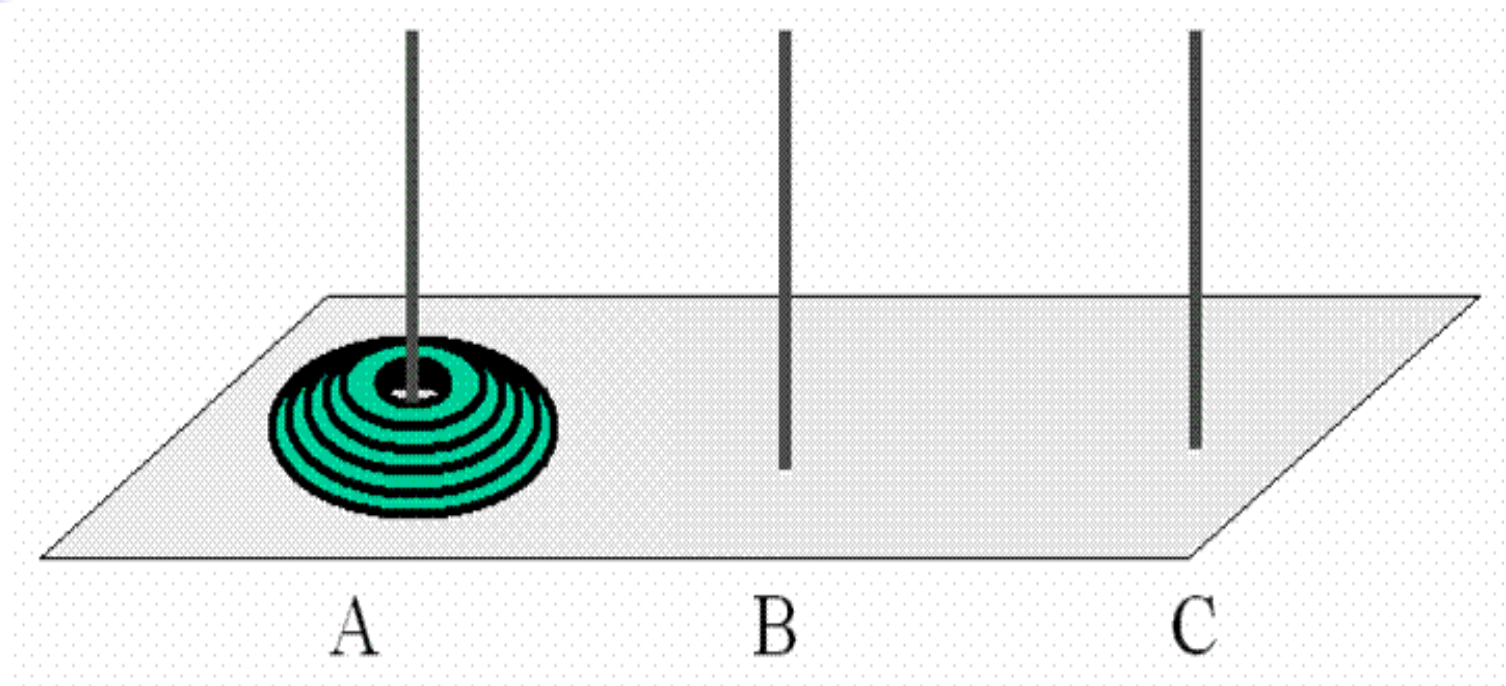
递归计算时内部栈情况



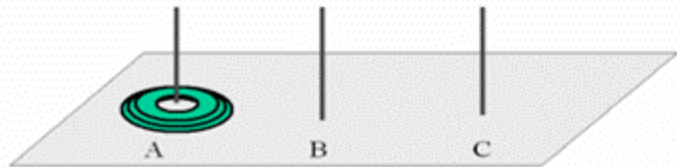




hanoi塔问题的递归求解

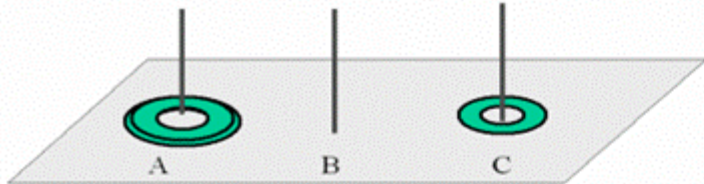


hanoi塔的执行过程



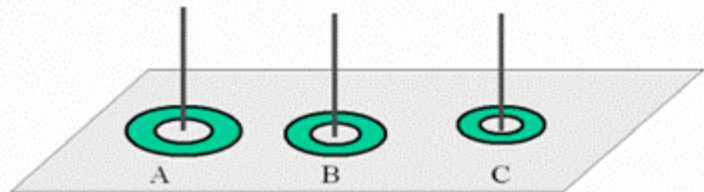
1

move(A,C)



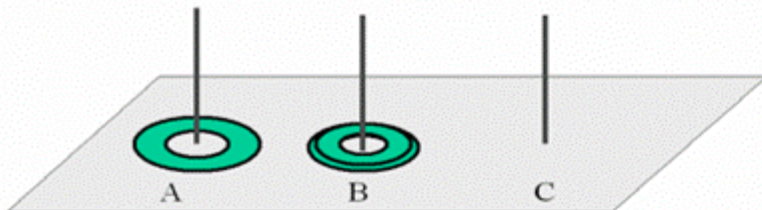
2

move(A,B)



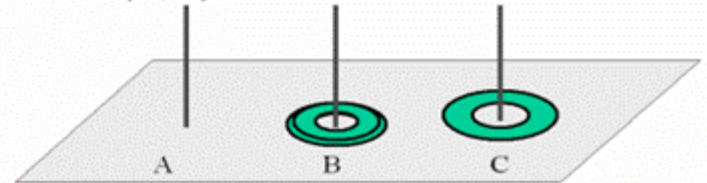
3

move(C,B)



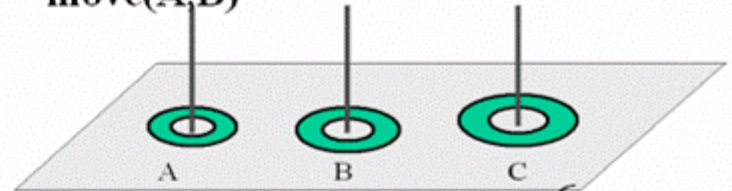
4

move(A,C)



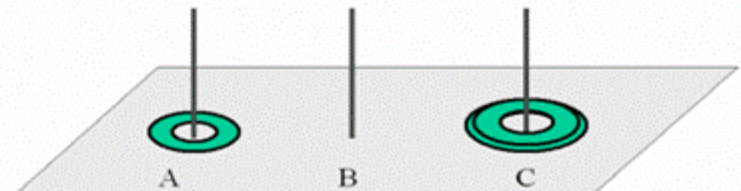
5

move(A,B)



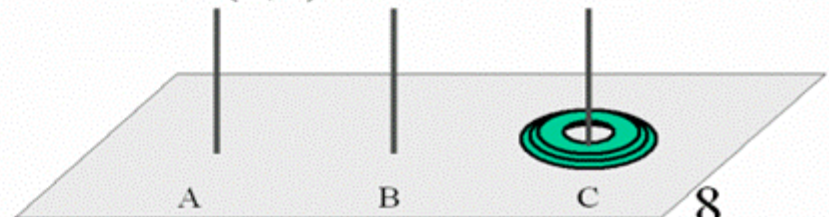
6

move(B,A)



7

move(A,C)



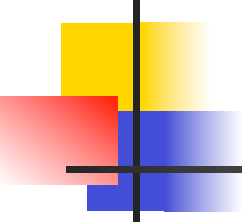
8



河内塔问题的递归求解程序

hanoi(n,X,Y,Z)的含义是：

移动n个槃环，由X柱子（出发柱）将槃环移动到Z柱（终点柱）。其移动过程中，Y柱和X柱皆可用于暂时存放环槃，不过注意每一步移动都必须严格遵循大盘不能压小盘的原则。例如，**hanoi(2, 'B', 'C', 'A')**，其含义是初始位于B柱上部的2个环槃移动到A柱，执行过程中可以使用C，B柱暂存环槃。



//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
```

```
    }
```

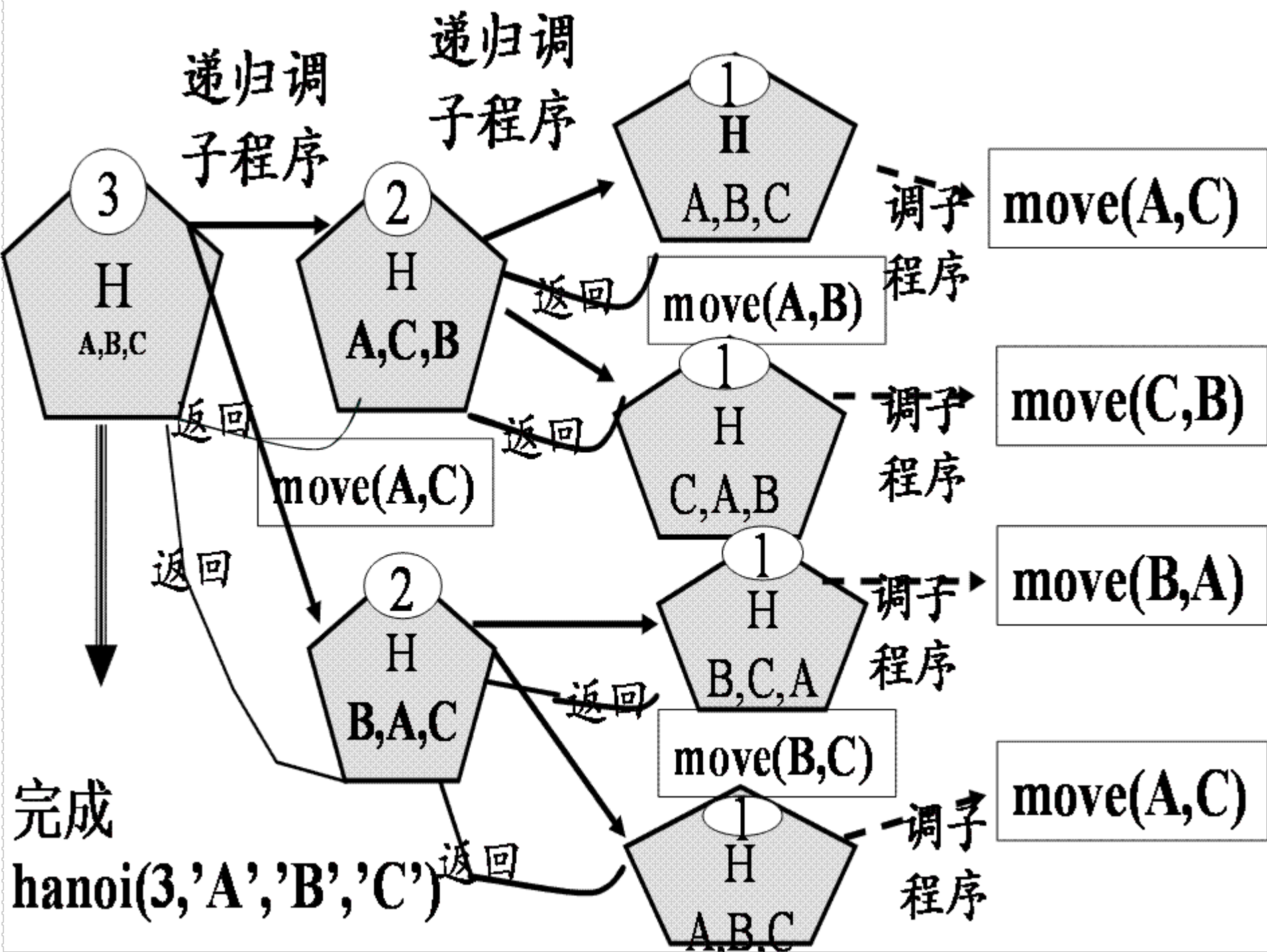
```
}
```

hanoi递归子程序的运行示意图

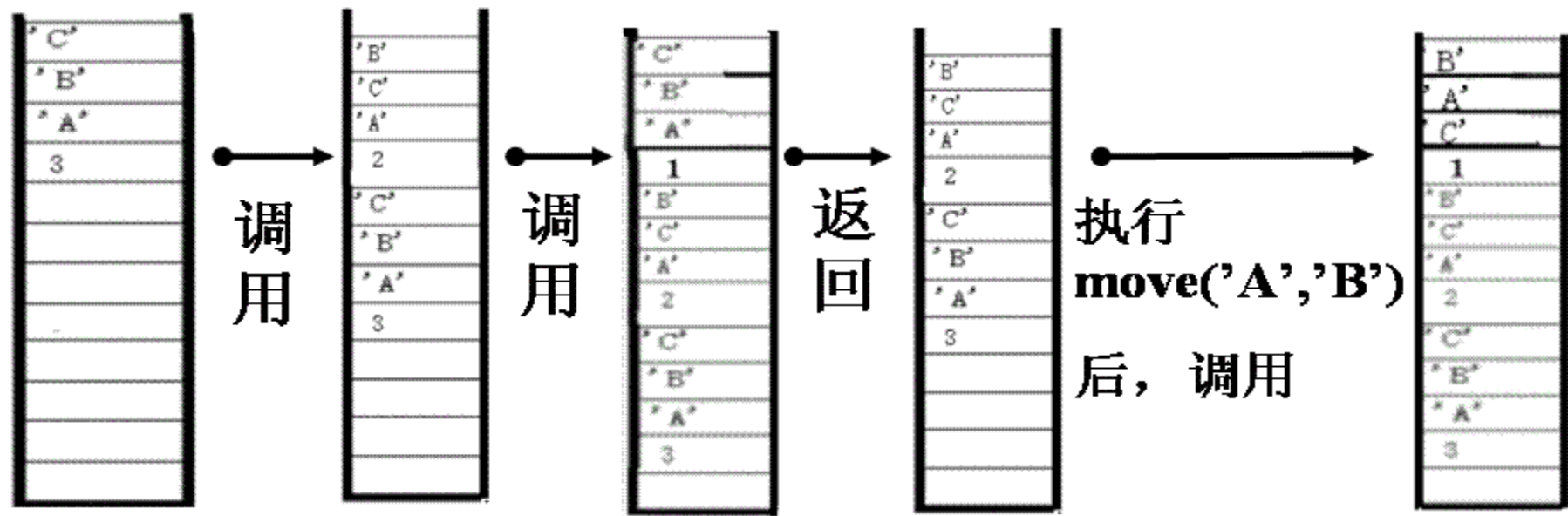


左边是H机器,它执行hanoi程序的指令流,并通过内部栈和子程序交换信息。
右边是它们所使用的栈

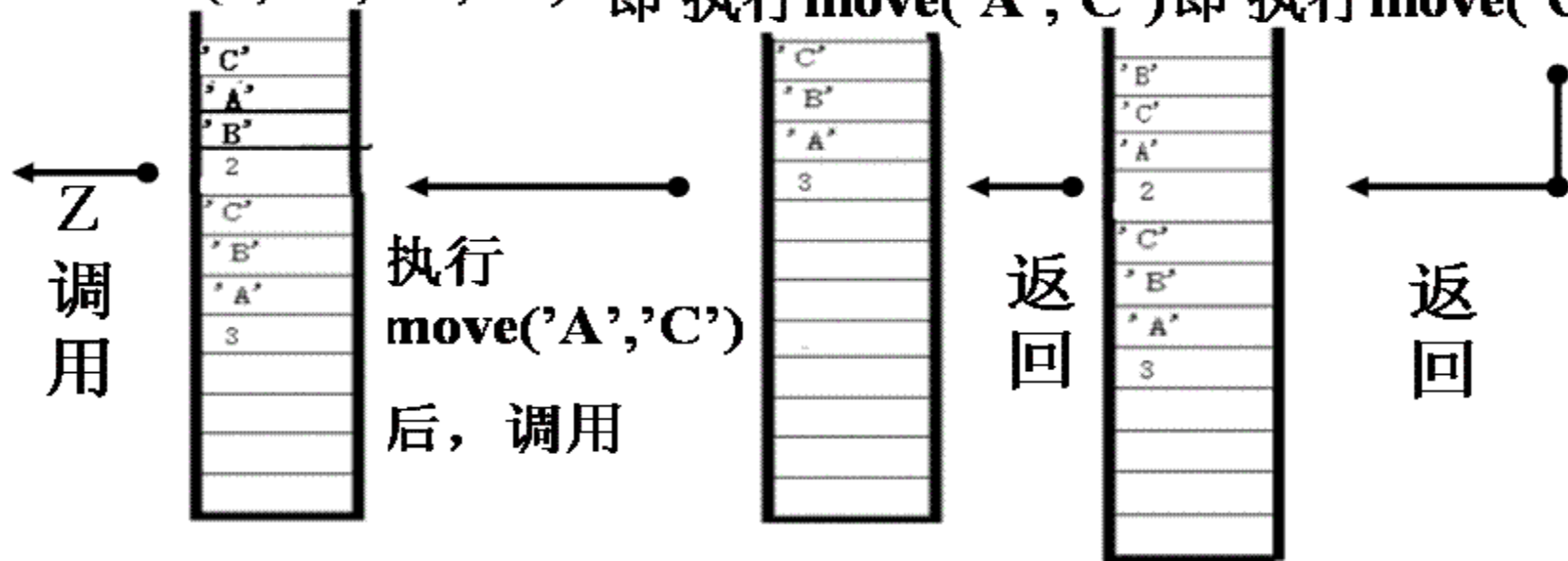


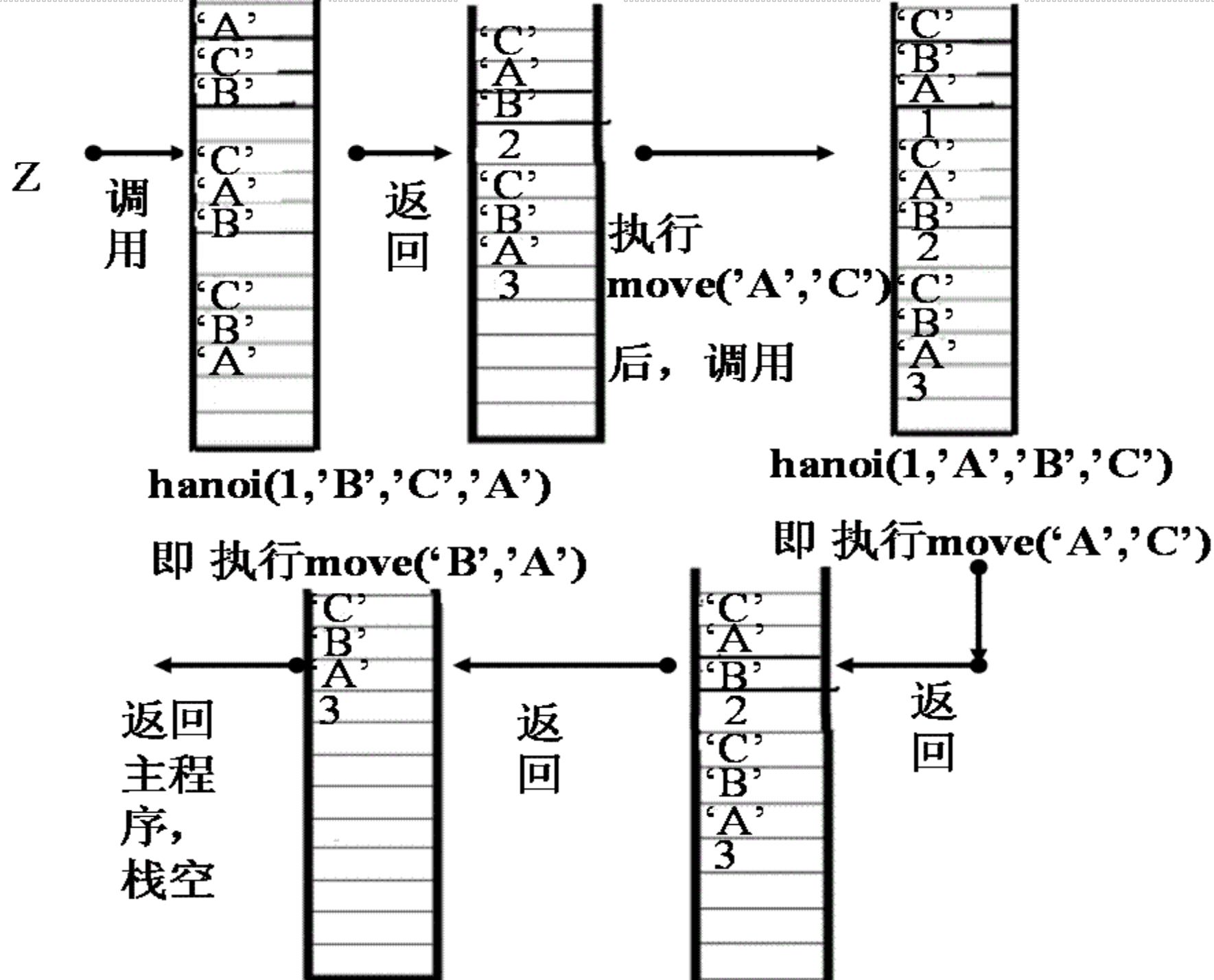


递归运行时, 堆栈的进退以及通过堆栈传递参数



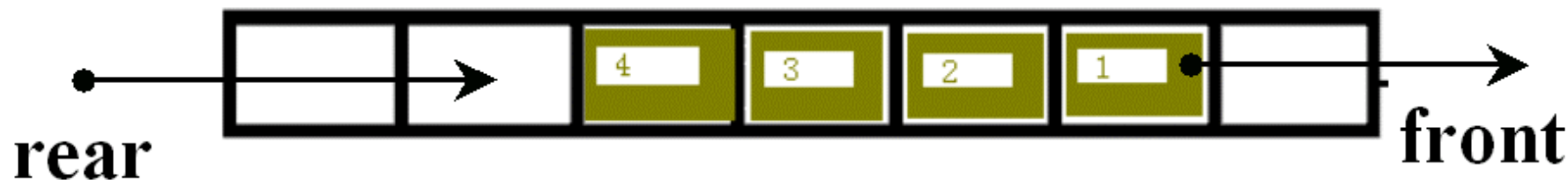
hanoi(3,'A','B','C') hanoi(1,'A','B','C') hanoi(1,'C','A','B')
 hanoi(2,'A','C','B') 即 执行move('A','C') 即 执行move('C','B')





2.6 队 列(queue)

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





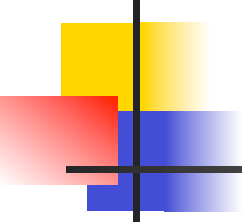
队列的应用

- 消息缓冲器
- 邮件缓冲器
- 计算机的硬设备之间的通信也需要队列作为数据缓冲
- 操作系统中也使用队列



队列的抽象数据类型

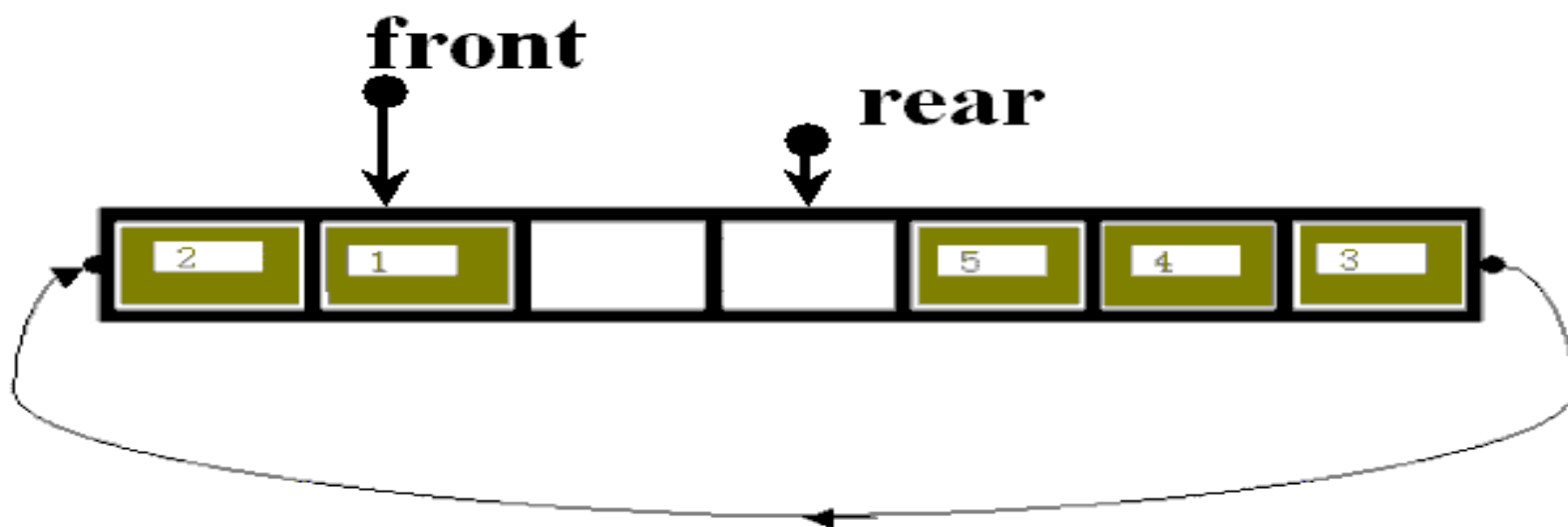
```
template <class T> class Queue
{ // 队列的元素类型为T，它们是按先后次序的
  //线性表结构
  // 一般使用front和rear指示队列的前端和尾端
  // 用curr_len 存储当时的队列长度
  //栈的运算集为：
  Queue(int s); //创建队列实例，最大长度为s
  ~ Queue();    //该实例消亡,释放全部空间
```



```
void EnQueue(T item);    //item进入队列前端  
//返回队列的前端元素内容，并从队列删去  
T DeQueue();  
//返回队列的前端元素内容，但不从尾部删去  
T GetFirst();  
void MakeEmpty();    //变为空队列  
int IsEmpty();        //返回真，若队列已空  
int IsFull();         //返回真，若队列已满  
};
```

2.6.1 顺序队列

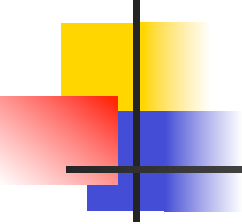
- 使用顺序表来实现队列。用向量存储队列元素，并用两个变量分别指向队列的前端和尾端





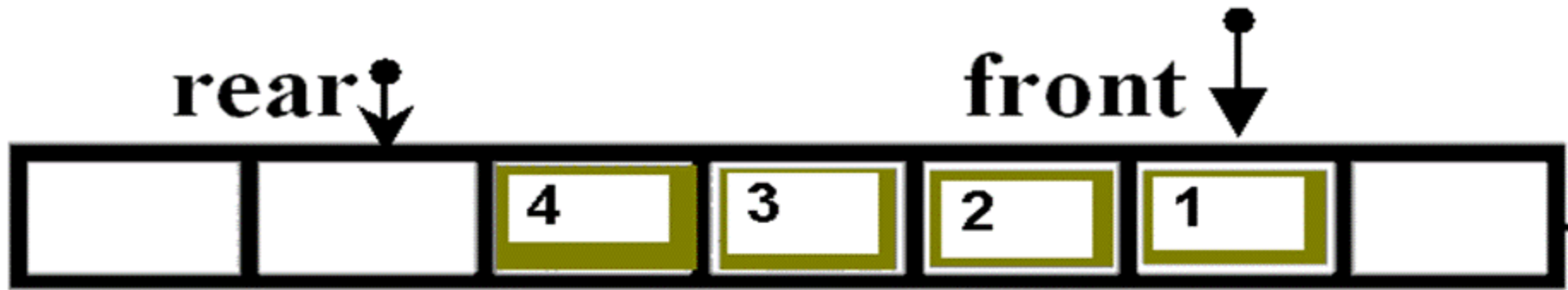
队列的类定义

```
#include <assert.h>    //包括逻辑断言函数库  
  
class Queue  
{  
private:  
    float *Qlist; //存放数据元素的向量  
    //队列前端和尾端向量的下标值  
    int front, rear;  
    //当新元素进入或队列尾端的元素取出，这两  
    //个变量值随之增减
```



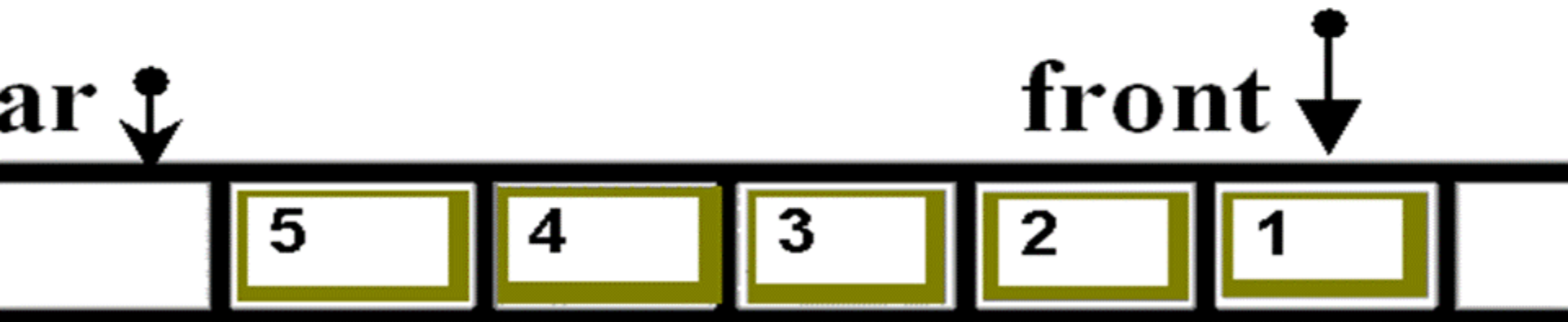
```
int  maxsize;  //队列最大长度  
int  curr_len;  //队列当前长度  
public:  
    //创建队列实例，指定该实例的向量空间长度  
    Queue(int size);  
    ...  
};
```

新元素压入队列的尾端



(a)

(a) 经过 enqueue 放进元素5以后变为 (b)



(b)



队列的创建

```
//队列实例的创建，指定最大长度7
maxsize=size;
//动态开辟向量存储空间
Qlist = new float[maxsize];
Queue::Queue(int size=7) {
    // 判new成功否，否则断言程序异常，
    //退出运行
    assert(ElmList!=NULL);
    front=rear=curr_len=0; // 让队列为空
}
```



压入队列顶

```
void Queue::EnQueue(float item)  
{  
    //判队列满，否则队列溢出异常，退出运行  
    assert(!curr_len== maxsize);  
    curr_len ++;  
    Qlist[rear] = item;           //在队列尾端加入队列  
    rear = (rear + 1) % maxsize; //  
}
```

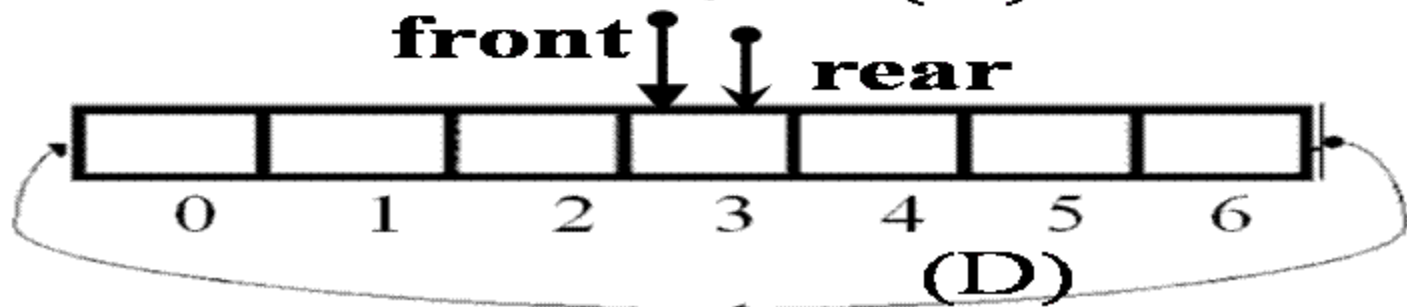
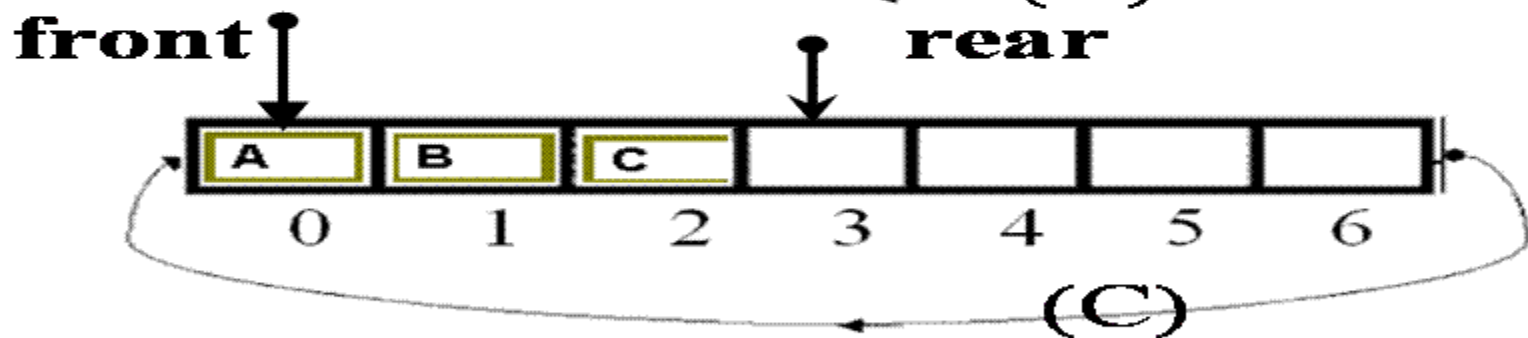
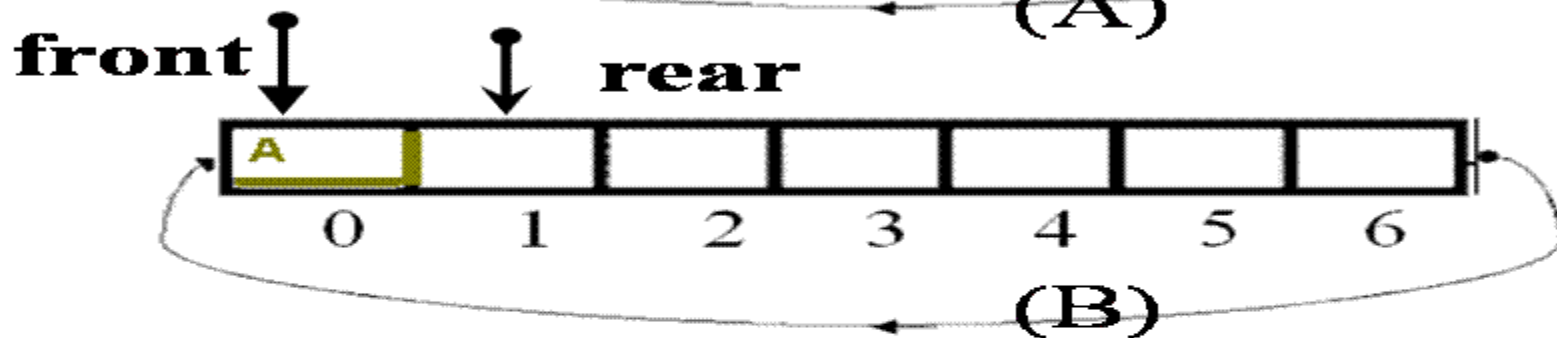
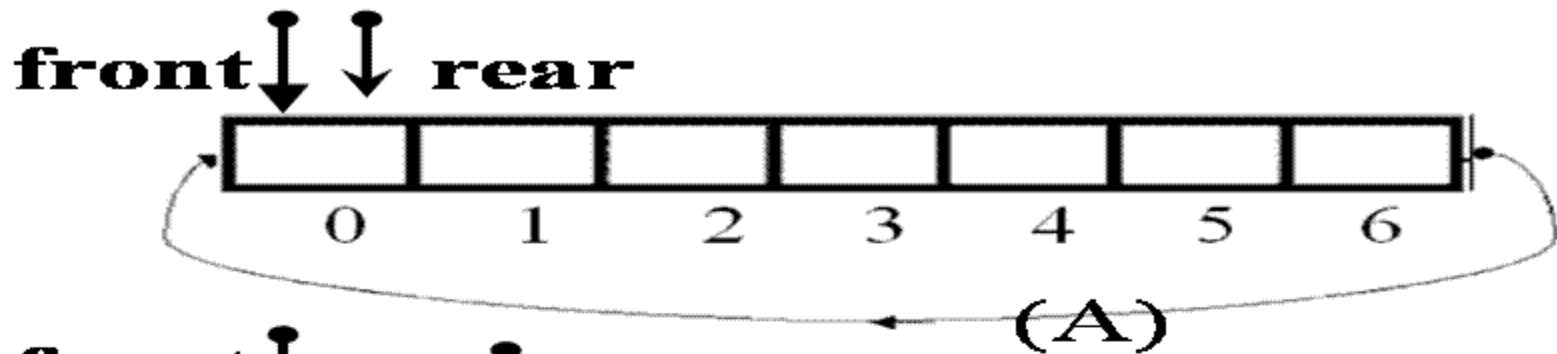



算法2-15 从队列前端取出

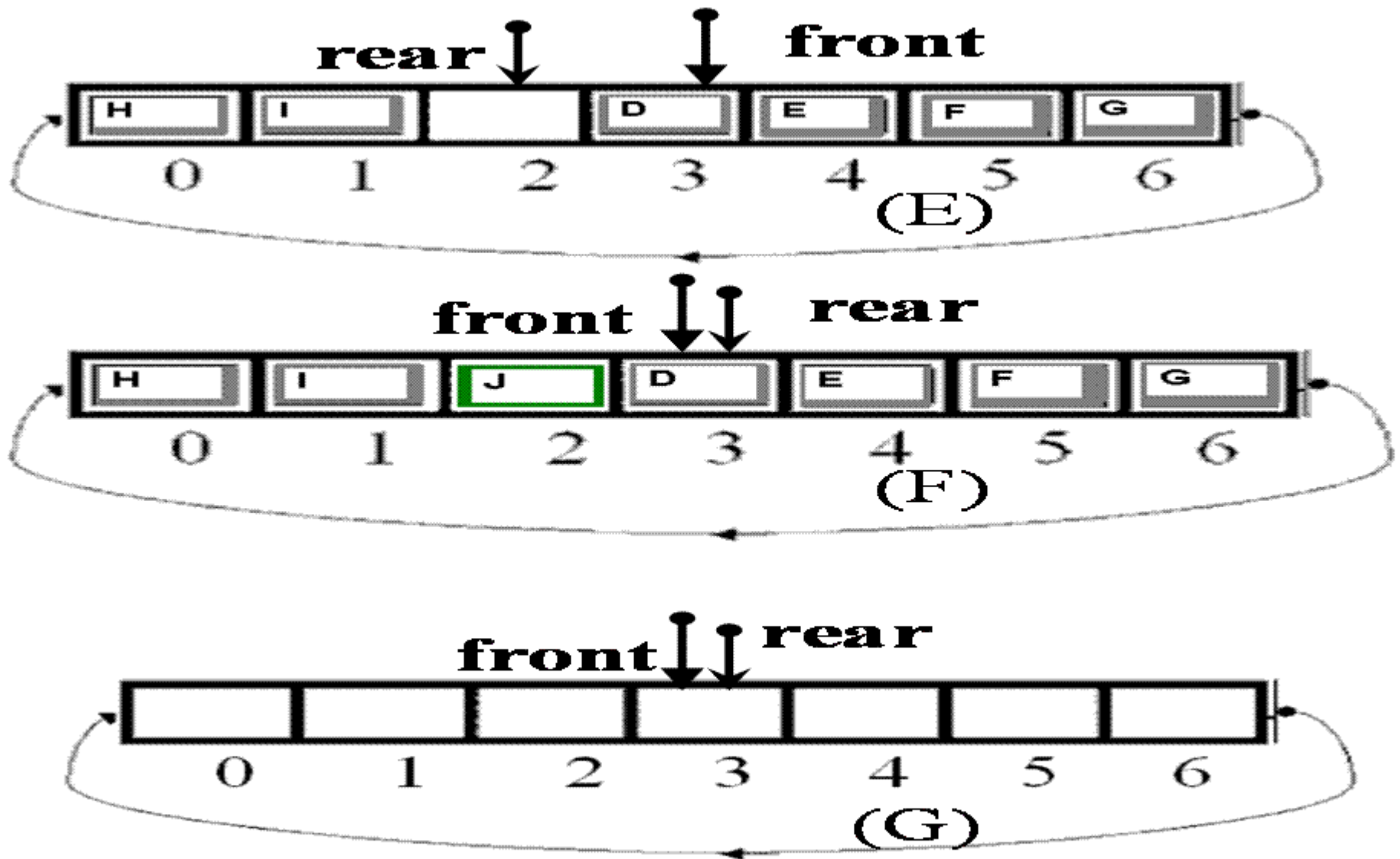
```
float Queue::DeQueue()
{
    float temp;
    //判队列非空，否则队列已空，异常退出运行
    assert(!curr_len== 0);
    temp = Qlist[front];
    curr_len--;
    front = (front+1) % maxsize;
    return temp;
}
```

北京大学软件与微电子学院 ©版权所有，转载或翻印必究

队列的运行示意图



队列的运行示意图

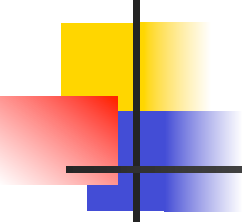




2.6.2 链式队列

- 单链表队列
- 链接指针的方向是从队列的前端向尾端链接

队列的类定义



```
#include <assert.h>
```

```
Class Queue {
```

```
    // linked Queue, 单链表, 其结点类型为ListNode
```

```
private:
```

```
    ListPtr front,rear;
```

```
    int curr_len;
```

```
public:
```

```
    //创建一个空队列, 不用指定最大长度
```

```
    Queue::Queue() {
```

```
        front = rear = NULL;
```

```
        curr_len = 0;
```

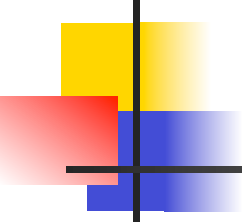
```
    };
```

```
    ...
```



将元素加入队列前端

```
void Queue::EnQueue( ELEM item)  
{  
    ListPtr temp;  
    temp = new ListNode;  
    assert(!temp==NULL); //若无存储空间则异常  
    temp->data = item;  
    temp->link = NULL;  
    if(curr_len != 0) { //队列尾端指针rear非NULL
```



```
rear->link = temp;
    rear = temp;    //新队列尾端指针
}
else
    //只有一个结点时，队列前端和尾端指针相同
    front = rear = temp;
curr_len++;
}
```



自单链队列前端取出

```
ELEM Queue:: DeQueue()
```

```
{
```

```
    //判队列非空，否则队空异常退出
```

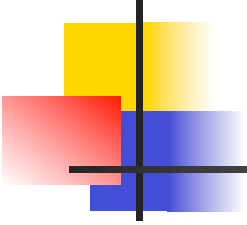
```
    assert(curr_len != 0);
```

```
    //暂存队列顶内容
```

```
    ELEM result = front->data;
```

```
    ListPtr temp;
```

```
    temp = front;           //老前端指针
```

```
front = front->link ; //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
- 方便访问队列内部元素

- 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



变种的栈或队列结构

- 双端队列
- 双栈
- 超队列
- 超栈



Homework

- 1.将两个非递减的有序链表合为一个非递减的有序链表。（要求利用原来两个链表的存储空间，不另外占用其他空间，表中不允许有重复数据）
- 2.将两个非递减的有序链表合并为一个非递增的有序链表。（要求同上）
- 3.设计算法，通过一趟遍历确定单链表中值最大的结点。
- 4.设计算法，通过遍历一趟，将链表中所有结点的链接方向逆转（使用原表存储空间）
- 5.设计一个算法，删除递增有序链表中值大于 $\min(i)$ 且小于 $\max(k)$ 的所有元素（ i 和 k 是给定的两个参数，其值可以和表中的元素相同，也可以不同）