

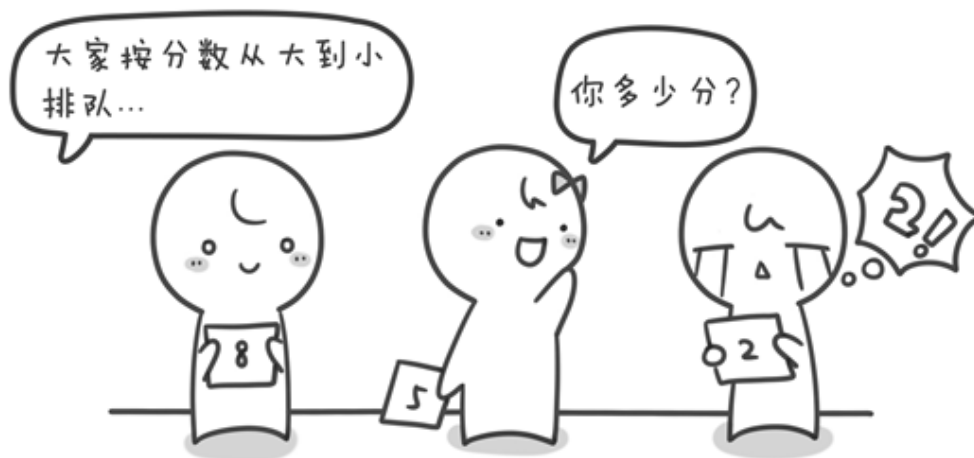
【坐在马桶上看算法】算法1：最快最简单的排序——桶排序

【啊哈！算法】

在我们生活的这个世界中到处都是被排序过的。站队的时候会按照身高排序，考试的名次需要按照分数排序，网上购物的时候会按照价格排序，电子邮箱中的邮件按照时间排序...总之很多东西都需要排序，可以说排序是无处不在。现在我们举个具体的例子来介绍一下排序算法。

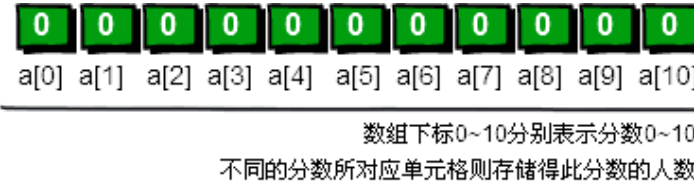


首先出场的我们的主人公小哼，上面这个可爱的娃就是啦。期末考试完了老师要将同学们的分数按照从高到低排序。小哼的班上只有5个同学，这5个同学分别考了5分、3分、5分、2分和8分，哎考的真是惨不忍睹（满分是10分）。接下来将分数进行从大到小排序，排序后是8 5 5 3 2。你有没有什么好方法编写一段程序，让计算机随机读入5个数然后将这5个数从大到小输出？请先想一想，至少想15分钟再往下看吧(*^__^*)。



我们这里只需借助一个一维数组就可以解决这个问题。请确定你真的仔细想过再往下看哦。

首先我们需要申请一个大小为11的数组int a[11]。OK现在你已经有了11个变量，编号从a[0]~a[10]。刚开始的时候，我们将a[0]~a[10]都初始化为0，表示这些分数还都没有人得过。例如a[0]等于0就表示目前还没有人得过0分，同理a[1]等于0就表示目前还没有人得过1分.....a[10]等于0就表示目前还没有人得过10分。



下面开始处理每一个人的分数，第一个人的分数是5分，我们就将相对应a[5]的值在原来的基础增加1，即将a[5]的值从0改为1，表示5分出现过了一次。



第二个人的分数是3分，我们就把相对应a[3]的值在原来的基础上增加1，即将a[3]的值从0改为1，表示3分出现过了一次。



注意啦！第三个人的分数也是“5分”，所以a[5]的值需要在此基础上再增加1，即将a[5]的值从1改为2。表示5分出现过两次。



按照刚才的方法处理第四个和第五个人的分数。最终结果就是下面这个图啦。



你发现没有，a[0]~a[10]中的数值其实就是0分到10分每个分数出现的次数。接下来，我们只需要将出现过的分数打印出来就可以了，出现几次就打印几次，具体如下。

a[0]为0，表示“0”没有出现，不打印。

a[1]为0，表示“1”没有出现，不打印。

a[2]为1，表示“2”出现过1次，打印2。

a[3]为1，表示“3”出现过1次，打印3。

a[4]为0，表示“4”没有出现，不打印。

a[5]为2，表示“5”出现过2次，打印5 5。

a[6]为0，表示“6”没有出现，不打印。

a[7]为0，表示“7”没有出现，不打印。

a[8]为1，表示“8”出现过1次，打印8。

a[9]为0，表示“9”没有出现，不打印。

a[10]为0，表示“10”没有出现，不打印。
最终屏幕输出“2 3 5 5 8”，完整的代码如下。



```
#include <stdio.h>
int main()
{
    int a[11],i,j,t;
    for(i=0;i<=10;i++)
        a[i]=0;    //初始化为0

    for(i=1;i<=5;i++)    //循环读入5个数
    {
        scanf("%d",&t);    //把每一个数读到变量t中
        a[t]++;    //进行计数
    }

    for(i=0;i<=10;i++)    //依次判断a[0]~a[10]
        for(j=1;j<=a[i];j++)    //出现了几次就打印几次
            printf("%d ",i);

    getchar();getchar();
    //这里的getchar();用来暂停程序，以便查看程序输出的内容
    //也可以用system("pause");等来代替
    return 0;
}
```



输入数据为

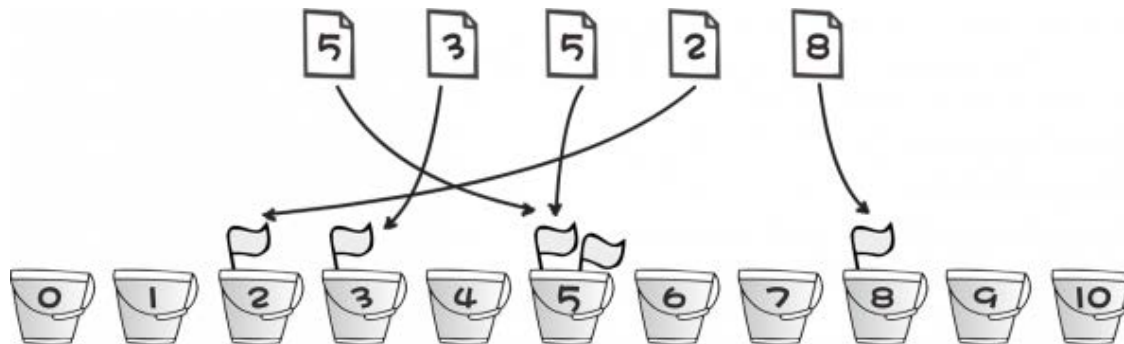
5 3 5 2 8

仔细观察的同学会发现，刚才实现的是从小到大排序。但是我们要求是从大到小排序，这该怎么办呢？还是先自己想一想再往下看哦。

其实很简单。只需要将for(i=0;i<=10;i++)改为for(i=10;i>=0;i--)就OK啦，快去试一试吧。

这种排序方法我们暂且叫他“桶排序”。因为其实真正的桶排序要比这个复杂一些，以后再详细讨论，目前此算法已经能够满足我们的需求了。

这个算法就好比有11个桶，编号从0~10。每出现一个数，就将对应编号的桶中的放一个小旗子，最后只要数数每个桶中有几个小旗子就OK了。例如2号桶中有1个小旗子，表示2出现了一次；3号桶中有1个小旗子，表示3出现了一次；5号桶中有2个小旗子，表示5出现了两次；8号桶中有1个小旗子，表示8出现了一次。



现在你可以尝试一下输入n个0~1000之间的整数，将他们从大到小排序。提醒一下如果需要对数据范围在0~1000之间的整数进行排序，我们需要1001个桶，来表示0~1000之间每一个数出现的次数，这一点一定要注意。另外此处的每一个桶的作用其实就是“标记”每个数出现的次数，因此我喜欢将之前的数组a换个更贴切的名字book（book这个单词有记录、标记的意思），代码实现如下。



```
#include <stdio.h>
int main()
{
    int book[1001],i,j,t,n;
    for(i=0;i<=1000;i++)
        book[i]=0;
    scanf("%d",&n); //输入一个数n，表示接下来有n个数
    for(i=1;i<=n;i++) //循环读入n个数，并进行桶排序
    {
        scanf("%d",&t); //把每一个数读到变量t中
        book[t]++; //进行计数，对编号为t的桶放一个小旗子
    }
    for(i=1000;i>=0;i--) //依次判断编号1000~0的桶
        for(j=1;j<=book[i];j++) //出现了几次就将桶的编号打印几次
            printf("%d ",i);

    getchar();getchar();
    return 0;
}
```



可以输入以下数据进行验证

```
10
8 100 50 22 15 6 1 1000 999 0
```

运行结果是

```
1000 999 100 50 22 15 8 6 1 0
```

最后来说下时间复杂度的问题。代码中第6行的循环一共循环了m次（m为桶的个数），第9行的代码循环了n次（n为待排序数的个数），第14和15行一共循环了m+n次。所以整个排序算法一共执行了m+n+m+n次。我们用大写字母O来表示时间复杂度，因此该算法的时间复杂度是O(m+n+m+n)即O(2*(m+n))。我们在说时间复杂度时候可以忽略较小的常数，最终桶排序的时间复杂度为O(m+n)。还有一点，在表示时间复杂度的时候，n和m通常用大写字母即O(M+N)。

这是一个非常快的排序算法。桶排序从1956年就开始被使用，该算法的基本思想是由E.J.Issac和R.C.Singleton提出来。之前我有说过，其实这并不是真正的桶排序算法，真正的桶排序算法要比这个更加复杂。但是考虑到此处是算法讲解的第一篇，我想还是越简单易懂越好，真正的桶排序留在以后再聊吧。需要说明一点的是：我们目前学习的简化版桶排序算法其本质上还不能算是一个真正意义上的排序算法。为什么呢？例如遇到下面这个例子就没辙了。

现在分别有5个人的名字和分数：huhu 5分、haha 3分、xixi 5分、hengheng 2分和gaoshou 8分。请按照分数从高到低，输出他们的名字。即应该输出gaoshou、huhu、xixi、haha、hengheng。发现问题了没有？如果使用我们刚才简化版的桶排序算法仅仅是把分数进行了排序。最终输出的也仅仅是分数，但没有对人本身进行排序。也就是说，我们现在并不知道排序后的分数原本对应着哪一个人！这该怎么办呢？不要着急请看下节——[冒泡排序](#)。

码字不容易啊，转载请标明出处^_^

【一周一算法】算法1：最快最简单的排序——桶排序

<http://bbs.ahalei.com/thread-4399-1-1.html>(出处: 啊哈磊_编程从这里起步)

分类: [啊哈！算法](#)

标签: 排序算法 桶排序

绿色通道：

好文要顶

已关注

收藏该文

与我联系



啊哈磊

关注 - 0

粉丝 - 181

我在关注他 [取消关注](#)

16

推荐

0

反对

(请您对文章做出评价)

» 下一篇: [【坐在马桶上看算法】算法2：邻居好说话：冒泡排序](#)

【坐在马桶上看算法】算法2：邻居好说话：冒泡排序

【啊哈！算法】

简化版的桶排序不仅仅有上一节所遗留的问题，更要命的是：它非常浪费空间！例如需要排序数的范围是0~2100000000之间，那你则需要申请2100000001个变量，也就是说要写成int a[2100000001]。因为我们需要用2100000001个“桶”来存储0~2100000000之间每一个数出现的次数。即便只给你5个数进行排序（例如这5个数是1，1912345678，2100000000，18000000和912345678），你也仍然需要2100000001个“桶”，这真是太浪费了空间了！还有，如果现在需要排序的不再是整数而是一些小数，比如将5.56789，2.12，1.1，3.123，4.1234这五个数进行从小大排序又该怎么办呢？现在我们来学习另一种新的排序算法：冒泡排序。它可以很好的解决这两个问题。

冒泡排序的基本思想是：每次比较两个相邻的元素，如果他们的顺序错误就把他们交换过来。

例如我们需要将12 35 99 18 76这5个数进行从大到小进行排序。既然是从大到小排序也就是说越小的越靠后，你是不是觉得我在说废话，但是这句话很关键(∩_∩)。

首先比较第1位和第2位的大小，现在第1位是12，第2位是35。发现12比35要小，因为我们希望越小越靠后嘛，因此需要交换这两个数的位置。交换之后这5个数的顺序是35 12 99 18 76。

按照刚才的方法，继续比较第2位和第3位的大小，第2位是12，第3位是99。12比99要小，因此需要交换这两个数的位置。交换之后这5个数的顺序是35 99 12 18 76。

根据刚才的规则，继续比较第3位和第4位的大小，如果第3位比第4位小，则交换位置。交换之后这5个数的顺序是35 99 18 12 76。

最后，比较第4位和第5位。4次比较之后5个数的顺序是35 99 18 76 12。

经过4次比较后我们发现最小的一个数已经就位（已经在最后一位，请注意12这个数的移动过程），是不是很神奇。现在再来回忆一下刚才比较的过程。每次都是比较相邻的两个数，如果后面的数比前面的数大，则交换这两个数的位置。一直比较下去直到最后两个数比较完毕后，最小的数就在最后一个了。就如同是一个气泡，一步一步往后“翻滚”，直到最后一位。所以这个排序的方法有一个很好听的名字“冒泡排序”。



说道这里其实我们的排序只将5个数中最小的一个归位了。每将一个数归位我们将其称为“一趟”。下面我们将继续重复刚才的过程，将剩下的4个数——归位。

好现在开始“第二趟”，目标是将第2小的数归位。首先还是先比较第1位和第2位，如果第1位比第2位小，则交换位置。交换之后这5个数的顺序是99 35 18 76 12。接下来你应该都会了，依次比较第2位和第3位，第3位和第4位。注意此时已经不需要再比较第4位和第5位。因为在第一趟结束后已经可以确定第5位上放的是最小的了。第二趟结束之后这5个数的顺序是99 35 76 18 12。

“第三趟”也是一样的。第三趟之后这5个数的顺序是99 76 35 18 12。

现在到了最后一趟“第四趟”。有的同学又要问了，这不是已经排好了吗？还要继续？当然，这里纯属巧合，你可以用别的数试一试可能就不是了。你能找出这样的数据样例来吗？请试一试。

“冒泡排序”原理是：每一趟只能确定将一个数归位。即第一趟只能确定将末位上的数（既第5位）归位，第二趟只能将倒数第2位上的数（既第4位）归位，第三趟只能将倒数第3位上的数（既第3位）归位，而现在前面还有两个位置上的数没有归位，因此我们仍然需要进行“第四趟”。

“第四趟”只需要比较第1位和第2位的大小。因为后面三个位置上的数归位了，现在第1位是99，第2位是76，无需交换。这5个数的顺序不变仍然是99 76 35 18 12。到此排序完美结束了，5个数已经有4个数归位，那最后一个数也只能放在第1位了。

最后我们总结一下：如果有n个数进行排序，只需将n-1个数归位，也就是说要进行n-1趟操作。而“每一趟”都需要从第1位开始进行相邻两个数的比较，将较小的一个数放在后面，比较完后向后挪一位继续比较下面两个相邻数的大小，重复此步骤，直到最后一个尚未归位的数，已经归位的数则无需再进行比较（已经归位的数你还比较个啥，浪费表情）。

这个算法是不是很强悍。记得我每次拍集体照的时候就总是被别人换来换去的，当时特别烦。不知道发明此算法的人当时的灵感是否来源于此。罗里吧嗦地说了这么多，下面是代码。建议先自己尝试去实现一下看看，再来看我是如何实现的。



```
#include <stdio.h>
int main()
{
    int a[100],i,j,t,n;
    scanf("%d",&n); //输入一个数n，表示接下来有n个数
    for(i=1;i<=n;i++) //循环读入n个数到数组a中
        scanf("%d",&a[i]);
    //冒泡排序的核心部分
    for(i=1;i<=n-1;i++) //n个数排序，只用进行n-1趟
    {
        for(j=1;j<=n-i;j++) //从第1位开始比较直到最后一个尚未归位的数，想一想为什么到n-i就可以了。
        {
            if(a[j]<a[j+1]) //比较大小并交换
            { t=a[j]; a[j]=a[j+1]; a[j+1]=t; }
        }
    }
    for(i=1;i<=n;i++) //输出结果
        printf("%d ",a[i]);
}
```

```
    getchar();getchar();
    return 0;
}
```



可以输入以下数据进行验证

```
10
8 100 50 22 15 6 1 1000 999 0
```

运行结果是

```
0 1 6 8 15 22 50 100 999 1000
```

将上面代码稍加修改，就可以解决第1节遗留的问题，如下。



```
#include <stdio.h>
struct student
{
    char name[21];
    char score;
}; //这里创建了一个结构体用来存储姓名和分数
int main()
{
    struct student a[100],t;
    int i,j,n;
    scanf("%d",&n); //输入一个数n
    for(i=1;i<=n;i++) //循环读入n个人名和分数
        scanf("%s %d",a[i].name,&a[i].score);
    //按分数从高到低进行排序
    for(i=1;i<=n-1;i++)
    {
        for(j=1;j<=n-i;j++)
        {
            if(a[j].score<a[j+1].score)//对分数进行比较
            {
                t=a[j]; a[j]=a[j+1]; a[j+1]=t;
            }
        }
    }
    for(i=1;i<=n;i++)//输出人名
        printf("%s\n",a[i].name);
    getchar();getchar();
}
```



```
return 0;
}
```



可以输入以下数据进行验证

```
5
huhu 5
haha 3
xixi 5
hengheng 2
gaoshou 8
```

运行结果是

```
gaoshou
huhu
xixi
haha
hengheng
```

冒泡排序的核心部分是双重嵌套循环。不难看出冒泡排序的时间复杂度是 $O(N^2)$ 。这是一个非常高的时间复杂度。冒泡排序早在1956年就有人开始研究，之后有很多人都尝试过对冒泡排序进行改进，但结果却令人失望。如Knuth (Donald E. Knuth中文名为高德纳，1974年图灵奖获得者) 所说：“冒泡排序除了它迷人的名字和导致了某些有趣的理论问题这一事实之外，似乎没有什么值得推荐的。” 你可能要问：那还有没有更好的排序算法呢？请期待下周更新——快速排序。

码字不容易啊，转载请标明出处^_^

【一周一算法】算法2：邻居好说话——冒泡排序

<http://bbs.ahalei.com/thread-4400-1-1.html> (出处: 啊哈磊_编程从这里起步)

分类: [啊哈！算法](#)

标签: [排序算法](#), [冒泡排序](#)

绿色通道：

好文要顶

已关注

收藏该文

与我联系



啊哈磊

关注 - 0

粉丝 - 181

我在关注他 [取消关注](#)

1

推荐

0

反对

(请您对文章做出评价)

« 上一篇：[【坐在马桶上看算法】算法1：最快最简单的排序——桶排序](#)

» 下一篇：[【坐在马桶上看算法】算法3：最常用的排序——快速排序](#)

posted @ 2014-02-24 01:25 [啊哈磊](#) 阅读(1279) 评论(6) [编辑](#) [收藏](#)

【坐在马桶上看算法】算法3：最常用的排序——快速排序

【啊哈！算法】系列

上一节的冒泡排序可以说是我们学习第一个真正的排序算法，并且解决了桶排序浪费空间的问题，但在算法的执行效率上却牺牲了很多，它的时间复杂度达到了 $O(N^2)$ 。假如我们的计算机每秒钟可以运行10亿次，那么对1亿个数进行排序，桶排序则只需要0.1秒，而冒泡排序则需要1千万秒，达到115天之久，是不是很吓人。那有没有既不浪费空间又可以快一点的排序算法呢？那就是“快速排序”啦！光听这个名字是不是就觉得很高端呢。

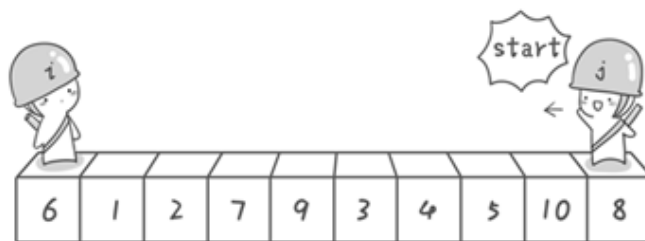
假设我们现在对“6 1 2 7 9 3 4 5 10 8”这个10个数进行排序。首先在这个序列中随便找一个数作为基准数（不要被这个名词吓到了，就是一个用来参照的数，待会你就知道它用来做啥的了）。为了方便，就让第一个数6作为基准数吧。接下来，需要将这个序列中所有比基准数大的数放在6的右边，比基准数小的数放在6的左边，类似下面这种排列。

3 1 2 5 4 6 9 7 10 8

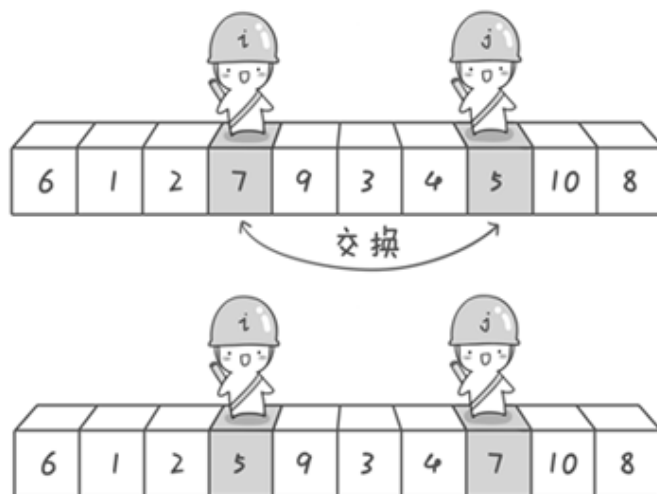
在初始状态下，数字6在序列的第1位。我们的目标是将6挪到序列中间的某个位置，假设这个位置是k。现在就需要寻找这个k，并且以第k位为分界点，左边的数都小于等于6，右边的数都大于等于6。想一想，你有办法可以做到这点吗？

给你一个提示吧。请回忆一下冒泡排序，是如何通过“交换”，一步步让每个数归位的。此时你也可以通过“交换”的方法来达到目的。具体是如何一步步交换呢？怎样交换才既方便又节省时间呢？先别急着往下看，拿出笔来，在纸上画画看。我高中时第一次学习冒泡排序算法的时候，就觉得冒泡排序很浪费时间，每次都只能对相邻的两个数进行比较，这显然太不合理了。于是我就想了一个办法，后来才知道原来这就是“快速排序”，请允许我小小的自恋一下(^o^)。

方法其实很简单：分别从初始序列“6 1 2 7 9 3 4 5 10 8”两端开始“探测”。先从右往左找一个小于6的数，再从左往右找一个大于6的数，然后交换他们。这里可以用两个变量i和j，分别指向序列最左边和最右边。我们为这两个变量起个好听的名字“哨兵i”和“哨兵j”。刚开始的时候让哨兵i指向序列的最左边（即i=1），指向数字6。让哨兵j指向序列的最右边（即j=10），指向数字8。

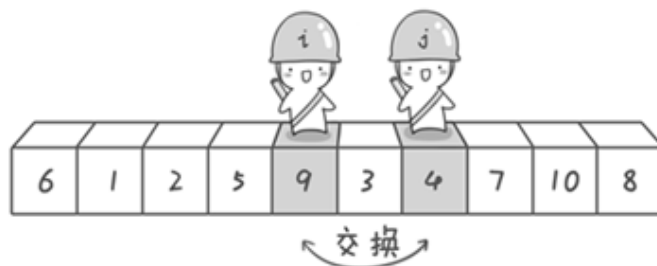


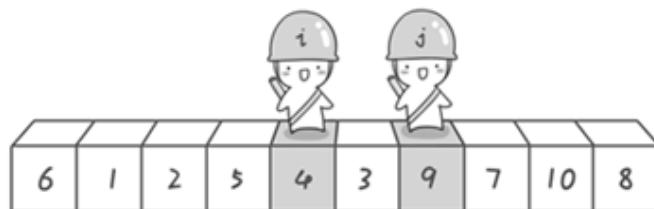
首先哨兵j开始出动。因为此处设置的基准数是最左边的数，所以需要让哨兵j先出动，这一点非常重要（请自己想一想为什么）。哨兵j一步一步地向左挪动（即j--），直到找到一个小于6的数停下来。接下来哨兵i再一步一步向右挪动（即i++），直到找到一个数大于6的数停下来。最后哨兵j停在了数字5面前，哨兵i停在了数字7面前。



现在交换哨兵i和哨兵j所指向的元素的值。交换之后的序列如下。

6 1 2 5 9 3 4 7 10 8



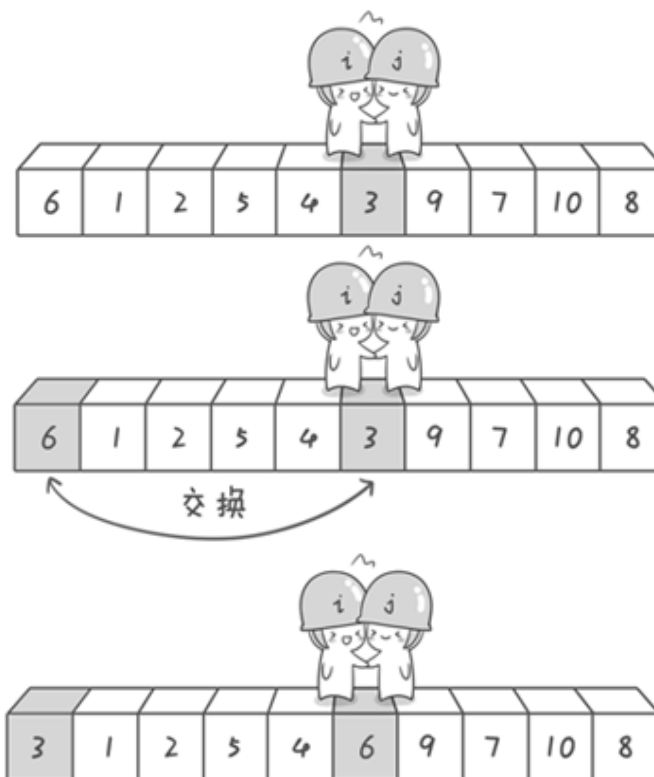


到此，第一次交换结束。接下来开始哨兵j继续向左挪动（再友情提醒，每次必须是哨兵j先出发）。他发现了4（比基准数6要小，满足要求）之后停了下来。哨兵i也继续向右挪动的，他发现了9（比基准数6要大，满足要求）之后停了下来。此时再次进行交换，交换之后的序列如下。

6 1 2 5 4 3 9 7 10 8

第二次交换结束，“探测”继续。哨兵j继续向左挪动，他发现了3（比基准数6要小，满足要求）之后又停了下来。哨兵i继续向右移动，糟啦！此时哨兵i和哨兵j相遇了，哨兵i和哨兵j都走到3面前。说明此时“探测”结束。我们将基准数6和3进行交换。交换之后的序列如下。

3 1 2 5 4 6 9 7 10 8



到此第一轮“探测”真正结束。此时以基准数6为分界点，6左边的数都小于等于6，6右边的数都大于等于6。回顾一下刚才的过程，其实哨兵j的使命就是要找小于基准数的数，而哨兵i的使命就是要找大于基准数的数，直到i和j碰头为止。

OK，解释完毕。现在基准数6已经归位，它正好处在序列的第6位。此时我们已经将原来的序列，以6为分界点拆分成了两个序列，左边的序列是“3 1 2 5 4”，右边的序列是“9 7 10 8”。接下来还需要分别处理这两个序列。因为6左边和右边的序列目前都还是很混乱的。不过不要紧，我们已经掌握了方法，接下来只要模拟刚才的方法分别处理6左边和右边的序列即可。现在先来处理6左边的序列现吧。

左边的序列是“3 1 2 5 4”。请将这个序列以3为基准数进行调整，使得3左边的数都小于等于3，3右边的数都大于等于3。好了开始动笔吧。

如果你模拟的没有错，调整完毕之后的序列的顺序应该是。

2 1 3 5 4

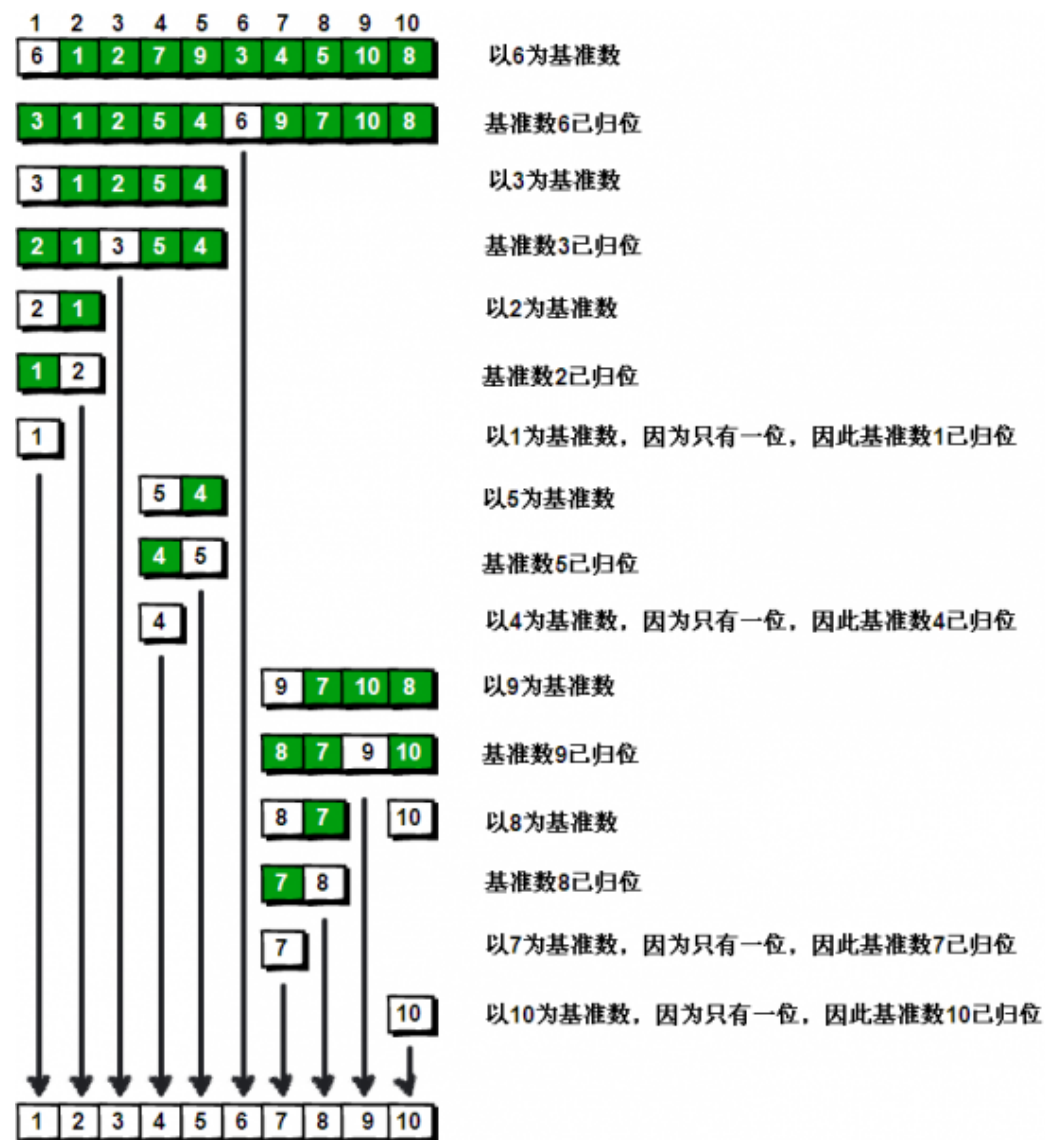
OK，现在3已经归位。接下来需要处理3左边的序列“2 1”和右边的序列“5 4”。对序列“2 1”以2为基准数进行调整，处理完毕之后的序列为“1 2”，到此2已经归位。序列“1”只有一个数，也不需要任何处理。至此我们对序列“2 1”已全部处理完毕，得到序列是“1 2”。序列“5 4”的处理也仿照此方法，最后得到的序列如下。

1 2 3 4 5 6 9 7 10 8

对于序列“9 7 10 8”也模拟刚才的过程，直到不可拆分出新的子序列为止。最终将会得到这样的序列，如下。

1 2 3 4 5 6 7 8 9 10

到此，排序完全结束。细心的同学可能已经发现，快速排序的每一轮处理其实就是将这一轮的基准数归位，直到所有的数都归位为止，排序就结束了。下面上个霸气的图来描述下整个算法的处理过程。



快速排序之所比较快，因为相比冒泡排序，每次交换是跳跃式的。每次排序的时候设置一个基准点，将小于等于基准点的数全部放到基准点的左边，将大于等于基准点的数全部放到基准点的右边。这样在每次交换的时候就不会像冒泡排序一样每次只能在相邻的数之间进行交换，交换的距离就大的多了。因此总的比较和交换次数就少了，速度自然就提高了。当然在最坏的情况下，仍可能是相邻的两个数进行了交换。因此快速排序的最差时间复杂度和冒泡排序是一样的都是 $O(N^2)$ ，它的平均时间复杂度为 $O(N\log N)$ 。其实快速排序是基于一种叫做“二分”的思想。我们后面还会遇到“二分”思想，到时候再聊。先上代码，如下。

```

#include <stdio.h>
int a[101],n;//定义全局变量，这两个变量需要在子函数中使用
void quicksort(int left,int right)
{
    int i,j,t,temp;
    if(left>right)
        return;

    temp=a[left]; //temp中存的就是基准数
    i=left;
    j=right;
    while(i!=j)
    {
        //顺序很重要，要先从右边开始找
        while(a[j]>=temp && i<j)
            j--;
        //再找左边的
        while(a[i]<=temp && i<j)
            i++;
        //交换两个数在数组中的位置
        if(i<j)
        {
            t=a[i];
            a[i]=a[j];
            a[j]=t;
        }
    }
    //最终将基准数归位
    a[left]=a[i];
    a[i]=temp;

    quicksort(left,i-1);//继续处理左边的，这里是一个递归的过程
    quicksort(i+1,right);//继续处理右边的，这里是一个递归的过程
}
int main()
{
    int i,j,t;
    //读入数据
    scanf("%d",&n);
    for(i=1;i<=n;i++)
        scanf("%d",&a[i]);
    quicksort(1,n); //快速排序调用

    //输出排序后的结果
    for(i=1;i<=n;i++)

```



```
    printf("%d ",a[i]);  
    getchar();getchar();  
    return 0;  
}
```



可以输入以下数据进行验证

```
10  
6  1  2  7  9  3  4  5  10  8
```

运行结果是

```
1 2 3 4 5 6 7 8 9 10
```

下面是程序执行过程中数组a的变化过程，带下划线的数表示的已归位的基准数。



```
6 1 2 7 9 3 4 5 10 8  
3 1 2 5 4 6 9 7 10 8  
2 1 3 5 4 6 9 7 10 8  
1 2 3 5 4 6 9 7 10 8  
1 2 3 5 4 6 9 7 10 8  
1 2 3 4 5 6 9 7 10 8  
1 2 3 4 5 6 9 7 10 8  
1 2 3 4 5 6 8 7 9 10  
1 2 3 4 5 6 7 8 9 10  
1 2 3 4 5 6 7 8 9 10  
1 2 3 4 5 6 7 8 9 10  
1 2 3 4 5 6 7 8 9 10
```



快速排序由 C. A. R. Hoare（东尼·霍尔，Charles Antony Richard Hoare）在1960年提出，之后又有许多人做了进一步的优化。如果你对快速排序感兴趣可以去看看东尼·霍尔1962年在Computer Journal发表的论文“Quicksort”以及《算法导论》的第七章。快速排序算法仅仅是东尼·霍尔在计算机领域才能的第一次显露，后来他受到了老板的赏识和重用，公司希望他为新机器设计一个新的高级语言。你要知道当时还没有PASCAL或者C语言这些高级的东东。后来东尼·霍尔参加了由Edsger Wybe Dijkstra（1972年图灵奖得主，这个大神我们后面还会遇到的到时候再细聊）举办的“ALGOL 60”培训班，他觉得自己与其没有把握去设计一个新的语言，还不如对现有的“ALGOL 60”进行改进，使之能在公司的新机器上使用。于是他便设计了“ALGOL 60”的一个子集版本。这个版本在执行效率和可靠性上都在当时“ALGOL 60”的各种版本中首屈一指，因此东尼·霍尔受到了国际学术界的重视。后来他在“ALGOL X”的设计中还发明了大家熟知的“case”语句，后来也被各种高级语言广泛采用，比如PASCAL、C、Java语言等等。当然，东尼·霍尔在

计算机领域的贡献还有很多很多，他在1980年获得了图灵奖。

码字不容易啊，转载请标明出处^_^

【一周一算法】算法3：最常用的排序——快速排序

<http://bbs.ahalei.com/thread-4419-1-1.html>(出处: 啊哈磊_编程从这里起步)

分类: [啊哈！算法](#)

标签: [排序算法](#), [快速排序算法](#), [快排](#)

绿色通道：

好文要顶

已关注

收藏该文

与我联系





[啊哈磊](#)

[关注 - 0](#)

[粉丝 - 181](#)

我在关注他 [取消关注](#)

18
[推荐](#)

0
[反对](#)

(请您对文章做出评价)

« 上一篇：[【坐在马桶上看算法】算法2：邻居好说话：冒泡排序](#)

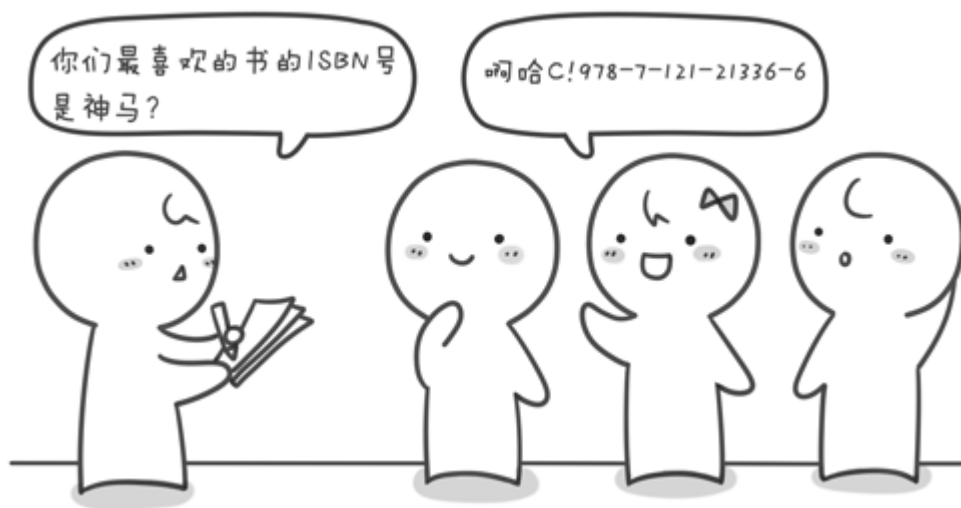
» 下一篇：[【坐在马桶上看算法】小哼买书](#)

posted @ 2014-02-26 10:04 [啊哈磊](#) 阅读(1565) 评论(19) [编辑](#) [收藏](#)

小哼买书

之前讲了三种常用的经典排序。排序算法还有很多，例如选择排序、计数排序、基数排序、插入排序、归并排序和堆排序等等。堆排序是基于二叉树的排序，以后再说吧。先分享一个超酷的排序算法的视频。

再来看一个具体的例子《小哼买书》来看看三个排序在应用上的区别和局限性。小哼的学校要建立一个图书角，老师派小哼去找一些同学做调查，看看同学们都喜欢读哪些书。小哼让每个同学写出一个自己最想读的书的ISBN号（你知道吗？每本书都有唯一的ISBN号，不信话你去找本书翻到背面看看）。当然有一些好书会有很多同学都喜欢，这样就会收集到很多重复的ISBN号。小哼需要去掉其中重复的ISBN号，即每个ISBN号只保留一个，也就是说同样的书只买一本（学校真是够抠门的）。然后再把这些ISBN号从小到大排序，小哼将按照排序好的ISBN号去书店去买书。请你协助小哼完成“去重”与“排序”的工作。



输入有2行，第1行为一个正整数，表示有 n 个同学参与调查 ($n \leq 100$)。第2行有 n 个用空格隔开的正整数，为每本图书的ISBN号（假设图书的ISBN号在1~1000之间）。

输出也是2行，第1行为一个正整数 k ，表示需要买多少本书。第2行为 k 个用空格隔开的正整数，为从小到大已排好序的需要购买的图书ISBN号。

例如输入

```
10
20 40 32 67 40 20 89 300 400 15
```

则输出

```
8
15 20 32 40 67 89 300 400
```

最后，程序运行的时间限制为：1秒。

解决这个问题的方法大致有两种，第一种方法：先将这 n 个图书的ISBN号去重，再进行从小到大排序并输出。第二种方法：先从小到大排序，输出的时候再去重。这两种方法都可以。

先来看第一种方法。通过第一节的学习我们发现桶排序稍加改动正好可以起到去重的效果，因此我们可以使用桶排序的方法来解决此问题。

```
#include <stdio.h>
int main()
{
    int a[1001], n, i, t;
    for(i=1; i<=1000; i++)
        a[i]=0; //初始化

    scanf("%d", &n); //读入n
    for(i=1; i<=n; i++) //循环读入n个图书的ISBN号
    {
        scanf("%d", &t); //把每一个ISBN号读到变量t中
        a[t]=1; //标记出现过的ISBN号
    }
}
```

```

}

for(i=1;i<=1000;i++) //依次判断1~1000这个1000个桶
{
    if(a[i]==1)//如果这个ISBN号出现过则打印出来
        printf("%d ",i);
}
getchar();getchar();
return 0;
}

```



这种方法的时间复杂度是就是桶排序的时间复杂度为 $O(N+M)$ 。

第二种方法我们需要先排序再去重。排序我们可以用冒泡排序或者快速排序。

20 40 32 67 40 20 89 300 400 15

将这10个数从小到大排序之后为 15 20 20 32 40 40 67 89 300 400。

接下来，要在输出的时候去掉重复的。因为我们已经排好序，因此相同的数都会紧挨在一起。只要在输出的时候，预先判断一下当前这个数 $a[i]$ 与前面一个数 $a[i-1]$ 是否相同。如果相同则表示这个数之前已经输出过了，不同再次输出。不同则表示这个数是第一次出现需要，则需要输出这个数。



```

#include <stdio.h>
int main()
{
    int a[101],n,i,j,t;

    scanf("%d",&n);    //读入n
    for(i=1;i<=n;i++) //循环读入n个图书ISBN号
    {
        scanf("%d",&a[i]);
    }

    //开始冒泡排序
    for(i=1;i<=n-1;i++)
    {
        for(j=1;j<=n-i;j++)
        {
            if(a[j]>a[j+1])
            { t=a[j]; a[j]=a[j+1]; a[j+1]=t; }
        }
    }
    printf("%d ",a[1]); //输出第1个数
    for(i=2;i<=n;i++) //从2循环到n
    {
        if( a[i] != a[i-1] ) //如果当前这个数是第一次出现则输出
            printf("%d ",a[i]);
    }
    getchar();getchar();
    return 0;
}

```



这种方法的时间复杂度由两部分组成，一部分是冒泡排序时间复杂度是 $O(N^2)$ ，另一部分是读入和输出都是 $O(N)$ ，因此整个算法的时间复杂度是 $O(2*N+N^2)$ 。相对于 N^2 来说， $2*N$ 可以忽略（我们通常忽略低阶），最终该方法的时间复杂度是 $O(N^2)$ 。

接下来我们还需要看下数据范围。每个图书ISBN号都是1~1000之间的整数，并且参加调查的同学人数不超过100，即 $n \leq 100$ 。之前已经说过，在粗略计算时间复杂度的时候，我们通常认为计算机每秒钟大约运行10亿次（当然实际情况要更快）。因此以上两种方法都可以在1秒钟内计算出解。如果题目图书ISBN号范围不是在1~1000之间，而是-2147483648~2147483647之间的话，那么第一种方法就不可行了，因为你无法申请出这么大数组来标记每一个ISBN号是否出现过。另外如果n的范围不是小于等于100而是小于等于10万，那么第二种方法的排序部分也不能使用冒泡排序。因为题目要求的时间限制是1秒，使用冒泡排序对10万个数的排序，计算机需要运行100亿次，需要10秒钟，需要替换为快速排序，快速排序只需要 $100000 \times \log 2100000 \approx 100000 \times 17 \approx 170$ 万次，这还不到0.0017秒。是不是很神奇，同样的问题使用不同的算法竟然有如此之大的时间差距，这就是算法的魅力！

我们来回顾一下本章三种排序算法的时间复杂度。桶排序是最快的，它的时间复杂度是 $O(N+M)$ ；冒泡排序是 $O(N^2)$ ；快速排序是 $O(N \log N)$ 。

码字不容易啊，转载请标明出处^_^.

<http://www.cnblogs.com/ahalei/p/3577792.html>

分类: [啊哈！算法](#)

标签: [排序算法比较](#), [noip](#)

绿色通道：

[好文要顶](#)

[已关注](#)

[收藏该文](#)

[与我联系](#)



[啊哈磊](#)

[关注 - 0](#)

[粉丝 - 181](#)

我在关注他 [取消关注](#)

8

[推荐](#)

0

[反对](#)

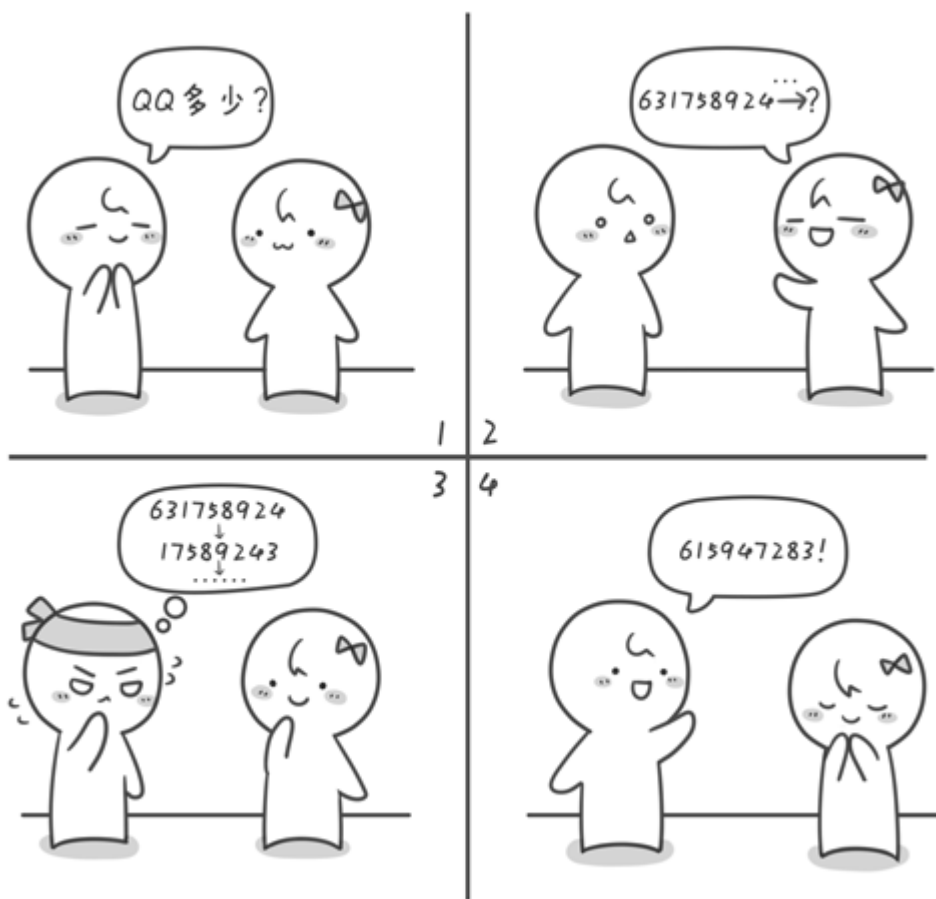
(请您对文章做出评价)

« 上一篇: [【坐在马桶上看算法】算法3：最常用的排序——快速排序](#)

» 下一篇: [【坐在马桶上看算法】算法4：队列——解密QQ号](#)

算法4：队列——解密QQ号

新学期开始了，小哈是小哼的新同桌（小哈是个小美女哦~），小哼向小哈询问QQ号，小哈当然不会直接告诉小哼啦，原因嘛你懂的。所以小哈给了小哼一串加密过的数字，同时小哈也告诉了小哼解密规则。规则是这样的：首先将第1个数删除，紧接着将第2个数放到这串数的末尾，再将第3个数删除并将第4个数再放到这串数的末尾，再将第5个数删除.....直到剩下最后一个数，将最后一个数也删除。按照刚才删除的顺序，把这些删除的数连在一起就是小哈的QQ啦。现在你来帮帮小哼吧。小哈给小哼加密过的一串数是“6 3 1 7 5 8 9 2 4”。



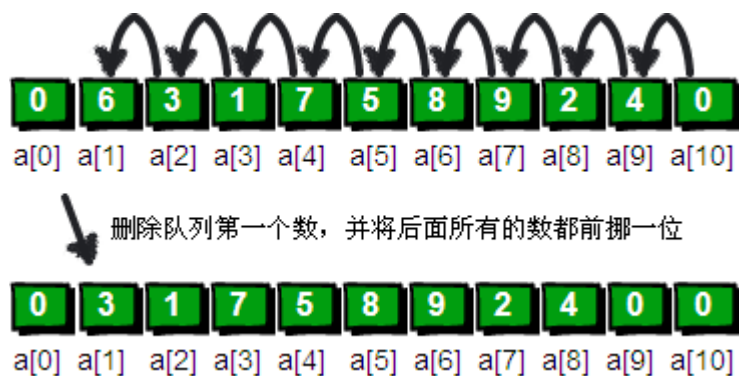
OK，现在轮到你动手的时候了。快去找9张便签或小纸片，将“6 3 1 7 5 8 9 2 4”这9个数分别写在9张便签上，模拟一下解密过程。如果你没有理解错解密规则的话，解密后小哈的QQ号应该是“6 1 5 9 4 7 2 8 3”。

其实解密的过程就像是把这些数“排队”。每次从最前面拿两个，第1个扔掉，第2个放到尾部。具体过程是这样的：刚开始这串数是“6 3 1 7 5 8 9 2 4”，首先删除6并将3放到这串数的末尾，这串数更新为“1 7 5 8 9 2 4 3”。接下来删除1并将7放到末尾，即更新为“5 8 9 2 4 3 7”。再删除5并将8放到末尾即“9 2 4 3 7 8”，删除9并将2放到末尾即“4 3 7 8 2”，删除4并将3放到末尾即“7 8 2 3”，删除7并将8放到末尾即“2 3 8”，删除2并将3放到末尾即“8 3”，删除8并将3放到末尾即“3”，最后删除3。因此被删除的顺序是“6 1 5 9 4 7 2 8 3”，这就是小哈的QQ号码了，你可以加她试试看^_^。

既然现在已经搞清楚了解密法则，不妨自己尝试一下去编程，我相信你一定可以写出来的。

首先需要 一个数组来存储这一串数即intq[101]。并初始化这个数组即intq[101]={0,6,3,1,7,5,8,9,2,4};（此处初始化是我多写了一个0，用来填充q[0]，因为我比较喜欢从q[1]开始用，对数组初始化不是很理解的同学可以去看下我的上一本书《啊哈C！思考快你一步》）接下来就是模拟解密的过程了。

解密的第一步是将第一个数删除，你可以想一下如何在数组中删除一个数呢？最简单的方法是将所有后面的数都往前面挪动一位，将前面的数覆盖。就好比我们在排队买票，最前面的人买好离开了，后面所有的人就需要全部向前面走一步，补上之前的空位，但是这样的做法很耗费时间。

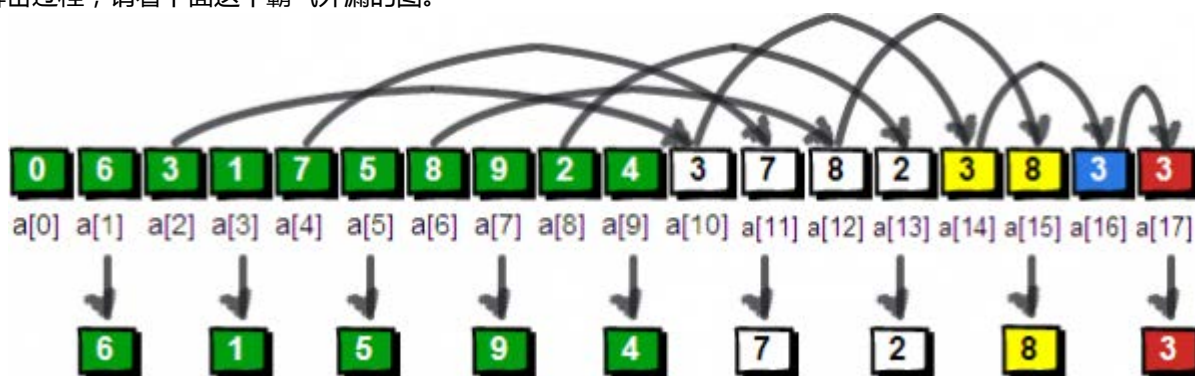


在这里，我将引入两个整型变量head和tail。head用来记录队列的队首（即第一位），tail用来记录队列的队尾（即最后一位）的下一个位置。你可能会问为什么tail不直接记录队尾，却要记录队尾的下一个位置呢？这是因为当队列当中只剩下一个元素时，队首和队尾重合会带来一些麻烦。我们这里规定队首和队尾重合时，队列为空。

现在有9个数，9个数全部放入队列之后head=1;tail=10;此时head和tail之间的数就是目前队列中“有效”的数。如果要删除一个数的话，就将head++就OK了，这样仍然可以保持head和tail之间的数为目前队列中“有效”的数。这样做虽然浪费了一个空间，却节省了大量的时间，这是非常划算的。新增加一个数也很简单，把需要增加的数放到队尾即q[tail]之后再tail++就欧克啦。



我们来小结一下，在队首删除一个数的操作是head++；
在队尾增加一个数（假设这个数是x）的操作是q[tail]=x;tail++；
整个解密过程，请看下面这个霸气外漏的图。



最后的输出就是6 1 5 9 4 7 2 8 3，代码实现如下。

```
#include <stdio.h>
int main()
{
    int q[102]={0,6,3,1,7,5,8,9,2,4},head,tail;
    int i;
    //初始化队列
    head=1;
    tail=10; //队列中已经有9个元素了，tail指向的队尾的后一个位置
    while(head<tail) //当队列不为空的时候执行循环
    {
        //打印队首并将队首出队
        printf("%d ",q[head]);
    }
}
```



```

    head++;
    //先将新队首的数添加到队尾
    q[tail]=q[head];
    tail++;
    //再将队首出队
    head++;
}

getchar();getchar();
return 0;
}

```

怎么样上面的代码运行成功没有？现在我们来总结一下队列的概念。队列是一种特殊的线性结构，它只允许在队列的首部（head）进行删除操作称之为“出队”，而在队列的尾部（tail）进行插入操作称之为“入队”。当队列中没有元素时（即head==tail），称为空队列。在我们的日常生活中有很多情况都符合队列的特性。比如我们之前提到过的买票，每个排队买票的窗口就是一个队列。在这个队列当中，新来的人总是站在队列的最后面，来的越早的人越靠前也就越早能买到票，就是先来的人先服务，我们称为“先进先出”（First InFirst Out，FIFO）原则。

队列将是我们今后学习广度优先搜索以及队列优化的Bellman-Ford最短路算法的核心数据结构。所以现在将队列的三个基本元素（一个数组，两个变量）封装为一个结构体类型，如下。

```

struct queue
{
    int data[100]; //队列的主体，用来存储内容
    int head; //队首
    int tail; //队尾
};

```

上面我们定义了一个结构体类型，我们通常将其放在main函数的外面，请注意结构体的定义末尾有个;号。struct是结构体的关键字，queue是我们为这个结构体起的名字。这个结构体有三个成员分别是：整型数组data、整型head和整型tail。这样我们就可以把这三个部分放在一起作为一个整体来对待。你可以这么理解：我们定义了一个新的数据类型，这个新类型非常强大，用这个新类型定义出的每一个变量可以同时存储一个整型数组和两个整数。



有了新的结构体类型，如何定义结构体变量呢？很简单，这与我们之前定义变量的方式是一样，如下。

```

struct queue q;

```

请注意struct queue需要整体使用，不能直接写queue q;这样我们就定义了一个结构体变量q。这个结构体变量q就可以满足队列的所有操作了。那又该如何访问结构体变量的内部成员呢？可以使用.号，它叫做成员运算符或者点号运算符，如下：

```

q.head=1;
q.tail=1;
scanf("%d",&q.data[q.tail]);

```

好了，下面这段代码就是使用结构体来实现的队列操作。



```
#include <stdio.h>

struct queue
{
    int data[100]; //队列的主体，用来存储内容
    int head; //队首
    int tail; //队尾
};

int main()
{
    struct queue q;
    int i;
    //初始化队列
    q.head=1;
    q.tail=1;
    for(i=1;i<=9;i++)
    {
        //依次向队列插入9个数
        scanf("%d",&q.data[q.tail]);
        q.tail++;
    }

    while(q.head<q.tail) //当队列不为空的时候执行循环
    {
        //打印队首并将队首出队
        printf("%d ",q.data[q.head]);
        q.head++;

        //先将新队首的数添加到队尾
        q.data[q.tail]=q.data[q.head];
        q.tail++;
        //再将队首出队
        q.head++;
    }

    getchar();getchar();
    return 0;
}
```



上面的这种写法看起来虽然冗余了一些，但是可以加强你对队列这个算法的理解。C++的STL库已经有队列的实现，有兴趣的同学可以参看相关材料。队列的起源已经无法追溯。在还没有数字计算机之前，数学应用中就已经有对队列的记载了。我们生活中队列的例子也比皆是，比如排队买票，又或者吃饭时候用来排队等候的叫号机，又或者拨打银行客服选择人工服务时，每次都会被提示“客服人员正忙，请耐心等待”，因为客服人员太少了，拨打电话的客户需要按照打进的时间顺序进行等候等等。这里表扬一下肯德基的宅急送，没有做广告的嫌疑啊，每次一打就通，基本不需要等待。但是我每次打银行的客服（具体是哪家银行就不点名了）基本都要等待很长时间，总是告诉我“正在转接，请稍后”，嘟嘟嘟三声后就变成“客服正忙，请耐心等待！”然后就给我放很难听的曲子。看来钱在谁那里谁就是老大啊.....

码字不容易啊，转载麻烦注明出处

【一周一算法】解密QQ号——队列

<http://bbs.ahalei.com/thread-4489-1-1.html>

(出处: 啊哈磊_编程从这里起步)

分类: [啊哈！算法](#)

标签: [超简单队列教程](#), [队列实现](#), [队列C语言代码](#), [队列C语言实现](#)

绿色通道：

好文要顶

已关注

收藏该文

与我联系





[啊哈磊](#)
[关注 - 0](#)
[粉丝 - 181](#)

我在关注他 [取消关注](#)

6

0

 推荐

 反对

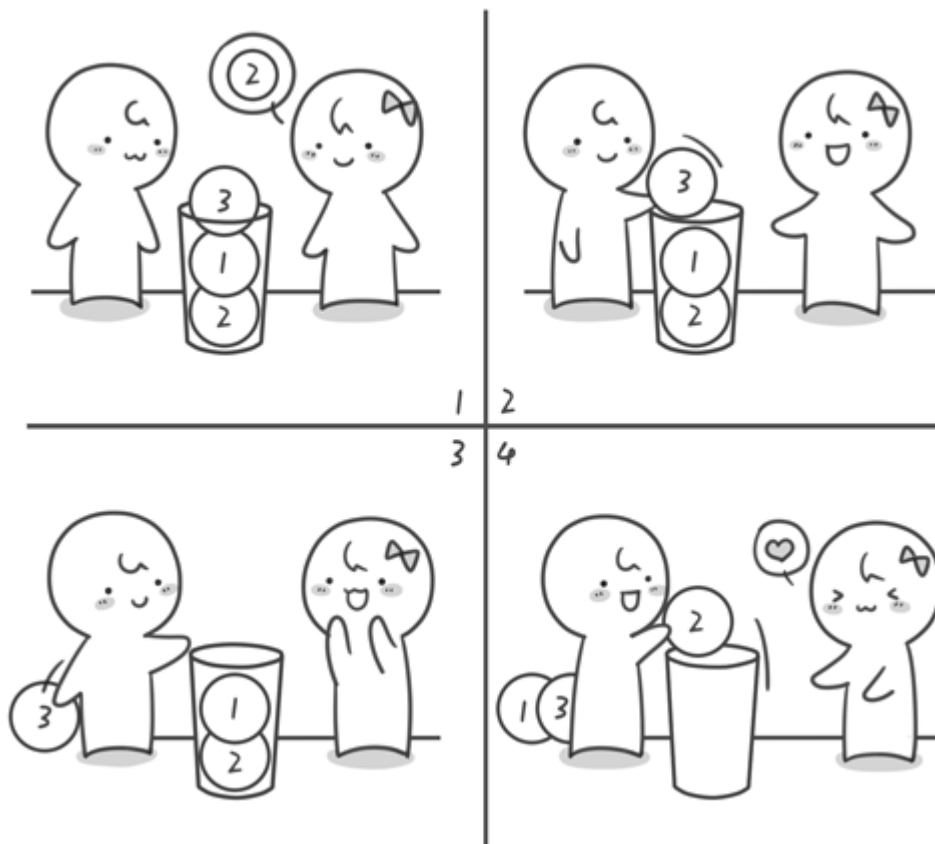
(请您对文章做出评价)

« 上一篇：[【坐在马桶上看算法】小哼买书](#)

» 下一篇：[【坐在马桶上看算法】算法5：解密回文——栈](#)

算法5：解密回文——栈

上一节中我们学习了队列，它是一种先进先出的数据结构。还有一种是后进先出的数据结构它叫做栈。栈限定只能在一端进行插入和删除操作。比如说有一个小桶，小桶的直径只能放一个小球，我们现在向小桶内依次放入2号、1号、3号小球。假如你现在需要拿出2号小球，那就必须先先将3号小球拿出，再拿出1号小球，最后才能将2号小球拿出来。在刚才取小球的过程中，我们最先放进去的小球最后才能拿出来，而最后放进去的小球却可以最先拿出来。这就是后进先出，也可以称为先进后出。



我们生活中还有很多这样的例子，比如我们在吃桶装薯片的时候，要想吃掉最后一块，就必须把前面的全部吃完（貌似现在的桶装薯片为了减少分量，在桶里面增加了一个透明的抽屉）；再比如我们浏览网页时候需要退回到之前的某个网页，我们需要一步步的点击后退键。还有手-枪的弹夹，在装子弹的时候，最后装的一发子弹，是被第一个打出去的。栈的实现也很简单，只需要一个一维数组和一个指向栈顶的变量top就可以了。我们通过变量top来对栈进行插入和删除操作。

这种特殊的数据结构栈究竟有哪些作用呢？我们来看一个例子。“xyzyx”是一个回文字符串，所谓回文字符串就是指正读反读均相同的字符序列，如“席主席”、“记书记”、“aha”和“ahaha”均是回文，但“ahah”不是回文。通过栈这个数据结构我们将很容易判断一个字符串是否为回文。

首先我们需要读取这行字符串，并求出这个字符串的长度。

```
char a[101]; //101是一个估算值，只需比待读入的字符串长度大即可
int len;
gets(a);
len=strlen(a);
```

如果一个字符串是回文的话，那么它必须是中间对称，我们需要求这个字符串的 中点，即：

```
mid=len/2-1;
```

接下来就轮到栈出场了。

我们先将mid之前的部分的字符全部入栈。因为这里的栈是用来存储字符的，所以这里用来实现栈的数组类型是字符数组即char s[101]; 初始化栈很简单，top=0;就可以了。入栈的操作是top++;s[top]=x;（假设需要入栈的字符存储暂存在字符变量x中）其实可以简写为s[++top]=x;

现在我们就来将mid之前的字符依次全部入栈。这里循环要0开始，因为刚才读取字符串使用了gets()函数，读取的第一个字

符存储在s[0]中，随后一个字符存储在s[len-1]中。

```
for(i=0;i<=mid;i++)
{
    s[++top]=a[i];
}
```

接下来进入判断回文的关键步骤。将当前栈中的字符依次出栈，看看是否能与mid之后的字符一一匹配，如果都能匹配则说明这个字符串是回文字符串，否则这个字符串就不是回文字符串。



```
for(i=mid+1;i<=len-1;i++) //其实这里并不一定是mid+1，需要讨论字符串长度的奇偶性
{
    if (a[i]!=s[top])
    {
        break;
    }
    top--;
}
if(top==0)
    printf("YES");
else
    printf("NO");
```



最后如果top的值为0，就说明栈内所有的字符都被一一匹配了，那么这个字符串就是回文字符串。完整的代码如下。



```
#include <stdio.h>
#include <string.h>
int main()
{
    char a[101],s[101];
    int i,len,mid,next,top;

    gets(a); //读入一行字符串
    len=strlen(a); //求字符串的长度
    mid=len/2-1; //求字符串的中点

    top=0; //栈的初始化
    //将mid前的字符依次入栈
    for(i=0;i<=mid;i++)
        s[++top]=a[i];

    //判断字符串的长度的奇偶性，并找出需要进行字符匹配的起始下标
    if(len%2==0)
        next=mid+1;
    else
        next=mid+2;

    //开始匹配
    for(i=next;i<=len-1;i++)
    {
        if(a[i]!=s[top])
```

```
        break;
    top--;
}

//如果top的值为0, 则说明栈内的所有的字符都被一一匹配了
if(top==0)
    printf("YES");
else
    printf("NO");

getchar();getchar();
return 0;
}
```



可以输入以下数据进行验证

ahaha

运行结果是

YES

栈还可以用来进行验证括号的匹配。比如输入一行只包含“(){}”的字符串，形如“([{}])”或者“{([]{})}”请判断是否可以正确匹配。显然上面两个例子都是可以正确匹配的。“([)]”是不能匹配的。有兴趣的同学可以自己动手来试一试。

堆栈最早由Alan M. Turing (艾伦·图灵) 于1946年提出，当时为了解决子程序的调用和返回。艾伦·图灵这个大帅哥可是个大牛人，图灵奖就是以他的名字命名的。如果你对他感兴趣不妨去读一读他的传记《艾伦·图灵传：如谜的解谜者》。

码字不容易啊，转载麻烦注明出处

【一周一算法】算法5：解密回文——栈

<http://bbs.ahalei.com/thread-4515-1-1.html>

(出处: 啊哈磊_编程从这里起步)

分类: [啊哈！算法](#)

标签: [栈的C语言代码](#), [栈的实现](#), [数据结构栈](#), [超简单栈C语言代码](#)

绿色通道：

[好文要顶](#)

[已关注](#)

[收藏该文](#)

[与我联系](#)



啊哈磊

[关注 - 0](#)

[粉丝 - 181](#)

我在关注他 [取消关注](#)

3

[推荐](#)

0

[反对](#)

(请您对文章做出评价)

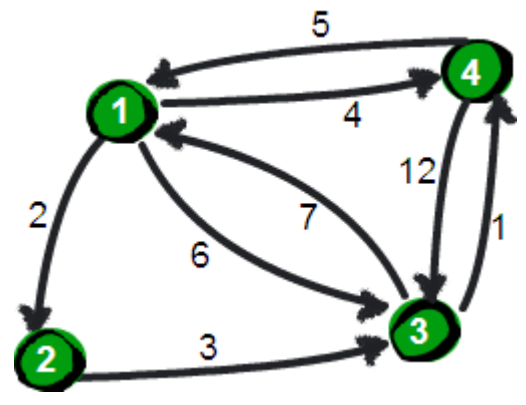
« 上一篇: [【坐在马桶上看算法】算法4：队列——解密QQ号](#)

» 下一篇: [【坐在马桶上看算法】算法6：只有五行的Floyd最短路算法](#)

算法6：只有五行的Floyd最短路算法



暑假，小哼准备去一些城市旅游。有些城市之间有公路，有些城市之间则没有，如下图。为了节省经费以及方便计划旅程，小哼希望在出发之前知道任意两个城市之前的最短路程。



上图中有4个城市8条公路，公路上的数字表示这条公路的长短。请注意这些公路是单向的。我们现在需要求任意两个城市之间的最短路程，也就是求任意两个点之间的最短路径。这个问题这也被称为“多源最短路径”问题。

现在需要一个数据结构来存储图的信息，我们仍然可以用一个4*4的矩阵（二维数组e）来存储。比如1号城市到2号城市的路程为2，则设e[1][2]的值为2。2号城市无法到达4号城市，则设置e[2][4]的值为 ∞ 。另外此处约定一个城市自己是到自己的也是0，例如e[1][1]为0，具体如下。

	1	2	3	4
1	0	2	6	4
2	∞	0	3	∞
3	7	∞	0	1
4	5	∞	12	0

现在回到问题：如何求任意两点之间最短路径呢？通过之前的学习我们知道通过深度或广度优先搜索

可以求出两点之间的最短路径。所以进行n2遍深度或广度优先搜索，即对每两个点都进行一次深度或广度优先搜索，便可以求得任意两点之间的最短路径。可是还有没有别的方法呢？

我们来想一想，根据我们以往的经验，如果要让任意两点（例如从顶点a点到顶点b）之间的路程变短，只能引入第三个点（顶点k），并通过这个顶点k中转即a->k->b，才可能缩短原来从顶点a点到顶点b的路程。那么这个中转的顶点k是1~n中的哪个点呢？甚至有时候不只通过一个点，而是经过两个点或者更多点中转会更短，即a->k1->k2b->或者a->k1->k2...->k-i...->b。比如上图中从4号城市到3号城市（4->3）的路程e[4][3]原本是12。如果只通过1号城市中转（4->1->3），路程将缩短为11（e[4][1]+e[1][3]=5+6=11）。其实1号城市到3号城市也可以通过2号城市中转，使得1号到3号城市的路程缩短为5（e[1][2]+e[2][3]=2+3=5）。所以如果同时经过1号和2号两个城市中转的话，从4号城市到3号城市的路程会进一步缩短为10。通过这个的例子，我们发现每个顶点都有可能使得另外两个顶点之间的路程变短。好，下面我们将这个问题一般化。

当任意两点之间不允许经过第三个点时，这些城市之间最短路程就是初始路程，如下。

	1	2	3	4
1	0	2	6	4
2	∞	0	3	∞
3	7	∞	0	1
4	5	∞	12	0

假如现在只允许经过1号顶点，求任意两点之间的最短路程，应该如何求呢？只需判断e[i][1]+e[1][j]是否比e[i][j]要小即可。e[i][j]表示的是从i号顶点到j号顶点之间的路程。e[i][1]+e[1][j]表示的是从i号顶点先到1号顶点，再从1号顶点到j号顶点的路程之和。其中i是1~n循环，j也是1~n循环，代码实现如下。

```
for(i=1;i<=n;i++)
{
    for(j=1;j<=n;j++)
    {
        if ( e[i][j] > e[i][1]+e[1][j] )
            e[i][j] = e[i][1]+e[1][j];
    }
}
```

在只允许经过1号顶点的情况下，任意两点之间的最短路程更新为：

	1	2	3	4
1	0	2	6	4
2	∞	0	3	∞
3	7	9	0	1
4	5	7	11	0

通过上图我们发现：在只通过1号顶点中转的情况下，3号顶点到2号顶点（ $e[3][2]$ ）、4号顶点到2号顶点（ $e[4][2]$ ）以及4号顶点到3号顶点（ $e[4][3]$ ）的路程都变短了。

接下来继续求在只允许经过1和2号两个顶点的情况下任意两点之间的最短路程。如何做呢？我们需要在只允许经过1号顶点时任意两点的最短路程的结果下，再判断如果经过2号顶点是否可以使得i号顶点到j号顶点之间的路程变得更短。即判断 $e[i][2]+e[2][j]$ 是否比 $e[i][j]$ 要小，代码实现为如下。

```
//经过1号顶点
for(i=1;i<=n;i++)
    for(j=1;j<=n;j++)
        if (e[i][j] > e[i][1]+e[1][j]) e[i][j]=e[i][1]+e[1][j];

//经过2号顶点
for(i=1;i<=n;i++)
    for(j=1;j<=n;j++)
        if (e[i][j] > e[i][2]+e[2][j]) e[i][j]=e[i][2]+e[2][j];
```

在只允许经过1和2号顶点的情况下，任意两点之间的最短路程更新为：

	1	2	3	4
1	0	2	5	4
2	∞	0	3	∞
3	7	9	0	1
4	5	7	10	0

通过上图得知，在相比只允许通过1号顶点进行中转的情况下，这里允许通过1和2号顶点进行中转，使得 $e[1][3]$ 和 $e[4][3]$ 的路程变得更短了。

同理，继续在只允许经过1、2和3号顶点进行中转的情况下，求任意两点之间的最短路程。任意两点之间的最短路程更新为：

	1	2	3	4
1	0	2	5	4
2	10	0	3	4
3	7	9	0	1
4	5	7	10	0

最后允许通过所有顶点作为中转，任意两点之间最终的最短路程为：

	1	2	3	4
1	0	2	5	4
2	9	0	3	4
3	6	8	0	1
4	5	7	10	0

整个算法过程虽然说起来很麻烦，但是代码实现却非常简单，核心代码只有五行：

```
for(k=1;k<=n;k++)
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
            if(e[i][j]>e[i][k]+e[k][j])
                e[i][j]=e[i][k]+e[k][j];
```

这段代码的基本思想就是：最开始只允许经过1号顶点进行中转，接下来只允许经过1和2号顶点进行中转.....允许经过1~n号所有顶点进行中转，求任意两点之间的最短路程。用一句话概括就是：从i号顶点到j号顶点只经过前k号点的最短路程。其实这是一种“动态规划”的思想，关于这个思想我们将在《啊哈！算法2——伟大思维闪耀时》在做详细的讨论。下面给出这个算法的完整代码：

```
#include <stdio.h>
int main()
{
    int e[10][10],k,i,j,n,m,t1,t2,t3;
    int inf=99999999; //用inf(infinity的缩写)存储一个我们认为的正无穷值
    //读入n和m, n表示顶点个数, m表示边的条数
    scanf("%d %d",&n,&m);

    //初始化
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
            if(i==j) e[i][j]=0;
            else e[i][j]=inf;

    //读入边
    for(i=1;i<=m;i++)
```

```

{
    scanf("%d %d %d",&t1,&t2,&t3);
    e[t1][t2]=t3;
}

//Floyd-Warshall算法核心语句
for(k=1;k<=n;k++)
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
            if(e[i][j]>e[i][k]+e[k][j] )
                e[i][j]=e[i][k]+e[k][j];

//输出最终的结果
for(i=1;i<=n;i++)
{
    for(j=1;j<=n;j++)
    {
        printf("%10d",e[i][j]);
    }
    printf("\n");
}

return 0;
}

```



有一点需要注意的是：如何表示正无穷。我们通常将正无穷定义为99999999，因为这样即使两个正无穷相加，其和仍然不超过int类型的范围（C语言int类型可以存储的最大正整数是2147483647）。在实际应用中最好估计一下最短路径的上限，只需要设置比它大一点既可以。例如有100条边，每条边不超过100的话，只需将正无穷设置为10001即可。如果你认为正无穷和其它值相加得到一个大于正无穷的数是不被允许的话，我们只需在比较的时候加两个判断条件就可以了，请注意下面代码中带有下划线的语句。

```

//Floyd-Warshall算法核心语句
for(k=1;k<=n;k++)
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
            if(e[i][k]<inf && e[k][j]<inf && e[i][j]>e[i][k]+e[k][j])
                e[i][j]=e[i][k]+e[k][j];

```

上面代码的输入数据样式为：



```

4 8
1 2 2
1 3 6
1 4 4
2 3 3
3 1 7
3 4 1
4 1 5
4 3 12

```



第一行两个数为n和m，n表示顶点个数，m表示边的条数。

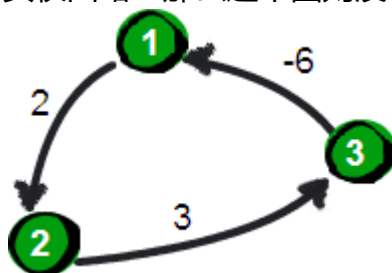
接下来m行，每一行有三个数t1、t2 和t3，表示顶点t1到顶点t2的路程是t3。

得到最终结果如下：

	1	2	3	4
1	0	2	5	4
2	9	0	3	4
3	6	8	0	1
4	5	7	10	0

通过这种方法我们可以求出任意两个点之间最短路径。它的时间复杂度是 $O(N^3)$ 。令人很震撼的是它竟然只有五行代码，实现起来非常容易。正是因为它实现起来非常容易，如果时间复杂度要求不高，使用Floyd-Warshall来求指定两点之间的最短路径或者指定一个点到其余各个顶点的最短路径也是可行的。当然也有更快的算法，请看下一节：Dijkstra算法。

另外需要注意的是：Floyd-Warshall算法不能解决带有“负权回路”（或者叫“负权环”）的图，因为带有“负权回路”的图没有最短路径。例如下面这个图就不存在1号顶点到3号顶点的最短路径。因为1->2->3->1->2->3->...->1->2->3这样路径中，每绕一次1->2->3这样的环，最短路径就会减少1，永远找不到最短路径。其实如果一个图中带有“负权回路”那么这个图则没有最短路径。



此算法由Robert W. Floyd（罗伯特·弗洛伊德）于1962年发表在“Communications of the ACM”上。同年Stephen Warshall（史蒂芬·沃舍尔）也独立发表了这个算法。Robert W. Floyd这个牛人是朵奇葩，他原本在芝加哥大学读的文学，但是因为当时美国经济不太景气，找工作比较困难，无奈之下到西屋电气公司当了一名计算机操作员，在IBM650机房值夜班，并由此开始了他的计算机生涯。此外他还和J.W.J. Williams（威廉姆斯）于1964年共同发明了著名的堆排序算法HEAPSORT。堆排序算法我们将在第七章学习。Robert W. Floyd在1978年获得了图灵奖。

码字不容易啊，转载麻烦注明出处

【一周一算法】算法6：只有五行的Floyd最短路径算法

<http://bbs.ahalei.com/thread-4554-1-1.html>

(出处: 啊哈磊_编程从这里起步)

分类: [啊哈！算法](#)

标签: [超简单Floyd最短路算法](#), [超简单Floyd算法C语言实现](#), [Floyd最短路](#)

绿色通道：

好文要顶

已关注

收藏该文

与我联系





[啊哈磊](#)
[关注 - 0](#)
[粉丝 - 181](#)
我在关注他 [取消关注](#)

80

 推荐

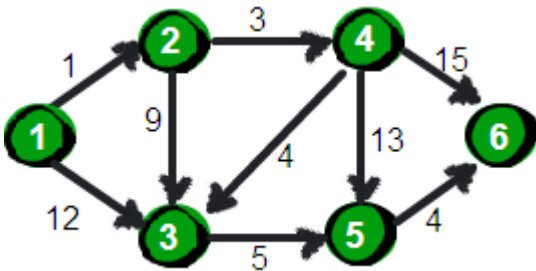
 反对

(请您对文章做出评价)

« 上一篇：[【坐在马桶上看算法】算法5：解密回文——栈](#)
» 下一篇：[【坐在马桶上看算法】算法7：Dijkstra最短路算法](#)

算法7：Dijkstra最短路算法

上周我们介绍了神奇的只有五行的Floyd最短路算法，它可以方便的求得任意两点的最短路径，这称为“多源最短路”。本周来介绍指定一个点（源点）到其余各个顶点的最短路径，也叫做“单源最短路径”。例如求下图中的1号顶点到2、3、4、5、6号顶点的最短路径。



与Floyd-Warshall算法一样这里仍然使用二维数组e来存储顶点之间边的关系，初始值如下。

e	1	2	3	4	5	6
1	0	1	12	∞	∞	∞
2	∞	0	9	3	∞	∞
3	∞	∞	0	∞	5	∞
4	∞	∞	4	0	13	15
5	∞	∞	∞	∞	0	4
6	∞	∞	∞	∞	∞	0

我们还需要用一个一维数组dis来存储1号顶点到其余各个顶点的初始路程，如下。

	1	2	3	4	5	6
dis	0	1	12	∞	∞	∞

我们将此时dis数组中的值称为最短路的“估计值”。

既然是求1号顶点到其余各个顶点的最短路程，那就先找一个离1号顶点最近的顶点。通过数组dis可知当前离1号顶点最近是2号顶点。当选择了2号顶点后，dis[2]的值就已经从“估计值”变为了“确定值”，即1号顶点到2号顶点的最短路程就是当前dis[2]值。为什么呢？你想啊，目前离1号顶点最近的是2号顶点，并且这个图所有的边都是正数，那么肯定不可能通过第三个顶点中转，使得1号顶点到2号顶点的路程进一步缩短了。因为1号顶点到其它顶点的路程肯定没有1号到2号顶点短，对吧O(∩_∩)O~

既然选了2号顶点，接下来再来看2号顶点有哪些出边呢。有2->3和2->4这两条边。先讨论通过2->3这条边能否让1号顶点到3号顶点的路程变短。也就是说现在来比较dis[3]和dis[2]+e[2][3]的大小。其

中dis[3]表示1号顶点到3号顶点的路程。dis[2]+e[2][3]中dis[2]表示1号顶点到2号顶点的路程，e[2][3]表示2->3这条边。所以dis[2]+e[2][3]就表示从1号顶点先到2号顶点，再通过2->3这条边，到达3号顶点的路程。

我们发现dis[3]=12，dis[2]+e[2][3]=1+9=10，dis[3]>dis[2]+e[2][3]，因此dis[3]要更新为10。这个过程有个专业术语叫做“松弛”。即1号顶点到3号顶点的路程即dis[3]，通过2->3这条边松弛成功。这便是Dijkstra算法的主要思想：通过“边”来松弛1号顶点到其余各个顶点的路程。

同理通过2->4 (e[2][4])，可以将dis[4]的值从 ∞ 松弛为4 (dis[4]初始为 ∞ ，dis[2]+e[2][4]=1+3=4，dis[4]>dis[2]+e[2][4]，因此dis[4]要更新为4)。

刚才我们对2号顶点所有的出边进行了松弛。松弛完毕之后dis数组为：

	1	2	3	4	5	6
dis	0	1	10	4	∞	∞

接下来，继续在剩下的3、4、5和6号顶点中，选出离1号顶点最近的顶点。通过上面更新过dis数组，当前离1号顶点最近是4号顶点。此时，dis[4]的值已经从“估计值”变为了“确定值”。下面继续对4号顶点的所有出边 (4->3，4->5和4->6) 用刚才的方法进行松弛。松弛完毕之后dis数组为：

	1	2	3	4	5	6
dis	0	1	8	4	17	19

继续在剩下的3、5和6号顶点中，选出离1号顶点最近的顶点，这次选择3号顶点。此时，dis[3]的值已经从“估计值”变为了“确定值”。对3号顶点的所有出边 (3->5) 进行松弛。松弛完毕之后dis数组为：

	1	2	3	4	5	6
dis	0	1	8	4	13	19

继续在剩下的5和6号顶点中，选出离1号顶点最近的顶点，这次选择5号顶点。此时，dis[5]的值已经从“估计值”变为了“确定值”。对5号顶点的所有出边 (5->4) 进行松弛。松弛完毕之后dis数组为：

	1	2	3	4	5	6
dis	0	1	8	4	13	17

最后对6号顶点所有出边进行松弛。因为这个例子中6号顶点没有出边，因此不用处理。到此，dis数组中所有的值都已经从“估计值”变为了“确定值”。

最终dis数组如下，这便是1号顶点到其余各个顶点的最短路径。

	1	2	3	4	5	6
dis	0	1	8	4	13	17

OK，现在来总结一下刚才的算法。算法的基本思想是：每次找到离源点（上面例子的源点就是1号顶点）最近的一个顶点，然后以该顶点为中心进行扩展，最终得到源点到其余所有点的最短路径。基本步骤如下：

- 将所有的顶点分为两部分：已知最短路径的顶点集合P和未知最短路径的顶点集合Q。最开始，已知最短路径的顶点集合P中只有源点一个顶点。我们这里用一个book[i]数组来记录哪些点在集合P中。例如对于某个顶点i，如果book[i]为1则表示这个顶点在集合P中，如果book[i]为0则表示这个顶点在集合Q中。
- 设置源点s到自己的最短路径为0即dis=0。若存在源点有能直接到达的顶点i，则把dis[i]设为e[s][i]。同时把所有其它（源点不能直接到达的）顶点的最短路径设为 ∞ 。
- 在集合Q的所有顶点中选择一个离源点s最近的顶点u（即dis[u]最小）加入到集合P。并考察所有以点u为起点的边，对每一条边进行松弛操作。例如存在一条从u到v的边，那么可以通过将边u->v添加到尾部来拓展一条从s到v的路径，这条路径的长度是dis[u]+e[u][v]。如果这个值比目前已知的dis[v]的值要小，我们可以用新值来替代当前dis[v]中的值。
- 重复第3步，如果集合Q为空，算法结束。最终dis数组中的值就是源点到所有顶点的最短路径。

完整的Dijkstra算法代码如下：

```
#include <stdio.h>
int main()
{
    int e[10][10],dis[10],book[10],i,j,n,m,t1,t2,t3,u,v,min;
    int inf=99999999; //用inf(infinity的缩写)存储一个我们认为的正无穷值
    //读入n和m, n表示顶点个数, m表示边的条数
    scanf("%d %d",&n,&m);

    //初始化
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
            if(i==j) e[i][j]=0;
            else e[i][j]=inf;

    //读入边
    for(i=1;i<=m;i++)
    {
        scanf("%d %d %d",&t1,&t2,&t3);
        e[t1][t2]=t3;
    }
}
```



```

//初始化dis数组，这里是1号顶点到其余各个顶点的初始路程
for(i=1;i<=n;i++)
    dis[i]=e[1][i];

//book数组初始化
for(i=1;i<=n;i++)
    book[i]=0;
book[1]=1;

//Dijkstra算法核心语句
for(i=1;i<=n-1;i++)
{
    //找到离1号顶点最近的顶点
    min=inf;
    for(j=1;j<=n;j++)
    {
        if(book[j]==0 && dis[j]<min)
        {
            min=dis[j];
            u=j;
        }
    }
    book[u]=1;
    for(v=1;v<=n;v++)
    {
        if(e[u][v]<inf)
        {
            if(dis[v]>dis[u]+e[u][v])
                dis[v]=dis[u]+e[u][v];
        }
    }
}

//输出最终的结果
for(i=1;i<=n;i++)
    printf("%d ",dis[i]);

getchar();
getchar();
return 0;
}

```



可以输入以下数据进行验证。第一行两个整数n m。n表示顶点个数（顶点编号为1~n），m表示边的条数。接下来m行表示，每行有3个数x y z。表示顶点x到顶点y边的权值为z。

```

6 9
1 2 1
1 3 12
2 3 9
2 4 3
3 5 5
4 3 4
4 5 13
4 6 15

```

5 6 4

运行结果是

0 1 8 4 13 17

通过上面的代码我们可以看出，这个算法的时间复杂度是 $O(N^2)$ 即 $O(N^2)$ 。其中每次找到离1号顶点最近的顶点的时间复杂度是 $O(N)$ ，这里我们可以用“堆”（以后再说）来优化，使得这一部分的时间复杂度降低到 $O(\log N)$ 。另外对于边数 M 少于 N^2 的稀疏图来说（我们把 M 远小于 N^2 的图称为稀疏图，而 M 相对较大的图称为稠密图），我们可以用邻接表（这是个神马东西？不要着急，下周再仔细讲解）来代替邻接矩阵，使得整个时间复杂度优化到 $O(M \log N)$ 。请注意！在最坏的情况下 M 就是 N^2 ，这样的话 $M \log N$ 要比 N^2 还要大。但是大多数情况下并不会那么多边，因此 $M \log N$ 要比 N^2 小很多。

欢迎转载，码字不容易啊，转载麻烦注明出处

【一周一算法】算法7：Dijkstra最短路算法

<http://bbs.ahalei.com/thread-4577-1-1.html>

(出处: 啊哈磊_编程从这里起步)

分类: [啊哈！算法](#)

标签: [Dijkstra最短路算法](#), [Dijkstra算法C语言实现](#), [超简单的Dijkstra最短路算法教程](#)

绿色通道：

好文要顶

已关注

收藏该文

与我联系



啊哈磊

关注 - 0

粉丝 - 181

我在关注他 [取消关注](#)

1

推荐

0

反对

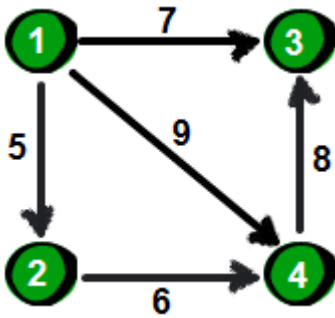
(请您对文章做出评价)

« 上一篇: [【坐在马桶上看算法】算法6：只有五行的Floyd最短路算法](#)

» 下一篇: [【坐在马桶上看算法】算法8：巧妙的邻接表（数组实现）](#)

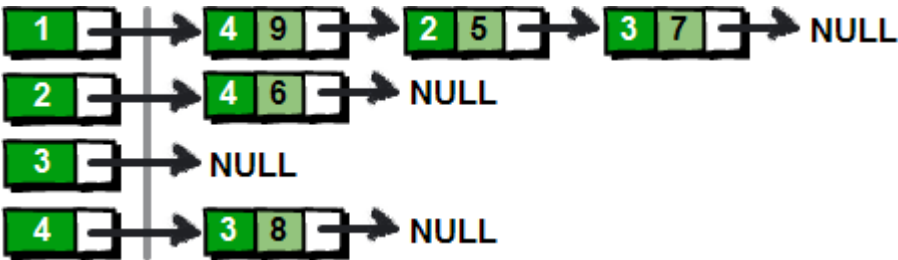
算法8：巧妙的邻接表（数组实现）

之前我们介绍过图的邻接矩阵存储法，它的空间和时间复杂度都是 N^2 ，现在我来介绍另外一种存储图的方法：邻接表，这样空间和时间复杂度就都是 M 。对于稀疏图来说， M 要远远小于 N^2 。先上数据，如下。



```
4 5
1 4 9
4 3 8
1 2 5
2 4 6
1 3 7
```

第一行两个整数 n m 。 n 表示顶点个数（顶点编号为 $1 \sim n$ ）， m 表示边的条数。接下来 m 行表示，每行有3个数 x y z ，表示顶点 x 到顶点 y 的边的权值为 z 。下图就是一种使用链表来实现邻接表的方法。



上面这种实现方法为图中的每一个顶点（左边部分）都建立了一个单链表（右边部分）。这样我们就可以通过遍历每个顶点的链表，从而得到该顶点所有的边了。使用链表来实现邻接表对于痛恨指针的的朋友来说，这简直就是噩梦。这里我将为大家介绍另一种使用数组来实现的邻接表，这是一种在实际应用中非常容易实现的方法。这种方法为每个顶点 i （ i 从 $1 \sim n$ ）也都保存了一个类似“链表”的东西，里面保存的是从顶点 i 出发的所有的边，具体如下。

首先我们按照读入的顺序为每一条边进行编号（ $1 \sim m$ ）。比如第一条边“1 4 9”的编号就是1，“1 3 7”这条边的编号是5。

这里用 u 、 v 和 w 三个数组用来记录每条边的具体信息，即 $u[i]$ 、 $v[i]$ 和 $w[i]$ 表示第 i 条边是从第 $u[i]$ 号顶点到 $v[i]$ 号顶点（ $u[i] \rightarrow v[i]$ ），且权值为 $w[i]$ 。

	U	V	W
1	1	4	9
2	4	3	8
3	1	2	5
4	2	4	6
5	1	3	7

再用一个first数组来存储每个顶点其中一条边的编号。以便待会我们来枚举每个顶点所有的边（你可能会问：存储其中一条边的编号就可以了？不可能吧，每个顶点都需要存储其所有边的编号才行吧！甭着急，继续往下看）。比如1号顶点有一条边是“1 4 9”（该条边的编号是1），那么就将first[1]的值设为1。如果某个顶点i没有以该顶点为起始点的边，则将first[i]的值设为-1。现在来看看具体如何操作，初始状态如下。

② 初始状态

	u	v	w	first	next
1				-1	
2				-1	
3				-1	
4				-1	
5					

咦？上图中怎么多了一个next数组，有什么作用呢？不着急，待会再解释，现在先读入第一条边“1 4 9”。

读入第1条边（1 4 9），将这条边的信息存储到u[1]、v[1]和w[1]中。同时为这条边赋予一个编号，因为这条边是最先读入的，存储在u、v和w数组下标为1的单元格中，因此编号就是1。这条边的起始点是1号顶点，因此将first[1]的值设为1。

另外这条“编号为1的边”是以1号顶点（即u[1]）为起始点的第一条边，所以要将next[1]的值设为-1。也就是说，如果当前这条“编号为i的边”，是我们发现的以u[i]为起始点的第一条边，就将next[i]的值设为-1（貌似的这个next数组很神秘啊○_○）。

① 读入第1条边后

	u	v	w	first	next
1	1	4	9	1	-1
2				-1	
3				-1	
4				-1	
5					

读入第2条边（4 3 8），将这条边的信息存储到u[2]、v[2]和w[2]中，这条边的编号为2。这条边的起始顶点是4号顶点，因此将first[4]的值设为2。另外这条“编号为2的边”是我们发现以4号顶点为起始点的第一条边，所以将next[2]的值设为-1。

② 读入第2条边后

	u	v	w	first	next
1	1	4	9	1	-1
2	4	3	8	2	-1
3				3	-1
4				4	2
5					

读入第3条边（1 2 5），将这条边的信息存储到u[3]、v[3]和w[3]中，这条边的编号为3，起始顶点是1号顶点。我们发现1号顶点已经有一条“编号为1的边”了，如果此时将first[1]的值设为3，那“编号为1的边”岂不是就丢失了？我有办法，此时只需将next[3]的值设为1即可。现在你知道next数组是用来做什么的吧。next[i]存储的是“编号为i的边”的“前一条边”的编号。

③ 读入第3条边后

	u	v	w	first	next
1	1	4	9	1	-1
2	4	3	8	2	-1
3	1	2	5	3	1
4				4	2
5					

读入第4条边（2 4 6），将这条边的信息存储到u[4]、v[4]和w[4]中，这条边的编号为4，起始顶点是2号顶点，因此将first[2]的值设为4。另外这条“编号为4的边”是我们发现以2号顶点为起始点的第一条边，所以将next[4]的值设为-1。

④ 读入第4条边后

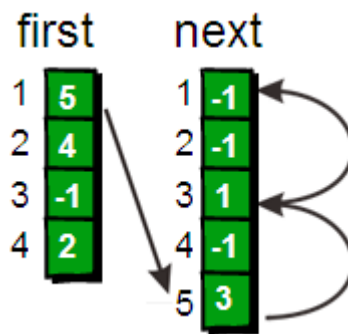
	u	v	w	first	next
1	1	4	9	1	-1
2	4	3	8	2	-1
3	1	2	5	3	1
4	2	4	6	4	-1
5					

读入第5条边（1 3 7），将这条边的信息存储到u[5]、v[5]和w[5]中，这条边的编号为5，起始顶点又是1号顶点。此时需要将first[1]的值设为5，并将next[5]的值改为3。

⑤ 读入第5条边后

	u	v	w		first		next
1	1	4	9	1	5	1	-1
2	4	3	8	2	4	2	-1
3	1	2	5	3	-1	3	1
4	2	4	6	4	2	4	-1
5	1	3	7	5		5	3

此时，如果我们想遍历1号顶点的每一条边就很简单了。1号顶点的其中一条边的编号存储在first[1]中。其余的边则可以通过next数组寻找到。请看下图。



细心的同学会发现，此时遍历边某个顶点边的时候的遍历顺序正好与读入时候的顺序相反。因为在为每个顶点插入边的时候都直接插入“链表”的首部而不是尾部。不过这并不会产生任何问题，这正是这种方法的其妙之处。

创建邻接表的代码如下。



```
int n,m,i;
//u、v和w的数组大小要根据实际情况来设置，要比m的最大值要大1
int u[6],v[6],w[6];
//first和next的数组大小要根据实际情况来设置，要比n的最大值要大1
int first[5],next[5];
scanf("%d %d",&n,&m);
//初始化first数组下标1~n的值为-1，表示1~n顶点暂时都没有边
for(i=1;i<=n;i++)
    first[i]=-1;
for(i=1;i<=m;i++)
{
    scanf("%d %d %d",&u[i],&v[i],&w[i]); //读入每一条边
    //下面两句是关键啦
    next[i]=first[u[i]];
    first[u[i]]=i;
}
```



接下来如何遍历每一条边呢？我们之前说过其实first数组存储的就是每个顶点i（i从1~n）的第一条边。比如1号顶点的第一条边是编号为5的边（1 3 7），2号顶点的第一条边是编号为4的边（2 4 6），3号顶点没有出向边，4号顶点的第一条边是编号

为2的边 (2 4 6)。那么如何遍历1号顶点的每一条边呢？也很简单。请看下图：

遍历1号顶点所有边的代码如下。

```
k=first[1]; // 1号顶点其中的一条边的编号（其实也是最后读入的边）
while(k!=-1) //其余的边都可以在next数组中依次找到
{
    printf("%d %d %d\n",u[k],v[k],w[k]);
    k=next[k];
}
```

遍历每个顶点的所有边的代码如下。

```
for(i=1;i<=n;i++)
{
    k=first[i];
    while(k!=-1)
    {
        printf("%d %d %d\n",u[k],v[k],w[k]);
        k=next[k];
    }
}
```

可以发现使用邻接表来存储图的时间空间复杂度是 $O(M)$ ，遍历每一条边的时间复杂度也是 $O(M)$ 。如果一个图是稀疏图的话， M 要远小于 N^2 。因此稀疏图选用邻接表来存储要比邻接矩阵来存储要好很多。

欢迎转载，码字不容易啊，转载麻烦注明出处

【啊哈！算法】系列8：巧妙的邻接表（数组实现） <http://www.cnblogs.com/ahalei/p/3651334.html>

分类: [啊哈！算法](#)

标签: [邻接链表C语言代码](#), [邻接表](#), [邻接表的数组实现](#), [邻接表C语言代码](#), [邻接链表C语言实现](#)

绿色通道：

好文要顶

已关注

收藏该文

与我联系



啊哈磊

关注 - 0

粉丝 - 181

我在关注他 [取消关注](#)

2

推荐

0

反对

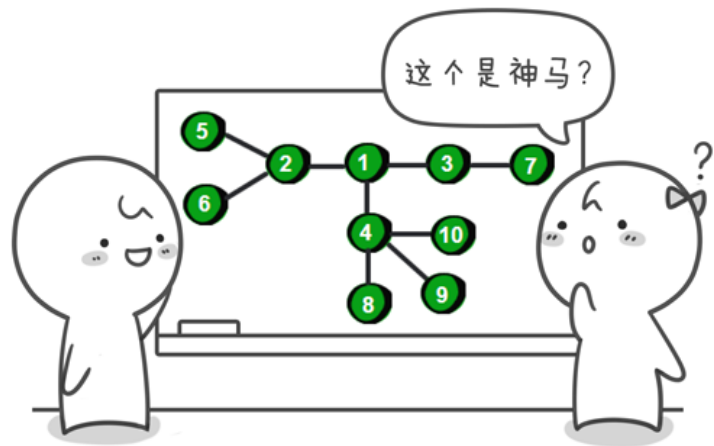
(请您对文章做出评价)

« 上一篇：[【坐在马桶上看算法】算法7：Dijkstra最短路算法](#)

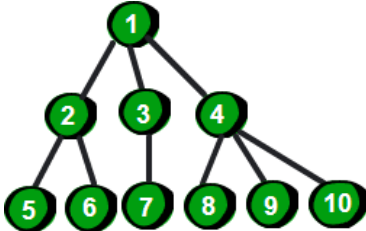
» 下一篇：[【坐在马桶上看算法】算法9：开启“树”之旅](#)

【坐在马桶上看算法】算法9：开启“树”之旅

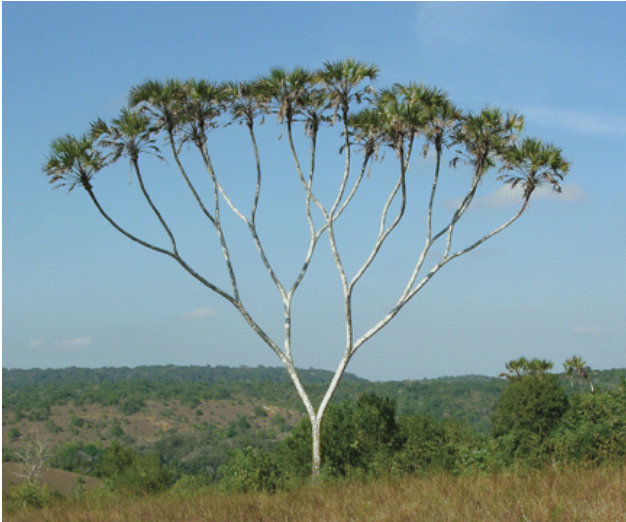
我们先来看一个例子。



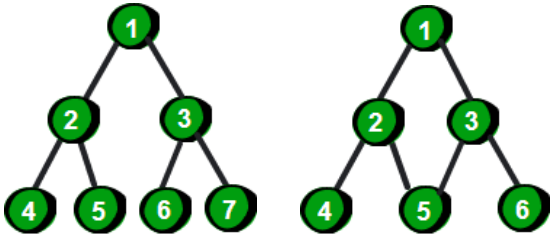
这是什么？是一个图？不对，确切的说这是一棵树。这哪里像树呢？不要着急我们来变换一下。



是不是很像一棵倒挂的树，也就是说它是根朝上，而叶子朝下的。不像？哈哈，看完下面这幅图你就会觉得像啦。



你可能会问：树和图有什么区别？这个称之为树的东西貌似和无向图差不多嘛。不要着急，继续往下看。树其实就是不包含回路的连通无向图。你可能还是无法理解这其中的差异，举个例子，如下。



上面这个例子中左边的是一棵树，而右边的是一个图。因为左边的没有回路，而右边的存在1->2->5->3->1这样的回路。

- 1、正是因为树有着“不包含回路”这个特点，所以树就被赋予了很多人特性。
- 2、一棵树中的任意两个结点有且仅有一条路径连通。
- 3、一棵树如果有n个结点，那么它一定恰好有n-1条边。

在一棵树中加一条边将会构成一个回路。树这个特殊的数据结构在哪里会用到呢？比如足球世界杯的晋级图，家族的族谱图、公司的组织结构图、书的目录、我们用的操作系统Windows、Linux或者Mac中的“目录（文件夹）”都是一棵树。下面就是“啊哈C”这个软件的目录结构。

1	C:\啊哈C
2	├codes
3	├core
4	─bin

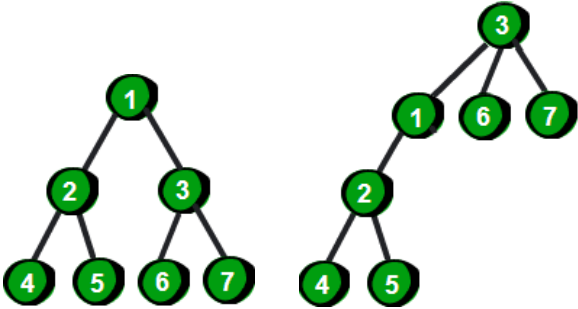

```
5 | | | include
6 | | | ddk
7 | | | gdb
8 | | | gdiplus
9 | | | GL
10 | | | sys
11 | | lib
12 | | | gcc
13 | | | mingw32
14 | | | 4.7.1
15 | | | | include
16 | | | | | include
17 | | | | | | ssp
18 | | | | | include-fixed
19 | | | | | install-tools
20 | | | | | include
21 | | | libexec
22 | | | | gcc
23 | | | | mingw32
24 | | | | 4.7.1
25 | | | | | install-tools
26 | | | | | mingw32
27 | | | bin
28 | | | | lib
29 | | | | | ldscripts
30 | | | | | skin
```

假如现在正处于libexec文件夹下，需要到gdiplus文件夹下。你必须先“向上”回到上层文件夹core，再进入include文件夹，最后才能进入gdiplus文件夹。因为一棵树中的任意两个结点（这里就是文件夹）有且仅有唯一的一条路径连通。

为了之后讲解的方便，我们这里对树进行一些定义。

首先，树是指任意两个结点间有且只有一条路径的无向图。或者说，只要是没有回路的连通无向图就是树。

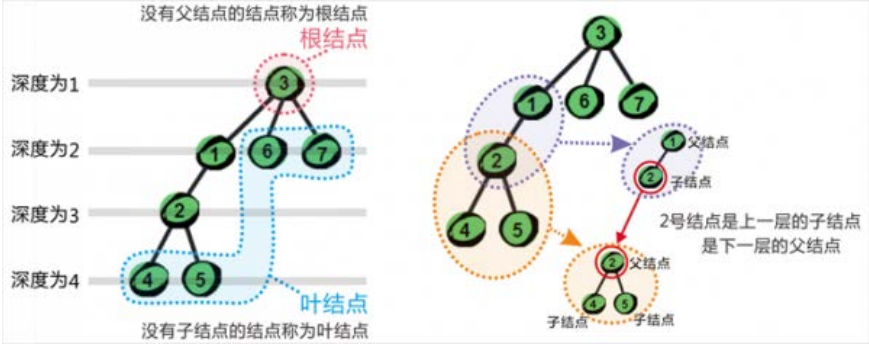
喜欢思考的同学可能会发现同一棵树可以有多种形态，比如下面这个两棵树。



为了确定一棵树的形态，在一棵树中可以指定一个特殊的结点——根。我们在对一棵树进行讨论的时候，将树中的每个点称为结点，有的书中也称为节点。有一个根的书叫做有根树（哎，这不是废话嘛）。比如上方左边这棵树的树根是1号结点，右边这棵树的树根是3号结点。

根又叫做根结点，一棵树有且只有一个根结点。根结点有时候也称为祖先。既然有祖先，理所当然就有父亲和儿子。比如上图右边这棵树中3号结点是1、6和7号结点的父亲，1、6和7号结点是3号结点的儿子。同时1号结点又是2号结点的父亲，2号结点是1号结点的儿子，2号结点与4、5号结点关系也显而易见了。

父亲结点简称为父结点，儿子结点简称为子结点。2号结点既是父结点也是子结点，它是1号结点的子结点，同时也是4和5号结点的父结点。另外如果一个结点没有子结点（即没有儿子）那么这个结点称为叶结点，例如4、5、6和7号结点都是叶结点。没有父结点（即没有父亲）的结点称为根结点（祖先）。如果一个结点既不是根结点也不是叶结点则称为内部结点。最后每个结点还有深度，比如5号结点的深度是4。哎，终于啰嗦完了，写的我汗都流出来了，没有理解的请看下面这幅插图吧。



说了这么多你可能都没有感受到树究竟有什么好处。不要着急，请看下回——二叉树。

欢迎转载，码字不容易啊，转载麻烦注明出处

【啊哈！算法】算法9：开启“树”之旅 <http://bbs.ahalei.com/thread-4684-1-1.html>

分类: [啊哈！算法](#)

标签: [数据结构树](#), [树的实现](#), [什么是树](#)

绿色通道：[好文要顶](#) [已关注](#) [收藏该文](#) [与我联系](#) 



啊哈磊

关注 - 0

粉丝 - 181

我在关注他 [取消关注](#)

10
 推荐

0
 反对

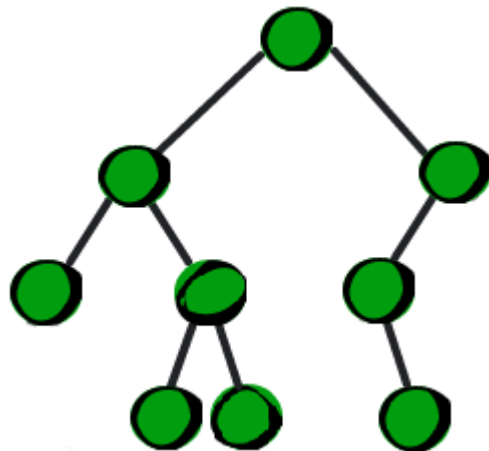
(请您对文章做出评价)

« 上一篇：[【坐在马桶上看算法】算法8：巧妙的邻接表（数组实现）](#)

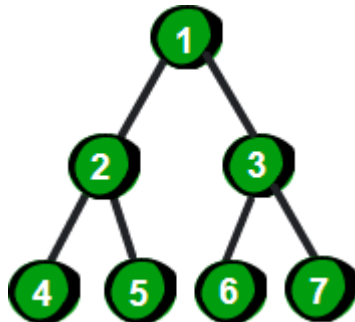
» 下一篇：[【坐在马桶上看算法】算法10：二叉树](#)

算法10：二叉树

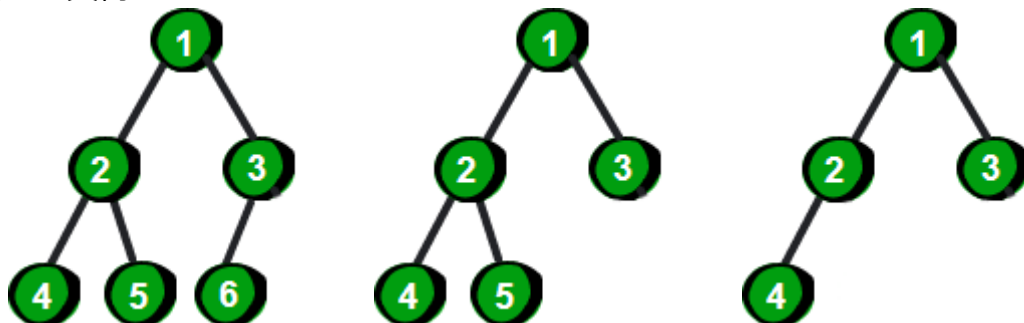
二叉树是一种特殊的树。二叉树的特点是每个结点最多有两个儿子，左边的叫做左儿子，右边的叫做右儿子，或者说每个结点最多有两棵子树。更加严格的递归定义是：二叉树要么为空，要么由根结点、左子树和右子树组成，而左子树和右子树分别是一棵二叉树。下面这棵树就是一棵二叉树。



二叉树的使用范围最广，一棵多叉树也可以转化为二叉树，因此我们将着重讲解二叉树。二叉树中还有连两种特殊的二叉树叫做满二叉树和完全二叉树。如果二叉树中每个内部结点都有两个儿子，这样的二叉树叫做满二叉树。或者说满二叉树所有的叶结点都有同样的深度。比如下面这棵二叉树，是不是感觉很“丰满”。满二叉树的严格的定义是一棵深度为 h 且有 2^h-1 个结点的二叉树。



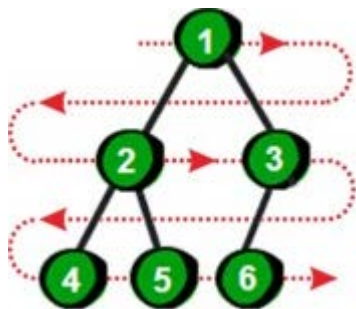
如果一棵二叉树除了最右边位置上一个或者几个叶结点缺少外其它是丰满的，那么这样的二叉树就是完全二叉树。严格的定义是：若设二叉树的高度为 h ，除第 h 层外，其它各层 $(1 \sim h-1)$ 的结点数都达到最大个数，第 h 层从右向左连续缺若干结点，就是完全二叉树。也就是说如果一个结点有右子结点，那么它一定也有左子结点。例如下面这三棵树都是完全二叉树。其实你可以将满二叉树理解成是一种特殊的或者极其完美的完全二叉树。



其实完全二叉树类似下面这个形状。



说到这里我们马上就要领略到完全二叉树的魅力了。先想一想一棵完全二叉树如何存储呢？其实完全二叉树中父亲和儿子之间有着神奇的规律，我们只需用一个一维数组就可以存储完全二叉树。首先将完全二叉树进行从上到下，从左到右编号。



通过上图我们发现如果完全二叉树的一个父结点编号为 k ，那么它左儿子的编号就是 $2*k$ ，右儿子的编号就是 $2*k+1$ 。如果已知儿子（左儿子或右儿子）的编号是 x ，那么它父结点的编号就是 $x/2$ ，注意这里只取商的整数部分。在C语言中如果除号‘/’两边都是整数的话，那么商也只有整数部分（即自动向下取整），即 $4/2$ 和 $5/2$ 都是2。另外如果一棵完全二叉树有 N 个结点，那么这个完全二叉树的高度为 $\log_2 N$ 简写为 $\log N$ ，即最多有 $\log N$ 层结点。完全二叉树的最典型应用就是——堆。那么堆又有什么作用呢？请关注下周更新：堆——神奇的优先队列。

欢迎转载，码字不容易啊，转载麻烦注明出处

《啊哈！算法》算法10：二叉树 <http://www.ahalei.com/thread-4850-1-1.html>

分类: [啊哈！算法](#)

标签: [二叉树](#), [C语言二叉树](#)

绿色通道：

[好文要顶](#)

[已关注](#)

[收藏该文](#)

[与我联系](#)



啊哈磊

[关注 - 0](#)

[粉丝 - 181](#)

我在关注他 [取消关注](#)

1

[推荐](#)

0

[反对](#)

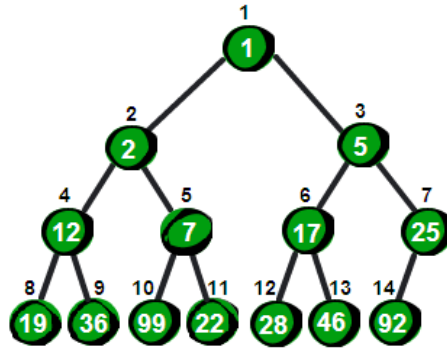
(请您对文章做出评价)

« 上一篇：[【坐在马桶上看算法】算法9：开启“树”之旅](#)

» 下一篇：[【坐在马桶上看算法】算法11：堆——神奇的优先队列（上）](#)

【坐在马桶上看算法】算法11：堆——神奇的优先队列（上）

堆是什么？是一种特殊的完全二叉树，就像下面这棵树一样。



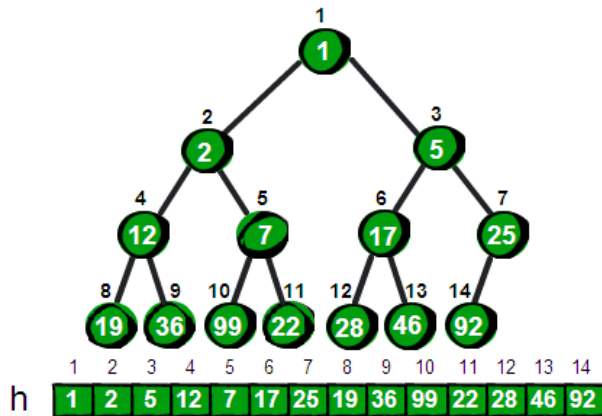
有没有发现这棵二叉树有一个特点，就是所有父结点都比子结点要小（注意：圆圈里面的数是值，圆圈上面的数是这个结点的编号，此规定仅适用于本节）。符合这样特点的完全二叉树我们称为最小堆。反之，如果所有父结点都比子结点要大，这样的完全二叉树称为最大堆。那这一特性究竟有什么用呢？

假如有14个数分别是99、5、36、7、22、17、46、12、2、19、25、28、1和92。请找出这14个数中最小的数，请问怎么办呢？最简单的方法就是将这14个数从头到尾依次扫一遍，用一个循环就可以解决。这种方法的时间复杂度是 $O(14)$ 也就是 $O(N)$ 。

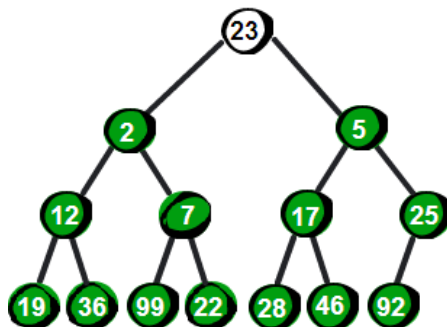
```
for(i=1;i<=14;i++)
{
    if(a[i]<min)    min=a[i];
}
```

现在我们需要删除其中最小的数，并增加一个新数23，再次求这14个数中最小的一个数。请问该怎么办呢？只能重新扫描所有的数，才能找到新的最小的数，这个时间复杂度也是 $O(N)$ 。假如现在有14次这样的操作（删除最小的数后并添加一个新数）。那么整个时间复杂度就是 $O(14^2)$ 即 $O(N^2)$ 。那有没有更好的方法呢？堆这个特殊的结构恰好能够很好地解决这个问题。

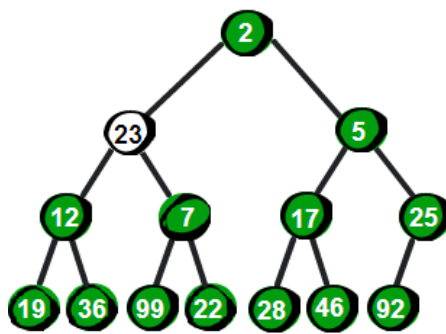
首先我们先把这个14个数按照最小堆的要求（就是所有父结点都比子结点要小）放入一棵完全二叉树，就像下面这棵树一样。



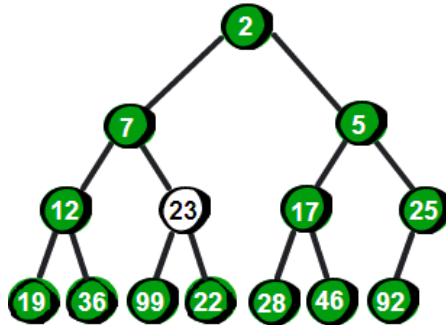
很显然最小的数就在堆顶，假设存储这个堆的数组叫做h的话，最小数就是 $h[1]$ 。接下来，我们将堆顶的数删除，并将新增加的数23放到堆顶。显然加了新数后已经不符合最小堆的特性，我们需要将新增加的数调整到合适的位置。那如何调整呢？



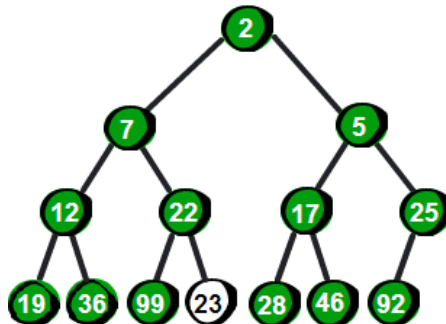
向下调整！我们需要将这个数与它的两个儿子2和5比较，并选择较小一个与它交换，交换之后如下。



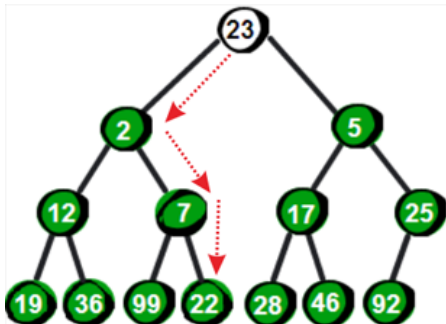
我们发现此时还是不符合最小堆的特性，因此还需要继续向下调整。于是继续将23与它的两个儿子12和7比较，并选择较小一个交换，交换之后如下。



到此，还是不符合最小堆的特性，仍需要继续向下调整直到符合最小堆的特性为止。



我们发现现在已经符合最小堆的特性了。综上所述，当新增加一个数被放置到堆顶时，如果此时不符合最小堆的特性，则需要将这个数向下调整，直到找到合适的位置为止，使其重新符合最小堆的特性。



向下调整的代码如下。

```
void siftDown(int i) //传入一个需要向下调整的结点编号i，这里传入1，即从堆的顶点开始向下调整
{
    int t, flag=0; //flag用来标记是否需要继续向下调整
    //当i结点有儿子的时候（其实是至少有左儿子的情况下）并且有需要继续调整的时候循环执行
    while( i*2<=n && flag==0 )
    {
        //首先判断他和他左儿子的关系，并用t记录值较小的结点编号
        if( h[ i ] > h[ i*2 ] )
            t=i*2;
        else
            t=i;
        //如果有右儿子的情况下，再对右儿子进行讨论
        if( i*2+1 <= n )
```

```

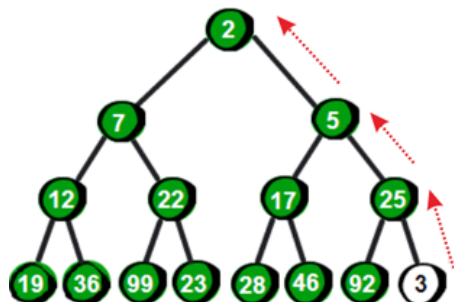
{
    //如果右儿子的值更小，更新较小的结点编号
    if(h[ t] > h[ i*2+1])
        t=i*2+1;
}
//如果发现最小的结点编号不是自己，说明子结点中有比父结点更小的
if(t!=i)
{
    swap(t,i); //交换它们，注意swap函数需要自己来写
    i=t; //更新i为刚才与它交换的儿子结点的编号，便于接下来继续向下调整
}
else
    flag=1; //则说明当前的父结点已经比两个子结点都要小了，不需要在进行调整了
}
}

```

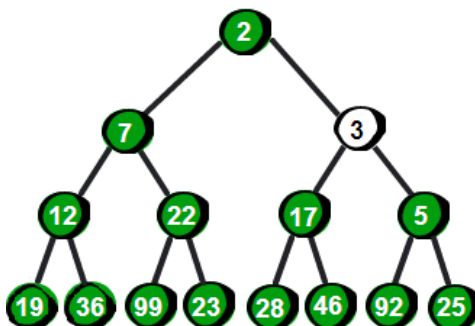


我们刚才在对23进行调整的时候，竟然只进行了3次比较，就重新恢复了最小堆的特性。现在最小的数依然在堆顶为2。之前那种从头到尾扫描的方法需要14次比较，现在只需要3次就够了。现在每次删除最小的数并新增一个数，并求当前最小数的时间复杂度是 $O(3)$ ，这恰好是 $O(\log_2 14)$ 即 $O(\log_2 N)$ 简写为 $O(\log N)$ 。假如现在有1亿个数（即 $N=1$ 亿），进行1亿次删除最小数并新增一个数的操作，使用原来扫描的方法计算机需要运行大约1亿的平方次，而现在只需要1亿 $\times \log_2 1$ 亿次，即27亿次。假设计算机每秒钟可以运行10亿次，那原来则需要一千万秒大约115天！而现在只要2.7秒。是不是很神奇，再次感受到算法的伟大了吧。

说到这里，如果只是想新增一个值，而不是删除最小值又该如何操作呢？即如何在原有的堆上直接插入一个新元素呢？只需要直接将新元素插入到末尾，再根据情况判断新元素是否需要上移，直到满足堆的特性为止。如果堆的大小为 N （即有 N 个元素），那么插入一个新元素所需要的时间也是 $O(\log N)$ 。例如我们现在要新增一个数3。



先将3与它的父结点25比较，发现比父结点小，为了维护最小堆的特性，需要与父结点的值进行交换。交换之后发现还是要比它此时的父结点5小，因此需要再次与父结点交换。至此又重新满足了最小堆的特性。向上调整完毕后如下。



向上调整的代码如下。



```

void siftup(int i) //传入一个需要向上调整的结点编号i
{
    int flag=0; //用来标记是否需要继续向上调整
    if(i==1) return; //如果是堆顶，就返回，不需要调整了
    //不在堆顶 并且 当前结点i的值比父结点小的时候继续向上调整
    while(i!=1 && flag==0)
    {
        //判断是否比父结点的小
    }
}

```



```
if(h[i]<h[i/2])
    swap(i,i/2); //交换他和爸爸的位置
else
    flag=1; //表示已经不需要调整了,当前结点的值比父结点的值要大
i=i/2; //这句话很重要,更新编号i为它父结点的编号,从而便于下一次继续向上调整
}
```

说了半天，我们忽略一个很重要的问题！就是如何建立这个堆。我们周一接着说。

BTW，《啊哈！算法》系列，坐在马桶上都能看懂的算法入门书，已经整理出版，下周一将是最后一次在线更新啦（把堆说完）。各位喜欢《啊哈！算法》的朋友要去买一本收藏哦😁 这年头写个东西不容易，多谢大家支持啦👏，当当网购买链接 <http://product.dangdang.com/23490849.html>

买了的朋友记得来啥单，还可以得到《啊哈！算法》的T恤哦~~~ <http://www.ahalei.com/thread-4969-1-1.html>



分类: [啊哈！算法](#)

标签: [堆](#), [二叉堆](#), [堆的C语言实现](#)

绿色通道：[好文要顶](#) [已关注](#) [收藏该文](#) [与我联系](#)

[啊哈磊](#)
[关注 - 0](#)
[粉丝 - 181](#)
我在关注他 [取消关注](#)


[4](#) [0](#)
[推荐](#) [反对](#)
(请您对文章做出评价)

« 上一篇: [【坐在马桶上看算法】算法10：二叉树](#)


» 下一篇: [【坐在马桶上看算法】算法12：堆——神奇的优先队列（下）](#)

算法12：堆——神奇的优先队列（下）

接着上一Pa说。就是如何建立这个堆呢。可以从空的堆开始，然后依次往堆中插入每一个元素，直到所有数都被插入（转移到堆中为止）。因为插入第*i*个元素的所用的时间是 $O(\log i)$ ，所以插入所有元素的整体时间复杂度是 $O(N\log N)$ ，代码如下。

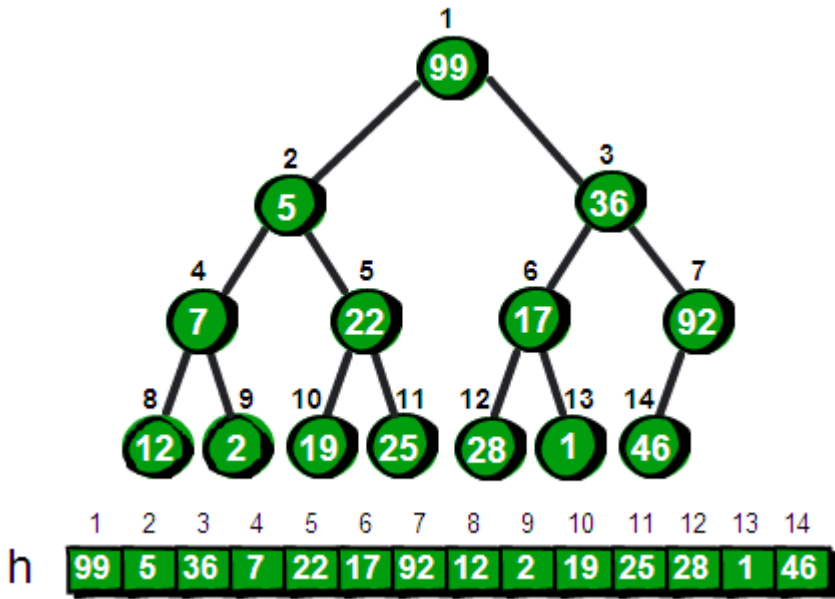


```
n=0;
for(i=1;i<=m;i++)
{
    n++;
    h[n]=a[i]; //或者写成scanf("%d",&h[n]);
    siftup();
}
```



其实我们还有更快得方法来建立堆。它是这样的。

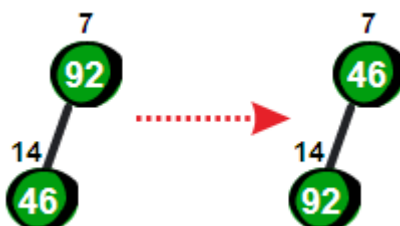
直接把99、5、36、7、22、17、46、12、2、19、25、28、1和92这14个数放入一个完全二叉树中（这里我们还是用一个一维数组来存储完全二叉树）。



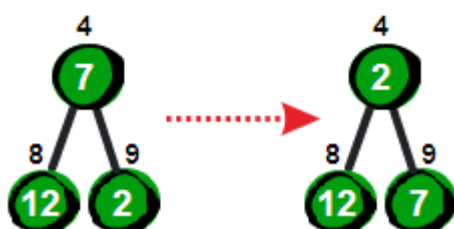
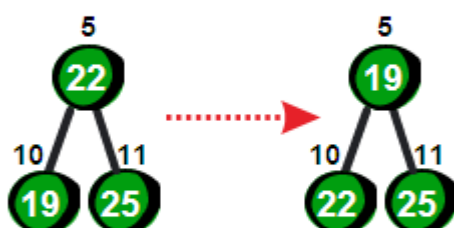
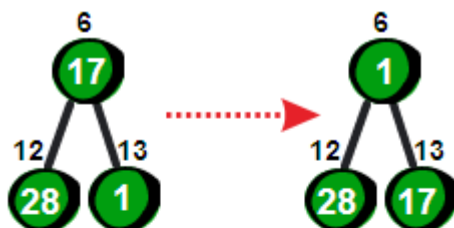
在这个棵完全二叉树中，我们从最后一个结点开始依次判断以这个结点为根的子树是否符合最小堆的特性。如果所有的子树都符合最小堆的特性，那么整棵树就是最小堆了。如果这句话没有理解不要着急，继续往下看。

首先我们从叶结点开始。因为叶结点没有儿子，所以所有以叶结点为根结点的子树（其实这个子树只有一个结点）都符合最小堆的特性（即父结点的值比子结点的值小）。这些叶结点压根就没有子节点，当然符合这个特性。因此所有叶结点都不需要处理，直接跳过。从第*n*/2个结点（*n*为完全二叉树的结点总数，这里即7号结点）开始处理这棵完全二叉树。注意完全二叉树有一个性质：最后一个非叶结点是第*n*/2个结点。

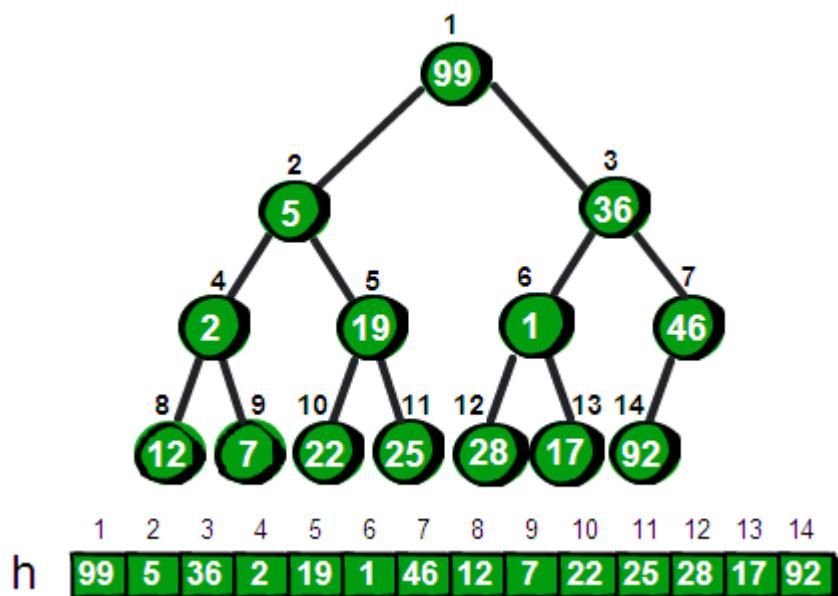
以7号结点为根的子树不符合最小堆的特性，因此要向下调整。



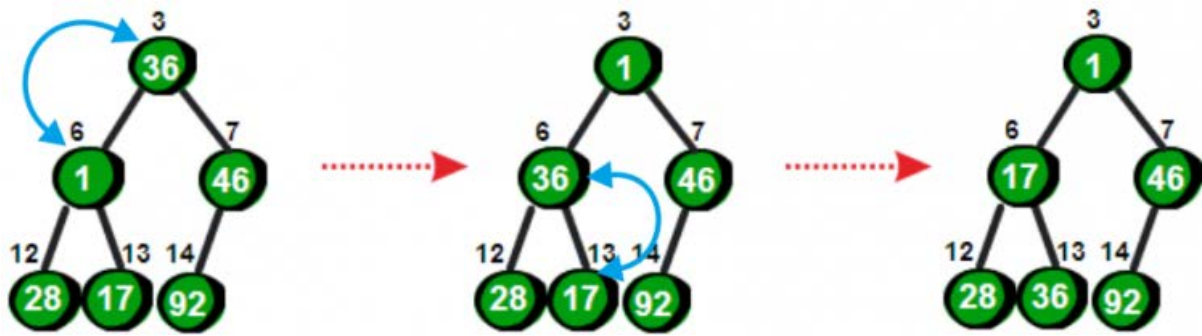
同理以6号、5号和4结点为根的子树也不符合最小堆的特性，都需要往下调整。



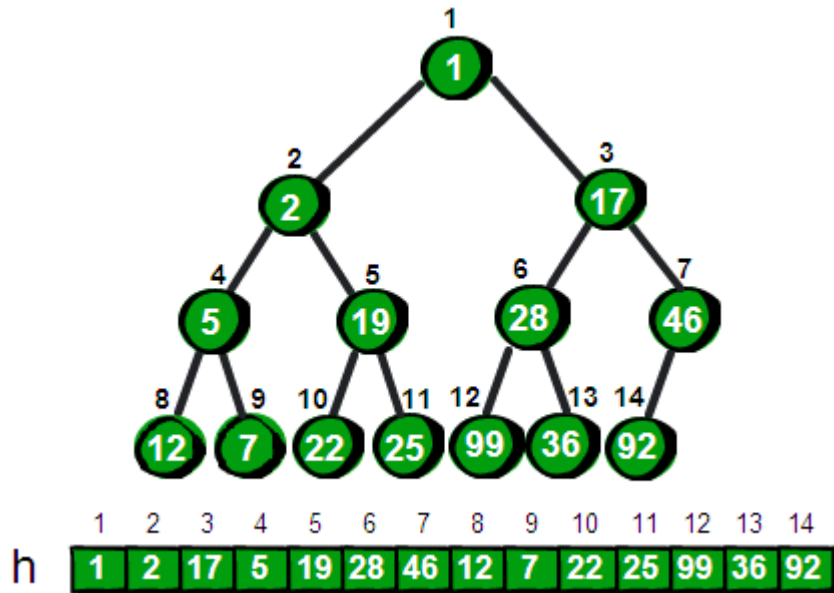
下面是已经对7号、6号、5号和4结点为根结点的子树调整完毕之后的状态。



当然目前这棵树仍然不符合最小堆的特性，我们需要继续调整以3号结点为根的子树，即将3号结点向下调整。



同理继续调整以2号结点为根的子树，最后调整以1号结点为根的子树。调整完毕之后，整棵树就符合最小堆的特性啦。



小结一下这个创建堆的算法。把 n 个元素建立一个堆，首先我可以将这 n 个结点以自顶向下、从左到右的方式从1到 n 编码。这样就可以把这 n 个结点转换成为一棵完全二叉树。紧接着从最后一个非叶结点（结点编号为 $n/2$ ）开始到根结点（结点编号为1），逐个扫描所有的结点，根据需要将当前结点向下调整，直到以当前结点为根结点的子树符合堆的特性。虽然讲起来起来很复杂，但是实现起来却很简单，只有两行代码如下：

```
for(i=n/2;i>=1;i--)
    siftdown(i);
```

用这种方法来建立一个堆的时间复杂度是 $O(N)$ ，如果你感兴趣可以尝试自己证明一下，嘿嘿。

堆还有一个作用就是堆排序，与快速排序一样堆排序的时间复杂度也是 $O(N\log N)$ 。堆排序的实现很简单，比如我们现在要进行从小到大排序，可以先建立最小堆，然后每次删除顶部元素并将顶部元素输出或者放入一个新的数组中，直到堆为空为止。最终输出的或者存放在新数组中数就已经是排序好的了。



```
//删除最大的元素
int deletemax()
{
    int t;
    t=h[ 1]; //用一个临时变量记录堆顶点的值
    h[ 1]=h[ n]; //将堆得最后一个点赋值到堆顶
    n--; //堆的元素减少1
    siftdown(1); //向下调整
    return    //
```

```
    t;    返回之前记录的堆得顶点的最大值  
}
```



建堆以及堆排序的完整代码如下：



```
#include <stdio.h>

int h[ 101]; //用来存放堆的数组
int n; //用来存储堆中元素的个数，也就是堆的大小

//交换函数，用来交换堆中的两个元素的值
void swap(int x,int y)
{
    int t;
    t=h[ x];
    h[ x]=h[ y];
    h[ y]=t;
}

//向下调整函数
void siftdown(int i) //传入一个需要向下调整的结点编号i，这里传入1，即从堆的顶点开始向下调整
{
    int t,flag=0; //flag用来标记是否需要继续向下调整
    //当i结点有儿子的时候（其实是至少有左儿子的情况下）并且有需要继续调整的时候循环室执行
    while( i*2<=n && flag==0 )
    {
        //首先判断他和他左儿子的关系，并用t记录值较小的结点编号
        if( h[ i] > h[ i*2] )
            t=i*2;
        else
            t=i;
        //如果有右儿子的情况下，再对右儿子进行讨论
        if(i*2+1 <= n)
        {
            //如果右儿子的值更小，更新较小的结点编号
            if(h[ t] > h[ i*2+1])
                t=i*2+1;
        }
        //如果发现最小的结点编号不是自己，说明子结点中有比父结点更小的
        if(t!=i)
        {
            swap(t,i); //交换它们，注意swap函数需要自己来写
        }
    }
}
```

```

        i=t; //更新i为刚才与它交换的儿子结点的编号，便于接下来继续向下调整
    }
    else
        flag=1; //则否说明当前的父结点已经比两个子结点都要小了，不需要在进行调整了
    }
}

//建立堆的函数
void creat()
{
    int i;
    //从最后一个非叶结点到第1个结点依次进行向上调整
    for(i=n/2; i>=1; i--)
    {
        sifttdown(i);
    }
}

//删除最大的元素
int deletemax()
{
    int t;
    t=h[ 1]; //用一个临时变量记录堆顶点的值
    h[ 1]=h[ n]; //将堆得最后一个点赋值到堆顶
    n--; //堆的元素减少1
    sifttdown(1); //向下调整
    return t; //返回之前记录的堆得顶点的最大值
}

int main()
{
    int i,num;
    //读入数的个数
    scanf( "%d", &num);

    for(i=1; i<=num; i++)
        scanf( "%d", &h[ i]);
    n=num;

    //建堆
    creat();

    //删除顶部元素，连续删除n次，其实夜就是从大到小把数输出来
    for(i=1; i<=num; i++)
        printf( "%d ", deletemax());

    getchar();
}

```

```
getchar();  
return 0;  
}
```



可以输入以下数据进行验证

14

99 5 36 7 22 17 46 12 2 19 25 28 1 92

运行结果是

1 2 5 7 12 17 19 22 25 28 36 46 92 99

当然堆排序还有一种更好的方法。从小到大排序的时候不建立最小堆而建立最大堆。最大堆建立好后，最大的元素在 $h[1]$ 。因为我们的需求是从小到大排序，希望最大的放在最后。因此我们将 $h[1]$ 和 $h[n]$ 交换，此时 $h[n]$ 就是数组中的最大的元素。请注意，交换后还需将 $h[1]$ 向下调整以保持堆的特性。OK现在最大的元素已经归位，需要将堆的大小减1即 $n--$ ，然后再将 $h[1]$ 和 $h[n]$ 交换，并将 $h[1]$ 向下调整。如此反复，直到堆的大小变成1为止。此时数组 h 中的数就已经是排序好的了。代码如下：



```
//堆排序  
void heapsort()  
{  
    while(n>1)  
    {  
        swap(1,n);  
        n--;  
        siftDown(1);  
    }  
}
```



完整的堆排序的代码如下，注意使用这种方法来进行从小到大排序需要建立最大堆。



```
#include <stdio.h>  
  
int h[ 101]; //用来存放堆的数组  
int n; //用来存储堆中元素的个数，也就是堆的大小  
  
//交换函数，用来交换堆中的两个元素的值  
void swap(int x,int y)  
{  
    int t;  
    t=h[ x];
```

```

    h[ x]=h[ y];
    h[ y]=t;
}

//向下调整函数
void sifttdown(int i) //传入一个需要向下调整的结点编号i, 这里传入1, 即从堆的顶点开始向下调整
{
    int t, flag=0; //flag用来标记是否需要继续向下调整
    //当i结点有儿子的时候 (其实是至少有左儿子的情况下) 并且有需要继续调整的时候循环室执行
    while( i*2<=n && flag==0 )
    {
        //首先判断他和他左儿子的关系, 并用t记录值较大的结点编号
        if( h[ i] < h[ i*2] )
            t=i*2;
        else
            t=i;
        //如果有右儿子的情况下, 再对右儿子进行讨论
        if(i*2+1 <= n)
        {
            //如果右儿子的值更大, 更新较小的结点编号
            if(h[ t] < h[ i*2+1])
                t=i*2+1;
        }
        //如果发现最大的结点编号不是自己, 说明子结点中有比父结点更大的
        if(t!=i)
        {
            swap(t,i); //交换它们, 注意swap函数需要自己来写
            i=t; //更新i为刚才与它交换的儿子结点的编号, 便于接下来继续向下调整
        }
        else
            flag=1; //则否说明当前的父结点已经比两个子结点都要大了, 不需要在进行调整了
    }
}

//建立堆的函数
void creat()
{
    int i;
    //从最后一个非叶结点到第1个结点依次进行向上调整
    for(i=n/2; i>=1; i--)
    {
        sifttdown(i);
    }
}

//堆排序

```

```

void heapsort()
{
    while(n>1)
    {
        swap(1,n);
        n--;
        siftdown(1);
    }
}

int main()
{
    int i,num;
    //读入n个数
    scanf("%d",&num);

    for(i=1;i<=num;i++)
        scanf("%d",&h[i]);
    n=num;

    //建堆
    creat();

    //堆排序
    heapsort();

    //输出
    for(i=1;i<=num;i++)
        printf("%d ",h[i]);

    getchar();
    getchar();
    return 0;
}

```



可以输入以下数据进行验证

14

99 5 36 7 22 17 46 12 2 19 25 28 1 92

运行结果是

1 2 5 7 12 17 19 22 25 28 36 46 92 99

OK，最后还是要总结一下。像这样支持插入元素和寻找最大（小）值元素的数据结构称之为优先队列。如果使用普通队列来实现这个两个功能，那么寻找最大元素需要枚举整个队列，这样的时间复杂度比较高。如果已排序好的数组，那么插入一个元素则需要移动很多元素，时间复杂度依旧很高。而堆就是一种优先队列的实现，可以很好的解决这两种操作。

另外Dijkstra算法中每次找离源点最近的一个顶点也可以用堆来优化，使算法的时间复杂度降到 $O((M+N)\log N)$ 。堆还经常被用来求一个数列中第K大的数。只需要建立一个大小为K的最小堆，堆顶就是第K大的数。如果求一个数列中第K小的数，只需要建立一个大小为K的最大堆，堆顶就是第K小的数，这种方法的时间复杂度是 $O(N\log K)$ 。当然你也可以用堆来求前K大的数和前K小的数。你还能想出更快的算法吗？有兴趣的同学可以去阅读《编程之美》第二章第五节。

堆排序算法是由J.W.J. Williams在1964年发明，他同时描述了如何使用堆来实现一个优先队列。同年，由Robert W. Floyd提出了建立堆的线性时间算法。

BTW，《啊哈！算法》系列，坐在马桶上都能读懂的算法入门书，已经整理出版，本次更新是最后一次在线更新啦。各位喜欢《啊哈！算法》的朋友要去买一本收藏哦😏 这年头写个东西不容易，多谢大家支持啦😁，当当网购买链接 <http://product.dangdang.com/23490849.html>

买了的朋友记得来啥单，还可以得到《啊哈！算法》的T恤哦~~~ <http://www.ahalei.com/thread-4969-1-1.html>



分类: [啊哈！算法](#)

标签: [堆的C语言实现](#), [数据结构堆](#)

绿色通道：

好文要顶

已关注

收藏该文

与我联系



啊哈磊

关注 - 0

粉丝 - 181

我在关注他 [取消关注](#)

1

推荐

1

反对

(请您对文章做出评价)

B~树

1.前言：

动态查找树主要有：二叉查找树 (Binary Search Tree)，平衡二叉查找树 (Balanced Binary Search Tree)，红黑树 (Red-Black Tree)，B-tree/B⁺-tree/ B^{*}-tree (B~Tree)。前三者是典型的二叉查找树结构，其查找的时间复杂度 $O(\log_2 N)$ 与树的深度相关，那么降低树的深度自然对查找效率是有所提高的；还有一个实际问题：就是大规模数据存储中，实现索引查询这样一个实际背景下，树节点存储的元素数量是有限的（如果元素数量非常多的话，查找就退化成节点内部的线性查找了），这样导致二叉查找树结构由于树的深度过大而造成磁盘I/O读写过于频繁，进而导致查询效率低下（为什么会出现这种情况，待会在外部存储器-磁盘中有所解释），那么如何减少树的深度（当然是不能减少查询的数据量），一个基本的想法就是：采用多叉树结构（由于树节点元素数量是有限的，自然该节点的子树数量也就是有限的）。

这样我们就提出了一个新的查找树结构——多路查找树。根据平衡二叉树的启发，自然就想到平衡多路查找树结构，也就是这篇文章所要阐述的主题B~tree(B树结构)，B-tree这棵神奇的树是在Rudolf Bayer, Edward M. McCreight(1970)写的一篇论文《Organization and Maintenance of Large Ordered Indices》中首次提出。具体介绍可以参考wikipedia中的介绍：<http://en.wikipedia.org/wiki/B-tree>，其中还阐述了B-tree名字来源以及相关的开源地址。

在开始介绍B~tree之前，先了解下相关的硬件知识，才能很好的了解为什么需要B~tree这种外存数据结构。

2.外存储器—磁盘

计算机存储设备一般分为两种：内存储器(main memory)和外存储器(external memory)。内存存取速度快，但容量小，价格昂贵，而且不能长期保存数据(在不通电情况下数据会消失)。

外存储器—磁盘是一种直接存取的存储设备(DASD)。它是以存取时间变化不大为特征的。可以直接存取任何字符组，且容量大、速度较其它外存设备更快。

2.1磁盘的构造

磁盘是一个扁平的圆盘(与电唱机的唱片类似)。盘面上有许多称为磁道的圆圈，数据就记录在这些磁道上。磁盘可以是单片的，也可以是由若干盘片组成的盘组，每一盘片上有两个面。如下图6片盘组为例，除去最顶端和最底端的外侧面不存储数据之外，一共有10个面可以用来保存信息。

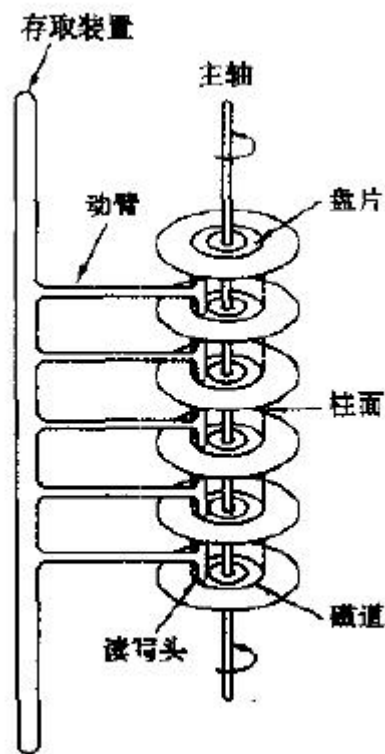


图 11.3 活动头盘示意图

当磁盘驱动器执行读/写功能时。盘片装在一个主轴上，并绕主轴高速旋转，当磁道在读/写头(又叫磁头)下通过时，就可以进行数据的读/写了。

一般磁盘分为固定头盘(磁头固定)和活动头盘。固定头盘的每一个磁道上都有独立的磁头，它是固定不动的，专门负责这一磁道上数据的读/写。

活动头盘(如上图)的磁头是可移动的。每一个盘面上只有一个磁头(磁头是双向的，因此正反盘面都能读写)。它可以从该面的一个磁道移动到另一个磁道。所有磁头都装在同一个动臂上，因此不同盘面上的所有磁头都是同时移动的(行动整齐划一)。当盘片绕主轴旋转的时候，磁头与旋转的盘片形成一个圆柱体。各个盘面上半径相同的磁道组成了一个圆柱面，我们称为柱面。因此，柱面的个数也就是盘面上的磁道数。

2.2磁盘的读/写原理和效率

磁盘上数据必须用一个三维地址唯一标示：柱面号、盘面号、块号(磁道上的盘块)。

读/写磁盘上某一指定数据需要下面3个步骤：

- (1) 首先移动臂根据柱面号使磁头移动到所需要的柱面上，这一过程被称为定位或查找。
- (2) 如上图6盘组示意图中，所有磁头都定位到了10个盘面的10条磁道上(磁头都是双向的)。这时根据盘面号来确定指定盘面上的磁道。
- (3) 盘面确定以后，盘片开始旋转，将指定块号的磁道段移动至磁头下。

经过上面三个步骤，指定数据的存储位置就被找到。这时就可以开始读/写操作了。

访问某一具体信息，由3部分时间组成：

- 查找时间(seek time) Ts: 完成上述步骤(1)所需要的时间。这部分时间代价最高，最大可达到0.1s左右。
- 等待时间(latency time) Tl: 完成上述步骤(3)所需要的时间。由于盘片绕主轴旋转速度很快，一般为7200转/分(电脑硬盘的性能指标之一，家用的普通硬盘的转速一般有5400rpm(笔记本)、7200rpm几种)。因此一般旋转一圈大约0.0083s。
- 传输时间(transmission time) Tt: 数据通过系统总线传送到内存的时间，一般传输一个字节(byte)大概 $0.02\mu s = 2 \times 10^{-8}s$

磁盘读取数据是以盘块(block)为基本单位的。位于同一盘块中的所有数据都能被一次性全部读取出来。而磁盘IO代价主要花费在查找时间Ts上。因此我们应该尽量将相关信息存放在同一盘块，同一磁道中。或者至少放在同一柱面或相邻柱面上，以求在读/写信息时尽量减少磁头来回移动的次数，避免过多的查找时间Ts。

所以，在大规模数据存储方面，大量数据存储在外存磁盘中，而在外存磁盘中读取/写入块(block)中某数据时，首先需要定位到磁盘中的某块，如何有效地查找磁盘中的数据，需要一种合理高效的外存数据结构，就是下面所要重点阐述的B-tree结构，以及相关的变种结构：**B⁺-tree结构和B^{*}-tree结构**。

3.B-tree

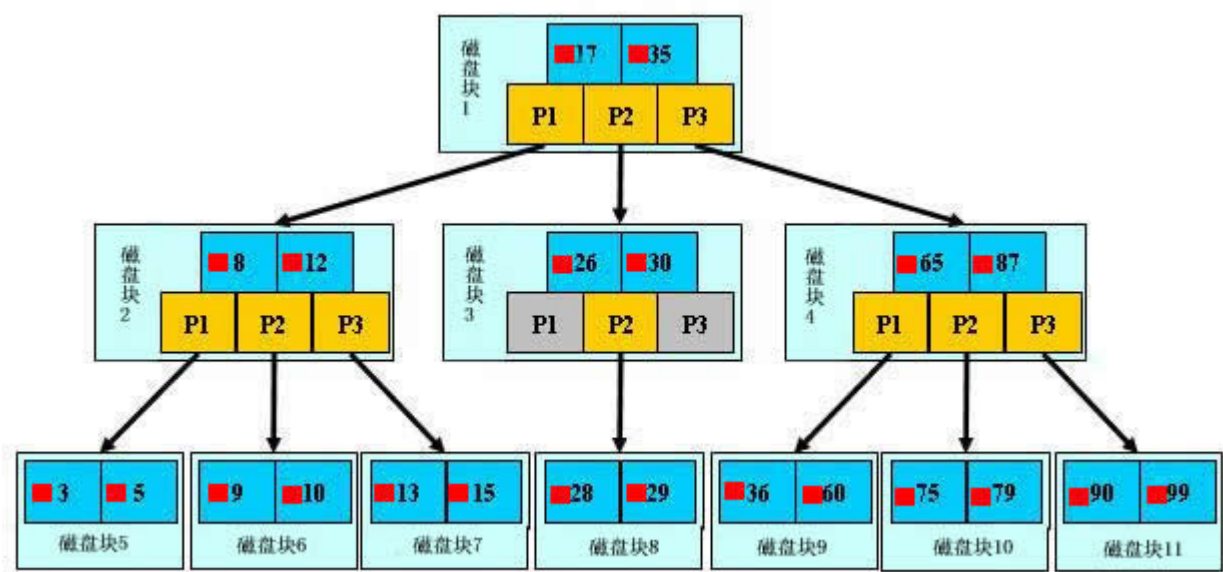
B-tree又叫平衡多路查找树。一棵m阶的B-tree (m叉树)的特性如下：

(其中 $\text{ceil}(x)$ 是一个取上限的函数)

- 1) 树中每个结点至多有m个孩子；
- 2) 除根结点和叶子结点外，其它每个结点至少有 $\text{ceil}(m / 2)$ 个孩子；
- 3) 若根结点不是叶子结点，则至少有2个孩子（特殊情况：没有孩子的根结点，即根结点为叶子结点，整棵树只有一个根节点）；
- 4) 所有叶子结点都出现在同一层，叶子结点不包含任何关键字信息(可以看做是外部结点或查询失败的结点，实际上这些结点不存在，指向这些结点的指针都为null)；
- 5) 每个非终端结点中包含有n个关键字信息： $(n, P_0, K_1, P_1, K_2, P_2, \dots, K_n, P_n)$ 。其中：
 - a) $K_i (i=1 \dots n)$ 为关键字，且关键字按顺序排序 $K_{i-1} < K_i$ 。
 - b) P_i 为指向子树根的接点，且指针 P_{i-1} 指向子树种所有结点的关键字均小于 K_i ，但都大于 K_{i-1} 。
 - c) 关键字的个数n必须满足： $\text{ceil}(m / 2) - 1 \leq n \leq m - 1$ 。

B-tree中的每个结点根据实际情况可以包含大量的关键字信息和分支(当然是不能超过磁盘块的大小，根据磁盘驱动(disk drives)的不同，一般块的大小在1k~4k左右)；这样树的深度降低了，这就意味着查找一个元素只要很少结

点从外存磁盘中读入内存，很快访问到要查找的数据。



为了简单，这里用少量数据构造一棵3叉树的形式。上面的图中比如根结点，其中17表示一个磁盘文件的文件名；小红方块表示这个17文件的内容在硬盘中的存储位置；p1表示指向17左子树的指针。

其结构可以简单定义为：

```
typedef struct {  
  
    /*文件数*/  
  
    int file_num;  
  
    /*文件名(key)*/  
  
    char * file_name[max_file_num];  
  
    /*指向子节点的指针*/  
  
    BTreeNode * BTptr[max_file_num+1];  
  
    /*文件在硬盘中的存储位置*/  
  
    FILE_HARD_ADDR offset[max_file_num];  
  
}BTreeNode;
```

假如每个盘块可以正好存放一个B-tree的结点（正好存放2个文件名）。那么一个BTreeNode结点就代表一个盘块，而子树指针就是存放另外一个盘块的地址。

模拟查找文件29的过程：

(1) 根据根结点指针找到文件目录的根磁盘块1，将其中的信息导入内存。【磁盘IO操作1次】

(2) 此时内存中有两个文件名17，35和三个存储其他磁盘页面地址的数据。根据算法我们发现 $17 < 29 < 35$ ，因此我们找到指针p2。

(3) 根据p2指针，我们定位到磁盘块3，并将其中的信息导入内存。【磁盘IO操作2次】

(4) 此时内存中有两个文件名26，30和三个存储其他磁盘页面地址的数据。根据算法我们发现 $26 < 29 < 30$ ，因此我们找到指针p2。

(5) 根据p2指针，我们定位到磁盘块8，并将其中的信息导入内存。【磁盘IO操作3次】

(6) 此时内存中有两个文件名28，29。根据算法我们查找到文件29，并定位了该文件内存的磁盘地址。

分析上面的过程，发现需要3次磁盘IO操作和3次内存查找操作。关于内存中的文件名查找，由于是一个有序表结构，可以利用折半查找提高效率。至于3次磁盘IO操作时影响整个B-tree查找效率的决定因素。

当然，如果我们使用平衡二叉树的磁盘存储结构来进行查找，磁盘IO操作最少4次，最多5次。而且文件越多，B-tree比平衡二叉树所用的磁盘IO操作次数将越少，效率也越高。

上面仅仅介绍了对于B-tree这种结构的查找过程，还有树节点的插入与删除过程，以及相关的算法和代码的实现，将在以后的深入学习中给出相应的实例。

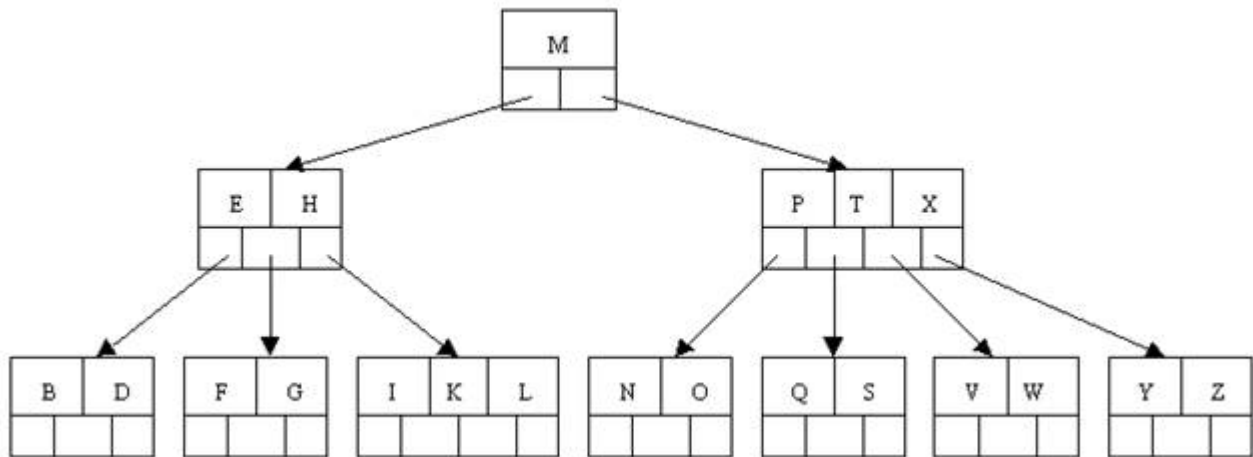
上面简单介绍了利用B-tree这种结构如何访问外存磁盘中的数据的情况，下面咱们通过另外一个实例来对这棵B-tree的插入（insert），删除（delete）基本操作进行详细的介绍：

下面以一棵5阶B-tree实例进行讲解(如下图所示)：

其满足上述条件：除根结点和叶子结点外，其它每个结点至少有 $\text{ceil}(5/2)=3$ 个孩子（至少2个关键字）；当然最多5个孩子（最多4个关键字）。下图中关键字为大写字母，顺序为字母升序。

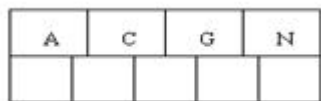
结点定义如下：

```
typedef struct{  
  
    int Count;        // 当前节点中关键元素数目  
  
    ItemType Key[4];  // 存储关键字元素的数组  
  
    long Branch[5];   // 伪指针数组，(记录数目)方便判断合并和分裂的情况  
  
} NodeType;
```

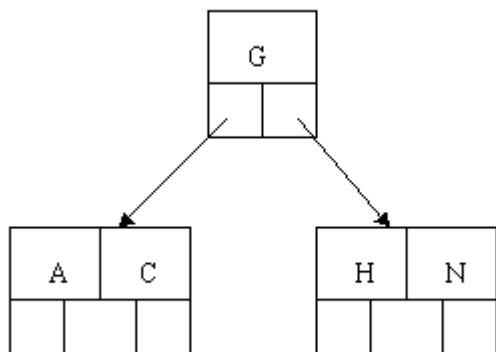



插入 (insert) 操作：插入一个元素时，首先在B-tree中是否存在，如果不存在，即在叶子结点处结束，然后在叶子结点中插入该新的元素，注意：如果叶子结点空间足够，这里需要向右移动该叶子结点中大于新插入关键字的元素，如果空间满了以致没有足够的空间去添加新的元素，则将该结点进行“分裂”，将一半数量的关键字元素分裂到新的其相邻右结点中，中间关键字元素上移到父结点中（当然，如果父结点空间满了，也同样需要“分裂”操作），而且当结点中关键元素向右移动了，相关的指针也需要向右移。如果在根结点插入新元素，空间满了，则进行分裂操作，这样原来的根结点中的中间关键字元素向上移动到新的根结点中，因此导致树的高度增加一层。

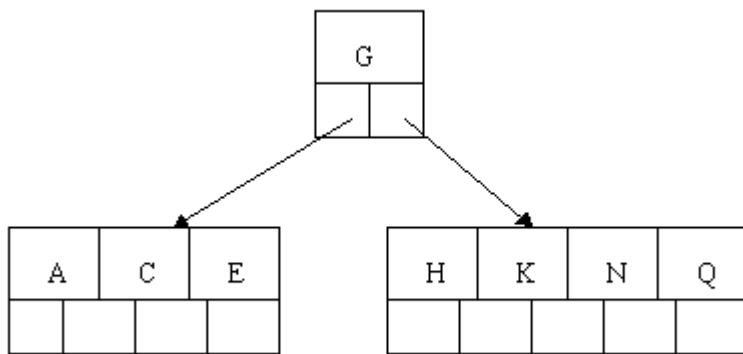
咱们通过一个实例来逐步讲解下。插入以下字符字母到空的5阶B-tree中：C N G A H E K Q M F W L T Z D P R X Y S，5序意味着一个结点最多有5个孩子和4个关键字，除根结点外其他结点至少有2个关键字，首先，结点空间足够，4个字母插入相同的结点中，如下图：



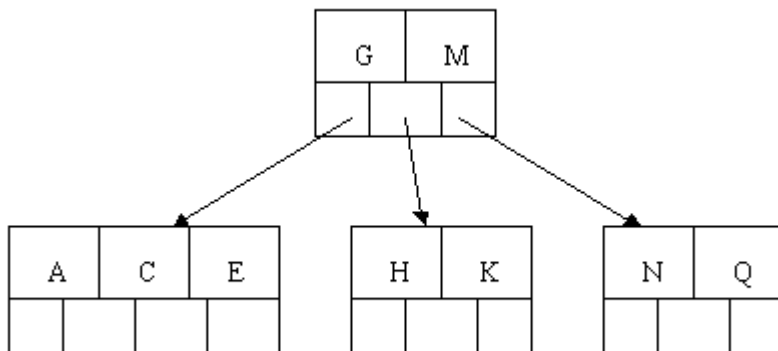
当咱们试着插入H时，结点发现空间不够，以致将其分裂成2个结点，移动中间元素G上移到新的根结点中，在实现过程中，咱们把A和C留在当前结点中，而H和N放置新的其右邻居结点中。如下图：



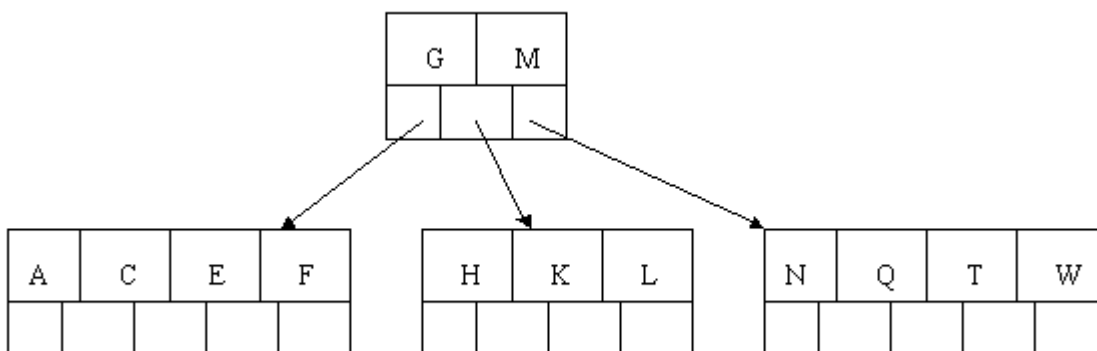
当咱们插入E,K,Q时，不需要任何分裂操作



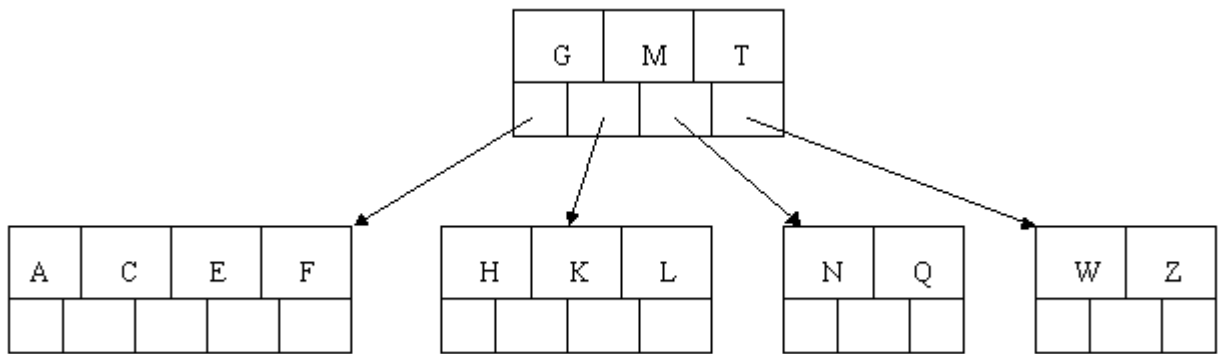
插入M需要一次分裂，注意M恰好是中间关键字元素，以致向上移到父节点中



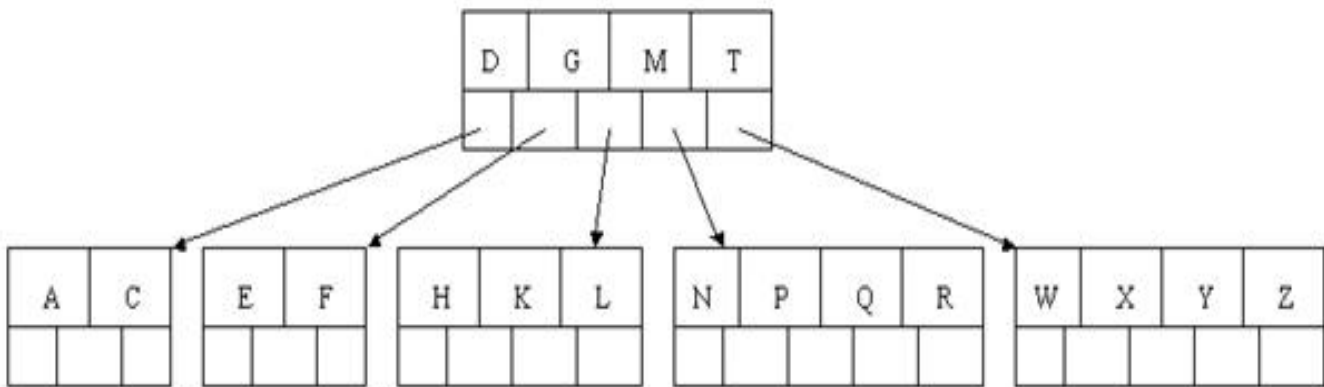
插入F,W,L,T不需要任何分裂操作



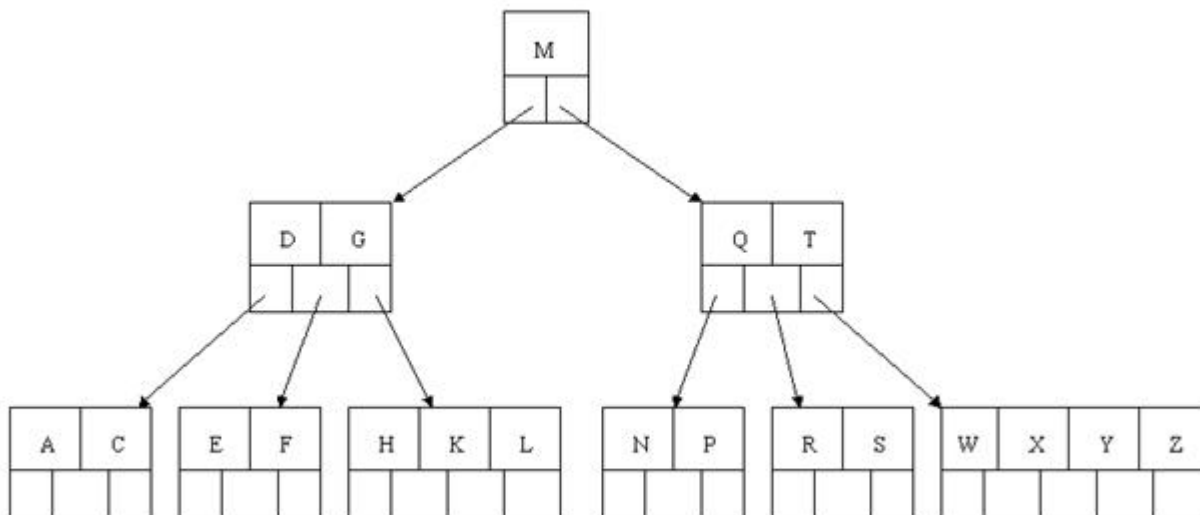
插入Z时，最右的叶子结点空间满了，需要进行分裂操作，中间元素T上移到父节点中，注意通过上移中间元素，树最终还是保持平衡，分裂结果的结点存在2个关键字元素。



插入D时，导致最左边的叶子结点被分裂，D恰好也是中间元素，上移到父节点中，然后字母P,R,X,Y陆续插入不需要任何分裂操作。



最后，当插入S时，含有N,P,Q,R的结点需要分裂，把中间元素Q上移到父节点中，但是情况来了，父节点中空间已经满了，所以也要进行分裂，将父节点中的中间元素M上移到新形成的根结点中，注意以前在父节点中的第三个指针在修改后包括D和G节点中。这样具体插入操作的完成，下面介绍删除操作，删除操作相对于插入操作要考虑的情况多点。

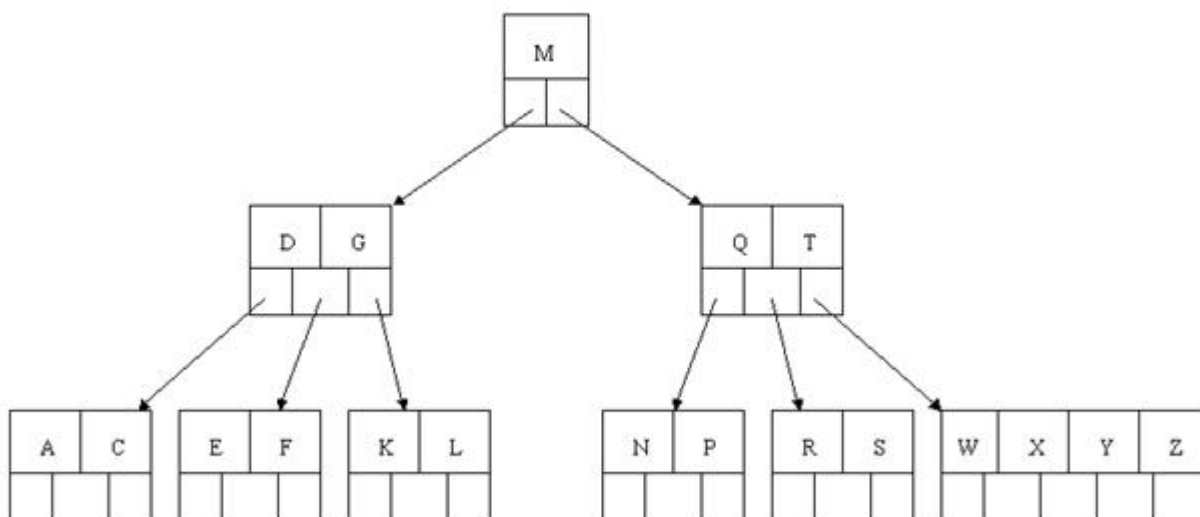


删除(delete)操作：首先查找B-tree中需删除的元素,如果该元素在B-tree中存在，则将该元素在其结点中进行删除，如果删除该元素后，首先判断该元素是否有左右孩子结点，如果有，则上移孩子结点中的某相近元素到父节点中，然后是移动之后的情况；如果没有，直接删除后，移动之后的情况。

删除元素，移动相应元素之后，如果某结点中元素数目小于 $\text{ceil}(m/2)-1$ ，则需要看其某相邻兄弟结点是否丰满（结点中元素个数大于 $\text{ceil}(m/2)-1$ ），如果丰满，则向父节点借一个元素来满足条件；如果其相邻兄弟都刚脱贫，即借了之后其结点数目小于 $\text{ceil}(m/2)-1$ ，则该结点与其相邻的某一兄弟结点进行“合并”成一个结点，以此来满足条件。那咱们通过下面实例来详细了解吧。

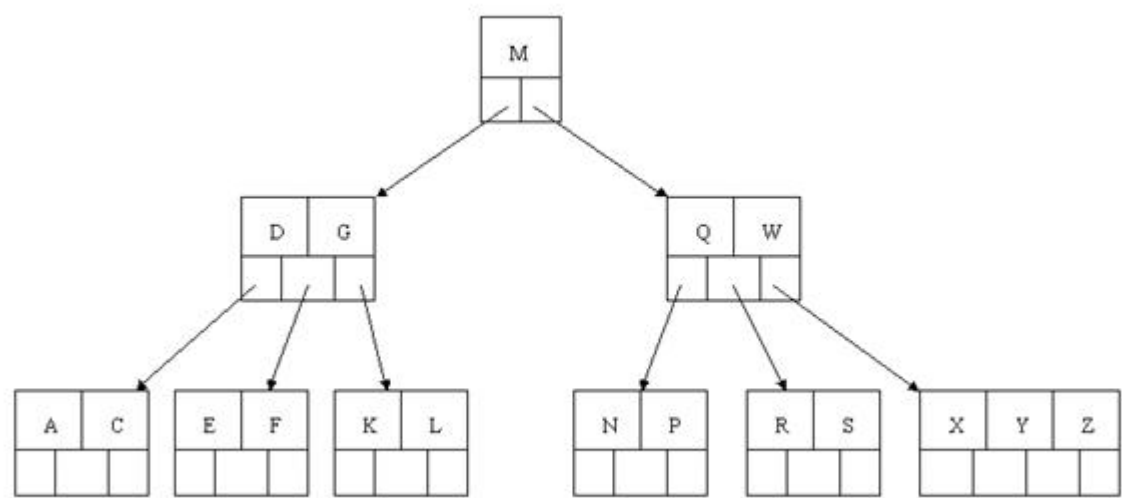
以上述插入操作构造的一棵5阶B-tree为例，依次删除H,T,R,E。

首先删除元素H，当然首先查找H，H在一个叶子结点中，且该叶子结点元素数目3大于最小元素数目 $\text{ceil}(m/2)-1=2$ ，则操作很简单，咱们只需要移动K至原来H的位置，移动L至K的位置（也就是结点中删除元素后面的元素向前移动）

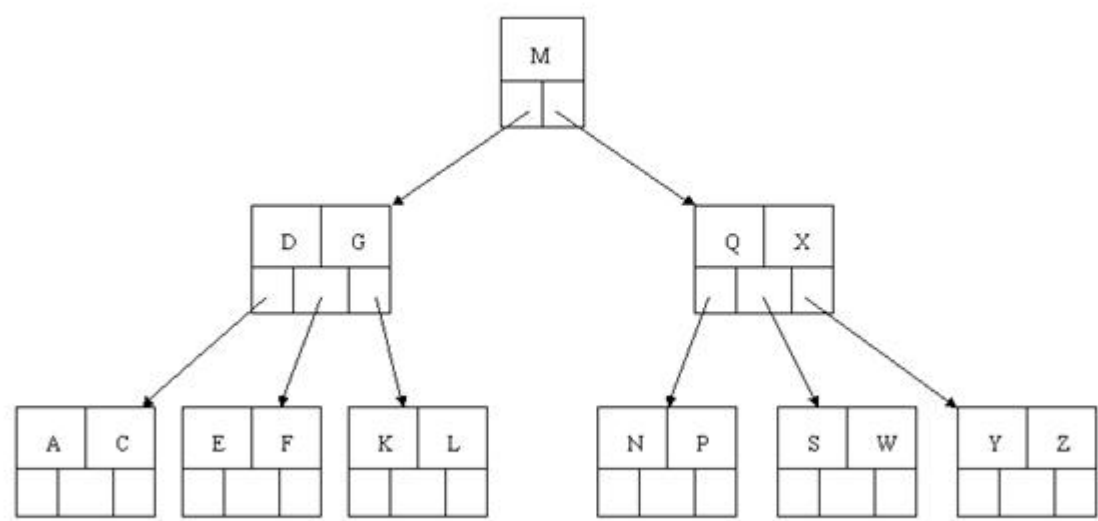


下一步，删除T,因为T没有在叶子结点中，而是在中间结点中找到，咱们发现他的继承者W(字母升序的下个元素)，

将W上移到T的位置，然后将原包含W的孩子结点中的W进行删除，这里恰好删除W后，该孩子结点中元素个数大于2，无需进行合并操作。

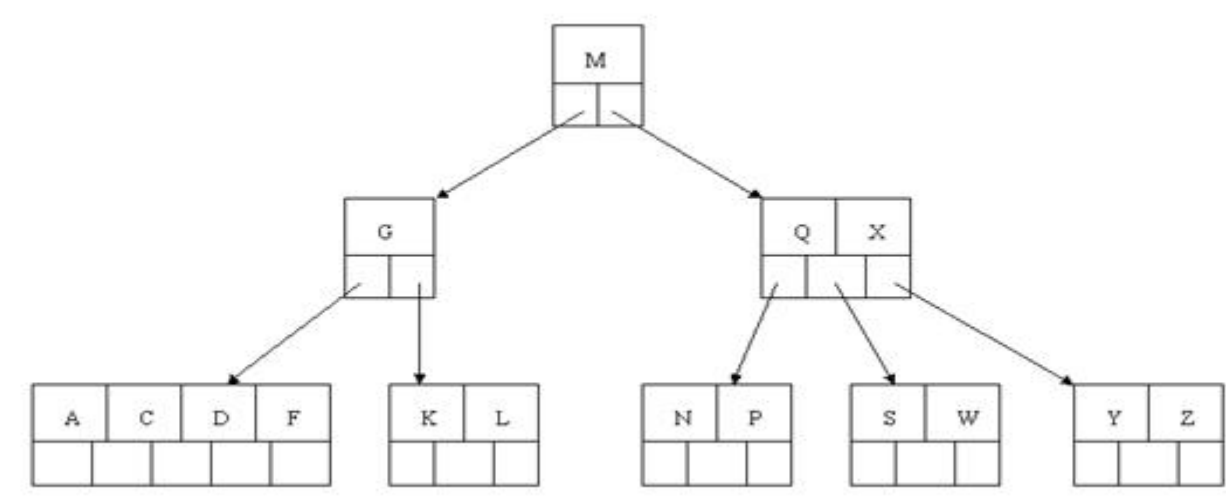


下一步删除R，R在叶子结点中,但是该结点中元素数目为2，删除导致只有1个元素，已经小于最小元素数目 $\text{ceil}(5/2)-1=2$,如果其某个相邻兄弟结点中比较丰满（元素个数大于 $\text{ceil}(5/2)-1=2$ ），则可以向父结点借一个元素，然后将最丰满的相邻兄弟结点中上移最后或最前一个元素到父节点中，在这个实例中，右相邻兄弟结点中比较丰满（3个元素大于2），所以先向父节点借一个元素W下移到该叶子结点中，代替原来S的位置，S前移；然后X在相邻右兄弟结点中上移到父结点中，最后在相邻右兄弟结点中删除X，后面元素前移。

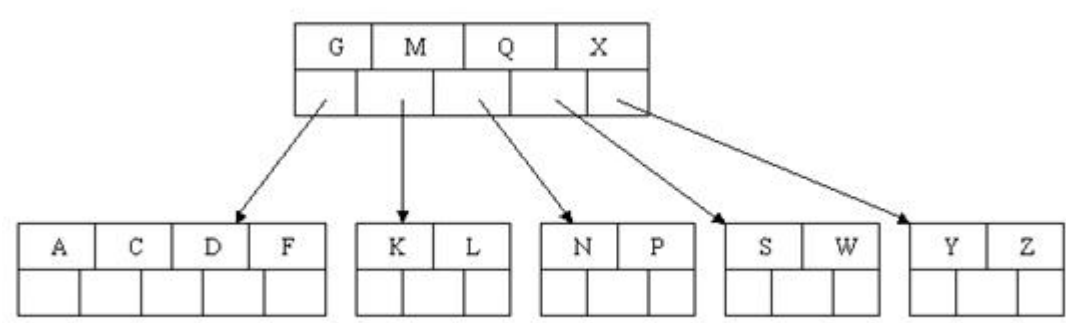


最后一步删除E，删除后会导致很多问题，因为E所在的结点数目刚好达标，刚好满足最小元素个数（ $\text{ceil}(5/2)-1=2$ ），而相邻的兄弟结点也是同样的情况，删除一个元素都不能满足条件，所以需要该节点与某相邻兄弟结点进行合并操作；首先移动父结点中的元素（该元素在两个需要合并的两个结点元素之间）下移到其子结点

中，然后将这两个结点进行合并成一个结点。所以在该实例中，咱们首先将父节点中的元素D下移到已经删除E而只有F的结点中，然后将含有D和F的结点和含有A,C的相邻兄弟结点进行合并成一个结点。

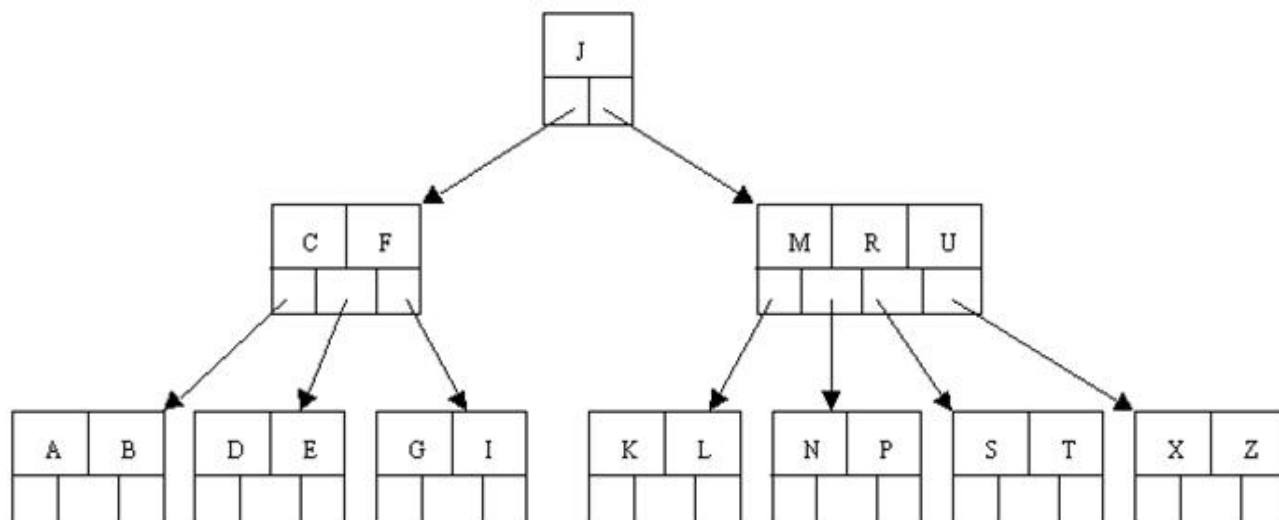


也许你认为这样删除操作已经结束了，其实不然，在看看上图，对于这种情况，你立即会发现父节点只包含一个元素G，没达标，这是不能够接受的。如果这个问题结点的相邻兄弟比较丰满，则可以向父结点借一个元素。假设这时右兄弟结点（含有Q,X）有一个以上的元素（Q右边还有元素），然后咱们将M下移到元素很少的子结点中，将Q上移到M的位置，这时，Q的左子树将变成M的右子树，也就是含有N，P结点被依附在M的右指针上。所以在这个实例中，咱们没有办法去借一个元素，只能与兄弟结点进行合并成一个结点，而根结点中的唯一元素M下移到子结点，这样，树的高度减少一层。

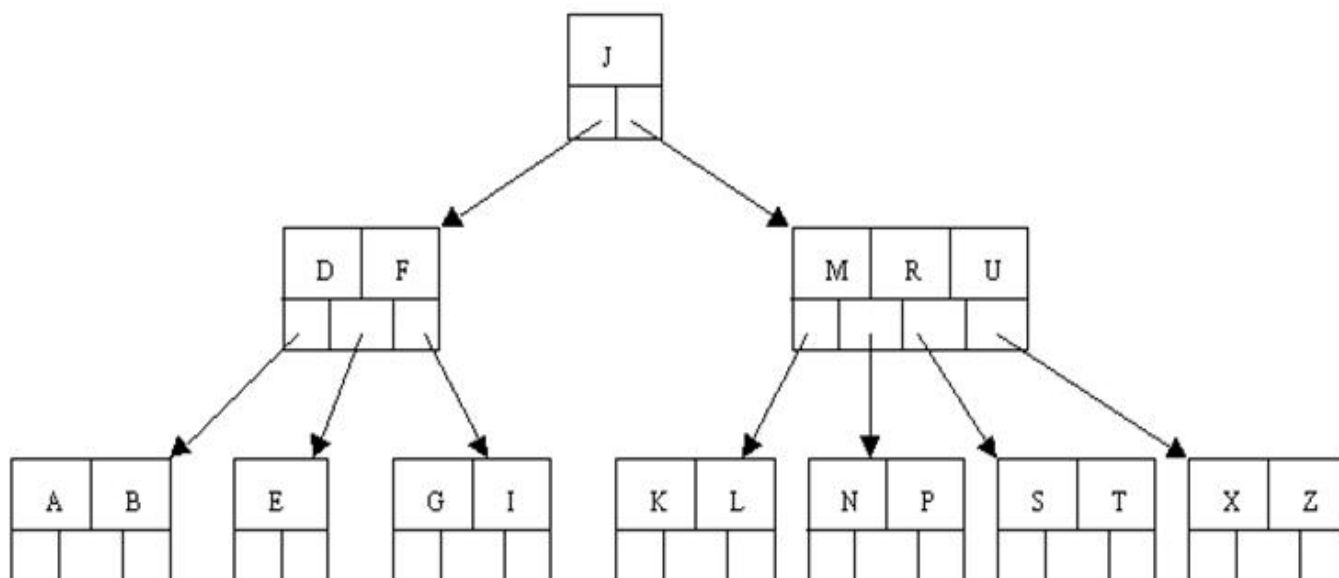


为了进一步详细讨论删除的情况。再举另外一个实例：

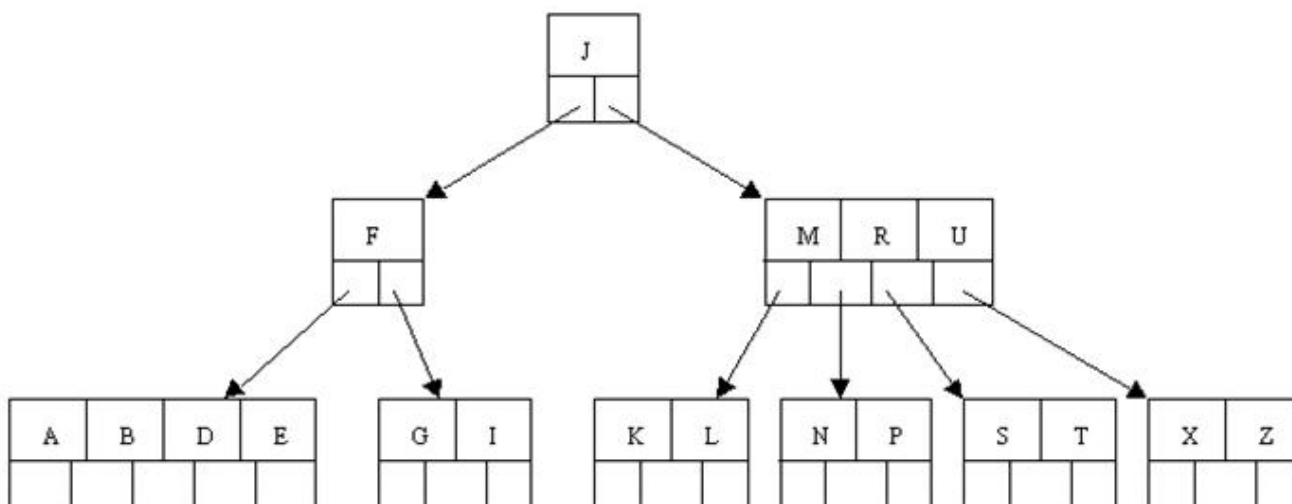
这里是一棵不同的5阶B-tree，那咱们试着删除C



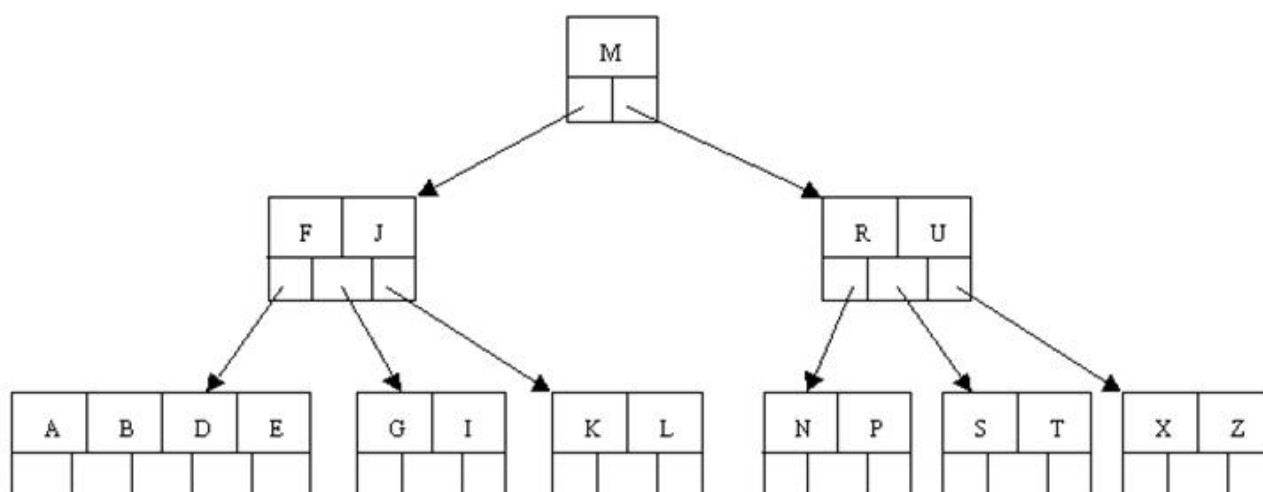
于是将删除元素C的右子结点中的D元素上移到C的位置，但是出现上移元素后，只有一个元素的结点的情况。



又因为含有E的结点，其相邻兄弟结点才刚脱贫（最少元素个数为2），不可能向父节点借元素，所以只能进行合并操作，于是这里将含有A,B的左兄弟结点和含有E的结点进行合并成一个结点。



这样又出现只含有一个元素F结点的情况，这时，其相邻的兄弟结点是丰满的（元素个数为3>最小元素个数2），这样就可以想父结点借元素了，把父结点中的J下移到该结点中，相应的如果结点中J后有元素则前移，然后相邻兄弟结点中的第一个元素（或者最后一个元素）上移到父节点中，后面的元素（或者前面的元素）前移（或者后移）；注意含有K，L的结点以前依附在M的左边，现在变为依附在J的右边。这样每个结点都满足B-tree结构性质。



如果想了解相关代码，见最后参考。

4.B⁺-tree

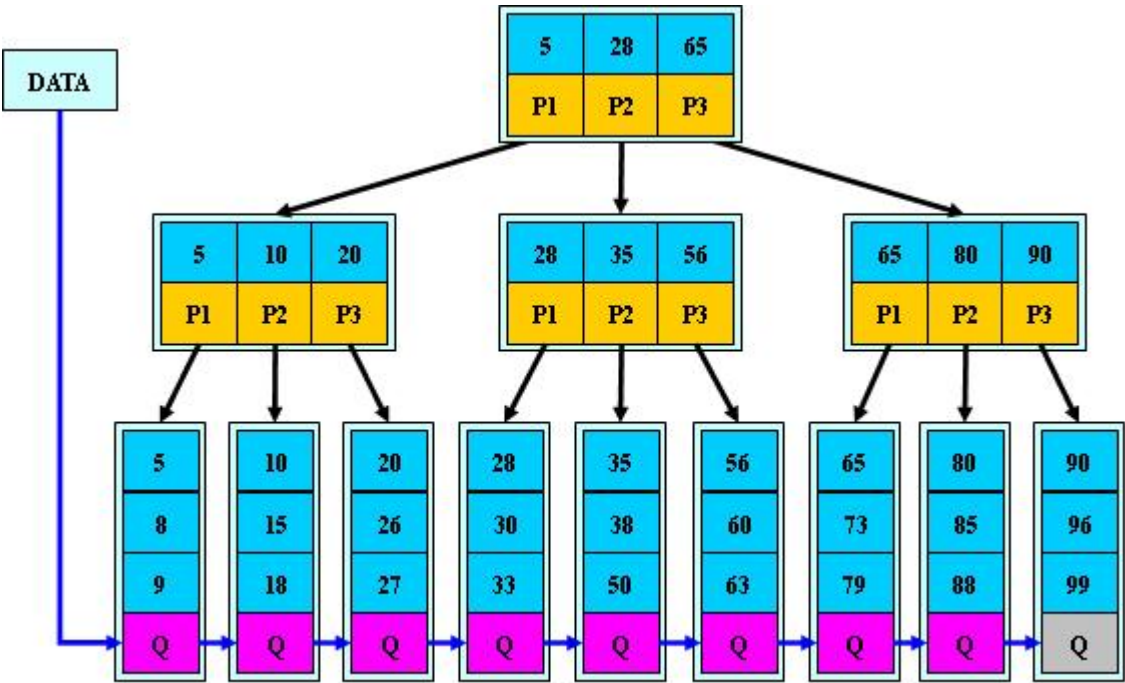
B⁺-tree：是应文件系统所需而产生的一种B-tree的变形树。

一棵m阶的**B⁺-tree**和m阶的B-tree的差异在于：

- 1.有n棵子树的结点中含有n个关键字；（B-tree是n棵子树有n-1个关键字）
- 2.所有的叶子结点中包含了全部关键字的信息，及指向含有这些关键字记录的指针，且叶子结点本身依关键字

的大小自小而大的顺序链接。(B-tree的叶子节点并没有包括全部需要查找的信息)

3.所有的非终端结点可以看成是索引部分，结点中仅含有其子树根结点中最大（或最小）关键字。(B-tree的非终端节点也包含需要查找的有效信息)



a) 为什么说B+树比B-tree更适合实际应用中操作系统的文件索引和数据库索引？

1) B⁺-tree的磁盘读写代价更低

B⁺-tree的内部结点并没有指向关键字具体信息的指针。因此其内部结点相对B-tree更小。如果把所有同一内部结点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多。一次性读入内存中的需要查找的关键字也就越多。相对来说IO读写次数也就降低了。

举个例子，假设磁盘中的一个盘块容纳16bytes，而一个关键字2bytes，一个关键字具体信息指针2bytes。一棵9阶B-tree(一个结点最多8个关键字)的内部结点需要2个盘快。而B⁺-tree内部结点只需要1个盘快。当需要把内部结点读入内存中的时候，B-tree就比B⁺-tree多一次盘块查找时间(在磁盘中就是盘片旋转的时间)。

2) B⁺-tree的查询效率更加稳定

由于非终结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。所以任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。

b) B⁺-tree的应用: VSAM(虚拟存储存取法)文件(来源论文the ubiquitous Btree 作者：D COMER - 1979)

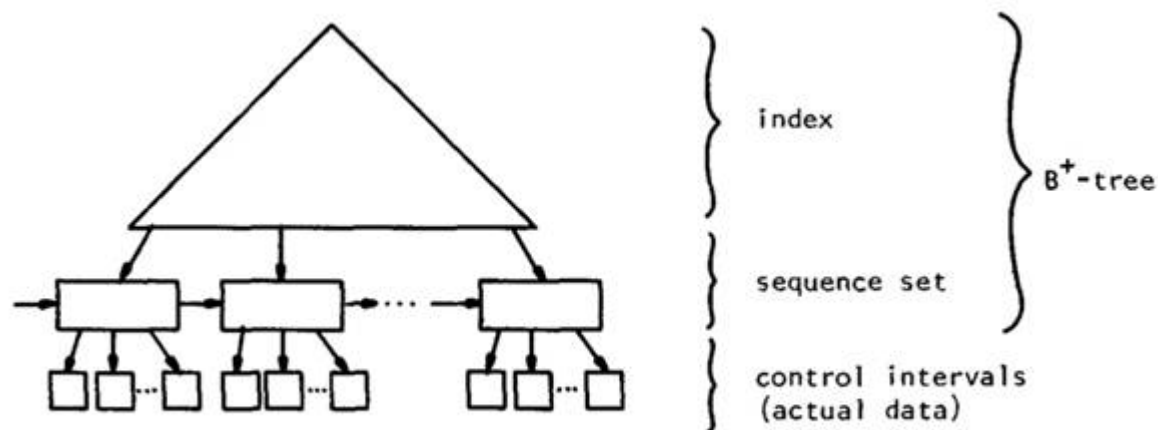
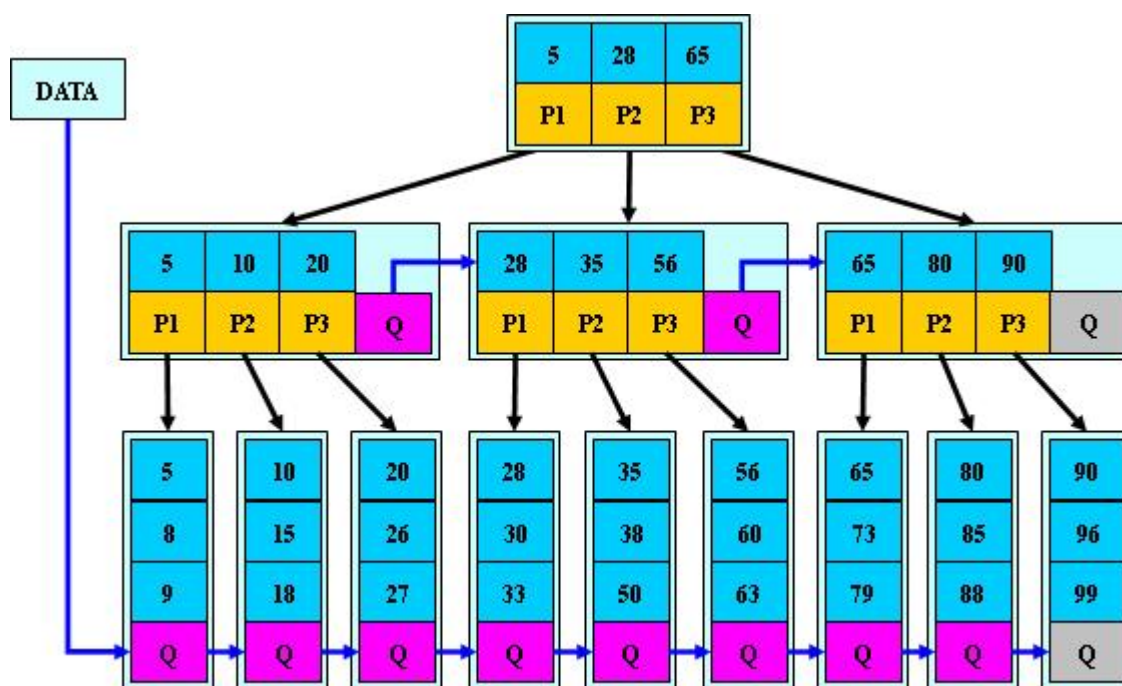


FIGURE 18. A VSAM file with actual data (associated information) stored in the leaves.

关于B⁺-tree的详细介绍将在以后的学习中给出实例，待写。。。

5.B^{*}-tree

B^{*}-tree是B⁺-tree的变体，在B⁺-tree的非根和非叶子结点再增加指向兄弟的指针；B^{*}-tree定义了非叶子结点关键字个数至少为 $(2/3)*M$ ，即块的最低使用率为 $2/3$ （代替B+树的 $1/2$ ）。给出了一个简单实例，如下图所示：



B⁺-tree的分裂：当一个结点满时，分配一个新的结点，并将原结点中 $1/2$ 的数据复制到新结点，最后在父结点中增加新结点的指针；B⁺-tree的分裂只影响原结点和父结点，而不会影响兄弟结点，所以它不需要指向兄弟的指针。

B^{*}-tree的分裂：当一个结点满时，如果它的下一个兄弟结点未满，那么将一部分数据移到兄弟结点中，再在原结点插入关键字，最后修改父结点中兄弟结点的关键字（因为兄弟结点的关键字范围改变了）；如果兄弟也满了，则在原结点与兄弟结点之间增加新结点，并各复制 $1/3$ 的数据到新结点，最后在父结点增加新结点的指针。

所以，B^{*}-tree分配新结点的概率比B⁺-tree要低，空间使用率更高；

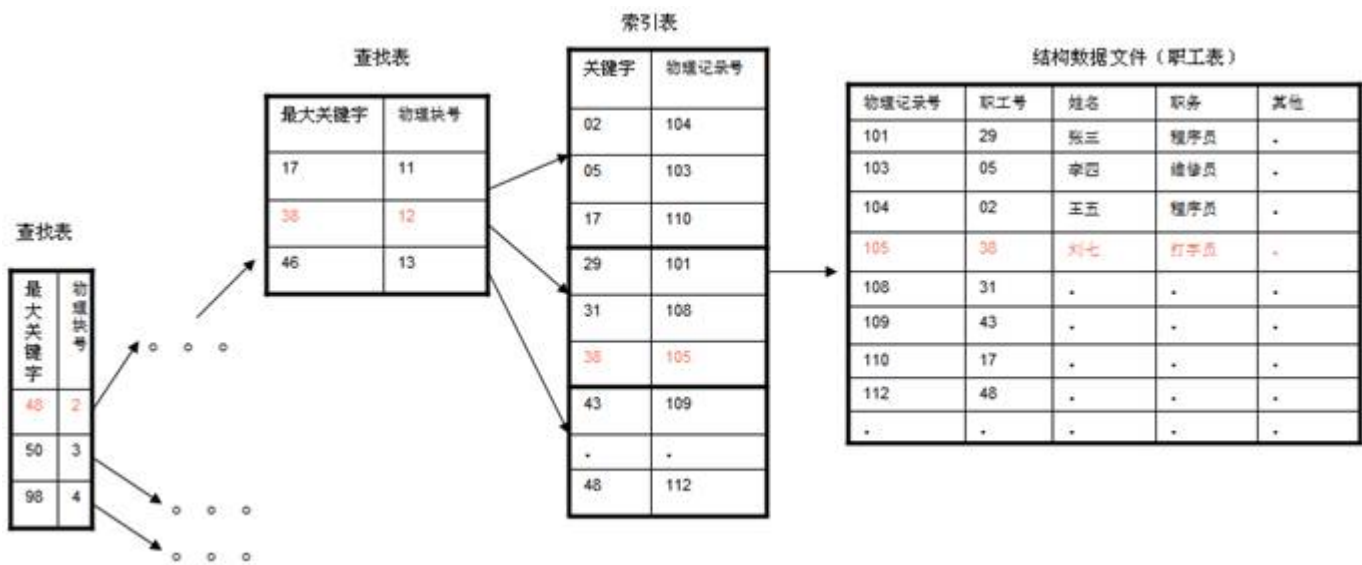
6.总结

B-tree , B⁺-tree , B^{*}-tree总结如下：

- B-tree：有序数组+平衡多叉树；
- B⁺-tree：有序数组链表+平衡多叉树；
- B^{*}-tree：一棵丰满的B⁺-tree。

在大规模数据存储的文件系统中，B~tree系列数据结构，起着很重要的作用，对于存储不同的数据，节点相关的信息也是有所不同，这里根据自己的理解，画的一个查找以职工号为关键字，职工号为38的记录简单示意图。(这里假设每个物理块容纳3个索引，磁盘的I/O操作的基本单位是块 (block),磁盘访问很费时，采用B⁺-tree有效的减少了访问磁盘的次数。)

对于像MySQL，DB2，Oracle等数据库中的索引结构有待深入的了解才行，不过网上可以找到很多B-tree相关的开源代码可以用来研究。





参考文献 (google下可以找到相关论文下载) 以及相关网址：

1. Organization and Maintenance of Large Ordered Indices
2. the ubiquitous B tree
3. <http://en.wikipedia.org/wiki/Btree> (给出了国外一些开源地址)
4. <http://cis.stvincent.edu/html/tutorials/swd/btree/btree.html> (include C++ source code)

5. <http://slady.net/java/bt/view.php> (如果了解了B-tree结构，该地址可以在线对该结构进行查找 (search) ，插入(insert)，删除(delete)操作。)



-  上一篇
- in与exist,not in 与not exist 的区别
-  下一篇
- MySQL优化之数据类型的使用

