

基于平衡因子的 AVL 树设计实现

杜薇薇¹, 张翼燕¹, 瞿春柳²

(1. 中国科学技术信息研究所, 北京 100038;

2. 北京掌上通网络技术有限公司, 北京 100083)

摘 要:平衡二叉树又称 AVL 树,得名于它的发明者 G. M. Adelson-Velsky 和 E. M. Landis。作为一种常用的数据结构,许多教科书都详细描述了实现的算法,但是基本都是根据不同树形 LL、RR、LR、RL 给出相应逻辑,而且都是直接给出结论。而文中则以平衡因子为出发点,揭示了不同树形的一致性算法,第一次以数学公式推演,论证了 AVL 插入和删除操作在不同树形情况下,哪个节点开始失去平衡,怎么平衡以及哪个节点平衡结束,并给出算法的完整实现代码,使 AVL 的实现一致、简单、易懂。

关键词:AVL; 二叉树; 平衡因子

中图分类号:TP311.12

文献标识码:A

文章编号:1673-629X(2010)03-0024-04

AVL Tree Design and Implementation Based on Balancing Factor

DU Wei-wei¹, ZHANG Yi-yan¹, QU Chun-liu²

(1. Institute of Scientific and Technical Information of China, Beijing 100038, China;

2. China Dotman Co. Ltd, Beijing 100083, China)

Abstract: AVL tree, also known as a balanced binary tree, named after its inventors G. M. Adelson-Velsky and E. M. Landis. As a common data structure, it is described in detail the realization of the algorithm in many textbooks, but based on different tree shape LL, RR, LR, RL are given the corresponding logic, and conclusions are given directly. The balance factor for this article was the starting point, revealing the coherence of different tree algorithm, the first mathematical formula to deduce, demonstrates the AVL insertion and deletion in different tree shape, the node which started to lose balance and how balance and which end of the balance node, and gives a complete code for the realization of the algorithm.

Key words: AVL; binary tree; balance factor

0 引言

在计算机科学中,程序代码被认为是数据结构+算法。可见数据结构在计算机科学中的地位。专门讲解数据结构的书籍已经非常多了,其论述大都包括堆栈、队列、链表等基本数据结构,还包括树、二叉树、图等复杂数据结构。

现在要接触到的就是其中树的一种,更准确讲是二叉树的一种,是一种二叉排序树,一种平衡的二叉排序树,史称 AVL 树,被认为是最早发明的自平衡二叉排序树,也是目前研究和应用最为广泛的一种平衡二叉排序树。AVL 树得名于它的发明者 G. M. Adelson-Velsky 和 E. M. Landis,他们在 1962 年的论文《An algorithm for the organization of information》中发表了

它。此后,对它的研究一直延续到现在,尤其在复杂的动态数据结构查找中应用广泛。

1 定义

AVL 树,全称平衡的二叉排序树^[1-3],其关键是平衡,用平衡因子体现。平衡因子定义为节点左子树的高度减去其右子树的高度^[4]。带有平衡因子 -1、0 或 1 的节点被认为是平衡的。当这个平衡因子为 -1、0、1 的时候分别是偏左平衡,等高平衡和偏右平衡。带有平衡因子 -2 或 2 的节点被认为是不平衡的,需要重新平衡这个树。平衡因子可以直接存储在每个节点中,或通过存储节点子树高度计算出来。常见二叉树 C++ 描述为:

```
template <class T>
struct Node
{
    T data; // 节点保存的数据
```

收稿日期:2009-07-22;修回日期:2009-10-15

作者简介:杜薇薇(1976-),女,硕士研究生,研究方向为电子技术、图情学。

```
Node * left; // 左子树节点
Node * right; // 右子树节点
Node * parent; // 双亲树节点
int bf; // 平衡因子 balance factor
```

};

2 旋 转

AVL树是一种平衡的二叉排序树^[5]。每次进行插入或者删除操作的时候,首先是执行二叉排序树的插入或者删除,此时可能破坏二叉排序树的平衡性,即平衡因子变为-2或2,这就需要重新平衡这个树,做一次或者多次平衡化操作可以使它重新平衡。

旋转^[6],就是平衡需要执行的操作,先看图1、图2。

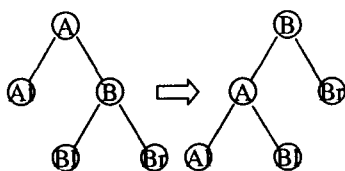


图1 左旋

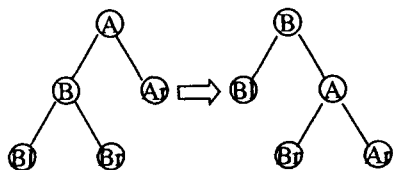


图2 右旋

看图1,假设A左子树高度小于右子树高度,平衡因子为负,要使A右子树高度减少,就必须使B上移到A的位置变成A,B的高度自然就少了1,然后使A成为B的子树,B的左子树成为A的右子树,此时A的高度是 A_l 和 B_l 中的大者+1。由于原来的A变成了B的子树,原来的B变成了A,所以B的高度至少增加了1。

先展示一下左旋的过程:

```
B = A->right;
B->parent = A->parent;
A->parent = B;
if (B->left) B->left->parent = A;
A->right = B->left;
B->left = A;
A = B;
```

同理,展示一下右旋的过程:

```
B = A->left;
B->parent = A->parent;
A->parent = B;
if (B->right) B->right->parent = A;
A->left = B->right;
```

B->right = A;

A = B;

现在,用简单的数学公式推导一下。先假设H表示旋转前的高度,h表示旋转后的高度,比如 H_A 表示A旋转前的高度, h_A 表示A旋转后的高度。假设BF表示旋转前的平衡因子,bf表示旋转后的平衡因子,比如 BF_A 表示A旋转前的平衡因子, bf_A 表示A旋转后的平衡因子。也就是大写表示旋转前,小写表示旋转后。另外MAX表示求大值,比如 $MAX(H_A, H_B)$ 表示 H_A 和 H_B 中的大值。还是先看左旋,旋转前:

$BF_A < 0$,又 $BF_A = H_{Al} - H_{Br}$,所以 $H_{Al} < H_{Br}$,这个非常显然。

$$BF_B = H_{Bl} - H_{Br}$$

旋转后

$$bf_A = h_{Al} - h_{Br}$$

$$bf_B = h_A - h_{Br}$$

开始演绎

$$bf_A - BF_A = (h_{Al} - h_{Br}) - (H_{Al} - H_{Br})$$

$$\text{由图可知 } h_{Al} = H_{Al}, h_{Br} = H_{Br}, h_{Br} = H_{Br}$$

所以 $bf_A - BF_A = H_{Br} - H_{Br}$,显然只决定于旋转前。

注意了,不要简单地认为上式的结果为1,先分析一下, H_{Br} 是什么?

$$H_{Br} = 1 + MAX(H_{Bl}, H_{Br}), \text{这样就很显然了,}$$

或许 $H_{Br} = 1 + H_{Bl}$,或许 $H_{Br} = 1 + H_{Br}$,那究竟是个呢?

关键看 BF_B , $BF_B = H_{Bl} - H_{Br}$

$$BF_B < 0, H_{Br} = 1 + H_{Br}; \text{否则, } H_{Br} = 1 + H_{Bl}$$

$$\text{那么 } BF_B < 0, \triangle_A = bf_A - BF_A = H_{Br} - H_{Bl} = 1 + H_{Br} - H_{Bl} = 1 - BF_B;$$

$$\text{否则, } \triangle_A = bf_A - BF_A = H_{Br} - H_{Bl} = 1 + H_{Bl} - H_{Bl} = 1$$

再看

$$bf_B - BF_B = (h_A - h_{Br}) - (H_{Bl} - H_{Br})$$

$$\text{同样 } h_{Al} = H_{Al}, h_{Bl} = H_{Bl}, h_{Br} = H_{Br}$$

所以 $bf_B - BF_B = h_A - H_{Bl} = h_A - h_{Bl}$,显然只决定于旋转后。

$$\text{而 } h_A = 1 + MAX(h_{Al}, h_{Bl})$$

同样,或许 $h_A = 1 + h_{Al}$,或许 $h_A = 1 + h_{Bl}$,那究竟是个呢?

关键看 bf_A , $bf_A = h_{Al} - h_{Br}$,

$$bf_A < 0, h_A = 1 + h_{Br}; \text{否则, } h_A = 1 + h_{Al}$$

$$\text{那么 } bf_A < 0, \triangle_B = bf_B - BF_B = h_A - h_{Bl} = 1 + h_{Br} - h_{Bl} = 1;$$

$$\text{否则, } \triangle_B = bf_B - BF_B = h_A - h_{Bl} = 1 + h_{Al} - h_{Bl} = 1 +$$

bf_A

综合一下,对于左旋 $BF_A < 0$, 即 $[-2, -1]$ 。

bf_A 首先肯定要 +1, 此时 bf_A 是 $[-1, 0]$, 如果 $BF_B < 0$, 其实就是 -1, 那么还要再 +1, 此时 bf_A 是 $[0, 1]$; 而 BF_B 是 $[-1, 0, 1]$, 首先肯定也是要 +1, 此时 bf_B 是 $[0, 1, 2]$, 如果 $0 < bf_A$, 其实就是 1, 而且此时的 BF_B 必是 -1, bf_B 先 +1 后再 +1, 结果等于 1, 综合 bf_B 取值是 $[0, 1, 2]$ 。

用一句话表述就是 $bf_A + 1$, 如果 $BF_B < 0$ 则再 - BF_B , $bf_B + 1$, 如果 $0 < bf_A$ 则再 + bf_A 。左旋时平衡因子变更 C++ 算法描述如下:

+ + A - > bf_- ; if (B - > bf_- < 0) A - > bf_- - = B - > bf_- ;

+ + B - > bf_- ; if (0 < A - > bf_-) B - > bf_- + = A - > bf_- ;

同理,

右旋, $0 < BF_A$,

那么 $BF_B < 0$, $\Delta_A = bf_A - BF_A = -1$;

否则, $\Delta_A = bf_A - BF_A = -1 - BF_B$

那么 $bf_A < 0$, $\Delta_B = bf_B - BF_B = -1 + bf_A$;

否则, $\Delta_B = bf_B - BF_B = -1$

综合一下,对于右旋 $0 < BF_A$, 即 $[1, 2]$ 。

bf_A 首先肯定要 -1, 此时 bf_A 是 $[0, 1]$, 如果 $0 < BF_B$, 其实就是 1, 那么还要再 -1, 此时 bf_A 是 $[-1, 0]$; 而 BF_B 是 $[-1, 0, 1]$, 首先肯定也是要 -1, 此时 bf_B 是 $[-2, -1, 0]$, 如果 $bf_A < 0$, 其实就是 -1, 而且此时的 BF_B 必是 1, bf_B 先 -1 后再 -1, 结果等于 -1, 综合 bf_B 取值是 $[-2, -1, 0]$ 。右旋时平衡因子变更 C++ 算法描述如下:

- - A - > bf_- ; if (0 < B - > bf_-) A - > bf_- - = B - > bf_- ;

- - B - > bf_- ; if (A - > bf_- < 0) B - > bf_- + = A - > bf_- ;

很显然,左旋 $BF_B = 1$ 时,旋转会使 B 失衡;右旋

$BF_B = -1$ 时,旋转会使 B 失衡。那么在左旋的时候就要避免 $BF_B = 1$, 从表 1 中不难发现在 $BF_A = 1$ 时作一次右旋时, bf_A 就不会出现 1, 太妙了, 那么只要在 $BF_B = 1$ 时, 对 B 作一次右旋, 就保证对 A 作左旋的时候不会失衡。同样, 在 $BF_B = -1$ 时, 对 B 作一次左旋, 就保证对 A 作右旋的时候不会失衡。

3 插 入

插入和删除首先要执行查找^[7]操作, 插入算法描述如下:

1) 类似查找操作, 区别是: 如果找到节点, 说明已经存在插入值了, 不能插入; 查找过程中还需要记录空节点的位置和其双亲节点, 即插入点是哪个节点的左子树或右子树。

2) 在找到空节点之后, 新建一个节点替代空节点, 填上插入值就完成了。

3) 平衡因子调整, 如果插入点是双亲的左子树, 双亲平衡因子 +1, 否则双亲平衡因子 -1。

4) 插入如果导致失衡, 那么失衡必定是在插入节点的双亲节点开始平衡, 其平衡因子 -2 或者 2, 相应的平衡就是左平衡或者右平衡^[8]。

由于插入节点必定是叶子节点, 而它的双亲节点插入前必定是平衡的, 平衡因子是 $[-1, 0, 1]$, 插入是左子树平衡因子只能是 $[-1, 0]$, 插入后是 $[0, 1]$; 插入是右子树平衡因子只能是 $[0, 1]$, 插入后是 $[-1, 0]$ 。或者换一个角度, 从高度来分析, 插入是左子树双亲高度是右子树的高度 $[0, 1]$, 插入后双亲的高度必是 1; 插入是右子树双亲高度是左子树的高度 $[0, 1]$, 插入后双亲的高度必是 1。双亲高度是 1, 必定双亲是平衡的。那是否就不需要平衡呢? 还是需要的, 但平衡是从插入节点双亲的双亲节点开始。这次考察的关键是在插入前后, 双亲节点的高度有没有发生变化, 如果没有变化, 那么双亲的双亲就可以保持平衡。显然如果双亲高度不变, 双亲的双亲必是平衡的, 就不需要平

表 1 平衡因子在左旋右旋时的变化情况

左旋 $BF_A < 0, BF_A = -2$				左旋 $0 < BF_A, BF_A = -1$			
	$BF_B = -1$	$BF_B = 0$	$BF_B = 1$		$BF_B = -1$	$BF_B = 0$	$BF_B = 1$
bf_A	0	-1	-1	bf_A	1	0	0
bf_B	0	1	2	bf_B	1	1	2
右旋 $0 < BF_A, BF_A = 1$				右旋 $0 < BF_A, BF_A = 2$			
	$BF_B = -1$	$BF_B = 0$	$BF_B = 1$		$BF_B = -1$	$BF_B = 0$	$BF_B = 1$
bf_A	0	0	-1	bf_A	1	1	0
bf_B	-2	-1	-1	bf_B	-2	-1	0

衡,整棵二叉树都是平衡的。如果双亲高度变了,那肯定是增加,因为是插入,如果双亲的双亲原来高度是1,就可能变为2,平衡因子就可能变成-2或2,就是失衡,这种情况就是双亲的双亲插入前平衡因子是-1或1,而必定是-1的时候发生右插入,1的时候发生左插入。

一共就这四种情况,而且都非常简单。平衡发生时,根据平衡的规则(见表1),图3、图4左边部分首先会发生B右旋[$BF_B=1$]或者右旋[$BF_B=-1$]。然后统一到两个图的右边部分再左旋或者右旋。注意,旋转后B将替代A,插入前A的高度显然是1,输入后变成2,平衡后B将替代A,而此时B的高度变成了1。就是说平衡前后高度没有发生变化,那么原来A双亲的高度也不可能发生变化,原来A双亲的双亲的高度也不可能发生变化,依次类推。插入是增加高度(+1),而平衡是减小高度(-1),正好抵消,而且结果也确实这样。最后,得到这样一个结论:如果插入导致失衡,失衡必发生在插入节点的双亲的双亲,且经过一次平衡之后,整个二叉树就是平衡二叉树了。如果插入没有失衡,那么插入节点的双亲的高度必然没有变化,更进一步就是插入后双亲的平衡因子是0。

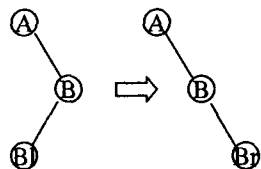


图3 插在双亲的双亲右子树

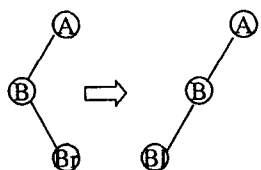


图4 插在双亲的双亲左子树

4 删除

删除操作算法描述:

1)类似查找操作,区别是:如果没有找到节点,说明不存在,删除失败;找到则记录节点的位置和其双亲节点,即是哪个节点的左子树或右子树^[9]。

2)在找到节点之后,如果存在右子树,就要找到它的中序前驱,来替换删除的节点,实际删除发生在那个中序前驱。如果不存在右子树,则直接删除。

3)平衡因子调整,如果是左子树,平衡因子-1,否则平衡因子+1。

4)删除如果导致失衡,那么失衡必定是在实际删

除节点的双亲节点开始平衡,其平衡因子-2或者2,相应的平衡就是左平衡或者右平衡^[10]。

还是先看图5、图6,删除节点前,B的左子树或者右子树必有一个是空,即它的高度必不超过1。删除B节点之后,B的左或者右子树将代替B,也可能是空。此时A平衡因子必定要发生变化,因为它的左或者右子树的高度减少了1。删除前A平衡因子是[-1,0,1],删除后发生失衡,就需要平衡了,所以平衡必开始于A,前面提到,平衡必导致高度减少1,所以平衡了A之后,A的双亲高度减少1,所以A的双亲可能失衡,也需要平衡。依次类推,一直传导到根。是否必定传导到根吗?未必,因为前面提到的操作都是可能,那反过来想一想,什么时候不可能,不失衡。显然要使 H_{A_L} 或者 H_{A_R} 减少1之后还不失衡,只要 H_A 不变,那么 H_A 决定于它的子节点高度减少的兄弟。更进一步可以看到平衡前 H_{A_L} 和 H_{A_R} 必相等,这样平衡后的 H_A 高度不变,此时 BF_A 必非0。

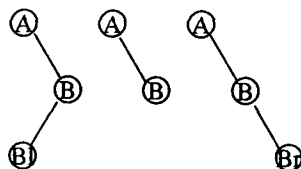


图5 删除节点在双亲的双亲右子树

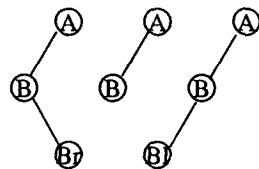


图6 删除节点在双亲的双亲左子树

5 结束语

现有查找算法中二分法查找效率很高为 $O(\log_n)$,但是二分法查找必须要求表中节点按关键字有序排列,而且不能使用链表作存储结构,因此,当插入或删除操作频繁发生时,为了维护表的有序排列,必然要移动很多节点。这样移动节点的操作引起的额外时间开销,就会影响二分法查找效率。所以,二分法查找一般在静态数据结构中使用,若要对动态数据结构进行高效率的查找,就很可能使用到AVL这样的平衡二叉排序树。查找效率几乎和二分法查找一致均为 $O(\log_n)$,插入最多只需一次平衡,其中最多包括两次旋转,删除最多 \log_n 次平衡,其中最多包括两倍 \log_n 旋转。

参考文献:

- [1] Kruse R L, Ryba A J. 数据结构与程序设计——C++语言
(下转第31页)

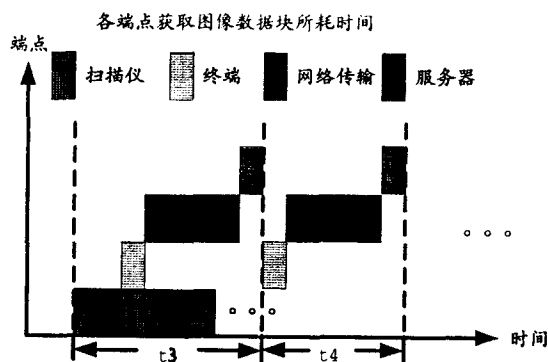


图5 不稳定网络下各端点获取图像数据块所耗时间

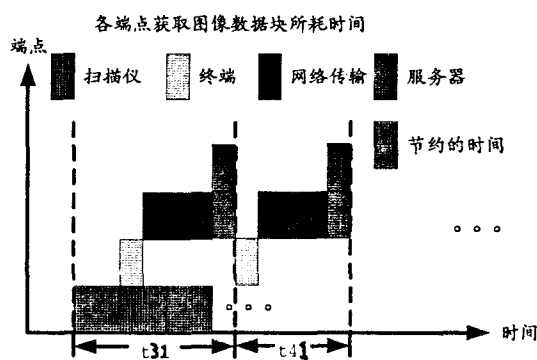


图6 压缩模式下各端点获取图像数据块所耗时间
 t_4 , 总的传输时间为:

$$T_4 = t_{31} + \underbrace{t_{41} + \dots + t_{41}}_{(N-1) \text{ 个 } t_{41}, N \text{ 为总图像数据块数}} \quad (4)$$

由上图可看出,在网络传输过程中,多节约了一个时间片,因此 $T_4 < T_3$,在恶劣的环境下,此模式是最优的。用户可以根据当前的网络状况选择是否需要进行图像压缩处理。

3 结束语

经过这些优化,整体的设计及实现已经能够很好地解决现实中的图像数据传输问题,满足用户需要。

目前该技术已成功应用于某省级移动公司。

其实还有改进的地方,亦有替代方案。由于时间及个人技术水平的有限,最后仅给出几个思路供参考。

可以更进一步的智能判断网络质量,在网络质量差的时候,使用压缩传输,压缩比可以进一步进行根据网络质量的好坏来进行。

同时,在内存分配上,目前使用的是分别对每一个图像数据块分配相应的空间,这样难免会增加系统的一些负荷,可以寻找更好的解决办法改进这一内存分配方案,减少系统开销。

针对扫描仪接口,同样可以实现一种 USB 映射方案,亦即将终端的 USB 接口映射到服务器端,这样就可以方便地使用各种 USB 接口设备,包括扫描仪、U 盘等设备。这种方式的好处就是可以无需为每一款 USB 设备开发特定的接口,从而大大减少了工作量,提高整体的实现效率。

参考文献:

- [1] 蔚江. 浅析 Windows 终端系统[J]. 科技情报开发与经济, 2006(15): 91-93.
- [2] 沈士根, 叶利华, 乐光学. 基于 RDP 协议的远程接入平台设计与实现[J]. 微电子学与计算机, 2008(3): 55-57.
- [3] Microsoft. Windows CE 程序设计[M]. [s.l.]: [s.n.], 1999: 317-339, 451-490.
- [4] Microsoft. Microsoft Windows CE Communications Guide 通信指南[M]. [s.l.]: [s.n.], 1999: 66-149.
- [5] 林涛. 嵌入式操作系统 Windows CE 的研究[J]. 微机计算机信息, 2006(17): 91-93.
- [6] 凌有铸, 徐晓光, 潘伟. 基于 WinCE 的嵌入式远程实时监控[J]. 计算机技术与发展, 2007, 17(7): 204-206.
- [7] Windows CE. NET 系统分析及实验教程[M]. 北京: 机械工业出版社, 2003: 255-263.
- [8] 张允刚, 刘常春, 刘伟, 等. 基于 Socket 和多线程的远程监控系统[J]. 控制工程, 2006(2): 175-177.

(上接第 27 页)

- 描述(影印版)[M]. 北京: 高等教育出版社, 2001.
- [2] Weiss M A. 数据结构与算法分析 C++ 描述[M]. 第 2 版. 英文影印版. 北京: 清华大学出版社, 2002.
- [3] Ford W, Topp W. 数据结构 C++ 语言描述——应用标准模板库(STL)[M]. 第 2 版. 北京: 清华大学出版社, 2003.
- [4] Cormen T H, Leiserson C E, Rivest R L, et al. 算法导论[M]. 北京: 机械工业出版社, 2006.
- [5] SAHNI S. 数据结构、算法与应用——C++ 语言描述[M]. 北京: 机械工业出版社, 2004.
- [6] Sedgewick R. 算法: C 语言实现(第 1~4 部分): 基础知识、数据结构、排序及搜索[M]. 英文影印版第 3 版. 北京: 机械工业出版社, 2006.
- [7] Collins W J. 数据结构与 STL[M]. 北京: 机械工业出版社, 2004.
- [8] Carrano F M, Prichard J J. 数据结构与 C++ 高级教程[M]. 第 3 版. 北京: 清华大学出版社, 2004.
- [9] Sahni S. 数据结构、算法与应用——C++ 语言描述[M]. 第 2 版. 北京: 机械工业出版社, 2006.
- [10] Gurari E, Autumn. CIS 680: DATA STRUCTURES[EB/OL]. 1999. <http://www.cse.ohio-state.edu/~gurari/course/cis680/cis680Ch10.html#QQ1-42-71>.