内容简介

本书由全国计算机专业排名领先的清华大学、中国科学院研究生院、国防科技大学等名校的资深教授、专家和一线教学骨干等组成的强大作者队伍精心编写，力求准确、精练、系统地阐述大纲规定的知识点，注重复习的系统性并与学生实际相结合，深入浅出，不仅让考生学懂学会，还给出大量例题和习题让考生学练结合，达到举一反三、事半功倍的复习效果。

图书在版编目(CIP)数据

2012年全国硕士研究生入学统一考试计算机专业基础综合考试大纲解析/全国硕士研究生入学统一考试辅导用书编委会编. —北京：高等教育出版社，2011.8

ISBN 978-7-04-033104-2

Ⅰ. ①2… Ⅱ. ①全… Ⅲ. ①电子计算机-研究生-入学考试-自学参考资料 Ⅳ. ①TP3

中国版本图书馆CIP数据核字(2011)第162412号

策划编辑 刘 佳　　责任编辑 何新权　　封面设计 王凌波
版式设计 余 杨　　责任校对 张小镝　　责任印制 朱学忠

出版发行	高等教育出版社	咨询电话	400-810-0598
社　　址	北京市西城区德外大街4号	网　　址	http://www.hep.edu.cn
邮政编码	100120		http://www.hep.com.cn
印　　刷	涿州市星河印刷有限公司	网上订购	http://www.landraco.com
开　　本	787mm×1092mm　1/16		http://www.landraco.com.cn
印　　张	33	版　　次	2011年8月第1版
字　　数	970千字	印　　次	2011年8月第1次印刷
购书热线	010-58581118	定　　价	65.00元

物 料 号　33104-00

出版前言

一、教育部制定的**《2012年全国硕士研究生入学统一考试计算机科学与技术学科联考计算机学科专业基础综合考试大纲》**（下称《考试大纲》）规定了2012年全国硕士研究生入学考试计算机科目的考试范围、考试要求、考试形式、试卷结构等。它既是2012年全国硕士研究生入学统一考试计算机专业命题的唯一依据，也是考生复习备考必不可少的工具书。

二、**《2012年全国硕士研究生入学统一考试计算机学科专业基础综合考试大纲解析》**根据《考试大纲》的要求和最新精神，深入研究考研命题的特点及动态，结合作者多年阅卷工作总结，特别注重与考生的实际相结合，注重与考研的要求相结合。

本书由数据结构、计算机组成原理、操作系统、计算机网络四部分组成。其中各部分包括以下三部分内容：

（一）复习要点——使考生明确各章的重点、难点及常考点，考生弄清各知识点之间的相互联系，以及多年考试中本章节的出题情况，以便对本章内容有一个全局性的认识和把握。

（二）考点精讲——本部分参考当前国内最权威的大学教材，对大纲所要求的知识点进行了全面、准确地阐述，以加深考生对基本概念和原理等重点内容的理解和正确应用。本部分讲解考点明确、重点突出、层次清晰、简明实用。

（三）例题与练习——通过对经典例题的分析教会考生分析问题解决问题的方法和技巧。通过大量练习题，使考生学练结合。更好地巩固所学知识，提高实战能力。

本书由清华大学殷人昆教授主编并审订，清华大学王诚、董长洪，中国科学院研究生院鲁士文，国防科技大学邹鹏、尹俊文等分别对所负责章节进行了认真的研究和撰写，在此对他们严谨的治学态度和付出的智慧与努力表示感谢！

三、**《2012年考研计算机考试大纲解析配套1000题》**本书由经验丰富的考研辅导专家根据全面调整后的《2012年全国硕士研究生入学统一考试计算机科学与技术学科联考计算机学科专业基础综合考试大纲》、《2012年全国硕士研究生入学统一考试计算机专业基础综合考试大纲解析》编写，将大纲和大纲解析中的考点、重点和难点与试题结合，使考生在学习《大纲解析》后通过难易适度的练习题达到检测复习效果、巩固基础、掌握重点、提高解题能力的目的，真正实现记、练、用的结合。

在开始复习的时候，最好把本书对照《2012年全国硕士研究生入学统一考试计算机专业基础综合考试大纲解析》复习，看一章即做一章相应的练习，以检测复习效果，帮助理解和掌握考点。本书可贯穿复习始终，前期可以作为同步训练，后期用于强化训练。

为了给考生提供更多的增值服务，凡购正版全国考研辅导班系列用书的考生都可以登录“中国教育考试在线”www.eduexam.com.cn做考研全真模拟试卷。

高等教育出版社
2011年8月

在链表中寻找第 i 个结点、两个有序链表的合并等。

(6) 单链表的递归算法,包括统计链表结点个数、在链表中寻找与给定值 x 匹配的结点、在链表中寻找第 i 个结点、求链表各结点值的和、平均值等。

(7) 循环链表的迭代算法、双向链表的迭代算法。

5. 多项式的建立,两个多项式的相加,两个多项式的相乘算法。

考点精讲

1.1 线性表的定义和基本操作

1.1.1 线性表的定义

通常,定义线性表为 n 个数据元素(或称为表元)的有限序列。记为 $L = (a_1, a_2, \cdots, a_n)$。其中,$L$ 是表名,a_i 是表中的结点,是不可再分割的数据。n 是表中表元的个数,也称为表的长度,若 $n = 0$ 叫做空表。线性表的特点是,在非空的数据元素集合中:

- 存在唯一的一个称作"第一个"的元素;
- 存在唯一的一个称作"最后一个"的元素;
- 除第一个元素外,集合中的每个元素均只有一个直接前驱;
- 除最后一个元素外,集合中的每个元素均只有一个直接后继。

其中,最后两个特点体现了线性表中元素之间的逻辑关系。

理解线性表的要点是:

1. 表中元素具有逻辑上的顺序性,在序列中各元素排列有其先后次序。各个数据元素在线性表中的逻辑位置只取决于其序号。

2. 表中元素个数有限。

3. 表中元素都是数据元素。就是说,每一个表元素都是不可再分的原子数据。

4. 表中元素的数据类型都相同。这意味着每一个表元素占有相同数量的存储空间。

5. 表中元素具有抽象性。就是说,仅讨论表元素之间的逻辑关系,不考虑元素究竟表示什么内容。

1.1.2 线性表的操作

线性表的主要操作有:

1. 表的初始化运算:将线性表置为空表。

2. 求表长度运算:统计线性表中表元素个数。

3. 查找运算:查找线性表中第 i 个表元素或查找表中具有给定关键字值的表元素。

4. 插入运算:将新表元素插入到线性表第 i 个位置上,或插入到具有给定关键字值的表元素的前面或后面。

5. 删除运算:删除线性表第 i 个表元素或具有给定关键字值的表元素。

6. 读取运算:读取线性表第 i 个表元素的值。

7. 复制运算:复制线性表所有表元素到另一个线性表中。

主要操作的实现取决于采用哪一种存储结构。存储结构不同,实现的算法也不同。

1.2 线性表的实现

1.2.1 线性表的顺序存储

线性表的顺序存储又称为顺序表。它用一组地址连续的存储单元依次存储线性表中的数据元素,从而使得逻辑关系相邻的两个元素在物理位置上也相邻。因此,顺序表的特点是表中各元素的逻辑顺序与其物理顺序相同。

用 C 语言描述时借用一个一维数组来存储这些表元素。

程序 1.1 顺序表的静态存储分配。

```
#define maxSize 100                          //显式地定义表的长度
typedef int DataType;                        //定义表元素的数据类型
typedef struct {                             //顺序表的定义
    DataType data[maxSize];                  //静态分配存储表元素的数组
    int n;                                   //实际表元素个数,0≤n≤maxSize
} SeqList;
```

在这种存储方式下,表元素 a_i 存储在 data[$i-1$]位置。存储结构如图 1.1.1 所示。

下标位置	0	1	…	$i-1$	i	…	$n-1$	…	maxSize − 1
数组(线性表)存储空间	a_1	a_2	…	a_i	a_{i+1}	…	a_n	…	…

图 1.1.1 顺序表的示意图

假设顺序表 A 的起始存储位置为 $Loc(1)$,第 i 个表项的存储位置为 $Loc(i)$,则有:

$$Loc(i) = Loc(1) + (i-1) \times \text{sizeof(DataType)}$$

其中,$Loc(1)$是第一个表项的存储位置,即数组中第 0 个元素位置。sizeof(DataType)是表中每个元素所占空间的大小。根据这个计算关系,可随机存取表中的任一个元素。

一维数组可以是静态分配的,也可以是动态分配的。在静态分配存储的情形下,由于数组的大小和空间事先已经固定分配,一旦数据空间占满,再加入新的数据就将产生溢出,此时存储空间不能扩充,就会导致程序停止工作。而在动态分配存储的情形下,存储数组的空间是在程序执行过程中通过动态存储分配的语句分配的,一旦数据空间占满,可以另外再分配一块更大的存储空间,用以代换原来的存储空间,从而达到扩充存储数组空间的目的,同时需将表示数组大小的常量 maxSize 放在顺序表的结构内定义,可以动态地记录扩充后数组空间的大小,提高结构的灵活性。

程序 1.2 顺序表的动态存储分配。

```
#define initSize 100                         //表长度的初始定义
typedef int DataType;                        //定义表元素的数据类型
typedef struct {                             //顺序表的定义
    DataType *data;                          //指示动态分配数组的指针
    int maxSize,n;                           //数组的最大容量和当前个数
} SeqList;
```

初始的动态分配语句为:

```
data=(DataType *) malloc (sizeof (DataType) * initSize);
//C++要简单得多,是 data=new Data Type[initSize];以后采用此种描述
maxSize=initSize;  n=0;
```

1.2.2 线性表的链式存储

线性表的链式存储又称为线性链表。在这种结构中数据元素存储在结点中,结点之间在空间上可以连续,也可以不连续,通过结点内附的链接指针来表示元素之间的逻辑关系。因此,在线性链表中逻辑上相邻的表元素在物理上不一定相邻。

最简单的线性链表是单链表,用 C 语言描述如下:

程序 1.3 单链表的定义。

```
typedef int DataType;
typedef struct node {
    DataType data;
    struct node *link;
} LinkedNode, *LinkList;
```

此时，使用一个指向链表结点的指针 htp 标识链表的表头：

LinkedNode * hpt 或 LinkList hpt

为了表示链表收尾，链表最后一个结点的链接指针应置为空。

1.2.3 顺序存储与链式存储的比较

从访问方式来看，顺序表可以顺序存取，也可以直接存取（注意它与一维数组的区别），线性链表只能从链头顺序存取。

从表元素的逻辑顺序与物理位置的对应关系来看，顺序表中表元素的逻辑顺序与它们的物理存储顺序是完全一致的，而线性链表中各个表元素的逻辑顺序与物理存储顺序不一定相同。

从存储空间的利用率来看，若定义存储密度为：存储密度 = 表中数据元素占有的空间/分配给表的总空间，则顺序表的存储密度为 1，因为数据元素之间的逻辑关系无须占用附加空间；而线性链表的存储密度小于 1，因为每个数据元素都需附加一个链接指针以指示元素之间的逻辑关系。

从查找速度来看，由于线性链表只能沿链逐个比较，而顺序表可以按照元素序号（下标）直接访问，故顺序表查找速度比线性链表要快；从插入和删除速度来看，如果要求插入和删除后表中其他元素的相对逻辑顺序保持不变，则顺序表平均需要移动大约一半元素，而线性链表只需修改链接指针，不需要移动元素，因此线性链表比顺序表的插入和删除速度快。

从 C 指针的使用来看，顺序表的情形下，指针 p 指示数据元素存储位置，用 * p 可取得该数据的值，用 p ++可以顺序进到物理上下一个数据元素的位置；在线性链表的情形下，指针 p 指示链表结点的地址，用 * p 不能取得该结点数据的值，用 p++也不能进到下一个结点位置，只能使用 p->data 取得结点数据的值，用 p = p->link 进到下一个结点。

从空间限制来看，顺序表在静态存储分配的情形下，一旦存储空间装满不能扩充，如果再加入新元素将出现存储溢出；在动态存储分配的情形下虽然存储可以扩充，但需要移动大量元素，将导致操作效率降低。线性链表的结点空间只有在需要的时候才申请，无需事先分配，因此，只要还有空间可分配，就没有存储溢出问题，操作效率也优于顺序表。

1.2.4 其他线性链表的形式

根据结点中指针信息的实现方式，还有其他几种链表结构：

1. 双向链表：每个结点包含两个指针，指明直接前驱和直接后继元素，可在两个方向上遍历其后及其前的元素。

2. 循环链表：表尾结点的后继指针指向表中的第一个结点，可在任何位置上遍历整个链表。

3. 静态链表：借助数组来描述线性表的链式存储结构。

在链式存储结构中，只需要一个指针（头指针）指向第一个结点，就可以顺序访问到表中的任意一个元素。为了简化对链表状态的判定和处理，特别引入一个不存储数据元素的结点，称为头结点，将其作为链表的第一个结点并令头指针指向该结点。

1.3 线性表的插入和删除运算

1.3.1 基于顺序存储结构的运算

如果在插入和删除后不需要保持表中元素的原有逻辑顺序，则可直接把新元素插入（或追加）到表尾，或把表尾元素直接覆盖表中的被删除元素，其平均移动元素个数为 1。如果在插入和删除后需要保持表中元素的原有逻辑顺序，则在插入元素前需要移动元素以挪出空的存储单元，然后再插入元素；删除元素时同样需要移动元素，以填充被删元素空出来的存储单元。如图 1.1.2 所示。

在等概率下平均移动元素的次数分别为：

$$E_{\text{insert}} = \sum_{i=1}^{n+1} p_i \times (n-i+1) = \frac{1}{n+1}\sum_{i=1}^{n+1}(n-i+1) = \frac{n}{2}$$

$$E_{\text{delete}} = \sum_{i=1}^{n} q_i \times (n-i) = \frac{1}{n}\sum_{i=1}^{n}(n-i) = \frac{n-1}{2}$$

```
    while ( i <= L.n ) {                                    //循环检测
        while ( L.data[i-1] != L.data[i] ) i++;             //寻找重复的元素 i 与元素 i+1
        j = i+2;                                            //发现重复
        while ( L.data[i-1] == L.data[j-1] ) j++;           //元素 j 是与元素 i 不重复的元素
        for ( k = i+1,u=j;u<= < L.n; k++, u++ )
          L.data[k-1] = L.data[u-1];                        //前移
        L.n = L.n-j+i+1;
        i++;
    }
    return true;
};
```

2. 针对带表头结点的单链表,试编写下列函数:

(1) 实现定位函数的算法

```
LinkedNode *Locate ( LinkList L, int i ) {
//取得单链表中第 i 个结点地址, i 从 0 开始计数, i = 0 定位于表头结点
    if ( i < 0 ) return NULL;                               //位置 i 在表中不存在
    LinkedNode *p = L;  int k = 0;                          //从表头结点开始检测
    while ( p != NULL && k < i )                            //p == NULL 表示链短, 无第 i 个结点
      { p = p->link;  k++; }
    return p;                                               //否则 k == i, 返回第 i 个结点地址
};
```

(2) 实现求最大值的函数

```
LinkedNode *Max ( LinkList L ) {
//在单链表中进行一趟检测,找出具有最大值的结点地址, 如果表空, 返回指针 NULL
    if ( L->link == NULL ) return NULL;                     //空表, 返回指针 NULL
    LinkedNode * pmax = L->link, p = L->link->link;         //假定头结点中数据值最大
    while ( p != NULL ) {                                   //循环, 下一个结点存在
        if ( p->data > pmax->data ) pmax = p;               //pmax 记忆找到的具有最大值结点
        p = p->link;                                        //检测下一个结点
    }
    return pmax;
};
```

(3) 实现统计单链表中具有给定值 x 的所有元素的函数

```
int Count ( LinkList L, DataType x ) {
//在单链表中进行一趟检测,统计具有给定值 x 的结点, 如果表空, 返回 0
    int n = 0;
    LinkedNode * p = L->link;                               //从第一个结点开始检测
    while ( p != NULL ) {                                   //循环, 下一个结点存在
        if ( p->data == x ) n++;                            //找到一个, 计数器加 1
        p = p->link;                                        //检测下一个结点
    }
    return n;
};
```

(4) 实现从一维数组 $A[n]$ 建立单链表的函数

```
void CreateList ( LinkList& L, DataType A[ ], int n ) {
//根据数组 A[n]建立一个单链表,单链表中各元素次序与 A[n]中各元素次序相同
    LinkedNode *rear;
    L = rear = new LinkedNode;                          //创建表头结点
    for ( int i = 0; i < n; i++ ) {
        rear->link = new LinkedNode;                    //链入一个新结点,值为 A[i]
        rear = rear->link;                              //rear 指向链中尾结点
        rear->data = A[i];
    }
    rear->link = NULL;
};
```

采用递归方法实现时,需要通过引用参数将已建立的单链表各个结点链接起来。为此,在递归地扫描数组 $A[n]$ 的过程中,先建立单链表的各个结点,在退出递归时将结点地址 p(被调用层的形参)带回上一层(调用层)的实参 p->link。

```
void createList ( DataType A[ ], int n, int i, LinkedNode *& p ) {
    if ( i == n ) p = NULL;
    else {
        p = new LinkedNode;
        p->data = A[i];  p->link = NULL;          //建立链表的新结点
        createList ( A, n, i+1, p->link );        //递归返回时通过 p->link 链接
    }
};
```

外部调用递归过程的语句为:

```
L = new LinkedNode;                               //建立表头结点
createList ( A, n, 0, L->link );                  //递归建立单链表
```

(5) 实现在非递减有序的单链表中删除值相同的多余结点的函数

```
void tidyup ( LinkList L ) {
    LinkedNode *p = L->link, temp;                //检测指针 p 指向首元结点
    while ( p != NULL && p->link != NULL )        //循环检测链表
      if ( p->data == p->link->data ) {           //若相邻结点数据值相等
        temp = p->link;  p->link = temp->link;
        delete temp;                              //删除后一个值相同的结点
      }
    else p = p->link;                             //指针 p 进到链表下一个结点
};
```

3. 已知 first 为单链表的表头指针,链表中存储的都是整型数据,试写出实现下列运算的递归算法:

(1) 求链表中的最大整数

```
int Max ( LinkedNode *first ) {
//递归算法 : 求链表中的最大值
    if ( first->link == NULL) return first->data;     //链表仅一个结点,其值即所求
    int temp = Max ( first->link );                   //否则递归求后继链表中最大值
    if ( first->data > temp ) return first->data;     //再与首元之值比较取大者
    else return temp;
};
```

```
typedef struct {
    DataType elem[maxSize];                    //循环队列的存储空间
    int front,rear,size;                       //队列的队头、队尾指针和实际元素个数
} seqQueue;
```

设循环队列的名字为 Q,则队空条件为 Q. size = =0,队满条件为:Q. size = =maxSize。此时都有 Q. front = =Q. rear。

3. 增设一个标志域 tag,来区分队空与队满条件。

程序 2.5 增设 tag 数据成员的循环队列结构。

```
#define maxSize 100
typedef int DataType;
typedef struct {
    DataType elem[maxSize];                    //循环队列的存储空间
    int front,rear,tag;                        //队列的队头、队尾指针和标志域
} seqQueue;
```

设循环队列的名字为 Q,则队空条件为 Q. front = =Q. rear && Q. tag = =0,而队满条件为 Q. front = =Q. rear && Q. tag = =1。

2.3.2 队列的链式存储结构

链式队列是基于单链表的一种存储表示。如图 1.2.4 所示。

在单链表的每一个结点中有两个域:data 域存放队列元素的值,link 域存放单链表下一个结点的地址。队列的队头指针指向单链表的第一个结点,队尾指针指向单链表的最后一个结点。这意味着队列的队头元素存放在单链表的第一个结点内,若要从队列中退出一个元素,必须从单链表中删去第一个结点,而存放着新元素的结点应插在队列的队尾,即单链表的最后一个结点后面,这个新结点将成为新的队尾。

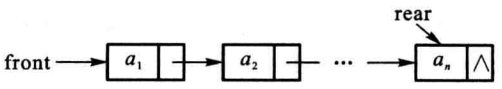


图 1.2.4 链式队列

用单链表表示的链式队列特别适合于数据元素变动比较大的情形,而且不存在队列满而产生溢出的情况。另外,假若程序中要使用多个队列,与多个栈的情形一样,最好使用链式队列。这样不会出现存储分配不合理的问题,也不需要进行存储的移动。

程序 2.6 链式队列的结构定义。

```
typedef int DataType;                          //队列元素的数据类型
typedef struct node {                          //链式队列结点
    DataType data;                             //结点的数据
    struct node * link;                        //结点的链接指针
} LinkedNode;
typedef struct {                               //链式队列
    LinkedNode * front, * rear;                //队列的队头和队尾指针
} LinkQueue;
```

2.3.3 队列操作的要点

1. 队列操作 deQueue (Q, x) 与 getFront (Q, x) 是有区别的。前者退出队列的队头元素,因此每执行一次 deQueue (Q,x) 操作,队列中的元素个数就少一个;后者仅复制出一份队头元素的值,队列中元素个数不发生变化。

2. 在循环队列中进行插入和删除时,不需要比较和移动任何元素,只需要修改队尾和队头指针,并向队尾插入元素或从队头取出元素。

3. 对循环队列的主要操作包括:

$$\begin{pmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,1} & a_{n-1,2} & \cdots & a_{n-1,n-1} \end{pmatrix} \quad \begin{pmatrix} a_{0,0} & & & \\ a_{1,0} & a_{1,1} & & \\ \vdots & \vdots & \ddots & \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,n-1} \end{pmatrix} \quad \begin{pmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ & a_{1,1} & \cdots & a_{1,n-1} \\ & & \ddots & \vdots \\ & & & a_{n-1,n-1} \end{pmatrix}$$

(a) 对称矩阵　　(b) 下三角阵　　(c) 上三角阵

图 1.2.9　对称矩阵、下(上)角阵

中的任一元素在一维数组中的下标位置，还要区分两种存储方式，即行优先方式和列优先方式。

设在一维数组 B 中从 0 号位置开始存放，$A[0][0]$ 存放于 $B[0]$。若只存下三角部分，并按行优先存储，则图 1.2.9(b) 的数组元素存于一维数组的一个线性序列如下：

B	$a_{0,0}$	$a_{1,0}$	$a_{1,1}$	$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$\cdots$	$a_{n-1,0}$	$a_{n-1,1}$	$\cdots$	$a_{n-1,n-1}$

对于矩阵 A 的任一数组元素 a_{ij}，在按行优先存放的情形下，当 $i \geqslant j$ 时，矩阵元素 a_{ij} 在 B 中有对应存放位置：

$$LOC(i,j) = 1+2+3+\cdots+i+j = (i+1)\times i \ /2+j$$

当 $i<j$ 时，矩阵元素 a_{ij} 在数组 B 中没有对应存放位置，但基于矩阵元素的对称性，可以通过寻找对称元素 a_{ji} 在数组 B 中的位置而访问到它的值。此时 a_{ij} 的值就是 a_{ji} 在数组 B 中存放的值。故：

$$LOC(i,j) = LOC(j,i) = (j+1)\times j \ /2+i$$

同样，若只存上三角部分，一维数组 B 中从 0 号位置开始存放，并按行优先存储，则图 1.2.9(c) 的数组元素存于一维数组的一个线性序列如下：

B	$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$\cdots$	$a_{0,n-1}$	$a_{1,1}$	$a_{1,2}$	$\cdots$	$a_{1,n-1}$	$a_{2,2}$	$\cdots$	$a_{n-1,n-1}$

由此可以看出，对矩阵 A 的任一矩阵元素 a_{ij}，在按行优先存储的情况下，当 $i \leqslant j$ 时，矩阵元素 a_{ij} 在一维数组 B 中有对应的存储位置：

$$LOC(i,j) = n+(n-1)+(n-2)+\cdots+(n-i+1)+(j-i) = (2\times n-i+1)\times i \ / \ 2+j-i$$
$$= (2\times n-i-1)\times i/2+j$$

当 $i>j$ 时，矩阵元素 a_{ij} 在数组 B 中没有存放，但可以通过寻找对称元素 a_{ji} 的位置而访问到它的值，此时 a_{ij} 的值就是 a_{ji} 在数组中位置存放的值。故：

$$LOC(i,j) = LOC(j,i) = (2\times n-j-1)\times j/2+i$$

2. 三对角线矩阵

设有一个 $n\times n$ 的方阵 A，对于矩阵 A 中的任一元素 a_{ij}，当 $|i-j|>1$ 时有 $a_{ij}=0(1\leqslant i\leqslant n, 1\leqslant j\leqslant n)$，则称这样的矩阵为三对角线矩阵。图 1.2.10 即为一个三对角线矩阵。

$$\begin{pmatrix} a_{11} & a_{12} & & & & \\ a_{21} & a_{22} & a_{23} & & 0 & \\ & a_{32} & a_{33} & a_{34} & & \\ & & \ddots & \ddots & \ddots & \\ & 0 & & a_{n-1n-2} & a_{n-1n-1} & a_{n-1n} \\ & & & & a_{nn-1} & a_{nn} \end{pmatrix}$$

图 1.2.10　三对角线矩阵

在该矩阵中除主对角线及在主对角线上下最临近的两条对角线上的元素外，所有其他元素均为 0。为了节省存储空间，只存储主对角线及其上、下两侧次对角线上的元素，主次对角线以外的零元素一律不存储。为此，可以用一个一维数组 B 来存储三对角线矩阵中位于三条对角线上的元素。这里同样要区分两种存储方式，即行优先方式和列优先方式。

现将三对角线矩阵 A 中三条对角线上的元素按行优先方式存放在一维数组 B 中，且 a_{11} 存放于 $B[0]$，则矩阵 A 的全部 $3n-2$ 个非零元素在数组 B 中的存放顺序为：

步序	扫描项	项类型	动作	栈的变化	输出
0			'#' 进栈,读下一符号	#	
1	*a*	操作数	直接输出,读下一符号	#	*a*
2	+	操作符	isp ('#') < icp ('+'),进栈,读下一符号	#+	
3	*b*	操作数	直接输出,读下一符号	#+	*b*
4	*	操作符	isp ('+') < icp ('*'),进栈,读下一符号	#+*	
5	(	操作符	isp ('*') < icp ('('),进栈,读下一符号	#+*(	
6	*c*	操作数	直接输出,读下一符号	#+*(	*c*
7	-	操作符	isp ('(') < icp ('-'),进栈,读下一符号	#+*(-	
8	*d*	操作数	直接输出,读下一符号	#+*(-	*d*
9	)	操作符	isp ('-') > icp (')'),退栈输出	#+*(	-
10			isp ('(')= =icp (')'),退栈,读下一符号	#+*	
11	+	操作符	isp ('*') > icp ('+'),退栈输出	#+	*
12			isp ('+') > icp ('+'),退栈输出	#	+
13			isp ('#') < icp ('+'),进栈,读下一符号	#+	
14	*e*	操作数	直接输出,读下一符号	#+	*e*
15	/	操作符	isp ('+') < icp ('/'),进栈,读下一符号	#+/	
16	*f*	操作数	直接输出,读下一符号	#+/	*f*
17	#	操作符	isp ('/') > icp ('#'),退栈输出	#+	/
18			isp ('+') > icp ('#'),退栈输出	#	+
19			isp ('#')= =icp ('#'),退栈,结束		

第3章 树与二叉树

本章主要涉及树与森林、二叉树的定义、性质、操作和相关算法的实现。特别是二叉树的遍历算法,它们与许多以此为基础的递归算法都必须认真复习。线索二叉树是直接利用二叉链表的空链指针记入前驱和后继线索,从而简化二叉树的遍历。在树的应用方面,主要涉及二叉排序树、平衡二叉树、Huffman 树、堆。二叉排序树和平衡二叉树在建立小型目录方面非常有用。Huffman 树的核心问题是 Huffman 算法,作为 Huffman 树的应用,引入 Huffman 编码,它是数据压缩的基础。堆主要用于实现优先级队列。

复习要点

1. 树与二叉树的概念:包括树的定义与二叉树的定义,树与二叉树的相关术语,结点的度、层次、深度、高度,树的度、深度、高度。

2. 二叉树的性质:包括二叉树中层次与结点个数的关系,二叉树中高度与结点个数的关系,二叉树中结点编号与层次的关系,完全二叉树的结点间关系,完全二叉树中高度与结点个数的关系,树与二叉树的定义的递归性质,以及相应递归算法。

3. 二叉树的存储结构:包括顺序存储结构和链式存储结构的 C 定义,不同存储结构的互换方法(不要求算法)。

4. 二叉树的遍历:包括二叉树的前序、中序、后序遍历的递归算法,利用前序与中序、中序与后序遍历结果构造二叉树的方法(不要求算法),不同种类的二叉树棵数的计数,使用栈的二叉树前序、中序、后序遍历的非递归算法,使用队列的二叉树的层次序遍历算法。

5. 二叉树遍历算法的应用:包括从二叉树的前序序列,借助前序遍历的思想构造二叉树的递归算法,统

后者。

3. 外结点和内结点。有三种场合要区分外结点和内结点：

(1) 判定树。外结点是决策的结果，内结点是决策过程的判决分支。

(2) Huffman 树。外结点是带有权重的具特殊含义的(树叶)结点，内结点是构造 Huffman 树过程中不断合成的更大二叉树的根结点。

(3) 分析查找效率的判定树。在分析一些查找算法的效率时可利用判定树。内结点是查找成功时检测指针停留的结点，代表查找数据集合中已有的数据；外结点是查找失败时检测指针走到的结点，它们实际上不存在，代表查找数据集合中没有的数据。

4. 结点的层次、结点的深度和高度、树的深度和高度。要注意的是，这些概念在国内外教材中也有分歧。有人认为根结点所在层次为 1，有人认为根结点所在层次为 0。本书倾向于前者。另外，树的深度是从根结点开始(其深度为 1)自顶向下逐层累加的，而高度是从叶结点开始(其高度为 1)自底向上逐层累加的，虽然树的深度与树的高度一样，但具体到树中某个结点，其深度和高度是不一样的。

5. 路径和路径长度。要注意的是，树中两个结点之间的路径是由这两个结点之间所经过的结点序列表征的，但其路径长度则是由这些结点经过的分支条数来确定的。树的外部路径长度是各外结点到根结点的路径长度之和，树的内部路径长度是各内结点到根结点的路径长度之和。

6. 有序树和无序树。

7. 森林。是 $m(m\geqslant 0)$ 棵互不相交的树的集合。

由于树的一些概念有不同的定义，故在考试时应注意题干中对树的概念的提法是哪一种。

3.2 二叉树

3.2.1 二叉树的定义与特性

1. 二叉树的定义

二叉树的定义为：二叉树或者是一棵空树，或者是一棵由一个根结点和两棵互不相交的分别称作根的左子树和右子树所组成的非空树，其左子树和右子树又同样是一棵二叉树。这是一个递归的定义，递归结束于空的子树(注意，不能说没有子树)。二叉树是有序树，根结点的两棵子树要区分左子树和右子树，它们的地位不能互换。

有的教材刻意声明二叉树不是树，本书不做这种区分。另外还有几个特殊的二叉树，即满二叉树、完全二叉树和正则二叉树。

设二叉树 T 有 n 个结点，令 $k=\lceil \log_2(n+1)\rceil$①，$r=n-(2^{k-1}-1)$，则 k 代表离树根最远的结点所处层次，即二叉树的深度，r 代表第 k 层结点个数。如果其中 $2^{k-1}-1$ 个结点放满第 1 到第 $k-1$ 层，则：

(1) 若 $0<r\leqslant 2^{k-1}$，且这 r 个结点集中存放在第 k 层的左侧，则称 T 是一棵完全二叉树。

(2) 特别地，若 $r=2^{k-1}$，则称 T 是一棵满二叉树。满二叉树是完全二叉树。

(3) 若 $0<r\leqslant 2^{k-1}$，但这 r 个结点随意分布在第 k 层上，则称 T 为丰满树，或称为理想平衡树。

2. 二叉树的性质

性质 1：二叉树上叶结点的结点数等于度为 2 的结点的结点个数加 1。

由此可以引申出以下结论：对有 n 个结点的完全二叉树：

(1) 若 n 为奇数，则树中只有度为 0 和度为 2 的结点，且度为 0 的结点有$\lceil n/2\rceil$ 个，度为 2 的结点有$\lfloor n/2\rfloor$ 个。

(2) 若 n 为偶数，则树中除了度为 0 和度为 2 的结点外，还有 1 个度为 1 的结点。

性质 2：二叉树第 i 层上至多有 2^{i-1} 个结点($i\geqslant 1$，根结点所在层次为 1)。

性质 3：高度为 h 的二叉树至多有 2^h-1 个结点($h\geqslant 0$，空树的高度为 0)。

性质 4：如果将一棵有 n 个结点的完全二叉树自顶向下，同一层自左向右连续给结点编号 $1,2,\cdots,n$，并

① $\lceil x\rceil$表示对 x 向上取整，如$\lceil 1.1\rceil=2$。而$\lfloor x\rfloor$表示对 x 向下取整，如$\lfloor 1.7\rfloor=1$。

```
void InOrder(BiTree root){
    stack S;  InitStack(S);                    //栈元素定义为 BitNode * 指针类型
    BiTNode  * p=root;                         //p 是遍历指针,从根结点开始
    do {                                       //每一趟循环访问一个结点
        while(p ! =NULL){                      //每遇到非空二叉树先向左走
            Push(S, p);  p=p->lchild;
        }
        if(! IsEmpty(S)){                      //左边走不动退栈访问根结点
            Pop(S, p);                         //退栈,访问根结点
            printf("%d", p->data);
            p=p->rchild;                       //再向右子树走
        }                                      //p 空而栈不空,还要回访更高层树根
    } while(p ! =NULL || ! IsEmpty(S));        //p 不为空但栈空,还要遍历右子树
};
```

二叉树的层次序遍历是指从根结点出发,自顶向下逐层访问二叉树的结点,对于每一层结点则从左到右依次访问,这个过程就是二叉树层次序遍历。

程序 3.6　二叉树的层次序遍历算法。

```
#include "queue.h"
void LevelOrder(BiTree root){
    BiTNode  * p;
    Queue Q;InitQueue(Q);                      //队列元素定义为 BitNode * 指针类型
    EnQueue(Q, root);                          //将根指针加入队列
    while(! Empty(Q)){                         //每一趟循环访问一个结点
        DeQueue(Q, p);                         //队头元素出队
        printf("%d", p->data);                 //访问结点
        if(p->lchild ! =NULL) EnQueue(p->lchild);     //两个子女加入队尾
        if(p->rchild ! =NULL) EnQueue(p->rchild);
    }
};
```

应用二叉树遍历,可以实现很多有关二叉树的操作,如计算二叉树的结点个数,计算二叉树高度,二叉树的复制,判断两棵二叉树是否同构(相等)。

3.2.4　线索二叉树

1. 线索二叉树的概念

二叉树的遍历实质上是对一个非线性结构进行线性化的过程,它使得每个结点(除第一个和最后一个外)在这些线性序列中有且仅有一个直接前驱和直接后继。但在二叉链表存储结构中,只能找到一个结点的左、右子女信息,不能直接得到结点在某种遍历序列中的前驱和后继信息,可以引入线索二叉树来保存这些信息。

为此,改造二叉树的结点,在结点的空指针域中存放该结点在某种遍历次序下的前驱结点或后继结点的指针,并称之为线索。对一棵二叉树中的所有结点的空指针域按照某种遍历次序加线索的过程叫做线索化,被线索化了的二叉树称作线索二叉树。在空的左指针域中存放的指向其前驱结点的指针叫做前驱线索,在空的右指针域中存放的指向其后继结点的指针叫做后继线索。

程序 3.7　线索二叉树的结点构造。

```
typedef int DataType;
typedef struct node {                          //线索二叉树的结点
```

程序 3.9 通过中序遍历对二叉树线索化的递归算法。

```
void InThread(ThreadNode *p, ThreadNode *& pre){
    if(p != NULL){
        InThread(p->lchild, pre);                    //递归,左子树线索化
        if(p->lchild==NULL)                          //建立当前结点的前驱线索
            {p->lchild=pre;p->ltag=1;}
        if(pre != NULL && pre->rchild==NULL)         //建立前驱结点的后继线索
            {pre->rchild=p;pre->rtag=1;}
        pre=p;                                       //前驱跟上,当前指针向前遍历
        InThread(p->rchild, pre);                    //递归, 右子树线索化
    }
};
```

注意,线索化后没有空的子女指针,只有空的线索。空的线索表明线索"悬空",没有前驱或后继。

4. 访问中序线索二叉树

访问运算主要是为遍历中序线索二叉树服务的。这种遍历不再需要栈,因为它利用了隐含在线索二叉树中的前驱和后继信息。

程序 3.10 求以 p 为根的中序线索二叉树中中序序列下的第一个结点。

```
ThreadNode *First(ThreadNode *p){
    while(p->ltag==0)p=p->lchild;                    //最左下结点(不一定是叶结点)
    return p;
};
```

程序 3.11 求在中序线索二叉树中结点 p 在中序序列下的后继结点。

```
ThreadNode *Next(ThreadNode *p){
    if(p->rtag==0) return First(p->rchild);
    else return p->rchild;                           //rtag==1,直接返回后继线索
};
```

如果把程序 First 的 ltag 和 lchild 换成 rtag 和 rchild,可得到求中序序列下最后一个结点的运算 Last;如果把程序 Next 的 rtag 和 rchild 换成 ltag 和 lchild,可得到求中序序列下前驱结点的运算 Prior。

程序 3.12 在中序线索二叉树上执行中序遍历的算法。

```
void Inorder(ThreadNode *root){
    for(ThreadNode *p=First(root); p != NULL; p=Next(p))
        printf("%d", p->data);
};
```

除了中序线索二叉树外,还有通过前序遍历二叉树得到的前序线索二叉树和通过后序遍历得到的后序线索二叉树。在不同序的线索二叉树上寻找前驱和后继的算法很不相同,需要具体分析。

3.3 树与森林

3.3.1 树的存储结构

树的存储结构常用的有三种。

1. 双亲表示法

用一个一维数组存储树中的结点,同时在每个结点中附设一个指示器指示该结点的父结点在数组中的位置,如图 1.3.4 所示。

```
    if(p==NULL) return false;                          //查找失败,不作删除
    if(p->lchild != NULL && p->rchild != NULL){
        s=p->lchild;                                   //找 p 的中序前驱 s
        while(s->rchild != NULL){f=s; s=s->rchild;}
        p->data=s->data;  p=s;
    }
    if(p->lchild != NULL) s=p->lchild;                 //记录非空子女结点
    else s=p->rchild;
    if(p==root) root=s;                                //被删结点为根结点
    else if(s->data<f->data) f->lchild=s;
        else f->rchild=s;
    free(p);                                           //释放被删结点
    return true;
};
```

5. 二叉排序树的性能分析

若设二叉排序树有 n 个结点,各结点查找概率分别为 $p_1,p_2,\cdots,p_n$,且 $\sum_{i=1}^{n} p_i = 1$,则其查找算法的平均查找长度 ASL 为:

$$ASL = \sum_{i=1}^{n} p_i \times c_i = \sum_{i=1}^{n} p_i \times l_i$$

其中,l_i 是查找第 i 个结点的比较次数,等于该结点所在层次编号。由此公式可知,在相等查找概率的情形下,ASL 的大小取决于树的高度。如果插入结点的关键字序列是一个按值递增的有序序列,会导致建立一个右斜的单枝树,使得查找性能显著变坏,其平均查找长度达到 $O(n)$。如果建立起来的二叉排序树的左右子树的深度之差的绝对值不超过 1,这样的二叉排序树称为平衡二叉树,简称为平衡树。它的平均查找长度达到 $O(\log_2 n)$。

3.4.2 平衡二叉树

1. 平衡二叉树的概念

平衡二叉树又称为 AVL 树或高度平衡的二叉排序树。它或者是一棵空树,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 AVL 树,且左子树和右子树的高度之差的绝对值不超过 1。

若将二叉树结点的平衡因子(Balance Factor,BF)定义为该结点左子树的高度减去右子树的高度所得的差,则平衡二叉树上所有结点的平衡因子只可能是-1、0 和 1。

只要树上有一个结点的平衡因子的绝对值大于 1,则该二叉树就是不平衡的。

分析二叉排序树的查找过程可知,设一棵二叉排序树有 n 个结点,在相等查找概率的情形下,其高度保持在 $O(\log_2 n)$,查找性能才能达到最佳。因此,希望在构造二叉排序树的过程中,保持其为一棵平衡二叉树。

使平衡二叉树保持平衡的基本思想是:每当在平衡二叉树中插入一个结点时,首先检查在其插入路径上的结点是否因为插入而导致了不平衡。若是,则找出其中的最小不平衡二叉树,在保持二叉排序树特性的情况下,调整最小不平衡子树中结点之间的关系,使之达到新的平衡。所谓最小不平衡子树,指在插入路径上离插入结点最近的平衡因子的绝对值大于 1 的结点作为根的子树。

2. 平衡二叉树的插入

在平衡二叉树上的查找和插入过程与二叉排序树相同,但在新结点作为叶结点插入后,必须从插入位置沿到根结点的路径回溯,检查在此路径上的结点是否变得不平衡。若设在平衡二叉树上因为插入新结点而失去平衡的最小子树根结点的指针为 a,则重新平衡化的调整的情况可归纳为下列 4 种情况:

(1) 右单旋转平衡处理。由于在 a 的左子树根结点的左子树上插入新结点,a 的平衡因子由 1 增至 2,

【解答】 A。用 n 个权值构造出来的 Huffman 树共有 $2n-1$ 个结点，其中叶结点 n 个，度为 2 的非叶结点 $n-1$ 个。

二、综合应用题

例 1 设二叉树根结点所在层次为 0，树的深度 d 为距离根最远的叶结点所在层次，试回答以下问题：

（1）试精确给出深度为 d 的完全二叉树的不同二叉树棵数；

（2）试精确给出深度为 d 的满二叉树的不同二叉树棵数。

【解答】 （1）这与教材上讲的根结点所在层次为 1 的情形相比，深度差 1。在第 d 层最多有 2^d 个结点。因此，深度为 d 的不同完全二叉树有 2^d 棵。

（2）深度为 d 的满二叉树只有 1 棵。

例 2 如果一棵有 n 个结点的满二叉树的高度为 h（根结点所在的层次为 1），则：

（1）用高度如何表示结点总数 n？用结点总数 n 如何表示高度 h？

（2）若对该树的结点从 0 开始按中序遍历次序进行编号，则如何用高度 h 表示根结点的编号？如何用高度 h 表示根结点的左子女结点的编号和右子女结点的编号？

【解答】 按照题意，该满二叉树的高度为 h，结点总数为 n，那么：

（1）按照二叉树的性质，$n = 2^h-1$，反之，$h = \log_2(n+1)$。

（2）用高度 h 确定根结点编号的依据有三：一、从满二叉树推知，结点数有 $n = 2^h-1$ 个，编号从 0 到 $n-1(=2^h-2)$；二、由于是按中序遍历次序所作的编号，根结点的左、右子树的结点数相等，根结点的编号应位于正中；三、按照求中点的公式，中点的编号应为 $\text{mid} = (\text{low}+\text{high}) / 2 = (0+2^h-2) / 2 = 2^{h-1}-1$，此即满二叉树根结点的编号。依此类推，根结点左子树有 $2^{h-1}-1$ 个结点，其结点编号从 0 到 $2^{h-1}-2$，左子树根结点，即根结点左子女结点的编号为 $2^{h-2}-1$；而右子女结点的编号为 $2^{h-1}+(2^{h-2}-1) = 3\times 2^{h-2}-1$。

例 3 一棵高度为 h 的满 m 叉树有如下性质：第 h 层上的结点都是叶结点，其余各层上每个结点都有 m 棵非空子树。如果按层次自顶向下，同一层自左向右，顺序从 1 开始对全部结点进行编号，试问：

（1）各层的结点个数是多少？

（2）编号为 i 的结点的父结点（若存在）的编号是多少？

（3）编号为 i 的结点的第 k 个子女结点（若存在）的编号是多少？

（4）编号为 i 的结点有右兄弟的条件是什么？其右兄弟结点的编号是多少？

（5）若结点个数为 n，则高度 h 是 n 的什么函数关系？

【解答】 可以参照二叉树的性质，将其扩展到 m 叉树。

（1）因为第 1 层有 1 个结点，第 2 层有 m 个结点，第 3 层有 m^2 个结点……一般地，第 i 层有 m^{i-1} 个结点（$1\leqslant i\leqslant h$）。

（2）在 m 叉树的情形，结点 i 的第 1 个子女编号为 $j = (i-1)\times m+2$，反过来，结点 i 的双亲的编号是 $\lfloor (i-2) / m\rfloor+1$，根结点没有双亲，所以以上计算式要求 $i>1$。如果考虑对于 $i = 1$ 也适用（根的双亲设为 0），可将上式改为 $\lfloor (i+m-2) / m\rfloor$。

（3）因为结点 i 的第 1 个子女编号为 $(i-1)\times m+2$，若设该结点子女的序号为 $k = 1, 2, \cdots, m$，则第 k 个子女结点的编号为 $(i-1)\times m+k+1 (1\leqslant k\leqslant m)$。

（4）当结点 i 不是其双亲的第 m 个子女时才有右兄弟。若设其双亲编号为 j，可得 $j = \lfloor (i+m-2) / m\rfloor$，结点 j 的第 m 个子女的编号为

$$(j-1)\times m+m+1 = j\times m+1 = \lfloor (i+m-2) / m\rfloor\times m+1$$

所以当结点的编号 $i\leqslant \lfloor (i+m-2) / m\rfloor\times m$ 时才有右兄弟，右兄弟的编号为 $i+1$。

（5）若结点个数为 n，则有

$$n = \sum_{i=1}^{h} m^{i-1} = 1 + m + m^2 + \cdots + m^{h-1} = \frac{1}{m-1}(m^h - 1)$$

反之，有 $h=\log_m(n\times(m-1)+1)$。

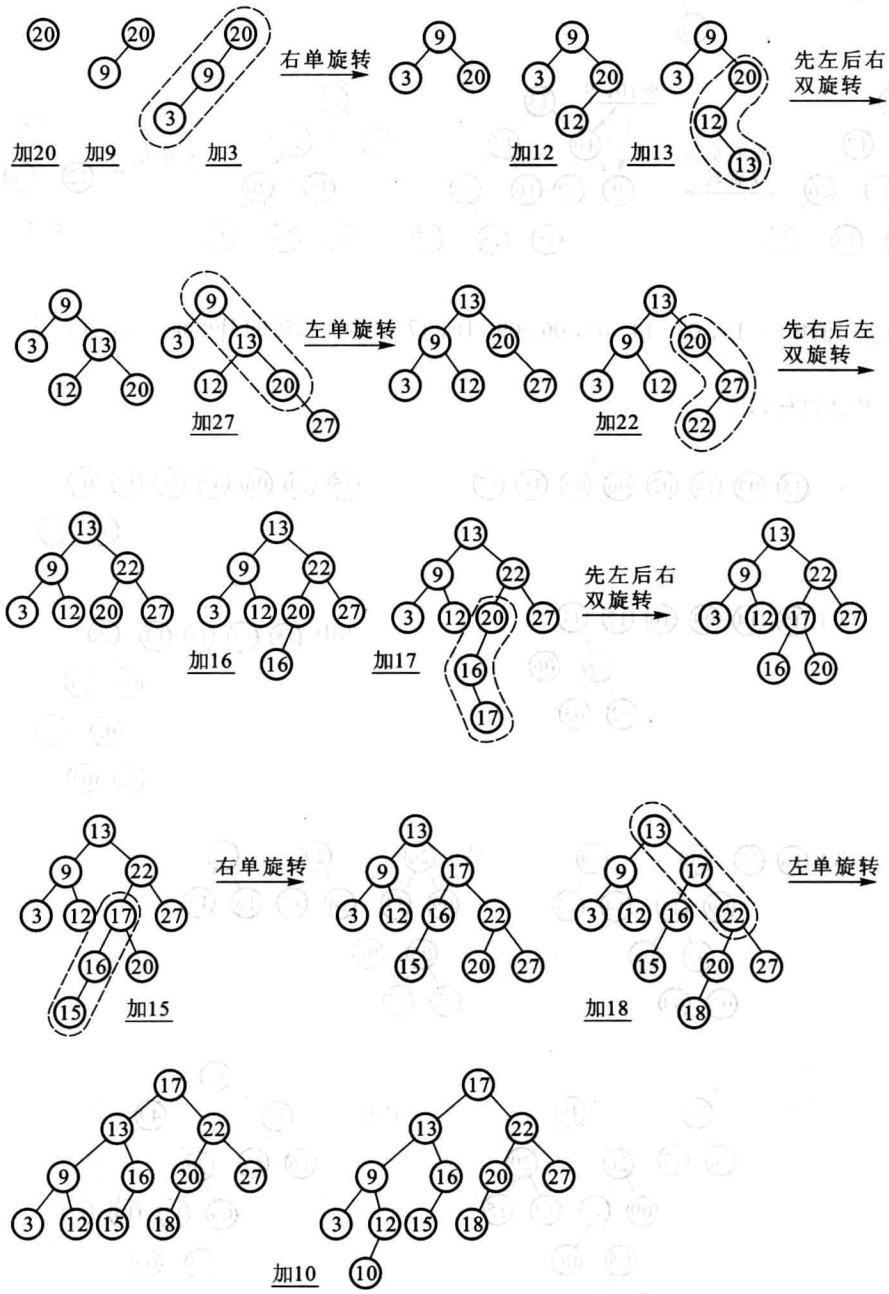
例 4 针对二叉树 BiTree，利用二叉树遍历的思想编写解决下列问题的递归算法。

旋转、先右后左双旋转。

（1）若关键字的输入序列为20，9，3，12，13，27，22，16，17，15，18，10，试从空树开始顺序输入各关键字建立平衡二叉树。画出每次插入时树的形态。若需要平衡化旋转则做旋转并注明旋转的类型。

（2）基于上面建树的结果，画出从树中删除22，删除3，删除10与9后树的形态和旋转的类型。要求以被删关键字的中序下的直接前驱替补该被删关键字。

【解答】（1）插入过程



此树的带权路径长度 $WPL=229$。

同步练习

一、单项选择题

1. 在下列关于二叉树遍历的说法中错误的是____。

A. 在一棵二叉树中，假定每个结点最多只有左子女，没有右子女，对它分别进行前序遍历和后序遍历，则具有相同的遍历结果

B. 在一棵二叉树中，假定每个结点最多只有左子女，没有右子女，对它分别进行中序遍历和后序遍历，则具有相同的遍历结果

C. 在一棵二叉树中，假定每个结点最多只有左子女，没有右子女，对它分别进行前序遍历和按层遍历，则具有相同的遍历结果

D. 在一棵二叉树中，假定每个结点最多只有右子女，没有左子女，对它分别进行前序遍历和中序遍历，则具有相同的遍历结果

2. 在下列关于二叉树遍历的说法中正确的是____。

A. 若有一个结点是二叉树中某个子树的中序遍历结果序列的最后一个结点，则它一定是该子树的前序遍历结果序列的最后一个结点

B. 若有一个结点是二叉树中某个子树的前序遍历结果序列的最后一个结点，则它一定是该子树的中序遍历结果序列的最后一个结点

C. 若有一个叶子结点是二叉树中某个子树的中序遍历结果序列的最后一个结点，则它一定是该子树的前序遍历结果序列的最后一个结点

D. 若有一个叶子结点是二叉树中某个子树的前序遍历结果序列的最后一个结点，则它一定是该子树的中序遍历结果序列的最后一个结点

3. 前序为 A、B、C，后序为 C、B、A 的二叉树共有____。

A. 1 颗　　B. 2 颗　　C. 3 颗　　D. 4 颗

4. 在一棵非空二叉树的中序遍历序列中，根结点的右边__1__；设 n 和 m 分别是一棵二叉树上的两个结点，在中序遍历时，n 在 m 前面访问的条件是__2__；

(1) A. 只有右子树上的所有结点　　B. 只有右子树上的部分结点
　　C. 只有左子树上的所有结点　　D. 只有左子树上的部分结点

(2) A. n 在 m 右侧分支　B. n 是 m 祖先　　C. n 在 m 左侧分支　　D. n 是 m 子孙

5. 设结点 x 和 y 是二叉树中任意的两个结点。在该二叉树的前序遍历序列中 x 在 y 之前，而在其后序遍历序列中 x 在 y 之后，则 x 和 y 的关系是____。

A. x 是 y 的左兄弟　　B. x 是 y 的右兄弟　　C. x 是 y 的祖先　　D. x 是 y 的后裔

6. 结点数目为 $n(n\geqslant 0)$ 的二叉排序树的最小高度为__1__、最大高度为__2__。

(1) (2) A. n　　B. $\frac{n}{2}$　　C. $\lfloor \log_2 n \rfloor+1$　　D. $\lceil \log_2(n+1) \rceil$

7. 在常用的描述二叉排序树的存储结构中，关键字值最大的结点____。

A. 左指针一定为空　　B. 右指针一定为空
C. 左右指针均为空　　D. 左右指针均不为空

8. 在下列关于平衡二叉树的说法中正确的是____。

A. 任意结点的左、右子树结点数目相同
B. 任意结点的左、右子树高度相同
C. 任意结点的左、右子树高度之差的绝对值不大于 1
D. 不存在度为 1 的结点

9. 具有 5 层结点的平衡二叉树至少有__1__个结点。若设树根结点在第 1 层，则深度最小的叶结点应