Algorithmica (1992) 7: 415–427

Algorithmica

Application Experience Section

Three Priority Queue Applications Revisited[1]

Andrew M. Liao[2]

Abstract. Results indicate that the two recently introduced self-adjusting heaps are the most competitive choices for the applications considered. Further, the results indicate that only some heap structures support lazymerge/lazydelete operations well, partially confirming that algorithms based on top-down skew heap compare more favorably than those based on binomial queues, that there are strong grounds for believing the conjectured amortized time bounds for pairing heap operations, and that pairing heaps are a competitive alternative to Fibonacci heaps.

Key Words. Heaps, Priority queues, Data structures, Applications, Sorting, Graphs, Performance.

1. Introduction. Although much work has been done to determine the performance effectiveness of priority queues (or "heaps"), this study reconsiders heaps under three actual applications. The applications considered are the minimum spanning tree, shortest path, and heapsort problems. The applications were chosen because of the author's belief that these three problems would serve well as paradigmatic heap applications. Aside from this paper, there has been only one other similar endeavor considering nearly optimal steiner trees undertaken by deCavahlo [3].

Currently, much of the literature on heap performances seems to focus on event-set simulation. Recent results by Jones [7] are insightful, but the contention is that those results do not adequately portray the true effectiveness of a given heap representation. It is generally not the case that event simulation application results will be comparable with graph application results. Further, such results tend to be very misleading. Some of the empirical results presented here by the author and independently observed by deCarvahlo give grounds for this remark [3], [8].

Besides determining the appropriate/effective heap for each application in the study, this empirical investigation was to determine how well top-down skew heaps performed relative to leftist and binomial heaps, and to determine the effectiveness of the pairing heap as a practical heap representation. In addition, this study might

[1] This work was carried out during the author's final graduate year (1987) at Rensselaer Polytechnic Institute. Though the author is currently with Thomson Financial Networks, this is an unaffiliated paper.

[2] Thomson Financial Networks, 85 Wells Avenue, Newton, MA 02159, USA.

Received September 22, 1988; revised August 9, 1989, and June 18, 1990. Communicated by D. T. Lee.

shed some light on the effectiveness of the self-adjusting heuristic for heaps. The five heaps studied are: leftist trees [10], binomial queues [18], top-down skew heaps [13], [15], Fibonacci heaps [6], and the one pass variant of pairing heaps [5] ("LHeap," "BHeap," "SHeap," "FHeap," and "PHeap," respectively) and the algorithms selected are Prim [16] and Cheriton–Tarjan round-robin minimum spanning tree algorithms [2], the Dijkstra shortest path algorithm [16], and the heapsort algorithm [10].

A motivation for this work stems from the dissatisfaction with heap efficiency investigations based upon alternating sequences of insert and deletemin operations upon a k-node heap. It may be necessary to merge arbitrarily sized heaps or carry out other operations that are either intrinsic to an application or crucial to its efficiency. In the Cheriton–Tarjan algorithm, it is necessary to make use of "lazymerge," "lazydelete," merge, and deletemin operations. For the Prim and Dijkstra algorithms, the heap operations necessary consist of choosing a source vertex and its heap node, then an insert or decreasekey operation for every graph edge examined (adjacent to the current source vertex), and then a deletemin operation. Finally, heapsort is just a sequence of insert operations followed by an equal number of deletemin operations. In short, the study examines the practical efficiency of the various heap representations in a more general setting.

While the study tested each heap representation under the heapsort and Cheriton–Tarjan algorithms, only the FHeaps and PHeaps were tested under the Prim and Dijkstra algorithms since only these two heaps were designed to support the "decreasekey" operation. A decreasekey could have been implemented for the other heap representations but that would intuitively require the use of the deletemin and merge routines and an extra pointer or two. The remainder of this paper presents the development considerations (as far as hardware and language development choices and common software interfaces), the generation of the test data, measurement methodology, implementation details of the applications, the resulting test data, and the conclusions reached.

2. Development Considerations. This section discusses the issues in developing the necessary tests in terms of the choice of machines and development languages. Pascal was selected as the development language for this investigation because of its variety of data-structuring primitives as well as the modularity and structure of its language constructs. In an attempt to show that the performance effectiveness results were generally independent of machine architecture over a broad class of architectures, an IBM PCXT (configured with 512K of main memory and no cache or disk page swapping facilities), a SUN-2 (configured with 2 Meg of main memory and an 8K cache for instructions and data), and a dual processor IBM 3081D (each processor had a private 32K cache; 6 Meg of user memory [4]) were selected.

Though the performance results will be affected to some extent by the presence of caches on the SUN-2 and the IBM 3081D—an unavoidable situation—the IBM PC is a "stripped-down, no-frills" machine whose results should provide an expectation of what the results from the other two machines might be like. Given the selected hardware, the study made use of the Microsoft Pascal (IBM PC), the

SUN Berkeley Pascal (SUN-2), and the Pascal-JB (IBM 3081D) compilers with the optimization switches enabled.

By writing all of the code to conform to a common calling convention and eliminating all recursion, a controlled situation was created for this investigation. This decision made it easier to demonstrate that the performance differences among versions of a given application were generally due to the heap used. Small supporting heap subroutines were expanded in-line unless the subroutine was considered a basic primitive (e.g., the BHeap and FHeap linking steps or the SHeap and LHeap meld primitives). The decision as to which "minor" subroutines were to be expanded in-line was made based on some empirical tests. Some of these development decisions enabled the author to develop the necessary variations of the applications using the different heaps, and helped ensure the independency characteristics of the eventual results.

One other development issue that must be addressed concerns the input data for the graph applications and the graph model used for the investigation. The study considered randomly generated graphs where the input parameters were the number of vertices, the random number of edges in the graph, and the random selection of edges (and their random weights). The emphasis was not on what specific edges were to exist in the graph but that there would be a random number of edges in the graph and that each edge had a random positive nonzero weight, though one of the possible considerations was to generate three different graphs (for each vertex size) of fixed increasing densities. One of the reasons for the decision was to avoid an objection that the results were "tailored" to a particular set of graph densities. Another reason was to provide unpredictable situations for the various heaps used in implementing the various incarnations of the graph algorithms used in the investigation.

To establish confidence that the implementations for this work were correct, various display traversals were used to confirm that heap invariants were maintained during various execution states as well as assuring that no nodes were lost and no extra nodes were added. Other methods used to ensure correctness were informal correctness proofs on key sections of various heap subroutines. A check of the output from the four application algorithm groups was also made to ensure that the resulting answers were correct.

3. Generation of Random Data. The data testing the verious heaps under the three applications was based on a linear congruential pseudorandom number generator that returned a value in the interval (0:1). The pseudorandom number generator was based on the summary by Knuth [9] and Sedgewick [12] and the word size for the Intel 8088 processor. This generator has at least passed the chi-squared test over 100 bins. (The testing program was based on the code presented by Sedgewick. The criteria are also essentially that presented in the text [12].)

The heapsort program was designed to build an n-item heap, empty it, and repeat the process with a heap of the next larger size until finishing with the maximum size data set. Based on each test stage data-set size, n integers were

Table 1

Undirected graphs					
Set 1		Set 2		Set 3	
Node	Edge	Node	Edge	Node	Edge
10	45	10	44	10	34
20	128	20	172	20	170
30	400	30	426	30	335
40	457	40	253	40	628
50	1108	50	567	50	1064
60	1386	60	1169	60	618
70	1236	70	2158	70	1825
80	1852	80	1617	80	903
90	2092	90	1696	90	3719
100	4055	100	2026	100	1414
Directed graphs					
10	40	10	33	10	45
20	118	20	88	20	139
30	153	30	189	30	277
40	685	40	725	40	331
50	640	50	926	50	1199
60	356	60	1425	60	994
70	2397	70	490	70	2120
80	992	80	3136	80	2544
90	2134	90	3515	90	1738
100	2043	100	1973	100	3932

generated over the interval $(1:n)$ using the pseudorandom number generator previously mentioned. To initialize the program correctly and still have the ability to duplicate the generation of heapsort data sets, there were off-line files with a single pseudorandomly generated value that would serve as a seed to generate the necessary pseudorandom values.

Generating the pseudorandom undirected graphs was a bit more difficult since one of the issues centered on the selection of random graphs. In essence, a graph was generated based on the number of vertices—specifically, the number of edges for any given graph of n-vertices was a pseudorandomly generated value between $n \operatorname{Lg} n$ and $n*(n-1)/2$. While avoiding duplicate edges, random edges were chosen to exist. Each generated edge was given a randomly generated weight value (over the interval $(1:n)$ with n = number of vertices) that had an equal chance of occurring at any time. This study did not consider multigraphs and disallowed edges of the form (v, v). Finally, the input graph data was generated off-line and stored in disk files for testing with the various graph applications programs using different heaps. This decision was based on the view that the graph application algorithm should strictly compute the problem at hand and not add code to generate the necessary random data in the same program.

4. Measurement Methodology. The maximum problem size that a PC could handle limited the choice of data sizes. It would be of little value to look at more than hypothetical IBM 3081D results whose problem size would be too large to be run on a PC or a SUN. Thus, the largest graph problems were those, given a set of vertices, whose potential clique would still fit in on a 512K IBM PC.

Since the heapsort problem used an apparently uniform pseudorandom data distribution, it was sufficient to average three different times from three different runs on the same data size with each run based on a different pseudorandomly chosen seed (see Section 3). As a result, the key contributing factor to the behavior in the performance (run times) is the actual data size.

Now we must consider measuring the results from the applications dealing with graphs. Since there are too many variables in the configurations of the random graphs it is unreasonable to do any time averaging. In the case of heapsort we were dealing with a set of singletons, whereas in the graphs a given graph of n-vertices may have any number of edges, and the vertex degree configurations may be different between any two n-vertex graphs. Thus we have no fixed structure to categorize three sets of tests into one and we must do three sets of comparisons and then correlate the results.

What exactly was measured using the timers? Briefly, the timers were set up to clock compute bound processing with any I/O processing "fenced away" from the actual heart of each of the algorithms used. In the heapsort tests the timer calls were placed within the control loop automating the multiple heapsort test runs: the initial timer call was placed just before the heapsort algorithm and the terminating timer call placed just after the actual algorithm (but before the result of a particular run was output to a file). In the Cheriton–Tarjan algorithm a similar action was done in placing the initial timer call at the start of the main processing loop and the terminal timer call was placed just after the main processing loop (and before the result was output to a file). We must note that the time for heap insertions was NOT measured, since the calls to insert alternated with an I/O call to read a file of graph edges. Similar steps were coded for the Prim and Dijkstra algorithms.

To take time measurements, we called the PC-DOS time-of-day system routine (with the necessary conversions) onf the IBM PC and the Berkeley Pascal system routine "system" on the SUN. On the IBM 3081D the clockings required the set up and retrieval of the contents for the job problem state time (the necessary timer code was provided by Kupferschmid [11]). Of course, there are some restrictions on how and when each of the tests (for each machine) could be done. On the IBM PC there are no restrictions since there is no multiprogramming/multiprocessing under the version of PC-DOS (v 2.0) being used. Thus, for tests with no I/O processing, the elapsed time is directly comparable with CPU time. On the IBM 3081D (under the MTS operating system) there are generally no restrictions as well since the values from the job problem state are (for all intents and purposes) unaffected by effects of time-sharing (though it is still affected by a cache flush) [11]. In the case of the SUN the tests had to be run when there were no other users (ascertained by a Unix WHO) to minimize the measurement biases due to interrupts, context switches, or actions due to memory management.

There is one matter worth noting where "fair" timing is concerned. Since there are no decreasekey operations in the heapsort and the Cheriton–Tarjan minimum spanning tree algorithms, the code manipulating the necessary FHeap and PHeap parent pointers (and in the case of FHeaps, the MARK field operations as well) was essentially taken out. Remembering the maximum rank index used during the FHeap deletemin (tree-rank sorting) step also proved effective in limiting the array scanning needed in rebuilding the FHeap.

5. Implementation Remarks. Heapsort was the easiest of the four applications to implement since it was basically a contiguous sequence of heap insertion operations followed by an equal number of deletemin operations. Automating the test over increasing data sizes simply meant nesting the heapsort algorithm in a loop ranging from the minimum data size to the maximum data size in increments of 50. In the IBM 3081D and SUN tests the data-set sizes ranged from (100:3300). However, the data range had to be limited to (100:2900) on the PC due to the limited default data address space on the PC.

In contrast to heapsort, the Cheriton–Tarjan algorithm was the most complex of the applications to implement and required three major data structures: an array set of vertices, edge heaps (one associated with every vertex having edges incident to it), and a circular list (with each node associated with a vertex) upon which the edge heaps are enqueued. Besides the merge, findmin, insert, and deletemin, the algorithm employs lazymerge and lazydelete operations. Except for routines maintaining a heap representation, much of the implementation remains fixed. The nonheap routines differing between versions are due to the configuration of the heap representation employed. Lazymerge is a "heap-concatenation" operation that enqueues a heap to a circular list node for processing at some later time. In the case of BHeaps, FHeaps, LHeaps, and SHeaps, we need to maintain an exogenous list structure whose list nodes point to a heap and another list node. In the PHeap variation we use the empty right sibling pointer in the root-level heap nodes to form an endogenous list structure supporting the lazymerge and lazydelete operations (This is infeasible with nodes of the other heaps since those root nodes generally employ all of their subtree pointers.) The breadth-first heap purging process of lazydelete is easy to code with the availability of a merge operation. A heap is dequeued from the process queue formed by the lazymerge operation(s). If the heap obtained has a useless root, we requeue its immediate subtrees and dispose of the root, otherwise we place the dequeued heap onto a "live/useful" process queue. This "purging" continues until the original process queue is empty. Lazydelete finishes with a pairwise multipass merge process by dequeueing two heaps off the "live/useful" process queue, merging them, and requeuing the result. This last step repeats until only one heap remains.

In addition to the merge, findmin, insert, and deletemin heap operations, the Dijkstra and Prim algorithms both require the use of the decreasekey operation. Assuming direct access to a heap node in question, decreasekey is an extraction of a heap subtree rooted at the node whose key value is to be lowered, a lowering of the key value, and a merge of the heap subtree and the heap from which it

came. Tarjan and Fredman [5], [6] discuss the decreasekey operation for FHeaps and PHeaps.

Implementing Dijkstra's algorithm requires three key data structures: an array set of vertices (each array element contains a pointer to a heap node, a parent vertex field, and a status field indicating whether a vertex is unlabeled, labeled, or done), lists of adjacent edges, and single heap nodes (one per vertex). Like the two previous graph algorithms, Prim's algorithm requires three key data structures which are an array set of vertices (each array element contains a pointer to a heap node, a parent vertex field, a status field indicating whether a vertex is unlabeled, labeled, or done, and a pointer to the current minimum edge), lists of adjacent edges, and single heap nodes (one per vertex). Interestingly, Prim's algorithm is nearly identical to Dijkstra's shortest path algorithm and thus employs the same heap operations as Dijkstra's algorithm.

6. Test Results

6.1. Heapsort. The heapsort tests echoed the results of Jones with several notable exceptions. On all machines tested, SHeaps clearly outperformed BHeaps. Further, the BHeap time results (for all of its complex merge code) was, at best, 106% of the SHeap time on the PC, as well on the SUN, and at best 100.4% of the SHeap time on the 3081D. Among the PHeap and SHeap test data, the observations are mostly within 2–3% of each other on each machine. As for LHeaps, the timed performance was within a range of 112–118% of the PHeap results on the PC, 112–115% on the SUN, and within the range of 134–142% of the PHeap times on the 3081D. Interestingly, FHeaps turned in the worst performance. Since the results among the three machines are essentially in agreement, Figure 1 for the

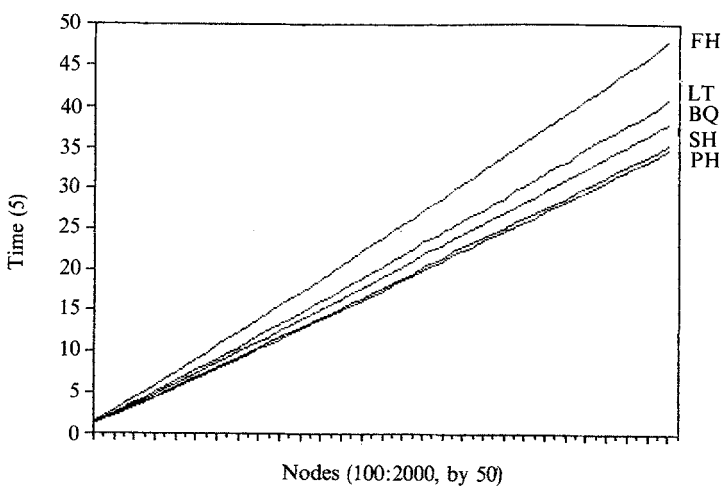

Fig. 1. PC Heapsort. BQ, binomial queue implementation; FH, Fibonacci heap implementation; LT, leftist tree implementation; PH, pairing heap implementation; SH, top-down skew heap implementation.

PC clearly illustrates the effectiveness of the various heaps under this application. The evidence and simplicity of implementation suggest that the self-adjusting heap structures are good choices for this application.

Surprisingly, SHeaps coded around the conceptual description rather than the implementation description [13], [15] actually ran slower than BHeaps on the 3081D. This is most likely because the procedural call costs on this architecture are proportionately more expensive than the PC or the SUN. Specifically, the IBM 370 instruction set does not support an intrinsic set of stack instructions while the Intel 8088 and Motorola 68010 do. As a result, proportionately more time must be spent on the 3081D simulating the runtime Pascal dynamic activation stack operations than on the PC and the SUN. Thus, the fewer procedural calls made on the 3081D implies a lower execution overhead.

6.2. Cheriton–Tarjan Minimum Spanning Tree Algorithm. Among the surprising results, LHeaps seemed to perform as well as BHeaps, if not better. Further, the algorithm with LHeaps did much better than the BHeap version on the PC and the SUN. Figure 2(a) and (b) illustrates an indication of this fact. Brown claims that a BHeap-based algorithm will always dominate LHeap-based ones [1]. The PC and SUN results seem to indicate otherwise. However, the IBM 3081D tests indicate that Brown's claim might still be true to some extent, yet somehow related to features of the hardware (particularly because of the 4K memory pages and the two private 32K caches). As is normally the case, the performances on the SUN and IBM 3081D will be affected by the behavior of the caches on the respective machines and on both the SUN and the IBM 3081D no effort is made to distinguish instructions from data in the cache(s). In the case of the IBM 3081D, there is a consideration worth noting that might explain the anomaly in the test result data between the BHeap-based implementation and the one based on LHeaps. In particular, generated code from the Pascal-JB compiler may be executed on and/or swapped to either of the IBM 3081D's two CPUs (exclusively). Now the timer code was implemented using an SVC call and avoided accounting the time for the swap between two processors [11]. However, it could not avoid accounting for the time necessary to load and flush the cache at the appropriate moments of execution [4], [11]. Thus much of the overhead could result from the time needed to do the necessary cache operations. The contradiction of Brown's claim can also be seen in the fact that findmin operations on LHeaps are $O(1)$ while it is $O(\text{Lg } n)$ for BHeaps which becomes especially important in considering the lazydelete step of the algorithm [2].

Another noteworthy result was that, for sufficiently large number of graph edges, the SHeap-based implementation clearly outperformed the BHeap version (i.e., "over a sufficient number of operations, SHeaps outperformed BHeaps"). Again, part of the reason is because findmin operations on SHeaps are $O(1)$ while it is $O(\text{Lg } n)$ for BHeaps. In addition, SHeaps need not worry about balance constraints, though pointer updatings could be expensive enough that a sufficiently large number of heap operations would be needed to make the algorithm "cost-effective." Unfortunately, results indicate FHeaps do not seem to be as cost-effective. Further, though FHeaps (without decreasekey or arbitrary delete

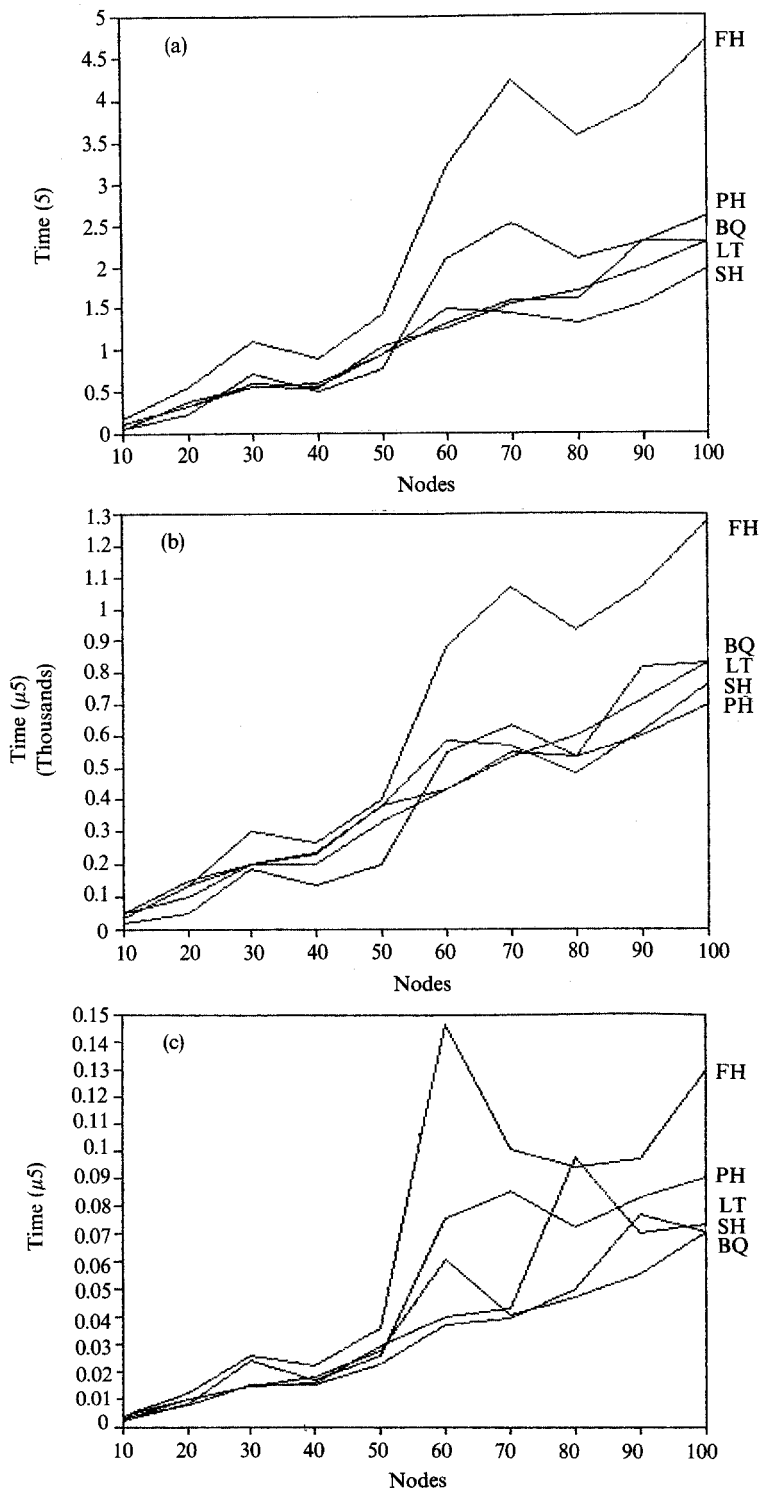

Fig. 2. Cheriton–Tarjan minimum spanning tree (graph set 2): (a) PC, (b) SUN, and (c) IBM 3081D. BQ, binomial queue implementation; FH, Fibonacci heap implementation; LT, leftist tree implementation; PH, pairing heap implementation; SH, top-down skew heap implementation.

operations) are "lazymerge" BHeaps, the expected performance with this data structure would not be much better.

Finally, the Cheriton–Tarjan algorithm with PHeaps seems to perform remarkably well for sufficiently sparse graphs. This is most likely due to the fact that the PHeap nodes can directly serve a dual role both as a heap node and a queue node simultaneously. Interestingly, the PHeap implementation seemed to do much better than it should have on the SUN in light of the IBM PC and IBM 3081D observations (see Figure 2). These results are no doubt perturbed by the effect of the SUN's cache behavior. However, the degradation in performance needs to be considered. It is probably the case that PHeap performance degrades because the structure has the property that the number of children of the root can be unbounded (recall the behavior of the merge routine).

6.3. Dijkstra's Shortest Path Algorithm. Now Dijkstra's algorithm calls an insert or decreasekey for every edge considered as a potential contributor to the resulting solution. Clearly, for FHeaps, these operations are $O(1)$ amortized time. For PHeaps we have an $O(\text{Lg } n)$ amortized time bound for such operations. Yet the test results indicated that the PHeap-based implementation clearly outperformed the FHeap version (like the heapsort tests, the results were unanimous and Figure 3 gives an idea just how well the PHeap version performed). Assuming that insert and decreasekey operations are $O(1)$ amortized time, the two dominant operations are deletemin (which is called $n - 1$ times, once per stage) and a linear-time examination (in total) of all the graph edges (quadratic time from the perspective of the number of vertices). As far as the results go, a majority of the time ratios between the two different implementations range between 1.4 and 1.6 on all three machines tested. The ratios outside this range could be due to the potentially large

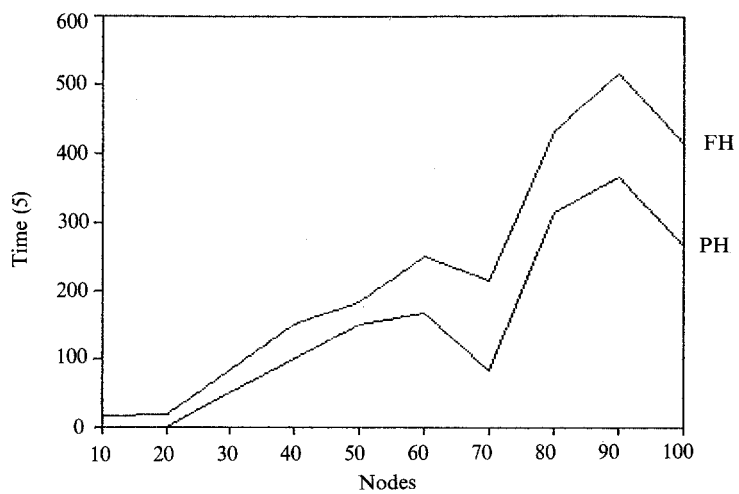

Fig. 3. SUN Dijkstra (graph set 2). FH, Fibonacci heap implementation; PH, pairing heap implementation.

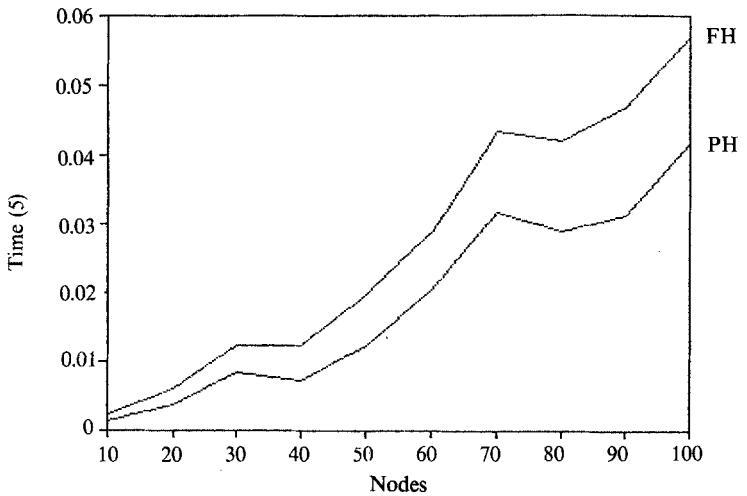

Fig. 4. IBM 3081D Prim (graph set 2). FH, Fibonacci heap implementation; PH, pairing heap implementation.

sequence of cut operations related to the FHeap or unbounded growth of children due to the merge as it relates to the deletemin operation. The upshot of this is that there are very strong grounds for believing that the conjectured decreasekey, insert (and thus merge) operations of $O(1)$ amortized may in fact be $O(1)$ amortized time.

6.4. Prim's Minimum Spanning Tree Algorithm. Because Prim's algorithm is, in essence, a two-way analogue of Dijkstra's algorithm, many of the comments made in the section about Dijkstra's algorithm will apply here as well. Interestingly, the time ratios noted in the previous section also occurred in the Prim tests. Like the heapsort and Dijkstra evidence, these results were also unanimous; Figure 4 gives an idea just how well the PHeap version performed. As in the tests on Dijkstra's algorithm, there may be very strong grounds for believing the conjectured time bounds on PHeap operations. In any event, it is clear that the PHeap-based implementation clearly outperformed the FHeap version.

7. Conclusions. The author has attempted to provide a representative sample of the results obtained in addition to recounting the other results encountered. The figures presented are only a fragment of the observations collected. Evaluating the heapsort and Cheriton–Tarjan minimum spanning tree tests found SHeaps outperforming BHeaps. However, the LHeaps do not fare as badly in the Cheriton–Tarjan minimum spanning tree tests as suggested by Brown, and thus should not be overlooked as a competitive candidate for the minimum spanning tree application. LHeaps and FHeaps are not promising candidates for the sorting applications. While of theoretical interest, FHeaps do not seem to be a good data structure in practice since its implementation is nearly as complicated as BHeaps

and do not seem to perform quite as well. This by no means excludes FHeaps from consideration for use in other applications, but we might want to consider other heap representations first. Surprisingly, the PHeaps performed remarkably well on at least three of the tests (heapsort, Prim, Dijkstra) and seemed to be a competitive candidate for sufficiently sparse graphs on the Cheriton–Tarjan minimum spanning tree tests.

Given the test results, the self-adjusting heuristic for heaps provides a practical alternative to heaps which are complicated by the maintenance of balance/accounting information. The fact that SHeaps outperformed BHeaps under two representative applications partially confirms that SHeaps do better than BHeaps in practice. Further, that a BHeap-based algorithm does not completely dominate an LHeap-based version on two machines gives grounds for questioning the claim made by Brown [1] and Jones [7].

In deciding the most appropriate heap representation for each of the problems, PHeaps seem to be the most promising candidate for heapsort, Prim, and Dijskstra applications, while the Cheriton–Tarjan minimum spanning tree observations indicate that LHeaps and SHeaps seem to be the structures of choice. In taking all of their complex code into account, it is interesting to see that BHeaps and FHeaps did not fare as well on the applications as its less complex cousins. In addition, LHeaps, SHeaps, and PHeaps seem to support lazy/delayed heap operations better than BHeaps and FHeaps (with varying degrees of magnitude). It should be noted, however, that the criteria for selecting efficient heap representation seem to be dependent upon factors relating to the choice of application, hardware, randomness of data, and problem size.

An extension to this work might be a decreasekey operation test for LHeaps, SHeaps, and BHeaps. Such an operation might be based on an arbitrary delete on the node whose key is to be decreased (returning a pointer to that node), decrease the node key in question, and then merge the updated node back into the tree from which it was taken. The operation would be $O(\text{Lg } n)$ in complexity, and the original $O(1)$ insert and decreasekey cost in the Prim and Dijkstra algorithms would rise by a logarithmic factor. Thus while it is asymptotically clear that Prim's and Dijkstra's algorithms using LHeaps, SHeaps, or BHeaps will be slower than using FHeaps and PHeaps, it would be interesting to know the problem size(s) where "break-even points" might occur.

Acknowledgments. I would like to thank Gary Schwartz, Jon Finke, and Brian Eliot of ITS at RPI for their consultations, and Mike Kupferschmid for his invaluable contribution of the necessary timer routines used for part of the study. I thank Robert Tarjan and Daniel Sleator for their encouragement, initial suggestions, and for bringing pairing heaps to my attention, Marcio deCarvalho and Douglas Jones for their comments, and William Yerazunis for his comments and support. I also whish to thank Dale Gulledge for his contributions to a preliminary endeavor to this study and Sara Stopek for her editing assistance. Finally, I wish to extend a special thanks to my advisor Thomas Spencer, without whom none of this would have been possible.

References

[1] Brown, M. R., Implementation and analysis of binomial queue algorithms, *SIAM J. Comput.*, **7** (1978), 298–319.

[2] Cheriton, D., and Tarjan, R. E., Finding minimum spanning trees, *SIAM J. Comput.*, **5** (1976), 724–742.

[3] deCarvahlo, M., Private communication.

[4] Eliot, B., Private communication.

[5] Fredman, M. L., *et al.*, The pairing heap: a new form of self-adjusting heap, *Algorithmica*, **1** (1986), 111–129.

[6] Fredman, M. L., and Tarjan, R. E., Fibonacci heaps and their uses in network optimization algorithms, *Proc. 25th Annual IEEE Symp. on Foundations of Computing*, 1984, pp. 338–345.

[7] Jones, D. W., An empirical comparison of priority queue and event set implementations, *Comm. ACM*, **29** (1986), 300–311.

[8] Jones, D. W., Private communication.

[9] Knuth, D. E., *The Art of Computer Programming*, Vol. 2, 2nd edn., Addison-Wesley, Reading, MA, 1973, pp. 9–24, 39–45.

[10] Knuth, D. E., *The Art of Computer Programming*, Vol. 3, 2nd edn., Addison-Wesley, Reading, MA, 1973, pp. 145–152.

[11] Kupferschmid, M., Private communication.

[12] Sedgewick, R., *Algorithms*, Addison-Wesley, Reading, MA, 1983, pp. 35–42, 398–405.

[13] Sleator, D. D., and Tarjan, R. E., Self-adjusting binary trees, *Proc. 15th ACM Symp. on Theory of Computing*, 1983, pp. 235–246.

[14] Sleator, D. D., and Tarjan, R. E., Self-adjusting binary search trees, *J. Assoc. Comput. Mach.*, **32** (1985), 652–686.

[15] Sleator, D. D., and Tarjan, R. E., Self-adjusting heaps, *SIAM J. Comput.*, **15** (1986), 52–69.

[16] Tarjan, R. E., *Data Structures and Network Algorithms*, CBMS Regional Conference Series in Applied Mathematics, Vol. 44, SIAM, Philadephia, PA, 1983.

[17] Vaucher, J., and Jones, D. W., Technical correspondence on "An empirical comparison of priority queue and event set implementations," *Comm. ACM*, **29** (1986), 1002–1005.

[18] Vuillemin, J., A data structure for manipulating priority queues, *Comm. ACM*, **21** (1978), 309–314.