数据结构

第 1 章　绪论

重点归纳

1、（1）**数据结构**课程主要是研究非数值计算的程序设计问题中的计算机操作对象以及它们之间的关系和操作的学科。

（2）**数据结构**是指互相之间存在着一种或多种关系的数据元素的集合。

（3）**数据结构**是一个二元组 Data_Structure ＝（D，R），其中，D 是数据元素的有限集，R 是 D 上关系的有限集。

2、数据是对信息的一种符号表示。在计算机科学中是指所有能输入到计算机中并被计算机程序处理的符号的总称。

数据元素是数据的基本单位。

数据项是数据的不可分割的最小单位。

一个数据元素可由若干个数据项组成。

数据对象是性质相同的数据元素的集合，是数据的一个子集。

抽象数据类型是指一个数学模型以及定义在该模型上的一组操作。它实际上就是对该数据结构的定义，定义了一个数据的逻辑结构以及在此结构上的一组算法。

抽象数据类型用三元组（D，S，P）描述.

3、逻辑结构是指数据之间的相互关系。通常分为四类结构：

（1）集合（2）线性结构（3）树型结构（4）图状结构或网状结构

4、存储结构是指数据结构在计算机中的表示，又称为数据的物理结构。通常由四种基本的存储方法实现：

（1）顺序存储方式（2）链式存储方式（3）索引存储方式（4）散列存储方式

5、算法是对特定问题求解步骤的一种描述，是指令的有限序列。其中每一条指令表示一个或多个操作。

具有下列特性：⑴有穷性⑵确定性⑶可行性⑷输入⑸输出。

评价算法：⑴正确⑵可读⑶健壮⑷高效。

6、时间复杂度：以基本运算的原操作重复执行的次数作为算法的时间度量。一般情况下，算法中原操作重复执行的次数是规模 n 的某个函数 T(n)。

常见的渐进时间复杂度有：

$O(1)<O(\log_2 n)<O(n)<O(n\log_2 n)<O(n^2)<O(n^3)<O(2^n)$ $<O(n!)$ $<O(n^n)$

注意：有的情况下，算法中基本操作重复执行的次数还随问题的输入数据集不同而不同。

7、算法的存储空间度量：若输入数据所占空间只取决与问题本身，和算法无关，则只需要分析除输入和程序之外的辅助变量所占额外空间。

原地工作：若所需额外空间相对于输入数据量来说是常数，则称此算法为原地工作。

8、（1）for (int i=1; i<=n; i++)

　　for (int j=1; j<=i; j++)

　　语句频度是 n (n+1)/2

（2）for(i=1; i<=n; i++)

　　for (j=1; j<=i; j++)

　　for (k=1; k<=j; k++)

语句频度是 n (n+1)(n+2)/6

（3）循环条件是 $(y+1)^2 \leqslant n$，即 $y \leqslant \sqrt{n}-1$，而 y 的初始值是 0，则语句频度应为 $\lfloor \sqrt{n} \rfloor$。

习题：

1. 以下属于逻辑结构的是（　　）。

A. 顺序表　　B. 散列表　C. 有序表　　D. 单链表

第 2 章　线性表

重点归纳

1、线性表是具有相同数据类型的n(n≥0)个数据元素的有限序列，是最简单、最基本、也是最常用的一种线性结构，

2、顺序表上基本运算的实现

（1）顺序表具有**按数据元素的序号随机存取**的特点，时间复杂度为O(1)。

（2）**按值x查找**：主要运算是比较，比较的次数与值x在表中的位置有关，也与表长有关，平均比较次数为（n+1）/2，时间复杂度为O(n)。

（3）插入运算：在第i个位置上插入x，从a_n到a_i都要向下移动一个位置，共移动**n－i＋1**个元素。

等概率情况下，平均移动数据元素的次数：$E_{in}=\sum_{i=1}^{n+1} p_i(n-i+1)=\frac{1}{n+1}\sum_{i=1}^{n+1}(n-i+1)=\frac{n}{2}$

说明：在顺序表上做插入操作需移动表中一半的数据元素，时间复杂度为O(n)。

（4）删除运算：删除第i个元素，从a_{i+1}到a_n都要向上移动一个位置，共移动 **n-i** 个元素。

等概率情况下，平均移动数据元素的次数：$E_{de}=\sum_{i=1}^{n} p_i(n-i)=\frac{1}{n}\sum_{i=1}^{n+1}(n-i)=\frac{n-1}{2}$

说明：顺序表上作删除运算时大约需要移动表中一半的元素，时间复杂度为O(n)。

3、单链表上基本运算的实现

（1）建立带头结点的单链表

●**头插法**：读入的数据元素的顺序与生成的链表中元素的顺序是相反的。

```
LinkList  Creat_LinkList1( )
{ LinkList L=(LNode *)malloc(sizeof(LNode));
  L->next=NULL;                              /*空表*/
  LNode *s;
  int x;                    /*设数据元素的类型为int*/
  scanf("%d",&x);
  while (x!=flag)
    { s=malloc(sizeof(LNode));
      s->data=x;
      s->next=L->next;  L->next=s;
     scanf ("%d",&x);
    }
  return L;
  }
```

●**尾插法**：读入的数据元素的顺序与生成的链表中元素的顺序是一致的。

```
LinkList  Creat_LinkList2( )
{ LinkList L=(LNode *)malloc(sizeof(LNode));
  L->next=NULL;                              /*空表*/
  LNode  *s,*r=L;
  int x;                    /*设数据元素的类型为int*/
  scanf("%d",&x);
  while (x!=flag)
  { s=malloc(sizeof(LNode));
    s->data=x;
    r->next=s; r=s;              /*r指向新的尾结点*/
   scanf("%d",&x);
  }
  r->next=NULL;
  return L;
}
```

（2）求表长

●设L是带头结点的单链表(线性表的长度不包括头结点)。

```
int  Length_LinkList1 (LinkList L)
{ LNode  * p=L;                          /*p指向头结点*/
  int  j=0;
  while (p->next)
    { p=p->next; j++ }       /*p所指的是第 j 个结点*/
    return  j;
}
```

●设L是不带头结点的单链表。

```
int  Length_LinkList2 (LinkList L)
{ LNode  * p=L;
  int  j;
  if (p==NULL) return  0;      /*空表的情况*/
      j=1;                     /*在非空表的情况下，p所指的是第一个结点*/
  while (p->next )
    { p=p->next;  j++ }
  return  j;
}
```

两个算法的时间复杂度均为O(n)。不带头结点的单链表空表情况要单独处理，而带上头结点之后则不用了。**在以后的算法中不加说明则认为单链表是带头结点的。**

（3）查找

●按序号查找 Get_Linklist(L,i)

```
Lnode  * Get_LinkList(LinkList  L, int  i);     /*在单链表L中查找第i个元素结点*/
{ LNode  * p=L;
  int  j=0;
  while (p->next !=NULL && j<i )
    {  p=p->next;  j++;  }
```

```
  if (j==i) return p;
  else  return NULL;
 }
```
●按值查找即定位 Locate_LinkList(L,x)
```
Lnode  * Locate_LinkList( LinkList L, ElemType x) /*在单链表L中查找值为x的结点*/
{ LNode  * p=L->next;
  while ( p!=NULL && p->data != x)
    p=p->next;
  return p;
}
```
以上两个算法的时间复杂度均为0(n)。

（4）插入

●后插结点：设p指向单链表中某结点，s指向待插入的值为x的新结点，将*s插入到*p的后面，插入过程如图1-2-1。

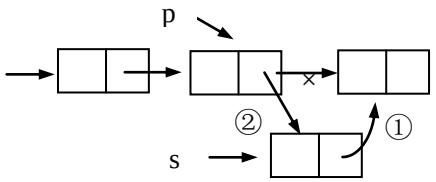


图 1-2-1　在*p 之后插入

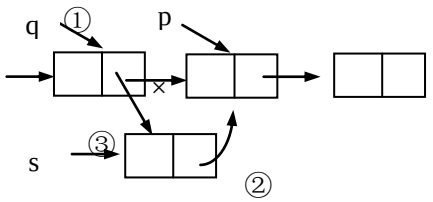


图 1-2-2　在*p 之前插入*s

操作如下：　①s->next=p->next;
　　②p->next=s;

注意：两个指针的操作顺序不能交换。后插操作的时间复杂度为0(1)。

●前插结点：设 p 指向链表中某结点， s 指向待插入的值为x的新结点，将*s插入到*p的前面，插入过程如下图1-2-2，与后插不同的是：首先要找到*p的前驱*q，然后再完成在*q之后插入*s，设单链表头指针为L。

操作如下：①q=L;
```
          while (q->next!=p)  q=q->next;       /*找*p的直接前驱*/
```
②s->next=q->next;
③q->next=s;

前插操作因为要找*p的前驱，时间性能为0(n)。其实前插操作可以用后插操作来实现，即仍然可以将 *s 插入到 *p 的后面，然后将 p->data与s->data交换即可，这样即满足了逻辑关系，也能使得时间复杂度为0(1)。

（5）删除

●删除p结点：设p指向单链表中某结点，删除*p。操作过程如下图1-2-3。要实现对结点*p的删除，首先要找到*p的前驱结点*q，然后完成指针的操作即可。

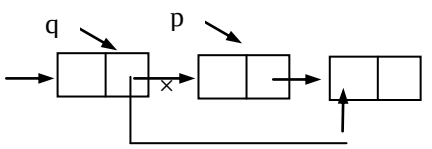


图 1-2-3　删除*p

操作如下：①q=L;

```
while (q->next!=p)  q=q->next;        /*找*p的直接前驱*/
②q->next=p->next;
③free(p);
```

显然找*p前驱的时间复杂度为O(n)。

●删除p结点的后继结点：若要删除*p的后继结点(假设存在)，则可以直接完成：

操作如下：①s=p->next;p->next=s->next;

②free(s);

该操作的时间复杂度为O(1)。

4、循环链表：在循环链表上的操作基本上与非循环链表相同，只是将原来判断指针是否为NULL变为是否是头指针而已，没有其它较大的变化。

对于单链表只能从头结点开始遍历整个链表，而对于单循环链表则可以从表中任意结点开始遍历整个链表。不仅如此，有时对链表常做的操作是在表尾、表头进行，此时可以改变一下链表的标识方法，不用头指针而用一个指向尾结点的指针 R 来标识，可以使得操作效率得以提高。

5、双向链表

（1）插入：设 p 指向双向链表中某结点，s 指向待插入的值为 x 的新结点，将*s 插入到*p 的前面，插入过程如下图 1-2-4。

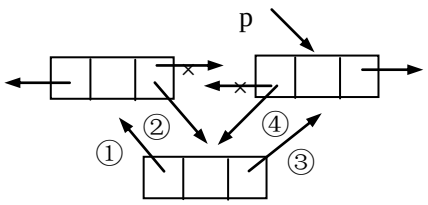


图1-2-4　双向链表中的结点插入

操作如下：①s->prior=p->prior;　②p->prior->next=s;

③s->next=p;　④p->prior=s;

指针操作的顺序不是唯一的，但也不是任意的，操作①必须要放到操作④的前面完成，否则*p 的前驱结点的指针就丢掉了。只要把每条指针操作的涵义搞清楚，就不难理解了。

（2）删除：设 p 指向双向链表中某结点，删除*p。操作过程如下图 1-2-5。

操作如下：①p->prior->next=p->next;

②p->next->prior=p->prior;

free(p);

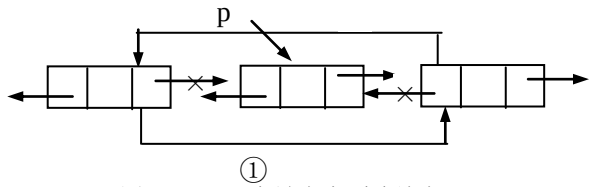


图 1-2-5　双向链表中删除结点

6、静态链表

静态链表借助数组来描述线性表的链式存储结构，这里的指针是结点的相对地址(数组的下标)。有关基于静态链表上的线性表的操作基本与动态链表相同，除了一些描述方法有些区别外，算法思路是相同的。

习题：

1. 表长为 n 的顺序存储的线性表，当在任何位置上删除一个元素的概率相等时，删除一个元素所需移动元素的平均个数为（　　）。

   A. n　　B. n/2　　C. (n-1)/2　　D. (n+1)/2

2. 在一个长度为 n 的顺序存储线性表中，删除第 i 个元素(1≤i≤n+1)时，需要从前向后依次前移（　　）个元素。

   A. n-i　　B. n-i+1　　C. n-i-1　　D. i

3. 设单链表中结点的结构为

```
TYPEDEF  STRUCT  NODE {   //链表结点定义
ELEMTYPE  DATA;            //数据
STRUCT  NODE  * NEXT;      //结点后继指针
} LISTNODE;
```

已知指针 P 所指结点不是尾结点,若在*P 之后插入结点*S,则应执行下列哪一个操作( )。

A. S->NEXT = P; P->NEXT = S; B. S->NEXT = P->NEXT; P->NEXT = S;

C. S->NEXT = P->NEXT; P = S; D. P->NEXT = S; S->NEXT = P;

4. 在一个单链表 HL 中，若要向表头插入一个由指针 p 指向的结点，则执行（ ）。

A. HL = p; p->next = HL; B. p->next = HL; HL = p;

C. p->next = HL; p = HL; D. p->next = HL->next; HL->next = p;

5. 非空的循环单链表 FIRST 的尾结点（由 P 所指向）满足：（ ）

A. P->NEXT = NULL; B. P = NULL; C. P->NEXT = FIRST; D. P = FIRST;

第 3 章 栈、队列、数组、串和广义表

重点归纳

1、栈和队列是限定**插入和删除只能**在表的“**端点**”进行的线性表。栈又称为后进先出的线性表，队列又称为先进先出的线性表。

线性表	栈	队列
Insert(L, i, x)1≤i≤n+1	Insert(S, n+1, x)	Insert(Q, n+1, x)
Delete(L, i) 1≤i≤n	Delete(S, n)	Delete(Q, 1)

2、**顺序栈**形式描述：

●静态分配：

```
#define STACKSIZE 100
typedef  struct{
  SElemType  elem[STACKSIZE];
  int   top;
}SqStack;
```

这里注意，非空栈时 top 始终在栈顶元素的位置。

●动态分配：

```
#define STACK_INIT_SIZE 100
#define STACKINCREMENT 10
typedef  struct{
  SElemType  *base;
  SElemType  *top;
  int  stacksize;
 }SqStack;
```

这里注意，非空栈时 top 始终在栈顶元素的下一个位置。

顺序栈上基本操作的实现

（1）入栈：若栈不满，则将 e 插入栈顶

●静态分配：

```
Status Push (SqStack &S, SElemType e)
{if(S.top==STACKSIZE-1) return ERROR;/*栈满不能入栈*/
 else { S.elem[++S.top]=e;/*top 始终在栈顶元素的位置*/
```

```
      return OK;
    }
 }
```

●动态分配：

```
Status Push (SqStack &S, SElemType e)
{if (S.top-S.base>=S.stacksize)
    {……}                          /*栈满，追加存储空间*/
  *S.top++ = e;      /*top 始终在栈顶元素的下一个位置*/
  return OK;
}
```

（2）出栈：若栈不空，则删除 S 的栈顶元素，用 e 返回其值，并返回 OK，否则返回 ERROR。

●静态分配：

```
Status Pop(Sqstack &s, SElemType &e)
 { if(S.top==-1)  return ERROR;
   e= S.elem[S.top- -];
   return OK;
 }
```

●动态分配：

```
Status Pop (SqStack &S, SElemType &e)
{ if (S.top==S.base) return ERROR;
  e = *--S.top;
  return OK;
}
```

3、链栈：因为栈中的主要运算是在栈顶插入、删除，显然在链表的头部做栈顶是最方便的，而且没有必要象单链表那样为了运算方便附加一个头结点。

链栈上基本操作的实现

（1）入栈：

```
Status Push (LinkStack &S, SElemType e)
  { StackNode  *q;
    q= (StackNode *) malloc (sizeof (StackNode)) ;
    q->data=e;
    q->next=S;  S=q;
   return OK;
   }
```

（2）出栈

```
Status Pop (LinkStack &S, SElemType &e)
{  StackNode  *p;
   if  (S= =NULL)  return ERROR;
   else { e = S->data;
          p = S; S = S->next;
          free (p);
          return OK;
        }
  }
```

4、链队列上基本操作的实现

（1）入队：插入元素 e 为 Q 的新的队尾元素

```
Status EnQueue (LinkQueue &Q, QElemType e)
  { QNode *p;
    p = (QNode *) malloc (sizeof (QNode)) ;
   p->data = e;
   p->next = NULL;
   Q.rear->next = p;    Q.rear = p;
    return OK;
}
```

（2）出队：若队列不空，则删除 Q 的队头元素，用 e 返回其值，并返回 OK；否则返回 ERROR。

```
Status DeQueue (LinkQueue &Q, QElemType &e)
{ if (Q.front == Q.rear)  return ERROR;
  p = Q.front->next;
  e = p->data;
  Q.front->next = p->next;
  if(Q.rear==p)   Q.rear= Q.front;
  free (p);
  return OK;
}
```

5、顺序队列：设队头指针指向队头元素前面一个位置，队尾指针指向队尾元素（这样的设置是为了某些运算的方便，并不是唯一的方法）。

（1）入队：　　sq->data[++sq->rear]=x;

（2）出队：　　x=sq->data[++sq->front];

6、循环队列上基本操作的实现

（1）入队：

```
Status EnQueue (SqQueue &Q, QElemType e)
{ if((Q.rear+1)%MAXQSIZE == Q.front) return ERROR;
    Q.base[Q.rear] = e;
    Q.rear = (Q.rear+1) % MAXQSIZE;
    return OK;
}
```

（2）出队：

```
Status DeQueue (SqQueue &Q, QElemType &e)
  { if (Q.front = = Q.rear)  return ERROR;
    e = Q.base[Q.front];
    Q.front = (Q.front+1) % MAXQSIZE;
    return OK;
  }
```

（3）循环队列元素个数：**(Q.rear-Q.front+MAXQSIZE) %MAXQSIZE**

7、数组：一般采用顺序存储，是一个随机存取结构。

二维数组**按行优先**寻址计算方法，每个数组元素占据 d 个地址单元。

设数组的基址为 LOC(a_{11})：LOC(a_{ij})=LOC(a_{11})+((i-1)*n+j-1)*d

设数组的基址为 LOC(a_{00})：LOC(a_{ij})=LOC(a_{00})+(i*n+j)*d

二维数组**按列优先**寻址计算方法。

设数组的基址为 LOC(a_{11})：LOC(a_{ij})=LOC(a_{11})+((j-1)*n+i-1)*d

设数组的基址为 LOC(a_{00})：LOC(a_{ij})=LOC(a_{00})+(j*n+i)*d

8、特殊矩阵的压缩存储（假设以行序为主序）

（1）对称矩阵：

将对称矩阵 A 压缩存储到 SA[n(n+1)/2]中，a_{ij}的**下标** i、j 与在 SA 中的对应元素的**下标** k 的关系。

（2）三角矩阵

与对称矩阵类似，不同之处在于存完下（上）三角中的元素之后，接着存储对角线上（下）方的常量，因为是同一个常数，所以存一个即可。将三角矩阵 A 压缩存储到 SA[n(n+1)/2+1]中，a_{ij}的**下标** i、j 与在 SA 中的对应元素的**下标** k 的关系。

（3）三对角矩阵

将三对角矩阵 A 压缩存储到 SA[3n-2]中，a_{ij}的**下标** i、j 与在 SA 中的对应元素的**下标** k 的关系。

9、串

重点掌握 KMP 算法，尤其三 next 值的计算，next 值的本质就是：

(1)如果下标从 0 算起

字符串中最长的相同的前后缀真子串的长度

(2)如果下标从 1 算起

字符串中最长的相同的前后缀真子串的长度+1

10、广义表

定义：广义表是线性表的推广，也称列表(Lists)。它是 n 个元素的有限序列，记作 A=（a1,a2,……an)。其中 A 是表名，n 是广义表的长度，ai 是广义表的元素，ai 既可以是单个元素，也可以是广义表。

表头(Head)：非空广义表的第一个元素 a1;

表尾(Tail)：除了表头的其余元素组成的表;

深度：广义表中括号嵌套的最大层数。

特点：广义表的元素可以是子表，子表的元素还可以是子表，存储空间难以确定，常采用链式存储。

举例：

（1）B=（e）

只含一个原子，长度为 1，深度为 1。

（2）C=(a,(b，c，d))

有一个原子，一个子表，长度为 2，深度为 2。

（3）D=(B,C)

二个元素都是列表，长度为 2，深度为 3。

（4）E=(a,E)

是一个递归表，长度为 2，深度无限，相当于 E=（a,（a,（a,（a,……)))）。

习题：

选择题：

1. 若已知一个栈的入栈序列是 1,2,3,.....n，其输出序列为 p1,p2,p3,.....pn，若 p1=n，则 pi 为（　　）

A. i　　B. n-i　　C. n-i+1　　D.不确定

2. 若一个栈的输入序列为 1，2，3 ...n，输出序列的第一个元素是 i，则第 j 个输出元素是（　）

A. i-j-1　　B. i-j　　C. j-i+1　　D. 不确定

3. 栈在（　）中应用。

A. 递归调用　　B. 子程序调用　　C. 表达式求值　　D. A，B，C

4. 设计一个判别表达式中左，右括号是否配对出现的算法，采用（　）数据结构最佳。

A. 线性表的顺序存储结构　B. 队列　　C. 线性表的链式存储结构　　D. 栈

5. 在解决计算机主机与打印机之间速度不匹配问题时通常设置一个打印数据缓冲区，主机将要输出的数据依次写入该缓冲区，而打印机则从该缓冲区中取出数据打印。该缓冲区应该是一个（　）结构。

A. 栈　　B. 队列　　C. 数组　　D. 线性表

6. 向一个带头结点 HS 的链栈中插入一个 s 所指结点时，则执行（　）

A.HS->next = s;

B. s->next = HS->next; HS->next = s ;

C.s->next = HS ; HS = s ;

D.s->next = HS ; HS = HS->next;

7. 一个循环队列 Q 最多可存储 m 个元素，已知其头尾指针分别是 front 和 rear，则判定该循环队列为满的条件是（　）

A.Q.rear - Q.front == m　　B.Q.rear != Q.front

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

8. 循环队列用数组 A[0..m-1]存放其元素值，已知其头尾指针分别为 front 和 rear，则当前元素个数为（　）。

A. (rear-front+m) mod m　　B. rear-front+1

C. rear-front-1　　D. rear-front

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

A. 1 和 5　　B. 2 和 4　　C. 4 和 2　　D. 5 和 1

10. 二维数组 A 的每个元素是由 6 个字符组成的串，其行下标 i=0,1,…,8,列下标 j=1,2,…,10。若 A 按行先存储，元素 A[8,5]的起始地址与当 A 按列先存储时的元素（　）的起始地址相同。设每个字符占一个字节。

A. A[8,5]　　B. A[3,10]　　C. A[5,8]　　D. A[0,9]

11. 设 n 阶方阵是一个上三角矩阵，则需存储的元素个数为（　）

A. n　　B. n*n　　C. n*n/2　　D. n(n+1)/2

12. 设有一个 10 阶的对称矩阵 A，采用压缩存储方式，以行序为主存储，a_{11} 为第一元素，其存储地址为 1,每个元素占一个地址空间,则 a_{85}(即该元素下标 i=85)的地址为(　　)。

A. 13　　B. 33　　C. 18　　D. 40

13. 若对 n 阶对称矩阵 A[1..n，1..n]以行序为主序方式下将其下三角的元素（包括主对角线上的所有元素）依次存放于一维数组 B[1..n(n+1)/2]中，则在 B 中确定 a_{ij} (i<j)的位置 k 的关系为（　）。

A. i*(i-1)/2+j　　B. j*(j-1)/2+i

C. i*(i+1)/2+j　　D. j*(j+1)/2+i

14. 已知模式串 t=”aaababcaabbcc”，则 t[4]=’b’，next[4]= _____。3

15. 已知模式串 t=”aaababcaabbcc”，则 t[3]=’b’，next[3]= _____。2

16. 广义表((a,b),(()),(a,(b)))有_____个元素。

17. 设有两个子串 s 和 t，判断 t 是否为 s 子串的算法称为_____。
    A. 求子串　　B. 串联接
    C. 串匹配　　D. 求串长
18. 广义表 A=(a,b,(c,d),(e,(f,g)))，则 head(tail(head(tail(tail(A))))) 的值为 _____。
    A. (g)　　B. (d)　　C. (c)　　D. d

第 4 章　树和二叉树

重点归纳

1、**二叉树**：是有序的，即若将其左、右子树颠倒，就成为另一棵不同的二叉树。即使树中结点只有一棵子树，也要区分它是左子树还是右子树。
满二叉树：在一棵二叉树中，如果所有分支结点都存在左子树和右子树，并且所有叶子结点都在同一层上，这样的一棵二叉树称作满二叉树。
完全二叉树：一棵深度为 k 的有 n 个结点的二叉树，对树中的结点按从上至下、从左到右的顺序进行编号，如果编号为 i（1≤i≤n）的结点与满二叉树中编号为 i 的结点在二叉树中的位置相同，则这棵二叉树称为完全二叉树。
完全二叉树的特点是：叶子结点只能出现在最下层和次下层，且最下层的叶子结点集中在树的左部。
2、二叉树的性质：
性质 1：一棵非空二叉树的第 i 层上最多有 2^{i-1} 个结点（i≥1）。
性质 2：一棵深度为 k 的二叉树中，最多具有 2^k-1 个结点。
性质 3：对于一棵非空的二叉树，如果叶子结点数为 n_0，度数为 2 的结点数为 n_2，则有：$n_0=n_2+1$
性质 4：具有 n 个结点的完全二叉树的深度 k 为 $\lfloor \log_2 n \rfloor+1$。
性质 5：对于具有 n 个结点的完全二叉树，如果按照从上至下和从左到右的顺序对二叉树中的所有结点从 1 开始顺序编号，则对于任意的序号为 i 的结点，有：

（1）如果 i＞1，则序号 i 的结点的双亲结点的序号为 $\lfloor i/2 \rfloor$；如果 i＝1，则序号为 i 的结点是根结点，无双亲结点。
（2）如果 2i≤n，则序号为 i 的结点的左孩子结点的序号为 2i；如果 2i＞n，则序号为 i 的结点无左孩子。
（3）如果 2i＋1≤n，则序号为 i 的结点的右孩子结点的序号为 2i＋1；如果 2i＋1＞n，则序号为 i 的结点无右孩子。

此外，若对二叉树的根结点从 0 开始编号，则相应的 i 号结点的双亲结点的编号为 $\lfloor (i-1)/2 \rfloor$，左孩子的编号为 2i＋1，右孩子的编号为 2i＋2。

3、二叉树的存储结构
二叉链表形式描述：

```
typedef struct BiTNode{
  TElemType data;
  struct BiTNode *lchild,*rchild;     /*左右孩子指针*/
```

```
}BiTNode,*BiTree;
```

尽管在二叉链表中无法由结点直接找到其双亲，但由于二叉链表结构灵活，操作方便，对于一般情况的二叉树，甚至比顺序存储结构还节省空间。**后面所涉及到的二叉树的链式存储结构不加特别说明的都是指二叉链表结构。**

4、遍历二叉树（必背的七个算法）

遍历二叉树是以一定规则将二叉树中结点排列成一个线性序列，实质是对一个非线性结构进行线性化操作。

（1）前序遍历的递归实现

```
void PreOrder (BiTree bt) /*前序遍历二叉树 bt*/
{ if (bt==NULL) return;   /*递归调用的结束条件*/
  Visit (bt->data) ;      /*访问结点的数据域*/
  PreOrder (bt->lchild) ; /*前序递归遍历 bt 的左子树*/
  PreOrder (bt->rchild) ; /*前序递归遍历 bt 的右子树*/
}
```

（2）中序遍历的递归实现

```
void InOrder (BiTree bt)  /*中序遍历二叉树 bt*/
{ if (bt==NULL) return;   /*递归调用的结束条件*/
  InOrder (bt->lchild) ;  /*中序递归遍历 bt 的左子树*/
  Visit (bt->data) ;      /*访问结点的数据域*/
  InOrder (bt->rchild) ;  /*中序递归遍历 bt 的右子树*/
}
```

（3）后序遍历的递归实现

```
void PostOrder (BiTree bt) /*后序遍历二叉树 bt*/
{ if (bt==NULL) return;    /*递归调用的结束条件*/
 PostOrder (bt->lchild) ; /*后序递归遍历 bt 的左子树*/
 PostOrder(bt->rchild); /*后序递归遍历 bt 的右子树*/  Visit(bt->data);       /*
 访问结点的数据域*/
 }
```

（4）层序遍历的实现

一维数组 Queue[MAX_TREE_SIZE]用以实现队列，变量 front 和 rear 分别表示当前对队首元素和队尾元素在数组中的位置。

```
void LevelOrder (BiTree bt)     /*层序遍历二叉树 bt*/
{BiTree Queue[MAX_TREE_SIZE];
 int front,rear;
 if (bt==NULL) return;
 front=-1;
 rear=0;
 queue[rear]=bt;
 while(front!=rear)
 {Visit(queue[++front]->data); /*访问队首结点数据域*/
  if (queue[front]->lchild!=NULL)  /*将队首结点的左孩子结点入队列*/
    { queue[++rear]=queue[front]->lchild;
    }
  if (queue[front]->rchild!=NULL)   /*将队首结点的右孩子结点入队列*/
```

```
    {queue[++rear]=queue[front]->rchild;
    }
  }
}
```

（5）前序遍历的非递归实现

二叉树以二叉链表存放，一维数组 stack[MAX_TREE_SIZE]用以实现栈，变量 top 用来表示当前栈顶的位置。

```
void NRPreOrder (BiTree bt) /*非递归先序遍历二叉树*/
{ BiTree stack[MAX_TREE_SIZE],p;
 int top;
 if (bt==NULL) return;
 top=0;
 p=bt;
  while(!(p==NULL&&top==0))
   { while(p!=NULL)
    { Visit(p->data);           /*访问结点的数据域*/
       if(top< MAX_TREE_SIZE-1)/*将当前指针 p 压栈*/
         {stack[top++]=p;
          }
       else {printf("栈溢出");
             return;
              }
      p=p->lchild;          /*指针指向 p 的左孩子结点*/
     }
     if (top<=0) return;    /*栈空时结束*/
     else{ p=stack[- -top];/*从栈中弹出栈顶元素*/
           p=p->rchild;     /*指针指向 p 的右孩子结点*/
         }
   }
}
```

（6）中序遍历的非递归实现

只需将**先序遍历的非递归算法**中的 **Visit(p->data)移到 p=stack[- -top]和 p=p->rchild 之间**即可。

（7）后序遍历的非递归实现

后序遍历与先序遍历和中序遍历不同，在后序遍历过程中，结点在第一次出栈后，还需再次入栈，也就是说，结点要入两次栈，出两次栈，而访问结点是在第二次出栈时访问。因此，为了区别同一个结点指针的两次出栈，设置一标志 flag，令：

$$\text{flag}=\begin{cases}1 & \text{第一次出栈，结点不能访问}\\ 2 & \text{第二次出栈，结点可以访问}\end{cases}$$

当结点指针进、出栈时，其标志 flag 也同时进、出栈。因此，可将栈中元素的数据类型定义为指针和标志 flag 合并的结构体类型。定义如下：

```
    typedef struct {
      BiTree  link;
```

```
      int  flag;
    }stacktype;
```

后序遍历二叉树的非递归算法如下。在算法中，一维数组 stack[MAX_TREE_SIZE]用于实现栈的结构，指针变量 p 指向当前要处理的结点，整型变量 top 用来表示当前栈顶的位置，整型变量 sign 为结点 p 的标志量。

```
void NRPostOrder(BiTree bt) /*非递归后序遍历二叉树 bt*/
{ stacktype stack[MAX_TREE_SIZE];
 BiTree p;
 int top,sign;
 if (bt==NULL) return;
 top=-1;                          /*栈顶位置初始化*/
 p=bt;
 while (!(p==NULL && top==-1))
  { if (p!=NULL)                  /*结点第一次进栈*/
    { stack[++top].link=p;
      stack[top].flag=1;
      p=p->lchild;               /*找该结点的左孩子*/
     }
   else{ p=stack[top].link;
         sign=stack[top- -].flag;
         if (sign==1)             /*结点第二次进栈*/
           { stack[++top].link=p;
             stack[top].flag=2;  /*标记第二次出栈*/
             p=p->rchild;
            }
         else { Visit(p->data); /*访问该结点数据域值*/
                p=NULL;
              }
         }
  }
}
```

5、建立二叉链表方式存储的二叉树

```
void CreateBinTree(BinTree *bt) /*以加入结点的前序序列输入，构造二叉链表*/
{char ch;
 scanf("%c",&ch);
 if (ch=='0') *bt=NULL;  /*读入 0 时，将相应结点置空*/
 else{ *bt=(BinTNode *)malloc(sizeof(BinTNode));  /*生成结点空间*/
       (*bt)->data=ch;
       CreateBinTree(&(*bt)->lchild);  /*构造左子树*/
       CreateBinTree(&(*bt)->rchild);  /*构造右子树*/
     }
}
```

6、线索二叉树

（1）按照某种遍历方式对二叉树进行遍历，可把二叉树中所有结点排列为一个线性序列，

二叉树中每个结点在这个序列中的直接前驱结点和直接后继结点是什么，二叉树的存储结构中并没有反映出来，只能在对二叉树遍历的动态过程中得到这些信息。为了保留结点在某种遍历序列中直接前驱和直接后继的位置信息，可以利用具有 n 个结点的二叉树中的叶子结点和一度结点的 **n＋1 个空指针域**来指示，这些指向直接前驱结点和指向直接后继结点的指针被称为线索，加了线索的二叉树称为线索二叉树。

线索二叉树将为二叉树的遍历提供许多遍历，遍历时可不需要栈，也不需要递归了。

（2）线索二叉树的存储表示的描述：

```
typedef enum PointerTag {Link,Thread};
typedef struct BiThrNode {
  TElemType         data;
  struct BiThrNode *lchild,*rchild;
  PointerTag        Ltag,Rtag;
}BiThrNode, *BiThrTree;
```

（3）建立一棵中序线索二叉树：

实质上就是遍历一棵二叉树。在遍历过程中，Visit 操作是检查当前结点的左、右指针域是否为空，如果为空，将它们改为指向前驱结点或后继结点的线索。为实现这一过程，设指针 pre 始终指向刚刚访问过的结点，即若指针 p 指向当前结点，则 pre 指向它的前驱，以便增设线索。

```
int  InOrderThr(BiThrTree &head,BiThrTree T)
/*中序遍历二叉树 T，将其中序线索化，head 指向头结点。*/
{if (!(head=(BiThrNode *)malloc(sizeof(BiThrNode))))  return 0;
 head->Ltag=0;
 head->Rtag=1;                    /*建立头结点*/
 head->rchild=head;               /*右指针回指*/
 if (!T) head->lchild =head;      /*若二叉树为空，则左指针回指*/
 else{ head->lchild=T;
       pre= head;
       InThreading(T);     /*中序遍历进行中序线索化*/
       pre->rchild=head;
       pre->rtag=1;              /*最后一个结点线索化*/
       head->rchild=pre;
     }
 return 1;
}
void InTreading(BiThrTree p)
{if (p)
  { InThreading(p->lchild);         /*左子树线索化*/
    if (!p->lchild)                     /*前驱线索*/
     { p->ltag=1;
       p->lchild=pre;
      }
    if (!pre->rchild)                   /*后继线索*/
     { pre->rtag=1;
       pre->rchild=p;
```

```
    }
    pre=p;
    InThreading(p->rchild);          /*右子树线索化*/
   }
}
```

7、树形结构：

在树形结构中，结点间具有分支层次关系，每一层上的结点只能和上一层中的至多一个结点相关，但可能和下一层的多个结点相关。

一棵树采用孩子兄弟表示法所建立的存储结构与它所对应的二叉树的二叉链表存储结构是完全相同的。

8、树、森林与二叉树的转换

设森林 $F = (T_1, T_2, \cdots, T_n)$;

$T_1 = (\ root, t_{11}, t_{12}, \cdots, t_{1m})$;

二叉树 B =(LBT, Node(root), RBT);

（1）森林转换为二叉树的方法：

若 $F = \Phi$，则 $B = \Phi$;

否则，由 ROOT(T_1) 对应得到 Node(root);

由 ($t_{11}, t_{12}, \cdots, t_{1m}$)对应得到 LBT;

由 ($T_2, T_3 \cdots, T_n$)对应得到 RBT。

（2）二叉树转换为树和森林：

树和森林都可以转换为二叉树，二者不同的是树转换成的二叉树，其根结点无右分支，而森林转换后的二叉树，其根结点有右分支。这一转换过程是可逆的，即可以依据二叉树的根结点有无右分支，将一棵二叉树还原为树或森林。

若 $B = \Phi$, 则 $F = \Phi$;

否则，由 Node(root) 对应得到 ROOT(T_1);

由 LBT 对应得到($t_{11}, t_{12}, \cdots, t_{1m}$);

由 RBT 对应得到($T_2, T_3 \cdots, T_n$)。

由此，树和森林的各种操作均可与二叉树的各种操作相对应。**应当注意的是，**和树对应的二叉树，其**左、右子树**的概念已改变为： **左是孩子，右是兄弟。**

9、树和森林的遍历：可采用对应二叉树的遍历算法来实现的

树	森林	二叉树
先根遍历	**前序遍历**	**前序遍历**
后根遍历	**中序遍历**	**中序遍历**

10、哈夫曼树：最小带权路径长度的二叉树

（1）构造哈夫曼树的基本思想是：

常见错误：在合并中不是选取根结点权值最小的两棵二叉树（包括已合并的和未合并的），而是选取未合并的根结点权值最小的一棵二叉树与已经合并的二叉树合并。

哈夫曼树的特点：具有 n 个叶子结点的哈夫曼树共有 2n－1 个结点。

（2）哈夫曼编码：求哈夫曼编码，实质上就是在已建立的哈夫曼树中，从叶结点开始，沿结点的双亲链域回退到根结点，每回退一步，就走过了哈夫曼树的一个分支，从而得到一位哈夫曼码值，由于一个字符的哈夫曼编码是从根结点到相应叶结点所经过的路径上各分支所组成的 0，1 序列，因此先得到的分支代码为所求编码的低位码，后得到的分支代码为

所求编码的高位码。

利用哈夫曼树可以构造一种不等长的二进制编码，并且构造所得的**哈夫曼编码是一种最优前缀编码**，即使所传电文的总长度最短。

难点释疑

遍历二叉树的应用

（1）二叉树递归算法的设计技巧

对于二叉树，一般递归模型如下：

f(b)=c　　当 b=NULL 时

f(b)=g(f(b->lchild),f(b->lchild),c)　　其他情况

其中，f()为递归函数，g 为非递归函数，c 为常量。

（2）统计一棵给定二叉树中的所有叶子结点数

解析：递归模型如下：

f(b)=0　　若 b=NULL

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

f(b)= f(b->lchild)+ f(b->rchild)　　其他情况

相应算法如下：

```
int LeafNodes (BiTree bt)
{ int num1,mun2;
  if (bt=NULL) return 0;
  else if (bt->lchild=NULL && bt->rchild=NULL)return 1;
       else { num1=LeafNodes(bt->lchild);
              num2=LeafNodes(bt->rchild);
              return (num1+num2);
            }
}
```

（3）求二叉树深（高）度

解析：递归模型如下：

f(b)=0　　若 b=NULL

f(b)= MAX{f(b->lchild),f(b->rchild)}+1　　其他情况

相应算法如下：

```
int BiTreeDepth(BiTree bt)
{int hl, hr;
 if (bt=NULL) return(0);
 else{hl=BiTreeDepth(bt->LChild);/*求左子树的深度 */
      hr=BiTreeDepth(bt->RChild);/*求右子树的深度 */
      return(hl>hr)?(hl+1):(hr+1);)
     }
 }
```

（4）求二叉树中值为 e 的结点层号。

解析：递归模型如下：

f(b,e,h,ih)：h=0　　当 b=NULL

f(b,e,h,ih)：h=ih　　当 b->data==e

f(b,e,h,ih)：f(b->lchild,e,h,ih+1);

若 h=0 则 f(b->rchild,e,h,ih+1)　其他情况

相应算法如下：

```
void NodeLevel(BiTree bt,ElemType e,int &h,int ih)
/*调用本算法时 ih 指出根结点的层次为 1*/
{ if (bt=NULL) h=0;
  else if (bt->data==e) h=ih;
       else{ NodeLevel(bt->lchild,e,h,ih+1);
             if(h=0) NodeLevel(bt->lchild,e,h,ih+1);
           }
}
```

（5）已知二叉树的前序序列、中序序列构造二叉树的算法。

```
BiTree CreateBT1(char *pre，char *in, int n)
/* pre 存放前序序列，in 存放中序序列，n 为 in 中字符个数，本算法执行后返回构造的二叉树的根结点指针*/
{ BiTree bt;
  char *p;
  int k;
  if (n<=0) return NULL;
  bt=(BiTree)malloc(sizeof(BiTNode));
                              /*创建二叉树结点*bt */
 bt->data=*pre;
 for(p=in;p<in+n;p++)
                /*在中序序列中找到等于*pre 的位置 k*/
     if (*p==*pre) break;
 k=p-in;
 bt->lchild= CreateBT1(pre+1,in,k);
 bt->rchild= CreateBT1(pre+k+1,p+1,n-k-1);
 return bt;
}
```

习题：

选择题：

1. 由元素序列（27，16，75，38，51）构造平衡二叉树，则首次出现的最小不平衡子树的根（即离插入结点最近且平衡因子的绝对值为 2 的结点）为（　）

   A. 27　　B. 38　　C. 51　　D. 75

2. 在线索化二叉树中，t 所指结点没有左子树的充要条件是（　）

   A. t->left=NULL　　B. t->ltag=1

   C. t->ltag=1 且 t-left=NULL　　D. 以上都不对

3. 树最适合用来表示（　）

   A. 有序数据元素　　B. 无序数据元素

   C. 元素之间无联系的数据　　D. 元素之间有分支层次关系

4. 某二叉树的前序和后序序列正好相反，则该二叉树一定是（　）的二叉树。

   A. 空或只有一个结点　　B. 高度等于其结点数

   C. 任一结点无左孩子　　D. 任一结点无右孩子

5. 已知一算术表达式的中缀形式为 A+B*C-D/E，后缀形式为 ABC*+DE/-，其前缀形式为（ ）。

A. -A+B*C/DE　　B. -A+B*CD/E　　C. -+*ABC/DE　　D. -+A*BC/DE

6. 中缀表达式 A-(B+C/D)*E 的后缀形式是（ ）

A. AB-C+D/E*　　B. ABC+D/-E*　　C. ABCD/E*+-　　D. ABCD/+E*-

7. 某二叉树的前序遍历序列为 IJKLMNO，中序遍历序列为 JLKINMO，则后序遍历序列为（ ）

A. JLKMNOI　　B. LKNJOMI　　C. LKJNOMI　　D. LKNOJMI

8. 二叉树的先序遍历和中序遍历如下： 先序遍历：EFHIGJK；中序遍历：HFIEJKG 。该二叉树根的右子树的根是（ ）

A. E　　B. F　　C. G　　D. H

9. 对二叉树的结点从 1 开始进行连续编号，要求每个结点的编号大于其左、右孩子的编号，同一结点的左右孩子中，其左孩子的编号小于其右孩子的编号，可采用（ ）次序的遍历实现编号。

A. 先序　　B. 中序　　C. 后序　　D. 从根开始按层次遍历

10. 在一棵具有 n 个结点的二叉树中，所有结点的空子树个数等于（ ）。

A. n　　B. n－1　　C. n＋1　　D. 2*n

11. 有 m 个叶子结点的哈夫曼树所具有的结点数为（ ）

A. m　　B. m+1　　C. 2m　　D. 2m-1

12. 由权值为 9、2、5、7 的四个叶子构造一棵哈夫曼树，该树的带权路径长度为（ ）

A. 23　　B. 37　　C. 44　　D. 46

13. 将有关二叉树的概念推广到三叉树，则一棵有 244 个结点的完全三叉树的高度（ ）

A. 4　　B. 5　　C. 6　　D. 7

14. 一个具有 1025 个结点的二叉树的高 h 为（ ）.

A. 11　　B. 10　　C. 11 至 1025 之间　　D. 10 至 1024 之间

15. 在一棵完全二叉树中，其根的序号为 1，（ ）可判定序号为 p 和 q 的两个结点是否在同一层。

A. $\lfloor \log_2 p \rfloor = \lfloor \log_2 q \rfloor$　　B. $\log_2 p = \log_2 q$

C. $\lfloor \log_2 p \rfloor + 1 = \lfloor \log_2 q \rfloor$　　D. $\lfloor \log_2 p \rfloor = \lfloor \log_2 q \rfloor + 1$

16. 设森林 F 中有三棵树，第一，第二，第三棵树的结点个数分别为 N1，N2 和 N3。与森林 F 对应的二叉树根结点的右子树上的结点个数是（ ）。

A. N1　　B. N1+N2　　C. N3　　D. N2+N3

应用题

1、已知二叉树采用二叉链表方式存放，要求返回二叉树 T 的后序序列中的第一个结点的指针，是否可不用递归且不用栈来完成？请简述原因。

答：可以。

原因：后序遍历的顺序是“左子树—右子树—根结点”。因此，二叉树最左下的叶子结点是遍历的第一个结点。下面的语句段说明了这一过程（设 p 是二叉树根结点的指针）。

```
if (p!=null)
  {
    while (p->lchild!=null || p->rchild!=null)
     {
```

```
            while(p->lchild!=null) p=p->lchild;
            if(p->rchild!=null) p=p->rchild;
          }
       }
       return(p);                    //返回后序序列第一个结点的指针
```

算法设计题：

1. 请利用队列的基本操作写出判定一棵二叉树是否为完全二叉树的算法。要求以二叉链表作为二叉树的存储结构。函数原型为：

```
int IsFull_Bitree(Bitree T);
```

解：

```
int IsFull_Bitree(Bitree T)
{   InitQueue(Q);
    flag=0;
    EnQueue(Q,T);
    while(!QueueEmpty(Q))
    {DeQueue(Q,p);
     if(!p)
        flag=1;
     else if(flag)  return 0;
        else
         {  EnQueue(Q,p->lchild);
            EnQueue(Q,p->lchild);
         }
    }
    return 1;
}
```

2. 请利用栈的基本操作写出复制一棵二叉树的非递归算法。要求以二叉链表作为二叉树的存储结构。设栈已定义：InitStack(S), Push(S,e), StackEmpty(S), Gettop(S,e), Pop(S,e)分别为栈初始化，入栈，判断栈空，获得栈顶元素，出栈等操作。函数原型如下：

```
void Bitree_Copy(Bitree T,Bitree &U);
```

解：

```
void Bitree_Copy(Bitree T,Bitree &U)
{   InitStack(S1);
    InitStack(S2);
    Push (S1,T) ;
    U=(BTNode*)malloc(sizeof(BTNode));
    U->data=T->data;
    q=U;
    Push(S2,U);
    while(!StackEmpty(S1))
    { while(Gettop(S1,p)&&p)
        {   q->lchild=(BTNode*)malloc(sizeof(BTNode));
            q=q->lchild;
            q->data=p->data;
```

```
            Push(S1,p->lchild);
            Push(S2,q);
        }
        Pop(S1,p);
        Pop(S2,q);
        if(!StackEmpty(S1))
        {   Pop(S1,p);  Pop(S2,q);
            q->rchild=(BTNode*)malloc(sizeof(BTNode));
            q=q->rchild;
            q->data=p->data;
            Push(S1,p->rchild);
            Push(S2,q);
        }
    }
}
```

第 5 章　图

重点归纳

1、**图形结构：**任意两个结点之间都可能相关，即结点之间的邻接关系可以是任意的。
无向完全图：在一个含有 n 个顶点的无向完全图中，有 n(n-1)/2 条边。
有向完全图：在一个含有 n 个顶点的有向完全图中，有 n(n-1)条边。
顶点的度：是指依附于某顶点 v 的边数，通常记为 TD (v)。

对于具有 n 个顶点、e 条边的图，顶点 vi 的度 TD (vi)与顶点的个数以及边的数目满足关系：$e=\frac{1}{2}\sum_{i=1}^{n}TD(v_i)$

边的权、网：与边有关的数据信息称为权。边上带权的图称为网图或网络。
路径、路径长度：顶点 v_p 到顶点 v_q 之间的路径是指**顶点序列** $v_p, v_{i1}, v_{i2}, \cdots, v_{im}, v_q$。路径上边的数目称为路径长度。
连通分量：无向图的极大连通子图。
强连通分量：有向图的极大强连通子图。
生成树：所谓连通图 G 的生成树，是 G 的包含其全部 n 个顶点的一个极小连通子图。它必定包含且仅包含 G 的 n-1 条边。
2、图的存储结构：
（1）邻接矩阵：就是用一维数组存储图中顶点的信息，用矩阵表示图中各顶点之间的邻接关系。

图的邻接矩阵存储方法具有以下特点：
① 无向图的邻接矩阵一定是一个对称矩阵。因此，在具体存放邻接矩阵时只需存放上（或下）三角矩阵的元素即可。
② 对于无向图，邻接矩阵的第 i 行（或第 i 列）非零元素（或非∞元素）的个数正好是第 i 个顶点的度 TD(vi)。
③ 对于有向图，邻接矩阵的第 i 行（或第 i 列）非零元素（或非∞元素）的个数正好是第 i 个顶点的出度 OD(vi)（或入度 ID(vi)）。

④用邻接矩阵方法存储图，很容易确定图中任意两个顶点之间是否有边相连；但是，要确定图中有多少条边，则必须按行、按列对每个元素进行检测，所花费的时间代价很大。这是用邻接矩阵存储图的局限性。

（2）邻接表：是图的一种顺序存储与链式存储结合的存储方法，类似于树的孩子链表表示法。就是对于图 G 中的每个顶点 v_i，将所有邻接于 v_i 的顶点 v_j 链成一个单链表，这个单链表就称为顶点 v_i 的邻接表，再将所有点的邻接表表头放到数组中，就构成了图的邻接表。

图的邻接表存储方法具有以下特点：

①若无向图中有 n 个顶点、e 条边，则它的邻接表需 n 个头结点和 2e 个表结点。稀疏图用邻接表表示比邻接矩阵节省存储空间，当和边相关的信息较多时更是如此。

②在无向图的邻接表中，顶点 v_i 的度恰为第 i 个链表中的结点数；在有向图中，第 i 个链表中的结点个数只是顶点 vi 的出度，为求入度，必须遍历整个邻接表。在所有链表中其邻接点域的值为 i 的结点的个数是顶点 v_i 的入度。

有时，为了便于确定顶点的入度或以顶点 v_i 为头的弧，可以建立一个有向图的逆邻接表，即对每个顶点 v_i 建立一个链接以 v_i 为头的弧的链表。

在建立邻接表或逆邻接表时，若输入的顶点信息即为顶点的编号，则建立邻接表的复杂度为 O(n+e)，否则，需要通过查找才能得到顶点在图中位置，则时间复杂度为 O(n·e)。

③在邻接表上容易找到任一顶点的第一个邻接点和下一个邻接点，但要判定任意两个顶点（v_i 和 v_j）之间是否有边或弧相连，则需搜索第 i 个或第 j 个链表，因此，不及邻接矩阵方便。

3、图的遍历：通常有深度优先搜索和广度优先搜索两种方式。

为了保证图中的各顶点在遍历过程中访问且仅访问一次，需要为每个顶点设一个访问标志，因此我们为图设置一个访问标志数组 visited[n]，用于标示图中每个顶点是否被访问过。

（1）深度优先搜索：类似于树的先根遍历，是树的先根遍历的推广。

●从图的某一点 v 出发，递归地进行深度优先遍历算法描述：

```
void DFSTraverse(Graph G)
{for (v=0; v<G.vexnum; ++v)
     visited[v] = FALSE;              /*访问标志数组初始化*/
  for (v=0; v<G.vexnum; ++v)
     if (!visited[v])  DFS(G, v);     /*对尚未访问的顶点调用 DFS*/
}
void DFS(Graph G,int v )              /*从第 v 个顶点出发递归地深度优先遍历图 G*/
{ visited[v]=TRUE;Visit(v);           /*访问第 v 个顶点*/
  for(w=FisrtAdjVex(G,v);w>=0; w=NextAdjVex(G,v,w))
    if (!visited[w]) DFS(G,w);        /*对 v 的尚未访问的邻接顶点 w 递归调用 DFS*/
}
```

●以邻接表为存储结构的整个图 G 进行深度优先遍历实现的 C 语言描述。

```
void DFSTraverseAL(ALGraph G)         /*深度优先遍历以邻接表存储的图 G*/
{ int i;
  for (i=0;i<G.vexnum;i++)
      visited[i]=FALSE;               /*标志向量初始化*/
  for (i=0;i<G.vexnum;i++)
      if (!visited[i]) DFSAL(G,i);    /*vi 未访问过，从 vi 开始 DFS 搜索*/
}
```

```
void DFSAL(ALGraph G,int i)        /*以 vi 为出发点对邻接表存储的图 G 进行 DFS 搜索*/
{ ArcNode *p;
  Visit(G.adjlist[i]);                  /*访问顶点 vi*/
  visited[i]=TRUE;                      /*标记 vi 已访问*/
  p=G.adjlist[i].firstarc;              /*取 vi 边表的头指针*/
  while(p)                            /*依次搜索 vi 的邻接点 vj，j=p->adjvex*/
   { if (!visited[p->adjvex])    /*若 vj 尚未访问，则以 vj 为出发点向纵深搜索*/
     DFSAL(G,p->adjvex);
     p=p->nextarc;              /*找 vi 的下一个邻接点*/
   }
 }
```

遍历图的过程实质上是对每个顶点查找其邻接点的过程，其耗费的时间则取决于所采用的存储结构。当以邻接矩阵为图的存储结构时，查找每个顶点的邻接点所需时间为 $O(n^2)$ ，其中 n 为图中顶点数。而当以邻接表作图的存储结构时，找邻接点所需时间为 O(e)，其中 e 为无向图中边的数或有向图中弧的数。由此，当以邻接表作存储结构时，深度优先搜索遍历图的时间复杂度为 O(n+e) 。

（2）广度优先搜索：类似于树的按层次遍历的过程。

●从图的某一点 v 出发，进行广度优先遍历算法描述：

```
void BFSTraverse (MGraph G)          /*按广度优先非递归遍历图 G，使用辅助队列 Q*/
{for (v=0; v<G.vexnum; ++v)
     visited[i] = FALSE;              /*访问标志数组初始化*/
 for (v=0; v<G.vexnum; ++v)
     if (!visited[v])  BFS(G, v);    /*对尚未访问的顶点调用 BFS*/
}
void BFS (Graph G,int v)
{InitQueue(Q);                      /*置空的辅助队列 Q*/
 visited[v]=TRUE; Visit(v);                  /*访问 v*/
 EnQueue(Q,v);                              /*v 入队列*/
 while (!QueueEmpty(Q))
  {DeQueue(Q,u);                 /*队头元素出队并置为 u*/
   for(w=FistAdjVex(G,u); w>=0; w=NextAdjVex(G,u,w))
   if(!visited[w])
     {visited[w]=TRUE; Visit(w);
      EnQueue(Q,w); /*u 尚未访问的邻接顶点 w 入队列 Q*/
      }
   }
}
```

●以邻接矩阵为存储结构的整个图 G 进行广度优先遍历实现的 C 语言描述。

```
void BFSTraverseAL(MGraph G)       /*广度优先遍历以邻接矩阵存储的图 G*/
{int i;
 for (i=0;i<G.vexnum;i++)
      visited[i]=FALSE;             /*标志向量初始化*/
   for (i=0;i<G.vexnum;i++)
     if (!visited[i]) BFSM(G,i); /* vi 未访问过，从 vi 开始 BFS 搜索*/
```

```
}
void BFSM(MGraph G,int k)  /*以 vi 为出发点，对邻接矩阵存储的图 G 进行 BFS 搜索*/
{int i,j;
 sqQueue Q;
 InitQueue(Q);
 Visit(G.vexs[k]);                    /*访问原点 Vk*/
 visited[k]=TRUE;
 EnQueue(Q,k);                        /*原点 Vk 入队列*/
 while (!QueueEmpty(Q))
  {i=DeQueue(Q);                      /*Vi 出队列*/
   for (j=0;j<G.vexnum;j++)  /*依次搜索 Vi 的邻接点 Vj*/
   if(G.edges[i][j]==1 && !visited[j]) /*若 Vj 未访问*/
     {Visit (G.vexs[j]);               /*访问 Vj */
      visited[j]=TRUE;
      EnQueue(Q,j);              /*访问过的 Vj 入队列*/
      }
  }
}
```

广度优先搜索遍历图的过程实质是通过边或弧找邻接点的过程，其时间复杂度和深度优先搜索遍历相同，两者不同之处仅仅在于对顶点访问的顺序不同。

4、图的重要应用

（1）最小生成树

●Prim **算法：**取图中任意一个顶点 v 作为生成树的根，之后往生成树上添加新的顶点 w。在添加的顶点 w 和已经在生成树上的顶点 v 之间必定存在一条边，并且该边的权值在所有连通顶点 v 和 w 之间的边中取值最小。之后继续往生成树上添加顶点，直至生成树上含有 n-1 个顶点为止。

●Kruskal **算法：**先构造一个只含 n 个顶点的子图 SG，然后从权值最小的边开始，若它的添加不使 SG 中产生回路，则在 SG 上加上这条边，如此重复，直至加上 n-1 条边为止。

算法名	Prim **算法**	Kruskal **算法**
时间复杂度	$O(n^2)$	O(eloge)
适用范围	边稠密的网	边稀疏的网

（2）最短路径

●从一个源点到其它各点的最短路径。

Dijkstra 算法：依最短路径的长度递增的次序求得各条路径。

设置辅助数组 Dist，其中每个分量 Dist[k] 表示当前所求得的从源点到其余各顶点 k 的最短路径。

Dist[k] = <源点到顶点 k 的弧上的权值>

或者 = <源点到其它顶点的路径长度>+ <其它顶点到顶点 k 的弧上的权值>

时间复杂度是 $O(n^2)$

●每一对顶点之间的最短路径。

Floyd 算法：从 V_i 到 V_j 的所有可能存在的路径中，选出一条长度最短的路径。时间复杂度是 $O(n^3)$。

（3）拓扑排序

整个算法的时间复杂度 为O（e+n）。**拓扑排序的序列可能不唯一。**

当有向图中无环时，也可用深度优先遍历的方法进行拓扑排序，按 DFS 算法的先后次序记录下的顶点序列为逆向的拓扑有序序列。

（4）关键路径

为了在 AOE 网中找出关键路径，需要定义几个参量，并且说明其计算方法。

假设第 i 条弧为<j,k>，dut(<j, k>)为弧<j,k>上的权值。

①事件的最早发生时间 ve[k]=从源点到顶点 k 的最长路径长度。

ve(源点) = 0;

ve(k) = Max{ve(j) + dut(<j, k>)}

②事件的最迟发生时间 vl[j]=从顶点 j 到汇点的最短路径长度。

vl(汇点) = ve(汇点);

vl(j) = Min{vl(k) - dut(<j, k>)}

③活动 i 的最早开始时间 ee[i]=ve[j]

④活动 i 的最晚开始时间 el[i]=vl[k]-dut(<j,k>)，

el[i]=ee[i]的活动就是关键活动，关键活动所在的路径就是关键路径。

整个工程完成的时间为：从有向图的**源点**到**汇点**的最长路径。

习题：

选择题：

1. 下面关于图的存储的叙述中，哪一个是正确的。 （ A ）

A. 用邻接矩阵法存储图,占用的存储空间数只与图中结点个数有关,而与边数无关

B. 用邻接矩阵法存储图,占用的存储空间数只与图中边数有关,而与结点个数无关

C. 用邻接表法存储图，占用的存储空间数只与图中结点个数有关，而与边数无关

D. 用邻接表法存储图，占用的存储空间数只与图中边数有关，而与结点个数无关

2. 在无向图中定义顶点 V_i 与 V_j 之间的路径为从 V_i 到达 V_j 的一个 （ A ）。

A. 顶点序列　　B. 边序列　　C. 权值总和　　D. 边的条数

3. 采用邻接表存储的图的广度优先遍历算法类似于树的 （ D ）。

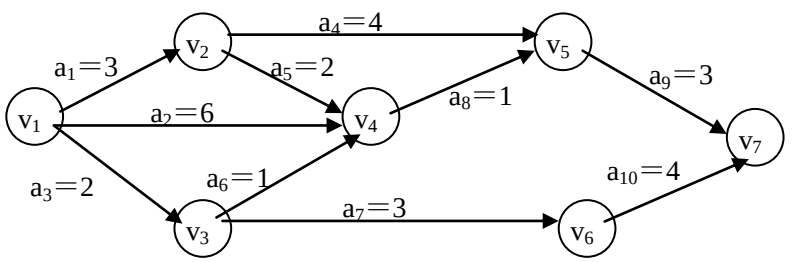
A. 中根遍历　　B. 先根遍历　　C. 后根遍历　　D. 按层次遍历

4. 在图采用邻接表存储时，求最小生成树的 Prim 算法的时间复杂度为（ B ）。

A. O(n)　　B. O(n+e)　　C. $O(n^2)$　　D. $O(n^3)$

应用题：

1、已知 AOE 网中顶点 $v_1, v_2, v_3, ...v_7$ 分别表示 7 个时间，有向线段 $a_1, a_2, a_3, ...a_{10}$ 分别表示 10 个活动，线段旁的数值表示每个活动花费的天数，如下图所示。请用顶点序列表示出关键路径，给出关键活动。



答：

事件	v_1	v_2	v_3	v_4	v_5	v_6	v_7
最早发生时间	0	3	2	6	7	5	10
最晚发生时间	0	3	3	6	7	6	10

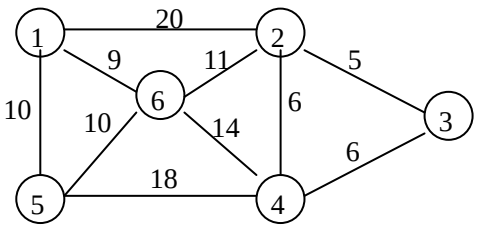
活动	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}
最早发生时间	0	0	0	3	3	2	2	6	7	5
最晚开始时间	0	0	1	0	1	3	1	0	0	1

关键路径：v_1 v_2 v_5 v_7

v_1 v_4 v_5 v_7

关键活动：a_1 a_2 a_4 a_8 a_9

2、已知一个无向图如下图所示，要求分别用 Prim 和 Kruskal 算法生成最小树（假设以①为起点），试画出构造过程，并说明方法的时间复杂度。



第 6 章　查找

重点归纳

1、**静态查找表：**仅作查询和检索操作的查找表。

（1）顺序查找

查找成功时，等概率情况下，平均查找长度为：

$$ASL_{ss}=\frac{1}{n}\sum_{i=1}^{n}(n-i+1)=\frac{n+1}{2}$$

查找不成功时，关键码的比较次数总是 n+1 次。

顺序查找算法的时间复杂度为 O(n)。缺点是当 n 很大时，平均查找长度较大，效率低；优点是表的存储结构通常为顺序结构，也可为链式结构。另外，对于线性链表，只能进行顺序查找。

（2）折半查找

适用条件：待查找的列表必须是按关键字大小有序排列的顺序表。

```
int Search_Bin (SSTable ST, KeyType key )
{low = 1;  high = ST.length; /*置区间初值*/
 while(low <= high)
  {mid=(low + high)/2;
   if (key == ST.elem[mid].key  )
      return  mid;            /*找到待查元素*/
   else if(key<ST.elem[mid].key) )
```

```
            high = mid - 1; /*继续在前半区间进行查找*/
        else low = mid + 1;/*继续在后半区间进行查找*/
    }
    return 0;                    /*顺序表中不存在待查元素*/
}
```

对表中每个数据元素的查找过程，可用二叉树来描述，称这个描述查找过程的二叉树为判定树。

等概率情况下，折半查找的平均查找长度为：

$$ASL_{bs}=\frac{1}{n}\sum_{i=1}^{n}C_i=\frac{1}{n}\left[\sum_{j=1}^{h}j\times 2^{j-1}\right]=\frac{n+1}{n}\log_2(n+1)-1$$

$$ASL_{bs}\approx\log_2(n+1)-1$$

折半查找的时间效率为 $O(\log_2 n)$。

顺序查找与折半查找对比：

	顺序查找	折半查找
表的特性	无序	有序
存储结构	顺序 或 链式	顺序
插入、删除操作	易于进行	需移动元素
ASL 的值	大	小

几种查找表的特性：

	查找	插入	删除
无序顺序表	O(n)	O(1)	O(n)
无序线性链表	O(n)	O(1)	O(1)
有序顺序表	O(logn)	O(n)	O(n)
有序线性链表	O(n)	O(1)	O(1)

可得如下结论：

①从查找性能看，最好情况能达 O(logn)，此时要求表有序并且为顺序存储结构；

②从插入和删除的性能看，最好情况能达 O(1)，此时要求存储结构是链表。

（3）索引查找

前提：①查找表要求顺序存储；②查找表分成 n 块，当 i>j 时，第 i 块中的最小元素大于第 j 块中的最大元素。

过程：①首先确定所要查找关键字在哪一块中。②在所确定的块中用顺序查找查找关键字。

2、**动态查找表：**有时在查询之后，还需要将“查询”结果为“不在查找表中”的数据元素插入到查找表中；或者，从查找表中删除其“查询”结果为“在查找表中”的数据元素。

（1）①二叉排序树定义：或者是一棵空树；或者是具有下列性质的二叉树：

●若左子树不空，则左子树上所有结点的值均小于根结点的值；若右子树不空，则右子树上所有结点的值均大于根结点的值。

●左右子树也都是二叉排序树。

通常，取二叉链表作为二叉排序树的存储结构。

②**二叉排序树查找过程：**是一个递归过程。

●递归算法：

BSTree SearchBST(BiTree bt, KeyType key)/*在根指针 bt 所指二叉排序树中，递归查找某关键字等于 key 的元素，若查找成功，则返回指向该元素结点指针，否则返回空指针 */

```
{if(!bt) return NULL;
 else if(bt->key==key) return bt;    /*查找成功*/
else if(key＜bt->key)
          return SearchBST(bt->lchild,key);   /*在左子树中继续查找*/
     else return SearchBST(bt->rchild,  key);  /*在右子树中继续查找*/
}
```

●非递归算法

```
BSTree  SearchBST(BiTree bt,  KeyType key)
{BiTree q;
 q=bt;
 while(q)
 {if (q->key==k)  return q;                /*查找成功*/
  if (key＜q->data.key)q=q->lchild;/*在左子树中查找*/
  else q=q->rchild;                 /*在右子树中查找*/
  }
  return NULL;  /*查找失败*/
 }
```

由值相同的 n 个关键字，构造所得的不同形态的各棵二叉排序树的平均查找长度的值不同，甚至可能差别很大。

③二叉排序树插入操作和构造一棵二叉排序树：构造一棵二叉排序树则是逐个插入结点的过程。

```
void InsertBST(BiTree *bt,  KeyType key)
/*若在二叉排序树中不存在关键字等于 key 的元素，  插入该元素 */
{BiTree s;
 if (*bt==NULL) /*递归结束条件*/
  {s=(BiTree)malloc(sizeof(BiTNode));
   s-> key=key;
   s->lchild=NULL;
   s->rchild=NULL;
   *bt=s;
   }
else if(key＜(*bt)->key)
  InsertBST(&((*bt)->lchild),key); /*将 s 插入左子树*/
  else  if(key＞(*bt)->key)
  InsertBST(&((*bt)->rchild),key); /*将 s 插入右子树*/
}
```

（2）平衡二叉树(AVL 树)定义：

或者是一棵空树，或者是具有下列性质的二叉排序树：它的左子树和右子树都是平衡二叉树，且左子树和右子树高度之差的绝对值不超过 1。

在插入过程中，采用平衡旋转技术：左单旋转；右单旋转；先左后右双向旋转；先右后左双向旋转。

在平衡树上进行查找的过程和二叉排序树相同，因此，查找过程中和给定值进行比较的关键字的个数不超过平衡 树的深度，和 $\log_2 n$ 相当。

（3）B-树及其查找

B-树是一种平衡的多路查找树，它在文件系统中很有用。

①定义：一棵 m 阶的 B-树，或者为空树，或为满足下列特性的 m 叉树：

●树中每个结点至多有 m 棵子树；

●若根结点不是叶子结点，则至少有两棵子树；

●除根结点之外的所有非终端结点至少有⌈m/2⌉ 棵子树；

●所有的非终端结点中包含以下信息数据：（n，A_0，K_1，A_1，K_2，…，K_n，A_n）

其中：K_i（i=1,2,…,n）为关键码，且 $K_i<K_{i+1}$，A_i 为指向子树根结点的指针(i=0,1,…,n)，且指针 A_{i-1} 所指子树中所有结点的关键码均小于 K_i (i=1,2,…,n)，A_n 所指子树中所有结点的关键码均大于 K_n，⌈m/2⌉ -1≤n≤m -1 ，n 为关键码的个数。

●所有的叶子结点都出现在同一层次上，并且不带信息（可以看作是外部结点或查找失败的结点，实际上这些结点不存在，指向这些结点的指针为空）。

②B-树的查找类似二叉排序树的查找，所不同的是 B-树每个结点上是多关键码的有序表，在到达某个结点时，先在有序表中查找，若找到，则查找成功；否则，到按照对应的指针信息指向的子树中去查找，当到达叶子结点时，则说明树中没有对应的关键码，查找失败。

在含 N 个关键字的 B-树上进行一次查找，需访问的结点个数不超过 $\log_{\lceil m/2\rceil}((N+1)/2)+1$。

3、散列表：基本思想是首先在元素的关键字 key 和元素的存储位置 p 之间建立一个对应关系 H，使得 p=H(key)，H 称为散列函数。创建散列表时，把关键字为 key 的元素直接存入地址为 H(key)的单元；以后当查找关键字为 key 的元素时，再利用散列函数计算出该元素的存储位置 p=H(key)，从而达到按关键字直接存取元素的目的。

散列方法需要解决以下两个问题：

问题一：构造好的散列函数：

●所选函数尽可能简单，以便提高转换速度；

●所选函数对关键码计算出的地址，应在散列地址集中大致均匀分布，以减少空间浪费；

●总的原则是使产生冲突的可能性降到尽可能地小。

问题二：制定解决冲突的方案。

（1）构造散列函数

①直接定址法：H(key) = a × key + b

适合于：地址集合的大小 = = 关键字集合的大小

②数字分析法：分析关键字集中的全体，并从中提取分布均匀的若干位或它们的组合作为地址。

适合于：能预先估计出全体关键字的每一位上各种数字出现的频度。

③平方取中法：以关键字的平方值的中间几位作为存储地址。

适合于：关键字中的每一位都有某些数字重复出现频度很高的现象。

④折叠法：将关键字分割成若干部分，然后取它们的叠加和为散列地址。

适合于：关键字的数字位数特别多。

⑤除留余数法：H(key) = key MOD p ，其中 p≤m (表长)并且 p 应为不大于 m 的素数

⑥随机数法：H(key) = Random(key)

适合于：对长度不等的关键字构造散列函数。

（2）解决冲突的方法

①开放定址法：当关键字 key 的散列地址 p= H（key）出现冲突时，以 p 为基础，产生另一个散列地址 p_1，如果 p_1 仍然冲突，再以 p 为基础，产生另一个散列地址 p_2，……，直到找出一个不冲突的散列地址 p_i，将相应元素存入其中。

$$H_i=(H(key)+d_i)\%m, i=1,2,\ldots,n$$

主要有以下三种：

●线性探测再散列　d_i=1，2，3，…:，　m-1

特点：冲突发生时，顺序查看表中下一单元，直到找出一个空单元或查遍全表。

●二次探测再散列　$d_i=\pm1^2$，$\pm2^2$，…，$\pm k^2$　　(k≤m/2)

特点：冲突发生时，在表的左右进行跳跃式探测，比较灵活。

●伪随机探测再散列　d_i=伪随机数序列。

②链地址法：将所有散列地址为 i 的元素构成一个称为同义词链的单链表，并将单链表的头指针存在散列表的第 i 个单元中，因而查找、插入和删除主要在同义词链中进行。 链地址法适用于经常进行插入和删除的情况。

③建立一个公共溢出区：将散列表分为基本表和溢出表两部分，凡是与基本表发生冲突的元素一律填入溢出表。

（3）散列表的查找过程

与散列表的创建过程是一致的。 当查找关键字为 key 的元素时，首先计算 p_0=H（key）。如果单元 p_0 为空，则所查元素不存在；如果单元 p_0 中元素的关键字为 key，则找到所查元素；否则重复下述解决冲突的过程：按解决冲突的方法，找出下一个散列地址 p_i， 如果单元 p_i 为空，则所查元素不存在；如果单元 p_i 中元素的关键字为 key，则找到所查元素。

几种不同处理冲突方法的平均查找长度：

	查找成功	查找失败
线性探测再散列	$S_{n1}=\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$	$U_{n1}=\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$
伪随机探测再散列、 二次探测	$S_{nr}=-\frac{1}{\alpha}1n(1-\alpha)$	$U_{nr}=\frac{1}{1-\alpha}$
链地址法	$S_{nc}=1+\frac{\alpha}{2}$	$U_{nc}=\alpha+e^{-\alpha}$

手工计算等概率情况下查找成功的平均查找长度公式：

$$ASL_{succ}=\frac{1}{\text{表中置入元素个数}n}\sum_{i=1}^{n}C_i$$

其中 C_i 为置入每个元素时所需的比较次数。

手工计算等概率情况下查找不成功的平均查找长度公式：

$$ASL_{unsucc}=\frac{1}{\text{哈希函数取值个数}}\sum_{i=1}^{r}C_i$$

其中 C_i 为函数取值为 i 时确定查找不成功时的比较次数。

习题：

选择题：

1. 对长度为 n 的有序单链表，若搜索每个元素的概率相等，则顺序搜索到表中任一元素的平均搜索长度为（ ）。

   A．n/2　　B．(n+1) /2　　C．(n－1) /2　　D．n/4

2. 有一个长度为 12 的有序表，按二分查找法对该表进行查找，在表内各元素等概率情况下，查找成功所需的平均比较次数为（　　）

A. 37/12　　B. 35/12　　C. 39/12　　D. 43/12

3. 有一个有序表为{1,3,9,12,32,41,45,62,75,77,82,95,100}，当二分查找值为 82 的结点时，（　　）次比较后查找成功。

A. 1　　B. 2　　C. 4　　D. 8

4. 在（ ）存储结构中，数据结构中元素的存储地址与其关键字之间存在某种映射关系。

A．树形存储结构　　B. 链式存储结构　C．索引存储结构　D. 散列存储结构

5. 每个存储结点只含有一个数据元素，存储结点存放在连续的存储空间，另外有一组指明存储位置的表，该存储方式是（ ）存储方式

A．顺序　　B．链接　C．索引　　D．散列

6. 已知一个线性表（38，25，74，63，52，48），假定采用散列函数 h(key)=key%7 计算散列地址，并散列存储在散列表 A[0…6]中，若采用线性探测方法解决冲突，则在该散列表上进行等概率成功查找的平均查找长度为（ ）

A．1.5　　B. 1.7　　C. 2.0　　D. 2.3

7. 设分块/索引查找中索引表长度为 a，每块的平均长度 b，则分块/索引查找的平均时间复杂性为_____。如果对索引表采取折半查找，则分块/索引查找的平均时间复杂性为_____。

应用题：

1、采用哈希函数 H(k) = 3*k mod 13 并用线性探测开放地址法处理冲突，在数列地址空间[0..12]中对关键字序列 22，41，53，46，30，13，1，67，51；

（1）构造哈希表（画示意图）；

（2）装填因子；

（3）等概率下成功的平均查找长度；

（4）等概率情况下查找失败的平均查找长度

答：（1）

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

（2）装填因子 = 9/13 ≈0.7

（3）ASL_{succ}=11/9

（4）ASL_{unsucc}=(3+2+1+3+2+1+4+3+2+1+2+1+4)/13=29/13

2、已知一组关键字为（26，36，41，38，44，15，68，12，6，51，25），用链地址法解决冲突。假设装填因子 α=0.75，哈希函数的形式为 H(K) = K MOD P，回答下列问题：

（1）构造哈希函数

（2）画出哈希表

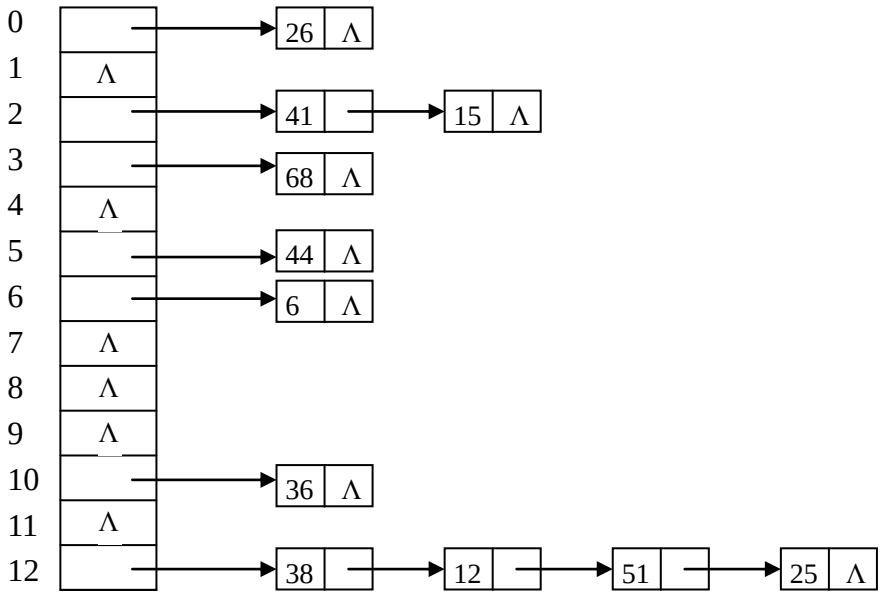
（3）计算出等概率情况下查找成功的平均查找长度。

（4）计算出等概率情况下查找失败的平均查找长度。

答：由 α=0.75，得表长 m=11/0.75=15.

（1）哈希函数 H(K) = K/13

（2）哈希表



（3）等概率情况下查找成功的平均查找长度：ASL=18/11

（4）等概率情况下查找失败的平均查找长度：ASL=11/13

第 7 章　排序

重点归纳

1、插入排序：是在一个已排好序的记录子集的基础上，每一步将下一个待排序的记录有序地插入到已排好序的记录子集中，直到将所有待排记录全部插入为止。 包括**直接插入排序**、**希尔排序**、折半插入排序等

2、交换排序：主要是通过两两比较待排记录的关键码，若发生与排序要求相逆，则交换之。包括冒泡排序和**快速排序**

3、选择排序主要是每一趟从待排序列中选取一个关键码最小的记录，也即第一趟从 n 个记录中选取关键码最小的记录，第二趟从剩下的 n-1 个记录中选取关键码最小的记录，直到整个序列的记录选完。这样，由选取记录的顺序，便得到按关键码有序的序列。包括简单选择排序和**堆排序**

4、**二路归并排序**的基本操作是将两个有序表合并为一个有序表。

5、**基数排序**是一种借助于多关键码排序的思想，是将单关键码按基数分成“多关键码”进行排序的方法。

习题：

选择题

1. 下列几种排序方法中，要求内存量最大的是（ ）

   A．插入排序　　B．快速排序　　C．归并排序　　D．选择排序

2. 以比较为基础的排序算法在最坏情况下的计算时间下界为（ ）

A. O(n)　　B. $O(n^2)$　　C. O(logn)　　D. O(nlogn)

3. （）从二叉树的任一结点出发到根的路径上，所经过的结点序列必按其关键字降序排列。
A. 二叉排序树　　B. 大顶堆　　C. 小顶堆　　D. 平衡二叉树

4. 若对 27 个元素只进行三趟多路归并排序，则选取的归并路数为（ ）。
A. 2　　B. 3　　C. 4　　D. 5

5. 快速排序最易发挥其长处的情况是（ ）
A. 被排序的数据中含有多个相同排序码　B. 被排序的数据已基本有序
C. 被排序的数据完全无序　　D. 被排序的数据中的最大值和最小值相差悬殊

6. 对于序列(49,38,65,97,76,13,27,50)按由小到大进行排序，（ ）是初始步长 d=4 的希尔排序法第一趟的结果。
A. 49,76,65,13,27,50,97,38　　B. 13,27,38,49,50,65,76,97
C. 97,76,65,50,49,38,27,13　　D. 49,13,27,50,76,38,65,97

7. 若一组纪录的排序码为（46，79，56，38，40，84），则利用堆排序的方法建立的初始堆为（ ）
A. 79，46，56，38，40，84　　B. 84，79，56，38，40，46
C. 40，38，46，56，79，84　　D. 40，38，46，84，56，79

8. 若对序列（tang,deng,an,wang,shi,bai,fang,liu）采用选择排序法按字典顺序进行排序，下面给出的四个序列中，（ ）是第三趟的结果
A. an,bai,deng,wang,tang,fang,shi,liu
B. an,bai,deng,wang,shi,tang,fang,liu
C. an,bai,deng,wang,shi,fang,tang,liu
D. an,bai,deng,wang,shi,liu,tang,fang

9. 下列序列中，满足堆定义的是（ ）
A.（100，86，48，73，35，39，42，57，66，21）
B.（12，70，33，65，24，56，48，92，86，33）
C.（103，97，56，38，66，23，42，12，30，52，6，26）
D.（5，56，20，23，40，38，29，61，36，76，28，100）

算法设计题：

1. 可按如下所述实现归并排序：假设序列中有 k 个长度为小于等于 n 的有序子序列。利用过程 merge 对它们进行两两归并，得到$\lceil k/2 \rceil$个长度小于等于 2n 的有序子序列（$\lceil\ \rceil$表示取整），称为一趟归并排序。反复调用一趟归并排序过程，使有序子序列的长度自 n=1 开始成倍地增加，直至使整个序列成为一个有序序列。试采用链表存储结构实现上述归并排序的非递归算法。函数原型如下：

```
void Linked_Mergesort(LinkedList &L);  //链表结构上的归并排序非递归算法
void Linked_Merge(LinkedList &L, LNode *p, Lnode *e1,Lnode *e2);
/*对链表上的子序列进行归并，第一个子序列是从 p->next 到 e1,第二个是从 e1->next 到 e2*/
```

答：

```
void Linked_Mergesort(LinkedList &L); //链表结构上的归并排序非递归算法
{   for(l=1;l<L.length;l*=2)
        for(p=L->next,e2=p; p->next; p=e2)
        {   for(i=1,q=p; i<=l && q->next; i++,q=q->next)
```

```
                e1=q;
            for(i=1; i<=l && q->next; i++,q=q->next)
                e2=q;  //求两个待归并子序列的尾指针
            if(e1!=e2)
                Linked_Merge(L,p,e1,e2);
        }
}
void Linked_Merge(LinkedList &L, LNode *p, Lnode *e1,Lnode *e2);
/*对链表上的子序列进行归并，第一个子序列是从 p->next 到 e1,第二个是从 e1->next 到
e2*/
{   q=p->next;  //q 和 r 为两个子序列的起始位置
    r=e1->next;
    while(q!=e1->next && r!=e2->next)
    { if(q->data < r->data)
        { p->next=q;
          p=q;
          q=q->next;
        }
        else
        { p->next=r;
          p=r;
          r=r->next;
        }
    }
    while(q!=e1->next)
      { p->next=q;
        p=q;
        q=q->next;
        }
    while(r!=e2->next)
      { p->next=r;
        p=r;
        r=r->next;
      }
}
```

2．快速排序算法中，如何选取一个界值（又称为轴元素），影响着快速排序的效率，而且界值也并不一定是被排序序列中的一个元素。例如，我们可以用被排序序列中所有元素的平均值作为界值。编写算法实现以平均值为界值的快速排序方法。

解：题目解析：保存划分的第一个元素。以平均值作为枢轴，进行普通的快速排序，最后枢轴的位置存入已保存的第一个元素，若此关键字小于平均值，则它属于左半部，否则属于右半部。

```
int partition (RecType r[],int l,h)
{ int i=l,j=h,avg=0;
  for(;i<=h;i++)  avg+=R[i].key;
```

```
  i=l;
  avg=avg/(h-l+1);
  while (i<j)
    { while (i<j &&R[j].key>=avg) j--;
      if (i<j) R[i]=R[j];
      while (i<j &&R[i].key<=avg) i++;
      if (i<j) R[j]=R[i];
     }
  if(R[i].key<=avg) return  i;
  else return  i-1;
 }
void quicksort (RecType R[],int S,T);
{if (S<T)
   {  k=partition (R,S,T);
      quicksart (R,S,k);
      quicksart (R,k+1,T);
    }
}
```

3．有一种简单的排序算法,叫做计数排序。这种排序算法对一个待排序的表(用数组表示)进行排序,并将排序结果存放到另一个新的表中。必须注意的是,表中所有待排序的关键字互不相同,计数排序算法针对表中的每个记录,扫描待排序的表一趟,统计表中有多少个记录的关键字比该记录的关键字小。假设对某一个记录,统计出数值为 c,那么这个记录在新的有序表中的合适的存放位置即为 c。

(1)给出适用于计数排序的数据表定义。

(2)编写实现计数排序的算法。

(3)对于有 n 个记录的表,比较次数是多少?

(4)与直接选择排序相比,这种方法是否更好?为什么?

解:

(1)
```
  typedef struct
    { ElemType data;
      KeyType key;
    }listtype;
```

(2)
```
 void countsort(listtype a[],listtype b[],int n)
{int i,j,count;
  for(i=0;i<n;i++)
  {count=0;
    for(j=0;j<n;j++)
    if(a[j].key<a[i].key) count++;
    b[count]=a[i];
  }
}
```

(3) 对于有 n 个记录的表，关键字比较的次数是 n^2.

(4)直接选择排序比这种计数排序好，因为直接选择排序的比较次数为 n*(n-1)/2，且可在原地进行排序（稳定排序），而计数排序为不稳定排序，需要辅助空间多，为 O(n).