紧凑(a, b) 树

张希　李万学

(成都科技大学计算机系软件研究室)

A COMPACT (a, b) TREE

Zhang Xi and Li Wanxue

(*Chengdu University of Science and Technology*)

ABSTRACT

In this paper, a new type of multiway height balance trees-Compact (a, b) tree is presented. In such a tree, the nearly optimal storage utilization and nearly optimal height are guaranted by intruducing a constrain between number of children and gradchildren with a Reorganize operation. It's seach opertion is quicker than the B^*-trees and B-trees. It's update operations (insert, delete) are the same performance as B-trees on the amortized computational complexity.

摘　要

本文提出一种新的多叉树——紧凑(a, b) 树。它通过一种整编操作对树中内结点的儿子和孙子个数之间建立制约关系。在元素个数$n \to \infty$，树结点的最大儿子个数$b >> 4$时，使树在最坏情况下的高度和空间利用率都接近最优。它的查找运算比B 类树都快，它的更新运算(插入和删除) 在折算意义下，即在以整个运算序列的最坏时间为代价下，与B 类树的性能相同。

§1. 引　言

以B 树为代表的多叉高平衡树是一种有效的数据结构。多叉高平衡树的高度是影响树上各种运算效率的重要因素，降低树的高度是多叉高平衡的主要改进和发展方向之一。同时，多叉树目前对存贮空间的利用率也不太理想，例如B 树的最坏情况是50%；B^* 树是2/3；弱B 树，又称(a, b) 树(其中$b \geq 2a$)，最坏情况更差，只有$a/b \leq 1/2$。所以改进存贮空间利用率是多叉高平衡树的另一个改进和发展方向。高度和空间利用率是多叉高平衡树最重要的静态特性。本文所述通过整编和移出操作得到的紧凑(a, b) 树，使这两种静态特性在$n \to \infty$、$b >> 4$ 时最坏情况接近最优。同时在更新时使插入或删除运算在折算

1989 年7 月8 日收到，1990 年3 月26 日定稿。本文是国家自然科学基金资助项目。

复杂性意义下与其它B类树有相同的性能。折算复杂性是求在最坏情况下整个运算序列执行时间的平均值。

§2. 紧凑(a, b)树定义与结点结构

定义　设a, b是正整数，且$a \geq 2$, $b \geq 2a$, v是树T中的结点，$s(v)$, $gs(v)$分别是v的儿子和孙子个数。T是一棵紧凑(a, b)树(简称C (a, b)树), 如果下列平衡条件同时成立:

i) 所有叶子在同一层

ii) 内结点v必须满足: $s(v) \leq b$

iii) 所有非根内结点v必须满足: $s(v) \geq a$

iv) 根r必须满足: $2 \leq s(r) \leq b$

v) 除叶子和底层内结点之外的所有结点v必须满足: $s(v) < 1 + (gs(v) + 1)/b$

定义中前四条是(a, b)树(即弱B树)的定义[7]。条件v)引入了儿子和孙子关系限制，要求树中对确定的孙子个数应保证有尽可能少的儿子个数。通过这条限制可以使树中内结点个数尽量少，从而达到提高空间利用率，降低树高的目的。

C (a, b)树所组织的集合元素都存放在叶子上，内结点中仅包含起分隔作用的码值和指向其儿子的指针，以及一个反映子孙个数关系的计数器(如图1)。结点中的码值是按升序从左到右排列。该结点第i棵子树(p_i所指)上的所有码值$> k_{i-1}, \leq k_i$。计数器counter的值: $counter(v) = gs(v) - (s(v) - 1)b + 1$, 显然，当$counter > 0$，条件v)成立；否则平衡被破坏，需要重平衡。另外底层内结点无孙子，它的计数器值无定义，可作它用，例如用作指向同层右邻结点的指针，以便实现区域搜索。图2是一棵C (a, b)树的例子。在有序集上，以其元素作为码值一次生成有n个叶子的紧凑(a, b)树可在O(n)时间完成[13]。

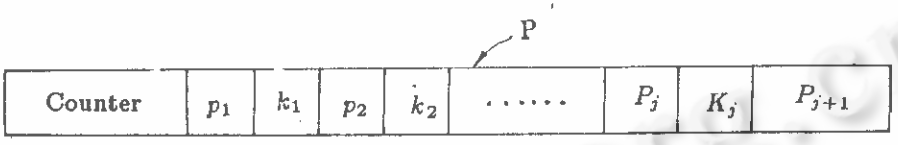

图1　一个有j+1个儿子的内结点，其中K_i是码值，P_i是指向该结点的第i个儿子的指针，counter是计数器

§3. C (a, b)树上各运算的实现算法

3.1 查找(Search)

C (a, b)树上的查找算法与B树的查找完全相同，区域搜索则与B^+树上的实现一致。它们都与结点中的counter值无关，而且不改变C (a, b)树的结构。

3.2 插入和删除(Insert and Delete)

当在C (a, b)树上执行插入或删除时，由于增加或减少了叶子，这就可能使树的平衡被破坏，例如上溢(结点儿子个数为b+1)、下溢(结点儿子个数为a-1)和counter ≤ 0，这就需要对C (a, b)树进行重平衡。在插入和删除中，我们把重平衡分解成以下操作: 分裂、移出、分享和整编。

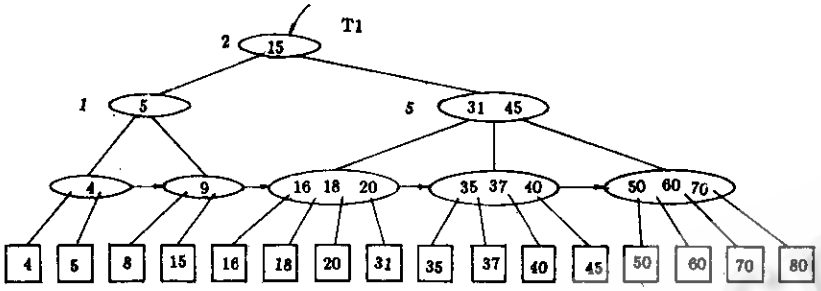

图2　一棵高度为3的C (2, 4) 树T_1, 其中椭圆框表示内结点, 方框表示叶子. 框外值表示计数器值

· 分裂Split (p, Newnode, Newkey): 上溢结点p 分裂出一个有$\lfloor (b+1)/2 \rfloor$ 个儿子的结点Newnode, p 留下$\lceil (b+1)/2 \rceil$ 个儿子, 同时把p↑ .key$[\lfloor (b+1)/2 \rfloor]$ 送入Newkey, 它将成为p和Newnode 在父点中的分隔元, 如图3。显然两个分裂点在分裂后counter 值只会非严格地上升。

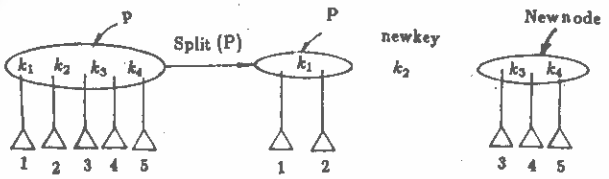

图3　在一个C(2, 4) 树上溢结点p 上执行分裂, 图中三角形代表子树

· 移出Flow (p): 把上溢结点的最左或最右儿子依次移到一个未满的兄弟上去, 移动时保持各兄弟和父点中码值的有序关系, 同时修改这些兄弟的counter, 如图4。移出时可能导致counter≤ 0。

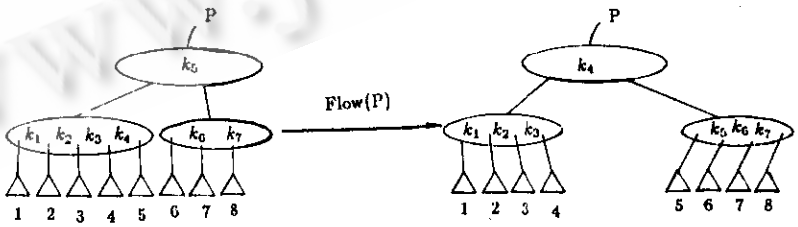

图4　在一个C (2, 4) 树上的上溢结点的移出

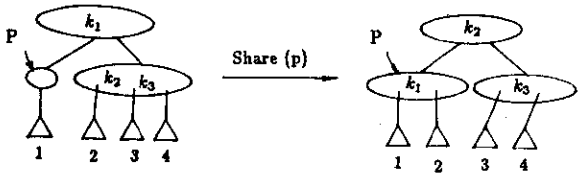

图5　在C (2, 4) 树中对Counter ≤ 0 的结点进行整编

· **分享Share (p)**: 从下溢结点p的兄弟处分享一个儿子过来，修改p及其兄弟的counter值，p及其兄弟的counter值可能下降，甚至≤ 0，如图5。

· **整编Reorganize (p)**: 凡是counter ≤ 0 的点p均用它使儿子数减1，(为使底层内点counter位置易修改，减去中间儿子为宜)，并把p的孙子们均分给余下的儿子，并修改p和儿子们的counter，他们有可能又变为≤ 0，如图6。

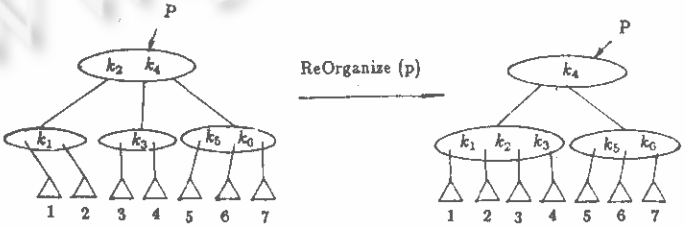

图6　在C (2, 4) 树上对一个下溢结点的“分享”

引理3.1　对一个结点p整编后，若p的counter> 0，则p的儿子中没有上溢点或下溢点。

证明: 整编前counter≤ 0保证了无上溢点，整编后counter> 0保证无下溢点(详证见[13])。

引理3.2　在执行“分享”时被分享的兄弟结点的儿子个数大于a的充分条件是其父的counter> 0。

证明: 显然(详证见[13])。

在上述四个重平衡操作中，分裂的执行可能导致其父点进一步分裂，最坏情况下分裂可能从叶子执行到根。移出和分享都是只执行一次。整编操作的每次执行都使树中内结点个数严格减1，使树得到“紧缩”，但是整编执行之后可导致其父点及儿子的counter≤ 0，从而引起进一步整编。但它是有限的，因为内结点个数是有限的。我们把对counter≤ 0的重平衡从插入和删除算法中独立出来，这部分处理称为“紧缩”(Condense)。

我们用自然语言叙述插入和删除算法，再用类pascal语言描述紧缩算法。算法中的X为要插入或删去的记录; $k(x)$为记录的码值; t, r为指向根的指针; p为当前结点的指针。pf指向p的父点。

算法3.1　插入(Insert)算法

I1，从根查到叶子，找到插入点的路径。

I2，若需要插入的记录x 在叶子中，则插入失败，返回；否则执行I3。

I3，产生一个新叶子存放x，并以$k(x)$ 为码值向路径的底层内结点插入。

I4，若插入点p 的儿子数≤ b，则修改其父点的counter，返回；否则执行I5。

I5，若p 是根r，则将p 分裂成两个结点p 和Newnode，由newkey 产生的新根r′ 是p 和newnode 之父，返回。

I6，检查p 的所有兄弟是否有未满者(即p 的父点counter ≤ b)？有则执行I7；否则执行I8。

I7，对p 执行移出，condense (pf)，返回。

I8，分裂p，以newkey 向p 的父点执行新的插入，p: =pf；执行I4。

此算法稍加修改便可实现对空树的插入。

算法3.2　删除(Delete) 算法

$D1$，从根查到叶子，找到被删点的路径。

$D2$，若需要删除的记录不在叶子内，则删除失败，返回；否则执行$D3$。

$D3$，将该叶子删去，并从其父点p 中删去$k(x)$。

$D4$，若被删点p 是根r，则执行$D5$，否则执行$D6$。

$D5$，若r 的儿子个数> 1，返回；否则删除r，r 的儿子成为新根，返回。

$D6$，修改pf 的counter，若$counter \leq 0$ 则执行紧缩Condense (pf)，返回；否则执行$D7$。

$D7$，若p 的儿子个数≥ a，则返回；否则执行$D8$。

$D8$，对p 执行分享；Condense (pf)；返回。

此算法稍加修改便可实现单点树的删除。

现在我们考虑紧缩算法Condense，首先在算法中要检查入口处结点及其儿子结点的counter 是否≤ 0，是则就进行整编。整编后可能导致整编点的父点和儿子的Counter ≤ 0，而且还可能导致结点下溢，所以需要再次整编及分享以便使$C(a,b)$ 树恢复平衡。算法是递归的。其中主要变量p 为当前紧缩点，L 为保存待整编结点的指针数组。

算法3.3　紧缩算法(Condense)

```
PROCEDURE condense (Var t, p: treenodepointer);
        Var n, i: integer;
          L: Array [1.. b] of treenodepointer;
begin
    if p 是底层内结点 then return;
    if p ↑. counter ≤ 0 then Reorganize (p);
    n: =0;
    for i: =1 to p ↑. son-number do
       if p ↑. son [i] ↑. counter ≤ 0 then
         begin n: =n+1; L [i]: =p↑. son [i] end;
    if p 是根 then
      begin if p ↑. son-number=1 then 删去归根，其儿子成为新根 end
    else (* p 不是根 *)
      if p ↑. father ↑. counter ≤ 0 then condense (p ↑ . father)
      else
       if p ↑. son-number <a then (* p 是下溢点并需分享 *)
         begin share (p); condense (p ↑ . father) end;
```

```
    while n > 0 do
      begin n:=n-1;
        if L [n+1] ↑. counter ≤ 0 then condense (L [n + 1])
      end;
end;
```

图7和图8给出删除和插入后的变化示意图。

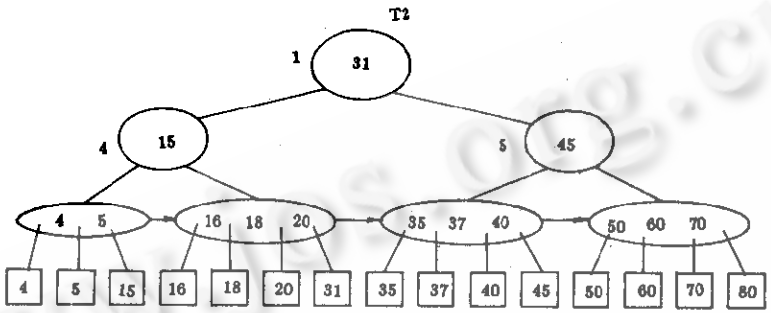

图7 从图2所示的C (2, 4) 树T_1中删除8后得到的结果树。删除后的重平衡操作有“整编”和“分享”

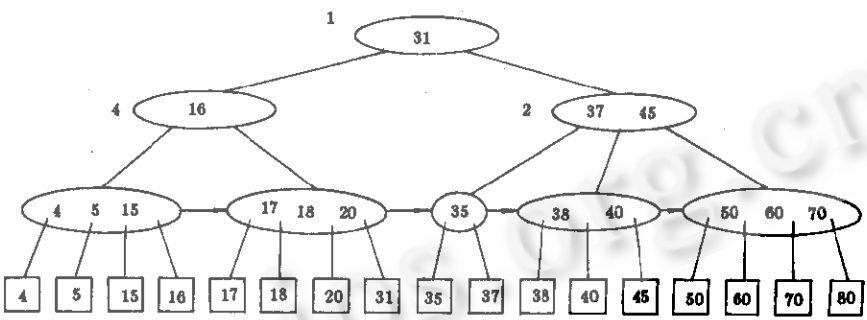

图8 从图7所示的T_2中先后插入17和38后的C(2, 4)树，插入17后导致一次“移出”，插入38后导致一次“分裂”

定理3.1　在C (a, b) 树上插入或删除一个记录后，该树仍然是C (a, b) 树。

证明: 因为紧缩阶段保证平衡条件V)，而且它可以在有限步内结束。由引理3.1和3.2可知，整编、分裂和分享保证了条件ii) 和iii)，插入和删除算法本身也保证了其它条件，所以得到的树仍然是C (a, b) 树。

§4. 性能分析

4.1 C (a, b) 树的静态特性

定理4.1　对于一棵叶子数为n 的C (a, b) 树T 的高度至多为$\log_{br} n + \log_{br} d$，其中

$$d = \begin{cases} b(r-r')/(2-br') & b > 4 \\ 1 & b = 4 \end{cases}$$

$$r = (1+\sqrt{1-4/b})/2 \qquad r' = (1-\sqrt{1-4/b})/2$$

证明：当C (a, b) 树中每个结点的Counter=1，并且根结点只有2 个儿子时，树中叶子个数给定，内结点的个数最多，树的高度最坏，空间利用率最差，对这种树T:

i) 当$b = 4$ 时，T 是二叉平衡时显然高度为$\log_2 n$。

ii) 当$b > 4$ 时，令$S(i)$ 为T 中第i 层结点的个数，由于一个结点的儿子与孙子个数s、gs 之间满足$gs \geq (s-1)b$，所以在最坏情况下得到:

$$\begin{cases} s(1) = 1, \quad s(2) = 2 \\ s(i) = (s(i-1) - s(i-2)) * b \quad (i > 2) \end{cases}$$

用生成函数求解此递归方程。我们得到$S(i) = A(br)^{i-1} + B(br')^{i-1}$，其中$A = (2 - br')/(b(r-r'))$, $B = (br-2)/(b(r-r'))$，$r = (1+\sqrt{1-4/b})/2$, $r' = (1-\sqrt{1-4/b})/2$。

考虑到高度为h 的树T 只有$h+1$ 层，所以叶子数$n = A(br)^h + B(br')^h$，又因为$b > 4$ 时，$A > 0$, $B > 0$，令$d = 1/A \geq 1$, 且$br > 2$，所以

$$h = \log_{br}((n - B(br')^h/A) \leq \log_{br}(n/A) = \log_{br} n + \log_{br} d$$

所以命题成立。

当C (a, b) 树中叶子个数$n \to \infty$ 时，定理4.1 高度公式中的常数项可以忽略，即$h \approx \log_{br}(n)$，当b 增大时，r 会迅速接近1(例如$b = 8, 11, 20, 50$ 时，$r \approx 0.85$，0.9, 0.95, 0.98)，即$h \approx \log_b n$，所以，当$n \to \infty$，$b >> 4$ 时，C (a, b) 树有近似最优的高度。

定理4.2　当$n \to \infty$，C(a, b) 树最坏情况空间利用率至少为$r(= (1+\sqrt{1-4/b})/2)$。

证明：在定理4.1 中高度最坏的树其空间利用率也最差。令$m(i)$ 表示有i 层的树的结点总数$m(i) = m(i-1) + s(i)$，我们用生成函数求解此递归方程容易得到$m(i) = D + E(br)^{i-1} + F(br')^{i-1}$，则高度为$h$ 的这种树的结点总数$m(h+1)$ 为

$$m(h+1) = D + E(br)^h + F(br')^h$$

其中$D = 3 - b$, $E = (1 - r'(r-2))/(r-r')$, $F = (r(b-2)-1)/(r-r')$。显然$D < 0$, $E > 0$, $F > 0$。而T 中指针个数为T 中除根以外的结点个数(包括叶子)，即$m(h+1) - 1$。T 中内结点个数为$m(h)$，所以在最坏情况下$C(a, b)$ 树的空间利用率为

$$U = \frac{T\text{中内结点中指针个数}}{(T\text{中内结点个数})(\text{内结点体积})} = \frac{m(h+1)-1}{m(h)\cdot b}$$

$$= \frac{D/(b^h r^{h-1}) + Er + Fr'^h/r^{h-1} - 1/(b^h r^{h-1})}{D/(b^{h-1}r^{h-1}) + E + Fr'^{h-1}/r^{h-1}}$$

当$n \to \infty$ 时，$h \to \infty$，若$b = 4$，$r = r'$，此时$U = \dfrac{0 + Er + Fr' - 0}{0 + E + F} = r$，若$b > 4$，则$r > r'$，此时$U = (0 + Er + 0 - 0)/(0 + E + 0) = r$。结论成立。

因为如前所述随差b 增大，r 迅速接近1，所以当$b >> 4$ 时，$r \to 1$，即空间利用率也近似最优。

4.2 运算的折算分析

我们用于分析运算效率的度量标准是折算时间复杂性(Amortized Computational Complexity)[10]。它是求在最坏情况下一个运算序列的平均执行时间。它与最坏时间的不同在于：折算时间不像最坏时间那样以单个运算为研究对象，它求取的是整个运算序列的最坏时间。在对相互影响和关联的运算序列进行分析时，折算时间比最坏时间，甚至比平均时间更接近实际情况[3,9]，因为一个运算执行后往往可能改善数据结构使后继运算代价降低。另外，折算时间的分析不依赖于对运算概率分布的假设，因而比平均时间更具牢固性。

折算分析的常用方法是建立一个数据结构D 到整数空间映射的势函数$\Phi(D)$。对运算序列$\theta_1, \theta_2, \cdots, \theta_m$，每个运算$\theta_i (i = 1, 2, \cdots, m)$ 执行之后，数据结构D_{i-1} 变为D_i，势函数的改变$\Delta\Phi_i = \Phi(D_i) - \Phi(D_{i-1})$，我们定义$\theta_i$ 的折算时间为$a_i = t_i + \Delta\Phi_i$，其中t_i 为完成θ_i 所需最大实际时间。运算序列执行之后$\sum a_i = \sum t_i + \sum \Delta\Phi_i$，即$\sum t_i = \sum a_i - \sum \Delta\Phi_i$ $= \sum a_i - (\Phi_m - \Phi_0)$。如果定义的势函数$\Phi_0 = 0$，$\Phi_n = \Phi(D_m) > 0$，则$\sum t_i \leq \sum a_i$，即运算序列的最大实际执行时间以各运算折算时间之和为上界。

以下分析中时间单位是访问一个结点的代价，对结点内的处理代价(不超过$O(b)$) 忽略不计。

4.2.1 查找运算和区域搜索均不改变树的结构，因此一个查找运算序列的折算时间就是查找运算的最坏时间，不超过$\log_{br} n + \log_{br} d = O(\log n)$，区域搜索的时间则要加上区域长度$K$，为$O(k + \log n)$。

4.2.2 在插入和删除的折算分析中，我们考虑长度为m 的运算序列由i 个插入和d 个删除组成$(m = i + d, i \geq d)$。开始树为空。首先对重平衡操作进行折算分析。分析中以一次插入或删除中各重衡操作的发生次数作为它们的代价。

对C (a, b) 树T 的势函数定义如下：每个结点v 的势$\Phi(v)$ 为v 的儿子数。T 的势$\Phi(T)$ 为T 中所有结点v 的势之和$\sum \Phi(v)$。显然初始树T_0 的势$\Phi(T_0) = 0$，在长为m 的运算序列执行之后，结果树有N 个点(包括叶子)，则$\Phi_m(T) = N - 1$。

下面就不同的操作分别讨论执行后对势的影响：

(1) 加入叶子(ADD)：只在底层内结点儿子数加1，所以$\Delta\Phi_{ADD}(T) = 1$。

(2) 删除叶子(prun)：底层内点儿子数减1，$\Delta\Phi_{prun}(T) = -1$。

(3) 移出(Flow) 和分享(share)：树中无结点个数变化，所以$\Delta\Phi_{Flow}(T) = \Delta\Phi_{share}(T) = 0$。

(4) 分裂(split)：分两种情况

i) 若分裂点是根r：有$b + 1$ 个儿子的r，分裂出一个新根r' 作为只有$\lceil (b + 1)/2 \rceil$ 个儿子的点V_1 和$\lfloor (b + 1)/2 \rfloor$ 个儿子的点v_2 的父点，所以$\Delta\Phi_{sp}(T) = \Phi(r') + \Phi(v_1) + \Phi(v_2) - \Phi(r)$ $= 2 + (\lceil (b + 1)/2 \rceil + \lfloor (b + 1)/2 \rfloor) - (b + 1) = 2$。

ii) 若分裂点p 不是根，p 分裂成有$\lceil (b + 1)/2 \rceil$ 个儿子和$\lfloor (b + 1)/2 \rfloor$ 个儿子的两个点，

中间码值上升到父点pf 中，pf 的儿子数加1，其它结点无变化，故$\Delta\Psi_{sp}(T)=1$。

所以对分裂操作：$1\le\Delta\Phi_{sp}(T)\le 2$。

(5) 整编(Reorganize)：也分两种情况

i) 若整编点是根r 且只有两个儿子时，删除根点和一个儿子，以另一个儿子为新根，所以$\Delta\Phi_{Reo}(T)=-2$。

ii) 若整编点p 不是根或是根但儿子个数> 2，则只减少该点的一个儿子并分摊孙子，其它点不变，所以$\Delta\Phi_{Reo}(T)=-1$。

所以，对整编操作$-2\le\Delta\Phi_{Reo}(T)\le -1$。

我们令FL、SP、SH 和Reo 分别表示移出、分裂、分享和整编在整个运算序列下的执行次数，序列执行后势的变化：

$$\begin{aligned}\Delta\Phi(T)=&i*\Delta\Phi_{ADD}(T)+d*\Phi_{prun}(T)+FL*\Delta\Phi_{Flow}+SH*\Delta\Phi_{shar}+SP*\Delta\Phi_{sp}(T)\\&+Reo*\Delta\Phi_{Reo}(T)=i-d+sp*\Delta\Phi_{sp}(T)+Reo*\Delta\Phi_{Reo}(T)\\=&n+SP*\Delta\Phi_{sp}(T)+Reo*\Delta\Phi_{Reo}(T)\end{aligned}$$

其中n 为运算序列执行后树上叶子点数。设N 为结点总数，则$\Delta\Phi(T)=\Phi_m(T)-\Phi_0(T)=N-1$，所以

$$-Reo*\Delta\Phi_{Reo}(T)=SP*\Delta\Phi_{sp}(T)+n+1-N$$

因为$n+1-N\le 0$，$1\le -\Delta\Phi_{Reo}(T)\le 2$, 所以

$$Reo\le\frac{SP*\Delta\Phi_{sp}(T)}{-\Delta\Phi_{Reo}(T)}\le 2SP$$

因为在一次插入中最坏情况的分裂次数$=O(\log n)$，所以，对整个运算序列$SP=O(i\log n)=O(m\log n)$，由此得到$Reo=O(m\log n)$。

分享操作除在可能由删除叶子引起以外，在紧缩阶段的分享都是由整编操作使结点减少所致，所以$SH=O(Reo)=O(m\log n)$。另外，每次插入至多可能发生一次移出，所以$FI=O(m)$。

引理4.1　一个长度为m 的插入删除序列在初始为空的C (a, b) 树上执行，其中分裂、整编和分享的折算时间不超过$O(\log n)$，移出的折算时间为$O(1)$。

证明：由上述讨论得出。

定理4.3　一个插入删除运算序列在初始为空的C (a, b) 树上执行，其插入和删除运算的折算时间都为$O(\log n)$，n 为执行运算序列后树的叶子数。

证明：插入和删除算法中找到目标的代价与查找相同，都是$O(\log n)$。重平衡部分代价主要指分裂和紧缩的代价，其中紧缩每递归一次就要执行一次整编，其它语句的执行都是$O(1)$ 级的，所以紧缩的代价正比于整编代价。由引理4.1 可知插入删除的重平衡代价为$O(\log n)$，所以插入和删除的折算时间都为$O(\log n)$。

§5. C(a, b) 树与其它B 类树的比较

5.1 静态特性比较

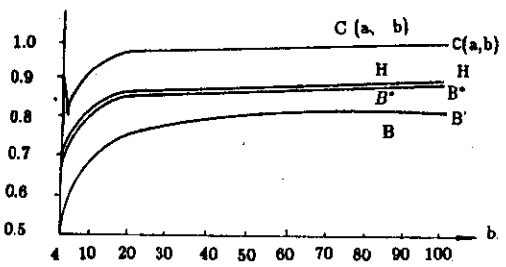

图9　最优多叉树 $\log_b(n)$ 与 B 树、B^* 树、H 树和C(a, b) 的最坏高度之比，在 b 从 4 → 100 的取值的情况

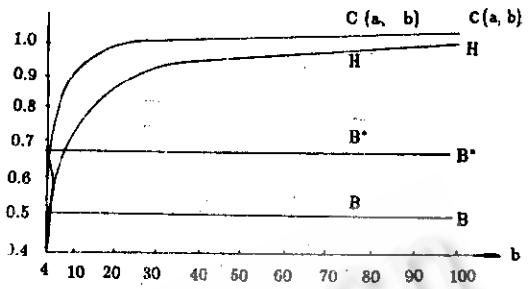

图10 空间利用率(b 从 4 → 100)

我们把各树的静态特性列在表I 中，它们是在 $n \to \infty$，$b >> 4$ 的情况下的比较。同时还把这些特性随 b 增长的变化曲线在图9 和图10 给出。

表I　各类树的最坏高度和空间利用率比较

树种	最坏高度	最坏空间利用率	注记
B 树	$\log_{b/2}(n)$	0.5	m=b
$B+$ 树	$\log_{2b/3}(n)$	2/3	m=b
(a, b) 树	$\log_a(n)$	a/b	$b \geq 2a$
$H(\beta,\gamma,\delta)$ 树	$\lceil 2+2\log_\gamma((n+1)/2\delta)$	$\delta/(\beta+1)$	$\beta=b, \gamma$ 最大孙子数, δ 底层最小儿子数
C (a, b) 树	$\log_{br}(n)+\log_{br}(d)$	r	$r=(1+\sqrt{1-4/b})/2$, $d=b(r-r')/(2-br')$

从表I 中可知C (a, b) 树的高度比其它树小，查找运算的时间与树高成正比，所以C (a, b) 树的查找速度比其它 B 树都快。空间利用率则比其它树更高。

5.2 运算代价的比较

表II 是折算意义下查找、插入和删除比较。

表II　各类树的几种运算时间代价的比较

	查找	插入	插入重平衡	删除	删除重平衡
B [MEH 84]	O (log n)	O (log n)	O (log n)	O (log n)	O (log n)
B*	O (log n)	O (log n)	O (log n)	O (log n)	O (log n)
H [HUA 85]	O (log n)	O (log n)	/	O (log n)	/
(a, b) [MEH 84]	O (log n)	O (log n)	O (1)	O (log n)	O (1)
C (a, b)	O (log n)	O (log n)	O (log n)	O (log n)	O (log n)

从表II 可知，C (a, b) 树与各类树的动态性能相同，只有(a, b) 树的重平衡时间比各类树更好。

§6. 结束语

大量实验结果说明，在随机插入删除运算序列中，引起整编和分裂的次数远小于移出操作的次数。以b=11，a=2 为例，n 很大时每百个运算的子序列引起的分裂次数约等于整编次数，均在(6, 9) 之内，远小于$n\log n$，而移出次数反而高达(16, 32) 之内，并随叶点数增加而稍有增大的可能。但是从前面分析可以看出，如果从C(a, b) 树的重平衡操作中去掉“移位”，树接近最优的静态特性仍然保持。“移位”只使树结构更加紧凑。由于C(a, b) 树具有很好的静态特性，它还能在O (n) 时间内生成有n 个叶子的C(a, b) 树，所以它是一种很好的静态结构。它的更新运算在折算意义下与各类B 树性能相同，所以用C(a, b) 树作为文件索引组织是一种很理想的选择。

参考文献

[1] Aho, A. Hopcroft, J. and Ullman, J., "Data Structures and Algorithms", Addison-Wesly 1983.

[2] Bayer, R. and McCright, E., "Organization and Maintain of Large Order Index", ACTA Informatica 1. (1972) p173–189.

[3] Bentley, J. L. and McGeoch, C., "Amortized Analysis of Self-Organizing Sequential Search Heuristics", Commun. ACM, Vol. 28, No. 4 (April, 1985).

[4] Culik II, K. Ottmann, Th. and Wood, D., "Dense Multiwaytree", ACM Trans. Database Syst., Vol. 6, No. 3 (Sept. 1981).

[5] Gonnet, G. H., "Handbook of Algorithms and Data Structures", Addison-Wesly, 1984.

[6] Huang, S.S., "Height-Balanced Tree of Order (β, γ, δ)", ACM Trans. Database Syst., Vol. 10, No. 2, 1985.

[7] Mehlhorn, Kurt., "Data Structures and Algorithms 1: Sorting and Searching", Springer-Verlay., 1984.

[8] Rosenberg, A. L. and Snyder. L., "Time and Space Optimality in B-tree", ACM Trans. Database Syst., Vol. 6, No. 3, (March. 1981) p179–183.

[9] Sleator, D. and Tarjan, R. E., "Amortized Efficiency of List Update and Paging Rule", Commun. ACM, Vol. 28, No. 2 (Feb. 1985).

[10] Tarjan, R. E, "Amortized Computational Complexity", SIAM. J. Alg. disc. meth, Vol. 6, No. 2 (April, 1985).

[11] 李万学，“高度近似最优2, 3 树”，科学通报，1985 年，第9 期，P653–657。

[12] 王珊，吴欣畴，“B+ 树效率分析和组织聚集算法”，电子计算机动态，1981 年，第9 期。

[13] 张希，“近似最优(a, b) 树”，成都科技大学计算机系，硕士论文，1989 年4 月。