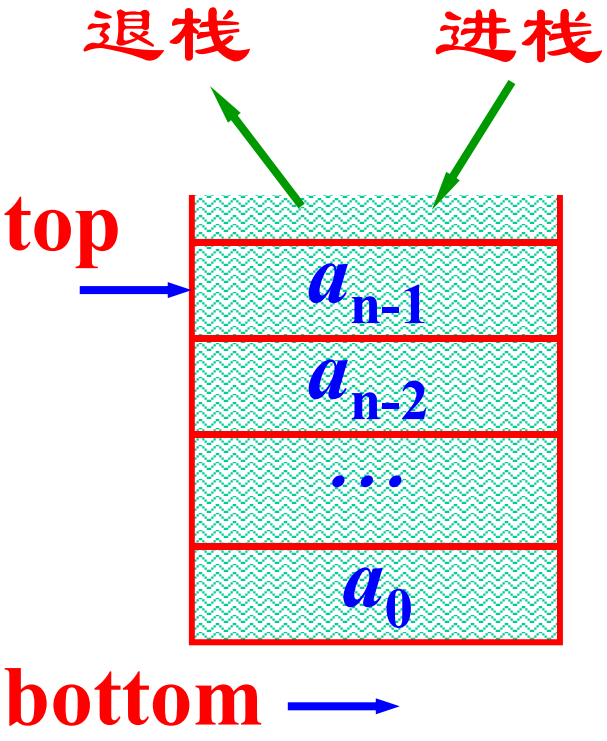
第三章 栈和队列

- 栈
- 栈与递归
- 队列
- 优先级队列
- 双端队列
- 本章小结

3.1 栈 (Stack)

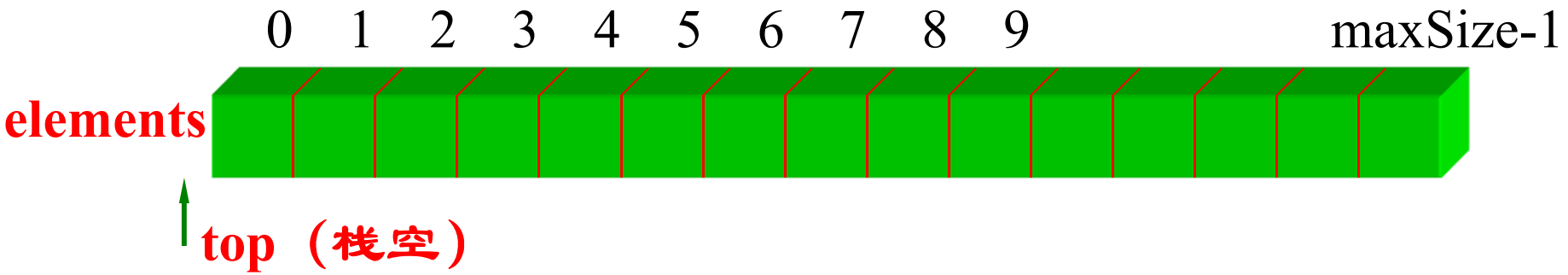
- 只允许在一端插入和删除的线性表。
- 允许插入和删除的一端称为栈顶(top)，另一端称为栈底(bottom)。
- 特点
  后进先出 (LIFO)



栈的抽象基类

```
template <class Type> class Stack {
public:
    Stack ( );  //构造函数
    virtual void Push ( const Type &x ) = 0;  //进栈
    virtual bool Pop ( Type &x ) = 0;  //出栈
    virtual bool GetTop ( Type &x ) const = 0;
    //取栈顶元素
    virtual bool IsEmpty ( ) const = 0;  //判栈空否
    virtual bool IsFull ( ) const = 0;  //判栈满否
    virtual int GetSize ( ) const = 0;
};
```

栈的数组存储表示 — 顺序栈



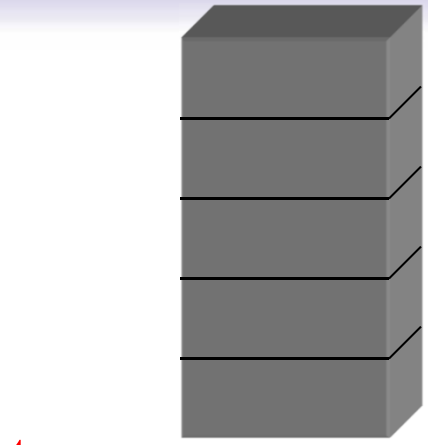

```
#include <assert.h>
template <class Type> class SeqStack {
private:
    int top;  //栈顶指针
    Type *elements;  //栈元素数组
    int maxSize;  //栈最大容量
    void OverflowProcess ( );  //栈的溢出处理
```

```
public:
   SeqStack ( int sz = 50 );  //构造函数
   ~SeqStack ( ) { delete [ ] elements; }
   void Push ( const Type &x );  //进栈
   bool Pop ( Type &x );  //出栈
   bool GetTop ( Type &x );  //取栈顶
   void MakeEmpty ( ) { top = -1; }  //置空栈
   bool IsEmpty ( ) const
        { return ( top == -1 ) ? true : false; }
   int IsFull ( ) const
      { return ( top == maxSize-1 ) ? true : false; }
   int  GetSize ( ) const { return top+1; }
};
```

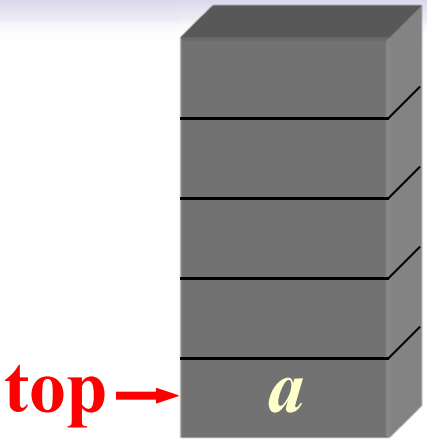
```
template <class Type> SeqStack <Type> ::
SeqStack ( int sz ) : top(-1), maxSize(sz) {
    elements = new Type [maxSize];
    assert ( elements != NULL );  //断言
}
```

```
template <class Type> void SeqStack <Type> ::
OverflowProcess ( ) {
    Type *NewArray = new Type
                    [maxSize+stackIncreament];
    if ( NewArray == NULL )
       { cerr << "存储分配失败！ " << endl;  exit(1); }
    for ( int i =0; i <= top; i++ )
        NewArray[i] = elements[i];
    maxSize = maxSize+stackIncreament;
    delete [ ] elements;
}
```

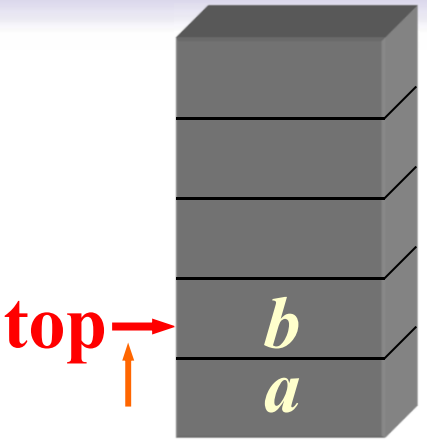
- **进栈操作**
  - **设栈的最后允许存放位置为*maxSize*-1，首先判断栈是否已满。**
    - **如果栈顶指针*top* = *maxSize*-1（栈已满），则栈中所有位置均已使用，若再有新元素进栈，将发生溢出，没有空间可以再利用，程序转入出错处理。**
    - **如果*top* < *maxSize*-1，则先让栈顶指针进1，指到当前可加入新元素的位置，再按栈顶指针所指位置将新元素插入，该元素成为新的栈顶元素。**



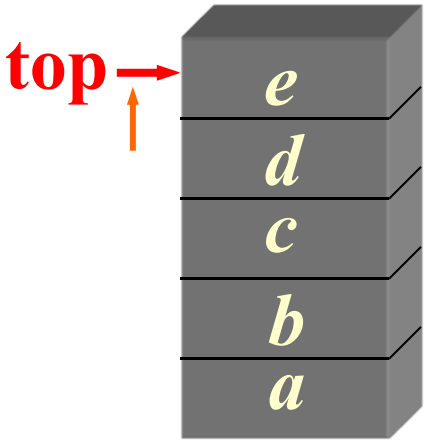
空栈



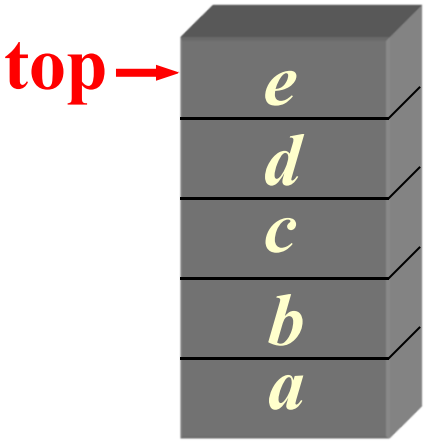
a 进栈



b 进栈

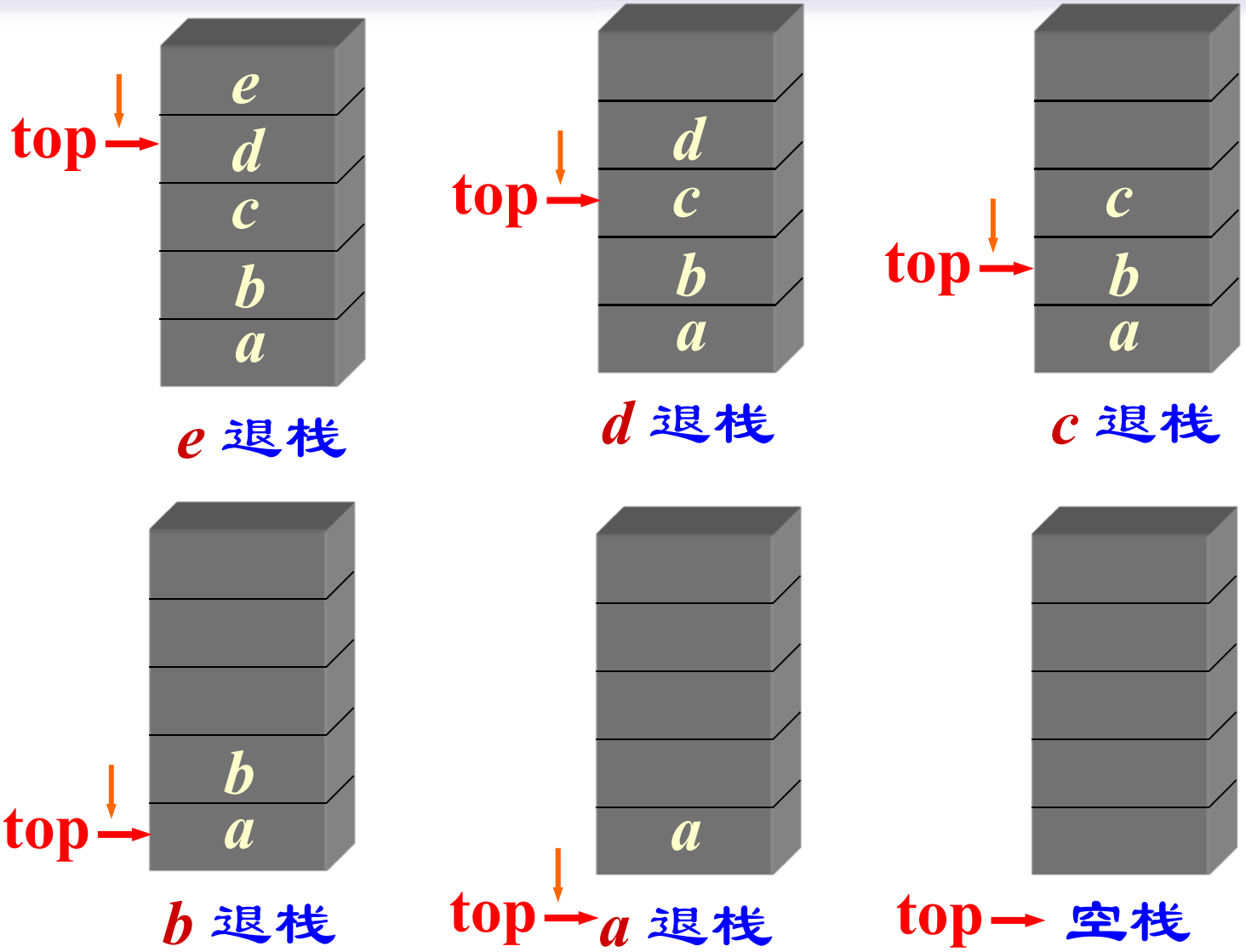


c, d, e 进栈



f 进栈溢出

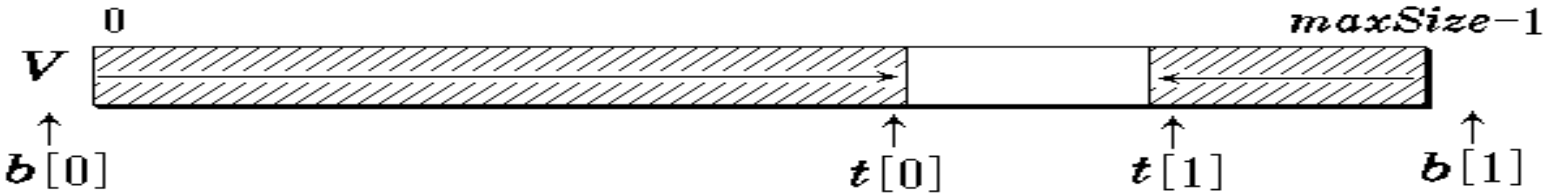
- **退栈操作**
  - 极端情况出现在栈底：
    - 如果*top* = -1（在退栈时发现是空栈），则执行栈空处理。
      - 栈空处理不是出错处理，而是使用该栈的算法结束时需要执行的处理。
  - 如果当前*top* >= 0，则可将栈顶指针减1，等于栈顶退回到次栈顶位置。


top
e
d
c
b
a
e 退栈
top
d
c
b
a
d 退栈
top
c
b
a
c 退栈
top
b
a
b 退栈
top
a
top a 退栈
top 空栈

```
template <class Type>
void SeqStack <Type> :: Push ( const Type &x ) {
    if ( IsFull ( ) == true ) OverflowProcess ( );
    elements[++top] = x;  //加入新元素
}

template <class Type>
bool Seqstack <Type> :: Pop ( Type &x ) {
    if ( IsEmpty ( ) == true ) return false;  //栈空不退栈
    x = elements[top];  top--;
    return true;  //退出栈顶元素
}
```

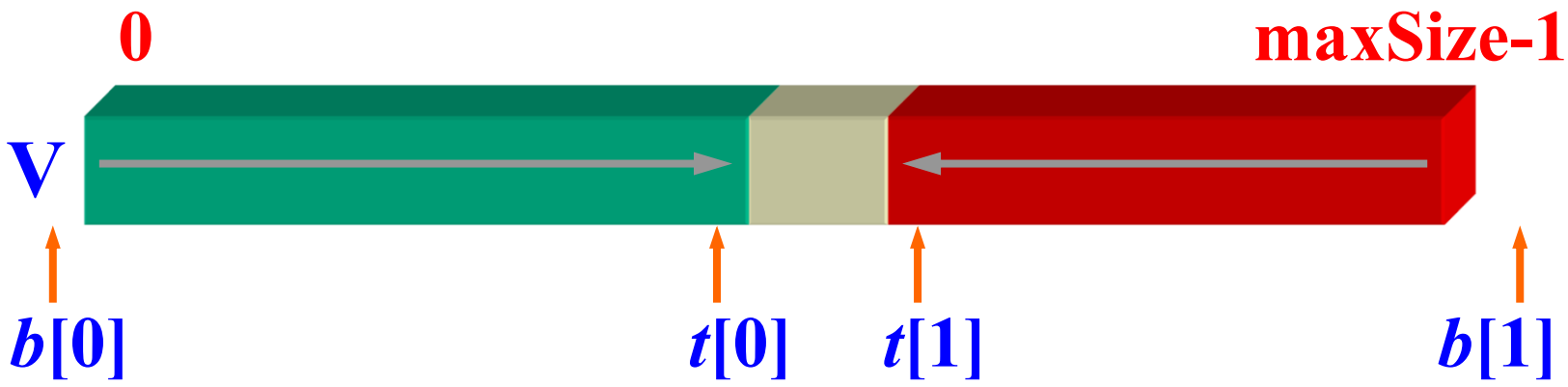
```
template <class Type>
bool SeqStack <Type> :: GetTop ( Type &x ) {
    if ( IsEmpty ( ) == true ) return false;
    x = elements[top];  //取出栈顶元素
    return true;
}
```

- 当栈满时要发生溢出，程序报错并终止运行。
  - 为避免这种情况，需要为栈设立一个足够大的空间。如果空间设置过大，而栈中实际只有几个元素，是一种空间浪费。
    - 当程序只需要一个栈时，溢出问题不算太大。
    - 当需要两个栈时，可定义一个足够大的栈空间。



  - 两个栈的栈顶都向中间伸展，直到两个栈的栈顶相遇，发生溢出。
  - 两个栈的大小不是固定不变，一个栈可能进栈元素多而体积大些，另一个则可能小些。
  - 两个栈共用一个栈空间，互相调剂，灵活性强。

双栈共享一个栈空间



- 两个栈共享一个数组空间 V[maxSize] 。
- 设立栈顶指针数组 t[2] 和栈底指针数组 b[2] ，
  t[i] 和 b[i] 分别指示第 i 个栈的栈顶与栈底。
- 初始 t[0] = b[0] = -1， t[1] = b[1] = maxSize 。
- 栈满条件： t[0]+1 == t[1] //栈顶指针相遇
- 栈空条件： t[0] = b[0] 或 t[1] = b[1] //栈顶指针退到栈底

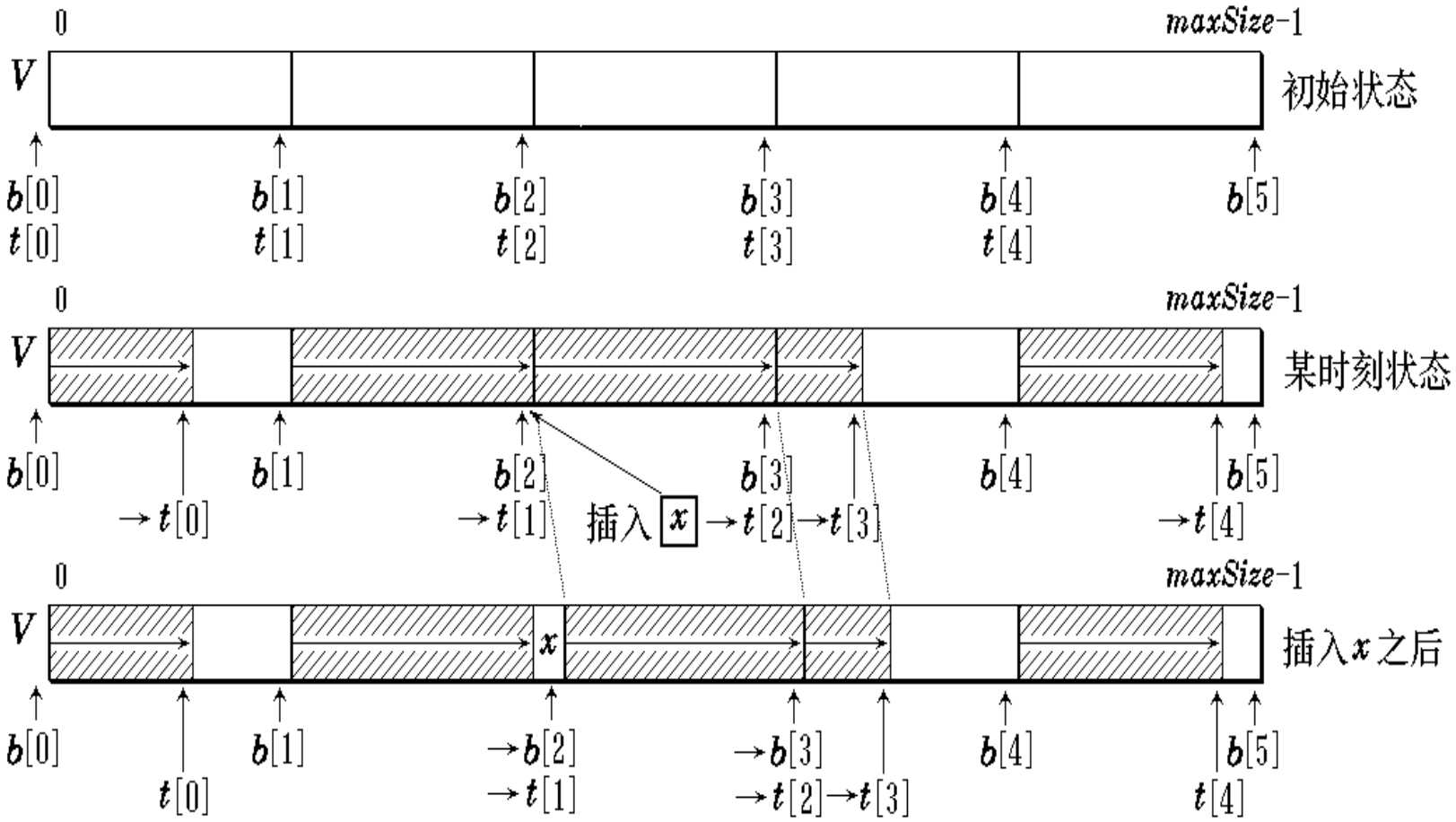
```
bool Push ( DualStack &DS, Type x, int d ) {
    if ( DS.t[0]+1 == DS.t[1] ) return false;
    if ( d == 0 ) DS.t[0]++;  else DS.t[1]--;
    DS.Vector[DS.t[i]] = x;
    return true;
}

bool Pop ( DualStack &DS, Type &x, int d ) {
    if ( DS.t[i] == DS.b[i] ) return false;
    x = DS.Vector[DS.t[i]];
    if ( d == 0 ) DS.t[0]--;  else DS.t[1]++;
    return true;
}
```

- 当程序中同时存在几个栈，问题较复杂。
  - 各个栈所需空间在运行中动态变化。
  - 如果给几个栈分配同样大小的空间，可能在实际运行中，有的栈膨胀得快，很快就产生溢出。
  - 其它栈可能还有许多空闲的空间，必须调整栈的空间，防止栈的溢出。
  - 在插入时元素的移动量非常大，因而代价较高。
  - 特别是当整个存储空间即将充满时，问题更加严重。

多栈处理 栈浮动技术

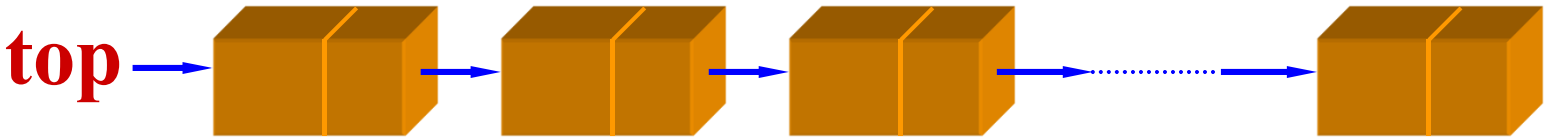
- n 栈共享一个数组空间 $V[m]$
- 设立栈顶指针数组 $t[n+1]$ 和
  栈底指针数组 $b[n+1]$
- $t[i]$ 和 $b[i]$ 分别指示第i个栈的栈顶与栈底，$b[n]$ 作为控制量，指到数组最高下标。
- 各栈初始分配空间 $s = \lfloor m / n \rfloor$
- 指针初始值 $t[0] = b[0] = -1, b[n] = m-1$

  $t[i] = b[i] = b[i-1] + s, i = 1, 2, \ldots, n-1$



插入新元素时的栈满处理 *StackFull ( )*

栈的链接存储表示 — 链式栈



- 链式栈无栈满问题，空间可扩充；
- 插入与删除仅在栈顶处执行；
- 链式栈的栈顶在链头；
- 适合于多栈操作。

链式栈类定义

```
template <class Type> class LinkedStack;
template <class Type> class StackNode {
friend class Stack <Type>;
private:
   Type data;  //结点数据
   StackNode <Type> *link;  //结点链指针
public:
   StackNode ( Type d = 0,
        StackNode <Type> *next = NULL ) :
                        data(d), link(next) { }
};
```

```
template <class Type> class LinkedStack {
private:
    StackNode <Type> *top;  //栈顶指针
public:
    LinkedStack ( ) : top(NULL) { }
    ~LinkedStack { MakeEmpty ( ); }
    void Push ( Type &x );  //进栈
    bool Pop ( Type &x );  //退栈
    bool GetTop ( Type &x );  //读取栈顶元素
    void MakeEmpty ( );  //实现与 ~LinkedStack( ) 同
    bool IsEmpty ( ) { return ( top == NULL) ? true : false; }
    int GetSize ( ) const;
    friend ostream & operator <<
                ( ostream &os, LinkedStack <Type> &s );
};
```

```
template <class Type>
LinkedStack <Type> :: MakeEmpty ( ) {
    StackNode <Type> *p;
    while ( top != NULL )  //逐结点回收
        { p = top;  top = top->link;  delete p; }
}

template <class Type>
void LinkedStack <Type> :: Push ( const Type &x ) {
//新结点链入 *top 之前，并成为新栈顶
   top = new StackNode <Type> ( x, top );
   assert ( top != NULL );
}
```

```
template <class Type>
bool LinkedStack <Type> :: Pop ( Type &x ) {
    if ( IsEmpty ( ) == true ) return false;
    StackNode <Type> *p = top;
    x = p->data;  top = top->link;  //修改栈顶
    delete p;  return true;
}

template <class Type>
int LinkedStack <Type> :: GetTop ( Type &x ) const {
    if ( IsEmpty ( ) == true ) return false;
    x = top->data;  return true;
}
```

随堂练习

例1：设栈的输入序列为1, 2, 3, …, n, 输出序列为a1, a2, …, an, 若存在1≤k≤n, 使得ak=n, 则当k≤i≤n时, ai为_____。

例2：设栈的输入序列为1, 2, …, n, 若输出序列的第一个元素为n, 则第i个输出元素为_____。

例3：在给定进栈序列的前提下，判断合理的出栈序列以及计算出栈序列的不同种数。

例4：设栈S和队列Q的初始状态为空，元素1, 2, 3, 4, 5, 6依次通过栈S，一个元素出栈后即进入队列Q，若6个元素出队的序列为2, 4, 3, 6, 5, 1，则栈S的容量至少应该是_____。

例1：设栈的输入序列为1, 2, 3, …, n，输出序列为a1, a2, …, an，若存在1≤k≤n，使得ak=n，则当k≤i≤n时，ai为不确定。

由于输出序列a1, a2, …, an并没有明确标示出具体的序列值，只是一个变量序列，即可能是根据输入序列所能形成的任何出栈序列；所以，当1≤k≤n且ak=n时，并不能确定k≤i≤n时的ai。

例2：设栈的输入序列为1, 2, …, n，若输出序列的第一个元素为n，则第i个输出元素为n-i+1。

若栈输出的第一个元素为n，则由栈定至栈底顺序存放的必然是n, n-1, …, 2, 1，故第i个输出的元素为n-i+1。

例3：在给定进栈序列的前提下，判断合理的出栈序列以及计算出栈序列的不同种数。

例4：设栈S和队列Q的初始状态为空，元素1, 2, 3, 4, 5, 6依次通过栈S，一个元素出栈后即进入队列Q，若6个元素出队的序列为2, 4, 3, 6, 5, 1，则栈S的容量至少应该是 3 。

铁路进行列车调度时，常把站台设计成栈式结构的站台。

(1) 设有编号为1, 2, 3, 4, 5, 6的六辆列车，顺序入栈式构的站台，则可能的出栈序列有多少种?

(2) 若进站的六辆列车顺序如上所述，那么是否能够得到435612、325641、154623和135426的出栈序列，如果不能，说明为什么不能；如果能，说明如何得到（即写出“进栈”或“出栈”的序列）。

基本思路：

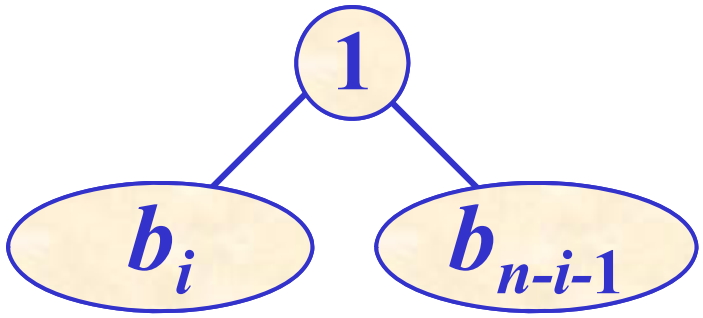
6辆火车顺序进栈，1号火车最先压进栈中。接下来，有可能1号火车直接出栈，2号以及其它火车再顺序进栈，在这些火车进栈的过程中，可能有的火车在其后边火车进栈之前已出栈。也有可能，1号火车仍然保留在栈底，2号以及其它火车再顺序进栈，那么在这些火车进栈的过程中，可能有的火车在其后边火车进栈之前已出栈，而且也有可能1号火车在其它火车进栈的过程中也要出栈。问题所要求解的就是有多少种这样的出栈序列。

解法：

因为1号火车最先压入栈底，将1号火车作为一个基点，1号火车可作为栈顶存在，也可作为非栈顶存在。如果1号火车作为栈顶，也就是1号火车上面未压入任何火车，1号肯定最先出栈，那么如果想求解出栈序列种数，就只需考虑其它5辆火车的进栈出栈组合数就可以了。如果1号火车上面已经压入若干火车，而且除这些火车以及1号火车之外，剩余火车仍然在栈外，这时不仅需要考虑1号火车上面已压入栈中的若干火车的进栈出栈组合数，而且也要考虑栈外剩余火车的进栈出栈组合数，将这两个组合数相乘，才是总的组合数。

计算具有n个元素的不同出列组合数

$$b_n = \sum_{i=0}^{n-1} b_i \cdot b_{n-i-1}$$



- b_i表示指定一个元素出栈前由i个元素构成的出栈序列；
- b_{n-i-1}表示指定一个元素出栈后由剩余$n-i-1$个元素构成的出栈序列；
- $b_i * b_{n-i-1}$表示由指定元素、指定元素之前出栈b_i和指定元素出栈之后出栈b_{n-i-1}组成的出栈序列。
  - 不同出栈序列数等于之前可能的不同出栈序列数与之后可能的不同出栈序列数的乘积。

括号匹配

```
const int maxLength = 100;
void PrintMatchPairs ( char *expression ) {
  Stack <int> s(maxLength);  int j, length = strlen ( expression );
  for ( int i = 1; i < length; i++ ) {
      if ( expression[i-1] == '(' ) s.Push ( i );
      else if ( expression[i-1] == ')' ) {
            if ( s.Pop ( j ) == true )
                cout << j << "与" << i << "匹配" << endl;
            else cout << "没有与第" << i
                      << "个右括号匹配的左括号！ " << endl; }
   while ( s.IsEmpty ( ) == false ) {
      s.Pop ( j );
       cout << "没有与第" << j
            << "左括号相匹配的右括号！ " << endl; }
}
```

表达式求值

- 一个表达式由操作数（亦称运算对象）、操作符（亦称运算符）和分界符组成。
- 算术表达式有三种表示
  - 中缀(Infix)表示

    <操作数> <操作符> <操作数>，如 A+B；
  - 前缀(Prefix)表示

    <操作符> <操作数> <操作数>，如 +AB；
  - 后缀(Postfix)表示

    <操作数> <操作数> <操作符>，如 AB+。

表达式示例

中缀表达式 A + B * (C - D) - E / F

后缀表达式 A B C D - * + E F / -

- 表达式中相邻两个操作符的计算次序为：
  - 优先级高的先计算；
  - 优先级相同的自左向右计算；
  - 当使用括号时从最内层括号开始计算。

- **一般表达式的操作符有4种类型：**

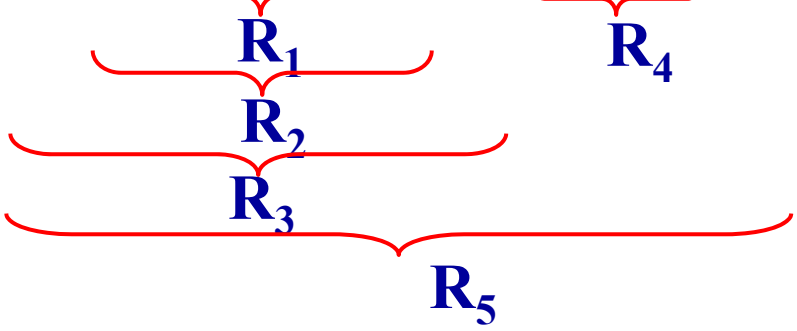
1. **算术操作符　如双目操作符（+、-、*、/和%）以及单目操作符（-）。**
2. **关系操作符　包括<、<=、==、!=、>=、>，这些操作符主要用于比较。**
3. **逻辑操作符　如与(&&)、或(||)、非(!)。**
4. **括号‘(’和‘)’　它们的作用是改变运算顺序。**

C++中操作符的优先级

优先级	操作符
1	单目-、！
2	*、/、%
3	+、-
4	<、<=、>、>=
5	==、!=
6	&&
7	||

应用后缀表示计算表达式的值

- 从左向右顺序地扫描表达式，并用一个栈暂存扫描到的操作数或计算结果。
- 在后缀表达式的计算顺序中已隐含了加括号的优先次序，括号在后缀表达式中不出现。
- 计算示例A B C D - * + E F / -



通过后缀表示计算表达式值的过程

- 顺序扫描表达式的每一项，根据它的类型做如下相应操作：
  - 若该项是操作数，则将其压栈；
  - 若该项是操作符<op>，则连续从栈中退出两个操作数Y和X，形成运算指令X<op>Y，并将计算结果重新压栈。
- 当表达式的所有项都扫描并处理完后，栈顶存放的就是最后的计算结果。

步	输入	类 型	动 作	栈内容
1			置空栈	空
2	A	操作数	进栈	A
3	B	操作数	进栈	AB
4	C	操作数	进栈	ABC
5	D	操作数	进栈	ABCD
6	-	操作符	D、C 退栈，计算 C-D，结果 R_1 进栈	ABR_1
7	*	操作符	R_1、B 退栈，计算 B*R_1，结果 R_2 进栈	AR_2
8	+	操作符	R_2、A 退栈，计算 A+R_2，结果 R_3 进栈	R_3

步	输入	类 型	动 作	栈内容
9	E	操作数	进栈	R_3E
10	F	操作数	进栈	R_3EF
11	/	操作符	F、E 退栈，计算 E/F，结果 R_4 进栈	R_3R_4
12	-	操作符	R_4、R_3 退栈，计算 R_3-R_4，结果 R_5 进栈	R_5

```
void Calculator :: Run ( ) {
  char ch;   double newoperand;
  while ( cin >> ch,  ch != ';' ) {
    switch ( ch ) {
      case '+' :  case '-' :   case '*' :   case '/' :
                  DoOperator ( ch );  break;  //计算
      default :  cin.putback ( ch );
                 //将字符放回输入流
                 cin >> newoperand;  //读操作数
                 S.Push ( newoperand );
    }
  }
}
```

```
void DoOperator ( char op ) {
//从栈S中取两个操作数，形成运算指令并计算进栈
   double left, right;
   bool succ = S.GetTop ( right );  S.Pop ( );
   if ( succ ) { succ = S.GetTop ( left );  S.Pop ( ); }
   if ( succ == false ) return;
   switch ( op ) {
      case '+': S.Push ( left + right);  break;  //加
      case '-': S.Push ( left - right);  break;  //减
      case '*': S.Push ( left * right);   break;  //乘
      case '/': if ( right != 0.0 ) S.Push ( left / right);  //除
                else { cout << "除数为0！ \n" );  exit(1); }
   }
}
```

利用栈将中缀表示转换为后缀表示

- 使用栈可将表达式的中缀表示转换成它的前缀表示和后缀表示。
- 为了实现这种转换，需要考虑各操作符的优先级。

各个算术操作符的优先级

操作符 ch	#	(	^	*, /, %	+, -	)
isp（栈内）	0	1	7	5	3	8
icp（栈外）	0	8	6	4	2	1

- isp叫做栈内(in stack priority)优先数，icp叫做栈外(in coming priority)优先数。
- 操作符优先数相等的情况只出现在括号配对或栈底的“#”号与输入流最后的“#”号配对时。

中缀表达式转换为后缀表达式的算法

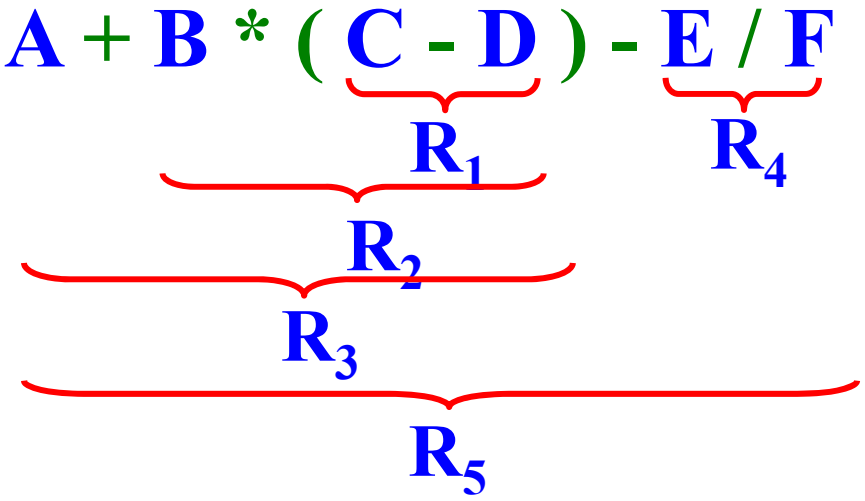
- 操作符栈初始化，将结束符‘#’进栈。然后，读入中缀表达式字符流的首字符ch。
- 重复执行以下步骤，直到ch = ‘#’，同时栈顶的操作符也是‘#’，停止循环。
  - 若ch是操作数直接输出，读入下一个字符ch。
  - 若ch是操作符，判断ch的优先级icp和位于栈顶的操作符op的优先级isp：
    - 若 icp (ch) > isp (op)，令ch进栈，读入下一个字符ch。
    - 若 icp (ch) < isp (op)，退栈并输出。
    - 若 icp (ch) == isp (op)，退栈但不输出，若退出的是“(”号，读入下一个字符ch。
- 算法结束，输出序列即为所需的后缀表达式。

步	输入	栈内容	语义	输出	动作
1		#			栈初始化
2	A	#		A	操作数A输出，读字符
3	+	#	+ > #		操作符+进栈，读字符
4	B	#+		B	操作数B输出，读字符
5	*	#+	* > +		操作符*进栈，读字符
6	(	#+*	(> *		操作符(进栈，读字符
7	C	# +*(		C	操作数C输出，读字符
8	-	#+*(	- > (		操作符-进栈，读字符
9	D	#+*(-		D	操作数D进栈，读字符
10	)	#+*(-	) < -	-	操作符-退栈输出
11		#+*(	) = (		(退栈，消括号，读字符

步	输入	栈内容	语义	输出	动作
12	-	#+*	- < *	*	操作符*退栈输出
13		#+	- < +	+	操作符+退栈输出
14		#	- > #		操作符-进栈，读字符
15	E	#-		E	操作数E输出，读字符
16	/	#-	/ > -		操作符/进栈，读字符
17	F	#-/		F	操作数F输出，读字符
18	#	#-/	# < /	/	操作符/退栈输出
19		#-	# < -	-	操作符-退栈输出
20		#	# = #		#配对，转换结束

```
void Postfix ( expression e ) {
  Stack <char> s;   char ch, op;
  s.Push ( '#' );  cin.Get ( ch );
 while ( s.IsEmpty ( ) == false && ch != '#' )
     if ( isdigit ( ch ) )
       { cout << ch;  cin.Get ( ch ); }
     else {
       if ( isp(s.GetTop ( )) < icp(ch) )
         { s.Push ( ch );  cin.Get ( ch ); }
       else if ( isp(s.GetTop ( )) > icp(ch) )
         { op = s.GetTop ( );  s.Pop ( );
           cout << op; }
        else { op = s.GetTop ( );  s.Pop ( );
               if ( op == '(' ) cin.Get ( ch ); }
     }
}
```

应用中缀表示计算表达式的值



- 使用两个栈，操作符栈OPTR (Operator)，操作数栈OPND (Operand)。
- 为了实现这种计算，需要考虑各操作符的优先级。

各个算术操作符的优先级

操作符 ch	#	(	*, /, %	+, -	)
isp (栈内)	0	1	5	3	6
icp (栈外)	0	6	4	2	1

- isp叫做栈内(in stack priority)优先数，
  icp叫做栈外(in coming priority)优先数。
- 操作符优先数相等的情况只出现在括号配对或栈底的“#”号与输入流最后的“#”号配对时。

中缀算术表达式求值

- 对中缀表达式求值的一般规则：

  (1) 建立并初始化OPTR栈和OPND栈，然后在OPTR栈中压入一个‘#’。

  (2) 从头扫描中缀表达式，取一字符送入ch。

  (3) 当ch != ‘#’或OPTR栈的栈顶 != ‘#’时, 执行以下工作, 否则结束算法。此时，在OPND栈的栈顶得到运算结果。

① 若ch是操作数，进OPND栈，从中缀表达式取下一字符送入ch；

② 若ch是操作符，比较icp(ch)的优先级和isp(OPTR)的优先级：

- 若icp(ch) ＞ isp(OPTR)，则ch进OPTR栈，从中缀表达式取下一字符送入ch；
- 若icp(ch) ＜ isp(OPTR)，则从OPND栈退出a2和a1，从OPTR栈退出θ, 形成运算指令 (a1)θ(a2)，结果进OPND栈；
- 若icp(ch) == isp(OPTR) 且ch == ‘)’，则从OPTR栈退出‘(’，对消括号，然后从中缀表达式取下一字符送入ch。

步	输入	OPND	OPTR	语义	动作
1			#		栈初始化
2	A				操作数A进栈，读字符
3	+	A	#	+ > #	操作符+进栈，读字符
4	B	A	#+		操作数B进栈，读字符
5	*	AB	#+	* > +	操作符*进栈，读字符
6	(	AB	#+*	(> *	操作符(进栈，读字符
7	C	AB	#+*(		操作数C进栈，读字符
8	-	ABC	#+*(	- > (	操作符-进栈，读字符
9	D	ABC	#+*(-		操作数D进栈，读字符

步	输入	OPND	OPTR	语义	动作
10	)	ABCD	#+*(-	) < -	D、C、-退栈，计算 C-D，结果 R_1 进栈
11		ABR_1	#+*(	) = (	(退栈，消括号，读字符
12	-	ABR_1	#+*	- < *	R_1、B、*退栈，计算 $B*R_1$，结果R_2进栈
13		AR_2	#+	- < +	R_2、A、+退栈，计算 $A+R_2$，结果R_3进栈
14		R_3	#	- > #	操作符-进栈，读字符
15	E	R_3	#-		操作数E进栈，读字符

步	输入	OPND	OPTR	语义	动作
16	/	R_3E	#-	/ > -	操作符/进栈，读字符
17	F	R_3E	#-/		操作数 F 进栈，读字符
18	#	R_3EF	#-/	# < /	F、E、/退栈，计算 E/F，结果 R_4 进栈
19		R_3R_4	#-	# < -	R_4、R_3、-退栈，计算 R_3-R_4，结果 R_5 进栈
20		R_5	#		算法结束，结果R_5

```
void InFixRun ( ) {
  Stack <char> OPTR, OPND;
  char ch, op;
  InitStack ( OPTR );  InitStack ( OPND );
  Push ( OPTR, '#' );  //两个栈初始化
  getchar ( ch );  //读入一个字符
  op = '#' ;
  while ( ch != '#' || op != '#' ) {
    if ( isdigit ( ch ) )  //是操作数，进栈
      { Push ( OPND, ch );  getchar ( ch ); }
    else {  //是操作符，比较优先级
      GetTop ( OPTR, op );  //读一个操作符
```

```
        if ( icp(ch) > isp(op) )  //栈顶优先级低
        { Push ( OPTR, ch );  getchar ( ch ); }
        else if ( icp(ch) < isp(op) ) {
            Pop ( OPTR, op );  //栈顶优先级高
            DoOperator ( OPND, op );
         }
         else if ( ch == ')' )
         //栈顶优先级等于扫描操作符优先级
         { Pop ( OPTR, op );  getchar ( ch ); }
       }
       GetTop ( OPTR, op );
    }  // end of while
}
```

3.2 栈与递归

- **递归的定义** 若一个对象部分地包含它自己，或用它自己给自己定义，则称这个对象是递归的；若一个过程**直接地或间接地调用自己**，则称这个过程是递归的过程。
- 以下三种情况常常用到递归方法：
  - ✓ 定义是递归的
  - ✓ 数据结构是递归的
  - ✓ 问题的解法是递归的

定义是递归的

例如，阶乘函数

$$n! = \begin{cases} 1, & \text{当 } n = 0 \text{ 时} \\ n * (n-1)!, & \text{当 } n \geq 1 \text{ 时} \end{cases}$$

求解阶乘函数的递归算法

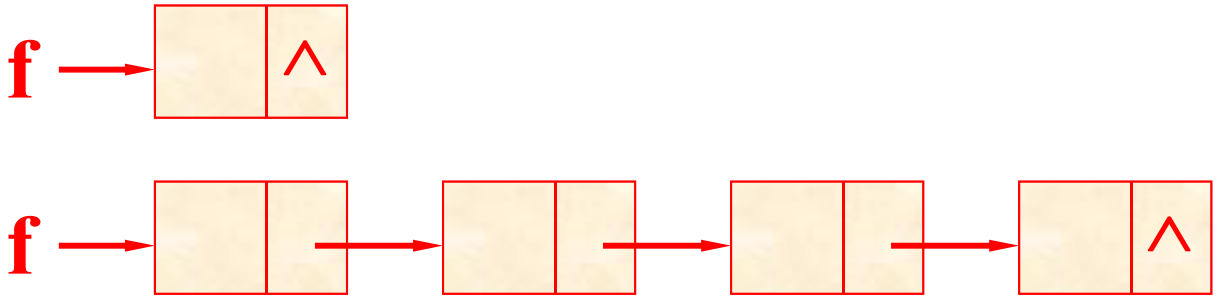
```
long Factorial ( long n ) {
    if ( n == 0 ) return 1;
    else return n * Factorial ( n-1 );
}
```

求解阶乘 $n!$ 的过程


主程序 main : fact(4)
参数 4 计算 4*fact(3) 返回 24
参数 3 计算 3*fact(2) 返回 6
参数 2 计算 2*fact(1) 返回 2
参数 1 计算 1*fact(0) 返回 1
参数 0 直接定值 = 1 返回 1
递归调用
参数传递
结果返回
回归求值

数据结构是递归的

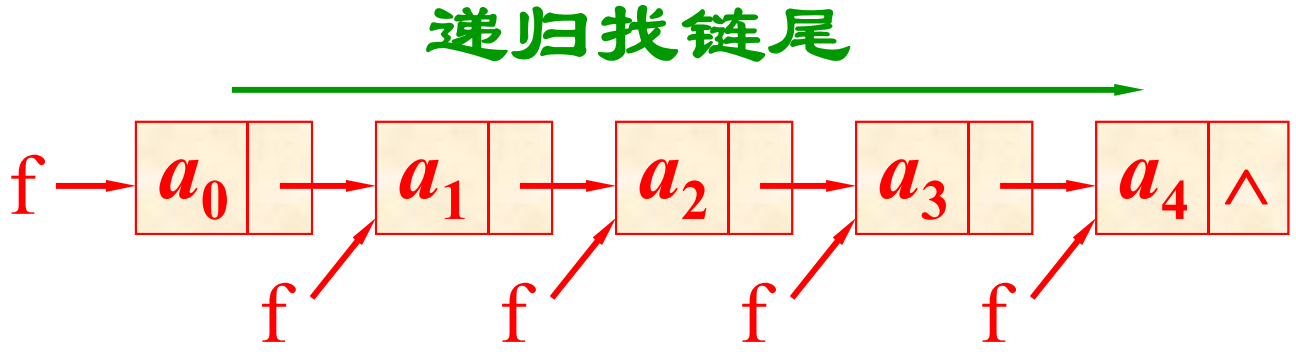
例如，单链表结构



- 一个结点，它的指针域为NULL，是一个单链表；
- 一个结点，它的指针域指向单链表，仍是一个单链表。

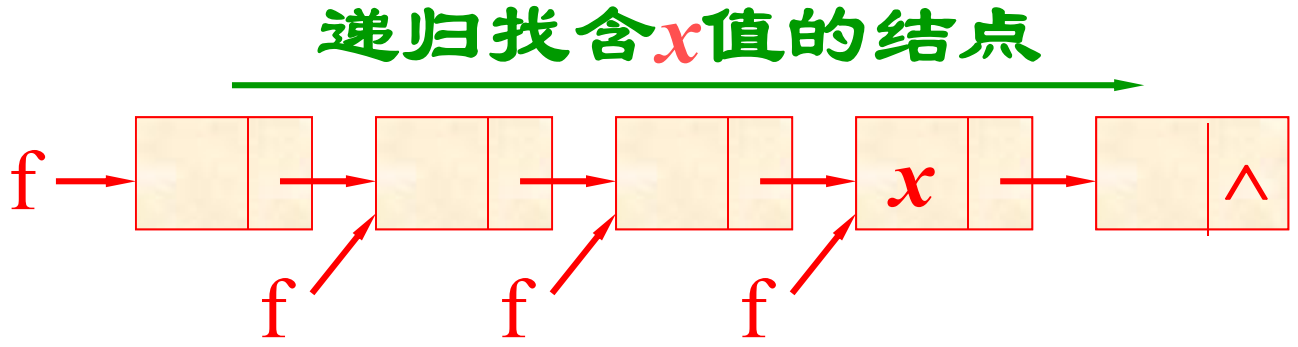
搜索链表最后一个结点并打印其数值

```
template <class Type>
void Print ( ListNode <Type> *f ) {
    if ( f->link == NULL )
        cout << f->data << endl;
    else Print ( f->link );
}
```



在链表中寻找等于给定值的结点并打印其数值

```
template <class Type>
void Print ( ListNode <Type> *f, Type &x ) {
    if ( f != NULL )
        if ( f->data == x )
            cout << f->data << endl;
        else Print ( f->link, x );
}
```



问题的解法是递归的

例如，汉诺塔(Tower of Hanoi)问题的解法

如果 n = 1，则将这一个盘子直接从 A 柱移到 C 柱上。否则，执行以下三步：

① 用 C 柱做过渡，将 A 柱上的 (n-1) 个盘子移到 B 柱上；

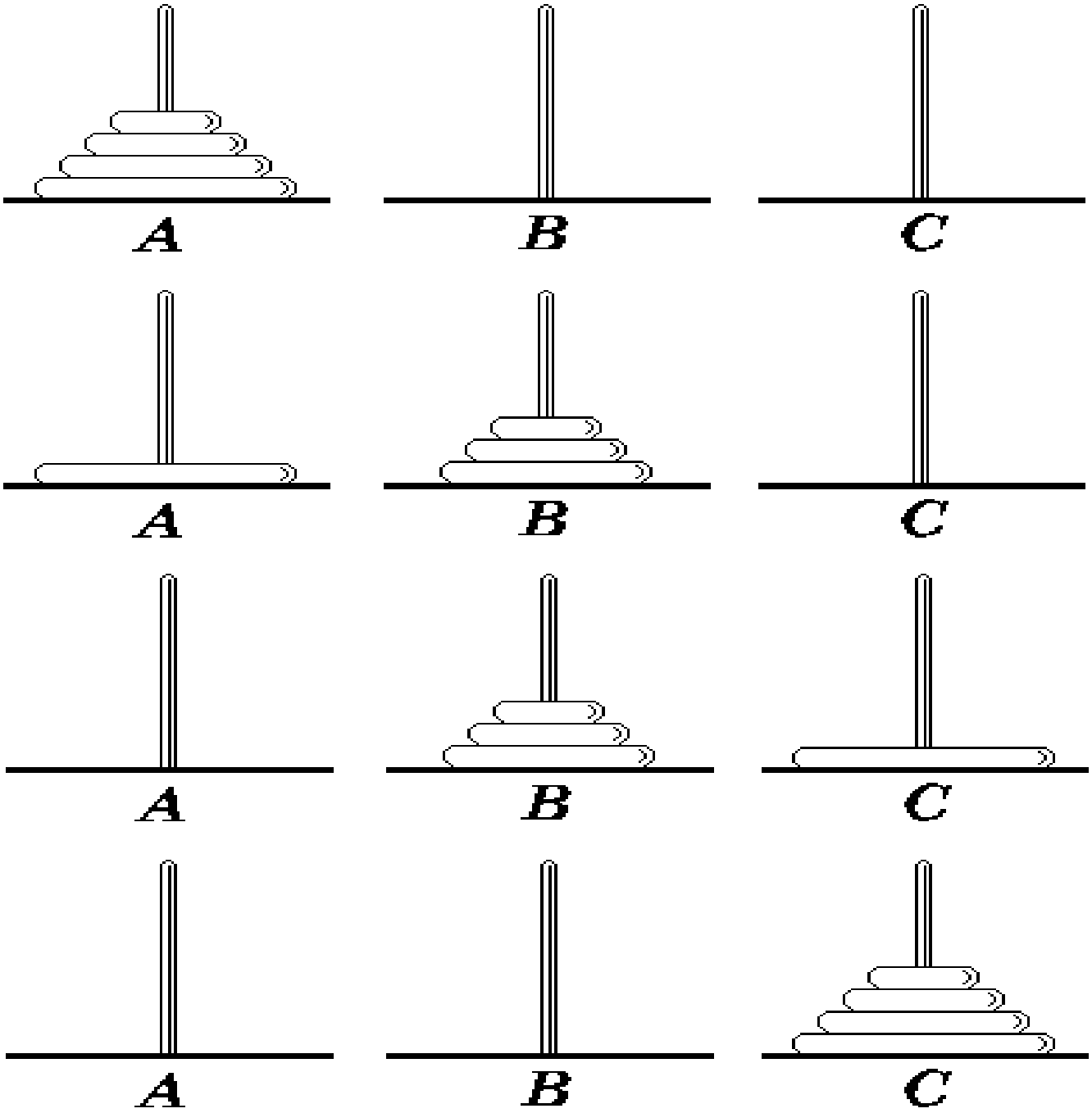
② 将 A 柱上最后一个盘子直接移到 C 柱上；

③ 用 A 柱做过渡，将 B 柱上的 (n-1) 个盘子移到 C 柱上。

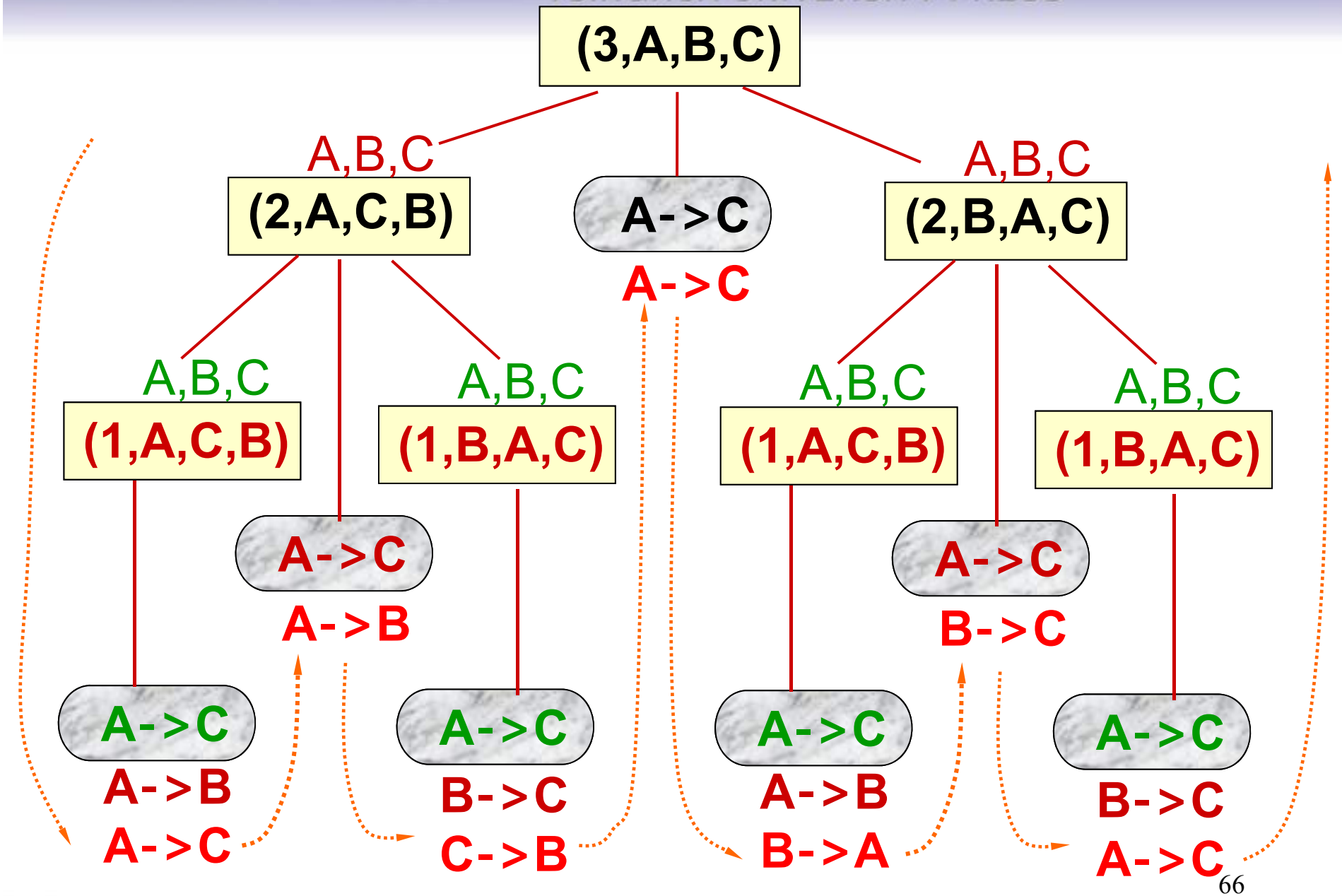
问题描述——*A*柱上每一个盘子都比下面略小一点，把*A*柱上的盘子全部移到*C*柱上。

移动条件——①一次只能移动一个盘子，

②移动过程中，大盘不能放在小盘上面。


A
B
C
A
B
C
A
B
C
A
B
C

```
#include <iostream.h>
#include "strclass.h"
void Hanoi ( int n, String A, String B, String C ) {
 //解决汉诺塔问题的算法
   if ( n == 1 )
      cout << " move " << A << " to  " << C << endl;
   else {
            Hanoi ( n-1, A, C, B );
            cout << " move " << A << " to " << C << endl;
            Hanoi ( n-1, B, A, C );
         }
}
```


(3,A,B,C)
A,B,C
(2,A,C,B)
A->C
A->C
A,B,C
(2,B,A,C)
A,B,C
(1,A,C,B)
A,B,C
(1,B,A,C)
A,B,C
(1,A,C,B)
A,B,C
(1,B,A,C)
A->C
A->B
A->C
B->C
A->C
A->B
A->C
B->C
A->C
A->B
A->C
B->C
C->B
B->A
A->C

自顶向下、逐步分解的策略

- 子问题应与原问题做同样的事情，且更为简单。
- 解决递归问题的策略是把一个规模比较大的问题分解为一个或若干规模比较小的问题，分别对这些比较小的问题求解，再综合它们的结果，从而得到原问题的解。
  — 分而治之策略（分治法）
- 这些比较小的问题的求解方法与原来问题的求解方法一样。

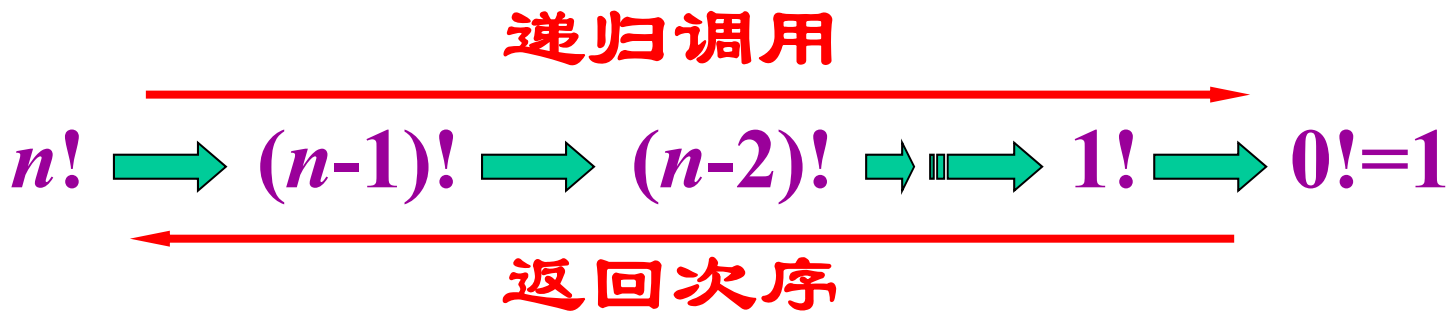
构成递归的条件

- 不能无限制地调用本身，必须有一个出口，化简为非递归状况直接处理。

```
Procedure <name> ( <parameter list> )
{
  if ( < initial condition> )
    return ( initial value );
  else
    return ( <name> ( parameter exchange ) );
}
```

递归过程与递归工作栈

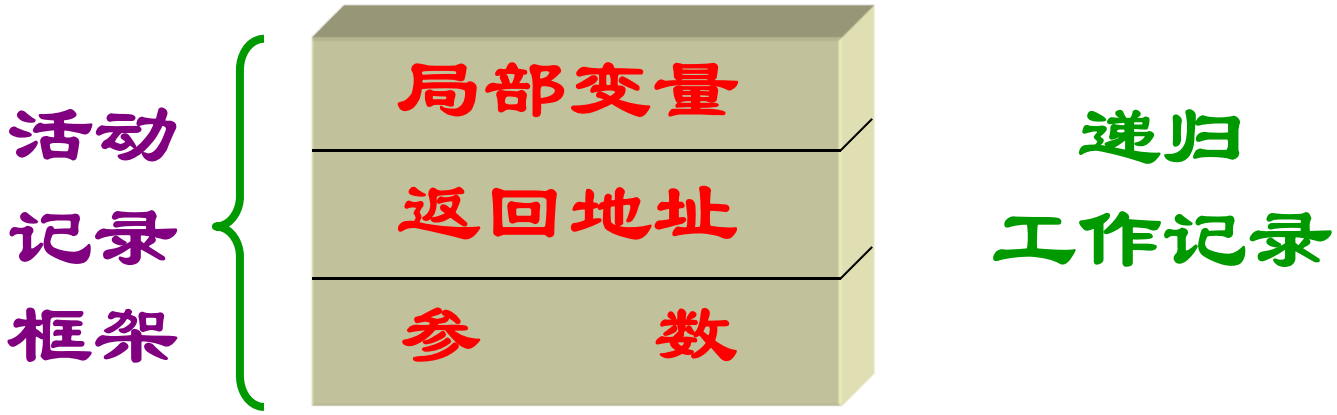
- 递归过程在实现时，需要自己调用自己。
- 层层向下递归，退出时的次序正好相反。



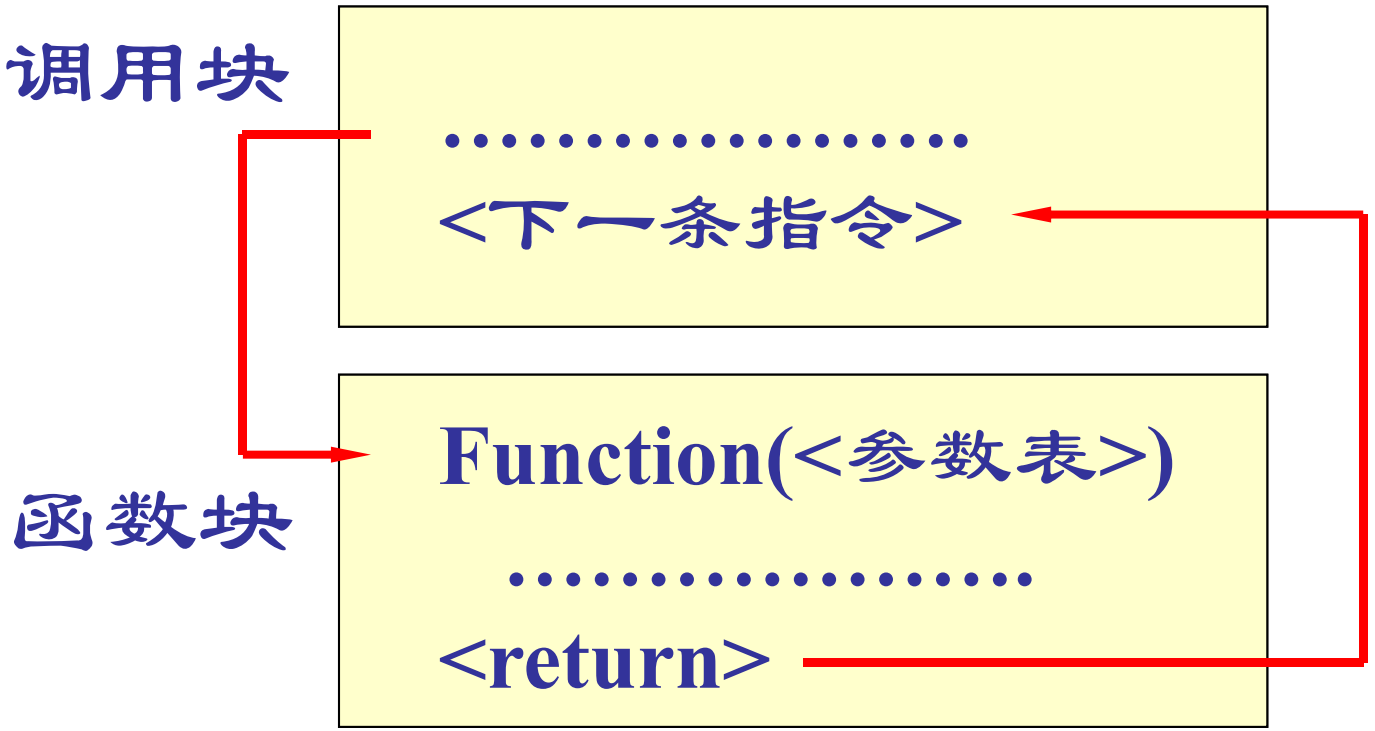
- 主程序第一次调用递归过程为外部调用；
- 递归过程每次递归调用自己为内部调用。
- 它们返回调用它的过程的地址不同。

递归工作栈

- 每一次递归调用时，需要为过程中使用的参数、局部变量等另外分配存储空间。
- 每层递归调用需分配的空间形成递归工作记录，按后进先出的栈组织。



函数递归时的活动记录



```
long Factorial ( long n ) {
    int temp;
    if ( n == 0 ) return 1;
    else temp = n * Factorial ( n-1 );
RetLoc2 ──────────────────────┐
    return temp;
}

void main ( ) {
    int n;
    n = Factorial (4);
RetLoc1 ──────┐
}
```

计算Fact时活动记录的内容

随堂练习

例1：按以下计算规则求已知两个整数的最大公约数。

$$gcd(a, b) = \begin{cases} a, & a\%b == 0 \\ b, & b\%a == 0 \\ gcd(a\%b, b), & a > b \\ gcd(a, b\%a), & a < b \end{cases}$$

例2：用递归函数实现已知十进制整数返回十进制的反向的整数。如已知十进制数1234，返回反向的十进制数是4321。

例1：按以下计算规则求已知两个整数的最大公约数。

```
//函数以两个已知整数为形参，返回整型结果。
int gcd int a, int b)
{
        if(a % b == 0) return b;
        if(b % a == 0) return a;
    if(a > b) return gca(a%b, b);
    if(a < b) return gcd(a, b % a);
}
```

例2：用递归函数实现已知十进制整数返回十进制的反向的整数。

一个数的反向过程是一个反复将数的个位作为对应的高位的转换过程：设一个数已被译出了一部分，例如，对于1234，已译出部分高位43，还有12还未被转换。问题是如何将12的个位数2转移到43，使新的高位变成432呢？只要将43乘10后加上2即可。若用循环实现有以下函数：

```
int reverInt(int n)
{   int s = 0;
    while(n) {
       s = s*10 + n%10;
       n /= 10;
      }
         return s;
}
```

如果将上述计算改写成递归，需要将循环计算中高位部分的值作为参数。初始调用时，对应s为0值。在被转换的数是个位数情况下，函数的返回值是s*10+n。如果n是多位数，则要继续转换的数变成n/10，新的高位的部分值是s*10+n%10。

```
int reverIntRe(int n, int s)
{
            if(n<10) return s*10+n;
            return reverIntRe(n/10, s*10 + n % 10);
}
```

递归过程改为非递归过程

- 递归过程简洁、易编、易懂；
- 递归过程效率低，重复计算多；
- 改为非递归过程的目的是提高效率；
- 单向递归和尾递归可直接用迭代实现其非递归过程；
- 其它情形必须借助栈实现非递归过程。

单向递归

- 比线性递归更复杂一些的是单向递归(Single Direction Recursive)。尾递归可以看作一种特殊的单向递归。
  - 普通单向递归可以有多个递归调用，但这些调用都必须处于递归调用过程的最后。
    - 单向递归同时要求递归过程中的这些递归调用的参数由主调用过程的参数决定，而处在同一递归层次的递归调用之间参数不会互相影响。
  - 单向递归同样可以较为方便地转化为循环实现。
    - 由于单向递归中的递归调用处于递归过程的最后，因此同样可以从递归的终止条件开始，通过循环逐渐构造出整个问题的解，每一次循环完成的工作相当于从一个递归层次中返回。
    - 用变量保存中间结果，不需要栈。
  - 以下的例子就是求解斐波那契数列的递归算法和非递归算法。

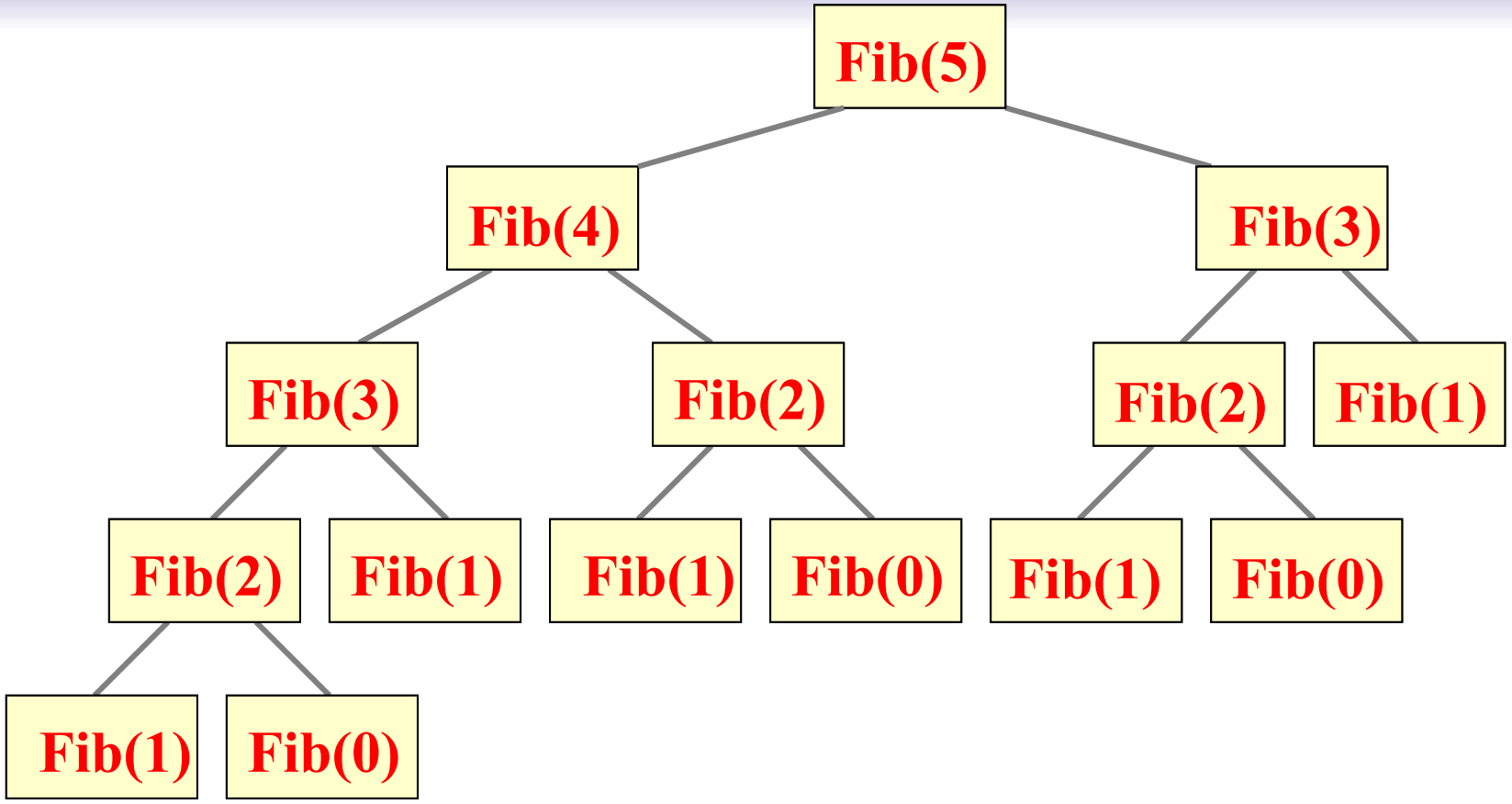
计算斐波那契数列的函数Fib(n)的定义

$$\mathrm{Fib}(n)=\begin{cases} n, & n=0,1 \\ \mathrm{Fib}(n-1)+\mathrm{Fib}(n-2), & n>1 \end{cases}$$

如 $F_0=0, F_1=1, F_2=1, F_3=2, F_4=3, F_5=5$

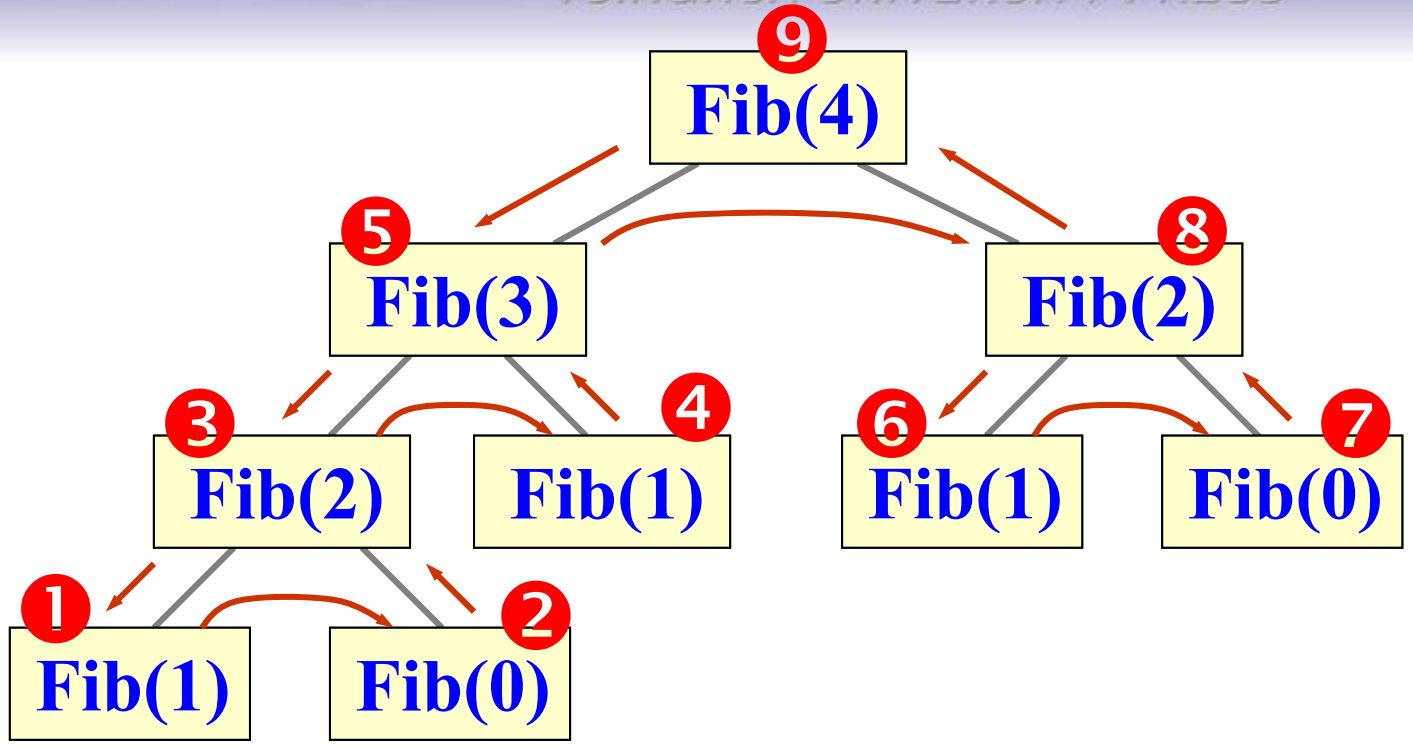
求解斐波那契数列的递归算法

```
long Fib ( long n ) {
    if ( n <= 1 ) return n;
    else return Fib ( n-1 ) + Fib ( n-2 );
}
```

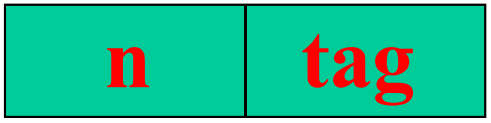


斐波那契数列的递归调用树

调用次数 NumCall(k) = 2*Fib(k+1) - 1



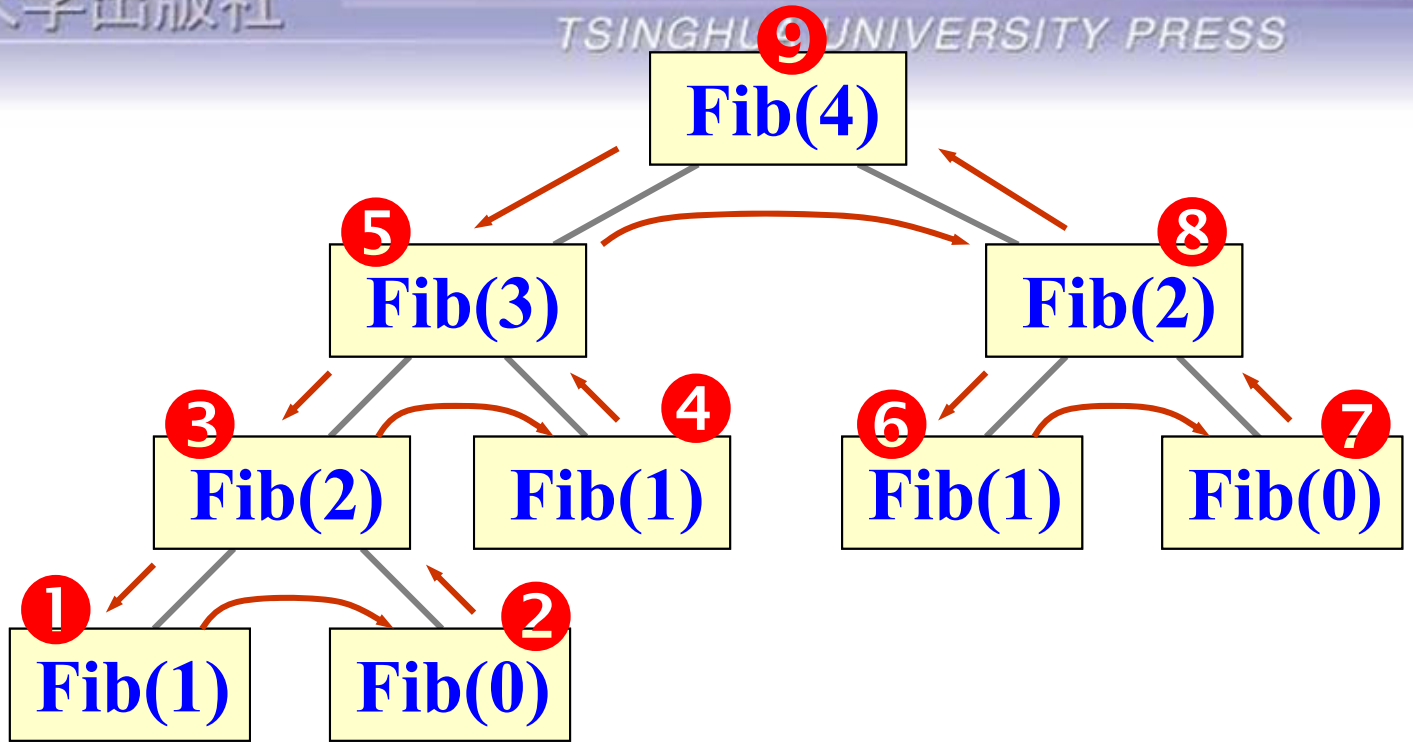
栈结点

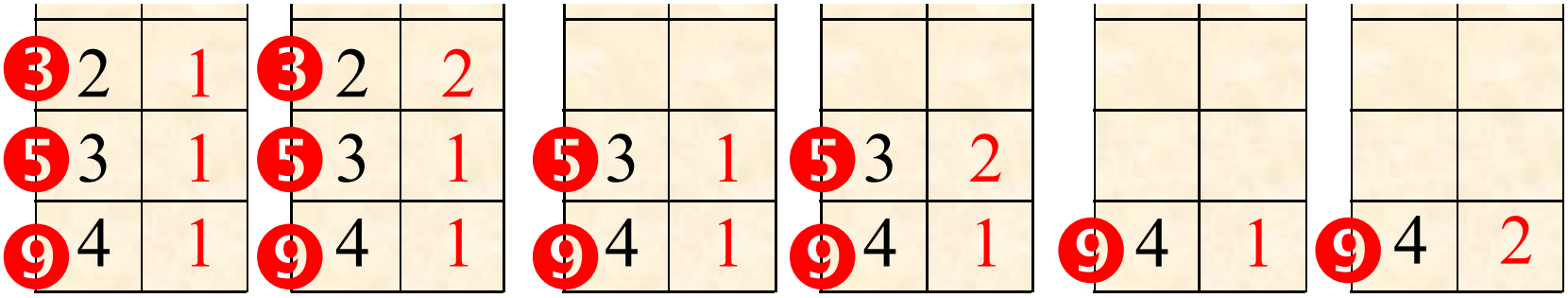


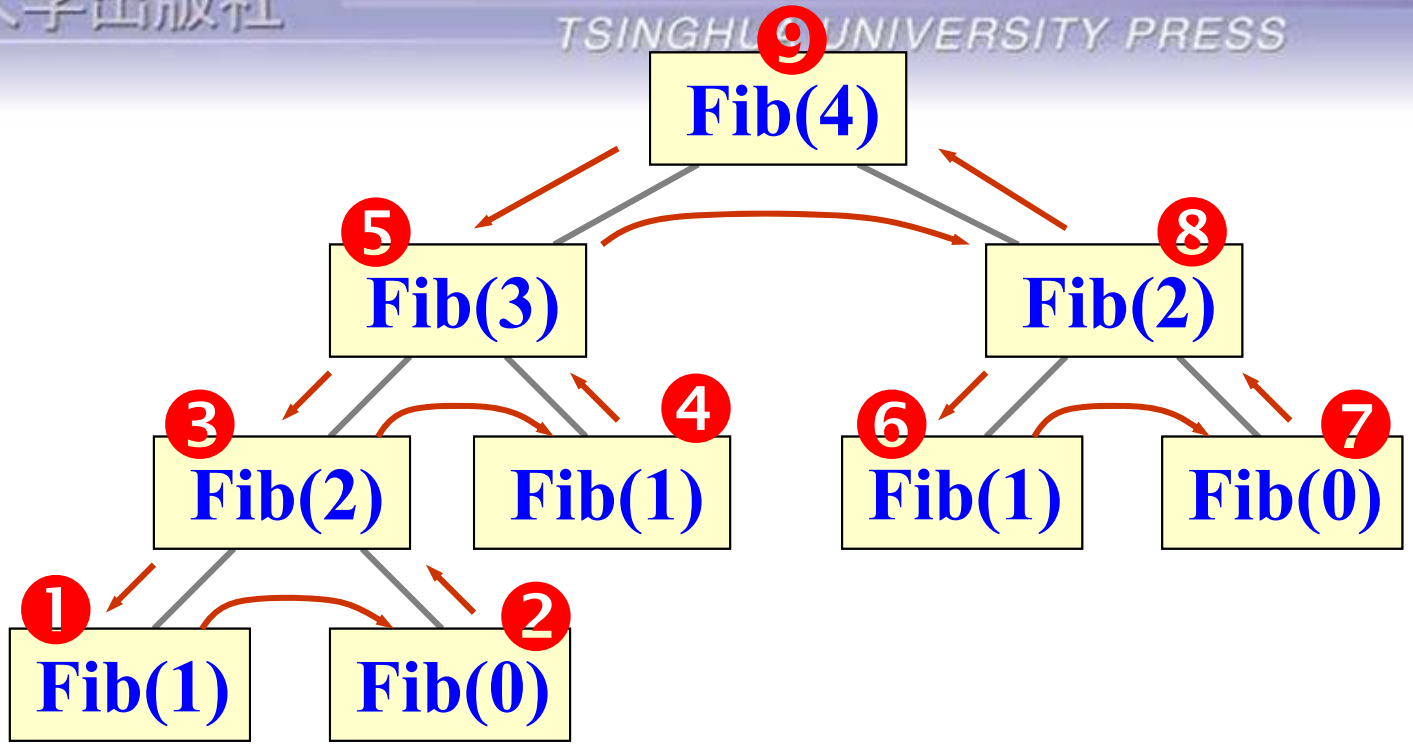

```
struct Node {
    long n;
    int tag;
};
```

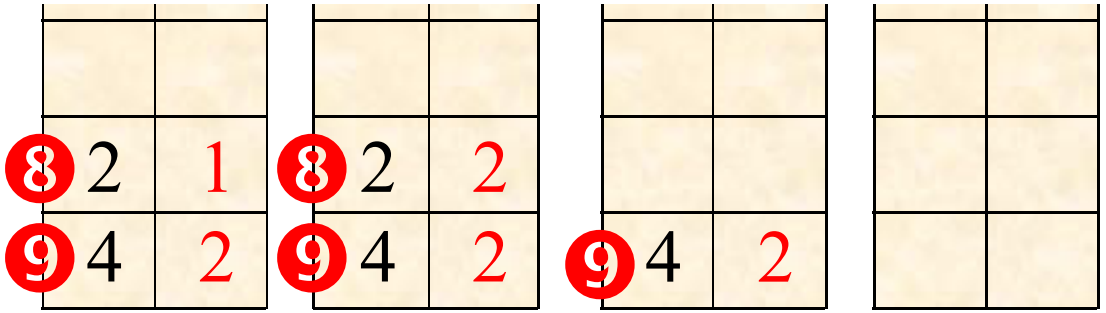
tag = 1，向左递归；

tag = 2，向右递归；


9
Fib(4)
5
Fib(3)
8
Fib(2)
3
Fib(2)
4
Fib(1)
6
Fib(1)
7
Fib(0)
1
Fib(1)
2
Fib(0)


3 2 1
5 3 1
9 4 1
1 n=1
sum=0+1
3 2 2
5 3 1
9 4 1
2 n=2-2
5 3 1
9 4 1
2 n=0
sum=1+0
5 3 2
9 4 1
4 n=3-2
9 4 1
4 n=1
sum=1+1
9 4 2
8 n=4-2





6 n=1 sum=2+1

7 n=2-2

7 n=0 sum=3+0

```
long Fibnacci ( long n ) {
    Stack <Node> S;   Node w;  long sum = 0;
    //反复执行直到所有终端结点数据累加完
    do {
      while ( n > 1 ) {
         w.n = n;  w.tag = 1;  S.Push (w);
         n--;
     } //向左递归到底，边走边进栈
     sum = sum + n;  //执行求和
```

```
        while ( S.IsEmpty ( ) == false ) {
            w = S.GetTop ( );  S.Pop ( );
            if ( w.tag == 1 ) {  //改为向右递归
              w.tag = 2;  S.Push ( w );
              n = w.n – 2;  // F(n) 右侧为 F(n-2)
              break;
            }
        }
    } while ( !S.IsEmpty ( ) );
    return sum;
}
```

单向递归用迭代法实现

```
long FibIter ( long n ) {
    if ( n <= 1 ) return n;
    long twoback = 0,  oneback = 1,  Current;
    for ( int i = 2; i <= n; i++ ) {
      Current = twoback + oneback;
      twoback = oneback;   oneback = Current;
    }
    return Current;
}
```

尾递归用迭代法实现

- 尾递归
  - 递归调用语句只有一个，放在过程最后。
  - 递归调用返回时，返回到上一层递归调用语句的下一语句——程序末尾。

25	36	72	18	99	49	54	63

```
void recfunc ( int A[ ],  int n ) {
    if ( n >= 0 ) {
        cout << A[n] << “  ”;
        n--;
        recfunc ( A, n );
    }
}
void sterfunc ( int A[ ], int n ) {
//消除了尾递归的非递归函数
    while ( n >= 0 ) {
        cout << “value    ” << A[n] << endl;
        n--;    }
}
```

- 比尾递归算法复杂一些的是线性递归(Linear Recursive)算法，线性递归算法有以下所示的一般形式：

```
proc(int n) {
    Stms1(n);
    if (Eval(n)) {
      proc(n-1); //递归调用
      Stms2(n);
    }
    else
      Stms3(n);
  }
```

- 其中，Stms1(*n*)、Eval(*n*)、Stms2(*n*)和Stms3(*n*)是可能与*n*有关的非递归计算的代码段，也可能为空。

- 线性递归能机械地被转换成一个功能等价的非递归描述：

```
proc(int n) {
    int k=n+1;
    do {
        k--;
        Stms1(k);
    } while Eval(k);
    Stms3(k);
    while (k<n) {
        k++;
        Stms2(k);
    }
}
```

- 例如对于求阶乘的问题，如果要考虑得更完善，需要防止在计算过程中发生溢出等异常情况。为此，按照线性递归的一般形式把求阶乘的程序改写为：

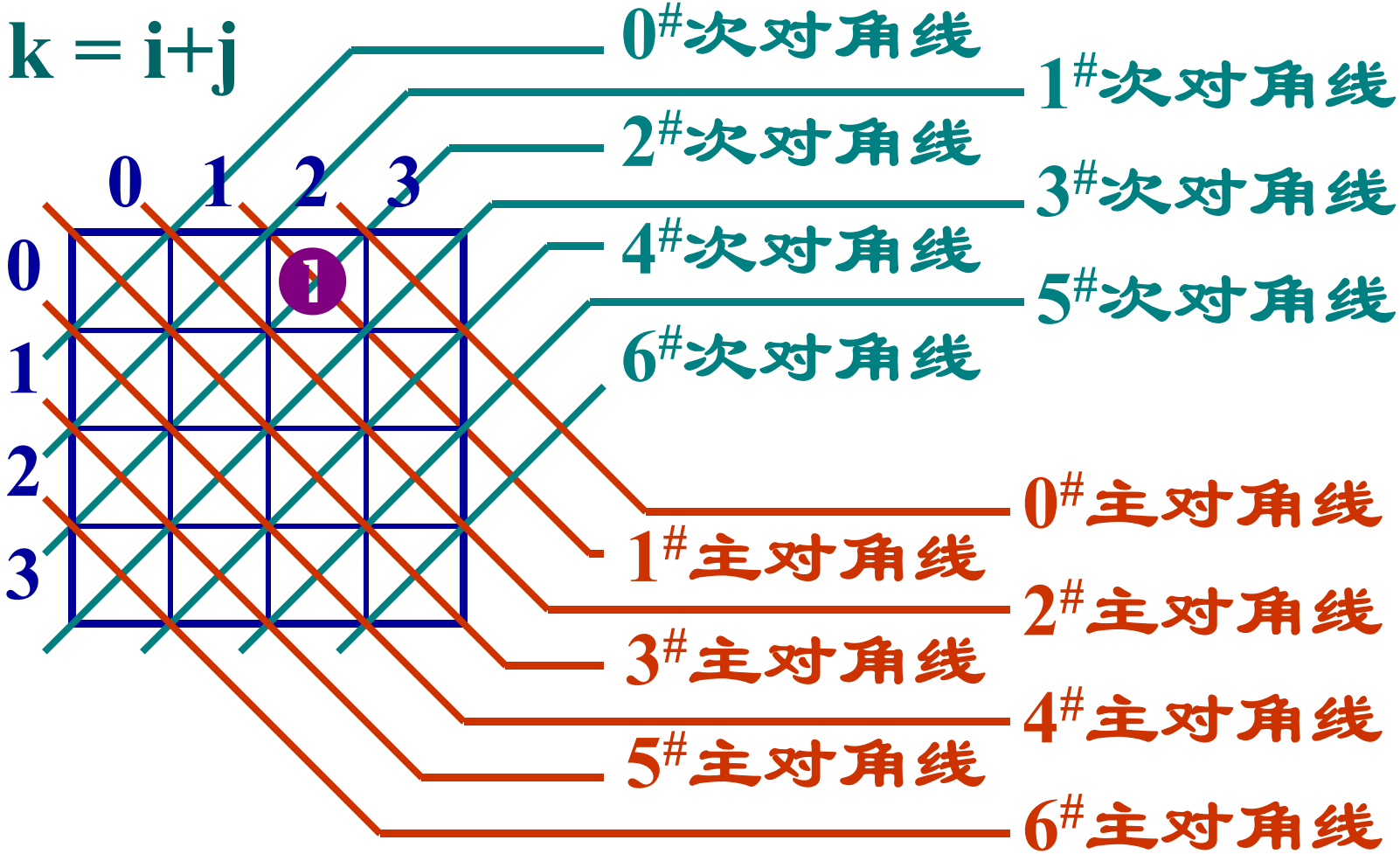
```
void factorial(int n, int &temp) {
    if (n<0) { cerr<<n<<"不能为负数"<<endl;  exit(1); }
    if (n>0) {
        factorial(n-1, temp);
        if (Max/temp>n) { temp=temp*n; }
        //Max为允许的最大整数
       else { cerr<<"计算结果溢出"<<endl;  exit(1); }
    }
    else temp=1;
}
```

- 这个程序可以套用线性递归转非递归的模式转换为：

```
void factorial(int n, int &temp) {
    int k=n+1;
    do {
        k--;
        if (k<0) { cerr<<k<<"不能为负数"<<endl;  exit(1); }
    } while (k>0);
    temp=1;
    while (k<n) {
      k++;
      if (Max/temp>k)  temp=temp*k;
      else { cerr<<"计算结果溢出"<<endl;  exit(1); }
    }
}
```

n皇后问题

在 n 行 n 列的国际象棋棋盘上，若两个皇后位于同一行、同一列、同一对角线上，则称为它们为互相攻击。n皇后问题是指找到这 n 个皇后的互不攻击的布局。


k = i+j
0 1 2 3
0
1
2
3
1
0#次对角线
1#次对角线
2#次对角线
3#次对角线
4#次对角线
5#次对角线
6#次对角线
0#主对角线
1#主对角线
2#主对角线
3#主对角线
4#主对角线
5#主对角线
6#主对角线
k = n+i-j-1

解题思路

- 安放第 i 行皇后时，需要在列的方向从 0 到 n-1 试探 (j = 0, …, n-1) 。
- 在第 j 列安放一个皇后：
  - 如果在列、主对角线、次对角线方向有其它皇后，则出现攻击，撤消在第 j 列安放的皇后。
  - 如果没有出现攻击，在第 j 列安放的皇后不动，递归安放第 i+1 行皇后。

- 设置 4 个数组
  - col[n]： col[i] 标识第 i 列是否安放了皇后；
  - md[2n-1]： md[k] 标识第 k 条主对角线是否安放了皇后；
  - sd[2n-1]： sd[k] 标识第 k 条次对角线是否安放了皇后；
  - q[n]： q[i] 记录第 i 行皇后在第几列。

```
void Queen( int i ) {
    for ( int j = 0;  j < n;  j++ ) {
        if ( 第 i 行第 j 列没有攻击 ) {
            在第 i 行第 j 列安放皇后;
            if ( i == n-1 ) 输出一个布局;
            else Queen ( i+1 );
            撤消第 i 行第 j 列的皇后;
        }
    }
}
```

算法求精

```
void Queen( int i ) {
    for ( int j = 0;  j < n;  j++ ) {
      if ( !col[j] && !md[n+i-j-1] && !sd[i+j] )
      {
          //第 i 行第 j 列没有攻击
          col[j] = md[n+i-j-1] = sd[i+j] = 1;
          q[i] = j;
          //在第 i 行第 j 列安放皇后
```

```
        if ( i == n-1 ) {  //输出一个布局
          for ( k = 0; k < n; k++ )
              cout << q[k] << ',';
          cout << endl;
        }
        else  Queen ( i+1 );
        col[j] = md[n+i-j-1] = sd[i+j] = 0;
        q[i] = 0;   //撤消第 i 行第 j 列的皇后
      }
    }
}
```

随堂练习

例1：已知Ackerman函数定义如下：

$$akm(m,n)=\begin{cases} n+1, & m=0 \\ akm(m-1,1), & m\neq 0, n=0 \\ akm(m-1,akm(m,n-1)), & m\neq 0, n\neq 0 \end{cases}$$

(1) 根据定义，写出它的递归求解算法；

(2) 利用栈，写出它的非递归求解算法。

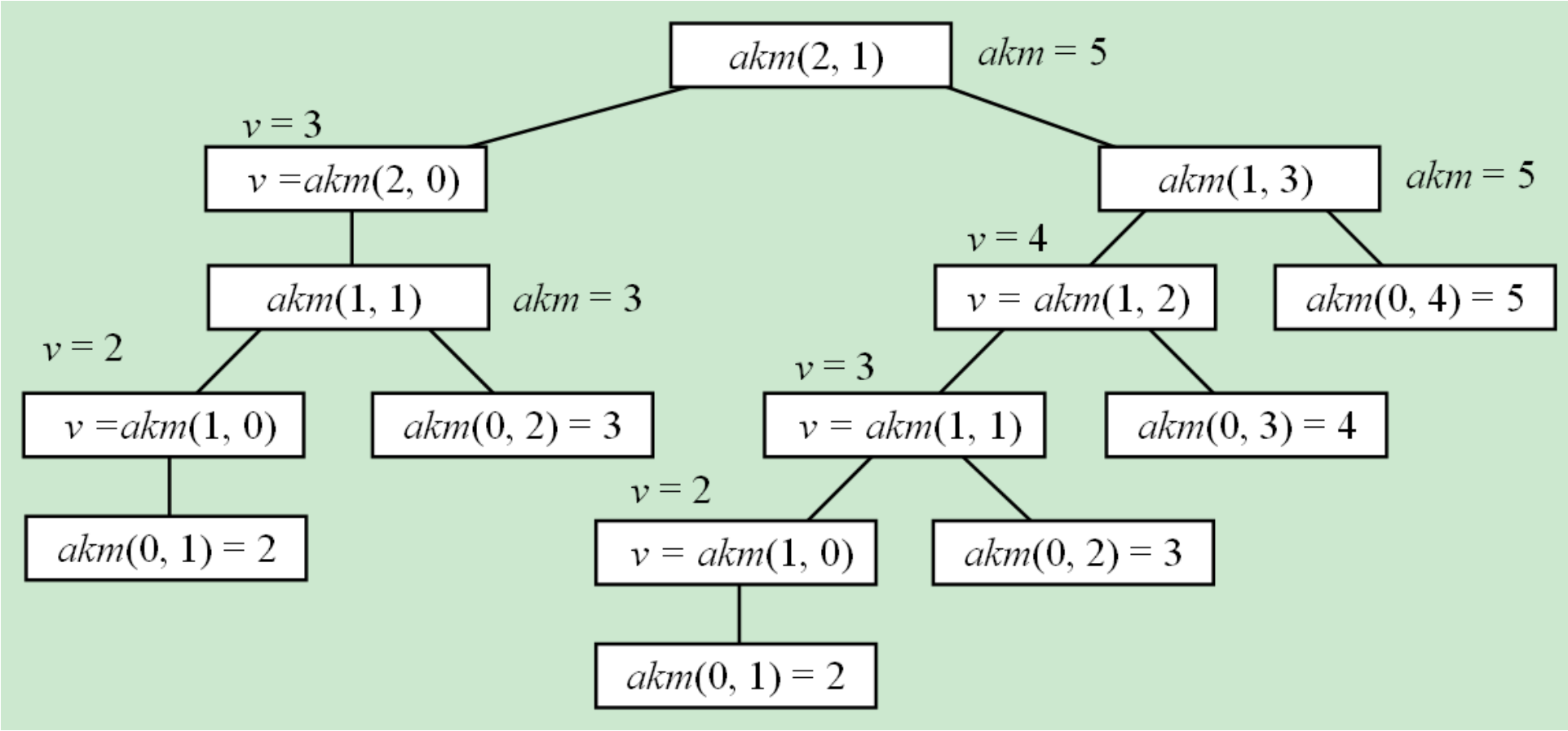
函数本身是递归定义的，所以可以用递归算法来解决：

```
unsigned akm ( unsigned m, unsigned n ) {
    if ( m == 0 ) return n+1;        // m == 0
    else if ( n == 0 ) return akm ( m-1, 1 );
        // m > 0, n == 0
        else return akm ( m-1, akm ( m, n-1 ) );
        // m > 0, n > 0
}
```

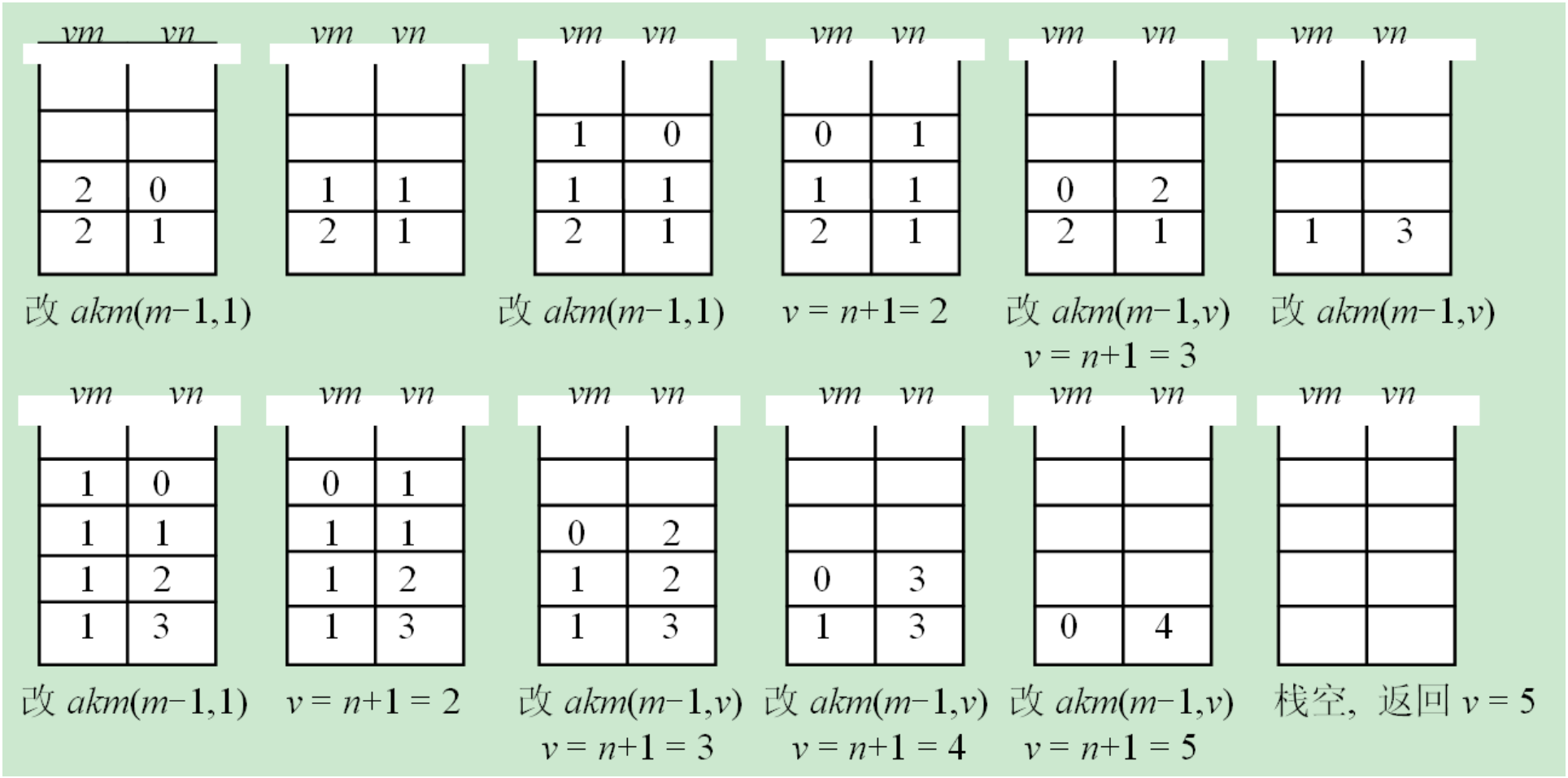
首先改写原来的递归算法，将递归语句从结构中独立出来：

```
unsigned akm ( unsigned m, unsigned n ) {
    unsigned v;
    if ( m == 0 ) return n+1;    // m == 0
    if ( n == 0 ) return akm ( m-1, 1 );
    // m > 0, n ==0
    v = akm ( m, n-1 );    // m > 0, n > 0
    return akm ( m-1, v );
}
```

akm(2, 1)的递归调用树



用栈记忆每次递归调用时的实参值，每个结点两个域{vm, vn}



```
unsigned akm ( unsigned m, unsigned n ) {
        struct node { unsigned vm, vn; }
        stack <node> st ( maxSize );  node w;  unsigned v;
        w.vm = m;  w.vn = n;  st.Push (w);
        do {
            while ( st.GetTop( ).vm > 0 ) {
             //计算akm(m-1, akm(m, n-1) )
                 while ( st.GetTop( ).vn > 0 )
                 //计算akm(m, n-1), 直到akm(m, 0)
                 { w.vn--;  st.Push( w ); }
                 w = st.GetTop( );  st.Pop( );
                 //计算akm(m-1, 1)
```

```
            w.vm--;  w.vn = 1;  st.Push( w );      }
            //直到akm( 0, akm( 1, * ) )
            w = st.GetTop();  st.Pop( );  v = w.vn+1;
            //计算v = akm( 1, * )+1
            if ( st.IsEmpty( ) == 0 )
            //如果栈不空, 改栈顶为( m-1, v )
            { w = st.GetTop();  st.Pop( );
            w.vm--;  w.vn = v;  st.Push( w ); }
        } while ( st.IsEmpty( ) == 0 );
        return v;
}
```

关于递归转非递归

- 理论上而言，所有递归程序都可以用非递归程序来实现；这种理论的基础是递归程序的计算总能用一颗树型结构来表示。
- 递归计算从求树根结点的值开始，树根结点的值依赖一个或多个子结点的值，子结点的值又依赖下一级子结点的值，如此直至树的叶子结点。叶子结点的值能直接计算出来，也就是递归程序的出口。

递归转非递归的两种方法

1、可用迭代的算法替代(单向递归、尾递归)。

2、自己用堆栈模拟系统的运行时的栈，通过分析只保存必须保存的信息，从而用非递归算法替代递归算法。

- 画递归调用树
- 分析出栈过程
- 根据出栈过程遍历这颗树的叶子结点

例2：汉诺塔问题的非递归解法。

分析递归解法可知：盘子移动的输出是在两种情况下进行的：其一是当递归到n=1时进行输出；其二是返回到上一层函数调用具有n>=1时进行输出。因此，可以使用一个stack[]数组来实现汉诺塔问题的非递归解法。具体步骤如下：

(1) 将初值n, x, y, z送数组stack[]；

(2) n减1并交换y, z值后将n, x, y, z值送数组stack[]，直到n=1时为止；

(3) 当n=1时输出x值指向z值；

(4) 回退到前一个数组元素，如果此时这个数组元素下标值大于等于1，则输出x值指向z值（相当于递归函数hanoi返回到上一层函数调用时的输出）；当i<=0时程序结束。

(5) 将n减1，如果n>=1则交换x和y值（相当于执行递归函数hanoi中的第二个hanoi调用语句参数替换时的情况）；并且：

A. 如果此时n=1，则输出x值指向z值并继续退回到前一个数组元素处（相当于返回到递归函数hanoi的上一层）；如果i>=1，则输出x值指向z值；

B. 如果此时n>1，则转(2)处继续执行（相当于递归函数hanoi中的第二个hanoi调用语句继续调用的情况）。

递归与回溯 常用于搜索过程

- 对一个包含有许多结点，且每个结点有多个分支的问题，可先选择一个分支进行搜索。当搜索到某一结点，发现无法再继续搜索下去时，可沿搜索路径回退到前一结点，沿另一分支继续搜索。
- 如果回退之后无其它选择，再沿搜索路径回退到更前结点，…。依此执行，直到搜索到问题的解，或搜索完全部可搜索分支没有解存在为止。
- 回溯法与分治法本质相同，可用递归算法求解。

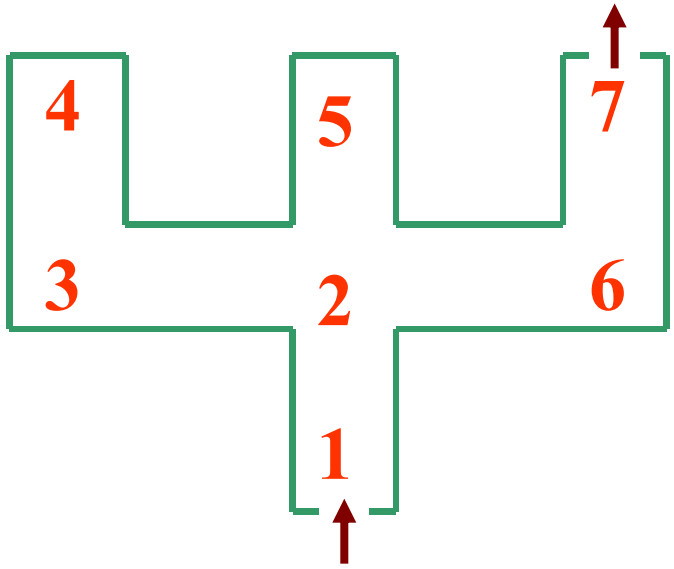
迷宫问题

- 迷宫中设置很多墙壁，对前进方向形成多处障碍。如果从迷宫入口到达出口，途中不出现行进方向错误，则得到一条最佳路线。

  ——递归方法

- 回溯
  - 沿某条路径逐步走向出口，一但发现走入死胡同走不通时，回溯一步或多步，寻找其它可走路径。

- 一系列交通路口的集合
  - 每个路口有3个相关联的参数
    - 直走、左拐和右拐
      - 参数值为0——该方向路堵死
      - 参数值不为0——该方向有路可通
  - 一个路口的3个参数都为0
    - 在迷宫中走入绝境
  - 到达出口
    - 成功通过迷宫

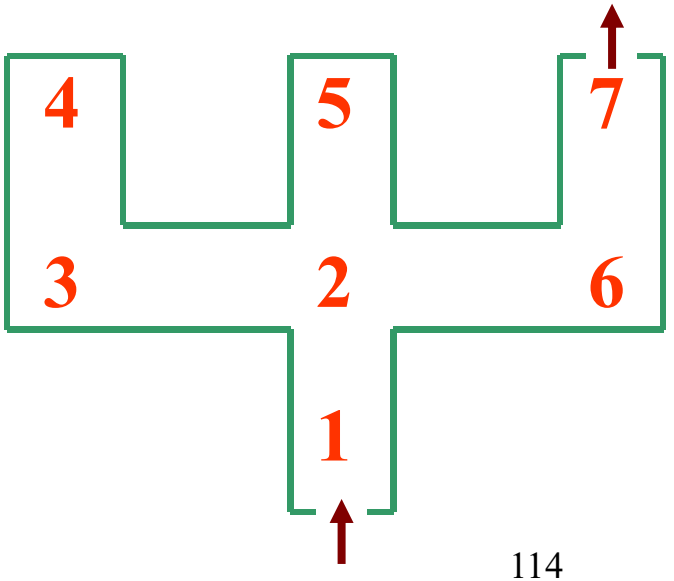


小型迷宫

- 在每个交通路口上
  - 如果可能，先向左走，当到达下一路口时，仍试探向左走。
  - 如果向左走不通，原路返回向前走。
  - 如果仍然走不通，再原路返回向右走。
  - 如果三个方向都遇到死路，需要退回上一个路口，重新选择一个新方向。
- 上述搜索策略单调冗长，效率不高，但能确保最终可以找到一条走出迷宫的路。

路口	动作	结果
1 (入口)	正向走	进到 2
2	左拐弯	进到 3
3	右拐弯	进到 4
4 (堵死)	回溯	退到 3
3 (堵死)	回溯	退到 2
2	正向走	进到 5
5 (堵死)	回溯	退到 2
2	右拐弯	进到 6
6	左拐弯	进到 7 (出口)

最后走到出口处，到达一条通路的终点，才能确定所有位于该路径上的交通路口，重描所走过的路。



- **迷宫数据由三种信息组成**
  - **迷宫交通路口数目 *MazeSize*；**
  - **迷宫出口 *EXIT*；**
  - **迷宫路口数组 **intsec***
    - **路口数据的结构为 *Intersection*， 每一路口的状况由 3 个参数描述。**

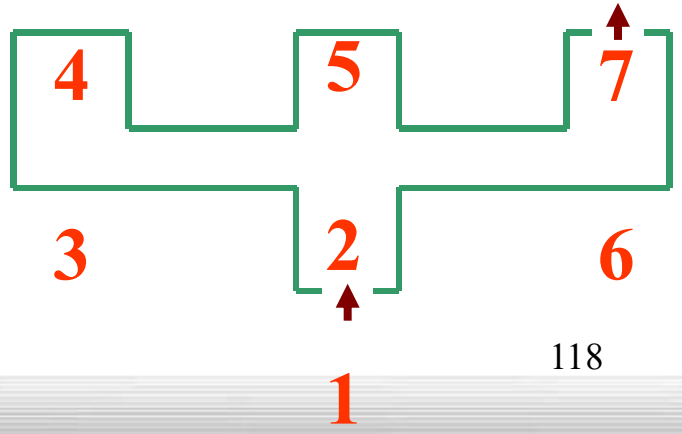
交通路口结构定义

```
struct Intersection {
//从当前路口出发，按左(left)、中(forward)、
//右(right)三个方向访问时所用到的结构
    int left; //左拐，堵死为 0
    int forward; //直走，堵死为 0
    int right; //右拐，堵死为 0
};
```

- 所有要访问的数据由迷宫构造函数在建立迷宫时提供。
- 迷宫的数据从文件读入：
  - 该文件包含三项信息：
    - 迷宫的交通路口数目
    - 每个路口的 3 个方向出口
    - 出口（退出迷宫的路口）值
      - 不再是路口信息，只是路口号码。

小型迷宫的数据

```
6                                   //迷宫路口数目
左 0  直 2  右 0                    //1: forward正向走到2
行 3  行 5  行 6                    //2: left左拐走到3
                                        forward正向走到5
                                        right右拐走到6
   0     0     4                    //3: right右拐走到4
   0     0     0                    //4: 堵死
   0     0     0                    //5: 堵死
   7     0     0                    //6: 左拐走到7

7                                   //迷宫出口的路口号码
```



迷宫类定义

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
class Maze {
private:
    int MazeSize; //迷宫中的路口数目
    int EXIT; //迷宫出口号码
    Intersection *intsec; //迷宫路口数组
public:
    Maze ( char *filename );
    //构造函数，从文件 filename 中读数据
    int TraverseMaze ( int CurrentPos ); //遍历并求解迷宫
};
```

```
Maze :: Maze ( char *filename ) {
//构造函数
//从文件 filename 中读取各路口和出口的数据
//并为路口信息数组动态分配存储空间
   ifstream fin;
   fin.open ( filename, ios::in | ios::nocreate );
   //为输入打开文件，文件不存在则打开失败
   if ( !fin ) {
       cout << "迷宫数据文件" << filename
                    << "打不开" << endl;
       exit (1);
    }
    fin >> MazeSize;  //输入迷宫路口数
```

```
    intsec = new Intersection[MazeSize+1];
    //创建迷宫路口数组
    for ( int i = 1; i <= MazeSize; i++ )
    //输入迷宫路口数据
      fin >> intsec[i].left >> intsec[i].forward
          >> intsec[i].right;
    fin >> EXIT;
    //输入迷宫出口号码
    fin.close ( );
}
```

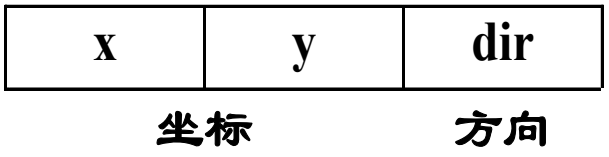
迷宫漫游与求解算法

```
int Maze::TraverseMaze ( int CurrentPos ) {
//递归函数：算法搜寻一个可能前进的方向
// CurrentPos 初值为 1，表明搜索者从路口 1 进入迷宫
//递归过程中，保持了当前处理的交通路口号码
  if ( CurrentPos > 0 ) { //路口为 0 表示到达路的尽头
  //否则试探找一个移动方向
    if ( CurrentPos == EXIT ) { //出口处理
       cout << CurrentPos << " "; return 1; }
    else //递归向左搜寻可行
    if ( TraverseMaze ( intsec[CurrentPos].left ) )
    { cout << CurrentPos << " "; return 1; }
    else //递归向前搜寻可行
    if ( TraverseMaze ( intsec[CurrentPos].forward ) )
    { cout << CurrentPos << " "; return 1; }
    else //递归向右搜寻可行
    if ( TraverseMaze ( intsec[CurrentPos].right ) )
    { cout << CurrentPos << " "; return 1; }
  } return 0;
}
```

迷宫问题的非递归解法

- 迷宫问题可用栈来存储试探过程中所走过的路径。
  - 一旦需要回退，可从栈中取得刚才走过位置的坐标和前进方向。
- 栈中数据可用如下三元组表示：

```
struct items {  //栈中的三元组结构
  int x, y;  //位置
  directions dir;  //前进方向
};
```



迷宫的表示

- **迷宫可用二维数组表示**

  int Maze[m+2][p+2];

- **迷宫数据的含义**

  Maze[i][j] 为 1 表示该位置是墙壁，不通;

  Maze[i][j] 为 0 表示该位置是通路;

  $1 \le i \le m, 1 \le j \le p$

  为便于程序处理，数组的第 0 行和第 m+1 行、第 0 列和第 p+1 列是迷宫的围墙。

用二维数组表示的迷宫

```
          1 1 1 1 1 1 1 1 1 1 1
入口      0 0 1 0 0 0 1 1 0 1
          1 1 0 0 0 1 1 0 1 1
          1 0 1 1 1 0 0 1 1 1
          1 0 1 0 1 1 1 0 0 0  出口
          1 1 1 1 1 1 1 1 1 1 1
```

在迷宫中任一时刻的位置可用数组下标 i 与 j 来表示。

可能的前进方向

NW（西北） [*i*-1][*j*-1]	*N*（北） [*i*-1][*j*]	*NE*（东北） [*i*-1][*j*+1]
W（西） [*i*][*j*-1]	*X* [*i*][*j*]	[*i*][*j*+1] *E*（东）
[*i*+1][*j*-1] *SW*（西南）	[*i*+1][*j*] *S*（南）	[*i*+1][*j*+1] *SE*（东南）

从 Maze[i][j] 出发，可能的前进方向有 8 个：

NW--Maze[i-1][j-1] N--Maze[i-1][j] NE--Maze[i-1][j+1]

W--Maze[i][j-1] E--Maze[i-1][j+1]

SW--Maze[i+1][j-1] S--Maze[i+1][j] SE--Maze[i+1][j+1]

- 设位置 [i][j] 标记为 X
  - 如果 X 周围的所有 8 个方向的位置都是 0 值，则可选择 8 个位置中的任一个位置作为下一位置。
  - 如果 X 周围的位置不都是 0 值，则只能在这几个相邻位置中选择下一位置。
- 前进方向表
  - 为有效地选择下一位置，则可将从位置 [i][j] 出发可能的前进方向定义在一个表内。
  - 给出向各个方向的偏移量。

前进方向表

q	*move[q].a*	*move[q].b*
N	-1	0
NE	-1	1
E	0	1
SE	1	1
S	1	0
SW	1	-1
W	0	-1
NW	-1	-1

当前位置在 [i][j] 时，
若向西南方向走，
下一相邻位置 [g][h] 为：

g = i+move[SW].a
  = i+1;
h = j+move[SW].b
  = j-1;

```
struct offsets { //位置是在直角坐标系下的偏移
  int a, b; // a 是 x 方向的偏移，b 是 y 方向的偏移 };

enum directions { N, NE, E, SE, S, SW, W, NW }; //全部 8 个方向
offsets move[8]; //各个方向的偏移表
```

- **在迷宫中进行搜索时，可能同时存在几个允许的前进方向。**
  - **利用三元组记下当前位置和上一步前进的方向，根据前进方向表，选择某一个允许的前进方向前进一步，并将活动记录进栈，以记下前进路径。**
  - **如果该前进方向走不通，则将位于栈顶的活动记录退栈，以在前进路径上回退一步，再尝试其它允许方向。**
  - **如果栈空，则表示已经回退到开始位置。**

标志矩阵*mark*防止走重复路

int mark[m+2][p+2]

其中所有元素都初始化为0。

- **标志矩阵的含义**

  mark[i][j] 为 1 表示该位置已走过;

  mark[i][j] 为 0 表示该位置尚未走过。

迷宫算法框架

```
//初始化工作栈、迷宫入口坐标和"E"方向首先进栈
//初始化 mark[m+2][p+2] 数组为 0，m 与 p 为迷宫行数和列数
// mark[i][j] == 0，表示该位置未走过
//每当一个位置 [i][j] 走到，mark[i][j] = 1（为防止走第二次）
while (栈非空) {
  (i, j, dir) 为从栈顶退出的坐标和方向;
  while (还有移动) {
    (g, h) 为下一移动的坐标;
    if ( g == m && h == p )
      迷宫搜索成功; //出口在 Maze[m][p]
```

迷宫算法框架（续）

```
if ( !Maze[g][h] && !mark[g][h] ){
//合法的移动且从未走过
  mark[g][h] = 1;
  dir = 下一次将要移动的方向;
  (i, j, dir) 进栈;
  i = g;  j = h;  //当前位置 [i][j] 移到 [g][h] 位置
  dir = N;  //方向置“北”
  }
 }
}
```

迷宫问题的非递归算法

```
void Path ( int m, int p ) {  //本算法输出迷宫的一条路径
// Maze[0][i]=Maze[m+1][i]=Maze[j][0]=Maze[j][p+1]=1
// 0≤ i ≤ p+1，  0 ≤ j ≤ m+1
   int i, j, d, g, h;
   mark[1][1] = 1;  // (1, 1) 是入口
   Stack <items> st (m*p); //设置工作栈
   items tmp;  tmp.x = 1;  tmp.y = 1;  tmp.dir = E;
   //初始坐标三元组
   st.Push (tmp);  //进栈
    while ( st.IsEmpty ( ) == false )  {  //栈非空，  继续前进
        tmp = st.Pop ( );  //退栈
        int i = tmp.x;  int j = tmp.y;  int d = tmp.dir;
        // d 为偏移表下标
```

迷宫问题的非递归算法

```
while ( d<8 ) { //还有移动，继续移动
   int g = i+move[d].a;  int h = j+move[d].b;
   //找下一位置 [g][h]
   if ( g == m && h == p ) { //达到出口
         cout << st;  //重载操作：输出路径
         cout << i << " " << j << endl;
         cout << m << " " << p << endl;
         return;
   }
```

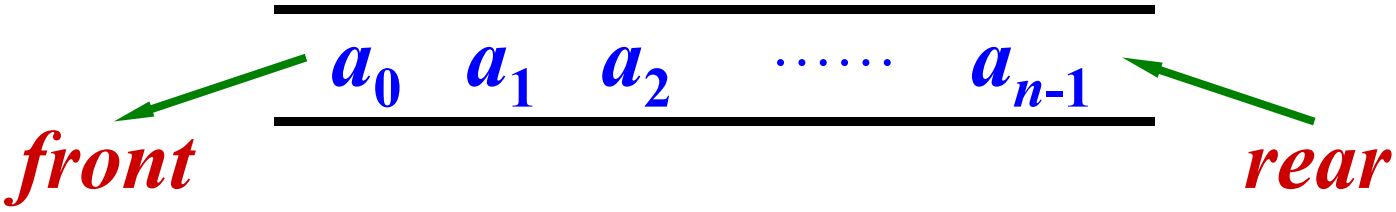
迷宫问题的非递归算法（续）

```
        if ( Maze[g][h] == 0 && mark[g][h] == 0 )
        { //未到达出口，新的位置
            mark[g][h] = 1; //标记为已访问过
            tmp.x = i; tmp.y = j;
            tmp.dir = (directions) (d+1);
            st.Push ( tmp ); //进栈
            i = g; j = h; d = 0; //移动到[g][h]   }
        else d++; //尝试下一个方向
      }
    }
    cout << "No path in Maze" << endl;
}
```

迷宫问题的递归算法

```
int SeekPath ( int x, int y ) {
   int i, g, h, d;
   if ( x == m && y == p) return 1;
   for ( i = 0; i < 8; i ++ ) {
        g = x+move[i].a;  h = y+move[i].b;
        d = i;
        if ( Maze[g][h] == 0 && mark[g][h] == 0 ) {
            mark[g][h] = 1;
            if ( SeekPath ( g, h ) ) {
                cout << "(" << g << "," << h << "),"  << "Direction"
                     << dir << ",";
                return 1;  }    }     }
   if ( x == 1 && y == 1 )  {
        cout << "No Path in Maze" << endl;  return 0;  }
}
```

3.3 队列 (Queue)



- 定义
  - 队列是只允许在一端删除，在另一端插入的线性表。
  - 允许删除的一端叫做队头(front)，允许插入的一端叫做队尾(rear)。
- 特性
  - 先进先出(First In First Out, FIFO)

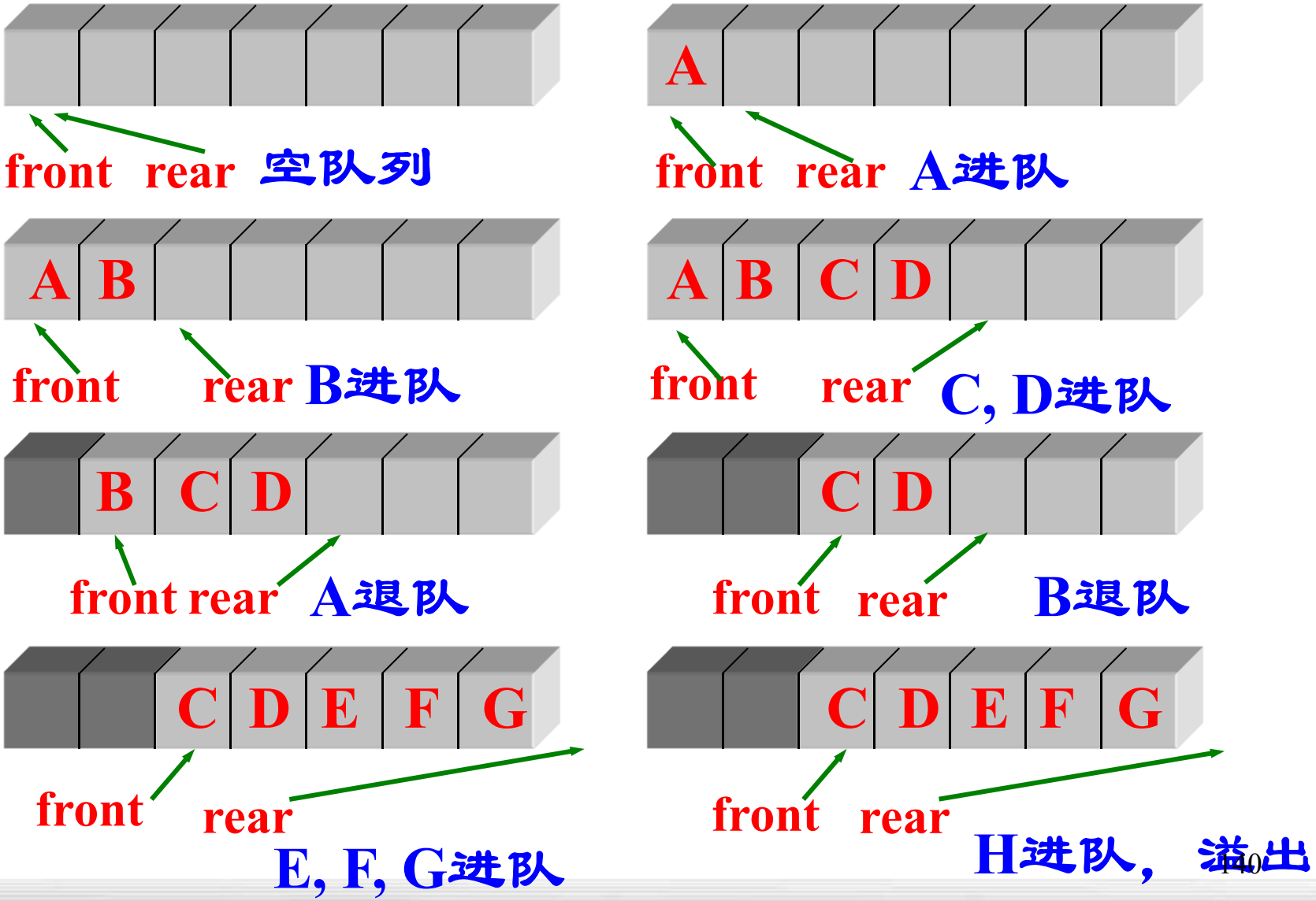
队列类定义

```
template <class Type> class Queue {
public:
    Queue ( ) { }
    ~Queue ( ) { }
    virtual void EnQueue ( const Type &x ) = 0;
    virtual bool DeQueue ( Type &x ) = 0;
    virtual bool GetFront ( Type &x ) = 0;
    virtual bool IsEmpty ( ) const = 0;
    virtual bool IsFull ( ) const = 0;
    virtual bool GetSize ( ) const = 0;
};
```

队列的数组存储表示 -- 顺序队列

```
#include <assert.h>
template <class Type> class SeqQueue {
private:
    int rear, front;  //队尾，队头指针
    Type *elements;  //队列元素数组
    int maxSize;  //最大元素个数
public:
    SeqQueue ( int sz = 10 );
    ~SeqQueue ( ) { delete [ ] elements; }
    bool EnQueue ( const Type &x );  //进队
```

队列的进队和出队

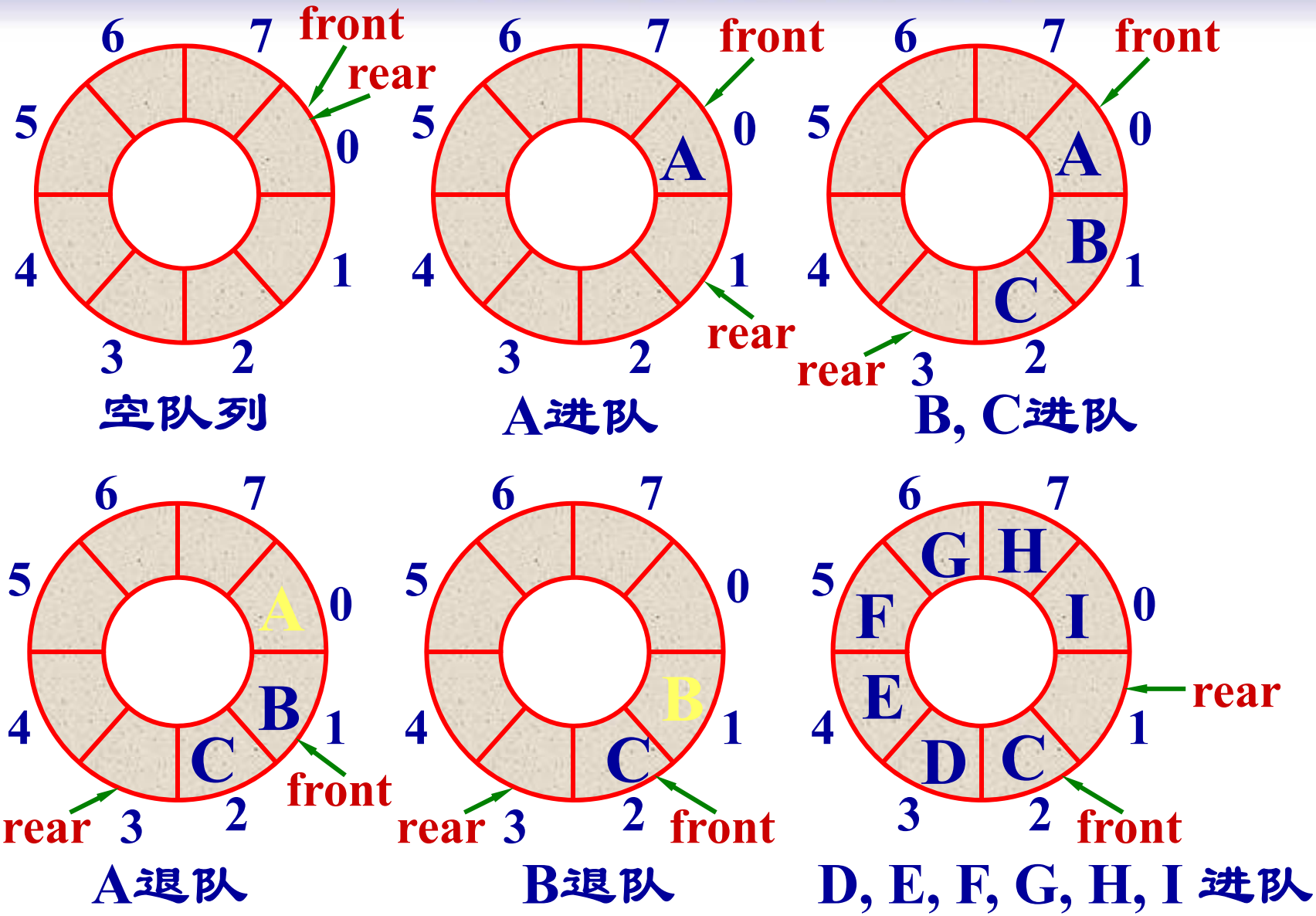

front rear 空队列
A
front rear A进队
A B
front rear B进队
A B C D
front rear C, D进队
B C D
front rear A退队
C D
front rear B退队
C D E F G
front rear E, F, G进队
C D E F G
front rear H进队，溢出

队列的进队和出队原则

- 进队时，先将新元素按 rear 指示位置加入，队尾指针再进一 rear = rear + 1。
  队尾指针指示实际队尾位置的后一位置。
- 出队时，先将下标为 front 的元素取出，队头指针再进一 front = front + 1。
  队头指针指示实际队头所在位置
- 队满时再进队将溢出出错（假溢出）。
- 队空时再出队将队空处理。
- 解决办法之一：将队列元素存放数组首尾相接，形成循环（环形）队列。

循环队列 (Circular Queue)

- 队列存放数组被当作首尾相接的表处理。
- 队头、队尾指针加1时从 maxSize-1 直接进到 0，可用语言的取模（余数）运算实现。
  - ☞队头指针进 1： front = (front+1) % maxSize
  - ☞队尾指针进 1： rear = (rear+1) % maxSize
  - ☞队列初始化：front = rear = 0
  - ☞队空条件：front == rear
  - ☞队满条件：(rear+1) % maxSize == front


6
7
front
rear
5
0
4
1
3
2
空队列
A
rear
A进队
B
C
B, C进队
A退队
B退队
G
H
F
I
E
D
C
D, E, F, G, H, I 进队

循环队列其它操作的定义

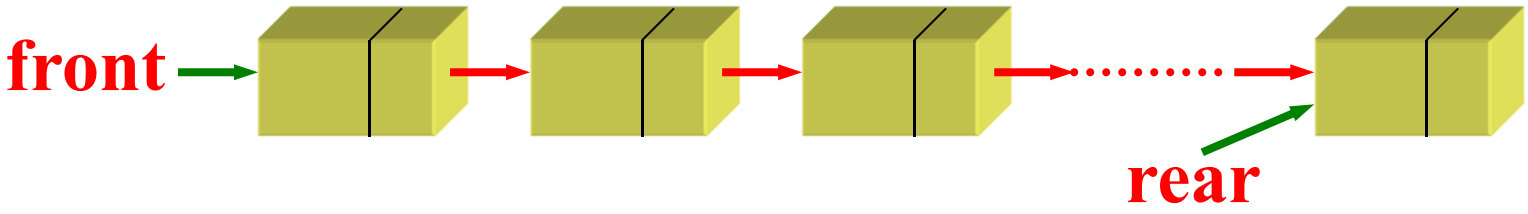
```
bool DeQueue ( Type &x );  //退队
bool GetFront ( Type &x );  //取队头元素
void MakeEmpty ( ) { front = rear = 0; }
bool IsEmpty ( ) const
  { return ( front == rear ) ? true : false; }
bool IsFull ( ) const
  { return ( (rear+1) % maxSize == front ) ? true : false; }
int GetSize ( ) const
  { return ( rear-front+maxSize ) % maxSize; }
friend ostream & operator <<
        ( ostream &os, SeqQueue <Type> &Q );
};
```

```
template <class Type> SeqQueue <Type>::
SeqQueue ( int sz ) : front (0), rear (0), maxSize (sz) {
    elements = new Type [maxSize];
    assert ( elements != NULL );
}

template <class Type> bool SeqQueue <Type>::
EnQueue ( const Type &x ) {
    if ( IsFull ( ) == true ) return false;
    elements[rear] = x;
    rear = (rear+1) % maxSize;
    return true;
}
```

```
template <class Type> bool SeqQueue <Type> ::
DeQueue ( Type &x ) {
    if ( IsEmpty ( ) == true ) return false;
    x = elements[front];
    front = (front+1) % maxSize;
    return true;
}

template <class Type> bool SeqQueue <Type> ::
GetFront (Type &x) const {
    if ( IsEmpty ( ) == true ) return false;
    x = elements[front];
    return true;
}
```

- 使用 maxSize 个位置，设置附加标记 tag 区分队空与队满。
- 最近一次执行
  - EnQueue ( ) — tag == 1；
  - DeQueue ( ) — tag == 0。
- front == rear
  - tag == 1，队满；
  - tag == 0，队空。

队列的链接存储表示 — 链式队列



- 队头在链头，队尾在链尾。
- 链式队列在进队时无队满问题，但有队空问题。
- 队空条件为 front == NULL 。

链式队列类定义

```
template <class Type> class LinkedQueue;

template <class Type> class QueueNode {
friend class LinkedQueue <Type>;
private:
   Type data;  //队列结点数据
   QueueNode <Type> *link;  //结点链指针
public:
   QueueNode ( Type d = 0, QueueNode <Type>
     *next = NULL ) : data(d), link(next) { }
};
```

```
template <class Type> class LinkedQueue {
private:
    QueueNode <Type> *front, *rear;
public:
    LinkedQueue ( ) : rear ( NULL ), front ( NULL ) { }
    ~LinkedQueue { MakeEmpty ( ); }
    bool EnQueue ( const Type &x );
    bool DeQueue ( Type &x );
    bool GetFront ( Type &x ) const;
    void MakeEmpty ( );
    bool IsEmpty ( ) const { return ( front == NULL ) ? true : false; }
    int GetSize ( ) const;
    friend ostream & operator <<
            ( ostream &os, LinkedQueue <Type> &Q );
};
```

```
template <class Type> void LinkedQueue <Type> ::
MakeEmpty ( ) {
    QueueNode <Type> *p;
    while ( front != NULL ) {  //逐个结点释放
        p = front;  front = front->link;
        delete p;
    }
}
```

```
template <class Type> bool LinkedQueue <Type> ::
EnQueue ( const Type &x ) {
//将新元素 x 插入到队列的队尾
   if ( front == NULL ) {  //创建第一个结点
        front = rear = new QueueNode
                                   <Type> (x, NULL);
        if ( front == NULL ) return false;   }
   else {  //队列不空，插入
       rear->link = new QueueNode <Type> (x, NULL);
       if ( rear->link == NULL ) return false;
       rear = rear->link;
}
```

```
template <class Type> bool LinkedQueue <Type> ::
DeQueue ( Type &x ) {
    if ( IsEmpty ( ) == true ) return false;
    QueueNode <Type> *p = front;
    x = front->data;  front = front->link;
    delete p;  return true;
}

template <class Type> bool LinkedQueue <Type> ::
GetFront ( Type &x ) const {
    if ( IsEmpty ( ) == true ) return false;
    x = front->data;  return true;
}
```

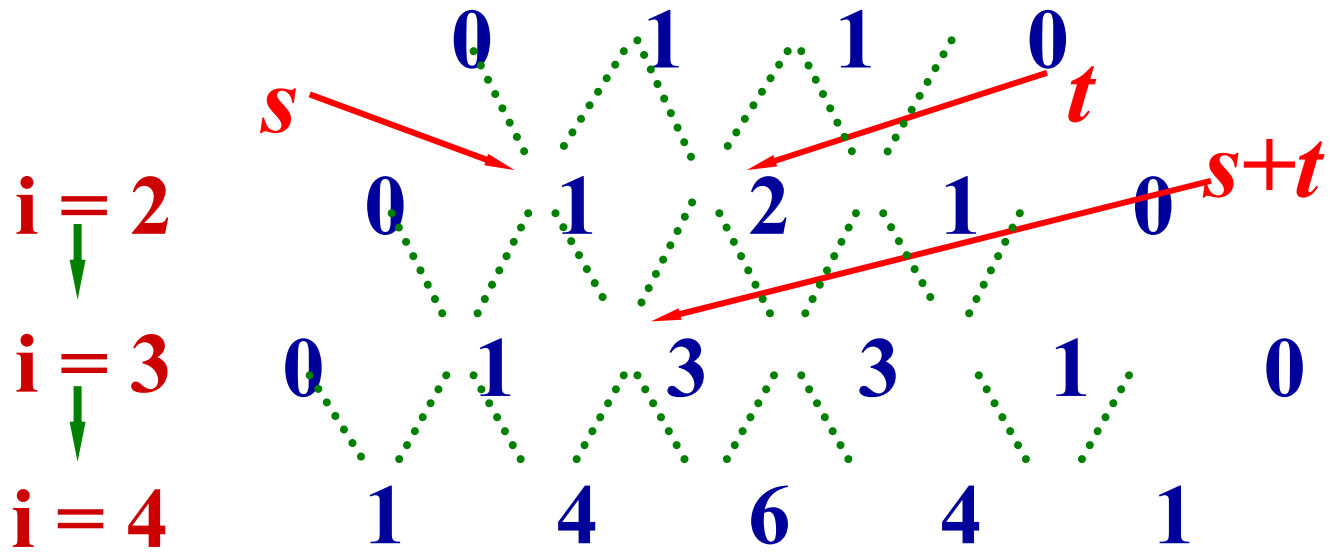
队列的应用举例 — 逐行打印

二项展开式 $(a+b)^i$ 的系数

杨辉三角形 (Pascal's Triangle)

				1		1						$i=1$
			1		2		1					2
		1		3		3		1				3
	1		4		6		4		1			4
1		5		10		10		5		1		5
1		6		15		20		15		6		1		6

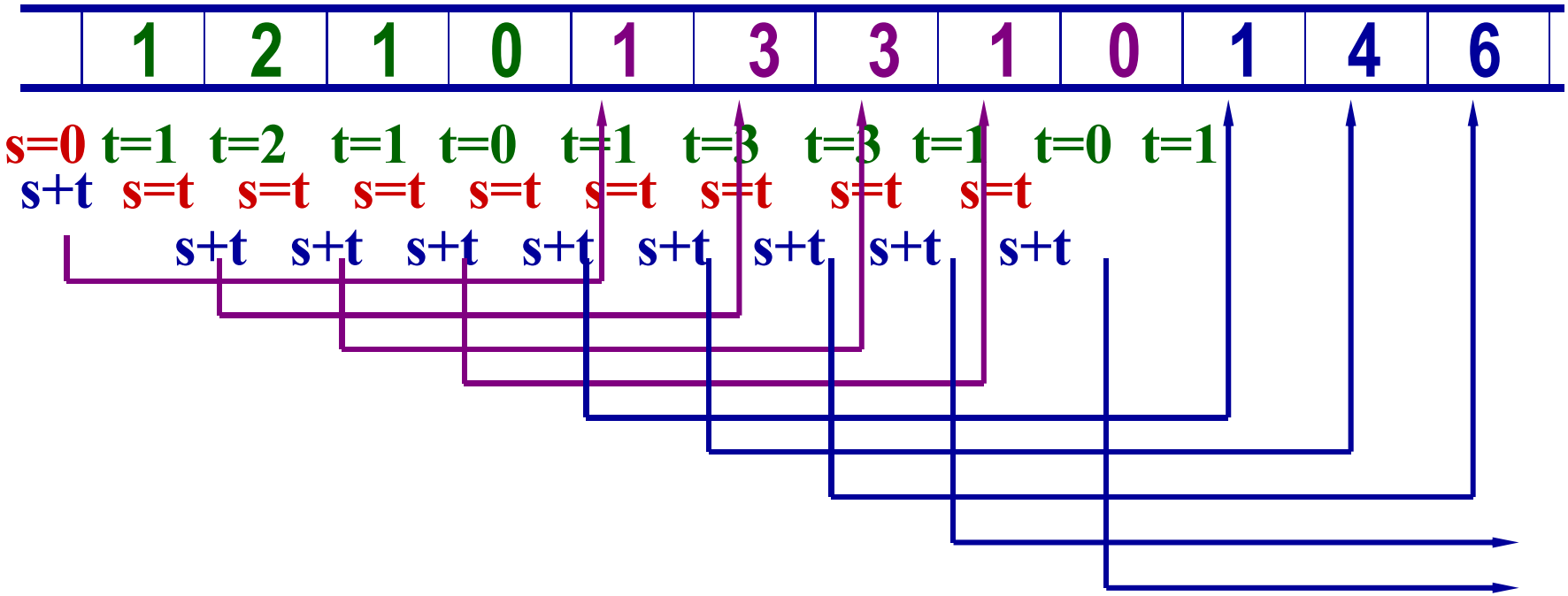
分析第 *i* 行元素与第 *i*+1 行元素的关系



从前一行的数据可以计算下一行的数据

每行的两侧各添加一个0，设*s*是第*i*行第*j*-1个元素的值，*s*初值为0，*t*是第*i*行第*j*个元素的值，则*s*+*t*是第*i*+1行第*j*个元素的值。

从第 i 行数据计算并存放第 $i+1$ 行数据



i=2时，从q中取出t=1，计算$s+t$，得到i=3时第1个数据‘1’，顺序存放在q的后面。让$s=t$，再从数组中取出t=2，计算$s+t$，得到i=3时的第2个数据‘3’，顺序存放在q的后部……一旦第i+1行数据形成，第i行的数据就不再放在数组q中，数组q——队列。

利用队列打印二项展开式系数的程序

```
void YANGHUI ( int n ) {
//在输出上一行系数时，将其下一行的系数预先
//放入队列中，在各行系数之间插入一个 0
    SeqQueue q(n+1);  //队列初始化
    int i = 1, j, s = k = 0, t, u;
    q.EnQueue (i);  q.EnQueue (i);
    //预先放入第一行的两个系数
    for ( int i = 1; i <= n; i++ ) {  //逐行计算
        cout << endl;
        q.EnQueue (k);
```

```
        for ( int j = 1; j <= i+2; j++ ) {  //下一行
        //处理第 i 行的 i+2 个系数（包括一个 0 ）
            q.DeQueue (t);
            u = s+t;  q.EnQueue (u);
            s = t;
            if ( j != i+2 ) cout << s << ' ';
            //打印一个系数，第 i+2 个是 0
        }
    }
}
```

随堂练习

例1：若用单链表来表示队列，则应该选用_____。

例2：利用两个栈来模拟一个队列，已知栈的三种运算定义为：PUSH(ST, x)、POP(ST, x)及SEMPTY(ST)。利用栈的运算来实现该队列的三个运算：进队、出队及判队空。

例3：以带头结点的循环链表表示队列，并且不设头指针，只设队尾指针，试编写相应的队初始化、判队空、入队和出队算法函数。

例1：若用单链表来表示队列，则应该选用带尾指针的循环链表。

设尾指针为tail，则通过tail可以访问队尾，通过tail->next可以访问队头。

例2：利用两个栈来模拟一个队列，已知栈的三种运算定义为：PUSH(ST, x)、POP(ST, x)及SEMPTY(ST)。利用栈的运算来实现该队列的三个运算：进队、出队及判队空。

由于队列是先进先出，而栈是先进后出；所以只有经过两个栈，即先在第一个栈里先进后出，再经过第二个栈后进先出来实现队列的先进先出。因此，用两个栈模拟一个队列运算就是用一个栈作为输入，而另一个栈作为输出。当进队列时，总是将数据进入到作为输入的栈中。在输出时，如果作为输出的栈已空，则从输入栈将已输入到输入栈的所有数据压入栈中，然后由输出栈输出数据；如果作为输出的栈不空，则就从输出栈输出数据。显然，只有在输入，输出栈均为空时队列才为空。一个栈用来插入元素，另一个栈用来删除元素，删除元素时应将前一栈中的所有元素读出，然后进入到第二个栈中。

例2：

```
void Enqueue(s1, x)
        stack s1;
        int x;
        {
            if (s1->top= =n)      /*已达到队列最大值*/
                printf(‘队列上溢”);
            else
                push(s1, x);
        }

        void Dequeue(s1, s2, x)
        stack s1, s2;
        int x;
        {
            s2->top=0;      /*将 s2 清空*/
            while (!Empty(s1))        /*将 s1 的所有元素退栈后压入 s2，此时 s2 为空*/
                push(s2, pop(s1));
            pop(s2, x);      /*弹出栈 s2 的栈顶元素（也即为队首元素）并赋给 x*/
            while (!Empty(s2))        /*将剩余元素重新压入栈 s1 恢复为原 s1 中的顺序*/
                push(s1, pop(s2));
        }

        int Queue_empty(s1)
        stack s1;
        {
            if Empty(s1)
                return (1);
            else return (0);
        }
```

例3：

```
void initLoopQueue(LinkQueue *qpt)
{
    qpt->rear = (QNODE *) malloc (sizeof(QNNODE));
    qpt->rear->next = qpt->rear;
}
int isLoopQEmpty(LinkQueue *qpt)
{
   return qpt->rear = = qpt->rear->next;
}
void inLoopQueue(LinkQueue *qpt, dataType x)
{
      QNODE *p = (QNODE *)malloc(sizeof(QNODE));
      p->data = x; p->next = qpt->rear->next;
      qpt->rear = qpt->rear->next = p;
}
int deLoopQueue(LinkQueue *qpt, dataType *xpt)
{
    QNODE *p = qpt->rear->next->next;//队列首结点指针
    if(isLoopQEmpty(qpt))
       return 0;
    *xpt = p->data;
    qpt->rear->next->next = p->next;
    free(p);
    return 1;
}
int lookFront(LinkQueue *qpt, dataType *xpt)
{
      if(isLoopQEmpty(qpt))
           return 0;
      *xpt = qpt->rear->next->next->data;
      return 1;
}
```

3.4 优先级队列 (Priority Queue)

- **优先级队列** 每次从队列中取出的是具有最高优先权的元素。
- 如下表：任务优先权及执行顺序的关系。

任务编号	1	2	3	4	5
优先权	20	0	40	30	10
执行顺序	3	1	5	4	2

数字越小，优先权越高

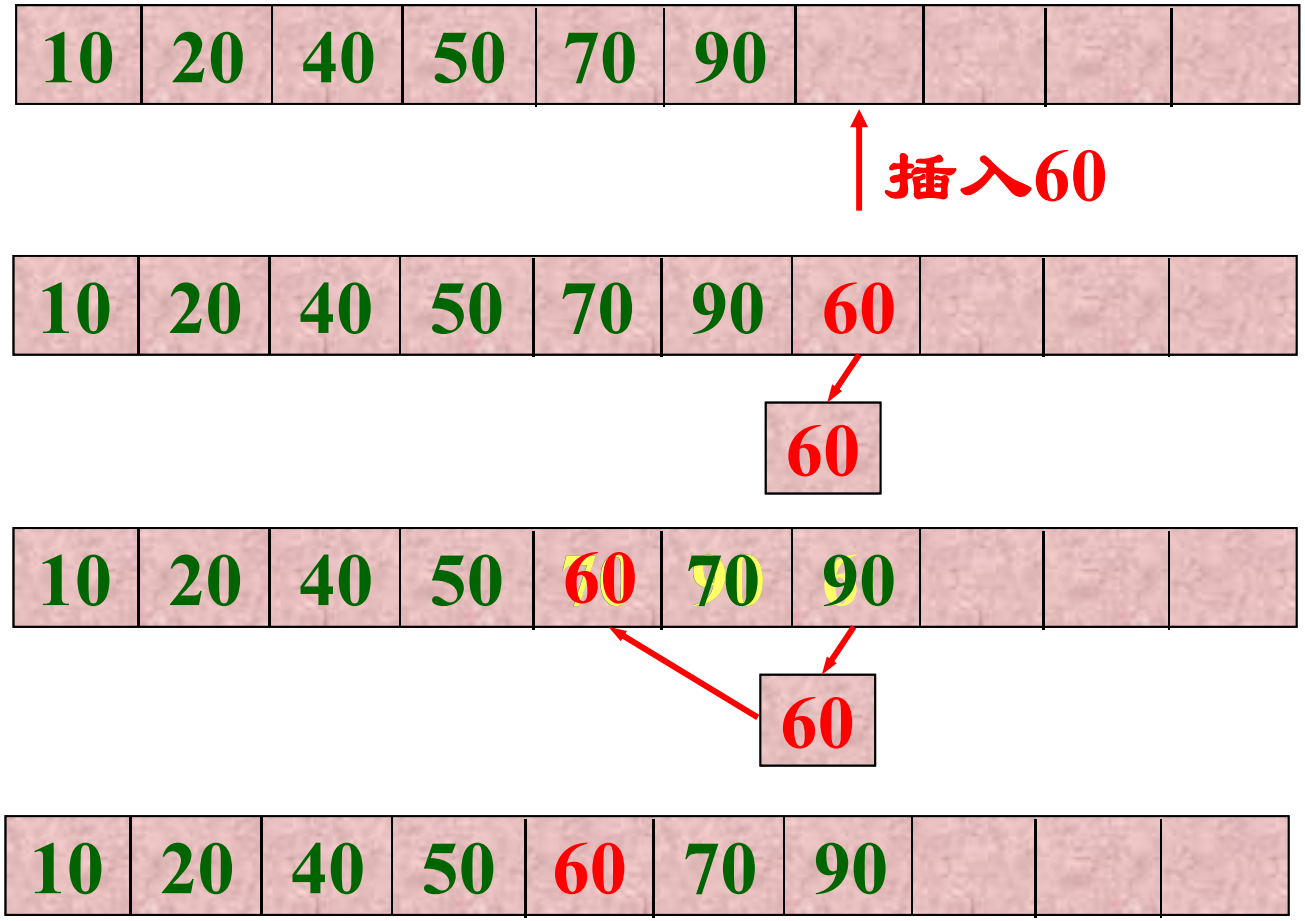
优先级队列类定义

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

template <class Type> class PQueue {
private:
    Type *pqelements;  //存放数组
    int count;  //队列元素计数
    int maxPQSize;  //最大元素个数
    void Adjust ( );  //调整
```

```
public:
    PQueue ( int sz = 50 );
    ~PQueue ( ) { delete [ ] pqelements; }
    bool Insert ( const Type &x );
    bool RemoveMin ( Type &x );
    bool GetFront ( Type &x );
    void MakeEmpty ( ) { count = 0; }
    bool IsEmpty ( ) const
        { return ( count == 0 ) ? true : false; }
    bool IsFull ( ) const
        { return ( count == maxPQSize ) ? true : false; }
    int GetSize ( ) const { return count; }
};
```

优先级队列部分成员函数的实现

```
template <class Type> PQueue <Type> ::
PQueue ( int sz ) {
    maxPQSize = sz;  count = 0;
    pqelements = new Type [maxPQSize];
    assert ( pqelements != NULL );
}

template <class Type> bool PQueue <Type> ::
Insert ( const Type &x ) {
    if ( count == maxSize ) return false;
    pqelements[count++] = x;
    Adjust ( );
}
```


10 20 40 50 70 90
插入60
10 20 40 50 70 90 60
60
10 20 40 50 60 70 90
60
10 20 40 50 60 70 90

```
template <class Type> void PQueue <Type> ::
Adjust ( ) {
    Type temp = pqelements[count-1];
     //将最后元素暂存再从后向前找插入位置
    for ( int j = count-2; j >= 0; j-- )
        if ( pqelements[j] <= temp ) break;
        else pqelements[j+1] = pqelements[j];
    pqelements[j+1] = temp;
}
```

```
template <class Type> bool PQueue <Type> ::
RemoveMin ( Type &x ) {
    if ( count == 0 ) return false;
    x = pqelements[0];  //取出 0 号元素
    for ( int i = 1; i < count; i++ )
        pqelements[i-1] = pqelements[i];
        //从前向后逐个移动元素填补空位
    count--;
    return true;
}
```

```
template <class Type>
bool PQueue <Type> :: GetFront ( Type &x ) {
    if ( count == 0 ) return false;
    x = pqelements[0];
    return true;
}
```

3.5 双端队列 (Double-Ended Queue)

- 定义
  - 在队列的两端进行插入和删除。

双端队列类定义

```
template <class Type> class DQueue {
public:
    virtual bool GetHead ( Type &x ) = 0;
    virtual bool GetTail ( Type &x ) = 0;
    virtual bool EnQueueHead ( const Type &x ) = 0;
    virtual bool EnQueueTail ( const Type &x ) = 0;
    virtual bool EnQueue ( const Type &x ) = 0;
    virtual bool DeQueue ( Type &x ) = 0;
    virtual bool DeQueueHead ( Type &x ) = 0;
    virtual bool DeQueueTail ( Type &x ) = 0;
};
```

```
template <class Type> bool DQueue <Type> ::
EnQueue ( const Type &x ) {
    return EnqueueTail (x);
}

template <class Type> bool DQueue <Type> ::
DeQueue ( Type &x ) {
    Type temp;
    bool tag = DeQueueHead (temp);
    x = temp;
    return tag;
}
```

双端队列的数组表示

```
template <class Type>
class SeqDQueue : public DQueue <Type>,
                  public SeqQueue <Type> {
public:
    SeqDQueue ( int sz );
    ……
};
```

```
template <class Type> bool SeqDQueue <Type> ::
GetHead ( Type &x ) const {
    Type temp;
    bool tag = SeqQueue <Type> :: GetFront (temp);
    x = temp;  return tag;
}
template <class Type> bool SeqDQueue <Type> ::
EnQueueTail ( Type &x ) {
    return SeqQueue <Type> :: Enqueue (x);
}
template <class Type> bool SeqDQueue <Type> ::
DeQueueHead ( Type &x ) {
    Type temp;
     bool tag = SeqQueue <Type> :: DeQueue (temp);
     x = temp;  return tag;
}
```

```
template <class Type> bool SeqDQueue <Type> ::
GetTail ( Type &x ) const {
    if ( front == rear ) return false;
    x = elements[( rear-1+maxSize ) % maxSize];
    return true;
}

template <class Type> bool SeqDQueue <Type> ::
EnQueueHead ( const Type &x ) {
    if ( ( rear+1 ) % maxSize == front ) return false;
    front = ( front-1+maxSize ) % maxSize;
    elements[front] = x;
    return true;
}
```

```
template <class Type> bool SeqDQueue <Type> ::
DeQueueTail ( Type &x ) {
    if ( front == rear ) return false;
    rear = ( rear-1+maxSize ) % maxSize;
    x = elements[rear];
    return true;
}
```

双端队列的链表表示

```
template <class Type>
class LinkedDQueue : public DQueue <Type>,
                     public LinkedQueue <Type> {
public:
    LinkedDeQue ( );
    ……
};
```

```
template <class Type> bool LinkedDQueue <Type> ::
GetHead ( Type &x ) const {
    Type temp;
     bool tag = LinkedQueue <Type> :: GetFront (temp);
     x = temp;  return tag;
}
template <class Type> bool LinkedDQueue <Type> ::
EnQueueTail ( Type &x ) {
     return LinkedQueue <Type> :: Enqueue (x);
}
template <class Type> bool LinkedDQueue <Type> ::
DeQueueHead ( Type &x ) {
      Type temp;
      bool tag = LinkedQueue <Type> :: DeQueue (temp);
      x = temp;  return tag;
}
```

```
template <class Type> bool LinkedDQueue <Type> ::
GetTail ( Type &x ) const {
    if ( front == NULL ) return false;
    x = rear->data;
    return true;
}

template <class Type> bool LinkedDQueue <Type> ::
EnQueueHead ( const Type &x ) {
    QueueNode <Type> *p = new QueueNode <Type> (x);
    if ( p == NULL ) return false;
    p->link = front;  front = p;
    return true;
}
```

```
template <class Type> bool LinkedDQueue <Type> ::
DeQueueTail ( Type &x ) {
    if ( front == NULL ) return false;
    while ( p->link != rear ) p = p->link;
    x = rear->data;  delete rear;
    p->link = NULL;  rear = p;
    return true;
}
```

本章小结

- 知识点
  - 栈：顺序表示和链式表示
  - 表达式求值
  - 队列：顺序表示和链式表示
  - 优先级队列和双端队列
  - 递归算法和非递归算法

常见题型

- 给定入栈顺序求合法的出栈顺序、判断栈的大小
- 表达式计算，中缀转后缀
- 队列的基本运算
- 与后续的遍历、回溯等问题的结合。

- **课程习题**

  - **笔做题——3.16, 3.23, 3.24, 3.26**
    (以作业形式提交)

  - **上机题——3.21, 3.27, 3.28**

  - **思考题——3.1, 3.5, 3.11, 3.25, 3.29, 3.30**