Algorithmica (1986) 1: 111–129

The Pairing Heap: A New Form of Self-Adjusting Heap

Michael L. Fredman[1,4], Robert Sedgewick[2,5], Daniel D. Sleator[3], and Robert E. Tarjan[2,3,6]

Abstract. Recently, Fredman and Tarjan invented a new, especially efficient form of heap (priority queue) called the *Fibonacci heap*. Although theoretically efficient, Fibonacci heaps are complicated to implement and not as fast in practice as other kinds of heaps. In this paper we describe a new form of heap, called the *pairing heap*, intended to be competitive with the Fibonacci heap in theory and easy to implement and fast in practice. We provide a partial complexity analysis of pairing heaps. Complete analysis remains an open problem.

Key Words. Data structure, Heap, Priority queue

1. Introduction.

A *heap* or *priority queue* is an abstract data structure consisting of a finite set of items, each having a real-valued *key*. The following operations on heaps are allowed:

make heap (h): Create a new, empty heap named h.
find min (h): Return an item of minimum key from heap h, without changing h.
insert (x, h): Insert item x, with predefined key, into heap h, not previously containing x.
delete min (h): Delete an item of minimum key from heap h and return it. If h is originally empty, return a special *null* item.

The *find min* operation can be implemented as a *delete min* followed by an *insert*, but it is generally more efficient to implement it independently. Additional operations on heaps are sometimes allowed, including the following:

meld (h_1, h_2): Return the heap formed by taking the union of the item-disjoint heaps h_1 and h_2. Melding destroys h_1 and h_2.
decrease key (Δ, x, h): Decrease the key of item x in heap h by subtracting the

[1] EECS Department, University of California, San Diego, La Jolla, California 92093, USA
[2] Computer Science Department, Princeton University, Princeton, New Jersey 08544, USA
[3] AT & T Bell Laboratories, Murray Hill, New Jersey 07974, USA
[4] Research partially supported by National Science Foundation Grant MCS 82-04031 and by Bell Communications Research
[5] Research partially supported by National Science Foundation Grant DCR 85-14922
[6] To whom correspondence should be addressed.

non-negative real number Δ.
delete (x, h): Delete item x from heap h, known to contain it.

In order for *decrease key* and *delete* to be efficiently implementable, the location of item x in the representation of heap h must be known; standard implementations of heaps do *not* support efficient searching for an item. In our discussion we shall assume that a given item is in only one heap at a time.

Since n real numbers can be sorted by performing n *insert* operations followed by n *delete min* operations on an initially empty heap, the amortized time* of a heap operation for any implementation that uses binary decisions is $\Omega(\log n)$, where n is the heap size. There are many well-known heap implementations for which this bound is tight in the worst case per operation. Such implementations include the *implicit heaps* of Williams [16], utilized by Floyd in an elegant in-place sorting algorithm [3]; the *leftist heaps* of Crane [2] as modified by Knuth [9]; and the *binomial heaps* of Vuillemin [15], studied extensively by Brown [1]. Implicit heaps do not support melding; both leftist and binomial heaps do.

Recently Fredman and Tarjan [4] invented a new kind of heap called the *Fibonacci heap*. The operations *make heap*, *find min*, *insert*, *meld*, and *decrease key* taken only $O(1)$ amortized time on Fibonacci heaps, whereas *delete min* and *delete* take $O(\log n)$ amortized time. The importance of Fibonacci heaps is that in many network optimization algorithms that use heaps, *decrease key* is the dominant operation, and reducing the time for this operation improves the overall efficiency of such algorithms. Thus improved running times for a variety of network optimization algorithms can be obtained. See [4, 5, 6].

Fibonacci heaps have two drawbacks: They are complicated to program, and they are not as efficient in practice as theoretically less efficient forms of heaps, since in their simplest version they require storage and manipulation of four pointers per node, compared to the two or three pointers per node needed for other structures. Our goal in this paper is to devise a "self-adjusting" form of heap having the same theoretical time bounds as the Fibonacci heap, yet easy to implement and fast in practice. A step in this direction was taken by Sleator and Tarjan [10, 11], who devised a data structure called the *skew heap*. The skew heap can be regarded as a self-adjusting version of the leftist heap. On the "bottom-up" form of skew heaps, *make heap*, *find min*, *insert*, and *meld* take $O(1)$ amortized time and *delete min*, *delete*, and *decrease key* take $O(\log n)$ amortized time. The problem remaining is to find a simple data structure that reduces the amortized time for *decrease key* to $O(1)$.

The data structure proposed in this paper, called the *pairing heap*, can be regarded as a self-adjusting version of the binomial heap. It shares with skew heaps ease of implementation and practical efficiency. We conjecture but are unable to prove that pairing heaps are theoretically as efficient as Fibonacci heaps

*By *amortized time* we mean roughly the time of an operation averaged over a worst-case sequence of operations. An exact definition is provided in Section 2. For a thorough discussion of this concept see [14].

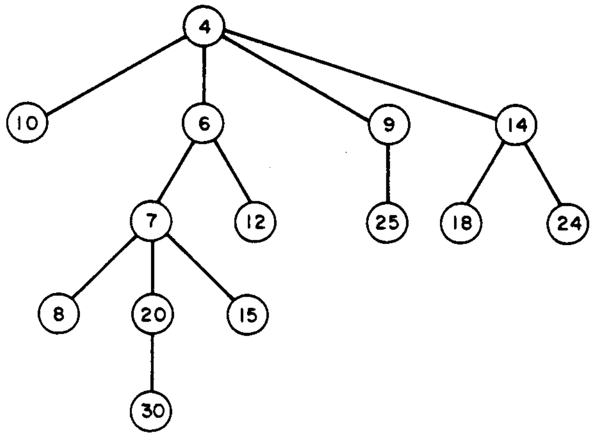

Fig. 1. An endogenous heap-ordered tree. The numbers in the nodes are keys.

(in the amortized case and ignoring constant factors). Our best analysis gives an $O(\log n)$ time bound per heap operation.

The paper contains two sections in addition to this introduction. In Section 2 we motivate, describe, and partially analyze pairing heaps. In Section 3 we propose some variants of pairing heaps that seem to have similar efficiency.

2. Pairing Heaps.

We shall represent a heap by an *endogenous heap-ordered tree*. (See Figure 1.) This is a rooted tree in which each node is a heap item, with the items arranged so that the parent of any node has key no greater than that of the node itself. (The term "endogenous" means that we do not distinguish between a tree node and the corresponding heap item; see [13].)

As a primitive for combining two heap-ordered trees, we use *linking*, which makes the root of smaller key the parent of the root of larger key, with a tie broken arbitrarily. (See Figure 2.) If we use an appropriate tree representation, a linking operation takes $O(1)$ time in the worst case.

Of the heap operations, *delete min* is the most important and the most complicated to implement. Thus we shall discuss the other operations first. We

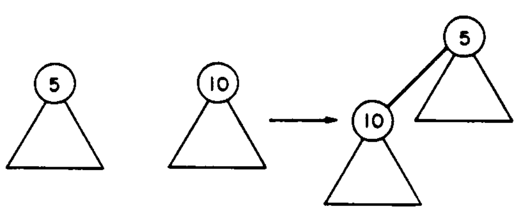

Fig. 2. Linking two heap-ordered trees. The triangles denote trees of arbitrary structure.

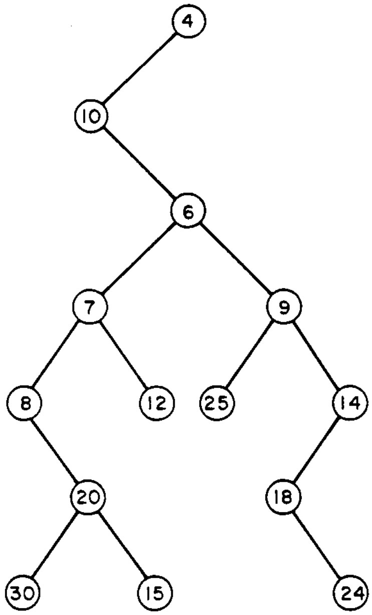

Fig. 3. The half-ordered binary tree corresponding to the heap-ordered tree of Figure 1.

carry out these operations as follows:

make heap (h): Create a new, empty tree named h.
find min (h): Return the root of tree h.
insert (x, h): Make x into a one-node tree and link it with tree h.
meld (h_1, h_2): Return the tree formed by linking trees h_1 and h_2.
decrease key (Δ, x, h): Subtract Δ from the key of item x. If x is not the root of tree h, cut the edge joining x to its parent and link the two trees formed by the cut.

We perform *delete* using *delete min*:
delete (x, h): If x is the root of tree h, perform a *delete min* on h. Otherwise, cut the edge joining x to its parent, perform a *delete min* on the tree rooted at x, and link the resulting tree with the other tree formed by the cut.

The data structure we use to make these implementations efficient is the *child, sibling representation* of a tree, also known as the *binary tree representation* [8]. Each node has a *left pointer* pointing to its first child and a *right pointer* pointing to its next older sibling. This representation allows us, and indeed forces us, to order the children of every node, a fact that we shall exploit below. The effect of the representation is to convert a heap-ordered tree into a half-ordered binary tree with empty right subtree, where by *half-ordered* we mean that the key of any node is at least as small as the key of any node in its left subtree. (See Figure 3.) In order to make *decrease key* and *delete* efficient, we must store with each node a third pointer, to its parent in the binary tree.

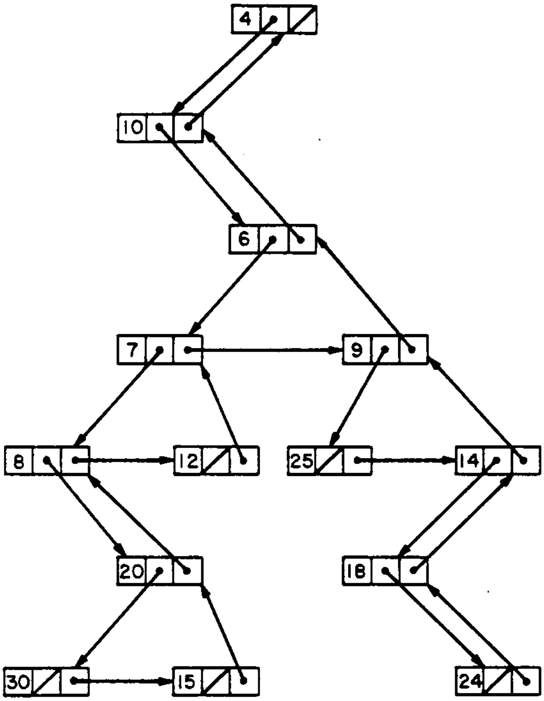

Fig. 4. Two-pointer representation allowing parent access of the binary tree in Figure 3. The bits indicating left and right only children are omitted.

Remark. Instead of using three pointers per node, we can manage with only two, at a cost of a constant factor in running time. We make each node in the binary tree point to its left child, and to its right sibling or to its parent if it has no right sibling. (See Figure 4.) With each only child we must also store a bit indicating whether it is a left child or a right child. □

Each of the operations *make heap*, *find min*, *insert*, *meld*, and *decrease key* has an $O(1)$ worst-case running time. A *delete* operation takes $O(1)$ time plus one *delete min* operation. Thus *delete min* and *delete* are the only non-constant-time operations.

Let us consider how to perform *delete min* on a heap-ordered tree. We begin by removing the tree root, which is an item of minimum key. This produces a collection of heap-ordered trees, one rooted at each child of the deleted node. We combine all these trees by linking operations to form one new tree. The order in which we combine the trees is important, however.

Whatever combining rule we choose will have a $\Theta(n)$ worst-case time bound, since we can build *any* n-node tree, in particular the tree with a root and $n - 1$ children, by a suitable sequence of $O(n)$ *make tree*, *insert*, and *meld* operations. However, for a suitable combining rule we shall be able to prove an $O(\log n)$ amortized bound.

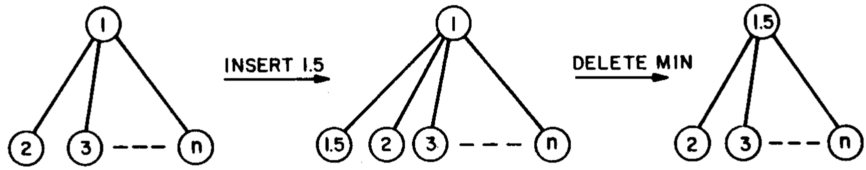

Fig. 5. The naive version of *delete min* takes $O(n)$ amortized time. After the root with key 1 is deleted, the second and successive children are linked to the first.

The naive combining rule is to choose one of the trees and successively link each of the remaining ones with it. Unfortunately this method takes $\Theta(n)$ amortized time per operation: Figure 5 shows that an *insert* followed by a *delete min* can take $\Omega(n)$ time while recreating the initial tree structure.

A more promising way of combining the trees is to make one pass linking them in pairs and then a second pass linking each of the remaining trees with a selected one. Still, if we are not careful about how the trees are paired during the first pass, this method can take $\Theta(\sqrt{n})$ amortized time. The example of Figure 6 shows that scrambling the trees before the pairing pass can cause an *insert* followed by a *delete min* to take $\Omega(\sqrt{n})$ time while recreating the initial tree structure. On the other hand, a simple analysis gives an $O(\sqrt{n})$ amortized bound no matter how the trees are scrambled, thus showing that this method is at least somewhat better than the naive one.

To derive the upper bound, we shall use the "potential" technique of Sleator (see [14]). Introducing this technique also allows us to clarify the notion of "amortized time." To each configuration of the data structure we assign a real number Φ called the *potential* of the configuration. For any sequence of m operations, we define the *amortized time* a_i of the i^{th} operation by $a_i = t_i + \Phi_i - \Phi_{i-1}$, where t_i is the actual time of the i^{th} operation and Φ_{i-1} and Φ_i are the potentials before and after the operation, respectively. That is, the amortized time of an operation is its actual running time plus the net increase in potential it causes. Summing over all the operations, we have

$$(1) \qquad \sum_{i=1}^{m} t_i = \sum_{i=1}^{m} (a_i - \Phi_i + \Phi_{i-1}) = \left(\sum_{i=1}^{m} a_i \right) - \Phi_m + \Phi_0$$

If the potential is chosen so that it is initially zero and is always non-negative, then (1) implies

$$(2) \qquad \sum_{i=1}^{m} t_i \le \sum_{i=1}^{m} a_i$$

That is, the total amortized time is an upper bound on the total actual time. This means that the amortized time of an operation can be used as a conservative estimate of its actual running time, as long as total running time is the measure of interest.

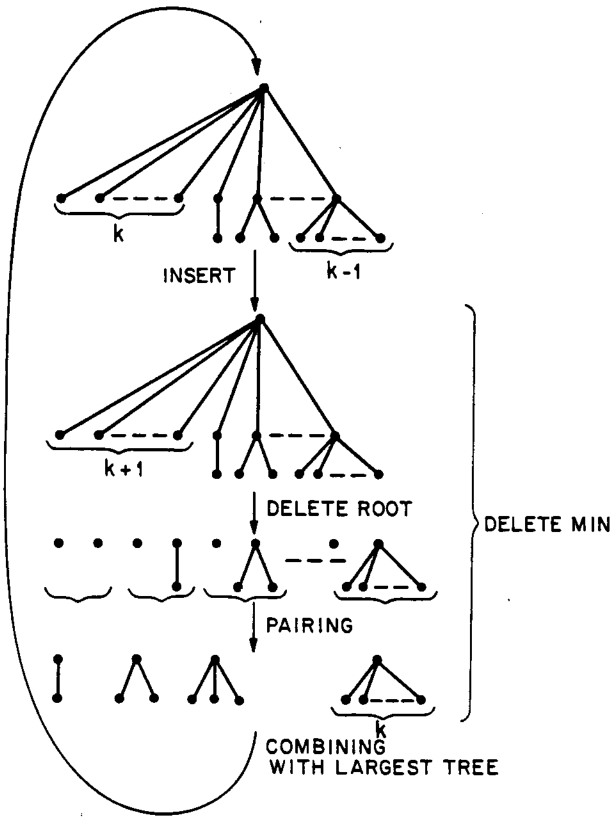

Fig. 6. Scrambled pairing during a *delete min* operation takes $\Omega(\sqrt{n})$ amortized time. Here $n = k(k+3)/2$. For clarity, the keys of nodes are not shown.

To analyze scrambled pairing, we define the potential of a node with d children in an n-node heap to be $1 - \min\{d, [\sqrt{n}]\}$. We define the potential of a collection of heaps to be the sum of the potentials of its nodes. Observe that the potential of an empty heap is zero, and the potential of any collection of heaps is non-negative, since the sum over n nodes of their numbers of children is the total number of nodes minus the number of trees. Thus (2) holds. A linking operation can only decrease the potential, and cutting an edge can increase the potential by at most one (as long as the heap size does not change). Since *make tree*, *find min*, *insert*, *meld*, and *decrease key* all take $O(1)$ actual time and perform $O(1)$ links and cuts, each has an $O(1)$ amortized time bound.

Consider a *delete min* operation. Removing the tree root causes a potential increase of at most $2\sqrt{n}$, of which one $\sqrt{n}$ accounts for the increase in the potential of the deleted root (from at least $1 - \sqrt{n}$ to 0), and the other $\sqrt{n}$ accounts for one unit of increase per node having at least $\sqrt{n}$ children (such an increase can be caused by the decrease in heap size by one.) Suppose that k trees remain after deleting the root. The actual time of the *delete min* is $O(k)$. Since we are ignoring constant factors, let us estimate this time as one plus the number of links in the pairing pass, or $\lfloor k/2 \rfloor + 1$. Each of the links in the pairing pass causes the potential to drop by one except for links that add a child to a node already having $\sqrt{n}$ children. There can be at most $\sqrt{n}$ of these exceptional links, since each corresponds to a disjoint tree containing at least $\sqrt{n}$ nodes. Thus the links cause a potential drop of at least $\lfloor k/2 \rfloor - \sqrt{n}$. Summing the estimate of actual time plus the potential changes, we see that the amortized time of *delete min* is $\lfloor k/2 \rfloor + 1 + 2\sqrt{n} + (\sqrt{n} - \lfloor k/2 \rfloor) = O(\sqrt{n})$. The same estimate holds for *delete*.

To obtain an algorithm that is theoretically competitive with the known heap implementations, we use the pairing method of combining trees but choose the trees to be paired carefully. We order the children of each node in the order they were attached by linking operations, with the first (youngest) child being the one most recently attached. That is, when a node y is made the child of a node by linking, y becomes the first child of x. Note that this ordering of children is independent of key order. To perform a *delete min* operation, we remove the tree root and link the first and second remaining trees, then the third and fourth, and so on. (If the original root had an odd number of children, one tree remains unlinked.) Then we link each remaining tree to the last one, working from the

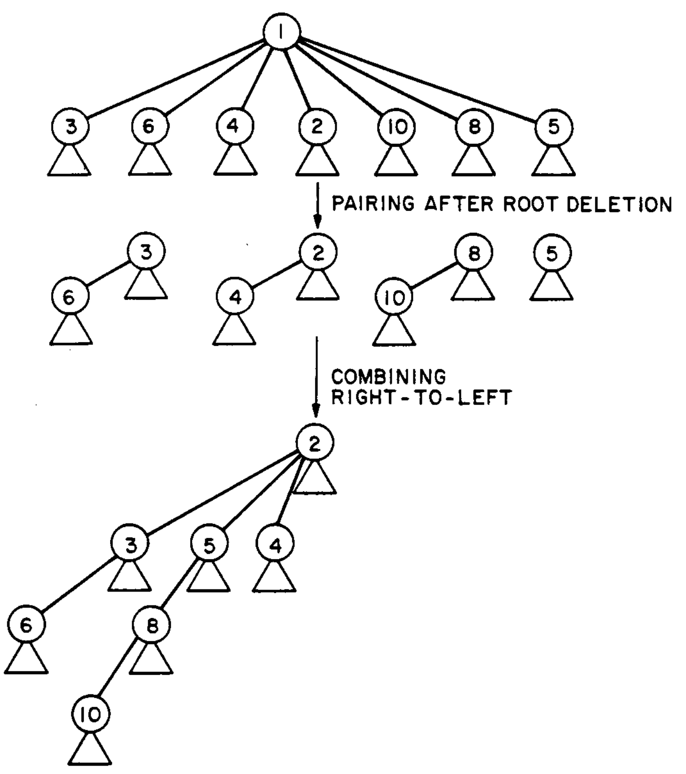

Fig. 7. A *delete min* operation on a pairing heap.

next-to-last back to the first, in the opposite order to that of the pairing pass. (See Figure 7.)

We call the resulting data structure the *pairing heap*. We believe that this data structure is as efficient as Fibonacci heaps in the amortized case. That is, we make the following conjecture:

Conjecture 1. The various operations on pairing heaps have the following amortized running times: $O(1)$ for *make heap*, *find min*, *insert*, *meld*, and *decrease key*, and $O(\log n)$ for *delete min* and *delete*.

We are unable to prove this conjecture. However, we can obtain the following weaker result:

THEOREM 1. On pairing heaps, the operations *make heap* and *find min* run in $O(1)$ amortized time, and the other operations run in $O(\log n)$ amortized time.

We shall prove Theorem 1 using the potential technique. To guide us in our choice of a potential function, let us examine the effects of a *delete min* operation

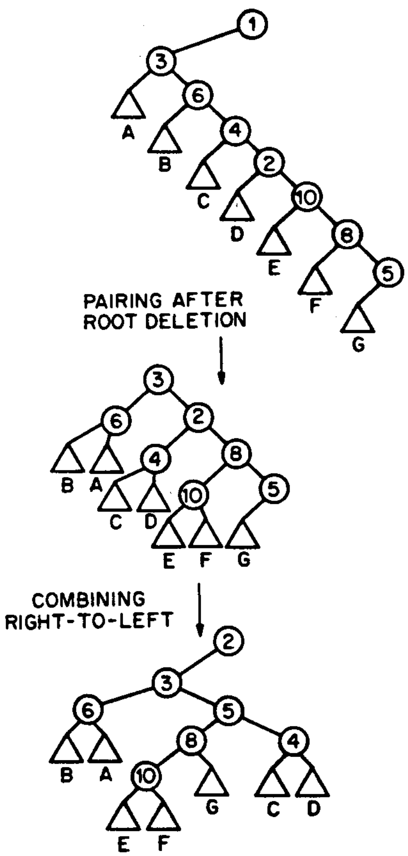

Fig. 8. A *delete min* operation on the binary tree form of the pairing heap in Figure 7. Note that a subtree in the ordinary form of a pairing heap corresponds to a node and its left subtree in the binary tree form.

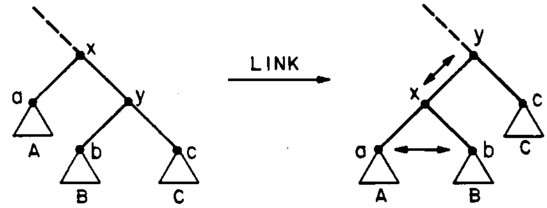

Fig. 9. The effect of a linking operation during a *delete min*. The figure shows the outcome if the key of node x is greater than the key of node y. If the key of x is less than that of y, nodes x and y are interchanged, as are subtrees A and B. This is indicated by the double arrows.

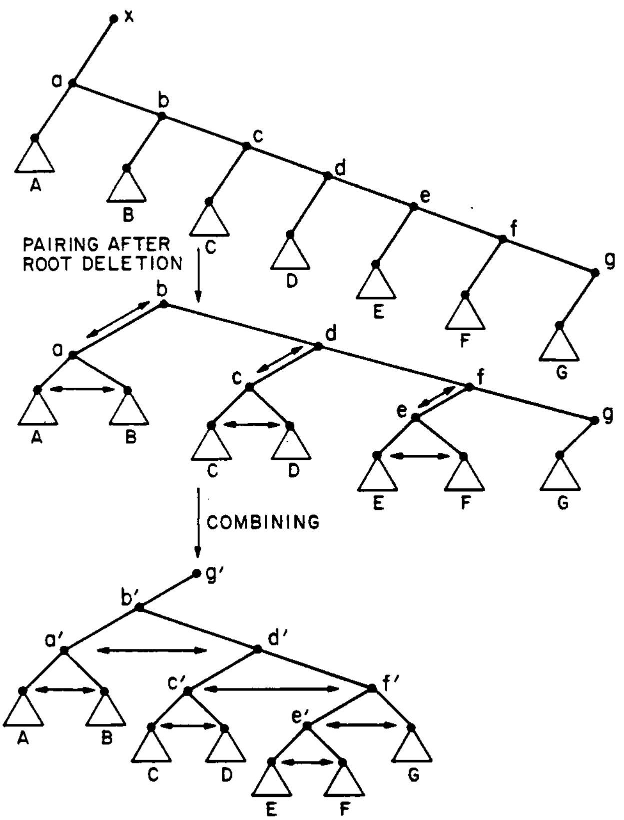

Fig. 10. The effect of a *delete min* on a half-ordered binary tree. The slanted double arrows (between a and b, c and d, e and f) denote possible interchange of single nodes. The horizontal double arrows denote possible interchange of the entire subtrees. Nodes $a', \ldots, g'$ are some permutation of nodes $a, \ldots, g$.

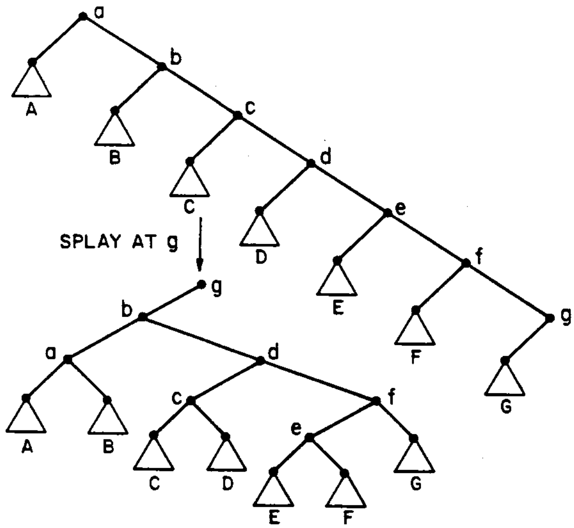

Fig. 11. Splaying at a node in a binary search tree.

on the binary tree representation of a heap. Figure 8 shows a *delete min* operation on the binary tree form of the heap in Figure 7. Figure 9 illustrates the general effect of a single linking operation. Figure 10 illustrates the general effect of an entire *delete min*. We see that up to permutation of nodes and exchange of left and right subtrees a *delete min* has essentially the same effect as discarding the root and splaying at the last node in symmetric order, where *splaying* is the heuristic used by Sleator and Tarjan in their self-adjusting search trees [10, 12]. (See Figure 11.) Thus it is not surprising that by using their potential function (which is invariant under exchange of left and right subtrees) we can prove Theorem 1.

We define the *size* $s(x)$ of a node x in a binary tree to be the number of nodes in its subtree including x, the *rank* $r(x)$ of x to be $\log s(x)$*, and the *potential* of a set of trees to be the sum of the ranks of all nodes in the trees. Then the potential of a set of no trees is zero and the potential of any set of trees is non-negative, so the sum of the amortized times is an upper bound on the sum of the actual times for any sequence of operations starting with no heaps.

Observe that every node in an n-node tree has rank between 0 and $\log n$. We immediately deduce that *make heap* and *find min* have an $O(1)$ amortized time bound, since they cause no change in potential. The operations *insert*, *meld*, and *decrease key* have an $O(\log n)$ amortized time bound, since each such operation causes an increase of at most $\log n + 1$ in potential: a link causes at most two

* We use binary logarithms throughout this paper.

nodes to increase in rank, one by at most $\log n$ and the other by at most 1, where n is the total number of nodes in the two trees. (Only the roots of the two trees can increase in rank. The root of initially smaller size can increase in rank by at most $\log n$, and the root of initially larger size can increase in rank by at most 1, since its size at most doubles.)

The hardest operation to analyze is *delete min*. Consider the effect of a *delete min* on a tree of n nodes. We shall estimate the running time of this operation as one plus the number of links performed. The number of links performed during the first pass (pairing) is at least as great as the number performed during the second pass (combining the remaining trees). Thus we shall charge two per link during the first pass. Let us estimate the potential change caused by a first-pass link. Referring to Figure 9, and assuming that subtree C is non-empty, we see that the increase in potential is $\log(s(a) + s(b) + 1) - \log(s(b) + (c) + 1)$. The concavity of the log function implies that $\log x + \log y$ for $x, y > 0$, $x + y \leq 1$ is maximized at value -2 when $x = y = 1/2$. It follows that

$$\begin{aligned}
&\log(s(a) + s(b) + 1) + \log(s(c)) - 2\log(s(a) + s(b) + s(c) + 2) \\
&\quad = \log((s(a) + s(b) + 1)/(s(a) + s(b) + s(c) + 2)) \\
&\qquad + \log(s(c)/(s(a) + s(b) + s(c) + 2)) \leq -2.
\end{aligned}$$

This and the inequality $\log(s(c)) \leq \log(s(b) + s(c) + 1)$ give $\log(s(a) + s(b) + 1) - \log(s(b) + s(c) + 1) \leq 2\log(s(a) + s(b) + s(c) + 2) - 2\log(s(c)) - 2$. Since $s(x) = s(a) + s(b) + s(c) + 2$, $2\log(s(x)) - 2\log(s(c)) - 2$ is an upper bound on the potential increase caused by the link. The only link during the first pass that can have subtree C empty is the last one. In this case the potential increase is at most $\log(s(a) + s(b) + 1) - \log(s(b) + 1) \leq 2\log(s(a) + s(b) + 2) = 2\log(s(x))$.

Now let us sum the potential increase over all first-pass links. Let $x_1, x_2, \ldots, x_{2k}$ be the nodes whose keys are compared during the first-pass links. That is, in the original binary tree x_1 is the left child of the root, x_{i+1} for $1 \leq i < 2k$ is the right child of x_i, and the last first-pass link involves comparing the keys of x_{2i-1} and x_{2i}. Let s denote the size function on the original binary tree. Then the potential increase caused by the first-pass links is at most

$$\begin{aligned}
&\sum_{i=1}^{k-1} (2\log s(x_{2i-1}) - 2\log s(x_{2i}) - 2) + 2\log s(x_{2k-1}) \\
&\quad \leq \sum_{i=1}^{k-1} (2\log s(x_{2i-1}) - 2\log s(x_{2i+1})) + 2\log s(x_{2k-1}) - 2(k-1) \\
&\qquad\qquad \text{since } s(x_{2i}) \geq s(x_{2i+1}) \\
&\quad \leq 2\log s(x_1) - 2(k-1) \quad \text{since the sum telescopes} \\
&\quad \leq 2\log n - 2(k-1) \qquad (3)
\end{aligned}$$

The other potential changes that take place during the *delete min* are a decrease of $\log n$ when the original tree root is removed and an increase of at most $\log(n-1)$ during the second pass. To verify the latter bound, we note that a one-to-one correspondence f can be established between the tree nodes before the second pass and the nodes after the second pass such that $s'(x) \geq s''(f(x))$ unless $f(x)$ is the tree root after the second pass. Here s' denotes the size function before the second pass and s'' denotes the size function after the second pass. (In Figure 10, the mapping f is given by $f(x) = x'$.) Thus the potential increase caused by the second pass can be associated with the tree root after the pass, and its magnitude is at most $\log(n-1)$.

It follows that the amortized time of the *delete min* operation is an actual time of $2k+1$ plus a potential increase of at most $2\log n - 2(k-1) - \log n + \log(n-1)$ for a total of at most $2\log n + 3$. An $O(\log n)$ bound on the amortized time of *decrease key* and *delete* follows immediately, finishing the proof of Theorem 1.

3. Variants of Pairing Heaps. The data structure proposed in Section 2 is not the only way to make use of the pairing idea. In this section we propose four variants of the structure. The first three involve changing only the implementation of

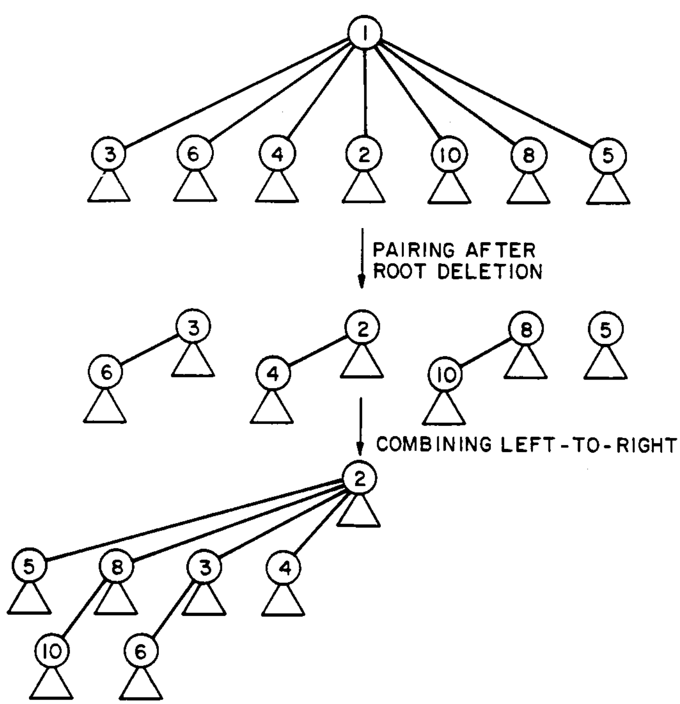

Fig. 12. A *delete min* operation using the front-to-back method.

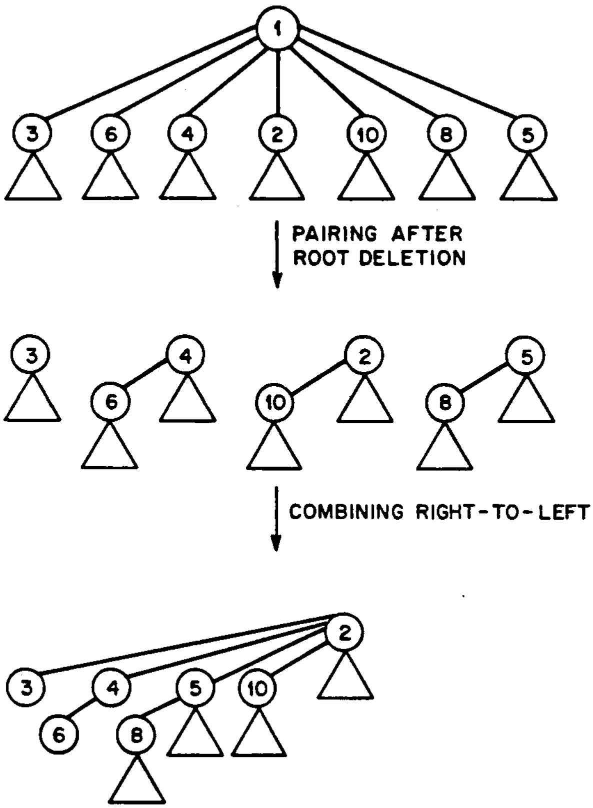

Fig. 13. A *delete min* operation using the back-to-front method.

delete min; the fourth uses a forest of trees instead of a single tree to represent a heap.

Instead of making the two passes of *delete min* in opposite directions (front-to-back followed by back-to-front), it seems natural to make them in the same direction, either both front-to-back (see Figure 12) or both back-to-front (see Figure 13). We call the former method the *front-to-back variant* and the latter the *back-to-front variant*. With either method the two passes can be combined into a single pass. In order to make the back-to-front variant a one-pass method, we must change the pointer structure representing the tree, since we must be able to access the children of a node in reverse order, older to youngest. One possibility is to use a *ring representation* in which the lists of children are singly linked in reverse order, with the first child pointing to the last and each parent pointing to its first child (see Figure 14). Additional pointers must be added to support *decrease key* and *delete*.

Another possible combining rule for *delete min* is to make repeated passes over the trees, linking them in pairs, until only one tree remains. (See Figure 15.) We call this the *multipass variant*.

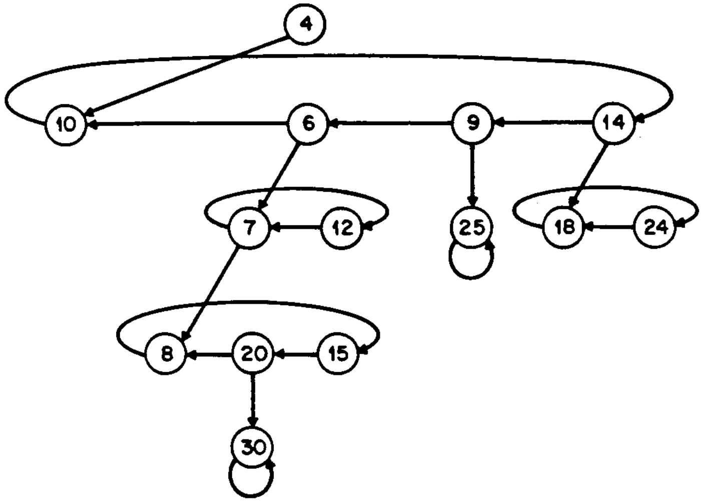

Fig. 14. The ring representation of the heap-ordered tree in Figure 1.

Our fourth method, the *lazy variant*, uses the multipass idea in combination with lazy linking. We represent a heap by a forest of rooted trees rather than a single tree. The trees in the forest are ordered in chronological order by the time they were added to the forest, least recent to most recent. To perform *find min*, we run through the trees once, linking them in pairs, and return any root of minimum key. To perform *insert*, we make the item to be inserted into a one-node tree and add it to the forest as the new last tree. To perform *delete min*, we carry out *find min*, delete the root of minimum key, and concatenate the list of subtrees rooted at its children to the back of the list of remaining trees. (See Figure 16.) To perform *meld*, we concatenate the two lists of trees. To perform *decrease key*(Δ, x, h), we subtract Δ from the key of x, cut the edge joining x to its parent if it has one, and if such a cut takes place we add the tree rooted at x to the back of the list of trees. To perform *delete*(x, h), we cut the edge joining x to its parent if it has one, delete node x, and concatenate the list of subtrees rooted at its children to the back of the list of remaining trees.

None of these variants is easy to analyze. We can prove Theorem 1 for the back-to-front variant using essentially the same analysis as in Section 2. For the multipass and lazy variants, we can prove an $O(\log n \log\log n/\log\log\log n)$ bound on the amortized time per heap operation, using a more complicated argument. For the front-to-back variant, we are unable to establish any useful bound, since our analogy with splaying breaks down in this case. We leave as an open problem to prove or disprove Conjecture 1 for the pairing heap or any of its variants. Theorem 2 below derives our bound for the amortized time of the multipass variant. The analysis of the lazy variant is similar but more complicated.

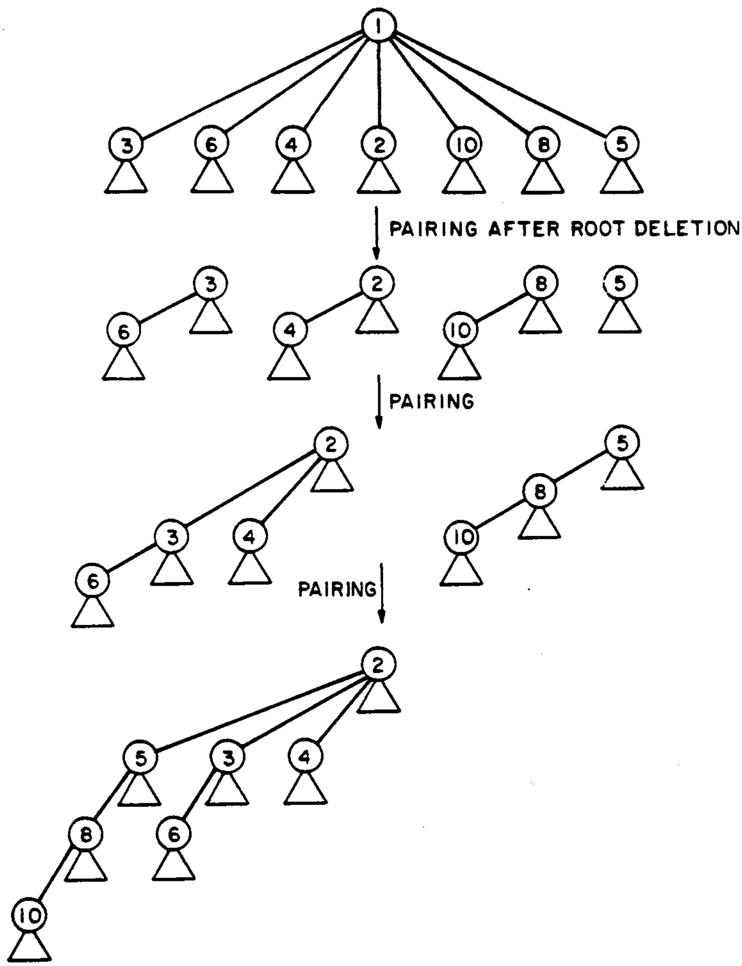

Fig. 15. A *delete min* operation using the multipass method.

THEOREM 2. The amortized time per heap operation for the multipass variant is $O(\log n \log\log / \log\log\log n)$.

To prove Theorem 2 we use a slight variant of the potential function used to prove Theorem 1. Let $P(T)$ be the potential of a binary tree defined as in the proof of Theorem 1. We shall use instead the potential function $Q(T) = P(T)/\log\log\log n$, where n is the number of nodes in T. The most difficult operation to analyze is *delete min*, and we proceed with this analysis. (The analysis of the other operations follows the proof of Theorem 1, and is omitted.)

Let T be a binary tree representing a heap and let T' be the tree that results by carrying out a *delete min* operation. Let k be the number of nodes on the right

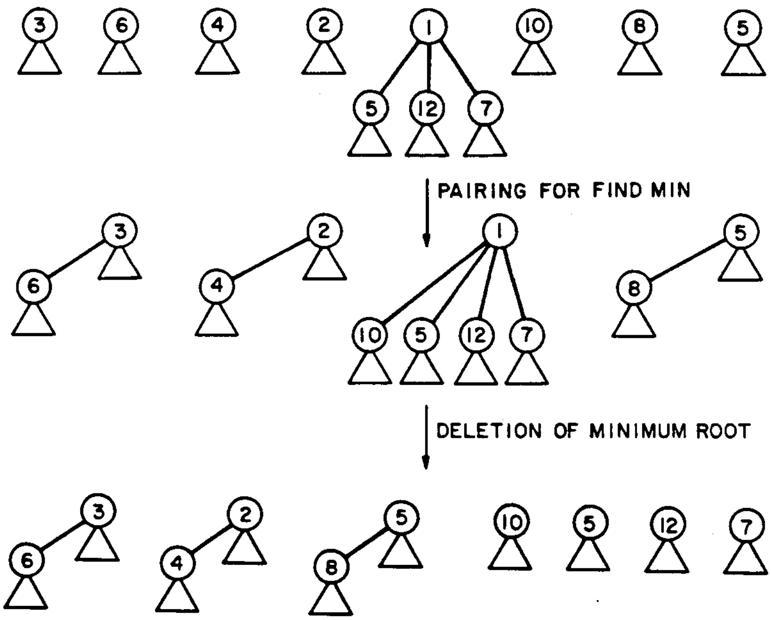

Fig. 16. A *delete min* operation using the lazy method.

path* of T after the root has been deleted. The time necessary for the *delete min* is $O(k + 1)$, since there are $k - 1$ link operations in total. We shall estimate the actual time taken by the *delete min* to be $\varepsilon(k + 1)$, where ε is a sufficiently small positive constant whose value we choose below. (That is, we assume that in one unit of time we can do a sufficiently large constant amount of work on the data structure). The amortized time of the *delete min* is thus $\varepsilon(k + 1) + P(T')/\log\log\log(n - 1) - P(T)/\log\log\log n$. Since $P(T') = O(n \log n)$, we have $P(T')/\log\log\log(n - 1) - P(T')/\log\log\log n = O(1)$. Thus the amortized time of the *delete min* is $\varepsilon k + (P(T') - P(T))/\log\log\log n + O(1)$.

Our main task is to estimate $P(T') - P(T)$. Let $n_1, n_2, \ldots, n_k$ be the nodes of the right path along which pairing takes place, with n_k being farthest from the root. Let s_i be the size of the subtree rooted at n_i. The change in the potential P resulting from linking n_i and n_{i+1} is at most $\log(s_i - s_{i+2}) - \log s_{i+1}$, where we let $s_{i+2} = 0$ if $i + 2 > k$. Referring to this potential change as t_i, we have $t_i \leq \log s_{i-1} - \log s_{i+1}$ if we let $s_0 = \log n$. We conclude that the sum of any subset of the t_i in any one pass is bounded by $\log n$, since the sum

* By the *right path* of a binary tree we mean the path from the root through right children to a node with no right child.

$\Sigma_{1 \le i < k, i \; odd}(\log s_{i-1} - \log s_{i+1})$ telescopes and all its terms are positive. Since there are $\lceil \log k \rceil$ pairing passes altogether, $P(T') - P(T) \le \lceil \log k \rceil (\log n)$.

This somewhat weak bound is enough to give a good estimate of the amortized *delete min* time if k is sufficiently small. Suppose

$$k \le c(\log n)(\log\log n)/(\log\log\log n),$$

where c is a sufficiently large positive constant, to be chosen below. Then the actual time of the *delete min* is $O(k) = O(\log n)(\log\log n)/(\log\log\log n)$, and the potential increase is at most $\lceil \log k \rceil (\log n)/\log\log\log n + O(1) = O(\log n)(\log\log n)/\log\log\log n$, so the amortized time of the *delete min* is $O((\log n)/(\log\log n)/\log\log\log n)$.

To obtain the same bound for the case of large k, i.e. $k > c(\log n)(\log\log n)/\log\log\log n$, we must estimate $P(T') - P(T)$ more carefully. In particular, we shall show that in this case the contribution of the negative t_i in just the first pass is enough to cause a negative potential change that makes the amortized time of the *delete min* $O(1)$. Since

$$\sum_{\substack{1 \le i < k-1 \\ i \; odd}} \log(s_i/s_{i+2}) \le \log n, \tag{4}$$

at least $k/4$ of the terms in (4) are bounded above by $(4 \log n)/k$. Since $k > c \log n$ it follows for each of these $k/4$ values of i (using the approximation $2^x = 1 + O(x)$ for bounded x) that

$$\frac{s_i - s_{i+2}}{s_{i+1}} < \frac{s_i - s_{i+2}}{s_{i+2}} = O\left(\frac{\log n}{k}\right) \tag{5}$$

From (5) we conclude that there are at least $k/4$ values of i for which $t_i \le -\log(k/(c' \log n))$ for some positive constant c'. Combining this with our previous estimate for the other t_i, we obtain $P(T') - P(T) \le \lceil \log k \rceil (\log n) - k/4 \log(k/(c' \log n))$.

Now if c is chosen sufficiently large, we obtain from the above estimate and $k > c(\log n)(\log\log n)/\log\log\log n$ that $P(T') - P(T) \le -c''k \log\log\log n + O(1)$, where c'' is a positive constant depending on c and c'. Thus the amortized time of the *delete min* is $\varepsilon k - c''k + O(1)$. Choosing $\varepsilon = c''$ gives an $O(1)$ bound.

We conclude that whether k is small or large the amortized time of *delete min* is $O((\log n)(\log\log n)/\log\log\log n)$, as desired.

Jones [7] has compared the experimental running times of pairing heaps and several other kinds of heaps. His experiments indicate that pairing heaps are competitive in practice with all known alternatives. Further experiments need to be done to determine the best implementation of the structure in practice.

References

[1] M. R. Brown, "Implementation and analysis of binomial queue algorithms," *SIAM J. Comput.* 7 (1978), 298–319.

[2] C. A. Crane, "Linear lists and priority queues as balanced binary trees," Technical Report STAN-CS-72-259, Computer Science Department, Stanford University, Stanford, CA, 1972.

[3] R. W. Floyd, "Algorithm 245: Treesort 3," *Comm. ACM* 7 (1964), 701.

[4] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *J. Assoc. Comput. Mach.*, submitted; also *Proc. 25th Annual IEEE Symp. on Found. of Comput. Sci.* (1984), 338–346.

[5] H. N. Gabow, Z. Galil, and T. Spencer, "Efficient implementation of graph algorithms using contraction," *Proc. 25th Annual IEEE Symp. on Found. of Comput. Sci.* (1984),

[6] H. N. Gabow, Z. Galil, T. Spencer, and R. E. Tarjan, "Efficient algorithms for finding minimum spanning trees in undirected and directed graphs," *Combinatorica*, to appear.

[7] D. W. Jones, "An empirical comparison of priority queues and event set algorithms," *Comm. ACM*, submitted.

[8] D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, Second Edition, Addison-Wesley, Reading, MA, 1973.

[9] D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

[10] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary trees, Proc. 15th Annual *ACM Symp. on Theory of Comput.* (1983), 235–245.

[11] D. D. Sleator and R. E. Tarjan, "Self-adjusting heaps," *SIAM J. Comput.*, to appear.

[12] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *J. Assoc. Comput. Mach.* 32 (1985), 652–686.

[13] R. E. Tarjan, *Data Structures and Network Algorithms*, CBMS 44, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[14] R. E. Tarjan, "Amortized computational complexity," *SIAM J. Alg. Disc. Meth.* 6 (1985), 306–318.

[15] J. Vuillemin, "A data structure for manipulating priority queues," *Comm. ACM* 21 (1978), 309–314.

[16] J. W. J. Williams, "Algorithm 232: Heapsort," *Comm. ACM* 7 (1964), 347–348.