SIAM J. COMPUT.
Vol. 15, No. 1, February 1986

004

SELF-ADJUSTING HEAPS*

DANIEL DOMINIC SLEATOR† AND ROBERT ENDRE TARJAN†

Abstract. In this paper we explore two themes in data structure design: *amortized computational complexity* and *self-adjustment.* We are motivated by the following observations. In most applications of data structures, we wish to perform not just a single operation but a sequence of operations, possibly having correlated behavior. By averaging the running time per operation over a worst-case sequence of operations, we can sometimes obtain an overall time bound much smaller than the worst-case time per operation multiplied by the number of operations. We call this kind of averaging *amortization.*

Standard kinds of data structures, such as the many varieties of balanced trees, are specifically designed so that the worst-case time per operation is small. Such efficiency is achieved by imposing an explicit structural constraint that must be maintained during updates, at a cost of both running time and storage space. However, if amortized running time is the complexity measure of interest, we can guarantee efficiency without maintaining a structural constraint. Instead, during each access or update operation we adjust the data structure in a simple, uniform way. We call such a data structure *self-adjusting.*

In this paper we develop the *skew heap,* a self-adjusting form of heap related to the leftist heaps of Crane and Knuth. (What we mean by a heap has also been called a "priority queue" or a "mergeable heap".) Skew heaps use less space than leftist heaps and similar worst-case-efficient data structures and are competitive in running time, both in theory and in practice, with worst-case structures. They are also easier to implement. We derive an information-theoretic lower bound showing that skew heaps have minimum possible amortized running time, to within a constant factor, on any sequence of certain heap operations.

Key words. Self-organizing data structure, amortized complexity, heap, priority queue

1. Introduction.

Many kinds of data structures have been designed with the aim of making the worst-case running time per operation as small as possible. However, in typical applications of data structures, it is not a single operation that is performed but rather a sequence of operations, and the relevant complexity measure is not the time taken by one operation but the total time of a sequence. If we average the time per operation over a worst-case sequence, we may be able to obtain a time per operation much smaller than the worst-case time. We shall call this kind of averaging over time *amortization.* A classical example of amortized efficiency is the compressed tree data structure for disjoint set union [15], which has a worst-case time per operation of $O(\log n)$ but an amortized time of $O(\alpha(m, n))$ [13], where n is the number of elements in the sets, m is the number of operations, and α is an inverse of Ackerman's function, which grows very slowly.

Data structures efficient in the worst case typically obtain their efficiency from an explicit structural constraint, such as the balance condition found in each of the many kinds of balanced trees. Maintaining such a structural constraint consumes both running time and storage space, and tends to produce complicated updating algorithms with many cases. Implementing such data structures can be tedious.

If we are content with a data structure that is efficient in only an amortized sense, there is another way to obtain efficiency. Instead of imposing any explicit structural constraint, we allow the data structure to be in an arbitrary state, but we design the access and update algorithms to adjust the structure in a simple, uniform way, so that the efficiency of future operations is improved. We call such a data structure *self-adjusting.*

* Received by the editors October 12, 1983, and in revised form September 15, 1984.
† AT&T Bell Laboratories, Murray Hill, New Jersey 07974.

Self-adjusting data structures have the following possible advantages over explicitly balanced structures:

(i) They need less space, since no balance information is kept.

(ii) Their access and update algorithms are easy to understand and to implement.

(iii) In an amortized sense, ignoring constant factors, they can be at least as efficient as balanced structures.

Self adjusting structures have two possible disadvantages:

(i) More local adjustments take place than in the corresponding balanced structures, especially during accesses. (In a balanced structure, adjustments usually take place only during updates, not during accesses.) This can cause inefficiency if local adjustments are expensive.

(ii) Individual operations within a sequence can be very expensive. Although expensive operations are likely to be rare, this can be a drawback in real-time applications.

In this paper we develop and analyze the *skew heap*, a self-adjusting form of heap (priority queue) analogous to leftist heaps [4], [7]. The fundamental operation on skew heaps is *melding*, which combines two disjoint heaps into one. In § 2 we present the basic form of skew heaps, which use top-down melding. In § 3 we discuss skew heaps with bottom-up melding, which are more efficient for insertion and melding. In § 4 we study various less common heap operations. In § 5 we show that in an amortized sense skew heaps are optimal to within a constant factor on any sequence of certain operations. Section 6 compares skew heaps to other heap implementations, and contains additional remarks and open problems. The appendix contains our tree terminology.

This paper represents only a part of our work on amortized complexity and self-adjusting data structures; companion papers discuss self-adjusting lists [12] and self-adjusting search trees [13]. Some of our results have previously appeared in preliminary form [11].

2. Skew heaps. A *heap* (sometimes called a *priority queue* [8] or *mergeable heap* [1]) is an abstract data structure consisting of a set of items selected from a totally ordered universe, on which the following operations are possible:

function *make heap*(h): Create a new, empty heap, named h.

function *find min*(h): Return the minimum item in heap h. If h is empty, return the special item "null".

procedure *insert*(x, h): Insert item x in heap h, not previously containing it.

function *delete min*(h): Delete the minimum item from heap h and return it. If the heap is initially empty, return null.

function *meld*(h_1, h_2): Return the heap formed by taking the union of disjoint heaps h_1 and h_2. This operation destroys h_1 and h_2.

There are several ways to implement heaps in a self-adjusting fashion. The one we shall discuss is an analogue of the leftist heaps proposed by Crane [4] and refined by Knuth [8]. To represent a heap, we use a *heap-ordered binary tree*, by which we mean a binary tree whose nodes are the items, arranged in heap order: if $p(x)$ is the parent of x, then $p(x) < x$. (For simplicity we shall assume that each item is in at most one heap; this restriction is easily lifted by regarding the tree nodes and heap items as distinct, with a pointer in each tree node pointing to the corresponding heap item.) To represent such a tree we store with each item x two pointers, *left*(x) and *right*(x), to its left child and right child respectively. If x has no left child we define *left*(x) = null; if x has no right child we define *right*(x) = null. Access to the tree is by a pointer to

its root; we represent an empty tree by a pointer to null. We shall sometimes denote an entire tree or subtree by its root, with the context resolving the resulting ambiguity.

With a representation of heaps as heap-ordered binary trees, we can carry out the various heap operations as follows. We perform *make heap*(h) in $O(1)$ time by initializing h to null. Since heap order implies that the root is the minimum item in a tree, we can carry out *find min* in $O(1)$ time by returning the root. We perform *insert* and *delete min* using *meld*. To carry out *insert*(x, h), we make x into a one-node heap and meld it with h. To carry out *delete min*(h), we replace h by the meld of its left and right subtrees and return the original root.

To perform *meld*(h_1, h_2), we form a single tree by traversing the right paths of h_1 and h_2, merging them into a single right path with items in increasing order. The left subtrees of nodes along the merge path do not change. (See Fig. 1.) The time for the meld is bounded by a constant times the length of the merge path. To make melding efficient, we must keep right paths short. In leftist heaps this is done by maintaining the invariant that, for any node x, the right path descending from x is a shortest path down to a null node. Maintaining this invariant requires storing at every node the length of a shortest path down to a missing node; after a meld we walk back up the merge path, updating shortest path lengths and swapping left and right children as necessary to maintain the leftist property. The length of the right path in a leftist tree of n nodes is at most $\lfloor \log n \rfloor$[1], implying an $O(\log n)$ worst-case time bound for each of the heap operations, where n is the number of nodes in the heap or heaps involved.

In our self-adjusting version of this data structure, we meld by merging the right paths of the two trees and then swapping the left and right children of *every* node on the merge path except the lowest. (See Fig. 1.) This makes the potentially long right path formed by the merge into a left path. We call the resulting data structure a *skew heap*.

In our analysis of skew heaps, we shall use the following general approach. We associate with each possible collection S of skew heaps a real number $\Phi(S)$ called the *potential* of S. For any sequence of m operations with running times $t_1, t_2, \cdots, t_m$, we define the *amortized time* a_i of operation i to be $a_i = t_i + \Phi_i - \Phi_{i-1}$, where Φ_i, for $i = 1, 2, \cdots, m$, is the potential of the skew heaps after operation i and Φ_0 is the potential of the skew heaps before the first operation. The total running time of the sequence of operations is then

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

That is, the total running time equals the total amortized time plus the decrease in potential from the initial to the final collection of heaps. In most of our analyses the potential will be zero initially and will remain nonnegative. If this is the case then the total amortized time is an upper bound on the actual running time.

This definition is purely formal; its utility depends on the ability to choose a potential function that results in small amortized times for the operations. Whenever we use this technique we shall define the potential of a single heap; the potential of a collection is the sum of the potentials of its members. Intuitively, a heap with high potential is one subject to unusually time-consuming operations; the extra time spent corresponds to a drop in potential.

We shall prove an $O(\log n)$ bound on the amortized time of skew heap operations. To do this we define the potential function using an idea of Sleator and Tarjan [9],

[1] Throughout this paper we use base-two logarithms.

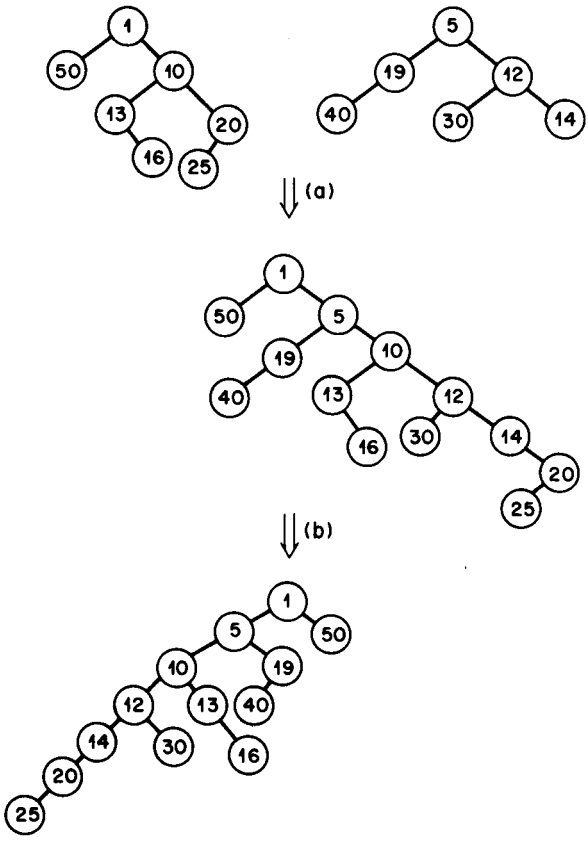

FIG. 1. *A meld of two skew heaps.* (a) *Merge of the right paths.* (b) *Swapping of children along the path formed by the merge.*

[10]. For any node x in a binary tree, we define the *weight* $wt(x)$ of x as the number of descendants of x, including x itself. We use the weights to partition the nonroot nodes into two classes: a nonroot node x is *heavy* if $wt(x) > wt(p(x))/2$ and *light* otherwise. We shall regard a root as being neither heavy nor light. The following lemmas are immediate from this definition.

LEMMA 1. *Of the children of any node, at most one is heavy.*

Proof. A heavy child has more than half the weight of its parent. This can be true of only one child. □

LEMMA 2. *On any path from a node x down to a descendent y, there are at most $\lfloor \log (wt(x)/wt(y)) \rfloor$ light nodes, not counting x. In particular, any path in an n-node tree contains at most $\lfloor \log n \rfloor$ light nodes.*

Proof. A light child has at most half the weight of its parent. Thus if there are k light nodes not including x along the path from x to y, $wt(y) \leqq wt(x)/2^k$, which implies $k \leqq \log (wt(x)/wt(y))$. □

We define the potential of a skew heap to be the total number of right heavy nodes it contains. (A nonroot node is *right* if it is a right child and *left* otherwise.) The intuition justifying this choice of potential is as follows. By Lemma 2, any path in a skew heap, and in particular any path traversed during melding, contains only $O(\log n)$ light nodes. Any heavy node on such a path is converted from right to left by the meld, causing a drop of one in the potential. As we shall prove rigorously below,

this implies that any melding time in excess of $O(\log n)$ is covered by a drop in the potential, giving an amortized melding time of $O(\log n)$.

Suppose we begin with no heaps and carry out an arbitrary sequence of skew heap operations. The initial potential is zero and the final potential is nonnegative, so the total running time is bounded by the sum of the amortized times of the operations. Furthermore, since the potential of a skew heap of n items is at most $n-1$, if we begin with any collection of skew heaps and carry out an arbitrary sequence of operations, the total time is bounded by the total amortized time plus $O(n)$, where n is the total size of the initial heaps.

The amortized time of a *find min* operation is $O(1)$, since the potential does not change. The amortized times of the other operations depend on the amortized time for *meld*. Consider a meld of two heaps h_1 and h_2, containing n_1 and n_2 items, respectively. Let $n = n_1 + n_2$ be the total number of items in the two heaps. As a measure of the melding time we shall charge one per node on the merge path. Thus the amortized time of the meld is the number of nodes on the merge path plus the change in potential. By Lemma 1, the number of light nodes on the right paths of h_1 and h_2 is at most $\lfloor \log n_1 \rfloor$ and $\lfloor \log n_2 \rfloor$, respectively. Thus the total number of light nodes on the two paths is at most $2\lfloor \log n \rfloor - 1$. (See Fig. 2.)

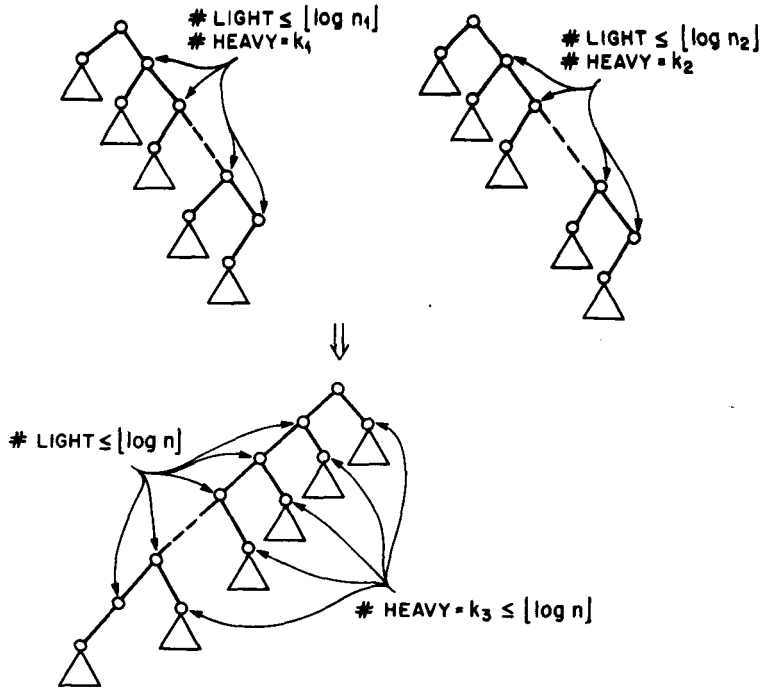

FIG. 2. *Analysis of right heavy nodes in meld.*

Let k_1 and k_2 be the number of heavy nodes on the right paths of h_1 and h_2, respectively, and let k_3 be the number of nodes that become right heavy children of nodes on the merge path. Every node counted by k_3 corresponds to a light node on the merge path. Thus Lemma 2 implies that $k_3 \leqq \lfloor \log n \rfloor$.

The number of nodes on the merge path is at most $2+\lfloor\log n_1\rfloor+k_1+\lfloor\log n_2\rfloor+k_2 \leqq 1+2\lfloor\log n\rfloor+k_1+k_2$. (The "2" counts the roots of h_1 and h_2.) The increase in potential caused by the meld is $k_3-k_1-k_2 \leqq \lfloor\log n\rfloor-k_1-k_2$. Thus the amortized time of the meld is at most $3\lfloor\log n\rfloor+1$.

THEOREM 1. *The amortized time of an insert, delete min, or meld skew heap operation is $O(\log n)$, where n is the number of items in the heap or heaps involved in the operation. The amortized time of a make heap or find min operation is $O(1)$.*

Proof. The analysis above gives the bound for *find min* and *meld*; the bound for *insert* and *delete min* follows immediately from that of *meld*. □

One may ask whether amortization is really necessary in the analysis of skew heaps, or whether skew heaps are efficient in a worst-case sense. Indeed they are not: we can construct sequences of operations in which some operations take $O(n)$ time. For example, suppose we insert n, $n+1$, $n-1$, $n+2$, $n-2$, $n+3, \cdots, 1$, $2n$ into an initially empty heap and then perform *delete min*. The tree resulting from the insertions has a right path of n nodes, and the *delete min* takes $\Omega(n)$ time. (See Fig. 3.) There are similar examples for the other versions of skew heaps we shall consider.

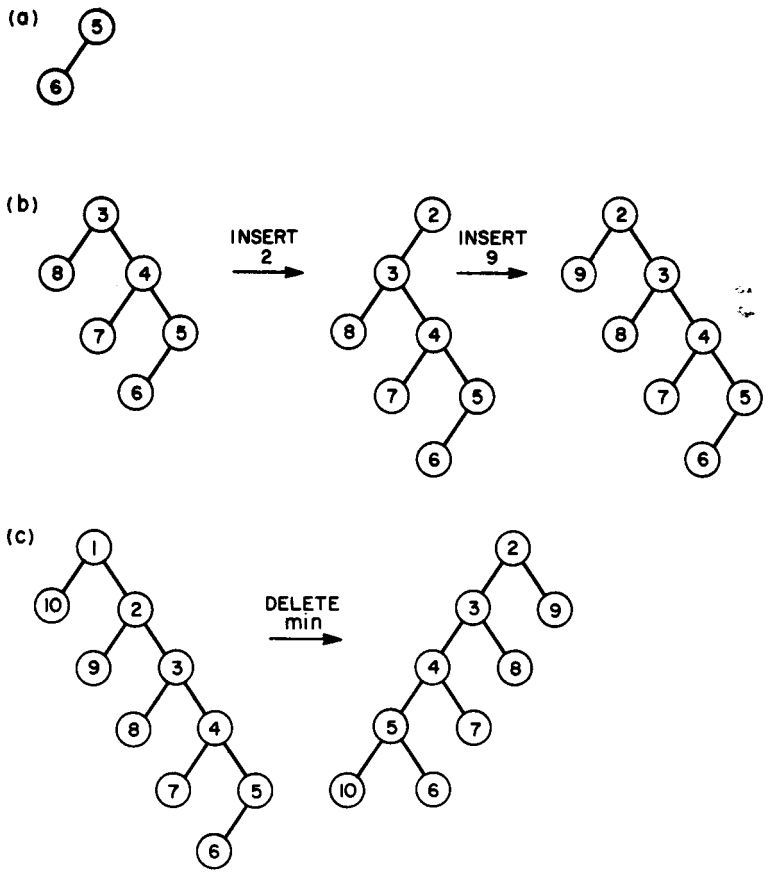

FIG. 3. *Insertion of* 5, 6, 4, 7, 3, 8, 2, 9, 1, 10 *into an initially empty heap, followed by delete min.* (a) *The heap after two insertions.* (b) *Insertion of* 2 *and* 9. (c) *The delete min operation.*

The following programs, written in an extension of Dijkstra's guarded command language [5], implement the various skew heap operations. A **val** parameter to a function or procedure is called by value; a **var** parameter is called by value and result. The double arrow "↔" denotes swapping. Parallel assignments all take place simultaneously.

```
function make heap;
  return null
end make heap;

function find min(val h);
  return h
end find min;

procedure insert(val x, var h);
  left(x) := right(x) := null;
  h := meld(x, h)
end insert;

function delete min(var h);
  var x;
  x := h; h := meld(left(h), right(h)); return x
end delete min;
```

Our implementation of *meld* differs slightly from the informal description. The program traverses the two right paths from the top down, merging them and simultaneously swapping left and right children. When the bottom of one of the paths is reached, the remainder of the other path is simply attached to the bottom of the merge path, and the process terminates. Only the nodes visited have their children exchanged; the last node whose children are exchanged is the lowest node on the one of the two paths that is completely traversed. (See Fig. 4.) In the informal description, all nodes on both right paths are visited. Theorem 1 holds for the actual implementation; the same proof applies if k_1 and k_2 are redefined to be the number of heavy nodes on the right paths of h_1 and h_2 actually traversed during the meld.

We shall give two versions of *meld*: a recursive version, *rmeld*, and an iterative version, *imeld*. The recursive version uses an auxiliary function *xmeld* to do the actual melding, in order to avoid redundant tests for null.

```
function rmeld(val h1, h2);
  return if h2 = null → h1 [] h2 ≠ null → xmeld(h1, h2) fi
end rmeld;

function xmeld(val h1, h2);
  [h2 ≠ null]
  if h1 = null → return h2 fi;
  if h1 > h2 → h1 ↔ h2 fi;
  left(h1), right(h1) := xmeld(right(h1), h2), left(h1);
  return h1
end xmeld;
```

The iterative version of *meld* uses four variables, x, y, h_1, and h_2, and maintains the following loop invariant: if $left(y)$ is replaced by null, then x, h_1, and h_2 are the roots of three disjoint heaps containing all the nodes; y is the bottommost node of the left path down from x (the merge path) and is such that $y < \min\{h_1, h_2\}$.

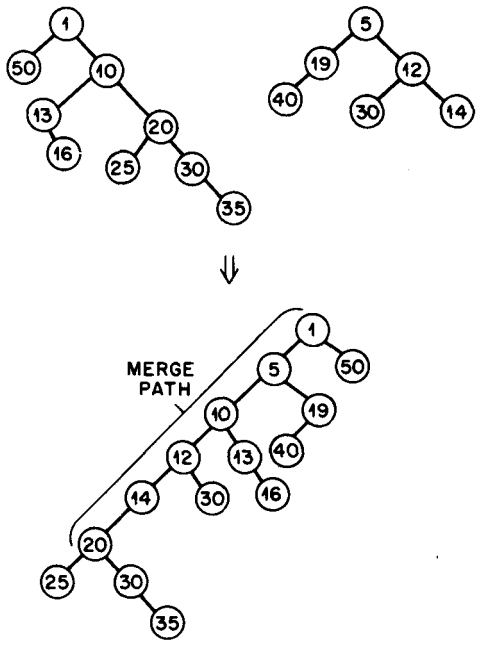

FIG. 4. *Implemented version of top-down melding.*

```
function imeld(val h1, h2);
  var x, y;
  if h1 = null → return h2 ▯ h2 = null → return h1 fi;
  if h1 > h2 → h1 ↔ h2 fi;
  x, y, h1, right(h1) := h1, h1, right(h1), left(h1);
  do h1 ≠ null →
    if h1 > h2 → h1 ↔ h2 fi;
    y, left(y), h1, right(h1) := h1, h1, right(h1), left(h1)
  od;
  left(y) := h2;
  return x
end imeld;
```

Note. The swapping of h_1 and h_2 in the loop can be avoided by writing different pieces of code for the cases $h_1 > h_2$ and $h_1 \leqq h_2$. The four-way parallel assignment can be written as the following four sequential assignments: $left(y) := h_1$; $y := h_1$; $h_1 := right(y)$; $right(y) := left(y)$. The assignment "$y := h_1$" can be deleted by unrolling the loop. With these changes a meld takes $O(1)$ time plus three assignments and two comparisons per node on the merge path.

One possible drawback of skew heaps is the number of pointer assignments needed. We can reduce the pointer updating by storing in each node a bit indicating that the pointers to the left and right children have been reversed. Children can then be swapped merely by changing a bit. This idea trades bit assignments for pointer assignments but takes extra space and complicates the implementation.

3. Bottom-up skew heaps. In some applications of heaps, such as in the computation of minimum spanning trees [3], [16], it is important that melding be as efficient as possible. By melding skew heaps bottom-up instead of top-down, we can reduce

the amortized time of *insert* and *meld* to $O(1)$ without affecting the time bounds of the other operations. If h_1 and h_2 are the heaps to be melded, we walk up the right paths of h_1 and h_2, merging them and exchanging the children of all nodes on the merge path except the lowest. When reaching the top of one of the heaps, say h_1, we attach the root of h_1 (the top node on the merge path) as the right child of the lowest node remaining on the right path of h_2. The root of h_1 is the last node to have its children swapped, unless h_1 is the *only* node on the merge path, in which case no swapping takes place. (See Fig. 5.)

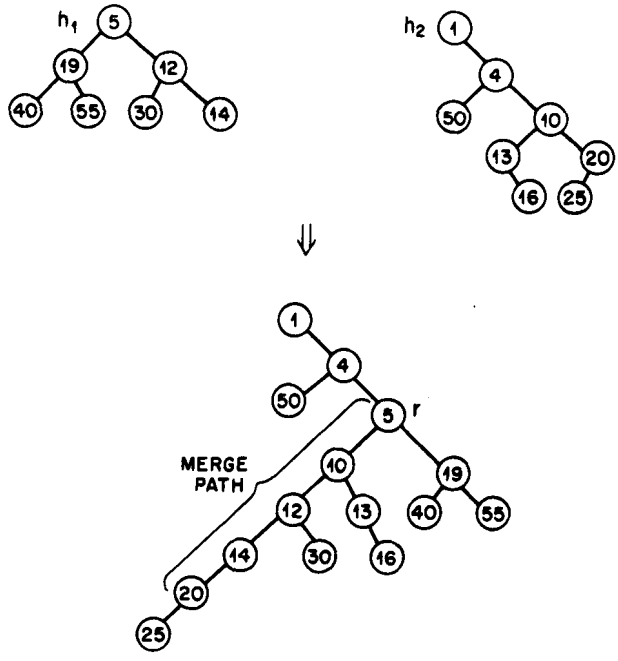

FIG. 5. *Bottom-up melding.*

We can implement this method by storing with each node x an extra pointer $up(x)$, defined to be the parent of x if x is a right child, or the lowest node on the right path descending from x if x is a left child or a root. Thus right paths are circularly linked, bottom-up. (See Fig. 6.) We call this the *ring representation.* We shall consider alternative representations at the end of the section.

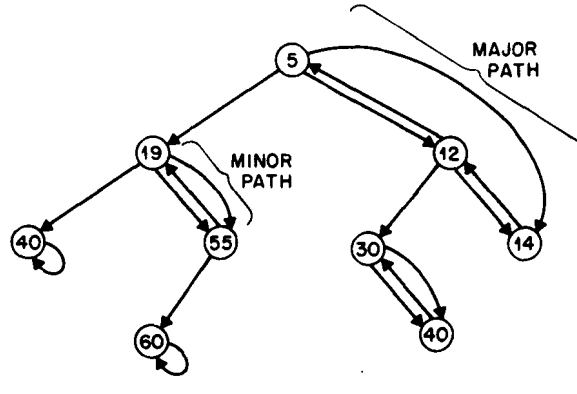

FIG. 6. *Ring representation of a skew heap.*

Obtaining good amortized time bounds for the various operations on bottom-up skew heaps requires a more complicated potential function than the one used in § 2. To define the potential, we need two subsidiary definitions. If T is any binary tree, we define the *major path* of T to be the right path descending from the root, and the *minor path* to be the right path descending from the left child of the root. (See Fig. 6.) We define node weights and light and heavy nodes as in § 2. We define the potential of a bottom-up skew heap to be the number of right heavy nodes in the tree plus twice the number of right light nodes on the major and minor paths.

Consider a bottom-up meld of two skew heaps h_1 and h_2. We shall show that the amortized time of the meld is $O(1)$, if we count one unit of running time per node on the merge path. Let h_3 be the melded heap, and suppose, without loss of generality, that h_1 is the heap whose top is reached during the meld. Let r be the root of h_1. (See Fig. 5.) The merge path is the top part of the left path descending from r in h_3. It contains all nodes on the major path of h_1 and possibly some of the nodes on the major path of h_2. The major path of h_3 consists of the nodes on the major path of h_2 not on the merge path, node r, and, if the merge path contains two or more nodes, the minor path of h_1. The minor path of h_3 is the minor path of h_2. The only nodes whose weights change during the merge are those on the major paths of h_1 and h_2; the weights of these nodes can increase but not decrease.

Consider the change in potential caused by the meld. Any node on the major path of h_2 not on the merge path can gain in weight, becoming a right heavy instead of a right light node. Each such change decreases the potential by one. No such node can change from heavy to light, because the weights of both it and its parent increase by the same amount. Node r, the root of h_1, becomes a node on the major path of h_3, increasing the potential by one if r becomes heavy or two if r becomes light. The top node on the minor path of h_1 also can become a node on the major path of h_3, increasing the potential by at most two. The remaining nodes on the minor path of h_1 and the nodes on the minor path of h_2 are associated with no change in the potential.

It remains for us to consider the nodes other than r on the merge path. Let x be such a node. If x is originally heavy, it remains heavy but becomes left, causing a decrease of one in potential (the new right sibling of x is light). If x is originally light, its new right sibling may be heavy, but there is still a decrease of at least one in potential (from two units for x as a light node on a major path to at most one unit for the new right sibling of x, which is not on a major or minor path).

Combining these estimates, we see that the amortized meld time, a, defined to be the number of nodes on the merge path, t, plus the change in potential, $\Delta\Phi$, satisfies $a = t + \Delta\Phi \leqq 5$: the potential decreases by at least one for each node on the merge path except r, and increases by at most two for r and two for the top node on the minor path of h_1.

THEOREM 2. *The amortized time of a make heap, find min, insert, or meld operation on bottom-up skew heaps is* $O(1)$. *The amortized time for a delete min operation on an* n*-node heap is* $O(\log n)$.

Proof. Both the worst-case and amortized times of *make heap* and *find min* are $O(1)$, since neither causes a change in potential. The $O(1)$ amortized time bound for *insert* and *meld* follows from the analysis above. Consider a *delete min* operation on an n-node skew heap h. Deleting the root of h takes $O(1)$ time and produces two skew heaps h_1 and h_2, whose roots are the left and right children of the root of h, respectively. This increases the potential by at most $4 \log n$: there is an increase of two units for each light right node on the minor paths of h_1 and h_2. (The major path of h, minus the root, becomes the minor path of h_2; the minor path of h becomes the major path

of h_1.) The meld that completes the deletion takes $O(1)$ amortized time, giving a total amortized deletion time of $O(\log n)$. □

Since any n-node bottom-up skew heap has a potential of $O(n)$, if we begin with any collection of such heaps and carry out any sequence of operations, the total time is bounded by the total amortized time plus $O(n)$, where n is the total size of the initial heaps. If we begin with no heaps, the total amortized time bounds the total running time.

The ring representation of bottom-up skew heaps is not entirely satisfactory, since it needs three pointers per node instead of two. The extra pointer is costly in both storage space and running time, because it must be updated each time children are swapped. There are several ways to reduce the number of pointers to two per node. We shall discuss two in this section and a third in § 4.

One possible change is to streamline the ring representation by not storing right child pointers. An even more appealing possibility is to store with each node an *up* pointer to its parent and a *down* pointer to the lowest node on the right path of its left subtree. The *up* pointer of the root, which has no parent, points to the lowest node on the major path. The *down* pointer of a node with empty left subtree points to the node itself. We shall call this the *bottom-up representation* of a skew heap.

Both of these representations will support all the heap operations in the time bounds of Theorem 2. We shall discuss the bottom-up representation; we favor it because swapping children, which is the local update operation on skew heaps, is especially easy.

The implementations of *make heap* and *find min* are exactly as presented in § 2. We shall give two implementations of *insert*. The first merely invokes *meld*. The second includes an in-line customized version of *meld* for greater efficiency.

```
procedure insert (val x, var h);
  up(x) := down(x) := x;
  h := meld(x, h)
end insert;
```

The more efficient version of *insert* tests for three special cases: $h = \text{null}$, $x < h$ (x becomes the root of the new tree), and $x > up(h)$ (x becomes the new lowest node on the major path). In the general case, x is inserted somewhere in the middle of the major path by a loop that uses two local variables, y, and z; y is the highest node on the major path so far known to satisfy $y > x$, and z is the lowest node on the major path (after the swapping of children that has taken place so far).

```
procedure insert (val x; var h);
  var y, z;
  if h = null → h := up(x) := down(x) := x; return fi;
  if x < h → down(x) := up(h); h := up(x) := up(h) := x; return fi;
  if x > up(h) → up(x) := up(h); down(x) := up(h) := x; return fi;
  y := z := up(h);
  do x < up(y) → y := up(y); z ↔ down(y) od;
  up(x), down(x) := up(y), z;
  up(y) := up(h) := x
end insert;
```

The following program implements *meld*. The program uses two variables, h_3 and x, to hold the output heap and the next item to be added to the output heap, respectively.

```
function meld(val h1, h2);
  var h3, x;
  if h1 = null → return h2 ▯ h2 = null → return h1 fi;
  if up(h1) < up(h2) → h1 ↔ h2 fi;
  [initialize h3 to hold the bottom right node of h1]
  h3 := up(h1); up(h1) := up(h3); up(h3) := h3;
  do h1 ≠ h3 →
    if up(h1) < up(h2) → h1 ↔ h2 fi;
    [remove from h1 its bottom right node, x]
    x := up(h1); up(h1) := up(x);
    [add x to the top of h3 and swap its children]
    up(x) := down(x); down(x) := up(h3); h3 := up(h3) := x
  od;
  [attach h3 to the bottom right of h2]
  up(h2) ↔ up(h3);
  return h2
end meld;
```

The only cleverness in this code is in the termination condition of the loop. Just before the last node is removed from heap h_1, $up(h_1) = h_1$, which means that at the beginning of the next iteration $h_1 = h_3$, and the loop will terminate. The code contains an annoyingly large number of assignments. Some of these can be removed, and the efficiency of the program improved, by storing $up(h_1)$, $up(h_2)$, and $up(h_3)$ in local variables and unrolling the **do** loop to store state information in the flow of control. This obscures the algorithm, however.

Implementing *delete min* poses a problem: there is no way to directly access the children of the root (the node to be deleted) to update their *up* pointers. We can overcome this problem by performing the deletion as follows: we merge the minor and major paths of the tree, swapping children all along the merge path. We stop the merge *only* when the root is reached along *both* paths. The following program implements this method. The program uses four local variables: h_3 is the output heap, y_1 and y_2 are the current nodes on the major and minor paths, and x is the next node to be added to the output heap. The correctness of the program depends on the assumption that an item appears only once in a heap. As in the case of *meld*, we can improve the efficiency somewhat by storing $up(h_3)$ in a local variable and unrolling the loop.

```
function delete min(var h);
  var x, y1, y2, h3;
  if h = null → return null fi;
  y1, y2 := up(h), down(h);
  if y1 < y2 → y1 ↔ y2 fi;
  if y1 = h → h = null; return y1 fi;
  [initialize h3 to hold y1]
  h3 := y1; y1 := up(y1); up(h3) := h3;
  do true →
    if y1 < y2 ↔ y2 fi;
    if y1 = h → h := h3; return y1 fi;
    [remove x = y1 from its path]
    x := y1; y1 := up(y1);
    [add x to the top of h3 and swap its children]
    up(x) := down(x); down(x) := up(h3); h3 := up(h3) := x
  od
end delete min;
```

The same potential function and the same argument used at the beginning of this section prove Theorem 2 for the bottom-up representation of skew heaps; namely, *make heap* and *find min* take $O(1)$ time (both amortized and worst case), *insert* and *meld* take $O(1)$ amortized time, and *delete min* on an n-node heap takes $O(\log n)$ amortized time.

4. Other operations on skew heaps. There are a variety of additional heap operations that are sometimes useful. In this section we shall consider the following four:

function *make heap*(s): Return a new heap whose items are the elements in set s.
function *find all*(x, h): Return the set of all items in h less than or equal to x.
procedure *delete*(x, h): Delete item x from heap h.
procedure *purge*(h): Assuming that each item in heap h is marked "good" or "bad", delete enough bad items from h so that the minimum item left in the heap is good.

The operation *make heap*(s) is an extension of *make heap* as defined in § 2 and allows us to initialize a heap of arbitrary size. The operation *find all* is a form of range query. The *delete* operation allows deletion from a heap of any item, not just the minimum one. An alternative way to delete arbitrary items is with the *purge* operation, using *lazy deletion*: When inserting an item in a heap, we mark it "good". When deleting an item we do not alter the structure of the heap but merely mark the item "bad". Before any *find min* or *delete min* operation, we purge the heap. Lazy deletion is especially useful when items can be marked for deletion implicitly, as in the computation of minimum spanning trees using heaps [3], [16]. Deleting many bad items simultaneously reduces the time per deletion. The only drawback of lazy deletion is that a deleted item cannot be reinserted until it has been purged. We can overcome this by copying the item, at a cost of extra storage space.

We shall begin by implementing the four new operations on the top-down skew heaps of § 2. The operations *make heap*(s) and *purge* are best treated as special cases of the following more general operation:

function *heapify*(s): Return a heap formed by melding all the heaps in the set s. This operation assumes that the heaps in s are disjoint and destroys them in the process of melding them.

As discussed by Tarjan [16], we can carry out *heapify*(s) by repeated pairwise melding. We perform a number of passes through the set s. During each pass, we meld the heaps in s in pairs; if the number of heaps is odd, one of them is not melded until a subsequent pass. We repeat such passes until there is only one heap left, which we return.

This method is efficient for any heap representation that allows two heaps of total size n to be melded in $O(\log n)$ time. To analyze its running time, consider a single pass. Let k be the number of heaps in s before the pass and let n be their total size. After the pass, s contains only $\lceil k/2 \rceil \leqq 2k/3$ heaps. The time for the pass is $O(k + \sum_{i=1}^{\lceil k/2 \rceil} \log n_i)$, where n_i is the number of items in the ith heap remaining after the pass. The sizes n_i satisfy $1 \leqq n_i \leqq n$ and

$$\sum_{i=1}^{\lceil k/2 \rceil} n_i \leqq n.$$

The convexity of the log function implies that the time bound is maximum when all

the n_i are equal, which gives a time bound for the pass of $O(k+\log(n/k))$. Summing over all passes, the time for the entire *heapify* is

$$O\left(\sum_{i=0}^{\lfloor \log 3/2k \rfloor} \left(\frac{2}{3}\right)^i \left(k + k\log\left(\left(\frac{3}{2}\right)^i \frac{n}{k}\right)\right)\right) = O\left(k + k\log\left(\frac{n}{k}\right)\right),$$

where k is the original number of heaps and n is the total number of items they contain. For skew heaps, this is an amortized time bound. For worst-case data structures such as leftist or binomial heaps [2], [16], this is a worst-case bound.

We can perform *make heap*(s) by making each item into a one-item heap and applying *heapify* to the set of these heaps. Since $k = n$ in this case, the time for *make heap* is $O(n)$, both amortized and worst-case.

We can perform *purge*(h) by traversing the tree representing h in preorder, deleting every bad node encountered and saving every subtree rooted at a good node encountered. (When visiting a good node, we immediately retreat, without visiting its proper descendants.) We complete the purge by heapifying the set of subtrees rooted at visited good nodes. If k nodes are purged from a heap of size n, the time for the purge is $O(k+k\log(n/k))$, since there are at most $k+1$ subtrees to be heapified. For skew heaps, this is an amortized bound.

We can carry out *find all* on any kind of heap-ordered binary tree in time proportional to the size of the set returned. We traverse the tree in preorder starting from the root, listing every node visited that does not exceed x. When we encounter a node greater than x, we immediately retreat, without visiting its proper descendants. Note that since heap order is not a total order, *find all* cannot return the selected items in sorted order without performing additional comparisons.

If *find all* is used in combination with lazy deletion and it is not to return bad items, it must purge the tree as it traverses it. The idea is simple enough: When traversing the tree looking for items not exceeding x, we discard every bad item encountered. This breaks the tree into a number of subtrees, which we heapify. It is not hard to show that a *find all* with purging that returns j good items and purges k bad items from an n-item heap takes $O(j+k+k\log(n/k))$ amortized time. We leave the proof of this as an exercise.

The fourth new operation is deletion. We can delete an arbitrary item x from a top-down skew heap (or, indeed, from any kind of heap-ordered binary tree) by replacing the subtree rooted at x by the meld of its left and right subtrees. This requires maintaining parent pointers for all the nodes, since deleting x changes one of the children of $p(x)$. The amortized time to delete an item from an n-node heap is $O(\log n)$. In addition to the $O(\log n)$ time for what is essentially a *delete min* on the subtree rooted at x, there is a possible additional $O(\log n)$ gain in potential, since deleting x reduces the weights of proper ancestors of x, causing at most $\log n$ of them to become light: their siblings, which may be right, may become heavy.

By changing the tree representation we can carry out all the heap operations, including arbitrary deletion, top-down using only two pointers per node. Each node points to its leftmost child (or to null if it has no children) and to its right sibling, or to its parent if it has no right sibling. (The root, having neither a right sibling nor a parent, points to null.) Knuth calls this the "binary tree representation of a tree, with right threads"; we shall call it the *threaded representation*. (See Fig. 7.) With the threaded representation, there is no way to tell whether an only child is left or right, but for our purposes this is irrelevant; we can regard every only child as being left. From any node we can access its parent, left child, or right child by following at most two pointers, which suffices for implementing all the heap operations.

Now let us consider the four new operations on bottom-up skew heaps. Assume for the moment that we use the ring representation. (See Fig. 6.) The operation *heapify*(s) is easy to carry out efficiently: We merely meld the heaps in s in any order. This takes $O(k)$ amortized time if there are k heaps in s. In particular, we can initialize a heap of n items in $O(n)$ time (amortized and worst-case) by performing n successive insertions into an initially empty heap. We can purge k items from a heap of n items in $O(k+k(\log n/k))$ amortized time by traversing the tree in preorder and melding the subtrees rooted at visited good nodes in arbitrary order. The main contribution to the time bound is not the melds, which take $O(k)$ amortized time, but the increase in potential caused by breaking the tree into subtrees: a subtree of size n_i gains in potential by at most $4\log n_i$ because of the light right nodes on its major and minor paths. (See the definition of potential used in § 3.) The total potential increase is maximum when all the subtrees are of equal size and is $O(k+k\log(n/k))$.

The operation *find all* is exactly the same on a bottom-up skew heap as on a top-down skew heap, and takes time proportional to the number of items returned. A *find all* with purging on a bottom-up skew heap is similar to the same operation on a top-down skew heap, except that we can meld the subtrees remaining after bad items are deleted in any order. The amortized time bound is the same, namely $O(j+k+k\log(n/k))$ for a *find all* with purging that returns j good items and purges k bad items from an n-item heap.

Arbitrary deletion on bottom-up skew heaps requires changing the tree representation, since the ring representation provides no access path from a left child to its parent. We shall describe a representation that supports all the heap operations, including bottom-up melding, and needs only two pointers per node. The idea is to use the threaded representation proposed above for top-down heaps, but to add a pointer to the lowest node on the right path. (See Fig. 7.) We shall call this extra pointer the *down pointer.*

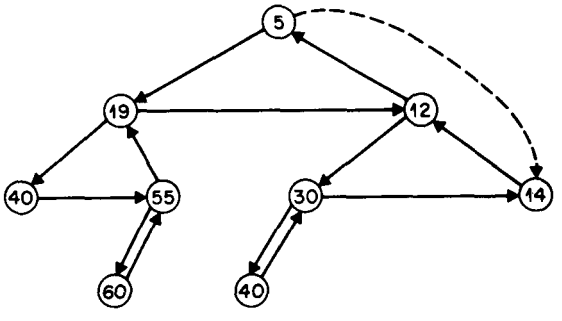

FIG. 7. *Threaded representation of a skew heap. All only children are regarded as being left. The dashed pointer is the down pointer, used only in the bottom-up version.*

When performing a heap operation, we reestablish the down pointer when necessary by running down the appropriate right path. For example, suppose we meld two heaps h_1 and h_2 bottom-up, and that heap h_1 is exhausted first, so that the root of h_1, say r, becomes the top node on the merge path. We establish the down pointer in the melded heap by descending the new right path from r; this is the minor path in the original heap h_1 unless the merge path contains only r, in which case r has a null right child. (See Fig. 5.)

To perform a *delete min* operation, we descend the minor path of the heap to establish a down pointer for the left subtree, and then we meld the left and right

subtrees; the down pointer for the right subtree is the same as the down pointer for the entire tree. We perform *purge* as described above for bottom-up skew heaps, establishing a down pointer for every subtree to be melded by traversing its right path.

Arbitrary deletion is the hardest operation to implement, because if the deleted node is on the right path the down pointer may become invalid, and discovering this requires walking up toward the root, which can be expensive. One way to delete an arbitrary node x is as follows. First, we replace the subtree rooted at x by the meld of its left and right subtrees, using either top-down or bottom-up melding. Next, we walk up from the root of the melded subtree until reaching either a left child or the root of the entire tree. During this walk we swap the children of every node on the path except the lowest. If the node reached is the root, we reestablish the left pointer by descending the major path (which was originally the minor path).

By using an appropriate potential function, we can derive good amortized time bounds for this representation. We must approximately double the potential used in § 3, to account for the extra time spent descending right paths to establish down pointers. We define the potential of a tree to be twice the number of heavy right nodes not on the major path, plus the number of heavy right nodes on the major path, plus four times the number of light right nodes on the minor path, plus three times the number of light right nodes on the major path. Notice that a right node on the minor path has one more credit than it would have if it were on the major path. The extra credits on the minor path pay for a traversal of it when necessary to establish the down pointer. A straightforward extension of the analysis in § 3 proves the following theorem.

THEOREM 3. *On bottom-up skew heaps represented in threaded fashion with down pointers, the heap operations have the following amortized running times:* $O(1)$ *for make heap, find min, insert, and meld;* $O(\log n)$ *for delete min or delete on an n-node heap;* $O(n)$ *for make heap(s) on a set of size n;* $O(k+k\log(n/k))$ *for purge on an n-node heap if k items are purged;* $O(j+k+k\log(n/k))$ *for find all with purging on an n-node heap if j items are returned and k items are purged.*

5. A lower bound. The notion of amortized complexity affords us the opportunity to derive lower bounds, as well as upper bounds, on the efficiency of data structures. For example, the compressed tree data structure for disjoint set union is optimal to within a constant factor in an amortized sense among a wide class of pointer manipulation algorithms [15]. We shall derive a similar but much simpler result for bottom-up skew heaps: in an amortized sense, skew heaps are optimal to within a constant factor on any sequence of certain heap operations.

To simplify matters, we shall allow only the operations *meld* and *delete min*. We assume that there is an arbitrary initial collection of single-item heaps and that an arbitrary but fixed sequence of *meld* and *delete min* operations is to be carried out. An algorithm for carrying out this sequence must return the correct answers to the *delete min* operations whatever the ordering of the items; this ordering is initially completely unspecified. The only restriction we make on the algorithm is that it make binary rather than multiway decisions; thus binary comparisons are allowed but not radix sorting, for example.

Suppose there are a total of m operations, and that the *ith delete min* operation is on a heap of size n_i. If we carry out the sequence using bottom-up skew heaps, the total running time is $O(m+\sum_i \log n_i)$. We shall prove that any correct algorithm must make at least $\sum_i \log n_i$ binary decisions; thus, if we assume that any operation takes $\Omega(1)$ time, bottom-up skew heaps are optimum to within a constant factor in an amortized sense.

The proof is a simple application of information theory. The various possible orderings of the items produce different correct outcomes for the *delete min* instructions. For an algorithm making binary decisions, the binary logarithm of the total number of possible outcomes is a lower bound on the number of decisions in the worst case. The ith *delete min* operation has n_i possible outcomes, regardless of the outcomes of the previous *delete min* operations. (Any item in the heap can be the minimum.) Thus the total number of possible outcomes of the entire sequence is $\prod_i n_i$, and the number of binary decisions needed in the worst case is $\sum_i \log n_i$.

This lower bound is more general than it may at first appear. For example, it allows insertions, which can be simulated by melds. However, the bound does not apply to situations in which some of the operations constrain the outcome of later ones, as for instance when we perform a *delete min* on a heap, reinsert the deleted item, and perform another *delete min*.

6. Remarks. The top-down skew heaps we have introduced in § 2 are simpler than leftist heaps and as efficient, to within a constant factor, on all the heap operations. By changing the data structure to allow bottom-up melding, we have reduced the amortized time of *insert* and *meld* to $O(1)$, thereby obtaining a data structure with optimal efficiency on any sequence of *meld* and *delete min* operations. Table 1 summarizes our complexity results.

TABLE 1
Amortized running times of skew heap operations.

	top-down skew heaps	bottom-up skew heaps
make heap	$O(1)$	$O(1)$
find min	$O(1)$	$O(1)$
insert	$O(\log n)$	$O(1)$
meld	$O(\log n)$	$O(1)$
delete min	$O(\log n)$	$O(\log n)$
delete	$O(\log n)$	$O(\log n)$

Several interesting open problems remain. On the practical side, there is the question of exactly what pointer structure and what implementation of the heap operations will give the best empirical behavior. On the theoretical side, there is the problem of extending the lower bound in § 5 to allow other combinations of operations, and of determining whether skew heaps or any other form of heaps are optimal in a more general setting. Two very recent results bear on this question. Fredman (private communication) has shown that the amortized bound of $O(k + k \log (n/k))$ we derived for k deletions followed by a *find min* is optimum for comparison-based algorithms. Fredman and Tarjan [6] have proposed a new kind of heap, called the *Fibonacci heap*, that has an amortized time bound of $O(\log n)$ for arbitrary deletion and $O(1)$ for *find min*, *insert*, *meld*, and the following operation, which we have not considered in this paper:

decrease(x, y, h): Replace item x in heap h by item y, known to be no greater than x.

The importance of Fibonacci heaps is that *decrease* is the dominant operation in many network optimization algorithms, and the use of Fibonacci heaps leads to improved time bounds for such algorithms [6]. The Fibonacci heap cannot properly be called a self-adjusting structure, because explicit balance information is stored in the nodes. This leads to the open problem of devising a self-adjusting heap implementa-

tion with the same amortized time bounds as Fibonacci heaps. Skew heaps do not solve this problem, because *decrease* (implemented as a deletion followed by an insertion or in any other obvious way) takes $\Omega(\log n)$ amortized time.

More generally, our results only scratch the surface of what is possible using the approach of studying the amortized complexity of self-adjusting data structures. We have also analyzed the amortized complexity of self-adjusting lists, and in particular the move-to-front heuristic, under various cost measures [12], and we have devised a form of self-adjusting search tree, the *splay tree*, which has a number of remarkable properties and applications [13]. The field is ripe for additional work.

Appendix.

Tree terminology. We consider binary trees as defined by Knuth [6]: every tree node has two children, a *left child* and a *right child*, either or both of which can be the special node null. If node y is a child of node x, then x is the *parent* of y, denoted by $p(y)$. The *root* of the tree is the unique node with no parent. If $x = p^i(y)$ for some $i \geq 0$, x is an *ancestor* of y and y is a *descendant* of x; if $i > 0$, x is the *proper ancestor* of y and y a *proper descendant* of x. The *right path* descending from a node x is the path obtained by starting at x and repeatedly proceeding to the right child of the current node until reaching a node with null right child; we define the *left path* descending from x similarly. The *right path* of a tree is the right path descending from its root; we define the *left path* similarly. A *path* from a node x to a *missing node* is a path from x to null, such that each succeeding node is a child of the previous one. The direction from parent to child is *downward* in the tree; from child to parent, *upward*.

REFERENCES

[1] A. V. AHO, J. E. HOPCROFT AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[2] M. R. BROWN, *Implementation and analysis of binomial queue algorithms*, this Journal, 7 (1978), pp. 298-319.

[3] D. CHERITON AND R. E. TARJAN, *Finding minimum spanning trees*, this Journal, 5 (1976), pp. 724-742.

[4] C. A. CRANE, *Linear lists and priority queues as balanced binary trees*, Technical Report STAN-Cs-72-259, Computer Science Dept, Stanford Univ., Stanford, CA, 1972.

[5] E. W. DIJKSTRA, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

[6] M. L. FREDMAN AND R. E. TARJAN, *Fibonacci heaps and their uses in network optimization algorithms*, Proc. 25th Symposium on Foundations of Computer Science, 1984, pp. 338-346.

[7] D. E. KNUTH, *The Art of Computer Programming, Volume* 1: *Fundamental Algorithms*, 2nd ed., Addison-Wesley, Reading, MA, 1973.

[8] ———, *The Art of Computer Programming, Volume* 3: *Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

[9] D. D. SLEATOR, *An $O(nm \log n)$ algorithm for maximum network flow*, Technical Report STAN-CS-80-831, Computer Science Dept, Stanford Univ., Stanford, CA, 1980.

[10] D. D. SLEATOR AND R. E. TARJAN, *A data structure for dynamic trees*, J. Comp. System Sci., 26 (1983), pp. 362-391; also Proc. Thirteenth Annual ACM Symposium on Theory of Computing, 1981, pp. 114-122.

[11] ———, *Self-adjusting binary trees*, Proc. Fifteenth Annual ACM Symposium on Theory of Computing, 1983, pp. 235-246.

[12] ———, *Amortized efficiency of list update and paging rules*, Comm. ACM, 28 (1985), pp. 202-208.

[13] ———, *Self-adjusting binary search trees*, J. Assoc. Comput. Mach., to appear.

[14] R. E. TARJAN, *Efficiency of a good but not linear set union algorithm*, J. Assoc. Comput. Mach., 22 (1975), pp. 215-225.

[15] ———, *A class of algorithms which require nonlinear time to maintain disjoint sets*, J. Comp. System Sci., 18 (1979), pp. 110-127.

[16] ———, *Data Structures and Network Algorithms*, CBMS Regional Conference Series in Applied Mathematics 44, Society for Industrial and Applied Mathematics, Philadelphia, 1983.

[17] J. VUILLEMIN, *A data structure for manipulating priority queues*, Comm. ACM, 21 (1978), pp. 309-314.