第 1 章　绪论

一、基础知识题

1.1　简述下列概念

数据，数据元素，数据类型，数据结构，逻辑结构，存储结构，算法。

【解答】数据是信息的载体，是描述客观事物的数、字符，以及所有能输入到计算机中并被计算机程序识别和处理的符号的集合。

数据元素是数据的基本单位。在不同的条件下，数据元素又可称为元素、结点、顶点、记录等。

数据类型是对数据的取值范围、数据元素之间的结构以及允许施加操作的一种总体描述。每一种计算机程序设计语言都定义有自己的数据类型。

“数据结构”这一术语有两种含义，一是作为一门课程的名称；二是作为一个科学的概念。作为科学概念，目前尚无公认定义，一般认为，讨论数据结构要包括三个方面，一是数据的逻辑结构，二是数据的存储结构，三是对数据进行的操作（运算）。而数据类型是值的集合和操作的集合，可以看作是已实现了的数据结构，后者是前者的一种简化情况。

数据的逻辑结构反映数据元素之间的逻辑关系（即数据元素之间的关联方式或“邻接关系”），数据的存储结构是数据结构在计算机中的表示，包括数据元素的表示及其关系的表示。数据的运算是对数据定义的一组操作，运算是定义在逻辑结构上的，和存储结构无关，而运算的实现则依赖于存储结构。

数据结构在计算机中的表示称为物理结构，又称存储结构。是逻辑结构在存储器中的映像，包括数据元素的表示和关系的表示。逻辑结构与计算机无关。

算法是对特定问题求解步骤的一种描述，是指令的有限序列。其中每一条指令表示一个或多个操作。一个算法应该具有下列特性：有穷性、确定性、可行性、输入和输出。

1.2　数据的逻辑结构分哪几种，为什么说逻辑结构是数据组织的主要方面？

【解答】数据的逻辑结构分为线性结构和非线性结构。（也可以分为集合、线性结构、树形结构和图形即网状结构）。

逻辑结构是数据组织的某种“本质性”的东西：

（1）逻辑结构与数据元素本身的形式、内容无关。

（2）逻辑结构与数据元素的相对位置无关。

（3）逻辑结构与所含数据元素的个数无关。

1.3　试举一个数据结构的例子，叙述其逻辑结构、存储结构、运算三方面的内容。

【解答】学生成绩表，逻辑结构是线性结构，可以顺序存储（也可以链式存储），运算可以有插入、删除、查询，等等。

1.4　简述算法的五个特性，对算法设计的要求。

【解答】算法的五个特性是：有穷性、确定性、可行性、零至多个输入和一至多个输出。

对算法设计的要求：正确性，易读性，健壮性，和高的时空间效率（运算速

度快，存储空间小）。

1.5　设 n 是正整数，求下列程序段中带@记号的语句的执行次数。

(1)
```
i=1;k=0;
while(i<n)
  {k=k+50*i; i++;    @
  }
```

(2)
```
i=1;j=0;
while(i+j<=n)
  {if(i>j)j++;
@
   else i++;  }
@
```

(3)
```
x=y=0;
for(i=0;i<n;i++)    @
for(j=0;j<n;j++)    @
  {x++;             @
@
   for(k=0;k<n;k++) @
     y++;           @
@
   }
```

(4)
```
x=91;y=100;
while(y>0)
  if(x>100)
    {x=x-10; y--;
@
    }
  else x++;
@
```

【解答】(1)n-1

(2)i=⌈n/2⌉ ，j=⌊n/2⌋

(3)n+1, n(n+1), n^2,(n+1)n^2, n^3

(4)100, 1000

1.6　有实现同一功能的两个算法 A1 和 A2，其中 A1 的时间复杂度为 T1=O(2^n)，A2 的时间复杂度为 T2=O(n^2)，仅就时间复杂度而言，请具体分析这两个算法哪一个好。

【解答】对算法 A1 和 A2 的时间复杂度 T1 和 T2 取对数，得 n$\log^2$和 2$\log^n$。显然，当 n<4 时，算法 A1 好于 A2；当 n=4 时，两个算法时间复杂度相同；当 n>4 时，算法 A2 好于 A1。

1.7　选择题：算法分析的目的是（　　）

A、找出数据结构的合理性　　B、研究算法中的输入和输出的关系

C、分析算法的效率以求改进　　D、分析算法的易懂性和文档特点

【解答】C

二、算法设计题

1.8　已知输入 x，y，z 三个不相等的整数，设计一个“高效”算法，使得这三个数按从小到大输出。“高效”的含义是用最少的元素比较次数、元素移动次数和输出次数。

【算法 1.8】

```
void Best()
{//按序输出三个整数的优化算法
int a,b,c,t;
scanf(“%d%d%d”,&a,&b,&c);
```

```
if(a>b)
    {t=a;  a=b; b=t:}       //a 和 b 已正序
if(b>c)
   {t=c; c=b;               //c 已到位
     if(a>t) {b=a; a=t;}    //a 和 b 已正序
     else b=t;
    }
printf(“%d,%d,%d\n”,a,b,c);
 //最佳 2 次比较，无移动；最差 3 次比较，7 个赋值
}
```

1.9　在数组 A[n]中查找值为 k 的元素，若找到则输出其位置 i(1≤i≤n)，否则输出 0 作为标志。设计算法求解此问题，并分析在最坏情况下的时间复杂度。

【题目分析】从后向前查找，若找到与 k 值相同的元素则返回其位置，否则返回 0。

【算法 1.9】

```
int Search(ElemType A[n+1], ElemType k)
{i=n;
 while(i>=1)&&(A[i]!=k)) i--;
 if(i>=1) return i;
 else return 0;
}
```

当查找不成功时，总的比较次数为 n+1 次，所以最坏情况下时间复杂度为 O(n)。

在学过第 9 章 “查找”后，可优化以上算法:设“监视哨”。算法如下：

```
int Search(ElemType A[n+1], ElemType k)
{i=n;  A[0]=k;
 while(A[i]!=k) i--;
 return i;
 }
```

研究表明，当 n>=1000 时，算法效率提高 50%。

第 2 章　线性表

一、基础知识题

2.1　试述头指针、头结点、元素结点、首元结点的区别，说明头指针和头结点的作

【解答】指向链表第一个结点（或为头结点或为首元结点）的指针称为**头指针**。“头指针”具有标识一个链表的作用，所以经常用头指针代表链表的名字，如链表 L 既是指链表的名字是 L，也是指链表的第一个结点的地址存储在指针变量 L 中，头指针为“NULL”则表示指针变量 L 没指向任何链表。

有时，我们在整个线性链表的第一个元素结点之前加入一个结点，称为**头结点**，它的数据域可以不存储任何信息（当然，作为“副产品”，头结点的数据域也可能做监视哨或存放线性表的长度等附加信息），指针域中存放的是第一个数据结点的地址，空表时为空。 “头结点”的加入，使插入和删除等操作方便、统一。

元素结点即是数据结点，至少包括元素自身信息和其后继元素的地址两个域。

首元结点是指链表中第一个数据元素的结点；为了操作方便，通常在链表的首元结点之前附设一个结点，称为**头结点**。

2.2 分析顺序存储结构和链式存储结构的优缺点，说明何时应该利用何种结构。

【解答】①从空间上来看，当线性表的长度变化较大，难以估计其规模时，选用动态的链表作为存储结构比较合适。由于链表除了需要设置数据域外，还要额外设置指针域，因此当线性表长度变化不大，易于事先确定规模时，为了节约存储空间，宜采用顺序存储结构。

②从时间上看，顺序表具有按元素序号随机访问的特点，在顺序表中按序号访问数据元素的时间复杂度为 O(1)；而链表中按序号访问的时间复杂度为 O(n)。所以如果经常按序号访问数据元素，使用顺序表优于链表。

在顺序表中做插入删除操作时，平均移动大约表中一半的元素，因此 n 较大时顺序表的插入和删除效率低。在链表中作插入、删除，虽然也要找插入位置，但操作主要是比较操作。从这个角度考虑显然链表优于顺序表。

总之，两种存储结构各有长短，选择那一种存储结构，由实际问题中的主要因素决定。

2.3 分析在顺序存储结构下插入和删除结点时平均需要移动多少个结点。

【解答】平均移动表中大约一半的结点。插入操作平均移动 $n/2$ 个结点，删除操作平均移动 $(n-1)/2$ 个结点。具体移动的次数取决于表长和插入、删除的结点的位置。

2.4 为什么在单循环链表中常使用尾指针，若只设头指针，插入元素的时间复杂度如何？

【解答】单循环链表中无论设置尾指针还是头指针都可以遍历到表中任一个结点。设置尾指针时，若在表尾进行插入元素或删除第一元素，操作可在 O(1)时间内完成；若只设置头指针，表尾进行插入或删除操作，需要遍历整个链表，时间复杂度为 O(n)。

2.5 在单链表、双链表、单循环链表中，若知道指针 p 指向某结点，能否删除该结点，时间复杂度如何？

【解答:】以上三种链表中，若知道指针 p 指向某结点，都能删除该结点。

双链表删除 p 所指向的结点的时间复杂度为 O(1)，而单链表和单循环链表上删除 p 所指向的结点的时间复杂度均为 O(n)。

2.6 下面算法的功能是什么？

```
LinkedList Unknown(LinkedList la)
   {LNode *q, *p;
     if(la && la->next)
         {q=la; la=la->next; p=la;
          while(p->next) p=p->next;
          p->next=q; q->next=null;
          }
      return la;
    }
```

【解答】将首元结点删除并插入到表尾（设链表长度大于 1）。

2.7 选择题：在循环双链表的*p 结点之后插入*s 结点的操作是（ ）

A 、 p->next=s; s->prior=p; p->next->prior=s; s->next=p->next;

B 、 p->next=s; p->next->prior=s; s->prior=p; s->next=p->next;

C 、 s->prior=p; s->next=p->next; p->next:=s; p->next->prior=s;

D、s->prior=p; s>next=p>next; p>next->prior =s; p->next=s;

【解答】D

2.8 选择题:若某线性表最常用的操作是存取任一指定序号的元素和在最后进行

插入和删除运算，则利用（　　）存储方式最节省时间。

A．顺序表　B．双链表　C．带头结点的双循环链表　D．单循环链表

【解答】A

二、算法设计题

2.9 设 *ha* 和 *hb* 分别是两个带头结点的非递减有序单链表的头指针， 试设计算法， 将这两个有序链表合并成一个非递增有序的单链表。要求使用原链表空间，表中无重复数据。

【题目分析】因为两链表已按元素值非递减次序排列，将其合并时，均从第一个结点起进行比较，将小的链入链表中，同时后移链表工作指针，若遇值相同的元素，则删除之。该问题要求结果链表按元素值非递增次序排列，故在合并的同时，将链表结点逆置。

【算法 2.9】

```
LinkedList Union(LinkedList ha,hb)
  //ha,hb 分别是带头结点的两个单链表的头指针，链表中的元素值按非递减有
    序
  //本算法将两链表合并成一个按元素值非递增有序的单链表,并删除重复元素
     { pa=ha->next;      //pa 是链表 ha 的工作指针
       pb=hb->next;      //pb 是链表 hb 的工作指针
       ha->next=null;   //ha 作结果链表的头指针,先将结果链表初始化为空
       while(pa!=null && pb!=null) //当两链表均不为空时作
         {while(pa->next && pa->data==pa->next->data)
            {u=pa->next; pa->next=u->next; free(u)}//删除 pa 链表中的
重复元素
          while(pb->next && pb->data==pb->next->data)
            {u=pb->next; pb->next=u->next; free(u)}//删除 pb 链表中的
重复元素
          if(pa->data<pb->data)
            {r=pa->next;             //将 pa 的后继结点暂存于 r
             pa->next=ha->next;      //将 pa 结点链于结果表中，同时逆置
             ha->next=pa;
             pa=r;                   //恢复 pa 为当前待比较结点
            }
          else if(pb->data<pa->data)
                 {r=pb->next;             // 将 pb 的后继结点暂存于 r
                  pb->next=ha->next;      //将 pb 结点链于结果表中,同时
逆置
                  ha->next=pb;
                  pb=r;                   //恢复 pb 为当前待比较结点
                 }
               else{u=pb;pb=pb->next;free(u)}//删除链表 pb 和 pa 中的
重复元素
       }// while(pa!=null && pb!=null)
```

```
    if(pa) pb=pa;       //避免再对 pa 写下面的 while 语句
    while(pb!=null)     //将尚未到尾的表逆置到结果表中
      {r=pb->next; pb->next=ha->next; ha->next=pb; pb=r; }
   return ha
  }//算法 Union 结束
```

2.10　设 la 是一个双向循环链表，其表中元素递增有序。试写一算法插入元素 x，使表中元素依然递增有序。

【问题分析】双向链表的插入与单链表类似，不同之处是需要修改双向指针。

【算法 2.10】

```
 DLinkedList DInsert(DLinkedList la, ElemType x)
   //在递增有序的双向循环链表 la 中插入元素 x，使表中元素依然递增有序
  {p=la->next;       //p 指向第一元素
   la->data=MaxElemType;//MaxElemType 是和 x 同类型的机器最大值，用做监视哨
   while(p->data<x) //寻找插入位置
      p=p->next  ;
   s=(DLNode *)malloc(sizeof(DLNode));//申请结点空间
   s->data=x;
   s->prior=p->prior; s->next=p;       //将插入结点链入链表
   p->prior->next=s; p->prior=s;
  }
```

2.11　设 p 指向头指针为 la 的单链表中某结点，试编写算法，删除结点*p 的直接前驱结点。

【题目分析】设*p是单链表中某结点，删除结点*p的直接前驱结点，要找到*p的前驱结点的前驱*pre。进行如下操作：u=pre->next; pre->next=u->next;free(u);

【算法2.11】

```
LinkedList LinkedListDel(LinkedList la, LNode *p)
 {//删除单链表la上的结点*p的直接前驱结点,假定*p存在
    pre=la;
    if(pre-next==p)
      printf(“*p是链表第一结点，无前驱\n”)  ; exit(0)  ; }
    while(pre->next->next !=p)
      pre=pre->next;
    u=pre->next; pre->next=u->next; free(u);
    return(la);
 }
```

2.12　设计一算法，将一个用循环链表表示的稀疏多项式分解成两个多项式，使这两个多项式各自仅有奇次幂或偶次幂项，并要求利用原链表中的结点空间来构造这两个链表。

【题目分析 】设循环链表表示的多项式的结点结构为：

```
typedef struct node
    {int power;          //幂
     float coef;         //系数
     ElemType other;     //其他信息
     struct node *next; //指向后继的指针
    }PNode,*PolyLinkedList;
```

则可以从第一个结点开始，根据结点的幂是奇数或偶数而将其插入到奇次幂或偶次幂项的链表中。假定用原链表保存偶次幂，要为奇次幂的链表生成一个表头，为了保持链表中结点的原来顺序，用一个指针指向奇次幂链表的表尾。注意链表分解时不能“断链”。

【算法 2.12】

```
void PolyDis(PolyLinkedList poly)
   //将 poly 表示的多项式链表分解为各含奇次幂或偶次幂项的两个循环链表
   {PolyLinkedList poly2=(PolyLinkedList)malloc(sizeof(PNode));
                        //poly2 表示只含奇次幂的多项式
    r2=poly2;           //r2 是只含奇次幂的多项式链表的尾指针
    r1=poly;            //r1 是只含偶次幂的多项式链表当前结点的前驱结点的指针
    p=poly->next;       //链表带头结点，p 指向第一个元素
    while(p!=poly)
      if(p->power % 2)//处理奇次幂
         {r=p->next;  //暂存后继
          r2->next=p; //结点链入奇次幂链表
          r2=p;       //尾指针后移
          p=r;        //恢复当前待处理结点
          }
      else //处理偶次幂
        {r1->next=p; r1=p; p=p->next;}
        }
      r->next=poly2; r1->next=poly;  //构成循环链表
    }//PolyDis
```

2.13　以带头结点的双向链表表示的线性表 $L=(a_1, a_2, \cdots, a_n)$，试写一时间复杂度为 O(n)的算法，将 L 改造为 $L=(a_1, a_3, \cdots, a_n, \cdots, a_4, a_2)$。

【题目分析】分析结果链表，易见链表中位置是奇数的结点保持原顺序，而位置是偶数的结点移到奇数结点之后，且以与原来相反的顺序存放。因此，可从链表第一个结点开始处理，位置是奇数的结点保留不动，位置是偶数的结点插入到链表尾部，并用一指针指向链表尾，以便对偶数结点“尾插入”。

【算法 2.13】

```
DLinkedList DInvert(DLinkedList L)
    //将双向循环链表 L 位置是偶数的结点逆置插入到链表尾部
    {p=L->next;       //p 指向第一元素
     Q=p->prior;      //Q 指向最后一个元素
```

```
    pre=L  ;          //pre 指向链表中位置为奇数的结点的前驱
    r=L  ;            //r 指向链表中偶数结点的尾结点
    i=0  ;            //i 记录结点序号
    while(p  != Q)    //寻找插入位置
       {i++  ;
        if(i%2)      //处理序号为奇数的结点
           {p->prior=pre  ;pre->next=p  ;pre=p; p=p->next;}
        else         //处理序号为偶数的结点
           {u=p  ;   //记住当前结点
            p=p->next  ;//p 指向下个待处理结点
            u->prior=r->prior;  //以下 4 个语句将结点插入链表尾
            u->next=r;
            r->prior->next=u;
            r->prior=u;
            r=u;          //指向新的表尾
            } //else
        } //while
    } //结束算法
```

2.14　设单向链表的头指针为 head，试设计算法，将链表按递增的顺序就地排序。

【题目分析】本题中的“就地排序”，可理解为不另辟空间，这里利用直接插入原则把链表整理成递增有序链表。

【算法 2.14】

```
LinkedList LinkListInsertSort(LinkedList head)
 //利用直接插入原则将带头结点的单链表 head 整理成递增的有序链表
 {if(head->next!=null)      //链表不为空表
   {p=head->next->next;     //p 指向第一结点的后继
    head->next->next=null;//第一元素有序,然后从第二元素起依次插入
    while(p!=null)
     {r=p->next;          //暂存 p 的后继
      q=head;
      while(q->next && q->next->data<p->data)
          q=q->next;//查找插入位置
      p->next=q->next;     //将 p 结点链入链表
      q->next=p;
      p=r;
      }
    }
  }
```

2.15　已知递增有序的三个单链表分别代表集合 A,B 和 C,设计算法实现 A=A∪（B∩C），并使结果链表仍保持递增。要求算法的时间复杂度为 O(|A|+|B|+|C|)。其中，|A|为集合 A 的元素个数。

【题目分析】本题首先求 B 和 C 的交集，即求 B 和 C 中的共有元素，再与 A 求并

集，同时删除重复元素，以保持结果 A 递增。

【算法 2.15】

```
LinkedList union(LinkedList A,B,C)
//A、B 和 C 均是带头结点的递增有序的单链表,本算法实现 A=A∪(B∩C)
//结果表 A 保持递增有序
{pa=A->next;pb=B->next;pc=C->next;//设置三个工作指针
 pre=A;  //pre 指向结果链表中当前待合并结点的前驱
 A->data=MaxElemType;//同类型元素最大值，起监视哨作用
  while(pa || pb && pc)
    {while(pb && pc)
       if(pb->data<pc->data) pb=pb->next;
       else if(pb->data>pc->data) pc=pc->next;
            else break;  //B 表和 C 表有公共元素
     if(pb && pc)
       {while(pa && pa->data<pb->data)//先将 A 中小于 B，C 公共元素
部分链入
          {pre->next=pa;pre=pa;pa=pa->next;}
        if(pre->data!=pb->data)
            {pre->next=pb;pre=pb;pb=pb->next;pc=pc->next;}
        else{pb=pb->next;pc=pc->next;}//A 中已有 B，C 公共元素
        }
     }// while(pa||pb&&pc)
    if(pa) pre->next=pa; //当 B，C 无公共元素，将 A 中剩余链入
    else pre->next=null; //A 已到尾，B，C 也无公共元素，
  }//算法 Union 结束
```

2.16　顺序表 la 与 lb 非递减有序，顺序表空间足够大。试设计一种高效算法，将 lb 中元素合到 la 中，使新的 la 的元素仍保持非递减有序。高效指最大限度地避免移动元素。

【题目分析】顺序存储结构的线性表的插入，其时间复杂度为 O(n)，平均移动近一半的元素。线性表 la 和 lb 合并时，若从第一个元素开始比较，一定会造成元素后移，这不符合本题“高效算法”的要求。应从线性表的最后一个元素开始比较，大者放到最终位置上。设两线性表的长度各为 m 和 n ，则结果表的最后一个元素应在 m+n 位置上。这样从后向前，直到第一个元素为止。

【算法 2.16】

```
SeqList Union(SeqList la, SeqList lb)
//算法将顺序存储的非递减有序表 la 和 lb 中的 lb 合并到 la 中,la 仍非递
减有序
   { m=la.last;n=lb.last;//m，n 分别为线性表 la 和 lb 的长度
     k=m+n-1;      //k 为结果线性表的工作指针（下标）
     i=m-1;j=n-1; //i，j 分别为线性表 la 和 lb 的工作指针（下标）
     while(i>=0 && j>=0)
       if(la.data[i]>=lb.data[j]) la.data[k--]=la.data[i--];
```

```
        else  la.data[k--]=lb.data[j--];
      while(j>=0) la.data[k--]=lb.data[j--];
      la.last=m+n;
   return la;
  }
```

【算法讨论】算法中数据移动是主要操作。在最佳情况下（lb 的最小元素大于 la 的最大元素），仅将 lb 的 n 个元素移（拷贝）到 la 中，时间复杂度为 O（n），最差情况，la 的所有元素都要移动，时间复杂度为 O（m+n）。因数据合并到 la 中，所以在退出第一个 **while** 循环后，只需要一个 **while** 循环，处理 lb 中剩余元素。第二个循环只有在 lb 有剩余元素时才执行，而在 la 有剩余元素时不执行。本算法 “最大限度的避免移动元素”，是“一种高效算法”。

2.17　已知非空线性链表由 head 指出，试写一算法，将链表中数据域值最小的那个结点移到链表的最前面。要求：不得额外申请新的链结点。

【题目分析】 本题首先要查找最小值结点。将其移到链表最前面，实质上是将该结点从链表上摘下（不是删除并回收空间），再插入到链表的最前面。

【算法 2.17】

```
LinkedList  Delinsert(LinkedList  head)
  //本算法将非空线性链表 head 中数据域值最小的那个结点移到链表的最前面
 {p=head->next; //p 是链表的工作指针
  pre=head;      //pre 指向链表中数据域最小值结点的前驱
  q=p;           //q 指向数据域最小值结点，初始假定是第一结点
  while(p->next)
    {if (p->next->data<q->data) {pre=p; q=p->next; } //找到新的最小值结点
     p=p->next;
    }
  if(q!=head->next)    //若最小值是第一元素结点，则不需再操作
     {pre->next=q->next;    //将最小值结点从链表上摘下
      q->next=head->next;  //将 q 结点插到链表最前面
      head->next=q;
      }
 }//Delinsert
```

2.18　设 la 是带头结点的非循环双向链表的指针，其结点中除有 prior，data 和 next 外，还有一访问频度域 freq，其值在链表初始使用时为 0。当在链表中进行 ListLocate(la，x)运算时，若查找失败，则在表尾插入值为 x 的结点；若查找成功，值为 x 的结点的 freq 值增 1，并要求链表按 freq 域值非增（递减）的顺序排列，且最近访问的结点排在频度相同的结点的后面，使频繁访问的结点总是靠近表头。试编写符合上述要求的 ListLocate(la，x)运算的算法，返回找到结点的指针。

【题目分析】首先在双向链表中查找数据值为 x 的结点，查到后，将结点从链表上摘下，然后再顺结点的前驱链查找该结点的位置。

【算法 2.18】

```
DLinkList  ListLocate(DLinkedList L, ElemType x)
 // L 是带头结点的按访问频度递减的双向链表,本算法先查找数据 x
 //查找成功时结点的访问频度域增 1，最后将该结点按频度递减插入链表中
{DLinkList p=L->next,q=L; //p 为 L 表的工作指针，q 为 p 的前驱，用于查找插入位置
 while(p && p->data!=x) {q=p;p=p->next;}// 查找值为 x 的结点
 if(!p) {printf(“不存在所查结点,现插入之\n”);
         s=(DNode *)malloc(sizeof(DNode));
         s->data=x; s->freq=0;     //插入值为 x 的结点
         s->next=p; s->prior=q;
         q->next=s; p->prior=s; p=s; //返回 p 结点;
        }
 else { p->freq++;                  // 令元素值为 x 的结点的 freq 域加 1
        p->next->prior=p->prior;   // 将 p 结点从链表上摘下
        p->prior->next=p->next;
        q=p->prior;                // 以下查找 p 结点的插入位置
        while(q !=L && q->freq<p->freq) q=q->prior;
        p->next=q->next; q->next->prior=p;// 将 p 结点插入
        p->prior=q; q->next=p;
       }
 return(p);    //返回值为 x 的结点的指针
}  // 算法结束
```

2.19　三个带头结点的线性链表 la、lb 和 lc 中的结点均依元素值自小至大非递减排列（可能存在两个以上值相同的结点），编写算法对 la 表进行如下操作：使操作后的 la 中仅留下三个表中均包含的数据元素的结点，且没有值相同的结点，并释放所有无用结点。限定算法的时间复杂度为 O（m+n+p），其中 m、n 和 p 分别为三个表的长度。

【题目分析】留下三个链表中公共数据,首先查找两表 la 和 lb 中公共数据，再去 lc 中找有无该数据。要消除重复元素，应记住前驱，要求时间复杂度 O（m+n+p），在查找每个链表时，指针不能回溯。

【算法 2.19】

```
LinkedList lcommon（LinkedList la，lb，lc）
 //本算法使 la 表留下 la、lb 和 lc 三个非递减有序表共同结点，无重复元素
 {pa=la->next; pb=lb->next; pc=lc->next;
     //pa,pb 和 pc 分别是 la,lb 和 lc 三个表的工作指针
 pre=la;
 la->data=MaxElemType //pre 是 la 表当前结点的前驱结点的指针,头结点作监视哨
 while（pa && pb && pc）     //当 la,lb 和 lc 表均不空时，查找三表共同元素
     {while(pa&&pa->data==pre->data)
```

```
        {u=pa; pa=pa->next; free(u);}//删 la 中相同元素
      while（pb && pc）
        if（pb->data<pc->data）pb=pb->next; //结点元素值小时，后移指针
        else if（pb->data>pc->data）pc=pc->next;
        else break ;//处理 lb 和 lc 表元素值相等的结点
      if(pb && pc)
        {while（pa && pa->data<pc->data）{u=pa; pa=pa->next; free（u); }
         if(pa && pa->data>pc->data){pb=pb->next; pc=pc->next;}
         else if(pa && pa->data==pc->data)//pc，pa 和 pb 对应结点元素值相等
                {pre->next=pa; pre=pa; pa=pa->next; //将新结点链入 la 表
                 pb=pb->next; pc=pc->next;          //链表的工作指针后移
                }//pc，pa 和 pb 对应结点元素值相等
      } }//while（pa && pb && pc）
  pre->next=null;      //置新 la 表表尾
  while（pa!=null）    //删除原 la 表剩余元素。
    {u=pa; pa=pa->next; free（u); }
  }//算法结束
```

【算法讨论】 算法中 la 表、lb 表和 lc 表均从头到尾（严格说 lb、lc 中最多一个到尾）遍历一遍，算法时间复杂度符合 O（m+n+p）。算法主要由 **while**（pa && pb && pc）控制。三表有一个到尾则结束循环。要注意头结点的监视哨的作用，否则第一个结点要特殊处理。算法最后要给新 la 表置结尾标记，同时若原 la 表没到尾，还应释放剩余结点所占的存储空间。本算法并未释放 lb 和 lc 中的结点。

第 3 章 栈和队列

一、基础知识题

3.1有五个数依次进栈：1，2，3，4，5。在各种出栈的序列中，以 3，4 先出的序列有哪几个。（ 3 在 4 之前出栈）。

【解答】34215 ，34251， 34521

铁路进行列车调度时， 常把站台设计成栈式结构，若进站的六辆列车顺序为：1，2，3，4，5，6， 那么是否能够得到 435612， 325641， 154623 和 135426 的出站序列， 如果不能，说明为什么不能； 如果能， 说明如何得到(即写出"进栈"或"出栈"的序列)。

【解答】输入序列为 123456，不能得出 435612 和 154623。不能得到 435612 的理由是，输出序列最后两元素是 12，因为前面 4 个元素（4356）得到后，栈中元素剩 12，且 2 在栈顶，不可能让栈底元素 1 在栈顶元素 2 之前出栈。不能得到 154623 的理由类似，当栈中元素只剩 23，且 3 在栈顶，2 不可能先于 3 出栈。

得到 325641 的过程如下：1 2 3 顺序入栈，32 出栈，得到部分输出序列 32；然后 45 入栈，5 出栈，部分输出序列变为 325；接着 6 入栈并退栈，部分输出序列变为 3256；最后 41 退栈，得最终结果 325641。

得到 135426 的过程如下：1 入栈并出栈，得到部分输出序列 1；然后 2 和 3 入栈，3 出栈，部分输出序列变为 13；接着 4 和 5 入栈，5，4 和 2 依次出栈，部分输出序列变为 13542；最后 6 入栈并退栈，得最终结果 135426。

3.2若用一个大小为 6 的数组来实现循环队列，且当前 rear 和 front 的值分别为 0 和 3，当从队列中删除一个元素，再加入两个元素后，rear 和 front 的值分别为多少？

【解答】2 和 4

3.3设栈 S 和队列 Q 的初始状态为空，元素 e1，e2，e3，e4，e5 和 e6 依次通过栈 S，一个元素出栈后即进队列 Q，若 6 个元素出队的序列是 e3，e5，e4，

e6，e2，e1，则栈 S 的容量至少应该是多少？

【解答】 4

3.4循环队列的优点是什么，如何判断“空”和“满”。

【解答】循环队列解决了常规用 0--m-1 的数组表示队列时出现的“假溢出”（即队列未满但不能入队）。在循环队列中，我们仍用队头指针等于队尾指针表示队空，而用牺牲一个单元的办法表示队满：即当队尾指针加 1（求模）等于队头指针时，表示队列满。也有通过设标记以及用一个队头或队尾指针加上队列中元素个数来区分队列的“空”和“满”的。

3.5设长度为 n 的链队列用单循环链表表示，若只设头指针，则入队和出队的时间如何？若只设尾指针呢？

【解答】若只设头指针，则入队的时间为 O(n)，出队的时间为 O(1)。若只设尾指针，则入队和出队的时间均为 O(1)。

3.6指出下面程序段的功能是什么？

(1)
```
void demo1(SeqStack S)
  {int i, arr[64], n=0;
   while(!StackEmpty(S)) arr[n++]=Pop(S);
   for(i=0;i<n;i++) Push(S, arr[i]);
  }
```
【解答】程序段的功能是实现了栈中元素的逆置。

(2)
```
void demo2(SeqStack S, int m)//设栈中元素类型为 int 型
  {int x;SeqStack T;
   StackInit(T);
   while(!StackEmpty(S))
      if((x=Pop(S))!=m) Push(T, x);
   while(!(StackEmpty(T)) {x=Pop(T); Push(S, x);}
  }
```
【解答】程序段的功能是删除了栈中值为 m 的元素。

(3)
```
void demo3(SeQueue Q, int m)//设队列中元素类型为 int 型
  {int x;SeqStack S;
   StackInit(S);
   while(!QueueEmpty(Q)){x=QueueOut(Q); Push(S, x);}
   while(!StackEmpty(S)){x=Pop(s); QueueIn(Q, x);}
  }
```
【解答】程序段的功能是实现了队列中元素的逆置。

3.7试将下列递推过程改写为递归过程。

```
void ditui(int n)
  {i=n;
   while(i>0) printf(i--);
```

```
}
```

【解答】

```
void digui(int n)
  {if(n>0){printf(n);
           digui(n-1);
          }
  }
```

3.8 写出下列中缀表达式的后缀表达式：

(1)A*B*C　(2)(A+B)*C-D　(3)A*B+C/(D-E)

(4)(A+B)*D+E/(F+A*D)+C

解答】

(1)ABC**

(2)AB+C*D-

(3)AB*CDE-/+

(4)AB+D*EFAD*+/+C+

3.9选择题：循环队列存储在数组 A[0..m]中，则入队时的操作为(　)。

A. rear=rear+1　　B. rear=(rear+1) % (m-1)

C. rear=(rear+1) % m　　D. rear=(rear+1) % (m+1)

【解答】D

3.11 选择题：4 个园盘的 Hahoi 塔，总的移动次数为(　)。

A.7　　B. 8　　C.15　　D.16

【解答】C

3.12 选择题：允许对队列进行的操作有(　)。

A. 对队列中的元素排序　　B. 取出最近进队的元素

C. 在队头元素之前插入元素　　D. 删除队头元素

【解答】D

二、算法设计题

3.13 利用栈的基本操作，编写求栈中元素个数的算法。

【题目分析】 将栈值元素出栈，出栈时计数，直至栈空。

【算法 3.13】

```
int StackLength(Stack S)
   {//求栈中元素个数
    int n=0;
    while(!StackEmpty(S)
       {n++; Pop(S);
       }
    return n;
    }
```

【算法讨论】：若要求统计完元素个数后，不能破坏原来栈，则在计数时，将原

栈导入另一临时栈，计数完毕，再将临时栈倒入原栈中。

```
int StackLength(Stack S)
 {//求栈中元素个数
  int n=0;
  Stack T;
  StackInit(T); //初始化临时栈 T
  while(!StackEmpty(S)
     {n++; Push(T,Pop(S));
     }
  while(!StackEmpty(T)
     {Push(S,Pop(T));
     }
  return n;
  }
```

3.14　双向栈 S 是在一个数组空间 V[m]内实现的两个栈，栈底分别处于数组空间的两端。试为此双向栈设计栈初始化 Init(S)、入栈 Push(S，i，x)、出栈 Pop(S，i)算法，其中 i 为 0 或 1，用以指示栈号。

[题目分析]两栈共享向量空间，将两栈栈底设在向量两端，初始时，s1 栈顶指针为-1，s2 栈顶为 m。两栈顶指针相邻时为栈满。两栈顶相向、迎面增长，栈顶指针指向栈顶元素。

【算法 3.14】

```
#define ElemType int      //假设元素类型为整型
typedef struct
{ElemType V[m];     //栈空间
 int top[2];        //top 为两个栈顶指针的数组
}stk;
stk S;              //S 是如上定义的结构类型变量，为全局变量
```

(1)栈初始化

```
int Init()
  {S.top[0]=-1;
   S.top[1]=m;
   return 1;  //初始化成功
  }
```

(2)入栈操作：

```
int push(stk S ,int i,int x)
//i 为栈号，i=0 表示左栈，i=1 为右栈，x 是入栈元素。入栈成功返回 1，失败返回 0
{if(i<0||i>1){printf(“栈号输入不对\n”);exit(0);}
 if(S.top[1]-S.top[0]==1) {printf(“栈已满\n”);return(0);}
 switch(i)
   {case 0: S.V[++S.top[0]]=x; return(1); break;
    case 1: S.V[--S.top[1]]=x; return(1);
   }
```

```
}//push
```
(3)退栈操作
```
ElemType pop(stk S,int i)
//退栈。i 代表栈号，i=0 时为左栈，i=1 时为右栈。退栈成功时返回退
 栈元素
//否则返回-1
{if(i<0 || i>1){printf(“栈号输入错误\n”); exit(0);}
  switch(i)
   {case 0: if(S.top[0]==-1) {printf(“栈空\n”); return (-1); }
            else return(S.V[S.top[0]--]);
    case 1: if(S.top[1]==m {printf(“栈空\n”); return(-1);}
            else return(S.V[S.top[1]++]);
 }//switch      }//算法结束
```
(4)判断栈空
```
int Empty();
{return (S.top[0]==-1 && S.top[1]==m);
}
```
[算法讨论] 请注意算法中两栈入栈和退栈时的栈顶指针的计算。s1(左栈)是通常意义下的栈，而 s2(右栈)入栈操作时，其栈顶指针左移（减 1），退栈时，栈顶指针右移（加 1）。

3.15　设以数组 Q[m]存放循环队列中的元素，同时设置一个标志 tag，以 tag=0 和 tag=1 来区别在队头指针(front)和队尾指针(rear)相等时，队列状态为“空”还是“不空”。试编写相应的入队(QueueIn)和出队(QueueOut)算法。

【算法 3.15】

（1） 初始化
```
SeQueue QueueInit(SeQueue Q)
   {//初始化队列
    Q.front=Q.rear=0; Q.tag=0;
    return Q;
   }
```
（2） 入队
```
SeQueue QueueIn(SeQueue Q,int e)
 {//入队列
  if((Q.tag==1) && (Q.rear==Q.front)) printf("队列已满\n");
  else  {Q.rear=(Q.rear+1) % m;
         Q.data[Q.rear]=e;
        if(Q.tag==0) Q.tag=1; //队列已不空
        }
  return Q;
 }
```
(3)出队
```
ElemType QueueOut(SeQueue Q)
{//出队列
```

```
    if(Q.tag==0) { printf("队列为空\n"); exit(0);}
    else
        {Q.front=(Q.front+1) % m;
         e=Q.data[Q.front];
         if(Q.front==Q.rear) Q.tag=0;  //空队列
        }
    return(e);
}
```

3.16　假设用变量 rear 和 length 分别指示循环队列中队尾元素的位置和内含元素的个数。试给出此循环队列的定义，并写出相应的入队(QueueIn)和出队(QueueOut)算法。

【算法 3.16】

（1）循环队列的定义

```
typedef struct
 {ElemType  Q[m];       // 循环队列占 m 个存储单元
  int  rear,length;     // rear 指向队尾元素,length 为元素个数
 }SeQueue;
```

(2) 初始化

```
SeQueue QueueInit(SeQueue cq)
   //cq 为循环队列，本算法进行队列初始化
   { cq.rear=0; cq.length=0; return cq;
   }
```

(3) 入队

```
   SeQueue  QueueIn(SeQueue cq，ElemType x)
    //cq 是以如上定义的循环队列，本算法将元素 x 入队
   {if(cq.length==m) return(0);        // 队满
    else {cq.rear=(cq.rear+1) % m;     // 计算插入元素位置
          cq.Q[cq.rear]=x;             // 将元素 x 入队列
          cq.length++;                 // 修改队列长度
          }
    return (cq);
   }
```

（4） 出队

```
   ElemType  QueueOut(SeQueue cq)
  // cq 是以如上定义的循环队列，本算法是出队算法，且返回出队元素
   {if(cq.length==0) return(0);                    // 队空
    else { int front=(cq.rear-cq.length+1+m) % m;// 出队元素位置
           cq.length--;                            // 修改队列长度
           return (cq.Q[front]);                   // 返回对头元素
          }
    }
```

3.17　已知 Ackerman 函数定义如下：

$$\text{Akm(m, n)}=\begin{cases} n+1 & m=0 \\ akm(m-1,1) & m\neq 0, n=0 \\ akm(m-1,akm(m,n-1)) & m\neq 0, n\neq 0 \end{cases}$$

试写出递归和非递归算法。

【算法 3.17】

(1)递归算法

```
int Ack(int m,n)
 {if(m==0) return(n+1);
  else if(m!=0 && n==0) return(Ack(m-1,1));
       else return(Ack(m-1,Ack(m,m-1));
 }
```

（2） **非递归算法**

```
int Ackerman( int m, int n)
{int akm[m][n];
 int i,j;
  for(j=0;j<n;j++) akm[0][j];=j+1;
  for(i=1;i<m;i++)
   {akm[i][0]=akm[i-1][1];
    for(j=1;j<n;j++)
      akm[i][j]=akm[i-1][akm[i][j-1]];
    }
   return(akm[m][n]);
 }
```

3.18　假设称正读和反读都相同的字符序列为“回文”，例如，“abba“和“abccba”是“回文”，“abcde”和“ababab”则不是“回文“，试写一个算法，判别读入的一个以@为结束符的字符序列是否是“回文”。

【题目分析】将字符串前一半入栈，然后，栈中元素和字符串后一半进行比较。即将第一个出栈元素和后一半串中第一个字符比较，若相等，则再出栈一个元素与后一个字符比较，……，直至栈空，结论为字符序列是回文。在出栈元素与串中字符比较不等时，结论字符序列不是回文。

【算法 3.18】

```
int  sympthy(char str[],char s[])
  { int i=0,j,n;
   while (str[i]!= '\0' ) i++;      // 查字符个数
   if(i==1){printf("字符串是回文\n"); exit(0);}
   n=i;
   for(i=0;i<n/2;i++) s[i]=str[i]; //前一半字符入栈
   j=i;
   if(n%2==0) i++;                 //若 n 为偶数，
   else i=i+2;                     //若 n 为奇数
   while (i<n && str[i] == s[j])   //比较字符串是否是回文
        {i++; j--;}
```

```
        if (i==n) printf("字符串是回文\n");
            else printf("字符串不是回文\n");
        }
    }
```

3.19　设整数序列 a_1，a_2，…，a_n，给出求解最大值的递归程序。

【算法 3.19】

```
int MaxValue (int a[],int n)
    //设 n 个整数存于数组 a 中，本算法求解其最大值
    {if(n==1) max=a[1];
     else if a[n]>MaxValue(a,n-1) max=a[n];
          else max=MaxValue(a,n-1);
     return(max);
```

3.20　已知栈的三个运算定义如下：

PUSH(ST, x):元素 x 入 ST 栈；

POP(ST, x)：ST 栈顶元素出栈并赋给变量 x；

Sempty(ST)：判 ST 栈是否为空。

利用栈的上述运算来实现队列的三个运算：

QueueIn:插入一个元素入队列；

QueueOut:删除一个元素出队列；

QueueEmpty:判队列为空。

【题目分析】栈的特点是后进先出，队列的特点是先进先出。所以，需要用两个栈 s1 和 s2 模拟一个队列的操作：s1 作输入栈，逐个元素压栈，以此模拟队列元素的入队；出队时，将栈 s1 退栈并逐个压入栈 s2 中，s1 中最先入栈的元素，在 s2 中处于栈顶，s2 退栈，相当于队列的出队，实现了先进先出。显然，只有栈 s2 为空且 s1 也为空，才算是队列空。

【算法 3.20】

(1) 入队

```
int QueueIn(stack s1,ElemType x)
//s1 是容量为 n 的栈，栈中元素类型是 ElemType
//本算法将 x 入栈，若入栈成功返回 1，否则返回 0
   {if(top1==n && !Sempty(s2))       //top1 是栈 s1 的栈顶指针，是全局变量
      {printf("栈满");return(0);} //s1 满 s2 非空,这时 s1 不能再入栈
     if(top1==n && Sempty(s2))       //若 s2 为空，先将 s1 退栈,元素再压栈到 s2
       {while(!Sempty(s1))
          {POP(s1,x);PUSH(s2,x);}
       PUSH(s1,x); return(1);          //x 入栈，实现了队列元素的入队
   }
```

(2)出队

```
void QueueOut(stack s2,s1)
   //s2 是输出栈，本算法将 s2 栈顶元素退栈，实现队列元素的出队
```

```
{if(!Sempty(s2))          //栈 s2 不空，则直接出队
  {POP(s2,x);   printf(“出队元素为”,x);  }
 else                     //处理 s2 空栈
   if(Sempty(s1))
      {printf(“队列空”);exit(0);}//若输入栈也为空，则判定队空
   else                   //先将栈 s1 倒入 s2 中，再作出队操作
     {while(!Sempty(s1)) {POP(s1,x);PUSH(s2,x);}
      POP(s2,x);          //s2 退栈相当队列出队
      printf(“出队元素\n”, x);
     }
}
```

(3)判栈空

```
int QueueEmpty ()
{//本算法判用栈 s1 和 s2 模拟的队列是否为空
 if(Sempty(s1) && Sempty(s2))  return(1);//队列空
 else return(0);                          //队列不空
}
```

[**算法讨论**]算法中假定栈 s1 和栈 s2 容量相同。出队从栈 s2 出，当 s2 为空时，若 s1 不空，则将 s1 倒入 s2 再出栈。入队在 s1，当 s1 满后，若 s2 空，则将 s1 倒入 s2，之后再入队。因此队列的容量为两栈容量之和。元素从栈 s1 倒入 s2，必须在 s2 空的情况下才能进行，即在要求出队操作时，若 s2 空，则不论 s1 元素多少（只要不空），就要全部倒入 s2 中。

第 4 章　串

一、基础知识题

4.1 简述下列每对术语的区别：

空串和空格串； 串常量与串变量；主串和子串；串变量的名字和串变量的值；静态分配的顺序串与动态分配的顺序串。

【解答】 不含任何字符的串称为空串，其长度为 0。仅含有空格字符的串称为空格串，其长度为串中空格字符的个数。空格符可用来分割一般的字符，便于人们识别和阅读，但计算串长时应包括这些空格符。空串在串处理中可作为任意串的子串。

用引号（数据结构教学中通常用单引号，而 C 语言中用双引号）括起来的字符序列称为串常量，其串值是常量。串值可以变化的量称为串变量。

串中任意个连续的字符组成的子序列被称为该串的**子串**。包含子串的串又被

称为该子串的**主串**。子串在主串中第一次出现的第一个字符的位置称子串在主串中的**位置**。

串变量与其它变量一样，要用名字引用其值，串变量的名字也是标识符，串变量的值可以修改。

串的存储也有静态存储和动态存储两种。静态存储指用一维数组，通常一个字符占用一个字节，需要静态定义串的长度，具有顺序存储结构的优缺点。若需要在程序执行过程中，动态地改变串的长度，则可以利用标准函数 malloc()和 free()动态地分配或释放存储单元，提高存储资源的利用率。在 C 语言中，动态分配和回收的存储单元都来自于一个被称之为“堆”的自由存储区，故该方法可称为**堆分配存储**。类型定义如下所示：

```
typedef struct
{ char *str;
  int length;
}HString;
```

4.2 设有串 S=’good’，T=’I␣am␣a␣student’，R=’！’，求：

（1）StringConcat(T, R)　（2）SubString(T, 8, 7)

（3）StringLength(T)　（4）Index(T, ’a’)

（5）StringInsert(T, 8, S)

（6）Replace(T, SubString(T, 8, 7), ’teacher’)

【解答】

(1) StringConcat(T, R)=’I␣am␣a␣student!’

(2) SubString(T, 8, 7）=’student’

(3) StringLength(T)=14

(4) Index(T, ’a’)=3

(5) StringInsert(T, 8, S)=’I␣am␣a␣goodstudent’

(6) Replace(T, SubString(T,8,7),’teacher’)=’I␣am␣a␣teacher’

4.3 若串 S_1=‘ABCDEFG’，S_2=‘9898’，S_3=‘###’，S_4=‘012345’，执行 concat(replace(S_1, substr(S_1, length(S_2), length(S_3)), S_3), substr(S_4, index(S_2, ‘8’), length(S_2)))

操作的结果是什么？

【解答】

concat(replace(S_1,substr(S_1,length(S_2),length(S_3)),S_3),substr(S_4 , index(S_2, ‘8’),length(S_2)))

= concat(replace(S_1,substr(S_1,4,3),S_3),substr(S_4,2,4))

= concat(replace(S_1,’DEF’,S_3),’1234’)

= concat(‘ABC###G’,’1234’)

= ‘ABC###G1234’

4.4 设 S 为一个长度为 n 的字符串，其中的字符各不相同，则 S 中的互异的非平凡子串（非空且不同于 S 本身）的个数是多少？

【解答】长度为 n 的字符串中互异的非平凡子串（非空且不同于 S 本身）的个数计算如下：

长度为 1 的子串有 n 个，长度为 2 的子串有 n-1 个，长度为 3 的子串有 n-2 个，……，长度为 n-1 的子串有 2 个，长度为 n 的子串有 1 个（按题意要求这个子串就是 S 本身，不能计算在总数内）。故子串个数为:(2+n)*(n-1)/2

4.5 KMP 算法(字符串匹配算法)较 Brute(朴素的字符串匹配)算法有哪些改进?

【解答】KMP 算法的最大特点是主串指针不回溯，在整个匹配过程中，对主串从头到尾扫描一遍，对于处理存储在外存上的大文件是非常有效的。

虽然 Brute(朴素的字符串匹配)算法的时间复杂度是 O(n*m)，但在一般情况下，其实际的执行时间近似于 O(n+m)，因此至今仍被采用。KMP 算法仅当主串与模式间存在许多“部分匹配”的情况下才显得比 Brute(朴素的字符串匹配)算法快得多。

4.6 求串’ababaaababaa’ 的 next 函数值。

【解答】

j	1	2	3	4	5	6	7	8	9	10	11	12
t 串	a	b	a	b	a	a	a	b	a	b	a	a
next[j]	0	1	1	2	3	4	2	2	3	4	5	6

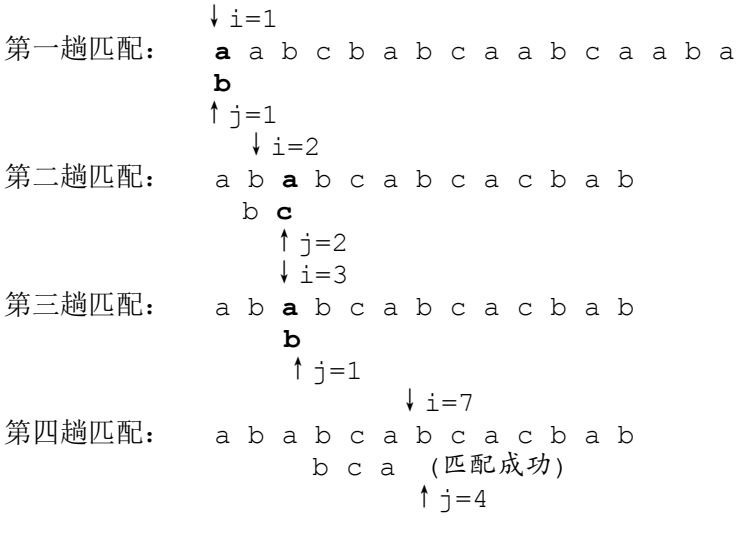
4.7 模式串 t=’abcaabbcaabdab’，求模式串的 next 和 nextval 函数的值。

【解答】

j	1	2	3	4	5	6	7	8	9	10	11	12	13	14
t 串	a	b	c	a	a	b	b	c	a	a	b	d	a	b
next[j]	0	1	1	1	2	2	3	1	1	2	2	3	1	2
nextval[j]	0	1	1	0	2	1	3	1	0	2	1	3	0	1

4.8 对 S=’aabcbabcaabcaaba’，T=’bca’，画出以 T 为模式串，S 为目标串的匹配过程。

解答】



4.9 选择题：下面关于串的的叙述中，哪一个是不正确的？

A. 串是字符的有限序列

B. 空串是由空格构成的串

C. 模式匹配是串的一种重要运算

D. 串既可以采用顺序存储，也可以采用链式存储

【解答】 B

4.10 选择题：串是一种特殊的线性表，下面哪个叙述体现了这种特殊性？

A. 数据元素是一个字符　　B. 可以顺序存储

C. 数据元素可以是多个字符　　D. 可以链接存储

【解答】A

二、算法设计题

4.11 试写出用单链表表示的字符串结点类型的定义，并依次实现它的计算串长度、串赋值、判断两串相等、求子串、两串连接、求子串在串中位置的 6 个函数。要求每个字符串结点中只存放一个字符。

【算法 4.11】

单链表结点的类型定义如下：

```
typedef struct Node
  {char data;
   struct Node *next;
  }LNode, *LinkedString;
```

(1) 计算串长度

```
int  StringLength(LinkedString L)
 {//求带头结点的用单链表表示的字符串的长度
  p=L->next;                  //p指向串中第一个字符结点
  j=0;                        //计数器初始化
  while(p)
      {j++; p=p->next;}       //计数器加1，指针后移
  return  j;
 }
```

(2) 串赋值

```
LinkedString StringAssign(LinkedString L)
{//字符串赋值
```

```
  LNode *s,*p,*r;
  s=(char *)malloc(sizeof(char));
   s->next=null; //头结点
   r=s;          //r记住尾结点
   L=L->next;    //串中第一字符
  while(L)
    {p=(char *)malloc(sizeof(char));
     p->data=L->data;  //赋值
     p->next=r->next;  //插入链表中
     r->next=p;
     r=p;              //指向新的尾结点
     L=L->next;
    }
  return s;
 }
```

(3) 判断两串相等

```
 int StringEqual(LinkedString s, LinkedString q)
 {//判断字符串的相等
  LNode *p=s->next,*r=q->next;
   while(p && r)
     if(p->data==r->data)
       {p=p->next; r=r->next;}
     else return 0;
     if(!p && !r)
         return 1;
      else
          return 0;
 }
```

(4) 求子串

```
 LinkedString Substring(LinkedString S, int start, int i)
 {//求串S从start开始的i个字符所组成的子串
    LNode *sub,*p,*r,*s=S->next;
    int j=1;
   if(start<1 || i<0) {printf(“参数错误\n”); exit(0);}
   sub=(char *)malloc(sizeof(char));
    sub->next=null; //头结点
    r=sub;
   while（s!=null && j<start）//查找起始结点
      {s=s->next; j++;}
   if（s==null）
      {printf(“起始位置太大\n”); exit(0);}
   else
     {j=0;
```

```
        while（s!=null && j<i）//若 i 太大，则截取到串尾
           {p=(char *)malloc(sizeof(char));
            p->data=s->data;
            p->next=r->next;
            r->next=p;
            r=p;
            j++;
           }
        }
    return sub;
    } //算法结束
```

(5) 两串连接

```
    LinkedString Concat(LinkedString S, LinkedString T)
    {//求串S和串T的连接，返回结果串
      LNode *p=S;
      while(p->next!=null)  //查找串尾
         p=p->next;
      p->next=T->next;
      free(T);              //释放头结点
      return S;
    }
```

(6) 求子串在主串中位置

```
    int Index(LinkedString S, LinkedString T)
    {//求子串在主串中的位置，成功时返回其在主串的位序，否则返回0表示失
败
      int i=1;j=1;        //i记主串当前结点，j记子串在主串中的位序
      p=S->next;          //p 是每趟匹配时 S 中的起始结点的指针
      q=S->next;          //q 是 S 中的工作指针
      r=T->next;          //r 是 T 中的工作指针
      while（q && r）
      if（q->data==r->data）
         {q=q->next;r=r->next; i++;} //对应字母相等，指针后移
      else        //失配时,S 起始结点后移,T 从首字符结点开始
         {i++;
          j=i;   //j 记子串在主串中的位序
          q=p->next; p=p->next;  r=T->next;
          }
      if（r==null）return (j);
        //j 是子串在主串的位序，p 是子串在主串第一字符结点的指针
      else return 0;  //T 并未在 S 中出现
    } //算法结束
```

4.12　用顺序结构存储的串 S,编写算法删除 S 中第 i 个字符开始的 j 个字符。

【题目分析】我们使用教材中定义的串的顺序存储结构。

【算法 4.12】

```
void StringDelete(STring S,int i,int j)
   {//在顺序存储的串 S 中删除自第 i 个字符开始的 j 个字符
    if(i<1 || i>S.curlen || j<0) {printf("参数错误\n"); exit(0);}
    if(i+j-1>S.curlen) //若 j 太大，则从第 i 个字符起，删除到串尾
      j=S.curlen-i+1;
    for(ii=i+j-1;ii<S.curlen;ii++)
      S.ch[i++]=S.ch[ii];
    S.curlen-=j;
   }
```

4.13　输入一个字符串，内有数字和非数字字符，如：ak123x456 17960?302gef4563，将其中连续的数字作为一个整体，依次存放到一数组 a 中，例如 123 放入 a[0]，456 放入 a[1]，……　。编写算法统计其共有多少个整数，并输出这些整数。

【题目分析】在一个字符串内，统计含多少整数的问题，核心是如何将数从字符串中分离出来。从左到右扫描字符串，初次碰到数字字符时，作为一个整数的开始。然后进行拼数，即将连续出现的数字字符拼成一个整数，直到碰到非数字字符为止，一个整数拼完，存入数组，再准备下一整数，如此下去，直至整个字符串扫描到结束。

【算法 4.13】

```
int CountInt（）
  // 从键盘输入字符串，连续的数字字符算作一个整数，统计其中整数的个数
   {char ch; int i=0, a[];        // 整数存储到数组 a，i 记整数个数
    scanf("%c", &ch);             // 从左到右读入字符串
    while（ch!='#'）              // '#'是字符串结束标记
      if(ch>='0' && ch<='9' ) // 是数字字符
        {num=0；                  // 数初始化
         while（ch>='0' && ch<='9'）// 拼数
           {num=num*10+'ch'-'0';
            scanf("%c", &ch);
           }
         a[i++]=num;
         if（ch!='#'）            // 若拼数中输入了'#'，则不再输入
            scanf ("%c", &ch);
        }
      else scanf ("%c", &ch);  //输入非数字且非#时，继续输入字符
    printf ("共有%d 个整数，它们是：\n",i);
    for（j=0； j<i; j++）
      {printf ("%6d", a[j]);
       if（(j+1）%10==0）printf ("\n"); } // 每 10 个数输出在一行上
```

```
} //CountInt
```

[算法讨论]假定字符串中的数均不超过 32767，否则，需用长整型数组及变量。

4.14　输入一文本行（最多 80 个字符），编写算法求某一个不包含空格的子串在文本行中出现的次数。

【题目分析】本题是模式匹配问题。将文本行看作主串，先用子串定位函数确定子串在主串中的位置，如子串在主串中，则计数，并将子串后的部分主串当作新主串，继续查子串是否在主串中，如此下去，直到子串不在主串中为止。下面用串的基本操作编写求解本题的算法。并给出子串定位函数 Index。

【算法 4.14】

```
int Index(String S,  String T)
{//子串定位函数，成功返回子串 T 的第一字符在 S 中的位置，否则返回 0
  n=StringLength(S); m=StringLength(T);
   i=1;
  while(i<=n-m+1)  //当 i 加上 T 的长度超过串 S 的长度时结束
     {StringAssign(sub, SubString(S, i, m));//取字串 sub
      if(StringEqual(sub, T)!=0) i++;
      else return i ;
     }//while
  return 0;          //S 中不存在与 T 相等的子串
}//Index

int NumberofSub(String main,  String sub)
  {//某一个不包含空格的子串在文本行中出现的次数
   int i=0,j;
   int n=StringLength(main), m=StringLength(sub);
   j=Index(main,sub);
   while(j!=0)
     {i++;
      StringAssign(main,SubString(main,j+m,n-(j+m)+1); //新主串
      n=StringLength(main); //求新主串长度
      j=Index(main,sub);
     }
   return i;
  }
```

4.15　函数 **void** insert(char*s, char*t, int pos)表示将字符串 t 插入到字符串 s 中，插入位置为 pos。请编写实现该函数的算法。假设分配给字符串 s 的空间足够让字符串 t 插入。

【题目分析】本题是字符串的插入问题，要求在字符串 s 的 pos 位置，插入字符串 t。首先应查找字符串 s 的 pos 位置，将第 pos 个字符到字符串 s 尾的子串向后移动字符串 t 的长度，然后将字符串 t 复制到字符串 s 的第 pos 位置后。

对插入位置 pos 要验证其合法性，小于 1 或大于串 s 的长度均为非法，因题目

假设给字符串 s 的空间足够大，故对插入不必判溢出。

【算法 4.15】

```
void  insert(char *s,char *t,int pos)
  //将字符串 t 插入字符串 s 的第 pos 个位置。
{int i=1,x=0;
 char *p=s,*q=t;  //p，q 分别为字符串 s 和 t 的工作指针
 if(pos<1) {printf("pos 参数位置非法\n");exit(0);}
 while(*p!='\0' && i<pos) {p++;i++;} //查 pos 位置
    //若 pos 小于串 s 长度，则查到 pos 位置时，i=pos。
 if(*p =='\0') {printf("%d 位置大于字符串 s 的长度\n",pos);exit(0);}
 else       //查找字符串的尾
   while(*p!= '\0')
      {p++; i++;}     //查到尾时，i 为字符'\0'的下标，p 也指向'\0'
 while(*q!= '\0')
      {q++; x++; }    //查找字符串 t 的长度 x，循环结束时 q 指向'\0'
 for(j=i;j>=pos ;j--)
   {*(p+x)=*p; p--;} //串 s 的 pos 后的子串右移，空出串 t 的位置
 q--;  //指针 q 回退到串 t 的最后一个字符
 for(j=1;j<=x;j++) *p--=*q--;  //将 t 串插入到 s 的 pos 位置上
```

【算法讨论】 串 s 的结束标记('\0')也后移了，而串 t 的结尾标记不应插入到 s 中。

4.16　设 S="$S_1S_2\cdots S_n$"是一个长为 N 的字符串，存放在一个数组中，编写算法将 S 改造之后输出：

(1)将 S 的所有第偶数个字符按照其原来的下标从大到小的次序放在 S 的后半部分；

(2)将 S 的所有第奇数个字符按照其原来的下标从小到大的次序放在 S 的前半部分；

例如：S='ABCDEFGHIJKL'，则改造后的 S 为'ACEGIKLJHFDB'。

【题目分析】对读入的字符串的第奇数个字符，直接放在数组前面，对第偶数个字符，先入栈，到读字符串结束，再将栈中字符出栈，送入数组中。

【算法 4.16】

```
void ReArrangeString()
 //将字符串的第偶数个字符放在串的后半部分，第奇数个字符放在前半部分
{char ch,s[],stk[];      //s 和 stk 是字符数组（表示字符串）和字符栈
 int i=1,j;              //i 和 j 字符串和字符栈指针
 while((ch=getchar())!='#')
    s[i++]=ch;           //读入字符串,'#'是字符串结束标志
 s[i]='\0';              //字符数组中字符串结束标志
 i=1;j=1;
 while(s[i])             //改造字符串
   {if(i%2==0)
       stk[i/2]=s[i];
```

```
        else
           s[j++]=s[i];
        i++;
      }//while
    i--; i=i/2;          //i 先从’\0’后退，然后其含义是第偶数字符的
个数
    while(i>0)
      s[j++]=stk[i--]  //将第偶数个字符逆序填入原字符数组
  }
```

第 5 章　数组和广义表

一、基础知识题

5.1 已知二维数组 A[3][5]，其每个元素占 3 个存储单元，并且 A[0][0]的存储地址为 1200。求元素 A[1][3]的存储地址（分别对以行序和列序为主序存储进行讨论），该数组共占用多少个存储单元？

【解答】按照以行序为主序存储公式：

LOC(i，j)=LOC(c_1，c_2)+[(i-c_1)*(d_2-c_2+1)+(j-c_2)]*L

在 C 语言中有：LOC(i，j)=LOC(0，0)+(i*(d_2+1)+j)*L

则：LOC(A[1][3])=1200+(1*5+3)*3=1224　　(按行序存储)

LOC(A[1][3])=1200+(3*3+1)*3=1230　　(按列序存储)

5.2有一个 10 阶的对称矩阵 A，采用压缩存储方式以行序为主序存储，A[1][1]为第一元素,其存储地址为 1,每个元素占一个地址空间,求 A[7][5]和 A[5][6]的地址。

【解答】按照公式：

$$k=\begin{cases}\dfrac{i(i-1)}{2}+j & \text{当} i\geq j \quad 1<=i,j<=n \\ \dfrac{j(j-1)}{2}+i & \text{当} i<j \quad 1<=i,j<=n\end{cases} \qquad 1<=k<=n(n+1)/2$$

LOC(A[7][5])=7(7-1)/2＋5 = 26

LOC(A[5][6])=LOC(A[6][5])=6(6-1)/2+5=20

5.3设有一个二维数组 A[m][n]，设 A[0][0]存放位置在 644，A[2][2]存放位置在 676，每个元素占一个空间，问 A[3][3]存放在什么位置？

【解答】因为 A[0][0]存放位置在 644，A[2][2]存放位置在 676，每个元素占一个空间，说明一行有 15 个元素（算法：（676－2－644）/2)。A[3][3]存放位置是 692。

5.4　二维数组 A[9][10]的元素都是 6 个字符组成的串，请回答下列问题：

（1）存放 A 至少需要(　)个字节；

（2）A 的第 7 列和第 4 行共占(　)个字节；

（3）若 A 按行存放，元素 A[7][4]的起始地址与 A 按列存放时哪一个元素的起始地址一致。

【解答】按照题 5.1 给出的公式：

（1）存放 A 需要 9*10*6=540 个字节

（2）A 的第 7 列和第 4 行共占(9+10-1)*6=108 个字节

（3）LOC(A[7][4])= LOC(A[0][0])+[7*10+4]*L　(按行序存储)

LOC(A[i][j])= LOC(A[0][0])+[j*9+i]*L （按列序存储，

0<=i<=8,0<=j<=9)所以,i=2,j=8。

即元素 A[7][4]的起始地址与 A 按列存放时 A[2][8]的起始地址一致。

5.5 将一个 A[1..100，1..100]的三对角矩阵，按行优先存入一维数组 B[1..m]中，试确定 m 的值，并求 A 中元素 A[77，78]（即元素下标 i=77，j=78）在 B 数组中的位置 K。

【解答】三对角矩阵共 3n-2 个元素，存入 B[1..3n-2]中,元素在一维数组 B 中的下标 k 和元素在矩阵中的下标 i 和 j 的对应关系为：

k = 3(i-1) （主对角线左下角，即 i=j+1)

k = 3(i-1)+1 （主对角线上，即 i=j)

k = 3(i-1)+2 （主对角线上，即 i=j-1)

由以上三式，得

k=2(i-1)+j (1≤i,j≤n; 1≤k≤3n-2)

故 A[77，78]（即元素下标 i=77，j=78）在 B 数组中的位置为 k=2(77-1)+78=230。

5.6 设有对称矩阵 A[n][n]，将其上三角元素逐行存于数组 B[0..m-1]中，使得 B[k]=a[i，j]且 k=f_1(i)+f_2(j)+c。试推导出函数 f_1，f_2和常数 c。

【解答】上三角矩阵第一行（下标为 0）有 n 个元素，下标 i-1 行有 n-(i-1）个元素，第一行（下标 0）到下标 i-1 行是梯形，而第 i 行上第 j 个元素（即 a_{ij}）是第 i 行上第 j-i+1 个元素，故元素 A_{ij}在一维数组中的存储位置（下标 k）为：

k=(n+(n-(i-1)))i/2+(j-i+1)=(2n-i+1)i/2+j-i+1

将上面的等式进一步整理为：

k=（n-1/2）i-i^2/2+j+1，

则得 f_1(i)=(n-1/2)i-i^2/2，f_2(j)=j，c=1。

5.7设 A 和 B 均为下三角矩阵，每一个都有 n 行。因此在下三角区域中各有 n(n+1)/2 个元素。另设有一个二维数组 C，它有 n 行 n+1 列。试设计一个方案，将两个矩阵 A 和 B 中的下三角区域元素存放于同一个 C 中。要求将 A 的下三角区域中的元素存放于 C 的下三角区域中，B 的下三角区域中的元素转置后存放于 C 的上三角区域中。并给出计算 A 的矩阵元素 a_{ij}和 B 的矩阵元素 b_{ij}在 C 中的存放位置下标的公式。

【解答】

```
void UnionTrans(int A[][],int B[][],int C[][],int n)
 //本算法将 n 阶方阵的下三角矩阵 A 和 B 置于 C 中，B 要逆置
  {for(i=0;i<n;i++)
    {for(j=0;j<=i;j++)   C[i][j]=A[i][j];
      for(j=i+1;j<=n;j++) C[i][j]=B[j-1][i];
    }
  }
```

A 的矩阵元素 a_{ij}和 B 的矩阵元素 b_{ij}在 C 中的存放位置下标的公式:

A[i][j]=C[i][j]; { 0<=i<n, 0<=j<=i}

B[i][j]=C[j][i+1]; { 0<=i<n, i<=j<=n}

5.8什么是广义表？请简述广义表和线性表的主要区别。
【解答】广义表是零至多个元素的有限序列，广义表中的元素可以是原子，也可以是子表。从“元素的有限序列”角度看，广义表满足线性结构的特性：在非空线性结构中，只有一个称为“第一个”的元素，只有一个称为“最后一个”的元素，第一元素有后继而没有前驱，最后一个元素有前驱而没有后继，其余每个元素有唯一前驱和唯一后继。从这个意义上说，广义表属于线性结构。当广义表中的元素都是原子时，广义表就蜕变为线性表。

5.9求广义表 D=(a，(b，()，c)，((d)，e))的长度和深度。
【解答】3 和 3

5.10　设广义表 L=(()，())，试问 GetHead(L)，GetTail(L)的值，求 L 的长度和深度各为多少?
【解答】GetHead(L)和 GetTail(L)的值分别是()和(())。
L 的长度和深度都是 2。

5.11　求下列广义表运算的结果：

(1) GetHead ((p，h，w))
(2) GetTail((b，k，p，h))
(3) GetHead(GetTail(((a，b)，(c，d))))
(4) GetTail(GetHead(((a，b)，(c，d))))

【解答】(1) GetHead ((p,h,w))=p
(2) GetTail((b,k,p,h))=(k,p,h)
(3)GetHead(GetTail(((a，b)，(c，d)))= GetHead(((c,d)))=(c,d)
(4) GetTail(GetHead(((a，b)，(c，d))))=GetTail((a,b))=(b)

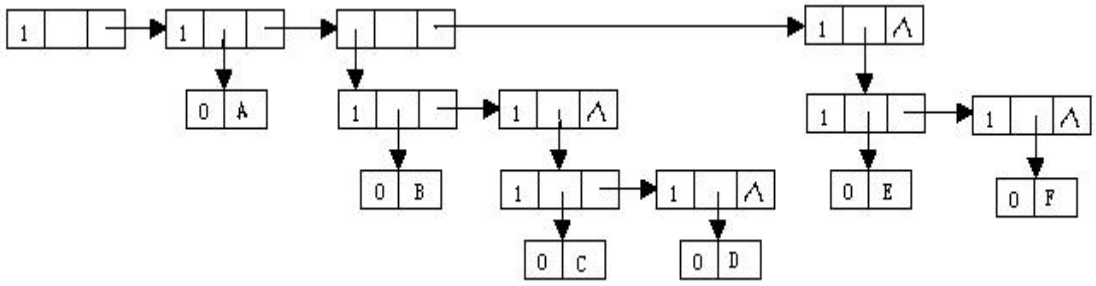
5.12　利用广义表的 GetHead 和 GetTail 操作写出函数表达式，把以下各题中的单元素 banana 从广义表中分离出来：
(1) (apple，pear，banana，orange)
(2) ((apple，pear)，(banana，orange))
(3) ((((apple)))，((pear))，(banana)，orange)
(4) (apple，(pear，(banana)，orange))
【解答】
(1) GetHead(GetTail(GetTail((apple,pear,banana,orange))))
(2) GetHead(GetHead(GetTail((apple,pear),(banana,orange)))))
(3)
GetHead(GetHead(GetTail(GetTail(((((apple))),((pear)),(banana),orange)))))
(4)
GetHead(GetHead(GetTail(GetHead(GetTail((apple,(pear,(banana),orange)))))))

5.13　画出下列广义表的两种存储结构图

(1) ((), A, (B, (C, D)), (E, F))
(2) ((), a, ((b, c), (), d), ((e)))
(3) (((a)), (b), c, (a), (((d, e))))
(4) (((b, c), d), (a), ((a), ((b, c), d)))

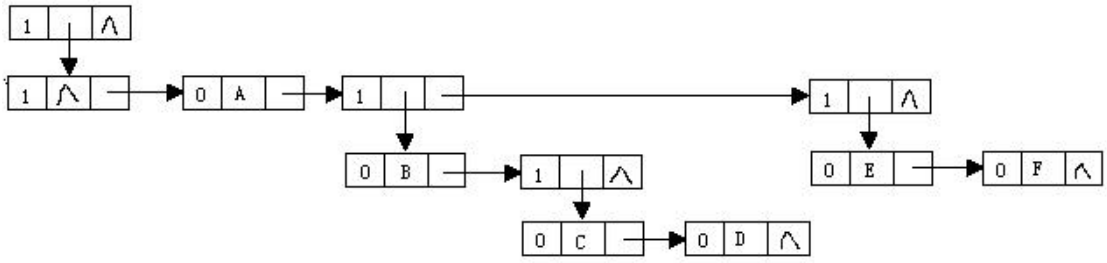
【解答】(1)广义表的第一种存储结构的理论基础是，非空广义表可唯一分解成表头和表尾两部分，而由表头和表尾可唯一构成一个广义表。这种存储结构中，原子和表采用不同的结点结构（“异构”，即结点域个数不同）。原子结点两个域：标志域 tag=0 表示原子结点，域 atom 表示原子的值；子表结点三个域：tag=1 表示子表，hp 和 tp 分别是指向表头和表尾的指针。在画存储结构时，对非空广义表不断进行表头和表尾的分解，表头可以是原子，也可以是子表，而表尾一定是表（包括空表）。下面是本题的第一种存储结构图。

(1) ((), A, (B, (C, D)), (E, F))



广义表的第二种存储结构的理论基础是，非空广义表最高层元素间具有逻辑关系：第一个元素无前驱有后继，最后一个元素无后继有前驱，其余元素有唯一前驱和唯一后继。有人将这种结构看作扩充线性结构。这种存储结构中，原子和表均采用三个域的结点结构(“同构”)。结点中都有一个指针域 tp 指向后继结点。原子结点中还包括标志域 tag=0 和原子值域 atom;子表结点还包括标志域 tag=1 和指向子表的指针 hp。在画存储结构时，从左往右一个元素一个元素的画，直至最后一个元素。下面是本题的第二种存储结构图。

(1) ((), A, (B, (C, D)), (E, F))



5.14 选择题：对矩阵压缩存储是为了:

A.方便运算　　B.方便存储　　C.提高运算速度　　D.减少存储空间

【解答】D

5.15 选择题：下面说法不正确的是:

A. 广义表的表头总是一个广义表　　B. 广义表的表尾总是一个广义表

C. 广义表难以用顺序存储结构　　D. 广义表可以是一个多层次的结构

【解答】A

二、算法设计题

5.16 设矩阵 A 中的某一个元素 A[i][j]是第 i 行中的最小值，而又是第 j 列中的最大值，则称 A[i][j]为矩阵中的一个鞍点，请写出一个可确定该鞍点位置的算法（如果这个鞍点存在），并给出算法的时间复杂度。

【题目分析】 寻找马鞍点最直接的方法,是在一行中找出一个最小值元素,然后检查该元素是否是元素所在列的最大元素,如是,则输出一个马鞍点,时间复杂度是O(m*(m+n)).本算法使用两个辅助数组max和min,存放每列中最大值元素的行号和每行中最小值元素的列号,时间复杂度为 O（m*n+m），但比较次数比前种算法会增加，也多使用向量空间。

【算法 5.16】

```
int m=10, n=10;
void Saddle(int A[m][n])
 //A 是 m*n 的矩阵,本算法求矩阵 A 中的马鞍点
{int i,j, max[n]={0}, //max 数组存放各列最大值元素的行号,初始化为行号 0
     min[m]={0};       //min 数组存放各行最小值元素的列号,初始化为列号 0
  for(i=0;i<m;i++)      //选各行最小值元素和各列最大值元素.
    for(j=0;j<n;j++)
      {if(A[max[j]][j]<A[i][j]) max[j]=i;  //修改第 j 列最大元素的行号
       if(A[i][min[i]]>A[i][j]) min[i]=j;  //修改第 i 行最小元素的列号
      }
  for(i=0;i<m;i++)
   {j=min[i];              //第 i 行最小元素的列号
    if(i==max[j])
       printf("A[%d][%d]是马鞍点，元素值是%d",i,j,A[i][j]);//是马鞍点
    }
 }// Saddle
```

5.17　设稀疏矩阵用三元组表示，编写算法将两个稀疏矩阵相加，结果矩阵仍用三元组表示。

【题目分析】设稀疏矩阵为 A 和 B,算法的基本思想是：依次扫描 A 和 B 的行号和列号： 若 A 的当前项的行号小于 B 的当前项的行号，则将 A 的当前项存入 C

中；若 A 的当前项的行号大于 B 的当前项的行号，则将 B 的当前项存入 C 中；若 A 的当前项的行号等于 B 的当前项的行号，则比较其列号，将列号较小的当前项存入 C 中；如果行号与列号都相等，且对应的元素值相加后不为 0，则将该非零元素存入 C 中。

【算法 5.17】

```
void matadd(TSMatrix A, TSMatrix B, TSMatrix C)
{//采用三元组顺序表方式存储，实现稀疏矩阵相加
 C.m=A.m;   C.n=A.n;   C.len=0;        //结果矩阵 C 初始化
 i=0; j=0;
 if(A.len || B.len)
  {k＝0;
    while(i<A.len && j<B.len)
      if(A.data[i].row<B.data[j].row) //A 的行号小于 B 的行号
         {C.data[k].row=A.data[i].row;
          C.data[k].col=A.data[i].col;
          C.data[k++].e=A.data[i++].e;
          }
       else if(A.data[i].row>B.data[j].row) //A 的行号大于 B 的行号
         {C.data[k].row=B.data[j].row;
          C.data[k].col=B.data[j].col;
          C.data[k++].e=B.data[j++].e;
          }
          else       //A 的行号等于 B 的行号
          if(A.data[i].col<B.data[j].col) //A 的列号小于 B 的列号
            {C.data[k].row=A.data[i].row;
             C.data[k].col=A.data[i].col;
             C.data[k++].e=A.data[i++].e;
            }
          else if(A.data[i].col>B.data[j].col)//A 的列号大于 B 的列号
                {C.data[k].row=B.data[j].row;
                C.data[k].col=B.data[j].col;
                C.data[k++].e=B.data[j++].e;
                }
               else //A 的行、列号分别等于 B 的行、列号
```

```
            {num=A.data[i++].e+B.data[j++].e;//对应元素相
            加
              if(num!=0)     //和不为0，则存入C中
                 {C.data[k].row=A.data[i].row;
                  C.data[k].col=A.data[i].col;
                  C.data[k++].e=num;
                 }
              } //else A的行、列号分别等于B的行、列号

        while(i<A.len) //A的元素还没有扫描完
          {C.data[k].row=A.data[i].row;
           C.data[k].col=A.data[i].col;
           C.data[k++].e=A.data[i++].e;
          }
        while(j<B.len) //B的元素还没有扫描完

          {C.data[k].row=B.data[j].row;
           C.data[k].col=B.data[j].col;
           C.data[k++].e=B.data[j++].e;
          }
      C.len=k;
      return C;
  }
```

5.18　三对角矩阵 A[n][n]，将其三条对角线上的元素逐行地存储到向量B[0..3n-3]中，使得B[k]=a_{ij}，写一算法求三对角矩阵在这种压缩存储表示下的转置矩阵。

【题目分析】对角矩阵逐行存储到B[0..3n-3]中时，三对角线上元素的坐标满足|j-i|<=1。

【算法5.18】

```
  void compress(int A[n][n],int B[ ],int n)
  {//三对角矩阵A[0..n-1,0..n-1]，将三条对角线上的元素逐行存放于数组B
  中
   k=0;
   for(i=0;i<n;i++)
     {if(i==0)                  //第一行（行下标0）上两个元素
        for(j=i;j<=i+1;j++)//其余每行上三个元素
```

```
            B[k++]=A[i][j];
        else if(i==n-1)        //最后一行（行下标 n-1）上两个元
素
              for(j=i-1;j<=i;j++)
                B[k++]=A[i][j];
            else
               for(j=i-1;j<=i+1;j++)
                 B[k++]=A[i][j];
}
```

(2)已知 k 求 i，j 时，则下标的对应关系是：

i=(k+1)/3；（0≤k≤3n-3） // (k+1)/3 取小于(k+1)/3 的最大整数

$$j=\begin{cases} i-1 & (\text{当}\quad (k+1)\%3=0) \\ i & (\text{当}\quad k\%3=0) \\ i+1 & (\text{当}\quad (k+1)\%3=0) \end{cases}$$

```
void uncompress(int B[ ],int A[n][n],int n)
{//由数组 B[0..3n-3]确定三对角矩阵 A[0..n-1,0..n-1]

for(i=0;i<n;i++)        //矩阵初始化
   for(j=0;j<n;j++)
      A[i][j]=0;
 for(k=0;k<=3*n-3;k++)
   {i=(k+1)/3;
    if(k+1)%3==0) j=i-1;
      else if(k%3==0) j=i;
        else if(k-1)%3==0) j=i+1;
     A[j][i]=B[k];    //转置
   }
}
```

5.19 设 A[1..100]是一个记录构成的数组，B[1..100]是一个整数数组，其值介于 1 至 100 之间，现要求按 B[1..100]的内容调整 A 中记录的次序，比如当 B[1]=11 时，则要求将 A[1]的内容调整到 A[11]中去。规定可使用的附加空间为 O(1)。

【题目分析】题目要求按 B 数组内容调整 A 数组中记录的次序,可以从 i=1 开始，检查是否 B[i]=i。如是，则 A[i]恰为正确位置，不需再调；否则，B[i]=k≠i，则将 A[i]和 A[k]对调，B[i]和 B[k]对调，直到 B[i]=i 为止。

【算法 5.19】

```
void CountSort(rectype A[],int B[])
```

```
//A 是 100 个记录的数组，B 是整型数组，本算法利用数组 B 对 A 进行计数排序
{int i,j,n=100;
 i=1;
 while(i<=n)
  {if(B[i]!=i)  //若 B[i]=i 则 A[i]正好在自己的位置上，则不需要调整
    { j=i;
      while(B[j]!=i)
        { k=B[j];  B[j]=B[k]; B[k]=k;// B[j]和 B[k]交换
          r0=A[j];A[j]=A[k]; A[k]=r0;//r0 是数组 A 的元素类型,A[j]和 A[k]交换
        }
     i++;
    } //完成了一个小循环，第 i 个已经安排好
 }//算法结束
```

5.20　请编写递归算法，逆转广义表中的数据元素。例如：将广义表：(a,((b,c),()),(((d),e),f))逆转为：((f,(e,(d))),((),(c,b)),a)。

【题目分析】采用广义表的第二种存储结构（扩充的线性链表），类似于单链表的逆置。

【算法 5.20】

```
Void   GListInvert(GList p,GList t)
    //将广义表 p 逆置为广义表 t
   {GList q=null;
   while(p)
      {if(p->tag!=0)  //若是字表，则递归逆置
          {m=p->val.hp;
          GListInvert(m,n)  ;
          p->val.hp=n;
          };
        r=p->tp;  //保留后继
        p->tp=q;
        q=p;
        p=r       //恢复当前待处理结点
      }
    t=q;
   }
```

第 6 章　树和二叉树

一、基础知识题

6.1 设树 T 的度为 4，其中度为 1，2，3 和 4 的结点个数分别为 4，2，1，1，求树 T 中的叶子数。

【解答】 设度为 m 的树中度为 0,1,2,…,m 的结点数分别为 $n_0, n_1, n_2,\ldots, n_m$，结点总数为 n，分枝数为 B，则下面二式成立

$$n= n_0+n_1+n_2+\ldots+n_m \quad (1)$$

$$n=B+1= n_1+2n_2 +\ldots+mn_m+1 \quad (2)$$

由(1)和(2)得叶子结点数 $n_0=1+\sum_{i=1}^{m}(i-1)n_i$

即： $n_0=1+(1-1)*4+(2-1)*2+(3-1)*1+(4-1)*1=8$

6.2 一棵完全二叉树上有 1001 个结点，求叶子结点的个数。

【解答】因为在任意二叉树中度为 2 的结点数 n_2 和叶子结点数 n_0 有如下关系：$n_2=n_0-1$,所以设二叉树的结点数为 n, 度为 1 的结点数为 n_1，则

$$n= n_0+ n_1+ n_2$$

$$n=2n_0+n_1-1$$

$$1002=2n_0+n_1$$

由于在完全二叉树中，度为 1 的结点数 n_1 至多为 1，叶子数 n_0 是整数。本题中度为 1 的结点数 n_1 只能是 0，故叶子结点的个数 n_0 为 501.

6.3 一棵 124 个叶结点的完全二叉树，最多有多少个结点。

【解答】由公式 $n=2n_0+n_1-1$,当 n_1 为 1 时，结点数达到最多 248 个。

6.4. 一棵完全二叉树有 500 个结点，请问该完全二叉树有多少个叶子结点？有多少个度为 1 的结点？有多少个度为 2 的结点？如果完全二叉树有 501 个结点，结果如何？请写出推导过程。

【解答】由公式 $n=2n_0+n_1-1$，带入具体数得，$500=2n_0+n_1-1$，叶子数是整数，度为 1 的结点数只能为 1，故叶子数为 250，度为 2 的结点数是 249。

若完全二叉树有 501 个结点，则叶子数 251，度为 2 的结点数是 250，度为 1 的结点数为 0。

6.5 某二叉树有 20 个叶子结点，有 30 个结点仅有一个孩子，则该二叉树的总结点数是多少。

【解答】由公式 $n=2n_0+n_1-1$，得该二叉树的总结点数是 69。

6.6 求一棵具有 1025 个结点的二叉树的高 h。

【解答】该二叉树最高为 1025（单支树），最低高为 11。因为 $2^{10}-1<1025<2^{11}-1$，故 1025 个结点的完全二叉树高 11。

6.7 一棵二叉树高度为 h，所有结点的度或为 0，或为 2，则这棵二叉树最少有多少结点。

【解答】第一层只一个根结点，其余各层都两个结点，这棵二叉树最少结点数是 2h-1。

6.8 将有关二叉树的概念推广到三叉树，则一棵有 244 个结点的完全三叉树的高度是多少。

【解答】设含 n 个结点的完全三叉树的高度为 h，则有

$$1+3+\cdots+3^{h-2}<n\leqslant 1+3+\cdots+3^{h-1}$$

$$\therefore (3^{h-1}-1)/2<n\leqslant (3^{h}-1)/2$$

$$3^{h-1}<2n<3^{h}$$

$$\therefore h=\lfloor \log_3 2n \rfloor+1$$

本题 n=244，故 h=6。

6.9 对二叉树的结点从 1 开始进行连续编号，要求每个结点的编号大于其左、右

孩子的编号，同一结点的左、右孩子中，其左孩子的编号小于其右孩子的编号，是采用何种次序的遍历实现编号的。

【解答】后序遍历二叉树，因为后序遍历顺序为左子树-右子树-根结点。

6.10 高度为 h(h>0)的满二叉树对应的森林由多少棵树构成。

【解答】因为在二叉树转换为森林时，二叉树的根结点，根结点的右子女，右子女的右子女，……，都是树的根，所以，高度为 h(h>0)的满二叉树对应的森林由 h 棵树构成。

6.11 某二叉树结点的中序序列为 BDAECF，后序序列为 DBEFCA，则该二叉树对应的森林包括几棵树？

【解答】3 棵树。（本题不需画出完整的二叉树，更不需要画出森林，只需画出二叉树的右子树就可求解。）

6.12 对任意一棵树，设它有 n 个结点，这 n 个结点的度数之和为多少？

【解答】n-1。度数其实就是分支个数。根结点无分支所指，其余结点有且只有一个分支所指。

6.13 一棵左子树为空的二叉树在先序线索化后，其中空的链域的个数是多少？

【解答】对二叉树线索化时，只有空链域才可加线索。一棵左子树为空的二叉树在先序线索化时，根结点的左链为空，应加上指向前驱的线索，但根结点无前驱，故该链域为空。同样分析知道最后遍历的结点的右链域为空。故一棵左子树为空的二叉树在先序线索化后，其中空的链域的个数是 2 个。

6.14 一棵左、右子树均不空的二叉树在先序线索化后，其中空的链域的个数是多少?

【解答】1 个。

6.15 设 B 是由森林 F 变换得的二叉树。若 F 中有 n 个非终端结点，则 B 中右指针域为空的结点有几个？

【解答】森林中任何一个非终端结点在转换成二叉树时，其第一个子女结点成为该非终端结点的左子女，其余子女结点成为刚生成的左子女结点的右子女，右子女结点的右子女，……，最右子女结点的右链域为空。照此分析，n 个非终端结点在转换后，其子女结点中共有 n 个空链域。另外，森林中各棵树的根结点可以看做互为兄弟，转换成二叉树后也产生 1 个空链域。因此，本题的答案是 n+1。

6.16 试分别找出满足以下条件的所有二叉树：

(1) 二叉树的前序序列与中序序列相同；

(2) 二叉树的中序序列与后序序列相同；

(3) 二叉树的前序序列与后序序列相同；

(4) 二叉树的前序序列与层次序列相同；

(5) 二叉树的前序、中序与后序序列均相同。

【解答】前序遍历二叉树的顺序是“根—左子树—右子树”，中序遍历的顺序是“左子树—根—右子树”，后序遍历顺序是：“左子树—右子树－根＂，根据以上原则，本题解答如下：

若前序序列与中序序列相同，则或为空树，或为任一结点至多只有右子树的二叉树。

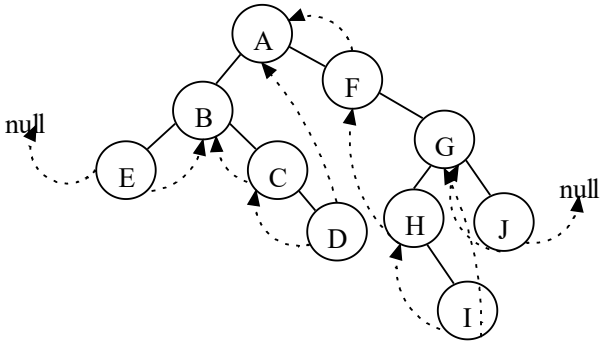
若中序序列与后序序列相同，则或为空树，或为任一结点至多只有左子树的二叉树。

若前序序列与后序序列相同，则或为空树，或为只有根结点的二叉树。

若二叉树的前序、中序与后序序列均相同，则或为空树，或为只有根结点的二叉树。

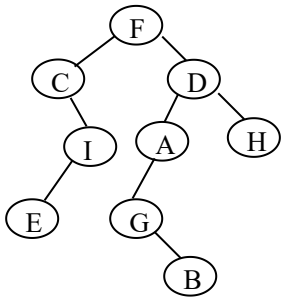
6.17 已知一棵二叉树的前序遍历的结果是 ABECDFGHIJ，中序遍历的结果是 EBCDAFHIGJ，试画出这棵二叉树，对二叉树进行中序线索化，并将该二叉树转换为森林。

【解答】



6.18 已知一棵二叉树的后序遍历序列为 EICBGAHDF，同时知道该二叉树的中序遍历序列为 CEIFGBADH，试画出该二叉树。

【解答】



6.19 设二叉树中每个结点均用一个字母表示，若一个结点的左子树或右子树为

空，用#表示，现前序遍历二叉树，访问的结点序列为ABD##C#E##F##，写出中序和后序遍历二叉树时结点的访问序列。

【解答】中序遍历二叉树时结点的访问序列：#D#B#C#E#A#F#

后序遍历二叉树时结点的访问序列。##D###ECB##FA

6.20 有 n 个结点的 k 叉树（k≥2）用 k 叉链表表示时，有多少个空指针？

【解答】k 叉树（k≥2）用 k 叉链表表示时,每个结点有 k 个指针，除根结点没有指针指向外，其余每个结点都有一个指针指向，故空指针的个数为：

nk-(n-1)=n(k-1)+1

6.21 一棵高度为 h 的满 k 叉树有如下性质：根结点所在层次为 0；第 h 层上的结点都是叶子结点；其余各层上每个结点都有 k 棵非空子树，如果按层次自顶向下，同一层自左向右，顺序从 1 开始对全部结点进行编号，试问：

(1)各层的结点个数是多少?

(2)编号为 i 的结点的双亲结点（若存在）的编号是多少?

(3)编号为 i 的结点的第 m 个孩子结点（若存在）的编号是多少？

(4)编号为 i 的结点有右兄弟的条件是什么？其右兄弟结点的编号是多少？

【解答】

(1)k^l(l 为层数，按题意，根结点为 0 层)

（2）因为该树每层上均有 k^l 个结点，从根开始编号为 1，则结点 i 的从右向左数第 2 个孩子的结点编号为 ki。设 n 为结点 i 的子女，则关系式(i-1)k+2<=n<=ik+1 成立，因 i 是整数，故结点 i 的双亲的编号为⌊(i-2)/k⌋+1。

(3) 结点 i(i>1)的前一结点编号为 i-1（其最右边子女编号是(i-1)*k+1），故结点 i 的第 m 个孩子的编号是(i-1)*k+1+m。

(4) 根据以上分析，结点 i 有右兄弟的条件是，它不是双亲的从右数的第一子女，即 (i-1)%k!=0，其右兄弟编号是 i+1。

6.22. 证明任一结点个数为 n(n>0) 的二叉树的高度至少为⌊(logn)⌋+1。

【解答】最低高度二叉树的特点是，除最下层结点个数不满外，其余各层的结点数都应达到各层的最大值。设 n 个结点的二叉树的最低高度是 h，则 n 应满足 2^{h-1}≦n≦2^h-1 关系式。解此不等式，并考虑 h 是整数，则有 h=⌊logn⌋+1，即任一结点个数为 n 的二叉树的高度至少为 O(logn)。

6.23 已知 A[1..N]是一棵顺序存储的完全二叉树，如何求出 A[i]和 A[j]的最近的共同祖先？

【解答】根据顺序存储的完全二叉树的性质，编号为 i 的结点的双亲的编号是⌊i/2⌋，故 A[i]和 A[j]的最近公共祖先可如下求出：

```
while(i/2!=j/2)
```

```
if(i>j) i=i/2;
else  j=j/2;
```

退出 while 后，若 i/2=0,则最近公共祖先为根结点，否则最近公共祖先是 i/2。

6.24 已知一棵满二叉树的结点个数为 20 到 40 之间的素数，此二叉树的叶子结点有多少个？

【解答】结点个数在 20 到 40 的满二叉树且结点数是素数的数是 31，其叶子数是 16。

6.25 求含有 n 个结点、采用顺序存储结构的完全二叉树中的序号最小的叶子结点的下标。要求写出简要步骤。

【解答】根据完全二叉树的性质，最后一个结点（编号为 n）的双亲结点的编号是⌊n/2⌋，这是最后一个分枝结点，在它之后是第一个终端（叶子）结点，故序号最小的叶子结点的下标是⌊n/2⌋+1。

6.26 试证明:同一棵二叉树的所有叶子结点，在前序序列、中序序列以及后序序列中都按相同的相对位置出现（即先后顺序相同），例如前序 abc，后序 bca，中序 bac。

【证明】前序遍历是“根－左－右”，中序遍历是“左－根－右”，后序遍历是“左－右－根”。三种遍历中只是访问“根”结点的时机不同，对左右子树均是按左右顺序来遍历的，因此所有叶子都按相同的相对位置出现。

6.27 设具有四个结点的二叉树的前序遍历序列为 a b c d； S 为长度等于 4 的由 a，b，c，d 排列构成的字符序列，若任取 S 作为上述算法的中序遍历序列，试问是否一定能构造出相应的二叉树，为什么？试列出具有四个结点二叉树的全部形态及相应的中序遍历序列。

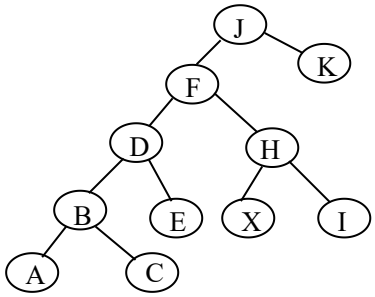
【解答】若前序序列是 abcd，并非由这四个字母的任意组合（4!=24）都能构造出二叉树。因为以 abcd 为输入序列,通过栈只能得到 1/(n+1)*2n!/(n!*n!)=14 种，即以 abcd 为前序序列的二叉树的数目是 14。任取以 abcd 作为中序遍历序列，并不全能与前序的 abcd 序列构成二叉树。例如：若取中序序列 dcab 就不能。

该 14 棵二叉树的形态及中序序列略。

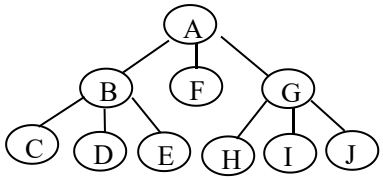
6.28 已知某二叉树的每个结点，要么其左、右子树皆为空，要么其左、右子树皆不空。又知该二叉树的前序序列为：JFDBACEHXIK；后序序列为：ACBEDXIHFKJ。请给出该二叉树的中序序列，并画出相应的二叉树树形。

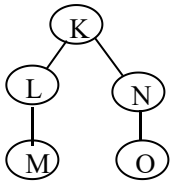
【解答】一般说来，仅仅知道二叉树的前序遍历序列和后序遍历序列并不能确定这棵二叉树，因为并不知道左子树和右子树两部分各有多少个结点。但本题有特殊性，即每个结点“要么其左、右子树皆为空，要么其左、右子树皆不空”。具体说，前序序列的第一个结点是二叉树的根，若该结点后再无其它结点，则二叉树只有根结点；否则，该结点必有左右子树，且根结点后的第一个结点就是“左子树的根”。到后序序列中查找这个“左子树的根”，它将后序序列分成左右两部分：左部分(包括所查到的“左子树的根结点”)是二叉树的左子树(可能为空)，右部分(除去最后的根结点)则是右子树(可能为空)。这样，在确定根结点后，就可以将后序遍历序列（从而也将前序遍历序列）分成左子树和右子树两部分了。

本题中，先看前序遍历序列，第一个结点是 J，所以 J 是二叉树的根，J 后面还有结点，说明 J 有左、右子树，J 后面的 F 必是左子树的根。到后序遍历序列中找到 F,F 将后序遍历序列分成两部分：左面 ACBEDXIH，说明 FACBEDXIH 是根 J 的左子树；右面 K(K 的右面 J 已知是根)，说明 K 是根 J 的右子树。这样，问题就转化为“以前序序列 FDBACEHXI 和后序序列 ACBEDXIHF 去构造根 J 的左子树”，以及“以前序序列 K 和后序序列 K 去构造根 J 的右子树”了。如此构造下去，所构造的二叉树如下。易见，中序序列为 ABCDEFXHIJK。

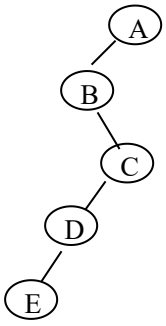


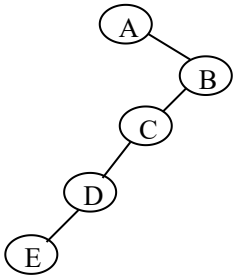
6.29 已知一个森林的先序序列和后序序列如下，请构造出该森林。
先序序列：ABCDEFGHIJKLMNO
后序序列：CDEBFHIJGAMLONK
【解答】森林的先序序列和后序序列对应其转换的二叉树的先序序列和中序序列，应先据此构造二叉树，再构造出森林。





6.30 画出同时满足下列两条件的两棵不同的二叉树。
(1)按先根序遍历二叉树顺序为 ABCDE。
(2)高度为 5 其对应的树（森林）的高度最大为 4。
【解答】





6.31 用一维数组存放的一棵完全二叉树；ABCDEFGHIJKL。请写出后序遍历该二叉树的结点访问序列。
【解答】后序遍历该二叉树的结点访问序列为：DECGHFBKJLIA

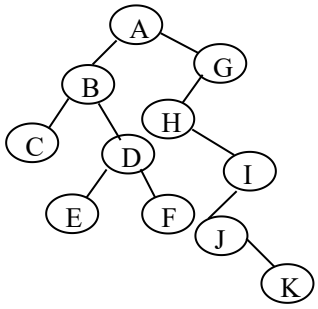
6.32 一棵二叉树的先序、中序和后序序列如下，其中有部分未标出，试构造出该二叉树。

先序序列为：_ _ C D E _ G H I _ K

中序序列为：C B _ _ F A _ J K I G

后序序列为：_ E F D B _ J I H _ A

【解答】



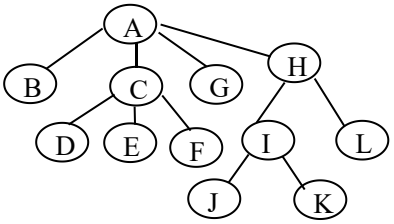
6.33 设树形 T 在后根次序下的结点排列和各结点相应的度数如下：

后根次序： B D E F C G J K I L H A

次　　数： 0 0 0 0 3 0 0 0 2 0 2 4

请画出 T 的树形结构图。

【解答】在树在后根遍历次序下，根结点在最后，任何结点的子树的所有结点都直接排在该结点之前。例如，挨着根结点的是根结点的最右边的子女。每棵子树的所有结点都聚集在一起，中间不会插入其它结点，也不会丢掉任何结点。按照这种理论解答本题，在遍历次序中从右到左分析，A 是根，它有 4 个子女，H 是它的最右边的子女（第 4 子女）。H 有 2 个子女，L 是 H 的最右边的子女，L 无子女，故 I 是 H 的第 1 子女。I 又有 2 个子女：K 和 J，二者均无子女。由此推断出下一个结点 G 是根结点 A 的第 3 子女。……，继续构造，直至最左面的结点 B，结果树形如下：



6.34 若森林共有 n 个结点和 b 条边(b<n)，则该森林中有多少棵树。

【解答】n-b

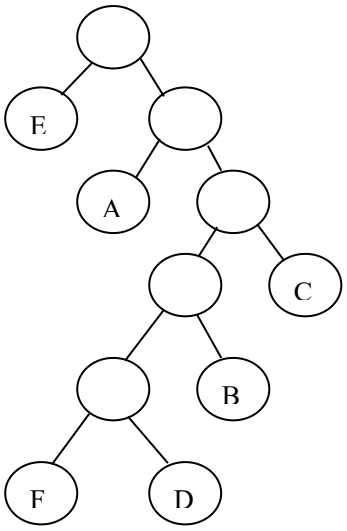
森林的 n 个结点开始可看作是 n 个连通分量，加入一条边将减少一个连通分量。因为树可以定义为无环的图，故加入 b 条边将减少 b 个连通分量，因而 n 个结点 b 条边的森林有 n-b 棵树。

6.35 求高度为 k 的完全二叉树至少有多少个叶结点？

【解答】当高度为 k 的完全二叉树的第 k 层只有一个结点时，结点数达到最少，这也是高度为 k 的完全二叉树具有的最少叶结点数，我们知道，第 k-1 层有 $2^{k-2}-1$ 个叶结点，第 k 层只有一个叶结点，但这个结点的双亲已不再是叶子结点，故高度为 k 的完全二叉树至少有 $2^{k-2}-1$ 个叶结点。

6.36 某通信电文由A、B、C、D、E、F六个字符组成，它们在电文中出现的次数分别是 16，5，9，3，20，1。试画出其哈夫曼树并确定其对应的哈夫曼编码。

【解答】



对应的哈夫曼编码：A—10，B—1101， C—111， D—11001， E—0，F—11000

二、算法设计题

6.37 以二叉链表作为存储结构，设计算法求出二叉树 T 中度为 0、度为 1 和度为 2 的结点数。

【题目分析】结点计数可以在遍历中解决。根据“访问根结点”在“递归遍历左子树”和“递归遍历右子树”中位置的不同，而有前序、后序和中序遍历。

【算法 6.37】

```
int n2,n1,n0;    //设置三个全局变量，分别记度为2，1和叶子结点的个数
void Count(BiTree t)
 {if(t)
   {if(t->lchild && t->rchild) n2++;
    else if(t->lchild && !t->rchild || !t->lchild && t->rchild) n1++;
         else  n0++;
    if(t->lchild!=null) Count(t->lchild);
    if(t->rchild!=null) Count(t->rchild);
 }//Count
```

6.38 一棵 n 个结点的完全二叉树存放在二叉树的顺序存储结构中，试编写非递归算法对该树进行先序遍历。

【题目分析】二叉树的顺序存储是按完全二叉树的顺序存储格式，双亲与子女结点下标间有确定关系。顺序存储结构的二叉树用结点下标大于 n（完全二叉树）或 0（对一般二叉树的“虚结点”）判空。本题是完全二叉树。

【算法 6.38】

```
void PreOrder(ElemType bt[],int n)
  //对以顺序结构存储的完全二叉树 bt 进行前序遍历
 {int i=1,top=0,s[];  //top 是栈 s 的栈顶指针，栈容量足够大
  while(i<=n || top>0)
    {while(i<=n)
       {printf(bt[i]);                    //访问根结点；
        if(2*i+1<=n) s[++top]=2*i+1;      //右子女的下标位置进栈
        i=2*i;                            //沿左子女向下
        }
     if(top>0) i=s[top--];                //取出栈顶元素
     }//while
 }//结束 PreOrder
```

6.39 以二叉链表作为存储结构的二叉树，按后序遍历时输出的结点顺序为 a_1，a_2，…，a_n。试编写一算法，要求输出后序序列的逆序 a_n，a_{n-1} …，a_2 ，a_1 。

【题目分析】二叉树后序遍历是按“左子树－右子树－根结点”的顺序遍历二叉树，根据题意，若将遍历顺序改为“根结点－右子树－左子树”，就可以实现题目要求。

【算法 6.39】

```
void PostOrder(BiTree bt)
//对二叉树 bt 进行先右后左的“先根”遍历
{if(bt)
     {printf(bt->data);        //访问根结点
      PostOrder(bt->rchild);  //先根遍历右子树
      PostOrder(bt->lchild);  //先根遍历左子树
      }
}
```

6.40 以二叉链表作为存储结构，设计算法交换二叉树中所有结点的左、右子树。

【算法 6.40】

```
void exchange(BiTree bt)
  //将二叉树 bt 所有结点的左右子树交换
  {if(bt){BiTree s;
          s=bt->lchild; bt->lchild=bt->rchild; bt->rchild=s; //左右子女交换
          exchange(bt->lchild);  //交换左子树上所有结点的左右子树
          exchange(bt->rchild);  //交换右子树上所有结点的左右子树
   }//if  }//结束
```

【算法讨论】将上述算法中两个递归调用语句放在前面，将交换语句放在最后，则是以后序遍历方式交换所有结点的左右子树。中序遍历方式不适合本题。

6.41 以二叉链表为存储结构，写出在二叉树中求值为 x 的结点在树中层次数的算法。

[题目分析] 按层次遍历，设一队列 Q，用 front 和 rear 分别指向队头和队尾元素，last 指向各层最右结点位置。

【算法 6.41】

```
int Level_x(BiTree bt)//求值为 x 的结点在树中层次数
 {if(bt!=null)
   {int front=0,last=1,rear=1,level=1;//level 记层次数
    BiTree Q[];Q[1]=bt;//根结点入队
    while(front<=last)
     {bt=Q[++front];
      if(bt->data==x)
        {printf(“%3d\n”,level); return level;}//值为 x 的结点在树中层次数
      if(bt->lchild!=null) Q[++rear]=bt->lchild;//左子女入队列
      if(bt->rchild!=null) Q[++rear]=bt->rchild;//右子女入队列
      if(front==last) {last=rear; level++; }//本层最后一个结点已处理完
     }
   }//算法结束
```

6.42 已知深度为 h 的二叉树以一维数组 $BT(1..2^h-1)$ 作为存储结构。试编写算法求

该二叉树中叶子的个数。

【题目分析】按完全二叉树形式顺序存储二叉树时，无元素的位置要当作“虚结点”。设虚结点取二叉树结点以外的值（这里设为 0）。设结点序号为 i,则当 $i<=(2^h-1)/2$ 时，若其 2i 和 2i+1 位置为虚结点，则 i 为叶子结点；当 $i>(2^h-1)/2$ 时，若 i 位置不是虚结点，则必为叶子结点。

【算法 6.42】

```
int Leaves(int BT[],int n)
   //计算深度为 h 以一维数组 BT 作为存储结构的二叉树的叶子结点数，n 为数组长度
  {int num=0;    //记叶子结点数
   for(i=0;i<n;i++)
     if(BT[i]!=0)
       {if(i<=n/2)
          {if(BT[2*i]==0 && 2*i+1<=n && BT[2*i+1]==0) num++;}
             //若结点无孩子，则是叶子
        else if(BT[i]!=0) num++;     //存储在数组后一半的元素是叶子结点
        }
    return num;
   }//结束 Leaves
```

6.43 已知二叉树以一维数组作为存储结构。试编写算法求下标为 i 和 j 的两个结点的最近共同祖先结点的值。

【题目分析】二叉树顺序存储，是按完全二叉树的格式存储，利用完全二叉树双亲结点与子女结点编号间的关系，求下标为 i 和 j 的两结点的双亲，双亲的双亲，等等，直至找到最近的公共祖先。

【算法 6.43】

```
void Ancestor(ElemType bt[],int n,i,j,)
//求顺序存储在 bt[1..n]的二叉树中下标为 i 和 j 的两个结点的最近公共祖先结点
 {if(i<1 || j<1) {printf("参数错误\n");exit(0);};
  if(i==j)
     {if(i==1) {printf("所查结点为根结点，无祖先\n");exit(0);};
      else {printf ("结点的最近公共祖先是%d，值是%d",i/2,A[i/2]);exit(0)}
     }
  while(i!=j)
     if(i>j) i=i/2; //下标为 i 的结点的双亲结点的下标
     else j=j/2;     //下标为 j 的结点的双亲结点的下标
   printf("所查结点的最近公共祖先的下标是%d，值是%d",i,A[i]);
     //设元素类型整型。
 }// Ancestor
```

6.44 已知一棵完全二叉树顺序存储于向量 s[1..n]中，试编写算法由此顺序存储结

构建立该二叉树的二叉链表。

【算法 6.44】

```
BiTree Creat(ElemType A[],int i)
 //n 个结点的完全二叉树存于一维数组 A 中，本算法建立二叉链表表示的完全二叉树
{BiTree tree;
 if(i<=n)
  {tree=(BiTree)malloc(sizeof(BiNode));
   tree->data=A[i];
   if(2*i>n) tree->lchild=null;
      else tree->lchild=Creat(A,2*i);
   if(2*i+1>n) tree->rchild=null;
      else tree->rchild=Creat(A,2*i+1);
   }
  return (tree);
 }//Creat
```

【算法讨论】初始调用时,i=1。

6.45 编写算法判别给定二叉树是否为完全二叉树。

【题目分析】判定是否是完全二叉树，可以使用队列，在遍历中利用完全二叉树“若某结点无左子女就不应有右子女”的原则进行判断。具体说，在层次遍历时，若碰到一个空指针后，在遍历结束前又碰到结点，则结论为该二叉树不是完全二叉树。

【算法 6.45】

```
int JudgeComplete(BiTree bt)
  //判断二叉树是否是完全二叉树,如是，返回 1，否则，返回 0
 {int tag=0;                    //出现空指针时，置 tag=1
  BiTree p=bt, Q[];             // Q 是队列，元素是二叉树结点指针，容量足够大
  if(p==null) return (1);
  QueueInit(Q); QueueIn(Q,p); //初始化队列，根结点指针入队
  while(!QueueEmpty(Q))
   {p=QueueOut(Q);                                  //出队
    if(p->lchild && !tag) QueueIn(Q,p->lchild);//左子女入队
    else if(p->lchild) return 0;                  //前边已有结点空，本结点不空
         else tag=1;                               //首次出现结点为空
    if(p->rchild && !tag) QueueIn(Q,p->rchild);//右子女入队
    else if(p->rchild) return 0;
         else tag=1;
   } //while
 return 1;  } //JudgeComplete
```

[算法讨论]完全二叉树证明还有其它方法。判断时易犯的错误是证明其左子树和右子树都是完全二叉树，由此推出整棵二叉树必是完全二叉树的错误结论。

6.46 设树以双亲表示法存储，编写计算树的深度的算法。

【题目分析】以双亲表示法作树的存储结构，对每一结点，找其双亲，双亲的双亲，直至（根）结点,就可求出每一结点的层次，取其结点的最大层次就是树的深度。

【算法 6.46】

```
int Depth(Ptree t)
  //求以双亲表示法为存储结构的树的深度
{int maxdepth=0;
 for(i=1;i<=t.n;i++)
   {temp=0; f=i;
    while(f>0)
      {temp++; f=t.nodes[f].parent; }     // 深度加 1,并取新的双亲
    if(temp>maxdepth)  maxdepth=temp;     //最大深度更新
   }
  return(maxdepth);//返回树的深度
  } //结束 Depth
```

6.47 已知在二叉树中，*root 为根结点，*p 和*q 为二叉树中两个结点，试编写求距离它们最近的共同祖先的算法。

【题目分析】后序遍历最后访问根结点，即在递归算法中，根是压在栈底的。采用后序非递归遍历，栈中存放二叉树结点的指针，当访问到某结点时，栈中所有元素均为该结点的祖先。本题要找 p 和 q 的最近共同祖先结点 r ,不失一般性，设 p 在 q 的左边。后序遍历必然先遍历到结点 p，栈中元素均为 p 的祖先。将栈拷入另一辅助栈中。再继续遍历到结点 q 时，将栈中元素从栈顶开始逐个到辅助栈中去匹配，第一个匹配（即相等）的元素就是结点 p 和 q 的最近公共祖先。

【算法 6.47】

先设二叉树的结点结构为：

```
typedef struct
  {BiTree t;
   int tag;     //tag=0 表示结点的左子女已访问，tag=1 为右子女已访问
   }stack;
stack s[],s1[];//栈，容量足够大
BiTree Ancestor(BiTree ROOT,p,q,r)
  //求二叉树上结点 p 和 q 的最近的共同祖先结点 r
 {top=0; bt=ROOT;
  while(bt!=null ||top>0)
    {while(bt!=null && bt!=p && bt!=q)                //结点入栈
       {s[++top].t=bt;
        s[top].tag=0;
        bt=bt->lchild;
        } //沿左分枝向下
     if(bt==p)
       //不失一般性，假定 p 在 q 的左侧,遇结点 p 时，栈中元素均为 p
```

```
的祖先结点
        for(i=1;i<=top;i++) {s1[i]=s[i]; top1=top;}
            //将栈 s 的元素转入辅助栈 s1 保存, top1 记住栈顶
      if(bt==q)                 //找到 q 结点
        for(i=top;i>0;i--)   //将栈中元素的树结点到 s1 去匹配
         {pp=s[i].t;
          for(j=top1;j>0;j--)
            if(s1[j].t==pp){printf("共同的祖先已找到\n");
        return (pp);}
         }
       while(top!=0 && s[top].tag==1) top--;          //退栈
       if(top!=0) {s[top].tag=1;bt=s[top].t->rchild;} //沿右分枝向
下遍历
     }//结束 while(bt!=null ||top>0)
    return(null);// q、p 无公共祖先
   } //结束 Ancestor
```

6.48 以二叉链表作为存储结构，设计按层次遍历二叉树的算法。

【算法 6.48】

```
void Level(BiTree bt) //层次遍历二叉树
    {if(bt)
       {QueueInit(Q); //Q 是以二叉树结点指针为元素的队列
        QueueIn(Q,bt);
        while(!QueueEmpty(Q))
         {p=QueueOut(Q);                          //出队
          printf(p->data);                        //访问结点
          if(p->lchild) QueueIn(Q,p->lchild); //非空左子女入队
          if(p->rchild) QueueIn(Q,p->rchild); //非空右子女入队
           }
        }//if(bt)
       }
```

6.49 编写算法查找二叉链表中数据域值为 x 的结点(假定各结点的数据域值各不相同)，并打印出 x 所有祖先的数据域值。

【题目分析】后序遍历最后访问根结点，当访问到值为 x 的结点时，栈中所有元素均为该结点的祖先。

【算法 6.49】

```
void Search(BiTree bt,ElemType x)
 //在二叉树 bt 中，查找值为 x 的结点，并打印其所有祖先
  {typedef struct
     {BiTree t;
      int tag;    //tag=0 表示左子女被访问，tag=1 右子女被访问
      }stack;
   stack s[];  //栈容量足够大
```

```
        top=0;
        while(bt!=null||top>0)
        {while(bt!=null && bt->data!=x)                    //结点入栈
           {s[++top].t=bt; s[top].tag=0; bt=bt->lchild;} //沿左分枝向
下
         if(bt->data==x)
            {printf("所查结点的所有祖先结点的值为:\n");//找到 x
            for(i=1;i<=top;i++)
               printf(s[i].t->data);  return;
            } //输出祖先值后结束
         while(top!=0 && s[top].tag==1) top--;             //退栈（空遍
历）
         if(top!=0)
             {s[top].tag=1;bt=s[top].t->rchild;}           //沿右分枝向
下遍历
        }// while(bt!=null||top>0) }结束 search
```

6.50 设计这样的二叉树，用它可以表示父子、夫妻和兄弟三种关系，并编写一个查找任意父亲结点的所有儿子结点的过程。

【题目分析】用二叉树表示出父子，夫妻和兄弟三种关系，可以用根结点表示父（祖先），根结点的左子女表示妻，妻的右子女表示子。这种二叉树可以看成类似树的孩子兄弟链表表示法；根结点是父，根无右子女，左子女表示妻，妻的右子女（右子女的右子女等）均可看成兄弟（即父的所有儿子），兄弟结点又成为新的父，其左子女是兄弟（父的儿子）妻，妻的右子女（右子女的右子女等）又为儿子的儿子等等。首先递归查找某父亲结点，若查找成功，则其左子女是妻，妻的右子女及右子女的右子女等均为父亲的儿子。

【算法 6.50】

```
BiTree Search(BiTree t,ElemType father)
 //在二叉树上查找值为 father 的结点
{int tag=0;
 if(t==null) p=null;                    //二叉树上无 father 结点
 else if(t->data==father)
         {tag=1; p=t;}//查找成功
       else{p=Search(t->lchild,father);
 if(tag==0)p=Search(t->rchild,father);}
 return p;
}//结束 Search
void PrintSons(BiTree t,ElemType x)   //在二叉树上查找结点值为 x 的
所有的儿子
 {p=Serach(t,x);                       //在二叉树 t 上查找父结点 x
  if(p && p->lchild)                   //存在父结点,且有妻
    {q=p->lchild; q=q->rchild;         //先指向其妻结点，再找到第一
个儿子
     while(q!=null)
```

```
        {printf(q->data); q=q->rchild;}//输出父的所有儿子
      }
    }//结束 PrintSons
```

6.51 编写递归算法判定两棵二叉树是否相等。

【题目分析】首先判断二叉树的根是否相等，如是，再判断其左、右子树是否相等。

【算法 6.51】

```
int BTEqual(BiTree t, BiTree x)
  {//判定二叉树 t 和二叉树 x 是否相等
   if(!t && !x) return true;
  if(t && x && t->data==x->data && BTEqual(t->lchild,x->lchild) &&
     BTEqual(t->rchild,x->rchild) return ture;
  else return false;
  }
```

6.52 已知一棵高度为 K 具有 n 个结点的二叉树，按顺序方式存储，编写将树中最大序号叶子结点的祖先结点全部打印输出的算法。

【题目分析】二叉树中最大序号的叶子结点,是在顺序存储方式下编号最大的结点

【算法 6.52】

```
void Ancesstor(ElemType bt[])
  //打印最大序号叶子结点的全部祖先
  {c=m;     //m=2^K-1
   while(bt[c]==0) c--;//找最大序号叶子结点,该结点存储时在最后
   f=c/2;                //c 的双亲结点 f
   while(f!=0) //从结点 c 的双亲结点直到根结点，路径上所有结点均为祖先结点
     {printf(bt[f]); f=f/2; }//逆序输出，最老的祖先最后输出
   }//结束
```

6.53 设二叉树以二叉链表作为存储结构，编写算法对二叉树进行非递归的中序遍历。

【算法 6.53】

```
void InOrder(BiTree bt)
  //对二叉树进行非递归中序遍历
 {BiTree s[],p=bt;  //s 是元素为二叉树结点指针的栈，容量足够大
  int top=0;
  while(p || top>0)
    {while(p) {s[++top]=p; bt=p->lchild;}                    //沿左子树向下
     if(top>0)
        {p=s[top--]; printf(p->data); p=p->rchild;} //退栈，访问，转右子树
```

```
        }
    }//结束
```

【算法讨论】若将访问语句 printf(p->data)，放到语句 s[++top]=p 的前面，则是前序遍历的非递归算法。

6.54 设 T 是一棵满二叉树，写一个把 T 的后序遍历序列转换为先序遍历序列的递归算法。

【题目分析】对一般二叉树，仅根据一个先序（或中序、或后序）遍历，不能确定另一个遍历序列。但由于满二叉树“任一结点的左右子树均含有数量相等的结点”，根据此性质，可将任一遍历序列转为另一遍历序列。

【算法 6.54】

```
void PostToPre(ElemType post[],pre[],int l1,h1,l2,h2)
//将满二叉树的后序序列转为先序序列，l1,h1,l2,h2 是序列初始和最后结点的下标。
  {if(h1>=l1)
    {pre[l2]=post[h1];     //根结点
     visit(pre[l2]);       //访问根结点
     half=(h1-l1)/2;       //左或右子树的结点数
     PostToPre(post,pre,l1,l1+half-1,l2+1,l2+half)
        //将左子树后序序列转为先序序列
     PostToPre(post,pre,l1+half,h1-1,l2+half+1,h2)
        //将右子树后序序列转为先序序列
     }
    }// PostToPre t
```

6.55 若二叉树 BT 的每个结点，其左、右子树都为空，或者其左、右子树都不空，这种二叉树有时称为“严格二叉树”。由“严格二叉树”的前序序列和后序序列可以唯一确定该二叉树。写出根据这种二叉树的前序序列和后序序列确定该二叉树的递归算法。

【问题分析】前序序列的第一个结点是二叉树的根，若该结点后再无其它结点，则该结点是叶子；否则，该结点必有左、右子树，且根结点后的第一个结点就是左子树的根。到后序序列中查找这个左子树的根，它将后序序列分成左、右两部分：左部分(包括所查到的结点)是二叉树的左子树(可能为空)，右部分(除去最后的根结点)则是右子树(可能为空)。这样，在确定根结点后，可递归确定左、右子树。

【算法 6.55】

```
void creat(BiTree *BT,char pre[n],char post[n],int l1,int h1,int l2,int h2)
        //由严格二叉树的前序序列 pre(l1:h1)和后序序列 post(l2:h2)建立二叉树
       { BiTree p=*BT;
         if(l1<=h1)
            {p=(BiNode *)malloc(sizeof(BiNode));
             p->data=pre[l1];//前序序列的第一个元素是根
```

```
            if(l1==h1) {p->lchild=p->rchild=null; }//叶子结点
            else //分支结点
               {for(int i=l2;i<=h2;i++)    //到后序序列中查左子树的
根
                   if(post[i]==pre[l1+1] break;
                L=i-l2+1;     //左子树结点数
                creat(&(p->lchild), pre,post,l1+1,l1+L,l2,i);
                 creat(&(p->rchild), pre,post,l1+L+1,h1,i+1,h2-1);
                 }//else
               }//if
              }//结束
```

6.56 编写一个递归算法，利用叶结点中空的右链指针域 rchild，将所有叶结点自左至右链接成一个单链表，算法返回最左叶结点的地址（链头）。

【题目分析】叶子结点只有在遍历中才能知道，这里使用中序递归遍历。设置前驱结点指针 pre，初始为空。第一个叶子结点由指针 head 指向，遍历到叶子结点时，就将它前驱的 rchild 指针指向它，最后叶子结点的 rchild 为空。

【算法 6.56】

```
LinkedList head,pre=null;   //全局变量
LinkedList InOrder(BiTree bt)
  //中序遍历二叉树 bt，将叶子结点从左到右链成一个单链表，表头指针为
head
 {if(bt){InOrder(bt->lchild);                          //中序遍历左子树
         if(bt->lchild==null && bt->rchild==null)//叶子结点
            if(pre==null) {head=bt; pre=bt;}      //处理第一个叶子
结点
            else{pre->rchild=bt; pre=bt; }        //将叶子结点链入
链表
         InOrder(bt->rchild);                          //中序遍历右子树
        }
    pre->rchild=null;                                  //设置链表尾
    return(head);
  } //InOrder
```

6.57 已知有一棵二叉链表表示的二叉树，编写算法，输出从根结点到叶子结点的最长一枝上的所有结点。

【题目分析】后序遍历时栈中保留当前结点的祖先的信息，用一变量保存栈的最高栈顶指针，每当退栈时，栈顶指针大于已保存的最高栈顶指针的值时，则将该栈倒入辅助栈中，辅助栈始终保存最长路径长度上的结点，直至后序遍历完毕，则辅助栈中内容即为所求。

【算法 6.57】

```
void LongestPath(BiTree bt)
   //求二叉树从根结点到叶子结点的最长一枝上的所有结点
 {BiTree p=bt,l[],s[];
```

```
      //l, s 是栈，元素是二叉树结点指针，l 中保留当前最长路径中的结点
  int i, top=0,tag[],longest=0;
  while(p || top>0)
    {while(p){s[++top]=p; tag[top]=0; p=p->lchild;}//沿左分枝向下
     if(tag[top]==1)                                  //当前结点的右分枝已遍历
      {if(!s[top]->lchild && !s[top]->rchild)
              //只有到叶子结点时，才查看路径长度
          if(top>longest)
             {for(i=1;i<=top;i++) l[i]=s[i];
              longest=top;
              top--;
              }
        }//保留当前最长路径到 l 栈，记住最高栈顶指针，退栈
       while(top>0 && tag[top]==1) top--;  //退栈
      if(top>0)
        {tag[top]=1; p=s[top].rchild;}      //沿右子分枝向下
    }//while(p!=null||top>0)
    }//结束 LongestPath
```

6.58 试编写一算法对二叉树进行前序线索化。

【题目分析】线索化是在遍历中完成的，因此，对于二叉树进行前序、中序、后序遍历，在“访问根结点”处进行加线索的改造，就可实现前序，中序和后序的线索化。

【算法 6.58】

```
BiThrTree pre=null;//设置前驱
void PreOrderThreat(BiThrTree BT)
  //对以线索链表为存储结构的二叉树 BT 进行前序线索化
{if(BT!=null)
  {if(BT->lchild==null){BT->ltag=1; BT->lchild=pre;}//设置左线索
   if(pre!=null && pre->rtag==1) pre->rchild=BT;     //设置前驱的右线索;
   if(BT->rchild==null) BT->rtag=1;                  //为建立右链作准备
   pre=BT;                                           //前驱后移
   if(BT->ltag==0) PreOrderThreat(BT->lchild);       //左子树前序线索化
   PreOrderThreat(BT->rchild);                        //右子树前序线索化
}//if(BT!=null) }//结束 PreOrderThreat
```

6.59 试编写一算法对二叉树进行中序线索化。

【算法 6.59】

```
BiThrTree pre==null;
void InOrderThreat(BiThrTree T)//对二叉树进行中序线索化
{if(T)
  {InOrderThreat(T->lchild);                    //左子树中序线
  索化
   if(T->lchild==null){T->ltag=1; T->lchild=pre;} //左线索为 pre
   if(pre!=null && pre->rtag==1) pre->rchild=T;}  //给前驱加后继线
索
   if(T->rchild==null) T->rtag=1;                //置右标记，为右
线索作准备
   pre=BT;                                       //前驱指针后移
   InOrderThreat(T->rchild);                     //右子树中序线
索化
   }//if
  }//结束 InOrderThreat
```

6.60 设 p 指向前序线索二叉树 t 的某结点，编写算法求*p 结点的后继结点。

【题目分析】在前序线索二叉树中，求*p 结点的后继结点，若*p 结点有左子女，则左子女是其后继结点；若*p 结点无左子女而有右子女，则右子女是其后继；若*p 结点无左、右子女，线索 p->rchild 指向其后继。

【算法 6.60**】**

```
BiThrTree PreorderNext(BiThrTree  p)
{if (p->ltag==0)        // 结点有左子女
    return(p->lchild); //结点的左子女为其前序后继
  else
     return(p->rchild);//p->rchild 为其前序后继
} // PreorderNext
```

6.61 设 p 指向后序线索二叉树 t 的某结点，编写算法求*p 结点的前驱结点。

【题目分析】在后序线索二叉树中，求*p 结点的前驱结点，若*p 结点有右子女，则右子女是其前驱结点；若*p 结点无右子女而有左子女，则左子女是其前驱；若*p 结点既无右子女又无左子女，线索 p->lchild 指向其前驱。

【算法 6.61**】**

```
BiThrTree PostorderPre(BiThrTree  p)
{ if (p->rtag==0)         // 结点有右子女
    return(p->rchild);   //结点的右子女为其后序前驱
  else
     return(p->lchild) ; //p->lchild 为其后序前驱
} // PreorderPre
```

6.62 设计算法求中序线索二叉树中指针 P 所指结点的前驱结点的指针。

【题目分析】中序线索二叉树中，指针 P 所指结点的前驱结点的特征是：若 p->ltag=1,p->lchild 指向其前驱，否则，P 的左子树上按中序遍历的最后结点是其中序前驱。

【算法 6.62】

```
BiThrTree InPre(BiThrTree T, BiThrTree p)
 //在中序线索树 T 中，查找给定结点 p 的中序前驱
{if（p->ltag==1）q=p->lchild;  //若 p 的左标志为 1，用其左指针指向前驱
 else {q=p->lchild;
       while(q->rtag==0)
          q=q->rchild; //p 的前驱为其左子树中最右下的结点
       }
 return (q);
}//结束 InPre
```

6.63 设计算法求中序线索二叉树中指针 P 所指结点的后继结点的指针。

【题目分析】中序线索二叉树中，指针 P 所指结点的后继结点的特征是：若 p->rtag=1,p->rchild 指向其后继，否则，P 的右子树上按中序遍历的第一个结点是其中序后继。

【算法 6.63】

```
BiThrTree InSucc(BiThrTree T, BiThrTree p)
 //在中序线索二叉树 T 中，查找给定结点 p 的中序后继
{if（p->rtag==1）
     q=p->rchild;   //若 p 的右标志为 1，用其右指针指向后继
 else {q=p->rchild;
       while(q->ltag==0)
          q=q->lchild;   //p 的后继为其右子树中最左下的结点
       }
 return (q);
}//结束 InSucc
```

第 7 章　图

一、基础知识题

7.1 设无向图的顶点个数为 n，则该图最多有多少条边？

【解答】n(n-1)/2

7.2 一个 n 个顶点的连通无向图，其边的个数至少为多少？
【解答】n-1

7.3 要连通具有 n 个顶点的有向图，至少需要多少条弧?
【解答】n

7.4 n 个顶点的完全有向图含有弧的数目是多少?
【解答】n(n-1)

7.5 一个有 n 个顶点的无向图，最少有多少个连通分量，最多有多少个连通分量。
【解答】1,　　n

7.6 图的 BFS 生成树的树高要小于等于同图 DFS 生成树的树高，对吗？
【解答】对

7.7 无向图 G=(V,E)，其中：V={a,b,c,d,e,f}，E={(a,b),(a,e),(a,c),(b,e),(c,f),(f,d),(e,d)}，写出对该图从顶点 a 出发进行深度优先遍历可能得到的全部顶点序列。
【解答】abedfc, acfdeb, aebdfc, aedfcb

7.8 在图采用邻接表存储时，求最小生成树的 Prim 算法的时间复杂度是多少？
【解答】O(n+e)

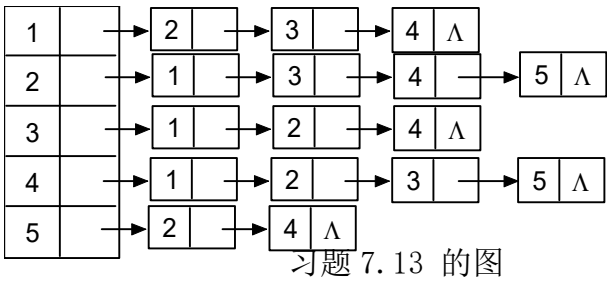
7.9 若一个具有 n 个顶点，e 条边的无向图是一个森林，则该森林中必有多少棵树？
【解答】n-e

7.10 n 个顶点的无向图的邻接矩阵至少有多少非零元素？
【解答】0

7.11 证明:具有 n 个顶点和多于 n-1 条边的无向连通图 G 一定不是树。
【证明】具有 n 个顶点 n-1 条边的无向连通图是自由树，即没有确定根结点的树，每个结点均可当根。若边数多于 n-1 条，因一条边要连接两个结点，则必因加上这一条边而使两个结点多了一条通路，即形成回路。形成回路的连通图不再是树。

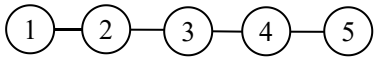
7.12 证明对有向图顶点适当编号，使其邻接矩阵为下三角形且主对角线为全零的充要条件是该图是无环图。
【证明】该有向图顶点编号的规律是让弧尾顶点的编号大于弧头顶点的编号。由于不允许从某顶点发出并回到自身顶点的弧，所以邻接矩阵主对角元素均为 0。先证明该命题的充分条件。由于弧尾顶点的编号均大于弧头顶点的编号，在邻接矩阵中，非零元素（A[i][j]=1）自然是落到下三角矩阵中；命题的必要条件是要使上三角为 0，则不允许出现弧头顶点编号大于弧尾顶点编号的弧，否则，就必然存在环路。（对该类有向无环图顶点编号，应按顶点出度的大小进行顺序编号。）

7.13 设 G=(V，E)以邻接表存储，如图所示，试画出从顶点 1 出发所得到的深度优先和广度优先生成树。

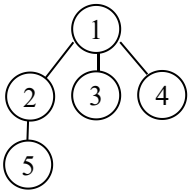


习题 7.13 的图

【解答】深度优先生成树



宽度优先生成树：



7.14 已知一个图的顶点集 V 和边集 E 分别为：

V={0，1，2，3，4，5，6，7}；

E={<0，2>，<1，3>，<1，4>，<2，4>，<2，5>，<3，6>，<3，7>，<4，7>，<4，8>，<5，7>，<6，7>，<7，8>}；

若存储它采用邻接表，并且每个顶点邻接表中的边结点都是按照顶点序号从小到大的次序链接的，则按教材中介绍的进行拓扑排序的算法，写出得到的拓扑序列。

【解答】1-3-6-0-2-5-4-7-8

7.15 一带权无向图的邻接矩阵如下图 ，试画出它的一棵最小生成树。

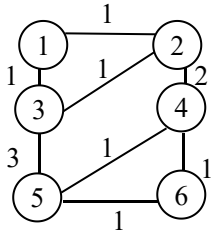
$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 2 & 0 & 0 \\ 1 & 1 & 0 & 0 & 3 & 0 \\ 0 & 2 & 0 & 0 & 1 & 1 \\ 0 & 0 & 3 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

习题 7.15 的图

【解答】设顶点集合为{1，2，3，4，5，6}，

由下边的逻辑图可以看出，在{1，2，3}和{4，5，6}回路中，

各任选两条边，加上（2，4），则可构成 9 棵不同的最小生成树。



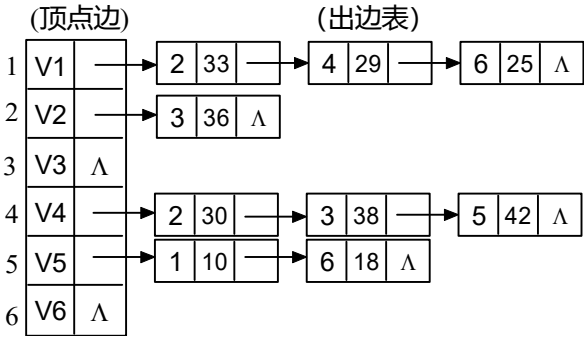
7.16 如图所示是一带权有向图的邻接表法存储表示。其中出边表中的每个结点均含有三个字段，依次为边的另一个顶点在顶点表中的序号、边上的权值和指向下一个边结点的指针。试求：

（1）. 该带权有向图的图形；

（2）. 从顶点 V1 为起点的广度优先遍历的顶点序列及对应的生成树；

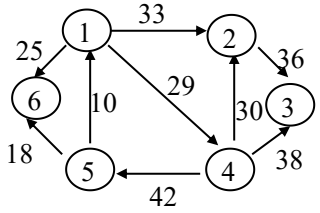
（3）. 以顶点 V1 为起点的深度优先遍历生成树；

（4）. 由顶点 V1 到顶点 V3 的最短路径。

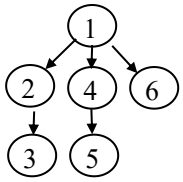


习题 7.16 的图

【解答】(1)



(2)V1,V2,V4,V6,V3,V5



(3)顶点集合 V(G)={V1,V2,V3,V4,V5,V6}

边的集合 E(G)={<V1,V2>,<V2,V3>,<V1,V4>,<V4,V5>,<V5,V6>}

(4)V1 到 V3 最短路径为 67: (V1—V4—V3)

迭代	集合 S	选择顶点	D[ ]
			D[2] D[3] D[4] D[5] D[6]
初	{ v_1}		33 ∞ 29 ∞

值			25			
1	{v1, v6}	v6	33	∞	29	∞
			25			
2	{ v1, v6, v4}	v4	33	67	29	71
3	{ v1, v6, v4, v2}	v2	33	67		71
4	{v1,v6, v4, v2,v3}	v3		67		71
5	{v1,v6,v4, v2,v3,v5}					71

7.17 已知一有向网的邻接矩阵如下，如需在其中一个顶点建立娱乐中心，要求该顶点距其它各顶点的最长往返路程最短，相同条件下总的往返路程越短越好，问娱乐中心应选址何处？给出解题过程。

$$\begin{array}{c} \\ V1 \\ V2 \\ V3 \\ V4 \\ V5 \\ V6 \end{array}\begin{bmatrix} 0 & 2 & \infty & \infty & \infty & 3 \\ \infty & 0 & 3 & 2 & \infty & \infty \\ 4 & \infty & 0 & \infty & 4 & \infty \\ 1 & \infty & \infty & 0 & 1 & \infty \\ \infty & 1 & \infty & \infty & 0 & 3 \\ \infty & \infty & 2 & 5 & \infty & 0 \end{bmatrix}$$

习题 7.17 的图

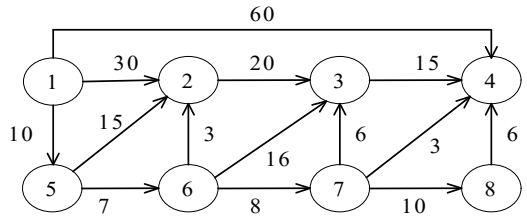
【解答】下面用 FLOYD 算法求出任意两顶点的最短路径（如图 $A^{(6)}$ 所示）。题目要求娱乐中心“距其它各结点的最长往返路程最短”，结点 V1，V3，V5 和 V6 最长往返路径最短都是 9。按着“相同条件下总的往返路径越短越好”，选顶点 V5，总的往返路径是 34。

$$A^{(0)}=\begin{bmatrix} 0 & 2 & \infty & \infty & \infty & 3 \\ \infty & 0 & 3 & 2 & \infty & \infty \\ 4 & \infty & 0 & \infty & 4 & \infty \\ 1 & \infty & \infty & 0 & 1 & \infty \\ \infty & 1 & \infty & \infty & 0 & 3 \\ \infty & \infty & 2 & 5 & \infty & 0 \end{bmatrix} \quad A^{(1)}=\begin{bmatrix} 0 & 2 & \infty & \infty & \infty & 3 \\ \infty & 0 & 3 & 2 & \infty & \infty \\ 4 & 6 & 0 & \infty & 4 & 7 \\ 1 & 3 & \infty & 0 & 1 & 4 \\ \infty & 1 & \infty & \infty & 0 & 3 \\ \infty & \infty & 2 & 5 & \infty & 0 \end{bmatrix}$$

$$A^{(2)}=\begin{bmatrix} 0 & 2 & 5 & 4 & \infty & 3 \\ \infty & 0 & 3 & 2 & \infty & \infty \\ 4 & 6 & 0 & 8 & 4 & 7 \\ 1 & 3 & 6 & 0 & 1 & 4 \\ \infty & 1 & 4 & 3 & 0 & 3 \\ \infty & \infty & 2 & 5 & \infty & 0 \end{bmatrix} \quad A^{(3)}=\begin{bmatrix} 0 & 2 & 5 & 4 & 9 & 3 \\ 7 & 0 & 3 & 2 & 7 & 10 \\ 4 & 6 & 0 & 8 & 4 & 7 \\ 1 & 3 & 6 & 0 & 1 & 4 \\ 8 & 1 & 4 & 3 & 0 & 3 \\ 6 & 8 & 2 & 5 & 6 & 0 \end{bmatrix}$$

$$A^{(4)}=\begin{bmatrix}0&2&5&4&5&3\\3&0&3&2&3&6\\4&6&0&8&4&7\\1&3&6&0&1&4\\4&1&4&3&0&3\\6&8&2&5&6&0\end{bmatrix}\quad A^{(5)}=\begin{bmatrix}0&2&5&4&5&3\\3&0&3&2&3&6\\4&5&0&7&4&7\\1&2&5&0&1&4\\4&1&4&3&0&3\\6&7&2&5&6&0\end{bmatrix}\quad A^{(6)}=\begin{bmatrix}0&2&5&4&5&3\\3&0&3&2&3&6\\4&5&0&7&4&7\\1&2&5&0&1&4\\4&1&4&3&0&3\\6&7&2&5&6&0\end{bmatrix}$$

7.18 求出图中顶点 1 到其余各顶点的最短路径。



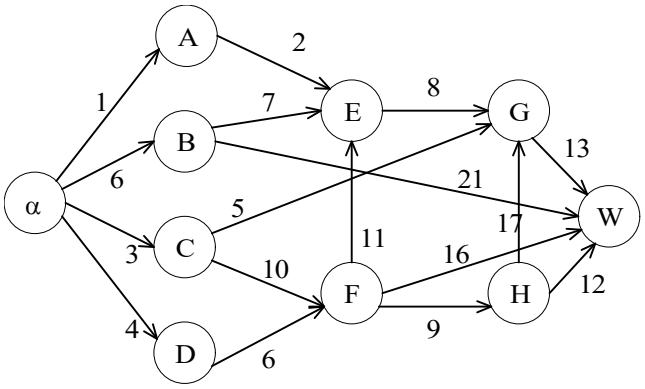
习题 7.18 的图

【解答】本表中 DIST 中各列最下方的数字是顶点 1 到顶点的最短通路。

所选顶点	S（已确定最短路径的顶点集合）	T(尚未确定最短路径的顶点集合)	DIST						
			[2]	[3]	[4]	[5]	[6]	[7]	[8]
初态	{1}	{2,3,4,5,6,7,8}	30	∞	60	10	∞	∞	∞
5	{1,5}	{2,3,4,6,7,8}	25	∞	60	10	17	∞	∞
6	{1,5,6}	{2,3,4,7,8 }	20	33	60		17	25	∞
2	{1,5,6,2}	{3,4,7,8}	20	33	60			25	∞
7	{1,5,6,2,7}	{3,4,8}		31	28			25	35
4	{1,5,6,2,7,4}	{3,8}		31	28				35
3	{1,5,6,2,7,4,3}	{5,8}		31					35
8	{1,5,6,2,7,4,3,8}	{8}							35

顶点 1 到其它顶点的最短路径依次是 20，31，28，10，17，25，35。按 Dijkstra 算法所选顶点依次是 5，6，2，7，4，3，8。

7.19 对图示的 AOE 网络，计算各活动弧的 e(a_i)和 l(a_i)的函数值，各事件（顶点）的 ve（vi）和 vl (vi)的函数值，列出各条关键路径。



【解答】

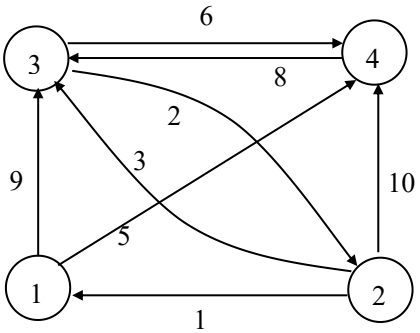
顶点	α	A	B	C	D	E	F	G	H	W
Ve(i)	0	1	6	3	4	24	13	39	22	52
Vl(i)	0	29	24	3	7	31	13	39	22	52

活动	a1	a2	a3	a4	a5	a6	a7	a8	a9	a10	a11	a12	a13	a14	a15	a16	a17
e(i)	0	0	0	0	1	6	6	3	3	4	24	13	13	13	39	22	22
l(i)	28	18	0	3	29	24	31	34	3	7	31	20	36	13	39	22	40

关键路径是：α → C → F → H → G → W ，长 52。

活动与顶点的对照表：a1<α,A> a2<α,B> a3<α,C> a4<α,D> a5<A,E>
a6<B,E> a7<B,W> a8<C,G>
a9<C,F> a10<D,F> a11<E,G> a12<F,E> a13<F,W> a14<F,H> a15<G,W>
a16<H,G> a17<H,W>

7.20　利用弗洛伊德算法，写出如图所示相应的带权邻接矩阵的变化。



【解答】

$$A^0=\begin{bmatrix}0&\infty&9&5\\1&0&3&10\\\infty&2&0&6\\\infty&\infty&8&0\end{bmatrix}\quad A^1=\begin{bmatrix}0&\infty&9&5\\1&0&3&6\\\infty&2&0&6\\\infty&\infty&8&0\end{bmatrix}\quad A^2=\begin{bmatrix}0&\infty&9&5\\1&0&3&6\\3&2&0&6\\\infty&\infty&8&0\end{bmatrix}\quad A^3=\begin{bmatrix}0&11&9&5\\1&0&3&6\\3&2&0&6\\11&10&8&0\end{bmatrix}$$

$$A^4=\begin{bmatrix}0 & 11 & 9 & 5\\ 1 & 0 & 3 & 6\\ 3 & 2 & 0 & 6\\ 11 & 10 & 8 & 0\end{bmatrix}$$

二、算法设计题

7.21 设无向图 G 有 n 个顶点，m 条边。试编写用邻接表存储该图的算法。

【算法 7.21】

```
void CreatGraph (AdjList g)//建立有 n 个顶点和 m 条边的无向图的邻接表存储结构
    {int n,m;
     scanf("%d%d",&n,&m);
     for(i=0,i<n;i++)    //输入顶点信息,建立顶点向量
       {scanf(&g[i].vertex);  g[i].firstarc=null;}
     for(k=0;k<m;k++)    //输入边信息
       {scanf(&v1,&v2);//输入两个顶点
        i=GraphLocateVertex (g,v1); j=GraphLocateVertex (g,v2);//顶点定位
        p=(ArcNode *)malloc(sizeof(ArcNode));//申请边结点
        p->adjvex=j; p->next=g[i].firstarc; g[i].firstarc=p;    //将边结点链入
        p=(ArcNode *)malloc(sizeof(ArcNode));
        p->adjvex=i;    p->next=g[j].firstarc;   g[j].frstarc=p;
       }//for
    }//算法 CreatGraph 结束
```

7.22　知有向图有 n 个顶点，请编写算法，根据用户输入的偶对建立该有向图的邻接表。

【算法 7.22】

```
void CreatAdjList(AdjList g)//建立有向图的邻接表存储结构
   {int n;
    scanf("%d",&n);
    for(i=0;i<n;j++)
      {scanf(&g[i].vertex); g[i].firstarc=null;}//输入顶点信息,下标从 0 开始
     scanf(&v1,.&v2);
    while(v1 && v2)//题目要求两顶点之一为 0 表示结束
       {i=GraphLocateVertex(g,v1);
        p=(ArcNode*)malloc(sizeof(ArcNode));
        p->adjvex=j;   p->next=g[i].firstarc;   g[i].firstarc=p;
        scanf(&v1,&v2);
       }//while
    }
```

7.23　设有向图 G 有 n 个点(用 1，2，…，n 表示)，e 条边，写一算法根据 G 的邻接表生成 G 的反向邻接表，要求算法时间复杂性为 O(n+e)。

【算法 7.24】

```
void InvertAdjList(AdjList gin,gout)
   //将有向图的出度邻接表改为按入度建立的逆邻接表
  {for(i=0;i<n;i++)//设有向图有 n 个顶点，建逆邻接表的顶点向量
     {gin[i].vertex=gout[i].vertex; gin[i].firstarc=null; }
   for(i=0;i<n;i++)        //邻接表转为逆邻接表。
     {p=gout[i].firstarc;//取指向邻接点的指针
      while(p!=null)
         {j=p->adjvex;
          s=(ArcNode *)malloc(sizeof(ArcNode));//申请结点空间
          s->adjvex=i; s->next=gin[j].firstarc; gin[j].firstarc=s;
          p=p->next;//下一个邻接点。
         }//while
      }//for
   }
```

7.24　写出从图的邻接表表示转换成邻接矩阵表示的算法。

【算法 7.25】

```
  void  AdjListToAdjMatrix(AdjList gl, AdjMatrix gm)
     //将图的邻接表表示转换为邻接矩阵表示
  {for(i=0;i<n;i++)    //设图有 n 个顶点，邻接矩阵初始化
     for(j=0;j<n;j++) gm[i][j]=0;
   for(i=0;i<n;i++)    //取第一个邻接点，填邻接矩阵元素值，并求下一个邻接点
     {p=gl[i].firstarc;
      while(p!=null)
        {gm[i][p->adjvex]=1;p=p->next; }
     }//for
  }//算法结束
```

7.25　试写出把图的邻接矩阵表示转换为邻接表表示的算法。

【算法 7.26】

```
  void  AdjMatrixToAdjList(AdjMatrix gm, AdjList gl)
   //将图的邻接矩阵表示转换为邻接表表示
  {for(i=0;i<n;i++)    //邻接表表头向量初始化。
     {scanf(&gl[i].vertex); gl[i].firstarc=null;}
   for(i=0;i<n;i++)
     for(j=0;j<n;j++)
        if(gm[i][j]==1)
           {p=(ArcNode *)malloc(sizeof(ArcNode)) ;//申请结点空间
```

```
            p->adjvex=j;//顶点 I 的邻接点是 j
            p->next=gl[i].firstarc;
            gl[i].firstarc=p; //链入顶点 i 的邻接点链表中
           }//if
  }//end
```

7.26　试编写建立有 n 个顶点，m 条边且以邻接多重表为存储结构表示的无向图的算法。

【算法 7.27】

```
  void CreatMGraph(AdjMulist g)
    //建立有 n 个顶点 m 条边的无向图的邻接多重表的存储结构
    {int n,m;
     scanf("%d%d",&n,&m);
     for(i=0,i<n;i++)     //建立顶点向量
       {scanf(&g[i].vertex);  g[i].firstedge=null;}
     for(k=0;k<m;k++)     //建立边结点
       {scanf(&v1,&v2);
        i=GraphLocateVertex(g,v1);j=GraphLocateVertex(g,v2);
        p=(ENode *)malloc(sizeof(ENode));
        p->ivex=i; p->jvex=j;
        p->ilink=g[i].firstedge; p->jlink=g[j].firstedge;
        g[i].firstedge=p; g[j].firstedge=p;
        }//for
    }//算法结束
```

7.28 已知某有向图（n 个结点）的邻接表，求该图各结点的入度数。

【题目分析】在有向图的邻接表存储结构中求顶点的入度，需要遍历整个邻接表。

【算法 7.28】

```
  void Indegree(AdjList g)//求以邻接表为存储结构的 n 个顶点有向图的各顶点入度
     {for(i=0;i<n;j++)
       {num=0;//入度初始为 0
         for(j=0;j<n;j++) //遍历整个邻接表，求一个顶点的入度
            if(i!=j)
               {p=g[j].firstarc;
                while(p)
                  {if(p->adjvex==i) num++; p=p->next; }
                }
         printf(“顶点%d 的入度为：%d\n”,g[i].vexdata,num); //设顶点数据为整型
       }
     }
```

7.29 已知无向图 G=(V，E)，给出求图 G 的连通分量个数的算法。

【题目分析】使用图的遍历可以求出图的连通分量。进入 dfs 或 bfs 一次，就可以访问到图的一个连通分量的所有顶点。

【算法 7.29】

```
void  dfs (v)
  {visited[v]=1; printf (“%3d”,v); //输出连通分量的顶点。
  p=g[v].firstarc;
  while(p!=null)
    {if（visited[p->adjvex==0]） dfs(p->adjvex);
     p=p->next;
    }//while
  }// dfs
void  Count()
  //求图中连通分量的个数
   {int k=0 ; static AdjList g ;
    for(i=0;i<n;i++ )     //设无向图 g 有 n 个结点
       if(visited[i]==0) { printf ("\n 第%d 个连通分量:\n",++k);
dfs(i);}//if
    }//Count
```

【算法讨论】算法中 visited[]数组是全程变量，每个连通分量的顶点集按遍历顺序输出。这里设顶点信息就是顶点编号，否则应取其 g[i].vertex 分量输出。

7.30 已知无向图采用邻接表存储方式，试写出删除边(i，j)的算法。

【算法 7.30】

```
void DeletEdge(AdjList g,int i,j)
 //在用邻接表方式存储的无向图 g 中，删除边(i,j)
 {p=g[i].firstarc;  pre=null; //删顶点 i 的边结点(i,j),pre 是前驱
 指针
  while(p)
    if(p->adjvex==j)
      {if(pre==null)g[i].firstarc=p->next;
       else pre->next=p->next;
       free(p);    //释放空间
      }
    else {pre=p; p=p->next;}   //沿链表继续查找
  p=g[j].firstarc;  pre=null;  //删顶点 j 的边结点(j,i)
  while(p)
    if(p->adjvex==i)
      {if(pre==null)g[j].firstarc=p->next;
       else pre->next=p->next;
       free(p);   //释放空间
       }
    else {pre=p; p=p->next;}   //沿链表继续查找
 }// DeletEdge
```

【算法讨论】 算法中假定给的 i，j 均存在，否则应检查其合法性。若未给顶点编号，而给出顶点信息，则先用顶点定位函数求出其在邻接表顶点向量中的下标 i 和 j。

7.31 假设有向图以邻接表存储，试编写算法删除弧<V_i, V_j>的算法。

【算法 7.31】

```
void  DeleteArc（AdjList g,vertype vi,vj）
    //删除以邻接表存储的有向图 g 的一条弧<vi,vj>，假定顶点 vi 和 vj 存在
  {i=GraphLocateVertex(g,vi);
   j=GraphLocateVertex(g,vj);    //顶点定位
   p=g[i].firstarc;  pre=null;
   while(p)
     if(p->adjvex==j)
      {if(pre==null) g[i].firstarc=p->next;
       else pre->next=p->next;
       free(p);
      }//释放结点空间
     else  {pre=p; p=p->next;}
  }//结束
```

7.32　设计一个算法利用遍历图的方法判别一个有向图 G 中是否存在从顶点 V_i 到 V_j 的长度为 k 的简单路径，假设有向图采用邻接表存储结构。

【题目分析】本题利用深度优先递归的搜索方法判断有向图 G 的顶点 i 到 j 是否存在长度为 k 的简单路径，先找到 i 的第一个邻接点 m，再从 m 出发递归的求是否存在 m 到 j 的长度为 k-1 的简单路径。

【算法 7.32】

```
int existpathlen(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->next)
         {m=p->adjvex;
          if(!visited[m])
          if(existpathlen(G,m,j,k-1)) return 1; //剩余路径长度减一
         }
       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
```
        }
      return 0; //没找到
    }
```

7.33　设有向图 G 采用邻接矩阵存储，编写算法求出 G 中顶点 i 到顶点 j 的不含回路的、长度为 k 的路径数。

【算法 7.32】

```
int GetPathNum(AdjMatrix GA,int i,int j,int k,int n)
 {//求邻接矩阵方式存储的有向图 G 的顶点 i 到 j 之间长度为 k 的简单路径条数
  //n 为顶点个数
  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)
       {sum=0;                      //sum 表示通过本结点的路径数
        visited[i]=1;
        for(k=0;k<n;k++)
            {if(GA[i][k]!=0 && !visited[k])
             sum+=GetPathNum(GA,k,j,k-1,n) //剩余路径长度减一
            }
        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
        }
      return sum;
    }
```

7.34　设计算法求出以邻接表存储的有向图 G 中由顶点 u 到 v 的所有的简单路径。

【算法 7.34】

```
void AllSPdfs(AdjList g,vertype u,vertype v)
 //求有向图 g 中顶点 u 到顶点 v 的所有简单路径
  { int top=0,s[];
    s[++top]=u; visited[u]=1;
    while(top>0 || p)
        {p=g[s[top]].firstarc;  //第一个邻接点
```

```
            while(p!=null && visited[p->adjvex]==1)
               p=p->next; //下一个访问邻接点表
            if(p==null) top--; //退栈
            else {i=p->adjvex; //取邻接点（编号）
                  if(i==v)      //找到从 u 到 v 的一条简单路径，输出
                      {for(k=1;k<=top;k++)
                       printf( "%3d",s[k]);
                       printf( "%3d\n",v);
                       }//if
                  else { visited[i]=1; s[++top]=i; } //else 深度优先遍历
                 }//else
            }//while
        }// AllSPdfs
```

7.35 以邻接表作存储结构，编写拓扑排序的算法。

【算法 7.35】

7.36 试写一算法，判断以邻接表方式存储的有向图中是否存在由顶点 V_i 到顶点 V_j 的路径（i<>j）。

【题目分析】从 V_i 深度优先遍历,若在未退出深度优先遍历时遍历到 V_j，说明 V_i 间 V_j 存在路径

【算法 7.36】

```
  int visited[n];  //设有向图有 n 个顶点
  int Pathitoj(AdjList g,int Vi,int Vj)
    // 判断以邻接表方式存储的有向图中是否存在由顶点 Vi 到顶点 Vj 的路径
  {if(Vi==Vj) return 1;  //Vi 到顶点 Vj 存在路径
   else{visited[Vi]=1;
        for(p=g[Vi].firstarc;p;p=p->next)
          {k=p->adjvex;
           if(!visited[k] && Pathitoj(g,k, Vj)) return 1;
           }//for
         return 0;  //Vi 到顶点 Vj 不存在路径
        }//else
  }//结束
```

【算法讨论】若顶点 vi 和 vj 不是编号，必须先用顶点定位函数,查出其在邻接表顶点向量中的下标。下面再对本题用非递归算法求解如下。

【算法 7.36.1】

```
int  Connectij (AdjList g , vertype Vi , Vj )
  //判断 n 个顶点以邻接表表示的有向图 g 中，顶点 Vi 各 Vj 是否有路径，有则返回 1，否则返回 0
  {for(i=1;i<n;i++)  visited[i]=0;//访问标记数组初始化
   i=GraphLocateVertex(g, Vi);      //顶点定位，不考虑 Vi 或 Vj 不在图中的情况
```

```
    j=GraphLocateVertex(g, Vj);
    int stack[],top=0;stack[++top]=i;
    while(top>0)
     {k=stack[top--]; p=g[k].firstarc;
      while(p && visited[p->adjvex]==1)
         p=p->next;//查第 k 个链表中第一个未访问的弧结点
      if(p==null) top--;
      else {i=p->adjvex;
             if(i==j) return(1); //顶点 Vi 和 Vj 间有路径
            else {visited[i]=1; stack[++top]=i;}
            }//else
     }while
    return(0); //顶点 Vi 和 Vj 间无通路
  }
```

7.37

7.38 已知 n 个顶点的有向图，用邻接矩阵表示，编写函数，计算每对顶点之间的最短路径。

本题用 FLOYD 算法直接求解如下：

【算法 7.38】

```
 void ShortPath_FLOYD(AdjMatrix g)
    //求具有 n 个顶点的有向图每对顶点间的最短路径
 {AdjMatrix length; //length[i][j]存放顶点 vi 到 vj 的最短路径长度。
  for(i=1;i<=n;i++)
     for(j=1;j<=n;j++) length[i][j]=g[i][j]; //初始化。
  for(k=1;k<=n;k++)
     for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
           if(length[i][k]+length[k][j]<length[i][j])
               length[i][j]=length[i][k]+length[k][j];
 }//算法结束
```

7.39 设计算法求距离顶点 V_0 的最短路径长度(以弧数为单位)为 K 的所有顶点，要求尽可能地节省时间。

【题目分析】 本题应用宽度优先遍历求解。若以 v0 作生成树的根为第 1 层，则距顶点 v0 最短路径长度为 K 的顶点均在第 K+1 层。可用队列存放顶点，将遍历访问顶点的操作改为入队操作。队列中设头尾指针 f 和 r，用 level 表示层数。

【算法 7.39】

```
void bfs_K(graph g,int v0,K)
    //输出无向连通图 g 中距顶点 v0 最短路径长度为 K 的顶点
 {int Q[]; //Q 为顶点队列，容量足够大
  int f=0,r=0,t=0;    //f 和 r 分别为队头和队尾指针，t 指向当前层最后顶点
  int level=0,flag=0;//层数和访问成功标记
```

```
visited[v0]=1;      //设 v0 为根
Q[++r]=v0; t=r; level=1; //v0 入队
while(f<r && level<=K+1)
  {v=Q[++f];
   w=GraphFirstAdj(g,v);
   while(w!=0)   //w!=0 表示邻接点存在
     {if(visited[w]==0)
       {Q[++r]=w; visited[w]=1;//邻接点入队列
        if(level==K+1){ printf("距顶点 v0 最短路径为 k 的顶点%d
        ",w); flag=1;}
        }//if
      w=GraphNextAdj(g ,v ,w);
     }//while(w!=0)
   if(f==t) {level++; t=r; }
    //当前层处理完，修改层数，t 指向下一层最后一个顶点
  }//while(f<r && level<=K+1)
 if(flag==0) printf("图中无距 v0 顶点最短路径为%d 的顶点。\n",K);
}//算法结束
```

[算法讨论]本题亦可采取另一个算法。由于在生成树中结点的层数等于其双亲层次数加 1，故可设顶点和层次数 2 个队列，其入队和出队操作同步，其核心语句段如下：

```
QueueInit(Q1) ; QueueInit(Q2); //Q1 和 Q2 是顶点和顶点所在层次数的
队列
visited[v0]=1;  //访问数组初始化，置 v0 被访问标记
level=1; flag=0; //是否有层次为 K 的顶点的标志
QueueIn(Q1,v0); QueueIn(Q2,level);   //顶点和层数入队列
while(!empty(Q1) && level<=K+1)
  {v=QueueOut(Q1); level=QueueOut(Q2);//顶点和层数出队
   w=GraphFirstAdj(g,v0);
   while(w!=0)  //邻接点存在
     {if(visited[w]==0)
        if(level==K+1)
          {printf("距离顶点 v0 最短路径长度为 K 的顶点是%d\n",w);
           visited[w]=1;      flag=1;      QueueIn(Q1      ,w);
QueueIn(Q2,level+1); }
      w=GraphNextAdj(g ,v ,w);
     }//while(w!=0)
  }//while(!empty(Q1) && level<K+1)
 if(flag==0) printf( "图中无距 v0 顶点最短路径为%d 的顶点。\n",K);
```

7.40 设有 n(n>0)个顶点的无向连通图 G， 可以邻接矩阵 A_{nxn} 存储，由于邻接矩阵的对称性，只将其下三角顺序存储在数组 S 中。请编写对以数组 S 存储的图 G 进行宽度优先遍历的算法。

【题目分析】 由宽度优先遍历的定义，首先访问任一顶点，然后访问该顶点的

未曾访问的邻接点，如此下去，直至全部顶点访问完成。在邻接矩阵中，第 i 行非零元素都是第 i 个顶点的邻接点，而在压缩存储下，找某顶点的邻接点要遍历整个数组。矩阵元素的下标 i 和 j 和其在一维数组 S 中的序号 k 的关系：

由 $k=i(i+1)/2+j \quad (i>=j,\ \ 0<=k<n(n+1)/2)$

得 $i=\left|(-3+\sqrt{9+8k})/2\right|$ 和 j=k-i(i+1)/2

在一维数组中，只有非零元素才是顶点。为简单计，当访问完一个顶点后，就将其在一维数组中的位序置零；由于是图的遍历，当全部顶点访问完后就直接结束算法。

【算法 7.40】

```
#define n  用户图的顶点数
#define m  n(n+1)/2
int nodes=0, visited[n]={0}; //顶点计数和访问标志数组
 void index(int k,*i,*j)
    //由非零元素在一维数组 S 中的序号 k 计算其在邻接矩阵中的下标 i 和 j
    {*i=⌈(-3+sqrt(9+8*k))/2⌉; *j=k-(*i)*(*i+1)/2
    }
void  Tri_bfs(int v)
   //对下三角存储的无向连通图作宽度优先遍历，v 是遍历的开始顶点
{nodes++; QueueInit(Q); QueueIn(Q,v);
 printf(v); visited[v]=1;              //初始化
 while(!QueueEmpty(Q) && nodes<=n)
   {v=QueueOut(Q);
    for(k=0; k<m; k++)
      if(s[k]!=0)
        {index(k,&i,&j);  //求非零元素在邻接矩阵中的下标
         if(i==v || j==v) //顶点 i 或 j 是顶点 v
            if(i==v)      //顶点 i 是顶点 v
              {if(visited[j]==0)  //顶点 j 是顶点 v 的邻接点，且尚未访问
                 {nodes++;   printf(j);   visited[j]=1;   s[k]=0; QueueIn(Q,j);}
               }
            else  //顶点 j 是顶点 v
              {if(visited[i]==0)  //顶点 i 是顶点 v 的邻接点，且尚未访问
                 {nodes++;   printf(i);   visited[i]=1;   s[k]=0; QueueIn(Q,i);}
               }//
           }//index(k,&i,&j)
       }//while(!QueueEmpty(Q) && nodes<=n)
    }//算法结束
```

[**算法讨论**]对于连通图，进入 BFS 一次就可访问完图的全部顶点；对于非连通图，进入 BFS 一次就可访问完图的一个连通分量。若要遍历全部顶点，在调用 BFS 的函数中加入语句：

```
for(vi=0; vi<n;vi++)
  if(visited[vi]==0) Tri_bfs(vi);
```

第 8 章 动态存储结构

8.1在伙伴系统中的伙伴是指任意两块大小相同、位置相邻的内存块。这种说法对吗？

【解答】不对。只有同一内存块分裂的两块才互称伙伴。

8.2 最佳适配法与最先适配法相比，前者容易增加闲置空间的碎片。这种说法对吗？

【解答】对。

8.3 设内存中可利用空间已连成一个单链表，对用户的存储空间需求，一般有哪三种分配策略？

【解答】

首次拟合法：从链表头指针开始查找，找到第一个大于等于所需空间的结点即分配。

最佳拟合法：链表结点大小增序排列，找到第一个大于等于所需空间的结点即分配。

最差拟合法：链表结点大小逆序排列，总从第一个结点开始分配，将分配后结点所剩空间插入到链表适当位置。

首次拟合法适合事先不知道请求分配和释放信息的情况,分配时需查询,释放时插在表头。最佳拟合法适用于请求分配内存大小范围较宽的系统,释放时容易产生存储量很小难以利用的内存碎片,同时保留那些很大的内存块以备将来可能发生的大内存量的需求，分配与回收均需查询。 最差拟合法适合请求分配内存大小范围较窄的系统,分配时不查询,回收时查询,以便插入适当位置。

8.4 计算起始二进制地址为 011011110000，长度为 4（十进制）的块的伙伴地址是多少？

【解答】011011110100

8.5 地址为（1664）$_{10}$大小为（128）$_{10}$的存储块的伙伴地址是什么？

地址为（2816）$_{10}$大小为（64）$_{10}$的存储块的伙伴地址是什么？

【解答】

（1）buddy(1664,7)=1664-128=1536

（2）buddy(2816,6)=2816+64=2880

8.6 试叙述动态存储分配伙伴系统的基本思想，它和边界标识法不同点是什么？

【解答】动态存储分配伙伴系统的基本思想是：在伙伴系统中，无论占用块或空闲块，其大小均为 2 的 k(k 为≥0 的正整数)次幂。若内存容量为 2^m，则空闲块大小只能是 2^0，2^1，2^2，…，2^m。由同一大块分裂而得的两个小块互称“伙伴空间”，如内存大小为 2^{10} 的块分裂成两个大小为 2^9 的块。只有两个“伙伴空间”才能合并成一个大空间。

起始地址为 p，大小为 2^k 的内存块，其伙伴的起始地址为：

$$\text{buddy}(p,k)=\begin{cases} p+2^k & (\text{若 } p \text{ MOD } 2^{k+1}=0) \\ p-2^k & (\text{若 } p \text{ MOD } 2^{k+1}=2^k) \end{cases}$$

边界标识法在每块的首尾均有“占用”/“空闲”标志，空闲块合并方便。伙伴系统算法简单，速度快，但只有互为伙伴的两个空闲块才可合并，因而易产生虽空闲但不能归并的碎片。

8.7 已知一个大小为 512 个字长的存储，假设先后有 6 个用户申请大小分别为 23，45，52，100，11 和 19 的存储空间，然后再顺序释放大小为 45，52，11 的占用块。假设以伙伴系统实现动态存储管理。

（1） 画出可利用空间表的初始状态。

（2） 画出为 6 个用户分配所需要的存储空间后可利用空间表的状态以及每个用户所得到的存储块的起始地址。

（3） 画出在回收 3 个占用块之后可利用空间表的状态。

【解答】因为 $512=2^9$，可利用空间表的初始状态图如 8-1 所示。

当用户申请大小为 23 的内存块时，因 $2^4<23<=2^5$，但没有大小为 2^5 的块，只有大小为 2^9 的块，故将 2^9 的块分裂成两个大小为 2^8 的块，其中大小为 2^8 的一块挂到可利用空间表上，另一块再分裂成两个大小为 2^7 的块。又将其中大小为 2^7 的一块挂到可利用空间表上，另一块再分裂成两个大小为 2^6 的块，一块 2^6 的块挂到可利用空间表上，另一块分裂成两个大小为 2^5 的块，其中一块挂到可利用空间表上，另一块分给用户(地址 0—31)。如此下去，最后每个用户得到的存储空间的起始地址如图 8-2，6 个用户分配所需要的存储空间后可利用空间表的状态如图 8-3。

在回收时，因为给申请 45 的用户分配了 2^6，其伙伴地址是 0，在占用中，不能合并，只能挂到可利用空间表上。在回收大小为 52 的占用块时，其伙伴地址是 192，也在占用。回收大小为 11 的占用块时，其伙伴地址是 48，可以合并为大小 2^5 的块，挂到可利用空间表上。回收 3 个占用块之后可利用空间表的状态如图 8-4。

存储大小	起始地址
23	0
45	64
52	128
100	256
11	32
19	192

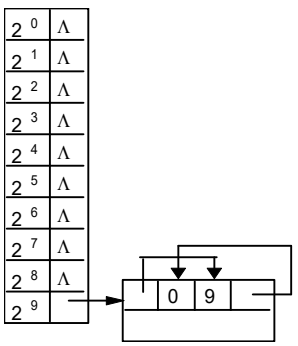


图 8-2　　图 8-1

（注：在图 8.3 和图 8.4 画上了占用块，从原理上，只有空闲块才出现在“可利用空间表”中。）

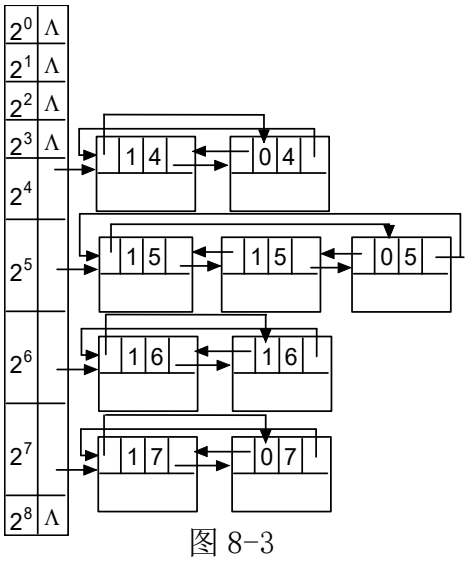


图 8-3

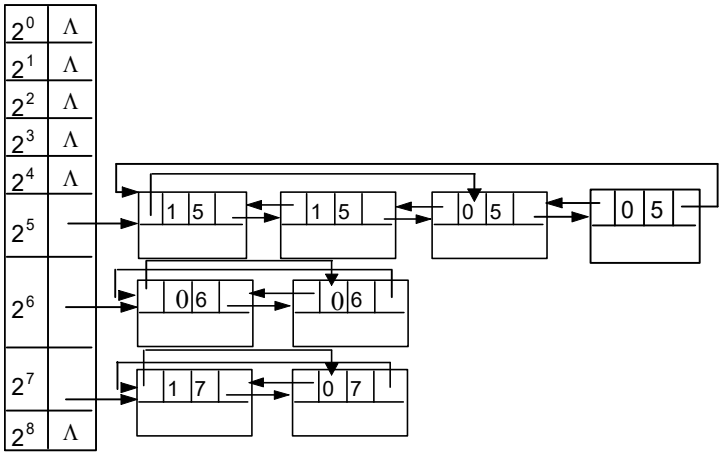
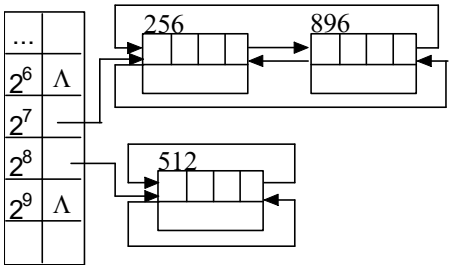
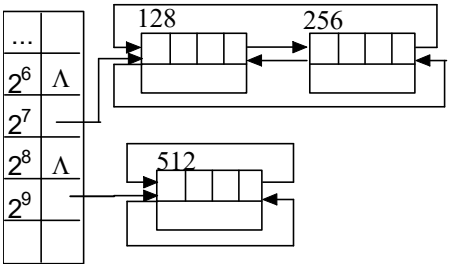


图 8-4

8.9 下图所示的伙伴系统中，回收两块首地址分别为 768 及 128，大小为 2^7 的存储块，请画出回收后该伙伴系统的状态图。



【解答】因为 $768 \% 2^{7+1}=0$，所以 768 和 $768+2^7=896$ 互为伙伴，伙伴合并后，首址为 768，块大小为 2^8。因为 $768 \% 2^{8+1}=2^8$，所以，所以首址 768 大小为 2^8 的块和首址 512 大小为 2^8 的块合并，成为首址 512 大小为 2^9 的空闲块。因为 $128 \% 2^{7+1}=2^7$，其伙伴地址为 $128-2^7=0$，将其插入可利用空间表中。回收后该伙伴系统的状态图如下。



第 9 章　集合

一、基础知识题

9.1 若对长度均为 n 的有序的顺序表和无序的顺序表分别进行顺序查找，试在下列三种情况下分别讨论二者在等概率情况下平均查找长度是否相同？

（1）查找不成功，即表中没有和关键字 K 相等的记录；

（2）查找成功，且表中只有一个和关键字 K 相等的记录；

（3）查找成功，且表中有多个和关键字 K 相等的记录，要求计算有多少个和关键字 K 相等的记录。

【解答】

（1）平均查找长度不相同。在有序的顺序表中查找时，在 n+1(小于任何一个和大于第 n)个位置均可能失败；查找无序的顺序表时，查找失败都是在第 n+1 个位置，其平均查找长度是 n+1。

（2）平均查找长度相同。在 n 个位置上均可能成功。

（3）平均查找长度不相同。前者在某个位置上(1<=i<=n)查找成功时，和关键字 K 相等的记录是连续的，而后者要查找完顺序表的全部记录。

9.2 在查找和排序算法中，监视哨的作用是什么？

【解答】监视哨的作用是免去查找过程中每次都要检测整个表是否查找完毕，提高了查找效率。

9.3 用分块查找法，有 2000 项的表分成多少块最理想？每块的理想长度是多少？若每块长度为 25 ，平均查找长度是多少？

【解答】分成 45 块，每块的理想长度为 45(最后一块长 20)。若每块长 25，则平均查找长度为 ASL=(80+1)/2+(25+1)/2=53.5(顺序查找确定块)，或 ASL=19(折半查找确定块，块内确定元素)。

9.4　　用不同的输入顺序输入 n 个关键字,可能构造出的二叉排序树具有多少种不同形态?

【解答】$\frac{1}{n+1}*\frac{(2n)!}{n!*n!}$

9.5证明若二叉排序树中的一个结点存在两个孩子，则它的中序后继结点没有左孩子，中序前驱结点没有右孩子。

【证明】根据中序遍历的定义，该结点的中序后继是其右子树上按中序遍历的第一个结点，即右子树上值最小的结点，即叶子结点或仅有右子树的结点，它们都没有左孩子；而其中序前驱是其左子树上按中序遍历的最后个结点，即左子树上值最大的结点，即叶子结点或仅有左子树的结点，没有右孩子。命题得证。

9.6 对于一个高度为 h 的 AVL 树，其最少结点数是多少？反之，对于一个有 n 个结点的 AVL 树， 其最大高度是多少？最小高度是多少?

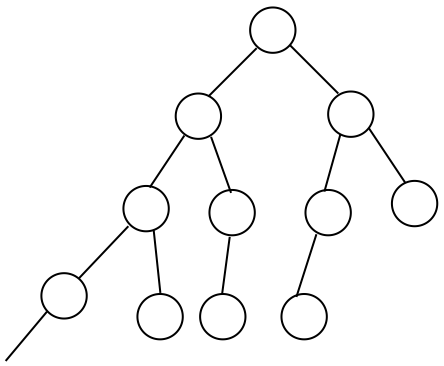
9.7【解答】设以 N_h表示深度为 h 的 AVL 树中含有的最少结点数。显然，N_0=0，N_1 =1，N_2 =2，且 $N_h = N_{h-1} + N_{h-2} +1$(h≥2)。这个关系与斐波那契序列类似，用归纳法可以证明：当 h≥0 时，$N_h = F_{h+2} -1$，而 F_h约等于 $\Phi^h/\sqrt{5}$ (其中 $\Phi=(1+\sqrt{5})/2$)，则 N_h约等于 $\Phi^{h+2}/\sqrt{5}-1$(即高度为 h 的 AVL 树具有的最少结点数),这也就是有 n 个结点的 AVL 树的最大高度。有 n 个结点的 AVL 树的　最小高度是$\lceil \log_2(n+1) \rceil$。

9.8 试推导含有 12 个结点的平衡二叉树的最大深度，并画出一棵这样的树。

【解答】深度为 n 的 AVL 树中的最少结点数为 $N_n = F_{n+2} - 1$

所以 12= $F_{n+2} -1$，有 F_{n+2}=13，求得 n+2=7(Fibonacci 数列第一项的值假设为 1，对应于二叉树表示有一个结点的二叉树的深度为 1)，所以 n=5。

可表示为如下图所示的 AVL 树：



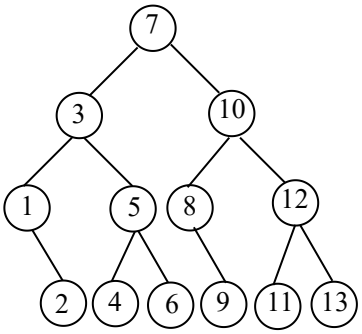
○

9.9 假定有 n 个关键字，它们具有相同的哈希函数值，用线性探测方法把这 n 个关键字存入到哈希表中要做多少次探测？

【解答】n 个关键字都是同义词，因此，用线性探测法将第一个关键字存入时不会发生冲突，对其余关键字存入时都会发生冲突，所以探测的次数应为 $1+2+\cdots+n=n(n+1)/2$ 次。

9.10 建立一棵具有 13 个结点的判定树，并求其成功和不成功的平均查找长度值各为多少。

【解答】



查找成功时的平均查找长度为：$ASL_{succ}=(1\cdot1+2\cdot2+3\cdot4+4\cdot6)/13=41/13$

查找不成功时的平均查找长度为：$ASL_{un}=(2\cdot3+12\cdot4)/14=54/14=27/7$

9.11 二叉排序树中关键字互不相同，则其中关键字最小值结点无左孩子，关键字最大值结点无右孩子，此命题是否正确？最小值结点和最大值结点一定是叶子吗？一个新结点总是插在二叉排序树的某叶子上吗？

【解答】对二叉排序树进行中序遍历可以得到结点的有序序列，中序遍历的第一个结点是最小值结点，中序遍历的最后一个结点是最大值结点。题目给出二叉排序树中关键字互不相同，则其中最小值结点必无左孩子，最大值结点必无右孩子，此命题是正确的。

最小值结点和最大值结点不一定是叶子结点。一个新结点不一定总是插在二叉排序树的某叶子上，但插入后一定是叶子结点。

9.12 回答问题并填空

（1）散列表存储的基本思想是什么？

（2）散列表存储中解决碰撞的基本方法有哪些？其基本思想是什么？

（3）用线性探查法解决碰撞时，如何处理被删除的结点？为什么？

【解答】

(1)散列表存储的基本思想是用关键字的值决定数据元素的存储地址。

(2)散列表存储中解决碰撞的基本方法：

① 开放定址法

形成地址序列的公式是：$H_i=(H(key)+d_i)\%\ m$，其中 m 是表长，d_i 是增量。根据 d_i 取法不同，又分为三种：

a．$d_i=1，2，\cdots，m-1$ 称为线性探测再散列，其特点是逐个探测表空间，只要散列表中有空闲空间，就可解决碰撞，缺点是容易造成“聚集”，即不是同义词的关键字争夺同一散列地址。

b．$d_i=1^2，-1^2，2^2，-2^2，\cdots，\pm k^2(k\leqslant m/2)$ 称为二次探测再散列，它减少了聚集，但不容易探测到全部表空间，只有当表长为形如 4j+3(j 为整数)的素数时才有可能。

c．d_i =伪随机数序列，称为随机探测再散列。

② 再散列法 $H_i=RH_i(key)$ i=1，2，…，k，是不同的散列函数，即在同义词产生碰撞时，用另一散列函数计算散列地址，直到解决碰撞。该方法不易产生“聚集”，但增加了计算时间。

③ 链地址法 将关键字为同义词的记录存储在同一链表中，散列表地址区间用 H[0..m-1]表示，数组元素初始值为空指针。凡散列地址为 i($0\leqslant i\leqslant m-1$)的记录均插在以 H[i]为头指针的链表中。这种解决方法中数据元素个数不受表长限制，插入和删除操作方便，但增加了指针的空间开销。这种散列表常称为开散列表，因为数据元素个数不受表长限制。而①中的散列表称闭散列表，含义是元素个数受表长限制。

④ 建立公共溢出区 设 H[0..m-1]为基本表，凡关键字为同义词的记录，都填入溢出区 O[0..m-1]。

(3) 用线性探查法解决碰撞时，要在被删除结点的散列地址处作标记，不能物理的删除。否则，中断了查找通路。

9.13 如何衡量哈希函数的优劣？简要叙述哈希表技术中的冲突概念，并指出三种解决冲突的方法。

【解答】评价哈希函数优劣的因素有：能否将关键字均匀影射到哈希空间上，有无好的解决冲突的方法，计算哈希函数是否简单高效。由于哈希函数是压缩映像，冲突难以避免。

解决冲突的方法见上面 9.12 题。

9.14 设有一组关键字{9，01，23，14，55，20，84，27}，采用哈希函数：H（key）=key % 7 ，表长为 10，用开放地址法的二次探测再散列方法 Hi=(H(key)+di) % 10(di=1^2，2^2，3^2，…，)解决冲突。要求：对该关键字序列构造哈希表，并计算查找成功的平均查找长度。

【解答】

散列地址	0	1	2	3	4	5	6	7	8	9
关键字	14	01	9	23	84	27	55	20		

比较次数	1	1	1	2	3	4	1	2		

平均查找长度：$ASL_{succ}=(1+1+1+2+3+4+1+2)/8=15/8$

以关键字 27 为例：H(27)=27%7=6(冲突)　　$H_1=(6+1)\%10=7$(冲突)

$H_2=(6+2^2)\%10=0$(冲突)　　$H_3=(6+3^3)\%10=5$　　所以比较了 4 次。

9.15　对下面的关键字集合{30，15，21，40，25，26，36，37}，若查找表的装填因子为 0.8，采用线性探测再散列方法解决冲突。要求：

（1）设计哈希函数；

（2）画出哈希表；

（3）计算查找成功和查找失败的平均查找长度。

【解答】由于装填因子为 0.8，关键字有 8 个，所以表长为 8/0.8=10。

(1)用除留余数法，哈希函数为 H(key)=key % 7

(2)

散列地址	0	1	2	3	4	5	6	7	8	9
关键字	21	15	30	36	25	40	26	37		
比较次数	1	1	1	3	1	1	2	6		

(3)计算查找失败时的平均查找长度，必须计算不在表中的关键字，当其哈希地址为 i(0≤i≤m-1)时的查找次数。本例中 m=10。故查找失败和查找成功时的平均查找长度分别为：

$ASL_{unsucc}=(9+8+7+6+5+4+3+2+1+1)/10=4.6$　　$ASL_{succ}=16/8=2$

9.16　设哈希函数 H（k）=3K％11，散列地址空间为 0～10，对关键字序列（32，13，49，24，38，21，4，12）按下述两种解决冲突的方法构造哈希表：

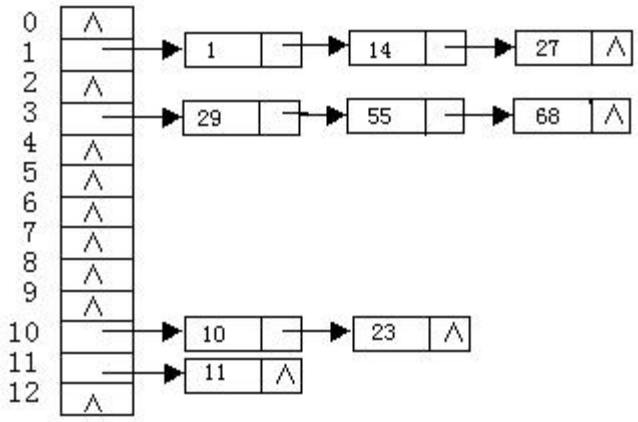
（1）线性探测再散列

（2）链地址法，

并分别求出等概率下查找成功时和查找失败时的平均查找长度。

【解答】. (1)

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键字	13	22		53	1		41	67	46		51		30
比较次数	1	1		1	2		1	2	1		1		1



(2)装填因子=9/13=0.7

(3)ASL_{succ}=11/9

(4)ASL_{unsucc}=29/13

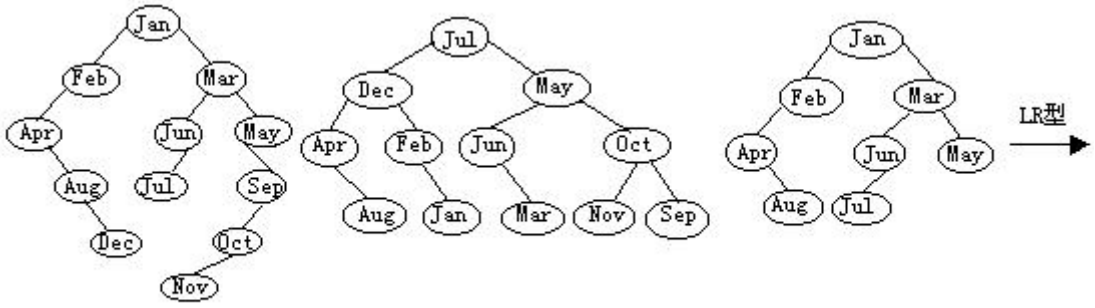
9.17　已知长度为 12 的表{Jan，Feb，Mar，Apr，May，June，July，Aug，Sep，Oct，Nov，Dec}

（1） 试按表中元素的次序依次插入一棵初始为空的二叉排序树，并求在等概率情况下查找成功的平均查找长度。

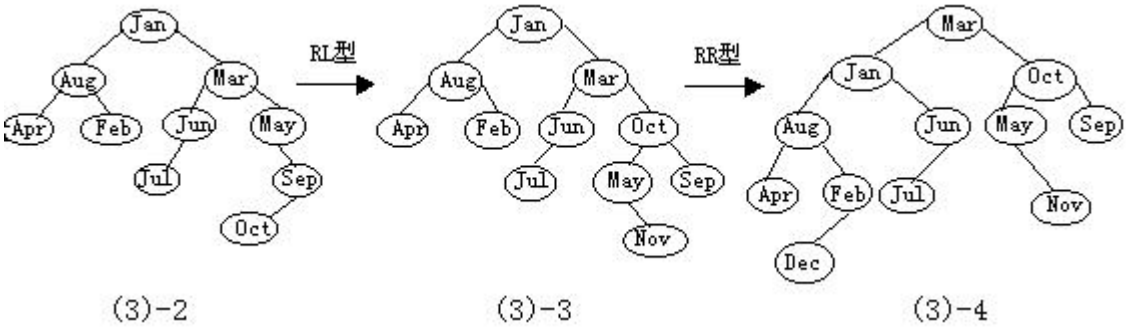
（2） 用表中元素构造一棵最佳二叉排序树，求在等概率的情况下查找成功的平均查找长度。

（3） 按表中元素顺序构造一棵 AVL 树，并求其在等概率情况下查找成功的平均查找长度。

【解答】



(1)　　(2)　　(3)-1



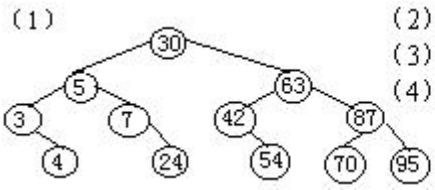
(3)-2　　(3)-3　　(3)-4

(1)ASL_{suss} =(1*1+2*2+3*3+4*3+5*2+6*1)/12=42/12
(2)ASL_{suss} =(1*1+2*2+4*3+5*4)/12=37/12
(3)ASL_{suss} =(1*1+2*2+4*3+4*4+5*1)/12=38/12

9.18　假定对有序表：(3，4，5，7，24，30，42，54，63，72，87，95)进行折半查找，试回答下列问题：

(1) 画出描述折半查找过程的判定树；
(2) 若查找元素 54，需依次与哪些元素比较?
(3) 若查找元素 90，需依次与哪些元素比较?
(4) 假定每个元素的查找概率相等，求查找成功时的平均查找长度。

【解答】



(2) 若查找元素 54，需依次和元素 30、63、42、54 比较，查找成功
(3) 若查找元素 90，需依次和元素 30、63、87、95 比较，查找失败
(4) ASL_{suss} = (1*1+2*2+4*3+5*4) /12=37/12

9.19　直接在二叉排序树中查找关键字 K 与在中序遍历输出的有序序列中查找关键字 K，其效率是否相同？输入关键字有序序列来构造一棵二叉排序树，然后对此树进行查找，其效率如何？为什么？

【解答】在二叉排序树上查找关键字 K，走了一条从根结点至多到叶子的路径，时间复杂度是 O(logn)，而在中序遍历输出的序列中查找关键字 K，时间复杂度是 O(n)。按序输入建立的二叉排序树，蜕变为单枝树，其平均查找长度是(n+1)/2，时间复杂度也是 O(n)。

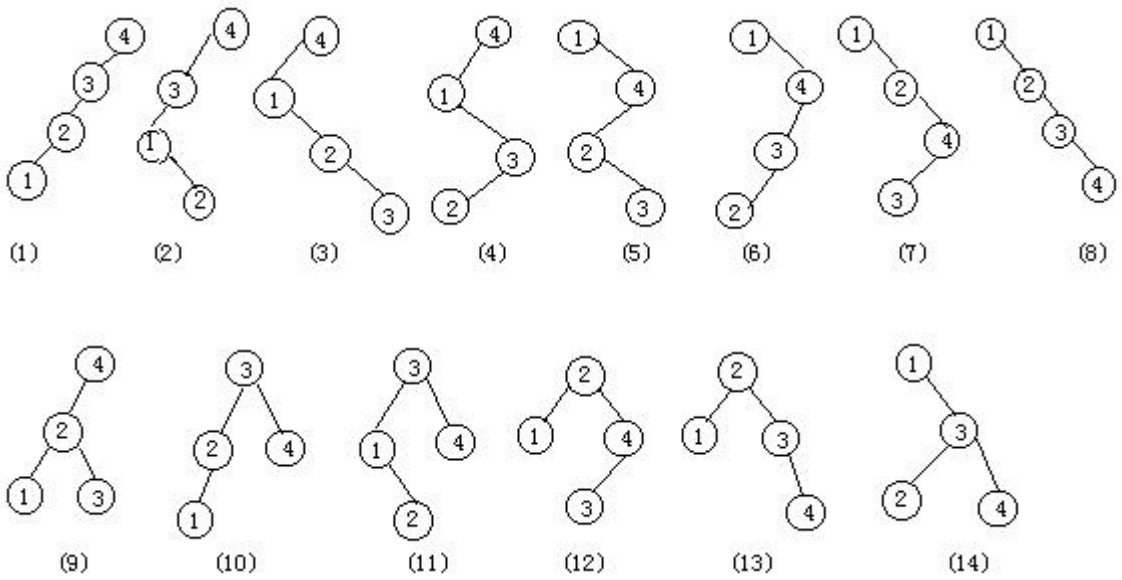
9.20　设二叉排序树中关键字由 1 到 1000 的整数组成,现要查找关键字为 363 的结点，下述关键字序列哪一个不可能是在二叉排序树中查到的序列？说明原因。

（1）51，250，501，390，320，340，382，363
（2）24，877，125，342，501，623，421，363

【解答】序列(2)不可能是二叉排序树中查到 363 的序列。查到 501 后，因 363<501，后面应出现小于 501 的数，但序列中出现了 623，故不可能。

9.21　用关键字 1，2，3，4 的四个结点（1）能构造出几种不同的二叉排序树？其中（2）最优查找树有几种？（3）AVL 树有几种？(4)完全二叉树有几种？试画出这些二叉排序树。

【解答】(1)本题的本质是给定中序序列 1、2、3、4，有几种不同的二叉排序树，即该中序序列相当多少不同的前序序列，这是树的计数问题。设中序序列中元素数为 n，则二叉数的数目为 $1/(n+1)C_{2n}^{n}$，这里 n=4，故有 14 种。图示如下：



(2)最优查找树有 4 种，图中⑽ ⑾ ⑿ ⒀

(3)AVL 树也有 4 种，图中⑽ ⑾ ⑿ ⒀

(4)完全二叉树有 1 种，图中⑽

9.22　简要叙述 B-树与 B+树的区别?

【解答】m 阶的 B+树和 B-树主要区别有三：

(1)有 n 棵子树的结点中含有 n(B-树中 n-1)个关键字；

(2)B+树叶子结点包含了全部关键字信息，及指向含关键字记录的指针，且叶子结点本身依关键字大小自小到大顺序链接；

(3)B+树的非终端结点可以看成是索引部分，结点中只含其子树(根结点)中最大(或最小)关键字。B+树的查找既可以顺序查找，也可以随机查找，B-树只能随机查找。

9.23　包括 n 个关键字的 m 阶 B-树在一次检索中最多涉及多少个结点？（要求写出推导过程。）

【解答】本题等价于“含有 n 个关键字的 m 阶 B-树的最大高度是多少”？一次检索中最多走一条从根到叶子的路径，由于根结点至少有两棵子树，其余每个(除叶子)结点至少有$\lceil m/2 \rceil$棵子树，则第三层至少有$\lceil m/2 \rceil$*2 个结点，第 1+1 层至少有 $2*\lceil m/2 \rceil^{l-1}$个结点。设 B-树深度为 1+1，即第 1+1 层是叶子结点，叶子结点数

是 n+1(下面推导)，故有 n+1≥2*⌈m/2⌉$^{l-1}$，即 l≤$\log_{\lceil m/2\rceil}(\frac{n+1}{2})$+1。

【注】 推导 B-树中叶子结点数 s 与关键字数 n 的关系式：s=n+1

设 B-树某结点的子树数为 C_i，则该结点的关键字数 $N_i=C_i-1$。对于有 k 个结点的 B-树，有

$$\sum N_i=\sum (C_i-1)=\sum C_i-k \quad (1\leqslant i\leqslant k) \quad \cdots\cdots(1)$$

因为 B 树上的关键字数，即 $\sum N_i=n$ $(1\leqslant i\leqslant k)$ ……(2)

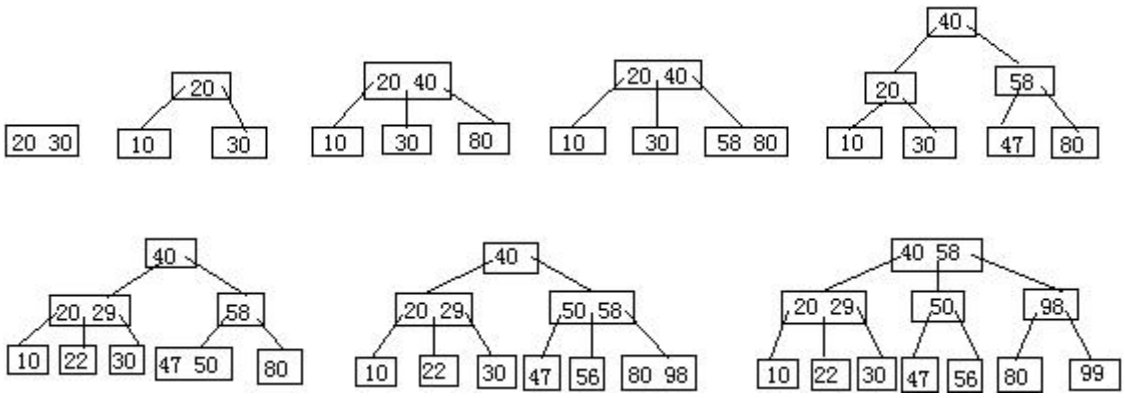
而 B-树上的子树数可这样计算：每个结点(除根结点)都是一棵子树，设叶子(子树)数为 s；则

$$\sum C_i=(k-1)+s \quad (1\leqslant i\leqslant k) \quad \cdots\cdots(3)$$

综合(1)(2)(3)式，有 s=n+1。证毕。

9.24 设有一棵空的 3 阶 B 树，依次插入关键字 30，20，10，40，80，58，47，50，29，22，56，98，99，请画出该树。

【解答】



9.25 含 9 个叶子结点的 3 阶 B-树中至少有多少个非叶子结点？含 10 个叶子结点的 3 阶 B-树中至多有多少个非叶子结点？

【解答】含 9 个叶子结点的 3 阶 B-树至少有 4 个非叶子结点，当每个非叶子结点均含 3 棵子树，第三层是叶子结点时就是这种情况。当 4 层 3 阶 B-树有 10 个叶子结点时，非叶子结点达到最大值 8 个：其中第一层 1 个，第二层 2 个，第三层 5 个。

9.26 选择题：对顺序表进行二分查找时，要求顺序表必须：

A. 以顺序方式存储　　B. 以顺序方式存储，且数据元素有序

C. 以链接方式存储　　D. 以链接方式存储，且数据元素有序

【解答】B

9.27 选择题：下列二叉排序树中查找效率最高的是：

A. 平衡二叉树　　B. 二叉查找树

C. 没有左子树的二叉排序树　　D. 没有右子树的二叉排序树

【解答】A

二、算法设计题

9.28　元素集合已存入整型数组 A[1..n]中，试写出依次取 A 中各值 A[i](1<=i<=n)构造一棵二叉排序树 T 的非递归算法。

【题目分析】本题以非递归形式建立二叉排序树

【算法 9.28】

```
void  CSBT(BSTree T，ElemType A[]，int n)
  //以存储在数组 K 中的 n 个关键字，建立一棵初始为空的二叉排序树
  {for(i=1；i≤n；i++)
    {p=T；f=null；//初始调用 CSBT 时，T=null
     while(p!=null)
        if(p->data<A[i])
            {f=p；p=p->rchild；  } // f 是 p 的双亲
        else if(p->data>A[i]){f=p；p=p->lchild；}
     s=(BSTree)malloc(sizeof (BiNode))；      // 申请结点空间
     s->data=A[i]；s->lchild=null；s->rchild=null；
     if(f==null)T=s；                          //根结点
     else if(s->data<f->data)f->lchild=s；//左子女
          else  f->rchild=s；//右子树根结点的值大于根结点的值
   }//算法结束
```

9.29　编写删除二叉排序树中值是 X 的结点的算法。要求删除结点后仍然是二叉排序树，并且高度没有增长。

【题目分析】在二叉排序树上删除结点，首先要查找该结点。查找成功后，若该结点无左子树，则可直接将其右子树的根结点接到其双亲结点上；若该结点有左子树，则用其左子树中按中序遍历的最后一个结点代替该结点，从而不增加树的高度。

【算法 9.29】

```
   void Delete(BSTree bst, keytype  X)
  //在二叉排序树 bst 上，删除其关键字为 X 的结点。
   {BSTree f,p=bst;
    while(p && p->key!=X) //查找值为 X 的结点
      if(p->key>X) {f=p; p=p->lchild;}
      else {f=p; p=p->rchild;}
    if(p==null) {printf(“无关键字为 X 的结点\n”); exit(0);}
    if {p->lchild==null} //被删结点无左子树
      if(f->lchild==p) f->lchild=p->rchild;//将被删结点的右子树接到其双亲上
       else f->rchild=p->rchild;
    else {q=p; s=p->lchild;            //被删结点有左子树
         while(s->rchild !=null)      //查左子树中最右下的结点(中序最后结点)
            {q=s; s=s->rchild;}
         p->key=s->key;                //结点值用其左子树最右下的结点的值代替
         if(q==p)
```

```
                p->lchild=s->lchild;//被删结点左子树的根结点无右子女
            else q->rchild=s->lchild; //s 是被删结点左子树中序序列最后一个结点
            free(s);
           }//else
    }//算法结束
```

9.30　假设二叉排序树的各个元素值均不相同，设计一个递归算法按递减次序打印各元素的值。

【题目分析】按着“右子树－根结点－左子树”遍历二叉排序树，并输出结点的值。

【算法 9.30】

```
 void InOrder(BSTree bt)//按递减次序输出二叉排序树结点的值
 {BiTree s[],p=bt;        //s 是元素为二叉树结点指针的栈，容量足够大
  int top=0;
  while(p || top>0)
    {while(p)
        {s[++top]=p; bt=p->rchild;} //沿右子树向下
     if(top>0)
        {p=s[top--]; printf(p->data); p=p->lchild;}
     }
  }//结束
```

以下是递归输出，算法思想是一样的。

```
 void InvertOrder(BSTree bt)//按递减次序输出二叉排序树结点的值
  {BiTree p=bt;
  if(p)
    {InOrder(bt->rchild); //中序遍历右子树
      printf(p->data);     //访问根结点
      InOrder(bt->lchild); //中序遍历左子树
    }//if
 }//结束
```

9.31　设记录 R1，R2，…，Rn 按关键字值从小到大顺序存储在数组 r[1..n]中，在 r[n+1]处设立一个监视哨，其关键字值为+∞；试写一查找给定关键字 k 的算法;并画出此查找过程的判定树，求出在等概率情况下查找成功时的平均查找长度。

【算法 9.31】

```
int Search(rectype r[], int  n, keytype k)
    //在 n 个关键字从小到大排列的顺序表中，查找关键字为 k 的结点
    {r[n+1].key=k;   //在高端设置监视哨
     i=1;
     while(r[i].key<k) i++;
     return i%(n+1);
    }//算法 search 结束
```

查找过程的判定树是单枝树，限于篇幅不再画出。本题中虽然表按关键字有序，但进行顺序查找，查找成功的平均查找长度亦为(n+1)/2。

9.32　在二叉排序树中查找值为 X 的结点，若找到，则记数（count）加 1；否则，作为一个新结点插入树中，插入后仍为二叉排序树，写出其递归和非递归算法（要求给出结点的定义）。

【算法 9.32】

```
typedef struct node
   {ElemType data;
    int count;
    struct  node *llink, *rlink;
   }BiTNode, *BSTree;
 void Search_InsertX(BSTree t, ElemType X)
   //在二叉排序树 t 中查找值为 X 的结点，若查到，则其结点的 count 域值增 1
   //否则，将其插入到二叉排序树中
    {p=t;
     while(p!=null && p->data!=X)  //查找值为 X 的结点，f 指向当前结点的双亲
          {f=p;
           if(p->data<X) p=p->rlink; else p=p->llink;
          }
       if(!p) // 无值为 x 的结点，插入之
          {p=(BiTNode *)malloc(sizeof (BiTNode));
           p->data=X; p->llink=null; p->rlink=null; p->count=0;
           if(t==null) t=p; //若初始为空树，则插入结点为根结点
           else if(f->data>X)
                   f->llink=p;
                else f->rlink=p;
          }
       else  p->count++; // 查询成功，值域为 X 的结点的 count 增 1
    }// Search_InsertX
```

9.33　假设一棵平衡二叉树的每个结点都标明了平衡因子 b，试设计一个算法，求平衡二叉树的高度。

【题目分析】 因为二叉树各结点已标明了平衡因子 b，故从根结点开始记树的层次。根结点的层次为 1，每下一层，层次加 1，直到层数最大的叶子结点，这就是平衡二叉树的高度。当结点的平衡因子 b 为 0 时，任选左、右一分枝向下查找，若 b 不为 0，则沿左(当 b=1 时)或右(当 b=-1 时)子树向下查找。

【算法 9.33】

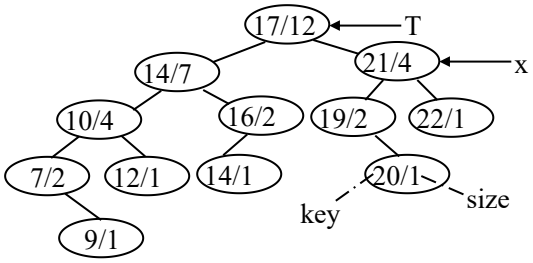
```
 int Height(AVLTree t)// 求平衡二叉树 t 的高度
   {level=0; p=t;
    while(p)
      {level++;  // 树的高度增 1
```

```
        if(p->bf<0)p=p->rchild; //bf=-1 沿右分枝向下
        else p=p->lchild;       //bf>=0 沿左分枝向下
       }//while
     return  (level); //平衡二叉树的高度
    } //算法结束
```

9.34　设二叉排序树的存储结构为：

```
typedef struct node
{ElemType  key;int size;struct node *lchild, *rchild, *parents;}node, *BiTree;
```

一个结点*x 的 size 域的值是以该结点为根的子树中结点的总数（包括*x 本身）。例如，下图中 x 所指结点的 size 值为 4。设树高为 h，试写一时间为 O(h)的算法 Rank(BiTree T, node *x),返回 x 所指结点在二叉排序树 T 的中序序列里的排序序号，即：求 x^结点是根为 T 的二叉排序树中第几个最小元素。例如，下图 x 所指结点是树 T 中第 11 个最小元素。



【题目分析】 因为 T 是二叉排序树，则可利用二叉排序树的性质，从根往下查找结点*x。若 T 的左子树为空，则其中序序号为 1，否则为 T->lchild->size+1。设 T 的中序序号为 r，其左子女 p 的中序序号和右子女 q 的中序序号分别为 r-p->rchild->size-1 和 r+q->lchild->size+1。

【算法 9.34】

```
int  Rank(tree T, node *x)
     // 在二叉排序树 T 上求结点 x 的中序序号
    {if(T->lchild) r=T->lchild->size+1; //根结点有左子树时的中序序号
     else  r=1;                         //根结点无左子树时中序序号为 1
     while(T)
       if(T->key>x->key)                //到左子树去查找
         {T=T->lchild;
          if(T)
             {if(T->rchild) r=r-T->rchild->size-1; else r=r-1}
         }
       else if(T->key<x->key)//到右子树去查找
          {T=T->rchild;
           if(T)
```

```
                {if(T->lchild)r=r+T->lchild->size+1; else r=r+1; }
              }
            else return (r); //返回*x 结点的中序序号
          return  (0);          //T 树上无 x 结点。
        }//结束算法 Rank
```

算法 2：本题的另一种解法是设 r 是以*x 为根的中序序号。初始时，若 x 的左子树为空，r=1;否则，r=x->lchild->size+1。利用结点的双亲域，上溯至根结点，即可求得*x 的中序序号。

```
int  Rank(tree T, node *x)//在二叉排序树 T 上，求结点*x 的中序序号
  {if(x->lchild)r=x->lchild->size+1; else r=1; //*x 的这个序号是暂时的
    p=x; //p 要上溯至根结点 T，求出*x 的中序序号
    while(p!=T)
      {if(p==p->parents->rchild)              //p 是其双亲的右子女，
         {if(p->parents->lchild==null) r++;   //p 结点的双亲排在 p 结点的前面
          else r=r+p->parent->lchild->size+1;//双亲及左子树均排在 p 前面
         }
       p=p->parents;
      }//while
    return   (r);
  }//Rank
```

9.35　已知某哈希表 HT 的装填因子小于 1,哈希函数 H(key)为关键字的第一个字母在字母表中的序号。

(1) 处理冲突的方法为线性探测再散列。编写按第一个字母的顺序输出哈希表中所有关键字的算法。

(2) 处理冲突的方法为链地址法。编写一个计算在等概率情况下查找不成功的平均查找长度的算法。

【题目分析】 本题未直接给出哈希表表长，但已给出装填因子小于 1，且哈希函数 H(k)为关键字第一个字母在字母表中的序号，字母‘A’的序号为 1，表长可设为 n(n≥27)，而链地址法中，表长 26。查找不成功是指碰到空指针为止(另一种观点是空指针不计算比较次数)。

【算法 9.35】

```
(1)void Print(rectype h[])//按关键字第一个字母在字母表中的顺序输出各关键字
    {int   i,j;
     for(i=1;i≤26; i++)     // 哈希地址 1 到 26
        {j=1; printf(“\n”);
         while(h[j]!=null)  // 设哈希表初始值为 null
            {if(ord(h[j])==i)// ord()取关键字第一字母在字母表中的序号
                    printf(“%s”, h[j]);
```

```
                j=(j+1)% n;
               }//while
            }//for
        }//end
```

(2)int ASLHash(rectype h[])//链地址解决冲突的哈希表查找不成功时平均查找长度

```
{int i,j; count=0;    //记查找不成功的总的次数
  LinkedList  p;
  for(i=1;i≤26; i++)
   {p=h[i]; j=1; //按我们约定，查找不成功指到空指针为止。
    while(p!=null){j++; p=p->next; }
    count+=j;
   }
 return  (count/26.0);
}
```

【注】值得指出，对用拉链法求查找失败时的平均查找长度有两种观点。其一，认为比较到空指针算失败。例如，若本题哈希地址 h[i](1≤i≤26)为空指针，则认为比较 1 次失败；若哈希地址 h[i](1≤i≤26)为非空指针，例如，h[i](1≤i≤26)链表中只有一个结点，则认为比较 2 次后失败，我们持这种观点。还有另一种观点：他们认为只有和关键字比较才计算比较次数，而和空指针比较不计算比较次数。照这种观点，上面两种情况失败时的比较次数分别为 0 和 1。

9.36　有一个 100*100 的稀疏矩阵，其中 1%的元素为非零元素，现要求用哈希表作存储结构。

（1）请设计一个哈希表

（2）请写一个对你所设计的哈希表中给定行值和列值存取矩阵元素的算法；并对算法所需时间和用一维数组(每个分量存放一个非零元素的行值、列值和元素值)作存储结构时存取元素的算法进行比较。

【题目分析】非零元素个数是 100，负载因子取 0.8，表长 125 左右，取表长 p 为 127，散列地址为 0 到 126。哈希函数用 H(k)=(3*i+2*j) % 127，i，j 为行值和列值。

【算法 9.36】

```
#define  m 127
 typedef struct {int  i, j; ElemType v;}triple;
 void CreatHT(triple H[m])//生成稀疏矩阵的哈希表，表中元素值初始化为 0
   {for(k=0; k<100; k++)
      {scanf(&i,&j,&val); //设元素值为整型
       h=(3*i+2*j)% m;     //计算哈希地址
       while(HT[h].v!=0)) h=(h+1) % m;     //线性探测哈希地址
       HT[h].i=i; HT[h].j=j; HT[h].v=val; //非零元素存入哈希表
  }//for  }//算法 CreatHT 结束
ElemType  Search(triple HT[m], int i, int j)
 //在哈希表中查找下标为 i，j 的非零元素，查找成功返回非零元素，否
```

则返回零值

```
{int  h=(3*i+2*j) % m;
 while((HT[h].i!=i || HT[h].j!=j) && HT[h].v!=0) h=(h+1)% m;
 return  (HT[h].v);
}//Search
```

第 10 章　排序

一、　基础知识题

10.1　基本概念：内排序，外排序，稳定排序，不稳定排序，顺串，败者树，最佳归并树。

【解答】

⑴内排序和外排序　若整个排序过程不需要访问外存便能完成，则称此类排序问题为**内部排序**（简称**内排序**）；反之，若参加排序的记录数量很大，整个序列的排序过程不可能在内存中完成，则称此类排序问题为**外部排序**（简称**外排序**）。内部排序适用于记录个数不多的文件，不需要访问外存，而外部排序适用于记录很多的大文件，整个排序过程需要在内外存之间多次交换数据才能得到排序的结果。

⑵稳定排序和不稳定排序　假设待排序记录中有关键字 $K_i=K_j$（$i\neq j$），且在排序前的序列中 R_i 领先于 R_j。经过排序后，R_i 与 R_j 的相对次序保持不变（即 R_i 仍领先于 R_j），则称这种排序方法是**稳定的**，否则称之为**不稳定的**。

⑶顺串　外部排序通常经过两个独立的阶段完成。第一阶段，根据内存大小，

每次把文件中一部分记录读入内存，用有效的内部排序方法（如快速排序、堆排序等）将其排成**有序段**，这有序段又称**顺串**或**归并段**。

⑷败者树 败者树是为提高外部排序的效率而采用的，是由参加比赛的 n 个元素作叶子结点而得到的完全二叉树。每个非叶（双亲）结点中存放的是两个子结点中的败者数据,而让胜者去参加更高一级的比赛。另外,还需增加一个结点,即结点 0,存放比赛的全局获胜者。

⑸最佳归并树 在外部排序的多路平衡归并的 k 叉树中，为了提高效率减少对外存的读写次数，按哈夫曼树构造的 k 叉树称最佳归并树。这棵树中只有度为 0 和度为 k 的结点。若用 m **表示**归并段个数，用 n_k 表示度为 k 的个数，若 (m-1)%(k-1)=0，则不需增加虚段，否则应附加 k-(m-1)%(k-1)-1 个虚段（即第一个 k 路归并使用(m-1)%(k-1)+1 个归并段）。

10.2 设待排序的关键字序列为(15， 21， 6， 30， 23， 6′， 20， 17)，试分别写出使用以下排序方法每趟排序后的结果。并说明做了多少次比较。

(1) 直接插入排序 (2) 希尔排序(增量为 5，2，1) (3) 起泡排序

(4) 快速排序 (5) 直接选择排序 (6) 锦标赛排序

(7) 堆排序 (8) 二路归并排序 (9) 基数排序

【解答】

(1) 直接插入排序

初始关键字序列： 15，21，6，30，23，6′，20，17
第一趟直接插入排序:【15，21】
第二趟直接插入排序:【6，15，21】
第三趟直接插入排序:【6，15，21，30】
第四趟直接插入排序:【6，15，21，23，30】
第五趟直接插入排序:【6，6′，15，21，23，30】
第六趟直接插入排序:【6，6′，15，20，21，23，30】
第七趟直接插入排序:【6，6′，15，17，20，21，23，30】

(2) 希尔排序(增量为 5，2，1)

初始关键字序列： 15，21，6，30，23，6′，20，17
第一趟希尔排序： 6′，20，6，30，23，15，21，17
第二趟希尔排序： 6′，15，6，17，21，20，23，30
第三趟希尔排序： 6′，6，15，17，20，21，23，30

(3) 起泡排序

初始关键字序列：15，21，6，30，23，6′，20，17
第一趟起泡排序：15，6，21，23，6′，20，17，30
第二趟起泡排序：6，15，21，6′，20，17，23，30
第三趟起泡排序：6，15，6′，20，17，21，23，30
第四趟起泡排序：6，6′，15，17，20，21，30，23
第五趟起泡排序：6，6′，15，17，20，21，30，23

(4) 快速排序

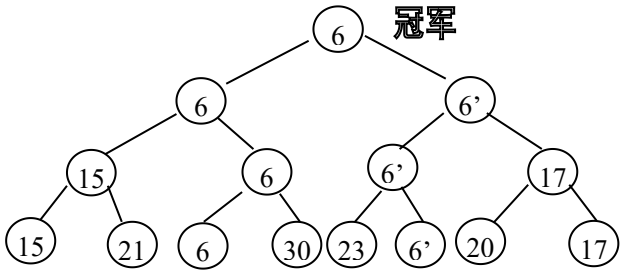
初始关键字序列：　15，21，6，30，23，6′，20，17
第一趟快速排序：　【6′，6】15【30，23，21，20，17】
第二趟快速排序：　6′，6，15【17，23，21，20】30
第三趟快速排序：　6′，6，15，17【23，21，20】30
第四趟快速排序：　6′，6，15，17,【20，21】23，30
第五趟快速排序：　6，6′，15，17，20，21，30，23

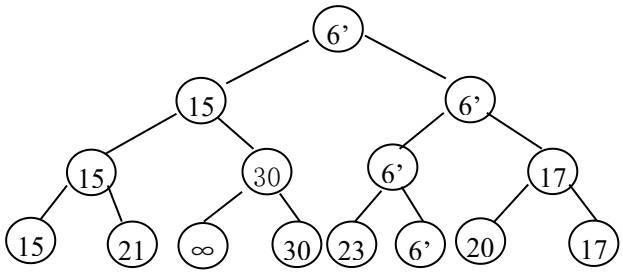
(5) 直接选择排序

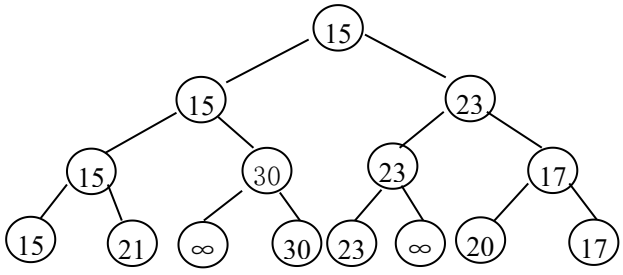
初始关键字序列：　15，21，6，30，23，6′，20，17
第一趟直接选择排序：6，21，15，30，23，6′，20，17
第二趟直接选择排序：6，6′，15，30，23，21，20，17
第三趟直接选择排序：6，6′，15，30，23，21，20，17
第四趟直接选择排序：6，6′，15，17，23，21，20，30
第五趟直接选择排序：6，6′，15，17，20，21，23，30
第六趟直接选择排序：6，6′，15，17，20，21，23，30
第七趟直接选择排序：6，6′，15，17，20，21，23，30

(6) 锦标赛排序

初始关键字序列：　15，21，6，30，23，6′，20，17







锦标赛排序的基本思想是：首先对 n 个待排序记录的关键字进行两两比较，从中选出⌈n/2⌉个较小者再两两比较，直到选出关键字最小的记录为止，此为一趟排序。我们将一趟选出的关键字最小的记录称为“冠军”，而“亚军”是从与“冠军”比较失败的记录中找出，具体做法为：输出“冠军”后，将（冠军）叶子结点关键字改为最大，继续进行锦标赛排序，直到选出关键字次小的记录为止，如此循环直到输出全部有序序列。上面给出了排在前三个的记录，详细过程略。

(7) 堆排序

初始关键字序列：15，21，6，30，23，6′，20，17
初始堆：　　6，17，6’，21，23，15，20，30
第一次调堆：　　6’，17，15，　21，23，30，20，【6】
第二次调堆：　　15，17，20，21，23，30，【6’，6】
第三次调堆：　　17，21，20，30，23，【15，6’，6】
第四次调堆：　　20，21，23，30，【17，15，6’，6】
第五次调堆：　　21，30，23，【20，17，15，6’，6】
第六次调堆：　　23，30，【21，20，17，15，6’，6】
第七次调堆：　　30，【23，21，20，17，15，6’，6】
堆排序结果调堆：【30，23，21，20，17，15，6’，6】

(8) 二路归并排序

初始关键字序列：　15，21，6，30，23，6′，20，17
二路归并排序结果：15，17，20，21，23，30，6，6’
final ↑　↑ first

(9) 基数排序

初始关键字序列：p→15→21→6→30→23→6′→20→17
第一次分配得到:
B[0]. f→30→20←B[0]. e
B[1]. f→21←B[1]. e
B[3]. f→23←B[3]. e
B[5]. f→15←B[5]. e
B[6]. f→6→6’←B[6]. e
B[7]. f→17←B[7]. e
第一次收集得到:
p→30→20→21→23→15→6→6’→17
第二次分配得到
B[0]. f→6→6’←B[0]. e
B[1]. f→15→17←B[1]. e
B[2]. f→20→21→23←B[5]. e
B[3]. f→30←B[3]. e
第二次收集得到
p→6→6’→15→17→20→21→23→30
基数排序结果：6，6′，15，17，20，21，23，30

10.3　在各种排序方法中，哪些是稳定的？哪些是不稳定的？并为每一种不稳定的排序方法举出一个不稳定的实例。

【解答】见下表：

排序方法	平均时间	最坏情况	辅助空间	稳定性	不稳定排序举例
直接插入排序	$O(n^2)$	$O(n^2)$	O(1)	稳定	
折半插入排序	$O(n^2)$	$O(n^2)$	O(1)	稳定	
二路插入排序	$O(n^2)$	$O(n^2)$	O(n)	稳定	
表插入排序	$O(n^2)$	$O(n^2)$	O(1)	稳定	
起泡排序	$O(n^2)$	$O(n^2)$	O(1)	稳定	
直接选择排序	$O(n^2)$	$O(n^2)$	O(1)	不稳定	2,2’,1
希尔排序	$O(n^{1.3})$	$O(n^{1.3})$	O(1)	不稳定	3,2,2’,1(d=2,d=1)
快速排序	$O(n\log_2 n)$	$O(n^2)$	$O(\log_2 n)$	不稳定	2,2’,1
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	O(1)	不稳定	2,1,1’（极大堆）
2-路归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	O(n)	稳定	
基数排序	O(d*(rd+n))	O(d*(rd+n))	O（rd）	稳定	

10.4　在执行某种排序算法的过程中出现了排序码朝着最终排序序列相反的方向移动，从而认为该排序算法是不稳定的，这种说法对吗？为什么？

【解答】这种说法不对。因为排序的不稳定性是指两个关键字值相同的元素的相对次序在排序前、后发生了变化，而题中叙述和排序中稳定性的定义无关，所以此说法不对。对 4，3，2，1 起泡排序就可否定本题结论。

10.5　在堆排序、快速排序和归并排序方法中：

(1)若只从存储空间考虑，则应首先选取哪种排序，其次选取哪种排序，最后选取哪种排序？

(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)若只从平均情况下排序最快考虑，则应选取哪种排序方法？

(4)若只从最坏情况下排序最快并且要节省内存考虑，则应选取哪种排序方法？

【解答】

(1)堆排序，快速排序，归并排序

(2)归并排序

(3)快速排序

(4)堆排序

10.6　设要求从大到小排序。问在什么情况下冒泡排序算法关键字交换的次数为最多。

【解答】对冒泡算法而言，初始序列为反序时交换次数最多。若要求从大到小排序，则表现为初始是上升序时关键字交换的次数为最多。

10.7　快速排序的最大递归深度是多少？最小递归深度是多少？

【解答】设待排序记录的个数为 n,则快速排序的最小递归深度为$\lfloor \log_2 n \rfloor+1$,最大递归深度 n。

10.8　我们知道，对于 n 个元素组成的顺序表进行快速排序时，所需进行的比较次数与这 n 个元素的初始排序有关。问：

(1) 当 n=7 时，在最好情况下需进行多少次比较？请说明理由。

(2) 当 n=7 时，给出一个最好情况的初始排序的实例。

(3) 当 n=7 时，在最坏情况下需进行多少次比较？请说明理由。

(4) 当 n=7 时，给出一个最坏情况的初始排序的实例。

【解答】

(1) 在最好情况下,每次划分能得到两个长度相等的子文件。假设文件的长度$n=2^k-1$,那么第一遍划分得到两个长度均为$\lfloor n/2 \rfloor$的子文件,第二遍划分得到 4 个长度均为$\lfloor n/4 \rfloor$的子文件,以此类推,总共进行 $k=\log_2(n+1)$遍划分,各子文件的长度均为 1,排序完毕。当 n=7 时,k=3,在最好情况下,第一遍需比较 6 次,第二遍分别对两个子文件（长度均为 3,k=2）进行排序,各需 2 次,共 10 次即可。

(2) 在最好情况下快速排序的原始序列实例：4,1,3,2,6,5,7。

(3) 在最坏情况下,若每次用来划分的记录的关键字具有最大(或最小)值,那么只能得到左(或右)子文件,其长度比原长度少 1。因此,若原文件中的记录按关键字递减次序排列,而要求排序后按递增次序排列时,快速排序的效率与冒泡排序相同,其时间复杂度为 $O(n^2)$。所以当 n=7 时,最坏情况下的比较次数为 21 次。

(4) 在最坏情况下快速排序的初始序列实例：7,6,5,4,3,2,1,要求按递增排序。

10.9　判断下面的每个结点序列是否表示一个堆，如果不是堆，请把它调整成堆。

(1)100，90，80，60，85，75，20，25，10，70，65，50

(2)100，70，50，20，90，75，60，25，10，85，65，80

【解答】

(1) 是堆

(2) 不是堆。　　调成大堆：　100,90,80,25,85,75,60,20,10,70,65,50

10.10　在多关键字排序时，LSD 和 MSD 两种方法的特点是什么？

【解答】

最高位优先（MSD）法：先对最高位关键字 K^0 进行排序,将序列分成若干子序列,每个子序列中的记录都具有相同的 K^0 值,然后,分别就每个子序列对关键字 K^1 进行排序,按 K^1 值不同再分成若干更小的子序列,……,依次重复,直至最后对最低位关键字排序完成,将所有子序列依次连接在一起,成为一个有序子序列。

最低位优先（LSD）法：先对最低位关键字 K^{d-1} 进行排序,然后对高一级关键字 K^{d-2} 进行排序,依次重复,直至对最高位关键字 K^0 排序后便成为一个有序序列。进行排序时,不必分成子序列,对每个关键字都是整个序列参加排序,但对 K^i (0<=i<d-1)排序时,只能用稳定的排序方法。另一方面,按 LSD 进行排序时,可以不通过关键字比较实现排序,而是通过若干次“分配”和“收集”来实现排序。

10.11　给出如下关键字序列 321，156，57，46，28，7，331，33，34，63 试按链式基数排序方法，列出一趟分配和收集的过程。

【解答】

按 LSD 法　→321→156→57→46→28→7→331→33→34→63

分配	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
		321		33	34		156	57	28	
		331		63			46	7		

收集　→321→331→33→63→34→156→46→57→7→28

10.12　奇偶交换排序如下所述：对于初始序列 A[1]，A[2]，…，A[n]，第一趟对所有奇数 i(1<=i<n)，将 A[i]和 A[i+1]进行比较，若 A[i]>A[i+1]，则将两者交换；第二趟对所有偶数 i(2<=i<n)，将 A[i]和 A[i+1]进行比较，若 A[i]>A[i+1]，则将两者交换；第三趟对所有奇数 i(1<=i<n)；第四趟对所有偶数 i(2<=i<n)，…，依次类推直至到整个序列有序为止。

(1) 分析这种排序方法的结束条件。

(2) 写出用这种排序方法对 35，70，33，65，24，21，33 进行排序时，每一趟的结果。

【解答】

(1)排序结束条件为，连续的第奇数趟排序和第偶数趟排序都没有交换。

(2)

第一趟奇数：35,70,33,65,21,24,33

第二趟偶数：35,33,70,21,65,24,33

第三趟奇数：33,35,21,70,24,65,33

第四趟偶数：33,21,35,24,70,33,65

第五趟奇数：21,33,24,35,33,70,65

第六趟偶数：21,24,33,33,35,65,70

第七趟奇数：21,24,33,33,35,65,70(无交换)

第八趟偶数：21,24,33,33,35,65,70(无交换) 结束

10.13　设某文件经内排序后得到 100 个初始归并段(初始顺串)，若使用多路归并排序算法，并要求三趟归并完成排序，问归并路数最少为多少？

【解答】 设归并路数为 k,归并趟数为 s,则 $s=\lceil \log_k 100 \rceil$,因$\lceil \log_k 100 \rceil=3$,且 k 为整数，故 k=5,即最少 5 路归并可以完成排序。

10.14　证明：置换-选择排序法产生的初始归并段的长度至少为 m。(m 是所用缓冲区的长度)。

【证明】 由置换选择排序思想,第一个归并段中第一个元素是缓冲区中最小的元素,以后每选一个元素都不应小于前一个选出的元素,故当产生第一个归并段时（即初始归并段）,缓冲区中 m 个元素中除最小元素之外,其他 m-1 个元素均大于第一个选出的元素,即使以后读入元素均小于输出元素时,初始归并段中也至少能有原有的 m 个元素。证毕。

10.15　设有 11 个长度(即包含记录的个数)不同的初始归并段,它们所包含的记录个数分别为 25，40，16，38，77，64，53，88，9，48，98。试根据它们做 4 路平衡归并，要求：

(1)指出总的归并趟数；

(2)构造最佳归并树；

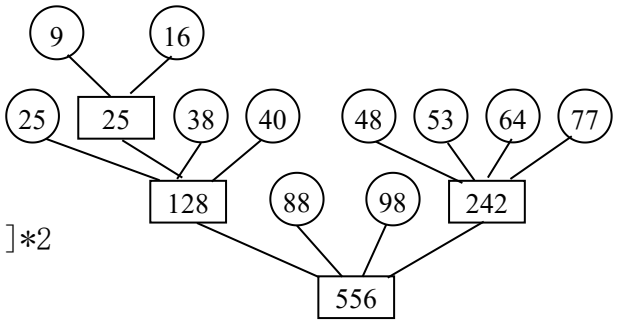
(3)根据最佳归并树计算每一趟及总的读记录数。

【解答】

因为(11-1)%(4-1)=1,所以加“虚段”,第一次由两个段合并。

(1)三趟归并

(2)最佳归并树如图



(3)设每次读写一个记录

第一趟 50 次读写

总的读写次数：2052

[(9+16)*3+(25+38+40)*2+

(48+53+64+77)*2+(88+98)]*2

10.16　对输入文件(101，51，19，61，3，71，31，17，19，100，55，20，9，30，50，6，90)，当 k=6 时，使用置换-选择算法，写出建立的初始败者树及生成的初始归并段。

【解答】

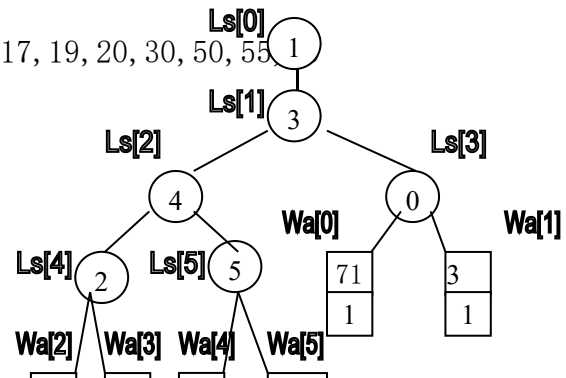
初始败者树

初　始　归　并　段　：

R_1:3,19,31,51,61,71,100,101

R_2:9,17,19,20,30,50,55

R_3:6



10.17　选择题：下面给出的四种排序方法中，排序过程中的比较次数与排序方法无关的是。

A. 选择排序法　　B. 插入排序法　　C. 快速排序法　　D. 堆排序法

【解答】A

10.18　选择题：一个排序算法的时间复杂度与以下哪项有关。

A. 排序算法的稳定性　　B. 所需比较关键字的次数

C. 所采用的存诸结构　　D. 所需辅助存诸空间的大小

【解答】B

二、算法设计题

10.19　请编写一个算法，在基于单链表表示的关键字序列上进行简单选择排序。

【算法 10.19】

```
void LinkedListSelectSort(pointer head);
  //本算法一趟找出一个关键字最小的结点,其数据和当前结点进行交换
 {p=head->next;
  while(p)
    {q=p->next;  r=p;     //设 r 是指向关键字最小的结点的指针
     while(q!=null)
       {if(q->data<r->data) r=q;
        q=q->next;
       }
     if (r!=p) r->data<-->p->data;
     p=p->next;              //选下一个最小元素
    }
```

【算法讨论】本算法只交换两个结点的数据，若要交换结点，则须记下当前结点和最小结点的前驱指针

10.20　设单链表头结点指针为 L，结点数据为整型。试写出对链表 L 按“直接插入方法”排序的算法。

【算法 10.20】

```
void LinkInserSort(LinkedList L)
  //本算法对单链表L按“直接插入方法”进行排序
    { p=L->next->next;     //链表至少一个结点,p初始指向链表中第二结
  点（若存在）
     L->next->next=null; //初始假定第一个记录有序
     while(p!=null)
       {q=p->next;       //q指向p的后继结点}
        s=L;
        while(s->next!=null && s->next->data<p->data)
           s=s->next;    //向后找插入位置
        p->next=s->next;
        s->next=p;       //插入结点
        p=q;             //恢复p指向当前结点
       }
     }
```

10.21　试设计一个双向冒泡排序算法，即在排序过程中交替改变扫描方向。

【算法10.21】

```
  void BubbleSort2(int a[],int n) //相邻两趟向相反方向起泡的冒泡排序
  算法
    {change=1;low=0;high=n-1;     //冒泡的上下界
      while(low<high && change)
        {change=0;                //设不发生交换
         for(i=low;i<high;i++)    //气泡上浮,大元素下沉（向右）
            if(a[i]>a[i+1])
              {a[i]<-->a[i+1];change=1;} //有交换,修改标志change
          high--; //修改上界
          for(i=high;i>low;i--)           //气泡下沉,小元素上浮（向
    左）
            if(a[i]<a[i-1]){a[i]<-->a[i-1];change=1;}
          low++; //修改下界
       }//while  }//BubbleSort2
```

10.22　写出快速排序的非递归算法。

【算法10.22】

```
  void  QuickSort(rectype  r[n+1]; int n)
    {// 对r[1..n]进行快速排序的非递归算法
     typedef struct{int low,high; }node
    node s[n+1]; //栈,容量足够大
    int quickpass(rectype r[],int,int);  // 函数声明
    int top=1; s[top].low=1; s[top].high=n;
    while(top>0)
     {ss=s[top].low; tt=s[top].high;  top--;
      if(ss<tt)
```

```
        {k=quickpass(r,ss,tt);
         if(k-ss>1) {s[++top].low=ss; s[top].high=k-1;}
         if(tt-k>1) {s[++top].low=k+1; s[top].high=tt;}
        }
      } // 算法结束
    int quickpass(rectype r[];int s,t)
      {i=s; j=t; rp=r[i]; x=r[i].key;
       while （i<j）
        {while(i<j && x<=r[j].key) j--;
          if(i<j) r[i++]=r[j];
          while(i<j && x>=r[j].key) i++;
          if(i<j) r[j--]=r[i];;
         }
        r[i]=rp;
        return (i);
       } // 一次划分算法结束
```

[算法讨论]可对以上算法进行两点改进：一是在一次划分后,先处理较短部分,较长的子序列进栈；二是用“三者取中法”改善快速排序在最坏情况下的性能。下面是部分语句片段：

```
  int top=1; s[top].low=1; s[top].high=n;
  ss=s[top].low; tt=s[top].high;  top--; flag=true;
  while(flag || top>0)
    {k=quickpass(r,ss,tt);
     if(k-ss>tt-k)    // 一趟排序后分割成左右两部分
       {if(k-ss>1)    // 左部子序列长度大于右部,左部进栈
          {s[++top].low=ss; s[top].high=k-1; }
        if(tt-k>1) ss=k+1;     // 右部短的直接处理
            else flag=false;  // 右部处理完,需退栈
        }
     else if(tt-k>1) //右部子序列长度大于左部,右部进栈
           {s[++top].low=k+1; s[top].high=tt; }
            if(k-ss>1) tt=k-1  // 左部短的直接处理
               else flag=false // 左部处理完,需退栈
           }
      if(!flag && top>0)         //退栈
        {ss=s[top].low; tt=s[top].high; top--; flag=true;}
    }  // end of while(flag || top>0)
  } // 算法结束
  int quickpass(rectype r[];int s,t)
  // 用“三者取中法”进行快速排序的一次划分
    { int  i=s, j=t, mid=(s+t)/2;
      rectype tmp;
      if(r[i].key>r[mid].key) {tmp=r[i];r[i]=r[mid];r[mid]=tmp }
      if(r[mid].key>r[j].key)
```

```
        {tmp=r[j];r[j]=r[mid];
         if(tmp>r[i]) r[mid]=tmp; else {r[mid]=r[i];r[i]=tmp }
        }
     {tmp=r[i];r[i]=r[mid];r[mid]=tmp }
       // 三者取中：最佳 2 次比较 3 次赋值；最差 3 次比较 10 次赋值
rp=r[i]; x=r[i].key;
   while （i<j）
    {while(i<j && x<=r[j].key) j--;
      if(i<j) r[i++]=r[j];
      while(i<j && x>=r[j].key) i++;
      if(i<j) r[j--]=r[i];;
     ]
    r[i]=rp;
return  (i);
  } // 一次划分算法结束
```

10.23　假设由 1000 个关键字为小于 10000 的整数的记录序列，请设计一种排序方法，要求以尽可能少的比较次数和移动次数实现排序，并按你的设计编写算法。

【题目分析】设关键字小于 10000 的整数的记录序列存于数组中，再设容量为 10000 的临时整数数组，按整数的大小直接放入下标为该整数的数组单元中，然后对该数组进行整理存回原容量为 1000 的数组中。

【算法 10.23】

```
void intsort(int R[],int n)
   {//关键字小于 10000 的 1000 个整数存于数组 R 中，本算法对整数进行排序
    int R1[10000]={0};    //初始化为 0
    for(i=0;i<1000;i++)
       R1[R[i]]=R[i];
    for(i=0,k=0;i<10000;i++)
       if(R1[i]!=0)
          R[k++]=R1[i];
   }
```

10.24　荷兰国旗问题：设有一个仅有红、白、蓝 三种颜色的条块组成的条块序列。编写一个时间复杂度为 O(n)的算法，使得这些条块按红、白、蓝的顺序排好，即排成荷兰国旗图案

【题目分析】设用整型数组 R 表示荷兰国旗,元素值 1、2 和 3 分别表示红、白和篮色。再设整型变量 i,j 和 k,排序结束后 R[1..i-1]表示红色,R[i..j-1]表示白色,R[j..n]表示篮色。i,j 和 k 的初始值分别是 1，1 和 n。

【算法 10.24】

```
void DutchFlag(int R[],int n)
//对红、白、篮三种颜色的条块,经排序形成荷兰国旗
  {int i=1,j=1,k=n;    //指针初始化,j 到 k 是待排序元素
```

```
    while(j<=k)
    if(r[j]==1)          //红色
       {r[i]<-->r[j]; i++;j++; }
    else if(r[j]==2) j++;           //白色
         else {r[j]<-->r[k]; k--;} //兰色
   }
```

[算法讨论]象将元素值为正数、负数和零排序成前面都是负数,接着是零,最后是正数的排序,以及字母字符、数字字符和其它字符的排序等,都属于这类荷兰国旗问题。排序后,红、白和蓝色的元素个数分别为 i-1,j-i,n-j+1。

10.25　已知记录序列 a[1..n]中的关键字各不相同，可按如下所述实现计数排序：另设数组 c[1..n]，对每个记录 a[i]，统计序列中关键字比它小的记录个数存于 c[i]，则 c[i]=0 的记录必为关键字最小的记录，然后依 c[i]值的大小对 a 中记录进行重新排列，编写算法实现上述排序方法。

【算法 10.25】

```
    void  CountSort(rectype r[],int n)
        {//对 r[1..n]进行计数排序
         c[1..n] =0;        //c 数组初始化，元素值指其在 r 中的位置
         for(i=1;i<n;i++)  //一趟比较选出大小，给数组 c 赋值
            for(j=i+1;j<=n;j++)
              if(r[i].key>r[j].key)
                  c[i]++; else c[j]++;
          i=1;
          while(i<n)     //若 c[i]+1= i，则 r[i] 正好是第 i 个元素;否则，需要调整
            {if(c[i]+1!=i)
               {j=i; rc=r[i];
               while(c[j]+1!=i)
                  {k=c[j]+1;rt=r[k];  //暂时保存 r[k]/
                   r[k]=rc;  j=c[k];  //取下一 j 值
                    c[k]=k-1; //第 k 个已排好
                    rc=rt
                  }
               r[i]=rc; c[i]=i-1;  //完成了一个小循环，第 i 个已排好
              }//if
             i=i+1
           }// while
```

上述调整也可用如下逻辑简单但效率低下的算法：

```
      c[1..n]= c[1..n]+1 {c 数组元素值指其在 r 中的位置。
       while(i<n)
          { while(c[i]!=i)
                {j=c[i];  c[i]←→c[j];  r[i]←→r[j]}
            i=i+1;
          }
```

10.26　若待排序序列用单链表存储，试给出其快速排序算法。

[题目分析]快速排序的思想是以第一个元素作“枢轴”,通过一趟的比较,将枢轴元素放在其排序的最终位置,使它左面的元素都小于等于它,而它右面的元素都大于等于它,从而再对其左右两部分递归进行快速排序。在链表中实现快速排序也必须使用这一原则。

【算法 10.26】

```
    void LinkQuickSort(LinkedList start,end)
      //对单链表 start 进行快速排序,end 是链表的尾指针,初始调用时为 null
    {LinkedList start1,end1,end2,p;
     int flag=0;    //tag 是是否结束排序的标志
     If(start==null || start==end) return;    //空表或只有一个结点
     start1=end1=start;  //start1 和 end1 是右一半链表头结点的前驱和尾结点的指针
     if(end1!=end) flag=1;
     p=start->next;  //p 为工作指针
     while(flag)      //进行一趟快速排序
       {if(p->data<start->data)  //结点插入前一半
          {end1->next=p->next;    //保留后继结点
           if(p==end) flag=0;     //一趟快速排序结束
           if(start==end1)
              end0=p;//end0 是遍历遇到的第一个小于枢轴的结点,将为前半的尾结点
           p->next=start; start=p;//修改左半部链表头指针
           p=end1->next;          //恢复当前待处理结点
           }
        else //处理右半部链表
           {if(p==end) flag=0;     //已到链表尾
            end1->next=p; end1=p; ;//end1 和 p 是前驱和后继关系
            p=p->next;
         }//else }//while
       LinkQuickSort(start,end0);        //对枢轴元素最终位置前的单链表快速排序
       LinkQuickSort(start1->next,end1);//对枢轴元素最终位置后的单链表快速排序
    }//LinkQuickSort
```

10.27　在数组 A[0..n-1]中存放有 n 个不同的整数，其值均在 1 到 n 之间。写出一个函数或过程，将 A 中的 n 个数从大到小排序后存入 B[0..n-1]数组中，要求算法的时间复杂度为 O(n)。

【题目分析】因为 n 个值不同且大小在 1 到 n 之间的整数,要求逆序放入另一数组,只要逐个取出放到适当位置即可。即值为 i(1<=i<=n)的元素就是数组下标为 n-i 的元素。

【算法 10.27】

```
void ReArrange(int A[], int B[],int n)
  {for(i=0;i<n;i++)
     B[n-A[i]]=A[i];
  }
```

第 11 章　文件

一、基础知识题

11.1 名词解释：索引文件，索引顺序文件，ISAM 文件，VSAM 文件，散列文件，倒排文件。

【解答】先介绍文件的概念：文件是由大量性质相同的记录组成的集合,按记录类型不同可分为操作系统文件和数据库文件。

文件的基本组织方式有顺序组织、索引组织、散列组织和链组织。文件的存

储结构可以采用将基本组织结合的方法,常用的结构有顺序结构、索引结构、散列结构。

（1）顺序结构,相应文件为**顺序文件**,其记录按存入文件的先后次序顺序存放。顺序文件本质上就是顺序表。若逻辑上相邻的两个记录在存储位置上相邻,则为**连续文**件；若记录之间以指针相链接,则称为**串联文件**。顺序文件只能顺序存取,要更新某个记录,必须复制整个文件。顺序文件连续存取的速度快,主要适用于顺序存取,批量修改的情况。

（2） 带索引的结构,相应文件为**索引文件**。索引文件包括索引表和数据表,索引表中的索引项包括数据表中数据的关键字和相应地址,索引表有序,其物理顺序体现了文件的逻辑次序,实现了文件的线性结构。索引文件只能是磁盘文件,既能顺序存取,又能随机存取。

（3） 散列结构,也称计算寻址结构,相应文件称为**散列文件**,其记录是根据关键字值经散列函数计算确定其地址,存取速度快,不需索引,节省存储空间。不能顺序存取,只能随机存取。

其它文件均由以上文件派生而得。

文件采用何种存储结构应综合考虑各种因素,如：存储介质类型、记录的类型、大小和关键字的数目以及对文件作何种操作。

索引文件：在主文件外,再建立索引表指示关键字及其物理记录的地址间一一对应关系。这种由索引表和主文件一起构成的文件称为索引文件。索引表依关键字有序。主文件若按关键字有序称为**索引顺序文件**,否则称为**索引非顺序文件**（通常简称索引文件）。索引顺序文件因主文件有序,一般用稀疏索引,占用空间较少。

ISAM 文件：ISAM 是专为磁盘存取设计的文件组织方式。即使主文件关键字有序,但因磁盘是以盘组、柱面和磁道（盘面）三级地址存取的设备,因此通常对磁盘上的数据文件建立盘组、柱面和磁道（盘面）三级索引。在 ISAM 文件上检索记录时,先从主索引（柱面索引的索引）找到相应柱面索引。再从柱面索引找到记录所在柱面的磁道索引,最后从磁道索引找到记录所在磁道的第一个记录的位置,由此出发在该磁道上进行顺序查找直到查到为止；反之,若找遍该磁道而未找到所查记录,则文件中无此记录。

VSAM 文件：VSAM 文件采用 B+树动态索引结构,文件只有控制区间和控制区域等逻辑存储单位,与外存储器中柱面、磁道等具体存储单位没有必然联系。VSAM 文件结构包括索引集、顺序集和数据集三部分,记录存于数据集中,顺序集和索引集构成 B+树,作为文件的索引部分可实现顺链查找和从根结点开始的随机查找。

散列文件：散列文件也称直接存取文件,根据关键字的散列函数值和处理冲突的方法,将记录散列到外存上。这种文件组织只适用于像磁盘那样的直接存取设备,其优点是文件随机存放,记录不必排序,插入、删除方便,存取速度快,无需索引区,节省存储空间。缺点是散列文件不能顺序存取,且只限于简单查询。经多次插入、删除后,文件结构变得不合理,需重组文件,这很费时。

倒排文件：倒排文件是一种多关键字的文件,主数据文件按关键字顺序构成串联文件,并建立主关键字索引。对次关键字也建立索引,该索引称为倒排表。倒排表包括两项,一项是次关键字,另一项是具有同一次关键字值的记录的物理记录号（若数据文件非串联文件,而是索引顺序文件—如 ISAM,则倒排表中存放记录的主关键字而不是物理记录号）。倒排表作索引的优点是索引记录快,缺点是维护困难。在同一索引表中,不同的关键字其记录数不同,各倒排表的长度不同,同

一倒排表中各项长度也不相等。

11.2 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？

【解答】文件的逻辑结构是用户或程序员能够看见的数据组织形式，是用户对数据的表示和存取方式。文件的各记录间也存在逻辑关系，可以把文件看作一种线性结构。即记录间满足唯一直接前驱和唯一直接后继的关系。

文件的物理结构是数据的物理表示和组织。文件的逻辑结构着眼于用户使用方便，而物理结构需考虑节约存储空间和减少存取时间。存储记录的方式依据实际需要及设备的特性的差异而不同：一个物理记录可以存放一条或多条逻辑记录；多个物理记录也可以表示一个逻辑记录。

11.3 索引顺序存取方法（ISAM）中，主文件已按关键字排序，为何还需要主关键字索引？

【解答】

ISAM 是专为磁盘存取设计的文件组织方式。即使主文件关键字有序,但因磁盘是以盘组、柱面和磁道（盘面）三级地址存取的设备,因此通常对磁盘上的数据文件建立盘组、柱面和磁道（盘面）三级索引。在 ISAM 文件上检索记录时,先从主索引（柱面索引的索引）找到相应柱面索引。再从柱面索引找到记录所在柱面的磁道索引,最后从磁道索引找到记录所在磁道的第一个记录的位置,由此出发在该磁道上进行顺序查找直到查到为止；反之,若找遍该磁道而未找到所查记录,则文件中无此记录。

11.4 分析 ISAM 文件和 VSAM 文件的应用场合、优缺点等。

【解答】ISAM 是一种专为磁盘存取设计的文件组织形式,采用静态索引结构,对磁盘上的数据文件建立盘组、柱面、磁道三级索引。ISAM 文件中记录按关键字顺序存放,插入记录时需移动记录并将同一磁道上最后的一个记录移至溢出区,同时修改磁道索引项,删除记录只需在存储位置作标记,不需移动记录和修改指针。经过多次插入和删除记录后,文件结构变得不合理,需周期整理 ISAM 文件。

VSAM 文件采用 B+树动态索引结构,文件只有控制区间和控制区域等逻辑存储单位,与外存储器中柱面、磁道等具体存储单位没有必然联系。VSAM 文件结构包括索引集、顺序集和数据集三部分,记录存于数据集中,顺序集和索引集构成 B+树,作为文件的索引部分可实现顺链查找和从根结点开始的随机查找。

与 ISAM 文件相比,VSAM 文件有如下优点：动态分配和释放存储空间,不需对文件进行重组；能保持较高的查找效率,且查找先后插入记录所需时间相同。因此,基于 B+树的 VSAM 文件通常作为大型索引顺序文件的标准组织。

11.5 一个 ISAM 文件除了主索引外，还包括哪两级索引？

【解答】ISAM 文件有三级索引：磁盘组、柱面和磁盘,柱面索引存放在某个柱面上,若柱面索引较大,占多个磁道时,可建立柱面索引的索引—主索引。故本题中所指的两级索引是盘组和磁道。

11.6 为什么在倒排文件组织中，实际记录中的关键字域可删除以节约空间？而在多重表结构中这样做为什么要牺牲性能？

【解答】因倒排文件组织中,倒排表有关键字值及同一关键字值的记录的所有物理记录号,可方便地查询具有同一关键字值的所有记录；而多重表文件中次关键字索引结构不同,删除关键字域后查询性能受到影响。

11.7 简单比较文件的多重表和倒排表组织方式各自特点。

【解答】多重表文件是把索引与链接结合而形成的组织方式。记录按主关键字顺序构成一个串联文件,建立主关键字的索引（主索引）。对每一次关键字建立次关键字索引,具有同一关键字的记录构成一个链表。主索引为非稠密索引,次索引为稠密索引,每个索引项包括次关键字,头指针和链表长度。多重表文件易于编程,也易于插入,但删除繁锁。需在各次关键字链表中删除。

倒排文件是一种多关键字的文件,主数据文件按关键字顺序构成串联文件,并建立主关键字索引。对次关键字也建立索引,该索引称为倒排表。倒排表包括两项,一项是次关键字,另一项是具有同一次关键字值的记录的物理记录号（若数据文件非串联文件,而是索引顺序文件—如 ISAM,则倒排表中存放记录的主关键字而不是物理记录号）。倒排表作索引的优点是索引记录快,缺点是维护困难。在同一索引表中,不同的关键字其记录数不同,各倒排表的长度不同,同一倒排表中各项长度也不相等。

11.8 组织待检索文件的倒排表的优点是什么？

【解答】倒排表作索引的优点是索引记录快,因为从次关键字值直接找到各相关记录的物理记录号,倒排因此而得名（因通常的查询是从关键字查到记录）。在插入和删除记录时,倒排表随之修改,倒排表中具有相同次关键字的记录号是有序的。

11.9 为什么文件的倒排表比多重表组织方式节省空间？

【解答】排表有两项,一是次关键字值,二是具有相同次关键字值的物理记录号,这些记录号有序且顺序存储,不使用多重表中的指针链接,因而节省了空间。

11.10 试比较顺序文件，索引非顺序文件，索引顺序文件，散列文件的存储代价，检索，插入，删除记录时的优点和缺点。

【解答】（1）顺序文件只能顺序查找,优点是批量检索速度快,不适于单个记录的检索。顺序文件不能象顺序表那样插入、删除和修改,因文件中的记录不能象向量空间中的元素那样“移动”,只能通过复制整个文件实现上述操作。

（2）索引非顺序文件适合随机存取,不适合顺序存取,因主关键字未排序,若顺序存取会引起磁头频繁移动。索引顺序文件是最常用的文件组织,因主文件有序,既可顺序存取也可随机存取。索引非顺序文件是稠密索引,可以“预查找”,索引顺序文件是稀疏索引,不能“预查找”,但由于索引占空间较少,管理要求低,提高了索引的查找速度。

（3）散列文件也称直接存取文件,根据关键字的散列函数值和处理冲突的方法,将记录散列到外存上。这种文件组织只适用于像磁盘那样的直接存取设备,其优点是文件随机存放,记录不必排序,插入、删除方便,存取速度快,无需索引区,节省存储空间。缺点是散列文件不能顺序存取,且只限于简单查询。经多次插入、删除后,文件结构不合理,需重组文件,这很费时。

11.11 某一文件有 15 个记录，关键字分别为 285，070，923，597，512，262，015，076，157，208，337，613，117，390，362。桶的容量 m=3，桶数 b=7,用除留余数法构造哈希函数 H(key)=key % 7，试构造其散列文件。