SIAM J. ALG. DISC. METH.
Vol. 6, No. 2, April 1985

016

# AMORTIZED COMPUTATIONAL COMPLEXITY*

ROBERT ENDRE TARJAN†

**Abstract.** A powerful technique in the complexity analysis of data structures is *amortization*, or averaging over time. Amortized running time is a realistic but robust complexity measure for which we can obtain surprisingly tight upper and lower bounds on a variety of algorithms. By following the principle of designing algorithms whose amortized complexity is low, we obtain "self-adjusting" data structures that are simple, flexible and efficient. This paper surveys recent work by several researchers on amortized complexity.

**ASM(MOS) subject classifications.** 68C25, 68E05

**1. Introduction.** Webster's [34] defines "amortize" as "to put money aside at intervals, as in a sinking fund, for gradual payment of (a debt, etc.)." We shall adapt this term to computational complexity, meaning by it "to average over time" or, more precisely, "to average the running times of operations in a sequence over the sequence." The following observation motivates our study of amortization: In many uses of data structures, a sequence of operations, rather than just a single operation, is performed, and we are interested in the total time of the sequence, rather than in the times of the individual operations. A worst-case analysis, in which we sum the worst-case times of the individual operations, may be unduly pessimistic, because it ignores correlated effects of the operations on the data structure. On the other hand, an average-case analysis may be inaccurate, since the probabilistic assumptions needed to carry out the analysis may be false. In such a situation, an amortized analysis, in which we average the running time per operation over a (worst-case) sequence of operations, can yield an answer that is both realistic and robust.

To make the idea of amortization and the motivation behind it more concrete, let us consider a very simple example. Consider the manipulation of a stack by a sequence of operations composed of two kinds of unit-time primitives: *push*, which adds a new item to the top of the stack, and *pop*, which removes and returns the top item on the stack. We wish to analyze the running time of a sequence of operations, each composed of zero or more pops followed by a push. Assume we start with an empty stack and carry out $m$ such operations. A single operation in the sequence can take up to $m$ time units, as happens if each of the first $m-1$ operations performs no pops and the last operation performs $m-1$ pops. However, altogether the $m$ operations can perform at most $2m$ pushes and pops, since there are only $m$ pushes altogether and each pop must correspond to an earlier push.

This example may seem too simple to be useful, but such stack manipulation indeed occurs in applications as diverse as planarity-testing [14] and related problems [24] and linear-time string matching [18]. In this paper we shall survey a number of settings in which amortization is useful. Not only does amortized running time provide a more exact way to measure the running time of known algorithms, but it suggests that there may be new algorithms efficient in an amortized rather than a worst-case sense. As we shall see, such algorithms do exist, and they are simpler, more efficient, and more flexible than their worst-case cousins.

* Received by the editors December 29, 1983. This work was presented at the SIAM Second Conference on the Applications of Discrete Mathematics held at Massachusetts Institute of Technology, Cambridge, Massachusetts, June 27–29, 1983.

† Bell Laboratories, Murray Hill, New Jersey 07974.

The paper contains five sections. In § 2 we develop a theoretical framework for analyzing the amortized running time of operations on a data structure. In § 3 we study three uses of amortization in the analysis of known algorithms. In § 4 we discuss two new data structures specifically designed to have good amortized efficiency. Section 5 contains conclusions.

**2. Two views of amortization.** In order to analyze the amortized running time of operations on a data structure, we need a technique for averaging over time. In general, on data structures of low amortized complexity, the running times of successive operations can fluctuate considerably, but only in such a way that the average running time of an operation in a sequence is small. To analyze such a situation, we must be able to bound the fluctuations. We shall consider two ways of doing this.

The first is the *banker's view* of amortization. We assume that our computer is coin-operated; inserting a single coin, which we call a *credit*, causes the machine to run for a fixed constant amount of time. To each operation we allocate a certain number of credits, defined to be the *amortized time* of the operation. Our goal is to show that all the operations can be performed with the allocated credits, assuming that we begin with no credits and that unused credits can be carried over to later operations. If desired we can also allow borrowing of credits, as long as any debt incurred is eventually paid off out of credits allocated to operations.

Saving credits amounts to averaging forward over time, borrowing to averaging backward. If we can prove that we never need to borrow credits to complete the operations, then the actual time of any initial part of a sequence of operations is bounded by the sum of the corresponding amortized times. If we need to borrow but such borrowing can be paid off by the end of the sequence, then the total time of the operations is bounded by the sum of all the amortized times, although in the middle of the sequence we may be running behind. That is, the current elapsed time may exceed the sum of the amortized times by the current amount of net borrowing.

In order to keep track of saved or borrowed credits, it is generally convenient to store them in the data structure. Regions of the structure containing credits are unusually hard to access or update (the credits saved are there to pay for extra work); regions containing "debits" are unusually easy to access or update. It is important to realize that this is only an accounting device; the programs that actually manipulate the data structure contain no mention of credits or debits.

The banker's view of amortization was used implicitly by Brown and Tarjan [8] in analyzing the amortized complexity of 2, 3 trees and was developed more fully by Huddleston and Mehlhorn [15], [16] in their analysis of generalized B-trees. We can cast our analysis of a stack in the banker's framework by allocating two credits per operation. The stack manipulation maintains the invariant that the number of saved credits equals the number of stacked items. During an operation, each pop is paid for by a saved credit, the push is paid for by one of the allocated credits, and the other allocated credit is saved, corresponding to the item stacked by the push.

Our second view of amortization is that of the *physicist*. We define a *potential function* $\Phi$ that maps any configuration $D$ of the data structure into a real number $\Phi(D)$ called the *potential* of $D$. We define the *amortized time* of an operation to be $t+\Phi(D')-\Phi(D)$, where $t$ is the actual time of the operation and $D$ and $D'$ are the configurations of the data structure before and after the operation, respectively. With this definition we have the following equality for any sequence of $m$ operations:

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

where $t_i$ and $a_i$ are the actual and amortized times of the $i$th operation, respectively, $\Phi_i$ is the potential after the $i$th operation, and $\Phi_0$ is the potential before the first operation. That is, the total time of the operations equals the sum of their amortized times plus the net decrease in potential from the initial to the final configuration.

We are free to choose the potential function in any way we wish; the more astute the choice, the more informative the amortized times will be. In most cases of interest, the initial potential is zero and the potential is always nonnegative. In such a situation the total amortized time is an upper bound on the total time. (This corresponds to the banker's view of amortization with no borrowing.)

The physicist's view of amortization was proposed by D. Sleator (private communication). To fit the stack manipulation example into the physicist's framework, we define the potential of a stack to be the number of items it contains. Then a stack operation consisting of $k$ pops and one push on a stack initially containing $i$ items has an amortized time of $(k+1)+(i-k+1)-i=2$. The initial potential is zero and the potential is always nonnegative, so $m$ operations take at most $2m$ pushes and pops.

The banker's and physicist's views of amortization are entirely equivalent; we can choose whichever view gives us more intuition about the problem at hand. It is perhaps more natural to deal with fractional amounts of time using the physicist's view, whereas the banker's view is more concrete, but both will yield the same bounds.

**3. Amortized analysis of known algorithms.** Amortization has been used in the analysis of several algorithms more complicated than the stack manipulation example. In this section we shall examine three such applications. We study the examples in the order of their conceptual complexity, which coincidentally happens to be reverse chronological order.

Our first example is the "move-to-front" list updating heuristic. Consider an abstract data structure consisting of a table of $n$ items, under the operation of accessing a specified item. We assume that the table is represented by a linear list of the items in arbitrary order, and that the time to access the $i$th item in the list is $i$. In addition, we allow the possibility of rearranging the list at any time (except in the middle of an access), by swapping any pair of contiguous items. Such a swap takes one unit of time.

We are interested in whether swapping can reduce the time for a sequence of accesses, and whether there is a simple heuristic for swapping that achieves whatever improvement is possible. These questions are only interesting if the access sequence is nonuniform, e.g. some items are accessed more frequently than others, or there is some correlation between successive accesses. Among the swapping heuristics that have been proposed are the following:

*Move-to-front.* After an access, move the accessed item to the front of the list, without changing the relative order of the other items.

*Transpose.* After an access of any item other than the first on the list, move the accessed item one position forward in the list by swapping it with its predecessor.

*Frequency count.* Swap after each access as necessary to maintain the items in non-decreasing order by cumulative access frequency.

The frequency count heuristic requires keeping track of access frequencies, whereas the other two rules depend only on the current access. There has been much research on these and similar update rules, the overwhelming majority of it average-case analysis [6], [7], [17], [23], [26]. All of the average-case studies known to the author are based on the assumption that the accesses are independent identically distributed random variables, i.e. for each successive access, each item $i$ has a fixed probability $p_i$ of being accessed. The usual measure of interest is the asymptotic average access time as a

function of $p_1, p_2, \cdots, p_n$, i.e. the average access time as $m$, the number of accesses, goes to infinity. (Letting $m$ go to infinity eliminates the effect of the initial ordering.)

Under these assumptions, the optimum strategy is to begin with the items in nondecreasing order by probability and leave them that way. The law of large numbers implies that the asymptotic average access time of the frequency count heuristic is minimum, and it has long been known that move-to-front is within a factor of two of minimum [17]. Rivest [23] showed that asymptotically transpose is never worse than move-to-front, although Bitner [7] showed that it converges much more slowly to its asymptotic behavior.

Bentley and McGeogh [6] performed several experiments on real data. Their tests indicate that in practice the transpose heuristic is inferior to frequency count but move-to-front is competitive with frequency count and sometimes better. This suggests that real access sequences have a locality of reference that is not captured by the standard probabilistic model, but that significantly affects the efficiency of the various heuristics. In an attempt to derive more meaningful theoretical results, Bentley and McGeogh did an amortized analysis. Consider any sequence of accesses. Among static access strategies (those that never reorder the list), the strategy that minimizes the total access time is that of beginning with the items in decreasing order by total access frequency. Bentley and McGeogh showed that the total access time of move-to-front is within a factor of two of that of the optimum static strategy, if move-to-front's initial list contains the items in order of first access. Furthermore frequency count but not transpose shares this property.

(Note that the move-to-front heuristic spends only about half its time doing accesses; the remainder is time spent on the swaps that move accessed items to the front of the list. Including swaps, the total time of move-to-front is at most four times the total time of the optimum static algorithm.)

Sleator and Tarjan [26], using the approach presented in § 2, extended Bentley and McGeogh's results to allow comparison between arbitrary dynamic strategies. In particular, they showed that for any initial list and any access sequence, the total time of move-to-front is within a constant factor (four) of minimum over all algorithms, including those with arbitrary swapping. Thus move-to-front is optimum in a very strong, uniform sense (to within a constant factor on *any* access sequence). Neither transpose nor frequency count shares this property.

To obtain the Sleator–Tarjan result we use the physicist's view of amortization. Consider running an arbitrary algorithm A and the move-to-front heuristic MTF in parallel on an arbitrary access sequence, starting with the same initial list for both methods. Define the potential of MTF's list to be the number of inversions in MTF's list with respect to A's list, where an inversion is a pair of items whose order is different in the two lists. The potential is initially zero and always nonnegative. It is straightforward to show that, with this definition of potential, the amortized time spent by MTF on any access is at most four times the actual time spent by A on the access.

The factor of four bound can be refined and extended to allow A and MTF to have different initial lists and to allow the access cost to be a nonlinear function of list position. The problem of minimizing page faults, which is essentially a version of list updating with a nonlinear access cost, can also be analyzed using amortization. Sleator and Tarjan's paper [26] contains the details.

Another use of amortization is in the analysis of insertion and deletion in balanced search trees. A balanced search tree is another way of representing a table, more complicated than a linear list but with faster access time. Extensive discussions of search trees can be found in many computer science texts (e.g. [2], [17], [32]), and we shall assume some familiarity with their properties. Generally speaking, a table

can be stored as a search tree if the items can be totally ordered, e.g. the items are real numbers, which are orderable numerically, or strings, which are orderable lexicographically. We store the items in the nodes of a tree in symmetric order. Depending upon the exact scheme used, the items may be stored in either the internal or the external nodes, with one or several items per node. We access an item by following the path from the tree root to the node containing the item. Thus the time to access an item is proportional to the depth in the tree of the node containing it.

*Balanced* search trees are constrained by some sort of local *balance condition* so that the depth of an $n$-node tree, and thus the worst-case access time, is $O(\log n)$. Typical kinds of balanced trees include AVL or height-balanced trees [1], trees of bounded balance or weight-balanced trees [21], and various kinds of $a$, $b$ trees including 2, 3 trees [2] and $B$-trees [5]]. (In an $a$, $b$ tree for integers $a$ and $b$ such that $2 \leqq a \leqq \lceil b/2 \rceil$, all external nodes have the same depth, and every internal node has at least $a$ and at most $b$ children, except the root, which if internal has at least two and at most $b$ children.) Indeed, the varieties of balanced trees are almost endless.

Maintaining a dynamic table (i.e. a table subject to insertions and deletions) as a balanced search tree requires storing local "balance information" at each tree node and, based on the balance information, rebalancing the tree using local transformations after each insertion or deletion. For standard kinds of balanced trees, the update transformations all take place along a single path in the tree, and the worst-case time for an insertion or deletion is $O(\log n)$. For some kinds of balanced search trees, however, the amortized update time is $O(1)$. Brown and Tarjan [8] showed that $m$ consecutive insertions or $m$ consecutive deletions in an $n$-node 2, 3 tree take $O(n+m)$ total rebalancing time, giving an $O(1)$ amortized time per update if $m = \Omega(n)$. This bound does not hold for intermixed insertions and deletions unless the insertions and deletions are far enough apart that they do not interact substantially. Maier and Salveter [20] and independently Huddleston and Mehlhorn [15, 16] showed that $m$ arbitrarily intermixed insertions and deletions in an $n$-node 2, 4 tree, or indeed in an $a$, $b$ tree with $a \leqq \lfloor b/2 \rfloor$, take $O(n+m)$ total rebalancing time.

To give the flavor of these results, we shall sketch an amortized analysis of insertions in balanced binary trees [31], [32], also known as "symmetric binary B-trees" [4], "red-black trees" [13], or "half-balanced trees" [22]. (See Fig. 1.) A balanced binary tree is a binary tree (each internal node has exactly two children: a left child and a right child) in which each internal node is colored either *red* or *black*, such that

(i) all paths from the root to an external node contain the same number of black nodes, and

(ii) any red node has a black parent. (In particular, the root, if internal, is black.)

Balanced binary trees are equivalent to 2, 4 trees: we obtain the 2, 4 tree corresponding to a balanced binary tree by contracting every red node into its parent.

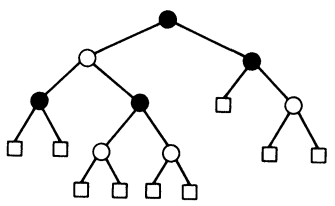

FIG. 1. *A balanced binary tree. Circles are internal nodes; squares are external. Internal nodes are solid if black, hollow if red.*

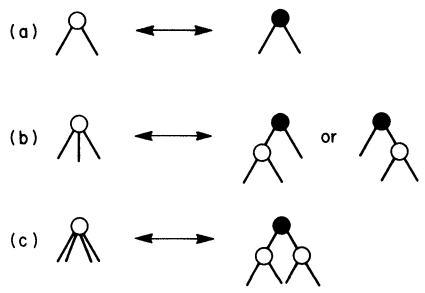


FIG. 2. *Correspondence between nodes of a* 2, 4 *tree and nodes of a balanced binary tree.*
(a) 2-*node.*
(b) 3-*node.*
(c) 4-*node.*

(See Fig. 2.) This correspondence is not one-to-one: a 2, 4 tree can correspond to several different balanced binary trees, because there are two different configurations corresponding to a 3-node (a node with three children).

We shall not go into the details of how a table can be represented by a balanced binary tree and why the depth of an $n$-node balanced binary tree is $O(\log n)$. (See [4], [13], [22], [31], [32].) For our purposes it suffices to know that the effect of an insertion is to convert some external node into a red internal node with two external children. This may produce a violation of property (ii). To restore (ii), we walk up the path from the violation, repeatedly applying the appropriate case from among the five cases illustrated in Figs. 3 and 4. Cases 2a, b, c are terminating: applying either of them restores (ii). Cases 1a, b are (possibly) nonterminating: after applying either of them we must look for a new violation.

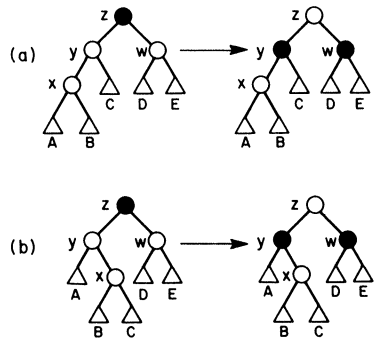


FIG. 3. *Nonterminating cases of a balanced binary tree insertion. Triangles denote subtrees whose root is either black or external. Node z may or may not be the root. Each case has a symmetric variant, not shown. After applying either case, we must check whether the parent of z is red.*
(a) *Case* 1a: *color flip.*
(b) *Case* 1b: *color flip.*

The net effect of rebalancing is to change the color of one or more nodes and possibly to change the structure of the tree by a "single rotation" (Case 2b) or a "double rotation" (Case 2c). We can prove that the total time for $m$ consecutive insertions in a tree of $n$-nodes is $O(n+m)$ by using the banker's view of amortization. We maintain the invariant that every black node contains either 0, 1, or 2 credits, depending on whether it has one red child, no red children, or two red children, respectively. To satisfy the invariant initially we must add $O(n)$ credits to the tree.

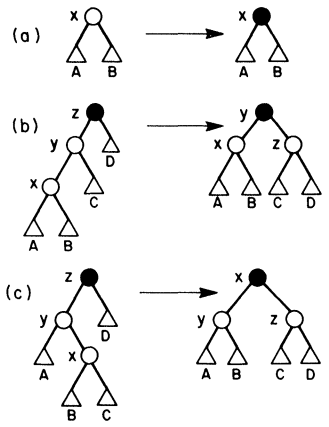


FIG. 4. *Terminating cases of a balanced binary tree insertion. Each case has a symmetric variant, not shown.*

(a) *Case* 2a: *color change at the root.*

(b) *Case* 2b: *single rotation.*

(c) *Case* 2c: *double rotation.*

(This accounts for the $O(n)$ term in the bound.) Each of Cases 2a, b, c requires the addition of $O(1)$ credits to the tree, but such a case terminates an insertion. Each of Cases 1a, b, if nonterminating, releases a credit from the tree to pay for the transformation.

This argument is an adaption of those in [15], [16], [20]. It is not hard to extend the argument to prove an $O(n+m)$ time bound for arbitrarily intermixed insertions and deletions if the deletion algorithm is suitable. (See [31], [32] for a suitable deletion algorithm.)

The $O(n+m)$ bound on update time does not take into account the time necessary to search for the positions at which the insertions and deletions are to take place. The practical importance of this bound is in situations where the search time is significantly faster than $O(\log n)$, as can occur if the search tree is augmented to allow searching from "fingers" [8], [12], [16], [19]. Search trees with fingers provide one way to take advantage of locality of reference in an access sequence, and a generalization of the argument we have sketched shows that in appropriate kinds of balanced trees with fingers, the total rebalancing time is bounded by a constant factor times the total search time, if we perform an arbitrary sequence of intermixed accesses, insertions, and deletions [16]. Brown and Tarjan [8] list several applications of such trees. The amortized approach to fingers [8], [16] is significantly simpler than the worst-case approach [12], [19].

Our third and most complicated example of amortization is in the analysis of path compression heuristics for the disjoint set union problem, sometimes called the "union-find problem" or the "equivalence problem." We shall formulate this problem as follows. We wish to represent a collection of disjoint sets, each with a distinguishing name, under two kinds of operations:

*find* $(x)$: Return the name of the set containing element $x$.

*unite* $(A, B)$: Form the union of the two sets named $A$ and $B$, naming the new set $A$. This operation destroys the old sets named $A$ and $B$.

We shall assume that the initial sets are all singletons. To solve this problem, we represent each set by a tree whose nodes are the elements in the set. Each node points to its parent; the root contains the set name. To carry out *find* $(x)$, we follow the path

of parent pointers from $x$ to the root of the tree containing it, and return the name stored there. To carry out *unite* $(A, B)$, we locate the nodes containing the names $A$ and $B$ and make one the parent of the other, moving the name $A$ to the new root if necessary.

This basic method is not very efficient; a sequence of $m$ operations beginning with $n$ singleton sets can take $O(nm)$ time, for an amortized bound of $O(n)$ per operation. We can improve the method considerably by adding heuristics to the *find* and *unite* algorithms to reduce the tree depths. When performing *unite*, we use *union by size*, making the root of the smaller tree point to the root of the larger. After performing *find* $(x)$, we use *path compression*, changing the parent of $x$ and all its ancestors except the tree root to be the root. (See Fig. 5.)

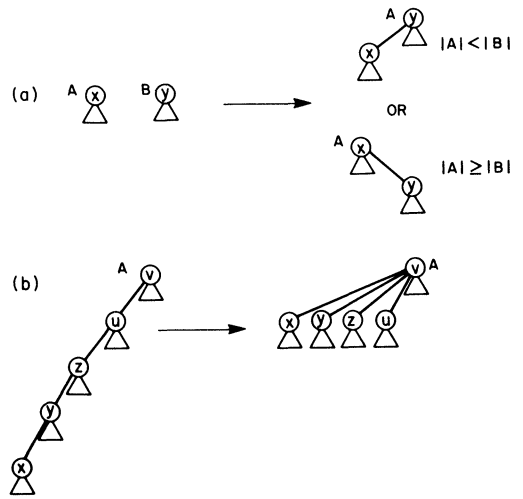


FIG. 5. *Implementation of set operations. Triangles denote subtrees.*
(a) *unite* $(A, B)$.
(b) *find* $(x)$.

Union by size was proposed by Galler and Fischer [11]; McIlroy and Morris devised path compression [2]. The set union algorithm with both heuristics is extremely hard to analyze. Tarjan [29] derived an $O(m\alpha(m, n))$ bound for $m$ operations starting with $n$ singletons, assuming $m = \Omega(n)$. Here $\alpha$ is a functional inverse of Ackermann's function. Tarjan's proof is a complicated amortized analysis that uses debits as well as credits. For a version of the proof in the banker's framework see [32]. The bound is tight to within a constant factor in the worst case for a large class of pointer multiplication algorithms [30]. Tarjan and van Leeuwen [33] extended the bound to allow values of $m$ much smaller than $n$ (the generalized bound is $O(n + m\alpha(n + m, n))$) and to cover a variant of union by size and several variants of path compression. Recently Gabow and Tarjan [10] found a linear-time algorithm for a special case of disjoint set union in which there is appropriate advance knowledge about the unions. Their algorithm combines path compression with table look-up on small sets and requires the power of a random access machine. The method apparently does not extend to the general problem.

**4. New "self-adjusting" data structures.** Data structures efficient in the worst case, such as the various kinds of balanced trees, have several disadvantages. The maintenance of a structural constraint, such as a balance condition, consumes both storage space (though possibly only one bit per node) and running time. Restructuring

after an update tends to be complicated, involving a number of cases. Perhaps more significantly, such data structures are inflexible in that they cannot take advantage of whatever nonuniformity there may be in the usage pattern.

The idea of amortization suggests another way to design data structures. Each time we access the structure, we modify it in a simple, uniform way, with the intent of decreasing the time required for future operations. This approach can produce a data structure with very simple access and update procedures that needs no extra storage for structural information and adapts to fit the usage pattern. An example of such a data structure is a linear list with the move-to-front rule, studied in § 3. Previous authors have used the term "self-organizing" for such data structures. We shall call them *self-adjusting*. In this section we describe two self-adjusting data structures recently invented by Sleator and Tarjan [25], [27], [28].

The first structure, the *skew heap*, is for the representation of meldable heaps, also called "priority queues" [17] and "mergable heaps" [2]. Suppose we wish to maintain a collection of disjoint sets called *heaps*, each initially containing a single element selected from a totally ordered universe, under two operations:

*delete min* $(h)$: Delete and return the minimum element in heap $h$.
*meld* $(h_1, h_2)$: Add all elements in heap $h_2$ to $h_1$, destroying $h_2$.

To represent a heap, we use a binary tree, each internal node of which is a heap element. We arrange the elements in *heap order*: the parent of any node is smaller than the node itself. Thus the root of the tree is the smallest element. Melding is the fundamental operation. We carry out *delete min* $(h)$ by deleting the root of $h$, replacing $h$ by the meld of its left and right subtrees, and returning the deleted node. We carry out *meld* $(h_1, h_2)$ by walking down the right paths from the roots of $h_1$ and $h_2$, merging them. The left subtrees of nodes along these paths are unaffected by the merge. The merge creates a potentially long right path in the new tree. As a heuristic to keep right paths short, we conclude the meld by swapping left and right children of all nodes except the deepest along the merge path. (See Fig. 6.) We call the resulting data structure a *skew heap*.

Skew heaps are a self-adjusting version of the leftist queues of Crane [9] and Knuth [17]; leftist queues are heap-ordered binary trees maintained so that the right path down from any node is a shortest path to an external node. In skew heaps, the amortized times of *delete min* and *meld* are $O(\log n)$, where $n$ is the total number of elements in the heap or heaps involved. To prove this, we define the *weight* of an internal node $x$ to be the total number of its internal node descendants, including $x$ itself. We define a node $x$ to be *heavy* if it is not a root and its weight is more than half the weight of its parent. We maintain the invariant that every heavy right child has a credit. An analysis of the effect of *delete min* and *meld* gives the $O(\log n)$ bound [25], [27].

Skew heaps require only a single top-down pass for melding, in contrast to leftist heaps, which need a top-down pass followed by a bottom-up pass. If we modify skew heaps so that melding is bottom-up, we can reduce the amortized time for *meld* to $O(1)$ while retaining the $O(\log n)$ bound for *delete min*. In an amortized sense, to within a constant factor, bottom-up skew heaps are optimum among all comparison-based methods for representing heaps. Preliminary experiments indicate that they are efficient in practice as well as in theory. For details of these results, see [27].

Our second structure, the *splay tree*, is a self-adjusting form of binary search tree. Consider the table look-up problem that we solved in § 2 using a self-adjusting list. As discussed in § 3, if the items are totally orderable, we can also represent such a

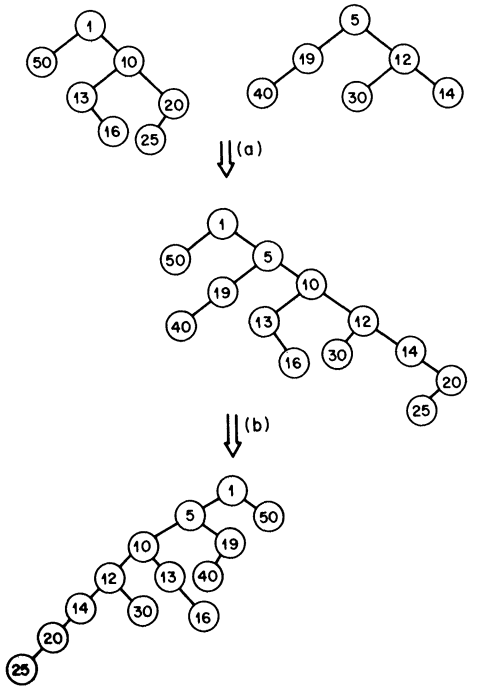


FIG. 6. *A meld of two skew heaps. External nodes are not shown.*
(a) *Merge of the right paths.*
(b) *Swapping of children along the path formed by the merge.*

table by a binary tree: Each item is an internal node of the tree, with items arranged in *symmetric order*: for any item $x$, all items in its left subtree are less than $x$ and all items in its right subtree are greater than $x$. To access an item $x$, we compare $x$ to the tree root, stop if the root is $x$, and otherwise proceed recursively in the left subtree if $x$ is less than the root, in the right subtree if $x$ is greater than the root. The time to access $x$ is proportional to its depth in the tree.

As a heuristic to keep the tree depth small, each time we access a node $x$ we *splay* it. To splay $x$, we repeatedly apply the appropriate one of the cases among those in Fig. 7, continuing until $x$ is the root of the tree. In effect, we walk up the path from $x$ two nodes at a time, performing rotations as we go up that move $x$ to the root and move the rest of the nodes on the access path about halfway or more toward the root. (See Fig. 8.) We call the resulting data structure a *splay tree.*

The amortized time of an access in an $n$-node splay tree is $O(\log n)$. To prove this, we define the potential of a splay tree to be the sum over all internal nodes $x$ of $\log w(x)$, where $w(x)$ is the weight of $x$, defined to be the number of (internal node) descendants of $x$, including $x$ itself. The algorithm and the analysis extend to handle insertion, deletion, and more drastic update operations. Several variants of the splay heuristic have the same efficiency (to within a constant factor) [25], [27]. Several heuristics for search trees proposed earlier [3], [7] are not as efficient in an amortized sense.

Splay trees are not only as efficient in an amortized sense as balanced trees, but also as efficient as static optimum search trees, as an extension of the analysis shows. In this they are like lists with move-to-front updating; they automatically adapt to fit the access frequencies. The result showing that splay trees are as efficient as optimum

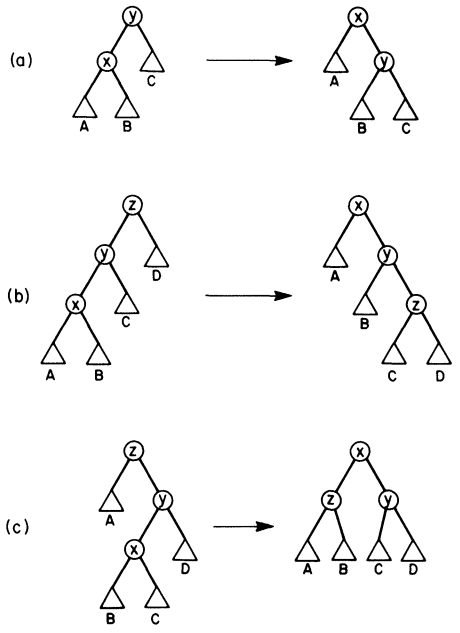


FIG 7. *Case of splay step at node x. Each case has a symmetric variant (not shown). In cases* (b) *and* (c), *if node z is not the root, the splay continues after the step.*

(a) *Terminating single rotation. Node y is the root.*

(b) *Two single rotations.*

(c) *Double rotation.*

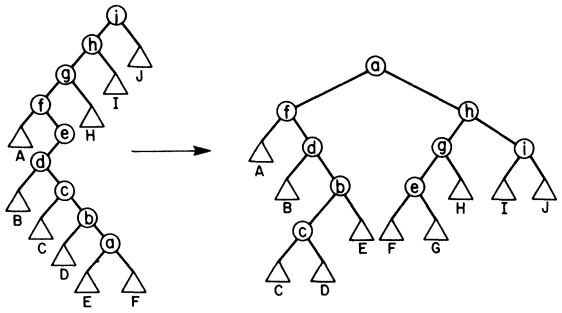

FIG. 8. *Splay at node a.*

trees is analogous to Bentley and McGeough's result comparing move-to-front with an optimum static ordering. We conjecture that a stronger result analogous to Sleator and Tarjan's result for move-to-front holds; namely splay trees minimize the amortized access time to within a constant factor among all search-tree-based algorithms. We are currently attempting to prove an appropriate formalization of this conjecture. As a special case, the truth of the conjecture would imply that splay trees are as efficient as the finger search trees mentioned in § 3, and thus that one can obtain the advantages of fingers using an ordinary search tree, without extra pointers. Details of the properties of splay trees and several applications to more elaborate self-adjusting data structures appear in [27].

**5. Conclusions.** We have seen that amortization is a powerful tool in the algorithmic analysis of data structures. Not only does it allow us to derive tighter bounds for known algorithms, but it suggests a methodology for algorithm development that leads to new simple, efficient, and flexible "self-adjusting" data structures. Amortization also provides a robust way to study the possible optimality of various data structures. It seems likely that amortization will find many more uses in the future.

## REFERENCES


[1] G. M. ADELSON-VELSKII AND E. M. LANDIS, *An algorithm for the organization of information*, Soviet Math. Dokl., 3 (1962), pp. 1259–1262.

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

[3] B. ALLEN AND I. MUNRO, *Self-organizing search trees*, J. ACM, 25 (1978), pp. 526–535.

[4] R. BAYER, *Symmetric binary B-trees: data structure and maintenance algorithms*, Acta. Inform., 1 (1972), pp. 290–306.

[5] R. BAYER AND E. MCCREIGHT, *Organization of large ordered indexes*, Acta Inform., 1 (1972), pp. 173–189.

[6] J. L. BENTLEY AND C. MCGEOGH, *Worst-case analysis of self-organizing sequential search heuristics*, Proc. 20th Allerton Conference on Communication, Control, and Computing, to appear.

[7] J. R. BITNER, *Heuristics that dynamically organize data structures*, SIAM J. Comput., 8 (1979), pp. 82–110.

[8] M. R. BROWN AND R. E. TARJAN, *Design and analysis of a data structure for representing sorted lists*, SIAM J. Comput., 9 (1980), pp. 594–614.

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

[10] H. N. GABOW AND R. E. TARJAN, *A linear-time algorithm for a special case of disjoint set union*, J. Comput. Sys. Sci., submitted.

[11] B. A. GALLER AND M. J. FISCHER, *An improved equivalence algorithm*, Comm. ACM, 7 (1964), pp. 301–303.

[12] L. J. GUIBAS, E. M. MCCREIGHT, M. F. PLASS AND J. R. ROBERTS, *A new representation for linear lists*, Proc. Ninth Annual ACM Symposium on Theory of Computing, 1977, pp. 49–60.

[13] L. J. GUIBAS AND R. SEDGEWICK, *A dichromatic framework for balanced trees*, Proc. Nineteenth Annual IEEE Symposium on Foundations of Computer Science, 1978, pp. 8–21.

[14] J. HOPCROFT AND R. TARJAN, *Efficient planarity testing*, J. ACM, 21 (1974), pp. 549–568.

[15] S. HUDDLESTON AND K. MEHLHORN, *Robust balancing in B-trees*, Proc. 5th GI-Conference on Theoretical Computer Science, Lecture Notes in Computer Science 104, Springer-Verlag, New York, 1981, pp. 234–244.

[16] S. HUDDLESTON AND K. MEHLHORN, *A new data structure for representing sorted lists*, Acta Inform., 17 (1982), pp. 157–184.

[17] D. E. KNUTH, *The Art of Computer Programing, Vol.* 3: *Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

[18] D. E. KNUTH, J. H. MORRIS JR. AND V. R. PRATT, *Fast pattern matching in strings*, SIAM J. Comput., 6 (1977), pp. 323–350.

[19] S. R. KOSARAJU, *Localized search in sorted lists*, Proc. Thirteenth Annual ACM Symposium on Theory of Computing, 1978, pp. 62–69.

[20] D. MAIER AND S. C. SALVETER, *Hysterical B-trees*, Inform. Proc. Letters, 12 (1981), pp. 199–202.

[21] J. NIEVERGELT AND E. M. REINGOLD, *Binary search trees of bounded balance*, SIAM J. Comput., 2 (1973), pp. 33–43.

[22] H. OLIVIE, *A new class of balanced search trees: half-balanced binary search trees*, RAIRO Inform. théorique/Theoretical Informatics, 6 (1982), pp. 51–71.

[23] R. RIVEST, *On self-organizing sequential search heuristics*, Comm. ACM, 19 (1976), pp. 63–67.

[24] P. ROSENSTIEHL AND R. E. TARJAN, *Gauss codes, planar Hamiltonian graphs, and stack-sortable permutations*, J. Algorithms, to appear.

[25] D. D. SLEATOR AND R. E. TARJAN, *Self-adjusting binary trees*, Proc. Fifteenth Annual ACM Symposium on Theory of Computing, 1983, pp. 235–245.

[26] ———, *Amortized efficiency of list update and paging rules*, Comm. ACM, to appear.

[27] D. D. SLEATOR AND R. E. TARJAN, *Self-adjusting heaps*, SIAM J. Comput., 15 (1986), to appear.
[28] ———, *Self-adjusting binary search trees*, to appear.
[29] R. E. TARJAN, *Efficiency of a good but not linear set union algorithm*, J. ACM, 22 (1975), pp. 215–225.
[30] ———, *A class of algorithms which require nonlinear time to maintain disjoint sets*, J. Comput. Sys. Sci., 18 (1979), pp. 110–227.
[31] ———, *Updating a balanced search tree in $O(1)$ rotations*, Inform. Proc. Letters, 16 (1983), pp. 253–257.
[32] ———, *Data Structures and Network Algorithms*, CBMS Regional Conference Series in Applied Mathematics 44, Society for Industrial and Applied Mathematics, Philadelphia, 1983.
[33] R. E. TARJAN AND J. VAN LEEUWEN, *Worst-case analysis of set union algorithms*, J. ACM, to appear.
[34] *Webster's New World Dictionary of the American Language*, College Edition, World, Cleveland, Ohio, 1964.