数据结构简明教程（第 2 版）

配套练习题参考答案

李春葆等

清华大学出版社 2019

1. 练习题 1 参考答案

1. 单项选择题

（1）D （2）C （3）C （4）A （5）C
（6）B （7）C （8）A （9）C （10）B

2. 填空题

（1）①逻辑结构 ②存储结构 ③运算（不限制顺序）
（2）①线性结构 ②非线性结构（不限制顺序）
（3）①数据元素 ②关系
（4）①没有 ②没有
（5）①前驱 ②一 ③后继 ④任意多个
（6）任意多个
（7）①顺序 ②链式 ③索引 ④哈希（不限制顺序）
（8）①时间 ②空间（不限制顺序）
（9）问题规模（通常用 n 表示）。
（10）辅助或临时空间

3. 简答题

（1）**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处理步骤。

（2）**答：**$T_1(n)=O(n\log_2 n)$，$T_2(n)=O(n^{\log_2 3})$，$T_3(n)=O(n^2)$，$T_4(n)=O(n\log_2 n)$。

（3）**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。

（4）**答：**语句 s++的执行次数 $\sum_{i=1}^{n-2}\sum_{j=n}^{i}1=\sum_{i=1}^{n-2}(n-i+1)=n+(n-1)+\square\ +3=\frac{(n+3)(n-2)}{2}$

。

（5）**答：**其中 x++语句为基本运算语句，$T(n)=\sum_{i=1}^{n}\sum_{j=i+1}^{n}1=\sum_{i=1}^{n}(n-i)=\frac{n(n-1)}{2}$

$=O(n^2)$。

（6）**答：**由于内循环 j 的取值范围，所以 $i\leqslant n/2$，则 $m=\sum_{i-1}^{n/2}\sum_{j=2i}^{n}=\sum_{i=1}^{n/2}(n-(2i-1))=n^2/4$，该程序段的时间复杂度为 $O(n^2)$。

2. 练习题 2 参考答案

1. 单项选择题

（1）A （2）C （3）A （4）B （5）C
（6）D （7）C （8）B （9）A （10）C
（11）B （12）A （13）C （14）D （15）D
（16）D （17）A （18）C （19）A （20）D

2. 填空题

（1）L.length=0
（2）O(1)
（3）O(*n*)
（4）*n*−*i*
（5）①物理存储位置 ②指针域
（6）①前驱 ②O(*n*)
（7）*q*=*p*−>next; *p*−>next=*q*−>next; free(*q*);
（8）*s*−>next= *p*−>next; *p*−>next=*s*;
（9）O(1)
（10）L−>next==L

3. 简答题

（1）**答：**顺序存储结构中，逻辑上相邻元素的存储空间也是相邻的，无需额外空间表示逻辑关系，所以存储密度大，同时具有随机存取特性。缺点是插入或删除元素时平均需要移动一半的元素，同时顺序存储结构的初始分配空间大小难以事先确定。

链式存储结构中，逻辑上相邻元素的存储空间不一定是相邻的，需要通过指针域需表示逻辑关系，所以存储密度较小，同时不具有随机存取特性。优点是插入或删除时不需要结点的移动，仅仅修改相关指针域，由于每个结点都是动态分配的，所以空间分配适应性好。

顺序表适宜于做查找这样的静态操作；链表宜于做插入、删除这样的动态操作。若线性表的长度变化不大，且其主要操作是查找，则采用顺序表；若线性表的长度变化较大，且其主要操作是插入、删除操作，则采用链表。

（2）**答：**对于表长为*n*的顺序表，在等概率的情况下，插入一个元素所需要移动元素的平均次数为*n*/2，删除一个元素所需要移动元素的平均次数为(*n*−1)/2。

（3）**答：**在链表中设置头结点后，不管链表是否为空表，头结点指针均不空；另外使得链表的操作（如插入和删除）在各种情况下得到统一，从而简化了算法的实现过程。

（4）**答：**对于双链表，在两个结点之间插入一个新结点时，需修改前驱结点的next域、后继结点的prior域和新插入结点的next、prior域。所以共修改4个指针。

对于单链表，在两个结点之间插入一个新结点时，需修改前一结点的next域，新插入

结点的next域。所以共修改两个指针。

（5）**答**：在单链表中，删除第一个结点的时间复杂度为O(1)。插入结点需找到前驱结点，所以在尾结点之后插入一个结点，需找到尾结点，对应的时间复杂度为O(*n*)。

在仅有头指针不带头结点的循环单链表中，删除第一个结点的时间复杂度O(*n*)，因为删除第一个结点后还要将其改为循环单链表；在尾结点之后插入一个结点的时间复杂度也为O(*n*)。

在双链表中，删除第一个结点的时间复杂度为O(1)；在尾结点之后插入一个结点，也需找到尾结点，对应的时间复杂度为O(*n*)。

在仅有尾指针的循环单链表中，通过该尾指针可以直接找到第一个结点，所以删除第一个结点的时间复杂度为O(1)；在尾结点之后插入一个结点也就是在尾指针所指结点之后插入一个结点，时间复杂度也为O(1)。因此④最节省运算时间。

4. 算法设计题

（1）**解**：设顺序表 L 中元素个数为 *n*。*i* 从 0 到 *n*−2 扫描顺序表 L：若 L.data[*i*]大于后面的元素 L.data[*i*+1]，则返回 false。循环结束后返回 true。对应的算法如下：

```
int Increase(SqList L)
{   int i;
    for (i=0;i<L.length-1;i++)
        if (L.data[i]>L.data[i+1])
            return 0;
    return 1;
}
```

（2）**解**：遍历顺序表 L 的前半部分元素，对于元素 L.data[*i*]（0≤*i*＜L.length/2），将其与后半部分对应元素 L.data[L.length−*i*−1]进行交换。对应的算法如下：

```
void Reverse(SqList &L)
{   int i;
    ElemType x;
    for (i=0;i<L.length/2;i++)
    {   x=L.data[i];           //L.data[i]与L.data[L.length-i-1]交换
        L.data[i]=L.data[L.length-i-1];
        L.data[L.length-i-1]=x;
    }
}
```

本算法的时间复杂度为 O(*n*)。

（3）**解**：通过扫描顺序表 L 求出最后一个最大元素的下标 maxi。将 L->data[maxi+1]元素及之后的所有元素均后移一个位置，将 *x* 放在 L->data[mai+1]处，顺序表长度增 1。对应的算法如下：

```
void Insertx(SqList &L,ElemType x)
{   int i,maxi=0;
```

```
    for (i=1;i<L.length;i++)
        if (L.data[i]>=L.data[maxi])
            maxi=i;
    for (i=L.length;i>maxi+1;i--)    //将data[maxi+1..n-1]后移
        L.data[i]=L.data[i-1];
    L.data[maxi+1]=x;                //插入元素x
    L.length++;                      //顺序表长度增1
}
```

（4）**解**：设顺序表 L 中元素个数为 n。i 从 1 到 n-1 扫描顺序表 L：如果 L.data[i-1] 大于 L.data[i]，将两者交换，扫描结束后最大元素移到 L 的最后面。对应的算法如下：

```
void Movemax(SqList &L)
{   int i;
    ElemType tmp;
    for (i=1;i<L.length;i++)
        if (L.data[i-1]>L.data[i])
        {   tmp=L.data[i-1];          //data[i-1]与data]i]交换
            L.data[i-1]=L.data[i];
            L.data[i]=tmp;
        }
}
```

（5）**解**：用 p 指针遍历整个单链表，pre 总是指向 p 结点的前驱结点（初始时 pre 指向头结点，p 指向首结点），当 p 指向第一个值为 x 的结点时，通过 pre 结点将其删除，返回 1；否则返回 0。对应的算法如下：

```
int Delx(SLinkNode *&L,ElemType x)
{   SLinkNode *pre=L,*p=L->next;      //pre指向p结点的前驱结点
    while (p!=NULL && p->data!=x)
    {   pre=p;
        p=p->next;                    //pre、p同步后移
    }
    if (p!=NULL)                      //找到值为x的结点
    {   pre->next=p->next;            //删除p结点
        free(p);
        return 1;
    }
    else return 0;                    //未找到值为x的结点
}
```

（6）**解**：判定链表 L 从第 2 个结点开始的每个结点值是否比其前驱结点值大。若有一个不成立，则整个链表便不是递增的；否则是递增的。对应的算法如下：

```
int increase(SLinkNode *L)
{   SLinkNode *pre=L->next,*p;        //pre指向第一个数据结点
```

```
    p=pre->next;                    //p指向pre结点的后继结点
    while (p!=NULL)
    {   if (p->data>=pre->data)     //若正序则继续判断下一个结点
        {   pre=p;                  //pre、p同步后移
            p=p->next;
        }
        else return 0;
    }
    return 1;
}
```

（7）**解**：采用重新单链表的方法，由于要保持相对次序，所以采用尾插法建立新表 *A*、*B*。用 *p* 遍历原单链表 *A* 的所有数据结点，若为偶数结点，将其链到 *A* 中，若为奇数结点，将其链到 *B* 中。对应的算法如下：

```
void Split(SLinkNode *&A,SLinkNode *&B)
{   SLinkNode *p=A->next,*ta,*tb;
    ta=A;                           //ta总是指向A链表的尾结点
    B=(SLinkNode *)malloc(sizeof(SLinkNode)); //建立头结点B
    tb=B;                           //tb总是指向B链表的尾结点
    while (p!=NULL)
    {   if (p->data%2==0)           //偶数结点
        {   ta->next=p;             //将p结点链到A中
            ta=p;
            p=p->next;
        }
        else                        //奇数结点
        {   tb->next=p;             //将p结点链到B中
            tb=p;
            p=p->next;
        }
    }
    ta->next=tb->next=NULL;
}
```

本算法的时间复杂度为 O(*n*)，空间复杂度为 O(1)。

（8）**解**：用 *p* 指针遍历整个单链表，pre 总是指向 *p* 结点的前驱结点（初始时 pre 指向头结点，*p* 指向首结点），当 *p* 指向结点的结点值刚好大于 *x* 时查找结束。新创建一个结点 *s* 存放元素 *x*，将其插入到 pre 结点之后。对应的算法如下：

```
void Insertorder(SLinkNode *&L,ElemType x)
{   SLinkNode *s,*pre,*p;
    pre=L; p=L->next;
    while (p!=NULL && p->data<=x)
```

```
    {   pre=p;
        p=p->next;          //pre、p同步后移
    }
    s=(SLinkNode *)malloc(sizeof(SLinkNode));
    s->data=x;              //建立一个待插入的结点
    pre->next=s;            //在结点pre之后插入s结点
    s->next=p;
}
```

（9）**解**：扫描 L 的所有数据结点，复制产生 L1 的结点，采用尾插法创建 L1。对应的算法如下：

```
void Copy(SLinkNode *L,SLinkNode *&L1)
{   SLinkNode *p,*s,*tc;
    L1=(SLinkNode *)malloc(sizeof(SLinkNode)); //创建L1的头结点
    tc=L1;
    p=L->next;
    while (p!=NULL)                 //扫描L的所有数据结点
    {   s=(SLinkNode *)malloc(sizeof(SLinkNode));
        s->data=p->data;            //由p结点复制产生s结点
        tc->next=s;                 //将s结点链接到L1末尾
        tc=s;
        p=p->next;
    }
    tc->next=NULL;                  //L1的尾结点next域置为空
}
```

（10）**解**：用 p 扫描单链表 L，pre 指向 *p* 结点的前驱结点，minp 指向最小值结点，minpre 指向最小值结点的前驱结点。当 *p* 不为 L 时循环：若 *p* 所指结点值小于等于 minp 所指结点值，置 minpre=pre，minp=p，然后 pre、p 同步后移一个结点。循环结束后 minp 指向最后一个最小结点，通过 minpre 结点将 minp 结点从中删除，然后将其插入到表头即头结点之后。对应的算法如下：

```
void Move(SLinkNode *&L)
{   SLinkNode *p=L->next,*pre=L;
    SLinkNode *minp=p,*minpre=L;
    while (p!=L)
    {   if (p->data<=minp->data)
        {   minp=p;
            minpre=pre;
        }
        pre=p;                  //pre、p同步后移
        p=p->next;
    }
    minpre->next=minp->next;    //删除minp结点
```

```
    minp->next=L->next;          //将minp结点插入到头结点之后
    L->next=minp;
}
```

（11）**解**：采用头插法重建循环单链表 L 的思路，先建立一个空的循环单链表，用 p 遍历所有数据结点，每次将 *p* 结点插入到前端。对应的算法如下：

```
void Reverse(SLinkNode *&L)
{   SLinkNode *p=L->next,*q;
    L->next=L;                   //建立一个空循环单链表
    while (p!=L)                 //扫描所有数据结点
    {   q=p->next;               //临时保存p结点的后继结点
        p->next=L->next;         //将p结点插入到前端
        L->next=p;
        p=q;
    }
}
```

（12）**解**：先找到第一个元素值为 *x* 的结点 *p*，pre 指向其前驱结点，本题是将 *p* 结点移到 pre 结点之前，实现过程是：删除 *p* 结点，再将其插入到 pre 结点之前。对应的算法如下：

```
int Swap(DLinkNode *L, ElemType x)
{   DLinkNode *p=L->next, *pre;
    while (p!=NULL && p->data!=x)
        p=p->next;
    if (p==NULL)                      //未找到值为x的结点
        return 0;
    else
    {   pre=p->prior;                 //pre指向结点p的前驱结点
        if (pre!=L)
        {   pre->next=p->next;        //先删除p结点
            if (p->next!=NULL)
                p->next->prior=pre;
            pre->prior->next=p;       //将p结点插入到pre结点之前
            p->prior=pre->prior;
            pre->prior=p;
            p->next=pre;
            return 1;
        }
        else
            return 0;                 //表示值为x的结点是首结点
    }
}
```

（13）**解：**首先找到双链表 L 的尾结点 p。pre 指向 p 结点的前驱结点，然后从链表中删除 p 结点，再将 p 结点插入到 pre 结点之前。本算法需要遍历整个双链表才能找到尾结点，所以算法的时间复杂度为 O(n)。对应的算法如下：

```
void SwapLast(DLinkNode *&L)
{   DLinkNode *p=L,*pre;
    while (p->next!=NULL)
        p=p->next;
    pre=p->prior;           //post指向p结点的前驱结点
    pre->next=p->next;      //删除p结点
    if (p->next!=NULL)
        p->next->prior=pre;
    pre->prior->next=p;     //将p结点插入到pre结点之前
    p->prior=pre->prior;
    pre->prior=p;
    p->next=pre;
}
```

（14）**解：**p 从循环双链表 L 的首结点开始扫描，当 p≠L 时循环：若 p 结点值为 x，post=p->next 临时保存其后继结点，通过 p 结点前后指针域删除 p 结点，并释放其空间，置 p=post 继续查找；若 p 结点值不为 x，执行 p=p->next。对应的算法如下：

```
void DeleteAllx(DLinkNode *&L,ElemType x)
{   DLinkNode *p=L->next,*post;
    while (p!=L)                        //扫描数据结点p
    {   if (p->data==x)
        {   post=p->next;
            p->prior->next=p->next;     //删除p结点
            p->next->prior=p->prior;
            free(p);                    //释放p结点
            p=post;
        }
        else p=p->next;
    }
}
```

（15）**解：**通过 L->prior 找到尾结点 p。pre 指向 p 结点的前驱结点，然后从链表中删除 p 结点，再将 p 结点插入到 pre 结点之前。本算法不含有循环语句，所以算法的时间复杂度为 O(1)。对应的算法如下：

```
void SwapLast(DLinkNode *&L)
{   DLinkNode *p=L->prior,*pre;
    pre=p->prior;               //pre指向p结点的前驱结点
    pre->next=p->next;          //删除p结点
    p->next->prior=pre;
```

```
    pre->prior->next=p;          //将p结点插入到pre结点之前
    p->prior=pre->prior;
    pre->prior=p;
    p->next=pre;
}
```

3. 练习题 3 参考答案

1. 单项选择题

（1）A	（2）D	（3）C	（4）D	（5）A
（6）D	（7）C	（8）A	（9）D	（10）A
（11）C	（12）①A ②D	（13）B	（14）A	（15）C
（16）D	（17）D	（18）B	（19）C	（20）D

2. 填空题

（1）①栈顶 ②栈底

（2）*n*

（3）1 进栈，1 出栈，2 进栈，2 出栈，3 进栈，3 出栈，4 进栈，4 出栈，5 进栈，5 出栈。

（4）*cdbae*、*cdeba*、*cdbea*。

（5）data[top]=*x*; top++;

（6）top−−; *x*=data[top];

（7）队列

（8）rear=(rear+1)%(*m*+1); *A*[rear]=*x*;

（9）front=(front+1)%(*m*+1); *x*=*A*[front];

（10）61

3. 简答题

（1）**答：**相同点：都属地线性结构，都可以用顺序存储或链表存储；栈和队列是两种特殊的线性表，即受限的线性表，只是对插入、删除运算加以限制。

不同点：①运算规则不同，线性表为随机存取，而栈是只允许在一端进行插入、删除运算，因而是后进先出表 LIFO；队列是只允许在一端进行插入、另一端进行删除运算，因而是先进先出表 FIFO。②用途不同，栈用于子程调用和保护现场等，队列用于多道作业处理、指令寄存及其他运算等等。

（2）**答：**栈的主要操作是进栈和出栈，栈底总是不变的，元素进栈是从栈底向栈顶一个方向伸长的，如果栈底设置在中间，伸长方向的另外一端空间难以使用，所以栈底总是设置在数组的一端而不是中间。

（3）**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）**答**：在循环队列中插入和删除元素时，不需要移动队中任何元素，只需要修改队尾或队头指针，并向队尾插入元素或从队头取出元素。

（5）**答**：一般的一维数组队列的尾指针已经到了数组的上界，不能再有进队操作，但其实数组中还有空位置，这就产生了“假溢出”。

采用循环队列是解决假溢出的途径，解决循环队列是空还是满的办法如下：

① 设置一个布尔变量以区别队满还是队空；

② 浪费一个元素的空间，用于区别队满还是队空。

③ 使用一个计数器记录队列中元素个数（即队列长度）。

通常采用法②，让队头指针 front 指向队首元素的前一位置，队尾指针 rear 指向队尾元素的位置，这样判断循环队列队空标志是：front=rear，队满标志是：(rear+1)%MaxSize=front。

4. 算法设计题

（1）**解**：定义一个栈 st，正向扫描 *a* 的所有字符，将每个字符进栈。再出栈所有字符，将其写入到 *a*[0..*n*−1]中。对应的算法如下：

```
void Reverse(char a[],int n)     //逆置a[0..n-1]
{   int i; char e;
    SqStack st;                  //定义一个顺序栈st
    InitStack(st);
    for (i=0;i<n;i++)            //依次将a[0..n-1]进栈
        Push(st,a[i]);
    for (i=0;i<n;i++)            //将a[n-1]～a[0]依次存放到a[0..n-1]
    {   Pop(st,e);
        a[i]=e;
    }
    DestroyStack(st);            //销毁栈st
}
```

（2）**解**：算法原理与《教程》中例 3.9 相同，仅仅将二进制改为八进制。对应的算法如下：

```
void trans(int d,char b[])       //b用于存放d转换成的八进制数的字符串
{   SqStack st;                  //定义一个顺序栈st
    InitStack(st);               //栈初始化
    char ch;
    int i=0;
```

```
    while (d!=0)
    {   ch='0'+d%8;             //求余数并转换为字符
        Push(st,ch);            //字符ch进栈
        d/=8;                   //继续求更高位
    }
    while (!StackEmpty(st))
    {   Pop(st,ch);             //出栈并存放在数组b中
        b[i]=ch;
        i++;
    }
    b[i]='\0';                  //添加字符串结束标志
    DestroyStack(st);           //销毁栈st
}
```

（3）**解**：对于给定的栈 st，设置一个临时栈 tmpst，先退栈 st 中所有元素并进入到栈 tmpst 中，最后的一个元素即为所求，然后将临时栈 tmpst 中的元素逐一出栈并进栈到 st 中，这样恢复 st 栈中原来的元素。对应算法如下：

```
int GetBottom(SqStack st,ElemType &x)//x在算法返回1时保存栈底元素
{   ElemType e;
    SqStack tmpst;                  //定义临时栈
    InitStack(tmpst);               //初始化临时栈
    if (StackEmpty(st))             //空栈返回0
    {   DestroyStack(tmpst);
        return 0;
    }
    while (!StackEmpty(st))         //临时栈tmpst中包含st栈中逆转元素
    {   Pop(st,x);
        Push(tmpst,x);
    }
    while (!StackEmpty(tmpst))      //恢复st栈中原来的内容
    {   Pop(tmpst,e);
        Push(st,e);
    }
    DestroyStack(tmpst);
    return 1;                       //返回1表示成功
}
```

（4）**解**：由于算法要求空间复杂度为 O(1)，所以不能使用一个临时队列。先利用例 3.13 的算法求出队列 qu 中的元素个数 m。循环 m 次，出次一个元素 x，再将元素 x 进队。最后的 x 即为队尾元素。对应的算法如下：

```
int Count(SqQueue sq)           //求循环队列qu中元素个数
{
    return (sq.rear+MaxSize-sq.front) % MaxSize;
```

```
}
int Last(SqQueue qu,ElemType &x) //求解算法
{    int i,m=Count(qu);
     if (m==0) return 0;          //队中元素个数为0返回0
     for (i=1;i<=m;i++)
     {    DeQueue(qu,x);          //出队元素x
          EnQueue(qu,x);          //将元素x进队
     }
     return 1;                    //成功找到队尾元素返回1
}
```

（5）**解：**先求出队列 qu 中的元素个数 *m*。*i* 从 1 到 *m* 循环，出队第 *i* 个元素 *e*，若 *e* 不为奇数，将其进队。对应的算法如下：

```
int Count(SqQueue sq)             //求循环队列qu中元素个数
{
     return (sq.rear+MaxSize-sq.front) % MaxSize;
}
void DelOdd(SqQueue &qu)          //求解算法
{    char e;
     int i,m=Count(qu);
     for (i=1;i<=m;i++)           //出队m次
     {    DeQueue(qu,e);
          if ((e-'0')%2!=1)       //将不是奇数的数字字符进队
               EnQueue(qu,e);
     }
}
```

4. 练习题 4 参考答案

1. 单项选择题

（1）B　　（2）D　　（3）D　　（4）B　　（5）B

2. 填空题

（1）①不包含任何字符（长度为 0）的串　②由一个或多个空格（仅由空格符）组成的串

（2）"abefg"

（3）O(*n*)

（4）*s*->next==NULL

（5）O(*n*)

3. 简答题

（1）**答**：由串 s 的特性可知，1 个字符的子串有 n 个，2 个字符的子串有 n−1 个，3 个字符的子串有 n−2 个，…，n−2 个字符的子串有 3 个，n−1 个字符的子串有 2 个。所以，非平凡子串的个数=n+(n−1)+(n−2)+…+2=$\frac{n(n+1)}{2}-1$。

（2）**答**：所有可能的条件为：

① s_1 和 s_2 为空串

② s_1 或 s_2 其中之一为空串

③ s_1 和 s_2 为相同的串

④ 若 s_1 和 s_2 两串长度不等，则长串由整数个短串组成。

（3）**答**：只要将线性表中元素类型改为 char，该线性表就是串，所以将单链表中结点数据域的类型改为 char，则基于单链表的算法设计方法都可以应用于链串。

4. 算法设计题

（1）**解**：因要求算法空间复杂度为 O(1)，所以只能对串 s 直接替换。从头开始扫描顺序串 s，一旦找到字符 x 便将其替换成 y。对应算法如下：

```
void Repchar(SqString &s,char x,char y)
{   int i;
    for (i=0;i<s.length;i++)
        if (s.data[i]==x)
            s.data[i]=y;
}
```

（2）**解**：置 i、j 均为 0。当 i<s.length 时循环：将 s.data[i]复制到 t.data[j]中，i 增大 2，j 增大 1。最后置串 t 的长度为 j。对应的算法如下：

```
void Oddcopy(SqString s,SqString &t)
{   int i=0,j=0;
    while (i<s.length)
    {   t.data[j]=s.data[i];
        i+=2; j++;
    }
    t.length=j;
}
```

（3）**解**：用 i 用于扫描串 s，j、k 分别表示串 s1、s2 的字符个数，初值均为 0。当 i<s.length 时循环：若 s.data[i]为数字字符，将其复制到 s1 中，置 j++；若 s.data[i]为字母字符，将其复制到 s2 中，置 k++。循环结束后，置 s1、s2 的长度分别为 j、k。对应的算法如下：

```
void Split(SqString s,SqString &s1,SqString &s2)
{   int i=0,j=0,k=0;
    while (i<s.length)
```

```
    {   if (s.data[i]>='0' && s.data[i]<='9')     //数字字符
        {   s1.data[j]=s.data[i];
            j++;
        }
        else if ((s.data[i]>='a' && s.data[i]<='z') ||
            (s.data[i]>='A' && s.data[i]<='Z'))  //字母字符
        {   s2.data[k]=s.data[i];
            k++;
        }
        i++;
    }
    s1.length=j; s2.length=k;
}
```

（4）**解**：用 pre 和 *p* 指向链串 *s* 的两个连续结点（初始时 pre=*s*->next，*p*=pre->next）。当 *p* 不空时循环：当 pre->data≤*p*->data 时，pre 和 *p* 同步后移一个结点；否则返回 0。当所有字符都是递增排列时返回 1。对应的算法如下：

```
int Increase(LinkString *s)
{   LinkString *pre=s->next,*p;
    if (pre==NULL) return 1;    //空串是递增的
    p=pre->next;
    if (p==NULL) return 1;      //含一个字符的串是递增的
    while (p!=NULL)
    {   if (pre->data<=p->data)//pre、p两个结点递增
        {   pre=p;              //pre、p同步后移
            p=p->next;
        }
        else                    //逆序时返回0
            return 0;
    }
    return 1;
}
```

（5）**解**：用 maxcount 存放串 *s* 中最长等值子串的长度（初始为 0）最大平台长度。用 *p* 扫描串 *s*，计算以 *p* 结点开始的等值子串的长度 count（该等值子串尾结点的后继结点为 post），若 count 大于 maxcount，则置 maxcount 为 count。再置 *p*=post 继续查找下一个等值子串直到 *p* 为空。对应的算法如下：

```
int Maxlength(LinkString *s)
{   int count,maxcount=0;
    LinkString *p=s->next,*post;
    while (p!=NULL)
    {   count=1;
        post=p->next;
```

```
            while (post!=NULL && post->data==p->data)
            {   count++;
                post=post->next;
            }
            if (count>maxcount) maxcount=count;
            p=post;
        }
        return maxcount;
}
```

5. 练习题 5 参考答案

1. 单项选择题

（1）B	（2）C	（3）B	（4）C	（5）C
（6）A	（7）B	（8）B	（9）D	（10）B
（11）D	（12）B	（13）C	（14）B	（15）D

2. 填空题

（1）$(d_1-c_1+1)\times(d_2-c_2+1)\times(d_3-c_3+1)$

（2）LOC(A[0][0])+$(n\times i+j)\times k$

（3）326

（4）1208

（5）42

（6）$i(i-1)/2+j$。

（7）行下标、列下标和元素值

3. 简答题

（1）**答：**数组的主要基本运算如下：

① 取值运算：给定一组下标，读取其对应的数组元素。

② 赋值运算：给定一组下标，存储或修改与其相对应的数组元素。

（2）**答：**从逻辑结构的角度看，一维数组是一种线性表；二维数组可以看成数组元素为一维数组的一维数组，所以二维数组是线性结构，可以看成是线性表，但就二维数组的形状而言，它又是非线性结构，因此将二维数组看成是线性表的推广更准确。三维及以上维的数组亦如此。

（3）**答：**因为数组使用链式结构存储时需要额外占用更多的存储空间，而且不具有随机存取特性，使得相关操作更复杂。

（4）**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*[]={1，1，1，5，5，5，5，3，3，3，3，4，4，4，4，4，4}，共有 17 个元素，对应的压缩存储 *B* 如下：

```
{{1，3}，{5，4}，{3，4}，{4，6}}
```

压缩数组 *B* 中仅有 8 个整数。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. 算法设计题

（1）**解**：从前向后找为零的元素 *A*[*i*]，从后向前找非零的元素 *A*[*j*]，将 *A*[*i*]和 *A*[*j*]进行交换。对应的算法如下：

```
void move(ElemType A[]，int n)
{    int i=0，j=n-1;
     ElemType tmp;
     while (i<j)
     {    while (A[i]!=0) i++;    //从前向后找零元素A[i]
          while (A[j]==0) j--;    //从后向前找非零元素A[j]
          if (i<j)                //A[i]与A[j]交换
          {    tmp=A[i]; A[i]=A[j];
               A[j]=tmp;
          }
     }
}
```

（2）**解**：当参数错误时算法返回 0；否则先置 mini=*i*，*k* 从 *i*+1 的 *j* 循环：比较将最小元素的下标放在 mini 中。对应的算法如下：

```
int Minij(int a[],int n,int i,int j,int &mini)
{    int k;
     if (i<0 || j>=n || i>j || j>=n)
          return 0;     //参数错误返回0
     mini=i;
     for (k=i+1;k<=j;k++)
          if (a[k]<a[mini])
               mini=k;
     return 1;          //成功找到返回1
}
```

（3）**解**：用 sum 记录 *a* 中下三角和主对角部分的所有元素和（初始为 0），*i* 从 0 到 *n*−1、*j* 从 0 到 *i* 循环，执行 sum+=*a*[*i*][*j*]，最后返回 sum。对应的算法如下：

```
int LowDiag(int a[][N],int n)
{    int i,j,sum=0;
     for (i=0;i<n;i++)
          for (j=0;j<=i;j++)
               sum+=a[i][j];
     return sum;
}
```

（4）**解：**对于二维数组 A，左上—右下对角线元素 $a[i][j]$满足 $i==j$，左下—右上对角线元素 $a[i][j]$满足 $i+j==n-1$。i 从 0 到 $n-1$、j 从 0 到 $n-1$ 循环，输出满足条件的元素值。对应的算法如下：

```
void Output(int a[][N],int n)
{    int i,j;
     for (i=0;i<n;i++)
     {    for (j=0;j<n;j++)
               if (i==j || i+j==n-1)
                    printf("%5d",a[i][j]);
               else
                    printf("     ");
          printf("\n");
     }
}
```

6. 练习题 6 参考答案

1. 单项选择题

（1）C （2）①A ②A ③C （3）C （4）A （5）C
（6）D （7）B （8）C （9）D （10）D
（11）B （12）B （13）C （14）D （15）D
（16）B （17）C （18）B （19）B （20）C

2. 填空题

（1）3
（2）$n-3$
（3）5
（4）①31 ②32
（5）350
（6）①500 ②499 ③1
（7）2^{i-1}
（8）① 2^{h-1} ② $2^{h}-1$ ③ 2^{h-1}。

（9）①a 结点最左下结点 ②a 结点的最右下结点。

（10）33

3. 简答题

（1）**答：**度为 2 的树中某个结点只有一个孩子时，不区分左右孩子，而二叉树中某个结点只有一个孩子时，严格区分是左孩子还是右孩子。一棵度为 2 的树至少有 3 个结点，而一棵二叉树的结点个数可以为 0。

（2）**答：**在该二叉树中，n_0=60，$n_2=n_0-1=59$，$n=n_0+n_1+n_2=119+n_1$，当 n_1=0 且为完全二叉树时高度最小，此时高度 $h=\lceil \log_2(n+1) \rceil = \lceil \log_2 120 \rceil = 7$。所以含有 60 个叶子结点的二叉树的最小高度是 7。

（3）**答：**由二叉树的性质可知，$n_2=n_0-1$。设这样的完全二叉树中结点数为 n，其中度为 1 的结点数 n_1 至多为 1，所以 $n=n_0+(n_0-1)+1=2n_0$ 或 $n=2n_0-1$。

（4）**答：**按层序编号时，完全二叉树的结点编号为 1～n，如果已知各类结点个数，该完全二叉树的形态一定是确定的。

若 n 已知，则可以根据其奇偶性确定 n_1：当 n 为偶数，n_1=1，当 n 为奇数，n_1=0，而 $n_0=n_2+1$，$n=n_0+n_1+n_2=2n_0-1+n_1$，$n_0=(n-n_1+1)/2$，从而 n_0 和 n_2 也确定了，所以这样的完全二叉树的形态就确定了。

（5）**答：**设度为 0、1、2 的结点个数及总结点数分别为 n_0、n_1、n_2 和 n，则有：

n_0=50，$n=n_0+n_1+n_2$，度之和=$n-1=n_1+2\times n_2$

由以上三式可得：n_2=49。

这样 $n=n_1+99$，所以当 n_1=0 时，n 最少，因此 n 至少有 99 个结点。

（6）**答：**① 该二叉树如图 6.1 所示。

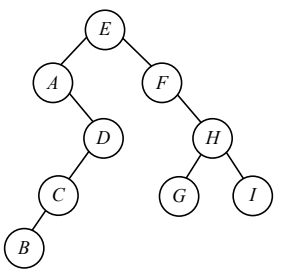


图 6.1　一棵二叉树

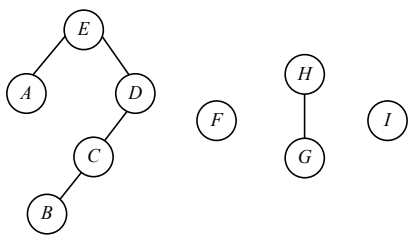


图 6.2　二叉树还原成的森林

② 结点 D 的双亲结点为结点 A，其左子树为以 C 为根结点的子树，其右子树为空。

③ 由此二叉树还原成的森林如图 6.2 所示。

（7）答：由这些显示部分推出二叉树如图 6.3 所示。则先序序列为 *ABDFKICEHJG*；中序序列为 *DBKFIAHEJCG*；后序序列为 *DKIFBHJEGCA*。

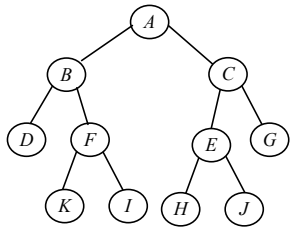


图 6.3　一棵二叉树

（8）**答**：二叉树的先序序列是 *NLR*（*N* 代表根结点，*L*、*R* 分别代表左右子树），后序序列是 *LRN*。要使 *NLR*=*LRN* 成立，则 *L* 和 *R* 均为空，所以满足条件的二叉树只有一个根结点。

（9）**答**：不是。如 *n*=3，该二叉树的先序序列为 1、2、3，它的中序序列不可能是 3、1、2，如果是的话，由先序序列可知 1 为根结点，由中序序列求出 1 的左孩子结点为 3，右孩子结点为 2，而先序序列中紧跟 1 的是结点 2，这样无法由先序和中序构造出一棵二叉树，说明该中序序列是错误的。

实际上，若先序遍历序列为 1、2、…、*n*，中序序列是 1～*n* 的出栈序列时，可以构造出一棵唯一的二叉树。

（10）**答**：一棵哈夫曼树中只有度为2和0的结点，没有度为1的结点，由非空二叉树的性质1可知，$n_0=n_2+1$，即$n_2=n_0-1$，则总结点数$n=n_0+n_2=2n_0-1$。

4. 算法设计题

（1）**解**：由二叉树顺序存储结构特点，可得到以下求离 *i* 和 *j* 的两个结点最近的公共祖先结点的算法：

```
ElemType Ancestor(ElemType A[], int i, int j)
{    int p=i, q=j;
     while (p!=q)
          if (p>q) p=p/2;
          else q=q/2;
     return A[p];
}
```

（2）**解**：先序遍历树中结点的递归算法如下：

```
void PreOrder1(ElemType A[], int i, int n)
{    if (i<n)
     {    if (A[i]!='#')                    //不为空结点时
          {    printf("%c ", A[i]);         //访问根结点
               PreOrder1(A, 2*i, n);        //遍历左子树
               PreOrder1(A, 2*i+1, n);      //遍历右子树
          }
     }
}
```

（3）**解**：求一个二叉树中的最大结点值的递归模型如下：

f(bt)=bt->data　　只有一个结点时

f(bt)=MAX{*f*(bt->lchild)，*f*(bt->rchild)，bt->data}　　其他情况

对应的算法如下：

```
ElemType MaxNode(BTNode *bt)
{    ElemType max, max1, max2;
     if (bt->lchild==NULL && bt->rchild==NUL)
          return bt->data;
     else
     {    max1=MaxNode(bt->lchild);    //求左子树的最大结点值
          max2=MaxNode(bt->rchild);    //求右子树的最大结点值
          max=bt->data;
          if (max1>max) max=max1;
          if (max2>max) max=max2;      //求最大值
          return(max);                 //返回最大值
     }
}
```

（4）**解**：设 Findx(*b*，*x*)用于返回二叉树 *b* 中值为 *x* 的结点地址。当 *b* 为空时返回 NULL。若当前 *b* 所指结点值为 *x*，返回 *b*；递归调用 *p*=Findx(*b*->lchild，*x*)在左子树中查找值为 *x* 的结点地址 *p*，若 *p* 不为空，表示找到了，返回 *p*；否则递归调用 Findx(*b*->rchild，*x*)在右子树中查找值为 *x* 的结点地址，并返回其结果。对应的算法如下：

```
BTNode *Findx(BTNode *b,char x)
{    BTNode *p;
     if (b==NULL)
          return NULL;
     else
     {    if (b->data==x)
               return b;
          p=Findx(b->lchild,x);
          if (p!=NULL)
               return p;
          return Findx(b->rchild,x);
     }
}
```

（5）**解**：设 *f*(*b*，*x*)返回二叉树 *b* 中所有结点值为 *x* 的结点个数，其递归模型如下：

f(*b*，*x*) = 0　　*b*=NULL

f(*b*，*x*) = 1+ *f*(*b*->lchild，*x*)+*f*(*b*->rchild，*x*);　　当 *b*->data=*x*

f(*b*，*x*) = *f*(*b*->lchild，*x*)+*f*(*b*->rchild，*x*);　　其他情况

对应的算法如下：

```
int FindCount(BTNode *b,char x)
{   int n,nl,nr;
    if (b==NULL)
        return 0;
    if (b->data==x) n=1;
    else n=0;
    nl=FindCount(b->lchild,x);
    nr=FindCount(b->rchild,x);
    return n+nl+nr;
}
```

（6）**解**：设计 PrintLNodes(*b*)算法用于从左到右输出二叉树 *b* 中的所有叶子结点。当 *b* 为空时返回。当 *b* 所指结点为叶子结点时输出 *b*->data 值；递归调用 PrintLNodes(*b*->lchild)输出左子树中叶子结点值，递归调用 PrintLNodes(*b*->rchild)输出右子树中叶子结点值。对应的算法如下：

```
void PrintLNodes(BTNode *b)
{   if (b!=NULL)
    {   if (b->lchild==NULL && b->rchild==NULL)
            printf("%c ",b->data);
        PrintLNodes(b->lchild);
        PrintLNodes(b->rchild);
    }
}
```

（7）**解**：设 *f*(b1，b2)用于判断两个二叉树 b1 和 b2 是否相同，对应的递归模型如下：

f(b1，b2)=1　　当b1、b2均为空

f(b1，b2)=0　　当b1、b2中一个为空，另一个不为空

f(b1，b2)=0　　当b1->data≠b2->data

f(b1，b2)=*f*(b1->lchild，b2->lchild) & *f*(b1->rchild，b2->rchild)　　其他情况

对应的算法如下：

```
int Same(BTNode *b1,BTNode *b2)
{   if (b1==NULL && b2==NULL)
        return 1;
    else if (b1==NULL || b2==NULL)
        return 0;
    else
    {   if (b1->data!=b2->data)
            return 0;
```

```
        return Same(b1->lchild,b2->lchild) & Same(b1->rchild,b2->rchild);
    }
}
```

（8）**解**：哈夫曼树 b 的带权路径长度（WPL）等于所有叶子结点权值乘以层次的总和，对应的算法如下：

```
void WPL1(HNode *b,int h,int &sum)
{   if (b->lchild==NULL && b->rchild==NULL)
        sum+=b->w*h;        //叶子结点时累计WPL
    WPL1(b->lchild,h+1,sum);
    WPL1(b->rchild,h+1,sum);
}
int WPL(HNode *b)           //求解算法
{   int sum=0;
    WPL1(b,1,sum);
    return sum;
}
```

7. 练习题 7 参考答案

1. 单项选择题

（1）C	（2）B	（3）C	（4）C	（5）A
（6）D	（7）A	（8）C	（9）A	（10）A
（11）B	（12）B	（13）A	（14）D	（15）D
（16）A	（17）D	（18）C	（19）B	（20）C
（21）A	（22）A	（23）C	（24）D	（25）D
（26）D	（27）A	（28）D	（29）A	（30）C

2. 填空题

（1）$n-1$

（2）①邻接矩阵　②邻接表

（3）45

（4）$2(n-1)$

（5）$O(n^2)$

（6）$O(n+e)$

（7）①$O(n^2)$　②$O(n+e)$

（8）①$O(n^2)$　②$O(n+e)$

（9）m

（10）$n-1$

（11）极小连通子图

（12）①O(n^2)　②O($e\log_2 e$)　③Kruskal　④Prim

（13）4 和 5。

（14）0，1，2，5，3，4

（15）①递增　②负值

3. 简答题

（1）**答：**设图的顶点个数和边数分别为 n 和 e。邻接矩阵的存储空间大小为 O(n^2)，与 e 无关，因此适合于稠密图的存储。邻接表的存储空间大小为 O($n+e$)（有向图）或 O($n+2e$)（无向图），与 e 有关，因此适合于稀疏图的存储。

（2）**答：**图的这两种遍历算法对无向图和有向图都适用。

但如果无向图不是连通的，调用一次遍历算法只能访问一个连通分量中的所有顶点。如果有向图不是强连通的，调用一次遍历算法可能不能访问图中全部顶点。

（3）**答：**图G对应的邻接矩阵为：

$$A=\begin{bmatrix}0&1&1&1&0\\1&0&1&0&1\\1&1&0&1&1\\1&0&1&0&1\\0&1&1&1&0\end{bmatrix}$$

图G对应的邻接表如图7.8所示，对于该邻接表，从顶点0出发的深度优先遍历和广度优先遍历序列都是0、1、2、3、4。

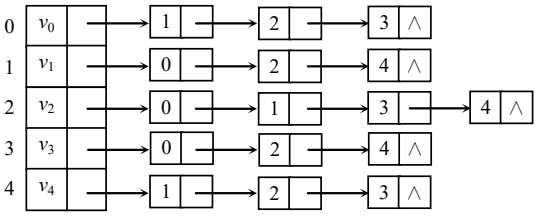


图 7.8　一个邻接表

（4）**答：**DFSTrav(G，v)算法是在顶点 v 的所有出边相邻点均访问后才输出该顶点的编号。从顶点 0 出发的深度优先遍历过程是：0→1，1→2，2 回退到 1，1→3，3→4，依次回退到 0。首先输出 2（最先没有出边相邻点），接着是 4（第 2 个没有出边相邻点），再是 3、1、0。

实际上，对于无环有向图，该算法输出序列的反序恰好是一个拓扑序列。

（5）**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)}。

（6）**答：**求解过程如下：

S
{0}

dist						
0	1	2	3	4	5	6
0	∞	∞	∞	11	∞	7

path						
0	1	2	3	4	5	6
0	−	−1	−1	0	−1	0

{0,6}	0	**22**	∞	**13**	11	∞	7
{0,6,4}	0	22	∞	13	11	∞	7
{0,6,4,3}	0	22	∞	13	11	**16**	7
{0,6,4,3,5}	0	22	**25**	13	11	16	7
{0,6,4,3,5,1}	0	22	25	13	11	16	7
{0,6,4,3,5,1,2}	0	22	25	13	11	16	7

	1					
0	**6**	−1	**6**	0	−1	0
0	6	−1	6	0	−1	0
0	6	−1	6	0	**3**	0
0	6	**5**	6	0	3	0
0	6	5	6	0	3	0
0	6	5	6	0	3	0

求解结果如下：

从 0 到 1 的最短路径长度为:22路径为:0，6，1

从 0 到 2 的最短路径长度为:25路径为:0，6，3，5，2

从 0 到 3 的最短路径长度为:13路径为:0，6，3

从 0 到 4 的最短路径长度为:11路径为:0，4

从 0 到 5 的最短路径长度为:16路径为:0，6，3，5

从 0 到 6 的最短路径长度为:7 路径为:0，6

（7）**答：**不一定。如图 7.9 所示的图 *G* 从顶点 0 出发的最小生成树如图 7.10 所示，而从顶点 0 到顶点的 2 的最短路径为 0→2，而不是最小生成树中的 0→1→2。

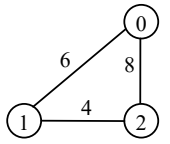


图 7.9 一个带权连通无向图

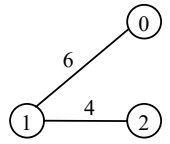


图 7.10 图的一棵最小生成树

（8）**答：**因为 Dijkstra 算法是一种贪心算法，没有回溯能力，在求出部分最短路径后，假设后面的顶点的最短路径更长，一旦出现权值为负的边，就有可能修改前面的已求出最短路径，而 Dijkstra 算法做不到这一点。

如图 7.11 所示，求以顶点 0 为源点的最短路径，先选取顶点 1，表示从顶点 0 到顶点 1 的最短路径长度为 2，以后不再改变，然后选取顶点 2，因为<2，1>的权值为负，需要修改 0→1 的最短路径为 0→2→1，而 Dijkstra 算法没有这种回溯修改路径的能力，所以要求所有边上的权值必须大于 0。

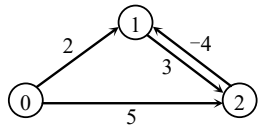


图 7.11 一个带权有向图

（9）**答：**这样的有向图的拓扑序列是唯一的，入度为 0 的顶点只有一个，每次输出一个入度为 0 的顶点后，剩余的图中都只有一个入度为 0 顶点。

（10）**答：**① 图 *G* 的邻接矩阵 *A* 如图 7.12 所示。

② 有向带权图 *G* 如图 7.13 所示。

③ 图 7.14 中粗线所标识的 4 个活动组成图 G 的关键路径。

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

图 7.12 邻接矩阵 A

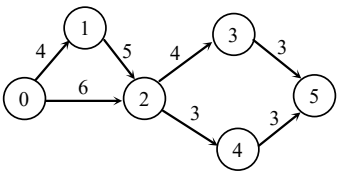


图 7.13 图 G

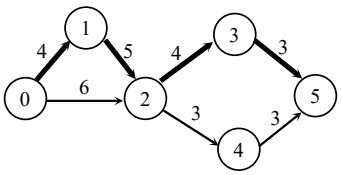


图 7.14 图 G 中的关键路径

（11）**答**：深度优先遍历序列为：A，B，C，E，G，D，F。

求最早开始时间和最迟开始时间的过程如下：

$ve(A)=0$

$ve(D)=3$

$ve(F)=ve(D)+3=6$

$ve(B)=1$

$ve(C)=\text{MAX}\{ve(B)=3，ve(A)+2，ve(F)+3\}=7$

$ve(E)=\text{MAX}\{ve(B)+1，ve(C)+2\}=9$

$ve(G)=\text{MAX}\{\text{ve}(E)+1，ve(F)+5\}=11$。

$vl(G)=11$

$vl(E)=vl(G)-1=10$

$vl(C)=vl(E)-2=8$

$vl(B)=\text{MIN}\{vl(E)-1，vl(C)-3\}=5$

$vl(F)=\text{MIN}\{vl(G)-5，vl(C)-1\}=6$

$vl(D)=vl(F)-3=3$

$vl(A)=\text{MIN}\{vl(B)-1，vl(C)-2，vl(D)-3\}=0$。

则 $l(FC)=vl(C)-1=7$，所以，事件 C 的最早开始时间为 7，活动 FC 的最迟开始时间为 7。

4. 算法设计题

（1）**解**：用 sum 累计出度为 0 的顶点数（初始为 0）。扫描邻接矩阵 g.edges，对于顶点 i，累计第 i 行中非零元素个数即为出度，若为零则 sum++。最后返回 sum。对应的算法如下：

```
int ZeroOutDs(MatGraph g)          //计算图G中出度为0的顶点数
```

```
{    int i,j,n,sum=0;
     for (i=0;i<g.n;i++)
     {    n=0;
          for (j=0;j<g.n;j++)
               if (g.edges[i][j]!=0 && g.edges[i][j]!=INF)
                    n++;          //存在i到j的一条边时
          if (n==0) sum++;
     }
     return sum;
}
```

（2）**解**：对于顶点 *v*，累计 *G*->adjlist[*v*]为头结点的单链表中结点个数即顶点 *v* 的出度。对应的算法如下：

```
int OutDsv(AdjGraph *G,int v)    //求顶点v的出度
{    int n=0;
     ArcNode *p;
     p=G->adjlist[v].firstarc;
     while (p!=NULL)             //扫描边表结点
     {    n++;                   //累计出边的数目
          p=p->nextarc;
     }
     return n;
}
void OutDs(AdjGraph *G)          //求解算法
{    for (int i=0;i<G->n;i++)
          printf("顶点%d的出度:%d\n",i,OutDsv(G,i));
}
```

（3）**解**：扫描邻接表，对于每个顶点 *i*，累计 *G*->adjlist[*i*]为头结点的单链表中 adjvex 为 *v* 的结点个数，最终结果为顶点 *v* 的入度。对应的算法如下：

```
int InDsv(AdjGraph *G,int v)          //求顶点v的入度
{    int i,n=0;
     ArcNode *p;
     for (i=0;i<G->n;i++)
     {    p=G->adjlist[i].firstarc;
          while (p!=NULL)             //扫描边表结点
          {    if (p->adjvex==v) n++; //累计顶点v的入边的数目
               p=p->nextarc;
          }
     }
     return n;
}
void InDs(AdjGraph *G)                //求解算法
```

```
{    for (int i=0;i<G->n;i++)
         printf("顶点%d的入度:%d\n",i,InDsv(G,i));
}
```

（4）**解**：若以 *G*->adjlist[*i*]为头结点的单链表中存在 adjvext 为 *j* 的结点，表示顶点 *i* 到顶点 *j* 有边，否则无边。对应的算法如下；

```
int Arc(ALGraph *G,int i,int j)  //判断图G中是否存在边<i,j>
{    ArcNode *p;
     p=G->adjlist[i].firstarc;
     while (p!=NULL && p->adjvex!=j)
          p=p->nextarc;
     if (p==NULL)
          return 0;              //不存在i到j的边
     else
          return 1;              //存在i到j的边
}
```

上述算法的时间复杂度为 O(*m*)，其中 *m* 表示邻接表中单链表结点个数的最大值。采用邻接矩阵存储图时实现该功能的时间复杂度为 O(1)。

（5）**解**：若非连通无向图依顶点次序由 G_1、G_2、…、G_k 连通分量构成的，则依次对 G_1、G_2、…、G_k 调用 DFS(G_i，*i*)算法。调用 DFS 的次数即为连通分量数。对应的算法如下：

```
int visited[MAXVEX]={0};             //全局变量，所有元素置初值0
void DFS(AdjGraph *G,int v)          //深度优先遍历算法
{    ArcNode *p;
     visited[v]=1;                   //置已访问标记
     p=G->adjlist[v].firstarc;       //p指向顶点v的第一个相邻点
     while (p!=NULL)
     {    if (visited[p->adjvex]==0)  //若p->adjvex顶点未访问,递归访问它
               DFS(G,p->adjvex);
          p=p->nextarc;              //p指向顶点v的下一个相邻点
     }
}
int Getnum(AdjGraph *G)              //求图G的连通分量
{    int i,count=0;                  //count累计连通分量个数
     for (i=0;i<G->n;i++)
          if (visited[i]==0)
          {    DFS(G,i);             //从顶点i出发深度优先遍历
               count++;              //调用DFS的次数即为连通分量数
          }
     return count;
}
```

（6）**解：**若非连通无向图依顶点次序由 G_1、G_2、…、G_k 连通分量构成的，则依次对 G_1、G_2、…、G_k 调用 BFS(G_i，i)算法。调用 BFS 的次数即为连通分量数。对应的算法如下：

```
int visited[MAXVEX]={0};                    //全局变量，所有元素置初值0
void BFS(AdjGraph *G,int v)                 //广度优先遍历算法
{   ArcNode *p;
    int qu[MAXVEX],front=0,rear=0;          //定义循环队列并初始化队头队尾
    int w;
    visited[v]=1;                           //置已访问标记
    rear=(rear+1)%MAXVEX;
    qu[rear]=v;                             //v进队
    while (front!=rear)                     //若队列不空时循环
    {   front=(front+1)%MAXVEX;
        w=qu[front];                        //出队并赋给w
        p=G->adjlist[w].firstarc;           //找顶点w的第一个相邻点
        while (p!=NULL)
        {   if (visited[p->adjvex]==0)      //若当前邻接顶点未被访问
            {   visited[p->adjvex]=1;       //置该顶点已被访问的标志
                rear=(rear+1)%MAXVEX;       //该顶点进队
                qu[rear]=p->adjvex;
            }
            p=p->nextarc;                   //找顶点w的下一个相邻点
        }
    }
}
int Getnum(AdjGraph *G)                     //求图G的连通分量
{   int i,count=0;                          //count累计连通分量个数
    for (i=0;i<G->n;i++)
        if (visited[i]==0)
        {   BFS(G,i);                       //从顶点i出发广度优先遍历
            count++;                        //调用BFS的次数即为连通分量数
        }
    return count;
}
```

（7）**解：**先置全局数组 visited[]所有元素为 0，然后从顶点 i 开始进行广度优先遍历。遍历结束之后，若 visited[j]=0，说明顶点 i 到顶点 j 没有路径，返回 0；否则说明顶点 i 到顶点 j 有路径，返回 1。对应的算法如下：

```
int visited[MAXVEX]={0};                    //全局变量，所有元素置初值0
void BFS(AdjGraph *G,int v)                 //广度优先遍历算法
{   ArcNode *p;
    int qu[MAXVEX],front=0,rear=0;          //定义循环队列并初始化队头队尾
```

```
    int w;
    visited[v]=1;                          //置已访问标记
    rear=(rear+1)%MAXVEX;
    qu[rear]=v;                            //v进队
    while (front!=rear)                    //若队列不空时循环
    {   front=(front+1)%MAXVEX;
        w=qu[front];                       //出队并赋给w
        p=G->adjlist[w].firstarc;          //找顶点w的第一个相邻点
        while (p!=NULL)
        {   if (visited[p->adjvex]==0)     //若当前邻接顶点未被访问
            {   visited[p->adjvex]=1;      //置该顶点已被访问的标志
                rear=(rear+1)%MAXVEX;      //该顶点进队
                qu[rear]=p->adjvex;
            }
            p=p->nextarc;                  //找顶点w的下一个相邻点
        }
    }
}
int BFSTrave(AdjGraph *G,int i,int j)      //求解算法
{   int k;
    for (k=0;k<G->n;k++) visited[k]=0;
    BFS(G,i);                              //从顶点i出发广度优先遍历
    if (visited[j]==0)
        return 0;
    else
        return 1;
}
```

（8）**解**：用 sum 累计 DFS 走过的边数（初始为 0），先置全局数组 visited[]所有元素为 0，然后从顶点 v 开始进行深度优先遍历，先访问顶点 v，置 visited[v]=1，查找顶点 v 的一个尚未访问的相邻点 w，从顶点 w 出发继续深度优先遍历，这里走了边<v，w>，所以置 sum++。对应的算法如下：

```
int visited[MAXVEX]={0};                   //全局变量，所有元素置初值0
void DFSEdges1(AdjGraph *G,int v,int &sum)//深度优先遍历算法
{   ArcNode *p;
    visited[v]=1;                          //置已访问标记
    p=G->adjlist[v].firstarc;              //p指向顶点v的第一个相邻点
    while (p!=NULL)
    {   if (visited[p->adjvex]==0)         //若p->adjvex顶点未访问,递归访问它
        {   sum++;                         //走v到p->adjvex的一条边
            DFSEdges1(G,p->adjvex,sum);
        }
        p=p->nextarc;                      //p指向顶点v的下一个相邻点
```

```
    }
}
int DFSEdges(AdjGraph *G,int v)            //求解算法
{   int sum=0;
    DFSEdges1(G,v,sum);
    return sum;
}
```

8. 练习题 8 参考答案

1. 单项选择题

（1）B　　（2）C　　（3）A　　（4）D

（5）A。查找范围为 R[0..9]=　　（4，6，10，12，20，30，50，70，88，100），mid=(0+9)/2=4，58>R[4]（20）。查找范围为 R[5..9]，mid=(5+9)/2=7，58<R[7]（70）。查找范围为 R[5..6]，mid=(5+6)/2=5，58>R[5]（30）。查找范围为 R[6..6]，mid=(6+6)/2=6，58>R[6]（60），查找失败。

（6）C。查找失败最多关键字比较次数=$\lceil \log_2(n+1) \rceil = \lceil \log_2 23 \rceil = 5$。

（7）D。成功时最大比较次数为$\lceil \log_2(n+1) \rceil = \lceil \log_2 101 \rceil = 7$。

（8）D。在折半查找的判定树中第 i 层的结点个数最多为 2^{i-1}。

（9）A　　（10）D

（11）A。在二叉排序树中关键字最小的结点是根结点的最左下结点，一定没有左孩子。

（12）B

（13）C。该二叉排序树的中序序列为 *abcdefgh*，选项 A、B 和 D 都不能与该中序序列构造出正确的二叉树，只有选项 C 可以。

（14）B　　（15）C　　（16）A

（17）B。设 N_h 表示高度为 h 的平衡二叉树中含有的最少结点数，有 $N_1=1$，$N_2=2$，$N_h=N_{h-1}+N_{h-2}+1$，由此，求出 $N_5=12$。

（18）C。A 结点的左子树是平衡的，所以调整选择右孩子 B，而右孩子 B 的平衡因子为 1，说明其左子树高，调整应选择 B 的左孩子 C，这样 A、B、C 三个结点构成 RL 调整。

（19）D

（20）B。m=3，结点关键字个数 1～2 个，最少结点的情况是每个结点只有 1 个关键字，此时类似一棵高度为 5 的满二叉树，其结点个数=$2^h-1=31$。

（21）A　　（22）C　　（23）D

（24）D。这 k 个同义词 R_1～R_k 在哈希表是依次相邻排列的，存入 R_i 需要进行的探测次数为 i，总的探测次数=$1+2+\cdots+k=k(k+1)/2$。

（25）C

2. 填空题

（1）①n　②n

（2）①1　②2　③4　④8　⑤5　⑥3.7

（3）①2　②4　③3。

（4）①索引表　②主数据表。

（6）n+1

（7）(n+1)/2。这样的二叉排序树是一个右单支树，此时查找退化为顺序查找，关键字的平均比较次数是(n+1)/2。

（8）①3 ②24 ③{13} ④{37，53，90}

（9）31

（10）①存取元素时发生冲突的可能性就越大②存取元素时发生冲突的可能性就越小

3. 简答题

（1）**答**：三种方法对查找的要求分别如下。

① 顺序查找法：表中元素可以任意次序存放。适合顺序表和链表的存储结构。

② 折半查找法：表中元素必须按关键字有序减排列，且更适合顺序表存储结构。

③　分块查找法：表中元素每块内的元素可以任意次序存放，但块与块之间必须以关键字的大小递增（或递减）排列，即前一块内所有元素的关键字都不能大（或小）于后一块内任何元素的关键字。

三种方法的平均查找长度分别如下。

① 顺序查找法：查找成功的平均查找长度为(n+1)/2。

② 折半查找法：查找成功的平均查找长度为 $\log_2(n+1)-1$。

③　分块查找法：若用顺序查找确定所在的块，平均查找长度为(b+s)/2+1；若用折半查找确定所在块，平均查找长度约为 $\log_2(b+1)+s/2$。其中，s 为每块含有的元素个数，b 为块数。

（2）**答**：折半查找不适合链表结构的序列。虽然有序单链表的结点是按关键字有序排列的，但难以确定查找的区间（对应的时间为 O(n)），故不适合进行折半查找。

在一般情况下折半查找的效率更高（所需要的关键字比较次数更少），但在特殊情况下未必如此，例如查找第一个元素时，或者查找的元素个数很少时，顺序查找可能更快。

（3）**答**：①　按输入顺序构造的二叉排序树如图 8.2 所示。在等概率情况下查找成功的平均查找长度为：

$$ASL_{succ}=\frac{1+2+3+4+5+6+7+8}{8}=4.5$$

② 构造的一棵平衡二叉树如图8.3所示。在等概率情况下查找成功的平均查找长度：

$$ASL_{succ}=\frac{1+2\times2+3\times4+5\times1}{8}=2.75$$

由此可见在同样序列的查找中，平衡二叉树比二叉排序树的平均查找长度要小，查找效率更高。

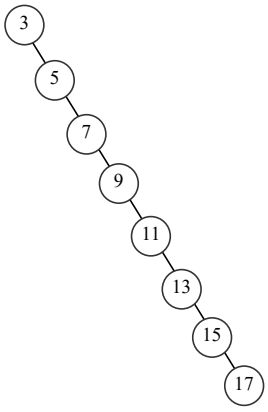


图 8.2　一棵二叉排序树

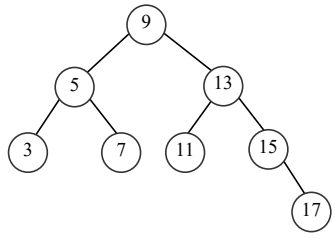


图 8.3 一棵平衡二叉树

（4）**答：**可以。由二叉排序树的先序序列可以确定其结点个数，将所有结点值递增排序构成其中序序列，由先序序列和中序序列可以唯一构造该二叉排序树。

（5）**答：**构造的二叉排序树如图 8.4 所示。为了删除结点 72，在其左子树中找到最大结点 54（只有一个结点），用其代替结点 72。删除之后的二叉排序树如图 8.5 所示。

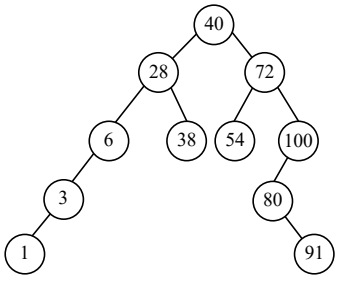


图 8.4　二叉排序树

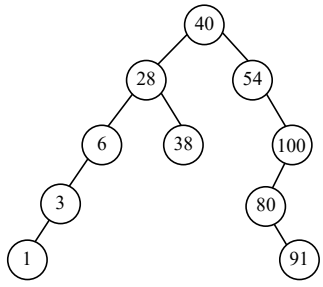
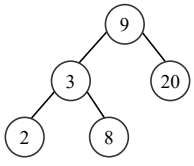
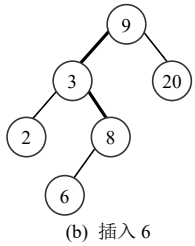


图 8.5　删除 72 后的二叉排序树

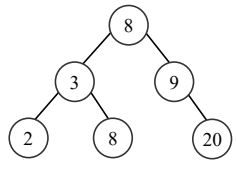
（6）**答：**插入关键字为6和10的两个结点，其调整过程如图8.6所示。



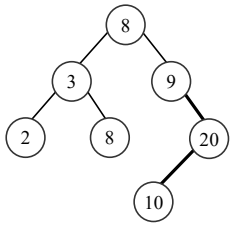
(a) AVL 树



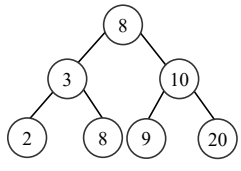
(b) 插入 6



(c) LR 调整



(d) 插入 10



(e) RL 调整

图 8.6　插入两个结点，AVL 树调整过程

（7）**答**：依题意，m=19，线性探测法计算下一地址计算公式为：

$d_0=h(\text{key})$

$d_j=(d_{j-1}+1)\ \%\ m \quad j=1，2，...$

构造哈希表 ha 的过程如下：

$h(19)=19\ \%\ 13=6$	将 19 存放在 ha[6]中
$h(01)=01\ \%\ 13=1$	将 01 存放在 ha[1]中
$h(23)=23\ \%\ 13=10$	将 23 存放在 ha[10]中
$h(14)=14\ \%\ 13=1$	冲突
$d_0=1，d_1=(1+1)\ \%\ 19=2$	将 14 存放在 ha[2]中
$h(55)=55\ \%\ 13=3$	将 55 存放在 ha[3]中
$h(20)=20\ \%\ 13=7$	将 20 存放在 ha[7]中
$h(84)=84\ \%\ 13=6$	冲突
$d_0=6，d_1=(6+1)\ \%\ 19=7$	仍冲突
$d_2=(7+1)\ \%\ 19=8$	将 84 存放在 ha[8]中
$h(27)=27\ \%\ 13=1$	冲突
$d_0=1，d_1=(1+1)\ \%\ 19=2$	仍冲突
$d_2=(2+1)\ \%\ 19=3$	仍冲突
$d_3=(3+1)\ \%\ 19=4$	将 27 存放在 ha[4]中
$h(68)=68\ \%\ 13=3$	冲突
$d_0=3，d_1=(3+1)\ \%\ 19=4$	仍冲突
$d_2=(4+1)\ \%\ 19=5$	将 68 存放在 ha[5]中
$h(11)=11\ \%\ 13=11$	将 11 存放在 ha[11]中
$h(10)=10\ \%\ 13=10$	冲突
$d_0=10，d_1=(10+1)\ \%\ 19=11$	仍冲突
$d_2=(11+1)\ \%\ 19=12$	将 10 存放在 ha[12]中
$h(77)=77\ \%\ 13=12$	冲突
$d_0=12，d_1=(12+1)\ \%\ 19=13$	将 77 存放在 ha[13]中

因此，构建的哈希表 ha 如表 8.1 所示。

表 8.1　哈希表 ha

下标	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
k		01	14	55	27	68	19	20	84		23	11	10	77					
探测次数		1	2	1	4	3	1	1	3		1	1	3	2					

$ASL_{成功}=(1+2+1+4+3+1+1+3+1+1+3+2)/12=1.92$

ASL $_{不成功}$=(1+9+8+7+6+5+4+3+2+1+5+4+3+2+1+1+1+1+1)/19=3.42

（8）**答**：依题意，得到：

h(87)=87 % 13=9

h(25)=25 % 13=12

h(310)=310 % 13=11

h(08)=08 % 13=8

h(27)=27 % 13=1

h(132)=132 % 13=2

h(68)=68 % 13=3

h(95)=95 % 13=4

h(187)=187 % 13=5

h(123)=123 % 13=6

h(70)=70 % 13=5

h(63)=63 % 13=11

h(47)=47 % 13=8

采用拉链法处理冲突的哈希表如图 8.7 所示。成功查找的平均查找长度：

ASL $_{成功}$=(1×10+2×3)/10=1.6

ASL $_{不成功}$=(1×7+2×3)/13=1

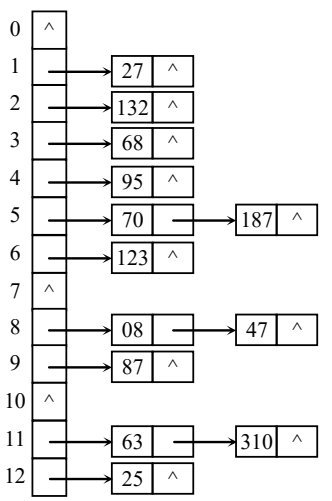


图 8.7　采用拉链法处理冲突的哈希表

4. 算法设计题

（1）**解**：通过一趟扫描并比较，可以找出最大元素 max 和最小元素 min。对应的算法如下：

```
void MaxMin(int A[], int n, int &min, int &max)
{    int i;
     min=max=A[0];
```

```
    for (i=1;i<n;i++)
        if (R[i]<min)
            min=R[i];
        else if (R[i]>max)
            max=R[i];
}
```

（2）**解**：对应的递归算法如下：

```
int BinSearch1(SqType R[], KeyType k, int low, int high)
{   int mid;
    if (low>high)
        return(-1);
    else
    {   mid=(low+high)/2;
        if (k==R[mid].key)
            return(mid);
        else if (k>R[mid].key)
            return(BinSearch1(R, k, mid+1, high));//在左子树中递归查找
        else
            return(BinSearch1(R, k, low, mid-1));//在右子树中递归查找
    }
}
```

（3）**解**：对应的两个算法如下：

```
void Findk1(SqType R1[],int n,KeyType k)  //无序顺序表R1的查找
{   int i=0;
    while (i<n)
    {   if (R1[i].key==k)
            printf("%d ",i);
        i++;
    }
}
void Findk2(SqType R2[],int n,KeyType k)  //递增有序顺序表R2的查找
{   int i=0;
    while (i<n)
    {   if (R2[i].key==k)
            printf("%d ",i);
        else if (k<R2[i].key)
            break;
        i++;
    }
}
```

无序顺序表 R1 中的查找是从头到尾进行的。递增有序顺序表 R2 中的查找是从头开

始，当遇到大于 k 的元素时退出循环。因为 Findk1 算法不成功时总是要比较所有的元素，所以不成功查找的平均查找长度=n。而 Findk2 算法可以在任何元素上（只要该元素的关键字大于 k）退出，所以不成功查找的平均查找长度=$(1+2+\cdots+n)/n=(n+1)/2$。

（4）**解**：按中序序列遍历二叉排序树即按递增次序遍历，对应值的算法如下：

```
void incrorder(BSTNode *bt)
{   if (bt!=NULL)
    {   incrorder(bt->lchild);
        printf("%d ", bt->data);
        incrorder(bt->rchild);
    }
}
```

（5）**解**：由二叉排序树的性质可知，右子树中所有结点值大于根结点值，左子树中所有结点值小于根结点值。为了从大到小输出，要先遍历右子树，再访问根结点，后遍历左子树。对应的算法如下：

```
void Output(BSTNode *bt, KeyType k)
{   if (bt!=NULL)
    {   Output(bt->rchild, k);
        if (bt->key>=k)
            printf("%d ", bt->key);
        Output(bt->lchild, k);
    }
}
```

（6）**解**：设二叉排序树采用二叉链存储结构。采用二叉排序树非递归查找算法，用 h 保存查找层次。对应的算法如下：

```
int level(BSTNode *bt, KeyType k)
{   int h=0;
    if (bt!=NULL)
    {   h++;
        while (bt->data!=k)
        {   if (k<bt->data)
                bt=bt->lchild;          //在左子树中查找
            else
                bt=bt->rchild;          //在右子树中查找
            h++;                        //层数增1
        }
        return h;
    }
}
```

9. 练习题 9 参考答案

1. 单项选择题

（1）C　（2）D　（3）C　（4）C　（5）B
（6）D　（7）D　（8）C　（9）C　（10）A
（11）C　（12）D　（13）A　（14）D　（15）C
（16）C　（17）B　（18）D　（19）C　（20）C
（21）C　（22）A　（23）C　（24）B　（25）B

2. 填空题

（1）①比较 ②移动
（2）①0　②$n-1$
（3）2。在插入 60 时，有序区为（4，12，23，48，96），依次与 96、48 进行比较，比较次数为 2。
（4）①$O(n^2)$　②$O(n^2)$
（5）（1，2，3，5，4，7，9，8，6，10）
（6）①直接插入 ②简单选择排序
（7）快速排序
（8）①堆排序 ②快速排序
（9）①$O(n\log_2 n)$ ②$O(n)$
（10）$\lceil \log_2 n \rceil$
（11）（16，12，15，10，5，7，2，8）。堆中删除操作总是删除根结点。
（12）某个叶子结点
（13）①k_1 ②递增
（14）①生成初始归并段　②对初始归并段采用多路归并方法归并为一个有序段。
（15）3

3. 简答题

（1）**答：**希尔排序过程如图 9.1 所示。

```
排序前:4,5,1,2,8,6,7,3,10,9
gap=5: 4,5,1,2,8,6,7,3,10,9
gap=2: 1,2,4,3,7,5,8,6,10,9
gap=1: 1,2,3,4,5,6,7,8,9,10
排序后:1,2,3,4,5,6,7,8,9,10
```

图 9.1 希尔排序各趟排序结果

（2）**答：**该数组一定构成一个堆，递增有序数组构成一个小根堆，递减有序数组构

成一个大根堆。

以递增有序数组为例，假设数组元素为 k_1、k_2、…、k_n 是递增有序的，从中看出下标越大的元素值也越大，对于任一元素 k_i，有 $k_i<k_{2i}$，$k_i<k_{2i+1}$（$i<n/2$），这正好满足小根堆的特性，所以构成一个小根堆。

（3）**答**：采用冒泡排序法排序的各趟的结果如下：

初始序列:	75	23	98	44	57	12	29	64	38	82
i=0(归位元素:12):	12	75	23	98	44	57	29	38	64	82
i=1(归位元素:23):	12	23	75	29	98	44	57	38	64	82
i=2(归位元素:29):	12	23	29	75	38	98	44	57	64	82
i=3(归位元素:38):	12	23	29	38	75	44	98	57	64	82
i=4(归位元素:44):	12	23	29	38	44	75	57	98	64	82
i=5(归位元素:57):	12	23	29	38	44	57	75	64	98	82
i=6(归位元素:64):	12	23	29	38	44	57	64	75	82	98

（4）**答**：采用快速排序法排序的各趟的结果如下：

初始序列:	75	23	98	44	57	12	29	64	38	82
区间:0～9 基准:75:	38	23	64	44	57	12	29	75	98	82
区间:0～6 基准:38:	29	23	12	38	57	44	64	75	98	82
区间:0～2 基准:29:	12	23	29	38	57	44	64	75	98	82
区间:0～1 基准:12:	12	23	29	38	57	44	64	75	98	82
区间:4～6 基准:57:	12	23	29	38	44	57	64	75	98	82
区间:8～9 基准:98:	12	23	29	38	44	57	64	75	82	98

（5）**答**：采用直接选择法排序的各趟的结果如下：

初始序列:	75	23	98	44	57	12	29	64	38	82
i=0 归位元素:12	12	23	98	44	57	75	29	64	38	82
i=1 归位元素:23	12	23	98	44	57	75	29	64	38	82
i=2 归位元素:29	12	23	29	44	57	75	98	64	38	82
i=3 归位元素:38	12	23	29	38	57	75	98	64	44	82
i=4 归位元素:44	12	23	29	38	44	75	98	64	57	82
i=5 归位元素:57	12	23	29	38	44	57	98	64	75	82
i=6 归位元素:64	12	23	29	38	44	57	64	98	75	82
i=7 归位元素:75	12	23	29	38	44	57	64	75	98	82
i=8 归位元素:82	12	23	29	38	44	57	64	75	82	98

（6）**答**：采用堆排序法排序的各趟的结果如下：

初始序列:	75	23	98	44	57	12	29	64	38	82
初始堆:	98	82	75	64	57	12	29	44	38	23
归位元素:98 调整成堆[1..9]:	82	64	75	44	57	12	29	23	38	98
归位元素:82 调整成堆[1..8]:	75	64	38	44	57	12	29	23	82	98
归位元素:75 调整成堆[1..7]:	64	57	38	44	23	12	29	75	82	98
归位元素:64 调整成堆[1..6]:	57	44	38	29	23	12	64	75	82	98

归位元素:57 调整成堆[1..5]: 44 29 38 12 23 57 64 75 82 98
归位元素:44 调整成堆[1..4]: 38 29 23 12 44 57 64 75 82 98
归位元素:38 调整成堆[1..3]: 29 12 23 38 44 57 64 75 82 98
归位元素:29 调整成堆[1..2]: 23 12 29 38 44 57 64 75 82 98
归位元素:23 调整成堆[1..1]: 12 23 29 38 44 57 64 75 82 98

（7）**答**：采用二路归并排序法排序的各趟的结果如下：

初始序列: 75 23 98 44 57 12 29 64 38 82
length=1: 23 75 44 98 12 57 29 64 38 82
length=2: 23 44 75 98 12 29 57 64 38 82
length=4: 12 23 29 44 57 64 75 98 38 82
length=8: 12 23 29 38 44 57 64 75 82 98

（8）**答**：采用基数排序法排序的各趟的结果如下：

初始序列: 503，187，512，161，908，170，897，275，653，462
按个位排序结果: 170，161，512，462，503，653，275，187，897，908
按十位排序结果: 503，908，512，653，161，462，170，275，187，897
按百位排序结果: 161，170，187，275，462，503，512，653，897，908

（9）**答**：采用堆排序方法，建立初始堆时间为 $4n$，每次选取一个最小元素后再筛选的时间为 $\log_2 n$，找前 10 个最小的元素的时间=$4n+9\log_2 n$。而冒泡排序和简单选择排序需 $10n$ 的时间。而直接插入排序、希尔排序和二路归并排序等必须全部排好序后才能找前 10 个最小的元素，显然不能采用。

（10）**答**：① 总的归并趟数=$\lceil \log_4 11 \rceil$=2。

② m=11，k=4，(m−1) % (k−1)=1≠0，需要附加 k−1−(m−1) % (k−1)=2 个长度为 0 的虚归并段，最佳归并树如图 9.2 所示。

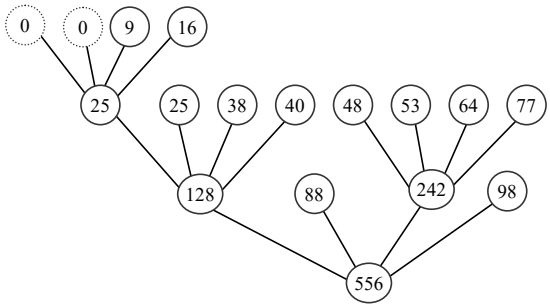


图 9.2 最佳归并树

③ 根据最佳归并树计算每一趟及总的读元素数：

第 1 趟的读元素数=9+16=25
第 2 趟的读元素数=25+25+38+40+48+53+64+77=370
第 3 趟的读元素数=128+88+242+98=556
总的读元素数=25+370+556+951。

4. 算法设计题

（1）**解**：只需将原直接插入排序（递增排序）算法中的关键字比较“>”改为“<”即可。对应的算法如下：

```
void InsertSort(SqType R[],int n)//对R[0..n-1]按递减有序进行直接插入排序
{   int i,j;
    SqType tmp;
    for (i=1;i<n;i++)           //从第二个元素即R[1]开始
    {   if (R[i-1].key<R[i].key)
        {   tmp=R[i];           //取出无序区的第一个元素
            j=i-1;              //从右向左在有序区R[0..i-1]中找R[i]的插入位置
            do
            {   R[j+1]=R[j];    //将关键字小于tmp.key的元素后移
                j--;            //继续向前比较
            } while (j>=0 && R[j].key<tmp.key);
            R[j+1]=tmp;         //在j+1处插入R[i]
        }
    }
}
```

（2）**解**：按照题目要求设计的直接插入排序算法如下：

```
void InsertSort(SqType R[],int n)
{   int i,j,k;
    SqType tmp;
    for (i=n-2;i>=0;i--)
    {   tmp=R[i];
        j=i+1;
        while (j<n && tmp.key>R[j].key)
            j++;                //在有序区找到一个刚大于tmp.key的位置R[j]
        for (k=i;k<j-1;k++)     //R[i..j-1]元素前移,以便腾出一个位置插入tmp
            R[k]=R[k+1];
        R[j-1]=tmp;             //在j-1位置处插入tmp
    }
}
```

（3）**解**：按照题目要求设计的折半插入排序算法如下：

```
void BinInsertSort(SqType R[],int n) //对R[0..n-1]按递增有序进行折半插入排序
{   int i,j,low,high,mid;
    SqType tmp;
    for (i=n-2;i>=0;i--)
    {   tmp=R[i];                   //将R[i]保存到tmp中
        low=i+1;high=n-1;
        while (low<=high)           //在R[low..high]中折半查找有序插入的位置
```

```
        {   mid=(low+high)/2;       //取中间位置
            if (tmp.key<R[mid].key)
                high=mid-1;         //插入点在左半区
            else
                low=mid+1;          //插入点在右半区
        }
        for (j=i;j<high;j++)        //元素前移
            R[j]=R[j+1];
        R[high]=tmp;                //插入R[i]
    }
}
```

（4）**解：**只需将原冒泡排序算法中的 $R[0..n-1]$的排序区间改为 $R[i..j]$即可。对应的算法如下：

```
void BubbleSort(SqType R[],int i,int j)
{   int i1,j1,exchange;
    SqType tmp;
    for (i1=i;i1<j;i1++)
    {   exchange=0;
        for (j1=j;j1>i1;j1--)   //比较,找出最小关键字的元素
            if (R[j1].key<R[j1-1].key)
            {   tmp=R[j1];    //R[j1]与R[j1-1]进行交换,将最小关键字元素前移
                R[j1]=R[j1-1]; R[j1-1]=tmp;
                exchange=1;
            }
        if (!exchange)          //本趟没有发生交换，中途结束算法
            return;
    }
}
```

（5）**解：**只需将原简单选择排序算法中的 $R[0..n-1]$的排序区间改为 $R[i..j]$即可。对应的算法如下：

```
void SelectSort(SqType R[],int i,int j)
{   int i1,j1,k;
    SqType tmp;
    for (i1=i;i1<j;i1++)            //做第i1趟排序
    {   k=i1;
        for (j1=i1+1;j1<=j;j1++)    //在当前无序区R[i1..j]中选key最小的R[k]
            if (R[j1].key<R[k].key)
                k=j1;               //k记下目前找到的最小关键字所在的位置
        if (k!=i1)                  //交换R[i]和R[k]
        {   tmp=R[i1];
            R[i1]=R[k]; R[k]=tmp;
```

```
        }
    }
}
```

（6）**解**：当数据个数 n 为偶数时，最后一个分支结点（编号为 $n/2$）只有左孩子（编号为 n），其余分支结点均为双分支结点；当 n 为奇数时，所有分支结点均为双分支结点。对每个分支结点进行判断，只有一个分支结点不满足小根堆的定义，返回 0；如果所有分支结点均满足小根堆的定义，返回 1。对应的算法如下：

```
int IsHeap(SqType R[],int n)          //判断R[1..n]是否为小根堆
{   int i;
    if (n%2==0)                       //n为偶数时
    {   if (R[n/2].key>R[n].key)      //最后分支结点(编号为n/2)只有左孩子(编号为n)
            return 0;
        for (i=n/2-1;i>=1;i--)        //判断所有双分支结点
            if (R[i].key>R[2*i].key || R[i].key>R[2*i+1].key)
                return 0;
    }
    else                              //n为奇数时
    {   for (i=n/2;i>=1;i--)          //所有分支结点均为双分支结点
            if (R[i].key>R[2*i].key || R[i].key>R[2*i+1].key)
                return 0;
    }
    return 1;                         //满足小根堆的定义,返回1
}
```