

Strict Fibonacci Heaps

Gerth Stølting Brodal
MADALGO*
Dept. of Computer Science
Aarhus University
Åbogade 34, 8200 Aarhus N
Denmark
gerth@cs.au.dk

George Lagogiannis
Agricultural University
of Athens
Iera Odos 75, 11855 Athens
Greece
lagogian@aua.gr

Robert E. Tarjan[†]
Dept. of Computer Science
Princeton University
and HP Labs
35 Olden Street, Princeton
New Jersey 08540, USA
ret@cs.princeton.edu

ABSTRACT

We present the first pointer-based heap implementation with time bounds matching those of Fibonacci heaps in the worst case. We 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, very complicated, 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. We use the pigeonhole principle in place of the redundant counter mechanism.

Categories and Subject Descriptors

E.1 [Data Structures]: Trees; F.2.2 [Theory of Computation]: Analysis of Algorithms and Problem Complexity—*Nonnumerical Algorithms and Problems*

General Terms

Algorithms

Keywords

Data structures, heaps, meld, decrease-key, worst-case complexity

*Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation.

[†]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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

STOC.12, May 19–22, 2012, New York, New York, USA.
Copyright 2012 ACM 978-1-4503-1245-5/12/05 ...\$10.00.

1. INTRODUCTION

Williams in 1964 introduced binary heaps [25]. 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 contains an associated key. No item can be in more than one heap at a time.

makeheap() Create a new, empty heap and return a reference to it.

insert(H, i) Insert item i , not currently in a heap, into heap H , and return a reference to where i is stored in H .

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

find-min(H) Return a reference to where the item with minimum key is stored in the heap H .

delete-min(H) Delete the item with minimum key from the heap H .

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

decrease-key(H, e, k) Decrease the key of the item given by the reference e in heap H to the new key k .

There are many heap implementations in the literature, with a variety of characteristics. 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 where the item stored in a node has a key not smaller than 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 [25], the leftist heaps of Crane [5] as modified by Knuth [20], and the binomial heaps of Vuillemin [24].

The introduction of Fibonacci heaps [15] 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 $O(\lg n)$ 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 other, 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 towards this direction. A self-adjusting data structure is a structure that does not maintain structural information (like size or height) within its nodes, but still can adjust itself to perform efficiently. Within this framework, Sleator and Tarjan introduced the skew heap [23], 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, Sedgwick, Sleator and Tarjan [14], was the amortized version of the binomial heap, and it achieved the same time complexity as Fibonacci heaps except again for the decrease-key, the time complexity of which remained unknown for many years. In 1999, Fredman [13] 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 [22] proved that the time complexity of the decrease-key operation is $2^{O(\sqrt{\lg \lg n})}$. Later, Elmasry [9] gave a variant of pairing heaps that needs only $O(\lg \lg n)$ amortized time 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 [6] proposed rank-relaxed heaps, Kaplan and Tarjan [19] presented thin heaps, Chan [4] introduced quake heaps, Haeupler, Sen and Tarjan introduced rank-pairing heaps [16], and Elmasry introduced violation heaps [10]. Elmasry improved the number of comparisons of Fibonacci heaps by a constant factor [7] and also examined versions of pairing heaps, skew heaps, and skew-pairing heaps [8]. Some researchers, aiming to match the amortized bounds of Fibonacci heaps in a simpler way, followed different directions. Peterson [21] presented a structure based on AVL trees and Høyer [17] presented several structures, including ones based on red-black trees, AVL trees, and (a, b) -trees.

Let us 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.

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 the implicit heaps of Carlsson, Munro and Poblete [3] supporting worst-case $O(1)$ time insertions and $O(\lg n)$ time deletions on a single heap stored in an array. Run-relaxed heaps [6] 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 [18] 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. Bro-

dal [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 is very difficult to achieve constant time for both operations in the same data structure. Brodal [2] managed to solve this problem, but his solution is very complicated and requires the use of (extendable) arrays. 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 [12] in which the number of key comparisons is reduced to $\lg n + 3 \lg \lg n + O(1)$ per delete operation. They also presented [11] 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 first heap implementation that matches the time bounds of Fibonacci heaps in the worst case on a pointer machine, i.e. we achieve a linear space data structure supporting 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 heap is possible.

Much of the previous work, including [1, 2, 3, 11, 12, 18], 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 essentially heap ordered trees, where the structural violations are subtrees being cut off (and attached 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 (active) nodes in the smaller tree passive in $O(1)$ time using an indirectly accessed shared flag.

In Sections 2-4 we describe our data structure, ignoring the pointer-level representation, and analyze it in Section 5. The pointer-level representation is given 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 is given in Section 6.

A heap storing n items is represented by a single ordered tree with n nodes. Each node stores one item. 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. We assume that all keys are distinct; if not, we break ties by item identifier. We let $x.key$ denote the key of the item stored in node x . The items satisfy *heap order*, i.e. if x is a child of y then $x.key > y.key$. Heap order implies that the item with minimum key is stored in the root.

The basic idea of our construction is to ensure that all nodes have logarithmic degree, that a meld operation makes the root with the larger key a child of the root with the

smaller key, and that a decrease-key operation on a node detaches the subtree rooted at the node and reattaches it as a subtree of the root. To guide the necessary restructuring, we need the following concepts and invariants.

Each node is marked either *active* or *passive*. An active node with a passive parent is called an *active root*. The *rank* of an active node is the number of active children. Each active node is assigned a non-negative integer *loss*. The *total loss* of a heap is the sum of the loss over all active nodes. A passive node is *linkable* if all its children are passive.

In order to keep the node degrees logarithmic during deletions, we maintain all nodes of a heap except for the root in a queue Q . A non-root node has *position* p if it is the p -th node on the queue Q .

Invariants

Let $R = 2 \lg n + 6$. Note that R is not necessarily an integer. We later show that R is a bound on the rank of active nodes (Corollary 1). The value of R is only needed for the analysis — it is not maintained by the algorithms.

I1 (Structure) For all nodes the active children are to the left of the passive children. The root is passive and the linkable passive children of the root are the rightmost children. For an active node, the i -th rightmost active child has rank+loss at least $i - 1$. An active root has loss zero.

I2 (Active roots) The total number of active roots is at most $R + 1$.

I3 (Loss) The total loss is at most $R + 1$.

I4 (Degrees) The maximum degree of the root is $R + 3$. Let x be a non-root node, and let p denote its position in Q . If x is a passive node or an active node with positive loss its degree is at most $2 \lg(2n - p) + 9$; otherwise, x is an active node with loss zero and is allowed to have degree one higher, i.e. degree at most $2 \lg(2n - p) + 10$.

Note that I4 implies that all nodes have degree at most $2 \lg n + 12$. The above invariants imply a bound on the maximum rank an active node can have. The following lemma captures how the maximum rank depends on the value of R , for arbitrary values of R .

LEMMA 1. *If I1 is satisfied and the total loss is L , then the maximum rank is at most $\lg n + \sqrt{2L} + 2$.*

PROOF. Assume x is an active node of maximum rank $r \geq k + 1 + \lg n$, where k is the minimum integer such that $k(k + 1)/2 \geq L$. We will prove the contradiction that the subtree rooted at x contains at least $n + 1$ nodes. Let T_x be the subtree rooted at x . We prune from T_x all subtrees rooted at passive nodes. 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 prune the subtree rooted at z and increase the loss of y by one (so that I1 remains satisfied). 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 only the children of x can have a positive loss. We reduce the rank+loss of the i -th rightmost child of x to $i - 1$, by lowering the loss and possibly pruning

grandchildren. Finally, for all nodes $v \neq x$ we repeatedly prune grandchildren such that the i -th rightmost child of v has degree exactly $i - 1$. The remaining subtrees T_v are binomial trees of size $2^{\text{degree}(v)}$. The minimum size of such a T_x is achieved by starting with a binomial tree of size 2^r , and repeating the following step L times: prune a grandchild of the root with maximum degree. Since the maximum grandchild degree of a binomial tree of size 2^r is $r - 2$, and generally there are j grandchildren of degree $r - j - 1$, there are $\sum_{j=1}^k j = k(k + 1)/2$ grandchildren of degree $\geq r - k - 1$. Since $k(k + 1)/2 \geq L$, no grandchild of degree $\leq r - k - 2$ is pruned, i.e. the $(r - k)$ -th rightmost child w of x has degree $r - k - 1$ and loss zero. By the assumption on r , the degree of w is $\geq \lg n$ and T_w has size $\geq n$. It follows that T_x has size at least $n + 1$, which is a contradiction. This gives $r < k + 1 + \lg n \leq \sqrt{2L} + 2 + \lg n$, since $(k - 1)k/2 < L$. \square

By I3 we have $L \leq R + 1$. Since $\lg n + \sqrt{2(R + 1)} + 2 \leq R$ for $R = 2 \lg n + 6$, we have the following corollary:

COROLLARY 1. *All nodes have rank $\leq R$.*

The corollary implies a bound on the maximal rank before a heap operation. If we violate I2 or I3 temporarily during a heap operation, then the pigeonhole principle guarantees that we can apply the transformations described in Section 3. If the total loss is $> R + 1$, then there exists either a node with loss at least two, or there exist two nodes with equal rank each with loss exactly one. Similarly if there are $> R + 1$ active roots, then at least two active roots have the same rank. Finally, if I1-I3 are satisfied but the root violates I4, then the root has at least three passive linkable children, since the root has at most $R + 1$ children or grandchildren that can be active roots, i.e. at most $R + 1$ children of the root are active roots or passive non-linkable nodes.

3. TRANSFORMATIONS

The basic transformation is to *link* a node x and its subtree below another node y , by removing x from the child list of its current parent and making x a child of y . If x is active it is made the leftmost child of y ; if x is passive it is made the rightmost child of y .

The following transformations use link to reestablish the invariants I2-I4 when they get violated. The transformations are illustrated in Figure 1 and the main properties of the transformations are captured by Table 1.

Active root reduction Let x and y be active roots of equal rank r . Compare $x.\text{key}$ and $y.\text{key}$. Assume w.l.o.g. $x.\text{key} < y.\text{key}$. Link y to x and increase the rank of x by one. If the rightmost child z of x is passive make z a child of the root. In this transform the number of active roots is decreased by one and the degree of the root possibly increased by one.

Root degree reduction Let x, y, z be the three rightmost passive linkable 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 and y as active. Link z to y , and link y to x . Make x the leftmost child of the root. Assign both x and y loss zero, and rank one and zero respectively. In this transform both x and y change from being passive to active with loss zero, both get one more child, and x becomes a new active root. The degree of the root decreases by two and the number of active roots increases by one.

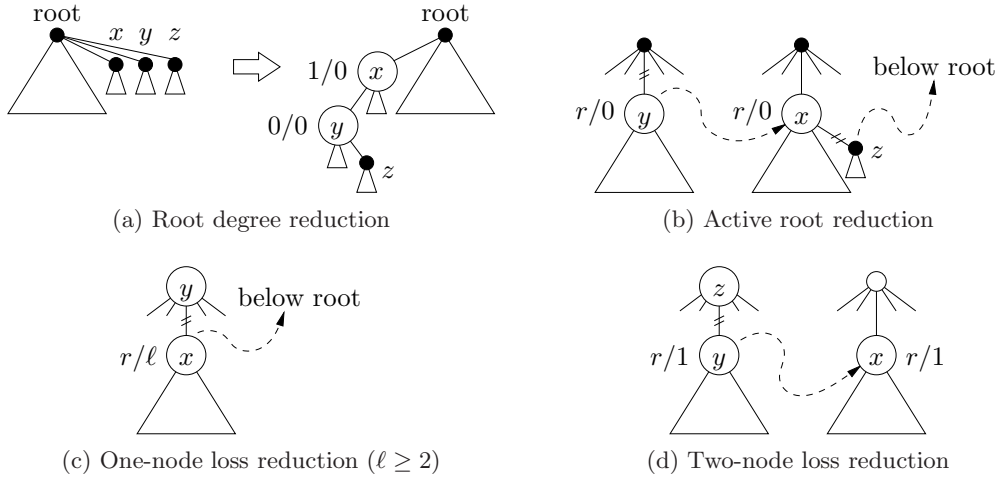


Figure 1: Transformations to reduce the root degree, the number of active roots, and the total loss. Black/white nodes are passive/active nodes. For an active node r/ℓ shows rank/loss.

Loss reduction To reduce the total loss we have two different transformations. The **one-node loss reduction** applies when there exists an active node x with loss ≥ 2 . Let y be the parent of x . In this case x is linked to the root and made an active root with loss zero, and the rank of y is increased by one. If y is not an active root, the loss of y is increased by one. Since the loss of x decreases by at least two, the total loss is decreased by at least one. The second transformation, **two-node loss reduction**, applies when two active nodes x and y with rank r both have a loss of exactly one. Compare $x.key$ and $y.key$. Assume w.l.o.g. $x.key < y.key$. Let z be the parent of y . Link y to x , increase the rank of x , and set the loss of x and y to zero. The degree and rank of z is decreased by one. If z is not an active root, the loss of z is increased by one.

Certain combinations of active root reductions and root degree reductions have only beneficial effects (see Table 1). 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. An active root reduction and a root degree reduction decrease the root degree by at least one. Two active root reductions and a root degree reduction decrease the number of active roots by one without increasing the root degree. Three active root reductions and two root degree reductions decrease both the number of active roots and the root degree by at least one.

It should be noted that the distinct key assumption together with the heap order invariant ensures that no cycles are created in the tree when an active root reduction or two-node loss reduction is performed.

4. IMPLEMENTATION OF THE HEAP OPERATIONS

The various heap operations are implemented as follows. To find the minimum in a heap, return the item in the root. To make an empty heap, return an empty tree. To insert an item into a heap, create a new one-node tree with a passive root containing the item, and meld this with the existing

heap. To delete an arbitrary item, decrease its key to minus infinity and do a minimum deletion.

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 we are done. Otherwise, if $x.key < z.key$, swap the items in x and z (actually we assume each node only stores a pointer to the item that is stored externally with a pointer to the node of the item). Let y be the parent of x . Make x a child of the root. If x was an active node but not an active root, then x becomes an active root with loss zero and the rank of y is decreased by one. If y is active but not an active root, then the loss of y is increased by one. Do a loss reduction if possible. Finally, do six active root reductions and four root degree reductions to the extent possible.

To delete the minimum in the tree with root z , first find the node x of minimum key among the children of the root. If x is active then make x passive and all active children of x become active roots. Make each of the other children of z a child of x . Make the passive linkable children of x the rightmost children of x . Remove x from Q . Destroy z . Repeat twice: move the front node y on Q to the back; link the two rightmost children of y to x , if they are passive. Do a loss reduction if possible. Do active root reductions and root degree reductions in any order until none of either is possible.

To meld two heaps with roots x and y , rename x and y if necessary so that the tree rooted at x has size at most the size of the tree rooted at y . Make all nodes in the tree rooted at x passive. (Do this implicitly, as described in Section 6, so that it takes $O(1)$ time.) Let u be the root of smaller key and v the other root. Make v a child of u . Set $Q = Q_x \& [v] \& Q_y$, where “&” denotes catenation and Q_x and Q_y are the queues of the heaps with root x and y respectively. Do an active root reduction and a 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 the presented data structure.

Table 1: Effect of the different transformations

	Root degree	Total loss	Active roots	Key comparisons
Active root reduction (A)	$\leq +1$	0	-1	+1
Root degree reduction (R)	-2	0	+1	+3
Loss reduction	$\leq +1$	≤ -1	$\leq +1$	$\leq +1$
– one-node	+1	≤ -1	+1	0
– two-node	0	-1	0	+1
(A) + (R)	≤ -1	0	0	+4
$2 \times (A) + (R)$	≤ 0	0	-1	+5
$3 \times (A) + 2 \times (R)$	≤ -1	0	-1	+9

Table 2: The changes caused by the different heap operations

	Root degree	Total loss	Active roots
decrease-key	$\leq 1 + 1 + 6 - 8$	$\leq 1 - 1 + 0 + 0$	$\leq 1 + 1 - 6 + 4$
meld	$\leq 1 + 0 + 1 - 2$	$\leq 0 + 0 + 0 + 0$	$\leq 0 + 0 - 1 + 1$
delete-min	$\leq (2 \lg n + 12 + 4) + 1$	$\leq 0 - 1$	$\leq R + 1$

5. CORRECTNESS

In the following we verify that each operation preserves the invariants I1-I4.

For I1 the interesting property to verify is that the i -th active child of an active node has $\text{rank} + \text{loss} \geq i - 1$. All other properties in I1 are straightforward to verify. We start by observing that if an active child x is detached from its parent y satisfying I1, then I1 is also satisfied for y after the detachment. This follows since the i -th rightmost active child z after the detachment was either the i -th or $(i + 1)$ -st active child before the detachment, i.e. for z the new $\text{rank} + \text{loss}$ is at least $i - 1$. An active node only gets a new active child as a result of an active root reduction, two-node loss reduction, or root degree reduction. In the first two cases a new $(r + 1)$ -st rightmost active child is added with rank r (and loss zero), and in the later case I1 holds by construction for the two new active nodes.

For invariants I2 and I3, Table 2 captures the change to the degree of the root, the total loss, and the number of active roots when performing the operations decrease-key, meld, and delete-min, respectively. Each entry is a sum of four terms stating the change caused by the initial transformations performed by the operations, and by the loss reduction transformations, active root reductions, and root degree reductions. Each entry is an upper bound on the change, except for the cases where no reduction is possible (e.g. the loss increases, but is still $\leq R + 1$). For meld the root degree is the change to the old root that becomes the new root, whereas total loss and number of active roots is compared to the heap that is not made passive. For delete-min the bounds are stated before the repeated active root and root degree transformations are applied.

Observe that for both decrease-key and meld all sums are zero, and that R increases during a meld, i.e. invariants I2 and I3 remain valid for each of these operations. Delete-min reduces the size of the heap by one, reducing $R = 2 \lg n + 6$ by at most one, if $n \geq 4$ before the delete-min operation (if $n \leq 3$ before the delete-min operation, I2 and I3 are trivially true after). The reduction in loss by one ensures that I3 is valid after delete-min. Since active root reductions are performed until they are not possible, I2 trivially holds —

provided that the repeated application of active root reductions and root degree reductions terminate. Termination immediately follows from the facts that both reductions reduce the measure

$$2 \cdot \text{root degree} + 3 \cdot \#\text{active roots}$$

by one, and that the initial value of this measure is $O(\log n)$. This immediately also implies the claimed time bounds for our heap.

To prove the validity of I4, we first observe that for the degree of the root we can use the same argument as above using Table 2.

During the transformations a non-root node can only increase its degree in three cases. During an active root reduction there is no passive right-child to detach. In this case all children of the node are active, and the degree bound follows from Corollary 1. During a two-node loss reduction the degree of the node x increases by one, but this is okay by I4 since the loss of the node decreases from one to zero. During a root degree transformation two passive nodes become active, both getting loss zero and degree increased by one. Again this is okay by I4. During a meld one root becomes a non-root, but since $R + 3 \leq 2 \lg(2n - p) + 9$ for all possible p , again I4 holds. The interesting case is when we perform a delete-min operation. I4 holds for the root trivially, since we repeatedly perform root degree reductions until none are possible. For non-root nodes we observe that their invariant is strengthened since R decreases. The role of Q is to deal with this case. By removing the two first nodes in the queue, all nodes get their position in Q decreased by two or three (depending if they were in front of the deleted node in the queue). By observing that $2n - p$ does not increase in this case, it follows that the invariant for non-root nodes is not strengthened and I4 remains valid. For the two nodes moved to the end of the queue the term $2 \lg(2n - p)$ decreases by two, implying that their degree constraint is strengthened by two. Since we detach two passive nodes from these nodes I4 remains valid for these nodes also. During meld all elements in the smaller heap become passive, and their degree constraint goes from the “+10” to the “+9” case. But since they remain in their position in the queue, and the result-

ing queue is at least twice the size, we have that $2n - p$ increases by a factor two, and I4 remains valid for the elements in the smaller heap. For the elements in the larger of the two heaps, they keep their active status. Both n and p increase for these elements by the size of the smaller queue. Therefore $2n - p$ is non-decreasing and I4 remains valid.

A note on the loss: In the previous description the loss of a node is assumed to be a non-negative integer. Even though the loss plays an essential role in invariant I1, only values 0, 1, and $\lceil \frac{1}{2} \rceil$ are relevant for the algorithm. As soon as the loss is ≥ 2 , then the loss can only decrease when it is set to zero by a one-node loss transformation. Since the algorithm only tests if the loss is zero, one or ≥ 2 , it is sufficient for the algorithm to keep track of these three states. It follows that we can store the loss using only two bits.

6. REPRESENTATION DETAILS

To represent a heap we have the following types of records, also illustrated in Figures 2 and 3. Each node is represented by a *node record*. To mark if a node is active or passive we will not store the flag directly in the node but indirectly in an *active record*. In particular all active nodes of a tree point to the same active record. This allows all nodes of a tree to be made passive in $O(1)$ time by only changing a single flag. The entry point to a heap is a pointer to a *heap record*, that has a pointer to the root of the tree and additional information required for accessing the relevant nodes for performing the operations described in Section 4.

We call a rank *active-root transformable*, if there are at least two active roots of that rank. We call a rank *loss transformable*, if the total loss of the nodes of that rank is at least two. For each rank we maintain a node, and all such nodes belong to the *rank-list*. The rightmost node corresponds to rank zero, and the node that corresponds to rank k is the left sibling of the one that corresponds to $k - 1$. (see Figure 3). We maintain all active nodes that potentially could participate in one of the transformations from Section 3 (i.e. active roots and active nodes with positive loss) in a list called the *fix-list*. Each node with rank k on the fix-list points to node k of the rank-list. The fix-list is divided left-to-right into four parts (1-4), where Parts 1-2 contain active roots and Parts 3-4 contain nodes with positive loss. Part 1 contains the active roots of active-root transformable ranks. All the active roots of the same rank are adjacent, and one of the nodes has a pointer from the corresponding node of the rank-list. Part 2 contains the remaining active roots. Each node has a pointer from the corresponding node of the rank-list. Part 3 contains active nodes with loss one and a rank that is not loss-transformable. Each node of this part has a pointer from the corresponding node of the rank-list. Finally, Part 4 contains all the active nodes of loss transformable rank. As in Part 1, all nodes of equal rank are adjacent and one of the nodes is pointed to by the corresponding node of the rank-list. Observe that for some ranks there may exist only one node in Part 1 (because if the loss of a node is at least two, its rank is loss-transformable).

From the above description it follows that we can always perform an active root reduction as long as Part 1 of the fix-list is nonempty, and we can always perform a loss reduction transformation as long as Part 4 is nonempty. We maintain a pointer (in the heap record) indicating the boundary between Parts 2 and 3. The above construction and pointers (see Figure 3 for more details) allow us to take the following

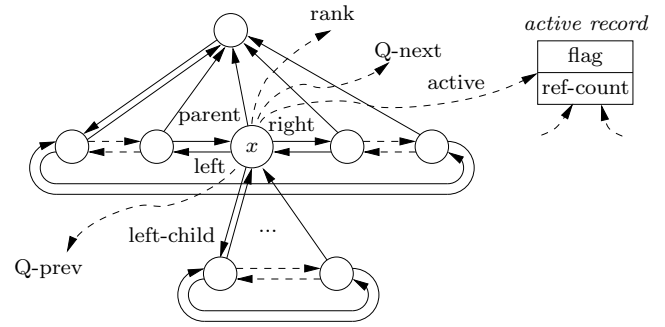


Figure 2: The fields of a node record x and an active record. The fields of x not shown are the item and the loss.

actions: In order to perform a loss reduction transformation, we go to the right-end of the fix-list and perform a one-node loss reduction (if we access a node of multiple loss) or a two-node loss reduction (if the two rightmost nodes of the fix-list have loss one). Otherwise Part 4 is empty. After a loss reduction transformation on a rank k , we may have to transfer one node of rank k into Part 3, if its loss is one and it is the last node in Part 3 with this rank. Whenever the loss of a node that has a loss-transformable rank increases, we insert it into (the appropriate group of) Part 4. If its rank is not loss transformable, we insert it into Part 3, unless there is another node of the same rank there, in which case we move both nodes into Part 4 at the right end of the fix-list.

In order to perform an active root reduction, we go to the left end of the fix-list and link the two leftmost active roots, if they have the same rank (otherwise, Part 1 is empty). If after the reduction, the two leftmost nodes in the fix-list are not active roots of equal degree, we transfer the leftmost node into Part 2. When an active node of rank k becomes an active root and there is no other active root of the same rank, we insert the new active root into Part 2. Otherwise, we insert it adjacent to the active roots of that rank, unless only one active root has this rank (i.e. it is located in Part 2), in which case we transfer the existing active root of rank k with the new one into Part 1 (at the left end of the fix-list). When the rank of a node that belongs to the fix-list changes, we can easily perform the necessary updates to the fix-list so that all parts of the list are consistent with the above description. The details are straightforward and thus omitted.

The details of the fields of the individual records are as follows (see also Figures 2 and 3).

Node record

- item** A pointer to the item (including its associated key).
- 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 linked list), the parent node, and the leftmost child. The later two are NULL if the nodes do not exist.
- active** Pointer to an active record, indicating if the node is active or passive.
- Q-prev, Q-next** Pointers to the node records for the previous and the next node on the queue Q , which is maintained as a cyclic linked list.

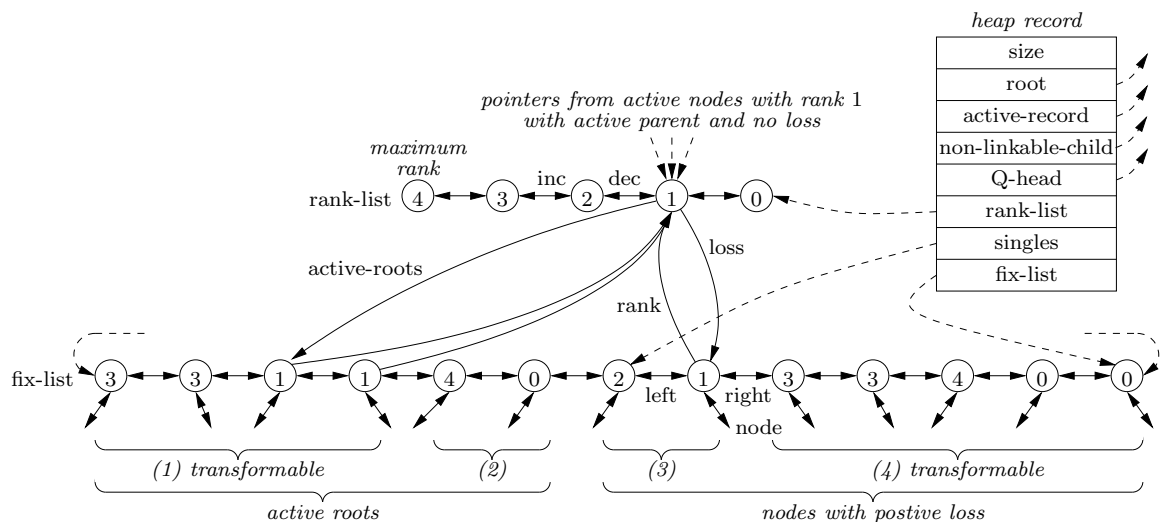


Figure 3: The heap-record, rank-list, and fix-list. The ref-count field in the records of the rank-list is not shown. Only pointers for nodes with rank equal to one are shown. The numbers in the nodes in the rank-list and fix-list are the ranks of the active nodes pointing to these nodes; these numbers are not stored.

loss Non-negative integer equal to the loss of the node. Undefined if the node is passive.

rank If the node is passive, the value of the pointer is not defined (it points to some old node that has been released for garbage collection). If the node is an active root or an active node with positive loss (i.e. it is on the fix-list), rank points to the corresponding record in the rank-list. Otherwise rank points to the record in the rank-list corresponding to the rank of the node. (The cases can be distinguished using the active field of the node and the parent together with the loss field).

Active record

flag A Boolean value. Nodes pointing to this record are active if and only if the flag is true.

ref-count The number of nodes pointing to the record. If flag is false and ref-count = 0, then the node is released for garbage collection.

Heap record

size An integer equal to the number of items in the heap.

root A pointer to the node record of the root (NULL if and only if the heap is empty).

active-record A pointer to the active record shared by all active nodes in the heap (one distinct active record for each heap).

non-linkable-child A pointer to the node record of the leftmost passive non-linkable child of the root. If all passive children of the root are linkable the pointer is to the rightmost active child of the root. Otherwise it is NULL.

Q-head A pointer to the node record for the node that is the head of the queue Q .

rank-list A pointer to the rightmost record in the rank-list.

fix-list A pointer to the rightmost node in the fix-list.

singles A pointer to the leftmost node in the fix-list with a positive loss, if such a node exists. Otherwise it is NULL.

Rank-list record (representing rank r)

inc, dec Pointers to the records on the rank-list for rank $r + 1$ and $r - 1$, if they exist. Otherwise they are NULL.

loss A pointer to a record in the fix-list for an active node with rank r and positive loss. NULL if no such node exists.

active-roots A pointer to a record in the fix-list for an active root with rank r . NULL if no such node exists.

ref-count The number of node records and fix-list records pointing to this record. If the leftmost record on the rank-list gets a ref-count = 0, then the record is deleted from the rank-list and is released for garbage collection.

Fix-list record

node A pointer to the node record for the node.

left, right Pointers to the left and right siblings on the fix-list, that is maintained as a cyclic linked list.

rank A pointer to the record in the rank-list corresponding to the rank of this node.

A detail concerning garbage collection: When performing a meld operation on two heaps, all nodes in one heap are first made passive by clearing the flag in the active record given by the heap record. The heap record, the rank-list, and the fix-list for this heap are released for incremental garbage collection.

We now bound the space required by our structure. For each item we have one node record and possibly one fix-list record. The number of active records is bounded by the number of nodes, since in the worst case each active record has a ref-count = 1. Finally for each heap we have one heap-record and a number of rank-list records bounded by one plus the maximum rank of a node, i.e. logarithmic in the size of the heap. It follows that the total space for a heap is linear in the number of stored items.

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. Based on the active/passive node idea we have considered many alternative implementations. One option is to allow a third explicit marking “active root” (instead of an active root being a function of the active/passive marks), such that a child of an active node can also be an active root. This eliminates the distinction between passive linkable and non-linkable nodes. Another option is to allow the root to be active also. This simplifies the decrease-key operation, since then the node getting a smaller key can also be made the root, without ever swapping items in the nodes. A third option is to adopt redundant counters instead of the pigeon-hole principle to bound the number of active roots and holes created by cutting of subtrees. Further alternatives are to consider ternary linking instead of binary linking, and using a different “inactivation” criterion during meld, e.g. size plus number of active nodes. All these variations can be combined, each solution implying different bounds on the maximum degrees, constants in the time bounds, and complexity in the reduction transformations. In the presented solution we aimed at reducing the complexity in the description, whereas the constants in the solution were of secondary interest.

8. REFERENCES

- [1] Gerth Stølting Brodal. Fast meldable priority queues. In *Proc. 4th International Workshop Algorithms and Data Structures*, volume 955 of *Lecture Notes in Computer Science*, pages 282–290. Springer, 1995.
- [2] Gerth Stølting Brodal. Worst-case efficient priority queues. In *Proc. 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 52–58. SIAM, 1996.
- [3] Svante Carlsson, J. Ian Munro, and Patricio V. Poblete. An implicit binomial queue with constant insertion time. In *Proc. 1st Scandinavian Workshop on Algorithm Theory*, volume 318 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 1988.
- [4] Timothy M. Chan. Quake heaps: a simple alternative to Fibonacci heaps. Manuscript, 2009.
- [5] Clark Allan Crane. *Linear lists and priority queues as balanced binary trees*. PhD thesis, Stanford University, Stanford, CA, USA, 1972.
- [6] James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert E. Tarjan. Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31(11):1343–1354, 1988.
- [7] Amr Elmasry. Layered heaps. In *Proc. 9th Scandinavian Workshop on Algorithm Theory*, volume 3111 of *Lecture Notes in Computer Science*, pages 212–222. Springer, 2004.
- [8] Amr Elmasry. Parameterized self-adjusting heaps. *J. Algorithms*, 52(2):103–119, 2004.
- [9] Amr Elmasry. Pairing heaps with $o(\log \log n)$ decrease cost. In *Proc. 20th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 471–476. SIAM, 2009.
- [10] Amr Elmasry. The violation heap: A relaxed Fibonacci-like heap. In *Proc. 16th Annual International Conference on Computing and Combinatorics*, volume 6196 of *Lecture Notes in Computer Science*, pages 479–488. Springer, 2010.
- [11] Amr Elmasry, Claus Jensen, and Jyrki Katajainen. On the power of structural violations in priority queues. In *Proc. 13th Computing: The Australasian Theory Symposium.*, volume 65 of *CRPIT*, pages 45–53. Australian Computer Society, 2007.
- [12] Amr Elmasry, Claus Jensen, and Jyrki Katajainen. Two-tier relaxed heaps. *Acta Informatica*, 45(3):193–210, 2008.
- [13] Michael L. Fredman. On the efficiency of pairing heaps and related data structures. *Journal of the ACM*, 46(4):473–501, 1999.
- [14] Michael L. Fredman, Robert Sedgewick, Daniel Dominic Sleator, and Robert Endre Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.
- [15] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.
- [16] Bernhard Haeupler, Siddhartha Sen, and Robert Endre Tarjan. Rank-pairing heaps. In *Proc. 17th Annual European Symposium on Algorithms*, volume 5757 of *Lecture Notes in Computer Science*, pages 659–670. Springer, 2009. *SIAM Journal of Computing*, to appear.
- [17] Peter Höyer. A general technique for implementation of efficient priority queues. In *Proc. Third Israel Symposium on the Theory of Computing and Systems*, pages 57–66, 1995.
- [18] Haim Kaplan and Robert E. Tarjan. New heap data structures. Technical report, Department of Computer Science, Princeton University, 1999.
- [19] Haim Kaplan and Robert Endre Tarjan. Thin heaps, thick heaps. *ACM Transactions on Algorithms*, 4(1), 2008.
- [20] Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms, 2nd Edition*. Addison-Wesley, 1973.
- [21] Gary L. 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.
- [22] Seth Pettie. Towards a final analysis of pairing heaps. In *Proc. 46th Annual IEEE Symposium on Foundations of Computer Science*, pages 174–183. IEEE Computer Society, 2005.
- [23] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting heaps. *SIAM Journal of Computing*, 15(1):52–69, 1986.
- [24] Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, 1978.
- [25] John William Joseph Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.