DIMACS Technical Report 2001-23
August 2001

Generalized Pairing Heaps

by

Amr Elmasry
Computer Science Department, Rutgers University
elmasry@paul.rutgers.edu

DIMACS is a partnership of Rutgers University, Princeton University, AT&T Labs-Research, Bell Labs, NEC Research Institute and Telcordia Technologies (formerly Bellcore).

DIMACS was founded as an NSF Science and Technology Center, and also receives support from the New Jersey Commission on Science and Technology.

ABSTRACT

We give a generalized form for the standard implementation of the pairing heaps. A new parameter k is introduced. When the node with the minimum value is to be deleted from the heap, the operations to combine the resulting sub-trees into one tree depend on the value of k. When the value of k is equal to 2, the implementation will be equivalent to the standard pairing heaps' implementation. We show that, for any constant k, this general form achieves the same bounds as the standard implementation. Finally, experimental results are conducted showing that, by tuning the value of k, the number of comparisons involved in this operation can be reduced.

1 Introduction

The pairing heap [3] is a heap-ordered general tree. The values in the heap are stored one value per tree node. The value stored in the parent of a node is smaller than or equal to 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. The basic operation on a pairing heap is the linking operation, in which two queues are combined, by making the root with the larger key value the leftmost child of the other root. The following operations are defined for the standard implementation of the pairing heaps:

- *makeheap*(h): Create a new empty tree named h.
- *insert*(x, h): Make x into a one-node tree and link it with tree h.
- *findmin*(h): return the value of the root of tree h.
- *decreasekey*(δ, x, h): Subtract δ from the key of item x. If x is not the root of tree h, cut the edge joining x to its parent and link the two trees formed by the cut.
- *deletemin*(h): remove the node at the root of h and return its value. The resulting queues are then combined together using the linking operation to form a single tree. Several variants of this data structure have been suggested, each suggests a different way of combining the resulting queues. In the standard two-pass variant of the pairing heaps, the *deletemin* operation proceeds as follows. In the first pass, the children of the deleted root are paired together starting from left to right, and each pair of trees is linked together. In other words, if the number of the sub-trees is odd, the rightmost one is left unpaired. In the second pass, the remaining sub-trees are linked, in order, from the right to the left. Each sub-tree is linked with the result of the linkings of the sub-trees to its right.
- *delete*(x, h): If x is the root of tree h, perform a *deletemin* on h. Otherwise, cut the edge joining x to its parent, perform a *deletemin* on the tree rooted at x, and link the resulting tree with the tree formed by the cut.

It has been proven [3] that , for an n node heap, the amortized cost of *makeheap* and *findmin* is $O(1)$, and that of the other operations is $O(\log n)$.

Stasko and vitter [6] suggested another variant to achieve an amortized running time of $O(1)$ for the *insert* operation. Their technique relies on having an auxiliary area where the inserted nodes are kept until a *deletemin* operation is to be performed. At that time, the single node trees in the auxiliary area are linked together in pairs, then the resulting trees are linked again in pairs, and the passes are repeated until one tree is left in the heap. This operation is known as the *multi-pass pairing*. The resulting tree is then linked with the root of the original tree in the heap. From this point on, the *deletemin* operation proceeds exactly as mentioned above.

Fredman [2] has shown that a constant amortized cost for the *decreasekey* operation is precluded by establishing a lower bound of $\Omega(\log \log n)$, n is the heap size, for a large family of data structures that generalize the pairing heap.

In this paper we generalize the *deletemin* operation by introducing a new parameter k. We prove that the same bounds can be achieved for any constant value of k. When k is equal to 2, the *deletemin* operation will be the same as that of the standard implementation. We also show that by tuning the value of k relative to the underlying application, we achieve better practical results.

2 The generalized heaps

The only operation that is different from the implementation of the standard pairing heaps is the *deletemin* operation. In the generalized form, the *deletemin* operation will be implemented as follows. First, the root of the heap is removed and its value is reported. Let c be the number of the sub-trees of this deleted root. Then, starting from either the leftmost or the rightmost child of the deleted root, these c sub-trees are grouped into g groups, each has k consecutive sub-trees, except possibly for the last group. Therefore, we have $g = \lceil \frac{c}{k} \rceil$. The first pass starts by combining the sub-trees in each group from the right to the left. In other words, within each group the sub-trees are linked as follows. Starting from the rightmost sub-tree within the group, each sub-tree is linked with the result of the linkings of the sub-trees to its right, in an incremental fashion. In the second pass, the resulting sub-trees are linked, in order, from the right to the left, in a similar fashion to the first pass. Note that by choosing k to be 2 and starting the grouping in the first pass from left to right, the implementation will be the same as the standard pairing heaps.

To achieve a constant amortized cost for the *insert* operation, we can apply the auxiliary heap technique of Stasko and vitter [6].

Theorem 1 *For the generalized pairing heaps, for any constant value of the parameter k, the operations makeheap, findmin and insert run in $O(1)$ amortized time, while the operations decreasekey, deletemin and delete run in $O(\log n)$ amortized time.*

Proof We use the potential technique [7] to prove the theorem. Instead of associating a potential with the nodes of the structure, we associate the potential with the links connecting nodes. Define the size of a sub-tree to be the number of nodes in this sub-tree. When a sub-tree whose size is $s1$ is linked, as the leftmost child, to another sub-tree whose size is $s2$, a potential of $\log(s1+s2)$ [1] is associated with this link.

Note that this potential function is quite similar to the function used in the original analysis of the standard pairing heaps [3]. The exact potential function was also used by Fredman to prove the following result [2]. For the standard pairing heap implementation, beginning with an initially empty heap, and performing a sequence of $m \geq n$ heap operations, which includes at most n *deletemin* operations, such that the heap size does not exceed n at any point. Then, the total execution cost of the sequence does not exceed $O(m \log_{2m/n} n)$.

Except for *deletemin* and *delete* operations, all the bounds in the theorem follows from the fact that any link may increase the potential by at most $O(\log n)$. Next we prove the bound on the *deletemin* operation, which implies the bound for the *delete* operation.

Let s_i be the size of the i^{th} sub-tree, from the right, of the deleted root. When the root is cut by the *deletemin* operation, all the links joining that root to its children are also cut, and the potential decreases by the amount on these links. Since all the links took place by making a sub-tree the leftmost child of the deleted root, the decrease in potential is $\sum_{i=1}^{c} \log(1 + \sum_{j=1}^{i} s_j)$. New links are created during the two linking passes. Assume without loss of generality that c divides k. (The proof is similar if c does not divide k.) The increase in potential as the result of the new links produced by the first pass is $\sum_{i=0}^{g-1} \sum_{j=2}^{k} \log(\sum_{l=1}^{j} s_{ik+l})$. The increase in potential as the result of the new links produced by the second pass is $\sum_{i=2}^{g} \log(\sum_{j=1}^{ik} s_j)$. Let the overall change in potential as a result of the *deletemin* operation be ΔP, then

$$\Delta P = \sum_{i=0}^{g-1} \sum_{j=2}^{k} \log(\sum_{l=1}^{j} s_{ik+l}) + \sum_{i=2}^{g} \log(\sum_{j=1}^{ik} s_j) - \sum_{i=1}^{c} \log(1 + \sum_{j=1}^{i} s_j)$$

Since

$$\begin{aligned} \sum_{i=1}^{c} \log(1 + \sum_{j=1}^{i} s_j) &> \sum_{i=0}^{g-1} \sum_{j=1}^{k} \log(\sum_{l=1}^{ik+j} s_l) \\ &> \sum_{i=0}^{g-1} \sum_{j=2}^{k-1} \log(\sum_{l=1}^{ik+j} s_l) + \sum_{i=1}^{g-1} \log(\sum_{j=1}^{ik+1} s_j) + \end{aligned}$$

[1]All logarithms are to the base 2

$$\sum_{i=1}^{g} \log(\sum_{j=1}^{ik} s_j)$$

$$\sum_{i=0}^{g-1}\sum_{j=2}^{k} \log(\sum_{l=1}^{j} s_{ik+l}) = \sum_{i=0}^{g-1}\sum_{j=2}^{k-1} \log(\sum_{l=1}^{j} s_{ik+l}) + \sum_{i=0}^{g-1} \log(\sum_{j=1}^{k} s_{ik+j})$$

Then

$$\begin{aligned}\Delta P &< \sum_{i=0}^{g-1}\sum_{j=2}^{k-1} \log(\sum_{l=1}^{j} s_{ik+l}) - \sum_{i=0}^{g-1}\sum_{j=2}^{k-1} \log(\sum_{l=1}^{ik+j} s_l) + \\ &\quad \sum_{i=0}^{g-1} \log(\sum_{j=1}^{k} s_{ik+j}) - \sum_{i=1}^{g-1} \log(\sum_{j=1}^{ik+1} s_j)\end{aligned}$$

Since

$$\begin{aligned}\sum_{i=0}^{g-1}\sum_{j=2}^{k-1} \log(\sum_{l=1}^{j} s_{ik+l}) - \sum_{i=0}^{g-1}\sum_{j=2}^{k-1} \log(\sum_{l=1}^{ik+j} s_l) &= \sum_{j=2}^{k-1}\sum_{i=0}^{g-1} (\log(\sum_{l=1}^{j} s_{ik+l}) - \log(\sum_{l=1}^{ik+j} s_l)) \\ \sum_{i=0}^{g-1} \log(\sum_{j=1}^{k} s_{ik+j}) &= \sum_{i=0}^{g-2} \log(\sum_{j=1}^{k} s_{ik+j}) + \log(\sum_{j=1}^{k} s_{c-k+j})\end{aligned}$$

Then

$$\begin{aligned}\Delta P &< \sum_{j=2}^{k-1}\sum_{i=0}^{g-1} (\log(\sum_{l=1}^{j} s_{ik+l}) - \log(\sum_{l=1}^{ik+j} s_l)) + \\ &\quad \sum_{i=0}^{g-2} \log(\sum_{j=1}^{k} s_{ik+j}) - \sum_{i=1}^{g-1} \log(\sum_{j=1}^{ik+1} s_j) + \log n\end{aligned}$$

For every value of j from 2 to $k-1$, let

$$T_j \stackrel{\text{def}}{=} \sum_{i=0}^{g-1} (\log(\sum_{l=1}^{j} s_{ik+l}) - \log(\sum_{l=1}^{ik+j} s_l))$$

And let

$$T_0 \stackrel{\text{def}}{=} \sum_{i=0}^{g-2} \log(\sum_{j=1}^{k} s_{ik+j}) - \sum_{i=1}^{g-1} \log(\sum_{j=1}^{ik+1} s_j)$$

Then

$$\Delta P < \sum_{j=2}^{k-1} T_j + T_0 + \log n$$

Consider any value of j from 2 to $k-1$. For all i from 0 to $g-1$, let

$$p_i \stackrel{\text{def}}{=} \sum_{l=1}^{j} s_{ik+l}$$

For all i from 1 to $g-1$, let

$$\begin{aligned} t_i &\stackrel{\text{def}}{=} \log p_i - \log(\sum_{l=0}^{i} p_l) \\ R_i &\stackrel{\text{def}}{=} \sum_{l=0}^{i} p_i / \sum_{l=0}^{i-1} p_i \end{aligned}$$

Then

$$\begin{aligned} T_j &< \sum_{i=1}^{g-1} t_i \\ t_i &= \log \frac{R_i - 1}{R_i} \\ \prod_{i=1}^{g-1} R_i &\leq n \\ \sum_{i=1}^{g-1} \log R_i &\leq \log n \end{aligned}$$

Using Markov's inequality, there are at least $\frac{g-1}{2}$ values of i satisfying

$$\begin{aligned} R_i &\leq 2^{\frac{2\log n}{g-1}} \\ t_i &\leq \log \frac{2^{\frac{2\log n}{g-1}} - 1}{2^{\frac{2\log n}{g-1}}} \end{aligned}$$

If $g - 1 \leq 2\log n$, then $c \leq k(2\log n + 1)$. And hence the actual number of comparisons is $O(\log n)$. Since $\Delta P = O(\log n)$, then, the amortized time of the *deletemin* operation is $O(\log n)$, and the theorem follows.

We may assume without loss of generality that $g - 1 > 2\log n$, then $\frac{2^{\frac{2\log n}{g-1}}-1}{2^{\frac{2\log n}{g-1}}} < \frac{1}{2}$. Hence, for at least $\frac{g-1}{2}$ values of i, $t_i < -1$. Then

$$T_j < -\frac{g-1}{2}$$

Similarly

$$T_0 < -\frac{g-1}{2}$$

Should we have scaled the potential function to be $\frac{2k}{k-1}\log(s1 + s2)$, then we have

$$\begin{aligned} \Delta P &< -k(g-1) + \frac{2k}{k-1}\log n \\ &< -c + k + \frac{2k}{k-1}\log n \end{aligned}$$

Since the actual number of comparisons is $c-1$, then the amortized time of the *deletemin* operation is $O(\log n)$, and the theorem follows.

3 Experimental findings

Several authors performed experiments on the pairing heaps, either comparing its performance with other forms of priority queues [4, 5], or comparing different variants of this data structure [1, 6].

In our experiments, we follow a similar strategy to [1, 6]. After building a priority queue of a specified size, the remaining operations are subdivided into multiple rounds, where each round consists of a specified number of *decreasekey* operations, a single insertion, and a single *deletemin* operation. A round of operations leaves the priority queue size unchanged. All of our experimental results are reported in terms of average number of comparisons per *deletemin* operation.

The heap operations are implemented as the standard two-pass variant of the pairing heaps. The only exception is the *deletemin* operation, which is implemented in the generalized form. No key values are assigned to the nodes. Every time a comparison is to be performed, the node with more children wins the comparison. A strategy that helps in increasing the number of comparisons accompanying the *deletemin* operations, and causing the performance of the heap to degrade accordingly. The *decreasekey* operations are performed on nodes selected at random.

As mentioned by Jones [4], the initial structure of the priority queue can influence subsequent performance. Thus, our results are measured when the heap reaches a steady state structure, after several rounds of operations are performed.

Our plots reflect runs involving different values for the *decreasekey* operations per round. As expected, the average number of comparisons per *deletemin* operation increases by increasing the number of *decreasekey* operations per round. Several values of the parameter k were tried. See Figures 1,2,3 and 4.

We conclude that a suitable value of k would improve the performance of the pairing heaps dramatically. In particular, when the application under consideration involves a lot of operations that cause the structure to degrade (*decreasekey* and *insert* operations), a larger value of k will be better in order to expedite a better structure.

References

[1] M. Fredman. *A priority queue transform.* 3rd Workshop on Algorithm Engineering in LNCS vol. 1668 (1999), 243-257.

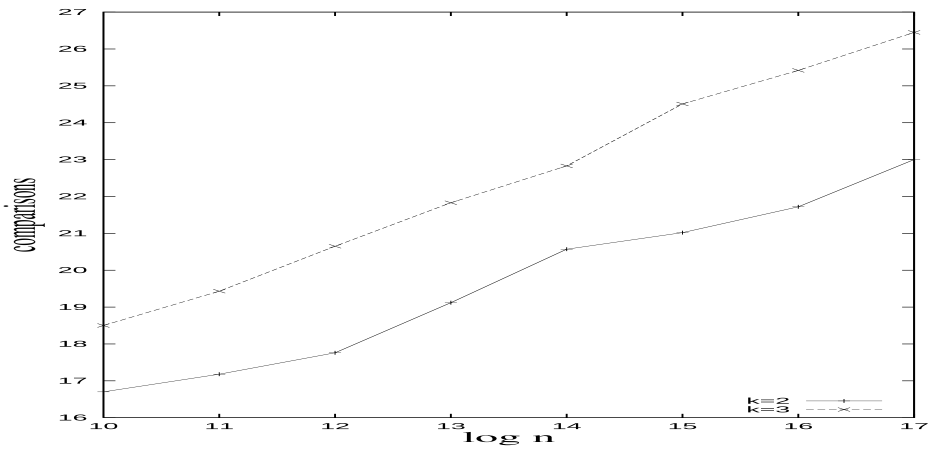

Figure 1: Comparisons per *deletemin* operation. 0 *decreasekey* operations per round.

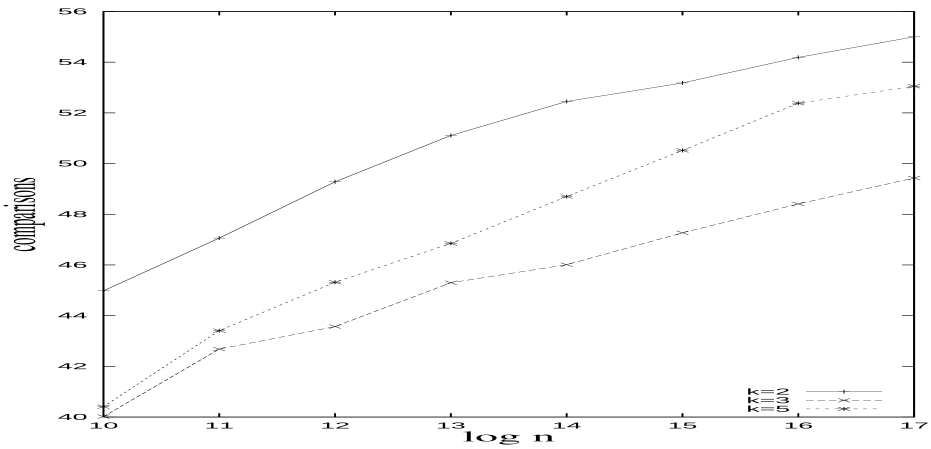

Figure 2: Comparisons per *deletemin* operation. 15 *decreasekey* operations per round.

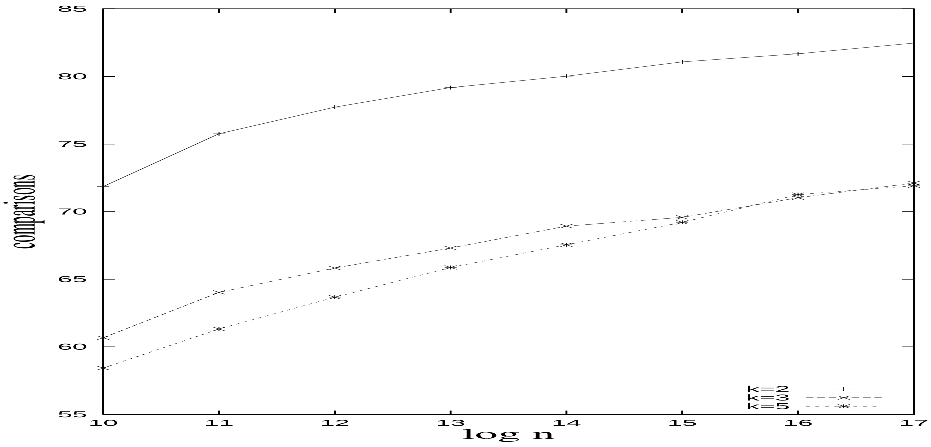

Figure 3: Comparisons per *deletemin* operation. 30 *decreasekey* operations per round.

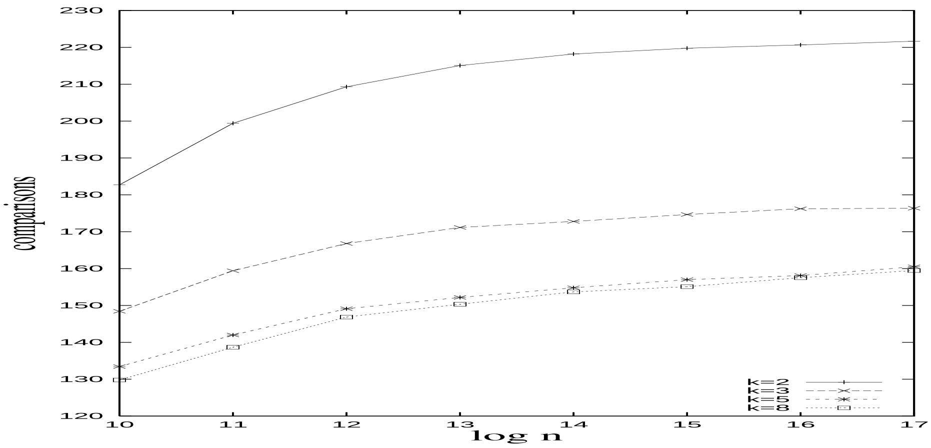

Figure 4: Comparisons per *deletemin* operation. 100 *decreasekey* operations per round.

[2] M. Fredman. *On the efficiency of pairing heaps and related data structures.* Journal of the ACM. 46(4) (1999), 473-501.

[3] M. Fredman, R. Sedgewick, D. Sleator, and R. Tarjan. *The Pairing heap: a new form of self_adjusting heap.* Algorithmica 1,1 (1986), 111-129.

[4] D. Jones. *An empirical comparison of priority-queue and event-set implementations.* Communications Of the ACM 29,4 (1986), 300-311.

[5] B. Moret and H. Shapiro. *An empirical assessment of algorithms for construsting a minimum spanning tree.* DIMACS Monographs In Disc. Math. and theor. Computer Science 15 (1994), 99-117.

[6] J. Stasko and J. Vitter. *Pairing heaps: experiments and analysis.* Communications of the ACM 30,3 (1987), 234-249.

[7] R. Tarjan. *Amortized computational complexity.* SIAM Journal Alg. Disc. Meth. 6 (1985), 306-318.