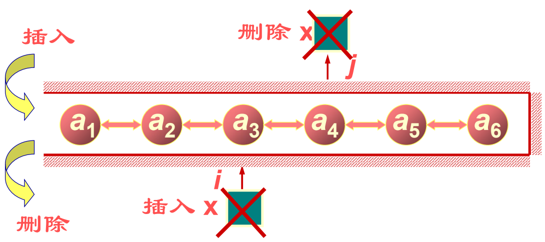
第三章 栈和队列

如此巧妙 存取如此规整的
栈和队列

第三章 栈和队列

- 栈 (Stack)
- 队列
- 队列的应用：打印杨辉三角形
- 栈的应用：括号配对
- 栈的应用：表达式求值
- 栈的应用：递归
- 双端队列
- 优先队列

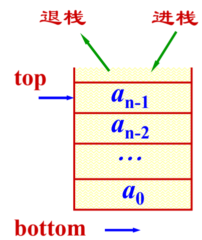
111-2



栈 (Stack)

栈 (Stack)

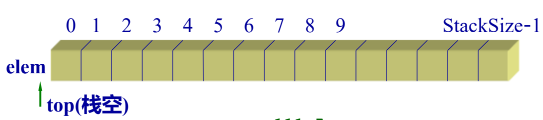
- 只允许在一端插入和删除的线性表。允许插入和删除的一端称为栈顶 (top)，另一端称为栈底 (bottom)
- 特点：后进先出(LIFO)
- 栈的主要操作
  - 进栈 Push(S, x)
  - 退栈 Pop(S, &x)
  - 看栈顶 getTop(S, &x)
  - 置空栈 initStack(S)
  - 判栈空，判栈满



111-4

栈的顺序表示 — 顺序栈

```
#define initSize 100
#define increament 20
typedef int SElemType;
typedef struct {            //顺序栈定义
    SElemType *elem;        //栈数组
    int top, maxSize;       //栈顶指针及栈大小
} SeqStack;
```



111-5

```
void initStack ( SeqStack& S ) {          //初始化
  S.elem = (SElemType *) malloc ( initSize*
        sizeof ( SElemType ));
  if (S.elem == NULL)
      { printf (“存储分配失败!\n”);  exit(1); }
   S.top = -1;  S.maxSize = initSize;
}
bool stackEmpty (SeqStack& S) {
//判断栈是否空? 空则返回true，否则返回false
   return S.top == -1;
}
```

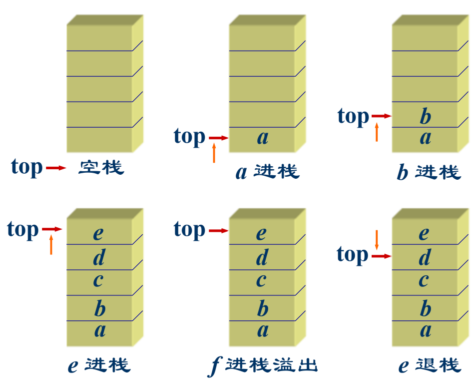
111-6

```
bool stackFull ( SeqStack& S ) {
//判断栈是否满？满则返回true，否则返回false
    return S.top == S.maxSize-1;
}

void Push ( SeqStack& S, SElemType x ) {
//若栈不满则新元素 x 进栈，否则扩大空间再进栈
    if ( stackFull(S) ) overFlow(S);  //栈满溢出处理
    S.top++;
    S.elem[S.top] = x;                //加入新元素
}
```

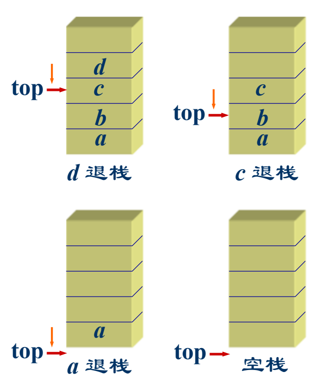
111-7

```
void overFlow(SeqStack& S) {  //栈满处理
    int newSize = S.maxSize + increment;
    SElemType *newS = (SElemType *) malloc (newSize*
        sizeof (SElemType ));     //创建新数组
    for ( int i = 0; i <= S.top; i++ )
        newS[i] = S.elem[i];      //向新数组传送数据
    free (S.elem);                //释放老数组
    S.elem = newS;                //新数组作为栈数组
    S.maxSize = newSize;          //新数组大小
}                                 //栈顶指针不变
```

111-8



111-9



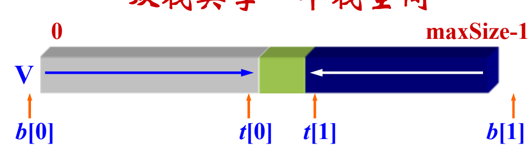
栈顶指针指示
实际栈顶位置
即最后加入新
元素的位置

111-10

```
bool Pop ( SeqStack& S, SElemType& x ) {
//若栈空返回false, 否则栈顶元素退出到x并返回true
    if ( stackEmpty(S) ) return false;  //栈空返回false
    x = S.elem[S.top];  S.top--;        //否则先取再退
    return true;                        //返回true
}

bool getTop ( SeqStack& S, SElemType& x ) {
//若栈空返回false, 否则栈顶元素读到x并返回true
    if ( stackEmpty(S) ) return false;  //栈空返回false
    x = S.elem[S.top];                  //否则读栈顶
    return true;                        //返回true
}
```

111-11

双栈共享一个栈空间



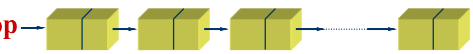
- 两个栈共享一个数组空间V[maxSize]，设立栈顶指针数组 t[2] 和栈底指针数组 b[2]
- t[i]和b[i]分别指示栈 i 的栈顶与栈底（i = 0, 1）
- 初始 t[0] = b[0] = –1， t[1] = b[1] = maxSize
- 栈满条件：t[0]+1 == t[1]　　//栈顶指针相遇
- 栈空条件：t[0] = b[0] 或 t[1] = b[1]　//退到栈底

111-12

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


栈的链接表示 — 链式栈

top



- 顺序栈有栈满问题，一旦栈满需要做溢出处理，扩充栈空间，时间和空间开销较大。
- 链式栈无栈满问题，只要存储空间还有就可扩充。
- 链式栈的栈顶 在链头，插入与删除仅在栈顶处执行。
- 链式栈适合于多栈操作，无需大量移动存储。
- 本课件所述的链式栈不带头结点。

链式栈的结构定义

```
typedef int SElemType;      //每个栈元素的数据类型

typedef struct node {          //栈元素结点定义
    SElemType data;            //结点数据
    struct node *link;         //结点链接指针
} LinkNode, *LinkList, *LinkStack;     //链式栈

void initStack ( LinkStack& S ) {   //栈初始化
    S = NULL;                //置空栈
}
```



```
bool stackEmpty ( LinkStack& S ) {    //判栈空否
    return S == NULL;
}
bool Push ( LinkStack& S, SElemType x ) {  //进栈
    LinkNode *p = (LinkNode *) malloc ( sizeof
        (LinkNode ));                        //创建新结点
    if ( p == NULL )
      { printf ("结点创建失败!\n");  exit (1); }
    p->data = x;                            //结点赋值
    p->link = S;  S = p;  return true;      //链入栈顶
}
```



```
bool Pop ( LinkStack& S, SElemType& x ) {   //退栈
    if ( stackEmpty(S) ) return false;
    LinkNode * p = S;
    x = p->data;  S = p->link;        //摘下原栈顶
    free (p);  return true;          //释放原栈顶结点
}
bool getTop ( LinkStack& S, SElemType& x ) {
//看栈顶元素
    if ( stackEmpty (S) ) return false;
    x = S->data;  return true;
}
```


栈的混洗

- “混洗”原意是重新洗牌。用在本领域，问题的提法是：当进栈元素的编号为1, 2, …, *n*时，可能的出栈序列有多少种?
- 当进栈序列为1, 2时，可能的出栈序列有 2种：1, 2（进1出1进2出2）和 2, 1（进1进2出2出1）；
- 当进栈序列为1, 2, 3时，可能的出栈序列有 5 种:
  - 1, 2, 3（进1 出1 进2 出2 进3 出3）
  - 1, 3, 2（进1 出1 进2 进3 出3 出2）
  - 2, 1, 3（进1 进2 出2 出1 进3 出3）

- 2, 3, 1 （进1 进2 出2 进3 出3 出1）
- 3, 2, 1 （进1 进2 进3 出3 出2 出1）

- 注意，3, 1, 2 是不可能的出栈序列，因为若 3 第1个出栈，栈内一定是 1 压在 2 的下面，1 不可能先于 2 出栈。
- 一般情形如何呢？若设进栈序列为1, 2, …, n，可能的出栈序列有 m_n 种，则
  - $n = 0$时，$m_0 = 1$: 出栈序列为{}。
  - $n = 1$时，$m_1 = 1$: 出栈序列为{1}。
  - $n = 2$时，$m_2 = 2$:

① 出栈序列中 1 在首位，1 左侧有 0 个数，右侧有 1 个数，有 $m_0*m_1 = 1$ 种出栈序列：{1, 2}

② 出栈序列中 1 在末位，1 左侧有 1 个数，右侧有 0 个数，有 $m_1*m_0 = 1$ 种出栈序列：{2, 1}。

➢ 可能出栈序列有 $m_0*m_1+m_1*m_0 = 2$种。

- $n = 3$时，$m_3 = 5$:

① 出栈序列中 1 在首位，1 左侧有 0 个数，右侧有 2 个数，有 $m_0*m_2 = 2$种出栈序列：{1, 2, 3}和{1, 3, 2}。

② 出栈序列中 1 在第 2 位，1 左侧 1 个数，右侧 1 个数，有 $m_1*m_1 = 1$ 种出栈序列：{2, 1, 3}。

③ 出栈序列中 1 在第 3 位。1 左侧 2 个数，右侧 0 个数，有 $m_2*m_0 = 2$ 种出栈序列：{2, 3, 1}和 {3, 2, 1}。

➢ 可能的出栈序列有 $m_0*m_2+m_1*m_1+m_2*m_0 = 2+1+2 = 5$种。

- $n = 4$，$m_4 = 14$:

① 出栈序列中 1 在第 1 位。1 左侧 0 个数，右侧 3 个数，有 $m_0*m_3 = 5$ 种出栈序列：

{1, 2, 3, 4}, {1, 2, 4, 3}, {1, 3, 2, 4}, {1, 3, 4, 2}, {1, 4, 3, 2}。

③ 出栈序列中 1 在第 2 位。1 左侧 1 个数，右侧 2 个数，有 $m_1*m_2 = 2$种出栈序列：{2, 1, 3, 4}, {2, 1, 4, 3}。

④ 出栈序列中 1 在第 3 位。1 左侧 2 个数，右侧 1 个数，有 $m_2*m_1 = 2$种出栈序列：{2, 3, 1, 4}, {3, 2, 1, 4}。

⑤ 出栈序列中 1 在第 4 位。1 左侧 3 个数，右侧 0 个数，有 $m_3*m_0 = 5$种出栈序列：{2, 3, 4, 1}, {2, 4, 3, 1}, {3, 2, 4, 1}, {3, 4, 2, 1}, {4, 3, 2, 1}。

➢ 可能出栈序列 $m_0*m_3 + m_1*m_2 + m_2*m_1 + m_3*m_0 = 5+2+2+5 = 14$ 种。

- 一般地，有 n 个元素按序号1, 2, …, n 进栈，轮流让 1 在出栈序列的第 1, 第 2, …第 n 位，则可能的出栈序列数为：

$$\sum_{i=0}^{n-1} m_i * m_{n-i-1} = m_0 * m_{n-1} + m_1 * m_{n-2} + \cdots + m_{n-1} * m_0$$

推导结果为：

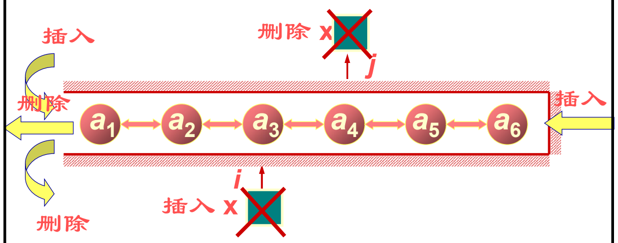
$$\sum_{i=0}^{n-1} m_i * m_{n-i-1} = \frac{1}{n+1} C_{2n}^{n}$$

例题：元素a, b, c, d, e 依次进入初始为空的栈中，若元素进栈后可停留、可出栈，直到所有元素都出栈，则在所有可能的出栈序列中，以元素 d 开头的序列个数是

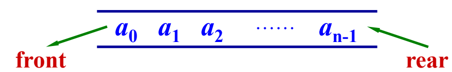
A．3　　B．4　　C．5　　D．6

解答：选B。如果 d 在出栈序列中排在开头，a, b, c 必须在 d 出栈前压在栈内，d 出栈后它们才可出栈，出栈顺序必须是c, b, a。e 可夹在它们之间进栈和出栈。因此，以d开头的出栈序列只可能是

d, e, c, b, a　d, c, e, b, a　d, c, b, e, a　d, c, b, a, e



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 (FIFO, First In First Out)

111-26

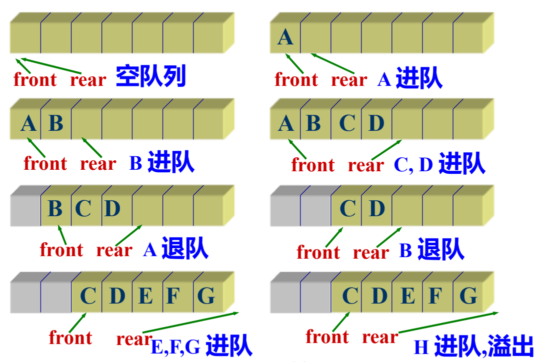
队列的顺序存储表示 — 顺序队列

```
#define queSize 50;
typedef int QElemType;      //每个元素的数据类型

typedef struct {             //顺序队列的结构定义
    QElemType elem[queSize];  //队列存储数组
    int rear, front;          //队尾与队头指针
} SeqQueue, CircQueue;
```

- 队列与栈的共性在于它们都是**限制了存取位置的线性表**；区别在于存取位置有所不同。

111-27

队列的进队和出队



111-28

队列的进队和出队的原则

- 有两种进/出队列的方案：

1. **先加元素再动指针**
   - 进队时先将新元素按 rear 指示位置加入，再让队尾指针进一 rear = rear + 1。
   - 队尾指针指示实际队尾的后一位置。
   - 出队时先将下标为 front 的元素取出，再将队头指针进一 front = front + 1。
   - 队头指针指示实际队头的位置。
   - 清华、北大教材均为此方案。

111-29

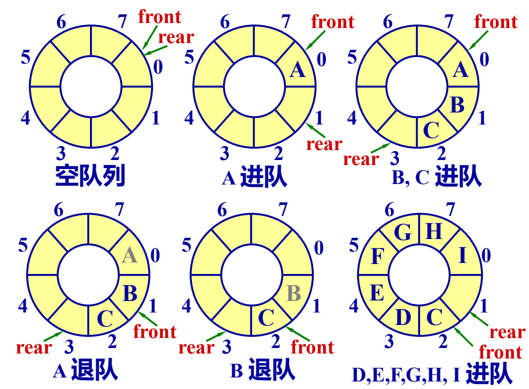
2. **先动指针再加元素**
   - 进队时先让队尾指针进一 rear = rear + 1，再将新元素按 rear 指示位置加入。
   - 队尾指针指示实际队尾的位置。
   - 出队时先将队头指针进一 front = front + 1，再将下标为 front 的元素取出。
   - 队头指针指示实际队头的前一位置。
   - 微软Visual C++ STL 按此处理。

- 队满时再进队出现的**溢出**往往是**假溢出**，即还有空间但用不上，为了有效利用队列空间，可将队列元素存放数组首尾相接，形成循环队列。

111-30

循环队列(Circular Queue)

- 队列存放数组被当作首尾相接的环形表处理。
- 队头、队尾指针加 1 时从 queSize-1 直接进到0，可用语言的取模（%）运算实现。
  - 队头指针进1：front = (front+1) % queSize;
  - 队尾指针进1：rear = (rear+1) % queSize;
  - 队列初始化：front = rear = 0;
  - 队空条件：front == rear;
  - 队满条件：(rear+1) % queSize == front。
- 注意，进队和出队时指针都是顺时针前进。



循环队列操作的实现

```
void initQueue ( CircQueue& Q ) {       //置空队列
    Q.rear = 0;  Q.front = 0;
}
bool queueEmpty ( CircQueue& Q ) {   //判队空否
    return Q.rear == Q.front;
}
bool queueFull ( CircQueue& Q ) {       //判队满否
    return (Q.rear+1) % queSize == Q.front;
}
```

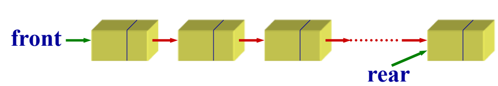


```
//当进队速度快于出队速度，rear追上front，造成了
//队满，为了区分队空条件，认定当rear+1 == front
//时队列已满。这样不必增加其他辅助单元。

bool enQueue ( CircQueue& Q, QElemType x ) {
//在循环队列Q的队尾加入新元素 x
    if ( queueFull(Q) ) return false;      //队已满
    Q.elem[Q.rear] = x;                    //否则，先加
    Q.rear = (Q.rear+1) % queSize;         //队尾指针进一
    return true;
}
```



```
bool deQueue ( CircQueue& Q, QElemType& x ) {
    if ( queueEmpty(Q) ) return false;
    x = Q.elem[Q.front];                   //先取队头的值
    Q.front = (Q.front+1) % queSize;       //队头指针进一
    return true;
}

bool getFront ( CircQueue& Q, QElemType& x ) {
    if ( queueEmpty(Q) ) return false;
    x = Q.elem[Q.front];  return true;
}
```


队列的链接表示 — 链式队列



- 链式队列采用不带头结点的单链表存储队列元素，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，不必判断是否 rear == front。
- 链式队列特别适合多个队列同时操作的情形。在并行处理、排序等方面有用。

链式队列的结构定义

```
typedef int QElemType;          //元素的数据类型

typedef struct node {
    QElemType data;             //队列结点数据
    struct node *link;          //结点链指针
} LinkNode;

typedef struct {                //队列结构定义
    LinkNode *rear, *front;     //队尾与队头指针
} LinkQueue;
```



```
void initQueue ( LinkQueue& Q ) {
    Q.front = NULL;  Q.rear = NULL;  //置空队列
}

bool queueEmpty ( LinkQueue& Q ) {
    return Q.front == NULL;          //判队空否
}

bool getFront ( LinkQueue& Q, QElemType& x ) {
    if ( queueEmpty(Q) ) return false;
    x = Q.front->data;  return true;    //读取队头元素
}
```



```
bool enQueue ( LinkQueue& Q, QElemType x ) {
    LinkNode *s = (LinkNode *) malloc ( sizeof
        (LinkNode));               //创建新队尾结点
    s->data = x;  s->link = NULL;
    if ( Q.rear == NULL )          //新结点加入空队
        { Q.rear = s;  Q.front = s; }
    else                           //新结点成新链尾
        { Q.rear->link = s;  Q.rear = s; }
    return true;
}
```



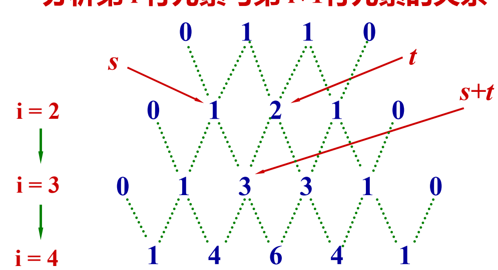
```
bool deQueue ( LinkQueue& Q, QElemType& x ) {
//删去队头结点，并返回队头元素的值
    if ( queueEmpty(Q) ) return false;   //队空不能删
    LinkNode *p = Q.front;
    x = p->data;                         //保存队头的值
    Q.front = p->link;                   //新队头
    if ( Q.front == NULL ) Q.rear = NULL;
        //若删除后队空，队尾指针置为空
    free (p);  return true;
}
```


队列的应用：打印杨辉三角形

- 算法逐行打印二项展开式 $(a + b)^i$ 的系数：杨辉三角形 (Pascal's triangle)

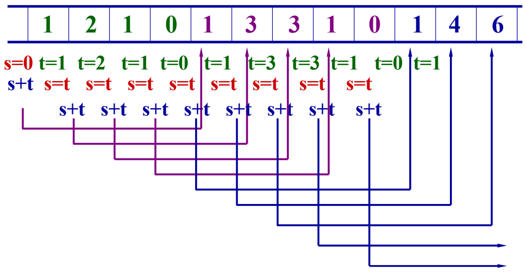
				1		1						i = 1
			1		2		1					2
		1		3		3		1				3
	1		4		6		4		1			4
1		5		10		10		5		1		5
1	6	15	20	15	6	1						6

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

从第 i 行数据计算并存放第 i+1 行数据



利用队列打印二项展开式系数的算法

```
#include "Linkqueue.cpp"
void Yanghui ( int n ) {
    LinkQueue Q;                    //创建队列Q
    InitQueue (Q);
    EnQueue(Q, 1);  EnQueue(Q, 1); //第 1 行系数进队
    int s = 0, t;
```



```
  for (int i = 1; i <= n; i++ ) {          //逐行输出
     printf ("\n");
     EnQueue (Q, 0);
     for ( int j = 1; j <= i+2; j++ ) {    //下一行
        DeQueue(Q, t);   EnQueue(Q, s+t);
            //计算下一行系数, 并进队列
        s = t;
        if ( j != i+2 ) printf ("%d", s);  //输出一个系数
     }
  }
}
```


栈的应用：括号匹配

- 若用字符串描述的表达式为“(a+(b-c)*d)+e/f”，设置一个栈 S 用于判断括号是否配对。然后从左向右扫描表达式中的每一个字符：
  - 当遇到左括号“(”，进栈，扫描下一字符；
  - 当遇到右括号“)”，判断栈是否空?
    - 若栈空，则“)”多于“(”，报错；
    - 若栈不空，则退栈顶的“(”，扫描下一字符；
  - 当遇到的不是括号，扫描下一字符；
  - 若表达式扫描完，栈不空，则“(”多于“)”，报错。

	栈内容	当前字符	动作	剩余字符
0	空	“(”		“(a+(b-c)*d)+e/f”
1	“(”	“a”	“(” 进栈	“a+(b-c)*d)+e/f”
2	“(”	“+”	“a”跳过	“+(b-c)*d)+e/f”
3	“(”	“(”	“+”跳过	“(b-c)*d)+e/f”
4	“(”，“(”	“b”	“(”进栈	“b-c)*d)+e/f”
5	“(”，“(”	“-”	“b”跳过	“-c)*d)+e/f”
6	“(”，“(”	“c”	“-”跳过	“c)*d)+e/f”
7	“(”，“(”	“)”	“c”跳过	“)*d)+e/f”
8	“(”	“*”	“(” 退栈	“*d)+e/f”

	栈内容	当前字符	动作	剩余字符
9	“(”	“d”	“*” 跳过	“d)+e/f”
10	“(”	“)”	“d” 跳过	“)+e/f”
11	空	“+”	“(”退栈	“+e/f”
12	空	“e”	“+”跳过	“e/f”
13	空	“/”	“e”跳过	“/f”
14	空	“f”	“/”跳过	“f”
15	空	–	“f”跳过	“”
16	空	–	结束	括号全部配对

```
int BracketsCheck ( char e[ ], int n ) {
//对字符数组e[n]中的表达式进行括号配对检查。
//若括号匹配，函数返回1，否则函数返回0。
  SeqStack S;  initStack(S);       //定义一个栈
  for ( int i = 0; i < n; i++ )    //顺序扫描e[n]中字符
    if ( e[i] == '{' || e[i] == '[' || e[i] == '(' )
      Push ( S, e[i] );            //左括号进栈
    else if ( e[i] == '}' ) {
      if ( stackEmpty(S) )
        { printf ( " '{'比'}'少!\n" ); return 0; }
      if ( getTop(S) != '{' ) {
```

111-49

```
        printf ( "%c与'}'不配对!\n", getTop(S) );
        return 0;
      }
      Pop(S);                      //花括号配对出栈
    }
    else if ( e[i] == ']' ) {
      if ( stackEmpty(S) )
        { printf ("缺'['!\n"); return 0; }
      if ( getTop(S) != '[' ) {
        printf ("%c与']'不配对!\n", getTop(S) );
        return 0;
      }
```

111-50

```
        Pop(S);                    //方括号配对出栈
    }
    else if ( e[i] == ')' ) {
      if ( stackEmpty(S) )
        { printf ("缺'('!\n"); return 0; }
      if ( getTop(S) != '(' ) {
        printf ("%c与')'不配对!\n ", getTop(S));
        return 0;
      }
      Pop(S);                      //圆括号配对出栈
    }
  if ( stackEmpty(S) )             //表达式扫描完且栈空
```

111-51

```
    { printf ("括号配对!  \n");  return 1; }
  else {
    while ( ! stackEmpty(S) )
      if ( getTop(S) == '{' )
        { printf ("缺'}' ");  Pop(S); }
      else if ( getTop(S) == '[' )
        { printf ("缺']' ");  Pop(S); }
      else if ( getTop(S) == '(' )
        { printf ("缺')' ");  Pop(S); }
    printf ("\n");  return 0;
  }
}
```

111-52

栈的应用：表达式求值

- 表达式求值是一种典型的栈的应用。
- 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+；

111-53

表达式事例

- 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：
  a) 优先级高的先计算；
  b) 优先级相同的自左向右计算；
  c) 当使用括号时从最内层括号开始计算。
- 当使用中缀表达式计算时，需要同时使用两个栈辅助求值；而使用后缀表达式求值，则只需要一个栈，相对简单一些。

111-54

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。
- 例 a b c d - * + e f ^ g / - （计算顺序见下标）

$$\underbrace{\underbrace{a\ \underbrace{b\ \underbrace{c\ d\ -}_{rst_1}\ *}_{rst_2}\ +}_{rst_3}\ \underbrace{\underbrace{e\ f\ \wedge}_{rst_4}\ g\ /}_{rst_5}\ -}_{rst_6}$$

- 一般表达式的操作符有 4 种类型：
  a) 算术操作符 如双目操作符（+、-、*、/ 和 %）以及单目操作符（-）。
  b) 关系操作符 包括<、<=、==、!=、>=、>。这些操作符主要用于比较。
  c) 逻辑操作符 如与(&&)、或(||)、非(!)。
  d) 括号 ‘(’和 ‘)’ 它们的作用是改变运算顺序。

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：
  a) 若该项是操作数，则将其压栈；
  b) 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。
- 举例 a b c d - * + e f ^ g / -

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	a b
4	c	操作数	进栈	a b c
5	d	操作数	进栈	a b c d
6	-	操作符	d、c 退栈, 计算 c-d, 结果 r_1 进栈	a b r_1
7	*	操作符	r_1、b退栈, 计算b*r_1, 结果r_2进栈	a r_2

8	+	操作符	r_2、a 退栈, 计算 a+r_2, 结果 r_3 进栈	r_3
9	e	操作数	进栈	r_3 e
10	f	操作数	进栈	r_3 e f
11	^	操作符	f、e 退栈, 计算 e^f, 结果 r_4 进栈	r_3 r_4
12	g	操作数	进栈	r_3 r_4 g
13	/	操作符	g、r_4退栈, 计算 r_4/g, 结果 r_5 进栈	r_3 r_5
14	-	操作符	r_5、r_3退栈, 计算r_3-r_5, 结果 r_6进栈	r_6

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。

C/C++中操作符的优先级

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

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。
- 在把中缀表达式转换为后缀表达式的过程中，需要检查算术运算符的优先级，以实现运算规则。

中缀表达式转换为后缀表达式

- 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。
  - a) 若ch是操作数，直接输出，读入下一个字符ch。
  - b) 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。（看后面是否有更高的）

- ➢ 若 icp (ch) < isp (op)，退栈并输出。（执行先前保存在栈内的优先级高的操作符）
- ➢ 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。（销括号）

- 算法结束，输出序列即为所需的后缀表达式。
- 举例，将中缀表达式

  a + b * (c − d) − e / f #

  转换为后缀表达式

  a b c d − * + e f / −

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 输出，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−,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

应用中缀表示计算表达式的值

$$\underbrace{\underbrace{a + \underbrace{b * \underbrace{(\,c - d\,)}_{rst1}}_{rst2}}_{rst3} - \underbrace{e / f}_{rst4}}_{rst5}$$

- 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，仍需要考虑各操作符的优先级，参看前面给出的优先级表。

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. 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。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	A	#		操作数 A 进栈, 读字符
3	+	A	#+	+ > #	操作符 + 进栈, 读字符
4	B	A B	#+		操作数 B 进栈, 读字符
5	*	A B	#+*	* > +	操作符 * 进栈, 读字符
6	(	A B	#+*(	(> *	操作符 (进栈, 读字符
7	C	A B C	#+*(		操作数 C 进栈, 读字符
8	-	A B C	#+*(-	- > (	操作符 - 进栈, 读字符
9	D	A B C D	#+*(-		操作数 D 进栈, 读字符
10	)	AB r_1	#+*(	) < -	D、C、- 退栈，计算 C-D，结果 r_1 进栈

步	输入	OPND	OPTR	语义	动作
10	)	AB r_1	#+*(	) < -	D、C、- 退栈，计算 C-D，结果 r_1 进栈
11		A B r_1	#+*	) = (	(退栈, 消括号, 读字符
12	-	A r_2	#+	- < *	r_1、B、* 退栈, 计算B*r_1, 结果 r_2 进栈
13		r_3	#	- < +	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 E	#-		操作数E进栈, 读字符
16	/	r_3 E	#-/	/ > -	操作符/进栈, 读字符
17	F	r_3 E F	#-/		操作数F进栈, 读字符

步	输入	OPND	OPTR	语义	动作
17	F	r_3 E F	#-/		操作数F进栈, 读字符
18	#	r_3 r_4	#-	# < /	F、E、/退栈, 计算E/F, 结果 r_4 进栈
19		r_5	#	# < -	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，结果在OPND

栈的应用：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(Factorial)

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

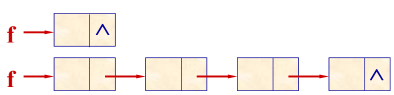
- 求解阶乘函数的递归算法

```
long Factorial ( long n ) {
    if (n < 0) exit(0);
    if (n == 0) return 1;              //递归终止
    else return n*Factorial (n-1);     //减一递归
}
```


求解阶乘 $n!$ 的过程

数据结构是递归的

- 例如，单链表结构



✧ 单链表中一个结点，它的指针域为NULL，其后继是一个链表，是空链表;

✧ 单链表中一个结点，它的指针域非空，其后继仍是一个单链表，是非空单链表。

✧ 从结构上，用指针定义结点，又用结点定义指针。

链表的递归结构定义

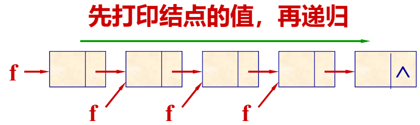
```
typedef struct node {       //单链表定义
    ElemType data;
    struct node *link;      //定义结点用到指针
} LinkNode, *LinkList;      //定义指针用到结点
```

- 基于递归定义的数据结构，相应算法的实现均可采用递归方式。下面举例。

- 在以 f 为表头指针的不带头结点的单链表中正向打印所有结点所存储的数值。

```
void printValue ( LinkNode *f ) {
    if ( f != NULL) {              //递归结束条件
        printf ( f ->data );       //打印当前结点的值
        PrintValue ( f->link );    //递归打印后续链表
    }
}
```

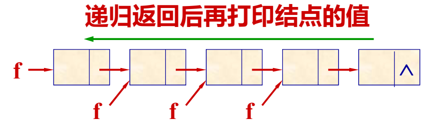
先打印结点的值，再递归



- 在以 f 为表头指针的不带头结点的单链表中反向打印所有结点所存储的数值。

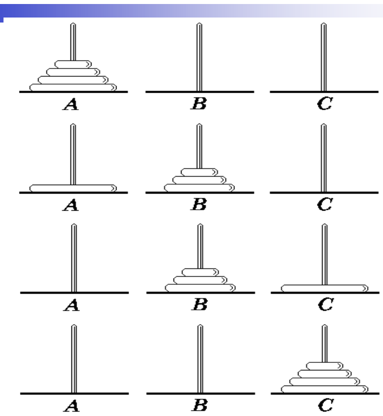
```
void printValue ( LinkNode *f ) {
    if ( f != NULL) {              //递归结束条件
        printValue ( f ->link );   //递归打印后续链表
        printf ( f ->data );       //返回后打印结点值
    }
}
```

递归返回后再打印结点的值

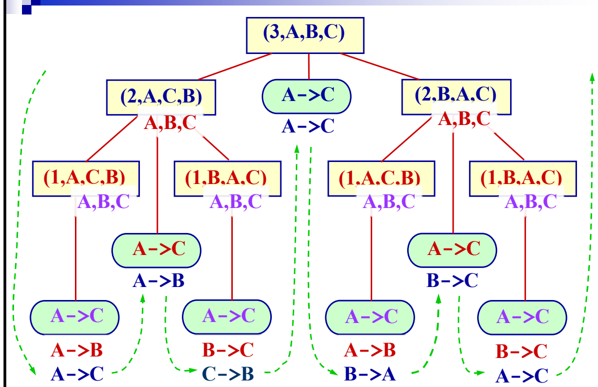


问题的解法是递归的

- 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。
- 这是典型的分治法问题。




```
#include <stdlib.h>
#include "string.h"
void Hanoi ( int n, char A, char B, char C ) {
 //用A、B、C代表三个柱子，算法模拟汉诺塔问题
    if (n == 1) printf ( " move %s", A, " to  %s ", C );
    else {
        Hanoi ( n-1, A, C, B );
        printf ( " move %s", A, " to  %s ", C );
        Hanoi ( n-1, B, A, C );
    }
}
```

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：

递归调用

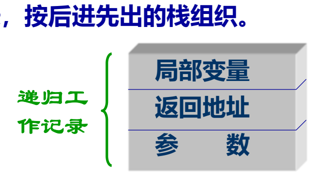
n! ⟹ (n-1)! ⟹ (n-2)! ⟹ ⟹ 1! ⟹ 0!=1

返回次序

- 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。
- 每次调用必须记下返回上层什么地方的地址。

递归工作栈

- 每一次递归调用，需要为过程中使用的参数、局部变量等另外分配存储空间，每个过程的工作空间互不干扰，回到上层还可恢复上层原来的值。
- 每层递归调用需分配的空间形成递归的工作记录，按后进先出的栈组织。



```
        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 ──────┘
        }
```

111-85

计算Fact时活动记录的内容

递归调用序列

参数	返回地址	返回时的指令
4	RetLoc1	RetLoc1 return 4*6 //返回24
3	RetLoc2	RetLoc2 return 3*2 //返回6
2	RetLoc2	RetLoc2 return 2*1 //返回2
1	RetLoc2	RetLoc2 return 1*1 //返回1
0	RetLoc2	RetLoc2 return 1 //返回1

111-86

递归过程改为非递归过程

- 递归过程简洁、易编、易懂。然而，递归过程效率低，重复计算多。
- 例如，定义一个计算斐波那契数列的递归函数Fib(n):

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

如 $F_0 = 0, F_1 = 1, F_2 = 1, F_3 = 2, F_4 = 3, F_5 = 5, F_6 = 8, \ldots$

- 求解斐波那契数列的递归算法为:

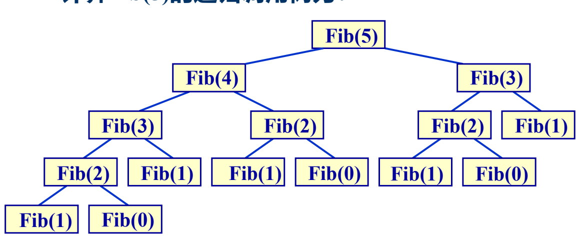
111-87

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

- 递归算法的缺点是重复计算多。从计算斐波那契数列的例子可知，递归调用次数可达
  $\mathrm{NumCall}(k) = 2*F_{k+1} - 1$
- 例如计算Fib(5)，Fib(5)计算 1 次，Fib(4)计算 1 次，Fib(3)计算 2 次，Fib(2)计算 3 次，Fib(1)计算 5 次，Fib(0)计算 3 次。计算次数 = 1 + 1 + 2 + 3 + 5 + 3 = $\mathrm{NumCall}(5) = 2*F_6-1 = 15$。

111-88

- 计算Fib(5)的递归调用树为:



- 递归深度达到 5。
- 为提高算法的计算效率，可以改递归过程为非递归过程。

111-89

- 尾递归和单向递归可直接用迭代实现其非递归过程，其他情形必须借助栈实现非递归过程。

尾递归用迭代法实现

- 例如，一个求阶乘的函数:

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

- 这是典型的尾递归。

111-90

- 程序内只有一个递归语句，且位于程序最后。它不再需要使用返回地址（反正回到上一层的最后），也不需继续使用局部变量。
- 递归函数传递的参数可以作为循环变量，从而把尾递归改为循环，加快了算法的执行速度。
- 求阶乘的非递归算法如下所示。

```
long Fact ( long n ) {        //尾递归改为迭代算法
    long f = 1;
    for ( int i = 2; i <= n; i++ ) f = i*f;
    return f;
}
```

111-91

- 求阶乘的递归算法称为“减治”法，它通过递归逐步降低问题的规模，直到能直接求解。

单向递归用迭代法实现

- 单向递归是指递归过程执行时虽然可能有多个分支，但可以保存前面计算的结果以供后面的语句使用。
- 例如计算斐波那契数列的递归算法，只要保存前两次计算的结果，就可以执行后续的计算。无需使用栈来保存递归工作记录。

111-92

用迭代法实现斐波那契数列的计算

```
long FibIter ( long n ) {
    if ( n <= 1 ) return n;
    long a = 0,  b = 1,  c;
    for ( int i = 2;  i <= n; i++ ) {
        c = a+b;            //求 F_i = F_i-2 + F_i-1
        a = b;   b = c;     //下一个 F_i-2 = 原来的 F_i-1
    }                       //下一个 F_i-1 = 原来的 F_i
    return c;
}
```

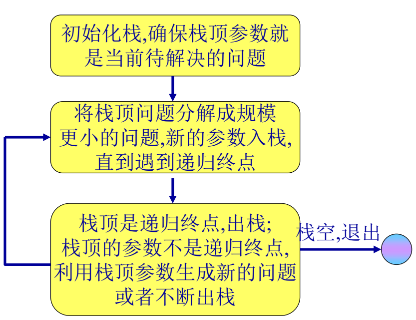
111-93

递归问题的非递归算法编写技巧

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。

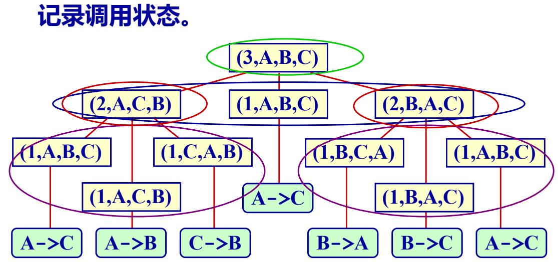
1. 确定入栈的返回信息，这些返回信息在出栈时可以帮助计算函数值或者新的返回信息
2. 确定入栈的条件，即非递归终点的条件
3. 确定出栈时对栈顶的操作，是替换成新的返回信息还是直接出栈
4. 确定算法终止的条件

递归问题的非递归算法编写技巧



用栈将递归算法改为非递归算法

- 以汉诺塔（Hanoi）为例，它有两个内部递归调用的语句，不属于单向递归和尾递归，必须利用栈记录调用状态。



111-96

- **定义栈元素的数据类型**

```
typedef struct {
    int m;  char a, b, c;
} item;
```

- **求解hanoi塔问题**

```
#define stackSize 100
void Hanoi ( int n, char A, char B, char C ) {
   item v, w, S[stackSize];   int top = -1;
   w.m = n;   w.a = A;   w.b = B;   w.c = C;
   S[++top] = w;                //初始布局进栈
```



```
while ( top != -1 ) {        //当栈非空时
   v = S[top--];              //取栈顶布局，退栈
   if ( v.m == 1 ) printf ("Move disk from peg
         %c to %c. \n", v.a, v.c );      //直接搬动
   else {
      w.a = v.b;  w.b = v.a;  w.c = v.c;
      w.m = v.m-1;  S[++top] = w;   //(n-1,B,A,C)
      w.a = v.a;  w.b = v.b;  w.c = v.c;
      w.m = 1;  S[++top] = w;         //(1,A,B,C)
      w.a = v.a;  w.b = v.c;  w.c = v.b;
      w.m = v.m-1;  S[++top] = w;   //(n-1,A,C,B)
```



```
      }  //end else
   }  //end while
}   //Hanoi
```

递归与分治法

- **在使用分而治之策略（即分治法）解决复杂问题时常用的方法即递归的方法。**
- **例如解决汉诺塔问题就属于分治法。所谓分治法就是在解决一个规模比较大的问题时，首先研究问题的结构，把它分解为一个或几个规模比较小的同类型问题，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。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。

- **把复杂化为简单，是分治法的精髓。**

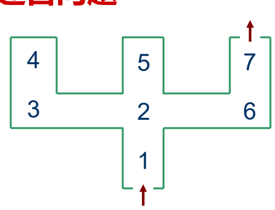
递归与回溯法

- **对一个包含有许多结点，且每个结点有多个分支的问题，可以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，可以沿搜索路径回退到前一结点，沿另一分支继续搜索。如果回退之后没有其他选择，再沿搜索路径回退到更前结点，…。**

- **依次执行，直到搜索到问题的解，或搜索完全部可搜索的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。**

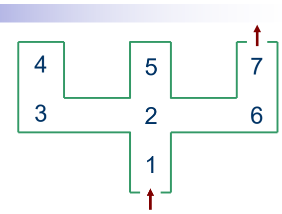
简化的迷宫问题

- **例，一个有 7 个路口的小型迷宫如图。路口 1 是入口，路口 7 是出口。所有路口的前进方向可以用一个前进方向表表示。**



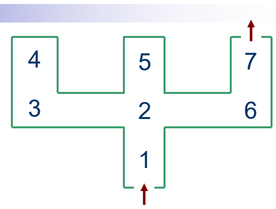
前进方向表

号码	左行	直行	右行
1	0	2	0
2	3	5	6
3	0	0	4
4	0	0	0
5	0	0	0
6	7	0	0
7	0	–	0



- **路口数据 0 表示该方向堵塞不能前进；**
- **向前试探顺序是左行、直行、右行；**
- **前进遇到堵塞则回溯，遇到出口则试探成功。**

迷宫问题求解过程



路口	动作	结果
1 (入口)	左行	堵塞
1	直行	进到 2
2	左行	进到 3
3	左行	堵塞
3	直行	堵塞
3	右行	进到 4
4	左行	堵塞
4	直行	堵塞
4	右行	堵塞(退)
3 (回溯)		(退)
2 (回溯)	直行	进到 5
5	左行	堵塞
5	直行	堵塞
5	右行	堵塞(退)
2 (回溯)	右行	进到 6
6	左行	进到 7
7 (出口)		

111-103

迷宫的结构定义

```
struct Maze {
    int MazeSize;                //迷宫大小（路口数）
    int EXIT;                    //出口号码
    Intersection *intsec;        //路口数组
}
```

前进路口结构定义

```
struct Intersection {
    int left, forwd, right;      //路口信息
}
```

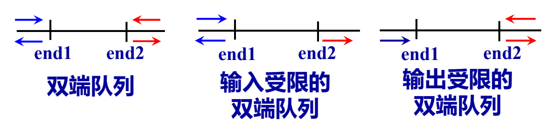
111-104

```
bool Traverse(int Pos) {              //迷宫漫游算法
    if (Pos > 0) {                    //路口从 1 开始
        if (Pos == EXIT)                         //出口
            { printf (Pos << “  ”); return 1; }
回溯    else if ( Traverse(intsec[Pos].left) )   //向左
            { printf (Pos << “  ”); return 1; }
回溯    else if (Traverse(intsec[Pos].forwd))    //向前
            { printf (Pos << “  ”); return 1; }
        else if (Traverse(intsec[Pos].right))    //向右
            { printf (Pos << "  “); return 1; }
    }
    return 0;
}
```

111-105

双端队列

- 双端队列（Deque）是对队列的扩展，它允许在队列的两端进行插入和删除。双端队列英文的全称是Double-ended queue。
- 我们可以把双端队列视为底靠底的双栈，但它们相通，成为双向队列。两端都可能是队头和队尾。



111-106

- 一般地，若有 n 个元素，进入双端队列的顺序是 1, 2, …, n（不管是何端进/出），可用数学归纳法证明：全进全出后可能的出队顺序有 $n!$ 种。
- 而普通的先进先出队列的可能的出队顺序仅有 1 种。
- 假设有 2 个整数 1, 2 顺序进入双端队列，再退出双端队列，则可能的出队序列 2 种，即 1, 2 和 2, 1。因为双端队列可视为底靠底的双栈，也可视为普通的队列。如果输入序列是 1, 2, 3，则 3 可以放在 1, 2 前面，也可以放在 1, 2 中间，还可放在 1, 2 后面，有 3！ 种输出顺序。

111-107

输入受限的双端队列

- 如果限定只能在双端队列的一端输入，可以在两端输出，那么对于一个确定的输入序列，输出只能有 3 种可能：在同一端输出，相当于栈；或者在另一端输出，相当于队列；或者混合进出。
- 假设输入整数是 1, 2, 3，同一端输入／输出，有 5 种（相当于栈），即1 2 3／1 3 2／2 1 3／2 3 1／3 2 1。还剩 3 1 2，它也是合理的输出序列：当1, 2, 3顺序入队之后，3 在同一端出队（相当于栈），1在另一端出队（相当于队列），在最后 2 出队。

111-108

- 如果1, 2, 3, 4 顺序入队，都在同一端出队（相当于栈）有 14 种出队顺序，除此之外还有 4! – 14 = 10 种可能的出队序列。
- 实际上，不合理的出队序列基本上都是以 4 打头的。当 4 先出队时，1, 2, 3 依次排在队列里，2 夹在中间，它不可能在 1 和 3 之前出队，所以不可能的出队序列只有 4 2 3 1和4 2 1 3。

输出受限的双端队列

- 这种双端队列限定只能在队列的一端输出，但可以在两端输入，对于一个确定的输入序列，输出

111-109

顺序也有 3 种可能：在同一端输入和输出，相当于栈；或者在一端输入一端输出，相当于队列；或者混合进出。

- 假设输入整数是 1, 2, 3，同一端输入／输出，有 5 种出队序列，还剩 3, 1, 2。如果限定在左端允许输出，可以在任意端先让 1 入队，再让 2 从右端入队，3 从左端入队，这样 3, 1, 2 也是合理的出队序列。
- 当输入整数是 1, 2, 3, 4 时，同样先排除允许在同一端输入／输出的14种情况，不可能的输出序列还是要在以 4 开头的排列中查找。

111-110

- 在以 4 开头的排列中，有问题的是 4, 1, 2, 3 / 4, 1, 3, 2 / 4, 3, 1, 2 / 4, 2, 1, 3 / 4, 2, 3, 1。
- 对于4, 1, 2, 3，先从右端输入1, 2, 3，再从左端输入4，即可从左端输出4, 1, 2, 3。
- 对于4, 1, 3, 2，必须最后从左端输入 4，在此之前在队列中需得到1, 3, 2的排列，这是不可能的。
- 对于4, 3, 1, 2，先从右端输入1, 2，再从左端输入3, 4，即可从左端输出4, 3, 1, 2。
- 对于4, 2, 1, 3， 先从左端输入1, 2，再从右端输入3，最后从左端输入4，即可从左端输出4, 2, 1, 3。

111-111

- 对于4, 2, 3, 1，同样在 4 输入前，在队列中得不到 2, 3, 1 这种排列。3 不可能夹在 1 和 2 中间，所以这是不可能的输出序列。
- 最后可知，4, 1, 3, 2 和 4, 2, 3, 1 是不可能的出队序列。问题出在 3 不可能在1, 2之间进队。

111-112

优先队列 (Priority Queue)

- 优先队列　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	50	40	10	30
执行顺序	2	5	4	1	3

- 数字越小，优先权越高

111-113

优先队列的定义

```
#define maxPQSize 50
typedef int PQElemType;
typedef struct {
    PQElemType elem[maxPQSize];  //存放数组
    int n;                       //当前元素计数
} PQueue;
```

- 每次在优先队列中插入新元素时，新元素总是插入在队尾；
- 在优先队列中每次从队列中查找权值最小的元素删除，再把队列最后元素填补到被删元素位置。

111-114

```
bool PQInsert ( PQueue& Q, PQElemType x ) {
   if ( Q.n == maxPQSize ) return false;   //队满不插入
   Q.elem[Q.n++] = x;                      //在队尾插入
   return true;
}
bool PQRemove ( PQueue& Q, PQElemType& x ) {
   if ( Q.n == 0 ) return flase;           //队空不能删除
   PQElemType min = Q.elem[0];  int k = 0;
   for ( int i = 1; i < Q.n; i++ )         //查找最小值
      if ( Q.elem[i] < min ) { min = Q.elem[i];  k = i; }
   x = Q.elem[k];  Q.n--;
   Q.elem[k] = Q.elem[Q.n];  return true;
}
```

111-115