细节——不可忽视的要素

广东北江中学 李锐喆

【关键词】细节 时间复杂度 算法

【摘要】一个再好的算法，如果在细节处理上不当，也会成为一个“垃圾算法”。但是人们往往在重视算法整个宏观实现的时候，却往往忽略了一些细节上的东西。本文通过列举细节处理优劣对算法实现的影响，以及分两个方面来阐述细节处理优劣对算法时间复杂度的影响，并举了三个比较典型的例子，旨在强调这个往往会被人忽视的细节。

【目录】

【正文】

一、引论

在程序设计中，算法是最核心的部分，往往一个优秀的算法能够取得比其他算法好的多的效率，但是在追求更好的算法的时候，我们往往会忽视算法中的一些细节处理。

在很多时候，我们自认为用到了最优的算法，但效果往往不尽如人意，为什么呢？细节！往往是细节上的一些瑕疵，导致算法的关键地方时间效率低下，甚至导致算法的时间复杂度远远高于正常的情况，最为严重的后果是导致整个算法的错误。

所以，我们说，细节在程序设计中是相当重要的。借用哲学原理来说，细节是算法这个整体的关键部分，而这个关键部分往往会对算法这个整体的性能状态起决定作用。按照对算法的影响的性质和程度，我把细节分为这几种情况：

1. **基本不影响算法的时间复杂度的细节处理。**这类细节处理对时间复杂度没有根本性的影响，仅仅对时间复杂度的系数产生影响。
2. **影响到算法的时间复杂度的细节处理。**这类细节相当隐蔽，往往不为人所注意。但是这种细节对算法的影响相当大，处理得好与处理得不好往往会使程序的时间效率有质的区别。
3. **影响到算法正确或错误的细节处理。**这类细节影响最大，在竞赛中很多选手在某些题目中已经找到解决方法却不能通过全部测试数据，往往就是这类的细节处理得不当导致。

第三种情况大家都肯定感受颇深，要讲的话也必然是长篇大论，这里就不再赘述了。下面，我们主要对前两种情况分别进行分析讨论。

二、基本不影响算法时间复杂度的细节处理

这类细节对算法影响不算太大，但往往在关键的时候，时间复杂度的系数的大小对算法的效率也是有比较明显的影响的。例如对于某个细节，用方法 A 来处理的时间复杂度为 $O(n)$，而方法 B 来处理的时间复杂度为 $O(2n)$（这里为了方便描述，在复杂度式中加入不需要加的系数），如果用方法 A 来处理整个算法的时间消耗为 1s，那么用方法 B 来处理整个算法的时间消耗就为 2s！因此，尽可能地优化细节处理，从而使算法的时间复杂度的系数降低到一个比较低的程度。

下面我们通过一个例子来看看对这类细节的处理：

〖例 1〗线性表维护

给出一个以字符串为数据类型的线性表，以及相应的若干个维护操作的规则，编写一个程序，模拟线性表的维护操作。

定义一个浮动指针，指向要处理的线性表元素。

定义四种对线性表的操作:

操作 1: 插入。在线性表末尾插入数据。

操作 2: 移动指针。把浮动指针向后移若干个单位。

操作 3: 删除。删除从浮动指针所指元素开始的若干个元素。

操作 4: 输出。输出浮动指针所指位置的元素。

为了便于程序实现，程序从 line.in 输入数据，并把输出结果输出到 line.out 中。输入

数据的规模不定，相应操作在输入文件中的格式如下：

1. 插入。分两行，第一行为 ADD，第二行为要插入线性表的数据。如：

   ADD
   abcdef

2. 移动指针。分两行，第一行为 MOVE，第二行为要移动的单位数。如：

   MOVE
   2

3. 删除。分两行，第一行为 DEL，第二行为要删除的元素个数。如：

   DEL

   3

4. 输出。一行 PRINT。如：

   PRINT

请编写程序完成给出的任务。

这道题目的模型很简单，由于需要动态增加、删除元素，而且空间没有限定范围，所以我们应该用动态指针来实现。相应的 Pascal 数据结构的建立也很简单：

```
PNode=^TNode;
TNode=Record
        Data:String;
        Next:PNode;
      End;
```

插入、移动、输出这些操作都不必细说，然而对于删除操作，这里是值得我们深究的。

由于操作可能频繁地插入、删除元素，所以对删除掉的元素搁置起来不进行处理是不可行的，否则很快就会让丢弃了的元素占满了内存。因此我们在删除操作时必须释放内存。

由此一个方法就形成了，每次删除元素时就把删除掉的每一个元素所占的空间释放出来，等以后需要插入元素时才重新分配使用。

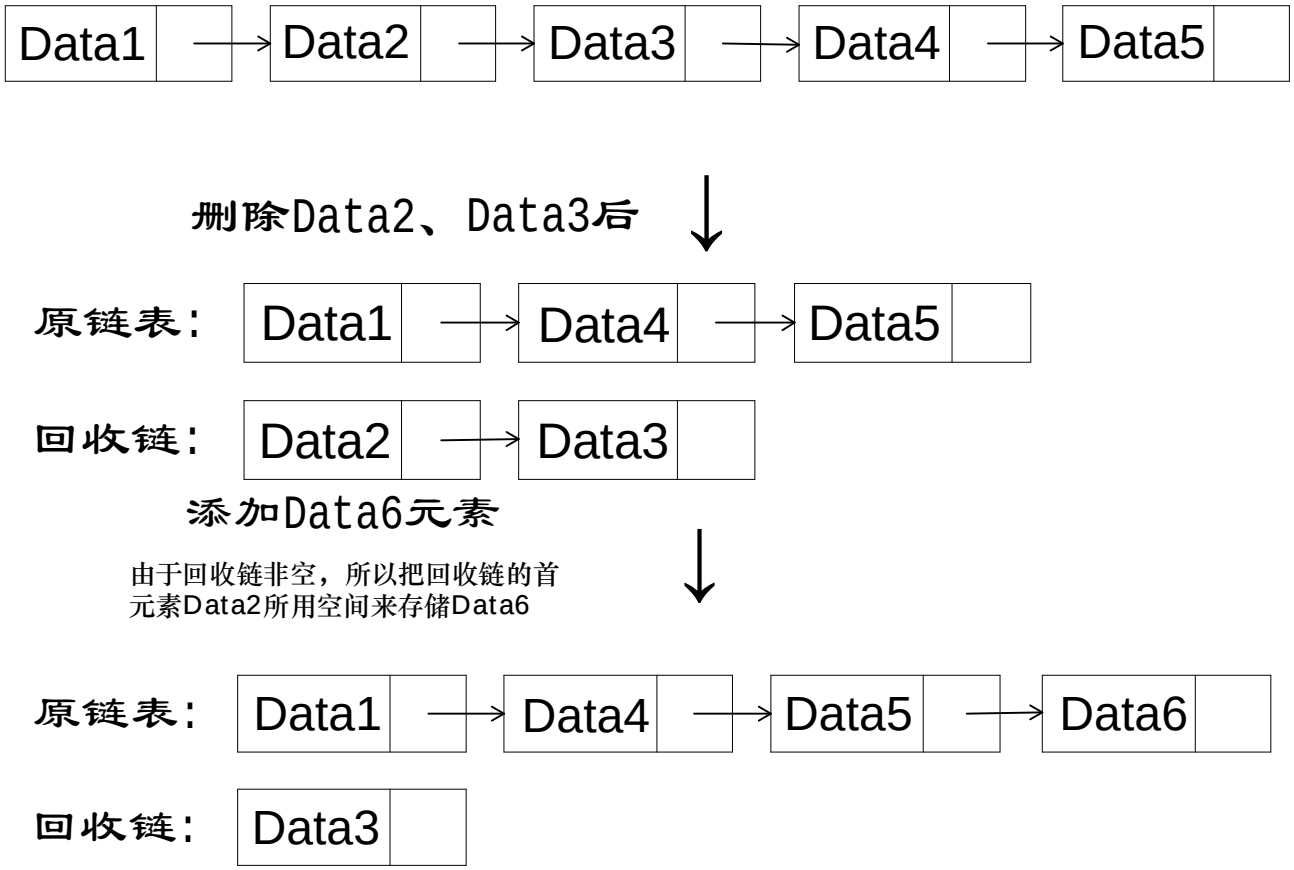
这样实现就解决了无法释放空间的问题，但是同时还存在一个问题，就是要频繁地分配和释放内存空间。释放内存空间操作的时间消耗并不大，但是分配内存空间的时间消耗却不小。

分配内存空间并不像我们写程序那样一个 new 就分配好了，实际上在系统的底层，需要进行一系列的操作。在进行空间分配的过程中，首先需要在内存中寻找一块足够大的空间来进行分配操作。由于内存空间的占用并不一定是连续的，会出现碎片的情况，因此寻找空间的时间消耗也是相当可观的。设进行内存空间分配的平均时间复杂度为 $O(h)$，那么进行 n 次内存空间分配的时间复杂度就为 $O(hn)$，远非我们想像中的 $O(n)$。

因此，我们如果要提高链表操作的效率的话，这里是一个很好的突破口。删除元素要释放内存，添加元素则要分配内存，是否可以直接利用要删除元素的空间来进行添加元素操作，而不需要分配呢？答案是肯定的。

我们把线性表称之为“原链表”，然后我们另外建一个链表，称之为“回收链”。那么

对于每一次删除操作，由于链表使用指针的灵活性，我们可以直接把要删除的部分从原链表中剥离出来后，添加到回收链的末尾，取代直接释放它们；在插入操作中，首先判断回收链是否为空，不为空就直接从回收链中取元素空间，直接添加到原链表中，如果为空，再向系统申请内存分配。具体模式如下：



经过这样优化之后，实际上需要向系统提出分配内存要求的次数就是整个维护操作中链表节点数目的最大值，而通常情况下，添加节点时是从回收链中获取空间进行分配，这样进行 n 次添加操作的时间复杂度仅仅是 $O(n)$。

经过实际的测试后，相应的测试结果如下：

测试点	原算法	优化后的算法
1	2.149s	2.012s
2	6.588s	5.672s
3	13.202s	12.924s
4	5.086s	4.911s
5	10.297s	9.918s

测试环境：AMD Duron 1.1GHz，256MB SDRAM，Debian Linux + Kernel 2.6.0
测试系统：Celiz 0.0.3

通过上面的测试结果，我们可以看出，经过优化后的算法比原算法具有一定的优势，虽然优势并不太明显，但是效率的提升也是显而易见的。

由此看来，虽然这类细节处理对算法的时间复杂度影响不算太大，但是处理得好，还是有不少益处的。

三、影响到算法时间复杂度的细节处理

相对于上面的那类细节，这种细节对算法的时间效率有相当的影响。而且我们在处理这类细节的时候往往会误入不恰当的处理方式之中，使算法的时间复杂度升高。因此，能否处理好这类细节，是算法是否能有效解决问题的关键。

下面我们来看一个例子，看看我们一个相当熟悉的算法是怎么处理好细节的。

〖例 2〗银河英雄传说（NOI2002 第一试第一题）

【问题描述】

公元五八〇一年，地球居民迁移至金牛座α第二行星，在那里发表银河联邦创立宣言，同年改元为宇宙历元年，并开始向银河系深处拓展。

宇宙历七九九年，银河系的两大军事集团在*巴米利恩星域*爆发战争。泰山压顶集团派宇宙舰队司令*莱因哈特*率领十万余艘战舰出征，气吞山河集团点名将*杨威利*组织麾下三万艘战舰迎敌。

杨威利擅长排兵布阵，巧妙运用各种战术屡次以少胜多，难免恣生骄气。在这次决战中，他将巴米利恩星域战场划分成 30000 列，每列依次编号为 1，2，…，30000。之后，他把自己的战舰也依次编号为 1，2，…，30000，让第 i 号战舰处于第 i 列(i = 1，2，…，30000)，形成“一字长蛇阵”，诱敌深入。这是初始阵形。当进犯之敌到达时，杨威利会多次发布合并指令，将大部分战舰集中在某几列上，实施密集攻击。合并指令为 M i j，含义为让第 i 号战舰所在的整个战舰队列，作为一个整体（头在前尾在后）接至第 j 号战舰所在的战舰队列的尾部。显然战舰队列是由处于同一列的一个或多个战舰组成的。合并指令的执行结果会使队列增大。

然而，老谋深算的莱因哈特早已在战略上取得了主动。在交战中，他可以通过庞大的情报网络随时监听杨威利的舰队调动指令。

在杨威利发布指令调动舰队的同时，莱因哈特为了及时了解当前杨威利的战舰分布情况，也会发出一些询问指令：C i j。该指令意思是，*询问电脑，杨威利的第 **i** 号战舰与第 **j** 号战舰当前是否在同一列中，如果在同一列中，那么它们之间布置有多少战舰。*

作为一个资深的高级程序设计员，你被要求编写程序分析杨威利的指令，以及回答莱因哈特的询问。

最终的决战已经展开，银河的历史又翻过了一页……

【输入文件】

输入文件 galaxy.in 的第一行有一个整数 T（1<=T<=500,000），表示总共有 T 条指令。
以下有 T 行，每行有一条指令。指令有两种格式:
M i j：i 和 j 是两个整数（1<=i，j<=30000），表示指令涉及的战舰编号。该指令是莱因哈特窃听到的杨威利发布的舰队调动指令，并且保证第 i 号战舰与第 j 号战舰不在同一列。
C i j：i 和 j 是两个整数（1<=i，j<=30000），表示指令涉及的战舰编号。该指令是莱因哈特发布的询问指令。

【输出文件】

输出文件为 galaxy.out。你的程序应当依次对输入的每一条指令进行分析和处理:
如果是杨威利发布的舰队调动指令，则表示舰队排列发生了变化，你的程序要注意到这一点，但是不要输出任何信息;

如果是莱因哈特发布的询问指令，你的程序要输出一行，仅包含一个整数，表示在同一列上，第 **i** 号战舰与第 **j** 号战舰之间布置的战舰数目。如果第 **i** 号战舰与第 **j** 号战舰当前不在同一列上，则输出 **-1**。

【样例输入】

```
4
M 2 3
C 1 2
M 2 4
C 4 2
```

【样例输出】

```
-1
1
```

【样例说明】

战舰位置图：表格中阿拉伯数字表示战舰编号

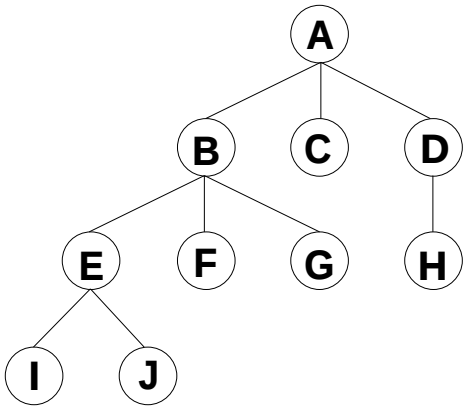
	第一列	第二列	第三列	第四列	……
初始时	1	2	3	4	……
M 2 3	1		3 2	4	……
C 1 2	1 号战舰与 2 号战舰不在同一列，因此输出 -1				
M 2 4	1			4 3 2	……
C 4 2	4 号战舰与 2 号战舰之间仅布置了一艘战舰，编号为 3，输出 1				

分析这道题目，可以把每列划分成一个集合，那么，舰队的合并、查询就是对集合的合并和查询了，这样就是一个很典型的并查集算法的模型。

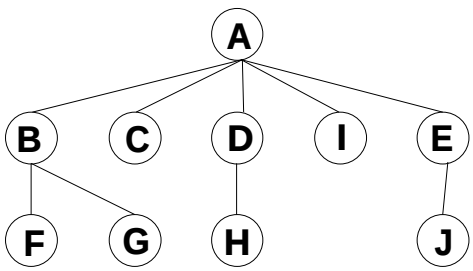
但是单纯的并查集算法还是有缺点的，合并可能使n个节点的树退化成一条链，这样将大大影响查询的效率，因此必须进行优化。并查集常用的两种优化方式是把小树合并到大树上以及路径压缩。

并查集和这两种优化的具体实现在各类书籍中提到很多，这里也不再赘述。各类教材都告诉我们，路径压缩是在查找的过程中进行的。对此我们也许会有这样的疑问，为什么要在查找中进行，而合并过程中并不进行呢？

我们来分析一下并查集合并、查询操作的时间复杂度，设 h 为需查询节点到根节点的深度，那么查找操作的时间复杂度为 $O(h)$ ，而合并操作的时间复杂度为 $O(1)$，但是实际上合并前往往要查找需合并的两个集合的根节点（例如本题），因此时间复杂度往



原并查集树



在查找I的过程中对从I到A的路径进行压缩后的并查集树

往也是$O(h)$。合并操作中并没有什么可以优化的地方，路径压缩实际上就是减少查找中 h 的大小，从而减少查找的时间消耗。

下面我们分别分析在查找过程和合并过程中进行路径压缩的复杂度。

如果我们在查找的过程中进行路径压缩，那么进行查询操作时，从需查询节点找到根节点的时间复杂度为$O(h)$，我们在找到根节点后，只要把需查询节点到根节点的路径中的所有节点的父辈节点指向根节点即可完成路径压缩，这个操作的时间复杂度也是$O(h)$，也就是说，查找+路径压缩的时间复杂度还是$O(h)$，仅仅是系数乘以了 2。合并操作的时间复杂度也是$O(h)$。而且可以证明，使用了把小树合并到大树上这种优化后，每一棵树的深度最大仅仅为$\log n$（n 为树的节点个数），那么这里的查找和合并的时间复杂度实际上就是$O(\log n)$。

如果我们在合并的过程中进行路径压缩的话，在合并过程中，被合并的树的所有节点的父辈节点都被修改为目标树的根节点，这个过程的时间复杂度变为$O(n)$，其中 n 为被合并树的节点个数。我们也很容易知道经过这样的操作后，每一棵树的深度都仅仅为 2，那么查找的复杂度变为$O(1)$了。

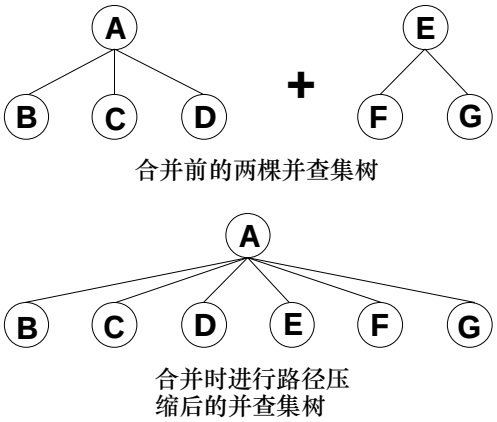


图 2：合并中进行路径压缩的模式

两种方法的时间复杂度总结为下表：

时间复杂度	查找	合并
查找时路径压缩	$O(\log n)$	$O(\log n)$
合并时路径压缩	$O(1)$	$O(n)$

由这里看起来，两种方法似乎各自有其优势的地方，在合并时路径压缩的方法，似乎在进行大量查找时有优势。但是在查找时路径压缩并非任何时间的时间复杂度都为$O(\log n)$，随着查找量的增加，路径不断地进行压缩，这个时间复杂度实际上也降了下来，而且已经证明出了这样的一个结论，n 次查找操作至多需要$O(n\alpha(n))$的时间，其中$\alpha(n)$是单变量阿克曼函数的逆，它是一个增长速度比$\log n$慢得多但又不是常数的函数。对于通常见到的正整数 n 而言$\alpha(n) \le 4$，那么查找操作的时间复杂度实际上已经接近$O(1)$了，因此在这方面查找时路径压缩对于合并时路径压缩并没有什么劣势。

而在合并方面，查找时路径压缩$O(\log n)$的时间复杂度对于合并时路径压缩$O(n)$的时间复杂度明显有较大的优势。

经过我们以上的论证，我们可以认定，查找时进行路径压缩的时间效率比合并时进行路径压缩的时间效率是要高的。

我们再来看另外一个例子，看看一个细节处理的不同带来的截然不同的效果。

〖例 3〗铁路

Byteotian 州铁道部决定赶上时代，为此他们引进了城市联网。由于缺乏高效的机器、清洁的车和笔直的轨道，他们只能建立一个这样的联网。电脑订座系统的缺乏则是另一个障碍。你的任务是编写这个系统的主要部分。

为简单起见，我们假设城市联网顺次连接着 c 个城市，从 1 到 c 编号(起始城市编号

为 1，终止城市编号为 c)。每辆火车有 s 个座位且在任何两个车站之间运送更多的乘客是不允许的。

电脑系统将收到连续的预订请求并决定是否满足他们的请求。当火车在被请求的路段上有足够的空位时，这个请求可以通过，否则不能通过。通过请求的一部分是不允许的，例如只允许一部分路线或一部分乘客。通过一个请求之后，火车里的空位数目将得到更新。请求应按照收到的顺序依次处理。

Task

写一个程序:

- 从文本文件 kol.in 读入铁路网和请求列表，
- 计算哪些请求应通过，哪些请求应被拒绝，
- 输出答案到 kol.out

Input

第一行是三个被空格隔开整数 c，s 和 r (1<=c<=60 000，1<=s<=60 000，1<=r<=60 000)。数字分别表示: 铁路上的城市个数，火车内的座位数，请求的数目。接下来 r 行是连串的请求。第 i+1 行描述第 i 个请求。描述包含三个整数 o、d 和 n (1<=o<d<=c，1<=n<=s)。它们分别表示起点车站的编号，目标车站的编号，座位的需求数。

Output

你的程序应向文件输出 r 行。第 i 应是一个字符:

- T (是) - 如果第 i 个请求可以通过，
- N (否) - 否则。

Example

对于下面的输入文件 kol.in:

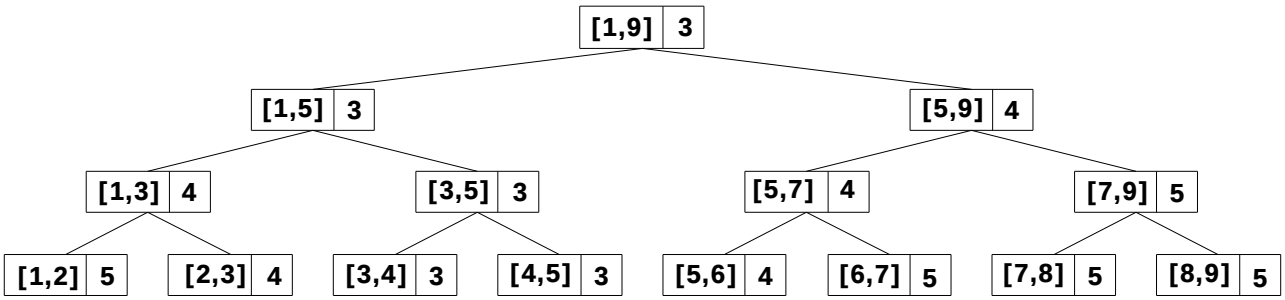
```
4 6 4
1 4 2
1 3 2
2 4 3
1 2 3
```

正确的输出文件 kol.out 应是:

```
T
T
N
N
```

分析题目，题目要求实现的是一个查询系统，系统有两种操作：查询和修改。如果用一般的线性表来实现的话，设 n 为查询中起始和目标车站之间的距离，那么每一次查询的时间复杂度是 $O(n)$，相应的每一次修改的时间复杂度也是 $O(n)$，对于本题中 n 最大可以等于 60000，而且操作限制不限，用这样的方法显然是行不通的。

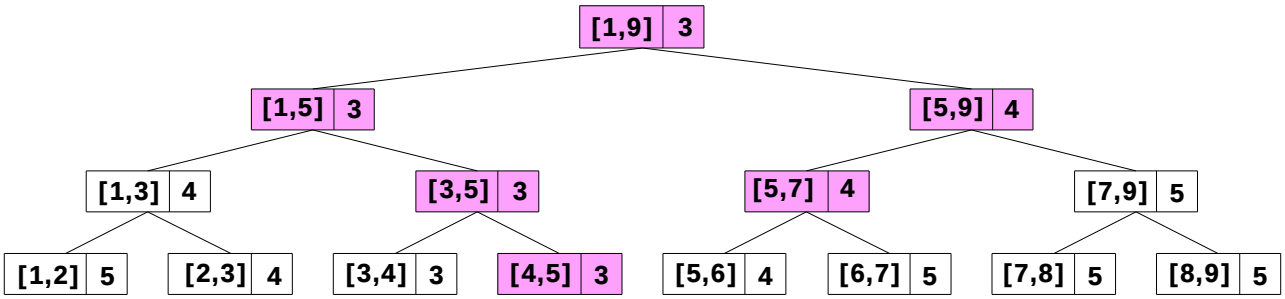
其实我们可以采用线段树的结构来实现，把整个铁路划分好之后，按照二叉树的思想来存储，如下图的例子所示：



其中“[1,9]”表示第 1 个车站和第 9 个车站直接的路段，后面的“3”表示这个路段中最少剩余空位的路段的空位数（下简称“空位数”），例如[1,2]区间有 5 个空位，[2,3]区间有 4 个空位，那么[1,3]区间的空位数就取最小的那个 4，表示通过[1,3]区间全路段的空位最多为 4 个。

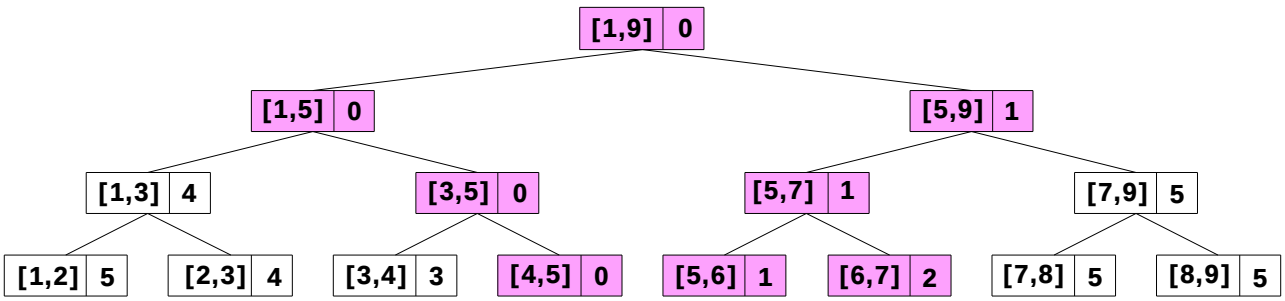
下面分析一下这种结构的复杂度。

对于查找操作，从树的根节点开始，不断地把查询区间分割后进入子节点递归查找，直到需查询区间与该节点区间相符后，再判断该区间的空位数是否满足需要。例如在上面例子中的线段数中查询[4,7]路段，相应要访问的节点如下：



在上图中，[4,5]、[5,7]这样的节点不需要再判断其子节点就可以得到该段是否满足需要的节点我们把他们暂称为“底端节点”。易证一次查询中最多会出现 4 个底端节点，且因为树的深度为 $\log n$，因此查找的效率为 $O(\log n)$。

对于修改操作，如果我们要把所有要改动的信息都进行修改的话，例如把[4,7]区间的座位数都减去 3，那么需要进行修改的节点如下：



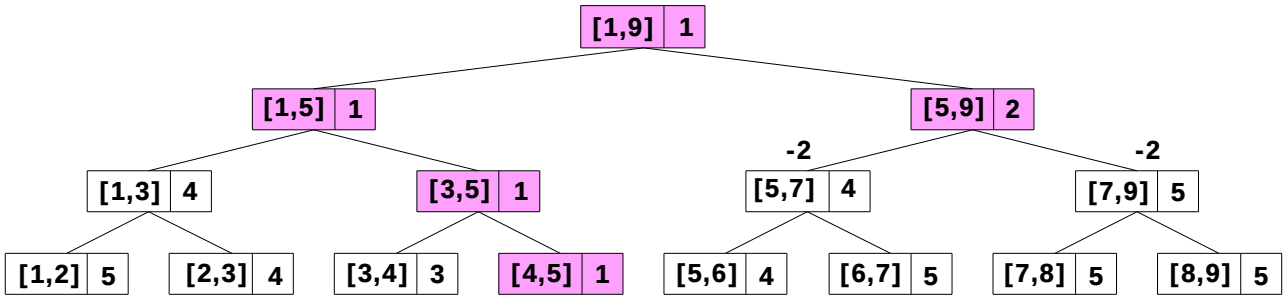
由这里可以看到，[4,7]实际上只有 3 个基本区间，但是我们需要修改的节点数却是 8！如果我们修改的区间范围更广呢？这样做时间效率比用线性表还低下。

这种做法其实局限在了一种惯性思维里面，例如上面的[5,6]、[6,7]这两个节点，我们对其进行了修改，但实际上，也许我们以后再也不会访问这两个节点了，也就是说，对这两个节点的操作毫无意义。

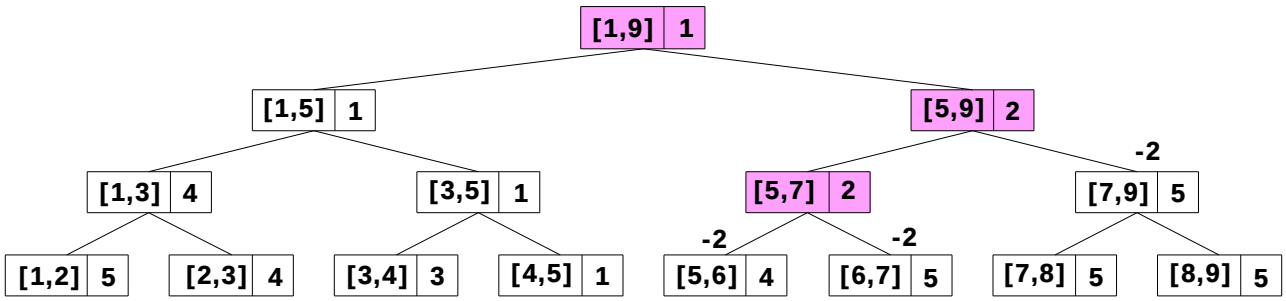
其实，我们换一个思维角度来考虑，其实是没有必要把所有的节点进行修改的，由于线段树是二叉树结构，所以我们完全可以记录下需要修改的东西，在需要时再修改！

例如我们需要在[4,9]区间减少 2 个座位，那么我们修改到[5,9]这个上面我们称做“底端

节点”的节点时，按照上面的方法，[5,9]的所有子节点及以下都要进行修改，但是我们完全可以不修改，而是给[5,9]这个节点的两个儿子节点一个修正标记“-2”，“告诉”他们要对空位数进行-2 的修正，如下图所示：



而当下次查找经过这些做了修正标记的节点时，例如查找[5,7]时，再用修正值对该节点的空位数进行修正，并把修正值传递给其子节点，如下图所示：



这样操作，对于不再查找的节点就不需要进行修改，而修改仅仅在查找时访问到该节点时进行，由于修改和向子节点传递修正标记仅仅是 O(1)的时间复杂度，因此查找效率并没有受到影响，而且通过这样的优化，每一次修改操作仅仅修改到底端节点就结束了，实际上修改的时间复杂度也降到了 $O(\log n)$ 。

由上看来，看似小小的一些细节处理，却让算法的时间复杂度产生了截然不同的两种情况。因此我们在注重算法的同时，也不应该忽略细节的处理，应该仔细揣度每个细节应该怎么处理，而不要在细节上犯错误，进而影响到整个算法的实现。

四、总结

关键性的细节对整个算法起着举足轻重的作用，正如我们上面所阐述的那样，关键细节处理的优劣，直接影响到算法的正确性，就算不影响正确性，也会或多或少地影响到算法的时间效率。我们对细节的处理不应该疏忽大意，否则将得不偿失。

【参考文献】

1. 数据结构与算法设计，王晓东编著，电子工业出版社
2. NOI2002 第一试试题

【参考程序】

〖例 1〗线性表维护

```
//没有优化的算法
Program Line_A;
```

```
Type
  PNode=^TNode;
  TNode=Record
          Data:String;
          Next:PNode;
        End;

Var
  Head,Now,Ending:PNode;

  Procedure Init;
  Begin
    Assign(Input,'line.in');
    Reset(Input);
    Assign(Output,'line.out');
    Rewrite(Output);
    New(Head);
    FillChar(Head^,SizeOf(Head^),0);
    Now:=Head;
    Ending:=Head;
  End;

  Procedure DoAdd;
  Var
    tmp:PNode;
    st:String;
  Begin
    Readln(st);
    New(tmp);
    tmp^.Data:=st;
    Ending^.Next:=tmp;
    tmp^.Next:=nil;
    Ending:=tmp;
  End;

  Procedure DoDel;
  Var
    a,i:Longint;
    tmp,tmp_:PNode;
  Begin
    Readln(a);
    tmp:=Now;
    For i:=1 To a+1 Do
    Begin
      tmp_:=tmp;
      tmp:=tmp^.Next;
      If i>1
      Then Dispose(tmp_);
      If tmp=nil
      Then Break;
    End;
    now^.Next:=tmp;
    If tmp=nil
    Then Ending:=now;
  End;

  Procedure DoMove;
  Var
    a,i:Longint;
  Begin
    Readln(a);
    If a=0
    Then Now:=Head
```

```
    Else For i:=1 To a Do
          If Now^.Next<>nil
          Then Now:=Now^.Next
          Else Break;
  End;

  Procedure DoPrint;
  Begin
    Writeln(Now^.Next^.Data);
  End;

  Procedure Main;
  Var
    Command:String;
  Begin
    While not Eof Do
    Begin
      Readln(Command);
      If Command='ADD'
      Then DoAdd;
      If Command='DEL'
      Then DoDel;
      If Command='MOVE'
      Then DoMove;
      If Command='PRINT'
      Then DoPrint;
    End;
  End;

  Procedure Done;
  Begin
    Close(Output);
  End;

Begin
  Init;
  Main;
  Done;
End.
```

```
//优化后的算法
Program Links;

Type
  PNode=^TNode;
  TNode=Record
         Data:String;
         Next:PNode;
        End;

Var
  Head,Now,Ending:PNode;
  Re_Head,Re_End:PNode;

  Procedure Init;
  Begin
    Assign(Input,'line.in');
    Reset(Input);
    Assign(Output,'line.out');
    Rewrite(Output);
    New(Head);
```

```
  FillChar(Head^,SizeOf(Head^),0);
  Now:=Head;
  Ending:=Head;
  New(Re_Head);
  FillChar(Re_Head^,SizeOf(Re_Head^),0);
  Re_End:=Re_Head;
End;

Procedure DoAdd;
Var
  tmp:PNode;
  st:String;
Begin
  Readln(st);
  If Re_Head^.Next=nil
  Then New(tmp)
  Else Begin
         tmp:=Re_Head^.Next;
         Re_Head^.Next:=Re_Head^.Next^.Next;
       End;
  tmp^.Data:=st;
  Ending^.Next:=tmp;
  tmp^.Next:=nil;
  Ending:=tmp;
End;

Procedure DoDel;
Var
  a,i:Longint;
  tmp:PNode;
Begin
  Readln(a);
  tmp:=Now;
  For i:=1 To a Do
  Begin
    tmp:=tmp^.Next;
    If tmp=nil
    Then Break;
  End;
  If Re_Head^.Next=nil
  Then Re_End:=Re_Head;
  Re_End^.Next:=now^.Next;
  Re_End:=tmp;
  now^.Next:=tmp^.Next;
  If tmp^.Next=nil
  Then Ending:=now;
  tmp^.Next:=nil;
End;

Procedure DoMove;
Var
  a,i:Longint;
Begin
  Readln(a);
  If a=0
  Then Now:=Head
  Else For i:=1 To a Do
         If Now^.Next<>nil
         Then Now:=Now^.Next
         Else Break;
End;

Procedure DoPrint;
Begin
```

```
    Writeln(Now^.Next^.Data);
  End;

  Procedure Main;
  Var
    Command:String;
  Begin
    While not Eof Do
    Begin
      Readln(Command);
      If Command='ADD'
      Then DoAdd;
      If Command='DEL'
      Then DoDel;
      If Command='MOVE'
      Then DoMove;
      If Command='PRINT'
      Then DoPrint;
    End;
  End;

  Procedure Done;
  Begin
    Close(Output);
  End;

Begin
  Init;
  Main;
  Done;
End.
```

```
//数据 1 的生成程序
Program Line_Make;

Var
  i,j:Longint;
  st:String;

Begin
  Assign(Output,'line.in');
  Rewrite(Output);
  For i:=1 To 10 Do
  Begin
    For j:=1 To 100000 Do
    Begin
      Str(j,st);
      Writeln('ADD');
      Writeln(st);
    End;
    Writeln('MOVE');
    Writeln(0);
    Writeln('PRINT');
    For j:=1 To 100000 Do
    Begin
      Writeln('DEL');
      Writeln(1);
    End;
  End;
  Close(Output);
End.
```

```
//数据 2 的生成程序
Program Line_Make;

Var
  i,j:Longint;
  st:String;

Begin
  Assign(Output,'line.in');
  Rewrite(Output);
  For i:=1 To 10 Do
  Begin
    For j:=1 To 400000 Do
    Begin
      Str(j,st);
      Writeln('ADD');
      Writeln(st);
    End;
    Writeln('MOVE');
    Writeln(0);
    Writeln('DEL');
    Writeln(300000);
    Writeln('PRINT');
    For j:=1 To 100000 Do
    Begin
      Writeln('DEL');
      Writeln(1);
    End;
  End;
  Close(Output);
End.
```

```
//数据 3 的生成程序
Program Line_Make;

Var
  i,j:Longint;
  st:String;

Begin
  Assign(Output,'line.in');
  Rewrite(Output);
  For i:=1 To 10 Do
  Begin
    For j:=1 To 400000 Do
    Begin
      Str(j,st);
      Writeln('ADD');
      Writeln(st);
      Writeln('MOVE');
      Writeln(0);
      Writeln('PRINT');
      Writeln('DEL');
      Writeln(1);
    End;
  End;
  Close(Output);
End.
```

```
//数据 4 的生成程序
Program Line_Make;

Var
  i,j,k,L:Longint;
  st:String;

Begin
  Assign(Output,'line.in');
  Rewrite(Output);
  Randomize;
  k:=0;
  For i:=1 To 10 Do
  Begin
    For j:=1 To 400000 Do
    Begin
      Repeat
        L:=Random(3);
      Until not ((L<>1)and(k=0));
      Case L Of
       0: Begin
            Writeln('MOVE');
            Writeln('0');
            Writeln('DEL');
            Repeat
              L:=Random(10)+1;
            Until L<=k;
            Writeln(L);
            Dec(k,L);
          End;
       1: Begin
            Writeln('ADD');
            Writeln(j);
            Inc(k);
          End;
       2: Begin
            Writeln('MOVE');
            Writeln('0');
            Writeln('PRINT');
          End;
      End;
    End;
  End;
  Close(Output);
End.



//数据 5 的生成程序
Program Line_Make;

Var
  i,j,k,L:Longint;
  st:String;

Begin
  Assign(Output,'line.in');
  Rewrite(Output);
  Randomize;
  k:=0;
  For i:=1 To 10 Do
```

```
  Begin
    For j:=1 To 800000 Do
    Begin
      Repeat
        L:=Random(3);
      Until not ((L<>1)and(k=0));
      Case L Of
       0: Begin
            Writeln('MOVE');
            Writeln('0');
            Writeln('DEL');
            Repeat
              L:=Random(10)+1;
            Until L<=k;
            Writeln(L);
            Dec(k,L);
          End;
       1: Begin
            Writeln('ADD');
            Writeln(j);
            Inc(k);
          End;
       2: Begin
            Writeln('MOVE');
            Writeln('0');
            Writeln('PRINT');
          End;
      End;
    End;
  End;
  Close(Output);
End.
```