Hollow Heaps [*]

Thomas Dueholm Hansen[†]   Haim Kaplan[‡]   Robert E. Tarjan[§]   Uri Zwick[¶]

October 23, 2015

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 on a heap of n items. 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.

Subject classification: 68P05 Data structures; 68Q25 Analysis of algorithms

Keywords: data structures, priority queues, heaps, amortized analysis

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.

$decrease\text{-}key(e, k, h)$: 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 [15].)

Fredman and Tarjan [10] 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 [5, 10] and fast algorithms for undirected and directed minimum spanning trees [7, 11]. Since the invention of Fibonacci heaps, a number of other heap implementations with the same amortized time bounds have been proposed [1, 2, 4, 8, 12, 13, 16, 19, 21]. 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 [12]. 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 final 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 non-trivial. 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., [17]). 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.

Starting from the ideas used in Fibonacci heaps, we develop hollow heaps in three steps. First, we show how *hollow nodes*, with an appropriate way of moving children and setting ranks, can replace the cascading cut process that makes Fibonacci heaps efficient. Second, we replace the set of trees representing a heap by a single tree, by allowing *unranked links* (links between roots of different ranks). This idea was used before in [12, 17] and is orthogonal to the idea of using hollow nodes. Third, we obtain our final data structure by showing how to avoid moving children of hollow nodes at all. To do this, we represent a heap by a (tree-like) dag rather than a tree: each node can have up to two parents, rather than just one.

The remainder of our paper consists of 6 sections. Section 2 presents a multi-root version of hollow heaps, the first of the three steps mentioned above. Section 3 presents a one-root version of hollow heaps. Section 4 presents the final version of our data structure, which replaces the movement of children by the use 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 4. Section 7 explores the design space of hollow heaps, identifying variants that are efficient and variants that are not.

2 Multi-root hollow heaps

As in the classic Fibonacci heap data structure, we represent each heap by a set of heap-ordered trees. In a Fibonacci heap, each node stores exactly one item, and each item is in exactly one node. We relax this invariant to allow nodes that do not hold items. We call a node *full* if it holds an item and *hollow* if not. A newly created node is initially full, but can later become hollow by having its item moved to a new node or deleted. A hollow node stays hollow until it is destroyed. Since items are moved among nodes, the structure is necessarily *exogenous* rather than *endogenous* [22]: nodes *hold* items rather than *being* items.

If u is a node, we denote by *u.item* the item held by u if u is full, or *null* if u is hollow. If e is an item, we denote by *e.node* the node holding e. Each node u has a key *u.key* associated with it: [1] if u is full, *u.key* is the current key of *u.item*; if u is hollow, *u.key* is the key of the item once held by u, just before it was moved from u.

We organize nodes into rooted trees. If (v, w) is a tree arc, we say that v is the *parent* of w and w is a *child* of v. (We direct tree arcs from parent to child.) If there is a path of zero or more arcs from v to w in a tree, we say that v is an *ancestor* of w and w is a *descendant* of v. A node with no parent is a *root*; a node with no children is a *leaf*. A tree is *heap-ordered* if and only if for every arc (v, w), $v.key \leq w.key$ (whether or not v and w are hollow). Heap-order implies that the root has a minimum key.

We maintain the invariant that the set of trees representing a heap is either empty or contains a full root whose key is minimum among those of all nodes in the trees, full and hollow. We maintain a pointer to such a root. We call the node indicated by this pointer the *minimum node* of the heap.

A generic way of implementing the heap update operations is via the *link* primitive. Given two full roots u and v, *link(u, v)* compares their keys and makes the root of larger key a child of the other, breaking a tie arbitrarily. 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. As in Fibonacci heaps, we do links only during *delete-min* or *delete* operations that delete the item in the minimum node.

To do *make-heap*, return an empty forest. To do *find-min*, return the item in the minimum node. To *meld* two heaps, if one is empty return the other; otherwise, unite their sets of trees and update the minimum node. 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.

The *decrease-key* operation uses a lazy form of deletion. To decrease the key of item e in heap h to k, let $u = e.node$ and do the following: create a new node v; move e from u to v (making v full and u hollow); set $v.key = k$; move some or all of the children of u, and their subtrees, to v; *meld* the one-root heap consisting of v and its descendants with the existing heap. The choice of which children to move is a critical design decision that we address shortly. As an optimization, if u is a

---
[1]Alternatively, one can store keys with items rather than with nodes. Which alternative is preferable is primarily an experimental question.

root one can avoid creating a new node by merely setting $u.key = k$ and updating the minimum node.

To do *delete-min*, delete the item in the minimum node. To delete an item e, remove e from the node $u = e.node$ holding it, making u hollow. This completes the deletion unless u is the minimum node. After deleting the item in the minimum node during either a *delete-min* or *delete*, proceed as follows: while there is a hollow root, destroy such a root, making each of its children into a root (unite the set of such new roots with the set of old roots). Once all roots are full, do zero or more links to reduce the number of roots. Make each remaining tree into a heap, and meld these heaps in any order. The choice of which links to do is another critical design decision that we address shortly.

Remark: This implementation of *delete* allows hollow roots. One can keep all roots full by proceeding as in the case of deletion of the item in the minimum node whenever an item in *any* root is deleted.

As in Fibonacci heaps, we make the data structure efficient by using *node ranks*. [2] We give each node u a non-negative integer rank $u.rank$. When a node is created by an insertion, its initial rank is 0. We do links only between roots of equal rank. Such a link increases the rank of the winner by one. We call such a link a *ranked link*. A *delete* that deletes the item in the minimum node does ranked links until none are possible (all roots have different ranks).

The remaining design decision is the choice of which children (and their subtrees) to move during a *decrease-key* operation (and what rank to give the newly created node). We achieve efficiency by maintaining the following *rank invariant*:

> A node u of rank r has exactly r children, of ranks $0, 1, \ldots, r - 1$, *unless* $r > 2$ and u was made hollow by a *decrease-key*, in which case u has exactly two children, of ranks $r - 2$ and $r - 1$.

To maintain the rank invariant, when doing a *decrease-key* operation that moves an item from node u to node v, initialize $v.rank = \max\{0, u.rank - 2\}$, and move to v each child of u of rank less than $v.rank$ (along with their subtrees). If $u.rank \geq 2$, u retains its children of ranks $u.rank - 2$ and $u.rank - 1$, along with their subtrees; if $u.rank = 1$, u retains its only child (of rank 0).

The resulting data structure is the *multi-root hollow heap*. Figure 1 illustrates operations on a multi-root hollow heap.

Theorem 2.1 *Multi-root hollow heaps correctly implement all the heap operations, maintain heap-ordered trees, and maintain the rank invariant.*

Proof: The proof is straightforward by induction on the number of heap operations. $\square$

The effect of our design decisions is to keep the trees *balanced* in an appropriate sense, as we show by an argument like that for Fibonacci heaps but simpler and more direct. 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* [18].

Lemma 2.2 *A node of rank r has at least $F_{r+3} - 1$ descendants, both full and hollow.*

---

[2] Fredman [9] has shown that obtaining a constant amortized bound for *decrease-key* in a data structure like ours *requires* storing $\Omega(n \log \log n)$ extra bits of information, such as node ranks, although his result has significant technical restrictions. See also Iacono and Özkan [14].
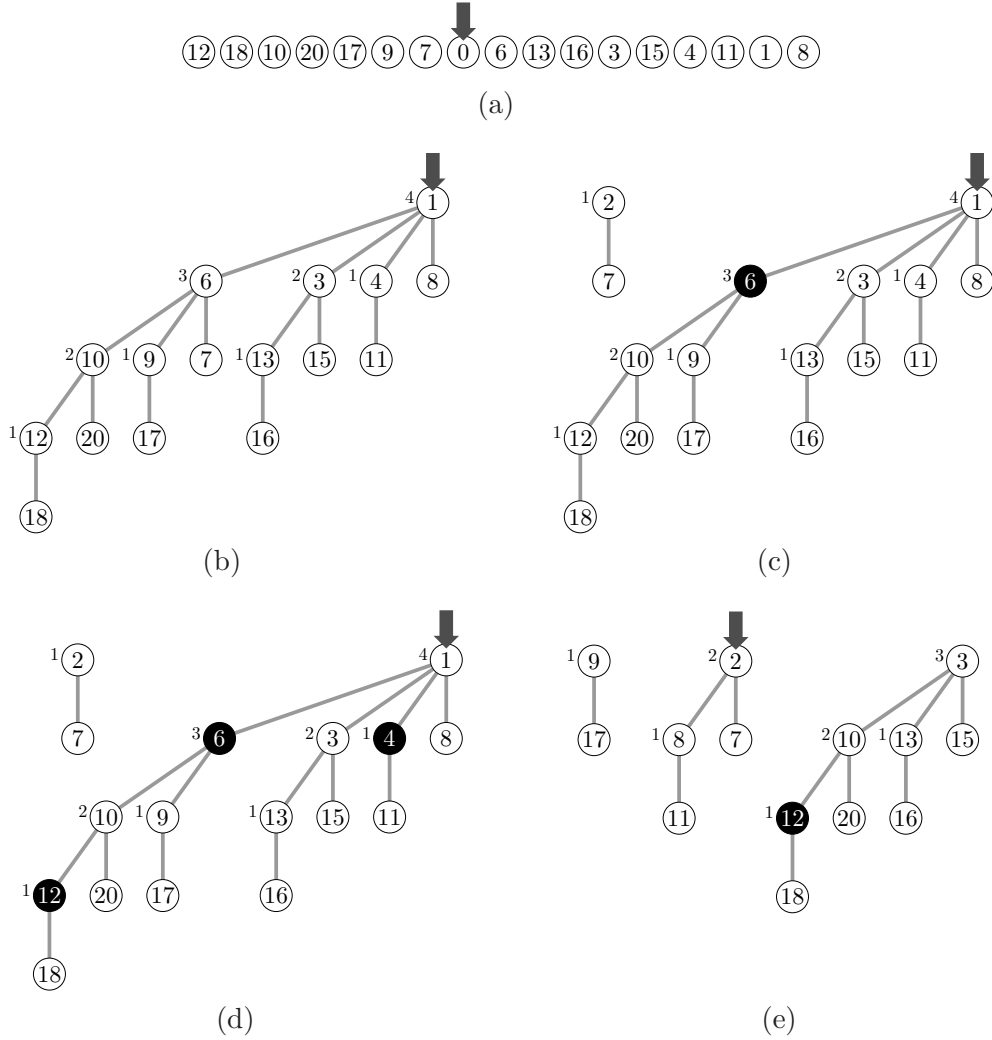
Figure 1: Operations on a multi-root hollow heap. Numbers in nodes are keys; black nodes are hollow. Numbers next to nodes are non-zero ranks. (a) The heap after the successive insertion of items with keys, 12, 18, 20, 20,.... Each item forms a singleton tree. The arrow pointing to the item with key 0 is the minimum pointer. (b) The heap after the deletion of the the minimum item, i.e., the item with key 0. (c) The heap after decreasing key 6 to 2. (d) The heap after deleting the items with keys 4 and 12. (e) The heap after a *delete-min* operation that deletes the item with key 1.

Proof: The proof is by induction on r. The claim is immediate for $r = 0$ and $r = 1$. If $r \geq 2$, the descendants of a node u of rank r include itself and all descendants of its children of ranks $r - 1$ and $r - 2$, which it has by the rank invariant, whether it is full or hollow. By the induction hypothesis, u has at least $1 + (F_{r+2} - 1) + (F_{r+1} - 1) = F_{r+3} - 1$ descendants. $\qquad \square$

Corollary 2.3 *The rank of a node in a multi-root hollow heap of N nodes is at most $\log_\phi N$.*

Proof: The corollary is immediate from Lemma 2.2 since $F_{r+3} - 1 \geq F_{r+2} \geq \phi^r$ for $r \geq 0$. $\qquad \square$

To obtain an efficient implementation of multi-root hollow heaps, we store the children of each node in a singly-linked list in decreasing order by rank. We store the roots of each heap in a singly-linked

circular list accessed via the minimum node. Circular linking allows lists of roots to be catenated in constant time, allowing *meld* to be done in constant time. When doing a ranked link, we make the new child the first child of its new parent; this keeps the children in decreasing rank order. When doing a *decrease-key* that moves an item from a node u to a node v, if $u.rank > 2$ we move to v all but the first two children of u; if $u.rank \leq 2$, we move no children. Each heap operation except *delete* takes constant time worst-case. To find roots to link during a delete, we use an array of roots indexed by rank, as in Fibonacci heaps [10]. (A single global array may be used for all heaps.) The time for a *delete* is $O(1)$ unless it deletes the item in the minimum node, in which case it takes $O(H + T)$ time, where H is the number of hollow roots destroyed and T is the number of trees remaining after destruction of hollow roots but before any links. After the links there is at most one tree per rank, totaling at most $\log_\phi N$ by Corollary 2.3. Thus the number of links is at least $T - \log_\phi N$, and the number of melds is at most $\log_\phi N$. It follows that the time for the *delete* is $O(1)$ per hollow root destroyed plus $O(1)$ per *link* plus $O(\log N)$.

We bound the amortized time per operation by modifying the analysis of Fibonacci heaps.

Theorem 2.4 *The amortized time per multi-root hollow heap operation is $O(1)$ for each operation other than a delete or delete-min, and $O(\log N)$ per delete or delete-min on a heap of N nodes.*

Proof: The worst-case time per operation is $O(1)$ except for a *delete* that deletes the item in the minimum node, which takes time $O(1)$ per hollow node destroyed plus $O(1)$ per *link* plus $O(\log N)$. We charge node destructions against the corresponding node creations, one per *insert* and one per *decrease-key*. To obtain the theorem, it remains to bound the number of links.

To count links use a *potential* function [23]. We give each full node that is not a child of another full node a potential of one, and all other nodes (hollow nodes and full children of full nodes) a potential of zero. The potential of a heap is the sum of the potentials of its nodes. We define the *amortized cost* of an operation to be the number of links plus the increase in potential. Since the initial potential is zero (the data structure is empty) and the potential is always non-negative, the total number of links is at most the sum of the amortized costs of all the operations.

All the links are in *delete* operations that delete the item in the minimum node, so the amortized cost of every other operation is the potential change. This is one per *insert* (one new root), zero per *find-min* or *meld*, and at most three per *decrease-key* (one new root and at most two full children of a new hollow node). In a *delete*, each *link* has an amortized cost of zero: it converts a full root into a full child of a full root, reducing the potential of the new child by one. The only other potential change in a *delete* is caused by removing an item. This creates a hollow node u, increasing the potential by one per full child of u, at most $\log_\phi N$ in total. We conclude that the number of links is $O(1)$ per heap operation plus $\log_\phi N$ per *delete* on a heap of N nodes. □

Remark: The proof of Theorem 2.4 also gives a bound on the amortized number of comparisons per operation: at most one per *insert* and *meld*, three per *decrease-key*, and $2 \log_\phi N$ per *delete* on a heap of N nodes.

We have obtained multi-root hollow heaps from Fibonacci heaps by making a very simple change: when doing a *decrease-key* operation at a node u, instead of moving the entire subtree rooted at u, we leave u and up to two of its children (and their subtrees) in place. This preserves enough of the tree balance to avoid doing cascading cuts as in Fibonacci heaps, or cascading rank changes as in [12, 17], or restructuring to reduce heap-order violations as in [2, 6, 16]. As a result, *decrease-key* operations take $O(1)$ time worst-case as well as amortized, and no additional state information, such as mark bits, is needed. The Fibonacci numbers enter the analysis (Lemma 2.2) more directly, and the amortized analysis (Theorem 2.4) becomes simpler. The implementation is also simpler:

singly-linked lists replace the doubly-linked lists needed in Fibonacci heaps, and no parent pointers are needed.

The price we pay for these changes is threefold: the data structure becomes exogenous rather than endogenous (items point to nodes and vice versa, rather than items being nodes); the amortized time of *delete* and *delete-min* becomes $O(\log N)$ rather than $O(\log n)$, and the space used by the data structure is $O(N)$ rather than $O(n)$, where n is the number of items in the heap and N is the number of nodes. The latter is at most the number of *insert* and *decrease-key* operations. We can reduce the time and space bounds to $O(\log n)$ and $O(n)$ by periodically rebuilding the data structure, as we discuss in Section 5.

Multi-root hollow heaps are asymptotically optimal, but they do not use all available key order information. In particular, they do not keep track of all key comparisons done when updating minimum nodes during melds, nor do they keep track of the decreasing key order of nodes successively holding the same item. In the next two sections we modify the data structure to use this additional information.

3   One-root hollow heaps

By allowing links between roots of different ranks, we can obtain a one-root version of hollow heaps. This idea is quite general and is orthogonal to the idea of hollow nodes: it was used in [12] to obtain a one-root version of rank-pairing heaps and in [17] to obtain a one-root version of Fibonacci heaps. Here we apply it to hollow heaps. In the next section, we modify hollow heaps more drastically, to avoid moving children during *decrease-key* operations.

As described in Section 2, a *ranked link* applies to two full roots of equal rank. It makes the node of larger key a child of the node of smaller key (the winner) and increases the rank of the winner by one, breaking a tie arbitrarily. In contrast, an *unranked link* applies to any two full roots, whether or not their ranks are equal. It makes the node of larger key a child of the node of smaller key and changes no ranks, breaking a tie arbitrarily. In either kind of link, the new child is the loser and its parent is the winner. The loser of a link is a *ranked* or *unranked* child if the link is ranked or unranked, respectively. A child retains its ranked or unranked state when moved to a new parent by a *decrease-key* operation; it can only change state by losing another link, which can only happen after its parent becomes hollow and is later destroyed.

A *one-root hollow heap* is either empty or is a single heap-ordered tree whose root is full. The root is the minimum node. We do the heap operations as in multi-root hollow heaps, with the following changes. To *meld* two non-empty heaps, do an unranked link of their roots. In a *decrease-key* that moves an item from a node u to a new node v, initialize $v.rank = \max\{0, u.rank - 2\}$, and move to v all *ranked* children of u of ranks less than $v.rank$ (along with their subtrees), and any or all of the unranked children of u (along with their subtrees).

The rank invariant for one-root hollow heaps is the following:

> A node u of rank r has exactly r ranked children, of ranks $0, 1, \ldots, r - 1$, unless $r > 2$ and u was made hollow by a *decrease-key*, in which case u has exactly two ranked children, of ranks $r - 2$ and $r - 1$. (A node can have any number of unranked children.)

Figure 2 illustrates operations on a one-root hollow heap.

Theorem 3.1 *One-root hollow heaps correctly implement all the heap operations, maintain heap-ordered trees, and maintain the rank invariant.*
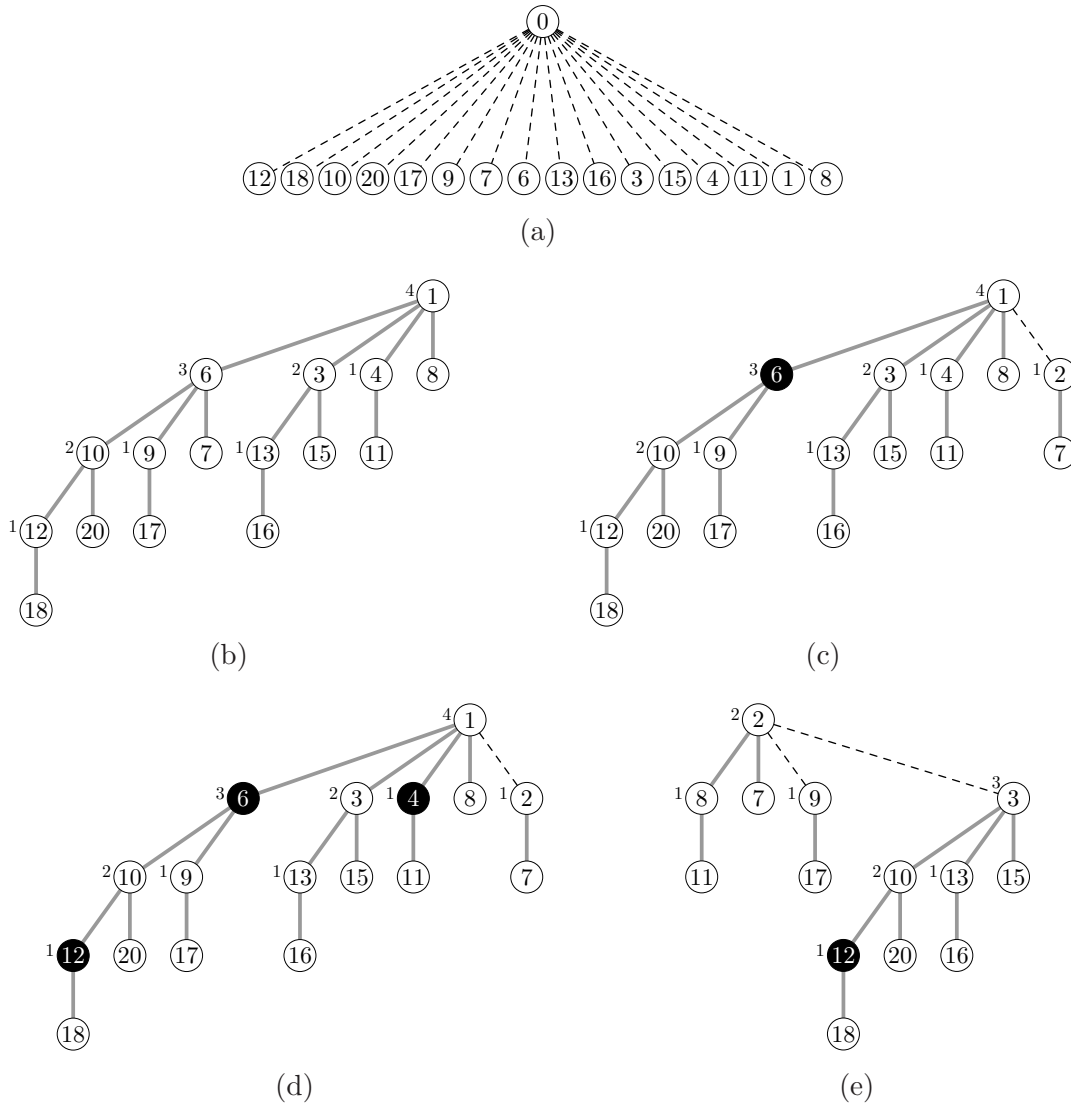
Figure 2: Operations on a one-root hollow heap. Numbers in nodes are keys; black nodes are hollow. Numbers next to nodes are non-zero ranks. Bold gray and dashed lines denote ranked and unranked links, respectively. (a) The heap after the successive insertion of items with keys $0, 12, 18, \ldots$. All links performed are unranked. (a) The heap after the deletion of the minimum item. All links performed are ranked. (c) The heap after decreasing key 6 to 2. (d) The heap after deleting the items with keys 4 and 12. (e) The heap after another *delete-min* operation.

Proof: The proof is straightforward by induction on the number of heap operations. □

Lemma 3.2 *A node of rank r in a one-root hollow heap has at least $F_{r+3} - 1$ descendants, both full and hollow.*

Proof: The proof is the same as that of Lemma 2.2. □

Corollary 3.3 *The rank of a node in a one-root hollow heap of N nodes is at most $\log_\phi N$.*

8

Proof: The proof is the same as that of Corollary 2.3, using Lemma 3.2. □

To obtain an efficient implementation of one-root hollow heaps, we store the children of each node in a singly-linked circular list, accessed via the *last* child on the list. The ranked children are first, in decreasing order by rank, followed by the unranked children. Making each list of children circular and accessing such a list via its last node allows doing links in constant time while maintaining the desired order of children: when doing a ranked link, add the new child to the *front* of the list of its new parent, when doing an unranked link, add it to the *back*. We do *not* need to store the states of children (ranked or unranked), only the node ranks. When doing a *decrease-key* that moves an item from a node u to a new node v, if $u.rank > 2$ we move all but the first two children of u to v, along with their subtrees. This leaves with v the two ranked children of highest rank, and moves all the unranked children. (The implementation is free to move none, some, or all of the unranked children.)

Theorem 3.4 *The amortized time per one-root hollow heap operation is $O(1)$ for each operation other than a delete or delete-min, and $O(\log N)$ per delete or delete-min on a heap of N nodes.*

Proof: The proof is like the proof of Theorem 2.4 but with a slightly different potential function: each full node has one unit of potential unless it is a full ranked child of a full node, in which case it has zero potential. A ranked link reduces the potential by one, giving it an amortized cost of zero. An unranked link does not change the potential giving it an amortized cost of one. The amortized cost of an operation other than a delete or delete-min is $O(1)$ by the same argument as in the proof of Theorem 2.4. In a *delete* or *delete-min*, making a node hollow by deleting its item increases the potential by $O(\log N)$, since only ranked children increase in potential. If the deletion is of the item in the minimum node, the ranked links have amortized cost zero, and there are $O(\log N)$ unranked links, at most one per rank. It follows that the amortized cost of a *delete* or *delete-min* is $O(\log N)$.
□

We conclude this section by discussing two options in doing links. First, at the end of a delete that removes the item in the minimum node, we are free to do the melds in any order. Repeatedly melding two trees of minimum rank is natural and easy to implement, but we do not have theory to justify this order.

Second, we can in certain cases increase the rank of the winner in an unranked link. In particular, in an unranked link of two roots of equal rank, we can increase the rank of the winner by one, making the link into a ranked link; in an unranked link of two roots of unequal rank, if the root of smaller rank is the winner, we can increase its rank to any value no greater than the rank of the loser. To allow increases of the second type, we need to change the algorithm slightly. We define any link that increases the rank of the winner to be a ranked link, and any other link to be an unranked link. As in the original implementation, the loser of a ranked or unranked link becomes the first or last child of its new parent, respectively. When doing a *decrease-key* that moves an item from a node u to a new node v, if u has at most two children we merely set the rank of v to zero; if u has more than two children we leave the first two of them with u, move the rest to v (preserving their order), and set the rank of v equal to that of its new first child. This method maintains the following invariant:

> A node of rank $r > 0$ either has a first child of rank at least r, or it has a first child of rank $r - 1$ and a second child of rank at least $r - 2$.

An extension of our analysis shows that this method is correct and efficient. With this method it is possible for a node to accumulate many more ranked children than its rank. To handle this in the

9

analysis, we give each full node additional potential equal to the maximum of zero and its number of ranked children minus its rank. We omit the details of the analysis, since they are so similar to what we have already presented.

Whether either or both of these options improves performance is an experimental question; they certainly complicate the algorithm.

4   Two-parent hollow heaps

Our second and more drastic change to hollow heaps eliminates the movement of children during *decrease-key* operations. This change comes at the cost of converting the data structure from a tree or set of trees to a (treelike) *dag* (directed acyclic graph). During a *decrease-key* that moves an item from a node u to a new node v (still of initial rank $\max\{0, u.rank - 2\}$), instead of moving certain children of u to v, we merely make u a child of v. This gives u a *second* parent. We make this change to the one-root hollow heaps of Section 3, but it applies equally to the multi-root hollow heaps of Section 2. We call the resulting data structure the *two-parent hollow heap*, in contrast to the data structures of Sections 2 and 3, which we jointly call *one-parent hollow heaps*.

To present the details of this idea, we extend our tree terminology to dags. If (v, w) is a dag arc, we say that v is the *parent* of w and w is a *child* of v. A node with no parent is a *root*; a node with no children is a *leaf*. A dag whose nodes have keys is *heap-ordered* if and only if for every arc (v, w), $v.key \leq w.key$. That is: any topological order of the dag arranges the nodes in non-decreasing order by key.

A *two-parent hollow heap* is either empty or is a heap-ordered dag with one full root and no hollow roots. The root is the minimum node. We do the heap operations as in one-root hollow heaps, with the following change: in a *decrease-key* that moves an item from a node u to a new node v, make u a child of v; do not move any of the children of u. Figure 3 illustrates operations on a two-parent hollow heap.

Theorem 4.1 *Two-parent hollow heaps correctly implement all the heap operations and maintain each heap as a heap-ordered dag with one full root and no empty roots.*

Proof: The proof is straightforward by induction on the number of heap operations.   □

The analysis of two-parent hollow heaps is less straightforward than that of one-parent hollow heaps. To do the analysis, we define a way of virtually moving children among nodes that mimics their movement in one-parent hollow heaps. This definition is part of the analysis only; it is *not* part of the algorithm. First we prove a few properties of the algorithm.

Lemma 4.2 *A node in a two-parent hollow heap has at most one parent if it is full, at most two if it is hollow. Once a node is hollow it cannot acquire a new parent.*

Proof: There are only two ways for a node to acquire a parent: a full root can acquire a parent by losing a *link*, and a full node can acquire a parent by becoming hollow in a *decrease-key* operation. A root has no parents, so when a node loses a *link* it has only one parent. Once a node becomes hollow, it cannot become full again, so it cannot acquire a new parent.   □

We call nodes that hold the same item (at different times) *clones*.

Lemma 4.3 *At any given time, any maximal set of clones forms a path in the dag, in decreasing order by creation time. Each clone on the path, except possibly the first, is hollow. The path changes only by insertion or deletion at the front. Thus the path forms a stack.*
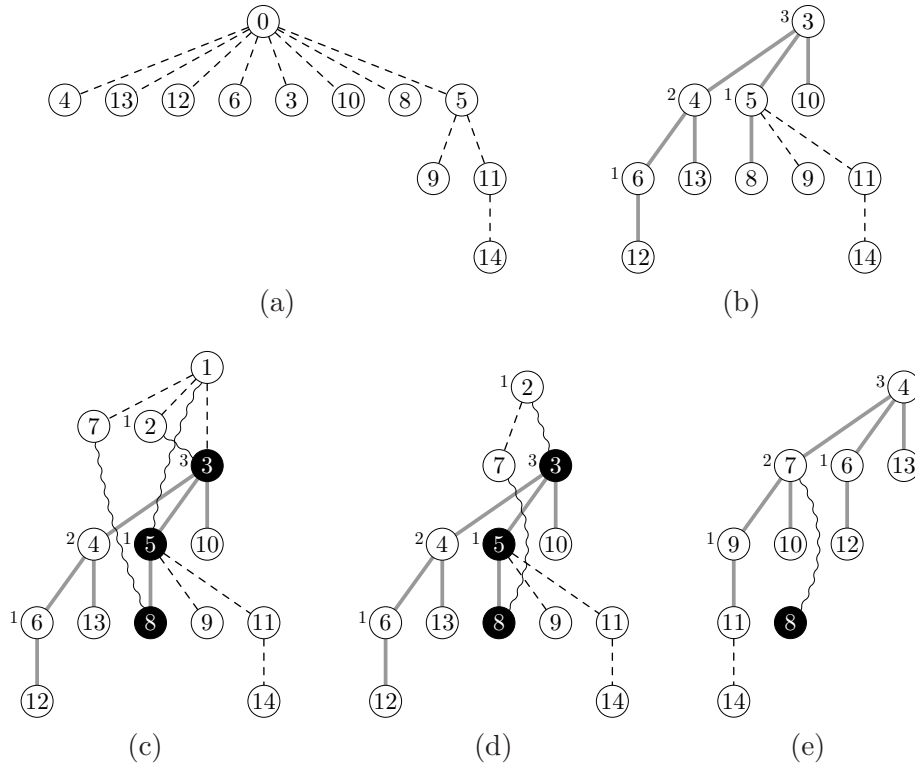
Figure 3: Operations on a two-parent hollow heap. Numbers in nodes are keys; black nodes are hollow. Bold gray, dashed, and squiggly lines denote ranked links, unranked links, and second parents, respectively. Numbers next to nodes are non-zero ranks. (a) Successive insertions of items with keys 14, 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 decreases of key 5 to 1, then key 3 to 2, and then key 8 to 7. (d) 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. (e) After a third *delete-min*. The hollow nodes with keys 3 and 5 become roots and are destroyed, the hollow root with key 8 loses one parent. All links are ranked.

Proof: The proof is by induction on the number of heap operations. Insertion creates a new path of clones consisting of one full node. A *decrease-key* that moves an item from a node u to a node v makes u hollow and adds v to the front of the path of clones of u. A *delete* or *delete-min* begins by removing an item from a node that is first on a path of its clones, making the node hollow. Subsequently, the path can change only by destruction of its nodes, from front to back. □

Now we are ready to describe the virtual movement of children and present a rank invariant for two-parent hollow heaps. We give nodes *virtual parents* as follows: When a node is created, it has no virtual parent. When a root with no virtual parent loses a ranked link, its new parent becomes its virtual parent. When the virtual parent of a node is destroyed, it loses its virtual parent. (If the node is full, it can acquire a new virtual parent later, by losing a link.) When a *decrease-key* moves an item from a node u to a node v, we make v the virtual parent of all the nodes whose virtual parent is u and whose rank is less than $v.rank = \max\{0, u.rank - 2\}$. A node is a *virtual child* of its virtual parent.

Lemma 4.4 *A root has no virtual parent.*

11

Proof: The proof is by induction on the number of heap operations. A newly created node has no virtual parent. Suppose a node w acquires a virtual parent by losing a *link* to u. Subsequently the virtual parent of w can change, but only to a later clone of u. For w to become a root again, u must be destroyed, but by Lemma 4.3 this can only happen after all later clones of u are destroyed, including the most recent virtual parent of w. Hence when w becomes a root again, it has no virtual parent. □

The rank invariant for two-parent hollow heaps is:

> A node u of rank r has exactly r virtual children, of ranks $0, 1, \ldots, r - 1$, unless $r > 2$ and u was made hollow by a *decrease-key*, in which case u has exactly two virtual children, of ranks $r - 2$ and $r - 1$.

Theorem 4.5 *Two-parent hollow heaps maintain the rank invariant.*

Proof: The proof is an induction on the number of heap operations like the proofs of the corresponding parts of Theorems 2.1 and 3.1, using Lemma 4.4: by the lemma, when a node acquires a child by a ranked link it becomes the virtual parent of that child. □

Since a node has at most one virtual parent at a time, the virtual parents define a set of rooted trees that we call *virtual trees*. A node w is a *virtual descendant* of a node v if $w = v$ or w is a virtual descendant of a virtual child of v.

Lemma 4.6 *A node of rank r in a two-parent hollow heap has at least $F_{r+3} - 1$ virtual descendants, both full and hollow.*

Proof: The proof is like that of Lemma 2.2. □

Corollary 4.7 *The rank of a node in a two-parent hollow heap of N nodes is at most $\log_\phi N$.*

Proof: The proof is the same as that of Corollary 2.3, using Lemma 4.6. □

A straightforward way to implement two-parent hollow heaps is to store each set of children in an exogenous singly-linked list. The order of children does not matter. A *link* makes the loser the first child of the winner. The lists must be exogenous rather than endogenous since nodes can be on two lists at the same time. We give each node a bit that is true if it has two parents. The bit is initially false. It becomes true when a *decrease-key* makes the node hollow, unless the node is currently the root. When a node is destroyed, each of its children with a false bit becomes a root; each of its children with a true bit has it set to false. In Section 6 we describe alternative implementations that use less space.

Theorem 4.8 *The amortized time per two-parent hollow heap operation is $O(1)$ for each operation other than a delete or delete-min, and $O(\log N)$ per delete or delete-min on a heap of N nodes.*

Proof: The proof is like the proof of Theorem 2.4 but with two changes. A hollow node can lose a parent without becoming a root and being destroyed (because it has another parent). We charge each such event to the *decrease-key* that gave the hollow node its second parent, at most one per *decrease-key*. To count ranked links, we use a slightly different potential function: each full node has one unit of potential unless it is a full *virtual* child of a full node. A ranked link reduces the potential by one, giving it an amortized cost of zero. The unranked links done in melds do not

change the potential. Making a node hollow by removing its item increases the potential by at most $\log_\phi N$, since only virtual children increase in potential. $\qquad\square$

We conclude this section by comparing two-parent and one-parent hollow heaps. Consider a *decrease-key* that moves an item from u to a new node v. Eventually one of u and v may be destroyed, but the one destroyed first depends on future heap operations, which are not known in advance. All children of the node destroyed become roots, and must themselves be destroyed (if hollow) or melded (if full). In one-parent hollow heaps, no matter how the children of u are distributed between u and v when the *decrease-key* occurs, in the worst case at least half of them will become roots when the first of u and v is destroyed. In a two-parent hollow heap, on the other hand, none of the children of u become roots until both u and v are destroyed. Thus not only do two-parent hollow heaps save work by not moving children, they delay the next access to these children by postponing when they become roots. We leave as an open question whether this local advantage translates into a global advantage.

5 Rebuilding

The number of nodes N in a hollow heap is at most the number of insertions 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, and only items in minimum nodes are deleted, then $\log N = O(\log n)$, where n is the number of heap items, so the amortized time per *delete* or *delete-min* 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 at most linear in the problem size. Both of these conditions hold for the heaps used in many graph algorithms, including Dijkstra's shortest path algorithm [5, 10], various minimum spanning tree algorithms [5, 10, 11, 20], and Edmonds' optimum branching algorithm [7, 11]. In these applications there is at most one *insert* and one *delete-min* per vertex, no *delete* operations other than those triggered by *delete-min* operations, and at most two *decrease-key* operations per edge or arc. Furthermore the number of edges or arcs is at most quadratic in the number of vertices. In these applications hollow heaps are asymptotically as time-efficient as Fibonacci heaps, although they need space linear in the number of edges or arcs, whereas Fibonacci heaps only need space linear in the number of vertices. Whether the space advantage of Fibonacci heaps is significant depends on the graph density and on implementation details.

For applications in which the number of *decrease-key* operations is huge compared to the number of insertions, or the extra space needed by hollow nodes becomes a bottleneck, 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 $c > 1$, eliminate all hollow nodes by rebuilding the heap.

We offer two ways to do the rebuilding. The first is to completely disassemble the heap and build a new one containing only the full nodes, as follows: Destroy every hollow node. Make each full node into a one-node heap whose node has rank 0. Do repeated melds until one heap remains. A second method that does no key comparisons is to give all the full nodes a rank of 0 and contract all the hollow nodes, as follows: In a two-parent hollow heap, eliminate one parent of every node that has two, making the dag a tree (or, in the multi-root case, a forest). Give each full node a rank of zero and give each full child a parent equal to its nearest full proper ancestor. Destroy all the hollow nodes. To extend the analysis in Sections 2-4 to cover the second rebuilding method, we define every child to be unranked after the rebuilding. Either way of rebuilding can be done in

a single traversal of the dag (or tree or set of trees), 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 destroyed. By charging the rebuilding time to the heap operations that created the hollow nodes, $O(1)$ per operation, we obtain the following theorem:

Theorem 5.1 *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. The space required by a heap is $O(n)$. These bounds hold for both two-parent and one-parent 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.

With rebuilding as described above, the worst-case time of a *decrease-key* operation is no longer $O(1)$, since such a *decrease-key* can trigger a rebuilding. (The amortized time remains $O(1)$.) One can preserve the $O(1)$ worst-case time of *decrease-key* operations by doing *incremental* rebuilding, although the details get a bit complicated. A simpler alternative is as follows: Use a multi-root version of hollow heaps (either one-parent or two-parent). During a *decrease-key*, if the item whose key decreases is in a root, decrease the key of this node and update the minimum node, instead of moving the item to a new node. Rebuild when a *delete* or *delete-min* causes $N > cn$, where $c > 1$. With this method, at most one hollow node per item is created between consecutive *delete* or *delete-min* operations. Thus if $N \le cn$ just after one such operation, $N \le (c+1)n$ just before the next one. By choosing c appropriately, one can guarantee that the fraction of nodes that are hollow is always at most $\frac{1}{2} - \varepsilon$, for any fixed $\varepsilon > 0$.

A natural question to ask is whether hollow nodes can be entirely eliminated from the data structure. The answer is yes, at least for one-parent hollow heaps. The idea is to "contract" hollow nodes as soon as they are created. We shall describe the application of this idea to the multi-root hollow heaps of Section 2, modified as described there to keep all roots full. In such a hollow heap, let u be a hollow child with full parent v. Let T be the maximal subtree of hollow nodes rooted at u, and let C be the list of (full) children of the leaves of T, in symmetric order (as defined by the order of nodes in lists of children). In the corresponding contracted structure, T is deleted and u is replaced as a child of v by list C, which becomes a sublist of children of v. For this idea to work, we need to keep track of such sublists of children, since it is these sublists, rather than individual children, that are moved during *decrease-key* operations.

Here are the (somewhat tedious) details. As in Section 2, maintain each list of children in decreasing order by rank, but partition each such list into sublists of consecutive children. The behavior of the algorithm defines these sublists. When a ranked link makes the loser u the first child of the winner v, u becomes the only member of a singleton sublist that is first among the sublists of children of v; the remaining sublists of children of v remain the same.

A *decrease-key* does not create a new node but merely changes the key of the old node holding the item and moves appropriate sublists of children. To do a *decrease-key* on the item in a node u, update the key of u. If u is a root, this completes the *decrease-key*. If not, let v be the parent of u and A the sublist of children of v containing u. Split A into two sublists A' and A'', with u first on A''. Sublist A' may be empty; A'' may contain only u. Delete u from A''. Form sublist B as follows: if $u.rank = 0$, let B be the empty list; if $u.rank = 1$, remove from u its first sublist of children and let this sublist be B; if $u.rank > 1$, remove from u its first two sublists of children and catenate them to form B. Set $v.rank = \max\{0, u.rank - 2\}$. Replace A in the list of sublists of children of v by the catenation of A', B, and A'' in this order. Add u (with its remaining sublists of children) to the list of roots, and update the minimum node.

We call this data structure the contracted hollow heap. The analysis of Section 2 extends to give the following theorem:

Theorem 5.2 *The amortized time per contracted 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 formed by N insert and decrease-key operations and any number of meld operations.*

It is straightforward to extend contraction to one-root one-parent hollow heaps, but we do not know how to extend it to two-parent hollow heaps in a nice way: contraction can cause a node to have an arbitrarily large number of parents.

Contracted hollow heaps contain no hollow nodes, but they have at least two significant drawbacks. The time per *delete* is $O(\log N)$, not $O(\log n)$: to obtain the latter, some form of additional rebuilding seems necessary. The number of pointers per node to implement the structure is large (exceeding that of Fibonacci heaps and rank-pairing heaps), although the structure can be made endogenous, saving some pointers. Thus we view contracted hollow heaps as primarily of theoretical interest.

6 Implementation of two-parent hollow heaps

As mentioned in Sections 2 and 3, one can implement one-parent hollow heaps (either multi-root or one-root) using a pointer per item (to the node holding it) and a rank and three pointers per node (to the first child, next sibling, and the item it holds). The implementation of two-parent hollow heaps uses more space, since each list of children must be exogenous rather than endogenous. In this section we describe implementations of two-parent hollow heaps that represent the lists of children endogenously and thereby reduce the space needed.

If v is a parent of u, we say that v is the *first* or *second parent* of u if u acquired parent v via a *link* or a *decrease-key*, respectively. Only a hollow node can have two parents; it can lose them in either order.

The way we have defined links to work in Section 4 is the key to making the lists of children endogenous. A link makes the loser the *first* child of the winner. This guarantees that a node u with two parents is always the *last* child of its second parent v: when u becomes a child of v, u is the only child of v, and any children that v later acquires are added in front of u on the list of children of v.

This allows us to use two pointers per node to represent lists of children, as in one-parent hollow heaps: if u is a node, $u.child$ is the first child of u, *null* if u has no children; if u is a child, $u.next$ is the next sibling of u on the list of children of its first parent, *null* if it is the last child 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) Assuming u has two parents, is v the first or the second? There are several methods that allow these questions to be answered in $O(1)$ time. We provide a detailed implementation of two-parent hollow heaps using one method and discuss alternatives below.

Each node u has a pointer $u.item$ to the item in u if u is full, *null* if u is hollow. Each node u has another pointer $u.ep$ (for extra parent) that is the second parent of u if u has two parents, *null* if u has at most one. In particular, $u.ep = null$ if u is full. As an optimization, a *decrease-key* on the item in the minimum node does not create a new node but merely changes the key of the minimum node. A decrease-key on an item not in the minimum node makes the newly hollow node u a child

15

of the new full node v by setting $v.child = u$ and $u.ep = v$ 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 $u.ep = v$ (u is hollow with two parents and v is its second parent); (ii) u has two parents if and only if $u.ep \neq null$; and (iii) Assuming u has two parents, v is the second if and only if $v = u.ep$.

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, each requiring $\lg \lg N + O(1)$ bits of space.

Implementation of all the heap operations except *delete* is straightforward. Figure 4 gives such implementations in pseudocode; Figure 5 gives implementations of auxiliary methods used in Figure 4.

```
make-heap():
    return null

insert(e, k, h):
    return meld(make-node(e, k), h)

meld(g, h):
{   if g = null: return h
    if h = null: return g
    return link(g, h)   }

find-min(h):
    if h = null: return null
    else: return h.item
```

```
decrease-key(e, k, h) :
{   u = e.node
    if u = h:
    {   u.key = k
        return h }
    v = make-node(e, k)
    u.item = null
    if u.rank > 2: v.rank = u.rank − 2
    v.child = u
    u.ep = v
    return link(v, h)   }

delete-min(h):
    return delete(h.item, h)
```

Figure 4: Implementations of all two-parent hollow heap operations except *delete*.

```
make-node(e, k):
{   u = new-node()
    u.item = e
    e.node = u
    u.child = null
    u.next = null
    u.ep = null
    u.key = k
    u.rank = 0
    return u   }
```

```
link(v, w):
    if v.key ≥ w.key:
    {   add-child(v, w)
        return w }
    else:
    {   add-child(w, v)
        return v }

add-child(v, w):
{   v.next = w.child
    w.child = v   }
```

Figure 5: Implementations of auxiliary methods used in Figure 4.

Implementation of *delete* requires keeping track of roots as they are destroyed 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.

16

```
delete(e, h):
{   e.node.item = null
    e.node = null
    if h.item ≠ null: return h          /* Non-minimum deletion */
    max-rank = 0
    while h ≠ null:              /* While L not empty */
    {   w = h.child
        v = h
        h = h.next
        while w ≠ null:
        {   u = w
            w = w.next
            if u.item = null:
            {   if u.ep = null:
                {   u.next = h
                    h = u  } }
                else:
                {   if u.ep = v: w = null
                    else: u.next = null
                    u.ep = null }
            else:
                do-ranked-links(u)
        destroy v  }
    do-unranked-links()
    return h }
```

Figure 6: Implementation of *delete* in two-parent hollow heaps. Rank updates during ranked links are done in the auxiliary method *do-ranked-links* in Figure 7.

```
do-ranked-links(u):                     do-unranked-links():
{   while A[u.rank] ≠ null:              for i = 0 to max-rank:
    {   u = link(u, A[u.rank])           {   if A[i] ≠ null:
        A[u.rank] = null                     {   if h = null: h = A[i]
        u.rank = u.rank + 1  }                   else: h = link(h, A[i])
    A[u.rank] = u                                A[i] = null  }  }
    if u.rank > max-rank:
        max-rank = u.rank  }  }
```

Figure 7: Implementations of auxiliary methods used in *delete*.

17

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 v from L, apply the appropriate one of the following cases to each child u of v, and then destroy v:

(a) u is hollow and v is its only parent: Add u to L: deletion of v makes u a root.

(b) u has two parents and v is the second: Set $u.ep = null$ and stop processing children of v: u is the last child of v.

(c) u has two parents and v is the first: Set $u.ep = null$ and $u.next = null$.

(d) 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.

Figure 6 gives pseudocode that implements *delete*. Since cases (ii) and (iii) both set $u.ep = null$, this assignment is factored out of these cases. Figure 7 gives auxiliary methods used by *delete* to do links. Array A is a global variable, assumed to be initialized to empty. Integer *max-rank* is also a global variable.

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 dag just before the *delete*, since *max-rank* $= O(\log_\phi N)$ by Corollary 4.7. These are the bounds needed to give Theorem 4.8.

We can reduce the number of pointers per node in this implementation from four to three by using the same field of a node u to hold $u.item$ and $u.ep$, since $u.ep$ is *null* if u is full and $u.item$ is *null* if u is hollow. This requires adding a bit per node to indicate whether the node is full or hollow, trading a bit per node for a pointer per node. We can avoid the extra bit per node by using the rank field to indicate hollow nodes, for example by setting the rank of a hollow node to 0 and that of a full node to its actual rank plus one. This variant has the disadvantage that the shared field for items and extra parents must be able to store pointers to two different types of objects.

An alternative trades three bits per node for one pointer per node but does not require fields storing pointers of different types: Eliminate *ep* pointers. Instead, store with each node u three Boolean variables; $u.new$, true if and only if u was created by a *decrease-key* and has only one child (the node whose item was moved to u); $u.penult$ (penultimate), true if and only if u is the next-to-last child of its first parent; and $u.two$, true if and only if u has two parents. Answer questions (i) and (ii) as follows: (i) u is last on the list of children of v if $v.new$ or $x.penult$, where x is the child before u on the list of children of v (visited just before u during the traversal of the list); and (ii) u has two parents if and only if $u.two$. Question (iii) does not need to be answered: during a delete, when processing each child u of a hollow root v, there are only three cases, not four:

(a') Same as Case (a).

(b') Combines Cases (b) and (c): u has two parents: Set $u.two = false$.

(c') Same as Case (d).

Another alternative eliminates *ep* pointers in a different way: delete the *ep* pointers, store a Boolean variable *u.two* with each node *u* that is true if and only if *u* has two parents, and modify the item field for a hollow node *u* to point to the item *u* once held. This alternative requires that each deleted item *e* has *e.node = null*; otherwise, hollow nodes can have dangling item pointers. With this method, a node *u* is full if and only if *u.item.node = u*, a child *u* of *v* is last on the list of children of *v* if and only if *u.next = null* or *u.item = v.item*, and *v* is the second parent of *u* if and only if *u.item = v.item*. Since item pointers replace *ep* pointers, an item cannot be destroyed until all hollow nodes previously containing it have been destroyed. This alternative is especially appealing if keys and ranks are stored with items instead of nodes. Then one can do a *decrease-key* operation by accessing only the item whose key decreases, not the node *u* containing it. In particular, creating a new node *v* and setting *e.item = v* automatically makes u hollow since then *u.item.node = v*.

Which alternative is best in practice is an experimental question and is likely to depend on the sequence of operations as well as the details of the computer and programming language.

7   Good and Bad Variants

In this section we explore the design space of hollow heaps. We show that our data structures occupy "sweet spots" in the design space: although small changes to these structures preserve their efficiency, larger changes destroy it. We explore variants of the one-root structures of Sections 3 and 4; our results extend to the analogous multi-root structures as well. We consider three classes of structures: 2p-k, 1p-k, and *naïve* 1p-k. Here k is an integer function specifying the rank of the new node v in a *decrease-key* operation as a function of the rank r of the node u made hollow by the operation. Data structure 2p-k is the data structure of Section 4, except that it sets the rank of v in *decrease-key* to be $\max\{k, 0\}$. Thus 2p-$(r-2)$ is exactly the data structure of Section 4. Data structure 1p-k is the data structure of Section 3, except that it sets the rank of v in *decrease-key* to be $\max\{k, 0\}$, and, if $r > k$, it moves to v all but the $r - k$ highest-ranked ranked children of u, as well as the unranked children of u. Thus 1p-$(r-2)$ is exactly the data structure of Section 3. Finally, *naïve* 1p-k is the data structure of Section 4, except that it sets the rank of v in *decrease-key* to be $\max\{k, 0\}$ and it never assigns second parents: when a hollow node u becomes a root, u is deleted and all its children become roots. We consider two regimes for k: *large*, in which $k = r - j$ for some fixed non-negative integer j; and *small*, in which $k = r - f(r)$, where $f(r)$ is a positive non-decreasing integer function that tends to infinity as r tends to infinity.

We begin with a positive result: for any fixed integer $j \geq 2$, both 2p-$(r-j)$ and 1p-$(r-j)$ have the efficiency of Fibonacci heaps. It is straightforward to prove this by adapting the analysis in Sections 3 and 4. As j increases, the rank bound (Corollaries 3.3 and 4.7) decreases by a constant factor, approaching $\lg N$ or $\lg n$, respectively, as j grows, where lg is the base-2 logarithm. The trade-off is that the amortized time bound for *decrease-key* is $O(j + 1)$, increasing linearly with j.

All other variants are inefficient. Specifically, if the amortized time per *delete-min* is $O(\log m)$, where m is the total number of operations, and the amortized time per *make-heap* and *insert* is $O(1)$, then the amortized time per *decrease-key* is $\omega(1)$. We demonstrate this by constructing costly sequences of operations for each variant. We content ourselves merely with showing that the amortized time per *decrease-key* is $\omega(1)$; for at least some variants, there are asymptotically worse sequences than ours. Our results are summarized in the following theorem. The individual constructions appear in Sections 7.1, 7.2, and 7.3.

Theorem 7.1 *Variants* 2p-$(r-j)$ *and* 1p-$(r-j)$ *are efficient for any choice of $j > 1$ fixed independent of r. All other variants, namely naïve* 1p-k *for all k,* 1p-r, 2p-r, 1p-$(r-1)$, 2p-$(r-1)$, *and* 1p-k *and* 2p-k *for k in the small regime are inefficient.*
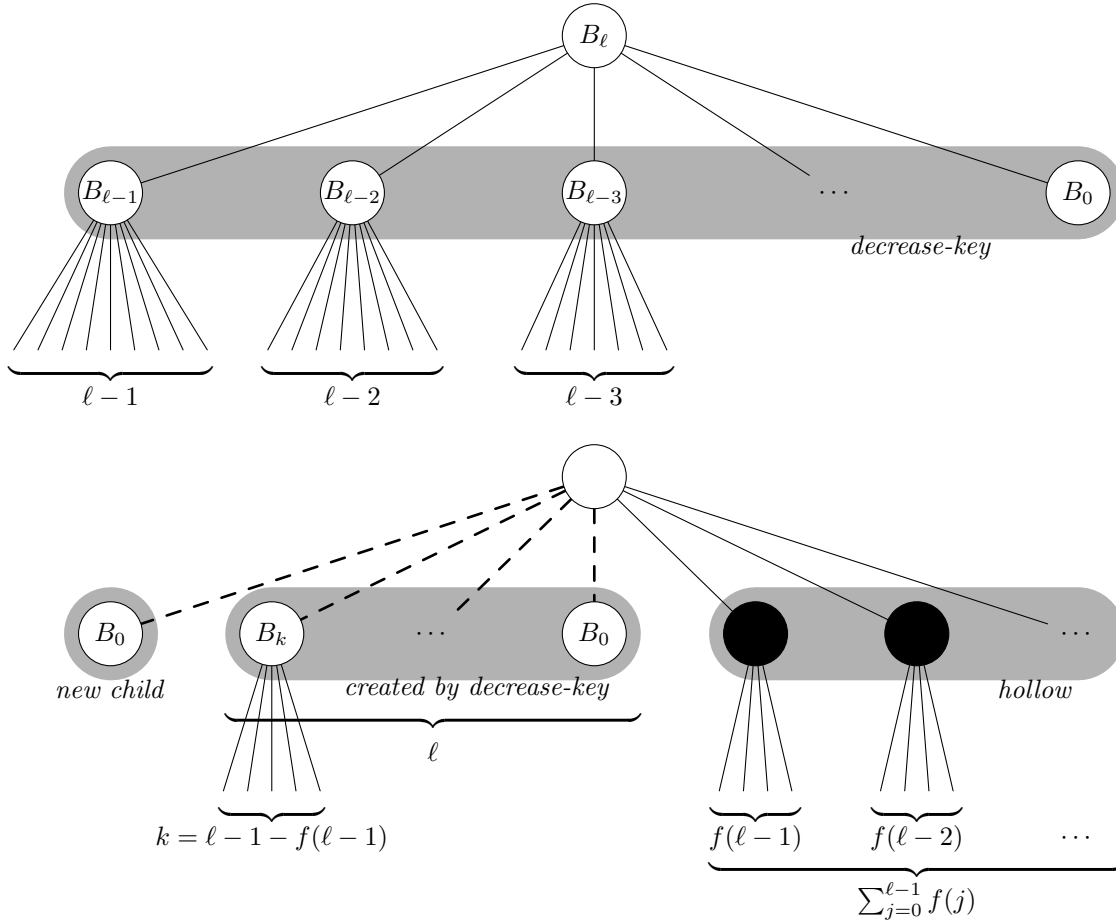
19

Figure 8: The construction for 1p-k. Roots of binomial trees are labeled, and black nodes are hollow. Solid and dashed lines denote ranked and unranked links, respectively. (Top) The initial configuration - a binomial tree B_ℓ. The shaded region shows nodes on whose items *decrease-key* operations are performed. (Bottom) The heap after performing *decrease-key* operations and inserting a new child. The keys of the items in the newly hollow nodes were decreased, resulting in the middle nodes being linked with the root. The number of children of each node is shown at the bottom.

7.1 1p-k for k in the small regime and *naïve* 1p-k for all k

We first consider 1p-k for k in the small regime, i.e., $k = r - f(r)$ where f is a positive non-decreasing function that tends to infinity. We obtain an expensive sequence of operations as follows. We define the *binomial tree* B_ℓ [3, 24] inductively: B_0 is a one-node tree; $B_{\ell+1}$ is formed by linking the roots of two copies of B_ℓ. Tree B_ℓ consists of a root whose children are the roots of copies of $B_0, B_1, \ldots, B_{\ell-1}$ [3, 24]. For any ℓ, build a B_ℓ by beginning with an empty tree and doing $2^\ell + 1$ insertions of items in increasing order by key followed by one *delete-min*. After the insertions, the tree will consist of a root with 2^ℓ children of rank 0. In the *delete-min*, all the links will be ranked, and they will produce a copy of B_ℓ in which each node that is the root of a copy of B_j has rank j. The tree B_ℓ is shown at the top of Figure 8.

Now repeat the following $\ell + 2$ operations 2^ℓ times: do ℓ *decrease-key* operations on the items in the children of the root of B_ℓ, making the new keys greater than that of the key of the item in the root. This makes the ℓ previous children of the root hollow, and gives the root ℓ new children.

Insert a new item whose key is greater than that of the item in the root. Finally, do a *delete-min*. The *delete-min* deletes the root and its ℓ hollow children, leaving the children of the hollow nodes to be linked. Since a hollow node of rank r has $f(r)$ children, the total number of nodes linked after the *delete-min* is $1 + \ell + \sum_{j=0}^{\ell-1} f(j) > (\ell/2)f(\ell/2)$. [3] Each node involved in linking is the root of a binomial tree. Since the total number of nodes remains 2^ℓ, the binomial trees are linked using only ranked links to form a new copy of B_ℓ, and the process is then repeated.

After B_ℓ is formed, each round of $\ell + 2$ consecutive subsequent operations contains only one *delete-min* but takes $\Theta(\ell f(\ell/2))$ time. The total number of operations is $m = O(\ell 2^\ell)$, of which $2^\ell + 1$ are *delete-min* operations. The total time for the operations is $\Theta(\ell 2^\ell f(\ell/2)) = \Theta(m f(\ell/2))$, but the desired time is $O(\ell 2^\ell) = O(m)$. In particular, if the amortized time per *delete-min* is $O(\ell)$ and the amortized time per *make-heap* and *insert* is $O(1)$, then the amortized time per decrease-key is $\Omega(f(\ell/2))$, which tends to infinity with ℓ.

We next consider *naïve* 1p-k. Note that *naïve* 1p-0 is identical to 1p-0, so the two methods do exactly the same thing for the example described above. An extension of the construction shows that *naïve* 1p-k is inefficient for every value of k, provided that we let the adversary choose which ranked link to do when more than one is possible. Method *naïve* 1p-k is identical to *naïve* 1p-0 except that nodes created by *decrease-key* may not have rank 0. The construction for *naïve* 1p-k deals with this issue by inserting new nodes with rank 0 that serve the function of nodes created by *decrease-key* for *naïve* 1p-0. The additional nodes with non-zero rank are linked so that they do not affect the construction.

We build an initial B_ℓ as before. Then we do ℓ *decrease-key* operations on the items in the children of the root, followed by $\ell + 1$ *insert* operations of items with keys greater than that of the root, followed by one *delete-min* operation, and repeat these operations 2^ℓ times. When doing the linking during the *delete-min*, the adversary preferentially links newly inserted nodes and grandchildren of the deleted root, avoiding links involving the new nodes created by the *decrease-key* operations until these are the only choices. Furthermore, it chooses keys for the newly inserted items so that one of them is the new minimum. Then the tree resulting from all the links will be a copy of B_ℓ with one or more additional children of the root, whose descendants are the nodes created by the *decrease-key* operations. After the construction of the initial B_ℓ, each round of $2\ell + 2$ subsequent operations maintains the invariant that the tree consists of a copy of B_ℓ with additional children of its root, whose descendants are all the nodes added by *decrease-key* operations.

The analysis is the same as for 1p-0, i.e. for the case $f(r) = r$. The total number of operations is $m = O(\ell 2^\ell)$, and the desired time is $O(\ell 2^\ell) = O(m)$. The total time for the operations is however $\Theta(\ell^2 2^\ell) = \Theta(m\ell)$. Thus, the construction shows that *naïve* 1p-k for any value of k takes at least logarithmic amortized time per *decrease-key*.

7.2   2p-r, 1p-r, 2p-$(r-1)$, and 1p-$(r-1)$

Next we consider 2p-r, 1p-r, 2p-$(r-1)$, and 1p-$(r-1)$. To get a bad example for each of these methods, we construct a tree T_ℓ with a full root, having full children of ranks $0, 1, \ldots, \ell - 1$, and in which all other nodes, if any, are hollow. Then we repeatedly do an *insert* followed by a *delete-min*, each repetition taking $\Omega(\ell)$ time.

In these constructions, all the *decrease-key* operations are on nodes having only hollow descendants, so the operations maintain the invariant that every hollow node has only hollow descendants. If this is true, the only effect of manipulating hollow nodes is to increase the cost of the operations, so we can ignore hollow nodes; or, equivalently, regard them as being deleted as soon as they are
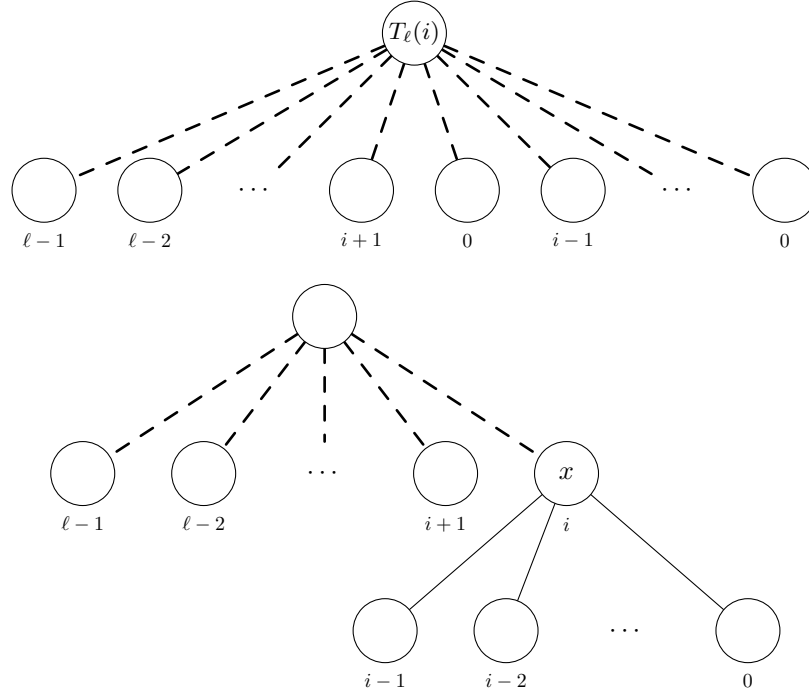
---

[3]Assume for simplicity that ℓ is even.

Figure 9: The construction for 2p-$(r-1)$ and 1p-$(r-1)$. Only full nodes are shown. Solid and dashed lines denote ranked and unranked links, respectively. Ranks are shown beneath nodes. (Top) The tree $T_\ell(i)$. (Bottom) The tree obtained from $T_\ell(i)$ by inserting an item and performing a *delete-min* operation.

created. Furthermore, with this restriction 2p-k and 1p-k have the same behavior, so one bad example suffices for both 2p-r and 1p-r, and one for 2p-$(r-1)$ and 1p-$(r-1)$.

Consider 2p-r and 1p-r. Given a copy of T_ℓ in which the root has rank ℓ, we can build a copy of $T_{\ell+1}$ in which the root has rank $\ell + 1$ as follows: First, insert an item whose key is less than that of the root, such that the new node becomes the root. Second, do a *decrease-key* on each item in a full child of the old root (a full grandchild of the new root), making each new key greater than that of the new root. Third, insert an item whose key is greater than that of the new root. Finally, do a *delete-min*. Just before the *delete-min*, the new root has one full child of each rank from 1 to ℓ, inclusive, and two full children of rank 0. In particular one of these children is the old root, which has rank ℓ. The *delete-min* produces a copy of $T_{\ell+1}$. (The *decrease-key* operations produce hollow nodes, but no full node is a descendant of a hollow node.) It follows by induction that one can build a copy of T_ℓ for an arbitrary value of ℓ in $O(\ell^2)$ operations. These operations followed by ℓ^2 repetitions of an *insert* followed by a *delete-min* form a sequence of $m = O(\ell^2)$ operations that take $\Omega(\ell^3) = \Omega(m^{3/2})$ time.

A similar but more elaborate example is bad for 2p-$(r-1)$ and 1p-$(r-1)$. Let $T_\ell(i)$ be T_ℓ with the child of rank i replaced by a child of rank 0. In particular, $T_\ell(0)$ is T_ℓ, and $T_{\ell+1}(\ell+1)$ is T_ℓ with the root having a second child of rank 0. $T_\ell(i)$ is shown at the top of Figure 9.

Given a copy of $T_\ell(i)$ with $i > 0$, we can build a copy of $T_\ell(i-1)$ as follows: First, insert an item whose key is greater than that of the root but less than that of all other items. Now the root has three children of rank 0. Second, do a *delete-min*. The just-inserted node will become the root, the other children of the old root having rank less than i will be linked by ranked links to form a tree whose root x has rank i and is a child of the new root, and the remaining children of the old root

22

will become children of the new root. Node x has exactly one full proper descendant of each rank from 0 to $i - 1$, inclusive. The tree obtained after performing the *delete-min* operation is shown at the bottom of Figure 9. (In the figure we assume that the key of the child of the old root of rank $j < i$ is smaller than the key of the child of the old root of rank $j - 1$ for every $1 \leq j < i$. In this case x is the child of rank $i - 1$ of the old root and its children after the *delete-min* are the children of the old root of rank $\leq i - 2$. But unlike the situation shown in the figure, the descendants of x can in general be linked arbitrarily.) Finally, do a *decrease-key* on each of the items in the full proper descendants of x in a bottom-up order (so that each *decrease-key* is on an item in a node with only hollow descendants), making each new key greater than that of the root. The rank of each new node created this way is 1 smaller than the rank of the node it came from, except for the node that already has rank 0. The root thus gets two new children of rank 0 and one new child of each rank from 1 to $i - 2$. The result is a copy of $T_\ell(i - 1)$, with some extra hollow nodes, which we ignore. We can convert a copy of $T_\ell(0) = T_\ell$ into a copy of $T_{\ell+1}(\ell + 1)$ by inserting a new item with key greater than that of the root. It follows by induction that one can build a copy of T_ℓ in $m = O(\ell^3)$ operations. These operations followed by ℓ^3 repetitions of an *insert* followed by a *delete-min* take a total of $\Omega(\ell^4) = \Omega(m^{4/3})$ time but the desired time is $O(\ell^3 \log \ell) = O(m \log m)$.

7.3   2p-k

Finally, we consider 2p-k for any k in the small regime. We again construct a tree for which we can repeat an expensive sequence of operations. We first give a construction for 2p-0 and then show how to generalize the construction to all choices of k in the small regime.

Define the tree S_ℓ inductively as follows. Tree S_0 is a single node. For $\ell > 0$, S_ℓ is a tree with a full root of rank ℓ, having one hollow child that is the root of $S_{\ell-1}$ and having full children of ranks $0, 1, \ldots, \ell - 1$, with the i-th full child being the root of a copy of B_i. The tree S_ℓ is shown at the top of Figure 10. Let R_ℓ be a tree obtained by linking copies of $S_0, S_1, \ldots, S_{\ell-1}$ to S_ℓ, with the root of S_ℓ winning every link. The tree R_ℓ is shown at the bottom of Figure 10. We show how to build a copy of R_ℓ for any ℓ. Then we show how to do an expensive sequence of operations that starts with a copy of R_ℓ and produces a new one. By building one R_ℓ and then doing enough repetitions of the expensive sequence of operations, we get a bad example.

To build a copy of R_ℓ for arbitrary ℓ, we build a related tree Q_ℓ that consists of a root whose children are the roots of copies of $S_0, S_1, \ldots, S_\ell$, with the root of S_ℓ having the smallest key among the children of the root of Q_ℓ. We obtain R_ℓ from Q_ℓ by doing a *delete-min*.

We build $Q_0, Q_1, \ldots, Q_\ell$ in succession. Tree Q_0 is just a node with one full child of rank 0, obtainable by a *make-heap* and two *insert* operations. Given Q_j, we obtain Q_{j+1} by a variant of the construction for 1p-0. Let x_i be the root of the existing copy of S_i for $i = 0, \ldots, j$. In the following, all new keys are greater than the key of the root, so that the root remains the same throughout the sequence of operations. First we do *decrease-key* operations on the roots $x_0, x_1, \ldots, x_j$ of the existing copies of $S_0, S_1, \ldots, S_j$. For $i = 0, \ldots, j$, the node x_i is thus made hollow and becomes a child of a new node y_i of rank 0. Note that a copy of S_{i+1} can be obtained from repeated, ranked linking of y_i and $2^{i+1} - 1$ nodes of rank 0 where y_i wins every link in which it participate. We next do enough *insert* operations to provide the nodes to build $S_1, S_2, \ldots, S_{j+1}$ in this way. The total number of nodes needed is $\sum_{i=0}^{j}(2^{i+1} - 1)$. Finally, we do two additional *insert* operations, followed by a *delete-min*. The two extra nodes are for a copy of S_0 and for a new root when the old root is deleted.

Deletion of hollow roots by *delete-min* makes y_i the only parent of x_i for all $i = 0, \ldots, j$. We are left with a collection of $1 + \sum_{i=0}^{j+1} 2^i$ roots of rank 0. We do ranked links to build the needed copies of $S_0, S_1, \ldots, S_{j+1}$ in decreasing order. Finally, we link the new root with each of the roots of the new copies of S_i.
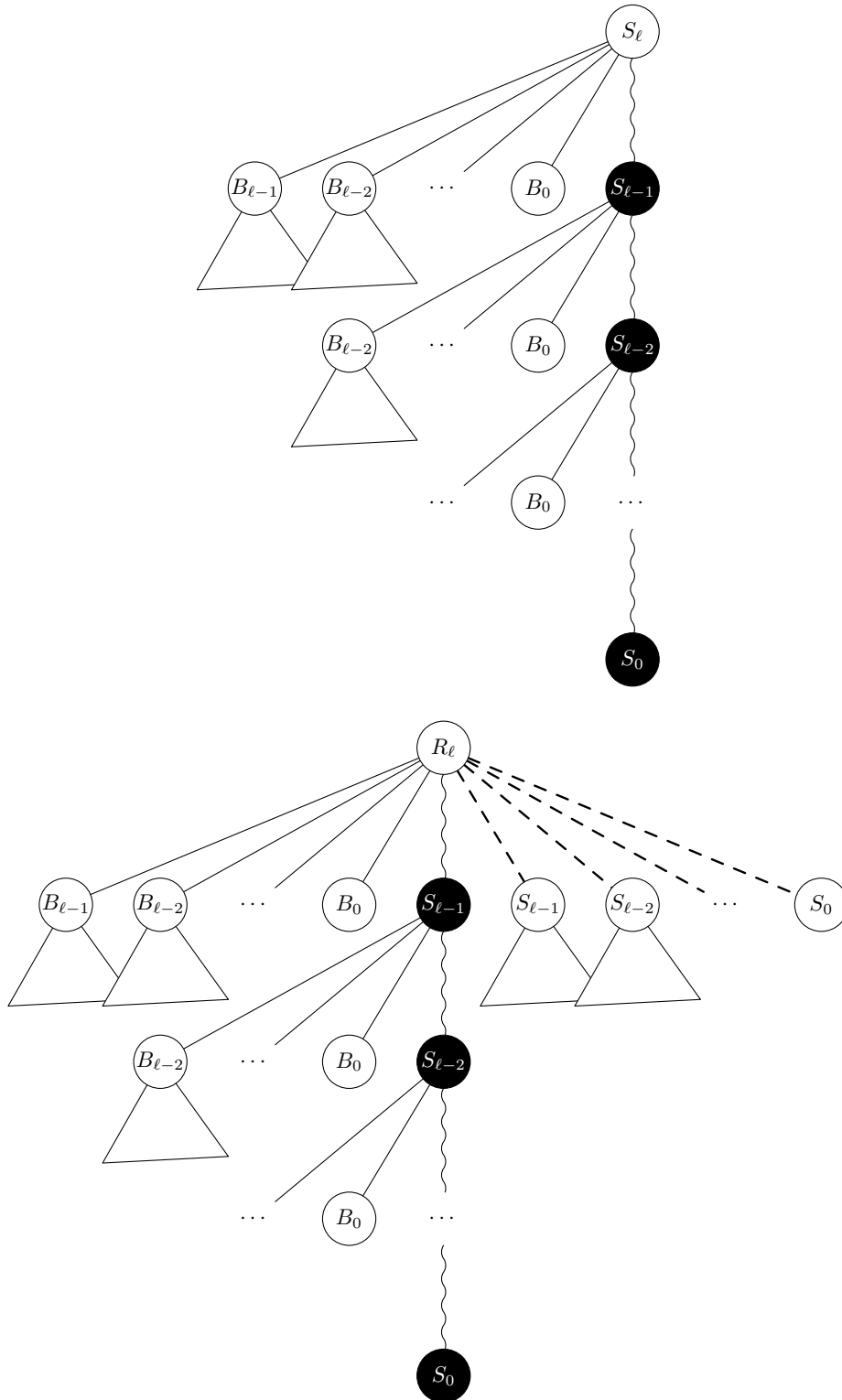
23

Figure 10: The trees S_ℓ (top) and R_ℓ (bottom). Every node is labeled by the type of its subtree. The triangles denote such subtrees. Black nodes are hollow. Solid and dashed lines denote ranked and unranked links, respectively. Squiggly lines denote second parents.
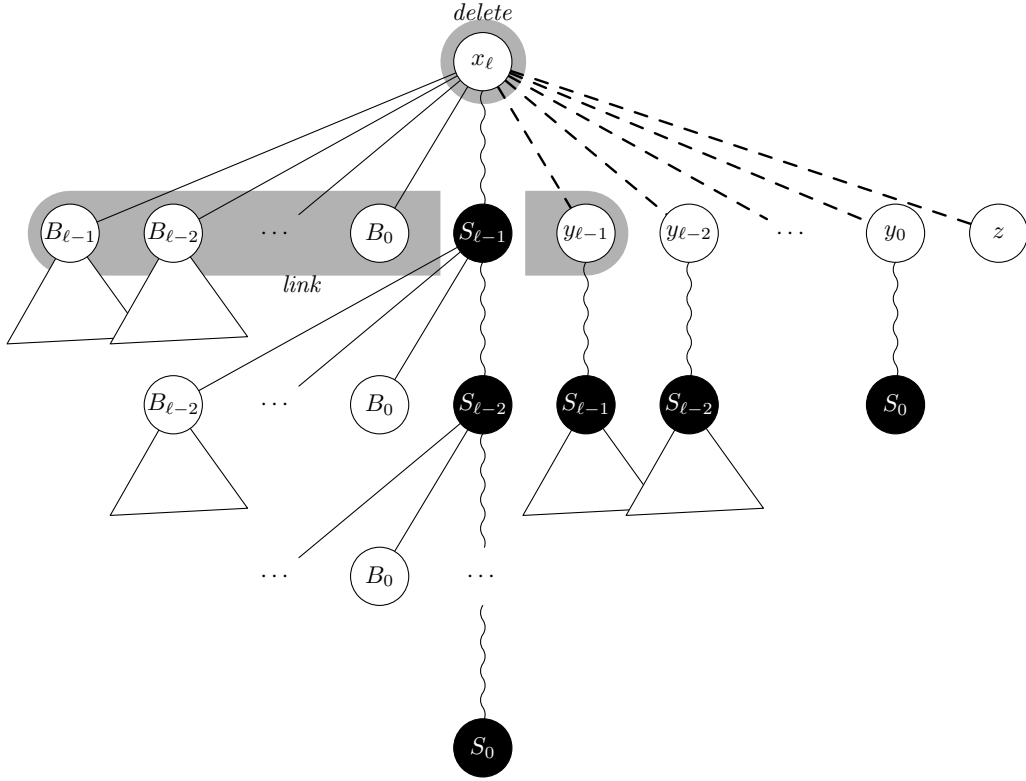
Figure 11: The tree obtained from R_ℓ by performing ℓ *decrease-key* operations and one *insert*. Every node is labeled by its name or the type of its subtree. The triangles denote such subtrees. Black nodes are hollow. Solid and dashed lines denote ranked and unranked links, respectively. Squiggly lines denote second parents. Edges connecting x_ℓ to the children of y_i for $i = 0, \ldots, \ell - 1$ have been omitted.

Suppose we are given a copy of R_ℓ. Let x_j, for $j = 0, 1, \ldots, \ell$, be the root of the copy of S_ℓ. In particular, x_ℓ is the root of R_ℓ. We can do an expensive sequence of operations that produces a new copy of R_ℓ as follows: Do *decrease-key* operations on x_j for $j = 0, 1, \ldots, \ell - 1$, giving each x_j a second parent y_j. Make all the new keys larger than that of x_ℓ and smaller than those of all children of x_ℓ; among them, make the key of $y_{\ell-1}$ the smallest. Next, insert a new item with key greater than that of $y_{\ell-1}$; let z be the new node holding the new item. Figure 11 shows the resulting situation. Next, do a *delete-min*. This makes y_j the only parent of x_j for $j = 0, \ldots, \ell - 1$. Once the hollow roots are deleted, the remaining roots are z, the y_j, and the roots of $\ell - i$ copies of B_i for $i = 0, 1, \ldots, \ell - 1$. Finish the *delete-min* by doing ranked links of each y_j with the roots of copies of B_i for $i = 0, 1, \ldots, j$, forming new copies of $S_0, S_1, \ldots, S_\ell$ (z is the root of a copy of S_0; y_j is the root of a copy of S_{j+1}), and link the roots of these copies by unranked links. The result is a new copy of R_ℓ. The sequence of operations consists of one *insert*, ℓ *decrease-key* operations, and one *delete-min* and takes $O(\ell^2)$ time.

The number of nodes in R_ℓ is $O(2^\ell)$, as is the number of operations needed to build it and the time these operations take. Having built R_ℓ, if we then do 2^ℓ repetitions of the expensive sequence of operations described above, the total number of operations is $m = O(\ell 2^\ell)$. The operations take $\Theta(\ell^2 2^\ell) = \Theta(m\ell)$ time, whereas the desired time is $O(m)$.

An extension of the same construction shows the inefficiency of 2p-k for k in the small regime: instead of doing one *decrease-key* on each appropriate item, we do enough to reduce to 0 the rank

of the full node holding the item. Suppose $k = r - f(r)$, where $f(r)$ is a positive non-decreasing function tending to infinity. Then the number of *decrease-key* operations needed to reduce the rank of the node holding an item to 0, given that the rank of the initial node holding the item is k, is at most $k/f(\sqrt{k}) + \sqrt{k}$. It follows that the extended construction does at most $\ell^2/f(\sqrt{\ell}) + \ell^{3/2}$ *decrease-key* operations per round, and the amortized time per decrease-key is $\Omega(f(\sqrt{\ell})) = \omega(1)$, assuming that the amortized time per *delete* is $O(\ell)$ and that of *make-heap* and *insert* is $O(1)$.

References

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

[2] G.S. Brodal, G. Lagogiannis, and R.E. Tarjan. Strict Fibonacci heaps. In *Proceedings of the 44th ACM Symposium on Theory of Computing (STOC)*, pages 1177–1184, 2012.

[3] M.R. Brown. Implementation and analysis of binomial queue algorithms. *SIAM Journal on Computing*, 7(3):298–319, 1978.

[4] T.M. Chan. Quake heaps: A simple alternative to Fibonacci heaps. In *Space-Efficient Data Structures, Streams, and Algorithms*, pages 27–32, 2013.

[5] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[6] J.R. Driscoll, H.N. Gabow, R. Shrairman, and R.E. Tarjan. Relaxed heaps: an alternative to Fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31(11):1343–1354, 1988.

[7] J. Edmonds. Optimum branchings. *J. Res. Nat. Bur. Standards*, 71B:233–240, 1967.

[8] A. Elmasry. The violation heap: a relaxed Fibonacci-like heap. *Discrete Math., Alg. and Appl.*, 2(4):493–504, 2010.

[9] M.L Fredman. On the efficiency of pairing heaps and related data structures. *Journal of the ACM*, 46(4):473–501, 1999.

[10] M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[11] H.N. Gabow, Z. Galil, T.H. Spencer, and R.E. Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6:109–122, 1986.

[12] B. Haeupler, S. Sen, and R.E. Tarjan. Rank-pairing heaps. *SIAM Journal on Computing*, 40(6):1463–1485, 2011.

[13] P. Høyer. A general technique for implementation of efficient priority queues. In *Proceedings of the 3rd Israeli Symposium on the Theory of Computing and Systems (ISTCS)*, pages 57–66, 1995.

[14] J. Iacono and Ö. Özkan. Why some heaps support constant-amortized-time decrease-key operations, and others do not. In *Proc. of 41st ICALP*, pages 637–649, 2014.

[15] H. Kaplan, N. Shafrir, and R.E. Tarjan. Meldable heaps and boolean union-find. In *Proceedings of the 34th ACM Symposium on Theory of Computing (STOC)*, pages 573–582, 2002.

[16] H. Kaplan and R.E. Tarjan. Thin heaps, thick heaps. *ACM Transactions on Algorithms*, 4(1):1–14, 2008.

[17] H. Kaplan, R.E. Tarjan, and U. Zwick. Fibonacci heaps revisited. *CoRR*, abs/1407.5750, 2014.

[18] D.E. Knuth. *Sorting and searching*, volume 3 of *The art of computer programming*. Addison-Wesley, second edition, 1998.

[19] G.L. Peterson. 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.

[20] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.

[21] T. Takaoka. Theory of 2-3 heaps. *Discrete Applied Mathematics*, 126(1):115–128, 2003.

[22] R.E. Tarjan. *Data structures and network algorithms*. SIAM, 1983.

[23] R.E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6:306–318, 1985.

[24] J. Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21:309–314, 1978.