# 习题一

## 一、选择题

1．B　2．C　3．B　4．D　5．C　6．D　7．A　8．C

## 二、填空题

1．数据元素　数据元素间关系

2．数据的组织形式，即数据元素之间逻辑关系的总体

3．有穷性　确定性　可行性

4．算法的时间复杂度和空间复杂度

5．集合　线性结构　树形结构　图状结构或网状结构

## 三、简述题

1．解答：

四种表示方法。

⑴顺序存储方式。数据元素顺序存放，每个存储结点只含一个元素。存储位置反映数据元素间的逻辑关系。存储密度大，但有些操作（如插入、删除）效率较差。

⑵链式存储方式。每个存储结点除包含数据元素信息外还包含一组（至少一个）指针。指针反映数据元素间的逻辑关系。这种方式不要求存储空间连续，便于动态操作（如插入、删除等），但存储空间开销大（用于指针），另外不能折半查找等。

⑶索引存储方式。除数据元素存储在一地址连续的内存空间外，尚需建立一个索引表，索引表中索引指示存储结点的存储位置（下标）或存储区间端点（下标），兼有静态和动态特性。

⑷散列存储方式。通过散列函数和解决冲突的方法，将关键字散列在连续的有限的地址空间内，并将散列函数的值解释成关键字所在元素的存储地址，这种存储方式称为散列存储。其特点是存取速度快，只能按关键字随机存取，不能顺序存取，也不能折半存取。

2．解答：

数据类型是程序设计语言中的一个概念，它是一个值的集合和操作的集合。如 C 语言中的整型、实型、字符型等，如整数的取值范围与具体机器和编译系统有关，其操作有加、减、乘、除、求余等。实际上数据类型是厂家提供给用户的已实现了的数据结构。“抽象数据类型（ADT）”指一个数学模型及定义在该模型上的一组操作。“抽象”的意义在于数据类型的数学抽象特性。抽象数据类型的定义仅取决于它的逻辑特性，而与其在计算机内部如何表示和实现无关。无论其内部结构如何变化，只要它的数学特性不变就不影响它的外部使用。抽象数据类型和数据类型实质上是一个概念。此外，抽象数据类型的范围更广，它已不再局限于机器已定义和实现的数据类型，还包括用户在设计软件系统时自行定义的数据类型。使用抽象数据类型定义的软件模块含定义、表示和实现三部分，封装在一起，对用户透明（提供接口），而不必了解实现细节。抽象数据类型的出现使程序设计不再是“艺术”，而是向“科学”迈进了一步。

3．解答：

评价好的算法有四个方面。一是算法的正确性；二是算法的易读性；三是算法的健壮性；四是算法的时空效率（运行）。

4．解答：

逻辑结构、存储结构、操作（运算）。

5．解答：

通常考虑算法所需要的存储空间量和算法所需要的时间量。后者又涉及到四方面：程序运行时所需输入的数据总量，对源程序进行编译所需时间，计算机执行每条指令所需时间和程序中指令重复执行的次数。

6．解答：

栈和队列的逻辑结构相同，其存储表示也可相同（顺序存储和链式存储），但由于其运算集合不同而成为不同的数据结构。

7．解答：

线性表中的插入、删除操作，在顺序存储方式下平均移动近一半的元素，时间复杂度为O(n)；而在链式存储方式下，插入和删除时间复杂度都是O(1)。

8．解答：

对算法A1和A2的时间复杂度T1和T2取对数，得$n\log_2 2$和$2\log_2 n$。显然，算法A2好于A1。

9．解答：

⑴语句执行的次数为n-2次，T(n)=O(n)。

⑵语句执行的次数为n-1次，T(n)=O(n)。

⑶语句执行的次数为n次，T(n)=O(n)。

⑷语句执行的次数为$n^{1/2}$次，T(n)=$O(n^{1/2})$。

⑸语句执行的次数为(n(n-1)(n-2))/6次，T(n)=$O(n^3)$。

# 习题二

## 一、选择题

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

## 二、填空题

1. 不管单链表是否为空表，头指针均不空，并使得对单链表的操作在各种情况下统一
2. 4　2
3. 随机　随机存取
4. O(n)　O(n)
5. 相同
6. O(1)　O(n)
7. 不同
8. 物理存储位置　链指针
9. O(1)　O(n)
10. L->next==L && L->prior==L

## 三、算法设计题

**说明：**以下解答中有关顺序表的习题要包含头文件：SqList.h，单链表的习题要包含头文件：LinkList.h。

1. 算法描述如下：

```
bool SLOrderInsert(SqList &L, ElemType x)
{
  int i;
  if(L.length>=L.listsize)  {              // 当前存储空间已满，增补空间
L.elem=(ElemType)realloc(L.elem,(L.listsize+L.incrementsize)*sizeof(ElemType));
    if(!L.elem) return false;              // 存储分配失败
    L.listsize+=L.incrementsize;           // 当前存储容量增加
    }
    for(i=L.length-1;i>=0&&x<L.elem[i];i--)
      L.elem[i+1]=L.elem[i];
    L.elem[i+1]=x;
    L.length++;
    return  true;
}
```

2. 算法描述如下：

```
void Converse_Sq(SqList &L)
{
   int mid, i;
   ElemType x;
   mid=L.length/2;
   for(i=0;i<mid;i++)
   {
      x=L.elem[i];
      L.elem[i]=L.elem[L.length-1-i];
      L.elem[L.length-1-i]=x;
   }
}
```

3. 算法描述如下：

```
void Converse2_Sq(ElemType a[],int low, int high)
 {
   int n,i;
   ElemType x;
   n=(high-low+1)/2;
   for(i=0;i<n;i++)
   {
     x=a[low+i];
     a[low+i]=a[high-i];
```

```
     a[high-i]=x;
    }
  }

void Exchange_Sq(ElemType a[],int m,int n)
 {
    Converse2_Sq(a,0,m+n-1);
    Converse2_Sq(a,0,n-1);
    Converse2_Sq(a,n,n+m-1);
 }
```

4. 算法描述如下：

```
void Delete_Sq(SqList &L,ElemType s,ElemType t)
{
   int i,k;
   for(i=0;i<L.length;i++)
    if(L.elem[i]>=s&&L.elem[i]<=t)
      L.elem[i]=0;
   for(i=L.length-1;i>=0;i--)
    if(L.elem[i]==0)
    {
      for(k=i;k<=(L.length-2);k++)
         L.elem[k]= L.elem[k+1];
      L.length--;
    }
 }
```

5. 算法描述如下：

```
void Converse_L(LinkList &head)
 {
   LinkList p, q;
   p=head->next;
   head->next=NULL;

   while(p)
   {
     q=p;
     p=p->next;
     q->next=head->next;
     head->next=q;
   }
 }
```

6. 算法描述如下：

```
void LinListSort(LinkList &head)
{
  LinkList curr,pre,p,q;
  p=head->next;
  head->next=NULL;

  while(p)
  {
    curr=head->next;
    pre=head;
    while(curr&& curr->data<=p->data)
    {
      pre=curr;
      curr=curr->next;
    }
    q=p;
    p=p->next;
    q->next=pre->next;
    pre->next=q;
    }
}
```

7. 算法描述如下：

```
bool NListInsert_L( LinkList &head, int i, ElemType e)
{    //在不带有头结点的单链表 L 中的第 i 个结点之前插入元素 e
  LinkList p,s;
  int j;
  p=head;  j=1;
  while(p->next&&j<i-1)  { p=p->next; j++; }
                                    // 寻找第 i-1 个结点,并让 p 指向此结点
  if(j!=i-1&&i!=1)   return false;          // i 的位置不合理
  if((s=(LNode *)malloc(sizeof(LNode)))==NULL) exit(1); // 存储分配失败
  s->data=e;
  if(i==1)                                  // 插入在表头
  {
    s->next=head;
    head=s;
  }
  else
  {  s->next=p->next;  p->next=s;   }      // 插入新结点
    return true;
 }

bool NListDelete_L( LinkList &head, int i, ElemType &e)
```

```
{    // 删除不带有头结点的单链表 L 中的第 i 个结点，并让 e 返回其值
  LinkList p,q;
  int j;
  p=head;  j=1;
  while(p->next&&p->next->next&&j<i-1){ p=p->next; j++; }
                              // 寻找第 i-1 个结点,并让 p 指向此结点
  if(j!=i-1&&i!=1)   return false;          // i 的位置不合理
  if(i==1)                                  // 删除表头结点
   {
     q=p;
     head=head->next;
   }
   else
   {
    q=p->next;                              // q 指向其后继
    p->next=q->next;                        // 删除 q 所指结点
    e=q->data;    free(q);
   }
   return true;
 }
```

8. 算法描述如下：

```
typedef struct DuNode  {
 ElemType data;
 struct  DuNode *prior;
 struct  DuNode *next;
}DuLNode,*DuLinkList;
int ListLength_DuL(DuLinkList &L)
{   // 统计带头结点的双向循环链表 L
  DuLinkList p=L->next;
  int num=0;
  while(p!=L)
  {
   num++;
   p=p->next;
  }
  return num;
}

bool ListGet_DuL(DuLinkList &L, int i, ElemType &e)
{   // 在带有头结点的双向循环链表 L 中的取第 i 个结点，并让 e 返回其值
  DuLinkList p;
  int j;
  p=L->next;  j=1;
```

```
  while(p->next!=L&&j<i){ p=p->next; j++; }
                                        // 寻找第 i 个结点,并让 p 指向此结点
   if(j!=i)   return false;                    // i 的位置不合理
   e=p->data;                                  // 被取元素的值赋给 e
  return true;
 }
```

9. 算法描述如下：

```
 bool IsSubSet(LinkList La, LinkList Lb)
{
  bool  ok1,ok2;
  LinkList p1,p2;
  ok1=true;
  p1=La->next;
  while(ok1&&p1)
  {
    ok2=false;
    p2=Lb->next;
    while(!ok2&& p2)
    {
      if(p2->data==p1->data)   ok2=true;
      else  p2=p2->next;
    }
    ok1=ok2;
    p1=p1->next;
    }
    return   ok1;
}
```

10. 算法描述如下：

```
void merge_L(LinkList La, LinkList Lb,  LinkList &Lc)
{
  LinkList pa,qa,pb,qb;
  Lc=La; pa=La->next; Lc->next=NULL;
  pb=Lb->next; free(Lb);
  while(pa&&pb)
  {
    if(pa->data<pb->data)
    {
      qa=pa;
      pa=pa->next;
       qa->next=Lc->next;
      Lc->next=qa;
    }
```

```
    else
    if(pa->data>pb->data)
    {
        qb=pb;
        pb=pb->next;
        qb->next=Lc->next;
        Lc->next=qb;
    }
    else
    {
       qa=pa;
       qb=pb;
       pa=pa->next;
       pb=pb->next;
       qa->next=Lc->next;
       Lc->next=qa;
       free(qb);
    }
   }
   while(pa)
   {
     qa=pa;
     pa=pa->next;
     qa->next=Lc->next;
     Lc->next=qa;
   }
   while(pb)
   {
     qb=pb;
     pb=pb->next;
     qb->next=Lc->next;
     Lc->next=qb;
   }
}
```

11. 算法描述如下：

```
void Interaction(LinkList ha,LinkList hb,LinkList &hc)
 {
   LinkList pa,pb,pc;
   LinListSort(ha);
   LinListSort(hb);    //调用第 6 题的排序函数
   pa=ha->next;
   pb=hb->next;
   hc=(LinkList)malloc(sizeof(LNode));
```

```
   hc->next=NULL;
   while(pa && pb)
    if(pa->data<pb->data)
      pa=pa->next;
    else  if(pa->data>pb->data)
            pb=pb->next;
          else
          {
            pc=(LinkList)malloc(sizeof(LNode));
            pc->data=pa->data;
            pc->next=hc->next;
            hc->next=pc;
            pa=pa->next;
            pb=pb->next;
          }
}
```

12. 算法描述如下：

```
void  Decompose(LinkList L,LinkList &ha,LinkList &hb,LinkList &hc)
 {
   LinkList p,q;
   p=L;
   ha=(LNode *)malloc(sizeof(LNode));
   hb=(LNode *)malloc(sizeof(LNode));
   hc=(LNode *)malloc(sizeof(LNode));
   ha->next=ha;
   hb->next=hb;
   hc->next=hc;
   while(p)
   {
     if('A'<=p->data && p->data<='Z'||'a'<=p->data && p->data<='z')
     {
       q=p; p=p->next;
       q->next=ha->next;
       ha->next=q;
     }
     else if('0'<=p->data && p->data<='9')
     {
       q=p; p=p->next;
       q->next=hb->next;
       hb->next=q;
     }
     else
     {
```

```
        q=p; p=p->next;
        q->next=hc->next;
        hc->next=q;
      }
     }
 }
```

13. 算法描述如下：

```
void Delete_L(LinkList L)
{  // 删除带头结点的单链表 L 中值相同的多余结点
  LinkList p,q,s;
  p=L->next;
  while(p->next&&p->next->next)
   {
     q=p;
     while(q->next)
      if(p->data==q->next->data)
      {
        s=q->next;
        q->next=s->next;
        free(s);
      }
      else  q=q->next;
      p=p->next;
     }
}
```

14. 算法描述如下：

```
int CountNode(LinkList head,ElemType x)
{
  int sum=0;
  LinkList p=head->next;
  while(p)
  {
    if(p->data==x)
    sum++;
    p=p->next;
  }
  return sum;
}
```

15. 算法描述如下：

```
void RetNode(LinkList L, LinkList &La, LinkList &Lb)
{
```

```
    LinkList p,pa,pb;
    p=L->next;
    pa=La=(LNode *)malloc(sizeof(LNode));
    pb=Lb=(LNode *)malloc(sizeof(LNode));
    while(p)
    {
      if(p->data%2)
      {
        pa->next=p; p=p->next;
        pa=pa->next;
      }
      else
     {
        pb->next=p; p=p->next;
        pb=pb->next;
      }
     }
    pa->next=NULL;
 pb->next=NULL;
}
```

# 习题三

## 一、选择题

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

## 二、填空题

1. SXSSXSXX
2. 18
3. (m+1)%n
4. 只允许在表的一端进行元素的插入而在另一端进行元素的删除
5. top1+1=top2

## 三、算法设计题

**说明：**以下解答中有关顺序栈的习题要包含头文件：SqStack.h，有关顺序循环队列的习题要包含头文件：SqQueue.h，有关单链表的习题要包含头文件：LinkList.h。

1. 算法描述如下：

```
bool ExpIsCorrect(char  exp[],int n)
{
  SqStack S;
  int i;
  char  x;
```

```
  InitStack_Sq(S);
  for(i=0;i<n;i++)
  {
    if(exp[i]=='('||exp[i]=='['||exp[i]=='{')
      Push_Sq(S,exp[i]);
    else if(exp[i]==')'&&!StackEmpty_Sq(S)&&GetTop_Sq(S,x)&&x=='(')
           Pop_Sq(S,x);
    else if(exp[i]==')'&&!StackEmpty_Sq(S)&&GetTop_Sq(S,x)&&x!='(')
          {  cout<<"左右括号配对次序不正确！"<<endl;
               return  false;
          }
    else if(exp[i]==']'&&!StackEmpty_Sq(S)&&GetTop_Sq(S,x)&&x=='[')
            Pop_Sq(S,x);
     else if(exp[i]==']'&&!StackEmpty_Sq(S)&&GetTop_Sq(S,x)&&x!='[')
           {  cout<<"左右括号配对次序不正确！"<<endl;
                return  false;
           }
    else if(exp[i]=='}'&&!StackEmpty_Sq(S)&&GetTop_Sq(S,x)&&x=='{')
            Pop_Sq(S,x);
    else if(exp[i]=='}'&&!StackEmpty_Sq(S)&&GetTop_Sq(S,x)&&x!='{')
         {  cout<<"括号匹配不正确！"<<endl;
              return  false;
         }
    else if((exp[i]==')'||exp[i]==']'||exp[i]=='}')&&StackEmpty_Sq(S))
         {
              cout<<"右括号多于左括号或配对次序有问题！"<<endl;
              return false;
         }
     }
  if(!StackEmpty_Sq(S))
  {
    cout<<"左括号多于右括号！"<<endl;
    return  false;
  }
  else
  {
    cout<<"括号匹配成功！"<<endl;
    return true;
  }
}
```

2. 算法描述如下：

```
void LQInitiate(LinkList &Rear)
{
```

```
   if(!(Rear=(LNode *)malloc(sizeof(LNode))))
    exit(1);
   Rear->next=Rear;
 }

bool LQAppend(LinkList &Rear,ElemType x)
{
  LinkList p;
  if(!(p=(LNode *)malloc(sizeof(LNode))))
    return false;
  p->data=x;
  p->next=Rear->next;
  Rear->next=p;
  Rear=p;
  return true;
}

bool LQDelete(LinkList Rear,ElemType &d)
{
   LinkList front,p;
   front=Rear->next;
   if(front->next==front)
   {
      cout<<"队列已空无数据元素出队列！"<<endl;
      return false;
   }
   else
   {
      p=front->next;
      d=p->data;
     front->next=p->next;
     free(p);
     return true;
   }
 }
```

3. 算法描述如下：

```
void Output()
{
  int x;
  SqStack S;
  InitStack_Sq(S);
  cin>>x;
  while(x!=0)
```

```
  {
 Push_Sq(S,x);
    cin>>x;
  }
  while(!StackEmpty_Sq(S))
  {
  Pop_Sq(S,x);
     cout<<x<<" ";
  }
  cout<<endl;
  DestroyStack_Sq(S);           // 撤销顺序栈
}
```

4. 算法描述如下：

```
# define MAX 20
typedef struct
{
  ElemType v[MAX];
  int front,rear,top;
}QueueStack;

void InitQS(QueueStack &QS)
{
     QS.front=QS.rear=-1;
     QS.top=MAX;
}

bool Estack(QueueStack &QS,ElemType e)
{
  if(QS.rear==QS.top)  return  false;
  else
  {
   QS.v[--QS.top]=e;
   return true;
  }
}

bool Equeue(QueueStack &QS,ElemType e)
{
  if(QS.rear==QS.top)  return false;
  else
  {
    QS.v[++QS.rear]=e;
    return true;
```

```
  }
}
```

5. 说明：为简单起见，解答中队列采用静态顺序结构。
算法描述如下：

```
# define QueueSize 100
typedef struct
{
  int  front;
  int  rear;
  int  tag;
  ElemType queue[QueueSize];
}CirQueue_1;

void InitQueue(CirQueue_1 &Q)
{
  Q.front=Q.rear=0;
  Q.tag=0;
}

bool EnQueue(CirQueue_1 &Q,ElemType e)
{
  if(Q.front==Q.rear&&Q.tag)
  {
   cout<<"队列已满，无法进队！"<<endl;
   return false;
  }
  Q.tag=1;
  Q.queue[Q.rear]=e;
  Q.rear=(Q.rear+1)%QueueSize;
  return true;
}

bool DeQueue(CirQueue_1 &Q,ElemType &e)
{
  if(Q.front==Q.rear&&!Q.tag)
  {
    cout<<"队列已空，无法出队！"<<endl;
    return false;
  }
  e=Q.queue[Q.front];
  Q.tag=0;
  Q.front=(Q.front+1)%QueueSize;
  return true;
```

```
}
```

6. 算法描述如下：

```
# define QueueSize 100
typedef struct
{
  int  front;
  int  rear;
  int  count;
  ElemType queue[QueueSize];
}CirQueue_2;

void InitQueue(CirQueue_2 &Q)
{
  Q.front=Q.rear=0;
  Q.count=0;
}

bool EnQueue(CirQueue_2 &Q,ElemType e)
{
  if(Q.front==Q.rear&&Q.count)
  {
   cout<<"队列已满，无法进队！"<<endl;
   return false;
  }
  Q.count++;
  Q.queue[Q.rear]=e;
  Q.rear=(Q.rear+1)%QueueSize;
  return true;
}

bool DeQueue(CirQueue_2 &Q,ElemType &e)
{
  if(!Q.count)
  {
    cout<<"队列已空，无法出队！"<<endl;
    return false;
  }
  e=Q.queue[Q.front];
  Q.count--;
  Q.front=(Q.front+1)%QueueSize;
  return true;
}
```

7. 算法描述如下：

队满条件：Q.length==QueueSize

```
# define QueueSize 100
typedef struct
{
  int  length;
  int  rear;
  ElemType queue[QueueSize];
}CirQueue_3;

void InitQueue_3(CirQueue_3 &Q)
{
  Q.rear=0;
  Q.length=0;
}

bool EnQueue_3(CirQueue_3 &Q,ElemType e)
{
  if(Q.length==QueueSize)
  {
   cout<<"队列已满，无法进队！"<<endl;
   return false;
  }
  Q.queue[Q.rear]=e;
  Q.rear=(Q.rear+1)%QueueSize;
  Q.length++;
  return true;
}

bool DeQueue_3(CirQueue_3 &Q,ElemType &e)
{
  if(Q.length==0)
  {
   cout<<"队列已空，无法出队！"<<endl;
   return false;
  }
  int tmpfront;
  tmpfront=(QueueSize+Q.rear-Q.length)%QueueSize;
  Q.length--;
  e=Q.queue[tmpfront];
  return true;
}
```

8. 算法描述如下：

```
bool IsHuiWen(char *str)
{
  SqQueue Q;
  SqStack S;
  char x,y;
  int i,length=strlen(str);
  InitQueue_Sq(Q);
  InitStack_Sq(S);
  for(i=0;i<length;i++)
  {
    EnQueue_Sq(Q,str[i]);
    Push_Sq(S,str[i]);
  }
  while(!QueueEmpty_Sq(Q)&&!StackEmpty_Sq(S))
  {
    if(DeQueue_Sq(Q,x)&&Pop_Sq(S,y)&&x!=y)
 {
      cout<<str<<"不是回文！"<<endl;
      return false;
    }
  }
  if(!QueueEmpty_Sq(Q)||!StackEmpty_Sq(S))
    {
      cout<<str<<"不是回文！"<<endl;
      return false;
    }
  else
    {
      cout<<str<<"是回文！"<<endl;
      return true;
    }
}
```

9. 算法描述如下：

```
void Creat(LinkList &head )
{
  SqQueue Q[10];
  LinkList p,q;
  int i,j,n,a,x;
  for(i=0;i<10;i++)
  InitQueue_Sq(Q[i]);
  cout<<"请输入数据的个数：";
  cin>>n;
  cout<<"请输入"<<n<<"个 10 以内的整数：";
```

```
  for(i=0;i<n;i++)
  {
   cin>>a;
    for(j=0;j<10;j++)
   if(a==j) EnQueue_Sq(Q[j],a);
  }
  head=(LinkList)malloc(sizeof(LNode));
  q=head;
  for(i=0;i<10;i++)
    while(!QueueEmpty_Sq(Q[i]))
    {
       DeQueue_Sq(Q[i],x);
       p=(LinkList)malloc(sizeof(LNode));
       p->data=x;
       q->next=p;
       q=p;
    }
  q->next=NULL;
}

void Print(LinkList head)
{
 LinkList p=head->next;
 while(p)
  {  cout<<setw(6)<<p->data;
     p=p->next;
  }
cout<<endl;
}
```

10. [题目分析]栈的特点是后进先出，队列的特点是先进先出。所以，用两个栈 S1 和 S2 模拟一个队列时，S1 作输入栈，逐个元素进栈，以此模拟队列元素的入队。当需要出队时，将栈 S1 退栈并逐个进栈 S2 中，S1 中最先入栈的元素，在 S2 中处于栈顶。S2 退栈，相当于队列的出队，实现了先进先出。显然，只有栈 S2 为空且 S1 也为空，才算是队列空。

算法描述如下：

```
typedef struct
 {
   ElemType stack[MaxStackSize];
   int top;
 }SqStack;
bool EnQueue(SqStack &S1,SqStack &S2,ElemType x)
{    // 栈 S1 和栈 S2 的容量均为 MaxStackSize。本算法将 x 入栈，
     // 若入栈成功返回 true，否则返回 false
   if(S1.top==MaxStackSize&&!StackEmpty(S2)) //S1 满 S2 非空,这时 S1 不能再进栈
```

```
    {
   cout<<"栈满"<<endl;
   return false;
    }
    if(S1.top==MaxStackSize&&StackEmpty(S2))
                                        //若 S2 为空,先将 S1 退栈,元素进 S2 栈
    {
     while(!StackEmpty(S1))
     {
         Pop(S1,x);
         Push(S2,x);
     }
    }
    Push(S1,x);
    return true;                        //x 入栈，实现了队列元素的入队
}

bool DeQueue(SqStack &S1,SqStack &S2,ElemType &x)
{   //S2 是输出栈，本算法将 S2 栈顶元素退栈，实现队列元素的出队
    if(!StackEmpty(S2))                //栈 S2 非空，则直接出队
    {
      Pop(S2,x);
      return true;
    }
    else                                //处理 S2 空栈
    if(StackEmpty(S1))
    {
       cout<<"队列空！"<<endl;
       return false;                    //若输入栈也为空，则判定队空
      }
     else                              //先将栈 S1 倒入 S2 中，再作出队操作
     {
      while(!StackEmpty(S1))
      {
        Pop(S1,x);
        Push(S2,x);
      }
      Pop(S2,x);                       //S2 退栈相当于队列出队
     return true;
    }
 }

bool QueueEmpty(SqStack S1,SqStack S2)
{   //本算法判用栈 S1 和 S2 模拟的队列是否为空
```

```
    if(StackEmpty(S1)&&StackEmpty(S2))          //队列空
      return true;
    else return false ;                          //队列非空
}
```

[算法讨论]算法中假定栈 S1 和栈 S2 容量相同。出队从栈 S2 出，当 S2 为空时，若 S1 不空，则将 S1 倒入 S2 再出栈。入队在 S1，当 S1 满后，若 S2 空，则将 S1 倒入 S2，之后再入队。因此队列的容量为两栈容量之和。元素从栈 S1 倒入 S2，必须在 S2 空的情况下才能进行，即在要求出队操作时，若 S2 空，则不论 S1 元素多少（只要不空），就要全部倒入 S2 中。

# 习题四

## 一、选择题

1. B　2. B　3. C　4. D　5. D

## 二、填空题

1. 顺序存储方式　链接存储方式
2. 两个串的长度相等且对应位置的字符相同
3. 零个字符的串　零
4. 由一个或多个空格字符组成的串　其包含的空格个数
5. 字符
6. 任意个连续的字符组成的子序列
7. 串的整体或串的某一部分子串　串复制　取子串　插入子串　删除子串等

## 三、算法设计题

**说明：**以下解答中有关动态顺序串的习题要包含头文件：DSqString.h，有关单链结构的习题要包含头文件:SLinkString.h,有关顺序循环队列的习题要包含头文件:SqQueue.h，有关单链表的习题要包含头文件：LinkList.h。

1. 本题的算法思想是：从头到尾扫描 S 串，对于值为 ch1 的元素直接替换成 ch2 即可。其算法描述如下：

```
void Trans_Sq(DSqString &S,char ch1,char ch2)
{
  int i;
  for(i=0;i<S.length;i++)
    if(S.str[i]==ch1)S.str[i]=ch2;
}
```

2. 算法描述如下：

```
void Delall_Sq(DSqString &S,char ch)
{
  int i=0,j=0,k;
  k=S.length;
```

```
  while(i<k)
  {
    if(S.str[i]!=ch) S.str[j++]=S.str[i];
    i++;
  }
  S.length=j;
}
```

3. 算法描述如下：

```
void Delall_L(SLinkString &S,char ch)
{
  SLinkString p,q;
  p=S;
  while(p->next)
  {
    if(p->next->str==ch)
    {
      q=p->next;
      p->next=q->next;
      free(q);
    }
    else p=p->next;
  }
}
```

4. 算法描述如下：

```
bool Equalsubstr(DSqString S,DSqString &sub)
{
   int i=0,j,k,head,max,count;
   head=0;                    //head 指向当前发现的最长等值子串的串头
   max=1;                     //max 记录子串的长度
   for(i=0,j=1;i<S.length&&j<S.length;i=j,j++)
   {
     count=1;
     while(S.str[i]==S.str[j]) // 统计当前等值子串的长度
     {
        j++;
        count++;
     }
     if(count>max)              //发现新的最长等值子串，更新 head 和 max
     {
        head=i;
        max=count;
     }
```

```
    }
    if(max>1)
    {
     if(!(sub.str=(char *)malloc(i*sizeof(char))))  // 给串 sub 申请空间
          return false;
     for(k=0;k<max;k++)
         sub.str[k]=S.str[k+head];
     sub.length=max;
     return true;
    }
    else return false;
}
```

5. 算法描述如下：

```
bool Replacestr_Sq(DSqString &S,int i,int j,DSqString T)
{
    int k,disc;
    if(i<0||i>S.length||j<0||j>S.length||i==j)  return false;
    disc=j-i-1;
    if(T.length<disc)
      for(k=j;k<S.length;k++)
      S.str[k-disc+T.length]=S.str[k];
    else  if(T.length>disc)
          {
if(!(S.str=(char )realloc(S.str,(S.length+T.length-disc)*sizeof(char))))
              return false;
             for(k=S.length-1;k>=j;k--)
             S.str[k+T.length-disc]=S.str[k];
          }
     for(k=0;k<T.length;k++)
      S.str[i+k+1]=T.str[k];
S.length=S.length+T.length-disc;
     return true;
}
```

6. 算法描述如下：

```
bool Replacestr_L(SLinkString &S,int i,int j,SLinkString T)
{
    int k;
    SLinkString p,q,r,h;
    if(i<0||j<0||i==j)  return false;
    q=p=S;
    k=0;
    while(p->next&&k<i)          // 寻找第 i 个结点
```

```
    {
      p=p->next;
      k++;
    }
    if(k!=i)   return false;
    k=0;
    while(q->next&&k<j)          // 寻找第 j 个结点
    {
      q=q->next;
      k++;
    }
    if(k!=j)   return false;

    r=T->next;
    while(r->next)               // r 指向串 T 的表尾
      r=r->next;
    h=p->next;
    p->next=T->next;             // 删除第 i 到第 j 个结点,并插入串 T
    free(T);
    r->next=q;
    while(h!=q&&h->next!=q)      // 释放空间
    {
       r=h;
       h=h->next;
       free(r);
    }
    return true;
}
```

7. 算法描述如下：

```
bool pattern_index(DSqString S,DSqString T,int &pos)
{
    int i=0,j=0;                          // i 和 j 分别扫描主串 S 和子串 T
    while(i<S.length&&j<T.length)
    {
     if(S.str[i]==T.str[j]||T.str[j]=='?')// 对应字符相同，继续比较下一个字符
     { i++;    j++;
     }
     else                                 // 主串指针回溯重新开始下一次匹配
     { i=i-j+1;
       j=0;
     }
    }
    if(j==T.length)  { pos=i-T.length; return true;  }
```

```
   else return false;
}// Index_Sq
```

8. 算法描述如下：

```
int Substr_count(DSqString substr,DSqString str)
{
   int i=0,j,k,count=0;
   for(i=0;i<str.length;i++)
     for(j=i,k=0;str.str[j]==substr.str[k];j++,k++)
       if(k==substr.length-1)
   count++;
   return(count);
}
```

9. 算法描述如下：

```
void Trans_L(SLinkString &S,char x,char y)
{
  SLinkString p;
  p=S->next;
  while(p)
  {
    if(p->str==x) p->str=y;
    p=p->next;
  }
}
```

10. 算法描述如下：

```
bool Found(SLinkString head,char x)
{   // 若 x 在链表 head 中,返回 true;否则返回 false
  while(head->next&&head->next->str!=x)
     head=head->next;
  if(head->next) return true;
  else return false;
}

bool FindFirst(SLinkString S,SLinkString T,char &ch)
{
  SLinkString p;
  if(!S->next) { cout<<"S 为空!"<<endl; return false;  }
  else
  {   p=S->next;
    while(Found(T,p->str))
       p=p->next;
    ch=p->str;
```

```
  }
}
```

11.本题采用动态顺序结构存储从键盘输入的字符串。
算法描述如下：

```
void Count(DSqString S)
{
  int i,j,num[36];
  for(i=0;i<36;i++)
   num[i]=0;
  for(j=0;j<S.length;j++)
    if(S.str[j]>='0'&&S.str[j]<='9')
     {
       i=S.str[j]-'0';
       num[i]++;
     }
    else if('A'<=S.str[j]&&S.str[j]<='Z')
     {
        i=S.str[j]-'A'+10;
        num[i]++;
     }
    for(i=0;i<10;i++)
      printf("数字%d 的个数=%d\n",i,num[i]);
    for(i=10;i<36;i++)
      printf("字母字符%c 的个数=%d\n",i+55,num[i]);
}
```

12. 算法描述如下：
块链结构的定义如下：

```
# define Number 80          // 可由用户定义的结点大小
# include "stdlib.h"
typedef struct Chunk {
char str[Number];           // 一个结点存放 Number 个字符
  struct Chunk *next;
}Chunk;                     // 结点类型定义
typedef struct BLhead{
Chunk  *head,*tail;         // 串的头尾指针
int  length;                // 串的当前长度
}BLhead,*BLinkString;

bool SubStr_BL(BLinkString S, BLinkString &sub, int pos, int len)
{
  Chunk *p,*q;
  int j,n,m,w,l,u;
```

```
  n=S->length;
  p=S->head;
  if(pos>=0&&pos<n&&len>=0&&len<=n-pos)
  {
     m=pos/Number+1;              //m 指向取子串的起始结点
     for(j=1;j<m;j++)             //p 指向第 m 个结点
      p=p->next;
     sub=(BLhead *)malloc(sizeof(BLhead));   //创建子串头结点
     sub->length=len;
     sub->head=(Chunk *)malloc(sizeof(Chunk)); //创建子串的第一个结点
     q=sub->head;
     w=1;                                  //指向子串中的当前结点
     u=m;                                  //指向主串中的当前结点
     l=pos;                                //指向主串中的当前位置

     for(j=0;j<len;j++)                    //创建子串
     {
      if(j>=Number *w)           //所取字符的个数超过一个结点中的最大字符数
      {
       w++;                                        //子串结点数加 1
       q->next=(Chunk *)malloc(sizeof(Chunk));  //创建新结点
       q=q->next;                                  //指向子串中下一个结点
      }
      if(l>=Number *u)       //主串中的字符的个数超过一个结点中的最大字符数
      {
       u++;                                  //主串结点数加 1
       p=p->next;                            //指向主串中下一个结点
      }
      q->str[j%Number]=p->str[l%Number]; //给子串中的相应结点赋值
      l++;                                  //指向主串中的下一个字符
     }
      return true;
  }
  else  return false;
}
```

13. 算法描述如下：

```
void DSqstrSort(DSqString &S)          //对串进行排序
{
 int i,j,t;
 for(i=0; i<S.length; i++)
  for(j=i+1; j<S.length; j++)
   if(S.str[i]>S.str[j])
    {
```

```
      t=S.str[i];
      S.str[i]=S.str[j];
      S.str[j]=t;
    }
}

bool Replacestr_Sq(DSqString S,DSqString T,DSqString &V)
{
   int i=0,j=0,k=0;
   if(!S.length||!T.length)
   {
      cout<<"有一个串为空串！"<<endl;
      return false;
   }
   DSqstrSort(S);          //调用排序函数
   DSqstrSort(T);
   if(!(V.str=(char *)malloc(i*sizeof(char))))  // 给串 V 申请空间
    return false;
   while(i<S.length&&j<T.length)
     if(S.str[i]<T.str[j])
        i++;
     else  if(S.str[i]>T.str[j])
             j++;
           else
           {
             V.str[k]=S.str[i];
             i++;j++;k++;
           }
   V.length=k;
   return true;
}
```

14. 算法描述如下：

```
void Delsubstr_Sq(DSqString &S,DSqString T)
{
   int  pos;
   while(Index_Sq(S,T,pos))                    // 判断 T 是否是 S 的子串
   StrDelete_Sq(S,pos,T.length);               // 删除子串 T
}
```

15. 本题要求字符串 S1 拆分成字符串 S2 和字符串 S3，要求字符串 S2“按给定长度 n 格式化成两端对齐的字符串”，即长度为 n 且首尾字符不得为空格字符。算法从左到右扫描字符串 S1，找到第一个非空格字符，计数到 n，第 n 个拷入字符串 S2 的字符不得为空格，然后将余下字符复制到字符串 S3 中。

算法描述如下：

```
bool format(DSqString S1,DSqString &S2,DSqString &S3,int n)
{   //将字符串 S1 拆分成字符串 S2 和字符串 S3，要求字符串 S2 是长 n 且两端对齐
  int i=0,j=0,k=0;
  if(!(S2.str=(char *)malloc(n*sizeof(char))))  // 给串 S2 申请空间
    return false;
  while(i<S1.length&&S1.str[i]==' ')  //滤掉 S1 左端空格
     i++;
  if(i==S1.length)
  {
     cout<<"字符串 S1 为空串或空格串"<<endl;
     return false;
  }
  while(i<S1.length&&k<n)           //字符串 S1 向字符串 S2 中复制
  {
     S2.str[j]=S1.str[i];
     i++; j++; k++;
  }
  if(i==S1.length)
  {
     cout<<"字符串 S1 没有"<<n<<"个有效字符"<<endl;
     return false;
  }
  if(S2.str[--j]==' ') //若最后一个字符为空格，则需向后找到第一个非空格字符
  {
    i--;              //i 后退
   while(S1.str[i]==' '&&i<S1.length) //往后查找一个非空格字符作串 S2 的尾字符
     i++;
   if(i==S1.length)
    {
      cout<<"S1 串没有"<<n<<"个两端对齐的字符串"<<endl;
      return false;
    }
   S2.str[j]=S1.str[i];       //字符串 S2 最后一个非空字符
     i++;
  }
  S2.length=n;
  j=0;                    //将 S1 串其余部分送字符串 S3。
  if(!(S3.str=(char *)malloc((S1.length-i)*sizeof(char)))) // 给串 S2 申请空间
    return false;
  while (i<S1.length)
    S3.str[j++]=S1.str[i++];
  S3.length=j;
  return true;
```

}

# 习题五

## 一、选择题

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

## 二、填空题

1. 332　　326
2. 42
3. 线性表
4. 深度
5. 多层次
6. ( )　　(( ))　　2　　2
7. GetHead(GetHead(GetTail(GetTail(GetHead(GetTail(GetTail(A)))))))
8. (d,e)

## 三、简述题

1. 解答：

⑴三元组线性表为：

(6,6,6),(0,0,3),(0,4,6),(1,4,16),(2,0,25),(3,2,66),(4,5,88)

⑵三元组顺序表如图 5.1 所示：

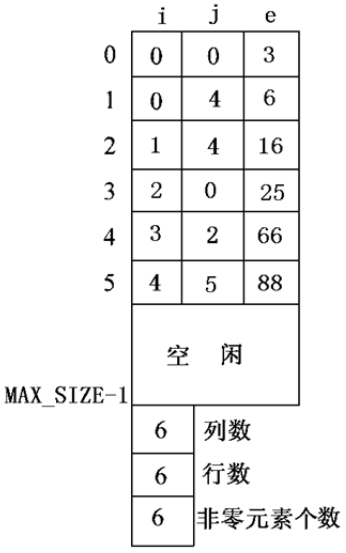



图 5.1　三元组顺序表

⑶带头结点的三元组单链表如图 5.2 所示：

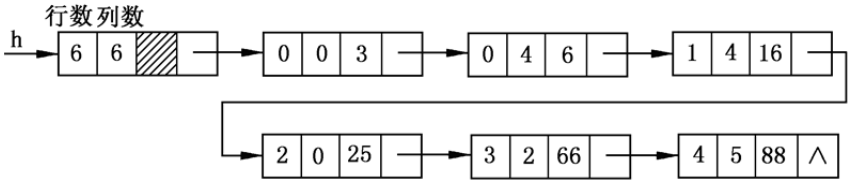


图 5.2　三元组单链表

⑷三元组行指针数组链表如图 5.3 所示：

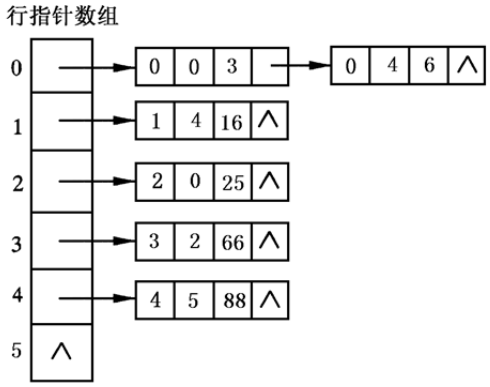



图 5.3　三元组行指针数组链表

⑸三元组十字链表如图 5.4 所示：

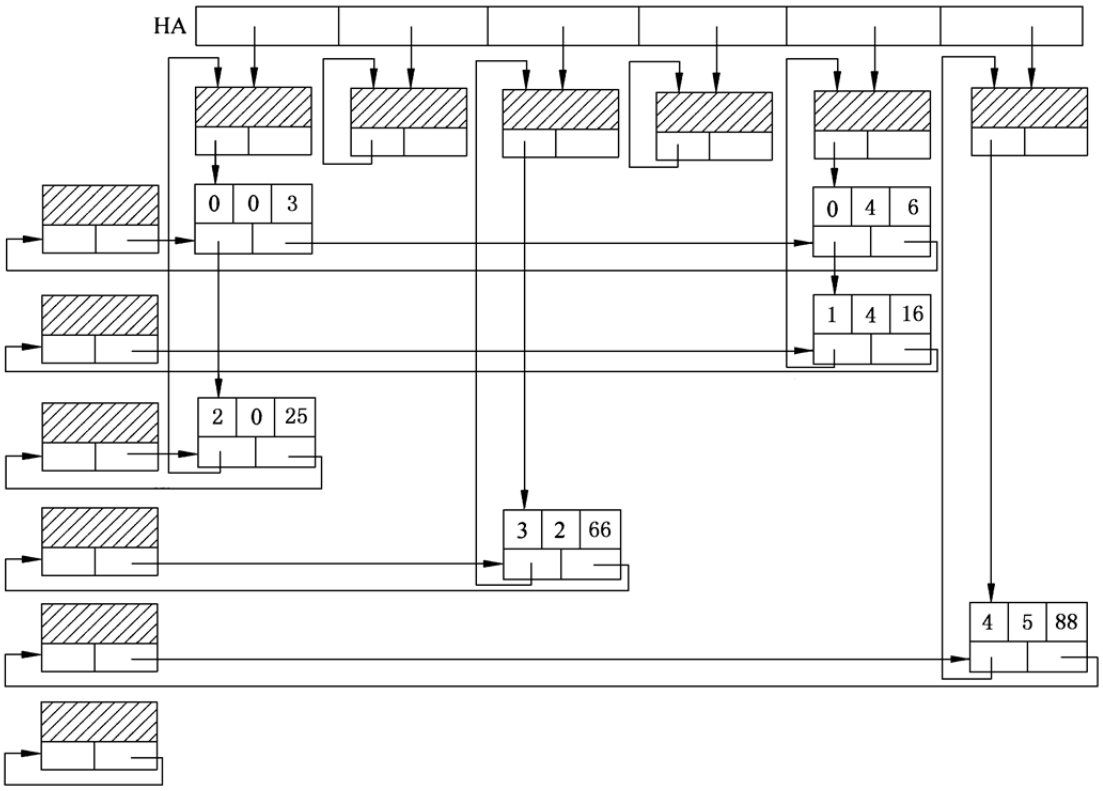



图 5.4　三元组十字链表

2．解答：

⑴对称矩阵 A 如图 5.5 所示：

$$\mathbf{A}=\begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 5 \\ 2 & 0 & 5 & 0 \end{bmatrix}$$

图 5.5　对称矩阵

⑵A 中任一元素的行列下标[i,j](0≤i,j<4)与 S 中元素的下标 k 之间的关系如下：

一个 n 阶对称矩阵 A，将 A 中包括主对角线的下三角元素按列的顺序压缩到数组 S 中，则A中任一元素的行列下标[i,j](0≤i,j<n与S中元素的下标k之间的对应关系推导如下：

当 i≥j 时，A[i][j]前面共有 j 列元素，每列元素的个数分别为：

n　n-1　n-2　…n-(-1)

A[i][j]所在列上面有 i-j 个元素，所以 A[i][j]前面共有元素个数为：

n+(n-1)+(n-2)+…+(n-(j-1))+-i-j=(2n-j+1)/2*j+i-j

当 i<j 时，A[i][j]前面元素的个数为：(2n-i+1)/2*i+j-i

所以 A 中任一元素的行列下标[i,j](0≤i,j<n)与 S 中元素的下标 k 之间的关系为（本是中 n=4)：

$$k=\begin{cases}\dfrac{2n-j+1}{2}\times j+i-j & i\geq j\\ \dfrac{2n-i+1}{2}\times i+j-i & i<j\end{cases}$$

⑶稀疏矩阵 A 的三元组线性表为：

(4,4,6),(0,0,1),(0,3,2),(1,1,3),(2,3,5),(3,0,2),(3,2,5)。其中第一个三元组是稀疏矩阵行数、列数和非零元素个数。其它三元组均为非零元素行值、列值和元素值。

3．解答：

第一种存储结构图如图 5.6 所示：

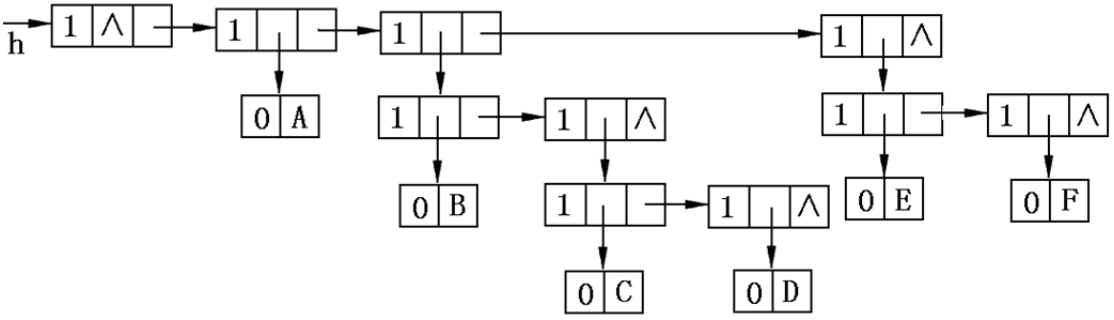



图 5.6　广义表的头尾链表存储结构

第二种存储结构图如图 5.7 所示：

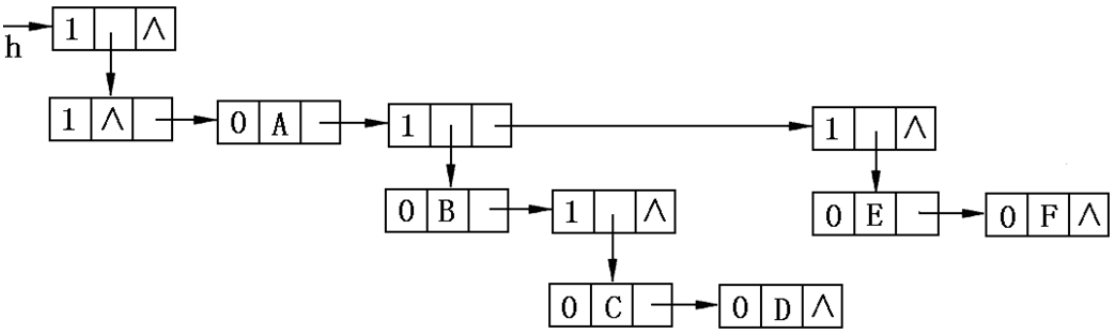



图 5.7　广义表的扩展线性链表存储结构

**四、算法设计题**

**说明：**以下解答中有关三元组顺序表的习题要包含头文件：TSMatrix.h，有关动太数组的习题要包含头文件：Array.h，有关广义表的习题要包含头文件：GList.h。

1. 算法描述如下：

```
void Minmax(Array A,int m,int n)
{
  int i,j,have=0;
  ElemType e,*min,*max;
  min=(ElemType *)malloc(m*sizeof(ElemType));
  max=(ElemType *)malloc(n*sizeof(ElemType));
  for(i=0;i<m;i++)
  {
    Value(min[i],A,i,0);              // 将A[i][0]的值赋给min[i]
    for(j=1;j<n;j++)
    {
      Value(e,A,i,j);                // 将A[i][j]的值赋给e
      if(e<min[i]) min[i]=e;
    }
  }
  for(j=0;j<n;j++)
  {
    Value(max[j],A,0,j);              // 将A[0][j]的值赋给max[j]
    for(i=1;i<m;i++)
    {
      Value(e,A,i,j);                // 将A[i][j]的值赋给e
      if(e>max[j]) max[j]=e;
    }
  }

  for(i=0;i<m;i++)
    for(j=0;j<n;j++)
      if(min[i]==max[j])
      {
        Value(e,A,i,j);               // 将A[i][j]的值赋给e
        cout<<"("<<i<<","<<j<<"):"<<e<<endl;
        have=1;
      }
  if(!have)
  cout<<"没有鞍点!"<<endl;
}
```

2.

⑴算法描述如下：

```
void Enter(ElemType A[],int n)
{
 int i;
```

```
 for(i=0;i<n*(n+1)/2;i++)
     cin>>A[i];
}
```

⑵算法描述如下：

```
void Add(ElemType A[],ElemType B[],ElemType C[],int n)
{
  int i;
   for(i=0;i<n*(n+1)/2;i++)
     C[i]=A[i]+B[i];
}
```

⑶算法描述如下：

```
void Mult(ElemType A[],ElemType B[],ElemType D[][N],int n)
{
   int i,j,k,p,q,sum;
   for(i=0;i<n;i++)
     for(j=0;j<n;j++)
     {
        sum=0;
        for(k=0;k<n;k++)
        {
         if(i>=k) p=i*(i+1)/2+k;
          else p=k*(k+1)/2+i;

         if(j>=k) q=j*(j+1)/2+k;
          else q=k*(k+1)/2+j;
          sum=sum+A[p]*B[q];
        }
        D[i][j]=sum;
     }
}
```

⑷算法描述如下：

```
void Print(ElemType A[],int n)
{
  int i,j;
  for(i=0;i<n;i++)
  {
     for(j=0;j<n;j++)
         if(i>=j) cout<<setw(2)<<A[i*(i+1)/2+j];
         else  cout<<setw(2)<<A[j*(j+1)/2+i];
     cout<<endl;
  }
```

```
}
```

⑸主函数如下：

```
void main()
{
   ElemType A[N],B[N],C[N],D[N][N];
   int i,j,n;
   cout<<"请输入矩阵的阶数：";
   cin>>n;
   cout<<"请输入矩阵 A："<<endl;
   Enter(A,n);
   cout<<"请输入矩阵 B："<<endl;
   Enter(B,n);
   cout<<"矩阵 A 为："<<endl;
   Print(A,n);
   cout<<"矩阵 B 为："<<endl;
   Print(B,n);
   Add(A,B,C,n);
   cout<<"矩阵 C 为："<<endl;
   Print(C,n);
   Mult(A,B,D,n);
   cout<<"矩阵 D 为："<<endl;
   for(i=0;i<n;i++)
   {
     for(j=0;j<n;j++)
     cout<<setw(5)<<D[i][j];
     cout<<endl;
   }
}
```

3．算法描述如下：

```
int Count(TSMatrix M)
{
  int sum=0;
  for(int i=0;i<M.t;i++)
   if(M.data[i].i==M.data[i].j||M.data[i].i+M.data[i].j==M.n-1)
     sum=sum+M.data[i].e;
  return sum;
}
```

4．题目中要求矩阵每行元素的平均值按递增顺序排序，由于每行元素个数相等，按平均值排列与按每行元素之和排列是一个意思。所以应先求出各行元素之和，放入一维数组中，然后选择一种排序方法，对该数组进行排序，注意在排序时若有元素移动，则与之相应的行中各元素也必须做相应变动。其算法描述如下：

```
void Translation(Array &A,int n)
{  //对 n×n 的矩阵 matrix，通过行变换，使其各行元素的平均值按递增排列
  int i,j,k;
  ElemType sum,min;              //sum 暂存各行元素之和
  ElemType *p, pi, pk;
  p=(ElemType *)malloc(n*sizeof(float));
  for(i=0; i<n; i++)
  {
     Value(pk,A,i,0);              // 将 A[i][0]的值赋给 pk
     sum=pk;
    for(j=1;j<n;j++)              //求一行元素之和
    {
      Value(pk,A,i,j);          // 将 A[i][j]的值赋给 pk
      sum+=pk;
     }
    *(p+i)=sum;                    //将一行元素之和存入一维数组
  }

  for(i=0; i<n-1; i++)             //用选择法对数组 p 进行排序
   {
     min=*(p+i); k=i;              //初始设第 i 行元素之和最小
    for(j=i+1;j<n;j++)
      if(p[j]<min)                //记新的最小值及行号
       {
          k=j; min=p[j];
       }
    if(i!=k)          //若最小行不是当前行,要进行交换(行元素及行元素之和)
    {
       for(j=0;j<n;j++)            //交换两行中对应元素
       {
         Value(pk,A,k,j);        // 将 A[k][j]的值赋给 pk
         Value(pi,A,i,j);        // 将 A[i][j]的值赋给 pi
         Assign(A,pi,k,j);        // 将 pi 赋值给 A[k][j]
         Assign(A,pk,i,j);        // 将 pk 赋值给 A[i][j]
       }
     sum=p[i]; p[i]=p[k]; p[k]=sum; //交换一维数组中元素之和.
   }
  }
  free(p);
}
```

5. 算法描述如下：

```
void Union(Array A,Array B,Array &C,int m,int n)
{//将长度为 m 的递增有序数组 A 和长度为 n 的递减有序数组 B 归并为递增有序的数组 C
```

```
    int i=0,j=n-1,k=0;          // i,j,k 分别是数组 A,B 和 C 的下标
    ElemType e1,e2;
    while(i<m && j>=0)
    {
      Value(e1,A,i);              // 将 A[i]的值赋给 e1
      Value(e2,B,j);              // 将 B[j]的值赋给 e2
      if(e1<e2)
     {
       Assign(C,e1,k);           // 将 e1 赋给 C[k]
       k++;  i++;
     }
     else
     {
       Assign(C,e2,k);           // 将 e2 赋给 C[k]
       k++;  j--;
    }
   }
   while(i<m)
   {
     Value(e1,A,i);           // 将 A[i]的值赋给 e1
     Assign(C,e1,k);          // 将 e1 赋给 C[k]
     k++;  i++;
   }

   while(j>=0)
   {
     Value(e2,B,j);            // 将 A[i]的值赋给 e1
     Assign(C,e2,k);           // 将 e2 赋给 C[k]
     k++;  j--;
   }
}
```

6. 算法描述如下：

```
int GListAtomNum(GList GL)
{
   if(!GL)  return  0;
   if(GL->tag==ATOM) return 1;
   if(GL->tag==LIST)
      return GListAtomNum(GL->ptr.hp)+GListAtomNum(GL->ptr.tp);
}
```

7. 算法描述如下：

```
int GListAtomSum(GList GL)
{
```

```
   if(!GL)  return  0;
   if(GL->tag==ATOM) return GL->data;
   if(GL->tag==LIST)
      return GListAtomSum(GL->ptr.hp)+GListAtomSum(GL->ptr.tp);
}
```

8. 算法描述如下：

```
GList GListLocateAtom(GList GL,ElemType x)
{
   GList p;
   if(!GL)  return  NULL;
   if(GL->tag==ATOM&&GL->data==x) return GL;
   if(GL->tag==LIST&&GL->ptr.hp)
   {
      p=GListLocateAtom(GL->ptr.hp,x);
      if(p)  return p;
   }
   if(GL->tag==LIST&&GL->ptr.tp)
    {
      p=GListLocateAtom(GL->ptr.tp,x);
      if(p)  return p;
   }
   return NULL;
}
```

9. 算法描述如下：

```
bool  GListSame(GList p, GList q)
{
   bool flag=true;
   if(p&&q)
   {
     if(p->tag==ATOM&&q->tag==ATOM)
       if(p->data!=q->data)  flag=false;
        else flag=true;
            else if(p->tag==LIST&&q->tag==LIST)

       flag=GListSame(p->ptr.hp,q->ptr.hp)&&GListSame(p->ptr.tp,q->ptr.tp);
   }
   else
   {
      if(!p&&q) flag=false;
      if(p&&!q) flag=false;
   }
   return(flag);
```

```
}
```

10．算法描述如下：

```
void GList_DelElem(GList &A,int x)
{  //从广义表 A 中删除所有值为 x 的原子
   GList  q;
   if(!A) return;
  if(A->ptr.hp&&A->ptr.hp->tag==LIST) GList_DelElem(A->ptr.hp,x);
  else  if(A->ptr.hp&&A->ptr.hp->tag==ATOM&&A->ptr.hp->data==x)
  { q=A; A=A->ptr.tp;              //删去元素值为 x 的表头
    free(q);
   return;
  }
  if(A->ptr.tp) GList_DelElem(A->ptr.tp,x);
}//GList_DelElem
```

# 习题六

## 一、选择题

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

## 二、填空题

1．$2^{k-1}$　$2^k-1$

2．0　(n-1)/2　(n+1)/2　$\lfloor \log_2 n \rfloor+1$

3．$2^{k-2}+1$（第 k 层 1 个结点，总结点个数是 $2^{k-1}$，编号最小的叶子在第 k-1 层最左边，其双亲是 $2^{H-1}/2=2^{k-2}$）　$2^{k-1}$　$\lfloor \log_2 i \rfloor+1$

4．完全二叉树　单枝树（树中任一结点（除最后一个结点是叶子外），只有左子女或只有右子女）

5．先序　中序

6．任何结点至多只有右孩子的二叉树

7．二叉树

8．6　261

9．(n×(k-1)+1)/k

设叶子结点个数为 $n_0$，则有分枝数 $n-1=(n-n_0)\times k$，即 $n_0=(n\times(k-1)+1)/k$

10．$2^{n-1}$

n 个结点且高度为 n 的二叉树为单支树，实际上是在高度为 n 的满二叉树中寻找该单支树的不同排列，即走了一条由根到达所有叶子的过程，它恰好为满二叉树的叶子数。（n 个结点的满二叉树的叶子数为(n+1)/2）

11．空指针域　存放该结点的前驱或后继信息

12．2　5

13．2n-1

14．n+1

15．先序遍历

**三、简述题**

1．解答：

目的：树和森林采用二叉树的存储结构，可以利用二叉树的已有算法解决树和森林的有关问题；

主要区别：树中结点的最大度数没有限制，而二叉树结点的最大度数为 2；树的结点无左右之分，而二叉树的结点有左右之分。

2．解答：

线性表属于约束最强的线性结构，在非空线性表中，只有一个“第一个”元素，也只有一个“最后一个”元素；除第一个元素外，每个元素有唯一前驱；除最后一个元素外，每个元素有唯一后继。树是一种层次结构，有且只有一个根结点，除叶子结点外（叶子结点无后继），每个结点可以有多个后继（孩子），除根结点外，每个结点只有一个前驱（双亲），根结点无前驱，从这个意义上说存在一（双亲）对多（孩子）的关系。广义表中的元素既可以是原子，也可以是子表，子表可以为它表共享。从表中套表意义上说，广义表也是层次结构。从逻辑上讲，树和广义表均属非线性结构。但在以下意义上，又蜕变为线性结构。如度为 1 的树，以及广义表中的元素都是原子时。另外，广义表从元素之间的关系可看成前驱和后继，也符合线性表，但这时元素有原子，也有子表，即元素并不属于同一数据对象。

3．解答：

⑴第 i 层结点数为 $d^{i-1}$

⑵当 n= 1 时，该结点为根，无双亲结点；否则，双亲结点的编号为$\lfloor (n+d-2)/d \rfloor$。

⑶当(n-1)d+i+1 小于等于树的总结点个数时，编号为 n 的结点存在第 i 个孩子结点，其编号为(n-1)d+i+1。

⑷当 (n-1)%d≠0 且 n+1 小于等于树的总结点个数时，编号为 n 的结点有右兄弟，其右兄弟的编号是 n+1。

分析：这个问题实际上是由完全 k 叉树的特点来求结点编号之间的关系，类似于完全二叉树的性质。

⑴在深度为 k 的满 d 叉树中，第 1 层有 1 个结点，第 2 层有 k 个结点……第 i 层有 $d^{i-1}$ 个结点。

⑵设编号为 n 的结点是在满 d 叉树的第 1 层上从左边数第 $j$ 个结点，那么

$$j = n - \frac{d^l - 1}{d - 1}$$

这 $j$ 个结点对应有$\lfloor (j-1)/d \rfloor + 1$个双亲结点。因此，编号为 n 的结点的双亲结点是第 $l-1$层的第$\lfloor (j-1)/d \rfloor + 1$个结点，其编号 p 为：

$$p = \frac{d^{l-1} - 1}{d - 1} + \left\lfloor \frac{j-1}{d} \right\rfloor + 1 = \left\lfloor \frac{d^{l-1} - 1}{d - 1} + \frac{j-1}{d} + 1 \right\rfloor + 1$$

将 $j = n - \frac{d^l - 1}{d - 1}$ 代入上式，化简得： $p = \left\lfloor \frac{n+d-2}{d} \right\rfloor$。

⑶设编号为n的结点是在满d叉树的第1层上从左边数第j个结点，那么n就等于前$l-1$层的结点数目加$j$，即：

$$n=\frac{d^{l-1}}{d-1}+j \qquad 或 \quad j=n-\frac{d^{l-1}-1}{d-1}$$

由于满 d 叉树的第 1 层上的第 j 个结点左边有 j-1 个结点，它们共有(j-1)d 个孩子。因此第 j 个结点的第 i 个孩子是第 l+1 层上从左边数第(j-1)d+i 个结点，其编号 p 为：

$$p=\frac{d^{l}-1}{d-1}+d(j-1)+i$$

将$j=n-\frac{d^{l-1}-1}{d-1}$代入上式，化简得：$p=(n-1)d+i+1$

⑷编号为n的结点不是其双亲结点的第d个孩子时，则它有右兄弟。根据⑵和⑶的结论，它的父结点编号为$\left\lfloor\frac{n+d-2}{d}\right\rfloor$，而它的双亲结点的第 d 个孩子编号为：

$$\left(\left\lfloor\frac{n+d-2}{d}\right\rfloor-1\right)d+d+1=d\left\lfloor\frac{n+d-2}{d}\right\rfloor-1$$

因此，当$n\neq d\left\lfloor\frac{n+d-2}{d}\right\rfloor+1$，即$\frac{n-1}{d}\neq\left\lfloor\frac{n+d-2}{d}\right\rfloor$时，编号为 n 的结点有右兄弟，也就是说当 n-1 不能被 d 整除时，它有右兄弟，其右兄弟的编号为 n+ 1。

4．解答：

设树中有$n_0$个叶子结点，那么树中总结点数目 N 为

$$N=n_0+n_1+\cdots+n_m \qquad （a）$$

另一方面，树中除根结点外，其他各结点都有且仅有一个分支指向它，所以树中的总结点个数恰好比分支数多 1。如果 B 是树中的总分支数，即有：

$$N = B + 1$$

但是，除度为 0 的结点没有分支外，每个度为 k 的结点有 k 个分支，所以总分支数为：

$$B=n_1+2\times n_2+\cdots+m\times n_m$$

即总结点数目为：

$$N=n_1+2\times n_2+\cdots+m\times n_m+1 \qquad （b）$$

由（a）和（b）两个等式有

$$n_0+n_1+\cdots+n_m=n_1+2\times n_2+\cdots+m\times n_m+1$$

即得：

$$n_0 = 1\times n_2 + 2\times n_3 + \cdots + (i-1)\times n_i + \cdots + (m-1)\times n_m + 1 = \sum_{i=2}^{m}(i-1)\times n_i + 1$$

5．解答：

先序序列：ABDFGHCE

中序序列：BFDHGACE]

后序序列：FHGDBECA

层序序列：ABCDEFGH

6．解答：

⑴若先序序列与后序序列相同，则或为空树，或为只有根结点的二叉树。

⑵若中序序列与后序序列相同，则或为空树，或为任一结点至多只有左子树的二叉树。

⑶若先序序列与中序序列相同，则或为空树，或为任一结点至多只有右子树的二叉树。

⑷若中序序列与层次遍历序列相同，则或为空树，或为任一结点至多只有右子树的二叉树。

7．解答：

⑴二叉树如图 7.1 所示：

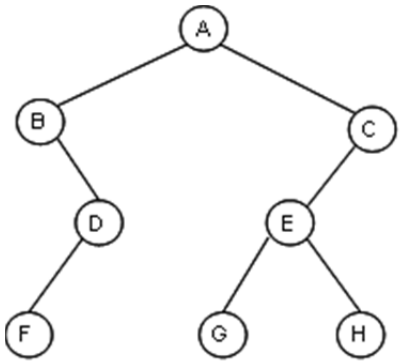



图 7.1　二叉树

⑵该二叉树的先序线索二叉树如图 7.2 所示：

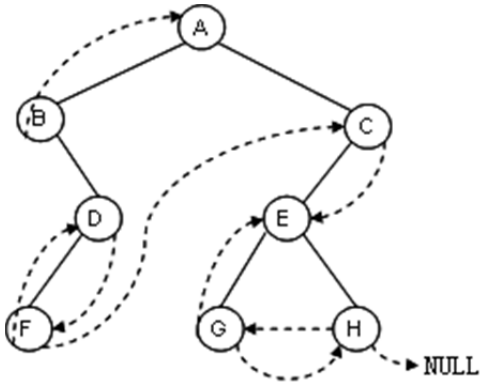



图 7.2　先序线索二叉树

该二叉树的中序线索二叉树如图 7.3 所示：

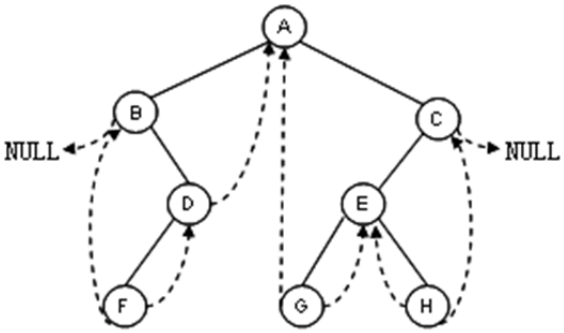



图 7.3　中序线索二叉树

该二叉树的后序线索二叉树如图 7.4 所示：

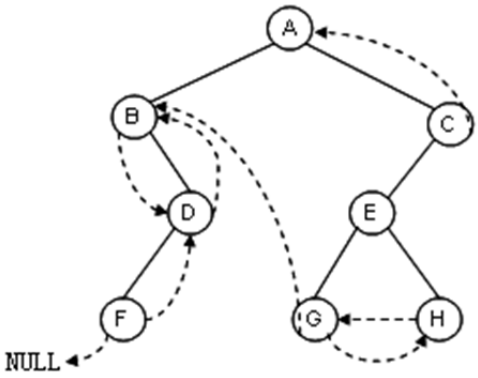



图 7.4　的序线索二叉树

8．解答：

虽然哈夫曼树的带权路径长度是唯一的，但形态不唯一。

本题各字母编码如下：

c1:0110　c2:10　c3:0010　c4:0111　c5:000　c6:010　c7:11　c8:0011

9．解答：

树的先序遍历序列：ABEFCGIJKDH

树的后序遍历序列：EFBIJKGCHDA

转换后二叉树如图 7.5 所示：

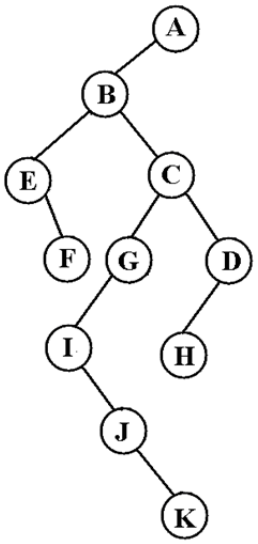


图 7.5　转换后的二叉树

其先序遍历序列：ABEFCGIJKDH

其中序遍历序列：EFBIJKGCHDA

其后序遍历序列：FEKJIGHDCBA

对比遍历树和其对应的二叉树的结果，可得出如下结论：

⑴先序遍历树与先序遍历该树所对应的二叉树具有相同的遍历结果，即它们的先序序列是相同的；

⑵后序遍历树与中序遍历与该树所对应的二叉树具有相同的遍历结果。

## 四、算法设计题

**说明：**以下解答中有关二叉树的习题要包含头文件：BiTree.h。

1．算法描述如下：

```
BiTree CreatTree(TElemType *nodelist,int position)
{
   BiTree p;
   if(nodelist[position]==0||position>15)  return NULL;
   else
   {
       p=(BiTree)malloc(sizeof(BiTNode));
       p->data=nodelist[position];
       p->lchild=CreatTree(nodelist,2*position);
       p->rchild=CreatTree(nodelist,2*position+1);
       return p;
   }
}
```

2．可以用两种常用的方法求解这个问题。

**方法一**　设置一个初始值为 0 的变量 leafNum 进行计数，在对二叉树进行遍历过程中判断当前所访问的结点是否为叶子结点，若为叶子结点，则 leafNum 加 1。因此当遍历完整个二叉树后，leafNum 的值即为叶子结点的数目。其算法描述如下：

```
int leafNum=0;
void CountLeaf_1(BiTree T)
{
    if(!T)   return;
    if(!T->lchild &&!T->rchild)  leafNum++;
    else{
        CountLeaf_1(T->lchild);
        CountLeaf_1(T->rchild);
    }
}
```

**方法二**　根据二叉树的特点可知：当二叉树为空时，叶子结点总数为 0；当二叉树只有一个结点时，叶子结点数为 1；否则，叶子结点数等于左、右子树叶子结点数之和，因此，可定义二叉树 t 的叶子结点数目 leaf（t）的递归计算模型为：

$$leaf(t)=\begin{cases}0, & \text{当 t = NULL时}\\ 1, & \text{当t为叶子时}\\ leaf(t->Lchild)+leaf(t->Rchild), & \text{否则}\end{cases}$$

根据这个计算模型，可以编写如下算法：

```
int CountLeaf_2(BiTree T)
{
   if(!T)   return 0;
   if(!T->lchild &&!T->rchild)  return 1;
       return(CountLeaf_2(T->lchild)+CountLeaf_2(T->rchild));
}
```

3．解答

⑴递归算法

用前序遍历算法交换二叉树中的各结点的左右子树，其算法描述如下：

```
void Swap_1(BiTree T)
{ BiTree r;
    if(!T)  return;
    else
    {
        r = T->lchild;
        T->lchild=T->rchild;
        T->rchild=r;
        Swap_1(T->lchild);
        Swap_1(T->rchild);
    }
}
```

此算法亦可用后序遍历的递归算法实现，但不宜用中序遍历递归算法实现，因为若用中序遍历的算法进行交换时，则只能交换根结点的左、右孩子。

⑵非递归算法

```
void Swap_2(BiTree T)
{
   int top;
   BiTree temp,stack[max];
   if(T)
   {
     top=1;
      stack[top]=T;                    //根结点入栈
     do{
         T=stack[top];                 //弹出栈顶结点
         top--;                      //退栈
         if(T->lchild||T->rchild)
         {
           temp=T->lchild;            //结点的左右指针交换
```

```
            T->lchild=T->rchild;
            T->rchild=temp;
          }
          if(T->lchild)
          {
            top++;                          //交换后的左指针入栈
            stack[top]=T->lchild;
          }
          if(T->rchild)
          {
             top++;                          //交换后的右指针入栈
             stack[top]=T->rchild;
          }
        }while(top!=0);                     //栈空时结束
    }
}
```

4．如果对二叉树的结点对照完全二叉树进行编号，即根结点编号为1，结点i的左孩子编号为2×i，右孩子编号为2×i＋1，那么对于完全二叉树，结点的编号一定是连续的，此时树中最大的结点编号一定等于树中的结点个数；而对于非完全二叉树，结点编号一定不连续，树中最大的结点编号一定大于树中的结点个数。因此，可以先对二叉树的结点按这种编号原则进行编号，求出树中结点个数和最大结点编号，然后再比较结点个数和最大编号是否相等。基于这种思想可编写如下算法：

```
int Num(BiTree T,int i,int &m)
{//T为子树的根，i为根结点编号,m中保存结点的最大编号,函数返回树中结点的个数
   if(!T)  return 0;
   if(m<i)   m=i;
   return(1+Num(T->lchild,2*i,m)+Num(T->rchild,2*i+1,m));
}

bool CheckTree_1(BiTree T)
{
   int n, maxno=0;
   n=Num(T,1,maxno); //计算结点个数，并求出最大编号maxno
   if(n==maxno)   return true;
   else  return false;
}
```

进一步讨论：本算法充分利用了完全二叉树结点编号连续的特点，根据这个特点，也可以编写另一种判断二叉树是否为完全二叉树的特点。基本思想是将结点按其编号存放在一个一维数组中，如果数组中的结点是连续存放的，则为完全二叉树。算法描述如下：

```
# define N 50                 //定义最大结点个数
bool CheckTree_2(BiTree T)
{
   BiTree sa[N+1];
```

```
  int i,n;
  if(!T) return true;              //空二叉树为完全二叉树
  for(i=1;i<=N;i++)
    sa[i]=NULL;                    //初始化数组 sa
  i=n=1;
  sa[1]=T;                         //将根结点指针存放到 sa 的第 1 个位置
  while(i<=n)
 {
    if(!sa[i])   return false;     //sa 中结点不连续
    if(sa[i]->lchild)
     {
         n=2*i;                    //对左孩子编号
         sa[n]=sa[i]->lchild;      //将左孩子指针存放到 sa 的第 n 个位置
     }
     if(sa[i]->rchild)
     {
         n=2*i+1;                  //对右孩子编号
         sa[n]=sa[i]->rchild;      //将右孩子指针存放到 sa 的第 n 个位置*/
     }
     i++;
    }
    return true;
}
```

5．采用先序遍历方法查找值为 x 的结点所有的层次。对应的算法描述如下：

```
void Level(BiTree T,TElemType x,int h1,int &h)
{   //查找值为 x 的结点所在的层次
   if(!T)  h1=0;
   else
   {
     if (T->data==x)  h=h1;
     Level(T->lchild,x,h1+1,h);          //在左子树中查找
     Level(T->rchild,x,h1+1,h);          //在右子树中查找
     }
}

int NodeLevel(BiTree bt,TElemType x)
{    //值为 x 的结点所有的层次
   int h;
   Level(bt,x,1,h);
   return(h);
}
```

6．根据祖先结点的定义，如果结点 r 是 p 的祖先，那么 p 必定是在 r 的左子树或右子

树中的结点。因此，定义函数 InTree(BT, p)检查 p 是否为子树 BT 中的结点，并在这个过程中输出 p 的祖先。基本思想为：

⑴若 BT= = NULL，则返回 0，表明 p 不在子树 BT 中;

⑵若 BT = = p，则返回 1，表明 p 包含在子树 BT 中;

⑶若 InTree(BT->lchild , p) = =1 或者 InTree(BT->rchild, p) = =1，则 BT 为 p 的祖先，输出 BT 并返回 1;

⑷否则，返回 0。

根据这一思路，可编写如下递归算法：

```
int InTree(BiTree BT,BiTree p)
{
 if(!BT) return 0;
 if(BT==p) return 1;
 if(InTree(BT->lchild,p)||InTree(BT->rchild,p))
 {
     cout<<BT->data<<' ';
     return 1;
 }
    return 0;
}
```

进一步讨论：⑴在上述算法中，通过检查 p 是否为子树 BT 中的结点来输出 p 的所有祖先。实际上检查 p 是否为子树 BT 中结点的过程等价于二叉树的先序遍历，即首先判断根结点是否为 p，若不是，再去检查 p 是否为 BT 的左子树或右子树中的结点。但本算法中，p 的祖先结点的输出顺序为：p 的父结点、祖父结点……最后输出根结点。

⑵可以从另一个角度来求解这个问题。根据祖先结点的定义，如果结点 r 是 p 的祖先，那么必须满足下列条件之一：

①p 为 r 的左孩子或右孩子；

②r 的左孩子或右孩子为 p 的祖先。

根据这个思路可编写递归算法如下：

```
int Ancestor(BiTree BT,BiTree p)
{
 if (!BT)    return 0;
 if(BT->lchild==p||BT->rchild==p||Ancestor(BT->lchild,p)||
      Ancestor(BT->rchild,p))
 {
     cout<<BT->data<<' ';
     return 1;
 }
    return 0;
}
```

⑶本题也可采用非递归后序遍历二叉树来实现，当后序遍历访问到 p 时，栈中所有结点均为 p 的祖先结点。

7．利用后序遍历二叉树的非递归遍历算法。为了叙述方便，不妨假定 p 比 q 先进栈，当 p 进栈时，栈中的结点均为 p 的祖先，此时可用变量 pos 保存 p 在栈中的位置，然后继续

进行后序遍历，直到 q 进栈。在这个过程中，如果 pos 位置上的元素需要出栈，那么将 pos 指向栈中前一个元素的位置，以保证 pos 位置上的元素一定是当前栈中距离 p 最近的祖先结点，那么当 q 进栈时，同样栈中结点均为 q 的祖先。因此，pos 位置上的元素就是为包含 p 和 q 的最小子树的根结点。

```
# define MaxLen 20        //定义栈的最大空间，大于树的最大深度
BiTree MinTree(BiTree BT,BiTree p,BiTree q)
{
    BiTree stack[MaxLen+1],r;
    int pos=0,top=0;
    r=BT;
    do{
       while(r)
       {
         stack[++top]=r;
         if(r==p||r==q)
           if(pos==0)  pos=top;          //第一个结点进栈
           else return (stack[pos]);     //第二个结点进栈
         r=r->lchild;
       }
       while(top>0 && stack[top]->rchild==r)
       {
         if(pos==top) pos--;             //pos 位置上的元素需要出栈
         r=stack[top--];                 //栈顶结点出栈
       }
       if(top>0) r=stack[top]->rchild; //开始遍历右子树
    }while(top>0);
    return  NULL;
}
```

8．可以借鉴对中序线索二叉树进行遍历的思想，即先找到中序序列的第一个结点，然后依次找结点的后继。在二叉树的中序序列中，第一个结点为二叉树中最左边的结点，即从根结点开始，沿结点的左指针向下查找，直到左指针为空。而对于二叉树中的任何一个结点 p，可分下列两种情况来找 p 的后继：

⑴若 p 的右子树不为空，则 p 的后继为其右子树的中序序列的第一个结点；

⑵否则，沿 p 的双亲指针向上查找 p 的祖先，直到找到第一个在左子树中包含 p 结点的祖先，这个祖先结点则为 p 的后继。

根据这个思想，可编写算法如下：

```
void InOrder(PBiTree BT,void Visit(TElemType))
{
    PBiTree p,q;
    if(!BT) return;
    p=BT;
    while(p->lchild)                //找中序序列的第一个结点
        p=p->lchild;
```

```
    while (p)
    {
        Visit(p->data);             //访问结点
        if(p->rchild)               //在 p 的右子树中找 p 的后继
        {
            p=p->rchild;
            while(p->lchild)
            p=p->lchild;
        }
        else                        //在 p 的祖先中找 p 的后继
        {
          q=p;
          p=q->parent;
          while(p&&p->lchild!=q)
          {
             q=p;
             p=q->parent;
          }
        }
    }
}
```

进一步讨论：（1）在这种存储结构中，由于每个结点增加了一个指向双亲结点的指针，利用这个指针和左、右孩子指针可以找到一个结点的后继。本算法正是利用这个特点来实现中序遍历的。因此，如果在二叉树中的一种存储结构中，如果能够由一个结点找到其左、右孩子及双亲结点（如二叉树的顺序存储结构），那么就可以不用栈实现中序遍历。

按照这个算法的思路，也可以在这种存储结构下编写不设栈进行前序和后序遍历的非递归算法。

在二叉树的前序序列中，第一个结点为根结点，而一个结点 p 的后继可按下列方法找到：

①若 p 的左子树不为空，则后继为其左孩子；

②若 p 的左子树为空但右子树不为空，则后继为其右孩子；

③否则，沿 p 的双亲指针一直向上找到这样一个祖先 f，使得 f 的左子树中包含 p，且 f 的右子树不为空。则 f 的左孩子为 p 的后继。

算法描述如下：

```
void PreOrder(PBiTree BT,void Visit(TElemType))
{
PBiTree p,q;
if(!BT)  return;
p=BT;
while(p)
{
    Visit(p->data);
    if(p->lchild) p=p->lchild;
    else if(p->rchild)  p=p->rchild;
    else {
```

```
            q=p;
            p=q->parent;
            while(p&&(p->lchild!=q||!p->rchild))
            {
               q=p;
               p=q->parent;
            }
            if(p)
            p=p->rchild;
        }
      }
    }
```

在二叉树的后序序列中，第一个结点为左边的叶子结点，即从根结点开始，若结点的左子树不空，则沿左指针向下查找，否则沿右指针向下查找，直到叶子结点。而一个结点 p 的后继可按下列方法找到：

①若 p 是其双亲的右孩子或是其双亲的左孩子且其双亲没有右孩子，则其后继即为双亲结点；

②若 p 是其双亲的左孩子且其双亲的右子树不空，则其后继为双亲右子树上按后序遍历的第一个结点；

算法描述如下：

```
 void PostOrder(PBiTree BT,void Visit(TElemType))
 {
     PBiTree p,q;
     if(!BT)  return;
     p=BT;
     while(p->lchild||p->rchild)         //找后序序列的第一个结点
       if(p->lchild) p=p->lchild;
       else if(p->rchild) p=p->rchild;
     while(p)
     {
         Visit(p->data);
         q=p;
         p=q->parent;
         if(p&&p->lchild==q&&p->rchild)
         {
           p=p->rchild;
           while(p->lchild||p->rchild)
             if(p->lchild) p=p->lchild;
             else if(p->rchild)  p=p->rchild;
         }
     }
 }
```

# 习题七

## 一、选择题

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

## 二、填空题

1. 邻接表
2. 有向图的顶点偶是有序的，无向图的顶点偶是无序的
3. 稠密　稀疏
4. 有向
5. 1
6. 求矩阵第 i 列非零元素之和
7. 将矩阵第 i 行全部置为零
8. n-1
9. n(n-1)/2　n-1　n(n-1)　n　n(n-1)-2　n-1
10. 对每个顶点查找其邻接点的过程
    O(n+e)，其中 n 为图中顶点数 e 是图中边的条数（以邻接表为例）
    遍历图的顺序不同
    DFS 采用栈存储访问过的结点，BFS 采用队列存储访问过的结点
11. 唯一
12. 2e+n　2e　n
13. 生成树
14. 2
15. 邻接矩阵　邻接表

## 三、简述题

1. 解答：

无向图的存储结构有邻接矩阵、邻接表和邻接多重表，有向图的存储结构有邻接矩阵、邻接表和十字邻接表，它们都可用边集数组。

⑴邻接矩阵：可判定图中任意两个顶点之间是否有边（或弧）相连，并容易求得各个顶点的度；此外，对于图的遍历也是可行的。

⑵邻接表：容易找到任一顶点的第一个邻接点和下一个邻接点；但要判断任意两个顶点之间是否有边或弧相连，则需搜索第 i 个及第 j 个链表，这不如邻接矩阵方便；此外，对于图的遍历和有向图的拓扑排序也是可行的。

⑶十字邻接表：容易找到以某顶点为头或为尾的弧，因此容易求得顶点的入度和出度；在有向图的应用中，十字邻接表是很有用的工具。

⑷邻接多重表：是无向图的一种非常有效的存储结构，在其中容易求得顶点和边的各种信息。

⑸边集数组：边集数组适合那些对边依次进行处理的运算，不适合对顶点的运算和对任一条边的运算。

2. 解答：

一个带权无向图的最小生成树不一定是唯一的。从 krushal 算法构造最小生成树的过程

可以看出，当从图中选择当前权值最小的边时，如果存在多条这样的边，并且这些边与已经选取的边构成回路，此时这些边就不可能同时出现在一棵最小生成树中，对这些边的不同选择结果可能会产生不同的最小生成树。

3．解答：

图 G 如图 7.1 所示：

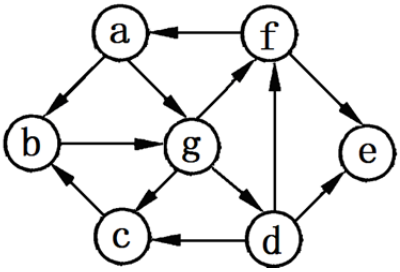



图 7.1　图 G

| | 顶点 |
|---|---|
| 0 | a |
| 1 | b |
| 2 | c |
| 3 | d |
| 4 | e |
| 5 | f |
| 6 | g |

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

图 7.2　图 G 的邻接矩阵

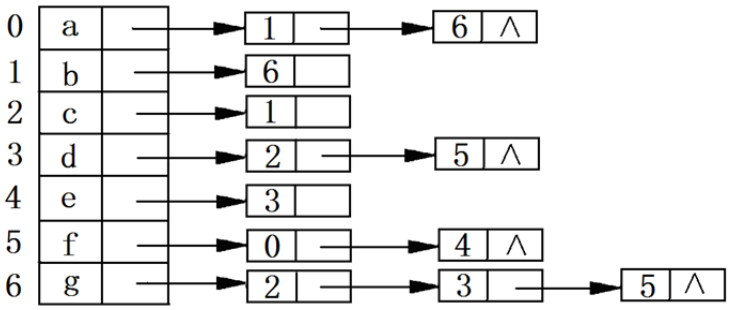



图 7.3　图 G 的邻接表

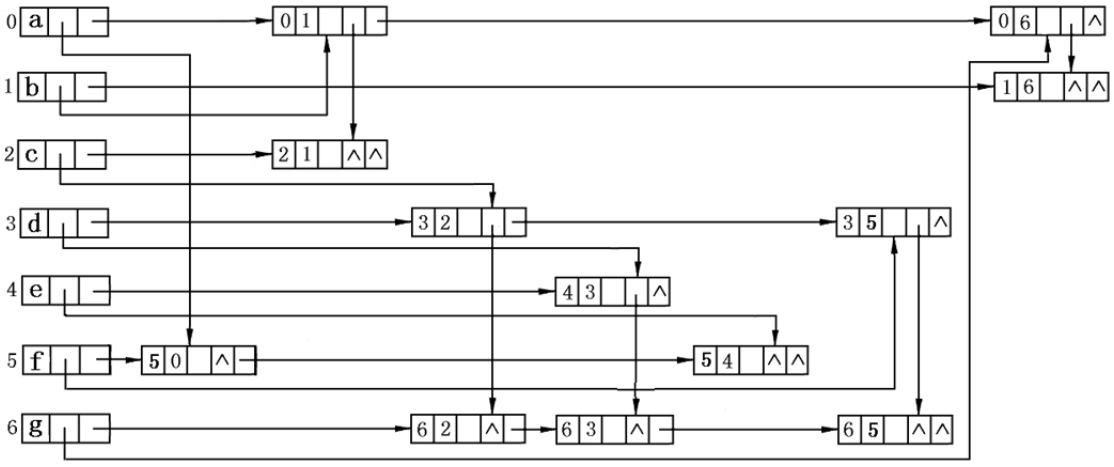



图 7.4　图 G 的十字邻接表

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 起点 | 0 | 0 | 1 | 2 | 3 | 3 | 4 | 5 | 5 | 6 | 6 | 6 |
| 终点 | 1 | 6 | 6 | 1 | 2 | 5 | 3 | 0 | 4 | 2 | 3 | 5 |

图 7.5　图 G 的边集数组

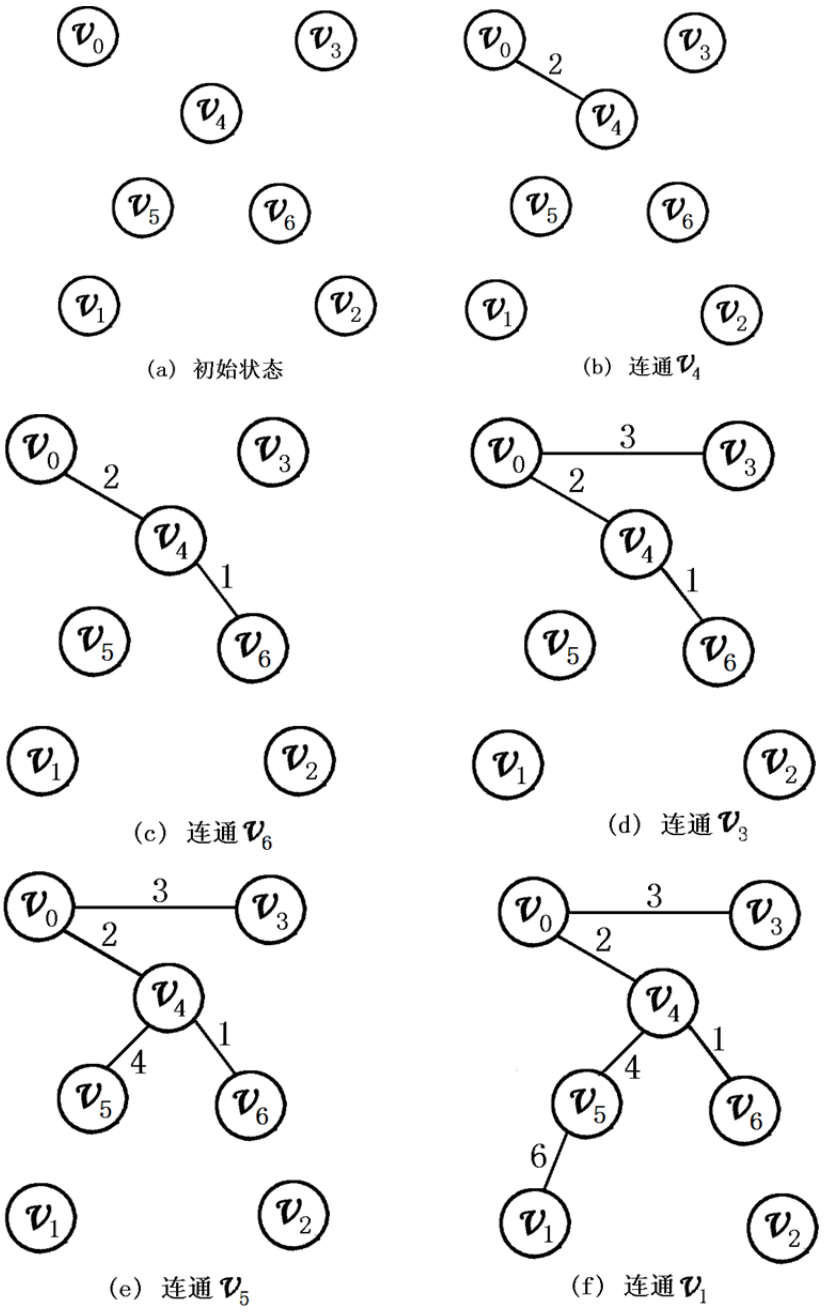
4．解答：

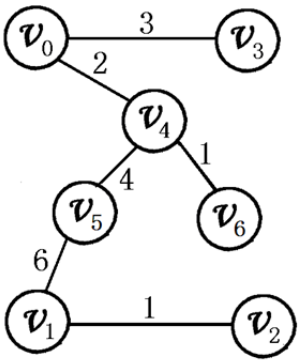
广度优先遍历的序列为：$v_0$，$v_1$，$v_2$，$v_5$，$v_3$，$v_4$，$v_7$，$v_6$；

深度优先遍历的序列为：$v_0$，$v_1$，$v_5$，$v_3$，$v_4$，$v_6$，$v_7$，$v_2$。

5．解答：

⑴利用普里姆算法从顶点 $v_0$开始构造最小生成树的过程如图 7.6 所示：

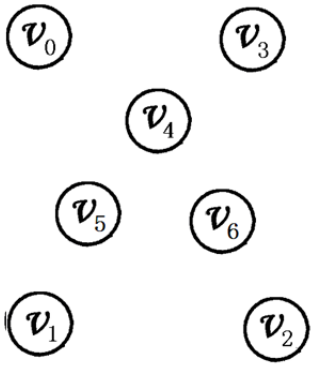




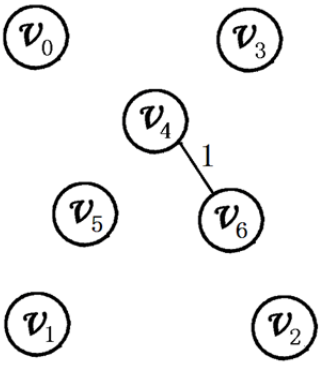

(g) 连通$v_2$

图 7.6　用 prime 算法构造最小生成树的过程

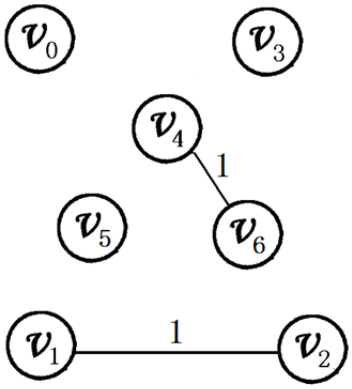
⑵利用克鲁斯卡尔算法从顶点 $v_0$开始构造最小生成树的过程如图 7.7 所示：



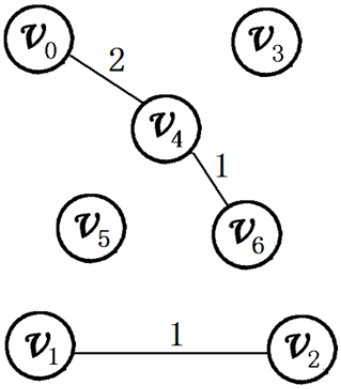

(a) 初始状态



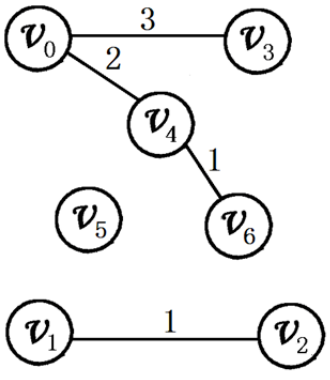

(b) 增加第1条边



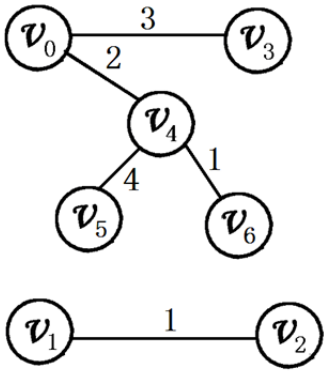

(c) 增加 2 第2条边




(d) 增加第3条边




(e) 增加第4条边




(f) 增加第5条边

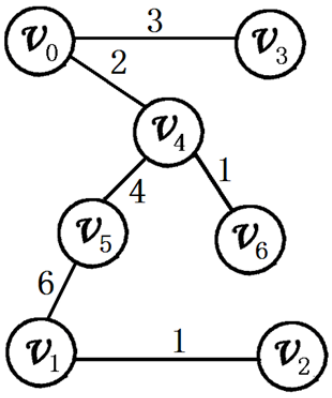



图 7.7　用 kruskal 算法构造最小生成树的过程

6．解答：

对于这个图，在狄克斯特拉算法执行过程中，每求得一条从 $v_0$ 到某个顶点的最短路径后，当前从 $v_0$ 到其余各顶点的最短路径及其长度的变化情况如下表所示：

| | | $v_1$ | $v_2$ | $V_3$ | $V_4$ | $V_5$ | 求得的最短路径 |
|---|---|---|---|---|---|---|---|
| 初态 | 最短路径 | 10<br>$(v_0, v_1)$ | 12<br>$(v_0, v_2)$ | ∞<br>$(v_0, v_3)$ | ∞<br>$(v_0, v_4)$ | ∞<br>$(v_0, v_5)$ | 10<br>$(v_0, v_1)$ |
| 1 | 最短路径 | | 12<br>$(v_0, v_2)$ | 26<br>$(v_0, v_1, v_3)$ | 35<br>$(v_0, v_1, v_4)$ | ∞<br>$(v_0, v_5)$ | 12<br>$(v_0, v_2)$ |
| 2 | 最短路径 | | | 24<br>$(v_0, v_2, v_3)$ | 35<br>$(v_0, v_1, v_4)$ | 20<br>$(v_0, v_2, v_5)$ | 20<br>$(v_0, v_2, v_5)$ |
| 3 | 最短路径 | | | 22<br>$(v_0, v_2, v_5, v_3)$ | 30<br>$(v_0, v_2, v_5, v_4)$ | | 22<br>$(v_0, v_2, v_5, v_3)$ |
| 4 | 最短路径 | | | | 29<br>$(v_0, v_2, v_5, v_3, v_4)$ | | 29<br>$(v_0, v_2, v_5, v_3, v_4)$ |

因此，利用狄克斯特拉算法求得从 $v_0$ 到其余各顶点的最终的最短路径及其长度分别为：

$v_0$ 到 $v_1$：最短路径为$(v_0, v_1)$，长度为 10；

$v_0$ 到 $v_2$：最短路径为$(v_0, v_2)$，长度为 12；

$v_0$ 到 $v_3$：最短路径为$(v_0, v_2, v_5, v_3)$，长度为 22；

$v_0$ 到 $v_4$：最短路径为$(v_0, v_2, v_5, v_3, v_4)$，长度为 29；

$v_0$ 到 $v_5$：最短路径为$(v_0, v_2, v_5)$，长度为 20。

7．解答：

⑴利用拓扑排序算法得到的拓扑序列为：$v_0$，$v_7$，$v_8$，$v_1$，$v_4$，$v_9$，$v_5$，$v_6$，$v_2$，$v_3$

⑵如果在拓扑排序算法中，用一个队列来替代栈，那么得到的拓扑序列为：

$v_0$，$v_1$，$v_7$，$v_2$，$v_4$，$v_8$，$v_5$，$v_9$，$v_3$，$v_6$

⑶从顶点 $v_0$ 开始，利用 DFS 遍历该图得到的逆拓扑序列为：

$v_3$，$v_2$，$v_6$，$v_5$，$v_9$，$v_4$，$v_1$，$v_8$，$v_7$，$v_0$.

**分析：**

一个有向无环图的拓扑排序序列可能有多个，但拓扑排序算法输出的拓扑序列只有一条，

至于输出哪一条拓扑序列取决于存储结构中各顶点及邻接点的排列顺序。在本题中明确地规定了顶点及其邻接点的顺序，因此拓扑排序的结果是唯一确定的。通常情况下，拓扑排序算法中利用一个栈来暂存入度为 0 的顶点，因此，当同时有多个入度为 0 的顶点时，拓扑排序算法总是先输出当前入度为 0 的顶点中的最后一个。如果用一个队列代替栈来存放入度为 0 的顶点，那么算法总是先输出当前入度为 0 的顶点中的第一个顶点，因此会得到一个不同的拓扑序列。

8．解答：

⑴每个事件的最早发生时间和最晚发生时间为：

| 事件 | $v_0$ | $v_1$ | $v_2$ | $v_3$ | $v_4$ | $v_5$ | $v_6$ | $v_7$ | $v_8$ | $v_9$ |
|---|---|---|---|---|---|---|---|---|---|---|
| ve | 0 | 5 | 7 | 13 | 17 | 18 | 23 | 22 | 26 | 31 |
| vl | 0 | 10 | 7 | 13 | 17 | 20 | 23 | 22 | 28 | 31 |

因此，完成整个工程最少需用的时间是 31。

⑵每个活动的最早开始时间和最迟开始时间为：

| 活动 | $a_1$ | $a_2$ | $a_3$ | $a_4$ | $a_5$ | $a_6$ | $a_7$ | $a_8$ | $a_9$ | $a_{10}$ | $a_{11}$ | $a_{12}$ | $a_{13}$ | $a_{14}$ | $a_{15}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| e | 0 | 0 | 5 | 7 | 7 | 13 | 13 | 13 | 17 | 17 | 18 | 23 | 23 | 22 | 26 |
| l | 5 | 0 | 10 | 7 | 14 | 13 | 15 | 19 | 17 | 17 | 20 | 26 | 23 | 22 | 28 |

根据各个活动的最早开始时间和最迟开始时间，可以看出 $a_2, a_4, a_6, a_9, a_{20}, a_{23}, a_{14}$ 为关键活动，因此关键路径为（$v_0$，$v_2$，$v_3$，$v_4$，$v_6$，$v_9$）和（$v_0$，$v_2$，$v_3$，$v_4$，$v_7$，$v_9$）

9．解答：

由图 7.35 无向图表示的通讯网，可得到两个最小生成树如图 7.8 的(a)和(b)所示，这两棵生成树即为所有可能的选择。

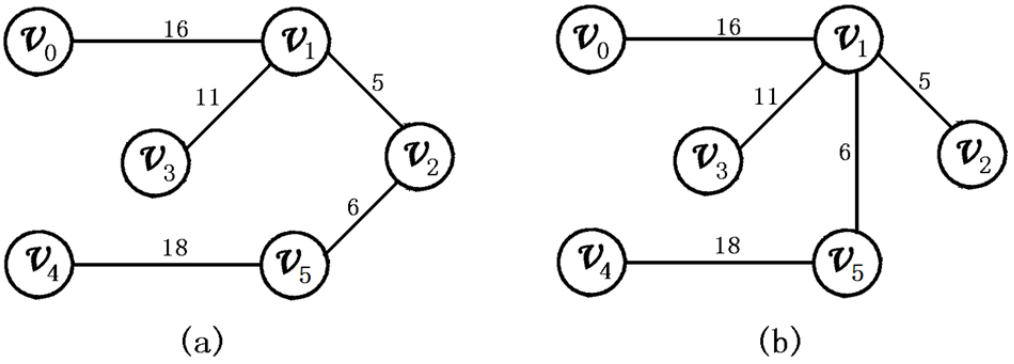



图 7.8　两棵最小生成树

10．解答：

首先根据邻接矩阵的三元组表画出带权无向图如图 7.9 所示：

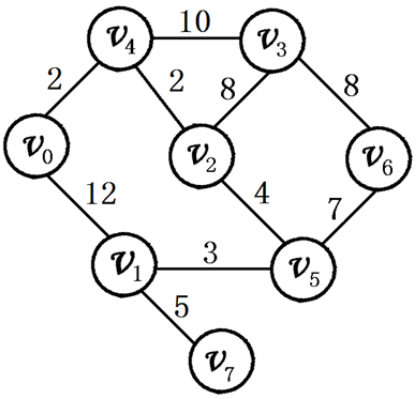



图 7.9　带权无向图

⑴图 7.31 的邻接表如图 7.10 所示：

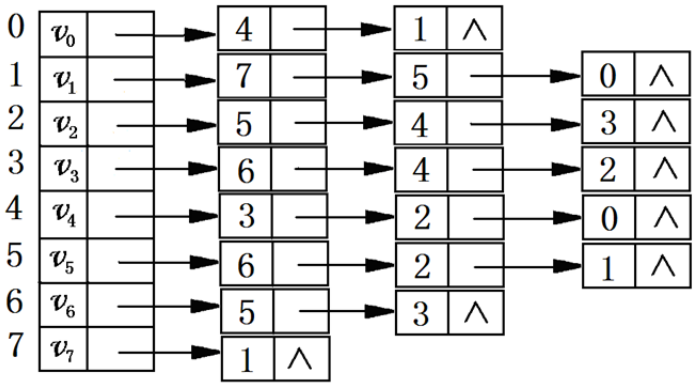



图 7.10　邻接表

⑵由于存在权值相同的边，则最小生成树可能不止一个，此题可能的最小生成树如图 7.11 的(a)和(b)所示：

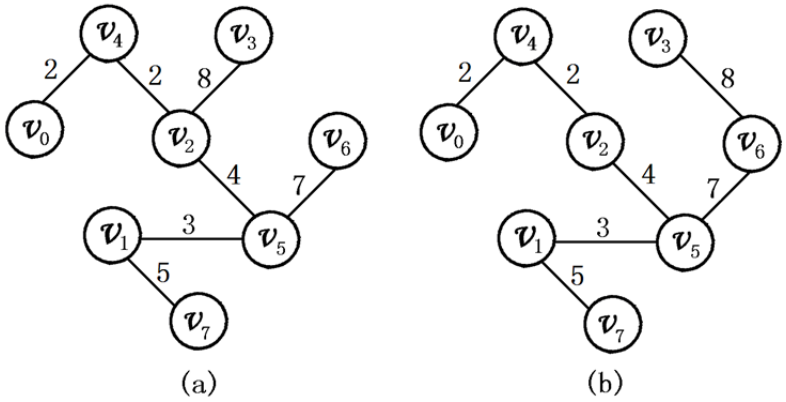



图 7.11　可能的最小生成树

⑶从 $v_0$ 到 $v_1$ 的最短路径为：$v_0$，$v_4$，$v_2$，$v_5$，$v_1$。

**四、算法设计题**

**说明：**以下解答中有关邻接矩阵的习题要包含头文件：MGraph.h，有关邻接表的习题要包含头文件:ALGraph.h,在头文件 ALGraph.h 中定义了全局数组 visited[MAX_VERTEX_NUM]。

1．算法描述如下：

```
void NRDFS(ALGraph G,int v,void Visit(VertexType))
```

```
{
  int j,top;
  ArcNode *p,*S[MAX_VERTEX_NUM];
  top=0;
     visited[v]=true;
     Visit(G.vertices[v].data);                    // 访问第 v 个顶点
     S[top++]=G.vertices[v].firstarc;
     while(top)
     {
        p=S[--top];
      if(p)
      {
          S[top++]=p->next;
          j=p->adjvex;
          if(!visited[j])
          {
              visited[j]=true;
             Visit(G.vertices[j].data);
             S[top++]=G.vertices[j].firstarc;
          }
      }
  }
}
```

2. 本题的算法思想是：从给定顶点 v 出发进行深度优先遍历，在遍历过程中判别当前访问的结点是否为 v，若是，则找到一条回路；否则继续遍历。为此，设一个顺序栈 cycle 记录成回路的顶点序列，把访问顶点的操作改为将当前访问的顶点入 stack 栈；相应地，若从某一顶点出以遍历完再回溯，则做退栈操作，同时要求找到的回路的路径大于 2。另外，还需要需设置一个 found，其初值为 false，当找到回路后为 true，其算法描述如下：

```
void DFScycle(ALGraph G,int v)
{
   int i,j,top=0,w=v;
   bool visited[MAX_VERTEX_NUM],found=false;
   int cycle[MAX_VERTEX_NUM];
   ArcNode  *p,*stack[MAX_VERTEX_NUM]; // 边结点指针
   for(i=0;i<G.vexnum;i++)
       visited[i]=false;
   i=0;
   cycle[i]=w;                          // 从 w 点开始搜索
   visited[w]=true;
   p=G.vertices[w].firstarc;            // p 指向顶点 v 的第 1 个邻接顶点
   while((p||top>0)&&!found)
   {
      while(p&&!found)
```

```
            if(p->adjvex==v&&i>=2)  found=true;// 找到路径大于 2 的回路
         else
             if(visited[p->adjvex]) p=p->next; // 找下一个邻接点
            else
            {
               w=p->adjvex;                    // 记下路径，继续搜索
               visited[w]=true;
               i++;
               cycle[i]=w;
               top++;
               stack[top]=p;
               p=G.vertices[w].firstarc;
            }
         if(!found && top>0)            // 沿原路径退回另选路径继续搜索
         {
            p=stack[top];
            top--;
            p=p->next;
            i--;
         }
      }
      if(found)
      {
         for(j=0;j<i;j++)
         cout<<cycle[j]<<"  ";
         cout<<v<<endl;
      }
      else cout<<"没有通过给定点 v 的回路"<<endl;
   }
```

3. 本题的算法思想是：在广度优先遍历算法中，使用了一个队列，用来存放已访问的顶点。如果以数组 vexno[n]替代队列，开始时设置指针 front 和 rear 为 0，那么一个顶点 k 入队操作就为 vexno[rear++]=k，顶点出队操作就为 k=vexno[front++]，判断队列非空的条件为 front<rear。这样就可依次将访问过的顶点序号存放在数组 vexno 中。其算法描述如下：

```
   void NBFSTrave(ALGraph G,int vexno[],int n)
   {
      int front,rear,k;
      ArcNode  *p;                  // 边结点指针
      front=rear=0;
      for(k=0;k<G.vexnum;k++)
           visited[k]=false;
      for(k=0;k<G.vexnum;k++)
      {
```

```
        if(!visited[k])
        {
           visited[k]=true;
           vexno[rear++]=k;
        }
        while(front<rear)
        {
           for(p=G.vertices[vexno[front++]].firstarc;p;p=p->next)
             if(!visited[p->adjvex])
             {
                visited[p->adjvex]=true;
                vexno[rear++]=p->adjvex;
             }
        }
     }
     return;
  }
```

4. 本题的算法思想是：可以用遍历方式判定图 G 是否连通：若从某顶点（假设为顶点 v0）出发进行深度（或广度）优先遍历能访问到所有顶点，则是连通的，否则不是连通的。采用深度优先遍的算法描述如下：

```
  bool Connect(ALGraph G)
  {
     int i;
     for(i=0;i<G.vexnum;i++)
     visited[i]=false;
     DFSTraverse_AL(G,0,Visit);     // 深度优先遍历
     for(i=0;i<G.vexnum;i++)
       if(!visited[i])  return false;
     return true;
  }
```

5. 本题的算法思想是：先置全局数组 visited[n]为 false，然后利用图遍历算法从顶点 i 开始进行某种遍历，遍历之后，若 visited[j]为 true，则说明顶点 i 与顶点 j 之间存在路径；否则说明它们之间不存在路径。

基于 DFS 遍历的算法描述如下：

```
  bool ISDFSTrave(ALGraph G,int i,int j)
  {
     int k;
     for(k=0;k<G.vexnum;k++)
        visited[k]=false;
     DFSTraverse_AL(G,0,Visit);     // 深度优先遍历
     if(visited[j])  return true;
     else  return false;
```

```
}
```

基于 BFS 遍历的算法描述如下：

```
int ISBFSTrave(ALGraph G,int i,int j)
{
   int k;
   for(k=0;k<G.vexnum;k++)
      visited[k]=false;
   BFSTraverse_AL(G,0,Visit);  // 广度优先遍历
   if(visited[j])  return true;
   else return false;
}
```

6.

**说明：**本题可以实现将任意图（有向图、有向网、无向图、无向网）的邻接矩阵转换为邻接表。如果要验证该算法的正确性，需要包含头文件：MGraph.h 和头文件：ALGraph.h，因为两个头文件中有关顶点类型 VertexType 和弧（边）信息 InfoType 的描述发生冲突，所以，进行测试时可以将其中之一的头文件中的顶点类型 VertexType 和弧(边)信息 InfoType 改变一下标识符，如将 ALGraph.h 中的 VertexType 和 InfoType 改为：VertexType1 和 InfoType1 即可。其算法描述如下：

```
void MGraph_ALGraph(ALGraph &G1,MGraph G2)
{
   int i,j;
   ArcNode  *p,*q;                          // 边结点指针
   G1.kind=G2.kind;
   G1.vexnum=G2.vexnum;
   G1.arcnum=G2.arcnum;
   for(i=0;i<G2.vexnum;++i)                 // 构造顶点向量
   {   strcpy(G1.vertices[i].data.name,G2.vexs[i].name);
       G1.vertices[i].firstarc=NULL;       // 初始化与该顶点有关的出弧(边)链表
   }

   for(i=0;i<G2.vexnum;i++)
     for(j=0;j<G2.vexnum;j++)
       if(G1.kind>=2&&j>=i) break;             // 无向图或网
       else
       {
         if(G2.arcs[i][j].adj!=0&&G2.arcs[i][j].adj!=MAX_VALUE)
         { p=(ArcNode *)malloc(sizeof(ArcNode)); // 动态生成待插边结点的信息空间
           p->info=NULL;                          // 图无弧(边)信息
           if(G2.kind%2)                          // 网
           {   p->info=(InfoType1 *)malloc(sizeof(InfoType1));
                                        // 动态生成存放弧(边)信息的空间
               p->info->weight=G2.arcs[i][j].adj; // 弧(边)的相关信息
```

```
            }
            p->adjvex=j;                        // 弧头
            p->next=G1.vertices[i].firstarc;
            G1.vertices[i].firstarc=p;
            if(G1.kind>=2)                      // 无向图或网，产生第 2 个表结点，
                                                // 并插在第 j 个元素(入弧)的表头
            {   q=(ArcNode *)malloc(sizeof(ArcNode));
                q->info=(InfoType1 *)malloc(sizeof(InfoType1));
                                                // 动态生成存放弧(边)信息的空间
                q->info=p->info;
                q->adjvex=i;                    // 弧头
                q->next=G1.vertices[j].firstarc;
                G1.vertices[j].firstarc=q;
            }
          }
        }
}
```

# 习题八

## 一、选择题

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

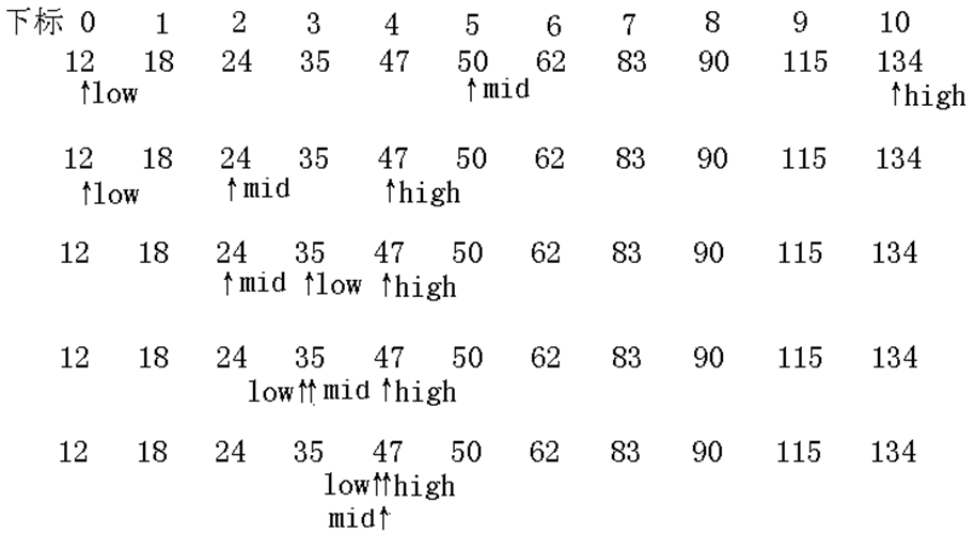
## 二、填空题

1. 哈希表查找法
2. 结点个数 n、生成过程
3. 二叉排序树
4. 直接定址
5. (n+1)/2
6. 顺指针查找、在结点的关键字中进行查找
7. 存取元素时发生冲突的可能性就越大、存取元素时发生冲突的可能性就越小

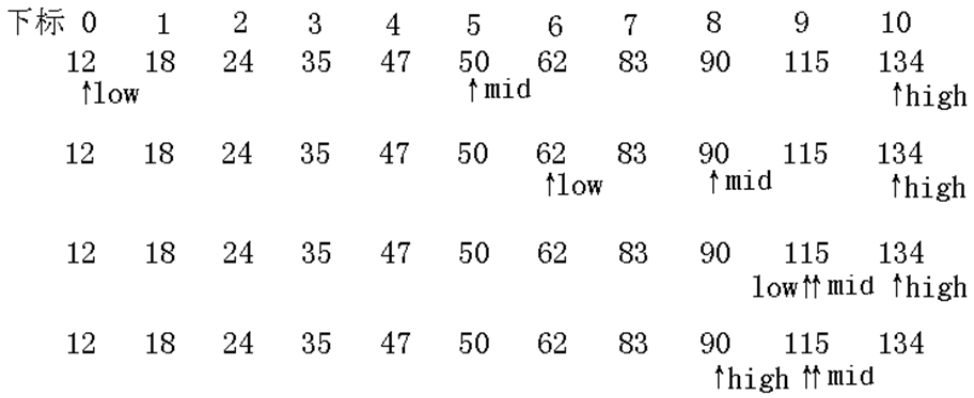
## 三、简述题

1. 解答：

二分法查找 47 和 100 的过程如图 8.1(a)和(b)所示：




(a)查找key=47的过程(五次比较后查找成功)

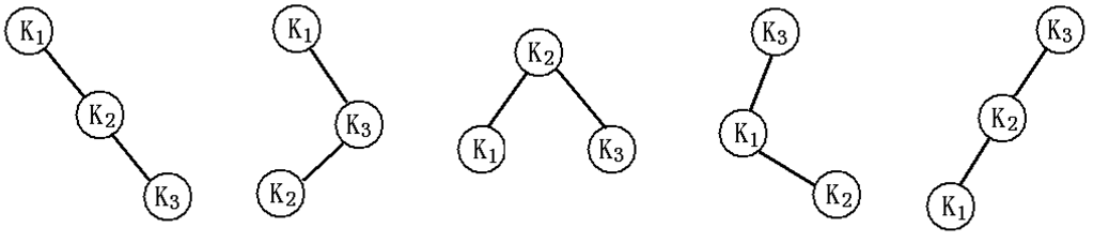



(b)查找key=100的过程(三次比较后查找失败)

图 8.1 二分查找的过程

2. 解答：

二叉排序树的建立过程如图 8.2 所示：




(a)输入$K_1$、$K_2$、$K_3$　(b)输入$K_1$、$K_3$、$K_2$　(c)输入$K_2$、$K_1$、$K_3$ 或 $K_2$、$K_3$、$K_1$　(d)输入$K_3$、$K_1$、$K_2$　(e)输入$K_3$、$K_2$、$K_1$

3. 解答：

⑴按数据元素顺序构造的二叉排序树如图 8.3 所示：

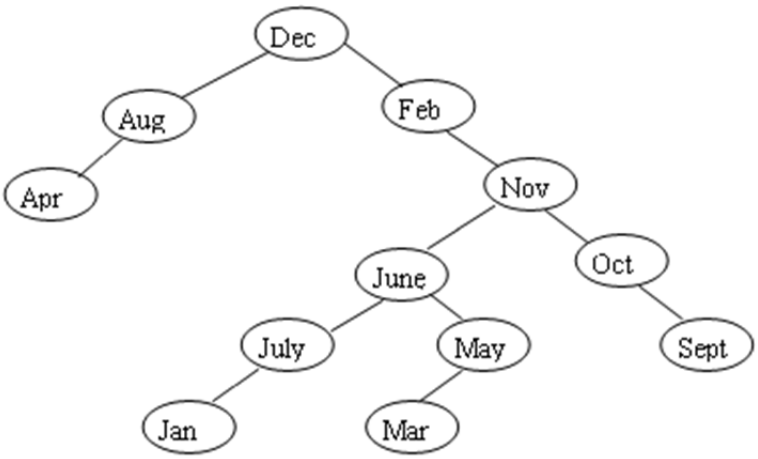



图 8.3 二叉排序树

⑵平均查找长度 ASL 为：

ASL=(1×1+2×2+3×2+4×2+5×3+6×1+7×1)/12=47/12

⑶删除结点 Nov 后的二叉排序树如图 8.4 所示：

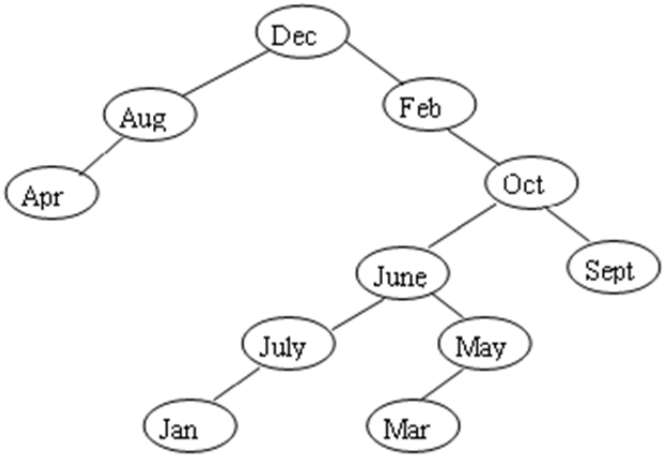



图 8.4 删除结点 Nov 后的二叉排序树

4. 解答：

进行二分查找时的二叉判定树如图 8.5 所示：

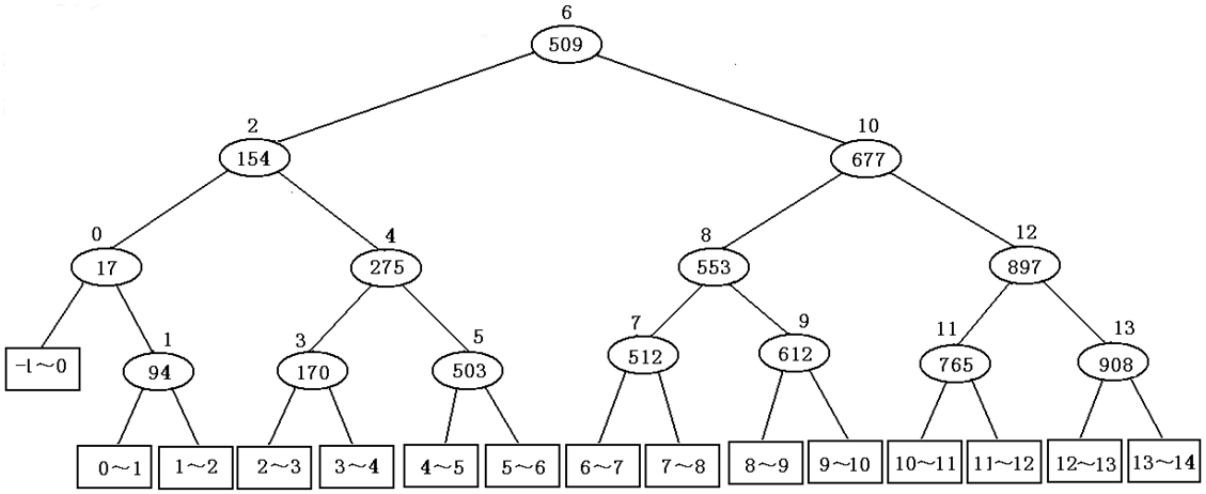


图 8.5 二叉判定树

查找成功的平均查找长度：ASL=(1×1+2×2+3×4+4×7)/14=45/14

查找不成功的平均查找长度：ASL=(3×1+4×14)/15=59/15

5．解答：

⑴调整后的 AVL 树如图 8.6 所示：

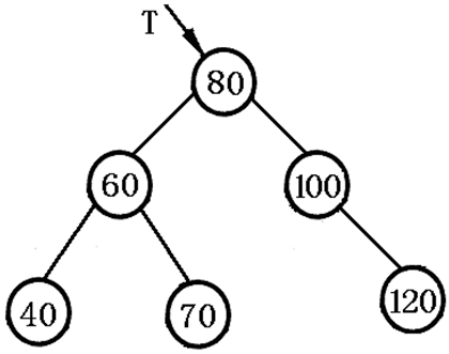



图 8.6 调整后的 AVL 树

⑵调整过程如下：

① p=T->lchild;q=p->rchild;

② T->lchild =q->rchild;

③ p->rlink=q->lchild;

④ q->rchild=T;

⑤ q->lchild=p;

⑥ T=q;

①～⑥的操作过程分别见图 8.7 的(a)～(f)所示：

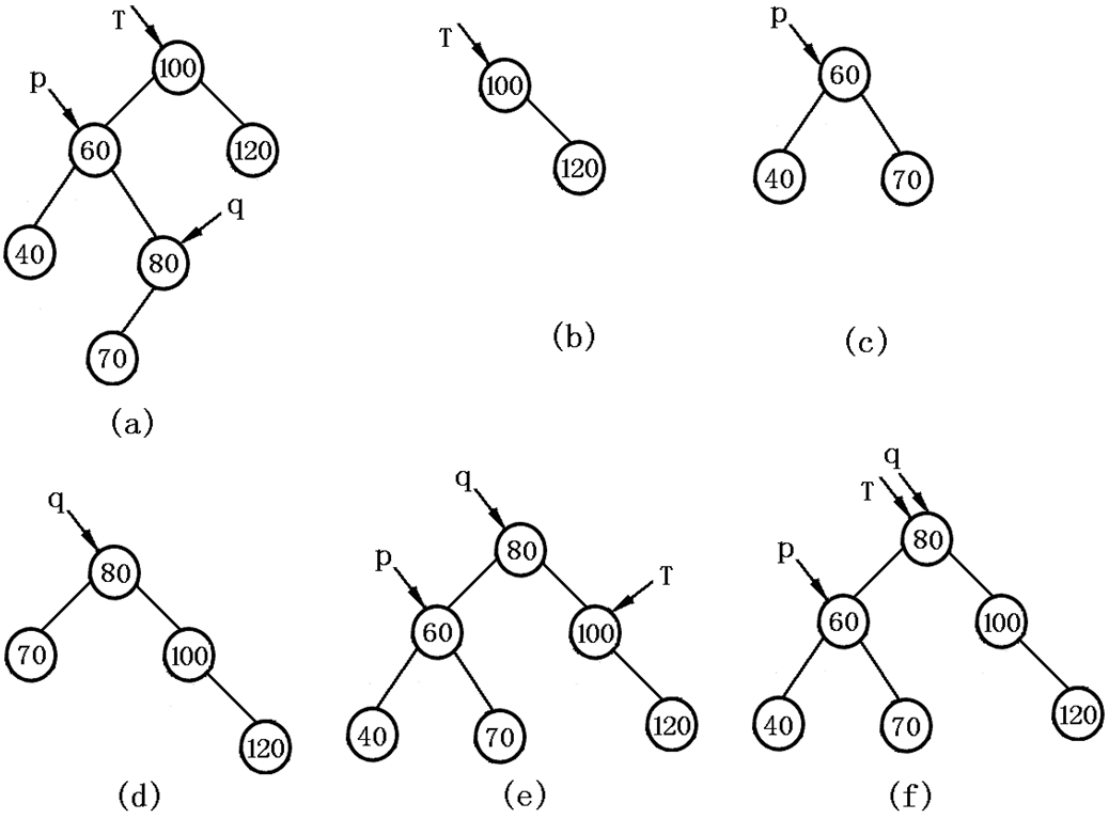


图 8.7 AVL 树的调整过程

6. 解答：

⑴建立 3 阶 B-树的过程如图所示：

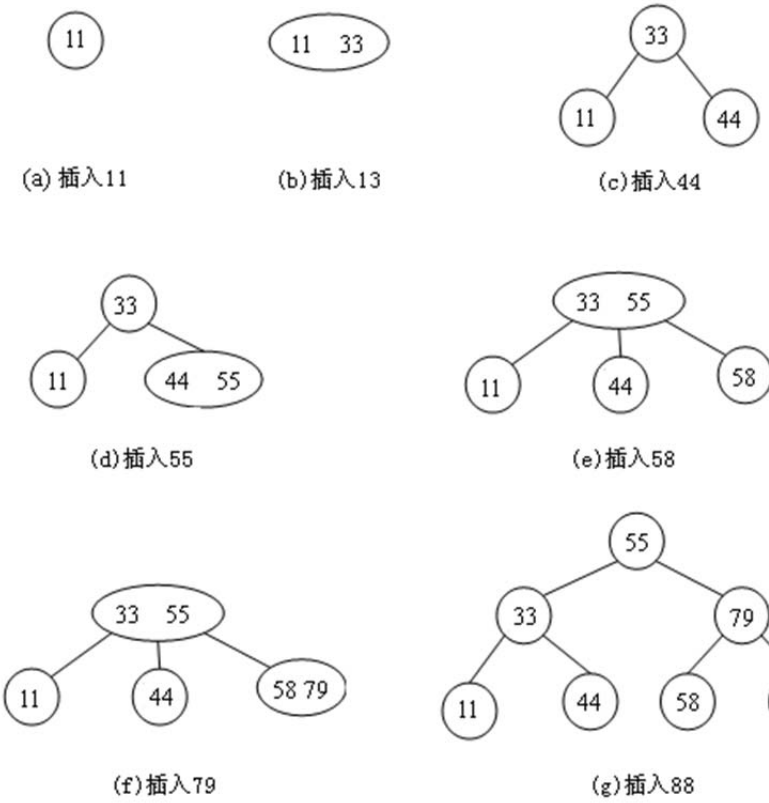



图 8.8 建立 3 阶 B_树的过程

⑵在图 8.8 中，删除关键字为 44 数据元素的过程如图 8.9(a)所示；在图 8.9(a)中再删除关键字为 79 数据元素的过程如图 8.9(b)所示：

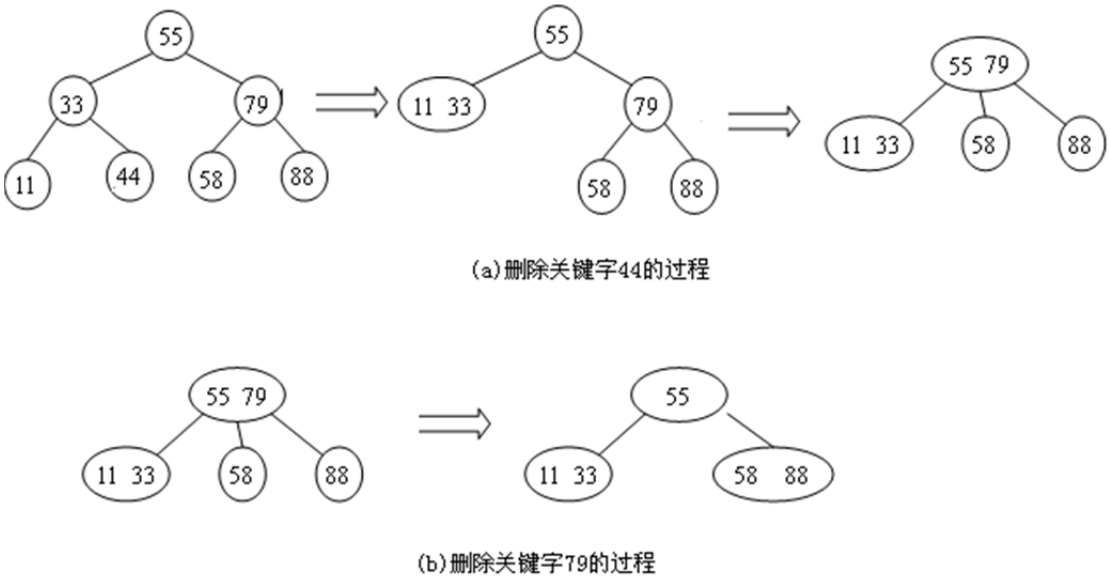



图 8.9 在 3 阶 B_树上删除元素的过程

7. 解答：（说明：因为原题中关键字序列中的关键字 67 在哈希地址为 0～3 处多次发生冲突，导致需要多次探测，这样当采用二次探测法解决冲突的时候出现了地址为负数的情况，所以，将关键字 67 改为 68）

⑴ 用线性探测法解决冲突构造的哈希表结构如图 8.10 所示：

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 22 | 33 | 46 | 13 | 01 | 68 | | | 41 | 53 | 30 |

图 8.10 线性探测法构造的哈希表

⑵用二次探测法解决冲突构造的哈希表结构如图 8.11 所示：

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 22 | 33 | 46 | 13 | | 01 | 68 | 30 | 41 | 53 | |

图 8.11 二次探测法构造的哈希表

⑶用链地址法解决冲突构造的哈希表结构如图 8.12 所示：

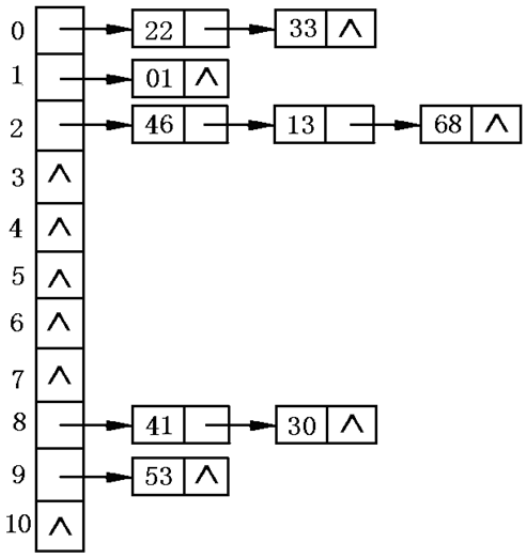



图 8.12 链地址法构造的哈希表

**四．算法设计题**

**说明：**以下解答中有关静态查找的顺序表习题采用一维数组 A 表示，其元素类型为：ElemType；而有关单链表的习题则要包含头文件：LinkList.h。

1．采用顺序存储结构的算法描述如下：

```
int SeqSearch(ElemType A[],int n,KeyType key)
{
   int i;
   ElemType temp;
   i=0;
   while(A[i].key!=key && (i<n))
     i++;
   if(i<n)
   {
```

```
        if(i!=0)
        {
          temp=A[i];
          A[i]=A[i-1];
          A[i-1]=temp;
          i--;
        }
       return i;
     }
     else
       return(0);
}
```

采用链式存储结构的算法描述如下：

```
 LNode *LinkSearch(LinkList head,KeyType key)
{
   if(head->data==key)  return head;
   else
   {
     LinkList p,q;
     ElemType temp;
     p=head->next;
     q=p->next;
     while(q&&q->data!=key)
     {
        p=q;
        q=q->next;
     }
     if(q)
     {
       temp=p->data;
       p->data=q->data;
       q->data=temp;
       q=p;
     }
     return q;
   }
}
```

2。本题的算法思想是：非递归中序遍历二叉树，当所访问的结点的 data 的值小于等于 x 时，删除此结点，删除操作后应该仍保持二叉排序树的特性。

本题的难点在于，删除结点后，如何能正确地访问到所删结点的后继结点。若一个结点 p 要被删除,那么此结点 p 的左孩子必为空(因为 p 的左孩子的关键字一定比 p 的关键字小)，若 p 的右孩子为空，则直接删除此结点，然后从栈中弹出此栈顶元素，继续进行中序遍历；若 p 的右孩子不为空,则将 p 的右孩子作为 p 的双亲的孩子(假设 p 的双亲由指针 q 指向)，

然后释放所 P 结点的空间，p 指向 q 的孩子，对 p 继续进行中序遍历。其算法描述如下：

```
void DeleteAllBST(BiTree &T,KeyType x)
{
    SqStack S;
    InitStack_Sq(S);
    BiTree p=T,q=NULL,r;
    loop:while(p||!StackEmpty_Sq(S))
    {
        if(p)
        {
            Push_Sq(S,p);
            p=p->lchild;
        }
        else
        {
          Pop_Sq(S,p);
          while(p->data.key<=x)        // 删除 p 所指结点
          {
              if(p==T)                // 待删结点是根结点
              {T=T->rchild; free(p);  p=T; goto loop;}
              else
              {
                if(!p->rchild)            // p 的右孩子为空，直接删除此结点
                {
                  free(p);
                  Pop_Sq(S,p);           // 左孩子已经被删除
                  p->lchild=NULL;
                  GetTop_Sq(S,q);       // q 指向 p 的双亲
                }
                else                     // p 的右孩子不空
                {
                   GetTop_Sq(S,q);      // q 指向 p 的双亲
                   if(p==q->lchild)
                   {  q->lchild=p->rchild; // 将 p 的右孩子作为 q 的左孩子
                      free(p);
                      p=q->lchild;
                   }
                   else
                   {  q->rchild=p->rchild; // 将 p 的右孩子作为 q 的右孩子
                      free(p);
                      p=q->rchild;
                   }
                }
              }
```

```
            }
            p=p->rchild;
        }
    }
}
```

3．本题的算法思想是：判定二叉树是否为二叉排序树同样是建立在中序遍历的框架基础下，在遍历中附设一指针 pre 指向当前访问结点的中序直接前驱，每访问一个结点便比较前驱结点 pre 和此结点是否有序，若遍历结束后各结点和其中序直接前驱均满足有序，则此二叉树为二叉排序树；否则只要有一个结点不满足，那么此树就不是二叉排序树。其算法描述如下：

```
void BiSortTree(BiTree T,BiTree &pre,bool &flag)
{  // 初始时 pre=NULL,flag=true;若结束时 flag 为 true,则此二叉树是二叉排序树
   if(T&&flag)
   {
      BiSortTree(T->lchild,pre,flag);
      if(pre==NULL)    // 访问中序序列的一第一个结点,不需要比较
      {
        flag=true;
        pre=T;
      }
      else             // 比较 T 与中序直接前驱 pre 的大小
      {
          if(pre->data.key<T->data.key)  // pre 与 T 有序
          {
           flag=true;
           pre=T;
          }
          else flag=false;               // pre 与 T 无序
       }
       BiSortTree(T->rchild,pre,flag);
   }
}
```

# 习题九

## 一、选择题

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

## 二、填空题

1．3

2．2　4　（23，38，15）

3．堆排序　　快速排序　　归并排序　　归并排序　　快速排序　　堆排序

4．希尔排序　　选择排序　　快速排序和堆排序

5．堆排序　　快速排序

6．插入排序　　选择排序

7．基数排序

8．完全二叉树　　$\lfloor n/2 \rfloor$

9．划分归并段　　归并排序

10．$\lceil n/m \rceil$

**三、简述题**

1．解答：

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

2．解答：

⑴直接插入排序各趟排序的结果如图 9.1 所示：

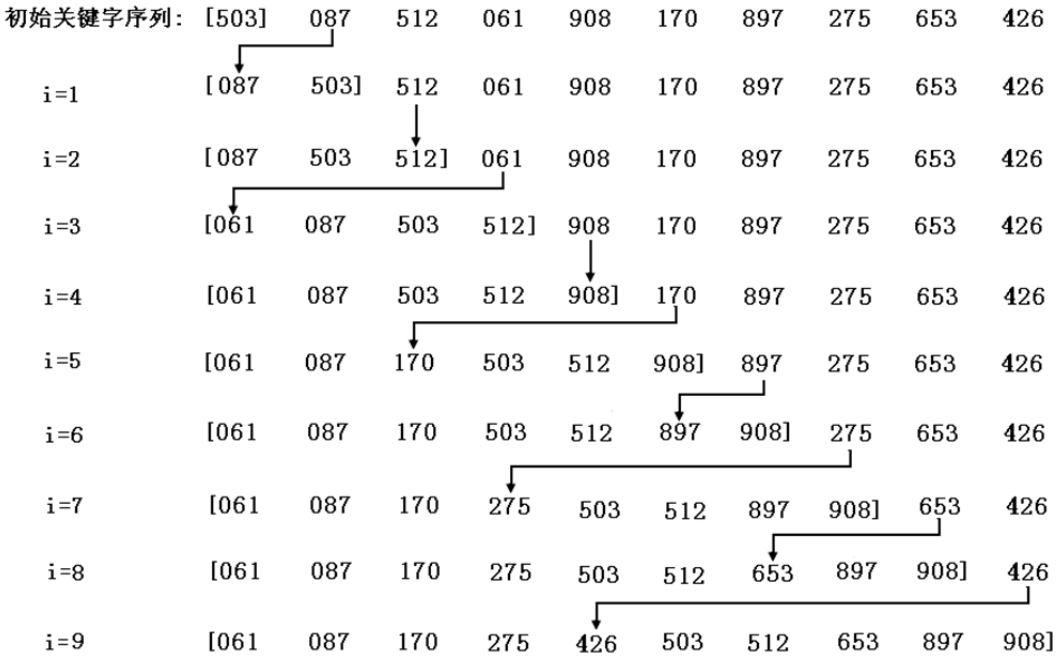



图 9.1 直接插入排序

⑵希尔排序各趟排序的结果如图 9.2 所示：

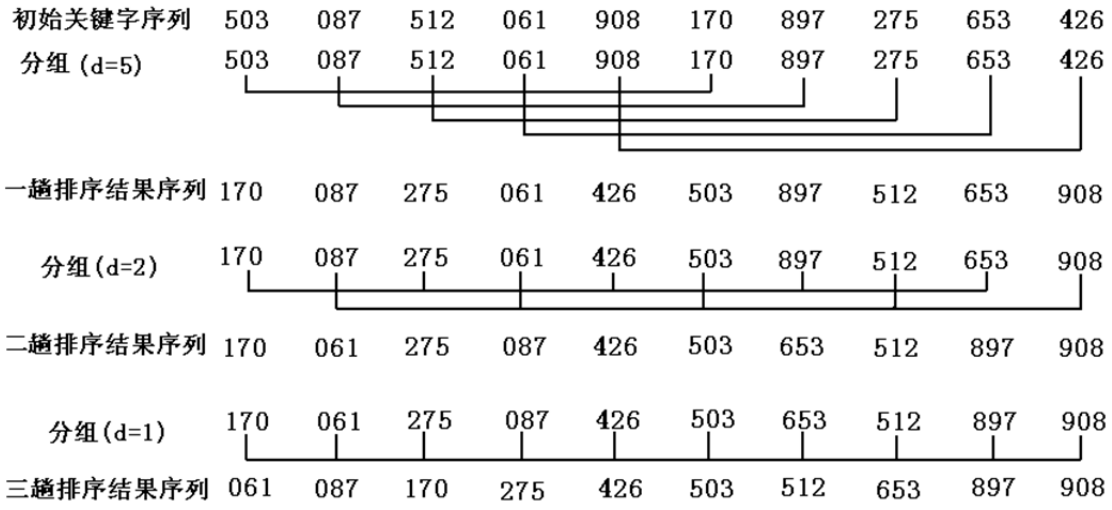



图 9.2 希尔排序

⑶直接选择排序各趟排序的结果如图 9.3 所示：

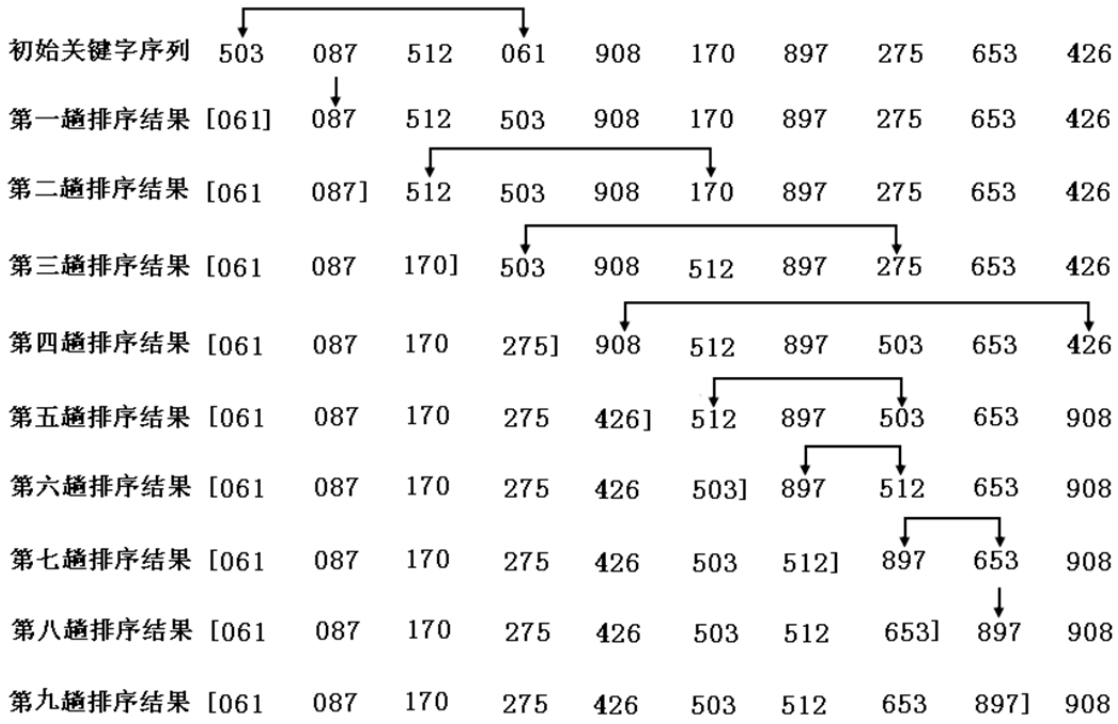
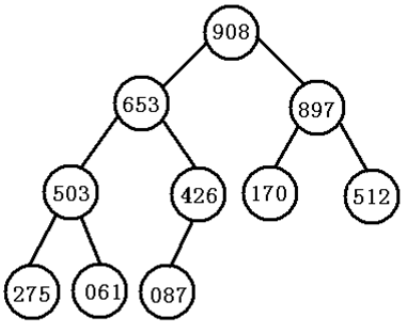
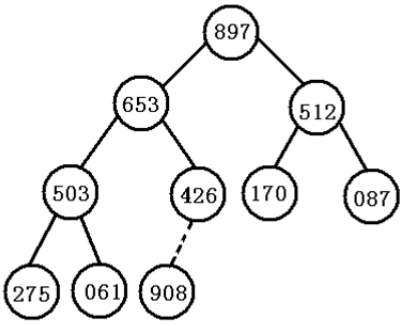



图 9.3 直接直接排序

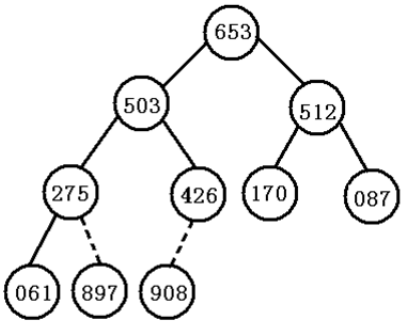
⑷堆排序各趟排序的结果如图 9.4 所示：



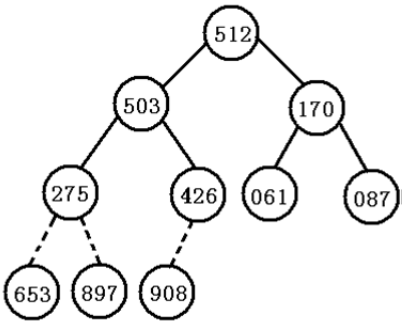

(a)初始大根堆



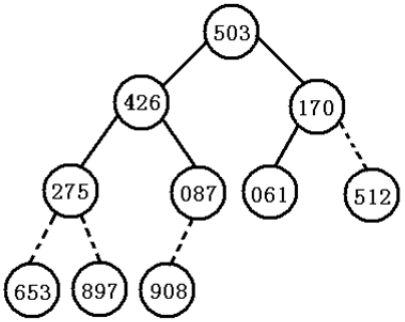

(b)908与087交换后再调整



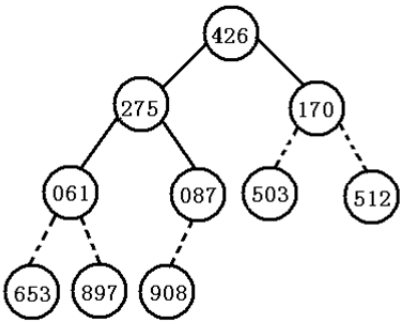

(c)897与061交换后再调整



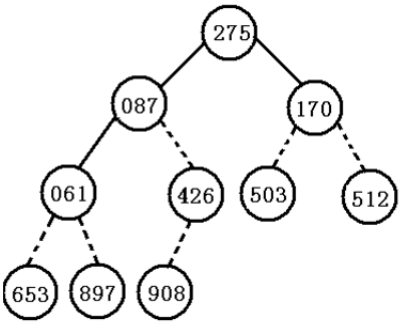

(d)653与061交换后再调整



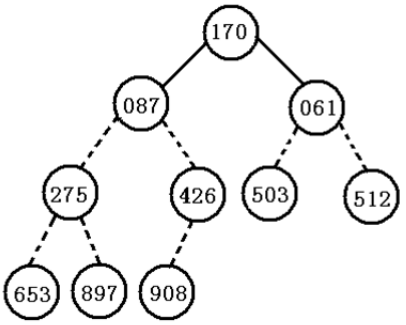

(e)512与087交换后再调整



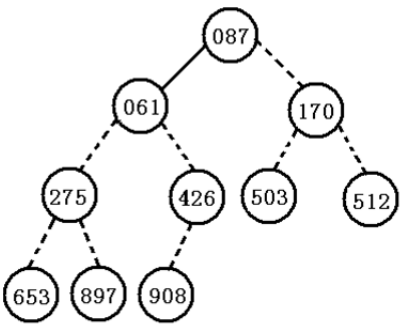

(f)503与061交换后再调整



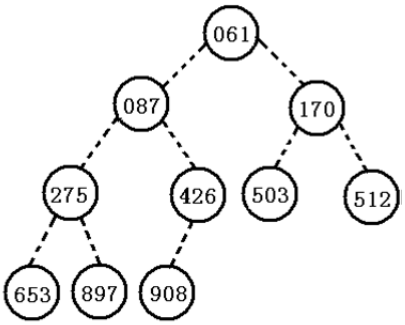

(g)426与087交换后再调整




(h)275与061交换后再调整




(i)170与061交换后再调整




(j)087与061交换后再调整

图 9.4 堆排序

⑸冒泡排序各趟排序的结果如图 9.5 所示：

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 初始关键字序列 | 503 | 087 | 512 | 061 | 908 | 170 | 897 | 275 | 653 | 426 |
| 第一趟排序结果 | 087 | 503 | 061 | 512 | 170 | 897 | 275 | 653 | 426 | [908] |
| 第二趟排序结果 | 087 | 061 | 503 | 170 | 512 | 275 | 653 | 426 | [897 | 908] |
| 第三趟排序结果 | 061 | 087 | 170 | 503 | 275 | 512 | 426 | [653 | 897 | 908] |
| 第四趟排序结果 | 061 | 087 | 170 | 275 | 503 | 426 | [512 | 653 | 897 | 908] |
| 第五趟排序结果 | 061 | 087 | 170 | 275 | 426 | [503 | 512 | 653 | 897 | 908] |
| 最后结果序列 | 061 | 087 | 170 | 275 | 426 | 503 | 512 | 653 | 897 | 908 |

图 9.5 冒泡排序

⑹快速排序各趟排序的结果如图 9.6 所示

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 初始关键字序列 | 503 | 087 | 512 | 061 | 908 | 170 | 897 | 275 | 653 | 426 |
| 第一趟划分结果 | [426 | 087 | 275 | 061 | 170] | 503 | [897 | 908 | 653 | 512] |
| 第二趟划分结果 | [170 | 087 | 275 | 061] | 426 | 503 | [897 | 908 | 653 | 512] |
| 第三趟划分结果 | [061 | 087] | 170 | [275] | 426 | 503 | [897 | 908 | 653 | 512] |
| 第四趟划分结果 | 061 | [087] | 170 | [275] | 426 | 503 | [897 | 908 | 653 | 512] |
| 第五趟划分结果 | 061 | [087] | 170 | [275] | 426 | 503 | [512 | 653] | 897 | [908] |
| 第六趟划分结果 | 061 | [087] | 170 | [275] | 426 | 503 | 512 | [653] | 897 | [908] |
| 最后结果序列 | 061 | 087 | 170 | 275 | 426 | 503 | 512 | 653 | 897 | 908 |

图 9.6 快速排序

⑺归并排序各趟排序的结果如图 9.7 所示：

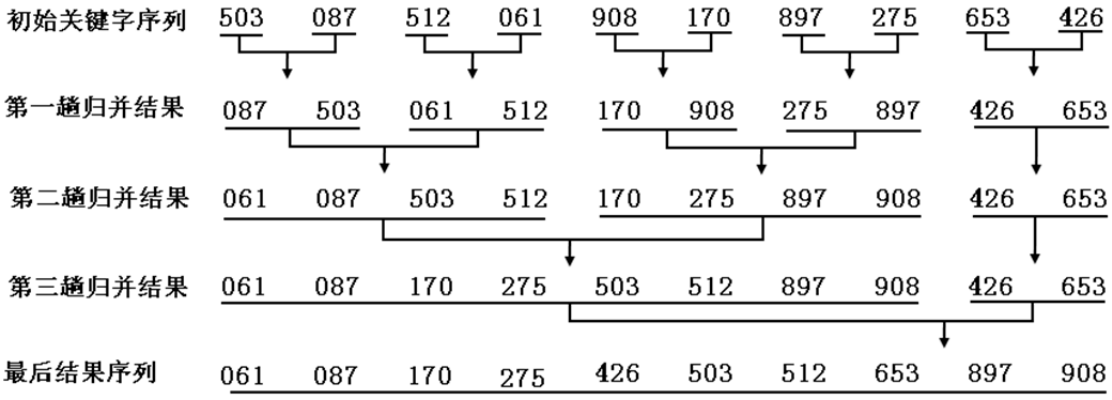



图 9.7 归并排序

⑻基数排序排序各趟排序的结果如图 9.8 所示：

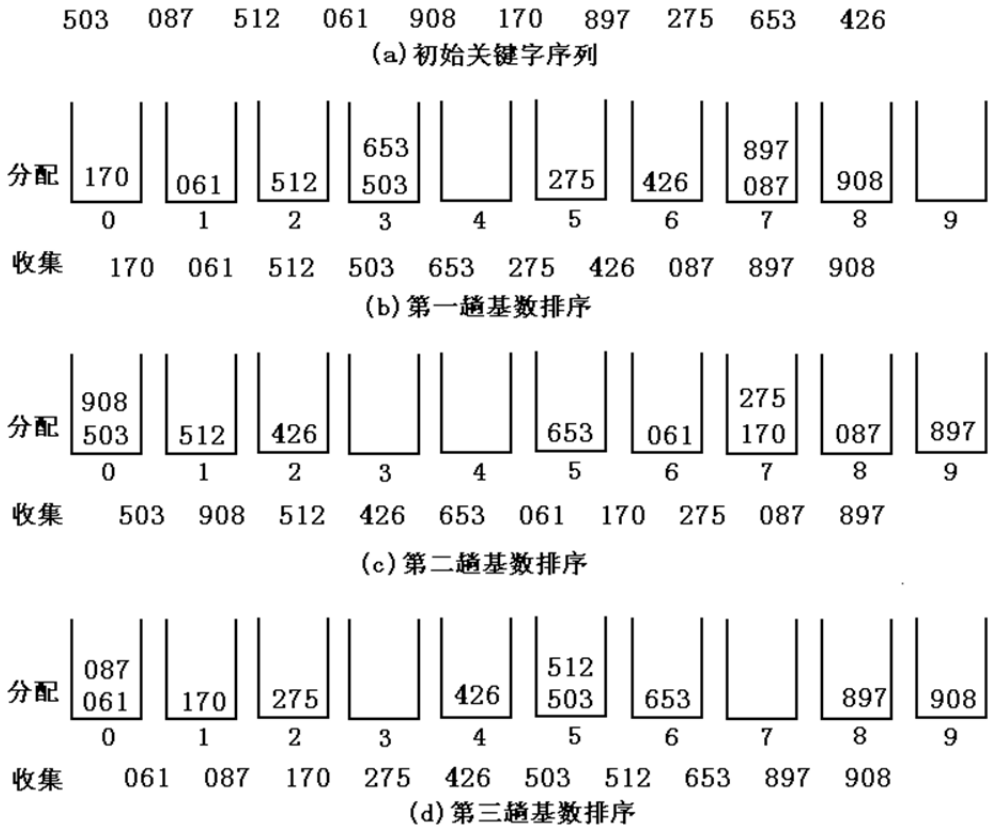



图 9.8 基数排序

3．解答：（说明：教材中的图 9.19 有误，其中的 R=4 应改为 R=6）

二分法插入第 8 个记录的比较过程如图 9.9 所示：

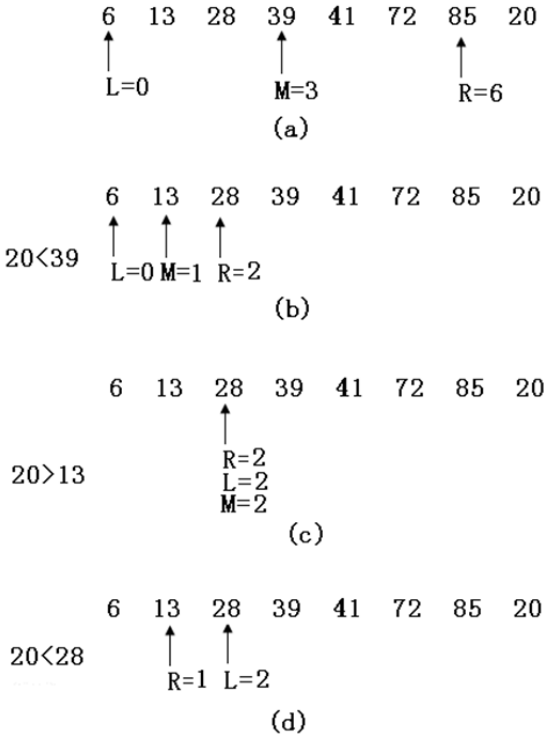



图 9.9 二分插入排序过程

将 R+1(即第 3 个)后的元素向后移动，并将 20 放入 R+1 处，结果为

6,13,20,28,39,41,72,85。

⑴使用二分法插入排序所要进行的比较次数，与待排序的记录的初始状态无关。不论待排序序列是否有序，已形成的部分子序列是有序的。二分法插入首先查找插入位置，插入位置是判定树查找失败的位置。失败位置只能在判定树的最下两层上。

⑵一些特殊情况下，二分法插入排序要比直接插入排序要执行更多的比较。例如，在待排序序列已有序的情况下就是如此。

4．解答：

⑴在最好情况下，假设每次划分均能够划分出左、右两个长度相等的区间：即线性表的长度 $n=2^k-1$，则第一趟快速排序后划分得到两个长度均 n/2 的子表，第二趟快速排序后划分得到 4 个长度均 n/4 的子表；以此类推，总共进行 $k=\log_2(n+1)$遍划分，即子表长度均为 1 排序完毕。当 n=7 时，k=3，在最好情况下，第一个元素由两端向中间扫描到正中位置，即需与其余 6 个元素都进行比较后找到其最终存储位置，因此需比较 6 次，第二趟分别对左右两个子表（长度均为 3，k=2）进行排序，各需比较 2 次，并且继续划分出的每个子表其长度均为 1，即排序完毕；故总共需比较 10 次。

⑵由⑴知，每趟排序排序都应使第一个元素存储于表的正中位置，因此最好情况的初始排序的实例可以为：4,1,3,2,6,5,7。

⑶快速排序最坏的情况是，每趟用来划分的基准元素总是定位于表的第一个位置或最后一个位置，这样划分的左、右子表一个长度为 0，另一个仅是原表长减 1；这样，快速排序的效率就蜕化为冒泡排序，其时间复杂度为 $O(n^2)$。所以当 n=7 时，最坏情况下的比较次数为 6+5+4+3+2+1=21 次。

⑷由⑶知。快速排序最坏的情况是初始序列有序。所以当 n=7 时最坏情况下的初始排序的例子为：

1,2,3,4,5,6,7 或 7,6,5,4,3,2,1

5．解答：

⑴是大根堆；

⑵是大根堆；

⑶不是堆，调成大根堆为：100，98，66，85，80，60，40，77，82，10，20

⑷是小根堆。

6．解答：

可以用队列实现，因为所划分出来的子表可以在任何次序上继续排序。

7．解答：

当初始关键字序列为正序时，直接插入排序需要进行 n-1 次关键字比较，需要 2（n-1）次元素移动；冒泡排序需要 n-1 次关键字比较，需要 0 次元素移动；直接选择排序需要 n(n-1)/2 次关键字比较，需要 0 次元素移动。

当初始关键字为逆序时，直接插入排序需要(n-1)(n+2)/2 次关键字比较，需要(n-1)(n+4)/2 次元素移动，冒泡排序需要 n(n-1)/2 次关键字比较，需要 3n(n-1)/2 次元素移动；直接选择排序需要 n(n-1)/2 次关键字比较，需要 $3\lfloor n/2 \rfloor$次元素移动。

8．解答：

将两个长度为 n 的有序表归并为一个长度为 2n 的有序表，最少需要 n 次关键字比较，

最多需要 2n-1 次关键字比较。

当一个有序表的所有关键字小于另一个有序表的任何一个关键字时，或者说一个有序表的第一个关键字大于另一个有序表的所有元素关键字时，归并所需要的关键字比较次数最少。

当一个有序表中最后一个元素关键字小于另一个有序表的最后一个元素关键字但大于其他元素的关键字时，即在两个有序表的所有元素中，最大的两个元素分别是两个有序表的最后一个元素，这时归并所需要的关键字比较次数最多。

**分析：**

当一个有序表的第一个元素的关键字大于另一个有序表所有元素的关键字时，不妨设第一个有序表的第一个元素的关键字大于第二个有序表所有元素的关键字，那么，在归并过程中，只需要将第一个有序表的第一个元素与第二个有序表的每个元素进行比较后，第二个有序表的所有元素就已经全部放到归并表中，这时就不需要关键字比较，可直接将第一个有序表的所有元素放入归并表中。这样，共需要 n 次比较但大于其他元素的关键字时。

当两个有序表的所有元素中最大的两个元素分别是分别是两个有序表的最后一个元素时，只有将两个有序表的其他元素归并完后，这两个元素才相互比较后依次放入归并表中，总共需要的比较次数为 2(n-1)+1=2n-1 次。

9．解答：

⑴这种排序的结束条件是：当经过 n 趟排序或连续两趟排序没有元素交换时，则排序结束，如果连续两趟排序没有元素交换，说明在序列中任意相邻两个元素之间没有逆序的情况，则整个序列已经排好序。在最坏情况下，经过 n 趟排序就可以将整个序列排好序。

⑵对序列(10,8,15,2,7,13,4)进行奇偶交换排序的过程如下：

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 第一趟： | 8 | 10 | 2 | 15 | 7 | 13 | 4 |
| 第二趟： | 8 | 2 | 10 | 7 | 15 | 4 | 13 |
| 第三趟： | 2 | 8 | 7 | 10 | 4 | 15 | 13 |
| 第四趟： | 2 | 7 | 8 | 4 | 10 | 13 | 15 |
| 第五趟： | 2 | 7 | 4 | 8 | 10 | 13 | 15 |
| 第六趟： | 2 | 4 | 7 | 8 | 10 | 13 | 15 |
| 第七趟： | 2 | 4 | 7 | 8 | 10 | 13 | 15 |

⑶在奇偶交换排序中，每个奇数趟排序需要比较$\lfloor n/2 \rfloor$次；每个偶数趟排序需要比较$\lfloor (n-1)/2 \rfloor$次。这样连续两趟排序所需要的比较次数为$\lfloor n/2 \rfloor$+$\lfloor (n-1)/2 \rfloor$=n-1 次。当初始序列为正序时，前两趟排序没有元素交换，这时就可结束排序过程，所以共需要 n-1 次比较。

当初始序列为逆序时，需要 n 趟排序。如果 n 为偶数，则需要进行 n/2 次奇数趟排序和 n/2 次偶数趟排序，共需要 n(n-1)/2 次比较；如果 n 为奇数，则需要进行[n/2]=(n+1)/2 次奇数趟排序和$\lceil n/2 \rceil$=(n-1)/2 次奇数趟排序和$\lfloor n/2 \rfloor$=(n-1)/2 次偶数趟排序，而每奇数趟排序需要比较$\lfloor n/2 \rfloor$=(n-1)/2 次，而每偶数趟排序需要比较$\lfloor (n-1)/2 \rfloor$=(n-1)/2 次，共需要的比较次数为：

$$[(n+1)/2+(n-1)/2]\times(n-1)/2=n(n-1)/2$$

所以，初始序列为逆序时，共需要的比较次数为 n(n-1)/2 次

**四、算法设计题**

**说明：**以下解答中有关顺序表习题采用一维数组 A 表示，其元素类型为：ElemType；而有关单链表的习题则要包含头文件：LinkList.h。

1. 算法描述如下：

```
void Link_InSort(LinkList L)
{
   LinkList p,q,r;
   p=L->next;
   L->next=NULL;
   while(p)
   {
    r=p;
    p=p->next;
    q=L;
    while(q->next&&q->next->data<=r->data)
      q=q->next;
    r->next=q->next;
    q->next=r;
   }
}
```

2. 算法描述如下：

```
void Select_Sort(ElemType A[],int n)
{
    int  i,j,k;
    ElemType temp;
    for(i=0;i<n-1;i++)
    {
      k=i;
      for(j=i+1;j<n;j++)
      {
       if(A[j].key<A[k].key)  k=j;
      }
      if(i!=k)
      {
          temp=A[k];
          for(j=k;j>i;j--)
              A[j]=A[j-1];
          A[i]=temp;
      }
    }
}
```

3．本题的算法思想是：用 i 和 k 分别记录无序序列的开始和结束位置，在经过一趟正向冒泡排序后，最大元素被交换到第 k 个位置，此时 k 向前移动一个位置；然后再进行一趟反向冒泡排序后，将最小元素交换到第 i 个位置，i 向后移动一个位置。这样正向和反向交替进行，直到 i 与 k 相等为止。其算法描述如下：

```
void Bubble_Sort(ElemType A[],int n)
{
   int  i,j,k;
   bool exchange;
   i=1;
   k=n;
   while(i<k)
   {
     exchange=false;
     for(j=i;j<k;j++)
     if(A[j+1].key<A[j].key)
     {
       exchange=true;
       A[0]=A[j];
       A[j]=A[j+1];
       A[j+1]=A[0];
     }
     if(!exchange)  return;
     k--;
     exchange=false;
     for(j=k;j>i;j--)
     if(A[j].key<A[j-1].key)
     {
        exchange=true;
        A[0]=A[j];
        A[j]=A[j-1];
        A[j-1]=A[0];
     }
     if(!exchange)  return;
     i++;
   }
}
```

4．本题的算法思想是：第一趟对所有偶数的 i，将 A[i].key 和 A[i+1].key 进行比较，第二趟对所有奇数的 i，将 A[i].key 和 A[i+1].key 进行比较，每次比较时若有 A[i].key>A[i+1].key，则将二者交换，以后重复上述二趟过程交换进行，直至整个数组有序。其算法描述如下：

```
void OEsort(ElemType A[],int n)
{
   int i;
```

```
   bool flag;
   ElemType temp;
   do
   {
     flag=false;
     for(i=0;i<n;i++)
     {
       if(A[i].key>A[i+1].key)
       {
          flag=true;
          temp=A[i+1];
          A[i+1]=A[i];
          A[i]=temp;
       }
       i++;
     }
     for(i=1;i<n;i++)
     {
       if(A[i].key>A[i+1].key)
       {
         flag=true;
         temp=A[i+1];
         A[i+1]=A[i];
         A[i]=temp;
       }
       i++;
     }
   }while(flag);
}
```

5. **说明：**快速排序的非递归算法需要利用堆栈，堆栈中元素类型定义如下：

```
typedef struct{
    int left;                    // 区间左端点
    int right;                   // 区间右端点
}Stack;
```

算法描述如下：

```
void NRQuickSort(ElemType A[],int low,int high)
{  // 采用快速序法对A[0]～A[n-1]排序
  int i,j;
  ElemType temp;
  int top=0;
  Stack *S;
  S=(Stack *)malloc(sizeof(Stack)*(high-low+1));
```

```
  S[top].left=low;                    // 区间[low,high]进栈
  S[top].right=high;
  top++;
  while(top>=0)                       // 栈非空
  {
    i=low;
    j=high;
    temp=A[low];
    while(i<j)                        // 划分左右区间
    {
       while(i<j&&A[j].key>=temp.key)j--;
       if(i<j)
       {  A[i]=A[j];
          i++;
       }
       while(i<j&&A[i].key<temp.key)i++;
       if(i<j)
       {  A[j]=A[i];
          j--;
       }
    }
    A[i]=temp;                        // i 或 j 的位置就是基准元素的最终位置
    if(low<i-1)                       // 左区间中不止一个元素,左区间进栈
    {
      S[top].left=low;
      S[top].right=i-1;
      top++;
    }
    if(j+1<high)                      // 右区间中不止一个元素,右区间进栈
    {

      S[top].left=j+1;
      S[top].right=high;
      top++;
    }
    if(top>=0)
    {
        top--;
        low=S[top].left; high=S[top].right;
    }
  }
free(S);
}
```

6.本题的算法思想是：设 n 个关键字互不相同的元素顺序存储在顺序表 A[0..n-1]中，首先，对每个元素统计关键字比它小的元素个数，将统计结果保存在数组 count[0..n-1]中，然后再根据数组 count[0..n-1]的值将每个元素移到正确的位置上。其算法描述如下：

```
void Count_Sort(ElemType A[],int n)
{
  int i,j,x,*count;
  ElemType temp;
  count=(int *)malloc(sizeof(int)*n);
  for(i=0;i<n-1;i++)
      count[i]=0;
  for(i=0;i<n-1;i++)
    for(j=i+1;j<n;j++)
      if(A[i].key<A[j].key)
          count[j]++;
      else
          count[i]++;
  for(i=0;i<n-1;i++)
      while(count[i]!=i)
      {
          x=count[i];
          count[i]=count[x];
          count[x]=x;
          temp=A[i];
          A[i]=A[x];
          A[x]=temp;
      }
      free(count);
}
```

**说明：**若元素序列中存在关键字相同的元素，那么只需要将上面算法中关键字比较运算改为小于等于（≤）比较，便能正确地进行计数排序，并且是稳定的。

7.

⑴算法描述如下：

```
void Append(KeyType  K[], int n)
{  // 由叶子至根将（K1.K2,K3,…，Kn）调整成为大根堆
   KeyType x;
   int i,j;
   x=K[n];
   i=n/2;
   j=n;
   while(x>K[i] && i>=1)
   {
     K[j]=K[i];
     j=i;
```

```
     i=i/2;
    }
    K[j]=x;
 }
```

⑵利用⑴的算法建大根堆的算法如下：

```
void buildheap(KeyType K[],int n)
{
   int  i;
   for(i=2;i<=n;i++)
     Append(K,i);
}
```

8. 算法描述如下：

```
# include "SqQueue.h"       //包含顺序队列数据对象的描述及相关操作
void Radix_Sort(ElemType A[],int n,int m,int d)
{     // 采用基数排序法对 A[0]～A[n-1]排序,A[i]的关键字为 d 进制 m 位
 int i,j,k,power=1;
 SqQueue *tub;
 tub=(SqQueue *)malloc(sizeof(SqQueue)*d);  // 申请 d 个桶空间
 for(i=0;i<d;i++)
     InitQueue_Sq(tub[i]);                          // d 个桶初始化
 for(i=0;i<m;i++)
 {  if(i==0)power=1;
    else   power=power*d;
    for(j=0;j<n;j++)                                 // 分配
    {   k=A[j].key/power-(A[j].key/(power*d))*d;  // 计算第 i 位上的数值
        EnQueue_Sq(tub[k],A[j]);                     // 元素 A[j]进第 k 号桶
    }
    k=0;
    for(j=0;j<d;j++)                                 // 收集
       while(!QueueEmpty_Sq(tub[j]))
       {   DeQueue_Sq(tub[j],A[k]);
           k++;
       }
   }
  }
```

9. 本题结点数据类型为：DuLinkList，其双向冒泡排序算法描述如下：

```
void TwoWayBubbleSort(DuLinkList L)
{  // 对带头结点的双向循环链表 L 中的元素进行双向冒泡排序
   bool exchange=true;           // 设标记
   DuLinkList p,temp,head,tail;
   head=L;          // head->next 是向下冒泡的开始结点
   tail=L;          // tail->next 是向上冒泡的开始结点
```

```
    while(exchange)
    {
        p=head->next;          //p 是工作指针，指向当前结点
        exchange=false;        //假定本趟无交换
        while(p->next!=tail)  // 向下(右)冒泡，一趟有一最大元素沉底
        if(p->data>p->next->data) //交换两结点指针，涉及 6 条链
        {
            temp=p->next;
            exchange=true;             //有交换
            p->next=temp->next;
            temp->next->prior=p;       //先将结点从链表上摘下
            temp->next=p;
            p->prior->next=temp;       //将 temp 插到 p 结点前
            temp->prior=p->prior;
            p->prior=temp;
        }
        else p=p->next;                //无交换，指针后移
        tail=p;                        //准备向上冒泡
        p=tail->prior;
        while(exchange&&p->prior!=head)  //向上(左)冒泡,一趟有一最小元素冒出
        if(p->data<p->prior->data)       //交换两结点指针，涉及 6 条链
        {
            temp=p->prior;
            exchange=true;                //有交换
            p->prior=temp->prior;
            temp->prior->next=p;           //先将 temp 结点从链表上摘下
            temp->prior=p;
            p->next->prior=temp;           //将 temp 插到 p 结点后(右)
            temp->next=p->next;
            p->next=temp;
        }
        else p=p->prior;                   //无交换，指针前移
        head=p;                            //准备向下起泡
    }
}
```

10．本题的基本思想是：把待查记录看作枢轴，先由后向前依次比较，若小于枢轴，则从前向后，直到查找成功返回其位置或失败返回-1 为止。

```
int Search(ElemType r[],int l,int h,KeyType key)
{
    int i=l,j=h;
    while(i<j)
    {
       while(i<=j&& r[j].key>key&&j>l) j--;
```

```
        if(r[j].key==key) return  j;
        else  j--;
        while (i<=j&& r[i].key<key&&i<h) i++;
        if(r[i].key==key) return  i;
        else   i++;
    }
    printf("not find");  return  -1;
}
```

11. 采用置换——选择排序法生成的初始归并段如下：
归并段 1：2，8，12，16，28，30
归并段 2：4，6，10，18，20

12.这里 k=4,m=8,k-(m-1)%(k-1)-1=2,则设两个虚段。4 阶最佳归并树如图 9.10 所示：

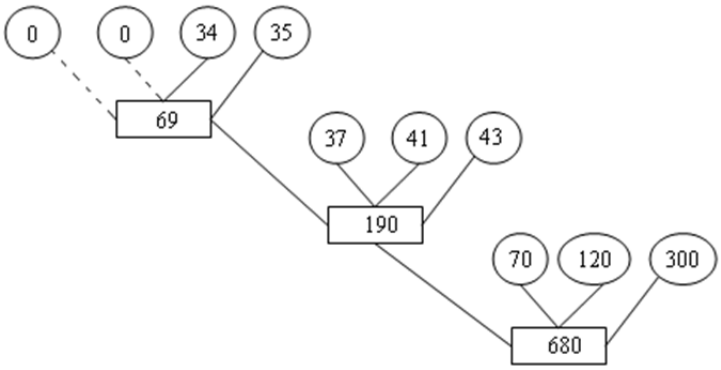



图 9.10 4 阶最佳归并树

第 1 趟读写记录数：34+35=69
第 2 趟读写记录数：69+37+41+43=190
第 3 趟读写记录数：190+70+120+300=680
总的读写记录数：69+190+680=939，总的读写字节数为 939。

# 习题十

## 一、选择题

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

## 二、填空题

1. 操作系统文件　数据库
2. 单关键字文件　多关键字文件
3. 顺序组织　索引组织　哈希组织　链组织
4. 第 i-1
5. 索引集　顺序集　数据集
6. 随机

7．提高查找速度

8．分配和释放存储空间　　重组　　对插入的记录

9．逻辑顺序和物理顺序

10．主关键字

**三、简述题**

1．解答：

⑴顺序结构，相应文件为顺序文件，其记录按存入文件的先后次序顺序存放。顺序文件本质上就是顺序表。若逻辑上相邻的两个记录在存储位置上相邻，则为连续文件；若记录之间以指针相链接，则称为串联文件。顺序文件只能顺序存取，要更新某个记录，必须复制整个文件。顺序文件连续存取的速度快，主要适用于顺序存取，批量修改的情况。

⑵带索引的结构，相应文件为索引文件。索引文件包括索引表和数据表，索引表中的索引项包括数据表中数据的关键字和相应地址，索引表有序，其物理顺序体现了文件的逻辑次序，实现了文件的线性结构。索引文件只能是磁盘文件，既能顺序存取，又能隋机存取。

⑶散列结构，也称计算寻址结构，相应文件称为散列（哈希）文件，其记录是根据关键字值经散列函数计算确定其地址，存取速度快，不需索引，节省存储空间。不能顺序存取，只能随机存取。

其它文件均由以上文件派生而得。

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

2．解答

在主文件外，再建立索引表指示关键字及其物理记录的地址间一一对应关系。这种由索引表和主文件一起构成的文件称为索引文件。索引表依关键字有序。主文件若按关键字有序称为索引顺序文件，否则称为索引非顺序文件（通常简称索引文件）。索引顺序文件因主文件有序，一般用稀疏索引，占用空间较少。常用索引顺序文件有 ISAM 和 VSAM。ISAM 采用静态索引结构，而 VSAM 采用 B+树的动态索引结构。索引文件既能顺序存取，也能随机存取。

3．解答：

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

4．解答：

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

VSAM 文件采用 B+树动态索引结构，文件只有控制区间和控制区域等逻辑存储单位，与外存储器中柱面、磁道等具体存储单位没有必然联系。VSAM 文件结构包括索引集、顺序集

和数据集三部分，记录存于数据集中，顺序集和索引集构成 B+树，作为文件的索引部分可实现顺链查找和从根结点开始的随机查找。

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

5．解答：

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

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

6．解答：

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

7．解答：

⑴顺序文件只能顺序查找，优点是批量检索速度快，不适于单个记录的检索。顺序文件不能象顺序表那样插入、删除和修改，因文件中的记录不能象向量空间中的元素那样“移动”，只能通过复制整个文件实现上述操作。

⑵索引非顺序文件适合随机存取，不适合顺序存取，因主关键字未排序，若顺序存取会引起磁头频繁移动。索引顺序文件是最常用的文件组织，因主文件有序，既可顺序存取也可随机存取。索引非顺序文件是稠密索引，可以“预查找”，索引顺序文件是稀疏索引，不能“预查找”，但由于索引占空间较少，管理要求低，提高了索引的查找速度。

⑶散列文件也称直接存取文件，根据关键字的散列函数值和处理冲突的方法，将记录散列到外存上。这种文件组织只适用于像磁盘那样的直接存取设备，其优点是文件随机存放，记录不必排序，插入、删除方便，存取速度快，无需索引区，节省存储空间。缺点是散列文件不能顺序存取，且只限于简单查询。经多次插入、删除后，文件结构不合理，需重组文件，但这个工作是很费时的。

8．解答：

类似最优二叉树（哈夫曼树），可先合并含较少记录的文件，后合并较多记录的文件，使移动次数减少。见下面的哈夫曼树。

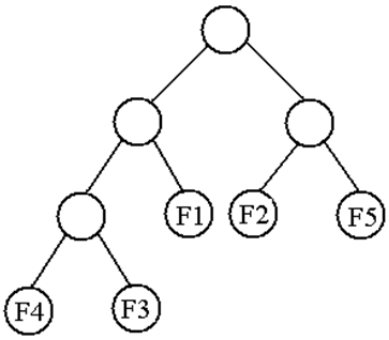



图 10.1 哈夫曼树

9．解答：

多重表文件如图 10.2 所示：

| 记录地址 | 职工号 | 姓　名 | 性别 | 性别链 | 年龄 | 年龄链 | 职　业 | 职业链 | 籍贯 | 月工资 | 工资链 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 10032 | 034 | 刘激扬 | 男 | 10104 | 29 | 10104 | 教　师 | 10068 | 山东 | 720 | 10176 |
| 10068 | 064 | 蔡晓莉 | 女 | 10248 | 40 | 10140 | 教　师 | 10140 | 辽宁 | 1200 | 10140 |
| 10104 | 073 | 朱　力 | 男 | 10140 | 26 | 10176 | 实验员 | 10284 | 广东 | 480 | 10212 |
| 10140 | 081 | 洪　伟 | 男 | 10176 | 36 | 10212 | 教　师 | 10176 | 北京 | 1400 | 10320 |
| 10176 | 092 | 卢声凯 | 男 | 10212 | 28 | 10248 | 教　师 | 10212 | 湖北 | 900 | 10248 |
| 10212 | 123 | 林德庚 | 男 | ∧ | 39 | ∧ | 教　师 | 10248 | 江西 | 480 | ∧ |
| 10248 | 140 | 熊南苏 | 女 | 10284 | 27 | ∧ | 教　师 | 10320 | 上海 | 1000 | 10284 |
| 10284 | 175 | 吕　颖 | 女 | 10320 | 48 | ∧ | 实验员 | ∧ | 江苏 | 580 | ∧ |
| 10320 | 209 | 袁秋慧 | 女 | ∧ | 32 | ∧ | 教　师 | ∧ | 广东 | 1350 | ∧ |

(a)多重表文件

| 性别 | 头指针 | 长度 |
|---|---|---|
| 男 | 10032 | 5 |
| 女 | 10068 | 4 |

(b)性别索引

| 年龄 | 头指针 | 长度 |
|---|---|---|
| ⩽30 | 10032 | 4 |
| 31～35 | 10320 | 1 |
| 36～40 | 10068 | 3 |
| 41～45 | ∧ | 0 |
| >45 | 10284 | 1 |

(c)年龄索引

| 职业 | 头指针 | 长度 |
|---|---|---|
| 教　师 | 10032 | 7 |
| 实验员 | 10104 | 2 |

(d)职业索引

| 月工资 | 头指针 | 长度 |
|---|---|---|
| <500 | 10104 | 2 |
| 500～1000 | 10032 | 4 |
| >1000 | 10140 | 3 |

(e)月工资索引

图 10.2 多重表文件及索引文件

10．解答：

主索引及相应的倒排索引如图 10.3 所示：

| 职工号 | 物理地址 |
|---|---|
| 034 | 10032 |
| 064 | 10068 |
| 073 | 10104 |
| 081 | 10140 |
| 092 | 10176 |
| 123 | 10212 |
| 140 | 10248 |
| 175 | 10284 |
| 209 | 10320 |

(a)主索引文件

| 性别 | 物理地址 |
|---|---|
| 男 | 10032, 10104, 10140, 10176, 10212 |
| 女 | 10068, 10248, 10284, 10320 |

(b)性别倒排表

| 年龄 | 物理地址 |
|---|---|
| ≤30 | 10032, 10104, 10176, 10248 |
| 31～35 | 10320 |
| 36～40 | 10068, 10104, 10212 |
| 41～45 | ∧ |
| >45 | 10284 |

(c)年龄倒排表

| 职业 | 物理地址 |
|---|---|
| 教　师 | 10032, 10068, 10140, 10176, 10212, 10248, 10320 |
| 实验员 | 10104, 10284 |

(d)职业倒排表

| 月工资 | 物理地址 |
|---|---|
| <500 | 10104, 10212 |
| 500～1000 | 10032, 10176, 10248, 10284 |
| >1000 | 10068, 10140, 10320 |

(e)月工资倒排表

图 10.3 倒排文件

在图 10.3 中进行题中的各类搜索，其结果如下：

⑴男性职工（搜索性别倒排表）：{10032, 10104, 10140, 10176, 10212}

⑵月工资超过 900 的职工（搜索工资倒排表）：{10176, 10248, 10068, 10140, 10320}

⑶月工资超过平均工资的职工（搜索工资倒排表）：月平均工资为 901 元，所以结果与⑵一样。

⑷职业为实验员的男性职工（搜索职业和性别倒排表）：

{10104，10284}∩{10032，10104，10140，10176，10212}

={10104}

⑸年龄超过 30 岁且职业为实验员或教师的女性职工（搜索性别和年龄倒排表）：

{10068, 10248, 10284, 10320}∩{10320, 10068, 10104, 10212, 10284}

={10320, 10068, 10284}