ALTERNATIVE PATH COMPRESSION TECHNIQUES

Jan van Leeuwen

Department of Computer Science
University of UTRECHT
3508 TA UTRECHT, the NETHERLANDS

and

Theo van der Weide

Department of Applied Mathematics and Computer Science
University of LEIDEN
Wassenaarseweg 80, P.O. Box 2060
LEIDEN, the NETHERLANDS

Technical Report RUU-CS-77-3

September 1977

Department of Computer Science
University of UTRECHT
3508 TA UTRECHT
the NETHERLANDS

ALTERNATIVE PATH COMPRESSION TECHNIQUES*

Jan van Leeuwen
Department of Computer Science
University of UTRECHT
UTRECHT 2506/the NETHERLANDS

and

Theo van der Weide
Department of Computer Science
University of LEIDEN
LEIDEN / the NETHERLANDS

Abstract. UNION - FIND programs symbolise a large class of algorithm for merging and searching disjoint sets, which are represented as trees In the "collapsing rule" for FINDs one must first locate a root and the traverse the same FIND-path a second time in order to attach each of its nodes to the root. In an attempt to find a more elegant imple mentation, we shall consider the problem of how the "second pass" can be eliminated while preserving good efficiency (with or without a balancing rule for UNIONs). We consider several alternative tree compression methods, including Rem's algorithm as presented by Dijkstra. We show that in worst case the algorithms are not as gooc as expected. We propose a simpler one-pass technique called "path halving", which performs well in both the balanced and unbalanced cases. We show that the time needed for executing $O(n)$ UNIONs an FINDs is bounded by $O(n \log n)$ when only path-halving is used. I the "weighted union" rule is used for set-merging and path-halving for FINDs, then the cost for $O(n)$ UNIONs and FINDs is boundec by $O(n \log^* n)$. We conclude that for all practical purposes path-halving is a viable alternative to the original collapsing rule

* Preliminary report. An outline of the results were presented at the Fac tagung on Algorithms and Complexity Theory, Oberwolfach, Oct 1977

1. Introduction We shall consider some programming problems concerning the manipulation of disjoint sets as occur for instance in the step-wise composition of an equivalence relation. It is common in such applications to structure each set as a tree, with each node-record representing a distinct element of the set. Nodes x only have a father-link $f(x)$, with $f(x)=x$ if and only if x is a root Typically, a set is named by the smallest element which it contains but we won't always be able to keep its record at the root of the corresponding tree. Therefore we assume that records contain an additional name-field, where for records x at a root $N(x)$ only gives the name of the smallest element in the set. It is always easy to maintain a mapping N^{-1} which enables us to find the root of a named set, and we shall usually omit such bookkeeping details below.

A large class of interesting set-manipulation algorithms can be formulated in terms of the following primitive operations (see Aho, Hopcroft and Ullman [1]):

FIND (x)	retrieve the name of the set to which x belongs (i.e. the N-image of the root of the tree where x is stored)
UNION (x,y)	(where x and y are set names) merge sets x and y and name the new set $\min\{x,y\}$

and also

CHECK (x,y)	test whether x and y belong to the same set
MERGE (x,y)	(where x and y are arbitrary elements)

merge the (two) sets to which x an y belong and name the new set properly.

The start configuration for UNION-FIND programs is a collection of singletons, i.e., a collection of n elements x with $x = f(x) = N(x)$. Typically, in a UNION the root of one tree is made a son of the root of the other tree and a FINC (x) is performed by following the fatherlinks up to the root of the tree:

```
y := x;
while $f(y) \neq y$ do  y := f(y) od;
output (N(y));
```

A program of $O(n)$ UNIONs and FINDs could take as many as $\Theta(n^2)$ steps this way.

The efficiency of UNION-FIND programs can be improved i various ways, and the subject has become almost a standard topic in any discussion of set-manipulation algorithm (see Aho, Hopcroft and Ullman [1]).

UNIONs are easy to perform, but if we create "unbalanced" trees we may have to traverse long paths in subsequent FINDs. The following strategy guarantees a constant, acceptable balance in trees. Let the weight of a node be the number of nodes (including itself) contained in its subtree. Implement UNIONs with the

"weighted union" rule : never attach a heavy tree to the root of a light tree (with ties broken arbitrarily)

One can show that no path in a tree can be longer than $O(\log n)$, and it follows that the cost for $O(n)$ UNIONs and FINDs is down to $O(n \log n)$.

From a different angle (i.e. while leaving the strategy for UNIONs open), one could try to change the internal structure of a tree after it was formed in some way or another in order that later FINDs can be performed at a reduced cost. As the cost of a FIND is proportional to the length of the path traversed up the tree, the following strategy seems to be most effective towards this goal

"collapsing" rule : traverse the FIND-path a second time and attach each of its nodes directly to the root

Paterson [7] proved that $O(n)$ UNIONs and FINDs using the collapsing rule take only $O(n \log n)$ steps also. Fischer [4] showed that this bound cannot be improved, by means of an interesting class of trees (which we shall introduce later).

The most efficient implementation of UNION-FIND programs currently known uses both the weighted union rule and the collapsing rule. Hopcroft and Ullman [5] showed that the cost for $O(n)$ UNIONs and FINDs is now bounded by $O(n \log^* n)$, and Tarjan [8] showed that the algorithm is even much closer to linearity. By a clever argument Tarjan ([8], see also van Leeuwen [10]) showed that the cost for $n-1$ UNIONs and m FINDs is actually bounded by $O(m\,\alpha(m,n))$, where $\alpha(m,n)$ is related to a functional inverse of Ackermann's function (thus a very slowly growing function indeed). Tarjan proved that the algorithm can attain the bound in worst case, and he has demonstrated recently that there can be no linear time algorithm at all for this problem in a rather general implementation model (Tarjan [9]).

Apparently few problems are left concerning UNION-FIND programs. A new issue was raised recently by Dijkstra [3] (Ch 23), who independently proposed the collapsing rule (but no explicit balancing) in a discussion of CHECK-MERGE programs.

Dijkstra (quoting a discussion with several colleagues) argued that the need to traverse FIND-paths twice made collapsing rather inelegant from a programmer's point of view As a possible remedy Dijkstra proposed an algorithm due to Rem, but our analysis will show that in this particular case the efficiency of the original algorithm is lost (i.e. as a general method for CHECK-MERGE programs).

In this paper we shall consider the problem of how the "second pass" of the collapsing strategy can be eliminated while preserving good efficiency (with or without balancing for UNION or MERGEs). We consider several alternative tree compression methods, and show that in worst case none are as good as expected. We propose a simpler, one-pass technique which performs well in both the unbalanced and balanced case. Using this technique (called path-halving) the cost for $O(n)$ UNIONs and FINDs is bounded by $O(n \log n)$. Using both path-halving for FINDs and balancing for UNIONs we prove that the cost for $O(n)$ UNIONs and FINDs reduces to $O(n \log^* n)$, again a "practically linear" bound. We conclude that path-halving is a viable alternative to the collapsing rule, for all practical purposes.

Whereas the algorithms are quite easy to present, it appears that much of the insight of previous problems for UNION-FIND programs is needed in the analysis. Thus, we shall be using just about every technique known from these problems in a more or less nontrivial form. The proof of the $O(n \log^* n)$ result is very similar to the argument of Hopcroft and Ullman [5] (see Aho, Hopcroft and Ullman [1]), and the only surprising aspect is the fact that their argument indeed remains valid for this weaker algorithm. A more detailed analysis like Tarjan's (see [8], also [10]) has not led us to a better result, and we conjecture that $O(n \log^* n)$ is the best possible bound for $O(n)$ UNION-FIND instructions using the weighted union rule and path-halving.

2. Some one-pass techniques.

Finding good implementations for CHECK-MERGE programs is intimately related to the problem of finding good strategies for executing UNION-FIND programs, because

CHECK (x,y) is equivalent to
$$\left[\begin{array}{l} q_1 = \text{FIND}(x) \\ q_2 = \text{FIND}(y) \\ \text{is } q_1 \text{ equal } q_2 \text{ ?} \end{array}\right.$$

MERGE (x,y) is equivalent to
$$\left[\begin{array}{l} q_1 = \text{FIND}(x) \\ q_2 = \text{FIND}(y) \\ \text{UNION}(q_1, q_2) \end{array}\right.$$

There appear to be essentially two different approaches for finding good "one-pass" collapsing strategies : improving the straightforward FIND-implementation while a "promising" rule for UNIONs is fixed, or improving the straightforward implementation of UNIONs or MERGEs while a rule for FINDs is fixed. In this section we shall explore the first approach, in fact assuming throughout that UNIONs are implemented follo wing the weighted union rule.

Let us recall the strategy for FIND (x) when the full colla sing rule is used (see fig 1).

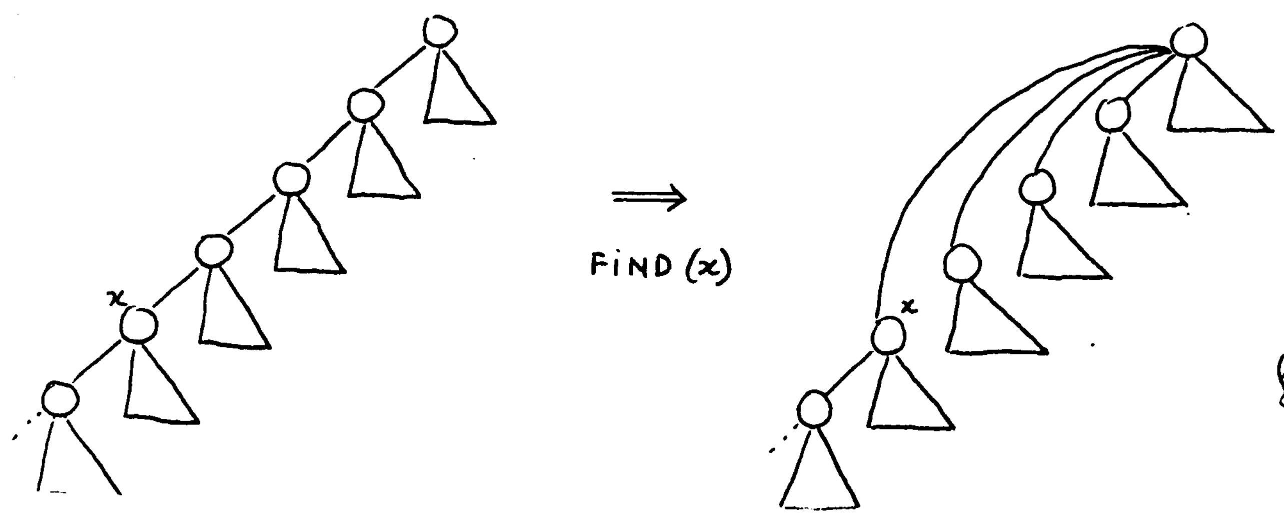


fig 1

One must traverse the FIND-path (from x up the tree) to locate the root first, and then traverse the very same path a second time to attach all its nodes to the root. As Tarjan's result [8] indicates that the average length of FIND-paths rapidly decreases, the effort to eliminate the "second pass" from the collapsing rule is not so much a matter of improving efficiency as it is a matter of pure programming elegance. We shall study some simple methods which attempt to create the same (approximate) effect as the full collapsing rule in a single pass.

As a first try one might implement the following (apparently reasonable) technique:

> <u>Strategy</u> A. Allocate a free dummy node. As you traverse the FIND-path up the tree, attach each of its nodes to the dummy. If the root is reached, attach the dummy to the root.

Fig 2 illustrates the strategy, with dummy nodes marked as ⊗.

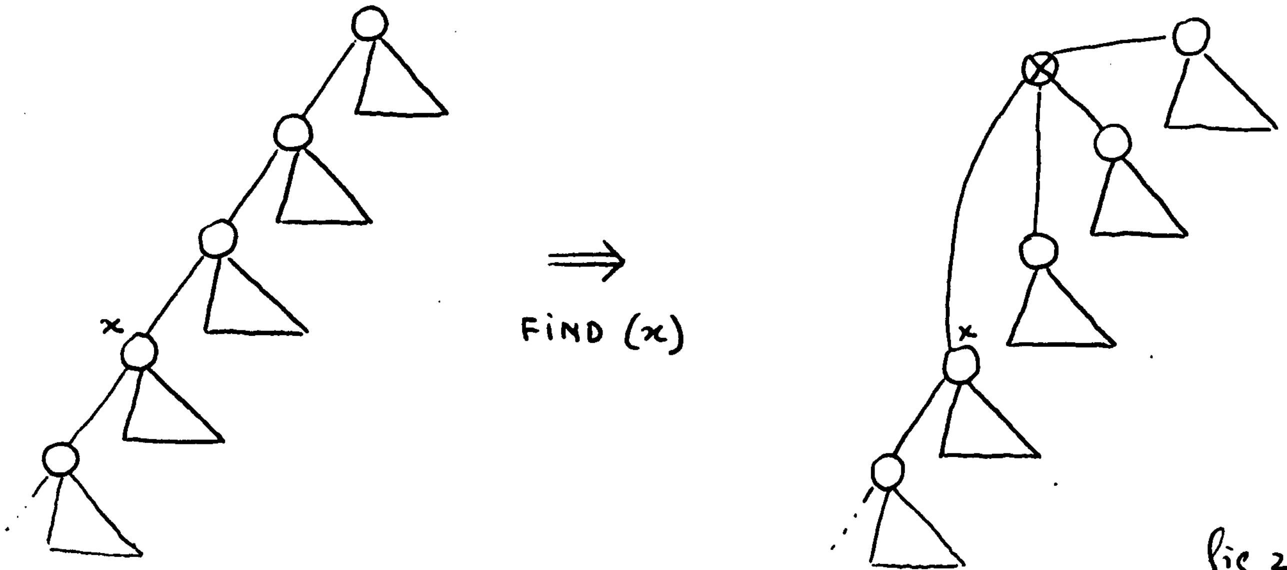


fig 2

At first sight strategy A (together with balancing for UNIONs) shouldn't leave us far from the excellent performance of e.g. Tarjan's algorithm. Some analysis will show that the programmer's intuition is wrong here, and we shall prove in section 3 that $O(n)$ UNIONs and FINDs may cost as much as $\Theta(n \log n)$ time. The most serious threat to a good performance of strategy A is the introduction of a new dummy node into the forest with every FIND. The dummy node tend to cluster in the toplevels of the tree, and keep set-node from getting nearer to the root as more FINDs are being performed. In fig 3 is indicated how the dummy nodes remain an impenetrable field which merely changes shape when a FIND(x) is performed.

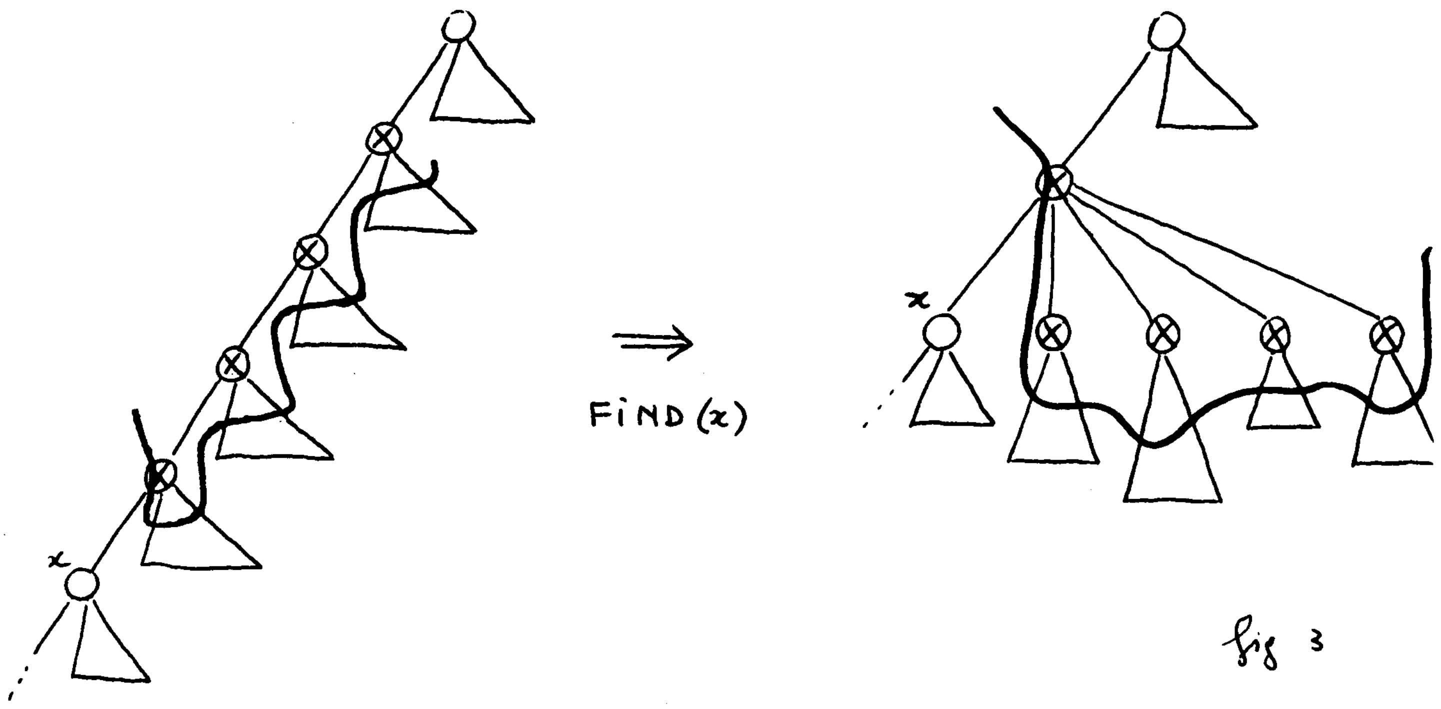


fig 3

As more and more dummy nodes are getting in the way of set-nodes, the effect can be that the cost for future FIND-increases rather than decreases. See fig 4, where repeated execution of FIND(x) instructions can make the cost of a subsequent FIND(y) arbitrarily large.

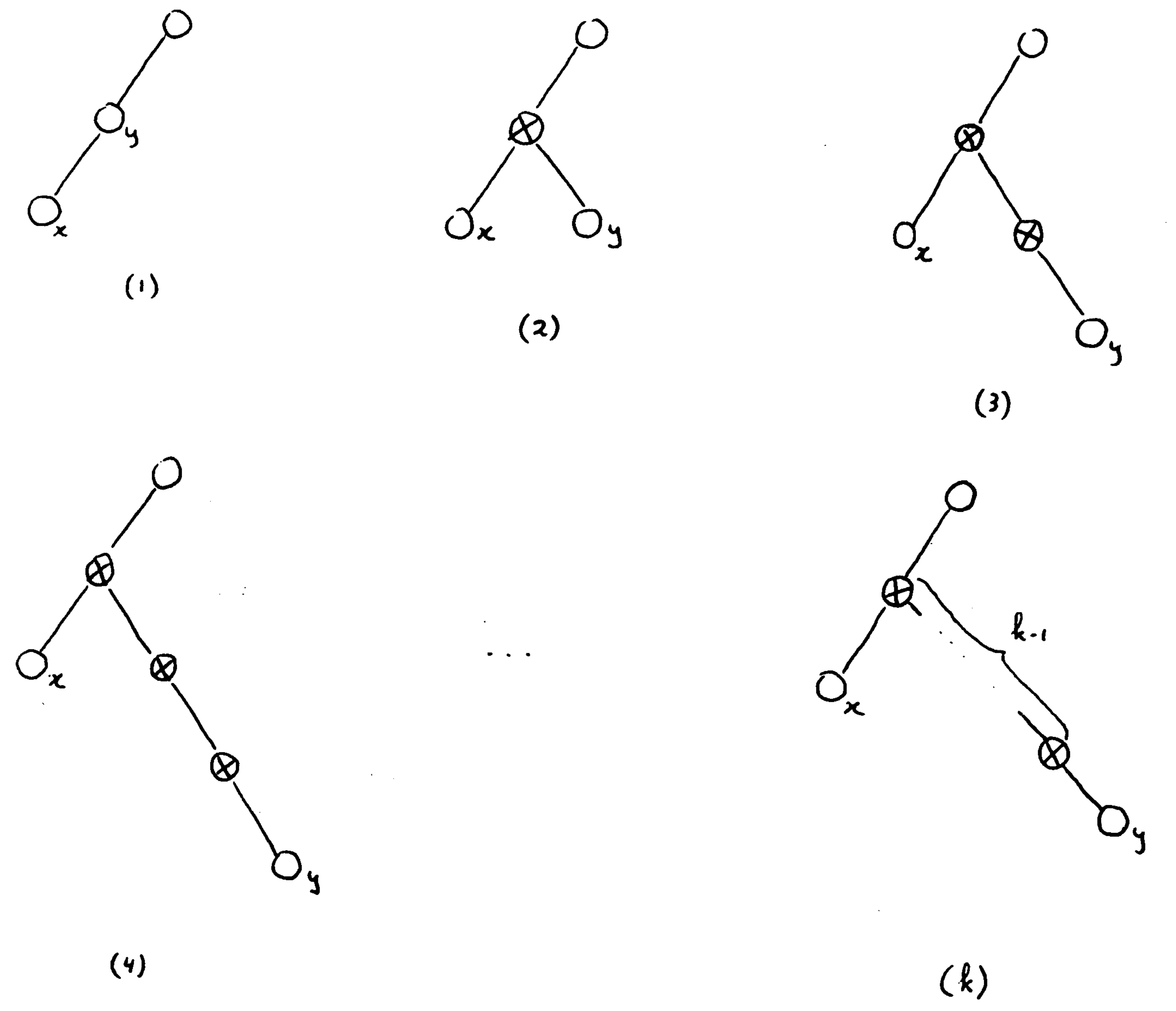


fig 4

There are several conclusions, which will be supported also by the further analysis in section 3. First, it could very well be that the initial tree of fig 4 is part of a balanced tree, built with UNIONs following the weighted union rule (see fig 5.a for an example). After performing k-1 FINDs on x the resulting tree with dummies will be highly unbalanced!

(see fig 5. b). It appears that strategy A is undoing the balance which UNIONs try to maintain so carefully.

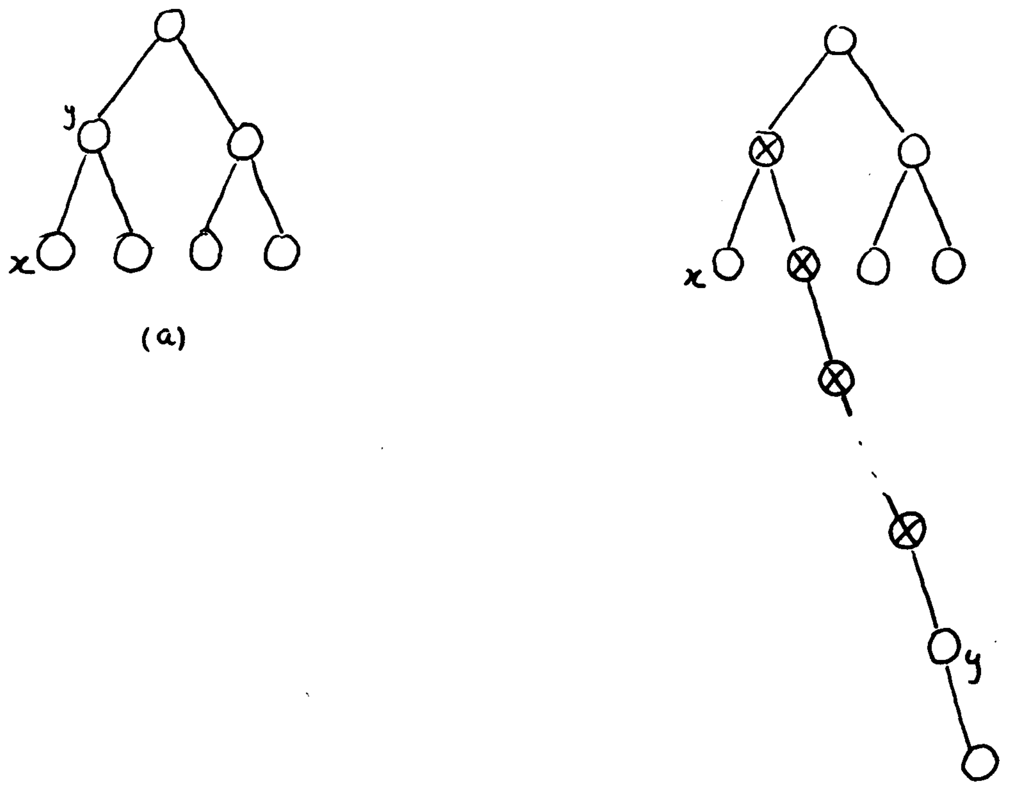


fig 5

A second, even more disastrous conclusion is that the number of dummy nodes will be increasing proportional to the number of FINDs and thus be potentially "unbounded" in terms of the size of the original set! This makes th use of strategy A prohibitive in any practical algorithm, in particular when there is no a priori bound on the number of FINDs.

We note that the cost of performing the $\text{FIND}(x)^{k-1}$; FIND(y) program in fig 4 nevertheless remains "only" $O(k)$, and we shall need a more precise analysis if strategy A is to be defeated also for its time-complexity. Such a result is of importance, because there are several modifications of strategy A which have the same (or comparable) time-complexity but avoid that a new dummy is needed for every FIND.

Observe that by the time more than n-1 dummy nodes have appeared in the tree (where n is the number of set-nodes), some dummies must have sifted down to the bottom of the tree (fig 6.a) and play no active role any-more in "keeping the set together" or chains of dummy nodes with no branching must have formed in the interior of the tree (fig 6.b). Dummies which become leaves should be detached and send to the "free pool", whereas linear chains should be collapsed as soon as they are formed and dump their intermediate dummies into the free pool also. It is not immediately obvious to find which dummies can be recycled at a particular moment: one may encounter a linear chain along the FIND-path (as for FIND(y) in fig 6.b) or all chains may remain "invisible" for the FIND (as for FIND(x) in fig 6.b). Note that when a FIND is executed one dummy is used from the free pool, thus it is sufficient to locate one dummy which can be recycled to keep the supply of dummies up. If we succeed, then the

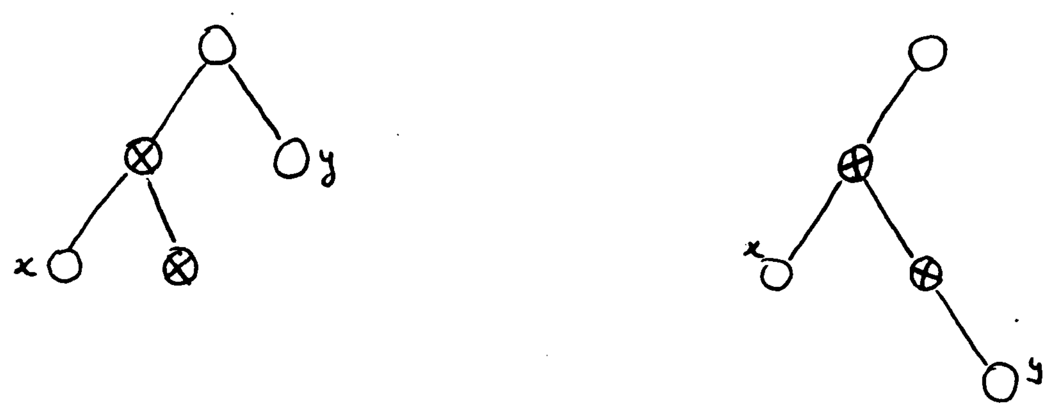


fig 6

number of dummies needed for strategy A can actually remain $O(n)$. The solution of the recycling problem is the main constituent of strategy B developed below. The solutio could be formulated in precise terms for reference machines.

Proposition 2.1. The recycling problem for strategy A can be solved without increasing the cost of FINDs (strategy B).
Proof.

Each node will carry a reference count and a pointer to the beginning of the doubly linked list of its sons. Th each node-record will have the following format:

name	ref. count	↑ to father	↑ to son-list	↑ to left sibling	↑ to right sibling

When a UNION is executed, the root of one tree is added to the son-list of the root of the other tree (with the proper update of the father-field of one and the ref. coun field of the other). See fig 7.

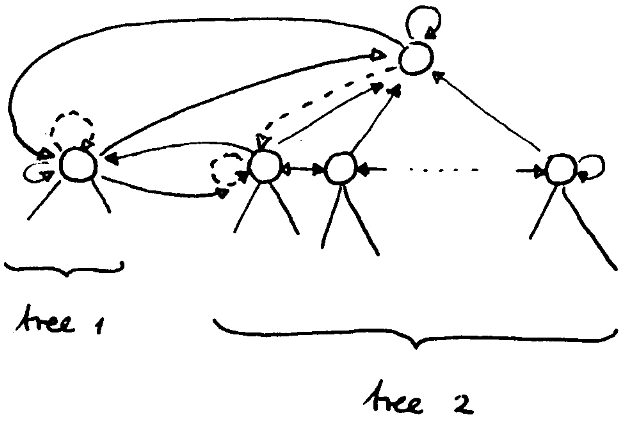


fig 7

For a FIND we begin by allocating a fresh dummy D from the free pool, and start the process of stepping throu the son-lists of nodes on the FIND-path. The unmodified strategy A would maintain the structure and treat sons as follows

(i) the son is deleted from the son-list of its father, meanwhile closing up the gap in the doubly linked list and decrementing the ref. count of the father by 1

(ii) the son is reattached and appended to the son-list of D, whose ref. count is subsequently incremented by 1.

and the process is repeated for the father, continuing until the root is reached. Finally D is appended to the son-list of the root.

The process will remain unaltered when set-nodes are treated, but for strategy B we modify the treatment of dummy nodes along the FIND-path as follows

(ii)' after deleting the dummy from its father's son-list we inspect its ref. count:

- if its ref. count is 0, then we do not attach it to D but send the dummy back to the free pool
- if its ref. count is 1, then we find its (only) son, attach this son to D and send the dummy itself to the free pool
- if the ref. count is > 1, then the dummy cannot be missed and should be attached to D as usual.

Note that a tree is transformed only along the FIND-path, so if chains are created they are immediately detected and collapsed.

□

The amount of bookkeeping needed in an implementation of strategy B is rather large, and thus the only virtue of the method is an "existence proof" showing that the use of dummy nodes in strategy A can be fully optimized. There is a simple further modification in which hardly any bookkeeping is needed at all, at the (acceptable) cost of having a full set of n dummy nodes in use at all time. No free pool has to be maintained.

The idea is to modify the initial forest of singletons, and give each set-node a dummy father (fig 8). UNIONs

fig 8

are executed in the usual way, following the weighted union rule. At all times the forest will consist of trees whose leaves are set-nodes and interior nodes are dummies. For FINDs one could implement the following technique

<u>Strategy</u> C. Follow strategy A, but in stead of allocating a fresh dummy node we use the father (a dummy) of the set-node on which the FIND is to be performed.

<u>Proposition</u> 2.2. Strategy C preserves the property that leaves are set-nodes and interior nodes are dummies.
<u>Proof</u>.
The property is easily observed from fig 9.

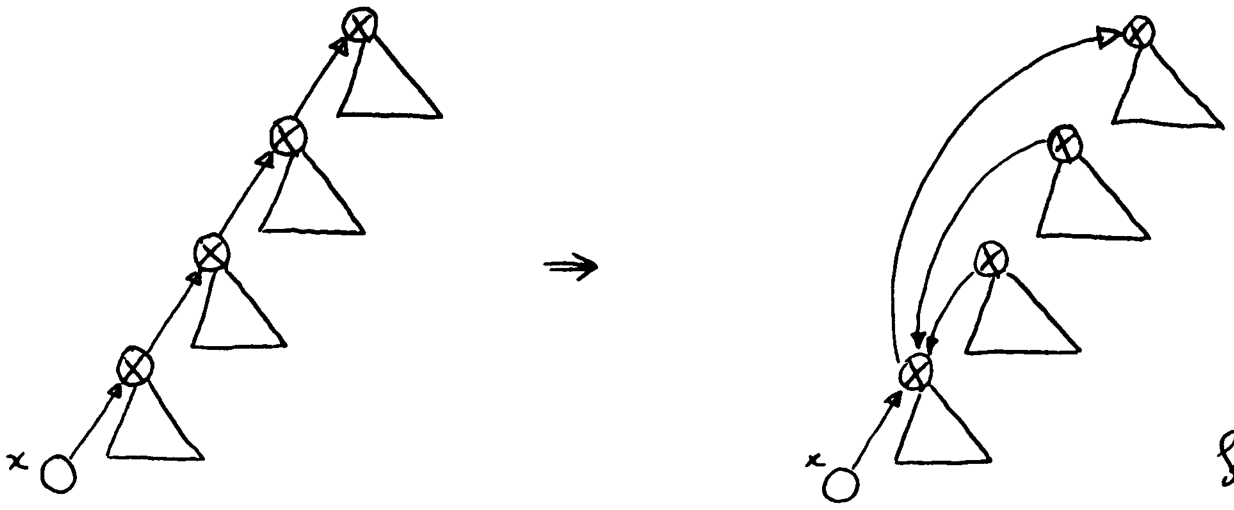


fig 9

□

Fig 9 already indicates that the "efficiency" of strategy C is likely to be deceiving. The tree is almost completely thrown off balance. Perhaps the ultimate simplication of strategy C is the following technique, which needs no dummy nodes at all!

> Strategy D. Follow strategy A, but in stead of allocating a dummy node use the very node on which the FIND is performed.

Strategy D is "one pass" in the easiest possible way: as we traverse the FIND-path starting at x, each of its nodes is reattached to x until we get to the root (where

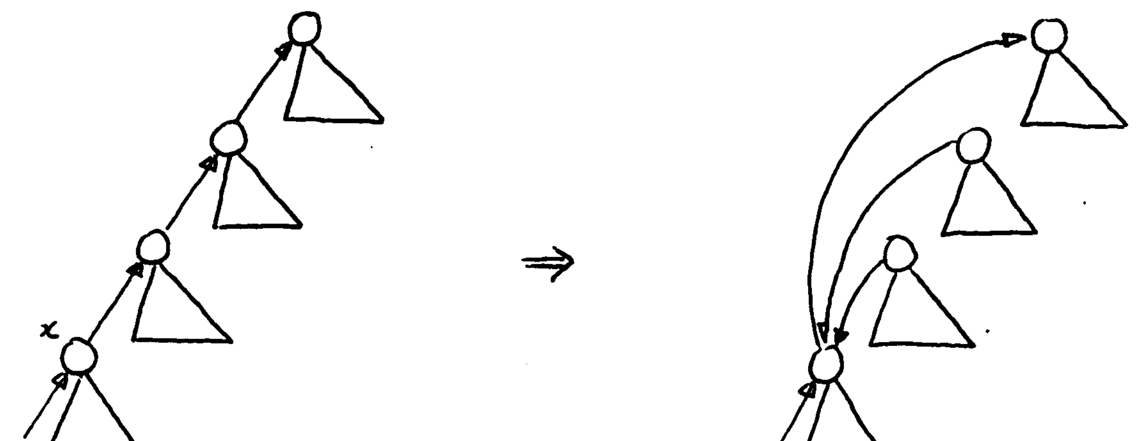


fig 10

we stop and attach x). See fig 10. In terms of storage-space requirements strategy D is certainly the simplest method we have seen, but its ominous effect on the balance of a tree will generally keep the time-complexity unfavorable.

Examining strategies A to D shows that each one in its own way tries to improve the path-length for future FINDs on nodes in the subtrees along the FIND-path. This is indeed achieved for all subtrees, with the notable exception of the last subtree on the FIND-path just before the root is reached. _The path-length to the root for all nodes in this one particular subtree actually increases by_ 1. As this is a major contributing factor in the unbalancing effect of each strategy we have seen thus far, one should modify them all to look one node ahead as the FIND-path is traversed in order that the last node below the root can be detected in time and saved from the reattachment step. Strategies A and B are in worst case not extremely sensitive for this improvement, but there will be good reasons to modify strategies C and D.

Strategies C and D are peculiar as they violate the _principle of monotonicity_, which claims that a viable collapsing strategy should not mix up the hierarchical ordering of nodes. The effect of not attaching a node to one of its ancestors usually is a destruction of the interior balance, and the analysis of such methods tends to be rather complicated. It will appear that the modified strategies at least improve a tree in some sense, and should be used if these technique are to be applied in practice.

3. A worst-case analysis of strategies A to D

Although we shall see better methods later, it is of interest to determine how well the simple strategies of section 2 perform. It will appear that the worst case time-complexity for $O(n)$ UNIONs and FINDs is about equal for either strategy, and the obvious conclusion within the limits of this complexity model indicates that one might as well choose for strategy D at this stage because it has the easiest implementation of all. The analysis of the other strategies, however, leads to some original questions concerning binomial trees, which are of considerable interest in their own right.

Strategy A was notorious for its continuous need for additional dummies, and its storage-complexity was shown to be proportional to the number of FINDs performed. We have seen that this can be repaired to large extent (strategy B), so we have yet to find an argument to reject strategy A for its (worst case) time complexity. We will show that strategy A leads to an achievable worst-case complexity of $O(n \log n)$ for $O(n)$ UNIONs and FINDs. In establishing the upperbound we make use of the technique of "algorithm reduction", i.e. we show that the cost of building a forest using balancing and strategy A is no greater than the cost of another program using full collapsing (but no balancing) which builds some other forest (on the very $n+m$ points)

Lemma 3.1. For any program performing $n-1$ UNIONs and m FINDs using balancing and strategy A one can construct a "bounding" program which performs $n+m-1$ UNIONs and m FINDs using the full collapsing rule (but no balancing)

Proof.

Given any program P of s ($\leq n-1$) UNIONs and m FINDs, we shall prove by induction on the total number of instructions (k) that a program P′ can be assembled which builds a perhaps not identical forest on n + m points by performing s + m unbalanced UNIONs and m fully collapsing FINDs in such a manner that the cost of FINDs in P is bounded by the cost of related FINDs in P′.

The induction basis ($k = 1$) is trivial. Assume that the induction hypothesis holds for k, and consider a program P of k+1 instructions using balancing and strategy A. Write $P = Q ; \sigma$, where σ is a single UNION or FIND. By hypothesis we can assemble a "bounding" program Q′, which uses full collapsing but no balancing and no smaller cost for related FINDs in Q. Distinguish the following cases for σ.

Case a: σ is a UNION

We let P′ perform the same instruction, i.e. $P' = Q' ; \sigma$

Case b: σ is a FIND on a ~~root~~

No dummy is introduced, and we take again $P' = Q' ; \sigma$.

Case c: σ is a FIND on a node of depth ≥ 1.

Whereas Q′ does not necessarily build the same forest as Q, we know from the induction hypothesis that the depth of any node x in the forest of Q′ is no smaller than the depth of this node in the forest of Q. Let $\sigma = \text{FIND}(x)$, and let fig 11.a represent the status after Q′. We know that the depth of x is at least as large here as it would have been in the forest after Q.

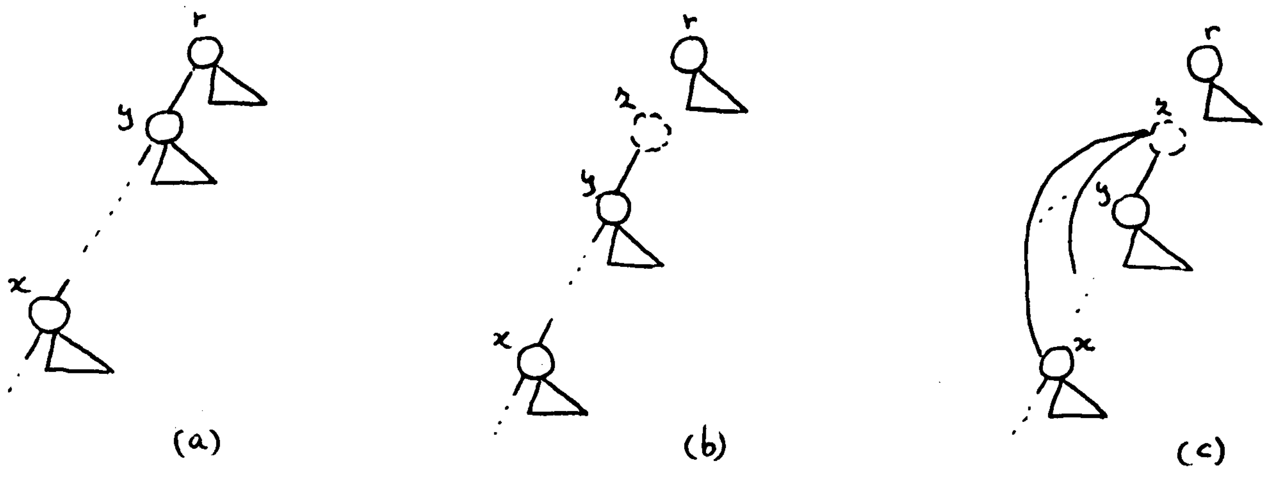


fig 11

Let y be the node on the FIND-path just below the root. Look in Q' for the instruction τ which made y a son of r. This τ is either a UNION or a FIND on a former descendant of y. In the latter case y was (at that time) in the subtree of a son y_1 of r. Iterating this same procedure on y_1, y_2, ⋯ we must eventually find the former ancestor y_t of y which was responsible for bringing y below r in a UNION(y_t, r) instruction. Let z be the dummy node which σ would have introduced in Q. Replace the UNION (y_t, r) instruction of Q' by UNION (y_t, z), in the unbalanced sense. Obviously it does not change the cost of all later FINDs on nodes in the subtree of y_t: in stead of stepping onto r in the end, we step on z. In other words, z "behaves" as if it were r for these nodes and the induction hypothesis for the modified Q' (Q'') remains valid. See fig 11. b.

Now consider P' defined as

$$P' = Q'' ; \text{FIND}(x) ; \text{UNION}(z, r) ;$$

The FIND(x) costs at least as much as it would have in P, and after the UNION(z, r) we have essentially the same sets and hierarchical relationships within the trees as we would have had with P, except that many subtrees may be stuck "lower" than before. The chief reason is that i Q" many nodes got no farther than z, whereas they we originally attached to r.

□

From the construction in 3.1 it is obvious how we loose the balance when strategy A is used: it is the "vir tual" UNION to insert the single dummy each time. The upperbound on the complexity of strategy A is now easily deduced, as the analysis of the full collapsing rule is known (see sect 1, Paterson [7], Tarjan [8], or van Leeuwen [10]).

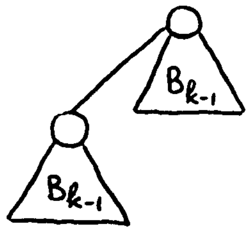
Theorem 3.2. Any program of n-1 UNIONs and m FINDs using balancing and strategy A runs in time $O(n + m \log(n+m))$.

It will be obvious from 3.1 that 3.2 remains valid eve if we do not use balancing in conjunction with strategy A!

In order to show that the bound in 3.2. cannot be improved, we shall use (balanced) UNION-instructions t build a particular tree B_k with 2^k nodes ($2^k \sim n$) in which a large number of "more and more" expensive FIND can be performed.

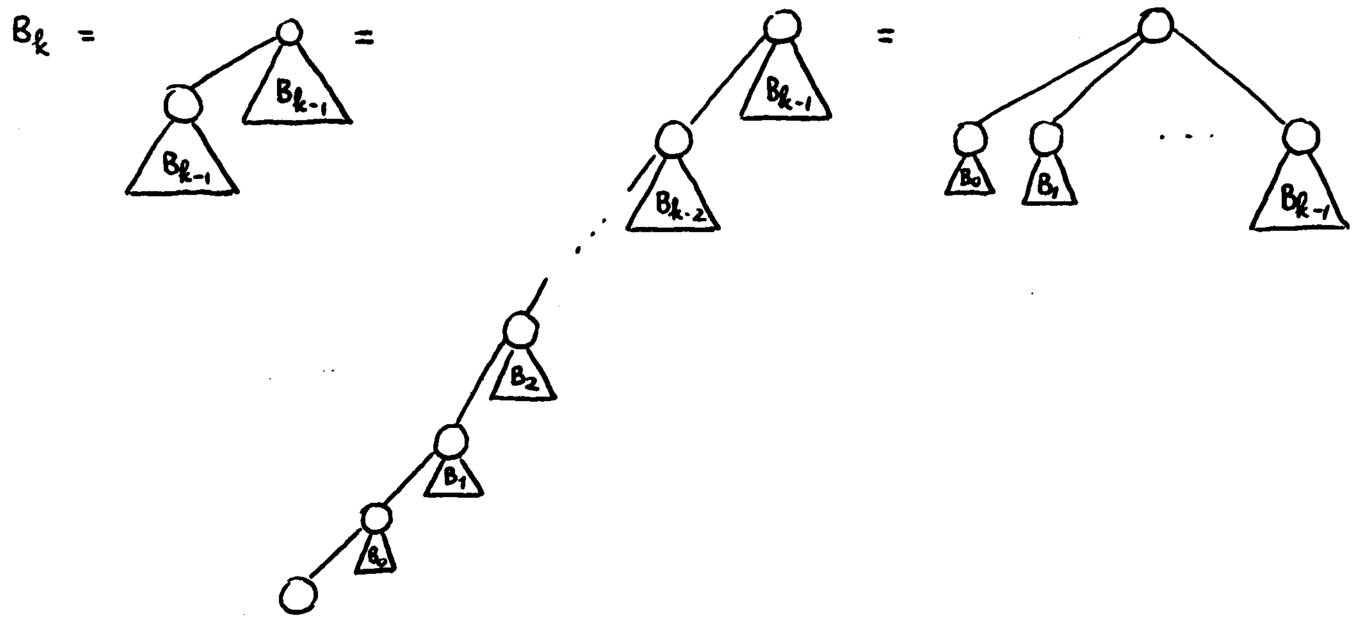
Definition (i) B_0 = O (a single node)

(ii) B_k =



The "binomial" trees B_k were introduced by M. Fischer [4], in his analysis of the full collapsing rule. Note also that the trees are precisely the kind of structures underlying Vuillemin's binomial queues ([11], see also Brown [2]). We need the following property of B_k-trees, easily provable by induction.

<u>Lemma</u> 3.3. Each B_k-tree may be arranged as

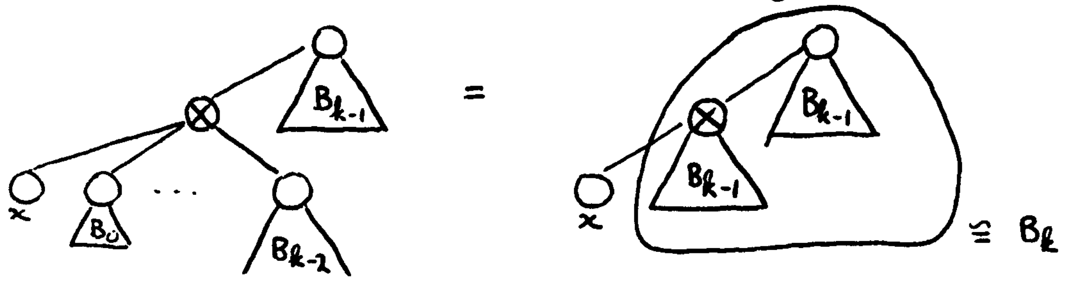


It should be obvious that a B_k-tree has 2^k nodes, and that it can be constructed by means of some 2^k <u>balanced</u> UNIONs. Further, a B_k-tree has precisely one node of depth k (a leaf), which is commonly called the handle of the B_k-tree.

<u>Lemma</u> 3.4. Using strategy A, a FIND instruction on the handle of a B_k-tree leads to a tree containing another B_k.

<u>Proof</u>.

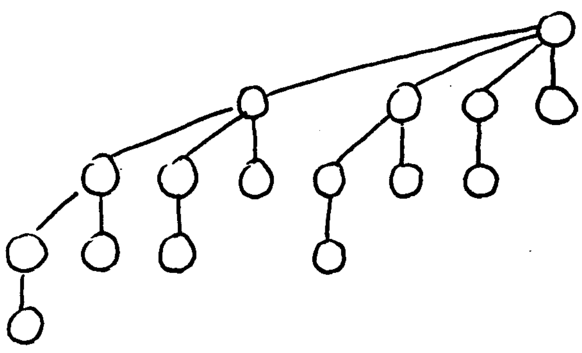
Performing the FIND on B_k gives

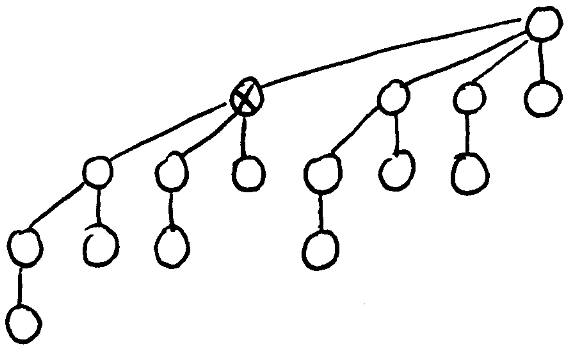


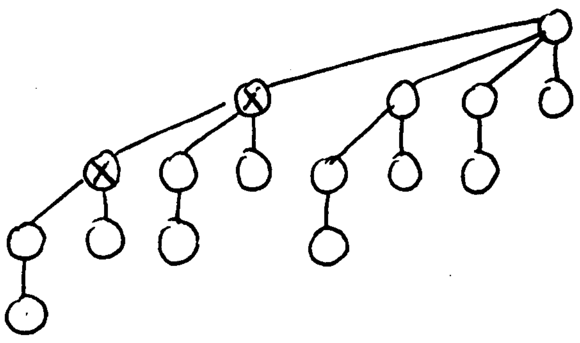
where x is the handle of the original B_k-tree.

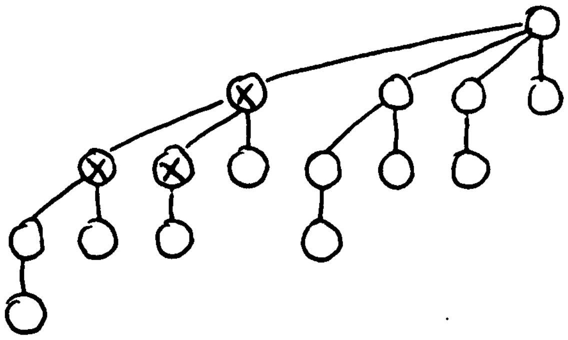
■

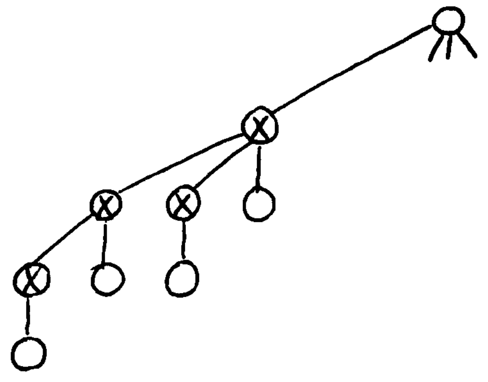
Ignoring x after a FIND as in 2.3 gives another B_k (with a new handle), and it is tempting to argue as follows. Apparently we can perform FINDs of cost k as long as we want. Thus $O(n)$ UNIONs and FINDs (with $2^k = n$) will require time $\sim 2^k \cdot k = n \log n$. This reasoning is correct only if at all times a handle can be chosen which is not a dummy node (because we shall obviously never do a FIND on such a node). The following example shows how a B_4 is transformed, with dummy nodes filling the "body" of the tree more and more The "handle node" which got attached to a new dummy is omitted in each snapshot

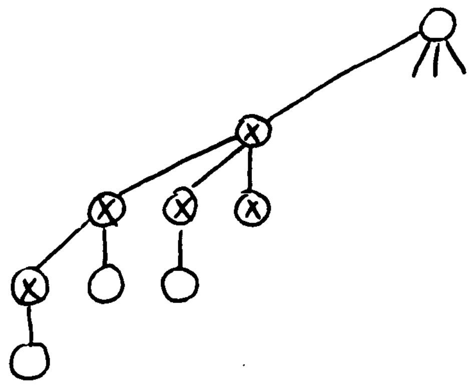

(0)

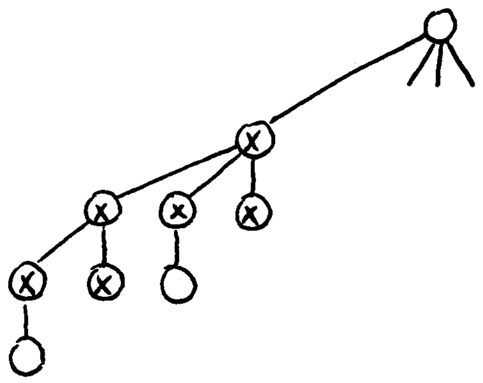

(1)

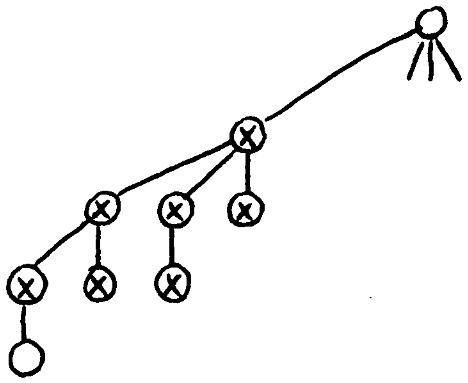

(2)


(3)


(4)


(5)


(6)


(7)

and B_8 has all $\otimes$'s.

Thus on B_4 one can perform $\frac{1}{2} \cdot 2^4$ "costly" FINDs. It becomes clear also that the number of dummy nodes needed can grow linearly with the number of FINDs performed. Before we discuss this fact further, we shall formulate the property of B_4 in general as yet another remarkable property of B_k-trees. We introduce the marking operation as an explicit operator, and it on B_k-trees.

<u>Definition</u>. Given a tree T and a node $x \in T$ ($x \neq$ root), the operation MARK(x) is performed as follows: introduce a new marked node, make all ancestors of x except the root a son of the marked node, omit x and make the marked node a son of the root.

By 3.4 we know that a MARK on the handle of a B_k-tree yields another B_k-tree with one more marked node, although we have yet to show that the marking pattern doesn't get reordered more drastically. If we begin with an unmarked B_k, then repeating MARKs on its handle introduces a marking in the left B_{k-1} subtree as follows (where indeed the marking at step $i+1$ is an extension of the marking at step i):

- mark the root of the B_{k-1} subtree
- mark its unmarked nodes which are the root of a B_{k-2} tree
-

⋮

- mark its unmarked nodes which are the root of a B_{k-i} subtree, where these nodes are enumerated per level from left to right proceeding from the high to the lower levels
-

⋮

Before we prove more facts, we need an additional lemma about how a B_k-tree is obtainable from a B_{k-1} tree.

<u>Lemma</u> 3.5. Any B_k is obtained from a B_{k-1} by attaching a leafnode to each node of the B_{k-1} tree. Conversely, omitting the leafs from any B_k-tree yields a B_{k-1} tree.

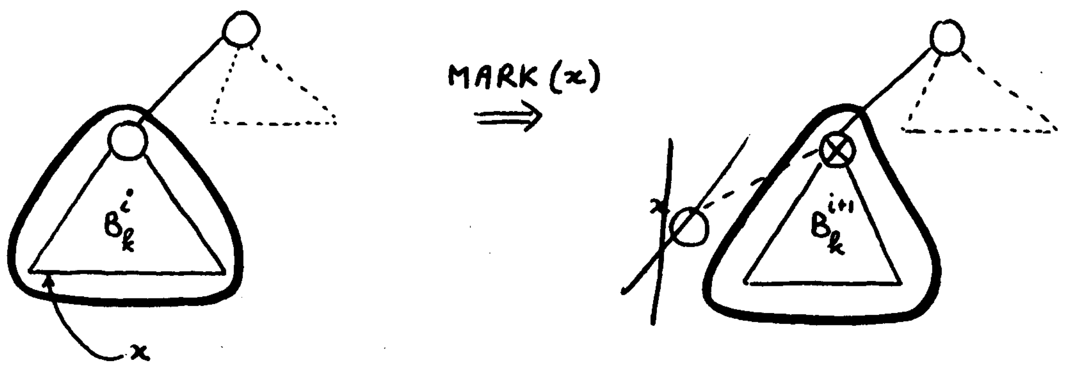
We shall demonstrate the progression of a marking in B_k in two ways.

<u>Lemma</u> 3.6 An initially unmarked B_k is turned into a B_k with a fully marked left B_{k-1} subtree after performing

exactly 2^{k-1} MARKs on its (ever changing) handle. Each such MARK has cost k, and each time the new handle is an unmarked node!

First proof of 3.6.

Our first demonstration of the lemma makes use of the traditional decomposition of B_k in lower order trees (as in 3.3). We shall define and inductively construct for each k a sequence $B_k^0, B_k^1, \dots, B_k^{2^k}$ such that for $i < 2^k$ a tree B_k^{i+1} is precisely obtained after performing a MARK on the handle of a B_k^i in the following sense:



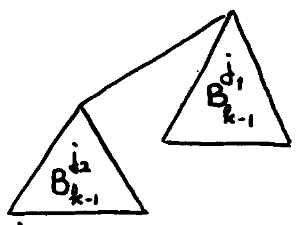
(In particular, B_k^0 is an unmarked B_k and $B_k^{2^k}$ will be a fully marked B_k).

Define

(i) $B_0^0 = \bigcirc$ (a single node)

$B_0^1 = \otimes$ (a single marked node)

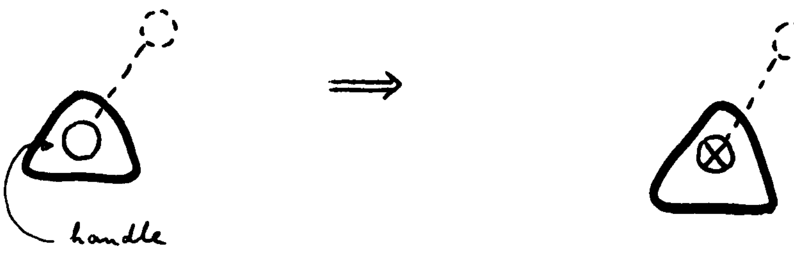
(ii) for each k and $i \leq 2^k$

$B_k^i =$  where $j_1 = \lceil i/2 \rceil$, $j_2 = \lfloor i/2 \rfloor$

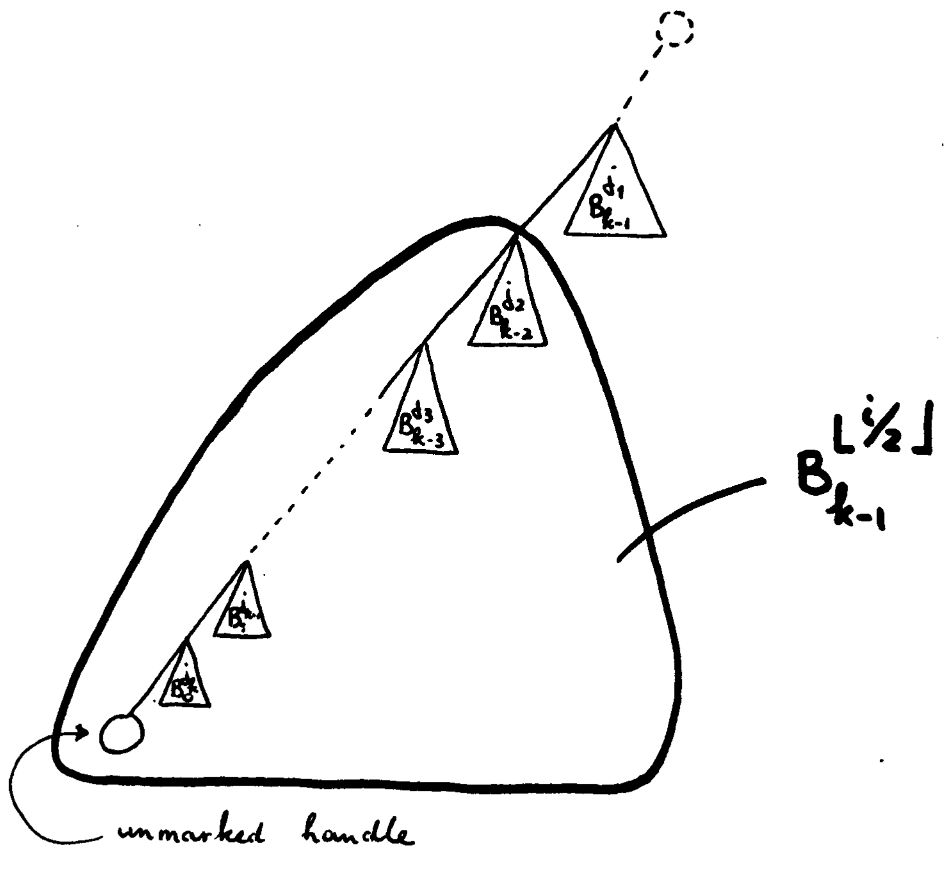
(the handle of B_k^i will be in the lower B_{k-1} subtree)

It is easy to show by induction on k that (as we migh
expect) the root of a B_k^i is a dummy for $i \geq 1$, but the handle of any B_k^i is not a dummy for $i < 2^k$ (which is pr
cisely what we wanted). Note that a B_k^i contains exactly i du
mies. It remains to be shown that a B_k^i transforms into a B_k^{i+1} as anticipated, after performing a MARK on its handle. W
proceed by induction on k.

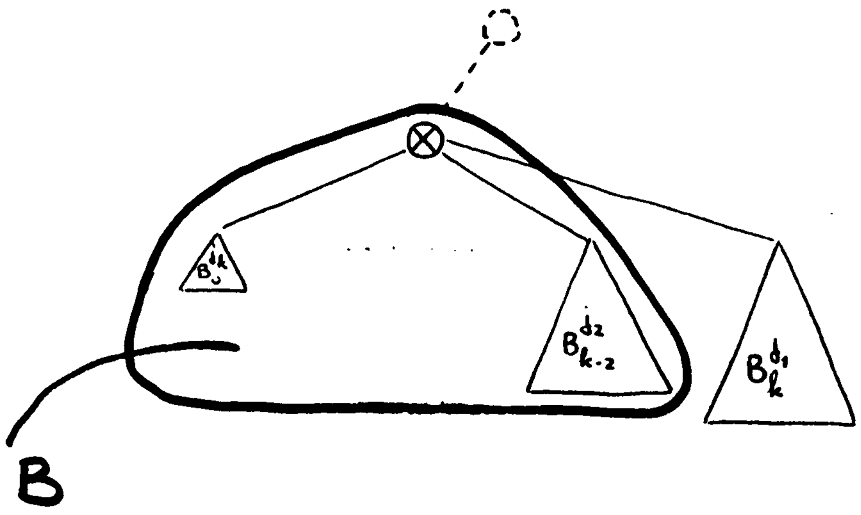
For $k = 0$ we observe that indeed



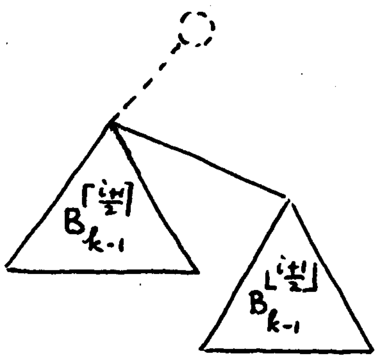
Assume the hypothesis holds for $k-1$, and consider how a B_k^i is transformed for some $i < 2^k$. From the definition we see that B_k^i may be written as follows (see also 3.3)



with $d_1 = \lceil i/2 \rceil$, $d_2 = \lceil i - d_1/2 \rceil$, etcetera. After a MARK on the handle we obtain



where B is the tree which would have been obtained after performing a similar operation on $B_{k-1}^{\lfloor i/2 \rfloor}$. From the induction hypothesis we conclude that $B = B_{k-1}^{\lfloor i/2 \rfloor + 1} = B_{k-1}^{\lceil \frac{i+1}{2} \rceil}$. Observing that $d_1 = \lceil i/2 \rceil = \lfloor i+1/2 \rfloor$, the tree may be written as

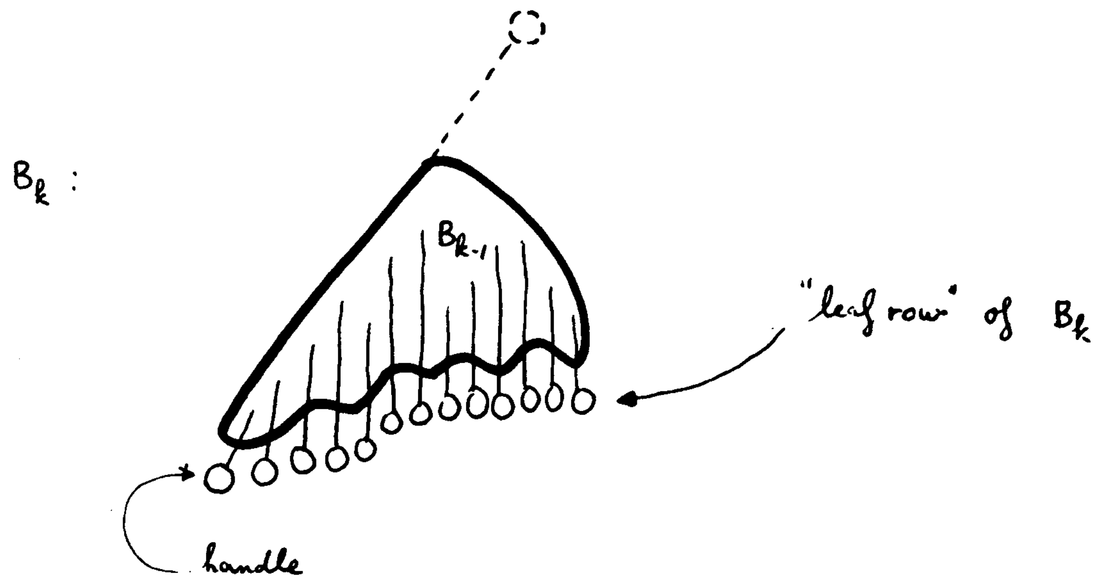


$B_{k-1}^{\lceil \frac{i+1}{2} \rceil}$
$B_{k-1}^{\lfloor \frac{i+1}{2} \rfloor}$

which is exactly B_k^{i+1}. This completes the proof.

Second proof of 3.6.

Perhaps an even more elegant argument to prove the lemma is based on 3.5. We mention it because the particular way of viewing B_k gives a different and valuable insight in the marking procedure. Let the "stripped" version of a B_k be the B_{k-1}-tree obtained after deleting its leaves. (Thus a B_k is reconstructed by adding a leaf back to its corresponding node in the stripped version).



Note that the handle of a B_k is attached to the handle of its stripped version, and so on.

By induction on k we shall argue that the following properties hold for the marking process on B_k :

(i) each MARK is performed on an unmarked handle

(ii) after performing 2^{k-1} MARKs on its (ever changing) handle, we have "pulled out" all nodes of the original leaf-row of B_k and replaced it by the row of its (yet unmarked) interior nodes

(iii) after 2^{k-1} additional MARKs on the handle are performed the entire tree is marked.

The argument depends crucially on property (ii), which should be precisely understood. It will become apparent that during each of the first 2^{k-1} MARKs one leaf of the <u>original</u> leaf-row of B_k (the one which currently appears in the handle position) is dropped, and an interior node of the <u>original</u> B_k is added to the leaf-row! The "replacement leaves" will not be appearing in the handle position until <u>after</u> the 2^{k-1} initial MARKs.

The ultimate effect is that after 2^{k-1} MARKs on the handle the interior of a B_k has become a fully marked B_{k-1}, yet its leaf-row still consists entirely of unmarked nodes. Observe that thus far only unmarked nodes appeared in the handle-position. Turning the crank, we can immediately conclude properties (i) and (iii). Suppose the dummies currently in the tree are colored red, and dummies introduced in future MARKs are blue. The next 2^{k-1} MARKs will transform the B_k-structure in precisely the same manner as before. The (unmarked) nodes in the current leaf-row are pulled out one after another, the red nodes move from the interior to the leaf-row, and the interior itself is filled with blue nodes. After exactly 2^{k-1} MARKs a B_k is obtained whose interior consists entirely of blue nodes and whose leaf-row consists entirely of red nodes. Thus all nodes are marked, and in particular a marked node has appeared in the handle position for the first time after $2^{k-1}+2^{k-1}=2^k$ MARKs.

It remains to argue for the correctness of (ii). We shall prove it by induction on k. As the induction basis is immediate, we shall only treat the general case of showing how the marking process proceeds and transforms a B_k in relation to the concurrent transformation of its interior (which will be intimately related to the marking of a B_{k-1}). Consider fig 12.a. A MARK on the (current) handle of a B_k has largely the same effect as a MARK on its father, which is the handle of the underlying B_{k-1}. As the leaves are completely passive in the process, it is as if we are actually performing the MARK on a B_{k-1} in which each node except the one on which the MARK is performed preserves the one B_k-leaf that it is carrying. The only distinction is that in stead of dropping the handle node (of the B_{k-1}) it is added as a leaf to the dummy which was just introduced (and which was yet to receive such a added son-leaf just like the other B_{k-1} nodes). See fig 12.b In this way the tree remains a B_k-structure, but during the first 2^{k-1} MARKs one may just as well ignore the leaves of dummies (as they will not be appearing in the handle positio of B_k).

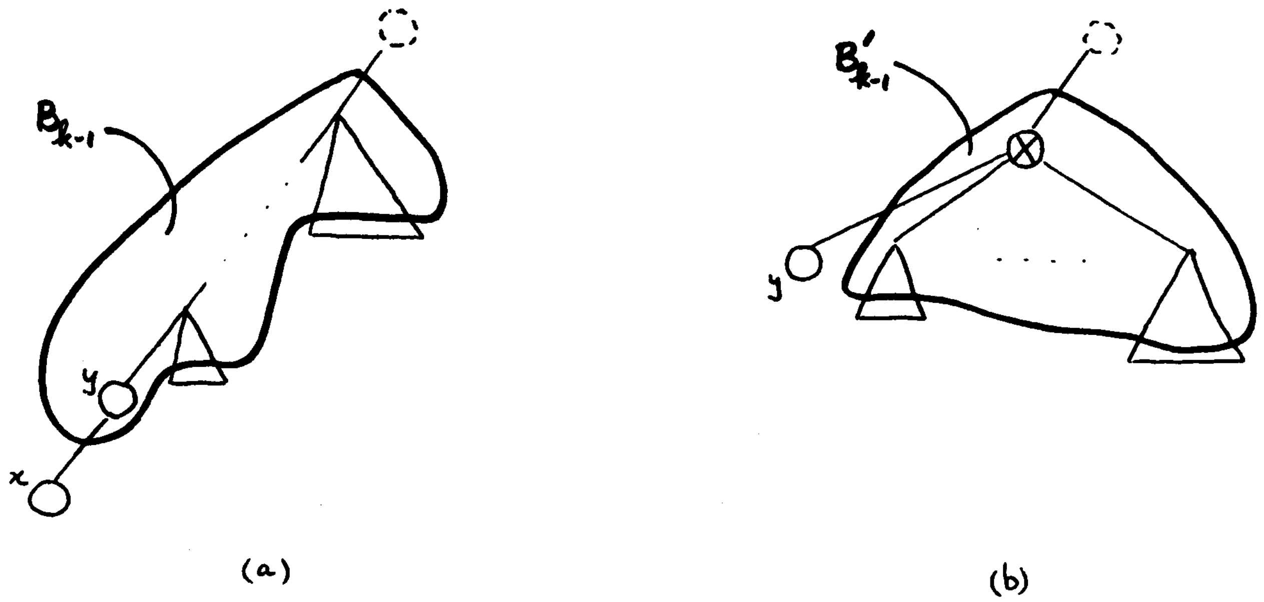


fig 12

Now observe the following. By the induction hypothesis the process makes all (unmarked) nodes of the <u>original</u> B_{k-1} appear in the B_{k-1}-handle position (y), one after another, while the B_{k-1}-structure itself is filling up with dummies. As the nodes of the original B_{k-1} are moving to the y-position, the (unmarked) leaves of B_k attached to these nodes will automatically be ap pearing in the B_k-handle position (x). It follows that the node from the leaf-row of the <u>original</u> B_k are being pulled out of the tree, one after another, during the first 2^{k-1} MARKs. After the initial set of 2^{k-1} MARKs is performed, the entire interior

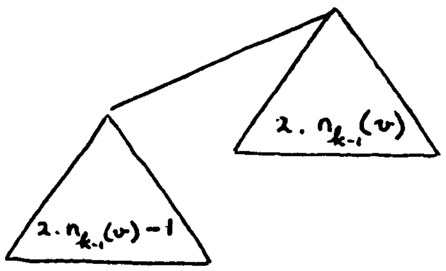
B_{k-1} consists of dummies and the first dummy is appearing in the y-position. Now recall that during the first 2^{k-1} MARKs each interior node of the original B_k (which were the nodes moving into the y-position) was re-inserted in the tree and attached to the latest dummy which got allocated. It means that we now have a B_k-structure whose leaf-row consists of the interior nodes of the original B_k. This completes the proof of the induction step.

■

The second proof of 3.6 demonstrates explicitly that the nodes of an initial B_k-structure are being pulled out of the tree one after another as the 2^k MARKs on its handle are iterated. It is natural to ask if one can determine beforehand in what order the nodes of B_k will be moving to the handle position for removal. It is not hard to show that this order is given by the numbering scheme $n_k(v)$, inductively defined from the numbering scheme of lower order binomial trees as follows

$n_0(v) = \bigcirc^1$

$n_k(v) =$



(Note that the definition follows the inductive construction of B_k from B_{k-1}-trees).

Using 3.6. we can immediately derive a lowerbound on the (worst case) complexity of strategy A.

Theorem 3.7. There are programs of n-1 UNIONs and m FINDs using balancing and strategy A which require at least $n + m \log n$ steps

Proof.

Without loss of generality we may take $n = 2^k$ (some k). Use n-1 balanced UNIONs to build a set whose "left"-subset is a B_k-tree. Recall how MARKs were derived from FINDs We immediately conclude from 3.6 that n FINDs of cost $k \sim \log n$ can be performed on distinct set-elements, in an order corresponding to the consecutive appearence of set-element in the B_k-handle position as indicated earlier.

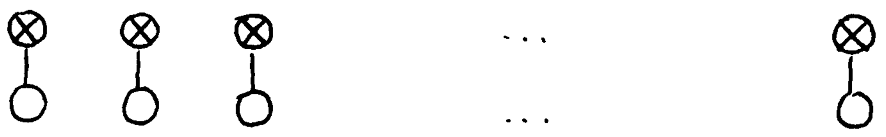
After n FINDs the entire B_k-structure is filled with dummies, and each dummy has the set-node on which the corresponding FIND was performed as an added son-leaf. Thus we have a B_{k+1} structure whose interior consists of dummies and whose leaf-row consists precisely of the original set-nodes. In the second proof of 3.6 it was shown that a next set of n MARKs (on the B_{k+1} handle this time) has the effect of pulling out all nodes of the leaf-row. It follows that we can perform a next series of n FINDs of cost $k+1 \sim \log n + 1$ on the distinct set-elements, in the order in which they are successively shifted out of the current leaf-row. After the FINDs are performed a dummy appears in the handle-position of the tree.

It is not immediately obvious how this process can be continued. We claim that the tree obtained by dropping all dummy nodes from the current leaf-row is another B_{k+1}-structure whose leaf-row contains the set-nodes again. We prove it by induction, starting from an initial B_{k+1}-structure of this form. Indeed, consider the execution of n FINDs on the handle of such a B_{k+1} and ignore for a moment that each set-node is re-inserted in the tree. Let the dummies currently in the tree be colored red, the dummies introduced during the series of n FINDs be colored blue. The effect of performing n FINDs is that the current leaf-row is pulled out and replaced by the row of red nodes, while the interior of the tree is filled with blue nodes. Dropping the red nodes yields the B_k-structure of dummies that were introduced during the FINDs. But recall that it were exactly these dummies to which we attached the set-nodes again, one after another. It follows that we have a B_{k+1} of the desired form again, after dropping the "red" nodes.

It follows that after each series of n FINDs on the set-elements (in the proper order), we can apply a next series with each instruction of cost $\sim \log n$. ■

The analysis of strategy B (with or without balancing) has little new to offer. In performing (say) $O(n)$ UNIONs and FINDs it will stay within the $O(n \log n)$ limit established for strategy A, whereas minor modifications of 3.7 show that this worst case limit is again achievable.

We now show that strategies C and D are essentially equivalent. Recall that in strategy C all set-nodes begin with a dummy father:

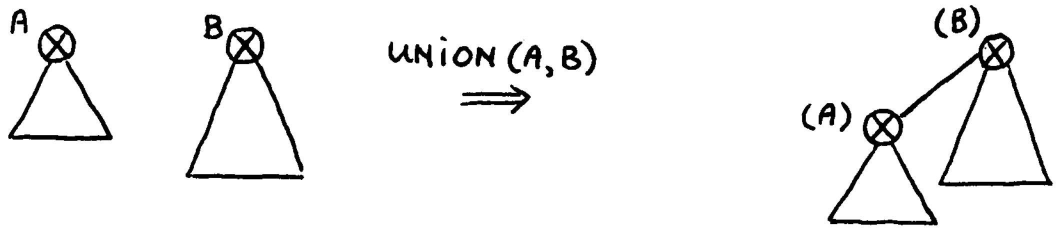


Lemma 3.8. Throughout any UNION - FIND program using strategy C and balancing, each set-node keeps the same dummy for a father.

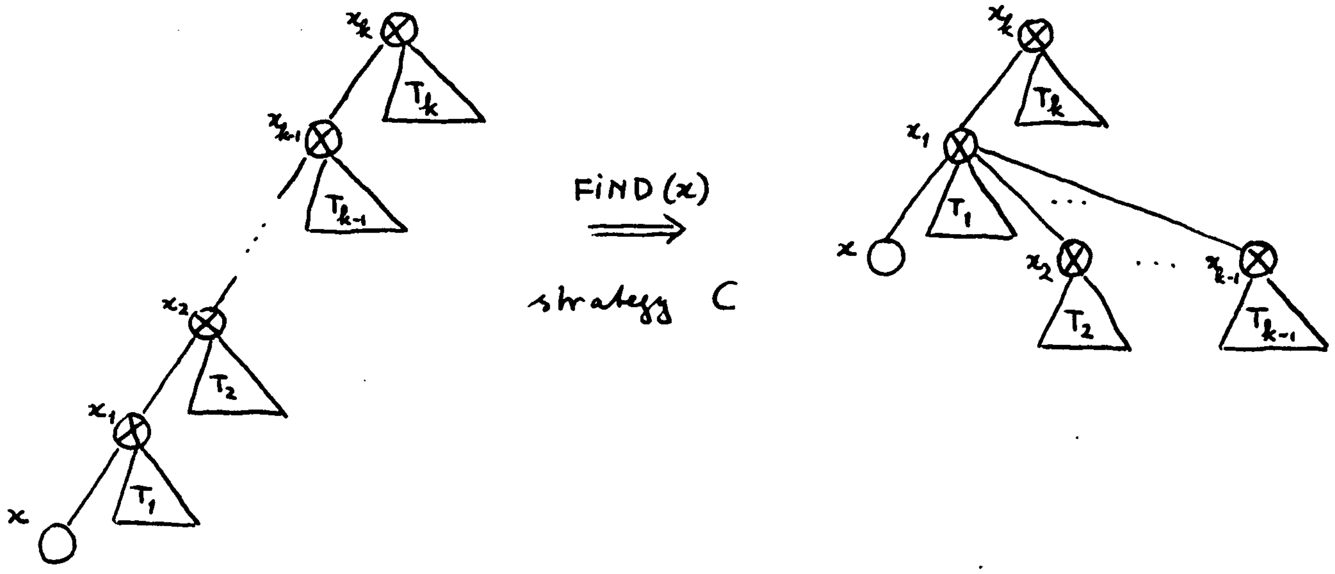
Proof.

We recall (2.2) that throughout any UNION - FIND program using strategy C the interior of trees in the forest will remain filled with dummies, while set-nodes are merely rearranged in the complete leaf-row.

It is obvious that set-nodes keep their father in a (balanced, or even unbalanced) UNION:



It also holds after FINDs, as the following figure shows (where we use that the interior of any tree at any time during the program consists entirely of dummies, in order to be sure that the FIND-path has only dummy nodes upwards).



■

From 3.8 we conclude that we can consistently identify a set-node with its dummy father throughout the program, and strategy C reduces to strategy D if we do so! Thus, it is sufficient to consider strategy D only for a precise worst-case analysis.

The last results of this section will show that $O(n)$ UNIONs and FINDs may require up to $O(n \log n)$ steps when strategy D is used, and the use of balancing will be essential here. Unlike other algorithms, it appears to be easier to establish the lowerbound on the worst-case performance than it is to determine an upperbound this time. We shall discover yet another remarkable invariant transformation for binomial trees.

Theorem 3.9. There are programs of $n-1$ UNIONs and m FINDs using balancing and strategy D which require $n + m \log n$ steps.

Proof.

Without loss of generality we may take $n = 2^k$ (some k). Use $n-1$ (balanced) UNIONs to build a binomial tree B_k, consisting precisely of all set-nodes. Consider the execution of a FIND on the handle of a B_k-tree (fig 13). It appears that strategy D leaves the B_k-structure invariant! Thus we can repeatedly perform as many FINDs on the handle of a B_k as we want, at a cost of $\log n$ per instruction.

■

In performing FINDs with strategy D the nodes in a tree are in a constant turmoil. The continuing change of ancestral relationships and disrespect for any form of interior

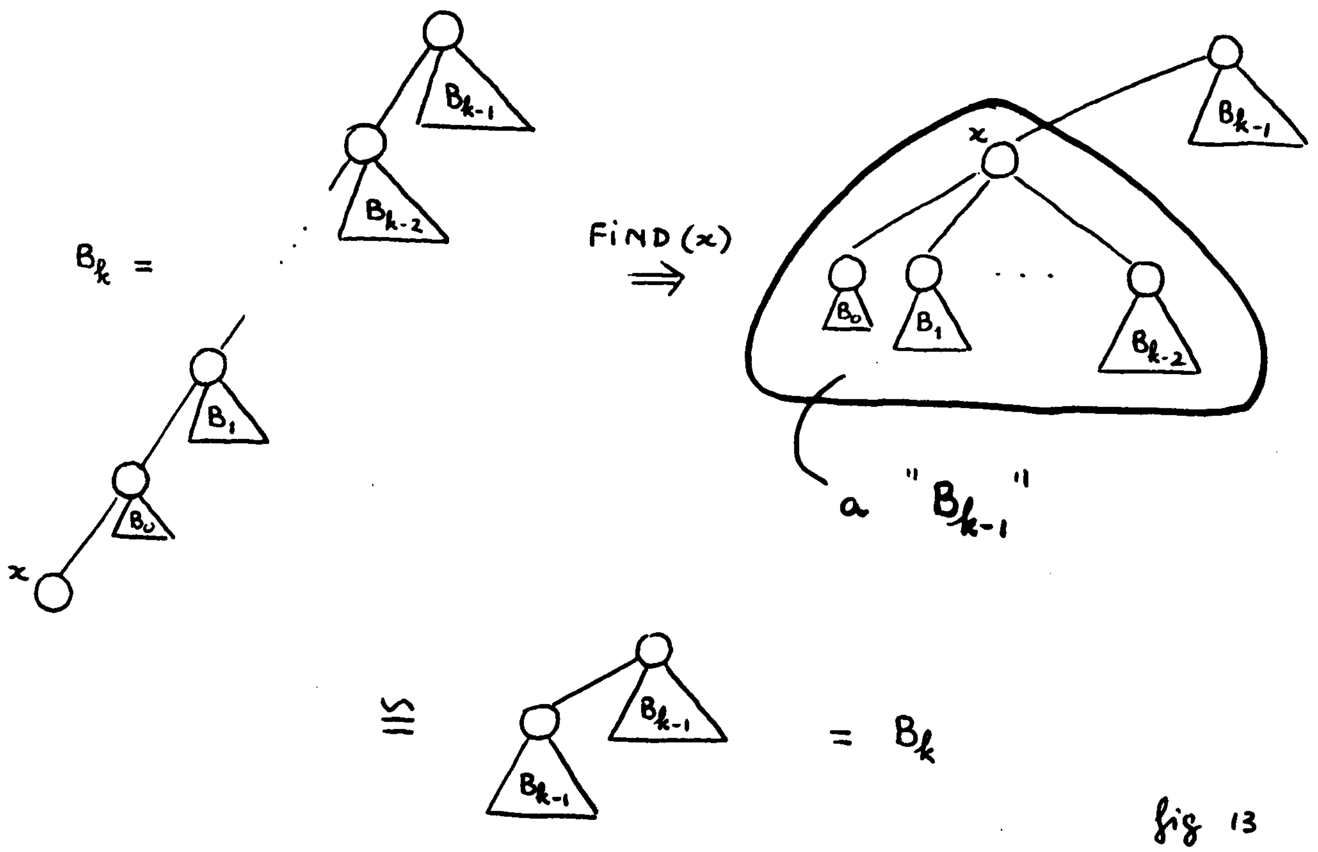


fig 13

balance make it hard to bound the running time of program with strategy D. It is certainly not true that strategy D works towards a collapsing of trees. It will be an interesting exercise to prove that any tree of the type shown in fig 14. a can be transformed into a "tree" as in fig 14. b, just by applying the right sequence of FINDs.

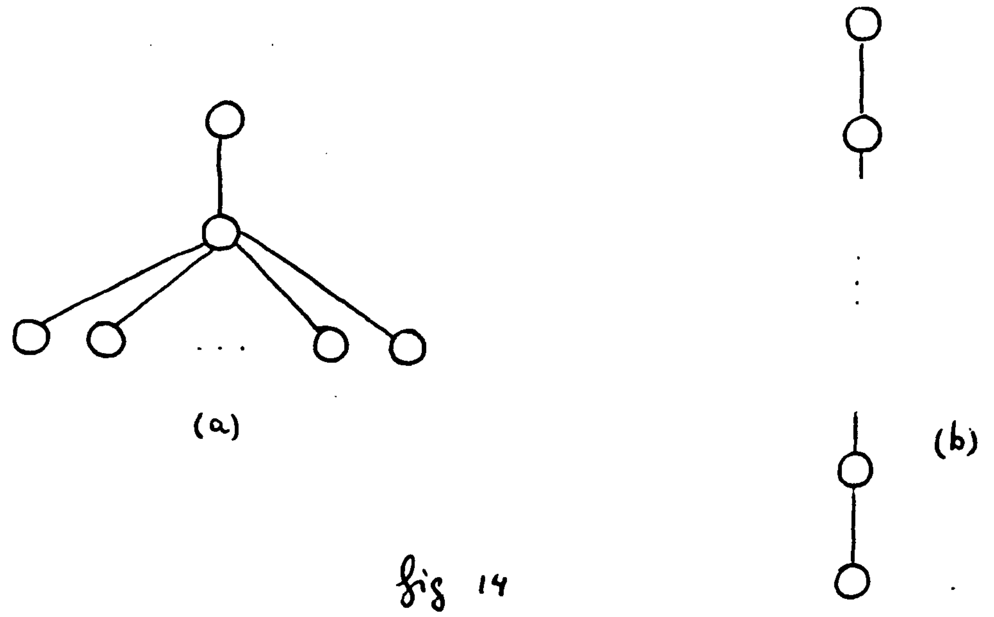


fig 14

We can save strategy D by modifying the procedure when the end of a FIND-path is reached : as noted in section 2, we should not disconnect and reattach the last node before the root. The modified strategy D will perform a FIND as follows

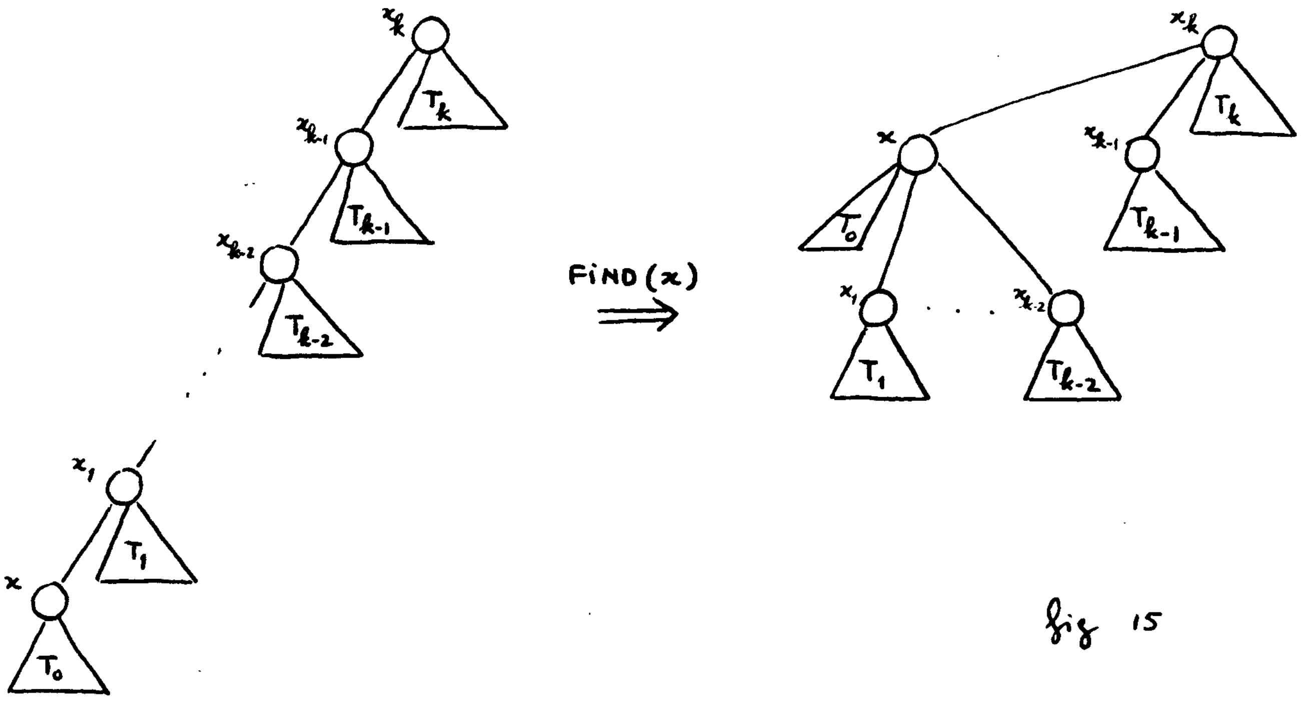


fig 15

The important fact to observe is that the <u>modified strategy D does not increase the maximum path-length in a tree</u>. As any reasonable balancing scheme for UNIONs keeps the maximum path-length of any tree within $O(\log n)$, we immediately have

<u>Theorem</u> 3.10 Any program of $n-1$ UNIONs and m FINDs using balancing and the modified strategy D runs within $O(n + m \log n)$ time.

As m gets larger trees will eventually become fully collapsed. The modified strategy D will quickly proceed towards this state, <u>regardless the use of balancing</u>! In order to see this, we consider the execution of m FINDs on a forest of trees on which for the moment (and perhaps forever after) no

UNIONs are performed. We make no assumptions about the degree of internal balance in the trees of the initial forest. The argument will prove the use of yet another technique for bounding changes in a forest, by bounding the effect on a suitably chosen entropy-function.

Theorem 3.11 Consider a forest F of trees on a total of n points, with maximum path-length d. Any program of m FINDs on F runs within $O(m + n \log d)$ time.

Proof.

Measure the cost of a FIND on x by the depth of x. Let the t^{th} FIND be executed at "time" t. Consider the following, time dependent entropy-function H:

$$H(t) = \sum_{i=1}^{n} \log(d_i(t) + 1)$$

where $d_i(t)$ is the depth of set-element i in the forest at time t. When a FIND is performed, the depth of a number of points will decrease (unless it was a FIND of cost 1) and for all t

$$\Delta H(t) = H(t+1) - H(t) \leq 0$$

In fact, $\Delta H(t) < 0$ if and only if at time t a FIND of cost greater than 1 was performed. As $H(t)$ must remain ≥ 0 at all times, its value cannot be decremented in an unlimited manner. We shall use this argument to bound the total cost of "expensive" FINDs.

Consider the execution of a FIND of cost $k \geq 2$ (at some time t), see fig 15. Even considering the reduction in entropy caused by the upward move of the points $x, x_1, \cdots, x_k$ only, we obtain the following estimate for $\Delta H(t)$:

$$
\begin{aligned}
-\Delta H(t) &= H(t) - H(t+1) \geq \\
&\geq \sum_{j=0}^{k} \log(k-j+1) - \left(\log 1 + \log 2 + \sum_{j=1}^{k-2} \log 3\right) = \\
&= \log (k+1)! - 1 - \log 3^{k-2} \geq \\
&\geq \alpha \cdot k
\end{aligned}
$$

for some positive constant α. (It is irrelevant that the bound could be sharpened). Thus $k_t \leq -\frac{\Delta H(t)}{\alpha}$ for any FIND of cost $k = k_t \geq 2$, and we have a way of bounding the cost of FINDs in terms of the entropy-reduction. (The index t indicates the time at which a FIND was performed).

Let $T = \{t_1, t_2, \cdots\}$ be the collection of times at which a FIND of cost ≥ 2 is performed. We can bound the total cost or run-time for such "expensive" FINDs by

$$
\begin{aligned}
\sum_{t\in T} k_t &\leq \frac{1}{\alpha} \sum_{t\in T} \{-\Delta H(t)\} \leq \\
&\leq \frac{1}{\alpha} \sum_{t=0}^{m-1} \{-\Delta H(t)\} = \\
&= \frac{1}{\alpha} \{H(0) - H(m)\} \leq \\
&\leq \frac{1}{\alpha} H(0) \leq \frac{1}{\alpha} n \log(d+1)
\end{aligned}
$$

Adding the run-time for FINDs of cost 1 we obtain the bound of $O(m + n \log d)$ for the entire program.

■

The result of 3.10 shows that even if FINDs following the modified strategy D are performed on an initially completely unbalanced forest ($d \sim O(n)$), then the runtime of the program segment is bounded by $O(m + n \log n)$! In other words, "it takes at most n FINDs of cost $\geq \log n$ to collaps the forest".

4. Another one-pass technique and Rem's algorithm.

In section 2 we made a first attempt to find efficien one-pass collapsing strategies by modifying the straightforward rule for FINDs. In section 5 we shall continue this approach and actually propose the (currently) best performing strategy of all. In this section we shall explore how a suitable one-pass technique might be obtained by modifying the execution of UNIONs, meanwhile fixing a simple rule for FINDs. <u>We shal assume that</u> FINDs <u>are performed as a straight upward traversal of their</u> FIND-path <u>without any collapsing</u>. If any collapsing is to occur, it must be the result or side-effect of UNION or of MERGE instructions.

Perhaps the simplest implementation of UNION-FIND program makes use of fully collapsed trees in which, in addition to the f-link to the root, each leaf-node has a g-link which connects it to an immediate successor in the set (fig 16).

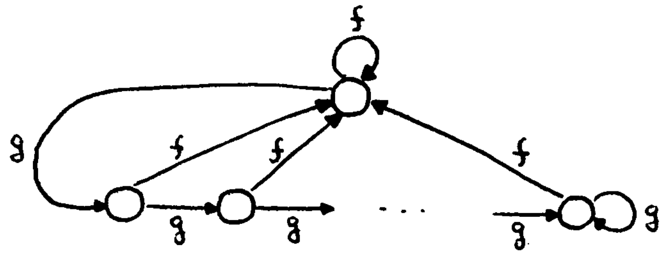


fig 16

With this representation each FIND(x) (or CHECK(x,y) for that matter) can be answered in constant time:

<u>output</u> (N (f (x)))

Each UNION (x,y) is executed according to the weighted union rule, merging the smaller set into the larger set. However, in addition to making the root of one set a son of the other, the resulting tree is being collapsed immediately as part of the UNION instruction, by traversing the smaller set and reconnecting its elements to the new root. The algorithm is wellknown (see Aho, Hopcroft and Ullman [1]), and will be obvious after contemplating fig 17.

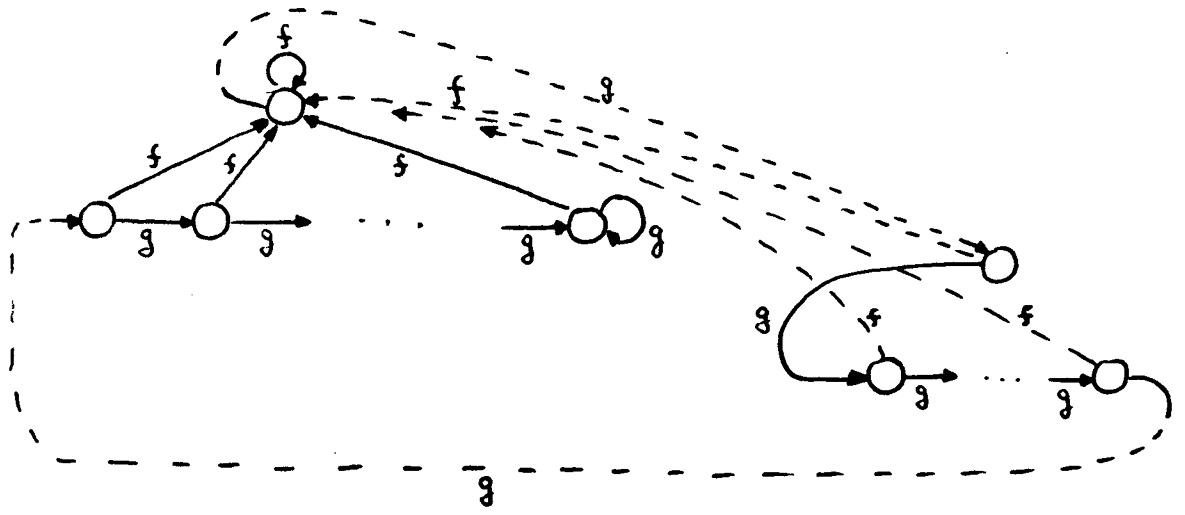


fig 17.

The effect of a UNION is that the smaller set, as a list, is attached in front of the list of elements of the larger set. The method is sometimes referred to as the "quick find / weight-ed" (QF-W) strategy.

The cost for resetting the links in a UNION is proportio-nal to the cost for traversing the small set through its g-links. It should be clear why the g-links are needed: only by traversing the set and reconnecting elements to the new father we can maintain the invariant that trees are always fully collapsed. Observe that when a set-element is "moving into" a new set, it becomes a member of a set at least twice as large as the small set it came from. It follows that the cost charged to a single set-element can be no more than log n. (In other words, if indivi-dual UNIONs are expensive there can't be too many of them)

Theorem 4.1 Any program of n-1 UNIONs and m FINDs using the QF-W strategy requires at most $O(m + n\log n)$ time, an this bound is achievable.

Thus, we have to pay for keeping trees fully collapsed bu FINDs are "free". The QF-W strategy is likely to perform as good as any other method if the number of UNIONs is small compared to the number of FINDs. Even as a general techniqu it may perform well, because Knuth & Schönhage [6] recently showed that on the average (in a precise sense) $O(n)$ UNIONs and FINDs will require just $O(n)$ steps.

If we drop the g-links (thus returning to the traditional tree-representation) and just use balancing for UNIONs, the worst case complexity for n-1 UNIONs and m FINDs is $O(n + m\log n)$. The succint difference with the QF-W strategy is that this time UNIONs are "free" and FINDs take "long". As the number of FINDs is usually dominating, we shall not consider this technique as it is merely the start of all our efforts to find a collapsing mechanism.

An original approach to improving trees during a set-manipulation program is due to M. Rem, as presented by Dijkstra [3]. We shall briefly outline the proper context for the method.

Implementing equivalence-classes of set-elements as tree Dijkstra wants to perform the following operations on a forest elegantly and expediently:

(i) test whether elements x and y currently belong to the same equivalence class.

(ii) merge the equivalence-classes containing x and y

For the sake of simplicity we use the CHECK and MERGE primitives for these operations. The original part of Rem's algorithm which makes it a one-pass collapsing strategy is the peculiar way MERGEs are executed. Recall that a MERGE (x, y) may be called for even if x and y are nodes below the root of their tree, unlike UNIONs. In fact, x and y need not even be in different trees for the strategy to work.

Let set-elements be numbered from 1 to n, the number of a node being called its <u>D-rank</u>. The invariant maintained throughout any CHECK-MERGE program is that <u>in any tree of the forest the</u> D-rank <u>of nodes is strictly decreasing along any path towards the root</u>. Thus, the elements with smaller D-rank can be expected to be "close" to the root. Fig 18 shows that there is no actual

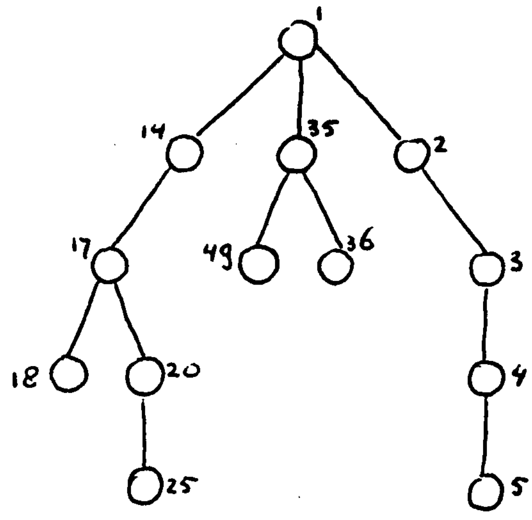
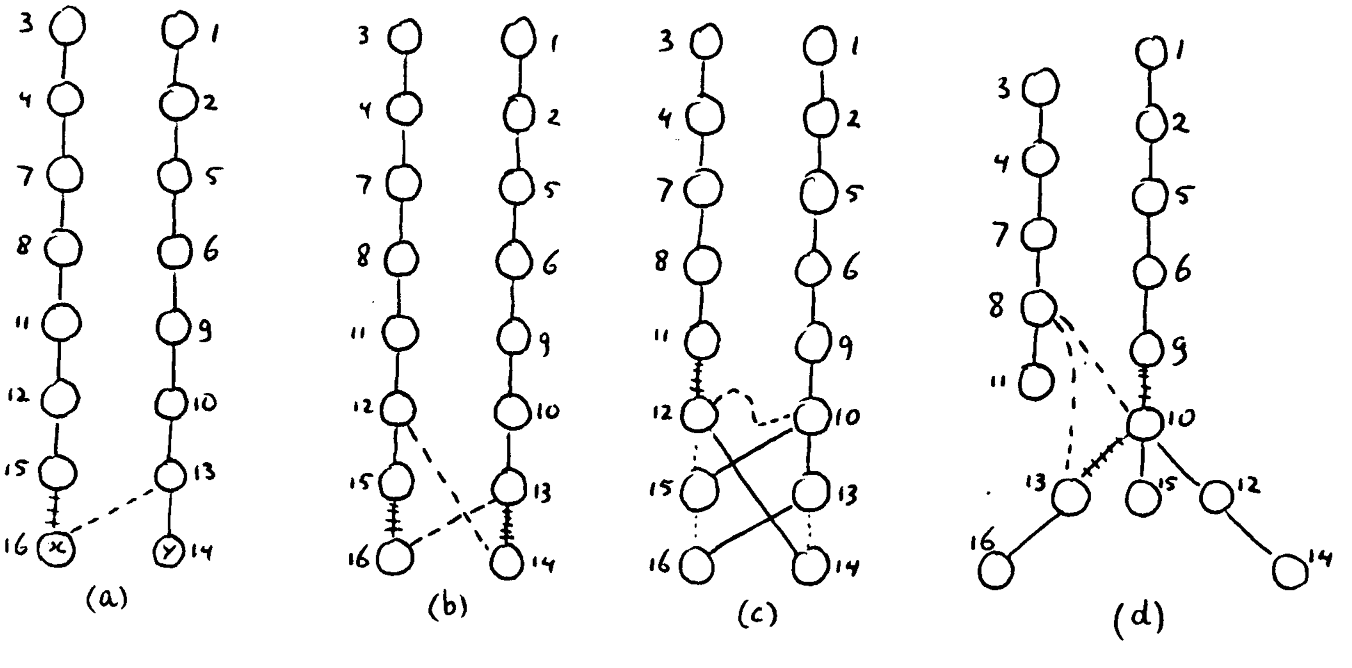


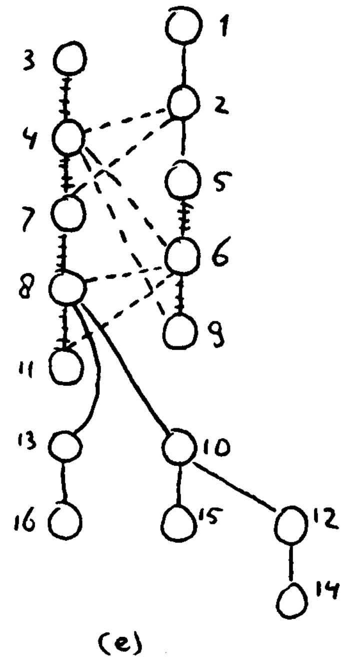
fig 18

guarantee for this implied by the invariant, but certainly the small-ranked elements will provide the confidence that they will eventually be close to the root. A MERGE is implemented by traversing the FIND-paths of x and y concurrently, each time reconnecting the node of one path to the father of the node reached along the other path if the D-rank of this father happens to be smaller than the D-rank of its original father. If the FIND paths intersect (just when x and y belong to the same tree), then the process automatically stops at the (unique) intersection point. More precisely, the procedure statement for MERGE (x, y) accordin to Rem's strategy is :

```
p := x ;
q := y ;
while f(p) ≠ f(q) do
    if f(p) < f(q) then  t := f(q) ; f(q) := f(p) ; q := t fi
    if f(p) > f(q) then  t := f(p) ; f(p) := f(q) ; p := t fi
od ;
```

Thus as a MERGE is being performed, there can be a considerable cross-connecting activity as the FIND-paths are traversed. See fig 19 for an example, in various stages of





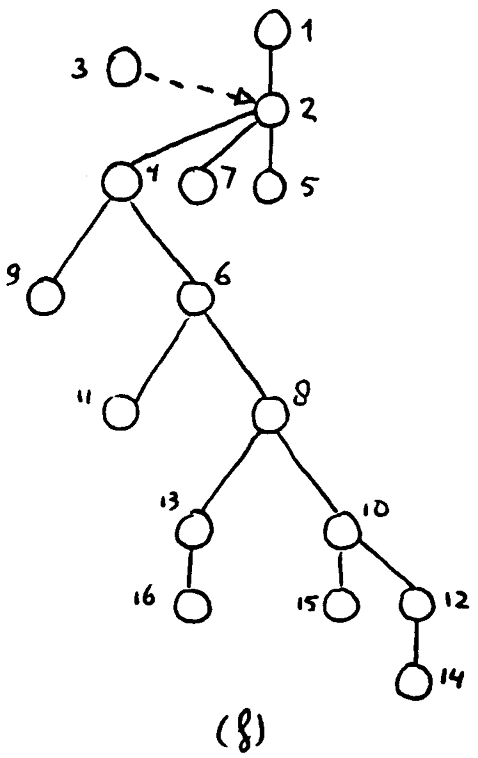


fig 19

the process. The example was intensionally chosen to show that the paths can be completely cut to pieces, and re-assembled to a merged version where little of the original paths is recognised. It also shows that the elements on which the procedure started (16, 14) can remain at the same distance from the root, or move very little in a step, although usually more is gained. See fig 20 for a different situation.

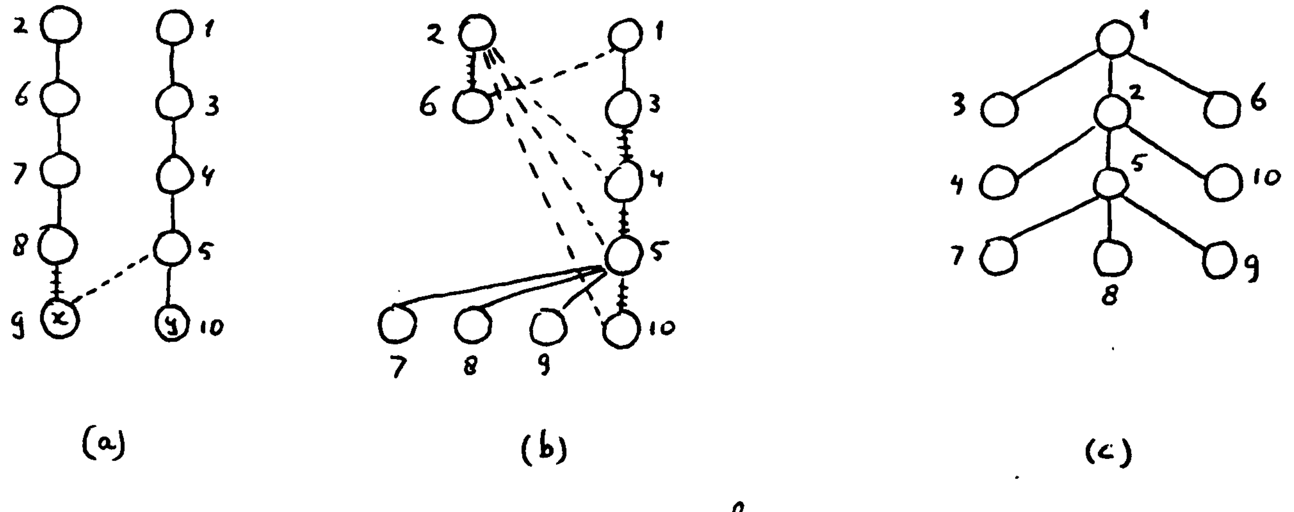


fig 20

The cross-connecting activity is an intrinsic feature of the MERGE-strategy. Each point along the FIND-paths of x and y is eventually reconnected to a smaller father, unless it was connected to the smallest root (if x and y are in disjoint trees) or to the lowest common ancestor of x and y (if they belong to the same tree and their FIND-paths consequently intersect). From this observation one may easily derive that eventually the forest must converge to a fully collapsed form, provided a sufficient number of MERGE instructions is executed.

Rem's algorithm does not attempt any collapsing during a FIND(x) or CHECK(x,y). It is understandable that no path-merging should take place in a CHECK, as we do not know beforehand whether x and y are in the same or in different equivalence classes. (In the latter case no merging of the classes may be intended). This makes Rem's algorithm worse than just about any strategy we have seen before, if we take the worst case performance on a program of n-1 MERGEs and m CHECK as a criterion.

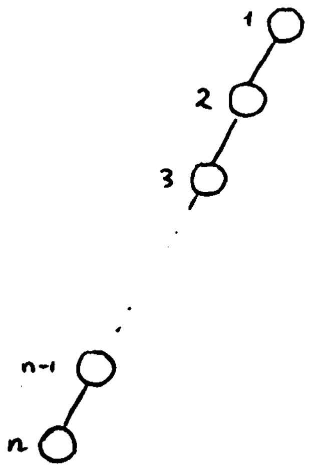
Theorem 4.2: There are programs of n-1 MERGEs and m CHECKs which require m.n steps when Rem's strategy is used

Proof.

Perform the following program of MERGEs on an initial forest of isolated points 1, ..., n :

```
MERGE (n, n-1) ;
MERGE (n-1, n-2);
  :
MERGE ( 2, 1 );
```

As we are continuously "merging" root-nodes, Rem's algorithm makes us attach the larger node to the smaller each time, resulting in the following tree in the end



Clearly, any subsequent CHECK instruction will require $\sim n$ steps on the average. For instance, m subsequent CHECK(i,j) instructions with $i, j \geq \frac{1}{2}n$ will force us to make mn steps or more.

■

In contrast to 4.2 recall that most "reasonable" one-pass collapsing strategies work within $O(m \log n)$ time. The proof of 4.2 depends on the high cost for FINDs and CHECKs, and it may not be a fair criterion to dismiss Rem's algorithm as "very inefficient". All indications are that the algorithm was intended for programs with a large number of MERGEs and perhaps no CHECKs at all. Even so, our next result will prove that Rem's algorithm doesn't perform better (in worst case) than the earlier strategies even when only MERGE instructions are considered.

<u>Theorem</u> 4.3 There are programs of $O(n)$ MERGE instructions which require $n \log n$ steps when Rem's algorithm is used

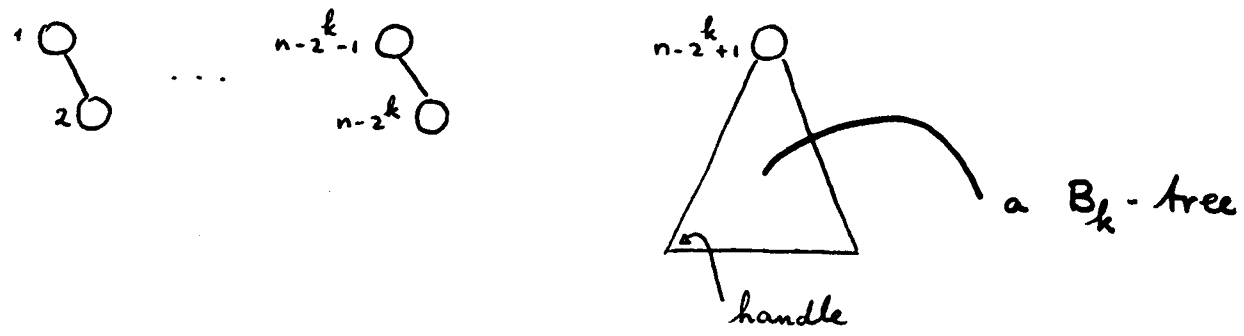
<u>Proof</u>.

The argument will be somewhat similar to M. Fischer's proof [ ] that full collapsing without balancing may force UNION-FIND programs to run for $n \log n$ steps. The way of applying the ideas to Rem's algorithm is as follows.

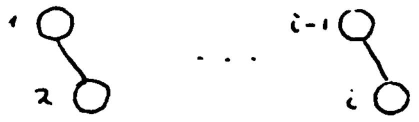
Pick a k (to be determined more precisely later), and spli the elements $1, \dots, n$ in two sets

$$\underbrace{1, \dots, n-2^k}_{A}, \underbrace{n-2^k+1, \dots, n}_{B}$$

Using $O(n)$ MERGE-instructions one can combine the elements of B into a B_k-structure, and the elements of A into pairs



(If $n-2^k$ is odd, then we omit 1 and pair the elements of A from 2 onwards). At all times during the remainder of the program we will have some pairs left



($i \le n-2^k$, i even), whereas the remaining elements of higher D-rank are all combined into one big tree which must contain a B_k as a sub-structure (see fig 20). Let x_i be the handle of the B_k at this time.

Consider the execution of a MERGE (i, x_i) according to Rem's algorithm :

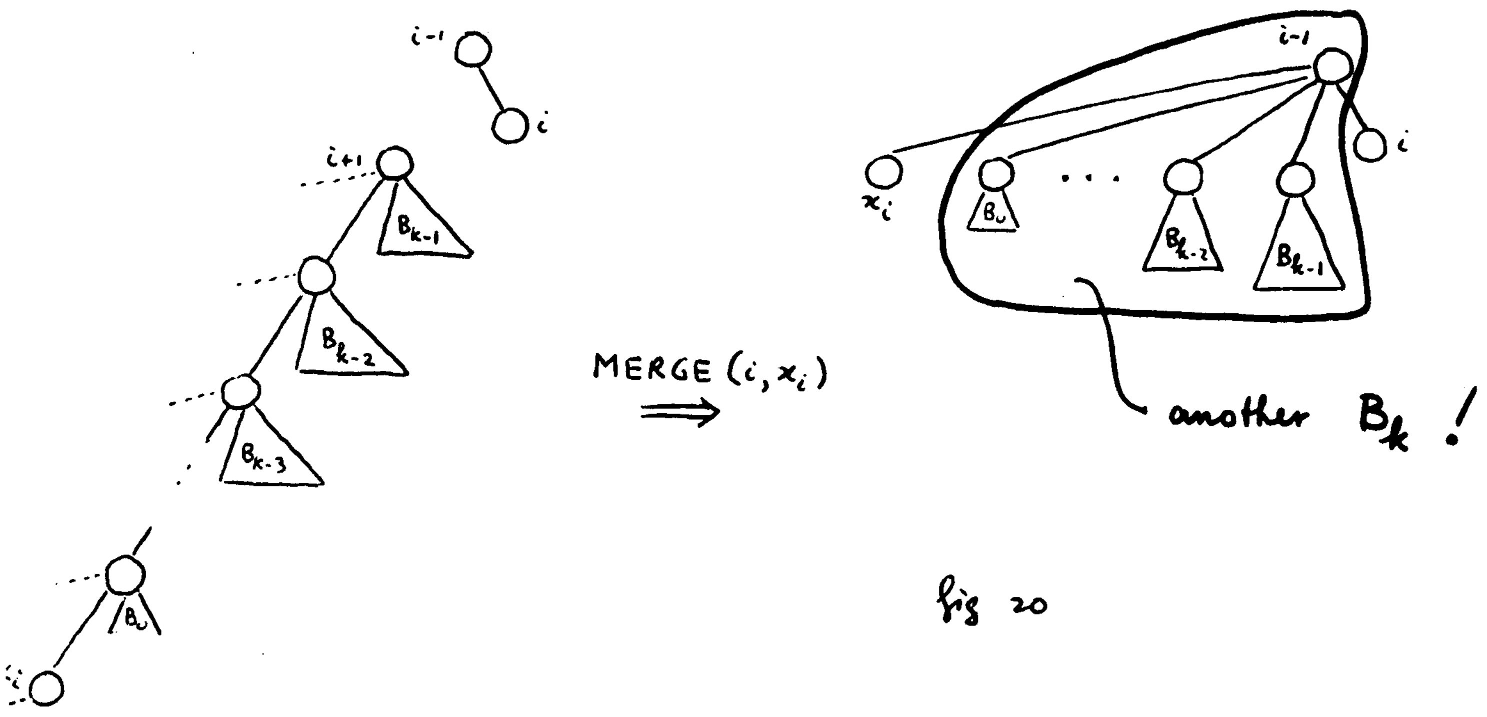


fig 20

Because all nodes (thus all fathers) in the big tree have D-rank $\geq i+1$, Rem's algorithm makes us attach all points on the FIND-path of x_i to the father of i. Thus Rem's algorithm has the effect of the full collapsing rule in this particular instance. We see that the invariant has been preserved, and can pick the next pair and MERGE its lower element with the handle of the new B_k-structure. And so on.

It follows that we can perform $\frac{n-2^k}{2}$ MERGEs this way, at a cost of k each. Chosing $k \sim \frac{1}{2}\log n$, the program requires

$$\frac{n-2^k}{2} \cdot k = \frac{n-\sqrt{n}}{2} \cdot \tfrac{1}{2}\log n \geq \alpha n \log n$$

for some constant α.

■

We conclude that the "compelling beauty" of Rem's algorithm is somewhat deceiving, and its efficiency remains under the expectations. The proofs of 4.2 and 4.3 indicate that the major drawback of Rem's algorithm is its inadequacy for detecting and circumventing unbalanced situations. Respec-ting D-rank never guarantees balancing. In the next sectio we shall present a much simpler, one-pass collapsing strateg which has a nearly linear running time. We conclude that Rem's algorithm has unexpected disadvantages, which prohibit its use for practical programs.

5. Path-halving

In this section we present some final strategies for approximating the effect of full collapsing by a one-pass technique Consider the FIND-path of a point x, fig 21.a

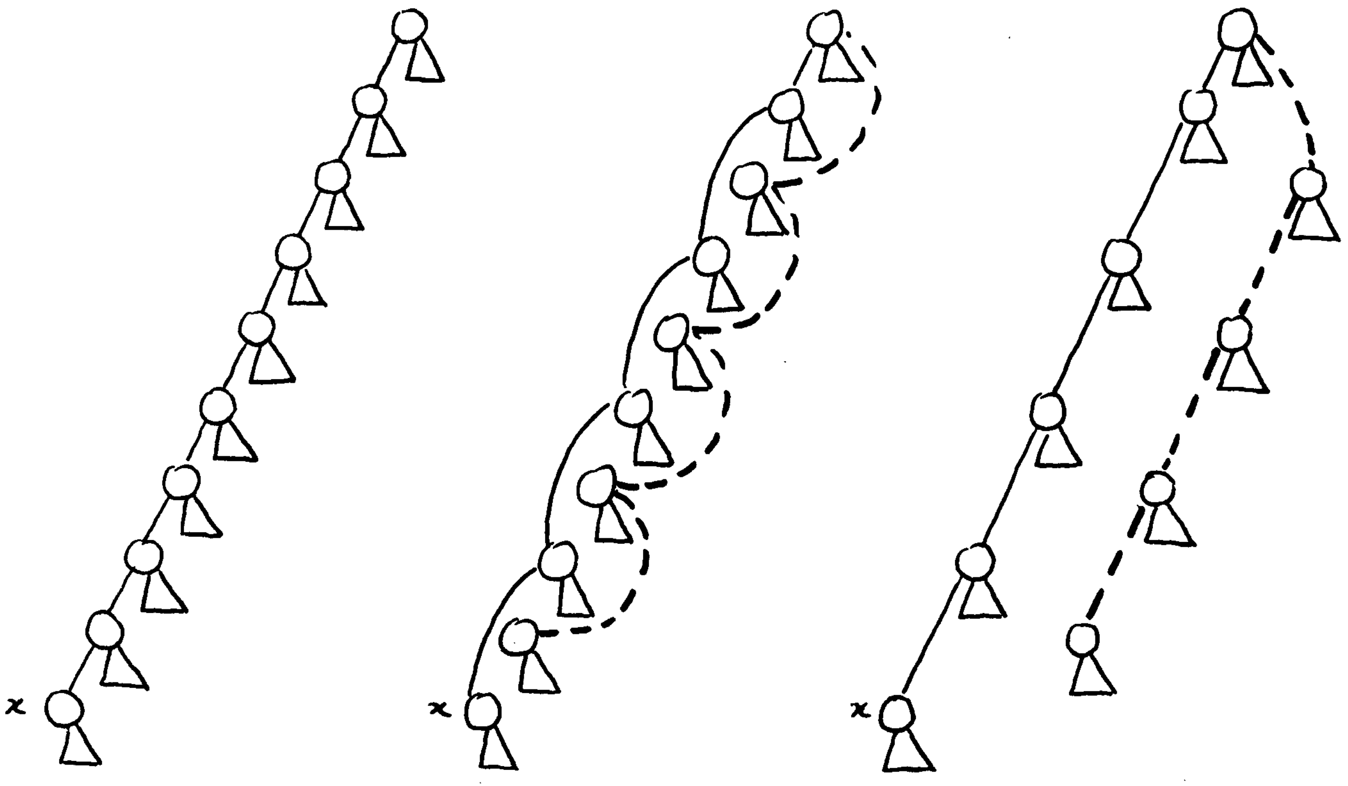


fig 21

As we are walking up the tree we can systematically reconnect each point to the node one further ahead, thus reducing the distance of each point along the path to the root by a factor ½. (See fig 21. b)

Strategy E. As the FIND-path is traversed, reconnect each node to its grandfather.

We shall see later that strategy E for FINDs and balancing for UNIONs indeed give a nearly linear running time for UNION-FIND programs, but let us momentarily deal with strategies without the added overhead of taking balancing into account.

The effect of strategy E is not just a reduction of the path-length for nodes along the FIND-path, but it also separates the nodes into two disjoint paths at the same time (fig 21.c). Consequently, if another FIND were executed on x only about half the points would benefit from the iterated compression of its path. This slight disadvantage is eliminated in the following variant of strategy E, in which we are careful not to separate the points into two totally independent paths. See fig 22.

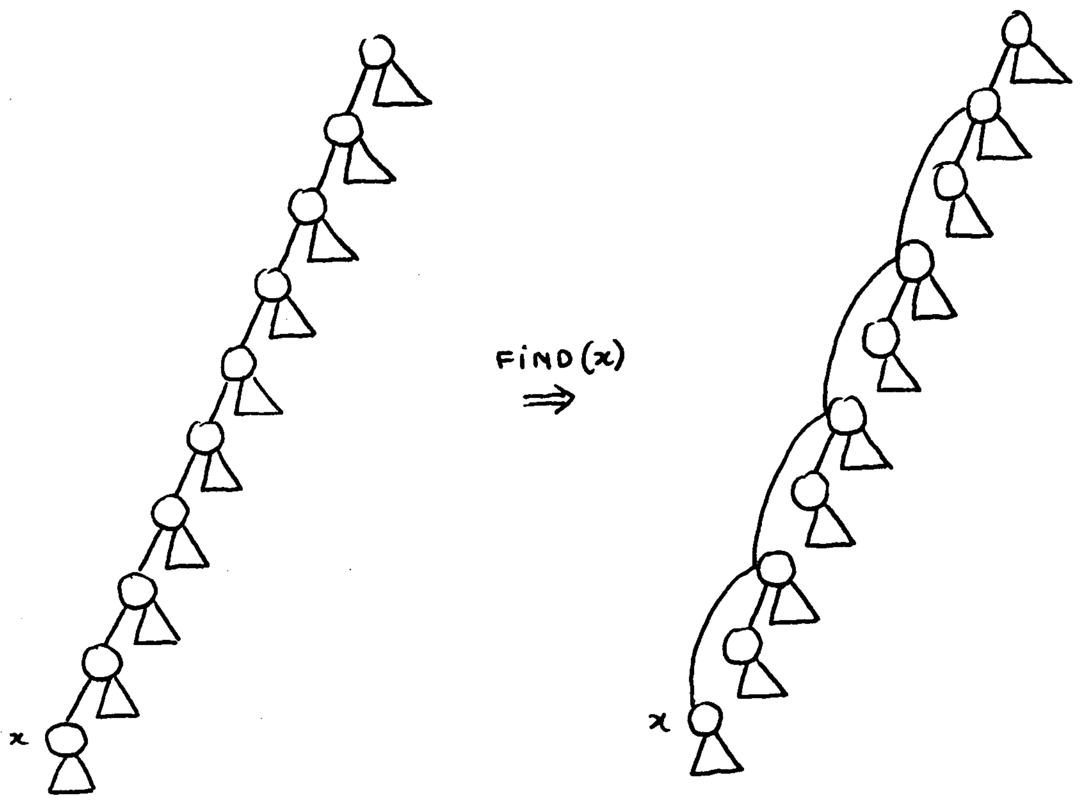


fig 22

<u>Strategy</u> F. As the FIND-path is traversed, reconnect each second node (beginning at x) to its grandfather.

This technique is called <u>path-halving</u>, and will be considered from now on. We shall prove first that in case no particular balancing rule for UNIONs is applied the run-time for $O(n)$ FINDs implemented following the path-halving technique is $O(n \log n)$, thus very much competitive with all previous strategies of this sort.

Theorem 5.1 The run-time for $n-1$ UNIONs and m FINDs using path-halving is bounded by $O(n + m \log n)$

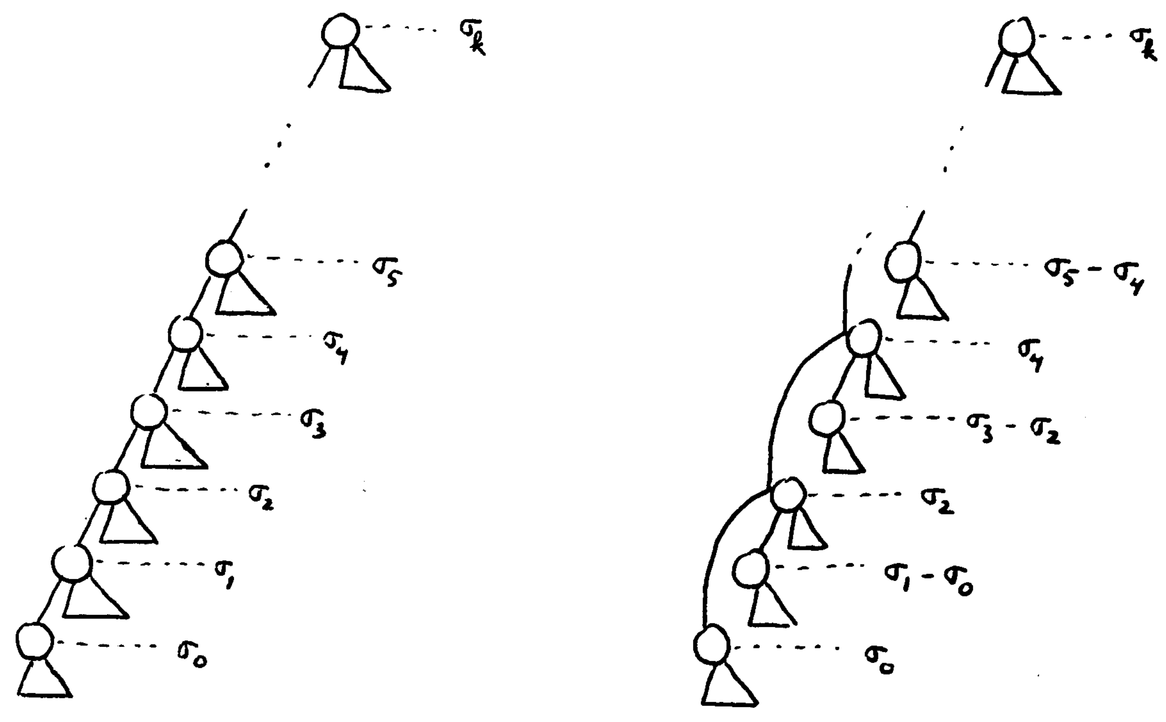
Proof.

The proof makes use of a somewhat more complicated entropy argument. Let the weight $\sigma_i(t)$ of element i be the number of nodes in its subtree at time t. Clearly $\sigma_i(t) \geq 1$, as the subtree always contains i itself. Consider the following entropy function H:

$$H(t) = \sum_{i=1}^{n} \log \sigma_i(t)$$

Let $\Delta H(t) = H(t+1) - H(t)$. In UNIONs the entropy increases ($\Delta H > 0$) because the root of one tree gets a larger weight. We shall soon see that the entropy decreases when a non-trivial FIND is performed ($\Delta H < 0$). Note that during a UNION only the contribution of the root may change, whereas in a FIND only the contribution of interior nodes to H may change. In other words, the entropy contribution of a root cannot change in a FIND. It follows that UNIONs cannot cause a total increment of more than $\log(\text{max. weight}) = \log n$ per point. We conclude that FINDs cannot decrement H for more than a total of $n \log n$.

It remains to correlate the cost of FINDs and the corresponding reduction of the entropy. Consider a FIND of cost k,



with $\sigma_0, \cdots, \sigma_k$ denoting the weights of the points along the FIND-path. Note that

$$1 \leq \sigma_0 < \sigma_1 < \cdots < \sigma_k \leq n$$

It follows that

$$\Delta H(*) = \sum_{j=0}^{\lfloor \frac{k}{2} \rfloor - 1} \left\{ \log(\sigma_{2j+1} - \sigma_{2j}) - \log \sigma_{2j+1} \right\} =$$

$$= \sum_{j=0}^{\lfloor \frac{k}{2} \rfloor - 1} \log\left(1 - \frac{\sigma_{2j}}{\sigma_{2j+1}}\right)$$

$$\leq \sum_{j=0}^{\lfloor \frac{k}{2} \rfloor - 1} - \frac{\sigma_{2j}}{\sigma_{2j+1}} = - \sum_{j=0}^{\lfloor \frac{k}{2} \rfloor - 1} \frac{\sigma_{2j}}{\sigma_{2j+1}}$$

or: $|\Delta H(*)| = - \Delta H(*) \geq \sum_{j=0}^{\lfloor \frac{k}{2} \rfloor - 1} \frac{\sigma_{2j}}{\sigma_{2j+1}}$.

In order to estimate this further, let us assume that $k > \log n$. Choose some constant $\alpha > 1$. Call σ_{2j} and σ_{2j+1} <u>close</u> if $\sigma_{2j}/\sigma_{2j+1} \geq \frac{1}{\alpha}$, <u>far apart</u> if $\sigma_{2j}/\sigma_{2j+1} < \frac{1}{\alpha}$. Let $\lfloor \frac{k}{2} \rfloor - 1 - l$ pairs be close, and l pairs be far apart.

If l pairs are far apart, then necessarily

$$n \geq \sigma_k > \alpha^l \sigma_0 \geq \alpha^l$$

thus $l < {}^{\log n}/_{\log \alpha}$. It follows that

$$|H(t)| = \sum_{j=0}^{\lfloor \frac{k}{2} \rfloor - 1} {}^{\sigma_{2j}}/_{\sigma_{2j+1}} \geq$$

$$\geq (\lfloor \tfrac{k}{2} \rfloor - 1 - l) \cdot \tfrac{1}{\alpha} \geq$$

$$\geq (\lfloor \tfrac{k}{2} \rfloor - 1 - {}^{\log n}/_{\log \alpha}) \cdot \tfrac{1}{\alpha}$$

$$\geq \beta \cdot k$$

for some constant $\beta > 0$ (assuming that $k > \log n$).

We can now estimate the cost for m FINDs as follows. FINDs of cost $\leq \log n$ are within the limit of the result. The total run-time needed to deal with FINDs of cost $> \log n$ is bounded by

$$\sum k_t \leq \frac{1}{\beta} \sum_t |H(t)| \leq \frac{1}{\beta} n \log n$$

where the summation extends over the appropriate times.

■

The proof of 5.1 indicates that in path-halving there can be only a "few" FINDs of cost $> \log n$, as the total runtime for such FINDs is bounded by $n \log n$. As path-halving cannot be better than full collapsing, we know that the $n \log n$ bound for $O(n)$ UNIONs and FINDs cannot be improved. If m gets larger, the bound of 5.1 is no longer accurate.

As for full collapsing, the worst case run-time of a program with path-halving improves considerably when balancing is used for UNIONs. We shall not treat this in great detail as the analysis can be carried out along very much the same lines as Hopcroft & Ullman's proof for full collapsing ([5], see also Aho, Hopcroft, and Ullman [1]). The analysis goes through in the following, general sense.

Consider any program of n-1 UNIONs and m FINDs using balancing for UNIONs and some collapsing strategy for FINDs to be qualified later. Let the rank of a point be its height in the forest as it would be if no collapsing had taken place. An important invariant which should be maintained by the FIND-strategy is the monotonicity of ranks along any path towards a root. The crucial point is that there is a correlation between balance and rank (if the weighted union rule is used), and as paths are compressed and nodes get reconnected to fathers of higher rank the number of such fathers rapidly decreases with the path-distance to the root. The level-crossing and bookkeeping technique of Hopcroft & Ullman [5] remains valid for any FIND-strategy which satisfies the following condition

> monotonicity : as a FIND(x) is performed, each node along the FIND-path gets reconnect to a father of higher rank than the father it had.

Clearly strategy E is monotone, and strictly speaking path-halving isn't. But the same analysis can be easily modified.

Theorem 5.2 The run-time of any program of n-1 UNIONs and m FINDs using balancing and path-halving (or strategy E) is bounded by $O(n + m\log^* n)$.

Thus we found some easy one-pass collapsing strategies for UNION-FIND programs which preserve the near linear running time adequately. It is the currently best solution to Dijkstra's problem, which should pass the test of programming elegance with ease.

Attempts to improve the analysis, to obtain a bound of $\Theta(m\,\alpha(m,n))$ for instance, have not been successful. At this time we must conjecture that 5.2. is the best possible bound for the path-halving strategy, and we conjecture that $n \log^* n$ is in fact the best bound for any one-pass collapsing technique.

6. References

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

[2] Brown, M.R., The analysis of a practical and nearly optimal priority queue, Dept of Computer Sci, Stanford University, STAN-CS-77-600 (March 1977), also: Proc 9th Ann. ACM Symp. Th. of Computing, Boulder (1977) 42-48

[3] Dijkstra, E.W., A discipline of programming, Prentice-Hall Inc, Englewood Cliffs, NJ (1976)

[4] Fischer, M.J., Efficiency of equivalence algorithms, in: R.E. Miller & J.W. Thatcher (eds), Complexity of computer computations, Plenum Press, New York (1972), 153-167

[5] Hopcroft, J.E., and J.D. Ullman, Set merging algorithms, Siam J. Computing 2 (1973) 294-303

[6] Knuth, D.E., and A. Schönhage, The expected linearity of a simple equivalence algorithm, Department of Computer Sci, Stanford University, STAN-CS-77-599 (March 1977)

[7] Paterson, M., unpublished (see [8])

[8] Tarjan, R.E., Efficiency of a good but not linear set union algorithm, JACM 22 (1975) 215-225

[9] Tarjan, R.E., Reference machines require non-linear time to maintain disjoint sets, Dept of Computer Sci, Stanford University, STAN-CS-77-603 (March 1977)

[10] van Leeuwen, J., The complexity of data organisation, in J.W. de Bakker (ed), Foundations of computer science (II vol 1, MC Tract 81, Mathematisch Centrum, Amsterdam (1976)

[11] Vuillemin, J., A data structure for manipulating priority queues, CACM (to appear).