Algorithmica (2002) 33: 494–510
DOI: 10.1007/s00453-002-0935-z

Algorithmica

Optimal Solutions for the Temporal Precedence Problem

G. S. Brodal,[1] C. Makris,[2] S. Sioutas,[2] A. Tsakalidis,[2] and K. Tsichlas[2]

Abstract. In this paper we refer to the *Temporal Precedence Problem* on *Pure Pointer Machines*. This problem asks for the design of a data structure, maintaining a set of stored elements and supporting the following two operations: *insert* and *precedes*. The operation *insert*(a) introduces a new element a in the structure, while the operation *precedes*(a, b) returns true iff element a was inserted before element b temporally. In [11] a solution was provided to the problem with worst-case time complexity $O(\log \log n)$ per operation and $O(n \log \log n)$ space, where n is the number of elements inserted. It was also demonstrated that the *precedes* operation has a lower bound of $\Omega(\log \log n)$ for the *Pure Pointer Machine* model of computation. In this paper we present two simple solutions with linear space and worst-case constant insertion time. In addition, we describe two algorithms that can handle the *precedes*(a, b) operation in $O(\log \log d)$ time, where d is the temporal distance between the elements a and b.

Key Words. Algorithms, Dynamic data structures, Computational complexity.

1. Introduction.

The topic of this paper is the Temporal Precedence problem [11] on a Pure Pointer Machine (PPM). In a PPM [8], [10]–[12], memory consists of a finite but extendable collection of nodes, with each node being uniquely identified through an address and a finite collection of registers. A special address *null* is used to denote an invalid address. This special address is the only *constant* which can be explicitly assigned to a register. Each register can contain an address while each *node* consists of a fixed number of fields, each of which contains only one address. The *instructions* permitted by this machine model are: (i) the creation of a new *node* (returning its address), (ii) the transfer of addresses between two registers, (iii) the transfer of addresses between registers and the fields of a *node*, (iv) conditional jumps, where the only conditions allowed are *true* and the equality comparison between two registers. The only possibility to access a node is by following pointers from previously accessed nodes. A *program* in this model of computation consists of a finite numbered sequence of instructions, with each instruction being uniquely identified by one number and being considered to have unit cost.

Based on the above description of the computational model used, we can now give a precise definition of the operations to be performed by the devised data structure:

- *insert*(a): Create a node in the data structure with address a, representing the new element. Insert the element pointed to by a into the data structure and return an

[1] BRICS (Basic Research in Computer Science), Department of Computer Science, University of Aarhus, Ny Munkegade, DK-8000 Arhus C, Denmark.
[2] Department of Computer Engineering & Informatics, University of Patras, 26500 Patras, Greece, and Computer Technology Institute (CTI), P.O. Box 1192, 26110 Patras, Greece.

Received July 31, 2000; revised June 25, 2001. Communicated by K. Mehlhorn.
Online publication April 19, 2002.

address that is a *handle* to the element a. How to store this return address is to be handled by the user of the data structure.

- *precedes*(a, b): Given two addresses a and b of nodes representing elements stored in the data structure, return *true* if the element with handle a was inserted before the element with handle b. The return value is encoded by an address, such that *true* is represented by an address different from *null* and *false* is the *null* address.

The problem is important in two distinct ways. Firstly, it is the first problem that clearly shows that the PPM is a less powerful computational model than that of a pointer machine, since in a pointer machine the specific problem can be solved trivially (using time-stamps) with worst-case constant time for each operation, by using linear space. In [11] a non-constant lower bound on the *precedes* operation has been proved and as a consequence it was concluded that a pointer machine is a more powerful computational model than that of a PPM. Secondly, the problem is related to a very concrete problem that arises in parallel implementations of logic programming languages. More specifically, in [8] a tight relationship between the *And-Parallelism* problem and the problem of *time-stamping* on pointer machines was presented. In the *And-Parallelism* problem, which arises from don't care non-determinism, given a resolvent $B_1, B_2, \ldots, B_n$ multiple subgoals in the resolvent can be concurrently reduced. The computation can be visualized through a tree, the so-called *And-Tree*, the root of which is labeled with the initial goal. If a node contains a conjunction $B_1, B_2, \ldots, B_n$, then it will have n children, the ith child of the node is labeled with the body of the clause used to solve the B_i. The main problem here is to find a way to manage efficiently the unifiers produced by the concurrent reduction of different subgoals. Two subgoals B_i, B_j $(1 \leq i \leq j \leq n)$ in the $B_1, \ldots, B_n$ resolvent should agree in the bindings of all the variables (*dependent* variables in terms of Prolog) that are common to them. In [8] it was shown how the problem of correct binding and assignment of variables in such parallel environments can be reduced to the problem of handling a dynamic tree, capable of growing and shrinking at the leaves and detecting whether the first node is the leftmost leaf in the subtree rooted at the second node. Both operations can be reduced (for more details see [8]) to the *insert* and *precedes* operations of the Temporal Precedence problem.

The problem is also related to the issue of maintaining order information in a dynamic list, which was studied by Tsakalidis [13] and Dietz and Sleator [3]. The problem, which was initially handled by Ranjan et al. [11], is simpler and more fundamental, and consequently the model used for a finer analysis was the PPM, as opposed to the pointer machine models with arithmetic capabilities, that were used in the aforementioned works.

The scheme proposed in [11] provides an optimal implementation of the *precedes* operation proving an upper and a lower bound. The solution described is based on the use of *balanced distribution trees*, whose main property is that the degree of a node at depth x is equal to the number of nodes at depth $x - 1$. Recursive restructuring is used in order to remedy the problem of the large degree of the nodes, which can be up to $\sqrt{n}$. However, this solution requires $O(n \log\log n)$ space and the insertion operation is not optimal, in the sense that it is not performed in worst-case constant time.

In this paper we show that by modifying appropriately the data structure of Ranjan et al. we can create two structures that can handle insertions in worst-case constant time and precedence queries optimally. We also provide two new data structures, using techniques from the finger search tree literature that achieve time complexity asymptotically equal to $\log\log d$, where d is the temporal distance between the elements that are given as input in the *precedes* operation.

The idea behind our first solution is based on the partition of the dynamic ordered set S into an ordered collection of buckets, each of which has $O(\log\log n)$ size. In this way we insert only the representatives of buckets into the structure of [11], in order to reduce both the space and insertion time complexities. The amortized constant insertion time is made worst-case by spreading the overall insertion (of a representative) cost over several updates to S. The second solution stems from a close inspection of the recursive application of the balanced distribution tree in the data structure presented in [11]. The third and the fourth solutions are based on known techniques used in finger search trees [2], [4]. Apart from the improvement on the time complexity of the *precedes* operation, the solutions are interesting since they show the applicability of finger tree techniques on trees of large degree, that are structured recursively at all levels.

We must mention here that the data structures presented exhibit some similarities with van Emde Boas' *stratified trees* [14], [15] implementing priority queues. This data structure simulates binary searching on the levels of a binary tree by recursive by applying the same logical organization a number of times. This logical organization allows the stratified tree to perform a search operation in $O(\log\log n)$ time. However, there are two basic differences between our structures and stratified trees: (i) the $O(\log\log n)$ time bound in the stratified trees maintenance operations stem from the use of the RAM model with uniform time measure, a direct translation of this structure in the Pure Pointer Machine model will incur a time cost of $O(\log n)$, (ii) stratified trees require an a priori knowledge of n, a requirement that is not a prerequisite for our structures.

In the following section we give a brief description of the structure proposed in [11] to handle the Temporal Precedence problem, in Section 3 we describe two data structures that allow us to achieve optimal time and space bounds, in Sections 4 and 5 we outline the finger search methods applied on this problem and finally in Section 6 we conclude with some remarks.

2. An Overview.

In this section we recall the construction of [11] for handling the Temporal Precedence (TP) problem. A straightforward solution is to maintain the inserted elements in a simple linear linked list. Insertions are allowed to be performed only at the end of the list (that is, the sequence of insertions uniquely determines the temporal relations), while the *precedes* operation requires the traversal of the list, until one of the two elements is found. The complexity of this operation is linear in the worst case. In a simple binary tree-based scheme the elements inserted are placed in a binary tree that is expanded level by level, from left to right. For an arbitrary tree structure we define the *levels* of the nodes as follows: the root is at level zero and the children of nodes with level i are the nodes with level $i + 1$.

The insertion of an element is performed easily in worst-case constant time by maintaining a pointer to the last inserted node, a pointer to the last node of the last level that

is full and a pointer to the first node in the current level. The operation *precedes*(a, b) can be implemented using the following idea:
precedes(a, b) should return true iff one of the following three conditions holds:

(i) a is the root and b is not;
(ii) a is immediately to the left of b;
(iii) $a' = parent(a)$ and $b' = parent(b)$ exist and *precedes*(a', b') is true.

The time complexity of the *precedes* operation is logarithmic in the number of elements n. The next step towards the asymptotic reduction of the time complexity of the *precedes* operation is to use a different tree that exhibits better asymptotic time complexities when applied to this problem. This tree is termed the *balanced distribution tree*.

In a balanced distribution tree the degree of the nodes at level i is defined to be $d(i) = t(i)$, where $t(i)$ indicates the number of nodes present at level i. This is required to hold for $i \geq 1$, while $d(0) = 2$ and $t(0) = 1$. It is easy to see that we also have $t(i) = t(i-1)d(i-1)$, so putting together the various components, we can solve the recurrence and obtain, for $i \geq 1, d(i) = t(i) = 2^{2^{i-1}}$. One of the merits of this tree is that its height is $O(\log\log n)$, where n is the number of elements stored in it. One could allege that by applying the conditions of the binary tree scheme it would be easy to support the *precedes* operation in $O(\log\log n)$ time. This is not completely true since the third condition cannot be applied efficiently, because of the non-constant number of nodes with a common father at the low levels of this tree. For example, two nodes at level i with the same direct ancestor belong to a collection of $2^{2^{i-2}}$ elements. Thus, a simple scan of these elements (as in the binary tree implementation) will result in a time inefficient implementation of the *precedes* operation. The organization proposed in [11] faces this problem by repeating the same kind of tree-structuring in each set of nodes having the same direct ancestor, and doing this recursively until no more than two nodes share the same direct ancestor (this is the innermost nesting level). Generally, we refer to a specific tree-structuring of a set of nodes with a common father as a nesting level. The first nesting level or innermost nesting level corresponds to the last recursive tree-structuring while the outermost nesting level corresponds to the tree that stores the entire set of elements. Thus, inductively, a set of trees corresponding to a nesting level k, structures the elements stored in a tree at nesting level $k + 1$. The number of nesting levels is $O(\log\log n)$ as shown in [11]. Thus, by making some small modifications to the *precedes* implementation we may achieve optimal $O(\log\log n)$ time complexity for the specific operation.

A final detail concerns the storage requirements for each element in the nested structure. Since there are a non-constant number of levels and the computational model used is the PPM, it is not possible to allocate a single memory node to represent each element. The solution is to keep distinct representations of the trees that are generated. Each element is represented by a linear list of nodes with each node representing the appearance of the element in each nested level. In each level we deal directly with the representatives of the elements and since we only move through successive levels it is easy to access the correct representative in constant time.

We now briefly give the implementations of the operations *insert* and *precedes*. The new element is inserted in all nesting levels by constructing a list as the one described above. The insertion of each node at each nesting level requires the creation and the modification of a constant number of nodes. The counting abilities needed to detect when one group of nodes, a level or a tree has been completed can be simulated by constant time pointer movements (for more details see [11]). Thus, the time complexity for the insertion of an element is bounded from above by $O(\log \log n)$. In the *precedes* operation we are given two elements to be compared, that is we are given handles to the first node (outermost nesting level) of the linear list of nodes, corresponding to each element. Firstly, an initial check is made to verify if one node is immediately to the right of the other, in which case the answer is obvious. Otherwise, we climb the tree structure at the outermost level of nesting until we reach two nodes with the same direct ancestor (call it v). Then we switch to the next level of nesting (the tree structure storing the set of sons of v) and we apply recursively the same process. By computing a telescoping sum it is easy to show (see [11]) that the time complexity is $O(\log \log n)$ while the space complexity is $O(n \log \log n)$.

3. Two Linear Space Solutions with Worst-Case Constant Insertion Time. In this section we describe two data structures for the TP problem. The first structure is based on the bucketing technique and improves the one described in [11]. The second structure is based on an observation concerning the structure of the nesting levels. The space complexity for both structures is linear while the insertion time is constant.

Firstly, observe that a linear list implementation of the TP problem supports fast insertions but is slow for precedence queries while the balanced distribution tree implementation supports fast queries but slow updates. The essence of the bucketing method is to get the best features of two different data structures, designed for the same problem, by combining them into a two-level data structure. The data to be stored is partitioned into buckets and the chosen data structure for the representation of each individual bucket is different from the representation of the top-level data structure, representing the collection of buckets (for similar applications of this data structuring paradigm see also [6], [9], and [13].

More specifically, we partition the elements of the stored set into contiguous buckets of size $O(\log \log n)$, with each bucket being represented by the linear list scheme and we store the first element of each bucket in the balanced distribution tree scheme as its representative. When an item is inserted, it is appended to the tail of the list implementing the last incomplete bucket. If the size of this bucket becomes $O(\log \log n)$, then a new bucket is created containing only the newly inserted element. We also spend $O(\log \log n)$ time in order to insert this element into the top-level data structure. We have a total of $O(n/\log \log n)$ representatives, each of which must be inserted in at most $O(\log \log(n/\log \log n)) = O(\log \log n)$ nesting levels. Amortizing the $O(\log \log n)$ insertion cost, over the $O(\log \log n)$ size of each bucket, we achieve an amortized constant insertion cost. With the same reasoning as above the total space is linear. We eliminate the amortization by spreading the time cost for the insertion of the representative over the next $O(\log \log n)$ updates in its bucket, that is each insertion in the last bucket triggers the insertion of the bucket's representative into a new nesting level. The *precedes* operation is similar to that described in [11]. First we check whether the two nodes belong to the same bucket. If they do, then a linear traversal of the list corresponding to this bucket is enough. If the nodes belong to different full buckets, then we proceed by checking the representatives of the respective buckets in the top-level data structure. If one of the two query nodes belongs to the bucket under construction we answer the query immediately. Since the query algorithm does not use any information concerning the incompletely inserted element it is obvious that the query algorithm is correct and has time complexity equal to $O(\log\log n)$.

One last issue we have to consider is that of the size of buckets. The size of the buckets must be asymptotically equal to $O(\log\log n)$. However, n is constantly changing and as a result the size of the buckets must also change accordingly. We remedy this problem by applying the *global rebuilding technique* [7]. More specifically, we partition the dynamic ordered set S, subject to insertions of elements, into an ordered collection $B = \{b_1, \ldots, b_l\}$ of buckets such that $\max b_i \le \min b_{i+1}$, $\forall i$ (the max, min and $\le$ relations refer to the temporal order of the elements). Due to the fact that we do not have a priori knowledge of n, the size of the buckets is not fixed. As a result, the size of each bucket is a function $s(\widetilde{n})$, where $\widetilde{n}$ is the current number of inserted elements.

We use two structures: *Old* and *New* and two functions $f(n)$ and $s(n)$ with the property that $s(f(n)) = s(n) + 1$. Normally only *Old* exists, and during the course of the algorithm only the *Old* structure is used for querying. Suppose that at some time (time is defined with respect to update operations) in the *Old* structure there are n elements and the bucket size is $s(n)$. We set the size of a bucket equal to $s(f(n)) = s(n) + 1$ and a construction procedure is initiated to build a new structure (*New*) containing all the elements of *Old* with the specific bucket size. Meanwhile, insertions and queries are made in *Old* and the changes must be reflected in *New* too. By the time we have inserted $n/2$ new elements in the *Old* structure (critical time) we would like *New* and *Old* to contain the same elements. Thus, the construction of *New* has an atomic operation cost that is $(3n/2)/(n/2) = 3$ times faster than the atomic operation handling in *Old*. At this critical time we discard the *Old* structure and the previously labeled *New* structure becomes *Old*. When the number of elements inserted becomes $f(n)$, we initiate another reconstruction of *New* by setting the bucket size $s(f(f(n))) = s(n) + 2$ and the new critical size becomes $f(n) + f(n)/2$. In the general case, when the number of elements inserted becomes $f^{(i-1)}(n)$ (the composition of the function f $i-1$ times) we make the bucket size $s(f^{(i)}(n)) = s(n) + i$. We choose as $f(n)$ and $s(n)$ the functions n^2 and $\log\log n$, respectively. The sequence of critical points becomes $n + n/2$, $n^2 + n^2/2$, $n^4 + n^4/2, \ldots$ and the sequence of reconstruction points becomes $n, n^2, n^4, \ldots$. In this way the current bucket size is always $O(\log\log \widetilde{n})$.

One problem arising from the above discussion is how to compute the function value $f^{(i)}(n)$. Due to the fact that the model used is the PPM and any arithmetic operation is forbidden we cannot compute $f^{(i)}(n)$ immediately (in constant time). Our solution is to use an auxiliary linked list of n nodes and two pointers. The first one (*counter*$_1$) is increased by one in each step (it means that it points to the next node) and when it reaches the last (nth) node then it comes back to the start and the other one (*counter*$_2$) is increased by one whenever *counter*$_1$ completes a full round. We give the counting process schematically in Figure 1 where it is shown how to compute the value n^2 incrementally over the time period between $\widetilde{n} = n$ and $\widetilde{n} = n^2$. The other reconstruction points $n^4, n^8, \ldots$ can be computed in the same way. In the nth turn of the pointer *counter*$_1$, the pointer *counter*$_2$ shows the nth node of the linked list and the counting process is

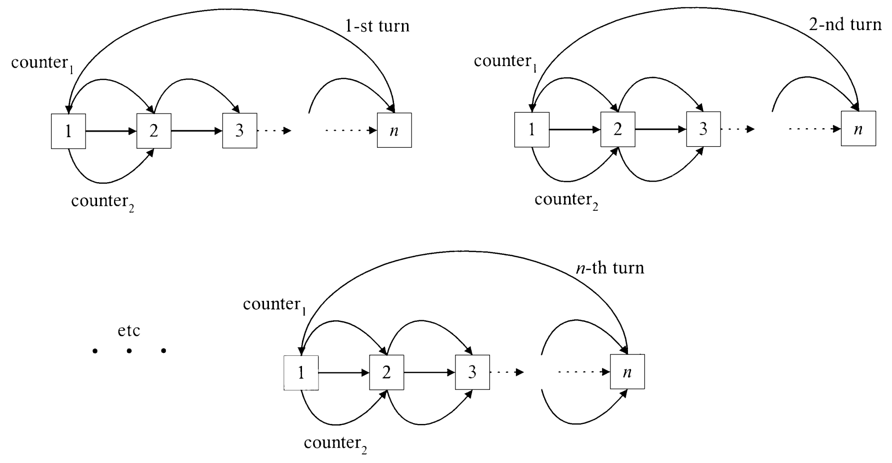

Fig. 1. Computation of n^2.

completed. Note that we could define the functions $f(n)$ and $s(n)$ such that $s(f(n)) = s(n) + c$, where $f(n) = n^{c'}$, $c' \in N$ and $s(n) = \log\log n$ (c and c' are constants and $c = \log c'$). Thus, we can postpone the initiation of the global rebuilding for a polynomially large number of insertions. Constant c' must be a natural number so that the counting mechanism can be applied (we need c' pointers in the counting mechanism).

Our result is summarized in the next theorem:

THEOREM 1. *There is a data structure for the TP problem that uses linear space, performs insertions in worst-case constant time and answers precedence queries in optimal $O(\log\log n)$ worst-case time.*

The data structure described above is complicated in the sense that it uses bucketing combined with the global rebuilding method. In addition, as in [11], the structure is recursively defined. We outline a structure that maintains the asymptotic complexities of the previous structure while at the same time is simple. The simplicity of this solution lies in the fact that the nesting levels are removed and the data structure consists only of a balanced distribution tree, where the nodes representing the elements are stored in the leaves of the structure and the internal nodes are merely subsidiary in the search process. In addition, bucketing and global rebuilding are not used in order to obtain these complexities.

The intuition behind this solution emerges by a close inspection of the recursive definition of the balanced distribution tree, that is a nesting level is identical to a subtree rooted at the root of the balanced distribution tree at the outermost level. Thus, one would wonder why not use the existing structure instead of constructing a nesting level. This observation leads us to the final solution described in this section.

In the new data structure each node has the following fields:

- a pointer to the parent node (p_{father}),
- a pointer to the right sibling (p_{r}),

- a pointer (p_{rep}) to the representative of this node in the above level (if node v is at level D we maintain a pointer at an appropriate node at level $D - 1$). This pointer simulates the nesting levels.

The algorithm for the *precedes* operation follows (assume that initially we are given pointers v_a and v_b to two nodes representing elements a and b, respectively):

(i) Check whether the father pointers point to the same address. If they do then goto step (ii), else goto step (i) after advancing the pointers v_a and v_b one level upwards by using the father pointers.
(ii) Advance pointers v_a and v_b one level upwards by using the pointers to the representative nodes and goto step (i).

Step (ii) corresponds to the traversal of the nesting levels. The time complexity of the *precedes* operation is clearly $O(\log\log n)$ because at every iteration of the algorithm we traverse at least one level of the tree upwards. Since the balanced distribution tree has height $O(\log\log n)$ the complexity follows.

Assume that the balanced distribution tree T has depth D, that is we insert elements at level D of the tree. Apart from the insertion pointer p_{ins} we maintain two traversal pointers that are needed to update the pointers to the representative nodes. The pointer p_D points to a leaf of the tree at depth D while the pointer p_{D-1} points to a node at depth $D - 1$. The algorithm for the *insert* operation is given below (in the description to follow we do not consider the case where the leaf level is full):

(i) Insert a new leaf to the right of the leaf pointed by p_{ins}. Update the p_{father} pointer of the new node and the p_{r} pointer of the leaf pointed by p_{ins}.
(ii) Update the representative pointer of the new leaf to point to the pointer p_{D-1}. Advance p_{D-1} to the right sibling (using the p_{r} pointer). If p_{D-1} is *null*, then the group of children of the node at level $D-1$ is full and thus we initialize p_{D-1} to the leftmost node at depth $D - 1$ (note that this happens only when the leaf capacity of a node at depth $D - 1$ is full).
(iii) Advance p_D to the right sibling. Insert a copy of the new leaf at depth $D + 1$ as a child of the leftmost node at node D. Set the representative node to point to the leaf pointed by pointer p_D. Update the p_{r} pointer.

In step (i) we perform the insertion. In step (ii) we set up the mechanism simulating the nesting levels of the previous structure. Step (iii) of the algorithm prepares the next level for the time when the current leaf level becomes full. An example of an insertion is depicted in Figure 2. The time complexity of the *insert* operation is worst-case constant since no iteration or recursion is performed in the algorithm above. Since the time complexity of the insert operation is $O(1)$, the space needed by the data structure will be $O(n)$. This is true because in constant time an operation may allocate at most constant space.

Our result is summarized in the next theorem:

THEOREM 2. *The TP problem can be solved by using the algorithm described above. Specifically, the operation insert*(a) *is executed in worst-case constant time while the operation precedes*(a, b) *is executed in* $O(\log\log n)$ *time. The data structure uses linear space.*

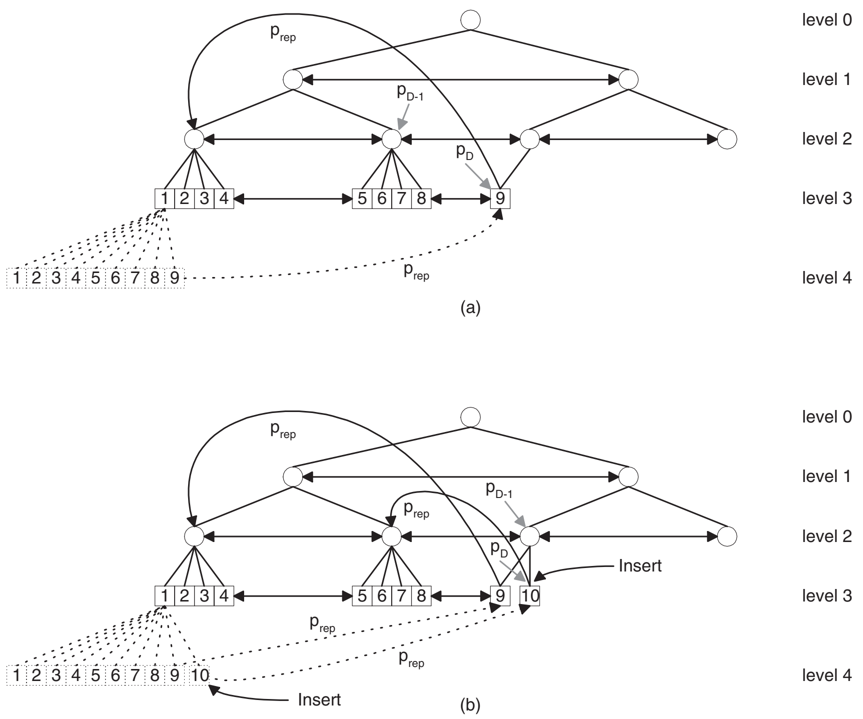

Fig. 2. A simple instance of the data structure described in Section 3 and the *insert* operation. The dotted part of the figure is associated to the part of the data structure that is going to be used when level 3 becomes full. For simplicity many pointers are not depicted. (a) An instance of the data structure before an insertion. Only the representative pointers of the last element are shown. Note the position of the pointers p_D and p_{D-1}, where $D = 3$. (b) An element is inserted. Pointers p_D and p_{D-1} are moved and the representative pointers are updated appropriately.

4. The Finger Search Method for the TP Problem. In this section we show how a slight modification of the structure presented in [11] permits the implementation of the operation *precedes*(a, b) in $O(\log\log d)$ time, where d is the temporal distance between the elements a and b. The distance expresses the time difference between the insertions of the queried elements. This time complexity for the *precedes* operation does not violate the lower bound given in [11] since in the worst case $d = n$. However, it is a substantial improvement since it is guaranteed that when the elements queried have $f(n)$ distance, then the time complexity will be $O(\log\log f(n))$. For example, if $f(n) = O(1)$, then the time complexity of the *precedes* operation is $O(1)$. Note that the structure of Section 3 or the structure described in [11] does not guarantee such a complexity. The proposed structure uses $O(n \log\log n)$ space and the insertion time is $O(\log\log n)$.

The main properties of the specific approach are the following:

1. The basic data structure used is the *balanced distribution tree*.
2. The elements are stored, as nodes, in the *leaves of the tree* and the internal nodes are merely subsidiary in the search process.
3. The tree is *level linked*.

4. For every group of nodes with a common father, we apply the *data structure recursively*.
5. Adjacent trees in every nesting level of the data structure are connected with *pointers located at their roots*.

The balanced distribution tree is chosen because its height is $O(\log\log n)$. Property 2 allows us to apply techniques known in the finger search problem. Properties 3 and 5 implement a well-known mechanism [4] that introduces distance in the time complexity of a given search problem. Property 4 remedies the large degree of the internal nodes of the balanced distribution tree that can be up to $\sqrt{n}$ for the leaf level of the tree. In order not to waste space we recursively apply the same structure only for the leaves of the trees. For the internal nodes we use the existing recursive structure of the leaves. This is possible due to the incremental construction of the nesting levels. See Figure 3 for an example of such a structure. Next we describe, in detail, the implementation of the operations *precedes* and *insert*.

The operation *precedes*(a, b) is a two-step procedure (initially we are given pointers to two nodes, corresponding to elements a and b, at the innermost nesting level of the leaves):

(i) Check if the two given nodes belong to the same balanced distribution tree in the current nesting level. This check is easy to perform since each node maintains a

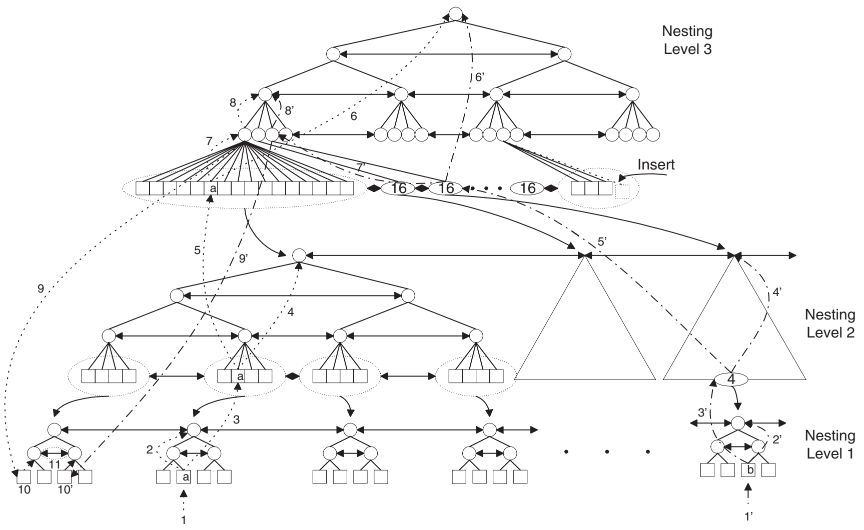

Fig. 3. An instance of the data structure described in Section 4. Insertions take place always at the end and all the recursion levels are appropriately updated (it is not depicted in this example). A paradigm of a *precedes* operation is depicted. For simplicity some pointers are omitted. Numbers give the order of each move. First we move from the node representing element a and then from the node representing b alternately in each move: $(1, 1')$ pointers to handles of a and b are given, $(2, 2')$ check if the nodes belong to the same tree, $(3, 3')$ go to the upper recursion level, $(4, 4')$ check if they belong to the same tree, $(5, 5')$ go to the upper recursion level, $(6, 6')$ check if they belong to the same tree, $(7, 7')$ go to the father of each leaf, $(8, 8')$ their fathers coincide, $(9, 9')$ go to the appropriate recursion level, at the appropriate leaves, $(10, 10')$ they belong to the same tree, $(11, 11')$ go up to their fathers, $(12, 12')$ the father of the search path of a is left of that of b and thus a precedes b.

pointer to the root of the tree that it belongs to. If they do, then go up the tree alternately until the children of the root are reached, where the answer to the *precedes* operation is immediate or until two nodes belonging to the same group of nodes with a common father are reached. In the latter case follow step (ii).

If the two given nodes belong to adjacent trees, then the answer is immediate, otherwise go up one nesting level and repeat step (i).

(ii) This step is exactly the same as the algorithm given in [11].

In the first step we traverse the nesting levels from the innermost to the outermost level, while at the second step we reverse the traversal direction. The above procedure is depicted in Figure 3.

LEMMA 1. *The above implementation of the operation precedes*(a, b) *has time complexity* $O(\log\log d)$, *where* d *is the temporal distance of the elements* a *and* b.

PROOF. The algorithm begins from the innermost nesting level and goes up until reaching a nesting level where the trees to which the elements belong are either adjacent, and thus we answer whether a precedes b immediately, or coincide. Constant time is spent in every nesting level accessed in this step.

First, we must observe that a balanced distribution tree may not be full and as a result its leaf level is not complete. This does not impose any problems in the time complexity of the operation since in the worst case we may have a balanced distribution tree for N elements but the actual number of elements is at least $n = O(\sqrt{N})$. This aggravates the time complexity by a small constant.

We must prove that the number of nesting levels visited is $O(\log\log d)$, since only constant time is spent in every such level. Assume that we stop the ascension of the nesting levels at level k. We will show that $k = O(\log\log d)$. In the kth nesting level the balanced distribution tree will have a leaf capacity of 2^{2^k} (proved by using a simple induction argument), while at most $\sqrt{2^{2^k}} = 2^{2^{k-1}}$ leaves will share a common father. The distance between a and b must be at least $2^{2^{k-1}}$ because otherwise we would have stopped in a lower nesting level. Indeed, if $d < 2^{2^{k-1}}$, then in the $(k-1)$th nesting level the two query elements either will be in the same tree or they will be in adjacent trees, because of the fact that in the $(k-1)$th nesting level the balanced distribution tree will have a leaf capacity of $2^{2^{k-1}}$. Therefore, $2^{2^{k-1}} \leq d \leq 2^{2^k}$. Taking the minimum distance, we derive $d = 2^{2^{k-1}} \Rightarrow k = \log\log d + 1 \Rightarrow k = O(\log\log d)$. Thus, we proved that the number of nesting levels accessed during step (i) is of order $O(\log\log d)$.

Assume now that we have found a balanced distribution tree at a specified nesting level that contains both elements. As we have mentioned above, all trees are level-linked and what really happens is that we move step-by-step towards the root alternately for elements a and b. We now try to estimate the time complexity of this part of the algorithm.

Assume that the node (for two nodes we use the level-link pointers and things are the same) has depth x in this specific tree. The situation is depicted in Figure 4. Assume that for element a we reached t through y and that for element b we reached t through w. In addition, assume that y and w are not brothers—if they were we would answer the *precedes* operation immediately by using the level pointers. All leaves that belong to the

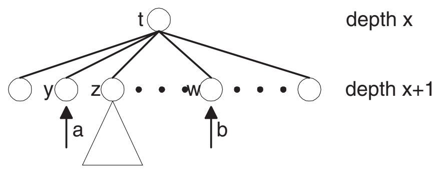

Fig. 4. The leaves of the subtree rooted at z determine the minimum number of leaves between nodes y and w.

subtree rooted at z lie between elements a and b. We denote the number of leaves of the subtree rooted at z by $|T_z|$.

Let H be the height of the tree and assume that the height and the depth of the tree are defined such that $H = D$. If h is the height of node t, then $x = H - h$, while a node of depth s has $2^{2^{s-1}}$ children. From the above we have

$$|T_z| = 2^{\sum_{i=x}^{H-1} 2^i} = 2^{2^H - 1 - (2^x - 1)} \quad \Rightarrow \quad |T_z| = 2^{2^H - 2^x}.$$

It is obvious that the distance between elements a and b will be at least $|T_z|$. Then

$$\begin{aligned} d \geq 2^{2^H - 2^x} \quad &\Rightarrow \quad \log d \geq 2^H - 2^x \\ &\overset{x=H-h}{\Rightarrow} \quad \log d \geq 2^H - 2^{H-h} \\ &\Rightarrow \quad 2^{H-h} \geq 2^H - \log d \\ &\Rightarrow \quad H - h \geq \log(2^H - \log d) \\ &\Rightarrow \quad h \leq H - \log(2^H - \log d). \end{aligned}$$

We distinguish two cases:

1. If $2^H \geq 2 \log d$, then
$$\log(2^H - \log d) \geq \log\left(\frac{2^H}{\log d}\right) - 1 = \log 2^H - \log\log d - 1,$$
$$(1) \quad \Rightarrow \quad h \leq H - \log 2^H + \log\log d + 1 \quad \Rightarrow \quad h \leq \log\log d + 1.$$
2. If $\log d > 2^{H-1} \Rightarrow d > 2^{2^{H-1}}$, then the distance between elements a and b is at least the square root of the total number of elements stored in the specific tree. Thus, $d = O(2^{2^H})$ and hence $h = O(\log\log d)$.

In this way we proved that the height reached while traversing a tree is no more than $O(\log\log d)$. Therefore, the total time needed for this second phase of step (i) of the algorithm is $O(\log\log d)$.

As we mentioned above, step (ii) is exactly the same algorithm introduced in [11]. This algorithm executes operation *precedes*(a, b) in $O(\log\log n)$ time. It is easy to see that when the algorithm arrives at step (ii) the number of elements we handle is of size $O(d^{\varepsilon})$, where ε is a constant such that $1 \leq \varepsilon \leq 2$. Thus, by using these algorithms the time complexity of this step will be

$$O(\log\log(d^{\varepsilon})) = O(\log(\varepsilon \log d)) = O(\log\log d).$$

Thus, we proved that the time complexity of the specific implementation of the operation *precedes*(a, b) is $O(\log\log d)$. □

At this point we describe the operation *insert*(a). The insertion must be performed in the $O(\log \log n)$ nesting levels of the structure. In addition, insertions are restricted to take place only at the end of the list due to the assumption that temporal relations are completely determined by the order of insertions. We will see by the discussion below that this operation is performed in $O(\log \log n)$ time.

First, note that insertions take place only at the leaves of the balanced distribution trees maintained in each nesting level. We always maintain a pointer to the last element inserted in each tree and thus we can find the insertion point in constant time. The pointer connecting a node to the same node in the immediately lower nesting level helps us to traverse the nesting levels in constant time. Thus, the insertion operation is executed in at most $O(\log \log n)$ steps.

The problem arising at this point is that of the finite capacity of the leaf level (assume the depth of the tree is equal to D) in each balanced distribution tree. Although the use of global rebuilding easily solves this problem we describe below a more efficient way to do this. That is, as we insert elements we may reach at some time the maximum capacity of the leaf level at depth D. At this moment we must construct a new tree with depth $D + 1$. This is done incrementally. Whenever we insert an element at the tree with depth D we also insert it as a child of the leftmost leaf of this tree (at depth $D + 1$). See Figure 5 for an example of this process. Because the number of children of a node in such a tree at depth D is equal to the number of nodes at depth $D - 1$ no additional counting mechanisms are needed. As a result, by the time level D of the tree is full, level

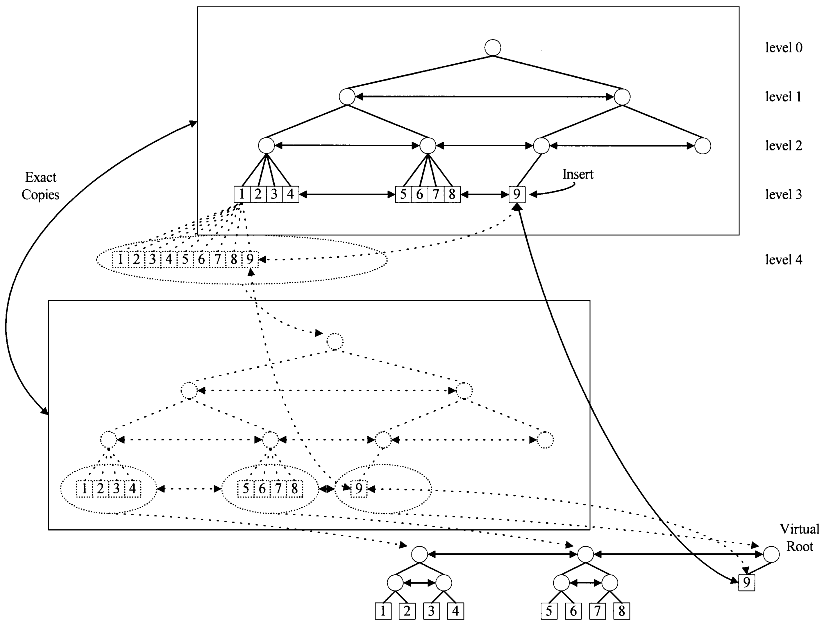

Fig. 5. A simple instance of the data structure described in Section 4 and the *insert* operation. The dotted part of the figure is associated to the part of the data structure that is going to be used when there is no more space to insert elements at level 3 and as a result a transition to level 4 takes place for the outermost nesting level. For simplicity many pointers are not depicted. The dotted pointers are not used until a transition takes place. The structures enclosed by the two rectangles are exactly the same.

$D + 1$ will also be ready to accept new insertions (in the example of Figure 5 we refer to the nesting level 3 with $D = 3$). The merits of such an incremental implementation are twofold. First, we do not have to build the whole structure from scratch but we use the existing structure. Second, during step (ii) of the *precedes* operation we need to have the internal nodes structured recursively. This is accomplished straightforwardly since the pointer structure of the nodes is maintained during the transition of a D-level tree to a $(D + 1)$-level tree that takes place when the leaf level is full.

After the transition, a gap will be formed in the nesting levels. That is, before the transition we had the tree with depth D (at the outermost nesting level) and the nesting levels consisting of balanced distribution trees of depth at most $D-1$. After the transition, the tree at the outermost nesting level has depth $D+1$ and there is a missing nesting level corresponding to the recursive structuring of its leaves (the missing level in Figure 5 is the dotted part of the figure enclosed in a rectangle). We construct this tree incrementally so as to be ready when the transition takes place. This procedure is in fact an exact copy of the tree before the transition as shown in Figure 5. During the incremental construction of this tree we update the pointers of the nodes of depth $D+1$ to point to the corresponding nodes of the new tree.

A final detail of the insertion algorithm must be clarified. In the proof of Lemma 1 we implicitly made the assumption that nesting levels and balanced distribution trees are full. However, this is not the case. The rightmost nesting levels may not be full. This imposes a problem since the proof technique of Lemma 1 is based on this assumption. To remove this assumption we virtually construct all nesting levels. That is, when a new element is inserted and the nesting levels where it should be inserted do not exist, we construct the appropriate number of nesting levels (a pointer to the previous full nesting level allows us to count the depth of nesting levels) by creating a virtual root that is in fact the root of the balanced distribution tree at the appropriate nesting level, when this tree is full. We then connect the virtual root to the sibling root of a tree at the same nesting level. We attach to this root the node inserted. Thus, we have all nesting levels and Lemma 1 applies. During each insertion at each level constant-time work is done on the respective tree. In this way we can remove the implicit assumption made in the proof technique of Lemma 1. When the *precedes* operation involves two elements lying in a balanced distribution tree, under construction, then no problem is incurred due to the sequential nature of insertions and the appropriate incremental building of the respective tree.

The space needed by the data structure presented is equal to $O(n \log\log n)$ because each element must be inserted into $O(\log\log n)$ recursion levels.

Finally, each node has the following fields:

- a pointer to the parent node,
- pointers to the left and right siblings (siblings may be adjacent nodes or roots of adjacent trees in the same nesting level),
- a pointer to the root of the tree where the node belongs,
- two pointers to the representatives of this node in the adjacent nesting levels.

THEOREM 3. *The TP problem can be solved by using the algorithm described above. Specifically, the operation insert*(a) *is executed in* $O(\log\log n)$ *time while the operation* *precedes*(a, b) *is executed in* $O(\log\log d)$ *time*, *where d is the temporal distance between elements a and b*. *The data structure uses* $O(n \log\log n)$ *space*.

5. The Final Solution. In this section we describe a data structure that can handle the *precedes* operation in $O(\log\log d)$ time while simultaneously achieving constant insertion time and linear space.

Firstly, a brief description of the data structure is given. We use a two-level data structure where the first level consists of an exponential tree similar to that described in [1]. The root of the exponential tree has degree equal to $O(\sqrt{n})$, where n is the total number of elements currently stored in the tree, while the subtrees rooted at the children of the root are exponential trees too. Due to the constrained form of insertions the tree is easily updated even in the PPM model of computation. The internal nodes of this tree are structured with one of the data structures described in Section 3. All elements are stored in the leaves of the exponential tree and the internal nodes are used only as subsidiary nodes in the execution of the *precedes* operation. We also use level linking between adjacent internal nodes:

The *precedes* operation is executed as follows. First we are given pointers to two leaf nodes corresponding to the elements we would like to check for their precedence relation:

(i) Go up the tree alternately for each node until either the nodes currently traversed are adjacent or until two nodes belonging to the same group of nodes with a common father are reached. In the latter case follow step (ii).
(ii) This step is exactly the same as one of the algorithms given in Section 3.

If we reach two adjacent nodes, then we can derive their precedence relation in constant time by using the level pointers. However, if we reach a single node, then no information can be extracted about the precedence relationship of the given elements. To resolve this conflict we structure the children of internal nodes with one of the data structures presented in Section 3. Thus, for a node with x children the data structure of Section 3 needs $O(x)$ space, $O(1)$ time for *insert* and $O(\log\log x)$ time for *precedes* queries. This linearity of space with respect to the degree of internal nodes implies that the total space used by the data structure is linear to the number of elements. Indeed, this remark follows from the fact that the number of internal nodes, in an exponential tree, is smaller than the number of its leaves.

We need now to investigate the time complexity of the *precedes* operation. It is obvious that when we traverse the exponential tree we need to reach a node or a pair of adjacent nodes at height at most $O(\log\log d)$, where d is the distance between the two input elements. In the latter case where we reach a pair of adjacent nodes we answer immediately about the temporal relation between the two input elements. However, in the former case we need to use the secondary structures. Assume that we have reached a node v at height h in the exponential tree. The degree of the node will be equal to $n^{1/2^{H-h+1}}$ (constants are not considered). We would like to give a lower bound on the distance of the two elements based on the degree of the internal nodes. The argument is similar to that used in the proof of Lemma 1. When we reach this node v from both leaves we are sure that the children of v, v' and v'', through which the search paths of both query elements passed, are non-adjacent (since otherwise we would not have reached node v). Thus, we are sure that between v' and v'' there exists at least one node z. As a result, the minimum distance between the two input elements is equal to the number of leaves of the subtree rooted at node z.

Because of the fact that z is at height $h-1$ the number of leaves at its subtree will be equal to

$$\prod_{i=1}^{h-1} n^{1/2^{H-i+1}} = n^{\sum_{i=1}^{h-1}(1/2^{H-i+1})} = n^{1/2^{H-h+1}} - 1/2^{H+1} = O(n^{1/2^{H-h+1}}).$$

Since the distance between the query elements is at least equal to the number of leaves in the subtree rooted at z we conclude that $d = \Omega(n^{1/2^{H-h+1}})$. Taking into account that exponential trees have the property that $H = O(\log\log n)$ (see [1]) we conclude that $h = O(\log\log d)$. Thus, we proved that the height of the node(s) reached during phase (i) of the algorithm is $O(\log\log d)$ where d is the distance between the query elements. When the algorithm goes into step (ii) then from the above discussion the node v reached has degree $O(n^{1/2^{H-h+1}})$. However, we also know from the discussion above that $d = \Omega(n^{1/2^{H-h+1}})$ and thus the degree of node v is asymptotically equal to the distance between the two query elements. The worst-case asymptotic complexity of the *precedes* operation follows immediately from the time and space complexities of the structure described in Section 3.

The main problem with this approach is the implementation of the *insert* operation. We must not forget that the PPM has no arithmetic capabilities and as a result we are compelled to simulate counting by using pointers. Due to the restrictive pattern of insertions we are able to control the size of the internal nodes of the exponential tree. We just have to observe that the degree of a node v at height h is the square of the degree of a node $child(v)$ at height $h-1$. Consequently, by using a scheme of two pointers (see Figure 1) we are able to keep the size constraints of the internal nodes. Assume for example that we have an exponential tree T of height H whose leaf level is full. The next insertion will result in the tree T' of height $H+1$. This implies that $O(H)$ nodes will be created as a result of a single insertion. Each such node is represented by one of the structures of Section 3. Since the insert operation of these structures is worst-case constant, the worst-case time complexity for the data structure described in this section is $O(\log\log n)$. However, the amortized cost is easily shown to be $O(1)$ by the properties of the exponential trees. We can easily make this worst-case by using a stack of deferred operations like the one described in [5]. Finally, we need to count the quantity $H+1$ in order to determine the number of nodes that must be created. We can easily accomplish this by using a pointer that traverses the path from the rightmost leaf to the root.

THEOREM 4. *The data structure described above can solve the TP problem. Specifically, the operation insert*(a) *is executed in* $O(1)$ *time while operation precedes*(a, b) *is executed in* $O(\log\log d)$ *time, where d is the temporal distance between elements a and b. The data structure uses linear space.*

6. Conclusion. In this paper we improved the complexity bounds of the TP problem in the two not optimal remaining axes: insertion time and space. The update operation became constant in the worst case and the space linear. Thus, now one can assert that in the PPM model this fundamental problem became optimal in all axes, such as in space in *precedes* and *insert* operations and that the only operation that has weaker implementation (in comparison with a pointer machine model with arithmetic capabilities) is the *precedes* operation. We also presented two different approaches to this problem that introduced in the time complexity the *temporal distance* between the two query elements.

References

[1] Andersson, A. Faster deterministic sorting and searching in linear space. In *Proc*. 37*th Annual IEEE Symposium on Foundations of Computer Science*, pp. 135–141, 1996.

[2] Brodal, G. S. Finger search trees with constant insertion time. In *Proc*. 9*th Annual ACM–SIAM Symposium on Discrete Algorithms*, pp. 540–549, 1998.

[3] Dietz, P., and Sleator, D. Two Algorithms for maintaining order in a list. In *Proc*. 19*th Annual ACM Symposium on Theory of Computing*, pp. 365–372, 1987.

[4] Huddleston, S., and Mehlhorn, K. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982.

[5] Kosaraju, S. R. An optimal RAM implementation of catenable min double-ended queues. In *Proc*. 5*th Annual ACM–SIAM Symposium on Discrete Algorithms*, pp. 195–203, 1994.

[6] Overmars, M. An average time update scheme for balanced binary search trees. *Bulletin of the EATCS*, 18:27–29, 1982.

[7] Overmars, M., and van Leeuwen, J. Worst case optimal insertion and deletion methods for decomposable searching problems. *Information Processing Letters*, 12:168–173, 1981.

[8] Pontelli, E., Ranjan, D., and Gupta, G. On the complexity of parallel implementation of logic programs. In *Proc. International Conference on Foundations of Software Technology and Theoretical Computer Science*, pp. 123–137, 1997.

[9] Raman, R. Eliminating Amortization: On Data Structures with Guaranteed Response Time. Ph.D. Thesis, Technical Report TR-439, Computer Science Department, University of Rochester, New York, 1992.

[10] Ranjan, D., Pontelli, E., and Gupta, G. Efficient algorithms for the Temporal Precedence problem. *Information Processing Letters*, 68:71–78, 1998.

[11] Ranjan, D., Pontelli, E., Gupta. G., and Longpre, L. The Temporal Precedence problem. *Algorithmica*, 28(3):288–306, 2000.

[12] Tarjan, R. A class of algorithms which require nonlinear time to maintain disjoint sets. *Journal of Computer and System Sciences*, 18(2):110–127,1979.

[13] Tsakalidis, A. Maintaining order in a generalized linked list. *ACTA Informatica*, 21:101–184, 1984.

[14] van Emde Boas, P. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6:80–82, 1977.

[15] van Emde Boas, P., Kaas, R., and Zijlstra, E. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.