Violation Heaps: A Better Substitute for Fibonacci Heaps

Amr Elmasry *

Max-Planck Institut für Informatik
Saarbrücken, Germany
elmasry@mpi-inf.mpg.de

Abstract. We give a priority queue that achieves the same amortized bounds as Fibonacci heaps. Namely, find-min requires $O(1)$ worst-case time, insert, meld and decrease-key require $O(1)$ amortized time, and delete-min requires $O(\log n)$ amortized time. Our structure is simple and promises a more efficient practical behavior compared to any other known Fibonacci-like heap.

1 Introduction

The binomial queues [18] is a basic structure that supports the operations: find-min in $O(1)$ worst-case time, insert and meld in $O(1)$ amortized time, decrease-key and delete-min in $O(\log n)$ worst-case time. It can also be extended to support insert in $O(1)$ worst-case time. Being so natural, simple and efficient, binomial queues does exist in most introductory textbooks for algorithms and data structures; see for example [2].

Realizing that many important network optimization and other algorithms can be efficiently implemented using a heap that better supports decrease-key, and that improving the bound for decrease-key is theoretically possible, Fredman and Tarjan [9] introduced Fibonacci heaps supporting the operations: find-min in $O(1)$ worst-case time, insert, meld and decrease-key in $O(1)$ amortized time, and delete-min in $O(\log n)$ amortized time. Using Fibonacci heaps, the asymptotic time bounds for many algorithms have been improved; see for example [9].

Around the same time, Fredman et al. introduced the pairing heaps [8], a self-adjusting alternative to Fibonacci heaps. They were only able to prove an $O(\log n)$ amortized time bound for all operations. Stasko and Vitter [16] improved the amortized bound for insert to $O(1)$. They also conducted experiments showing that pairing heaps is practically more efficient than Fibonacci heaps and than other known heap structures, even for applications requiring many decrease-key operations! More experiments were also conducted [13] illustrating the practical efficiency of pairing heaps. However, Fredman [7] showed that pairing heaps is not theoretically as efficient as Fibonacci heaps by giving a lower bound of $\Omega(\log \log n)$, and precluded the possibility of achieving an $O(1)$

* Supported by an Alexander von Humboldt Fellowship.

decrease-key unless every node carries an $\Omega(\log\log n)$ information bits. Later, Pettie [15] improved the analysis for the decrease-key operation to achieve an $O(2^{2\sqrt{\log\log n}})$ amortized bound. Recently, Elmasry [4] introduced a variant with an $O(\log\log n)$ amortized bound per decrease-key.

Towards a heap that achieves good worst-case time bounds, Driscoll et al. [3] introduced the relaxed heaps. The rank-relaxed heaps achieves the same amortized bounds as Fibonacci heaps, and were easily extended to the run-relaxed heaps that achieve the bounds in the worst case except for meld. Relaxed heaps are good candidates for applications with possible parallel computations. Still, relaxed heaps are not practically efficient and are more complicated than Fibonacci heaps (they are not actually relaxed). Another priority queue that achieves the same worst-case time bounds as run-relaxed heaps is the fat heaps [12]. Later, Brodal [1] introduced a priority queue that achieves the same bounds as the Fibonacci heaps, including an $O(1)$ meld, but in the worst case. Brodal's structure is impractical and even more complicated than relaxed heaps.

Several attempts were made [10, 11, 14, 17], trying to come up with a priority queue that is theoretically as efficient as Fibonacci heaps without sacrificing practicality. Among these, we find thin heaps [11] the most natural and promising. Being able to improve the space requirements by getting rid of the parent pointers [11, 17], or equivalently by using a binary-tree implementation [10, 14], the practicality issue is not resolved yet (or at least thats our expectation!).

In this paper, we claim that we resolved this issue by introducing a priority queue that has the same amortized bounds as Fibonacci heaps, and that is expected to perform in practice in a more efficient manner than other Fibonacci-like heaps and compete with pairing heaps.

Unlike Fibonacci heaps and the like, that can be implemented on a Pointer Machine [11], violation heaps requires the capabilities of a Random Access Machine in order to support integer arithmetic (integer addition and right shifts).

Our amortized bounds are: $O(1)$ per find-min, insert, meld and decrease-key, and $O(\log n)$ per delete-min. Our bound on the number of children per node is shown to be $2\log n + \delta$ (where δ is the amortized number of children resulting from decrease-key operations), while that for Fibonacci heaps and thin heaps [11] is $\approx 1.44\log n$, and that for 2-3 heaps [17] and thick heaps [11] is $\log n$. Insuring only such a looser bound is not a disadvantage though; it only indicates how much relaxed the structure is. The reason is that a tighter bound would require more effort to restrict the structure to such bound. As an alibi, the amortized bound on the number of children per node for pairing heaps is bounded by $2\log n + \delta$ [8]. In addition, one can resort to known techniques [5, 6] to reduce the number of comparisons performed in a delete-min operation almost by a factor of two.

In the next section we give our motivation: why there is a need for a new structure, what our objectives are, and how to achieve them. Then, we introduce the data structure: design, operations, time bounds, and possible variations. We end the paper with a short conclusion.

2 Motivation

In this section we show why we need a new Fibonacci-like heap structure. We start with the drawbacks of other such heaps, then we summarize our objectives and the features required for a better heap structure, and end the section with some ideas that lead to the violation heaps.

Drawbacks of other structures

The pairing heaps [8] is the most efficient among other Fibonacci-like heaps from the practical point of view [13, 16]. Still, it is theoretically inefficient according to the following fact [7].

- The amortized cost per decrease-key is not a constant.

In contrary to pairing heaps, all known heaps achieving a constant amortized cost per decrease-key [3, 9–11, 14, 17] impose the following constraint, which makes them practically inefficient.

- Every subtree has to permanently maintain some balancing constraint to insure that its size is exponential with respect to its height.

The Fibonacci heaps [9] has the following drawbacks.

- Every node has a parent pointer, for a total of four pointers per node.
- A subtree is cut and its rank decreases if its root loses two of its children. If the subtrees of these two children are small in size compared to the other children, the cascaded cut is immature. The reason is that the size of this cut-subtree is still exponential in its rank. This results in practical inefficiency as we are unnecessarily losing previously-gained information.
- The worst-case cost per decrease-key can be $\Theta(n)$; See exercise 20.4-1 page 496 of [2].

Being able to remedy the drawbacks of Fibonacci heaps, other heap structures have not yet achieved the goal though! The trend of imposing more restricted balancing conditions on every subtree would result in the following pitfalls that accompany a decrease-key operation [10, 11, 14, 17].

- More cuts resulting in the loss of gained information.
- More checks among several cases resulting in a performance slow down.

The rank-relaxed heaps [3] is even more restricted, allowing for no structural violations but for only a logarithmic number of heap-order violations. This requires even more case-based checks accompanying decrease-key operations.

Objectives for a new design

To avoid the above drawbacks, we need a heap with the following properties.

- No parent pointers.
- Structural violations in the subtrees of small children do not matter.
- A relaxed structure (structural fixes postponed as long as possible).
- Avoid cascaded cuts.
- Fewer case-based checks following a decrease-key operation.

Insights

The main intuition behind our design is to allow structural violations resulting from decrease-key operations, and only record the amount of violations in every subtree within the root node of this subtree. The violations are later fixed, by repeatedly cutting the violating largest children of a root, only before linking these roots in a delete-min operation. We rely on the following two ideas:

The first idea, which kills two birds by one stone, is to only consider violations taking place in the first two children of a node (one child is not enough). Similar to [11], we use the unused left pointer of the first child to point to the parent. This makes it easy to convey such violations to a parent node without having a parent pointer.

The second idea is about how to record the violations. As a compensation for the structural violation it has done, a decrease-key operation can pay 1 credit to its parent, 1/2 credit to its grandparent, and in general $1/2^{i-1}$ to its i-th ancestor. Once the credits in a node sum up to at least 1, we declare its subtree as violating. Later, the fix of this violation can be charged to this credit. Fortunately, as long as the credits on a node are still less than 1, the subtree of this node is maintaining a good structure. The question is how to implement this idea; The details follow.

3 The violation heaps

Structure

Similar to Fibonacci heaps, 2-3 heaps and thin heaps, the violation heap is a set of heap-ordered node-disjoint trees. Every node has one real-valued key. We maintain the following data:

a. A singly-linked circular list of tree roots, with a root of minimum key first.
b. A doubly-linked list of the children of each node, with a pointer to its first child.
c. For each first child, the unused pointer points to its parent.
d. For each node: its rank field and a goodness field.

Every node then has: three pointers, and two integers. The *rank field* holds an integer which is an upper bound on the number of children of this node. The *violation* of a node indicates how bad the structure of the subtree of this node is. If the violation of a node is zero, then the size of the subtree of this node is exponential with respect to its rank. For any node z, let r_z be the rank of z, and v_z be the violation of z. The *goodness* of z is defined as $g_z = r_z - v_z$. It is practically more efficient to store and maintain a goodness field instead of a violation field. We use the following equation to calculate g_z from, g_{z1} and g_{z2}, the goodness of its first two children: $g_z = \lfloor (g_{z1} + g_{z2})/2 \rfloor + 2$. Computing the goodness of a node requires an integer addition, a right shift, and two increments. We use $g_z = \lfloor g_{z1}/2 \rfloor + 1$ if z has one child, and $g_z = 0$ if it has no children.

The basic primitive *link* is used for linking two trees of the same rank, by making the root of the larger key value the first child of the other root and incrementing the rank and the goodness of the surviving root. Another primitive is the *unlink*, which is performed by cutting the first child of a root and making its subtree a separate tree. Both linking and unlinking require a constant time. Checking whether a node is one of the first two children can also be done in constant time.

Operations

- *find-min(h)*: Return the first root of h.
- *insert(x,h)*: A new tree with the single node x is inserted into the root list of h. Both the rank field and the goodness field of x are initially set to zero. If the key of x is smaller than the minimum of h, x is inserted in the first position, otherwise in the second position.
- *meld(h_1, h_2)*: The root lists of h_1 and h_2 are combined in a new list whose first root is the smaller between the minimums of the two heaps.
- *decrease-key(d,x,h)*: Start by subtracting d from the key of x. If x is a root, stop after making x the first root of h if its new value is smaller than the minimum of h. If x is a first or second child whose key is still not smaller than its parent, stop. Otherwise, cut the subtree of x and make it a tree in h. Repeatedly traverse the path of ancestors of x, starting from x's position, as long as the visited node is the first or second child of its parent and as long as the recalculated value of the goodness field of the node's parent is smaller than the old value. (Note that the recalculated value of the goodness field of the parent of x may even be larger than the old value; in such case we stop the traversal without further violation updates.)
- *delete-min(h)*: Remove from h the first root and make each of its subtrees a tree in the root list. A *cleaning phase* is then performed, making sure that the first two children of every root have no violation (the goodness of each equals its rank) and that the ranks of these two children have consecutive values. This is done by repeatedly unlinking the first children of each root until the condition is maintained or until the root loses all its children. The cleaning should be done repeatedly for the trees resulting from the unlinking. Every root is then given a rank that is one more than that of its first child, and

its violation is reset to zero. Next, a *linking phase* is performed, linking trees of equal ranks until no two trees of the same rank remain. As for Fibonacci heaps [9], this can be performed in $O(1)$ time per link by using a temporary array indexed by rank to store tree roots. Finally, the root with the minimum key value is moved to the first position in the list.

Analysis

First, we show the validity of our procedures, by proving that the assigned goodness values fulfill the requirements, and that the violations are always non-negative and less or equal to the rank values.

Lemma 1. *For any node z, the following relations are maintained by our operations:*

$$g_z \begin{cases} = 0 & \text{if } z \text{ has no children,} \\ \leq \lfloor g_{z1}/2 \rfloor + 1 & \text{if } z \text{ has one child,} \\ \leq \lfloor (g_{z1} + g_{z2})/2 \rfloor + 2 & \text{otherwise.} \end{cases}$$

Moreover, $0 \leq g_z \leq r_z$.

Proof. When a node is inserted, both its rank and goodness fields are set to 0.

When a subtree is cut by a decrease-key operation, the value of the goodness fields of the ancestors of this node are updated by traversing the affected path upwards. As long as the violation of the nodes on the path is to be increased (goodness should decrease), the decrease-key operation resumes the upward traversal and decreases the goodness field. Once the goodness field of a node is not to be changed, the goodness fields of the other nodes along the path are already valid. For the case when the new goodness value is more than the old value no updates are needed.

The cleaning phase of the delete-min operation ensures that the first two children of each root have zero violation value and consecutive ranks, and that the rank of the root is one more than its first child. Consider any root z after the cleaning phase or after it had gained some children in the linking phase. Let $g_{z2} = g$, then $g_{z1} = g + 1$ and $g_z = g + 2$, and the relations are clearly fulfilled.

Next, we show that $g_z \leq r_z$. By induction, $g_{z1} \leq r_{z1}$ and $g_{z2} \leq r_{z2}$, then $g_z \leq \lfloor (r_{z1} + r_{z2})/2 \rfloor + 2$. Since $r_{z1} > r_{z2}$, then $\lfloor (r_{z1} + r_{z2})/2 \rfloor \leq r_{z1} - 1$ and $g_z \leq r_{z1} + 1$. Since $r_{z1} < r_z$, then $g_z \leq r_z$. □

Now, we prove the *structural* lemma, illustrating that the size of a subtree is exponential with respect to its rank as long as the value of the violation is small.

Lemma 2. *Let s_z be the size of the subtree of any node z, then* $s_z \geq 2^{g_z/2+1} - 1$.

Proof. If z has no children, then $g_z = 0$ and the lemma holds. If z has one child that is a single node, then $g_z = 1$ and $s_z = 2$ implying that the lemma holds as well. If z has one child whose goodness $g_{z1} > 0$, then $g_z \leq \lfloor g_{z1}/2 \rfloor + 1$ implies $g_z \leq g_{z1}$. Using induction, $s_z \geq 2^{g_{z1}/2+1} > 2^{g_z/2+1} - 1$. If z has at least two children, again using induction, then

$$s_z \geq 2^{g_{z1}/2+1} + 2^{g_{z2}/2+1} - 1. \tag{1}$$

Let g be $\min\{g_{z1}, g_{z2}\}$. Once fixing g_z, the right-hand-side of (1) is minimized when $|g_{z1} - g_{z2}| \leq 1$. Using $g_z \leq \lfloor (g_{z1} + g_{z2})/2 \rfloor + 2$ when $|g_{z1} - g_{z2}| \leq 1$, then $g_z \leq g + 2$. Substituting in (1) for $g_{z1} = g_{z2} = g$, then $s_z \geq 2^{g/2+2} - 1 \geq 2^{g_z/2+1} - 1$. The lemma follows. □

Corollary 1. *If $v_z = 0$, then $r_z \leq 2 \log (s_z + 1) - 2$.*

Next, we prove the *key* lemma, which shows that two credits per decrease-key operation are enough to pay for the violations on the nodes of the heap.

Lemma 3. *Consider the heap at any given point of time. Let Δ be the number of decrease-key operations performed so far, and let V be an integer representing the sum of the violation units added to the heap nodes so far. Then, $V \leq 2\Delta$.*

Proof. We think about every node as surviving several epochs. A node starts an epoch having no violations, gains violations until it reaches the maximum for this epoch, then it either leaves the heap or starts a new epoch when its violation is reset to zero in a cleaning phase. Let $\mu_{z,i}$ be the maximum violation of node z within its i-th epoch, then $V = \sum_z \sum_i \mu_{z,i}$. Consider z at the moment when its violation v_z is increased to $\mu_{z,i}$ during the i-th epoch.

First, assume that z has at least two children. Then, $g_z = \lfloor (g_{z1} + g_{z2})/2 \rfloor + 2$ implies that $r_z - v_z = \lfloor (r_{z1} - v_{z1} + r_{z2} - v_{z2})/2 \rfloor + 2$. Since $r_{z2} < r_{z1}$, then $v_z \leq r_z - r_{z2} - 2 + \lfloor (v_{z1} + v_{z2})/2 \rfloor$. Among the children of z whose ranks are $r_z - 1, r_z - 2, \cdots, r_{z2}$ there are two remaining, implying that $r_z - r_{z2} - 2$ has been lost via decrease-key operations. The way the cleaning phase works allows us to dedicate these $\delta_{z,i} = r_z - r_{z2} - 2$ operations to pay for z's violation during this epoch, implying

$$\mu_{z,i} \leq \delta_{z,i} + (v_{z1} + v_{z2})/2. \tag{2}$$

Next, assume that z has one child. Then, $g_z = \lfloor g_{z1}/2 \rfloor + 1$ implies that $r_z - v_z = \lfloor g_{z1}/2 \rfloor + 1$. Since $g_{z1} \geq 0$, then $v_z \leq r_z - 1$. Among the children of z there is only one remaining, implying that $r_z - 1$ has been lost via decrease-key operations. We also dedicate these $\delta_{z,i} = r_z - 1$ operations to pay for z's violation during this epoch. It follows that $\mu_{z,i} \leq \delta_{z,i}$.

Lastly, assume that z has no children. Then, $g_z = 0$ is equivalent to $v_z = r_z$. Since z must have lost all its r_z children via decrease-key operations, these $\delta_{z,i} = r_z$ operations are dedicated to pay for the violation of z during this epoch. It similarly follows that $\mu_{z,i} \leq \delta_{z,i}$.

The value of the violations v_{z1} and v_{z2} in the right-hand-side of (2) are bounded as $v_{z1} \leq \mu_{z1,i'}$ and $v_{z2} \leq \mu_{z2,i''}$, where i' and i'' are not necessarily equal to i. Since a node can be a child of only one node during a single epoch, then $\mu_{z1,i'}$ and $\mu_{z2,i''}$ can be dedicated for bounding $\mu_{z,i}$. Summing up over all nodes and epochs, then $\sum_z \sum_i \mu_{z,i} \leq \sum_z \sum_i \delta_{z,i} + \sum_z \sum_i \mu_{z,i}/2$, or in other words $V \leq \Delta + V/2$. The lemma follows. □

Finally, we prove the main theorem concerning the time bounds for the operations of the violation heaps.

Theorem 1. *The violation heaps requires $O(1)$ amortized time per find-min, insert, meld, and decrease-key; and $O(\log n)$ amortized time per delete-min.*

Proof. Let τ be the number of trees in the heap. We use $\tau + 3\sum_z v_z$ for our potential function. The actual time for find-min, insert, and meld are $O(1)$. Both find-min and meld do not change the potential, and an insertion increases τ by one. It follows that the amortized cost of these operations is $O(1)$. The decrease-key consumes an $O(1)$ time in addition to the time it traverses the tree upwards to increase the violations. The actual time performed by all the decrease-key operations is then proportional to the sum of the violation units added to the heap nodes V, which is amortized $O(1)$ per decrease-key following Lemma 3. The potential function increases by $3V$ as a result of all the decrease-key operations, and every decrease-key increases τ by one. Overall, the amortized cost per decrease key is still $O(1)$. The delete-min operation increases τ by the number of children of the minimum root, whose count excluding those having violations is $O(\log n)$ by Lemma 2. The actual work done in the cleaning phase is compensated for by the drop in the violations performed by every other unlink operation (for every two unlink operations, τ increases by two and $\sum_z v_z$ decreases by at least one) . The actual work done in the linking phase is compensated for by the drop in τ accompanying every link operation. It follows that the amortized cost per delete-min is $O(\log n)$. □

Variations

There are several possibilities and compromises for implementing violation heaps while getting the same asymptotic bounds. Choosing which alternative to use depends on the application:

- A subtree may still be considered valid even if the violation value of its root is a small integer. In such case, the cleaning phase may end up with some of such subtrees being the first two children of some roots. This decreases the work done during the cleaning phase, but makes the structure less restricted and increases the number of children per node.
- Another idea, which we have recently used to improve the amortized cost of the decrease-key operation for Pairing heaps to $O(\log \log n)$ [4], is not to cut the whole subtree of a node whose key is decreased, if the subtree of

its first child has few violations. Instead, we replace such subtree with the subtree of the first child. The idea is that in this case the subtree of the first child is a constant fraction of the cut subtree, and hence the resulting structural violation will be smoother. We also reduce the lost information resulting from the cut by keeping a good portion of the subtree.

- For applications that have fewer decrease-key rate, and machines that are faster when performing increments, comparisons, shifts and adding small integers rather than big integers, it may be better to store and maintain a violation field instead of a goodness field. Accordingly, we use

$$v_z = (r_z - r_{z1} - 1) + \lfloor((r_{z1} - r_{z2} - 1) + v_{z1} + v_{z2})/2\rfloor.$$

  To evaluate $r_{z1} - r_{z2} - 1$, increment r_{z2} until it is equal to r_{z1} counting the increments starting from -1. Similarly, $r_z - r_{z1} - 1$ is evaluated.
- Instead of checking for a node being one of the first two children of another node or not, we can store and maintain a bit per node for this purpose. That way such a check is done faster by decrease-key operations, but these bits have to be maintained during the cleaning and the linking phases and by decrease-key operations when performed on one of the first two children.
- The proofs work equally well if the goodness value of every node that has one child is set to one. Declaring such nodes with more violations would increase the work done by the decrease-key operations when updating the violations of ancestor nodes. On the other hand, the cleaning phase will tend to get rid of such nodes faster, leaning towards a more restricted structure.
- It is theoretically possible to consider more than two children to contribute to the violations in a node. The immediate extension is to consider the first three children. In such case, the following equation can be used to compute the goodness: $g_z = \lfloor(g_{z1} + g_{z2} + g_{z3})/3\rfloor + 2$.

4 Conclusion

We have given a priority queue that performs the same functions as a Fibonacci heap and with the same running-time asymptotic bounds. The main feature of our priority queue is that it allows any possible structure, with no structural restrictions soever, but only record the structure violations in the form of one integer per node, then gets rid of violations only when necessary during the structuring phase. We expect our priority queue to be practically more efficient than the other known Fibonacci-like priority queues. Experimental results still need to be conducted to support our intuitive claims.

References

1. G. Brodal, *Worst-case efficient priority queues*, 7th ACM-SIAM symposium on Discrete Algorithms (1996), 52-58.
2. T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd edition, The MIT Press, Cambridge (2001).

3. J. Driscoll, H. Gabow, R. Shrairman, and R. Tarjan, *Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation*, Communications of the ACM 31 (1988), 1343-1354.
4. A. Elmasry, *Pairing heaps with $O(\log \log n)$ decrease cost*, 20th ACM-SIAM symposium on Discrete Algorithms (2009), 471-476.
5. A. Elmasry, *Layered heaps*, 9th Scandinavian Workshop on Algorithm Theory, Lecture Notes in Computer Science 3111 (2004), 212-222.
6. A. Elmasry, C. Jensen, and J. Katajainen, *Two-tier relaxed heaps*, Acta Informatica 45 (2008), 193-210.
7. M. Fredman, *On the efficiency of pairing heaps and related data structures*, Journal of the ACM 46(4) (1999), 473-501.
8. 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.
9. M. Fredman and R. Tarjan, *Fibonacci heaps and their uses in improved network optimization algorithms*, Journal of the ACM 34 (1987), 596-615.
10. P. Høyer, *A general technique for implementation of efficient priority queues*, 3rd Israel Symposium on the Theory of Computing Systems (1995), 57-66.
11. H. Kaplan and R. Tarjan, *Thin heaps, thick heaps*, ACM Transactions on Algorithms, 4(1) (2008), Article 3.
12. H. Kaplan and R. Tarjan, *New heap data structures*, Technical Report TR-597-99, Princeton University (1999).
13. B. Moret and H. Shapiro, *An empirical assessment of algorithms for constructing a minimum spanning tree*, DIMACS Monographs in Discrete Mathematics and Theoretical Computer Science 15 (1994), 99-117.
14. G. Peterson, *A balanced tree scheme for meldable heaps with updates*, Technical Report GIT-ICS-87-23, School of Information and Computer Science, Georgia Institute of Technology (1987).
15. S. Pettie, *Towards a final analysis of pairing heaps*, 46th IEEE Symposium on Foundations of Computer Science (2005), 174-183.
16. J. Stasko and J. Vitter, *Pairing heaps: experiments and analysis.* Communications of the ACM 30(3) (1987), 234-249.
17. T. Takaoka, *Theory of 2-3 heaps*, Discrete Applied Mathematics, 126(1) (2003), 115-128.
18. J. Vuillemin, *A data structure for manipulating priority queues.* Communications of the ACM 21 (1978), 309-314.