Pairing heaps:experiments and analysis

J.T. Stasko, J.S. Vitter

▶ **To cite this version:**

J.T. Stasko, J.S. Vitter. Pairing heaps:experiments and analysis. RR-0600, INRIA. 1987. inria-00075954

HAL Id: inria-00075954
https://inria.hal.science/inria-00075954

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France

Tél. (1) 39 63 55 11

Rapports de Recherche

N° 600

PAIRING HEAPS: EXPERIMENTS AND ANALYSIS

John T. STASKO
Jeffrey Scott VITTER

Février 1987

Pairing Heaps: Experiments and Analysis[1]
Pairing Heaps: Expériences et Analyse[1]

John T. Stasko[2,3] and *Jeffrey Scott Vitter*[2,4,5]

Abstract. The pairing heap has recently been introduced as a new data structure for priority queues. Pairing heaps are extremely simple to implement and seem to be very efficient in practice, but they are difficult to analyze theoretically, and open problems remain. It has been conjectured that they achieve the same amortized time bounds as Fibonacci heaps, namely, $O(\log n)$ time for *delete* and *delete_min* and $O(1)$ for all other operations, where n is the size of the priority queue at the time of the operation. In this paper we provide empirical evidence that supports this conjecture. The most promising algorithm in our simulations was a new variant of the twopass method, called auxiliary twopass. We prove that, assuming no *decrease_key* operations are performed, it achieves the same amortized time bounds as Fibonacci heaps.

Résumé. Le «pairing heap» a été récemment présenté comme nouvelle structure de données pour les queues de priorité. Les pairing heaps sont efficaces et simples à construire, mais il est difficile de les analyser exactement. Il a été conjecturé qu'ils ont les mêmes coûts amortis que les heaps de Fibonacci, c'est à dire, un coût $O(\log n)$ pour la suppression et la suppression du minimum et un coût constant pour les autres opérations, où n est la taille de la queue de priorité a l'instant de l'opération. On présente dans cet article des données expérimentales à l'appui de cette conjecture. L'algorithme le plus rapide dans les expériences était une nouvelle forme de la méthode «twopass», appellée «auxiliary twopass». On prouve que le coût de cette méthode est le même que pour le heap de Fibonacci, en supposant qu'il n'y a pas d'opérations de décrementation de priorité.

---

[1] An earlier version of this manuscript appeared as Department of Computer Science Technical Report CS–86–02, Brown University (February 1986).

[2] Support was provided in part by NSF research grant DCR–84–03613, by an NSF Presidential Young Investigator Award with matching funds from an IBM Faculty Development Award and an AT&T research grant, and by a Guggenheim Fellowship.

[3] Current address: Department of Computer Science, Brown University, Box 1910, Providence, R.I. 02912, USA.

[4] Current address: I. N. R. I. A., Bâtiment 8, Domaine de Voluceau, Rocquencourt, B. P. 105, 78153 Le Chesnay Cedex, France. Part of this research was also performed at Brown University in Providence, R. I., and at the Mathematical Sciences Research Institute in Berkeley, California.

[5] To whom correspondence should be addressed.

PAPIER RECUPERÉ ET RECYCLÉ

1. Introduction

A priority queue is an abstract data type for maintaining and manipulating a set of items based on priority. Priority queues derive great theoretical and practical importance from their use in solving a wide range of combinatorial problems, including job scheduling, minimal spanning tree, shortest path, and graph traversal.

Priority queues support the operations *insert*, *find_min*, and *delete_min*; additional operations often include *decrease_key* and *delete*. The *insert*(t, v) operation adds item t with key value v to the priority queue. The *find_min* operation returns the item with minimum key value. The *delete_min* operation returns the item with minimum key value and removes it from the priority queue. The *decrease_key*(t, d) operation reduces item t's key value by d. The *delete*(t) operation removes item t from the priority queue. The *decrease_key* and *delete* operations require that a pointer to the location in the priority queue of item t be supplied explicitly, since priority queues do not support searching for arbitrary items by value. Some priority queues also support the *merge* operation, which combines together two item-disjoint priority queues.

In this paper, we shall concentrate on the *insert*, *delete_min*, and *decrease_key* operations, because they are the operations that primarily distinguish priority queues from other set manipulation algorithms, and since they are the critical operations as far as the time bounds are concerned.

Several priority queue implementations, such as implicit heaps [Williams, 64], leftist heaps [Crane, 72], [Knuth, 73], and binomial heaps [Vuillemin, 78], [Brown, 78] have been shown to exhibit an $O(\log n)$ worst-case time bound for all operations, where n is the size of the priority queue at the time of the operation. Fibonacci heaps [Fredman and Tarjan, 84] provide a dramatic improvement on the general logarithmic bound by achieving amortized time bounds of $O(1)$ for *insert*, *decrease_key*, and *find_min* and $O(\log n)$ for *delete_min* and *delete*. This greatly improves the best known theoretical bounds for the time required to solve several combinatorial problems.[6] Following the approach of [Tarjan, 85], a sequence of operations op_1, op_2,

[6] For example, a standard implementation of Dijkstra's algorithm (which finds the shortest path from a specified vertex x to all other vertices in a graph with nonnegative edge lengths) uses a priority queue as follows: Let us denote the number of vertices in the graph by V and the number of edges by E. The key value of each item y in the priority queue represents the length of the shortest path from vertex x to vertex y using only the edges in the graph already processed. Initially, no edges are processed, and the priority queue contains V items; the key value of item x is 0 and all other items have key value ∞. The algorithm successively performs *delete_mins* until the priority queue is empty. Each time a *delete_min* is performed (say, the vertex y is deleted), the algorithm outputs the shortest path between x and y, and each unprocessed edge (y, z) incident to y in the graph is processed; this may require that a *decrease_key* be performed in order to lower the key value of z in the priority queue. Thus, there are at most V *inserts*, V *delete_mins*, and E *decrease_keys* during the course of the algorithm. If a Fibonacci heap is used to implement the priority queue, the resulting running time is $O(E + V \log V)$, which is a significant improvement over $O((E + V) \log V)$ using the other heap representations. Other examples of how Fibonacci heaps can improve worst-case running times are given in [Fredman and Tarjan, 84].

..., op_k is said to have *amortized time bounds* $b_1, b_2, \ldots, b_k$ if

$$\sum_{1 \le i \le j} t_i \le \sum_{1 \le i \le j} b_i, \qquad \text{for all } 1 \le j \le k,$$

where t_i is the actual time used by op_i. Intuitively, if operation op_i uses less time than its allotted b_i units, then the leftover time may be held in reserve to be used by later operations.

Fibonacci heaps achieve their time bounds by complicated invariants with significant overhead, so they are not the method of choice in practice. Recently, a self-adjusting data structure called the *pairing heap* was proposed [Fredman et al, 86]. Pairing heaps are much simpler than Fibonacci heaps, both conceptually and in implementation, and they have less overhead per operation. The best amortized bound proved so far for pairing heaps is $O(\log n)$ time per operation. It is conjectured that pairing heaps achieve the same amortized time bounds as Fibonacci heaps, namely, $O(1)$ per operation except $O(\log n)$ for *delete_min* and *delete*.

To test whether the conjecture is true, we performed several simulations of the pairing heap algorithms. These simulations differed significantly from the ones independently done in [Jones, 86], since the latter ones did not address the conjecture. In our simulations we tested several different pairing heaps and used "greedy" heuristics and the appropriate sequences of commands to make the pairing heaps perform as poorly as we could. The results were positive in that the pairing heaps always performed extremely well. This does not prove that the desired time bounds do hold, but it is reassuring and makes us optimistic that the conjecture is true.

In this paper, we study the "twopass" and "multipass" versions of pairing heaps; the names arise from the method used to do the *delete_min* in each version [Fredman et al, 86]. We also introduce new variants called "auxiliary twopass" and "auxiliary multipass." All versions are described in the next section. In Section 3, we discuss our simulations and the empirical data. Auxiliary twopass performed best in the simulations, based upon our measure of performance. In Section 4 we provide a partial theoretical analysis of pairing heaps by introducing the concept of "batched potential." We show, for example, that auxiliary twopass uses $O(1)$ amortized time per *insert* and *find_min* and $O(\log n)$ amortized time for the other operations. Conjectures and open problems follow in Section 5.

2. Pairing Heap Algorithms

A comprehensive description of pairing heaps appears in [Fredman et al, 86]. A summary is given below. Our studies involve the twopass algorithm, which was the subject of most of the analysis in [Fredman et al, 86], and the multipass algorithm.

Pairing heaps are represented by heap-ordered trees and forests. The key value of each node in the heap is less than or equal to those of its children. Consequently, the node with minimum value (for simplicity, we shall stop referring to a key value, and just associate the value directly with the heap node) is the root of its tree. Groups of siblings, such as tree roots in a forest, have no intrinsic ordering.

In the general sense, pairing heaps are represented by multiway trees with no restriction on the number of children that a node may have. Because this multiple child representation is difficult to implement directly, the child-sibling binary tree representation of a multiway tree is used, as illustrated in Figure 1. In this representation, the left pointer of a node accesses its first child, while the right pointer of a node accesses its next sibling. In terms of the binary tree representation, it then follows that the value of a node is less than or equal to all the values of nodes in its left subtree. A third pointer, to the previous sibling, is also included in each node in order to facilitate the *decrease_key* and *delete* operations. The number of pointers can be reduced from three to two, as explained in [Fredman et al, 86] at the expense of a constant factor increase in running time. Unless stated otherwise, the terms "child," "parent," and "subtree" will be used in the multiway tree sense; their corresponding meaning in the binary tree representation should be clear.

(Figure 1)

The primary action performed in pairing heap operations is a *comparison-link*, in which the values of two nodes are compared. The node with larger value is demoted in the sense that it becomes the first child of the smaller-valued node. The previous first child of the smaller node becomes the second child, the previous second child becomes the third child, and so forth. Ties can be broken arbitrarily. The binary tree representation of the comparison-link is given in Figure 2. This comparison-link action is performed repeatedly during the *delete_min* operation of a priority queue. It is the primary action that we seek to minimize in order to reduce execution times.

(Figure 2)

The *twopass algorithm* that we examined was the variant that yielded the $O(\log n)$-time amortized bounds for *insert*, *decrease_key*, and *delete_min* in [Fredman et al, 86]. Only one tree is maintained. Hence, in the binary tree representation, the root node always has a null right pointer. The *insert*(t, v) operation performs a comparison-link between t and the tree root; the node with smaller value becomes the root of the resulting tree. The *decrease_key*(t, d) operation begins by reducing t's value by d. This means that t may now have a value smaller than its parent. Consequently, it must be removed from the tree (with its own subtree intact) and comparison-linked with the tree root. Again, the node with smaller value becomes the root of the resulting tree.

The *delete_min* operation is what gives the twopass algorithm its name. First, the tree root node is deleted and its value returned. This leaves a forest of former children and their subtrees. Next, two comparison-linking passes are made over the roots of this forest. Pass 1 is a left-to-right pass, in which a comparison-link is performed on successive pairs of root nodes. Pass 2 then proceeds from right-to-left. In each step, the two rightmost trees are replaced by the tree resulting from a comparison-link; the "cumulative" rightmost tree is continually updated in this manner until it is the only remaining tree. The root of this final tree is the minimum of all the nodes in the tree. Figure 3 illustrates the *delete_min* procedure in terms of the binary tree representation.

(Figure 3)

The *delete*(t) operation works as follows: If the node, t, to be deleted is the root of the main tree, then a *delete_min* operation is performed. Otherwise, t is deleted from the tree. The former subtrees of t are recombined into a single tree via the twopass linking procedure. This tree is then comparison-linked to the root of the main tree.

The *multipass algorithm* that we studied was also presented in [Fredman et al, 86]. Both the *insert*(t, d) and *decrease_key*(t, d) operations function exactly as those in the twopass algorithm. The *delete_min* operation, however, is what distinguishes multipass from twopass. The first operation in the multipass *delete_min* is the deletion of the root node; its value is returned. This leaves the heap with some number of trees, say r. Next, we repeatedly perform pairwise linking passes on the roots of these trees until the heap is left with only one tree. Each comparison-link reduces the number of trees by one, and each pass cuts the number of trees roughly in half. For r trees, a total of $\lceil \log r \rceil$[7] passes are made. A simpler heuristic is to comparison-link the first two tree roots and place the "winning" root (smaller value) at the tail of the forest list. Alternatively, a circular list could be used to store the siblings. Both ways, a round robin effect emerges, and we see that for r tree roots, exactly $r - 1$ link operations are performed. Following the linking phase, the heap is again left with a single tree; its root is the node with minimum key value. Figure 4 illustrates the multipass *delete_min* in terms of the binary tree representation. The *delete*(t) operation is the same as in twopass except that multipass comparison-linking is used on the children of the deleted node t.

(Figure 4)

While working with these algorithms, we designed two new variations, which we call *auxiliary twopass* and *auxiliary multipass*. Auxiliary twopass works as follows: In addition to the main tree in the heap, we maintain an auxiliary area that consists of an ordered list of other trees. It is convenient in the implementation to store the auxiliary area as the right subtree (in the binary tree sense) of the root.

The *insert*(t, v) and *decrease_key*(t, d) operations function as in the regular multipass algorithm except for one major difference. Rather than comparison-linking node t with the tree root, the node is added to the end of the list of auxiliary trees. (As usual, in the case of a *decrease_key* operation, the subtree rooted at t remains intact.) If the *find_min* operation is to be implemented, a separate minimum pointer must be maintained; each *insert* or *decrease_key* node must be checked against the minimum pointer so that the pointer can be updated if necessary.

The *delete_min* operation, which is illustrated in Figure 5, begins by "batching" the auxiliary area, that is, by running the multipass pairing procedure on the auxiliary area. (Note that although the method being described is called auxiliary twopass, the auxiliary area is linked together using the multipass method.) When this linking is complete, the auxiliary area consists of a single tree. If the auxiliary area originally consists of 2^k singleton trees, for some $k \geq 0$, the resulting tree is a binomial tree

[7] All logarithms in this paper are base 2.

[Vuillemin, 78], [Brown, 78]. The next action is a comparison-link between the main tree root and the new auxiliary root. After this comparison-link, the heap again contains only one root node, that of minimum key value. From this point on, the *delete_min* operation proceeds exactly as in the twopass algorithm. The root node is removed, and we link the remaining forest of trees via the twopass procedure.

(Figure 5)

The rationale for maintaining the auxiliary area is that it prevents many comparisons between single nodes from an *insert* and large trees already in the forest. We shall prove in Section 4 that if there are no *decrease_key* operations, auxiliary twopass achieves the amortized time bounds of $O(1)$ for *insert* and $O(\log n)$ for *delete_min*.

The *delete*(t) operation works as in twopass. If node t is the current minimum, then a *delete_min* is performed. Otherwise, the children of t are recombined into a single tree, which is then comparison-linked to the main tree.

Auxiliary multipass is identical except that the multipass algorithm is used on the regular tree. The auxiliary area is still batched using multipass. Section 4 derives slightly weaker amortized time bounds than for auxiliary twopass, under the assumption that no *decrease_key* operations are performed: $O(1)$ per *insert* and $O\big((\log n \log\log n)/\log\log\log n\big)$ per *delete_min*.

Lazy variants of these algorithms are also possible, in which the heap consists of a forest of trees rather than a single tree. One possible implementation is described in [Fredman et al, 86]. However, extra comparisons other than in comparison-link actions are required to implement *find_min*, since the heap no longer has a single root. As a result, the *find_min* operation in the lazy variants often does some restructuring of the tree. The *find_min* operation for the auxiliary variants can be done in constant time, since a pointer to the current minimum node can easily be maintained during *inserts* and *decrease_keys*; the extra comparison to do this is balanced by the fact that *inserts* and *decrease_keys* do not perform any comparison-links. In order to make our simulation results of *insert*, *decrease_key*, and *delete_min* fair, we have excluded lazy variants from our study. Their performances are similar.

Although the twopass algorithm has provided the fastest general amortized time bounds so far, our intuition suggested that the multipass variants should run faster. While all make roughly the same number of comparison-link actions on a similar heap configuration, the multipass variants tend to build a more "structured" forest configuration over time. All the uppermost nodes that are directly involved in link actions will be formed into a binomial-like tree, which helps limit the number of links during subsequent *delete_min* operations. The simulation results described in the next section are somewhat surprising; auxiliary twopass consistently outperforms the multipass versions.

3. Simulations

Our test simulations of the pairing heap algorithms consisted of structured sets of *insert*, *decrease_key*, and *delete_min* operations. 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 node gained one child in the link operation. Our greedy approach allowed us to keep the nodes with many children at the uppermost levels in the heap. Since the number of children at the root level is what determines how much work a *delete_min* performs, this greedy approach forced the priority queue to do significantly more work than would have been the case if the key values were assigned randomly. Two methods were used for determining which nodes to use for *decrease_key* operations: In a *random decrease_key*, we chose a non-root node at random. In a *greedy decrease_key*, we used the greedy heuristic and chose the node with the most children, subject to the constraint that the node could not be the root or a child of the root.

The binary tree representation of a multiway tree was used to implement the pairing heaps. Heap nodes were implemented as record structures with left (first child) and right (next sibling) pointer fields.

In this section we report on nine simulations of twopass and multipass pairing heap algorithms. Each simulation consisted of several *phases*. A phase consisted of some set of *inserts* and *decrease_keys* followed by a *delete_min*. In the descriptions that follow, n refers to the size of the priority queue at the beginning of the phase. The phases of the nine simulations consisted of, respectively:

(1) $\log n$ *inserts*, followed by one *delete_min*.

(2) $0.5 \log n$ ⟨*insert*, random *decrease_key*⟩ pairs, followed by one *delete_min*.

(3) $0.5 \log n$ ⟨*insert*, greedy *decrease_key*⟩ pairs, followed by one *delete_min*.

(4) one *insert*, then $x(\log n) - 1$ greedy *decrease_keys*, followed by one *delete_min*, for $x = 0.25$, with an initial binomial tree of size 2^{12}.

(5) same for $x = 1.0$.

(6) same for $x = 4.0$.

(7) one *insert* then $x(\log n) - 1$ greedy *decrease_keys*, followed by one *delete_min*, for $x = 0.22$, with an initial binomial tree of size 2^{18}.

(8) same for $x = 1.0$.

(9) same for $x = 4.0$.

Our measure of performance compared the actual work done by each algorithm against an allowance for the operations processed. The actual work done was considered to be one unit for an *insert*, one unit for a *decrease_key*, and one unit for the delete plus one unit for each comparison-link that occurred during a *delete_min*. Allowances for the operations corresponded to the amortized time bounds sought for them: the *insert* and *decrease_key* allowances were each one unit, and the *delete_min* allowance was $\log n$, where n was the heap size at the time of the *delete_min*.

Phases were grouped into a smaller number of increments to facilitate graphical display of the results; in the first three simulations the size of the heap grew by a fixed amount in each increment, and in Simulations 4–9 increments consisted of a fixed number of phases. For each increment in a simulation, we calculated its *work ratio*. The work ratio is defined as actual work performed divided by the operations' allowances. By seeing how the work ratio changed over time across these increments, we were able to judge the performances of the algorithms. If the work ratio increased

over time, the $O(1)$-time *insert* and *decrease_key* allowances and the $O(\log n)$-time *delete_min* allowance would not be bounding the actual work growth. If instead the work ratio stayed constant or decreased, then the experiments would provide encouragement that the sought-for amortized time bounds are possible.

In our simulations we chose the particular order and frequency of operations that we did so that if any of the conjectured amortized time bounds of $O(1)$ for *insert* and *decrease_key* and $O(\log n)$ for *delete_min* did not hold, we would detect a discrepancy in the results. The simulations of pairing heaps in [Jones, 86], on the other hand, were limited for several reasons: First, no *decrease_key* operations were performed. We shall see in Section 4 that if no *decrease_keys* are done, we can prove the conjectured bounds analytically for auxiliary twopass. More importantly, however, the simulations performed the same number of *inserts* as *delete_mins*. It is already known from [Fredman et al, 86] that each operation can be done in $O(\log n)$ amortized time. Therefore, it was impossible in Jones's simulations to distinguish between $O(1)$ time and $O(\log n)$ time per *insert*. Our simulations, on the other hand, tested the time bound conjecture more effectively by performing $O(\log n)$ *insert* and *decrease_key* operations for each *delete_min*.

Simulation 1 allowed us to examine how the heaps performed when no *decrease_key* operations were used. Work ratios were calculated over increments of 12,000 nodes of heap growth, with the heap size eventually reaching 1,200,000 nodes. Four initial *inserts* were used to "start-up" the simulation. The results are graphed in Figure 6. Only multipass showed a steady increase in work ratio; however, there was a marked decrease near the simulation's end. Both twopass and auxiliary twopass remained mostly steady, while auxiliary multipass exhibited a clear decrease. Auxiliary twopass was the fastest algorithm, a fact that would continue through most of the following simulations. It is interesting to note that the auxiliary algorithms were not subject to wide fluctuations in work ratio as were the regular algorithms.

Simulation 2 utilized random *decrease_key* operations primarily to see how random disruptions in the heap structure would affect overall algorithm performances. Again work ratios were calculated in 12,000 node increments, and the total heap size grew to 1,200,000 nodes. Sixteen *inserts* were used to initialize the simulation. This simulation's results, which are shown in Figure 7, were quite similar to those of Simulation 1. No steady work ratio increases were evident, nor was there an appreciable gain in work ratio values from Simulation 1. Both multipass algorithms exhibited work ratio decreases, with regular multipass remaining slightly superior. Auxiliary twopass was again the fastest algorithm, and twopass was the slowest. Curiously, the total work ratio for auxiliary twopass over the entire simulation was slightly less than its total in Simulation 1. In essence, the random *decrease_keys* helped the algorithm run faster.

Simulation 3 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. This simulation's *decrease_key* operation was intended to move nodes with many children from the central heap up to the top root level, thereby forcing the *delete_min* operations to do even more work. Because of the extra storage required, work ratios were calculated in increments of 6,000 nodes. The final heap size was 546,000 nodes. A start-up set of sixteen *insert* operations was used.

After showing small jumps in the work ratios, all four algorithms maintained steady levels at the simulation's end. The results are graphed in Figure 8. Clearly, the greedy *decrease_key* operations did have an effect, as work ratio values were higher than those in the first two simulations. Auxiliary twopass was again the fastest algorithm, twopass was the slowest, and the multipass algorithms were quite similar. The general mulitpass algorithm had a definite superiority at smaller heap sizes, however.

Simulations 4–9 primarily examined how the *decrease_key* operation affected algorithm performances. An initial binomial tree of some size was built. This provided all four algorithms with the same starting point, so no initial bias was introduced. Each phase contained only one *insert* operation; hence, a constant heap size was maintained, that of the initial binomial tree. By varying the number of *decrease_key* operations between the *insert* and the *delete_min* in a phase, we could see exactly how this number affected the work ratio.

In Simulations 4–6, we used an initial binomial tree size of $n = 2^{12} = 4096$ nodes. Simulations 4, 5, and 6 used $0.25 \log n = 3$, $\log n = 12$, and $4 \log n = 48$ *decrease_key* operations per phase, respectively. We grouped 100 phases into an increment for work ratio calculations, then we ran the simulation for 100 increments. The respective results are shown in Figures 9, 10, and 11. All three tests showed very steady work ratio rates, which is encouraging. The actual values were quite different, however. The twopass variants had lower work ratios when there were a smaller number of *decrease_key* operations per phase, whereas the multipass variants exhibited an opposite behavior; they performed better as the number of *decrease_keys* per phase increased. Auxiliary twopass was overall the best algorithm, but regular twopass exhibited a curious variation on its usual slowest behavior. In the two simulations with more *decrease_key* operations, twopass was clearly slowest. In fact, in Simulation 6 with 48 *decrease_key* operations per phase, twopass was blatantly behind the other three algorithms' performances. But in Simulation 4 with the fewest (3) *decrease_keys* per phase, twopass was the fastest algorithm! It appears that twopass performs best when there are relatively few *insert* and *decrease_key* operations compared to the number of *delete_min* operations.

In Simulations 7–9, we used a much larger initial binomial tree size of $n = 2^{18} = 262,144$ nodes. Simulations 7, 8 and 9 used $0.22 \log n = 4$, $\log n = 18$, and $4 \log n = 72$ *decrease_key* operations per phase, respectively. We grouped 1000 phases into an increment for work ratio calculations, then we ran the simulation for 100 increments. The results are shown in Figures 12, 13, and 14. The relative performances of the four algorithms were quite similar to those of Simulations 4–6. The curious behavior of the twopass algorithm was again evident, as it performed poorly with many *decrease_key* operations per phase, but improved dramatically with few. With the larger initial heap size, however, it never overtook auxiliary twopass as the fastest algorithm, as it did in Simulation 4.

In the last six simulations with constant heap size, the work ratio during the formation of the initial binomial tree was just under 2.0; we performed n *insert* operations, followed by $n - 1$ comparison-links. Therefore, actual work was $2n - 1$ units, while the *insert* allowance was n units, giving a work ratio of $2 - 1/n$. This amount was included in the total work ratio for the simulation.

Besides Simulations 1–9, we also performed two randomized simulations to verify

that our data were not dependent on the fixed structure of each phase. In the first, we kept the heap size constant as in Simulations 4–9, but the number of *decrease_keys* per phase was uniformly distributed between 0 and $2 \log n$. The second began by performing 64 initial *inserts* which were followed by a random sequence of *insert*, greedy *decrease_key*, and *delete_min* operations, all having the same probability of occurrence. Both results were consistent with those above; work ratio values stayed steady or showed a small decrease, and auxiliary twopass again exhibited the lowest overall work ratios. In the random sequence simulations, however, twopass and auxiliary twopass performed almost identically.

The data from these simulations allowed us to make the following conclusions: First, the $O(1)$-time bounds for *insert* and *decrease_key* and the $O(\log n)$-time bound for *delete_min* appear to hold in the amortized sense. Our data provided no evidence to the contrary. In fact, they provide some clue as to the actual coefficients implicit in the big-oh terms. Let us make the simplifying assumption that the amortized running time for each *insert* and *decrease_key* operation in the simulations is c time units and that the amortized time per *delete_min* is $d \log n$ units. Solving a set of linear equations obtained from Simulations 4–9 gives $c \approx 2.9$ and $d \approx 1.5$ for twopass, $c \approx 1.8$ and $d \approx 2.2$ for multipass, $c \approx 2.0$ and $d \approx 1.7$ for auxiliary twopass, and $c \approx 1.9$ and $d \approx 2.3$ for auxiliary multipass. Second, the auxiliary twopass algorithm was clearly the best overall. It typically exhibited lower work ratios than the other three algorithms. Third, adding the auxiliary area to multipass caused no great improvement to the algorithm. In our tests, the multipass algorithm was almost always superior to its auxiliary variant. Finally, the regular twopass algorithm's performance was quite variable. It was often the worst, especially when many *insert* and *decrease_key* operations were processed. As the number of these operations declined, however, its performance improved to rival that of auxiliary twopass.

4. Batched Potential

The best known amortized time bounds of the *insert*, *decrease_key*, and *delete_min* operations for the twopass pairing heap are all $O(\log n)$, due to [Fredman et al, 86]. In order to equal the time bounds for Fibonacci heaps, the *insert* and *decrease_key* time bounds must be shown to be $O(1)$. In an effort to prove the constant time bounds for pairing heaps, we shall use the auxiliary twopass algorithm and introduce *batched potential*. But before that, let us briefly review the concept of potential as it applies to amortized algorithmic analysis.

The potential technique for amortized analysis is discussed in [Tarjan, 85]. Each configuration of the pairing heap is assigned some real value Φ, known as the the "potential" of that configuration. For example, one could define the potential of a pairing heap configuration to be the number of trees it contains. For any sequence of n operations, the amortized time of the ith operation is defined to be the actual running time of the operation plus the change in potential, namely, $t_i + \Phi(i) - \Phi(i-1)$, where t_i is the actual time of the ith operation, $\Phi(i)$ is the potential after the ith operation, and $\Phi(i-1)$ is the potential before the ith operation. If we start with potential 0 and end up with positive potential, then the total running time is bounded by the total amortized time, via the telescoping effect of the potential changes.

Theorem 1. *The auxiliary twopass pairing heap algorithm achieves amortized time bounds of $O(1)$ for insert and find_min and $O(\log n)$ for delete_min and delete if no decrease_key operations are allowed, where n is the size of the priority queue at the time of the operation.*

Proof. Without loss of generality, we can restrict ourselves to *insert* and *delete_min* operations exclusively. Let us define the rank of a node to be the binary logarithm of the number of nodes in the subtree rooted there (in the binary tree sense). We use a variant of the potential function used to analyze twopass in [Fredman et al, 86]. We define the potential Φ of a heap configuration to be the sum of the ranks of all nodes in the main tree (that is, not counting the auxiliary area) plus 5 times the number of roots in the auxiliary area.

Each *insert* places a new node into the auxiliary area and increases the potential by 5; its amortized time is thus 6. There is no large single change in potential until a *delete_min* is performed, when the nodes in the auxiliary area are added to the main tree. We refer to this as "batched potential," because in effect changes in potential are not considered until a *delete_min* is performed.

Let us consider the case in which the auxiliary area is nonempty when a *delete_min* takes place. We let $i > 0$ denote the number of (root) nodes in the auxiliary area. The multipass linking spends $i - 1$ units of work building these i nodes into a single tree, reducing the potential by $5i - 5$. We can show that the sum of the ranks of the nodes in the resulting auxiliary tree is bounded by $4i - 4$, as follows: If i is a power of 2, then the auxiliary tree in the binary sense has a complete left subtree of size $i - 1$ and no right subtree. Since the number of nodes on descending levels of the binary tree doubles as subtree sizes are roughly halved, the sum of the ranks is

$$\log i + \sum_{0 \le j < \log i} 2^j \log\left(\frac{i}{2^j} - 1\right).$$

Now let $m = \log i$ and simplify. The sum is bounded by

$$\begin{aligned} m + m \sum_{0 \le j \le m} 2^j - \sum_{0 \le j \le m} j2^j \\ &= m + m(2^{m+1} - 1) - m2^{m+2} + (m+1)2^{m+1} - 2 \\ &= 2^{m+1} - 2 \\ &= 2i - 2 \end{aligned}$$

If i is not a power of 2, then we can append $\le i - 2$ extra dummy root nodes to the auxiliary area so that there are $2^{\lceil \log i \rceil}$ root nodes. It is straightforward to show by induction that the tree resulting from the multipass linking of the auxiliary area without the dummy nodes can be "embedded" in the tree resulting from the multipass linking of the auxiliary area with the dummy nodes. By the analysis given above for the case when i is a power of 2, the sum of the ranks is bounded by $2(2i-2) - 2 + 2\delta_{i=1} = 4i - 6 + 2\delta_{i=1} \le 4i - 4$. (The Kronecker delta $\delta_{i=1}$ denotes 1 if $i = 1$ and 0 otherwise.) We can get a better bound by considering the contribution of the dummy nodes, but for our purposes this bound is adequate, since we are ignoring constant factors.

When the auxiliary tree is linked with the main tree (so that all the nodes are in the main tree and the auxiliary area is empty), one unit of work is expended, and the change in potential is bounded by $\log n + 4i - 4 - 5$, where n is the number of nodes in the priority queue. Next, the root is deleted via one unit of work, reducing the potential by $\log n$, and the resulting subtrees are recombined via the twopass scheme. If there are k subtrees, a total of $k - 1$ units of work are used to recombine; the resulting potential increase is shown in [Fredman et al, 86] to be bounded by $2 \log n - k + 3$. The total amount of work spent during the *delete_min* and the net change in potential can thus be bounded by $i + k$ and $2 \log n - i - k - 1$, respectively, which bounds the amortized time by $2 \log n - 1$.

The other case to consider is when the auxiliary area is empty at the time of a *delete_min*, that is, when $i = 0$. In that case, the reasoning in the last paragraph shows that the total amount of work and the net change in potential are bounded by k and $\log n - k + 3$, respectively. This completes the proof. ∎

The same approach combined with the analysis in [Fredman et al, 86] proves the following about auxiliary multipass:

Theorem 2. *The auxiliary multipass algorithm achieves amortized time bounds of* $O(1)$ *for insert and find_min and* $O\big((\log n \log\log n)/\log\log\log n\big)$ *for delete_min and delete if no decrease_key operations are allowed.*

Unfortunately, we cannot as yet extend either algorithm's analysis to include *decrease_key* nodes in the auxiliary area. The problem lies in the subtrees attached to nodes whose values are decreased. We are hopeful that some variant of this batching technique will prove that the $O(1)$-time amortized bounds for *insert* and *decrease_key* and the $O(\log n)$-time amortized bound for *delete_min* do hold. Note that we can get a weaker result by a slight modification of our algorithms. If the auxiliary area is batched whenever a *decrease_key* or *delete_min* is performed, then we get the desired bounds, except that *decrease_key* and *delete_min* use $O(\log n)$ time for auxiliary twopass and $O\big((\log n \log\log n)/\log\log\log n\big)$ time for auxiliary multipass.

5. Conclusions and Open Problems

The experimental data gathered from our simulations provide empirical evidence that the $O(1)$-time bounds for *insert*, *decrease_key*, and *find_min* and the $O(\log n)$-time bounds for *delete_min* and *delete* do hold in the amortized sense, where n is the size of the priority queue at the time of the operation. All the pairing heap methods performed well in our simulations. The auxiliary twopass variant clearly did the best. This result is satisfying because we have shown analytically that the auxiliary twopass algorithm achieves the above mentioned bounds, assuming that no *decrease_key* operations occur. Or if the auxiliary area is batched and merged with the main tree whenever a *decrease_key* or *delete_min* operation is done, then *insert* and *find_min* have $O(1)$-time bounds and all other operations have $O(\log n)$-time bounds in the amortized sense.

Proving amortized time bounds of $O(1)$ for *insert* and *decrease_key* and $O(\log n)$ for *delete_min* for some pairing heap implementation remains the major open problem. [Fredman et al, 86] has shown that an $O(\log n)$-time amortized bound can be

proven for all operations of the twopass variant. No multipass variant has been proven to achieve the same $O(\log n)$-time amortized bound for all operations; *delete_min* remains the primary stumbling block. Demonstrating this logarithmic time bound for all multipass operations is another interesting open problem.

References

A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA (1974).

M. R. Brown. Implementation and Analysis of Binomial Queue Algorithms. *SIAM Journal on Computing, 7* (August 1978), 298–319.

C. A. Crane. Linear Lists and Priority Queues as Balanced Binary Trees. Technical Report STAN–CS–72–259. Stanford University (February 1972).

M. L. Fredman and R. E. Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *Proc. 25th Annual Symposium on Foundations of Computer Science*, West Palm Beach, FL (October 1984), 338–344.

M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan. The Pairing Heap: A New Form of Self-Adjusting Heap. *Algorithmica, 1* (March 1986), 111–129.

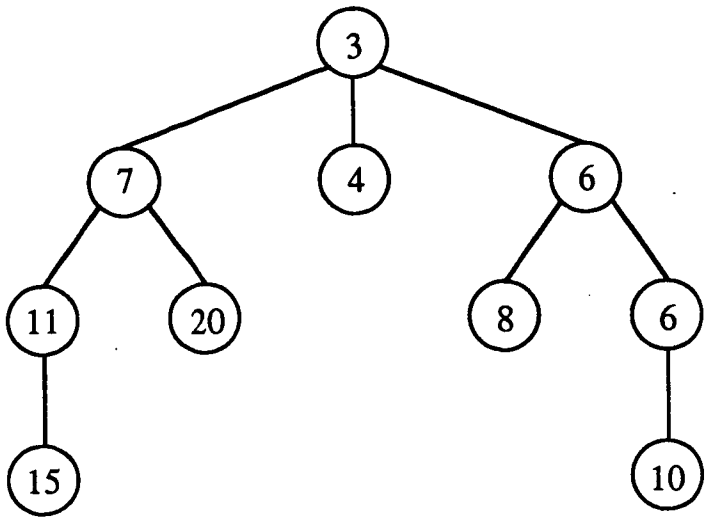
D. W. Jones. An Empirical Comparison of Priority Queue and Event Set Implementations. *Communications of the ACM, 29* (April 1986), 300–311.

D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching.* Addison-Wesley, Reading, MA (1973).

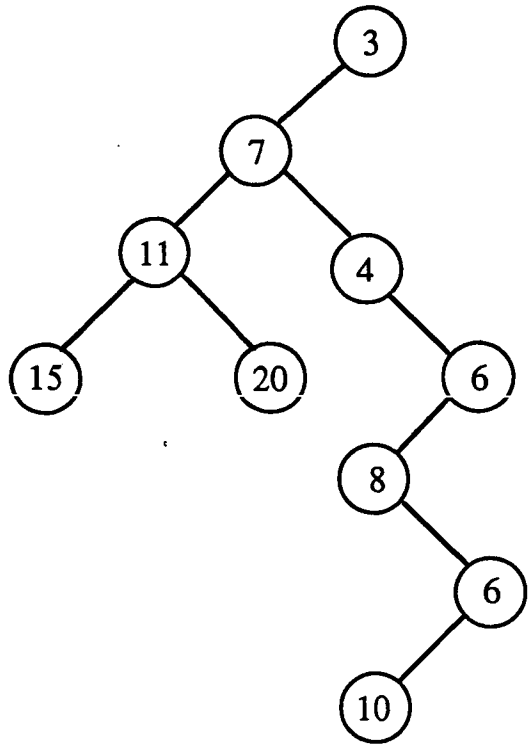
R. E. Tarjan. Amortized Computational Complexity. *SIAM J. Alg. Disc. Meth., 6* (April 1985), 306–318.

J. Vuillemin. A Data Structure for Manipulating Priority Queues. *Communications of the ACM, 21* (April 1978), 309–314.

J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM, 7* (June 1964), 347–348.



(a) multiway tree heap representation



(b) corresponding binary tree representation

Figure 1. An example of a heap ordered tree

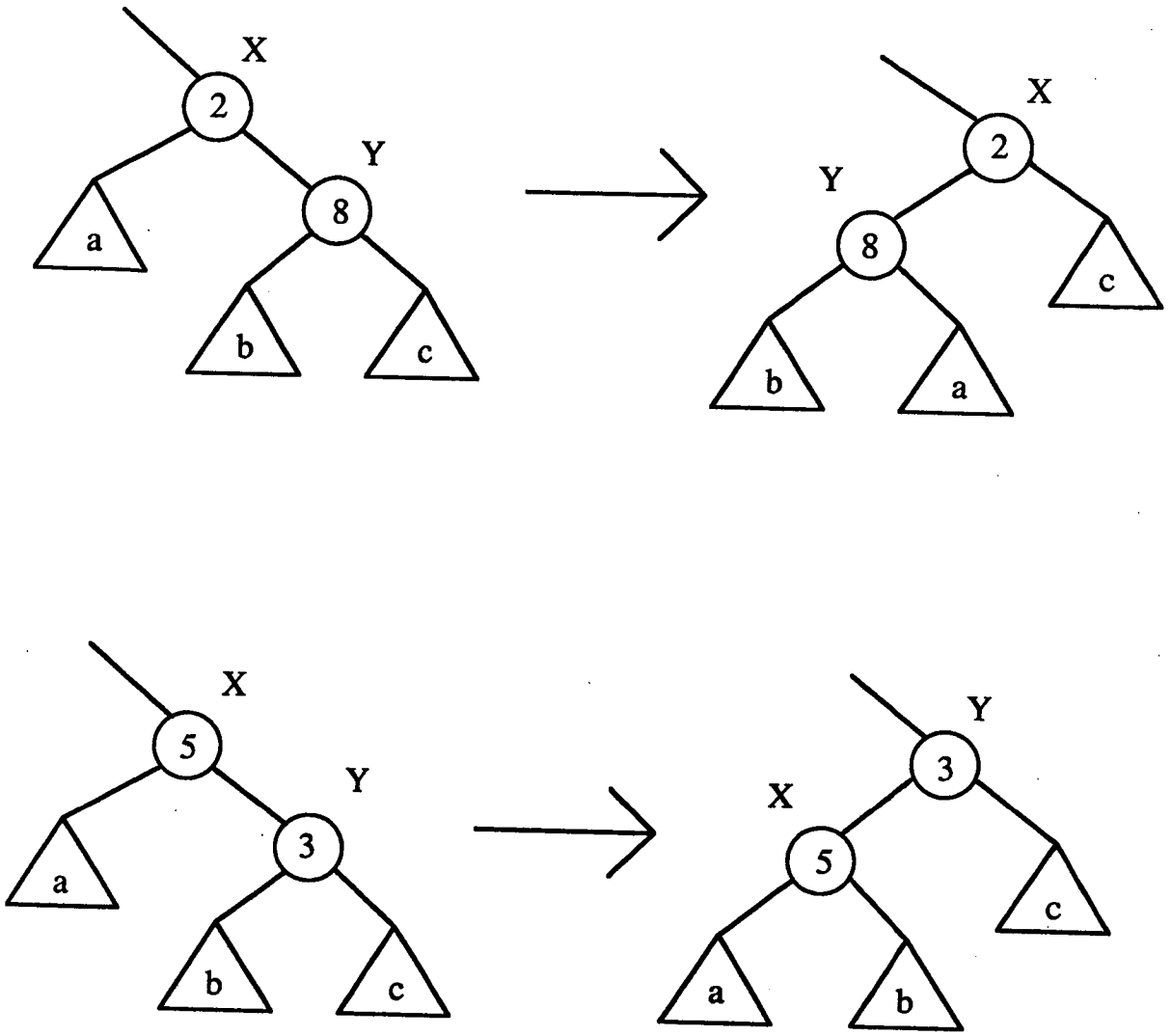
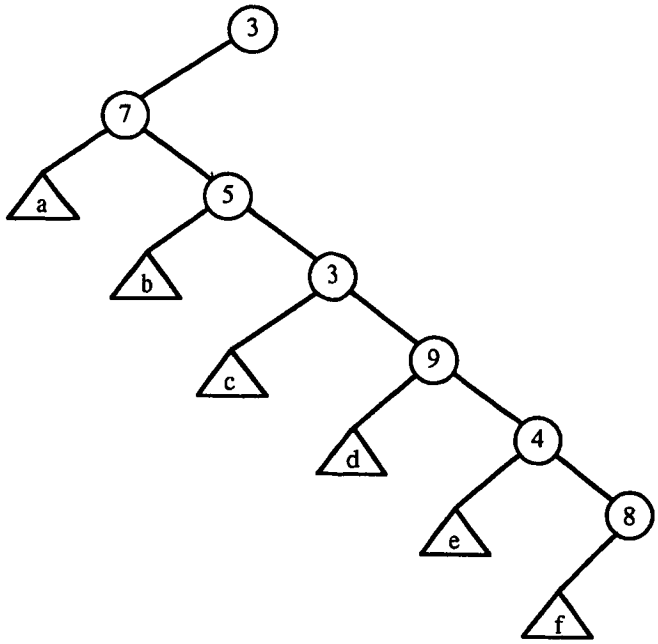
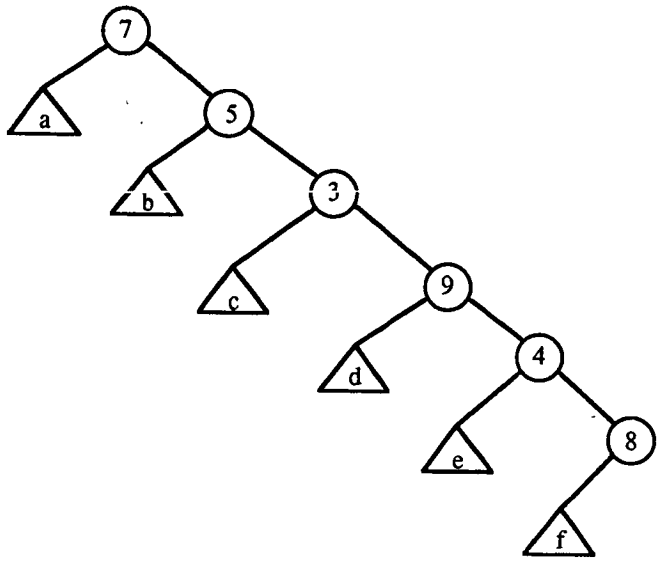


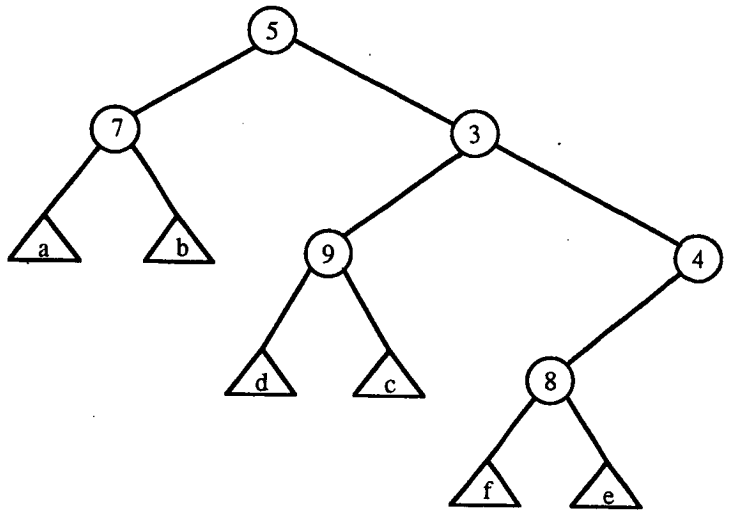
Figure 2. Two binary tree heap configurations and the resulting structures from a comparison-link between nodes X and Y



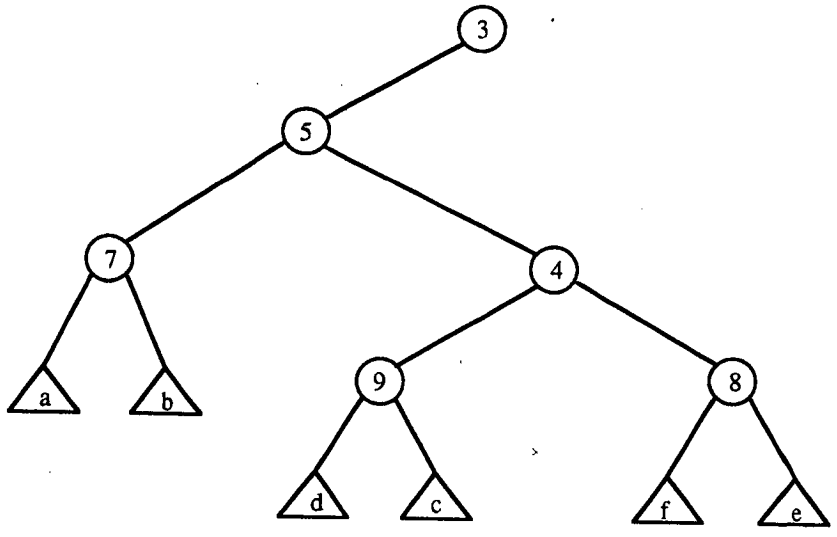
(a) beginning heap configuration



(b) root (minimum) deletion

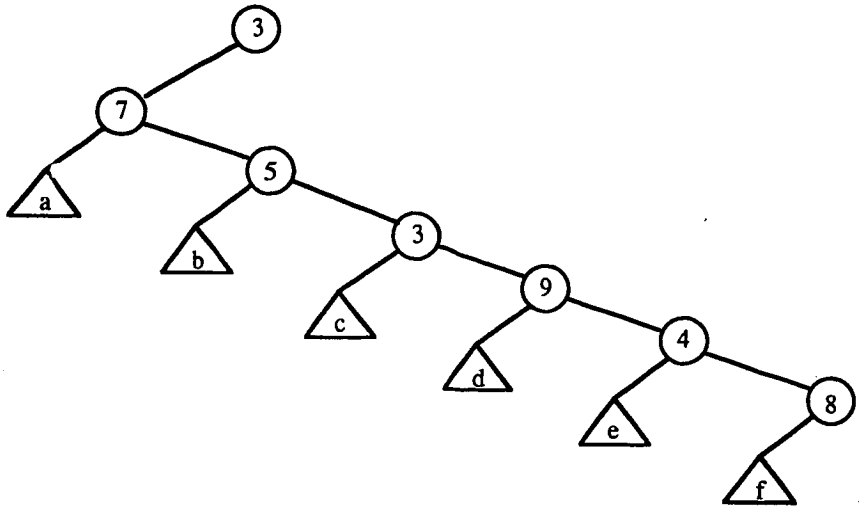


(c) pass one

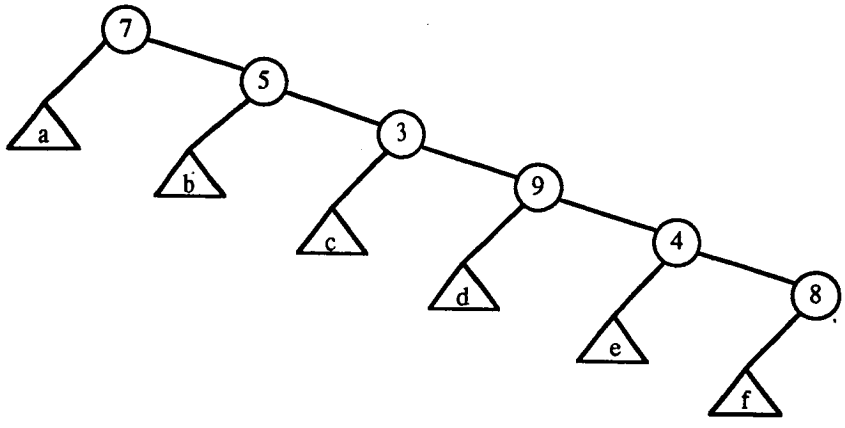


(d) pass two

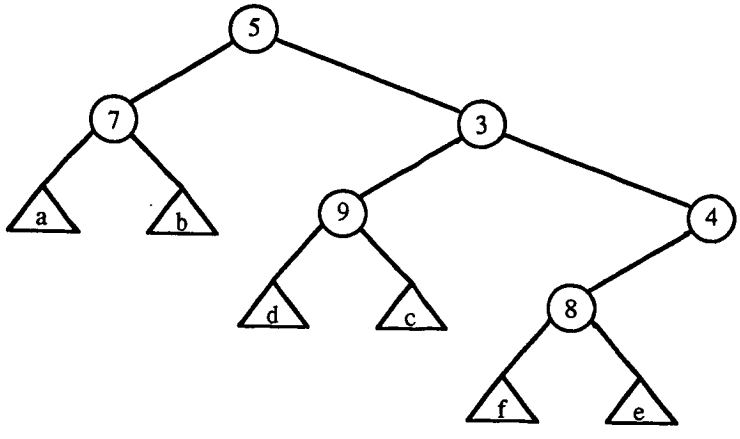
Figure 3. Twopass delete min procedure,
using the binary tree representation



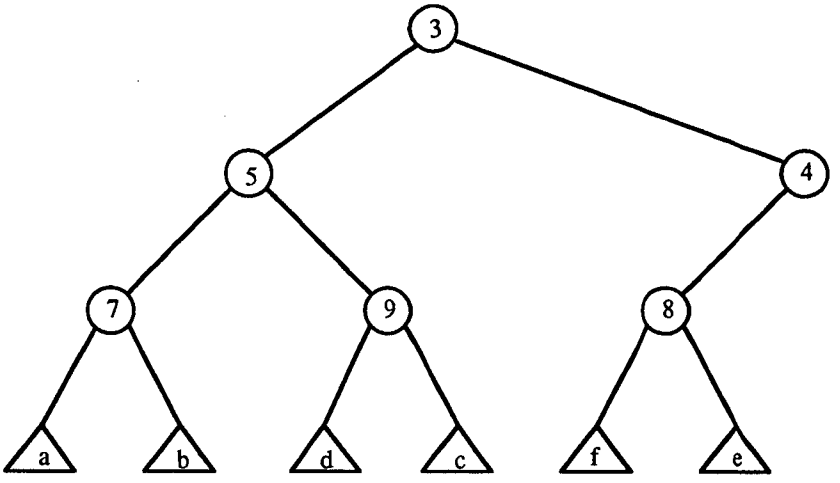
(a) beginning heap configuration



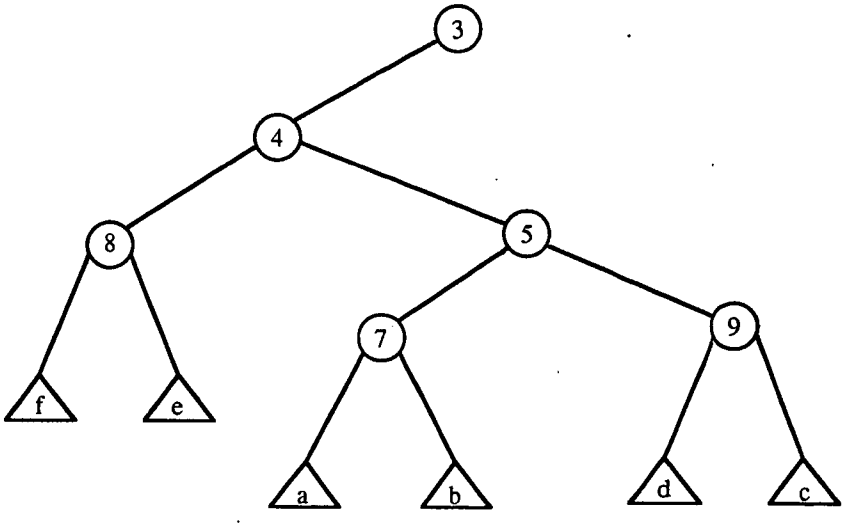
(b) root (minimum) deletion



(c) pass one

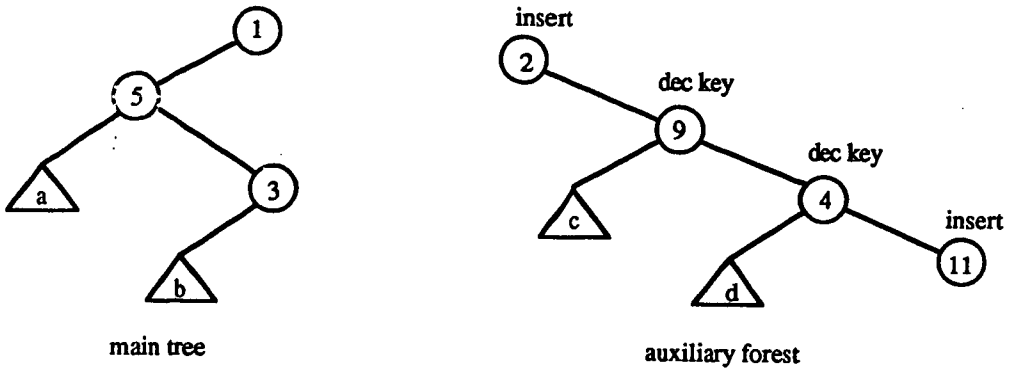


(d) pass two

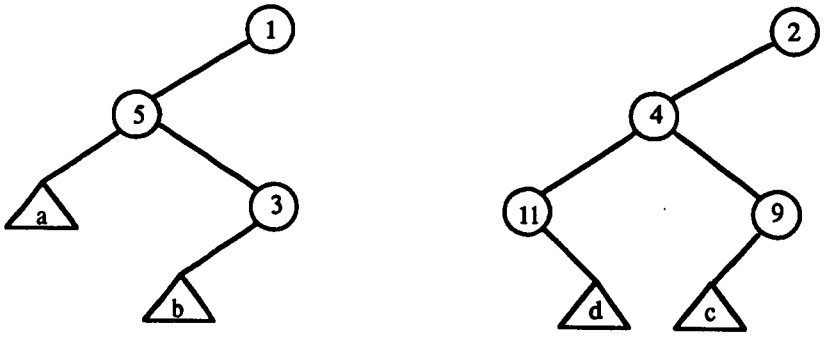


(e) pass three

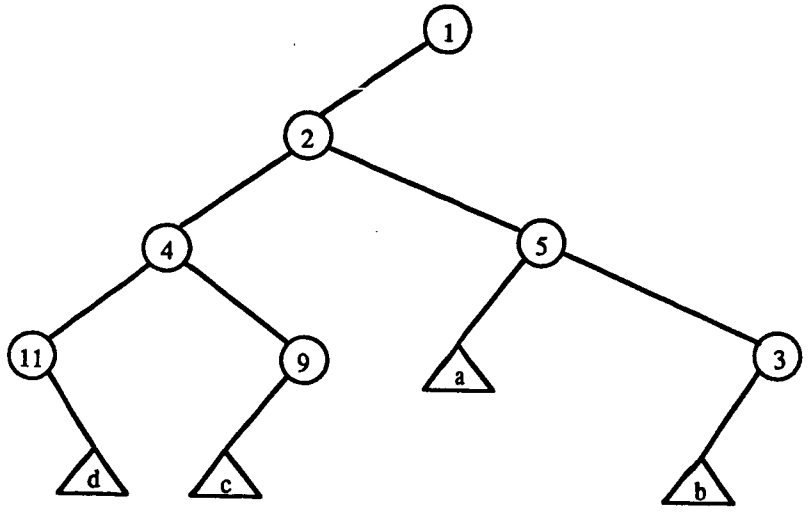
Figure 4. Multipass delete min procedure,
using the binary tree representation



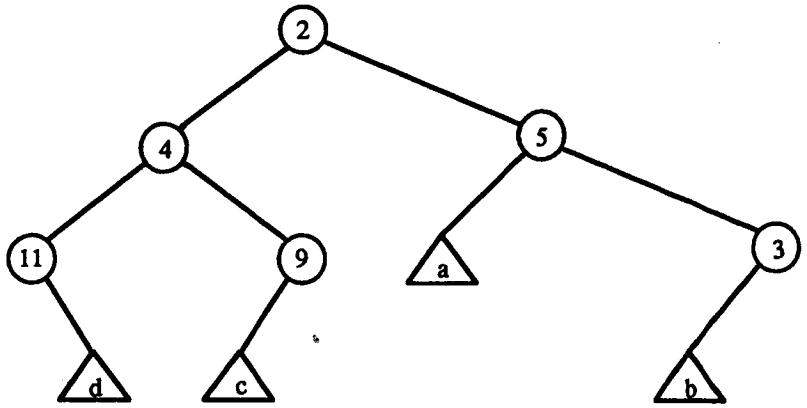
(a) initial heap configuration



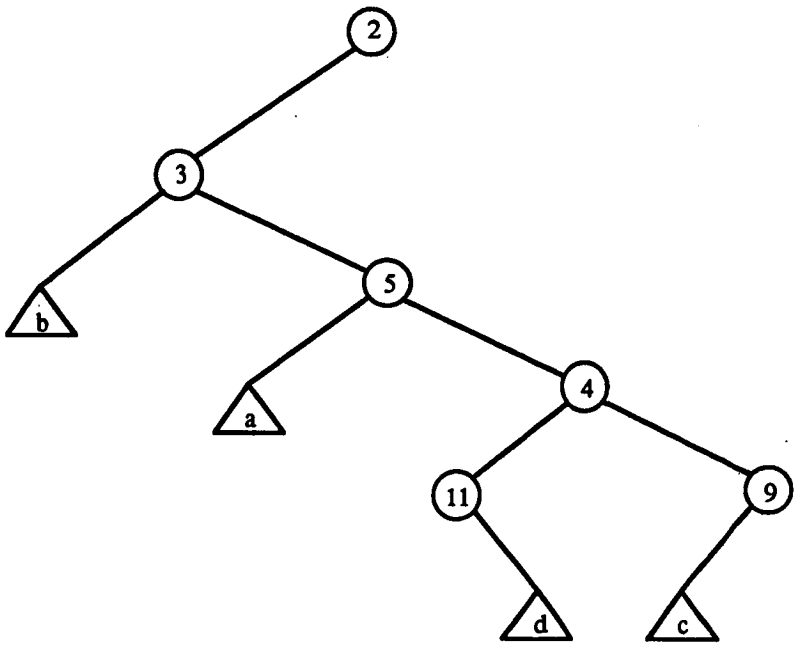
(b) multipass on auxiliary forest



(c) link auxiliary root to main root



(d) root (minimum) deletion



(e) twopass back to one root

Figure 5. Auxiliary twopass delete min procedure, using the binary tree representation

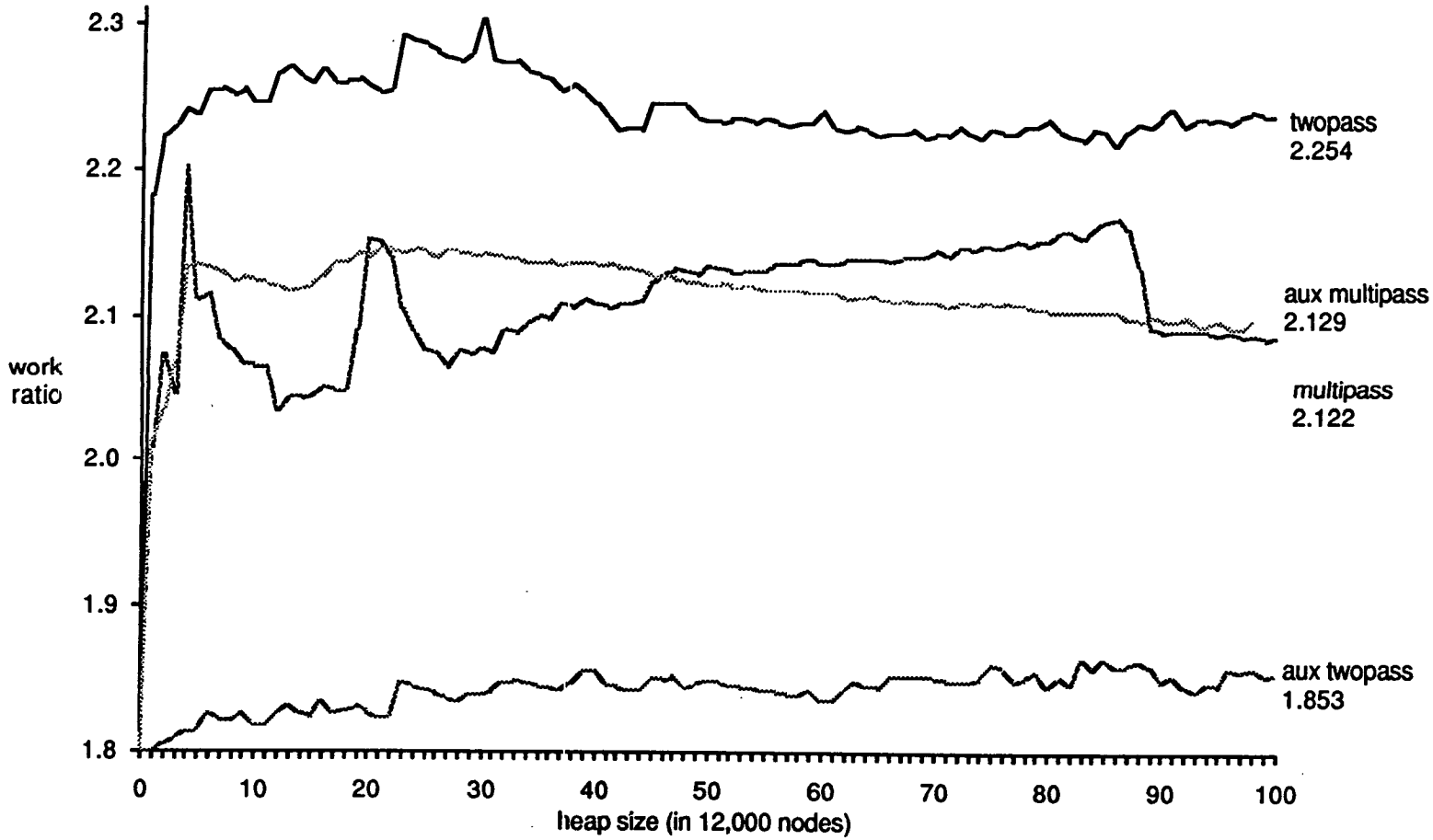


Figure 6. log n inserts, 1 delete min per phase

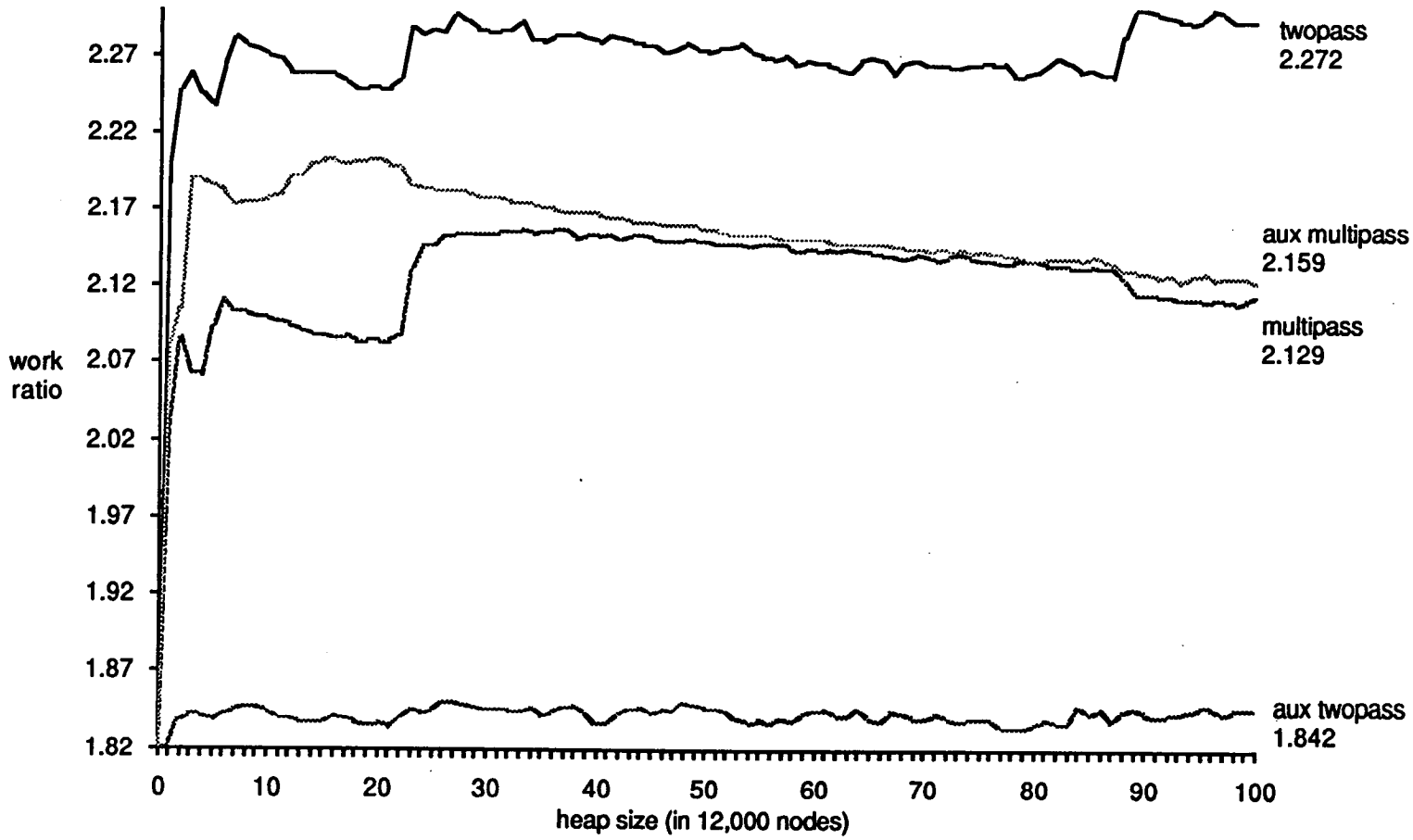


Figure 7. 0.5 log n <insert, random decrease key> pairs, 1 delete min per phase

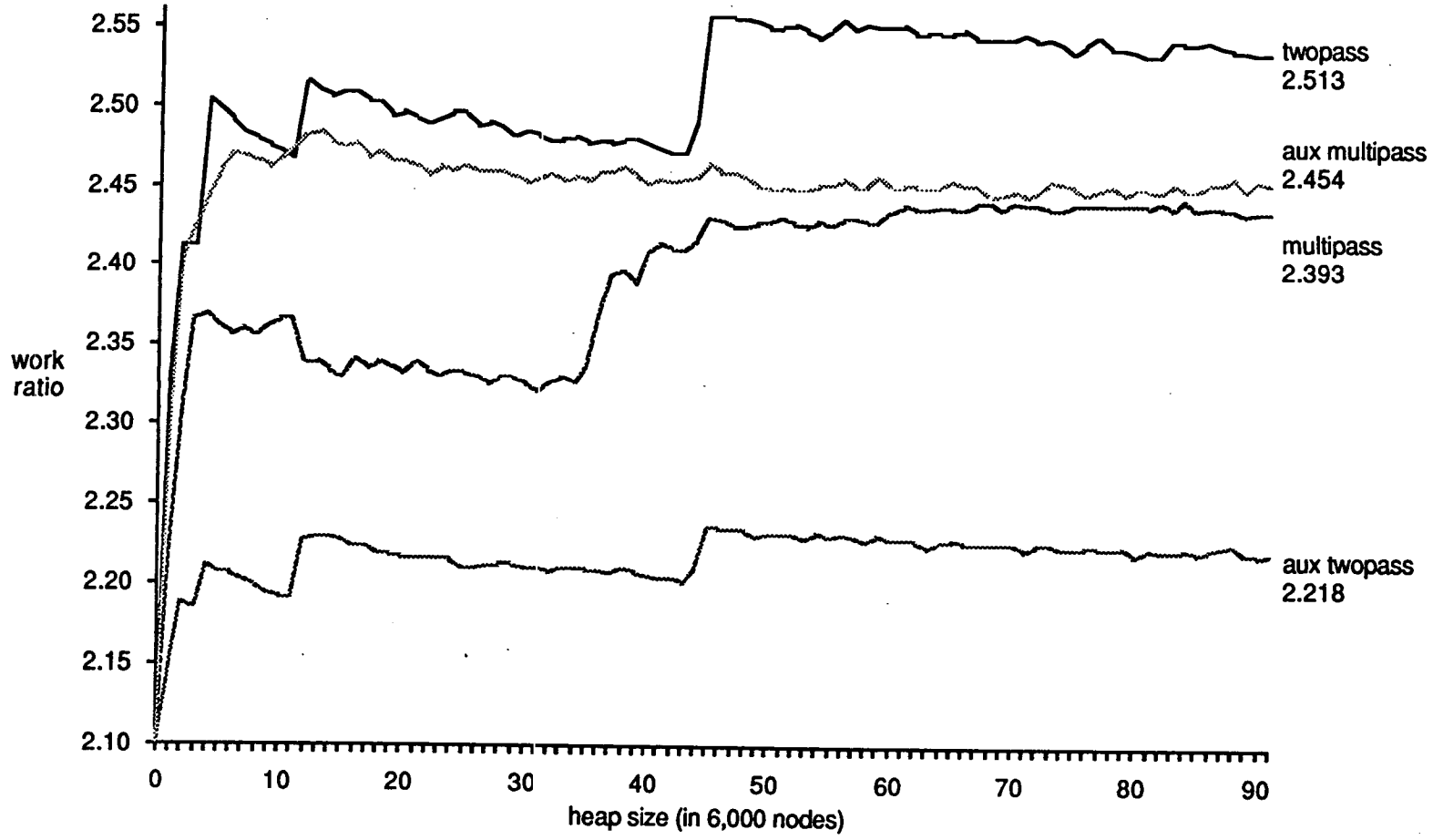


Figure 8. 0.5 log n <insert, greedy decrease key> pairs, 1 delete min per phase

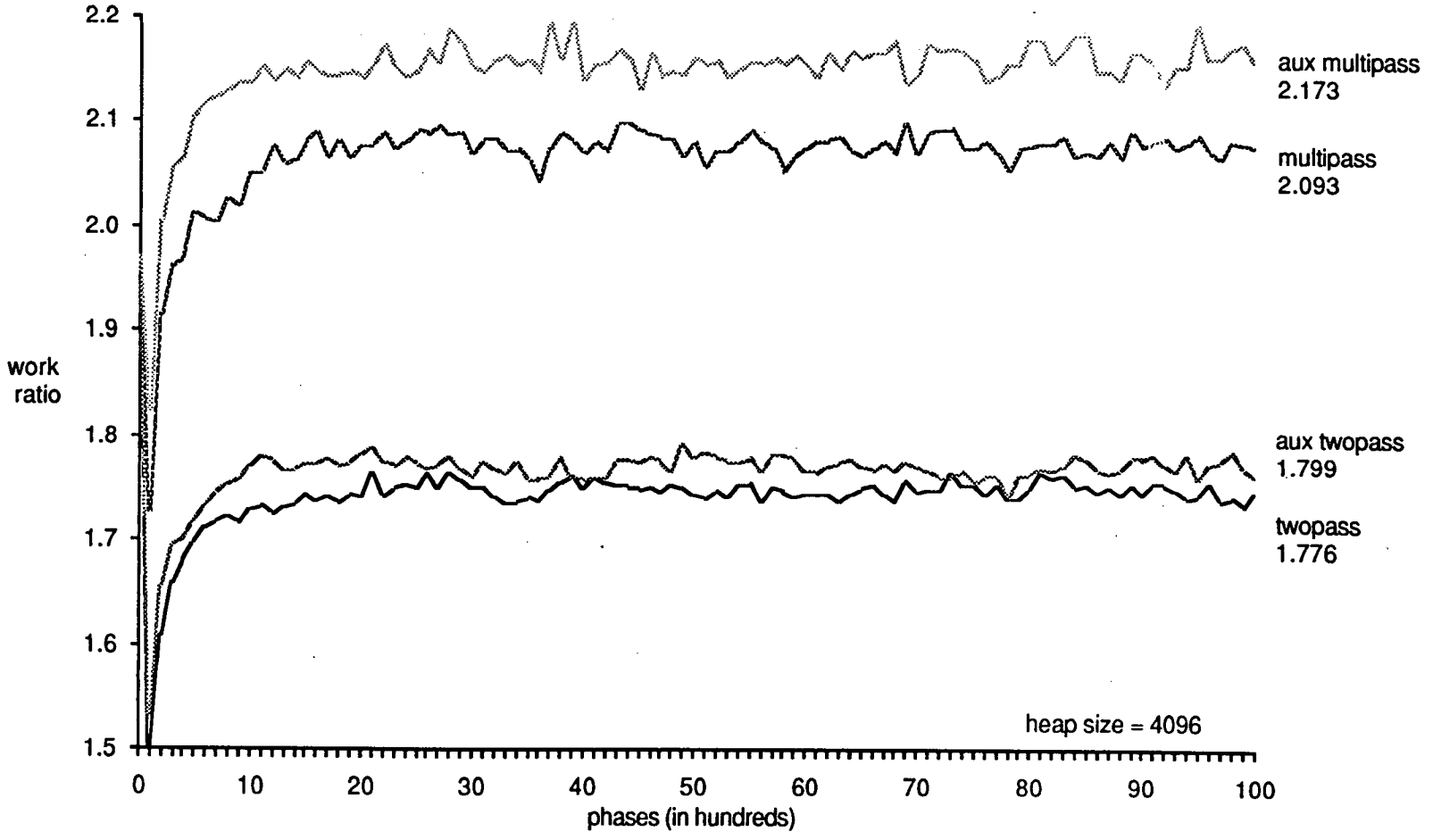


Figure 9. 1 insert, (0.25 logn) - 1 greedy decrease keys, 1 delete min per phase

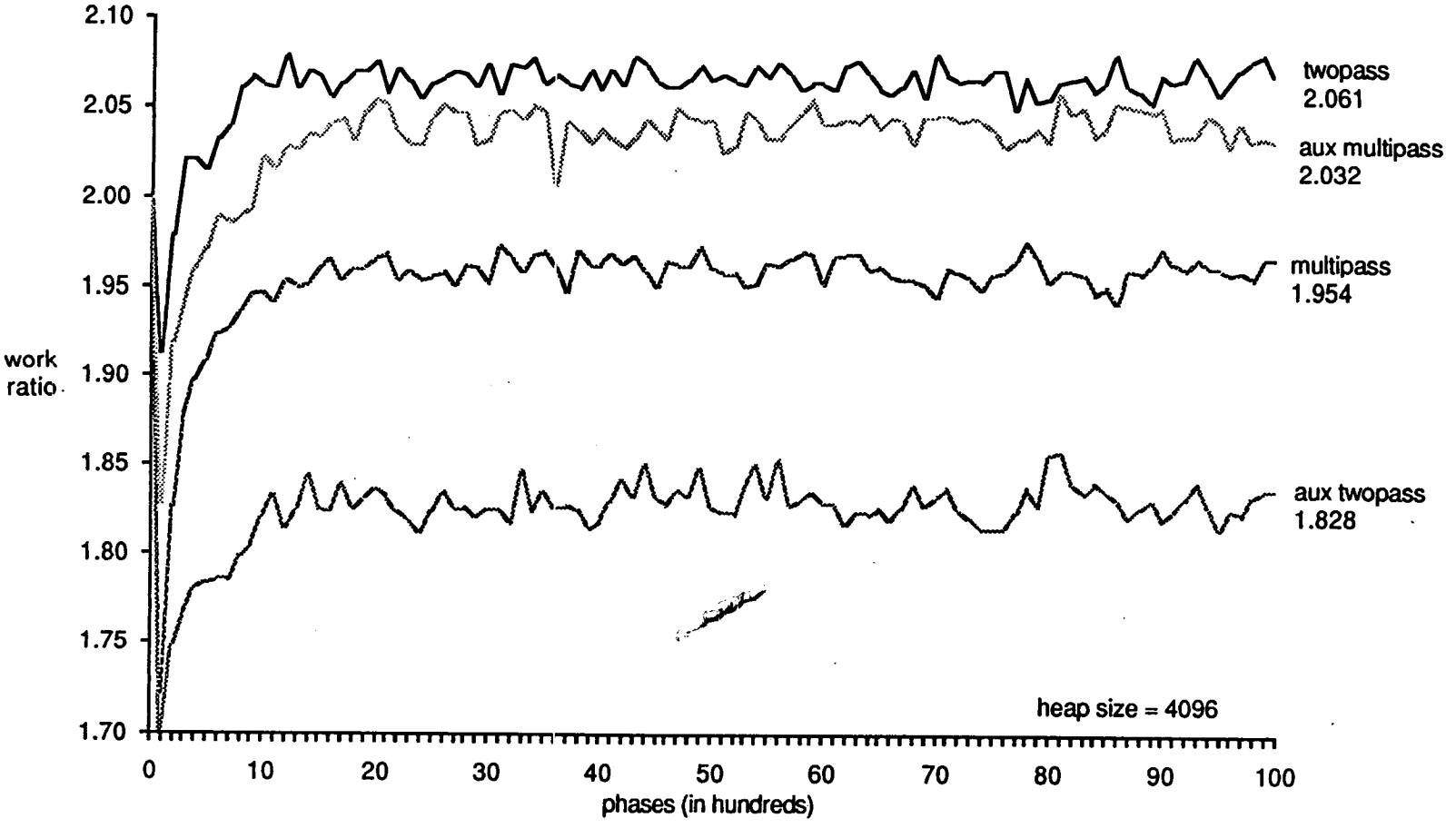


Figure 10. 1 insert, (log n) - 1 greedy decrease keys, 1 delete min per phase

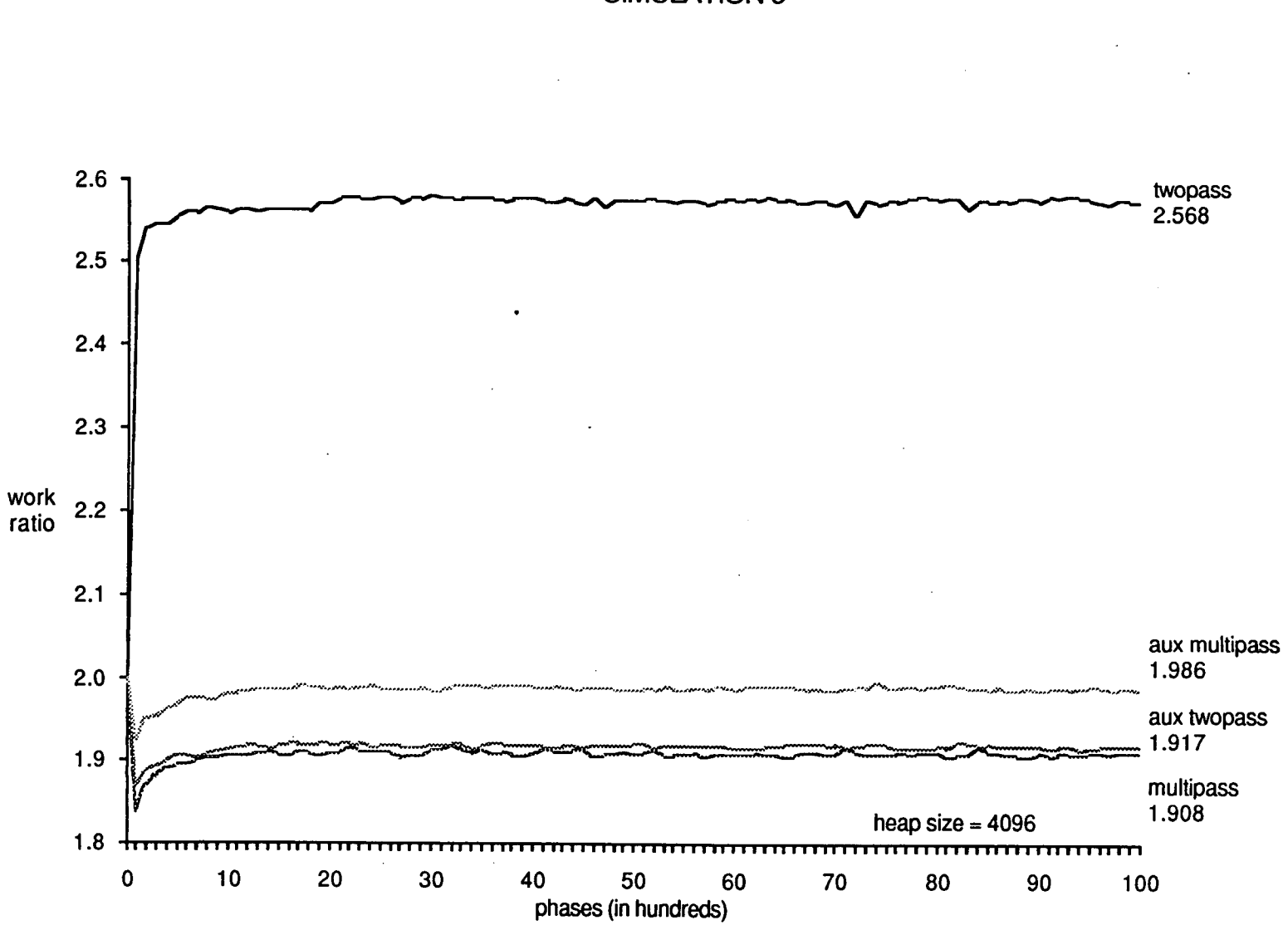


Figure 11. 1 insert, (4.0 log n) - 1 greedy decrease keys, 1 delete min per phase

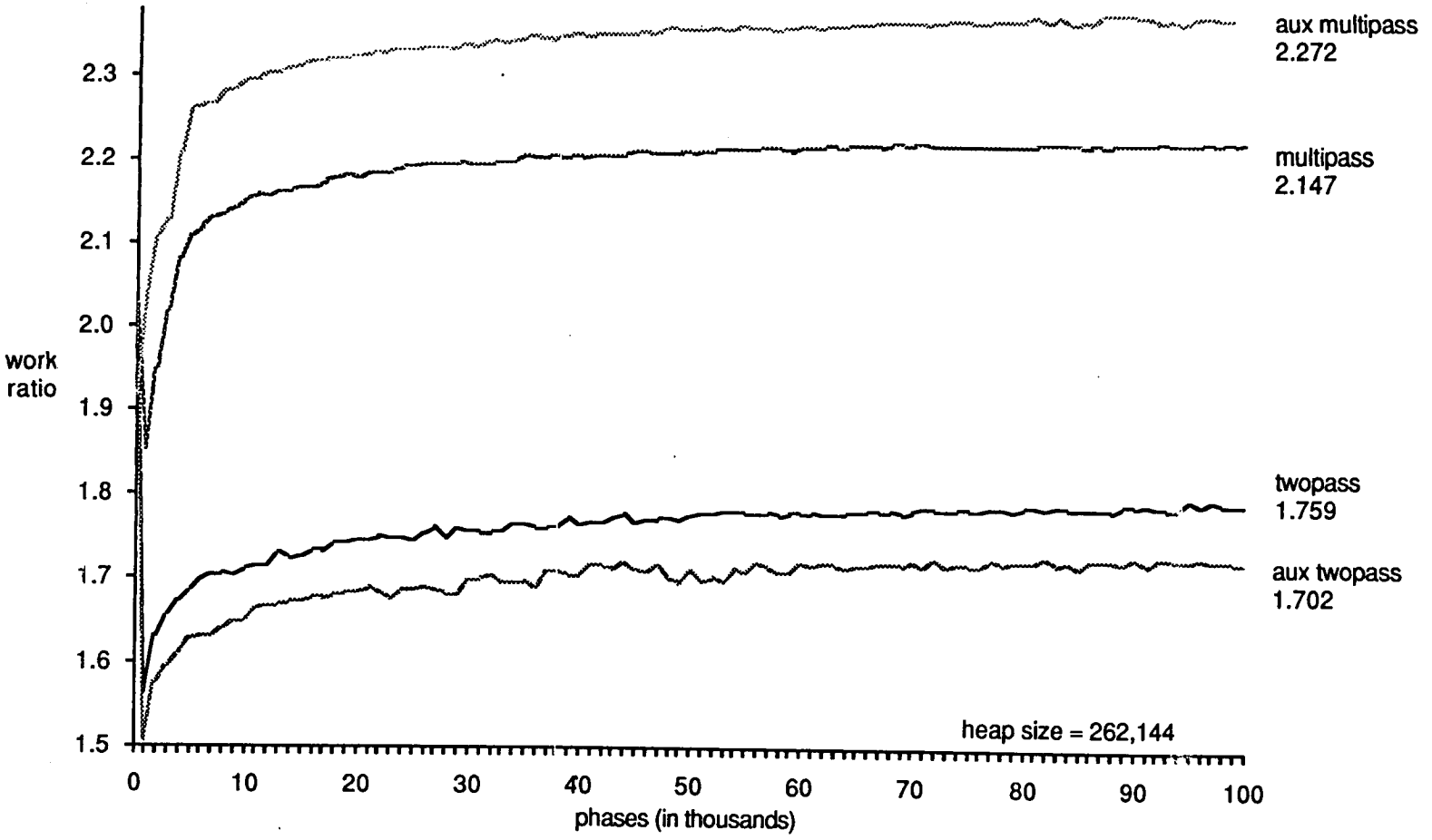


Figure 12. 1 insert, (0.22 logn) - 1 greedy decrease keys, 1 delete min per phase

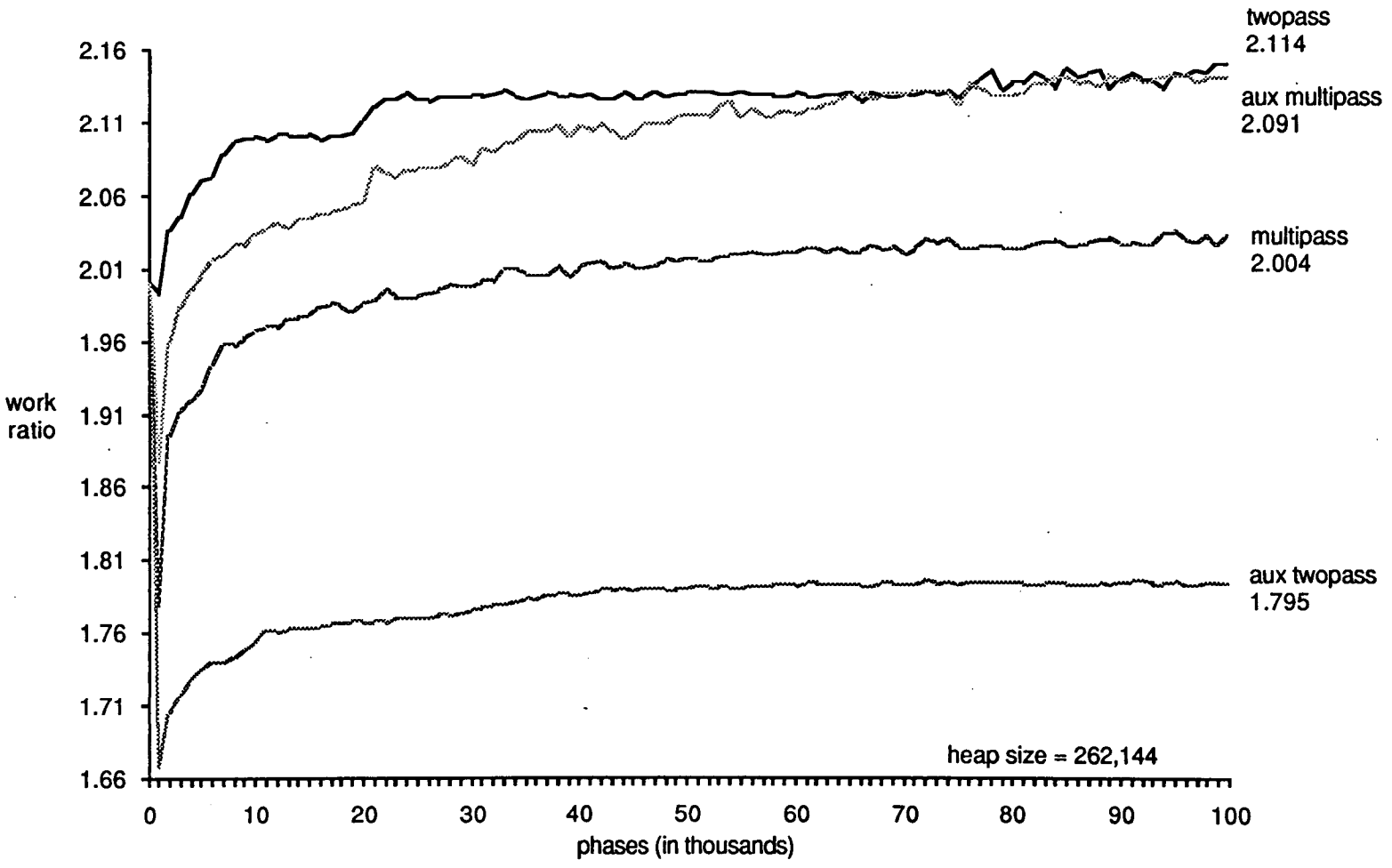


Figure 13. 1 insert, (log n) - 1 greedy decrease keys, 1 delete min per phase

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique

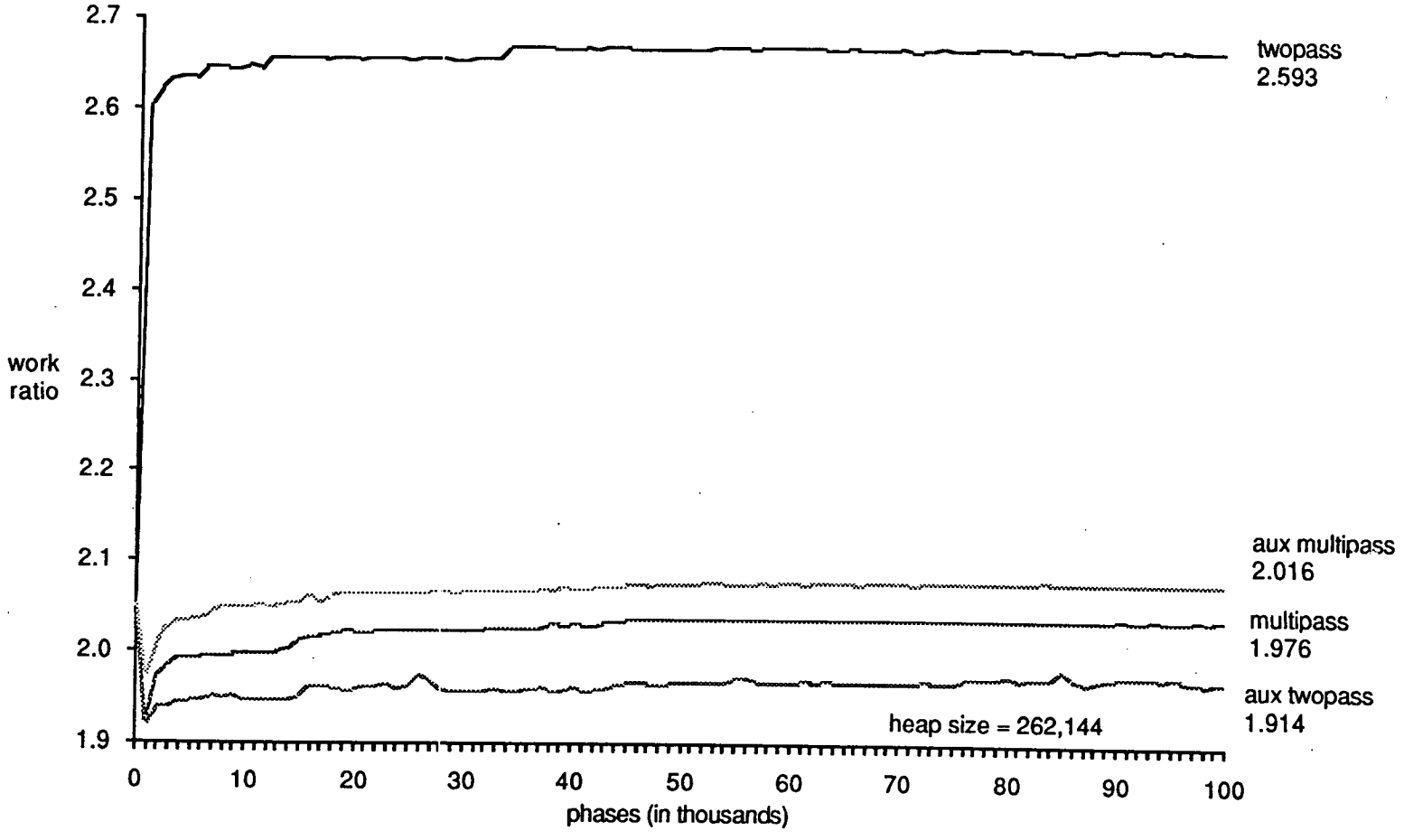


Figure 14. 1 insert, (4.0 log n) - 1 greedy decrease keys, 1 delete min per phase