Strict Fibonacci Heaps*

GERTH STØLTING BRODAL†, Aarhus University, Denmark
GEORGE LAGOGIANNIS, Agricultural University of Athens, Greece
ROBERT E. TARJAN‡, Princeton University, USA and Intertrust Technologies, USA

We present the *strict Fibonacci heap*, the first pointer-based heap implementation with time bounds matching those of Fibonacci heaps in the worst case. Strict Fibonacci heaps support make-heap, insert, find-min, meld and decrease-key in worst-case $O(1)$ time, and delete and delete-min in worst-case $O(\lg n)$ time, where n is the size of the heap. The data structure uses linear space.

A previous solution achieving the same time bounds in the RAM model made essential use of arrays and extensive use of redundant counter schemes to maintain balance. Our solution uses neither. Our key simplification is to discard the structure of the smaller heap when doing a meld, and to use the pigeonhole principle in place of the redundant counter mechanism to maintain balance.

CCS Concepts: • **Theory of computation → Data structures design and analysis**.

Additional Key Words and Phrases: Data structures, heaps, meld, decrease-key, worst-case complexity

ACM Reference Format:
Gerth Stølting Brodal, George Lagogiannis, and Robert E. Tarjan. 2024. Strict Fibonacci Heaps. *ACM Trans. Algor.* 1, 1 (December 2024), 18 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Williams in 1964 introduced binary heaps [38]. Since then, the design and analysis of heaps has been thoroughly investigated. The most common operations supported by the heaps in the literature are those listed below. We assume that each item stored in a heap is a (key, value) pair, with the key being selected from a totally ordered universe. In the following we let H interchangeably denote a heap and a reference to the heap.

make-heap() Create a new, empty heap H and return a reference to H.

insert(H, k, v) Insert key k and value v as an item (k, v) into the heap H and return a reference to the item.

*An earlier version of this paper appeared in the Proceedings of the 44th Annual ACM Symposium on Theory of Computing (STOC), 2012 [4].

†Partially supported by Independent Research Fund Denmark, grant 9131-00113B, and Danish National Research Foundation, grant DNRF84, Center for Massive Data Algorithmics (MADALGO).

‡Partially supported by NSF grant CCF-0830676, US-Israel Binational Science Foundation Grant 2006204, and the Distinguished Visitor Program of the Stanford University Computer Science Department. The information contained herein does not necessarily reflect the opinion or policy of the federal government and no official endorsement should be inferred.

Authors' Contact Information: Gerth Stølting Brodal, Aarhus University, Department of Computer Science, Aarhus, Denmark, gerth@cs.au.dk; George Lagogiannis, Agricultural University of Athens, Athens, Greece, lagogian@aua.gr; Robert E. Tarjan, Princeton University, Department of Computer Science, Princeton, NJ, USA and Intertrust Technologies, Sunnyvale, CA, USA, ret@cs.princeton.edu.

ACM 1549-6333/2024/12-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

meld(H_1, H_2) Return a new heap H containing all items in the two heaps H_1 and H_2. The heaps H_1 and H_2 cannot be accessed after meld.

find-min(H) Return a reference to an item with minimum key in the heap H. Heap H must be non-empty.

delete-min(H) Delete from heap H the item whose reference is returned by find-min(H). Heap H must be non-empty.

delete(H, e) Delete an item from heap H given a reference e to the item.

decrease-key(H, e, k) Given a reference e to an item in heap H, replace the key of the item by k. Key k must be smaller than or equal to the previous key of the item.

There are many heap implementations in the literature, with a variety of characteristics, see e.g. the survey by Brodal [3]. We can divide them into two main categories, depending on whether the time bounds are worst-case or amortized. Most of the heaps in the literature are based on *heap-ordered* trees, i.e. tree structures in which each node stores one item and the item stored in a node has a key greater than or equal to the key of the item stored in its parent. Heap-ordered trees give heap implementations that achieve logarithmic time for all the operations. Early examples are the implicit binary heaps of Williams [38], the leftist heaps of Crane [7] as modified by Knuth [28], and the binomial heaps of Vuillemin [37].

The introduction of Fibonacci heaps [20] by Fredman and Tarjan was a breakthrough, since they achieved $O(1)$ amortized time for all the operations above except for delete and delete-min, which require $\Theta(\lg n)$ amortized time, where n is the number of items in the heap and lg the base-two logarithm. The drawback of Fibonacci heaps is that they are complicated compared to existing solutions and not as efficient in practice as others, theoretically less efficient solutions. Thus, Fibonacci heaps opened the way for further progress on the problem of heaps, and many solutions based on the amortized approach have been presented since then, trying to match the time complexities of Fibonacci heaps while being at the same time simpler and more efficient in practice.

Self-adjusting data structures provided a framework in this direction. A self-adjusting data structure does not maintain structural information (such as size or height) within its nodes, but still can adjust itself to perform efficiently. Within this framework, Sleator and Tarjan introduced the skew heap [36], which was an amortized version of the leftist heap. They matched the complexity of Fibonacci heaps on all the operations except for decrease-key, which takes $O(\lg n)$ amortized time. The pairing heap, introduced by Fredman, Sedgewick, Sleator and Tarjan [19], was an amortized version of the binomial heap, and it achieved the same time complexity as Fibonacci heaps except again for decrease-key, the time complexity of which remained unknown for many years. In 1999, Fredman [18] proved that the lower bound for the decrease-key operation on pairing heaps is $\Omega(\lg\lg n)$; thus the amortized performance of pairing heaps does not match the amortized performance of Fibonacci heaps. In 2005, Pettie [32] proved that the time complexity of the decrease-key operation is $2^{O(\sqrt{\lg\lg N})}$, where N is the maximum number of items ever in the heap. Dorfman, Kaplan, Kozma and Zwick [9] gave an improved analysis of the forward variant of pairing heaps (introduced by Fremand et al. [19] as the front-to-back variant), and Dorfman, Kaplan, Kozma, Pettie and Zwick [8] gave an improved analysis for multi-pass pairing heaps. Elmasry [13, 15] gave a variant of pairing heaps that needs $O(\lg\lg n)$ amortized time for decrease-key and meld. Recently, Sinnamon and Tarjan [33, 35] have shown that multipass pairing heaps have the bounds of Fibonacci heaps except for decrease-key, whose amortized time bound is $O((\lg\lg n)(\lg\lg\lg n))$, within a factor of $\lg\lg\lg n$ of Fredman's lower bound. They also showed [34, 35] that slim and smooth heaps [23, 29], two more-recently-introduced self-adjusting heaps, match the bounds of

Fibonacci heaps for all operations except decrease-key and match Fredman's lower bound for decrease-key.

Heaps having amortized performance matching the amortized time complexities of Fibonacci heaps have also been presented. In particular, Driscoll, Gabow, Shrairman and Tarjan [10] proposed rank-relaxed heaps, Kaplan and Tarjan [27] presented thin heaps, Chan [6] introduced quake heaps, Haeupler, Sen and Tarjan introduced rank-pairing heaps [21], Elmasry introduced violation heaps [14] and more recently Hansen, Kaplan and Tarjan [22] presented hollow heaps. Elmasry improved the number of comparisons of Fibonacci heaps by a constant factor [11] and also examined versions of pairing heaps, skew heaps, and skew-pairing heaps [12]. Some researchers, aiming to match the amortized bounds of Fibonacci heaps in a simpler way, followed different directions. Peterson [31] presented a structure based on AVL trees and Høyer [24] presented several structures, including ones based on red-black trees, AVL trees, and (a, b)-trees. Li and Peebles [30] considered a randomized version of Fibonacci heaps.

We now review the progress on this problem based on the worst-case approach. The goal of worst-case efficient heaps is to eliminate the unpredictability of amortized ones, since this unpredictability is not desired in e.g. real time applications. Fibonacci heaps, e.g. require worst-case $\Theta(n)$ time for decrease-key operations.

The targets for the worst-case approach were given by Fibonacci heaps, i.e. the time bounds of Fibonacci heaps should ideally be matched in the worst case. The next improvement after binomial heaps came with the implicit heaps of Carlsson, Munro and Poblete [5] supporting worst-case $O(1)$-time insertions and $O(\lg n)$-time deletions on a single heap stored in an array. Run-relaxed heaps [10] achieve the amortized bounds given by Fibonacci heaps in the worst case, with the exception of the meld operation, which is supported in $O(\lg n)$ time in the worst case. The same result was later also achieved by Kaplan and Tarjan [26] with fat heaps. Fat heaps without meld can be implemented on a pointer machine, but to support meld in $O(\lg n)$ time arrays are required. The meld operation was the next target for achieving constant time in the worst-case framework, in order to match the time complexities of Fibonacci heaps. Brodal [1] achieved $O(1)$ worst-case time for the meld operation on a pointer machine, but not for the decrease-key operation. It then became obvious that although the decrease-key and the meld operation can be achieved in $O(1)$ worst-case time separately, it was open to achieve constant time for both operations in the same data structure. Brodal [2] managed to solve this problem, but his solution requires the use of (extendable) arrays. Kaplan et al. [25] showed that constant-time decrease-key, find-min and meld operations are only possible if the operations get the heap H as an argument. For the pointer machine model of computation, the problem of matching the time bounds of Fibonacci heaps remained open until now, and progress within the worst-case framework has been accomplished only in other directions. In particular, Elmasry, Jensen and Katajainen presented two-tier relaxed heaps [17] in which the number of key comparisons is reduced to $\lg n + 3 \lg \lg n + O(1)$ per delete operation. In [16] they also presented a new idea (which we adapt in this paper) for handling decrease-key operations by introducing structural violations instead of heap order violations.

1.1 Our contribution

In this paper we present the *strict Fibonacci heap*, the first heap implementation that matches the time bounds of Fibonacci heaps in the worst case on a pointer machine, i.e. it uses linear space; and it supports make-heap, insert, find-min, meld and decrease-key in worst-case $O(1)$ time, and delete and delete-min in worst-case $O(\lg n)$ time. This adds the final step after the previous step made by Brodal [2] and answers the long-standing open problem of whether such a pointer-based heap is possible.

Much of the previous work, including [1, 2, 5, 16, 17, 26], used redundant binary counting schemes to keep the structural violations logarithmically bounded during operations. For the heaps described in this paper we use the simpler approach of applying the pigeonhole principle. Our heaps are heap-ordered trees in which the structural violations are subtrees cut off and reattached to the root, as in Fibonacci heaps. The crucial new idea is that when melding two heaps, the data structures maintained for the smaller tree are discarded by marking all these nodes as being *passive*. We mark all nodes in the smaller tree passive in $O(1)$ time using an indirectly accessed shared flag.

We call our data structure the *strict Fibonacci heap* since it achieves the same time bounds as Fibonacci heaps but strictly; that is, in the worst case. Our data structure also uses ideas from the original version of Fibonacci heaps [20] (node ranks, linking by rank).

In Sections 2–4 we describe our data structure, ignoring the pointer-level representation. In Section 2 we state the invariants capturing the properties of our data structure. In Section 3 we describe basic transformations to be used in Section 4 in the implementation of the heap operations. We analyze the operations in Section 5 and give the pointer-level representation in Section 6. In Section 7 we give some concluding remarks and discuss possible variations.

2 Data structure and invariants

In this section we describe our data structure on an abstract level. The representation at the pointer level appears in Section 6.

We represent a heap storing n items by a single rooted tree with n nodes. Each node stores one item and the item stays in the same node until the item is deleted from the heap. The *size* of a tree is the number of nodes it contains. The *degree* of a node is the number of children of the node. Each node has a left-to-right ordered list of its children. We let x.key denote the key of the item stored in node x. We assume that all keys are distinct; if not, we break ties by node identifier (by using node identifier as the tiebreaker the same item can be inserted multiple times in a heap, but all copies are considered distinct). The items satisfy *heap order*, i.e. if y is a child of x then $x.\text{key} < y.\text{key}$. Heap order implies that the item with minimum key is stored in the root. Each node is marked either *active* or *passive*. Each active node is again marked as either *fixed* or *free*. The *rank* of an active node is the number of its fixed children. Each fixed node is assigned a non-negative integer *loss*. The *total loss* of a heap is the sum of the loss over all fixed nodes. See Figure 1 for an example.

The basic idea of our construction is to ensure that (1) all nodes have logarithmic degree by linking active nodes of equal rank, (2) a meld operation makes the root with the larger key a child of the root with the smaller key, and (3) a decrease-key operation on a node cuts the subtree rooted at the node and links it with the root. At a high level, these are the ideas used in Fibonacci heaps. If no meld operations are performed, all nodes will be active. Passive nodes are only created during melds, which make passive all active nodes in the heap of smaller size. (A crucial lower-level implementation detail is that all active nodes in a heap share the same active flag, allowing all active nodes in a heap to be marked passive in $O(1)$ time). Passive nodes will be gradually converted into active nodes of rank zero during heap operations. In the worst case, essentially all nodes can be passive, e.g. when n singleton heaps are melded in a binary tree fashion, then the resulting heap has only $O(\lg n)$ active nodes. When a node becomes passive, it enters a heap at least twice the size, which means intuitively that a passive node is allowed to get one more child when it is made active. In any other case, when a node gets a new child, we use tree transformations to guarantee that the maximum rank and degree are logarithmic. Let us now focus on active nodes and the two versions of active nodes, fixed and free. The loss of a fixed node is increased by one whenever a fixed child of the node is cut, and this child becomes free. The loss of a node corresponds to the marking in Fibonacci heaps, except that Fibonacci heaps only allow a node to have loss at most one. A free node (unlike a passive node) has a rank, and we link free nodes with (only) free nodes of the

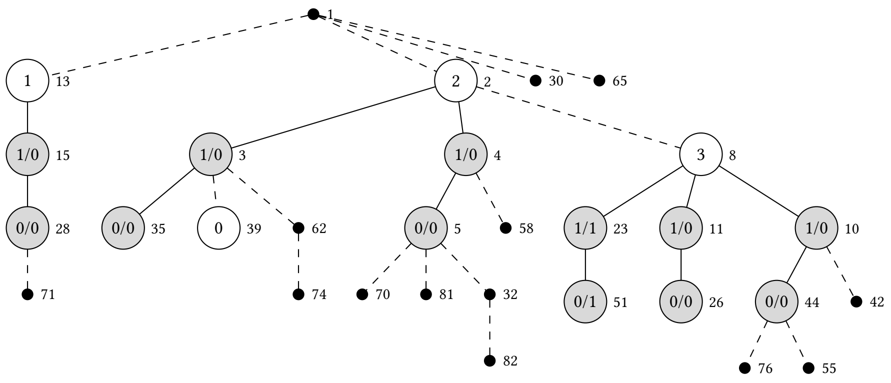


Fig. 1. A heap satisfying invariants I1-I4. White nodes are free nodes, grey nodes are fixed nodes, and black nodes are passive nodes. Solid edges connect a fixed node to its parent, whereas dashed edges connect a free or passive node to its parent. The numbers in the nodes are the ranks of free nodes and ranks/loss of fixed nodes. Note that the rank is equal to the number of solid child edges. The values next to the nodes are the keys of the items stored in the nodes.

same rank, analogous to linking equal rank roots in Fibonacci heaps. The free node that becomes a child of the other node becomes fixed. Such a link reduces the number of free nodes. If no two free nodes of the same rank exist, we cannot reduce the number of free nodes, but their number is bounded by the maximum rank plus one. Since we obtain a logarithmic bound on the maximum rank, it follows that the number of free nodes is also kept logarithmic. We are able to bound the total loss in the same way, i.e. the total loss is kept logarithmic.

Invariants

Our heaps will maintain the invariants below, where R is a non-decreasing function of n that is an upper bound on the maximum rank of any active node. In Section 5 (Lemma 5.7), we prove that $R = \frac{5}{4}\lg n + 6$ is such a bound.[1]

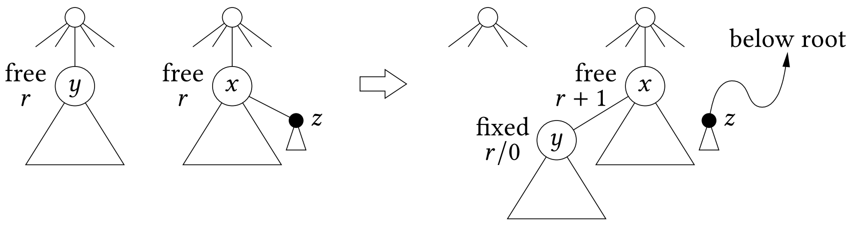
I1 (Structure) All fixed nodes have an active parent. For all nodes the active children are to the left of the passive children. The i-th rightmost fixed child has rank + loss $\geq i - 1$.
I2 (Free nodes) The total number of free nodes is at most $R + 1$.
I3 (Loss) The total loss is at most $R + 1$.
I4 (Degrees) Let $\Delta = \frac{5}{2}\lg(3n - p) + 14$, where p is the number of passive nodes in the heap. Active nodes have degree at most Δ. Passive nodes have degree at most $\Delta - 1$.

Note that I1 does not place any requirement on the parent of a passive or free node. In particular the root can be passive or free. Figure 1 shows an example of a heap satisfying the invariants.

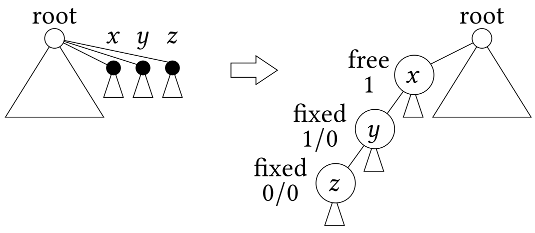
3 Reductions

To maintain the invariants, we need to bound the number of free nodes F, the total loss L, and the degree of the root D in a heap. We will do this by the following transformations that we call *reductions*. Figure 2 illustrates the reductions and the upper part of Table 1 captures their main properties. The only reductions that change L are one-node and two-node loss reductions, both of which reduce L by at least one. A one-node loss reduction increases F by one; a two-node loss

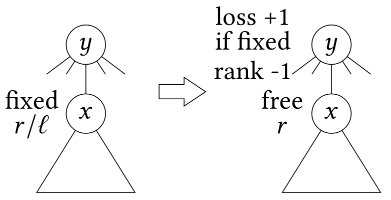
[1]Compared to the conference version of this paper [4], we have changed the invariants to allow the heap operations to be implemented efficiently without ever swapping the items in the nodes.



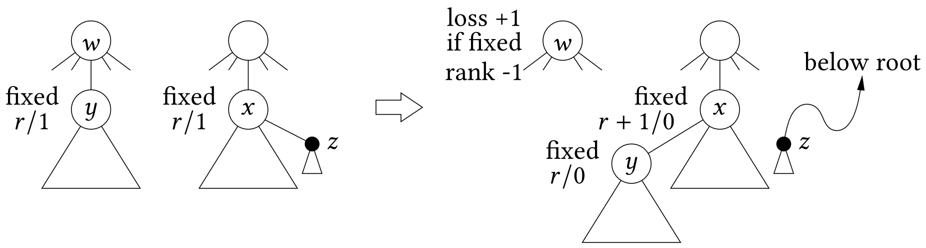
(a) Free node reduction



(b) Root degree reduction



(c) One-node loss reduction (loss $\ell \geq 2$)



(d) Two-node loss reduction

Fig. 2. Transformations to reduce the number of free nodes, the root degree, and the total loss. White nodes are active nodes, small black nodes are passive nodes, and small white nodes can be either active or passive. For a fixed node r/ℓ shows rank/loss, whereas for free nodes only the rank is shown.

reduction increases D by at most one. Free node and root degree reductions each decrease $2D + 3F$ by at least one.

The reductions use the following primitive *cut* and *link* transformations. A *cut* of a child y breaks the link from y to its parent, and makes y a new root. Given two nodes x and y with $x.\text{key} < y.\text{key}$, a *link* of x and y cuts y from its parent (provided y has a parent) and makes y a child of x. If y is active it is made the leftmost child of x; if y is passive it is made the rightmost child of x.

Free node reduction: Let x and y be two free nodes of equal rank r. Compare $x.\text{key}$ and $y.\text{key}$. Assume w.l.o.g. $x.\text{key} < y.\text{key}$. Mark y as fixed with loss zero. Link y to x and increase the rank of x by one. If the rightmost child z of x is passive, link z to the root. This reduction decreases F by one and possibly increases D by one.

Root degree reduction: Let x, y, z be the three rightmost passive children of the root. Using three comparisons, sort x, y, z by key. Assume w.l.o.g. $x.\text{key} < y.\text{key} < z.\text{key}$. Mark x, y and z as active, and mark x as free, and y and z as fixed with loss zero. Assign x and y rank one and z rank zero. Link y to x, and link z to y. Link x to the root. In this reduction x, y and z change from being passive to active, x and y each get one more child, x becomes a new free node (the leftmost child

Table 1. Effect of the different reductions and operations. For insert, meld and decrease-key the sum counts the initial restructuring cost of the operation plus the cost of one-node and two-node loss reductions, free node reductions and root degree reductions, respectively; for delete-min only the initial restructuring cost is stated.

	Root degree (D)	Total loss (L)	Free nodes (F)	Key comparisons
Free node reduction (FR)	$\leq +1$	0	-1	$+1$
Root degree reduction (DR)	-2	0	$+1$	$+3$
Loss reduction (LR)	$\leq +1$	≤ -1	$\leq +1$	$\leq +1$
– one-node	0	≤ -1	$+1$	0
– two-node	$\leq +1$	≤ -1	0	$+1$
insert (3FR+2DR)	$\leq 1+0+3-4$	$\leq 0+0+0+0$	$\leq 1+0-3+2$	$\leq 1+0+3+6$
meld (FR+DR)	$\leq 1+0+1-2$	$\leq 0+0+0+0$	$\leq 0+0-1+1$	$\leq 1+0+1+3$
decrease-key (LR+6FR+4DR)	$\leq 1+1+6-8$	$\leq 1-1+0+0$	$\leq 1+1-6+4$	$\leq 2+1+6+12$
delete-min	$\leq \frac{5}{2}\lg n + 17$	≤ 0	$\leq \frac{5}{4}\lg n + 6$	$\leq \frac{5}{2}\lg n + 17$

of the root), y becomes a new fixed node (the only fixed child of x) with rank one and loss zero, and z becomes a new fixed node (the only fixed child of y), with rank and loss zero. This reduction decreases D by two and increases F by one.

Loss reductions: To reduce the total loss we have two different reductions. The *one-node loss reduction* applies when there is a fixed node x with loss ≥ 2 and is used as a worst-case version of the "cascading cuts" in the original version of Fibonacci heaps. Let y be the parent of x. In this case x is marked a free node and the rank of y is decreased by one. If y is a fixed node, the loss of y increases by one. Since the loss of x decreases by at least two, L decreases by at least one. To reduce L when a one-node loss reduction is not possible, we use a second reduction, *two-node loss reduction*. It applies when two fixed nodes x and y with the same rank r both have a loss of exactly one. Compare x.key and y.key. Assume w.l.o.g. $x.\text{key} < y.\text{key}$. Let w be the parent of y. Link y to x, increase the rank of x by one, and set the loss of x and y to zero. The rank of w is decreased by one. If w is fixed, the loss of w is increased by one. If the rightmost child z of x is passive, link z to the root. Both loss reductions reduce L by at least one, increase F by at most one, and increase D by at most one.

Certain combinations of free node reductions and root degree reductions have only beneficial effects (see Table 1). Specifically, one free node reduction and one root degree reduction combine to decrease D by one without increasing F or L; two free node reductions and one root degree reduction combine to reduce F by one without increasing D or L. When doing such combinations, we do the reductions "to the extent possible": we do them in any order, stopping only when all reductions are done or when no undone reduction can be done.

Whenever one of the above reductions is needed, references to the nodes that will allow for the reduction to be performed are provided by the *fix-list* described in Section 6, if such nodes exist.

The distinct-key assumption and the heap order invariant together ensure that no cycles are created by a free node reduction or a two-node loss reduction.

4 Implementation of the heap operations

The various heap operations are implemented as follows. To find the minimum in a non-empty heap, return the item in the root. To make an empty heap, return an empty tree. To delete an arbitrary item, decrease its key to minus infinity and do a minimum deletion.

To insert a new item, create a new free node of rank zero storing the item, link it with the root, and perform three free node reductions and two root degree reductions to the extent possible.

To meld two heaps with roots x and y, mark all active nodes in the tree of smaller size passive (this is done implicitly, as described in Section 6, so that it takes $O(1)$ time; if both trees have equal size the nodes in one or the other are made passive). If $x.\text{key} > y.\text{key}$ then link x to y, otherwise link y to x. Do one free node reduction and one root degree reduction to the extent possible. This method of melding "forgets" the structure of the tree of smaller size, which eliminates the need to combine complicated data structures during melding. This is the main novelty in our data structure.

To decrease the key of the item in node x in the tree with root z, begin by decreasing the key of the item. If x is the root or $x.\text{key} > y.\text{key}$, where y is the parent of x, we are done. Otherwise, cut x from its parent y. If x is fixed then make x free and decrease the rank of y by one. If x was fixed and y is fixed, increase the loss of y by one. If $x.\text{key} > z.\text{key}$ then link x to z, otherwise link z to x. Finally, do one loss reduction, six free node reductions and four root degree reductions to the extent possible.

To delete the minimum in the tree with root z, first make all fixed children of z free. Then find the node x of minimum key among the children of z (assuming z was not the only item in the heap). Link all other children of z to x. Release z for garbage collection. Mark three arbitrary passive nodes as free nodes with rank zero, and link each to its parent to make it the leftmost child. (Find the passive nodes using the fix-list described in Section 6. This could be considered a *passive node reduction*.) Do loss reductions until none is possible. Then do free node reductions and root degree reductions in any order until none is possible.

The lower part of Table 1 captures the change to D, L, and F when performing the operations insert, decrease-key and meld respectively. Each entry is a sum of four terms stating (1) the change caused by the initial restructuring performed by the operations, and by (2) the loss reductions, (3) free node reductions and (4) root degree reductions. Each entry is an upper bound on the change except for the cases in which no reduction is possible. For meld the stated root degree change is the change to the old root that becomes the new root, whereas the total loss and the number of free nodes are as compared to the heap that is not made passive. For delete-min the bounds are the numbers (not changes) before the repeated free node, loss and root degree reductions, in terms of n, the number of items in the heap before the delete-min. The bounds are immediate consequences of the invariants (to be proved in Section 5).

5 Analysis

Recall that we assume R is a non-decreasing function of n that is an upper bound on the maximum rank of any active node. To show that our algorithm is correct, we will start by proving that the delete-min operation terminates. We will then show that our algorithm maintains invariants I1–I3, and use this to identify valid values for R. Using this bound we can then show that our algorithm maintains I4. Finally we will prove the claimed time complexities.

Lemma 5.1. *Delete-min terminates.*

Proof. Each loss reduction reduces L by at least one. Since L remains non-negative, eventually no loss reduction is possible. Each free node reduction and each root degree reduction reduces $2D + 3F$ by at least one. Since $2D+3F$ remains non-negative, eventually no free node or root degree reduction is possible. (See Table 1 for the effect of the different reductions.) □

Lemma 5.2. *Invariant I1 is satisfied.*

Proof. We need to verify the three different properties. The definition of the link operation and its use whenever a node changes from passive to active guarantees that all the active children of a node are to the left of all the passive ones. To show that fixed nodes have an active parent, we first observe that when a node becomes fixed it is because of a free node reduction or root degree reduction, in which it gets a free parent. Whenever a fixed node changes parent it is because of a two-node loss reduction, in which the new parent is guaranteed to be fixed. Finally, while a node is fixed the status of the parent can change between fixed and free, but it never becomes passive, since the transition from active to passive is applied to all the active nodes of the heap simultaneously, and only during meld. For the third property, that the i-th fixed child of an active node has rank + loss $\geq i-1$, we start by observing that when an active node gets a new fixed child it is the result of a free node reduction, a two-node loss reduction, or a root degree reduction. In the first two cases a new $(r+1)$-st rightmost fixed child is added with r (and loss zero), and in the third case I1 holds by construction for the three new active nodes. Whenever a fixed child y is cut from its active parent x satisfying I1, then I1 is also satisfied for x after the cut. This follows since the i-th rightmost fixed child z after the cut was either the i-th or $(i+1)$-st fixed child before the cut, i.e. after the cut rank + loss is at least $i-1$ for z. Furthermore, if x is fixed, then the rank + loss of x remains unchanged, since the rank of x decreases by one and the loss of x increases by one. It follows that the parent of x will satisfy I1 after the cut if it satisfied I1 before the cut. □

Lemma 5.3. *Invariant I2 is satisfied.*

Proof. When a new empty heap is created there are no free nodes, i.e. I2 is satisfied. We will now show that if I2 is satisfied before an operation, it remains satisfied after the operation. To do this we need to study the changes in F during the operations. These are shown in Table 1. For insert, meld and decrease-key it follows that if all the included reductions are performed, F does not increase, thus preserving I2. Since the reductions are performed to the extent possible, however, one or more of the required free node reductions may not be possible. By the pigeonhole principle, if a free node reduction is not performed then all free nodes must have distinct ranks. Thus, the total number of free nodes is at most $R+1$, so I2 holds. For delete-min, it terminates (by Lemma 5.1). The fact that all reductions are performed to the extent possible means that after the operation, F cannot be reduced anymore, i.e. the lemma holds by the pigeonhole principle. □

Lemma 5.4. *Invariant I3 is satisfied.*

Proof. When a new empty heap is created the total loss is zero, i.e. I3 is satisfied. We will now show that if I3 is satisfied before an operation, it remains satisfied after the operation. This is trivial to conclude for insert and meld because both operations do not affect L. A decrease-key may increase L by one, but it does one loss reduction if possible. If this loss reduction is done, L returns to its value before the operation. Thus, when the operation ends, I3 remains satisfied. If this loss reduction is not performed, then by the pigeonhole principle no fixed node has loss at least 2 and all fixed nodes of loss 1 have distinct rank which means that the total loss is at most $R+1$, i.e. again I3 is satisfied. For delete-min, the fact that all loss reductions are performed to the extent possible, and free node reductions and root degree reductions change no losses, means that after the operation the total loss cannot be reduced anymore, i.e. the lemma holds by the pigeonhole principle and the fact that delete-min terminates (by Lemma 5.1). □

Lemma 5.5 below captures how the maximum rank depends on L given that I1 is satisfied. It makes use of the *binomial trees* B_k, defined recursively as follows: B_0 is a tree of one node, and B_{k+1} is formed from two copies of B_k by making the root of one the leftmost child of the root of the other. Tree B_k contains 2^k nodes; it consists of a root whose children are the roots of copies of $B_{k-1}, B_{k-2}, \ldots, B_0$.

Lemma 5.5. *All active nodes have rank at most* $\lg n + \sqrt{2L} + 2$.

Proof. Assume x is an active node of $r \geq \lg n + k + 1$, where $k = \lceil\sqrt{2L}\rceil$. We shall derive the contradiction that the subtree rooted at x contains at least $n + 1$ nodes. Let T_x be the subtree rooted at x. We cut from T_x all subtrees rooted at passive and free nodes other than x. Now all remaining nodes other than x are fixed. If y is a child of x, z is a child of y, and there is a node with positive loss in the subtree rooted at z, then we cut the subtree rooted at z and increase the loss of y by one (so that I1 remains true for y). This removes the positive loss contributed by the subtree rooted at z, and only increases the loss of y by one, i.e. the total loss is still bounded by L. Now the only descendants of x which can have a positive loss are the children of x. We reduce the + loss of the i-th rightmost child of x to exactly $i - 1$, by lowering the loss and possibly pruning grandchildren of x. Finally, for all nodes $w \neq x$ we repeatedly cut grandchildren of w so that the i-th rightmost child of w has degree exactly $i - 1$. This makes T_w isomorphic to the binomial tree $B_{\text{degree}(w)}$. The minimum size of the resulting tree T_x is achieved by starting with B_r and repeating the following step L times: cut a grandchild of the root with maximum degree and increase the loss of its parent by one. Since the maximum grandchild degree of B_r is $r-2$ and there are exactly j grandchildren of degree $r-j-1$, the number of grandchildren of degree at least $r - k - 1$ is $\sum_{j=1}^{k} j = k(k + 1)/2 \geq k^2/2 = \lceil\sqrt{2L}\rceil^2/2 \geq L$. It follows that no grandchild of x of degree $\leq r - k - 2$ is cut, i.e. the $(r - k)$-th rightmost child w of x has degree $r - k - 1$ and loss zero. Since we assumed $r \geq \lg n + k + 1$, the degree of w is $\geq \lg n$, so T_w has size $2^{\text{degree}(w)} \geq n$. It follows that T_x has size at least $n + 1$, which is a contradiction. This gives $r < \lg n + k + 1 = \lg n + \lceil\sqrt{2L}\rceil + 1 \leq \lg n + \sqrt{2L} + 2$. □

By Lemma 5.4 and Lemma 5.5 we have the following bound on the ranks.

Corollary 5.6. *All nodes have rank at most* $\lg n + \sqrt{2(R + 1)} + 2$.

We are now ready to identify an appropriate value for R. From Corollary 5.6 it follows that it is sufficient that our upper bound on the rank satisfies $R \geq \lg n + \sqrt{2(R + 1)} + 2$.

Lemma 5.7. $R = \frac{5}{4}\lg n + 6$ *is an upper bound on the rank of all active nodes.*

Proof. To find a sufficient condition to satisfy $R \geq \lg n + \sqrt{2(R + 1)} + 2$, we use $\sqrt{x} \leq \frac{1}{10}x + \frac{5}{2}$. The condition then follows from $R \geq \lg n + \frac{1}{10}(2(R + 1)) + \frac{5}{2} + 2 = \lg n + \frac{1}{5}R + \frac{47}{10}$, which is equivalent to $\frac{4}{5}R \geq \lg n + \frac{47}{10}$, i.e. $R \geq \frac{5}{4}\lg n + \frac{47}{8}$. □

All active children of a node are either free or fixed. By the definition of rank a node has at most R fixed children. By the fact that I2 holds, any node has at most $R + 1$ free children. It follows that:

Corollary 5.8. *All nodes have at most* $2R + 1 = \frac{5}{2}\lg n + 13$ *active children.*

Lemma 5.9. *Invariant I4 is satisfied.*

Proof. To prove the validity of I4, we first observe that it trivially holds for an initial empty heap. We next consider the cases in which the invariant can fail. Failure is due to one or more of the following: (1) $\Delta = \frac{5}{2}\lg(3n - p) + 14$ decreases (i.e. n decreases or p increases), (2) a node increases its degree, or (3) a node changes status to passive.

We first argue that we can restrict our attention to the case in which Δ does not decrease. None of the reductions (free node, root degree and loss reductions) introduce new passive nodes, i.e. the reductions do not decrease Δ. Decrease-key can only change the number of passive nodes by root degree reductions, in which case the number of passive nodes decreases and Δ increases. During a delete-min n decreases by one. If we can make three passive nodes active, we are able to counteract the initial decrease to Δ (because $3n - p = 3(n - 1) - (p - 3)$), so Δ does not decrease. If less than three passive nodes exist in the heap before the delete-min operation, then by Corollary 5.8, the inequality $p \leq n$, and the validity of I1–I3 after the operation (as proved above), all nodes have degree at most $\frac{5}{2}\lg n + 13 + 2 \le (\frac{5}{2}\lg(3n-p)+14) - 1 = \Delta - 1$, i.e. I4 is satisfied for all nodes. Finally, for meld let n_1 and n_2 be the sizes of the smaller and the larger tree, respectively, and let p_1 and p_2 be the number of passive nodes in each tree, respectively. For the nodes in the larger tree Δ does not decrease since $3n - p = 3(n_1+n_2) - (p_2+n_1) \ge 3n_2 - p_2$ by $p = p_2 + n_1$. Similarly, for the nodes in the small tree Δ does not decrease since $3n - p = 3(n_1+n_2) - (p_2+n_1) \ge 3n_1 \ge 3n_1 - p_1$ by $n_1 \le n_2$ and $p_2 \le n_2$. We conclude that I4 holds when Δ decreases.

Next, we consider when the degree of a node can increase. The degree of a non-root node may change because of a free node reduction, a root degree reduction or a two-node loss reduction. A free node reduction or a two-node loss reduction can only increase the degree of an active node if this node has no passive children (one of these will be cut and made a child of the root). According to Corollary 5.8, however, nodes with no passive children have degree at most $2R + 1 = \frac{5}{2}\lg n + 13 \le \frac{5}{2}\lg(3n-p) + 13 = \Delta - 1$, so they do not violate I4. During a root degree reduction, the degrees of two non-root nodes increase. In particular, three passive nodes become active and the degrees of two of them increase by one. Again I4 continues to hold for these nodes since by I4 active nodes are allowed to have one more child than passive ones. The degree of the root (i.e. D) may also increase. Any operation that increases D counteracts this increase through root degree reductions, as long as such reductions are possible. For insert, meld and decrease-key, it is obvious that as long as the involved root degree reductions are performed, D is not increased, which means that if I4 was satisfied before the operation, it remains satisfied after the operation. We need to argue that if a root degree reduction is not possible (which is also the case when delete-min terminates), I4 is satisfied at the root. This follows from the fact that if I4 is violated at the root, then by Corollary 5.8 and $p \le n$, the number of passive children of the root is greater than $\Delta - 1 - (2R+1) \ge (\frac{5}{2}\lg(3n-p)+14) - 1 - (\frac{5}{2}\lg n + 13) \ge 5/2$, i.e. the root has at least 3 passive children and a root degree reduction is possible.

What remains is to argue that any node changing status from active to passive continues to satisfy I4 after the operations. Insert, delete-min, and decrease-key do not change the status of any node to passive. During meld all nodes of the smaller heap become passive. Let n_1 and p_1 be the number of nodes and the number of passive nodes in the smaller heap, respectively. Let n and p be the number of nodes and the number of passive nodes in the heap that results from the meld operation, respectively. It is sufficient to show that if the nodes in the smaller tree have degree at most $\Delta_1 = \frac{5}{2}\lg(3n_1 - p_1) + 14$ before the meld, they have degree at most $\Delta - 1 = (\frac{5}{2}\lg(3n-p)+14) - 1$ afterwards. Using $2n_1 \le n$, we have $3n_1 - p_1 \le 3n_1 = \frac{3}{4}(4n_1) \le \frac{3}{4}(2n) \le \frac{3}{4}(3n-p)$, implying

$$\Delta_1 = \frac{5}{2}\lg(3n_1 - p_1) + 14 \le \frac{5}{2}\lg\left(\frac{3}{4}(3n-p)\right) + 14 = \Delta + \frac{5}{2}\lg\frac{3}{4} \le \Delta - 1 .$$

□

Invariant I4 immediately implies that all nodes have degree at most $\frac{5}{2}\lg(3n) + 14 < \frac{5}{2}\lg n + 18$.

THEOREM 5.10. *The time complexities of delete and delete-min are $O(\lg n)$, and of find-min, insert, meld and decrease-key are $O(1)$.*

PROOF. We will prove the time complexities based on the fact that in constant time we can: (1) access the root of a given heap, (2) determine whether a given node is passive, fixed or free, and determine its loss (if it is fixed), (3) make a passive node active, (4) make all (active) nodes in a heap passive and (5) perform any of the reductions of Figure 3, if such a reduction is possible.

Given that the above tasks are performed in constant time (this will be shown in Section 6), it is trivial to show that find-min, insert, meld and decrease-key need O(1) time. Delete-min manipulates the children of the root and performs all reductions to the extent possible. Manipulating the children of the root needs logarithmic time. The time complexity for performing the involved reductions is also logarithmic by the proof of Lemma 5.1, and by the fact that the maximum degree, F, and L are all logarithmically bounded. □

A note on the loss: The loss of a node is a non-negative integer. Even though the loss plays an essential role in invariant I1, only values 0, 1, and "at least 2" are relevant for the algorithm. As soon as the loss is at least two, the loss can only decrease when it is set to zero by a one-node loss reduction. Since the algorithm only tests if the loss is zero, one or at least two, it is sufficient for the algorithm to keep track of these three states. It follows that we can represent the loss using only two bits.

6 Representation details

To represent a heap we use three types of records: *node records*, *heap records* and *rank records*. Below we describe the details of these records. Figures 3 and 4 illustrate them.

Each node is represented by a *node record* storing an item and pointers to its left and right siblings, leftmost child, and parent. An active node also contains a mark indicating whether it is free or fixed and whether its loss is zero, one, or at least two. An insertion returns a reference to the node record created by the insertion. To mark if a node is active or passive we do not store the flag directly in the node but in a *heap record*. Each heap has one heap record marked active and possibly one or more marked passive. When a heap is created it has one heap record marked active. When two heaps are melded, the heap record of the smaller one is marked passive, and the heap record of the larger one becomes the heap record of the melded heap. A reference to a heap is a reference to the active heap record. This record stores a pointer to the node record for the root of the heap (i.e. the root of a given heap is accessible in constant time as needed in Theorem 5.10), as well as the size of the heap. Each node is associated with exactly one heap record (which is reachable from the node record via a rank record, as defined below). Each active node of a heap is associated with the active heap record; each passive node is associated with a passive heap record.

For each heap record we have a *rank-list* storing the set of ranks of all nodes associated with the heap record. The rank of a passive node is the rank it had the last time it changed status from active to passive. A rank-list consists of a doubly linked list of *rank records*, each of which stores a unique integer rank. The rank records are stored in increasing rank order. For a node of rank r, the node record has a pointer to the rank record of rank r. Each rank record stores a pointer to the heap record, and the heap record stores a pointer to the rank record with minimum rank. To determine if a node is active, we follow the rank pointer from the node record to the rank record. From there we follow a pointer to the heap record, where we read the active flag (we have now showed that we can in constant time determine whether a given node is passive, fixed or free and its loss, as needed in Theorem 5.10).

This representation allows all active nodes of the smaller tree to be made passive during a meld in $O(1)$ time by simply changing the active flag in the active heap record (this is also used in Theorem 5.10).

To be able to perform the reductions and operations in Sections 3 and 4, we maintain a *fix-list* containing all node records of a heap in a doubly linked cyclic list. The fix-list is divided into seven parts (possibly empty). Although these parts can be arranged in any order, we have used the numbers 1 to 7 to name them, implying thus a (first to last) order of the parts that eases the description. The fix-list starts at the beginning of Part 1 and ends at the end of Part 7. For a fix-list node, the next (previous) node is its neighbor to the side towards the end (beginning) of the fix-list. Part 1 contains the passive nodes, Parts 2–3 contain the free nodes, and Parts 4–7 contain the fixed nodes. The requirements for the organization of the seven parts are:

Part 1: Passive nodes, in arbitrary order.

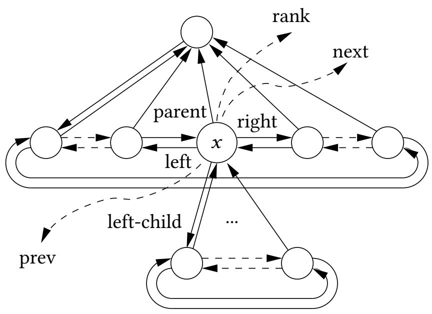


Fig. 3. The fields of a node record x. Fields not shown are key, value, and the free/loss mark.

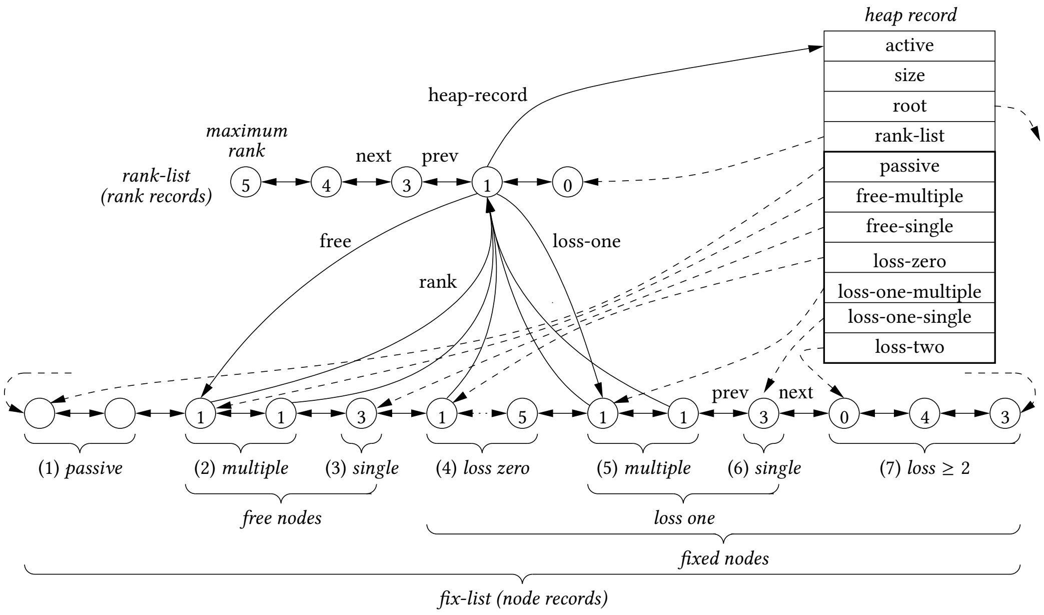


Fig. 4. A heap record, -list and fix-list. The reference-count field in the records of the -list are not shown. Only a subset of the pointers for nodes with one are shown. The numbers in the nodes in the -list are the stored ranks and in the fix-list the s of the nodes.

- **Part 2:** Free nodes of ranks such that there are at least two free nodes of the rank. All free nodes of the same rank are consecutive, but otherwise the nodes can appear in arbitrary order.
- **Part 3:** Free nodes, such that there is only one free node of the rank, in arbitrary order.
- **Part 4:** Fixed nodes with loss zero, in arbitrary order.
- **Part 5:** Fixed nodes with loss one such that there are at least two fixed nodes of the rank. All nodes of the same rank are consecutive, but otherwise the nodes can appear in arbitrary order.
- **Part 6:** Fixed nodes with loss one, such that there is only one node with loss one of the rank, in arbitrary order.
- **Part 7:** Fixed nodes with loss at least two, in arbitrary order.

Parts 2, 5 and 7 contain the nodes on which we can perform free node reductions, two-node loss reductions and one-node loss reductions, respectively. To be able to move nodes between Parts 2 and 3 and Parts 5 and 6, a rank record of rank r stores pointers to the first (i.e. closest to the beginning of the fix-list) free node of rank r and to the first fixed node of rank r and loss one (both of which point back to this rank record). The active heap record stores for each part of the fix-list, a pointer to the first (leftmost) node of the part.

The following is a complete list of the fields of the different records (see also Figures 3 and 4).

Node record.

key, value The key and value stored in the node.

left, right, parent, left-child Pointers to the node records for the left and right sibling of the node (the left and right pointers form a cyclic doubly linked list), the parent node, and the leftmost child. The latter two are NULL if the nodes do not exist.

free/loss A mark only relevant for active nodes, indicating one of the following four states: (1) Free, (2) fixed with loss zero, (3) fixed with loss one, and (4) fixed with loss ≥ 2. We do not store the actual loss if it is at least two.

A pointer to a rank record. If the node is active it is the rank record storing the rank of the node. For a passive node it is the rank record corresponding to the rank of the node when it last changed status from active to passive. The rank pointer of a passive node can only change when the node becomes active again.

prev, next Pointers to the previous and next node records on the fix-list, which is maintained as a cyclic doubly linked list.

Heap record.

active A Boolean flag. A heap record is active or passive if and only if this flag is true or false, respectively. For a heap there is exactly one active heap record. All node records for active nodes point to the rank-list of this heap record. Passive nodes point to rank-lists of passive heap records. There can be zero, one or more passive heap records for a heap.

size The total number of items in the heap if the heap record is active, otherwise zero.

root A pointer to the node record for the root if the heap record is active and has nonzero size, otherwise NULL.

rank-list A pointer to the rank record of the smallest rank in the rank-list of this heap record, NULL if the rank-list is empty. If the rank-list of a passive heap record becomes empty, the heap record is released for garbage collection.

passive, free-multiple, free-single, loss-zero, loss-one-multiple, loss-one-single, loss-two For an active heap record, pointers to the first node of the corresponding seven parts of the fix-list. NULL if the heap record is passive or if the corresponding part is empty.

Rank record (representing r).

rank The integer rank r represented by the rank record.

next, prev Pointers to the next and previous records on the rank-list, in sorted rank order. NULL if no such rank exists.

heap-record Pointer to the heap record (that points to the rank-list containing this rank record).

reference-count The number of node records pointing to this rank record. If a rank record gets a reference-count of zero it is removed from the -list and released for garbage collection. This possibly releases the heap record for garbage collection if it is passive and the rank-list becomes empty.

free A pointer to the first node record on the fix-list for a free node of rank r. NULL if no such node exists.

loss-one A pointer to the first node record on the fix-list for a fixed node with r and loss one. NULL if no such node exists.

6.1 Space usage

We can now bound the space required by our structure. For each item we have one node record, one heap record, and one rank record, the last two of which can be shared by several items. The only other records are active heap records of empty heaps. It follows that the total space for a heap is linear in the number of stored items, plus one.

6.2 Working with the representation

From the above description and pointers (see Figure 4) it follows that we can always perform a free node reduction in constant time as long as Part 2 of the fix-list is non-empty, and we can always perform a loss reduction in constant time unless Parts 5 and 7 are both empty.

In order to perform a free node reduction, we need two free nodes of the same rank. We go to the first node of Part 2. If Part 2 is non-empty (observe that if it is non-empty, it has at least two nodes) we access the first and second node of Part 2. According to the description of the fix-list, they both have the same rank.

In order to perform a loss reduction, we first go to the first node of Part 7 (if Part 7 is non-empty). We perform a one node loss reduction on this node, and we are done. Otherwise, let v and w be the first and second node of Part 5, provided Part 5 is non-empty. According to the structure of Part 5, node w also has loss one and rank equal to the rank of v, i.e. we are able to perform a two-node loss reduction.

We have shown that the fix-list allows performing free node reductions and loss reductions to the extent possible, assuming that we are able to maintain its structure. We now show that the structure of the fix-list can be maintained through the various changes of the status of nodes caused by the reductions and operations. In order to avoid too many technical details, we assume all parts of the fix-list to be non-empty.

Fixed nodes of loss ≥ 2 and of loss zero are easy to handle: To insert a new node of loss ≥ 2, we insert it in the beginning of Part 7. When a fixed node of loss ≥ 2 becomes free we simply remove it from Part 7 and proceed to the rest of the actions, depending on the part of the fix-list in which the node must be re-inserted. Fixed nodes of loss zero in Part 4 are handled similarly.

To insert into the fix-list a new fixed node v of loss one and rank r, we access the loss-one pointer in the rank record corresponding to r. If it is NULL, we insert v in the beginning of Part 6 and update the loss-one pointer in the rank record to point to v. Otherwise, let w be the node referenced by the loss-one pointer in the rank record. If the next node of w has rank r and loss one, we insert v (in Part 5) after w. Otherwise, we remove w from Part 6 and insert w and v as the first and second nodes of Part 5.

To remove a node v of loss one and rank r from the fix-list, we have to take into account that it may lie in Part 5 or in Part 6. If v is referenced by the loss-one pointer in the rank record corresponding to r and the node after v is not of rank r and loss one, we conclude that v lies in Part 6, and we simply remove it and set the loss-one pointer in the rank record to NULL. Otherwise v lies in Part 5 and we proceed as follows: If v is referenced by the loss-one pointer in the rank record corresponding to r, we set the loss-one pointer to point to the node after v. We remove v from the fix-list. Let w be the rank-r node now referenced by the loss-one pointer. If the node after w is not of rank r and loss one (which means that w is now the only node of rank r in Part 5), we remove w from Part 5 and re-insert it in Part 6.

The details of inserting or removing a free node v of rank r from the fix-list are similar to those of inserting or removing a node of loss one and thus omitted.

When the of a node changes, we first remove the node from the fix-list as described above. If no rank node corresponding to the new rank exists in the tree, we have to create a new rank record for the new rank and insert it into the rank-list. This is done in constant time since the new rank record is adjacent to the old rank record (the rank only changes by one). Then we set the rank pointer in the node record to point to the new rank record. The reference-count of the rank node corresponding to the old (new) rank is decreased (increased) by one. If the reference-count of the old rank record is now equal to zero, we delete it from the rank list; release it for garbage collection; and, if needed, update the rank-list pointer in the heap record. Then we re-insert the node record into the fix-list as described above. At this point we have presented all the implementation details needed for performing each of the reductions of Figure 3 in constant time, if such a reduction is possible. (This property is used in Theorem 5.10.)

It remains to describe the actions performed at this representation level when a passive node becomes active. Passive node records point to rank-lists associated with passive heap records. (A heap record may become passive during a meld.) When a passive node becomes active, we follow the rank pointer to access the referenced rank record and from there we follow the heap-record pointer to the passive heap record. We decrease by one the reference-count in the rank record. If the reference-count is now zero, we delete the rank record from the rank-list and free the space occupied by it. (If needed we update the rank-list pointer in the passive heap record so that it continues to point to the rightmost node of the rank-list.) If the rank-list is now empty, we free the space occupied by the (passive) heap record. Garbage collection actions are now over, and the passive node gets a new rank. (It becomes fixed or free.) Following the description above we insert it into the part of the fix-list corresponding to its status. (We have now proved all the properties needed in Theorem 5.10.)

During meld, we make all the nodes of the smaller heap passive by setting the active flag in the corresponding heap record to false. We then link the entire fix-list of the smaller heap into the beginning of the fix-list of the larger heap in constant time (i.e. the fix-list of the smaller heap is inserted into Part 1 of the fix-list of the larger heap). Then we increase the size of the active heap record by the size of the heap record made passive and set the fields of the passive heap record according to their description.

7 Conclusion

We have described the first pointer-based heap implementation achieving the performance of Fibonacci heaps in the worst case. What we presented is just one possible implementation. We have considered many alternative implementations based on the active/passive node idea. In our presentation we have focused on a solution that never needs to move items between nodes, to allow for a clean interface. In the conference version of the paper [4], we took a different approach, which achieves slightly better degree bounds but swaps items between nodes during a decrease-key operation. That version required free nodes to have passive parents and the root to be passive. Another option is to adopt redundant counters instead of the pigeonhole principle to bound the numbers of free nodes and holes created by cutting of subtrees. Further alternatives are to consider ternary linking instead of binary linking, and to use a different "inactivation" criterion during meld, e.g. size plus number of active nodes. Yet another option is to eliminate passive nodes and to use free nodes of rank zero in their place: A meld makes all nodes in the smaller tree free of rank zero, instead of passive. This requires some extra restructuring during decrease-key and delete-min operations, since the cut in a decrease-key, and a one-node loss reduction, can each create a new free node of rank zero. All these variations can be combined, each solution implying different bounds on the maximum degrees, constants in the time bounds, and complexity in the reductions. In the solution presented here, we have aimed at reducing the complexity in the description and having simple transformations, without trying to exactly minimize the constant in the solution.

We have implemented the presented data structure in Python 3 to support the correctness of the presented construction. A comprehensive set of assertions check for the structural integrity of the data structure and a sequence of stress test were performed. The implementation is available at https://github.com/gsbrodal/strict-fibonacci-heaps.

References

[1] Gerth Stølting Brodal. 1995. Fast Meldable Priority Queues. In *Proc. 4th International Workshop Algorithms and Data Structures (Lecture Notes in Computer Science, Vol. 955)*. Springer, 282–290. https://doi.org/10.1007/3-540-60220-8_70

[2] Gerth Stølting Brodal. 1996. Worst-Case Efficient Priority Queues. In *Proc. 7th Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 52–58. http://dl.acm.org/citation.cfm?id=313852.313883

[3] Gerth Stølting Brodal. 2013. A Survey on Priority Queues. In *Space-Efficient Data Structures, Streams, and Algorithms: Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*. Lecture Notes in Computer Science, Vol. 8066. Springer, 150–163. https://doi.org/10.1007/978-3-642-40273-9_11

[4] Gerth Stølting Brodal, George Lagogiannis, and Robert E. Tarjan. 2012. Strict Fibonacci Heaps. In *Proc. 44th Annual ACM Symposium on Theory of Computing*. ACM, 1177–1184. https://doi.org/10.1145/2213977.2214082

[5] Svante Carlsson, J. Ian Munro, and Patricio V. Poblete. 1988. An Implicit Binomial Queue with Constant Insertion Time. In *Proc. 1st Scandinavian Workshop on Algorithm Theory (Lecture Notes in Computer Science, Vol. 318)*. Springer, 1–13. https://doi.org/10.1007/3-540-19487-8_1

[6] Timothy M. Chan. 2013. Quake Heaps: A Simple Alternative to Fibonacci Heaps. In *Space-Efficient Data Structures, Streams, and Algorithms: Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*. Lecture Notes in Computer Science, Vol. 8066. Springer, 27–32. https://doi.org/10.1007/978-3-642-40273-9_3

[7] Clark Allan Crane. 1972. *Linear lists and priority queues as balanced binary trees*. Ph. D. Dissertation. Stanford University, Stanford, CA, USA.

[8] Dani Dorfman, Haim Kaplan, László Kozma, Seth Pettie, and Uri Zwick. 2018. Improved Bounds for Multipass Pairing Heaps and Path-Balanced Binary Search Trees. In *26th Annual European Symposium on Algorithms, ESA 2018 (LIPIcs, Vol. 112)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 24:1–24:13. https://doi.org/10.4230/LIPIcs.ESA.2018.24

[9] Dani Dorfman, Haim Kaplan, László Kozma, and Uri Zwick. 2018. Pairing heaps: the forward variant. In *Proc. 43rd International Symposium on Mathematical Foundations of Computer Science (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 117)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 13:1–13:14. https://doi.org/10.4230/LIPIcs.MFCS.2018.13

[10] James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert E. Tarjan. 1988. Relaxed Heaps: An Alternative to Fibonacci Heaps with Applications to Parallel Computation. *Commun. ACM* 31, 11 (1988), 1343–1354. https://doi.org/10.1145/50087.50096

[11] Amr Elmasry. 2004. Layered Heaps. In *Proc. 9th Scandinavian Workshop on Algorithm Theory (Lecture Notes in Computer Science, Vol. 3111)*. Springer, 212–222. https://doi.org/10.1007/978-3-540-27810-8_19

[12] Amr Elmasry. 2004. Parameterized self-adjusting heaps. *Journal of Algorithms* 52, 2 (2004), 103–119. https://doi.org/10.1016/j.jalgor.2004.03.002

[13] Amr Elmasry. 2009. Pairing heaps with $O(\log\log n)$ decrease cost. In *Proc. 20th Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 471–476. http://doi.acm.org/10.1145/1496770.1496822

[14] Amr Elmasry. 2010. The Violation Heap: a Relaxed Fibonacci-like Heap. *Discrete Mathematics, Algorithms and Applications* 2, 4 (2010), 493–504. https://doi.org/10.1142/S1793830910000838

[15] Amr Elmasry. 2017. Toward Optimal Self-Adjusting Heaps. *ACM Transactions on Algorithms* 13, 4, Article 55 (2017), 14 pages. https://doi.org/10.1145/3147138

[16] Amr Elmasry, Claus Jensen, and Jyrki Katajainen. 2007. On the Power of Structural Violations in Priority Queues. In *Proc. 13th Computing: The Australasian Theory Symposium. (CRPIT, Vol. 65)*. Australian Computer Society, 45–53. http://dl.acm.org/citation.cfm?id=1273694.1273700

[17] Amr Elmasry, Claus Jensen, and Jyrki Katajainen. 2008. Two-Tier Relaxed Heaps. *Acta Informatica* 45, 3 (2008), 193–210. https://doi.org/10.1007/s00236-008-0070-7

[18] Michael L. Fredman. 1999. On the Efficiency of Pairing Heaps and Related Data Structures. *J. ACM* 46, 4 (1999), 473–501. https://doi.org/10.1145/320211.320214

[19] Michael L. Fredman, Robert Sedgewick, Daniel Dominic Sleator, and Robert Endre Tarjan. 1986. The Pairing Heap: A New Form of Self-Adjusting Heap. *Algorithmica* 1, 1 (1986), 111–129. https://doi.org/10.1007/BF01840439

[20] Michael L. Fredman and Robert Endre Tarjan. 1987. Fibonacci heaps and their uses in improved network optimizatio algorithms. *J. ACM* 34, 3 (1987), 596–615. https://doi.org/10.1145/28869.28874

[21] Bernhard Haeupler, Siddhartha Sen, and Robert Endre Tarjan. 2011. Rank-Pairing Heaps. *SIAM Journal of Computing* 40, 6 (2011), 1463–1485. https://doi.org/10.1137/100785351

[22] Thomas Dueholm Hansen, Haim Kaplan, Robert E. Tarjan, and Uri Zwick. 2017. Hollow Heaps. *ACM Transactions on Algorithms* 13, 3 (2017), 42:1–42:27. https://doi.org/10.1145/3093240

[23] Maria Hartmann, László Kozma, Corwin Sinnamon, and Robert E. Tarjan. 2021. Analysis of Smooth Heaps and Slim Heaps. In *48th International Colloquium on Automata, Languages, and Programming (ICALP 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 198).* Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 79:1–79:20. https://doi.org/10.4230/LIPIcs.ICALP.2021.79

[24] Peter Høyer. 1995. A General Technique for Implementation of Efficient Priority Queues. In *Proc. 3rd Israel Symposium on the Theory of Computing and Systems.* IEEE Computer Society, 57–66. https://doi.org/10.1109/ISTCS.1995.377045

[25] Haim Kaplan, Nira Shafrir, and Robert E. Tarjan. 2002. Meldable Heaps and Boolean Union-find. In *Proc. 34th Annual ACM Symposium on Theory of Computing* (Montreal, Quebec, Canada) *(STOC '02).* ACM, 573–582. https://doi.org/10.1145/509907.509990

[26] Haim Kaplan and Robert E. Tarjan. 1999. *New heap data structures.* Technical Report TR-597-99. Department of Computer Science, Princeton University. https://www.cs.princeton.edu/research/techreps/TR-597-99

[27] Haim Kaplan and Robert Endre Tarjan. 2008. Thin heaps, thick heaps. *ACM Transactions on Algorithms* 4, 1 (2008), 3:1–3:14. https://doi.org/10.1145/1328911.1328914

[28] Donald E. Knuth. 1973. *The Art of Computer Programming, Volume I: Fundamental Algorithms, 2nd Edition.* Addison-Wesley.

[29] László Kozma and Thatchaphol Saranurak. 2020. Smooth Heaps and a Dual View of Self-Adjusting Data Structures. *SIAM Journal of Computing* 49, 5 (2020), STOC18–45–STOC18–93. https://doi.org/10.1137/18M1195188

[30] Jerry Li and John Peebles. 2015. Replacing Mark Bits with Randomness in Fibonacci Heaps. In *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9134).* Springer, 886–897. https://doi.org/10.1007/978-3-662-47672-7_72

[31] Gary L. Peterson. 1987. *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.

[32] Seth Pettie. 2005. Towards a Final Analysis of Pairing Heaps. In *Proc. 46th Annual IEEE Symposium on Foundations of Computer Science.* IEEE Computer Society, 174–183. https://doi.org/10.1109/SFCS.2005.75

[33] Corwin Sinnamon and Robert E. Tarjan. 2023. A Nearly-Tight Analysis of Multipass Pairing Heaps. In *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms.* SIAM, 535–548. https://doi.org/10.1137/1.9781611977554.CH23

[34] Corwin Sinnamon and Robert E. Tarjan. 2023. A Tight Analysis of Slim Heaps and Smooth Heaps. In *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms.* SIAM, 549–567. https://doi.org/10.1137/1.9781611977554.CH24

[35] Corwin Sinnamon and Robert E. Tarjan. 2025. Efficiency of Self-Adjusting Heaps. *ACM Transactions on Algorithms* (2025).

[36] Daniel Dominic Sleator and Robert Endre Tarjan. 1986. Self-Adjusting Heaps. *SIAM Journal of Computing* 15, 1 (1986), 52–69. https://doi.org/10.1137/0215004

[37] Jean Vuillemin. 1978. A Data Structure for Manipulating Priority Queues. *Commun. ACM* 21, 4 (1978), 309–315. https://doi.org/10.1145/359460.359478

[38] John William Joseph Williams. 1964. Algorithm 232: Heapsort. *Commun. ACM* 7, 6 (1964), 347–348. https://doi.org/10.1145/512274.512284