The Temporal Precedence Problem

D. Ranjan, E. Pontelli, G. Gupta
Dept. of Computer Science
New Mexico State University
{dranjan,epontell,gupta}@cs.nmsu.edu

L. Longpre
Dept. of Computer Science
University of Texas at El Paso
longpre@cs.utep.edu

Abstract

In this paper we analyze the complexity of the *Temporal Precedence Problem* on pointer machines. Simply stated, the problem is to efficiently support two operations: **insert** and **precedes**. The operation **insert**(a) introduces a new element a, while **precedes**(a, b) returns true iff element a was inserted before element b temporally. We provide a solution to the problem with worst-case time complexity $O(\lg \lg n)$ per operation, where n is the number of elements inserted. We also demonstrate that the problem has a lower bound of $\Omega(\lg \lg n)$ on pointer machines. Thus the proposed scheme is optimal on pointer machines.
Keywords: Algorithms, Dynamic Data Structures, Complexity.

1 Introduction

In this paper we study the complexity of what we call the *Temporal Precedence ($\mathcal{TP}$) Problem* on pointer machines. Informally, the problem is to manage the dynamic *insertion* of elements, with the ability of determining, given two elements, which one was inserted first. The problem is related to the issue of maintaining order information in a dynamic list, which is studied by Tsakalidis [16] and Dietz and Sleator [4]. The problem we consider is simpler and more fundamental, and consequently we use the pointer machine model for a finer analysis, as opposed to the RAM model used in the aforementioned works. We are not aware of any study regarding the complexity of this problem on pointer machines.

The outcome of this work is the determination of tight lower and upper bounds for the $\mathcal{TP}$ problem. We show that the problem has a lower bound complexity of $\Omega(\lg \lg n)$. We provide efficient data structures which allow a solution to the problem with a worst-case time complexity of $O(\lg \lg n)$. Thus the implementation scheme proposed is optimal. It must also be noted that the existing schemes proposed in the literature to solve this problem (on RAM) do not translate into optimal solutions for the ($\mathcal{TP}$) Problem on pointer machines.

The $\mathcal{TP}$ problem is related to a very concrete problem that arises in parallel implementation of logic programming languages [12]. In such parallel implementation one is faced with the following problem: given a dynamic tree, capable of growing and shrinking at the leaves, and given two nodes, detect whether the first is the leftmost leaf in the subtree rooted at the second node. This operation is fundamental to correctly order the execution of order-sensitive operations (e.g., side-effects). We have observed that this problem is equivalent to the (simpler) problem of Temporal Precedence [12].

1.1 Pointer Machines

We will consider the *Pure Pointer Machines (PPM)* as model of computation [7, 15, 3, 8, 14]. An overview of the concept of Pointer Machine and references to relevant works is presented in [1].

Even though RAM is the most commonly used reference model in studies of complexity of sequential algorithms, the pointer machine model has received increasing attention as a viable model to study complexity of algorithms. The pointer machine model is particularly advantageous for complexity studies involving finer level of details [15]. The Pointer Machine model is simpler, thus making it more suitable for analysis of lower bounds of time complexity. Furthermore, RAM commonly hides the actual cost of arithmetic operations, by allowing operations on numbers of size up to $\lg n$ (n being the size of the input to the problem) to be treated as constant time operations. PPM instead makes these (relevant) costs explicit. Indeed, many lower bound results have been developed using the PPM model [15, 2, 13].

A *Pure Pointer Machine (PPM)* [7, 15] consists of a finite but expandable collection R of *records*, called *global memory*, and a finite collection of *registers*. Each record is uniquely identified through an address (let $\mathcal{N}$ be the set of all the addresses). A special address *nil* is used to denote an invalid address. Each record is a finite collection of c fields; each field can contain only an address. Given a record n belonging to R, we indicate with n^i the ith field of the record. From what we said above, the value stored in a field n^i will always be an element from $\mathcal{N}$. The pointer machine is also supplied with a finite collection of registers $r_1, r_2, \ldots, r_k$ (pointer registers). Each register r_i can contain one element of $\mathcal{N}$. We assume that the register r_k is a read-only register (i.e., its content is never modified) which contains the value *nil*. The machine can execute *programs*. A program P is a finite, numbered sequence of instructions (where each instruction in P is uniquely identified by one number). Given a program P, $|P|$ denotes the number of instructions present in it. The instruction set of the PPM consists of two types of instructions – pointers manipulation and control management. The instructions set of a pure PPM is described below.

$r_i := r_j$ where $1 \leq i, j \leq k$; this instruction allows to copy the content of register r_j into register r_i.

$r_i := r_j^a$ where $1 \leq i, j \leq k$; the instruction copies the a^{th} field of the record pointed by r_j into register r_i.

$r_i^a := r_j$ where $1 \leq i, j \leq k$; the instruction copies the register r_j into the a^{th} field of the record pointed by r_i.

$r_i :=$ new where $1 \leq i \leq k$; the instruction assigns to register r_i the address of a previously unused record;

halt terminates the execution;

if $r_i = r_j$ goto i where $1 \leq i, j \leq k$, and $1 \leq i \leq |P|$; transfer the control of execution to the instruction labeled i if the registers r_i and r_j are pointing to the same record.

Given a program P, execution starts with the instruction labeled 1 and continues until either the halt instruction is encountered or the execution reaches the program position numbered $|P|+1$. Pointer machines were first designed by Kolmogorov and Uspenskii [8], and later rediscovered by Knuth [7] (the *linking automaton*). Schönhage [14] provided the first study in understanding the connection between this model of computation (that he calls *Storage Modification Machine*) and the more traditional models based on Turing machines and RAM. Extended versions of the pointer machine, capable of performing elementary arithmetic, have been proposed by Chazelle [3] and Ranjan et al. [12].

1.2 Temporal Precedence Problem

In this section we formalize the specific problem that we intend to study. Formally, the Temporal Precedence problem can be defined as follows: given a collection of distinct elements $a_1, a_2, \ldots, a_n$, we want to carry out a sequence of operations of the following two types:

insert(a_i)**:** which marks the element a_i as inserted

precedes(a_i, a_j)**:** checks if a_i was inserted before a_j

The operations are to be carried out *on-line*, that is each instruction has to be completed before the next one is considered. For the sake of simplicity, we assume that each sequence of operations is correct (e.g., each element is inserted at most once) and contains exactly n **insert** operations—thus, all the elements in the sequence are inserted at some point of time. The **precedes** operations can be arbitrarily intermixed with occurrences of the **insert** operation. We assume that the **precedes** operations are always *correct*, i.e., a **precedes**(a_i, a_j) is executed only after both a_i and a_j have been inserted in the sequence. The problem is very basic, as it captures the fundamental concept of *ordering* elements, and is closely related to the problem of *counting*. Note that in the $\mathcal{TP}$ problem we cannot assume anything about the structure of the input data a_i and for all practical purposes the data are abstract entities. In the algorithms we propose, data are assumed to be available in a reasonable format. We shall explain later how we can present the data to the algorithms and how we can analyze the complexity of the algorithms in a meaningful way.

It is easily observed that the $\mathcal{TP}$ problem can be solved in $O(1)$ time on the RAM that allows constant-time comparison for numbers up to size $\lg n$. The essential idea is to "time-stamp" the elements during insertion. The **precedes** operation then is just the comparison of the time-stamps of the two elements. Since the size of the time-stamps needed, in case n elements are inserted, is bounded by $\lg n$, this is an $O(1)$ scheme to solve the $\mathcal{TP}$ problem.

2 Related Work

To the best of our knowledge, the $\mathcal{TP}$ problem presented in this work has not been studied in the context of pointer machines.

A somewhat more general form of this problem has been considered in the context of lists manipulation. The list manipulation problem allows insertion in arbitrary points of the sequence, as well as deletions. Two relevant works are those by Dietz and Sleator [4] and by Tsakalidis [16]. Tsakalidis' solution allows to perform a sequence of n insertion and deletions in time $O(n)$ and precedence tests in time $O(1)$. Dietz and Sleator propose another solution which allows all the three operations to be realized in time $O(1)$. Thus it seems that the $\mathcal{TP}$ problem can be solved in $O(1)$ time. Unfortunately, both schemes use a RAM model where integers of size up to $\lg n$ (n being the size of the list) can be compared in constant time. Thus, the $O(1)$ solutions provided by [4, 16] do not directly translate to an optimal solution to the $\mathcal{TP}$ problem on pointer machines.

In absence of the RAM model the schemes mentioned in [4, 16] will incur a cost of $\Omega(\lg n)$ per operation. In our opinion, the use of the pointer machine model is more suitable to the study of the $\mathcal{TP}$ problem, which we believe is fundamental. The use of PPM model allows us to study the problem at a finer level of detail. Both the upper and lower bounds provided in this paper use new techniques that are different from [4, 16].

Additionally, Dietz and Sleator's schemes assume an *a priori* knowledge of the number of elements to be inserted. Our solution does not require this information and hence is truly dynamic in nature.

The problem is also related to the least common ancestor problem and some connections are apparent in the paper [17, 6].

3 Upper-bound of $\mathcal{TP}$ on PPMs

The purpose of this section is design a data structure which can allow solving the $\mathcal{TP}$ problem efficiently on PPMs. The $\mathcal{TP}$ problem can be solved in time $O(1)$ on generic RAM models. The solution described essentially uses the fact that a *counter* can be implemented very efficiently on the RAMs, and that integer comparisons can be performed very efficiently ($O(1)$ per operation for $\lg n$-bits numbers). Thus, in RAM models initializing locations to zero, incrementing their content, and performing comparisons (i.e., $\leq$) between them are all constant-time operations. Removing any of these abilities from the model leads to a *strictly* less efficient model. For example, the RAM_1 and RAM_0 models presented in [14] are in this class, as they are not allowed constant-time general comparisons. Our conjecture is that the $\mathcal{TP}$ problem ceases to be solvable in $O(1)$ time in any of these less efficient models. There seems to be an inherent connection between $\mathcal{TP}$ and the implementation of a basic counter.

Our focus here is to study the problem in the context of pointer machines. The results transfer immediately to equivalent models (like RAM_1 and RAM_0). In this section we present upper bounds for the $\mathcal{TP}$ problem on pointer machines. We are not aware of previous results for this particular problem. To provide an intuition as to how we achieved our results, we present several successively better schemes, finally ending with the best result.

The data structure we design and use to obtain the optimal upper bound of $O(\lg \lg n)$ has some similarities with van Emde Boas' *stratified-tree structure* designed to efficiently implement priority queues [18] and used to provide efficient solutions to various set manipulation problems (e.g., [18, 9, 10]). Both these data structures use the idea of a nested tree structure, where the same logical organization is repeated at different levels of nesting. In our case the levels of nesting are represented by different "colored" trees, while in the stratified-tree structured the levels are represented by subtrees with different *ranks* [18]. The stratified-tree structure yields a worst case processing time of $O(\lg \lg n)$ per instruction.

Despite these similarities, it is not clear if and how the stratified-tree structure can be used to achieve an $O(\lg\lg n)$ solution for the $\mathcal{TP}$ problem. One of the major difficulties encountered is that the $O(\lg\lg n)$ time complexity achieved for supporting various set manipulation operations using the stratified-tree structure is based on the use of a RAM model with uniform time measure. As the authors themselves point out [18], if a logarithmic time measure is used their scheme yields an $O(\lg n)$ solution. Consequently their scheme will translate to an $O(\lg n)$ solution on a pointer machine.

Additional problem is posed by the fact that the stratified-tree structure requires an *a priori* knowledge of the number of elements to be inserted. In contrast, our scheme is totally dynamic, as required by the $\mathcal{TP}$ problem. Also the stratified-tree structure requires an $O(n \lg\lg n)$ time pre-processing phase to build the structure, which is not required in our scheme.

Finally, our scheme being a specific solution to the $\mathcal{TP}$ problem, is considerably simpler and easier to understand and implement.

3.1 Some Simple Schemes

A naive approach to the problem easily leads to a complexity of $O(1)$ for the **insert** operation and $O(n)$ time for the **precedes** operation. Essentially, one can maintain the inserted elements as a simple linked list. Two registers are used to maintain pointers to the head and the tail of the list. The **insert** operation requires simply the creation of a new record in the list, which is appended to the tail—clearly a constant time operation. The **precedes** operation, on the other hand, requires traversing the list in order to detect the relative position of the two elements. This has a worst–case time complexity $O(n)$.

An alternative solution can be obtained by using *time-stamps*. As in the RAM implementation, each element can be labeled with the time of insertion. The **precedes** operation is then reduced to the comparison of the labels of the two elements in question. Unfortunately, pointer machines are not supplied with any form of arithmetic, they are only capable of creating records and manipulating pointers. Thus, time stamps have to be implemented using records. One possible way is to code the numbers in binary format, representing each digit using a record, and each digit's value as the value of the first field of the record (e.g., *nil* encodes 0, while any other pointer will encode 1). Using this approach a **precedes** operation requires comparison of two labels, whose length is $O(\lg n)$. On the other hand, the need to maintain a label implies a non-constant cost also during the **insert** phase. The cost of creating a label is proportional to the length of the label—as we need to either traverse or copy an arbitrarily large part of the previously created label. Thus the insertion operation has itself an individual cost of $O(\lg n)$.

3.2 A Simple Tree-Based Scheme

An improvement over the $O(\lg n)$ time cost per operation presented in the previous section can be obtained by switching from a linear list structure (explicitly represented or implicit in the growing time labels) to a tree structure.

Let us start from the simple case in which a binary tree structure is employed. We name this approach the *Binary-Tree scheme*. The elements which are inserted are placed in a binary tree which is expanded level by level from left to right. Each node has a pointer to the node immediately to its right and to its parent (see Figure 1). The insertion of elements can be done in $O(1)$ time easily 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**(u, v) can now be implemented using the following idea – **precedes**(u, v) should return true iff one of the following three conditions holds:

(*i*) u is the root and v is not;

(*ii*) v is immediately to the right of u;

(*iii*) u'=parent(u) and v'=parent(v) exist and **precedes**(u', v') is true.

It is clear that a straightforward implementation of **precedes** gives us an $O(\lg n)$ scheme because the height of the binary tree is bounded by $\lfloor \lg n \rfloor$. Figure 2 gives the pseudo-code for the **insert** operation[1]; for the sake of simplicity we are also assuming a model with at least four pointers per record (without loss of generality). The code assumes[2] that (see Figure 2)

- register r_0 stores the pointer to the element to be inserted;
- each record n uses n^1 to point to the parent (*nil* for the root), n^2 to point to *nil* iff this record is the left child of its parent, n^3 to point to the node on its immediate right, and n^4 to point to the object record.
- register r_1 points to the rightmost node in the last full layer of the tree;
- r_2 points to the leftmost node in the current layer of the tree;
- register r_3 points to the last node created;
- newly created records have all fields set to *nil*.

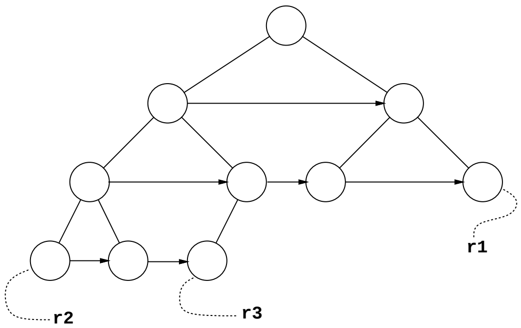

Figure 1: Binary Tree Representation

insert:
  temp := new;
  $\text{temp}^4 := r_0$;
  if ($r_3^1 = r_1$ and $r_1^2 \neq nil$) then
    $\text{temp}^1 := r_2$; $\text{temp}^2 := nil$;
    $r_1 := r_3$; $r_2 :=$ temp;
  else
    if ($r_3^2 = nil$) then
      $\text{temp}^1 := r_3^1$; $\text{temp}^2 :=$ temp;
      $r_3^3 :=$ temp;
    else
      $\text{temp}^1 := (r_3^1)^3$;
      $\text{temp}^2 := nil$; $r_3^3 :=$ temp;
    endif
  endif
  $r_3 =$ temp;

Figure 2: Pseudo-code for **insert**

As can be seen from the code in figure 2, the cost of each insertion operation is $O(1)$. The code in figure 3 describes the operation **precedes**. This operation is based on the tree representation described above and assumes that r_0 and r_1 contain the pointers to the two arguments of **precedes**.

Lemma 3.1 (Complexity of Binary-Tree) *In the Binary-Tree algorithm* **insert** *has a worst–case time com-*

[1] The corresponding pointer-machine code can be obtained in a straightforward way.
[2] r_i denotes the i^{th} register of the pointer machine, while n^j indicates the j^{th} field of record n.

```
precedes:
    found := false;
    while (not found)
        if (r_0 = r_1^3) then
            output "FALSE"
            found := true;
        elseif (r_0^3 = r_1) then
            output "TRUE"
            found := true;
        elseif (r_0 = root) then
            output "TRUE"
            found := true;
        elseif (r_1 = root) then
            output "FALSE"
            found := true;
        else
            r_0 := r_0^1; r_1 := r_1^1;
        endif
    endwhile
```

Figure 3: Operation **precedes**

plexity of $O(1)$ *while* **precedes** *has a worst–case time of* $O(\lg n)$*, where* n *is the number of elements inserted.*

3.3 Double Trees

In the scheme presented above we use a binary tree for organizing the elements that are progressively inserted. Clearly, one can think of generalizing this to schemes using d-ary tree structure for any d. The order of complexity will not change as long as d is a constant; d steps will be required if the compared nodes have the same ancestor, while $\lg_d n$ will be the maximum number of levels that can be traversed before getting to the root of the tree. The **insert** operation is still $O(1)$, whereas the **precedes** operation will take time $O(d + \lg_d n)$ (in the worst case). Intuitively, this scheme partitions the set of n inserted elements into groups of size d; these groups are themselves organized into groups of size d and so on. Identifying the ordering between two elements reduces to identifying the "smallest" class which contains both the element.

The next step in the generalization is to allow the degree of the nodes in the tree to vary. We can think of generalizing in two directions: keeping the degree at each node the same but making it a function of n or letting the degree vary in some systematic way. For the first type of generalization, an *a priori* knowledge of the number of elements to be inserted seems necessary. We discuss some schemes of this type in subsection 3.5. The following subsections will instead discuss schemes where no *a priori* knowledge of n is assumed. Note that also the schemes discussed to this point do not assume any *a priori* knowledge of n.

Let us start by considering the simplest generalization. Instead of using a binary tree, we use a tree in which the degree of the nodes at each level is twice the degree of the nodes at the previous level. Thus, if we indicate with $d(i)$ the degree of the nodes at depth i, then we have the recurrence

$$d(i+1) = 2d(i) \qquad d(0) = 2$$

Figure 4 shows the first levels of this tree.

We can easily compute the number of nodes $t(i)$ that are present in the i^{th} level of the tree: $t(i+1) = t(i)d(i)$ and $t(0) = 1$.

By solving the recurrences we have $d(i) = 2^{i+1}$ and

$$t(i+1) = t(i)d(i) = t(i)2^{i+1} = 2^{\frac{(i+1)(i+2)}{2}}$$

The increasing number of nodes present at each level requires a smarter approach in handling the **precedes**

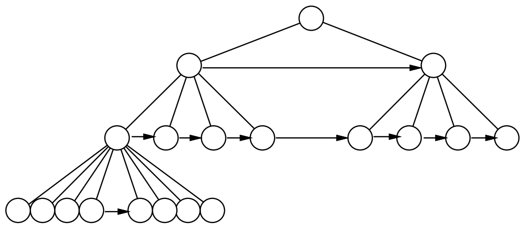

Figure 4: Tree for $d(i+1) = 2d(i)$

operation in the case where the two nodes have the same direct ancestor. In fact, if the two nodes considered are at level i and they have the same direct ancestor, then they belong to a collection of $d(i-1) = 2^i$ elements having the same parent. Thus, a linear scan of the siblings as done in the previous case (where only two siblings were present) becomes now inefficient, due to the excessive length of the sequence.

Let us start by estimating the number of levels (i.e., the depth) that this sort of tree is going to have for a sequence of n **insert** operations. From the fact that $t(i) = 2^{\frac{i(i+1)}{2}}$ it follows that the depth of the tree is no more than $\lceil\sqrt{2\lg n}\rceil$ which is $O(\sqrt{\lg n})$. One interesting fact to note is that, if $t(j)$ is the number of nodes present at level j, then

$$\sum_{i=0}^{j-1} t(i) < t(j)$$

Thus, each level contains strictly more nodes than all the previous levels together.

Now, we can derive what the worst case situation will be for the **precedes** operation. The worst–case situation will arise in scanning linearly the largest group of nodes having the same direct ancestor. Since at level l a group of 2^l nodes will share the same direct ancestor, the worst–case linear scan will require $O(2^{\lceil\sqrt{2\lg n}\rceil})$ steps. Thus the cost of **precedes** in case the two nodes being compared were from one of these large groups would be excessive. Note that in the completely dual case, where the two nodes being compared never have the same ancestor, the cost is considerably lower, as it is bounded by the depth of the tree, which is $O(\sqrt{\lg n})$.

The aim then is to reduce the cost of **precedes** in the troublesome situations. We can achieve this by repeating the same kind of tree-structuring in each group of nodes having the same ancestor, and doing this recursively. This structure may be imposed through another set of pointers (it helps to think of these as different color pointers). The innermost level of nesting will be characterized by having a tree-structure, in which no more than two nodes share the same direct ancestor[3]. Figure 5 illustrates a simple example (for the sake of clarity we have omitted from the picture the links between nodes with the same ancestor). Thus, multiple independent tree structures are imposed on the collection of nodes inserted. Each element inserted contains pointers to its representatives in each of the trees it belongs to (e.g., node 18 in Figure 5 contains a pointer to its representative in three different trees).

The next observation that we need to make regards the maximum level of nesting of trees (or the number of different colored trees needed) in this scheme. From previous considerations, the maximum number of nodes present in a group with the same ancestor is $2^{\lceil\sqrt{2\lg n}\rceil}$. Furthermore, for a m nodes tree, the depth of the tree is $\sqrt{2\lg m}$. Thus, the first time we create a nesting we are going to obtain a tree with a depth of no more than

$$(\lceil\sqrt{\lg 2^{\lceil\sqrt{2\lg n}\rceil}}\rceil) \leq 2(\lg n)^{\frac{1}{4}}$$

In fact we can show that at the kth level of nesting of trees, we are going to have trees with depth at most $2(\lg n)^{\frac{1}{2^k}}$. Since we can stop further nesting once there are no more than $O(\sqrt{\lg n})$ nodes with the same ancestor, then we can estimate the maximum depth of nesting k by solving the inequality

$$2^{2(\lg n)^{\frac{1}{2^k}}} < (\lg n)^{\frac{1}{2}}$$

[3]Actually we could stop the nesting earlier, as soon as there are at most $O(\sqrt{\lg n})$ nodes with the same ancestor.

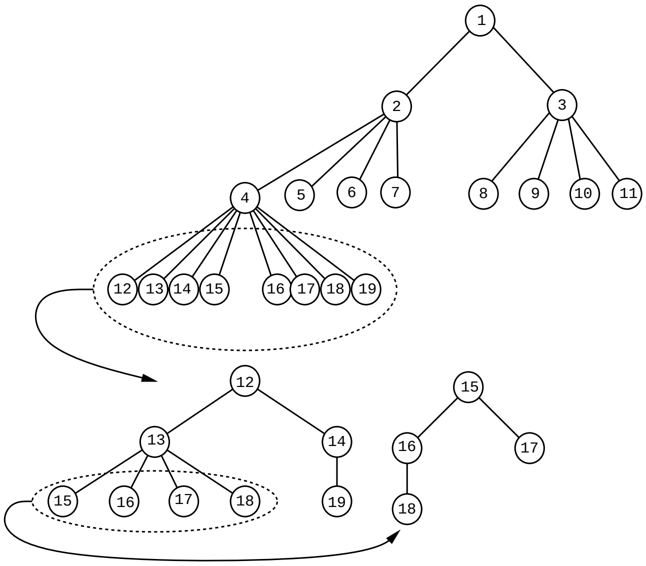

Figure 5: Nesting of Trees

which leads to the result $k = O(\lg\lg\lg n)$. Similar order of magnitude is achieved even if we impose the simpler constraint of having $2^{(\lg n)^{\frac{1}{2^k}}}$ lower than a given constant (e.g., 4) instead of $\sqrt{\lg n}$.

One important thing to note here is that we have a non-constant (k) number for nesting levels. This seems to imply that, in order to maintain the nesting information for each node, our records must have a non-constant number of fields. This is not allowed in the PPM. We can overcome this by maintaining the k fields in one single field as a linked list, and paying a penalty of $O(k)$ for each access. In the specific case of double-trees the penalty is $O(\lg\lg\lg n)$. The problem can be overcome by keeping *distinct* representation of the different trees that are generated. This is illustrated in Figure 6. Each element inserted is actually represented by a list of records, each representing the membership of the element to a distinct tree. In Figure 6, the elements at the top are the records inserted (the number identifies the order of insertion). From each record there is a pointer to a list which links the representatives of each node in the different trees. For example, the element 15 belongs to three different trees. Observe that, once we are positioned in a tree of a given color, we can directly traverse such tree, *without having to scan the list of pointers associated to the original record.* For example, in Figure 6, if we want to compare the relative positions of the nodes 17 and 18 in the green tree, we just need to detect their representatives in such tree and then move inside the green tree; this will involve visiting the green representatives of nodes 15 and 16, but this will not require scanning the list of pointers of 15 and 16, since such representatives are directly accessible from the green representatives of 17 and 18. This allows to avoid the $O(k)$ penalty mentioned above.

The implementation of the scheme is based on the following steps:

insert: whenever a node is inserted in the tree structure, it needs to be enriched with the appropriate pointers to allow identifying its position within each nested tree. The intuition is that each node n is supplied with a list of pointers, $n_1, \ldots, n_s$, where node n_i is supposed to indicate the ancestor of n with respect to the ith level of nesting (i^{th} colored pointer)—as in Figure 6. The value of s can reach, in the worst case, a value equivalent to the maximum number of nesting of tree structures, i.e., $O(\lg\lg\lg n)$. The node has to be inserted at each level of nesting (i.e., for each color a pointer may have to be set appropriately). Hence, in the worst case it may require modifying at most $O(\lg\lg\lg n)$ pointers. Insertion in each level of nesting requires modification of a constant number of pointers in a constant number of records (a simple generalization of the $O(1)$ insertion presented in the Binary-tree case)—with a worst–case time complexity $O(1)$, thanks to the previously described organization of each level of nesting (where all nodes in a colored tree are directly accessible without having to search for them starting from the original node). Also, it may seem that counting abilities are needed to detect when one group, level or tree has been completed (e.g., to determine when the $2d(i)$ value has been reached and an ancestor has been saturated). But, one should notice that we need only special types of counting abilitites. For example, to

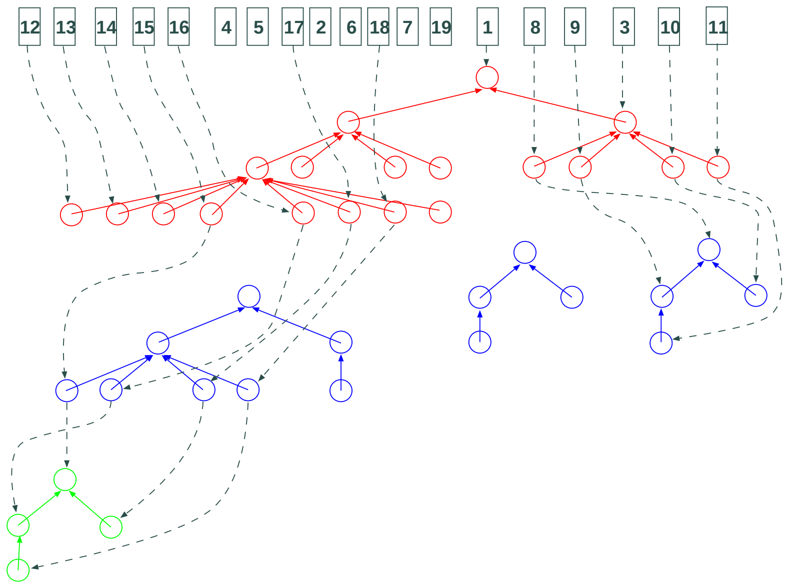

Figure 6: Organization of the elements

double the degrees at next level, the actual counting can be avoided by simply moving a pointer over the previous level of the tree (every two insertions the pointer is moved by one); whenever the pointer reaches the end of one group of nodes with same ancestor, then the $2d(i)$ quota has been reached. It is then clear that the cost of an insertion will be $O(\lg\lg\lg n)$ in the worst case ($O(\lg\lg\lg n)$ nodes to modify, with a cost of $O(1)$ for each modification).

precedes: the search procedes very similarly to the scheme described in the previous section. Given the two nodes n and m to be compared, we first check if one is immediately to the right of the other in which case the answer is obvious. Otherwise, the parents of the two nodes at the first level of nesting (i.e., their parents in the outermost tree) are compared. If they are different then **precedes** is called recursively on the parents. Otherwise, the attention is switched to the next level of nesting (the next set of colored pointers) and the same process is repeated. Observe that while applying this process, if at depth r we apply precedes to two nodes with the same parent, then the test proceeds in the "next" tree. Since the next tree contains at most 2^{r+1} nodes, then its height is at most $O(\sqrt{2(r+1)})$ and it may contain at most $O(\lg\lg(r+1))$ other colored trees—thus both these quantities are strictly smaller than r. One can easily argue that in the worst case **precedes** performs at most $O(\sqrt{\lg n} + \lg\lg\lg n) = O(\sqrt{\lg n})$ steps "walking-up" the trees at any level of nesting. Hence the complexity of **precedes** in the worst case is $O(\sqrt{\lg n})$.

Lemma 3.2 (Complexity for $d(i) = 2d(i-1)$) *The Double-tree algorithm implements* **precedes** *with a worst–case time complexity of* $O(\sqrt{\lg n})$ *and* **insert** *with a worst–case time complexity of* $O(\lg\lg\lg n)$ *per operation, where* n *is the number of elements inserted.*

3.4 Balanced Distribution Trees

In the double tree, the degree of nodes was selected to satisfy the relationship $d(i) = 2d(i-1)$. One of the problems in doing this is the fact that the tree is still too deep, and solving the case where two nodes have different ancestors may require more time than the other cases. The solution that we now introduce to improve this situation reduces the depth of the tree, by increasing the degree of the nodes.

We consider the case where the degree of nodes at level i is defined to be $d(i) = t(i)$, where, as before, $t(i)$ indicates the number of nodes present at level i. This is required to hold for $i \geq 1$, while $d(0)$ is defined to be 2 and $t(0) = 1$. $t(i)$ can be defined again as $t(i) = t(i-1)d(i-1)$. Putting together the various components, we can solve the recurrence relation and obtain ($i \geq 1$):

$$d(i) = 2^{2^{i-1}} \qquad\qquad t(i) = 2^{2^{i-1}}$$

Let us assume that n nodes are inserted in the tree. Thus we can compute the depth of the tree. Let us indicate with h the number of levels in the tree. Clearly, n should satisfy the inequalities $t(h-1) < n \leq t(h)$ By solving this we obtain $h = \Theta(\lg \lg n)$. This shows that we have actually managed to reduce the height of the tree with respect to the double-trees case.

We need now to determine what will be the maximum number of nestings of trees that can occur for n elements. Observe that the maximum number of nodes with the same direct ancestor is $d(h-1)$. Would it be possible for a second level tree to have the same (or bigger) depth than the outermost one? This would imply that

$$\sum_{j=0}^{h-1} t(j) < d(h-1)$$

as otherwise we would be able to fit all the $d(h-1)$ elements within the first $h-1$ levels. But we need to remember that $d(i) = t(i)$, thus

$$d(h-1) + \sum_{j=0}^{h-2} d(j) < d(h-1)$$

Which would imply that the number of nodes in the first $h-2$ levels is negative, clearly impossible. Thus, the second level tree will have depth strictly lower than the depth of the outermost tree. As a consequence, the maximum number of nesting of trees k that we can have is itself $O(\lg \lg n)$.

Considering the two operations, **insert** and **precedes**, we can observe that their implementations can be obtained as a straightforward modification of the algorithms used for the double-trees in the previous section. To perform an insertion, **insert** updates a number of pointers proportional to the number of nesting of trees, which is $O(\lg \lg n)$. As described in the previous subsection, we can organize the different nested (colored) trees in such a way to avoid having to pay any penalty for detecting the "correct" representative of a node in a tree. The **precedes** operation is implemented exactly as in the double-tree case. It is easy to show that it has worst–case time complexity $O(\lg \lg n)$. In fact, by moving up in a tree from depth $r+1$ to depth r we are guaranteed that level r will contain at least one colored tree less than level $r+1$. Furthermore, the depth of the next colored tree at level r will be at most r.

Lemma 3.3 (Complexity for $d(i) = t(i)$) *The Balanced Distribution tree algorithm implements both* **insert** *and* **precedes** *with a worst–case time complexity of $O(\lg \lg n)$ per operation, where n is the number of elements inserted in the structure.*

Thus the solution has improved over the double tree case, by balancing the cost of the two operations.

3.5 Does a priori knowledge of n help?

All the previous schemes mentioned do not assume any *a priori* knowledge about the number of elements, n, that will be inserted. A natural question that emerges from the previous discussion is the following: would an *a priori* knowledge of n allow a better solution? In particular, can one use a tree–based scheme where the degrees of the tree nodes are a function of n that will yield a better solution than any of the previous scheme. This is similar in spirit to the analysis of union-find algorithms where the number of elements to be inserted is known in advance [2]. In this section, we discuss and analyze a scheme that makes use of this information. For the sake of simplicity, in this section we will ignore the additional penalty incurred for manipulating records with variable number of fields, and we ignore ceilings and floors in our calculations.

The first scheme we discuss the degree of each tree node is the same but it is a function of n. The choice is made so as to minimize the worst–case time complexity of **insert** and **precedes**. Let us consider n to be known in advance and let us assume that we are using a d-ary tree to store the elements. We can use the knowledge of n to choose d optimally. Let, $d = t(n)$. The tree has a constant arity. As a consequence, we have that the depth of the tree is $\lg_{t(n)} n$.

Insertion can be done in the simplest possible way (i.e., each node is pointing to its parent in the tree, and the only additional structures required are those to compute when a node received all its d successors and when a level of the tree has been completed—all this require $O(d)$ time). The execution of a **precedes** operation is performed as in the case of simple binary trees presented in a previous section. If the two nodes have different parents, then the comparison is repeated between their parents, otherwise the list of nodes associated to the parent is scanned to detect the order of the two nodes considered. The cost of **precedes** is $\lg_d n + d$. Since $d = t(n)$, we can rewrite the expression as $\lg_{t(n)} n + t(n)$.

At this point we can estimate what is the "best" choice for $t(n)$. This will be obtained when $\lg_{t(n)} n = t(n)$. This can be transformed into the equation $\lg n = t(n) \lg t(n)$. This equation leads to

$$t(n) = \Theta(\frac{\lg n}{\lg \lg n})$$

(replacing $t(n)$ by $\frac{\lg n}{\lg \lg n}$ we obtain that $t(n) \lg t(n) \leq \lg n$, while using the value $2\frac{\lg n}{\lg \lg n}$ for $t(n)$ we obtain $t(n) \lg t(n) \geq \lg n$). Thus this leads to little improvement with respect to the *Binary-Tree* algorithm. This scheme is very similar in idea to the one given by Blum [2] for the union-find problem.

Let us now consider a more complex scheme where we use the nested tree ideas together with constant arity at each level. We will choose $d = t(n)$ for some monotonically growing function t. A second-level tree will contain $t(n)$ nodes, and it will have degree $t(t(n))$; each third-level tree will contain $t(t(n))$ nodes and will have degree $t(t(t(n)))$ and so on.

Let us focus on the cost of performing a **precedes** operation in this structure; we indicate with $c(n)$ the cost paid for a tree containing n nodes. The **precedes** operation requires

- walking up the tree from the two nodes until a common ancestor is found;
- repeating the search in the nested tree localized as the set of successors of the previously found common ancestor;

Hence, a crude overestimate gives us that $c(n) \leq 1 + \lg_{t(n)} n + c(t(n))$ where the 1 originates from the comparison required to verify if the two nodes have the same ancestor, $\lg_{t(n)} n$ is the worst case number of steps required to walk up the tree in search of a common ancestor, and $c(t(n))$ is the cost for repeating the activity in a nested tree. The problem is now to find the best t which allows to minimize $c(n)$. The recurrence relation can be rewritten as:

$$c(n) = 1 + \frac{\lg n}{\lg t(n)} + c(t(n))$$

Assuming that the recurrence will stop as soon as $t(n)$ is a constant (e.g., 2), and assuming that this happens in i steps of recursion, then

$$c(n) = i + \sum_{j=0}^{i} \frac{\lg t^i(n)}{\lg t^{i+1}(n)}$$

where $t^i(n)$ indicates i applications of t. The intuition is that t has to be selected in such a way to create the proper balance between i and the result of the summation.

A first possible choice for t is $t(n) = 2^{\sqrt{\lg n}}$. With this choice we can show that:

$$t(n) = 2^{\sqrt{\lg n}} \qquad t^i(n) = 2^{(\lg n)^{1/2^i}}$$

Consequently

$$c(n) = i + \sum_{j=1}^{i} (\lg n)^{1/2^j}$$

The second term is clearly dominated by the first element of the sum, $\sqrt{\lg n}$. The number i of recursions required to get to a constant can be bound by simply comparing it to $\sqrt{\lg n}$: and we obtain $i = O(\lg \lg n)$. Thus for this choice of t we obtain $c(n) = O(\sqrt{\lg n})$. The same reasoning applied to $t(n) = 2^{\sqrt[3]{(\lg n)^2}}$ will produce $c(n) = O(\sqrt[3]{\lg n})$. More in general, selecting

$$t(n) = 2^{(\lg n)^{\frac{k-1}{k}}}$$

it is possible to prove that the corresponding complexity is

$$O(\lg\lg\lg n + (\lg n)^{\frac{1}{k}}) = O((\lg n)^{\frac{1}{k}}).$$

In general, we need to find the best function $t(n)$ such that, if d is a value such that $t^d(n) \approx c$ (c is a constant), then

$$O(d) = O(\sum_{j=0}^{d} \frac{\lg t^j(n)}{\lg t^{j+1}(n)})$$

This originates from the general cost expression

$$f(n) = d + \sum_{j=0}^{d} \frac{\lg t^j(n)}{\lg t^{j+1}(n)}$$

This has an upper bound corresponding to

$$d(1 + \max_{0\leq j\leq d} \frac{\lg t^j(n)}{\lg t^{j+1}(n)})$$

This clearly implies that the complexity will be at least $\Omega(d)$ for this scheme. Let us consider the following choice for t: $t(n) = 2^{(\lg n)^{1-1/\lg\lg n}}$. By simplifying it we obtain $t(n) = \sqrt{n}$. We can easily observe that $t^i(n) = (n)^{1/2^i}$ and

$$\frac{\lg t^i(n)}{\lg t^{i+1}(n)} = \frac{2^{i+1}}{2^i} = 2$$

Furthermore, if we compute d such that $t^d(n) = 2$ we obtain

$$n^{\frac{1}{2^d}} = 2 \leftrightarrow \frac{1}{2^d}\lg n = 1 \leftrightarrow d = \lg\lg n$$

Hence, the total cost $f(n)$ is given by:

$$f(n) = d + \sum_{j=0}^{d} 2 = \lg\lg n + 2\lg\lg n$$

Thus $f(n) = O(\lg\lg n)$. Hence, the schemes presented in this section do not improve the upper bounds, even with an *a priori* knowledge of n. This is not surprising as in the next section we prove that this is the best that can be done.

3.6 Discussion

The above schemes are based on idea of using a tree organization and increasing the arity of the tree nodes as a function of their depth. We noticed that faster the rate at which these degrees grew the better performance we had in terms of the worst–case complexity for **insert** and **precedes**. A natural extension of the above schemes would be to let the degrees of the nodes grow at an even faster rate than prescribed by the last scheme in the above section. It is possible to show that natural extensions of this scheme do not provide better results than the scheme with $d(i) = t(i)$. For example, using $d(i) = \sum t(i)$ or $d(i) = d(i-1)^2$ we still obtain a worst-case time complexity of $\Theta(\lg\lg n)$ for the two operations. Schemes that make $d(i)$ grow at a faster rate may lead to a degradation of performance. For example, using $d(i) = 2^{d(i-1)}$ leads to a $\Omega(n)$ worst-case time complexity per operation for both **insert** and **precedes**.

4 Lower Bound of $\mathcal{TP}$ on PPM

The purpose of this section is to demonstrate a lower bound on PPMs for the $\mathcal{TP}$ problem.

Theorem 4.1 *The temporal precedence problem requires $\Omega(\log\log n)$ time on a pointer machine.*

Proof: Without loss of generality, we can assume that each record points to at most two other records. Any algorithm to implement the **precedes** operation of the $\mathcal{TP}$ problem receives just two pointers to records representing the elements to be compared. It may also keep a constant number of records in its registers but this can be modeled by having every element pointing to those "global" records through a chain of pointers. This will not change the asymptotic performance of the algorithm. So without loss of generality, the algorithm only receives the two pointers and must access other records by following pointers from the two pointers. Assuming this, we prove that any algorithm P that implements the **precedes** operation must inspect at least

$$\frac{1}{3}\lg\lg n$$

records, where n is the number of elements inserted. The proof will show that this is true even if the number of elements is known in advance and all the insertions are made right in the beginning before any precedence question is asked.

We can view the data structure that results after the sequence of insertions as a directed graph, where each record is a node and pointers from the record to other records are edges. Since the fields in the records are ordered, we can give each edge a label, 1 or 2, based on whether it is the first or the second field of the record. The distance of a record v from another record u, $d(u,v)$, is the minimum number of edges needed to go from u to v in the induced graph.

If the algorithm can inspect only

$$\frac{1}{3}\lg\lg n$$

records, it means it can only look at nodes at distance at most $\frac{1}{3}\lg\lg n$ from the starting records. The number, m, of such nodes satisfies

$$m \leq 2^{\frac{1}{3}\lg\lg n} = (\lg n)^{\frac{1}{3}}$$

Let S_i be the set of nodes at distance at most $\frac{1}{3}\lg\lg n$ from the record associated with element i.

We now need a collection of those sets with a special property. A *sunflower* is a collection of sets $S_1, S_2, ...S_p$ called *petals*, each of cardinality at most m, such that all pairs of sets in the collection have the same intersection, called the *core* of the sunflower. The Erdös-Rado lemma [5] says that any collection of more than $M = (p-1)^m m!$ non-empty sets, each of cardinality m or less will contain a sunflower with p petals. From this lemma it follows that our collection $S_1...S_n$ contains a sunflower with $p = n^{\frac{1}{2m}}$ petals, because $(n^{\frac{1}{2m}} - 1)^m m!$ is at most n.

Now, let's focus on the p elements generating the petals $S_1, ..., S_p$, and let C be the core of the sunflower. For each i, consider the subgraph G_i induced by the nodes in S_i. For i and j, we say that G_i is *similar* to G_j if: there exists an isomorphism f from G_i to G_j

- $f(i) = j$;
- f respects the labeling of the edges, i.e., (u,v) is an edge labeled x in G_i, iff $(f(u), f(v))$ is an edge of G_j labeled x;
- for each $v \in C$, $f(v) = v$.

Observe that with these three conditions, it is also true that for every $v \in C$, a labeled path $\pi = \langle a_1, \ldots, a_k\rangle$ (where each a_l is either 1 or 2) that leads from i to v in G_i, also leads from j to v in G_j. node from the core C occupies exactly the same position in G_i and G_j. Thus the nodes in C occupy the "same position" in G_i and G_j with respect to the "starting" node (i in G_i and j in G_j). Notice that even though this similarity relation is not in general an equivalence relation on all sets $S_1...S_n$, it is an equivalence relation on the petals by the property of the sunflower.

We first argue that if G_i and G_j are similar, for $i \neq j$, then the algorithm cannot correctly decide whether i precedes j. We then use a counting argument to show that there must exist similar graphs among those induced by the petals.

Since the algorithm exploring the graph can only compare pointers, if G_i and G_j are similar, the algorithm, given i and j, will behave exactly the same way as when given j and i, forcing it to produce the same answer, which should be actually be the opposite. This is because the result of each comparison between pointers will be exactly the same in both cases. This is quite obvious if one or both of the pointer involved in the comparison is a pointer to an element in the core (core elements are at the exactly the same position in G_i and G_j). If the comparison involve no element from the core, then the result of the comparison is definitely $\neq$.

Since we have p graphs, we now have to show that the number of equivalence classes in the similarity relation is smaller than p, guaranteeing duplicates. Let $c_1, ..., c_r$ be the elements of the core. To describe a graph, we need a list of vertices. Each vertex can be identified by an integer between 0 and r, 0 representing "not in the core", and other integers identifying which core element is used, and we need an additional bit to mark the "starting node" (the i of G_i). Then, we have the list of at most $2m$ edges, each identified by a pair of vertices. Since $r \leq m$, the total representation will use at most $O(m \lg m)$ bits. So the number of different (non similar) graphs is at most $2^{O(m \lg m)}$. To show this is $< p$, replacing m and p by their value in terms of n, we need:

$$2^{O(m \lg m)} < n^{\frac{1}{2m}}$$

$$2^{O(m^2 \lg m)} < n$$

$$O(m^2 \lg m) < \lg n$$

$$O((\lg n)^{\frac{2}{3}} \lg \lg n) < \lg n$$

which is always true for n sufficiently large. □

Thus, the lower bound for the $\mathcal{TP}$ problem on PPMs corresponds to the upper bound obtained using the tree-based schemes presented in the previous section. Thus, the tree-based scheme with $d(i) = t(i)$ is indeed optimal on pointer machines.

5 Conclusions

In this paper we have described and studied the Temporal Precedence Problem, i.e., the problem of dynamically supporting the execution of the operations **insert** and **precedes**, on a pointer machine. We have proposed various algorithms to implement these operations, studied their worst–case time complexity, and proved an upper bound of $O(\lg \lg n)$ per operation for the problem. We have demonstrated that the problem has a lower bound time complexity of $\Omega(\lg \lg n)$ on pointer machines. This proves that the data structures described are optimal on PPMs.

These results can be applied directly to state upper bounds for some problems related to parallel execution in non-deterministic programming frameworks (e.g., Prolog)—we have developed [12] and implemented [11] more efficient execution schemes for and-parallel Prolog using some of the ideas presented. In particular, the lower bound presented in this paper confirms the conjecture presented in [11], which claimed the impossibility of providing a solution to the problem of supporting dependent and-parallelism in parallel logic programming with a constant-time overhead per operation.

References

[1] Ben-Amram, A. What is a Pointer Machine? Tech. rep., DIKU, University of Copenhagen, 1995.

[2] Blum, N. On the single-operation worst-case time complexity of the disjoint set union problem. *SIAM Journal on Computing 15*, 4 (1986).

[3] Chazelle, B. A Functional Approach to Data Structures and its Use in Multidimensional Searching. *SIAM Journal of Computing 17*, 3 (1988).

[4] Dietz, P., and Sleator, D. Two Algorithms for Maintaining Order in a List. In *Symp. on Theory of Computing* (1987), ACM Press.

[5] Erdos, P., and Rado, R. Intersection Theorems for Systems of Sets. *J. London Math. Soc. 35* (1960).

[6] Harel, D., and Tarjan, R. Fast Algorithms for Finding Nearest Common Ancestor. *SIAM Journal of Computing 13*, 2 (1984).

[7] Knuth, D. *The Art of Computer Programming*, vol. 1. Addison-Wesley, 1968.

[8] Kolmogorov, A., and Uspenskii, V. On the Definition of an Algorithm. *Uspehi Mat. Nauk 13* (1958).

[9] Mehlhorn, K., and Naher, S. Dynamic Fractional Cascading. *Algorithmica 5* (1990).

[10] Mhlhorn, K., Naher, S., and Alt, H. A Lower Bound on the Complexity of the Union-Split-Find Problem. *SIAM Journal of Computing 17*, 6 (1988).

[11] Pontelli, E., and Gupta, G. Implementation mechanisms for dependent and-parallelism. In *International Conference on Logic Programming* (1997), MIT Press.

[12] Pontelli, E., Ranjan, D., and Gupta, G. On the Complexity of Parallel Implementation of Logic Programs. In *Proceedings of the International Conference on Foundations of Software Technology and Theoretical Computer Science* (1997), Springer Verlag.

[13] Poutré, H. L. Lower Bounds for the Union-Find and the Split-Find Problem on Pointer Machines. *Journal of Computer and System Sciences 52* (1996), 87–99.

[14] Schönhage, A. Storage Modification Machines. *SIAM Journal of Computing 9*, 3 (August 1980), 490–508.

[15] Tarjan, R. A Class of Algorithms which Require Nonlinear Time to Mantain Disjoint Sets. *Journal of Computer and System Sciences 2*, 18 (1979).

[16] Tsakalidis, A. Maintining Order in a Generalized Linked List. *ACTA Informatica 21* (1984).

[17] Tsakalidis, A. The Nearest Common Ancestor in a Dynamic Tree. *ACTA Informatica 25* (1988).

[18] van Emde Boas, P., Kaas, R., and Zijlstra, E. Design and Implementation of an Efficient Priority Queue. *Mathematical Systems Theory 10* (1977).