SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 15(10), 925–941 (OCTOBER 1985)

Efficient Algorithms to Create and Maintain Balanced and Threaded Binary Search Trees

S. SITHARAMA IYENGAR AND HSI CHANG
Department of Computer Science, Louisiana State University, Baton Rouge, LA 70803, U.S.A.

SUMMARY

The algorithm proposed by Chang and Iyengar[1] to perfectly balance binary search trees has been modified to not only balance but also thread binary search trees. Threads are constructed in the same sequence as normal pointers during the balancing process. No extra workspace is necessary, and the running time is also linear for the modified algorithm. Such produced tree structure has minimal average path length for fast information retrieval, and threads to facilitate more flexible and efficient traversing schemes. Maintenance and manipulation of the data structure are discussed and relevant algorithms given.

KEY WORDS Algorithm Binary search tree In-order traversal Balanced tree Threaded tree

INTRODUCTION

It is nice to be able to balance binary search trees once in a while in a dynamical information environment, so that the performance of searching can be in control. Also, it may be of interest if we use the $(n + 1)$ null pointers out of the total $2n$ pointers of the balanced tree in the formulation of a threaded binary tree by replacing the right null pointer with a pointer to its immediate successor and the left null pointer with a pointer to its immediate predecessor according to the traversal order. Usually we call these replaced pointers threads.

The advantage of threads is to provide a more flexible and efficient way for traversal, because we can easily determine the predecessor and the successor for any node in a threaded tree and traverse either forward or backward from that arbitrary point without incurring the overhead of using a stack mechanism as in the case of unthreaded trees. With such a data representation, random access and sequential retrieval of information are feasible and could coexist. Although we benefit from the threaded tree structure, we have to make a little more effort, however, to insert a node or delete a node from a threaded tree because threads must be maintained as well as normal pointers.

In an earlier paper, Chang and Iyengar presented an algorithm[1] to perfectly balance binary search trees in linear time. This algorithm forms balanced left half and right half trees in a parallel fashion, thus making the execution more efficient. Here, we shall first describe a modified algorithm that not only balances but also threads binary search trees, and then tackle the problems of maintaining such a data structure. Since the algorithm is essentially based on the previous tree-balancing algorithm, and the portion dealing with balancing is about the same, readers can obtain detailed knowledge on tree balancing from related material.[1–3, 7, 8]

0038–0644/85/100925–17$01.70

Received 19 August 1982
Revised 12 December 1984

BALANCING AND THREADING BINARY SEARCH TREES

To differentiate threads from normal pointers, one way is to append a tag bit to the pointer to specify whether the associated pointer is a normal pointer or a thread. The other way is to store a thread with a negative sign.[4, 5] By adding tag bits, extra memory space is required and more fields need to be monitored. The negative-sign method, on the other hand, is simpler for maintenance and does not need additional storage, but it requires sign conversion from negative to positive every time a thread is used to link to another node. In this paper, we choose the tag-bit approach for its explicitness, whereas on other occasions one might use the negative-sign method for its compactness.

With such a data structure, each node contains basic fields: KEY, left subtree pointer LSON, right subtree pointer RSON, and tag bit fields LBIT and RBIT associated with LSON and RSON, respectively. A tag bit is 'on', with value 1, when the associated pointer is a normal pointer and 'off', with value 0, when the associated pointer is a thread. In convention, a head node which serves as the predecessor of the first node and the successor of the last node is used to impose a circular structure upon the tree structure. In our algorithm, however, instead of keeping a head node, we set a pointer HEAD which points to the root node of the binary search tree, and let the left thread of the first node, which contains the smallest key value, and the right thread of the last node, which contains the largest key value, take values 0 to mark the beginning and the end of an in-order sequence.

The algorithm to balance and thread binary search trees is formally described in the procedure BALTHR (see Figure 1) adopting a PL/I-like language. BALTHR takes as input a random binary search tree, establishes operational environment for subprocedures TRAVBIND and GROW, and returns a completely balanced and threaded binary search tree as a result.

```
procedure BALTHR(HEAD, LSON, RSON, LBIT, RBIT):

    N <-- 0 /* initialize item counter */
    /* traverse and bind the original tree */
    call TRAVBIND(HEAD)
    case
     : N = 0 :
         print('empty tree')
         return

     : N = 1 :
         HEAD <-- LINK(1)
         LSON(HEAD) <-- RSON(HEAD) <-- 0
         LBIT(HEAD) <-- RBIT(HEAD) <-- 0
```

```
    return

: N = 2 :
    HEAD <-- LINK(1)
    LSON(HEAD) <-- 0
    LBIT(HEAD) <-- 0
    RSON(HEAD) <-- LINK(2)
    RBIT(HEAD) <-- 1
    LSON(LINK(2)) <-- HEAD
    RSON(LINK(2)) <-- 0
    LBIT(LINK(2)) <-- RBIT(LINK(2)) <-- 0
    return

: otherwise :
    LINK(0) <-- LINK(N+1) <-- 0 /* mark boundaries */
    M <-- ⌊(N+1)/2⌋  /* find folding factor M */
    HEAD <-- LINK(M) /* locate HEAD through M */
    if N = 2 * M
       then do /* when N is even */
            M <-- M - 1
            /* balance the left half tree */
            call GROW(1, M-2)
            /* put the node bound to M as a left
               descendant of its successor */
            LSON(LINK(M)) <-- HEAD
            RSON(LINK(M)) <-- LINK(M+1)
            LBIT(LINK(M)) <-- RBIT(LINK(M)) <-- 0
            LSON(LINK(M+1)) <-- LINK(M)
            LBIT(LINK(M+1)) <-- 1
            end
       else /* when N is odd */
            /* balance the left half tree */
            call GROW(1, M-1)
    /* left subtree pointer is returned via ANSL */
    LSON(HEAD) <-- ANSL
```

```
      LBIT(HEAD) <-- 1
      /* right subtree pointer is returned via ANSR */
      RSON(HEAD) <-- ANSR
      RBIT(HEAD) <-- 1
      return
end
recursive procedure TRAVBIND(T):
/* TRAVBIND performs inorder traversal and binding
   between ascending sequence # and item #; local
   variable T points to the node to be visited */
    /* return when a null pointer is encountered */
    if T = 0  then return
    call TRAVBIND(LSON(T)) /* go down left branch */
    N <-- N + 1 /* count total number of nodes visited */
    LINK(N) <-- T /* bind sequence # N to item # T
                     of the Nth node via LINK */
    call TRAVBIND(RSON(T)) /* go down right branch */
    return
    end TRAVBIND
recursive procedure GROW(HIGH, LOW):
/* GROW concurrently balances and threads the left half
   and the right half tree.  ROOTL, ROOTR, LOW, HIGH,
   and MID are local variables */
   case
    : LOW > HIGH : /* when null nodes are encountered */
          /* return null pointers */
          ANSL <-- ANSR <-- 0
          return
    : LOW = HIGH : /* when terminal nodes are met */
          /* locate terminal nodes via LINK and build
             threads */
          ANSL <-- LINK(LOW)
          ANSR <-- LINK(LOW+N)
```

```
        LSON(ANSL) <-- LINK(LOW-1)
        LSON(ANSR) <-- LINK(LOW+M-1)
        LBIT(ANSL) <-- LBIT(ANSR) <-- 0
        RSON(ANSL) <-- LINK(LOW+1)
        RBIT(ANSL) <-- RBIT(ANSR) <-- 0
        return

: LOW < HIGH : /* when more to be partitioned */
        /* find median of a set */
        MID <-- ⌊(LOW + HIGH)/2⌋
        /* find subtree roots throught MID:
           ROOTL points to the root node in
           the left half tree;
           ROOTR is the counterpart of ROOTL
           in the right half tree */
        ROOTL <-- LINK(MID)
        ROOTR <-- LINK(MID+M)

        /* balance the left subtree */
        call GROW(LOW, MID-1)
        if ANSL = 0
           then do /*change null pointers to threads*/
                LSON(ROOTL) <-- LINK(MID-1)
                LSON(ROOTR) <-- LINK(MID+M-1)
                LBIT(ROOTL) <-- LBIT(ROOTR) <-- 0
                end
           else do /* set left subtree pointers */
                LSON(ROOTL) <-- ANSL
                LSON(ROOTR) <-- ANSR
                LBIT(ROOTL) <-- LBIT(ROOTR) <-- 1
                end

        /* balance the right subtree */
        call GROW(MID+1, HIGH)
        if ANSL = 0
```

```
                then do /*change null pointers to threads*/
                     RSON(ROOTL) <-- LINK(MID+1)
                     RSON(ROOTR) <-- LINK(MID+M+1)
                     RBIT(ROOTL) <-- RBIT(ROOTR) <-- 0
                     end
                else do /* set right subtree pointers */
                     RSON(ROOTL) <-- ANSL
                     RSON(ROOTR) <-- ANSR
                     RBIT(ROOTL) <-- RBIT(ROOTR) <-- 1
                     end

           /* return root node pointers */
           ANSL <-- ROOTL
           ANSR <-- ROOTR
           return
      end
      end GROW

end BALTHM
```

Figure 1

Procedure TRAVBIND is to traverse the original tree in-orderly and bind sequence numbers to the item numbers of the visited nodes. After traversal, we know the ascending sequence of all the nodes, i.e. the ith node in ascending order is the node whose item number is bound to i; hence, the ith node can be located by a direct reference to i. Likewise, the predecessor of the ith node can be determined by linking through $(i - 1)$ and its successor through $(i + 1)$.

Procedure GROW balances and threads the tree through partitioning the set of the sequence numbers obtained from TRAVBIND. Two parameters LOW and HIGH are involved in GROW; LOW is the lower bound and HIGH is the upper bound for each subset to be partitioned. The subtree root is the node bound to the median MID of a subset. The actual process includes raising MID from the subset as a subtree root and then equally splitting the remaining subset into another two subsets each to form a subtree with the number of elements in each subset differing by at most one. Roots of the two such derived subtrees then in turn become the LSON and RSON of the previous root MID. In this way, partitioning continues recursively until the subset is empty (LOW $>$ HIGH) or contains only one element (LOW = HIGH); the former case indicates that a null pointer is reached, and the latter indicates that a terminal node is met. Threads are built for terminal nodes and nodes with single descendants; at the same time pointers are

built during the restructuring process. The completed tree structure is mapped out by substituting sequence numbers with the bound item numbers.

Furthermore, this process is curtailed by folding via a factor M which is the median of a set with odd total number of sequence numbers. The symmetrical property that the ith and $(i + M)$th nodes are counterparts of the left half and right half trees permits us to GROW only half of the set to form a fully balanced and threaded tree. An example is given in Figure 2 to show the process of converting a random tree into a balanced and threaded tree. Now, if the total number of nodes N is even, the tree cannot be divided into equal halves after the root node is singled out. The folding factor M is obtained by $N/2$, and after the root is found by linking through M, we then increase M by 1 for the folding computation. The excess node with the item number associated with the new M is placed as the leftmost node in the right half tree. Figure 3 shows the configuration of a balanced and threaded tree structure with an even total number of nodes.

MAINTAINING THREADED SEARCH TREES

The problems of maintaining a threaded binary search tree include inserting new items into an existing tree and deleting existing items from a tree. The consequences are the dynamical growing or shrinking of the tree structure and the allocation of new nodes from storage and return of freed nodes to storage.

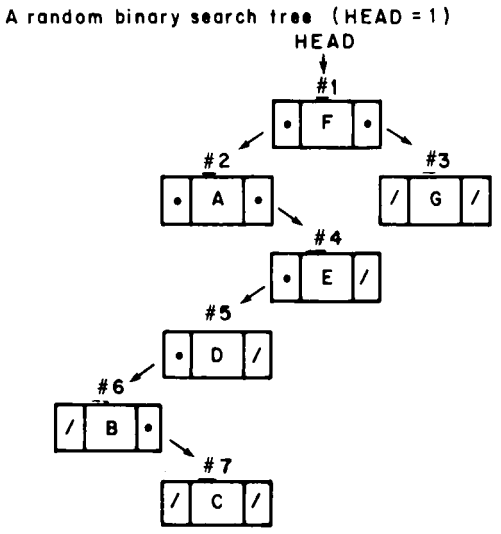
Listed in Figure 4 is the procedure INSERT with an invoking statement. The insertion process always begins at the root node pointed to by HEAD and descends down the branches by comparisons of the key value, denoted by NEWKEY, of the node to be inserted with key values of the visited nodes following the in-order sequence. Assuming that no duplicated key is found, the final step of the process will fall to the left or right side, which must have a thread, of a node; and that is the place where the new node fits into the tree structure.

Subroutines ATTACH_LEFT and ATTACH_RIGHT set pointers and tag bits for the new node and update pointers and tag bits for its direct ancestor. Procedure GETNODE is presumed to provide a pointer NEW to every newly allocated node. Figure 5 shows the placement of a new node.

Removal of an element from a tree is not generally as simple as insertion, since the obsolete node could be the root of a tree or any subtree, and taking threads into account further complicates the issue. It is straightforward only if the node to be deleted is a terminal node, for we could simply cut it loose and change only one pointer. The difficulty lies in removing a node with one direct descendant or two, in which cases we have to figure out ways to circumvent or replace the unwanted nodes.

Details of the deletion algorithm are given in the procedure DELETE (Figure 6) which also starts from HEAD searching for a node containing the key that matches the OLDKEY, the key to be deleted. If OLDKEY is found, subroutine DETACH will be invoked to remove the node.

When left and right tag bits are equal, it means that the node is either a terminal node or one with two direct descendants. A terminal node is deleted by replacing the LSON (RSON) of its right (left) ancestor with the LSON (RSON) of the unwanted node and update the associated bit to indicate that the changed pointer is a thread. A root node with two direct descendants cannot be removed directly, so we decide to replace the relevant information of the unwanted node, excluding pointers and tag bits, with that of its successor S. After replacement, the successor becomes redundant and should also be



An ordered set generated after inorder traversal

Order	1	2	3	4	5	6	7
Item	#2	#6	#7	#5	#4	#1	#3
Key	A	B	C	D	E	F	G

Tree size: $N = 7$
Folding factor: $M = (7+1)/2 = 4$

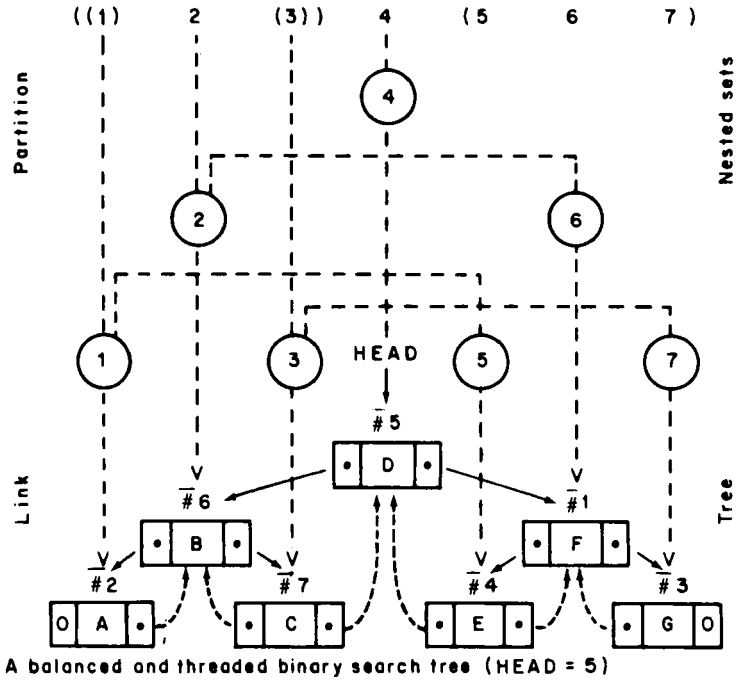


Figure 2. Process of balancing and threading a tree with odd total number of nodes

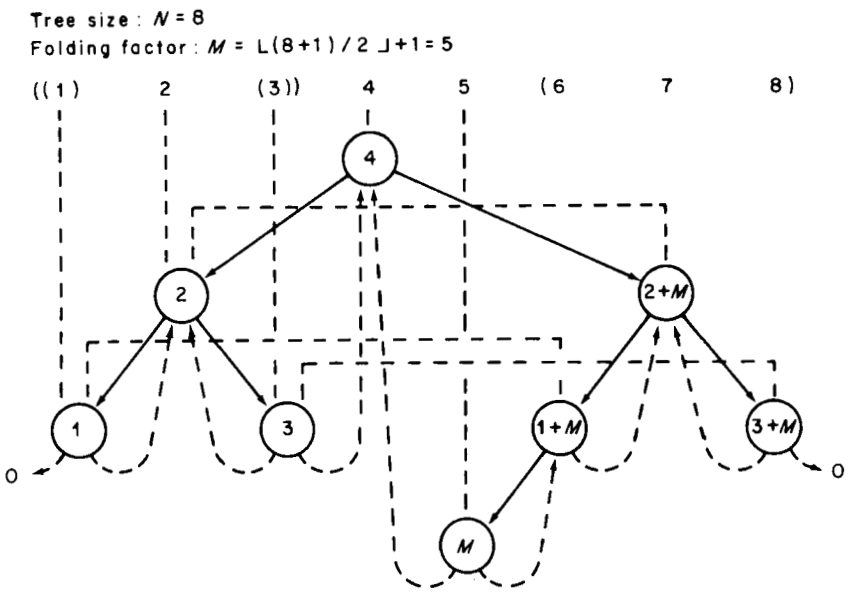


Figure 3. A balanced and threaded tree structure with even total number of nodes

```
call INSERT(HEAD, NEWKEY)
      .
      .
      .
recursive procedure INSERT(T, NEWKEY)
    case
     : T = 0 : /* empty tree */
         call ALLOCATE(NEW)
         T <-- NEW
         KEY(NEW) <-- NEWKEY
         LSON(NEW) <-- RSON(NEW) <-- 0
         LBIT(NEW) <-- RBIT(NEW) <-- 0
         return

     : NEWKEY < KEY(T) :
         if LBIT(T) = 0
            then call ATTACH_LEFT(T, NEWKEY)
            else call INSERT(LSON(T), NEWKEY)
         return

     : NEWKEY > KEY(T) :
```

```
            if RBIT(T) = 0
               then call ATTACH_RIGHT(T, NEWKEY)
               else call INSERT(RSON(T), NEWKEY)
            return

        : NEWKEY = KEY(T) :
            print('duplicated key')
            return
    end
    end INSERT

procedure ATTACH_LEFT(T, NEWKEY)
    call ALLOCATE(NEW)
    LSON(NEW) <-- LSON(T)
    RSON(NEW) <-- T
    LBIT(NEW) <-- RBIT(NEW) <-- 0
    LBIT(T) <-- 1
    T <-- NEW
    return
    end ATTACH_LEFT



procedure ATTACH_RIGHT(T, NEWKEY)
    call ALLOCATE(NEW)
    RSON(NEW) <-- RSON(T)
    LSON(NEW) <-- T
    LBIT(NEW) <-- RBIT(NEW) <-- 0
    RBIT(T) <-- 1
    T <-- NEW
    return
    end ATTACH_RIGHT
```

Figure 4

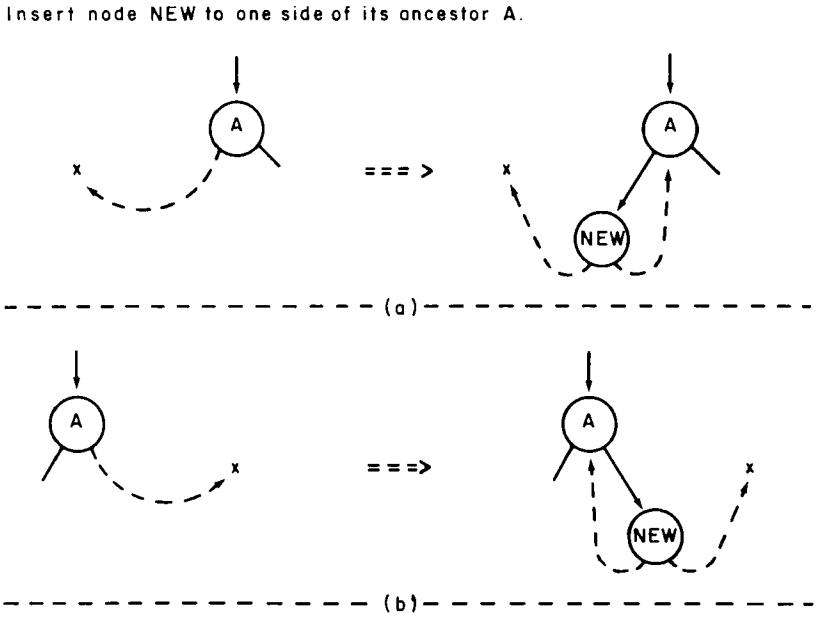


Figure 5. Insertion of a new node when (a) KEY(NEW) < KEY(A) ; (b) KEY(NEW) > KEY(A)

```
call DELETE(HEAD, OLDKEY)
     .
     .
     .
recursive procedure DELETE(T, OLDKEY)
    case
     : T = 0 :
         print('empty tree')
         return

     : OLDKEY < KEY(T) :
         if LSON(T) = 0
            then print('key not found')
            else call DELETE(LSON(T), OLDKEY)
         return

     : OLDKEY > KEY(T) :
         if RSON(T) = 0
```

```
            then print('key not found')
            else call DELETE(RSON(T), OLDKEY)
        return

    : OLDKEY = KEY(T) :
        call DETACH(T)
        return
    end
    end DELETE

recursive procedure DETACH(OLD)
    case
     : LSON(OLD) = RSON(OLD) :
        call DEALLOCATE(OLD)
        OLD <-- 0  /* empty tree */
        return

     : LBIT(OLD) = RBIT(OLD) :
        if LBIT(OLD) = 0
           then do  /* direct removal */
                call DEALLOCATE(OLD)
                /* LSON(0) pre-defined */
                if LSON(RSON(OLD)) = OLD
                   then do
                        LBIT(RSON(OLD)) <-- 0
                        OLD <-- LSON(OLD)
                        end
                   else do
                        RBIT(LSON(OLD)) <-- 0
                        OLD <-- RSON(OLD)
                        end
                end
           else do  /* replacement */
                S <-- SUCC(OLD)
                KEY(OLD) <-- KEY(S)
```

```
                call DELETE(RSON(OLD), KEY(S))
                end
            return

        : LBIT(OLD) ≠ RBIT(OLD) : /* circumvention */
            call DEALLOCATE(OLD)
            P <-- PRED(OLD)
            S <-- SUCC(OLD)
            if LBIT(OLD) = 0
               then do
                    OLD <-- RSON(OLD)
                    LSON(S) <-- P
                    end
               else do
                    OLD <-- LSON(OLD)
                    RSON(P) <-- S
                    end
            return
    end
    end DETACH

procedure PRED(T)
    if LBIT(T) = 0 then return(LSON(T))
    T' <-- LSON(T)
    while (RBIT(T') = 1) do
        T' <-- RSON(T')
        end
    return(T')
    end PRED



procedure SUCC(T)
    if RBIT(T) = 0 then return(RSON(T))
    T' <-- RSON(T)
```

```
while (LBIT(T') = 1) do
    T' <-- LSON(T')
    end
return(T')
end SUCC
```

Figure 6

deleted. Because the successor of a node with two direct descendants cannot be another one with two direct descendants, we expect to complete the deletion in the next step, and no degeneracy would ever occur. In the last case, to delete a node with one direct descendant, we circumvent the unwanted node by letting its ancestor A point to its descendant D and its successor S (predecessor P) takes its left (right) thread.

Two auxiliary functions are needed; one, SUCC, to find a successor and the other, PRED, to find a predecessor. The freed space of deleted nodes is re-collected by returning the pointers to a presumed procedure DEALLOCATE. The three cases of deleting a node are illustrated in Figure 7.

A balanced tree becomes unbalanced due to node insertions and deletions. To restore perfect balance we can again use the procedure BALTHR with slight modification on the TRAVBIND routine as follows:

```
procedure TRAVBIND:
    T <- - HEAD
    /* find the first node */
    while (LSON(T) ≠ 0) do
        T <- - LSON (T)
        end
    loop /* traversal and binding */
        N <- - N + 1
        LINK(N) <- - T
        T <- - SUCC (T)
        if T = 0 then return
    forever
    end TRAVBIND
```

From the algorithms given in this section we also see how a threaded tree structure could be manipulated for searching, traversal, and locating predecessors or successors.

CONCLUSION

Threaded binary search trees provide a flexible and efficient data organization that is particularly useful in real-time applications in which searching and sorting are both needed. This is because fast retrieval in logarithmic expected time as well as sequential access in either ascending or descending order are applicable. Although record insertions

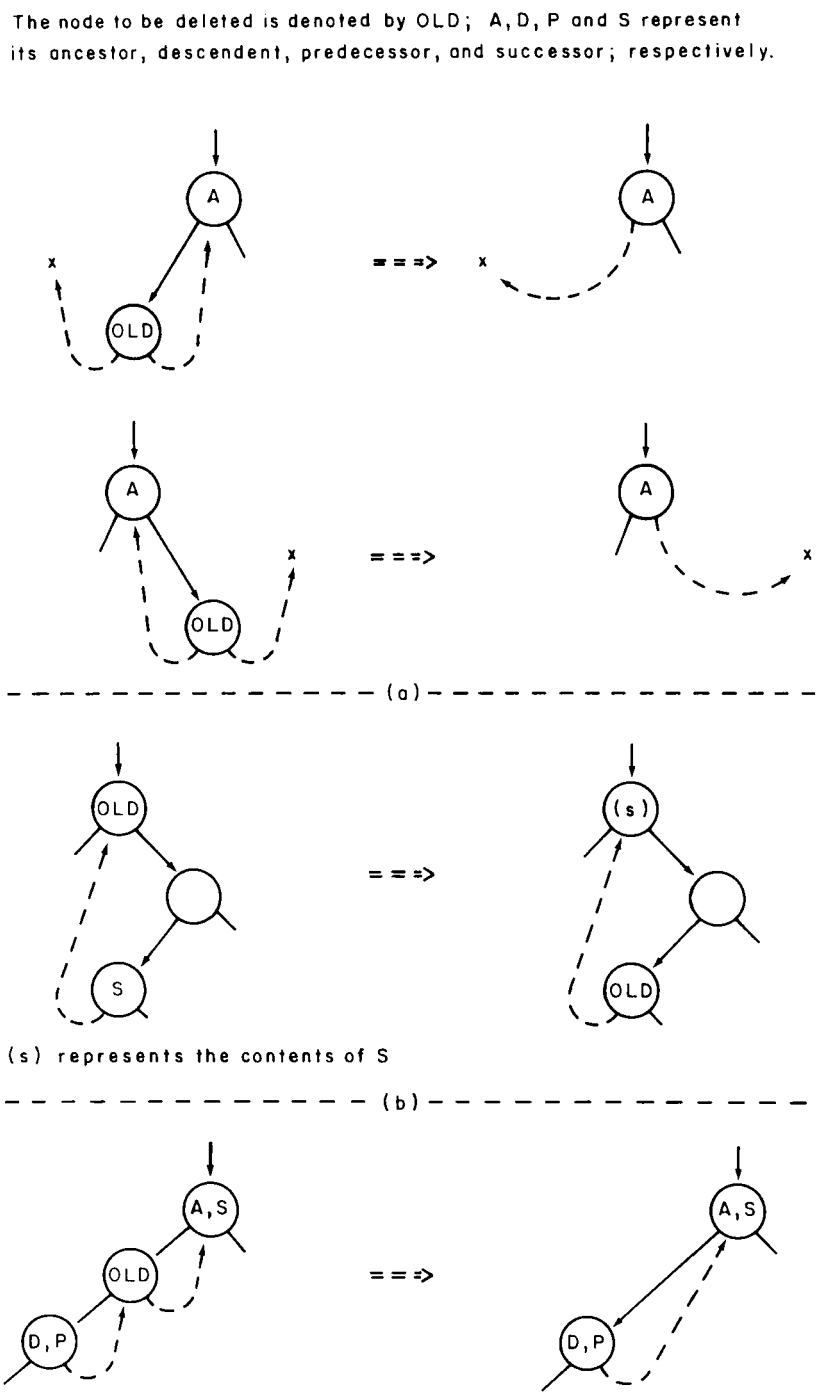


Figure 7. Deletion of a node by (a) direct removal; (b) replacement; (c) circumvention

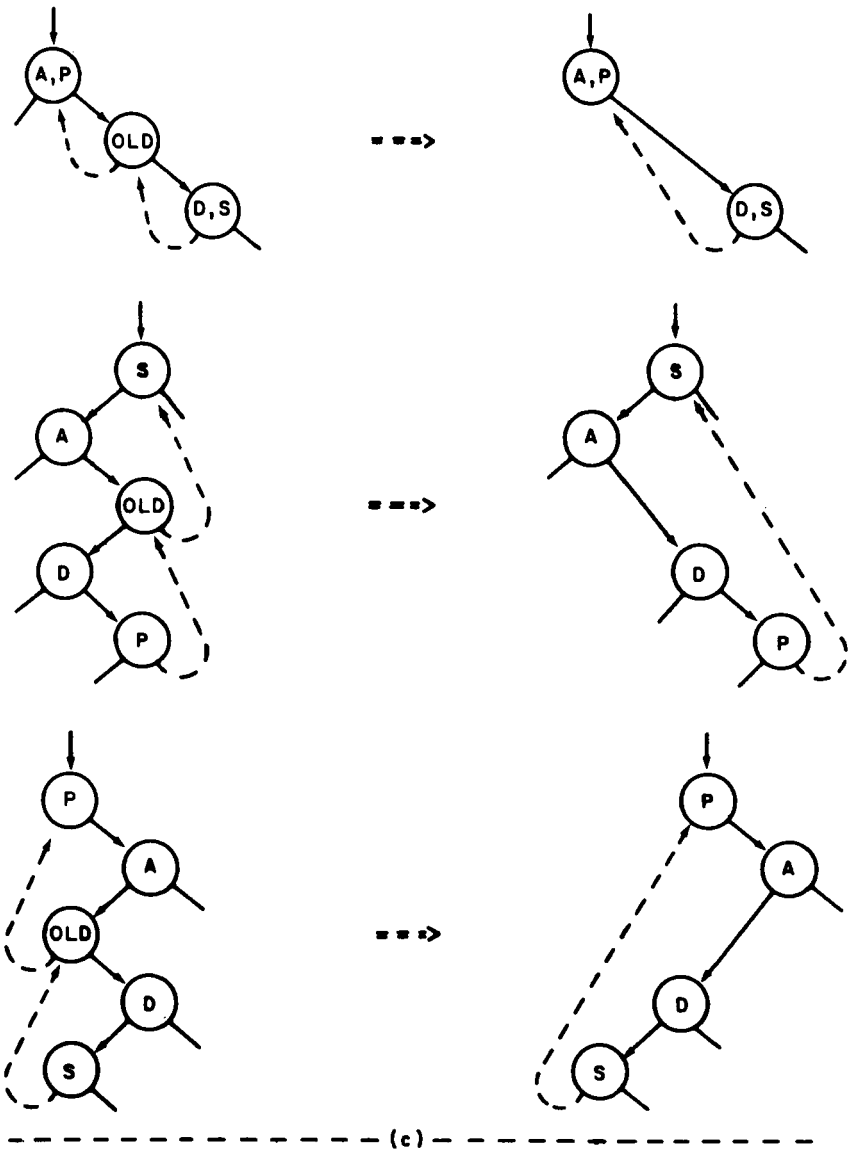


Figure 7.—cont'd

and deletions may disturb the perfect balance of the search tree, for random updating actions in a limited period of time we expect little impact on the performance of the entire system.

Many algorithms have been developed for maintaining the balance of binary trees during record insertion and deletion. But these algorithms usually tend to be complicated and involve too much balancing overhead. The rules should be somewhat relaxed, e.g. to separate record insertion and deletion procedures from the balancing procedure. The balancing algorithm presented in this paper is one suitable for optimizing a search

tree whenever the tree becomes too unbalanced. In addition, although not many algorithms deal with maintaining and optimizing threaded trees, the algorithms presented here seemingly well provide a system to do all these.

Finally, as a reflection on the modification of TRAVBIND in the preceding section, one may find that it is feasible to generate a balanced and threaded binary search tree on any set of data provided that the ordering information is available. Hence, the procedure GROW is not necessarily restricted to restructure search trees only. For recent results on parallel version of these algorithms — see Reference 6.

REFERENCES

1. H. Chang and S. S. Iyengar, 'Efficient algorithms to globally balance a binary search tree', *Communication of the ACM*, **27**, (7) 695–702 (1984).
2. N. Wirth, *Algorithm + Data Structure = Programs*. Prentice-Hall, Englewood Cliffs, N. J., 1976, pp. 189–242.
3. D. E. Knuth, *The Art of Computer Programming, Vol. 3, Sorting and Searching*, Addison-Wesley, Reading, Mass., 1968, p. 722.
4. E. Horowitz and S. Sahni, *Fundamentals of Data Structures*, Computer Sci. Press, Potomac, Md., 1976, pp. 442–456.
5. J. P. Tremblay and P. G. Sorenson, *An Introduction to Data Structure with Applications*, McGraw-Hill, 1976, pp. 326–329.
6. A. Moitra and S. S. Iyengar, 'A maximally parallel algorithm to balance binary search trees', to appear in *IEEE Trans. Computers* (1985).
7. A. C. Day, 'Balancing a binary tree', *The Computer Journal*, **19**, 360–361 (1976).
8. W. A. Martin and D. N. Ness, 'Optimal binary trees grown with a sorting algorithm', *Communications of the ACM*, **15**, 88–93 (1972).