



The University of Sydney

Are Fibonacci Heaps Optimal?

Technical Report Number 476

January, 1994

Diab Abuaiadh and Jeffrey H Kingston

ISBN 0 86758 902 7

**Basser Department of Computer Science
University of Sydney NSW 2006**

Are Fibonacci Heaps Optimal?

Diab Abuaiadh and Jeffrey H. Kingston

ABSTRACT

In this paper we investigate the inherent complexity of the priority queue abstract data type. We show that, under reasonable assumptions, there exist sequences of n *Insert*, n *Delete*, m *DecreaseKey* and t *FindMin* operations, where $1 \leq t \leq n$, which have $\Omega(n \log t + n + m)$ complexity. Although Fibonacci heaps do not achieve this bound, we present a modified Fibonacci heap which does, and so is optimal under our assumptions.

4 January, 1994

Are Fibonacci Heaps Optimal?

Diab Abuaiadh and Jeffrey H. Kingston

1. Introduction

A *heap* or *priority queue* is an abstract data type consisting of a finite set of items. Each item has a real-valued *key*. The following operations on heaps are allowed:

- *MakeHeap*(h): return a new, empty heap h ;
- *Insert*(x, h): insert a new item x with a given key into heap h ;
- *FindMin*(h): return an item x of minimum key in heap h ;
- *DeleteMin*(h): delete an item of minimum key from h and return it.
- *Delete*(x, h): delete an arbitrary item x from heap h .
- *DecreaseKey*(∇, x, h): decrease the key of item x in heap h by subtracting ∇ .

Delete and *DecreaseKey* assume that the position of item x in h is known.

Fredman and Tarjan [5] invented an implementation of the priority queue called Fibonacci heaps. The time measure is that of amortized time, where the efficiency of each operation is considered over a series of operations rather than the worst case of each operation [9]. As shown in Table 1 column (A), Fibonacci heaps implement all operations in $O(1)$ time except for *DeleteMin* and *Delete*, which are $O(\log n)$. It follows from the $\Omega(n \log n)$ sorting lower bound that, under the decision tree model, at least one of $\{\textit{Insert}, \textit{DeleteMin}\}$ must be $\Omega(\log n)$, and that at least one of $\{\textit{Insert}, \textit{FindMin}, \textit{Delete}\}$ must be $\Omega(\log n)$. Thus no reduction in the cost of either *DeleteMin* or *Delete* is possible without increasing the cost of *Insert* or *FindMin* to $\Omega(\log n)$. For further discussion of heaps, see [1, 2, 4, 7, 8, 10].

One simple way to shift the complexity around is shown in Table 1 column (B). This result is achieved by making each *Insert* deposit $O(\log n)$ which is used later to pay for one *DeleteMin* or *Delete*. However, there is no sequence of operations whose cost is less using (B) than (A). Column (C) presents the only other interesting possibility permitted by the sorting bound. Sequences in which a significant number of *DeleteMin* operations were replaced by *Delete* would cost less under (C) than (A). Such sequences arise in applications [3].

Consider any sequence of n *Insert*, n *Delete*, m *DecreaseKey* and t *FindMin* operations, where $1 \leq t \leq n$. By Table 1, such a sequence has $O(n \log n + m)$ complexity using Fibonacci heaps and $O(t \log n + n + m)$ complexity using implementations that achieve (C).

In Section 2 we show that, under reasonable assumptions, such sequences have $\Omega(n \log t + n + m)$ complexity, ruling out the possibility of implementations that achieve (C). In Section 3 we present a modified Fibonacci heap which achieves this lower bound.

Operation	(A) Fibonacci heaps	(B) Deposit Fibonacci heaps	(C) Conjecture
<i>MakeHeap</i>	$O(1)$	$O(1)$	$O(1)$
<i>Insert</i>	$O(1)$	$O(\log n)$	$O(1)$
<i>FindMin</i>	$O(1)$	$O(1)$	$O(\log n)$
<i>DeleteMin</i>	$O(\log n)$	$O(1)$	$O(\log n)$
<i>Delete</i>	$O(\log n)$	$O(1)$	$O(1)$
<i>DecreaseKey</i>	$O(1)$	$O(1)$	$O(1)$

Table 1. Time complexity of priority queue implementations

2. Lower bound

Suppose we have a heap. Informally, a loser x_1 is an item of h whose key is known to be larger than the key of some other item x_j of h . In other words, comparisons have been performed which show that $x'_1 > x'_2 > \dots > x'_j$, where x'_i , $1 \leq i \leq j$, is the key of the item x_i , all the x_i are present or past members of h , and x_1 and x_j are present members of h . If an item is not a loser then it is a candidate to be an item of minimum key. The status of an item might change from candidate to loser as a result of a comparison and from loser to candidate as a result of a delete operation. Using the adversary method, we will prove the lower bound under the following assumptions:

- (1) The only operations permitted on keys are pairwise comparisons.
- (2) No comparison involves a loser item.

Assumption (1) is standard while assumption (2) will be discussed in Section 4.

Theorem 1: For any implementation I of the heap satisfying assumptions (1) and (2), there exist a sequence s of t *FindMin*, $2n$ *Insert* and n *Delete* operations whose total complexity in I is $\Omega(n \log t + n)$.

Given an implementation I , to prove Theorem 1, we will construct an adversary A who constructs the sequence s . The adversary will maintain a list L of roots of trees. When I invokes *MakeHeap*, A will set L to nil. When I invokes *Insert*(x, h), A will create a tree with a single node containing x and add the tree to L . For *Delete*(x, h), A will locate the node w in L which contains the item x , remove the node w , add all its children to L and destroy w . Consider that there is a comparison in I between the keys of the items x and y . Suppose the items x and y appear in L in the roots of the trees T_x and T_y respectively. If the number of nodes in T_x is greater than or equal to the number in T_y then A will decide that the key of x is strictly less than the key of y and A will link the trees as shown in Figure 1. Otherwise, A will decide that the key of y is strictly less than the key of x and will link T_x to T_y . We will see that assumption (2) ensures that if there is a comparison between the keys of two items, then these items must appear in L in root nodes.

Let s be a sequence of t , $t > 1$, *FindMin*, $2n$ *Insert* and n *Delete* operations, let n/t be an integer, and let $k = n/t$. Let the first n operations be *Insert*, and after this between each two consecutive *FindMin* operations (and after the last *FindMin* operation) let there be k *Delete* and

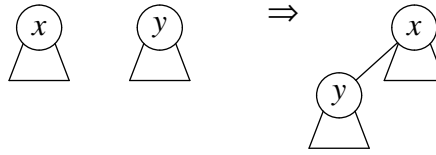


Figure 1. Linking two trees in L after a comparison.

k *Insert* operations. Each *Delete* operation is followed by an *Insert* operation (of a new item). There are no *Delete* operations before the first *FindMin* operation. For each *Delete* operation, let the adversary choose to delete an item in a root of a tree in L that has the maximum number of nodes. We will show that the cost of s is $\Omega(n \log t + n)$.

In this paper, logarithms are to the base two if not specified.

Lemma 1: For each node x in L , the number of its children is at least $\log s(x)$, where $s(x)$ is the number of nodes in the subtree rooted at x .

Proof: By induction on the number of operations on L . It is obvious that the Lemma is correct when L is nil. Let us assume the Lemma is correct after the first j operations. If the $(j + 1)$ th operation is *Insert* or *Delete*, it is obvious the Lemma is correct because the adversary chooses to delete an item which appears in a root node. Let the $(j + 1)$ th operation be a comparison between the keys of two items appearing in nodes v and w such that $s(v) \geq s(w)$ (v is the winner). For each node x except v , the number of the children of x and $s(x)$ are the same as before the comparison. Let $s(v)$ and $s(w)$ be the two subtree sizes before the comparison. After the comparison, the number of the children of v is at least

$$1 + \log s(v) = \log 2s(v) \geq \log(s(v) + s(w))$$

and $s(v) + s(w)$ is the number of nodes in the subtree rooted at v after the comparison. □

Each *Insert* operation adds one tree to L , and each *Delete* (of a root node) removes one tree and adds a number of trees equal to the number of children of the deleted node. After j *Delete-Insert* pairs, the number of trees added to L is equal to the total number of children of the j deleted nodes.

In essence, our argument is that *Delete* operations can be found which cause many trees to be added to L . These trees must be combined into a single tree using comparisons by the time we reach the end of the next *FindMin*. We can say little about the number of trees in L , since this depends on whether I chooses to perform the comparisons early or late; but if i trees are added to L after a *FindMin* operation, then i comparisons must be made by the end of the next *FindMin* operation.

Lemma 2: Immediately after each *FindMin* operation in the sequence s , the first j *Delete* and j *Insert* operations, interspersed by any number of comparisons, add totally to the list L at least $\alpha j \log(n/j) - 1$ trees, where α is a constant equal to the minimum value of $(1 - \log \log x / \log x)$, where $x \geq 2$.

Proof: We prove the Lemma by induction on j . For $j = 1$. Immediately after each *FindMin*

operation there is only one tree in L . By Lemma 1 and since $\alpha \leq 1$ the correctness is immediate. Let the number of trees added during the j *Delete* and j *Insert* operations be $\alpha j \log(n/j) - 1 + r$, where $r \geq 0$. It is obvious that there is a tree with size at least $n/(\alpha j \log(n/j) + r)$. Since the adversary chooses to delete an item which appears in the root of a tree with maximal size, by Lemma 1 the number of trees added is at least $\alpha j \log(n/j) - 1 + r + \log(n/(\alpha j \log(n/j) + r))$. Since the value of the function $f(x) = x + \log(c/(c+x))$, where $c \geq 2$, is non-negative for $x \geq 0$, it follows that:

$$\alpha j \log(n/j) - 1 + r + \log(n/(\alpha j \log(n/j) + r)) \geq \alpha j \log(n/j) - 1 + \log(n/(\alpha j \log(n/j)))$$

Simplifying the above expression :

$$\begin{aligned} & \alpha j \log(n/j) - 1 + \log(n/(\alpha j \log(n/j))) \\ &= \alpha j \log(n/j) - 1 + \log(n/j) - \log \alpha - \log \log(n/j) \\ &\geq \alpha j \log(n/j) - 1 + \alpha \log(n/j) + (1 - \alpha) \log(n/j) - \log \log(n/j) \\ &\geq \alpha(j+1) \log(n/(j+1)) - 1 + (1 - \alpha) \log(n/j) - \log \log(n/j) \end{aligned}$$

Since $1 \leq j \leq k$ and $k \leq n/2$ it follows that $n/j \geq 2$. To complete the proof we need only to show that $(1 - \alpha) \log(n/j) - \log \log(n/j) \geq 0$, and this is equivalent to $\alpha \leq (1 - \log \log x / \log x)$ for $x = n/j$. This follows immediately from the definition of α . \square

Proof of Theorem 1: If $t = 1$ the correctness is immediate. Let $t > 1$. It is obvious that we need $n - 1$ comparisons for the first *FindMin* operation. Let us consider the cost of the i th, $2 \leq i$, *FindMin* operation. The k *Delete* and k *Insert* operations before the i th *FindMin* operation added to L , by Lemma 2, at least $\alpha k \log(n/k) - 1$ new trees. By the discussion preceding Lemma 2, we need $\alpha k \log(n/k) - 1$ comparisons to find an item of minimum key. From this we can see that the total cost of the sequence s is at least

$$n - 1 + \sum_{i=2}^t (\alpha k \log(n/k) - 1) = n - t + \alpha k(t - 1) \log(n/k).$$

Since $k = n/t$ the total cost of s is at least $n - t + \alpha(n - k) \log t$. Investigation of the function $f(x) = 1 - \log \log x / \log x$, $x \geq 2$, shows that $\alpha = 1 - (\log e)/e \approx 0.4693$. From this, it follows that the total cost of s is $\Omega(n \log t + n)$. \square

It follows from Theorem 1 that, under our assumptions, the time complexity of n *Insert*, n *Delete*, m *DecreaseKey* and t *FindMin* operations is $\Omega(n \log t + n + m)$, where $1 \leq t \leq n$.

3. The modified Fibonacci heap

We assume that the reader is familiar with amortized time complexity [9] and the Fibonacci heap data structure as presented in [5]. In our modification of the Fibonacci heap the basic ideas of linking together nodes of equal rank, marking nodes, and cascading cuts are preserved. However we do not maintain a pointer to a minimum node, and the operations are implemented as follows:

- *MakeHeap(h)*: set h to nil;

- *Insert*(x, h): add a new node to the list of roots;
- *FindMin*(h): repeatedly link together pairs of nodes of equal rank until there are no more such pairs, determining a minimum node while scanning the list of roots;
- *DecreaseKey*(∇, x, h): subtract ∇ from the key of x , cascade-cut the node v containing x from its parent, and add the tree rooted at v to the list of roots;
- *Delete*(x, h): cascade-cut the node v containing x from its parent, append the list of the children of v to the list of roots, and destroy v .

We do not have a *DeleteMin* operation; its effect is obtained by *FindMin* followed by *Delete*.

Following Fredman and Tarjan's analysis, the amortized complexity of this implementation is easily shown to be $O(1)$ for *MakeHeap*, $O(1)$ for *Insert*, $O(\log n)$ for *FindMin*, $O(1)$ for *DecreaseKey* and $O(\log n)$ for *Delete*. Assuming $t \leq n$, the total time complexity of a sequence of n *Insert*, n *Delete*, m *DecreaseKey* and t *FindMin* operations is $O(n \log n + t \log n + m) = O(n \log n + m)$. However, we will now show that the sum of the amortized time complexity of the n *Delete* operations is $O(n \log t + n)$. This reduces the total to $O(n \log t + t \log n + n + m) = O(n \log t + n + m)$.

Lemma 3: In a Fibonacci heap containing n nodes, the total number of children of any k nodes is at most $k \log_{\phi}(n/k) + k$, where $\phi = (1 + \sqrt{5})/2$.

Proof: Let v_1, \dots, v_k be any k distinct nodes, which we term *special*. Cascade-cut these special nodes from their parents and add the resulting independent subtrees to the list of roots. Clearly the result is a Fibonacci heap, since it could have arisen from k *DecreaseKey* operations.

Let n_i be the number of descendants of special node v_i in this final heap, including v_i itself. By [5], v_i has at most $\log_{\phi} n_i$ children, so the total number of children of special nodes is at most $\sum_{i=1}^k \log_{\phi} n_i$. Furthermore, $\sum_{i=1}^k n_i \leq n$ since the v_i are now the roots of different subtrees. By the convexity of the log function, $\sum_{i=1}^k \log_{\phi} n_i$ is maximized when $n_i = n/k$ for all i , when its value is $k \log_{\phi}(n/k)$.

However, we desire to know the initial number of children, not the final number, and cascade cuts could well cause these to differ. We insist that if v_i is an ancestor of v_j , then v_i be cut before v_j . This ensures that at the moment each v_j is cut it can have at most one of the other special nodes among its ancestors (at the root, in fact). This ancestor is the only special node affected by the cascade cut, and it loses at most one child by it. We conclude that the special nodes lose at most one child per cascade cut, and hence that the initial total number of children is at most $k \log_{\phi}(n/k) + k$. \square

Lemma 4: If $\sum_{i=1}^k r_i \leq n$ and each $r_i > 0$, then $\sum_{i=1}^k r_i \log_{\phi}(n/r_i) \leq n \log_{\phi} k$.

Proof: Letting $p_i = r_i/n$, we find that

$$\sum_{i=1}^k r_i \log_{\phi}(n/r_i) = n \sum_{i=1}^k p_i \log_{\phi}(1/p_i)$$

where each $p_i > 0$ and $\sum_{i=1}^k p_i \leq 1$. This sum is the entropy of the p_i , maximized when $p_i = 1/k$ for all i as is well known [6]; the result follows immediately. \square

Theorem 2: Let s' be any sequence of n *Delete* operations interspersed with t *FindMin* operations and any number of *DecreaseKey* operations, and suppose the heap contains n nodes when s' begins. Then the total amortized complexity of the n *Delete* operations is $O(n \log t + n)$.

Proof: Suppose we perform r_1 *Delete* operations before the first *FindMin*, r_2 after the first *FindMin* but before the second, and so on until r_{t+1} after the last *FindMin*. According to Lemma 3, the r_i *Delete* operations bring at most $r_i \log_{\phi}(n/r_i) + r_i$ nodes to the list of roots, and this is also the amortized complexity of those operations. *DecreaseKey* operations can only decrease this number. The total amortized complexity of all n *Delete* operations is therefore

$$\sum_{i=1}^{t+1} (r_i \log_{\phi}(n/r_i) + r_i) \leq n \log_{\phi}(t+1) + n$$

by Lemma 4. \square

4. Discussion

For the sequence s we presented for the lower bound, many data structures obey assumption (2) including binomial queues [10], pairing heaps [4] and Fibonacci heaps [5].

It is interesting to know whether the lower bound is still valid without assumption (2). It is natural to think that for each implementation I , we can construct an implementation I' such that I' obeys assumption (2) and the total cost of the comparisons in I' is less than I . The following example rules out this possibility. Suppose we have 5 items and we do 5 *FindMin* operations. Each *FindMin* is followed by a *Delete* operation, where we delete the item of minimum key. By Knuth [7, pp. 184-185] we can sort any 5 items by 7 comparisons and hence we can implement the above sequence by 7 comparisons. On the other hand, for a method which obeys assumption (2) the adversary can force us to do at least 8 comparisons.

However, when the number of *FindMin* operations is significantly less than n , asymptotically, we conjecture that the lower bound is valid without assumption (2).

References

1. Aho, A. V., J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
2. Brown, M. R., Implementation and analysis of binomial queue algorithms. *SIAM J. Comput.* **7**, 298-319 (1978).

3. Abuaiadh, Diab and Jeffrey H. Kingston, *An efficient algorithm for the shortest path problem*. Tech. Rep. TR93-473 (1993), Basser Department of Computer Science F09, The University of Sydney 2006, Australia.
4. Fredman, M. L. , Sedgewick, R., Sleator, D. D., and Tarjan, R. E., The pairing heap: A new form of Self-Adjusting heap. *Algorithmica* **1**, 111-129 (1986).
5. Fredman, M. L. and R. E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM* **34**, 569-615 (1987).
6. Kingston, J. H., *Algorithms and Data Structures: Design, Correctness, Analysis*. Addison-Wesley, 1990.
7. Knuth, D. E., *The Art of Computer Programming Volume 1: sorting and searching*. Addison-Wesley, 1968.
8. Peterson, G. L., *A balanced tree scheme for meldable heaps with updates*. Tech. Rep. GIT-ICS-87-23 (1987), School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA 30332.
9. Tarjan, R. E., Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods* **6**, 306-318 (1985).
10. Vuillemin, J., A data structure for manipulating priority queues. *Comm. ACM* **21**, 309-314 (1978).