PRESERVING ORDER IN A FOREST IN LESS THAN LOGARITHMIC TIME

P. van Emde Boas*
Mathematical Centre, Amsterdam, Netherlands /
Mathematical Institute, University of Amsterdam

ABSTRACT

We present a data structure, based upon a stratified binary tree, which enables us to manipulate on-line a priority queue whose priorities are selected from the interval 1...n, with an average and worst case processing time of $O(\log \log n)$ per instruction. The structure is used to obtain a mergeable heap whose time requirements are about as good.

1. INTRODUCTION

The main problems in the design of efficient algorithms for set-manipulation result from the incompatible requests posed by the distinct operations one likes to execute simultaneously. Instructions for inserting or deleting or for testing membership of elements in sets require a data structure supporting random access. On the other hand instructions for computing the value of the smallest or largest element, or the successor or predecessor of a given element, require an ordered representation. Finally instructions which unite two sets, so far, have only been implemented efficiently using a tree structure.

An example of an efficient algorithm which resolves one of these conflicts is the well-known union-find algorithm; its worst case average processing time per instruction has been shown to be of the order A(n) in case of $O(n)$ instructions on an n-elements universe, where A is the functional inverse of a function with Ackerman-like order of growth (cf. [1], [9]).

The algorithms published until now to resolve the conflicting demands of order and random access all show a worst case processing time of $O(\log n)$ per instruction for a program of $O(n)$ instructions on an n-elements universe which has to be executed on-line. Clearly we should remember that each instruction repertoire which enables us to sort n reals by issuing $O(n)$ instructions needs an $O(\log n)$ processing time for the average instruction in doing so. However if the universe is assumed to consist of the integers 0...n-1 only, the information-theoretical lowerbound on the complexity of sorting does not apply; moreover it is known that n integers in the range 1...n can be sorted in linear time.

Data structures which have been used to solve the conflict between order and random access are (among others) the binary heap, AVL trees and 2-3 trees. In AHO, HOPCROFT & ULLMAN [1] 2-3 trees are used to support the instruction repertoire INSERT, DELETE, UNION and MIN with a worst case processing time of order $O(\log n)$ per instruction. The authors introduce the name *mergeable heap* (resp. *priority queue*) for a structure supporting the above operations (excluding UNION).

The $O(\log n)$ processing time for manipulating priority queues and mergeable heaps sometimes becomes the bottleneck in some algorithms; as an example I mention TARJAN's recent algorithms to compute dominators in directed graphs [10]. Consequently, if we can do the mergeable heaps more efficiently, the order of complexity of this algorithm can be reduced.

Another example is given by the efficient algorithm for generating optimal Prefix Code, which was published recently by PERL, GAREY & EVEN [5]. The factor log n appearing in the worst-case run-time is caused by the use of a binary heap for implementing a priority queue.

In the present paper I describe a data structure which represents a priority queue with a worst case processing time of $O(\log \log n)$ per instruction, on a Random Access Machine. The storage requirement is of the order $O(n \log \log n)$. The structure can be used in combination with the tree-structure from the efficient union-find algorithm to produce a mergeable heap with a worst-case processing time of $O((\log \log n) \cdot A(n))$ and a space-requirement of order $O(n^2)$. The possible improvements of the space requirements form a subject of continued research.

1.1. Structure of the paper

Section 2 contains some notations and background information, among which a description of the efficient union-find algorithm. In section 3 we present a "silly" implementation of a priority queue with an $O(\log n)$ processing time per instruction. Reconsidering this implementation we indicate two possible ways to improve its efficiency to $O(\log \log n)$.

In section 4 we describe our stratified trees and their decomposition into canonical subtrees. Next we show how these trees can be used to describe a priority queue with an $O(\log \log n)$ worst and average case processing time per instruction. The algorithms for performing the elementary manipulations on stratified trees are presented and explained in section 5. The algorithms are derived from a PASCAL implementation of our priority queue which was written by R. KAAS & E. ZIJLSTRA at the University of Amsterdam [8]. It is explained how the complete stratified tree is initialized using time $O(n \log \log n)$. Section 6 discusses how the structure can be used if more than one priority queue has to be dealt with; the latter situation arises if we use our structure for implementing an efficient mergeable heap. Finally, in section 7, we indicate a few relations with other set-manipulation problems.

Throughout sections 4, 5 and 6 identifiers typed in this different type font denote the values and meanings of the same identifiers in the PASCAL implementation.

2. GENERAL BACKGROUNDS

2.1. Instructions

Let n be a fixed positive integer. Our universe will consist of the subsets of the set $\{1,\ldots,n\}$. For a set S in our universe we consider the following instructions to be executed on S:

MIN	: Compute the least element of S
MAX	: Compute the largest element of S
INSERT (j)	: $S := S \cup \{j\}$
DELETE (j)	: $S := S \setminus \{j\}$
MEMBER (j)	: Compute whether $j \in S$
EXTRACT MIN	: Delete the least element from S
EXTRACT MAX	: Delete the largest element from S
PRECEDECESSOR (j)	: Compute the largest element in $S < j$
SUCCESSOR (j)	: Compute the least element in $S > j$

*) Work supported by grant CR 62-50. Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

NEIGHBOUR (j): Compute the neighbour of j in S (see definition below).
ALL MIN (j) : remove from S all elements $\leq$ j
ALL MAX (j) : remove from S all elements $\geq$ j.

(If an instruction cannot be executed properly, e.g. MIN if S = $\emptyset$, an appropriate action is taken).

The neighbour in S of a number j $\in \{1,\ldots,n\}$ is the element i $\in$ S such that i - 1 has the largest segment of significant digits in its binary development in common with j - 1; if more than one element in S fits this description, the one among them which is the nearest to j in the usual sense is selected.

The neighbour of j is always to be found among the predecessor and successor of j, but it is difficult to tell in advance which of the two it will be. In section 3 we explain in what "geometrical" sense the neighbour of j is indeed the element in S which is the nearest to j. For this moment we give the following

Example: Let n = 16. S = {1,5,13,14}. The corresponding binary representations are 0000, 0100, 1100 and 1101. The neighbour of 4 (corresponding to 0011) equals 1 whereas the neighbour of 15 (corresponding to 1110) equals 14, which is nearer in the usual sense to 15 as 13.

2.2. Priority queues

A *priority queue* is a data structure representing a single set S $\subset \{1,\ldots,n\}$ on which the instructions INSERT, DELETE, and MIN can be executed *on-line* (i.e. some arbitrary order and such that each instruction should be executed before reading the next one). Although the priority queue is our main target, we mention at this point that actually the complete instruction repertoire given above is supported on our data structure with a worst and average case processing time of $O(\log \log n)$ per instruction (except for the last two instructions where the processing time is $O(\log \log n)$ for each element removed).

The complete list of instructions above will be called the *extended repertoire* hereafter.

2.3. Union-find problem

For arbitrary partitions Π = {A,B,...} of {1,...,n} we consider the following instructions:

FIND (i) : compute the set currently containing i
UNION (A,B,C): Form the union of the sets A and B and give the name C to this union.

There is no specific name for a data structure supporting these two instructions; the problem of manipulating such a structure is known as the *union-find problem*.

The well known efficient union-find algorithm uses a representation of sets by means of trees. Each node in a tree corresponds to a member of a set and contains a pointer which either points to the name of the set if the node happens to be the root of his tree, or to his father in the tree otherwise. A UNION instruction is executed by making the root of the smaller tree a direct son of the larger one (balancing). To execute a FIND instruction the node corresponding to the element asked for is accessed directly, and his pointers are followed until the root of his tree is found; in the mean time all nodes which are encountered during this process are made direct descendants of the root, thus reducing the processing time at subsequent searches.

It has been established only recently how efficient the above algorithm is. Whereas its average processing time has been estimated originally as $O(\log \log n)$ (FISHER [6]) and $O(\log^* n)$ (HOPCROFT & ULLMAN [7] and independently PATERSON (unpublished)), a final upper and lowerbound $O(A(n))$ has been proved by TARJAN [9]. Remember that $\log^* n$ is the functional inverse of the function $2^{2^{\cdot^{\cdot^{2}}}}$ n i.e. $\log^* n$ equals the number of applications of the base-two logarithm needed to reduce n to zero. The function A(n) is a functional inverse of a function of Ackerman-type which is defined as follows: Define a by: a(0,x) = 2x; a(i,0) = 0; a(i,1) = 2 for i $\geq$ 0; and a(i+1,x+1) = a(i,a(i+1,x)). Then we let A(n) = min{j | a(j,j) $\geq$ n}.
(The above definitions differ only inessentially from the ones given by TARJAN).

2.4. Mergeable heaps

A *mergeable heap* is a data structure which supports the instructions INSERT, DELETE, MEMBER and MIN on sets which themselves can be united and searched, i.e. UNION and FIND are also supported. A mergeable heap may be obtained from the union-find structure by replacing the unordered collection of sons of a certain node by a priority queue where the "value" of a node equals the minimal element in the set formed by this node and its descendants.

In such a representation the instructions are executed as follows:

UNION: The root-priority queue of the structure containing the least number of elements is inserted in the root-priority queue of the other structure at the place corresponding to its least element.

FIND: First one proceeds from the element itself "upwards" to find the root-priority queue of the structure to which it belongs. Next, going downwards from this root back to the element the priority queues along this path are disconnected by delete operations. The queues are then inserted in the root priority queue at the position of their (possibly modified) least element.

MIN: By executing a min-instruction at the root-priority queue of a structure its least element will become known; a FIND instruction on this element will yield access to the location where it is stored.

INSERT & DELETE: These operations are reduced to the priority queue insert and delete by first executing a FIND instruction. The same holds for MEMBER.

In doing so the average processing time for an instruction becomes A(n) times the processing time for the priority queue instructions used. As long as the latter time is not reduced below $O(\log n)$ the proposed representation of a mergeable heap should be considered inefficient, since there are $O(\log n)$ structures known for mergeable heaps (2-3-trees with unordered leaves [1]). Using our new efficient priority queue the proposed scheme becomes (as far as time is concerned) more efficient than the traditional ones. For the space requirements the reader is referred to section 6.

3. A "SILLY" PRIORITY QUEUE WITH $O(\log n)$ PROCESSING TIME

3.1. The structure

The scheme described in this section is designed primarily in order to explain the ideas behind the operations to be executed on the much more complicated structure in the next section.

We assume in this section that $n = 2^k$. We consider a fixed binary tree of height k. The $2^k = n$ leaves of this tree will represent the numbers 1...n in their natural order from left to right. The leaves thus represent the potential members of the set S. If we had counted from 0 to n - 1 this order is nothing but the interpretation of the binary representation of a number as an encoding of the path from the root to the leaf; the binary digits are read from left to right where 0 denotes "go left" and 1 means "go right".

To each node in the tree we associate three pointers, linking the node to its father and its left- and righthand son. Moreover each node has a one-bit mark

field.

A subset $S \subseteq [1...n]$ is represented by marking all the leaves corresponding to members of S, together with all nodes on the paths from these leaves to the root of the tree; see diagram 1.

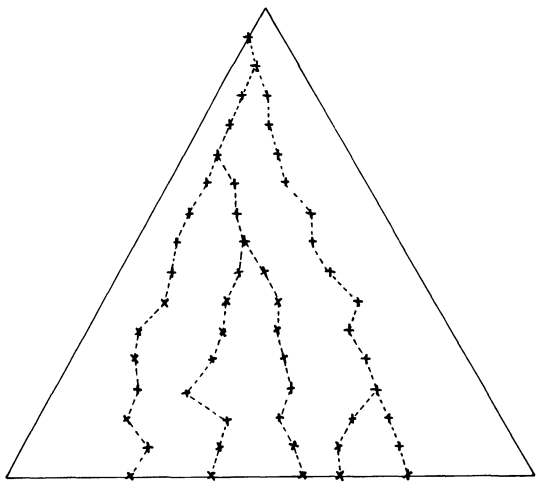


Diagram 1. Example of a five-element set representation using mark bits.

It is easy to see that, using this representation, the operations in the list in section 2 can be executed in time $O(k) = O(\log n)$ for each item processed. (Note that the operations ALLMIN and ALLMAX may remove more than one item.)

We present the following sketches of algorithms:

INSERT (i) : mark leaf i and all nodes on the path from leaf i to the root, until you encounter a node which was already marked.

DELETE (i) : unmark leaf i and all nodes on the path from leaf i to the root upto but not including the lowest node on this path having two marked sons.

MEMBER (i) : test whether leaf i is marked.

MIN (MAX) : proceed from the root to the leaves selecting always the leftmost (rightmost) present son.

EXTRACT MIN (EXTRACT MAX): min (max) followed by delete

ALLMIN (j) : while MIN ≤ j do EXTRACTMIN od

ALLMAX is defined analogously.

PREDECESSOR (j): proceed from leaf j to the root until a node is encountered having j as a righthand side descendent where the lefthand son is marked. Proceed from this lefthand son to the leaves always taking the rightmost present son.

SUCCESSOR (j) is defined analgously.

Note that all instructions except PREDECESSOR and SUCCESSOR use the lowest marked node on the path from an unmarked leaf to the root or the lowest *branchpoint* (i.e. a node having both sons marked) on the path from a marked leaf to the root. An analogous instruction which does not climb beyond the lowest "interesting" node is the instruction

NEIGHBOUR (j) : proceed from leaf j to the lowest node such that the "other" son of this node is marked. If this other son is a lefthand son then proceed from this node to the leaves always selecting the rightmost marked leaf; otherwise select always the leftmost marked leaf.

If we represent the set furthermore using a doubly linked list, such that each marked leaf contains (a pointer to) the corresponding entry in the list, a call of NEIGHBOUR followed by one or two steps in the list will be adequate to execute a call of PRECEDECESSOR or SUCCESSOR.

From the descrintion of the instruction NEIGHBOUR it is clear in which sense the neighbour of a number j is the "nearest" present element to j; the neighbour has the largest possible number of common ancestors and in case this condition does not define the neighbour unambigously the neighbour is the other descendent of the lowest common ancestor which has developed as near to j as possible.

3.2. Improvements of the time efficiency

It is clear from the above descriptions that our "silly" structure supports the extended repertoire with an $O(\log n)$ processing time per instruction. Using the doubly linked list as an "extra" representation INSERT and DELETE and NEIGHBOUR take time proportional to the distance in the tree traversed upon a call whereas MIN and MAX take constant time.

The remaining instructions are "composite". This observation opens a way to improve the efficiency. The time saved by not climbing high upwards in the tree can be used to perform more work at a single node. For example, if we decide to use at each node a linear list of present sons instead of a fixed number, we can easily accomodate for a tree with branching orders increasing from the root to the leaves without disturbing the $O(k)$ processing time. Using a tree with branching orders $2,3,4,\ldots,k$ which contains $k! = n$ leaves, we can maintain a priority queue of size $O(k!)$ in time $O(k)$; the above set up yields therefore an $O(\log n/\log\log n)$ priority queue which is already better than we had before.

There is, however, much more room for improvement. The operations which we like to execute at a single node are themselves priority queue operations. Consequently using a binary heap we can accomodate for the branching orders $2,4,8,\ldots,2^k$, which yields a priority queue of size $O(2^{k^2/2})$, and the processing time is reduced to $O(\sqrt{\log n})$.

Note that in both modifications the space requirements remain of $O(n)$, which is not true for the structure described in §4.

According to the "divide and conquer" strategy, we should however use at each node the same efficient structure which we are describing. This suggests the following approach. The universe $[1...n]$ is divided into $\sqrt{n}$ blocks of size $\sqrt{n}$. Each block is made a priority queue of size $\sqrt{n}$, whereas the blocks themselves form another priority queue of this size. To execute an INSERT we first test whether the block containing the element to be inserted contains already a present element. If so, the new element is inserted in the block; otherwise the element is inserted as first element in its block and the complete block is inserted in the "hyper-queue". A DELETE instruction can be executed analogously.

Assuming that we can implement the above idea in such a way that inserting a first and deleting the last element in a block takes constant time *independent of the size of the block*, the above description yields for the run-time a recurrence equation of the type $T(n) \leq \leq T(\sqrt{n}) + 1$ which has as a solution $T(n) \leq O(\log\log n)$.

Another way to improve the "silly" representation which leads again to the same efficiency is conceived as follows. As indicated the "hard" instructions proceed by traversing the tree upwards upto the lowest "interesting" node (e.g. a branchpoint), and proceding downwards along a path of present node.

If these traversals could be executed by means of a "binary search on the levels" strategy, the processing time is reduced from $O(k)$ to $O(\log k) = O(\log\log n)$. A similar idea is involved in the effi-

cient solution of a special case of the lowest common ancestor problem given by AHO, HOPCROFT & ULLMAN [2].

The reader should keep both approaches in mind while reading the sequel of this paper.

4. A STRATIFIED-TREE STRUCTURE

4.1. Canonical subtrees and static information

In this section we let h be a fixed positive integer. Let $k = 2^h$ and $n = 2^k$. We consider a fixed binary tree T of height k with root t having n leaves.

For $1 \leq j$ we define RANK (j) to be the largest number d such that $2^d \mid j$ and $2^{d+1} \nmid j$. For example RANK (12) = 2 and RANK (17) = 0. By convention we take RANK (0) = h + 1.

Note that for $j > 0$, RANK (j) = d and $j - 2^d > 0$ we have RANK (j) < RANK ($j+2^d$) and RANK (j) < RANK ($j-2^d$); moreover RANK ($j+2^d$) ≠ RANK ($j-2^d$).

The *level* of a node v in T is the length of the path from the leaves of T to v; the *rank* of v is the rank of the level of v. Note that the rank of the leaves equals h + 1, and the rank of the top equals h; all other nodes have lower ranks. The *position* of a leaf is the number in the set {1,...,n} represented by this leaf. The *position* of an internal node v equals the position of the rightmost descendent leaf of its left-hand son; this number indicates where the borderline lies from the two parts resulting from splitting the tree along the path from v to the root.

A *canonical subtree* (CS) of T is a binary subtree of height 2^d having as root a node of rank ≥ d; the number d is called the rank of the CS. The subtree of a CS consisting of its root with all its left(right) hand side descendents is called a *left(right) canonical subtree*.

Clearly the complete tree is a canonical subtree of rank h; it is decomposed into a top tree of rank h - 1 and $2^{k/2}$ (= $\sqrt{n}$) bottom trees of the same rank, which is in accordance with the "divide and conquer" approach of a "hyper-queue" of "subqueues" suggested in the preceding section.

To any node v of T we associate the following subtrees which are called the canonical subtrees of v. Let d = RANK (v).

UC(v): the unique canonical subtree of rank d having v as a leaf.
LC(v): the unique canonical subtree of rank d having v as a root.

Note that UC(v) is not defined if v is the root whereas LC(v) is not defined if v is a leaf of T. When d = 0, UC(v) and LC(v) consist of three nodes. Note moreover that the rank of the root of UC(v) and the rank of the leaves of LC(v) is higher than d.

The left(right) canonical subtree of LC(v) is denoted LLC(v) (RLC(V)). LC(v) and the half of UC(v) containing v together form the *reach* of v, denoted R(v). The dynamical information stored at v depends only on what happens within its reach. The reach of the top is the complete tree, whereas the reach of a leaf is the set of leaves. See diagram 2 for an illustration.

Clearly the reach of an internal node v of rank d is a subset of some canonical subtree of rank d + 1, denoted C(v). We say that v lies at the *center-level* of C(v); moreover, v is called the *center* of its reach R(v).

For each node v and each $j \leq h$ we denote by FATHER(v,j) the lowest proper ancestor of v having rank ≥ j. Clearly FATHER(v,h) equals the root t of T, whereas FATHER(v,0) is the "real" father of v in T (provided v ≠ t). At each node we have an array of h pointers father[0 : h-1] such that father[i] yields the rank - i father of v. Since FATHER(v,h) always yields the root of the tree this element doesn't need to be included. These pointers enable us to climb along a path in the tree to a predetermined level in $O(h)$ steps. Moreover, given the root of a cs U and one of its leaves, we can proceed in a single step to the center of the smallest reach containing the two which is entirely contained within U.

The static information at a node contains moreover its position and if it is an internal node its rank and level. The static information can be allocated and initialized in time $O(n \log \log n)$; details will be given in the next section.

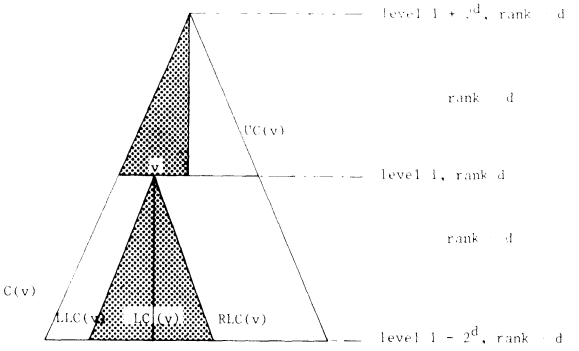


Diagram 2: The canonical subtrees of v. R(v) is the shaded area.

4.2. Dynamical information

The dynamical information at internal nodes is stored using four pointers l min, l max, r min and r max and an indicator field ub, which can assume the values plus, minus and undefined. At leaves the dynamical information consists of two pointers successor and predecessor, and a boolean present.

Let $S \subset \{1,...,n\}$ be a set which has to be represented in our stratified tree. We say that the leaves corresponding to members of S and all their ancestors in the tree are *present*; the present nodes are exactly the nodes which were marked in our silly structure. A present node can become *active* and in this case its information fields contain meaninful information. The values of these fields of a non-active internal node are: l min = nil, l max = nil, r min = nil, r max = nil and ub = undefined. For a non-active leaf these values are predecessor = nil, successor = nil, present = false. For an active leaf v the meaning of these fields should be:

predecessor: points to the leaf corresponding to the predecessor in S of the number corresponding to v if existent; otherwise predecessor = nil.
successor : analogous for the successor
present = true

Remember that a branchpoint is an internal node having two present sons.

Let v be an internal node, and denote the top of C(v) by t. If v is active its dynamical information fields have the following meaning:

l min: points to the leftmost present leaf of LLC(v) if such node exists; otherwise l min = nil.
l max: idem for the rightmost present leaf of LLC(v)
r min: idem for the leftmost present leaf of RLC(v)
r max: idem for the rightmost present leaf of RLC(v)
ub = plus if there occurs a branchpoint in between v and t, and minus otherwise.

If v is an active internal node it is present and consequently LC(v) contains at least one present leaf; this shows that it is impossible to have an active internal node with four pointers equal to nil.

As suggested in the preceding section the time

needed to insert a first or to delete a last element should be independent of the size of the tree. This is realized by preventing present nodes from becoming active unless their activity is needed. This is expressed by the following.

Properness condition: Let v be a present internal node. Then v is active if and only if there exists a branchpoint in the interior of the reach of v (i.e. there exists a branchpoint u $\in$ R(v) which is neither the top nor a leaf of C(v)).

A leaf is active if and only if it is present; the root is active iff the set is non-empty.

(Actually the case where S = $\emptyset$ is degenerate and leads to several programming problems, which were prevented in practice by including n in S as a permanent member.)

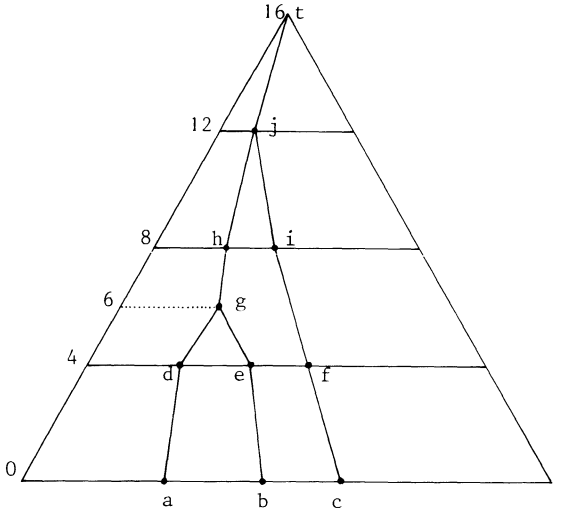
If the internal node v is non-active but present then there is a unique path of present nodes going from the top t of R(v) to a unique present leaf w of C(v) contained in R(v). In our approach we can proceed from t to w and backwards without ever having to visit v, making it meaningless to store information at v.

If some canonical half-tree has two present leaves then all its present nodes at its center level are active. Also if a node v of rank d is active then FATHER (v,d) is active as well. We leave the verifications of these assertions as an exercise to the reader.

The set S $\subset$ {1,...,n} is represented as follows. First the leaves corresponding to the elements of S and all their ancestors are declared to be present. Next we compute using the properness condition which present nodes become active. Finally the dynamical fields of all active and non-active nodes are given their proper values. The resulting information content is called the representation the the set S. We leave it to the reader to convince himself that this representation is unique.

(In our actual program the structure is initialized at S = {n}, representations of all other sets being the result of execution of a sequence of instructions from the extended repertoire.)

An example of a proper information content is given in diagram 3 (omitting the evident doubly linked list data). The symbol $\wedge$ denotes nil resp. undefined.



	LEVEL	RANK	LMIN	LMAX	RMIN	RMAX	UB
t	16	4	a	c	$\wedge$	$\wedge$	-
j	12	2	h	h	i	i	-
h	8	3	$\wedge$	$\wedge$	a	b	+
i	8	3	$\wedge$	$\wedge$	c	c	+
g	6	1	d	d	e	e	-
d	4	2	a	a	$\wedge$	$\wedge$	+
e	4	2	$\wedge$	$\wedge$	b	b	+
f	4	2	$\wedge$	$\wedge$	$\wedge$	$\wedge$	$\wedge$

Diagram 3: Example of a proper information content.

5. OPERATIONS ON THE STRATIFIED TREE STRUCTURE

Once having described the representation of a set S by assigning values to particular fields in the stratified tree, the next step is to indicate how the set-manipulation operations mentioned in section 1 can be executed such that

(i) a processing time of $O(h) = O(\log \log n)$ is realized;

(ii) the structure of the representation is preserved, i.e. the properness condition should remain valid.

Moreover we must indicate how the static information together with a legitimate initial state for the dynamic information can be created in the proper time and space (i.e. both of order n log log n).

In this section we pay no attention to the self-evident operations needed to manipulate the doubly linked list structure formed by the leaves of our tree. Furthermore we assume that always n $\in$ S; the driver will insert this element at initialization and will take care that this element is never deleted from S.

5.1. Initialization

Initialization takes place during a single tree-transversal in pre-order. When a node is processed its father-pointers and its position, and in case of internal nodes its rank and level are stored in the appropriate field. The needed computations are based on the following relations:

(i) the fathers of the top are nil; the fathers of a direct son v of a node w where RANK(w) = d satisfy

FATHER (v,j) = w for j $\leq$ d
FATHER (v,j) = FATHER (w,j) for j > d.

The node w is accessible during the processing of v by use of a parameter fath in the recursive procedure which executes the tree-transversal.

(ii) the level of a node is one less that the level of its father

(iii) the position of the leftmost node at level i > 0 equals 2^{i-1}; the position of any other node at level i equals 2^i + the position of the last node at level i processed before; the leaves are processed in increasing order of their position

(iv) the rank of a node depends only on the level, and can be stored using a pre-computed table of size k = log n.

Once having pre-computed the needed powers of 2 by repeated additions, the above relations show how the static structure is initialized without having "illegitimate" instructions like multiplications and bit-manipulations, in time $O(\log \log n)$ per node processed. Since there exist 2.n - 1 nodes this shows that the initialization takes time $O(n \log \log n)$. The space $O(n \log \log n)$ follows since the space needed for each node is $O(\log \log n)$.

Pre-computing of the ranks in time $O(\log n)$ using only additions is left as an exercise to the reader.

5.2. Operations

The extended instruction repertoire can be expressed (disregarding the doubly linked list operations) in terms of three primitive operations insert, delete and neighbour. Each of these operations is described by a linearly recursive procedure. The procedures are called upon the complete tree of rank h. If called upon a canonical subtree the procedures either terminate within constant time independent of the rank, or the procedure executes a single call of a top or bottom canonical subtree of rank one less preceded and followed by a sequence of instructions taking constant time independent of the rank. A call upon a subtree of rank 0 terminates without further recursive calls of the procedure. From

the above assertions which can be verified by inspection of the procedure bodies, it follows directly that the run-time of each procedure is of order h = log log n. Concerning the preservation of the correct structure, I refer to the PASCAL implementation which has worked without errors. Moreover I feel that the correctness of the algorithms can be proved using one of the more informal approaches based on recursion-induction, but no such proof has been given till now; this approach was used successfully during the debugging stage of the development of the implementation. To stimulate research in correctness proofs, I will award the prize of ten dollars (US $10.00) to the first person submitting a convincing correctness proof of my procedures along the lines sketched above.

In the execution of an algorithm we have frequently the situation that we have a CS with root t and leaf v and that we want to inspect or modify the fields at t in the direction of v, i.e. the left-hand fields at t if v is a left-hand descendent of t etc. To decide whether a certain descendant of t lies in the left- or right-hand subtree it is sufficient to compare the positions of the two nodes. We have in general:

The descendant v of t is a left-hand descendant iff the position of v is not greater than the position of t.

Actually the position of a node was introduced to facilitate this easy test on the handiness of a descendant.

The procedures insert, delete and neighbour use the following primitive operations.

myfields (v,t)	yields	a pointer to the fields at t in the direction of v. This pointer is of the type fieldptr.
mymin (v,t)	yields	the value of the min-field at t in the direction of v (which happens to be a pointer).
mymax (v,t)		analogeous for the maxfield.
yourfields (v,t), yourmin (v,t) and yourmax (v,t)	yield	the analogeous values of the field at t in the other direction.
minof (t)	yields	the leftmost value of the four pointed fields at t if t is active, and nil otherwise.
maxof (t)	yields	the rightmost value analogeously.

The type ranktp is the subrange 0..h.

Finally the procedure clear gives the dynamic fields at its argument the values corresponding to the non-active state. The identifiers mentioned in the procedures mostly are of the type "pointer to node" (ptr) where "node" is a record-type containing the fields mentioned in the preceding sections.

5.2.1. The procedure insert

insert is a function procedure yielding as result the value of a pointer to the neighbour of the node being inserted. This neighbour is subsequently used for inserting the mode into the doubly linked list. (It should be mentioned that we tacitly have generalized the meaning of neighbour to the case of a CS which is not the complete tree.)

insert has five parameters called by value; its procedure heading reads:

```
function insert (leaf, top, pres: ptr; no branchpoint:
                 boolean; order: ranktp): ptr;
```

The meaning of the parameters is as follows:

order:	the rank of the CS on which the procedure is called
leaf:	the node to be inserted
top:	the root of the CS on which the procedure is called
pres:	a present leaf of the CS on which the procedure is called of the same handiness as leaf at top
nobranchpoint:	true iff leaf's side of the CS on which the procedure is called contains no branchpoint.

At first glance the parameter pres seems to be unnecessary since its value can be derived from the values of myfields(leaf, top). However in the case where the CS under consideration is a top-CS of a CS of next higher rank the fields at top refer to nodes at a level far below the level of leaf and consequently their values may be misunderstood. This danger (to be dealt with by "dynamic address translation" in the preliminary version of our data structure [3]) can not be solved using bit manipulation instructions on node-addresses since their run-time should be charged according to their length: log n, which clearly is prohibitive. Actually this "mistake" was responsible for the major bugs discovered during the process of implementing our structure.

A call of insert terminates without further recursive calls if leaf's side of the CS under consideration does not contain a present leaf (pres = nil). Otherwise the nodes hl = FATHER (leaf, order-1) and hp = FATHER (pres, order-1) are computed. Now if nobranchpoint is true then hp is present without being active and special actions should be undertaken in this case. In this case hl is present iff hl = hp and depending on this equality either the bottom-call

```
insert(leaf, hl, mymin (leaf, hl), true, order-1)
```

or the top-call

```
insert (hl, top, hp, true, order-1)
```

is executed after having "activated" the right fields at hp and hl.

In this situation the procedure delivers pres as its value.

If nobranchpoint is false then hl is present iff it is active which is tested by inspecting its ub-field. If hl is active the bottom-call

```
insert (leaf, hl, mymin (leaf, hl), mymin (leaf, hl) =
        mymax (leaf, hl), order-1)
```

is executed and its value is yielded as the result of insert. Otherwise, the top-call

```
insert (hl, top, hp, nobranchpoint, order-1)
```

is executed after having set nobranchpoint := (hp↑.ub = minus) and having activated the fields at hl and hp. This call yields as a result the neighbour of hl in the top-tree in nb, and depending the outcome of a comparison between the positions of hl and nb the value of insert equals minof (nb) or maxof (nb).

After these activities the fields at the top may have to be adjusted if the current call is a call on a bottom-CS, which is the case iff order equals the rank of top. From this point of view the complete tree has to be considered a bottom-CS, which explains why the levels are numbered from the leaves to the top instead of the reverse order as was done in the preliminary reports on our structure [3,4].

The initial call of insert reads:

```
insert (pt, root, mymin (pt, root),
        mymin (pt, root) = mymax (pt, root), h)
```

where it is assumed that root is active and pt is not a present leaf. (These conditions are enforced by the driver.)

We now give the complete PASCAL text of insert.

```
function insert(leaf, top, pres : ptr;
     nonbranchpoint : boolean; order : ranktp) : ptr;

    var hl, hp, nb : ptr; fptr : fieldptr;

begin if pres = nil then
  begin fptr:= myfields(leaf, top);
    with fptr↑ do
    begin min:= leaf; max:= leaf end;
     if leaf↑.position <= top↑.position then
      insert:= top↑.right↑.min
     else insert:= top↑.left↑.max
  end else
  begin hl:= leaf↑.fathers[order - 1];
        hp:= pres↑.fathers[order - 1];
    if nobranchpoint then
      if hp <> hl then
        begin fptr:= myfields(leaf, hl);
         with fptr↑ do
          begin min:= leaf; max:= leaf end;
         fptr:= myfields(pres, hp);
         with fptr↑ do
          begin min:= pres; max:= pres end;
         hl↑.ub:= plus; hp↑.ub:= plus;
         nb:= insert(hl, top, hp, true, order - 1);
         insert:= pres
        end else
        begin fptr:= myfields(pres, hp);
         with fptr↑ do
          begin min:= pres; max:= pres end;
         hp↑.ub:= minus;
         insert:= insert(leaf, hl, mymin(leaf, hl),
                         true,order - 1)
        end
    else if hl↑.ub <> undefined then
      insert:= insert(leaf, hl, mymin(leaf, hl),
        mymin(leaf, hl) = mymax(leaf, hl), order - 1)
    else
      begin fptr:= myfields(leaf, hl);
       with fptr↑ do
        begin min:= leaf: max:= leaf end;
       nobranchpoint:= hp↑.ub = minus;
       hl↑.ub:= plus; hp↑.ub:= plus;
       nb:= insert(hl, top, hp,
             nobranchpoint, order - 1);
       if hl↑.position <= nb↑.position then
        insert:= minof(nb) else insert:= maxof(nb);
      end;
    fptr:= myfields(leaf, top);
      if top↑.rank = order then with fptr↑ do
       if leaf↑.position < min↑.position
        then min:= leaf   else
       if leaf↑.position > max↑. position
        then max:= leaf
  end;
end;
```

5.2.2. The procedure delete

The procedure delete yields no value. It has six parameters, the first three of which are called by value, the others being called by reference (although calling them by result should be as good; this is however not possible in PASCAL). The procedure heading reads:

```
procedure delete (leaf, top : ptr; order : ranktp;
                  var pres 1, pres 2 : ptr;
                  var nobranchpoint : boolean);
```

The meaning of the value-parameters is as follows:

leaf: the leaf to be deleted
top: the root of the CS considered
order: the rank of the CS considered

The remaining parameters have after a call of delete the following meaning:

pres 1, pres 2: present leaves in the CS considered, one of them being the neighbour of leaf (see explanation below)
nobranchpoint: true iff there occurs no branchpoint on the path from top to pres 1.

A call of delete should make non-present leaf and its ancestors up to the lowest branchpoint but in doing of other nodes on different paths which were active may have to become inactive. As long as this holds nobranchpoint remains true.

Proceeding downwards from the other son of the lowest branchpoint as near as possible we arrive at the neighbour; if we however select always the remotest present node, we arrive at a node which might be called the *extreme* of leaf in the tree. The extreme, as a "binary approximation" of leaf is as good as the neighbour, but in the usual sense it is as far away as possible.

After a call of delete pres 1 and pres 2 are the neighbour and the extreme of leaf ordered according to their positions (i.e. pres 1↑.pos ≤ pres 2↑.pos).

delete terminates without inner call if the lowest branchpoint equals top; at this time pres 1 and pres 2 are initialized with the values yourmin(leaf, top) and yourmax(leaf, top) and nobranchpoint is made true if these two values are equal.

Updating of these values proceeds depending on whether the call just terminated was a top or a bottom call (which is known to the current incarnation of delete). If the last call was a top call then pres 1:= minof(pres 1); pres 2:= maxof(pres 2) and their equality is tested again to decide whether nobranchpoint should remain true; if so the node formerly pointed at by pres 1 is disactivated.

If the last call was a bottom call the ub field at the former top and the pointers away from pres 1 at this node are inspected to decide whether there occurs a branchpoint at or above this node; if not the former top is disactivated.

The fields at the current top are adjusted only when the current call is a bottom call.

The initial call to delete reads:

```
delete(pt, root, h, pres 1, pres 2, nobranchpoint);
```

The driver makes sure that pt is a present leaf which is not the unique present leaf. The complete text of delete is given below.

```
procedure delete(leaf, top : ptr; order : ranktp;
    var pres1,pres2: ptr; var nobranchpoint : boolean);

    var fptr : fieldptr; hl, hp : ptr;

begin fptr:= myfields(leaf, top);
  with fptr↑ do if min = max then
    begin min:= nil; max:= nil;
      pres1:= yourmin(leaf, top);
      pres2:= yourmax(leaf, top);
      nobranchpoint:= pres1 = pres2
    end else
    begin hl:= leaf↑.fathers[order - 1];
      if minof(hl) = maxof(hl) then
       begin delete(hl, top, order - 1,
              pres1, pres2, nobranchpoint);
         clear(hl); hp:= pres1;
         if nobranchpoint then hp↑.ub:= minus;
         pres1:= minof(pres1); pres2:= maxof(pres2);
         if nobranchpoint then
           if (pres1 = pres2) then clear(hp)
           else nobranchpoint:= false
       end else
       begin delete(leaf, hl, order - 1,
              pres1, pres2, nobranchpoint);
         if nobranchpoint then
           if (hl↑.ub = minus)
             and (yourmin(pres1, hl) = nil)
           then clear(hl)
           else nobranchpoint:= false
       end;
```

```
      if top↑.rank = order then
        if min = leaf then min:= pres1 else
        if max = leaf then max:= pres2
    end
end;
```

5.2.3. The procedure neighbour

The function neighbour has five parameters which are called by value. Their meaning is about equal to the meaning of the parameters in insert, however pres is replaced by the pair pmin and pmax.

neighbour may be called both for present and non-present leaves. This is justified by the fact that without expensive bit-manipulation on the positions it is impossible to decide whether the neighbour is the predecessor or the successor of the given argument.

pmin and pmax are the left and rightmost present leaf on leaf's side of the CS under consideration.

neighbour terminates without an inner call in the following cases:

(i) pmin = nil; now the neighbour resides on the other side of the tree
(ii) pmin = pmax = leaf; idem
(iii) leaf lies outside the interval pmin - pmax; in this case neighbour yields the nearest of the two in the usual sense without needing to investigate the inner structure of the tree.

The short-cut (iii) is unique to the procedure neighbour. If none of these situations occurs a recursive call is performed. This inner call is a top call if either the node hl at the center level in between leaf and top is not present (which in these circumstances is equivalent to non-active) or if leaf is the unique present descendent of hl; otherwise a bottom call is executed.

The initial call of neighbour reads:

```
neighbour(pt, root, mymin(pt, root), mymax(pt, root), h)
```

If called upon an empty tree or on the unique present leaf neighbour yields nil as its result; the driver takes care that these degenerate cases are looked after.

The text of neighbour is given below:

```
function neighbour(leaf, top, pmin, pmax : ptr;
                   order : ranktp) : ptr;

    var y, z, nb, hl : ptr; pos : 1..n;

begin pos:= leaf↑.position;
  if (pmin = nil)
     or ((pmin = pmax) and (pmin = leaf)) then
    if pos <= top↑.position
    then neighbour:= yourmin(leaf, top)
    else neighbour:= yourmax(leaf, top)
  else if pmin↑.position > pos then neighbour:= pmin
  else if pmax↑.position < pos then neighbour:= pmax
  else
  begin hl:= leaf↑.fathers[order - 1];
    y:= minof(hl); z:= maxof(hl);
    if ((y = z) and (y = leaf))
       or (hl↑.ub = undefined) then
    begin nb:= neighbour(hl, top,
     pmin↑.fathers[order -1], pmax↑.fathers[order -1],
                          order - 1);
     if hl↑.position < nb↑.position
     then neighbour:= minof(nb)
     else neighbour:= maxof(nb)
    end
    else neighbour:= neighbour(leaf, hl,
        mymin(leaf, hl), mymax(leaf, hl), order - 1)
  end
end;
```

5.2.4. Some remarks concerning the procedures

(1) The procedures insert, delete and neighbour all have the property that their innermost call is a bottom call, where we consider the complete tree to be a bottom tree as well. This observation is due to KAAS & ZIJLSTRA [8].

(2) At a node of rank d the father pointers of rank ≥ d are never inspected by the procedures. This results from the fact that their values are preserved in the stack of local variables of the envelopping recursive calls; in particular during a d-th order call the d-th rank father of all nodes within the CS under consideration (excluding top) is passed on in the parameter top. By omitting the space needed for these pointers one might reduce the storage requirements by a constant factor.

6. APPLICATIONS OF THE STRATIFIED TREE

In this section we discuss the topics of the representation of off-size priority queues (i.e. n not of the form 2^{2^h}), and the problem of manipulating a large number of equal size priority queues at once. The problem of reducing the storage requirements in the latter case without losing the $O(\log \log n)$ processing time is left unsolved.

6.1. Off-size priority queues

Let n be an arbitrary number and select h such that $2^{2^{h-1}} < n \leq 2^{2^h}$. Using the rank-h stratified tree to represent a priority queue of size n seems prohibitive since both its size and its initialization time are of order $h.2^{2^h}$ which might be about as large as $n^2.\log \log n$. To prevent this space explosion we can either eliminate bottom-subtrees or levels from the the rank-h tree.

6.1.1. Elimination of lower subtrees

In this approach all lower CS of rank h - 1 which have no leaves corresponding to numbers ≤ n are neither allocated nor initialized. In practice this means that the right-hand side of the tree is never used as long as $n < 2^{2^h}/2$. If the driver takes care about degeneracies the procedures of the preceding section work correctly without notifying that a large part of the tree is not physically present. The overhead in time and space is bounded by a constant factor 3.

6.1.2. Elimination of levels

Let $k = \lceil \log n \rceil$. A binary tree of height k is divided into a top tree of height $\lceil k/2 \rceil$ and bottom trees of height $\lfloor k/2 \rfloor$, which trees are divided themselves analogously. This leads to a canonical decomposition where certain rank-0 levels are not physically present. Once having pre-computed the function which attaches a rank to each level (which can be solved in time $O(\log n \,.\, \log \log n)$), the algorithms of the preceding section can be used without modifications. The needed overhead factor in time and space is bounded by a constant factor 2.

6.2. Representation of many priority queues

If one has to represent several priority queues it makes sense to separate the static and dynamical information in the nodes. The static information is about equal for each queue. More in particular, using an "address plus displacement" strategy, where the position of a node is used as its address, one has access to each node whose position is known. Since all nodes are accessed by father pointers from below, or by the downward pointers from the dynamical information, it is sufficient to have available a single pre-computed copy of the static information in a stratified tree. For each queue involved in the algorithm a $O(n)$ size block of memory, directly accessible by the position of a node, should be allocated for the dynamical information.

Using the above strategy we arrive at the $O(n \log \log n \cdot A(n))$-time, $O(n^2)$-space representation of a mergeable heap promised in the introduction. It is clear that the larger part of the space required is never used, and luckily there is a well-known trick which allows us to use this much space without initializing it [1]. Still it is a reasonable question whether some dynamical storage allocation mechanism can be designed which will cut down the storage requirement to a more reasonable level.

A direct approach should be to allocate storage for a node at the time this node is activated. This method, however, seems to be incorrect. One must be able to give the correct answer to questions of the following type: "Here I am considering a certain CS with root top and some leaves pres and leaf, where pres is present and leaf is not. Let hl be the ancestor of leaf at the center level. To decide whether hl is active, and, if so, where it is allocated." Inspection of the ancestor at center level of pres will yield the correct answer only if pres is actually the neighbour of leaf; this however is not guaranteed in our algorithm.

The same problem arises if one first tries to compute the neighbour of leaf. Consequently it seems necessary to reserve a predetermined location to store hl which can be accessed knowing the position of hl in the CS under consideration and having access at its root.

The following approach yields a representation of a mergeable heap in space $O(n.\sqrt{n})$ without disturbing the $O(\log \log n)$ processing time. Consider a rank d tree. As long as its left or right-hand side subtree contains not more than one present leaf, all necessary information can be stored at the root of the tree. If at a certain stage a second leaf at the same side must be inserted, the complete storage for the top tree is allocated as a consecutive segment, and a pointer at the root is made to refer to its initial address. In particular the nodes at the center level now have been given fixed addresses which are accessible via the root. The center-level nodes themselves are considered to be the roots of bottom trees of rank d - 1 which are treated analogously. In this manner a call of insert will allocate not more than $O(\sqrt{n})$ memory cells, whereas neighbour does not use extra memory and delete may return the space for a top tree if both sides of its enveloping CS have been exhausted except of a single leaf.

The initial address of the current relevant storage segment is given as a new parameter to the procedures whose value is passed on to an inner call, unless all envelopping calls are bottom calls.

The $O(n\sqrt{n})$ bound on the used memory for the mergeable heap algorithm is obtained by noting that at each intermediate stage the information contents are equal to one obtained by executing not more than n insert instructions.

We complete this section by noting that the storage requirements may be further reduced by replacing the binary division of the levels by an r-ary one for $r > 2$, which might result for each $\varepsilon > 0$ in an $O(n \log \log n \cdot A(n))$-time, $O(n^{1+\varepsilon})$-space representation of a mergeable heap.

7. REDUCIBILITIES AMONG SET-MANIPULATION PROBLEMS

The on-line manipulation of a priority queue, which is also known as the on-line insert-extract min problem, is one out of a multitude of set manipulation problems. Each of these problems has moreover a corresponding off-line variant. In the off-line variant the sequence of instructions is given in advance and the sequence of answers should be produced, the programmer being free to choose the order in which the answers are given.

Clearly, each on-line algorithm can be used to solve the off-line variant, but the converse does not hold.

In [3] we have investigated the reducibilities among the on-line and off-line versions of the insert-extract-min-, union-find- and insert-allmin problems. Here we say that a problem A can be reduced to a problem B if an alogirthm for B can be used to design an algorithm for A having the same order of complexity. If moreover A and B are both off-line problems it should be possible to translate an $O(n)$-size A problem on a $O(n)$-size structure into an $O(n)$-size B problem on a $O(n)$-size structure in time $O(n)$.

It has been shown by HOPCROFT, AHO & ULLMAN that the off-line insert-extract min problem is reducible to the on-line union-find problem [2]. The author has shown that the off-line union-find problem is equivalent to the off-line insert-allmin problem [3]. Together with the "natural" reduction of on-line insert-allmin to on-line insert-extractmin these reducibilities are represented in diagram 4 (the acronyms denoting the problems discussed).

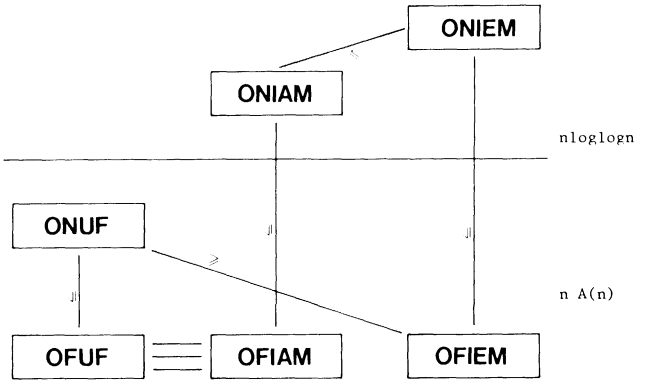


Diagram 4: Reducibilities among set-manipulation problems

ACKNOWLEDGEMENTS

I wish to thank J. Hopcroft for suggesting the problem and other useful suggestions. For the current state of the algorithms I am heavily indebted to R. Kaas and E. Zijlstra who, during the process of implementing the priority queue structure, discovered several hideous bugs in my earlier versions, and who contributed some nice and clever tricks; their use of a position field should be mentioned in particular. They also found the first recursive version of the delete procedure. Finally, I would like to thank R.E. Tarjan, Z. Galil and J. van Leeuwen for valuable ideas.

Author's addresses:

Mathematical Institute, University of Amseterdam, Roetersstraat 15, Amsterdam; or

Mathematical Centre, 2e Boerhaavestraat 49, Amsterdam.

REFERENCES

[1] AHO, A.V., J.E. HOPCROFT & J.D. ULLMAN, *The design and analysis of computer Algorithms*, Addison Wesley, Reading, Mass. (1974).

[2] AHO, A.V., J.E. HOPCROFT & J.D. ULLMAN, *On finding lowest common ancestors in trees*, Proc. 5-th ACM symp. Theory of Computing (1973), 253-265.

[3] EMDE BOAS, P. VAN, *An O(n log log n) On-Line Algorithm for the Insert-Extract Min Problem*, Rep. TR 74-221 Dept. of Comp. Sci., Cornell

Univ., Ithaca 14853, N.Y.

[4] EMDE BOAS, P. VAN, *The On-Line Insert-Extract Min Problem*, Rep. 75-04, Math. Institute, Univ. of Amsterdam.

[5] EVEN, S., M.R. GAREY & Y. PERL, *Efficient Generation of Optimal Prefix Code: Equiprobable Words Using Unequal Cost Letters*, J. Assoc. Comput. Mach. 22 (1975), 202-214.

[6] FISHER, M.J., *Efficiency of equivalence algorithms*, in: R.E. MILLER & J.W. THATCHER (eds.), *Complexity of Computer Computations*, Plenum Press, New York (1972), 158-168.

[7] HOPCROFT, J. & J.D. ULLMAN, *Set-merging Algorithms*, SIAM J. Comput. 2 (Dec. 1973), 294-303.

[8] KAAS, R. & E. ZIJLSTRA, *A PASCAL implementation of an efficient priority queue*, Rep. Math. Institute, Univ. of Amsterdam (to appear).

[9] TARJAN, R.E., *Efficiency of a good but non linear set union algorithm*, J. Assoc. Comput. Mach. 22 (1975), 215-224.

[10] TARJAN, R.E., *Edge disjoint spanning trees, dominators and depth first search*, Rep. CS-74-455 (Sept. 1974), Stanford.

[11] WIRTH, N., *The Programming Language PASCAL (revised report)*, in K. JENSEN & N. WIRTH *PASCAL User Manual and Report*, Lecture Notes in Computer Science 18, Springer, Berlin (1974).