对块状链表的一点研究

山西大学附中 苏煜

【摘要】

本文主要介绍了块状链表的概念，如何扩展块状链表,讨论了块状链表的性能以及在信息学竞赛中应用块状链表的利与弊，最后简要介绍了块状链表思想在实际生活中的应用。

【关键词】

块状链表 分块大小 性能 块状链表的扩展 模拟 骗分

一、什么是块状链表

我们先从题目入手，看看什么是块状链表：

NOI2003 editor

【题目大意】

一些定义：

文本：由 0 个或多个 ASCII 码在闭区间[32, 126]内的字符（即空格和可见字符）构成的序列。

光标：在一段文本中用于指示位置的标记，可以位于文本首部，文本尾部或文本的某两个字符之间。

文本编辑器：为一个包含一段文本和该文本中的一个光标的，并可以对其进行如下六条操作的程序。如果这段文本为空，我们就说这个文本编辑器是空的。

操作名称	输入文件中的格式	功能
MOVE(k)	Move k	将光标移动到第 k 个字符之后，如果 k=0，将光标移到文本开头
INSERT(n, s)	Insert n↵ S	在光标处插入长度为 n 的字符串 s，光标位置不变，$n \geq 1$
DELETE(n)	Delete n	删除光标后的 n 个字符，光标位置不变，$n \geq 1$
GET(n)	Get n	输出光标后的 n 个字符，光标位置不变，$n \geq 1$
PREV()	Prev	光标前移一个字符
NEXT()	Next	光标后移一个字符

比如一个空的文本编辑器依次执行操作 INSERT(13, “Balanced tree”)，MOVE(2)，DELETE(5)，NEXT()，INSERT(7, “ editor”)，MOVE(0)，GET(16)后，会输出“Bad editor tree”。

你的任务是：

建立一个空的文本编辑器。
从输入文件中读入一些操作并执行。
对所有执行过的 GET 操作，将指定的内容写入输出文件。

【数据范围】

- MOVE 操作不超过 50000 个，INSERT 和 DELETE 操作的总个数不超过 4000，PREV 和 NEXT 操作的总个数不超过 200000。
- 所有 INSERT 插入的字符数之和不超过 2M（1M=1024*1024），正确的输出文件长度不超过 3M 字节。
- DELETE 操作和 GET 操作执行时光标后必然有足够的字符。MOVE、PREV、NEXT 操作必然不会试图把光标移动到非法位置。
- 输入文件没有错误。

首先分析题目：
这道题的命令其实只有两类：1.定位；2.添加或删除。
这两类操作也正是两种常见顺序表实现方式的主要区别：

	数组	链表
定位	O(1)	O(N)
添加或删除	O(N)	O(1)

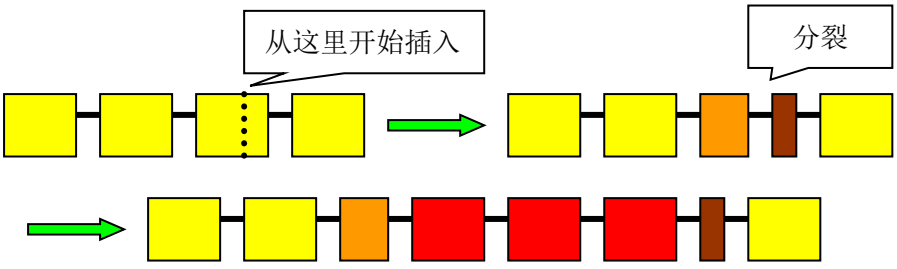
因为单个操作 O(N)复杂度的存在，无论我们用哪一种方法，都不可能 AC 这个题。但是如果我们将这两种方法结合起来，比如在整体上用链表，具体每一个链表节点改为一个大小适当（比如 1000、1500）的数组，那么就可以“优势互补”，得到两种操作更加平衡的数据结构，也就是所谓的“块状链表”。

再回到题目，如果用一个整数记录当前位置，那么我们需要一个支持以下操作的数据结构：
1、Insert 在指定位置添加指定长度信息；
2、Erase 从指定位置开始删除指定长度的信息；
3、Get 得到指定位置开始指定长度的信息。

具体实现很简单[1]：
首先我们实现两个内部基本操作：

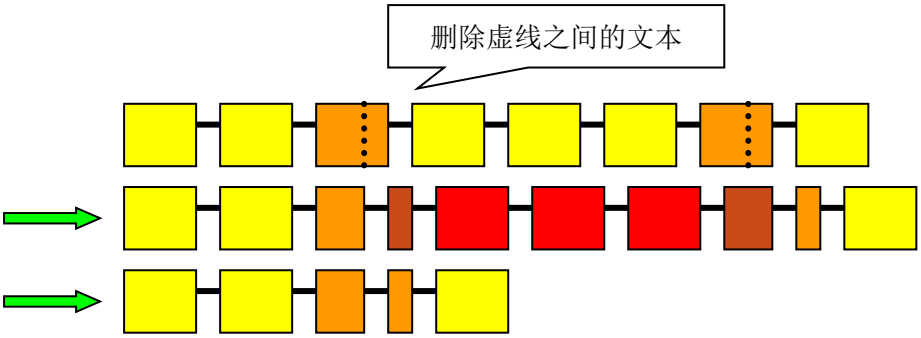
a. 定位：从第一个分块开始向后直到找到指定位置所在的分块和他在分块内的位置。
b. 分裂：将指定分块从指定位置分裂成为两个分块。

接下来实现外部基本操作：
1、Insert：找到指定位置，分裂块，添加新块直到添加完成。



[1]当然有更快的写法，但是这个方法可以省去不少判断，出错概率小一些。

2、Erase：找到起始位置，分裂首尾块，删掉中间的所有节点。



3、Get：找到指定位置，向后扫描直到找完所需数目。

还有一个问题：频繁的分裂操作可能会导致很多连续的块实际储存的数据都很少，大大降低了块状链表的效率，我们可以在每次操作后把过于小的连续分块合并起来。

这样就满足了题目的所有要求，同时也完成了一个最基本的块状链表。

二、块状链表的简单应用

可以感觉到，块状链表其实是对普通模拟操作的一种优化，它将两种并不优秀的方法结合起来，得到了一个比较实用的数据结构。利用它的这种性质，对有些题目，我们可以“强行”模拟地做：

Northeastern Europe 2003, Northern Subregion, KeyInsertion

【题目大意[2]】

N(1 <= N <= 131 072)个士兵在进行队列训练，从左至右有 M（1 <= M <= 131 072）个位置。每次将军可以下达一个命令，表示为 Goto(L,S)。

若队列 L 位置上为空，那么士兵 S 站在 L 上。

若队列 L 位置上有士兵 K，那么士兵 S 站在 L 上，执行 Goto（L+1，K）。

将军对 N 个士兵依次下达 N 个命令，每个士兵被下达命令一次且仅一次。要你求出最后队列的状态。（有可能在命令执行过程中，士兵站的位置标号超过 M，所以你最后首先要求出最终的队列长度。0 表示空位置）。

我们可以用一个比较笨的方法：插入操作其实就是把当前位置及其之后的那一个连续块后移一位，然后再在这个地方加入新的元素，其实也就是把这个连续块后面紧跟的一个空位置删掉，再在这个位置插入一个新元素。动态维护某个位置后的第一个空位置是并查集的经典应用，而对大数列的插入删除操作又是块状链表的强项，这样，我们就可以直接套用并查集与块状链表解决这个题。

其实本来这个题是这样做的[3]：

想到“第 x 个空位置”和这个题的运作方式很类似：某一个元素的变化会使得它后面的一串元素发生连锁反应。用并查集维护每一个不相邻的分块，并用链表存储这个分块中的元素，当士兵 A 直接插入到一个已经属于某一个块 B 的位置中时，就将与 A 放在 B 块的链表首。当士兵 A 的插入引起了一个或者多个块相连时，就把位置在后的分块的链表接到靠前的分块的链表之前。插入结束后，

[2] 题目翻译来自 05 年龙凡的冬令营论文《序的应用》。

[3] 详细讨论超出本文范围，更多内容请参考 05 年龙凡的冬令营论文《序的应用》。

从最后一个分块开始按着分块链表的顺序依次把每一个元素 x 插入到第 L[x]个空位上。这样就得到了最终的序列。

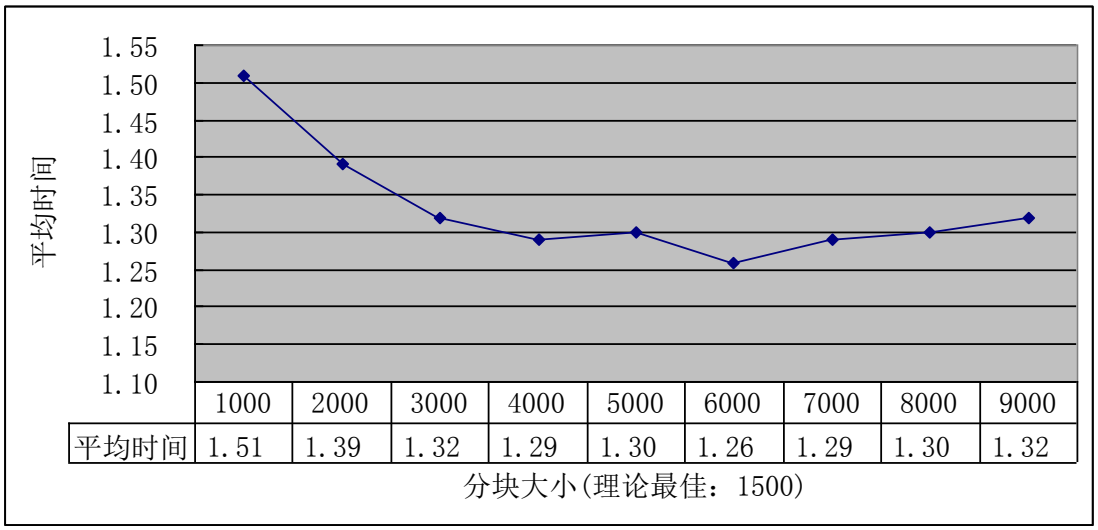
这个做法的描述就已经略显复杂，更别说在比赛的时候想出来了（事实上，当年比赛的时候没有一个队伍做出了这个题），而如果用块状链表，基本上是不用怎么思考的，甚至可以讲就是在“骗分”了。

三、性能分析：单块大小的选择

在理想情况下，假设我们把小数组的大小设为 x，那么就会有 $y = n/x$ 个“小块”，那么，定位操作 $O(y)$，添加的附加影响 $O(x)$，显然，当 $x = \sqrt{n}$ 的时候两类操作的复杂度最接近，两类操作也就最平衡（这并不意味着速度最快）。这样，整体的复杂度就是 $O(n\sqrt{n})$。

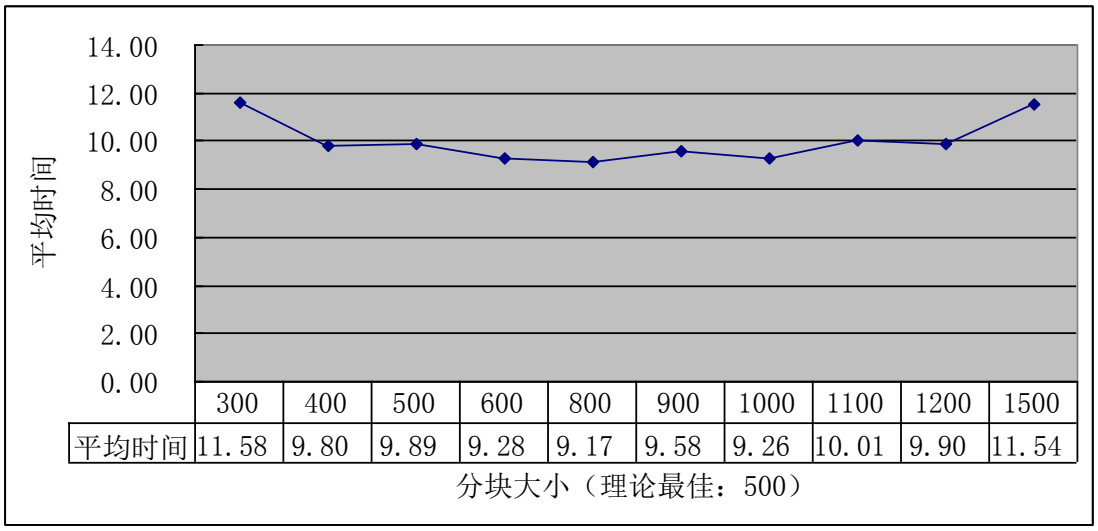
虽然实际应用中 n 是不断变化的，但是取 $\sqrt{\max n}$ 就可以在最坏情况下仍有 $O(n\sqrt{n})$ 的效率，整体的复杂度依然大致是 $O(n\sqrt{n})$。也就是说，我们用块状链表其实是在追求一种平衡。

我们看一下取不同的分块大小通过 NOI 2003 editor 的 10 个数据所用的时间[4]：



下面是不同的分块大小通过 Key Insertion 的 38 个数据所需的运行时间：

[4] 测试用机：CPU：1.8GHz*2 内存：1G
测试用代码见附件。



	300	400	500	600	800	900	1000	1100	1200	1500
平均时间	11.58	9.80	9.89	9.28	9.17	9.58	9.26	10.01	9.90	11.54

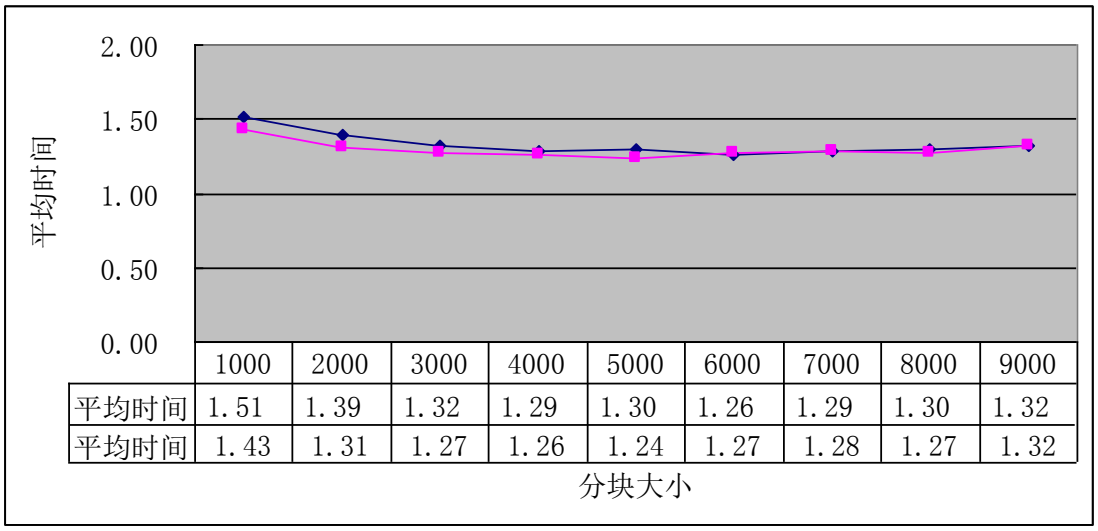
我们发现分块大小选择理论值并不是最快的，而选择比理论最优值较大的分块大小速度比较快，这其实是因为我的写法的问题：插入删除操作都依赖分裂操作，这样使得分裂操作比较频繁，从而使绝大数分块达不到饱和，也就是每个分块的实际存储大小要比给它限制的最大值小，所以为了使分块的实际存储大小接近 $\sqrt{n}$，分块的最大容量应该比 $\sqrt{n}$ 大。又因为当相邻两个分块存储大小都小于最大容量一半的时候会合并，所以这个最大容量不需要比 $2\sqrt{n}$ 大。

editor 一题我们看到实际上分块容量达到了 6000 左右时速度才最快，为什么与理论分析差别比较大呢？分析数据我发现很多数据里都有大量这样的命令组：

MOVE x	PREV	NEXT
GET 1	GET 1	GET 1

这三种命令组的特点就是不影响分块结构，而且因为只取一个字符，取字符操作与分块大小无关。但是在取字符之前，需要一次定位操作，也就是说，这三种命令组其实就是变相的定位操作。显然，分块越大定位操作越快，而数据的主要操作就是定位，所以总体速度会加快。另一个原因是题目规定效率与分块大小关系比较紧密的插入和删除操作总共不超过 4000 条[5]，这样使得分块大小对整个程序的影响不是很明显，主要的时间都花在定位操作上了。我另外又写了一个使用两个变量（当前位置所在块，当前位置在块内的相对位置）来记录当前位置的版本（因为我使用的是单向链表，所以 PREV 操作难免还要使用定位操作），它的运行时间和原先的比较：

[5] 如此少的插入删除操作使得一些人想到了一种比较特殊的方法：“实时离散化”，具体做法可以参考 2007 年余江伟的集训队论文《如何解决好动态统计问题》



	1000	2000	3000	4000	5000	6000	7000	8000	9000
平均时间	1.51	1.39	1.32	1.29	1.30	1.26	1.29	1.30	1.32
平均时间	1.43	1.31	1.27	1.26	1.24	1.27	1.28	1.27	1.32

我们看到效率有所提高，现在效率最高的分块大小在 5000 左右，而且分块越大提高的效果越差，说明原先大分块效率高有很大一部分原因在于定位操作。如果使用双向链表，作出一个类似迭代器的东西，我想小分块的速度会再次提高，不过所需的代码也会加长，而之所以我的基本块状链表里没有迭代器这么一个元素就是因为迭代器的应用并不是很广（比如 KeyInsertion 就不需要迭代器），而维护一个迭代器会使我们添加很多不必要的代码，容易出错。

而 KeyInsertion 一题分块大小从 400 到 1200 所需的时间都差不多，在 800 左右有最高的效率，也就是在 $\sqrt{n}$ 和 $2\sqrt{n}$ 之间效率最高。

由以上的时间数据可以看出，虽然块状链表的效率是与实际数据分不开的，但是一般来说，选用比 $\sqrt{n}$ 大一些的分块容量能得到比较不错的效率，而选用比 $\sqrt{n}$ 大一些的分块容量其实是为了每个分块实际存储 $\sqrt{n}$ 的信息，也就是说，“追求平衡”在一般情况下能得到不错的效率。

另外，因为频繁的分裂操作使得程序经常要申请、释放内存，全部的程序我使用的都是静态数组并自己维护一个可用节点列表。

四、块状链表的扩展

我们再看一道比较复杂的题目：

NOI2005 sequence

【题目大意】

请写一个程序，要求维护一个数列，支持以下6种操作：（请注意，格式栏中的下划线‘ _ ’表示实际输入文件中的空格）

操作编号 输入文件中的格式 说明

1. 插入 INSERT_*posi*_*tot*_*c*1_*c*2_..._*ctot* 在当前数列的第*posi*个数字后插入*tot*个数字：*c*1, *c*2, …, *ctot*；若在数列首插入，则*posi* 为0
2. 删除 DELETE_*posi*_*tot* 从当前数列的第*posi* 个数字开始连续删除*tot*个数字
3. 修改 MAKE-SAME_*posi*_*tot*_*c* 将当前数列的第*posi* 个数字开始的连续*tot*个数字统一修改为*c*

4. 翻转 REVERSE_*posi_tot* 取出从当前数列的第*posi* 个数字开始的*tot*个数字，翻转后放入原来的位置
5. 求和 GET-SUM_*posi_tot* 计算从当前数列开始的第*posi* 个数字开始的*tot*个数字的和并输出
6. 求和最大的子列 MAX-SUM 求出当前数列中和最大的一段子列，并输出最大和

【数据规模和约定】
你可以认为在任何时刻，数列中至少有1个数。
输入数据一定是正确的，即指定位置的数在数列中一定存在。
50%的数据中，任何时刻数列中最多含有30 000个数；
100%的数据中，任何时刻数列中最多含有500 000个数。
100%的数据中，任何时刻数列中任何一个数字均在[-1 000, 1 000]内。
100%的数据中，*M* ≤20 000，插入的数字总数不超过4 000 000个，输入文件大小不超过20MBytes。

很明显，直接模拟就可以做，只不过不能 AC，而且看起来没有什么别的什么算法可以解决这种题，这时候，又需要块状链表了。

这次操作增加了一些：
1、将指定位置开始的指定长度的子串反转；
2、将指定位置开始的指定长度的子串设置为某一个指定的值；
3、求出指定连续子串的和；
4、求出当前数列的最大连续子串和。

块状链表的优越性就在于它可以整块整块地操作，所以对于以上几种操作，我们要对分块保存的信息做一些扩充，以便于可以整块整块地操作。

要求最大连续子串和，又要整块整块地操作，很明显的方法就是记录子块内的最大连续子串和，子块的总和，以及分别包含子块的左右两个端点的最大连续子串和。

要快速的将某一个子块设置为某一特定值，最快的方法就是给子块设置一个标记，标记当前子块内的元素是否都一样，另设一个整数记录这个特定值。

同理，每个子块添加反转标记，记录当前子块是否是反转的。

这时候就看出先前不太快但简单的写法的好处了：插入删除都不用考虑这些新加的信息，REVERSE 和 MAKE-SAME 也可以套用相似的结构。这样，信息的更新就集中在分裂操作上，集中精力写好分裂操作就行了。

一个小技巧：记录包含左右端点的最大连续子串和可以用一个数组 sidemax[2]，sidemax[0]记录子块中包含左端点（不管是否反转）的最大连续子串和；sidemax[1]记录子块中包含右端点（不管是否反转）的最大连续子串和，这样 sidemax[revesed] 就正好是实际上包含左端点的最大连续子串和，sidemax[!reversed]就是实际上包含右端点的最大连续子串和。

五、应用更加复杂的块状链表

CERC2007 sort

【题目大意】

在一个车间里有 N（1<=N<=100000）个零件排成一列，已知他们各自的高度，现在要将他们按高度排列成升序序列，规定只能使用如下方法：

找到最低的零件的位置 P1，将区间[1,P1]反转，再找到第二低的零件的位置 P2，将区间[2,P2]反转……

要求你的程序输出 P1,P2,P3…

如果有一样高的零件，那么优先处理在原始序列中靠前的零件。

有什么特殊算法么？平衡树？线段树？堆？好像一下子想不出来。

对，干脆直接模拟做算了！

依然是块状链表，因为需要反转，所以添加反转操作；因为每一次都是找当前序列中最小的元素，所以分块添加当前块最小值属性。

每一步，找到包含所需最小值的子块，再在这个子块里找到那个最小值，也就得到了它的位置，然后反转，删掉第一个元素，如此反复操作就可以了。

唯一的问题：可能会有相同权值的元素，而且还要求优先处理在原始数列里先出现的。这个其实可以转化掉，既然我们可以确定原来序列中每个元素的具体处理次序，那么就把它们的权值换成它们的处理次序，这样就避免了相同权值的问题。接下来直接写块链就完事了。

NOI2007 necklace

【题目大意】

TH 公司打算推出一款项链自助生产系统，使用该系统顾客可以自行设计心目中的美丽项链。请帮助 TH 公司编写一个软件模拟系统。

一条项链包含 N（1<=N<=500000） 个珠子，每个珠子的颜色是 1, 2, …, c 中的一种。项链被固定在一个平板上，平板的某个位置被标记位置 1，按顺时针方向其他位置被记为 2,3,…,N。

你将要编写的软件系统应支持如下命令：

R k (0<k<N) 意为 Rotate k。将项链在平板上顺时针旋转 k 个位置, 即原来处于位置 1 的珠子将转至位置 k+1，处于位置 2 的珠子将转至位置 k+2，依次类推。

F 意为 Flip。将平板沿着给定的对称轴翻转，原来处于位置 1 的珠子不动，位置 2 上的珠子与位置 N 上的珠子互换，位置 3 上的珠子与位置 N-1 上的珠子互换，依次类推。

S i j (1≤i , j≤N) 意为 Swap i , j。将位置 i 上的珠子与位置 j 上的珠子互换。

P i j x (1≤i , j≤N, x≤c) 意为 Paint i , j , x。将位置 i 沿顺时针方向到位置 j 的一段染为颜色 x。

C 意为 Count。查询当前的项链由多少个“部分”组成，我们称项链中颜色相同的一段为一个“部分”。

CS i j (1≤i , j≤N) 意为 CountSegment i , j。查询从位置 i 沿顺时针方向到位置 j 的一段中有多少个部分组成。

这次需要维护的信息变成了颜色段数，为了方便，我们再记录左右端点的颜色，而翻转，旋转都可以用原先的 Reverse 实现；Count 操作可以利用 CountSegment 实现。也就是我们需要两个新操作：Swap，CountSegment。

Swap 这个操作很简单：找到要交换的那两个元素，如果它们不同，就交换，然后分别更新一下子块信息。

CountSegment 和以前的求和操作很类似：首尾块特殊计算，中间块利用块信息。

六、性能分析：编程复杂度与时间效率

CERC2007 sort 我写了 3.44kb 的代码，速度却不是很快，由于不知道当时考试的时限，我拿标程（4.94kb，包含大量注释与空白）和我的程序做了对比：标程 0.61s，我的 3.63s。

NOI2007 necklace 我写了 6.34kb 的代码，只能过 7 个点[6]。如果能意识到反转、旋转都是“骗人的”，加上优化后可以过 9 个点（当然，效果这么差也有我个人的水平问题，但至少说明对于一般人来说，很难把块状链表写到足够快）。而且如果你能意识到可以用坐标变换解决反转和旋转操作，就应该能想到用更加简单快速的线段树了。毕竟 $\sqrt{n}$ 和 $\log_2 n$ 有几十倍的差距，块状链表的常数又比线段树大很多。余林韵同学写的线段树的 pascal 代码才 4.12kb，但比起我的块状链表速度飞快。

七、总结

其实我本来是想介绍用块状链表来骗分，因为用模拟的方法做题很省时间思考，但是最后我发现用块状链表骗分效果并不好，首先，块状链表一般都比较长，要花不少时间来完成，虽然思考正解也会花很多时间，但是过长的代码很容易写错，尤其是现在的题目往往是两种或者更多数据结构的结合（比如 KeyInsertion），或是要求在模拟之前做一些转换（比如 sort），或者要求写有更多功能的数据结构（比如 sequence），甚至几种兼有（比如 necklace），这样很难保证代码的正确性。其次，即使能够很快很正确地写好所需要的块状链表，它的效率也难以保证，如果分块大小选择不慎效率将会比较低，有可能超出时限（虽然比模拟快很多），有时甚至选择了最佳分块也有可能 TLE，毕竟对于大数据，$n\sqrt{n}$ 和 $n\log_2 n$（有些题目甚至是 $O(n)$ 的）差了几十倍。

但是，块状链表并不是一无是处，它结合了数组和链表这两个基本的各有缺陷的数据结构，利用整体处理的方法，平衡了两类基本操作的复杂度，从而提高了效率。这种结合基本元素甚至是效率比较差的基本元素得到高效结果的思想还是很有借鉴意义的。

而且块状链表非常好扩展，只要是序列操作，比如：统一赋值，翻转，求和，维护最小值等等，都可以使用块状链表得到 $\sqrt{n}$ 的复杂度，而如果将整个块状链表维护成有序的，它甚至可以实现平衡树的一些操作[7]，毕竟平衡树也可以看作是一种维护序列的方法。

又因为块状链表只在每个分块记录一些额外信息，它的空间利用率很高，而同是模拟方法的 Splay 需要在每个节点上维护全部额外信息，虽然速度比较快，

[6] 我写了一个稍微复杂的块状链表，能过 8 个，第 9 个略微超时，如果再复杂一些，也许是可以 AC 的。

[7] 我用块状链表做 NOI2006 happybirthday，略微比我写的 SBT 短一些，可以过 8 个点（第 8 个点接近时限）。

却占用大量内存[8]。

其实，在日常生活中我们经常会用到块状链表：传统的 FAT 文件系统就是将磁盘扇区分簇，然后用 FAT 表(File Allocation Table 文件分配表)来记录每一个簇的状态：是否损坏，是否被使用，如果被使用那么它的下一个簇是哪一个簇。可见，FAT 文件系统的思想和块状链表是一致的。

而且因为块状链表空间利用率很高，分块的结构又能很方便的和缓冲区结合使用，Vim[9]也使用了块状链表，在内存的存储和在磁盘上的缓冲都使用了类似块状链表的结构[10]。试想如果用 Splay 去写一个文本编辑器会是多么复杂而抽象，它又如何方便地利用缓冲区，一旦发生崩溃、断电等意外事件，又如何从磁盘缓冲中重构树结构、恢复数据？

另外，已经有人在 g++的<ext/rope>库中写了一个基本的块状链表模板：__gnu_cxx::rope<T, Alloc>，也就是说，使用 C++的同学可以很方便的得到一个现成的块状链表[11]。

【感谢】

感谢刘汝佳老师对论文选题、论文内容给予的帮助，感谢候晓静老师对论文内容的修改意见，感谢陈延辉同学提供 NOI2005 sequence 的代码，让我学到了不少写块状链表的经验，感谢高逸函同学解释块状链表的复杂度估计，感谢周梦宇同学告知<ext/rope>库中的块状链表模板。

【参考资料】

2005 年龙凡的集训队论文《序的应用》
2007 年余江伟的集训队论文《如何解决好动态统计问题》
余林韵同学的 NOI2007 题解
http://neerc.ifmo.ru/past/2003.html 上 KeyInsertion 一题的标程、数据
http://contest.felk.cvut.cz/07cerc/上 sort 一题的标程、数据
http://www.free-soft.org/FSM/english/issue01/vim.html 上 Bram Moolenaar 写的 vim 介绍，以及不知道谁翻译的中文文档
http://www.sgi.com/tech/stl/上的有关 rope 模板的相关内容

[8] 利用 Splay 可以把模拟操作的复杂度降到 $\log_2 n$（其实这也是空间换时间的例子：线段树、Splay 维护的额外信息多，空间占用大，但是速度也快），关于 Splay 的讨论可以参考 2007 年余江伟的集训队论文《如何解决好动态统计问题》，2004 年杨思雨的集训队论文《伸展树的基本操作与应用》。

[9] Vi IMproved，应该算是很有名了吧。

[10] 就像 Bram Moolenaar 所说：A text editor is used all day by many people; it is worth investing time and effort in making it work well. Vim 使用的数据结构很复杂，但是原理和块状链表是相似的。

[11] http://www.sgi.com/tech/stl/Rope.html 上有一份 rope<T, Alloc>的文档。