附录　习题参考答案

第 1 章　习题参考答案

一、选择题

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

二、填空题

1. 线性结构　　非线性结构
2. 顺序存储结构　　链式存储结构
3. 联系　图（或图结构）
4. 集合、图
5. 一对一　　一对多
6. 数据元素　　关系
7. 后继　任意多个
8. 操作对象　　关系
9. 没有　没有
10. 逻辑结构　　存储结构
11. 前驱　　1
12. 顺序　　索引

三、判断题

1. ×	2. √	3. ×	4. ×

四、简答题

1. 因为 x++共执行了 n-1+n-2+……＋1= n(n-1)/2，所以执行时间为 O（n^2）
2. O（$\log_3^n$）　　3. O（n^2）　　4. O（m*n）

第 2 章　习题参考答案

一、选择题

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

二、填空题

1. 首元素 其直接前驱结点的链域的值　　2. HL→next==NULL；HL==HL→next

3. 有限、一对一　　4. O(1)　　随机存取

5. 表中一半　　表长和该元素在表中的位置　　6. 必定　不一定

7. O（1）　O（n）　　8. 前驱结点的地址　　O（n）

9. n-i+1　n-i　　10. s->left=p　　p->right

三、判断题

1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.
×	×	×	×	×	×	√	×	×	×	×

四、简答题

1. 线性表为：（78，50，40，60，34，90）

2. (36, 12, 8, 50, 25, 5, 15)

3. 解答：

尾指针是指向终端结点的指针，用它来表示单循环链表可以使得查找链表的开始结点和终端结点都很方便，设一带头结点的单循环链表，其尾指针为 rear，则开始结点和终端结点的位置分别是 rear->next->next 和 rear, 查找时间都是 O(1)。

若用头指针来表示该链表，则查找终端结点的时间为 O(n)。

五、编程题

1. 解答：由于在单链表中只给出一个头指针，所以只能用遍历的方法来数单链表中的结点个数了。算法如下：

```
int ListLength ( LinkList L )
{    int len=0 ;
     ListNode *p;
     p=L; //设该表有头结点
     while ( p->next )
     {    p=p->next;
          len++;
     }
     return len;
}
```

2. int searchmin(linklist l)

```
{    int min;
     int *p;
     p=l;
     min=p->data;
     p=p->next;
     while (p->next!=NULL)
     {      if (min>p->data) min=p->data;
            p=p->next;
     }
     return min;
```

```
}
 3. int   searchmax(linklist l)
{       int max;
        int *p;
        p=l;
        max=p->data;
        p=p->next;
        while (p->next!=NULL)
        {   if (max<p->data) max=p->data;
            p=p->next;
        }
        return max;
}
```

4. 顺序表：要将该表逆置，可以将表中的开始结点与终端结点互换，第二个结点与倒数第二个结点互换，如此反复，就可将整个表逆置了。

算法如下：

```
void ReverseList( Seqlist *L)
{       DataType temp ; //设置临时空间用于存放 data
        int i;
        for (i=0;i<=L->length/2;i++)//L->length/2 为整除运算
        {    temp = L->data[i]; //交换数据
             L -> data[ i ] = L -> data[ L -> length-1-i];
             L -> data[ L -> length - 1 - i ] = temp;
        }
}
 5. int CountX(LNode* HL,ElemType x)
{       int i=0; LNode* p=HL;//i 为计数器
        while(p!=NULL)
        {    if (P->data==x) i++;
             p=p->next;
        }//while, 出循环时 i 中的值即为 x 结点个数
        return i;
}//CountX
```

6. 解答：因已知顺序表 L 是递增有序表，所以只要从顺序表终端结点（设为 i 位置元素）开始向前寻找到第一个小于或等于 x 的元素位置 i 后插入该位置即可。

在寻找过程中，由于大于 x 的元素都应放在 x 之后，所以可边寻找，边后移元素，当找到第一个小于或等于 x 的元素位置 i 时，该位置也空出来了。

算法如下：

```
void InsertIncreaseList( Seqlist *L , Datatype x )
{    int i;
     if(L->length>=ListSize)
          Error("overflow");
     for(i=L->length-1;i>0 && L->data[i-1]>x;i--)
```

```
        L->data[i+1]=L->data[i]; // 比较并移动元素
    L->data[i] =x;
    L->length++;
    }
```

第 3 章　习题参考答案

一、选择题

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

二、填空题

1. 队列　　2. FILO　　FIFO
3. 栈顶　栈底　　4. 前一个位置/当前位置；　n-1
5. 尾　首　　6. 队尾　队头
7. 线性表　后进先出表　　8. s->next=h->next　　h-next=s;
9. q->front->next=NULL　　q->rear=q->front;　　10. stack

三、判断题

1. ×	2. √	3. √	4. ×	5. √	6. √	7. √	8. √	9. ×	10. ×

四、简答题

1. 顺序队列的结构:

```
typedef struct
{    datatype data[maxsize]
     int front,rear;
}sequeue;
sequeue *sq;
```

2. 解答：链栈不需要在头部附加头结点，因为栈都是在头部进行操作的，如果加了头结点，等于要对头结点之后的结点进行操作，反而使算法更复杂，所以只要有链表的头指针就可以了。

3. (1)出栈序列为：1324

(2)不能得到 1423 序列。因为要得到 14 的出栈序列，则应做 Push(1)，Pop()，Push(2)，Push(3)，Push(4)，Pop()。这样，3 在栈顶，2 在栈底，所以不能得到 23 的出栈序列。能得到 1432 的出栈序列。具体操作为：Push(1)，Pop()，Push(2)，Push(3)，Push(4)，Pop()，Pop()，Pop()。

4. 用队列长度计算公式：(N＋r－f)% N

① L=（40＋19－11）% 40=8　　② L=（40＋11－19）% 40=32

5. 答：该算法的功能是：利用堆栈做辅助，将队列中的数据元素进行逆置。

6. 栈是仅允许在一端进行插入和删除的线性表，又称为后进先出表。

队列是允许在一端插入，在另一端删除的线性表，允许插入的一端的称为队尾，允许删除的一端称为队头，又称为先进先出表。

7. 顺序栈的结构:

```
typedef int    datatype;
#define maxsize 64
typedef struct
{     datatype data[maxsize];
      int top
}seqstack;
seqstack *s;
```

8. 解答：循环队列的优点是：它可以克服顺序队列的"假上溢"现象，能够使存储队列的向量空间得到充分的利用。判别循环队列的"空"以头尾指针是否相等来确定；判别循环队列的"满"通过少用一个元素的空间，每次入队前测试入队后头尾指针是否会重合，如果会重合就认为队列已满。

9. 队满条件:　(q->rear+1)% m==q->front

队空条件:　q->front==q=>rear

10. 解答：算法如下

```
void ClearStack (SeqStack *S)
{ //  删除栈中所有结点
      S->top = -1; //其实只是将栈置空
}
```

因为要置空的是栈 S，如果不用指针来做参数传递，那么函数进行的操作不能对原来的栈产生影响，系统将会在内存中开辟另外的单元来对形参进行函数操作。结果等于什么也没有做。所以想要把函数操作的结果返回给实参的话，就只能用指针来做参数传递了。

五、编程题

```
 1. linkstack *POPSTACK(linkstack *HS,datatype *data)
{     linkstack *p
      if(HS==NULL){printf("under flow");    return NULL;)}
      else
         {    *data=HS->data;
              p=HS;
              HS=HS->next;
              free(p);
              return HS;
         }
}
 2. int ENQUEUE(sequeue *sq,datatype x)
{         if(sq->front==(sq->rear+1)%maxsize)
          {     printf("queue is full"); return NULL;
```

```
        }
        else
        {    sq->rear=(sq->rear+1)%maxsize;
             sq->data[sq->rear]=x;     return TRUE ;
        }
}
 3. datatype DEQUEUE(sequeue *sq)
{
     if(sq->front==sq->rear)
     {    printf("queue is empty");
     }
     else
     {   sq->front=(sq->front+1)%maxsize;
         return(sq->data[sq->front];
     }
}
 4. 解答：根据题意，可定义该循环队列的存储结构：
#define QueueSize 100
typedef char Datatype ; //设元素的类型为 char 型
typedef struct
{       int quelen;
        int rear;
        Datatype Data[QueueSize];
}CirQueue;
CirQueue *Q;
循环队列的队满条件是：Q->quelen==QueueSize
知道了尾指针和元素个数，当然就能计算出队头元素的位置。算法如下：
(1)判断队满
int FullQueue( CirQueue *Q)
{//判队满,队中元素个数等于空间大小
        return Q->quelen==QueueSize;
}
(2)入队
void EnQueue( CirQueue *Q, Datatype x)
{       if(FullQueue( Q))
             Error("队已满，无法入队");
        Q->Data[Q->rear]=x;
        Q->rear=(Q->rear+1)%QueueSize;//在循环意义上的加 1
        Q->quelen++;
}
(3)出队
Datatype DeQueue( CirQueue *Q)
{       int tmpfront; //设一个临时队头指针
        if(Q->quelen==0)
```

```
        Error("队已空，无元素可出队");
    tmpfront=(QueueSize+Q->rear-Q->quelen+1)%QueueSize;//计算头指针位置
    Q->quelen--;
    return Q->Data[tmpfront];
}
```

第 4 章　习题参考答案

一、填空题

1. 元素值　　2. 3　3
3. p; (k,p,h);　　4. (a,b)　(b,c)
5. x　((x,y))　　6. ((c,d))　(b)
7. (a,b)　(c,d)　　8. b　（d）

二、选择题

1. A	2. D	3. D	4. B

第 5 章　习题参考答案

一、选择题

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

二、填空题

1. 前趋（双亲）　根　　2. 左右　不能
3. 子树个数　5　　4. 有序　N-1
5. 2^{h-1}　2^h-1　　6. 50　100
7. 4　16　　8. 后继　任意多个
9. 63　32　　10. A[2*i+1]　A[(i-1)/2]
11. 2n　n-1　　12. 二叉树
13. 500、499　　14. O(n)　350
15. n、$\log_k^n+1$　　16. i/2　2i+1
17. 1　33　　18. 9　5
19. （3^h-1）/2　　20. LRD、FEGHDCB
21. $n_1+n_2=0+ n_2= n_0-1=31$　　$2^{6-1}=32$

三、判断题

1. ×	2. ×	3. √	4. √	5. ×	6. √	7. √	8. ×	9. ×	10. √
11. √	12. √	13. ×	14. ×	15. √	16. ×	17. ×	18. ×	19. √	20. √
21. √	22. √	23. ×	24. ×	25. ×	26. √				

四、简答题

1. 此二叉树的后序遍历结果是：EDCBIHJGFA

2. 先序遍历序列为：ABCDFEGHIJKLM

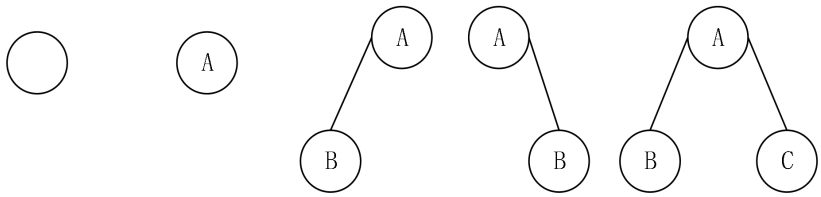
中序遍历序列为：CFDEBGJILMKHA

后序遍历序列为：FEDCJMLKIHGBA

3. 后序：16,24,42,57,53,35,88,84,92,60

4. 高度为 h 的完全二叉树至少有 2^{h-1} 个结点，至多有 2^h-1 个结点(也就是满二叉树)。

5. 有五种。

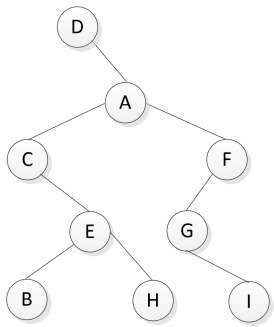


(a)空二叉树　(b)根节点　(c)左子树　(d)右子树　(e)完全二叉树

6. （1）3;2　（2）4;E,F,I,J,H　（3）A,C　（4）C,D;E,F

7. 度为 2 的树从形式上看与二叉树很相似，但它的子树是无序的，而二叉树是有序的。即，在一般树中若某结点只有一个孩子，就无需区分其左右次序，而在二叉树中即使是一个孩子也有左右之分。

8. 解：方法是：由前序先确定 root，由中序可确定 root 的左、右子树。然后由其左子树的元素集合和右子树的集合对应前序遍历序列中的元素集合，可继续确定 root 的左右孩子。将他们分别作为新的 root，不断递归，则所有元素都将被唯一确定，问题得解。

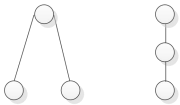


9. 答：DLR：A B D F J G K C E H I L M

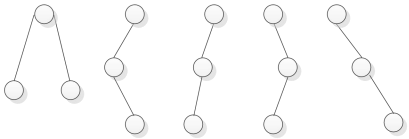
LDR:　B F J D G K A C H E L I M

LRD：J F K G D B H L M I E C A

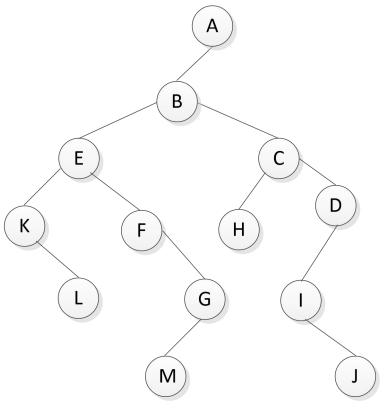
10. 三个结点的树有：

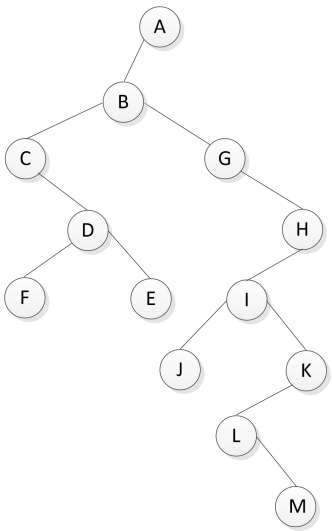


三个结点的二叉树有：



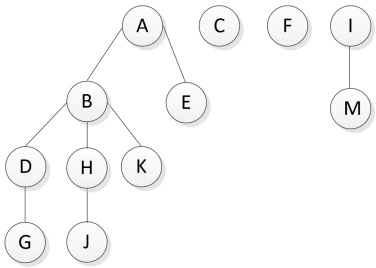
11. 注意全部兄弟之间都要连线（包括度为 2 的兄弟）,并注意原有连线结点一律归入左子树，新添连线结点一律归入右子树。



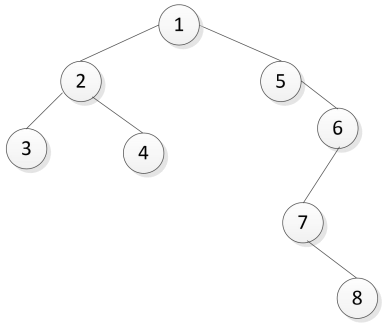


12.

13. 注意根右边的子树肯定是森林，而孩子结点的右子树均为兄弟。



14. （1）有 无左子树　　（2）有 只有根节点　　（3）有 无右子树

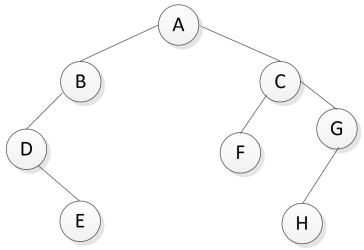


15.

16. 这是“先根再左再根再右”，比前序遍历多打印各结点一次，输出结果为：A B C C E E B A D F F D G G

特点：①每个结点肯定都会被打印两次；②但出现的顺序不同，其规律是：凡是有左子树的结点，必间隔左子树的全部结点后再重复出现；如 A，B，D 等结点。反之马上就会重复出现。如 C，E，F，G 等结点。

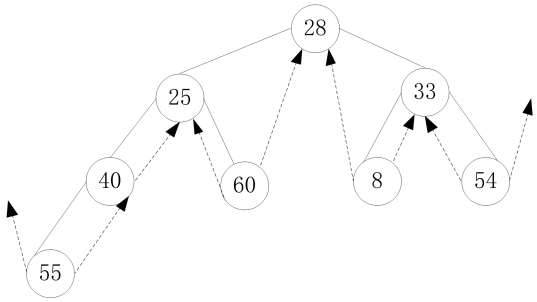
时间复杂度以访问结点的次数为主，精确值为 2*n，时间渐近度为 O(n)。



17.

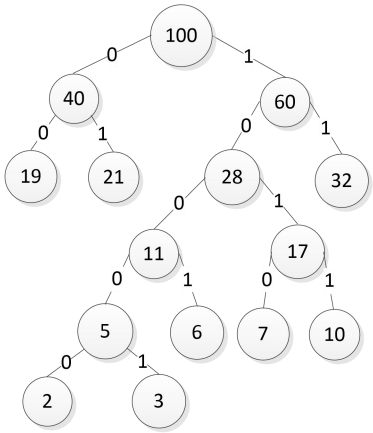
18. 解：要遵循中序遍历的轨迹来画出每个前驱和后继。

中序遍历序列：55　40　25　60　28　08　33　54



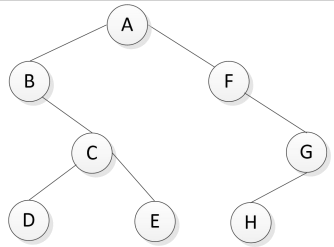
19. 解：先将概率放大 100 倍，以方便构造哈夫曼树。

w={7,19,2,6,32,3,21,10}，按哈夫曼规则：[(19,21),[[[(2,3), 6], (7,10)], 32]]



字母编号	对应编码	出现频率
1	1010	0.07
2	00	0.19
3	10000	0.02
4	1001	0.06
5	11	0.32
6	10001	0.03
7	01	0.21
8	1011	0.10

20.



五、编程题

```
 1. INORDER(bitree *t)
{    if(t)
     {    INORDER(t->lchild);
          printf("%c ",t->data);
          INORDER(t->rchild);
     }
}
 2.   BtreeDepth(BT->left)
     BtreeDepth(BT->right)
     dep1+1
     dep2+1
3. 求二叉树高度算法
int HIGH(bintree *tree)
```

```
{     int m=0;k=0;l=0
      if (tree==NULL) m=0
      else m=1
      if (tree->lchild!=NULL)
            k=HIGH(tree->lchild);
      if (tree->rchild!=NULL)
            l=HIGH(tree->rchild);
      if (k>l) m=k+1;
      else m=l+1
      return(m);
}
 4. typedef struct node
{       datatype data;
      struct node *lchild,*rchild;
} bitree;
int judgebitree(bitree *bt1,bitree *bt2)
{
   if (bt1==NULL && bt2== NULL) return(1);
   else if (bt1== NULL || bt2== NULL ||bt1->data!=bt2->data) return(0);
   else return(judgebitree(bt1->lchild,bt2->lchild)*judgebitree(bt1->rchild,bt2->rchild));
}
 5. void EXBINTREE(bintree *root)
{       bintree *t,*s;
      t=root;
      if(!t)return;
      s=t->rchild;
      t->rchild=t->lchild;
      t->lchild=s;
      EXBINTREE(t->lchild);
      EXBINTREE(t->rchild);
}
```

6. 思路：输出叶子结点比较简单，用任何一种遍历递归算法，凡是左右指针均空者，则为叶子，将其打印出来。

```
DLR(liuyu *root)        /*中序遍历     递归函数*/
{     if(root!=NULL)
      {     if((root->lchild==NULL)&&(root->rchild==NULL))
            {       sum++; printf("%d\n",root->data);}
            DLR(root->lchild);
            DLR(root->rchild);
      }
      return 0;
}
 7.    NodeLevel( BT->right, X)
       (c2>=1)    return c2+1
```

8. 按层遍历二叉树

```
LayerOrder(BiTree  root)
{    LinkQueue  Q;
     InitQueue(&Q);
     if(root==NULL)  return;
     EnterQueue(&Q,root);
     while(!Empty(&Q))
     {
          DelQueue(&Q,&p);
          visite(p->data);
          if(p->LChild!=NULL)
               EnterQueue(&Q,p->LChild);
          if(p->RChild!=NULL)
               EnterQueue(&Q,p->RChild);
     }
}
```

第 6 章　习题参考答案

一、选择题

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

二、填空题

1. 邻接矩阵、邻接表
2. 深度优先遍历、广度优先遍历
3. n(n-1)　n-1
4. 邻接表、广度优先遍历
5. 有向　将邻接矩阵的第 i 行全部置 0
6. 邻接表　邻接矩阵
7. 出度　n
8. $O(n^2)$,　O(n+e)
9. n, 2e
10. 克鲁斯卡尔(Kruskal)　普里姆(Prim)
11. $O(n^2)$　O(n+e)
12. 某一顶点　递增
13. 出度　入度
14. 相等　权
15. $O(n^2)$、　O(n+e)
16. $O(n^2)$、　O(n+e)

三、判断题

1. ×	2. ×	3. ×	4. ×	5. √	6. ×	7. √	8. √	9. ×	10. √
11. ×									

四、简答题

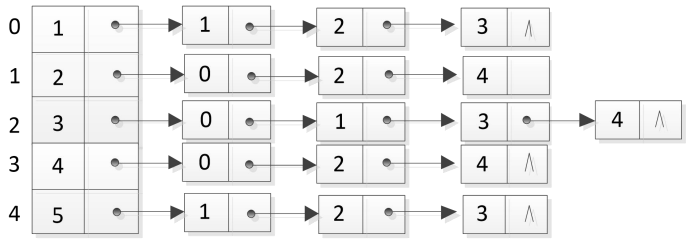
1. 用克鲁斯卡尔算法得到的最小生成树为：(1,2)3，(4,6)4，(1,3)5，(1,4)8，(2,5)10，

(4,7)20

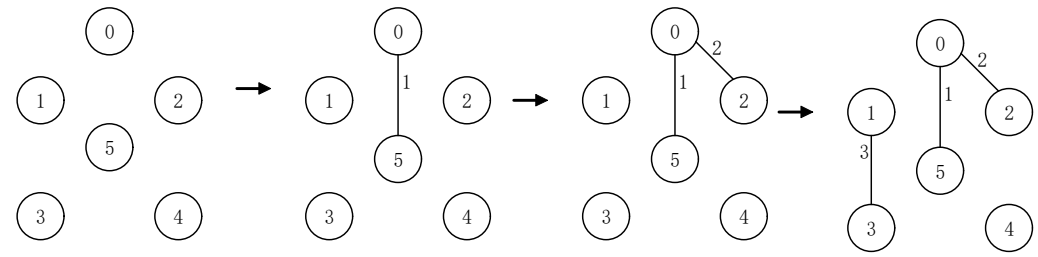
2. （1）DFS：①,ƒ„…　　（2）BFS：,ƒ„…①

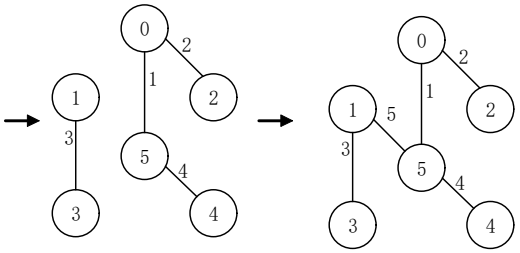
3. 邻接矩阵和邻接表：

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



4. 根据克鲁斯卡尔算法思想画图表示最小生成树的生成过程如下：





5.　(1)是有向图，因为其邻接矩阵是不对称的

(2)顶点 1 的入度为 1，出度为 2

顶点 2 的入度为 2，出度为 2

顶点 3 的入度为 1，出度为 2

顶点 4 的入度为 3，出度为 2

6.　顶点 1 的入度为 2，出度为 2

顶点 2 的入度为 2，出度为 2

顶点 3 的入度为 2，出度为 1

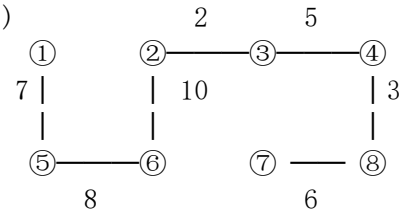
顶点 4 的入度为 1，出度为 2

7. (1)

$$\begin{vmatrix} 0 & 12 & \infty & \infty & 7 & \infty & \infty & \infty \\ 12 & 0 & 2 & \infty & \infty & 10 & \infty & \infty \\ \infty & 2 & 0 & 5 & \infty & \infty & 15 & \infty \\ \infty & \infty & 5 & 0 & \infty & \infty & \infty & 3 \\ 7 & \infty & \infty & \infty & 0 & 8 & \infty & \infty \\ \infty & 10 & \infty & \infty & 8 & 0 & 11 & \infty \\ \infty & \infty & 15 & \infty & \infty & 11 & 0 & 6 \end{vmatrix}$$

| ∞ ∞ ∞ 3 ∞ ∞ 6 0 |

(2)



8. 31

9. （1）深度优先搜索

顶点序列：1-2-3-4-5-6

边的序列：（1，2）（2，3）（3，4）（4，5）（5，6）

（2）广度优先搜索

顶点序列：1-2-3-6-5-4

边的序列：（1，2）（1，3）（1，6）（1，5）（5，4）

注：本题中所求深度优先序列和广度优先序列有多种，以上为其中一种。

10. (1) 顶点 1 的入度为 3，出度为 0

顶点 2 的入度为 2，出度为 2

顶点 3 的入度为 1，出度为 2

顶点 4 的入度为 1，出度为 3

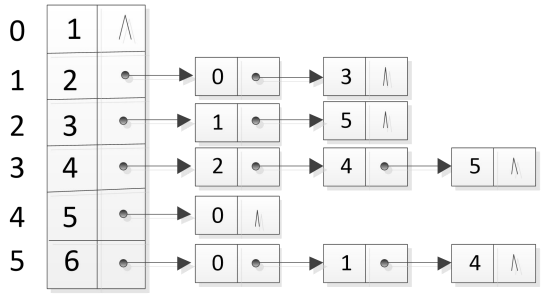
顶点 5 的入度为 2，出度为 1

顶点 6 的入度为 2，出度为 3

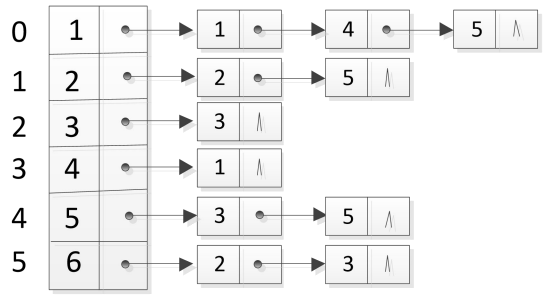
(2)邻接矩阵为：

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

(3)邻接表为：



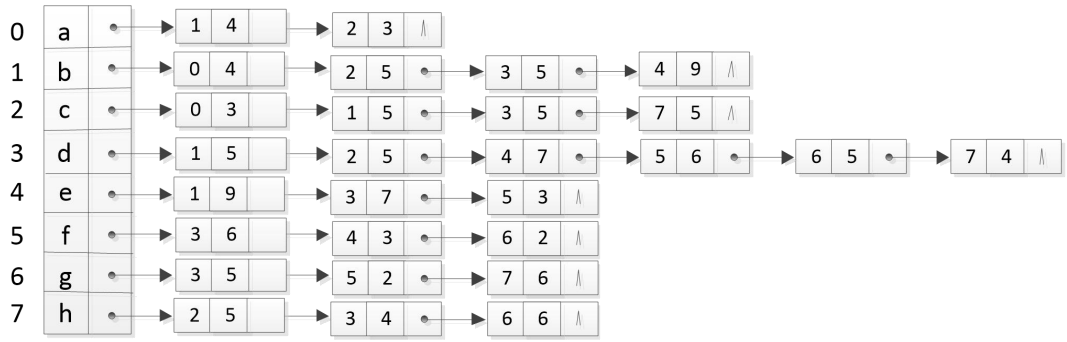
(4)逆邻接表为：



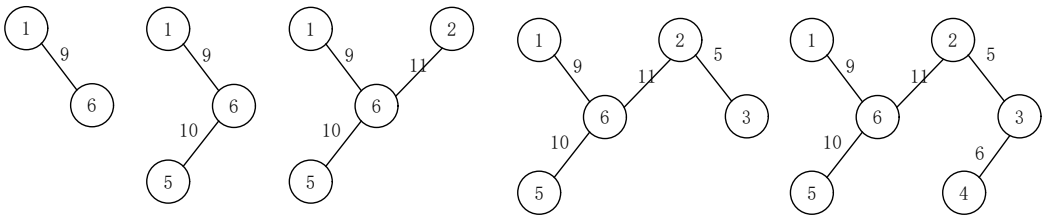
11.(1)邻接矩阵为：

$$\begin{bmatrix} 0 & 4 & 3 & \infty & \infty & \infty & \infty & \infty \\ 4 & 0 & 5 & 5 & 9 & \infty & \infty & \infty \\ 3 & 5 & 0 & 5 & \infty & \infty & \infty & 5 \\ \infty & 5 & 5 & 0 & 7 & 6 & 5 & 4 \\ \infty & 9 & \infty & 7 & 0 & 3 & \infty & \infty \\ \infty & \infty & \infty & 6 & 3 & 0 & 2 & \infty \\ \infty & \infty & \infty & 5 & \infty & 2 & 0 & 6 \\ \infty & \infty & 5 & 4 & \infty & \infty & 6 & 0 \end{bmatrix}$$

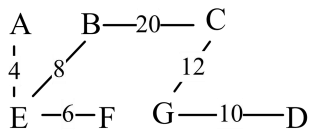
(2)邻接表为：



12. 解答：根据普里姆算法的基本思想，构造最小生成树，在此，由于边的权值出现了相同值，因此，在构造实际的最小生成树时结果是不唯一的，下图给出了其中的一种构造过程：



13. 解：(1) 最小生成树可直接画出，如下图所示。



(2)可用邻接矩阵和邻接表来描述：

$$\begin{bmatrix} \infty & 12 & \infty & \infty & 4 & \infty & \infty \\ 12 & \infty & 20 & \infty & 8 & 9 & \infty \\ \infty & 20 & \infty & 15 & \infty & \infty & 12 \\ \infty & \infty & 15 & \infty & \infty & \infty & 10 \\ 4 & 8 & \infty & \infty & \infty & 6 & \infty \\ \infty & 9 & \infty & \infty & 6 & \infty & \infty \\ \infty & \infty & 12 & 10 & \infty & \infty & \infty \end{bmatrix}$$

14. 拓扑序列为：1　4　0　2　5　7　3　6　8　9

15. 解：

终点 v_i	从 a 顶点出发到其余各顶点的最短路径 path[i]的变化情况						
	i=0	i=1	i=2	i=3	i=4	i=5	i=6
dist[a] path[a]	0						
dist[b] path[b]		15 <a,b>	15 <a,b>	15 <a,b>	15 <a,b>	15 <a,b>	15 <a,b>
dist[c] path[c]		2 <a,c>					
dist[d] path[d]		12 <a,d>	12 <a,d>	11 <a,c,f,d>	11 <a,c,f,d>		
dist[e] path[e]		∞	10 <a,c,e>	10 <a,c,e>			
dist[f] path[f]		∞	6 <a,c,f>				
dist[g] path[g]		∞	∞	16 <a,c,f,g>	16 <a,c,f,g>	15 <a,d,g>	
v_j		c	f	e	d	g	b
S	{a}	{a,c}	{a,c,f}	{a,c,f,e}	{a,c,f,e,d}	{a,c,f,e,d,g}	{a,c,f,e,d,g,b}

最短路径为：（a,c,f,e,d,g,b）或者（a,c,f,e,d,b,g）

16. 求顶点 0 其余各顶点的最短路径

终点 v_i	从 0 顶点出发到其余各顶点的最短路径 path[i]的变化情况				
	i=0	i=1	i=2	i=3	i=4
dist[0] path[0]	0				
dist[1] path[1]		10 <0,1>			
dist[2] path[2]		∞	60 <0,1,2>	50 <0,3,2>	
dist[3] path[3]		30 <0,3>	30 <0,3>		

dist[4]		100	100	90	60
path[4]		<0,4>	<0,4>	<0,3,4>	<0,3,2,4>
v_j		1	3	2	4
S	{0}	{0,1}	{0,1,3}	{0,1,3,2}	{0,1,3,2,4}

五、编程题

1. /*利用邻接矩阵实现图的遍历*/

```
/*从第 i 个顶点出发递归地深度优先遍历图 G*/
void DFS(Graph G,int i)
{    int j;
     printf("%3c",G.V[i]);                    /*访问第 i 个顶点*/
     visited[i]=1;
     for(j=0;j<G.n;j++)
          if((G.arcs[i][j]==1)&&(visited[j]==0))
               DFS(G,j);                    /*对顶点 i 的尚未访问的邻接顶点 j 递归调用 DFS */
}
/*【算法 6.10  对图 G 进行深度优先遍历】*/
```

2. /*【算法 6.13 用邻接表实现图的广度优先搜索遍历】*/

```
void BFS(ALGraph G,int k)
{    int i;
     ArcNode *p;
     InitQueue(&Q);                         /*初始化队列*/
     printf("%3c",G.adjList[k].data);       /*访问第 k 个顶点*/
     visited[k]=1;
     EnQueue(&Q,k);                         /*第 k 个顶点进队*/
     while(!QueueEmpty(&Q))
     {                                      /*队列非空*/
          DelQueue(&Q,&i);
          p=G.adjList[i].firstArc;          /*获取第 1 个邻接点*/
          while(p!=NULL)
          {    if(visited[p->adjVex]==0)    /*访问第 i 个顶点的未曾访问的邻接顶点*/
               {printf("%3c",G.adjList[p->adjVex].data);
                    visited[p->adjVex]=1;
                    EnQueue(&Q,p->adjVex);  /*第 k 个顶点进队*/
               }
               p=p->nextArc;
          }
     }
}
```

3.

```
int visited[MAXSIZE]; //指示顶点是否在当前路径上
int level＝1;//递归进行的层数
int exist_path_DFS(ALGraph G,int i,int j)//深度优先判断有向图 G 中顶点 i 到顶点 j
```

```
是否有路径,是则返回 1,否则返回 0
{    if(i==j) return 1; //i 就是 j
     else
     {    visited[i]=1;
          for(p=G.vertices[i].firstarc;p;p=p->nextarc,level--)
          {       level++;
                  k=p->adjvex;
                  if(!visited[k]&&exist_path(k,j)) return 1;//i 下游的顶点到 j 有路径
          }//for
     }//else
     if (level==1)    return 0;
}//exist_path_DFS
```

4.

```
int visited[MAXSIZE];
int exist_path_len(ALGraph G,int i,int j,int k)//判断邻接表方式存储的有向图 G 的顶点 i 到 j 是否存在长度为 k 的简单路径
{    if(i==j&&k==0) return 1; //找到了一条路径,且长度符合要求
     else if(k>0)
     {    visited[i]=1;
          for(p=G.vertices[i].firstarc;p;p=p->nextarc)
          {       l=p->adjvex;
                  if(!visited[l])
                     if(exist_path_len(G,l,j,k-1)) return 1; //剩余路径长度减一
          }//for
          visited[i]=0; //本题允许曾经被访问过的结点出现在另一条路径中
     }//
     else
          return 0; //没找到
}//exist_path_len
```

5. 解：//本题中的图 G 均为有向无权图。

```
Status Delete_Arc(MGraph &G,char v,char w)//在邻接矩阵表示的图 G 上删除边(v,w)
{   if((i=LocateVex(G,v))<0) return ERROR;
    if((j=LocateVex(G,w))<0) return ERROR;
    if(G.arcs[i][j].adj)
    {   G.arcs[i][j].adj=0;
        G.arcnum--;
    }
    return OK;
}//Delete_Arc
```

第 7 章　习题参考答案

一、选择题

1. B	2. A	3. A	4. C	5. C	6. A	7. C	8. B	9. A

二、填空题

1. 散列法　　关键字　　2. 2　5

3. 构造一个好的 HASH 函数　　确定解决冲突的方法　　4. 9　7

5. 顺序查找（线性查找）　28，6，12，20　　6. 7/6　8/6

7. 1, 3

三、简答题

1. (1)查找不成功时，需进行 n+1 次比较才能确定查找失败。因此平均查找长度为 n+1，这时有序表和无序表是一样的。

(2)查找成功时，平均查找长度为(n+1)/2,有序表和无序表也是一样的。因为顺序查找与表的初始序列状态无关。

2. n 的查找过程如下(其中方括号表示当前查找区间，圆括号表示当前比较的关键字)

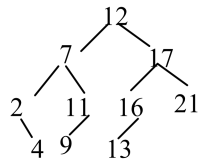
下标：　1 2 3 4 5 6 7 8 9 10 11 12 13
第一次比较：[a b c d e f (g) h i j k p q]
第二次比较：a b c d e f g [h i (j) k p q]
第三次比较：a b c d e f g h i j [k (p) q]
第四次比较：a b c d e f g h i j [(k)] p q]
经过四次比较，查找失败。

3. 解答：

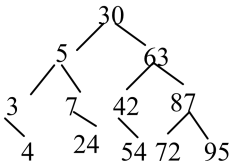
查找 b 的过程如下(其中方括号表示当前查找区间，圆括号表示当前比较的关键字)

下标：　1 2 3 4 5 6 7 8 9 10 11 12 13
第一次比较：[a b c d e f (g) h i j k p q]
第二次比较：[a b (c) d e f] g h i j k p q
第三次比较：[a (b)]c d e f g h i j k p q
经过三次比较，查找成功。

4.



5. (1)判定树如下（注：mid=⌊(1+12)/2⌋=6）：



(2) 查找元素 54，需依次与 30, 63, 42, 54 等元素比较；

(3) 查找元素 90，需依次与 30, 63,87, 95 等元素比较；

(4) 求 ASL 之前，需要统计每个元素的查找次数。判定树的前 3 层共查找 1＋2×2＋4×3=17 次；

但最后一层未满，不能用 8×4，只能用 5×4=20 次，

所以 ASL＝1/12（17＋20）＝37/12≈3.08

6. 不适合！虽然有序的单链表的结点是按从小到大（或从大到小）顺序排列，但因其存储结构为单链表，查找结点时只能从头指针开始逐步搜索，故不能进行折半查找。

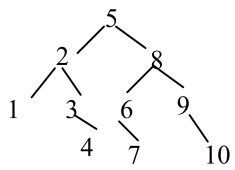
二分查找的速度在一般情况下是快些，但在特殊情况下未必快。例如所查数据位于首位时，则线性查找快；而二分查找则慢得多。

7. 解答：根据哈希函数和处理冲突的方法为二次探测再散列，构造哈希表如图所示：

0	1	2	3	4	5	6	7	8	9
14	1	9	23	84	27	55	20		

8. 查找某个元素的时间复杂度下限，如果理解为最短查找时间，则当关键字值与表头元素相同时，比较 1 次即可。要想降低时间复杂度，可以改用 Hash 查找法。此方法对表内每个元素的比较次数都是 O（1）。

9. 判定树应当描述每次查找的位置：



ASL＝1/10（1＋2×2＋3×4＋4×3）
＝1/10（1＋4＋12＋12）
＝29/10=2.9

10 解：由题意知，m=11(刚好为素数)

则(22*3)%11=6……0　　(41*3)%11=11……2

(53*3)%11=14……5　　(08*3)%11=2……2　(08*3+1)%11=3

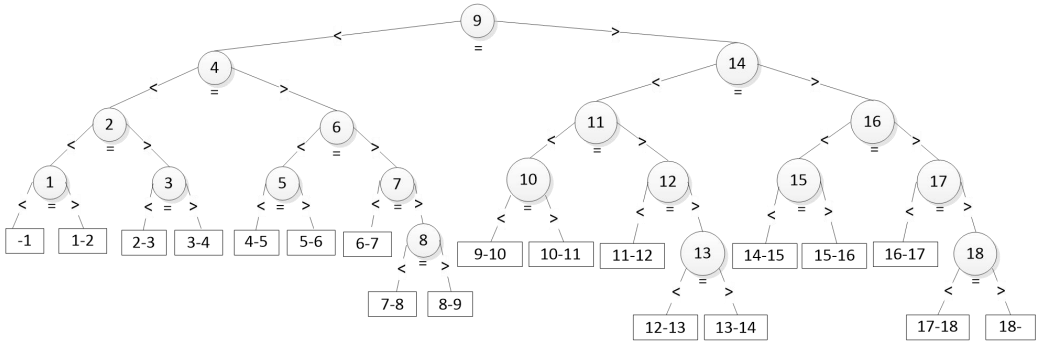
(46*3)%11=12……6　　(30*3)%11=8……2　(30*3+2)%11=4

(01*3)%11=0……3　(01*3+6)%11=7　　(31*3)%11=8……5　(31*3+3)%11=8

(66*3)%11=9……0　(66*3+1)%11=1

22	66	41	8	30	53	46	1	31		
0	1	2	3	4	5	6	7	8	9	10
1		3	4，7							

11.



等概率情况下，查找成功的平均查找长度为：

ASL=(1+2*2+3*4+4*8+5*3)/18=3.556

查找失败时，最多的关键字比较次树不超过判定树的深度，此处为 5。

12. （1）画表如下：

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
32	17	63	49					24	40	10				30	31	46	47

（2） 查找 63,首先要与 H(63)=63%16=15 号单元内容 31 比较，即 63 vs 31 ,no;

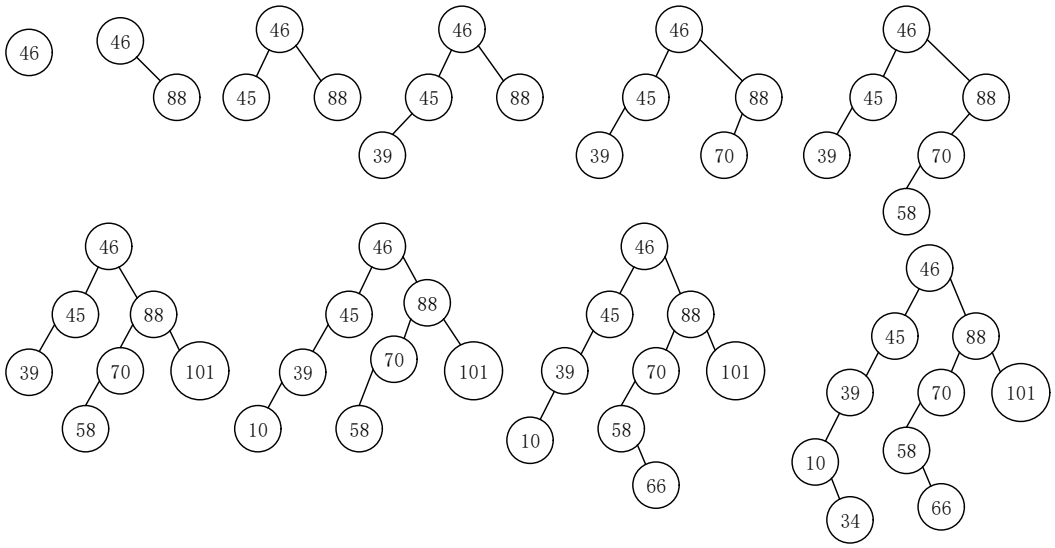
然后顺移，与 46,47,32,17,63 相比，一共比较了 6 次！

（3）查找 60,首先要与 H(60)=60%16=12 号单元内容比较，但因为 12 号单元为空（应当有空标记），所以应当只比较这一次即可。

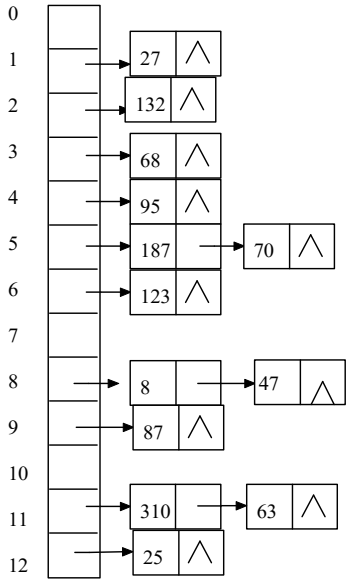
（4） 对于黑色数据元素 10，24，32，17，31，30，各比较 1 次；共 6 次；

对红色元素则各不相同，要统计移位的位数。“63”需要 6 次，“49”需要 3 次，“40”需要 2 次，“46”需要 3 次，“47”需要 3 次，所以 ASL=1/11×（6+6+2+3×3）=23/11=2.09

13. 等概率情况下，平均查找长度为：（5*2+4*2+3*3+2*2+1*1）/10=3.21



14.



等概率情况下，该哈希表的平均查找长度:(1*10+2*3)/13=16/13。

四、编程题

```
 1. void PrintWord(Hash Table ht)
{//按第一个字母的顺序输出哈希表 ht 中的标识符。哈希函数为表示符的第一个字母在字母表中的序号，处理冲突的方法是线性探测开放定址法。
    for(i=1; i<=26; i++)
    {    j=i;
         while(ht.elem[j].key)
         {    if(Hash(ht.elem[j].key==i)printf(ht.elem[j].key);
              j=(j+1)%m;
         }
    }
}//PrintWord
 2. int Search_Bin_Recursive(SSTable ST, int key, int low, int high)   //折半查找的递归算法
{   if(low>high) return -1;     //查找不到时返回-1
    mid=(low+high)/2;
    if(ST.elem[mid].key= =key) return mid;
    else if(ST.elem[mid].key>key)
       return Search_Bin_Recursive(ST, key, low, mid-1);
    else return Search_Bin_Recursive(ST, key, mid+1, high);
   }
}//Search_Bin_Recursive
3. int Search_Bin_Recursive(SSTable ST, int key, int low, int high)   //折半查找的递归算法
{   if(low>high) return 0;     //查找不到时返回 0
    mid=(low+high)/2;
    if(ST.elem[mid].key= =key) return mid;
    else if(ST.elem[mid].key>key)
```

```
        return Search_Bin_Recursive(ST, key, low, mid-1);
    else return Search_Bin_Recursive(ST, key, mid+1, high);
}//Search_Bin_Recursive
```

第 8 章　习题参考答案

一、选择题

一、选择题

1. B	2. D	3. A	4. C	5. C	6. C	7. B	8. D	9. C	10. D
11. A									

二、填空题

1. 比较、移动　　2. 插入、选择

3. $\log_2^n$　H Q C Y A P M S D R F X　　4. 堆排序、快速排序

5. 直接插入,选择排序　　6. 快速　归并

7. 归并排序　　快速排序　　8. 6

9. H C Q P A M S R D F X Y　　P A C S Q D F X R H M Y

三、简答题

1. 直接选择排序：(方括号为无序区)

```
初始态    [265 301 751 129 937 863 742 694 076 438]
第一趟：  076 [301 751 129 937 863 742 694 265 438]
第二趟：  076 129 [751 301 937 863 742 694 265 438]
第三趟：  076 129 265[ 301 937 863 742 694 751 438]
第四趟：  076 129 265 301 [937 863 742 694 751 438]
第五趟：  076 129 265 301 438 [863 742 694 751 937]
第六趟：  076 129 265 301 438 694 [742 863 751 937]
第七趟：  076 129 265 301 438 694 742 [863 751 937]
第八趟：  076 129 265 301 438 694 742 751 [863 937]
第九趟：  076 129 265 301 438 694 742 751 863 937
```

2. 冒泡排序(方括号为无序区)

```
初始态：[265 301 751 129 937 863 742 694 076 438]
第一趟：[265 301 129 751 863 742 694 076 438] 937
第二趟：[265 129 301 751 742 694 076 438] 863 937
第三趟：[129 265 301 742 694 076 438] 751 863 937
第四趟：[129 265 301 694 076 438] 742 751 863 937
第五趟：[129 265 301 076 438] 694 742 751 863 937
第六趟：[129 265 076 301] 438 694 742 751 863 937
第七趟：[129 076 265] 301 438 694 742 751 863 937
第八趟：[076 129] 265 301 438 694 742 751 863 937
第九趟：[076] 129 265 301 438 694 742 751 863 937
```

3.

划分次序	划分结果
初始态	46　79　56　38　40　80　95　24
第一次	[24　40　38] 46　[56　80　95　79]
第二次	24　[38　40] 46　[56　80　95　79]
第三次	24　38　40　46　[56　80　95　79]
第四次	24　38　40　46　56　[80　95　79]
第五次	24　38　40　46　56　79　[80　95]
第六次	24　38　40　46　56　79　80　95

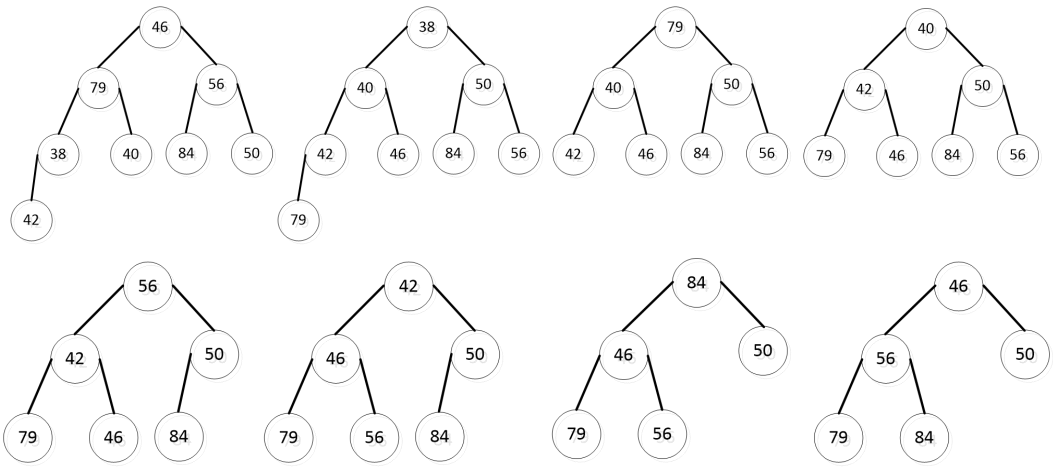
4. 归并排序（为了表示方便，采用自底向上的归并，方括号为有序区）

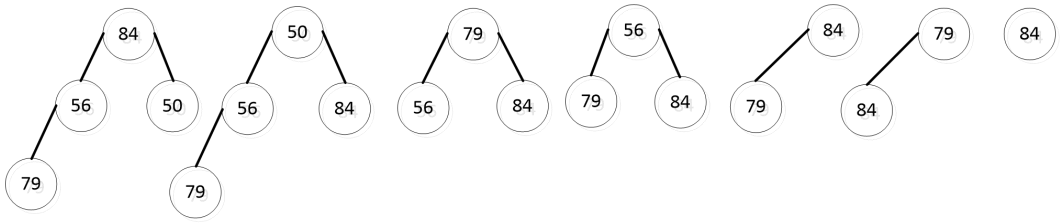
初始态：[265] [301] [751] [129] [937] [863] [742] [694] [076] [438]
第一趟：[265 301] [129 751] [863 937] [694 742] [076 438]
第二趟：[129 265 301 751] [694 742 863 937] [076 438]
第三趟：[129 265 301 694 742 751 863 937] [076 438]
第四趟：[076 129 265 301 438 694 742 751 863 937]

5. 直接插入排序:(方括号表示无序区)

初始态: 265[301 751 129 937 863 742 694 076 438]
第一趟：265 301[751 129 937 863 742 694 076 438]
第二趟：265 301 751[129 937 863 742 694 076 438]
第三趟：129 265 301 751[937 863 742 694 076 438]
第四趟：129 265 301 751 937[863 742 694 076 438]
第五趟：129 265 301 751 863 937[742 694 076 438]
第六趟：129 265 301 742 751 863 937[694 076 438]
第七趟：129 265 301 694 742 751 863 937[076 438]
第八趟：076 129 265 301 694 742 751 863 937[438]
第九趟：076 129 265 301 438 694 742 751 863 937

6.



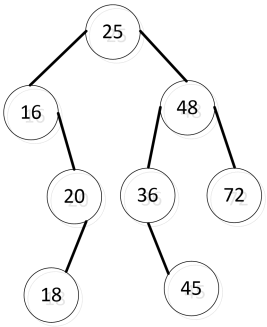


7. 二叉排序树或者是一棵空树,或者是具有如下性质的二叉树:

(1)若它的左子树不空,则左子树上所有结点的值均小于根结点的值;

(2)若它的右子树不空,则右子树上所有结点的值均大于根结点的值;

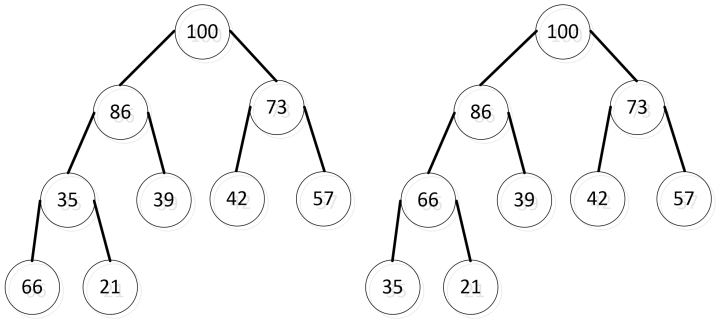
(3)左右子树又各是一棵二义排序树。



8. 解答:

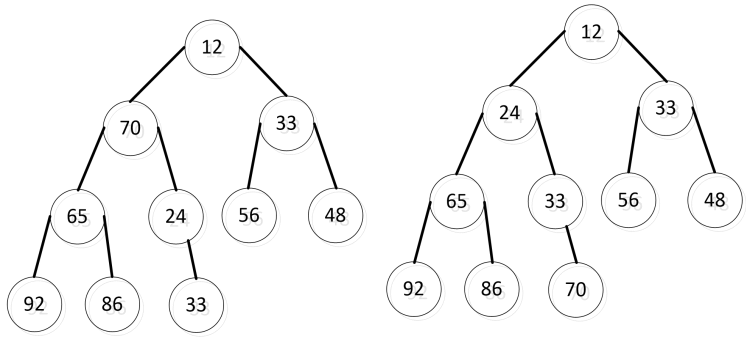
堆的性质是:任一非叶结点上的关键字均不大于(或不小于)其孩子结点上的关键字。据此我们可以通过画二叉树来进行判断和调整:

(1) 此序列不是堆,经调整后成为大根堆:



(100,86,73,66,39,42,57,35,21)

(2) 此序列不是堆,经调整后成为小根堆:

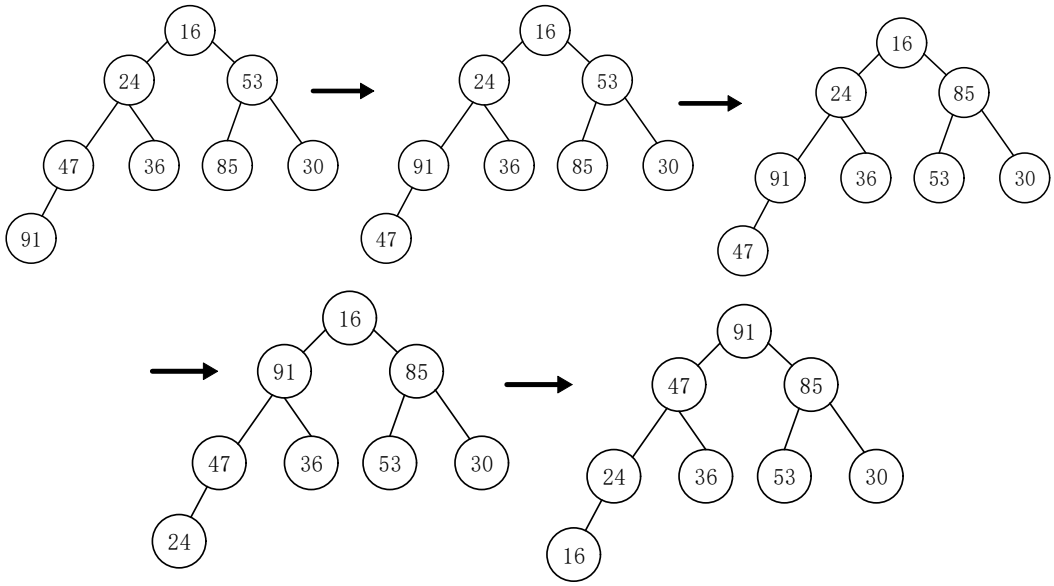


(12，24，33，65，33，56，48，92，86，70)

9. 希尔排序(增量为 5，3，1)

初始态: 265 301 751 129 937 863 742 694 076 438
第一趟：265 301 694 076 438 863 742 751 129 937
第二趟：076 301 129 265 438 694 742 751 863 937
第三趟：076 129 265 301 438 694 742 751 863 937

10. 序列{16，24，53，47，36，85，30，91}建堆的过程如下图所示：



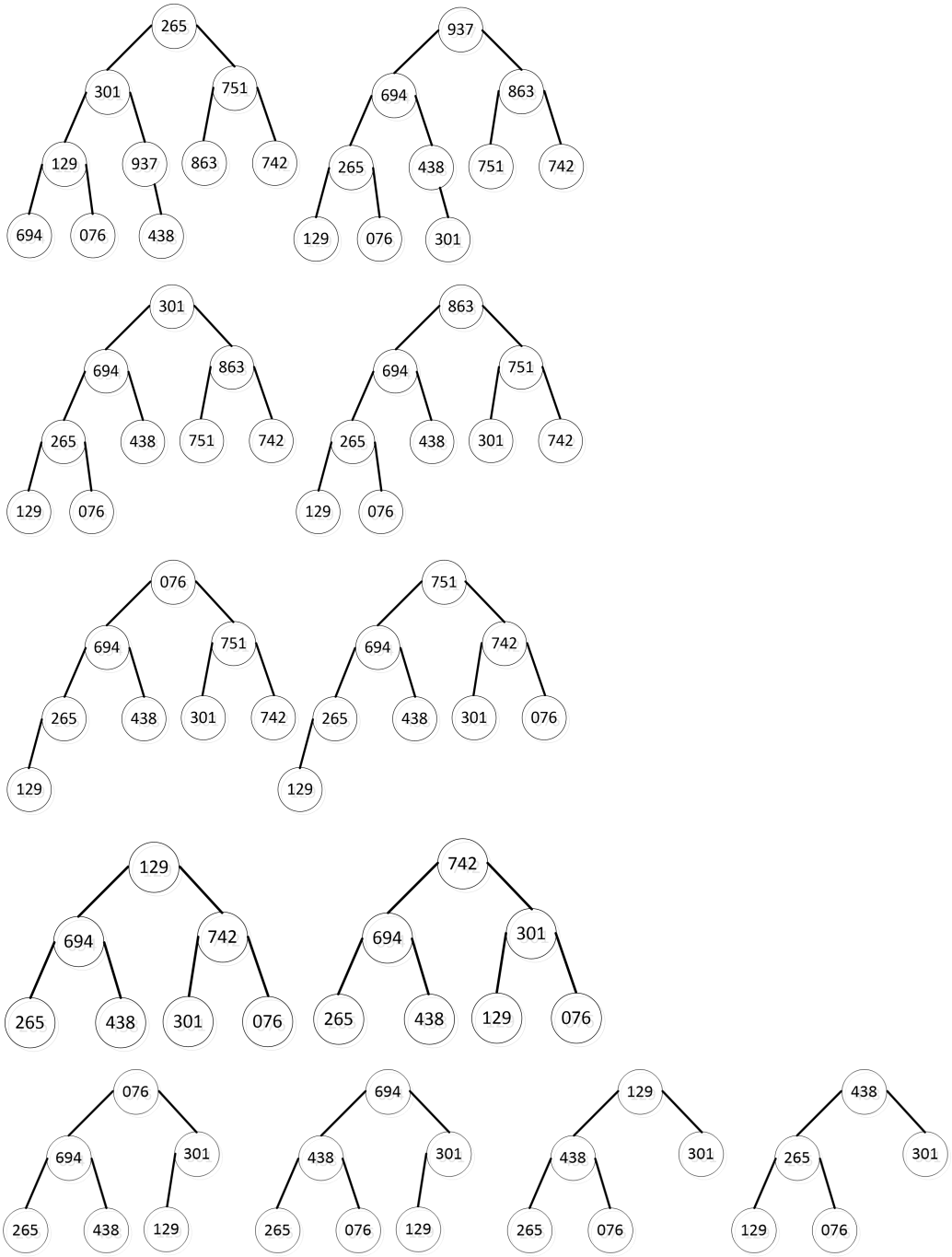
11. 快速排序：（方括号表示无序区，层表示对应的递归树的层数）

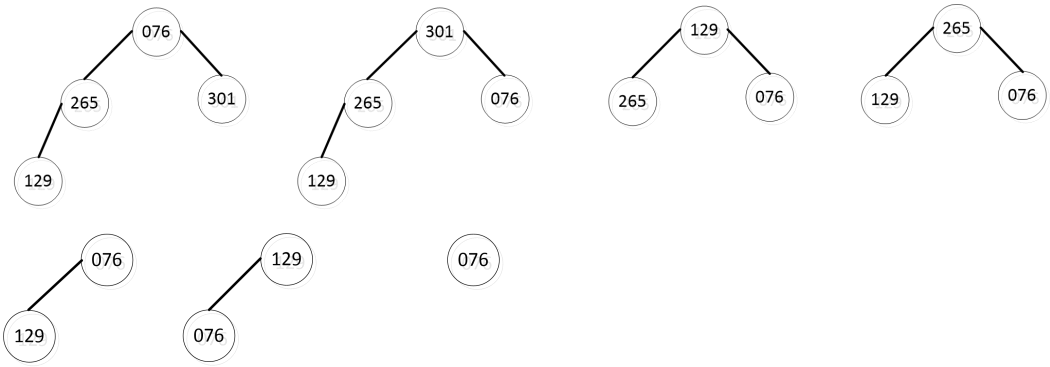
初始态：　[265 301 751 129 937 863 742 694 076 438]
第二层：　[076 129] 265 [751 937 863 742 694 301 438]
第三层：　076 [129] 265 [438 301 694 742] 751 [937 863]
第四层：　076 129 265 [301] 438 [694 742] 751 [863] 937
第五层：　076 129 265 301 438 694 [742] 751 863 937
第六层：　076 129 265 301 438 694 742 751 863 937

12. 堆排序：（通过画二叉树可以一步步得出排序结果）

初始态　　[265 301 751 129 937 863 742 694 076 438]
建立初始堆：　[937 694 863 265 438 751 742 129 075 301]
第一次排序重建堆：[863 694 751 765 438 301 742 129 075] 937

第二次排序重建堆：[751 694 742 265 438 301 075 129] 863 937
第三次排序重建堆：[742 694 301 265 438 129 075] 751 863 937
第四次排序重建堆：[694 438 301 265 075 129] 742 751 863 937
第五次排序重建堆：[438 265 301 129 075] 694 742 751 863 937
第六次排序重建堆：[301 265 075 129] 438 694 742 751 863 937
第七次排序重建堆：[265 129 075] 301 438 694 742 751 863 937
第八次排序重建堆：[129 075]265 301 438 694 742 751 863 937
第九次排序重建堆：075 129 265 301 438 694 742 751 863 937




076
265
301
129
301
265
076
129
129
265
076
265
129
076
076
129
129
076
076