Acta Informatica 1, 290—306 (1972)

Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms*

R. Bayer

Received January 24, 1972

Summary. A class of binary trees is described for maintaining ordered sets of data. Random insertions, deletions, and retrievals of keys can be done in time proportional to $\log N$ where N is the cardinality of the data-set. Symmetric B-Trees are a modification of B-trees described previously by Bayer and McCreight. This class of trees properly contains the balanced trees.

This paper will describe a further solution to the following well-known problem in information processing: Organize and maintain an index, i.e. an ordered set of keys or virtual addresses, used to access the elements in a set of data, in such a way that random and sequential insertions, deletions, and retrievals can be performed efficiently.

Other solutions to this problem have been described for a one-level store in [1, 3–5, 7] and for a two-level store with a pseudo-random access backup store in [2]. All these techniques use trees to represent the data sets. The class of trees to be described in this paper is a generalization of the trees described in [1, 3–5], but it is not comparable with the BB-trees described in [7]. The following technique is suitable for a one-level store.

Readers familiar with [2] and [3] may recognize the technique as a further modification of B-trees introduced in [2]. In [3] binary B-trees were considered as a special case and a modified representation of the B-trees of [2]. Binary B-trees are derived in a straightforward way from B-trees, they do exhibit, however, an asymmetry in the sense that the left arcs in a binary B-tree must be δ-arcs (downward), whereas the right arcs can be either δ-arcs or ϱ-arcs (horizontal). Removing this asymmetry naturally leads to the symmetric binary B-trees described here.

After this brief digression on the relationship of this paper to earlier work we will now proceed with a self-contained presentation of symmetric binary B-trees.

Notation. We will use t, u, v, w, x, y, z to denote trees and p, q, r, s to denote nodes of trees, usually the root nodes. We assume that "nodes", or to be precise "the values stored at the nodes", are taken from some set K of data elements or "keys" on which a total order, denoted by $<$, is defined. Except in very few

* This work was partially supported by an NSF grant while the author was at Purdue University, Lafayette, Indiana, USA.

cases it is not necessary to distinguish between the nodes of a tree and the keys stored there, the meaning will be clear by context. "$*$" is a special symbol used in describing B-trees. Its presence should convey the intuitive notion of horizontal arcs (ϱ-arcs) to the left and right of a node as opposed to vertical arcs (δ-arcs).

Definition of Symmetric Binary B-Trees

Symmetric binary B-trees, henceforth simply called *B-trees*, are defined recursively as follows:

i) Let e be the empty tree and let Φ be the empty set. Then define $T_\delta(0) = \Phi$, $T_\varrho(0) = \{e\}$, i.e. the set with the single member e.

ii) For all integers $h > 0$ define

$$T_\delta(h) = \{(x, r, y) \mid x, y \in T_{\delta\varrho}(h-1), r \in K\}$$

$$\begin{aligned} T_\varrho(h) = &\{(x, *r, y) \mid x \in T_\delta(h), y \in T_{\delta\varrho}(h-1), r \in K\} \\ &\cup \{(x, r*, y) \mid x \in T_{\delta\varrho}(h-1), y \in T_\delta(h), r \in K\} \\ &\cup \{(x, *r*, y) \mid x, y \in T_\delta(h), r \in K\} \end{aligned}$$

$$T_{\delta\varrho}(h) = T_\delta(h) \cup T_\varrho(h).$$

iii) A B-tree is a member of $T_{\delta\varrho}(h)$ for some integer $h \geqq 0$.

Note. We call x the left subtree, y the right subtree, r the root, and h the δ-level or δ-height of a B-tree in $T_{\delta\varrho}(h)$. We also say that the node r is at the δ-level or δ-height h.

For the purpose of this paper we will use the following list data structure to represent B-trees: Pointers (or arcs) attached to the root of a tree will point to its subtrees, a left pointer to the left subtree and a right pointer to the right subtree. We consider a node and the two attached pointers as a group of physically adjacent data items, also called an "entry". The presence of the special symbols $*$ to the left of r and of $*$ to the right of r shall be represented by left and right ϱ-pointers resp., their absence by δ-pointers. In graphical representations of B-trees ϱ-pointers appear as horizontal, δ-pointers as downward arcs. In the computer implementation of B-trees, one bit is used to distinguish ϱ-pointers (1 bit) from δ-pointers (0 bit). Empty trees and pointers to them are omitted.

A less formal, but intuitively more appealing definition of B-trees using the terminology of [6] and ignoring empty B-trees is the following:

B-trees are directed binary trees with two kinds of arcs (pointers), namely δ-arcs (downward or vertical pointers) and ϱ-arcs (horizontal pointers) such that:

i) The paths from the root to every leaf all have the same number of δ-arcs.

ii) All nodes except those at the lowest δ-level have 2 sons.

iii) Some of the arcs may be ϱ-arcs, but there may be no successive ϱ-arcs.

In addition, the keys shall be stored at the nodes of a B-tree in such a way that postorder traversal [6] of the tree yields the keys in increasing order, where postorder traversal is defined recursively as follows:

1. If the tree is empty, do nothing.
2. Traverse left subtree.
3. Visit root.
4. Traverse right subtree.

Fig. 1. shows a graphical representation of a B-tree. Readers familiar with balanced trees [1, 4, 5] should observe that B-trees are not balanced trees as shown by the B-tree in Fig. 1.

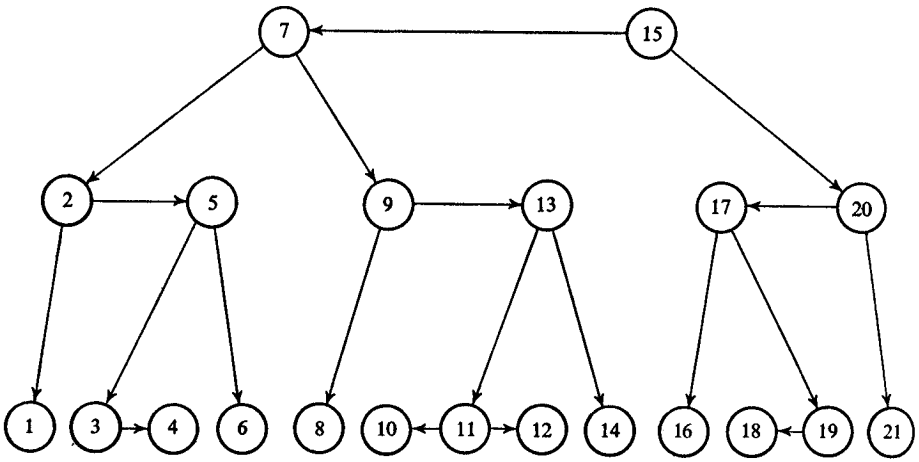


Fig. 1. Example of a symmetric binary B-tree

Number of Nodes and Height of a B-Tree

Let the height k of a B-tree be the maximal number of nodes in any path from the root to a leaf. Note that the height k is larger than the δ-height h whenever the tree has ϱ-pointers, in particular for a given tree we have:

$$h \leqq k \leqq 2h.$$

The height k of a B-tree is, as we shall see, related to the amount of work necessary in the worst cases to insert, retrieve, and delete keys in the tree. To obtain bounds on k we need the following theorem which characterizes $T_{\min}(k)$, the class of those B-trees of a given height k with the least number of nodes. We state the theorem and sketch a proof using largely the terminology of the graphical representation of B-trees.

Theorem 1. i) $T_{\min}(0) = \{e\}$; $T_{\min}(1) = T_\delta(1)$.

ii) $t \in T_{\min}(k)$; $k > 1$ iff

a) there is exactly one longest path, say λ.

b) λ ends with a ϱ-pointer; in λ ϱ-pointers and δ-pointers alternate.

c) t contains no ϱ-pointers except those in λ.

To prove the theorem we need the following lemmas:

Lemma 1. Every longest path in $t \in T_{\min}(k)$, $k > 1$, ends with a ϱ-pointer.

Proof. If there is a longest path λ without this property, then attach a new node via a ϱ-pointer at the end of λ, delete from t the root-node and the subtree of the root not containing λ. This results in a B-tree t' of the same height as t, but with fewer nodes, a contradiction.

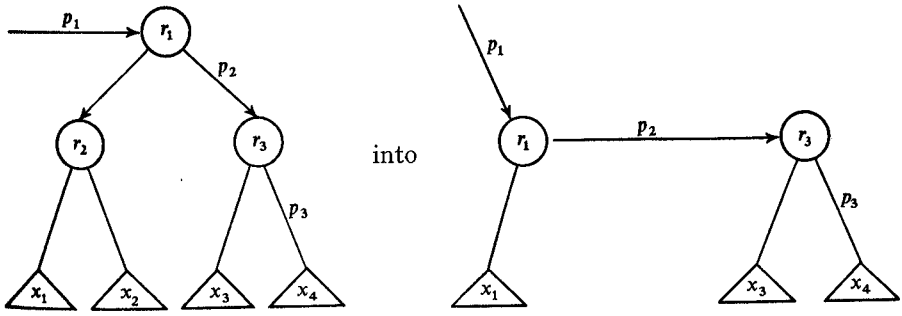
Lemma 2. If $t \in T_{\min}(k)$, $k > 1$, and t has a longest path λ, then every ϱ-pointer in t is in λ.

Proof. Assume there is a ϱ-pointer p not in λ. p points to some subtree $u \in T_\delta(m)$ for some m. Let u be of the form (u_1, r_1, u_2). Replace u by u_1, and p by a δ-pointer to u_1. This results in a B-tree t' of height k, but with fewer nodes than t, a contradiction.

Lemma 3. If λ is a longest path in $t \in T_{\min}(k)$, $k > 1$, then λ does not contain two or more successive δ-pointers.

Proof. Case 1. λ contains 3 successive δ-pointers, say p_1, p_2, p_3 pointing to $t_1 \in T_\delta(m)$, $t_2 \in T_\delta(m-1)$, $t_3 \in T_{\delta\varrho}(m-2)$, respectively. Modify t as follows to obtain a B-tree t' of height k but with fewer nodes than t: Delete all nodes in t at δ-level 1 except those nodes in t_2. Change p_2 into a ϱ-pointer.

Case 2. Case 1 does not apply, but λ has two successive δ-pointers p_2, p_3 and maybe a ϱ-pointer p_1 preceding p_2. Several specific cases arise, but in all cases modify t as follows to obtain a B-tree t' of height k but with fewer nodes than t: Change p_2 into a ϱ-pointer, p_1 into a δ-pointer. Delete one node (r_2) and one of its subtrees (x_2). We illustrate just one case. The described modification will transform the tree



Following [7] we indicate nodes by circles and trees by triangles.

Proof of Theorem. i) Obvious.

ii) Assume that $t \in T_{\min}(k)$; $k > 1$. Then

a) follows from Lemmas 1 and 2,

b) follows from Lemmas 1 and 3,

c) follows from Lemma 2.

Now assume that properties a), b), and c) hold. Let $|t|$ be the number of nodes in t.

Let $t_{\text{bal}}(h)$ be a completely balanced tree of height h, $t_{\min}(h)$ a minimal B-tree of height h, $t_{abc}(h)$ a B-tree of height h with properties a), b), c). Then from those properties it is clear that every tree $t_{abc}(h)$ satisfies the following recurrence relation:

$$|t_{abc}(h)| = 1 + \left|t_{\text{bal}}\left(\frac{h}{2} - 1\right)\right| + |t_{abc}(h-1)|; \qquad h \text{ even}$$

$$|t_{abc}(h)| = 1 + \left|t_{\text{bal}}\left(\frac{h-1}{2}\right)\right| + |t_{abc}(h-1)|; \qquad h \text{ odd}$$

$$|t_{abc}(0)| = 0$$

$$|t_{abc}(1)| = 1.$$

Thus all B-trees with properties a), b), c) and a given height h have the same number of nodes and therefore must be minimal. q.e.d.

We now solve the above recurrence relations with $|t_{abc}(h)| = |t_{\min}(h)|$. For even h we get

$$\begin{aligned}
|t_{\min}(h)| &= 1 + \left|t_{\text{bal}}\left(\frac{h}{2} - 1\right)\right| + |t_{\min}(h-1)| \\
&= 1 + \left|t_{\text{bal}}\left(\frac{h}{2} - 1\right)\right| + 1 + \left|t_{\text{bal}}\left(\frac{h-2}{2}\right)\right| + |t_{\min}(h-2)| \\
&= 2 + 2\left|t_{\text{bal}}\left(\frac{h}{2} - 1\right)\right| + |t_{\min}(h-2)| \\
&= 2 + 2 \cdot (2^{h/2-1} - 1) + |t_{\min}(h-2)| \\
&= 2^{h/2} + t_{\min}(h-2) \\
&= 2^{h/2} + 2^{h/2-1} + \cdots + 2^1 + 0 \\
&= 2^{h/2+1} - 2.
\end{aligned}$$

Similarly one obtains for odd h:

$$|t_{\min}(h)| = 3 \cdot 2^{\frac{h-1}{2}} - 2.$$

This bound is better than the bound obtained for even h. Let t be a B-tree of height h. Using the worse bound obtained for even h, we obtain as bounds for $|t|$:

$$2^{h/2+1} - 2 \leqq |t_{\min}(h)| \leqq |t| \leqq |t_{\text{bal}}(h)| = 2^h - 1$$

Taking logarithms we obtain:

$$\frac{h}{2} + 1 \leqq \log_2(|t| + 2)$$

$$\log_2(|t| + 1) \leqq h$$

and consequently as bounds for the height h of a B-tree t with $|t|$ nodes:

$$\log_2(|t| + 1) \leqq h \leqq 2 \log_2(|t| + 2) - 2. \tag{1}$$

B-Trees and Balanced Trees

We wish to compare the class of balanced trees with the class of B-trees. The use of the special symbol $*$, i.e. the distinction between horizontal and vertical pointers, in the definition of B-trees makes B-trees strictly speaking different objects than binary trees. Ignoring that distinction, however, we can consider a B-tree as a binary tree since each node has 0, 1, or 2 sons. Given a binary tree t and a B-tree u we want to consider t and u as essentially the same trees if u has the same structure as t except for the use of ϱ-pointers and δ-pointers. The notion of "similarity" makes this idea precise.

Definition. Let t be a binary tree and let u be a B-tree. Following [6] we define the binary relation $\simeq$ of *similarity* between t and u recursively as follows:

$t \simeq u$ if $t = e$ and $u = e$ or

if $t = (x, r, y)$ and [u is (v, s, w) or $(v, *s, w)$ or $(v, s*, w)$ or $(v, *s*, w)$]
and $x \simeq v$ and $r = s$ and $y \simeq w$.

$t \not\simeq u$ otherwise.

The following theorem is a precise formulation of the statement, that each balanced tree can also be considered as a B-tree. We will see that the converse does not hold.

Theorem. Let t be a balanced tree, then there is a B-tree u such that $t \simeq u$.

Proof. We recursively define a function β mapping balanced trees to B-trees such that $t \simeq \beta(t)$. From the definition of β it will be easy to see that:

i) if t is a balanced tree of odd height h, then $\beta(t) \in T_\delta\left(\frac{h+1}{2}\right)$

ii) if t is of even height h, then $\beta(t) \in T_\varrho\left(\frac{h}{2}\right)$. Thus $\beta(t) \in T_{\delta\varrho}\left(\left\lceil\frac{h}{2}\right\rceil\right)$ always. (2)

iii) $t \simeq \beta(t)$.

The function β will not change the structure of t, β only decides which pointers of t should be ϱ-pointers or δ-pointers in considering t as a B-tree.

Definition of ß. Denote by x_{h-1}, y_{h-2} etc. balanced trees of height $h-1$, $h-2$ etc. resp.

Case 1. $\beta(t) = e$ if $t = e$; remember that $e \in T_\varrho(0)$.

Case 2. t is a balanced tree of odd height h.

$$\beta(t) = \begin{cases} (\beta(x_{h-1}), r, \beta(y_{h-1})) & \text{if } t = (x_{h-1}, r, y_{h-1}); \quad h \geqq 1 \\ (\beta(x_{h-2}), r, \beta(y_{h-1})) & \text{if } t = (x_{h-2}, r, y_{h-1}); \quad h \geqq 3 \\ (\beta(x_{h-1}), r, \beta(y_{h-2})) & \text{if } t = (x_{h-1}, r, y_{h-2}); \quad h \geqq 3. \end{cases}$$

Case 3. t is a balanced tree of even height $h \geqq 2$.

$$\beta(t) = \begin{cases} (\beta(x_{h-1}), *r*, \beta(y_{h-1})) & \text{if } t = (x_{h-1}, r, y_{h-1}) \\ (\beta(x_{h-2}), r*, \beta(y_{h-1})) & \text{if } t = (x_{h-2}, r, y_{h-1}) \\ (\beta(x_{h-1}), *r, \beta(y_{h-2})) & \text{if } t = (x_{h-1}, r, y_{h-2}). \end{cases}$$

The proof of properties (2) is straightforward by induction on h. q.e.d.

This theorem says that essentially, i.e. up to similarity, the class of balanced trees is a subclass of the class of B-trees. That it is a proper subclass can easily be seen from the fact that there is no balanced tree similar to the B-tree in Fig. 1.

Fig. 3 shows a B-tree obtained from the balanced tree in Fig. 2 according to the function β.

The upper bound on the height of a B-tree obtained in (1) is approximately $2\log_2(|t|)$ instead of $1.5\log_2(|t|)$ for the height of a balanced-tree [4]. This means that the upper bound for the retrieval time is better for balanced-trees than for

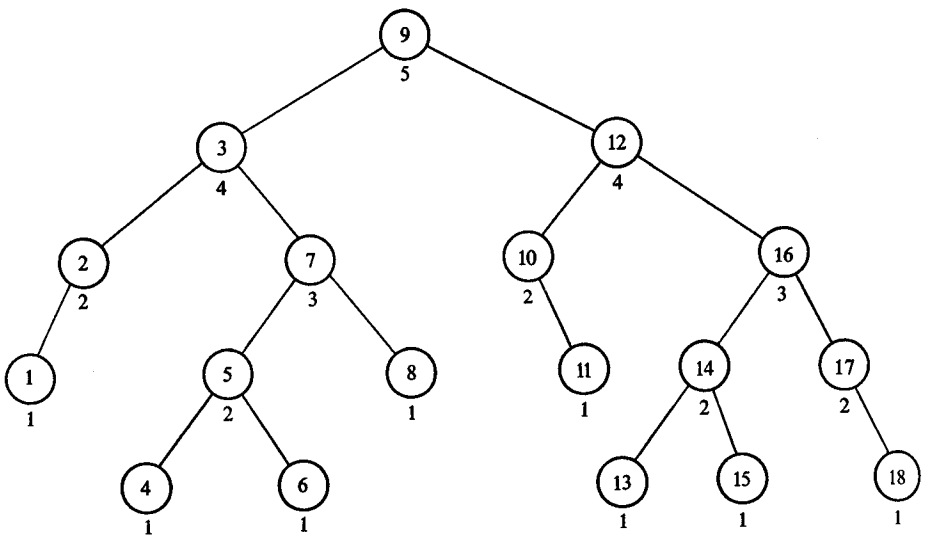


Fig. 2. A balanced tree of height 5

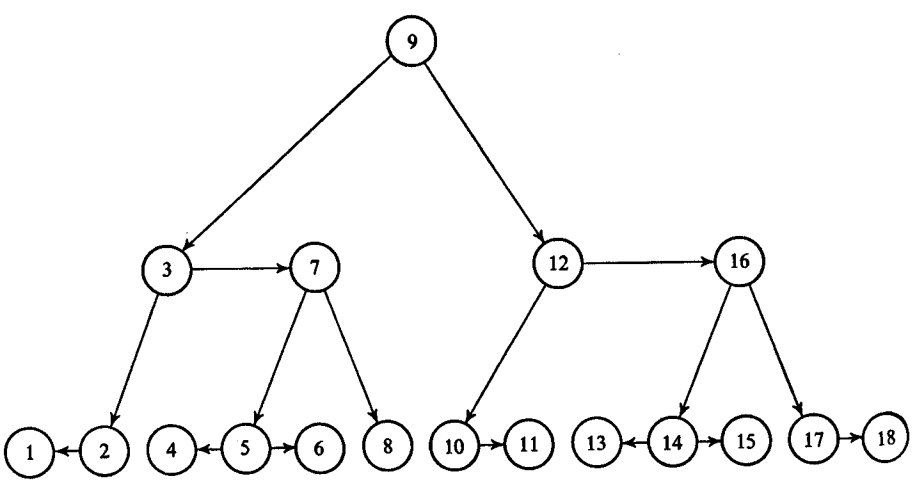


Fig. 3. The balanced tree of Fig. 2 considered as a B-tree of δ-height 3

B-trees. On the other hand, these same bounds and the fact at balanced-trees are a proper subclass of B-trees also suggest that less work sho 'd be required to update B-trees than to update balanced-trees.

Maintenance Algorithms

We now consider the algorithms for maintaining B-trees if keys are inserted and deleted randomly. The algorithm to retrieve keys is straightforward and will not be described here. The following transformations will be needed both in the insertion and the deletion processes whenever two successive ϱ-pointers arise and must be removed by "splitting".

Splitting Algorithm

The function σ will transform certain trees, which are no longer B-trees, back into B-trees. σ will be applied only to trees of the form

$$(u, *r, z), \quad (u, *r*, y), \quad (v, *r*, x), \quad (w, r*, x)$$

where

$$u, x \in T_\varrho(h)$$

$$v, y \in T_\delta(h)$$

$$w, z \in T_{\delta\varrho}(h-1).$$

σ applied to any one of those trees will result in a tree in the class $T_\delta(h+1)$. Intuitively σ will transform trees to remove successive ϱ-pointers, but it will raise the δ-level of the tree by one. Furthermore, if t yields increasing keys on postorder traversal, then $\sigma(t)$ will.

Case 1. Let u be of the form $(u_1, *r_1, u_2)$. Then define:

$$\sigma\big((u, *r, z)\big) = \sigma\big(\big((u_1, *r_1, u_2), *r, z\big)\big) = \big(u_1, r_1, (u_2, r, z)\big).$$

Now $u_1 \in T_\delta(h)$; $(u_2, r, z) \in T_\delta(h)$ since $u_2 \in T_{\delta\varrho}(h-1)$, $z \in T_{\delta\varrho}(h-1)$. Thus $\big(u_1, r_1, (u_2, r, z)\big) \in T_\delta(h+1)$. Furthermore postorder traversal of the resulting tree yields the keys in the same order as postorder traversal of the old tree. For trees of the general form $(u, *r*, y)$ define: $\sigma\big((u, *r*, y)\big) = \big(u_1, r_1, (u_2, r*, y)\big)$. A similar argument as before shows that this tree is in $T_\delta(h+1)$.

Case 2. u is of the form $(u_1, *r_1*, u_3)$ or (u_4, r_1*, u_3) and u_3 is (u_5, r_3, u_6). Then define

$$\sigma\big((u, *r, z)\big) = \begin{cases} \big((u_1, *r_1, u_5), r_3, (u_6, r, z)\big) & \text{if} \quad u = (u_1, *r_1*, u_3) \\ \big((u_4, r_1, u_5), r_3, (u_6, r, z)\big) & \text{if} \quad u = (u_4, r_1*, u_3) \end{cases}$$

$$\sigma\big((u, *r*, y)\big) = \begin{cases} \big((u_1, *r_1, u_5), r_3, (u_6, r*, y)\big) & \text{if} \quad u = (u_1, *r_1*, u_3) \\ \big((u_4, r_1, u_5), r_3, (u_6, r*, y)\big) & \text{if} \quad u = (u_4, r_1*, u_3). \end{cases}$$

Similarly as in case 1 the resulting trees after applying σ are in $T_\delta(h+1)$ and yield the keys on postorder traversal in the same order as before.

The definition of σ for $(v, *r*, x)$ and for $(w, r*, z)$ is left right symmetric with cases 1 and 2. The details are straight-forward and are omitted. They can also be found in our implementation in the procedures SPLITRR and SPLITRL. Case 1 is implemented in SPLITLL, case 2 in SPLITLR.

Insertion Algorithm

We will recursively define a function ι which will insert a node s into a B-tree t. Starting with an empty tree, ι will build a B-tree by repeated insertion of nodes in such a way that postorder traversal of the tree will yield the keys stored at the nodes in increasing order. We will not give an explicit proof that ι will build and maintain B-trees properly, but the following main-observation in the definition of ι will make the construction of an induction proof straightforward.

Denote by $\iota(q, z)$ the tree obtained by inserting node q into tree z. Then

$z \in T_\delta(k) \Rightarrow \iota(q, z) \in T_\delta(k) \cup T_\varrho(k)$. Furthermore, if $\iota(q, z) \in T_\varrho(k)$, then $\iota(q, z)$ is $(x, *r, y)$ or $(x, r*, y)$ but not $(x, *r*, y)$, i.e. the root of $\iota(q, z)$ has exactly one ϱ-pointer. (3)

$z \in T_\varrho(k) \Rightarrow \iota(q, z) \in T_\varrho(k) \cup T_\varrho(k+1)$.

The double arrow means "implies that".

Definition of ι. To insert a node s into the B-tree t:

i) if $t = e$ then $\iota(s, t) = (e, s, e)$. Observe that $t \in T_\varrho(0)$, $\iota(s, t) \in T_\delta(1)$.

ii) t has one of the forms (x, r, y), $(v, *r, y)$, $(x, r*, w)$, $(v, *r*, w)$ and s is equal to r. Let $\iota(s, t) = t$. This means that s is already in the tree. In an implementation of ι some special action may then be taken.

iii) t has one of the forms as in ii) and $s < r$. Assume $t \in T_{\delta\varrho}(h)$, thus $x, y \in T_{\delta\varrho}(h-1)$ and $v, w \in T_\delta(h)$.

Case 1. t is (x, r, y) or $(x, r*, w)$. Then $x \in T_{\delta\varrho}(h-1)$ by the definition of B-trees. Insert s into x, i.e. construct $\iota(s, x)$ and proceed according to one of the following two cases.

Case 1a. $\iota(s, x) \in T_{\delta\varrho}(h-1)$. Then define

$$\iota(s, t) = \begin{cases} (\iota(s, x), r, y) & \text{if } t \text{ is } (x, r, y), \\ (\iota(s, x), r*, w) & \text{if } t \text{ is } (x, r*, w). \end{cases}$$

This means that we simply insert s into the left subtree x, but do not change t further. Note that $\iota(s, t) \in T_{\delta\varrho}(h)$.

Case 1b. $\iota(s, x) \in T_\delta(h)$. Then define

$$\iota(s, t) = \begin{cases} (\iota(s, x), *r, y) & \text{if } t \text{ is } (x, r, y), \\ (\iota(s, x), *r*, w) & \text{if } t \text{ is } (x, r*, w). \end{cases}$$

This means that we insert s into the left subtree x, but since this increases the δ-height of the left subtree by 1, we also have to change the left pointer from r to become a ϱ-pointer. Note that $\iota(s, t) \in T_\varrho(h)$ and it has one of the forms described in (3).

Case 2. t is $(v, *r, y)$ or $(v, *r*, w)$, i.e. $t \in T_\varrho(h)$. Thus $v \in T_\delta(h)$ by the definition of a B-tree. Insert s into v, i.e. construct $\iota(s, v)$ and proceed according to one of the following two cases:

Case 2a. $\iota(s, v) \in T_\delta(h)$. Then define

$$\iota(s, t) = \begin{cases} (\iota(s, v), *r, y) & \text{if } t \text{ is } (v, *r, y), \\ (\iota(s, v), *r*, w) & \text{if } t \text{ is } (v, *r*, w). \end{cases}$$

This means that we simply insert s into the left subtree, but we do not change t further. Note that $\iota(s, t) \in T_\varrho(h)$.

Case 2b. $\iota(s, v) \in T_\varrho(h)$. Now $(\iota(s, v), *r, y)$ or $(\iota(s, v), *r*, w)$ are no longer B-trees, but $\sigma(\iota(s, v), *r, y)$ and $\sigma(\iota(s, v), *r*, w)$ are B-trees in $T_\delta(h+1)$. We define

$$\iota(s, t) = \begin{cases} \sigma(\iota(s, v), *r, y) & \text{if } t = (v, *r, y), \\ \sigma(\iota(s, v), *r*, w) & \text{if } t = (v, *r*, w). \end{cases}$$

iv) t has one the forms as in ii) and $r < s$. The definition of ι is left-right symmetric to case iii). The details are omitted.

It is crucial to observe that the depth of recursion of ι is limited by the height of the tree. Now one can represent B-trees in such a way, as e.g. in our implementation, that the work for the transformation performed by ι (and by σ) on a B-tree is at each level bounded by a constant. Then the total amount of work required for the insertion of a single node into a tree t is at worst proportional to the height of the tree, i.e. to $2\log_2(|t|+2) - 2$.

Deletion Algorithm

In this algorithm we must distinguish between a node and the key stored at a node. To delete a key s from a B-tree t, first locate the node, say n, containing s in t. Then, if n has a nonempty left (right) subtree, replace s by the next smallest (next largest) key, say q, in t. q is found easily proceeding from n one step along the left (right) pointer and then along the right (left) pointer as long as possible. Now s is no longer in the tree and q is stored at node n. The node m containing q originally is at the lowest δ-level and will have at least one empty subtree. We will then delete m and the copy of q stored at m from t. Thus we may assume without loss of generality that s will always be deleted from the lowest δ-level in the tree and that s will have at least one empty subtree. In our implementation the replacement of s by q and the deletion of m from t are merged into a single algorithm.

We now define recursively a function α which deletes a key s from a B-tree t —under the assumption that s is stored at a node (at the lowest δ-level) with at least one empty subtree—and results in a B-tree $\alpha(s, t)$.

Observe that in the definition of α we will get the following transformations:

$$y \in T_\varrho(k) \Rightarrow \alpha(s, y) \in T_\delta(k) \cup T_\varrho(k)$$

$$y \in T_\delta(k) \Rightarrow \alpha(s, y) \in T_\delta(k) \quad \text{or} \quad \alpha(s, y) \in T_\varrho(k-1) \quad \text{and is of one of the forms} \quad (4)$$

$$e,\ (y_1, *r_1, y_2),\ (y_1, r_1*, y_2), \quad \text{but not} \quad (y_1, *r_1*, y_2).$$

Definition of α. i) $t \in T_{\delta\varrho}(1)$ and t has at least one empty subtree.

$$\alpha(s,t) = \begin{cases} e & \text{if } t \text{ is } (e,s,e) \\ x & \text{if } t \text{ is } (x,*s,e) \text{ or } (e,s*,x) \\ t & \begin{cases} \text{if } t \text{ is } (e,r,e) & \text{and } s \neq r \text{ or} \\ \text{if } t \text{ is } (x,*r,e) & \text{and } s > r \text{ or} \\ \text{if } t \text{ is } (e,r*,x) & \text{and } s < r. \end{cases} \\ \text{In the last three cases the key } s \text{ to be deleted is not even in the tree and} \\ \text{generally in an implementation some special action will be taken.} \end{cases}$$

This is case A in the implementation.

ii) Assume that t is of one of the forms (x,r,y), $(v,*r,y)$, $(x,r*,w)$, $(v,*r*,w)$ and $s>r$ and $x,y,v,w \neq e$.

Case 1. $y \in T_{\delta\varrho}(h)$; $\alpha(s,y) \in T_{\delta\varrho}(h)$;

$$\alpha(s,t) = \begin{cases} (x,r,\alpha(s,y)) & \text{if } t=(x,r,y) \in T_\delta(h+1) \\ (v,*r,\alpha(s,y)) & \text{if } t=(v,*r,y) \in T_\varrho(h+1). \end{cases}$$

Thus $\alpha(s,t) \in T_{\delta\varrho}(h+1)$.

Case 2. $w \in T_\delta(h+1)$; $\alpha(s,w) \in T_\delta(h+1)$;

$$\alpha(s,t) = \begin{cases} (x,r*,\alpha(s,w)) & \text{if } t=(x,r*,w) \in T_\varrho(h+1) \\ (v,*r*,\alpha(s,w)) & \text{if } t=(v,*r*,w) \in T_\varrho(h+1). \end{cases}$$

Thus $\alpha(s,t) \in T_\varrho(h+1)$.

In our implementation cases 1 and 2 are taken care of by shortcutting the recursion of σ (**goto** QUIT) as soon as no further modifications of the tree are required.

Case 3a. $y \in T_\delta(h)$; $\alpha(s,y) \in T_\varrho(h-1)$; $t=(x,r,y) \in T_\delta(h+1)$;

$$\alpha(s,t) = \begin{cases} (x,*r,\alpha(s,y)) & \text{if } x \in T_\delta(h) \\ \sigma(x,*r,\alpha(s,y)) & \text{if } x \in T_\varrho(h). \end{cases}$$

Thus $\alpha(s,t) \in T_\delta(h+1) \cup T_\varrho(h)$. This is case B in the implementation.

Case 3b. $y \in T_\delta(h)$; $\alpha(s,y) \in T_\varrho(h-1)$; $t=(y,*r,y) \in T_\varrho(h+1)$ and $v=(v_1,r_1,v_2) \in T_\delta(h+1)$

$$\alpha(s,t) = \begin{cases} (v_1,r_1,(v_2,*r,\alpha(s,y))) & \text{if } v_2 \in T_\delta(h) \\ (v_1,r_1*,\sigma(v_2,*r,\alpha(s,y))) & \text{if } v_2 \in T_\varrho(h). \end{cases}$$

Thus $\alpha(s,t) \in T_{\delta\varrho}(h+1)$. This is case C in the implementation.

Case 4. $w \in T_\delta(h+1)$; $\alpha(s,w) \in T_\varrho(h)$;

$$\alpha(s,t) = \begin{cases} (x,r,\alpha(s,w)) & \text{if } t=(x,r*,w) \in T_\varrho(h+1) \\ (v,*r,\alpha(s,w)) & \text{if } t=(v,*r*,w) \in T_\varrho(h+1). \end{cases}$$

Thus $\alpha(s,t) \in T_{\delta\varrho}(h+1)$. This is case F in the implementation.

iii) If s is smaller than r then definition of α is left-right symmetric with ii). The details are straightforward and are omitted.

Note that the depth of recursion necessary for α is limited by the height of the tree. Similarly as in the insertion process the total amount of work required for the deletion of a single key is at worst proportional to the height of the tree.

Main Result

The work that must be performed for random retrievals, insertions, and deletions is even in the worst cases proportional to the height of the B-tree, i.e. to $\log_2(|t|)$ where $|t|$ is the number of keys in the tree t.

Generalization

From the insertion and deletion algorithms discussed in this paper, it is quite clear that the class of binary B-trees could be enlarged by allowing up to n successive ϱ-pointers for $n = 2, 3, 4, \ldots$ before requiring any modification or "rebalancing" of the tree. This would require less rebalancing, but performance proportional in time to $\log(|t|)$ would still be guaranteed.

Implementation of Insertion and Deletion Algorithms for B-Trees

For the ALGOL 60 implementation to be considered here a node in a B-tree shall consist of five fields, namely:

LBIT: a Boolean variable to indicate that the left arc is a ϱ-arc (**true**) or a δ-arc (**false**)

LP: the left downward pointer, an integer

KEY: the key in the node, a real

RP: the right pointer, downward or horizontal, also an integer

RBIT: a Boolean variable to indicate that the right pointer is a ϱ-arc (**true**) or a δ-arc (**false**)

The absence of a pointer shall be represented by the value 0. Thus the insertion and deletion procedure have array parameters LBIT, LP, KEY, RP, RBIT to store the nodes of the tree. The parameter X is the key to be inserted into or deleted from the tree to whose root the parameter ROOT is pointing (ROOT = 0 for an empty tree). The Boolean ROOTBIT indicates ROOT as a ϱ-arc or as a δ-arc. There are two procedure parameters to maintain a list of free nodes, namely ADDQ for the deletion procedure to enter a freed node into the free list, and GETQ for the insertion procedure to obtain a free node from the free list. Both ADDQ and GETQ have one integer parameter pointing to the node added to or obtained from the free list. If the key to be inserted is already in the tree, control will be transferred to the label parameter FOUNDX. If the key to be deleted is not in the tree, control will be transferred to the label parameter XNOTINTREE by the deletion procedure.

The parameter P in SYMSERT and SYMDELETE is the pointer to the root of the subtree in which the insertion or deletion must be performed. The parameter BIT in SYMSERT indicates whether P is a ϱ-arc or a δ-arc.

The four procedures SPLITRR, SPLITRL, SPLITLL, and SPLITLR modify the B-tree in order to remove successive ϱ-pointers. They are used both in the insertion procedure SYMINS and in the deletion procedure SYMDEL, and are the implementation of the function σ.

Other local quantities in the procedures are:

AUXP: an auxiliary integer variable used as a temporary store for pointers.

DONE: a label to which control is transferred after completing an insertion in order to shortcut the full recursion of SYMSERT.

AUXX: an auxiliary integer variable pointing to the key X after it has been found in the tree. AUXX = 0 otherwise.

QUIT: a label to which control is transferred after completing the deletion of the key in order to shortcut the full recursion of SYMDELETE.

AUXD: an auxiliary integer variable used as temporary store for pointers.

SL: a label from where deletion of the key from the left subtree (smaller) is continued.

GL: a label from where deletion of the key from the right subtree (greater) is continued.

The insertion (deletion) algorithm has been written as two procedures, a non-recursive outer procedure SYMINS (SYMDEL) and a recursive inner procedure SYMSERT (SYMDELETE). The outer procedure SYMINS (SYMDEL) allows shortcutting the full recursion of SYMINSERT (SYMDELETE) via the label DONE (QUIT). The inner procedure SYMINSERT (SYMDELETE) performs insertions (deletions) in a B-tree recursively.

It is assumed that the six procedures SPLITRR, SPLITRL, SPLITLL, SPLITLR, SYMINS, and SYMDEL are all declared in the same block or in such a way that SPLITRR, SPLITRL, SPLITLL, and SPLITLR can be used both in SYMINS and in SYMDEL.

Note. The tree in Fig. 1 is a suitable tree for testing. Inserting the keys in the order 8, 9, 11, 15, 19, 20, 21, 7, 3, 2, 1, 5, 6, 4, 13, 14, 10, 12, 17, 16, 18 will build up the tree. Deleting the keys in the order 1, 6, 2, 21, 16, 20, 8, 14, 11, 9, 5, 10, 12, 13, 3, 4, 7, 15, 17, 18, 19 will exercise all the cases which can arise in any deletion process.

```
procedure SPLITRR (P, LP, RP, RBIT);
  integer P; integer array RP, LP;
  Boolean array RBIT;
  begin integer AUXP;
    AUXP:=RP [P]; RBIT [AUXP]:=false;
    RP [P]:=LP [AUXP]; RBIT [P]:=false;
    LP [AUXP]:=P; P:=AUXP
  end OF SPLITRR
```

```
procedure SPLITRL (P, LP, RP, LBIT, RBIT);
  integer P; integer array LP, RP;
  Boolean array LBIT, RBIT;
  begin integer AUXP;
    AUXP:=LP [RP[P]];
    LBIT [RP[P]]:=false; LP [RP[P]]:=RP [AUXP];
    RP [AUXP]:=RP [P]; RP [P]:=LP [AUXP];
    RBIT [P]:=false; LP [AUXP]:=P; P:=AUXP
  end OF SPLITRL

procedure SPLITLL (P, LP, RP, LBIT);
  integer P; integer array LP, RP;
  Boolean array LBIT;
  begin integer AUXP;
    AUXP:=LP [P]; LBIT [AUXP]:=false;
    LP [P]:=RP [AUXP]; LBIT [P]:=false;
    LP [AUXP]:=P; P:=AUXP
  end of SPLITLL

procedure SPLITLR (P, LP, RP, LBIT, RBIT);
  integer P; integer array LP, RP;
  Boolean array LBIT, RBIT;
  begin integer AUXP;
    AUXP:=RP [LP [P]];
    RP [LP [P]]:=LP [AUXP]; RBIT [LP [P]]:=false;
    LP [AUXP]:=LP [P]; LP [P]:=RP [AUXP];
    LBIT [P]:=false; RP [AUXP]:=P; P:=AUXP
  end OF SPLITLR

procedure SYMINS (X, ROOT, ROOTBIT, FOUNDX, LP, RP, KEY, LBIT,
RBIT, GETQ);
  value X; real X; integer ROOT; Boolean ROOTBIT;
  label FOUNDX; integer array LP, RP; array KEY;
  Boolean array LBIT, RBIT; procedure GETQ;
  begin procedure SYMSERT (P, BIT);
    integer P; Boolean BIT;
    if P=0 then
      begin comment INSERT X AS NEW LEAF;
        GETQ (P); KEY [P]:=X; BIT:=true;
        LP [P]:=RP [P]:=0; LBIT [P]:=RBIT [P]:=false
      end
    else if X=KEY [P] then goto FOUNDX
    else if X<KEY [P] then
      begin comment INSERT X IN LEFT SUBTREE;
        SYMSERT (LP [P], LBIT [P]);
          if LBIT [P] then begin
```

```
            if LBIT [LP[P]] then begin SPLITLL (P, LP, RP, LBIT);
              BIT:=true end
            else if RBIT [LP[P]] then begin
              SPLITLR (P, LP, RP, LBIT, RBIT); BIT:=true end end
            else goto DONE
        end

    else begin comment INSERT X IN RIGHT SUBTREE;
        SYMSERT (RP [P], RBIT [P]);
          if RBIT [P] then begin if RBIT [RP[P]] then
            begin SPLITRR (P, LP, RP, RBIT);
              BIT:=true end
            else if LBIT [RP[P]] then begin
              SPLITRL (P, LP, RP, LBIT, RBIT); BIT:=true end end
            else goto DONE
        end OF SYMSERT;
    SYMSERT (ROOT, ROOTBIT); DONE:
end OF SYMINS

procedure SYMDEL (X, ROOT, XNOTINTREE, LP, RP, KEY, LBIT,
RBIT, ADDQ);
    value X; real X; integer ROOT;
    label XNOTINTREE; integer array LP, RP; array KEY;
    Boolean array LBIT, RBIT; procedure ADDQ;
    begin integer AUXX, AUXD;

      comment RECURSIVE B-TREE DELETION ALGORITHM;
      procedure SYMDELETE (P); integer P;
        begin comment DID WE FIND THE KEY TO BE DELETED;
        if X=KEY [P] then AUXX:=P;
        if X≦KEY [P]∧LP [P]≠0 then
      SL: begin SYMDELETE (LP[P]);
        comment CASES D, E, G;
        if LBIT [P] then begin comment CASE G;
          LBIT [P]:=false; goto QUIT end OF CASE G

        else begin comment CASES E, D;
          if RBIT [P] then begin comment CASE E;
          AUXD:=RP [P]; RP [P]:=LP [AUXD];
          LP [AUXD]:=P; P:=AUXD;
            if LBIT [RP[LP[P]]] then
              begin SPLITRL (LP[P], LP, RP, LBIT, RBIT);
                LBIT [P]:=true end
            else if RBIT [RP[LP[P]]] then
            begin SPLITRR (LP[P], LP, RP, RBIT);
              LBIT [P]:=true end;
            goto QUIT
        end OF CASE E
```

```
      else begin comment CASE D;
      RBIT [P]:=true; if LBIT [RP[P]] then begin
      SPLITRL (P, LP, RP, LBIT, RBIT); goto QUIT end
      else if RBIT [RP[P]] then begin
      SPLITRR (P, LP, RP, RBIT); goto QUIT END
      end OF CASE D
    end OF CASES D, E
  end OF SL AND CASES D, E, G

  else if X≧KEY [P]∧RP [P]≠0 then
GL: begin SYMDELETE (RP[P]);
  comment CASES B, C, F;
  if RBIT [P] then begin comment CASE F;
    RBIT [P]:=false; goto QUIT end OF CASE F

      else begin comment CASES B, C;
        if LBIT [P] then begin comment CASE C;
        AUXD:=LP [P]; LP [P]:=RP [AUXD];
        RP [AUXD]:=P; P:=AUXD;
          if RBIT [LP[RP[P]]] then
            begin SPLITLR (RP [P], LP, RP, LBIT, RBIT);
              RBIT [P]:=true end
            else if LBIT [LP[RP[P]]] then
            begin SPLITLL (RP[P], LP, RP, LBIT);
              RBIT [P]:=true end;
          goto QUIT
        end OF CASE C

        else begin comment CASE B;
        LBIT [P]:=true;
          if RBIT [LP[P]] then begin
        SPLITLR (P, LP, RP, LBIT, RBIT); goto QUIT end
        else if LBIT [LP[P]] then begin
        SPLITLL (P, LP, RP, LBIT); goto QUIT end
        end OF CASE B
      end OF CASES B, C
    end OF GL AND CASES B, C, F

  else begin comment ARRIVED AT LEAF OR NEXT TO ONE, CASE A;
      if AUXX=0 then goto XNOTINTREE;
    KEY [AUXX]:=KEY [P];
    AUXD:=if LBIT [P] then LP [P] else RP [P];
    ADDQ (P); P:=AUXD; if P≠0 then goto QUIT
  end
end OF SYMDELETE;
```

```
  AUXX:=0;
    if ROOT =0 then goto XNOTINTREE else
      SYMDELETE (ROOT);
  QUIT:
end OF SYMDEL
```

References

1. Adelson-Velskii, G. M., Landis, E. M.: An information organization algorithm. DANSSSR **146**, 263–266 (1962).
2. Bayer, R., McCreight, E. M.: Organization and maintenance of large ordered indexes. Acta Informatica **1**, 173–189 (1972).
3. Bayer, R.: Binary B-trees for virtual memory. Proceedings of 1971 ACM SIGFIDET Workshop on Data Description, Access and Control, edited by E. F. Codd and A. L. Dean. pp. 219–235 (Nov. 11–12, 1971), San Diego.
4. Foster, C. C.: Information storage and retrieval using AVL-trees. Proc. ACM 20th Nat'l. Conf., p. 192–205. 1965.
5. Knott, G. D.: A balanced tree structure and retrieval algorithm. Proc. of Symposium on Information Storage and Retrieval, Univ. of Maryland, April 1–2, 1971, pp. 175–196.
6. Knuth, D. E.: The art of computer programming, vol. 1. Addison-Wesley, 1969.
7. Nievergelt, J., Reingold, E. M.: Binary search trees of bounded balance. To appear, Proceedings of 4th ACM SIGACT Conference 1972.

Prof. Dr. Rudolf Bayer
Mathematisches Institut
der Technischen Universität, München
D-8000 München 2
Arcisstr. 21
Federal Republic of Germany