AN OPTIMAL POINTER MACHINE ALGORITHM FOR FINDING NEAREST COMMON ANCESTORS

J. van Leeuwen & A.K. Tsakalides

RUU-CS-88-17
April 1988

Rijksuniversiteit Utrecht

Vakgroep informatica

Budapestlaan 6 3584 CD Utrecht
Corr. adres: Postbus 80.012 3508 TA Utrecht
Telefoon 030-53 1454
The Netherlands

AN OPTIMAL POINTER MACHINE ALGORITHM FOR FINDING NEAREST COMMON ANCESTORS

J. van Leeuwen & A.K. Tsakalides

Technical Report RUU-CS-88-17
April 1988

Department of Computer Science
University of Utrecht
P.O.Box 80.012, 3508 TA Utrecht
The Netherlands

An Optimal Pointer Machine Algorithm for finding Nearest Common Ancestors

Jan van Leeuwen* and Athanasios K. Tsakalidis**

Technical Report, April 1988

Computer Technology Institute, Patras, Greece

* *Dept. of Computer Science, University of Utrecht, P.O. Box 80.012, 3508 TA Utrecht, The Netherlands.*

** *Dept. of Computer Science, Universtity of Patras and Computer Technology Institute, P.O. Box 1122, 26110 Patras, Greece.*

ABSTRACT: We present an optimal pointer machine algorithm for computing the nearest common ancestor of any two given nodes in a static arbitrary tree with n nodes in $O(\log\log n)$ time. The algorithm requires $O(n)$ preprocessing time and only $O(n)$ space.

1. Introduction

Suppose a rooted tree T with n nodes is available for preprocessing. We consider the following problem: answer *on-line* queries of the form "Which node in T is the nearest common ancestor of two given nodes x and y?". We shall denote the answer to such a query $nca(x,y)$ and refer to the problem of computing it as the static nca-problem.

Several versions of the nca-problem have been considered in the literature, differing in whether the *nca* queries are handled *off-line* or *on-line* , whether the tree T is static or not and, in the dynamic case, in the kind of dynamic

operations (linkings and cuttings) allowed on trees. These versions of the nca-problem have been investigated by [Aho, Hopcroft, Ullman, 76], [Maier, 79], [van Leeuwen, 76], [Harel and Tarjan, 84], [Sleator and Tarjan, 83], [Schieber and Vishkin, 87] and [Tsakalidis, 88].

In this paper we are interested in a solution of the static nca-problem that runs efficiently on pointer machines. We combine the results of [van Leeuwen, 76] and [Tsakalidis, 88] and provide an *optimal* pointer machine algorithm for computing each *nca* query in $O(\log\log n)$ time, with *linear* preprocessing time and space. It follows from a result in [Harel and Tarjan, 84] that this query time is optimal. The computation model will be discussed in more detail later in this section.

Given an arbitrary static tree T with n nodes, we obtain the solution to the static nca-problem by the following steps: First, T will be embedded into a balanced tree BT of depth $O(\log n)$ and arbitrary degree and $nca(x,y)$ is computed in BT. Given the $nca(x,y)$ in BT, it is then possible to compute the $nca(x,y)$ in T in $O(1)$ time , by using the information kept during the embedding. Secondly, we preprocess BT in such a way that it is possible to compute $nca(x,y)$ in BT in $O(\log\log n)$ time. More precisely, the query time is $O(\log min\{dbt(x), dbt(y)\})$, where $dbt(x)$ is the depth of a node x in BT.

The paper is organized as follows. In the remainder of this section we define the model of computation we are working with and present the optimality proof of Harel and Tarjan for the sake of completeness. In section 2 we explain the embedding of T into BT and in section 3 we give the necessary preprocessing of BT in order to support the query time mentioned.

In our discussion we shall use n to denote the current number of nodes in the tree T. Our model of computation is a *pointer machine* as defined by [Tarjan, 79], which is different from a *random access machine* or RAM [Aho, Hopcroft, Ullman, 74]. In a pointer machine memory consists of a collection of *records*. Each record consists of a fixed number of *cells*. The cells have associated *types*, such as *pointer, integer, real*, and access to memory is only possible by *"pointers"* and not by address arithmetic. Note that in a RAM the memory consists of an array of cells, i.e. we can access a cell by its address, using address arithmetic when neccessary.

For measuring time we use the *uniform cost measure*, in which each arithmetic and pointer operation requires $O(1)$ time. We assume that each number used can be completely stored in a cell. We also assume that the tree is represented by a list structure (which may change in the course of computation), with each tree node being represented by a single record. The structure may contain additional records which do not represent specific tree node. Each record contains a fixed number of pointers, independent of n. As an input for a *nca* query, the algorithm is given pointers which point to the records corresponding to the tree nodes x and y of which the nearest common ancestors is required. In order to answer the query, the algorithm must return a pointer which points to the record corresponding to node $nca(x,y)$. We consider the

on-line version of the problem in which queries arrive *on-line* and we assume that the algorithm remembers nothing between consecutive queries.

[Harel and Tarjan, 84] demonstrate the weakness of a pointer machine in comparison to a RAM in the following way. They consider the nearest common ancestor problem on static trees and show that any pointer machine algorithm requires $\Omega(\log \log n)$ time per *nca* query. Additionally they give an algorithm running on a RAM which solves the nca-problem in $O(1)$ time per query using $O(n)$ preprocessing time and $O(n)$ space.

The following Theorem with its proof is due to [Harel and Tarjan, 84] and garantees the optimality of the $O(\log \log n)$ query time for the case of pointer machines.

Theorem 1: Let T be a complete binary tree with n leaves. Any pointer machine requires $\Omega(\log \log n)$ time to answer any *nca-query* in the worst case, independent of the representation of the tree.

Proof: Let $n = 2^h$, with h the height of T. Let us fix our attention on the time just before a query. The key point is that, from any node in the data structure, at most $2^{j+1} - 1$ nodes are accessible in j steps or less. Let k be such that any possible *nca* query can be answered in k steps or less. For each leaf x of T let A_x denote the set of nodes representing tree nodes that are accessible from x in k steps or less. Let w be a nonleaf node of T and let u and v be its two children. We claim that either w belongs to A_x for every leaf x that is a descendant of u, or w belongs to A_y for every leaf y that is a descendant of v. Otherwise w would be accessible from neither x nor y in k steps for well-chosen leaves x and y in the subtrees of u and v, and an *nca* query on x and y would be unanswerable in k steps.

We conclude that w belongs to A_x for at least half the leaves x that are descendants of w. If w has height $i \geq 1$, then w has 2^i leaf descendants, and thus w occurs in at least 2^{i-1} sets A_x. Since there are 2^{h-i} nodes of height i for any i with $1 \leq i \leq h = \log n$, we see that nodes of height i contribute $2^{h-i}2^{i-1} = 2^{h-1} = n/2$ occurrences to the collection of sets A_x. Summing over all heights i from 1 to h, we find that

$$\sum_{x \in L} |A_x| \geq \frac{n}{2} \log n,$$

where L is the set of leaves of T. Since for any leaf x we have $|A_x| < 2^{k+1}$, we get

$$n2^{k+1} > \frac{n}{2} \log n,$$

which implies that $k > \log \log n - 2$

■

We will show that there exists a pointer machine algorithm that achieves the optimal bound of theorem, with only linear preprocessing time and linear space.

2. Embedding an arbitrary tree into a balanced tree

In this section we explain a technique of embedding an arbitrary tree T with n nodes into a balanced tree BT with height $O(\log n)$, while keeping information about the ancestral relations of nodes [cf. van Leeuwen, 76].

The technique will be explained by the following three steps and is presented in subsequent subsections:

Step 1: (Section 2.1) A data structure for a forest of trees is presented which can merge two trees at their roots, by making the root of one tree a son of the root of the other tree, in such a way that the new tree remains balanced. Hence, each tree which is the result of merging two other trees appears in two versions; first as a non-balanced merged tree (the "real" version) and secondly as a balanced merged tree (the "imaginary" version).

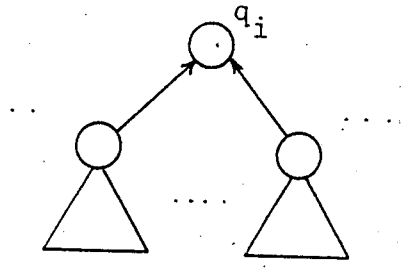
Step 2: (Section 2.2) Information is stored in such a way that if a nearest common ancestor is found in the balanced merged tree, then we can compute the nearest common ancestor in the non-balanced merged tree in $O(1)$ time.

Step 3: (Section 2.3) Given an arbitrary static tree T, we present a preorder traversal of T and we merge the subtrees visited according to steps 1 and 2. The resulting tree BT has height $O(\log n)$ and the property that if we find $nca(x, y)$ in BT, then $nca(x, y)$ in T can be computed in $O(1)$ time.

2.1 A data structure for a forest of trees

Like in [Aho, Hopcroft, Ullman, 76], efficiency in representing a forest is gained by keeping a *shadow* structure that codes frequently needed information of the forest in a sufficiently balanced or compressed manner. However, the data structure which we shall use here is fundamentally different from the organisation used in [Aho, Hopcroft, Ullman, 76].

At each moment the record kept for each node in the forest will contain much more information than "just" a pointer to its current father as it shown below.



In fact, q_i may not be the "real" root of this tree at all but only a "representative" which is used to make the tree balanced. Thus, the physical tree observed from the father-son relations in the data structure may not be the "real" tree at all. Additional information is stored in each (internal) node to

enable us to work with this tree and still get every information needed of it as if it were the real tree. For each node a structure is kept that consists of a record with global information and a list (or "filter") to which a selection of its descendants are "attached". An important fact is that the direct sons of q_i (except if they are leaves) are not *directly* connected to that node, but pass the "filter" which is kept in the structure for q_i. More precisely, with each q-node we have the following global structure, which is easily formulated as a "type" in PASCAL. (We use q_i to refer to the node, Q_i to refer to its record.)

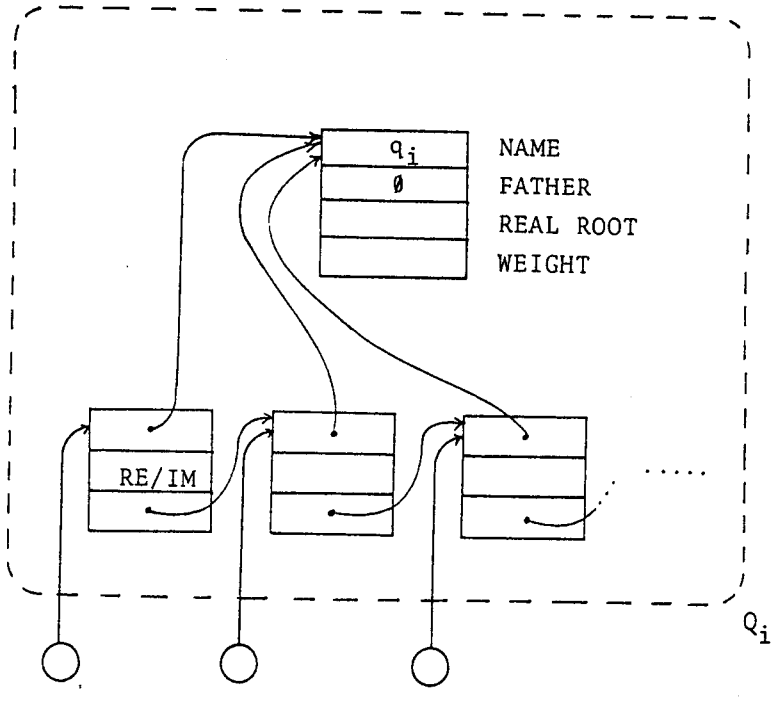


Figure 1

The records in the linked list contain a few more fields than shown in Figure 1, but this will be discussed later.

Some parts of this structure are self-explanatory. Notice the point that sons are entered in a *linked list*, part of the node, before being connected to the main record.

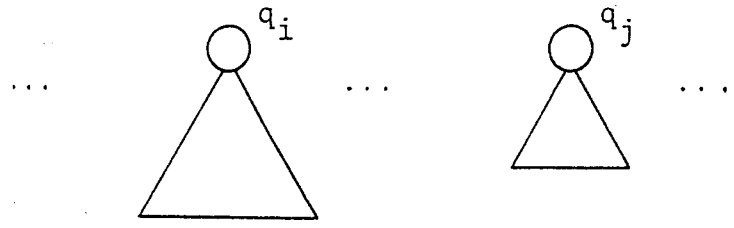
Sons are entered in the linked list *in the order* in which they must be attached to the "node" according to the appropriate interpretation of MERGE - instructions, which are explained below.

Definition 1:

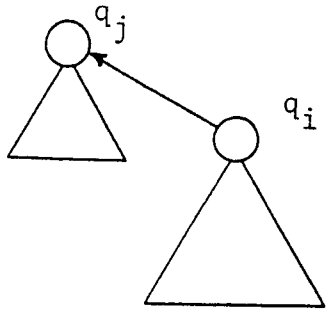
$MERGE(q_i, q_j)$ is an operation between the roots q_i and q_j which makes q_i the rightmost son of q_j.

$Q_i.WEIGHT$ denotes the number of internal nodes and leaves stored in the tree rooted at q_i.

Effectively, the operation $MERGE(q_i, q_j)$ is transforming the forest from

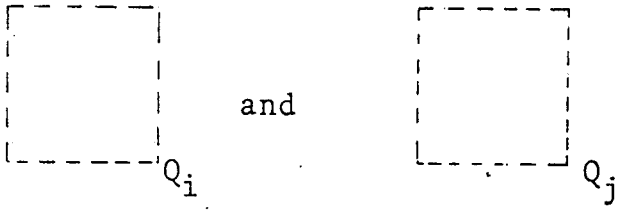


into



Consider what would happen in case a $MERGE(q_i, q_j)$ - instruction is executed.

Suppose the records are



If $Q_i.\ WEIGHT \leq Q_j.\ WEIGHT$ we simply attach q_i to q_j by setting $Q_j.\ WEIGHT = Q_j.\ WEIGHT + Q_i.\ WEIGHT$ and setting $Q_i.\ FATHER$ equal to the address of a freshly allocated record at the *end* of the current list of q_j registrating a *real* link (because this attachment is indeed what was intended by the MERGE - instruction). We do *not* change the contents of $Q_j.\ REAL\ ROOT$ here, and it seems as if we attached q_i to *that* node.

Note that $Q.\ FATHER$ is always defined according to the physical (apparent) tree.

Figure 2 illustrates the operations mentioned.

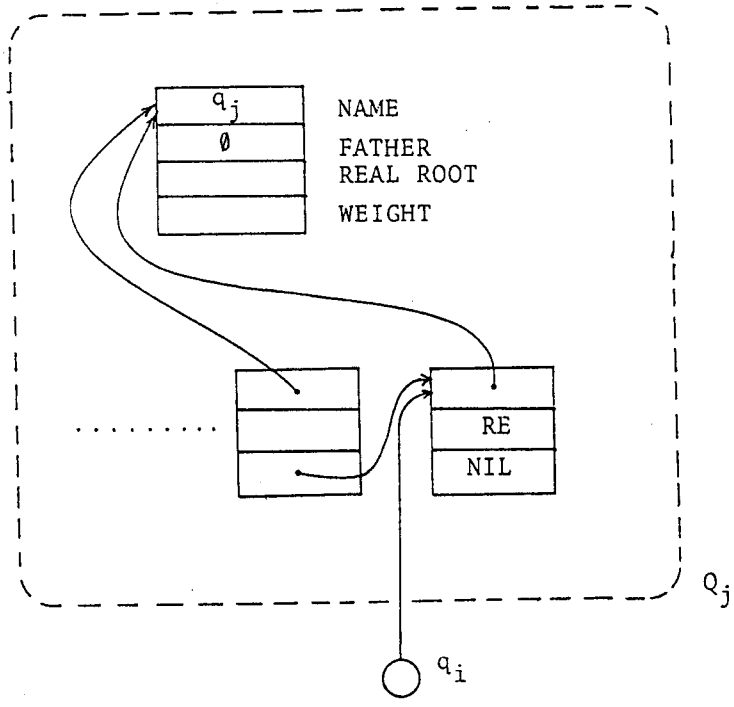


Figure 2

In case we have $Q_i.\,WEIGHT > Q_j.\,WEIGHT$, we have to operate in a different way, because in order to keep the tree balanced we must do the exact opposite of what the MERGE - instruction wants us to. Thus we attach q_j to q_i but record the link as *imaginary* (because we should really proceed the other way round). We set

$Q_i.\,REAL\ ROOT = Q_j.\,REAL\ ROOT$ (which is the most important step here)

$$Q_i.\,WEIGHT = Q_i.\,WEIGHT + Q_j.\,WEIGHT$$

and $Q_j.\,FATHER$ points to the address of a freshly allocated record at the end of the current list of q_i registering an *imaginary* link. We illustrate this in

Figure 3.

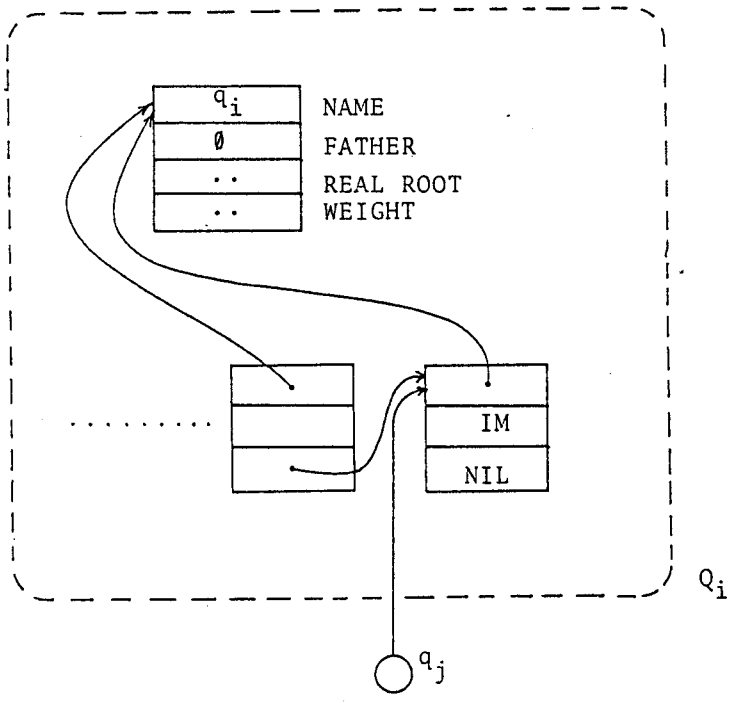
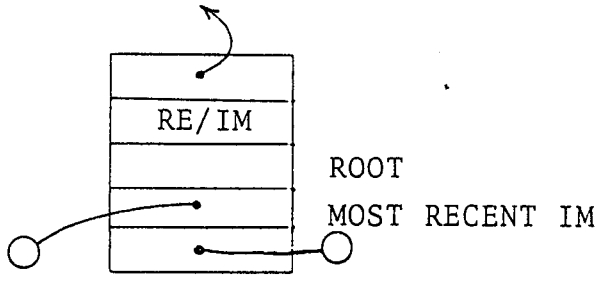


Figure 3

In steps like these the value stored in field *REAL ROOT* can change.

At certain steps in algorithms later which traverse the tree following FATHER - links we may want to know the "real root" at the time of entering a particular son. It could be stored in an additional field of the particular record in the list, but for later purposes it is still insufficient.

Instead, we will store even *more* information and now fix the records in the list of the form



In the course of the construction , *ROOT* will always denote the "REAL ROOT" of the attached q-node, and "MOST RECENT IM" will point to the list-element containing the most recent *imaginary* link *to the left.* (In particular, in case the record itself has IM in field 2, the pointer will simply hold the address of the record itself).

It is important to observe that as the list is built and records are added on the right, the contents of the field "MOST RECENT IM" can be determined *solely* on the basis of information *in the current record* and *in the immediately*

preceding record.

Effectively, keeping track of the MOST RECENT IM field is a form of ***path - compression***, of the only type permissible here.

The initial information q-nodes carry can be described as follows:

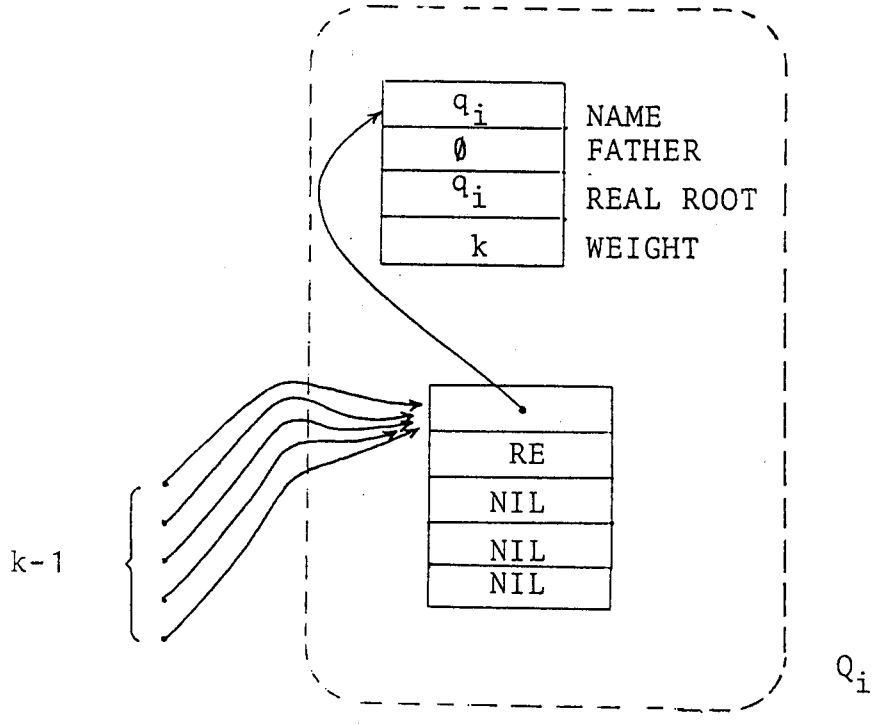


Figure 4

The description of the data structure also explains how MERGE - instructions will be carried out.

It should be noted that the apparent tree that is built, is constructed essentiallly following the regime of a UNION(-FIND) programm with balanced unions [Aho, Hopcroft, Ullman, 74]. Thus we conclude

Proposition 1: The "height" of any apparent physical tree in the given forest-implementation is $O(\log n)$. ∎

2.2 Finding Nearest Common Ancestors

The algorithm for answering *nca* - requests consists of two essential stages. First we find the nearest common ancestor in the *apparent tree*, and next we use the information stored at the ancestral node to determine which one is the *"real"* nearest common ancestor.

It is important to note here what we really want to find in the apparent tree. We do not just want to obtain the nearest common ancestor, but rather the *explicit* list-element in the filter (the list) through which we *enter* the nearest common ancestor.

Given two nodes x and y in the apparent tree, let p and q be the respective ancestors of x and y which are sons of $nca(x, y)$.

Now we follow the pointers $p.FATHER$ and $q.FATHER$, entering the "filter" of node q_i at records S and T (by inspecting a counter stored in yet another field of these records we can find out that S precedes T in the list, although different ways of establishing this may be given).

Because $p \neq q$, we know that $S \neq T$ and we may assume the situation as sketched in Figure 5.

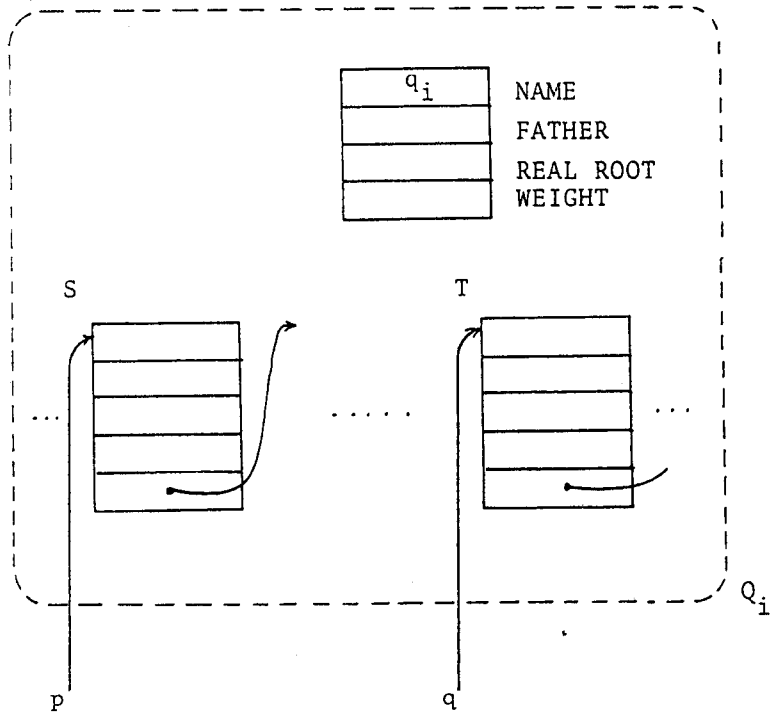
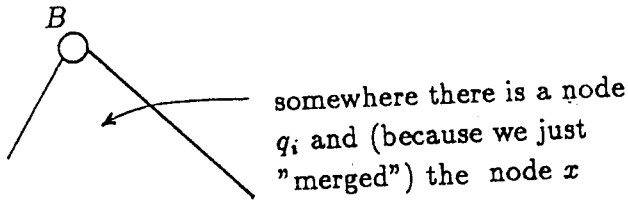


Figure 5

The "real" nearest common ancestor of x and y lies *somewhere* in the subtree headed by q_i, and we claim that we can find out where it is from direct inspection of the record - information.

Proposition 2: The "real" nearest common ancestor of x and y is
T. MOST RECENT IM. ROOT.

Proof: Suppose the "real" tree looked as follows at the time record S was added to the list



By construction the "root" of the real tree at this moment is *S. ROOT* (if the link was *imaginary*) or *S. MOST RECENT IM. ROOT* (if the link in *S* was *real*).

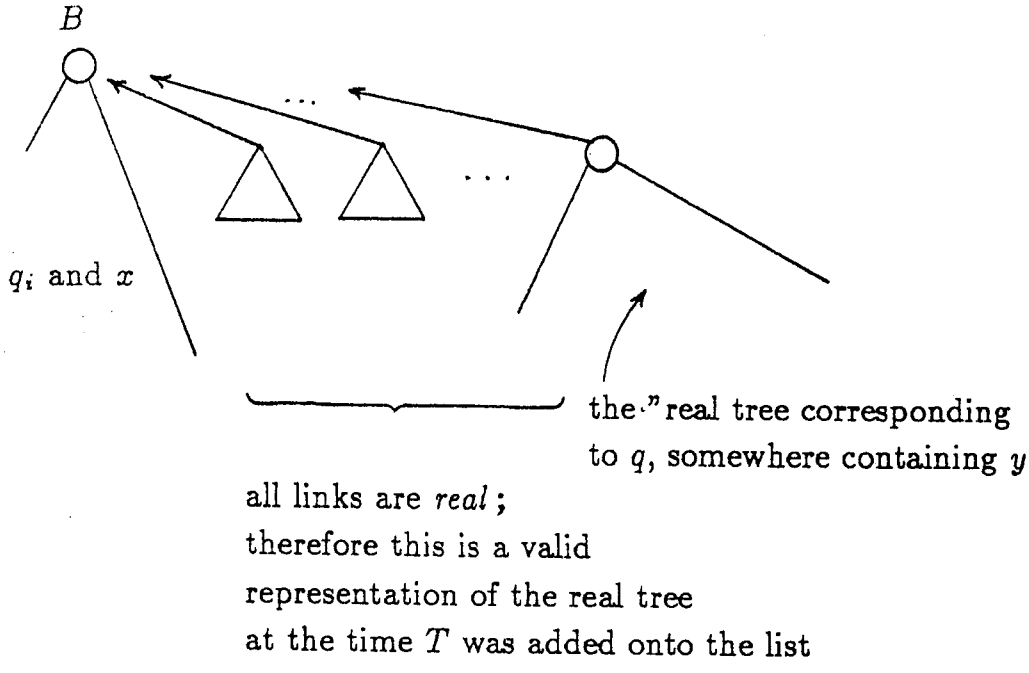
Now consider two cases.

Let us first assume that

S. MOST RECENT IM = *T. MOST RECENT IM.*

That means that all links stored *after S* must be *real*, and the situation can be reconstructed (in the *real* tree) as follows

(note that y is somewhere in the subtree headed by q or, in other words, "headed" by T)



Clearly, the nearest common ancestor of x and y is the "real" root of the tree at the time *S* was added, and thus equal to

S. ROOT (if the link stored in *S* was imaginary),

or

S. MOST RECENT IM. ROOT (otherwise).

In the first case, however, *MOST RECENT IM* was made to point to *S itself* (the consistency of which will now be apparent) and in *both* cases the real root is

$S.\ MOST\ RECENT\ IM.\ ROOT = T.\ MOST\ RECENT\ IM.\ ROOT$

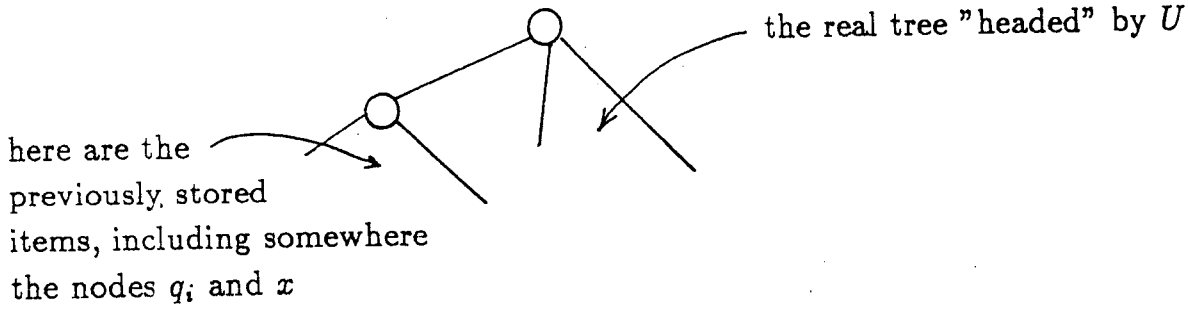
as it was shown.

Let us next assume that

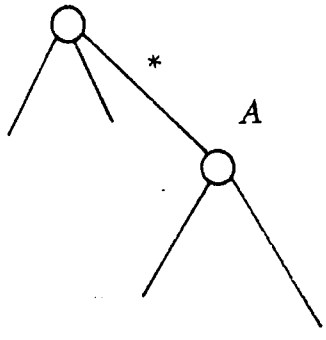
$S.\ MOST\ RECENT\ IM \neq T.\ MOST\ RECENT\ IM.$

That means that some links stored *after* S must be *imaginary* and the apparent tree must be "rotated" back to obtain the real tree.

Let the record pointed to by $T.\ MOST\ RECENT\ IM$ be U. This record will be somewhere between S and T (and let us assume it is $\neq\ T$). The following argument holds just as well when $U = T$. There was an imaginary link stored in U, and thus the real tree at the time U was added is of the form



Note that in the apparent tree we recorded

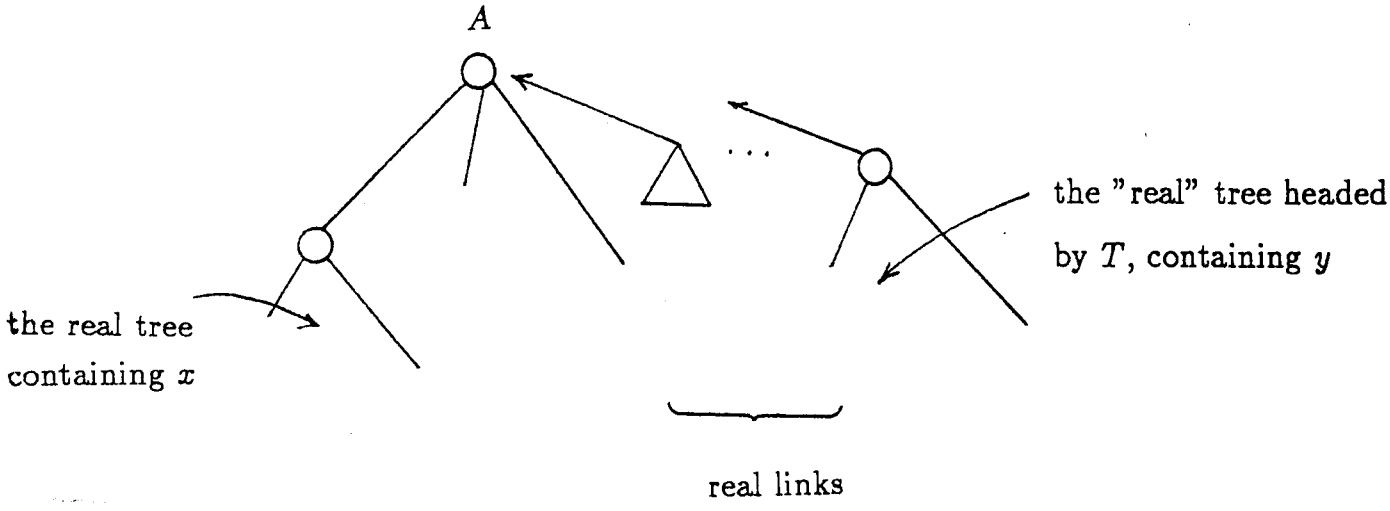


By construction of the data structure the "real" tree at that time was

$(A\ =)\ U.\ ROOT$

All links stored *after* U (but up to T) must be real, and the real tree must

be as follows



Therefore A is the real nearest common ancestor of x and y, and it is equal to

$$U.\,ROOT = T.\,MOST\ RECENT\ IM.\,ROOT$$

as was to be shown. ∎

2.3 Preprocessing on the static tree T

The following procedure $PREMERGE$ visits the given tree T in a pre-order traversal and merges every time the visited father with its sons. The resulting tree is the balanced tree BT. In addition to the information given in section 2.1 we also use the information $v.\,REPR$, which points to the recent root of the compressed tree which includes node v after a sequence of MERGE-instructions.

Let r be the root of T.

```
Proc PREMERGE(r);

begin
      comment Let son(v) be a son of v;
         Let bit(v) = 1 if v is completely visited and equal to 0 otherwise;
         Let father(v) be the father of v in tree T and v. FATHER
          the father of v in the compressed tree;
      end of comment;
    for all v in T
      do
         v. WEIGHT := 1;
         v. REPR := v;
         if v is a leaf then bit(v) := 1
      od;

    if bit(v)
    then
          MERGE(v, father(v). REPR);
          if father(v). REPR. WEIGHT < v. WEIGHT
          then father(v). REPR := v fi;
          Set bit(father(v)) = 1 if for each son s of father holds: bit(s) = 1

    else
         for each son of v
           do
              PREMERGE(son(v))
           od
    fi
end
```

Figure 6b illustrates the result of the procedure $PREMERGE$ when ap-

plied to the tree of Figure 6a. Nodes are identified by their preorder-number.

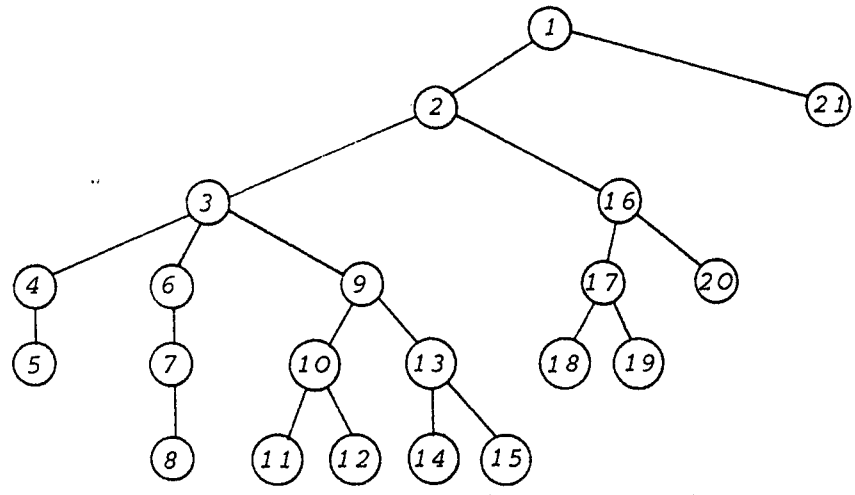


Figure 6a: The static tree T

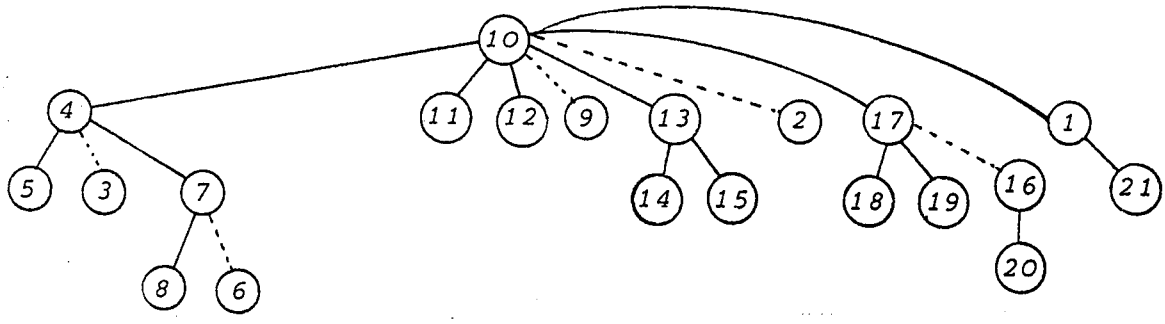


Figure 6b: The balanced tree BT

3. The preprocessing of BT

In this section we show how to preprocess the balanced tree BT in which tree T is embedded in order to be able to quickly answer *nca* queries. We base our construction on the following theorem proved in [Tsakalidis, 84] using the following operations on the nodes of the tree:

1. Insert-Node(x, y): Insert a new node x as the rightmost son of y in the tree.
2. Pre(x, y): Return true iff x occurs before y in a preorder traversal of the tree.
3. Post(x, y): Return true iff x occurs before y in a postorder traversal of the tree.
4. Ancestor(x, y): Return true iff x is an ancestor of y in the tree.

These operations will be performed *on-line*, i.e., each operation is completely executed before the next operation is considered.

Theorem 2: If the positions of the nodes referred to in the operations are given, then we can perform a sequence of m arbitrary Insert-Node in an initially empty tree structure BT in time $O(m)$, and each Pre, Post and Ancestor operation needs time $O(1)$. The space used is $O(m)$.

Proof: (Sketch)

In addition to the usual implementation of the tree BT we store the nodes of BT in a list K, which combines the information about the preorder and postorder traversal of BT. The implementation chosen for the list K allows us to perform m arbitrary insertions on given positions in time $O(m)$ and to answer in $O(1)$ time whether a given element x lies to the left or to the right of another given element y in the list K.

Each tree node x is represented by two elements x_1 and x_2 in K and has two pointers which point to these elements; the order of the element x_1 [x_2] correspondes to the time instant, at which the node x is visited for the first [last] time by a traversal of BT. Then a question about the ancestor-relationship between x and y can be answered by two questions on the relative positions of x_1, y_1 and x_2, y_2 in *constant* time, because

Ancestor(x,y)=true iff Pre(x,y)=true and Post(x,y)=false

and the following holds:

Pre(x,y)=true iff x_1 occurs before y_1 in K, and

Post(x,y)=true iff x_2 occurs before y_2 in K.

Each insertion of a node x in T will be followed by the respective insertions of elements x_1 and x_2 in K. All lists used need no more than $O(m)$ space. (For a detailed proof, see [Tsakalidis, 84].) ∎

3.1. The Data Structure

Let BT be a tree.

Definition 2: A *path* or *search path* from the root R of BT to a node x is a sequence of nodes $v_0, \cdots, v_k$ with $v_0 = R$, $v_k = x$ and $father(v_i) = v_{i-1}$ for $1 \leq i \leq k$, where $father(R) = nil$. If $d(x)$ denotes the depth of node x, i.e., the distance of node x from R, then $d(x) = k$ is the length of the search path from R to x.

A *segment* S of a search path is a sequence of nodes $v_i, \cdots, v_{i+s}$ with $father(v_j) = v_{j-1}$ for $i+1 \leq j \leq i+s$. Then s is the *length* of the segment S. The *highest [lowest]* node of a segment is the node with the minimal [maximal] distance to the root R of BT.

A segment is called *block* if its lowest node has an additional pointer which connects it with its highest node. Blocks or segments can overlap if they have the last highest nodes in common.

Our problem is to efficiently compute the nearest common ancestor of two given nodes x and y of BT (denoted by $nca(x,y)$).

Let $d(x) < d(y)$ and $v_0, ..., v_k = x$ be the search path of x. We can start from x and we search between $v_0, ..., v_k$ for the minimal i such that

$ancestor(v_{i+1}, y) = false$ and $ancestor(v_i, y) = true$.

Then $v_i = nca(x,y)$ and, according to Theorem 2, each comparison about ancestral relationship among y and a node visited can be tested in $O(1)$ time.

We try to speed up this search on the search path. If the search path were stored in arrays we could use binary search. But we are working on a pointer machine and arrays are not allowed. In order to simulate this binary search we have to equip some nodes with pointers. Since additional pointers need more space, we try to save space by grouping the nodes into *blocks* of special length. $D.1, \cdots, D.4$ present the data structure used.

($D.1$) The tree BT is represented by a list structure.

($D.2$) The nodes of BT are also stored in a *list* K precisely as it is described in Theorem 2. List K allows us to know in $O(1)$ time whether two given nodes x and y have any ancestor-relationship in T.

($D.3$) The nodes of BT are grouped into *blocks* of special length. Each newly inserted node v in BT belongs to a block of length *at most* $f(d(v))$ ($f : N \rightarrow N$); this block is defined as a *segment* of the path from v upon to the root of BT. Blocks on the same search path are disjoint but blocks on different search paths can overlap if these paths overlap.

($D.4$) For each block B we define its **representative** r; r is a node of B and the highest ancestor in BT of all nodes of B. It is equipped with at most $\log(d(r))$ pointers pointing to some ancestors of r which are declared as representatives of higher blocks. In order to save space we group the representatives $r_1, \ldots, r_k$ into a horizontal block so that all r_i's, $1 \leq i \leq k$, have the same depth and if re is a representative which is the nearest common ancestor of all r_i's, $1 \leq i \leq k$, in BT then there is no other representative on the paths from r_i upon to re. We specify r_1 (the one on the left) as the **main representative**. Only r_1 is equipped with the pointers mentioned before.

For more details and illustrations concerning the data structure used and the necessary information we refer to [Tsakalidis, 88].

Next we give precisely the information stored in the pointers of the main representatives. The main idea is to simulate the binary searching as it can be performed on the RAM on the pointer machine in the case that the pointers point to the same direction. For each pair of nodes x, y we determine the searching direction on the path $v_k, \ldots, v_1, v_0$ from the node $x = v_k$ to the root v_0 according to the value of $ancestor(v_i, y)$ and we continue this search following the proper pointer of the main representative.

Definition 3: Let r be a main representative node and $RD(r)$ its *representative − depth*, defined as the number of representatives on the path in BT between r and the root of BT. Then $p_i(r)$ points to the representative s which is an ancestor of r with $RD(s) = RD(r) - 2^i$ for $0 \leq i \leq \log RD(r)$.

We consider now the embedding of an arbitrary tree T into the balanced tree BT.

In addition to the information of the embedding of T where each node includes the proper information (see section 2) and pointers to the respective sons, we store the following information for the nodes of BT:

I.1: Each node includes a father pointer.

I.2: $d(v)$ is stored in each node v.

I.3: Each node v includes:

a) A pointer $pr(v)$ to the representative node $r(v)$ of the block into which v belongs.

b) a cell which stores the distance $bd(v)$ from v to $r(v)$, i.e., $bd(v)$ is the number of nodes between v and $r(v)$ on the path in T.

I.4: a) A main representative node includes the pointers p_i according to the definitions given. These pointers are realized as doubly linked list. More precisely, let r be a main representative and $p_0, \ldots, p_k$ its pointers which point to the representatives $r_0, \ldots, r_k$. Then r includes a pointer to p_0 and all $p_0, \ldots, p_k$ are realized as a doubly linked list, i.e. there is a pointer from p_{i-1} to p_i and from p_i to p_{i-1}.

b) A representative r includes a *horizontal* pointer $hp(r)$ which points to the respective *main representative* of the same depth and a vertical pointer $vp(v)$ which points to the nearest main representative which is a descendant of r.

c) Each pointer list $p_0, p_1, \ldots, p_k$ of a main representative r is equipped with a doubly linked list $q_0, q_1, \ldots, q_k$. The q_i's are defined as follows. Let s be a representative; then $mr(s)$ denotes either s if s is a main representative or otherwise the node $hp(s)$. Then $q_i(r)$ points to the i-th pointer of the main representative $mr(p_i(r))$, if this pointer exists, otherwise $q_i(r)$ does not exist, i.e. there is a pointer from $p_i(r)$ to $q_i(r)$ and $q_i(r)$ points to p_i-pointer of $mr(p_i(r))$, if this pointer exists.

I.5: Each node A includes two pointers which point to the elements A_1 and A_2 in the list K respectively as it was explained in Theorem 2.

I.6: Each node v includes a pointer which points to the respective node in the set structure D if the node v is deleted from T.

3.2 The Algorithm and its Complexity

Given the balanced tree BT, we have to preprocess it by inserting all its nodes anew and constructing the data structure. For an inserted node, we have to arrange the block and the pointer of the newly created representative. The

following algorithm $Insert(x, y)$ inserts the new node x as the rightmost son of a given node y.

```
  Proc Insert(x,y);
begin
      if T is empty then make x the root of T fi;
      Make x the rightmost son of y by adding a new pointer from y to x;
      Define the father-pointer on x (I.1)
      Set d(x) := d(y) + 1; (I.2)

      if f(d(x)) = f(d(y)) and
      y belongs into a block of length < f(d(y))   co check I.3.b for y;
L1:   then Insert x into this block by defining the pointer
            of I.3.a which points to the same representative of y;
             bd(x) := bd(y) + 1;    co information I.3.b on x;
L2:   else Create a new representative x.
                    co let s be the next representative which is ancestor of x;
            s := pr(y);
            if x is a main representative   co check if vp(s) is undefined;
            then vp(s) := x

                    Arrange all the pointers p_i(x)'s and q_i(x)'s;

                      co Lemma 1 explains exactly this arrangement;

            else hp(x) := vp(s);
            fi
      fi
      Insert x in the list K;
 end
```

For preprocessing BT we traverse BT in a breadth-first manner and insert each node visited by means of procedure $Insert(x, y)$.

Next we estimate the complexity of a sequence of m insertions and deletions in a tree BT. First we give a lemma that is necessary for estimating the cost of the arrangement of the pointers for I.4.a.

Definition 4: Let the *representative path* of a node v be the sequence of representatives on the search path from v up to the root and the *ancestor representative* of v any representative lying on the representative path of v.

Lemma 1: Let r_d be a newly created main representative on the representative path $r_d, r_{d-1}, \ldots, r_1, r_0$, where r_0 is the root of T. Then the arrangement of all $p_j(r_d)$'s and $q_j(r_d)$'s according to I.4.a and I.4.c costs $O(\log d)$ time.

Proof: Immediately from Theorem 3 and Lemma 2. ∎

More precisely, the query time is $O(\log min\{dbt(x), dbt(y)\})$, where $dbt(x)$ is the depth of node x in the balanced tree BT.

References

AHO, A., J. HOPCROFT, J. ULLMAN, "The Design and Analysis of Computer Algorithms", Addison-Wesley Publ. Comp., Reading, MA, (1974)

AHO, A., J. HOPCROFT, J. ULLMAN, "On finding Lowest Common Ancestors in Trees", *SIAM Journal of Computing*, Vol. 1, No 1, pp. 115-132 (1976)

DIETZ, P., and D. SLEATOR, "Two Algorithms for Maintaining Order in a List", in : Proc. 19-th Annual ACM Symp. on Theory of Computing, New York, pp. 365-372, (1987)

HAREL, D., and R.E. TARJAN, "Fast Algorithms for finding Nearest Common Ancestors", *SIAM Journal of Computing*, Vol. 13, pp. 338-355 (1984)

MAIER, D., "An Efficient Method for storing Ancestor Information in Trees", *SIAM Journal of Computing*, Vol. 8, No 4, pp. 599-618 (1979)

SCHIEBER, B., and U. VISHKIN, "On Finding Lowest Common Ancestors: Simplification and Parallelization", Technical Report 63/87, New York University, New York, (1987)

SLEATOR, D., and R.E. TARJAN, "A Data Structure for Dynamic Trees", *J. of Comput. System Sci. 26*, pp. 362-391 (1983)

TARJAN, R.E., "A Class of Algorithms which require non-linear time to maintain Disjoint Sets", *J. Comput. System Sci. 18*, pp. 110-127 (1979)

TSAKALIDIS, A.K., "Maintaining Order in a Generalized Linked List", *Acta Informatica 21*, pp. 101-112 (1984)

TSAKALIDIS, A.K., "The Nearest Common Ancestor in a Dynamic Tree", *Acta Informatica 25*, pp. 37-54 (1988)

van LEEUWEN, J., "Finding Lowest Common Ancestors in less than Logarithmic Time", unpublished report (1976)