MATHEMATICAL SYSTEMS THEORY 10, 99–127 (1977)

Design and Implementation of an Efficient Priority Queue*

P. VAN EMDE BOAS, R. KAAS, and E. ZIJLSTRA

Mathematical Centre, 2e Boerhaavestraat 49, Amsterdam,
and Mathematical Institute, University of Amsterdam,
Roeterstraat 15, Amsterdam, The Netherlands

ABSTRACT

We present a data structure, based upon a hierarchically decomposed tree, which enables us to manipulate on-line a priority queue whose priorities are selected from the interval $1, \ldots, n$ with a worst case processing time of $\mathcal{O}(\log \log n)$ per instruction. The structure can be used to obtain a mergeable heap whose time requirements are about as good. Full details are explained based upon an implementation of the structure in a PASCAL program contained in the paper.

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 $\mathcal{O}(n)$ instructions on an n-elements universe, where A is the functional inverse of a function with Ackerman-like order of growth [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 $\mathcal{O}(\log n)$ per instruction for a program of $\mathcal{O}(n)$ instructions on an n-elements universe which has to be executed on-line. Moreover, we should remember that an instruction repertoire which enables us to sort n reals by issuing $\mathcal{O}(n)$ instructions seems to need an $\mathcal{O}(\log n)$ processing time for the average instruction in doing so. For the decision tree model an order $n \log n$ lowerbound is known. However, if the universe is assumed to consist of the integers $1, \ldots, n$ only, this information-theoretical lowerbound on the complexity of sorting can be outwitted and it is known that n integers in the range $1, \ldots, n$ can be sorted in linear time.

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

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 and 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 $\mathcal{O}(\log n)$ per instruction. The authors introduce the name *mergeable heap* (resp. *priority queue*) for a structure supporting the above operations (excluding UNION).

In the present paper we describe a data structure which represents a priority queue with a worst case processing time of $\mathcal{O}(\log \log n)$ per instruction, on a Random Access Machine. The storage requirement is of the order $\mathcal{O}(n \log \log n)$ RAM-words. 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 $\mathcal{O}((\log \log n) \cdot A(n))$ and a space-requirement of order $\mathcal{O}(n^2)$. The possible improvements of the space requirements form a subject of continued research.

The mentioned $\mathcal{O}(\log n)$ processing time for manipulating priority queues and mergeable heaps sometimes forms the bottleneck in algorithms; in most cases, however, the priority values involved are assumed to be reals, and consequently our structure can not be used. In other examples like Tarjan's recent algorithm to compute dominators in directed graphs [10], the algorithm based on a mergeable heap has in the mean time been replaced [8] by one which does not use such a structure and which runs in inverse Ackermann time $A(n)$. It seems reasonable to assume that applications will be described soon; such applications were suggested by Hunt and Szymanski [7] and Even and Kariv [4].

Disadvantages of the structure are, beside the impossibility of manipulating real priority values, the large storage requirements of $\mathcal{O}(n)$ for each additional copy of a priority queue, and the theoretical objection that the time is measured using the uniform measure on a RAM; if charged according to the logarithmic measure the improvements of efficiency are lost because of the size of the involved arguments and values.

1.1. Structure of the Paper

Section 2 contains some notations and background information. In section 3 we present a "silly" implementation of a priority queue with an $\mathcal{O}(\log n)$ processing time per instruction. Reconsidering this implementation we indicate two possible ways to improve its efficiency to $\mathcal{O}(\log \log n)$.

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 $\mathcal{O}(\log \log n)$ worst and average case processing time per instruction. The algorithms for performing the elementary manipulations on stratified trees are explained in section 5. These algorithms are given explicitly by the PASCAL implementation of our priority queue in section 8. It is explained how the complete stratified tree is initialized using time $\mathcal{O}(n \log \log n)$. 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)	: Find whether $j \in S$
EXTRACT MIN	: Delete the least element from S
EXTRACT MAX	: Delete the largest element from S
PREDECESSOR (j)	: Compute the largest element in $S < j$
SUCCESSOR (j)	: Compute the least element in $S > j$
NEIGHBOUR (j)	: Compute the neighbour of j in S (see definition in section 3)
ALL MIN (j)	: Remove from S all elements $\leq j$.
ALL MAX (j)	: Remove from S all elements $\geq j$.

For arbitrary partitions $\Pi = \{A, B, \ldots\}$ of $\{1, \ldots, n\}$ we consider the following instructions:

FIND (i)	: Find 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.

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

In the next subsection we describe the data structures supporting subsets of these instructions, which will be discussed in the sequel of the paper.

2.2. Data structures

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., in 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 case processing time of $\mathcal{O}(\log \log n)$ per instruction (except for the last two instructions where the processing time is $\mathcal{O}(\log \log n)$ for each element removed).

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

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

For the famous union-find algorithm based on Tritter-trees, which is described in more details in section 6, Tarjan [9] has proved an upper and lower bound of order $n\ A(n)$.

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 internal nodes of the Tritter trees by complete priority queues. The resulting structure has an $\mathcal{O}(\log \log(n)A(n))$ average processing time per instruction; its space requirements are $\mathcal{O}(n^2)$. Details are given in section 6.

3. A "silly" Priority Queue with $\mathcal{O}$(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 n leaves of this tree will represent the numbers $1, \ldots, n$ 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 k-bit 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 $S \subseteq \{1, \ldots, n\}$ 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 figure 1.

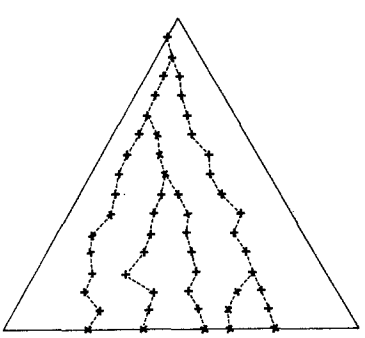


Figure 1. Example of a five-element set representation using mark bits.

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 $\leq j$ *do* EXTRACTMIN *od*.

ALLMAX (j) 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 analogously.

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.

Our final instruction is defined by the following algorithmic description:

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.

For the instruction NEIGHBOUR (j) it is less clear what this instruction intends to compute. Whereas the instructions PREDECESSOR (j) and SUCCESSOR (j) yield the "best" approximations of j unequal to j in the set S in terms of the usual order on the integers, the instruction NEIGHBOUR selects out of these two present leaves the "nearest" one in terms of the distances along the binary tree. It turns out that NEIGHBOUR is a more fundamental instruction than PREDECESSOR and SUCCESSOR. If we use an additional doubly linked list representation of the set S, a call of NEIGHBOUR, followed by at most one step in the list, will suffice to compute both the predecessor and the successor of a given element in S.

Conversely, to select the neighbour of j out of the pair consisting of its predecessor and successor requires either a subsequent investigation of the tree, or a few bit-manipulation instructions which are charged on a RAM according to the lengths of their arguments (the uniform cost is only used for additions and subtractions etc.).

3.2. Improvements of the Time Efficiency

It is clear from the above description that our "silly" structure supports the extended repertoire with an $\mathcal{O}(\log n)$ 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 accommodate a tree with branching orders increasing from the root to the leaves without disturbing the $\mathcal{O}(k)$ processing time, where k is the height of the tree. Using a tree with branching orders 2, 3, 4, . . . , k which contains $k! = n$ leaves, we can maintain a priority queue of size $\mathcal{O}(k!)$ in time $\mathcal{O}(k)$; the above set up yields therefore an $\mathcal{O}(\log n/\log\log n)$ 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 accommodate the branching orders 2, 4, 8, . . . , 2^k, which yields a priority queue of size $\mathcal{O}(2^{k^2/2})$, and the processing time is reduced to $\mathcal{O}(\sqrt{\log n})$.

Note that in both modifications the space requirements remain $\mathcal{O}(n)$, 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 $\{1, \ldots, n\}$ 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 $T(n) \leq T(\sqrt{n}) + 1$ which has a solution $T(n) \leq \mathcal{O}(\log\log n)$.

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 nodes.

If these traversals could be executed by means of a "binary search on the levels" strategy, the processing time is reduced from $\mathcal{O}(k)$ to $\mathcal{O}(\log k) = \mathcal{O}(\log \log n)$. A similar idea is involved in the efficient solution of a special case of the lowest common ancestor problem given by Aho, Hopcroft and 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

The hierarchical decomposition of a tree in top and bottom trees, as suggested by the divide and conquer strategy, can be performed perfectly only if the height of the tree k is a perfect power of two, i.e. $k = 2^h$ for some natural number h. Since in approximating the given size of the universe $\{1 \ldots n\}$ by numbers of the form 2^{2^h} the size of n might have to be almost squared (with the resulting prohibitive time- and space requirements) we settle for a less perfect decomposition where $n = 2^k$ and where the characteristic processing time is of order $h = \lceil \log \log n \rceil$. Replacing n by the next largest power of two, the overhead for the initializing time and storage requirements is less than a constant factor 2.

The hierarchical decomposition of our tree is based on a precomputed *rank function*. If $k = 2^h$ a function RANK is defined as follows:

For $j > 0$ we define RANK (j) to be the number d such that $2^d | j$ and $2^{d+1} \nmid j$. For example RANK (12) = 2 and RANK (7) = 0. By convention we take RANK (0) $= h+1$.

Note that if RANK $(j) = d$ and $j \geq 2^d$ we have RANK $(j) <$ RANK $(j+2^d)$ and RANK $(j) <$ RANK $(j-2^d)$; moreover RANK $(j+2^d) \neq$ RANK $(j+2^d)$. Inspired by these properties we define for less perfect tree heights k the concept of a rank function as follows:

A function $f : \{i, \ldots, j\} \to N$, with $i < j$ is called a *rank function* provided:

(i) $f(i) \neq f(j)$

(ii) $f(m) < \min(f(i), f(j))$ for $i < m < j$

(iii) either $j = i+1$ or otherwise there exists a unique element m such that $i < m < j$, $f(m) = \min(f(i), f(j)) - 1$ and the restrictions $f|\{i, \ldots, m\}$ and $f|\{m, \ldots, j\}$ are rank functions.

Note that if f: $\{i, \ldots, j\} \to \{0, \ldots, g+1\}$ is a rank function then clearly $j - i + 1 \leq 2^g$.

This definition of a rank function allows both "optimal" rank functions and useless ones. A rank function will be called optimal if its highest value is the lowest possible one over all rank functions over the same interval. The data structure and algorithms we are going to describe behave well for any rank function. However, the running time of the algorithms depends linearly on the highest value of the rank function so we do better to restrict ourselves to optimal rank functions.

For example if $\{i, \ldots, j\} = \{0, \ldots, 8\}$ then the function described by the sequence (4 0 1 0 2 0 1 0 3) is an optimal rank function, whereas the sequence (8 6 5 4 3 2 1 0 7) represents an expensive one. The algorithms based upon this final rank function will turn out to resemble those for the "silly" structure from section 3 on a tree of height k.

An optimal rank function RANK: $\{0, \ldots, k\} \to \{0, \ldots, h+1\}$ may be defined inductively as follows: RANK $(0) = h+1$; RANK $(k) = h$; next as long as there exist intervals $\{i, \ldots, j\}$ such that RANK (i) and RANK (j) are already defined and RANK (m) is not yet defined for $i < m < j$ one defines RANK $([(i+j)/2])$ = min (RANK (i), RANK $(j)) - 1$. For an explicit algorithm the reader is referred to the procedures **fillranks** and **start** in the program in section 8, where it is shown that this function can be precomputed on a RAM in time $0(k)$ (without use of division by two).

In the sequel we consider a fixed binary tree T with n leaves and k levels; RANK denotes a fixed optimal rank function mapping $\{0, \ldots, k\}$ onto $\{0, \ldots, h+1\}$.

The *level* of a node v in T is the length of the path from v to its descendent leaves; 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 $\{1, \ldots, n\}$ represented by this leaf. The *position* of an internal node v equals the position of the rightmost descendent leaf of its lefthand 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 rank d* of T is a binary subtree having as root a node of rank $\geq d$, as leaves all descendents of the root at the nearest level of rank $\geq d$ (and consequently internal nodes of rank $< d$). From the definition of a rank function it follows that for a non-trivial *CS* of rank d either the rank of the top or the ranks of the leaves equals d. In the first case the tree is called a *bottom tree* whereas it is called a *top tree* otherwise. A *CS* of rank d either is a trivial three-element tree, or it is decomposed in a top and bottom *CS*'s of rank $d-1$. Canonical subtrees of rank 0 are always trivial. The subtree of a *CS* consisting of its root with all its left (right) hand side descendents is called a *left* (*right*) *canonical subtree.*

To any node v of T we associate the following subtrees which are called the canonical subtrees of v. Let d be the rank of 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. In general there may exist more than one *CS* having v as root (leaf); $LC(v)$ ($UC(v)$) is the largest one.

The left (right) canonical subtree of $LC(v)$ is denoted by $LLC(v)$ ($RLC(v)$). For internal nodes v we say that $LC(v)$ and the left- or right hand canonical subtree of $UC(v)$ containing v together form the *reach* of v, denoted by $R(v)$. 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 figure 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 $\geq j$. Clearly FATHER (v, h) equals the root t of T,

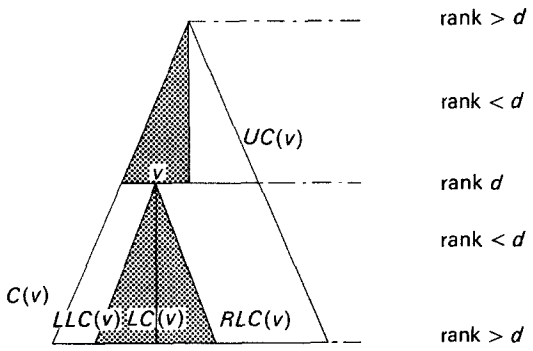


Figure 2. The canonical subtrees of v. $R(v)$ is the shaded area.

whereas FATHER $(v, 0)$ is the "real" father of v in T(provided $v \neq t$). At each node we have an array of h pointers **father** $[0: h-1]$ such that **father** $[i]$ yields FATHER (v, i). Since FATHER (v, h) always yields the root of the tree this element doesn't need to be included. These pointers enable to climb along a path in the tree to a predetermined level in $\mathcal{O}(h)$ steps. Moreover, given the root of a non trivial CS U and one of its leaves, we can proceed in a single step to the center of the smallest reach containing the two.

Actually it will be seen that within our recursive algorithms in section 8 for a node of rank d a father pointer to a father of rank $\geq d$ is never asked for! Consequently these pointers might be omitted from the structure; the resulting structure however should consist of rank-dependent nodes, whereas the gain in storage efficiency is at most a constant factor; therefore this improvement is not implemented.

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

4.2. Dynamical information

The dynamical information at internal nodes is stored using four pointers **lmin**, **lmax**, **rmin** and **rmax** 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 $S \subset \{1, \ldots, n\}$ 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 meaningful information. The values of these fields of a non-active internal node are: **lmin** = **nil**, **lmax** = **nil**, **rmin** = **nil**, **rmax** = **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:

lmin: points to the leftmost present leaf of $LLC(v)$ if such node exists; otherwise **lmin** = **nil**.

lmax: idem for the rightmost present leaf of $LLC(v)$

rmin: idem for the leftmost present leaf of $RLC(v)$

rmax: idem for the rightmost present leaf of $RLC(v)$

ub = **plus** if there occurs a branchpoint 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 iff 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 iff it is present; the root is active iff the set is non-empty.

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 left or right hand subtree 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.

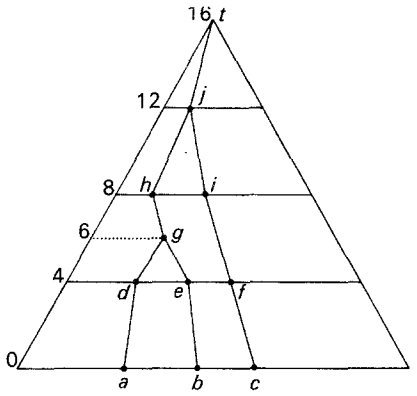
The set $S \subset \{1, \ldots, n\}$ is represented as follows. First the leaves corresponding to the elements of S and all their ancestors are declared to be present. Next we compute using the properness condition which present nodes become active. Finally the dynamical fields of all active and non-active nodes are given their proper values. The resulting information content is called the representation of 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 $S = \varnothing$, 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 figure 3 (omitting the evident doubly linked list data). The symbol $\sim$ denotes **nil** resp. **undefined**.

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 2 can be executed such that



	level	rank	lmin	lmax	rmin	rmax	ub
t	16	4	*a*	*c*	~	~	−
j	12	2	*h*	*h*	*i*	*i*	−
h	8	3	~	~	*a*	*b*	+
i	8	3	~	~	*c*	*c*	+
g	6	1	*d*	*d*	*e*	*e*	−
d	4	2	*a*	*a*	~	~	+
e	4	2	~	~	*b*	*b*	+
f	4	2	~	~	~	~	~

Figure 3. Example of a proper information content. *f* is not active.

(i) a processing time of $\mathcal{O}(h) = \mathcal{O}(\log \log n)$ 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; they are described explicitly in the program in section 8.

5.1. Initialization

Initialization takes place during a single tree-transversal in mixed pre- and in-order. When a node is processed its father-pointers and its position and in case of internal nodes its rank are stored in the appropriate fields. 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 than the level of its father
(iii) the position of an internal node equals the number of leaves encountered after its lefthand subtree has been traversed in in-order. This number is accessible by means of a global variable **count**. (It is this evaluation of the position where in-order traversal is used).

(iv) The rank of a node depends only on the level and can be stored using a table **ranks** of size $k+1$.

Once having precomputed the ranks using the local procedure **fillranks**, the above relations show how the static structure is initialized without having "illegitimate" instructions like multiplications and bit-manipulations, in time $\mathcal{O}(\log \log n)$ per node processed. Since there exist $2.n-1$ nodes this shows that the initialization takes time $\mathcal{O}(n \log \log n)$. The space requirements $\mathcal{O}(n \log \log n)$ follow since the space needed for each node is $\mathcal{O}(\log \log n)$. The space requirements are expressed in terms of RAM words, which are used to store addresses up to length $\log n$. Hence the storage requirements in terms of "bits" become $\mathcal{O}(n \log n \log \log n)$.

Since the level is not part of the static information it may be represented as a local variable of the recursive initialization routine.

5.2. Operations

The extended instruction repertoire can be expressed (disregarding the doubly linked list operations) in terms of three primitive operations **ins**, **del** and **neighb**. 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 on 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, or any other trivial *CS* 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 $h = \lceil \log \log n \rceil$. The procedures **ins**, **del**, and **neighb** 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.

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 descendent 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 descendent v of t is a left-hand descendent if 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 descendent.

The procedures **ins**, **del** and **neighb** 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**)		analogous for the max-field.
yourfields (**v**, **t**), **yourmin** (**v**, **t**) and **yourmax** (**v**, **t**)	yield	the analogous values of the fields at t in the other direction.

minof (t) yields the leftmost defined value of the four pointer fields at *t* if *t* is active, and **nil** otherwise.

maxof (t) yields the rightmost value analogously.

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 ins

ins 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 node 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.)

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

function **ins (leaf, top, pres: ptr; nobranchpoint: boolean; order: integer): 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 (actually the minimal one) of the *CS* on which the procedure is called, belonging to the same left or right hand canonical subtree as **leaf**.

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.

A call of **ins** 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

ins (leaf, hl, mymin (leaf, hl), true, order-1)

or the top-call

ins (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

ins (leaf, hl, mymin (leaf, hl), mymin (leaf, hl) = mymax (leaf, hl), order-1)

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

ins (hl, top, hp, nobranchpoint, order-1)
is executed after having set **nobranchpoint** := (**hp↑.ub** = **minus**) and having activated the fields at **hl** and (if needed so) the **ub** field at **hp**. This call yields as a result the neighbour of **hl** in the top-tree in **nb**, and depending on 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. In order to have correctness of this assertion for the complete tree as well, the complete tree has to be considered a bottom-*CS*, which is realized by numbering the levels from the leaves to the top.

The initial call of **ins** reads:
ins (ej, root, mymin (ej, root), mymin (ej, root) = mymax (ej, root), h) where it is assumed that **ej** is not a present leaf. This condition is enforced by the calling routine **insert**.

For the complete PASCAL text of ins we refer to section 8. The four possible situations are illustrated in figure 4.

5.2.2. The procedure del

The procedure **del** yields no value. It has six parameters, the first three of which are called by value, the others being called by reference. The procedure heading reads:

procedure **del (leaf, top: ptr; order: integer;** ***var*** **pres1, pres2: 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 **del** the following meaning:

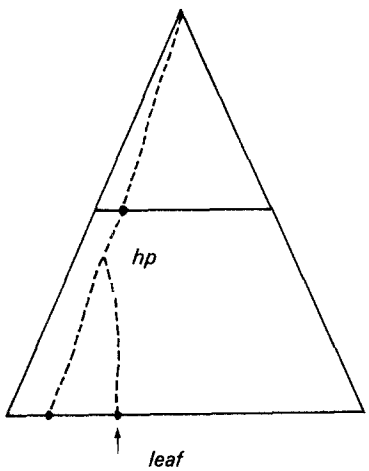
pres1, pres2: present leaves in the *CS* considered, one of them being the neighbour of **leaf** (see explanation below)
nobranchpoint: true iff there occurs no branchpoint on the path from **top** to **pres1**.

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

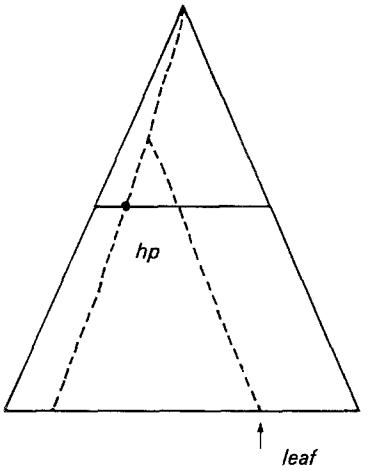
Proceeding downwards from the other son of the lowest branchpoint as near as possible we arrive at the neighbour; if we however select always the remotest present node, we arrive at a node which might be called the *extreme* of **leaf** in the tree. The extreme, in terms of the distance along the binary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

After a call of **del**, **pres1** and **pres2** are the neighbour and the extreme of **leaf** ordered according to their positions (i.e. **pres1↑.pos** ≤ **pres2↑.pos**).

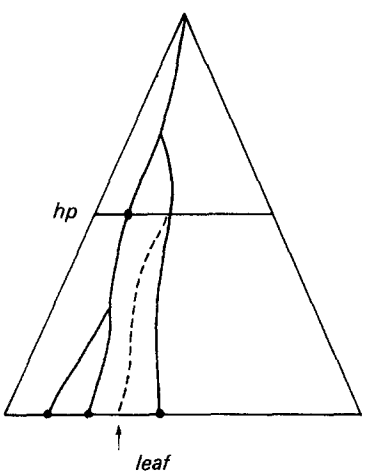
The procedure **del** terminates without inner call if the lowest branchpoint equals **top**; at this time **pres1** and **pres2** are initialized with the values



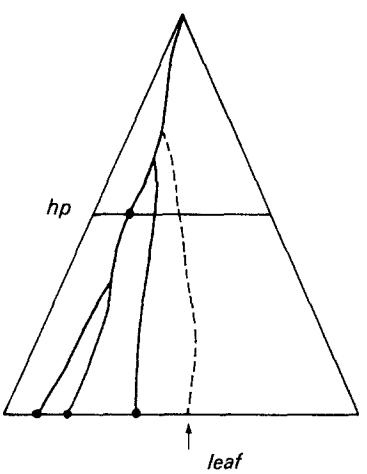
hp inactive; $hp = hl$
bottom call



hp inactive; $hp \neq hl$
top call



hp active; *hl* active
bottom call



hp active; *hl* inactive
top call

Figure 4. The four possible inner call situations for **ins**.

yourmin(leaf, top) and **yourmax(leaf, top)** and **nobranchpoint** is made true if these two values are equal.

Otherwise **hl** becomes the node halfway inbetween **leaf** and **root** and depending on whether **leaf** is the unique present son in its lower canonical half tree a recursive call of **del** is initialized on *UC*(**hl**) or *LC*(**hl**). After completion of this inner call the current call of **del** is completed as follows:

The values **pres1** and **pres2** are updated depending on whether the inner call just terminated was a top or a bottom call. If the last call was a top call then **pres1: = minof(pres1); pres2: = maxof(pres2)** and if **nobranchpoint** still is true their equality is tested again to decide whether **nobranchpoint** should remain true; if so the node formerly pointed at by **pres1** is inactivated.

If the last call was a bottom call and if **nobranchpoint** still is true the ub field at the former top and the pointers away from **pres1** at this node are inspected to decide whether there occurs a branchpoint at or above this node; if not the former top is inactivated.

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

The initial call to **del** reads:

del (ej, root, h, p1, p2, nobrpt);

For the complete text of **del** see section 8. The calling routine **delete** makes sure that **ej** is a present leaf. For a situation leading to the inactivation of a present node see figure 5.

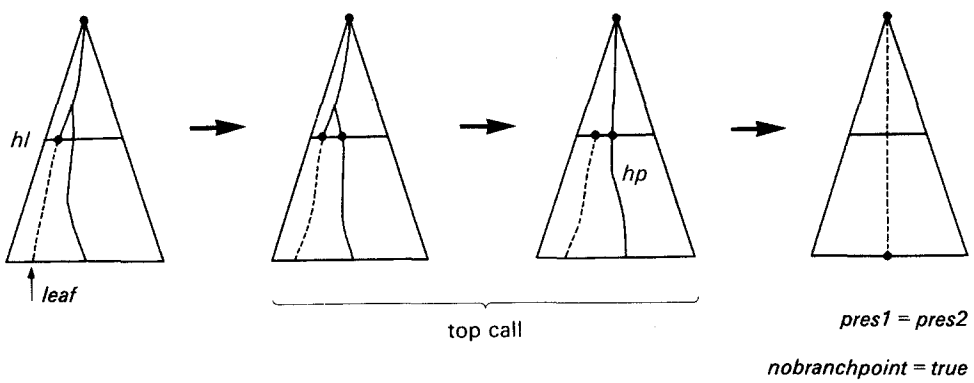


Figure 5. **del** inactivates a present node.

5.2.3. The procedure neighb

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

The procedure **neighb** may be called both for present and nonpresent 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.

neighb 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 **neighb** yields the nearest of the two in the usual sense without needing to investigate the inner structure of the tree. (See figure 6).

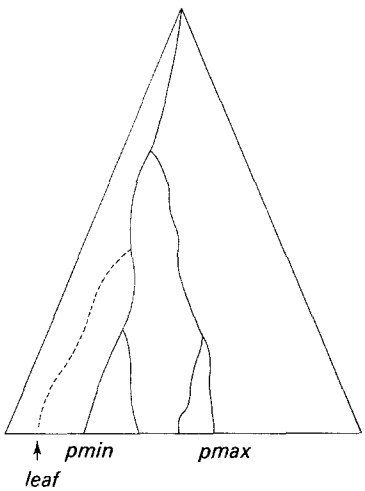


Figure 6. The short-cut in **neighb.**

The short-cut (iii) is unique to the procedure **neighb**. 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. A bottom call yields the neighbour as result directly; in case of a top call we obtain the ancestor at center level of the neighbour, and the neighbour is obtained as the nearest present descendant leaf of this intermediate result.

The initial call of **neighb** reads:

neighb (pt, root, mymin(pt, root), mymax(pt, root), h)

If called upon an empty tree or on the unique present leaf **neighb** yields **nil** as its result.

6. A Time Efficient Mergeable Heap

The well known efficient union-find algorithm uses a representation of sets by means of the so-called Tritter 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 (path compression).

It has been established only recently how efficient the above algorithm is. Whereas its average processing time has been estimated originally as $\mathcal{O}(\log \log n)$ (Fischer [5]) and $\mathcal{O}(\log^* n)$ (Hopcroft and Ullman [6] and independently Paterson (unpublished)), a final upper and lowerbound order $A(n)$ 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 Ackermann-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).

To obtain a mergeable heap we replace 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; see figure 7.

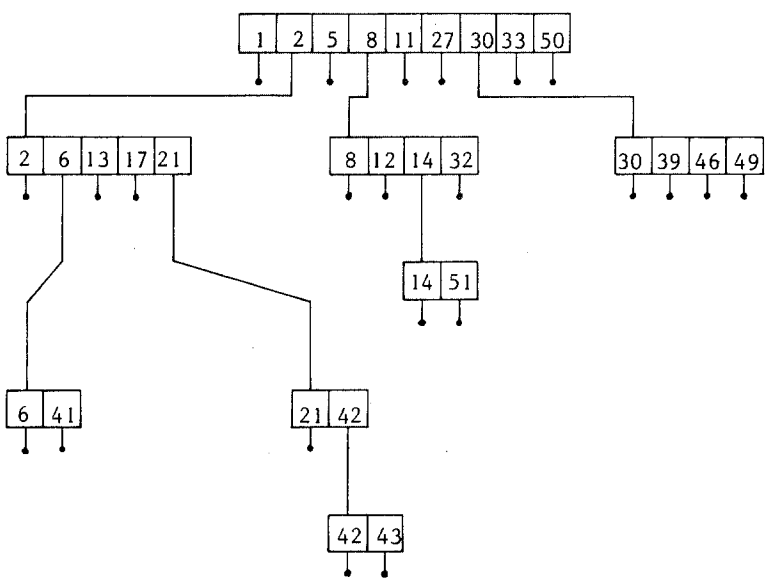


Figure 7. A Tritter tree with priority queues at its internal nodes.

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 $\mathcal{O}(\log n)$ the proposed representation of a mergeable heap should be considered inefficient, since there are $\mathcal{O}(\log n)$ 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 above applications we have to deal with a large number ($\mathcal{O}(n)$) of distinct priority queues simultaneously.

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 $\mathcal{O}(n)$ 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 $\mathcal{O}(n \log \log n \cdot A(n))$-time, $\mathcal{O}(n^2)$-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. Decide whether **hl** is active, and, if so, where it is allocated." If **pres** is actually the neighbour of **leaf** we know that either **hl** equals the ancestor of **pres** at center level, or **hl** is not active and consequently not allocated. It is however not certain that **pres** equals the neighbour of **leaf**.

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

$\mathcal{O}(n\sqrt{n})$ without disturbing the $\mathcal{O}(\log \log n)$ 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 **ins** will allocate not more than $\mathcal{O}(\sqrt{n})$ memory cells, whereas **neighb** does not use extra memory and **del** 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 $\mathcal{O}(n\sqrt{n})$ 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 **ins** 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 $\mathcal{O}(n \log \log n \cdot A(n))$-time, $\mathcal{O}(n^{1+\varepsilon})$-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 this section we investigate 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 algorithm 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 $\mathcal{O}(n)$-size A problem on a $\mathcal{O}(n)$-size structure into an $\mathcal{O}(n)$-size B problem on a $\mathcal{O}(n)$-size structure in time $\mathcal{O}(n)$.

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 [1]. Below we indicate how the off-line union-find problem and the off-line insert-allmin problem are reduced to each other. Together with the "natural" reduction of on-line insert-allmin to online insert-extractmin these reducibilities are represented in figure 8 (the acronyms denoting the problems discussed).

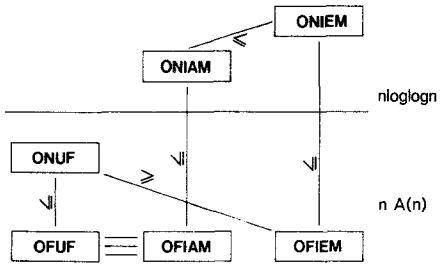


Figure 8. Reducibilities among set-manipulation problems.

7.1. The reduction OFIAM ≤ OFUF

The reduction below is a modification of the reduction OFIEM ≤ ONUF in [1].

Consider a sequence of insert and allmin instructions. For each instruction insert (j) we construct an object x_j. For each instruction allmin (i) we construct a barrier b_i. The value i is called the height of b_i.

Consider the sequence of objects and barriers in their original order. Clearly object x_j will be removed by the allmin instruction corresponding to the first following barrier with height $\geq j$.

We gather all objects x_k between two consecutive barriers b_j and b_i in a set $S(b_i)$ having the above barrier as name. Clearly the union instructions needed to build these sets can be generated off-line in linear time.

Next we generate for $j = 1, 2, \ldots, n$ the instruction find(j), followed by instructions to remove all barriers having height j. A barrier b_j is removed by uniting the set having this barrier as a name with the set named by the nearest remaining barrier b_k above b_j. Note that it is illegal to include "computed arguments" in a sequence of instructions which are to be executed off-line. However by keeping the active barriers in a doubly linked list we can compute b_k during the process of translating the insert-allmin instructions into union-find instructions. Clearly the above translation can be executed in time $\mathcal{O}(n)$.

It is left to the reader to verify that the result of executing a find(i) instruction in the resulting sequence yields the barrier corresponding to the allmin instruction which removes i.

7.2. The reduction OFUF ≤ OFIAM

Consider a sequence of union and find instructions. We first build a binary tree by executing the subsequence of all union instructions as follows: For each union instruction a new vertex is created having the two sets from which the union is formed as sons. We store, moreover, at each vertex the name which is given to the union, and the time (number of the instruction) at which the union is formed. Without loss of generality we may assume that the final result is a single tree (otherwise add a number of extra union instructions).

Clearly construction of this tree takes time $\mathcal{O}(n)$.

By running through the sequence of instructions another time we can organize for each element j a linear list of all the times at which instruction find(j) must be executed.

Next we will traverse the tree in post order (i.e. first visit the subtrees and next visit the root). While traversing the tree we have available a bucket containing a set of find instructions.

Whenever we visit a leaf of the tree (i.e. a vertex which represents a singleton), we throw into the bucket all the find instructions corresponding to this leaf (i.e. all instructions find(i) where $\{i\}$ is the set represented by the leaf).

If we process an arbitrary vertex v we remember that the set which is represented by v contains the elements represented by the leaves of the subtree with root v, during the period starting with the formation of the union (which time T is stored at vertex v) and the time T' at which this set is united into a larger set (which time is stored at the father of v). Hence we should throw out of the bucket all find instructions having times satisfying $T \leq t < T'$. Since the two sons of v already have been processed, find's with time $< T$ are no longer in the bucket and consequently we should remove from the bucket all find's with time $< T'$. This is an allmin instruction. It is clear that if find(i) is removed at vertex v then find(i) is answered by the name of the set stored at v.

Up to this point everything that we have described can be executed in time $\mathcal{O}(n)$ for the complete tree, but this allmin instruction makes the algorithm nonlinear (assuming that no linear algorithm is known for the insert-allmin problem).

It is clear that, after having constructed the tree, the complete sequence of insert and allmin instructions which is going to be executed on the bucket can be computed by traversing the tree once in post order. This shows that the off-line union-find problem is reducible to the off-line insert-allmin problem.

8. The Program

The program follows exactly the syntax of Standard PASCAL [11], with the exception that (* and *) are used to begin and end a comment.

Together with a driver here omitted it has worked correctly to maintain a priority queue of size 2048 in (in octal) 150000 words of central memory.

No attempts have been made to gain efficiency at the price of transparency by expanding inessential procedures or replacing recursion by iteration, etc. On a reasonable PASCAL implementation such modifications will reduce the running time by at most a constant factor.

```
program priorityqueue(input, output);
                                  (*        k   h-1              h*)
const n = 512; k = 9; h = 4; (* n = 2  ; 2       < k < = 2    *)
type ptr = ↑node; fieldptr = ↑subtree;
  subtree = record min, max: ptr end;
```

```
  node = record position: 1 . . n; fathers: array [0 . . h] of ptr;
    case (* internal: *) boolean of
      true: (ub: (minus, undef, plus);
        left, right: fieldptr;
        rank: 0 . . h);
      false: (present: boolean; pred, succ: ptr);
      end;
var root: ptr; cardinality: 0 . . n; elements: array[1 . . n] of ptr;
function mymin(leaf, top: ptr): ptr;
begin if leaf↑.position < = top↑.position then
  mymin:= top↑.left↑.min else mymin:= top↑.right↑.min
end;
function mymax(leaf, top: ptr): ptr;
begin if leaf↑.position < = top↑.position then
  mymax:= top↑.left↑.max else mymax:= top↑.right↑.max
end;
function myfields(leaf, top: ptr): fieldptr;
begin if leaf↑.position < = top↑.position then
  myfields:= top↑.left else myfields:= top↑.right
end;
function yourmin(leaf, top: ptr): ptr;
begin if leaf↑.position > top↑.position then
  yourmin:= top↑.left↑.min else yourmin:= top↑.right↑.min
end;
function yourmax(leaf, top: ptr): ptr;
begin if leaf↑.position > top↑.position then
  yourmax:= top↑.left↑.max else yourmax:= top↑.right↑.max
end;
function yourfields(leaf, top: ptr): fieldptr;
begin if leaf↑.position > top↑.position then
  yourfields:= top↑.left else yourfields:= top↑.right
end;
function minof(top: ptr): ptr;
  var temp: ptr;
begin temp:= top↑.left↑.min; if temp = nil then
  minof:= top↑.right↑.min else minof:= temp
end;
function maxof(top: ptr): ptr;
  var temp: ptr;
begin temp:= top↑.right↑.max; if temp = nil then
  maxof:= top↑.left↑.max else maxof:= temp
end;
procedure intolist(low, med, upp: ptr);
begin if low < > nil then low↑.succ:= med;
  if upp < > nil then upp ↑.pred:= med;
  med↑.pred:= low; med↑.succ:= upp;
  med↑.present:= true; cardinality:= succ(cardinality)
end;
```

```
procedure fromlist(low, med, upp: ptr);
begin if low <> nil then low↑.succ:= upp;
  if upp <> nil then upp↑.pred:= low;
  med↑.pred:= nil; med↑.succ:= nil;
  med↑.present:= false; cardinality:= pred(cardinality)
end;
function min: integer;
  var pt: ptr;
begin pt: = minof(root); if pt = nil then min: = 0 else
  min:= pt↑.position
end;
function max: integer;
  var pt: ptr;
begin pt: = maxof(root): if pt = nil then max: = 0 else
  max:= pt↑.position
end;
function member(j: integer): boolean;
begin if (j > 0) and (j < = n) then member: = elements[j]↑.present
  else member:= false
end;
procedure clear(v: ptr);
begin with v↑ do
  begin left↑.min:= nil; left↑.max:= nil;
    right↑:= left↑; ub:= undef
  end
end;
procedure start;
  var i, count, lastdiv2: integer; even: boolean;
    ranks, div2: array [0 . . k] of integer:
procedure fillranks(1, u, r: integer);
  var m: integer;
begin m: = 1+div2[u-1]; ranks[m]: = r; if m > 1+1 then
  begin fillranks(l, m, r-1); fillranks(m, u, r-1) end
end;
procedure initialize(fath: ptr; level: integer);
  var i, rnk: integer; v: ptr;
begin if level > 0 then new(v, true) else new(v, false);
  with v↑ do
  begin rnk: = fath↑.rank;
    for i: = 0 to rnk do fathers[i]: = fath;
    for i: = rnk+1 to h do fathers[i]: = fath↑.fathers[i];
    if level = 0 then
    begin present: = false; succ: = nil; pred: = nil;
      count: = count + 1; position: = count; elements [position]: = v
    end else
    begin new(left); new(right); clear(v); rank: = ranks[level];
      initialize (v, level-1); position: = count;
```

```
        initialize (v, level - 1)
      end
    end
  end;
begin div2[0]: = 0; lastdiv2: = 0; even: = true;
  for i: = 1 to k do
  begin even: = not even;
    if even then lastdiv2: = lastdiv2 + 1; div2[i]: = lastdiv2
  end;
  ranks[k]: = h; fillranks(0, k, h - 1); cardinality: = 0; count: = 0;
  new(root, true): with root↑ do
  begin rank: = h; new(left); new(right);
    clear(root); for i: = 0 to h do fathers[i]: = nil
  end;
  initialize(root, k - 1); root↑.position: = count;
  initialize(root, k - 1);
end;
procedure insert(j: integer);
  var ej, nb, min: ptr;
  function ins(leaf, top, pres: ptr; nobranchpoint: boolean;
        order: integer): 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
      ins:= top↑.right↑.min else ins:= 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:= ins(hl, top, hp, true, order - 1);
            ins:= pres
          end else
          begin fptr:= myfields(pres, hp); with fptr↑ do
            begin min:= pres; max:= pres end;
            hp↑.ub:= minus;
            ins:= ins(leaf, hl, mymin(leaf, hl), true, order - 1)
          end
        else if hl↑.ub < > undef then
        ins:= ins(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:= ins(hl, top, hp, nobranchpoint, order−1);
          if hl↑.position <= nb↑.position then
          ins:= minof(nb) else ins:= 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;
begin if not member(j) then
  begin ej: = elements[j]; min: = mymin(ej, root);
    nb: = ins(ej, root, min, min = mymax(ej, root), h);
    if nb = nil then intolist(nil, ej, nil) else
    if nb↑.position < j then intolist(nb, ej, nb↑.succ) else
    intolist(nb↑.pred, ej, nb)
  end
end;
procedure delete(j: integer);
    var ej, p1, p2: ptr; nobrpt: boolean;
    procedure del(leaf, top: ptr; order: integer; 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 del(hl, top, order−1, pres1, pres2, nobranchpoint);
          clear(hl); hp:= pres1;
          pres1:= minof(pres1); pres2:= maxof(pres2);
          if nobranchpoint then
            if (pres1= pres2) then clear(hp)
            else begin nobranchpoint:= false; hp↑.ub:= minus end
        end else
        begin del(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;
begin if member(j) then
  begin ej:= elements[j]; del(ej, root, h, p1, p2, nobrpt);
    fromlist(ej↑.pred, ej, ej↑.succ)
  end
end;
procedure extractmin; begin delete(min) end;
procedure extractmax; begin delete(max) end;
procedure allmin(j: integer);
  var m: integer;
begin m: = min; while (m > 0 ) and m < = j) do
  begin delete(m); m: = min end
end;
procedure allmax(j: integer);
  var m: integer;
begin m: = max; if j < 1 then j: = 1; while m > = j do
  begin delete(m); m: = max end
end;
function neighbour(j: integer): integer;
  var pt: ptr;
  function neighb(leaf, top, pmin, pmax: ptr; order: integer): 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 neighb:= yourmin(leaf, top)
      else neighb:= yourmax(leaf, top)
    else if pmin↑.position > pos then neighb:= pmin
    else if pmax↑.position < pos then neighb:= pmax
    else
    begin hl:= leaf↑.fathers[order - 1]; y:= minof(hl); z:= maxof(hl);
      if ((y = z) and (y = leaf)) or (hl↑.ub = undef) then
      begin nb:= neighb(hl, top, pmin↑.fathers[order - 1],
          pmax↑.fathers[order - 1], order - 1);
        if hl↑.position < nb↑.position then neighb:= minof(nb)
        else neighb:= maxof(nb)
      end
      else neighb:= neighb(leaf, hl, mymin(leaf, hl),
          mymax(leaf, hl), order - 1)
    end
  end;
begin pt: = elements[j];
  pt: = neighb(pt, root, mymin(pt, root), mymax(pt, root), h);
  if pt = nil then neighbour: = 0 else neighbour: = pt↑.position
```

```
end;
function predecessor(j: integer): integer;
  var pt: ptr; nb: integer;
begin if not member(j) then nb: = neighbour(j) else nb: = j;
  if nb < j then predecessor:= nb else
  begin pt: = elements[nb]↑.pred;
    if pt = nil then predecessor: = 0 else predecessor: = pt↑.position
  end
end;
function successor(j: integer): integer;
  var pt: ptr; nb: integer;
begin if not member(j) then nb: = neighbour(j) else nb: = j;
  if nb = 0 then successor: = n+1 else
  if nb > j then successor: = nb else
  begin pt: = elements[nb]↑.succ;
    if pt = nil then successor: = n+1 else successor: = pt↑.position
  end
end;
procedure usersprogram;
begin (* to be supplied by the user *) end;
begin start; usersprogram end.
```

ACKNOWLEDGEMENTS

The efficient priority queue was designed by the first author during a three months visit to the Cornell dept. of Computer science, supported by ZWO grant CR 62/50. Thanks are due to J. Hopcroft for suggesting the problem and other ideas and to Z. Galil for checking the original argumentations and other useful suggestions. The mergeable heap was obtained on base of suggestions by R. E. Tarjan.

The PASCAL implementation was designed by the remaining authors at the University of Amsterdam. During the process of implementing the original structure was improved and all bugs in the original desk-written algorithms in [3] were eliminated. Our gratitude goes to the Institute for Appl. Math. of the University of Amsterdam, and to T. J. Dekker in particular for offering the technical facilities to complete our work.

REFERENCES

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

[2] AHO, A. V., J. E. HOPCROFT and 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. Dec. 1974.

[4] EVEN, S. and O. KARIV, *Oral Commun.*, Berkeley, October 1975.

[5] FISCHER, M. J., *Efficiency of equivalence algorithms*, in: R. E. Miller and J. W. THATCHER (eds.), *Complexity of Computer Computations*, Plenum Press, New York (1972), 158–168.

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

[7] HUNT, J. W. and T. G. SZYMANSKI, *A fast algorithm for computing longest common subsequences.* Manuscript. Dept. Electr. Eng. Princeton Univ. Princeton, N.J. 08540. Oct. 1975.

[8] TARJAN, R. E., *Applications of path compression on balanced trees.* Manuscript. Stanford Oct. 75. (Submitted to *JACM*).

[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 and N. Wirth *PASCAL User Manual and Report*, Lecture Notes in Computer Science 18, Springer, Berlin (1974).

Received December 29, 1975
and in revised form June 7, 1976