《数据结构》

第五版

清华大学自动化系
李宛洲
2004 年 5 月

目录

第一章 数据结构--概念与基本类型

1.1 概述

1.1.1 数据结构应用对象

计算机应用可以分为两大类，一类是科学计算和工业控制，另一类是商业数据处理。相应的计算机语言也是如此，比如 FORTRAN 语言、C、汇编语言主要适应于前者，比如 JAVA、Powerbuilder(关系数据库平台开发工具)、Visual C 等主要适应于后者。面向工业控制与科学计算的内容主要涉及它的计算方法、效率与速度等因素，某一特定的测控对象有特定的算法，在这里我们主要侧重于解决问题的方法研究，比如高次方程的叠代算法，快速富氏变换的蝶型算法等。面向商业管理是要解决海量数据的管理与关联分析，即使是一个特定的对象也有通用的数据管理形式，比如商业数据库系统，无论何种具体应用，它都是大量的表格一类的数据处理形式，在海量数据中检索与查询是一类至关重要的操作工具，于是，数据的逻辑结构与物理组织形式是我们要解决的主要问题，比如表数据的存储形式，索引结构等，也就是数据结构问题。

什么是数据结构？数据结构的研究对象是数据元素，目的是建立数据元素在计算机中的表达方法，简单的说，在一群有限的数据元素集合里，元素与元素之间相互关系的描述，称为它的数据结构。比如，例 1.1 描述了有限个数据元素集合的字典的数据结构关系。

例 1.1 字典的数据结构

D={(able,能干的),(apple,苹果), (bug,虫), (code,代码), (cool,酷),..., (x-ray,X 光),(year,年),(zoo,动物园)}

这里，单词是数据元素检索关键字，单词与注释构成数据元素（节点），元素节点之间所表达的关系是按字母的顺序排列，这就是我们给字典这一特定对象选定的数据结构。另一个例子 1.2 描述了事务处理中经常见到表格的数据结构形式。

例 1.2 线性表数据结构

表 1.1 设备统计清单

序号	设备名称	型号	单价（元）	数量
1	车床	A64	5500	5
2	台钻	C7	3200	29
3	铣床	X－2	4000	14
4	铣床	X－34	6700	1

如表 1.1 所示设备统计表是一种线性结构，为了把一个线性表转换成可以用计算机处理的形式，或者说选择表在计算机中的数据结构形式，需要采取如下步骤：首先，水平方

向看表的每一行是一条记录，我们称之为向量 a_i，a_i=（序号，设备名称，型号，单价，数量），a_i 的各分量是设备这一客观实体的属性，属性的取值就是实体记录，所以，从纵向看，表是成由一组记录所组成的，记录是表的数据结构元素，定义如下：

```
struct  BILL{
                char    Facility[20];
                char    Type[10];
                int     Cost;
                int     Number;
                };
```

表结构表达的记录（节点元素）之间的关系是<a_i，a_{i+1}>，所以我们称表结构是线性的，可以用 C 语言的数组变量定义相应的数据关系为：

```
                        struct  BILL    a[4];
```

同所有的数组变量一样，结构数组的下标也是从 0 开始的。因此，在计算机中可以用 BILL 结构变量型数组 A[]来描述表 1.1 所表达的关系，也就是线性表的数据结构形式：

a_0=（1，车床，A64，5500，5）

a_1=（2，台钻，C7，3200，29）

a_2=（3，铣床，X-2，4000，14）

a_3=（4，铣，X-34，6700，1）

1.1.2 学习数据结构的基础

数据结构建立在计算机语言之上。学习计算机语言是学习编程方法，我们应该如何用一种具体的计算机语言实现一个算法。学习数据结构，是学习如何描述一个应用对象的数据元素（属性构成），如何根据应用对象的特点构造数据元素之间的逻辑关系以及内存中的存储实现，这是二者的区别。

设计数据结构的时候要有相应的计算机语言工具支持，在 BASIC、FORTRAN、C 语言中，只有 C 是面向数据结构应用的工具语言。比较一下 C 和其它语言的区别就可以知道原因，因为它有定义数据结构基本单元的能力，并有地址的运算能力，这两点是非常重要的。通过定义数据结构的基本单元，我们可以把不同数据类型的变量聚集在一个节点内；通过地址运算，我们可以把数据结构的逻辑关系在计算机内存中用不同存储方式实现。在 C 语言中定义数据结构元素是通过结构体实现的。

1.1.2.1 C 语言中的结构体

在学习 C 语言的时候，同学对数组很熟悉，比如一个整型量的数组定义如下：

```
int array[100];
```

它表达了一组整型量的集合，在 C 语言中基本变量的类型有整型量，浮点变量，字符变量等，将所有基本变量聚合在一起的方法是定义结构体，用结构体作为基本元素描述事物的属性信息，比如表 1.1 那样，我们称之为数据结构元素，或者节点。关于数据结构元素在 C 语言中给出了明确定义：

结构元素是一种被命名为一个标识符的各种变量的集合，是提供将各种基本数据类型汇集到一块的手段，它提供了结构变量的格式。

比如一个电话簿的结构元素如下定义：

```
struct  ADDER{
                char    Name[20];
                char    Street[40];
                char    City[20];
                char    STATE[2];
                unisgnedlong    Zip;
                };
```

通过结构体定义，ADDER 结构变量代表了一组基本数据类型的聚合结构，它就是所谓数据结构的基本单元，我们定义，数据结构就是描述这样一组结构变量之间关系的形式，例如：

```
struct  ADDER   adder_info[100];
```

给出了结构变量 ADDER 的数组结合形式，是一种线性关系数据结构。

1.1.2.2 C 语言的指针在数据结构中的关联作用

结构化的程序模块和指针的应用是 C 语言程序设计的基本风格，随着 BC 和 VC 的出现，面向对象的程序设计方法以及多线程技术给我们提供了在 Windows 平台上开发应用软件的多样化风格，但是，指针的应用依然是我们程序设计最基本的特征。

指针是地址，它指向数据变量在内存中的地址，请同学牢牢记住这个概念，我们之所以对指针理解的非常容易混淆，是因为我们没有把指针的概念与变量的存储位置关联在一起进行考虑。请同学清楚下面几点：

- 指针的值是地址；
- 任何一个变量都有一个地址，变量类型不同占用的地址单元数量不同；
- 指针也是一个变量，所以它也有地址；
- 给指针赋值是让指针变量指向与其同类的一个数据变量的地址；
- 没有赋值的空指针其指向不确定，所以绝对不能在程序中使用；

程序中定义任何一个名称的变量都对应着一个物理地址，因为需要保存变量值在内存该地址单元内，比如：

```
char name[20];        //编译程序分配地址单元
scanf("%s",name);     //给变量 name 赋值
```

一个变量在内存占用的地址单元多少由变量的类型决定，比如，字符型变量占用一个字节，整型量占用 2 个字节，浮点型占用 4 个字节等，而一个结构元素占用的内存字节数由它所聚集的基本变量类型及数量决定。

指针是程序中定义的，因而指针也是一个变量，为了区别数据与地址的关系，我们将元素变量称之为数据变量，称指针为地址变量，所以指针也需要保存，指针本身也有地址：

```
char *cp,name[20];   //编译程序给指针变量 cp 分配地址单元
cp=name;             //给指针变量 cp 赋值，让它指向数据变量 name
```

- C 程序中指针的用法

指针变量的基本概念是地址，它用地址运算符取得某一数据变量在内存的地址，从而指向了该变量。指针存储着一个数据变量的地址，既然不同类型的数据变量占用的存储单元数不同，所以，指针变量的类型应该与其所指向的数据变量同类，也就是有整型量指针，字符型指针，浮点数指针和结构体指针，这样，在变量集合内，指针加一操作时所跨过的地址单元数，是该类数据变量占用的内存单元长度，从而能正确的指向下一个变量位置。指针如下操作得以指向一个数据变量：

```
int val=10,y,*p;
p=&val;
y=*p;
*p=20;
```

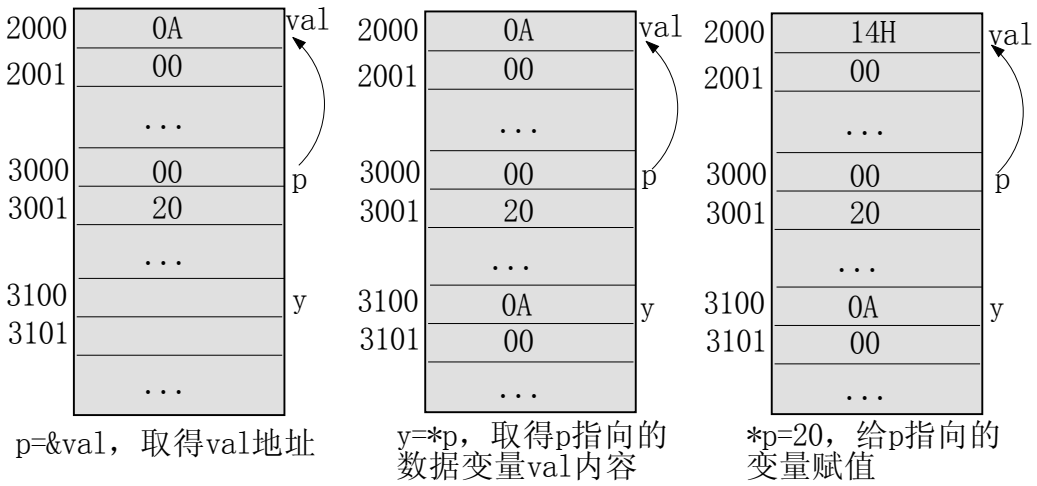


图 1.1 指针应用—指针 p 指向变量，操作指针等于操作变量

我们首先定义了整型数据变量 val、y 和整型指针变量 p，第二条语句让指针 p 取得了 val 的地址，即指针 p 指向了变量 val，第三条语句将指针所指向的变量 val 的值赋给了变量 y，第四条语句将指针 p 指向的变量 val 的值修改为 20，见图 1.1(a)。于是，对指针的操作等价于对变量的操作，程序变得非常简洁，比如下面程序对数组进行线性赋值：

```
int array[100],*p, i;
p=&array[0];
for(i=0;i<100;i++)*(p+i)=i;
```

切记，一定不能给一个没有值的指针，也就是空指针赋值，也不能给指针任意赋一个值，比如零，图 1.2 显示了给一个空指针置数的结果，一般 0000H 是计算机操作系统保留区域，比如是软中断引导程序的入口地址，假设你给指针指向的地址单元赋值，那就是说你破坏了系统程序入口地址，如果编译系统没有检查功能，你的程序运行时候将破坏整个计算机系统运行状态。如果指针的值是任意一个随机数，它可能指向任何可能的应用程序正在使用的数据区或者栈区域，你的赋值操作就破坏了该应用程序，比如说它的返回地址。

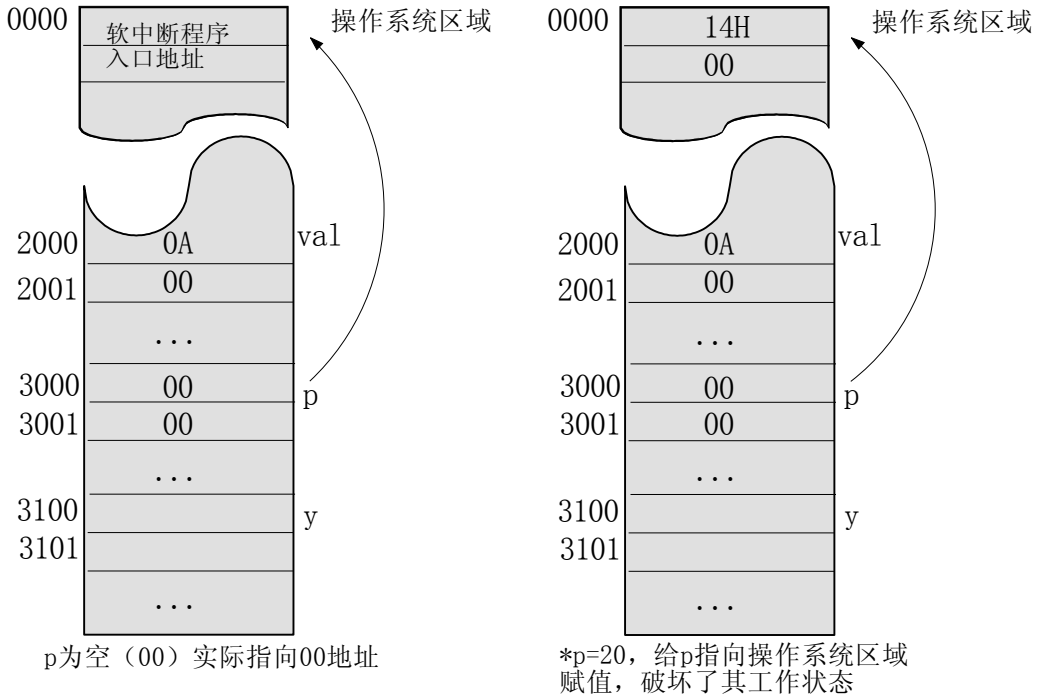


图 1.2 对空指针赋值

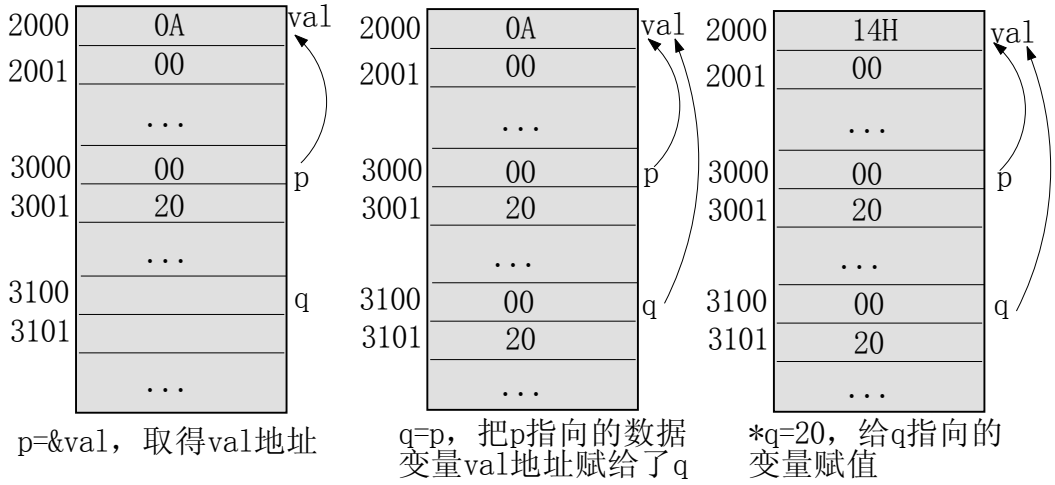


图 1.3 地址传递

指针的另一种用法是地址的传递，数据结构中经常将一个指针的值（某一节点元素的

地址）传递给另一个指针，比如，图 1.3 表示了如下程序段的执行结果。

```
int val=10,*p,*q;
p=&val;
q=p;
*q=20;
```

另外，为数据节点申请内存空间时，用指针指向调用函数返回的节点地址：

```
p=(struct node *)malloc(sizeof(node)); //p 是指针变量
```

- 指针在数据结构中的关联作用

指针在数据结构起到关联节点的作用，让指针从一个节点元素指向另一个节点元素，换句话说，通过指针连接节点元素之间的存储位置，从而让它们之间关联在一起，进而表达了它们之间的逻辑关系。

让指针从一个节点指向另一（或者是多个）节点，需要在节点定义中加入指针变量，指针在节点内，它指向下一个节点，如果你能找到当前节点位置，就能根据指针找到后续节点所在，这就是节点关联。现在讨论如何用指针关联两个节点元素，我们看例 1.3。

例 1.3 用节点内部指针关联两个节点。

如下一个学生数据节点的定义：

```
struct student_node{
        int number;
        char name[40];
        char gender;
        struct node *student_next;
        };
```

在这个结构体内，我们不但提供了描述学生个体属性的基本变量聚合，而且还有该节点类型的指针变量 next，用 next 可以指向学生集合中的其它个体或者说是节点，从而表达了集合中节点之间的关系，使它们关联在一起。比如，设指针 head 已指向内存里的一个节点 a_1，当再申请一个节点比如 a_2 时，通过对 a_1 的 next 赋值使其指向 a_2，从而让 a_1 与 a_2 关联起来，如图 1.4 所示。方法实例如程序 1.1。

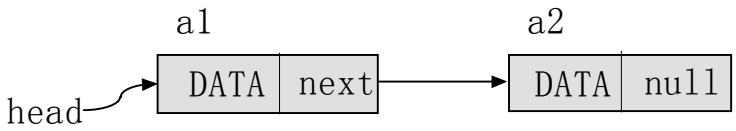


图 1.4 节点关联

程序 1.1

```
#include<stdio.h>
#include<malloc.h>
```

```
#define null 0
int main(void)
{
    struct student_node{
                    char number[20];
                    char name[40];
                    char gender;
                    struct student_node *next;
                    }*q,*head;
    q=(struct student_node *)malloc(sizeof(student_node));//节点a1
    head=q;                         //head指向a1
    q=(struct student_node *)malloc(sizeof(student_node));//节点a2
    printf("请输入名字\n");
    scanf("%s",q->name);     //给naem赋值，输入名字
    printf("请输入学号\n");
    scanf("%s",q->number);       //给number赋值，输入学号
    q->next=null;
    head->next=q;                   //给a1的指针赋值，a1的指针指向a2
    printf("节点a2记录内容是:");
    printf("%s  %s\n",head->next->number,head->next->name);//打印节点a2输入的名字
    return(0);
}
```

程序运行结果：

请输入名字

张三

请输入学号

2003w1234

节点a2记录内容是：2003w1234 张三

1.1.2.3 C语言的共用体(union)数据类型

在链表和树结构中往往要求内部节点与外部（比如树权是内部节点而叶子是外部节点）同构，处理的方法是c语言中的共用体(union)数据类型，这里简要的回顾共用体的概念。

- 共用体说明和共用体变量定义

  共用体是一种数据类型，它是一种特殊形式的变量。共用体说明和共用体变量定义与

结构十分相似。其形式为:

```
union 共用体名{
    数据类型 成员名;
    数据类型 成员名;
    ...
} 共用体变量名;
```

共用体表示几个变量共用一个内存位置，在不同的时间保存不同的数据类型和不同长度的变量。下例表示说明一个共用体 a_bc:

```
union data {
        int i;
    char ch;
        float f;
    };
```

再用已说明的共用体可定义共用体变量。例如用上面说明的共用体定义一个名为 lgc 的共用体变量可写成:

```
union data lgc;
```

共用体变量 lgc 中整型量 i、字符 ch 以及浮点变量 f 共用同一内存区域，如图 1.5 所示。因此，对三个变量中任何一个变量的赋值操作，都会影响其余变量的值。

图 1.5 共用体内变量共用内存的同一个区域

当一个共用体被说明时，编译程序自动地产生一个变量，其长度为共用体中最大的变量长度。共用体访问其成员的方法与结构相同。同样共用体变量也可以定义成数组或指针，但定义为指针时也要用"->"符号，此时共用体访问成员可表示成:

```
共用体名->成员名
```

共用体也可以出现在结构内，图 1.6 描述了下述结构定义的变量之间关系:

```
struct {
    int age;
    char sex;
    union {
        int i;
        char *ch;
        }x;
    }y;
```

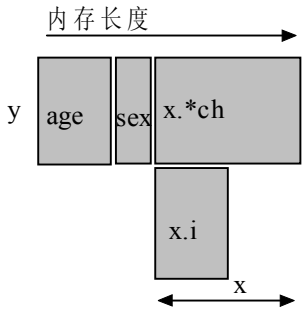


图 1.6 结构与共用体关系

若要访问结构变量 y 中共用体 x 的成员 i，可以写成:

y.x.i;

若要访问结构变量 y 中共用体 x 的字符指针 ch 所指向的内容，可写成:

*y.x.ch;

若写成"y.x.*ch;"是错误的。

- 结构和共用体的区别

1.结构和共用体都是由多个不同的数据类型成员组成，但在任何同一时刻，共用体中只存放了一个被选中的成员，而结构的所有成员都存在。

2.对于共用体的不同成员赋值，将会对其它成员重写，原来成员的值就不存在了，而对于结构的不同成员赋值是互不影响的。因此，共用体中的指针操作需要特别小心，它很容易被误操作。作为加深对共用体理解的例子，请读者参见程序 1.2。

程序 1.2 共用体的应用

```
#include <stdio.h>
int main(void)
{
    char a;
    union {                    //注意和图 1.6 的区别
        int age;
        char sex;
        struct {
            int i;
            char *ch;
            }x;
        }y;
    y.age=10;
    y.x.i=20;                  //覆盖了 y.age，注意高位是 0
    y.sex='b';                 //覆盖了 y.x.i 的低位，'b'=98
```

```
    y.x.ch=&a;                    //y.x.ch 在地址上与共用体内其它变量无关
    *(y.x.ch)='a';
    printf("y.x.i=%d\n",y.x.i);
    printf("y.age=%d\n",y.age);
    printf("y.sex=%c\n",y.sex);
    printf("*(y.x.ch)=%c\n",*(y.x.ch));
    return (0);
}
```

运行结果：

```
y.x.i=98
y.age=98
y.sex=b
*(y.x.ch)=a
```

从程序 1.2 结果可以看出，当给 y.sex 赋值后，也就是 y.age 和 y.x.i 的其低八位值置成字符‘b’的 ASCII 码 98，这两个字的高八位都是零。

1.1.3 数据结构定义

不但指针，数组也是 C 语言中提供的聚合某类元素的工具，它所表达的关系是相邻元素之间只存在一种线性有序关系$<a_i, a_{i+1}>$，如例 1.2 的表格数据结构，就使用了数组形式。因为数组提供的数据元素之间的关系只能是线性相邻的，往往不能满足现实中具有非线性关系的对象需要，因此，必须借助指针来表达复杂逻辑结构的数据关系。比如，现实中还可以举出含有多种关系存在的数据结构，设一批数据元素的逻辑结构是：

$K=\{K_1, K_2, \ldots, K_{10}\}$ //一组元素

$R=\{r_1, r_2\}$ //一组关系

$r_1=\{<k_1, k_2>, <k_1, k_8>, <k_2, k_3>, <k_2, k_7>, <k_3, k_4>, <k_3, k_5>, <k_3, k_6>, <k_8, k_9>, <k_8, k_{10}>\}$

$r_2=\{<k_{10}, k_4>, <k_4, k_2>\}$

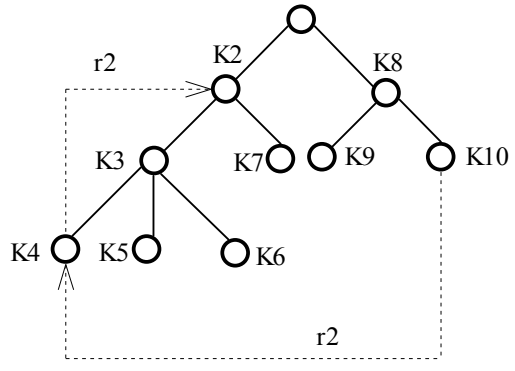


图 1.7 2 元关系的逻辑结构图示法

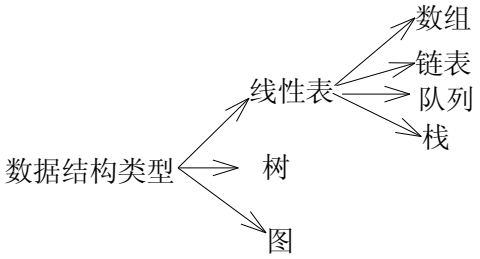


图 1.8 基本的数据逻辑结构类型

这是有两元数据关系的逻辑结构，分别用实、虚线表示如图 1.7 所示。关系 r_1 表示了一种树形逻辑关系，r_2 表示了（k_{10},k_4,k_2）的顺序相邻关系。树形结构是非线性的，当然不能用线性的数组关系描述，所以必须借助指针。指针既能表达线性相邻也能表达非线性分支，这是数据结构使用指针的原因所在。现在我们给出数据结构的定义：

定义 1.1：数据结构是一个二元组集合 S＝（D，R），其中 D 是结构变量的非空有限集合，R 是描述在 D 上的有限个关系的集合。

所以，我们说数据结构研究的是客观事物个体属性在计算机中表达及描述的方法。在节点元素中，用计算机语言的基本变量的聚合，刻画了事物的客观属性，指针或数组的地址连接，则描述了节点之间的关联关系。学习数据结构的内容主要是 3 点：

① 数据结构的逻辑结构。根据应用对象设计有限元素集合中节点之间的逻辑关系，如，线性表、树、图等；

② 逻辑结构在计算机中的物理实现。如顺序表、链表、二叉树等；

③ 数据结构中节点的操作运算。如，插入、删除、检索等。

图 1.8 给出了几种基本的数据结构形式。要设计应用于计算机处理的数据结构形式，上述的定义必须联系于计算机的物理实现才有实际意义。

数据结构在计算机内存中的表示方法，我们称为数据结构的物理结构，以区别于前者的逻辑结构形式。物理结构有四种基本的形式，见图 1.9 所示，其中，索引结构用于文件操作，散列结构是对数据检索时采用的一种形式。

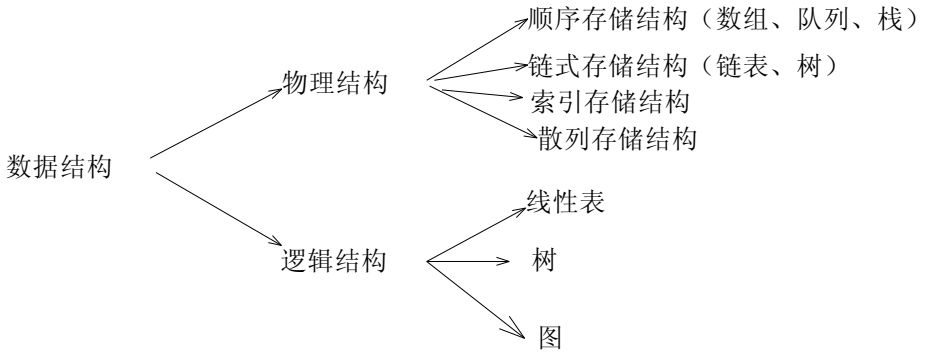
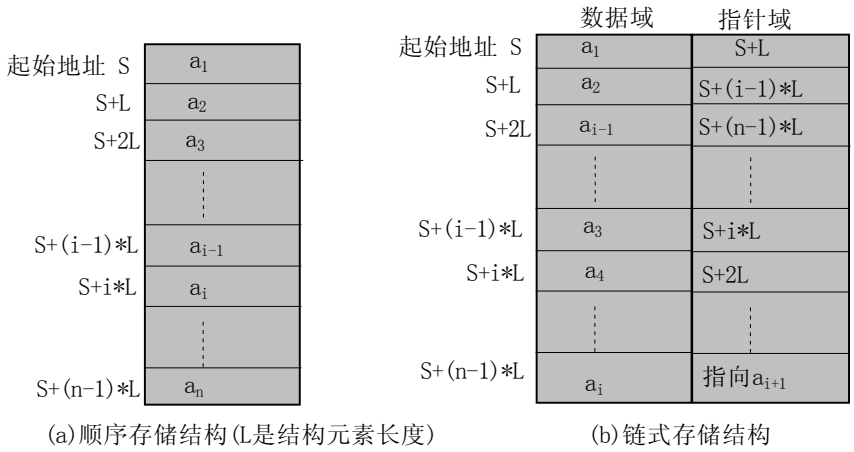


图 1.9 基本的数据物理结构类型

所谓顺序存储结构，是指将数据元素顺序的存放于计算机内存中一个连续的存储区域里，借助元素在存储器中的相对位置来表示元素之间的逻辑关系，也就是用数组描述的一群有限数据元素集合。

链式存储结构的特点，是在每个元素中加入一个指针域，它指向逻辑上相邻接的元素的存放地址。而数据元素在内存中的存放顺序与逻辑关系无关。即链式存储结构是用指针的指向来表达节点的逻辑关系，这也就是 C、Pascal 适用于数据结构设计的原因。图 1.10 给出顺序、链式存储结构示意。它们都是描述，或者说存储了线性关系$<a_i, a_{i+1}>$，但方式不同。



(a)顺序存储结构(L是结构元素长度)　　(b)链式存储结构

图 1.10 向量的顺序存储结构与链式存储结构

数据结构有线性与非线性之分。一个数据结构的关系里，除去端点外，每一个节点有且仅有一个前趋和后继时，这个数据结构它是线性的。如数组、链表。如果数据结构关系中，其节点有一个以上的前驱或后继，则称为非线性的数据结构。如树、图。一般情况下，我们讨论非空有限集合 D 上只有单一关系 r 的数据结构。但是，在关系数据库设计时，讨论的则是非空有限集合 D 上的一组关系 R 的数据结构设计问题。

数据结构的物理表达问题，在有关参考书已经明确给出，读者可以仔细阅读理解，对 C 语言不熟悉的读者要尤其注意指针和链式存储结构。在我们的课程中，线性表、树是学习的重点，而线性表中链表设计是重点内容。它应用了指针的概念，在 C 语言设计中，掌握了指针的应用就掌握了 C 语言的设计风格，对有关指针和指针型函数还有不清楚地方的需要复习。图的内容不在本教材讨论范围。排序、检索在数据结构之后学习。

在结束有关数据结构的概念讨论之前，我们再次明确的给出数据结构内容的三要素是：

数据结构 ＝数据逻辑结构 ＋物理结构＋数据运算

数据运算是指对数据结构的检索、插入、排序、删除、更新等操作。此外，不同数据结构之间的运算效率也是我们要重点考虑的内容。

1.2 线性表

线性表是我们接触的最基本、最普通的一种数据结构，我们给出它的定义：**定义 1.2：一个线性表是数据元素的有限序列{$a_1, a_2, a_3, \cdots, a_n$}，其中 $a_1, a_2, a_3, \cdots, a_n$ 是结构元素，下标是元素序号，它的数据结构表示是：**

Linear_list=(D,R)，$R = \{\langle a_i,\ a_{i+1}\rangle | a_i,\ a_{i+1} \in D \qquad D = (a_i, i = 1,2,\ldots,n)\}$

这里，关系 R 给出了元素的一种先后次序：a_1 为表起点，a_n 为表终点，除第一元素之外，表中每一元素只有一个前趋，除最后一个元素之外，表中每一元素仅有一个后继。

根据线性表在内存中的存储方式不同，我们分为顺序存储结构的顺序表和链式存储结构的链表。顺序表由元素在内存中顺序存储的相对关系来表达逻辑上的线性有序相邻关系。

链表则是用指针的指向来表达这种逻辑上的相邻关系，它在物理上是非顺序排列的。从图 1.10 中可以看出它们的共性与特性，即用不同的物理存储结构表达同一种线性逻辑关系。

1.2.1 顺序表

顺序表是顺序存储结构的线性表，其结构已在图 1.10 中描述过，就是用数组定义的数据元素集合、以及元素之间顺序相邻的关系。顺序表内相邻数据元素的物理地址也相邻，因而物理存储关系表达了逻辑顺序相邻的关系。

顺序表结构主要方便实现一些简单的数据操作，比如对半检索等。顺序表的特点是简单，它的主要缺点是需要程序预先定义线性表结构，占用固定的内存空间。这在大多数应用中非常不方便。比如，一个学校的学生关系数据库，有关学生信息的数据纪录条目随着年代的增加而增加，你不可能预先设定一个最大存储记录的上限，那样非常不便。

一般说，顺序存储结构和链式存储结构的区别主要体现在数据结构的操作效率上。现在讨论顺序表在插入与删除运算方面与链表的区别点。设有长度为 n 的顺序表：

$$\{a_1, a_2, a_3, \cdots, a_n\}$$

在第 i 个元素前插入数据 a_b，如图 1.11 所示，即：

$$\{a_1, a_2, a_3, \cdots a_b, a_i, \cdots, a_n\}$$

为此，我们需要移动数组中的元素以腾出一个空位插入 a_b 元素。显然，在包含 a_i 元素一块移动时，移动元素个数 j 是:

$$j=n-i+1 \qquad i=1，1，2，...，n$$

如果在表中任一位置插入元素的概率相等，则有 $p_i = \dfrac{1}{n+1}$，其平均移动次数是：

$$E = \frac{1}{n+1}\sum_{i=1}^{n}(n-i+1)$$

$$= \frac{n}{2}$$

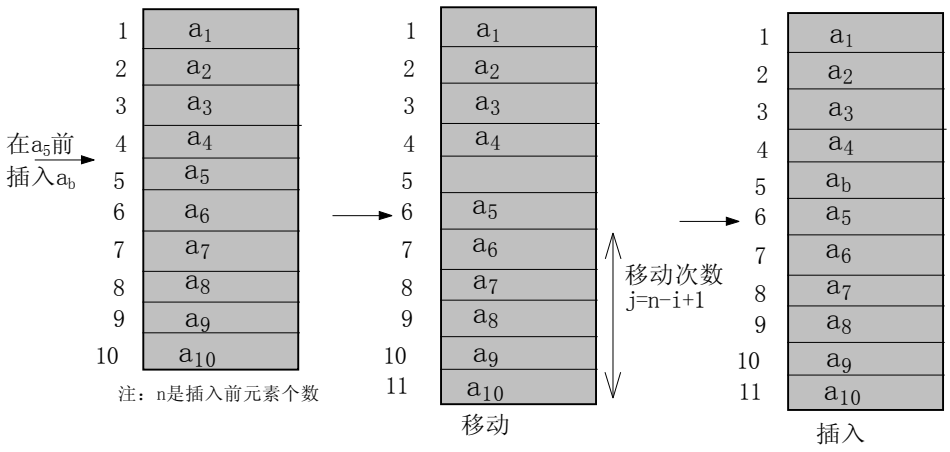


图 1.11 顺序存储的插入操作

同样，在删除 i 节点上元素时，需要顺序移动其后面的元素序列补充这个位置，在顺序表中元素的移动个数是 j=n-i，所以，其平均移动次数是：

$$E = \frac{1}{n} \cdot \sum_{i=1}^{n} (n-i)$$
$$= \frac{n-1}{2}$$

因此，当 n 很大时，顺序表插入与删除运算所占用的时间很多，为 0（n）时间复杂度（所花费的时间与规模 n 成线性关系）。一般说顺序表在读取元素时效率高（是随机存储结构），而在插入与删除时效率比较低。

顺序存储结构可以描述线性表，但并不是说顺序存储结构只能描述线性表。在一个特定应用对象时，如果考虑节省存储空间的目的，我们往往也用顺序存储结构描述一棵树形结构，如果读者认为顺序存储结构很简单，没有什么概念可做的话，请看下面例子。

例 1.4. 树的顺序存储结构。设一棵树如图 1.12 所示，问如何用顺序存储结构以最小空间存储在内存中？

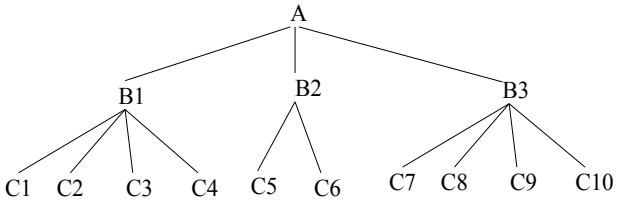


图 1.12　一棵树结构

解：

首先分析题意，它的实质是如何在顺序存储结构中描述出一个元素所具有的多个后继节点的关系，换句话说，我们要用一个序列来表达这棵树的每一层元素、以及各层元素之间的关系，最简单的想法是按层次结构写出这个序列：

$$\{AB_1B_2B_3C_1C_2C_3C_4C_5C_6C_7C_8C_9C_{10}\}$$

显然，这样只能限定在这颗特定的树结构上，不具有存储树形结构的普遍意义，因为除非事先指定，否则你不能在程序中区分每一层元素的边界。我们仔细分析一下可以发现，树是一个递归的结构，根有分叉节点，分叉节点与它的后继仍然形成一棵子树，直到叶子，所以，我们如果能表达出第一层根和后继节点的关系，就可以递归的用这个形式结构表达下去，设用“(”边界符表示一层边界的起点，“)”表示终点，于是树形结构的元素展开序列为：

$$\{A\ (B_1(C_1C_2C_3C_4)B_2(C_5C_6)B_3\ (C_7C_8C_9C_{10}))\ \}$$

因此，树节点的顺序存储结构是：

A	(	B1	(	C1	C2	C3	C4	)	B2	(	C5	C6	)	B3	(	C7	C8	C9	C10	)	)

程序中，每遇到一个左括弧表示一层节点的开始，每遇到一个右括弧表示最近与其相配的左括弧代表的那层节点结束。（题毕）

在数据结构中，用顺序存储结构实现的还有队列、栈等，因为它们都是线性表的一种，

所以也可以称为顺序表，但是操作上有其特殊性，它们将在后面内容中继续予以讨论。

1.2.2 链表

链表相对来说是一个新的内容，也是学习数据结构的入门。我们首先给出它的基本概念，然后重点讨论链表设计问题。

1.2.2.1 链表的基本结构及概念

在线性数据结构中为什么使用链式存储结构？前面说过，如果事先知道一个文件占用内存空间的大小，那么用数组描述（顺序存储结构）是最简洁的。但是，大多数应用例子表明无法预先给定一个文件记录数的上限，那样非常不经济。最适宜的处理方法是仅在输入一条文件记录时，向计算机内存管理系统申请一个记录节点所需的空间。系统根据内存当前占用情况进行动态地址分配。然后程序根据所得到的地址，输入这条纪录的数据到这个地址单元区域内，并把这条记录的地址连接到文件当前纪录的末端，使所有纪录串起来形成一个链。

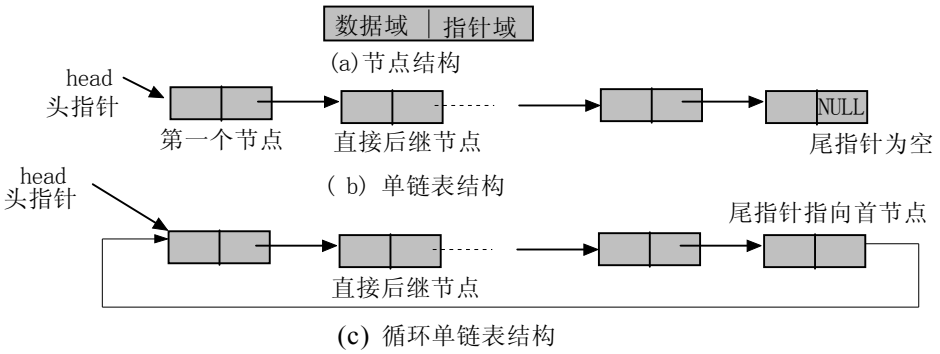
只要内存未满（一般情况下可以假设计算机硬盘足够的大），并且程序能正确的连接文件所有记录的地址，则文件记录长度可以动态生长，当然也可以动态删除，而连接所有纪录地址的方法就是指针。

如何用指针链接各个数据节点？让我们回顾例 1.3 描述的指针关联作用，定义在节点内部有一个指针域，只要初始有一个头指针 head，并让它指向头节点 a_1,那么，通过把每次输入纪录 a_i 的地址赋值给其前驱节点 a_{i-1} 所含的指针 next，就可以让 a_{i-1} 指向 a_i：

$$a_{i-1}\text{->next}=a_i;$$

它描述了<a_i, a_{i+1}>的逻辑关系。

线性表的链式存储结构特点是用内存中随机分布的存储单元来存储线性表的数据元素，而关系<a_i, a_{i+1}>，是用节点 a_i 指针域所含的后继节点 a_{i+1} 地址信息来表达的。即，节点分为数据域与指针域两部分，如图 1.13（a）所示。



(c) 循环单链表结构

图 1.13 单链表结构

所有节点指针的指向形成了一条数据链。图 1.14 给出了链表的基本类型，图 1.15 是双链表结构。

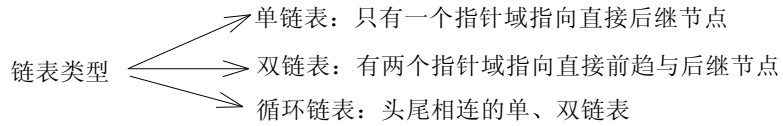


图 1.14 基本的链表数据结构分类

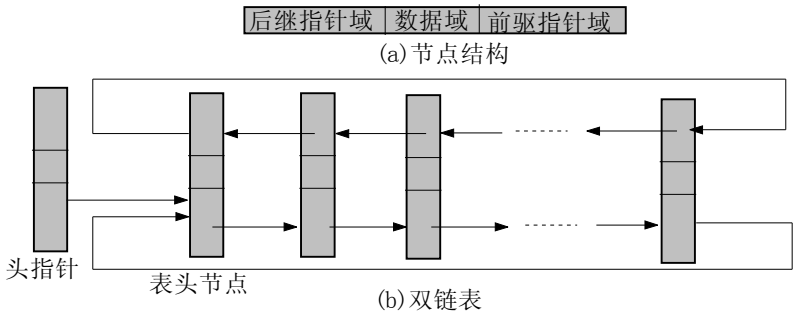


图 1.15 双链表结构

我们要注意头指针的作用，空表时指针亦为空。此外，单、双链表的循环结构和非循环结构没有本质的区别。程序设计时，有时也用尾指针来取代头指针，详细见后述例题。有的参考书设置有专门的头节点，在它数据域设置一个特定的赋值，不把它当成链表中的一个节点，于是链表为空时仍有头节点存在，这取决于具体应用例子的程序设计技巧问题。

关于节点结构，主要在于理解指针的概念。这里，它表达了线性逻辑关系在存储器里的映象，或者说是逻辑结构在内存中的存储实现。所以，链表的节点结构定义中含有一个附加的指针域。它的 C 语言描述是在节点定义中增加一个指针变量，如前描述的 BILL 类型节点，在单链表节点定义时是：

```
struct  BILL{
                char    facility[20];
                char    type[10];
                int     cost;
                int     num;
                struct  BILL    *next;
                };
```

指针 next 与要指向的后继节点同构，所以它的类型必须是 BILL 同类。同样，双链表节点的定义是：

```
struct  BILL{
                char    facility[20];
                char    type[10];
                int     cost;
                int     num;
                struct  BILL    *next, *pri;
                };
```

链表设计中最为困惑的是指针运用。包括地址传递和初始化指针等，特别是指向指针的指针这一概念。比如，图 1.16 是使用尾指针的循环链表应用于动态存储器管理的例子。

计算机的内存管理系统负责管理存储器中当前剩余的空闲区域，该区域被分成一组空闲内存块的形式，形成如图 1.16（a）所示的一个循环链结构。它的左边是一个用户正在使用存储块，也被串成一个循环链表结构。假设现在该用户退出计算机，系统要将该用户正在使用的全部内存块释放，因为我们有一个尾指针总是指向用户占用链的最后一个内存块节点。于是，用户内存块全部收回的操作只需要把系统链的栈指针与尾指针所指向的尾节点指针内容相交换即可。

```
q=sp;              //暂存
sp=rear->next; //取得首节点
rear->next=q;      //指向原空闲块 1
```

链表概念中最重要的是一个动态的生长过程，即链表的长度在程序中是动态的生长与消亡的，表明了链表设计中的灵活性所在。数组定义时必须指明它的长度，如果事先不知道输入节点数是多少，可选择链表结构，只有在存入记录时向系统申请一个节点，而删除一个元素时又可以释放这个节点，在内存中链表占用的区域可以动态变化。

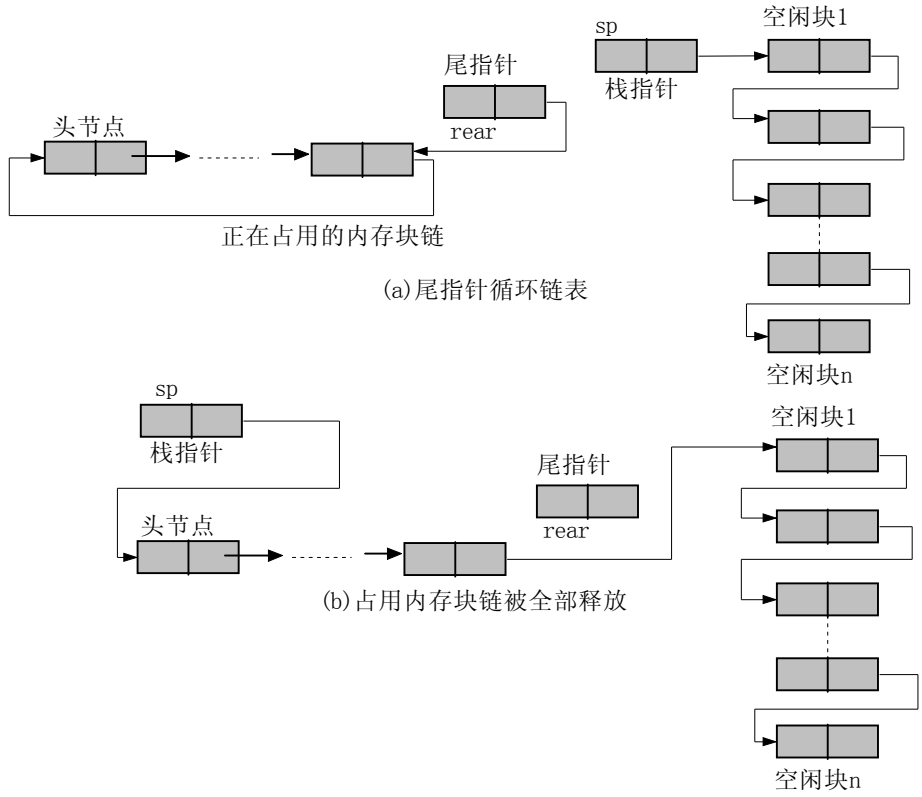


图 1.16 循环单链表应用

1.2.2.2 单链表设计

单链表是我们掌握链表程序设计的基础。链表设计首先要定义节点结构，沿用前面的例子，一个具体的单链表设计是如下步骤：

一、建立空表并定义节点结构

```
struct  BILL{
                int     key;            //用关键码排序
                char    facility[20];
                char    type[10];
                int     cost;
                int     num;
                struct  BILL    *next;
                };
```

在主程序中定义一个头部指针：

```
    struct  BILL *head=NULL;
```

现在我们已经建立了单链表的头节点指针并完成了初始化。在 C 语言中有一个 malloc()函数，它用于动态申请内存分配，节点插入时，我们用它每次从内存申请一个节点所需的内存，即前面所说的链表的动态成长，此函数具体功能可参考 C 语言手册。

二、插入节点生成链表

输入新的节点可以看成是对单链表的插入运算。图 1.17 给出了插入节点的过程示意，设节点递增有序。它表明了指针的修改方法及要点。

在 p 节点前插入 S 时，插入函数要区分三种不同情况：①表空，S 成为表头；②表中无 p 节点，S 插入链尾；③找到 p 节点，将 S 在其之前插入。插入时指针修改的顺序特别注意，要在切断 q 与 p 的节点链之前，先把 S 的指针指向 p，以免丢失指针链信息，原则上是：

（1）修改 S 指针指向 p 节点，取得后继节点指针信息：

```
S->next=P;  //定位 S
```

（2）修改 p 节点前趋 q 的指针指向 S，插入 S 到链表中：

```
q->next=S;  //修改 q 指针
```

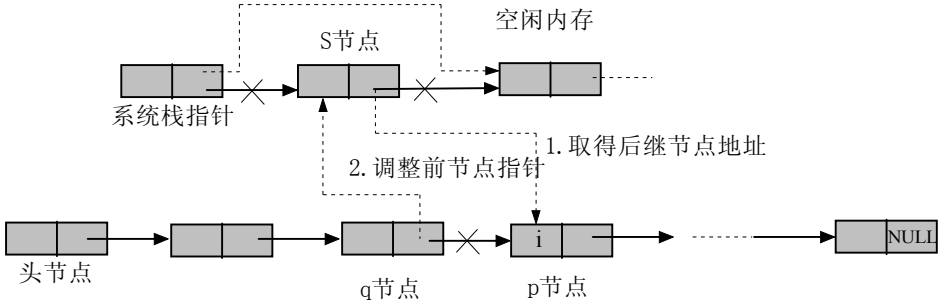


图 1.17 在 p 节点前插入 S 的过程

下面是单链表插入程序的例子，程序定义的节点结构是前述的 BILL 单链表节点。插入前节点序列是：

```
head->...q，p，...n
```

插入后是：

head->...q，S，p，... n

链表按关键字递增有序。

程序 1.3 单链表节点插入

```
struct  BILL *dls_store(struct  BILL *S,struct  BILL *head)
{
    struct  BILL *p,*q;       //定义中间变量
    if(!head){                //表空，返回 S 为头部节点
            head=S;
            S->next=NULL;
            return(S);
            }
    p=head;                    //从头开始搜索 p 节点
    q=p;
    while(p){
        if(p->key<S->key){ //当前节点关键字值小于 S 节点关键字值，搜索下一个节点
                q=p;
                p=p->next;
                }
        else{                  //找到插入位置，在节点 p 之前
            if(p==head){    //是头部?
                    S->next=head;
                    head=S;
                    return(S);
                    }
            q->next=S;          //是链表中间, 在节点 p 之前插入 S 节点
            S->next=p;          //因有中间变量定义，所以指针修改顺序可以不考虑
            return(head);       //返回头节点给调用程序
            }
        }
//走出循环体，则该表非空且无关键字值大于 S 节点，S 插入链尾
    q->next=S;                  //如用 p-->next=S 则错，因此时 p 为空
    S->next=NULL;
```

```
    return(head);
}
```

这个例子头部节点不占用实际内存，即 head 只是一个指针，此函数调用形式为：

head=dls_store(S,head);

因为 head 是指针，所以函数被定义为指针型函数，它返回一个指针。不同的链表结构有不同的程序实现，即使是同一链表结构其程序实现也是不同的，重要的是了解链表设计的基本要点与概念。此程序的节点数据输入如下：

程序 1.4 单链表节点数据输入

```
struct  BILL *enter()
{
    struct  BILL *S;
    S=(struct BILL *)malloc(sizeof(BILL)); //向内存申请一个节点
    if(!S)exit(-1);                        //如果失败则退出程序
    cout<<"输入序号"<<endl;
    scanf("%i",&(S->key));
    cout<<"输入设备名称"<<endl;
    scanf("%s",S->facility);
    cout<<"输入型号"<<endl;
    scanf("%s",S->type);
    cout<<"输入单价"<<endl;
    scanf("%i",&(S->cost));
    cout<<"输入数量"<<endl;
    scanf("%i",&(S->num));
    return(S);
}
```

程序 1.5 单链表节点输出

```
void list(struct BILL *p)
{
printf("序号名称    型号    单价    数量\n");
if(p){
  while(p){
    printf("%i  %s  %s  %i  %i\n",p->key,p->facility,p->type,p->cost,p->num);
    p = p->next;
```

```
    }
  }
}
```

三、删除节点

节点的删除运算与插入是一对相辅相成的功能函数。与插入相反，在搜索到要删除节点后，修改其前项指针，并释放节点占用的内存（malloc()的反函数 free()）于存储器管理系统，单链表删除 p 节点操作见图 1.18 所示。因为程序比较简单，我们直接给出如下的删除函数 c 语言程序。

程序 1.6 单链表节点删除

```
struct  BILL *del(int key,struct  BILL *head)
{
    struct  BILL *p,*q;          //定义中间变量
    if(!head)return(0);          //表空返回
    if(head->key==key){
            p=head;              //暂存头部信息
            head=head->next;//先修改指针后释放节点内存
            free(p);
            return(head);
            }
    p=head;                      //从头开始搜索 p 节点
    q=p;
    while(p){
        if(p->key!=key){//当前节点关键字值不等于输入关键字值，搜索下一个节点
                q=p;
                p=p->next;
                }
        else{                        //找到 i 值节点 p
            q->next=p->next;     //修改指针删除 p 节点
            free(p);             //释放节点占用的内存
            return  (head);
            }
        }
    cout<<"无指定节点"<<endl;    //走出循环体，表明该表无关键字值的节点
```

```
    return(head);
}
```

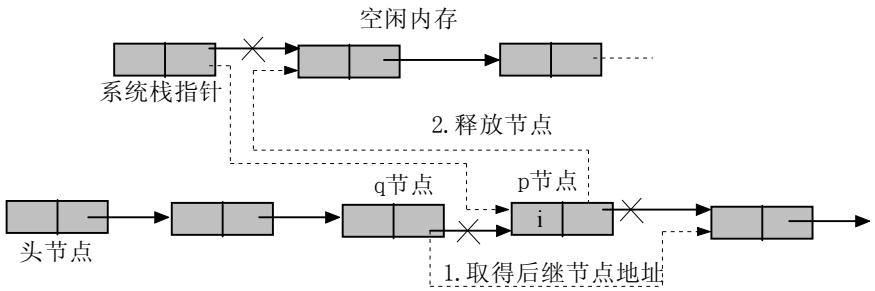


图 1.18 删除 p 节点的过程

现在我们可以给出单链表插入的主函数程序如程序 1.7。

程序 1.7 单链表生成程序

```
#include<stdio.h>
#include<malloc.h>
#include <stdlib.h>
#include<iostream.h>
#include<conio.h>
struct  BILL *del(int,struct BILL *);
struct  BILL *enter();
void    list(struct BILL *);
struct  BILL *dls_store(struct  BILL *,struct  BILL *);
struct  BILL{
            int     key;            //用关键码排序
            char    facility[20];
            char    type[10];
            int     cost;
            int     num;
            struct  BILL    *next;
            };
int main(void)
{
    struct  BILL *head,*S;
    int key,flg=0;
    head=NULL;
    for(;;){
```

```
        cout<<"插入：i；退出:q；列表：l；删除：d"<<endl;
        switch(getch()){
            case 'i':
                S=enter();
                head=dls_store(S,head);
                list(head);      //列表
                break;
            case 'l':
                list(head);      //列表
                break;
            case 'q':
                flg=1;       //退出程序
                break;
            case 'd':
                cout<<"输入要删除节点的序号"<<endl;
                cin>>key;
                head=del(key,head);
                if(head)list(head);
                break;
            }
        if(flg==1)break;
        }
    return(0);
}
```

程序 1.7 只包含了建立单链表的基本功能函数，此外还有检索、存储等操作，请读者参考图 1.19，编程并上机调试，为后面的双链表设计做准备。

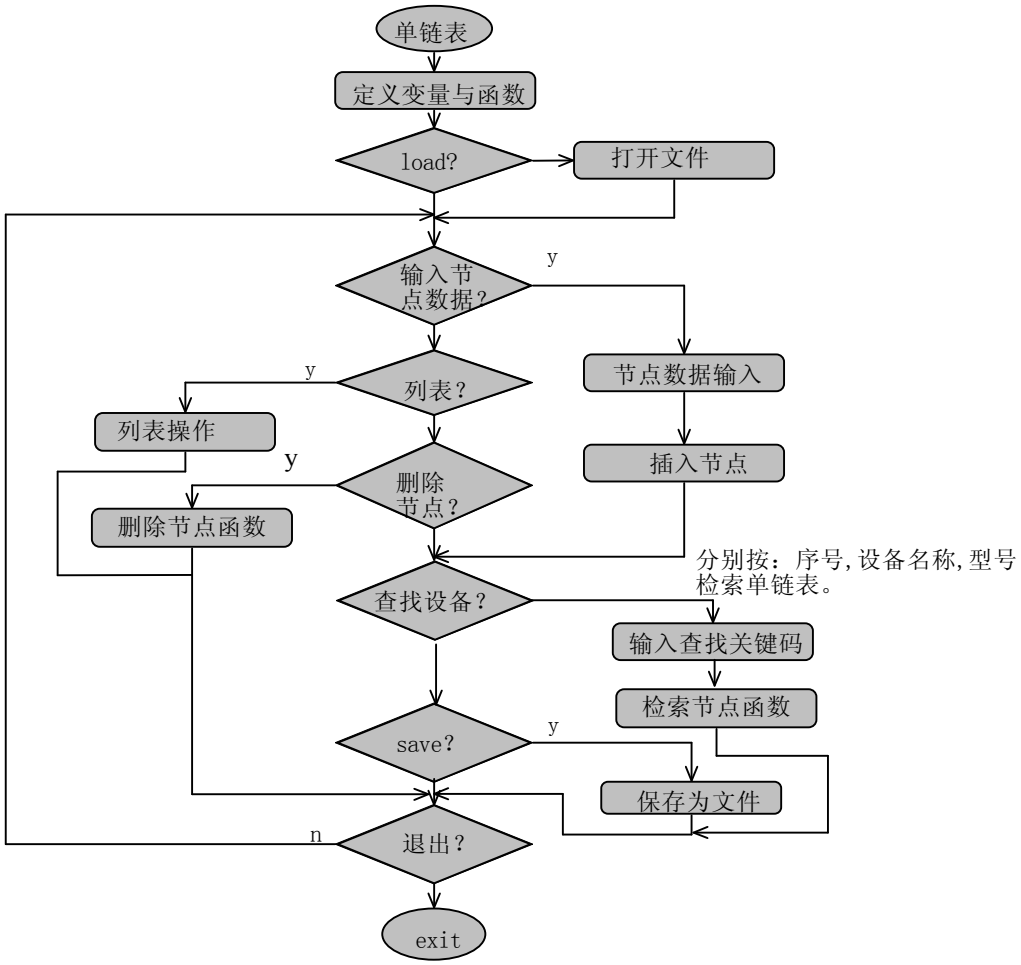


图 1.19 单链表操作流程

1.2.2.3 单链表操作效率

链表的特点是在存储结构上用指针表达了相邻元素的逻辑关系，因此它的效率体现在插入与删除方面很高，没有顺序表要移动元素的问题。另一方面，链式存储是顺序存取结构的（与顺序表是随机存储结构不同），设链表长为 n，单链表操作时每次访问、查找一个元素必须从表头开始，如果查找任一元素的概率相等，则有 $p_i = \frac{1}{n}$，其平均查找长度是：

$$E = \frac{1}{n}\sum_{i=1}^{n} i$$
$$= \frac{n+1}{2}$$

即最好情况是一次查找成功（为表头元素），最坏情况是查找了 n 次（表尾，且不考虑失败），平均是 $\frac{n+1}{2}$ 。我们知道顺序存储结构 ADDR(i)=S_0+（i-1）*L。这里，i 是序号，i=1，2，...，n；L 是节点实际占用内存长度，L=sizeof(变量)，S_0是顺序表在内存的起始地址。即只要给定了元素下标 i，顺序表可以由表达式求出它相应在内存中的位置，进行元素存取操作，所以称为随机存取结构。随机存储结构在读操作上其效率比链式存储结构要

高得多，但在插入与删除运算上则是 0（n）时间复杂度。至于检索运算要区分不同的情况，即是有序或无序表。

1.2.2.4 双链表设计

用单链表来表示线性表，其检索任何一个节点都只能从头部节点开始向后继节点方向搜索（或是从尾部向头部），即运算是单向的。如果在每一节点中再增加一个指针域，指向其直接前趋节点，则运算就可以双向进行，效率也会随之提高，这就是我们前面介绍过的双链表。双链表的节点定义已经在前节介绍过，在单链表设计的基础上，现在讨论双链表设计内容，与单链表最大的不同就是在对指针作修改时，出现了指向指针的指针概念。

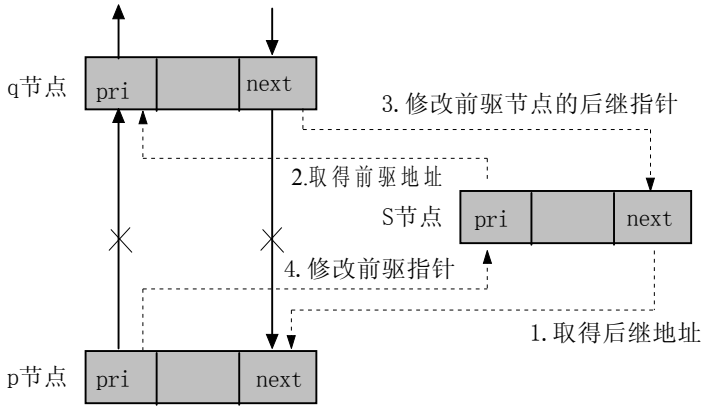


图 1.20 双链表的节点插入

一、双链表节点的插入

图 1.20 是双链表节点插入，指针的修改要点示意。修改原则是先取得直接前趋和后继节点的地址信息，以避免丢失指针链。步骤如下：

（1）S->next=p; 取得后继地址信息

（2）S->pri=p->pri; 取得前趋地址信息

（3）p->pri-->next=S; 修改 p 前趋节点的后继指针信息，指向新插入的 S

（4）p->pri=S; 修改 p 的前趋指针指向新插入的 S

设插入前节点序列是：

head-->...q，p，...，

插入后是：

head-->...q，S，p，...，

按关键字递增有序，重写双链表节点结构如下：

```
struct  BILL{
                int     key;
                char    facility[20];
                char    type[10];
                int     cost;
```

```
                int     num;
                struct  BILL    *next, *pri;
                };
```

现在我们给出双链表插入程序 1.8。

程序 1.8 双链表节点插入

```
struct  BILL *dls_store(struct  BILL *S,struct  BILL *head)
{
    struct  BILL *p,*q;          //定义中间变量
    if(!head){                   //表空，返回 S 为头部节点
            S->next=NULL;
            S->pro=NULL;
            return(S);
            }
    p=head;                      //从头开始搜索 p 节点
    q=p;
    while(p){
        if(p->key<S->key){  //当前节点关键字值小于 S 节点关键字值，搜索下一节点
                q=p;
                p=p->next;
                }
        else{                        //找到 i 值节点 p
            if(p==head){         //是头部?
                    S->next=head;    //表头节点信息也参加排序
                    S->pro=NULL;//非循环链表故表头前驱指针为空
                    p->pro=S;
                    return(S);
                    }
            S->next=p;           //链表中间插入 S 节点先取后继节点地址
            S->pro=p->pro;       //取得前趋节点地址
            p->pro->next=S;      //修改原 p 前趋节点的后继指针，指向 S 节点
            p->pro=S;            //修改 p 的前趋指针，指向 S
            return(head);
            }
```

```
        }
//走出循环体表明该表非空且无关键字值大于 S 节点，故 S 插入链尾
    q->next=S;                    //如果用 p-->next=S 则错，因此时 p 为空
    S->next=NULL;                 //非循环表，尾指针为空
    S->pro=q;
    return(head);
}
```

程序 1.8 增加了一个尾指针 last，利用双链表的前驱指针域，查询节点的操作可以由链表尾部通过 last 向头部方向检索。注意，程序中假设尾指针是一个全局变量。此函数调用形式为：

```
head=storage(S,head);
```

二、双链表节点删除

双链节点的删除同样是单链节点操作的推广，见图 1.21 示意。有关的具体程序读者可以参考双链表节点插入程序。需要注意的是头节点和尾结点删除处理情况。

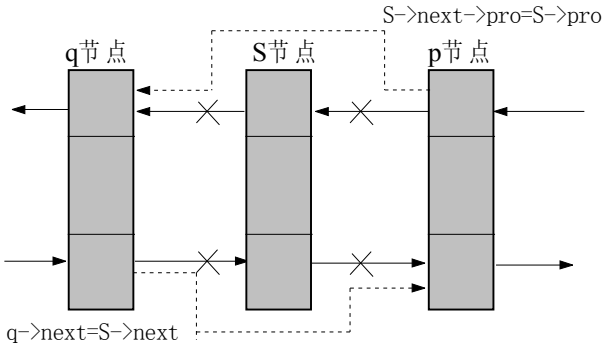


图 1.21 双链表的节点删除

1.2.2.5 链表深入学习

- **指针的初始化**

在数据结构中，链表程序设计相对简单，但有关它的基本概念依然不能忽视，最基本的仍然是指针的活用问题。下面我们看几个例子。

例 1.5 单链表复制。 单链表如图 1.22（a），请编写一 C 语言函数，将此单链表复制一份拷贝如图 1.22（b）。

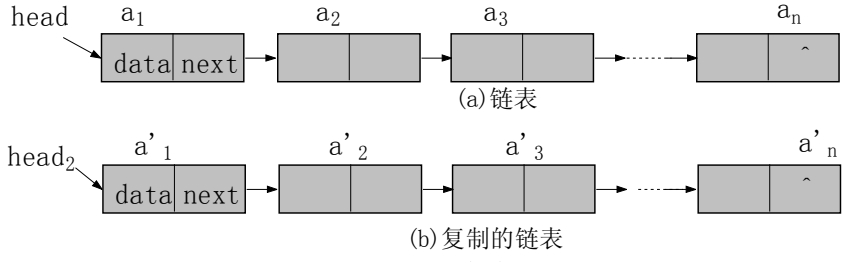


图 1.22 复制一个单链表

解：

题意要求在内存中建立一个链表的副本，其实质是考察同学对头指针的理解问题。我们假设在程序中如下处理：

```
struct node *copy(struct node *head)
{
    struct node *head2,*p,*q,*s;
    p=head;
    q=head2;              //取复制后的头节点指针
    while(p){
        s=(struct node *)malloc(sizeof(node)); //申请一个节点的内存
        s->data=p->data;//复制链表的数据域
        q->next=s;            //q 初始从 head2 开始顺序指向复制的节点地址
        q=s;                  //q 在递推更新指向复制链表的当前末节点
        p=p->next;            //p 也在递推更新指向要链表的下一个节点
        }
    ..........
}
```

程序对 head2 处理有什么问题呢？回顾一下我们在 1.1.2.2 中关于指针应用的几个要点中特别指出：一定不要使用一个没有赋值的空指针。现在 head2 就存在这个问题。它初始为空，在 while()循环体内的第一次循环时，q->next=s 实际上是 head2->next=s，无意中我们使用了一个空指针！后果是程序运行被终止。

指针必需指向一个变量才能获得有效的地址，我们必须非常清楚这个概念。现在只要给指针变量 head2 一个初值就行，程序 1.9 是修改后的完整复制函数，其节点定义与程序 1.7 完全相同，节点的数据域复制包括了字符串拷贝等操作。

程序 1.9 单链表复制

```
struct BILL *copy(struct BILL *head)
{
    struct BILL *head2,*p,*q,*s;
    if(!head)return(0);
    head2=(struct BILL *)malloc(sizeof(BILL));//为 head2 申请一个内存地址
    p=head;
    q=head2;
    while(p){
```

```
        s=(struct BILL *)malloc(sizeof(BILL)); //申请节点空间
        s->key=p->key;        //复制链表的数据域
        strcpy(s->facility,p->facility);
        strcpy(s->type,p->type);
        s->cost=p->cost;
        s->num=p->num;
        q->next=s;
        q=s;
        p=p->next;
        }
    q->next=NULL;
    q=head2->next;
    free(head2);
    return(q);
}
```

- **对称链表--指针概念的拓宽**

一般说，链表节点指针域存储的是相邻节点的地址，但在一种称之为对称表的结构中，节点指针域存储的却是相邻节点地址的运算结果，请看例 1.6。

例 1.6 对称单链表。设 X 和 Y 是两个 n 位的二进制数，称之为二进制位串。X⊕Y 是 X 和 Y 各对应的二进制位进行异或运算后所得结果，它仍然是一个二进制位串。而基于异或运算的对称单链表定义为其第 K_i 个节点指针域存储的是地址异或运算后的中间信息，现问：

（1）基于异或运算的对称单链表，其节点 K_i 的指针段信息内容、以及如何通过该信息求得与 K_i 相邻节点的地址。

（2）设表长为 n，头指针指向 K_1 节点，请画出基于异或运算的对称单链表的结构，标明各节点指针段内容，包括第一个节点、第二个节点、第 i 个节点和第 n 个节点。

解：

普通单链表只有一个后继节点指针域，所以检索只能沿单方向进行。为了在不增加指针域的情况下具有双链表的检索功能，可以采用对称单链表形式提高单链表的检索效率，本题要点仍然在于指针的应用。指针是地址，也就是二进制位串。我们知道，C 语言位操作运算符中有位异或操作，符号表示为“^”。异或是一种逻辑运算，两者相同，运算结果为逻辑零，两者相异，运算结果为逻辑 1。异或运算有下列特性：

（X⊕Y）⊕X=Y

（X⊕Y）⊕Y=X

根据异或特性，对称单链表节点指针域存储的信息内容，是通过对相邻节点地址做异或运算所得的结果。既第 K_i 个节点指针域存储的是 K_{i-1} 节点和 K_{i+1} 节点地址的异或运算值 $\xi_{i-1} \oplus \xi_{i+1}$，显然：

$(\xi_{i-1} \oplus \xi_{i+1}) \oplus \xi_{i-1} = \xi_{i+1}$

$(\xi_{i-1} \oplus \xi_{i+1}) \oplus \xi_{i+1} = \xi_{i-1}$

定义 NULL=0，则 $(NULL \oplus \xi_i) = \xi_i$，所以节点 K_1 指针段内容是 $(NULL \oplus \xi_2) = \xi_2$，节点 K_n 指针段内容是 $(NULL \oplus \xi_{n-1}) = \xi_{n-1}$。基于异或运算的对称单链表结构如图 1.23 所示。

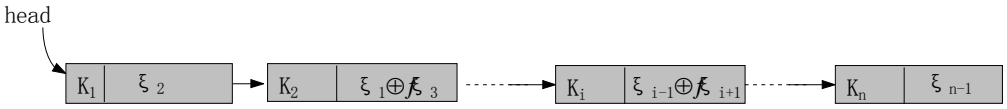


图 1.23 基于异或运算的对称单链表结构

下面这段程序在对称表中单方向搜索关键字值等于 key 的节点，找到后返回该节点指针位置，否则返回空。

程序 1.10 对称表检索

```
struct  BILL *search(int key,struct BILL *head)
{
    long ia;
    struct  BILL *ps,*p,*q,*qq,*pp;
    p=head;
    q=NULL;                              //前趋节点地址为空
    while(p){
        if(p->key!=key){
            ia=(long)q^(long)(p->next); //前趋节点指针和当前节点指针域异或运算
            ps=(struct  BILL *)ia;       //取得后继节点地址
            q=p;
            p=ps;
            }
        else    return(p);               //找到关键码值相等节点 p,返回指向它的指针
        }
    return(NULL);                        //检索失败，返回空指针
}
```

因节点 K_n 指针内容是 $(NULL \oplus \xi_{n-1}) = \xi_{n-1}$，当 p 指向节点 K_n 时 q 指向节点 K_{n-1}，所以 $q \oplus (p->next) = \xi_{n-1} \oplus \xi_{n-1} = NULL$。

作为习题，请读者参考程序 1.10 设计基于异或运算的对称单链表节点插入程序。链表

数据结构很简单，但前面几个例子说明，有关链表程序设计方面的概念依然不可轻视，它是我们学习数据结构的基础。

1.2.2.6 稀疏矩阵的三元组与十字链表

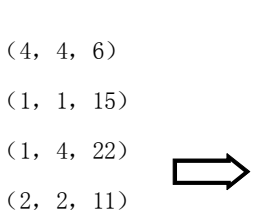
有关数组的内容在 c 语言课程学习过程中就已经很熟悉了。我们现在讨论稀疏矩阵的压缩表示方法，讨论目的是通过十字链表的内容加深读者对链表的理解广度。

- **三元组表**

设有 4×4 矩阵 M 如下：

$$\begin{pmatrix} 15 & 0 & 0 & 22 \\ 0 & 11 & 0 & 0 \\ 0 & 0 & -6 & 0 \\ 91 & 0 & 0 & -5 \end{pmatrix}$$

基于节约存储空间的目的，需要对矩阵 M 存在的零元素作压缩处理。算法是只存储非零元素及其下标，用一个 2 维的 3 列、（t＋1）行数组表示，此 2 维数组即为三元组。这里 t 是 M 的非零元素个数的总和，如图 1.24 所示。



	m	[0]	[1]	[2]
（4，4，6）	[0]	4，	4，	6
（1，1，15）	[1]	1，	1，	15
（1，4，22）	[2]	1，	4，	22
（2，2，11）	[3]	2，	2，	11
（3，3，-6）	[4]	3，	3，	-6
（4，1，91）	[5]	4，	1，	91
（4，4，-5）	[6]	4，	4，	-5

（a）矩阵M　　（b）三元组

图 1.24 稀疏矩阵的三元组表示

其中，M[0][0]～M[0][2]是(m，n，t)，而 M[i][j]是(i，j，a_{ij})。1≤i≤ m，1≤j≤n。因 m・n・sizeof(类型)是 m×n 矩阵占用的存储空间，而 3・t・sizeof(类型)是三元组表占用的存储空间（不考虑表首三字节），所以，三元组压缩存储空间条件是 m・n>>3t，或者 $t << \frac{mn}{3}$。 按照这个条件，上例 M 并不适于作三元组压缩，压缩算法也只是在大型矩阵运算时要考虑的问题。此外，读者要了解数组的特性有①均匀性，数组中每个元素的数据类型是一致的；②有序性，数组中元素的位置是有序的。这实际上也是线性表的特性。三元组是一个 2 维数组，它也必须满足以上两个条件。此外，定义三元组的数据类型必须与稀疏矩阵一致。如矩阵 A 为：

$$A=\begin{pmatrix} a_{11} & 0 & 0 & a_{14} \\ 0 & a_{22} & 0 & 0 \\ 0 & 0 & a_{33} & 0 \\ a_{41} & 0 & 0 & a_{44} \end{pmatrix}$$

定义 A 的三元组矩阵是：

```
float    array[6][3];
```

因为 a_{ij} 是浮点型，虽然 i,j 是整型量，其三元组也必须与 A 矩阵元素同时定义为浮点型。

- **十字链表**

十字链表是链表中的一个应用例子。用三元组表示稀疏矩阵有一个缺点，即只适用于转置运算，因为转置运算不会造成非零元素的增减。但是在加、减、乘、除运算中，矩阵非零元素的个数会发生变化，因而造成三元组的结构改变，为此，采用十字链表存储结构来表达稀疏矩阵适合于矩阵运算需要。十字链表程序设计需要注意有关内外部节点的区别问题。首先，我们选定十字链表的节点结构形式如图 1.25 所示。图 1.26 是十字链表逻辑结构。

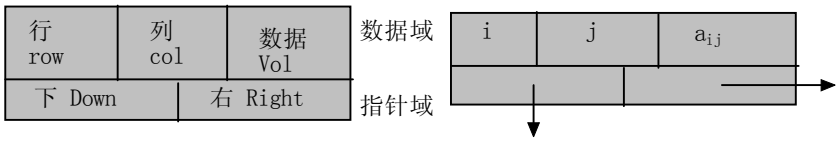


图 1.25 十字链表节点结构

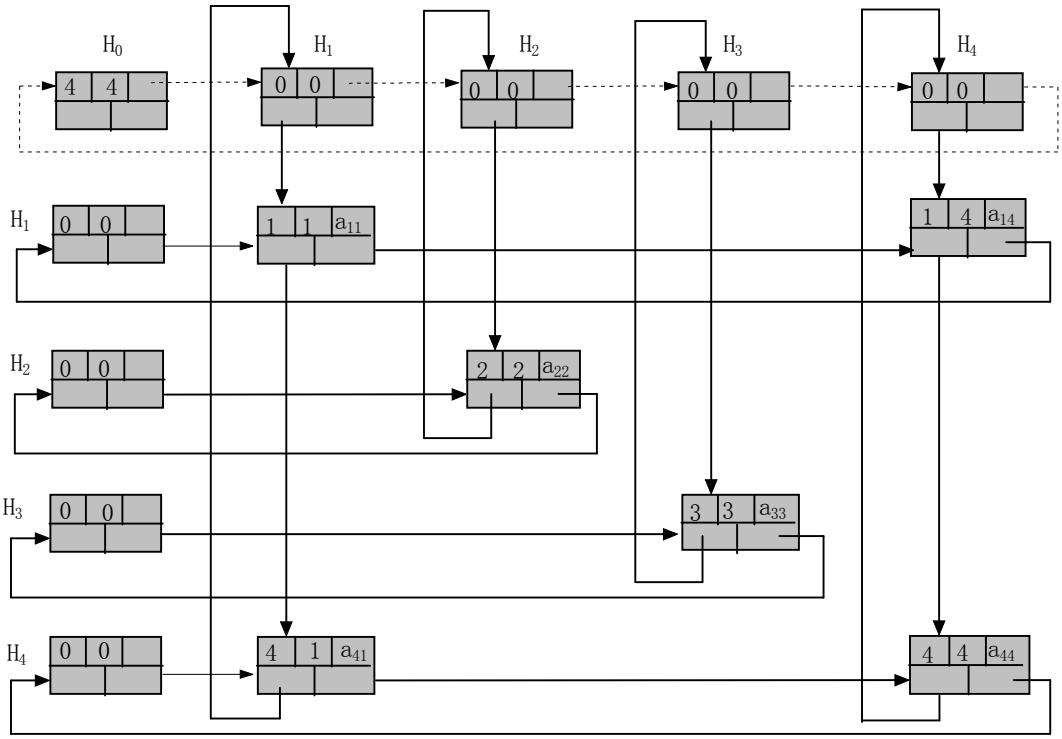


图 1.26 十字链表结构

这里，稀疏矩阵的每一非零元素是表中的一个节点（内部节点），分别在数据域存储它的行、列和元素本身。内部节点指针域的下指针指向本列中的下一个非零元素节点，它的

右指针指向本行中的下一个非零元素节点。

H_0～H_4是十字链表的辅助表头节点组，称之为外部节点。其中，H_0是三元组表的首节点。矩阵的每一行由一个行表头节点与该行的非零元素节点串成一个行循环链；矩阵的每一列也是由列表头节点与该列的非零元素节点串成一个列循环链，行列循环链的交叉称为十字链表。所以，H_0～H_4的右、下指针域分别是各行、列循环链的头指针。

因为行、列循环链表头的列、行指针域相互为空，所以可共用一个表头节点，即图 1.26 中的 H_1～H_4既是行表头，也是列表头节点。

注意观察图 1.26 的虚线，因为 H_1～H_4的指针域已经被各水平、垂直循环链的头节点占用，因此，要表达 H_0～H_4的线性相邻关系，必须考虑辅助手段，即虚线的指向关系。我们看到，如果用表头节点的数据域作为附加指针域，就形成如虚线所示的表头循环链，实际上是给出了各行 （或列）之间的线性有序关系。

外部节点的数据域要用做指针，所以首节点的非零元素个数 t 就不能用数据域表示。这里，就一些概念上的问题做一个讨论：

（1）二维数组是一个线性表

N 维数组也是一个线性表。因为它们每一行、列都可以看成是一个线性数据结构关系，因此十字链表也是线性数据关系，它要满足均匀性、有序性条件，即节点是同构的。

（2）内外节点同构问题

我们要求外部头节点的结构与内部非零元素节点的结构一致，那么，表头节点的 Val 域改为指针类型和内部节点的数据域类型必然有矛盾。所以，程序设计的问题是，在保持节点同构要求下，如何改动头节点的 VAL 域为指针类型，使之达到同构的目的又实现了表头循环链的构成。比如，内部节点定义是：

```
struct   node{
            int col,row;
            float   val;
            struct  node  *down，*right;
            };
```

如果外部节点是把 Val 直接改成指针数据类型的话，则有：

```
struct  headnode{
            int col，row;
            struct  headnode   *next;
            struct  node  *down，*right;
            };
```

由 C 语言的指针性质可知，结构 node != 结构 headnode，不符合链表设计中的结构一

致要求。实际上因为行、列循环链的尾节点必须指向头节点，而链中元素的指针类型与头节点不一致就会造成程序设计中的语法类型错误。

图 1.26 中的虚线用 Val 域作为指针，是把表头节点 H_1～H_4的线性有序关系在物理结构上表达出来，这样的辅助头指针循环链，使我们可以由表头 head 开始，沿表头节点链的 Val 域关系达到或者说搜索到任一行、列的非零元素节点，这就是图中虚线所表达的意图。没有辅助头节点指针链，则各行、列就是互相独立的循环链，矩阵行与列之间的线性有序关系无法表达。

既然程序设计上不允许定义两种节点结构于同一链表中，就不能简单的把节点数据域 Val 改为指针类型来达到链接表头节点的目的。实际上有两种方法实现：

（1） 外部节点辅助向量

定义一个向量 cp[M]，我们可以这样理解：

struct node cp[M]; //M=max{n, m}

这里，n 和 m 分别是矩阵行列值。cp[0]就是 head 节点，cp[i]（i=1，2，...，M）就是 H_1～H_M，向量元素之间所具有的线性关系表达了链接表头节点的目的，即图 1.26 中的虚线功能。于是，一个辅助向量解决了行列表头节点之间的顺序关系建立问题，也达到了链表中节点同构要求。程序 1.11 给出了使用辅助向量方式的十字链表节点插入 c 语言函数。

程序 1.11 辅助向量方式的十字链表节点插入函数

```
struct  BILL *dls_store(struct BILL *s,struct BILL *cp)
{
    struct  BILL *q,*p;
    q=(*(cp+s->row)).right;      //插入的行
    if(q==(cp+s->row)){
            s->right=(cp+s->row);
            (*(cp+s->row)).right=s;          //插入头节点
            }
    else{
        while((q!=(cp+s->row))&&(q->col<s->col)){
            p=q;
            q=q->right;              //查找插入列位置
            }
        if(q==(cp+s->row)){
                s->right=q;
                p->right=s;          //插入头节点
```

```
                }
        else {
            cout<<"输入列错误"<<endl;
            return(0);
            }
        }
    q=(*(cp+s->col)).down;          //插入的列
    if(q==(cp+s->col)){
            s->down=(cp+s->col);
            (*(cp+s->col)).down=s;  //插入到头节点
            }
    else{
        while((q!=(cp+s->col))&&(q->row<s->row)){
            p=q;
            q=q->down;              //查找插入行位置
            }
        if(q==(cp+s->col)){
                s->down=q;
                p->down=s;                  //插入
                }
        else {
            cout<<"输入行错误"<<endl;
            return(0);
            }
        }
    return(cp);     //正常返回向量首地址
}
//以下是初始化过程
int initialization(int *n,int *m,int *t,struct BILL *cp)
{
    int s,i;
    struct BILL *p;
    cout<<"输入矩阵行、列和非零元素个数"<<endl;
```

```
    cin>>*n>>*m>>*t;
    if(*m>*n)s=*m;
    else s=*n;
    (*(cp+0)).row=*m;
    (*(cp+0)).col=*n;
    for(i=1;i<=s;i++){
        (*(cp+i)).row=0;
        (*(cp+i)).col=0;
        (*(cp+i)).right=(cp+i);
        (*(cp+i)).down=(cp+i);
        }
    return(s);
}
```

（2） 节点中使用共用体定义

对节点的数据域 Val 定义共用体体：

```
struct   node{
            int col,row;
            union{
                float   val;
                struct  node  *next;
                }x;
            struct  node  *down, *right;
            };
```

请读者用 C 语言编写十字链表程序，节点结构如图 1.25，内外节点使用共用体形式。

1.2.3 堆栈

1.2.3.1 堆栈结构

堆栈是一种重要的数据结构形式，图 1.27 所示的是其逻辑数据结构，如果不看堆栈的操作方式，从线性表的角度上看堆栈完全符合线性结构定义：

（1）堆栈中的数据元素是有限的；

（2）每一节点只有一个前趋与后继；

（3）仅在端点元素时只有一个后继或前趋。

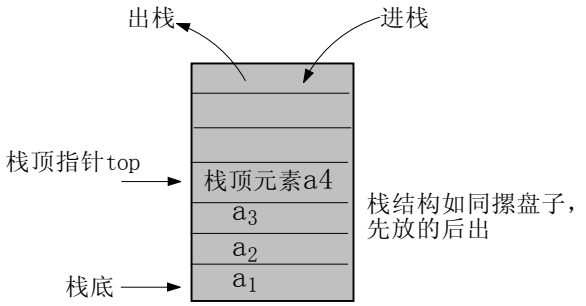


图 1.27 堆栈的逻辑结构

堆栈是操作受到一定限制的线性表，即元素只能栈顶进出栈。既然堆栈是线性表结构，其存储结构也如同线性表，也有顺序存储与链式存储之分。

1.2.3.2 基本操作

一、概念

堆栈，顾名思义它有客栈、货栈的用途。程序中的数据栈起到存储（或暂存）数据、地址的功能，与一般的存储方式不同的是，它要按照一种秩序，即先进后出的方式来存储程序的地址或数据。

为什么要按照先进后出的方式设计栈？请看图 1.28 所示程序运行流程。程序是顺序执行的，一般说在内存的某点开始执行，如图 1.28。当遇到第 1 断点时，主程序调用函数 1，即计算机转移到该函数所在的内存地址处开始执行该函数程序。同时，为了保证执行完毕后能正确返回到被中断处继续主程序的执行，计算机的操作系统需要保存断点地址，或称为返回地址，它被压入栈中，并在调用程序的结尾弹出。当出现嵌套中断时，栈的功能就充分显示出来。地址的保存顺序如图 1.29。

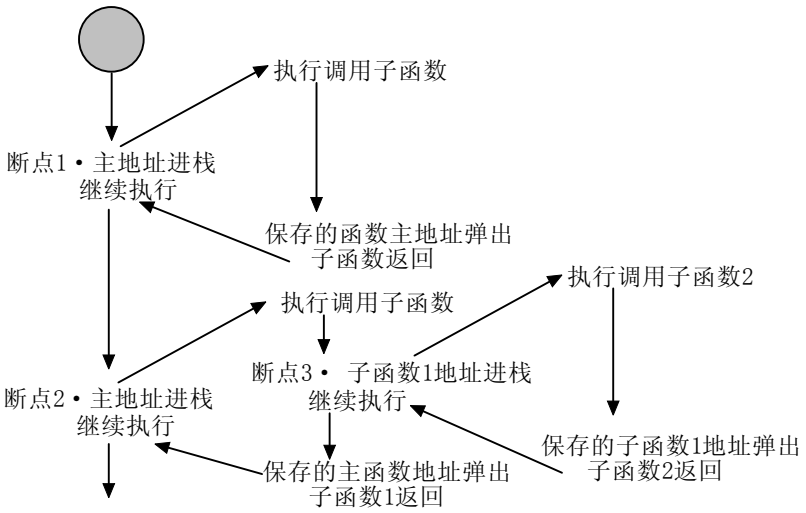


图 1.28 函数调用过程中栈的进出关系

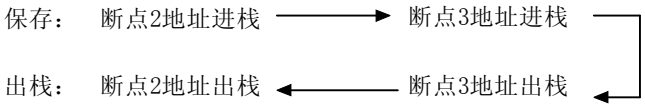


图 1.29 函数返回地址关系

它正好符合先进后出的顺序。因此，在程序设计中栈是一种重要的数据结构。

二、栈操作

基本运算有：

1) Push(st,x,top)　往栈 st 中压入 x；

2) Pop(st,&x,top)　从栈 st 弹出栈顶元素给 x；

3) Sempty(st)　　布尔函数，若栈空则为真（true）；

顺序栈的最大深度是限定的，设为 m。则建立一个顺序存储结构栈的步骤是：

```
int top=-1,Stack[m];     //设栈空时指针为负数
```

定义栈顶指针 top 意义如图 1.30 所示。

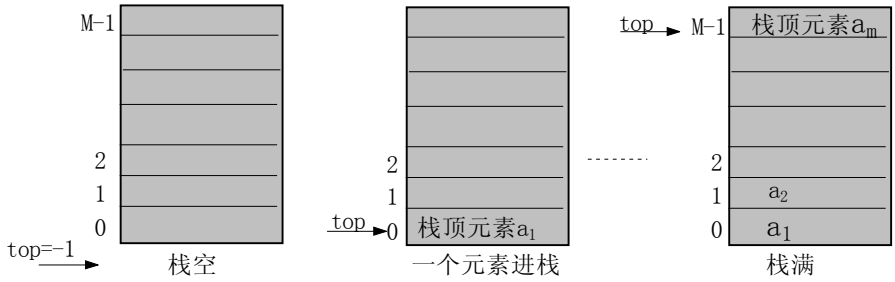


图 1.30 栈指针状态

其中，进栈函数 push(st,x,top)的 c 语言实现如程序 1.12 出栈函数 pop(st,&x,top)的 c 语言实现如程序 1.13 示。

程序 1.12 进栈函数

```
int push (struct node *p,struct node x,int top)
{
    if(top==M-1)printf("overflow");     //如栈满提示错误信息
    esle  {
          top++;                        //调整栈顶指针
          *(p+top)=x;                   //元素 x 进栈
          }
    return(top);
}
```

程序 1.13 出栈函数

```
int pop (struct node *p, struct node *x,int top)
{
    if(top<0)printf("overflow");    //如栈空提示错误信息
    else  {
          *x=*(p+top);              //出栈
          top--;                    //调整栈顶指针
```

```
            }
      return(top);
}
```

例 1.7 回文识别。回文是指一个字符串从前读和从后读都有一样的字母顺序。设长度为 n 的字符串在数组 array[n]中，求判别 array[n]中的字符串是否为回文的 C 程序实现（程序输出结果为 true 或 flase)。

解：设栈 stack[n]，元素 x 进栈操作为 push(s,x,top)，函数如下：

程序 1.14

```
void palindrome(char *array,int n)
{
      char *p,stack[M];
      int top=-1,i;
      for(i=n-1;i>=0;i--)top=push(stack,array[i],top);
      top=push(stack,0,top);
      if(strcmp(array,stack)) printf("flase\n");
      else printf("true\n");
}
```

三、栈的存储结构

栈是一个线性表，有顺序存储结构与链式存储结构两种实现方式。所谓顺序栈，就是定义一个数组。当栈的容量事先不确定时，我们可以采用链式存储结构，有关内容读者可以参考链表设计一节。

1.2.3.3 堆栈与递归

- 堆栈在递归程序中的应用

读者在 c 语言程序设计中已经接触过递归的概念和程序设计方法。一个直接调用自己，或通过一系列的过程调用语句间接的调用自己的过程（函数）称为递归调用过程（函数)。

递归是程序设计中很难掌握的内容，应用非常广泛。树、二叉树、广义表中的数据结构都是递归结构。某些数学函数，比如阶乘函数 n！求值，也可以表达为递归形式：

$$Fact(n)=\begin{cases}1, & n=0;\\ n\cdot Fact(n-1) & n>0;\end{cases}$$

如果对象是递归结构的，用递归函数实现程序就非常简洁，但设计方法比较难掌握。而栈在递归调用中有着重要作用，调用一个函数需要完成：

（1） 将实参与断点地址传送给被调用函数；

（2） 为被调用函数的局部变量在栈中分配数据区；

（3） 从被调用函数入口地址开始执行。

从被调用函数返回时正好相反：

（1） 传递被调用函数的运行结果给调用函数；

（2） 释放被调函数的数据区；

（3） 由保存的断点地址返回调用函数。

当一个递归函数被调用时，操作系统的工作栈必须是递归结构的。在一些计算机语言中并不支持递归函数。读者在 C 语言中已学习过递归的概念，知道它的每一步都由其前身来定义，在递归调用过程中，主调函数又是被调函数。执行递归函数将反复调用其自身。每调用一次就进入新的一层，如果编程中没有设定可以中止递归调用的出口条件，则递归过程会无限制的进行下去，最终会造成系统溢出错误。所以，程序必须有递归出口，即在满足一定条件时不再递归调用。

解决一个现实问题的算法是否应该设计成一个递归调用的形式，完全取决于实际应用问题本身的特性，只有在待处理对象本身具有递归结构特征的情况下，程序才应该设计为递归结构。比如在现实世界中描述一棵树的定义说，树是一个或多个节点组成的有限集合，其中：

a）必有且仅有一个特定的称为根（root）的节点；

b) 剩下的节点被分成 m≥0 个互不相交的集合 T_1，T_2，•••，T_m，而且其中的每一元素又都是一棵树，称为根的子树（Subtree）。

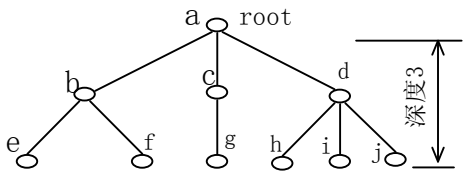


图 1.31 树的形式

显然树的定义是递归的。所以，有关树的函数结构都是递归形式的。如果任务对象本身不具备递归性质，比如，计算一个高阶方程式，就不可能设计递归形式的程序。

一个规模为 n 的问题，若该问题可以容易地解决（比如说规模 n 较小）则直接解决，否则将其分解为 k 个规模较小的子问题，这些子问题互相独立且与原问题形式相同，递归地解这些子问题，然后将各子问题的解合并得到原问题的解。这种算法设计策略叫做分治法。 使用分治法处理问题的一个例子是求 n 的阶乘。

从减小 n 的规模考虑，n 的阶乘可以看成是 n(n-1)!，而求(n-1)!与求 n！之间互相独立且与问题形式相同，显然，这是一个递归求解，因为我们追求将问题的规模一直分解到它的原子形式，也就是 1！=1，这就是出口条件，从底层回头，再将各子问题的解合并得到原问题的解，于是，2!=2，3!=6，...。阶乘函数的递归程序见程序 1.15

程序 1.15

```
int f(int n)
{
    if(n==1)return(1);
    return(n*f(n-1));
}
```

1.2.3.4 递归与分治算法

分治法的基本设计思想是将一个难以直接解决的大问题，分割成一些规模较小的相同问题，以便各个击破，分而治之。

如果原问题可分割成 k 个子问题，$1<k\leqslant n$ ，且这些子问题都可解，并可利用这些子问题的解求出原问题的解，那么这种分治法就是可行的。由分治法产生的子问题往往是原问题的较小模式，这就为使用递归技术提供了方便。在这种情况下，反复应用分治手段，可以使子问题与原问题类型一致而其规模却不断缩小，最终使子问题缩小到很容易直接求出其解。这自然导致递归过程的产生。分治与递归像一对孪生兄弟，经常同时应用在算法设计之中，并由此产生许多高效算法。

什么时候适用分治法？分治法所能解决的问题一般具有以下几个特征：

（1）该问题的规模缩小到一定的程度就可以容易地解决；

（2）该问题可以分解为若干个规模较小的相同问题，即该问题具有最优子结构性质。

（3）利用该问题分解出的子问题的解可以合并为该问题的解；

（4）该问题所分解出的各个子问题相互独立，即子问题之间不包含公共的子问题。

上述第一条特征是绝大多数问题都可以满足的，因为问题的计算复杂性一般是随着问题规模的增加而增加；第二条特征是应用分治法的前提，它也是大多数问题可以满足的，此特征反映了递归思想的应用；第三条特征是关键，能否利用分治法，完全取决于问题是否具有第三条特征，如果具备了第一条和第二条特征，而不具备第三条特征，则可以考虑贪心法或动态规划法。第四条特征涉及到分治法的效率，如果各子问题是不独立的，则分治法要做许多不必要的工作，重复地解公共的子问题，此时虽然可用分治法，但一般用动态规划法较好。分治法在每一层递归上都有三个步骤：

分解：将原问题分解为若干个规模较小，相互独立，与原问题形式相同的子问题；

解决：若子问题规模较小而容易被解决则直接解，否则递归地解各个子问题；

合并：将各个子问题的解合并为原问题的解。

根据分治法的分割原则，原问题应该分为多少个子问题才较适宜？各个子问题的规模应该怎样才为适当？这些问题很难予以肯定的回答。但人们从大量实践中发现，在用分治法设计算法时，最好使子问题的规模大致相同。换句话说，将一个问题分成大小相等的 k 个子问题的处理方法是行之有效的。许多问题可以取 k=2。这种使子问题规模大致相等的做

法是出自一种**平衡(balancing)子问题**的思想，它几乎总是比子问题规模不等的做法要好。

分治法的合并步骤是算法的关键所在。有些问题的合并方法比较明显，如对半检索的例子；有些问题合并方法比较复杂，或者是有多种合并方案，或者是合并方案不明显。究竟应该怎样合并，没有统一的模式，需要具体问题具体分析，这也是递归程序设计无一定之规的原因。下面我们讨论具体的例子。

- 进出栈元素序列的排列组合问题

堆栈在数据结构中的应用非常有技巧，涉及到的概念很活。比如，一个进栈元素序列是{a_1，a_2，…，a_n}，如问出栈的元素序列有多少种排列？如果从程序设计角度看栈的工作方式，它只是按先进后出的方式，简单的保存断点和传递实参，那么，我们也许会先入为主的认为其出栈排列显然就只有一种，即{a_n，a_{n-1}，…，a_1}。实际上，我们说栈的先进后出方式只是限定了元素进出栈规则，并没有限定元素进出栈之间的操作顺序，看例 1.8 所示。

例 1.8 现有一栈的深度为 n，设进栈元素序列是{a_1，a_2，…，a_n}，问出栈的元素序列有多少种排列？

解：

据题意可知进栈序列只有一种顺序：a_1，a_2，…，a_n，但是，元素进出栈之间的顺序没有规定。那么，当 n=1 时只有一种出栈序列，当 n=2 时，出栈元素序列有 2 种可能，操作方式对应的 2 种排列顺序是：

（1） a_1进栈→a_2进栈→ a_2出栈→a_1出栈，出栈序列是 a_2，a_1；

（2） a_1进栈→a_1出栈→ a_2进栈→a_2出栈，出栈序列是 a_1，a_2。

n=3 的情况如图 1.32 所示。由于排列 a_3，a_1，a_2逆序无法实现，该排列不可能实现，故出栈排列数不是 3 的全排列数 6，而是 5。

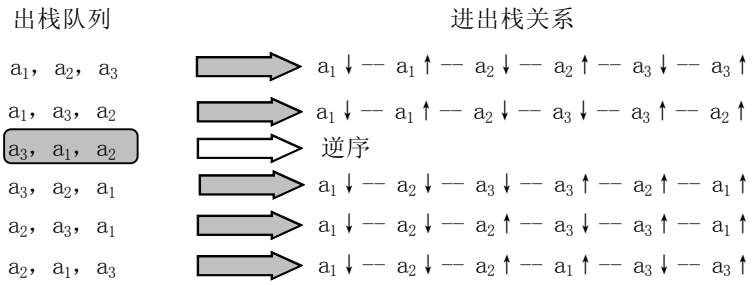


图 1.32 n=3 的出栈排列

n≥4 后，其出栈序列很难遍历，但是，根据分治法的设计思想，将原问题分解为若干个规模较小，相互独立，与原问题形式相同的子问题是可能的。

分解：设 n 个元素进栈的出栈排列有 x_n种。根据堆栈的原理，从元素 a_1进栈，再到 a_1出栈（a1'）之间有 i 个元素进出栈。则必有下式成立：

$$a_1,(a_i,\ \ldots,\ a_i'),a_1'(a_{n-i-1})$$

设（a_i，…，a_i’）的排列有 x_i 种，则有：

a_1（x_i）a_1’（x_{n-i-1}）

关系成立。所以，$x_i \cdot x_{n-i-1}$ 是在 a_1，a_1’间进栈了 i 个元素后，所余的 n-i-1 个元素可能的排列组合。

现在，我们寻找最小子问题的解：若 n=1，显然 x_i=1，若 n=2，已知 x_2=2，并且我们定义 n=0 时 x_0=1。

合并：将各个子问题的解合并为原问题的解是：

$$x(n)=\begin{cases}1 & n=0,1\\ 2 & n=2\\ \sum_{i=0}^{n-1} x_i x_{n-i-1} & n>2\end{cases}$$

程序 1.16 给出了其递归结构的 c 语言实现。

程序 1.16

```
int x(int n)
{
    int s=0,i=0;
    if(n==0)return(1);
    if(n==1)return(1);
    for(i=0;i<n;i++)s=s+x(i)*x(n-i-1);
    return(s);
}
```

要证明上述关系十分棘手，比如用归纳法证明的话，我们很难找到 n 和 n+1 之间存在的关联。为此换一个思路考虑，这个题目需要先转化一下模型来做，就相对比较简单了。模型转化如下：假设在一个直角坐标系中，从（0，0）出发走到（n，n），每次只能向右或者向上走一个单位长度，并且不能走到对角线之上，也就是（0，0）和（n，n）连接而成的线段。 这样，向右走可以看成是元素进栈，向上走可以看成是元素出栈。之所以要求不能走到对角线之上，是为了保证堆栈里有元素可出。这样问题就转化为格路问题，这是典型的组合数学问题。图 1.33 给出了过程示意。

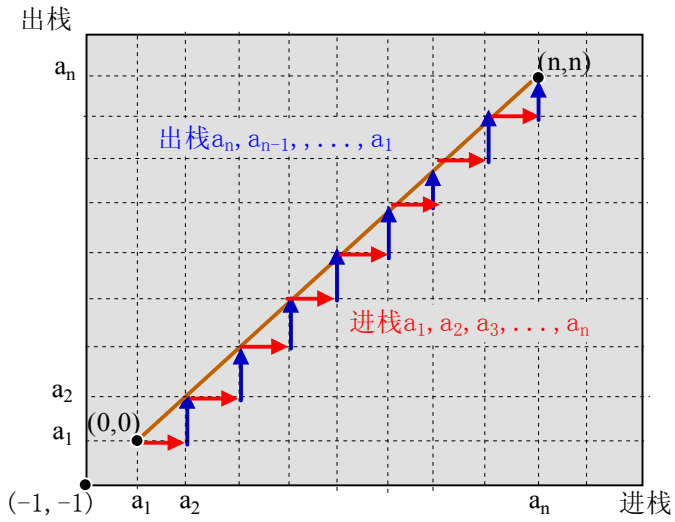


图 1.33 格路问题

从（0,0）走到（n,n）路径组合是 $C_{2n}^{n}=\frac{(2n)!}{n!n!}$，要想排除对角线以上（不包含对角线上的点）的路径组合，就是减去 $C_{2n}^{n+1}=\frac{(2n)!}{(n+1)!(n-1)!}$，而 $C_{2n}^{n}-C_{2n}^{n+1}=\frac{(2n)!}{n!(n+1)!}$。容易验证，对于任意给定的 n，它和程序 1.1 的结果完全相同（实际上我们仍然没有严格的证明）。

- 汉诺塔算法

Hanoi 塔问题：一个平面上有三根立柱：A，B，C。A 柱上套有 n 个大小不等的圆盘，大的在下，小的在上。如图 1.34 所示。要把这 n 个圆盘从 A 柱上移动 C 柱上，每次只能移动一个圆盘，移动可以借助 B 柱进行。但在任何时候，任何柱上的圆盘都必须保持大盘在下，小盘在上。求移动的步骤。

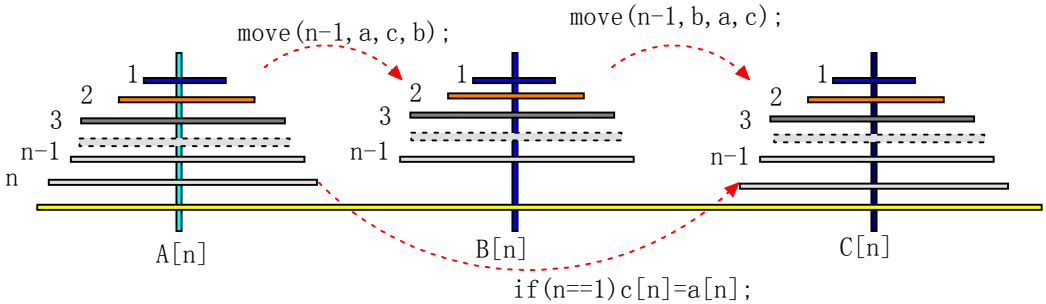


图 1.34 Hanoi 塔问题的递归求解

分析方法过程如下。

①简化问题：设盘子只有一个，则本问题可简化为 a→c。

②对于多于一个盘子的情况，首先减小问题规模，将问题分为两部分：第 n 个盘子和除 n 以外的 n-1 个盘子。如果将除 n 以外的 n-1 个盘子看成一个整体，则要解决本问题，可按以下步骤：

a、将 a 杆上 n-1 个盘子借助于 c 先移到 b 杆； a→b (n-1, a, c, b)

b、将 a 杆上第 n 个盘子从 a 移到 c 杆； a→c

c、将 b 杆上 n-1 个盘子借助 a 移到 c 杆。 b→c （n-1,b,a,c）

现在，已经知道最小问题的解，也知道各个子问题的描述，于是从 n 开始递归求解各个子问题，由递归出口条件求得最小问题的解，再合并为原问题的解，过程如程序 1.17。

程序 1.17 汉诺塔

```
void move(int n,int *a,int *b,int *c)
{
    if(n==1)*(c+n)=*(a+n);        //出口条件
    else {
        move(n-1,a,c,b);     //递归调用
        *(c+n)=*(a+n);
        move(n-1,b,a,c);
        }
}
```

1.2.3.5 递归与递推

递推（迭代）是从给定的初值开始计算一个序列随规模 n 递增的函数值。用分治法求解问题，则是从达到规模 n 的函数表达式，追朔到一个序列的原始初值，然后再层层返回。递归程序虽然非常简洁，但并不是所有场合适用。如果一个算法存在明显的递推关系，那么递归求解往往不是一个明智的选择。比如，求 n 的阶乘函数就存在清晰的迭代关系，而且程序 1.18 与递归算法同样的简洁直观。

程序 1.18

```
#include<stdio.h>
int main(void)
{
      long i=0,n,sum=1;
      printf("请输入 n:\n");
      scanf("%d",&n);
      while(i<n){
        i+=1;
        sum*=i;
        }
      printf("n!=%d\n",sum);
      return(0);
}
```

就算法效率而言，程序 1.18 和程序 1.15 似乎没有明显的差异，但实际上在函数调用过程中，进出栈操作要比赋值语句复杂得多。一个更明显的例子是求解 Fibonacci 序列的算法。Fibonacci 函数定义如下：

$$Fib(n)=\begin{cases}1 & n=0\\ 1 & n=1\\ Fib(n-1)+Fib(n-2) & n>1\end{cases}$$

它的递归实现如下所示。

```
int fib(int n)
{
    if(n==0)return(1);
    if(n==1)return(1);
    return(fib(n-1)+fib(n-2));
}
```

显然，对于 n>1，每一次 Fib()函数调用都会引起两个新的调用过程，见图 1.35 所示。所以调用的总次数是按指数增长的，当规模 n 很大的时候，算法所耗费的时间的增长量说明它不切实际。

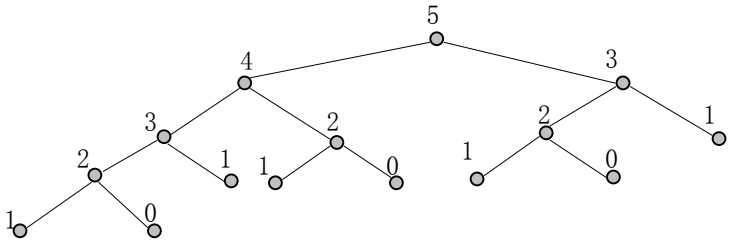


图 1.35 求 Fib(5)的 15 次调用过程

另一方面，Fibonacci 序列存在着明显的迭代关系，设初值 x=1，y=0，则程序 1.19 给出了 Fibonacci 序列的迭代求解过程。

程序 1.19

```
int finonacci(int n)
{
    int i=0,x=1,y=0;
    while(i<n){
        i+=1;
        x=x+y;
        y=x-y;
        }
    return(x);
```

}

迭代算法避免了 Fibonacci 序列某一数值的重复调用。因此，我们说除非必要，否则应尽量避免递归结构的程序设计。但是，对于某些数据结构本身就存在着递归关系的场合，那就是例外了。比如树型结构，定义在一棵树上的所有操作都应该考虑递归程序设计问题。

实际上，我们往往很难确切的分析出一个算法的运算效率，比如背包问题的回朔算法。

回朔是说，某些问题的求解过程是一个试探的过程，在探索解的过程中，保留着返回的路径。当求解受阻的时候，需要逆序退回到原路径的某一点上，重新选择新的探索方向。路径记忆可以用堆栈，也可以用队列（比如迷宫求解的例子）。因此，堆栈存储了算法求解序列曾经到达的每一种状态。

设有一个背包可以放入的物品重量为 S，现有 n 件物品，重量分别为 W1，W2，...，Wn。问能否从这 n 件物品中选择若干件放入到背包中，使得放入的重量之和正好为 S。如果存在一种符合要求的选择，则称此背包问题有解（真：true），否则此问题无解（假：false）。

背包问题可以用回朔和递归两种形式求解，要想直接写出背包问题的非递归求解过程不是一件简单的事情，这里，我们仅描述递归处理方法如下。

用 pack（S,n）表示上述背包问题的解，这是一个布尔函数，其参数满足 S>0，n≥1。背包问题如果有解,其选择只有两种可能,一是选择的一组物品中不包含 Wn,于是 pack(S,n)的解就是 pack(S,n-1)的解。另一种可能是选择物品中包含有 Wn，这时 pack(S,n)的解就是 pack(S-Wn,n-1)。现在，我们已经找到可以将规模减小的处理思路了。另外，当 S=0 时，背包问题总是有解，即 pack(0,n)=true，也就是不选择任何东西放到背包中。于是，我们又找到了原子问题的解。最后，当 S<0 时，背包问题总是无解，即 pack(S,n)=false，因为无论怎样选择物品都不能使其重量之和为负数。当 S>0，但 n<1 时，背包问题也是无解，即 pack(S,n)=false，因为不取任何东西就使其重量为正值也是不可能的。现在，我们归纳背包问题的递归定义如下：

$$\text{pack(S,n)} = \begin{cases} \text{true} & \text{当} S = 0 \\ false & \text{当} S < 0 \\ false & \text{当} S > 0 \text{且} n < 1 \\ \text{pack(S,n-1)或pack(S-Wn,n-1)} & \text{当} S > 0 \text{且} n \geq 1 \end{cases}$$

因为每递归一次 n 都减一，S 也可能减少 Wn，所以，最终程序一定会出现 S≦0 或者 n=0 的情形，也就是递归出口（引自许桌群，《数据结构》，高教出版社，1987，p30-31）。程序 1.20 给出了背包问题的递归函数，其中，数组 w[n]定义为全局变量，n 件物品的重量存放在 w[1]至 w[n]中，S 为背包标称重量。

程序 1.20

```
int pack(int s,int n)
{
```

```
    if(s==0)return(1);        // w[n]是全局变量，n个物品的重量在w[1]至w[n]中
    if((s<0)||((s>0)&&(n<1)))return(0); //用1代表true,0代表false
    if(pack(s-w[n],n-1)==1){
            printf("%d\n",w[n]);
            return(1);
            }
    return(pack(s,n-1));
}
```

显然，即使背包问题有解，其解也不是唯一的。

1.2.3.6 栈应用

- 编译程序扫描问题

堆栈结构在程序编译中被广泛应用，一段程序需要编译成为CPU可执行的机器码才能运行，称为执行文件（*.exe）。当编译程序扫描每一行语句时，首先需要检查是否存在语法错误，例1.9说明了利用栈结构检查左右括弧是否匹配的方法。

例1.9 设堆栈上限为100，输入一行c语句的字符串，长度不超过80，求：

（1）程序从左至右扫描字符串时判别该字符串中左、右圆括弧是否平衡。如果字符串不平衡，返回字符串中第一个不匹配的圆括弧位置；若平衡则返回正常匹配信息。即遇见第一个不匹配的右圆括弧时，中断扫描并返回其在字符串中的位置，如有多个左圆括弧不匹配就返回第一个不匹配的左圆括弧在字符串中的位置。

（2）设长度0至n-1的字符串在数组str[80]中，写出用栈函数实现该算法的c程序。

解（1）：编译程序扫描下列语句存在左右括弧不平衡情况。

```
if((a>b)&&(c<d)c=10;//第3个位置的左圆括弧不匹配
if(a>b)&&(c>d))c=10;//第15个位置的右圆括弧不匹配
```

算法思想：用栈存储扫描过程中遇见的左圆括弧在字符串中的位置，每遇见一个左圆括弧就将其在字符串中的位置进栈，每遇见一个右圆括弧就弹出一个栈顶元素，因为右圆括弧总是与最近一个左圆括弧相匹配，即其位置最近进入堆栈的那个左圆括弧，利用堆栈先进后出原理可以检查左右匹配情况，如果栈空，则当前是不匹配的右圆括弧，将其位置返回；如果扫描结束并且栈不为空，则有左圆括弧不匹配，返回栈底元素。图1.36给出了用堆栈结构扫描语句：

```
if((a>b)&&(c<d)c=10;
```

的过程。由于该语句的右括弧比左括弧少一个，当扫描到分号‘;’时，栈并不为空，栈顶元素是3，表明该语句行的第三个字符位置上的左括弧没有匹配。

解（2）：栈函数算法的c程序实现如程序1.21。

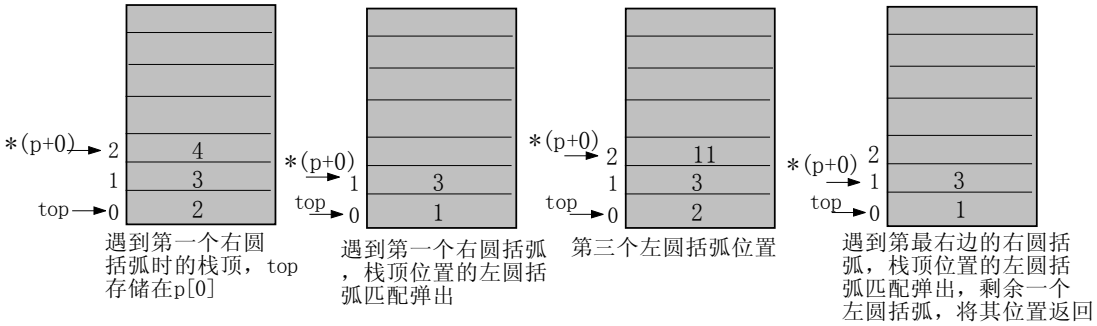


图 1.36 扫描例 1.9 第一条语句时匹配栈的状态

程序 1.21

```
int balance(char *p,int n,char *message) //p 指向输入行，n 是长度，message 返回信息
{
      int i,val,stack[LENGTH];
      stack[0]=0;                          //建立一个栈，stack[0]是栈顶指针 top
      for(i=0;i<n;i++){
            if(*(p+i)=='(')push(stack,i);//'('所在位置 i 进栈
            else {
                 if(*(p+i)==')'){
                         if(pop(stack,&val)==-1){
                              strcpy(message,"右圆括弧不匹配，位置：");
                              return(i+1);     //返回非法右圆括弧位置
                              }
                         }
                 }
            }
      if((i==n)&&(stack[0]!=0)){//左圆括弧多余，栈底元素是第一个非法的左圆括弧位置
            while(pop(stack,&val)!=-1){}
            strcpy(message,"左圆括弧不匹配，位置：");
            return(val);     //返回失衡位置
            }
      strcpy(message,"圆括弧匹配正常");
      return (-1);                         //正常返回为-1
}
int push(int *p,int x)
{
      if(*(p+0)==LENGTH)return(-1);        //如栈满提示错误信息
```

```
    else  {
          *(p+0)+=1;                    //调整栈顶指针
          *(p+*(p+0))=x+1;              //元素位置进栈
          }
    return(1);                          //如栈非满返回正常信息
}
int pop(int *p,int *val)
{
    if(*(p+0)==0)return(-1);            //如栈空返回错误状态
    else  {
          *val=*(p+*(p+0));             //出栈
          *(p+0)-=1;                    //调整栈顶指针
          }
    return(1);                          //如栈非空返回正常状态
}
```

- 表达式求值

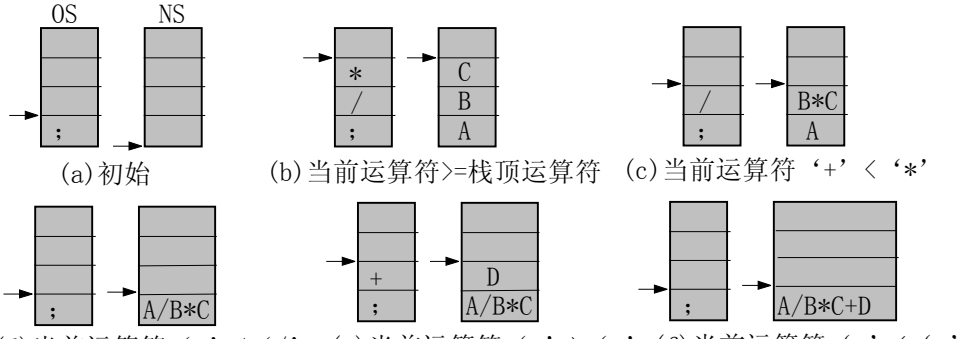
表达式求值是编译程序中最基本的问题。C 语言中每一种运算符对应着相应的优先数，优先数大的级别高，在表达式中优先处理。表 1.2 给出了 c 语言运算符优先级排列。

编译系统使用两个工作栈按运算符的优先级处理表达式求值问题。一个数据栈 NS，一个是运算符栈 OS，且 OS 初始装入运算符‘;’。工作时，编译程序从左至右扫描表达式，遇到操作数就压入数据栈;遇到运算符，则比较该运算符优先数和 OS 栈顶元素优先数的差别，若大于栈顶运算符的优先数，将该运算符压入 OS 栈成为新栈顶元素。否则，OS 栈顶运算符出栈（设为 θ），同时数据栈弹出两个操作数（设为 x 和 y），以出栈运算符 θ 连接这两个操作数进行运算（x θ y），并将结果压入数据栈 NS。扫描过程一直到遇见边界符‘;’且 OS 栈顶运算符也是‘;’为止，此时的 NS 栈顶元素就是表达式值。表达式 A/B*C+D;的求值过程如图 1.37 所示。参考函数如程序 1.22。

表 1.2 运算符的优先级与结合律

优先级	运算符	优先数	结合律
从高到低	( ) [ ] -> .	15	从左至右
	! ~ ++ -- (类型) sizeof + - * &	14	从右至左
	* / %	13	从左至右
	+ -	12	从左至右
	<< >>	11	从左至右
	< <= > >=	10	从左至右
	== !=	9	从左至右
	&	8	从左至右
	^	7	从左至右

排列	|	6	从左至右
	&&	5	从左至右
	||	4	从右至左
	?:	3	从右至左
	= += -= *= /= %= &= ^= |= <<= >>=	2	从左至右
	;	1	



(a)初始 (b)当前运算符>=栈顶运算符 (c)当前运算符‘+’<‘*’

(d)当前运算符‘+’<‘/’ (e)当前运算符‘+’>‘;’ (f)当前运算符‘;’<‘+’

图 1.37 扫描表达式 A/B*C+D;栈的状态

程序 1.22 表达式求值函数

```
int exp(char *p,int n)
{
    int i,j=0,val;
    char os_stack[LENGTH],ns_stack[LENGTH];
    char t,w,q,iw,iq,z,x,y;
    os_stack[0]=0;                    //建立一个栈，stack[0]是栈顶指针
    ns_stack[0]=0;
    push(os_stack,';');               //初始运算符
    t=0;                              //t=0 表示扫描下一个符号
    while(t!=2){
        if(t==0){
            w=*(p+j);                 //w 是当前扫描符号
            if(w=='=')w=';';
            }
        if((w!='+')&&(w!='-')&&(w!='*')&&(w!='/')&&(w!=';')){
            push(ns_stack,w);
            j++;
            }                         //假设仅限定 short 类型数据运算
        else {
            pop(os_stack,&q);         //取栈顶运算符
            table(w,&iw);
```

```
            table(q,&iq);              //取优先级数
            if(iw>iq){
                push(os_stack,q);      //恢复栈顶
                push(os_stack,w);      //新操作符进栈
                t=0;
                j++;
                }
            else{
                if((q==';')&&(w==';')){pop(ns_stack,&z);t=2;} //表达式求值结束
                else {
                    pop(ns_stack,&y);          //数据出栈
                    pop(ns_stack,&x);
                    x=operand(x,q,y);          // operand()是运算函数，q 是运算符
                    push(ns_stack,x);
                    t=1;
                    }
                }
            }
        }
    return(z);                 //返回表达式值
}
```

1.2.4 队列

1.2.4.1 队列结构

队列是一种重要数据结构形式，其逻辑结构如图 1.38 所示。从线性表的角度上看队列符合线性表定义：

（1）队列中的数据元素是有限的；

（2）每一节点只有一个前趋与后继；

（3）仅在端点元素时只有一个后继或前趋。

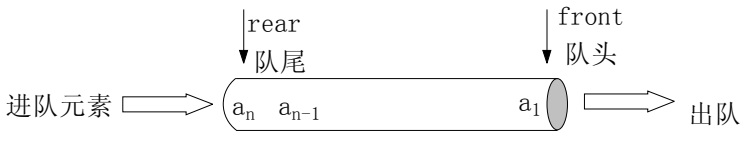


图 1.38 队列逻辑结构

所以，队列的存储结构也有顺序存储与链式存储之分。队列和栈的区别在于，队列的元素只能在队列的一端进，另一端出。所以，队列也是操作受限的线性表结构。队列头尾各有两个指针，分别指示元素从队列尾部进入，从队列头部弹出，称之为先进先出的存储方式。

- **队列的操作**

1・建立一个空队列

front=rear=0;

这里，front 是队列头指针，rear 是队列尾指针。

2・向队列尾推入一新元素

Q[rear]=x;

rear++;

3・从队列头推出一个元素

x=Q[front];

front++;

- **循环队列结构**

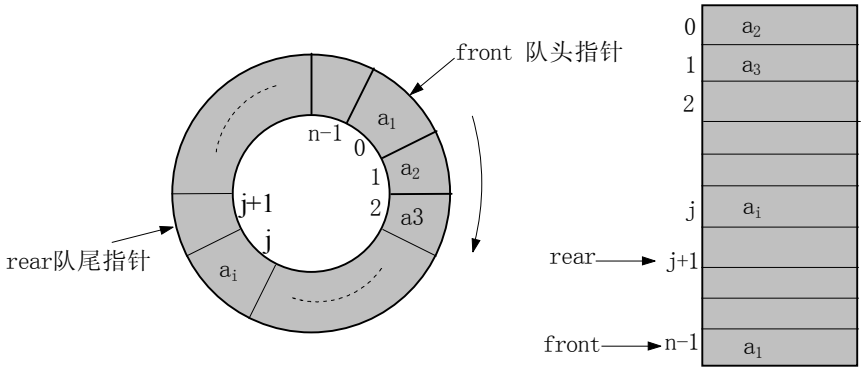


图 1.39 循环队列结构

很多实际问题经常使用如图 1.39 所示的循环队列结构。规定指针是顺时针运动，一个逻辑上的环形队列在物理上既可以采用顺序存储结构，也可以采用链式存储结构实现。每当指针 front、rear 达到存储区尾部时，后续的元素进队操作就将指针调整到存储区头部，当然，如果是循环链结构也就没有尾节点的处理问题。

除去队列初始为空的状态以外，指针 front、rear 不能重叠。否则，初始队空与当前队满的状态无法分别。所以，一般说队列头指针与队列尾指针需要间隔一个空单元。

因 front 和 rear 的相对值只能取 0～n-1，表达队列从初始的空状态（front=rear），到队满状态（front-rear=n-1）。如果认为通过修改 front 和 rear 的定义，就可以让 front 和 rear 表达出 n+1 种不同的状态，根据鸽笼原理这是不可能的。鸽笼原理确定，如果 n 只鸽笼有 n+1 只鸽子，则全部鸽子进入鸽笼时至少一个鸽笼中有 2 个鸽子。所以，当 front 和 rear 定义为队列指针时，front 一定指向队头元素的位置，rear 指向后续入队元素将要

存储的位置，并且用长度为 n+1 个元素的队列存储 n 个元素。所以进出队列的算法是：

初始：设循环队列长度为 M，front=rear=0。

入队：向队列尾推入一新元素

```
int queue_in(int front,int *rear,int *Q,int x)
{
    if(((*rear + 1) % M)==front)return(-1);       // 队列上溢错误
    Q[*rear]=x;
    *rear=(*rear+1) % M;          //rear 指向当前队中最末一个元素后的空单元
    return(0);
}
```

出队：从队列头部推出一个元素

```
int queue_out(int *front,int rear,int *Q,int *x)
{
    if(rear==*front)return(-1); // 队列下溢错误
    *x=Q[*front];
    Q[*front]=0;              //清除为零
    *front=(*front + 1) % M;//front 指向当前队头元素的位置
    return(0);
}
```

1.2.3.2 队列应用

例 1.10 循环队列。顺序存储的循环队列 Q 结构如图 1.40。设长度为 10,队列有头指针 front 和一个记录队列中节点个数的计数器 count，初始 front=0，count=0。元素进出队列操作算法如下：

进队列描述为：

```
if(count==10)队列满处理;
else{
    count+=1;
    p=(front+count)% 10
    Q[p]= 进队列元素;
    }
```

出队列描述为：

```
if(count==0)队列空处理;
    else{
        front=(front + 1)% 10
        count-=1;
        x= Q[front];
    }
```

现问，该循环队列最多能容纳的元素个数。

解：

我们知道一个长度为 N 的循环队列存储元素个数是 N-1,因为队列头指针与队列尾指针

需要间隔一个空单元来判别队列是满还是空，本题正是考查读者对这个概念的理解。题中的顺序存储队列只有一个队头指针，尾指针用一个计数器取代，它和指针的区别是具有空队列判别能力。注意初始 count=0，front=0，以后，当队列为空时 count=0，当队列为满时 count=10（队列位置下标从零开始），因此，该队列没有指针间隔问题，所以它最多能存储的元素个数是 10。

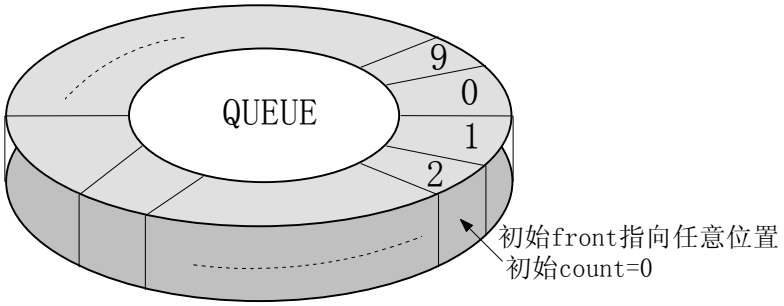


图 1.40 使用计数器控制循环队列

例 1.11 滚动显示。工业记录仪和心电图仪的波形是队列结构的先进先出滚动显示方式，循环队列深度为 n，满屏时指针 sampling 指向队列尾端，采样信号到来时刻，当前采样数据被写入队列尾部，最旧的数据点在队列头部（屏幕顶端）被推出，每次显示时，指针 scan 总是从头部开始扫描输出整个队列的 n 点数据，并且，在没有最新数据点写入时，队列的数据被重复扫描输出。所以，画面只在有新采样点进入时被逐点滚动更新，如图 1.41 所示。

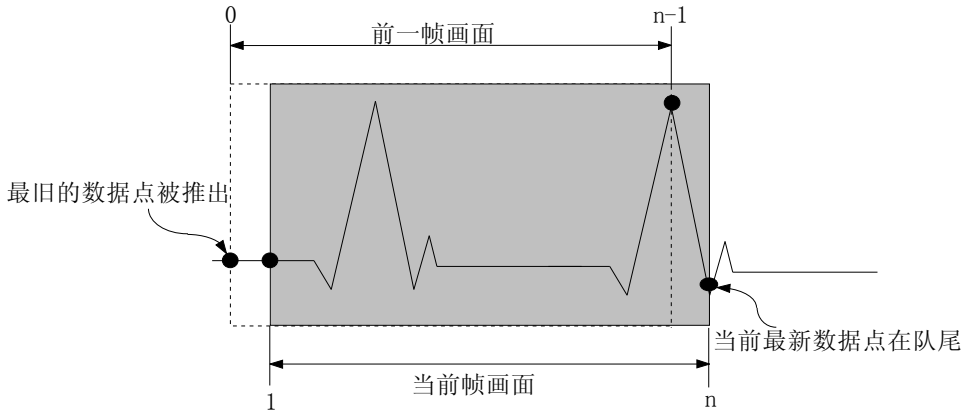


图 1.41 滚动显示原理

求（1）：根据题图画出长度为 n 的循环队列结构，标出 Sampling 和 Scan 指针位置。

求（2）：请描述最新采样数据点被写入时，Sampling 和 Scan 指针的动作过程（可以忽略实际地址关系）。

解 1：

滚动采样广泛应用于工业记录仪和医用心电图显示，采样数据形成一个队列，队列头对应屏幕左端，队列尾对应屏幕右端。没有新采样点进入的时候数据被重复显示，一个新采样点进入的时候被置入队列尾，同时，队列头一个数据点出队列。本题是考查同学对队列头和队列尾两个指针动作的关联性理解，当元素进出队列没有时序上的关联时，俩个指针动作是独立进行的，当元素进出队列有时序关联的时候，指针操作也同样具有关联，依

题意，滚动采样进出队列是同步进行的，则需要同步调整两个指针指向即可，见图 1.42 所示。

解 2：

指针同步调整方式为：

Sampling++;

Q[Sampling]=数据;

Scan= Sampling+1;

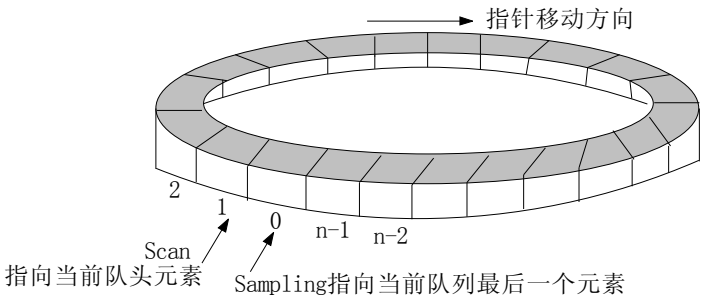


图 1.42 长度为 n 的滚动采样循环存储器指针位置

例 1.12 迷宫问题。n*m 迷宫是一个矩形区域，如表 1.3 所示（绿色区域是程序 1.18 搜索路径示意）。矩阵元素（1，1）为入口，（n，m）为出口，0 表示该方格可通过，1 表示该方格有阻碍不能通过。规则是每次只能从一个无障碍方格向其周围 8 各方向的邻接任一无障碍方格移动一步，问，当迷宫有解时如何寻找一条由入口到出口的路径并返回这个序列，或者不能连通时给出无解标志。求①提出思想；②给出算法。

表 1.3 一个带边界哨的 10×15 迷宫

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

解：

首先表 1.3 可以用 2 维数组表示 array[n][m]，其中 array[0][0]～array[0][m]，array[0][0]～array[n][0]，array[n][0]～array[n][m]，array[0][m]～array[n][m]，这两行两列是我们设置的边界哨，取值为 1。

解题思路。初始我们不知道从哪一个方格沿哪个方向走，并通过邻接的方格逐步可以达到最后出口。但是，可以，如果把它每一步所有可能走的方向上的邻接方格都排列起来，

下一步是把那些邻接方格它们所有可能走的方向上的所有邻接方格也继续排列起来，于是：

① 如果一个方格任何方向上都没有可走的邻接方格，我们就不再需要它，剔除；

② 同样，一个方格所有可能走的方向上的邻接方格都已经列出来后，我们已经知道了后续搜索方向，也就不再需要这个方格，也可以剔除；

③ 重复下去，我们会搜索到所有能走的到方格，当发现有一个方格已是出口后停止；

④ 选择数据结构。因为先搜索到的方格在排列出其后续要搜索的邻接方格之后，先被剔除，如果把排列方格看成是按顺序进入一个队列，剔除方格就等于是队列头部元素的出队操作，所以可以用队列结构解决迷宫路径搜索问题。

⑤ 搜索方向设置。设置一个队列 sq[]记录搜索路径，从数组 array[1][1]开始搜索时将每个方格下标推入队列，以（i,j）为中心向 8 个邻域搜索时下标变化如表 1.4 所示。

表 1.4 以（i，j）为中心搜索 8 个邻接区域时下标增量设置

7		0		1
	i-1,j-1	i-1,j	i-1,j+1	
6	i,j-1	i,j	i,j+1	2
	i+1,j-1	i+1,j	i+1,j+1	
5		4		3

算法步骤。首先定义节点及变量：

```
struct node{
        int x,y;     //方格点坐标
        int pre;          //前趋点指示
        };
int array[N+2][M+2],i;  //迷宫数组
struct node sq[M*N];     //既是纪录搜索路经的数组又是队列结构（非循环队列）
//初始化方向增量
int zx[8]={-1,-1,0,1,1,1,0,-1};  //zx[0]=-1;zx[1]=-1;zx[2]=0;zx[3]=1;
                                 //zx[4]=1;zx[5]=1;zx[6]=0; zx[7]=-1;
int zy[8]={0,1,1,1,0,-1,-1,-1};  //zy[0]=0; zy[1]=1;zy[2]=1;zy[3]=1;
                                 //zy[4]=0;zy[5]=-1;zy[6]=-1;zy[7]=-1;
```

初始化迷宫边界（略）。

程序 1.19 搜索迷宫函数

```
int seartch(int array[N+2][M+2],struct node *q,int zx[8],int zy[8])
{
    int x,y,i,j,v;
    int front=1,rear=1;                    //设置队列指针
    struct node sq[N*M];                  //建立一个队列
```

```
    sq[1].x=1;
    sq[1].y=1;
    sq[1].pre=0;                          //从迷宫入口开始搜索
    *(*(array+1)+1)=-1;                   //第一行第一列，已经搜寻过方格的标记
    while(front<=rear){
        x=sq[front].x;
        y=sq[front].y;                    //当前搜索方格的坐标
        for(v=0;v<8;v++){                 //搜索8个方向邻接点
            i=x+zx[v];
            j=y+zy[v];                    //求得邻接点坐标
            if(*(*(array+i)+j)==0){
                rear++;                   //该方格可走，准备进队列
                sq[rear].x=i;
                sq[rear].y=j;
                sq[rear].pre=front;       //该方格的前趋
                *(*(array+i)+j)=-1;       //当前方格已搜索过，打上标记防止重复搜索
                }
            if((i==N)&&(j==M)){
                i=rear;                   //从队列尾开始列出路经
                j=0;
                while(i){
                    (q+j)->x=sq[i].x;     //返回的q中是搜索走过的路径
                    (q+j)->y=sq[i].y;
                    i=sq[i].pre;          //根据前驱指针回朔
                    j++;
                    }
                return(--j);
                }
            }
        front++;//8个方向搜索完毕，该a[ij]元素出队列，注意只是指针在移动，
                //搜索过的方格坐标仍在sq[]内，
                //front指向的sq[]中是已经列出的所有可能走的方向上的后续方格位置，
                //程序循环搜索它们8个方向上的所有可能走的后续方格位置
        }
    return(-1); //走出循环体是因为迷宫无解
}
```

程序调用返回值非负时表示迷宫得解，返回的q中是搜索路径过程中走过的点的行列坐标，而返回值指向搜索路径起点，下面是程序1.19对表1.3所示迷宫的运行结果：

迷宫：

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

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

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

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

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

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

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

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

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

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

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

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

搜索开始

(1,1)(1,2)(1,3)(2,4)(1,5)(1,6)(2,7)(3,7)(4,8)(5,8)(6,9)(7,9)(8,9)

(9,10)(8,11)(9,12)(10,13)(9,14)(10,15)

搜索完毕

这是宽度优先求解迷宫问题，如以深度优先为准则，请读者考虑如何用堆栈构造带方向加权的深度优先算法求解迷宫问题。要求①给出迷宫求解两种方法的c语言程序；②在根据数据分析比较两种算法效率。

- **多进程下的管道通讯结构**

在 UNIX、NT 操作系统中，管道是一个通讯通道，它把一个进程与另一个进程连结起来，这样，一个进程能用系统调用函数 write()将数据输入管道，另一个进程用系统调用函数 read()在管道另一端把数据读出，从而实现进程之间的数据通讯。管道与消息缓冲区的不同在于它以文件为传输介质，以自然流进行数据传输而不是以消息为单位，它工作方式如图 1.43，是 FIFO 的队列操作形式。

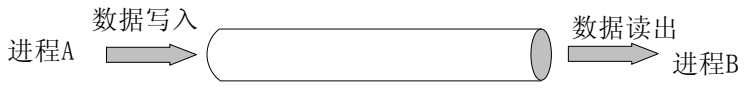


图 1.43(a)进程通讯的管道结构

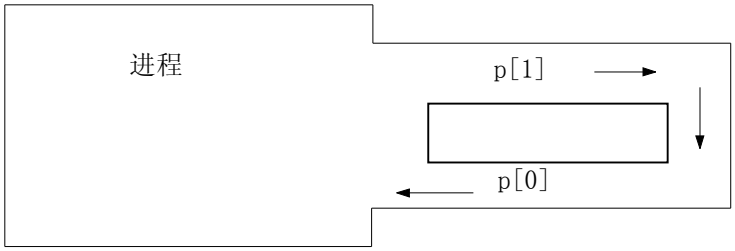


图 1.43(b)在进程中打开的管道是双向的

管道以队列方式工作，但它却是一个双向操作的队列，系统调用函数 pipe()建立一个管道，它返回两个文件描述符来代表输出、输入文件用于对管道的读写操作。

```
int filedes[2],retval;
retval=pipe(filedes);
```

系统调用成功后，filedes[0]用于从管道读，filedes[1]用于向打开的管道写数据文件。Retval 值为-1 则调用失败。管道一旦建立，就可以直接用 read()，write()对其操作。

1.3 非线性数据结构--树

1.3.1 概念与术语

1.3.1.1 引入非线性数据结构的目的

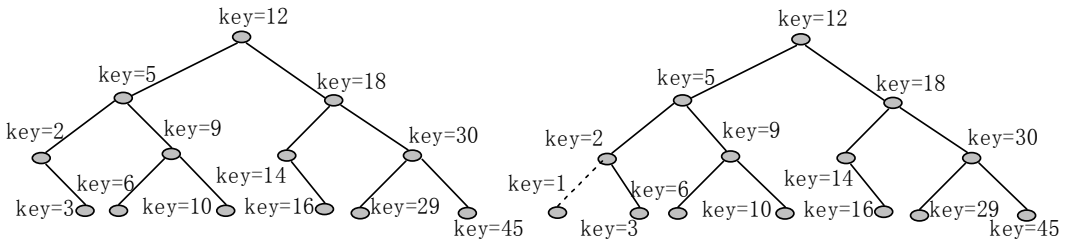
在树的内容中我们重点讨论二叉树。因为任何一棵树都可以转换为二叉树，因此，学习重点是二叉树的有关概念及生长、删除方法。与线性结构不同，二叉树的概念比较难于理解，同样，它的程序设计也要复杂得多。此外，我们要强调的是二叉树是一个动态的生长过程。

引入非线性数据结构二叉树的目的是改善数据结构的操作效率。我们知道任何一门学科上都存在着线性结构简单但是运行效率低的问题，在数据结构的检索操作上，链表和顺序表的平均检索效率都是O(n)数量级,因为程序算法是一个循环体结构,如在数组array[n]中检索关键字等于 key 的节点过程是：

```
array[n].key=key;          //设置监视哨
while(array[i].key !=key)i++;
return(i);                 //找到节点后返回它在数组中的位置
```

当 n 很大时检索效率非常低。有一些方法可以改善检索效率，对一个长度为 n 且有序的顺序表使用对半检索方法，其效率是$long_2n$量级的（详细见检索章节内容）。但插入一个节点平均花费的时间还是 0（n），因为要移动节点。同样，链表插入节点所需的搜索时间也是 0（n），如果我们想得到插入和检索效率均在$long_2n$量级的，就必须考虑改变数据结

构的形式，即按树型结构组织。在一个有序的二叉树中检索和插入一个节点的平均效率就是$long_2n$量级的，因为从根开始每达到一层，它都将后续的子树节点数量作了对半分割，见图 1.44 所示，不过，其前提条件为树是平衡（或者基本平衡）的。所以，构造一棵二叉树有两个要点：①节点有序；②结构平衡。



(a)一个平衡二叉树其左右子树节点数基本相等 (b)插入一个节点只需修改指针，效率和检索操作相当

图 1.44 二叉有序树

1.3.1.2 树的定义与术语

- 定义：树是一个或多个节点组成的有限集合，其中：
  ① 必有且仅有一个特定的称为根（root）的节点。
  ② 剩下的节点被分成 m≥0 个互不相交的集合 T1，T2，...，Tm，而且其中的每一元素又都是一棵树，称为根的子树（Subtree）。
- 节点：树的元素，含有数据域和多重指针域。
- 节点的度：某一节点所拥有的子树个数。
- 树的度：在一棵树中最大的节点度数是树的度。
- 叶子：树的端节点，其度为零。
- 孩子：节点子树的根。
- 双亲：孩子节点的前驱节点。
- 节点层次：从根算起节点在层中的位置，root 层次为 1。
- 深度：树中节点所具有的最大层次。
- 有序树：树中节点同层间从左至右有次序的排序，不能互换。
- 树的存储结构：树是一种非线性结构，它的物理存储结构有多种形式，基于内存的效率一般用链式存储结构实现树的存储，并且要求所有节点是同构的，即指针域与节点的度数无关，选取树的度为指针域的个数。如图 1.45（a）的树结构有图 1.45（b）的链式存储结构。因为指针域的个数 R 有下列关系成立：

R＝节点数 n×树的度 k

所以非空指针域是 n-1（指向除去根节点以外的 n-1 个节点），而空指针域 m 是：

m= n×k －n+1

= n(k-1)+1

因此，k 越大所用存储区域就越多，浪费也越大。

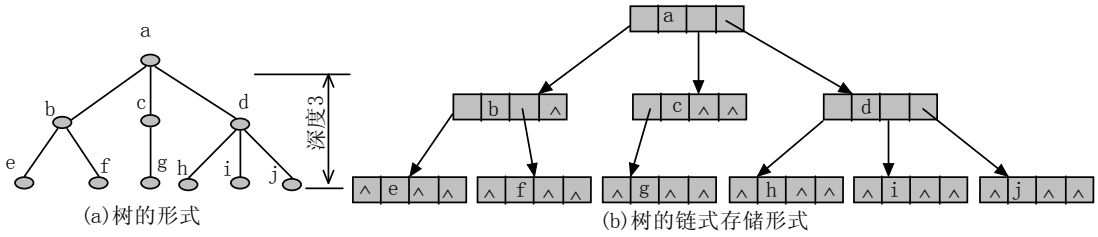


图 1.45 树的逻辑结构和存储结构

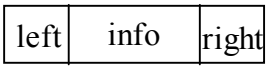
1.3.1.3 树的内部节点与叶子节点存储结构问题

一般我们要求树的内部节点与叶子具有相同的存储类型，即同构，但是在很多应用中内部节点存储的信息与叶子节点存储信息不同，比如，一个 m 阶 B^+树，其内部节点存储的是索引叶子节点的占位符、指向子树的长度为 m-1 的指针数组，以及一对左右指针。而叶子节点存储的是数据，或者是长度不定的一组记录的关键码与指向记录内存位置的指针对。为了达到节点同构的要求，在 c 语言中处理的方法是共用体(union)数据类型。使用共用体定义内外节点结构的最大弊病，是当叶子节点和内部节点的子类型长度差别很大的时候，共用体结构存在着大量的空间浪费问题。在 C++语言中，利用 C++的类继承，在 C++中基类给出对象的一般定义，而子类比基类增加一些细节，即，基类可以定义为一般的节点，而子类可以定义为内部节点还是叶子节点。详细读者参考 C++教材。

1.3.2 二叉树

1.3.2.1 二叉树基本概念

- 定义：二叉树是 n(n≥0)个节点的有限集，它或为空树，或由一个根节点和两棵互不相交的左右子树组成。
- 二叉树的存储结构：



- 二叉树的主要性质：

①二叉树的第 i 层上至多有 2^{i-1}个节点。

证明：i=1，则 $2^0=1$，第一层有 1 个节点。设 i-1 层上有 2^{i-2}个节点，由于二叉树每个节点的度数最大为 2，所以，第 i 层上节点数目最多是第 i-1 层上的 2 倍，为 2^{i-1}个节点。

②深度为 h 的二叉树中至多有 2^h-1 个节点。

证明：由①可得：深度为 h 的二叉树含有节点数 n 为：

$$n \le \sum_{i=1}^{h} 2^{i-1} = 2^h - 1$$

③任意一棵二叉树有关系 $n_0=n_2+1$ 成立。n_0 是叶子数，n_2 是度为 2 的节点数。

证明：因为节点数为 n 的二叉树除去根结点以外，其余的每一个节点一定有一个指针指向它，设指针总数为 b，显然 $b=n_1+2n_2$，所以：

$$n = n_1 + 2n_2 + 1$$
$$n = n_0 + n_1 + n_2$$

两式相减得：

$$n_0 = n_2 + 1$$

- 几种特殊性质的二叉树

①满二叉树。深度为 h 且含有 2^h-1 个节点的二叉树为满二叉树，如图 1.46 所示。有的教材对满二叉树有另外的定义：如果一棵二叉树的任何节点，或者是树叶，或者有两棵非空子树，则此树为满二叉树。根据此定义则图 1.46（b）的结构也是满二叉树。理论上讲，上述定义没有问题，因为它的任何一非叶子节点都是满的，但跟下面要看到的完全二叉树不好对应，所以我们还是以图 1.46（a）的定义为基准。

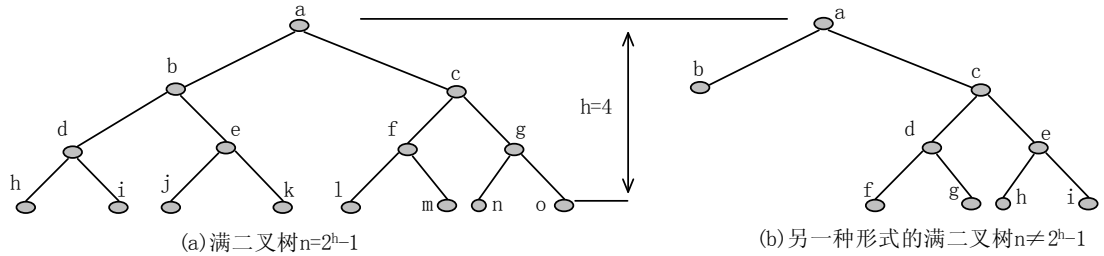


图 1.46 满二叉树的形式

②完全二叉树：具有 n 个节点，深度为 k 的二叉树，当且仅当其所有的节点对应于深度为 k 的满二叉树中编号由 1 至 n 的那些节点时，称为完全二叉树，如图 1.47 所示，显然满二叉树也是完全二叉树。

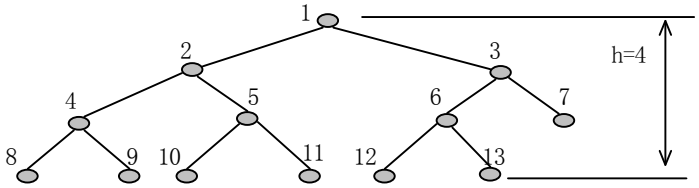


图 1.46 完全二叉树

③平衡二叉树（AVL 树）：一个平衡二叉树是空树，或者是具有下列性质的二叉树，即它任一节点的左右子树都是平衡的，且左右子树的深度之差的绝对值≤1。如同二叉树的定义一样，平衡二叉树也是一个递归的定义，这一类的二叉树构造很困难，更困难的是随着节点的插入、删除操作，维护树的平衡非常复杂。

- 二叉树的形态

二叉树与树的不同点在于它有左右子树之分，其基本形态是图 1.48 所示 5 种。

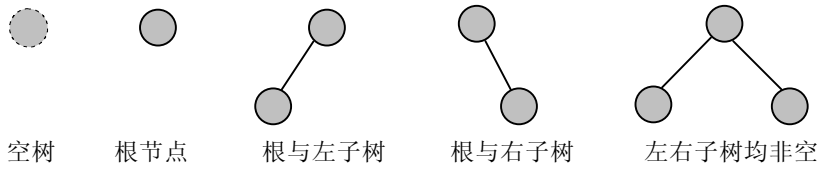


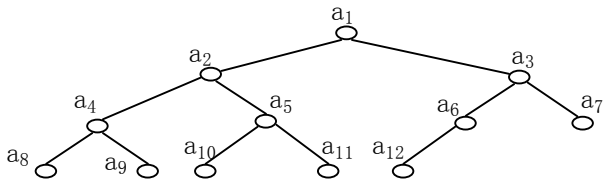
图 1.48 二叉树的 5 种形态

- 树向二叉树的转化

任意一棵树都可以找到唯一的一棵二叉树与之对应，反之亦然。转化的方法读者可以参考其它教材，本书不详细讨论。

1.3.2.2 完全二叉树的顺序存储结构

在顺序表一节中我们讨论了树的顺序存储形式，如果是一棵完全二叉树，那么它的顺序存储结构非常简单，从完全二叉树的定义知道，除去底层以外，其每一层的节点都是满的，并且我们限定底层的叶子是从左至右排列的，则 n 个节点的完全二叉树可以用大小为 n 的数组存放，节点的逻辑编号就是节点存储位置的数组下标（如树根编号由 1 开始，则其存储位置的数组下标为节点编号减一），根据节点所在的层次 r 可以通过简单的计算得到其亲属关系节点，图 1.49 是用顺序存储结构存储一棵有 12 个节点的完全二叉树例子。



（a）逻辑结构

0	1	2	3	4	5	6	7	8	9	10	11
a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}

（b）存储结构

图 1.49 一棵完全二叉树的顺序存储形式

表 1.5 图 1.49 完全二叉树的顺序存储下标增量

数组下标	0	1	2	3	4	5	6	7	8	9	10	11
父节点	–	a1	a1	a2	a2	a3	a3	a4	a4	a5	a5	a6
左孩子	a2	a4	a6	a8	a10	a12	–	–	–	–	–	–
右孩子	a3	a5	a7	a9	a11	–	–	–	–	–	–	–
左兄弟	–	–	a2	–	a4	–	a6	–	a8	–	a10	–
右兄弟	–	a3	–	a5	–	a7	–	a9	–	a11	–	–

我们给出每个节点其亲属（父节点 parent，左孩子 L-child，右孩子 R_child，左兄弟 L_ sibling，右兄弟 R_ sibling）在数组内的地址偏移 r 计算公式如下（r=0，1，2，…，n-1）：

$$parent(r) = \left\lfloor \frac{r-1}{2} \right\rfloor \qquad 0 < r < n$$

$$L\_child(r) = 2r + 1, \qquad 2r + 1 < n$$

$$R\_child(r)=2r+2, \qquad 2r+2<n$$

$$L\_sibling(r)=r-1, \qquad r\text{为偶数且}0<r<n$$

$$R\_sibling(r)=r+1, \qquad r\text{为奇数且}r+1<n$$

比如，根节点 a_1（地址偏移 r=0）没有父亲，其左孩子的址偏移是 1，右孩子的址偏移是 2；而节点 a_4 父亲的地址偏移是 1，其左孩子的址偏移是 9，右孩子的址偏移是 10。如表 1.5 所示。

1.3.2.3 二叉树遍历

二叉树的操作有多种，但最重要几种运算是插入节点，删除节点和遍历操作。

插入函数是将节点插入到根为 root 的二叉树中，因为二叉树是有序的，故插入后仍要保持它的有序性。删除函数是从根为 root 的二叉树中删除一关键字为 key 的节点，要求删除后的二叉树仍然有序。而遍历可以说是二叉树中最重要的一种操作。前面提到的插入与删除都是在有序二叉树中进行的，所谓有序就是指按遍历的顺序做插入与删除。

遍历的定义：访问二叉树中每个节点一次且仅一次的过程称为树的遍历。按根节点访问的先后定义有先序遍历、中序遍历和后序遍历，注意遍历的定义是递归的。

- 先序遍历

  访问根节点

  先序遍历左子树

  先序遍历右子树

- 中序遍历

  中序遍历左子树

  访问根节点

  中序遍历右子树

- 后序遍历

  后序遍历左子树

  后序遍历右子树

  访问根节点

设二叉树结构如图 1.50 所示，节点 treenode 定义如下，则中序遍历函数如程序 1.20。

```
struct   treenode{
            int  info;
            struct  treenode *left, *right;
            };
```

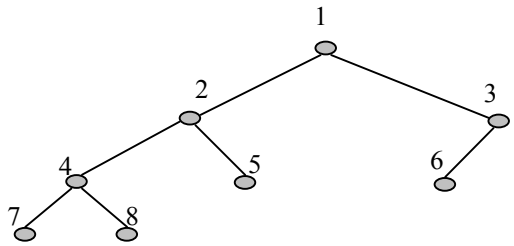
程序 1.20 中序遍历二叉树

```
struct  treenode  *inorder(struct  treenode  *root)
{
      if(!root)return(0);          //递归出口条件
      inorder(root->left);         //先遍历左子树
      printf("%d", root->info);    //操作是打印数据域值
      inorder(root->right);        //再遍历右子树
}
```

程序一直递归调用到叶子，未满足出口条件就继续调用叶子的左指针，如果进入函数判别为空，就是递归到叶子的返回出口，可以进行叶子遍历操作。然后调用叶子的右指针，也是为空返回，即叶子的父节点的左分枝中序遍历结束，对其父节点进行遍历操作之后，进而中序遍历父节点的右分枝，结束后逐级回归。图 1.50 的递归过程见图 1.51 所示。

遍历操作有多种形式，程序 1.20 出数据域的值到屏幕。

树是递归结构的，且遍历定义也是递归的，故用递归设计程序结构简单明了。我们要求关于二叉树的操作函数都用递归的形式给出，请读者参考程序 1.18 设计二叉树的前序、后序遍历程序。



中序遍历：74825163
先序遍历：12478536
后序遍历：78452631

图 1.50 遍历一棵二叉树

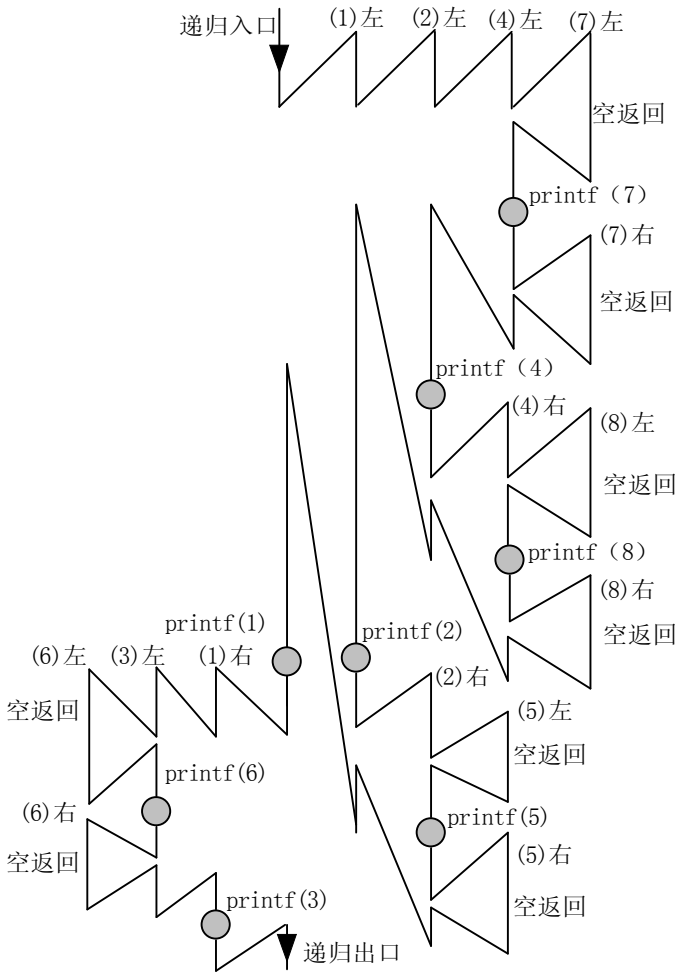


图 1.51 中序遍历图 1.26 二叉树时递归调用过程

1.3.2.4 二叉树唯一性问题

假设我们已知一棵二叉树的前序和中序遍历序列，是否可以唯一的恢复这棵二叉树？这是二叉树唯一性问题。由一棵二叉树的前序和中序遍历序列，可以唯一的恢复一棵二叉树。证明如下：

首先看图 1.52 所示的同一棵二叉树的前序和中序遍历序列。

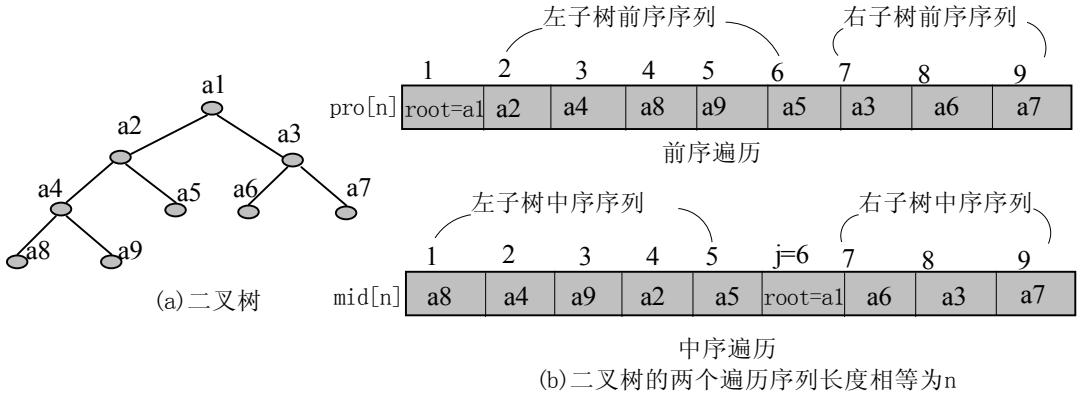


图 1.52 二叉树的前序和中序遍历序列

当节点数为 1 的时候，显然序列里就是根节点，它们是唯一的。假设节点数为 n-1 时由前序和中序遍历序列可以唯一确定二叉树，那么节点数为 n 的时候，假设此时的根节点在

中序序列的位置是 j，即 pro[1]=mid[j]，因为序列：

mid[1], mid[2],..., mid[j-1]

同时是 pro[1]左子树的中序序列，而序列：

mid[j+1], mid[j+2],..., mid[n]

同时也是 pro[1]右子树的中序序列，两个序列长度相等，仅是排列不同。所以，我们可知 pro[2]～pro[j]是左子树的前序序列，而 pro[j+1]～pro[n] 是右子树的前序序列。

显然，j-1<n，n-j<n，根据归纳假设，由 mid[1], mid[2],..., mid[j-1]和 pro[2], pro[3],..., pro[j]它们能唯一确定左子树。同理，由 mid[j+1], mid[j+2],..., mid[n]和 pro[j+1],..., pro[n] 能唯一确定右子树。

所以，当节点数为 n 的时候，根据前序和中序序列能唯一的确定一棵二叉树。照此容易证明，已知中序和后序序列能唯一的确定一棵二叉树。

1.3.3 二叉排序树

1.3.3.1 基本概念

我们已经讨论了二叉树的基本概念与遍历操作。用输入节点序列构造一棵二叉树的过程，是根据它的生长规则在二叉树中找到节点位置并插入的过程。因为我们总是建立一个有序的二叉树，所以，二叉树的每一节点都用一关键码表征，并且让二叉树按关键码有序生长。我们称为二叉排序树。

二叉排序树或者是一棵空树，或者其任何一个节点都具有下列性质：

（1） 若它左子树不空，则左子树上每一节点的关键码均小于它的根节点关键码；

（2） 若它右子树不空，则右子树上每一节点的关键码均大于它的根节点关键码；

（3） 其左右子树也分别是二叉排序树。

二叉排序树的定义是递归形式的，因此，其程序设计也是递归的。设节点序列输入为{45，23，53，45，12，24，90}，则按二叉排序树规则，其生长过程如图 1.53 所示，特点是：

（1） 第一个输入的节点是树根；

（2） 节点总是被插入到叶子位置，不用移动其它节点（没有平衡约束条件）；

（3） 不同的关键码输入序列其生长的二叉排序树不同。

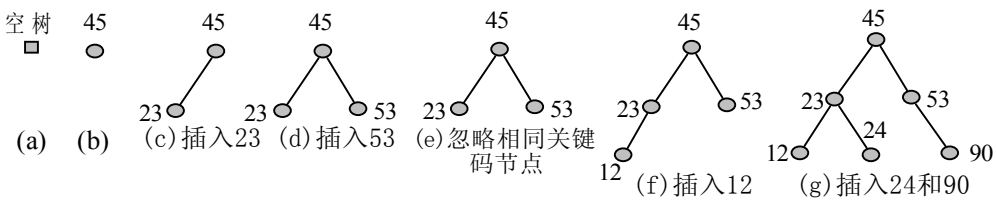


图 1.53 一棵二叉排序树的生长过程

二叉树在插入和删除节点的过程中不需要移动节点，而且从根节点开始搜寻要插入或者

删除的节点位置所走过的结点数目，也少于链表，因此说二叉树操作效率优于线性表。

如果将上例中的输入序列改为如{90，53，45，24，45，23，12}，根据二叉排序树规则，其生长过程退化为一单链。需要注意的是树作为动态生长的数据结构，节点数据输入序列的变化其数据结构特征也会相应的变化。

1.3.3.2 程序设计

一、节点定义

设有二叉树节点定义为:

```
struct tree{
        int  key; //关键码
        struct  tree *left, *right; //左右指针域
        };
```

二、插入一个节点

由图 1.53 可知,二叉树插入操作是寻找叶子结点的过程。程序进入时 S->key 为关键码，按序搜索二叉树，找到当前节点 S 要插入的位置，因为必定是一个叶子的左分枝（S->key 小于该叶子节点关键码）或右分枝（S->key 大于该叶子节点关键码值），或者是根节点（树为空时），因此，将 S 插入就是简单的修改二叉树某个叶子节点的指针指向 S，使 S 成为新的叶子。

程序 1.21 二叉排序树插入函数

```
struct tree *addtree(struct tree *root, struct tree *r, struct tree *s)
{               //递归中 root 是当前节点 r 的父亲
    if(!r){             //根、叶子节点时进入，S 被插入在根或叶子上
        r=s;
        r->left=NULL;    //空树被插入根、非空被插入到叶子，它们左、右指针均空
        r->right=NULL;
        if(!root)return(r); //空树时返回插入的根
        if(r->key<root->key)root->left=r;//叶子，入左节点
        else    root->right=r;//叶子，入右节点（相等情况可以不考虑）
        return(root);
        }
    else{           //非空且非叶子，则开始递归寻找叶子
      if(s->key<r->key)addtree(r,r->left,s);        //关键码小于父节点搜索左子树
      else if(s->key>r->key)addtree(r,r->right,s);//大于父节点搜索右子树
      }
```

}

这个插入程序的关键是一个递归结构中，根据排序规则由子树的父节点、子树的左右分枝一直搜索到当前 S 节点要插入的位置。即：

addtree(子树父节点，子树左或右分枝，插入节点);

三、删除二叉排序树节点

在二叉排序树的插入过程中,每次插入的节点都是成为叶子,不用移动其它节点的位置。但删除操作过程就完全不同。我们删除节点的前提是保证删除以后的二叉树仍然是二叉排序树。因此，一个节点的删除一般要改变一些节点在二叉树中的位置，或者说二叉树的结构会有相应的变化，所以程序设计也要因考虑各种情况而变得复杂起来。

要保证删除节点后二叉树仍然有序的原则，我们分下面几种情况讨论：

（1） 被删节点度为 0。删除叶子，这种情况容易处理，只要修改双亲的指针，置空。

（2） 被删节点度为 1，如图 1.54 所示。被删除节点只有单分枝子树，若是左分枝，则应把双亲指针指向被删节点左分枝子树的根；若是右分枝，则应把双亲指针指向被删节点右分枝子树的根。

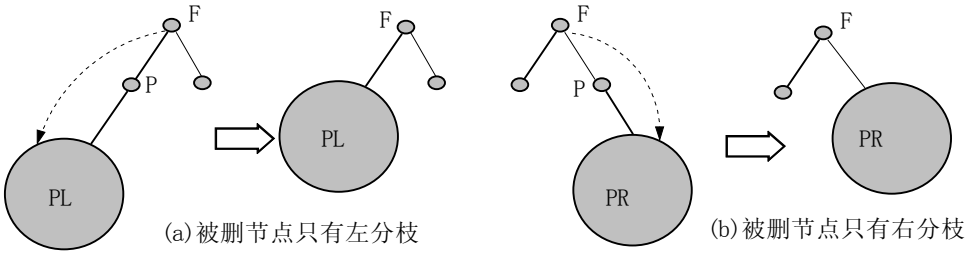


图 1.54 被删除节点度数为 1 的情况

图 1.54（a）中的删除操作结果应该是：

F→left=P→left;

二叉树程序设计是递归结构的。从根开始，每搜索一层就是递归调用自己一次的过程，给其父节点（这里的 F 节点就是被删除节点 P 的父节点）赋值就是函数的返回值：

```
if(root->key==key){              //找到被删节点，当前子树的根就是被删节点 P
      if(root->right ==NULL){ //若右分枝空就处理左子树
          p=root->left;          //取被删节点左子树的根
          free(root);            //释放
          return (p) ;     //返回给当前 root 的父节点的左或右指针赋值
          }
```

同样，图 1.54（b）中的删除操作结果应该是：

F→left=P→right;

那么给 F 节点赋值是：

```
if(root->key==key){              //找到被删节点，当前子树的根就是被删节点 P
```

```
if(root->right ==NULL){ //右分枝空
    p=root->left;       //取被删节点左子树的根
    free(root);
    return(p);          //返回给当前 root 的父节点的左或右指针赋值
    }
```

（3） 被删节点度为 2。左右分枝均非空,删除节点后必须重新对二叉树排序，方法有三种：

①按二叉排序树的规则，可以继承该节点的是其左子树中关键码最大的节点，显然，这是 P 节点左子树的最右端节点 S，它或者为叶子或者右分枝为空，见图 1.55 所示。图 1.56 给出了一个具体例子。指针修改如下：

S_{root}->right=S->left;//如 S 度为 1 有左分枝，则应将左分枝赋给其父节点 S_{root} 的右指针

S->left=P->left;　　//注意与前一条语句顺序不能颠倒

F->left=S;　　//如果 P 节点是 F 的右分枝，则应该是对 F 的右指针赋值

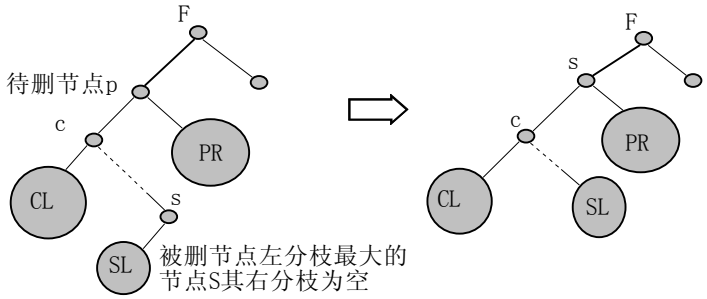


图 1.55 被删除节点度数为 2 的情况（1）

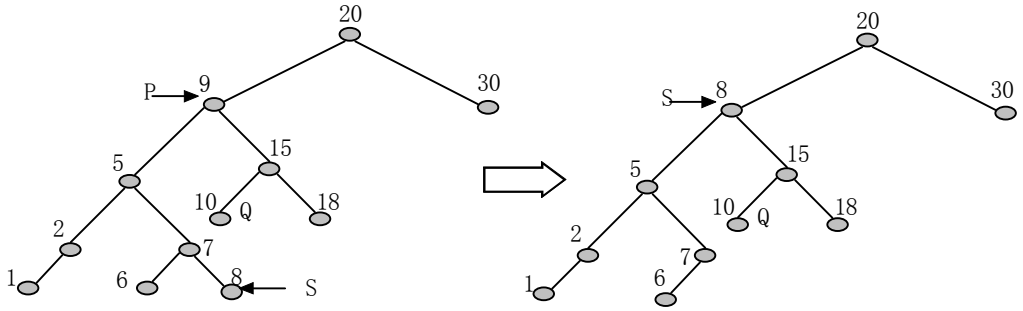


图 1.56 删除节点 P 的左子树最大节点 S

②同样由二叉排序树的规则，我们也可以用 P 节点右子树的最小关键码节点来继承 P 节点位置（节点 Q 或者是叶子或者是左分枝为空）。按照上面的例子，被删除节点的右子树最小节点是 Q＝10，因此得如图 1.57 所示的二叉树结构。

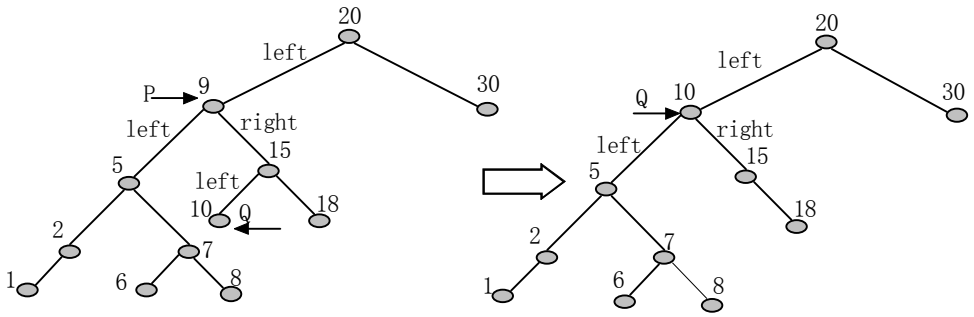


图 1.57 被删除节点度为 2 的情况（2），用节点 P 的右子树最小节点 Q 取代 P

指针修改如下：

Q_{root}->left=Q->right;//如 Q 度为 1 有右分枝，则应将右分枝赋给其父节点 Q_{root} 的左指针

Q->left=P->left; //与前一条语句顺序无关

F->left=Q; //如果 P 节点是 F 的右分枝，则应该是给 F 右指针赋值

函数参数使用引用方式会使程序设计更为方便，删除一棵二叉排序树最小节点的 c 函数如下：

```
struct tree *deletemin(struct tree *&root)
{
  struct tree *temp;
  if(!root)return(0);
  if(root->left)return(deletemin(root->left));
  else {
    temp=root;
    root=root->right;//root 是被删节点父指针的引用，让它指向被删节点右端
    return(temp);//注意，返回的是指向被删节点的指针
  }
}
```

依据此方法设计的递归结构的二叉排序树节点删除程序如程序 1.22。删除操作是根据输入的关键字值在二叉树中遍历寻找一个关键字值等于输入关键字值的节点，找到后进行删除，遍历的过程是递归的：

```
if(root->key<key)root->right=removehelp(root->right,key); //递归搜索右分枝
else              root->left=removehelp(root->left,key); //递归搜索左分枝
```

需要注意的是，我们对二叉排序树节点插入及删除操作的约束条件是，插入或者删除节点之后，它仍是一棵二叉排序树。我们没有考虑插入与删除节点的操作对二叉树结构特性的影响，即，二叉排序树是否平衡的问题。

没有平衡约束条件的二叉排序树没有实际的应用意义。在文件检索应用中，为防止插入与删除操作之后，检索效率变坏，需要考虑插入与删除节点操作过程中，如何保持树仍然

处于平衡状态的方法，这是 2-3 树和 B 树程序设计要解决的问题。

程序 1.22

```
struct tree *removehelp(struct tree *root,int key)
{
      struct tree *p,*q,*temp;
      if(!root)return(NULL); //递归出口条件，防止二叉树无此被删关键码
      if(root->key==key){         //找到被删节点
        if(root->left==root->right){//度为零只需删除操作
                free(root);
                return(NULL);
                }
        else{
            if(root->left ==NULL){       //左分枝空
                p=root->right;           //取被删节点 root 右子树根
                free(root);
                return(p);               //返回取代 root 的节点指针
                }
            else{
                if(root->right ==NULL){ //右分枝空
                    p=root->left;        //取被删节点左子树的根
                    free(root);
                    return(p);           //返回取代 root 节点指针
                    }
                else{   //左右均非空，找右分枝的最小节点
                    temp=deletemin(root->right);//temp 指向右子树最小节点
                    root->key=temp->key;          //与被删节点值交换
                    return(root);                 //返回取代 root 的节点指针
                    }
                }
            }
        }
      else{          //递归寻找
        if(root->key<key)root->right=removehelp(root->right,key);
```

```
        else              root->left=removehelp(root->left,key);
        return(root);
        }
}
```

程序每次从搜索子树根出发，递归返回的是删除后子树根新指针，要注意当前输入关键码不在二叉树中时应由叶子节点处退出，程序调用形式是：

新树根指针 = removehelp(树根指针,key);

主程序中这个程序调用形式为：

root = removehelp(root, key);

一般说来，我们要求二叉树操作函数都采用递归结构。读者可以通过上机去验证删除操作几种方法的不同之处，前提是删除前后二叉树都是有序的。关于二叉树的上机试验，要求读者做到节点的插入、中序遍历、删除、检索及二叉树打印输出等基本功能。

- **基本概念例题**

例题 1.12 现有一棵 n=28 节点的二叉树，问：

（1） 它的叶子节点数最少是多少，发生在什么情况下？

（2） 它的叶子节点数最多为多少？

（3） 它的深度最小为多少？

（4） 它的深度最大为多少？

解 1：两个基本关系：

$$n_0=n_2+1$$

$$n=n_0+n_1+n_2 \qquad \text{或者 } n_0=(n-n_1+1)/2$$

所以，n_1最大时 n_0最小，当 $n_2=0$，既全部是度为 1 的节点时退化为单链，仅有一个叶子，故 $n_{1(min)}=1$。

解 2：当 n_1最小时 n_0最大，即完全二叉树，显然满二叉树时 $n_1=0$，但 n＝28，不符合 2^h-1 的条件，故 n_1不为零。设深度为 k，因为：

$$4< \log_2(28+1)< 5$$

所以，$4< k\leqslant 5$，即 k=5。因此到 k-1 层共有节点数 $n=2^4-1=15$，而 28-15=13，在第 5 层上比 $2^{5-1}=16$ 缺 3 个节点。3 除 2 有商为 1，即第 4 层有 1 个叶子、第 5 层上有 13 个叶子节点，$n_0=14$。

解 3：深度最大为 h=28，单链。

解 4：深度最小为完全二叉树，h=5。

例题 1.13 表达式树。利用指针实现二叉树需要注意叶节点和内部节点是否同类的问题。很多应用中我们只需要在叶节点存储数据而没有指针信息，指针只是在内部节点中作为寻找

叶子节点位置的地址信息使用，还有一些应用，比如表达式树，要求内部节点和叶子节点分别存储不同的数据类型，也需要分别定义叶子节点和内部节点结构。表达式树表示一个代数式，其内部节点数据域用一个字符变量存储运算符加、减、乘、除等，叶子节点只有数据域，定义为一个整型量用来存储操作数，图 1.58 的表达式树表示了 4x（2x+a）-c，问如何实现该表达式树的存储结构，以及输出该表达式的遍历算法。

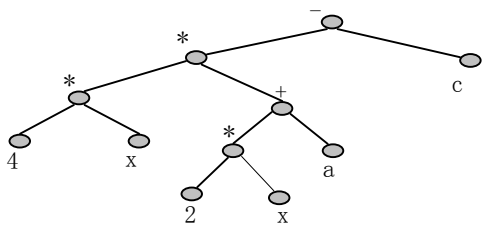


图 1.58 表达式树

解（1）：这是一个节点数据域同构问题，因为字符类型和整型变量长度只差一个字节，我们用共用体实现，定义如下：

```
union{
    int val;
    struct node{
          char symbol;
          struct node *left,*Right;
           }x;
    }y;
```

定义(a)

```
struct node{
        union{
            int val;
           char symble;
            }x;
         struct node *left,*Right;
        }y;
```

定义(b)

你认为两个定义哪个合适？显然是定义（a)。因为我们知道叶子节点不需要指针，定义（b）在使用共用体的同时，对叶子增加了不必要的指针域。

解（2）中序遍历

1.3.4 穿线二叉树

我们在 1.3.1.2 节中指出，一棵树的空闲指针域数量 m= n(k-1)+1，对于二叉树来说就是 m=n+1，即有一半的指针在闲置。为避免浪费这些指针域，我们给它们一个附加功能，让其指向节点的中序前趋和后继节点，具体说，如果一个节点的左指针空，就用其指向该节点的中序前趋节点；如果是右指针为空，就用其指向该节点的中序后继节点。这些附加的指针称之为“线索”，加进线索功能的二叉树称之为穿线二叉树。

要将一棵二叉树线索化，我们可以按中序遍历这棵二叉树，在遍历过程中用线索代替空的指针。图 1.59 是一棵线索化的二叉树。实线表示原来的指针，虚线表示新增的线索，原来为空的左指针都被用来作为指向节点中序前趋的线索。原来为空的右指针被作为指向节点中序后继的线索。

新的问题是，我们如何区分这些指针域存的是指针还是线索？很容易，地址总是正的，所以指针域为正值的话，存的就是指针。如果是线索，我们就取值为负（通过取反得到其

中序前趋或后继节点的地址)。图 1.59 中节点 c 的左指针域仍然为空，表示它是中序序列中的第一个节点，没有前趋。节点 e 的右指针域也为空，表明它是中序序列中的最后一个节点，没有后继。

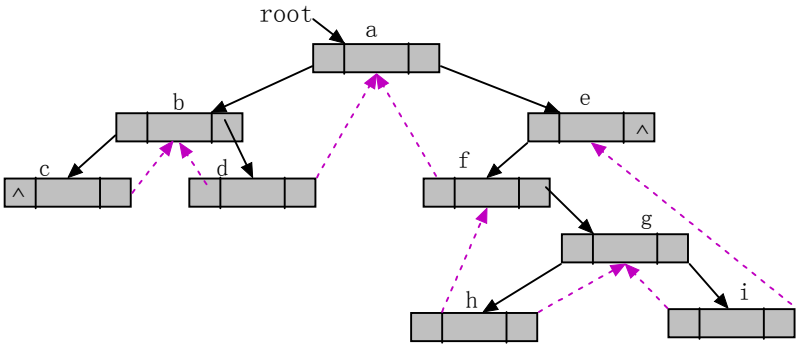


图 1.59 中序线索树

1.3.4.1 二叉树的中序线索化

首先定义节点结构：

```
struct tree{
        int key;
        struct tree left,*right;
        }
```

中序遍历二叉树的顺序是遍历左子树，遍历根，遍历右子树。所以，我们使用一个栈记忆遍历过程中的向左下降序列，就可以得到每个子树的根序列。

算法：

① 初始化

```
p=root;pr=Null;top=-1;
```

② 访问节点，建立线索

```
while(1){
    while(p){push(stack,p,top);p=p->left;}//寻找中序序列第一个元素
    if(top==-1)算法结束;
    else{
        top=pop(stack,p,top); //上升一层
        if(pr){
            if(!pr->right)pr->right=-p;//pr 是 p 前驱，故其右指针指向后继 p
            else if(!p->left)p->left=-pr; //p 是 pr 后继，故其左指针指向前驱 pr
            }
        pr=p;    //先出栈者为前驱
```

```
            p=p->right; //右子树的最左端节点是 p 后继
            }
        }
```

程序 1.23 是算法的 C 语言实现。

程序 1.23 中序线索化函数

```
struct tree *inorder_clew(struct tree *root)
{
    struct tree *p,*&rp=p,*pr=Null,*s[M];
    int top=-1;
    long ia;
    p=root;
    for(;;){
        while(p){top=push(s,p,top);p=p->left;}//在子树内寻找中序序列第一个元素
        if(top==-1)break;
        else{
            top=pop(s,rp,top);//rp 就是 p
            if(pr){//仅在整个树中序序列起点时为空
                if(!pr->right){
                    ia=-(long)p;//以补码为线索标记
                    pr->right=(struct tree *)ia;//pr 是 p 前驱，故其右指针指向后继 p
                    }
                else if(!p->left){
                        ia=-(long)pr;
                        p->left=(struct tree *)ia;//p 是 pr 后继故其左指针指向前驱 pr
                        }
                }
            pr=p;       //递推，当前节点变成后继节点的前趋
            p=p->right; //递推，p 指向当前节点的后继
            }
        }
    return(0);
}
```

1.3.4.2 中序遍历线索化的二叉树

有了穿线树，从任何一个节点搜寻其前驱或者后继都非常方便，无需每次从树根开始。要按中序遍历穿线树非常简单，首先从根节点开始沿着左指针达到最左端点，也就是中序序列的头节点，然后反复找它的后继。如果一个节点的右指针小于零就是线索，则该指针就指向着后继；如果一个节点的右指针大于零，则它一定指向该节点右子树的根，那么，它的后继节点就是它的右子树的最左端点。我们直接给出中序遍历线索化二叉树，搜寻后继节点序列的函数如程序 1.24。

程序 1.24 中序线索化树后继节点遍历函数

```
struct tree *clewprintf(struct tree *root)
{
    struct tree *q,*p;
    long ia;
    q=root;p=q;
    while(q){p=q;q=q->left;}     //中序序列起点
    while(1){
        printf("%d,",p->key);    //访问节点（打印键值）
        if(!p->right)return(0); //无后继则结束遍历
        if((long)p->right<0){    //小于零则是线索
                ia=-(long)p->right;
                p=(struct tree *)ia; //恢复指针
                }
        else {
            p=p->right; //取右子树根节点
            while((long)p->left>0)p=p->left; //子树的中序序列起点
            }
        }
}
```

虽然在遍历操作上线索化带来了简便的搜索方式，但显而易见的是，穿线树的节点插入和删除操作一定比非穿线树要复杂，这是由于随着节点的动态增删，它的前趋和后继也发生变化，必须要同步的改变线索。因为在一棵二叉树中插入的一个新节点总是被插成叶子，所以线索的调整并不是很困难，但删除就比较麻烦，我们不做更多的讨论。

1.3.5 堆

堆的存储结构是数组形式，但是堆的逻辑结构则是二叉树形式。堆有两个基本性质表征：①堆是一棵完全二叉树，所以它可以简单的用顺序存储结构的数组实现；②堆中存储的数据局部有序，即，节点的关键码与其左右孩子的关键码有确定的顺序。堆有两种定义形式：

- 最大值堆（max-heap）

最大值堆的性质是任意一个节点关键码都大于或等于其任意一个孩子的关键码，且定义是递归的。即根节点关键码大于或等于其左右孩子关键码的值，而其左右孩子关键码又依次的大于或等于其各自孩子关键码的值，所以根节点关键码具有所有节点中最大的关键码。

- 最小值堆（min-heap）

最小值堆的性质是任意一个节点关键码都小于或等于任意一个孩子的关键码，且定义是递归的。即根节点的关键码小于或等于其左右孩子关键码的值，而其左右孩子关键码又依次的小于或等于其各自孩子关键码的值，所以根节点关键码具有所有节点中最小的关键码。

注意，无论是最大值堆还是最小值堆，有序只是限定在父亲与孩子之间的，任何一个节点与其兄弟之间的关键码值大小都没有必然的联系。基于排序上的原因，我们只讨论最大值堆的问题。

1.3.5.1 建堆过程

我们假定一棵完全二叉树已经存储在数组内，如图 1.60 给出了建堆的示例，图 1.60（a）的方法是从底层最左边叶子节点开始，比较每一个节点与其父节点的关键码值，若子节点值大，则父子位置交换，逐层向上比较，直至根节点。然后再从这个叶子节点的兄弟开始，继续该过程。具体到图 1.60（a）的交换过程是（4-2），（4-1），（2-1），（5-2），（5-4），（6-3），（6-5），（7-5），（7-6），需要 9 次交换。

另一种方法是，如果我们假设结点 R 的左右子树已经符合堆结构，则只需要比较 R 与其左孩子、右孩子的关键码值，可能有：①R 大于等于它们的孩子，则该树符合堆结构；②R 小于其一个或者 2 个孩子的关键码值，则选择较大一个孩子与 R 的位置交换，形成一个堆。因为交换后的节点 R 被下拉了一层，如果仍然小于其一个或者俩个的新子节点关键码值，则继续节点 R 的下拉过程，直至到达某一层使 R 大于其子节点，或者 R 成为叶子。因为叶子是不会被下拉的，所以，建堆过程应该从倒数第二层最右边一棵子树的根开始，如图 1.60（b）所示，其建堆交换过程是（7-3），（5-2），（7-1），（6-1），需要 4 次交换。显然，不同的建堆方法构造的堆是不同的，所花费的交换次数也不相同。图 1.60（b）建堆起点位置是逻辑结构编号第$\left\lfloor \frac{n}{2} \right\rfloor$个节点处，见图 1.61 所示，它的存储位置是数组下标的$\left\lfloor \frac{n}{2} \right\rfloor - 1$处，因为数组位置从 0 至 n-1。

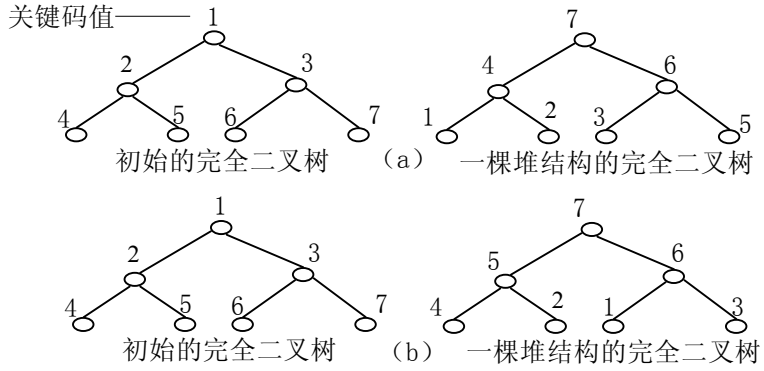


图 1.60 对一棵完全二叉树建堆

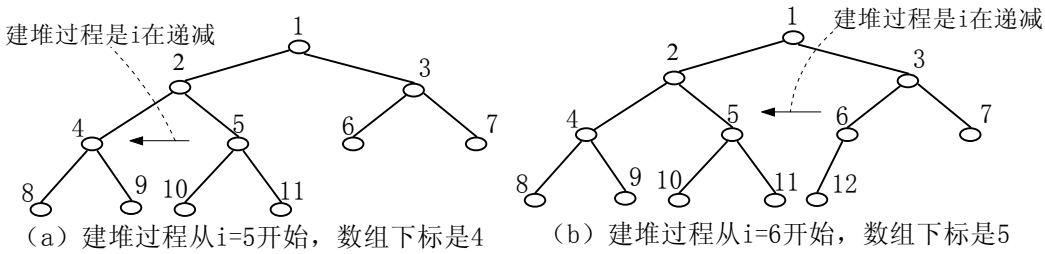


图 1.61 采用图 1.60（b）建堆方法的起点示意

- 堆函数

建堆函数由如下函数组成：①实现图 1.60（b）建堆算法的 siftdown(int)函数；②实现建堆过程的 buildheap()函数；③辅助函数 isleaf(int)在 i 指向叶子节点的时候返回的布尔值为真；④辅助函数 leftchild(int)返回根节点左孩子的位置；⑤交换堆数组 heap[i]和 heap[j]位置的值是 swap(heap[int i],heap[int j])；⑥assert(int)是系统函数，如果输入参数为 0，就调用 abort()函数中的程序运行（abort()是立即中止程序运行）。assert(int)头文件是 assert.h。

堆元素定义如下：

```
struct heapnode{
    int key;
    }array[n];
```

程序 1.25 建堆 C 函数

```
void siftdown(struct node *heap,int n,int pos) //heap 指向堆数组 array[0]
{
    int j;
    assert((pos>=0)&&(pos<n));//系统函数检查 pos 在边界之内，pos 是建堆节点
    while(!isleaf(pos,n)){
      j=leftchild(pos,n); //取得其左孩子坐标
      if((j<(n-1))&&(heap[j].key<heap[j+1].key))j++;//判别哪个孩子关键码大
      if(heap[pos].key>=heap[j].key)return; //若 R 节点大于其左右孩子，符合堆结构
```

```
        swap(&heap[pos],&heap[j]);//否则父与大关键码值的孩子交换位置，R 被下拉一层
        pos=j; //pos 仍指向当前建堆节点 R
        }
}
void buildheap(struct node *heap,int n) //heap 指向 heaparray[0]，n 是堆数组长度
{
    int i;
    for(i=n/2-1;i>=0;i--)siftdown(heap,n,i);
}
bool isleaf(int pos,int n)  //pos 指向堆数组当前位置，n 是堆数组长度
{
    return((pos>=n/2)&&(pos<n));
}
int leftchild(int pos,int n)
{
    assert(pos<n/2);
    return (2*pos+1);
}
void swap(struct node *a,struct node *b)
{
    struct node i;
    i=*a;*a=*b;*b=i;
}
```

对图 1.61（b）建堆所得如图 1.62 所示。

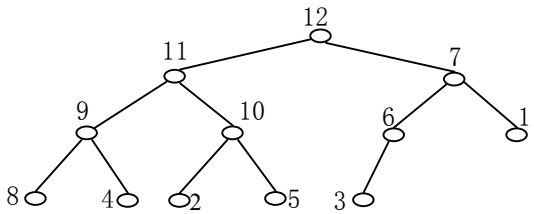


图 1.62 对图 1.61(b)建堆所得的结果

- 建堆效率

在函数 siftdown()调用中考虑一个元素在堆中向下移动的距离，设该堆深度为 d，则堆中的深度为 d-1 的节点约有一半，它们不会向下移动，深度为 d-2 的节点约有四分之一，它们最多可以向下移动一层，建堆的过程中，每在完全二叉树中向上一层，节点数目是前

一层的一半，而已建好的堆的高度增加 1，所以，元素在建堆过程中移动的最大距离总数是：

$$\sum_{i=1}^{\log_2 n}(i-1)\frac{n}{2^i}=O(n)$$

1.3.5.2 在堆中插入节点

在一个堆中插入一个节点可以分解成两步：①新节点是插入到一棵顺序存储的完全二叉树的叶子；②在堆中寻找新节点的正确位置。比如，在图 1.62 的堆中插入一个节点，首先是形如图 1.63 虚线所示的完全二叉树。然后，搜索新节点在堆中的位置（就不一定是叶子了）。显然，根据堆的规则，我们只需比较孩子与其父节点的值大小。如果孩子的值大于父节点，交换，并且继续搜索。否则，退出循环，将新节点插入。设开始的叶子位置是 n-1，则其父节点位置就是$\left\lfloor\frac{i}{2}\right\rfloor-1$。于是，我们从叶子位置开始沿着父节点指向一直往根节点搜索，或者插入到根，或者插入到搜索路径上的某一点。程序 1.26 是对插入函数。

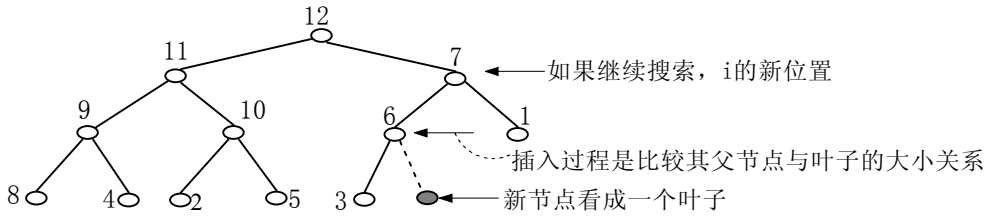


图 1.63 在一个堆中插入节点

程序 1.26 堆插入函数

```
int insertMAXheap(struct node array[],struct node x,int *n)
{
    int i,j;
    if(*n==M)return(-1);//空间不足
    (*n)++;
    i=*n;
    j=i/2-1;      //指向新叶子的父节点
    while((i>=0)&&(x.key>array[j].key)){ //如果孩子的值大于父节点
     array[i-1]=array[j]; //父节点下沉
     i/=2;
     j=i/2-1;     //指向上一层父节点
     }
    array[i-1]=x; //插入
    return(0);
}
```

堆中节点的删除就是删除根节点的过程，我们将在堆排序中讨论。

1.3.6 哈夫曼树

哈夫曼树是二叉树的应用例子，它在通讯编码中寻求具有总编码长度最短的二叉树结构，我们先看有关的概念。

1.3.6.1 最佳检索树

同一元素集合构造出不同深度的二叉树代表着检索效率的不同，如图1.64所示的情况。

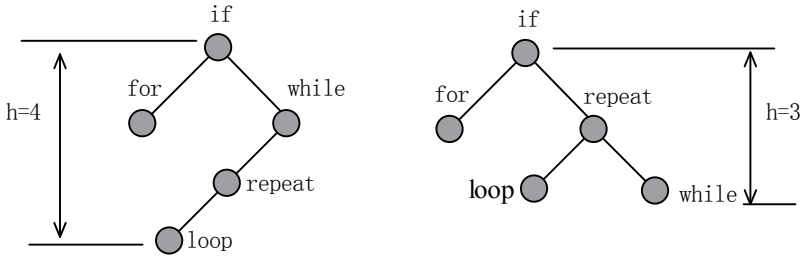


图 1.64 节点所处的深度不同，其检索效率不同

表 1.6 深度不同的二叉树检索效率比较

层次	图1.64左边树该层节点数	该层节点所用检索次数	图1.64右边树该层节点数	该层节点所用检索次数
根	1	1×1	1	1×1
2	2	2×2	2	2×2
3	1	1×3	2	2×3
4	1	1×4		
总检索次数		12		11
平均次数		12/5		11/5

设检索概率相等，显然检索Loop节点在图1.64左边的树结构要比较4次，而图右边的树结构只用比较3次，因而节点所在的层次反映了它的检索效率。在等概率条件下，可以认为图1.64右边的树结构优于左边的树结构。从定量分析看，深度不同造成了两棵树的平均检索次数不同，见表1.6所示。我们知道，在同样节点数构造的二叉树中完全二叉树具有最小深度，因此它的检索效率最佳，实际上最佳二叉排序树的特点是只有最下面两层树节点的度数可以小于2，其它节点度数必须等于2，图1.65给出了一个例子。

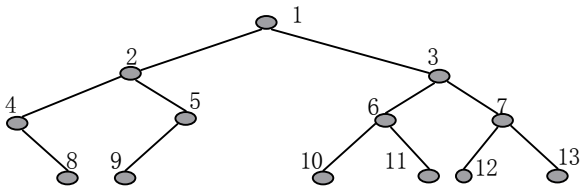


图 1.65 最佳二叉排序树

现在引入路径的概念：从一个节点到另一个节点之间的分枝数称为两节点间的路径长度，从树根到每一节点的路径长度之和，是树的路径长度。图1.64左边树的路径长度是：

$$PL=0+1+1+2+3=7$$

图1.64右边树的路径长度是：

PL＝0＋1＋1＋2＋2＝6

显然，在节点数相同条件下，只有最佳二叉排序树具有最短路径。设第 i 层上的节点路径长度为 L_i，则：

$L_i=i-1$　　i=1，2，...，h；

而第 i 层上路径总长为

PL_i = L_i×i 层上节点总数 ≤$(i-1)\cdot 2^{i-1}$

所以，在任何二叉树成立：

路径为零的节点至多有 1 个

路径为 1 的节点至多有 2 个

路径为 2 的节点至多有 4 个

••••

路径为 k 的节点至多有 2^k 个

在节点总数 n 一定条件下，最佳二叉排序树每层的 PL_i =$(i-1)\cdot 2^{i-1}$，树的路径长度是下面展开级数的前 n 项和：

0, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4...

i=1　i=2　i=3　i=4　i=5

$$PL=\sum_{j=1}^{n}\lfloor \log_2 j \rfloor$$

在非最佳二叉排序树情况下，级数前 n 项中某些项的权值变大，如：

0，1，1，2，2，3，3，4，5，5，5，6，…

由于 n 一定，则其前 n 项和显然比最佳二叉排序树变大，即：

$$PL>\sum_{j=1}^{n}\lfloor \log_2 j \rfloor$$

现在的问题是，已经证明了在检索概率相等条件下，最佳二叉排序树具有最短路径。那么，在检索每个节点的概率不等情况下，什么样的二叉树结构具有最佳检索效率？我们又如何构造这种树？这就是讨论哈夫曼树要达到的目的。

1.3.6.2 哈夫曼树结构与算法

- **概念**

如果一棵树的某些节点被访问的次数不同于另外一些节点，那么平均检索时间不仅与节点所在的层次有关，且与它的访问概率有关，哈夫曼树是在加权概率下的平均检索时间，或路径长度为最小的树结构。在考虑加权概率情况下，定义二叉树的加权路径长度为：

$$WPL=\sum_{i=1}^{n}W_i\times L_i$$

这里，W_i是叶子节点 i 的加权，L_i是从根到叶子节点 i 的路径长度，n 是二叉树中叶子节点个数，此树也称为空心二叉树。

定义：给一棵树的叶子加权，设有$\{W_1,W_2,....,W_n\}$一组实数与二叉树的叶子相结合做加权概率，则此树为空心二叉树或称为叶子二叉树，见图 1.65。使上式空心二叉树路径 WPL 为最小的二叉树，称为加权概率下最佳二叉树或哈夫曼树。

- **算法**

设 WG＝$\{W_1,W_2,....,W_n\}$是一组实数集合，则哈夫曼树的生成是：

1・以此 n 个数为权，赋给 n 个节点构成 n 棵二叉树的根 T_i，其左右子树为空，形成森林 F＝$\{T_1，T_2，...，T_n\}$。

2・将 F 中的树根按其权值由小到大从左至右排序。

3・从 F 中取出第 1、第 2 棵树归并成新二叉树，新树根权值是两棵左右子树根的权值之和，重新放回到森林 F 中。

4・重复步骤 2、3，直到 F 中只剩一棵二叉树为止，即为哈夫曼树。

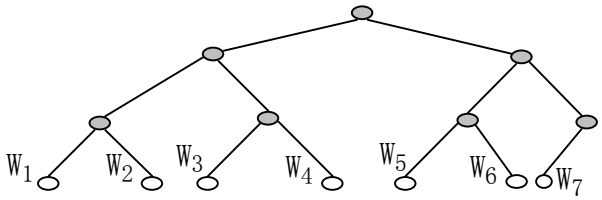


图 1.65 空心二叉树

例 1.13．有一组权值是{7，5，2，4}，构造的哈夫曼树是图 1.66 所示。直观上看，权重的节点因靠近树根而路径短，即经常访问的节点所需检索次数少，使全部节点平均检索时间减少，这是哈夫曼树意义所在（哈夫曼树不是二叉排序树）。

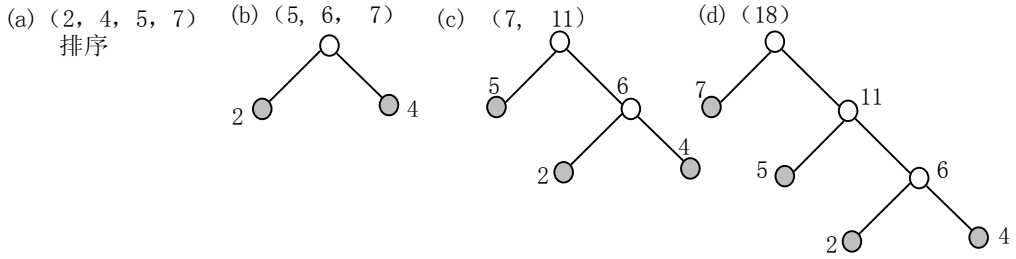
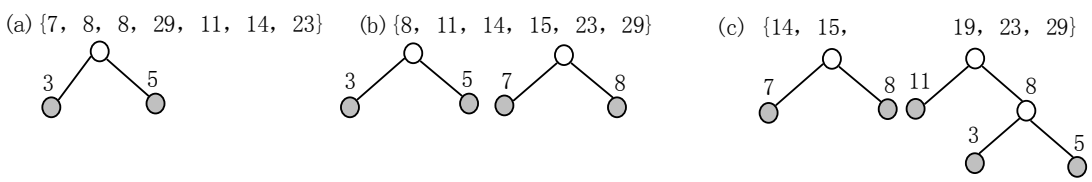


图 1.66 构造哈夫曼树

例 1.14．有一组权值是{5，29，7，8，14，23，3，11}，构造的哈夫曼树是图 1.67 所示。



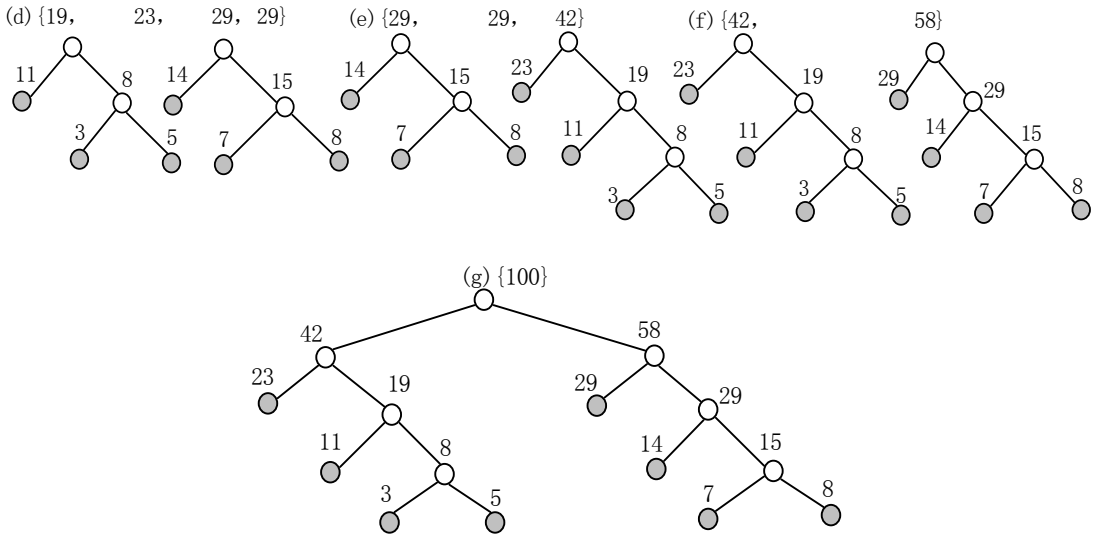


图 1.67 哈夫曼树

哈夫曼树的生成过程是典型的贪心算法。

1.3.6.3 哈夫曼树应用

- **哈夫曼树编码**

哈夫曼树的应用主要在通讯编码上，它在一定字符集、不同的字母出现频率条件下寻求一种总编码长度最短的编码方式。表 1.7 是英文 26 个字母在文献中出现的相对频率表。

表 1.7 英文字母相对频率表

字母	频率	字母	频率	字母	频率
a	77	j	4	s	67
b	17	k	7	t	85
c	32	l	42	u	37
d	42	m	24	v	12
e	120	n	67	w	22
f	24	0	67	x	4
g	17	p	20	y	22
h	50	q	5	z	2
i	76	r	59		

如果我们把它们用代码在计算机中表示，比如 7 位 ASCII 码，字母 A 到 Z 的 ASCII 码是 40H～5FH，是等长的编码方式。既，假定在计算机处理中所有字母它们出现的频率相同，于是，总体的编码长度最短。但实际上字母在文档中出现概率在统计意义上是不相等的，让出现频率多的字母与出现频率少的字母占用同长度的编码位数，显然不合理，比如，在通讯中发报的总体时间就长。于是，我们自然想到常用的字母尽量的用短位数编码，不常用的可以是长位数编码表示，即哈夫曼编码。

例 1.15 哈夫曼树编码。已知如下 8 个字母出现频率，求哈夫曼树编码树。

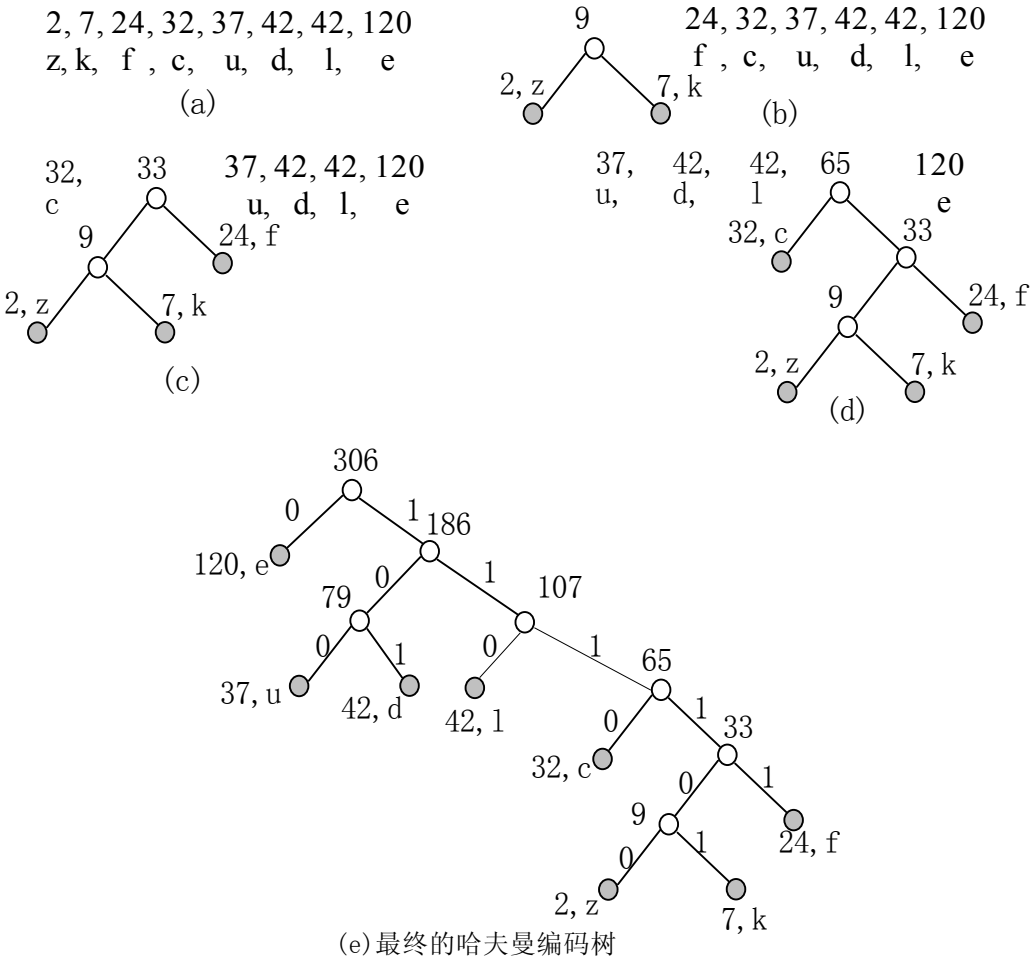
字母	c	d	e	f	k	L	u	Z
频率	32	42	120	24	7	42	37	2

解：

求得的哈夫曼树如图 1.68 所示。将得到的哈夫曼树所有左分枝用 0 标记，右分枝用 1 标记，从根走到任何一片叶子所经历的 0、1 序列，就是所求的字母编码。比如单词 deed，

其哈夫曼编码串是：10100101。

对信息的反编码是从根节点开始，对接收到的二进制代码串从左至右逐位判别，直至叶子，确定一个字母，然后再从根节点开始搜寻。比如数字串 1011001110111101 的反编码是 101，100，1110，111101，既 duck。



(e)最终的哈夫曼编码树

图 1.68 哈夫曼树编码树

实际应用上哈夫曼编码的效率取决于实际文本的信息情况，比如，虽然英文文献中 e 出现的频率最高，但作为单词的第一个字母 t 出现的频率最高，所以，哈夫曼编码在实际上并不是最优的。另外，当编码表中出现的字母频率相差很大时，哈夫曼编码效率高。

- **哈夫曼树在归并排序上的应用**

如下 4 个已排序的数列要归成一个序列，时间复杂度以元素移动次数计算，我们如何决定数列两两归并的顺序才能得到最小的时间开销？

A={1，2}

B={3，4，5}

C={6，7，8，9}

D={10，11，12，13，14}

因为归并两个序列操作的结果是合并为一个新的序列，长度是两子序列长度之和，所以

元素所需的移动次数就是新序列的长度，图 1.69 的（a）、（b）、（c）、（d）四种情况给出了两两归并序列的几种不同顺序下所需移动次数的比较，（a）的归并顺序所需移动次数最少。

那么，我们如何确定外排序中归并不同长度子序列的顺序以求得总的花费时间最少？如果把子序列看成是一个外部节点，序列长度（元素个数）是节点的权，而每次归并两个序列就是两个节点权的合并问题，显然，上述求解最佳归并顺序问题就转化为求一组加权节点其总的路径长度为最短的问题。

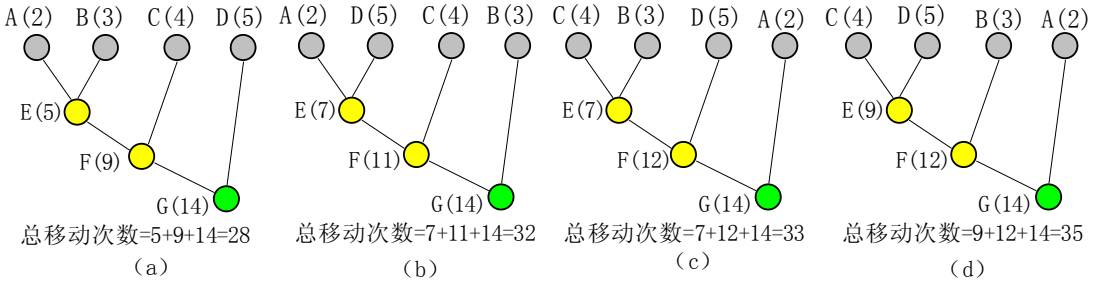


图 1.69 两两归并序列的移动次数比较

设 n 个待归并子序列的长度为 W_i，i=1，2，…，n，每个子序列内元素有序，以序列 i 为叶子节点 i，W_i 为该叶子的权，L_i 是从根到叶子节点 i 的路径长度，用加权节点构造出的哈夫曼树就对应着一个最佳归并顺序，其总的路径长度 $WPL=\sum_{i=1}^{n} W_i \times L_i$ 就是总的移动次数。实际上，图 1.69（a）就是一棵哈夫曼树，归并顺序如图 1.70 所示。

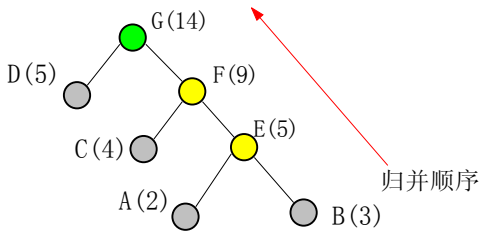


图 1.70 对图 1.69 序列构造一棵哈夫曼树所得的最佳归并顺序

1.3.6.4 哈夫曼树程序设计

哈夫曼树的程序设计需要定义出两个结构体，即字母/频率对节点（权节点）和树节点：

```
struct LettFreq{                    struct treenode{
        char lett;                          strucr LettFreq *var;
        int freq;                           struct treenode *left,*right;
        }                                   }
```

哈夫曼树节点 treenode 的数据域是一个指向字母/频率对（权节点）的指针。初始化的时候我们定义一个指针数组 w[]：

```
struct treenode *w[size];//size 是最大叶子节点数目
```

森林 w[i]的数据域（指针）指向一组字母/频率对，比如图 1.71 所示 5 个字母/频率对：

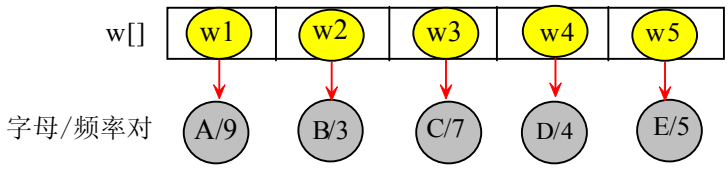


图 1.71 森林与一组字母/频率对

对森林构建最小堆结果如图 1.72 所示，因为指针指向字母/频率对，对 w[i]建堆就是按字母权重建堆。每次取出森林的第一棵树之后，都对森林进行一次最小堆构建。因此，取出的两棵树木就是权最轻的两棵树木，合并之后放回森林如图 1.73 所示。

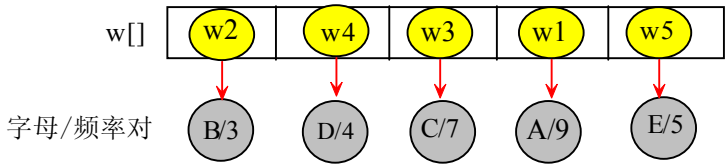


图 1.72 对森林建最小堆

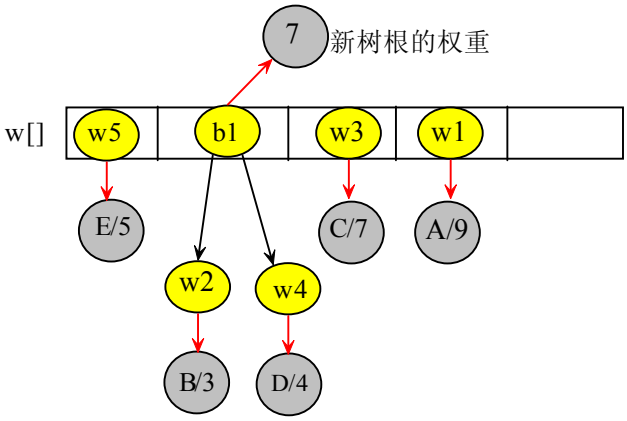


图 1.73 新树根与权重节点示意（合并两个最小权重树之后的最小堆）

合并过程直至森林中只剩余一棵树木为止，即得到所求的哈夫曼树。见程序 1.27 所示。

程序 1.27 Huffman 树函数

```
struct treenode *buildtree(struct treenode **&rp,int &numinlist)
{
    struct treenode *temp1,*temp2,*temp3;
    struct lettfreq *tempnode;
    buildMINheap(rp,numinlist);//在堆数组[0～numinlist-1]建立最小堆
    while(numinlist>1){
        temp1=remove(rp,numinlist);//取表头第 1 棵树并建最小堆，注意 numinlis 已经减一
        temp2=remove(rp,numinlist);
        tempnode=lettfreq((temp1->var->freq)+(temp2->var->freq));//新树根的权重
        temp3=huffman(tempnode,temp1,temp2);//构造 huffman 树
        insert(rp,temp3,numinlist);//把新树放回到森林尾部，numinlis 增一
        buildMINheap(rp,numinlist);//重建最小堆
```

```
        }
    return(rp[0]);//如果森林只有一棵树则 return the tree
}
```

在构造哈夫曼树中,对森林排序使用了堆函数,程序 1.27 首先建立一个最小堆,取 w[i]按字母/频率对的最小权轻者。然后，每次取出森林顶部的一棵树 w[0]，并将森林剩余的树木重新构建一个最小堆。

函数 remove（）删除森林顶部树木，并返回指向该树根的指针，同时森林长度减一。于是，程序 1.27 中的 temp1 和 temp2 就是哈夫曼树的左右子树根。函数 lettfreq()申请一个字母/频率对节点，并将传递过来的实参（左右子树根权重之和）赋给节点数据域的 freq。接下来的语句调用 huffman 函数，它申请一个树节点（treenode），左指针指向实参 temp1，右指针指向实参 temp2，而数据域的指针 var 指向实参 tempnode。Huffman()返回指向新树根的指针给 temp3。最后，函数 insert()把新树放回到森林尾部，numinlis 增一，并将当前森林里的树木构建成一个最小堆。各函数方法参考如下：

```
struct treenode *remove(struct treenode **&rp,int &numinlist)
{//建立最小堆时先把堆顶 rp[0]删除（return）然后再建堆
    struct treenode *item;
    item=rp[0];//store removed item
    swap(rp[0],rp[numinlist-1]);//将当前堆数组的最后一个节点与堆顶节点交换
    numinlist--;              //森林里的树木个数减一
    if(numinlist>0)buildMINheap(rp,numinlist);//最小堆
    return(item);                  //返回被删除的最小权值的树木根
}
void insert(struct treenode **&rp,struct treenode *item,int &numinlist)
{//在当前 numinlist 位置上插入一个元素
    assert(numinlist<msize-1);   //msize 是数组最大长度
    rp[numinlist]=item;           //新树根放在堆数组尾部
    numinlist++;              //森林数目增加一个
}
struct lettfreq *lettfreq(int f)
{
    struct lettfreq *p;
    p=(struct lettfreq *)malloc(sizeof(struct lettfreq)); //申请一字母频率对节点
    p->freq=f;            //给权值赋值
```

```
    return(p);
}
struct treenode *huffman(struct lettfreq *tempnode,struct treenode *temp1,struct
treenode *temp2)
{
    struct treenode *p;
    p=(struct treenode *)malloc(sizeof(struct treenode)); //申请一个树节点
    p->left=temp1;       //左指针域指向一个子树根
    p->right=temp2; //右指针域指向一个子树根
    p->var=tempnode; //数据域指针指向一个字母/频率对节点
    return(p);
}
```

初始化过程如下：

```
void input(struct treenode **&rp,int n)
{
    int i;
    //以下申请叶子节点
    for(i=0;i<n;i++){
        rp[i]=(struct treenode *)malloc(sizeof(struct treenode));
        rp[i]->left=0;rp[i]->right=0;
        }
    //以下给叶子的指针域赋值
    for(i=0;i<n;i++){
        rp[i]->var=(struct lettfreq *)malloc(sizeof(struct lettfreq ));
        cout<<"请输入第"<<i+1<<"个叶子的字符和权重"<<endl;
        scanf("%c %d",&(rp[i]->var->lett),&(rp[i]->var->freq));
        fflush(stdin);
        }
}
```

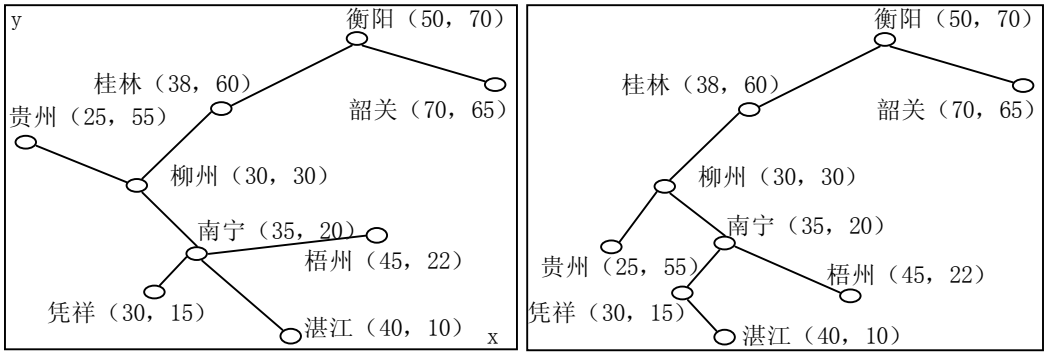
1.3.7 空间数据结构----二叉树深入学习导读

有关二叉树的内容讨论到此，都是根据一个一维的整数关键码值检索一个节点，我们可以理解成它只能检索沿一个数轴分布的数据集合中各个数据记录，比如，商场中某类商品

随时间分布的销售情况，流程工业过程控制实时数据库记录的工艺流程数据，学生数据库随学号分布的学生记录信息等。

实际上，很多场合需要在一个平面检索一个记录信息，比如地理信息系统需要在一个地图平面上根据（x，y）坐标检索一个城市记录信息，或者数据分析中用多维关键码关联特性查询数据库中的纪录等，图 1.74 给出了一个平面分布的城市记录示例，如何根据（x,y）坐标索引一个城市，这是空间数据结构应用问题，我们现在讨论二叉排序树的空间扩展 k-d 树，对其它方法更有兴趣的同学可以参考有关书籍。

输入序列：{衡阳，韶关，桂林，柳州，贵州，南宁，凭祥，湛江，梧州}



（a）平面分布的城市　　（b）k-d树结构

图 1.74 平面分布的城市与生成的 k-d 树

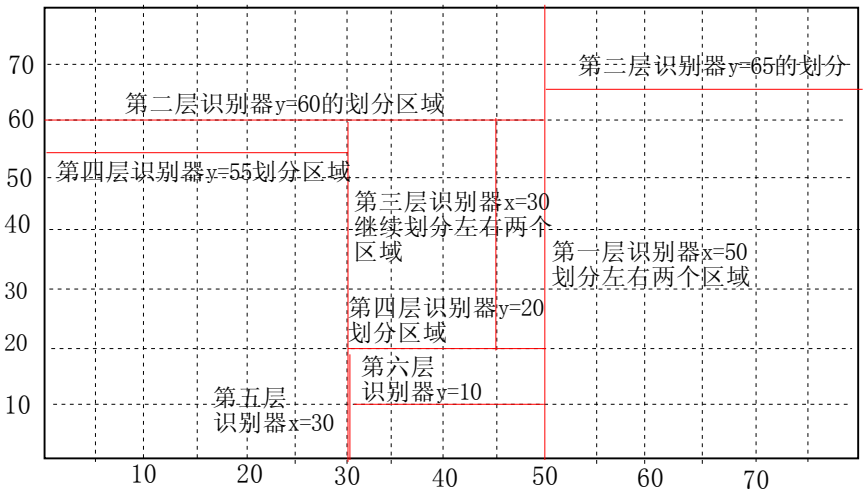


图 1.75 图 1.74（b）的 k-d 树检索空间划分

1.3.7.1k-d 树概念

- **识别器**

k-d 树是二叉树的空间扩展，其不同于二叉树的地方是 k-d 树的每一层用一个识别器（discriminator），从多维关键码中选定某一维关键码值做 k-d 树分支走向的决策，决定 k 维关键码在第 i 层所用维数分量的识别器是：

i MOD k

比如，图 1.74 是（x,y）二维关键码，即 k=2，设 x 坐标定义为关键码 0，y 坐标定义为关键码 1，显然，识别器随层数 i 深入在 0 和 1 之间取值，以树根为第 0 层，则 k-d 树每

一层的分支走向决策所用的关键码值是交互由坐标 x、y 值决定。

注意，识别器只是选择在第 i 层用多维关键码中某个分量决策分支走向，但检索中判别是否找到匹配的关键码节点，仍需要使用多维关键码整体信息，以在图 1.74k-d 树中检索关键码为（35，20）节点过程为例，根据定义已知识别器由树根起交互用 x、y 坐标值做分支走向决策：

（1） 树根的 x 坐标是 50 而输入关键码值是 35，走左分支；

（2） 桂林节点的 y 坐标是 60，而输入关键码值是 20，仍然走左分支；

（3） 柳州节点的 x 坐标是 30，而输入关键码值是 35，走右分支；

（4） 南宁节点的 y 坐标是 20，而输入关键码值是 20，y 值匹配；

现在的问题是，图 1.74 中具有 y=20 的节点只有南宁城市一个，因而可以用单维信息匹配，不过这并不适于一般的处理情况，比如同一图中检索关键码是（30,15），其走过的路径是：衡阳（50,70），桂林（38,60），柳州（30,30），如果只用单维分量匹配则柳州的 x 坐标符合，但显然具有坐标（30,15）的城市是凭祥。

- **符合函数**

为此，我们定义一个符合函数：某一类节点信息检索的多维关键码特征用 k 个分量描述，组成 k 维矢量 X，而节点集合中第 i 个节点的关键码是 $X_i=(x_{i1},x_{i2},\cdots,x_{ik})$，定义该集合信息元符合函数为：

$$f(x_r,x_i)=\sqrt{\sum_{j=1}^{k}(x_{rj}-x_{ij})^2}$$

其中，x_r 是检索关键码，x_i 是节点关键码，当 d=2，则符合函数就是熟知的两点间距离公式：

$$f(x_r,x_i)=\sqrt{(x_{r0}-x_{i0})^2+(x_{r1}-x_{i1})^2}$$

其中，$x_{r0}=x_r,x_{i0}=x_i,x_{r1}=y_r,x_{i1}=y_i$。设两点距离为零表明匹配成功，则在图 1.74 中检索关键码向量为（30,15）的节点过程是：衡阳（50,70），桂林（38,60），柳州（30,30）南宁（35,20），凭祥（30,15）。

同学可能会发现一个问题，已知通过比较识别器选择的关键码分量和当前层次节点对应的特征分量大小来确定搜索路径，如果关键码值小于节点特征值则选择走左分支，如果大于则走右分支搜索路径，那么，图 1.74 中检索关键码（30,15）在搜索到节点柳州的时候，其识别器输出是以 x 坐标为当前层的分支走向判定，而检索关键码的 x 坐标和柳州的 x 坐标相等，如何判定后续搜索方向？我们约定如下，k-d 树节点插入时候，所有小于当前层决策特征信息的节点被放到 k-d 树的左子树，所有大于或等于当前层决策特征信息的节点放到 k-d 树的右子树。

1.3.7.2k-d 树程序设计初步

我们讨论二维 k-d 树的程序设计基本问题。

- **检索操作**

一个二维 k-d 树是二叉树向平面空间的自然扩展,它是二叉排序树,其关键码由坐标(x，y）表达，定义 x 是 0 维信息，y 是 1 维信息，其二叉树分枝规则是在第 i 层用 i mod 2 所确定的 0 或 1 取相对应的维信息做该层节点（x，y）的关键码识别值（根节点所在为 0 层），k-d 树按关键码有序，设节点结构定义如下：

```
struct kdnode{
   int val[2];  //the val[0] value of x- axis; val[1] value of y- axis;
   struct kdnode *left,*right;
   };
```

关键字（x,y）格式为 int key[2]，其中 key[0]是 x、key[1]是 y 坐标，lev 初值为 0，识别器函数 dkey 定义如下：

```
int dkey(int *p, int lev)
{
   if(lev==0)return *(p+0);
   else return *(p+1);
 }
```

符合函数 vector 定义如下：

```
int vector(int *p, int *s)
{
return sqrt((*(p+0)-*(s+0))*(*(p+0)-*(s+0))+(*(p+1)-*(s+1))*(*(p+1)-*(s+1)));
}
```

则二维 k-d 树按（x，y）关键字检索一个节点的过程如下：

程序 1.28

```
struct kdnode *kdsearch(struct kdnode *root, int *key, int lev)
{
   if(root==NULL)return NULL;
   if(vector(&key[0],&(root→val[0]))==0)return root;
   else {
        if(dkey(&key[0],lev)<dkey(&(root→val[0]),lev))
            return(kdsearch(root→left,&key[0],(lev+1)%2));
        else return(kdsearch(root→right,&key[0],(lev+1)%2));
```

```
        }
}
```

例 1.16 k-d 树检索。节点输入序列是 A（40,45），B（15,70），C（70,10），D（69,50），E（55,80），F（80,90），求：

（1） 画出生成的 k-d 树；

（2） 设检索关键字值是（69，50），描述在该 k-d 树上的检索过程；

（3） 画出该 k-d 树每一节点所划分的检索空间。

解：

生成 k-d 树如图 1.76（a）所示，检索路径是 A，C，D（D 点距离空间为 0），检索空间如图 1.76（b）所示。

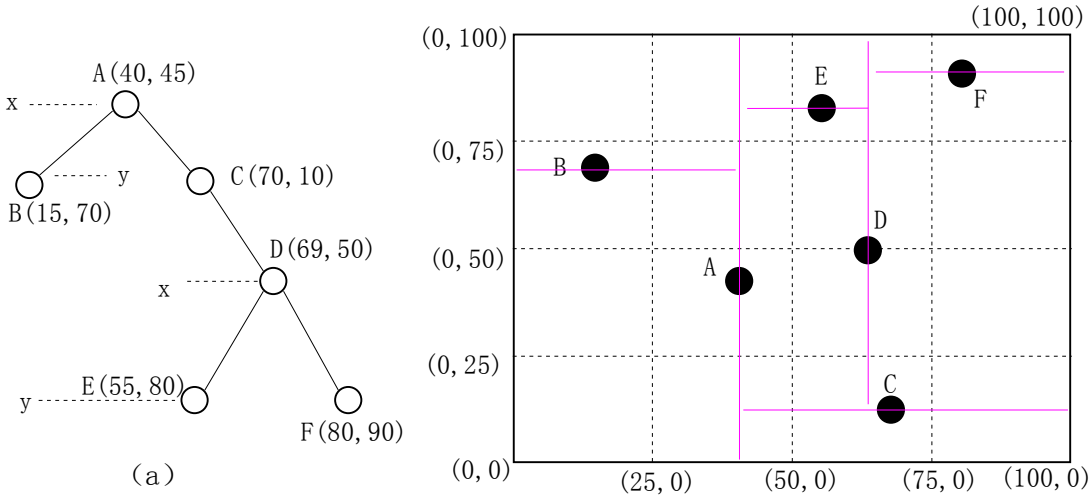


图 1.76 k-d 树举例

- **删除操作**

k-d 树的节点插入与二叉排序树一样，新进来的节点总是被插入到叶子位置，通过一个检索操作找到插入点，修改父指针插入到叶子位置。但是，删除操作和二叉排序树有所不同，如果是叶子当然没有问题，如果度不为零则有些困难，比如在图 1.58 中删除节点桂林，其有一个分支，你不能像二叉树那样简单的修改指针，让桂林的父节点指向其子节点柳州，因为提升柳州节点的层次会让识别器的输出结果改变，即柳州节点在第三层器识别器输出结果是 1，提升到第二层之后器识别器输出结果是 0，于是，简单的提升柳州节点就破坏了 k-d 树特性，比如，南宁应该改在柳州左分支而贵州在其右分支。

删除的思想和二叉树一样，设要删除节点为 P，则 P 位置或者被其右子树中具有 P 节点识别器的最小值节点所取代，或者被其左子树中具有 P 节点识别器的最大值节点所取代。假定 P 在奇数层，识别器是 y，如果要在 P 节点的右子树寻找有最小 y 值的节点，显然它并不一定在最左边，因为识别器输出是 x，y 交互的，如图 1.77 所示。所以，我们首先要一个程序在右子树中找到最小值。

在 k-d 树上查找最小值的参考程序如下：

```
//discrim:discriminator key used for mininum search;
//level:current level (mod k);
//k:number of levels in key
Struct kdnode *Kdfindmin(Struct kdnode * rt,int discrim,int level,int k)
{
Struct kdnode *temp1,*temp2;
if(rt==Null)return(Null);
temp1=Kdfindmin(rt→left,discrim,(level+1)%k,k);
if(discrim!=level){          //Min value could be on either side
    temp2= Kdfindmin(rt→right,discrim,(level+1)%k,k);
    if((temp1==Null)||((temp2!=Null)&&
         (dkey(temp2->val,discrim)< dkey(temp1->val,discrim))))temp1=temp2;
    }//Now,temp1 has the smallest value in children (if any)
if((temp1==Null)||(dkey(rt->val,discrim)<dkey(temp1->val,discrim)))return(rt);
else return(temp1);
}
```

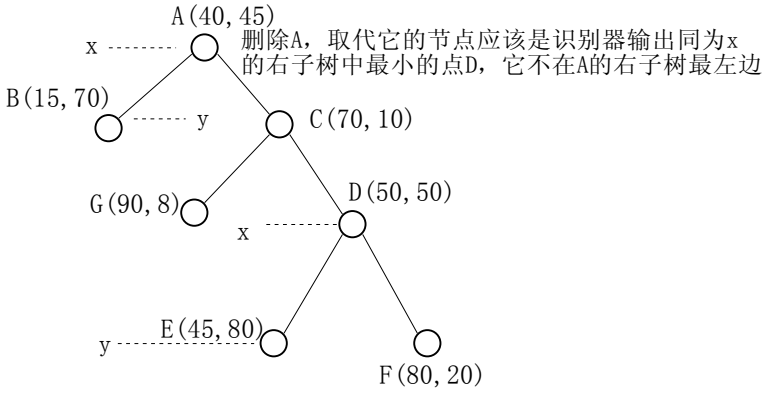


图 1.77 k-d 树删除

有关 k-d 树删除的程序比较复杂，可以在上机实验中讨论。

1.4 非线性数据结构--图

在数据结构二元组 K=(D,R)定义中，D 是有限元素集合，R 是定义在 D 上的有限个关系。通过限制一个关系 r 的前趋和后继元素的个数，我们分别得到了线性表和树形结构。如果不限制 r 的前趋和后继节点个数，所得结果就是图。图形数据结构是现实世界中广泛存在的一种非线性数据结构，本节不讨论图论中的内容，主要是研究图形数据结构的计算机描

述（存储）方式，为此，首先叙述它的一些基本概念和术语。

1.4.1 图的基本概念

如果 V(G)是数据元素的有限集合，E(G)是它的笛卡尔积 V×V：

$$V \times V = \{(v,u) | v,u \in V且v \neq u\}$$

的子集，则称图 G=(V,E)是一个图。其中，V(G)中的元素称为图 G 的顶点，E(G)中的元素称为图 G 的边。例如图 G1 的关系如下：

V(G1)={v1,v2,v3,v4}

E(G1)={(v1,v2),(v1,v3),(v1,v4),(v2,v3),(v2,v4),(v3,v4)}

其数据结构如图 1.78（a）所示。而图 1.78（b）的 G2、（c）G3 关系分别是：

V(G2)={v1,v2,v3,v4,v5,v6,v7}

E(G2)={(v1,v2),(v1,v3),(v2,v4),(v2,v5),(v3,v6),(v3,v7)}

V(G3)={v1,v2,v3}

E(G4)={<v1,v2>,<v2,v1>,<v2,v3>}

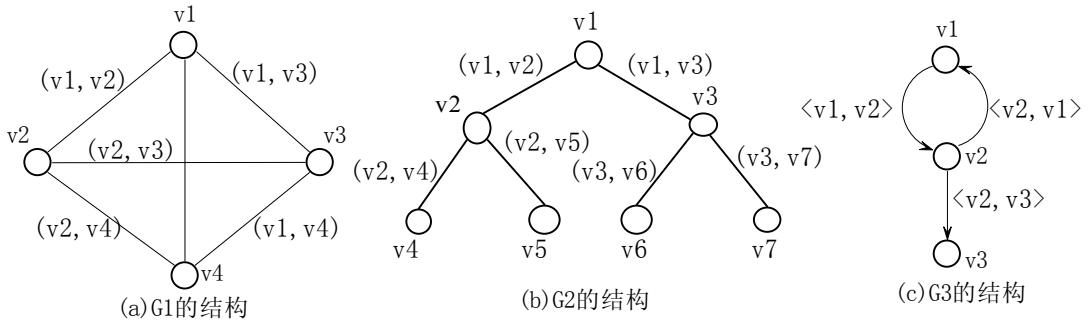


图 1.78 若干图形结构

如果图中一条边的节点偶对是无序的，则称此图是**无向图**，在无向图中，(v1,v2)和(v2,v1)代表同一条边。

如果图中一条边的节点偶对是有序的，则称此图是**有向图**，在有向图中，<v1,v2>表示一条边，v1 是始点，v2 是终点。而<v1,v2>和<v2,v1>代表不同的边。在后面的讨论中，假定我们不考虑节点到其自身的边，即如果(v1,v2)或者<v1,v2>是一条边，则 v1≠v2。且不允许一条边在图中重复出现。

现在讨论**完全有向图**和**完全无向图**。假定一个无向图 G 中，边的集合 E 包含了 V 的笛卡尔积 V×V 所有分项，则称此图是完全无向图。即图 G 中任何两个顶点之间都有一条边相连。显然，图 1.78（a）是 4 个节点的完全无向图。在一个有 n 个顶点的无向图中，因为任何一顶点到其余 n-1 个顶点之间都有 n-1 条边相连，容易知道，第一个顶点 v1 有 n-1 条边与其余 n-1 个顶点相连，第二个顶点 v2 有 n-2 条边和其余 n-2 个顶点相连，因为 v1 已经连接了 v2，它们之间无需连接（因为(v1,v2)和(v2,v1)代表同一条边）。依此类推，最后一

个顶点需要连接其它顶点的边数是零。即：

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

所以，一个具有 n 个顶点的无向图其边数小于等于$\frac{1}{2}n(n-1)$，当边数等于$\frac{1}{2}n(n-1)$时，它是完全无向图。

假定一个有向图 G 中，任何两个顶点之间都有方向相反的两条边相连，则称此图是完全有向图。显然，完全有向图的边数等于$n(n-1)$。

若(v1,v2)∈E，我们说 v1 和 v2 **相邻**，而边(v1,v2)则是与顶点 v1 和 v2 相**关联**的边。若<v1,v2>为有向图的一条边，则称顶点 v1 邻接到顶点 v2，而 v2 也邻接到 v1，而边<v1,v2>是与顶点 v1 和 v2 相关联的。

与树的度的概念类似，一个顶点的**度**就是与该顶点相关联的边的数目。若是一个有向图，则把以顶点 v 为终点的边的数目称之为 v 的**入度**，把以顶点 v 为始点的边的数目称之为 v 的**出度**。

图 1.78（c）v2 的出度为 2，入度为 1，所以 v2 的度为 3。有向图中出度为 0 的顶点称为**叶子**。入度为 0 的顶点称为**根**。

若无向图 G 有 n 个顶点，t 条边，设 d_i 为顶点 v_i 的度数，因为一条边连接两个顶点，则：

$$t=\frac{1}{2}\sum_{i=1}^{n}d_i$$

图 G=（V,E），G'=（V',E'），若 V'∈V，E'∈E，并且 E'中的边所关联的顶点全部在 V'中，则称图 G'是图 G 的子图。图 1.79 是图 1.78（a）的几个子图。

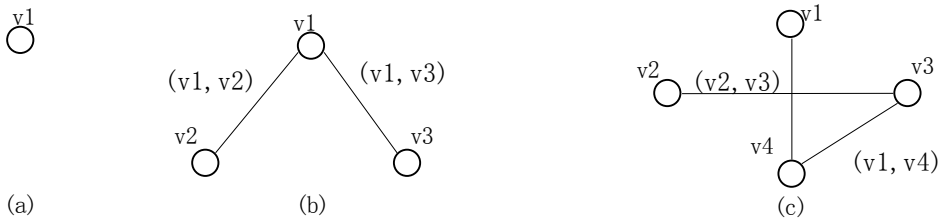


图 1.79 图 G1 的几个子图

图 G=(V,E)中，如果存在顶点序列 v_p，v_{i1}，v_{i2}，…，v_{in}，v_g，使得（v_p,v_{i1}），（v_{i1},v_{i2}），…，（v_{in},v_g）都在 E 中（若是有向图，则使得<v_p,v_{i1}>，<v_{i1},v_{i2}>，…，<v_{in},v_g>都在 E 中），则称从顶点 v_p 到 v_g 之间存在一条**路经**，路径长度是这条路径上的边数。如果一条路径上的顶点除 v_p 和 v_g 可以相同之外，其余顶点均不相同，则称此路径是一条**简单路径**。

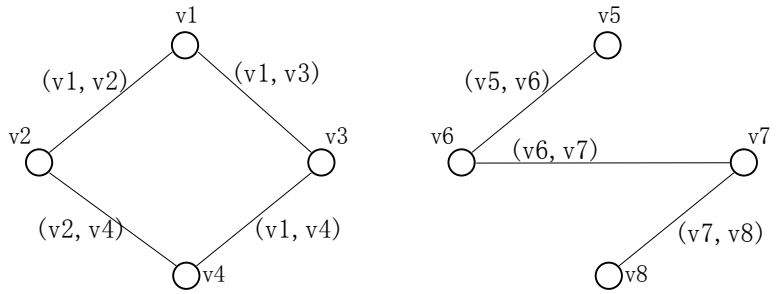
一般把路径（v_1,v_2），（v_2,v_3），（v_3,v_4）简单写成 v_1，v_2，v_3，v_4。图 1.78（a）的 G1 中，v1，v2，v3 和 v1，v3，v4，v1，v3 是两条路经，前者是简单路径，后者不是。图 1.78（c）的 G3 中，v1，v2，v3 是一条简单的有向路径，而 v1，v2，v3，v2 不是路径（没有< v3,v2>

连接边)。

$v_p=v_g$的简单路径称为**环**。一个有向图中，若存在一个顶点 v_0，从它出发有路径可以达到图中任何一个顶点，则称此有向图有根，根为 v_0。

现在讨论有向图和无向图的**连通**概念。对无向图 G=(V,E)来说，如果从顶点 v_i到 v_j有一条路经，我们称 v_j和 v_j是连通的。若 G 中任意两个顶点 v_i和 v_j都是连通的（i≠j），则称无向图 G 是连通的。

图 1.78 的（a）和（b）是连通的。不连通的图如图 1.80 的 G4 所示。一个无向图的连通分支定义为该图的最大连通子图。

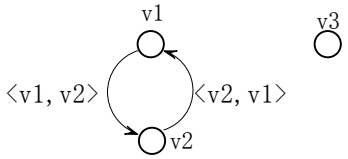


图G4的结构

图 1.80 不连通的无向图 G4 存在两个连通分支

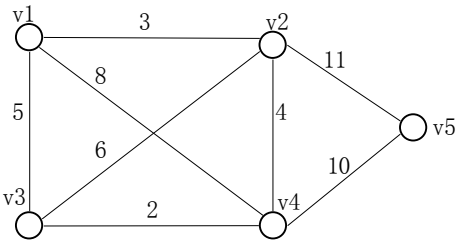
对有向图 G=(V,E)来说，若 G 中任意两个顶点 v_i和 v_j（i≠j）都有一条从 v_i到 v_j的有向路径，且也存在从 v_j和 v_i的有向路径，则称有向图 G 是**强连通**的。图 1.78（c）的 G3 不是强连通的，因为从 v1 到 v3 不存在一条路经。

一个有向图的强连通分支定义为此图的强连通最大子图。图 1.81 给出了图 1.78（c）G3 的两个强连通分支。



G3的强连通分支

图 1.81 G3 的强连通分支



带权的图G5

图 1.82 网络

如果给图的每一条边增加一个数值作为权，我们称为带权的图，带权的连通图称为网络，如图 1.82 所示。

1.4.2 图形结构的物理存储方式

图形结构的物理存储方式有三种，我们分别叙述。

1.4.2.1 相邻矩阵

图 G 的相邻矩阵是表示 G 中顶点 i 和顶点 j 之间相邻关系的矩阵。若顶点 i 和顶点 j

相邻，则矩阵元素 a_{ij}=1，否则为 0。具体地说，设 G 有 n 个顶点，则相邻矩阵是如下定义的 n×n 矩阵。

$$A[i,j]=\begin{cases}1, & \text{若}(v_i,v_j)\text{或}<v_i,v_j>\text{是图G的边}\\ 0, & \text{若}(v_i,v_j)\text{或}<v_i,v_j>\text{不是图G的边}\end{cases}$$

图 1.78 的 G1 和 G3 的相邻矩阵 A1 和 A3 分别表示在下面。容易知道，带权图（网络）的相邻矩阵仅需将矩阵中的 1 代换为权值。图 1.82 所示网络的相邻矩阵 A5 也在下面给出。

$$A1=\begin{bmatrix}0&1&1&1\\1&0&1&1\\1&1&0&1\\1&1&1&0\end{bmatrix},\quad A3=\begin{bmatrix}0&1&0\\1&0&1\\0&0&0\end{bmatrix},\quad A5=\begin{bmatrix}0&3&5&8&0\\3&0&6&4&11\\5&6&0&2&0\\8&4&2&0&10\\0&11&0&10&0\end{bmatrix}$$

设图 G 有 n 个顶点，我们用相邻矩阵存储图型结构的边的关系，用长度为 n 的顺序表存储图的 n 个顶点数据，或者是指向顶点数据的指针。对于有向图，相邻矩阵需要 n^2 个单元，对于无向图，因为相邻矩阵是对称的，只需要存储它的下三角部分。

显然，无向图的相邻矩阵第 i 行元素值之和就是第 i 个顶点的度（和顶点 i 相连的边数）。对于有向图，相邻矩阵第 i 行元素值之和是第 i 个顶点的出度（以顶点 i 为始点的边数），第 i 列元素值之和就是第 i 个顶点的入度（到顶点 i 的边数）。

此外，G 中顶点 i 和顶点 j 之间如果存在一条长度为 m 的路径，则相邻矩阵 A 的 A^m 的第 i 行第 j 列元素为 0。

1.4.2.2 图的邻接表示

在十字链表中我们学习过稀疏矩阵的链式存储方法，图的邻接表示与其类似。对于图 G 中的某一个顶点 v_i，用一个链表来记录与其相邻的所有顶点，也就是所有的边，我们称之为边表。然后用一个顺序表存储顶点 v_i（i=1，2，...，n）的数据以及指向 v_i 的边表的指针。比如，图 1.83 是图 1.82 的 G5 的邻接表示（没有考虑权）。

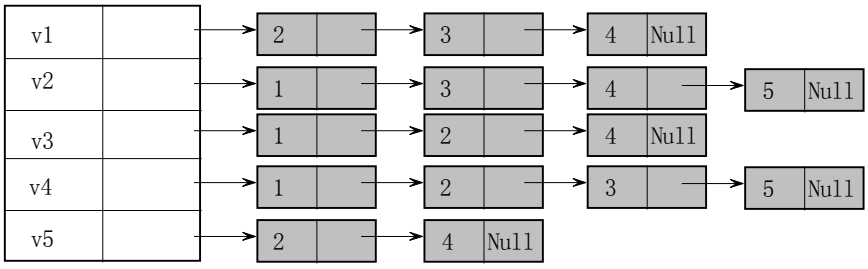


图 1.83 G5 的邻接表示

顶点 v_i 的边表的每个节点对应一个与 v_i 相连的边，节点数据域是与 v_i 相邻顶点的序号，指针域则指向下一个与 v_i 相邻的顶点。用邻接法表示无向图，则每条边在它两个端点的边表中各占一个节点。程序 1.29 给出了邻接法图数据输入的实例。

对于有向图，出边表和入边表分开保存，比如图 1.84 是图 1.78（c）G3 的邻接表示。

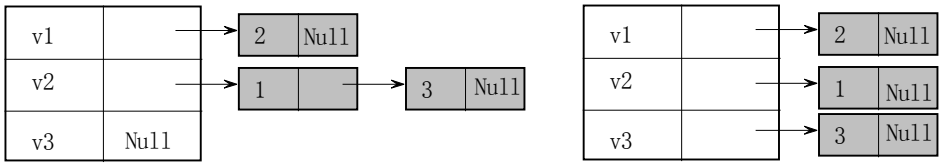


图 1.84 有向图 G3 的邻接表示

数据结构定义如下：

```
struct node{
    bool mark;          //访问标志
    char letter;    //顶点数据域
    struct edge *out;   //指向边表的指针
    };
struct edge{
    bool mark;          //访问标志
    int no;             //顶点编号
    struct edge *link;  //指向边表的后继
    };
```

程序 1.29 图的邻接表示

```
void graphinput(struct node *nodelist,int n)
{
    int i,j,m;
    struct edge *q,*p;
    for(i=0;i<n;i++){//以下输入顶点数据
        printf("输入顶点%d 编号\n",i+1);
        cin>>nodelist[i].letter;
        nodelist[i].mark=false; //设置访问标志初始为 false
        nodelist[i].out=Null;   //边表初始置空
        }
    for(i=0;i<n;i++){//以下设置顶点的边表
        cout<<"输入第"<<i+1<<"个顶点的关联边数"<<endl;
        cin>>m;
        if(m){
            nodelist[i].out=(struct edge *)malloc(sizeof(struct edge));
            cout<<"输入第"<<i+1<<"个顶点的第"<<1<<"条边"<<endl;
```

```
            scanf("%d",&(nodelist[i].out->no));//与第 i 个顶点相邻的图顶点编号
            nodelist[i].out->link=Null;
            nodelist[i].out->mark=false;
            p=nodelist[i].out;
            for(j=1;j<m;j++){
                q=(struct edge *)malloc(sizeof(struct edge));
                cout<<"输入第"<<i+1<<"个顶点的第"<<j+1<<"条边"<<endl;
                scanf("%d",&(q->no));
                q->link=Null;
                q->mark=false;
                p->link=q;
                p=q;
                }
            }
        }
}
```

1.4.2.3 图的多重邻接表示

图的邻接表用数组存储顶点信息，因而每条边会分别出现在其相邻两个顶点的边表中。如果我们把图的所有顶点信息用一个链表来描述（图节点），每个图节点指向一个边表（表节点），表节点存储的不是顶点的序号，而是指向边（或者说弧）另一端相邻顶点的指针，我们称之为图的多重链表表示。我们分别为图和其边表设计了动态的数据结构如下：

```
struct node{
    bool mark;              //访问标志
    char letter;         //顶点数据域
    struct node *nextnode;  //指向图顶点集合中下一个元素的指针
    struct arc *out;     //指向该顶点边表的指针
    };
struct arc{
    bool mark;              //访问标志
    struct node *link;      //指向该弧（边）的另一端顶点的指针
    struct arc *nextarc;// 指向与该顶点连接的其余弧（边）的指针
    };
```

图节点结构由顶点数据域、指向顶点集合中下一个元素的指针、以及指向顶点边表节

点的指针构成。

设顶点 v_i 有 k 条边与顶点 v_{j1}，v_{j2}，…，v_{jk} 相连，顶点 v_i 的边表节点结构由指向顶点 v_{jn}（n=1，2，…，k）的指针、以及指向边表后继节点的指针构成。图 1.85（a）G6 是一个有向图，图 1.85（b）是它的多重链表节点结构。图 1.86 是图 1.85（a）G6 的多重链表表示。

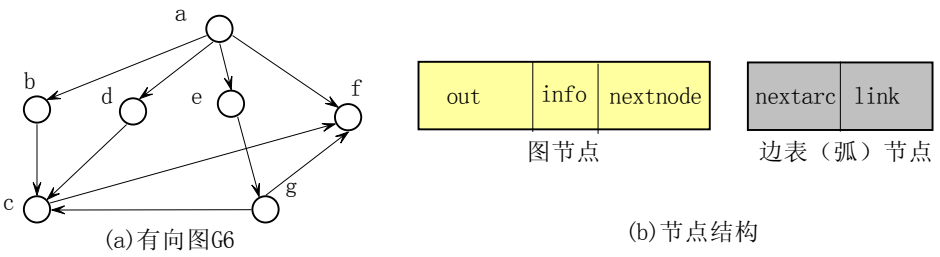


图 1.85 有向图 G6 及其多重链表节点结构

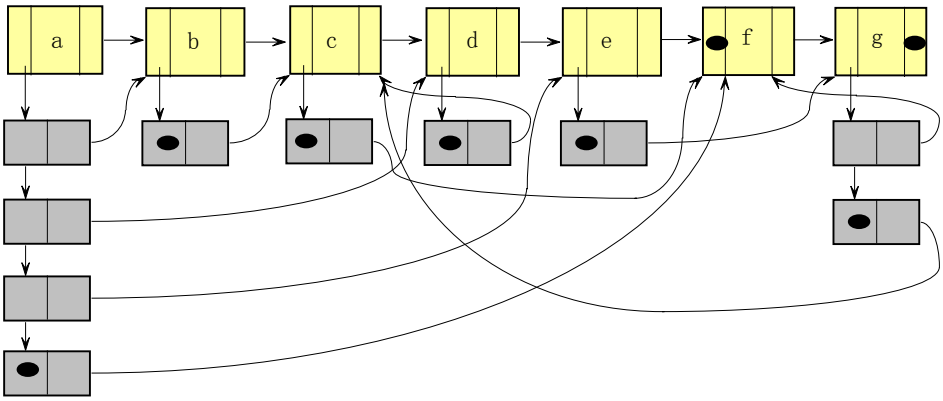


图 1.85 有向图 G6 的多重链表表示

1.4.3 图形结构的遍历

给出一个图 G 和其中任意一个顶点 v_0，由 v_0 开始访问 G 中的每一个顶点一次且仅一次的操作称之为图的遍历。由于子图中可能存在循环回路，所以树的遍历算法不能简单的应用于图。为防止程序进入死循环，我们必须对每一个访问过的顶点做标记，避免重复访问。

图的遍历方式有深度优先和广度优先两种方式，分别对应着栈和队两种数据结构运用。

深度优先(depth-first search:DFS)：访问顶点 v_0，然后选择一个 v_0 邻接到的，且未被访问的顶点 v_i，再从 v_i 出发按深度优先访问其余邻接。当遇见一个所有邻接顶点都被访问过的顶点 U 的时候，回到已访问序列中最后一个有未被访问过的邻接的顶点 W，再从 W 出发按深度优先遍历图。当图中任何一个已被访问过的顶点都没有未被访问的邻接顶点时，遍历结束。

宽度优先(breadth-first search:BFS)：访问顶点 v_0，然后访问 v_0 邻接到的所有未被访问的顶点 v_1，v_2，v_3，…，v_t。然后在依次访问 v_1，v_2，v_3，…，v_t。所邻接的、未被访问的顶点，继续这一过程，直至访问全部顶点。宽度遍历非常类似于树的层次遍历，使用

的是队列这种数据结构。

按深度优先遍历图 1.85（a）所得顶点序列是：a，b，c，f，d，e，g。

按宽度优先遍历图 1.85（a）所得顶点序列是：a，b，d，e，f，c，g。

若图是连通的无向图或者是强连通的有向图，则从任何一个顶点出发都能遍历图的所有的顶点，所得结果如图 1.87 的（a）和（b），结构像一棵树。若图是有根的有向图，则从根出发，可以系统的遍历所有顶点。图的所有顶点、加上遍历过程中访问的边所构成的子图，我们称之为图的生成树。比如图 1.87 的（a）和（b）。显然，根据访问起点的不同，生成树不同。

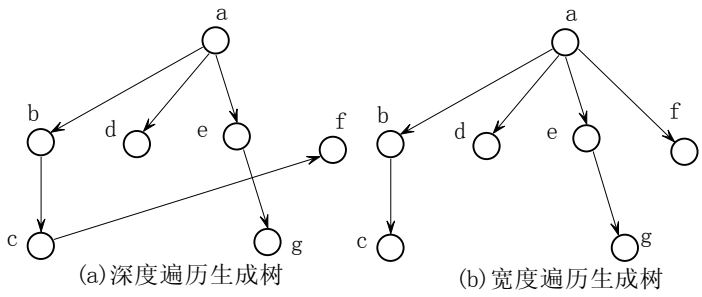


图 1.87 有向图 G6 的遍历

对于非连通无向图和非强连通有向图，从任意顶点出发不一定能遍历图的所有顶点。而只能得到以该顶点为根的连通分支的生成树。因此，继续图的遍历过程需要从一个没有访问过的顶点开始，所得的也是那个顶点的连通分支的生成树，于是，图的遍历结果就是一个生成树的森林。

程序 1.30 是采用邻接存储方式的图深度优先遍历 c 语言函数示例。程序使用了一个辅助数组 next[]存储每一个顶点的下一个要检查的边。每达到一条未检查过的边，程序沿着边节点指示的顶点序号，按深度优先搜寻边下降路径上的顶点，并将沿途顶点的序号入栈。此时有两种情况：

① next[]空，深度方向上没有后继，则弹出栈顶，将搜索路径上最近入栈的顶点序号取出，检查它是否有尚未搜索的边，有则沿该边进行深度优先搜寻，否则更换边。

② 该顶点已经被标记，也需要更换新的边。

程序 1.30 图的深度优先遍历（邻接表存储结构）

```
void DFS(struct node *nodelist,int n)
{
    int i,k,top=-1,p;
    bool notfinished;
    int stack[size];
    struct edge *next[size]; //next[i]是每个顶点边表要检查的下一条边
```

```
    for(i=0;i<n;i++)next[i]=nodelist[i].out; //next[i]初始化指向各顶点边表第一条边
    for(k=0;k<n;k++){//搜索顶点集合中未被访问的顶点 k，并从此出发遍历该顶点生成子树
        if(nodelist[k].mark==false){//此顶点未访问
            i=k;
            printf("%c,",nodelist[i].letter);//访问此顶点
            nodelist[i].mark=true;  //该顶点已访问标记
            notfinished=true;
            while(notfinished){          //从此出发遍历该顶点的生成子树
                while(next[i]==Null){    //如果边表空则可以回朔顶点
                    if(top==-1){//若栈空则该顶点生成子树遍历结束跳出循环
                        notfinished=false;
                        printf("\n");
                        break;
                        }
                    top=pop(stack,&i,top); //否则深度搜索路径上的一个顶点出栈
                    }
                if(notfinished){//检查与第 i 个顶点关联的一条尚未搜索的边
                    p=next[i]->no;  //指向边的另一端邻接顶点 p
                    p-=1;           //因为顶点编号从 1 开始而数组下标从 0 开始
                    if(nodelist[p].mark==false){//顺此顶点的深度方向遍历
                        top=push(stack,i,top);  //当前顶点的前趋进栈
                        next[i]->mark=true;     //前趋边访问过标记
                        printf("%c,",nodelist[p].letter);//访问此顶点
                        nodelist[p].mark=true;  //访问过标记
                        next[i]=next[i]->link;  //指向顶点 i 边表的下一节点（边）
                        i=p;    //更换到邻接顶点 p 的边表
                        }
                    else next[i]=next[i]->link; //此顶点已经标记过，更换边
                    }
                }
            }
        }
}
```

图 1.85（a）G6 的邻接存储结构如图 1.88 所示，其深度遍历生成树如图 1.87（a）所示，程序 1.30 的输出是：

a, b, c, f, d, e, g,

对下面图 1.89（a）有向图 G7 从顶点 a 出发做深度优先遍历的结果是：

a,

b, e,

c, d,

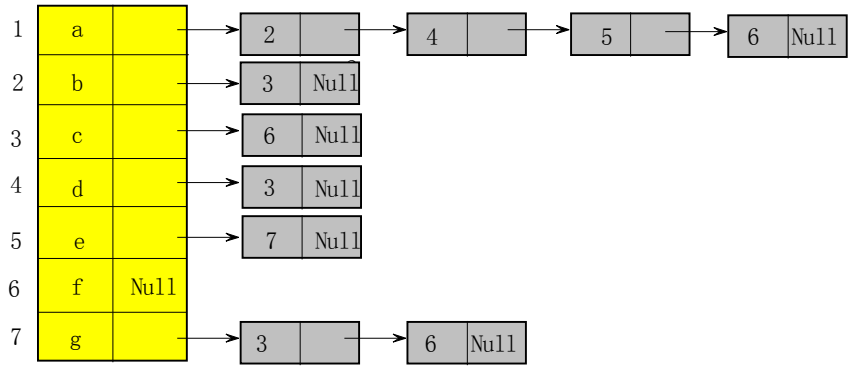


图 1.88 有向图 G6 的邻接存储结构

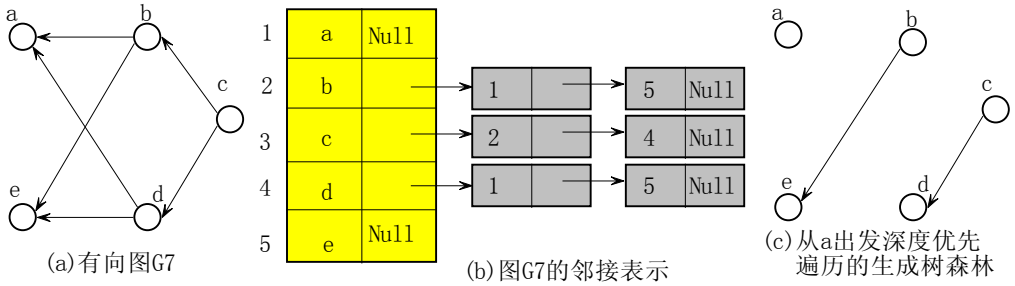


图 1.89 有向图 G7 从 a 点出发做深度优先遍历得到的生成树森林

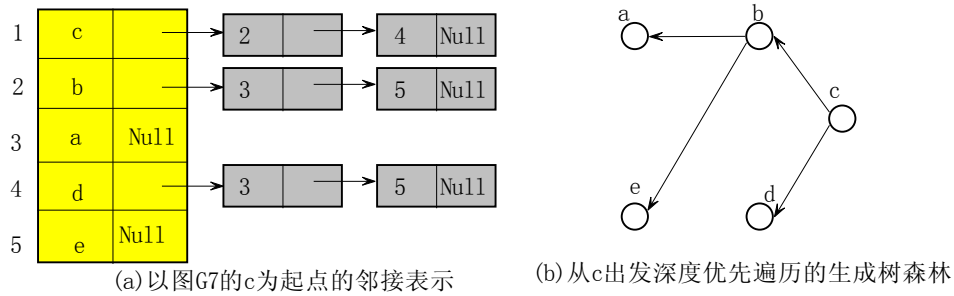


图 1.90 有向图 G7 从 c 点出发做深度优先遍历得到的生成树

图 1.90（a）和（b）是对有向图 G7 从顶点 c 出发做深度优先遍历的结果，程序 1.30 输出是：

c, b, a, e, d,

程序 1.30 对每条边访问一次，对于顶点表从头到尾检查一次，所以花费时间数量级是 O(n+m)。这里的 n 是顶点数目，m 是边数。

1.4.4 无向连通图的最小生成树（minimum-cost spanning tree:MST）

既然从不同的顶点出发会有不同的生成树，而 n 个顶点的生成树有 n-1 条边，那么，当边带权的时候（网络），如何寻找一个（网络中）的最小生成树（即树中各边权值之和最小）？

图 1.91（a）显示了一个城市之间公路网络的例子。各边权值是距离，要把 6 个城市联结起来至少需要构筑 5 条道路，所谓最小生成树就是求距离总和为最短的网络连接。

给定一个无向连通图 G，构造最小生成树的 prim 算法得到的 MST 是一个包括 G 的所有顶点、及其边的子集的图，而这个边的子集满足：

（1） 该子集的所有边的权值之和是图 G 所有边子集之中最小的；

（2） 子集的边能够保证图是连通的。

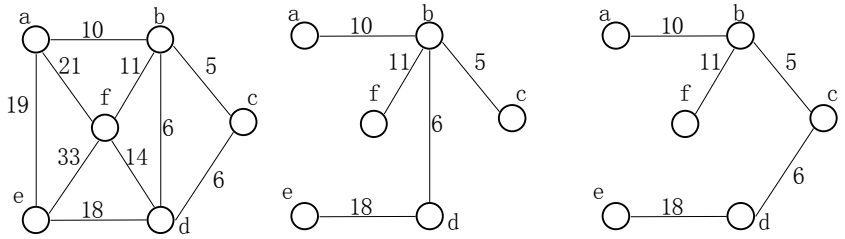


图 1.91 城际公路网络与最小生成树

显然,MST 的边集中没有回路,否则可以通过去掉回路中某条边而得到更小权值的边集。因此，n 个顶点的 MST 有 n-1 条边。

Prim 算法很简单：从任意顶点 n 开始，首先把这个顶点包括进 MST 中（初始化 MST 为 n），在与 n 相关联的边中选出权值为最小的边及其与 n 相邻的顶点 m。把 m 和边（n,m）加入到 MST 中。然后，选出与 n 或 m 相关联的边中权值最小的边及相邻顶点 k，同样，把 k 也加入到 MST 中。继续这一过程，每一步都通过选出连接当前已经在 MST 中的某个顶点、以及另一尚未在 MST 中顶点的权值为最小的边而扩展 MST。直至把所有顶点包括进 MST。

若有两个权值相等的边，可以任选其一加入到 MST 中，因此，MST 不唯一。图 1.91（b）和（c）显示了这种情况。

MST 的生成过程可以这样理解,反复在图 G 中选择具有最小权值的边及相邻顶点加入到 MST 中，直至所有顶点进入到 MST。比如图 1.91（b）的生成过程如图 1.92 所示。

显然，prim 算法也是典型的贪心算法步骤，有定理证明 prim 算法产生的就是最小生成树，详细证明过程可以参考文献[1]。

程序 1.31 是 prim 算法的 c 语言实现。程序假设权值始终大于零，图用相邻矩阵表示。若（v_i,v_j）是边，则矩阵 A[i,j]是它的权值。若（v_i,v_j）不是边（即顶点 v_i 与 v_j 不相邻），则矩阵 A[i,j]的值是一个比任何权值都大得多的正数（INFINITY ）。因为是无向图的相邻矩阵关于对角线对称，且对角线元素为零，所以我们只存储下三角矩阵。

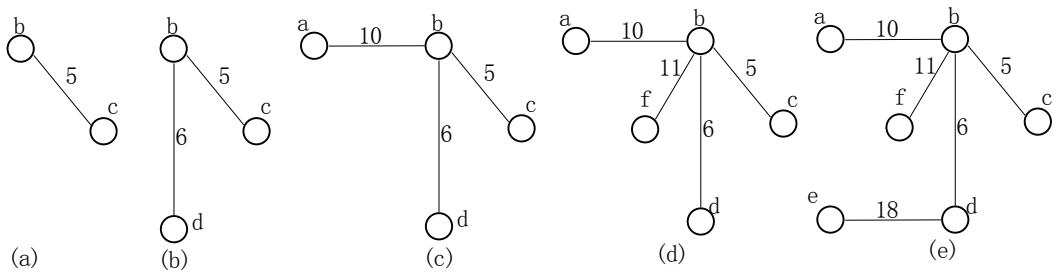


图 1.92 MST 的生成过程

用一维数组 array 存储相邻矩阵的下三角矩阵，显然，下三角矩阵的元素总数是：

$$total=\sum_{i=1}^{n} i=\frac{n(n+1)}{2}$$

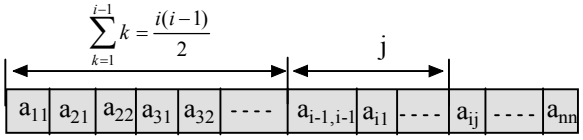


图 1.92 元素下标计算

图 1.92 给出了矩阵元素下标计算方法。边 A[i,j]在 array 中的位置下标 l 就是前 i-1 行元素的个数加上 j：

$$l=j+\sum_{k=1}^{i-1} k=\frac{i(i-1)}{2}+j$$

程序在构造 MST 过程中，如果顶点 v_i 已经在 MST 中，则置 A[i,i]=1。若边（v_i,v_j）在 MST 中，则置 A[i,j]=- A[i,j]。也就是说，程序调用结束的时候，相邻矩阵中为负的元素是 MST 中的边。

程序 1.31 无向连通图的 MST

```
void MSTtree(int *array,int n)   //数据存储在相邻矩阵 array[1]～array[n]
{
    int i,j,k,m,p=0,q=0,min;
    array[n*(n+1)/2]=1;           //从顶点 n 开始构造 MST
    for(k=2;k<=n;k++){
        min=INFINITY;             //每次选择新加入 MST 的边都重新设置权值门限为+∞
        for(i=1;i<=n;i++){        //选择符合条件的最小权的边
            if(array[i*(i+1)/2]==1){//如果顶点 i 已在 MST 中
                for(j=1;j<=n;j++){
                    if(array[j*(j+1)/2]==0){//如果顶点 j 尚未在 MST 中
                        if(j<i)m=i*(i-1)/2+j;//j<i 则边在对角线左侧的下三角中，
                                             //直接取边（i,j）的位置
                        else m=j*(j-1)/2+i; //j>i 则边在对角线右侧的上三角中，
```

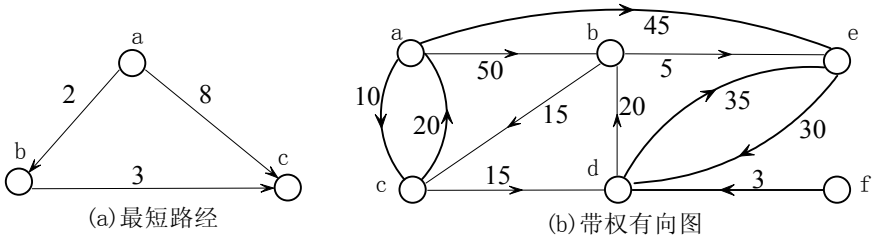
```
                                        //对应下三角中是（j,i）的位置
                        if(array[m]<min){
                            min=array[m];
                            p=m;q=j;
                            }
                        }
                    }
                }
            }
        array[q*(q+1)/2]=1;//将选中的边加入到MST
        array[p]=-array[p];//将选中的顶点加入到MST
        }
}
```

图 1.91（a）无向连通图的相邻矩阵是下面的矩阵 A，程序 1.31 输出的 MST 相邻矩阵是下面的矩阵 A’（假设权值均小于 INFINITY :99)。

$$A=\begin{bmatrix} 0 & & & & & \\ 10 & 0 & & & & \\ 99 & 5 & 0 & & & \\ 99 & 6 & 6 & 0 & & \\ 19 & 99 & 99 & 18 & 0 & \\ 21 & 11 & 99 & 14 & 33 & 0 \end{bmatrix},\quad A'=\begin{bmatrix} 1 & & & & & \\ -10 & 1 & & & & \\ 99 & -5 & 1 & & & \\ 99 & -6 & 6 & 1 & & \\ 19 & 99 & 99 & -18 & 1 & \\ 21 & -11 & 99 & 14 & 33 & 1 \end{bmatrix}$$

1.4.5 有向图的最短路径

城际公路网络有距离标定。如果两城市之间有路连通，且经中间城市可以有多条道路到达，那么，距离最短的连通路径是我们的首选。注意，距离最短是指路径上的边带权总和最小，而不是路径上的边数最少。比如图 1.93(a)，顶点 a 和顶点 c 之间的最短路径是 abc。



(a)最短路经　　(b)带权有向图

图 1.93

1.4.5.1 单源最短路径（single-source shortest paths）

图 1.93（b）的相邻矩阵 A（顶点 a，b，c，d，e，f 对应序号 1，2，3，4，5，6）如下所示。单源最短路径就是从某一顶点 v_0 出发，到达图中其它各顶点的最短路径。

$$A=\begin{bmatrix} 0 & 50 & 10 & +\infty & 45 & +\infty \\ +\infty & 0 & 15 & +\infty & 5 & +\infty \\ 20 & +\infty & 0 & 15 & +\infty & +\infty \\ +\infty & 20 & +\infty & 0 & 35 & +\infty \\ +\infty & +\infty & +\infty & 30 & 0 & +\infty \\ +\infty & +\infty & +\infty & 3 & +\infty & 0 \end{bmatrix}$$

求最短路径的 Dijkstra 算法基本思想是：把图中所有顶点分成两组，第一组是相对于顶点 v_0 的路径已经确定为最短路径的那些顶点，第二组中的顶点是尚未确定最短路径的顶点，每次从第二组中挑选出路径最小的那个顶点，加入到第一组中，直至从顶点 v_0 出发可以达到的所有顶点都包括进第一组内。

算法进行过程中，总是保持从 v_0 到第一组各顶点的最短路径长度均不大于从 v_0 到第二组的任何顶点的最短路径长度。因为每个顶点对应一个距离值，第一组内各顶点的距离值是相对 v_0 到该顶点的最短路径长度值，第二组内各顶点的距离值是从 v_0 到该顶点的、只允许经第一组内顶点中间转接的最短路径长度值。算法步骤：

① 初始第一组内只有出发点顶点 v_0，它的最短路径值相对于自己就是 0，第二组内包括图内其它所有顶点。第二组内各顶点的距离值如下确定：若图中有边$\langle v_0, v_k\rangle$，则 v_k 的距离值就是此边的权值 $A[i_0, j_k]$，否则 v_k 的距离值的初值就是$+\infty$；

② 从第二组中挑选出路径（距离值）最小的那个顶点 v_m 放入第一组中；

③ 第一组每增加一个顶点 v_m，从 v_0 经该顶点中间转接达到第二组各顶点的最短路径就可能发生变化，因此各顶点距离长度值都需要修正。方法是：如果经 v_m 做中间转接，使得 v_0 到 v_k 的最短路径比不经过 v_m 转接，直接到达 v_k 的距离短，则修改 v_k 的距离值为 $A[i_0, j_m]+ A[i_m, j_k]$。

④ 重复步骤，直至图中所有顶点或者都进入第一组，或者仅剩余那些与 v_0 没有路经可达的顶点。

Dijkstra 算法仍然是贪心算法的基本思想。要证明算法的正确性，只需证明第二组内距离值最小的点 v_m，就是从第一组的 v_0 到第二组 v_m 的最短路径长度[2]：

① 若 v_m 到距离值不是从 v_0 到 v_m 的最短路径长度，那么，必有一条从 v_0 到经过第二组内其它顶点到达 v_m 的路径，其长度比 v_m 的距离值小。假设该路径经过第二组的顶点 v_s，则：

v_s 的距离值$<$从 v_0 到 v_m 的最短路径长度$<v_m$ 的距离值

这与 v_m 是第二组内距离值最小的点矛盾，所以，v_m 到的距离值就是从 v_0 到 v_m 的最

短路径长度。

② 假设 v_x 是第二组中的任意一顶点，若 v_0 到 v_x 的最短路径只包含第一组内的顶点中转（无需经过 v_m），则由距离定义可知其路经长度必然不小于 v_0 到 v_m 的最短路径长度，若 v_0 到 v_x 的最短路径不仅包含第一组内顶点，而且也经过了第二组的顶点 v_y，因为，v_0 到 v_y 的路径就是 v_y 的距离值，它一定大于或等于从 v_0 到 v_m 的最短路径长度，再考虑到 v_y 到 v_x 的路径长度，那么，从 v_0 到 v_x 的最短路径长度必定大于从 v_0 到 v_m 的最短路径长度，因此，第二组内距离值最小的顶点 v_m 到，就是从 v_0 到第二组内的最短路径点。

Dijkstra 算法的程序实现对有向图采用了相邻矩阵存储。若<v_i,v_j>是边，则 A[i,j]的值等于边的权。否则设置 A[i,j]=INFINITY。初始 A 的对角线元素为 0，处理中用 A[i,j]=1 表示第 i 个顶点进入第一组。辅助数组 dist[]的每个元素包括两个字段，length 字段的值是顶点相对 v_0 的距离，pre 字段则指示从 v_0 到该顶点 v_k 的路径上前趋顶点的序号。程序结束时，沿着前趋序号可以回朔到 v_0，从而确定从 v_0 到 v_k 的最短路径，最短路径长度值存储在 v_k 的 length 字段。程序 1.32 是单源最短路径算法的 C 语言实现，而变量 k 指明了出发点 v_0。

程序 1.32 单源最短路径

```
void shortestPaths(struct node *dist,int *array,int n,int k)
{//权值存储在相邻矩阵 array[1]～array[n*n]中
    int i,u,temp,min;
    for(i=1;i<=n;i++){
        dist[i].length=array[n*(k-1)+i];//相邻矩阵第 k 行元素值就是与 k 关联的边
        if(dist[i].length!=INFINITY)dist[i].pre=k;//(k,i)之间有弧存在，前趋是 k
        else dist[i].pre=0; //(k,i)之间没有弧存在
        }
    array[n*(k-1)+k]=1;      //顶点 k 进入第一组
    while(1){
        min=INFINITY;
        u=0;
        for(i=1;i<=n;i++){  //在第二组中寻找最小距离点
            if((array[n*(i-1)+i]==0)&&(dist[i].length<min)){
                u=i;min=dist[i].length;
                }
            }
```

```
        if(u==0)break;        //若 vk 和其它点均不相邻程序结束
        array[n*(u-1)+u]=1; //vi 放进第一组
        for(i=1;i<=n;i++){  //修改第二组中各点距离
            temp=dist[u].length+array[n*(u-1)+i];
            if((array[n*(i-1)+i]==0)&&(dist[i].length>temp)){
                dist[i].length=temp;
                dist[i].pre=u;
                }
            }
        }
}
```

输入图 1.93（b）的相邻矩阵 A，指定 k=1(即从 v_1 开始)，程序 1.32 的 disp[]输出是：

(0,1),(45,4),(10,1),(25,3),(45,1),(99,0)

返回的 dist[]各元素 pri 字段值指示我们从 v_1 到各顶点的最短路径。比如，想求 v_1 至 v_4 的最短路径，从 disp[4].pre=3 可知，最短路径上的 v_4 前趋是 v_3，而 disp[3].pri=1 说明 v_3 的前趋就是 v_1，即 v_1 至 v_4 的最短路径是 v_1，v_3，v_4，最短路径值是 disp[4].length=25。

Dijkstra 算法有两重循环。主循环中处理顶点 v_k 到图中其余 n-1 个顶点的最短距离，内循环中寻找第二组内具有最小距离的点，也是扫描 n 个顶点，因此，其计算效率是 n^2 数量级的。

1.4.5.2 每对顶点间最短路经（all-pairs shortest paths）

实际运用中，我们不但要知道从某一顶点到网络内其余顶点的最短路径，往往还需要知道网络内每两个顶点之间的最短路径。一种方法是通过反复 n 次调用程序 1.32 可以简单的解决这个问题。另一种方法是如下所述，概念上更为清晰。

设有向图用相邻矩阵描述。我们这样定义相邻矩阵的阶 k：k 阶相邻矩阵 $A^k[i,j]$ 的元素值等于从顶点 i 到顶点 j 之间的最短路径上，允许经过中间顶点数目（边数）不大于 k 的最短路径长度。而 n 阶相邻矩阵 $A^n[i,j]$ 的元素值，等于从顶点 i 到顶点 j 之间的最短路径上经过中间顶点数目不大于 n 的最短路径长度，即从顶点 v_i 开始，允许经过图中所有 n 个顶点达到 v_j 的最短路径。

我们定义 $A^0[i,j]=A[i,j]$，由 $A^{k-1}[i,j]$ 递推求得 $A^k[i,j]$ 的过程，就是允许越来越多的顶点作为 v_i 到 v_j 路径上的中间顶点的过程，直至包括图内所有的顶点都可以作为中间顶点，从而求得 v_i 到 v_j 的最短路径。

如果 $A^{k-1}[i,j]$ 已知，递推求 $A^k[i,j]$ 就是在 $A^{k-1}[i,j]$ 上增加顶点 v_k 作为中间点，这有两种情况：

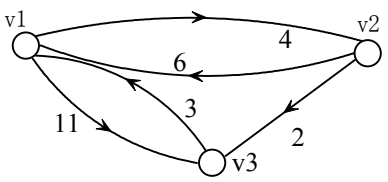
① v_i到v_j的最短路径上经过v_k，即$A^k[i,j]<A^{k-1}[i,j]$（否则没有增加v_k的必要），$A^k[i,j]$由两段构成，一段是从v_i开始经过k-1个顶点达到v_k的最短路径$A^{k-1}[i,k]$，另一段是从v_k开始经过k-1个顶点达到v_j的最短路径$A^{k-1}[k,j]$，因此，$A^k[i,j]$是它们之和：$A^k[i,j]=A^{k-1}[i,k]+A^{k-1}[k,j]$。

② 第二种情况是v_i到v_j的最短路径上没有经过v_k，即$A^k[i,j]=A^{k-1}[i,j]$。

程序1.33是算法的c语言实现。初值$A^0[i,j]$就是相邻矩阵，结束时$A^n[i,j]$是每对顶点之间的最短路径长度。辅助矩阵path[i,j]是从v_i开始经过k-1个顶点达到v_j的最短路径上v_j的前趋顶点序号。我们可以由path[i,j] 的元素值回朔最短路径。

程序1.33 每对顶点间最短路经

```
void AllshortestPaths(int *path,int *array,int n)
{
    int i,j,k,temp;
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++){
            if(array[n*(i-1)+j]!=INFINITY)path[n*(i-1)+j]=i;//初始前趋
            else path[n*(i-1)+j]=0; //没有前趋
            }
    for(k=1;k<=n;k++)                    //递推更新最短路径矩阵
        for(i=1;i<=n;i++)
            for(j=1;j<=n;j++){
                temp=array[n*(i-1)+k]+array[n*(k-1)+j]; //两段最短路径之和
                if(temp<array[n*(i-1)+j]){
                    array[n*(i-1)+j]=temp; //取新的最短路径
                    path[n*(i-1)+j]=path[n*(k-1)+j]; //(i,j)的前驱是(k,j)
                    }
                }
}
```



(a)带权有向图

$$A=\begin{bmatrix}0 & 4 & 11\\ 6 & 0 & 2\\ 3 & +\infty & 0\end{bmatrix}$$

(b)相邻矩阵

图1.94

图1.94（a）和（b）分别给出了一个带权有向图和其相邻矩阵。由程序1.33得到的每

对顶点间最短路径矩阵 A^n 和辅助矩阵 path 如下：

$$A=\begin{bmatrix}0 & 4 & 6\\ 5 & 0 & 2\\ 3 & 7 & 0\end{bmatrix},\ \text{path}=\begin{bmatrix}1 & 1 & 2\\ 3 & 2 & 2\\ 3 & 1 & 3\end{bmatrix}$$

由 path[i,j] 的元素值可以回朔最短路径，比如，由 A^n[2,1]可知 v_2 到 v_1 的最短路径长度是 5，由 path[2,1]=3 可知 v_2 到 v_1 最短路径上，v_1 的前趋是 v_3，由 path[2,3]=2 可知 v_3 的前驱是 v_2。

显然，程序 1.33 的求解效率是 n^3 数量级的。

1.4.6 拓扑排序

拓扑排序是对一个拓扑结构上存在部分有序的元素进行分类的过程。 比如如下情形是部分有序：

（1） 在字典中，单词是用别的单词来定义的。如果单词 u 用单词 w 来定义，记为 $w \prec u$，在字典中对单词进行拓扑分类是说把词汇排列得没有前项引用[3]；

（2） 一项工程项目由多个任务子项组成。某些子项必须在其它子任务实施之前完成，如果子任务 w 必须在 u 之前完成，记为 $w \prec u$，项目的拓扑分类是说把所有任务子项排列得在每一个子任务开始的时刻，其前项准备工作已经完成；

（3） 在大学课程中，因为课程要依赖它的预备知识，所以某些课程必须在其它课程之前先修。如果课程 w 是课程 u 的先修，记为 $w \prec u$，教学计划的拓扑分类是这样排列课程，使其任何课程不会安排在先修课程之前。

一般说，集合 V 上的一个部分有序是 V 的元素之间存在一种关系，记为符号 $\prec$，称之为“先于”。并且，符合先于关系的 V 中的任何元素必须具有如下性质：

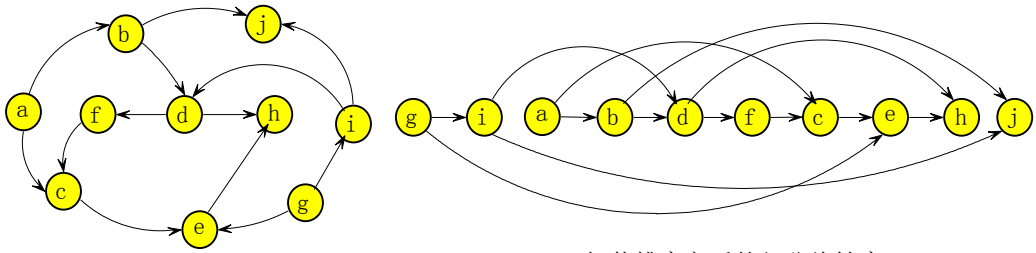
（1） 若 $x \prec y$ 且 $y \prec z$，则 $x \prec z$（传递性质）；

（2） 若 $x \prec y$，则不能有 $y \prec x$（反对称性质）；

（3） $x \prec x$ 不成立（非自反性质）。

可以用一个有向图描述部分序关系，比如图 1.95（a）。从图形结构上看，拓扑排序就是将存在部分有序的元素排列成线性有序。即将图的顶点排列成一排，使得所有的有向边箭头都指向右边，如图 1.95（b）所示。

前述的性质（1）和性质（2）保证了拓扑图中不会出现回路，这是插入成线性有序的先决条件。我们如下进行拓扑排序的算法：①寻找一个这样的顶点，它没有先于的元素存在（拓扑图中至少存在一个满足条件的顶点，否则图中就会存在回路）。②把该顶点放到拓扑排序结果表的首部，并从集合 V 中删除该顶点。③剩余的集合中仍然是部分有序的，因此，重复上述过程，直至集合为空。



（a）部分有序的拓扑结构图　　（ b）拓扑排序之后的部分线性序

图 1.95

图结构选择多重链表形式，节点定义如下：

```
struct leader{
    char letter;          //顶点标识
    int count;                //前趋顶点个数
    struct leader *next;//指向图顶点集合中下一个元素的指针
    struct trail *trail;//指向该顶点边表的指针
    };
struct trailer{
    struct node *id;          //指向该弧（边）的另一端顶点的指针
    struct leader *next;//指向与该顶点连接的其余弧（边）的指针
    };
```

集合元素（图的顶点）以及次序关系的输入方式是按相邻顶点对的形式输入，比如，图 1.95（a）中的元素及边的关系输入形式是：

<a, b> <b, d> <d, f> <b, j> <d, h> <f, c> <a, c> <c, e> <e, h> <g, e> <g, i> <i, d> <i, j>

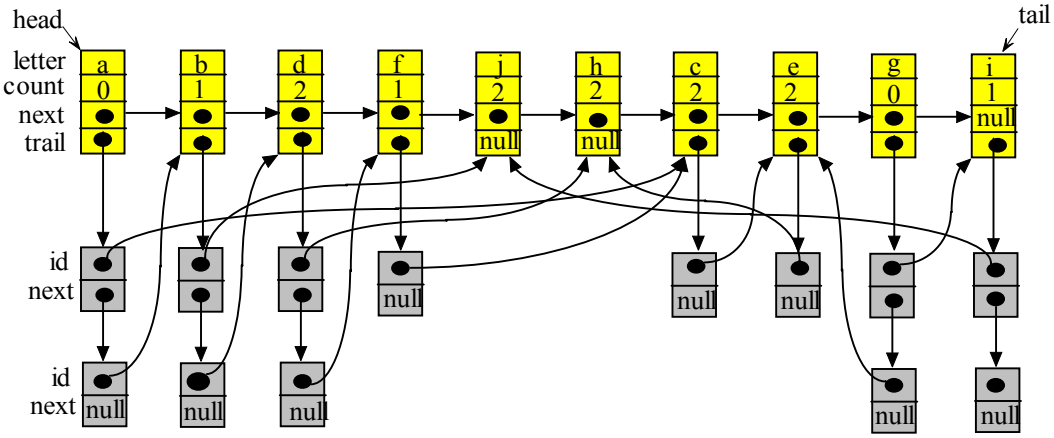


图 1.96 拓扑结构的多重链表存储形式

程序 1.34 是多重表数据输入程序，对于输入的 w，如果表中有此顶点，则它返回指向标识为 w 的图节点的指针，否则，新增一个图节点 t，标识为 w 并返回指向顶点 t 的指针。程序 1.35 是根据部分序建立多重表的程序。每次输入顶点对<v_i, v_j>，若拓扑图中已有顶点

v_i，则为它增加一条边，否则新增一个图节点 v_i。若拓扑图中已有顶点 v_j，让 v_i 指向 v_j，修改 v_j 的前趋个数，否则新增一个图节点 v_j，让 v_i 指向 v_j，并且设置 v_j 的前趋个数为一。

程序 1.34 数据输入

```
struct leader *insert(struct leader *head,struct leader *tail,char w,int &z)
{
    struct leader *h,*p;
    h=head;
    tail->key=w;     //监视哨
    while(h->key!=w){p=h;h=h->next;}
    if(h==tail){     //表中无 w 的元素，插入 w
        h=new(leader);  //申请图节点
        z++;            //图节点总数加一
        h->key=w;       //标识为 w
        h->count=0;     //初始的前趋节点数为 0
        h->trail=Null;
        p->next=h;      //插入到多重表末端
        h->next=tail;
        }
    return(h);          //返回指向标识为 w 的顶点的指针
}
```

程序 1.35 建立多重表

```
void link(struct leader *head,struct leader *tail,int &z)
{
    char x,&rx=x,y,&ry=y;
    struct leader *p,*q;
    struct trail *t;
    read(rx,ry);
    while(x!='0'){                    //x=0 则节点对输入终止
        printf("<%c,%c>\n",x,y);
        p=insert(head,tail,x,z);//插入 x，返回指向 x 的指针
        q=insert(head,tail,y,z);//插入 y，返回指向 y 的指针
        t=new(trail);                 //申请边节点
        t->id=q;                  //x 指向 y
```

```
        t->next=p->trail;              //把新边节点插入到 x 的边表
        p->trail=t;
        (q->count)+=1;                 //y 的前趋个数增加一
        read(rx,ry);                 //继续输入<x,y>
        }
}
```

建立了部分序的多重链表之后，可以开始拓扑分类。在一个拓扑结构上对部分有序的元素进行分类的过程如下：

（1） 在图节点链上寻找前趋为零的节点 q，并建立新链 leader；

（2） 从新链头部开始输出 q，并从主链上删除，节点总数减一；

（3） 沿 q 点边表搜索，并将 q 所有后继的前趋计数值减 1(因它们的前趋 q 被删除)；

（4） 若某一点的前趋计数值为变为 0，则把它插入新链；

（5） 若新链非空，回到步骤②循环，继续输出有部分序的节点。

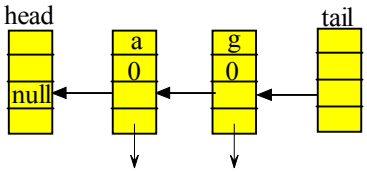


图 1.97 初始的无前趋节点链

图 1.97 是无前趋节点的新链，因为主链表达节点输入序列关系已经不需要，所以新链实际上使用了主链来连接无前趋节点，注意它们的边表关系没有改变。程序 1.36 是多重表结构的拓扑排序函数。程序首先寻找所有 count=0 的起点，并将它们插入到主链。然后从头开始输出主链的节点，每输出一个就将节点总数减一，并沿指向 q 的后继节点的指针，搜索 q 的后继关系。将 q 每一个后继节点的前趋数减一之后，前趋计数值为变为 0，则该后继节点只有 q 为前趋，程序也把它插入到主链，应该紧接着 q 之后输出。

当程序结束时拓扑图中已经没有无前趋的元素，如果还有剩余节点，表明该拓扑结构不是部分有序的。

程序 1.36 拓扑排序

```
void topsort(struct leader *head,struct leader *tail,int &z)
{
    struct trail *t;
    struct leader *p,*q;
    p=head;head=Null;
    while(p!=tail){          //寻找表内所有 count=0 的起点
        q=p;
```

```
        p=p->next;
        if(q->count==0){
            q->next=head;    //插入到主链
            head=q;
            }
        }
    q=head;
    while(q){
        printf("%c,",q->key);// 输出主链节点
        z--;                 //节点总数减一
        t=q->trail;          //指向取 q 的后继节点
        q=q->next;           //指向主链下一个部分序的起点
        while(t){
            p=t->id;
            (p->count)-=1;   //q 后继节点的前趋数减一
            if(p->count==0){//如果该后继节点只有 q 为前趋则将其插入主链准备输出
                p->next=q;
                q=p;
                }
            t=t->next;           //搜索 q 的其余后继
            }
        }
    //程序结束时拓扑图中已经没有无前趋的元素，如果还有剩余节点，表明该拓扑结构不是部分有序
    if(z!=0)cout<<"This set is not partially ordered"<<endl;
}
```

第二章 检索

检索又叫查找。它是对某一同类型元素集合构成的检索表做某种操作，因此它不是数据结构，而是与数据结构相关的运算问题。检索的一般含意有：

1·查询特定的元素是否在检索表中。

2·检索一个元素的各种信息（属性）。

3·如果特定的元素不在检索表中则插入此元素到表内。

4·检索到特定的元素后从表中删除此元素。

一般我们把前两种操作的检索称为静态检索，因为它不改变检索表的结构，而把包含后两种操作的检索称为动态检索，它有可能改变检索表中的元素。检索需要元素具有特定的标识，即所谓的关键字。

定义：关键字是数据元素中某个数据项的值，如果用它能唯一标识一个元素，则称为主关键字，若几个元素具有相同的关键字，则称此数据项为次关键字。

检索有成功，也有失败的可能。我们定义它的过程是：检索是根据一个给定值在检索表中确定一个关键字值等于该值的数据元素的过程。如果检索成功，其给出的结果是元素在检索表中位置的指针（或元素某些数据项信息），如果检索失败，即表中不存在关键字值等于给定值的元素，应返回空指针。

2.1 顺序检索

顺序检索的特点：从表头开始对检索表元素逐一比较，用给定值检索关键字。它适应的存储结构是链式或顺序存储结均可，对表中元素无排序要求。因为顺序检索的思想很简单，我们直接给出它 C 语言程序如下：

- **算法**

程序 2.1 顺序检索

```
int search(struct  node *p,int n,int  key)
{
      int i=0;                  //返回时 i 为 n 则检索失败，否则为元素在表中地址偏移
      *(p+n).key=key;           //设置监视哨
      while(*(p+i).key !=key)i++;
      return(i);
}
```

程序表达了一个重要的技巧，即在数组的第 n+1 个位置设置了一个监视哨，返回时根据

i 值做检索成功与否的判别。

- **检索性能分析**

定义：为确定元素在检索表中的位置，需和给定值进行比较的次数的期望值是检索算法在检索成功时的平均检索长度 ASL。设表长为 n，则顺序检索的 ASL 是：

$$ASL = \sum_{i=1}^{n} P_i \times C_i$$

这里，P_i 是检索表中第 i 个元素的概率，且 $\sum_{i=1}^{n} P_i = 1$。上式意为检索表中必有欲检索的元素存在，实际上检索有成功也有失败，二者必居其一，因此有 p+q=1。在考虑检索失败情况下，我们定义此时的平均检索长度是检索成功时的平均检索长度与检索失败时的平均检索长度之和。

设 C_i 是找到表中其关键字与给定值相等的第 i 个记录时和给定值已进行过比较的关键字个数。在顺序表中检索到第 1 个元素需比较 1 次，检索到第 2 个元素时需比较 2 次，即检索到第 i 个元素时已比较过的次数是 i，并设检索每一元素的概率相等，则有：

$$ASL = \frac{1}{n}\sum_{i=1}^{n} i$$
$$= \frac{n+1}{2}$$

可以直接理解为检索时最好的情况是欲检索元素在表中第一个位置上，一次比较后检索成功，最坏情况是元素在表的末尾，需 n 次比较后才确定，两种情况的平均即是我们说的平均检索长度。当元素不在表中时，我们总是需要比较 n＋1 次才可以确定，设检索成功的概率是 p，则检索失败的概率是 1-p，在设有监视哨的程序算法中，检索失败的时候是比较了 n+1 次，所以有：

$$ASL = \frac{p(n+1)}{2} + (1-p)(n+1)$$
$$= (n+1)(1-\frac{p}{2})$$

2.2 对半检索

2.2.1 对半检索与二叉平衡树

对半检索的特点是仅适用于已排好序的检索表，且要求是顺序存储结构。这种方法也叫二分检索，是在已排好序的检索表中每次取它的中点关键字值比较，形成两个前后子表，如检索成功则退出，否则根据结果判别下一次检索在哪个子表中进行，重复分割该子表直

至找到要检索的元素或子表长度为零。

算法：设表长为 n,表头指针为 low,表尾指针为 high，key 为关键字。

1・若 low ≤ high 则：

mid=int[(low+high)/2]

if(key==*(p+mid).key)检索成功

else if(key<*(p+mid).key)high=mid-1；重复步骤 1。

else if(key>*(p+mid).key)low=mid+1；重复步骤 1。

这个算法比较简单，读者要注意其中一些判别条件的设置，本教材不直接给出程序。

例 2.1 已知一有序表 T 中元素是{5，13，19，21，37，56，64，75，80，88，92}， 图 2.1 给出了检索 21 的情况。

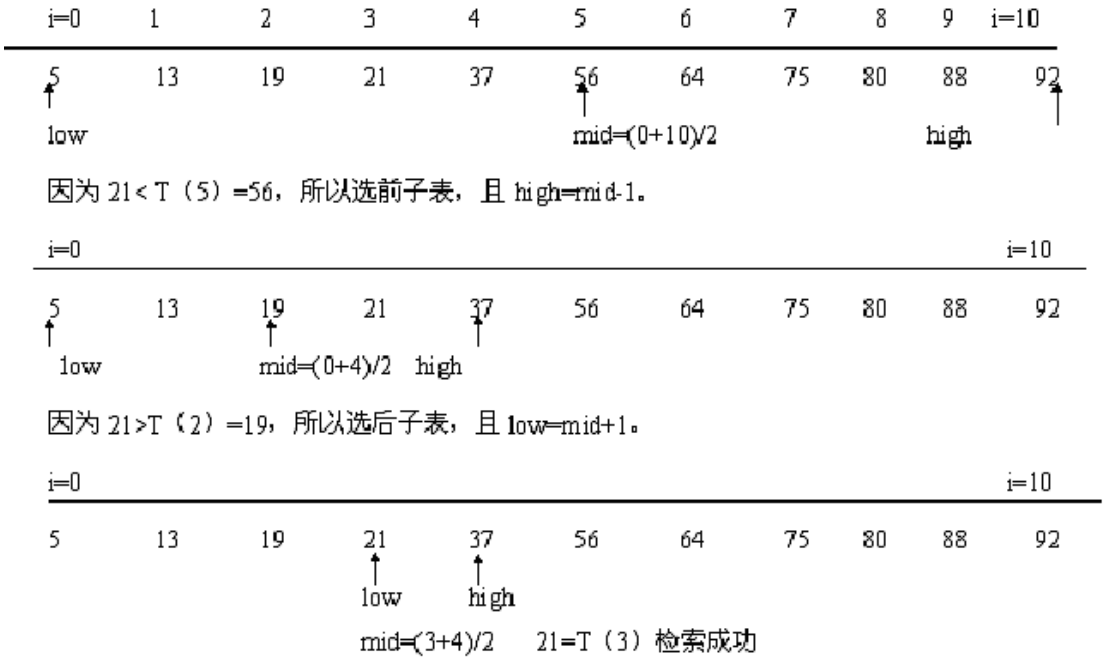


图 2.1 对半检索表及关键字为 21 的检索过程

现在我们分析检索失败的情况。比如检索 85，它处于检索表外，3 次对半检索之后有 high=8，而 low=9，因为 high<low 循环条件破坏而检索失败，过程如下所示：

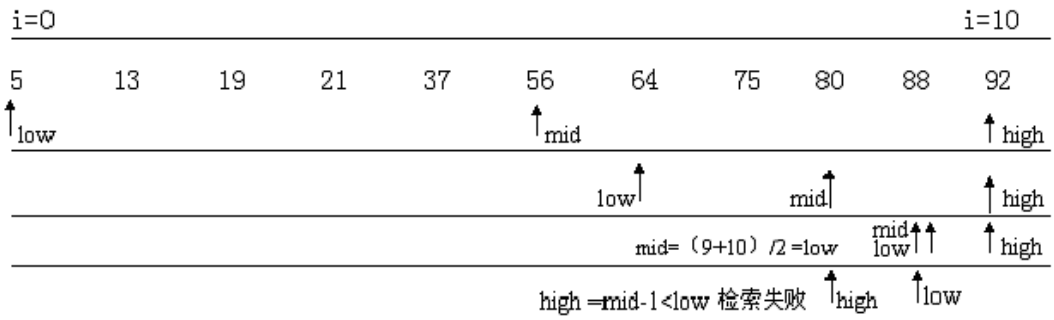


图 2.2 检索关键字为 85 的检索过程

关于性能分析，从判定树来看上面两种情况是如图 2.3 所示。一旦有序检索表确定，则判定树确定，不同的给定值的检索过程是判定树上不同的路径表达。检索表中每一元素

都是判定树上不同层次的节点，有着不同的检索长度，最大检索长度是深度 h。

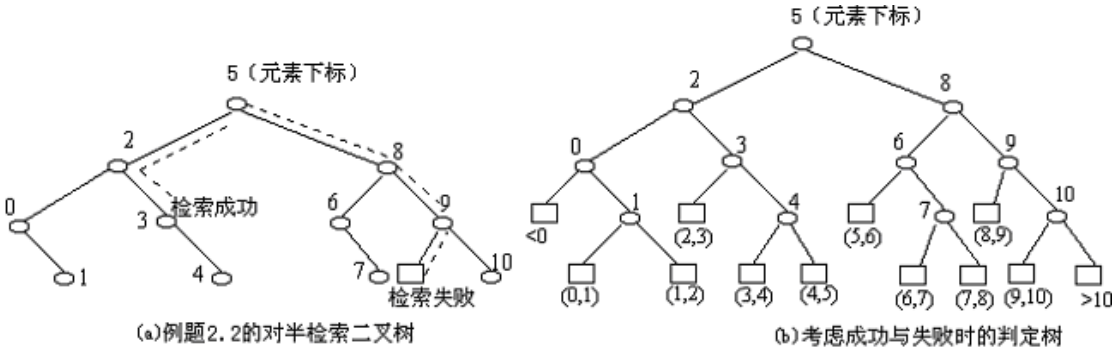


图 2.3 对半检索树

如果表内节点数 $n=2^h-1$，则判定树是满二叉树，显然，层次为 1 检索长度为 1 的节点有 1 个，层次为 2 检索长度为 2 的节点有 2 个，层次为 3 检索长度为 3 的节点有 4 个，•••，层次为 h 检索长度为 h 的节点有 2^{h-1} 个，设检索任一节点的概率相等，则平均检索长度是：

$$ASL=\frac{1}{n}\sum_{i=1}^{h} i\cdot 2^{i-1}$$

不考虑系数 $\frac{1}{n}$，直接展开上式有：

$$ASL=1\times 2^0+2\times 2^1+3\times 2^2+4\times 2^3+\ldots.+(h-1)\times 2^{h-2}+h\times 2^{h-1} \quad (1)$$

用 2×<1>式两边有：

$$2ASL=1\times 2^1+2\times 2^2+3\times 2^3+4\times 2^4+\ldots.+(h-1)\times 2^{h-1}+h\times 2^h \quad (2)$$

用（1）式－（2）式有：

$$-ASL=(1+2^1+2^2+2^3+2^4+\ldots\ldots+2^{h-1})-h\times 2^h$$

再用系数 $\frac{1}{n}$ 同乘等式两边得平均检索长度为：

$$ASL=h\cdot 2^h-\left(2^h-1\right)$$
$$=\frac{1}{n}\left[(h-1)2^h+1\right]$$

因为 $n=2^h-1$，所以 $h=\log_2(n+1)$：

$$ASL=\frac{1}{n}\left[(\log_2(n+1)-1)\cdot(n+1)+1\right]$$
$$=\frac{n+1}{n}\left[\log_2(n+1)\right]-1$$

实际上，表内节点数一般不满足 $n=2^h-1$ 条件，则判定树不是满二叉树，因为：

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

则判定树具有和完全二叉树一样的深度 $h=\lceil\log_2 n\rceil+1$（注：符号⌈ ⌉是取运算结果的上限整数），即最大检索长度不超过此值。在检索失败情况下，判定树是一个扩充二叉树

(扩充树的叶子节点是内部节点数加一：$n_0=n_2+1$,则 $2n_0+n_1=n_0+n_2+1+n_1=n_{内}+1$)，检索失败的过程就是从根走到了外部节点，其最大检索长度也不超过 $h=\lceil \log_2 n \rceil+1$。

在非等概率情况下，对半检索的判定树效率未必是最佳的。在只考虑检索成功时，我们是求以检索概率带权的内部路径长度之和为最小的判定树，称为静态最优检索树，在此不再讨论。

例 2.2 链表检索。设单链表有 n 个节点 a_1，a_2，…，a_n，递增有序，只有检索而无插入与删除操作，为提高访问第 i 个元素的效率，对链表每个节点增加一附加指针，使之检索成功的平均检索次数达到 0（log n），要求头指针仍指向 a_1。

问 1：如何设置各节点指针。

问 2：设 n=11，请画出该逻辑结构。

解 1：没有插入与删除操作说明不用考虑检索表维护问题，因此本题意是问什么链式存储结构在一个有序节点序列检索中可以达到相当于对半检索的 0（log n）平均检索效率。显然，我们可以用节点附加的指针做成一个对半检索树，各节点指针分别指向其前后子序列中点的元素，而头节点 a_1 的左指针指向序列中点。

解 2：n=11，画出对半检索树逻辑结构如图 2.4 所示。

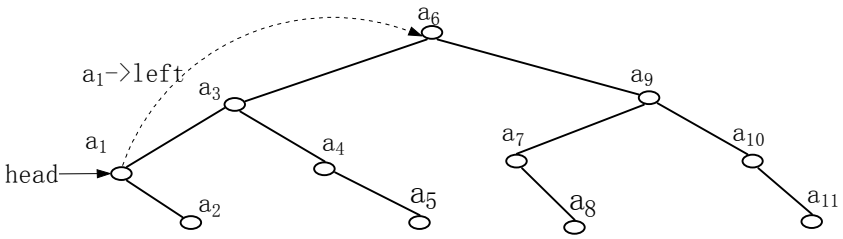


图 2.4 n=11 的对半检索树

例 2.3 对半查找。设用顺序存储结构构成的一有序表是{a_1，a_2，a_3，a_4，a_5}，已知各元素的查找概率相应是{0.1， 0.2， 0.1， 0.4， 0.2}，现用对半查找法，求在查找成功时的平均查找长度 ASL。

解：本题考查对半检索树的概念，是访问节点的概率与寻找该节点所需的比较次数乘积和。

$$ASL=\sum C_iP_i=2\times0.1+3\times0.2+1\times0.1+\times2\times0.4+3\times0.2=2.3$$

2.2.2 对半检索思想在链式存储结构中的应用---跳跃表

设计跳跃表的目的是为了解决线性表的检索效率与表的长度成线性关系的问题，跳跃表是基于概率数据结构（probabilistic data structure）确定其节点级数，因而其插入操作不需要刻意维持平衡，比平衡二叉树和 2-3 树更容易维护。

我们知道，如果检索表有序并且是顺序存储结构的，采用对半检索则平均检索时间（或称为时间复杂度）可以达到 $O(\log_2 n)$，在线性表的链式存储结构中，无论元素是否有序，其检索效率都是 0（n），实际应用中，线性表很多情况下使用链式存储结构。那么，我们如

何在链表中构造出具有对半检索效率的结构形式？ 这就是跳跃表要回答的问题。

- **跳跃表概念**

之所以链表的检索效率为 0（n），是因为遍历一个链表的过程需要沿着头节点指针域一次一个地逐点比较，搜索其后继节点。如图 2.5（a）。如果链表有序，那么我们假想给链表添加一个指针如图 2.5（b）所示，于是搜索过程可以跳跃进行，我们称之为一级跳跃表。

搜索时，首先沿着一级指针跳跃，一直到有一个节点的关键码大于检索关键码值，然后回到该节点前驱的零级指针，再多走一个节点就可以确定检索结果（相等是检索成功，不等是检索失败），这样可以把检索效率提高一半。

如果我们继续以这种方式增加节点的指针域，如图 2.5（c）所示。这是一个有 n=8 个节点的跳跃表，其第一个节点和中间节点有 $\log_2 8 = 3$ 个指针域，分别称之为 0 级，1 级和 2 级。第一次检索时候从跳跃表的最高一级指针开始，直接比较跳跃表位于 $\frac{n}{2}$ 处的中间节点。也就是说，好象是顺序表的对半检索那样。如果中间节点的关键码大于检索关键码值，退回到前一个节点，并降低一级指针级数，检索位于 $\frac{n}{4}$ 的节点，如此，根据比较结果使得跳跃步伐逐级减少，最终可以确定检索结果（相等是检索成功，不等是检索失败），因为跳跃表的检索过程完全类似于顺序表的对半检索，所以其平均检索时间有可能也是 $O(\log_2 n)$。

跳跃表节点数据结构中定义了一个指针数组，存储可能的最大级数指针，如图 2.5（c）的节点定义如下：

```
struct node{
    int key;
    struct node *forward[level];//图 2.5（c）的 level=2
    }
```

其中，forward[0]存储 0 级指针，forward[1]存储 1 级指针，依此类推。在表内按关键字检索一个节点的过程如下：

程序 2.2 跳跃表检索

```
struct node *search(struct node *head,int key,int level)
{
    int i;
    struct node *p;
    p=head;
    for(i=level;i>=0;i--)
```

```
        while((p->forward[i]!=0)&&(p->forward[i]->key<key))p=p->forward[i];
    p=p->forward[0];      //回到 0 级链，当前 p 或者空或者指向比搜索关键字小的前一个节点
    if(p->key==key)return(p);
    else return(0);
}
```

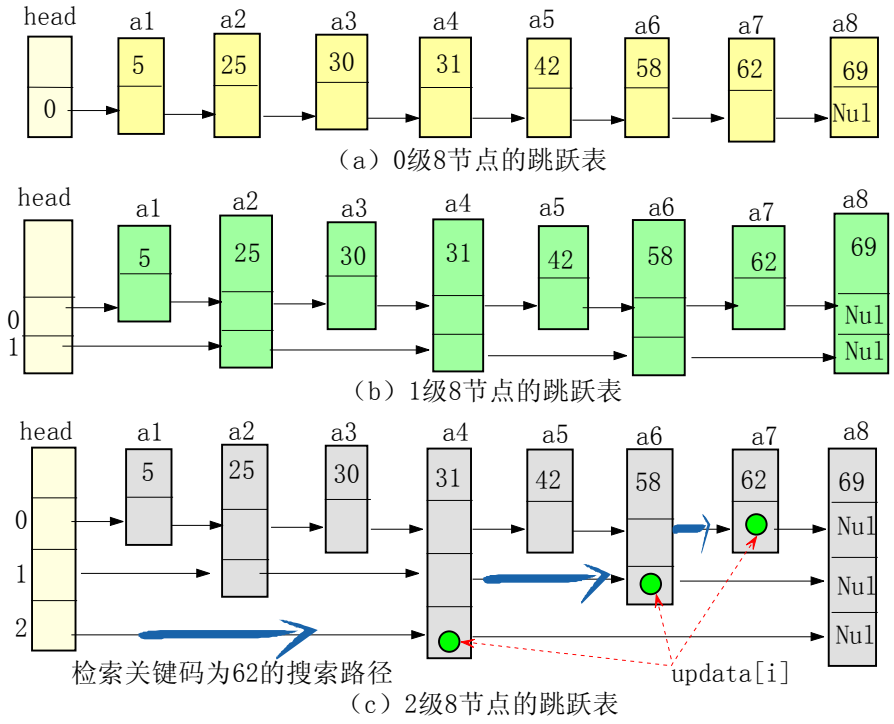


图 2.5 跳跃表

比如，在图 2.5 中检索关键码为 62 的节点。检索从头节点最高一级（level=2）指针开始，因为 head->forward[2]指向节点 a4，所以条件表达式为(p->forward[2]->key<key)，它首先比较 a4，因为 a4→key=31，小于 62，while()语句条件成立。表明检索需要继续向后搜索。

p=p->forward[i]让当前指针 p 指向 a4，而 a4→forward[2]指向 a8，则条件表达式(p->forward[2]->key<key)是 a8→key=69 和检索关键码 62 相比较，69>62，while()语句条件不成立，当前指针 p 在 for 语句循环中级数减一，是 a4->forward[1]，指向 a6。whlie()语句条件表达式(p->forward[1]->key<key)为 a6→key=58，它和 62 进行比较，满足 while()语句条件，p=p->forward[1]让当前指针 p 指向 a6。

因 a6→forward[1]=a8，而 a8→key=69>62，于是退出 whlie()语句，当前指针 p 在 for 语句循环中级数继续减一，是 a6->forward[0]，指向 a7。因为 a7→key=62，于是 while()中因（p->forward[0]->key<key）条件不满足退出，又因 i=0 而同时退出 for()循环语句。此时，p 或者空或者指向比搜索关键字小的前一个节点，我们只需要检验 p 在 0 级链上的后继是否与检索关键字相等（检索成功），否则就是检索失败。即语句：

```
    p=p->forward[0];
```

if(p->key==key)return（p）；

两语句中用 a7→key=62 和 62 进行比较，检索成功并返回指向 a7 的指针。我们可以画出图 2.5 检索表的二叉检索判定树如图 2.6 所示。在图 2.5 中用关键字值 62 进行检索，检索过程中走过的节点及对应的级数 i 序列分别是 a4、i=2，a6、i=1，a7、i=0。

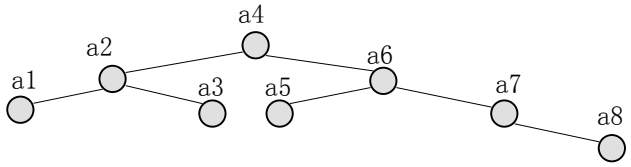


图 2.6 对应图 2.5 跳跃表的检索判定树

- **跳跃表性能**

图 2.5 给出的是完全平衡的理想跳跃表情况，其距离是平均划分的，和平衡二叉树一样，要在表的插入与删除过程维护完全平衡所花费的代价太大，而所谓基于概率数据结构的方法是说，每插入一个节点为其分配的级别（指针数）是随机的，用随机分布函数给定，其得到一个指针的概率是 50%，有两个指针的概率是 25%，以此类推。程序 2.3 可以得到一个随机级数分配数：

程序 2.3 随机级数产生函数

```
int randomlevel()
{
    return(rand()%RANGE);
}
```

在函数调用前应该使用初始化随机数发生器函数 srand()：

srand((unsigned int)time(0));

srand()的头部函数是 stdlib.h，而时间函数 time()的头部函数是 time.h。取模操作将随机数范围限制在 RANGE 以内。下面是 RANGE=7，一个循环内连续 100 次调用 randomlevel()所返回的随机函数值在 0～7 之间的分布关系（由左至右分别是指针的级数 0,1,...9），共有五次循环过程。

16, 15, 9, 14, 17, 15, 14, 0, 0, 0

9, 18, 17, 12, 16, 16, 12, 0, 0, 0

20, 12, 10, 19, 20, 8, 11, 0, 0, 0

17, 13, 9, 9, 18, 18, 16, 0, 0, 0

18, 15, 11, 15, 14, 10, 17, 0, 0, 0

即，获得 0 级指针的节点占总数的 16%，获得 1 级指针的节点占总数的 14%，获得 2 级指针的节点占总数的 11%，获得 3 级指针的节点占总数的 13%，...，获得 7 级指针的节点占总数的 14%。这样的随机分布如果是模拟掷骰子倒是不错，因为是分布平均。但显然并不符

合我们的需求。看来我们应该寻找一种更合适的随机函数。

在具有 n 个节点的平衡跳跃表结构中，其 0 级链上有 $\frac{n}{2^0}$ 个节点元素，在 1 级链上有 $\frac{n}{2^1}$ 个节点元素，…，其 i 级链上有 $\frac{n}{2^i}$ 个节点元素。所以，构建跳跃表的时候应该尽量逼近这种结构。关键是如何确定随机函数分布关系。程序 2.4 给出了以概率 $\frac{1}{2^j}$ 产生新的指针级数的方法。其中，RAND_MAX 是随机函数所能返回的最大值，MAXlevel 是我们为跳跃表设定的级数上限，它与表节点数 n 是 $\log_2 n$ 的关系。

程序 2.4 按指数分布的随机级数产生函数

```
int randX(int &level)
{
    int i,j,t;
    t=rand();
    for(i=0,j=2;i<MAXlevel;i++,j+=j)if(t>RAND_MAX/j)break;
    if(i>level)level=i;  //level 是跳跃表当前最大级数，每插入一个新节点都应更新
    return(i);           //返回随机分配给新节点的级数 i
}
```

下面是 MAXlevel=7，一个循环内连续 100 次调用 randX()所返回的随机函数值在 0～7 之间的分布关系（由左至右分别是指针的级数 0,1,...9），也是共有五次循环过程。

44,27,16,5,5,1,1,1,0,0,

53,26,11,3,6,0,0,1,0,0,

54,20,13,6,3,2,0,2,0,0,

47,28,13,6,2,2,1,1,0,0,

48,23,19,6,1,1,1,1,0,0,

显然，它基本满足我们的要求。一旦确定了一个新节点的级别，下一步就是找到其插入的位置，链接到跳跃表中。我们很关心的事情是有没有这种可能出现，就是 level 连续返回多个级别很高的值，比如 5 或者 6，造成跳跃表中许多节点都有多个指针，因为没有跳过足够多的节点，于是插入和检索性能都比较差，类似于线性表关系。反之，很多级别低的节点也不好，极端情况是都为零级节点，跳跃表退化成一个链表，其效率也成为 0（n）。我们说，极端性能的跳跃表有可能出现，但是其概率非常低，比如，连续插入 10 个节点其都为 0 级节点的可能性是千分之一左右。

跳跃表优于二叉排序树的另一点是其性能与节点插入顺序无关，随着跳跃表节点数的增加，其出现最差效率的可能性会以指数方式减少，而达到平均情况 $O(\log_2 n)$ 的可能性则迅

速增加。

- **跳跃表插入过程**

插入步骤和检索函数中根据给定的关键字搜索表中节点过程非常类似。假设跳跃表按递增有序。程序首先从最高级数链对跳跃表进行插入位置的搜索，并在搜索过程中用一辅助指针数组 updata[]纪录在每一级链的搜索路径上所走过的最远一个节点位置，见图 2.5(c)所示，它也就是要插入的新节点在每一级链上的前趋节点。因为每插入一个新节点都有可能增加跳跃表的指针级数，从而影响到搜索的起点指针级数设置。因此，必须在随机分配新节点指针级数的同时，更新当前跳跃表的最大级数 level。程序 2.4 是跳跃表插入函数。

程序 2.4 跳跃表插入

```
void insert(struct node *head,int key,int &level)
{
    struct node *p,*updata[MAXlevel];
    int i,newlevel;
    p=head;
    newlevel=randX(level);        //随机取得新节点的级数同时更新 level
    for(i=level;i>=0;i--){
         while((p->forward[i]!=0)&&(p->forward[i]->key<key))p=p->forward[i];
         updata[i]=p;          //updata[i]记录了搜索过程中在各级走过的最大节点位置
         }
    p=new(struct node);
    p->key=key;                          //设置新节点
    for(i=0;i<MAXlevel;i++)p->forward[i]=0;
    for(i=0;i<=newlevel;i++){          //插入是从最高的 newlevel 层链直至 0 层链
         p->forward[i]=updata[i]->forward[i];//插入到分配到的级数链
         updata[i]->forward[i]=p;
         }
}
```

我们为跳跃表设置一个独立头节点，初始化设置函数如程序 2.5。对于较小的跳跃表，最大级数限制 MAXlevel 取 10 就足够了。注意，主程序中要对随机函数作初始化设置：

```
srand((unsigned int)time(0));          //随机函数种子
```

程序 2.5 跳跃表初始化设置

```
struct node *initialization(int &level,int &total)
{
```

```
    int i;
    struct node *head;
    head=new(struct node);
    for(i=0;i<MAXlevel;i++)head->forward[i]=0;//head 节点的初始设置
    head->key=0;
    level=0;          //设置跳跃表当前的级数为 0
    total=0;          //节点总数为 0
    return(head);
}
```

2.3 分块检索

分块检索是存储器索引上经常采用的一种方法，效率介于顺序检索与对半检索之间。它要求检索表分块有序。若以关键码检索，则要求每一块内的关键码是此块内最大或最小的。因此，块内的关键码排列可以无序，但块与块之间的关键码是有序的。假设按递增有序，则第一块内所有关键码均小于第二块内的关键码，第二块内所有关键码又均小于第三块内的关键码，...，等等。

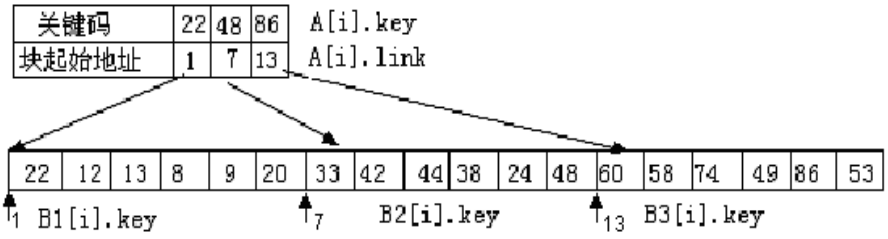


图 2.7 分块检索

分块检索首先建立一个索引表，把每块中最大的一个关键码按组顺序存放，显然此数组也是递增有序的。检索时，先用对半或顺序检索方法检索此索引表，确定满足条件的节点在哪一块中，然后，根据块地址索引找到块首所在的存储位置，再对该检索块内的元素做顺序检索。

比如图 2.7 分块检索示意。索引表元素包含了每块中节点的上边界（最大关键码）与块起始地址对，假设检索关键码是 24，首先检索索引表 A[1].key<24，继续是 A[2].key>24，确定 24 在第二块内，按 A[2].link=7 找到第二块起始地址，逐次比较到 B2[5].key=24，检索成功。分块检索效率取决于分块长度及块数：

$$E(n) = E_a + E_b$$

E_a 是索引表中确定搜索节点所在块位置的平均检索长度，E_b 是在块内检索节点的平均检索长度。设有 n 个节点平均分成 b 块，每块有 $s = \frac{n}{b}$ 个节点，再设检索概率相等且只考

虑检索成功，则：

$$E_a = \frac{1}{b}\sum_{i=1}^{b} i = \frac{b+1}{2}$$

显然 $E_b = \frac{s+1}{2}$，所以：分块检索的平均检索长度是：

$$E(n) = \frac{b+s}{2} + 1 = \frac{n+s^2}{2s} + 1$$

当 $s = \sqrt{n}$ 时分块检索的平均检索长度：

$$E(n) = \sqrt{n} + 1 \approx \sqrt{n}$$

为最小，当索引表很大的时候可以用对半检索，或者二级索引表结构进一步提高检索效率。

2.4 哈希检索

哈希检索（散列检索）是一类完全不同的检索方法，它的思想是用关键码构造一个散列函数来生成与确定要插入或待查节点的地址，因此，可以认为它检索时间与表长无关，只是一个函数的运算过程。

设 F 是一个包含 n 个节点的文件空间，R_i 是其中一个节点，i=1，2，...，n；K_i 是其关键码，如果关键码 K_i 与节点地址之间有一种函数关系存在，则可以通过该函数唯一确定的把关键码值转换为相应节点在文件中的地址：

$$ADDR\_R_i = H(K_i)$$

这里，$ADDR\_R_i$ 是 R_i 的地址，$H(K_i)$ 是地址散列函数，也叫哈希函数。所以，一旦选定了哈希函数，就可以由关键码确定任一节点在文件中的位置，例如一文件有节点{R_1，R_2，R_3}，其关键码是：

ABCD，BCDE，CDEF

选关键码第一个字符的 ASCII 值加上一常数 1000，0000H 为散列函数：

H（K）＝ASCII（K 的首字符）＋1000，0000H

H（1）＝ASCII（A）＋1000，0000H = 1100，0001H

H（2）＝ASCII（B）＋1000，0000H = 1100，0010H

H（3）＝ASCII（C）＋1000，0000H = 1100，0011H

把节点按地址存放在内存空间中相应位置就形成了哈希表，用哈希函数构造表的过程，即通过哈希函数实现由关键码到存储地址的转换过程叫哈希造表或地址散列。以同样的函数用关键码对哈希表进行节点检索，称为哈希检索。显然，元素的散列存储是一种新的存储结构，完全不同于链表或者是顺序存储结构。

哈希检索的实质是构造哈希函数，哈希函数的实质是实现关键码到地址的转换，即把关键码空间映射成地址空间。因为关键码空间远大于哈希表地址空间，所以会产生不同的关键码映射到同一哈希地址上的现象，我们称为‘地址冲突’。比如上例文件中另有一些节点 R_4，R_5，R_6关键码是 A1，B1，C1，则用该哈希函数散列后生成的地址如表 2.1 所示。关键码 ABCD 不等于 A1，但经过地址散列后它们具有相同的哈希地址，即‘地址冲突’。我们定义，具有相同函数值的关键码对该哈希函数来说是同义词，因此我们要求运用哈希检索时有：

- 由给定关键码集合构造计算简便、且地址散列均匀的哈希函数以减少冲突。
- 拟定处理冲突的办法。

表 2.1

关键码	哈希函数	哈希地址
ABCD	ASCII(首字符)+常数	1100，0001H
BCDE	ASCII(首字符)+常数	1100，0010H
CDEF	ASCII(首字符)+常数	1100，0011H
A1	ASCII(首字符)+常数	1100，0001H
B1	ASCII(首字符)+常数	1100，0010H
C1	ASCII(首字符)+常数	1100，0011H

所谓地址散列均匀是指构造的哈希函数应尽可能的与关键码的所有部分都产生相关，因而可以最大程度的反映不同关键码的差异，比如前例中的 A1、B1、C1，如果我们不仅考虑只取首位字符的 ASCII 码，而是取关键码各字母的 ASCIIA 值的平方和作为哈希函数，就可以减少冲突。至于设定处理冲突的办法，是因为一般说冲突是不可避免的，我们需要寻求一种有效处理它的手段。

2.4.1 哈希函数

一般说关键码分布于一个相对大的范围而哈希表的大小却是有限的，既是如此，我们也不能保证根据哈希函数得到的散列地址可以均匀的填满哈希表的每一个位置（槽），一个好的哈希函数应该让大部分的元素记录可以存储在根据散列地址组织的（存储结构）槽位中，或者说至少表的一半是满的，而产生的地址冲突可以由处理冲突的方法解决。

哈希函数的选择取决于具体应用条件下的关键码分布状况，如果预先知道其分布概率就可以设计比较好的哈希函数，否则比较困难。

例 2.3 下面这个函数把一个整数散列到表长为 16 的哈希表中。

```
int hx(int x){  return(x % 16);}
```

对于 2 字节二进制串来说，函数返回值仅由其最低四个比特位决定，分布应该很差，如二进制串 1000 0000 0000 1111（32783），对 16 取模运算，就是将其右移 4 位（空出的高位填零补进）：0000 1000 0000 0000（2048），余数是 1111，而二进制串 0000 0000 1111 1111（255）对 16 取模运算结果是 0000 0000 0000 1111，余数也是 1111。

例 2.4 下面是用于长度限制在 10 个大写英文字母的字符串哈希函数

```
int hx(ch x[10])
{
    int i,sum;
    for(sum=0;i=0;i<10;i++)sum+=(int)x[i];
    return(sum % M);
}
```

该函数用输入字符串 10 个字符的 ASCIIM 之和，再取 M 的模，因为字母的 ASCII 码值分布 65～90 之间，所以 10 个字符之和在 650～900 之间，显然，如果表长 M 在 100 以内其散列地址分布的比较好（因为（650%100）=65，（900%100）=0，大致填满表的一半左右），如果 M 在 1000 左右其散列地址会相当的差。下面列出几种常用的哈希函数方法。

- 直接定址法。直接取关键码或关键码的某个线性函数值为哈希地址：

$$H_i = a \cdot key + b \qquad (a,b = const)$$

- 除留余数法。取关键码被某个不大于哈希表长 m 的数 p 除后所得的余数为哈希地址。

$$H_i = key \mod p \qquad \text{p是质数且小于表长}M$$

- 平方取中法。取关键码平方后的中间几位为哈希地址。
- 随机数法。选择一个随机函数，取关键码的随机函数值为它的哈希地址。

实际工作中我们主要考虑如下因素为选择哈希函数的条件：

1・哈希函数的计算复杂性

2・哈希表大小

3・关键码分布情况

4・检索概率

2.4.2 闭地址散列

设哈希地址集为 0～M-1，冲突是在由关键码得到哈希地址为 i，而此地址上已存放有其它节点元素时发生，处理冲突就是为该关键码的节点寻找另一个空槽（哈希地址），称之为再探测，探测方法有多种，在再探测过程中仍然可能会遇见槽位不空的情况，于是在探测处理冲突可能会得到一个哈希地址序列，即多次冲突处理后才有可能找到空地址。

2.4.2.1 线性探测法和基本聚集问题

这是基本的地址冲突处理方法，思想很简单，如果通过哈希函数得到基地址 i，发现该槽位不空，那么就由紧邻的地址开始，线性遍历哈希表内所有的地址，并将元素放到所发现的第一个空槽内，线性探测函数是：

$$H_i = (H(key) + i) \mod M \qquad i = 1,2,...,M-1 \qquad (2.1)$$

这里 H_i 是由线性探测产生的哈希地址序列，H(key)是哈希函数，M 是表长，i 是增量序列，称为线性探测再散列。即当哈希函数计算得地址 i 时如果位置 i 上已有节点存在，则继续探测 i+1 地址是否为空，并且在不空时持续探测下去。 设 H(r)是哈希函数，哈希表 Table[M]初始化为 NULL，关键码 r 不能为 NULL，基于线性探测再散列方法的插入程序如下：

程序 2.6

```
void hashlnsert(int r)
{
int i,h0;
int pos=h0=H(r);
for(i=1;Table[pos] !=NULL;i++ ){
    if(Table[pos])==r)return（-1）;//如果和后一条语句交换顺序则可能在基位置重复插入 r;
   pos=(h0+i) %M;
   }
Table[pos]=r;
}
```

该程序假定在插入或检索过程中，哈希表至少有一个槽位为空，否则会出现无限循环过程。从哈希表中检索一个元素是否在表内的过程，和插入元素的过程所使用的方法必须一样，从而可以准确的找到不在基位置上的元素，返回其在哈希表内的位置。若返回值为空则表示检索失败。

程序 2.7

```
void hashlnsert(int r)
{
int i,h0;
int pos=h0=H(r);
for(i=1;(Table[pos] !=key)&&( Table[pos] !=NULL);i++ )pos=(h0+i) %M;
if(Table[pos]==key)return(pos);
else return(NULL);
}
```

例 2.6 线性探测产生的基本聚集问题。哈希表长 M=11，哈希函数采用除模取余，且 p=M。处理冲突的策略是线性探测，再探测函数 $H_i = (H(key)+i) \mod 11$，i=1,2,..,M-1，输入元素序列是{9874，2009，1001，9537，3016，9875}，得到哈希表如下图 2.8（a）所示。

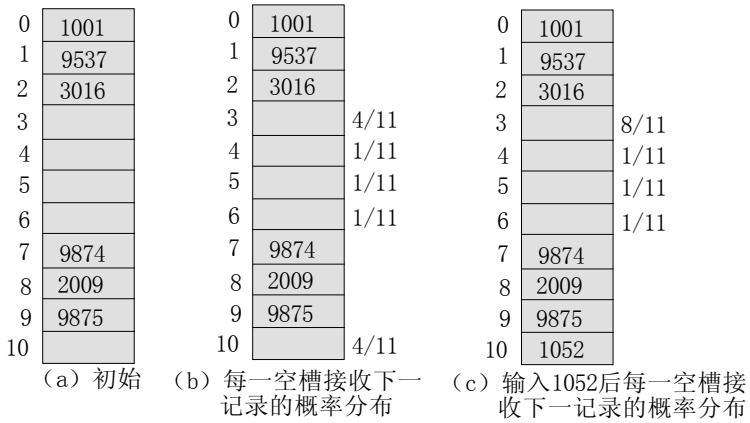


图 2.8 线性探测产生的聚集现象

问（1）表中剩余的每一空槽接收下一个记录的概率；

问（2）继续输入关键字值 1052 后，表中剩余的每一空槽接收下一个记录的概率；

问（3）根据以上计算，说明随着输入记录的增加，线性探测法处理冲突所产生的问题。

解（1）：理想情况下表中剩余的每一空槽接收下一个记录的概率应该是等分的，但是如图 2.8（a）的初始分布之后，假设新一个输入元素基地址是 0，则它经过线性再探测一定会被分配到空槽 3 的位置，同样，如果新元素基地址是 1 或者 2，也会被分配到空槽 3 的位置，考虑到新元素基地址也可能就是 3，所以，空槽 3 接收新元素的可能性是$\frac{4}{11}$，空槽 4，5，6 因为在其前的槽位是空，所以接收下一个新元素的时候不会产生线性探测问题，其接收概率为$\frac{1}{11}$，而槽位 10 的情况与槽位 3 完全相同，它有可能接收前面非空槽位 7，8，9 的线性探测结果，所以接收新元素的可能性也是$\frac{4}{11}$，因此，表中剩余的每一空槽接收下一个记录的概率分布如图 2.8（b）所示。

解（2）：继续输入 1052，哈希地址是（1052 % 11）=7，经过线性再探测它被放置到槽位 10，此时，由于槽位 3 前面连续 7 个槽位全部非空，所有基地址散列到这 7 个地址上的元素都有可能被放置到槽位 3，于是，经过元素 1052 的进入及线性再探测过程，槽位 3 现在接收下一个新元素的概率已经变成$\frac{8}{11}$，现在表中剩余每一空槽接收下一个记录的概率分布如图 2.8（c）所示。

解（3）：随着输入记录的增加，线性探测法处理冲突所产生的问题是记录在表内的分布出现聚集倾向，称为基本聚集。随着聚集程度的增加，它将导致新元素进入或者检索过程中出现很长的探测序列，严重的降低哈希表使用效率。

2.4.2.2 删除操作造成检索链的中断问题

如果假设哈希表至少有一个槽位是空的（实际应用中一般不会占满整个哈系表空间），

则程序 2.7 的线性再探测过程是以或者找到一个匹配检索关键码的节点，或者找到一个空槽为结束标志，这会带来所谓的删除操作造成检索链中断问题。

设已输入节点关键码序列是{K_1，K_2，…，K_i，K_{i+1}，…}，通过哈希函数散列得到哈希表如图 2.9（a）所示。我们进行如下操作：

① 现在输入关键码 K_j，设它的基地址 H（K_j）=i，因非空而产生冲突，线性再探测序列从 i+1 开始遍历哈希表寻找一个空槽，假定地址为 i 至 j-1 之间的槽位均非空，但槽位 j 空，于是 K_i 被放到表中的空槽 j，如图 2.9（b）所示；

② 现在将 K_{i+1} 节点关键码删除，并重新设置槽位 i+1 为空；

③ 对哈希表检索 K_j 节点关键码，程序首先比较基地址 i 槽位，非空，再探测地址 j，空槽，于是检索结果是关键码为 K_j 的节点不在哈希表中。

造成这种情况的原因是由于删除操作造成了检索链中断，程序发现一个空槽后就会判别已经达到检索链末端。解决的办法是设置一个标记，表明该位置曾经有元素插入过，这个标记我们称之为墓碑，它的设置使得删除操作既不会影响该单元的继续使用，因为插入操作的时候，如果是墓碑标记就可以直接覆盖，不会使槽位浪费；同时，墓碑也避免了因删除操作中断了再探测的检索过程问题，因此，程序 2.7 的 for 语句中需要增加一个墓碑判断条件。

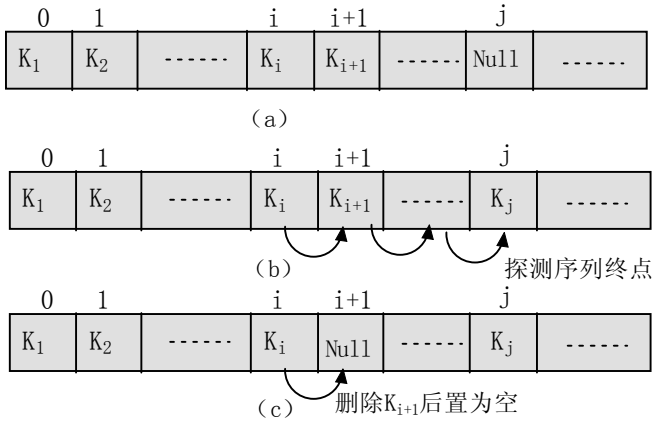


图 2.9 删除操作造成检索链中断

删除操作带来的检索链中断问题无论对哪种再探测方法都是同样的。

2.4.2.3 随机探测法

解决线性探测造成的基本聚集问题的原则，是让表中每一空槽接收下一记录的概率应尽可能的相等，方法之一是采用随机探测序列，产生的再探测地址是从哈希表剩余的空槽中随机选取的，其在探测函数定义如下：

$$H_i = (H(key) + d_i) \mod M \qquad i = 1,2,...,k \qquad (2.2)$$

d_i 为一组随机数序列。

例 2.7 随机探测 顺序输入一组关键字（K_1，K_2，K_3，K_4，K_5，K_6，K_7）得到的哈希地址是（2，

25，4，2，1，1，2)，假设哈希表长 29（表长是一个素数对提高哈希表性能很重要），哈希函数是除模取余，模长 p=M,用随机探测方法确定冲突后地址的函数是：

$$H_i = (H(key) + d_i) \mod 29 \qquad i = 1,2,...,k$$

随机步长序列 d_i 是 23，2，19，14，…，求：

（1）画出该组输入下的哈希表，并写出产生冲突后地址探测函数的求值过程。

（2）指出对该哈希表进行哪类操作可能产生问题，说明原因并提出解决办法。

解（1）

K_1=2，K_2=25，K_3=4，均为空槽，基地址一次散列成功；

K_4：a_1=((2+23)%29)=25，a_2=((2+2)%29)=4，a_3=（(2+19)%29）=21，3 次探测成功；

K_5=1，空槽，基地址一次散列成功；

K_6：a_1=((1+23)%29)=24；

K_7：a_1=((2+23)% 29)=25，a_2=((2+2)%29)=4，a_3=((2+19)%29)=21，a_4=((2+14)%29)=16；

所得哈希表是：

1	2	3	4	……	16	……	21	22	23	24	25		29
K_5	K_1		K_3	……	K_7	……	K_4			K_6	K_2	…	

解（2）

如果对哈希表进行删除操作有可能会产生检索链中断问题，因为删除后该单元位置为空，使在该冲突序列链上的后续记录探查不能进行，比如删除关键字 K_4 后，位置 21 为空，如果要检索 K_7 就不可能，因为在插入 K_7 的探测序列中达到过槽位 21，由于槽位 21 不空继续探测到槽位 16（注意，检索时使用的随机探测序列就是插入过程使用的随机序列），当 K_4 删除后由于槽位 21 变为空，用关键码 K_7 检索并用探测序列达到槽位 21 时，由于其为空程序会判别是 K_7 不在哈希表内，检索失败，因此，删除造成了 K_7 检索链中断。

2.4.2.4 平方探测法

设哈希函数是 $H(key)$，再探测函数可以写成 $P(H,i) = f$，基于平方探测再散列的 f 描述就是 $P(H,i) = i^2$，即：

$$H_i = (H(key) + i^2) \mod M \qquad i = 1,2,...,k \qquad (2.3)$$

显然，可以把线性再探测和随机再探测函数统一描述如下：

$$P(H,i) = i, \quad P(H,i) = array[i]$$

其中数组 array[]长度为 M-1，存储的是 1～M-1 的随机序列。

例 2.8 平方探测 设哈希表长 M=101，输入关键码 K_1，K_2 的哈希地址是 $H(K_1)$=30，$H(K_2)$=29，根据式（2.3）分别写出它们探测序列的前 4 个地址是：

K_1 探测序列={30，31，34，39，…}

K_2 探测序列={29，30，33，38，…}

显然，具有不同基位置的两个关键码在散列过程中，使用平方探测可以很快分开它们的探测序列。但是，对于再探测序列来说，我们希望尽可能多的探测到哈希表的每一个槽位，比如线性探测，它能遍历整个哈希表去搜寻每一个槽位是否为空，但是容易产生聚集，平方探测显然会遗漏一些槽位，因为它是以跳跃方式进行再探测过程，不过我们可以证明，它至少能探查到哈希表一半以上的地址。

例 2.9 设 2 次探测序列是 $H_i = (H(key) + i^2) \mod M, i = 1,2,...,k$，M 是哈希表长（M 为质数），请证明，2 次探测序列至少可以访问到表中的一半地址。

解：

只需证明当探测序列产生地址冲突时，序列下标大于等于 $\frac{M}{2}$。

设 $i \neq j$ 而 $d_i = d_j$，根据同余的定义有：

$i^2 (\mathrm{mod}\, M) \equiv j^2 (\mathrm{mod}\, M)$

$\therefore (i^2 - j^2) \mathrm{mod}\, M \equiv 0$，或：$i^2 - j^2 \equiv 0 (\mathrm{mod}\, M)$

即 $(i^2 - j^2)$ 能被 M 除尽，因式分解后有：

$(i + j)(i - j) \equiv 0 (\mathrm{mod}\, M)$

因为 M 为质数且 $i, j < M$，所以 $(i - j) \neq 0 (\mathrm{mod}\, M)$，因而：

$(i + j) \equiv 0 (\mathrm{mod}\, M)$

$i + j = cM$

c 为整数，所以

i 或 $j \geq \frac{M}{2}$，证毕。

这里，使用了数论中同余概念与同余定理，描述如下：

同余定义：若 a 和 b 为整数，而 m 为正整数，如果 m 整除 a-b，就说 a 与 b 模 m 同余，记为：

$$a \equiv b (\mathrm{mod}\, m)$$

可以证明 $a \equiv b (\mathrm{mod}\, m)$，当且仅当 $a (\mathrm{mod}\, m) = b (\mathrm{mod}\, m)$ 时成立。根据定义，如果整数 a 和 b 模 m 同余，则 a-b 被 m 整除，显然，a-b-0 也被 m 整除，所以，如果 $a \equiv b (\mathrm{mod}\, m)$ 成立，则必有 $a - b \equiv 0 (\mathrm{mod}\, m)$ 成立。

同余定理：令 m 为正整数，整数 a 和 b 模 m 同余的充分必要条件是存在整数 k 使得：

$$a = b + km$$

2.4.2.5 二次聚集问题与双散列探测方法

我们注意到，虽然随机探测和平方探测能解决基本聚集问题，但是它们产生的探测序列是基位置的函数，如果构造的哈希函数让不同的关键码具有相同的基地址，那么它们就有相同的探测序列而无法分开。由于哈希函数散列到一个特定基位置导致的地址聚集，我们称之为二次聚集。

为避免二次聚集我们需要让探测序列是原来关键码值的函数，而不是基位置的函数，一

种简单的处理方法是仍然采用线性探测方法，但是我们设计有两个哈希函数 $H_1(key)$和 $H_2(key)$，它们都以关键码为自变量散列地址，其中，$H_1(key)$产生一个 0 到 M-1 之间的散列地址，而 $H_2(key)$ 产生一个 1 到 M-1 之间、并且是和 M 互素的数做为地址补偿，双散列探测序列是：

$$H_i = (H_1(key) + iH_2(key)) \mod M \qquad i = 1,2,...,k \qquad (2.4)$$

例 2.10 双散列方法 仍设哈希表长 M=101，输入关键码 K_1，K_2和 K_3，它们的哈希地址分别是 $H_1(K_1)=30$，$H_1(K_2)=28$，$H_1(K_3)=30$，$H_2(K_1)=2$，$H_2(K_2)=5$，$H_2(K_3)=5$，根据式（2.4）分别写出它们探测序列的前 4 个地址是：

K_1探测序列={30，32，34，36，...}

K_2探测序列={28，33，38，43，...}

K_3探测序列={30，35，40，45，...}

实际上，$H_1(key)$和 $H_2(key)$仍有可能产生相同的探测序列，比如 $H_1(K_4)=28$，$H_2(K_4)=2$，其探测序列与 K_1相同（注意，所有探测序列都是从基位置以后开始的）：

K_4探测序列={28，30，32，34，36，...}

我们可以进一步的考虑让随机探测与双散列方法结合，让 i 成为一个随机序列中选取的随机数。

2.4.3 开地址散列

设哈希函数产生的地址集在 0～M-1 区间，则可以设立指针向量组 ARRAY[M]，其每个分量初值为空，我们将具有哈希地址 j 上的同义词所包含的关键码节点存储在以向量组第 j 个分量为头指针的同一个线性链表内，存储按关键码有序。即，所有产生冲突的元素都被放到一个链表内，于是，哈希散列过程转换为对链表的操作过程。

例 2.11：设输入关键码为（19，14，23，1，68，20，84，27，55，11，10，79），其表长为 13，选择哈希函数是 $H_i = key \mod 13$。则链地址法解决冲突后得到的哈希表如图 2.10 所示。

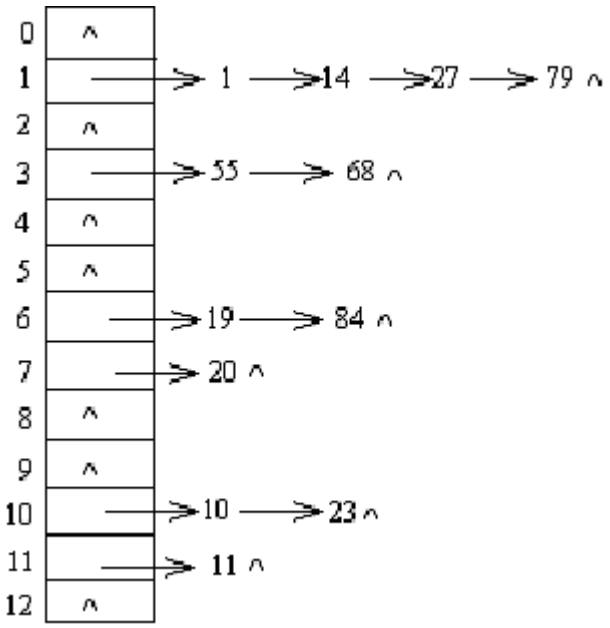


图 2.10 链地址法散列得到的哈希表

2.4.4 哈希表检索效率

散列存储结构是通过哈希函数运算得到元素散列地址的，但由于冲突的存在使得在检索过程中它仍然是一个给定值和关键码的比较过程，因此平均检索长度仍是哈希检索的效率量度，而检索过程中给定值和关键码的比较个数取决于哈希函数质量、处理冲突的方法以及哈希表的装填因子。

当散列表为空的时候，第一条纪录直接插入到其基位置上，随着存储纪录的不断增加，把记录插入到基位置上的可能性也越来越小，如果纪录被散列到一个基位置而该槽位已经不空，则探测序列必须在表内搜索到另一空槽才行，换句话说散列过程增加了比较环节，可以预计，随着纪录数的增加，越来越多的新纪录有可能被放到远离基位置的空槽内，即探测序列越来越长而比较次数越来越多，因此，哈希检索效率预期是表填充程度的一个函数，设表长为 M，已存储记录数为 N，定义装填因子是$\alpha = \frac{N}{M}$，即，装填因子＝表中填入节点个数/哈希表长度。

可以认为，新纪录插入的时候基位置被占用的可能性就是α，假定可以不考虑任何聚集问题，而发现基位置和探测序列下一个位置均非空的可能性是：

$$\frac{N(N-1)}{M(M-1)}$$

此时探测序列长度为 2，档探测序列达到 i+1 时，表明在第 i 次探测仍然发生冲突，其可能性是：

$$\frac{N(N-1)...(N-i+1)}{M(M-1)...(M-i+1)}$$

当 N 和 M 都很大时近似有 $(\frac{N}{M})^i$，所以，预期探测次数的期望值是 1 加上第 i 次探测产生冲突的概率之和，约为：

$$1+\sum_{i=1}^{\infty}(\frac{N}{M})^i=\frac{1}{1-\alpha}$$

即一次检索成功的代价与哈希表为空时相同，随着纪录数目的增加，平均检索长度（或者说插入代价的均值）是装填因子从零到当前 α 的累积：

$$\frac{1}{\alpha}\int_0^{\alpha}\frac{1}{1-x}dx=\frac{1}{\alpha}\ln(\frac{1}{1-\alpha})$$

无论从哪方面看，哈希检索效率远高于 O（log n），随着 α 的增加效率会降低，当 α 足够小的时候，效率仍然可以小于 2，当 α 接近 50%的时候，效率接近 2，因此，我们要求哈希表工作的时候应该在半满状态，太小则表的空间浪费，太大则检索效率降低过多。

例 2.12：设输入关键码为（13，29，1，23，44，55，20，84，27，68，11，10，79，14），选择装填因子 α =0.75，哈希表长 M=19，哈希函数采用除模余数法，取模 p=17，求：

（1） 线性探测产生的哈希表；

（2） 随机探测产生的哈希表，随机序列是{3，16，55，44，…}；

（3） 平方探测产生的哈希表；

解 1：线性探测法，注意线性探测中 a_j=(H(Key)+i)%M，使用的模是表长度 M，而哈希散列使用的模是 p=17。

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
68	1		20	55		23				44	27	29	13	11	10	84	79	14

27：a_1=27%17=10，a_2=（H(27)+1）%19=11；

11：a_1=11%17=11，a_2=（11+1）%19=12，a_3=（11+2）%19=13，a_4=(11+3)%19=14；

10：a_1=10%17=10，a_2=（10+1）%19=11，a_3=（10+2）%19=12，a_4=(10+3)%19=13，

a_5=（10+4）%19=14，a_6=（10+5）%19=15；

79：a_1=79%17=11，a_2=（79+1）%19=12，a_3=（79+2）%19=13，a_4=(79+3)%19=14，

a_5=（79+4）%19=15，a_6=（79+5）%19=16，a_7=（79+6）%19=17；

14：a_1=14%17=14，a_2=（14+1）%19=15，a_3=（14+2）%19=16，a_4=(14+3)%19=17，

a_5=（14+4）%19=18；

解 2：随机探测法：已知 a_j=(H(K)+d_j)%19，随机步长序列 d_j 是{3，16，55，44，…}，则：

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
68	1		20	55	27	23	10		79	44		29	13	11		84	14	

27：a_1=27%17=10，a_2=（27+3）%19=13，a_3=（27+16）%19=5；

11：a_1=11%17=11；

10：a_1=10%17=10，a_2=（10+3）%19=13，a_3=（10+16）%19=7；

79：a_1=79%17=11，a_2=（79+3）%19=6，a_3=（79+16）%19=0，a_4=(79+55)%19=1，

a_5=（79+44）%19=9；

14：a_1=14%17=14，a_2=（14+3）%19=17；

解 3：平方探测法，已知 H_i=(H(key)+i^2)%19：

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
68	1		20	55		23	79		27	44	11	29	13	10	14	84		

27：a_1=27%17=10，a_2=（27+1）%19=9；

11：a_1=11%17=11；

10：a_1=10%17=10，a_2=（10+1）%19=11，a_3=（10+4）%19=14；

79：a_1=79%17=11，a_2=（79+1）%19=6，a_3=（79+4）%19=7；

14：a_1=14%17=14，a_2=（14+1）%19=15；

例 2.13 求成功检索图 2.10 哈希表的平均检索长度。

解：比较次数为 1 的节点有 6，比较次数为 2 的节点有 4，比较次数为 3 和 4 的节点分别只有 1 个，所以，平均比较次数是：

$$ASL = \frac{1\times 6 + 2\times 4 + 3 + 4}{12} = \frac{21}{12}$$

第三章 排序

排序是程序设计中的重要内容，它的功能是按元素的关键码把元素集合排成一个关键码有序序列。排序有内部排序与外部排序之分，内部指计算机内部存储器，内部排序是元素待排序列在内存中，当内存不足以容下所有元素集合时，我们把它存储在外部存储器上进行排序运算，称为外部排序。我们首先给出排序定义，进而再讨论内部排序问题。

设有 n 个元素序列{R_1，R_2，...，R_n}，其相应的关键码序列是{K_1，K_2，...，K_n}，需确定 1，2，•••，n 的一种排列 P_1，P_2，•••，P_n，使其相应的关键码满足如下非递减（或非递增）关系：

$$K_{p1} \leqslant K_{p2} \leqslant K_{p3} \leqslant ，\ldots，\ \leqslant K_{pn}$$

即序列按{R_{p1}，R_{p2}，R_{p3}，...，R_{pn}}成关键码有序序列，这种操作的过程称为排序。

当 K_1，K_2，...，K_n 是元素主关键码时，即任何不同的元素有不同的关键码，此排序结果是唯一的，上面的等号不成立。当 K_1，K_2，...，K_n 是元素次主关键码时，排序结果不唯一，此时涉及到排序稳定性问题，我们定义：

设 $K_i=K_j$， $1 \leqslant i$ ，$j \leqslant n$，且 i 不等于 j

若排序前 R_i 在 R_j 之前（$i<j$），而排序后仍有 $R_{pi}<R_{pj}$，即具有相同关键码的元素其在序列中的相对顺序在排序前后不发生变化，则称此排序方法是稳定的。反之，若排序改变了 R_{pi}，R_{pj} 的相对顺序，则称此排序方法是不稳定的。

在排序过程中主要有两种运算，即关键码的比较运算和元素位置的交换运算，我们以此衡量排序算法的效率。对于简单的排序算法，其时间复杂度大约是 $O(n^2)$，对于快速排序算法其时间复杂度是 $O(n\log_2 n)$数量级的。我们要求必须掌握几种基本方法是线性插入排序，快速排序，堆排序，归并排序等。下面从最基本的直接插入排序方法开始，进而再研究快速排序方法的程序设计问题。

3.1 交换排序方法

3.1.1 直接插入排序

直接插入排序是在插入第 i 个元素时，假设序列的前 i-1 个元素 R_1，R_2，...，R_{i-1} 是已排好序的，我们用 K_i 与 K_1，K_2，...，K_{i-1} 依次相比较，找出 K_i 应插入的位置将其插入。原位置上的元素顺序向后推移一位，存储结构采用顺序存储形式。为了在检索插入位置过程中避免数组下界溢出，设置了一个监视哨在 R[0]处。

- 算法 3.1

1・待排序的 n 个元素在数组 ARRAY[n+1]中，按递增排序。

```
2・for(i=2; i<=n; i++){ /*第一个元素已经有序*/
        array[0]=array[i];        /*监视哨*/
        j=i-1;
        while(array[0].key<array[j].key){
                array[j+1]=array[j]; /*顺序向后移动一位*/
                j--;
                }/*循环中止时 j+1 指向第 i 个元素应插入的位置*/
        array[j+1]=array[0];
        }
```

程序本身并不复杂，要注意监视哨的作用，当 $R_i<R_1$ 时程序也能正常中止循环，把 R_i 插入到 R_1 位置。一个具体的线性排序例子过程见图 3.1 所示。

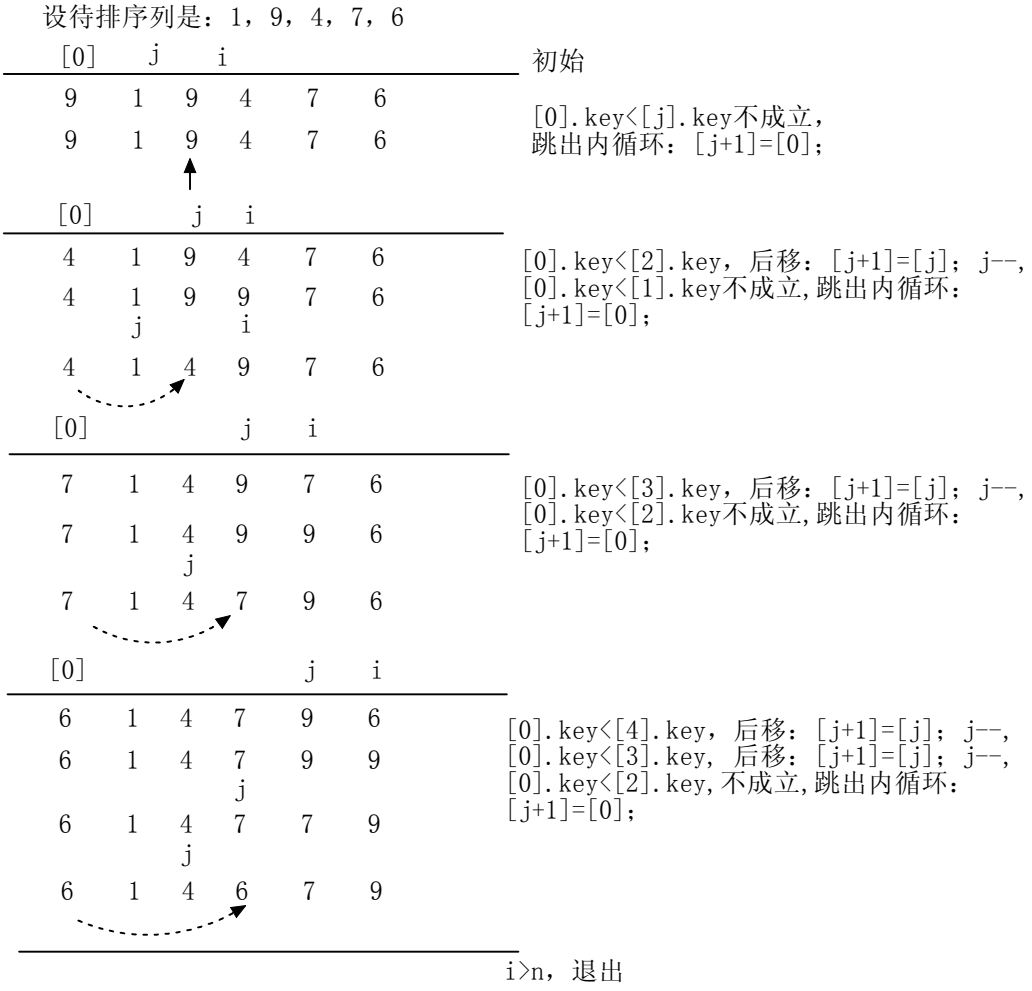


图 3.1 线性排序过程

- 效率分析

直观上看，其两重循环最大都是 n，因此其时间复杂度是 $O(n^2)$。而在一个元素 i 排序

过程中,要在已排好序的 i-1 个元素中插入第 i 个元素其比较次数 C_i 最多是 i 次,此时 $R_i<R_1$ 被插到第一个元素位置。比较次数最少是 1 次，此时 $R_i \geqslant R_{i-1}$，位置没有移动。因此 n 次循环的最小比较次数：

$$C_{min}=n-1$$

最大比较次数：

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

而一次插入搜索过程中，为插入元素 i 所需移动次数其最大是 C_{i+1}，最小是 2 次（包括 array[0]的 1 次移动)。那么，n 个元素排序时，其 n-1 个元素需 2(n-1)次，所以最小移动次数：

$$M_{min}=2\times(n-1)$$

最大移动次数：

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

直接插入排序也可以考虑用链表来实现，此时没有元素移动问题，但最大、最小比较次数一样，另外，直接插入排序的一种改进是二分法插入排序，即检索插入位置时用二分法进行，其检索效率有所提高，但不能改变元素的移动次数，所以其平均时间复杂度不变，且只适用于顺序存储结构。

算法判别条件 while(array[0].key<array[j].key)是当前 j 指向关键码若小于排序元素 array[0]关键码，则序列顺序后移，找到有序的位置后交换元素；若等于或大于就不发生交换，因此，具有相同关键码的元素在排序前、后的相对位置不会发生变化，所以它是一个稳定的排序方法。

3.1.2 冒泡排序

冒泡（bubble sort）的意思是每一趟排序将数组内一个具有最小关键码的元素排出到数组顶部，算法有一个双重循环，其中内循环从数组底部开始比较相邻元素关键码大小，位居小者向上交换，并在内循环中通过两两交换将最小元素者直接排出到顶部，此时外循环减一指向数组顶部减一位置，继续内循环过程，将数组内具有次最小关键码的元素排出至数组顶部减一位置，如此直至循环结束，每次循环长度比前次减一，最终结果是一个递增排序的数组。图 3.2 是冒泡排序示意，算法直接见程序 3.1。

程序 3.1 冒泡排序

```
void bubsort(struct node *p,int n)
{
struct node s;
for(int i=0;i<n-1;i++){
    int flag=0;
    fot(int j=n-1;j>i;j--)if(*(p+j).key<*(p+j-1).key){
                                        s=*(p+j);
                                        *(p+j)=*(p+j-1);
                                        *(p+j-1)=s;
                                        flag=1;
                                        }
    if(flag==0)retrun;
    }
}
```

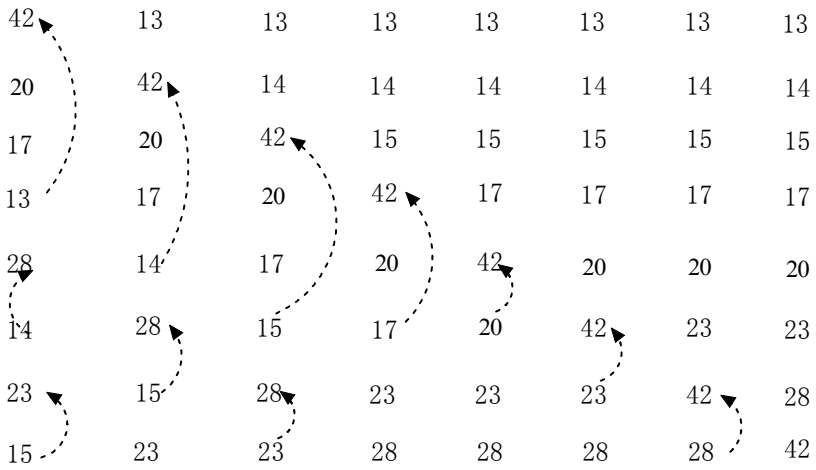


图 3.2 冒泡排序过程

程序 3.1 如果没有 flag 这个标志位，那么即使数组已经有序，程序也会继续双重循环直至结束为止。另外，我们注意到，如果*（p+j）.key>=*（p+j）.key，相邻元素不发生交换，所以冒泡排序是一个稳定的排序方法。

冒泡排序是一个双重循环过程，所以其比较次数是：

$$\sum_{i=1}^{n} i = O(n^2)$$

其最佳、平均和最差情况基本是相同的。

3.1.3 选择排序

选择排序（selection sort）每次寻找待排序元素中最小的排序码，并与其最终排序位置上的元素一次交换到位，避免冒泡排序算法有元素在交换过程中不断变位的问题，比如，首先选择 n 个元素的最小排序码，将其与排序数组[0]位置的元素交换，然后是选择剩余 n-1 个元素的最小排序码，将其与排序数组[1]位置上的元素交换，即，第 i 次排序过程是选择剩余 n-i 个元素的最小排序码，并与排序数组[i]位置的元素交换，它的特点是 n 个元素排序最多只有 n-1 次交换。图 3.3 是选择排序过程示意，程序 3.2 是选择排序函数。

程序 3.2 选择排序

```
void selsort(struct node *p,int n)
{
struct node s;
for(int i=0;i<n-1;i++){
    int lowindex=i;
    for(int j=n-1;j>i;j--)if(p[j].key<p[lowindex].key)lowindex=j;
    s=p[i];
    p[i]=p[lowindex];
    p[lowindex]=s;
    }
}
```

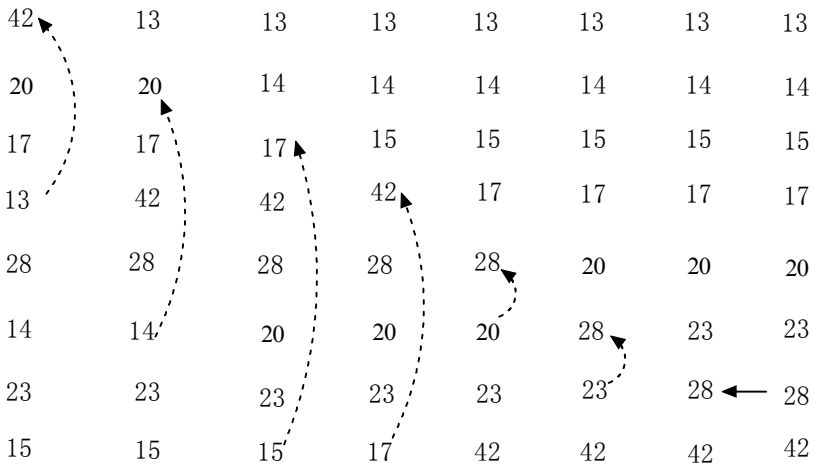


图 3.3 选择排序过程

选择排序实际上仍然是冒泡排序，程序记住最小排序元素的位置，并一次交换到位，它的比较次数仍然是 $O(n^2)$ 量级，但交换次数最多只有 n-1 次。如果*（p+j）.key>=*（lowindex）.key，不发生交换，所以选择排序是一个稳定的排序方法。

3.1.4 树型选择排序

直接插入排序的问题在于为了从 n 个排序码中找出最小的排序码，需要比较 n-1 次，然后又从剩下的 n-1 个排序码中比较 n-2 次，而事实上，这 n-2 次比较中有多个排序码已经在前面比较过大小，只是没有保留结果而已，以至有多次比较重复进行，造成效率下降。如果我们这样考虑，设 n 个排序元素为叶子，第一步是将相邻的叶子两两比较，取出较小排序码者作为子树的根，共有$\left\lfloor \frac{n}{2} \right\rfloor$棵子树，然后将这$\left\lfloor \frac{n}{2} \right\rfloor$棵子树的根再次按相邻顺序两两比较，取出较小排序码者作为生长一层后的子树根，共有$\left\lfloor \frac{n}{4} \right\rfloor$棵，循环反复直至排出最小排序码成为排序树的树根为止，我们将树根移至另一个数组，并且将叶子数组中最小排序码标记为无穷大，然后继续从剩余的 n-1 个叶子中选择次最小排序码，重复上述步骤的过程，实际上只需要修改从树根到刚刚标记为无穷大的叶子节点这一条路径上的各节点的值，而不用比较其它的节点，除去第一次以外，等于每次寻找排序码的过程是走过深度为$\log_2 n$的二叉树，即只需要比较$\log_2 n$次，我们称之为树型选择排序。图 3.4 是树型排序的过程前三次循环示意，排序数组为{72，73，71，23，94，16，5，68}，图 3.4（a）是构造选择排序二叉树，图 3.4（b）是标记最小排序码后调整得到的次最小排序码，图 3.4（c）是又一次循环过程。

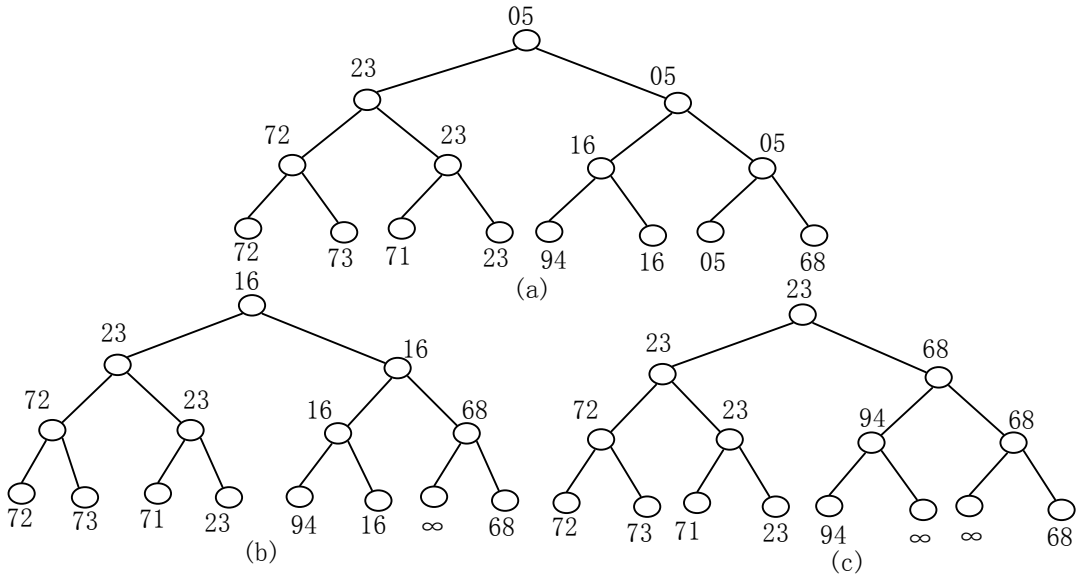


图 3.4 树型选择排序过程

因为第一次构造这棵二叉树需要比较 n-1 次才能找到最小的排序码，以后每次在这棵树上检索最小排序码需要比较$\log_2 n$，共有 n-1 次检索，所以，树型选择排序总的比较次数为：

$$(n-1)+(n-1)\log_2 n$$

时间复杂度是$O(n\log_2 n)$。

3.2　Shell 排序

shell 排序也称之为缩小增量法（diminishing increment sort）。它不是根据相邻元素的大小进行比较和交换，而是把总长度为 n 的待排序序列以步长$d_i=\frac{n}{2^i}$进行分割，间隔为 d_i的元素构成一组，组内用直接插入、或者是选择插入法排序。下标 i 是第 i 次分组的间隔，i=1，2，…。随着间隔 d_i的不断缩小，组内元素逐步地增多，但因为是在 d_{i-1}的有序组内基础上新增待排元素，所以比较容易排序。

若 n 不是 2 的整数冥，不妨对排序数组长度补零到整数冥的长度为止。

图 3.5 显示了排序数组长度为 8 的排序过程。程序 3.3 是 shell 排序实现。

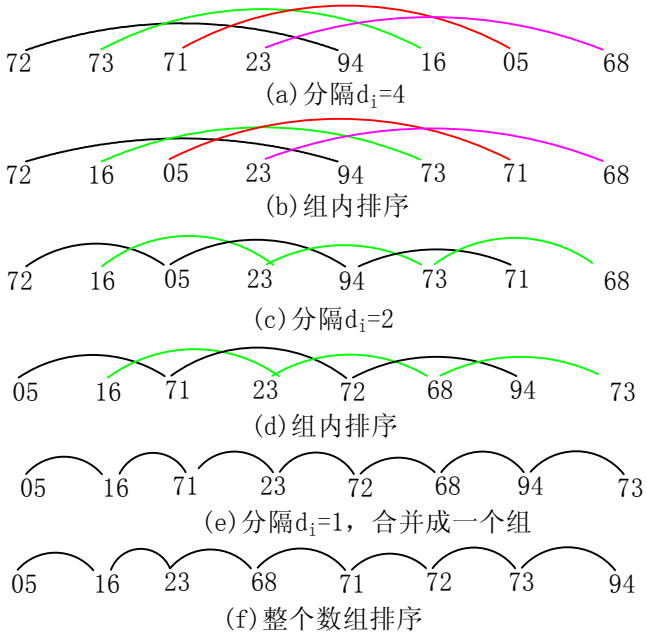


图 3.5 shell 排序过程

程序 3.3 Shell 排序

```
void shellsort(struct node *p,int n)
{
    for(int i=n/2;i>2;i/=2)
        for(int j=0;j<i;j++)inssort2((p+j),n-j,i);
    inssort2(p,n,1);
}
void inssort2(struct node *p,int n,int incr)
{
    for(int i=incr;i<n;i+=incr)
```

```
        for(int j=i;(j>=incr)&&(p[j].key<p[j-incr].key);j-=incr)
                swap((p+j),(p+j-incr));
}
```

3.3 快速排序

对已知的元素序列排序，在一般的排序方法中，一直到最后一个元素排好序之前，所有元素在文件中（排序表）中的位置都有可能移动。所以，一个元素在文件中的位置是随着排序过程而不断移动的，造成了不必要的时间浪费。快速排序基于这样一种思想，第 i 次排序不是确定排序表中前 i-1 个元素的相对顺序，而是只确定第 i 个元素在排序表中的最终位置。方法是每次排序后形成前后两个子序列，关键码小于或等于该元素关键码的所有元素均处于它左侧，构成前子序列；关键码大于该元素关键码的所有元素均处于它右侧，形成后子序列。我们称此元素所处的位置为枢轴，元素为枢轴元素。这样，可以对每一子序列重复同样的处理过程，形成递归形式，直到每一元素子序列只剩一个元素为止。

此方法关键在于确定序列或子序列的哪一个元素作为枢轴元素，一般选取待排序列的第一个元素作为枢轴元素，首先给出算法。

- 算法 3.2

设有序列{R[s]，R[s+1]，...，R[t]}

1・设置指针 i=s，j=t，R[s].key 为待排元素关键码 k_1，指针 p=&R[s]；

2・执行循环：

```
  while(i<j)do{
    while((i<j)&&(*(p+j).key>=k1))j--; //递增排序*(p+j)>枢轴 k 则顺序不变继续搜索
    *(p+i)<==>*(p+j)//由尾部起找到一小于枢轴的元素*(p+j),把它交换到前子序列 i 的位置
    while((i<j)&&(*(p+i).key<=k1))i++; //递增排序*(p+i)<枢轴 k 则顺序不变继续搜索
    *(p+i)<==>*(p+j); //自头部起找到一大于枢轴的元素*(p+i)把它交换到后子序列 j 的位
置
    }
```

3・当 i=j 时一个元素被排好序，可以递归调用此过程直到全部排序结束。

图 3.6 给出了一个元素的排序过程示意，如果枢轴元素 K_1 选择合适，则每次平分前后子序列可以得到较高的效率，程序 3.3 是 C 语言快速排序的完整程序。

程序 3.3 快速排序

```
void  qksort(struct node *p,int  l,int t)
{
   int  i,j;
```

```
    struct node x;
    i=l;j=t;x=p[i];  //取序列第一个元素作为枢轴
    do{
     while((p[j].key>x.key)&&(j>i))j--;//从尾部开始只要没找到小于 X 的元素就递减循环
     if(j>i){
          p[i]=p[j]; //向前单向交换，j 位置空出会在后续循环中被另外元素补上
          i++;                    //准备开始从头部向尾部搜寻
          }
     while((p[i].key<x.key)&&(j>i))i++;
     if(j>i){
          p[j]=p[i];//向后单向交换，i 位置空出会在后续循环中被另外元素补上
          j--;
          }
     }while(i!=j);         //只要 i 不等于 j 一趟排序尚未完成
    p[i]=x;//找到枢轴，把 X 放到其在序列的最终位置上，一趟排序完成
    i++;
    j--;
    if(l<j)qksort(p,l,j);//对前子序列 0～j 元素递归排序
    if(i<t)qksort(p,i,t);//对后子序列 i～n-1 元素递归排序
}
```

函数在主程序中调用形式为：

```
qksort(p, 0, n-1);
```

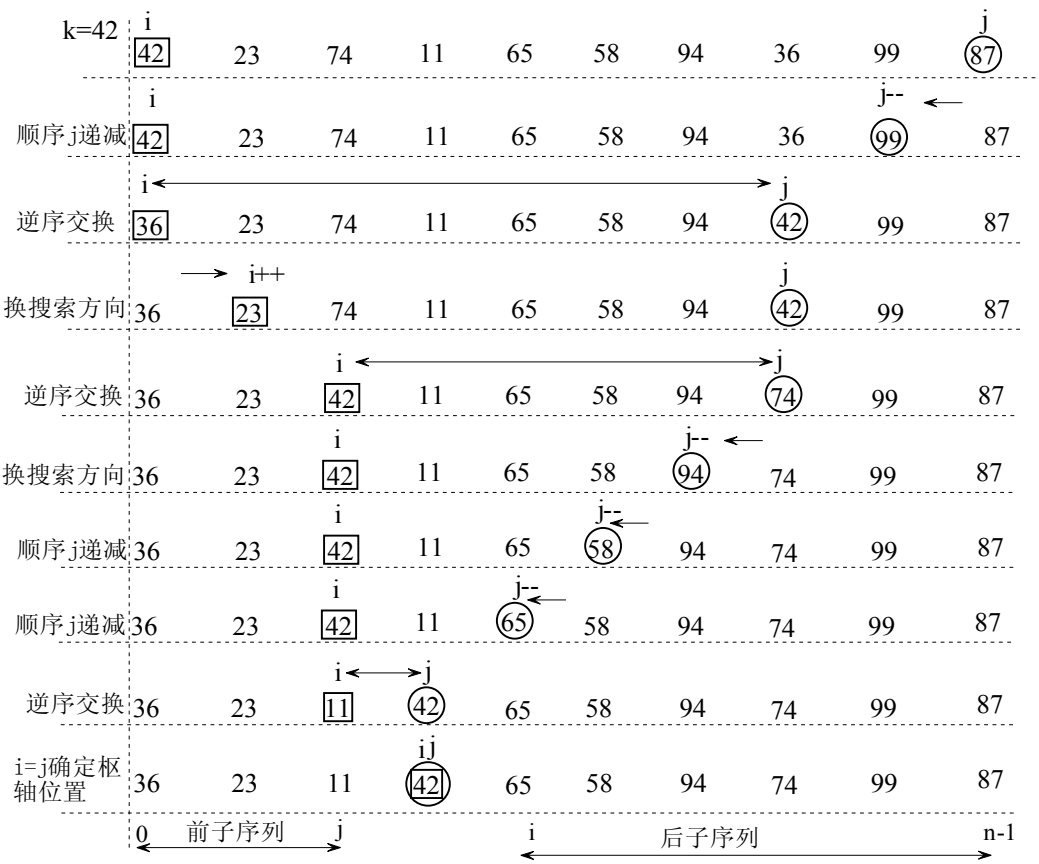


图 3.6 快速排序的一个元素排序过程图解

- 效率

快速算法的效率分析与枢轴元素选取密切相关，最差的情况是每趟排序选取的枢轴元素在排序后处于将长度为 n_i 的待排序列分割成一个子序列没有元素而另一个子序列有 n_i-1 个元素的位置上，于是，下次排序需要比较和交换的次数都是 n_i-1，如果在总长为 n 的排序数组中每次都发生这种情况，则其时间花费是 $\sum_{i=1}^{n} i = O(n^2)$，所以，最差情况下快速排序效率与冒泡排序相当，如果枢轴是随机选取的，发生这种情况的可能性不大。

快速排序最好的情况是每趟排序选取的枢轴元素在排序后处于将长度为 n_i 的待排序列分割成两个长度相等的子序列位置上，下次排序需要比较和交换的次数都是 $\frac{n_i}{2}$，如果每趟排序都发生这种情况，则总长为 n 的排序数组会被分割 $\log_2 n$ 次，每次的交换和比较次数是 $\sum_{i=0}^{\log_2 n} \frac{n}{2^i}$，这里假设 n 是 2 的整数幂，其时间花费是 $O(n \log_2 n)$。

快速排序的平均效率在最好与最差情况之间，假定选取的枢轴元素位置将第 i 趟排序数组的长度分割成 0，1，2，…，n_i-2，n_i-1 情况的可能性相等，概率为 $\frac{1}{n_i}$，则平均效率是：

$$T(n)=cn+\frac{1}{n}\sum_{i=0}^{n-1}(T(i)+T(n-1-i)) \qquad T(0)=c,T(1)=c$$

它仍然为$O(n\log_2 n)$数量级。快速排序算法中，我们要注意待排元素 K_1的选取方法不是唯一的，它对排序效率有很大影响。

3.4　堆排序

快速排序在其枢轴元素每次都选到位于其前后子序列的中点时，相当于每次递归找到一棵平衡二叉树的根，这时其效率最高。显然，如果我们能找到总是在一棵平衡二叉树进行排序的方法，就有可能得到比快速排序更高的效率。堆是一棵完全二叉树，堆的特点又是根为最大值，那么基于最大值堆排序的思想就很简单，将待排序的 n 个元素组建一个最大值堆，把根取出放到排序数组的位置[n-1]处，重新对剩下的 n-1 个元素建堆，再次取出其根并放置到排序数组的[n-2]位置处，循环直至堆空，堆排序完成。

实际上，排序数组就是堆数组，每次取出的根直接和堆数组的[n-1]位置元素交换，根被放到堆数组的[n-1]位置，而原来[n-1]位置的元素成为树根，于是，堆数组[0～n-2]的那些剩余元素在逻辑上仍然保持了完全二叉树的形状，可以继续对这些 n-1 个剩余元素建堆，图 3.7 显示了一个堆排序过程的前四步情况，程序 3.3 是堆排序程序。

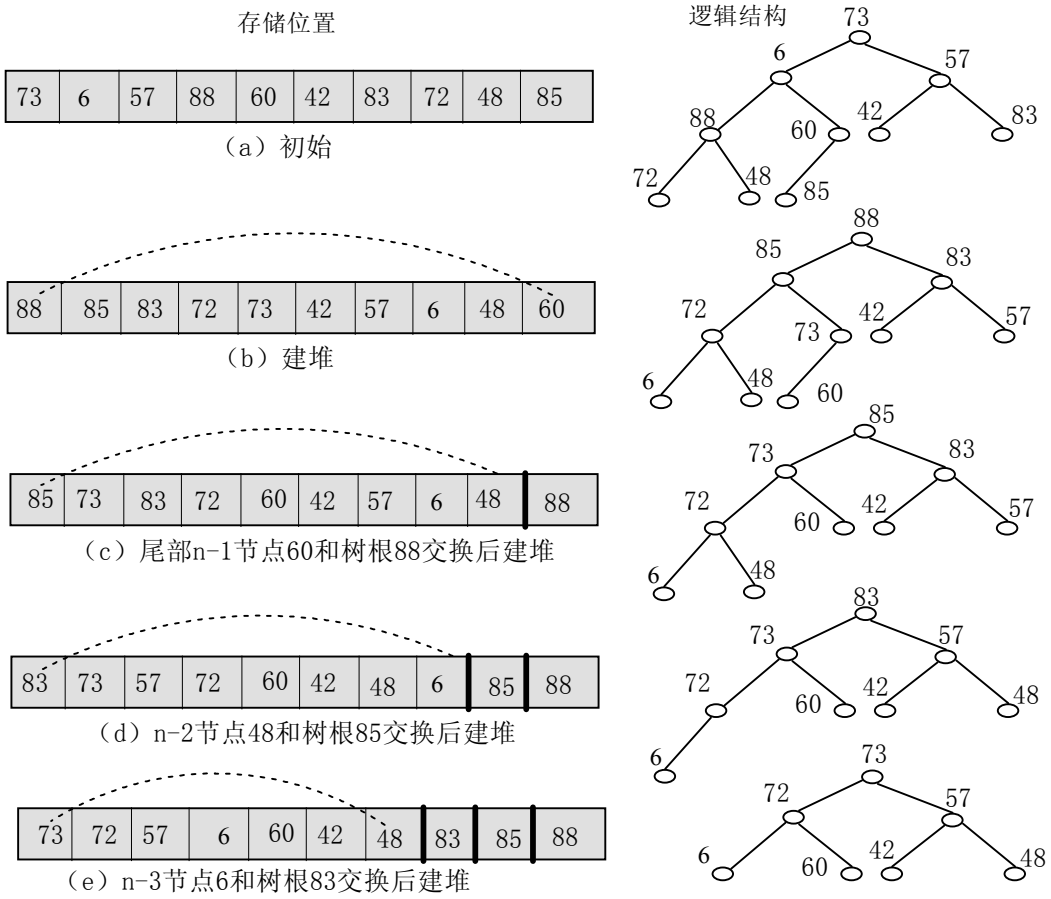


图 3.7{73,6,57,88,60,42,83,72,48,85}堆排序的前四步过程

程序 3.3 堆排序

```
void heapsort(struct node *heap,int n) //heap 指向 heaparray[0]，n 是堆数组长度
{
      int i;
      buildMAXheap(heap,n);//在堆数组[0～n-1]建立堆
      swap((heap+0),(heap+n-1));//将当前堆数组的最后一个节点与堆顶节点交换
      n--;                          //堆元素减一
      while(n){                     //排序过程直至堆剩余一个元素为止
         buildMAXheap(heap,n);
         swap((heap+0),(heap+n-1));
         n=n-1;
         }
}
```

- 堆排序效率

因为建堆效率是 O（n），而将位置 i 上的元素与堆顶元素交换需要 n-1 次，并重建 n-1 次堆，在最坏情况下，每次恢复堆的需要移动 $\log_2 i$ 次，那么，$\sum_{i=1}^{n-1}\log_2 i$ 次移动需要时间开销是 O（$n\log_2 n$），即堆排序最坏情况下的效率是 $O(n\log_2 n)$。

3.5 归并排序

- 两路归并排序思想

将两个有序的数组合并为一个有序的数组称之为归并（Mergesort）。设待排序数组有 n 个元素{R_1，R_2，...，R_n}，在前面讨论的直接插入排序方法中，我们对第 i 个元素排序时假定前 i-1 个元素是已经排好序的，初始 i 从 2 开始。与此类似，归并排序初始时将 n 个元素的数组看成是含有 n 个长度为 1 的有序子数组，然后将相邻子数组两两归并，归并后的子数组的长度倍增为 2，而个数减少一半为 $\frac{n}{2}$，反复归并子数组，直至最后归并到一个长度为 n 的数组为止。归并排序不同于快速排序，它的运行效率与元素在数组中的排列方式无关，因此避免了快速排序中最差的情形。

归并排序和增量排序有些类似，它们的区别在于，shell 是把长度为 n 的待排序序列以步长 $d_i=\frac{n}{2^i}$ 进行分割，以间隔为 d_i 的元素构成一组。而归并排序是相邻的元素为一组，继而以相邻组归并。

对包含 n 个元素的数组应用归并排序方法，需要一个长度为 n 的辅助数组暂存两路归并的中间结果，空间开销的增加是归并排序的一个弱点，但是，任何试图通过程序技巧来取消辅助数组的代价是程序变得极其复杂，这并不可取。

图 3.8 是两路归并排序示意。图中显示，一次归并结束时，序列尾部可能有 1 个子数组及元素不能归并，需要进行尾部处理。因为，一次归并过程中子数组能两两归并的条件是当前归并元素位置 i≤n-2L+1，否则，余下的待排序元素一定不足两个子数组的长度，需要进行尾部处理，方法有 2：①当前归并位置 i<n-L+1，即余数多于一个子数组，我们将它们看成是一个归并序列直接进行归并处理，并放入到归并结果序列的尾部；②i>n-L+1，余下元素个数少于一个子数组长度，我们将这些元素直接移入到归并结果序列的尾部。

- 两路归算法

  包含 n 个元素的数组两两归并排序的算法包含有两个函数，分别是①两组归并函数：

  merge(归并起点,子数组 1 终点位置,子数组 2 终点位置,待排序数组，中间数组)；

  ②一趟归并函数：

  mergepass(待排序数组长度，子数组长度，待排序数组，中间数组)；

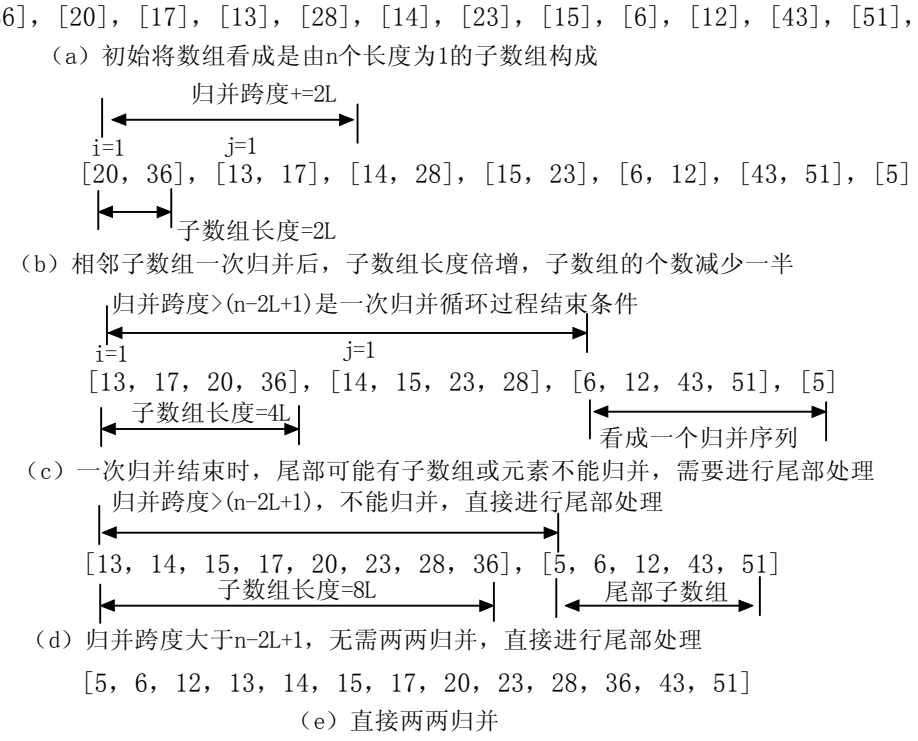


图 3.8 归并排序过程

我们先看子数组归并排序算法：

算法 3.3 两组归并

```
void merge(int L,int m,int m1,struct node *array,struct node *temp)
{
    int i=L,k=L,j=m+1;
```

```
    while((i<=m)&&(j<=m1)){
        if(array[i].key<=array[j].key){
                temp[k]=array[i];
                i++;
                }
        else{
            temp[k]=array[j];
            j++;
            }
        k++;
        }
    if(i>m)for(i=j;i<=m1;i++){
            temp[k]=array[i];
            k++;
            }
    else for(j=i;j<=m;j++){
                temp[k]=array[j];
                k++;
                }
}
```

一趟归并排序算法如下：

算法 3.4 一趟归并排序

```
void mergepass(int n,int L,struct node *array,struct node *temp)
{
    for(int i=1;i<=n-2*L+1;i+=2*L)merge(i,i+L-1,i+2*L-1,array,temp);
    if(i<n-L+1) merge(i,i+L-1,n,array,temp);
    else for(int j=i;j<=n;j++)temp[j]=array[j];
}
```

最后，我们得到两路归并排序算法如下：

算法 3.5 两路归并排序算法

```
int mergesort(int n,struct node *array)
{
    int L=1;
    struct node temp[N+1];//排序元素个数是N
```

```
    while(L<n){
        mergepass(n,L,array,temp);
        L*=2;
        if(L>=n){
            for(int i=1;i<=n;i++) array[i]=temp[i];
            return(0);
            }
        mergepass(n,L,temp,array);
        L*=2;
        }
}
```

当 $2^{i-1}<n\leq 2^{i}$ 时，mergepass()调用了 i 次（$i\approx\log_2 n$），每次调用 mergepass()的时间开销是 O(n)数量级，在 mergesort()中，最后有可能需要从 temp[]向 array[]移动 n 次，所以，两路归并排序的时间花费是 O(n $\log_2 n$)数量级，相当于快速排序方法。

3.6 数据结构小结

有关数据结构的内容就讨论到此，我们希望通过这一部份的学习使读者掌握基本的数据结构内容以及程序设计初步。希望通过教学、习题、上机试验几个环节为读者建立分析和设计数据结构的基本概念，包括理解数据逻辑结构、存储结构、排序与检索等内容，为大家将来的学习提供良好的基础。现在重新回顾已经学过的内容。

3.6.1 数据结构的基本概念

一、什么是数据结构

这个问题涉及到两个基本概念。

1. 数据类型：程序设计语言中各个变量所具有的数据种类。比如 FORTRON 的基本数据类型是整型、实数型、逻辑型、复数型。C 语言中除整型、字符型、实数型等基本数据类型外还有结构、共用体、枚举、位域等用户定义的数据类型。
2. 数据集合：指某种数据类型的元素集合。

在此基础上我们说数据结构是数据元素及其相互关系的集合。数据结构不仅描述数据元素，而且要描述它们之间相互关系所表达的元素之间存在的某种联系，称为结构，我们定义过二元组：

$$S=(D, R)$$

就是表达了这个意义。

程序设计中需要将数据结构存储在计算机内存中，即数据结构的物理映象，我们说数据的逻辑结构与物理结构是不可分割的两个内容，这里再次列出数据结构划分类型如图 3.9 所示。

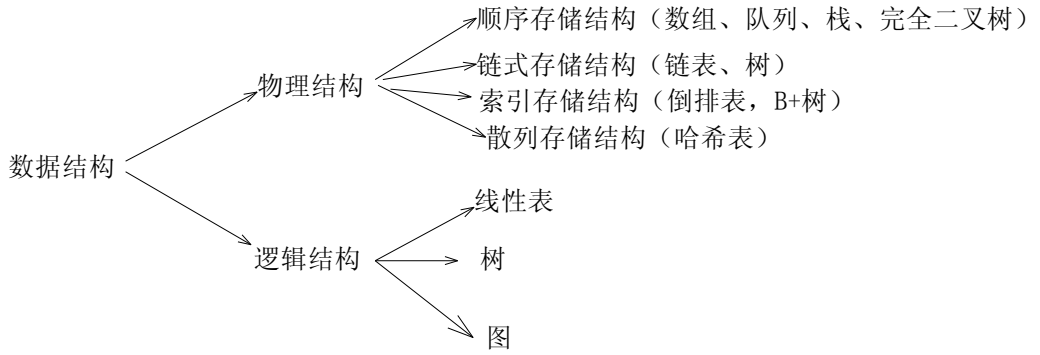


图 3.9 数据结构类型划分

3.6.2 数据结构分类

我们有三种基本的数据结构类型，即表、树、图。

3.6.2.1 数据结构中的指针问题

表结构中我们重点学习了链表设计，包括它的生长，删除等基本操作。在这里我们强调了 C 语言中指针的运用问题，对指针的理解一直是困扰我们学习 C 语言和数据结构的一个主要问题，我们应该清楚，指针首先是一个变量，它也有地址，其次，才是这个变量存储的是一个指向其它变量的地址，最后，这个指针必须和它所指向的那个变量的数据类型相同，因为在数组元素序列中，我们知道每个元素占用的存储空间大小由其数据类型决定，指针与其同类型，表明当对指针变量（内容）进行加一操作的时候，我们可以正确的指向数组内下一个元素的起始地址。

注意，指针只是指向变量所占存储空间起始地址，它是一个存储单元的地址，不是一段存储区域的空间，请看如下问题：

```
struct node {
        int key;
        ……
        struct node next;
        }*hash[40], *p,a1;
```

程序定义了一个 node 结构类型的指针数组 hash[40]和指针 p，操作如下：

```
&a1=malloc(sizeof(node));
hash[i]=a1;
p=hash[i];
```

问，p = hash [i]操作是把元素 hash[i]指向的 a1 的地址赋给了 p，还是把 hash[i]->next 赋给了 p 呢？是否可以写成 p=hash[i]->next 呢？

问题出在我们对指针的理解上。

（1）首先，定义是一个 node 节点类型的指针数组，所以，hash[i]就是指针，它指向一个地址 a1，比如是某一个链表的第一个数据节点。

（2）注意，指针是指向某数据类型变量的地址，它和该数据类型的内部结构无关。hash[i]只是一个 node 类型的指针变量，是地址的概念，它没有实际占有空间，不是 node 节点，当然也就没有 node 节点变量的指针域，比如：

```
struct node a;
```

变量 a 是有指针域 next 的，你可以用：

```
a-->next=a2;
```

语句让 a 指向 a2，但是，hash[i]指针没有 next 指针域这一说，所以：

```
p=hash[i]-->next;
```

是错误的，实际上，hash[i]指向了链表头节点 a1 的地址，所以，p=hash[i]就可以了。

3.6.2.2 线性表的效率问题

顺序表、栈、队列等是基本要掌握的内容，特别在某一条件下的效率分析问题，比如，删除操作时顺序表平均移动次数是表长 n 的函数：$ASL = \frac{n-1}{2}$，当 n 很大时它的效率明显低于链表结构，但在随机读写效率上顺序表优于链表。我们特别注重顺序表与链表在概念上的区别。此外，递归也是同学必须掌握的内容之一。

3.6.2.3 二叉树

在树结构中我们重点讨论了二叉排序树。包括二叉树的基本概念、几种特殊的二叉树结构如满二叉树，完全二叉树，平衡二叉树等是同学必须了解的基本内容。二叉树程序设计是我们学习的重点，它比链表设计增加了递归设计内容的难度，读者必须通过实际编程，才能熟练掌握和很好的理解二叉树设计与应用问题。

我们指出二叉树的存储结构是多种形式的，有顺序存储、链式存储方式等，一般我们采用链式存储结构，它含有两个指针域一个数据域，对于哈夫曼树还增加了父指针域。我认为，在二叉树程序设计中最基本的思想是递归程序设计的运用。用递归实现它的动态生长，删除，检索，遍历操作，原则上要求读者设计二叉树相关操作时应是递归结构实现的。

我们还指出，从概念上讲数据结构另一种分类方式是静态与动态数据结构的区别。所谓静态和动态是指数据结构特性在该数据结构存在期间的变动情况。顺序表是静态的，因其结构特性在它存在期间是不变的，只有长度的变化。树是动态的，因为它的结构特性依赖于它的生长过程。应该说链表也是静态的数据结构，至少不能说它是动态的。因为链表插

入与删除不改变它本身特性，只是表长的变化，这点与教材看法不同，大家可以商榷。

无论是表结构还是树结构，简洁明快的结构化程序设计风格是我们所追求的，数据结构内容本身读者可以在将来的学习中不断加深理解，但良好的 C 语言程序设计能力与风格则是你们整个软件技术学习中的基础。

3.6.3 排序与检索

在数据结构内容之内，属于数据结构操作的一类问题是检索与排序。在检索方面，顺序检索、对半检索都是基本内容，我们重点讨论了它们的检索效率。

在描述数据结构的物理结构时我们知道有顺序、链表、散列和索引四种基本结构，其中散列存储结构就是通过哈希函数实现关键码值到存储地址的转换，即哈希造表或地址散列。这是一个重要的概念，它通过计算函数表达式求解元素地址，它所注意的是由于关键码集合空间大于内存可分配地址空间而造成的地址冲突问题。构造哈希函数的标准是它与关键码的相关性，以期达到地址均匀散列。解决地址冲突的办法是线性探测与链地址法。

在排序方面，对于基本概念我们主要考虑了稳定性和排序效率问题，此外还讨论了直接插入排序与快速排序方法，我们特别注意的是快速排序中枢轴元素的选取问题，如果每次选取到序列的中值则可以平分前后子序列，使效率最佳。

3.7 算法分析的基本概念

算法的内容是数据结构设计中经常遇到的术语与概念。我们说程序设计是算法与数据结构在计算机上的实现，算法也就是解决问题的步骤。衡量各种数据结构优劣时，就需要用到算法分析内容。

3.7.1 基本概念

算法分析是程序执行时间的定性估算，也就是运行时间的上、下限范围。为此，我们需要明确一些基本概念。首先，算法分析有时间复杂度与空间复杂度两个方面，目前一般的计算机内存已足够大，在简单程序设计情况下，我们可以做到用程序的内存工作长度空间来换取时间上的处理速度，因此，有时候我们说对于算法分析来说更看重的是时间复杂度。

实际上不能这样简单处理。计算机内存资源是我们程序运行时间的基本条件，在当前普遍应用的多任务操作系统中，每一个进程分配的内存资源是有限的，当你程序算法占用的空间超过操作系统分配给你的资源时候，操作系统会在硬盘物理空间上给你开设缓存区域，或者在你的数据文件太大（它总是保存在硬盘物理空间上），运行中需要分阶段从内存调进的情况下，程序执行过程中会出现频繁的内外存数据交换，也就是 I/O 操作，常识告诉我

们，任何一种 I/O 操作耗时远远大于 CPU 与内存之间的数据交换时间，差异大于一个到两个数量级。因此，算法设计中空间复杂度依然是一个重要因素。

衡量算法优略的一个基本标准是说，处理一定规模（Size）的输入时该算法所执行的基本操作（basic operation）的数量。

这里，规模的概念依赖于具体算法，比如，检索和排序算法中的输入元素数量。基本操作的概念依据计算机硬件资源不同而有所不同，比如，一般的计算机 CPU 都支持两个变量之间的加减乘除的整数操作、浮点数操作，但是不支持 n 个整数之间的累加操作，我们定义基本操作的性质是完成该操作所需的时间与操作数具体值无关，所以，两个变量之间的加减乘除的整数操作可以看成是基本操作，但是 n 个数累加所花的时间就要 n 来决定，比如 for 语句的循环次数。

程序 3.4 是在一维数组中检索最大值，它依次遍历数组每一个元素，比较每一元素值并保存当前最大元素。

程序 3.4 求数组最大值

```
int iargest(int *array,int n)
{
int currlarge=0;
for(int i=0;i<n;i++)if(array[i]>currlarge)crrrlarge=array[i];
renturn(currlarge);
}
```

这里，n 是任务规模，基本操作是比较运算、赋值操作，它们所需的时间与其在数组中的位置 i 无关，也与元素数值的大小无关。影响程序运行时间的因素是规模 n，设基本操作时间是 c，则程序 3.4 的运行时间为 t=cn，定性的说，程序 3.4 算法时间代价是 T(n),它与输入规模呈线性关系增长。现在看程序 3.5 描述的情况。

程序 3.5 变量累加

```
long add(int n)
{
long sum=0;
for(int i=1;i<=n;i++)for(j=1;j<=n;j++)sum++;
renturn(sum);
}
```

程序 3.5 的输入规模是 n^2，基本操作是加法，设基本操作时间为 c，其运行时间是 cn^2，我们估计它的运行时间代价是 $T(n^2)$，这里的 c 包含了程序变量初始设置等因素。当规模为平方项的时候，时间代价的增长率就很可观了。

时间函数的增长率是衡量算法性能的关键指标。时间复杂度是定性分析，要定量确定某一算法所用时间是很困难的，我们只要确定它随规模 n 的增长率在某一数量级，确切的说是确定那些具有最大执行频度（规模）的语句。所谓频度可以从下列程序语句中看到它的概念。即频度就是某一语句（基本操作）的循环执行次数，程序的频度是程序中具有最大频度的语句所具有的频度。下例中，因为程序 1 只是单一命令语句，所以它的频度为 1；程序 2 有一个 n 次循环语句，它的频度是 n；程序 3 是双循环语句，当然它的频度就是 n×m。

```
程序 1          程序 2                  程序 3
...             ....                      ...
x++;        for(i=0; i<n; i++){...}     for(i=0; i<n; i++){
                    ...                       for(j=0; j<m; j++){..}
                                              }
                                          ...
                                          }
频度为 1        频度为 n                  频度为 n×m
```

据此，我们估计程序 1 算法时间复杂度是 O(1)，程序 2 是 O(n)，程序 3 是 O(n×m)数量级的。

表 3.1 几种时间复杂度随规模 n 的增长率比较

n	$\log_2 n$	$n\log_2 n$	n^2	n^3	2^n
1	0	0	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4096	65536
32	5	160	1024	32768	483648

3.7.2 上限分析

一个算法的运行时间上限用来描述该算法可能有的时间花销的最高增长率，它与输入规模 n 有关，我们知道，一个算法的具体时间花费与输入元素的分布特性有关，比如快速排序过程中的最差、最佳和平均时间代价的不同。因此，时间估计的上限也分成该算法平均条件下的增长率上限、最佳条件下的增长率上限或最差条件下的增长率上限。

现在我们给出算法时间复杂度的上限定义：如果存在两个常数 C≥0，n0≥0，当 n≥n0 时有|f(n)|≤C|g(n)|，则称函数 f(n)是 O(g(n))的时间复杂度。

这里，大 O 表示时间估计上限，|f(n)|是某算法的运算时间，定义表明，如果说一个算法的时间复杂度是 O(g(n))数量级，则表明该算法实际所耗时间只是 O(g(n))的某个常数倍，n 是算法规模参数。因此，当我们写 O(1)时间复杂度时，意味着算法执行时间是一个常数，而 O(n)是 n 的线性函数，$O(\log_2 n)$是对数函数。我们知道当 n 充分大时 $\log_2 n$ 远小于 n，$n\log_2 n$

是远小于 n^2 的，一些常见的关系已经在表 3.1 给出。

例 3.1 设有一算法，分析其每一语句的频数为 C_kn^k，则它们的和是：

$$p(n)= C_kn^k + C_{k-1}n^{k-1}+ C_{k-2}n^{k-2}+ \bullet\bullet\bullet + C_1n^1+ C_0$$

这里 C_i 为常数，C_k 不为零，n 是程序执行规模，因为：

$$\left|c_k n^k + c_{k-1}n^{k-1} + \ldots + c_0\right| \le \left|c_k n^k + c_{k-1}n^k + \ldots + c_0 n^k\right| \le (c_k + c_{k-1} + \ldots + c_0)\left|n^k\right|$$

由定义我们可知，该算法的时间复杂度是 $P(n)=O(n^k)$，即程序的算法时间复杂度只考虑具有最大频度的语句。

例 3.2 顺序检索的时间复杂度上限估计。我们知道，线性检索表的检索成功的平均比较次数与规模 n 的关系是 $\frac{n-1}{2}$，或者说它的平均检索时间估计是 $c\frac{n}{2}$，对于 n〉1，我们有关系 $\left|c\frac{n}{2}\right| \le c|n|$ 成立，所以它的平均时间复杂度上限是 O(n)。

3.7.3 下限分析

上限估计告诉我们一个算法时间的增长率极限情况，有时候我们想知道一个算法至少它需要多少运行时间，也就是它执行时间的下限程度，我们用符号 Ω 表示。定义：如果存在两个常数 C≥0，n0≥0，当 n≥n0 时有|f(n)|≧C|g(n)|，则称函数 f(n)的时间复杂度下限是 Ω(g(n))。

例 3.3 设一算法时间花费是 $T(n)= C_kn^k + C_{k-1}n^{k-1}$，因为 $\left|c_k n^k + c_1 n\right| \ge \left|c_k n^k\right| \ge c_k\left|n^k\right|$，根据定义，它的时间估计下限是 $\Omega(n^k)$。

对于线性检索，在数组中要找到关键码为 k 的元素，最差的情况下，我们可能需要检索完整个数组长度才能确定，最好的情况下是 1 次，平均是数组长度的一半，因此，在平均和最差检索情况下，至少需要检索 cn 次（c 是基本操作），所以它的下限估计在这两种情况下也是 Ω（n）。

当一个算法的时间估计上下限相等，都是 g(n)的时候，我们说该算法它的时间估计是 $\Theta(g(n))$ 的，即 g(n)既在 O()中，也在 Ω（）中。

3.7.4 空间代价与时间代价转换

算法的空间开销与时间开销是一对矛盾体。比如，计算 sin、cos 函数值是用级数求和的过程，如果在算法中多次使用 sin、cos 函数值，可以假定一个合理的精度将 sin、cos 函数值制作成一个表，比如以每分为一个步长，于是，计算 sin、cos 函数值的过程就转换为查找表的过程，给出表的基地址和偏移量，我们可以很快查找到相应的函数值，而这一

过程的代价是查找表所占用的内存空间。

另外一个例子是布尔变量的问题，它取值只有两种状态，真或假，假定一个算法使用了32个布尔变量，一种方法是每个布尔变量用以字符类型量表示，需要占用32字节。我们也可以用一个long类型变量表示，它的每一比特（bit）位代表一个布尔变量，只需要4个字节，显然，程序上前者简单后者繁琐，但是空间开销正好相反。

再看一个实际例子，假定一个算法中需要多次使用阶乘，最大是12的阶乘，当然可以很容易的用递归函数计算阶乘，它需要一个长整型量和多次递归调用。而从查找表概念理解，预先计算12！如表3.2所示。

表3.2 阶乘查找表

12!	11!	10!	9!	8!	7!	6!	5!	4!	3!	2!	1!
479001600	39916800	3628800	362880	40320	5040	720	120	24	6	2	1

表3.2是一个数组，计算12以内的阶乘就是将给定值作为偏移量访问查找表的过程，它非常简单，是空间换时间的典型例子。

我们应该清楚，存储空间和时间代价互换的基本原则是，避免程序执行过程中出现不必要的内外存数据交换，即，外存硬盘空间开销越小，程序执行越快，我们在前面说过，I/O数据交换是计算机运行的瓶颈。

有关算法的详细内容同学可以参考其它教科书，算法与数据结构是密切相关的，选择一种数据结构也就确定了元素之间的关系，它对应不同的算法，我们当然希望有一种好的数据结构得到高效率的算法表达，使时间复杂度最小，所以有

算法＋数据结构＝程序

这一说法，有关数据结构的基本内容就讨论到此。

第 6 章 高级数据结构内容--索引技术

6.1 基本概念

索引文件用于组织磁盘中大量数据纪录检索的排列方式，主要是为提高关系数据库的操作效率而设计的。一个应用关系数据库设计时，从逻辑结构上看每一客观实体（关系）至少有一个能唯一的标识其所有属性或该关系的主属性，称为主关键字。若一个属性不能唯一的标识一个关系，或者说它对应多个关系实体，则称其为该关系的次关键字。在关系数据库设计时，我们总是按照主关键字组织数据字典的全局逻辑结构及联接关系的，在物理实现上，也是用主关键字与纪录的物理地址相关联的。

当大量的数据纪录在内存时，为了提高检索效率就必须按检索的形式进行排序，显然，主关键字检索是检索关系实体的基本操作之一，比如，一个学生数据库用学号唯一标识学生这个客观实体的所有属性，要检索特定学生的情况时，输入该学生的学号可以检索到该生所有属性值（如姓名、年龄、性别、籍贯、家庭所在地、系别等），于是，我们自然会以主关键字对学生记录排序。

现在的问题是，检索是多角度、综合性进行的，比如，我们需要查看自动化系、来自浙江杭州的学生情况，查找符合这一检索条件所要求的记录的方法可以有多种，比如，将记录按‘系别’进行排序，在检索到‘系别’等于‘自动化系’的记录中，筛选出‘家庭所在地’的属性值等于‘杭州’的记录。然而，要对‘系别’这一属性进行排序，就得把内存物理地址与该属性相关联，换句话说，就是所有记录需要按‘系别’属性重排。此外，同一关系数据库还有可能遇到查询‘年龄’等于 20 岁的女生记录的情况，或者是‘姓名’等于‘xxx’的检索要求等。显然，我们不可能每次都按检索条件重排硬盘中关系数据库的所有记录。

避免重排数据库记录的另一种选择是索引技术（index file）。我们称之为索引技术或文件的是，其文件内每个关键码与标识该记录在数据库的物理位置的指针相关联（一起存储）。因此，索引文件为记录提供了一种按索引关键字排列的顺序，而不需要改变记录的物理位置。一个数据库系统允许有多个索引文件，每个索引文件都通过一个不同的关键字段支持对记录有效率的访问，即，索引文件提供了用多个关键字访问数据记录的功能，也避免了数据库记录的重排操作。简而言之，索引技术的应用使记录的检索与其物理顺序无关。

显然，我们不可能让每个关键码都与记录的物理指针相关联，这样，记录的物理位置变化会造成所有索引文件的修改。次关键码索引文件实际上是把一个次关键码与具有这个次关键码的每一个记录的主关键码相关联起来，而主关键码索引再与一个指向记录物理位

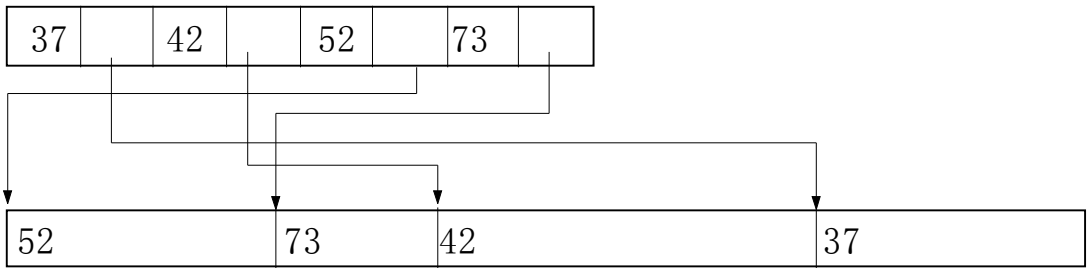
置的指针相关联，即，其访问顺序是次关键码-主关键码-记录物理位置。

可以说，索引技术是组织大型数据库的关键技术，其中，线性表索引主要用于数据库记录的检索操作，对于记录的插入与删除操作我们广泛使用的是树形索引，即 B^+树。

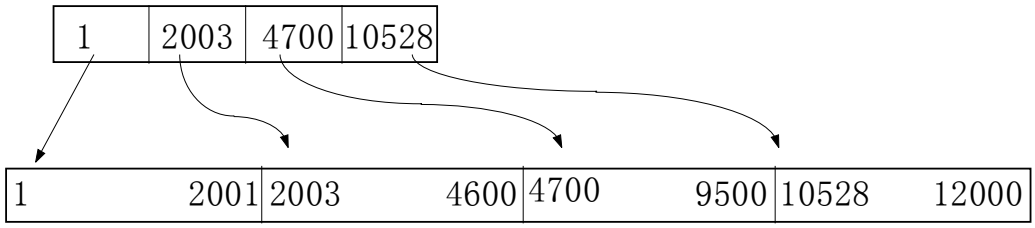
6.2 线性索引

6.2.1 线性索引

线性索引是一个按关键码-指针对顺序组织的索引文件，该文件按关键码的顺序排序，指针指向记录的物理位置，见图 6.1 所示。线性索引文件本身的记录是定长的，其检索效率是 O(n)，随数据库记录增大而降低，即使索引文件是顺序存储的能使用二分检索，但是当数据库记录数目很大时，索引文件本身也无法在内存中一次装入，运用二分检索过程中会多次访问硬盘，效率可能还不如线性检索。

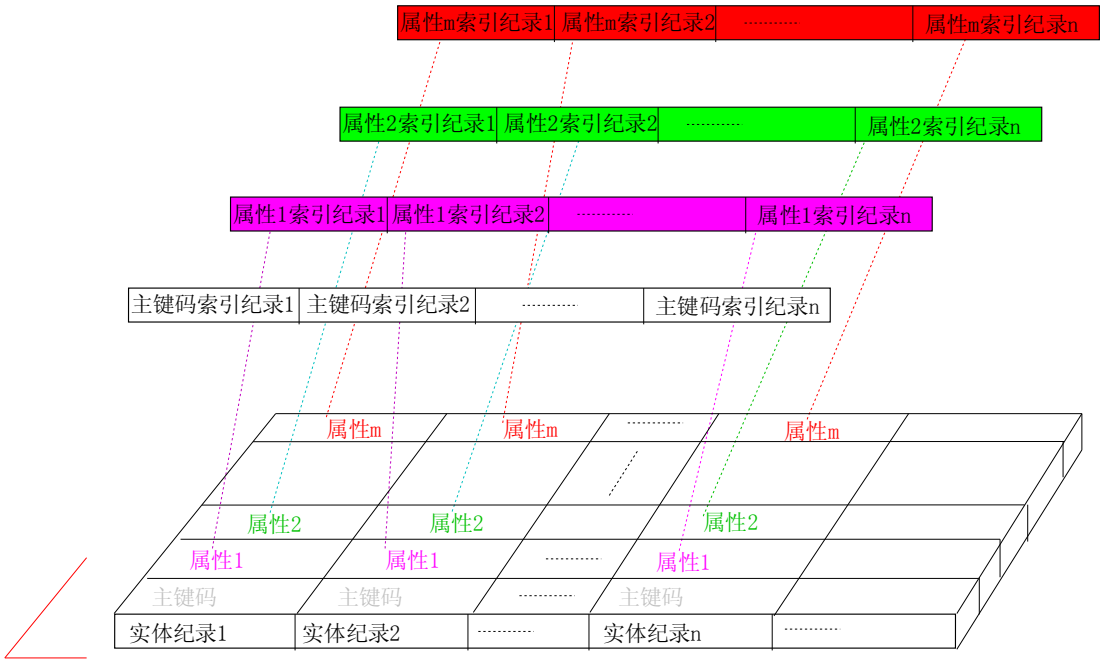


我们熟知的多级索引结构是一种解决方案，比如，一个关键字/指针对需要 8 个字节，磁盘中一个物理页大小是 1024 字节，即可以存储 128 个索引记录，设，数据库记录的线性索引文件长度为 100 磁盘页（100kB），每页内记录（关键码值）按递增有序，则在内存中建立一个二级索引文件，其每个记录是每个磁盘页的第一个关键码值，该二级索引文件长度只有 100 项，数据库检索时，首先根据检索关键码值在这个顺序表文件中用二分检索找到小于或等于它的最大值，就可以找到包含该关键码的索引文件磁盘页，将该磁盘页调入内存只需继续在 128 个记录中查找即可完成，见图 6.2 所示。



6.2.2 倒排表

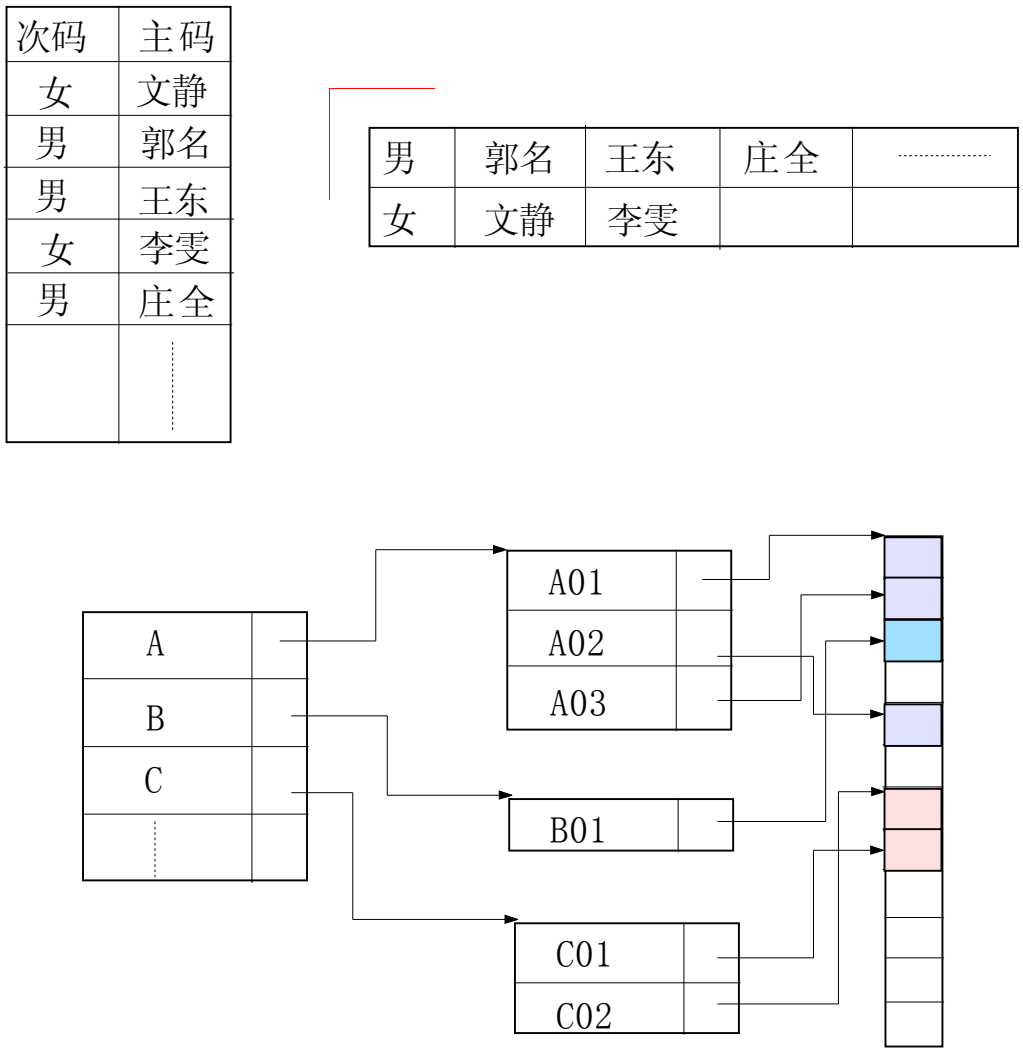
二级索引存在的一个问题是，当我们向数据库插入或删除一条记录时，必须同时修改磁盘与内存中的一、二级索引文件，当索引文件很大时，这种更改涉及到所有磁盘页数据（索引记录）的移动，代价很大。特别是当索引文件所有纪录均与物理纪录相关联的情况下，随着一个实体纪录的插入或者删除，数据库的所有索引文件都需要修改。另外，当次关键码取值范围很窄时，数据库记录中有大量的记录取值相同，造成索引文件中大量的重复引用。比如一个极端的例子是学生记录中的性别属性，非男即女，于是，索引文件中有一半记录是重复的。



如果所有索引文件指针均和物理记录关联，则实体纪录数目变更时将影响所有索引文件

一种改进的方法是二维数组形式，其每一行对一个次关键码值，每一列包含了具有该次关键码值的所有记录的主关键码值，于是，首先通过次关键码检索索引表找到它所对应的所有主关键码值，下一步可以通过主关键码检索数据库记录，从而得到满足检索条件的所有记录列表，见图 6.3 所示。二维索引表不但可以减少空间，而且数据库更新操作对索引文件的影响也被限制在索引文件的一行之内，极限情况是次关键码取值范围变化，比如新添一个次关键码值时候，需要移动二维数组，但这种情况实际中很少见。

它不足之处是数组必须有一个固定的长度，则有①与次关键码相关联的主关键码记录数受限（限定了数据库具有相同次关键码值的最大允许记录数）；②当次关键码取值对应的数据库记录数目变化很大时，二维数组长度无法兼顾，从而造成一个次关键码值对应很少记录数（比如只有一个主关键码相关联）的话，存储空间浪费很大。



既然二维数组也是一个线性表，显然链表是一种解决办法。我们定义的倒排表是：每一次关键码值有一指针指向它所对应的所有主关键码形成的数组，数组中的每一主关键码与磁盘中的数据库记录位置有唯一的指针相关联，即，检索记录顺序是由次关键码到主关键码，再到数据库记录，见图 6.4 所示。

6.3 2-3 树

基于线性索引技术的关系数据库存储方式存在的最大问题是，当大量的记录频繁更新的时候，其操作效率非常低，原因是每更新一条纪录就要修改一个索引文件内的所有的指针内容，当你有多个索引文件，以及海量的记录数目时，线性索引效率很低， 而且，修改指针的操作效率与线性索引结构无关，多级索引结构虽然能提高检索效率，但是记录物理地址变更引起的指针修改是全索引文件范围内的，既线性索引文件其所有指针信息都是相关的，如果我们有一种方法，记录的插入与删除只是影响索引文件局部区域，那么它作为

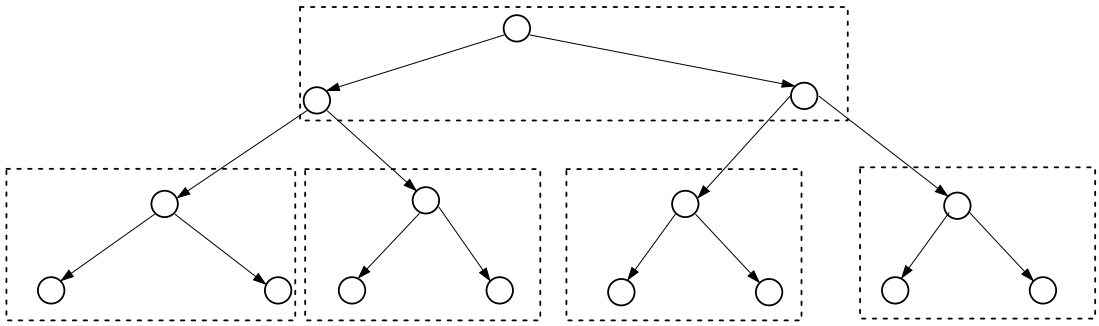
主关键码的索引就会完成得很好。树形索引是一种很好的索引文件组织方式，它本身是非线性结构，磁盘页之间指针相关性很低，它可以提供有效的插入与删除操作，从以前的二叉检索树讨论中知道，其效率是树深度的对数关系。

二叉树当然不适合作为索引文件结构，第一，它只有两个子树，不符合磁盘页划分；第二，它会因为更新节点操作变得不平衡，尤其检索树存储在磁盘中时，不平衡情况对检索效率影响更为重要，当一个节点深度跨越了多个磁盘页时，对节点的访问就是从第一个磁盘页（根节点）开始达到这个节点所在磁盘页为止的所有节点路径之和，随着磁盘页在内外存中的调进调出，它涉及到多次内外存交换，效率变得非常低下。因此，要采用树形索引结构，必须寻找一种新的树结构，它能解决插入与删除操作带来的不平衡问题。这种树形结构经过多次更新操作之后能自动保持平衡，并且适合按页存储，既，我们要求它的算法具有下列特性:

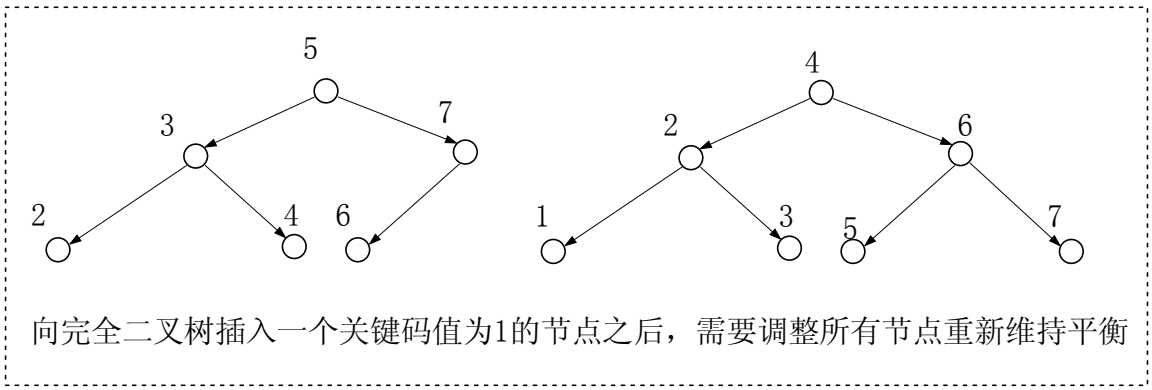
(1) 以一个磁盘页为单位；

(2) 插入与删除操作之后能自动保持高度平衡；

(3) 平均访问效率最佳。



基于磁盘页的树形索引结构



向完全二叉树插入一个关键码值为1的节点之后，需要调整所有节点重新维持平衡

首先，我们讨论 2-3 树概念，在此基础上引伸出 B^+树。

6.3.1 2-3 树定义

一棵 2-3 树具有下例性质：

1. 一个节点包含一个或者两个关键码；
2. 每个内部节点有 2 个子女（如果它包含一个关键码），或者 3 个子女（包含 2 个关键码）；
3. 所有叶子节点在树的同一层，因此树总是高度平衡的。

类似于二叉排序树，2-3 树每一个节点的左子树中所有后继节点的值都小于其父节点第一个关键码的值，而中间子树所有后继节点的值都大于或等于其父节点第一个关键码的值而小于第二个关键码的值，如果有右子树，则右子树所有后继节点都大于或等于其父节点第二个关键码的值。见图 6.5 所示。

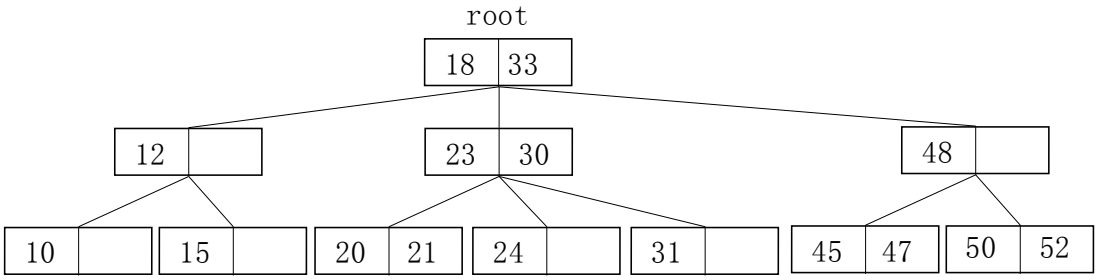


图6.5 2-3树 [因自 文献]

一个在 2-3 树中检索特定关键码值的函数类似于二叉排序树检索过程，见例 6.1 所示。

例 6.1 2-3 树检索函数实现

```
struct node *findnode(struct node *root,int key)
{
if(root==Null)return Null;
if(key==root→lkey)return root;
if((root→Numkeys==2)&&(key==root→rkey))return root;
if(key<root→lkey)return findnode(root→left,key);
else{
    if(root→Numkeys==1)return findnode(root→center,key);
    else{
        if(key<root→rkey)return findnode(root→center,key);
        else return findnode(root→right,key);
        }
    }
}
```

显然，2-3 树节点定义为：

```
struct node {
        int lkey,rkey,Numkeys;
        struct node *left,*center,*right;
        };
```

6.3.2 2-3 树节点插入

向 2-3 树插入记录（不是节点）与二叉排序树相同的是新纪录始终是被插入到叶子节点；不同的是 2-3 树不向下生长叶子，而是向上提升分列出来的记录。分以下几种情况：

（1） 被插入到的叶子节点只有一个关键码（代表一个记录），则新记录按左小右大原则被放置到空位置上。见图 6.6 所示。

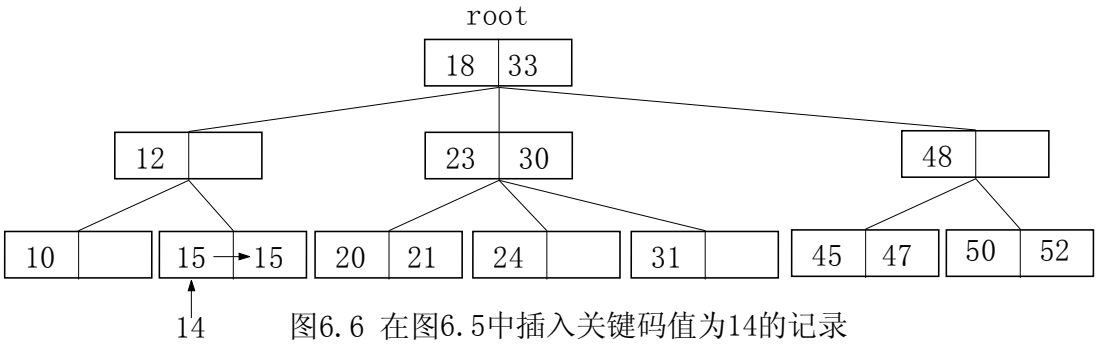


图6.6 在图6.5中插入关键码值为14的记录

（2） 被插入的叶子节点已经有 2 个关键码，但其父节点只有 1 个关键码。当被插入叶子节点内部已经没有空位置时，我们要创建一个节点容纳新增记录和原先 2 个记录。设原叶子节点为 L，首先将 L 分裂为 2 个节点 L 和 L’，L 取 3 个节点中值为最小者，L’取值为最大者，居中的关键码和指向 L’的指针被传回 L 的父节点，即，所谓提升一次的过程。被提升到父节点的关键码按左小右大排序，并插入到父节点空位置中。见图 6.7 所示。

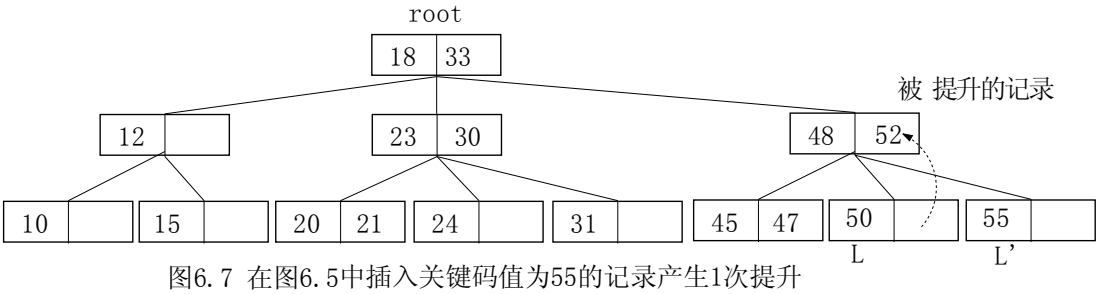
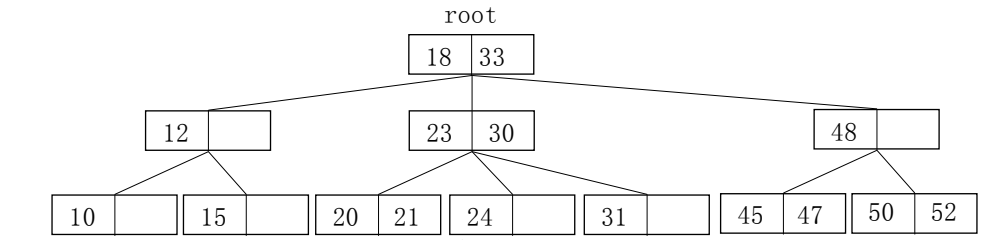
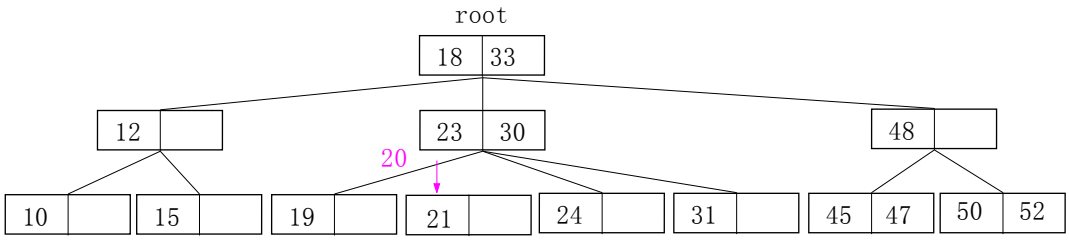


图6.7 在图6.5中插入关键码值为55的记录产生1次提升

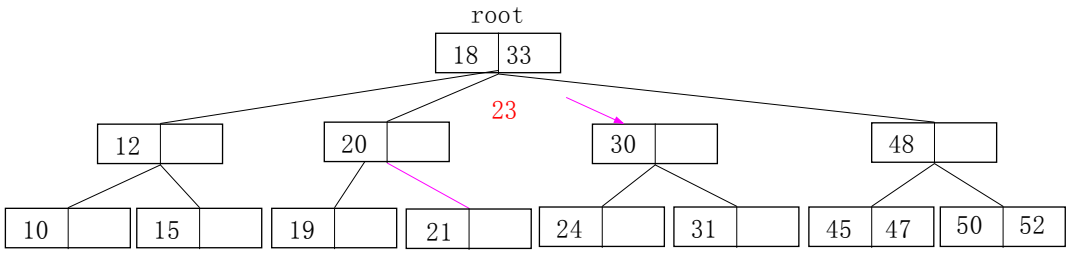
（3） 被插入的叶子节点已经有 2 个关键码，而且其父节点内部亦满。此时，我们用从叶子节点提升上来的关键码对父节点重复一次分裂--提升过程，将一个关键码从父节点中向更上一层提升，直至根结点，如果根节点被分裂，则继续提升的关键码形成新的根节点，此时，2-3 树新增一层。见图 6.8 所示。



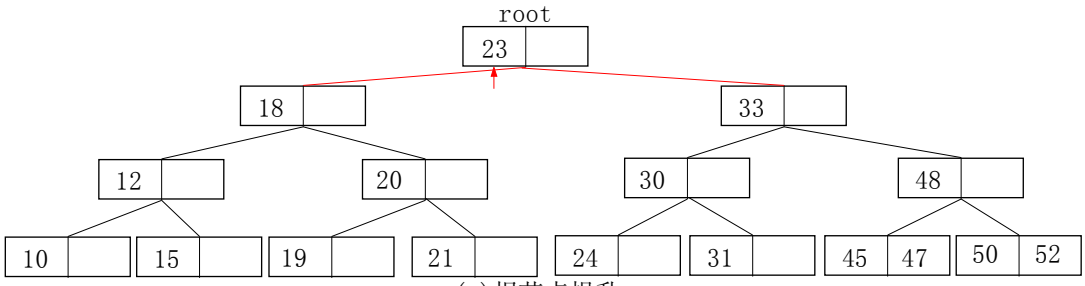
初始



(a)插入19产生叶子分裂与提升



(b)父节点产生分裂与提升



(c)根节点提升

图6.8

例 6.2 2-3 树插入函数

```
struct node *insert(struct node *root,int key, struct node *retptr,int retkey)
{
int myretv;
struct node *myretp=Null;
if(root==Null){
        root=(struct node*)malloc(sizeof(struct node));
        root→lkey=key;
        root→Numkeys=1;
        }
else{
    if(root→left==Null){//叶子节点
```

```
        if(root→Numkeys==1){//只有一个关键码
                root→Numkeys=2;
                if(key>=root→lkey)root→rkey=key;
                else{
                    root→rkey=root→lkey;
                    root→lkey=key;
                    }
                }
        else {//关键码满，分裂提升
            retptr=(struct node*)malloc(sizeof(struct node));//申请 L’节点
        且返回指针指向 L’
            if(key>root→rkey){// L’节点取最大值的关键码
                        retptr→lkey=key;
                        retkey=root→rkey;//提升中间值的关键码
                            }
            else{// root→rkey 是最大值的关键码
                        retptr→lkey=root→rkey;
                        if(key<root→lkey){//判别中间值的关键码
                                retkey=root→lkey;
                                root→lkey=key;
                                }
                        else retkey=key;
                        }
            root→Numkeys=retptr→Numkeys=1;//置 L 和 L’关键码数为 1
            }
    }
else {//非叶子节点小于左关键码值搜寻左子树*/
            if (key<root→lkey) insert(root→left,key, myretp,myretv);
            else {/*子树为 2 叉或小于右关键码搜寻中间子树*/
            if( (root→Numkeys==1) ||(key<root→rkey))
                                        insert(root→center,key,myretp,myr
                                        etv);
                else{//搜寻右子树
```

```
        insert(root→right,key, myretp,myretv);
        }
    }
  if(myretp!=Null){// 有孩子节点分裂而形成提升
if (root→Numkeys==2) {//分裂并提升父节点
          retptr=(struct node*)malloc(sizeof(struct node));
        root→Numkeys=retptr→Numkeys=1;
        if(myretv<root→lkey){//*提升左关键码
                  retkey=root→lkey;//*返回值
                  root→lkey=myretv;//*原 root 为 L
                     retptr→lkey=root→rkey;//L’关键码
                     retptr→left=root→center;
                     retptr→center=root→right;
                     root→center=myrept; //指向 L’
                     }
        else{
           if(myretv<root→rkey){ //提升中间点
                      retkey=myretv;
                      retptr→lkey=root→rkey;//L’键
                         retptr→left=myrept;
                         retptr→center=root→right;
                         }
            else{//提升右关键码
                     retkey=root→rkey;
                      retptr→lkey=myretv;
                      retptr→left=root→right;
                      retptr→center=myrept;
                              }
               }
           }
else{//root 节点内只有一个键，可增加一个
        root→Numkeys=2;
        if(myretv<root→lkey){
```

```
                    root→rkey=root→lkey;
                    root→lkey=meretv;
                    root→right=root→center;
                    root→center=myretp;
                    }
                else{
                root→rkey=myretv;
                root→right=myretp;
                    }
                }
            }
        }
    }
```

调用例 6.2 返回的是提升关键码的值（如有则为新的根节点键）与指向（如有则为新的根节点指向）L’的指针。其中要注意的是分裂父节点几种处理情况：①提升左关键码值，此时一定是从左子树中分裂提升，见图 6.9 所示；②提升中间关键码值是从中间子树中分裂提升上来的，见图 6.10 所示；③提升右关键码情况见图 6.11 所示。

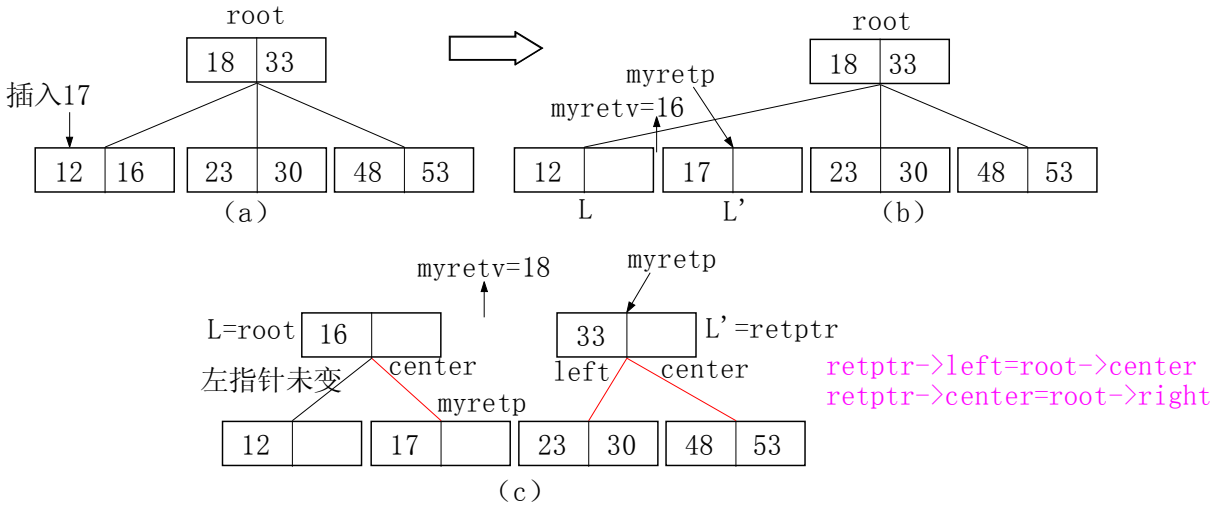


图6.9父节点分裂提升左关键码

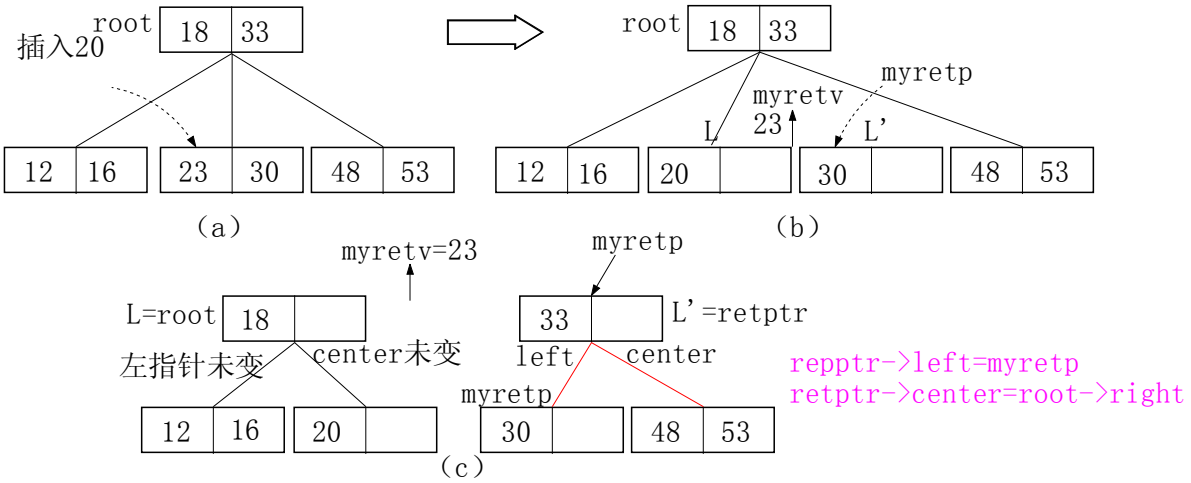


图6.10父节点分裂提升中间关键码

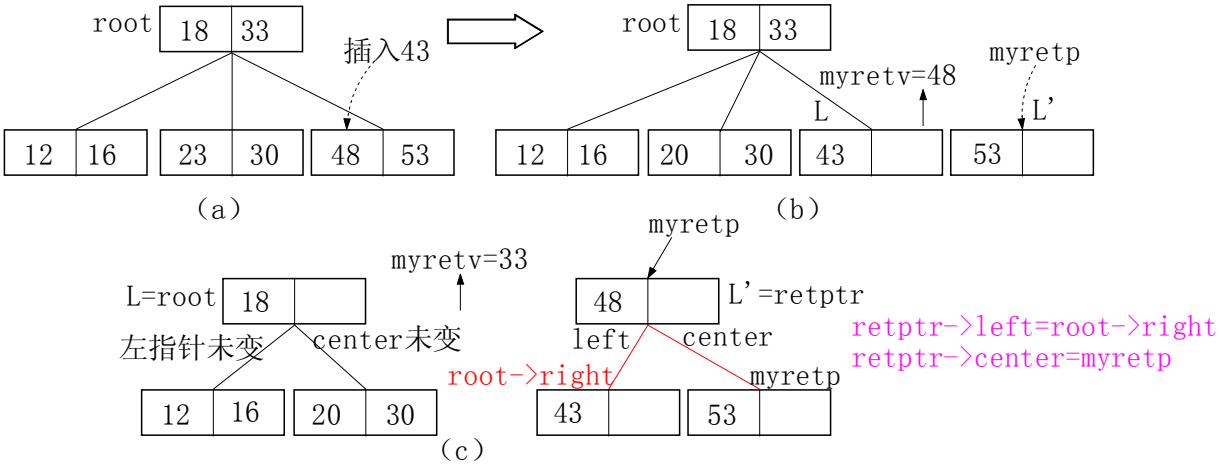


图6.11父节点分裂提升右关键码

关于 2-3 树的删除操作，需要考虑三种情况：①从包含两个关键码的叶节点中删除一个关键码时只需要简单清除即可，不会影响其它节点；②唯一个关键码从叶节点中删除；③从一个内部节点删除一个关键码。后两种情况特别复杂，我们留待 B+树时讨论。

6.4 B+树

6.4.1 B+树定义

定义：一个 m 阶的 B+树具有以下特性：

（1） 根是一个叶节点或者至少有两个子女；

（2） 除了根节点和叶节点以外，每个节点有 m/2 到 m 个子女，存储 m-1 个关键码；

（3） 所有叶节点在树的同一层，因此树总是高度平衡的；

（4） 记录只存储在叶节点，内部节点关键码值只是用于引导检索路径的占位符；

（5） 叶节点用指针联接成一个链表。

（6） 类比于二叉排序树的检索特性。

B^+树的叶节点与内部节点不同的是，叶节点存储实际记录，当作为索引树应用时，就是记录的关键码值与指向记录位置的指针，叶节点存储的信息可能多于或少于 m 个记录。见图 6.12 所示。

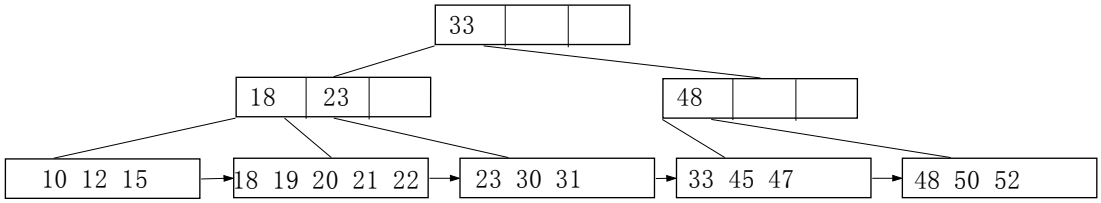


图6.12 4阶 B^+树

B^+树节点结构定义为：

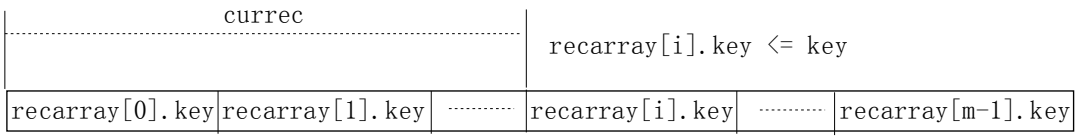
```
Struct Bpnode {
        Struct PAIR recarray[MAXSIZE];//关键码/指针对数组
        int numrec;
        Bpnode *left,*right;
        }
```

其中，PAIR 结构定义为：

```
Struct PAIR {
        int key;
        Struct BPnode *point;
        }
```

因为 point 同时也是指向文件记录的指针，我们需要注意同构问题，这里假设文件记录与节点结构相同，当然，实际是不可能的。此外，这里定义的叶子节点只是存储了指向记录位置的指针与关键码 key，实际上应该是记录的关键码与数据信息（文件名等）。

一个 B^+树的检索函数如例 6.3 所示，子函数 binaryle()调用后返回数组 recarray[]内等于或小于检索关键码值 key 的那个最大关键码的位置偏移。



具有m个子女的B+树节点k的关键码-指针对数组

例 6.3 B^+树检索函数

```
struct BPnode *find(struct BPnode *root,int key)
{
    int currec;
```

```
    currec=binaryle(root->recarray,root->numrec,key);
    if(root->left==Null){//叶子节点
            if(root->recarray[currec].key==key)
                        return root->recarray[currec].point;
            else return Null;
            }
    else find(root->recarray[currec].point,key);
}
```

我们注意到，一个节点的左指针为空时表明到达了叶子节点，从节点结构定义可知，内部节点的左指针应该指向其左子树的根节点，而叶子节点链上的每个节点左指针为空，只有右指针指向其兄弟节点，且链尾右指针亦为空。

6.4.2 B+树插入与删除

- 插入操作过程

一棵 B^+的生长过程见图 6.13 所示，首先找到包含记录的叶子节点，如果叶子未满，则只需简单将关键码（与指向其物理位置的指针）放置到数组中，记录数加一；如果叶子已经满了，则分裂叶子节点为两个，记录在两个节点之间平均分配，然后提升右边节点关键码值最小（数组第一个位置上的记录关键码）的一份拷贝，提升处理过程与 2-3 树一样，可能会形成父节点直至根节点的分裂过程，最终可能让 B^+树增加一层。

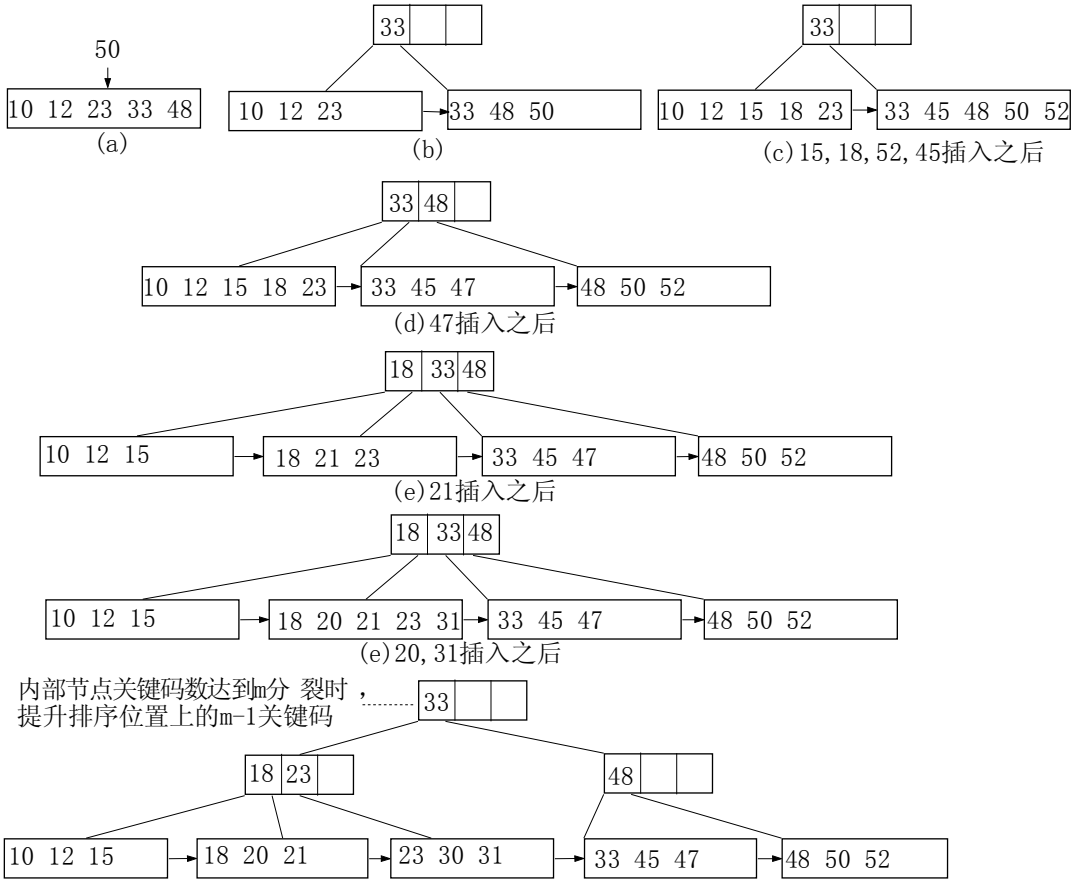


图6.13

- 删除操作过程

在 B^+树中删除一个记录要首先找到包含记录的叶子节点，如果该叶子内的记录数超过 m/2，我们只需简单的清除该记录，因为剩下的记录数至少仍是 m/2。

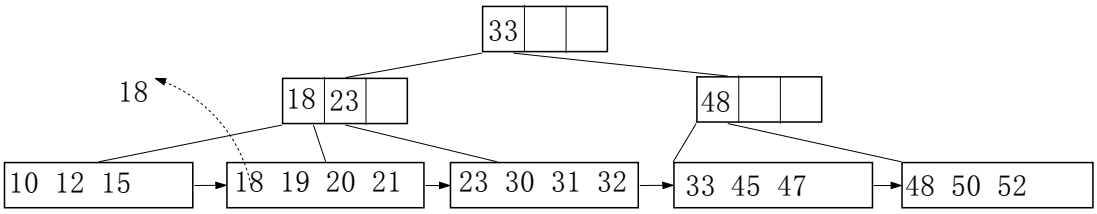
如果一个叶子节点内的记录删除后其余数小于 m/2 则称为下溢，于是我们需要采取如下处理：

（1） 如果它的兄弟节点记录数超过 m/2，可以从兄弟节点中移入，移入数量应让兄弟节点能平分记录数，以避免短时间内再次发生下溢。同时，因为移动后兄弟节点的第一个记录关键码值产生变化，所以需要相应的修改其父节点中的占位符关键码值，以保证占位符指向的节点其第一个关键码值一定是大于或等于该占位符；

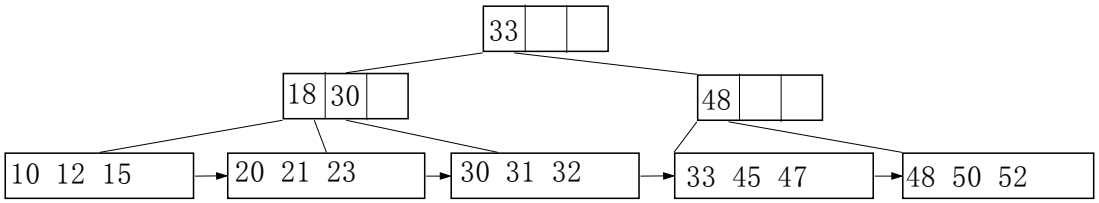
（2） 如果没有左右兄弟节点能移入记录（均小于或等于 m/2），则将当前叶节点的记录移出到兄弟节点，且其和一定小于等于 m，然后将本节点删除。把一个父节点下的两棵子树合并之后，因为要删除父节点中的一个占位符就可能会造成父节点下溢，产生节点合并，并继续引发直至根节点的合并过程，从而树减少一层。

（3） 一对叶节点合并的时候应清除右边节点。

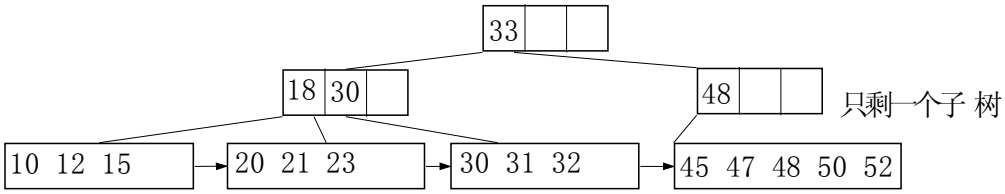
对一棵 B^+树的删除操作过程见图 6.14 所示。



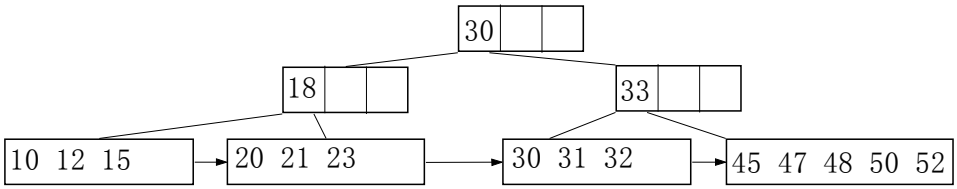
(a)设下限为3，删除关键码值18的记录不影 响占位符 18



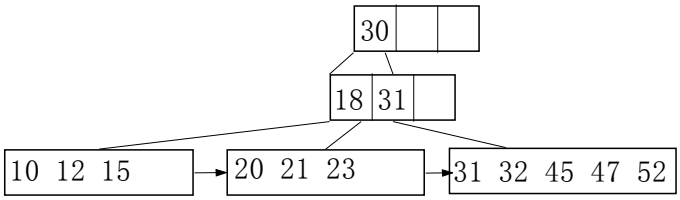
(b)再删除关键码值19的记录之后从兄弟节点移入23并修改符 节点占位符 为30



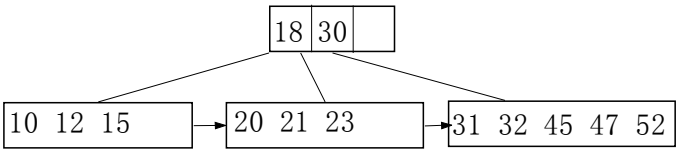
(c)删除关键码值33的记录合并兄弟叶节点



(d)传递叶节点到右子 树 并调 整占位符



(e)删除右子 树 的48，50，30之后叶节点合并引起父节点合并



(f)根节点合并

图6.14

6.4.3 B+树实验设计

设计一个 4 阶 B+树，要求：

(1) 记录应该包括 4 字节（long）关键码值和 60 字节的数据字段(存储文件名等)，设每个叶子可以存储 5 条记录，而内部节点应该是关键码值/指针对。此外，每个节点还应该有指向同层下一个节点的指针、本节点存储的关键码数等；

(2) 此 4 阶 B+树应该支持插入、删除以及根据给定关键码值进行精确检索与关键码范围

检索；

(3) 显示（打印）此 4 阶 B^+树的生长(含删除节点)过程实例；

(4) 有能力的同学增加 4 阶 B^+树的存/取（从硬盘文件）功能。

程序设计时建议注意以下几点：

- 节点结构定义，即内部节点与外部节点同构问题；
- 初始化设置；
- B^+树输入（文件名与关键码）/输出（显示）功能；