第1部分　概述

算法的概念

算法（**Algorithm**）：是对特定问题求解步骤的一种描述，它是指令（规则）的有限序列，其中每一条指令表示一个或多个操作。

- 算法是在有限步骤内求解某一问题所使用的一组定义明确的规则。
- 通俗点说,就是计算机解题的过程。
- 在这个过程中，无论是形成解题思路还是编写程序,都是在实施某种算法。前者是推理实现的算法，后者是操作实现的算法。

递归技术 —— 最常用的算法设计思想，体现于许多优秀算法之中
分治法 —— 分而制之的算法思想，体现了一分为二的哲学思想
模拟法 —— 用计算机模拟实际场景，经常用于与概率有关的问题
贪心算法 —— 采用贪心策略的算法设计
状态空间搜索法 —— 被称为“万能算法”的算法设计策略
随机算法 —— 利用随机选择自适应地决定优先搜索的方向
动态规划 —— 常用的最优化问题解决方法

算法的分析

算法中基本操作重复执行的次数是问题规模n的某个函数f（n），算法的时间度量记作：

T（n）= O（f（n））

它表示随问题规模n的增大，算法执行时间的增长率和f（n）的增长率相同。

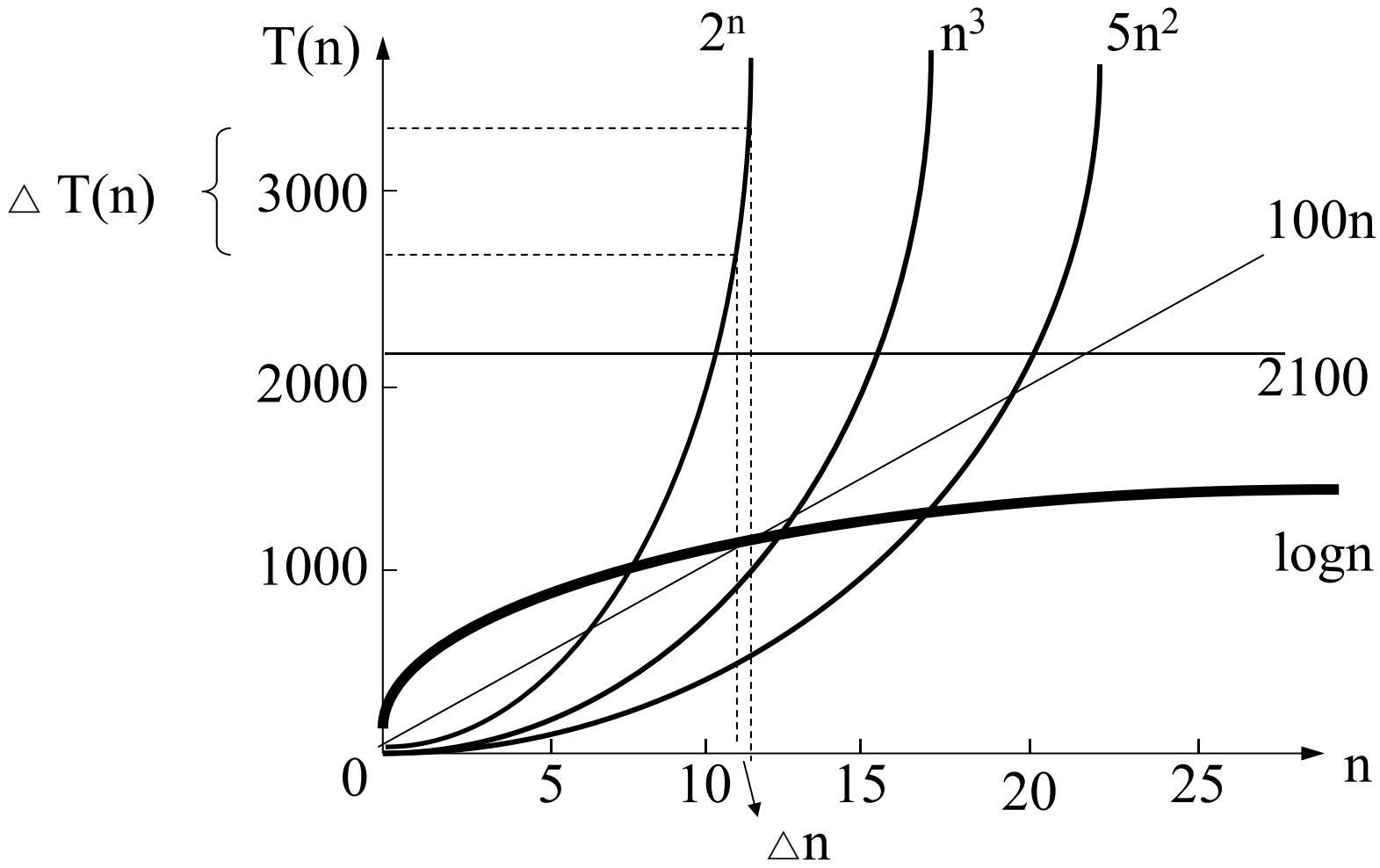
渐近空间复杂度（空间复杂度）S（n）

运算法则：

设：$T_1(n)=O(f(n))$，$T_2(n)=O(g(n))$

加法规则：$T_1(n)+T_2(n) = O(\max\{f(n), g(n)\})$

乘法规则：$T_1(n) \cdot T_2(n) = O(f(n) \cdot g(n))$



程序运行时间比较　T（n）= O（f（n））

时间复杂度举例：

①s = 0 ;

→ f(n) = 1; T1(n) = O(f(n)) = O(1)　　*常量阶*

②for (i=1 ; i <= n ; ++i) { ++x; s += x; }

→ f(n) = 3n+1; T2(n) = O(f(n)) = O(n)　　*线性阶*

③for (i=1; i<=n ; ++i)

for(j=1 ; j <=n ; ++j) { ++x ; s += x; }

→ f(n) = 3n2+2n+1; T3(n) = O(f(n)) = O(n2)　　*平方阶*

④for (i=1; i<=n ; ++i)

for (j=1 ; j <=n ; ++j)

{ c[i][j] = 0;

for (k=1 ; k <= n; ++k)

c[i][j] += a[i][k] * b[k][j] ; }

→ f(n) = 2n3+3n2+2n+1; T4(n) = O(f(n)) = O(n3)　　*立方阶*

⑤ int count = 1;

while (count < n) count = count * 2;

→ $2^x=n$, $f(n)=\log_2 n$　　*对数阶*

举例：

```
Long  fact ( int  n)
{   if ( n==0 ) || ( n ==1 )
        return( 1 );
    else
        return( n * fact( n – 1 ) );
}
```

$$f(n)=\begin{cases} C & \text{当 } n=0,\ n=1 \\ G+f(n-1) & \text{当 } n>1 \end{cases}$$

$$\begin{aligned} f(n) &= G_1+f(n-1) \\ f(n-1) &= G_2+f(n-2) \\ f(n-2) &= G_3+f(n-3) \\ &\cdots\cdots \\ f(2) &= G_{n-1}+f(1) \\ +\ f(1) &= C \\ \hline f(n) &= G_1+G_2+G_3+\cdots\cdots+G_{n-1}+C \end{aligned}$$

$\Rightarrow$

$$f(n)=nG'$$

$$\therefore T(n)=O(f(n))=O(n)$$

$$\log_2 n<n^{1/2}+\log_2 n<n<n\log_2 n<n^2+\log_2 n<n^3<n-n^3+7n^5<2^{n/2}<(3/2)^n<n!$$

一般地：常量阶 < 对数阶 < 线性阶 < 指数阶 < 幂次阶 < 阶乘阶

提示： 分析算法复杂度时要注意n个结点完全二叉树的高度的应用

线性表

（一）线性表的定义和基本操作

（二）线性表的实现

1、顺序存储结构

2、链式存储结构——线性链表及其操作

3、线性表的应用

栈、队列和数组

（一）栈和队列的基本概念

（二）栈和队列的顺序存储结构

（三）栈和队列的链式存储结构

（四）栈和队列的应用

（五）特殊矩阵的压缩存储——三角/对称阵，带状、稀疏矩阵

类型说明：

```
Struct Node {
    elementtype data;
    Struct Node *next;  }
Typedef struct node * NODE;
```

有关线性表的基本操作：

- 结点的插入和删除
- 链表建立（正序/反序）
- 线性表的遍历
- 有序线性表（数组/链表）
- 逆序重排（数组/链表）
- 线性链表，求倒数第K个数
- 线性链表，求中位数
- 数组按照升序排列，给定一个数M，求数组中的两个元素，其合等于M

2-1

设m×n阶稀疏矩阵A有t个非零元素，其三元组表表示如下：
LTMA[1:(t+1)，1:3]，
试问：非零元素的个数t达到什么时候时用LTMA表示A才有意义？

解答：
A的最后一个元素是a_{mn}，其余元素的下标积必小于mn
即有：3(t+1) ≤mn
因而：当t≤mn/3-1时，用LTMA表示A才有意义。

2-2

假设按低下标优先存储整数数组A(-3:8,3:5,-4:0,0:7)时，
第一个元素的字节存储地址是100，每个整数占4个字节。
试问：A(0,4,-2,5)的存储地址是什么？

坐标平移：
A(-3:8,3:5,-4:0,0:7)对应规范化的数组为：A(0:11,0:2,0:4,0:7)
元素A(0,4,-2,5) 相当于 A(3,1,2,5)
所以有：
L(3,1,2,5)=100+4(3x2x4x7+1x4x7+2x7+5)=1164

2-3

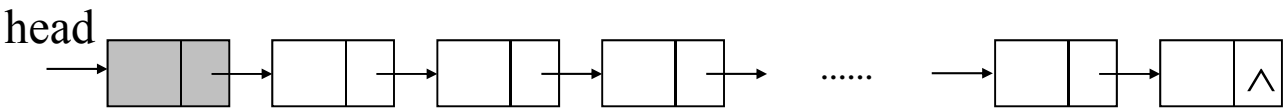
已知由n个整数构成序列的数据分布为先下降再上升，即一开始数据是严格递减的，后来数据是严格递增的，设计尽可能高效算法，找到序列中的最小值。

```
int FindMin(int Arr[], int  Left , int Right)
 {
    int Mid,Result;
    while (Left < Right)
    {   Mid=(Left+Right)/2;
        if (Arr[Mid] < Arr[Mid-1] && Arr[Mid] < Arr[Mid+1])
            return Mid;
        if (Arr[Mid] < Arr[Mid-1] && Arr[Mid] > Arr[Mid+1])
            Left = Mid+1;
        if (Arr[Mid] > Arr[Mid-1] && Arr[Mid] < Arr[Mid+1])
            Right = Mid - 1;
     }
    if (Left == Right)          Result = Left;
    if (Left < Right)           Result = Arr[Left] < Arr[Right] ? Left : Right;
    return Result;
}
```

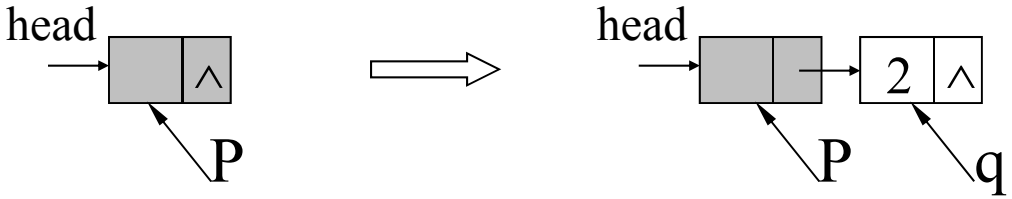
已知由n个整数构成序列的数据分布为先上升再下降，即一开始数据是严格递增的，后来数据是严格递减的，设计尽可能高效算法，找到序列中的最大值。

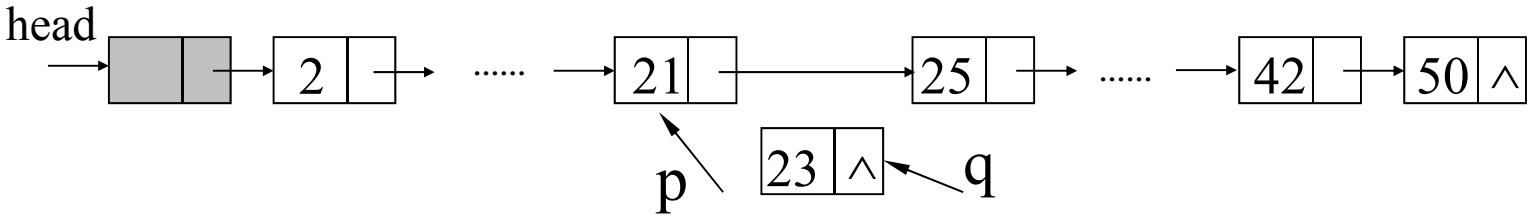
```
int FindMax(int Arr[], int  Left , int Right)
 {   int Mid,Result;
     while (Left < Right)
     {   Mid=(Left+Right)/2;
         if (Arr[Mid] > Arr[Mid-1] && Arr[Mid] > Arr[Mid+1])
             return Mid;
         if (Arr[Mid] > Arr[Mid-1] && Arr[Mid] < Arr[Mid+1])
             Left = Mid+1;
         if (Arr[Mid] < Arr[Mid-1] && Arr[Mid] > Arr[Mid+1])
             Right = Mid - 1;
      }
      if (Left == Right)       Result = Left;
      if (Left < Right)        Result = Arr[Left] > Arr[Right] ? Left : Right;
       return Result;
}
```

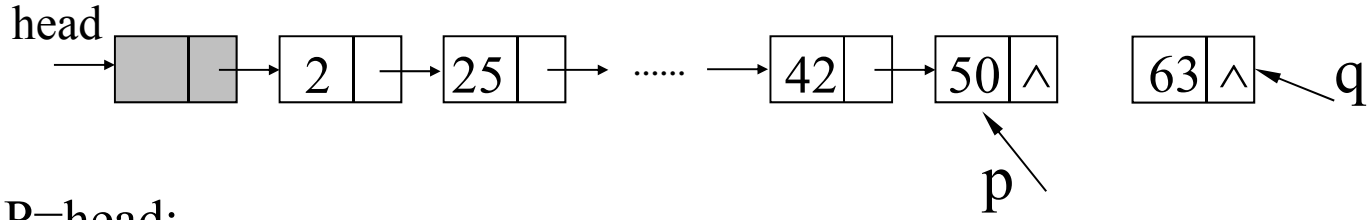
2-4 输入系列整数，建立按升序排列的有序链表



分析：





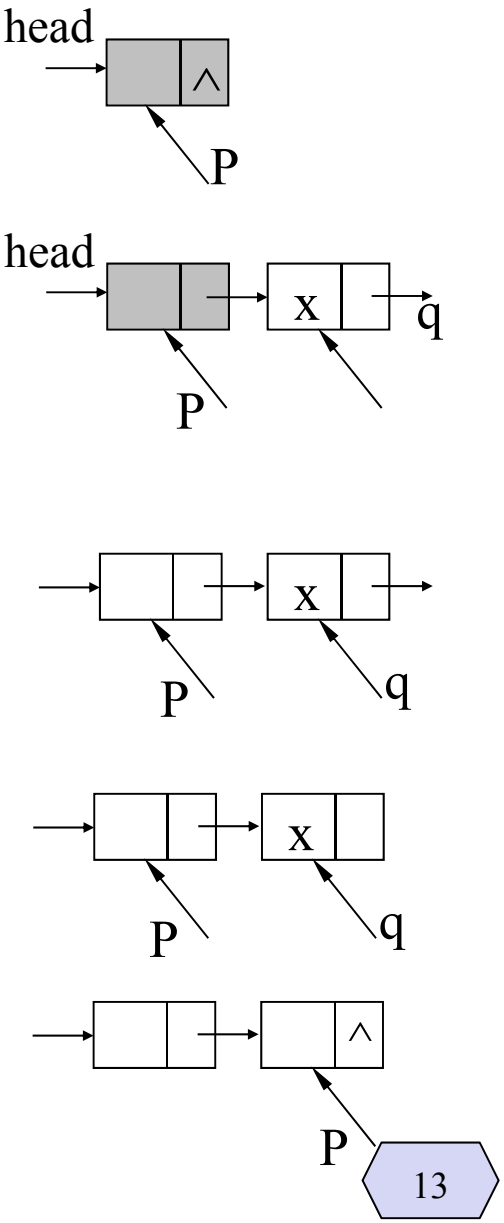



```
P=head;
while((p->next!=NULL)&&(p->next->data<=x))   p=p->next;
```

```
NODE *creat_link()
{    NODE  *head,*p,*q;
     int  x;
     head=(NODE *)malloc(sizeof(NODE *));
     head->next=NULL;
     scanf("%d",&x);
     while(x!=-999)
          {  q=(NODE *)malloc(sizeof(NODE *));
             q->data=x;
             q->next=NULL;
             p=head;
             while((p->next!=NULL)&&(p->next->data<=x))
             p=p->next;
             q->next=p->next;
             p->next=q;
             scanf("%d",&x);
          }
      return(head);
}
```

2-5 删除值为x的结点

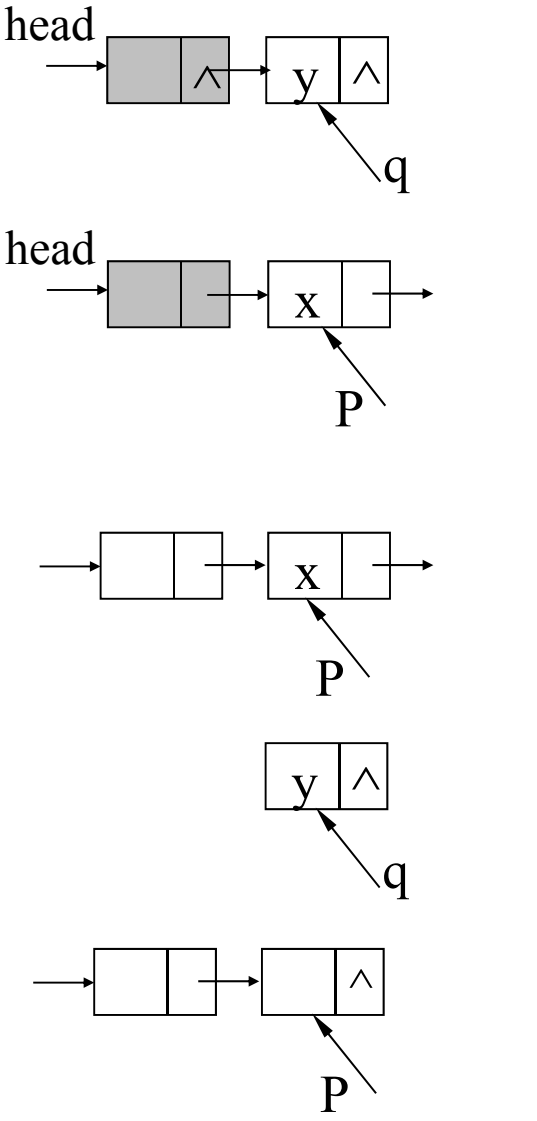
```
Void delete_x(NODE *head,int x)
  {    NODE  *p,*q;
       p=head;
       while((p->next!=NULL)&&(p->next->data!=x))
                p=p->next;
       If(p->next)
          {  q=p->next;
             p->next=q->next;
             free(q) ;   }
}
```



2-6

在值为x的结点后插入一个新的结点y，若x不存在，则插入在表尾

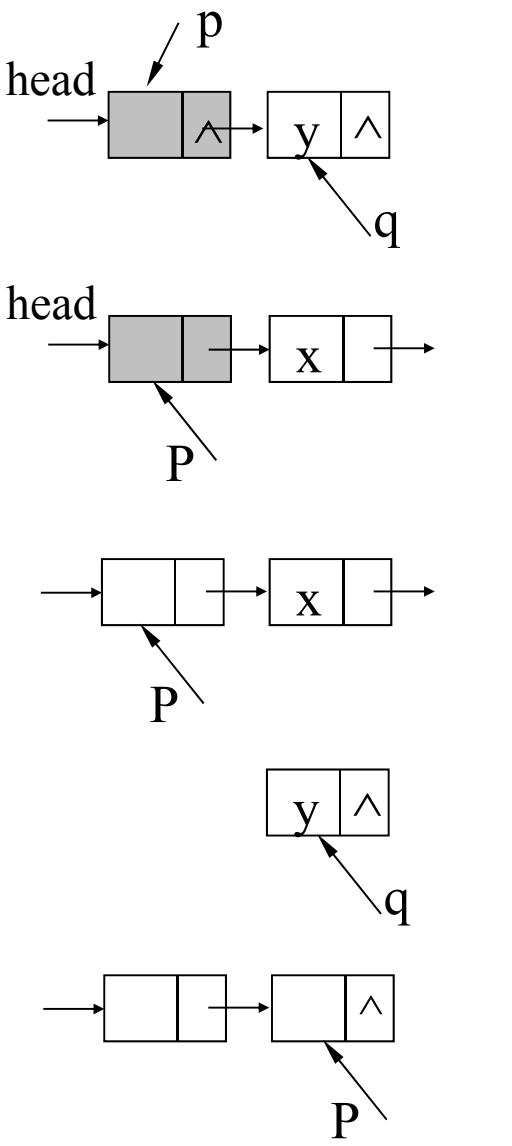
```
Void insert_after_x(NODE *head,int x,int y)
 {    NODE  *p,*q;
      q=(NODE *)malloc(sizeof(NODE *));
      q->data=y ;     q->next=NULL;
      if(head->next) { head->next=q; return;}
      p=head->next;
      while((p->next!=NULL)&&(p->data!=x))
              p=p->next;
      q->next=p->next;
      p->next=q;
}
```



2-7

在值为x的结点前插入一个新的结点y，若x不存在，则插入在表尾

```
Void insert_after_x(NODE *head,int x,int y)
{    NODE  *p,*q;
     q=(NODE *)malloc(sizeof(NODE *));
     q->data=y;     q->next=NULL;
     p=head;
     while((p->next!=NULL)&&(p->next->data!=x))
               p=p->next;
     q->next=p->next;
     p->next=q;
}
```



```
NODE *delete_i_len(NODE *heada,int i,int len)
{   NODE *p,*q;
    int  k;
    if (i==1)
       for(k=1;k<=len;k++)
          {  q=heada;
             heada=heada->next;
              free(q);          }
     else
       {  p=heada;
          for(k=1;k<=i-2;k++) p=p->next;
          for(k=1;k<=len;k++)
             {  q=p->next;
                p->next=q->next;
                free(q)        }
        }
      return(heada);
}
```

2-8

删除链表heada自第i个元素起共len个元素，然后将heada插入到链表headb中第j个元素位置

```
NODE *insert_j (NODE *heada,NODE *headb,int j)
{   NODE *p,*q;
    int  k;
    p=heada;
    while(p!=NULL)  p=p->next ;
    if(j==1)
       {  p->next=headb;
          headb=heada;      }
   else
       {  q=headb;
          for(k=1;k<=j-2;k++) q=q->next;
          p->next=q->next;
          q->next=heada;                }
   return(headb);
}
```

调用：

```
p=delete_i_len(head,i,len);
q=insert_j(p,head,j);
```

2-9 线性链表，求倒数第K个数

```
int backwards-k (NODE *head, int k)
{
  int i=0;
  NODE *p,*q;
  p=q=head->next;
  while(q&&i<=k)
    {
       q=q->next;
        i++;
    }
  if(i<k) return  -1;  //  不足k个元素
  while(q!=NULL)
    {
       q=q->next;
       p=p->next;
    }
   return(p->data);
}
```

2-10 线性链表，求中位数

```
int MidLocate (NODE *head)
{
  NODE *p,*q;
  p=head;q=head->next;
  while(q!=NULL)
     {
        q=q->next;
        if(q) q=q->next;
        p=p->next;
     }
   return(p->data);
}
```

注意几个概念：

- 平均数
  所有元素的和除以元素个数，
  平均水平
- 中位数
  先排序，中间位置的元素，若元素个数为偶数，则为中间两个数的平均数
  代表中等水平
- 众数
  整个数据中出现频率最高的元素

2-11 输入一个已经按升序排序过的数组和一个数字，在数组中查找两个数，使得它们的和正好是输入的那个数字。

思路：

（1）让指针指向数组的头部和尾部，相加，如果小于M，则增大头指针，
　　如果大于则减小尾指针

（2）退出的条件，相等或者头部=尾部

算法：

```
void function（int a[],int n,int M）
{
    int  i=0,j=n-1;
    while(i!=j)
    {
        if(a[i]+a[j]==M)
        {
            printf("%d,%d",a[i],a[j]);
             break;
        }
        a[i]+a[j]>M?j--:i++;
    }
}
```

2-12

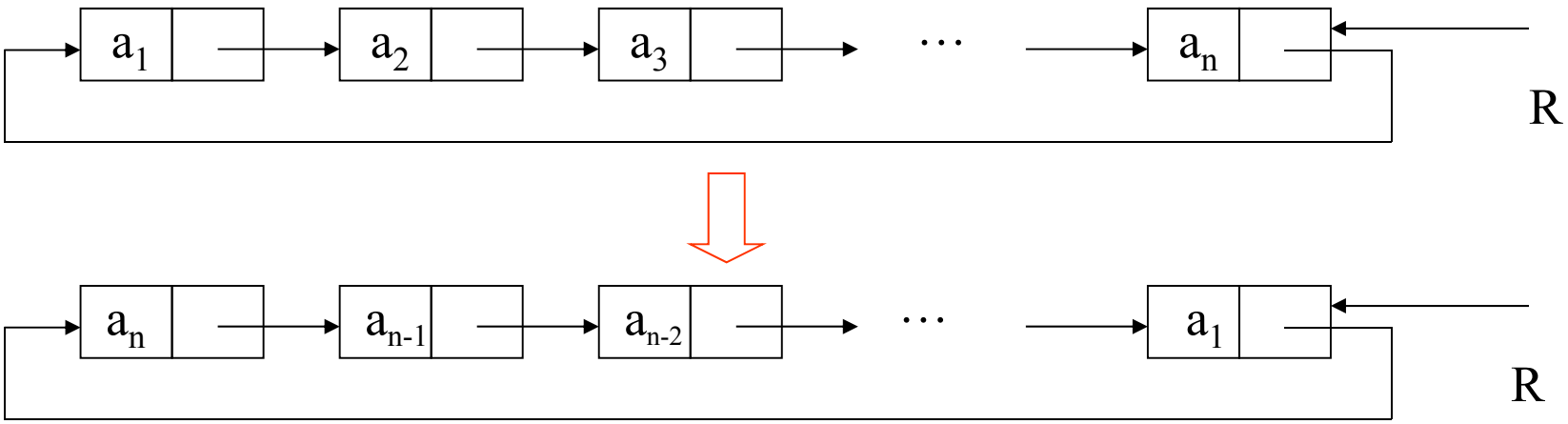
某百货公司仓库中有一批电视机，按其价格从低到高的次序构成了一个单链表存在计算机中，链表的每个结点表示同样价格的若干台电视。现在有新到 m 台价格为 h 的电视机入库。试编写出仓库电视机链表增加电视机的算法。

```
struct CellType {
                    float    price;
                      int    num;
            struct CellType  *next;    };

 typedef struct CellType * LIST;
```

```
LIST  create_link()   //创建仓库
{   LIST  p,q,head;
    float   x; int y;
    head=new CellType;  //建立表头结点
    head->next=NULL;
    printf("输入库中现有电视机的价格和台数，以逗号间隔，输入0,0结束:\n");
    scanf("%f,%d",&x,&y);
    while(x!=0)
            {  q=new CellType;
               q->price=x; q->num=y;
               q->next=NULL;
               p=head;
               while((p->next!=NULL)&&(p->next->price<=x))
                        p=p->next;
               q->next=p->next;
               p->next=q;
               scanf("%f,%d",&x,&y);
            }
   return(head);
}
```

```
void insert( LIST head )      //新的电视到货
{
   LIST p,q;
   p=head->next;
   float x;int y;
   printf("新到电视的价格，台数？:");
   scanf("%f,%d",&x,&y);
   while((p->next!=NULL)&&(p->next->price<=x))
            p=p->next;
   if(p->price==x)
            p->num=p->num+y;
   else
      {  q=new CellType;
         q->price=x; q->num=y;
         q->next=p->next;
         p->next=q;
      }
};
```

举例： 设计算法，将一个单向环形链表反向。头元素变成尾元素，尾元素变成新的头元素，依次类推。



算法如下：

```
Void  REVERS( LIST  &R )
{  position  p, q;
   p = R→next ;
   R→next = p→next ;
   p→next = p ;
   while ( R != NULL )
         {  q = R→next ;
            R→next = q→next ;
            q→next = p→next ;
            p→next = q ;
         }
   R = p ;
} ;
```

2-13

假设有一个单循环链表，其结点含有三个域：*pre*,*data*和*link*，其中*data*为数据域，*link*为指针域，他指向后继结点，*pre*为指针域，他的值为空指针。试编写算法，将此表改成双向链表。

```
struct CellType {
                    int    data;
        struct CellType   *pre,*link;  };
typedef struct CellType   * LIST;
```

```
void output_double( LIST head )
 //输出双向链表
{
   LIST p;
   p=head;
   printf("\n双向循环链表如下：\n");
   do{
            printf("%d,",p->data);
          p=p->pre;
      }while(p!=head);
};
```

```
void double_link(LIST head)
 //单向链表变成双向链表
{
    LIST p,q;
    p=q=head->link;
    do{
          q->link->pre=q;
          q=q->link;
       }while(q!=p);
}
```

```
LIST  create_link()                    //创建单向循环链表
{   LIST  p,q,head;
    int x;
    printf("输入链表结点值，-999结束:\n");
    cin>>x;
    if(x==-999) return(NULL);
    head=new CellType;                 //建立表头结点
    head->data=x;
    head->link=NULL;    head->pre=NULL;    p=head;
    scanf("%d",&x);
    while(x!=-999)
          {  q=new CellType;
             q->data=x; q->link=NULL;q->pre=NULL;
            q->link=p->link;
            p->link=q;   p=q;
             cin>>x;                   }
    p->link=head;
    return(p);
}
```

2-14

设有一个双向链表，每个结点中除有*pre*、*data*和*next*三个域外，还增设了一个访问频度域*freq*。在链表被启用后，频度域*freq*的值均初始化为零，而每当对链表进行一次**LOCATE(*L*,*x*)**的操作后，被访问的节点（即元素值等于*x*的结点）中的频度域*freq*的值便增**1**，同时，调整链表中结点之间的次序，使其按访问频度非递增的次序顺利排列，以便始终保持被频繁访问的节点总是靠近表头结点。试编写符合上述要求的**LOCATE** 操作的算法。

```
struct CellType {
                        int   data;
                        int   freq;
        struct CellType   *next, *pre;  };
typedef struct CellType   * LIST;
```

```
LIST  create_link()   //创建按freq有序的双向链表
{   LIST  p,q,head;
    int   x, f;
    head=new CellType;  //建立表头结点
    head->next=NULL;head->pre=NULL;
    cout<<"输入链表各结点的值和访问频度，以逗号间隔，-999，0结束:"<<endl;
    scanf("%d,%d",&x,&f);
    while(x!=-999)
        {  q=new CellType;
           q->data=x;
           q->freq=f;
           q->next=NULL;q->pre=NULL;
           p=head;
           while((p->next!=NULL)&&(p->next->freq>f))
                        p=p->next;
           q->next=p->next;     q->pre=p;
           if(p->next) p->next->pre=q;
           p->next=q;
           scanf("%d,%d",&x,&f);
        }
    return(head);
}
```

```
void locate(LIST head,int x)
{
   LIST p,q;
   p=head->next;
   while((p!=NULL)&&(p->data!=x)) p=p->next;
   if(p==NULL)    printf("表中无%d结点!\n",x);
   else
      {
            p->freq=p->freq+1;
            if(p->freq>p->pre->freq)
                { q=p;                          //删除
                  p->pre->next=q->next;
                  p->next->pre=p->pre;
                  p=head;                       //重新插入，并保持有序
                  while((p->next!=NULL)&&(p->next->freq>q->freq))
                        p=p->next;
                  q->next=p->next;  q->pre=p;
                  if(p->next) p->next->pre=q;
                  p->next=q;
                }
       }
}
```

2-15

设一单向链表的头指针为***Head***，链表的纪录中包含着整数类型关键字***key***域，试设计算法，将此链表的记录按照***key***递增的顺序进行就地排序。

```
struct CellType {
                        int    data;
            struct CellType    *next;};
typedef struct CellType * LIST;
```

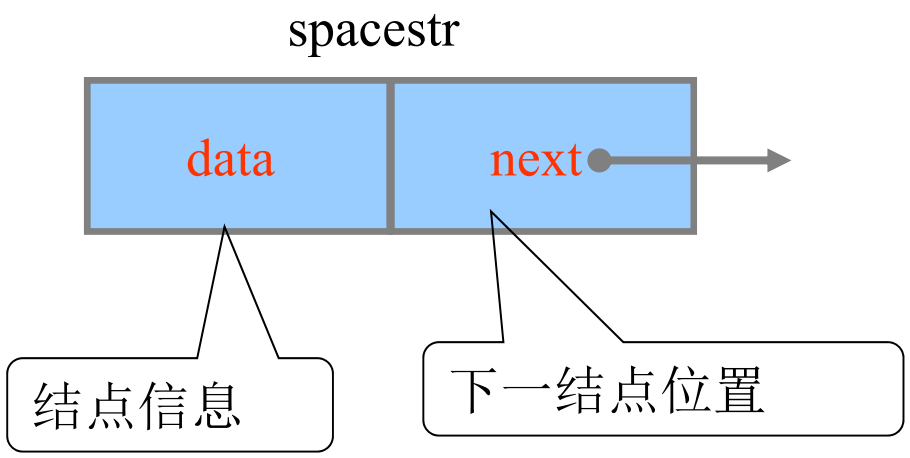
```
LIST  create_link( )   //创建链表
{   LIST  p,q,head;
    int   x;
    head=new CellType;  //建立表头结点
    head->next=NULL;
    cout<<"输入链表各结点的值，以空格间隔，-999结束:"<<endl;
    cin >> x;
    p=head;
    while(x!=-999)
            {  q=new CellType;
             q->data=x;
             q->next=NULL;
               p->next=q;
               p=q;
               cin >> x ;
            }
    return(head);
}
```

```
void order_link(LIST head)   //排序
{   LIST  p,q,r;
    p=head->next;
    head->next=NULL;
    while(p)
            {   printf(".");
                q=p;
                p=p->next;
                r=head;
                while((r->next!=NULL)&&(r->next->data<=q->data))
                        r=r->next;
                q->next=r->next;
                 r->next=q;
            }
}
```

2-16

已知A、B、C是三个线性表且其元素按递增顺序排列，每个表中元素均无重复，在表A删去既在表B中出现又在表C中出现的元素。试设计实现上述删除操作的算法Delete，并分析其时间复杂性。

结点形式



结点类型

```
struct  spacestr  {
      elementtype  data ;
      int    next ;
} ;
```

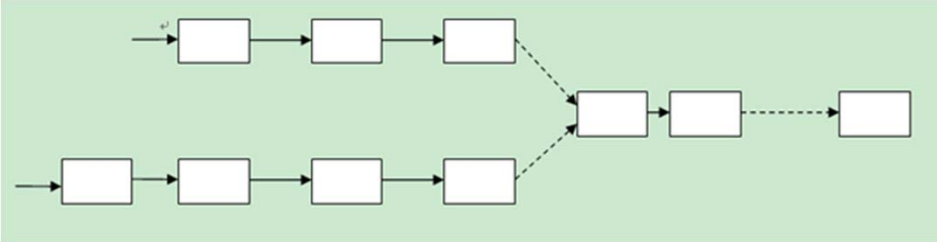
线性表：

```
spacestr  SPACE[ maxsize ] ;
typedef  int  position;
```

万变不离其中，要熟记结点形式，熟练指针操作！

2-17 判断两个链表是否相交，如果相交找出他们的交点。

```
Node *find_node(node *head1, node *head2)
{
    node *p1, *p2; int len1=0;int len2=0;int diff = 0;
    if(NULL == head1 || NULL == head2)
        return NULL;      //有为空的链表，不相交
    p1 = head1;
    p2 = head2;
    while(NULL != p1->next)
    {
        p1 = p1->next;
       len1++;
     }
     while(NULL != p2->next)
    {
        p2 = p2->next;
        len2++;
     }
     if(p1 != p2) //最后一个节点不相同,返回NULL
       return NULL;
```



两个相交链表的最后一个元素肯定相同

```
      diff = abs(len1 - len2);
      if(len1 > len2)
      {
         p1 = head1;
         p2 = head2;
       }
       else
      {
          p1 = head2;
          p2 = head1;
        }
      for(int i=0; i<diff; i++)
          p1 = p1->next;
       while(p1 != p2)
      {
         p1 = p1->next;
         p2 = p2->next;
      }
       return p1;     //相交入口点
}
```

算法时间复杂度：O(len1+len2)

判断两个链表是否相交：（假设两个链表都没有环）

1、判断第一个链表的每个节点是否在第二个链表中

2、把第二个链表连接到第一个后面，判断得到的链表是否有环，有环则相交

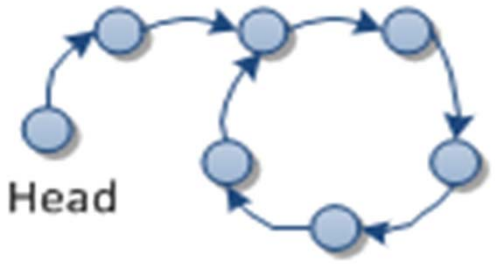
3、先遍历第一个链表，记住最后一个节点，再遍历第二个链表，得到最后一个节点时和第一个链表的最后一个节点做比较，如果相同，则相交

如何判断一个单链表是有环的？（注意不能用标志位，最多只能用两个额外指针）

一种0（n）的办法就是（用两个指针，一个每次递增一步，一个每次递增两步，如果有环的话两者必然重合，反之亦然）：

这两个指针要在环里转多少圈才能相遇呢？

会不会转几千几万圈都无法相遇？



分析：

第一个（速度慢的）指针在环里转满一圈之前，两个指针必然相遇。不妨设环长为L，第一个指针P1第一次进入环时，第二个指针P2在P1前方第a个结点处（0 < a < L），设经过x次移动后两个指针相遇，那么应该有0+x = a + 2x (mod L)，显然x = L-a。下面这张图可以清晰地表明这种关系，经过x = L-a次移动，P1向前移动了L-a个位置（相当于后退了a），到达P1′处，而P2向前移动了2L-2a个位置（相当于后退了2a），到达P2′处，显然P1′和P2′是同一点。

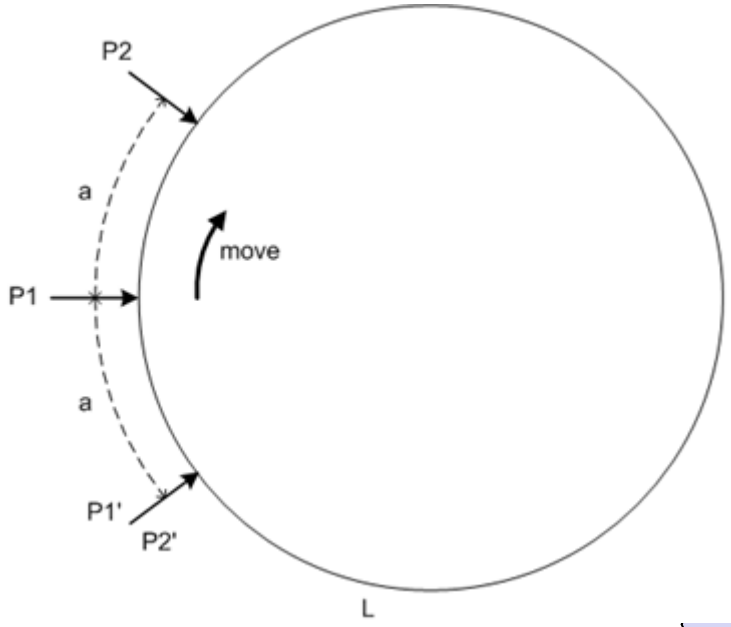
慢指针（P1）转一周之内，必然与快指针（P2）相遇。

环长：

从刚才那个相遇点开始，固定P2，继续移动P1，直到P1与P2再次相遇，所经过的步数。

前面那段子链的长度：

让P1和P2都回到链表起点，然后让P2先往前走L次（每次往前走一步），然后P1和P2再同时往前走，当它们再次相遇时，P1所走的步数就是环前面的子链长度



如何判断一个单链表是有环的？

用两个指针，一个每次递增一步，一个每次递增两步，如果有环的话两者必然重合，反之亦然

```
bool IsLoop(const node* head)
{
   node *low, *fast;
   if(head==NULL)   return false;
   low=head;
   fast=head->next;
   while(fast  && fast->next)
   {
      low=low->next;
      fast=fast->next->next;
      if(low==fast) return true;
   }
   return false;
}
```

如果链表可能有环，如何判断两个链表是否相交?

链表1 步长为1， 链表2步长为2 ，
如果有环且相交则肯定相遇，否则不相交

```
list1 head: p1
list2 head: p2
while( p1 != p2 && p1  && p2  )
{
     p1 = p1->next;
     if ( p2->next )
        p2 = p2->next->next;
     else
        p2 = p2->next;
}
if ( p1 == p2 && p1 && p2)   //相交
else //不相交
```

- ✓ 数组是由下标（index）和值（value）组成的序对（index，value）的集合。
- ➢ 也可以定义为是由相同类型的数据元素组成有限序列。
- ➢ 数组在内存中是采用一组连续的地址空间存储的，正是由于此种原因，才可以实现下标运算。
- ➢ 所有数组都是一个一维向量。

特殊矩阵：

若n 阶矩阵 A 中的元素满足下述性质：$a_{ij}=a_{ji}$，$1\leqslant i,j\leqslant n$

则称 n 阶对称阵。

对于对称矩阵，为实现节约存储空间，我们可以为每一对对称元素分配一个存储空间，这样，原来需要的 n^2 个元素压缩存储到 n(n+1)/2 个元素空间。

对称关系：

设sa[0..n(n+1)/2] 做为n阶对称阵A的存储结构，则Sa[k]和a_{ij}的一一对应关系为：

$$K=\begin{cases} i(i-1)/2+j & \text{当} i\geqslant j \\ j(j-1)/2+i & \text{当} i<j \end{cases}$$

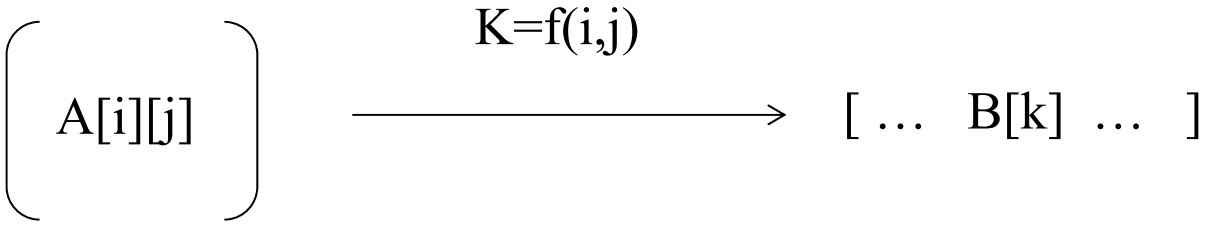
映像关系：

B	a_{11}	a_{21}	a_{22}	a_{31}	a_{32}	a_{33}	…	a_{ij}	…	a_{n1}	…	a_{nn}
k =	1	2	3	4	5	6		i(i-1)/2+j		n(n-1)/2+1		n(n+1)/2

——对上（下）三角矩阵可采用同样的方法。

对所有非零元素按行排序，其排列序号 K = f(i , j)

将所有非零元素存入一维数组 B[K]=A[i][j]，即：



对称矩阵： A[i][j] => A(i,j)

```
elementtype  A( elementtype B[ ] , int i, int j )
{
  int k ;
  if ( i >= j )
     k= i*(i-1)/2 + j ;
  else
     k= j*(j-1)/2 + i ;
  return( B[k] ) ;
}
```

```
elementtype  A(elementtype B[ ] , int i, int j)
{
  int k ;
  if ( i >= j )
```

```
elementtype  A(elementtype B[ ] , int i, int j)
{
  int k ;
  if ( i <= j )
     {  k =  i*(i-1)/2 + j ;
        return ( B[k] );     }
  else
        return( 0 ) ;
}
```

上三角阵

稀疏矩阵压缩存储

稀疏矩阵中，零元素的个数远远多于非零元素的个数。为实现压缩存储，我们仍考虑只存储非零元素。

$$M=\begin{bmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 14 & 0 \\ 0 & 0 & 24 & 0 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & -7 & 0 & 0 & 0 \end{bmatrix}$$

$$T=M^{\cdot}=\begin{bmatrix} 0 & 0 & -3 & 0 & 0 & 15 \\ 12 & 0 & 0 & 0 & 18 & 0 \\ 9 & 0 & 0 & 24 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -7 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 14 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

i	j	v
1	2	12
1	3	9
3	1	-3
3	6	14
4	3	24
5	2	18
6	1	15
6	4	-7

a.data

i	j	v
1	3	-3
1	6	15
2	1	12
2	5	18
3	1	9
3	4	24
4	6	-7
6	3	14

b.data

```
#define  MAXSIZE  12500
typedef  struct {
        int  i, j ;
        elementtype  e ;
}  Triple ;
typedef  struct  {
         Triple  data[MAXSIZE+1];
          int  mu, nu, tu;
} TSMatrix;
```

```
Void  FastTransposMatrix( TSMatrix  M，  TSMatrix  &T )
{  T.mu = M.nu; T.nu = M.mu; T.tu = M.tu;
   if  ( T.tu )
       {    for ( col = 1; col <= M.nu; ++col )  num[ col ] = 0 ;
            for ( t = 1; t <= M.tu; ++t )  ++num[ M.data[ t ].j ] ;
            cpot [ 1 ] = 1 ;
            for(col=2;col<M.nu;col++)
                  cpot [ col ] = cpot[ col –1 ] + num[ col – 1 ] ;
            for ( p = 1;  p <= M.tu;  ++p )
              {   col = M.data[ p ].j ; q = cpot[ col ] ;
                  T.data[ q ].i = M.data[ p ].j ;
                  T.data[ q ].j = M.data[ p ].i ;
                  T.data[ q ].e = M.data[ p ].e ;
                  ++cpot[ col ] ;
               }
        }
}
```

转置的改进算法:

$T(n) = O(nu + tu)$

树与二叉树

（一）树的概念

（二）二叉树

1、二叉树的定义及其主要特征

2、二叉树的顺序存储结构和链式存储结构

3、二叉树的遍历

4、线索二叉树的基本概念和构造

（三）树、森林

1、树的存储结构

2、森林与二叉树的转换

3、树和森林的遍历

（四）树与二叉树的应用

1、二叉排序树

2、平衡二叉树

3、哈夫曼（huffman）树和哈夫曼编码

```
Struct  node  {
    Struct  node  *lchild ;
    Struct  node  *rchild ;
          datatype  data ;  } ;
Typedef  struct   node    * BTREE ;
```

二叉树的基本操作：

- 二叉树的建立
- 二叉树遍历（先序/中序/后序/层序，递归和非递归）
- 各类结点数量统计（度为0、1、2，结点总数）
- 判断满二叉树/完全二叉树
- 二叉树左右孩子交换/二叉树镜像
- 二叉树的深度/宽度
- 二叉树最大/最短路径
- 任意两个结点的最近公共祖先
- 任意两个结点的路径
- 结点层号、祖先

几种特殊的二叉树：

- 满二叉树
- 完全二叉树
- 二叉排序树
- 平衡二叉树
- 哈夫曼树

3-1

一棵二叉树树的先序、中序和后序序列分别如下，其中一部分未显示出来，试求出空格处的内容，并画出该二叉树。

先序：_B_F_ICEH_G

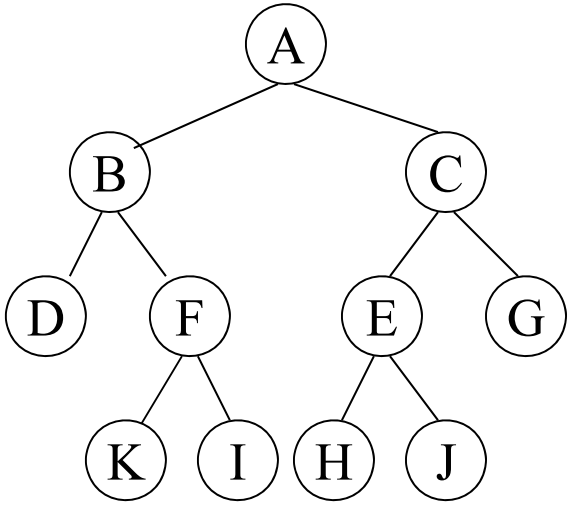
中序：D_KFIA_EJC_

后序：_K_FBHJ_G_A

先序：ABDFKICEHJG

中序：DBKFIAHEJCG

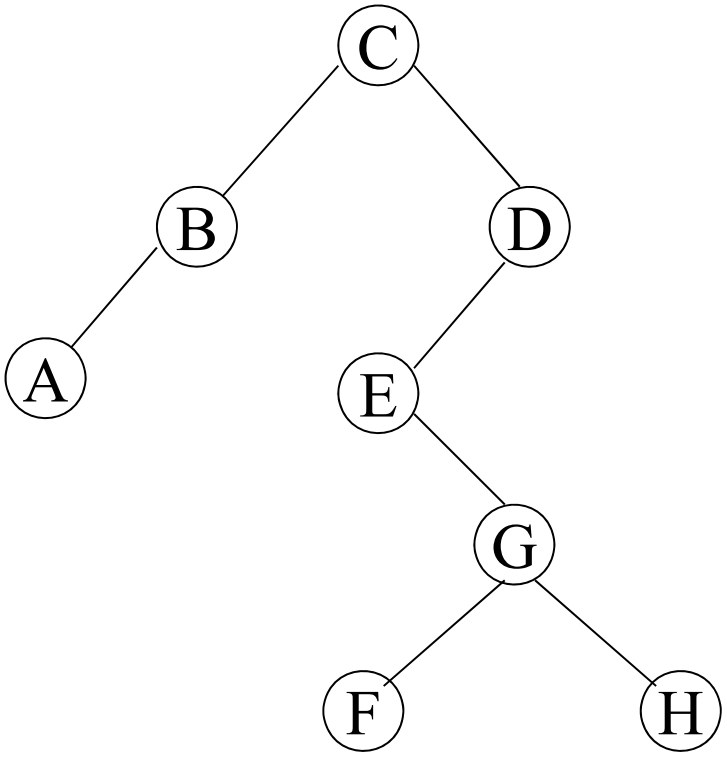
后序：DKIFBHJEGCA

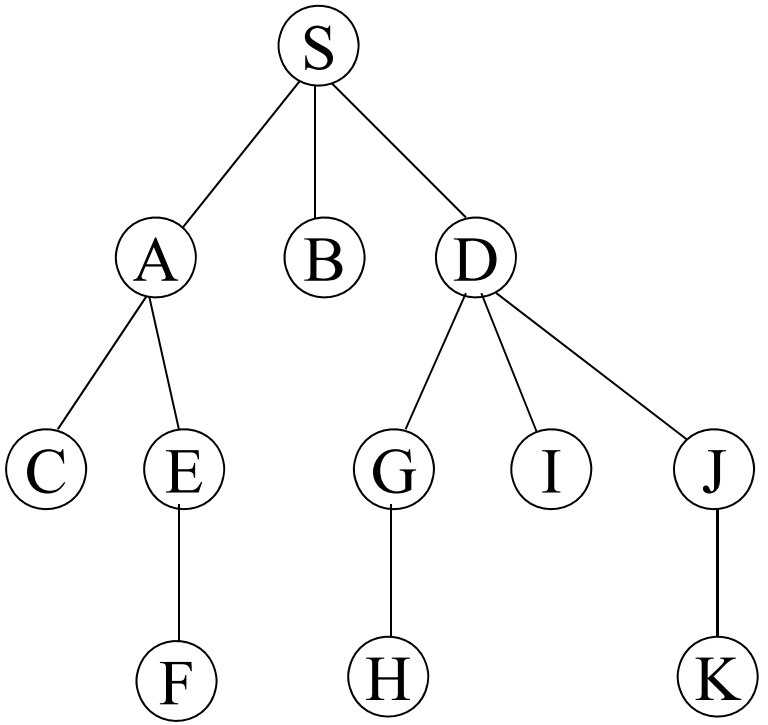


3-2

二叉树中序序列为：ABCEFGHD，后序序列为：ABFHGEDC
画出此二叉树

假设先根次序遍历某棵树的结点次序为SACEFBDGHIJK，后根次序遍历该树的结点次序为CFEABHGIKJDS，请画出这棵树。





3-3

试举出：
先序遍历和中序遍历相同的二叉树？
先序遍历和后序遍历相同的二叉树？
中序遍历和后序遍历相同的二叉树？

完全二叉树的某结点若无左孩子结点，则它必是叶结点？对否

3-4

一棵有124个叶子结点的完全二叉树，最多有__？__个结点。

$n_0=n_2+1$
$n=n_0+n_1+n_2$
$\Longrightarrow$ $n=n_1+2n_0-1$

但在完全二叉树中，n_1不是 0 就是 1
只有$n_1=1$时，n 取最大值为$2n_0$

3-5

设n0为哈夫曼树的叶子结点数目，简要推出该哈夫曼树有多少个结点？

总结点数$n=n_0+n_1+n_2$

而根据二叉树性质有：$n_0=n_2+1$

又因哈夫曼树的$n_1=0$

所以： $n=n_0+n_2=2n_0-1$

3-6

证明题：

任意一棵有n个结点二叉树，已知它有m个叶子结点，试证明非叶子结点有（m-1）个度为2，其余度为1。

证明：

设n_1为二叉树中度为1的结点数，n_2为度为2的结点数，

则总结点数$n=n_1+n_2+m$

又设分支数B，有$n=B_{入}+1$

而 $B_{出}=n_1+2n_2$

由$B_{入}=B_{出}$，所以有：

$n=n_1+2n_2+1$

$n_1+n_2+m=n_1+2n_2+1$

所以： $n_2=m-1$

3-7

证明任一棵满二叉树T中的分支数B 满足：

$B=2(n_0-1)$，其中n_0为叶子结点数

证明：

满二叉树中不存在度为1的节点，设度为2的结点数为n_2

则： $n=n_0+n_2$

又： $n=B+1$

所以有： $B=n_0+n_2-1$ ， 而 $n_0=n_2+1, n_2=n_0-1$

$B=n_0+n_0-1-1=2(n_0-1)$

熟练掌握完全二叉树、满二叉树、哈夫曼数的结构特点，

度为0、1、2各类结点个数，并能拓宽并应用到 k 叉树上。

3-8 具有n个结点的满二叉树，其叶子结点的个数为多少？

方法一：设满二叉树的高度为 h，
则根据二叉树的性质，叶子结点数为2^{h-1}
二叉树总结点数$n=2^h-1$
可导出:$2^{h-1}=(n+1)/2$

方法二：结点总数：$n=n_0+n_1+n_2$
但对满二叉树，除有$n_0=n_2+1$外，还有$n_1=0$
故有：$n=n_0+n_0-1$
$n_0=(n+1)/2$

具有n个结点的完全二叉树，其叶子结点的个数为多少？

设高为h的二叉树只有度为0和度为2的结点，则此类二叉树的结点树至少为____，至多为_____。

答案： $2h-1$ 和 2^h-1

3-9 二叉树非递归遍历

先序遍历非递归算法

```
Loop:
{
   if (BT 非空)
      { 输出;
        进栈;
        左一步;}
   else
      { 退栈;
        右一步;}
};
```

中序遍历非递归算法

```
Loop:
{
   if (BT 非空)
      { 进栈;
        左一步;}
   else
      { 退栈;
        输出;
        右一步;}
};
```

先序遍历非递归算法

```
Loop:
{
   if(BT 非空)
      { 进栈;
        左一步;}
   else
      { 当栈顶指针
        所指结点的
        右子树不存
        在或已访问,
        退栈并访问;
        否则右一步; }
};
```

中序遍历非递归算法

```
Loop:
{
   if (BT 非空)
      { 进栈;
        左一步;}
   else
      { 退栈;
        输出;
        右一步;}
};
```

```
Void  inorder(BiTree &root)
{   BiTNode *stack[MAX];
     int top=0;
     do{   while(root!=NULL）
              {   top++;
                  if(top>max) printf(“stack is full);
                  else stack[top]=root;
                  root=root->lchild;
              }
              if(top!=0)
                 {  root=stack[top];
                    top--;
                    printf(root->data);
                    root=root->rchild;   }
       }while(top==0)&(root==NULL);
}
```

先序遍历非递归算法

```
Loop:
{
   if (BT 非空)
      { 输出;
        进栈;
        左一步;}
   else
      { 退栈;
        右一步;}
};
```

```
Void  preorder(BiTree &root)
{   BiTNode *stack[MAX];
     int top=0;
     do{   while(root!=NULL）
              {   printf(root->data);
                  top++;
                  if(top>max) printf(“stack is full);
                  else stack[top]=root;
                  root=root->lchild;
              }
              if(top!=0)
                 {  root=stack[top];
                    top--;
                    root=root->rchild;   }
       }while(top==0)&(root==NULL);
}
```

后序遍历非递归算法

```
Loop:
{
   if (BT 非空)
      { 进栈;
        左一步;}
   else
      {  当栈顶指针
         所指结点的
         右子树不存
         在或已访问,
         退栈并访问;
         否则右一步; }
};
```

```
Void  postorder(BiTree &root)
{   BiTNode *stack[MAX];
     int top=0;  BiTNode p;
     do{   while(root!=NULL）
           {   top++;
               if(top>max) printf(“stack is full);
               else stack[top]=root;
               root=root->lchild;
           }
           p=NULL;  b=TRUE;
           while(top!=0)&&(b)    //右子树不存在或已访问
           {    root=stack[top];
                if(root->rchild==p)
                     {  printf(root->data);    //访问根结点
                        top--;   p=root;        } //p指向刚访问
                                                    结点
                else {  root=root->rchild;
                         b=FALSE;              }
           }
      }while(top==0);
}
```

3-10 **按层序遍历二叉树！**

```
Void  Level_list_Btree( BTREE  T)
{
   QUEUE  Q;
   MAKENULL(Q);
   ENQUEUE(T,Q)
   while(!EMPTY(Q))
    {   P=FRONT(Q);
        DEQUEUE(Q);
        visite(P->data);
        if(P->lchild)   ENQUEUE(P->lchild,Q)
        if(P->rchild)   ENQUEUE(P->rchild,Q);
    }
}
```

下标	0	1	2	3	…
结点↑					
层号					
度					
双亲					
…					

同时可得到：
宽度、高度等信息
判断：
满二叉树
完全二叉树等

3-11 求任意二叉树结点总数

$$
\begin{cases}
f(b)=0 & \text{若b=NULL} \\
f(b)=1 & \text{若b->lchild=b->rchild=NULL} \\
f(b)=f(b\text{->lchild})+f(b\text{->rchild})+1 & \text{其它}
\end{cases}
$$

```
int  nodes( BTREE *b)
{    int  num1,num2 ;
     if ( b==NULL ) return(0);
     else  if (( b->lchild==NULL) && ( b->rchild==NULL)) return(1);
     else {   num1=nodes(b->lchild);
              num2=nodes(b->rchild);
              return(num1+num2+1);    }
}
```

求任意二叉树叶子结点数

$$
\begin{cases}
f(b)=0 & \text{若b=NULL} \\
f(b)=1 & \text{若b->lchild=b->rchild=NULL} \\
f(b)=f(b\text{->lchild})+f(b\text{->rchild}) & \text{其它}
\end{cases}
$$

```
int  leafs( BTREE *b)
{  int  num1,num2 ;
   if ( b==NULL ) return(0);
   else  if (( b->lchild==NULL) && ( b->rchild==NULL)) return(1);
   else {  num1=leafs(b->lchild);
           num2=leafs(b->rchild);
           return(num1+num2);    }
 }
```

求任意二叉树单孩子结点（度为1）数

f(b)=0　　若b=NULL

f(b)=1　　若b->lchild、b->rchild 其中一个为NULL

f(b)=f(b->lchild)+f(b->rchild)　　其它

```
int  onechild( BTREE *b)
{    int  num1,num2 ;
     if ( b==NULL ) return(0);
    else  if ( (( b->lchild==NULL) && ( b->rchild<>NULL)) ||
     (( b->rchild==NULL) && ( b->lchild<>NULL)) )
       return(1);
      else {  num1=onechild(b->lchild);
              num2=onechild(b->rchild);
              return(num1+num2);    }
      }
```

求任意二叉树双孩子结点（度为2）数

f(b)=0　　若b=NULL

f(b)=1　　若b->lchild<>NULL,且b->rchild<>NULL

f(b)=f(b->lchild)+f(b->rchild)　　其它

```
int  twochild( BTREE *b)
{    int  num1,num2 ;
     if ( b==NULL ) return(0);
     else  if (( b->lchild!=NULL) && ( b->rchild!=NULL)) return(1);
     else {  num1=twochild(b->lchild);
             num2=twochild(b->rchild);
             return(num1+num2);    }
      }
```

3-12 设计算法，以（*Key*，*LT*，*RT*）的形式打印二叉树，其中*Key*为结点的值，LT和RT是以括号形式表示的二叉树。

例如：A(B,C(D,E(F)))

```
void DispBTNode(BTREE *B)
{    if ( B!=NULL)
     {  printf("%c",B->data);
        if(B->lchild!=NULL || B->rchild!=NULL)
         {  printf("(");
            DispBTNode(B->lchild);
            if(B->rchild!=NULL)
                printf(",");
            DispBTNode(B->rchild);
            printf(")");
         }
     }
}
```

以（*Key*，*层号*）的形式输出二叉树的各结点。

```
void PrintTree(BTREE *T, int n)
{
   if(T)
  {
     printf("(%c,%d)",T->data,n);

     PrintTree(T->lchild,n+1);
     PrintTree(T->rchild,n+1);
  }
}
```

3-13 输出二叉树所有叶子结点

```
void DispLeaf(BTREE *B)
{
    if(B!=NULL)
   {
       if(B->lchild==NULL && B->rchild==NULL)
           printf("%c ",B->data);
       else
       {
          DispLeaf(B->lchild);
          DispLeaf(B->rchild);
       }
    }
}
```

3-14

给定一棵用链表表示的二叉树，其根节点为Root，试写出求各结点的层数的算法。

```
Void  Layer(BTREE  &root)
{   BTREE  *que[MAXN];       //循环队列
    int hp , tp , lc , level ;
    if (root!=NULL)
       {  hp=0;tp=1;lc=1; que[0]=root; level=1;
          do{  hp=(hp%MAXN)+1;
               root=que[hp];
               if  (root->lchild!=NULL)
                   {  tp=(tp%MAXN)+1;
                      que[tp]=root->lchild;  }
                      if  (root->rchild!=NULL)
                          {  tp=(tp%MAXN)+1;
                             que[tp]=root->rchild;  }
                      printf("Level(",root->data:2,")=",level);
                      if  (hp==lc) {  level++;  lc=tp;}
           }while(hp==tp);
       }
}
```

3-15

在左右链表示的二叉树中，查找值为x的结点，并求x结点所在的层数。

```
BinTree SearchBTree(BTREEe *T, ElementType x)
{  if(T){  if(T-data==x) return(T);
           SearchBTree(T->lchild,x);
           SearchBTree(T->rchild,x);  }
}

Int Inlevel(BTREE  *T , ElementType x)
{    int Static l=0;
     if(T){   if(l==0) {  l++;
                          if(T->data==x)  return(l);
                          if(T->lchild!!T->rchild) l++;   }
              else { if(T->data==x) return(l);
                     if(T->lchild!!T->rchild) l++;
                     else return(0);                          }
              Inlevel(T->lchild,x);
              Inlevel(T->rchild,x);
           }
}
```

3-16

给定一棵用链表表示的二叉树，其根节点为Root,试写出求二叉树的深度的算法。

说明：

hp: 搜索结点

lc:搜索过程中的最后结点

当hp=lc时，说明此时搜索的结点正好是该层中的最后一个结点。

```
int DepthOfTree(BTREE  *root)
{   BTREE  *que[MAXN];
    int hp,tp,lc,depth;
    depth=0;
    if (root!=NULL)
      {  hp=0;tp=1;lc=1;que[0]=root;
        do{  hp=(hp%MAXN)+1;
             root=que[hp];
             if (root->lchild!NULL)
                {   tp=(tp%MAXN)+1;
                    que[tp]=root->rchild;  }
             if (root->rchild!=NULL)
                {   tp=(tp%MAXN)+1;
                    que[tp]=root->lchild;  }
             if (hp==lc)
                {   depth++;
                    lc=tp;              }
          while(hp==tp)
        }
     return(depth);
}
```

设二叉树结点表示的数据元素类型为**Elementtype**，二叉树用左右链表示。一棵二叉树的最大枝长和最小枝长分别定义如下：

最大枝长就是二叉树的层数；最小枝长就是离根结点距离最近的叶结点到根路径的边数。

请设计一个算法，同时求出一棵二叉树的最大和最小枝长。

最长路径，最短路径，最长枝长，最短枝长。。。。。

3-17 采用递归输出从叶子结点到根结点的路径

```
void LeafToRoot( BTREE *B , char path[] , int pathlen )
{  int i;
   if(B!=NULL)
   {
      if(B->lchild==NULL && b->rchild==NULL)  //叶子结点
      {   printf("%c 到根的路径：%c ", B->data , B->data);
          for(i=pathlen-1;i>=0;i--)
               printf("%c ",path[i]);
          printf("\n");
      }
      else
      {   path[pathlen]=B->data;  //当前结点放入路径中
          pathlen++;
           LeafToRoot (B->lchild, path , pathlen);   //递归扫描左子树
           LeafToRoot (B->rchild, path , pathlen);   //递归扫描右子数
          pathlen--;
      }
   }
}
```

3-18 采用递归输出给定结点 x 到根结点的路径

```
void XToRoot( BTREE *B , char path[] , int pathlen , char x)
{  int i;
   if(B!=NULL)
   {
      if(B->data == x)    //找到结点x
      {   printf("%c 到根的路径：%c ", x, x);
          for(i=pathlen-1;i>0;i--)
               printf("%c ",path[i]);
          printf("\n");
      }
      else
      {   path[pathlen]=B->data;  //当前结点放入路径中
          pathlen++;
          XToRoot(B->lchild, path , pathlen, x);   //递归扫描左子树
          XToRoot(B->rchild, path , pathlen, x);       //递归扫描右子数
          pathlen--;
      }
   }
}
```

3-19 输出二叉树的最长路径

```
int Depth(BTREE *T)   // 求二叉树T的深度
 {
     if(T==NULL)
         return(0);         //空树深度为0
      return 1+(Depth(T->lchild)>Depth(T->rchild)? Depth(T->lchild):Depth(T->rchild));
         //选择左右孩子深度高的然后加上根节点这一层就是深度了
}



void   LongestPath(BTREE *T)
{
    if(T!=NULL)  //非空二叉树树
    {
         printf("%c,",T->data);         //输出根节点
         if(Depth(T->lchild)>Depth(T->rchild))       //判断左右子树的深度
                  LongestPath(T->lchild);
         else
                  LongestPath(T->rchild);
     }
}
```

3-20 输出二叉树的最短路长

```
int ShortestDepth(BTREE *root)
{
   int rdepth,ldepth;

   if(!root)    //空树
         return 0;
   else if(!root->rchild && !root->rchild)        //只有根节点
         return 1;
   else if(root->lchild && !root->rchild)         //只有左子树
         return ShortestDepth(root->lchild) + 1;
   else if(!root->lchild && root->rchild)         //只有右子树
         return ShortestDepth(root->rchild) + 1;
   ldepth=ShortestDepth(root->lchild);
   rdepth=ShortestDepth(root->rchild);
   if(ldepth<rdepth)
         return ldepth+1;
  else
         return rdepth+1;
}
```

```
void Ancestors(BTREE *root,char x)
{
    int b,top=0;
    BTREE *stack[MAX], *p,*q;
    p=root;
    if(p->data==x)     printf("%c 根节点",p->data);
    else
        do{
            while((p->lchild!=NULL)&&(p->data!=x))
              { stack[++top]=p;   p=p->lchild;        }
            if(p->data==x)
                while(top!=0)
                    {   p=stack[top--];
                        printf("%c ",p->data);      }
            q=NULL;  b=1;
            while((top!=0)&&b)
              {   p=stack[top];
                  if(p->rchild==q)
                    { top--;   q=p;  }
                  else
                    { p=p->rchild;  b=0;  }
              }
        }while(top!=0);
}
```

3-21

在二叉树中查找值为x的结点，请编写算法，打印值为x的结点的所有祖先。假设值为x的结点不多于一个。

用后序遍历的非递归算法查找值为x的结点。用栈保留查找过程中所搜索过的结点，当查找到x时，栈中保留的结点即为x的祖先。

3-22 设计算法判断给定的二叉树是否为满二叉树

f(b)=TRUE　　若b->lchild=NULL且b->rchild=NULL

f(b)=FALSE　　若b->lchild,b->rchild其中的一个为NULL

f(b)=f(b->lchild)&&f(b->rchild)　　若b->lchild<>NULL且b->rchild<>NULL

```
int  isfulltree( BTREE *b)
{    if ( b!=NULL )
     if (( b->lchild==NULL) && ( b->rchild==NULL)) return(1);
     else if (( b->lchild==NULL) || ( b->rchild==NULL)) return(0);
     else  return(isfulltree(b->lchild)&&isfulltree(b->rchild));
 }
```

3-23 设计算法判断给定的二叉树是否为完全二叉树

算法思想：

根据完全二叉树的定义，对完全二叉树按照从上到下、从左到右的层次遍历，应该满足一下两条要求：

- ✓ 某节点没有左孩子，则一定无右孩子
- ✓ 若某节点缺左或右孩子，则其所有后继一定无孩子

若不满足上述任何一条，均不为完全二叉树。

算法思路：采用层序遍历算法，用cm变量值表示迄今为止二叉树为完全二叉树（其初值为1，一旦发现不满足上述条件之一，则置cm为0），bj变量值表示迄今为止所有节点均有左右孩子（其初值为1），一旦发现一个节点没有左孩子或没有右孩子时置bj为0），在遍历完毕后返回cm的值。

```
int   IsCompleteBTree1(BTREE  *b)
{     BTREE  *Qu[MaxSize],*p;                          //定义一个队列，用于层次遍历
      int front=0,rear=0，cm=1，bj=1;
      if(b!=NULL) return 1;
      Qu[++rear]=b;                                    //根节点进队
      while(front!=rear)
      {     p=Qu[++front];
            if(p->lchild==NULL)                        //*p节点没有左孩子
            {    bj=0;
                 if(p->rchild!=NULL)                   //没有左孩子但有右孩子
                 cm=0;                                 //则不是完全二叉树
            }
            else                                       //*p节点有左子树
            {   if(bj==1)                              //迄今为止，所有节点均有左右孩子
                {   Qu[++rear]=b->lchild;              //左孩子进队
                    if(p->rchild==NULL)                //*p有左孩子但没有右孩子
                          bj=0;
                    else
                          Qu[++rear]=p->rchild;        //右孩子进队
                }
                else                                   //bj=0:迄今为止，已有节点缺孩子
                     cm=0;                             //而此时*p节点有左孩子，违反（2）
            }
      }
      return cm;
}
```

思路：在层序遍历的过程中，找到第一个非满节点。满节点指的是同时拥有左右孩子的节点。在找到第一个非满节点之后，剩下的节点不应该有孩子节点；如果有，那么该二叉树就不是完全二叉树。

```
Bool  IsCompleteTree2(BTREE *root)
{  QUEUE *Q; BTREE *temp; bool flag = false;
   EnQueue(root,Q);
   while(!EMPTY(Q))
   {    temp = DeQueue(Q);
        if(flag)
        {     if(temp->lchild || temp->rchild)    //
              return false;
        }
       else
        {   if(temp->lchild && temp->rchild)   //左右孩子同时存在，则压入队列
            {  EnQueue(temp->lchild,Q);
               EnQueue(temp->rchild,Q);
            }
            else if(temp->rchild)   //如果只存在右孩子，则，一定不满足满二叉树
               return false;
            else if(temp->lchild)
            {  EnQueue(temp->lchild);  //改变标志位
               flag = true;
            }
            else    flag = true;
        }
   }
   return true;
}
```

任意的一个二叉树，都可以补成一个满二叉树。这样中间就会有很多空洞。在广度优先遍历的时候，如果是满二叉树，或者完全二叉树，这些空洞是在广度优先的遍历的末尾，所以，但我们遍历到空洞的时候，整个二叉树就已经遍历完成了。而如果，是非完全二叉树，我们遍历到空洞的时候，就会发现，空洞后面还有没有遍历到的值。这样，只要根据是否遍历到空洞，整个树的遍历是否结束来判断是否是完全的二叉树。

算法如下：

```
bool IsCompleteBTree3(BTREE *root)
{
    QUEUE  *Q;
    BTREE  *ptr;
    EnQueue(root，Q);              // 进行广度优先遍历（层次遍历），并把NULL节点也放入队列
    while ((ptr = DeQueue(Q)) != NULL)
    {
      EnQueue(ptr->left, Q);
      EnQueue(ptr->right, Q);
    }

   while (! EMPTY(Q))      // 判断是否还有未被访问到的节点
   {
      ptr = DeQueue(Q);
                              // 有未访问到的的非NULL节点，则树存在空洞，为非完全二叉树
      if (ptr !=NULL)      return false;
    }
   return true;
}
```

3-24 设计算法判断给定的二叉树是否为二叉树排序树

```
Status IsBSTree(BTREE  *T)
/* 判别给定二叉树t是否为二叉排序树。若是，则返回TRUE，否则FALSE */
{
    BTREE  *p;
    if(!T || !T->lchild&&!T->rchild)  return TRUE;    // 空树或只有根节点
    if(T->lchild)
    {    p=T->lchild;
         while(p->rchild)   p=p->rchild;// 找左子树最右的，也是左子树最大的
         if(T->data.key< p->data.key)    return FALSE;    //
     }
   if(T->rchild)
    {    p=T->rchild;
         while(p->lchild)      p=p->lchild;// 找右子树最左的，也是右子树最小的
         if(T->data.key> p->data.key)    return FALSE; //
    }
    return   IsBSTree(t->lchild)&&IsBSTree(t->rchild);
}
```

3-25 设计算法判断给定的二叉树是否为平衡二叉树

```
bool   IsBalanced1(BTREE  *root)
{
    int LeftDepth,RightDepth,diff;
    if(root== NULL)
        return true;
    LeftDepth   = Depth(root->lchild);
    RightDepth = Depth(root->rchild);
    diff = RightDepth - LeftDepth;
    if (diff>1 || diff<-1)
        return false;
   return IsBalanced1(root->lchild)&&IsBalanced(root->rchild);
}
```

方法二：由于上述方法在求该结点的的左右子树深度时遍历一遍树，再次判断子树的平衡性时又遍历一遍树结构，造成遍历多次。因此方法二是一边遍历树一边判断每个结点是否具有平衡性。

```
bool  IsBalanced2(BTREE  *root, int *depth)
{    bool  bLeft, bRight;
     int  LeftDepth, RightDepth, diff;
     if(root== NULL)    { *depth = 0;   return true; }
     bLeft   = IsBalanced2(pRoot->lchild, &LeftDepth);
     bRight = IsBalanced2(pRoot->rchild, &RightDepth);
     if (bLeft && bRight)
     {    diff = RightDepth - LeftDepth;
          if (diff<=1 || diff>=-1)
          {    *depth = 1+(LeftDepth > RightDepth ? LeftDepth : RightDepth);
               return  true;
          }
     }
    return false;
}
```

```
bool  IsBalanced2(BTREE   *root)
{
   int depth = 0;
   return  IsBalanced2(root, &depth);
}
```

3-26

对于给定的一个排好序的整数序列，设计一个算法构造一棵二叉树，使得在该二叉树中，以任意结点为根的子树的高度之差的绝对值不大于1。

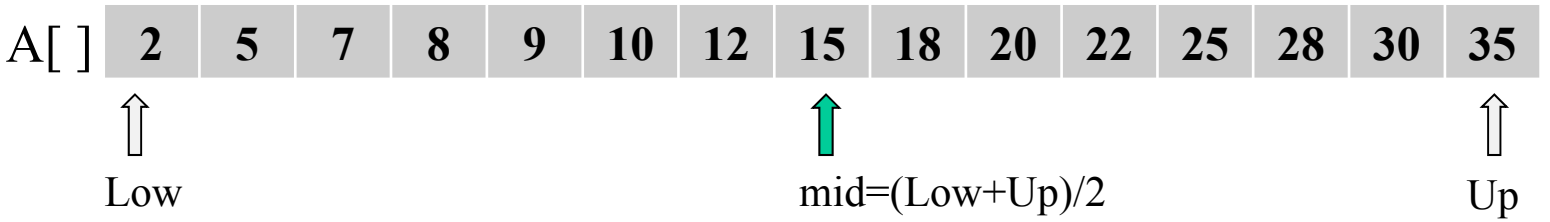
分析：

1）已知条件“排好序” 的整数序列？

2）构造二叉树？二叉排序树/二叉查找树？

3）任意结点为根的子树，左右子树的高度差不超过 | $D_l - D_r$ | <=1, 平衡

例如：



T=CreateTreeFromArray(A , Low, Up)

```
T = new BNODE;
T = a[mid];  root->rchild =root->lchild = NULL;
T -> lchild =CreateTreeFromArray(a, Low ,mid-1);
T -> rchild =CreateTreeFromArray(a, mid + 1 ,Up);
```

```
BTREE   *CreateTreeFromArray(int a[], int Low, int Up)
 {  BTREE *root;
    int mid;
    if (Low > Up)
          return NULL;
    else
    {     mid = (Low + Up) / 2;
          root=new BNODE;
          root->data = a[mid];
          root->rchild = NULL;  root->lchild = NULL;
          root->lchild=CreateTreeFromArray(a, Low ,mid -1);
          root->rchild=CreateTreeFromArray(a, mid+1 ,Up);
          return root;
    }
}
```

判断T1是否为T的子二叉树问题？

3-27

分析：

1）在T中查找结点T2，满足T2->data==T1->data；

2）判断T2与T1是否等价（在根结点相同前提下，判等价算法可以简化）

注意在T中可能还有多个T1结点

问题：有两棵很大的二叉树：设计算法判断T2是否为T1的子树。
思路：先在T1中找到T2的根结点，然后依次去匹配它们的左右子树即可。
注意：在T1中查找T2的根结点时，如果找到与T2匹配的子树，则返回真值；否则，还要继续查找， 直到在T1中找到一棵匹配的子树或是T1中的结点都查找完毕。

```
bool Match(BTREE  *r1, BTREE *r2)       //匹配左右子树
{   if(r1 == NULL && r2 == NULL)
        return true;
    else if(r1 == NULL || r2 == NULL)
       return false;
    else if(r1->key != r2->key)
      return false;
    else
       return Match(r1->lchild, r2->lchild) && Match(r1->rchild, r2->rchild);
}
bool SubTree(BTREE  *r1, BTREE *r2)   //寻找T1中和T2的根相同的节点，找到了，匹配左右子树，
{                                     //没找到，分别向T1的左右子树寻找
   if(r1 == NULL)
         return false;                //说明没有在T1中找到想要的根节点
   else if(r1->key == r2->key)
    {    if(Match(r1, r2))
            return true;
       else
           return SubTree(r1->lchild, r2) || SubTree(r1->rchild, r2);
    }
   else
        return SubTree(r1->lchild, r2) || SubTree(r1->rchild, r2);
}
```

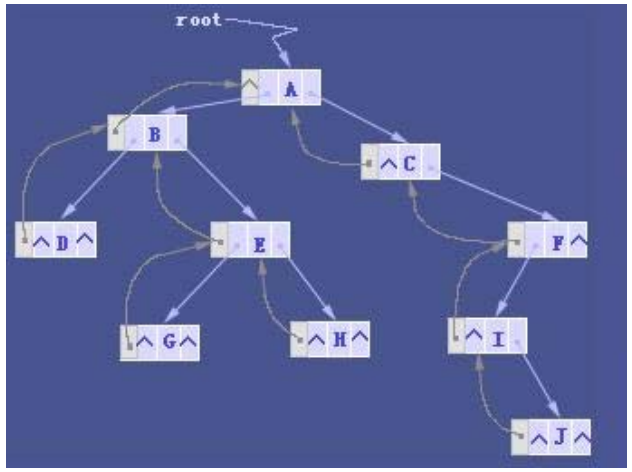
调用：

```
bool ContainTree(BTREE *r1, BTREE *r2)
{   if(r2 == NULL)
        return true;
    else
        return SubTree(r1, r2);
}
```

注意三叉链表的应用

3-28

三叉链表是二叉树的另一种主要的链式存储结构。三叉链表与二叉链表的主要区别在于，它的结点比二叉链表的结点多一个指针域，该域用于存储一个指向本结点双亲的指针。



由于有了指向父节点的指针，实现非递归遍历不再需要使用栈结构。

用三叉链表作二叉数的存储结构，当二叉树有n个结点时，有多少个空指针

注意三叉树的应用

```
void PostByParent(BTREE *T)
{   BTREE *pre, *cur;
    cur=T;
   while(cur)
   {   while(cur->lchild)   cur = cur->lchild;   //遇到左子结点，则一直向左
       if(cur->rchild)                    //如果结点有右子树进入右子树
           cur = cur->rchild;
       else                               //否则，输出并向上回溯
       {   printf("1%c,",cur->data);
           pre = cur;     cur = cur->parent;
          while(cur)
           {   if(cur->lchild == pre)          //如果回溯时，是从左子树回溯的
               {   if(cur->rchild)
                   {   cur = cur->rchild;
                       break;                  //break后回到第6行
                    }
                }
               else                            //从右子树回溯的，则继续向上多回溯一个
               {   printf("2%c,",cur->data);     //从右子树回溯,要输出父结点
                   pre = cur;     cur = cur->parent;
               }
           }
        }
   } //while(cur!=T);
}
```

图

（一）图的概念
（二）图的存储及基本操作
　　1、邻接矩阵法
　　2、邻接表法
（三）图的遍历
　　1、深度优先搜索
　　2、广度优先搜索
（四）图的基本应用
　　1、最小（代价）生成树
　　2、最短路径
　　3、拓扑排序
　　4、关键路径

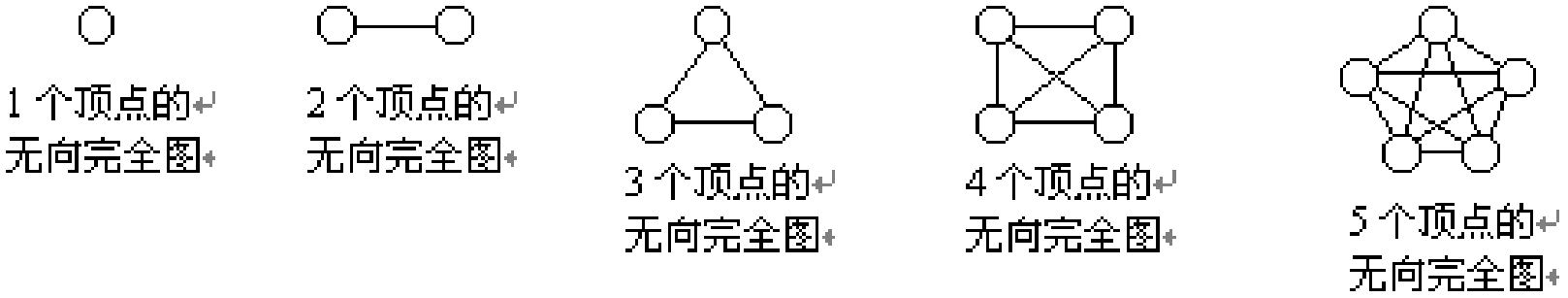
基本操作：

1、图的路径问题
　(1) 无向图两点之间是否有路径存在？
　(2) 有向图两点之间是否有路径存在？
　(3) 如果有路径，路径经过哪些顶点？

2、图的环路问题
　(1) 无向图是否存在环路？
　(2) 有向图是否存在环路？
　(3) 有几条环路？
　(4) 环路经过哪些点，环路轨迹是什么？

4-1

画出1个顶点、2个顶点、3个顶点、4个顶点和5个顶点的无向完全图。试证明在n个顶点的无向完全图中，边的条数为：n(n-1)/2。



【证明】

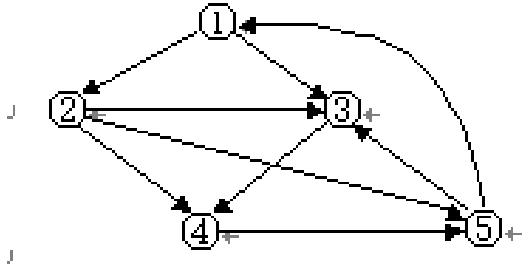
在有n个顶点的无向完全图中，每一个顶点都有一条边与其它某一顶点相连，所以每一个顶点有n-1条边与其他n-1个顶点相连，总计n个顶点有n(n-1)条边。但在无向图中，顶点i到顶点j与顶点j到顶点i是同一条边，所以总共有n(n-1)/2条边。

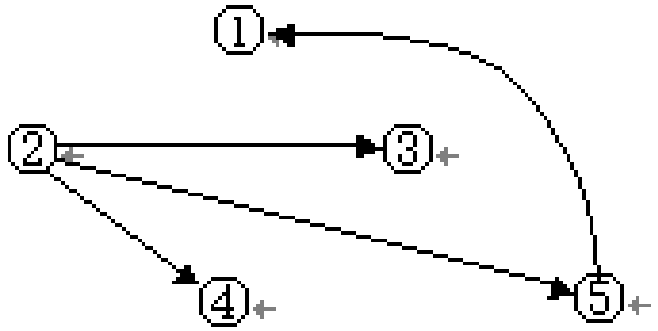
4-2

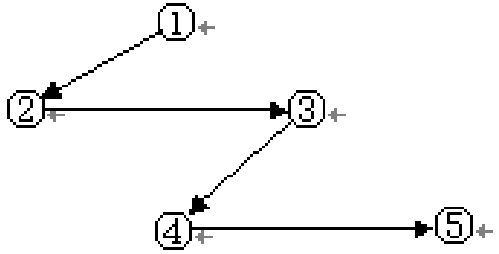
对于如图所示的有向图，试写出：

(1)从顶点②出发进行广度优先搜索所得到的广度优先生成树；

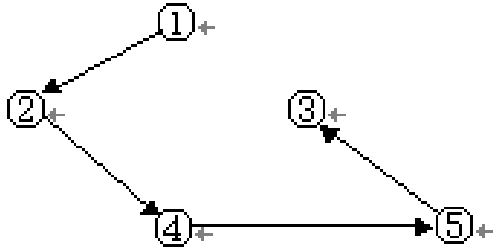
(2)从顶点①出发进行深度优先搜索所得到的深度优先生成树；







或



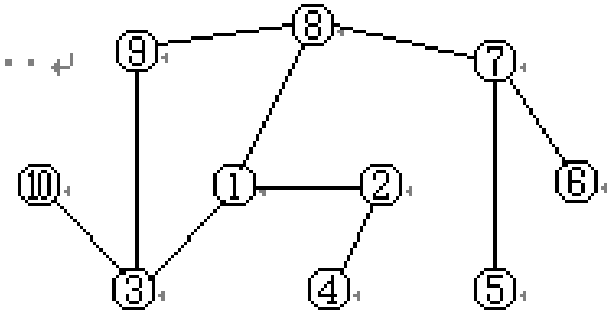
4-3

右图是一个连通图，请画出

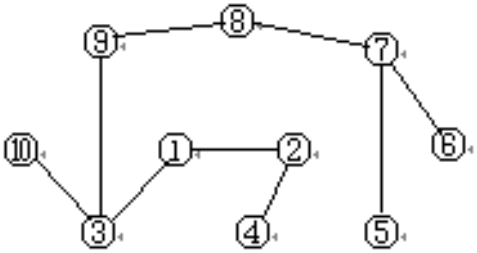
(1) 以顶点①为根的深度优先生成树；

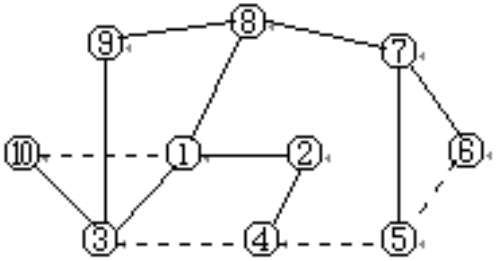
(2) 如果有关节点，请找出所有的关节点。

(3) 如果想把该连通图变成重连通图，至少在图中加几条边？



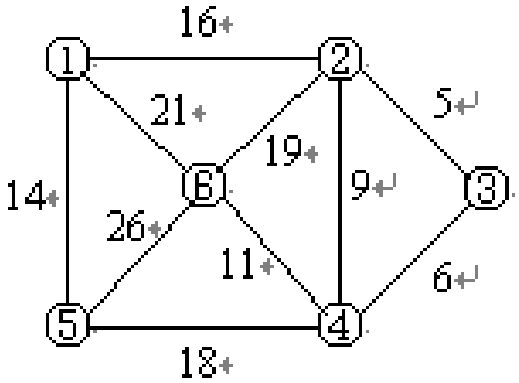
关节点为 ①, ②, ③, ⑦, ⑧

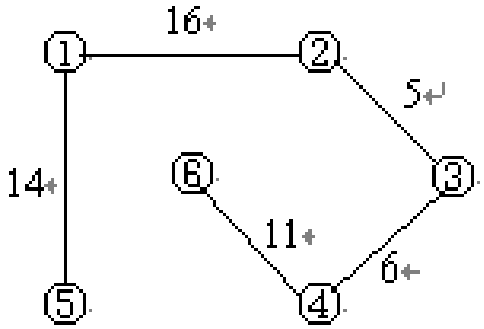




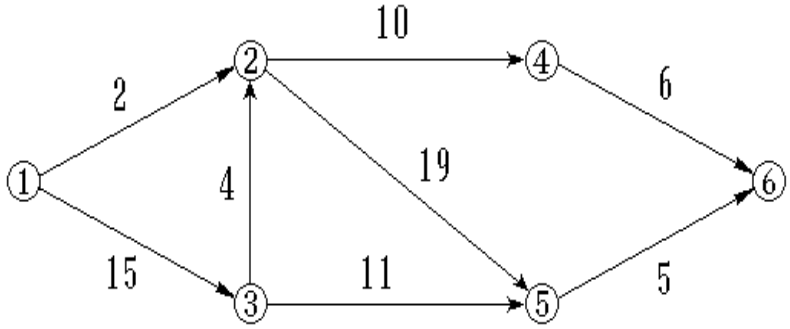
(3) 至少加四条边 (1, 10), (3, 4), (4, 5), (5, 6)。从③的子孙结点⑩到③的祖先结点①引一条边，从②的子孙结点④到根①的另一分支③引一条边，并将⑦的子孙结点⑤、⑥与结点④连结起来，可使其变为重连通图。

4-4 求最小生成树





4-5 给出所有的拓扑序列



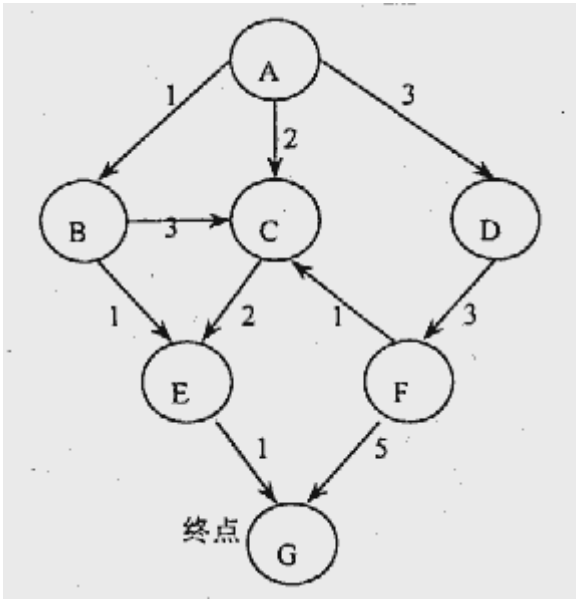
1、3、2、4、5、6

1、3、2、5、4、6

4-6

求单源最短路径问题

如右图所示，设源点为A，求源点到其它各个顶点的最短路径

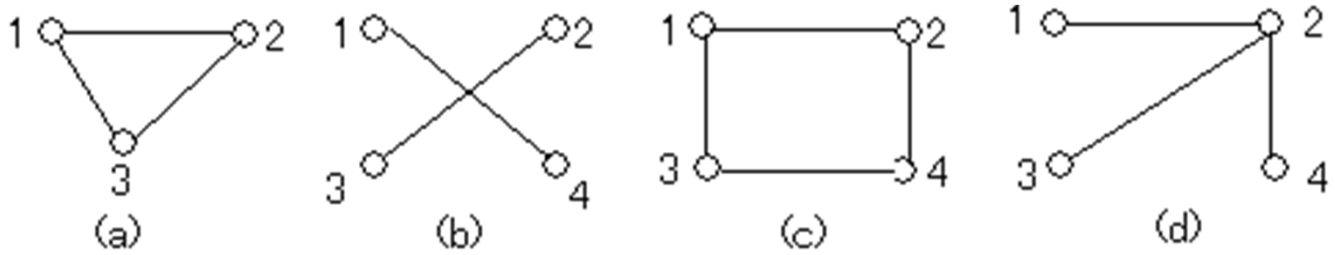


结果如下表所示

step	S	w	D:B	D:C	D:D	D:E	D:F	D:G
0	{A}	-	1	2	3	∞	∞	∞
1	{AB}	B		2	3	2	∞	∞
2	{ABC}	C			3	2	∞	∞
3	{ABCE}	E			3		∞	3
4	{ABCED}	D					6	3
5	{ABCEDG}	G					6	
6	{ABCEDGF}	F	1	2	3	2	6	3

4-7

在如图所示的各无向图中：
(1)找出所有的简单环。
(2)哪些图是连通图?对非连通图给出其连通分量。
(3)哪些图是自由树(或森林)?



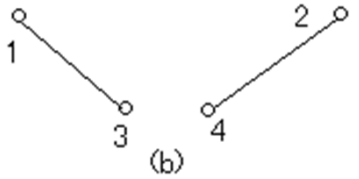
(1)所有的简单环：(同一个环可以任一顶点作为起点)

(a)1231　　(b)无

(c)12431　　(d)无

(2) (a)、(c)、(d)是连通图，

(b)不是连通图，因为从1到2没有路径。具体连通分量为：



(3)自由树(森林)：自由树是指没有确定根的树，无回路的连通图称为自由树：

(a)不是自由树，因为有回路。

(b)是自由森林，其两个连通分量为两棵自由树。

(c)不是自由树。　　(d)是自由树。

4-8

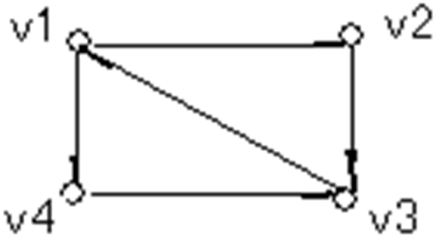
在右图所示的有向图中：

(1) 该图是强连通的吗? 若不是，则给出其强连通分量。

(2) 请给出所有的简单路径及有向环。

(3) 请给出每个顶点的度，入度和出度。

(4) 请给出其邻接表、邻接矩阵及逆邻接表。



(1)该图是强连通的，所谓强连通是指有向图中任意顶点都存在到其他各顶点的路径。

(2)简单路径是指在一条路径上只有起点和终点可以相同的路径：

有v1v2、v2v3、v3v1、v1v4、v4v3、v1v2v3、v2v3v1、v3v1v2、v1v4v3、v4v3v1、v3v1v4、

另包括所有有向环,有向环如下：

v1v2v3v1、v1v4v3v1(这两个有向环可以任一顶点作为起点和终点)

(3)每个顶点的度、入度和出度：

D(v1)=3　ID(v1)=1　OD(v1)=2

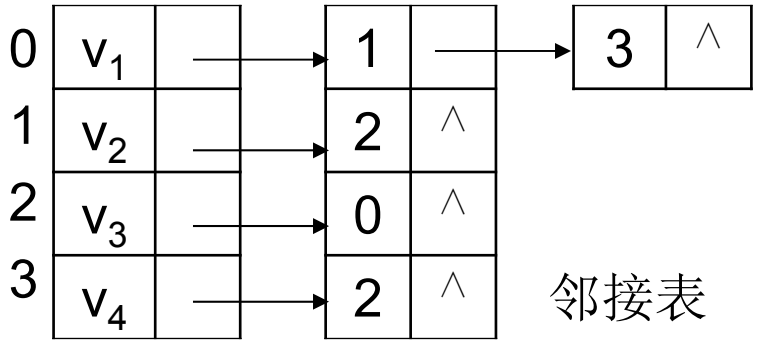
D(v2)=2　ID(v2)=1　OD(v2)=1

D(v3)=3　ID(v3)=2　OD(v3)=1

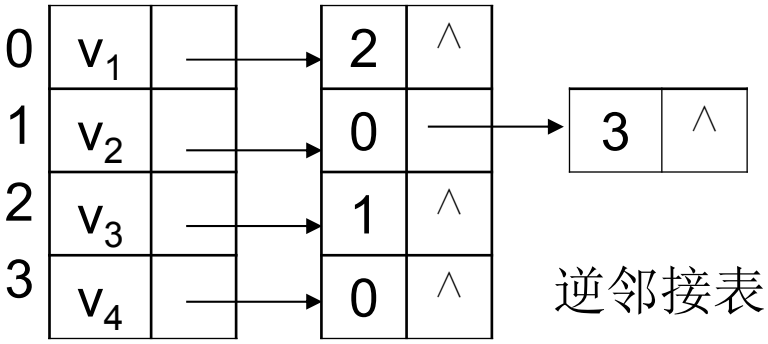
D(v4)=2　ID(v4)=1　OD(v4)=1

(4)邻接表：

vertex firstedge next



邻接表



逆邻接表

邻接矩阵:

0	1	0	1
0	0	1	0
1	0	0	0
0	0	1	0

4-9

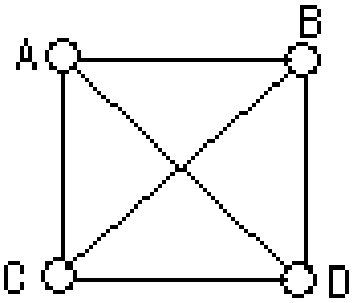
假设图的顶点是A, B. . . ,
请根据下述的邻接矩阵画出相应的无向图或有向图。

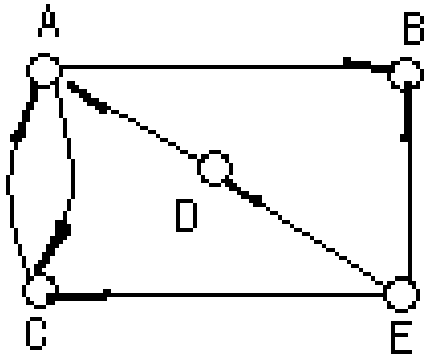
0	1	1	1
1	0	1	1
1	1	0	1
1	1	1	0

(a)

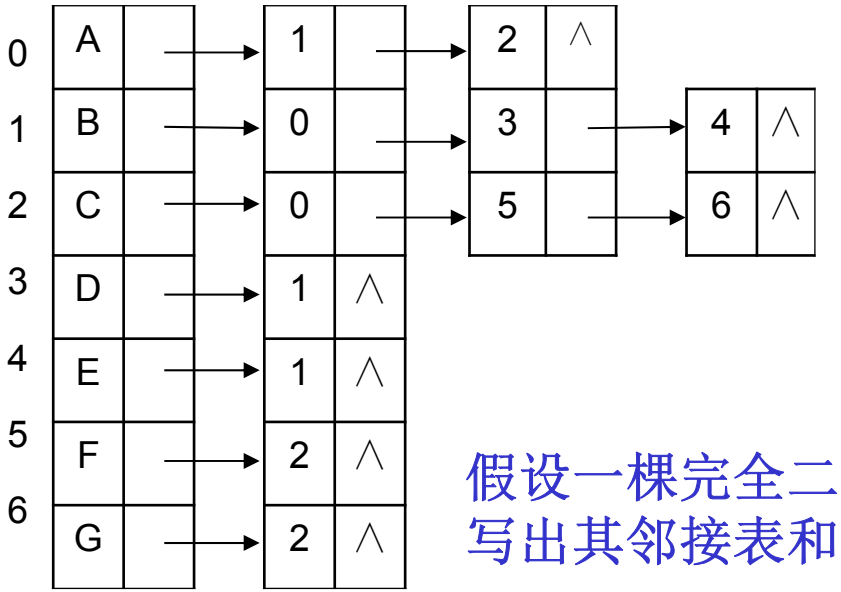
0	1	1	0	0
0	0	0	1	0
0	0	0	1	0
1	0	0	0	1
0	1	0	1	0

(b)





邻接表：



邻接矩阵：

0	1	1	0	0	0	0
1	0	0	1	1	0	0
1	0	0	0	0	1	1
0	1	0	0	0	0	0
0	1	0	0	0	0	0
0	0	1	0	0	0	0
0	0	1	0	0	0	0

假设一棵完全二叉树包括**A,B,C...**等七个结点，
写出其邻接表和邻接矩阵。

3-10

对 n 个顶点的无向图和有向图，采用邻接矩阵和邻接表表示时，如何判别下列有关问题?

(1) 图中有多少条边?

(2) 任意两个顶点 i 和 j 是否有边相连?

(3) 任意一个顶点的度是多少?

对于 n 个顶点的无向图和有向图，用邻接矩阵表示时：

(1) 设m为矩阵中非零元素的个数

无向图的边数 = m/2

有向图的边数 = m

(2) 无论是有向图还是无向图，在矩阵中第 i 行,第 j 列的元素若为非零值，则该两顶点有边相连。

(3) 对于无向图，任一顶点 i 的度为第 i 行中非零元素的个数。

对于有向图，任一顶点 i 的入度为第 i 列中非零元素的个数，出度为第 i 行中非零元素的个数，度为入度出度之和。

当用邻接表表示时：

(1)对于无向图，图中的边数=边表中结点总数的一半。
对于有向图，图中的边数=边表中结点总数。

(2)对于无向图，任意两顶点间是否有边相连，可看其中一个顶点的邻接表，若表中的adjvex域有另一顶点位置的结点，则表示有边相连。
对于有向图，则表示有出边相连。

(3)对于无向图，任意一个顶点的度则由该顶点的边表中结点的个数来决。
对于有向图，任意一个顶点的出度由该顶点的边表中结点的个数来决定，入度则需遍历各顶点的边表。
(用逆邻接表可容易地得到其入度。)

4-11

n个顶点的连通图至少有几条边？ 强连通图呢？

n个顶点的连通图至少有n-1条边，
强连通图至少有2(n-1)条边。

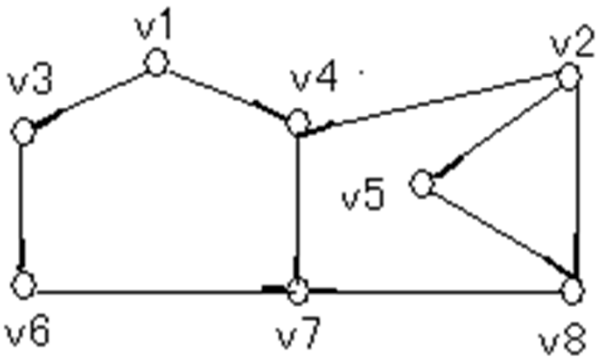
4-12

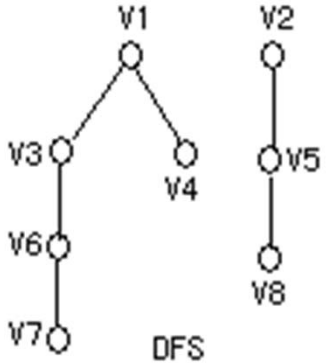
DFS和BFS遍历各采用什么样的数据结构来暂存顶点?当要求连通图的生成树的高度最小，应采用何种遍历?

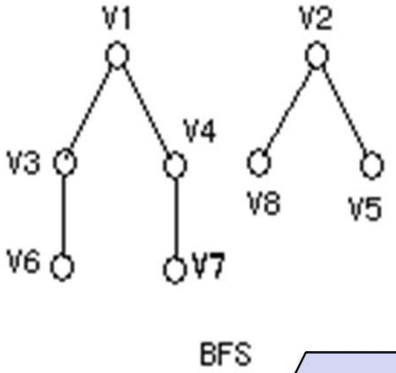
DFS遍历采用栈来暂存顶点。BFS采用队列来暂存顶点。
当要求连通图的生成树的高度最小时，应采用BFS遍历。

4-13

画出以顶点v1为初始源点遍历下图所示的有向图所得到的DFS 和BFS生成森林。

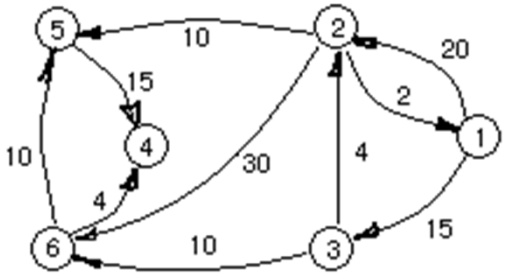






4-14

对右图所示的有向图，试利用Dijkstra算法求出从源点1到其它各顶点的最短路径，并写出执行算法过程中扩充红点集的每次循环状态



循环	红点集	K	D[i]						P[i]					
			1	2	3	4	5	6	1	2	3	4	5	6
初始化	{1}	-	0	20	15	∞	∞	∞	-1	1	1	-1	-1	-1
1	{1,3}	3	0	19	15	∞	∞	25	-1	3	1	-1	-1	3
2	{1,3,2}	2	0	19	15	∞	29	25	-1	3	1	-1	2	3
3	{1,3,2,6}	6	0	19	15	29	29	25	-1	3	1	6	2	3
4	{1,3,2,6,4}	4	0	19	15	29	29	25	-1	3	1	6	2	3
5	{1,3,2,6,4,5}	5	0	19	15	29	29	25	-1	3	1	6	2	3
6	-	-	-	-	-	-	-	-	-	-	-	-	-	-

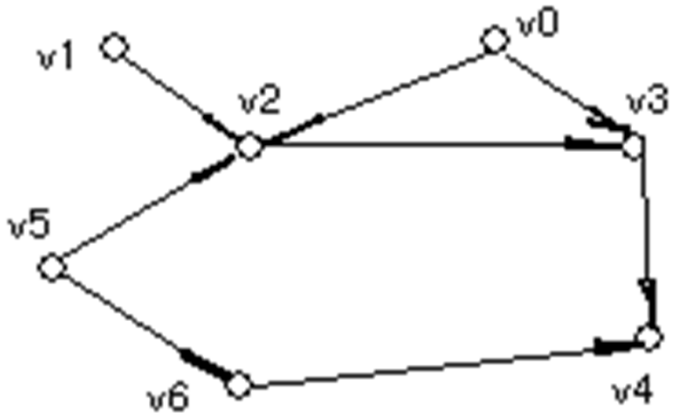
从源点1到各点的路径如下：

1到2：132　　1到3：1　　1到4：1364　　1到5：1325　　1到6：136

4-15 **什么样的DAG的拓扑序列是唯一的?**

确定了排序的源点，DAG图中无前趋顶点只有一个且从该点到终点只有一条路径时，它的拓扑序列才是唯一的。

请以V_0为源点，给出用DFS搜索下图得到的逆拓扑序列。 3-16



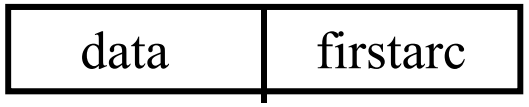
逆拓扑序列是：V4 V2 V1 V0 V1 V6 V5

图的邻接表类型定义

```
#define   MAX_VERTEX_NUM   20
Typedef   enum { DG, DN, AG, AN }  GraphKind ;

Typedef   struct   ArcNode  {
          int                    adjvex ;
          struct   ArcNode   *nextarc ;
          InfoType             *info ;
}  ArcNode ;

Typedef   struct  Vnode  {
          VertexType      data ;
          ArcNode          *firstarc ;
}  Vnode, AdjList[MAX_VERTEX_NUM] ;

Typedef   struct {
          AdjList     vertices ;
          Int           vexnum ;
          Int           kind ;
}  ALGraph ;
```

表结点

头结点



例：

图类型变量：ALGraph　G ;

顶点个数：　　G.vexnum

图的类型：　　G.kind = (DG,DN,AG,AN)

顶点 i 信息：　　G.vertices[i].dada

顶点 i 的第一个邻接点:

G.vertices[i].firstarc->adjvex

G.vertices[G.vertices[i].firstarc->adjvex].data

G.vertices[i].firstarc->info

顶点 i 的第二个邻接点：

G.vertices[i].firstarc->nextarc->adjvex

图中常用的集合的表示方法：

1、顶点集S

$$S[u]=\begin{cases}1 & u\in S\\ 0 & u\notin S\end{cases} \qquad u\in V,\ u=1,2,\ldots,n$$

例如：*for(i=1;i<=n;i++) if(S[i]) ……*

2、边/弧集T

$$T[u][v]=\begin{cases}1 & (u,v)\in T\\ 0 & (u,v)\notin T\end{cases} \qquad u,v\in V,\ u,v=1,2,\ldots,n$$

4-17 利用拓扑排序算法的思想写一算法判别有向图中是否存在有向环，当有向环存在时，输出构成环的顶点。

```
typedef enum{FALSE，TRUE}Boolean；  //FALSE为0，TRUE为1
Boolean visited[MaxVertexNum]；     //访问标志向量是全局量
int i；

for(i=0;i<G->n;i++)
    visited[i]=FALSE；              //标志向量初始化
```

以邻接表作为存储结构

```
void NonSuccFirstTopSort(ALGraph G)
{                                          //优先输出无后继的顶点,此处用逆邻接表存储
   int outdegree[MaxVertexNum];     //出度向量，此处MaxVertexNum>=G.n
   SeqStack S;                              //将栈中data向量的基类型改为int
   int i,j,count=0;                         //count对输出的顶点数目计数，初值为0
   EdgeNode *p;
   for(i=0;i<G.n;i++)
     { outdegree[i]=0;
       visited[i]=FALSE;                    //标志向量初始化
     }

   for(i=0;i<G.n;i++)
      for(p=G.adjlist[i].firstedge;p;p=p->next)          //扫描i的入边表
            outdegree[p->adjvex]++; //设p->adjvex=j，则将<j,i>的起点j出度加1
   InitStack(&s);
```

```
for(i=0;i<G.n;i++)
  if (outdegree[i]==0) Push(&S, i);    //出度为0的顶点i入栈
while(!StackEmpty(S))                  //栈非空，有出度为0的顶点
{  i=pop(&s);visited[i]=TRUE;
   count++;                            //顶点计数加1
   for(p=G.adjlist[i].firstedge;p;p=p->next)
                                       //修改以i为弧头的弧的弧尾顶点的出度
    { j=p->adjvex;
      outdegree[j]--;
      if(outdegree[j]==0)              //将新生成的出度为0的顶点入栈
        Push(&S, j);                   //出度为0的顶点j入栈
    }                                  //end of for
}                                      //end of while
if (count<G.n)                         //输出顶点数小于n
    {  printf("G中存在有向环，排序失败!");
       for(i=0;i<G.n;i++)
          if (visited[i]==FALSE)
             printf("%c",G.adjlist[i].vertex);
    }
else printf("G中无有向环!");
}
```

4-18 写一算法求有向图的所有根(若存在)，分析算法的时间复杂度。

```
typedef enum{FALSE，TRUE}Boolean；//FALSE为0，TRUE为1
Boolean visited[MaxVertexNum]；          //访问标志向量是全局量
void DFSTraverse(ALGraph *G)
{                                        //对以邻接表表示的有向图G，求所有根
   int i,j；
   for (j=0;j<G->n;j++)
   {  for(i=0;i<G->n;i++)
         visited[i]=FALSE；  //标志向量初始化
      DFS(G，j)；             //以vj为源点开始DFS搜索，也可用BFS(G,j)
      i=0;
      while(i<G->n)&&(visited[i]==TRUE)     i++;
      if (i==G->n) printf("root:%c",G->adjlist[j].vertex);
   }
} //DFSTraverse
```

该算法的为二重循环，若调用的DFS算法的复杂度为O(n+e), 该算法的时间复杂度为O(n(n+e)),调用的DFSM算法的复杂度为O(n*n),所以该算法的时间复杂度为$O(n^3)$

判断是否存在从u到v的路径


```
int  ExistPathDfs1(ALGraph G,int *visited,int u,int v)
//实现1：判断是否存在从u到v的路径，返回1或0
{
   ArcNode *p;
   int w;
   if(u==v)     return 1;
   else
    {
        visited[u]=1;                    //访问标志
        for(p=G.vertices[u].firstarc;p;p=p->nextarc)
        {
            w=p->adjvex;
            if(!visited[w]&&ExistPathDfs1(G,visited,w,v))
            return 1;
         }//for
    }//else
    return(0);
}//ExistPathDfs1
```

```
int ExistPathDfs2(ALGraph G,int *visited,int u,int v)
//实现2：判断u到v是否有通路,返回1或0
{    ArcNode *p;
     int w;
     int static flag=0;
     visited[u] = 1;                    //访问标志
     p = G.vertices[u].firstarc;        //第一个邻接点
     while(p!=NULL)
     {    w = p->adjvex;
          if(v==w)
          {   flag = 1;
              return (1);    }          // u和v有通路
           if(!visited[w])
          ExistPathDfs2(G,visited,w,v);
          p=p->nextarc;
      }//while
      if(!flag)  return(0) ;
}  //ExistPathDfs2
```

求u到v所有简单路径

```
int   FindAllPath(ALGraph G,int *visited,int *path,int u,int v,int k)
 {    ArcNode *p;    int static paths=0;      //paths控制指输出第几条有效路径
      int n,i;
      path[k]=u;        visited[u]=1;
      if(u==v)
      {   if(path[1])
          {   if(!paths)  printf("找到如下路径：\n");
              paths++;
              printf("路径%d：  %d",paths,path[0]);
              for(i=1;path[i];i++)  printf("--%d",path[i]);   printf("\n");   }
       }
      else
          for(p=G.vertices[u].firstarc;p;p=p->nextarc)
          {   n=p->adjvex;
             if(!visited[n])
             FindAllPath(G,visited,path,n,v,k+1);      }
             for(i=1;i<=G.vexnum;i++)
             {     visited[i]=0;
                   path[i]=0;       }
      return(paths);
}
```

判断图是否存在包含顶点u的回路

4-21

```
int Cycle(ALGraph G, int *visited,int u)
{   ArcNode *p;
    int w, flag =0;
    visited[u]=1;
    p=G.vertices[u].firstarc;
    while(p&&!flag)
    {    w=p->adjvex;
         if(visited[w]!=1)  //w未访问过(0)或访问且其邻接点已访问完(2)
         {     if(!visited[w])  //w未访问过，从w开始继续dfs
               flag=Cycle(G,visited,w);
         }
         else                    //顶点w已经访问过，并且其邻接点已经访问完
               flag=1;
          p=p->nextarc;
     }
     visited[u]=2;
     return(flag);
}
```

判断图是否有回路存在

```
void IsCycle(ALGraph G)        {
      int visited[MAX_VERTEX_NUM],u, CycleFlag;
      for(u=1;u<=G.vexnum;u++)
           visited[u]=0;
      for(u=1;u<=G.vexnum;u++)
           if(!visited[u])
           {
               CycleFlag=Cycle(G,visited,u);
               if(CycleFlag)  break;
           }
      if(CycleFlag)
            printf("图中存在回路！ \n");
      else
           printf("图中不存在回路！ \n");
}
```

4-23 设计一个找无环路有向图每对顶点间“最长简单路径”（所谓“最长简单路径” 是指该简单路径包含边数最多）的算法，即以一个无环路有向图作为输入，对于每对顶点，如果它们之间存在简单路径，则输出其中最长的，否则输出空。

4-24 可以使用“破圈法”求解带权连通无向图的一棵最小生成树。所谓“破圈法”就是任取一个圈并去掉圈上权最大的边，反复执行这一步骤，直到没圈为止。请设计该算法求解给定带权连通无向图的最小生成树。（注：圈即为环路）

已知图的存储结构（邻接表、邻接矩阵），
求：BST, 拓扑序列，最短路径，关键路径等。
还要注意邻接矩阵可能会以压缩存储方式给出。

在图这一章中，几个重要算法往往都给出了两种形式，要注意他们的区别！

查找

（一）查找的基本概念

（二）顺序查找法

（三）折半查找法

（四）动态查找

（五）B-树及其基本操作、

B+树的基本概念

（六）散列（hash）表

（七）查找算法的分析及应用

5-1

将序列13，15，22，8，34，19，21插入到一个初始时为空的哈希表，哈希函数采用H(x)=1+(x%7)
(1)使用线性探测法解决冲突
(2)使用步长为3的线性探测法解决冲突
(3)使用再哈希法解决冲突，冲突时的哈希函数H(x)=1+(x%6)

解：设哈希表长度为8。

(1) 使用线性探测法解决冲突，即步长为1，对应地址为：

H(13)=1+(13%7)=7
H(15)=1+(15%7)=2
H(22)=1+(22%7)=2(冲突)，　$H_1(22)=(2+1)\%8=3$
H(8)=1+(8%7)=2(冲突)，　$H_1(8)=(2+1)\%8=3$　(仍冲突)
　　$H_2(8)=(3+1)\%8=4$
H(34)=1+(34%7)=7(冲突)，　$H_1(34)=(7+1)\%8=0$
H(19)=1+(19%7)=6
H(21)=1+(21%7)=1

哈希表

地址	0	1	2	3	4	5	6	7
Key	34	21	15	22	8		19	13
探测次数	2	1	1	2	3		1	1

(2) 使用步长为3的线性探测法解决冲突，对应地址为：

H(13)=1+(13%7)=7
H(15)=1+(15%7)=2
H(22)=1+(22%7)=2(冲突)，　$H_1(22)=(2+3)\%8=5$
H(8)=1+(8%7)=2　(冲突)，　$H_1(8)=(2+3)\%8=5$　(仍冲突)
　　$H_2(8)=(5+3)\%8=0$
H(34)=1+(34%7)=7(冲突)，　$H_1(34)=(7+3)\%8=2$ (仍冲突)
　　$H_2(34)=(2+3)\%8=5$ (仍冲突)
　　$H_3(34)=(5+3)\%8=0$ (仍冲突)
　　$H_4(34)=(0+3)\%8=3$
H(19)=1+(19%7)=6
H(21)=1+(21%7)=1

哈希表

地址	0	1	2	3	4	5	6	7
Key	8	21	15	34		22	19	13
探测次数	3	1	1	5		2	1	1

(3) 使用再哈希法解决冲突， 再哈希函数为H(x)=1+(x%6)

H(13)=1+(13%7)=7
H(15)=1+(15%7)=2
H(22)=1+(22%7)=2(冲突)， H_1(22)=1+(22%6)=5
H(8)=1+(8%7)=2 (冲突)， H_1(8)=1+(8%6)=3
H(34)=1+(34%7)=7(冲突)， H_1(34)=1+(34%6)=5 (仍冲突)
H_2(34)=1+(34%5)=5 (仍冲突)
H_3(34)=1+(34%4)=3 (仍冲突)
H_4(34)=1+(34%3)=2 (仍冲突)
H_5(34)=1+(34%2)=1
H(19)=1+(19%7)=6
H(21)=1+(21%7)=1(冲突)， H_1(21)=1+(21%6)=4

哈希表

地址	0	1	2	3	4	5	6	7
Key		34	15	8	21	22	19	13
探测次数		6	1	2	2	2	1	1

5-2

已知一组记录的键值为{1，9，25，11，12，35，17，29}，请构造一个散列表。

(1)采用除留余数法构造散列函数，线性探测法处理冲突，要求新插入键值的平均探测次数不多于2.5次，请确定散列表的长度m及相应的散列函数。分别计算查找成功和查找失败的平均查找长度。

(2)采用(1)中的散列函数，但用链地址法处理冲突，构造散列表，分别计算查找成功和查找失败的平均查找长度。

解：（1）$ASL_u = \frac{1}{2}\left(1+\frac{1}{1-\alpha}\right) \le 2.5$　　得　$\alpha \le 1/2$

因为　8/m ≤1/2,　　所以 m ≥ 16

取 m=16 , 散列函数为 H(key)=key%13

给定键值序列为:{1,9,25,11,12,35,17,29}

散列地址为 :　d:1,9,12,11,12,9,4,3

用线性探测解决冲突，见下表：

地址	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Key		1		29	17					9	35	11	25	12		
成功探测次数		1		1	1					1	2	1	1	2		
失败探测次数	1	2	1	3	2	1	1	1	1	6	5	4	3			

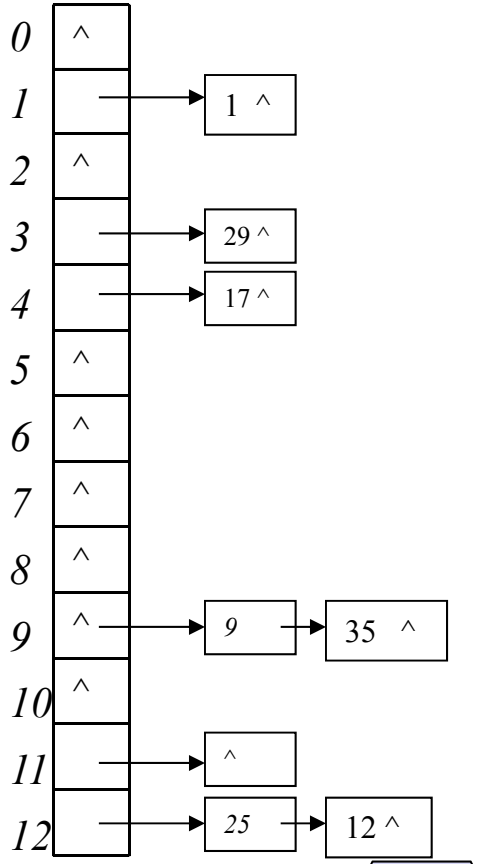
查找成功的平均查找长度：

$ASL_s=(6+4)/8=1.25$

查找失败的平均查找长度：

$ASL_u=(6\times1+2+3+2+6+5+4+3)/13=2.4$

(2)使用链地址法处理冲突构造散列表。



查找成功的平均查找长度：

$ASL_s=(6+4)/8=1.25$

查找失败的平均查找长度：

$ASL_u=(4\times1+2\times2)/13\approx2.4$

5-3

将关键字序列（7，8，30，11，18，9，14）散列存储到散列表中，散列表的存储空间是一个下标从0开始的一维数组，散列函数为：H(key)=(key×3) MOD 7，处理冲突采用线性探测再散列法，要求装填（载）因子为0.7。

(1)请画出所构造的散列表

(2)分别计算等概率情况下查找成功和查找不成功的平均查找长度。

下标	0	1	2	3	4	5	6	7	8	9
关键字	7	14		8		11	30	18	9	

查找成功的平均查找长度：$ASL_{成功}=12/7$

查找不成功的平均查找长度：$ASL_{不成功}=18/7$

对含有 ***n*** 个互不相同元素的集合，
同时找最大元和最小元至少需进行多少次比较?

5-4

设变量*max*和*min*用于存放最大元和最小元(的位置)，第一次取两个元素进行比较，大的放入*max*，小的放入*min*。从第*2* 次开始，每次取一个元素先和*max*比较，如果大于*max*则以它替换*max*，并结束本次比较；若小于*max*则再与*min*相比较，在最好的情况下，一路比较下去都不用和*min*相比较，所以这种情况下，至少要进行 *n-1* 次比较就能找到最大元和最小元。

若对具有 ***n*** 个元素的有序的顺序表和无序的顺序表分别进行顺序查找，试在下述两种情况下分别讨论两者在等概率时的平均查找长度：
(1)查找不成功，即表中无关键字等于给定值 ***K*** 的记录；
(2)查找成功，即表中有关键字等于给定值 ***K*** 的记录。

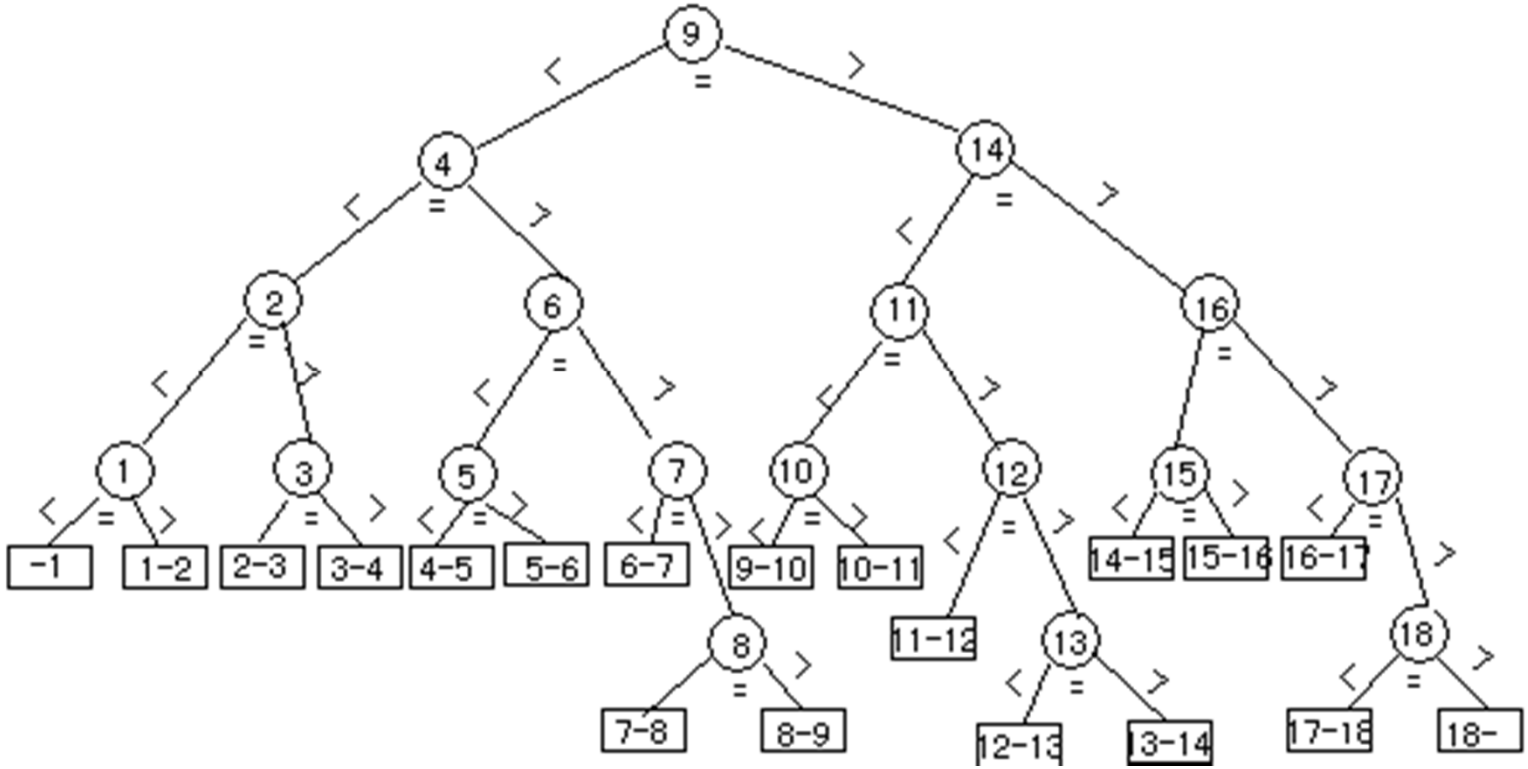
5-5

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

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

5-6

画出对长度为*18*的有序的顺序表进行二分查找的判定树，并指出在等概率时查找成功的平均查找长度，以及查找失败时所需的最多的关键字比较次数。



等概率情况下，查找成功的平均查找长度为：

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

查找失败时，最多的关键字比较次数不超过判定树的深度，此处为*5*.

5-7

设有序表为*(a, b, c, e, f, g, i, j, k, p, q)*,请分别画出对给定值*b*, *g*和*n* 进行折半查找的过程。

(1)查找b的过程如下：

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

(2)g的查找过程如下：一次比较成功。 [a b c d e f (g) h i j k p q]

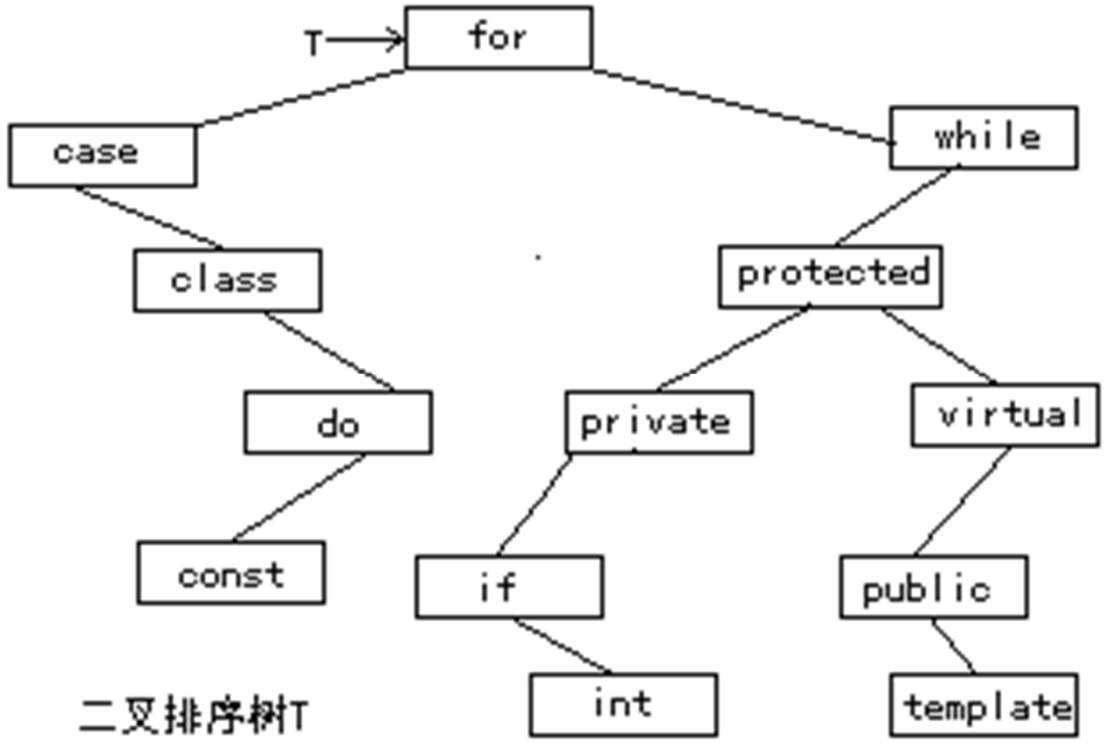
(3)n的查找过程如下： 四次比较，查找失败。

下标	1	2	3	4	5	6	7	8	9	10	11	12	13
第一次比较	[a	b	c	d	e	f	**(g)**	h	i	j	k	p	q]
第二次比较	a	b	c	d	e	f	g	[h	i	**(j)**	k	p	Q]
第三次比较	a	b	c	d	e	f	g	h	i	j	[k	**(p)**	q]
第三次比较	a	b	c	d	e	f	g	h	i	j	[k]	p	q

5-8

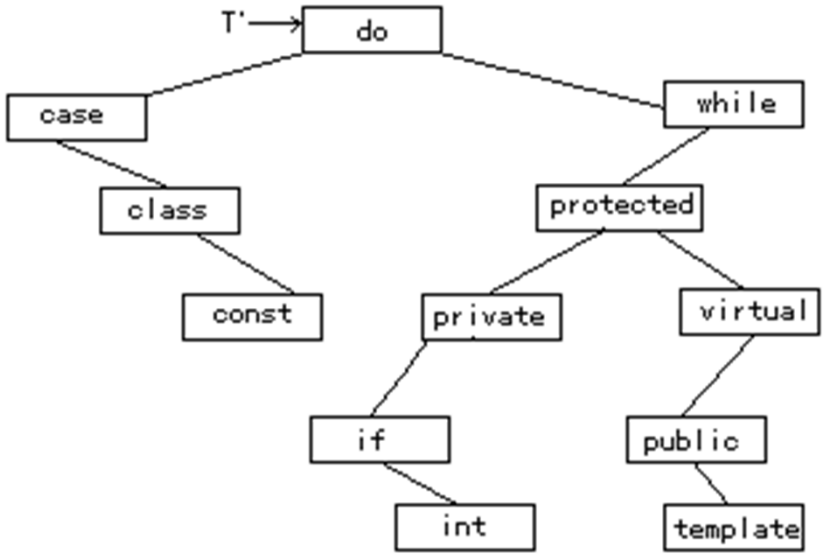
将(*for, case, while, class, protected, virtual, public, private, do, template, const ,if, int*)中的关键字依次插入初态为空的二叉排序树中，请画出所得到的树 *T*。然后画出删去*for*之后的二叉排序树*T'*，若再将*for* 插入T'中得到的二叉排序树*T''*是否与T相同?最后给出*T''*的先序、中序和后序序列。

二叉排序树*T*如下图：

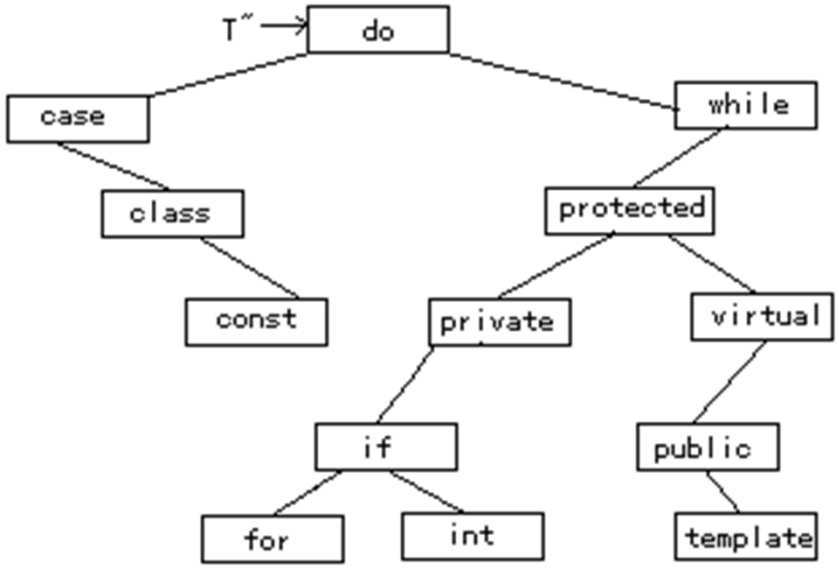


二叉排序树T

删去*for*后的二叉排序树如下图：



再插入结点*for*后的二叉排序树T"：



二叉排序树*T* "与*T*不同

*T"*的先序序列是：*do case class const while protected private if for int virtual public template*

*T"*的中序序列是：*case class const do for if int private protected public template virtual while*

*T"*的后序序列是：*const class case for int if private template public virtual protected while do*

5-9 对给定的关键字集合，以不同的次序插入初始为空的树中，是否有可能得到同一棵二叉排序树?

有可能。
如有两个序列：3，1，2，4 和 3，4，1，2，它们插入空树所得的二叉排序树是相同的。

5-10 设二叉排序树中关键字由1至1000的整数构成，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上查找到的序列?

(a) 2，252，401，398，330，344，397，363；
(b) 924，220，911，244，898，258，362，363；
(c) 925，202，911，240，912，245，363；
(d) 2，399，387，219，266，382，381，278，363.

(c)是不可能查找到的序列。把这四个序列各插入到一个初始为空的二叉排序树中，结果可以发现，(c)序列所形成的不是一条路径，而是有分支的，可见它是不可能在查找过程中访问到的序列。

5-11

设二叉排序树中关键字互不相同，则其中最小元必无左孩子，最大元必无右孩子。此命题是否正确?最小元和最大元一定是叶子吗?一个新结点总是插在二叉排序树的某叶子上吗?

此命题正确。假设最小元有左孩子，则根据二叉排序树性质，此左孩子应比最小元更小，如此一来就产生矛盾了，因此最小元不可能有左孩子，对于最大元也是这个道理。

但最大元和最小元不一定是叶子，它也可以是根、内部结点(分支结点)等，这得根据插入结点时的次序而定。

新结点总是作为叶子插入在二叉排序树中的。

5-12

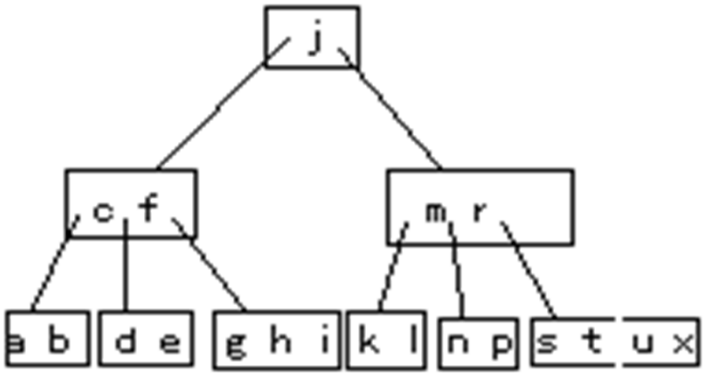
在一棵 m阶的B-树中，当将一关键字插入某结点而引起该结点的分裂时，此结点原有多少个关键字 ？若删去某结点中的一个关键字，而导致结点合并时，该结点中原有几个关键字?

在一棵m阶的B-树中，若由于一关键字的插入某结点而引起该结点的分裂时，则该结点原有m-1个关键字。

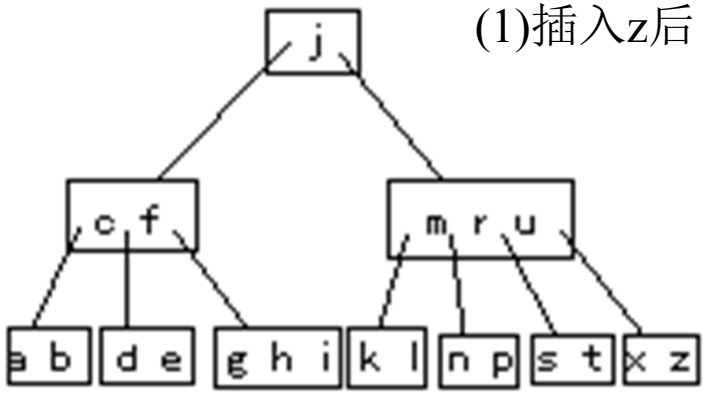
若删去某结点中一个关键字而导致结点合并时，该结点中原有 $\lceil m/2 \rceil - 1$ 个关键字。

5-13

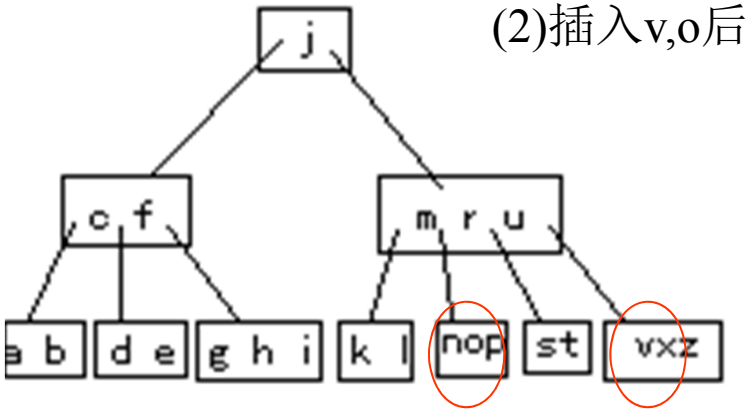
画出依次插入*z, v, o, p, w, y* 到如图所示的*5* 阶 **B-**树的过程。



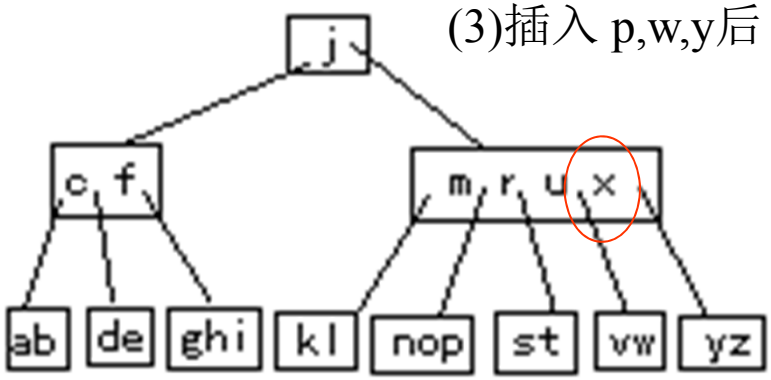
(1)插入z后



(2)插入v,o后



(3)插入 p,w,y后



5-14

设散列表长度为11，散列函数 *h(x)=x%11*，给定的关键字序列为：

1，13，13，34，38，33，27，22.

试画出分别用链地址法和线性探查法解决冲突时所构造的散列表，并求出在等概率情况下，这两种方法查找成功和失败时的平均查找长度。请问装填因子的值是什么？

用链地址法的查找成功平均查找长度为：

*ASLsucc=(1*4+2*3+3*1)/8=1.625*

查找失败时平均查找长度为：

ASLunsucc=(2+3+1+0+0+0+2+0+0+0+0)/11=0.73

用线性探查法查找成功时平均查找长度为：

ASLsucc=(1+1+1+3+4+1+7+8)/8=3.25

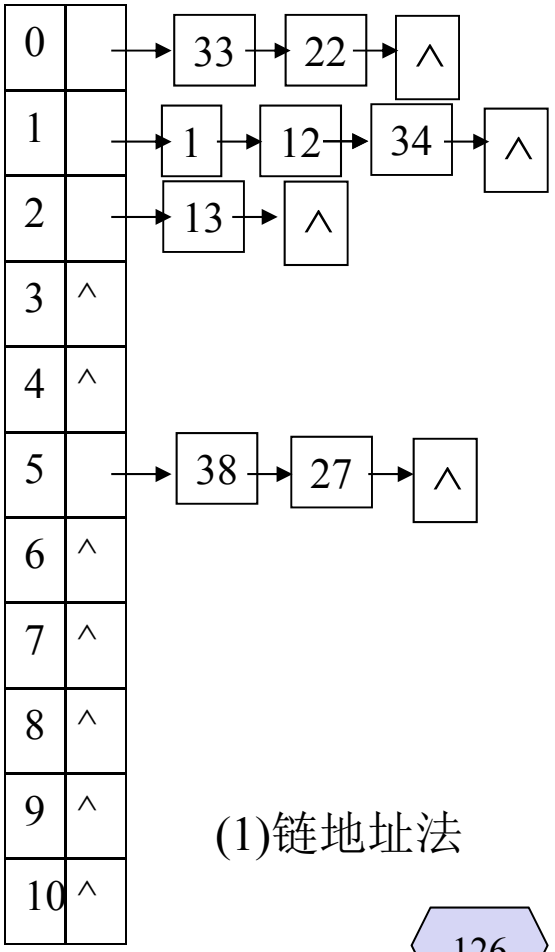
查找失败时平均查找长度为：

ASLunsucc=(9+8+7+6+5+4+3+2+1+1+1)/11=4.3

装填因子$\alpha_{拉链}=4/11=0.36$,$\alpha_{线性探查}=8/11=0.73$

(2)线性探查法如下图：

下标	0	1	2	3	4	5	6	7	8	9	10
T[0..10]	33	1	13	12	34	38	27	22			
探查次数	1	1	1	3	4	1	7	8			



(1)链地址法

5-15 设顺序表中关键字是递增有序的，试写一顺序查找算法，将哨兵设在表的高下标端。然后求出等概率情况下查找成功与失败时的*ASL*.

```
typedef struct{
        KeyType key;
        InfoType otherinfo;  //此类型依赖于应用
    }NodeType;
 typedef NodeType SeqList[n+1];  //n号单元用作哨兵

 int SeqSearch(Seqlist R，KeyType K)
       { //在关键字递增有序的顺序表R[0..n-1]中顺序查找关键字为K的结点，
          //成功时返回找到的结点位置，失败时返回-1
          int i;
          R[n].key=K;  //设置哨兵
          for(i=0; R[i].key<=K;i--);  //从表前往后找
          if (i<n&&R[i].key==K) return i; //R[i]是要找的结点
          else return -1
       } //SeqSearch
```

等概率情况下查找成功的 $ASL=(1+2+3+...+n)/n$

等概率情况下查找失败时的 $ASL=(1+2+3+...+n+n+1)/(n+1)$

5-16

试写一递归算法，从大到小输出二叉排序树中所有其值不小于*x*的关键字。要求算法的时间为*O(lgn+m)*，*n*为树中结点数，*m*为输出关键字个数。(提示：先遍历右子树，后遍历左子树)。

```
typedef int KeyType;  //假定关键字类型为整数
typedef struct node { //结点类型
        KeyType key;  //关键字项
        InfoType otherinfo;  //其它数据域，InfoType视应用情况而定，下面不处理它
        struct node *lchild, *rchild;  //左右孩子指针
    } BSTNode;
typedef BSTNode *BSTree;

void OUTPUTNODE(BSTree T,KeyType x)
    {//从大到小输出二叉排序树中所有其值不小于x的关键字
        if (T)
            {
                OUTPUTNODE( T->rchild,x);
                if (T->key>=x) printf("%d",T->key);
                OUTPUTNODE( T->Lchild,x);
            }
    }
```

[m-路查找树定义2]

m 分支查找树 *T*，是指其所有结点的度都不大于 *m* 的查找树。

空树 *T* 是一株 *m* 分支查找树。当 *T* 非空时，它有下述性质：

（1）*T* 是形式如下的一个结点：

$n, A_0, (K_1, A_1), (K_2, A_2), \ldots\ldots, (K_n, A_n)$

其中，$A_i (0 \le i \le n < m)$ 是指向 *T* 之子树的指针，

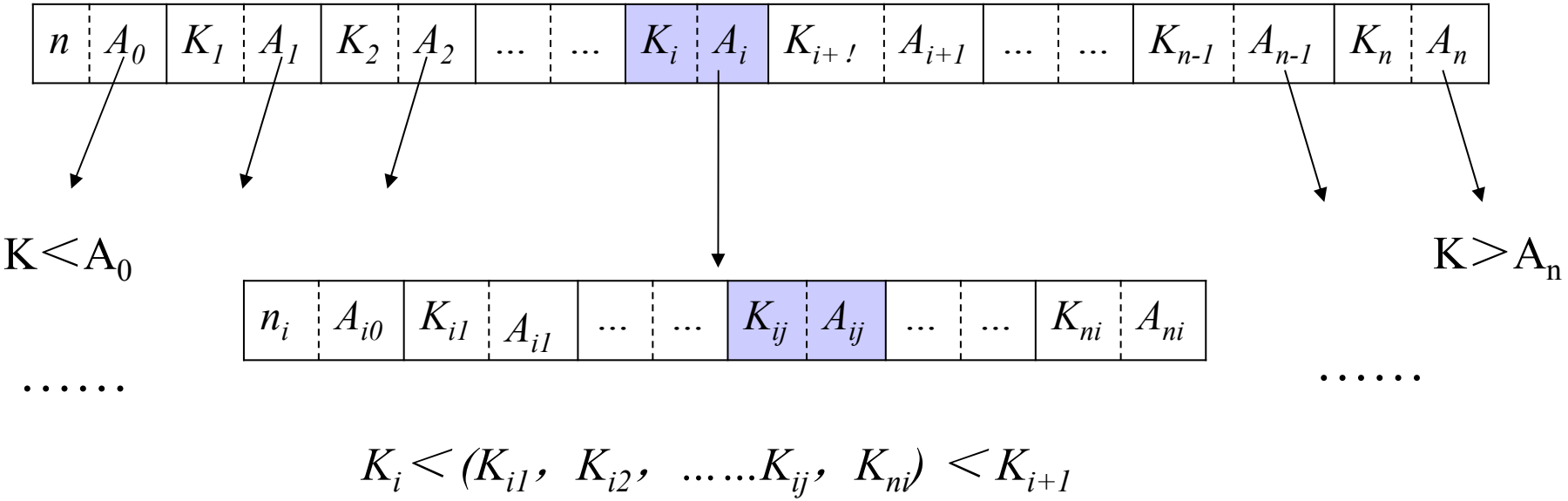
而 $K_i (0 \le i \le n)$ 是元素，并且 $0 \le i \le n < m$；

（2）$K_i < K_{i+1}, (1 \le i < n)$；

（3）子树 A_i 中的所有元素值小于 $K_{i+1}\ (0 \le i < n)$；

（4）子树 A_n 中的所有元素值大于 K_n；

（5）子树 A_i（$0 \le i \le n$）也是 *m* 分支查找树。



$$K_i < (K_{i1}，K_{i2}，\ldots\ldots K_{ij}，K_{ni}) < K_{i+1}$$

B- 树

B-树：B-树是一种非二叉的查找树

除了要满足查找树的特性，还要满足以下结构特性：

一棵 *m* 阶的 B- 树：

（1）树的根或者是一片叶子(一个节点的树),或者其儿子数在 2 和 *m* 之间。

（2）除根外，所有的非叶子结点的孩子数在 *m/2* 和 *m* 之间。

（3）所有的叶子结点都在相同的深度。

以关键字序列

(a，g，f，b，k，d，h，m，i，e，s，i，r，x，c，l，n，t，u，p)

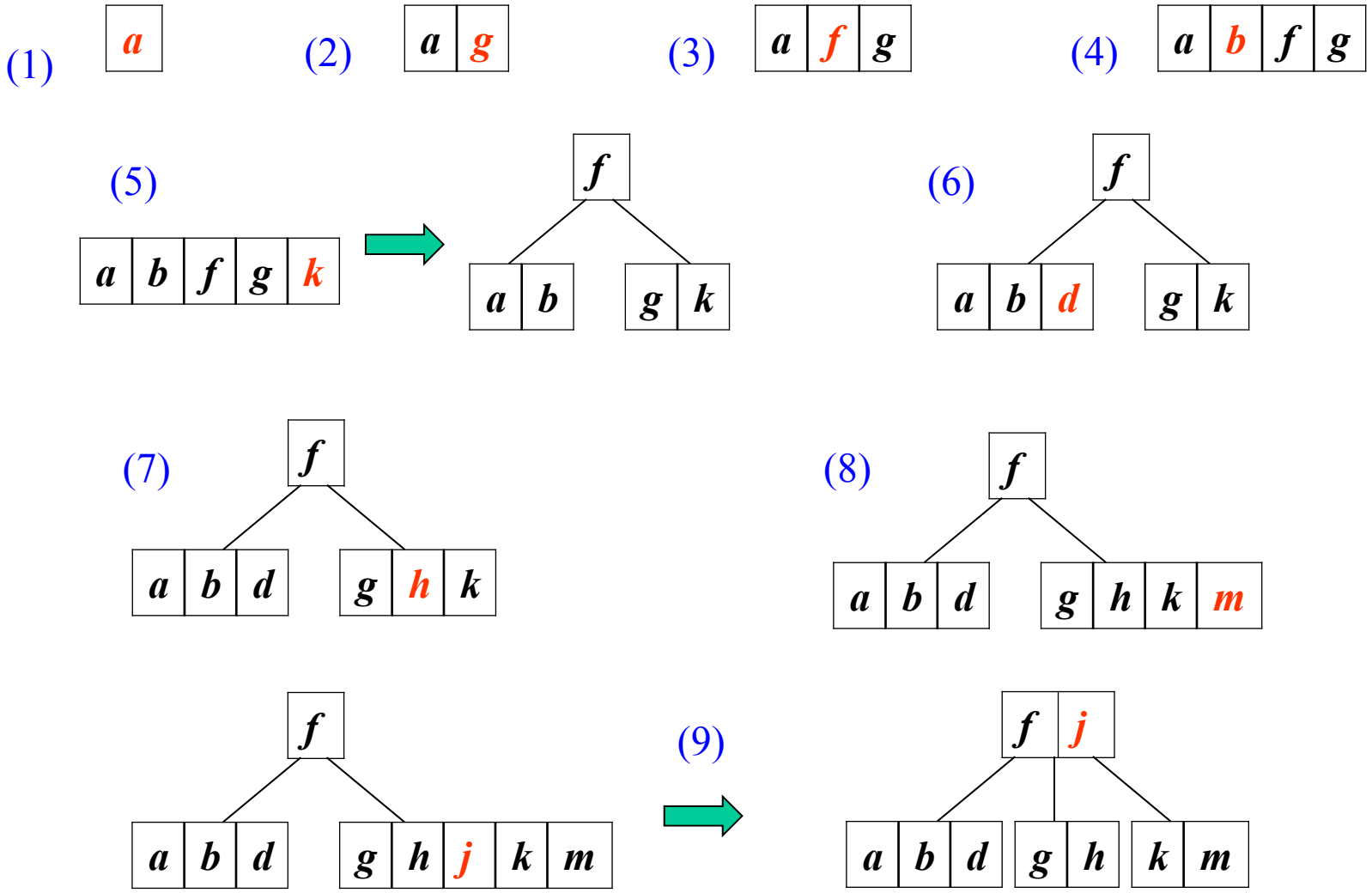
建立一棵5阶B-树的生长过程

注意：

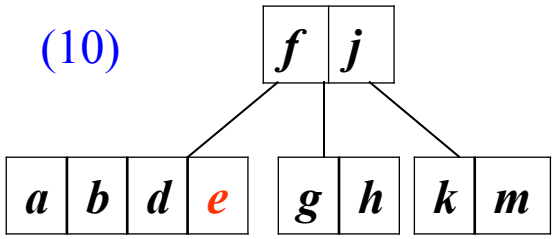
①当一结点分裂时所产生的两个结点大约是半满的，这就为后续的插入腾出了较多的空间,尤其是当 m较大时，往这些半满的空间中插入新的关键字不会很快引起新的分裂。

②向上插人的关键字总是分裂结点的中间位置上的关键字，它未必是正待插入该分裂结点的关键字。因此，无论按何次序插入关键字序列，树都是平衡的。

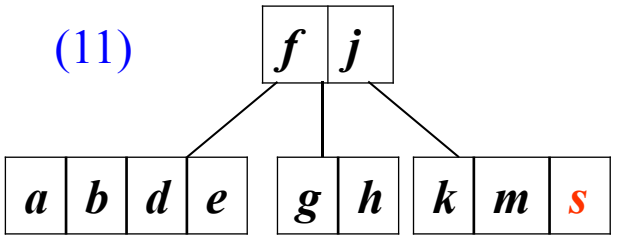
上述关键字序列 (a, g, f, b, k, d, h, m, j, e, s, i, r, x, c, l, n, t, u, p)



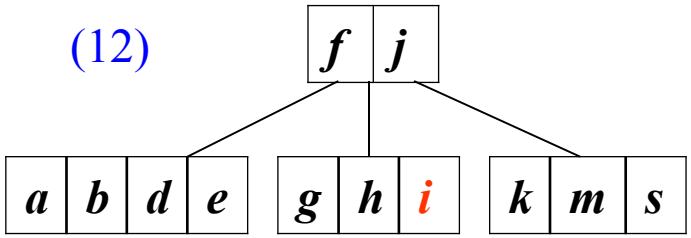
(10)


f j
a b d e
g h
k m

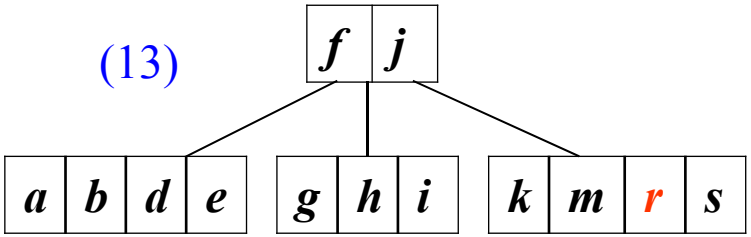
(11)

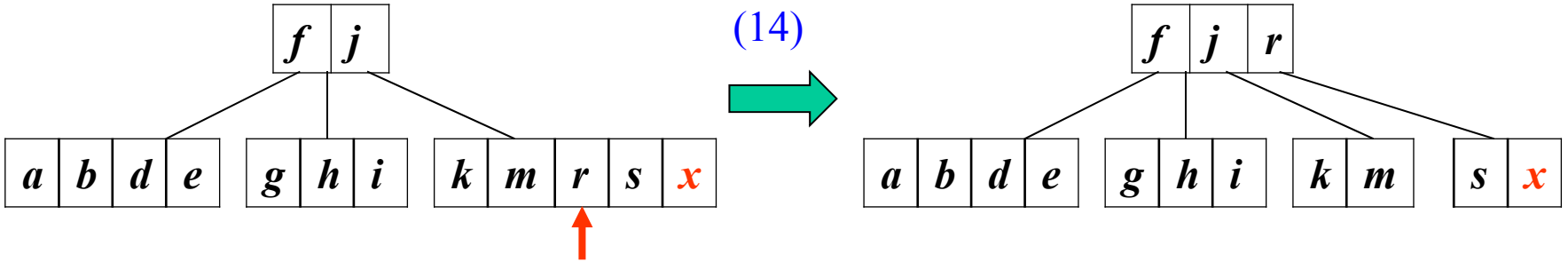

f j
a b d e
g h
k m s

(12)

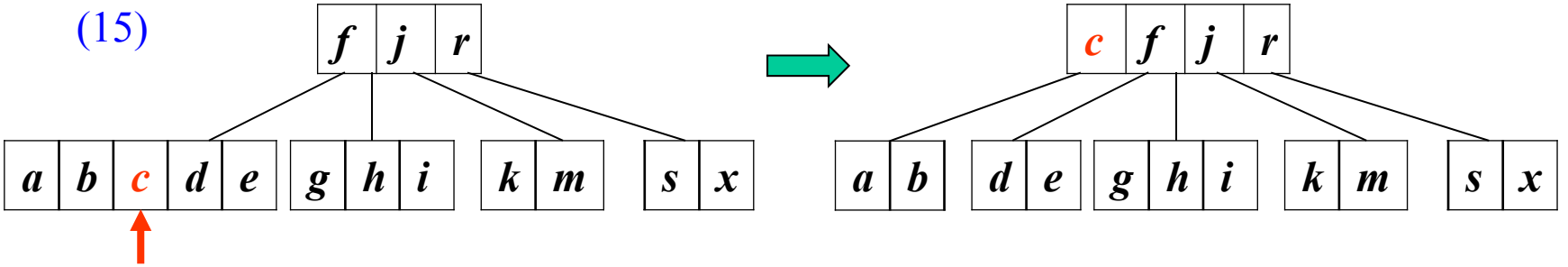

f j
a b d e
g h i
k m s

(13)

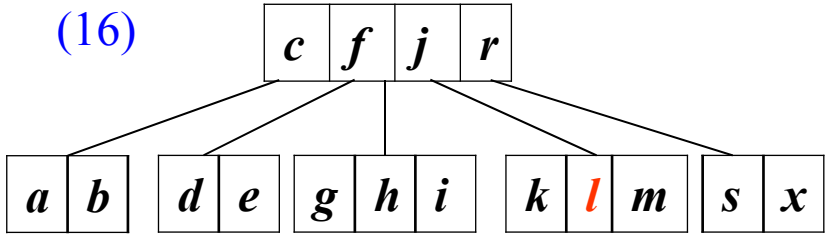

f j
a b d e
g h i
k m r s


f j
a b d e
g h i
k m r s x
(14)
f j r
a b d e
g h i
k m
s x

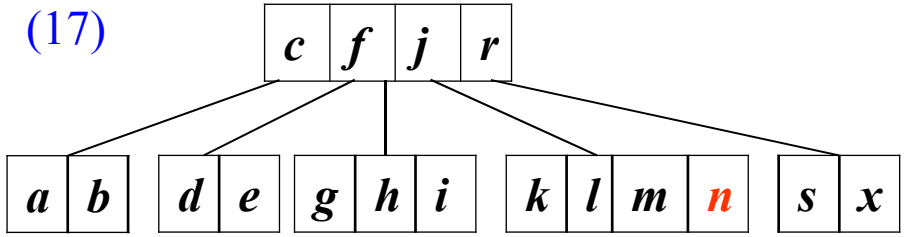
(15)


f j r
a b c d e g h i k m s x
c f j r
a b d e g h i k m s x

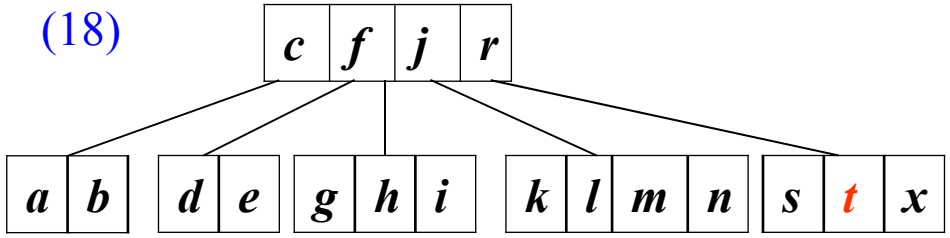
(16)


c f j r
a b d e g h i k l m s x

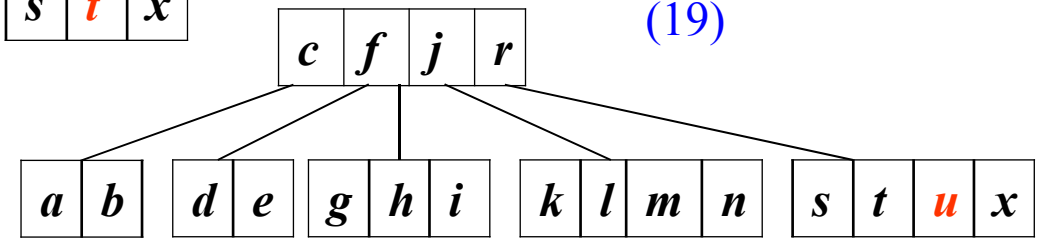
(17)


c f j r
a b d e g h i k l m n s x

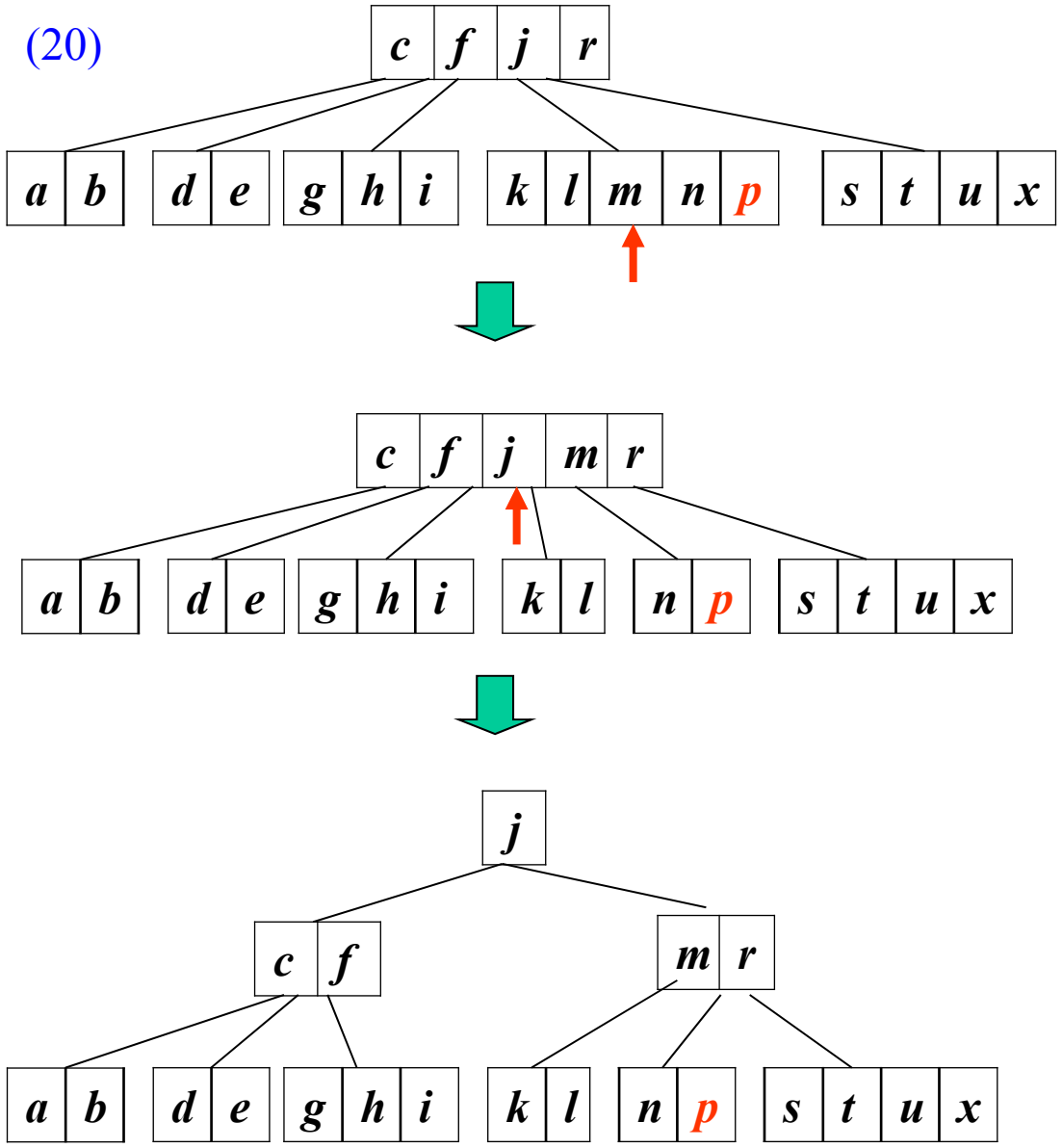
(18)


c f j r
a b d e g h i k l m n s t x

(19)


c f j r
a b d e g h i k l m n s t u x

(20)

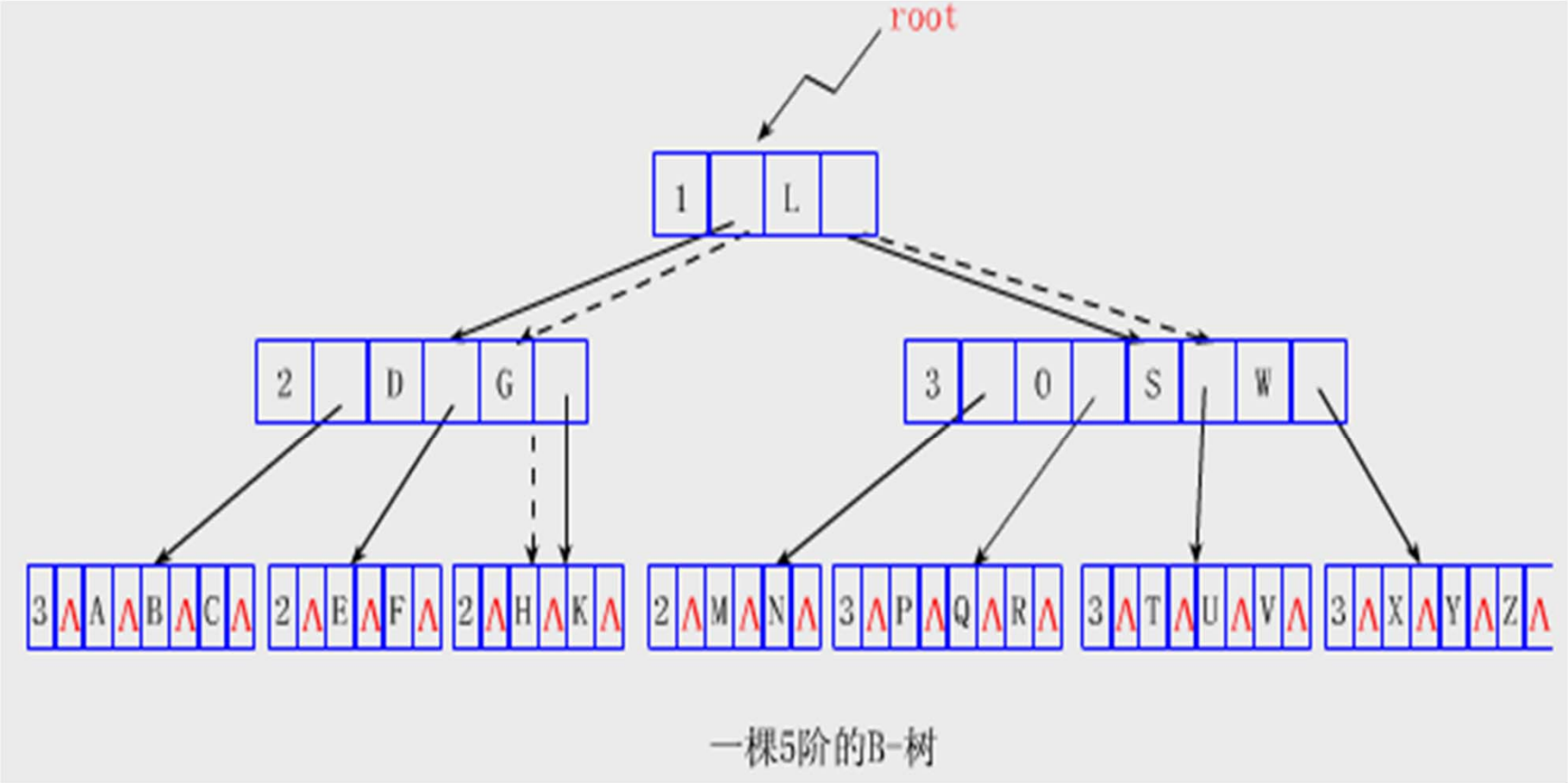


重申：

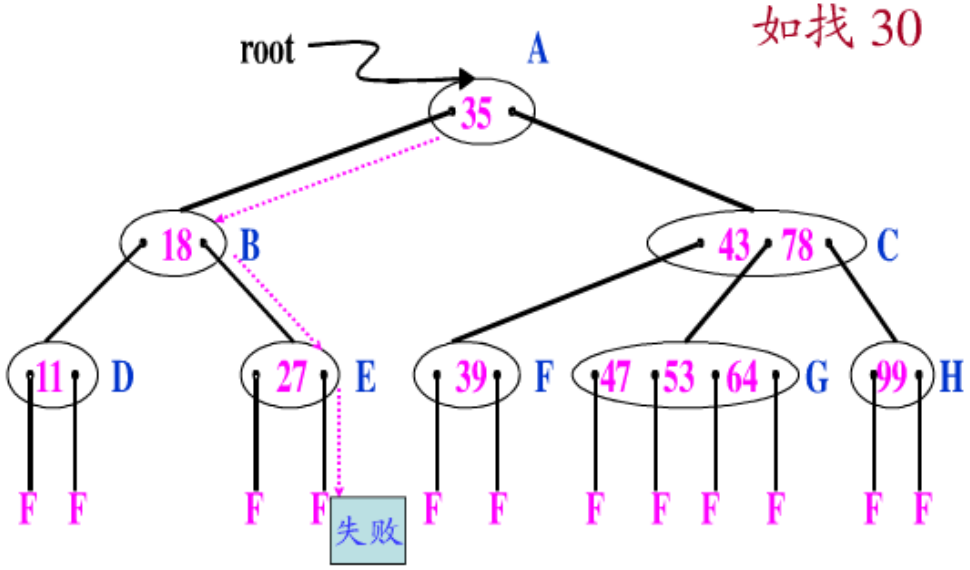
①当一结点分裂时所产生的两个结点大约是半满的，这就为后续的插入腾出了较多的空间，尤其是当 m 较大时，往这些半满的空间中插入新的关键字不会很快引起新的分裂。

②向上插人的关键字总是分裂结点的中间位置上的关键字，它未必是正待插入该分裂结点的关键字。因此，无论按何次序插入关键字序列，树都是平衡的。

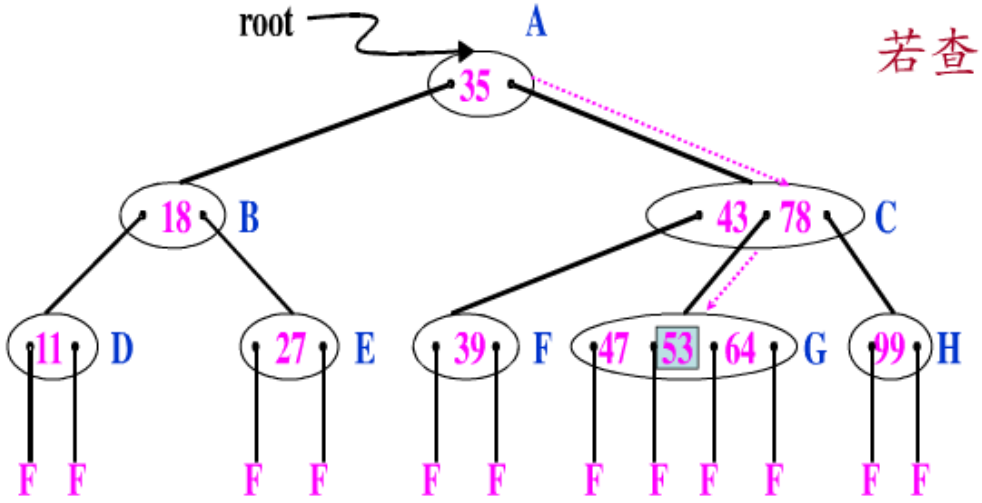
例：下图中左边的虚线表示查找关键字*1*的过程，它失败于叶结点的H和K之间空指针上；右边的虚线表示查找关键字S的过程，并成功地返回S所在结点的地址和S在key[1..keynum]中的位置2。



一棵5阶的B-树



若查找不成功，则返回插入位置。



若查找成功，则返回指向被查关键字所在结点的指针和关键字在结点中的位置

例： 删除5阶B-树的h、r、p、 d等关键字的过程

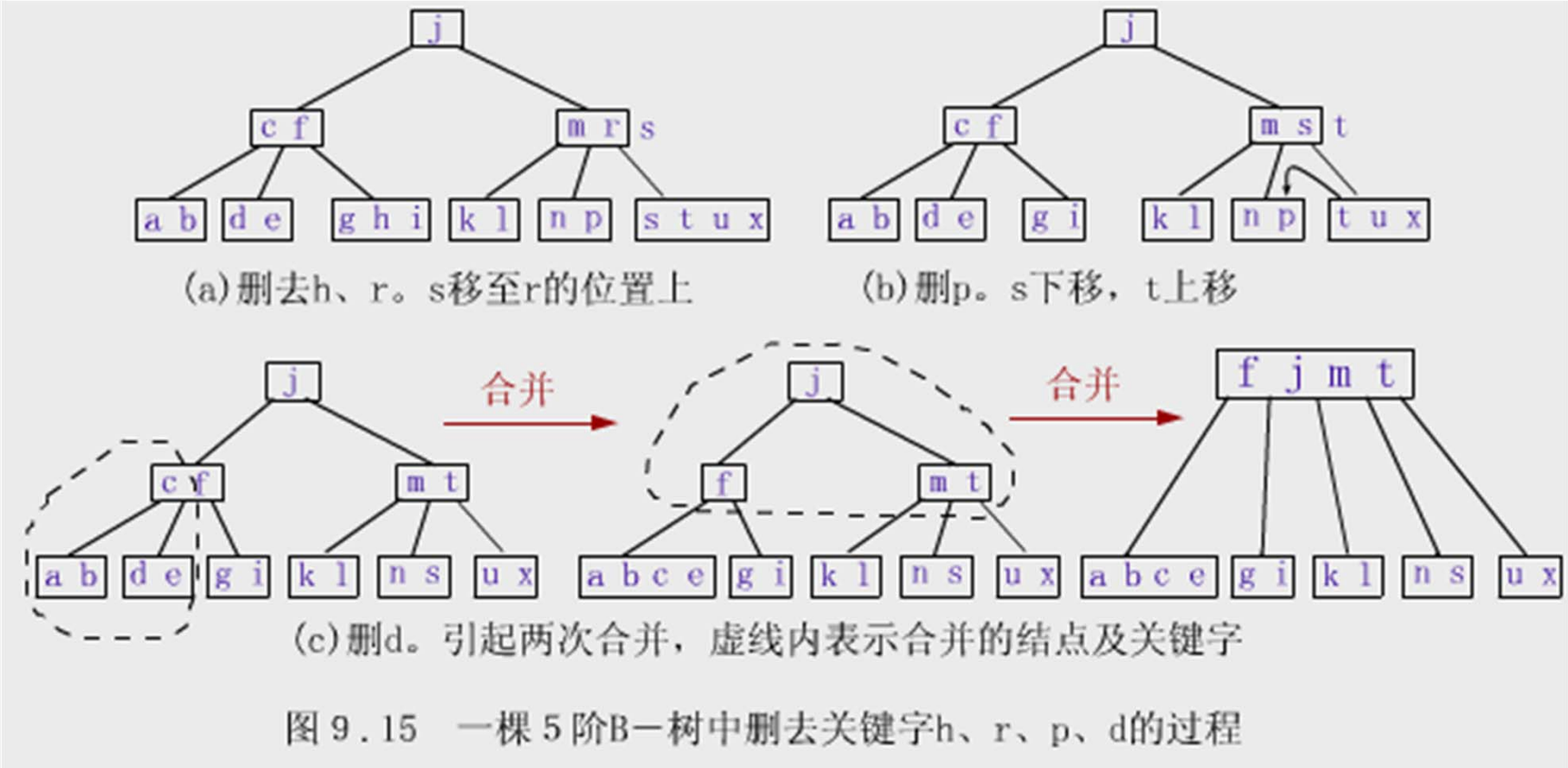


图 9.15 一棵 5 阶B—树中删去关键字h、r、p、d的过程

[B-树定义]

一颗***m* 阶B-树**是一颗m-路查找树，它或为空，或满足以下性质：

1、根结点至少包含*2*个儿子

2、除根结点和失败结点以外，所有结点都至少具有*m/2*个儿子

3、所有失败结点都在同一层

- 所有*2*阶*B*-树都是满二叉树，只有当关键字个数等于2^k-1时，才存在*2*阶B-树
- 对于任意给定的关键字和任意的*m(>2),*一定存在一棵包含上述所有关键字的 *m*阶B-树。

B+树是B-树的一种变形，二者区别在于：

1、有k个子结点的结点必然有k个关键码

2、非叶子结点仅具有索引作用，与记录有关的信息均放在叶结点中

- 查找：即使找到关键码，也必须向下找到叶子结点
- 插入：在叶子上，分裂，改变上层两个最大关键码
- 删除：在叶结点，上层关键马可以保留，删除后若关键字少于m/2，则与B-同样处理

B-树只适合随机检索，但B+树同时支持随机检索和顺序检索

排序：

（一）排序的基本概念
（二）插入排序　1、直接插入排序
　　2、折半插入排序
（三）起泡排序（bubble sort）
（四）简单选择排序
（五）希尔排序（shell sort）
（六）快速排序
（七）堆排序
（八）二路归并排序（merge sort）
（九）基数排序
（十）外部排序
（十一）各种内部排序算法的比较
（十一）排序算法的应用

各种排序的比较

序号	排序方法	平均时间	最好情况	最坏情况	辅助空间	稳定性
1	简单选择排序	$O(n^2)$	O（n）	$O(n^2)$	O(1)	稳定
	直接插入排序	$O(n^2)$	O(n)	$O(n^2)$	O(1)	稳定
	折半插入排序	$O(n^2)$	$O(n\cdot\log_2 n)$	$O(n^2)$	O(1)	不稳定
	冒泡排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	稳定
2	希尔排序	$O(n^{1.3})$	---	---	O(1)	不稳定
3	堆排序	$O(n\cdot\log_2 n)$	$O(n\cdot\log_2 n)$	$O(n\cdot\log_2 n)$	O(1)	不稳定
4	归并排序	$O(n\cdot\log_2 n)$	$O(n\cdot\log_2 n)$	$O(n\cdot\log_2 n)$	O(n)	稳定
5	快速排序	$O(n\cdot\log_2 n)$	$O(n\cdot\log_2 n)$	$O(n^2)$	$O(\log_2 n)$	不稳定
6	基数排序	$O(d\cdot(n+r\cdot d))$	$O(d\cdot(n+r\cdot d))$	$O(d\cdot(n+r\cdot d))$	$O(n+r\cdot d)$	稳定

① 若n较小(如n≤50)，可采用直接插入或直接选择排序。当记录规模较小时，直接插入排序较好；否则因为直接选择移动的记录数少于直接插人，应选直接选择排序为宜。

② 若文件初始状态基本有序(指正序)，则应选用直接插人、冒泡或随机的快速排序为宜；且以直接插入排序最佳。

③ 若n较大，则应采用时间复杂度为O(nlgn)的排序方法：

- 快速排序、堆排序或归并排序。快速排序是目前基于比较的内部排序中被认为是最好的方法，当待排序的关键字是随机分布时，快速排序的平均时间最短；
- 堆排序所需的辅助空间少于快速排序，并且不会出现快速排序可能出现的最坏况。 这两种排序都是不稳定的。
- 基数排序适用于 n 值很大而关键字较小的序列。
- 若要求排序稳定，则可选用归并排序，基数排序稳定性最佳。

重点

[定义]把具有如下性质的数组A表示的二元树称为堆（Heap）：

(1) 若2*i≤n，则A[i].key ≤ A[2*i].key ;

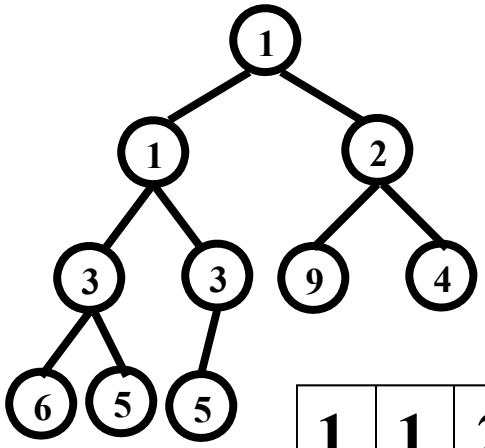
(2) 若2*i+1≤n，则A[i].key≤A[2*i+1].key ;　　小顶堆

或：　把具有如下性质的数组A表示的二元树称为堆（Heap）：

(1) 若2*i≤n，则A[i].key ≥ A[2*i].key ;

(2) 若2*i+1≤n，则A[i].key≥ A[2*i+1].key ;　　大顶堆

例：



1	1	2	3	3	9	4	6	5	5

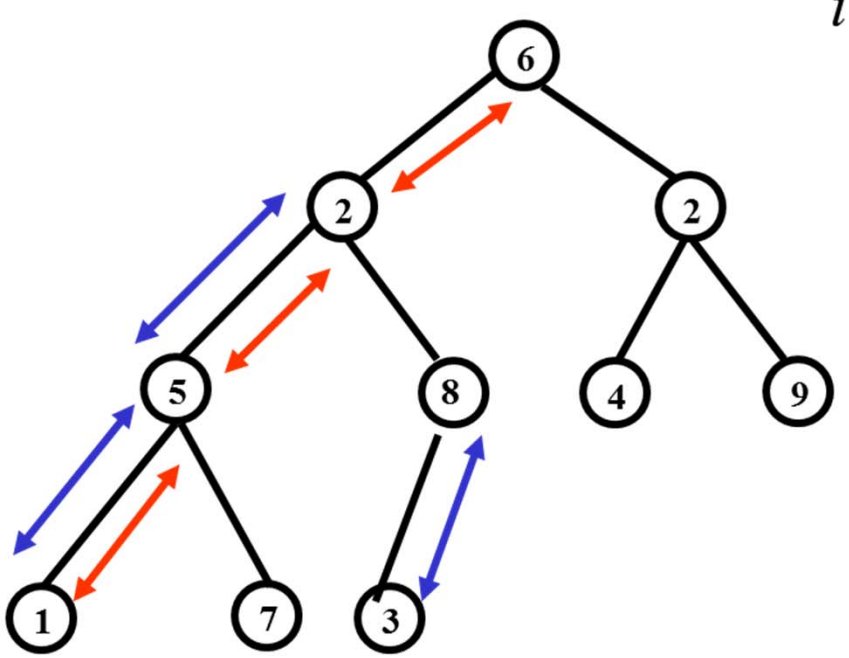
整理堆：

```
Void  pushdown( first, last )
Int  first , last ;
{   int r ;
    r = first ;
    while ( r <= last/2 )
      if ( ( r == last/2 ) && ( last%2 == 0 ) )
        {  if ( A[r].key > A[2*r].key )
              swap ( A[r], A[2*r] ) ;
           r = last ;   /*结束循环*/       }
      else  if ( ( A[r].key > A[2*r].key ) && ( A[2*r].key <= A[2*r+1].key ) )
        {  swap ( A[r], A[2*r] ) ;          /*左孩子比右孩子小，与左孩子交换*/
           r = 2*r ;                        }
      else  if ( ( A[r].key > A[2*r+1].key ) && ( A[2*r+1].key < A[2*r].key ) )
        {  swap ( A[r], A[2*r+1] ) ;     /*右孩子比左孩子小，与右孩子交换*/
           r = 2*r+1 ;                      }
      else
          r = last ;
}
```

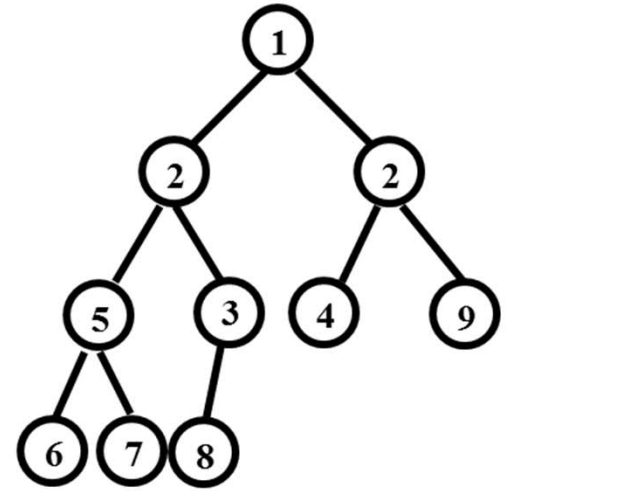
last/2?

O(last/first)

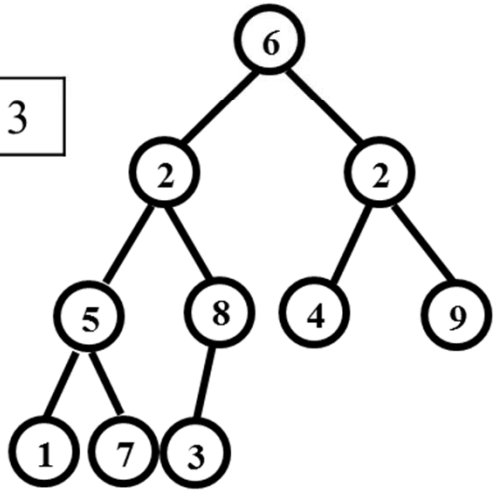
$i = n/2 \ldots\ldots 1$



6	2	2	5	8	4	9	1	7	3



6	2	2	5	3	4	9	6	7	8



堆分类

```
Void   sort ( n,  A )
 int   n ;  LIST   A ;
{   int  i ;
    for ( i = n/2 ; i >= 1 ; i-- )
        pushdown( i , n ) ;
    for ( i = n ; i >= 2 ; i-- )
        {   swap ( A[1], A[i] ) ;
            pushdown ( 1,  i -1 ) ;  }
}
```

整理堆➔建堆

?

堆排序

first ?

last ?

$T(n) = O(n \cdot \log_2 n)$

6-1

以关键字序列 (265, 301, 751, 129, 937, 863, 742, 694, 076, 438)为例，分别写出执行以下排序算法的各趟排序结束时，关键字序列的状态。

(1) 直接插入排序 (2)希尔排序 (3)冒泡排序 (4)快速排序
(5) 直接选择排序 (6) 归并排序 (7) 堆排序 (8)基数排序

上述方法中：(1)哪些是稳定的排序?
(2)哪些是非稳定的排序?
(3)对不稳定的排序试举出一个不稳定的实例。

在上面的排序方法中:
(1)直接插入排序、冒泡排序、归并排序和基数排序是稳定的。
(2)其他排序算法均是不稳定的。
(3)以带 * 号的表示区别：

希尔排序：	[8, 1, 10, 5, 6, 8*]
快速排序：	[2, 2*, 1]
直接选择排序：	[2, 2*, 1]
堆排序：	[2, 2*, 1]

(1)直接插入排序(方括号表示无序区)

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

(2)希尔排序(增量为5，3，1)

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

(3)冒泡排序(方括号为无序区)

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

(4)快速排序(方括号表示无序区，层表示对应的递归树的层数)

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

为什么有序的单链表不能进行折半查找 ？

(5)直接选择排序(方括号为无序区)

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

(6)归并排序(为了表示方便，采用自底向上的归并，方括号为有序区)

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

(6)堆排序（通过画二叉树可以一步步得出排序结果）

初始态：[265 301 751 129 937 863 742 694 076 438]

建立初始堆：[937 694 863 265 438 751 742 129 075 301]

第一次排序重建堆：[863 694 751 765 438 301 742 129 075] 937

第二次排序重建堆：[751 694 742 265 438 301 075 129] 863 937

第三次排序重建堆：[742 694 301 265 438 129 075] 751 863 937

第四次排序重建堆：[694 438 301 265 075 129] 742 751 863 937

第五次排序重建堆：[438 265 301 129 075] 694 742 751 863 937

第六次排序重建堆：[301 265 075 129] 438 694 742 751 863 937

第七次排序重建堆：[265 129 075] 301 438 694 742 751 863 937

第八次排序重建堆：[129 075]265 301 438 694 742 751 863 937

第九次排序重建堆： 075 129 265 301 438 694 742 751 863 937

(8)基数排序(方括号内表示一个箱子共有10个箱子，箱号从0到9)

初始态：265 301 751 129 937 863 742 694 076 438

第一趟：[] [301 751] [742] [863] [694] [265] [076] [937] [438] [129]

第二趟：[301] [] [129] [937 438] [742] [751] [863 265] [076] [] [694]

第三趟：[075] [129] [265] [301] [438] [] [694] [742 751] [863] [937]

6-2

当R[low..high]中的关键字均相同时，Partion返回值是什么？此时快速排序的的运行时间是多少？能否修改Partion,使得划分结果是 平衡的(即划分后左右区间的长度大致相等)？

此时Partion 返回值是low.此时快速排序的运行时间是：

(high-low)(high-low-1)/2=O((high-low)^2)，

可以修改Partion，将其中RecType pivot=R[i];句改为：

RecType pivot=R[(j+i)/2];

也就是取中间的关键字为基准，这样就能使划分的结果是平衡的。

6-3

若关键字是非负整数，快速排序、归并、堆和基数排序哪一个最快?若要求辅助空间为O(1)，则应选择谁？若要求排序是稳定的，且关键字是实数，则应选择谁？

若关键字是非负整数，则基数排序最快；
若要求辅助空间为O(1)，则应选堆排序；
若要求排序是稳定的，且关键字是实数，则应选归并排序，
因为基数排序不适用于实数，虽然它也是稳定的。

6-4

判别下列序列是否为堆(小顶堆或大顶堆)，若不是，则将其调整为堆：

(1) (100,86,73,35,39,42,57,66,21)
(2) (12,70,33,65,24,56,48,92,86,33)
(3) (103,97,56,38,66,23,42,12,30,52,06,20)
(4) (05,56,20,23,40,38,29,61,35,76,28,100)

堆的性质是：

任一非叶结点上的关键字均不大于(或不小于)其孩子结点上的关键字。

据此我们可以通过画二叉树来进行判断和调整：

(1) 此序列不是堆，经调整后成为大顶堆：

(100, 80, 73, 66, 39, 42, 57, 35, 21)

(2) 此序列不是堆，经调整后成为小顶堆：

(12, 24, 33, 65, 33, 56, 48, 92, 86, 70)

(3) 此序列是一大顶堆。

(4) 此序列不是堆，经调整后成为小顶堆：

(05, 23, 20, 56, 28, 38, 29, 61, 35, 76, 100)

6-5 有序数组是堆吗?

有序数组是堆。因为有序数组中的关键字序列满足堆的性质。若数组为降序，则此堆为大顶堆，反之为小顶堆。

6-6 若文件初态是反序的，且要求稳定排序，则在直接插入、直接选择、冒泡排序和快速排序中选哪则种方法为宜?

这四种排序算法中，直接选择、快速排序均是不稳定的，因此先予以排除，剩下两种算法中，由于直接插入算法所费时间比冒泡法更少，因此选择直接插入排序算法为宜。

6-7 若文件初态是反序的，则直接插入，直接选择和冒泡排序哪一个更好？

应选直接选择排序为更好。分析如下：

(1)在直接插入排序算法中，反序输入时是最坏情况，此时：

关键字的比较次数：$C_{max}=(n+2)(n-2)/2$

记录移动次数为：$M_{max}=(n-1)(n+4)/2$

$T_{max}=n^2-4n-3$ (以上二者相加)

(2)在冒泡排序算法中，反序也是最坏情况，此时：

$C_{max}=n(n-1)/2$　　$M_{max}=3n(n-1)/2$

$T_{max}=2n^2-2n$

(3)在选择排序算法中，

$C_{max}=n(n-1)/2$　　$M_{max}=3(n-1)$

$T_{max}=n^2/2-5n/2-3$

虽然它们的时间复杂度都是$O(n^2)$,但是选择排序的常数因子为1/2，因此选择排序最省时间。

6-8 **将哨兵放在R[n]中，被排序的记录放在R[0..n-1]中，重写直接插入排序算法。**

重写的算法如下：

```
 void InsertSort(SeqList R)
{  //对顺序表中记录R[0..n-1]按递增序进行插入排序
   int i,j;
   for(i=n-2;i>=0;i--)                //在有序区中依次插入R[n-2]..R[0]
     if(R[i].key>R[i+1].key)          //若不是这样则R[i]原位不动
     {
        R[n]=R[i];j=i+1;              //R[n]是哨兵
        do{                           //从左向右在有序区中查找插入位置
              R[j-1]=R[j];            //将关键字小于R[i].key的记录向右移
              j++;
        }while(R[j].key<R[n].key]);
       R[j-1]=R[n];                   //将R[i]插入到正确位置上
     }                                //endif
}
```

6-9

对给定的j(*1≤j≤n*),要求在无序的记录区*R*[*1..n*]中找到按关键字自小到大排在第 *j* 个位置上的记录(即在无序集合中找到第*j*个最小元)，试利用快速排序的划分思想编写算法实现上述的查找操作。

```
int QuickSort ( SeqList R, int j, int low, int high)
{                                  //对R[low..high]快速排序
   int pivotpos;                   //划分后的基准记录的位置
   if (low<high){                  //仅当区间长度大于1时才须排序
      pivotpos = Partition(R，low，high)；  //对R[low..high]做划分
      if (pivotpos == j) return r[j];
      else if ( pivotpos > j) return (R, j, low, pivotpos-1);
      else return quicksort (R, j, pivotpos+1, high);
   } //Endif
} //QuickSort
```

6-10

设向量 *A*[*0..n-1*]中存有 *n* 个互不相同的整数，且每个元素的值均在 *0* 到 *n-1*之间。试写一时间为*O*(*n*)的算法将向量*A*排序，结果可输出 到另一个向量 **B**[*0..n-1*] 中。

```
Sort ( int *A , int *B )
{   //将向量A排序后送入B向量中
    int i;
    for(i=0;i<=n-1;i++)
            B[A[i]]=A[i];
}
```

```
Sort ( int *A )
{   int i;
    for(i=0;i<=n-1;i++)
            A[i] = i ;
}
```

6-11

写一个**heapInsert(*R*,*key*)**算法，将关键字插入到堆***R***中去，并保证插入***R***后仍是堆。提示：应为堆***R***增加一个长度属性描述(即改写本章定义的**SeqList**类型描述，使其含有长度域)；将***key***先插入***R***中已有元素的尾部(即原堆的长度加***1***的位置，插入后堆的长度加**1**)，然后从下往上调整，使插入的关键字满足性质。请分析算法的时间。

```
类型定义：
  #define n 100//假设文件的最长可能长度
   typedef int KeyType;          //定义KeyType 为int型
   typedef struct node{
         KeyType key;            //关键字域
         OtherInfoType info;     //其它信息域，
      }Rectype;                  //记录结点类型
   typedef struct{
         Rectype data[n] ;       //存放记录的空间
         int length;             //文件长度
      }seqlist;
```

```
void heapInsert(seqlist *R,KeyType key)
{ int large;
  int low,high;
  int i;
   R->length++;  R->data[R->length].key=key;    //插入新的记录
   for(i=R->length/2;i>0;i--)                   //建堆
   {  low=i;high=R->length;
      R->data[0].key=R->data[low].key;          //R->data[low]是当前调整的结点
      for(large=2*low;large<=high;large*=2)
      {  //若large>high，则表示R->data[low]是叶子，调整结束；
         否则large指向R->data[low]的左孩子
         if(large<high&&R->data[large].key<R->data[large+1].key)
            large++;      //若R->data[low]的右孩子存在，且关键字大于左兄弟，则令large指向它
         if (R->data[0].key<R->data[large].key)
           {   R->data[low].key= R->data[large].key;
               low=large;                          }      //令low指向新的调整结点
          else break; /当前调整结点不小于其孩子结点的关键字，结束调整
      }//endfor
      R->data[low].key=R->data[0].key;          //将被调整结点放入最终的位置上
   }//end of for
}end of heapinsert
```

原有堆元素在R->data[1]..R->data[R->length]，
将新的关键字key插入到R->data[R->length+1]位置后，
以R->data[0]为辅助空间，调整为堆(此处设为大顶堆)

最坏时间复杂度为O(nlgn)，辅助空间为O(1).

6-12

写一个建堆算法：从空堆开始，依次读入元素调用上题中堆插入算法将其插入堆中。

```
void BuildHeap(seqlist *R)
{
      KeyType key;
      R->length=0;//建一个空堆
      scanf("%d",&key);//设MAXINT为不可能的关键字
      while(key!=MAXINT)
         {
            heapInsert(R,key);
            scanf("%d",&key);
         }
   }
```

6-13

写一个堆删除算法：HeapDelete(R,i), 将 R[i] 从堆中删去，并分析算法时间。

提示：先将R[i]和堆中最后一个元素交换，并将堆长度减1，然后从位置i开始向下调整，使其满足堆性质。

```
//原有堆元素在 R->data[1]~R->data[R->length],
//将R->data[i]删除，即将 R->data[R->length]放入R->data[i] 中后
//将R->length减1，再进行堆的调整，
//以R->data[0]为辅助空间，调整为堆(此处设为大顶堆)
```

```
void HeapDelete(seqlist *R,int i)
{    int large;        //large指向调整结点的左右孩子中关键字较大者
     int low,high;   //low和high分别指向待调整堆的第一个和最后一个记录
     int j;
     if (i>R->length)   Error("have no such node");
     R->data[i].key=R->data[R->length].key;
     R->length--;  R->data[R->length].key=key;//插入新的记录
     for(j=i/2;j>0;j--)//建堆
      {  low=j;high=R->length;
         R->data[0].key=R->data[low].key;   //R->data[low]是当前调整的结点
         for(large=2*low;large<=high;large*=2)
         {   //若large>high，则表示R->data[low]是叶子，调整结束；
             //否则令large指向R->data[low]的左孩子
             if(large<high&&R->data[large].key<R->data[large+1].key)
                large++;//若R->data[low]的右孩子存在且关键字大于左兄弟，则令large指向它
             if (R->data[0].key<R->data[large].key)
                { R->data[low].key= R->data[large].key;
                  low=large;//令low指向新的调整结点   }
             else break;      //当前调整结点不小于其孩子结点的关键字，结束调整
          }//endfor
          R->data[low].key=R->data[0].key;    //将被调整结点放入最终的位置上
       }//end of for
}end of HeapDelete
```

6-14

在一棵m阶的B-树中，当将一关键字插入某结点而引起该结点的分裂时，此结点原有多少个关键字?若删去某结点中的一个关键字，而导致结点合并时，该结点中原有几个关键字?

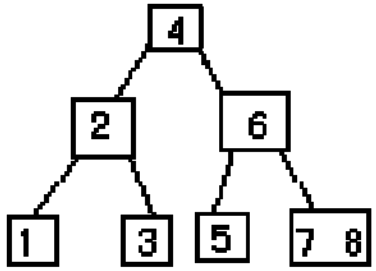
答：在一棵 m 阶的B-树中，若由于一关键字的插入某结点而引起该结点的分裂,则该结点原有m-1个关键字。

若删去某结点中一个关键字而导致结点合并，该结点中原有m/2-1个关键字。

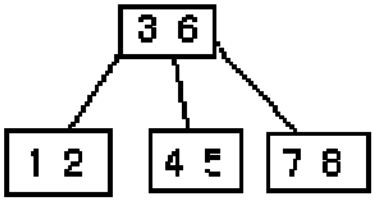
6-15

一棵B-树中，空指针数总是比关键字数多一个，此说法是否正确?请问包含8个关键字的3阶B-树(即2-3树)最多有几个结点?最少有几个结点?画出这两种情况的 B- 树。

这个说法是正确的。包含8个关键字的3阶B-树最多有7个结点，最少有4个结点。



(1)



(2)

例：下述二叉树中,哪一种满足性质:从任一结点出发到根的路径上所经过的结点序列按其关键字有序

(A) 二叉排序树　(B) 赫夫曼树　(C) AVL树　(D) 堆

【分析】

- 对于选项A，根据二叉排序树的结构特点我们可以知道，二叉排序树的中序遍历结果是一个有序序列，而在中序遍历中，父结点并不总是出现在孩子结点的前面（或后面），故该选项不正确。
- 对于选项B，根据赫夫曼树的结构特点我们可以知道，在赫夫曼树中所有的关键字只出现在叶结点上，其非叶结点上并没有关键字值，显然不正确。
- 对于选项C，AVL树其本质上也是一种二叉排序树，只不过是平衡化之后的二叉排序树，故该选项也是不正确的。
- 对于选项D，堆的概念我们会在堆排序中给大家介绍，根据建堆的过程，不断地把大者“上浮”，将小者“筛选”下去，最终得到的正是一个从任一结点出发到根的路径上所经过的结点序列按其关键字有序的树状结构。

D是正确答案，你选对了吗？为什么错？

6-16

设计双向气泡排序算法，在排序过程中交替改变扫描方向。

```
Void DoubleBubbleSort( LIST &A, int  n)
{  int i=1,flag=1;
   while( flag )
    {   flag = 0;
        for( j=n-i+1; j>=i+1; j--)     //较小元素A[j]
           if( A[j].key < A[j-1].key ) {   flag=1;
                                           swap(A[j],A[j-1]);   }
        for( j=i+1; j<=n-i+1; j++)   //较大元素A[n-i+1]
           if( A[j].key > A[j+1].key ) {   flag=1;
                                           swap(A[j],A[j+1]);  }
        i++;
    }
}
```

T(*n*)=O(n^2)　**flag ?**

6-17

设计算法，实现奇偶转换排序。

基本思想：

第一趟对所有奇数的i，将A[i]和A[i+1]进行比较；

第二趟对所有偶数的i，将A[i]和A[i+1]进行比较；

if(A[i]>A[i+1]) swap(A[i],A[i+1]);

重复上述二趟交换过程交换进行，直至整个数组有序。

问：(1)结束条件是什么？ (2)实现算法。

```
Void  OESort(LIST &A , int n)
{   int  i , flag ;
    do{  flag = 0 ;
         for( i=0; i<n ; i+=2)
              if( A[i] > A[i+1] )  {  flag = 1;
                                      swap(A[i],A[i+1]); }
         for( i=1; i<n ; i+=2)
              if( A[i] > A[i+1] )  {  flag = 1;
                                      swap(A[i],A[i+1]); }
     }while(flag);
}
```