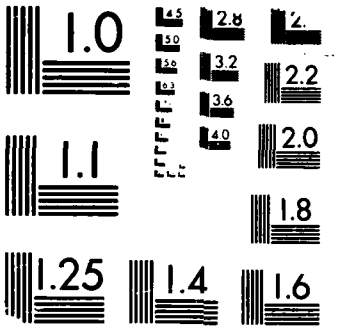
AD-A194 030 RELAXED HEAPS: AN ALTERNATIVE TO FIBONACCI HEAPS(U) 1/1
PRINCETON UNIV NJ DEPT OF COMPUTER SCIENCE
J R DRISCOLL ET AL. JUL 87 CS-TR-109-87
UNCLASSIFIED N00014-87-K-0467 F/G 12/4 NL



MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS 1963-A

(4)

Princeton University

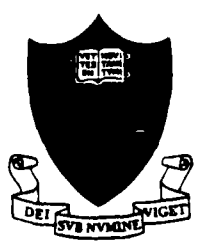
RELAXED HEAPS: AN ALTERNATIVE TO FIBONACCI HEAPS

James R. Driscoll
Harold N. Gabow
Ruth Shrairman
Robert E. Tarjan

CS-TR-109-87

July 1987

Department of Computer Science



(4)

RELAXED HEAPS: AN ALTERNATIVE TO FIBONACCI HEAPS

James R. Driscoll
Harold N. Gabow
Ruth Shrairman
Robert E. Tarjan

CS-TR-109-87

July 1987

Relaxed Heaps: An Alternative to Fibonacci Heaps

by James R. Driscoll[1], Harold N. Gabow[1*], Ruth Shrairman[1] and Robert E. Tarjan[2,3**]

[1] Department of Computer Science
University of Colorado
Boulder, CO
80309

[2] Computer Science Department
Princeton University
Princeton, NJ
08544

[3] AT&T Bell Laboratories
Murray Hill, NJ
07974

July, 1987

Abstract.

The relaxed heap is a priority queue data structure that achieves the same amortized time bounds as the Fibonacci heap— a sequence of m *decrease_key* and n *delete_min* operations takes time $O(m + n \log n)$. A variant of relaxed heaps achieves similar bounds in the worst case— $O(1)$ time for *decrease_key* and $O(\log n)$ for *delete_min*. A relaxed heap is a type of binomial queue that allows heap order to be violated.

---

* Research supported in part by NSF Grant No. MCS-8302648 and AT&T Bell Laboratories.

** Research supported in part by NSF Grant No. DCR-8605962 and ONR Contract No. N00014-87-K-0467.

1. Introduction.

The Fibonacci heap data structure of Fredman and Tarjan allows an optimal implementation of Dijkstra's shortest path algorithm [FT]. It is central to the best-known algorithm for minimum spanning trees [GGST] and many other algorithms. These applications are based on the fact that, with a Fibonacci heap, a sequence of m *decrease_key* and n *delete_min* operations takes time $O(m + n \log n)$. Equivalently, Fibonacci heaps achieve an amortized time of $O(1)$ for *decrease_key* and $O(\log n)$ for *delete_min*. This paper presents a new data structure called the relaxed heap. One implementation of relaxed heaps achieves the same amortized bounds as Fibonacci heaps, but maintains greater structure. It is hoped that this structure will make relaxed heaps faster in practice. A second implementation of relaxed heaps gives a theoretical improvement over Fibonacci heaps: it achieves the above time bounds for *decrease_key* and *delete_min* in the worst case, rather than in the amortized case.

Relaxed heaps are based on a more structured family of trees than Fibonacci heaps, namely the binomial trees. The height of a binomial tree of n nodes is a factor $\log_2 \phi = .69+$ times the height of a Fibonacci tree. This improves the constant in asymptotic estimates. Whether these savings are actually realized in practice is a matter for experimental verification, which we have not done. (Brown [B] shows that the closely related binomial queues are efficient in practice). Relaxed heaps give a large family of alternatives to Fibonacci heaps. These alternatives provide both theoretical insight and possibly the flexibility needed for efficient practical implementation.

This section closes by reviewing terminology and motivating relaxed heaps. Section 2 presents the rank relaxed heap, which achieves the same amortized time bounds as Fibonacci heaps. Section 3 presents the run relaxed heap, which makes the time bounds worst-case.

In this paper $\log n$ denotes logarithm to the base two. A *priority queue* is a data structure for storing a set of *items* x, each having a numerical *key* denoted $k(x)$. The main operations are

make_heap— initializes a heap to store the empty set;

insert(x)— make x a new item in the heap;

decrease_key(x, v)— decreases key $k(x)$ to a smaller value v;

delete_min(x)— deletes an item of minimum key from the queue and returns it as x.

Other operations will be introduced as needed.

Binomial queues were introduced by Vuillemin [V]. The *binomial trees* B_r are defined recursively as follows: B_0 is one node; B_{r+1} consists of two B_r trees, the root of one being a child of the root of the other. See Figure 1(a). In all figures, a triangle labeled r represents the binomial tree

B_r. Figure 1(b) shows an equivalent description of B_{r+1}: For any k, $0 \leq k \leq r$, B_{r+1} consists of a B_k tree with additional children of the root that are themselves roots of $B_k, B_{k+1}, \ldots, B_r$ trees. For any node x in a binomial tree, $rank(x)$ is the index r of the maximal subtree B_r rooted at x. A binomial tree is an ordered tree, with the children of a node ordered by increasing rank. The *last child* of a node is the child of highest rank. Clearly B_r has 2^r nodes.

In a *heap-ordered tree*, each node stores one item and any node c with parent p has $k(p) \leq k(c)$. Node c is a *good child* if this inequality holds; otherwise it is *bad*. A *binomial queue* for 2^r items is a heap-ordered tree B_r. A binomial queue for n items, n arbitrary, consists of at most $\lfloor \log n \rfloor + 1$ heap-ordered binomial trees, a tree corresponding to each one bit in the binary expansion of n. The *link* operation for binomial queues takes two root nodes of equal rank r and creates a heap-ordered tree B_{r+1} by making the node with larger key a child of the smaller.

It seems difficult to process *decrease_key* in $O(1)$ time and maintain heap order. Relaxed heaps avoid this difficulty by violating heap order (whence the name), in a limited sense. The rank relaxed heaps of Section 2 allow just one bad child per rank. The run relaxed heaps of Section 3 are more permissive and allow runs of bad children.

2. Rank Relaxed Heaps.

A *relaxed tree* is a tree where each node stores one item, nodes can be good or bad, but some nodes are distinguished as *active* and any bad node is active. The terms relaxed binomial tree and relaxed binomial queue are interpreted according to this definition. A *rank relaxed heap* is a relaxed binomial queue that satisfies two conditions:

(a) For any r there is at most one active node of rank r.

(b) Any active node is a last child.

Condition (a) implies there are at most $\lfloor \log n \rfloor$ active nodes. Condition (b) is not crucial, but rather it determines various programming details; we return to this point below. In the rest of this section relaxed heap means rank relaxed heap.

The *decrease_key* algorithm works by rearranging nodes to keep the heap relaxed. It does this with three transformations. More precisely *decrease_key*(x, v) resets $k(x)$, after which it may stop or execute a transformation; a transformation does $O(1)$ work, after which it may stop or execute another transformation. This gives rise to a sequence of tranformations. To achieve the time bound, let α denote the number of active nodes at any point in the algorithm. If resetting $k(x)$

in *decrease_key* makes x bad, x is designated active. This increases α by one. Each transformation either

(i) decreases α, or

(ii) does not change α and does not execute another transformation.

Observe that (i) - (ii) imply any sequence of m *decrease_keys* uses time $O(m)$: There are at most m type (ii) transformations (each is last in its sequence) and there are at most m type (i) transformations (only *decrease_key* increases α; assume there are no active nodes initially). This argument will not be affected by *delete_min* operations (see the proof of Theorem 2.1 below).

Now we describe the transformations informally. They are illustrated in Figures 2-5. In these figures an edge joining a child c to its parent p is labeled in one of four ways: An arrow from c to p indicates that c is good; a cross mark on the edge indicates c is active (and so can be good or bad); no mark indicates the status of c is unknown; an arrow pointing to the edge indicates c may be a new active node. The transformations use an operation "make node x the rank r child of node y". This means that the entire B_r tree rooted at x becomes the B_r subtree of y. For each transformation, assume *decrease_key* (or a transformation) has created an active node a of rank r with parent p and grandparent g (a is actually a bad node, although this fact is not used).

The main idea is embodied in the *pair transformation* (Figure 2). It applies when a is the last child of p, and further, the relaxed heap already contains an active node a' of rank r, with parent p' and grandparent g'. The transform removes the active nodes from their parents, so nodes a, a', p, p' all have rank r. Without loss of generality assume $k(p) \leq k(p')$. The transform makes p' the rank r child of p (hence p remains a rank $r+1$ node). Then it links a and a' to form a B_{r+1} tree with root c (so c is a or a'). It makes c the rank $r+1$ child of g'. If c is now a bad child it is active, and a transformation is done for it.

The last detail concerns linking a and a'. In general as in Figure 3 suppose nodes q and q' of rank $s+1$ have just been linked, making q the new root. If x, the rank s child of q, is active, it now violates condition (b) of the relaxed heap structure. Figure 3 fixes this up by doing a *cleaning operation*. It uses the fact that x', the rank s child of q', is good if x is active. This follows from the definition of a relaxed heap. (Care should be taken here, since the transformations are applied to heaps where the relaxed heap structure has been violated; however we will use the clean operation only when this deduction is valid). The operation repairs the damage by interchanging x and x'. In what follows, to *clean* node q means to apply the operation of Figure 3 if x is active; otherwise do nothing. To *combine* two nodes means to link them and then clean the new root. Thus, in Figure 2, the *pair* transformation combines a and a'.

The *pair* transformation achieves property (i), since two initially active nodes are replaced by at most one. Also note that the reason this transformation is used only when a and a' are last children is to achieve the $O(1)$ time bound: If a' has rank r but is not the last child, its siblings must not be transferred with p'. Assuming a reasonable data structure (e.g., a parent pointer for each node) this can consume more than constant time. Note that this objection does not apply if a' is a last child but a is not. This is the case in the good sibling transformation described below, which does a pair transform of this kind.

The remaining transformations are "sibling transformations". For these assume that a has next larger sibling s. The *active sibling transformation* applies when s is active (Figure 4). The definition of a relaxed heap implies that s is a last child, so p has rank $r+2$. The transform removes the two active children from their parent p. It *combines* p and a, making a a rank $r+1$ node. Then it *combines* a and s into a tree whose root c becomes the rank $r+2$ child of g. If c is now bad, a transformation is done for it. The active sibling transformation achieves property (i), since again two active nodes are replaced by at most one.

The *good sibling transformation* applies when s is good. Let c be the last child of s; c has rank r. There are two cases. If c is active the algorithm does a pair transform for a and c (Figure 5). As noted above, the transformation works correctly even though a is not a last child; the only delicate point is to make sure that if $k(p) = k(s)$ the algorithm makes s the child of p, not vice versa. This case achieves property (i), as above. The second case occurs if c a good child. Then the cleaning operation of Figure 3 is applicable. The transform *cleans* p, making a a bad last child of s. It then processes a as a new active last child. If there is an active node of rank r a *pair* transform achieves property (i); otherwise the sequence of transformations stops, achieving (ii).

Now we describe *delete_min*. Note that in most applications (e.g., all those in [FT, GGST]) it is unnecessary to reclaim the storage used by a deleted node. We give two implementations of *delete_min*, the simpler of which does not reclaim storage. Both implementations start by finding the smallest node x. Since x is either active or the root of a tree in the queue, it can be found in $O(\log n)$ time.

The nonreclaiming algorithm sets $k(x)$ to ∞ and changes $rank(x)$ from r to 0. Then it merges x and its former children into a new rank r node, by repeatedly *combining* the two nodes of smallest previous rank. The new rank r node replaces x in the tree. Note that x becomes a leaf and will not participate in any future transformation.

The reclaiming algorithm is similar. It deletes x and removes the root node of smallest rank y from its tree; this makes the previous children of y into roots with the smallest ranks in the queue.

Then it processes y (not x) and the former children of x as in the nonreclaiming algorithm.

Figure 6 gives a more detailed description of the algorithm in pseudo-Algol. The *delete_min* implementation uses the reclaiming algorithm. The following data structure is assumed. Each node x has a record containing $k(x)$, $rank(x)$, and pointers to its last child, its two neighboring siblings, and its parent. (A sibling pointer is needed in the sibling transformations, so pointers to both siblings are needed to allow nodes to be moved; a last child pointer is needed in the good sibling transformation). There is a dummy node treated as the root of the entire queue, so the roots of binomial trees are treated as siblings and are not special cases. In addition there is an array $A[0..\lfloor \log n \rfloor - 1]$; each $A(r)$ is a pointer to the bad last child of rank r, if it exists. The A array is used to check if a node is active, e.g., *promote* tests if s is active by checking the condition $A(r+1) = s$. It should be clear that this data structure supports the desired operations and can be maintained in time $O(1)$ per transformation.

Note that the data structure can be initialized in $O(n)$ time (assuming, as is often the case, that the number of items n is known in advance). One way is to construct a binomial queue on n items, with each key equal to ∞. The operation *insert*(x) is done by executing *decrease_key*$(x, k(x))$.

Theorem 2.1. Rank relaxed heaps correctly process a sequence of m *decrease_key* and $k \leq n$ *delete_min* operations in time $O(m + k \log n)$.

Proof. It is easy to check that the algorithm maintains this invariant: At the start of each call to *promote*, making the edge between a and its parent good gives a valid relaxed heap structure. This implies correctness. (Note that an active node can be good or bad: An entry in the A array starts out as a bad child; it may become good without being processed in a transformation, if the key of its parent is sufficiently decreased).

For the timing, observe that the *decrease_key* routine uses $O(1)$ time and the *delete_min* routine uses $O(\log n)$ time. The time for transformations is bounded as above (*delete_min* decreases α by one or zero, and so only improves bound). ∎

The transformations in this section were selected for economy of description. Probably different ones would be more efficient in practice. For instance, in the *combine* routine, a more productive way to *clean* the tree of the new root is to repeatedly make an active bad child of the root into the new root, until the root has no active child. This approach has the advantage of decreasing the number of active nodes. Alternatively, cleaning can be eliminated entirely by dropping the requirement that an active node in a relaxed heap be last. Instead, the active sibling transfomation

is extended to process a string of consecutive active siblings; the processing is similar to *delete_min*. Such transformations might work better if the processing of a *decrease_key* operation is delayed until the next *delete_min* (in the hope of getting longer strings of active nodes).

A drawback in practice is the large number of pointers per node. The sibling and last child pointers are redundant, and increase both time and space. These three pointers per node can be replaced as follows. Each node x has a *child table*, i.e., an array of pointers to its children, indexed by rank. The siblings of a node can be read from the child table of its parent in $O(1)$ time.

The disadvantage of child tables is that a rank r node needs a table of r entries, and r can be $\lfloor \log n \rfloor$. Hence a uniform node size increases the space requirement to $\Theta(n \log n)$. However a two-tiered data structure can reduce the space to linear— in fact to $(1 + \epsilon)n$ words, for any $\epsilon > 0$. Further, the disadvantage of a nonuniform data structure seems to be compensated by the simplicity of the second tier. Details are as follows.

Form $n/\log n$ *groups* of $\log n$ items (the last group may have fewer items). The grouping can be arbitrary. For instance if the items are numbered from 1 to n and accessed by number (as can be arranged in many applications) the groups are consecutively numbered items; no extra storage is needed for the group structure. Each group is represented by a node g in a relaxed heap of $n/\log n$ nodes; the key of g is the smallest key of an item x in the group corresponding to g; g stores a pointer to x (so $k(g)$ need not be stored); the relaxed heap uses child tables. To do *decrease_key*(x, v) the algorithm updates the key of the group corresponding to x; if it decreases, the group is *promoted* in the relaxed heap. To do *delete_min*(x), the group corresponding to x is scanned for the undeleted item y with smallest key; a pointer to y replaces the pointer to x in the relaxed heap node, and the algorithm follows the *delete_min* algorithm without reclamation. Clearly the storage is $O(n)$; if the relaxed heap uses w words per node, choosing a group size of w/ϵ gives ϵn storage for the relaxed heap. The time bound is unchanged if $w = O(\log n)$.

Child tables may be used with Fibonacci heaps but are not as effective. In Fibonacci heaps an arbitrary child of a node may be deleted without replacement. This causes arbitrary deletions of entries in the child tables. This forces the algorithm to keep track of the free entries in a child table, by a free storage list. This overhead is not present in relaxed heaps, where only a highest rank entry gets deleted without replacement.

The algorithm given so far can be used instead of Fibonacci heaps in two important applications, Dijkstra's shortest path algorithm and the computation of minimum spanning trees. It is easy to extend the algorithm to support all the operations supported by Fibonacci heaps. To delete an arbitrary node, decrease its key to $-\infty$ and then do a *delete_min*. The amortized time

is $O(\log n)$. A node can be inserted in $O(1)$ amortized time. This is done by the technique given above for initializing the heap, if n is known in advance. Alternatively, the algorithm makes the inserted node into a B_0 tree, and then repeatedly links B_r trees of equal rank until this is no longer possible. The amortized time for an insert is $O(1)$, since in a sequence of i inserts and d deletes the number of times B_r trees are linked by inserts is at most $i + d \log n$.

To merge two relaxed heaps, repeatedly link B_r trees of equal rank, until this is no longer possible. Use the A array of the first heap as the A array of the merged heap, but before discarding the A array of the second heap, *promote* each node in it. (Doing the promotions in order of decreasing rank ensures correctness: In a sequence of transformations $rank(b)$ is nondecreasing. Thus the transformations only work on portions of the heap that have valid relaxed heap structure). This algorithm achieves $O(\log n)$ amortized time for merging. This can be reduced to $O(1)$ amortized time (although such a reduction would probably not change the asymptotic running time for an applications program). The idea is to keep a list of all nonnull entries in each A array, and further, to delay linking B_r trees and processing A arrays until a *delete_min* operation occurs.

3. Run Relaxed Heaps.

Consider a relaxed binomial queue (defined as in Section 2). A *run* is a maximal sequence of two or more active siblings. A *singleton* is an active node that is not in a run. Rank relaxed heaps clearly do not have runs, but the heaps of this section do.

A *run relaxed heap* is a relaxed binomial queue with at most $\lfloor \log n \rfloor$ active nodes, i.e., $\alpha \leq \lfloor \log n \rfloor$ (recall that α is the number of active nodes). This definition allows *decrease_key* to be implemented in $O(1)$ time. The idea is that if a *decrease_key* makes $\alpha > \lfloor \log n \rfloor$ then there are two active nodes a, a' with equal rank. Hence a *pair transformation* can decrease α in $O(1)$ time. Unfortunately if a or a' is not a last child, the *pair transformation* does not apply. A new transformation for runs can be used to handle this situation. The details are as follows. In the rest of this section relaxed heap refers to run relaxed heap, unless stated otherwise.

The algorithm keeps track of the run- and singleton- structure of the active nodes, which we refer to as the *run-singleton* structure. The run-singleton structure can change as a result of a transformation creating a new bad node or rearranging nodes. The bookkeeping details for the run-singleton structure are relatively straightforward and will be postponed until after the transformations.

There are two transformations, for runs and for pairs. In this section a transformation does $O(1)$ work and decreases α, and then stops. Unlike Section 2, there are no sequences of tranformations. Instead if a transformation creates a new bad node b, b is added to the run-singleton structure. Transformations do the rearrangements of Figures 2-5, but these figures are interpreted with one difference: Nodes are *linked* rather than *combined*. There is no need to combine nodes since the new definition of relaxed heap does not require a cleaning operation. Now we give the details of the transformations.

The *pair transformation* is a combination of transformations of Section 2. It is given two singletons a, a' of equal rank r. We use the same notation as Section 2— a and a' have parents p and p' and grandparents g and g', respectively. There are three cases. The first is when a and a' are both last children. Then Figure 2 applies: Without loss of generality $k(p) \leq k(p')$. The transform makes p' the rank r child of p, and links a and a' to form a B_{r+1} tree, whose root becomes the rank $r+1$ child of g'. This case of the pair transformation operates as desired: it does $O(1)$ work and decreases α. Note this transformation is still correct if a or a' is in a run. The only difference is how the transform affects the run-singleton structure: When a and a' are singletons the only change is a possible new run or singleton if the new child of g' is bad. When a or a' starts out in a run that run also changes. The second case of the pair transform uses this first case, with a or a' possibly in a run.

The second case of the pair transform is when exactly one of a and a', say a, is not a last child. Thus a has a rank $r+1$ sibling s; since a is a singleton, s is good. Let c be the last child of s. If c is bad Figure 5 applies: a first-case pair transform is done for a and c (note c may be in a run). If c is good Figure 3 applies: a cleaning operation makes a a last child. Now a first-case pair transform applies to a and a' (although a may enter a run in the cleaning operation, it leaves the run in the pair transform).

The last case of the pair transform, when neither a nor a' is last, is similar: The algorithm processes a as above; if Figure 5 applies and a first-case pair transform is done (so α decreases) it stops. Otherwise it processes a' similarly. If α still has not decreased, a first-case pair transform is done for a and a'.

The second transformation is a *run transformation*. Let a be the largest rank child of the given run. Let a have rank r, parent p, and rank $r-1$ sibling t; t must be active. The first case is if a is a last child. Then Figure 4 applies: t and p are linked, and then t and a are linked. This decreases α as desired. (The only possible change in the run-singleton structure involves the new child of g).

The last case is when a is not last. Then a has a rank $r+1$ sibling s, which must be good (see

Figure 7). Let d and c be the rank $r-1$ and rank r children of s, respectively. Children t, b, s, d and c are removed from their parents. Nodes d and c are made the rank $r-1$ and rank r children of p, respectively. Since s was good, this does not increase α. Then t and s are linked to form a B_r tree with root t; next t and a are linked, and the new root is made the rank $r+1$ child of p. The result is that α decreases by one, so this transformation operates as desired. (The possible changes in the run-singleton structure involve the new children of p of ranks $r-1$ through $r+1$, and the remainder of a possible run initially containing d). This completes the description of the run transformation.

Now we give the details of the run-singleton data structure. A bit is used to *mark* each active node. Each rank r has a list $S(r)$ of all rank r singletons. The *pair list* is a list of ranks r with $|S(r)| \geq 2$. The *run list* is a list of all active nodes that have the largest rank in a run. All lists are double-linked, and each node points to its occurrence in a list (if any). In addition, the nodes of the binomial trees have the same pointers as in Section 2.

The following procedure updates the run-singleton data structure when a node a becomes active. Let a have rank r, with rank $r+1$ sibling s and rank $r-1$ sibling t (s or t may not exist). Node a is marked active. Then the appropriate one of these three cases is executed:

(1) If neither s nor t is active then a is a singleton. It is added to list $S(r)$. If now $|S(r)| = 2$, r is added to the pair list.

(2) If s is active then a is in a run. If s was previously a singleton it is removed from $S(r+1)$ and added to the run list. If this makes $|S(r+1)| = 1$ then $r+1$ is removed from the pair list.

(3) If s is not active but t is, then a is in a run. a is added to the run list. t is removed from the run list or $S(r-1)$; if in the latter case this makes $|S(r-1)| = 1$ then $r-1$ is removed from the pair list.

This concludes the update algorithm.

To do *decrease_key*(x, v), $k(x)$ is changed to v. If this makes x bad it is inserted into the run-singleton structure, using the above update algorithm. The rest of the processing ensures $\alpha \leq \lfloor \log n \rfloor$: If the pair list is nonempty, a rank r is removed from it; a pair transformation is done for the first two elements on $S(r)$ (any necessary changes to the run-singleton structure are made); if we still have $|S(r)| \geq 2$, r is added to the pair list. Similarly, if the run list is nonempty a node a is removed from it; a run transformation is done for the run ending at a (any necessary changes to the run-singleton structure are made); if nodes are still left in the run previously containing a, the largest node is added to the run list or a singleton list, as appropriate.

The time for *decrease_key*(x, v) is clearly $O(1)$. Correctness follows from the fact that if x becomes active and makes $\alpha > \lfloor \log n \rfloor$ then two active nodes have the same rank. Hence the pair list or run list is nonempty, so a transformation decreases α to a permissible value.

The *delete_min* routine is similar to Section 2. First the smallest node x is found. This involves examining all tree roots and all active nodes. Singletons are found using the S lists; runs are found by getting the largest node from the run list and following sibling pointers in the tree. Then x is replaced by a node resulting from linking all of its former children. Various nodes that become good are removed from the run-singleton structure. (The procedure for removing nodes from the structure is similar to the above update routine). The definition of relaxed heap ensures that the total time is $O(\log n)$.

The data structure can be extended to handle the operation *insert*(x) in $O(1)$ time. The approach is similar to *decrease_key*. Let r denote the number of trees in any collection of trees. A run relaxed heap is now defined as a collection of relaxed binomial trees with $\alpha \leq \lfloor \log n \rfloor$ and $r \leq \lfloor \log n \rfloor + 1$ (note α refers to the number of active nodes in the entire collection). The algorithm keeps a pointer $T(r)$ to a binomial tree of rank r, for each r; the remaining trees are kept in the *tree list*, a list of pairs of trees of the same rank.

The subroutine *add_tree*(x) puts a new binomial tree with root x into the structure: Letting $r = rank(x)$, if $T(r) = nil$ then x becomes $T(r)$; otherwise x and $T(r)$ are made into a pair which is added to the tree list and $T(r)$ becomes *nil*. The operation *insert*(x) makes x into a B_0 tree and does *add_tree*(x). Then it decreases the number of trees, if possible: It removes the first pair from the tree list, links the two trees to form a tree with root y, and does *add_tree*(y). The time for *insert*(x) is clearly $O(1)$.

Correctness follows from the fact that if in *insert*(x), *add_tree*(x) makes $r > \lfloor \log n \rfloor + 1$ then the tree list is nonempty. Hence the second step of *insert* decreases r to a permissible value.

The new data structures have no effect on *decrease_key*, since the transformations and the run-singleton data structure are independent of the arrangement of the trees. The reclaiming version of *delete_min* creates new trees; it is modified to repeatedly link trees of equal rank, so the tree list becomes empty. This does not change the time bound since $O(\log n)$ trees are involved.

Theorem 3.1. Run relaxed heaps correctly process *decrease_key* and *insert* in $O(1)$ time and *delete_min* in $O(\log n)$ time. ∎

Many details of the run relaxed heap algorithms can be modified without changing the above bounds. For instance the pair transform can be simplified if it is only done when there are no runs.

References.

[B] M. R. Brown, "Implementation and analysis of binomial queue algorithms", *SIAM J. Comput.*, 7, 3, 1978, pp. 298-319.

[FT] M.L. Fredman and R.E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms", *Proc. 25th Annual Symp. on Found. of Comp. Sci.*, 1984, pp. 338-346; *J. ACM*, to appear.

[GGST] H.N. Gabow, Z. Galil, T.H. Spencer and R.E. Tarjan, "Efficient algorithms for finding minimum spanning trees in undirected and directed graphs", *Combinatorica 6*, 2, 1986, pp. 109-122.

[V] J. Vuillemin, "A data structure for manipulating priority queues", *C. ACM*, 21, 1978, pp. 309-314.

Figure Captions.

Figure 1. Recursive definition of binomial tree B_{r+1}.

Figure 2. Pair transformation.

Figure 3. Cleaning.

Figure 4. Active sibling transformation.

Figure 5. Good sibling transformation: c active.

Figure 6. Rank relaxed heap algorithm.

Figure 7. Run transformation.

```
procedure decrease_key(x, v);
begin k(x) ← v; promote(x) end ;

procedure promote(a);
begin
let a have rank r, parent p, next larger sibling s and grandparent g;
{these variables are global for the transformations }
if k(a) < k(p) then
    if a is the last child of p then begin
        if A(r) = nil then A(r) ← a else if A(r) ≠ a then pair_transform end
    else if s is active then active_sibling_transform else good_sibling_transform
end ;

procedure combine(a1, a2, c);
begin {a1 and a2 have equal rank k }
link a1 and a2 to a rank k + 1 tree with root c ; clean c;
end ;

procedure pair_transform;
begin
a' ← A(r); A(r) ← nil; let a' have parent p' and grandparent g';
remove a and a' from their parents {a, a', p, p' now have rank r } ;
wlog k(p) ≤ k(p');
make p' the rank r child of p;
combine(a, a', c); make c the rank r + 1 child of g';
if A(r + 1) = p' then A(r + 1) ← c else promote(c);
end ;

procedure active_sibling_transform;
begin
remove a, s, p from their parents {a and p now have rank r, and s has rank r+1} ; A(r+1) ← nil;
combine(p, a, a);
combine(a, s, c); make c the rank r + 2 child of g;
if A(r + 2) = p then A(r + 2) ← c else promote(c);
```

```
end ;

procedure good_sibling_transform;
begin
let c be the last child of s;
if c is active then pair_transform
else begin clean p; {now c is good, a is a bad last child } ; promote(a) end end ;

procedure delete_min(x);
begin
x ← the node of smallest key from among the root nodes of the queue and the A(r) values;
y ← the root node of smallest rank;
remove y from its tree, creating rank(y) new roots with the smallest ranks in the queue;
if x ≠ y then
    begin
    for each child c of x in order of increasing rank do combine(y, c, y);
    delete x from its tree and replace it by y;
    let p be the parent of y;
    A(rank(y)) ← if k(y) < k(p) then y else nil;
end end ;
```

Figure 6. Rank relaxed heap algorithm.

B_{r+1} :

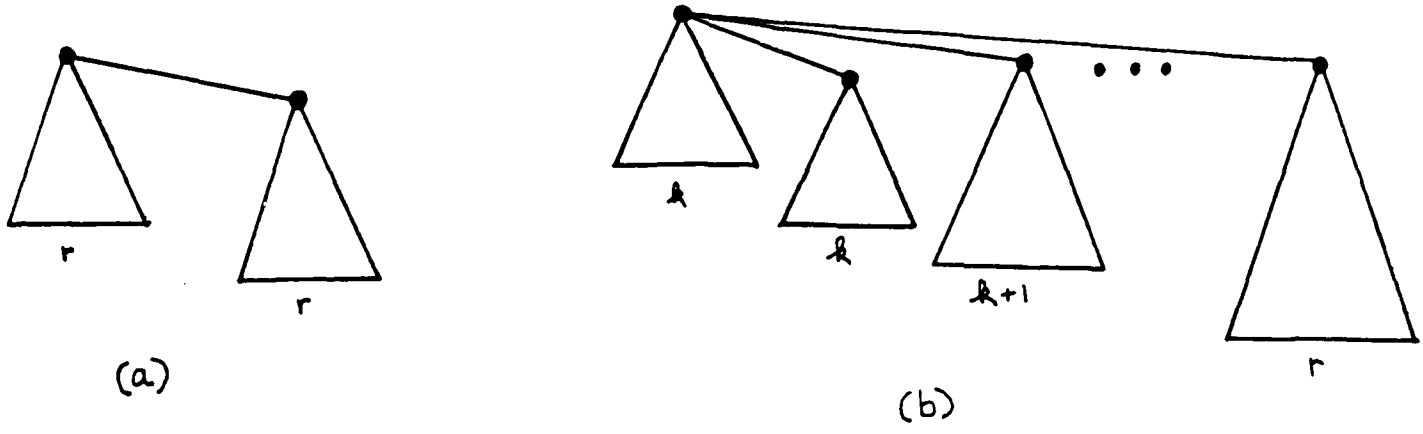


Fig. 1

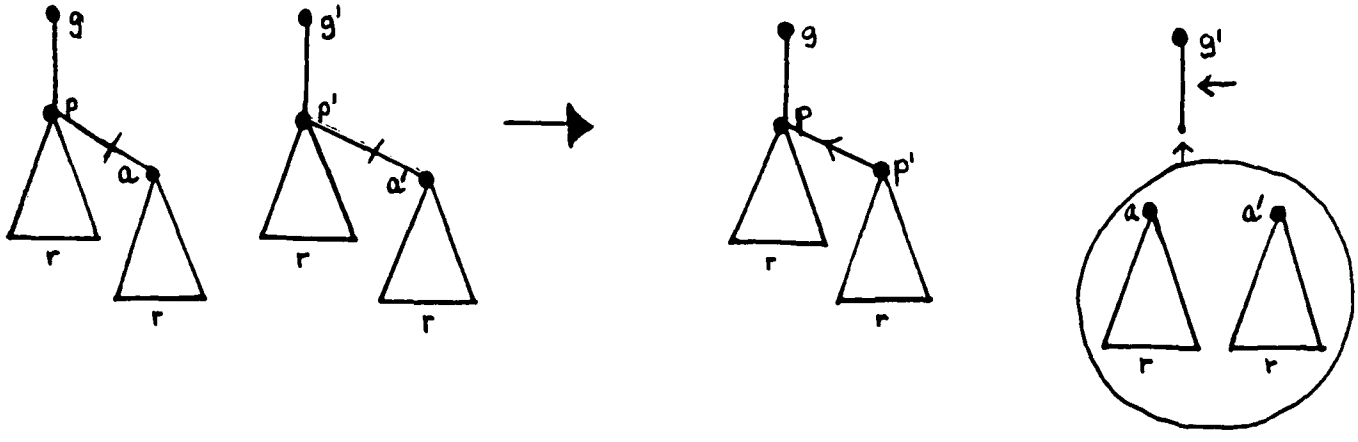


Fig. 2

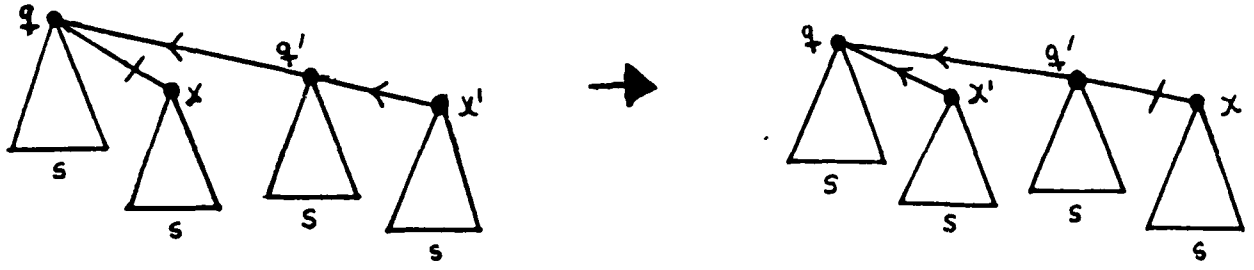


Fig. 3

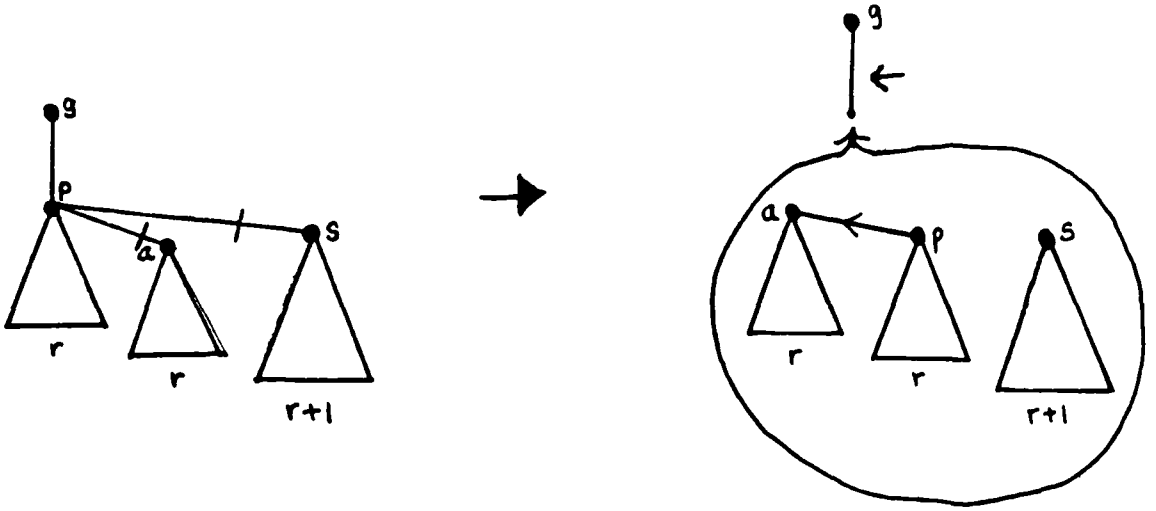


Fig. 4

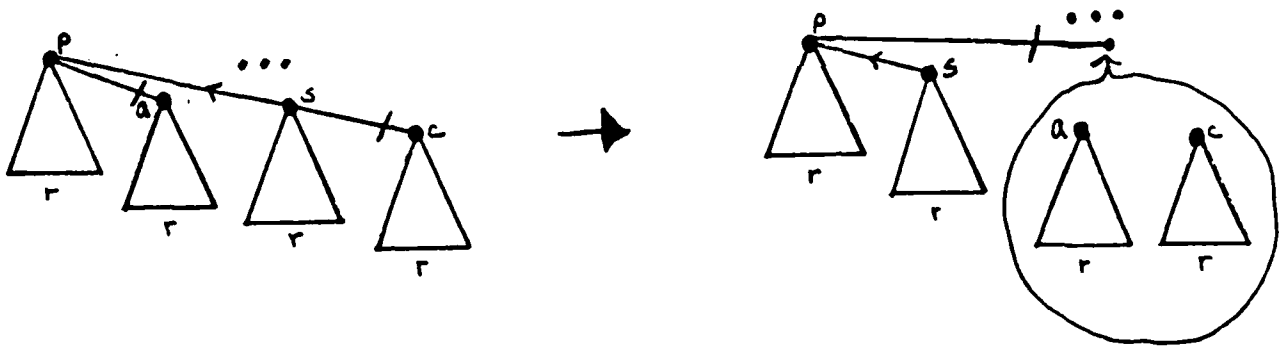


Fig. 5

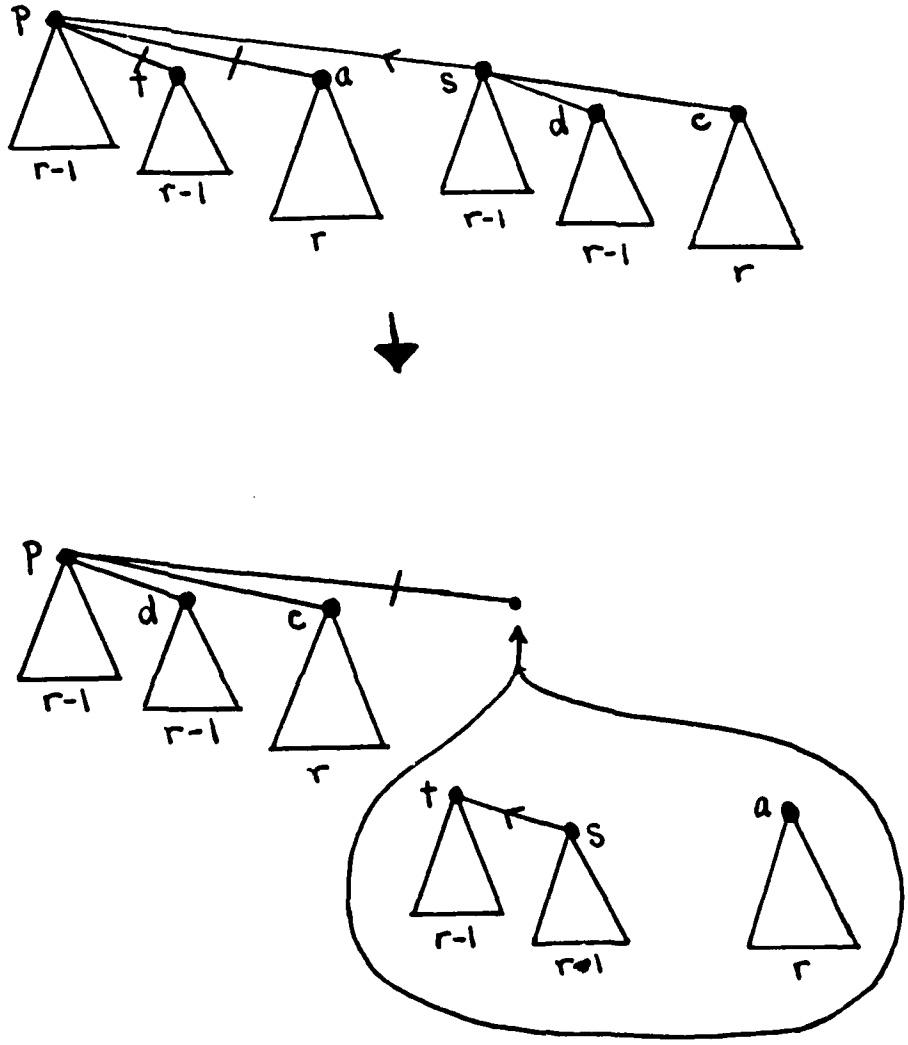


Fig. 7

END
DATE
FILMED
7-88
DTIC