Hollow Heaps

Thomas Dueholm Hansen[1], Haim Kaplan[2(✉)], Robert E. Tarjan[3,4], and Uri Zwick[2]

[1] Department of Computer Science, Aarhus University, Aarhus, Denmark
tdh@cs.au.dk
[2] Blavatnik School of Computer Science, Tel Aviv University, Tel Aviv-Yafo, Israel
zwick@tau.ac.il, haimk@post.tau.ac.il
[3] Department of Computer Science, Princeton University, Princeton, NJ 08540, USA
[4] Intertrust Technologies, Sunnyvale, CA 94085, USA
ret@CS.Princeton.EDU

Abstract. We introduce the *hollow heap*, a very simple data structure with the same amortized efficiency as the classical Fibonacci heap. All heap operations except *delete* and *delete-min* take $O(1)$ time, worst case as well as amortized; *delete* and *delete-min* take $O(\log n)$ amortized time. Hollow heaps are by far the simplest structure to achieve this. Hollow heaps combine two novel ideas: the use of lazy deletion and re-insertion to do *decrease-key* operations, and the use of a dag (directed acyclic graph) instead of a tree or set of trees to represent a heap. Lazy deletion produces hollow nodes (nodes without items), giving the data structure its name.

1 Introduction

A *heap* is a data structure consisting of a set of *items*, each with a *key* selected from a totally ordered universe. Heaps support the following operations:

make-heap(): Return a new, empty heap.
find-min(h) : Return an item of minimum key in heap h, or *null* if h is empty.
insert(e, k, h): Return a heap formed from heap h by inserting item e, with key k. Item e must be in no heap.
delete-min(h): Return a heap formed from non-empty heap h by deleting the item returned by *find-min*(h).
meld(h_1, h_2): Return a heap containing all items in item-disjoint heaps h_1 and h_2.
decrease-key(e, k, h): Given that e is an item in heap h with key greater than k, return a heap formed from h by changing the key of e to k.
delete(e, h) : Return a heap formed by deleting e, assumed to be in h, from h.

The original heap h passed to *insert*, *delete-min*, *decrease-key*, and *delete*, and the heaps h_1 and h_2 passed to *meld*, are destroyed by the operations. Heaps do *not* support search by key; operations *decrease-key* and *delete* are given the location of item e in heap h. The parameter h can be omitted from *decrease-key* and *delete*, but then to make *decrease-key* operations efficient if there are intermixed *meld* operations, a separate disjoint set data structure is needed to keep track of the partition of items into heaps. (See the discussion in [12].)

M.M. Halldórsson et al. (Eds.): ICALP 2015, Part I, LNCS 9134, pp. 689–700, 2015.
DOI: 10.1007/978-3-662-47672-7_56

Fredman and Tarjan [8] invented the *Fibonacci heap*, an implementation of heaps that supports *delete-min* and *delete* on an n-item heap in $O(\log n)$ amortized time and each of the other operations in $O(1)$ amortized time. Applications of Fibonacci heaps include a fast implementation of Dijkstra's shortest path algorithm [4,8] and fast algorithms for undirected and directed minimum spanning trees [6,9]. Since the invention of Fibonacci heaps, a number of other heap implementations with the same amortized time bounds have been proposed [1–3,7,10,11,13,16,18]. Notably, Brodal [1] invented a very complicated heap implementation that achieves the time bounds of Fibonacci heaps in the worst case. Brodal et al. [2] later simplified this data structure, but it is still significantly more complicated than any of the amortized-efficient structures. For further discussion of these and related results, see [10]. We focus here on the *amortized* efficiency of heaps.

In spite of its many competitors, Fibonacci heaps remain one of the simplest heap implementations to describe and code, and are taught in numerous undergraduate and graduate data structures courses. We present *hollow heaps*, a data structure that we believe surpasses Fibonacci heaps in its simplicity. Our data structure has two novelties: it uses lazy deletion to do *decrease-key* operations in a simple and natural way, avoiding the *cascading cut* process used by Fibonacci heaps, and it represents a heap by a dag (directed acyclic graph) instead of a tree or a set of trees. The amortized analysis of hollow heaps is simple, yet nontrivial. We believe that simplifying fundamental data structures, while retaining their performance, is an important endeavor.

In a Fibonacci heap, a *decrease-key* produces a heap-order violation if the new key is less than that of the parent node. This causes a *cut* of the violating node and its subtree from its parent. Such cuts can eventually destroy the "balance" of the data structure. To maintain balance, each such cut may trigger a cascade of cuts at ancestors of the originally cut node. The cutting process results in loss of information about the outcomes of previous comparisons. It also makes the worst-case time of a *decrease-key* operation $\Theta(n)$ (although modifying the data structure reduces this to $\Theta(\log n)$; see e.g., [14]). In a hollow heap, the item whose key decreases is merely moved to a new node, preserving the existing structure. Doing such lazy deletions carefully is what makes hollow heaps simple but efficient.

The remainder of this paper consists of six sections. Section 2 describes hollow heaps at a high level. Section 3 analyzes them. Section 4 presents an alternative version of hollow heaps that uses a tree representation instead of a dag representation. Section 5 describes a rebuilding process that can be used to improve the time and space efficiency of hollow heaps. Section 6 gives implementation details for the data structure in Section 2. The full version of this paper also contains implementation details of the data structure in Section 4 and further explores the design space of the data structures, identifying variants that are efficient and variants that are not.

2 Hollow Heaps

Our data structure extends and refines a well-known generic representation of heaps. The structure is *exogenous* rather than *endogenous* [19]: nodes *hold* items rather than *being* items. Moving items among nodes precludes the possibility of making the data structure endogenous.

Many previous heap implementations, including Fibonacci heaps, represent a heap by a set of heap-ordered trees: each node holds an item, with each child holding an item having key no less than that of the item in its parent. We extend this idea from trees to dags, and to dags whose nodes may or may not hold items. Since the data structure is an extension of a tree, we extend standard tree terminology to describe it. If (u, v) is a dag arc, we say u is a *parent* of v and v is a *child* of u. A node that is not a child of any other node is a *root*.

We represent a non-empty heap by a dag whose nodes hold the heap items, at most one per node. If e is an item, $e.node$ is the node holding e. We call a node *full* if it holds an item and *hollow* if not. If u is a full node, $u.item$ is the item u holds. Thus if e is an item, $e.node.item = e$. A node is full when created but can later become hollow, by having its item moved to a newly created node or deleted. A hollow node remains hollow until it is destroyed. Each node, full or hollow, has a key. The key of a full node is the key of the item it holds. The key of a hollow node is the key of the item it once held, just before that item was moved to another node or deleted. A full node is a child of at most one other node; a hollow node is a child of at most two other nodes.

The dag is topologically ordered by key: if u is a parent of v, then $u.key \leq v.key$. Henceforth we call this *heap order*. Except in the middle of a *delete* operation, the dag has one full root and no hollow roots. Heap order guarantees that the root holds an item of minimum key. We access the dag via its root. We call the item in the root the *root item*.

We do the heap operations with the help of the *link* primitive. Given two full roots v and w, $link(v, w)$ compares the keys of v and w and makes the root of larger key a child of the other; if the keys are equal, it makes v a child of w. The new child is the *loser* of the link, its new parent is the *winner*. Linking eliminates one full root, preserves heap order, and gives the loser a parent, its *first parent*.

To make a heap, return an empty dag. To do *find-min*, return the item in the root. To meld two heaps, if one is empty return the other; if both are non-empty, link the roots of their dags and return the winner. To insert an item into a heap, create a new node, store the item in it (making the node full), and meld the resulting one-node heap with the existing heap.

We do *decrease-key* and *delete* operations using lazy deletion. To decrease the key of item e in heap h to k, let $u = e.node$. If $u = h$ (u is the root of the dag), merely set $u.key = k$. Otherwise (u is a child), proceed as follows. Create a new node v; move e from u to v, making u hollow; set $v.key = k$; do $link(h, v)$; and, if v is the loser of this link, make u a child of v. If u becomes a child of v, then v is the *second parent* of u, in contrast to its first parent, previously acquired via a

link with a full node. A node only becomes hollow once, so it acquires a second parent at most once.

Remark. The arc (v, u) added to the dag by decrease-key represents the inequality $v.key < u.key$. If such arcs are not added, the resulting algorithm does not have the desired efficiency, as we show in the full version of this paper.

To do a *delete-min*, do a *find-min* followed by a deletion of the returned item. To delete an item e, remove e from the node holding it, say u, making u hollow. A node u made hollow in this way never acquires a second parent. If u is not the root of the dag, the deletion is complete. Otherwise, repeatedly destroy hollow roots and link full roots until there are no hollow roots and at most one full root. The proof of the following theorem is immediate.

Theorem 1. *The hollow heap operations perform the heap operations correctly and maintain the invariants that the graph representing a heap is a heap-ordered dag; each full node has at most one parent; each hollow node has at most two parents; and, except in the middle of a delete operation, the dag representing a heap has no hollow roots and at most one full root.*

The only flexibility in this implementation is the choice of which links to do in deletions of root items. To keep the number of links small, we give each node u a non-negative integer rank $u.rank$. We use ranks in a special kind of link called a *ranked link*. A ranked link of two roots is allowed only if they have the same rank; it links them and increases the rank of the winner (the remaining root) by 1. In contrast to a ranked link, an *unranked link* links any two roots and changes no ranks. We call a child *ranked* or *unranked* if it most recently acquired a first parent via a ranked or unranked link, respectively.

When linking two roots of equal rank, we can do either a ranked or an unranked link. We do ranked links only when needed to guarantee efficiency. Specifically, links in *meld* and *decrease-key* are unranked. Each *delete-min* operation destroys hollow roots and does ranked links until none are possible (there are no hollow roots and all full roots have different ranks); then it does unranked links until there is at most one root.

The last design choice is the initial node ranks. We give a node created by an *insert* a rank of 0. In a *decrease-key* that moves an item from a node u to a new node v, we give v a rank of $\max\{0, u.rank - 2\}$. The latter choice is what makes hollow heaps efficient.

We conclude this section by mentioning some benefits of using hollow nodes and a dag representation. Hollow nodes allow us to treat *decrease-key* as a special kind of insertion, allowing us to avoid cutting subtrees as in Fibonacci heaps. As a consequence, *decrease-key* takes $O(1)$ time worst case: there are no cascading cuts as in [8], no cascading rank changes as in [10,14], and no restructuring steps to eliminate heap-order violations as in [2,5,13]. The dag representation explicitly maintains all key comparisons between undeleted items, allowing us to avoid restructuring altogether: links are cut only when hollow roots are destroyed.

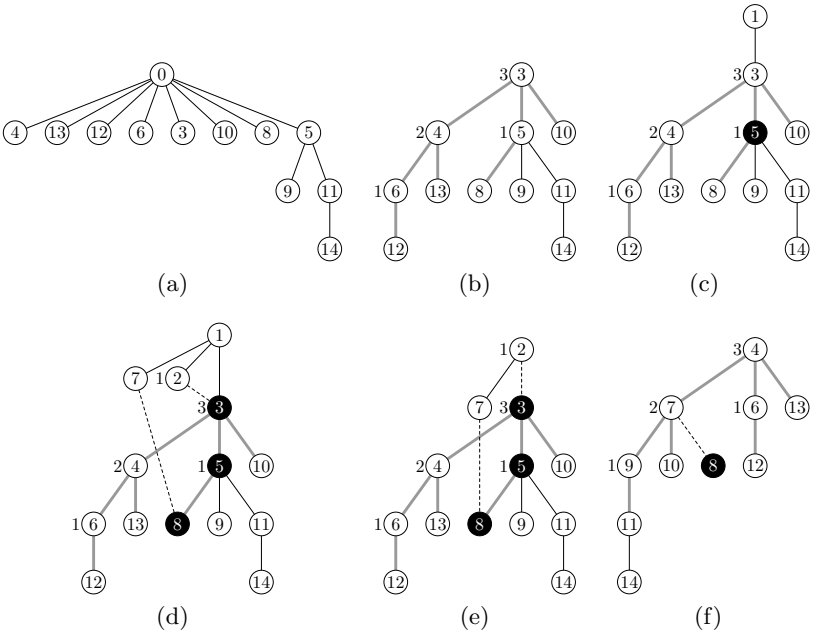

Fig. 1. Operations on a hollow heap. Numbers in nodes are keys; black nodes are hollow. Bold gray, solid, and dashed lines denote ranked links, unranked links, and second parents, respectively. Numbers next to nodes are non-zero ranks. (a) Successive insertions of items with keys 14, 11, 5, 9, 0, 8, 10, 3, 6, 12, 13, 4 into an initially empty heap. (b) After a *delete-min* operation. All links during the *delete-min* are ranked. (c) After a decrease of key 5 to 1. (d) After a decrease of key 3 to 2 followed by a decrease of key 8 to 7. The two new hollow nodes both have two parents. (e) After a second *delete-min*. The only hollow node that becomes a root is the original root. One unranked link, between the nodes holding keys 2 and 7 occurs. (f) After a third *delete-min*. Two hollow nodes become roots; the other loses one parent. All links are ranked.

3 Analysis

The most mysterious detail of hollow heaps is the way ranks are updated in *decrease-key* operations. Our analysis reveals the reason for this choice. We need to show that the rank of a heap node is at most logarithmic in the number of nodes in the dag representing the heap, and that the amortized number of ranked children per node is also at most logarithmic.

To do both, we assign *virtual parents* to certain nodes. We use virtual parents in the analysis only; they are not part of the data structure in Section 2. (Section 4 presents a version of hollow heaps that *does* use them.)

A node may acquire a virtual parent, have its virtual parent changed, or lose its virtual parent. As we shall see, virtual parents define a *virtual forest*. In particular, each node has at most one virtual parent at a time. If v is the virtual parent of u, we say that u is a *virtual child* of v. A node u is a *virtual descendant* of a node v if there is a path from v to u via virtual children.

When a node is created, it has no virtual parent. When a root u loses a link to a node v, v becomes the virtual parent of u (as well as its first parent). If u already has a virtual parent, v replaces it. (By Lemma 1 below, a root cannot have a virtual parent, so such a replacement never happens.) When a *decrease-key* moves an item from a node u to a new node v, if u has more than two ranked virtual children, two of its ranked virtual children of highest ranks remain virtual children of u, and the rest of its virtual children become virtual children of v. (By Lemma 2 below, the ranked virtual children of a node have distinct ranks, so the two that remain virtual children of u are uniquely defined.) If the virtual parent of a node u is destroyed, u loses its virtual parent. If u is full it can subsequently acquire a new virtual parent by losing a link.

Lemma 1. *If w is a virtual child of u, there is a path in the dag from u to w.*

Proof. We prove the lemma for a given node w by induction on time. When w is created it has no virtual parent. It may acquire a virtual parent only by losing a link to a node u, which then becomes both its parent and its virtual parent, so the lemma holds after the link. Suppose that u is currently the virtual parent of w. By the induction hypothesis, there is a path from u to w in the dag, so w is not a root and cannot participate in link operations. The virtual parent of w can change only as a result of a *decrease-key* operation on the item $e = u.item$. If $u \neq h$, such a *decrease-key* operation creates a new node v, moves e to v, and then links v and h. The operation may also make v the new virtual parent of w. If v wins the link, it becomes the unique root, so there is a path from v to w in the dag. If v loses the link, the arc (v, u) is added to the dag, making v the second parent of u. Since there was a path in the dag from u to w, there is now also a path from v to w. Finally, note that dag arcs are only destroyed when hollow roots are destroyed. Thus a path to w from its virtual parent u in the dag, present when u becomes the virtual parent of w, cannot be destroyed unless u is destroyed, in which case w loses its virtual parent, so the lemma holds vacuously. □

Corollary 1. *Virtual parents define a forest. If w is a root of the dag, it has no virtual parent. If w is a virtual child of u, then w stops being a virtual child of u only when u is destroyed or when a decrease-key operation is applied to the item residing in u.*

Lemma 2. *Let u be a node of rank r. If u is full, or u is a node made hollow by a delete, u has exactly one ranked virtual child of each rank from 0 to $r - 1$ inclusive, and none of rank r or greater. If u was made hollow by a decrease-key and $r > 1$, u has exactly two ranked virtual children, of ranks $r - 1$ and $r - 2$. If u was made hollow by a decrease-key and $r = 1$, u has exactly one ranked virtual child, of rank 0. If u was made hollow by a decrease-key and $r = 0$, u has no ranked virtual children.*

Proof. The proof is by induction on the number of operations. The lemma is immediate for nodes created by insertions. Both ranked and unranked links preserve the truth of the lemma, as does the removal of an item from a node

by a *delete*. By Corollary 1, a node loses virtual children only as a result of a *decrease-key* operation. Suppose the lemma is true before a *decrease-key* on the item in a node u of rank r. By the induction hypothesis, u has exactly one ranked virtual child of rank i for $0 \leq i < r$, and none of rank r or greater. If the *decrease-key* makes u hollow, the new node v created by the *decrease-key* has rank $\max\{0, u.rank - 2\}$, and v acquires all the virtual children of u except the two ranked virtual children of ranks $r-1$ and $r-2$ if $r > 1$, or the one ranked virtual child of rank 0 if $r = 1$. Thus the lemma holds after the *decrease-key*. □

Recall the definition of the Fibonacci numbers: $F_0 = 0$, $F_1 = 1$, $F_i = F_{i-1} + F_{i-2}$ for $i \geq 2$. These numbers satisfy $F_{i+2} \geq \phi^i$, where $\phi = (1 + \sqrt{5})/2$ is the golden ratio [15].

Corollary 2. *A node of rank r has at least $F_{r+3} - 1$ virtual descendants.*

Proof. The proof is by induction on r using Lemma 2. The corollary is immediate for $r = 0$ and $r = 1$. If $r > 1$, the virtual descendants of a node u of rank r include itself and all virtual descendants of its virtual children v and w of ranks $r-1$ and $r-2$, which it has by Lemma 2. By Corollary 1, virtual parents define a forest, so the sets of virtual descendants of v and w are disjoint. By the induction hypothesis, u has at least $1 + F_{r+2} - 1 + F_{r+1} - 1 = F_{r+3} - 1$ virtual descendants. □

Theorem 2. *The maximum rank of a node in a hollow heap of N nodes is at most $\log_\phi N$.*

Proof. Immediate from Corollary 2 since $F_{r+3} - 1 \geq F_{r+2} \geq \phi^r$ for $r \geq 0$. □

To complete our analysis, we need to bound the time of an arbitrary sequence of heap operations that starts with no heaps. It is straightforward to implement the operations so that the worst-case time per operation other than *delete-min* and *delete* is $O(1)$, and that of a *delete* on a heap of N nodes is $O(1)$ plus $O(1)$ per hollow node that loses a parent plus $O(1)$ per link plus $O(\log N)$. In Section 6 we give an implementation that satisfies these bounds and is space-efficient. We shall show that the amortized time for a *delete* on a heap of N nodes is $O(\log N)$ by charging the parent losses of hollow nodes and some of the links to other operations, $O(1)$ per operation.

Suppose a hollow node u loses a parent in a *delete*. This either makes u a root, in which case u is destroyed by the same *delete*, or it reduces the number of parents of u from two to one. We charge the former case to the *insert* or *decrease-key* that created u, and the latter case to the *decrease-key* that gave u its second parent. Since an *insert* or *decrease-key* can create at most one node, and a *decrease-key* can give at most one node a second parent, the total charge, and hence the total number of parent losses of hollow nodes, is at most 1 per *insert* and 2 per *decrease-key*.

A *delete* does unranked links only once there is at most one root per rank. Thus the number of unranked links is at most the maximum node rank, which is at most $\log_\phi N$ by Theorem 2. To bound the number of ranked links, we use a

potential argument. We give each root and each unranked child a potential of 1. We give a ranked child a potential of 0 if it has a full virtual parent, 1 otherwise (its virtual parent is hollow or has been deleted). We define the potential of a set of dags to be the sum of the potentials of their nodes. With this definition the initial potential is 0 (there are no nodes), and the potential is always non-negative. Each ranked link reduces the potential by 1: a root becomes a ranked child of a full node. It follows that the total number of ranked links over a sequence of operations is at most the sum of the increases in potential produced by the operations.

An unranked link does not change the potential: a root becomes an unranked child. An *insert* increases the potential by 1: it creates a new root (+1) and does an unranked link (+0). A *decrease-key* increases the potential by at most 3: it creates a new root (+1), it creates a hollow node that has at most two ranked virtual children by Lemma 2 (+2), and it does an unranked link (+0). Removing the item in a node u during a *delete* increases the potential by $u.rank$, also by Lemma 2: each of the $u.rank$ ranked virtual children of u gains 1 in potential. By Theorem 2, $u.rank = O(\log N)$. We conclude that the total number of ranked links is at most 1 per *insert* plus 3 per *decrease-key* plus $O(\log N)$ per *delete* on a heap with N nodes. Combining our bounds gives the following theorem:

Theorem 3. *The amortized time per hollow heap operation is $O(1)$ for each operation other than a delete, and $O(\log N)$ per delete on a heap of N nodes.*

4 Eager Hollow Heaps

It is natural to ask whether there is a way to represent a hollow heap by a tree instead of a dag. The answer is yes: we maintain the structure defined by the virtual parents instead of that defined by the parents. We call this the *eager version* of hollow heaps: it moves children among nodes, which the *lazy version* in Section 2 does not do. As a result it can do different links than the lazy version, but it has the same amortized efficiency.

To obtain eager hollow heaps, we modify *decrease-key* as follows: When a new node v is created to hold the item previously in a node u, if $u.rank > 2$, make v the parent of all but the two ranked children of u of highest ranks; optionally, make v the parent of some or all of the unranked children of u. Do not make u a child of v.

In an eager hollow heap, each node has at most one parent. Thus each heap is represented by a tree, accessed via its root. The analysis of eager hollow heaps differs from that of lazy hollow heaps only in using parents instead of virtual parents. Only the parents of ranked children matter in the analysis.

The proofs of the following results are essentially identical to the proofs of the results in Section 2, with the word "virtual" deleted.

Lemma 3. *Let u be a node of rank r in an eager hollow heap. If u is full, or u is a node made hollow by a delete, u has exactly one ranked child of each rank from 0 to $r-1$ inclusive, and none of rank r or greater. If u was made hollow*

by a decrease-key and $r > 1$, u has exactly two ranked children, of ranks $r-1$ and $r-2$. If u was made hollow by a decrease-key and $r = 1$, u has exactly one ranked child, of rank 0. If u was made hollow by a decrease-key and $r = 0$, u has no ranked children.

Corollary 3. *A node of rank r in an eager hollow heap has at least $F_{r+3} - 1$ descendants.*

Theorem 4. *The maximum rank of a node in an eager hollow heap of N nodes is at most $\log_\phi N$.*

Theorem 5. *The amortized time per eager hollow heap operation is $O(1)$ for each operation other than a delete, and $O(\log N)$ per delete on an N-node heap.*

An alternative way to think about eager hollow heaps is as a variant of Fibonacci heaps. In a Fibonacci heap, the cascading cuts that occur during a *decrease-key* prune the tree in a way that guarantees that ranks remain logarithmic in subtree sizes. Eager hollow heaps guarantee logarithmic ranks by leaving (at least) two children and a hollow node behind at the site of the cut. This avoids the need for cascading cuts or rank changes, and makes the *decrease-key* operation $O(1)$ time in the worst case.

5 Rebuilding

The number of nodes N in a heap is at most the number of items n plus the number of *decrease-key* operations on items that were ever in the heap or in heaps melded into it. If the number of *decrease-key* operations is polynomial in the number of insertions, $\log N = O(\log n)$, so the amortized time per *delete* is $O(\log n)$, the same as for Fibonacci heaps. In applications in which the storage required for the problem input is at least linear in the number of heap operations, the extra space needed for hollow nodes is linear in the problem size. Both of these conditions hold for the heaps used in many graph algorithms, including Dijkstra's shortest path algorithm [4,8], various minimum spanning tree algorithms [4,8,9,17], and Edmonds' optimum branching algorithm [6,9]. In these applications there is at most one *insert* per vertex and one or two *decrease-key* operations per edge or arc, and the number of edges or arcs is at most quadratic in the number of vertices. In such applications hollow heaps are asymptotically as efficient as Fibonacci heaps.

For applications in which the number of *decrease-key* operations is huge compared to the heap sizes, we can use periodic rebuilding to guarantee that $N = O(n)$ for every heap. To do this, keep track of N and n for every heap. When $N > cn$ for a suitable constant $c > 1$, rebuild. We offer two ways to do the rebuilding. The first is to completely disassemble the dag and reinsert all its items into a new, initially empty heap. A second method that does no key comparisons is to convert the dag into a tree containing only full nodes, as follows: For each node that has two parents, eliminate the second parent, making the dag a tree. Give each full child a rank of 0 and a parent equal to its nearest full proper

ancestor. Delete all the hollow nodes. To extend the analysis in Sections 3 and 4 to cover the second rebuilding method, we define every child to be unranked after rebuilding. Either way of rebuilding can be done in a single traversal of the dag, taking $O(N)$ time. Since $N > cn$ and $c > 1$, $O(N) = O(N - n)$. That is, the rebuilding time is $O(1)$ per hollow node. By charging the rebuilding time to the *decrease-key* and *delete* operations that created the hollow nodes, $O(1)$ per operation, we obtain the following theorem:

Theorem 6. *With rebuilding, the amortized time per hollow heap operation is* $O(1)$ *for each operation other than a delete-min or delete, and* $O(\log n)$ *per delete-min or delete on a heap of* n *items. These bounds hold for both lazy and eager hollow heaps.*

By making c sufficiently large, we can arbitrarily reduce the rebuilding overhead, at a constant factor cost in space and an additive constant cost in the amortized time of *delete*. Whether rebuilding is actually a good idea in any particular application is a question to be answered by experiments.

6 Implementation of Hollow Heaps

In this section we develop an implementation of the data structure in Section 2 that satisfies the time bounds in Section 3 and that is tuned to save space. We store each set of children in a list. Each new child of a node v is added to the front of the list of children of v. Since hollow nodes can be in two lists of children, it might seem that we need to make the lists of children exogenous. But we can make them endogenous by observing that only hollow nodes can have two parents, and a hollow node with two parents is last on the list of children of its second parent (since it is the earliest child, and later children are added to the front of the list). This allows us to use two pointers per node u to represent lists of children: $u.child$ is the first child of u, *null* if u has no children; $u.next$ is the next sibling of u on the list of children of its first parent.

With this representation, given a child u of a node v, we need ways to answer three questions: (i) Is u last on the list of children of v? (ii) Does u have two parents? (iii) Is v the first or the second parent of u? If u has only one parent, the first question is easy to answer: u is the last child of v if and only if $u.next = null$. There are several ways to answer the second two questions in $O(1)$ time. We develop a detailed implementation using one method, and we discuss alternatives in the full version of the paper.

Each node u stores a pointer $u.item$ to the item it holds if it is full; if u is hollow, $u.item = null$. Each hollow node u stores a pointer to its second parent $u.sp$; if u is hollow but has at most one parent, $u.sp = null$. A *decrease-key* operation makes a newly hollow node u a child of a new node v by setting $v.child = u$ but not changing $u.next$: $u.next$ is the next sibling of u on the list of children of the first parent of u. We answer the three questions as follows: (i) A child u of v is last on the list of children of v if and only if $u.next = null$ (u is last on any list of children containing it) or $u.sp = v$ (u is hollow with two parents

and v is its second parent); (ii) u has two parents if and only if $u.sp \neq null$; (iii) v is the second parent of u if and only if $u.sp = v$.

Each node u also stores its key and rank, and each item e stores the node $e.node$ holding it. The total space needed is four pointers, a key and a rank per node, and one pointer per item. Ranks are small integers, requiring $\lg \lg N + O(1)$ bits each.

Implementation of *delete* requires keeping track of roots as they are deleted and linked. To do this, we maintain a list L of hollow roots, singly linked by *next* pointers. We also maintain an array A of full roots, indexed by rank, at most one per rank. When a *delete* makes a root hollow, do the following. First, initialize L to contain the hollow root and A to be empty. Second, repeat the following until L is empty: Delete a node x from L, apply the appropriate one of the following cases to each child u of x, and then destroy x:

(i) u is hollow and v is its only parent: Add u to L: deletion of x makes u a root.
(ii) u has two parents and v is the second: Set $u.sp = null$ and stop processing children of x: u is the last child of x. Since u still has its first parent, it does not become a root.
(iii) u has two parents and v is the first: Set $u.sp = null$ and $u.next = null$.
(iv) u is full: Add u to A unless A contains a root of the same rank. If it does, link u with this root via a ranked link and repeat this with the winner until A does not contain a root of the same rank; then add the final winner to A.

Third and finally (once L is empty), empty A and link full roots via unranked links until there is at most one.

With this implementation, the worst-case time per operation is $O(1)$ except for *delete* operations that remove root items. A *delete* that removes a root item takes $O(1)$ time plus $O(1)$ time per hollow node that loses a parent plus $O(1)$ time per link plus $O(\log_\phi N)$ time, where N is the number of nodes in the tree just before the *delete*, since $max\text{-}rank = O(\log_\phi N)$ by Theorem 2. These are the bounds claimed in Section 3.

Acknowledgement. Thomas Dueholm Hansen is supported by The Danish Council for Independent Research | Natural Sciences (grant no. 12-126512); and the Sino-Danish Center for the Theory of Interactive Computation, funded by the Danish National Research Foundation and the National Science Foundation of China (under the grant 61061130540). Haim Kaplan is supported by the Israel Science Foundation grants no. 822-10 and 1841/14, the German-Israeli Foundation for Scientific Research and Development (GIF) grant no. 1161/2011, and the Israeli Centers of Research Excellence (I-CORE) program (Center No. 4/11). Uri Zwick is supported by BSF grant no. 2012338 and by The Israeli Centers of Research Excellence (I-CORE) program (Center No. 4/11).

References

1. Brodal, G.S.: Worst-case efficient priority queues. In: Proceedings of the 7th ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 52–58 (1996)

2. Brodal, G.S., Lagogiannis, G., Tarjan, R.E.: Strict Fibonacci heaps. In: Proc. of the 44th ACM STOC, pp. 1177–1184 (2012)
3. Chan, T.M.: Quake heaps: a simple alternative to fibonacci heaps. In: Brodnik, A., López-Ortiz, A., Raman, V., Viola, A. (eds.) Space-Efficient Data Structures, Streams, and Algorithms. LNCS, vol. 8066, pp. 27–32. Springer, Heidelberg (2013)
4. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik **1**, 269–271 (1959)
5. Driscoll, J.R., Gabow, H.N., Shrairman, R., Tarjan, R.E.: Relaxed heaps: an alternative to Fibonacci heaps with applications to parallel computation. Communications of the ACM **31**(11), 1343–1354 (1988)
6. Edmonds, J.: Optimum branchings. J. Res. Nat. Bur. Standards **71B**, 233–240 (1967)
7. Elmasry, A.: The violation heap: a relaxed Fibonacci-like heap. Discrete Math., Alg. and Appl., **2**(4), 493–504 (2010)
8. Fredman, M.L., Tarjan, R.E.: Fibonacci heaps and their uses in improved network optimization algorithms. Journal of the ACM **34**(3), 596–615 (1987)
9. Gabow, H.N., Galil, Z., Spencer, T.H., Tarjan, R.E.: Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. Combinatorica **6**, 109–122 (1986)
10. Haeupler, B., Sen, S., Tarjan, R.E.: Rank-pairing heaps. SIAM Journal on Computing **40**(6), 1463–1485 (2011)
11. Høyer, P.: A general technique for implementation of efficient priority queues. In: Proceedings of the 3rd Israeli Symposium on the Theory of Computing and Systems (ISTCS), pp. 57–66 (1995)
12. Kaplan, H., Shafrir, N., Tarjan, R.E.: Meldable heaps and boolean union-find. In: Proc. of the 34th ACM STOC, pp. 573–582 (2002)
13. Kaplan, H., Tarjan, R.E.: Thin heaps, thick heaps. ACM Transactions on Algorithms **4**(1), 1–14 (2008)
14. Kaplan, H., Tarjan, R.E., Zwick, U.: Fibonacci heaps revisited. CoRR, abs/1407.5750 (2014)
15. Knuth, D.E.: Sorting and searching. The art of computer programming, vol. 3, 2nd edn. Addison-Wesley (1998)
16. Peterson, G.L.: A balanced tree scheme for meldable heaps with updates. Technical Report GIT-ICS-87-23, School of Informatics and Computer Science, Georgia Institute of Technology, Atlanta, GA (1987)
17. Prim, R.C.: Shortest connection networks and some generalizations. Bell System Technical Journal **36**, 1389–1401 (1957)
18. Takaoka, T.: Theory of 2–3 heaps. Discrete Appl. Math. **126**(1), 115–128 (2003)
19. Tarjan, R.E.: Data structures and network algorithms. SIAM (1983)