An O(nloglogn) On-line Algorithm for the Insert-Extract Min Problem

P. van Emde Boas †

TR 74-221

December 1974

Computer Science Department
Cornell University
Ithaca, N.Y. 14853

---

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

Author's address: Math. Inst. Univ. of Amsterdam, Roetersstraat 15, or Math. Centre 2nd Boerhaavestraat 49, both in Amsterdam, Netherlands.

An O(nloglogn) On-line Algorithm for the Insert-Extract Min Problem

P. van Emde Boas†

ABSTRACT

Integers within the range 1,...,n are inserted in a set, and on several occasions the minimal element is extracted from the set. We present an algorithm to execute a sequence of O(n) of these instructions on-line in time O(nloglogn) on a Random Access Machine. The instruction repertoire can be extended by instructions like allmin(i) (delete all elements not greater than i), extract max, or predecessor(i) (find the largest element < i), without disturbing the O(loglogn) processing time per item.

Whereas the off-line insert-extract min problem is known to be reducible to the on-line union-find problem, we prove that the off-line insert-allmin problem is equivalent to the off-line union-find problem; hence the off-line problems have faster algorithms.

As an application we show that our algorithm can be used to process a sequence of O(n) instructions of the types: "split an interval", "unite two adjacent intervals", and "find the interval currently containing element j", on-line in time O(nloglogn).

Keywords: set-manipulation, Analysis of Algorithms, binary tree.

CR categories: 5.25, 5.30, 5.32

AMS MOS 70 classification: 68A20

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

Author's address: Math. Inst. Univ. of Amsterdam Roetersstraat 15, or Math. Centre 2nd Boerhaavestraat 49, both in Amsterdam, Netherlands.

1. Introduction

Suppose that one is handling a sequence of jobs at a single processing unit. The jobs are given priorities ranging from 1 to n , and they must be processed in order of their priorities. Each time a new job is offered it must be inserted in the priority system. When the processing unit has completed a job, a new job is selected from the jobs having the lowest present priority value.

A possible implementation for this priority system might be a single priority queue, or one could gather all jobs having the same priority in a linear list, and keep some data structure representing the currently active priority values.

If we use a binary heap to represent the active priority values this part of the structure can be maintained in time $O(\log n)$ for each operation. Consequently the time to process a sequence of n jobs becomes $O(n \log n)$ (disregarding the time to process the jobs themselves).

In this report we present a more efficient method to implement the underlying structure, and we sketch the algorithms for the primitive operations of inserting, deleting and finding the minimum. These algorithms can be seen to have an $O(\log\log n)$ processing time for each item, thus yielding an $O(n \log\log n)$ algorithm for executing a sequence of $O(n)$ instructions on-line.

Clearly we must use somehow the fact that our elements are selected from the fixed domain $\{1,\ldots,n\}$; otherwise our method should make it possible to sort arbitrary sequences of length n in time $O(n \log\log n)$, and this is known to be impossible.

In the case where the sequence of insert, delete and extract-mi instructions is to be executed off-line, a faster algorithm exists. The off-line case of the problem is reducible to the on-line union-find problem [1] for which an $O(nA(n))$ algorithm† is known [4].

A similar problem is the execution of a sequence of $O(n)$ instructions of the types insert(x) and allmin(i), where the latter instruction removes all elements from the set having value $\leq i$. In the on-line case this problem is directly reducible to the on-line insert-extract min problem. For the off-line case this reducibility seems impossible, but we can prove an equivalence between this problem and the off-line union-find problem [1]. Consequently the problem is solvable in time $O(n\log\log n)$ and $O(nA(n))$ in the on-line and off-line cases, respectively.

The data structure which we use in our on-line algorithms allows a number of other instructions which can be executed in time $O(\log\log n)$ for each item. These instructions are extract max, predecessor(i) (find the largest element in the set < i), and successor(i) (find the least element in the set > i).

Using the above instructions we can solve the following problem in time $O(n\log\log n)$ as well. Consider partitions of the set $\{1,\ldots,n\}$ in consecutive intervals. The instruction meld unites two adjacent intervals; the instruction split divides an interval into two adjacent ones, and the instruction find computes the interval currently containing a given element. A sequence of $O(n)$ instructions of the above type can be executed in time $O(n\log\log n)$.

2. Definitions

Throughout this paper n is a fixed integer, which we assume

---

† $A(n)$ is the inverse of a function that grows about as fast as the Ackerman function.

for simplicity to be of the form $n = 2^{2^k}$. Clearly n equals the number of leaves in a binary tree of height 2^k.

If m is an integer then the largest number d such that $2^d \mid m$ is called the <u>rank</u> of m. After having chosen n fixed we let rank(0) = k+1. For example, rank(12) = 2.

Consider a binary tree of height 2^k. The <u>level</u> of a vertex is its distance to the root. The <u>rank</u> of a vertex is the rank of its level.

A <u>canonical subtree</u> (CS) is a subtree of height 2^d having a root of rank $\geq d$. The number d is called the rank of the CS. The left (right)-hand subtree of a CS (including its root) is called a left (right) canonical subtree (LCS and RCS).

With a vertex v of rank d we associate the following canonical subtrees:

UC(v): the <u>upper canonical subtree of v</u> is the unique CS of rank d having v as a leaf.

LC(v): the <u>lower canonical subtree of v</u> is the CS of rank d with root v.

LLC(v): the <u>left canonical subtree of v</u> is the LCS of rank d with root v.

RLC(v): the <u>right canonical subtree of v</u> is the RCS of rank d with root v.

The <u>sign</u> of a vertex v at some ancestor w equals ℓ or r depending on whether v belongs to the left- or right-hand subtree of w.

An <u>internal vertex</u> of a subtree of height j with root v is a descendant w of v satisfying level(v) < level(w) < level(v)+j.

For each vertex v except the root and $0 \leq j \leq k+1$ we denote by father(j,v) the nearest proper ancestor of v whose rank is $\geq j$. Clearly father$(0,v)$ is the father of v.

father(j,v) can be described alternatively as the root of the CS of rank j which contains v as a non-root.

If v is the root of a CS of rank $d > 0$ and if w is a leaf of this CS then father$(w,d-1)$ is called the vertex <u>half-way inbetween v and w</u>; notation hw(w,v).

For a subset $S \subset \{1,\ldots,n\}$ we consider the following instructions:

min (max) : Compute the least (largest) element of S.

insert(j) : put j in S, if not already present; otherwise undefined.

delete(j) : delete j from S if present; otherwise undefined.

member(j) : compute whether $j \in S$.

extract min : delete the least element from S provided S not empty; otherwise undefined.

extract max : idem for the largest element.

allmin(i) : remove from S all elements $\leq i$.

allmax(i) : remove from S all elements $\geq i$.

predecessor(i): compute the largest element in $S < i$, if such element exists; otherwise undefined.

successor(i) : idem for the least element in $S > i$.

For a partition $\Pi = \{I_1,\ldots,I_r\}$ of $\{1,\ldots,n\}$ in adjacent intervals, we consider the following instructions:

find(i) : compute the interval currently containing i.

meld(I,J) : unite the intervals I and J if adjacent; otherwise undefined.

split(I,i) : split the interval I into two intervals $I \cap [1,i)$

and $I \cap [i,n]$; consequently split(I,i) is a dummy statement when $i \notin I$ or when i is the lowest integer in I .

For an arbitrary partition $\Pi = \{A,B,\ldots,X\}$ of $\{1,\ldots,n\}$ 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.

3. The representation of sets in the tree

To represent a subset S of $\{1,\ldots,n\}$ where $n = 2^{2^k}$ we use a binary tree of height 2^k. The leaves of the tree correspond to the elements $1,\ldots,n$.

At each vertex a number of information fields will be available the nature of which is described below. This information is used to describe which elements are currently contained in the set S .

We call the leaves corresponding to elements of S , together with all their ancestors in the tree, present; other vertices will be called non-present. The situation that no vertex in the tree is present is excluded by presuming that S always contains the element n .

A traditional method of representing the set is the marking of all present vertices; this however yields at best O(logn) algorithms for inserting and deleting elements in S .

Instead we use the trick of finding the key information needed to locate specific members of S by doing binary search on the levels. This trick has been used before by Aho, Hopcroft and Ullman for an O(nloglogn) algorithm for the lowest ancestor problem [2].

Suppose that we consider a canonical subtree, and assume that its top is present; moreover precisely one of its leaves in the left-hand subtree is present. In this case we might as well store this unique leaf in a specific information field at the top, in this way enabling ourselves to proceed in one step from the top to the level of the leaves.

Next assume that more than one leaf in the left-hand subtree is present. In this case the left-hand subtree must contain an internal vertex such that both its sons are present. Such a vertex will be called a branch-point. Clearly the branch-point should be a reasonable place to store information about both vertices to which it is leading.

The problem remains how to locate a branch-point. This can be done by a binary search procedure, if in case a branch-point occurs inbetween the top t and some leaf v we store, at the vertex w which lies halfway inbetween t and v, information telling us whether a branch point occurs at w, inbetween t and w, or inbetween w and v (the three may happen simultaneously in the case that more than one branch-point exists on the path from t to v).

Assume that v is a vertex at level $j < 2^k$ and let d be the rank of v. The information stored at the vertex v will tell us what is happening in the canonical subtrees of v. The information will be stored in five fields. The first two fields are called $\ell min[v]$ and $rmin[v]$. If $\ell min[v]$ is defined it contains the leftmost present leaf of $LLC(v)$. The field $\ell min[v]$ will be undefined if no such leaf exists. The field $rmin[v]$ similarly contains the leftmost present leaf of $RLC(v)$ if such a leaf exists.

The three other fields are called $ub[v]$, $\ell b[v]$ and $rb[v]$; they can contain a boolean value + or - , or be undefined. If $\ell b[v] = +$ this indicates that on the path from v to $\ell min[v]$ a branch-point occurs. In the case that $\ell b[v] = -$ there is only a unique leaf present in LLC(v), and $\ell b[v]$ is undefined if no such leaf exists. The field $rb[v]$ has a similar meaning for RLC(v).

Finally the field $ub[v] = +$ or $-$ depending on whether a branch-point occurs on the path from v to the top of UC(v).

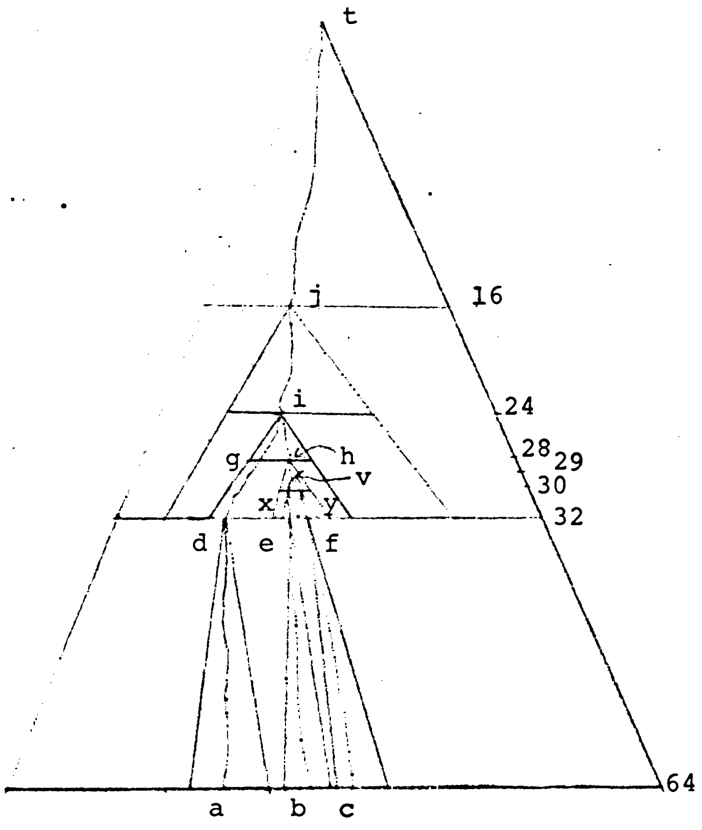
There is a further situation in which all information fields at v are undefined although the vertex v is present. This situation arises if v is a present vertex such that there exist t and w such that w is the unique present leaf in LLC(t) or RLC(t), and such that v lies on the path from t to w. This is the situation we mentioned at the beginning of this section. There is no reason to store information at v since inbetween t and w nothing happens.

We can summarize the above by stating the following assertions about proper information contents:

<u>properness condition:</u> If LLC(v) has only one present leaf w then for each internal vertex in LLC(v) all information fields are undefined. If LLC(v) has more than one present leaf and if w is any present leaf then at the vertex x halfway inbetween v and w the field $ub[x]$ and the two left fields ($\ell min[x]$ and $\ell b[x]$) or the two right-hand fields are defined; all fields at x are defined in the case that x itself is a branch-point.

See also figure 1.

Note that for a vertex v of rank d at level j the field ℓmin[v] contains a vertex w at level $j+2^d$. The field ℓmin[w] or rmin[w] contains a left-most leaf of its lower canonical subtree, and proceding this way we find in O(k) steps a leaf at the bottom of our original tree. On the other hand the field ℓmin[v] by itself can be used to find the present vertex on the path from v to w at level i where $j < i < j+2^d$. If we encode the vertices by binary strings we can use elementary bit manipulation for performing this implicit address translation.



v = [ℓmin, rmin, ub, ℓb, rb]

~ denotes undefined

	level	rank
t = [a,~,-,+,~]	0	7
j = [d,~,-,+,~]	16	4
i = [d,e,-,-,+]	24	3
g = [~,~,~,~,~]	28	2
h = [e,~,-,+,~]	28	2
d = [a,~,+,-,~]	32	5
e = [b,~,+,-,~]	32	5
f = [~,c,+,~,-]	32	5
x = [e,~,+,-,~]	30	1
y = [~,f,+,~,-]	30	1
v = [x,y,-,-,-]	29	0

figure 1. representation of a three element set in a tree of height 64.

4. Algorithms for the primitive operations

Since we are frequently in the situation where we consider a vertex v and some descendant w, without knowing in advance whether w lies in the left- or right-hand subtree of v, we introduce the following addressing primitives and elementary actions;

~ describes the undefined value

mymin[w,v] is the minfield of v in the direction of w.

yourmin[w,v] is the minfield of v in the other direction.

myb[w,v] and yourb[w,v] are analogously defined.

myfields[w,v] is the pair consisting of mymin[w,v] and myb[w,v]; yourfields[w,v] is similarly defined.

minof(v) = <u>if</u> min[v] $\neq$ ~ <u>then</u> ℓmin[v] <u>else</u> rmin[v] <u>fi</u>

minof(v) gives the least leaf in LC(v) if such a leaf is registered at v.

used(v) = (ub[v] $\neq$ ~)

used(v) tests whether information is stored at v.

desc(v,w,h) computes the vertex at distance 2^h from v on the path from v to w, if w is a descendant of v at distance $\geq 2^h$; otherwise desc is undefined.

clear(v) sets all fields at v at undefined.

We will consider the following four primitive operations:

1) Locate the lowest present vertex on a path from a non-present leaf to the root.
2) insert a non-present leaf and update the structure.
3) locate the lowest branch-point on the path from a present leaf to the root.
4) delete a present leaf from the tree (i.e. make it non-present), and update the structure.

It is clear that these operations must be programmed in such a way that the properness condition is preserved.

Note that operation 1) is not used to solve the insert-extract min problem. We will use it however in our further applications.

We describe the algorithms by ALGOL-like programs, combining some features from ALGOL 60 and ALGOL 68 (and omitting declarations, etc.). The procedures for 1) , 2) and 3) are recursive, whereas we prefer an iterative implementation for 4), to make it more clear in which way information is transferred from smaller to larger canonical subtrees.

We assume that whenever a procedure is called upon a subtree, the root of this subtree is a present vertex for which used(v) is true. Moreover, the subtree should be a CS although not necessarily the lower canonical subtree of the root.

```
procedure  locate(leaf,top,h);
     ¢ locates the lowest present vertex on the path from leaf to top;
      it is assumed that top is present and used, and that leaf is
      not present. The value delivered by locate is the vertex we
      are looking for. ¢
begin  if mymin[leaf,top] = ~
       then locate:= top; return
       fi;

       pres:= mymin[leaf,top]; pres:= desc(top,pres,h);
          ¢ address translation ¢
       hℓ:= hw(leaf,top); hp:= hw(pres,top);

       if myb[leaf,top] = -
          ¢ no intermediate information stored ¢
```

```
        then if hp ≠ hℓ
             then locate:= locate(hℓ,top,h-1)
               ¢ look in upper subtree ¢
             else myfields[pres,hp]:= (pres,-);
               ¢ insert temporary information ¢
               locate:= locate(leaf,hp,h-1);
                 ¢ and look in lower subtree ¢
               myfields[pres,hp]:= (~,~)
                 ¢ and erase temporary information ¢
             fi
        else
          ¢ intermediate information is already stored ¢
             if used(hℓ)
               ¢ there exists already a present path through hℓ ¢
             then locate:= locate(leaf,hℓ,h-1)
               ¢ look in lower subtree ¢
             else locate:= locate(hℓ,top,h-1)
               ¢ look in upper subtree ¢
             fi
        fi; return: skip
  end locate;
```

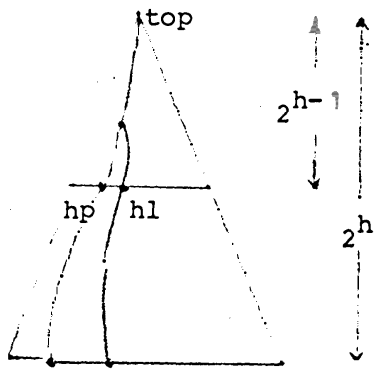


figure 2. a call of locate resulting in recursive call on the upper subtree

```
procedure  insert(leaf,top,h);
  ¢ inserts leaf in tree; it is assumed that top is present and leaf
    is not present.  Insert is given no value.  The structures of
    insert and locate are about equal, and hence we have omitted the
    explanatory comments ¢
begin if mymin[leaf,top] = ~
      then myfields[leaf,top]:= (leaf,-); return
      fi;

      pres:= mymin[leaf,top]; pres:= desc(top,pres,leaf);
      hℓ:= hw(leaf,top); hp:= hw(pres,top);

      if myb[leaf,top] = -
      then
           if hp ≠ hℓ
           then myfields[leaf,hℓ]:= (leaf,-);
                myfields[pres,hp]:= (pres,-);
                ub[hℓ]:= ub[hp]:= +;
                insert(hℓ,top,h-1)
           else myfields[pres,hp]:= (pres,-); ub[hp]:= -;
                insert(leaf,hp,h-1)
           fi; myb[leaf,top]:= +
      else if used(hℓ)
           then insert(leaf,hℓ,h-1)
           else ub[hℓ]:= ub[hp]:= +;
                insert(hℓ,top,h-1)
           fi
      fi;
      mymin[leaf,top]:= min(leaf,pres); return: skip
end insert;
```

```
procedure lowbranchp(leaf,top,h);
  ¢ locates the lowest branch-point on the path from leaf to the
    root; it is assumed that leaf is present and that it is not the
    unique present leaf.  The value delivered by lowbranchp is this
    branch-point ¢
begin if myb[leaf,top] = -  then lowbranchp:= top; return fi
      hℓ:= hw(leaf,h-1);
      lowbranchp:= if myb[leaf,hℓ] = +
                      ¢ branch point below hℓ ¢
                   then lowbranchp(leaf,hℓ,h-1)
                   elif yourb[leaf,hℓ] ≠ ~
                      ¢ hℓ is itself a branch-point ¢
                   then hℓ
                   else lowbranchp(hℓ,top,h-1)
                      ¢ branchpoint above hℓ ¢
                   fi; return: skip
end lowbranchp;
```

For the procedure delete we use a nonrecursive description. delete locates the lowest branch point. From this point it traces back the sequence of CS's containing this point, deleting the information related to the leaf which is removed, and, if the remaining present part of the tree within the CS under consideration happens to be reduced to a single present path, deleting also the internal information which by now has become superfluous. Consequently, delete must transfer the information as to whether a CS contains a unique present path to the next CS. This information is stored in the boolean "deletable". See also figure 3.

The meaning of the variables in delete is the following:

br: initially the lowest branch-point on the path from the leaf to be deleted to the root; during sequel of execution of delete br represents the "midpoint" of a remaining present path through a CS currently under consideration.

top: the top of the CS currently considered.

pres: a present leaf of the CS currently considered; initially the left-most remaining present leaf of the lowest branch-point.

laf: the leaf in the CS currently considered on the path to leaf of the complete tree which is going to be deleted.

lif: Initially the lowest branch-point on the path from the leaf to be deleted to the top; during sequel of execution of delete, lif contains the old value of laf, after the vertex represented by this value has become an internal vertex of the CS currently under consideration (which happens if this CS has the same top as the former one).

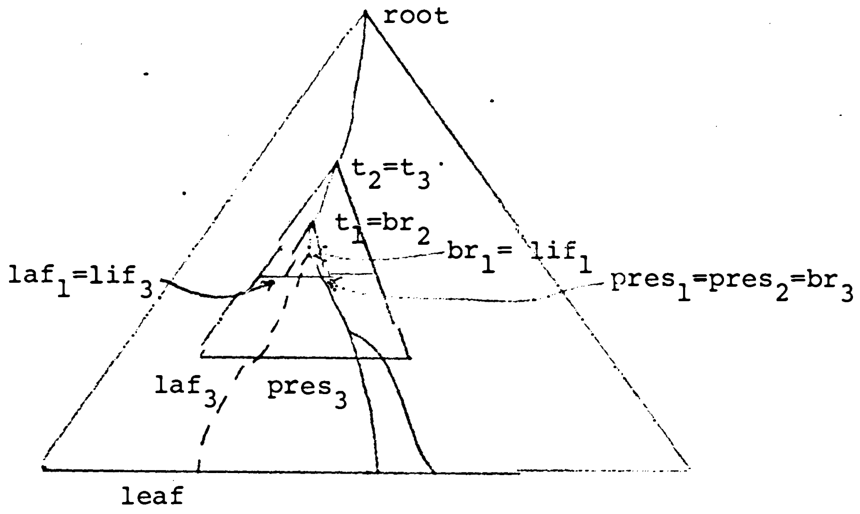


figure 3. three stages in the execution of a call of delete

```
procedure  delete(leaf);
  ¢ since this procedure is nonrecursive the parameters  h  and  top
    are unnecessary.  leaf  is assumed to be present but not to be
    the unique present leaf of the complete tree.  ¢
begin  br:= lowbranchp(leaf,root,k); j:= rank(br);
       t:= father(br,j);
         ¢ top UC(br) ¢
       pres:= yourmin[leaf,br];
         ¢ left-most remaining leaf of LC(br) ¢
       laf:= mymin[leaf,br];
         ¢ first of the chain of vertices on path to leaf to be
           cleared ¢
       lif:= br;
         ¢ extra pointer to be used in the clearing process ¢
       myfields[laf,lif]:= (~,~);
       deletable:= true;
         ¢ remains true as long as no other branch-point on the
           paths from  t  to  pres  is detected. ¢;
       while j < k do
         ¢ process the CS's containing  br  in order of increasing ı
       deletable:= deletable and ub[br] = - and
                 ((ℓb[br] = - and rb[br] = ~) or (ℓb[br] = ~ and
                 rb[br] = -));
         ¢ there is no branch-point on the path from  t  to  pres ¢
       if deletable then clear(br) fi;
         ¢ next we go to the CS of rank j+1 containing the current
           one.  Comparing the ranks of  t  and  pres  we can decic
           whether we have to choose a new top or a new bottom leve
           to find this CS ¢
```

```
proc  neighbor(j);
begin br:= if present(j) then lowbranchp(j,root,k)
           else locate(j,root,k) fi;
       s:= sign(j,br); pres:= if s = ℓ then yourmin[j,br] else
          yourmax[j,br] fi;
      while rank(pres) < k do
            pres:= if s = ℓ then minof(pres) else maxof(pres) fi  od;
      neighbor:= pres
end
```

The procedure neighbor computes either the predecessor or the successor of j in the set, depending on whether the sign of j at the lowest branch-point (or lowest present point if j is not present) on the path of j to the root equals r or ℓ. Note that in order to compute the predecessor directly we must find the lowest branch-point where j has sign + . This operation is not easy to perform on our data structure.

In order to be able to find the predecessor of j regardless of the sign of j at br we may use a second representation of the set by a doubly linked list, having pointers from the present leaves in the tree to cells in the list.

In this situation a call of neighbor provides us one of the two neighbors in the list; following the links in the list we find the other in constant time.

5. The O(nloglogn) algorithm for the insert-extract min problem

Let k be selected such that $m = 2^{2^k} > n$ ($m = n$ is excluded by our requirement that the last element always is present). We construct the binary tree of height 2^k described in section 3. The vertex m is inserted by initializing the fields at the top at

(~,m,¯,~,+) for ℓmin, rmin, ub, ℓb, rb respectively.

The translations of the two operations of the insert-extract min problems are the following:

insert(j) → insert(j,root,k)

extract min → delete(minof(root)).

Since the operation allmin(i) is equivalent to

<u>while</u> minof(root) $\leq$ i <u>do</u> extract min <u>od</u>

we can also execute an allmin instruction in time O(k) for each removed item. Note that there are never more items removed than there have been inserted by the sequence of insert instructions executed before.

If we have the extended structure available which was suggested at the end of Section 4, we can execute the instruction predecessor by: <u>if</u> present(i) <u>then</u> predecessor:= pred(i)

¢ in the list ¢

<u>else</u> j:= neighbor(i); predecessor:= <u>if</u> j < i <u>then</u> j <u>else</u> pred(j

¢ in the list ¢ <u>fi</u>

<u>fi</u>;

pred(i) denotes the predecessor of i in the doubly linked list.

Leaving the remaining instructions to the reader we summarize our observations by:

<u>Theorem 1</u>: a) A sequence of O(n) instructions of the types: insert(:
member(i); delete(i); min; extract min; allmin(i) can be executed in time O(nloglogn) on a Random Access machine using the algorithms of the preceding section.

b) A sequence of O(n) instructions of the types: insert(
member(i); delete(i); min; extractmin; allmin(i); max; extract
allmax(i); predecessor(i); successor(i) can be executed in ti

```
    if rank(t) > rank(pres)
      ¢ top remains top; pres becomes next internal point ¢
    then lif:= laf;
           ¢ vertex on the death path ¢
         laf:= mymin[leaf,laf];
           ¢ next vertex on the death path ¢
         clear(lif);
         if deletable then ub[pres]:= - fi;
           ¢ last branch-point on this segment of path was deleted ¢
         br:= pres; pres:= minof(pres)
    else
      ¢ pres remains at bottom, top becomes new internal point ¢
         if mymin[br,t] = laf then mymin[br,t]:= pres fi;
           ¢ laf no longer is present ¢
         if deletable then myb[br,t]:= - fi;
           ¢ the last branch-point in LC(t) was deleted ¢
         br:= t; t:= father(t,j+1)
    fi;
    j:= j+1 od
end delete;
```

¢ The precise reader may object that the above procedure, during execution of the loop with j = k-1 calls for operations on leaves which are only defined on internal vertices ¢

We leave it to the reader to convince himself that the first three procedures are straight line algorithms, except for a single call of the procedure itself with the value of h decreased by one, and that the procedure if called with h=0 must terminate, since the procedures are called recursively only in the case that an internal branch-point exists, and a CS of rank 0 has height 1 and consequen no internal vertices at all.

The procedure delete contains a loop which is executed at most k times, whereas the program within the loop is a straight line program.

From these observations it is easy to see that the run-time of each of the above procedures is of order $k = O(\log\log n)$.

The reader should also convince himself that the algorithms presented above preserve the properness condition.

Before discussing the applications of our data structure we mention a further extension of the structure, which we use in the sequel. The choice of inserting at each vertex the least present leaf in its half lower canonical subtrees is quite arbitrary; in fact, the algorithms should also work in the case that we had stored an arbitrary present leaf. The motivation for storing the least lea: is that we want to solve the insert-extract min problem.

There is however no reason why we should not introduce two more fields: $\ell max[v]$ and $rmax[v]$, representing the largest present leaf of a half-CS. By having these fields available the branch-poin indicators ℓb and rb become superfluous, since their values can be derived from the other fields; a branch-point is visible by the fact that a max and a min field have unequal values.

We leave the actual programming work of extending the data structure and the algorithms (without disturbing the $O(k)$ processing time) to the interested reader.

Having the maxfields available we can consider the following procedure:

O(nloglogn) on a Random Access machine using the extended data structure of the preceding section.

6. The algorithm for the meld-split-find problem

Let $\{I_1,\ldots,I_k\}$ be a partition of the interval $\{1,\ldots,n\}$ into consecutive intervals and let j_r be the leftmost point of the interval I_r. We call j_r the <u>endpoint</u> of I_r.

Clearly the correspondence $\{I_1,\ldots,I_k\} \rightleftarrows \{j_1,\ldots,j_k\}$ is a bijective mapping from partitions of $\{1,\ldots,n\}$ into adjacent intervals and subsets $\{1,\ldots,n\}$ of S.

This shows that we may represent a particular partition by the left-most points of its members. If we now can express the manipulations meld(I,J), split(I,j) and find(i) by means of the operations on subsets of $\{1,\ldots,n\}$ mentioned in Theorem 1, in such a way that each manipulation is expressed by a finite sequence of operations which does not depend on the arguments, we are done.

The expressions are given below.

```
proc find(i);
    if member(i) then i else predecessor(i) fi;
```

If i itself is an endpoint of an interval, i itself represents the interval containing it; otherwise this interval is represented by the largest element $<i$ in the set of endpoints of intervals.

```
proc meld(I,J);
     ¢ i = leftend(I) , j = leftend(J) ¢
       if i = successor(j) then delete(i)
       elif j = successor(i) then delete(j)
       else error fi
```

Adjacent intervals are recognizable by the fact that one endpoint is the successor of the other in the set of endpoints of intervals.

```
proc split(I,j)
    ¢ i = leftend(I) ¢
      if predecessor(j) = i and not present(j)
       then insert(j) fi
```

The operation split is vacuous unless j lies in the interval I without being its endpoint; if so, j becomes itself a new endpoint.

The above observations yield:

Theorem 2: A sequence of O(n) instructions of the type find(i), meld(I,J), and split(I,j) , to be executed on partitions of the set {i,...,n} into adjacent intervals, can be executed in time O(nloglogn on a Random Access machine.

Remark: If either operation meld or split is not requested, O(nG resp. O(nA(n)) algorithms are known [1,3,4] where G(n) is the least number k such that $2^{2^{\cdot^{\cdot^{2}}}}\Big\}k \geq n$.

7. The off-line problems

Up to now we have restricted ourselves to on-line algorithms where each instruction must be executed at the time that they are issued. The situation is quite different if the instructions have t be executed off-line, i.e. the sequence of instructions is given in advance and we are free to choose the order in which the answers to the questions occurring in the sequence are generated.

Clearly each on-line algorithm can be used in the off-line case as well but the reverse is not generally true. Consequently, the complexity of the off-line problem is never greater than the complex of the corresponding on-line problem.

In the sequel we use the following fact:

fact 1: A sequence of O(n) instructions of the type union(A,B,C) and find(i), to be executed on partitions of the set $\{1,\ldots,n\}$, can be executed on-line in time $O(nA(n))$ [1,4].

The instruction union(A,B,C) unites the sets named A and B and gives the name C to the union. The instruction find(i) yields the name of the set to which i currently belongs. Initially each element forms a singleton with name i .

When we say in the sequel that problem A is reducible to problem B we mean that a sequence of O(n) instructions of the types dealt with by problem A on a structure of size n can be executed by an algorithm which takes time O(n) + the time needed for O(n) calls of the operations from problem B on a structure of size O(n). Notation $A \leq B$.

Hence if $A \leq B$ then the order of the complexity of A is not greater than the order of the complexity of B (assuming that neither A nor B have complexity growing less than linearly in n).

In the case that B is an off-line problem we must be able to translate the sequence of A instructions into a straight line program containing B instructions; moreover this translation must be performed in time O(n). If we are unable to do so we may still be able to reduce an off-line problem A to an on-line problem B. An example of this situation is given by

fact 2: off-line insert-extract min $\leq$ on-line union-find [1].

This shows that off-line insert-extract min is an $O(n\,A(n))$ problem, hence this problem might be easier to solve off-line than on-line.

In Section 5 we gave actually a proof of:

fact 3: on-line insert-allmin $\leq$ on-line insert-extract min

It should be noted however that a similar reducibility for the off-line problems seems not to be possible, since one cannot predict, in advance, how many extract-min instructions will be executed by a single allmin instruction.

The reduction of an on-line problem to an off-line one seems to be impossible.

In this section we probe the following two reducibilities:

Theorem 3: off-line insert-allmin $\leq$ off-line union-find.

Theorem 4: off-line union-find $\leq$ off-line insert-allmin .

In the formulation of the off-line insert-allmin problem it is tacitly assumed that each element is inserted at most once.

These two theorems together show that the off-line union-find problem and the off-line insert-allmin problem are equivalent.

Proof (of Theorem 3): We use a modification of the method used to prove fact 2 [1,2].

Consider a sequence of insert and allmin instructions. For each instruction insert(i) 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_j between two consecutive barriers b_j and b_i in a set $S(b_i)$ having the top 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 $O(n)$.

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. ∎

Proof (Theorem 4): 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. At each vertex we store, moreover, 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 $O(n)$.

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 the leaf (i.e. all instructions find(i) where $\{i\}$ is the set represented by the leaf).

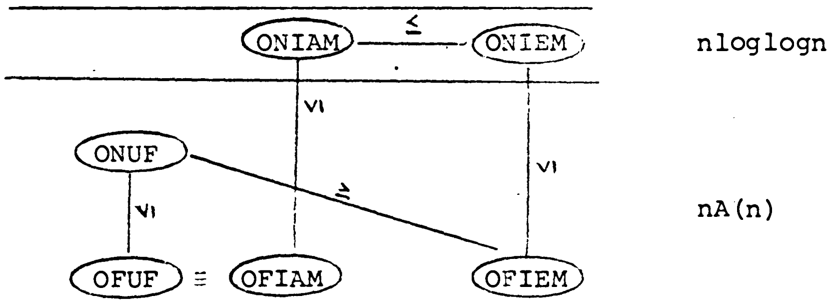
If we process an arbitrary vertex v we remember that the set which is represented by v contains the elements represented by the leaves of the subtree with root v , during the period starting with the formation of the union (which time T is stored at vertex v) and the time T' at which this set is united into a larger set (whic time is stored at the father of v). Hence we should throw out of the bucket all find instructions having time 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 everywhing that we have described can be executed in time O(n) for the complete tree, but this allmin instrution makes the algorithm nonlinear (at least for the time 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. Conclusions

The reducibilities mentioned in the preceding sections are collected in the diagram below (acronyms representing the problems).



This scheme leaves a number of questions unanswered. From the six considered problems, only two have been proven to be equivalent. Furthermore there seems to be no method known to reduce on-line union-find to any of the on-line insert-min problems, although the known algorithm for the on-line union-find problem runs much faster than the nloglogn algorithms for the on-line insert-min problems presented in this report.

From the scheme it is clear that the on-line insert-extract min problem may be the hardest of the six problems. If we want to improve our upper bound we must invent a new algorithm since it is easy to construct a program for which our tree algorithm takes time O(nloglogn). Consider for example the program:

insert(n); insert(1); insert(2); extractmin; extractmin;

insert(3); insert(4); extractmin; ...

About half of the instructions in this sequence take time O(loglogn) since the rank of the branch points which are constructed and subsequently deleted is zero.

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. 5th ACM Symp. Th. of Comp. 1973, 253-265.

3. Hopcroft, J.E. Personal communication. November 1974.

4. Tarjan, R.E. Efficiency of a good but not linear set union algorithm. Mem. ERL-M434 Electronics Research Laboratory College of Engineering, University of California, Berkeley, March 1974. (To appear in JACM).