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.

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.*

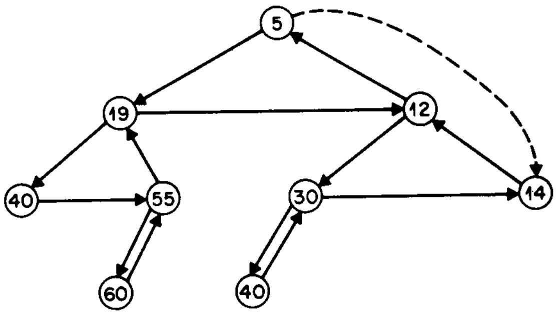

FIG. 7. *Threaded representation of a skew heap. All only children are regarded as being left. The dashed pointer is the down pointer, used only in the bottom-up version.*

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 implementation 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 $i \geqq 0$, 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.