PRESERVING ORDER IN A FOREST IN LESS THAN LOGARITHMIC TIME AND LINEAR SPACE

P. van EMDE BOAS
ITW/VPW University of Amsterdam, Amsterdam

Received 9 March 1977

Set-manipulation, trees, analysis of algorithms

1. Introduction

Consider a fixed universe $U = \{1, ..., u\}$. We consider data-structures supporting the complete repertoire of single-set manipulation on subsets of U. The instructions in this repertoire are the following:

Insert(x):	put x into S
Delete(x):	remove x from S
Member(x):	does x belong to S?
Empty:	is S empty?
Min:	gives the least element in S,
Max:	gives the largest element in S,
Predecessor(x):	gives the largest element $<x$ in S,
Successor(x):	gives the least element $>x$ in S.

By combination of these basic instructions we obtain moreover the instructions *Extractmin*, *Extractmax* (remove the least resp. the largest element from S) and the iterated instructions *Allmin*(i) and *Allmax*(i) (remove from S all elements $\leq i$ resp. $\geq i$).

Traditionally a *priority queue* is a data structure supporting the instructions *Insert*, *Delete*, *Member*, *Min* and *Extractmin*, but in the present note we will use this term for structures supporting the complete repertoire given above.

In [2] we introduced a data structure supporting all instructions with a worst-case processing time O(log log u) for each element processed, where we use the Uniform RAM time measure [1]. The structure proposed in [2] has two important disadvantages. In the first place it cannot be used in situations where the priority queue consists of real-valued items, like in the case of minimal path computations. Secondly the structure requires more than linear space. To be precise the storage needed is of order $u \cdot \log\log u$; this space moreover must be initialized in time O($u \cdot \log\log u$) before the structure can be used.

In this note we show that the second objection can be amended by combining some ideas used before in [2]. The present and previous results are expressed by the following lemma's and theorem:

Lemma 1. *There exists a datastructure $P(u)$ supporting the full repertoire in space* O(u) *with* O(log u) *processing time per element processed.*

Lemma 2. *There exists a datastructure $Q(u)$ supporting the full repertoire in space* O($u \cdot \log\log u$) *with* O(log log u) *processing time per element processed.*

Theorem 3. *There exists a datastructure $R(u)$ supporting the full repertoire in space* O(u) *with* O(log log u) *processing time per element processed.*

The new structure R thus combines the good characteristics of both P and Q respectively.

The structure required by Lemma 2 is described in [2] for the case that u is a power of 2, the later restriction being inessential. Structures satisfying the requirements of Lemma 1 are well-known. E.g. the 2–3 trees described in [1] are an example; another example is the "silly" structure described in [2].

2. Two-level priority queues

The structure R is obtained from the structures P and Q above using a two-level approach which is used in [2] as the fundamental divide and conquer trick on which the structure Q is based. To obtain R from P and Q the decomposition however does not need to be applied recursively.

Assume that $u = n \cdot m$. We decompose the universe $U = \{1, ..., u\}$ into a "cluster" of n "galaxies", each of size m. So the ith galaxy equals

$$V_i = \{(i-1)m+1, ..., im\}\,.$$

A subset $S \subset U$ is completely determined by its intersections $S_i = S \cap V_i$. Moreover it is useful to consider as well the subcluster of nonempty galaxies $T = \{i \mid S_i \neq \emptyset\}$.

It turns out that each instruction from the single-set manipulation repertoire can be executed by performing a program of similar instructions involving the set T or some sets S_i. For example, to insert an element in S first insert this element into the set S_i to which it should belong and next insert the entire galaxy V_i in T if the set S_i was empty before. The programs for the decomposed instructions are given in Section 3. The procedures there are to be understood as follows: from the suffix of the procedure name (*un*iverse, *cl*uster or *gal*axy) it can be seen whether the procedure operates on S, T, or some S_i in which latter case the value of i is given as an additional parameter of the procedure.

The priority queues used to support the S_i will be called the bottom queues, whereas the structure used to support T is called the top queue. Clearly we need n bottom queues of size m and a single top queue of size n. Each leaf of the top queue will contain a pointer to the bottom queue represented by this leaf. With this decomposition in mind we now can prove theorem 3.

Proof of Theorem 3. Assume $u = k \cdot \lceil \log\log k \rceil$ for some k. Take $n = k$ and $m = \lceil \log\log k \rceil$. By at most doubling u we may moreover assume that n is a power of 2.

To represent the top queue T we use a single copy $Q(n)$ of the structure from Lemma 2, whereas the bottom queues are represented using structure $P(m)$ from Lemma 1. The time and storage requirements are estimated as follows:

Storage. We need space $O(n \cdot \log\log n)$ for the $Q(n)$ top queue plus n times the $O(m)$ space for the n $P(m)$ bottom queues. This yields

$$O(n \cdot \log\log n) + n \cdot O(m) = O(k \cdot \log\log k) + O(k \cdot \log\log k) = O(u)\,.$$

Time. Each two-level program involves a bounded number of top and/or bottom calls (which number is never larger than three). The time needed for executing this program is therefore estimated by adding the time needed for a single bottom and top call. This yields

$$O(\log\log n) + O(\log m) = O(\log\log k) + O(\log\log\log k) = O(\log\log u)\,.$$

This completes the proof.

Note that we might have used for the bottom queues even a structure like an unsorted list which uses time $O(m)$ and space $O(m)$.

3. The two-level instructions

The ALGOL-like procedures below give the decompositions for the most relevant priority queue instruc-

Table 1

```
proc insert un(x);
        begin y:=gal axy(x);
        if empty gal(y)   then begin insert gal(x,y); insert cl(y) end
                          else insert gal(x,y)
        end;

proc delete un(x);
        begin y:=gal axy(x); delete gal(x,y);
        if empty gal(y) then delete cl(y)
        end;

fun member un(x);
        begin y:=gal axy(x);
        if member cl(y)   then member un := member gal(x,y)
                          else member un := false
        end;

fun empty un ; empty un := empty cl ;
fun min un ; min un := min gal(min cl) ;
fun successor un(x);
        begin y:=gal axy(x); t:=successor gal(x,y);
        if t = nil # no successor in Sy #
        then begin z := successor cl(y); successor un := min gal(z) end
        else successor un := t
        end;
```

tions. It can be read from their bodies that each instruction involves at most three top and/or bottom operations.

The function *gal axy* computes for each x in the range $1, ..., u$ the value j such that $x \in S_j$. Since division is not a permitted instruction on a RAM, we use a precomputed table for computing this function. See Table 1. Procedures for *extract min, max, extract max,* and *predecessor* are analogous and left to the reader.

References

[1] A.V. Aho, J.E. Hopcroft, J.D. Ullman, The design and analysis of computer algorithms (Addison Wesley, Reading, MA, 1974).

[2] P. van Emde Boas, R. Kaas, E. Zijlstra, Design and implementation of an efficient priority queue, Math. Systems Theory, to appear.