China-pub.com
下载

第2章　算法与数据结构

总而言之，只有熟悉了这个领域的工具和技术才能对特殊的问题提供正确解答，只有丰富的经验才能提供坚实的专业性结果。

Raymond Fielding，《特殊效果立体电影的技术》

算法和数据结构的研究是计算机科学的重要基石。这是一个富集优雅技术和复杂数学分析结果的领域。这个领域并不是理论嗜好者们的乐园和游戏的场所：一个好的算法或数据结构可能使某个原来需要用成年累月才能完成的问题在分秒之中得到解决。

在某些特殊领域，例如图形学、数据库、语法分析、数值分析和模拟等等，解决问题的能力几乎完全依赖于最新的算法和数据结构。如果你正要进入一个新领域去开发程序，那么首先需要弄清楚在这里已经有了些什么，以免无谓地把时间浪费在别人早已做好的东西上。

每个程序都要依靠算法与数据结构，但很少有程序依赖于必须发明一批全新的东西。即使是很复杂的程序，比如在编译器或者网络浏览器里，主要的数据结构也是数组、表、树和散列表等等。如果在一个程序里要求某些更精巧的东西，它多半也是基于这些简单东西构造起来的。因此，对大部分程序员而言，所需要的是知道有哪些合适的、可用的算法和数据结构，知道如何在各种可以互相替代的东西之中做出选择。

这里要讲的是一个核桃壳里的故事。实际上基本算法只有屈指可数的几个，它们几乎出现在每个程序中，可能已经被包含在程序库里，这就是基本检索和排序。与此类似，几乎所有的数据结构都是从几个基本东西中产生出来的。这样，本章包含的材料对于每个程序员都是熟悉的。我们写了些能够实际工作的程序，以使下面的讨论更具体。如果需要，你可以抄录这些代码，但要事先检查用的是哪种语言，你是否有它所需要的库。

2.1　检索

要存储静态的表格式数据当然应该用数组。由于可以做编译时的初始化，构建这种数组非常容易(Java的初始化在运行中进行。只要数组不是很大，这就是一个无关紧要的实现细节)。要检查学生练习中是否对某些废话用得太多，在程序里可能会定义：

```
char *flab[] = {
    "actually",
    "just",
    "quite",
    "really",
    NULL
};
```

检索程序必须知道数组里有多少元素。对这个问题的一种处理方法是传递一个数组长度参数。这里采用的是另一种方法，在数组最后放一个 NULL作为结束标志。

```
/* lookup: sequential search for word in array */
int lookup(char *word, char *array[])
```

China-pub.com 下载

```
{
    int i;

    for (i = 0; array[i] != NULL; i++)
        if (strcmp(word, array[i]) == 0)
            return i;
    return -1;
}
```

在C和C++ 里，字符串数组参数可以说明为`char *array[]`或者`char **array`。虽然这两种形式是等价的，但前一种形式能把参数的使用方式更清楚地表现出来。

这里采用的检索算法称为顺序检索，它逐个查看每个数据元素是不是要找的那一个。如果数据的数目不多，顺序检索就足够快了。标准库提供了一些函数，它们可以处理某些特定类型的顺序检索问题。例如，函数`strchr`和`strstr`能在C或C++ 字符串里检索给定的字符或子串，Java的`String`类里有一个`indexOf`方法，C++的类属算法`find`几乎能用于任何数据类型。如果对某个数据类型有这种函数，我们就应该直接用它。

顺序检索非常简单，但是它的工作量与被检索数据的数目成正比。如果要找的数据并不存在，数据量加倍也会使检索的工作量加倍。这是一种线性关系——运行时间是数据规模的线性函数，因此这种检索也被称为线性检索。

下面的程序段来自一个分析 HTML的程序，这里有一个具有实际规模的数组，其中为成百个独立的字符定义了文字名：

```
typedef struct Nameval Nameval;
struct Nameval {
    char    *name;
    int     value;
};

/* HTML characters, e.g. AElig is ligature of A and E. */
/* Values are Unicode/ISO10646 encoding. */

Nameval htmlchars[] = {
    "AElig",    0x00c6,
    "Aacute",   0x00c1,
    "Acirc",    0x00c2,
    /* ... */
    "zeta",     0x03b6,
};
```

对这样的大数组，使用**二分检索**方法效率将更高些。二分检索是人在字典里查单词时采用的一种有条理性的方法：先看位于中间的元素，如果那里的值比想要找的值更大，那么就去查前边的一半；否则就去查后边一半。反复这样做，直到要找的东西被发现，或者确定它根本不存在为止。

为了能做二分检索，表格本身必须是排好序的，就像上面做的那样 (无论如何，这样做都是一种好的风格，人们在排序的表格里找东西也更快一些)。另一方面，程序还必须知道表格的长度，第1章定义的`NELEMS`宏可以用在这里：

```
printf("The HTML table has %d words\n", NELEMS(htmlchars));
```

对这种表格的二分检索函数可以写成下面的样子：

```
/* lookup: binary search for name in tab; return index */
int lookup(char *name, Nameval tab[], int ntab)
{
```

China-pub.com
下载


```
    int low, high, mid, cmp;

    low = 0;
    high = ntab - 1;
    while (low <= high) {
        mid = (low + high) / 2;
        cmp = strcmp(name, tab[mid].name);
        if (cmp < 0)
            high = mid - 1;
        else if (cmp > 0)
            low = mid + 1;
        else    /* found match */
            return mid;
    }
    return -1;  /* no match */
}
```

把这些放在一起，检索htmlchars的操作应写为：

```
half = lookup("frac12", htmlchars, NELEMS(htmlchars));
```

这里要查找的是字符数组下标的½。

二分检索在每个工作步骤中丢掉一半数据。这样，完成检索需要做的步数相当于对表的长度n反复除以2，直至最后剩下一个元素时所做的除法次数。忽略舍入后得到的是 $\log_2 n$。如果被检索的数据有1000个，采用线性检索就要做1000步，而做二分检索大约只要10步。如果被检索的数据项有1百万个，二分检索只需要做20步。可见，项目越多，二分检索的优势也就越明显。超过某个界限后(这个界限因实现不同可能有差别)二分检索一定会比线性检索更快。

2.2 排序

二分检索只能用在元素已经排好序的数组上。如果需要对某个数据集反复进行检索，先把它排序，然后再用二分检索就会是很值得的。如果数据集事先已经知道，写程序时就可以直接将数据排好序，利用编译时的初始化构建数组。如果事先不知道被处理的数据，那么就必须在程序运行中做排序。

最好的排序算法之一是**快速排序**(quicksort)，这个算法是1960年由C. A. R. Hoare发明的。快速排序是尽量避免额外计算的一个极好例子，其工作方式就是在数组中划分出小的和大的元素：

从数组中取出一个元素(基准值)。
把其他元素分为两组：
“小的”是那些小于基准值的元素；
“大的”是那些大于基准值的元素。
递归地对这两个组做排序。

当这个过程结束时，整个数组已经有序了。快速排序非常快，原因是：一旦知道了某个元素比基准值小，它就不必再与那些大的元素进行比较了。同样，大的元素也不必再与小的做比较。这个性质使快速排序远比简单排序算法(如插入排序和起泡排序)快得多。因为在简单排序算法中，每个元素都需要与所有其他元素进行比较。

快速排序是一种实用而且高效的算法。人们对它做了深入研究，提出了许多变形。在这里要展示的大概是其中最简单的一种实现，但它肯定不是最快的。

下面的quicksort函数做整数数组的排序：

China-pub.com 下载

```
/* quicksort: sort v[0]..v[n-1] into increasing order */
void quicksort(int v[], int n)
{
    int i, last;

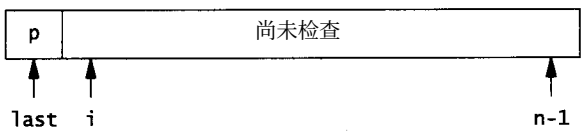
    if (n <= 1)  /* nothing to do */
        return;
    swap(v, 0, rand() % n);    /* move pivot elem to v[0] */
    last = 0;
    for (i = 1; i < n; i++)    /* partition */
        if (v[i] < v[0])
            swap(v, ++last, i);
    swap(v, 0, last);          /* restore pivot */
    quicksort(v, last);        /* recursively sort */
    quicksort(v+last+1, n-last-1); /*    each part */
}
```

函数里的swap操作交换两个元素的值。swap在quicksort里出现三次，所以最好定义为一个单独的函数：

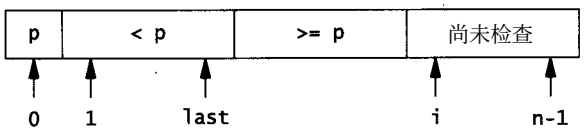
```
/* swap:  interchange v[i] and v[j] */
void swap(int v[], int i, int j)
{
    int temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

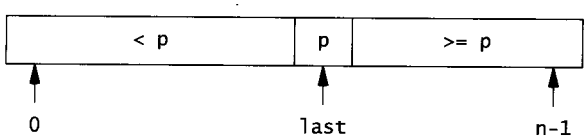
划分动作需要选一个随机元素作为基准值，并将它临时性地交换到最前边。此后，程序扫描其他元素，把小于基准值的元素 (小的)向前面交换(到位置last)，大的元素向后面交换(到位置i)。在这个过程开始时，基准值被放到数组的最前端，last=0且从i=1到n-1的元素都还没有检查，这时的状态是：



在for循环体的开始处，从1到last的元素严格地小于基准值，从last+1到i-1的元素大于或等于基准值，而从i到n-1的元素至今还没有检查过。当v[i]>=v[0]时，算法有可能交换v[i]和它自己，这会浪费了一点时间，但是没什么大影响。



在所有元素都划分完毕后，接着交换位置 0的元素与last处的元素，把基准值放到它的最终位置，这样就维护了正确的顺序。现在数组的样子变成了：



China-pub.com
下载

再把同样的过程用到左边和右边的部分数组上。当这一切都结束时，整个数组的排序就完成了。

快速排序有多快？在最好的情况下：

- 第一遍划分把n个元素分成各有$n/2$个元素的两组。
- 第二层把两个组，每个大约$n/2$个元素，划分为每组大约$n/4$个元素的4组。
- 下一层把有大约$n/4$个元素的4组划分为每组大约$n/8$个元素的8组。
- 如此下去。

这个过程将经历大约$\log_2 n$层。也就是说，在最好的情况下，快速排序算法的总工作量正比于$n+2\times n/2+4\times n/4+8\times n/8\cdots(\log_2 n$项)，也就是$n\log_2 n$。在平均情况下，它花的时间将稍微多一点。计算机领域的人都习惯于使用以2为底的对数，因此我们说排序算法花费的时间正比于$n\log n$。

快速排序的这个实现看起来很清楚。但是它也有另一面，有一个致命的弱点。如果每次对基准值的选择都能将元素划分为数目差不多相等的两组，上面的分析就是正确的。但是，如果执行中经常出现不平均的划分，算法的运行时间就可能接近于按n^2增长。我们在实现中采用随机选取元素作为基准值的方法，减少不正常的输入数据造成过多不平均划分的情况[⊖]。但如果数组里所有的值都一样，这个实现每次都只能切下一个元素，这就会使算法的运行时间达到与n^2成比例。

有些算法的行为对输入数据有很强的依赖性。如果遇到反常的、不好的输入数据，一个平常工作得很好的算法就可能运行极长时间，或者耗费极多存储。就快速排序而言，虽然上面的简单实现有时候确实可能运行得很慢，但采用另外一些更复杂的实现方式，就能把这种病态行为减小到几乎等于0。

2.3 库

C和C++ 的标准库里已经包含了排序函数。它应该能够对付恶劣的输入数据，并运行得尽可能快。

这里的库函数能用于对任何数据类型的排序工作，但是，与此同时，我们写出的程序也必须适应它的界面，这个界面比上面定义的函数要复杂很多。 C函数库中排序函数的名字是`qsort`，在调用`qsort`时必须为它提供一个比较函数，因为在排序中需要比较两个值。由于这里的值可以是任何类型的，比较函数的参数是两个指向被比较数据的 void*指针，在函数里应该把指针强制转换到适当类型，然后提取数据值加以比较，并返回结果 (根据比较中的第一个值小于、等于或者大于第二个值，分别返回负数、零或者正数)。

首先考虑如何实现对字符串数组的排序，这是程序中经常需要做的。下面定义函数 `scmp`，它首先对两个参数做强制转换，然后调用`strcmp`做比较。

```
/* scmp: string compare of *p1 and *p2 */
int scmp(const void *p1, const void *p2)
{
    char *v1, *v2;

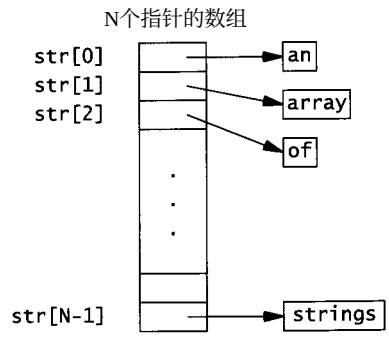
    v1 = *(char **) p1;
    v2 = *(char **) p2;
    return strcmp(v1, v2);
}
```

⊖ 这个说法是错误的，从概率上看，采用随机选取方法并不能减少不平均划分的可能性。 —— 译者

China-pub.com 下载

我们也可以写一个一行的函数，增加临时变量是为了使代码更容易读。

我们不能直接用strcmp作为比较函数，因为qsort传递的是数组里元素的地址，也就是说，是&str[i](类型为char**)而不是str[i](类型为char*)，如下图所示：



如果需要对字符串数组的元素str[0]到str[N-1]排序，那么就应该以这个数组、数组的长度、被排序元素(数组元素)的大小以及比较函数作为参数，调用qsort：

```
char *str[N];

qsort(str, N, sizeof(str[0]), scmp);
```

下面是一个用于比较整数的类似函数icmp：

```
/* icmp: integer compare of *p1 and *p2 */
int icmp(const void *p1, const void *p2)
{
    int v1, v2;

    v1 = *(int *) p1;
    v2 = *(int *) p2;
    if (v1 < v2)
        return -1;
    else if (v1 == v2)
        return 0;
    else
        return 1;
}
```

我们或许想写：

```
    return v1-v2;
```

但在这种情况下，如果v1是很大的正数而v2是大负数，或者相反，这个计算结果就可能溢出，并由此产生不正确的回答。采用直接比较写起来虽然长一点，但是却更安全。

调用qsort同样要用数组、数组长度、被排序元素的大小以及比较函数做为参数：

```
int arr[N];

qsort(arr, N, sizeof(arr[0]), icmp);
```

ANSI C也定义了一个二分检索函数bsearch。与qsort类似，bsearch也要求一个指向比较函数的指针(常用与qsort同样的函数)。bsearch返回一个指针，指向检索到的那个元素；如果没找到有关元素，bsearch返回NULL。下面是用bsearch重写的 HTML 查询函数：

```
/* lookup: use bsearch to find name in tab, return index */
int lookup(char *name, Nameval tab[], int ntab)
{
```

China-pub.com
下载


```
    Nameval key, *np;

    key.name = name;
    key.value = 0;  /* unused; anything will do */
    np = (Nameval *) bsearch(&key, tab, ntab,
                        sizeof(tab[0]), nvcmp);
    if (np == NULL)
        return -1;
    else
        return np-tab;
}
```

与qsort的情况相同，比较函数得到的也是被比较项的地址，所以，这里的 key也必须具有项的类型。在这个例子里，我们必须先专门造出一个 Nameval项，将它传给比较函数。函数nvcmp比较两个Nameval项，方法是对它们的字符串部分调用函数 strcmp，值部分在这里完全不看。

```
/* nvcmp: compare two Nameval names */
int nvcmp(const void *va, const void *vb)
{
    const Nameval *a, *b;

    a = (Nameval *) va;
    b = (Nameval *) vb;
    return strcmp(a->name, b->name);
}
```

这个函数与scmp类似，不同点是这里的字符串是结构的成员。

为bsearch提供一个key就这么费劲，这实际上意味着它的杠杆作用比不上 qsort。写一个好的通用排序程序大概需要一两页代码，而二分检索程序的长度并不比为 bsearch做接口所需要的代码长多少。即使这样，使用 bsearch而不是自己另外写仍然是个好主意。多年的历史证明，程序员能把二分检索程序写正确也是很不容易的。

标准C++库里有一个名字为 sort的类属算法，它保证 $O(n\log n)$的执行性质。使用它的代码非常简单，因为这个函数不需要强制转换，也不需要知道元素的大小。对于已知排序关系的类型，它甚至也不要求显式的比较函数。

```
int arr[N];

sort(arr, arr+N);
```

C++库里也有一个类属的二分检索函数，使用情况也类似。

练习2-1 快速排序用递归方式写起来非常自然。请你用循环把它写出来，并比较这两个版本(Hoare 描述了用循环写快速排序是如何困难，进而发现用递归做快速排序实在是太方便了)。

2.4 一个 Java 快速排序

Java语言的情况有所不同，其早期的发布中没有标准的排序函数，所以在那里就必须自己写。Java更新的版本已经提供了一个sort函数，但是它只能对实现Comparable界面的类使用。但不管怎么说，我们可以用库函数做排序了。由于这里牵涉到的技术在其他地方也很有用，因此，在本节里我们将仔细研究在 Java里实现快速排序的细节。

很容易对quicksort函数做一些改造，使它能处理我们需要排序的类型，但是，更有意义的是写一个类属的排序函数，它具有类似于前面 qsort的界面，而且可以用于任何种类的

China-pub.com 下载

对象。

Java与C、C++ 有一个重要差别：在这里不能把比较函数传递给另一个函数，因为这里没有函数指针。作为其替代物，在这里需要建立一个界面，其中仅有的内容就是一个函数，它完成两个Object的比较。对每个需要排序的数据类型，我们也需要建立一个类，其内容只包括一个成员函数，实现针对这个数据类型的界面。我们将把这个类的一个实例传递给排序函数，排序函数里用这个类中定义的比较函数做元素比较。

我们从定义这个界面开始，将它命名为 Cmp，它的惟一成员就是比较函数 cmp，这个函数做两个Object的比较：

```
interface Cmp {
    int cmp(Object x, Object y);
}
```

此后就可以写实现这个界面的各种具体比较函数了。例如，下面的类里定义的是对 Integer 类型进行比较的函数：

```
// Icmp: Integer comparison
class Icmp implements Cmp {
    public int cmp(Object o1, Object o2)
    {
        int i1 = ((Integer) o1).intValue();
        int i2 = ((Integer) o2).intValue();
        if (i1 < i2)
            return -1;
        else if (i1 == i2)
            return 0;
        else
            return 1;
    }
}
```

下面的类定义了字符串的比较：

```
// Scmp: String comparison
class Scmp implements Cmp {
    public int cmp(Object o1, Object o2)
    {
        String s1 = (String) o1;
        String s2 = (String) o2;
        return s1.compareTo(s2);
    }
}
```

这种方法有一个限制，只能对所有由 Object导出的并带有上面这种机制的类型做排序工作，因此我们定义的排序函数将无法用到各种基本类型，如 int或double上。这就是为什么我们选择对Integer做排序，而不对int做的原因。

有了上面这些东西之后，现在可以把 C语言的quicksort翻译到Java了，这里需要在作为参数的Cmp对象中调用比较函数。另一个最重要的改造是改用下标 left和right，因为Java不允许有指向数组的指针。

```
// Quicksort.sort: quicksort v[left]..v[right]
static void sort(Object[] v, int left, int right, Cmp cmp)
{
    int i, last;
```

China-pub.com
下载


```
        if (left >= right)  // nothing to do
            return;
        swap(v, left, rand(left,right));  // move pivot elem
        last = left;                      //   to v[left]
        for (i = left+1; i <= right; i++) // partition
            if (cmp.cmp(v[i], v[left]) < 0)
                swap(v, ++last, i);
        swap(v, left, last);              // restore pivot elem
        sort(v, left, last-1, cmp);       // recursively sort
        sort(v, last+1, right, cmp);      //   each part
    }
```

Quicksort.sort用cmp比较两个对象，像前面一样调用swap交换对象的位置。

```
    // Quicksort.swap: swap v[i] and v[j]
    static void swap(Object[] v, int i, int j)
    {
        Object temp;

        temp = v[i];
        v[i] = v[j];
        v[j] = temp;
    }
```

随机数由一个函数生成，它产生范围在left到right之间(包括两端)的随机数：

```
    static Random rgen = new Random();

    // Quicksort.rand: return random integer in [left,right]
    static int rand(int left, int right)
    {
        return left + Math.abs(rgen.nextInt())%(right-left+1);
    }
```

这里需要用Math.abs计算绝对值，因为Java随机数生成器的返回值可正可负。

上面这些函数，包括sort、swap和rand，以及生成器对象rgen都是类Quicksort的成员。

最后，如果要调用Quicksort.sort对一个String数组做排序，就应该写：

```
String[] sarr = new String[n];

// fill n elements of sarr...

Quicksort.sort(sarr, 0, sarr.length-1, new Scmp());
```

这里用一个当时建立起来的字符串比较对象调用 sort。

练习2-2 我们的Java快速排序要做许多类型转换工作，把对象从原来类型 (例如Integer)转换为Object，而后又转回。试验Quicksort.sort的另一个版本，它只对某个特定类型做排序，估计由于类型转换带来的性能下降情况。

2.5 大O 记法

我们常需要描述特定算法相对于 n(输入元素的个数)需要做的工作量。在一组未排序的数据中检索，所需的时间与 n成正比；如果是对排序数据用二分检索，花费的时间正比于 $\log n$。排序时间可能正比于n^2或者$n\log n$。

我们需要有一种方式，用它能把这种说法弄得更精确，同时又能排除掉其中的一些具体

China-pub.com 下载

细节，如CPU速度，编译系统(以及程序员)的质量等。我们希望能够比较算法的运行时间和空间要求，并使这种比较能与程序设计语言、编译系统、机器结构、处理器的速度及系统的负载等等复杂因素无关。

为了这个目的，人们提出了一种标准的记法，称为“大 O记法”。在这种描述中使用的基本参数是n，即问题实例的规模，把复杂性或运行时间表达为 n的函数[⊖]。这里的“O”表示量级(order)，比如说“二分检索是$O(\log n)$的”，也就是说它需要“通过$\log n$量级的步骤去检索一个规模为n的数组”。记法$O(f(n))$表示当n增大时，运行时间至多将以正比于 $f(n)$的速度增长。$O(n^2)$和$O(n\log n)$都是具体的例子。这种渐进估计对算法的理论分析和大致比较是非常有价值的，但在实践中细节也可能造成差异。例如，一个低附加代价的 $O(n^2)$算法在n较小的情况下可能比一个高附加代价的$O(n\log n)$算法运行得更快。当然，随着 n足够大以后，具有较慢上升函数的算法必然工作得更快。

我们还应该区分算法的最坏情况的行为和期望行为。要定义好“期望”的意义非常困难，因为它实际上还依赖于对可能出现的输入有什么假定。在另一方面，我们通常能够比较精确地了解最坏情况，虽然有时它会造成误解。前面讲过，快速排序的最坏情况运行时间是 $O(n^2)$，但期望时间是$O(n\log n)$。通过每次都仔细地选择基准值，我们有可能把平方情况 (即$O(n^2)$情况)的概率减小到几乎等于0。在实际中，精心实现的快速排序一般都能以 ($O(n\log n)$时间运行。

下表是一些最重要的情况：

记　法	名　字	例　子
$O(1)$	常数	下标数组访问
$O(\log n)$	对数	二分检索
$O(n)$	线性	字符串比较
$O(n\log n)$	$n\log n$	快速排序
$O(n^2)$	平方	简单排序算法
$O(n^3)$	立方	矩阵乘法
$O(2^n)$	指数	集合划分

访问数组中的元素是常数时间操作，或说 $O(1)$操作。一个算法如果能在每个步骤去掉一半数据元素，如二分检索，通常它就取 $O(\log n)$时间。用`strcmp`比较两个具有n个字符的串需要$O(n)$时间。常规的矩阵乘算法是$O(n^3)$，因为算出每个元素都需要将n对元素相乘并加到一起，所有元素的个数是n^2。

指数时间算法通常来源于需要求出所有可能结果。例如，　n个元素的集合共有 2^n个子集，所以要求出所有子集的算法将是 $O(2^n)$的。指数算法一般说来是太昂贵了，除非 n的值非常小，因为，在这里问题中增加一个元素就导致运行时间加倍。不幸的是，确实有许多问题 (如著名的“巡回售货员问题”)，到目前为止找到的算法都是指数的。如果我们真的遇到这种情况，通常应该用寻找近似最佳结果的算法替代之。

练习2-3　什么样的输入序列可能导致快速排序算法产生最坏情况的行为？设法找出一些情况，使你所用的库提供的排序函数运行得非常慢。设法将这个过程自动化，以便你能很容易描述并完成大量的试验。

练习2-4　设计和实现一个算法，它尽可能慢地对 n个整数的数组做排序。你的算法必须合乎

⊖ 这里的复杂性应该看作是算法的时间或空间开销的一种抽象，并不就是运行时间本身。——译者

China-pub.com
下载

情理，也就是说：算法必须不断前进并能最终结束，实现中不能有欺诈，如浪费时间的循环等。算法的复杂性(作为*n*的函数)是什么？

2.6 可增长数组

前面几节使用的数组都是静态的，它们的大小和内容都在编译时确定了。如果前面用的废话词表或者HTML字符表需要在程序运行中改变，散列表可能就是更合适的结构。向一个 *n* 元排序数组中逐个插入 *n*个元素，使其逐渐增大，这是个 $O(2^n)$的操作。当*n*比较大时这种操作应该尽量避免。

我们常常需要记录一个包含某些东西的变量的变化情况，在这里数组仍然是一种可能选择。为减小重新分配的代价，数组应当以成块方式重新确定大小。为了清楚起见，维护数组所需要的信息必须与数组放在一起。在C++和Java的标准库里可以找到具有这种性质的类。在C里我们可以通过`struct`实现类似的东西。

下面的代码定义了一个元素为`Nameval`类型的可增长数组，新元素将被加到有关数组的最后。在必要时数组将自动增大以提供新的空间。任何元素都可以通过下标在常数时间里访问。这种东西类似于Java和C++库中的向量类。

```
typedef struct Nameval Nameval;
struct Nameval {
    char    *name;
    int     value;
};

struct NVtab {
    int     nval;       /* current number of values */
    int     max;        /* allocated number of values */
    Nameval *nameval;   /* array of name-value pairs */
} nvtab;

enum { NVINIT = 1, NVGROW = 2 };

/* addname: add new name and value to nvtab */
int addname(Nameval newname)
{
    Nameval *nvp;

    if (nvtab.nameval == NULL) { /* first time */
        nvtab.nameval =
            (Nameval *) malloc(NVINIT * sizeof(Nameval));
        if (nvtab.nameval == NULL)
            return -1;
        nvtab.max = NVINIT;
        nvtab.nval = 0;
    } else if (nvtab.nval >= nvtab.max) { /* grow */
        nvp = (Nameval *) realloc(nvtab.nameval,
                (NVGROW*nvtab.max) * sizeof(Nameval));
        if (nvp == NULL)
            return -1;
        nvtab.max *= NVGROW;
        nvtab.nameval = nvp;
    }
    nvtab.nameval[nvtab.nval] = newname;
    return nvtab.nval++;
}
```

China-pub.com 下载

函数addname返回刚加入数组的项的下标，出错的时候返回- 1。

对realloc调用将把数组增长到一个新的规模，并保持已有的元素不变。这个函数返回指向新数组的指针；当存储不够时返回 NULL。这里采用每次调用 realloc时将数组规模加倍的方式，是为了保证复制数组元素的期望代价仍然是常数。如果数组在调用 realloc时一次增长一个元素，那么执行代价就会变成 $O(2n^2)$。由于在重新分配后数组的位置可能改变，程序其他部分对数组元素的访问必须通过下标进行，而不能通过指针。注意，在上面的代码中没采用下面这种方式：

```
? nvtab.nameval = (Nameval *) realloc(nvtab.nameval,
?         (NVGROW*nvtab.max) * sizeof(Nameval));
```

如果采用这种形式，当重新分配失败时原来的数组就会丢失。

我们开始用一个非常小的初值(NVINIT=1)确定数组规模。这迫使程序一定要增长其数组，保证这段程序能够被执行。如果这段代码放在产品里使用，初始值可以改得大一些。当然，由小的初始值引起的代价也是微不足道的。

按说realloc的返回值不必强制到最后类型，因为 C能自动完成对 void*的提升。而C++ 中就不同，在那里必须做类型强制。有人在这里会争辩，究竟是应该做强制 (这样做清晰而认真)还是不做强制(因为强制实际上可能掩盖真正的错误)。我们选择写强制，是因为这种写法能同时适用于C和C++，付出的代价是减少了 C编译器检出错误的可能性。通过允许使用两种编译器，我们也得到了一点补偿。

删除名字需要一点诀窍，因为必须决定怎样填补删除后数组中留下的空隙。如果元素的顺序并不重要，最简单的方法就是把位于数组的最后元素复制到这里。如果还必须保持原有顺序，我们就只能把空洞后面的所有元素前移一个位置：

```
/* delname: remove first matching nameval from nvtab */
int delname(char *name)
{
    int i;

    for (i = 0; i < nvtab.nval; i++)
        if (strcmp(nvtab.nameval[i].name, name) == 0) {
            memmove(nvtab.nameval+i, nvtab.nameval+i+1,
                (nvtab.nval-(i+1)) * sizeof(Nameval));
            nvtab.nval--;
            return 1;
        }
    return 0;
}
```

这个程序里调用memmove，通过把元素向下移一个位置的方法将数组缩短。 memmove是标准库函数，用于复制任意大小的存储区块。

在ANSI C的标准库里定义了两个相关的函数： memcpy的速度快，但是如果源位置和目标位置重叠，它有可能覆盖掉存储区中的某些部分； memmove函数的速度可能慢些，但总能保证复制的正确完成。选择正确性而不是速度不应该是程序员的责任，对这种情况实际上应该只提供一个函数。姑且认为这里只有一个，并总是使用 memmove。

也可以用下面的循环代替程序里对 memmove的调用：

```
int j;
for (j = i; j < nvtab.nval-1; j++)
    nvtab.nameval[j] = nvtab.nameval[j+1];
```

China-pub.com
下载

我们喜欢用memmove，因为这样可以避免人很容易犯的复制顺序错误。例如，如果这里要做的是插入而不是删除，那么循环的顺序就必须反过来，以避免覆盖掉数组元素。通过调用memmove完成工作就不必为这些事操心了。

也可以采用其他方法，不做数组元素移动。例如把被删除数组元素标记为未用的。随后再要插入元素时，首先检索是否存在无用位置，只有在找不到空位时才做数组增长。对于这里的例子，可以采用把名字域置为NULL的方式表示无用的情况。

数组是组织数据的最简单方式。所以大部分语言都提供了有效而方便的下标数组，字符串也用字符的数组表示。数组的使用非常简单，它提供对元素的 $O(1)$访问，又能很好地使用二分检索和快速排序，空间开销也比较小。对于固定规模的数据集合，数组甚至可以在编译时建构起来。所以，如果能保证数据不太多，数组就是最佳选择。但事情也有另一方面，在数组里维护一组不断变化的数据代价很高。所以，如果元素数量无法预计，或者可能会非常多，选择其他数据结构可能就更合适些。

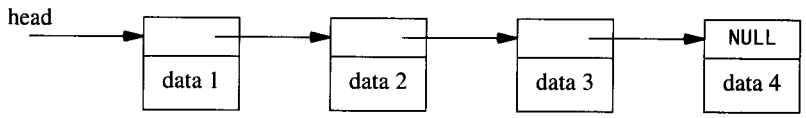
练习2-5 在上面的代码里，delname没有通过调用realloc去返回删除后释放的存储。那样做值得吗？你怎么决定是做这件事还是不做？

练习2-6 对addname和delname做适当修改，通过把删除项标记为未用的来删除元素。如何使程序的其余部分同这种修改隔离开来？

2.7 表

除了数组之外，表(链表)是典型程序里使用最多的数据结构。许多程序设计语言里有系统内部定义的表，有些语言 —— 如Lisp —— 就完全是基于这种结构构造起来的。虽然在 C++和Java里表已经由程序库实现了，我们还是需要知道如何使用以及何时使用它。而在 C语言里我们就必须自己实现。这一节我们准备讨论C的表，从中学到的东西可以用到更广泛的地方去。

一个**单链表**包含一组项，每个项都包含了有关数据和指向下一个项的指针。表的头就是一个指针，它指向第一个项，而表的结束则用空指针表示。下面是一个包含 4个元素的表：



数组和表之间有一些很重要的差别。首先，数组具有固定的大小，而表则永远具有恰好能容纳其所有内容的大小，在这里每个项目都需要一个指针的附加存储开销。第二，通过修改几个指针，表里的各种情况很容易重新进行安排，与数组里需要做大面积的元素移动相比，修改几个指针的代价要小得多。最后，当有某些项被插入或者删除时，其他的项都不必移动。如果把指向一些项的指针存入其他数据结构的元素中，表的修改也不会使这些指针变为非法的[⊖]。

这些情况说明，如果一个数据集合里的项经常变化，特别是如果项的数目无法预计时，表是一种可行的存储它们的方式。经过这些比较，我们容易看到，数组更适合存储相对静态的数据。

对表有一些基本操作：在表的最前面或最后加入一个新项，检索一个特定项，在某个指定项的前面或后面加入一个新项，可能还有删除一个项等等。表的简单性使我们很容易根据需要给它增加一些其他操作。

⊖ 这点不能一概而论，如果被指向的表项(表元素)随后被删除，有关指针就将变为非法的。 —— 译者

China-pub.com 下载

在C里通常不是直接定义表的类型 List，而是从某种元素类型开始，例如 HTML的元素 Nameval，给它加一个指针，以便能链接到下一个元素：

```
typedef struct Nameval Nameval;
struct Nameval {
    char    *name;
    int     value;
    Nameval *next;  /* in list */
};
```

我们无法在编译时初始化一个非空的表，这点也与数组不同。表应该完全是动态构造起来的。首先我们需要有方法来构造一个项。最直接的方式是定义适当的函数，完成有关的分配工作，这里称它为newitem：

```
/* newitem: create new item from name and value */
Nameval *newitem(char *name, int value)
{
    Nameval *newp;

    newp = (Nameval *) emalloc(sizeof(Nameval));
    newp->name = name;
    newp->value = value;
    newp->next = NULL;
    return newp;
}
```

这本书里的许多地方都要使用函数 emalloc，这个函数将调用 malloc，如果分配失败，它报告一个错误并结束程序的执行。函数的定义代码在第 4章给出，现在只要认定 emalloc是个存储分配函数，它绝不会以失败的情况返回。

构造表的最简单而又快速的方法就是把每个元素都加在表的最前面：

```
/* addfront: add newp to front of listp */
Nameval *addfront(Nameval *listp, Nameval *newp)
{
    newp->next = listp;
    return newp;
}
```

当一个表被修改时，结果可能得到另一个首元素，就像 addfront里的情况。对表做更新操作的函数必须返回指向新首元素的指针，这个指针将被存入保持着这个表的变量里。函数addfront和这组函数里的其他函数都返回指向表中首元素的指针，以此作为它们的返回值。函数的典型使用方式是：

```
nvlist = addfront(nvlist, newitem("smiley", 0x263A));
```

这种设计对于原表为空的情况同样可以用，函数也可以方便地用在表达式里。另一种可能的方式是传递一个指针给保存表头的指针。我们这里的方法更自然些。

如果要在表末尾加一个元素，这就是一个 $O(n)$操作，因为函数必须穿越整个表，直到找到了表的末端：

```
/* addend: add newp to end of listp */
Nameval *addend(Nameval *listp, Nameval *newp)
{
    Nameval *p;

    if (listp == NULL)
        return newp;
    for (p = listp; p->next != NULL; p = p->next)
        ;
```

China-pub.com
下载


```
        p->next = newp;
        return listp;
    }
```

如果想把addend做成一个$O(1)$的操作，那么就必须维持另一个独立的指向表尾的指针。除了总需要费心维护表尾指针外，这种做法还有另一个缺点：表再不是由一个指针变量表示的东西了。下面我们将坚持最简单的风格。

要检索具有某个特定名字的项，应该沿着next指针走下去：

```
/* lookup: sequential search for name in listp */
Nameval *lookup(Nameval *listp, char *name)
{
    for ( ; listp != NULL; listp = listp->next)
        if (strcmp(name, listp->name) == 0)
            return listp;
    return NULL;    /* no match */
}
```

这也需要$O(n)$时间，不存在能改进这个限度的一般性方法。即使一个表是排序的，我们也必须沿着表前进，以便找到特定的元素。二分检索完全不能适用于表。

如果要打印表里的所有元素，可以直接写一个函数，它穿越整个表并打印每个元素；要计算表的长度，可以写一个函数穿越整个表，其中使用一个计数器；如此等等。这里再提出另一种解决问题的方式，那就是写一个名为apply的函数，它穿越表并对表的每个元素调用另一个函数。我们可以把apply做得更具一般性，为它提供一个参数，把它传递给apply对表元素调用的那个函数。这样，apply就有了三个参数：一个表、一个将要被作用于表里每个元素的函数以及提供给该函数使用的一个参数：

```
/* apply: execute fn for each element of listp */
void apply(Nameval *listp,
        void (*fn)(Nameval*, void*), void *arg)
{
    for ( ; listp != NULL; listp = listp->next)
        (*fn)(listp, arg);  /* call the function */
}
```

apply的第二个参数是一个函数指针，这个函数有两个参数，返回类型是void。这种标准描述方式在语法上很难看：

```
void (*fn)(Nameval*, void*)
```

这说明了fn是一个指向void函数的指针。也就是说，fn本身是个变量，它将以一个返回值为void的函数的地址作为值。被fn指向的函数应该有两个参数，一个参数的类型是Nameval*，即表的元素类型；另一个是void*，是个通用指针，它将作为fn所指函数的一个参数。

要使用apply，例如打印一个表的元素，我们可以写一个简单的函数，其参数包括一个格式描述串：

```
/* printnv: print name and value using format in arg */
void printnv(Nameval *p, void *arg)
{
    char *fmt;

    fmt = (char *) arg;
    printf(fmt, p->name, p->value);
}
```

China-pub.com 下载

它的调用形式是：

```
apply(nvlist, printnv, "%s: %x\n");
```

为统计表中的元素个数也需要定义一个函数，其特殊参数是一个指向整数的指针，该整数被用作计数器：

```
/* inccounter: increment counter *arg */
void inccounter(Nameval *p, void *arg)
{
    int *ip;

    /* p is unused */
    ip = (int *) arg;
    (*ip)++;
}
```

对这个函数的调用可以是：

```
int n;

n = 0;
apply(nvlist, inccounter, &n);
printf("%d elements in nvlist\n", n);
```

并不是每个表操作都能很方便地以这种方式实现。要销毁一个表就必须特别小心：

```
/* freeall: free all elements of listp */
void freeall(Nameval *listp)
{
    Nameval *next;

    for ( ; listp != NULL; listp = next) {
        next = listp->next;
        /* assumes name is freed elsewhere */
        free(listp);
    }
}
```

当一块存储区被释放之后，程序里就不能再使用它了。因此，在释放 `listp`所指向的元素之前，必须首先把`listp->next`的值保存到一个局部变量(`next`)里。如果把上面的循环写成下面的样子：

```
?     for ( ; listp != NULL; listp = listp->next)
?         free(listp);
```

由于`listp->next`原来的值完全可能被`free`复写掉，这个代码有可能会失败。

注意，函数`freeall`并没有释放`listp->name`，它实际上假定每个`Nameval`的`name`域会在程序中其他地方释放掉，或者它们根本就没有分配过。要保证在项的分配和释放方面的一致性，要求`newitem`和`freeall`之间有一种相互协调。既要保证所有存储都能被释放，又能弄清楚哪些东西没有释放并应该释放，在这两方面之间有一个平衡问题。如果不能正确处理就可能导致程序错误。在另一些语言里，例如在 Java里，有一个废料收集程序，它能帮人做这些事。在第4章讨论资源管理时还要回到这个问题。

如果想从表里删除一个元素，需要做的事情更多一些：

```
/* delitem: delete first "name" from listp */
Nameval *delitem(Nameval *listp, char *name)
{
    Nameval *p, *prev;
```

China-pub.com
下载


```
        prev = NULL;
        for (p = listp; p != NULL; p = p->next) {
            if (strcmp(name, p->name) == 0) {
                if (prev == NULL)
                    listp = p->next;
                else
                    prev->next = p->next;
                free(p);
                return listp;
            }
            prev = p;
        }
        eprintf("delitem: %s not in list", name);
        return NULL;    /* can't get here */
    }
```

与freeall里的处理方式一样，delitem也不释放name域。

函数eprintf显示一条错误信息并终止程序执行，这至多是一种笨办法。想从错误中得体地恢复程序执行是很困难的，需要一段很长的讨论。我们把这个讨论推迟到第 4章，那里还要给出eprintf的实现。

上面是关于基本表结构和有关基本操作的讨论，这里也介绍了在写普通程序时可能遇到的许多重要应用。在这个方面也有许多可能的变化。有些程序库，例如 C++的标准模板库(STL)支持双链表，其中的每个元素包含两个指针，一个指向其后继元素，另一个指向其前驱。双链表需要更多的存储开销，但它也使找最后元素和删除当前元素都变为 *O*(1)操作。有的实现方式为表指针另行分配位置，与被它们链接在一起的数据分开放。这样用起来稍微麻烦一点，但却能允许一个项同时隶属于多个不同的表。

表特别适合那些需要在中间插入和删除的情况，也适用于管理一批规模经常变动的无顺序数据，特别是当数据的访问方式接近后进先出 (LIFO)的情况时(类似于栈的情况)。如果程序里存在多个互相独立地增长和收缩的栈，采用表比用数组能更有效地利用存储。当信息之间有一种内在顺序，就像一些事先不知道长短的链，例如文本中顺序的一系列单词，用表实现也非常合适。如果同时还必须对付频繁的更新和随机访问，那么最好就是使用某些非线性的数据结构，例如树或者散列表等。

练习2-7 实现其他一些表的操作，例如复制、归并、分裂以及在特定元素前面或者后面做插入等。两种不同插入操作在实现的困难性方面有什么差异？前面写的程序代码你可以用多少？哪些操作必须自己来做？

练习2-8 写一个递归的和一个非递归的reverse，它们能把一个表翻转过来。操作中不要建立新的表项，只用已经有的项。

练习2-9 在C里写一个类属的List类型。最简单的方法是令每个表项包含一个 void*指针，它指向有关数据。在C++ 里利用模板，在Java里通过定义一个包含Object类型的表的方式重做这件事。对于这个工作，各种语言有哪些有利条件，又存在哪些弱点？

练习2-10 设计和实现一组测试，验证你写的各种表操作是正确的。第6章将讨论各种测试策略。

2.8 树

树是一种分层性数据结构。在一棵树里存储着一组项，每个项保存一个值，它可以有指

China-pub.com 下载

针指向0个或多个元素，但只能被另一个项所指。树根是其中惟一的例外，没有其他项的指针指向它。

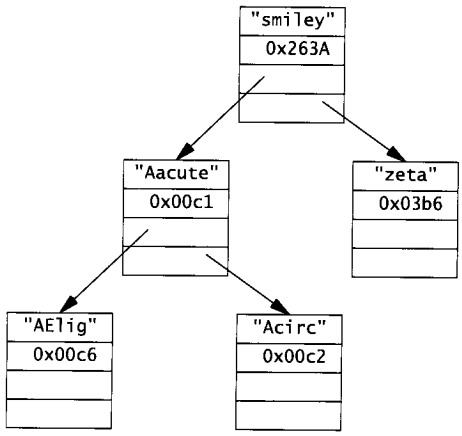
实际存在许多不同种类的树，它们表示各种复杂的结构。例如，语法树表示句子或程序的语法，家族树描述了人们互相间的关系。下面用二分检索树来说明树的原理，这种树最容易实现，也能很好地表现树的一般性质。二分检索树的每个结点带有两个子结点指针，`left`和`right`，它们分别指向对应的子结点。子结点指针可能取空值，当实际子结点少于两个时就会出现这种情况。在二分检索树里，结点中存储的值定义了树结构：对于一个特定结点，其左子树中存储着较小的值，而右子树里存储着较大的值。由于有这个性质，我们可以采用二分检索的一种变形在这种树里检索特定的值，或确定其存在性。

定义`Nameval`的树结点形式也很方便：

```
typedef struct Nameval Nameval;
struct Nameval {
    char    *name;
    int     value;
    Nameval *left;  /* lesser */
    Nameval *right; /* greater */
};
```

这里的`lesser`和`greater`注释指明了有关链接的性质：左子树里存储着较小的值，而右子树里存储较大的值。

作为一个实际例子，下面的图中显示的是一个`Nameval`二分检索树，其中存储着字符名字表的一部分，按照名字的ASCII编码排序。



由于树的许多结点包含多个指向其他元素的指针，所以，很多在表或者数组结构中需要$O(n)$时间的操作，在树中只需要$O(\log n)$时间。结点存在多个指针能减少为寻找一个结点所需要访问的结点个数，从而降低操作的复杂性。

二分检索树(在本节下面将直接称它为“树”)的建构方式是：在树里递归地向下，根据情况确定向左或向右，直到找到了链接新结点的正确位置。结点应该正确地初始化为`Nameval`类型的对象，它包含一个名字、一个值和两个空指针。新结点总是作为叶子加进去的，它没有子结点。

```
/* insert: insert newp in treep, return treep */
Nameval *insert(Nameval *treep, Nameval *newp)
{
```

China-pub.com
下载


```
    int cmp;

    if (treep == NULL)
        return newp;
    cmp = strcmp(newp->name, treep->name);
    if (cmp == 0)
        weprintf("insert: duplicate entry %s ignored",
            newp->name);
    else if (cmp < 0)
        treep->left = insert(treep->left, newp);
    else
        treep->right = insert(treep->right, newp);
    return treep;
}
```

至此我们一直没提重复项的问题。在上面这个insert函数里，对企图向树中加入重复项的情况(cmp==0)将输出一个“抱怨”信息。在表插入函数里没有这样做，因为如果要做这件事，就必须检索整个表，这就把插入由一个 $O(1)$操作变成了$O(n)$操作。对树而言，这个测试完全不需要额外开销。但是另一方面，在出现重复时应该怎么办，数据结构本身并没有清楚的定义。对具体应用，有时可能应该接受重复情况，有时最合理的处理是完全忽略它。

函数weprintf是eprintf的一个变形，它打印错误信息，在输出信息的前面加上前缀词warning。这个函数并不终止程序执行，这一点与 eprintf不同。

在一棵树里，如果从根到叶的每条路径长度大致都相同，这棵树被称为是平衡的[⊖]。在平衡树里做项检索是个 $O(\log n)$操作，因为不同可能性的数目在每一步减少一半，就像在二分检索中那样。

如果项是按照出现的顺序插入一棵树中，那么这棵树就可能成为不平衡的，实际上它完全可能变成极不平衡的。例如元素以排好序的方式出现，上面的代码就会总通过树的一个分支下降，这样产生出来的可能是一个由 right链接起来的表。这样的树将与表结构具有同样的性能问题。如果元素以随机的顺序到达，那么就很难出现这种情况，产生出的树或多或少是平衡的。

要实现某种永远能够保证平衡的树是很复杂的，这也是为什么实际中存在很多不同种类的树的一个原因。对于我们的目的而言，还是先把这个问题放到一边，假定数据本身具有足够的随机性，生成的树是足够平衡的。

对树的lookup操作与insert很像：

```
/* lookup: look up name in tree treep */
Nameval *lookup(Nameval *treep, char *name)
{
    int cmp;

    if (treep == NULL)
        return NULL;
    cmp = strcmp(name, treep->name);
    if (cmp == 0)
        return treep;
    else if (cmp < 0)
```

⊖ 作者希望把“平衡树”概念弄得既不那么难懂，又具有一般性。但是这个说法实际上是错误的。对于只有一个分支的树，所有结点都在该分支上，只有一个叶结点。显然这个树满足作者的定义，但它绝不是一般数据结构意义下的平衡树。可见正文中的定义必须附加其他条件，或改用其他方式。这里提出平衡概念的目的，就是希望由树根到所有叶结点的最长距离与树全部结点个数成对数关系。——译者

China-pub.com 下载

```
        return lookup(treep->left, name);
    else
        return lookup(treep->right, name);
}
```

对于lookup和insert，还有几个情况应该注意。首先，这两个算法看起来非常像本章开始时给出的二分检索算法。这实际上并不奇怪，因为它们的基本思想与二分检索相同，都是“分而治之”，这是产生对数时间性能的根源。

第二，这些程序是递归的。如果将其改写成循环算法，它们就更像二分检索。实际上，从lookup的递归形式通过一个优雅的变换，就可以直接构造出它的循环形式。除了已经找到对应项的情况，lookup的最后一个动作是返回对它自己递归调用的结果，这种情况称为**尾递归**。尾递归可以直接转化为循环，方法是修补有关参数，并重新开始这个过程。最直接的方法是用一个goto语句，更清楚的方法是用一个while循环：

```
/* nrlookup: non-recursively look up name in tree treep */
Nameval *nrlookup(Nameval *treep, char *name)
{
    int cmp;

    while (treep != NULL) {
        cmp = strcmp(name, treep->name);
        if (cmp == 0)
            return treep;
        else if (cmp < 0)
            treep = treep->left;
        else
            treep = treep->right;
    }
    return NULL;
}
```

一旦知道如何在树上穿越，其他的常用操作都很自然了。我们可以采用管理表的某些技术，写一个一般的树遍历器，它对树的每个结点调用另一个函数。但是这次还需要做一些选择，确定什么时候对这个项进行操作，什么时候处理树的其余部分。相应的回答依赖于我们用树表示的是什么东西。如果这里表示的是排序的数据 (例如二分排序树)，那么就该先访问左边一半，然后是右边。有时树的结构反映了数据的某种内在顺序 (例如一棵家族树)，这时对树叶的访问顺序应该根据树所表示的关系确定。

在**中序**遍历中对数据项的操作在访问了左子树之后，并在访问右子树之前：

```
/* applyinorder: inorder application of fn to treep */
void applyinorder(Nameval *treep,
        void (*fn)(Nameval*, void*), void *arg)
{
    if (treep == NULL)
        return;
    applyinorder(treep->left, fn, arg);
    (*fn)(treep, arg);
    applyinorder(treep->right, fn, arg);
}
```

当结点需要按照排序方式顺序处理时，就应该使用这种方式。例如要按序打印树中数据，可以这样做：

```
applyinorder(treep, printnv, "%s: %x\n");
```

China-pub.com
下载

这实际上也提出了一种合理的排序方法：首先把数据项插入一棵树，接着分配一个正确大小的数组，然后用中序遍历方式顺序地把数据存入数组中。

后序遍历在访问了子树之后才对当前结点进行操作：

```
/* applypostorder: postorder application of fn to treep */
void applypostorder(Nameval *treep,
        void (*fn)(Nameval*, void*), void *arg)
{
    if (treep == NULL)
        return;
    applypostorder(treep->left, fn, arg);
    applypostorder(treep->right, fn, arg);
    (*fn)(treep, arg);
}
```

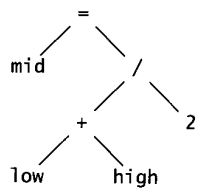
如果对各个结点的操作依赖于对它下面的子树的操作，那么就应该采用后序遍历。在所举的例子中，包含计算一棵树的高度 (取两棵子树的高度中大的一个，再加上 1)，用绘图程序包编排一棵树的图形(在页面上为每个子树安排空间，并将它们组合起来构成本结点的空间)，计算总的存储量，等等。

第三种选择称为前序，实际中很少使用，所以这里不讨论了。

在现实中，二分检索树的使用并不很多。另一种 B树有非常多的分支，它常被用在二级存储的数据组织中。在日常的程序设计里，树的常见用途之一是用于表示语句或表达式结构。例如，语句：

```
mid = (low + high) / 2;
```

可以表示为下面的语法分析树。要对这棵树做求值，只要对它做一次后序遍历，并在每个结点做适当的操作。



在第9章我们还有一大段关于语法分析树的讨论。

练习2-11 比较lookup和nrlookup的执行性能。递归与循环相比耗费高多少？

练习2-12 利用中序遍历方式建立一个排序过程。它的复杂性怎样？在什么条件下它的性能会很差？它的性能与快速排序或库函数比较起来又怎么样？

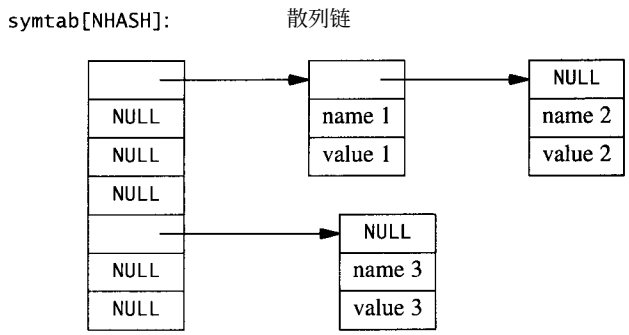
练习2-13 设计并实现一组测试，用它们验证各种树操作函数的正确性。

2.9 散列表

散列表是计算机科学里的一个伟大发明，它是由数组、表和一些数学方法相结合，构造起来的一种能够有效支持动态数据的存储和提取的结构。散列表的一个典型应用是符号表，在一些值(数据)与动态的字符串(关键码)集合的成员间建立一种关联。你最喜欢用的编译系统十之八九是使用了散列表，用于管理你的程序里各个变量的信息。你的网络浏览器可能也很好地使用了一个散列表来维持最近使用的页面踪迹。你与 Internet的连接可能也用到一个散列表，缓存最近使用的域名和它们的 IP地址。

China-pub.com 下载

散列表的思想就是把关键码送给一个散列函数，产生出一个散列值，这种值平均分布在一个适当的整数区间中。散列值被用作存储信息的表的下标。 Java提供了散列表的标准界面。在C和C++ 里，常见做法是为每一个散列值 (或称“桶”)关联一个项的链表，这些项共有这同一个散列值，如下图所示：



在实践中，散列函数应该预先定义好，事先分配好一个适当大小的数组，这些通常在编译时完成。数组的每个元素是一个链表，链接起具有该散列值的所有数据项。换句话说，一个具有*n*个项的散列表是个数组，其元素是一组平均长度为 *n*/(数组大小) 的链表。这里提取一个项是*O*(1)操作，条件是选择了好的散列函数，而且表并不太长[⊖]。

由于散列表是链接表的数组，其基本元素类型与链接表相同：

```
typedef struct Nameval Nameval;
struct Nameval {
    char    *name;
    int     value;
    Nameval *next;      /* in chain */
};

Nameval *symtab[NHASH]; /* a symbol table */
```

在2.7节已经讨论过的链接表技术可以用于维护这里的表。一旦有了一个好的散列函数，随后的事就轻而易举了：直接取得散列桶 (链接表)位置，然后穿越这个链表寻找正确的匹配。下面是为在散列表中做查询 /插入的代码。如果找到了有关项目，函数将它返回。如果找不到，而且放置了create标志，lookup在表中加入一个新项目。与以前一样，这里也不为对应的名字建立新拷贝，假定调用的程序已经建立了安全的拷贝。

```
/* lookup: find name in symtab, with optional create */
Nameval* lookup(char *name, int create, int value)
{
    int h;
    Nameval *sym;

    h = hash(name);
    for (sym = symtab[h]; sym != NULL; sym = sym->next)
        if (strcmp(name, sym->name) == 0)
            return sym;
    if (create) {
        sym = (Nameval *) emalloc(sizeof(Nameval));
```

⊖ 文中说在这些条件下数据提取是一个 *O*(1)操作，在理论上看这种说法是不对的。但作者的本意是，在这些条件下操作效率很高，这个意思是很清楚的。—— 译者

China-pub.com
下载


```
        sym->name = name; /* assumed allocated elsewhere */
        sym->value = value;
        sym->next = symtab[h];
        symtab[h] = sym;
    }
    return sym;
}
```

把查询和可能的插入操作组合在一起，这也是很常见的情况。如果不这样做，常常会出现许多重复性工作。例如：

```
if (lookup("name") == NULL)
    additem(newitem("name", value));
```

在这里散列就计算了两次。

数组到底应该取多大？普遍的想法是要求它足够大，使每个链表至多只有几个元素，以保证查询能够是$O(1)$操作。例如，一个编译程序用的数组可能有数千项，因为一个大的源文件可能有几千行，估计每行代码中不会有多于一个的新标识符。

我们现在必须考虑散列函数hash应该计算出什么东西。这个函数必须是确定性的，应该能算得很快，应该把数据均匀地散布到数组里。对于字符串，最常见的散列算法之一就是：逐个把字节加到已经构造的部分散列值的一个倍数上。乘法能把新字节在已有的值中散开来。这样，最后结果将是所有输入字节的一种彻底混合。根据经验，在对 ASCII串的散列函数中，选择31和37作为乘数是很好的。

```
enum { MULTIPLIER = 31 };

/* hash: compute hash value of string */
unsigned int hash(char *str)
{
    unsigned int h;
    unsigned char *p;

    h = 0;
    for (p = (unsigned char *) str; *p != '\0'; p++)
        h = MULTIPLIER * h + *p;
    return h % NHASH;
}
```

在这个计算中用到了无符号字符。这样做的原因是， C和C++对于char是不是有符号数据没有给出明确定义。而我们需要散列函数总返回正值。

散列函数的返回值已经根据数组的大小取模。如果散列函数能把关键码均匀地散开，那么有关数组到底有多大在这里就不必特别关心了。但是，实际上我们很难确定散列函数具有真正的独立性。进一步说，即使是最好的函数在遇到某些输入集合时也会有麻烦。考虑到这些情况，用素数作为数组的大小是比较明智的，因为这样能保证在数组大小、散列的乘数和可能的数据值之间不存在公因子。

经验表明，对于范围广泛的字符串，要想构造出一个真正能比上面给出的这个更好的散列函数，那是非常困难的。但是要给出一个比它差的就容易多了。 Java的一个早期版本里有一个字符串散列函数，它对长字符串的效率比较高。该函数节约时间的方法是：对那些长于16个字符的串，它只从开头起按照固定间隔检查 8个或9个字符。不幸的是，虽然这个散列函数稍微快一点，它的统计性质却比较差，完全抵消了性能上的优点。由于在散列中要跳过一

China-pub.com 下载

些片段，该函数常常忽略掉字符串中仅有的某些特征部分。实际上文件名字常有很长的共同前缀——目录名等，有的互相间只有最后几个字符不同（如.java和.class等）。URL的开头通常都是http://www.，结束都是.html，它们的差异只是在中间。如果一个散列函数经常只检查到名字里那些不变化的部分，结果就会造成很长的散列链，并使检索速度变慢。Java的这个问题后来解决了，用的就是与我们上面给出的散列函数等价的函数（用乘数37），操作中还是检查字符串里的每个字符。

对一类输入集合(例如短的变量名字)工作得非常好的散列函数，也可能对另一类输入集合(例如URL)工作得很差。所以，对一个散列函数，应该用各种各样的典型输入集合做一些测试。例如，它对于短小的字符串散列得好不好？对于长的串如何？对于长度相同但有微小差别的串怎么样？

字符串并不是能被散列的惟一东西。我们也可以散列在物理模拟中表示粒子位置的三个坐标，以减少对存储需求，用一个线性的表（O(粒子数目)）代替一个三维的数组（其大小是O(xsize×ysize×zsize)。

散列技术有一个很值得一提的例子，那就是Gerand Holzman为分析通信规程和并发系统而开发的Supertrace程序。Supertrace取得被分析系统每个可能状态的全部信息，将这种信息散列到存储器中单个二进制位的地址上。如果一个位已置位，那就表示对应状态已经检查过，否则就是没检查过。Supertrace使用一个几兆字节的散列表，而其中的每个桶只有一位长。这里没有链，如果两个状态发生冲突（它们具有相同的散列值），程序根本就是视而不见。Supertrace的工作依赖于冲突的概率很低(不必是0，因为Supertrace采用的是概率模型，而不是精确模型)。因此，这里用的散列函数必须特别小心。Supertrace采用一种循环冗余检查方式，函数从数据出发做了彻底的混合。

要实现符号表，采用散列表结构是最好的，因为它对元素访问提供了一个$O(1)$的期望性能。散列表也有一些缺点。如果散列函数不好，或者所用的数组太小，其中的链接表就可能变得很长。由于这些链接表没有排序，得到的将是$O(n)$操作。即使元素排了序也无法直接访问它们。但是这后一个问题比较容易对付：可以分配一个数组，在里面存储指向各个元素的指针，然后对它做排序。总之，散列表如果使用得当，常数时间的检索、插入和删除操作是任何其他技术都望尘莫及的。

练习2-14 我们的散列函数对字符串而言是个极好的散列函数，但是，某些特殊数据也可能使它表现欠佳。请设法构造出能使这个散列函数性能极差的数据集。对于NHASH的不同值，要找到这样的坏数据集合很容易吗？

练习2-15 写一个函数，以某种顺序访问散列表里所有的元素。

练习2-16 修改lookup函数，使得当链表的平均长度大于某个x值时，数组将按照y的某个因子扩大，并重新构造整个散列表。

练习2-17 设计一个散列函数，用它存储二维点的坐标。如果要改变坐标的类型，你的函数很容易修改吗？例如从整数坐标改为浮点数坐标，从笛卡儿坐标改为极坐标，或者从二维改到高维？

2.10 小结

选择算法有几个步骤。首先，应参考所有可能的算法和数据结构，考虑程序将要处理的

China-pub.com
下载

数据大概有多少。如果被处理数据的量不大，那么就选择最简单的技术。如果数据可能增长，请删掉那些不能对付大数据集合的东西。然后，如果有库或者语言本身的特征可以使用，就应该使用。如果没有，那么就写或者借用一个短的、简单的和容易理解的实现。如果实际测试说明它太慢，那么就需要改用某种更高级的技术。

虽然实际存在许多不同的数据结构，某些结构在一些特殊环境中对于达到较高性能是至关重要的，但是一般程序员主要使用的仍然是数组、链表、树和散列表。这些结构中的每一个都支持一组基本操作，通常包括建立新元素，寻找一个元素，在某处增加一个元素，或许还有删除一个元素，以及把某个操作作用于所有的元素，等等。

各种操作都具有一定的期望计算时间，这常常决定了一个数据结构 (一个实现)是否适于某种特定应用。数组支持在常数时间里访问任何元素，但不能方便地增长或缩短。链表对于插入、删除可以很好地适应，但对它做随机元素访问要求 $O(n)$的时间。树与散列表提供了好的折衷，既支持对特定元素的快速访问，又支持方便的增长，条件是只要能维持好某种平衡性。

对于某些特殊问题，还存在一些更复杂的数据结构。但是，上面这个基本集合对构造绝大部分软件而言，已经完全够用了。

补充阅读

Bob Sedgewick的一套“算法”丛书(Algorithms，Addison-Wesley)是查找有用算法的绝好去处，其处理也很容易理解。《C++算法》(Algorithms in C++)的第3版有一段关于散列函数和表大小的讨论非常有意思。 Don Knuth的《计算机程序设计技巧》 (The Art of Computer Programming，Addison-Wesley)永远是对许多算法的严格分析的信息源，它的第 3卷(第 2 版，1998)专门讨论排序和检索。

Gerard Holzman的《计算机规程的设计与验证》 (Design and Validation of Computer Protocols，Prentice Hall，1991)里讨论了Supertrace的问题。

Jon Bentley和Doug McIlroy在文章“Engineering a sort function”(Software——Practice and Experience，**23**，1，pp.1249~1265，1993)里讨论了如何构造出一个高速的、强壮的快速排序程序。