1971

Symmetric Binary B-Trees: Data Structure and Algorithms for Random and Sequential Information Processing

Rudolf Bayer

Report Number:
71-054

SYMMETRIC BINARY B-TREES: DATA STRUCTURE AND ALGORITHMS FOR RANDOM AND SEQUENTIAL INFORMATION PROCESSING*

Rudolf Bayer
Computer Sciences
Purdue University
Lafayette, Indiana 47907

CSD TR 54

November 1971

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. 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 work was partially supported by an NSF grant.

SYMMETRIC BINARY B-TREES:
DATA STRUCTURE AND ALGORITHMS FOR
RANDOM AND SEQUENTIAL INFORMATION PROCESSING

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], [4], [5] 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.

Readers familiar with [2] and [3] will 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 subsequent modification of the B-trees of [2]. Binary B-trees are derived in a straightforward way from B-trees, they do exhibit, however, a surprising asymmetry: 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 somewhat artificial distinction between left and right arcs 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.

Definition:

Symmetric binary B-trees (henceforth simply called 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) All leaves are at the same δ-level.
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:

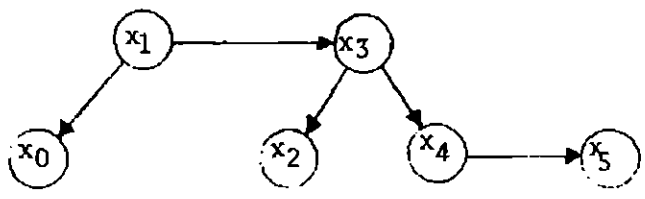
1) If the tree is empty, do nothing
2) Traverse left subtree
3) Visit root
4) Traverse right subtree.

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

Number of Nodes and Height of a B-tree:

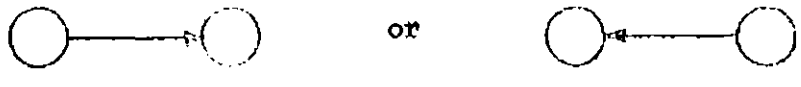
Let the height h of a B-tree be the maximal number of nodes in any path from the root to a leaf.

Then an example of a 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:

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

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 B-tree of odd height h we obtain:

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

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

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 B-tree of height h, then we obtain as bounds for N:

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

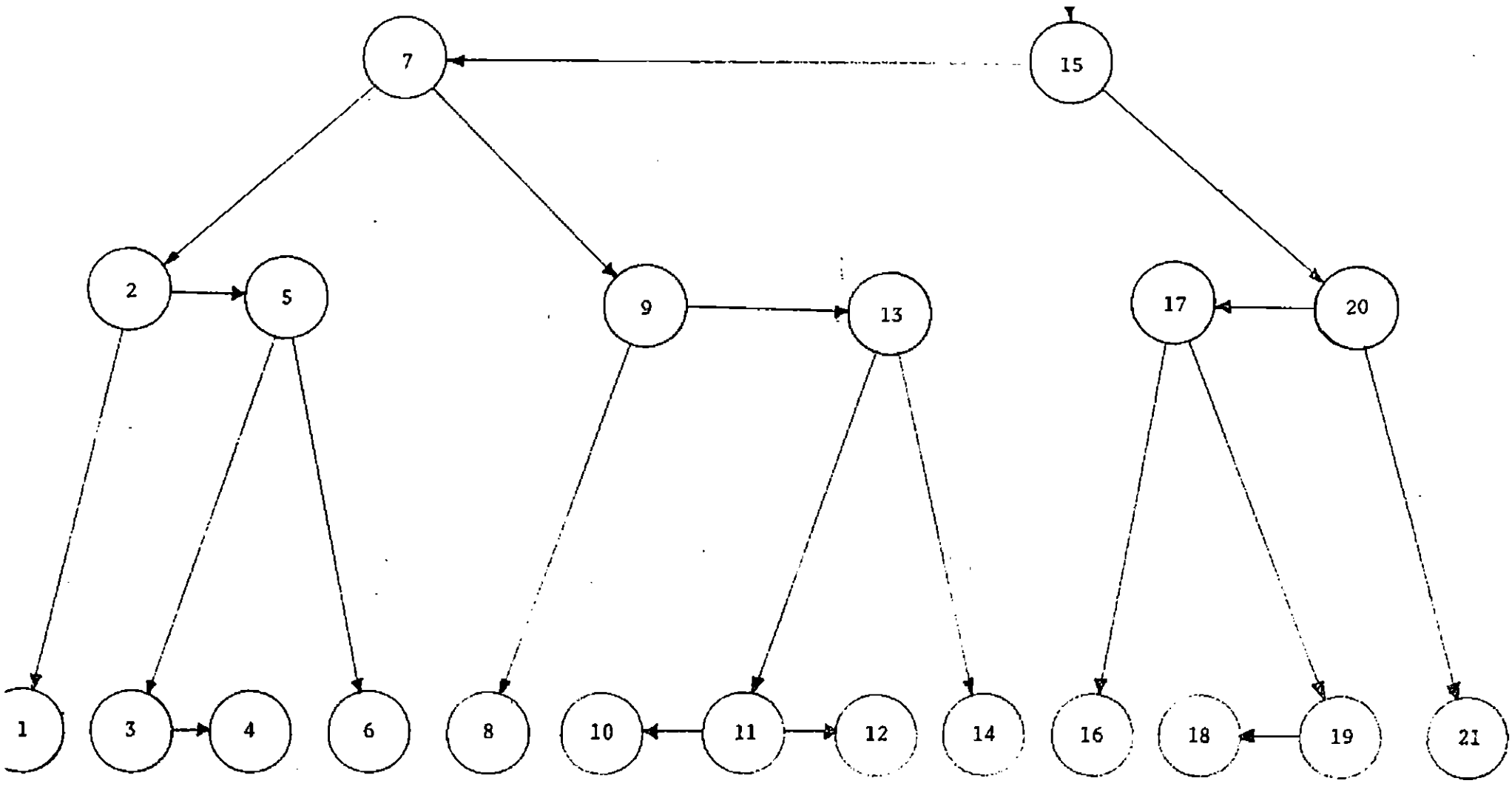


Fig. 1: Example of a symmetric binary B-tree

Taking logarithms we obtain:

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

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

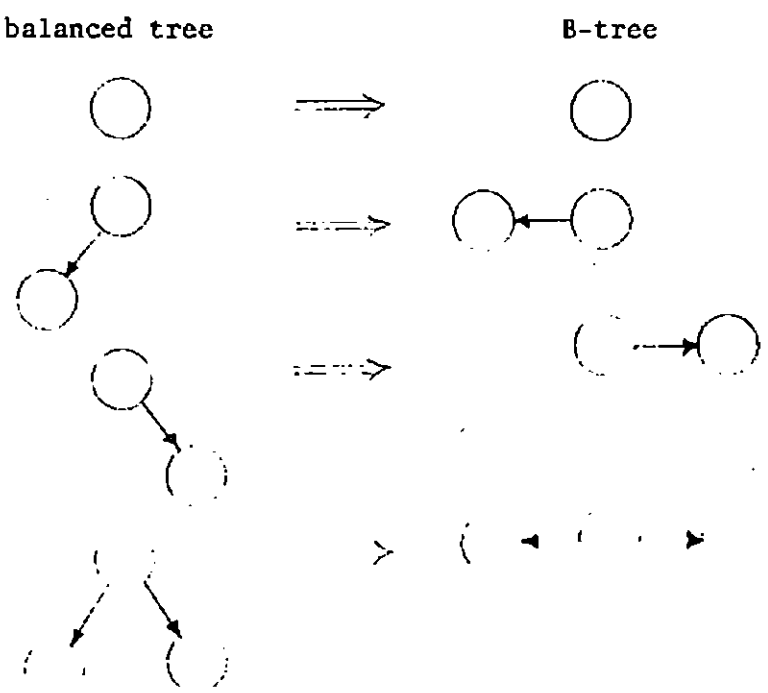
and consequently as sharp bounds for the height h of a B-tree with N nodes:

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

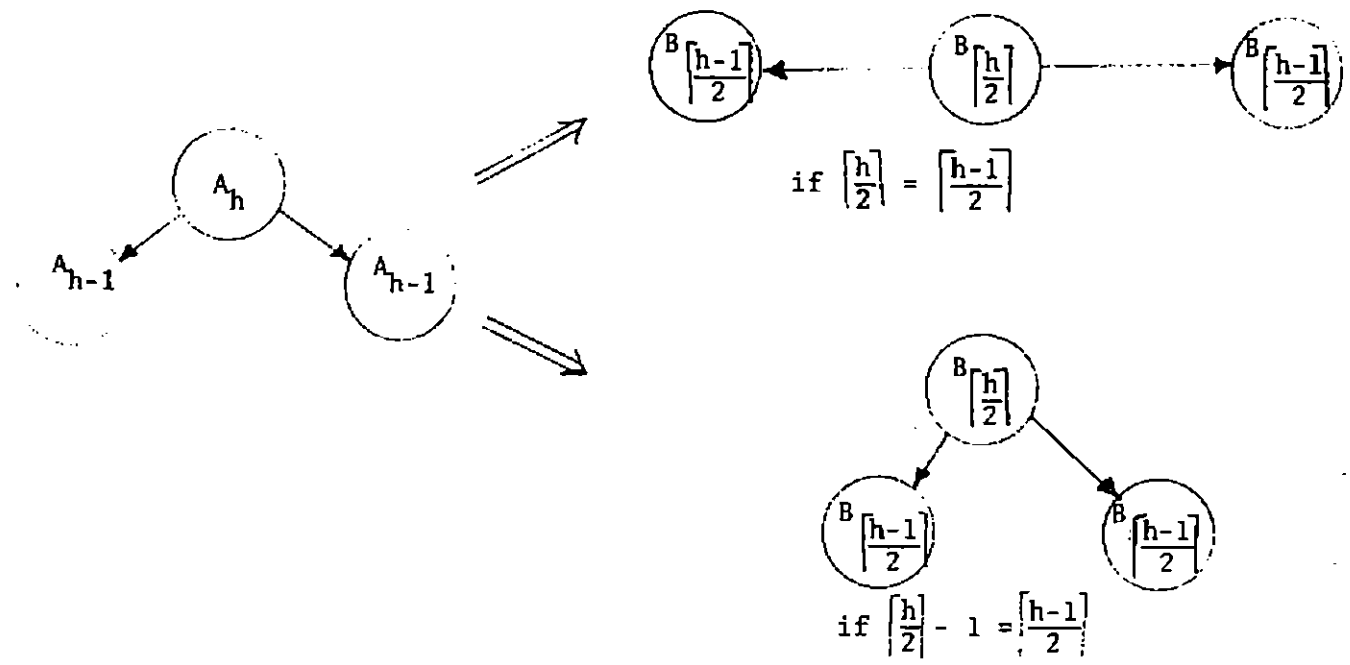
B-trees and balanced trees:

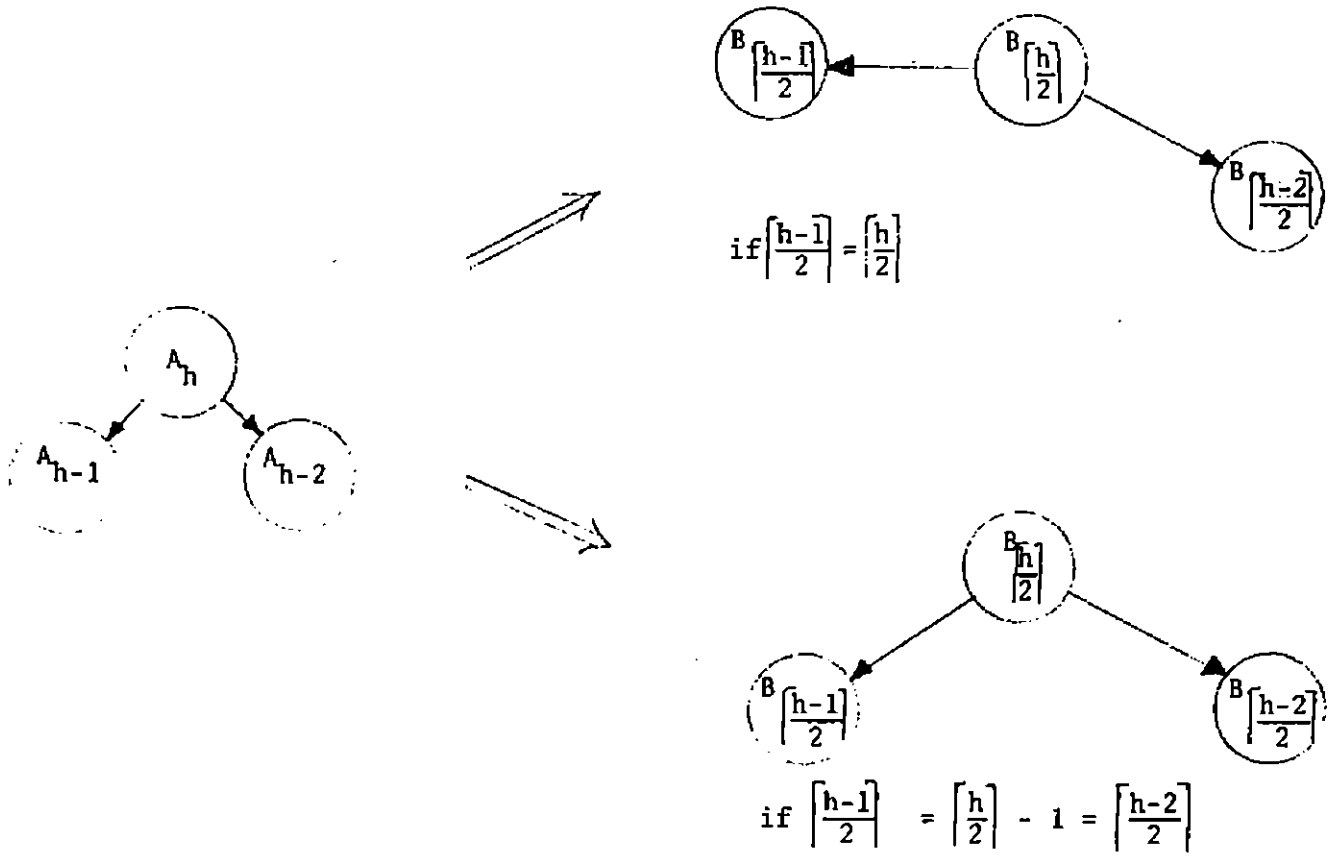
Theorem: The class of B-trees properly contains the class of balanced trees.

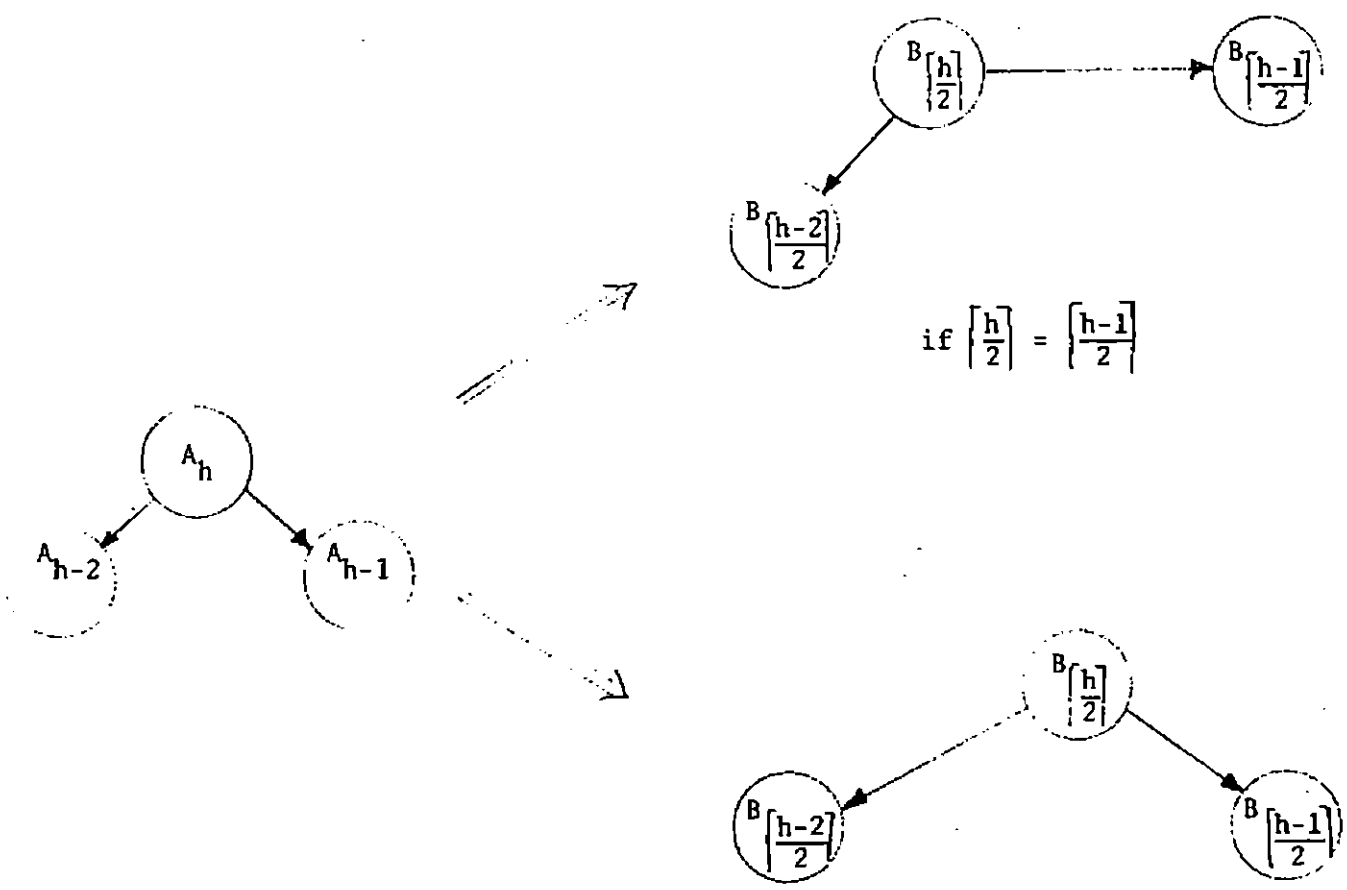
Proof: Let the δ-height of a B-tree be defined as the number of "levels" in a B-tree, i.e. as the number of δ-arcs plus one in any path from the root to a leaf. Then a balanced tree of height h can be transformed into a B-tree of δ-height $\left\lceil \frac{h}{2} \right\rceil$ by simply labelling the arcs as δ-arcs or ρ-arcs. The proof is by induction on h. For h = 1,2 we label as follows:



and in general, letting A_h stand for the root of a balanced tree of height h and B_ℓ for the root of a B-tree of δ-height ℓ we obtain:







The proper containment can be seen from the B-tree in Fig. 1 which is not a balanced tree. This completes the proof.

Figures 2 and 3 show a balanced tree and a B-tree obtained by labelling the arcs according to the algorithm implied in the proof.

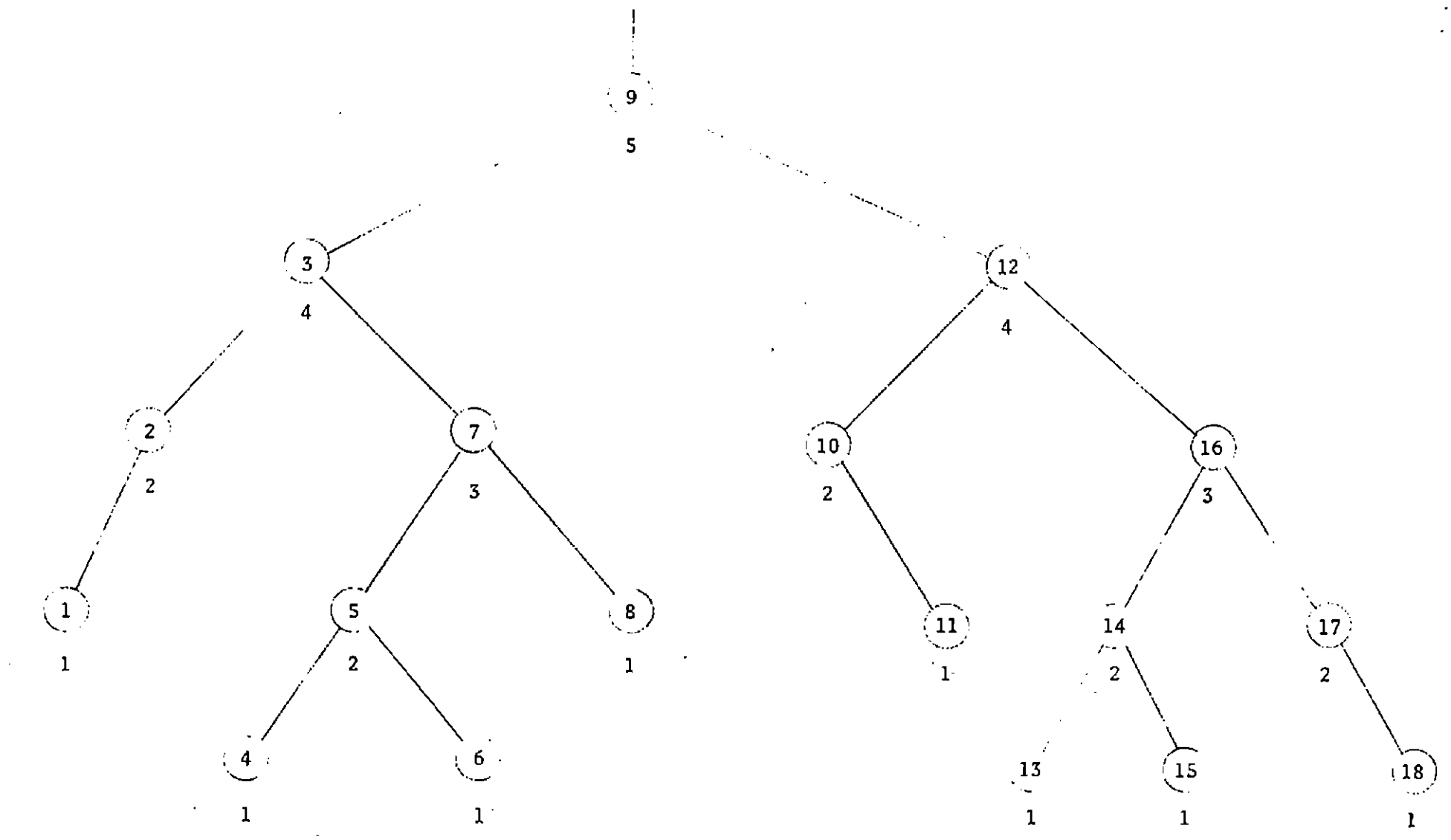


Fig. 2: A balanced tree

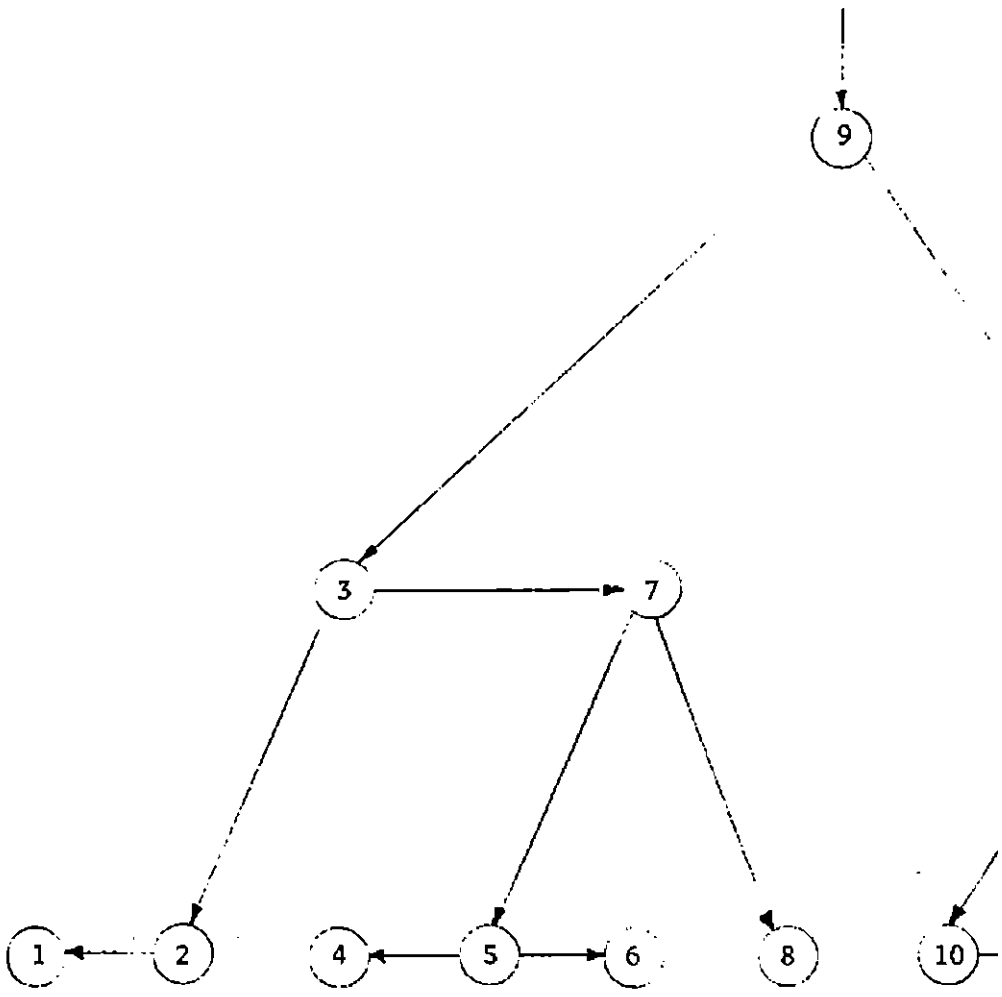
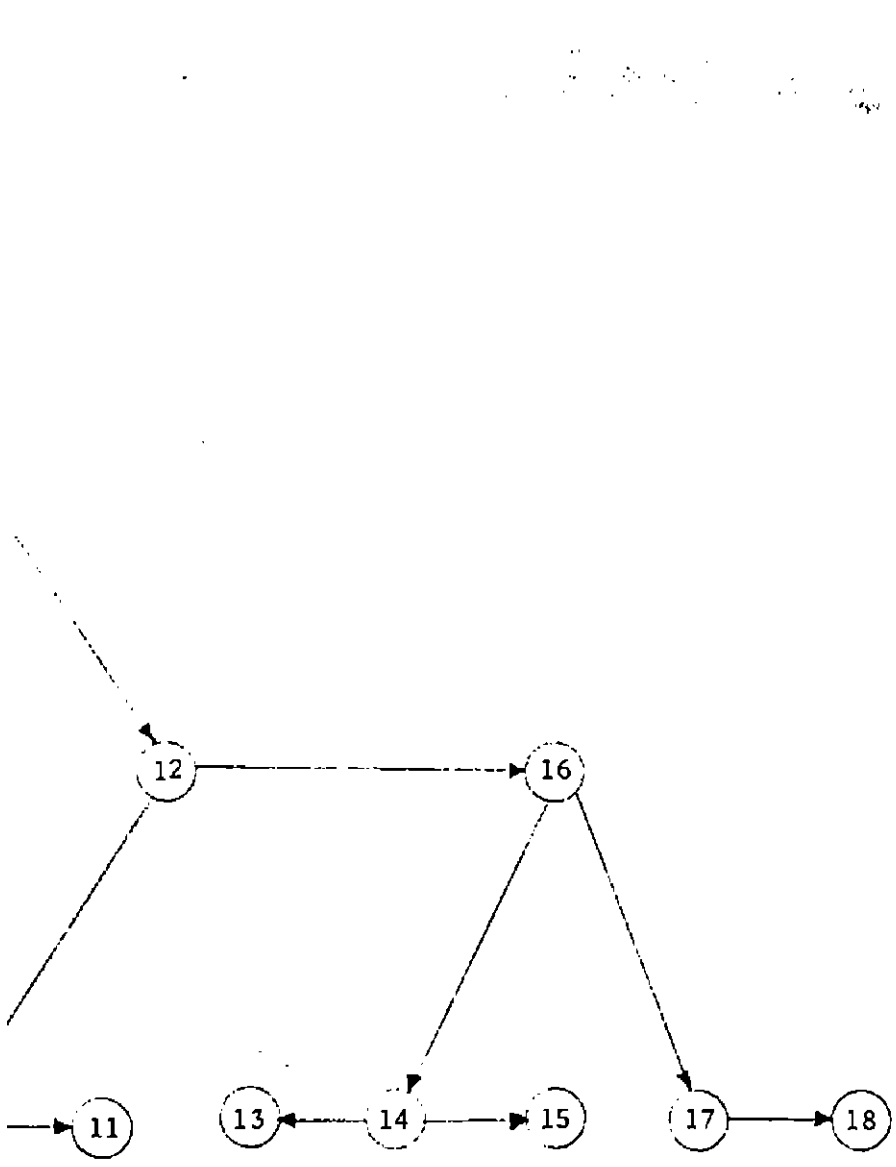


Fig. 3: The balanced tree of Fig.



2 considered as a B-tree

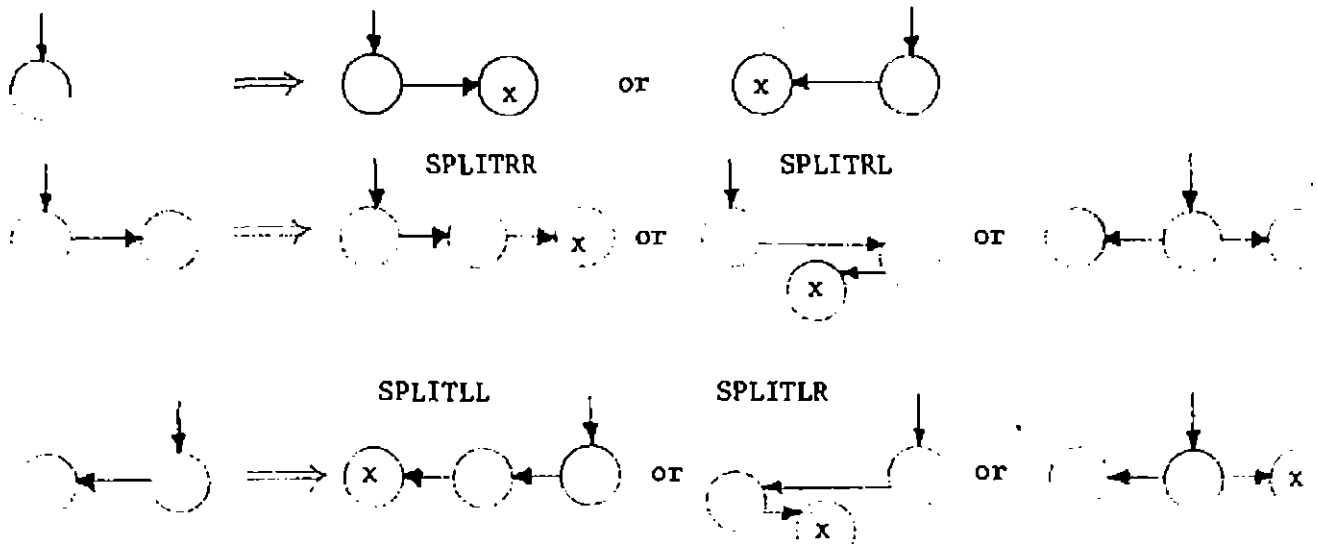
The upper bound on the height of a B-tree obtained in (1) is approximately 2 $\log_2$ (N) instead of 1.5 $\log_2$ (N) for the height of a balanced tree.[4]. This means that the upper bound for the retrieval time is better for balanced trees than for B-trees. On the other hand, these same bounds and the fact that balanced trees are a proper subclass of B-trees also suggest that less work should be required to update B-trees than to update balanced trees.

Maintenance Algorithms:

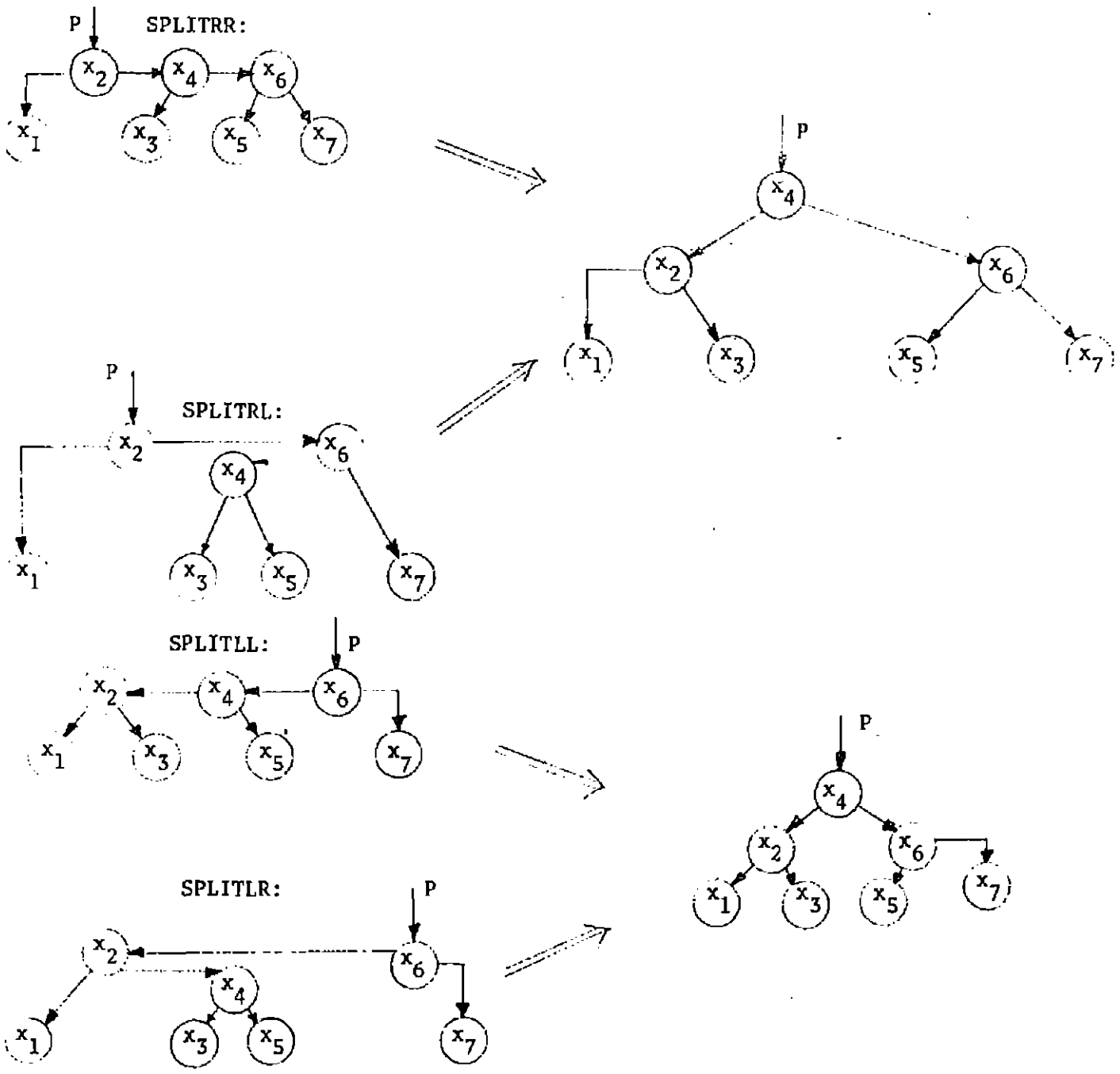
We now consider the algorithms for maintaining B-trees if keys are inserted and deleted randomly. The algorithm to retrieve keys is straightforward and will not be described here.

Insertion Algorithm:

A new key x is inserted into the tree by attaching it with a new p-arc at the lowest δ-level. x is attached exactly at that place where the retrieval algorithm for x tries to proceed along a non-existing arc or "falls out of the tree." The possible cases are indicated by the following illustrations:



Whenever two successive ρ-arcs arise the tree must be modified according to one of the following four cases, which are named like the Algol procedures performing those modifications in our implementation. It is assumed that $x_i < x_{i+1}$ for all keys. An arc of rectangular shape can be a ρ-arc or a δ-arc.

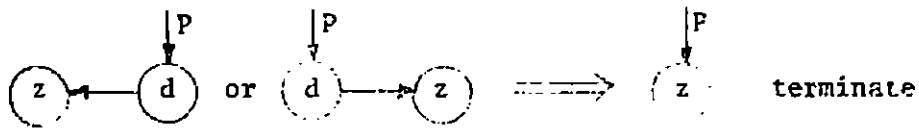


In all four cases P is a left or a right δ-arc (pointer) and must then become a left or right ρ-arc respectively. This may, of course, gives rise to successive ρ-arcs at the next δ-level closer to the root, requiring again one of the modifications just described to remove successive ρ-arcs from the tree. These modifications may recursively propagate along the retrieval path all the way up to the root of the tree. It is crucial to observe, however, that these modifications can propagate only along the retrieval path and will not affect any other parts of the tree. Thus the total work required to modify the tree in order to remove successive ρ-arcs is at worst proportional to the length of the retrieval path for x, i.e. to $2 \log_2 (N+2) - 2$.

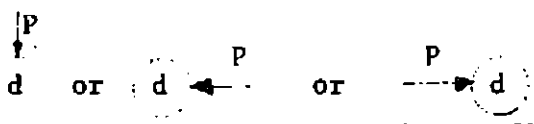
Deletion Algorithm:

To delete an element x from the tree, we first have to locate it and to replace it by the next smallest (next largest) key, say y, in the tree. y is found easily proceeding from x one step along the left (right) pointer and then along the right (left) pointer as long as possible. The node containing y originally is then replaced by a dummy node d which we will delete from the tree in one of the following ways. Note that at any one time d has at most one successor. We use d only as a conceptual device for illustrative purposes. In the implementation the dummy node d is not physically represented, instead the pointer P simply points "through" to the successor of d.

Case A1: at the lowest δ-level:

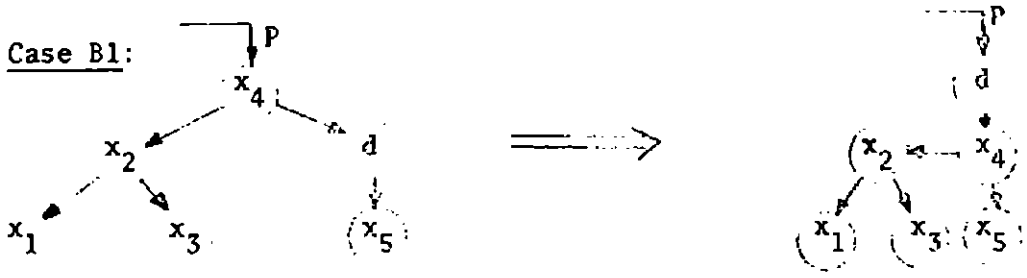


Case A2: at the lowest δ-level, if d is a leaf:



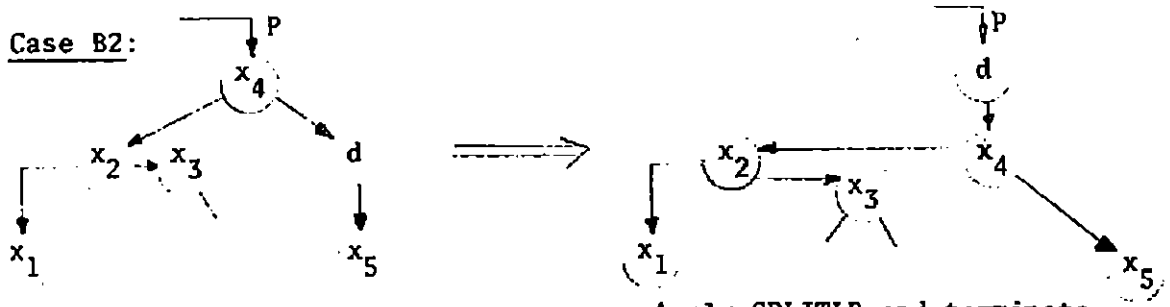
Set P := 0 and proceed according to one of the following cases.

Case B1:



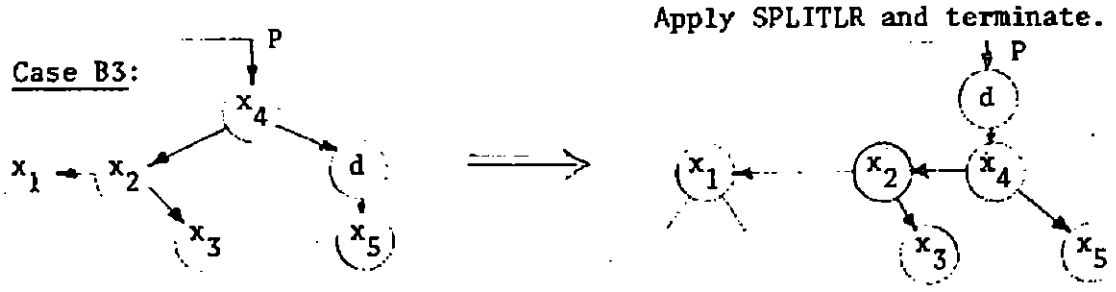
Continue recursive deletion.

Case B2:



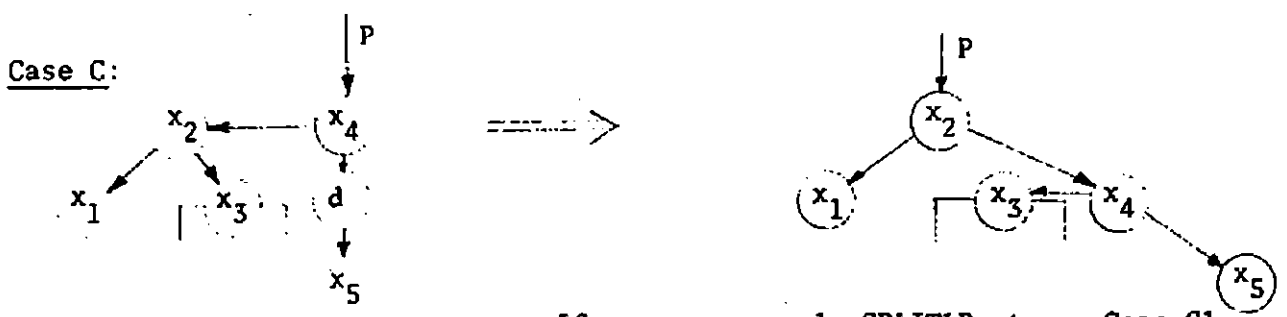
Apply SPLITLR and terminate.

Case B3:



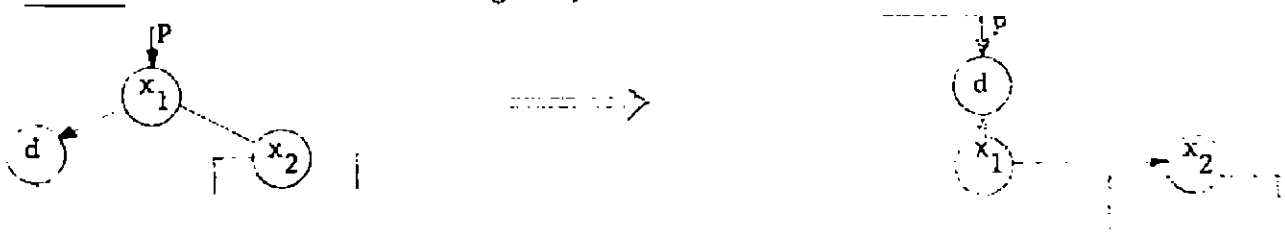
Apply SPLITLL and terminate.

Case C:



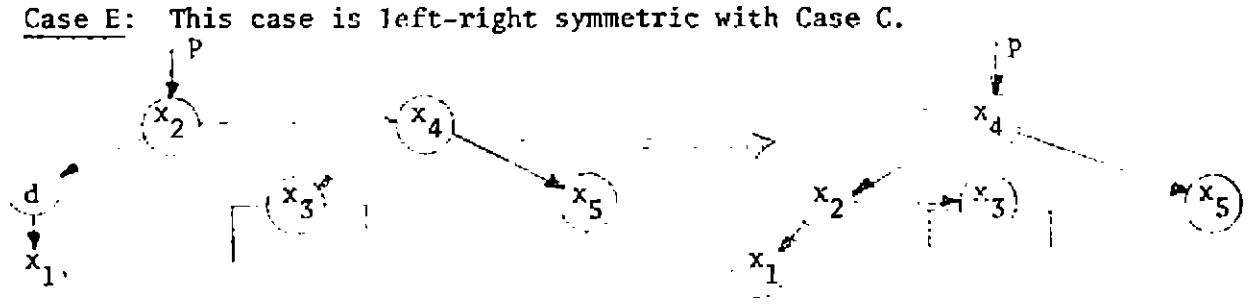
If necessary apply SPLITLR at x_4, Case C1,
or if necessary apply SPLITLL at x_4, Case C2,
or do nothing further, Case C3.
In all three cases terminate.

Case D: This case is left-right symmetric with Case B.



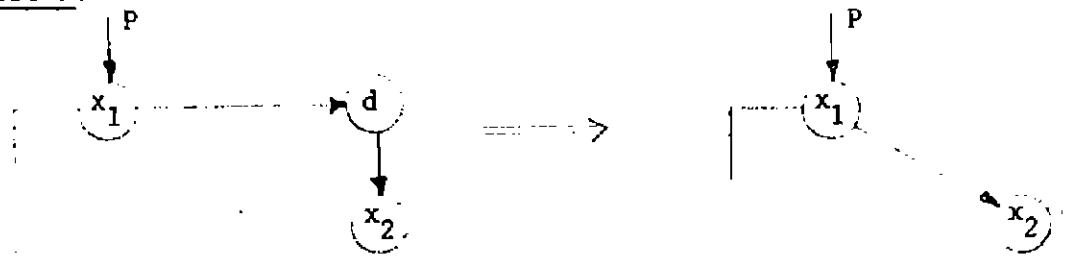
Apply SPLITRL at x_1 if possible and terminate, Case D2,
or apply SPLITRR at x_1 if possible and terminate, Case D3,
or continue recursive deletion process, Case D1.

Case E: This case is left-right symmetric with Case C.



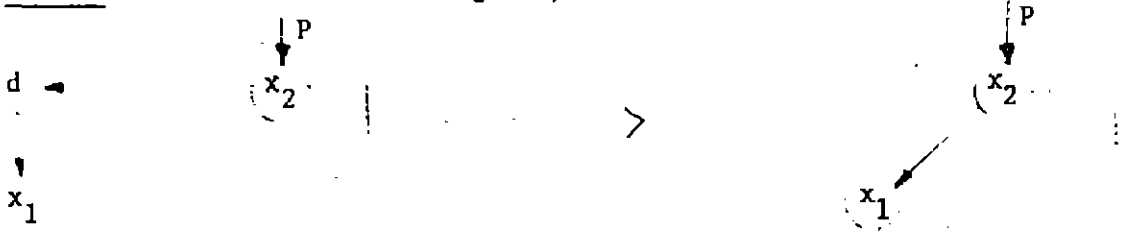
Apply SPLITRL at x_2 if possible, Case E1.
Apply SPLITRR at x_2 if possible, Case E2.
Do nothing, Case E3.
In all three cases terminate.

Case F:



Terminate.

Case G: This case is left-right symmetric with Case F.



Terminate.

Note that the recursive deletion process terminates in all cases except A2, B1, and D1. If 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. Also, 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 height of a 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 B-tree, i.e. to $\log_2(N+2)$ where N is the number of keys in the tree.

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 p-pointers for $n = 2,3,4,\ldots$ before requiring any modification or "rebalancing" of the tree. This would require less rebalancing, but performance in time log(N) 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.

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 dummy node d 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 less 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 notgreater KEY[P] and LP[P] notequal 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 notless KEY[P] and RP[P] notequal 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, P


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 notequal 0 then goto QUIT
   end
end OF SYMDELETE;


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

BIBLIOGRAPHY

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

[2] Bayer, R. and McCreight, E.M., Organization and Maintenance of Large Ordered Indexes, Record of the 1970 ACM SICFIDET Workshop on Data Description and Access, (Nov. 1970), Houston, Texas pp. 107-141 (to appear in Acta Informatica, December 1971).

[3] Bayer, R., Binary B-trees for Virtual Memory, Proceedings of 1971 ACM SIGFIDET Workshop on Data Description, Access and Control, pp. 219-235, (Nov. 11-12, 1971), San Diego, edited by E.F. Codd and A.L. Dean.

[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).