OBSERVATIONS ON THE PERFORMANCE OF AVL TREES

R. E. Scroggs, P. L. Karlton,
S. H. Fuller, and E. B. Kaehler

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania

July, 1973

This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F44620-73-C-0074) and is monitored by the Air Force Office of Scientific Research.

ABSTRACT

This paper presents the results of a series of simulations that investigate the performance of AVL trees. It is shown that the only statistic of AVL trees that is a function of the size of the tree is the time to search for an item in the tree; the performance of all other procedures for maintaining AVL trees are independent of the size of the tree for trees greater than ~ 30 nodes. In particular it was discovered that an average of .465 restructures are required per insertion and .214 restructures per deletion. Moreover, an average of 2.78 nodes are revisited to restore the AVL property on insertion, and 1.91 nodes are revisited on deletion. Actual timings of the AVL procedures for insertion, searching, and deletion are presented to provide a practical guide to estimating the cost of using AVL trees.

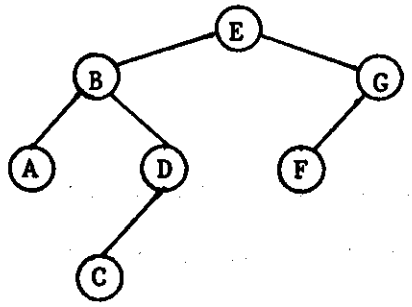
INTRODUCTION

This paper empirically examines the computational cost of insertion, deletion, and retrieval in AVL trees. An AVL tree is any rooted, binary tree with every node having the following property:

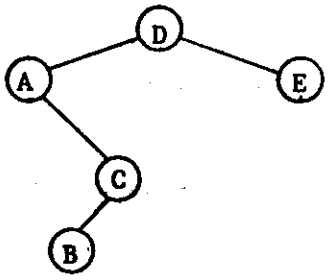
> AVL property. The height of the left subtree differs by at most one from the height of the right subtree. (The height of a tree is the length of the longest path from the root node to a leaf node.)

For example, the tree in Figure 1(a) is an AVL tree but the tree in Figure 1(b) is not because nodes A and D do not exhibit the AVL property. Given that a node possesses the AVL property, we will refer to it as balanced, left heavy, or right heavy depending on whether the height of the left subtree is equal to, greater than, or less than the height of the right subtree.

Immediately after inserting or deleting a node from an AVL tree one or more nodes may lose the AVL property. Figure 2 shows the two cases that can occur on insertion (and the two most common cases for deletion) and how to locally restructure the tree to restore the AVL property to all the nodes. A third case restructuring exists in deletion. It is similar to the single rotation case shown in Figure 2(a) except subtree β has a height of h+1, i.e., node C is balanced. A single rotation is sufficient to restore the AVL property to the critical node and we will subsequently refer to this case as the modified single rotation ease. For a more detailed description of AVL trees see [1, 2, 3, 4, 5, 6, 7].

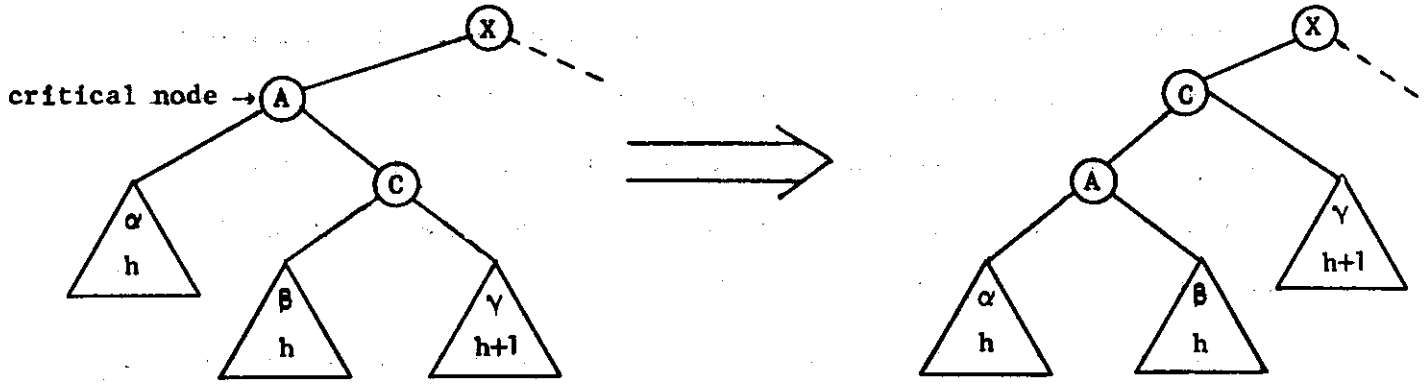


(a) An AVL tree

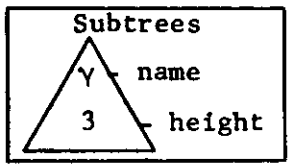


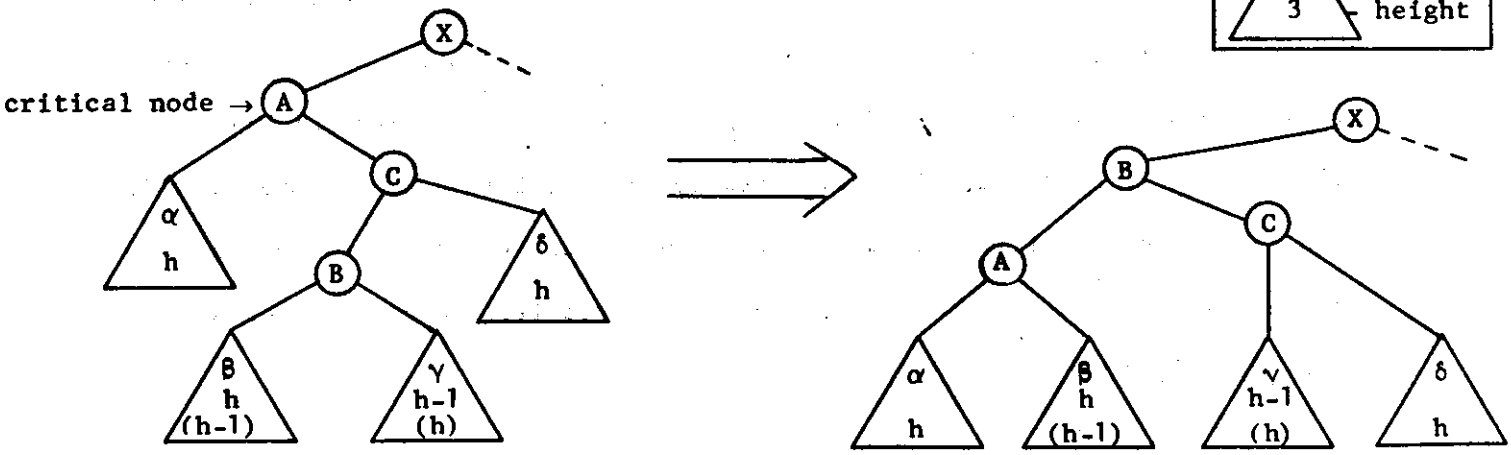
(b) A non-AVL tree

Figure 1. Search Trees



(a) The single rotation case.





(b) The double rotation case.

Figure 2. The Two Restructuring Cases for Insertion

For a tree structure in which insertion and deletion operations are frequently performed, it is important to know the costs of performing those operations as well as the cost of locating a node in the tree. AVL trees have the attractive property that all three operations (insertion, deletion, and retrieval) can be performed in O(log N) steps, i.e., on the order of log N steps, where N is the number of nodes in the tree. This is in contrast to random trees and completely balanced trees in which the worst case of at least one of the three operations can take as many as O(N) steps. Our goals here are to empirically find the average number of comparisons to find a node in the tree and to obtain more detailed estimates of the costs for restructuring the tree after an insertion or deletion.

EMPIRICAL OBSERVATIONS

In order to observe the performance of AVL trees, we presented random permutations of an ordered list to the procedure that inserts nodes into the AVL trees. Specifically, we used a uniform random number generator* to provide the values of the successive nodes to be inserted. To study the deletion process we selected any node in the tree for deletion with equal probability. To minimize correlation in the simulation we did N insertions and then N deletions, etc. Therefore, all observations of inserting (deleting) a node into an N node tree are independent events.

For the statistics that follow, 500 trees of size 5000 nodes were built up and then broken down, collecting statistics on trees of size 1 to 5000 in the process.

---

*The random number generator used was:

$$x_{i+1} = 3141592631\ x_i + 14522135347 \text{ modulo } 2^{35}.$$

On insertion the properties we tabulated were: (1) the average number of comparisons necessary to locate the position where a new node should be added (this is the average depth of the leaves and semi-leaves), (2) the percentage of insertions that caused a restructuring to be performed (statistics were kept for both types of restructuring), and (3) the average number of nodes visited during the traceback procedure (counted from the father of the node just added to the tree to the node at which tracebuack terminated).

On deletion the properties we tabulated were: (1) the average number of comparisons necessary to locate the node to be deleted (this is the average depth of all nodes in the tree), (2) the probable number of restructurings necessary on each deletion (statistics were kept for each of the three types of restructuring), and (3) the number of nodes visited during the traceback procedure (counted from the father of the node deleted from the tree to the node at which traceback terminated). On deletion, if the node to be deleted was not a leaf or a semi-leaf, we interchanged that node with its predecessor or successor* before deleting it.

Table 1 and Figure 3 present the results for insertion and deletion. The graphs for the average number of comparisons on insertion and deletion show that the retrieval time is logarithmic in the number of nodes in the tree. All other statistics, however, when plotted on graphs similar to Figure 3, were observed to be asymtotically independent of the size of the tree and, to within the precision of the simulation, the statistics had reached their asymtotic values for trees greater than ~ 30 nodes. Our results for the

*The node to be deleted was interchanged with its predecessor or successor depending on whether the node was heavy to the left or right, respectively. If the node was balanced, the node was interchanged with its predecessor.

Table 1. Insertion and Deletion Statistics

	Mean	Standard Deviation	95% Confidence Interval for Mean
Insertion:			
Single Rotation rebalance	.2327	.4226	±.0006
Double Rotation rebalance	.2324	.4223	±.0006
Number of nodes visited in traceback	2.778	1.625	±.003
Deletion:			
Modified Single Rotation rebalance	.0536	.2253	±.0003
Single Rotation rebalances	.0781	.2838	±.0004
Double Rotation rebalances	.0826	.2888	±.0004
Number of nodes visited in traceback	1.912	1.410	±.002

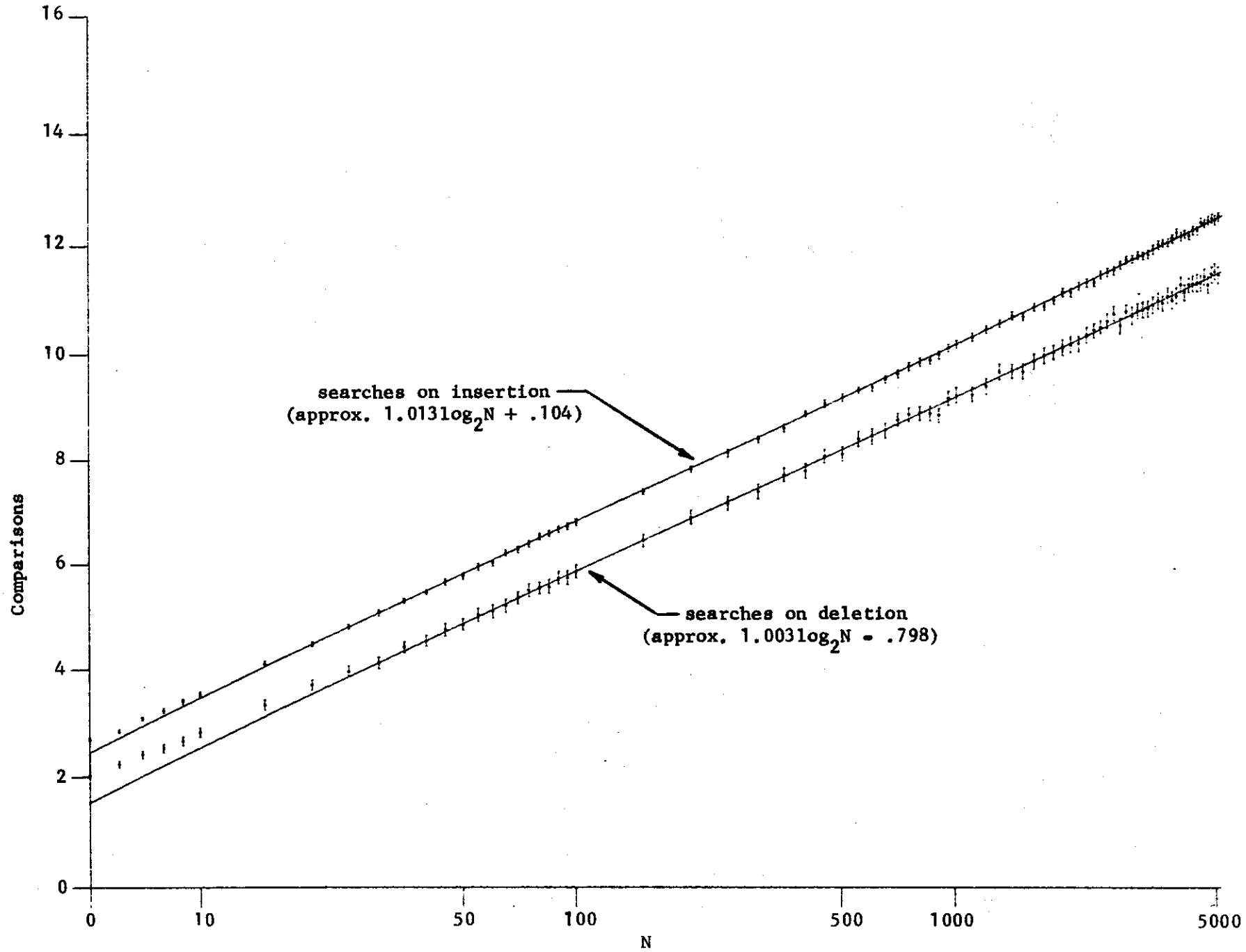


Figure 3. Expected Number of Comparisons to Locate an Item in an AVL tree of N Nodes. (Each point is surrounded by its 95% confidence interval.)

Table 1. Insertion and Deletion Statistics

	Mean	Standard Deviation	95% Confidence Interval for Mean
Insertion:			
Single Rotation rebalance	.2327	.4226	±.0006
Double Rotation rebalance	.2324	.4223	±.0006
Number of nodes visited in traceback	2.778	1.625	±.003
Deletion:			
Modified Single Rotation rebalance	.0536	.2253	±.0003
Single Rotation rebalances	.0781	.2838	±.0004
Double Rotation rebalances	.0826	.2888	±.0004
Number of nodes visited in traceback	1.912	1.410	±.002

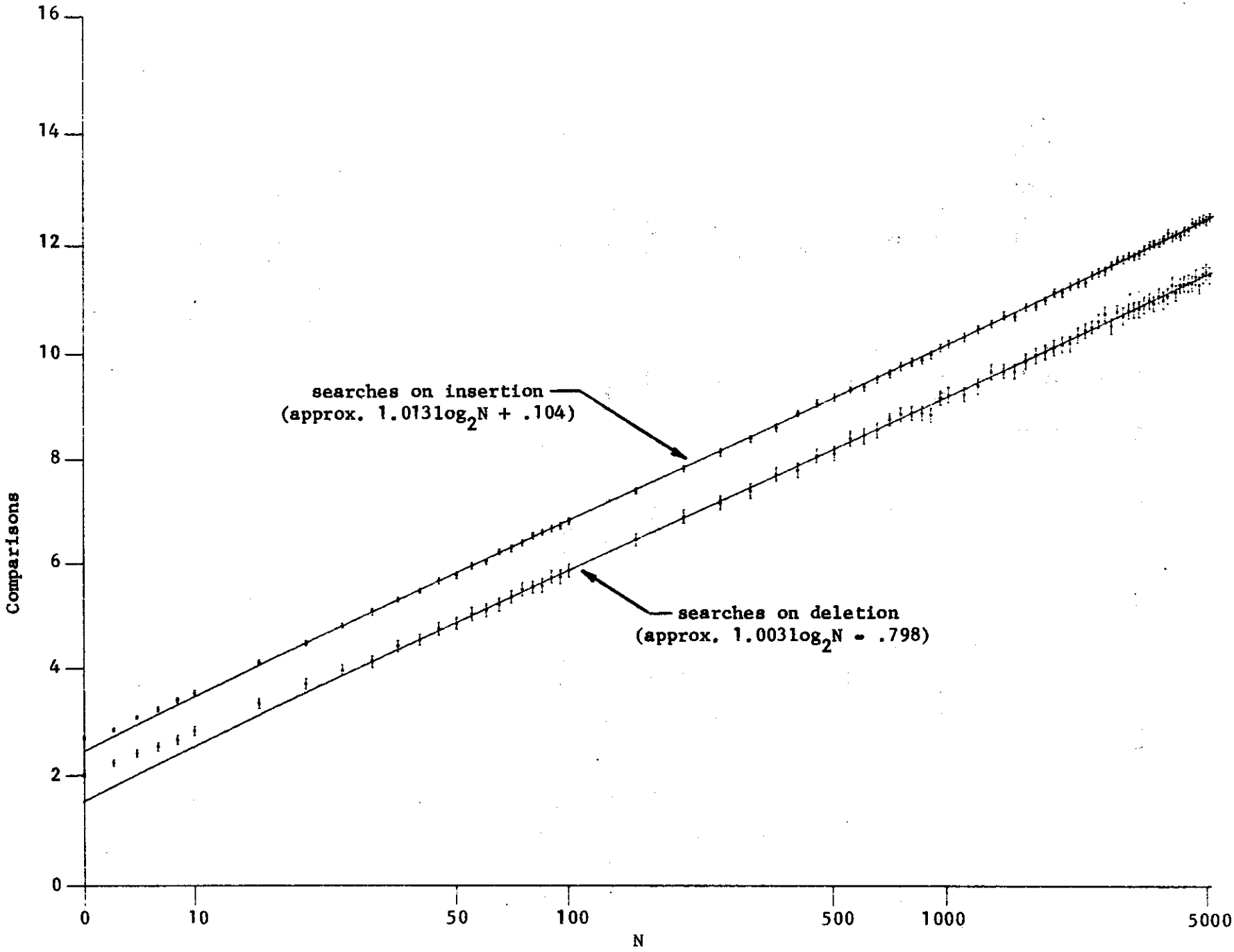


Figure 3. Expected Number of Comparisons to Locate an Item in an AVL tree of N Nodes. (Each point is surrounded by its 95% confidence interval.)

insertion costs concur with those of others [4]. The most surprising results are those for deletion; the worst case restructuring might involve $\log_2 N$ rebalances, but we observed that the probable number of rebalances per deletion was half that expected for insertion. A related observation is that the traceback on deletion visited approximately one less node per operation than on insertion.

It is interesting to note on insertion, to within the 95% confidence intervals of the simulation, that the single and double rotation cases appear to occur with equal frequency. In fact, a simple argument shows they must be equally probable. In Figure 2(a) consider the subtree rooted at C: it includes all items in the interval bounded by the values of nodes A and X. Since we require the nodes in the entire tree to arrive in random order, all the nodes that are within the interval bounded by A and X must also arrive in random order. Hence the subtree rooted at C is an AVL tree that is as probable to be left heavy as right heavy. However, the single feature that distinguishes the single rotation case from the double rotation case is whether or not node C is heavy in the same, or opposite, direction as the critical node.

TIMING STATISTICS

To compare the actual costs involved in performing the retrieval and restructuring operations, we gathered timing statistics for the operations. The following cost functions were derived from an implementation of the AVL procedures written in BLISS/10 [7] and run on a PDP-10 (all times are in microseconds):

Insertion	
search for the location to attach the new node	150 + 90 per compare*
attach node to the tree	80
restructure the tree	350
total	$580 + 90 \log_2 N$
Deletion	
search for the node to be deleted	150 + 90 per compare*
detach the node from the tree	325
restructure the tree	250
total	$725 + 90 \log_2 (N)$

These results show that deletion is slightly more expensive than insertion, but that for large trees, the search time is the dominant factor in both operations.

*Figure 3 gives the expected number of comparisons as a function of the size of the tree.

References

1. Adel'son-Vel'skii, G. M. and E. M. Landis, "An Algorithm for the Organization of Information," Doklady Akad. Nauk USSR Moscow 16, No. 2 (1962), pp. 263-266 (Russian), English translation in Soviet Math. Doklady 3 (1962), pp. 1259-1263.

2. Foster, C. C., "Information Storage and Retrieval Using AVL Trees, Proc. ACM 20th Nat. Conf., 1965, pp. 192-205.

3. Knott, G. D., "A Balanced Tree Storage and Retrieval Algorithm," Proc. Symp. on Inform. Storage and Retrieval, April 1971, pp. 175-196.

4. Knuth, D. E., The Art of Computer Programming, Vol. 3, Sorting and Searching, Addison-Wesley, Reading, Mass., 1973, Sec. 6.2.3.

5. Kosajaru, S. Rao, On Information Storage and Retrieval by AVL Trees, Department of Electrical Engineering, The Johns Hopkins University, Baltimore, Maryland.

6. Nievergelt, J. and E. M. Reingold, Binary Search Trees of Bounded Balance, Department of Computer Science, University of Illinois, Urbana, Illinois.

7. Stone, H. S., Introduction to Computer Organization and Data Structures, McGraw-Hill, New York, N. Y., 1972, Sec. 11.2.3.

8. Wulf, W. A., D. B. Russell, and A. N. Habermann, "BLISS: A Language for Systems Programming," Comm. ACM 14, 12 (December 1971), pp. 780-790.

APPENDIX: A Set of Procedures for Inserting into, Deleting from, and Searching AVL Trees

```
integer array tree[0:n,1:6],drct,path[0:2.25*ln(n+2)];
integer treebase,level;
```

comment These identifiers represent the data structures to be used in the procedures.

1) *tree* represents the AVL tree, it is accessed with two values, a pointer to a node and a field specification within that node. *tree* is declared as an array, but any data structure accessed in the above manner is acceptable. The actual tree constructed is built as the left subtree of a special header node pointed to by *treebase*.

2) *path* and *drct* are used as a stack to record movements through the tree. *path* holds pointers to nodes and *drct* holds the direction moved when stepping down from the node identified by the corresponding pointer in *path*. *level* is used as a stack pointer into *path* and *drct*. ;

```
integer nullink,maxkey;
integer llink,rlink,rank,key,infoptr,balance;
```

comment The first two identifiers above represent constants and the second six represent the field specifiers for each node in the tree. In this implementation, the latter six take on the values 1,...,6 respectively, corresponding to the second index of the array *tree*.

1) The *llink* and *rlink* of each node contain pointers to its left and right sons. The value *nullink*(which must be chosen to be distinct from all valid link values) in a field indicates that no son is present on that side of the node. *llink* in the header node holds a pointer to the root node of the tree, *rlink* in the header always has the value *nullink*.

2) The *rank* of each node specifies the relative position of that node in the subtree of which it is the root, i.e., one plus the number of nodes in its left subtree. The *rank* of the header node, therefore, specifies one plus the number of nodes in the tree.

3) If the AVL tree is being used to store nodes by value, then the *key* of each node contains the value that uniquely identifies the node and the nodes are stored in lexicographic order based on the value of *key*. If the AVL tree is being used to store a linear list, then the key values are not necessarily distinct. In this case, each node is accessed by specifying its position in the postorder sequence of the tree. The *key* of the header node contains the value *maxkey* which must be chosen larger than any key to be added to the tree.

4) the ***infoptr*** of each node contains a pointer to the information associated with that node. If only a small, constant amount of information is associated with each node, the pointer can be eliminated and be replaced by the information itself. The ***infoptr*** of the header node always has the value ***nullink***.
5) the ***balance*** of each node indicates the balance of the subtree for which that node is the root. The possible values are: ***left*** heavy, ***right*** heavy, or ***balanced***. The ***balance*** of the header node is undefined. ;

```
integer balanced,left,right,found,notfound,bounderror,
        delete,retrieve,insert;

comment These identifiers represent constants in the
procedures. They may be assigned any value such that
within each of the following three sets, each member has a
distinct value.
                {balanced,left,right}
                {found,notfound,bounderror}
                {delete,retrieve,insert} ;

integer procedure OPPSIDE(side);
integer side; value side;
  OPPSIDE:=if side = left then right else left;

integer procedure OPPSLINK(side);
integer side; value side;
  OPPSLINK:=if side = left then rlink else llink;

integer procedure SIDELINK(side);
integer side; value side;
  SIDELINK:=if side = left then llink else rlink;

procedure INITHEADER;
begin
comment
Function: to initialize the special header node for the
tree. This procedure must be called before any tree
operations are performed.

Side effects: None. ;

  tree[treebase,rank]:=1;
  tree[treebase,key]:=maxkey;
  tree[treebase,infoptr]:=nullink;
  tree[treebase,llink]:=tree[treebase,rlink]:=nullink;
  tree[treebase,balance]:=balanced
end INITHEADER;
```

integer procedure *SEARCH*(*positionsearch,searchkey,typesearch*);
boolean *positionsearch*; integer *searchkey,typesearch*;
value *positionsearch,searchkey,typesearch*;
begin integer *tpoint,field,loopdummy,chgval*;
comment
Permissible values for the parameters:

1) ***positionsearch*** - true, false
2) ***searchkey*** - a list position or a key value
3) ***typesearch*** - ***insert***, ***retrieve***, ***delete***

Function: to search for a node in the tree. If ***positionsearch*** = true, then ***searchkey*** is interpreted as a position in the postorder sequence of the tree, otherwise ***searchkey*** is interpreted as a key value.

Side effects:

1) ***path*** and ***drct*** are filled with the links and directions taken on the path from the header node to the desired node. ***path***[***level***] points to the desired node when it is found in the tree, otherwise it points to the leaf or semi-leaf from which the desired node could be a son. On insertion ***drct***[***level***] specifies which side of the node pointed to by ***path***[***level***] the new node will be added.
2) ***typesearch*** indicates the purpose of the search, i.e., a search for a position to insert a new node, or a search to find a node so that it may be deleted, or a search for retrieval of information. The ***rank*** field of the nodes on the path are adjusted for the insert and delete cases with the assumption that the operation will subsequently be performed.

Value of *SEARCH*:

1) ***found*** - node located in tree, pointed to by ***path***[***level***].
2) ***notfound*** - node not located in tree, ***path***[***level***] points to node which could be the father of desired node.
3) ***bounderror*** - position specified does not occur in list(on search by position only). ;

```
procedure RESETRANK(typesearch);
integer typesearch; value typesearch;
begin integer chgval,tlevel;
comment
Permissible values for the parameter typesearch: insert,
delete.

Function: to restore the value of the rank field in
each node.  This only has to be done if an attempt
was made to insert(by value) a node which already existed
in the tree or an attempt was made to delete a node which
did not exist in the tree. typesearch specifies for which
case the correction is being made(insert, delete).

Side effects: None.

  chgval:=if typesearch = insert then -1 else 1;
  for tlevel:=(if chgsw = insert
               then level-1 else level)
    step -1 until 0 do
    if drct[tlevel] = left
      then tree[path[tlevel],rank]:=
           tree[path[tlevel],rank]+chgval
end RESETRANK;

comment The body of SEARCH begins here. ;
 if (positionsearch ∧
   ((searchkey ≤ 0) ∨
    (searchkey > tree[treebase,rank]) ∨
    (typesearch ≠ insert ∧ searchkey = tree[treebase,rank])) ∨
   (¬ positionsearch ∧ searchkey ≥ maxkey)
  then
    begin
      SEARCH:=bounderror;
      goto endsrch
    end;
 tpoint:=treebase;
 level:=0;
 chgval:=if typesearch = insert
         then 1 else (if typesearch = delete
                      then -1 else 0);
 field:=if positionsearch then rank else key;
```

```
for loopdummy:=1 while true do
begin
  path[level]:=tpoint;
  if searchkey ≠ tree[tpoint,field]
    then
      begin integer side;
        drct[level]:=side:=if searchkey < tree[tpoint,field]
                           then left else right;
        if positionsearch ∧ (side = right)
          then searchkey:=searchkey-tree[tpoint,rank];
        tree[tpoint,rank]:=tree[tpoint,rank]+
          (if side = left then chgval else 0);
        tpoint:=tree[tpoint,SIDELINK(side)];
        if (¬ positionsearch) ∧ (tpoint = nullink)
          then
            begin
              if typesearch = delete then RESETRANK(delete);
              SEARCH:=notfound;
              goto endsrch
            end;
        level:=level+1
      end
    else
      begin
        if typesearch = insert
          then
            begin
              if ¬ positionsearch
                then RESETRANK(insert)
                else
                  begin
                    drct[level]:=left;
                    tree[path[level],rank]:=
                      tree[path[level],rank]+1;
                    if tree[path[level],llink] ≠ nullink
                      then
                        begin
                          level:=level+1;
                          path[level]:=
                            tree[path[level-1],llink];
                          drct[level]:=right;
                          for path[level+1]:=
                              tree[path[level],rlink]
                            while (path[level+1] ≠ nullink) do
                          begin
                            level:=level+1;
                            drct[level]:=right
                          end
                        end
                  end
            end;
        SEARCH:=found;
        goto endsrch
      end
  end;
endsrch:
end SEARCH;
```

```
procedure SINGLEROTATE;
begin integer pfather,pcritical,pcritson1,side;
comment
Function: to perform a 'single' rotation at a critical node
to re-establish the AVL property. The llink, rlink,
balance, and rank values are adjusted for the critical
node, its father and the the heavy side son of the
critical node.

Side effects: none. ;

  pfather:=path[level-1];
  pcritical:=path[level];
  pcritson1:=path[level+1];
  side:=tree[pcritical,balance];
  tree[pfather,SIDELINK(drct[level-1])]:=pcritson1;
  tree[pcritical,SIDELINK(side)]:=
    tree[pcritson1,OPPSLINK(side)];
  tree[pcritson1,OPPSLINK(side)]:=pcritical;
  tree[pcritical,rank]:=
    tree[pcritical,rank]-
      (if side = left then tree[pcritson1,rank] else 0);
  tree[pcritson1,rank]:=
    tree[pcritson1,rank]+
      (if side = right then tree[pcritical,rank] else 0);
  tree[pcritical,balance]:=tree[pcritson1,balance]:=balanced
end SINGLEROTATE;
```

```
procedure DOUBLEROTATE;
begin integer pfather,peritical,pcritson1,pcritson2,side;
comment
Function: to perform a 'double' rotation at a critical node
to re-establish the AVL property. The llink, rlink,
balance, and rank values are adjusted for the critical
node, its father, the the heavy side son of the
critical node, and the heavy side son of that son.

Side effects: none. ;

  pfather:=path[level-1];
  peritical:=path[level];
  pcritson1:=path[level+1];
  pcritson2:=path[level+2];
  side:=tree[peritical,balance];
  tree[pfather,SIDELINK(dret[level-1])]:=pcritson2;
  tree[pcritson1,OPPSLINK(side)]:=
    tree[pcritson2,SIDELINK(side)];
  tree[peritical,SIDELINK(side)]:=
    tree[pcritson2,OPPSLINK(side)];
  tree[pcritson2,SIDELINK(side)]:=pcritson1;
  tree[pcritson2,OPPSLINK(side)]:=peritical;
  tree[peritical,rank]:=
    tree[peritical,rank]-
      (if side = left
         then tree[pcritson1,rank]+tree[pcritson2,rank] else 0);
  tree[pcritson1,rank]:=
    tree[pcritson1,rank]-
      (if side = right then tree[pcritson2,rank] else 0);
  tree[pcritson2,rank]:=
    tree[pcritson2,rank]+
      (if side = left then tree[pcritson1,rank]
                      else tree[peritical,rank]);
  tree[peritical,balance]:=
    if tree[pcritson2,balance] = side
      then OPPSIDE(side) else balanced;
  tree[pcritson1,balance]:=
    if tree[pcritson2,balance] = OPPSIDE(side)
      then side else balanced;
  tree[pcritson2,balance]:=balanced
end DOUBLEROTATE;
```

```
procedure ATTACHNODE(pfree,newkey,newinfoptr);
integer pfree,newkey,newinfoptr;
value   pfree,newkey,newinfoptr;
begin
comment
Permissible values for the parameters:
    1) pfree - a pointer to any empty node
    2) newkey - value of key for the new node
    3) newinfoptr - value of information pointer for the
    new node

Function: to insert a new node into the tree.
The new node is attached to the node pointed to by path[level]
on the side specified by drct[level].

Side effects: pfree is placed in path[level+1] for use by
the rotation procedures. ;

  tree[path[level],SIDELINK(drct[level])]:=pfree;
  tree[pfree,rank]:=1;
  tree[pfree,llink]:=tree[pfree,rlink]:=nullink;
  tree[pfree,balance]:=balanced;
  tree[pfree,key]:=newkey;
  tree[pfree,infoptr]:=newinfoptr;
  path[level+1]:=pfree
end ATTACHNODE;
```

```
procedure REBUILDINSERT;
begin
comment
Function: to trace back along the path from the father of
the node just attached to the tree, checking
that the AVL property has been maintained. A rotation is
performed if the property no longer holds at a node. At
most one rotation is performed but traceback may terminate
without performing a rotation or reaching the top of the
tree.

Side effects: none. ;

  tree[path[level],balance]:=
    if tree[path[level],balance] = balanced
      then drct[level] else balanced;
  if tree[path[level],balance] ≠ balanced
    then
      begin integer loopdummy; boolean critsw;
        critsw:=false;
        for loopdummy:=1 while (level > 1) do
        begin
          level:=level-1;
          if tree[path[level],balance] ≠ balanced
            then
              begin
                tree[path[level],balance]:=
                  if tree[path[level],balance] = drct[level]
                    then drct[level] else balanced;
                critsw:=
                  if tree[path[level],balance] = drct[level]
                    then true else false;
                goto chkcrit
              end
            else tree[path[level],balance]:=drct[level]
        end;
chkcrit:
        if critsw
         then
          begin
            if tree[path[level],balance] =
                 tree[path[level+1],balance]
              then SINGLEROTATE
              else DOUBLEROTATE
          end
      end
end REBUILDINSERT;
```

```
procedure DETACHNODE;
begin integer pdel;
comment
Function: to delete the node specified by path[level] from
the tree.

Side effects: If the node to be deleted is not a leaf or a
semi-leaf, then the node is interchanged with its postorder
predecessor(successor) before being deleted. path and drct
are filled out with the path down to the
predecessor(successor). The predecessor is chosen if the node is left
heavy or balanced, otherwise the successor is chosen. ;

  pdel:=path[level];
  if (tree[pdel,llink] ≠ nullink) ∧
     (tree[pdel,rlink] ≠ nullink)
    then
      begin integer tpoint,tlevel,temp,side,chgval,slink;
      comment Node is not a leaf or semi-leaf, find its
              predecessor(successor). ;
        tlevel:=level-1;
        drct[level]:=side:=if tree[pdel,balance] = right
              then right else left;
        chgval:=if side = left then 0 else 1;
        tree[pdel,rank]:=
          tree[pdel,rank]-(1-chgval);
        level:=level+1;
        path[level]:=tree[pdel,SIDELINK(side)];
        side:=OPPSIDE(side);
        slink:=SIDELINK(side);
        for path[level+1]:=tree[path[level],slink]
          while (path[level+1] ≠ nullink) do
        begin
          drct[level]:=side;
          tree[path[level],rank]:=
            tree[path[level],rank]-chgval;
          level:=level+1
        end;
        comment Perform interchange. ;
        tpoint:=path[level];
        tree[path[tlevel],SIDELINK(drct[tlevel])]:=tpoint;
        temp:=tree[pdel,llink];
        tree[pdel,llink]:=tree[tpoint,llink];
        tree[tpoint,llink]:=temp;
        temp:=tree[pdel,rlink];
        tree[pdel,rlink]:=tree[tpoint,rlink];
        tree[tpoint,rlink]:=temp;
        tree[tpoint,rank]:=tree[pdel,rank];
        tree[tpoint,balance]:=tree[pdel,balance];
        path[tlevel+1]:=tpoint
      end;
  comment Delete leaf or semi-leaf. ;
  tree[path[level-1],SIDELINK(drct[level-1])]:=
    tree[pdel,if tree[pdel,llink] = nullink
              then rlink else llink]
end DETACHNODE;
```

```
procedure REBUILDDELETE;
begin integer loopdummy;
comment
Function: to trace back along the path from the father of
the node deleted from the tree, checking that
the AVL property has been maintained. A rotation is
performed if the property no longer holds at a node.
Several rotations may be necessary but traceback may
terminate without performing a rotation or reaching the
top of the tree.

Side effects: none. ;

  for loopdummy:=1 while (level > 1) do
  begin
    level:=level-1;
    if tree[path[level],balance] = balanced
      then
        begin
          tree[path[level],balance]:=OPPSIDE(drct[level]);
          goto endrbld
        end
      else if tree[path[level],balance] = drct[level]
             then tree[path[level],balance]:=balanced
             else
               begin
                 path[level+1]:=
                   tree[path[level],OPPSLINK(drct[level])];
                 if tree[path[level+1],balance] = balanced
                   then
                     begin
                       SINGLEROTATE;
                       tree[path[level],balance]:=
                         OPPSIDE(drct[level]);
                       tree[path[level+1],balance]:=drct[level];
                       goto endrbld
                     end
                   else if tree[path[level],balance] =
                             tree[path[level+1],balance]
                          then SINGLEROTATE
                          else
                            begin
                              path[level+2]:=
                                tree[path[level+1],
                                  SIDELINK(tree[path[level+1],
                                           balance])];
                              DOUBLEROTATE
                            end
               end
  end;
endrbld:
end REBUILDDELETE;
```