ARTICLE

On the efficiency of pairing heaps and related data structures

MICHAEL L FREDMAN, Rutgers University–New Brunswick, New Brunswick, NJ, United States

Journal of the ACM (JACM), Volume 46, Issue 4 (July 1999)
https://doi.org/10.1145/320211.320214
EISSN: 1557-735X

On the Efficiency of Pairing Heaps and Related Data Structures

MICHAEL L. FREDMAN

Rutgers University, New Brunswick, New Jersey

Abstract. The pairing heap is well regarded as an efficient data structure for implementing priority queue operations. It is included in the GNU C++ library. Strikingly simple in design, the pairing heap data structure nonetheless seems difficult to analyze, belonging to the genre of self-adjusting data structures. With its design originating as a self-adjusting analogue of the Fibonacci heap, it has been previously conjectured that the pairing heap provides constant amortized time decrease-key operations, and experimental studies have supported this conjecture. This paper demonstrates, contrary to conjecture, that the pairing heap requires more than constant amortized time to perform decrease-key operations. Moreover, new experimental findings are presented that reveal detectable growth in the amortized cost of the decrease-key operation.

Second, a unifying framework is developed that includes both pairing heaps and Fibonacci heaps. The parameter of interest in this framework is the storage capacity available in the nodes of the data structure for auxiliary balance information fields. In this respect Fibonacci heaps require log log n bits per node when n items are present. This is shown to be asymptotically optimal for data structures that achieve the same asymptotic performance bounds as Fibonacci heaps and fall within this framework.

Categories and Subject Descriptors: E.1 [**Data Structures**]: *queues*; E.2 [**Data Storage Representations**]: *linked representations*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: [Nonnumerical Algorithms and Problems]–*sequencing and scheduling, sorting and searching*

General Terms: Algorithms, Performance, Theory

Additional Key Words and Phrases: Amortized complexity analysis, Fibonacci heaps, lower bounds, priority queues, self-adjusting data structures

In memory of my father, Burton K. Fredman

1. *Introduction*

Pairing heaps were introduced [Fredman et al. 1986] as a self-adjusting alternative to Fibonacci heaps [Fredman and Tarjan 1987]. They are easy to code and provably enjoy log n amortized costs for the standard heap operations. Although it has not been verified that pairing heaps perform the decrease-key operation in constant amortized time (the raison d'être of Fibonacci heaps), this has been

This research was supported in part by National Science Foundation (NSF) grant CCR 97-32689.

Author's address: Department of Computer Science, Rutgers University, New Brunswick, NJ 08903.

conjectured [Fredman et al. 1986] and extensive experimental evidence [Liao 1992; Stasko and Vitter 1987] supports this conjecture. These same experimental studies suggest that pairing heaps are superior to Fibonacci heaps in practice. The good observed behavior of pairing heaps has led to their wide-spread use; they are included, for example, as one of several implementations of priority queues in the GNU C++ library. However, as demonstrated in this paper, pairing heaps do not accommodate decrease-key operations in constant amortized time.

Apart from the practical usefulness of the pairing heap, the question of its efficiency has theoretical interest deriving from the connection between pairing heaps and splaying. The mechanism underlying the pairing heap can be viewed as an application of the self-adjusting splay heuristic of Sleator and Tarjan [1985]. A full explanation of this connection appears in Fredman et al. [1986]; we remark here that in terms of its effect on the pointer structure of the pairing heap, the deletemin operation (as described below) can be viewed as a splay operation. The splay heuristic has been demonstrated to have surprising power in the context of search trees [Sleator and Tarjan 1985]. A recent and notable example concerns the work of Cole [1995] and Cole et al. [1995] on the dynamic finger conjecture for splay trees. As an issue, the competitiveness of the pairing heap thus constitutes an interesting and significant challenge for splaying, testing the limits of this remarkable and powerful heuristic.

In the remainder of this section, we review the basic pairing heap data structure and provide some intuition supporting our main result, that pairing heaps do not accommodate decrease-key operations in constant amortized time. In Section 2, we define a class of data structures, *generalized pairing heaps*, that includes the pairing heap data structure and the variants of this data structure that have been suggested [Fredman et al. 1986; Stasko and Vitter 1987]. We then proceed to formally derive our lower bound, applicable to all data structures in this class. We also develop a unifying framework that includes both generalized pairing heaps and Fibonacci heaps. The parameter of interest in this framework is the storage capacity available in the nodes of the data structure for auxiliary balance information fields. In this respect, Fibonacci heaps require $\log \log n$ bits per node. This is shown to be asymptotically optimal for data structures that achieve the same asymptotic performance bounds as Fibonacci heaps and fall within this framework. In Section 3, we address the issue of whether our lower bound for pairing heaps is subject to experimental detection. We present some new experimental findings and contrast them with previous experimental results.

1.1. PAIRING HEAPS. We begin with a brief description of the pairing heap data structure. (We refer the reader to Fredman et al. [1986] for more details.) This structure is best viewed as a stripped-down, no-frills relative of the Fibonacci heap, designed to be highly efficient from the standpoint of actual implementation. The pairing heap uses a single tree structure to store the values in the heap, one value per tree node. The placement of stored values respects the *heap-order* condition: the value stored in the parent of a node is at most the value stored in the node itself. We thus find the minimum heap value stored in the tree root. There is no restriction on the number of children a node may have, and the children of a node are maintained in a list of siblings. By design, pairing heaps

maintain minimal structure. In particular, parent pointers are not present in the tree nodes.

The heap operations for a pairing heap are implemented as a series of *linking operations*. Let H_1 and H_2 be heap-ordered trees with respective roots x_1 and x_2, and respective root values v_1 and v_2. A linking operation applied to the pair (x_1, x_2) inserts H_1 into H_2 as the leftmost subtree of H_2 if $v_1 \geq v_2$, and otherwise inserts H_2 into H_1 as the leftmost subtree of H_1. To perform an insertion of a value v into a heap whose tree is H, a single node tree containing v is linked with the root of H. To perform a decrease-key operation on the value v stored in a node x of H which is not the root of H, we first remove the subtree rooted at x from H by removing x from its list of siblings. Next, we link x with the root of H, having first decreased the value v stored in x. (If the updated value v of x happens to exceed the value stored in the parent of x, then in principle it would not be necessary to remove x as a child of its parent. However, because the pairing heap does not maintain parent pointers in the tree nodes, this condition cannot be efficiently checked. As a consequence, the decrease-key operation uniformly proceeds by removing x as a child of its parent.) If x is the root of H, then we simply decrease v. To perform a deletemin operation, we remove the root of H and proceed to perform linkings among the children of this root in a certain prescribed order until a single root is restored. The order of the linkings is as follows. Let $x_1, \ldots, x_k$ be the children of the root in left-to-right order. We start by linking the pairs (x_1, x_2), (x_3, x_4) $\ldots$. Let $y_1, \ldots, y_h$, $h = \lceil k/2 \rceil$, be the surviving roots. (If k is odd, then $y_{\lceil k/2 \rceil}$ is x_k.) We finish by linking the pair (y_{h-1}, y_h), then linking y_{h-2} with the root that results from the preceding linking etc., until finally we link y_1 with the root of the structure formed from the linkings of $y_2, \ldots, y_h$.

Several variants of this data structure have been suggested [Fredman et al. 1986; Stasko and Vitter 1987]; the one we have described is the simplest. It bears striking resemblance to the single tree variation of the Fibonacci heap [Fredman and Tarjan 1987], but foregoes certain structural aspects of that data structure that provably ensure asymptotically optimal amortized costs. In particular, with its fixed pattern of node linkings during the deletemin operation, the node pairs that get linked together are not selected on the basis of structural attributes (e.g., node rank), in contrast with the manipulation of Fibonacci heaps. As a consequence, the pairing heap is considered to be a *self-adjusting* data structure.

1.2. Main Results and Intuition. With its brazen abandonment of structure, the pairing heap enjoys local implementation efficiencies that explain the excellent experimental results that have been obtained [Liao 1992; Stasko and Vitter 1987]. Our main result, however, shows that *asymptotic* efficiency is sacrificed; under some circumstances pairing heaps require $\Omega(\log \log n)$ amortized time per decrease-key operation. (This lower bound is *not* known to be tight.) The demonstration of our lower bound stems from information-theoretic considerations. In a nutshell, our argument ultimately captures the following intuition. Consider the family of heap representations that (a) utilize heap-ordered tree structures, (b) link tree roots to reduce the number of potential locations of the minimum heap value, and (c) perform decrease-key operations by repositioning the subtree of the affected node as a separate tree (if only momentarily). In order that the heap operations take place with optimal

asymptotic efficiency, a sizable fraction of the root linkings that take place must be information-theoretically efficient. (Define the *entropy* of the data structure to be the logarithm of the number of linear orderings of the heap nodes, consistent with the heap-order condition. An efficient linking reduces this entropy by an amount that is bounded away from 0.) A data structure in this family, such as the pairing heap, that is constrained to a fixed pattern of node linkings, might plausibly fail to perform a sufficient fraction of efficient node linkings as a consequence of the scrambling impact of the decrease-key operations. Explicating this scrambling effect in the context of a lower bound argument requires that we recognize a *secondary* function served by node pairings, apart from entropy reduction: Node pairings serve to position nodes relative to one another so that subsequent rounds of pairings are information-theoretically efficient. Our goal is to demonstrate that this secondary effect is too limited to efficiently compensate for the scrambling impact of the decrease-key operations.

Suppose we hypothesize that a sizable fraction of node pairings can be suitably efficient only if the heap nodes are suitably arranged. Roughly speaking, each decrease-key operation deforms this arrangement by more than a constant number of bits, whereas each linking can reverse only one bit of deformation. Moreover, these corrective linkings are information-theoretically inefficient. Fibonacci heaps escape these conclusions; their patterns of node linkings are not constrained and the hypothesis of suitable node arrangement is thus circumvented. For technical reasons, our arguments do not utilize the vocabulary of this imagery, but they are nonetheless motivated by it.

Our second result concerns a class of data structures that includes both pairing heaps and Fibonacci heaps, allowing the nodes in the data structure to include balance fields with b bits of information. If $b \leq (1 - c) \log_2 \log_2 n$ where $c > 0$ is any fixed constant, then we find that any such data structure requires $\Omega(\log \log n)$ amortized time per decrease-key operation. As a point of contrast, Fibonacci heaps only require $\log_2 \log_2 n + O(1)$ bits per node. Thus, the lower bound derived for pairing heaps cannot be circumvented, short of providing sufficient node capacity to implement Fibonacci heaps.

2. *Analysis*

The first lower bound in this section applies to all data structures within the class of *generalized pairing heaps*, a class of self-adjusting data structures which we proceed to define. Generalized pairing heaps include as special cases the pairing heap data structure and its variants that have been suggested [Fredman et al. 1986; Stasko and Vitter 1987]. A generalized pairing heap is represented as a sequence of heap-ordered trees, referred to as the *forest* of the heap, along with a control state referred to as the *forest state* of the heap. The *heap structure* refers to both the forest state and the structures of the individual trees (indicated by position) within the forest. More will be said about the forest state, but for the present it is understood that it includes the number of trees in the forest and that it gets updated as operations are performed. An insertion operation is performed by inserting a tree consisting of one node at the end of the forest. A decrease-key operation is performed by changing the value stored in the referenced node. If this node is not a tree root, then the subtree rooted at this node is deleted from the tree containing it and inserted as a new tree at the end of the forest.

Decrease-key and insertion operations are both considered to have 0 *actual* cost. A deletemin operation is executed by (a) performing a sequence of linking operations among tree roots in the forest, simultaneously searching for the minimum tree root; (b) removing the minimum tree root found in step (a), placing its subtrees at the end of the forest as new trees (preserving their relative order as subtrees of the minimum tree root); and (c) possibly performing further linking operations among tree roots in the forest. (A single linking operation combines two trees as described above. The tree, whose root becomes the child of the other tree root, is removed from its position in the forest without otherwise changing the ordering of the trees; likewise, for the tree whose root gets deleted, and whose subtrees get relocated.) We define the actual cost of this operation in terms of the number of comparisons performed, and note that this quantity is at least the number of trees in the forest when the operation commences, less one (the number of comparisons required to find the minimum root in the forest).

It can be seen that the pairing heap data structure itself belongs to this class of data structures, as thus far described, provided that we defer the linkings that are performed when decrease-key and insertion operations are executed, performing them in batch mode just as the next deletemin operation gets underway. Although the above description seemingly limits the manner in which the trees in the forest are potentially arranged or grouped, this is primarily a matter of descriptive convenience and not in actuality *functionally* limiting; there is (as yet) no constraint on the choices of node pairs that get linked during the execution of deletemin operations.

The following constraints are assumed, and for our purposes sufficiently capture the notion of uniformity, concomitant with the "spirit" of a pairing heap. In the following F_1 and F_2 denote heaps, reflecting two instances of the data structure that have identical forest states, and thus equal numbers of trees (but possibly different heap structures). The *position* of a tree or an acknowledged tree root refers to the position that the (associated) tree occupies within the forest.

(a) If an insertion is performed into both F_1 and F_2, then their respective forest states remain identical.

(b) If decrease-key operations are performed on nonroot nodes x_1 in F_1 and x_2 in F_2, and these nodes belong to trees that are correspondingly positioned within F_1 and F_2, then the respective forest states remain identical.

(c) Consider the linkings that take place when a deletemin operation is performed, but fix the root node that gets deleted. Then these linkings can be modeled in terms of a binary *linking-decision* tree that specifies the adaptive sequence of linkings that take place among the tree roots and the children of the deleted root. (Each linking-decision involving the tree root being deleted shows just one potential outcome: that consistent with this being the minimum root.) In other words, upon fixing the root node that gets deleted, the nonlinking comparisons that take place in search of the minimum tree root do not affect the generation of linkings that take place when performing a deletemin operation with a generalized pairing heap.

(d) Suppose that a deletemin operation is performed, respectively, on F_1 and F_2, and that the minimum respective tree roots r_1 and r_2 being deleted are

correspondingly positioned, respectively, within F_1 and F_2, and moreover, have identical numbers of children. Consider the respective linking-decision trees that model these operations as described in (c), but suppress all names of heap nodes within these linking-decision trees, replacing these references with their corresponding positions as tree roots (so indicated), or their corresponding relative positions among the children of the root which is being deleted (so indicated). Then, the resulting linking-decision trees are identical.

(e) Continuing with (d), suppose that the executions of these respective deletemin operations follow identical paths through this common linking-decision tree. (It follows from this assumption that the numbers of remaining trees within the respective transformed forests are identical. Moreover, tree roots correspondingly positioned within the transformed forests would have occupied either corresponding positions as tree roots within the original forests or corresponding relative positions as children of r_1, respectively r_2.) Then, the resulting respective forest states remain identical. In other words, the resulting side-effects are a function only of the linking outcomes.

(f) The forest state of the empty heap is uniquely defined.

The pairing heap and its variants [Fredman et al. 1986] satisfy the above constraints, simply choosing the forest state to consist of the number of trees in the forest. The forest state of the heap can be used to effect the implementation of various constructs, such as the technique of maintaining an *auxiliary area*, introduced by Stasko and Vitter [1987]. In this instance the forest state indicates the number of trees in the forest and also indicates which of these trees are placed in the auxiliary area.

Our lower bound is established by considering sequences of operations structured as follows: First, n insertions are performed so that n items are present in the heap. The remaining operations are partitioned into multiple *rounds*. Each round begins with at most $L = \lceil \frac{1}{2} \log_3 n \rceil$ decrease-key operations performed on the children of a specifically chosen root node ζ in the forest, and concludes with an insertion, followed by deletemin. The deletemin operation, as a consequence of the adversary we define below, will result in the deletion of ζ from the forest. Upon its conclusion, the execution of a round leaves unchanged the number of items n in the heap. Operation sequences structured in this manner, with certain additional requirements described below, are referred to as *O-sequences*.

If generalized pairing heaps enjoyed constant amortized decrease-key costs (and $O(\log n)$ amortized costs for the deletemin and insertion operations), then the total amortized cost of executing a given round of an O-sequence would be $O(L)$, and the total execution cost of an O-sequence consisting of r rounds would be bounded by $O(r \cdot L + n \log n)$. We will see that this is not the case. Before proceeding further with our description of O-sequences, we need to describe the *adversary* that will be utilized to determine the outcomes of linkings posed by a generalized pairing heap algorithm as well as the selections of nodes deleted by the deletemin operations.

2.1. ADVERSARY. Our adversary for linking nodes is defined in terms of a *rank* function assigning integer values to the nodes. For nodes v and w with

respective ranks r_v and r_w, a linking of v and w, with v the left operand, will place v in the root position if and only if $r_v > r_w$. We refer to this as the *rank rule*. The rank function requires a nonnegative parameter $d = d(n)$ chosen in a manner depending on n. A node with no children has rank 0. As children become linked to a node v, with each new child positioned as its leftmost child, the rank of v gets incremented in accordance with the following two cases: First, if v has rank k and a new child with rank $\geq k - d$ is linked to v, then v's rank gets incremented. We refer to this linking as an *efficient* linking, and we describe the child as being *efficiently* linked to v. Also, we define the *efficiency* of a linking to be an indicator specifying whether or not the linking is efficient. Second, the linking of a new child causes v's rank to be incremented, irrespective of the child's rank, if the linking takes place immediately following 3^d consecutive prior linkings, none of which resulted in v's rank being incremented. Observe that the efficiencies of the linkings joining a node v to a specified sublist of its children, in conjunction with the ranks of the remaining children of v, uniquely determine the rank of v as well as the respective efficiencies of *all* linkings to the children of v.

A linking is called *incremental* if it causes the rank of the parent node to be incremented. An incremental linking that is not efficient is referred to as a *default* incremental linking. Finally, the rank of a tree is defined to be the rank of its root. Some properties of our adversary are embodied in the following lemmas. They assume that trees have been formed by linking in accordance with the rank rule.

LEMMA 1. *A tree of rank k has at most 3^k nodes.*

PROOF. Let s_k denote the maximum possible size of a tree of rank k. A tree of this size is formed by starting with a maximal size tree of rank $k - 1$, adding (with an incremental linking) a maximal size subtree of rank $k - 1$, and then, assuming $k \geq d + 1$, adding 3^d maximal size subtrees of rank $k - d - 1$ (this being the maximum allowable rank for nonincremental linkings). Thus, we obtain $s_k \leq 2s_{k-1} + 3^d s_{k-d-1}$. The conclusion follows by induction. □

LEMMA 2. *Let v be a node that has rank $\geq k$ and exactly $C \cdot k$ children, where $C \leq (3^d + 1)/2$. Then v has at least $k/2$ efficiently linked children, each having rank $< k$.*

PROOF. Let u be the number of children joined to v with default incremental linkings, so that the number of efficiently linked children with rank $< k$ is at least $k - u$. Each default incremental linking accounts for $3^d + 1$ children of v (the linking itself and the immediately preceding 3^d nonincremental linkings). Thus, $u \leq C \cdot k/(3^d + 1) \leq k/2$. The lemma follows immediately. □

LEMMA 3. *Among the efficiently linked children of a node at most $d + 1$ are of the same rank.*

PROOF. This follows as a consequence of the facts that an efficient linking increases the rank of the parent node, and that the rank of the child is within d of the rank of the parent at the moment immediately preceding its linking. □

We now complete our description of O-sequences. The individual rounds are structured as follows: At the onset of a given round, let ζ be the root in the forest

having maximal rank (the rightmost such node if there is more than one). Our adversary selects the node ζ to be deleted during this round, and ζ is referred to as the *designated minimum root* as the round gets underway. The decrease-key operations of this round are performed on those children of ζ that, at the onset of the round, are efficiently linked to ζ and have rank $< L$. If there are more than L such nodes, then the decrease-key operations are confined to the rightmost L of them. The order in which these decrease-key operations take place is random (with uniform distribution). The adversary obeys the convention that any linking operation involving ζ has an outcome consistent with ζ being the minimum root node, overriding the rank rule as necessary. Our analysis will estimate the expected execution cost of an O-sequence. We emphasize that the criteria under which a given sequence of heap operations constitutes an O-sequence, are intrinsically tied to the understanding that our adversary is invoked during the execution of the sequence.

We claim that the heap structure, as it appears between operations of an O-sequence, is uniquely determined by the preceding operations of the sequence. Moreover, the sequence of linkings that take place during the execution of an O-sequence is uniquely determined by the O-sequence. This is established by induction on the number of preceding operations of the sequence, using the assumptions concerning the functioning of a generalized pairing heap and the fact that our adversary is invoked. (It is important to recognize that our adversary is incompletely defined relative to the manner in which this notion is typically understood. Specifically, our adversary does not explicitly address comparisons that do not result in linkings. Our claim holds, in essence, because these non-linking comparisons are devoid of structural side-effects during the execution of an O-sequence by a generalized pairing heap. "Implicit" data structuring is in effect precluded.)

We demonstrate that our adversary can be invoked without violation of consistency. This is accomplished by proving the following stronger assertion: Let σ be an arbitrary O-sequence, let σ_j consist of the initial portion of σ ending with the jth round of σ, and let σ_0 consist of the first n insertions of σ (the operations preceding the first round of σ). Now let S_j be the set of nodes in the heap upon executing σ_j, and let ρ_j be an arbitrary linear ordering of these nodes that satisfies the heap-order condition for the tree structures in the heap that emerge from this execution. Then for each j there exist key values for the decrease-key and insertion operations of σ_j that are both consistent with the actions of our adversary, and moreover, induce the linear ordering ρ_j of S_j upon executing σ_j. We prove this by induction on j. Upon completing σ_0, the heap consists of singleton node trees, and indeed, key values can be assigned to these nodes as they are inserted that induce any specified ordering of these nodes. Now assume that our assertion holds when $j = k$, and that the nodes in S_k are given by $y_1, \ldots, y_n$. Consider the situation upon executing σ_{k+1}, at which point the nodes in S_{k+1} are given by $y_2, \ldots, y_{n+1}$ (say); the node y_1 having been deleted and the node y_{n+1} inserted. The tree structures of the heap at this point are obtained from those present, upon executing σ_k, by removing the root y_1, adding the singleton node tree consisting of y_{n+1}, and then performing just those linkings that take place during the execution of the last round of σ_{k+1} that occur among the roots of the remaining trees and the children of y_1 (but omitting those that involve y_1). (We are using here the fact that the decrease-key operations of the

$(k + 1)$st round are confined to the children of y_1.) Let L_{k+1} denote the linkings just described. Now let ρ be an arbitrary linear ordering of the nodes in S_{k+1} that satisfies the heap-order condition relative to these tree structures, and extend this to an ordering ρ' of the nodes $y_1, \ldots, y_{n+1}$, imposing the condition that y_1 is the least among these nodes. The restriction ρ'' of ρ' to the nodes in S_k satisfies the heap-order condition, relative to the tree structures present upon executing σ_k, since the linkings in L_{k+1} only involve nodes that form an anti-chain with respect to these tree structures, and the root y_1 is not among these nodes. Thus, by providing key values for the insertion and decrease-key operations of σ_k to induce the ordering ρ'' (which our induction hypothesis enables), assigning a key value to the node y_{n+1} consistent with ρ' when this node is inserted during the $(k + 1)$st round, and performing the decrease-key operations of the $(k + 1)$st round so as to not violate the ordering ρ', we find that the operations of the $(k + 1)$st round result in the deletion of y_1, and moreover, the ordering ρ is induced upon completion of the $(k + 1)$st round.

Armed with the above observations, we shall adopt the point of view that the execution of an O-sequence is only a skeletal computation that performs structural manipulations that are being governed by our adversary. With this in mind we define the actual cost of a deletemin execution to be the maximum of (i) the number of node linkings that take place, and (ii) the number of trees in the forest when the execution commences less one. We note that this measure provides a lower bound for the cost of a corresponding execution with actual key values that realize the actions of the adversary.

The following observations will prove useful. Because decrease-key operations are applied only to children of the designated minimum root (in an O-sequence), a given node in the heap can only accumulate children until it finally becomes the designated minimum root. Thus, until a node becomes the designated minimum root, its rank can only increase over time. Moreover, because only root nodes can be linked during the manipulations of generalized pairing heaps, the subtree rooted at a given node which has a parent remains fixed (in structure) until that parent becomes the designated minimum root.

We will find it useful to utilize a particular scheme for generating random O-sequences. Let $\omega = \omega_1, \omega_2, \ldots$ be a random source expressed as a sequence of i.i.d. random variables uniformly distributed over the integers from 1 to $L!$. An O-sequence is generated from ω as follows. Assume that at the onset of the ith round of the O-sequence being generated there are $q \leq L$ children of the designated minimum root upon which decrease-key operations are to be performed. The order in which these decrease-key operations are to take place is given by $\omega_i \bmod q!$, where it is understood that this quantity designates a particular permutation from some specified enumeration of the $q!$ possible permutations; the identity permutation corresponding to a left-to-right processing of these q nodes. We readily observe that this scheme defines a uniform distribution on the possible orders of the decrease-key operations of each round. The following additional property of this scheme will meet a subsequent need. *Suppose that the source ω from which the O-sequence is generated is fixed beyond round $i - 1$, and suppose two distinct prefixes are given that result in the same respective number q of nodes to be acted upon by the decrease-key operations of the ith round. Then the permutation choice to be applied for this ith round will be the same in both instances.*

2.2. SCHEDULE OF CHARGES. During the execution of an O-sequence, an *efficient event* for a node, other than the designated minimum root, is said to take place either when (i) the node becomes a child of another node, other than the designated minimum root, with an efficient linking; or (ii) when a new child becomes linked to the node, causing its rank to increase from $L - 1$ to L. An efficient event of the form (ii) is referred to as a *terminating* event. Once a node has had a terminating event, it cannot participate in a subsequent decrease-key operation (in an O-sequence) since its rank is then $\geq L$.

At the onset of each round of an O-sequence, we associate with each node to be acted upon by the decrease-key operations of the round (determined at the onset) its individual *schedule of charges* that accounts for certain costs being charged to the node. The schedule of charges specifies for each possible continuation of the O-sequence a *charge* ℓ which is defined in terms of three cases. If the node has a subsequent efficient event during the specified O-sequence continuation, then ℓ is given by the number of subsequent linkings involving that node, up to (and including) its next efficient event.

A given continuation of the O-sequence may not result in a subsequent efficient event for our node, and we consider two additional cases, depending on whether or not the node remains in the heap upon termination of the O-sequence. If the node gets removed from the heap in a subsequent deletemin operation, we say that the node experiences an *early deletion* and ℓ is defined to be the actual cost of this deletemin operation. Finally, if the node remains in the heap upon termination of the O-sequence, then, for the convenience of our analysis, a *stipulated* charge is assigned (in the sequel) that has no connection with actual manipulations that take place.

For a given O-sequence, we claim that between two successive decrease-key operations acting upon a node x, there must occur an efficient event for x; a fact which we refer to as the *separation property*. This follows from the requirement that x must be efficiently linked to the designated minimum root at the onset of the round in which each decrease-key operation takes place. The associated linking thus takes place *prior* to when the new parent has become the designated minimum root, and thus constitutes an efficient event for our node.

The following lemma establishes a relationship between the total execution cost of an O-sequence and the schedules of charges. Given an O-sequence σ and a decrease-key operation included in σ, acting upon a node x, we let $\ell_{\sigma,x}$ denote the charge for the unique continuation compatible with σ, contained in the schedule of charges associated with x at the onset of the round of σ in which the decrease-key operation takes place.

LEMMA 4. *Given an O-sequence* σ, *let* $\Sigma\ell$ *denote the sum of the charges* $\ell_{\sigma,x}$, *where the sum extends over all decrease-key operations included in* σ. *Let* u_σ *denote the sum of all stipulated charges associated with nodes remaining in the heap at the termination of* σ (*the sum of those terms in* $\Sigma\ell$ *such that* $\ell_{\sigma,x}$ *is a stipulated charge*), *and let C denote the total execution cost of* σ. *Then* $C \geq \frac{1}{3}(\Sigma\ell - u_\sigma)$.

PROOF. The quantity $Q = (\Sigma\ell) - u_\sigma$ is given by $\Sigma\ell^{(1)} + \Sigma\ell^{(2)}$, where the first term sums the charges that reflect the linkings that precede efficient events, and the second term sums the charges that reflect early deletions. The separation property implies that the actual cost of a given deletemin operation contributes at most one term to the sum $\Sigma\ell^{(2)}$. Thus, $C \geq \Sigma\ell^{(2)}$. The separation property

also implies that a given linking contributes at most two to the sum $\Sigma\ell^{(1)}$ (potentially contributing one each to the charges to the two nodes involved in the linking), and thus $2C \geq \Sigma\ell^{(1)}$. The lemma follows by combining these two inequalities. □

LEMMA 5. *Given a node x and a continuation for which x experiences an early deletion, the corresponding entry ℓ in the schedule of charges for x satisfies $\ell \geq \sqrt{n}$.*

PROOF. At the onset of the round in which x is deleted its rank is at most $L - 1$ (otherwise, x would have had a terminating event), and by the construction of our adversary, all roots in the heap have rank at most $L - 1$ at this point. By Lemma 1, no tree in the forest has more than 3^{L-1} nodes, and the forest therefore has at least $n/3^{L-1}$ trees. Including consideration of the node inserted into the heap in this round, we conclude that $\ell \geq n/3^{L-1} \geq \sqrt{n}$. (Recall $L = \lceil \frac{1}{2}\log_3 n \rceil$.) □

2.3. AN OVERVIEW. Imagine that our data structure is operating efficiently; that despite the presence of the decrease-key operations, the current *actual* costs of the deletemin operations are typically only $O(\log n)$ per operation. We would thus find that the forest of the data structure typically has only $O(\log n)$ trees, and moreover, that tree roots typically have only $O(\log n)$ children. Under these circumstances, since the rank of a node grows at least logarithmically with the size of its subtree (Lemma 1), some tree root in the forest must have rank at least $\Omega(\log n)$, and therefore the rank of this tree root is comparable to the number of children it has. But then Lemma 2 implies that a positive fraction of the children of this root are efficiently linked to it, and moreover, participate in the decrease-key operations of the round during which this root is deleted. Now Lemma 3 implies that the ranks of these efficiently linked children are reasonably scattered. Therefore, as a consequence of the subsequent random placement of these nodes (due to the random ordering of the decrease-key operations), we expect that each such node typically winds up in a *context* in which it must participate in many subsequent linkings before encountering a linking partner having similar rank. Those subsequent linkings that occur prior to this encounter are inefficient. (This is where *charge* enters the picture.) In other words, when our data structure is operating efficiently, seeds are inexorably being sewn that give rise to a high frequency of inefficient linkings (per decrease-key operation), inevitably slowing it down. Our task now centers upon the matter of charge estimation.

The above scenario would seem equally plausible, if, as an alternative, we were to define the rank of a node to be simply the logarithm of its subtree size, and consider a linking to be efficient when the ranks of the involved nodes are appropriately close. The difficulty with this more natural approach is that *node context*, as considered above, seems to be an elusive notion. We avoid having to explicitly grapple with this notion by exploiting a particular property enjoyed by our original rank function:

(**) The rank of a node is uniquely determined by the efficiencies of the linkings to its children.

2.4. 8-WAY LINKING-DECISION TREE. At the onset of a given round of an O-sequence, round η (say), let $v_1, \ldots, v_p$ be among the nodes upon which

decrease-key operations are scheduled to take place during this round. Fix the order (in which they are operated upon) of all the nodes participating in the decrease-key operations of round η, but leave unspecified the placement of these p nodes apart from specifying the set of their p positions within this order. There are $p!$ possible placements of these nodes, and each placement determines the positions where the nodes $v_1, \ldots, v_p$ arrive as new roots in the forest (as it appears just prior to the next deletemin operation), within a fixed set of p positions. Similarly, the other tree roots of the forest occupy fixed positions independently of these v_i placements. Number these p particular positions from 1 to p, and let U_i be a variable whose value is the node v_j arriving in position i. Now fix the random source from which the O-sequence is generated beyond round η, and let R denote the restrictions we have defined; the O-sequence is completely determined from R if we additionally specify the assignment of the v_j's to the variables U_i. Consider the moment that the next efficient event takes place for one of these v_i. Subsequent to this moment we say that v_i is *post-efficient*, and up to this moment we say that v_i is *pre-efficient*. The number of linkings involving v_i while it is pre-efficient represents the entry in the associated schedule of charges for v_i corresponding to the specified O-sequence continuation, assuming that v_i has a subsequent efficient event. In the sequel, we bound the expected total of the charges to the v_i's, averaging over the $p!$ possible assignments to the U_i's.

We need the following notation. Given heap nodes x_1 and x_2, the expression, link(x_1,x_2), denotes the operation that links x_1 to x_2 with x_1 as the leftmost child of x_2. The expression b-link(x_1,x_2) represents a binary decision for the linking of x_1 and x_2; the outcome is either link(x_1,x_2) or link(x_2,x_1). Given a node in the heap, its *position in the heap* refers to its position in the tree that contains the node *and* the position of that tree in the forest of the heap.

Assume restrictions R are in effect and let α be a specified assignment of the v_j's to the variables U_i. As the subsequent operations of the specified O-sequence take place, starting with the deletemin operation of round η, their combined execution can be expressed as a directed path consisting of linking-decision nodes, b-link(x_1,x_2), with adjoining edges indicating the actual outcomes of these decisions (as dictated by the adversary), where the operands x_1 and x_2 may consist of:

(a) a variable U_i,

(b) a node which is a proper descendant of some v_j node at the onset of round η, specified as being in a designated position (at the onset of round η) of the subtree rooted at the value of a designated variable U_i (e.g., the node which is the second child of the third child of U_2's assigned value, at the onset of round η),

(c) a node which is in the heap at the onset of round η, but not in the subtree of any v_i node, specified by its position in the heap at the onset of round η,

(d) a node specified as being the jth node inserted since the onset of round η, for some $j \geq 1$.

b-link nodes reflecting linkings that involve the designated minimum root (as determined for the round during which the linking takes place) are omitted from this path. Thus, the outcome of any linking on this path is determined by the

ranks of the nodes involved. Provided that the ranks of the v_i's are suitably spaced, our goal is to utilize these execution paths to construct an encoding scheme for assignments α belonging to an appropriate subset of the possible assignments.

To achieve this encoding, we *augment* the execution path corresponding to a given assignment α as follows. Define the expression 8-link(x,y), where x and y are of the form (a)–(d) above, to denote an *eight-fold* decision for the linking of x and y: the resulting outcome includes the result of the corresponding b-link, an indicator specifying whether or not the linking is efficient, and an indicator specifying whether or not the linking constitutes a terminating event for the node that becomes the parent: eight possible outcomes in all. Now for each node b-link(x,y), we replace the b-link with the corresponding 8-link and augment the outgoing edge of this node to indicate which of the eight possible outcomes apply for this linking. Let π_α denote the resulting augmented path.

For a fixed assignment α, each node in the heap at the onset of round η, or subsequently inserted, has a unique description in terms of (a)–(d) above. We refer to this description as the *specification* of the node (relative to the assignment α); by definition, it remains fixed even as subsequent heap operations take place.

DISCUSSION. Assume that the ranks of the v_i's are separated by large gaps, and for the moment assume that node ranks are static entities. Also assume that each v_i eventually has an efficient event. When the (unknown) assigned value of a specified variable U_i has an efficient event, the constraint on node rank required in order for this to happen uniquely determines the particular value v_j assigned to U_i. Thus, our data structure, operating in conjunction with the adversary, is effectively solving the problem of determining the assignment of the v_j's to the U_i's. Using the facts that node ranks are static (assumed for now) and that our adversary makes its decisions on the basis of rank, and appealing to the decision-tree complexity of this *assignment determination problem*, we are tempted to conclude that a super-linear number of linkings are required, *even if we are only counting the linkings that involve at least one U_i whose assigned value hasn't yet been deduced* (*thus remaining pre-efficient*). We thereby infer a nonconstant lower bound on the average charge to the v_i's.

Now there are difficulties with this approach as it applies to our situation: (i) the node ranks change as linkings take place, and (ii) there is the possibility that an "unseen hand" might simply present to each U_i a linking in which its assigned value has an efficient event, effectively solving our problem non-deterministically. (This is what Fibonacci heaps do!) In disposing of this latter concern (Lemma 7, below), we exploit the uniformity characteristic of a self-adjusting data structure, incorporated within our definition of generalized pairing heap. As for the first concern, we can assume that the ranks of the v_i's change little while these nodes remain pre-efficient, since many linkings involving a node are required to substantially change its rank. But there are two interesting issues in connection with the non-v_i and post-efficient v_i nodes. First, does our device actually solve the assignment determination problem? Even if the unknown assigned value of U_i has an efficient linking, becoming a child of x (say), we cannot deduce its approximate rank if the rank of x has been changing, unless we have been monitoring these changes. Our encoding scheme, which includes all

(relevant) linkings that take place, enables this monitoring (part (a) of Lemma 6) due to the added presence of the efficiency indicators. (Property (**) implies that the efficiencies of the linkings constitute the sole source of variability in rank changes as the linkings take place.)

The second issue centers on the possibility that information about U_i's assigned value is being "leaked" when it becomes linked to another node x, as reflected by the change (or absence of change) in the rank of x. Since operations among the non-v_i and post-efficient v_i nodes are "free" in terms our cost measure (charge), and because these nodes can potentially carry information about the assigned values of the U_i's as a consequence of this information leakage, our lower bound is conceivably at risk. However, our encoding scheme effectively discounts the potential impact of leakage (part (b) of Lemma 6); the efficiency indicators fully account for it.

LEMMA 6. *Assume that the minimum gap between the respective ranks of the nodes $v_1, \ldots, v_p$ is g with $g > d$. Let A be a set of assignments α such that in the course of the execution π_α, no v_i node has its rank increase by as much as $g - d$ while remaining pre-efficient, and such that no v_i experiences an early deletion from the heap. Let α_1 and α_2 be two assignments in A. Suppose that the first k nodes and decision outcomes of the execution paths π_{α_1} and π_{α_2} are identical for some $k \geq 0$. Then*

(*a*) *Let Γ_k consist of the variables U_i whose assigned values v_j are post-efficient following the kth linking. With respect to both assignments α_1 and α_2, the set Γ_k is the same, and the respective assignments to the variables in Γ_k are identical.*

(*b*) *Suppose that the $(k+1)$st nodes of π_{α_1} and π_{α_2} are identical and neither of the operands of this node are U_i variables whose assigned values are pre-efficient at this point of the computation. (In other words, U_i can be an operand of this $(k+1)$st node only if $U_i \in \Gamma_k$.) Then the respective outcomes for the $(k+1)$st node on the paths π_{α_1} and π_{α_2} are identical.*

PROOF. We first establish that for assignments α in A, a node x, that has some v_h as a proper ancestor (at the onset of round η), can participate in a linking only if this v_h ancestor has had a prior efficient event. In order for x to be participating in a linking it must be the case that the v_h ancestor of x has either been previously deleted, or has become the designated minimum root (and a decrease-key operation has since been performed on x). Since by assumption v_h does not experience an early deletion for assignments in A, we conclude in either case that v_h has had an efficient event prior to the linking in question involving x.

We next establish the following: Let x be a node in the heap and assume that with respect to both assignments, α_1 and α_2, x has the same specification. Consider a point in time along the common portion of the paths π_{α_1} and π_{α_2} such that for both assignments, α_1 and α_2, x is in the heap and not the designated minimum root. Then, at this point in time, the rank of x is independent of the applicable assignment, α_1 or α_2. First, we note that the children of x, at the specified point in time, consist of those that were originally present and those subsequently linked to x as indicated by the computation path. (Because x is not the designated minimum root, all of the linkings to x of these subsequently linked children are accounted for by the computation path. Moreover, all nodes that

were originally or subsequently linked to x remain linked to x.) Now the rank of x is uniquely determined by the efficiencies of the linkings to these subsequently linked children (provided by the computation path), along with the ranks of the children that were initially present, which remain fixed. We conclude that the rank of x is fixed, independently of the applicable assignment, α_1 or α_2. Note that in applying this claim to a node specified by an operand of a linking-decision node, the requirement that the specified node is not the designated minimum root is satisfied at the corresponding point in time (since the execution paths π_α omit linkings involving the designated minimum root).

We proceed to establish (a). The 8-link outcomes on the common portion of the paths π_{α_1} and π_{α_2} uniquely determine the occurrences, if any, in which the assigned value of a U_i variable has an efficient event. We refer to the first such occurrence, for a given U_i variable, as its *critical* event; up to this point U_i's assigned value remains pre-efficient. In particular, this establishes the uniqueness of Γ_k.

Next, we demonstrate that the respective assignments to the variables in Γ_k are fixed, independently of α_1 and α_2. We refer to the assumptions concerning the minimum gap between the (initial) ranks of the v_h's and the limit on the extent to which these ranks can grow, while the nodes remain pre-efficient, as the *rank constraints*. Suppose first that the critical event for U_i's assigned value is a terminating event. In order to have a terminating event, a node must have rank $L - 1$. Since our v_h nodes initially have rank $< L$, the rank constraints imply that only one can gain sufficient rank for this to happen while remaining pre-efficient; namely, the one having largest rank. Thus, there can be only one U_i to which this discussion applies, and its value is fixed. Next, assume that the critical event for U_i's assigned value is indicated by the outcome of an 8-link node with operands U_i and x, in which U_i's assigned value becomes the child of x with an efficient linking. The operand x cannot have the form U_j since the rank constraints keep the ranks of the respectively assigned values of U_i and U_j too far apart for the linking to be efficient. We can now argue that the node specified by x is independent of the applicable assignment, α_1 or α_2. This is obvious unless x is specified as being a proper descendant of the value assigned to some variable U_j. Considering this possibility, since x is participating in a linking, the v_h ancestor of x assigned to U_j has had a prior efficient event, from which it follows that U_j belongs to Γ_{k-1}. By induction on k, this uniquely identifies v_h, from which it follows that x is uniquely identified. As previously established, it now follows that the rank of the node specified by x is independent of the applicable assignment, α_1 or α_2. Now the rank of U_i's assigned value is within d of the rank of the node specified by x since the linking taking place is efficient. The rank constraints therefore serve to fix the assigned value of U_i, independently of the applicable assignment α_1 or α_2. This completes the proof of part (a).

We turn next to part (b). Consider the $(k + 1)$st node of our common portion of the paths π_{α_1} and π_{α_2}. Unless an operand x of this linking-decision node is a U_i variable, the node specified by x is uniquely determined, as argued in the proof of part (a) (invoking here the fact that the variables in Γ_k are uniquely assigned). On the other hand, if x is a U_i variable, then by assumption, this variable belongs to Γ_k, and it likewise follows that the node specified by x is uniquely determined. Since x specifies a unique node it follows, as previously established, that the rank of this node is independent of the applicable assignment α_1 or α_2. Moreover, for

each operand, the efficiencies of the linkings to the children of the specified node are independent of the applicable assignment, α_1 or α_2. Now the outcome of the linking decision (which node becomes the parent) is dictated strictly by the ranks of these operands since our adversary is defined this way, the efficient linking indicator is uniquely determined by these ranks, and the terminating event indicator for the node becoming the parent is uniquely determined by the efficiencies of the linkings to its current children and the rank of the new child (the other operand). Thus, the outcome of this 8-link node is independent of the applicable assignment, α_1 or α_2. □

The following corollary is established in our proof of Lemma 6.

COROLLARY 1. *Suppose that A, α_1, α_2, π_{α_1}, π_{α_2}, and the v_i's are as in Lemma* 6. *Let x be a node in the heap and assume that with respect to both assignments, α_1 and α_2, x has the same specification. Consider a point in time along the common portion of the paths π_{α_1} and π_{α_2} such that for both assignments, α_1 and α_2, x is in the heap and not the designated minimum root. Then, at this point in time, the rank of x is independent of the applicable assignment, α_1 or α_2.*

LEMMA 7. *Assume that A and the v_i's are as in Lemma* 6. *Let π be an execution path such that $\pi = \pi_\alpha$ for some unspecified assignment α in A. Then the content of each node on the execution path π, and the point of termination of π, are uniquely determined by the prior nodes and decision outcomes on this path.*

PROOF. First, some terminology. A node specification is said to be *unambiguous* at a given point along the execution path π provided that the node that it specifies is uniquely determined by the prior linking-decision nodes and outcomes along the execution path. The following are consequences of Lemma 6 and our assumption that none of the v_i's experience an early deletion for assignments in A. First, a node specification, corresponding to a node which happens to be a tree root at a given point along π, is unambiguous at that point, unless the specification is given as a U_i variable whose assigned value remains pre-efficient. Second, given a point along the computation path π and a node specification that is unambiguous at that point and that doesn't refer to the designated minimum root, the number of children of this uniquely specified node x (say) is fixed, and moreover, these children have specifications that are uniquely determined as a function of relative position. (The execution path π uniquely determines the specifications of the children linked to x subsequent to the onset of this execution.) Furthermore, the positions of the efficiently linked children of x are uniquely determined and their specifications are unambiguous at this point. (Those children efficiently linked to x during the computation π either have unambiguous specifications prior to this linking, or, for the case in which the child's specification is given as a U_i variable, its assigned value is post-efficient following this linking.)

The heap structure is said to be *semidetermined* at a specified point of the computation provided that the following hold, given the linking-decision nodes and outcomes along the execution path π that precede this point:

(i) the forest state of the heap is uniquely determined,
(ii) the specification of each tree root in the forest is uniquely determined by its position in the forest, and

(iii) the position and identity of the designated minimum root are uniquely determined, and the number of children it has, and their specifications by relative position, are uniquely determined.

Now we argue by induction on k, for k not exceeding the number m of remaining deletemin operations of the O-sequence encompassed by the computation π, that upon partitioning the path π into $k + 1$ segments, such that each of the first k segments corresponds to the execution of a single deletemin operation, we find that (a) the content of each node of π appearing within these first k segments is uniquely determined by the prior linking-decision nodes and decision outcomes on this path; (b) the point of termination of the kth segment is uniquely determined by the prior nodes and decision outcomes on the path; and (c) if $k < m$, then the heap structure is semidetermined at the point immediately preceding the execution of the first deletemin operation of the $(k + 1)$st segment. (When k reaches its maximum value m, the $(k + 1)$st segment is empty.)

Consider first the case $k = 0$. The heap structure and the designated minimum root are uniquely determined at the onset of the round that includes the first deletemin operation encompassed by the computation π. Also, the ordering of the decrease-key operations that are performed in this round is such that the above conditions (ii) and (iii) are satisfied at the point immediately prior to the first deletemin operation. Moreover, the constraints (a) and (b) satisfied by a generalized pairing heap imply that the forest state is uniquely determined at this point, implying that the heap structure is semi-determined immediately preceding the first deletemin operation. The conditions (a) and (b) of our claim are vacuous when $k = 0$, and the claim is thus established in this instance.

Now assume that the claim holds for $k = j$ with $j < m$. Just as the $(j + 1)$st deletemin execution gets underway, the conditions required for the heap structure to be considered semi-determined, in particular conditions (i) and (iii), imply that the binary linking-decision tree that models the execution of this deletemin operation, defined in the constraint (c) satisfied by a generalized pairing heap, is uniquely determined in the sense conveyed by the constraint (d). Now all of the nodes participating in the linkings of the ensuing deletemin execution are either tree roots or children of the designated minimum root, and since the specifications of these nodes are uniquely determined as a function of position (respectively, relative position), as indicated by conditions (ii) and (iii), it follows that we can uniquely substitute node specifications into our binary linking-decision tree. Thus, the content of each node of π and the point of completion of the $(j + 1)$st deletemin operation are uniquely determined by the preceding nodes and outcomes along π during this phase of the computation. It follows that condition (ii) is satisfied upon completion of the execution of this deletemin operation, and moreover, constraint (e) implies that the condition (i) is also satisfied.

Now assuming that there is a subsequent round in the O-sequence ($j + 1 < m$), the designated minimum root for this subsequent round is a node whose specification is *not* given as a U_i variable, where the assigned value of U_i is pre-efficient, since none of the v_i's experience an early deletion. Thus, it is a node whose specification is unambiguous at this point. It follows that the designated minimum root and its position (the rightmost root having maximal

rank) are uniquely determined for this subsequent round since the above corollary implies that the ranks of all of the potentially eligible roots are uniquely determined. Moreover, the above corollary implies that the efficiently linked children of the designated minimum root for this subsequent round (all of which have unambiguous specifications), that participate in the decrease-key operations of this subsequent round, are uniquely determined (the rightmost L that have rank $< L$). The order in which these decrease-key operations take place is uniquely determined since this is specified by the restrictions R. (Here is where we make use of our particular scheme for generating random O-sequences.) Condition (ii) thus remains satisfied just prior to the execution of the deletemin operation of this subsequent round, and also the position of the designated minimum root remains uniquely determined. Constraints (a) and (b) also imply that condition (i) remains satisfied at this point. Moreover, the specifications of the remaining children of the designated minimum root (those not having been removed by the decrease-key operations) are uniquely determined by relative position. Thus, condition (iii) is satisfied just prior to the execution of the deletemin operation of this subsequent round, and we conclude that the heap structure is semi-determined at this point. Thus, our claim is established for $k = j + 1$. □

Remark 1. Lemma 7 represents the crucial point of departure between generalized pairing heaps and Fibonacci heaps, upon which our analysis rests.

Assume that the v_i's and the set of assignments A are as described in Lemma 6. Assume additionally that for each assignment in A each v_i eventually has an efficient event. We now proceed to embed the paths π_α for $\alpha \in A$ within an 8-way linking-decision tree τ that satisfies the following properties:

(i) The set of paths π_α for $\alpha \in A$ and the set of paths in τ terminating at its leaves are the same. Moreover, τ has $|A|$ distinct leaves.
(ii) A node of τ can have two or more children only if it contains an operand consisting of some U_i variable whose assigned value is pre-efficient (as determined by the path leading to the node).

The construction commences by setting τ to consist of a single leaf. We proceed to build τ, replacing a leaf with a node at each step. Our inductive hypothesis is that after each step the following assertion (H) holds: *For each assignment* $\alpha \in A$, *some initial portion of* π_α (*linking-decision nodes and their outcomes*) *coincides with a path in* τ *ending at a leaf. Conversely, the path through* τ *terminating at a given leaf coincides with an initial portion of* π_α *for some* $\alpha \in A$.

For each leaf ℓ of τ, as constructed thus far, let A_ℓ denote the nonempty subset of $\alpha \in A$ associated with ℓ in accordance with (H). Lemma 7 implies that either (a) each π_α for $\alpha \in A_\ell$ terminates at ℓ, or (b) these paths continue with identical successor nodes. If (b) holds let δ be the common subsequent node on these paths. We then replace ℓ with δ. The leaves connected to δ correspond to the realizable outcomes as reflected by the paths π_α for $\alpha \in A_\ell$. Clearly (H) holds after each step. The construction of τ terminates when (a) holds for each leaf. Part (a) of Lemma 6 implies that for each leaf ℓ of our completed tree we have $|A_\ell| = 1$ (since each v_i is post-efficient when π_α terminates, by assumption). Thus, τ has $|A|$ distinct leaves, establishing property (i).

Now consider any node v in τ that involves the linking of two heap nodes, neither of which is represented by some U_i variable whose assigned value remains pre-efficient. Lemma 6, part (b) implies that v has only one child since only one outcome of the linking associated with v can be realized. Thus, property (ii) holds.

Property (ii) implies that the number of branch points on a given path ρ of τ constitutes a lower bound for the total number of linkings that involve at least one pre-efficient v_i, for the assignment corresponding to ρ. The following lemma summarizes our construction:

LEMMA 8. *Assume that restrictions R apply and that the minimum gap between the respective ranks of the nodes* $v_1, \ldots, v_p$ *is* g *with* $g > d$. *Let* A *be a set of assignments* α *such that in the course of the execution* π_α, *no* v_i *node has its rank increase by as much as* $g - d$ *while remaining pre-efficient, and such that each* v_i *eventually has an efficient event. Then the executions* π_α *for* $\alpha \in A$ *can be embedded within an 8-way linking-decision tree* τ, *satisfying the two properties stated above. For a given assignment* α, *the number of branch points on the path in* τ *coinciding with* π_α *provides a lower bound for the total number of linkings that involve at least one pre-efficient* v_i.

LEMMA 9. *Assume that restrictions R apply and that the minimum gap g between the ranks of the nodes* $v_1, \ldots, v_p$ *satisfies* $g - d \geq p^{4/3}$. *Assume that we stipulate a charge* $\geq \lceil p^{4/3} \rceil$ *for each* v_i *that does not have a subsequent efficient event and remains in the heap at the termination of the O-sequence corresponding to a given assignment. Then upon averaging over the* $p!$ *possible assignments, the expected total of the charges to these p nodes, as specified in their associated schedules at the onset of round* η, *is bounded by* $\Omega(p \log p)$.

PROOF. An assignment α of the v_j's to the U_i's is said to have *jumps* provided that one or more of the v_j's has a gain of at least $p^{4/3}$ in rank while remaining pre-efficient, during the execution of the corresponding O-sequence continuation. Consider an assignment for which each v_i has a subsequent efficient event. If additionally this assignment has jumps, then it results in a charge of at least $p^{4/3}$ for at least one of the v_i, since a single linking can increase the rank of a node by only one unit. Let A consist of the assignments for which no v_i has an associated charge as large as $p^{4/3}$. Without loss of generality, we may assume that at least one-half of the $p!$ assignments belong to A. For each assignment in A, no v_i experiences an early deletion; otherwise, Lemma 5 implies that its associated charge would be $\geq p^{4/3}$ since $p \leq L$. For each assignment in A, each v_i has a subsequent efficient event; an exceptional v_i would have a stipulated charge $\geq p^{4/3}$. No assignment in A has jumps. Therefore A satisfies the hypothesis of Lemma 8 since $g - d \geq p^{4/3}$. We apply Lemma 8 to the set A, obtaining the 8-way tree τ which embeds the executions π_α for the assignments $\alpha \in A$, and which has $|A|$ leaves. The average number of branch points on the paths of τ bounds from below the total of the charges to the p nodes averaged over the assignments in A. Upon collapsing all links in τ for which the parent node has just one child, we conclude that this average number of branch points is given by the average path length in this transformed tree. This latter quantity is at least $\log_8 |A| = \Omega(p \log p)$. The conclusion of the lemma follows immediately. □

2.5. Summing

LEMMA 10. *Assume that at the onset of a given round there are q nodes to be acted upon by the decrease-key operations of that round, where $q^{1/4} \geq 4(d+1)$. Assume that we stipulate a charge of $\lceil q^{1/3} \rceil$ for each of these selected nodes that does not have a subsequent efficient event and remains in the heap at the termination of the corresponding O-sequence continuation. Then for a random continuation of the O-sequence the expected total of the charges to these q nodes, as specified in their associated schedules at the onset of this round, is bounded by $\Omega(q \log q)$.*

PROOF. Let $p = \lfloor q^{1/4} \rfloor$. We begin by constructing disjoint subsets of the q nodes, with each subset having size p, such that the union of these subsets consists of $\geq q/2$ nodes, and such that the ranks of the nodes in each subset are separated by gaps of at least $2p^2$. These sets are constructed inductively as follows: Suppose that at least $2(d+1)p^3$ nodes among the initial q nodes remain, not having been placed in the subsets thus far constructed. By Lemma 3, there are at most $d+1$ nodes of any given rank among the q nodes. Thus, the ranks of the remaining nodes comprise $\geq 2p^3$ distinct values, and it follows that we can select a subset of p nodes from these remaining nodes whose ranks are separated by gaps of at least $2p^2$. When this process finally stops, it is because there are fewer than $2(d+1)p^3$ remaining nodes, and our assumption implies that this quantity is $\leq q/2$. Let $g = 2p^2$ denote our guaranteed gap between ranks. Then, $g - d \geq p^2$, and it follows that each of the selected sets of p nodes satisfies the hypothesis of Lemma 9. Let S_j, $j \geq 1$, denote the subsets of p nodes that we have constructed. If we now fix j, then averaging over the assignments of the nodes in S_j, conditioning upon an arbitrary instantiation of the restrictions R defined at the start of our section on the 8-way linking-decision tree, we conclude from Lemma 9 that the expected total of the charges to these nodes is bounded by $\Omega(|S_j| \log |S_j|) = \Omega(|S_j| \log q)$, provided that we stipulate a charge of $\lceil q^{1/3} \rceil \geq \lceil p^{4/3} \rceil$ for each node of S_j that fails to have a subsequent efficient event, remaining in the heap at the termination of the O-sequence. Thus, the unconditional expected total of the charges to the nodes in S_j is bounded by $\Omega(|S_j| \log q)$. Summing over j we obtain our $\Omega(q \log q)$ bound. □

THEOREM 1. *Assume $n > 9$ and choose $C = \log_3 L$ and $d = \lceil \log_3(2C) \rceil$. Using a generalized pairing heap to implement the operations, let T_r denote the total expected cost of executing a random O-sequence consisting of r rounds. Then $T_r = \Omega(r \cdot L \log L) - O(nL^{1/3})$. (Recall $L = \lceil \frac{1}{2} \log_3 n \rceil$.)*

PROOF. Given an arbitrary O-sequence and a round of this O-sequence in which (say) q decrease-key operations are to be performed (determined at the onset of the round), we choose $\lceil q^{1/3} \rceil$ as the common value for all stipulated charges associated with this round. Defined in this way, the schedules of charges associated with a given round are uniquely determined by the operations that precede the round. (This is the setting in Lemma 10.)

For a random O-sequence σ, let ℓ_i denote the sum of the charges $\ell_{\sigma,x}$, where the sum extends over the decrease-key operations that take place during the ith round of σ; let c_i denote the actual cost of the deletemin operation of the ith round, and let d_i denote the number of children of the designated minimum root at the onset of the ith round. We consider three cases relative to the onset of the

ith round of σ (reflecting the operations that precede the round). The first case assumes that $d_i \geq C \cdot L$.

The second case assumes that $d_i < C \cdot L$ and that the designated minimum root has rank $\geq L$. Lemma 2 implies that there are $q \geq L/2$ children efficiently linked to this root with ranks $< L$ that participate in the decrease-key operations of the round. The condition $q^{1/4} \geq 4(d + 1)$ of Lemma 10 trivially holds when $q \geq L/2$ (for n sufficiently large). It follows from Lemma 10 that the expected value of ℓ_i, conditioned on the assumptions that define this case, is bounded by $\Omega(L \log L)$.

The third case considers the remaining possibility, for which the designated minimum root has rank $< L$. In this case Lemma 1 implies that $c_i \geq n/3^{L-1} = \Omega(L \log L)$.

Combining the above cases and considering the unconditioned expected values of the quantities d_i, ℓ_i, and c_i, we conclude that

$$E[d_i] + E[\ell_i] + E[c_i] = \Omega(L \log L). \tag{1}$$

Each of the d_i children of the designated minimum root, as i ranges over the rounds of σ, uniquely represents a linking that has occurred during the execution of σ, and it follows that

$$\sum_i c_i \geq \sum_i d_i. \tag{2}$$

Summing over the rounds of σ and applying Lemma 4, we have

$$\sum_i c_i \geq \frac{(\Sigma_i \ell_i - u_\sigma)}{3}. \tag{3}$$

Combining (2) and (3) we conclude that

$$5 \sum_i c_i \geq \sum_i d_i + \sum_i \ell_i + \sum_i c_i - u_\sigma. \tag{4}$$

Because no node remaining in the heap at the termination of an O-sequence has a stipulated charge exceeding $\lceil L^{1/3} \rceil$, we have $u_\sigma \leq n\lceil L^{1/3} \rceil$. (The separation property implies that a given node contributes at most one charge term to the quantity u_σ.) Substituting into (4) and applying (1), we conclude that the total expected cost of executing a random O-sequence consisting of r rounds is bounded by $\Omega(r \cdot L \log L) - O(nL^{1/3})$. □

COROLLARY 2. *Given any generalized pairing heap, it is not the case that insertion and deletemin operations have $O(\log n)$ amortized costs, and decrease-key operations have constant amortized cost, where n is the number of items in the heap.*

2.6. A UNIFYING FRAMEWORK FOR PAIRING HEAPS AND FIBONACCI HEAPS. We proceed to bridge the gap separating generalized pairing heaps from Fibonacci heaps in order to determine precisely what is necessary to attain the performance bounds achieved by Fibonacci heaps. We extend the scope of our analysis to include data structures that allow for the nodes to contain balance information. The class of *generalized pairing heaps with balance fields* is defined in

the same way that generalized pairing heaps are defined, but with the following differences (again, F_1 and F_2 denote heaps, reflecting two instances of the data structure that have identical forest states, but possibly different heap structures):

—Each node in the data structure contains a dynamic auxiliary information field referred to as its *balance field*, consisting of b bits of information, where b is a parameter of this computational model. The balance field of a node is uniquely determined by its subtree, and is defined inductively as follows:

(i) The balance field of a node with no children is given by some constant c.
(ii) Let x and y be two nodes having respective balance fields b_x and b_y. Upon linking x to y as its leftmost child, the balance field of y becomes $b'_y = g(b_x, b_y)$, where g is a fixed function.

—In considering the implementation of the decrease-key operation, we do not speak to how this is generally done. Instead, we impose the restriction that if the affected node happens to be a child of the root of a tree in the forest, then, as is the case for generalized pairing heaps, we remove the subtree rooted at this node, inserting it at the end of the forest (a restriction satisfied by Fibonacci heaps). Now consider the constraint (b) from the definition of generalized pairing heaps. With the understanding that x_1 and x_2 are children of the roots in their respective trees, this constraint holds in the present context. (With respect to O-sequences, this does not represent a change.)

—We replace the constraint (d) with the following:

(d′) Suppose that a deletemin operation is performed, respectively, on F_1 and F_2, and that the minimum respective tree roots r_1 and r_2 being deleted are correspondingly positioned, respectively, within F_1 and F_2, and moreover, have identical numbers of children. Consider the respective linking-decision trees that model these operations as described in (c), but suppress all names of heap nodes within these linking-decision trees, replacing these references with their corresponding positions as tree roots (so indicated), or their corresponding relative positions among the children of the root which is being deleted (so indicated). *Provided that the correspondingly positioned tree roots and the correspondingly positioned children of r_1 and r_2 have identical balance fields*, the resulting linking-decision trees are identical.

With Fibonacci heaps, the balance field of a node, referred to as its *rank* [Fredman and Tarjan 1987] is defined to be the number of children of the node. Defining the forest state to consist of the number of trees, it is readily verified that Fibonacci heaps fall within this framework. As for generalized pairing heaps, by assigning dummy balance fields (having fixed value) to the nodes in the data structure, we find that these data structures likewise fall within this framework.

Our treatment of generalized pairing heaps with balance fields exactly parallels the preceding analysis, up to and including Lemma 6. The only difference is that the induction argument, showing that the structural manipulations that take place during the execution of an O-sequence are uniquely determined by the sequence, takes into consideration the fact that the balance field of a node is uniquely determined by its subtree. (This is necessary to justify the application of the constraint (d′) in place of the constraint (d).)

If we add to the statement of Lemma 7 the assumption that the balance fields of the respective v_i's are identical, then we find that the lemma holds true, as we now demonstrate. Referring to the original proof of Lemma 7, the application of constraint (d) takes place in a context under which the heap structure is semi-determined. However, the additional precondition, necessary to justify an application of constraint (d′), asserts that the very nodes, whose specifications by (relative) position must be uniquely determined in order for the heap structure to be considered semidetermined, must also have uniquely determined balance fields. To demonstrate that this is indeed the case, we need the following lemma.

LEMMA 11. *Assume that A and the v_i's are as in Lemma* 6, *and moreover, that the respective balance fields of the v_i's are identical at the onset of the round η in which the decrease-key operations acting upon the v_i's take place. Let π be an execution path such that $\pi = \pi_\alpha$ for some unspecified assignment α in A. Let x be node specification that specifies a node that is either* (*a*) *a tree root other than the designated minimum root, or* (*b*) *a child of another node whose specification is unambiguous, at a given point of the computation encompassed by π. Then at this point of the computation, the balance field of the node specified by x is uniquely determined by the linking-decision nodes and outcomes along the execution path π that precede this point.*

PROOF. First, we establish the result for the case (a), for which x specifies a tree root at the given point t of the computation. We argue by induction on the number k of linkings of π that precede t. We observe first that x is either given as a U_i variable or is unambiguous at this point of the computation. (This holds as a consequence of Lemma 6 and the assumption that none of the v_i's experience an early deletion for assignments in A.) The balance field of the specified node, as defined at the point immediately prior to the first linking of π, is uniquely determined, either because the specified node is one of the v_i's or because the node and its subtree are uniquely determined. (This settles the case $k = 0$.) The path π provides the specifications of all children linked to the node specified by x during the portion of the computation up to the point t of interest (since x does not specify the designated minimum root). At the moment of each such linking, the induction hypothesis implies that the balance field of the specified child is uniquely determined since the specified child is a tree root just prior to the linking. Now property (ii), satisfied by the balance fields, implies that the balance field of a node is uniquely determined, given its value at some initial point and the balance fields of its subsequently linked children. Applying this to the situation at hand, we conclude that the balance field of the node specified by x is uniquely determined at the point t.

Now consider a node specification x that specifies a node which is a child of another node whose specification is unambiguous at a given point t of the computation π (case (b)). If the presence of the child specified by x precedes the computation π, then the specification of x is unambiguous at the point t, and moreover, the balance field of the specified node, as defined at the onset of the computation π, is uniquely determined. Its balance field at the point t remains unchanged. Otherwise, the linking of the child specified by x takes place subsequent to the onset of the computation π, and case (a) implies that the balance field of this node is uniquely determined at that prior point (whereupon it remains fixed). □

Returning to the proof of Lemma 7, modified so that the respective v_i's are assumed to have identical initial balance fields, Lemma 11 nearly fulfills the added precondition necessary for the application of constraint (d′); only the uniqueness of the balance field of the designated minimum root remains to be demonstrated. However, the balance fields of the children (by relative position) of the designated minimum root are uniquely determined when the heap structure is semidetermined (as just demonstrated). Since properties (i) and (ii), satisfied by the balance fields, imply that the balance field of a node is uniquely determined given those of its children (by relative position), we conclude that the balance field of the designated minimum root is uniquely determined when the heap structure is semidetermined. Lemma 7, modified as described, therefore holds.

With the added assumption that the balance fields of the respective v_i's are identical, Lemmas 8 and 9 likewise hold since these follow directly from the lemmas that precede them. Considering the statement of Lemma 10, if we modify the meaning of q so that it refers to the nodes acted upon by the decrease-key operations *that have respective balance fields given by a common specified value*, then with this modification the lemma holds (again, since it is a direct consequence of the lemmas that precede it). Building upon Lemma 10, as modified, we proceed as follows:

LEMMA 12. *Suppose that there are f possible values for the balance field of a node and that at the onset of a given round there are q nodes to be acted upon by the decrease-key operations of that round, where $q \geq 2f \cdot (4(d+1))^4$. Assume that we stipulate a charge of $\lceil q^{1/3} \rceil$ for each of these selected nodes that does not have a subsequent efficient event and remains in the heap at the termination of the corresponding O-sequence continuation. Then for a random continuation of the O-sequence the expected total of the charges to these q nodes, as specified in their associated schedules at the onset of this round, is bounded by $\Omega(q \log(q/f))$.*

PROOF. Assume that the possible balance fields for a node ranges over the integers from 1 to f, and let q_i be the number of nodes to be acted upon by the decrease-key operations having balance field i, so that $\Sigma_i\, q_i = q$. For a given threshold t, we have

$$\sum_{i:\, q_i > t} q_i \geq q - ft. \tag{5}$$

Choosing $t = (4(d+1))^4$, the condition $q_i > t$ is equivalent to $q_i^{1/4} > 4(d+1)$, and from Lemma 10, modified as described above, it follows that when $q_i > t$, the expected total of the charges to the nodes reflected by q_i is bounded by $\Omega(q_i \log q_i)$. Applying Jensen's inequality and using (5), we conclude that the expected total of the charges to all q of the nodes is bounded by $\Omega(\Sigma_{i:\, q_i > t}\, q_i \log q_i) = \Omega((q - ft)\log(q/f - t))$. For $q \geq 2f(4(d+1))^4 = 2ft$, we have $(q - ft)\log(q/f - t) = \Omega(q \log(q/f))$, completing the proof. (Note that the stipulated charge $\lceil q^{1/3} \rceil$ is at least as large as any of the stipulated charges $\lceil q_i^{1/3} \rceil$ required for the application of Lemma 10.) □

The following theorem generalizes Theorem 1.

THEOREM 2. *Let c be a fixed positive constant, assume $n > 9$ and choose $C = \log_3 L$ and $d = \lceil \log_3(2C) \rceil$. Using a generalized pairing heap with balance fields to implement the operations, where the number b of bits in a balance field is at most $(1 - c) \log_2 \log_2 n$, let T_r denote the total expected cost of executing a random O-sequence consisting of r rounds. Then $T_r = \Omega(r \cdot L \log L) - O(nL^{1/3})$.* (*Recall* $L = \lceil \frac{1}{2} \log_3 n \rceil$.)

PROOF. We observe first that when $q \geq L/2$, $f \leq 2^b$, and $b \leq (1 - c) \log_2 \log_2 n$, the condition $q \geq 2f \cdot (4(d + 1))^4$ of Lemma 12 is satisfied (for n sufficiently large). Moreover, $q \log(q/f) = \Omega(L \log L)$. Our proof now proceeds exactly as the proof of Theorem 1, except that we use Lemma 12 in place of Lemma 10. □

3. *Experimental Findings*

The lower bound established in Theorem 1, which implies that the expected amortized cost of a round of an O-sequence is $\Omega(\log n \cdot \log \log n)$, stands in contrast with experimental results [Liao 1992; Stasko and Vitter 1987], that suggest that pairing heaps perform decrease-key operations in constant amortized time. One set of experiments, undertaken by Stasko and Vitter [1987], involves runs that perform a sequence of operations subdivided into multiple rounds, acting upon an initial structure, a binomial heap with n nodes. Each round contains a single insertion, $\log_2 n - 1$ decrease-key operations, and a single deletemin operation. No increase in the quantity

$$\frac{\text{average round cost}}{\log_2 n}$$

is observed between the cases $n = 2^{12}$ versus $n = 2^{18}$. Now the quantity log log n grows with sufficient rapidity that one might plausibly expect an experiment of this sort to distinguish between these two choices for the parameter n, provided that the experiment captures the underlying mechanism responsible for this growth. (Seriously—as we shall see!) Because this experimental setup is very similar to the framework of the analysis in Section 2, we feel compelled to address this matter.

The best performance for this experimental setup is achieved [Stasko and Vitter 1987] by a clever variation of the pairing heap referred to as the *auxiliary twopass* method, introduced by Stasko and Vitter [1987]. We proceed to describe this method. (The reader is referred to Stasko and Vitter [1987] for more details.) The auxiliary twopass method always completes a deletemin operation with a single tree remaining, referred to as the *main* tree. Between successive deletemin operations, the nodes and subtrees resulting from insertions and decrease-key operations are stored in what is referred to as the *auxiliary area*. When the next deletemin operation takes place, the trees in the auxiliary area are coalesced into a single tree using what is referred to as the *multipass method* [Fredman et al. 1986] (to be described shortly). This tree is then combined with the main tree using a linking operation. The root of the resulting tree is then removed, and its subtrees are then combined in the same manner as for the pairing heap described in Section 1.

The multipass method for coalescing a list of trees begins by linking the trees in pairs. This is now repeated for the list of trees resulting from this first pass (of which there are half as many as initially present), and then repeated again, as necessary, until a single tree remains. (An alternative and preferable implementation places the result of each linking at the end of a queue consisting of the trees being linked, proceeding with the linkings in a round robin manner until a single tree remains.)

The Stasko–Vitter [1987] experiment under consideration uses the following *greedy* heuristics in designing an adversary for testing the performance of the data structure. First is the matter of determining the outcomes of linking decisions. Quoting from Stasko and Vitter [1987]:

> No key values were ever assigned to nodes. Instead, we used a "greedy" heuristic to determine the winners of comparisons, in hopes of causing a worst case scenario. Every time a comparison-link operation was performed, the node with more children won the comparison: that is, it was judged to have the smaller key value.

This heuristic bears similarity to our rank rule in Section 2, particularly when choosing a small value for the parameter d in its definition.

In selecting operands for the decrease-key operations, the Stasko–Vitter experiment (again, quoting from Stasko and Vitter [1987]):

> utilized greedy decrease-key operations in which the node with the most children was chosen for the operation. Nodes such as the root and children of the root, whose choice would have no effect on the heap structure, were excluded from being candidates.

In assessing this heuristic, we speculate, borrowing terminology from the analysis in Section 2, that a substantial fraction of the linkings taking place among the subtrees placed in the auxiliary area might well be efficient, as a consequence of choosing greedy decrease-key operations that select these subtrees on the basis of having largest, hence similar, ranks. This might explain the absence of growth observed in the Stasko–Vitter data. In the sequel, we refer to the use of these greedy heuristics as *strategy G*.

We now present some new experimental results. As with the Stasko–Vitter experiments, we likewise test the performance of the auxiliary twopass method on runs that perform a sequence of operations subdivided into multiple rounds (as described above), acting upon an initial structure, a binomial heap with n nodes. As an alternative to strategy G, these experiments use information-theoretic heuristics. These heuristics are similar to and perhaps more natural than the rank and efficiency constructs from the analysis in Section 2, and serve to motivate those constructs.

Linking outcomes are determined on the basis of tree size; the root of the larger tree wins the comparison. An exception, however, is that the root of the main tree at the onset of a round of operations wins the single comparison in which it participates during the deletemin operation that ends the round. (This exception corresponds to our adversary's override of the rank rule in Section 2.)

The operands selected for decrease-key operations are children of the root of the main tree, selected on the following basis. Associate with each link in the

main tree an efficiency value defined as

$$\frac{\text{size of child tree}}{\text{size of parent tree}},$$

where the size values are assessed just prior to the linking. In view of the basis for our linking decisions, the efficiency value associated with a link cannot exceed 1. We choose the decrease-key operands for a round of operations to consist of those children of the main tree root whose links to the root have the highest efficiency values. The decrease-key operations on these nodes are performed in random order. (Observe that a high efficiency value corresponds to the notion of efficient linking used in our analysis.) In the sequel, we refer to the use of these information-theoretic heuristics as *strategy I-T*.

Contrasting our experiment with the Stasko–Vitter experiment, one difference is particularly noteworthy. Whereas strategy G selects its decrease-key operands from the lower levels of the tree, strategy I-T selects these operands from among the children of the root of the main tree. In terms of immediate impact on the cost of the deletemin operation that ends a particular round, under strategy G each decrease-key operation in the round increases the cost of this deletemin operation, whereas under strategy I-T, this differential cost is zero. In this sense, strategy I-T yields an advantage to strategy G since our goal is to maximize the observed performance costs.

We have implemented both strategies G and I-T, measuring the resulting performance costs as follows: Our measure of performance is in terms of average round cost, where the cost associated with a given round of operations is defined to be the number of linkings that take place, when executing the round, divided by $\log_2 n$, n = heap size. The results of the Stasko–Vitter [1987] experiments are reported in terms of a work ratio definition, which differs from average round cost by almost a constant factor (plus a constant offset). Comparing the results we have obtained for strategy G with the previously reported results [Stasko and Vitter 1987], we find that our implementation yields slightly greater work ratios, but happily quite close to those previously reported (within 4.5%). (The description of the greedy strategy [Stasko and Vitter 1987], quoted above, allows for a small amount of variation in its implementation.)

Our experiments evaluate average round costs over intervals consisting of $n/16$ rounds (n = heap size). Our runs consist of 32 intervals, so that $2n$ rounds are executed in each run. Strategy G is deterministic, and we have executed a run for each of the cases $n = 2^{12}$ and $n = 2^{18}$ (the values used in Stasko and Vitter [1987]), as well as for the additional intermediate case $n = 2^{15}$. For each run, we compute the quantities q_i, $1 \le i \le 32$, where q_i is defined to be the average of the values (round cost)/$\log_2 n$, where the average is taken over all rounds in the ith interval. Strategy I-T uses randomization, and we have executed 100 runs for each of the cases $n = 2^{12}$, and $n = 2^{18}$, as well as for the additional intermediate case $n = 2^{15}$. For each of these three values for n, we compute for the rth run the 32 interval averages, $q_{i,r}$, $1 \le i \le 32$, for $1 \le r \le 100$. Then, for each i, we compute the average and standard deviation of the 100 quantities, $q_{i,r}$, $1 \le r \le 100$, obtaining values c_i and σ_i, $1 \le i \le 32$. The quantities c_i reflect average round costs during corresponding stages of the data structure evolution, for the ensemble of 100 runs. For cases $n = 2^{12}$ and 2^{18}, the plots

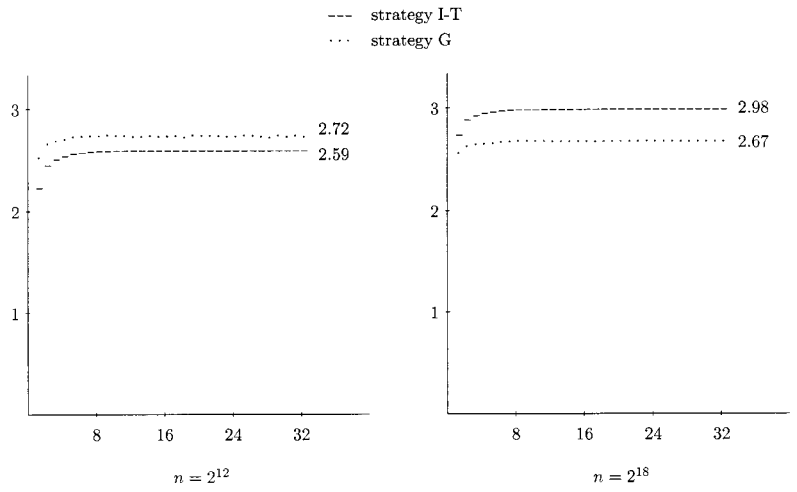

FIG. 1. $n = 2^{12}$ and 2^{18}.

shown in Figure 1 display the quantities q_i, associated with strategy G, and the corresponding quantities c_i associated with strategy I-T.

Although we have not presented a plot for the case, $n = 2^{15}$, we find that the q_i values for strategy G settle into a range from 2.71 to 2.72. Thus, strategy G reveals no increase in the observed costs as n ranges over the three values, 2^{12}, 2^{15}, and 2^{18}.

When $n = 2^{15}$, strategy I-T generates c_i values close to 2.81. Thus, strategy I-T reveals an increase in the observed costs as n ranges over the three values, 2^{12}, 2^{15}, and 2^{18}, generating respective costs, 2.59, 2.81, and 2.98, and overtakes strategy G at $n = 2^{15}$. The standard deviations σ_i, $1 \leq i \leq 32$, for the 32 sets of corresponding interval data are small, being uniformly bounded, respectively, by 0.02, 0.005, and 0.003, for the respective cases, $n = 2^{12}$, 2^{15}, and 2^{18}.

4. *Concluding Remarks and Open Problems*

We have shown that generalized pairing heaps do not perform decrease-key operations in constant amortized time, contrary to conjecture and experimental evidence. More precisely, we have established that the amortized cost of the decrease-key operation can be as high as $\Omega(\log \log n)$ when n items are present in the heap. The particular variants of pairing heaps that have been suggested in Fredman et al. [1986] and Stasko and Vitter [1987] are all instances of generalized pairing heaps, and therefore subject to this lower bound. We have further demonstrated that even if we allow $(1 - c) \log_2 \log_2 n$ bits of balance information in the tree nodes, the same result holds. Additionally, we have presented experimental findings concerning pairing heaps, suggesting that growth in the amortized cost of decrease-key operations is a detectable phenomenon.

We mention a positive result in connection with pairing heaps. Consider performing a sequence of $m \geq n$ heap operations, beginning with an initially empty heap, which includes at most n deletemin operations, and such that the heap size does not exceed n at any point. Then the total execution cost of the sequence does not exceed $O(m \log_{2m/n} n)$. Thus, under these circumstances

pairing heaps are at least as efficient as d-heaps [Johnson 1975]. In particular, for applications involving graph algorithms (e.g., shortest path, minimum spanning tree) on dense graphs, where the number of edges grows as $n^{1+\epsilon}$, this result implies that the decrease-key operations contribute only constant cost per operation to the total execution cost. This result is derived by using the same potential function used for the analysis of pairing heaps in Fredman et al. [1986], but within the context of a *nonlinear* amortized analysis; a deletemin operation with actual cost t decreases the potential by at least $t \log(t/\log n) - \log n$.

Our lower bound analysis does not readily generalize to include the following modification of the pairing heap data structures. Consider maintaining parent pointers in the data structure; when performing a decrease-key operation a check is first performed to determine whether heap-order is violated by the new key value, performing a cut only in this instance. It should be noted, however, that this modification, even if theoretically fruitful, diminishes the practical effectiveness of these data structures.

We close with two open problems. First, the analysis of pairing heaps is far from complete; the gap between the upper and lower bounds remains large. The second problem is to analyze pairing heaps, modified to include parent pointers in the nodes as described above, for which our lower bound analysis is not applicable.

REFERENCES

COLE, R. 1995. On the dynamic finger conjecture for splay trees Part II: The proof. Tech. Rep. TR1995-701 (Aug.). New York Univ., New York.

COLE, R., MISHRA, B., SCHMIDT, J., AND SIEGEL, A. 1995. On the dynamic finger conjecture for splay trees Part I: Splay sorting log n-Block sequences. Tech. Rep. TR1995-700 (Aug.). New York Univ., New York.

FREDMAN, M. L., SEDGEWICK, R., SLEATOR, D. D., AND TARJAN, R. E. 1986. The pairing heap: a new form of self-adjusting heap. *Algorithmica 1*, 1, 111–129.

FREDMAN, M. L., AND TARJAN, R. E. 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM 34*, 3 (July), 596–615.

JOHNSON, D. B. 1975. Priority queues with update and finding minimum spanning trees. *Inf. Proc. Lett. 4*, 53–57.

LIAO, A. M. 1992. Three priority queue applications revisited. *Algorithmica 7*, 4, 415–427.

SLEATOR, D. D., AND TARJAN, R. E. 1985. Self-adjusting binary search trees. *J. ACM 32*, 3 (July), 652–686.

STASKO, J. T., AND VITTER, J. S. 1987. Pairing heaps: Experiments and analysis. *Commun. ACM 30*, 3 (Mar.), 234–249.

RECEIVED JUNE 1998; REVISED DECEMBER 1998; ACCEPTED FEBRUARY 1999