BINARY B-TREES FOR VIRTUAL MEMORY*

Rudolf Bayer
Computer Sciences
Purdue University
Lafayette, Indiana 47907

ABSTRACT

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. Binary B-trees are a modification of B-trees described previously by Bayer and McCreight. They avoid the storage overhead encountered with B-trees and are suitable for processing in a one-level store.

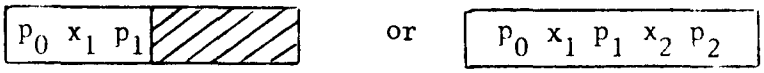
* This work was partially supported by an NSF grant.

BINARY B-TREES FOR VIRTUAL MEMORY

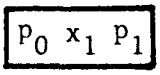
This paper will describe a further solution to the following well-known problem in information processing:

> 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.

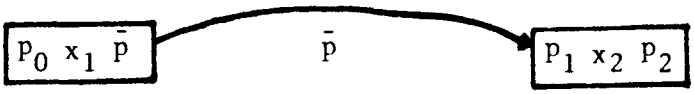
Other solutions to this problem have been described for a one-level store in [4] and for a two-level store with a pseudo-random access backup store in [2]. The following technique is suitable for a one-level store and avoids the storage overhead incurred by B-trees. Readers familiar with [2] and [4] will recognize it as a modification of B-trees [2] by considering B-trees with k = 1, thus consisting of pages containing one or two keys:



where the x_i are keys and the p_i are pointers to other pages in a B-tree. Instead of storing pages as "physical" pages, thus possibly half-empty, we can store them as threaded lists of 1 or 2 elements, each element consisting of one key, two pointers, and, as we shall see, one additional bit. Thus the half-empty page above would be represented as:



and the full page would be represented as:



Clearly 1 bit is needed to indicate, whether the right pointer in such an element is a "horizontal" (ρ-pointer) pointer to the second element of a "full" page or a "vertical" pointer (δ-pointer) to a page in the next level of the B-tree.

Representing B-trees with k = 1 in this way, however, leads to a certain class of binary trees, called "binary B-trees". The example in Fig. 1 shows a B-tree and the corresponding binary B-tree using an obvious graphical representation.

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

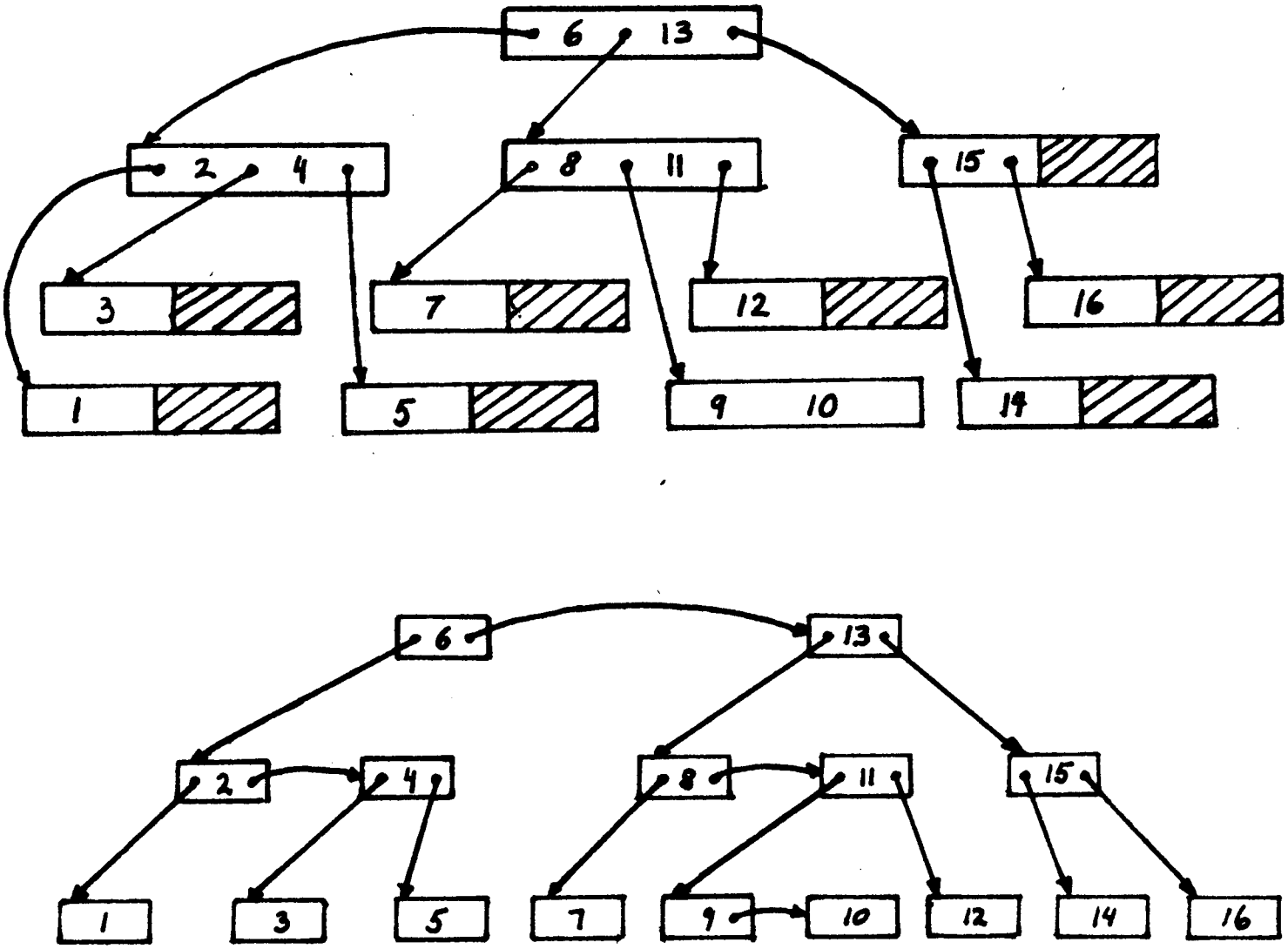


Fig. 1: A B-tree of pages (top) and the corresponding binary B-tree (bottom).

Definition:

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

i) all left arcs are δ-arcs

ii) each path from the root to any leaf contains the same number of δ-arcs

iii) some of the right arcs may be ρ-arcs, but there may be no successive ρ-arcs, i.e. each ρ-arc must be followed by at least one right δ-arc or must lead to a leaf.

In addition, the keys shall be stored at the nodes of a binary 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:

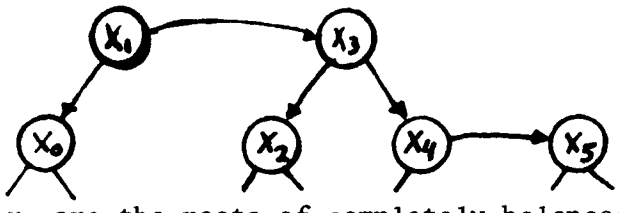
1) If the tree is empty, do nothing
2) Traverse left subtree
3) Visit root
4) Traverse right subtree.

The second tree in Fig. 1 is an example of a binary B-tree. Readers familiar with AVL-trees [1], [4], [5], should observe that binary B-trees are not AVL-trees, e.g. the subtree with key 8 as the root in the binary B-tree in Fig. 1 does not satisfy the AVL-criterion.

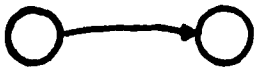
Number of Nodes and Height of a Binary B-tree

Let the height h of a binary B-tree be the maximal number of nodes in any path from the root to a leaf.

Then a binary B-tree $T_{min}(h)$ of even height h with the smallest number of nodes is of the form



where x_0, x_2 are the roots of completely balanced binary trees of height $\frac{h}{2} - 1$ and x_4 is the root of a tree $T_{min}(h-2)$. $T_{min}(2)$ is of the form



Let N(T) be the number of nodes in tree T. Let $T_{bal}(\ell)$ be a completely balanced binary tree of height ℓ. Then we have:

$$N(T_{min}(h)) = 2N(T_{bal}(\frac{h}{2} - 1)) + 2 + N(T_{min}(h-2))$$

Since

$$N(T_{bal}(\ell)) = 2^0 + 2^1 + 2^2 + \ldots + 2^{\ell-1} = 2^\ell - 1$$

we obtain:

$$N(T_{min}(h)) = 2 \cdot (2^{\frac{h}{2}-1} - 1) + 2 + N(T_{min}(h-2))$$

$$= 2^{\frac{h}{2}} + N(T_{min}(h-2))$$

$$= 2^{\frac{h}{2}} + 2^{\frac{h}{2}-1} + \ldots + 2^1$$

$$= 2^{\frac{h}{2}+1} - 2$$

For a binary B-tree of odd height h we obtain:

$$N(T_{min}(h)) = 1 + N(T_{bal}(\frac{h-1}{2})) + N(T_{min}(h-1))$$

$$= 2^{\frac{h-1}{2}} + 2^{\frac{h+1}{2}} - 2 = 3 \cdot 2^{\frac{h-1}{2}} - 2$$

This bound is better than the bound obtained for even h.

Using the worse bound obtained for even h, and if N is the number of nodes in a binary B-tree of height h, then we obtain as bounds for N:

$$2^{\frac{h}{2}+1} - 2 \leq N(T_{min}(h)) \leq N \leq N(T_{bal}(h)) = 2^h - 1$$

Taking logarithms we obtain:

$$\frac{h}{2} + 1 \leq \log_2(N+2)$$

$$\log_2(N+1) \leq h$$

and consequently as sharp bounds for the height h of a binary B-tree with N nodes:

$$\log_2(N+1) \leq h \leq 2 \log_2(N+2) - 2 \qquad (1)$$

Comparing AVL and binary B-trees one notes:

i) The upper bound on the height of the tree for a given number of nodes is not as good as for AVL-trees, [4], meaning that binary B-trees do not have to be quite as well balanced as AVL-trees.

ii) Binary B-trees require only one bit per node to store the "balancing" information as opposed to two bits per node for AVL-trees.

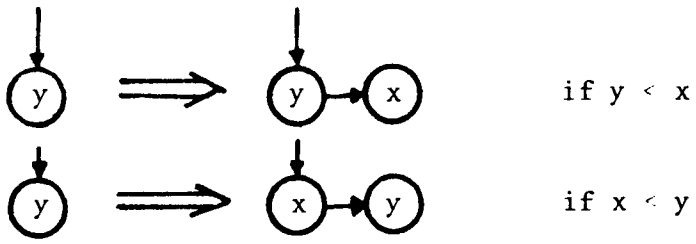
iii) The algorithms for maintaining binary B-trees seem to be simpler than those for AVL-trees. This might be of particular importance for running the algorithms on peripheral processors or for using micro-programmed implementations.

We now consider the algorithms for maintaining binary B-trees if keys are inserted and deleted randomly. The algorithm to retrieve keys is straightforward and will not be described here.

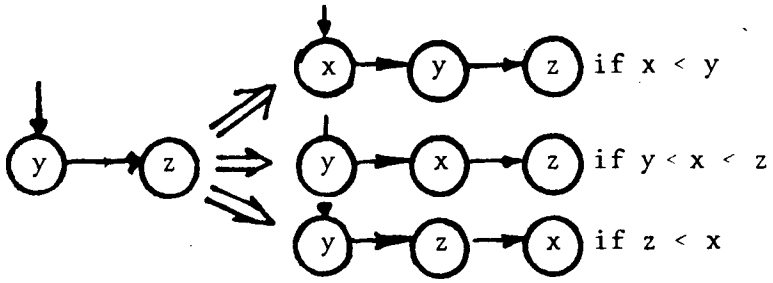
Insertion Algorithm:

First a new key x is inserted into its proper place next to a leaf y, where y is the leaf reached by trying to retrieve x from the tree. x is attached via a horizontal pointer to the left of y if x < y and y has no left successor, or to the right of y if x > y and y has no right successor. This leads to one of the following transformations at the lowest δ-level of the tree:

Case 1:



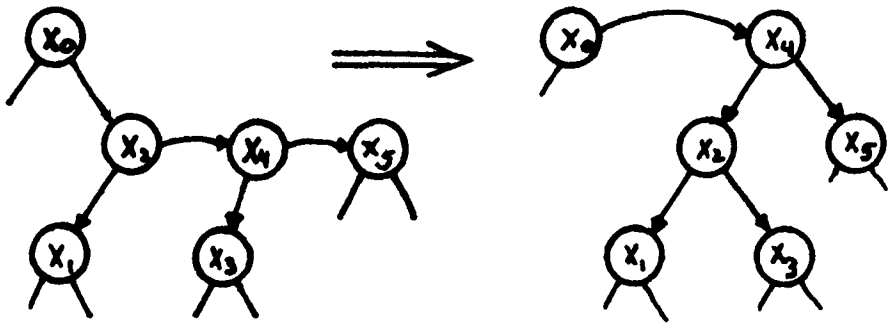
Case 2:



In Case 1 the insertion process is complete. But in Case 2 two successive horizontal pointers have been introduced and must then be removed according to one of the following cases. Removing successive horizontal pointers is a recursive process. Starting at the δ-level of the leaves as in Case 2 above, this removal can lead to successive horizontal pointers at the next δ-level closer to the root which must then be removed, etc. until no new pair of successive horizontal pointers has been created or a new root has been introduced. This corresponds to the process of splitting pages described in [2].

We assume that we start out with a binary B-tree and want to perform one insertion to obtain again a binary B-tree. Considering Cases 1 to 4 it will become clear that there will never be more than one instance of two successive horizontal pointers in the binary B-tree at any time during the update process. We get the following two cases of transformations or rearrangements in the tree:

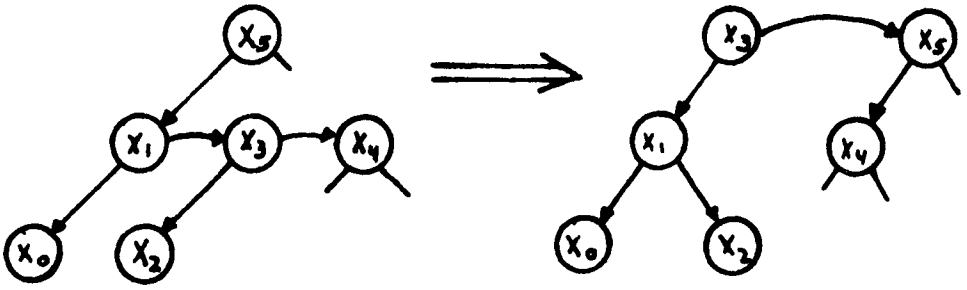
Case 3:



where $x_i < x_j$ iff $i < j$. The subtrees, of which x_1, x_3, and x_5 are the roots, are not affected by this transformation. Introducing the new horizontal pointer between x_0 and x_4 may now lead to two successive horizontal pointers and may necessitate a transformation of the tree at a level closer to the root.

If x_2 us the root, i.e. x_o is missing, then x_4 simply becomes the new root.

Case 4:

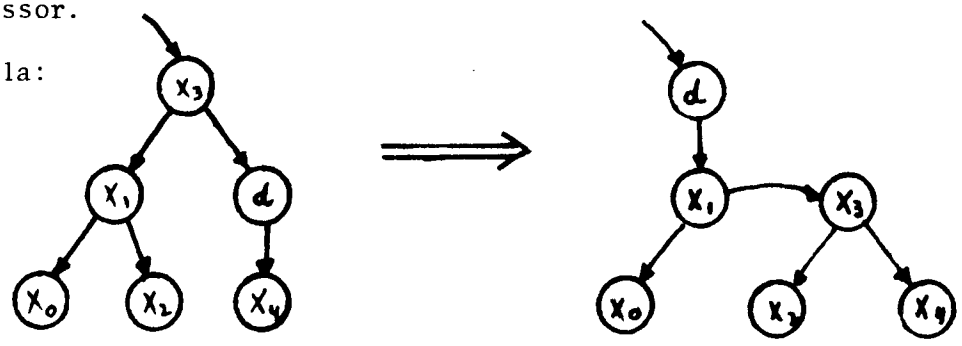


Deletion Algorithm:

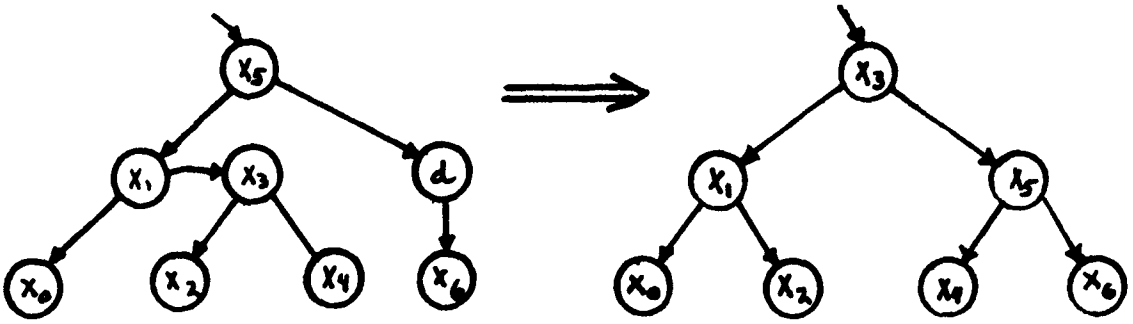
To delete an element x from the tree, we first have to locate it and replace it by the next largest key, say y, in the tree. y can be found easily by first proceeding from x one step along the right pointer and then along the left pointer as long as possible. The node containing y originally is then replaced by a dummy node d. If y had a right (horizontal) successor z then the transformation

completes the deletion process. Otherwise d will then be removed from the tree by recursively performing one of the following transformations on the tree. Note that at any one time d has at most one downward successor.

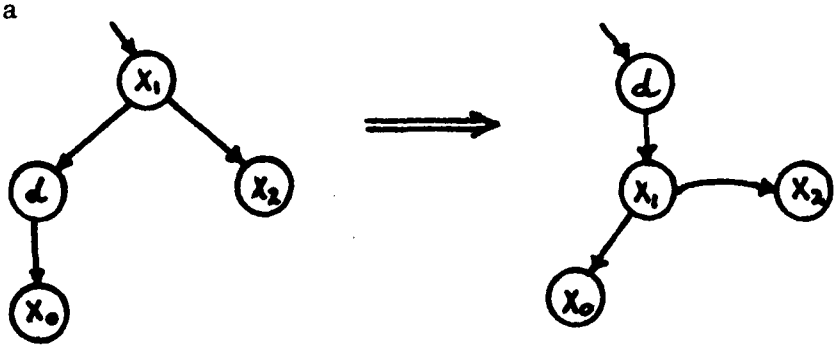
Case 1a:



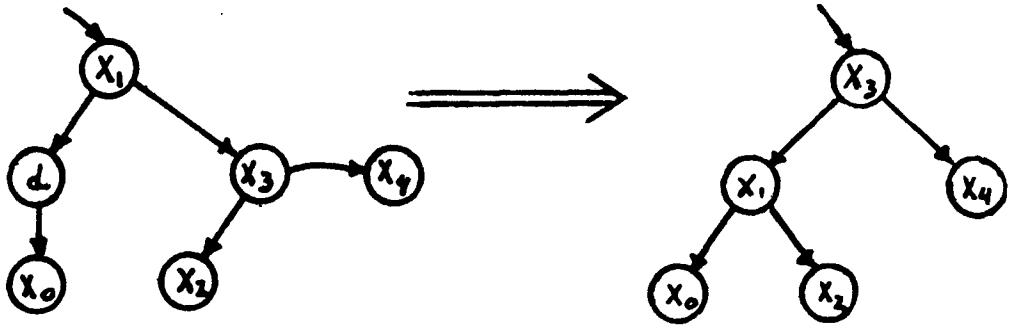
Case 1 b



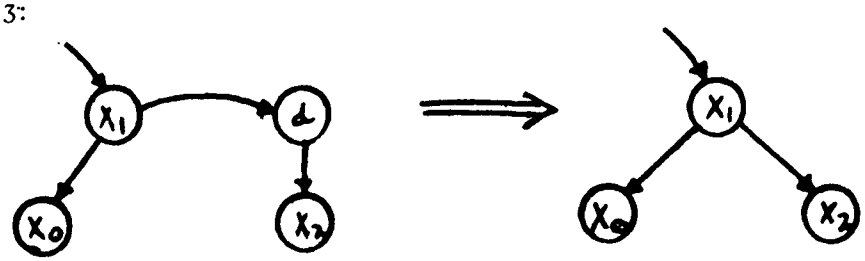
Case 2a



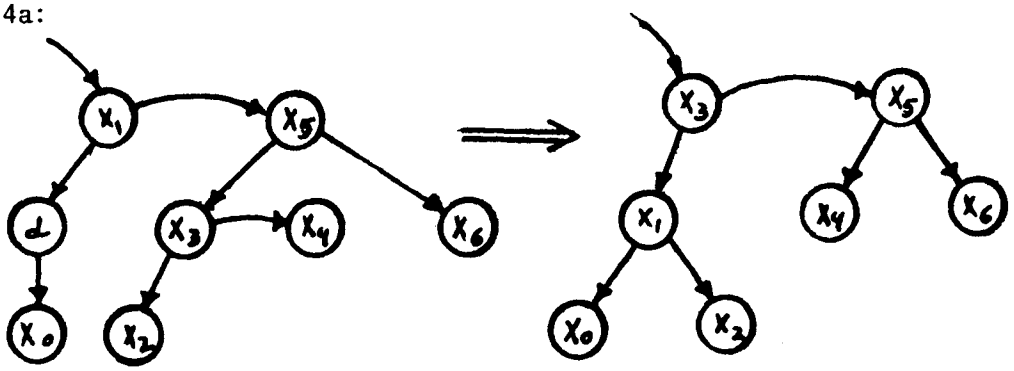
Case 2b



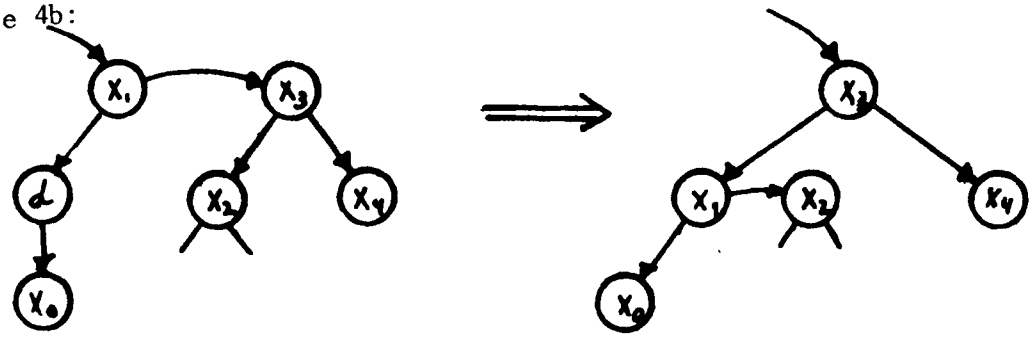
Case 3:



Case 4a:



Case 4b:



Note that after one of the cases 1b, 2b, 3, 4a, 4b the deletion process is complete. If, however, d was moved all the way up to the root of the tree, then it will be deleted and the successor of d will become the new root of the tree.

Note that a single retrieval, insertion, or deletion requires inspection and modification of the tree only along a single path from the root to a leaf. As a consequence of this observation and of the bounds for the heights of a binary B-tree obtained in (1) the following main result of this paper is obtained:

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 tree, i.e. to $\log_2 N$ where N is the number of keys in the tree.

IMPLEMENTATION OF INSERTION AND DELETION ALGORITHMS FOR BINARY B-TREES

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

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

HBIT: a Boolean variable to indicate that the right pointer is horizontal (true) or downward (false) in the tree.

The absence of a pointer shall be represented by the value 0. Thus the insertion and deletion procedure have array parameters LP, KEY, RP, HBIT 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 P is pointing (P=0 for an empty tree).

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.

Other local quantities in the insertion procedure INS are:

AUX1: 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.

Other local quantities in the deletion procedure DEL:

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 dummy node d in order to shortcut the full recursion.

AUXD: an auxiliary variable of type integer pointing to the successor of the dummy node d during the deletion process.

L: a label from where the search for the next largest key is continued after finding x.

LEAF: a label where processing is continued after arriving at a leaf in the deletion process.

Note: The insertion (deletion) algorithm has been written as two procedures, a non-recursive outer procedure INS(DEL) and a recursive inner procedure INSERT(DELETE). The outer procedure INS(DEL) handles the special case of an empty tree and allows shortcutting of the full recursion of INSERT(DELETE) via the label DONE(QUIT). The inner procedure INSERT(DELETE) performs insertions (deletions) in a non-empty tree recursively.

```
#PROCEDURE# INS(X,P,FOUNDX,LP,RP,KEY,HBIT,GETQ).,
    #VALUE# X.,
    #REAL# X.; #INTEGER# P.; #LABEL# FOUNDX.,
    #INTEGER# #ARRAY# LP, RP.,  #ARRAY# KEY.,
    #BOOLEAN# #ARRAY# HBIT.,  #PROCEDURE# GETQ.,


#BEGIN#  #INTEGER# AUX1.,
#PROCEDURE# INSERT(P).; #INTEGER# P.,
  #BEGIN# #IF# X = KEY(/P/) #THEN# #GOTO# FOUNDX
    #ELSE# #IF# X #GREATER# KEY(/P/) #THEN#
     #BEGIN#  #IF# RP(/P/) = 0 #THEN#
      #BEGIN# #COMMENT# INSERT KEY TO RIGHT OF LEAF.,
        GETQ(AUX1).; KEY(/AUX1/) .= X.,
        LP(/AUX1/) .= RP(/AUX1/) .= 0.; HBIT(/AUX1/) .= #FALSE#.,
        RP(/P/) .= AUX1.; HBIT(/P/) .= #TRUE#
      #END#
      #ELSE#
      #BEGIN#  #COMMENT#  INSERT X IN RIGHT SUBTREE.,
        INSERT(RP(/P/)).,
        #IF# HBIT(/RP(/P/)/) #THEN#
        #BEGIN# #IF# HBIT(/RP(/RP(/P/)/)/) #THEN#
          #BEGIN#  #COMMENT# HAVE TO SPLIT A PAGE, CASE 3.,
            AUX1 .= RP(/P/).,
            RP(/P/) .= RP(/AUX1/).; HBIT(/P/) .= #TRUE#.,
            RP(/AUX1/).=LP(/RP(/P/)/).; HBIT(/AUX1/) .= #FALSE#.,
            LP(/RP(/P/)/) .= AUX1.; HBIT(/RP(/P/)/) .= #FALSE#
          #END#
        #END#
        #ELSE# #GOTO# DONE
      #END#
     #END#
    #ELSE# #IF# LP(/P/) = 0 #THEN#
      #BEGIN# #COMMENT#  INSERT KEY TO LEFT OF LEAF.,
        GETQ(AUX1).;  KEY(/AUX1/) .= X.,
        LP(/AUX1/) .= 0.;  RP(/AUX1/) .= P.;  HBIT(/AUX1/) .= #TRUE#.,
        P .= AUX1
      #END#
    #ELSE#
      #BEGIN# #COMMENT#  INSERT X IN LEFT SUBTREE.,
        INSERT(LP(/P/)).,
        #IF# HBIT(/LP(/P/)/) #THEN#
        #BEGIN# #IF# HBIT(/RP(/LP(/P/)/)/) #THEN#
          #BEGIN# #COMMENT#  HAVE TO SPLIT PAGE, CASE 4.,
            AUX1 .= RP(/LP(/P/)/).,
            RP(/LP(/P/)/) .= LP(/AUX1/).;  HBIT(/LP(/P/)/) .= #FALSE#.,
            LP(/AUX1/) .= LP(/P/).,
            LP(/P/) .= RP(/AUX1/).,
            RP(/AUX1/) .= P.; P .= AUX1
          #END#
        #END#
        #ELSE# #GOTO# DONE
      #END#
  #END#  OF INSERT PROCEDURE, NOTE THAT THIS PROCEDURE ASSUMES
         A NONEMPTY TREE.,
```

```
#IF# P = 0 #THEN#
  #BEGIN#  #COMMENT#  PUT ROOT NODE INTO EMPTY TREE.;
    GETQ(P).;  LP(/P/) .= RP(/P/) .= 0.;
    HBIT(/P/) .= #FALSE#.;  KEY(/P/) .= X.;
  #END#
#ELSE#
  #BEGIN# INSERT(P).;
    #IF# HBIT(/P/) #THEN#
      #BEGIN# #IF# HBIT(/RP(/P/)/) #THEN#
        #BEGIN#  #COMMENT#  HAVE TO SPLIT ROOT PAGE.;
          AUX1 .= RP(/P/).;  RP(/P/) .= LP(/AUX1/).;
          HBIT(/P/) .= #FALSE#.;  LP(/AUX1/) .= P.;
          HBIT(/AUX1/) .= #FALSE#.;  P .= AUX1
        #END#
      #END#
  #END#.;
DONE..
#END# OF PROCEDURE INS
```

```
  #PROCEDURE# DEL(X,P,XNOTINTREE,LP,RP,KEY,HBIT,ADDQ).;
  #VALUE# X.;
    #INTEGER# P.; #REAL# X.; #LABEL# XNOTINTREE.;
    #INTEGER# #ARRAY# LP, RP.;
    #ARRAY# KEY.; #BOOLEAN# #ARRAY# HBIT.;
    #PROCEDURE# ADDQ.;
    #BEGIN# #INTEGER# AUXX,AUXD.;

#COMMENT# RECURSIVE B-TREE DELETION ALGORITHM.;
  #PROCEDURE# DELETE(P).; #INTEGER# P.;
    #BEGIN# #IF# X #GREATER# KEY(/P/) #AND# RP(/P/) #NOT  EQUAL# 0
      #THEN# #BEGIN# DELETE(RP(/P/)).;
        #IF# HBIT(/P/) #THEN#
          #BEGIN# #COMMENT# CASE 3.;
            RP(/P/) .:= AUXD.; HBIT(/P/) .= #FALSE#.;
            #GOTO# QUIT
          #END#
        #ELSE# #IF# HBIT(/LP(/P/)/) #THEN#
          #BEGIN# #COMMENT# CASE 1B.;
            RP(/P/) .= AUXD.; AUXD .= LP(/P/).;
            LP(/P/) .= RP(/RP(/AUXD/)/).; RP(/RP(/AUXD/)/) .= P.;
            P .= RP(/AUXD/).; RP(/AUXD/) .= LP(/P/).;
            LP(/P/) .= AUXD.; HBIT(/AUXD/) .= #FALSE#.;
            #GOTO# QUIT
          #END#
        #ELSE# #BEGIN# #COMMENT# CASE 1A.;
            RP(/P/) .= AUXD.; AUXD .= LP(/P/).;
            LP(/P/) .= RP(/AUXD/).; RP(/AUXD/) .= P.;
            HBIT(/AUXD/) .= #TRUE#
          #END#
      #END#
      #ELSE# #IF# X #LESS# KEY(/P/) #AND# LP(/P/) #NOT  EQUAL# 0
        #THEN# #BEGIN# DELETE(LP(/P/)).;
          #IF# HBIT(/P/) #THEN#
            #BEGIN# #COMMENT# CASE 4.;
              #IF# HBIT(/LP(/RP(/P/)/)/) #THEN#
                #BEGIN# #COMMENT# CASE 4A.;
                  LP(/P/) .= AUXD.; HBIT(/P/) .= #FALSE#.;
                  AUXD .= LP(/RP(/P/)/).; LP(/RP(/P/)/) .= RP(/AUXD/).;
                  RP(/AUXD/) .=RP(/P/).; RP(/P/) .= LP(/AUXD/).;
                  LP(/AUXD/) .= P.; P .= AUXD.;
                  #GOTO# QUIT
                #END#
              #ELSE# #BEGIN# #COMMENT# CASE 4B.;
                  LP(/P/) .= AUXD.; AUXD .= RP(/P/).;
                  RP(/P/) .= LP(/AUXD/).; LP(/AUXD/) .= P.;
                  P .= AUXD.; #GOTO# QUIT
                #END#
            #END#
              #ELSE# #IF# HBIT(/RP(/P/)/) #THEN#
                #BEGIN# #COMMENT# CASE 2B.;
                  LP(/P/) .= AUXD.; AUXD .= RP(/P/).;
                  RP(/P/) .= LP(/AUXD/).; LP(/AUXD/) .= P.;
                  HBIT(/AUXD/) .= #FALSE#.;
                  P .= AUXD.; #GOTO# QUIT
                #END#
              #ELSE# #BEGIN# #COMMENT# CASE 2A.;
```

```
              LP(/P/) .= AUXD.,  HBIT(/P/) .= ≠TRUE≠.,
              AUXD .= P
            ≠END≠
    ≠END≠
  ≠ELSE≠ ≠IF≠ X = KEY(/P/) ≠THEN≠
    ≠BEGIN≠  ≠COMMENT≠ HAVE FOUND X.,
      AUXX .= P.,
      ≠COMMENT≠  NOW FIND NEXT LARGEST KEY, UNLESS X IS A LEAF.,
      ≠IF≠ RP(/P/) ≠NOT  EQUAL≠ 0 ≠THEN≠ ≠GOTO≠ L ≠ELSE≠ ≠GOTO≠ LEAF
    ≠END≠
  ≠ELSE≠ ≠BEGIN≠ LEAF.,  ≠IF≠ AUXX = 0 ≠THEN≠ ≠GOTO≠ XNOTINTREE.,
      ≠COMMENT≠  FOUND LEAF TO REPLACE X AND TO BE DELETED.,
      KEY(/AUXX/) .= KEY(/P/)., AUXD .= RP(/P/).,
      ADDQ(P).,
      ≠IF≠ AUXD ≠NOTEQUAL≠ 0 ≠THEN≠ ≠BEGIN≠ P .= AUXD.,≠GOTO≠ QUIT.,
                                          ≠END≠.,
    ≠END≠
≠END≠ OF DELETE.,


  AUXX .= 0.,
  ≠IF≠ P = 0 ≠THEN≠ ≠GOTO≠ XNOTINTREE
    ≠ELSE≠ ≠BEGIN≠ DELETE(P).,   P.= AUXD ≠END≠.,
QUIT.. ≠END≠ OF DEL
```

BIBLIOGRAPHY

[1] Adelson-Velskii, G.M. and Landis, E.M., An Information Organization Algorithm, DANSSSR, No. 2, 1962.

[2] Bayer, R. and McCreight, E.M., Organization and Maintenance of Large Ordered Indices, Record of the 1970 ACM SICFIDET Workshop on Data Description and Access, (Nov. 1970), Houston, Texas pp. 107-141.

[3] Coffman, E.G. and Eve, J., File Structures Using Hashing Functions, CACM 13, 7(July 70), pp. 427-436.

[4] Foster, C.C., Information Storage and Retrieval Using AVL Trees, Proc. ACM 20th Nat'l. Conf. (1965), pp. 192-205.

[5] Knott, G.D., A Balanced Tree Structure and Retrieval Algorithm, Proc. of the 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] Landauer, W.I., The Balanced Tree and Its Utilization in Information Retrieval. IEEE Trans. on Electronic Computers, Vol. EC-12, No. 6, December 1963.

[8] Sussenguth, E.H., Jr., The Use of Tree Structures for Processing Files, Comm. ACM, Vol. 6, No. 5, May 1963.