STUDIA UNIV. BABEŞ-BOLYAI, MATHEMATICA, XXXVII, 3, 1992

EXTENDED B-TREE

F. M. BOIAN*

Received: January 3, 1993
AMS Subject Classification: 68P10, 68P20

REZUMAT. B-arbore extins. In lucrare se prezintă o extindere a conceptului de *B*-arbore. Prin această extindere se permite accesul relativ în acest tip de structură de date. Prin acces relativ se înţelege posibilitatea de deplasare optimală în *B*-arbore peste *n* chei faţă de cheia curentă.

DEFINITIONS. In [3] B-tree was formally defined. We denote by m the order of the B-tree, and we denote by e the number of keys from the current node from B-tree. By p, p_0, p_1, p_2 etc. We denote some pointers to nodes from B-tree. At last, by K, with possible subscripts, we denote value(s) of key(s) from B-tree.

If p is a pointer to a node from B-tree, we denote by $S(p)$ the sub B-tree having the root in the node pointed by p.

DEFINITION 1. *The possession of $S(p)$ is the total number of keys from $S(p)$. We denote this number by $Z(p)$.*

Let $a = K_{i+1}K_{i+2}...K_{i+r}$ be the r succesive keys from the same node of B-tree. Let p_i, p_{i+1}, p_{i+2}, ..., p_{i+r} be the neighbours pointers for the keys from a.

Notations. By $S(a)$ we denote the sub B-tree which has in its root only the keys from a and the descendents $S(p_i)$, $S(p_{i+1})$, $S(p_{i+2})$, ..., $S(p_{i+r})$.

We denote by $Z(a)$ the possession of $S(a)$.

By $| a |$ we denote the number r (the number of keys from a).

---

* *University of Cluj-Napoca, Department of Computer Science, 3400 Cluj-Napoca, Romania*

THEOREM 1. *The following relations (with the above notations), holds:*

$Z(a) = r + Z(p_i) + Z(p_{i+1}) + Z(p_{i+2}) + \ldots + Z(p_{i+r})$

For each j from 1 *to r,*

$Z(a) = Z(K_{i+1} \ldots K_{i+j}) + Z(K_{i+j+1} \ldots K_{i+r}) - Z(p_{i+j})$ *and*

$Z(a) = Z(K_{i+1} \ldots K_{i+j-1}) + 1 + Z(K_{i+j+1} \ldots K_{i+r})$

The *proof* of this theorem immediately follows from the definition of possession.

With these considerations, we continue to define an extended *B*-tree.

DEFINITION 2. *An* ***Extended B-tree*** [1] *is a B-tree having in its nodes the following information:*

z_0p_0	K_1	z_1p_1	K_2	z_2p_2	...	$ze^{-1}pe^{-1}$	Ke	zepe
↓		↓		↓		↓		↓

where $z_i = Z(p_i)$, $i = 0, 1, \ldots, e$

An example. In fig. 1, an extended *B*-tree is presented. In each node, only values of keys are presented. For leafless nodes, there are two arrows near each key: one on the left and the other on the right. On the left of each arrow, in brackets, the value of possession appears, and on the right, the value of the pointer (here is the number of the node) appears.

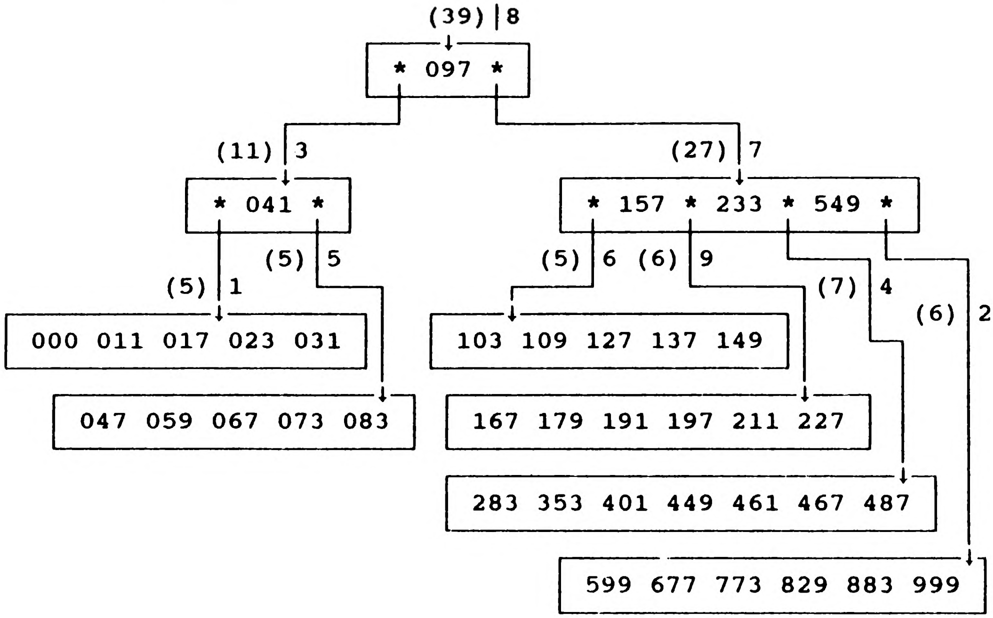
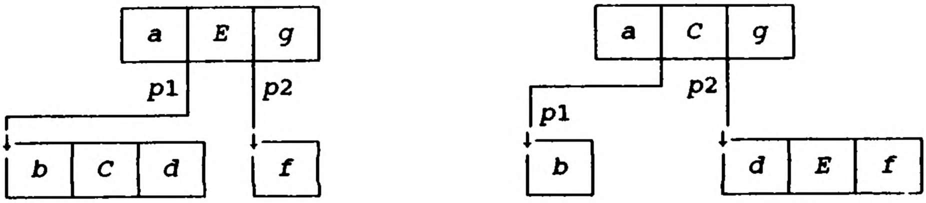


Figure 1. An extended *B*-tree

For *example*, $S(7)$ has the nodes 7, 6, 9, 4, 2, and $Z(7)$ = 27. If a = "157 233", then $S(a)$ is $S(7)$ without the key "549" and without the node 2, and $Z(a) = 20$.

Extended *B*-tree transformation. The operations with *B*-tree are presented in [3]. In [2] and [1] we have described some ideas to implement an extended *B*-tree. In figures 2, 3 and 4 three pairs of transformations are presented: rotate left /right, transform a node into two or reverse, transform two nodes into three or reverse. In these figures, we note by lower case (*a*, *b*, ..., *h*, *i*) the sequences, possibly empty, from consecutive keys (from the same node), and by an uppercase (*C*, *E*, *G*) a key from a node.

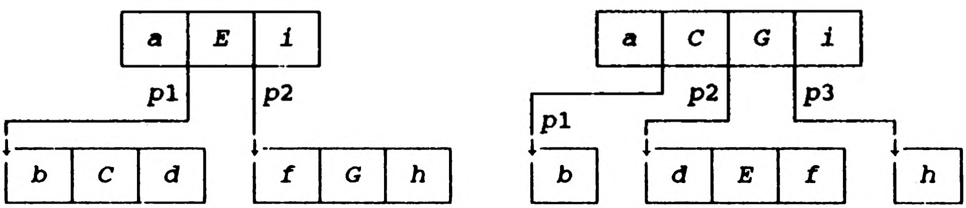


(I) (II)

Figure 2 Rotate left / right

(I) (II)

Figure 3 Transformation between one node - two nodes



(I) (II)

Figure 4 Transformation between two nodes - three nodes

When a transformation is applied, the possessions for new nodes must be computed only from the olds, without to see the others nodes from *B*-tree. In the following, for the fourth usualy transformations, the new possessions are:

1) From (I) to (II) of the fig. 2 (rotate to right):

$Z(p_1) := Z(b);$

$Z(p_2) := Z(d) + 1 + Z(f);$

2) From (II) to (I) of the fig. 2 (rotate to left):

$Z(p_1) := Z(b) + 1 + Z(d);$

$Z(p_2) := Z(f);$

3) From (II) to (I) of the fig. 3 (fusion of two nodes into one):

$Z(p_1) := Z(b) + 1 + Z(d);$

4) From (I) to (II) of the fig. 4 (transformations two nodes into three):

$Z(p_1) := Z(b);$

$Z(p_2) := Z(d) + 1 + Z(f);$

$Z(p_3) := Z(h).$

We have used these four transformatios in [1] for implementation. If only these are used, at most two nodes are necessary for operations with *B*-tree.

When these transformations must be applieds? From [3] these are applied, possibily, after deleting a key or inserting a key, if after that the number of keys from the current node are less than $m / 2$ or great m. If after a deletion, in node remain less than $m / 2$ keys, then this event is called **undersized**. If after a insertion, in the node there are great m keys, then this event is called **overflow**. The following rules are applied, in this order:

1 If ($|bCd| = m+1$ (overflow) and $|f| < m$) or
($|f| = m / 2 - 1$ (undersize) and $|bCd| > m / 2$)
then
b and *d* are choisen so that $|\ |bCd| - |f|\ | \leq 1$
and rotations on the right are applied (see I to II in fig 2).

2 If ($|dEf| = m+1$ (overflow) and $|b| < m$) or
($|b| = m / 2 - 1$ (undersize) and $|dEf| > m / 2$)

then

d and f are choisen so that $| \, |dEf| - |b| \, | \leq 1$

and rotations on the left are applied (see II to I in fig. 2).

3 If undersize and $|b| + |d| < m$

then

two node join into one (see II to I in fig. 3).

4 If overflow and $|bCd| + |fGh| = 2m+1$

then

transform two node into three other, with (approximate) the same numbers of keys: b, d, f and h are choise so that:

$||b|-|dEf|| \leq 1$ and $||dEf|-|h|| \leq 1$ and $||b|-|h|| \leq 1$

(see I to II in fig. 4).

Relative access in extended B-tree. Let K_c (current key) and K_t (target key) two keys from a B-tree. Suppose that between K_c and K_t, in ascendent order, there are other n-1 keys. The problem is to construct an algorithm so that to minimize the number of moves in B-tree to find K_t when K_c is the current key.

Let p be the pointer to the nearest node so that both K_c and K_t can be accessed from it. This node is called **common ancestor** from both keys. Its clear that all the n-1 keys between K_c and K_t are in $S(p)$. Because each other ancestor of K_c and K_t is ancestor for their common ancestor, it results that minimal number of moves is from common ancestor to K_t.

To find common ancestor between current key and any other key, we purpose to create and update a stack. When a key K_c is found, for each ancestor of K_c an record is pushed in this stack. An record from stack has the following structure:

$$(p_i, j_i, zl_i, zr_i, Kl_i, Kr_i)$$

where:

i is the current level in B-tree (the root has the level 1);

p_i is the pointer to the node;

In the following, we suppose that the node p_i has the form:

$$z_{i0}p_{i0} \; K_{i1} \; . \; . \; . \; K_{ij} \; z_{ij}p_{ij} \; . \; . \; . \; K_{ie} \; z_{ie}p_{ie}$$

j_i is the index of the key K_c, if $K_{ij} = K_c$, or K_c is in $S(p_{ij})$, if $K_{ij} \neq K_c$;

$zl_i = Z(K_{i1} \ldots K_{ij}) - Z(p_{ij}) - 1$ (the possession to left of K_{ij});

$zr_i = Z(K_{ij} \ldots K_{ie})$ (the possession to right of K_{ij});

Kl_i is the minimum value from $S(p_i)$;

Kr_i is the maximum value from $S(p_i)$;

For example, in the *B*-tree from fig. 1, if $K_c = 211$, the stack is:

i	*p i*	*j i*	*zl i*	*zr i*	*Kl i*	*Kr i*
1	8	1	11	28	-oo	+oo
2	7	1	5	22	097	+oo
3	9	5	4	2	157	233

The fields of this stack can be completed during the search for a key. All informations for a record are known from the current node or from its father. The last record from stack corresponds to a node having the current key in it. The maximum size of this stack is very small (see [3] for details).

Now, suppose that the current key is K_c and we want to skip over n keys (forward or backward if $n < 0$). For that, we pop from stack until $n \leq zr_i$ when $n \geq 0$, or until $-n \leq zl_i$ when $n < 0$. The p_i from top of stack pointed to common ancestor to K_c and K_t over n keys over K_c.

This stack helps to reduce the nùmber of nodes accessed when looking for a key having a value. For that, its suffices to pop from stack until the value of the new key is between Kl_i and Kr_i. In the most cases, the search a new key begins instead the root with an it's descendant for a same level.

R E F E R E N C E S

1. Boian F. M., *Sistem de fişiere bazat pe B-arbori*, în Lucrările celui de-al VII-lea colocviu naţional de informatică, INFO-IASI, 1989, pp. 33-40.
2. Boian F. M., *Căutare rapidă în B-arbori*, în Lucrările simpozionului "Informatica şi aplicaţiile sale", Zilele academice Clujene, Cluj-Napoca, 1989.
3. Knuth D. E., *Tratat de programarea calculatoarelor*; vol III, Sortare şi căutare. Ed. Tehnică, Bucureşti, 1976.