第3章 栈和队列

本章学习要点

(1) 掌握栈和队列这两种数据结构的特点，了解在什么问题中应该使用哪种结构。

(2) 熟悉栈(队列)和线性表的关系、顺序栈(顺序队列)和顺序表的关系、链栈(链队列)和链表的关系。

(3) 重点掌握在顺序栈和链栈上实现的栈的7种基本运算，特别注意栈满和栈空的条件及它们的描述。

(4) 重点掌握在循环队列和链队列上实现的7种基本运算，应特别注意队满和队空的描述方法。

(5) 熟悉栈和队列的下溢和上溢的概念；顺序队列中产生假上溢的原因；循环队列消除假上溢的方法。

(6) 了解递归算法执行过程中工作记录的变化情况。

栈和队列与线性表有着密切的联系，一方面，栈和队列的逻辑结构也是线性结构；另一方面，栈和队列的基本操作是线性表操作的子集，因此，可将栈和队列看成两种特殊的线性表。

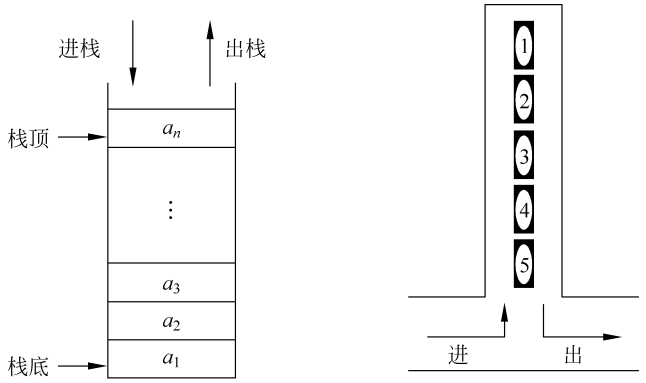
3.1 栈

3.1.1 栈的基本概念

日常生活中有不少类似于栈(如图3.1(a)所示)的例子。假设有一个很窄的死胡同，其宽度只能容纳一辆车，现有5辆车，分别编号为①～⑤，按编号顺序依次进入此胡同，若要退出④，必须先退出⑤；若要退出①必须将⑤、④、③、②依次都退出才行。这个死胡同就是一个栈，如图3.1(b)所示。

栈(stack)是允许仅在表的一端进行插入和删除操作的线性表。允许进行插入和删除的一端称为栈顶(top)，不允许插入和删除的一端称为栈底(bottom)。不含元素的空表称为空栈。

假设栈 $S=(a_1,a_2,\cdots,a_n)$，如图3.1(a)所示，a_1 为栈底元素，a_n 为栈顶元素。栈中元素按 $a_1,a_2,\cdots,a_n$ 的次序进栈，退栈的第一个元素应为栈顶元素。也就是说，栈的特点是后进先出(last in first out，LIFO)，因此，栈又称为后进先出的线性表，简称LIFO线性表。



(a) 栈的示例　　(b) 胡同及车示例

图 3.1　栈及其示例

在已知栈的逻辑结构和确定常用运算后,就可以定义栈的抽象数据类型。

ADT3.1 是栈的抽象数据类型描述,其中包含最常见的栈运算。

ADT3.1 栈 ADT

ADT stack{

数据对象:

$D=\{a_i \mid a_i \in$ 元素集合,$i=1,2,\cdots,n,n\geqslant 0\}$

数据关系:

$R_1=\{\langle a_{i-1},a_i\rangle \mid a_{i-1},a_i\in D,i=2,\cdots,n\}$,约定 a_n 端为栈顶,a_1 为栈底。

基本操作:

InitStack():创建一个空栈。

Destroy():撤销一个栈。

StackEmpty():若栈空,则返回 1,否则返回 0。

StackFull():若栈满,则返回 1,否则返回 0。

Top(x):在 x 中返回栈顶元素。若操作成功,则返回 1,否则返回 0。

Push():在栈顶插入元素 x(入栈)。若操作成功,则返回 1,否则返回 0。

Pop():从栈中删除栈顶元素(出栈)。若操作成功,则返回 1,否则返回 0。

Clear():清除栈中全部元素。

}ADT stack

栈的应用非常广泛,例如进位记数制之间的转换问题。在将十进制数转换成二进制数时,常采用除法。用初始十进制数除以 2,把余数记录下来,若商不为 0,则再用商去除以 2,直到商为 0,这时把所有的余数按出现的逆序排列起来(先出现的余数排在后面,后出现的余数排在前面)就得到了相应的二进制数。例如把十进制数 35 转换成二进制数的过程如图 3.2 所示。

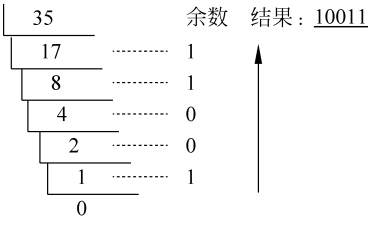


图 3.2　十进制数 35 转换成二进制数的过程

根据上述操作的描述,可以采用一个栈来保存所有

的余数，当商为 0 时让栈中的所有余数出栈，这样就得到了正确的二进制数。

算法 3.1　十进制数转换成二进制数算法。

```
void conversion()
{
    Stack S;
    int n;
    InitStack(&S);
    printf("Input a number to convert:\n");
    scanf("%d",&n);
    if(n<0)
    {
        printf("\nThe number must be over 0.");
        return 0;
    }
    if(n==0) Push(&S,0);
    while(n!=0)
    {
        Push(&S,n%2);
        n=n/2;
    }
    printf("the result is: ");
    while(!StackEmpty(&S))
    {
        printf("%d", Pop(&S));
    }
}
```

3.1.2　栈的顺序存储结构

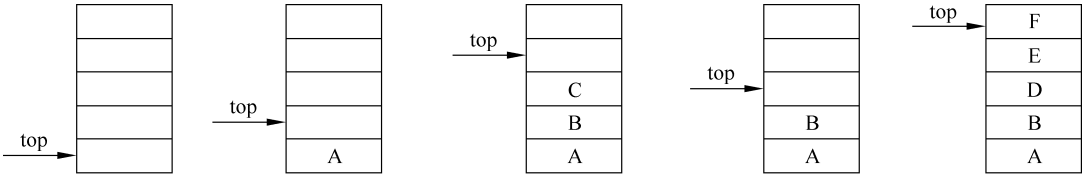
由栈的定义以及栈的实例，很容易想到用数组或类似的结构去存储它。栈的顺序存储结构简称顺序栈(sequential stack)。顺序栈利用一组地址连续的存储单元依次存放从栈底到栈顶的数据元素，通常用一维数组存放栈的元素，同时设“指针”top 指示栈顶元素的当前位置。注意，top 并不是指针型变量，只是整型变量，它指示栈顶元素在数组中的位置，空栈的 top 值为零。

在 C 语言中，顺序栈的类型说明如下：

```
#define maxsize <栈可能的最大数据元素的数目>          //栈的最大容量
typedef struct
{
    elemtype elem[maxsize];
    int top;
}sqstacktp;
```

设 s 为 sqstacktp 型变量，即 s 表示一个顺序栈。图 3.3 说明了这个顺序栈的几种状态。其中：图 3.3(a)表示顺序栈为空，s.top＝0；图 3.3(b)表示栈中只含一个元素 A，s.top＝1，在图 3.3(a)的基础上用进栈操作 Push(s,A)可以得到这种状态；图 3.3(c)表示在图 3.3(b)的基础上将元素 B、C 依次进栈后的状态，s.top＝3；图 3.3(d)表示在图 3.3(c)

状态下将元素 C 退栈后的情况,s.top=2,由执行一次 Pop(s)得到;图 3.3(e)表示栈中有 5 个元素,s.top=maxsize,这种状态称为栈满,此时若有元素进栈则将产生“数组越界”的错误,称为栈溢出。



(a) 栈空　(b) 入栈1个元素的栈　(c) 入栈2个元素的栈　(d) 出栈1个元素　(e) 栈满

图 3.3　顺序栈的几种状态

因此,s.top=0 表示空栈,出栈和读栈顶元素之前应判断栈是否为空。s.top=maxsize 表示栈满,进栈之前应判断是否栈满。

下面讨论在顺序栈上实现的操作。

1. 初始化(栈置空)操作

算法 3.2　顺序栈置空算法。

```
void InitStack(sqstacktp * s)
{
    //将顺序栈 s 置为空
    s->top = 0;
}
```

如下函数生成了一个顺序栈,并完成了对该顺序栈的初始化。

```
void main()
{
    void InitStack(sqstacktp * s);
    sqstacktp * s;
    s = (sqstacktp * )malloc(sizeof(sqstacktp));
    InitStack(s);
}
```

2. 判栈空操作

算法 3.3　顺序栈判栈空算法。

```
int StackEmpty(sqstacktp * s)
{
    if(s->top>0)
        return 0;
    else
        return 1;
}
```

3. 进栈操作

算法 3.4　顺序栈进栈算法。

```
void Push(sqstacktp * s,elemtype x)
{
    //若栈 s 未满,将元素 x 压入栈中;否则,栈的状态不变并给出出错信息
    if(s->top == maxsize)
         printf("Overflow");
    else
         s->elem[s->top++] = x;            //x 进栈
}
```

4. 出栈操作

算法 3.5　顺序栈出栈算法。

```
elemtype Pop(sqstacktp * s)
{
    //若栈 s 不空,则删去栈顶元素并返回元素值,否则返回空元素 NULL
    if(s->top == 0)
        return NULL;
    else
    {
        s->top--;                          //栈顶指针减 1
        return s->elem[s->top];            //返回原栈顶元素值
    }
}
```

5. 求栈深操作

算法 3.6　顺序栈求栈深算法。

```
int Size(sqstacktp * s)
{
    return(s->top);
}
```

6. 读取栈顶元素操作

算法 3.7　顺序栈读取栈顶元素算法。

```
elemtype Top(sqstacktp * s)
{
    if(s->top == 0)
        return NULL;
    else
        return(s->elem[s->top-1]);
}
```

顺序栈使用起来比较简单,但是必须预先为它分配存储空间。而且,为了避免栈溢出,通常必须分配较大的存储空间。显然,这样操作在大多数时候都会造成存储空间的浪费。但是当在程序中同时使用两个栈时,有一种方法可以提高存储空间的使用效率:将两个栈的栈底设在一维数组空间的两端,让两个栈各自向中间延伸,仅当两个栈的栈顶相遇时才可能发生上溢,如图 3.4 所示。这样当一个栈里的元素较多,超过向量空间的一半时,只要另一个栈的元素不多,那么前者就可以占用后者的部分存储空间。所以,当两个栈共享一个长度为 maxsize 的数组空间时,每个栈实际可利用的最大空间大于 maxsize/2。

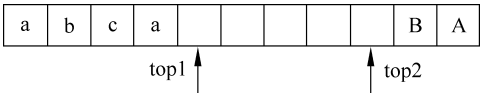


图 3.4 两个栈共享空间示意图

共享空间的双栈结构的类型描述如下:

```
#define maxsize <栈可能的最大数据元素的数目>          //栈的最大容量
typedef struct
{
  elemtype elem[MAXSIZE];
  int top[2];
}dustacktp
```

若 ds 为 dustacktp 型变量,显然:

(1) 栈 1 的顶由 ds.top[0]指示,ds.top[0]=0 表示栈 1 为空。

(2) 栈 2 的顶由 ds.top[1]指示,ds.top[1]=maxsize-1 表示栈 2 为空。

(3) ds.top[0]+1=ds.top[1]表示栈满。

3.1.3 栈的链式存储结构

由栈的顺序存储结构可知,顺序栈的最大缺点是:为了保证不溢出,必须预先为栈分配一个较大的空间,这很有可能造成存储空间的浪费,而且在很多时候并不能保证所分配的空间足够使用。这些缺陷大大降低了顺序栈的可用性,这时可以考虑采用栈的链式存储结构。

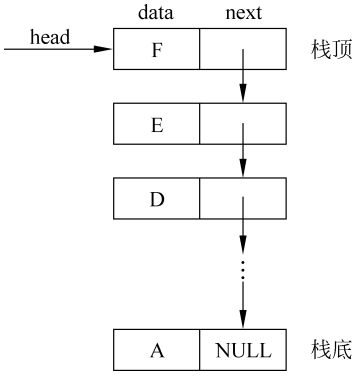


图 3.5 链栈示意图

栈的链式存储结构简称链栈(linked stack),其组织形式与单链表类似,链表的尾部结点是栈底,链表的头部结点是栈顶。由于只在链栈的头部进行操作,故链栈没有必要设置头结点。如图 3.5 所示,其中,单链表的头指针 head 作为栈顶指针。链栈由栈顶指针 head 唯一确定,栈底结点的 next 域为 NULL。

链栈的类型定义如下:

```
typedef struct stacknode
{
    elemtype data;
    struct stacknode * next;
}stacknode;
typedef struct
```

```
{
    stacknode * top;                    //栈顶指针
}LinkStack;
```

下面给出初始化、进栈和出栈操作在链栈上的实现。

1. 初始化操作

算法 3.8　链栈初始化算法。

```
void InitStack(LinkStack * ls)
{
    //建立一个空栈 ls
    ls -> top = NULL;
}
```

下面函数完成了对一个链栈的创建和初始化操作。

```
void main()
{
    LinkStack * ls;
    ls = (LinkStack * )malloc(sizeof(LinkStack));
    InitStack(ls);
}
```

2. 进栈操作

算法 3.9　链栈进栈算法。

```
void Push(LinkStack * ls,elemtype x)
{
    stacknode * s = NULL;
    s = (stacknode * )malloc(sizeof(stacknode));        //生成新结点
    s -> data = x;
    s -> next = ls -> top;                              //链入新结点
    ls -> top = s;                                      //修改栈顶指针
}
```

3. 出栈操作

算法 3.10　链栈出栈算法。

```
elemtype Pop(LinkStack * ls)
{
    //若栈 ls 不空,删去栈顶元素并返回元素值,否则返回空元素 NULL
    stacknode * p = NULL;
    elemtype x;
    if(ls -> top == NULL)
        return NULL;
    else
    {
        x = (ls -> top) -> data;
```

```
        p = ls -> top;
        ls -> top = p -> next;
        free(p);
        return x;                                    //返回原栈顶元素值
    }
}
```

显然,链栈一般不会发生栈满,只有当整个可用空间都被占满,malloc()函数过程无法实现时才可能发生上溢。

另外,链栈占用的空间是动态分配的,所以多个链栈可以共享存储区域。

3.2 栈的应用实例

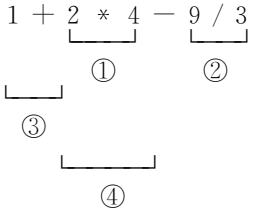
栈在计算机科学领域具有广泛的应用,只要问题满足后进先出原则,均可使用栈作为其数据结构。例如,在编译和运行计算机高级语言程序的过程中,需要利用栈进行语法检查(如检查 PASCAL 语言中 begin 和 end、(和)、[和]是否配对等);实现递归过程和函数的调用、计算表达式的值时需要利用栈实现其功能。

3.2.1 表达式求值

要将一个表达式翻译成能正确求值的机器指令,或者直接对表达式求值,首先要能正确解释表达式。例如,对下面的算术表达式求值

$$1+2*4-9/3$$

必须遵循先乘除后加减、先左后右及先括号内后括号外的四则运算法则,其计算顺序应为



那么,如何让机器也按照这样的规则求值呢?通常采用“运算符优先数法”。

一般表达式中会遇到操作数、运算符和表达式结束符(为了简化问题,这里仅讨论只含加、减、乘、除 4 种运算,并且不含括号的情况)。对每种运算符赋予一个优先数,如表 3.1 所示,其中 # 是表达式结束符。

表 3.1 运算符的优先数

运算符	*	/	+	-	#
优先数	2	2	1	1	0

对表达式求值时,一般设立两个栈:一个称为运算符栈(OPTR);另一个称为操作数栈(OPND),以便分别存放表达式中的运算符和操作数。

具体处理方法是:从左至右扫描表达式。

(1) 凡遇操作数,一律进 OPND 栈。

(2) 若遇运算符,则比较它与 OPTR 栈的栈顶元素的优先数。若它的优先数大,则将该运算符进 OPTR 栈; 反之,则弹出 OPTR 栈顶的运算符 θ,并从 OPND 栈连续弹出两个栈顶元素 y 和 x,进行运算 xθy,并将运算结果压进 OPND 栈。

为了使算法简捷,在表达式的最左边也虚设一个 #,一旦左边的 # 与右边的 # 相遇,说明表达式求值结束。

算法 3.11　算术表达式求值算法。

```
int precedence(char ch)          //求运算符优先数
{
    int z = 0;
    switch (ch)
    {
        '+':z = 1;break;
        '-':z = 1;break;
        '*':z = 2;break;
        '/':z = 2;break;
        '#':z = 0;break;
    default:printf("error!\n");
    }
    return z;
}
int operate(int x,char ch,int y)  //进行二元运算 xθy
{
    int z = 0;
    switch (ch)
    {
        '+':z = x + y;break;
        '-':z = x - y;break;
        '*':z = x * y;break;
        '/':z = x/y;break;
    default:printf("error!\n");
    }
    return z;
}
operandtype exp_reduced()        //算术表达式求值的运算符优先数算法,假定表达式无语法错误
{
    char ch,theta;
    operandtype x,y,result;
    strcpy(op," +- * / # ");     //op 为运算符的集合
    InitStack(OPTR);
    Push(OPTR,'#');              //栈初始化,并在运算符栈的栈底压入表达式左边虚设的字符"#"
    InitStack(OPND);
    scanf(" %c",&ch);            //从终端读入一个字符
    while(ch!= '#' || Top(OPTR)!= '#')
    {
        if(!strchr(op,ch))
        {
          Push(OPND,ch);
```

```
            ch = getchar();
        }
        else if(precedence(ch)> precedence(Top(OPTR)))        //比较优先数
            {
                Push(OPTR,ch);
                ch = getchar();
            }
            else
            {
                theta = Pop(OPTR);                            //弹出栈顶运算符
                y = Pop(OPND),x = Pop(OPND);                  //连续弹出两个操作数
                result = operate(x,theta,y);                  //进行运算 xθy
                Push(OPND,result);                            //将运算结果压入操作数栈
            }
    }
    return(Top(OPND));                                        //从操作数栈顶取出表达式运算结果返回
}
```

上述算法中使用了有关栈的基本操作的若干函数,另外,还调用了两个函数,其中precedence(w)是求运算符优先数的函数,operate(x,theta,y)是进行二元运算 xθy 的函数。

例 3.1 利用算法 3.11,写出对算术表达式 1+2*4-9/3 求值的操作过程。

利用算法 3.11 对算术表达式 1+2*4-9/3 求值的操作过程如图 3.6 所示。

步骤	OPTR 栈	OPND 栈	输入字符	主要操作
1	#		1+2*4-9/3#	Push(OPND,'1')
2	#	1	+2*4-9/3#	Push(OPTR,'+')
3	# +	1	2*4-9/3#	Push(OPND,'2')
4	# +	1 2	*4-9/3#	Push(OPTR,'*')
5	# + *	1 2	4-9/3#	Push(OPND,'4')
6	# + *	1 2 4	-9/3#	operate('2','*','4')
7	# +	1 8	-9/3#	operate('1','+','8')
8	#	9	-9/3#	Push(OPTR,'-')
9	# -	9	9/3#	Push(OPND,'9')
10	# -	9 9	/3#	Push(OPTR,'/')
11	# - /	9 9	3#	Push(OPND,'3')
12	# - /	9 9 3	#	operate('9','/','3')
13	# -	9 3	#	operate('9','-','3')
14	#	6	#	return(Top(OPND))

图 3.6 算术表达式 1+2*4-9/3 求值的操作过程

3.2.2 栈与函数调用

在模块化程序设计的思想中,模块(或函数、过程)是功能相对独立的一个程序段,在主函数(主程序)中调用模块来解决复杂的实际问题。由于函数调用后,需要返回调用处,所以在调用时,需要用栈记录断点的地址以及有关信息,以便返回。

函数调用的执行过程如图 3.7 所示。

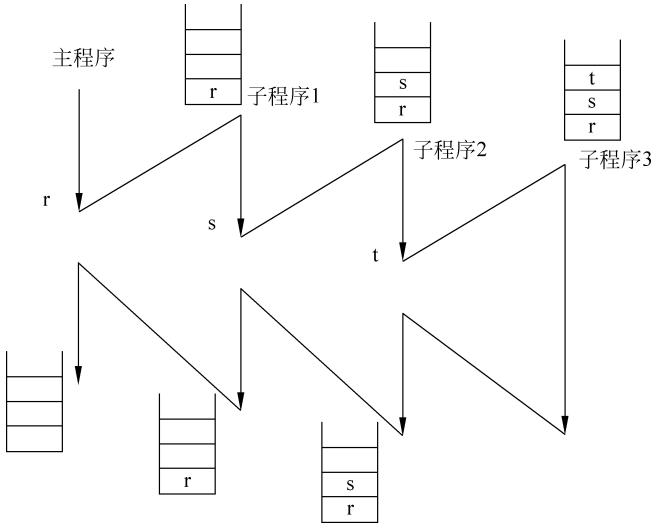


图 3.7　函数调用的执行过程

递归调用是一种特殊的函数调用，关于栈在递归调用中的应用详见 3.5 节。

3.2.3　栈在回溯法中的应用

在某些问题的求解过程中常常采用试探方法，当某一路径受阻时，需要逆序退回，重新选择新路径，这样必须用栈记录曾经到达的每一状态，栈顶状态即是回退的第一站，例如迷宫问题、地图四染色问题等。下面讨论地图四染色问题。

四染色定理：可以用不多于四种颜色对地图着色，使相邻的地区不重色。

算法思想：回溯法。

从第一号地区开始逐一染色，每一个地区逐次用色号 1、2、3、4 进行试探，若当前所取的色号与周围已染色的地区不重色，则用栈记下该地区的色号，否则依次用下一色号进行试探；若试探 4 种颜色均与相邻地区发生重色，则需退栈回溯，修改当前栈顶的色号。

数据结构：

r[n][n]：n * n 的关系矩阵，r[i][j]＝0 表示 i 号地区与 j 号地区不相邻，r[i][j]＝1 表示 i 号地区与 j 号地区相邻。

s[n]：栈的顺序存储，s[i]表示第 i 号地区的染色号。

在下述算法中，n 代表地区号，为方便描述，r 和 s 分别定义如下：

```
int r[n+1][n+1];
int s[n+1];
```

算法 3.12　地图四染色算法。

```
void mapcolor(int r[][n+1],int n,int s[])                    //n 为地区号
{
    int i,j,k;
    s[1]=1;                                                  //1 号地区染 1 色
    i=2;
```

```
    j = 1;                                                    //i 为地区号,j 为染色号
    while (i < = n)
{
    while ((j < = 4)&&(i < = n))
    {
        k = 1;                                                //k 指示一个染色地区号
        while((k < i)&&(s[k] * r[i][k]!= j))
            k++;
        if (k < i)
            j++;                                              //用 j + 1 色号继续试探
        else
            s[i] = j,i++,j = 1;
    //若不与相邻地区重色,进栈记录染色结果,继续对下一地区从 1 色号起试探
    }
    if(j > 4)
        i--,j = s[i] + 1;                                     //改变栈顶地区的色号
}
}
```

3.3 队列

日常生活中,队列的例子比比皆是,如等待购物的顾客总是按先来后到的次序排成队列,排在队头的人先得到服务,后到的人总是排在队列的末尾。在计算机系统中,队列的应用例子也很多,例如,操作系统中的作业排队。在允许多道程序运行的计算机中,同时有几个作业运行,如果运行的结果都要输出,则要按请求输出的先后次序排队。每当通道传输完毕,可以接受新的输出任务时,就从等待输出的队列中取出队头的作业进行输出。凡是申请输出的作业都从队尾进入队列。

3.3.1 队列的基本概念

队列(queue)是只允许在表的一端进行插入,在表的另一端进行删除的线性表。允许插入的一端叫作队尾(rear),允许删除的一端称为队头(front),不含元素的队列称为空队列。

假设队列为 $q=(a,b,c,\cdots,h,i,g)$,如图 3.8 所示,a 是队头元素,g 则是队尾元素。队列中的元素是按照 a,b,c,…,h,i,g 的顺序进入的,退出队列也只能按照这个次序依次退出,也就是说,只有在 a,b 离队后,c 才能退出队列,同理 a,b,c,…,h,i 都离队之后,g 才能退出队列。因此,队列的特点是先进先出(First In First Out,FIFO),队列又称为先进先出的线性表,简称 FIFO 表。

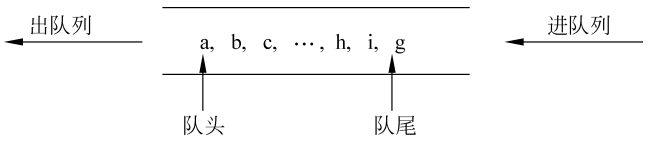


图 3.8 队列的示意图

在已知队列的逻辑结构和确定常用运算后就可以定义队列的抽象数据类型。ADT3.2是队列的抽象数据类型描述，其中只包含最常见的队列运算。

ADT3.2 队列 ADT

ADT queue{

数据对象：

$D=\{a_i \mid a_i \in$ 元素集合，$i=1,2,\cdots,n,n\geqslant 0\}$

数据关系：

$R_1=\{\langle a_{i-1},a_i\rangle \mid a_{i-1},a_i \in D,i=2,\cdots,n\}$，约定 a_1 为队列头，a_n 端为队列尾。

基本操作：

InitQueue()：创建一个空队列。

Destroy()：撤销一个队列。

QueueEmpty()：若队列空，则返回1，否则返回0。

QueueFull()：若队列满，则返回1，否则返回0。

Front(x)：在x中返回队列头元素。若操作成功，则返回1，否则返回0。

EnQueue()：在队列尾插入元素x(入队)。若操作成功，则返回1，否则返回0。

DelQueue()：从队列中删除队列头元素(出队)。若操作成功，则返回1，否则返回0。

Clear()：清除队列中全部元素。

}ADT queue

3.3.2　队列的顺序存储结构

队列的顺序存储结构，简称顺序队列(sequential queue)，它由一个存放队列元素的一维数组，以及分别指示队头和队尾的“指针”所组成。通常约定：队尾指针指示队尾元素在一维数组中的当前位置，队头指针指示队头元素在一维数组中当前位置的前一个位置，如图3.9所示。

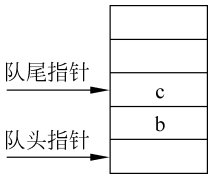


图3.9　顺序队列示意图

顺序队列的类型定义如下：

```
#define maxsize <队列可能的最大长度>              //maxsize为队列可能达到的最大长度
typedef struct
{
    elemtype elem[maxsize];
    int front,rear;
}squeuetp;
```

假设Sq为squeuetp变量，即Sq表示一个顺序队列，入队操作为：

```
Sq.rear = Sq.rear + 1;                          //修改队尾指针 rear
Sq.elem[Sq.rear] = x;                           //将 x 放入 rear 所指位置
```

类似地，出队列需要修改队头指针：

```
Sq.front = Sq.front + 1
```

图3.10说明了在顺序队列上按上述方法入队、出队的几种状态。

图 3.10(a)为空队列,Sq.rear=-1,Sq.front=-1;

图 3.10(b)中 a,b,c,d,e 依次入队后,Sq.rear=4,Sq.front=-1;

图 3.10(c)中 a,b,c 依次出队后,Sq.rear=4,Sq.front=2。

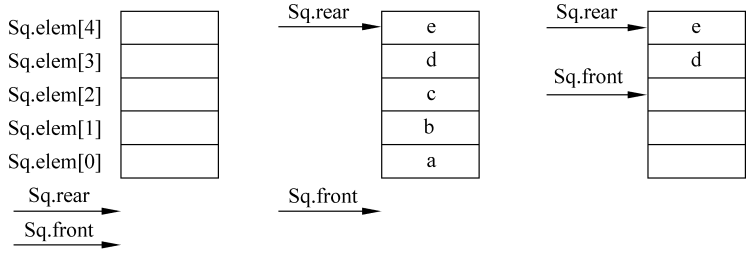


图 3.10 顺序队列的几种状态

由此可见,Sq.front=Sq.rear 表示队列空。但是队列满的条件是什么呢?

在图 3.10(b)和图 3.10(c)状态下,Sq.rear=maxsize,显然按上述方法不能再做入队操作。然而在图 3.10(c)状态下顺序队列的存储空间并没有被占满,因此这是一种假溢出现象。

为了克服假溢出现象,一个巧妙的办法是把队列设想为一个循环的表,设想 Sq.elem[0]接在 Sq.elem[maxsize-1]之后。这种存储结构称为循环队列。利用取余运算(%),很容易实现队头、队尾指针在循环意义下的加 1 操作。循环队列示意图如图 3.11 所示。

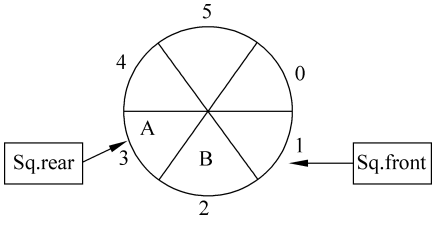


图 3.11 循环队列示意图

循环队列上的入队操作为:

```
Sq.rear = (Sq.rear + 1) % maxsize;
Sq.elem[Sq.rear] = x;
```

出队操作为:

```
Sq.front = (Sq.front + 1) % maxsize
```

图 3.12 说明了在循环队列上入队、出队的几种状态,其中:

图 3.12(a)为空队列,Sq.rear=0,Sq.front=0;

图 3.12(b)中 A,B,C,D 依次入队后,Sq.rear=4,Sq.front=0;

图 3.12(c)中 A,B,C,D 依次出队后,Sq.rear=4,Sq.front=4;

图 3.12(d)中 E 入队后,Sq.rear=5,Sq.front=4;

图 3.12(e)中 F 入队,Sq.rear=(Sq.rear+1)%maxsize=(5+1)%6=0,Sq.front=4;

图 3.12(f)中 G,H,I,J 依次入队后,Sq.rear=4,Sq.front=4。

在图 3.12(a)和图 3.12(c)的状态下,队列空,Sq.front=Sq.rear;在图 3.12(f)的状态下,队列满,Sq.front=Sq.rear。由此可见,不能只凭等式 Sq.front=Sq.rear 来判定循环队列的状态是空还是满。为此,有两种处理方法:第一种,另设一个标志位以区别队列是空还是满;第二种,约定队头指针指示的位置不用来存放元素,这样当队尾指针“绕一圈”后追上

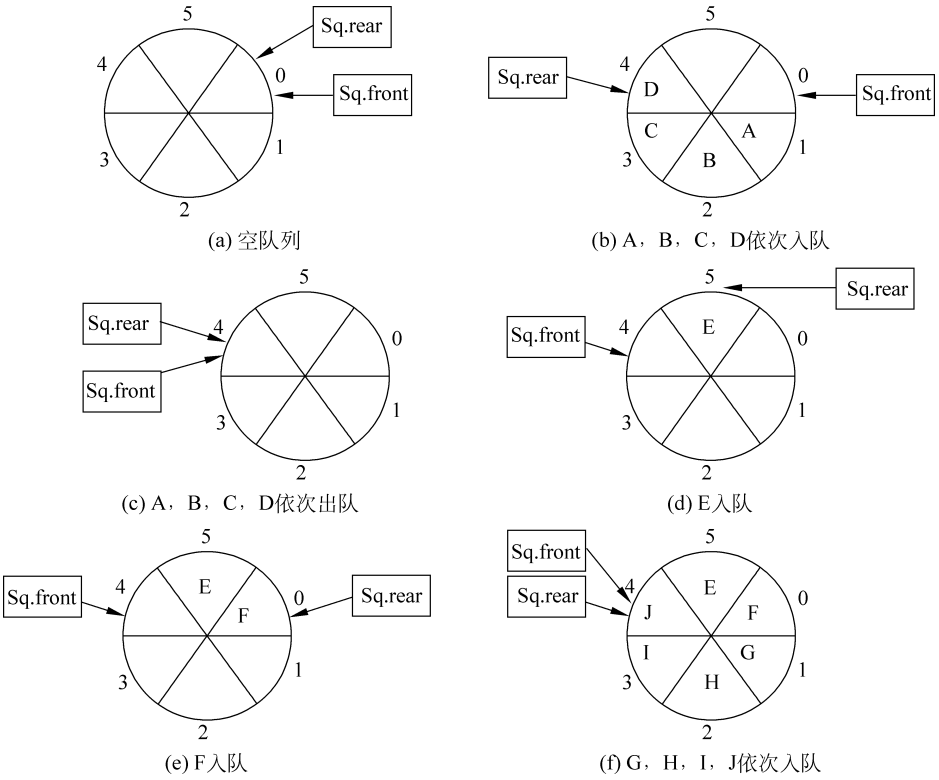


图 3.12　循环队列的几种状态

队头指针时，视为队满。故队满的条件为：

```
(Sq.rear + 1) % maxsize = Sq.front
```

显然，队空的条件为：

```
Sq.front = Sq.rear
```

在此采用第二种方法。因此，循环队列的定义为：

```
#define maxsize <为队列可能达到的最大长度> + 1
typedef struct
{
    elemtype elem[maxsize];
    int front,rear;
}cqueuetp;
```

下面给出在循环队列上实现的操作。

1. 队列的初始化操作

算法 3.13　循环队列初始化算法。

```
void InitQueue(cqueuetp * sq)
{
    //设置 sq 为空的循环队列
```

```
    sq->front = 0;
    sq->rear = 0;
}
```

下面的函数实现了对一个循环队列的定义和初始化操作。

```
void main()
{
    cqueuetp *sq;
    sq = (cqueuetp *)malloc(sizeof(cqueuetp));
    InitQueue(sq);
}
```

2. 判队空操作

算法 3.14 循环队列判队空算法。

```
int QueueEmpty(cqueuetp *sq)
{
    //判别队列 sq 是否为空;若为空则返回真值,否则返回假值
    if (sq->rear == sq->front)
        return 1;
    return 0;
}
```

3. 求队长度操作

算法 3.15 求循环队列长度算法。

```
int Size(cqueuetp *sq)
{
    //取模运算的被除数加上 maxsize 是考虑 sq->rear - sq->front < 0 的情况
    return((maxsize + sq->rear - sq->front) % maxsize);
}
```

4. 读队头元素操作

算法 3.16 读循环队列队头元素算法。

```
elemtype Head(cqueuetp *sq)
{
    //若循环队列 sq 不空,则返回队头元素值,否则返回空元素 NULL
    if(sq->front == sq->rear)
        return NULL;
    else
        return(sq->elem[(sq->front + 1) % maxsize]);
}
```

5. 入队操作

算法 3.17 循环队列入队算法。

```
void EnQueue(cqueuetp *sq,elemtype x)
```

```
{
   //若循环队列 sq 未满,插入 x 为新的队尾元素,否则队列状态不变并给出错误信息
    if ((sq->rear + 1) % maxsize == sq->front)
        printf("Overflow");
    else
    {
        sq->rear = (sq->rear + 1) % maxsize;
        sq->elem[sq->rear] = x;
    }
}
```

6. 出队操作

算法 3.18　循环队列出队算法。

```
elemtype DelQueue(cqueuetp * sq)
{
//若循环队列 sq 不空,则删去队头元素并返回元素值,否则返回空元素 NULL
if(sq->front == sq->rear)
    return NULL;
else
{
    sq->front = (sq->front + 1) % maxsize;
    return(sq->elem[sq->front]);
}
}
```

3.3.3　队列的链式存储结构

与栈的顺序存储一样,队列的顺序存储也存在溢出的情况,因此可以考虑采用队列的链式存储结构。

队列的链式存储结构简称链队列,它实际上是一个同时带有头指针和尾指针的单链表,头指针指向队头结点,尾指针指向队尾结点,如图 3.13 所示。虽然用头指针就可以唯一确定这个单链表,但是插入操作总是在队尾进行,如果没有尾指针,入队操作的时间复杂度将由 $O(1)$升到 $O(n)$。

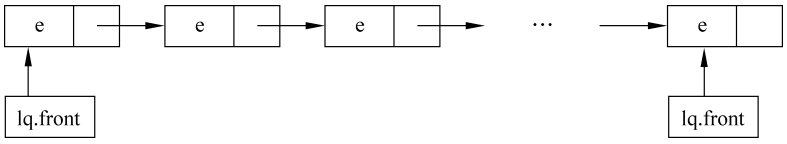


图 3.13　链队列示意图

链队列的类型定义如下:

```
typedef struct NODETYPE
{
    elemtype data;
    struct NODETYPE * next;
```

```
}nodetype;
typedef struct
{
    nodetype * front;
    nodetype * rear;
}lqueuetp;
```

其中,头、尾指针被封装在一起,将链队列的类型 lqueuetp 定义为一个结构类型,若做如下操作:

```
lqueuetp  lq;
```

则得到一个链队列变量,其好处是使得对 lq 的操作,仅涉及结构变量 lq。

由图 3.13 容易看出,链队列空的判断条件为:

```
lq.front = lq.rear
```

下面给出在链队列上实现的操作。

1. 初始化操作

算法 3.19 链队列初始化算法。

```
void InitQueue(lqueuetp * lq)
{
    //设置一个空的链队列 lq
    lq->front = (nodetype * )malloc(sizeof(nodetype));
    lq->front->next = NULL;
    lq->rear = lq->front;
}
```

如下函数实现了对一个链队列的定义以及初始化操作。

```
void main()
{
    lqueuetp * lq;
    lq = (lqueuetp * )malloc(sizeof(lqueuetp));
    InitQueue(lq);
}
```

2. 判队空操作

算法 3.20 链队列判队空算法。

```
int QueueEmpty(lqueuetp * lq)
{
    if (lq->front == lq->rear)
        return 1;
    return 0;
}
```

3. 求队长度操作

算法 3.21 求链队列长度算法。

```
int Size(lqueuetp * lq)
{
   //返回队列中元素个数
    int i = 0;
    nodetype * p = lq -> front -> next;
    while(p)
    {
        i++;
        p = p -> next;
    }
    return i;
}
```

4. 读队头元素操作

算法 3.22 读链队列队头元素算法。

```
elemtype Head(lqueuetp * lq)
{
    //若链队列 lq 不空,则返回队头元素值,否则返回空元素 NULL
    if(lq -> front == lq -> rear)
        return NULL;
    else
        return (lq -> front -> next -> data);
}
```

5. 入队操作

算法 3.23 链队列入队算法。

```
void EnQueue(lqueuetp * lq,elemtype x)
{
    //在链队列 lq 中,插入 x 为新的队尾元素
    nodetype * s;
    s = (nodetype * )malloc(sizeof(nodetype));
    s -> data = x;
    s -> next = NULL;
    lq -> rear -> next = s;
    lq -> rear = s;
}
```

6. 出队操作

算法 3.24 链队列出队算法。

```
elemtype DelQueue(lqueuetp * lq)
```

```
{
    //若链队列 lq 不空,则删去队头元素并返回元素值,否则返回空元素 NULL
    elemtype x;
    nodetype * p;
    if(lq -> front == lq -> rear)
        return NULL;
    else
    {
        p = lq -> front -> next;
        lq -> front -> next = p -> next;
        if(p -> next == NULL)
           lq -> rear = lq -> front;      //当链队列中仅有一个结点时,出队时还需修改尾指针
        x = p -> data;
        free(p);
        return x;
    }
}
```

注意:

(1) 链队列和链栈类似,无须判队满操作。

(2) 在出队算法中,当原队中只有一个结点时,该结点既是队头也是队尾,故删去此结点时也需修改尾指针,且删去此结点后队列变空。

(3) 和链栈情况相同,对于链队列,一般不会产生队列满。由于队列的长度变化一般比较大,所以用链式存储结构比用顺序存储结构更有利。

3.4 队列的应用实例

在日常生活中有很多队列的应用,如银行窗口排队等待服务、车辆在单行道上行驶等。本节介绍两个队列应用的实例:舞伴问题和打印队列的模拟管理。前者是用两个队列来完成匹配;后者是一个典型的离散事件的模拟。

3.4.1 舞伴问题

1. 问题叙述

在周末舞会上,男士们和女士们进入舞厅时,各自排成一队,跳舞开始时,依次从男队和女队的队头上各出一人配成舞伴,若两队初始人数不相同,则较长的一队中未配对者需等待下一支舞曲。现要求写一算法模拟上述舞伴配对问题。

2. 问题分析

从问题叙述看,先入队的男士和女士分别先出队配成舞伴,因此该问题具有典型的先进先出特性,可用队列作为算法的数据结构。

在算法中,假设男士和女士的记录存放在一个数组中作为输入,然后依次扫描该数组的各元素,并根据性别来决定是进入男队还是女队。当这两个队列构造完成后,依次将两队当

前的队头元素出队来配成舞伴，直至某队列变空为止。此时，若某队仍有等待配对者，输出此队列中等待者的人数及排在队头的等待者的名字，他（或她）将是下一支舞曲开始时第一个可获得舞伴的人。

3. 相关类型定义及具体算法

算法 3.25　舞伴配对算法。

```
typedef struct
{
    char name[20];
    char sex;                                   //性别,'F'表示女性,'M'表示男性
}Person;
typedef Person DataType;                        //将队列中元素的数据类型改为 Person
typedef struct
{
    DataType elem[maxsize];
    int front,rear;
}squeuetp;
void DancePartner(Person dancer[],int num)
{  //结构数组 dancer 中存放跳舞的男女,num 是跳舞的总人数
    int i;
    Person p;
    squeuetp Mdancers,Fdancers;
    InitQueue(&Mdancers);                       //男士队列初始化
    InitQueue(&Fdancers);                       //女士队列初始化
    for(i=0;i<num;i++)                          //依次将跳舞者依其性别入队
    {
        p=dancer[i];
        if(p.sex=='F')
            EnQueue(&Fdancers,p);               //排入女队
        else
            EnQueue(&Mdancers,p);               //排入男队
    }
    printf("The dancing partners are: \n \n");
    while(!QueueEmpty(&Fdancers)&&!QueueEmpty(&Mdancers))
    {
        p=DelQueue(&Fdancers);                  //女士出队
        printf("%s ",p.name);
        p=DelQueue(&Mdancers);                  //男士出队
        printf("%s\n",p.name);
    }
    if(!QueueEmpty(&Fdancers))
    {                                           //输出女士剩余人数及队头女士的名字
        printf("\n There are %d women waitin for the next round.\n",Size(&Fdancers));
        p=Head(&Fdancers);                      //取队头
        printf("%s will be the first to get a partner. \n",p.name);
    }
    else if(!QueueEmpty(&Mdancers))
    {                                           //输出男队剩余人数及队头男士的名字
```

```
            printf("\n There are %d men waiting for the next round.\n",Size(&Mdancers));
            p = Head(&Mdancers);
            printf(" %s will be the first to get a partner.\n",p.name);
        }
    }
```

3.4.2 打印队列的模拟管理

1. 问题描述

计算机中有一个缓冲区用来保存打印队列,所有的打印任务按先后顺序进入此队列,当打印机空闲时,就从队列中调出一个任务执行打印工作。现在要求编写一个程序,模拟在一个时段内(时间单位为分钟)的打印过程,要求输出每个打印任务的开始时间和打印时间。

2. 问题分析

该问题需要模拟出每个打印任务的开始时间和打印时间,这是一个典型的离散事件模拟。可以把一个任务的开始打印和打印完成这两个时刻称为事件,整个模拟程序会按事件发生的先后顺序进行处理。

在这个问题中,不妨将事件逐个存放在一个队列里,用循环依次触发这些事件。在每一个开始打印事件发生时,显示其任务编号和开始时间;在每一个打印结束事件发生时,显示该任务的打印时间,流程图如图 3.14 所示。

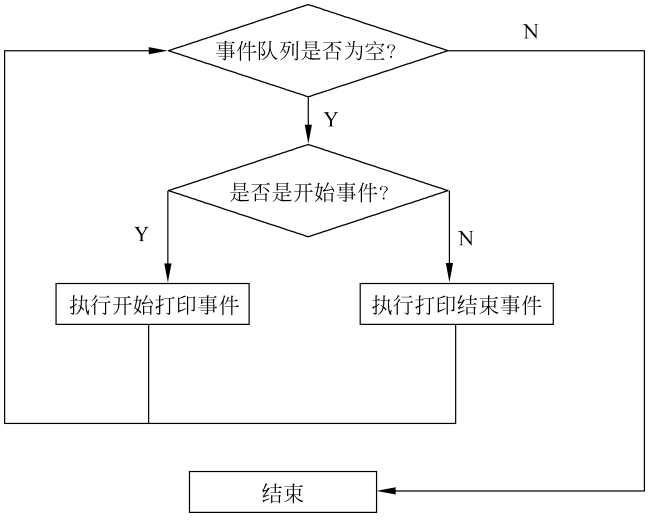


图 3.14 模拟打印队列的流程图

现在的问题是,如何添加这些事件呢?

解决方法是:可以在一个任务的开始事件中添加这个任务的结束事件,在一个任务的结束事件中添加下一个任务的开始事件(当然要先判断是否已超过了打印机的工作时间)。

下面是算法中要使用的一些数据结构和操作的说明。

```
typedef struct EVENT                    //事件
```

```
{
    int number;                                  //任务编号
    int starttime;                               //事件开始时间
    int durtime;                                 //任务执行时间
    int type;                                    //type 为 0 表示开始打印事件,否则打印结束事件
    struct EVENT * next;
}event;
typedef struct
{
    event * front;
    event * rear;
}event_list;                                     //事件队列
void task_over(event_list ev,event e,int lasttime,int tasknumber);
void task_start(event_list ev,event e);
void init_event_list(event_list * ev);          //初始化队列
void ins_event_list(event_list * ev,event * ee);   //在队列中插入一个事件
event del_event_list(event_list * ev);          //第一个结点出队列
int isempty(event_list * ev);                   //判断事件队列是否为空
int random();                                    //得到一个随机数
```

算法 3.26 是在上述工作的基础上实现。

算法 3.26　模拟打印队列。

```
void main()
{
    int lasttime = 100;                          //任务到达的最迟时刻
    event_list ev;
    int tasknumber = 0;                          //任务编号
    //添加一个任务开始事件
    event ee,en;
    ee.type = 0;
    ee.starttime = 0;
    ee.durtime = random();
    ee.number = tasknumber++;
    ins_event_list(&ev,&ee);
    while(!isempty(&ev))
    {
        en = del_event_list(&ev);
        if(en.type == 0)
            task_start(ev,en);
        else
            task_over(ev,en,lasttime,tasknumber);
    }
}
//任务开始事件执行的程序
void task_start(event_list ev,event e)
{
    printf("\n%d: tast%d now begin!",e.starttime,e.number);   //打印任务开始时间
    //添加任务结束事件
    event ee;
    ee.type = 1;
```

```
    ee.starttime = e.starttime + e.durtime;
    ee.number = e.number;
    ins_event_list(&ev,&ee);
}
//任务开始事件执行的程序
void task_over(event_list ev,event e,int lasttime,int tasknumber)
{
   printf("\n%d: tast%d now over!",e.starttime,e.number);      //打印任务结束时间
   int temp = random();                                        //下一个任务间隔多少时间
   if(e.starttime + temp <= lasttime)
   { //添加下一任务开始事件
     event ee;
     ee.type = 0;
     ee.starttime = e.starttime + temp;
     ee.durtime = random();                                    //下一个任务的执行时间
     ee.number = tasknumber++;
     ins_event_list(&ev,&ee);
   }
}
```

该算法能够模拟一个打印机的执行情况。当然,在实际应用中还可能有多种情况,如每个打印任务的最短时间、最长时间以及两个任务的最长和最短时间间隔可以由用户设定。另外,若有多个打印机从一个打印队列提取任务该怎么处理?若有多个打印机从多个队列提取任务又该怎么做?这些问题留给读者思考。

3.5 递归

递归是一个数学概念,也是一种有用的程序设计方法。在程序设计中,处理重复性计算最常用的方法是组织迭代循环,此外还可以采用递归计算的方法,特别是非数值计算领域中更是如此。

递归本质上也是一种循环的程序结构,它把较复杂的计算逐次归结为较简单的情形的计算,一直归结到最简单的情形的计算,并得到计算结果为止。许多问题都采用递归方法来编写程序,使得程序非常简洁和清晰,易于分析。

3.5.1 递归的定义及递归模型

1. 递归的定义

程序调用自身的编程方法称为递归。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。

直接递归:

```
fun_a()
{…
    fun_a()
```

```
…
}
```

间接递归：

```
fun_a()
{…
    fun_b()
    …
}
fun_b()
{   …
    fun_a()
    …
}
```

许多计算机问题采用递归的方法来解决非常方便。但是，一个问题要采用递归方法来解决时，必须符合以下 3 个条件。

(1) 解决问题时，可以把一个问题转化为一个新的问题，而这个新的问题的解决方法仍与原问题的解法相同，只是所处理的对象有所不同，这些被处理的对象之间是有规律地递增或递减的。

(2) 可以通过转化过程使问题得到简化。

(3) 必定要有一个明确的结束递归的条件，否则递归将会无休止地进行下去，直到耗尽系统资源。也就是说必须要有某个终止递归的条件。

例如求阶乘的问题。如果要求 n 的阶乘($n!$)，可以把这个问题转化为 $n*(n-1)!$；而要求$(n-1)!$ 又可转化为$(n-1)*(n-2)!$ ……这里面都有一个数乘以另一个数的阶乘的问题，被处理的对象分别是 n，$n-1$，…，这些数是有规律地递减。为了避免程序无休止地进行下去，必须要给一个结束条件。该问题恰好有一个结束条件，即当 $n=0$ 时，$0!=1$。

下述 3 种情况常常会用到递归的方法。

(1) 定义是递归的。

阶乘函数的定义就是采用的递归方法，其定义为

$$n! = \begin{cases} 1 & n=0 \quad //递归终止条件 \\ n*(n-1) & n>0 \quad //递归步骤 \end{cases}$$

求解阶乘函数的递归算法如算法 3.27。

算法 3.27　求解阶乘函数的递归算法。

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

(2) 问题的解法是递归的。

汉诺塔问题：古代有一个梵塔，塔内有 A、B、C 3 个柱，A 柱上有 64 个盘子，盘子大小不等，大的在下，小的在上，如图 3.15 所示。有一个和尚想把这 64 个盘子从 A 柱移到 C 柱，

但每次只允许移动一个盘子,并且在移动过程中,3 个柱上的盘子始终保持大盘在下,小盘在上。在移动过程中可以利用 B 柱,要求给出移动的步骤。

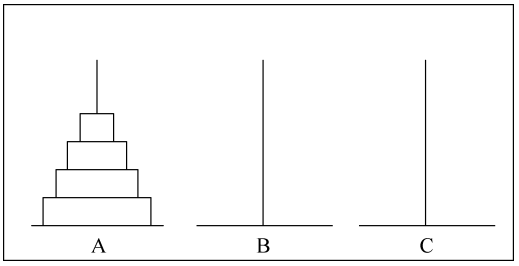


图 3.15 汉诺塔问题

汉诺塔问题的解法就是按递归算法实现的。

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 3 步:

① 用 C 柱作过渡,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 柱作过渡,将 B 柱上的$(n-1)$个盘子移到 C 柱上。

算法 3.28 汉诺塔问题算法

```
void hanoi(int n,char A,char B,char C)
  {
     if(n==1)
         move(1,A,C);
     else{
            hanoi(n-1,A,C,B);
            move(n,A,C);
            hanoi(n-1,B,A,C);
          }
  }
```

(3) 数据结构是递归定义的。

在第 5 章介绍二叉树时,将看到二叉树是一种递归定义的数据结构,其遍历等操作常常采用递归的方法实现。

单链表也可以看成是一个递归的数据结构,其定义为: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。

算法 3.29 搜索单链表最后一个结点并打印其数值算法

```
void Print(Lnode *f)                                    //f 是单链表的头指针
{
    if (f!=NULL){
        if (f->next == NULL)
            printf("%d\n", f->data);
        else Print(f->next);
    }
}
```

2. 递归模型

递归设计首先要确定求解问题的递归模型，了解递归的执行过程，在此基础上进行递归程序设计，同时还要掌握从递归到非递归的转换过程。

递归模型反映一个递归问题的递归结构，例如：

(1) $f(0)=1$；

(2) $f(n)=n*f(n-1)$，$n>0$。

(1)给出了递归的终止条件，(2)给出了 $f(n)$的值与 $f(n-1)$的值的关系，在这个问题中，把(1)称为递归出口，(2)称为递归体。

一般地，一个递归模型是由递归出口和递归体两部分组成，前者确定递归到何时为止，后者确定递归的方式。

(1) 递归出口的一般格式为：

$$f(s_0)=m_0$$

这里的 s_0 与 m_0 均为常量。有的递归问题可能有几个递归出口。

(2) 递归体的一般格式为：

$$f(s)=g(f(s_1),f(s_2),\cdots,f(s_n),c_1,c_2,\cdots,c_m)$$

其中，s 是一个递归的“大问题”；$s_1,s_2,\cdots,s_n$ 是递归的“小问题”；$c_1,c_2,\cdots,c_m$ 是若干个可以直接(用非递归方法)解决的问题；g 是一个非递归函数，反映了递归问题的结构。

例如，无穷数列 1,1,2,3,5,8,13,21,34,55,…，称为 Fibonacci 数列，它可以递归地定义为：

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

其中，$n=0$ 和 $n=1$ 时是递归出口，$n>1$ 时是递归体。

3.5.2　递归的实现

递归过程在实现时，需要自己调用自己，层层向下递归，退出时的次序则正好相反，例如阶乘函数的递归过程如图 3.16 所示。

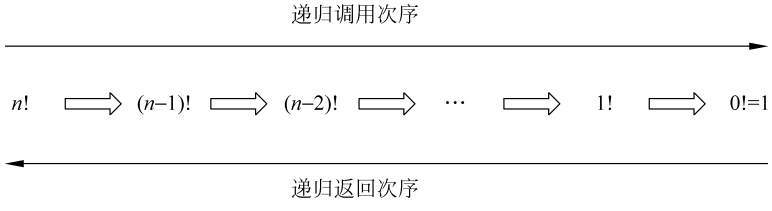


图 3.16　阶乘函数的递归过程

为保证递归过程的每次调用和返回得以正确执行，必须解决调用时的参数传递和返回地址问题。因此，在每次递归过程调用时，必须做地址保存、参数传递等工作。一般地，每一层递归调用所需要保存的信息构成一个工作记录，一个工作记录通常包括如下内容：

(1) 返回地址,即本次递归过程调用语句的后继语句的地址;

(2) 本次调用中与形参结合的实参值,包括函数名、引用参数与数值参数等;

(3) 本次递归调用中的局部变量值。

每执行一次递归调用,系统就需要建立一个新的工作记录,并将其压入递归工作栈;每退出一层递归,就从递归工作栈中弹出一个工作记录。由此可见,栈顶的工作记录必定是当前正在执行的这一层递归调用的工作记录,所以又称为当前活动工作记录。

例 3.2 说明下列阶乘函数 Factorial()的活动记录组成部分以及每一次调用时的活动记录的值。

```
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 ____↑
  }
```

说明:阶乘函数 Factorial()的工作记录由实参值 n、返回位置和局部变量 temp 3 个域组成,为了简化问题,只考察由实参值 n 和返回位置这两个域组成的工作记录。

阶乘函数 Factorial()每一次递归调用时的活动记录如图 3.17 所示。

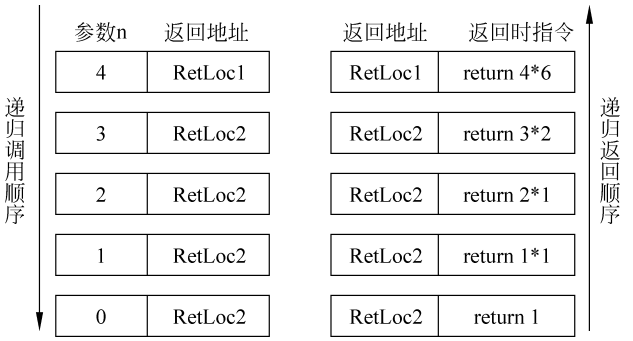


图 3.17 阶乘函数 Factorial()每一次递归调用时的活动记录

3.5.3 递归设计

进行递归设计时,首先要给出递归模型,然后再转换成对应的 C 语言函数。

从递归的执行过程看,要解决 $f(s)$,不是直接求其解,而是转化为计算 $f(s')$和一个常量 c'。求解 $f(s')$的方法与环境和求解 $f(s)$的方法与环境是相似的,但 $f(s)$是一个“大问题”,而 $f(s')$是一个“较小问题”,尽管 $f(s')$还未解决,但向解决目标靠近了一步,这就是一个“量变”,如此到达递归出口,便发生了质变,递归问题就解决了。因此,递归设计就是要给

出合理的“较小问题”，然后确定“大问题”的解与“较小问题”之间的关系，即确定递归体；最后朝此方向分解，必然有一个简单基本问题解，以此作为递归出口。由此得出递归设计的步骤如下：

(1) 对原问题 $f(s)$进行分析，假设出合理的较小问题 $f(s')$；

(2) 假设 $f(s')$是可解的，在此基础上确定 $f(s)$的解，即给出 $f(s)$与 $f(s')$的关系；

(3) 确定一个特定情况($f(1)$或 $f(0)$)的解，由此作为递归出口。

递归函数编写注意事项如下：

(1) 利用递归边界书写出口/入口条件；

(2) 递归调用时参数要朝出口方向修改，每次递归发生改变的量要作为参数出现；

(3) 当递归函数有返回值时，在函数体内对每个分支的返回值赋值；

(4) 谨慎使用循环语句；

(5) 注意理解当前的工作环境(即工作栈的内容)。

递归函数编写注意事项的说明见例 3.3。

例 3.3　用递归函数实现归并排序算法。

```
void MergeSort(Type a[],int left,int right)     //对数组 a 进行归并排序
//排序区间[left,right]会发生改变,为此 left 和 right 作为参数出现
   {
      if (left < right)                          //出口/入口条件
      {
               int i = (left + right)/2;
               mergeSort(a,left,i);              //参数朝出口方向修改
               mergeSort(a,i + 1,right);         //参数朝出口方向修改
               merge(a,b,left,i,right);          //两个有序表归并为一个有序表,达到整体有序
               copy(a,b,left,right);
      }
   }
```

3.5.4　递归到非递归的转换

递归方法虽然在解决某些问题时是最直观、最方便的方法，但却并不是一种高效的方法，主要原因在于递归方法过于频繁地进行函数调用和参数传递。在这种情况下，若采用循环或递归算法的非递归实现，将会大大提高算法的执行效率。

求解递归问题有两种方法：一种是直接求值，不需要回溯的；另一种是不能直接求值，需要回溯的。这两种方式在转换成非递归问题时采用的方式也不相同：前者使用一些中间变量保存中间结果，称为直接转换法；后者需要回溯，所以要用栈来保存中间结果，称为间接转换法。

1. 直接转换法

该方法使用一些中间变量保存中间结果。

例 3.4 编写一个函数,用非递归方法计算如下递归函数的值:

$$f(1)=1$$
$$f(2)=1$$
$$f(n)=f(n-1)+f(n-2) \quad n>2$$

解:

```
int f(int n)
{
    int i,s,s1,s2;
    s1 = 1;            //s1 用于保存 f(n-1)的值
    s2 = 1;            //s2 用于保存 f(n-2)的值
    s = 1;
    for (i = 3;i <= n;i++)
    {
        s = s1 + s2;
        s2 = s1;
        s1 = s;
    }
    return(s);
}
```

2. 间接转换法

该方法使用栈保存中间结果。其一般过程如下。

```
将初始状态 s 进栈;
while (栈不为空)
{
    退栈,将栈顶元素赋给 s;
    if (s 是要找的结果) 返回;
    else
        {
        寻找到 s 的相关状态 s1;
        将 s1 进栈;
        }
}
```

间接转换法的示例见算法 5.4。

3.6 C++中的栈和队列

3.6.1 C++中的栈

借助 C++的模板抽象类来定义栈抽象数据类型 stack,它作为顺序栈类 seqstack 的基类。例 3.5 给出了栈类 stack 的规范定义,保存在头文件 stack.h 中。

例 3.5 栈类。

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

```
template < class T >
class stack
{
    public:
        virtual bool isempty( ) const = 0;
        virtual bool isfull( ) const = 0;
        virtual bool top(T& x) const = 0;
        virtual bool push(T x) = 0;
        virtual bool pop(T& x) = 0;
        virtual void clear( ) = 0;
};
```

栈的顺序表示方式也用 C++中的一维数组加以描述。在顺序栈类 seqstack 中，私有成员包括最大栈顶指针(下标)maxtop、当前栈顶指针 top 和指向数组的指针 elem。

例 3.6　给出顺序栈类的定义和实现，它是 stack 类的派生类，其定义保存在头文件 seqstack.h 中。

```
//顺序栈类
#include < stack.h >
template < class T >
class seqstack:public stack < T >
{
    public:
        seqstack(int msize);
        ~seqstack(){delete[ ] elem;}
        bool isempty( ) const {return top == - 1;}
        bool isfull( ) const{return top == maxtop;}
        bool top(T& x) const;
        bool push(T x) = 0;
        bool pop(T& x) = 0;
        void clear(){ top = - 1;}
    private:
        int top;                                    //栈顶指针
        int maxtop;                                 //最大栈顶指针
        T * elem;
};
template < class T >
seqstack < T >::seqstack(int msize)
{
    maxtop = msize - 1;
    elem = new T[msize];
    top = - 1;
}
template < class T >
bool seqstack < T >::isempty( ) const
{
    return n == 0;
}
template < class T >
bool seqstack < T >::top(T &x) const
```

```
{
    if (isempty()){
        cout <<"empty"<< endl;
        return false;
    }
    x = elem[top];
    return true;
}
template < class T >
bool seqstack < T >::push(T x) const
{
    if (isfull()){
        cout <<"overflow"<< endl;
        return false;
    }
    s[top++] = x;
    return true;
}
template < class T >
bool seqstack < T >::pop(T& x) const
{
    if (isempty()){
        cout <<"underflow"<< endl;
        return false;
    }
    x = elem[ --top];
    return true;
}
```

3.6.2 C++中的队列

借助C++的模板抽象类来定义队列抽象数据类型queue,它作为循环队列类seqqueue的基类。

例3.7 给出队列类queue的规范定义,保存在头文件queue.h中。

```
//队列类
#include < iostream.h >
template < class T >
class queue
{
    public:
        virtual bool isempty() const = 0;
        virtual bool isfull() const = 0;
        virtual bool front(T& x) const = 0;
        virtual bool enqueue(T x) = 0;
        virtual bool dequeue(T& x) = 0;
        virtual void clear() = 0;
};
```

队列的顺序表示方式也可以用C++中的一维数组加以描述。循环队列类seqqueue中,

私有成员包括最大队列容量 maxsize、队头指针 front、队尾指针 rear 和指向数组的指针 elem。

例 3.8 给出循环队列类的定义和实现，它是 queue 类的派生类，保存在头文件 seqqueue.h 中。

```
//循环队列类
#include <queue.h>
template <class T>
class seqqueue:public queue <T>
{
    public:
        seqqueue(int msize);
        ~seqqueue(){delete[] elem;}
        bool isempty() const {return front == rear;}
        bool isfull() const{return (rear + 1) % maxsize == front;}
        bool front(T& x) const;
        bool enqueue(T x);
        bool dequeue(T& x);
        void clear(){front = rear = 0;}
    private:
        int front,rear;                          //栈顶指针
        int maxsize;                             //最大栈顶指针
        T *elem;
};
template <class T>
seqqueue <T>::seqqueue(int msize)
{
    maxsize = msize;
    elem = new T[msize];
    front = rear = 0;
}
template <class T>
bool seqqueue <T>::front(T &x) const
{
    if (isempty()){
        cout <<"empty"<< endl;
        return false;
    }
    x = elem[(front + 1) % maxsize];
    return true;
}
template <class T>
bool seqqueue <T>::enqueue(T x) const
{
    if (isfull()){
        cout <<"overflow"<< endl;
        return false;
    }
    elem[rear = (rear + 1) % maxsize] = x;
    return true;
```

```
}
template < class T >
bool seqqueue < T >::dequeue(T& x) const
{
    if (isempty()){
        cout <<"underflow"<< endl;
        return false;
    }
    x = elem[front = (front + 1) % maxsize];
    return true;
}
```

习题 3

1. 设将整数 a,b,c,d 依次进栈,但只要出栈时栈非空,则可将出栈操作按任何次序加入其中,请回答下述问题:

(1) 若执行以下操作序列 Push(a),Pop(),Push(b),Push(c),Pop(),Pop(),Push(d),Pop(),则出栈的数字序列为何(这里 Push(i)表示 i 进栈,Pop()表示出栈)?

(2) 能否得到出栈序列 adbc 和 adcb 并说明为什么不能得到或者如何得到?

(3) 请分析 a,b,c,d 的所有排列中,哪些序列是可以通过相应的入、出栈操作得到的。

2. 分别借助顺序栈和链栈,将单链表倒置。

3. 有两个栈 A,B 分别存储一个升序数列,现要求编写算法把这两个栈中的数合成一个升序队列。

4. 设两个栈共享一个数组空间,其类型定义见 3.1.2 节,试写出两个栈公用的读栈顶元算法 elemtp top_dustack(dustacktp ds,p; int i)、进栈操作算法 void push_dustack(dustacktp ds,p; int i,elemtp x)及出栈算法 elemtp pop_dustack(dustacktp ds,p; int i)。其中 i 的取值是 1 或 2,用以指示栈号。

5. 假设以数组 sequ(0..m−1)存放循环队列元素,同时设变量 rear 和 quelen 分别指示循环队列中队尾元素和内含元素的个数。试给出此循环队列的队满条件,并写出相应的入队列和出队列的算法(在出队列的算法中要返回队头元素)。

6. 假设以带头结点的环形链表表示队列,并且只设一个指针指向队尾元素结点(注意不设头指针),试编写相应的初始化队列、入队列、出队列的算法。

上机练习 3

1. 设单链表中存放 n 个字符,设计一个算法,使用栈判断该字符串是否中心对称,如 abccba 即为中心对称字符串(根据题目填空完善程序)。

提示:先用 create()函数从用户输入的字符串创建相应的单链表,然后调用 judge()函数判断是否为中心对称字符串。在 judge()函数中先将字符串进栈,然后将栈中的字符逐个与单链表中字符进行比较。

```
#include <stdio.h>
#include <malloc.h>
#define MaxLen 100
typedef struct node
{
    char data;
    struct node *next;
}cnode;
cnode *create(char s[])
{
    int I=0;
    cnode *h,*p,*r;
    while(s[I]!='\0')
     {
          p=(cnode *)malloc(sizeof(cnode));
          p->data=s[I];p->next=NULL;
          if(I==0)
              {
              h=p;
              ._____;                          /*r始终指向最后一个结点*/
              }
          else
              {
              r->next=p;r=p;
              }
              I++;
     }
return h;
}

int judge(cnode *h)
{
  char st[MaxLen];
  int top=0;
  cnode *p=h;
  while(p!=NULL)
       {
       st[top]=p->data;
       top++;
       p=p->next;
       }
  p=h;
  while(p!=NULL)
       {
       top--;
       if(p->data==st[top])
       p=p->next;
       else
       break;
       }
  if(p==NULL)
```

```
        return ________;
    else
        return ________;
}
void main()
{
    char str[maxlen];
    cnode * h;
    printf("输入一个字符串:");
    scanf("%c",str);
    h=create(________);
    if (judge(h)==1)
        printf("str 是中心对称字符串\n");
    else
        printf("str 不是中心对称字符串\n");
}
输入一个字符串:abccba
输出:________
```

2.设一个算术表达式中包含圆括号、方括号和花括号3种类型的括号,编写一个算法判断其中的括号是否匹配。

提示:本题使用一个运算符栈st,当遇到(、[或{时进栈,当遇到}、]、)时判断栈顶是否为相应的括号,若是退栈则继续执行,否则算法结束。

3. 设计一个程序,演示用运算符优先法对算术表达式求值的过程。

基本要求:以字符序列的形式从终端输入语法正确的、不含变量的整数表达式。利用书中表3.1给出的运算符优先关系,实现对算术四则混合运算表达式的求值,并仿照书中例3.1演示在求值中运算符栈、运算数栈、输入字符和主要操作的变化过程。

测试数据:3*(7-2)。