Hollow Heaps

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

Abstract

We introduce the *hollow heap*, a very simple data structure whose performance matches that of the classical *Fibonacci heap*. Hollow heaps have the following advantages over Fibonacci heaps:

(i) *decrease-key* operations are particularly simple: each one takes $O(1)$ time worst-case and does no cuts.

(ii) The outcomes of all comparisons are explicitly represented in the data structure, so none are wasted.

(iii) Each node in a hollow heap has only three pointers: no parent pointers are needed, and lists of children are singly linked.

Hollow heaps combine two novel ideas. The first is to use lazy deletion and re-insertion to do *decrease-key* operations. Such operations produce *hollow* nodes, i.e., nodes that do not contain items, giving the data structure its name. The second is to represent a heap by a dag (directed acyclic graph) instead of a tree or a set of trees.

[*]Department of Management Science & Engineering, Stanford University, USA; and Department of Computer Science, Aarhus University, Denmark. 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). E-mail: `tdh@cs.au.dk`.

[†]Blavatnik School of Computer Science, Tel Aviv University, Israel. Research supported by the Israel Science Foundation grant no. 822-10, 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). E-mail: `haimk@post.tau.ac.il`.

[‡]Department of Computer Science, Princeton University, Princeton, NJ 08540, USA and Intertrust Technologies, Sunnyvale, CA 94085, USA.

[§]Blavatnik School of Computer Science, Tel Aviv University, Israel. Research supported by BSF grant no. 2012338 and by The Israeli Centers of Research Excellence (I-CORE) program (Center No. 4/11). E-mail: `zwick@tau.ac.il`.

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. If h is empty, return *null*.

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) : Given that e is an item in heap h, return a heap formed by deleting e 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 as part of the input. 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, one needs a separate disjoint set data structure to keep track of the partition of items into heaps. For further discussion of this see [14].

Fredman and Tarjan [9] 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, 9] and fast algorithms for undirected and directed minimum spanning trees [7, 10]. 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].

In spite of its many competitors, Fibonacci heaps remain one of the simplest heap implementations to describe and code. We present yet another competitor that we believe surpasses Fibonacci heaps in its simplicity, and possibly in its practicality. Our data structure has two novelties: it uses lazy deletion to do *decrease-key* operations in a simple and natural way, thereby 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.

In a Fibonacci heap, a *decrease-key* operation 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 triggers 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., [15]). 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 efficient.

The remainder of our paper consists of eight 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. Sections 6 and 7 give implementation details for the data structures in Sections 2 and 4, respectively. Section 8 explores the design space of the data structures in Sections 2 and 4, identifying variants that are efficient and variants that are not. Section 9 contains final remarks.

2 Hollow Heaps

Our data structure extends and refines a well-known generic representation of heaps. The structure is *exogenous* rather than *endogenous* [22]: 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 v is a *parent* of u and u is a *child* of v. A node with no parents is a *root*.

We represent a non-empty heap by a dag whose nodes contain the heap items, at most one per node. We call a node *full* if it holds an item and *hollow* if not. 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. If u is a full node, $u.item$ is the item u holds, and $u.key$ is the current key of $u.item$. If u is a hollow node, $u.item$ is *null*, and $u.key$ is the key of the item u once held just before the item was moved from u or deleted. A full node has at most one parent, a hollow node has at most two.

The dag is topologically ordered by key: if v is a parent of u, then $u.key \geq v.key$. 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 contains an item of minimum key. We access the dag via its root.

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 dag with the existing dag.

We do *decrease-key* and *delete* operations using lazy deletion. To do *decrease-key*(e, k, h), let u be the node holding e. If $u = h$ (u is the root of the dag), merely set $u.key = k$. Otherwise (u is a child), create a new node v, move e from u to v, set $v.key = k$, do $link(h, v)$, and if v is the loser of this link, make u a child of v. Moving e to v makes u hollow. If the *decrease-key* makes v a parent of u, the number of parents of u increases from one to two. We call v the *second parent* of u, in contrast to its first parent, previously acquired via a link with a full root. A node only becomes hollow once, so it acquires a second parent at most once.

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 heap, the deletion is complete. Otherwise, repeatedly delete hollow roots and link full roots until there are no hollow roots and at most one full root.

Theorem 2.1 *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 one.*

Proof: A straightforward induction on time shows that heap order is maintained, which implies that the graph representing a heap is a dag. Correctness and the other invariants also follow by induction on time. □

The dag representing a hollow heap is almost a tree, since each node has at most two parents and there is at most one root. If there are no *decrease-key* operations, it *is* a tree, since then each node has at most one parent. If there are *decrease-key* operations, the dag can be converted into a tree by choosing one parent (either one) for each hollow node that has two parents. There is at most one such node per *decrease-key*. In this sense the dag is "tree-like". The dag captures the outcomes of all key comparisons between undeleted items.

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* operation deletes 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.

Figure 1 shows a sequence of operations on a hollow heap.

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 in the worst case: there are no cascading cuts as in [9], no cascading rank changes as in [12, 17], no restructuring steps to eliminate heap-order violations as in [2, 6, 16], and significant space savings (See Section 6). The dag representation represents all key comparisons between unlabeled items, allowing us to avoid restructuring altogether: links are cut only when hollow roots are deleted.

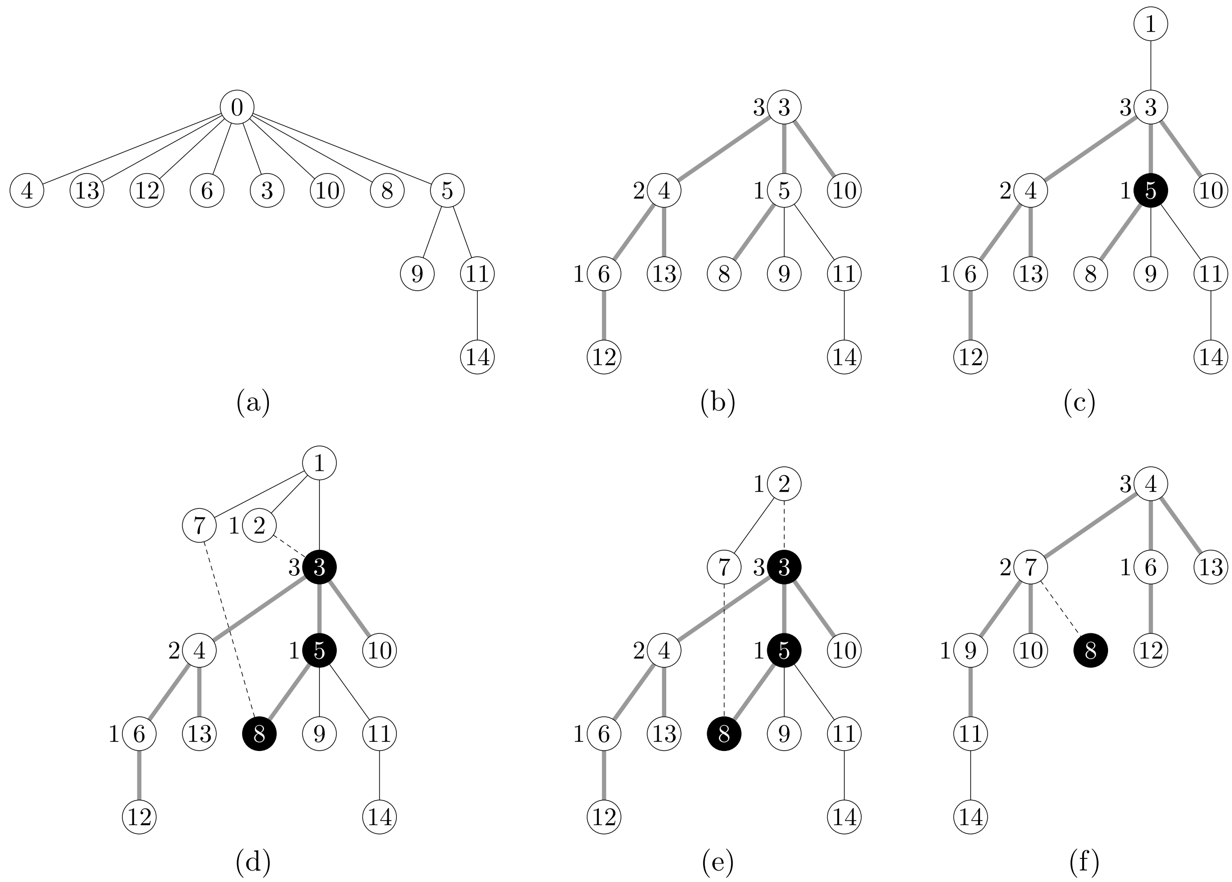

Figure 1: Operations on a hollow heap. Numbers in nodes are keys. Black nodes are hollow. Bold gray, solid, and dashed lines denote ranked links, unranked links, and second parents, respectively. Numbers next to nodes are non-zero ranks. (a) Successive insertions of items with keys 14, 11, 5, 9, 0, 8, 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 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 initial rank of nodes created by *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 say a node with a virtual parent is a *virtual child* of its virtual parent, and a node u is a *virtual descendant* of a node v if there is a path from u to v via virtual parents. 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.)

When a root becomes a child, its first parent becomes its virtual parent. 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, all but the two of highest ranks become virtual children of v.

The next two lemmas show that virtual parents are well-defined: each node has at most one virtual parent at a time, and all ranked virtual children of a given node have different ranks, so the two of highest ranks are unique.

Lemma 3.1 *A node x has at most one virtual parent at a time. There is a path in the dag from x to its virtual parent if it has one.*

Proof: We prove the lemma for a given node x by induction on time. When x is created, it has no virtual parent, so the lemma holds. It acquires a virtual parent u when it loses a link to a node u. Since the link makes x a child of u, the lemma holds for x after the link. While x has a virtual parent and the lemma holds, x cannot be a root, so it cannot acquire another virtual parent via losing a link. Its virtual parent can change via a *decrease-key*, however. Suppose x has virtual parent u and a *decrease-key* changes its virtual parent to v. Before the *decrease-key* there is a path in the dag from x to u by the induction hypothesis. There is also a path from u to the root h of the dag. If v wins the link with h, there is a path from u to h to v after the *decrease-key*. If v loses the link with h, the *decrease-key* makes v the second parent of u, creating a path in the dag from x to u to v. Hence the lemma holds after the *decrease-key*. The only other action that might affect the truth of the lemma is destruction of a node. Since only roots are destroyed, destruction of a node either leaves the path from x to its virtual parent intact, or it destroys the virtual parent. Either case maintains the truth of the lemma. □

The arc (u, v) added to the dag by *decrease-key* represents the inequality $u.key \geq v.key$. If this arc is not redundant and *decrease-key* fails to add it, our analysis breaks down. Indeed, the resulting algorithm does not have the desired efficiency, as we show by example in Section 8. Adding the arc only when v loses the link to h is an optimization: if v wins the link, the dag has a path from u to v without it.

Lemma 3.2 *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*. 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 [18].

Corollary 3.3 *A node of rank r has at least $F_{r+3} - 1$ virtual descendants.*

Proof: The proof is by induction on r using Lemma 3.2. 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 3.2. By Lemma 3.1 each virtual child has a unique virtual parent, 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 3.4 *The maximum rank of a node in a hollow heap of N nodes is at most $\log_\phi N$.*

Proof: The theorem is immediate from Corollary 3.3 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 start 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 3.4. 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 3.2 (+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 3.2: each ranked virtual child of u gains 1 in potential. By Theorem 3.4, $u.rank = O(\log N)$. 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 3.5 *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.

Lemma 4.1 *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.*

Proof: Delete the word "virtual" in the proof of Lemma 3.2. □

Corollary 4.2 *A node of rank r in an eager hollow heap has at least $F_{r+3} - 1$ descendants.*

Proof: Delete the word "virtual" in the proof of Corollary 3.3. □

Theorem 4.3 *The maximum rank of a node in an eager hollow heap of N nodes is at most $\log_\phi N$.*

Proof: The proof is identical to that of Theorem 3.4. □

Theorem 4.4 *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 a heap of N nodes.*

Proof: Delete the word "virtual" in the proof of Theorem 3.5. □

Whether lazy or eager hollow heaps are better in practice is an experimental question. Eager hollow heaps are slightly more complicated to implement (see Section 7), and they do unnecessary restructuring. But they do avoid the (small) overhead of maintaining a dag instead of a tree.

An alternative way to think about eager hollow heaps is as a variant of Fibonacci heaps. In a Fibonacci heap, a *decrease-key* on a child u that makes $u.key < u.p.key$ begins by cutting u and its entire subtree from $u.p$. Such a cut triggers a cascade of cuts of (old) ancestors of u along with their subtrees. These cuts 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. It also eliminates the need for double linking of sibling lists and for parent pointers, both of which are needed in Fibonacci heaps.

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 [5, 9], various minimum spanning tree algorithms [5, 9, 10, 20], and Edmonds' optimum branching algorithm [7, 10]. 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 as follows: Make the dag into a tree by ignoring the second parent of each node with two parents. Give each full node a rank of 0. Make each full child a child of its nearest full proper ancestor. Finally, destroy all the hollow nodes. Rebuilding can be done in a single tree traversal, taking $O(N)$ time but no key comparisons. 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 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. 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 reduce 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 on 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. 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 we need a way to test whether a child u of a node v is last on the list of children of v. If u has only one parent, this is easy: u is last iff $u.next = null$. To make this test possible for hollow nodes with two parents, and to support testing whether a hollow node *has* two parents, we store with each hollow node u a pointer $u.sp$ to the second parent of u if u has two parents, *null* if not. Node u is last on the list of children of a parent v iff $u.next = null$ (u is last on any list of children containing u) or $u.sp = v$ (u has two parents, and v is its second one). A *decrease-key* operation makes newly hollow node u a child of new node $v = u.sp$ by setting

$v.child = u$ but not changing $u.next$: $u.next$ is the next sibling of u in the child list of its first parent.

With each node u we also store $u.item$, $u.key$, and $u.rank$. Ranks are small integers, each requiring $\lg\lg N + O(1)$ bits of space. With each item e we store $e.node$, the node holding item e.

Implementation of all the heap operations except *delete* is straightforward. Figure 2 gives such implementations in pseudocode; Figure 3 gives implementations of auxiliary methods used in the code of Figure 2.

```
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
    if k > h.key:
    {   v.child = u
        u.sp = v }
    else: u.sp = null
    return link(v, h)   }

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

Figure 2: Implementations of all lazy hollow heap operations except *delete*.

```
make-node(e, k):
{   u = new-node()
    u.item = e
    e.node = u
    u.child = 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 3: Implementations of auxiliary methods used in the code of Figure 2. Rank updates during ranked links are done in *delete*. See Figure 4.

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 $u.sp = null$ (x is the only parent of u): Add u to L: deletion of x makes u a root.

(ii) u is hollow and $u.sp = x$ (u has two parents; x is the second): Set $u.sp = null$ and stop processing children of x: u is the last child of x.

(iii) u is hollow, $u.sp \neq null$, and $u.sp \neq x$ (u has two parents; x 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.

Figure 4 gives pseudocode that implements *delete*. Since cases (ii) and (iii) both set $u.sp = null$, this assignment is factored out of these cases.

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 $\textit{max-rank} = O(\log_\phi N)$ by Theorem 3.4. These are the bounds claimed in Section 3.

This implementation uses a pointer per item and four pointers, a key, and a rank per node. We can reduce the number of pointers per node from four to three by observing that only a full node needs an item pointer, and only a hollow node needs a second parent pointer. Thus we can use the same node field to store both, assuming the type system permits it. Doing this requires a new way to test whether a node is hollow, since the test $u.item = null$ will no longer suffice. One option is to add a bit to each node indicating whether it is hollow. Another is to encode this bit in the rank field, since hollow nodes do not need ranks. We can for example set the rank of a hollow node to -1. Equivalently, we can give each hollow node a rank of 0 and each full node a rank one greater than its actual rank, by initializing the rank of a node created by an insert to 1 instead of 0 and changing the rank test in *decrease-key* to "$u.rank > 3$".

As a matter of theoretical interest, we note that one can eliminate the second parent pointer entirely and instead use two bits $u.one$ and $u.nl$ per node u to identify last nodes on lists of children. Bit $u.one$ is *true* if u has exactly one child, and $u.nl$ is *true* if u is the next-to-last node on the list of children of its first parent. These bits need to be initialized when nodes are created and set appropriately in *link* and *decrease-key*. The only purpose of $u.one$ is to support setting $u.nl$ correctly: just before a link makes u a child of v, if $v.one$ is *true*, set $u.nl$ *true* and $v.one$ *false*; if $v.child = null$, set $v.one = true$. When processing a child u of x in *delete*, if $x.one = true$ then u is the only child of x, and hence the last one. Otherwise, the last child of x is $u.next$, where u is the unique child that has $u.nl = true$. This method also needs a bit per hollow node that is *true* if the node has two parents. This bit can be encoded in the rank field as discussed above, but the one-child and next-to-last bits cannot, since every node needs these bits, not just hollow nodes.

Another possible way to save space is to store keys with items instead of nodes. (An actual application may need to store keys with items anyway.) Then we need one key per item instead of one key per node. This may save space, but it costs time since obtaining the key of a node requires following its item pointer.

```
delete(e, h):
{   e.node.item = null
    e.node.sp = null
    e.node = null
    if h.item ≠ null: return h            /* Non-minimum deletion */
    max-rank = 0
    h.next = null
    while h ≠ null:            /* While L not empty */
    {   w = h.child
        x = h
        h = h.next
        while w ≠ null:
        {   u = w
            w = w.next
            if u.item = null:
            {   if u.sp = null:
                {   u.next = h
                    h = u  }  }
                else:
                {   if u.sp = x: w = null
                    else: u.next = null
                    u.sp = null }
            else:
            {   while A[u.rank] ≠ null:
                {   u = link(u, A[u.rank])
                    A[u.rank] = null
                    u.rank = u.rank + 1  }
                A[u.rank] = u
                if u.rank > max-rank: max-rank = u.rank  }  }
        destroy x  }
    for i = 0 to max-rank:
    {   if A[i] ≠ null:
        {   if h = null: h = A[i]
            else: h = link(h, A[i])
            A[i] = null  }  }
    return h }
```

Figure 4: Implementation of *delete* in lazy hollow heaps.

We can save time by putting the auxiliary functions in-line and eliminating redundant assignments, but a good optimizing complier may do this anyway.

7 Implementation of Eager Hollow Heaps

Now we turn to the implementation of eager hollow heaps (the data structure of Section 4). To support the movement of children, we need to maintain the set of children of a vertex in an appropriate order. We store each set of children in a singly linked circular list, with the ranked children first, in decreasing order by rank, followed by the unranked children, in any order. This takes two pointers per node: $u.child$ is the *last* child of u, *null* if u has no children; $u.next$ is the next sibling of u, *null* if u has no siblings. This representation allows us to add a child to the front or back of a list of children, and to catenate a list of children to another list. Each node u also has a pointer $u.item$ to the item it holds, *null* if u is hollow, and each item e has a pointer $e.node$ to the node holding it. The total space needed is three pointers, a key, and a rank per node, and one pointer per item.

Except for *delete*, all the heap operations have straightforward implementations. Only that of *decrease-key* differs significantly from its implementation in lazy hollow heaps. We implement both ranked and unranked links by one method, $link(ranked, x, y)$, where *ranked* is a bit that is *true* if the link is a ranked link and *false* otherwise. As in the implementation of hollow heaps, rank updates are done in *delete*. The implementations of *make-heap*, *find-min*, *insert*, and *delete-min* are identical to those in lazy hollow heaps; that of *meld* differs only in using $link(false, v, w)$ in place of $link(v, w)$. Figure 5 gives an implementation of *decrease-key* in eager hollow heaps; Figure 6 gives implementations of the auxiliary methods *make-node*, *link*, and *add-child*.

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

Figure 5: Implementation of *decrease-key* in eager hollow heaps.

The implementation of *delete* uses only a single list of roots, instead of a list of hollow roots and a list of the children of a hollow root, since circular linking of lists of children allows their catenation in $O(1)$ time. Whereas *decrease-key* is more complicated in eager hollow heaps than in lazy ones, *delete* is simpler, since each node has only one parent at a time. Figure 7 gives an implementation of *delete*.

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

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

```
add-child(ranked, v, w):
if w.child = null:
{   w.child = v
    v.next = v }
else:
{   v.next = w.child.next
    w.child.next = v
    if not ranked: w.child = v   }
```

Figure 6: Implementations of auxiliary methods in eager hollow heaps.

As in lazy hollow heaps, we can store keys with items instead of nodes. Two alternatives to making lists of children circular that simplify linking are to make each list of children singly linked, with pointers to the first and last child, or to maintain separate singly linked lists of ranked and unranked children for each node. The latter requires implementing *delete* as in Section 6, keeping track of two lists of nodes. These alternatives use an extra pointer per node. An alternative that does not take extra pointers is to avoid unranked links altogether and use a multi-tree representation, as in the original version of Fibonacci heaps. We represent a heap by a singly linked circular list of full roots of its trees, with a pointer to a root of minimum key. Details are analogous to those of Fibonacci heaps.

A final alternative that does not need extra pointers is to use the flexibility of hollow nodes to maintain the invariant that every root has rank 0, except in the middle of a *delete*. Once a node wins a ranked link, it participates only in ranked links, not unranked ones. This guarantees that all its ranked children are at the front of its list of children, in decreasing order by rank. To maintain the invariant during *decrease-key*(e, k, h), if $k < h.key$, move the item in h into a new node v of rank 0, make v a child of h, and move e into h. To maintain the invariant during *delete*, once all roots are full and have different ranks, if there is more than one, create a new node v of rank 0, make all the old roots children of v, and move the item in a child of v of smallest key into v. This takes one extra node per *delete*.

8 Good and Bad Variants

In this section we explore the design space of hollow heaps. We show that lazy and eager hollow heaps occupy "sweet spots" in the design space: although small changes to these data structures preserve their efficiency, larger changes destroy it. We consider three classes of data structures: *lazy-k*, *eager-k*, and *naïve-k*. Here k is an integer parameter specifying the rank of the new node v in a *decrease-key* operation. In specifying k we use r to denote the rank of the node u made hollow by the *decrease-key* operation, i.e., initially $v = u.sp$. Data structure *lazy-k* is the data structure of Section 2, except that it sets the rank of v in *decrease-key* to be $\max\{k, 0\}$. Thus *lazy*-$(r-2)$ is

```
delete(e, h):
{   e.node.item = null
    e.node = null
    if h.item ≠ null: return h
    max-rank = 0
    h.next = null
    while h ≠ null:
    {   u = h
        h = h.next
        if u.item = null:
        {   if u.child ≠ null:
            {   x = u.child.next
                u.child.next = h
                h = x  }
            destroy u  }
        else:
        {   while A[u.rank] ≠ null:
            {   u = link(true, u, A[u.rank])
                A[u.rank] = null
                u.rank = u.rank + 1  }
            A[u.rank] = u
            if u.rank > max-rank: max-rank = u.rank  }  }
    for i = 0 to max-rank:
    {   if A[i] ≠ null:
        {   if h = null: h = A[i]
            else: h = link(false, h, A[i])
            A[i] = null  }  }
    return h }
```

Figure 7: Implementation of *delete* in eager hollow heaps.

exactly the data structure of Section 2. Data structure *eager*-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, if $r > k$, it moves to v all but the $r-k$ highest ranked children of u, including all the unranked children of u. Thus *eager*-$(r-2)$ is exactly the data structure of Section 4. Finally, *naïve*-k is the data structure of Section 2, 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, whether or not u has a second parent. 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 *lazy*-$(r-j)$ and *eager*-$(r-j)$ 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 (Theorems 3.4 and 4.3) 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 by the following theorem. The individual constructions appear in sections 8.1, 8.2, and 8.3.

Theorem 8.1 *Variants lazy-$(r-j)$ and eager-$(r-j)$ are efficient for any choice of $j > 1$ fixed independent of r. All other variants, namely naïve-k for all k, eager-r, lazy-r, eager-$(r-1)$, lazy-$(r-1)$, and eager-k and lazy-k for k in the small regime are inefficient.*

8.1 Constructions for *eager*-k and *naïve*-k

We first consider *eager*-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_n [3, 23] inductively: B_0 is a one-node tree; B_{n+1} is formed by linking the roots of two copies of B_n. Tree B_n consists of a root whose children are the roots of copies of $B_0, B_1, \ldots, B_{n-1}$ [3, 23]. For any n, build a B_n by beginning with an empty tree and doing $2^n + 1$ 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^n children of rank 0. In the *delete-min*, all the links will be ranked, and they will produce a copy of B_n in which each node that is the root of a copy of B_j has rank j. The tree B_n is shown at the top of Figure 8.

Now repeat the following operations 2^n times: do n *decrease-key* operations on the items in the children of the root of B_n, making the new keys greater than that of the key of the item in the root. This makes the n previous children of the root hollow, and gives the root n 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 n 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 $1 + n + \sum_{j=0}^{n-1} f(j) > (n/2)f(n/2)$. Each node involved in linking is the root of a binomial tree. Since the total number of nodes remains 2^n, the binomial trees are linked using only ranked links to form a new copy of B_n, and the process is then repeated.

After B_n is formed, each round of $n+2$ consecutive subsequent operations contains only one *delete-min* but takes $\Theta(nf(n/2))$ time for n even. The total number of operations is $m = O(n2^n)$,

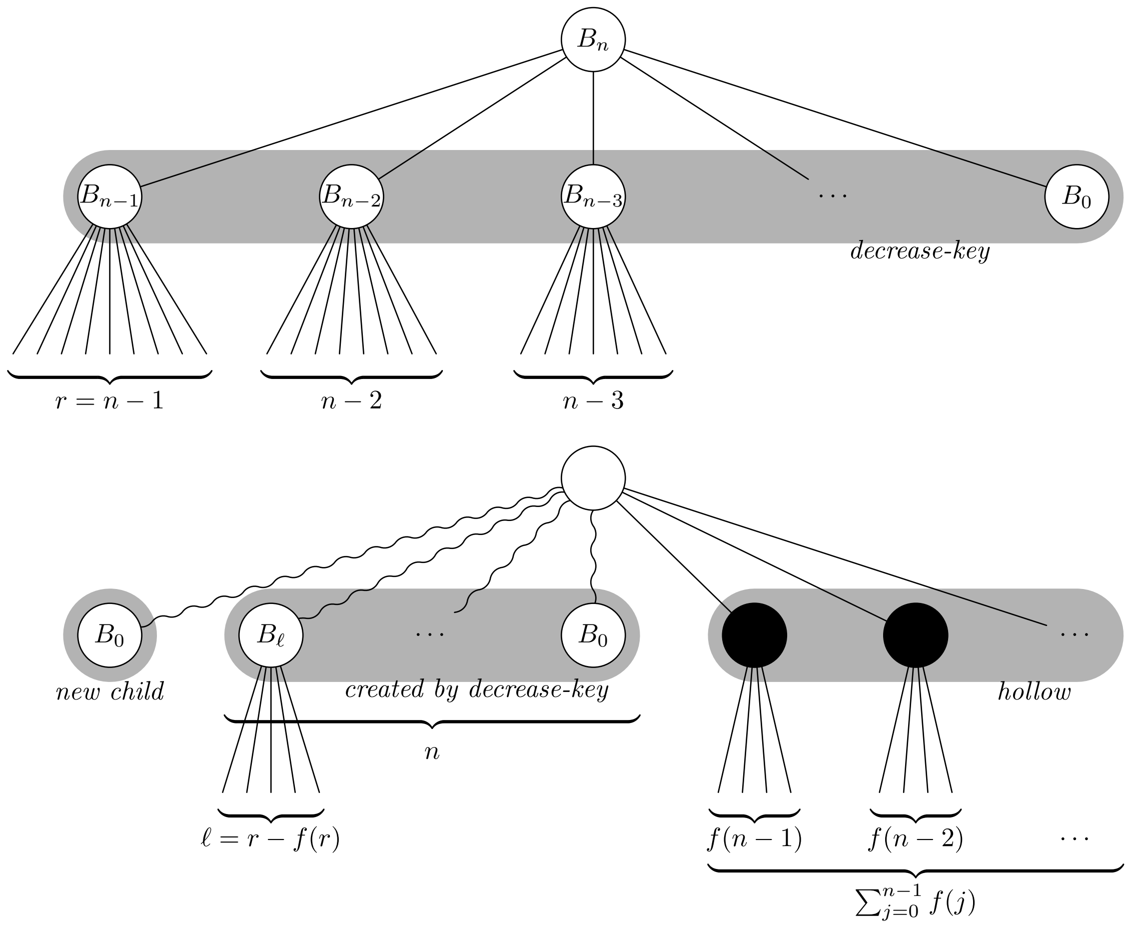

Figure 8: The construction for *eager-k*. Roots of binomial trees are labelled, and black nodes are hollow. Solid and squigly lines denote ranked and unranked links, respectively. (Top) The initial configuration; a binomial tree B_n. The shaded region shows which *decrease-key* operations are performed. (Bottom) The heap after performing *decrease-key* operations and inserting a new child. The keys of the 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.

of which $2^n + 1$ are *delete-min* operations. The total time for the operations is $\Theta(n2^n f(n/2)) = \Theta(mf(n/2))$, but the desired time is $O(n2^n) = O(m)$. In particular, if the amortized time per *delete-min* is $O(n)$ and the amortized time per *make-heap* and *insert* is $O(1)$, then the amortized time per decrease-key is $\Omega(f(n/2))$, which tends to infinity with n.

We next consider *naïve-k*. Note that *naïve*-0 is identical to *eager*-0, so the two heaps do exactly the same thing for the above example. An extension of the construction shows that *naïve-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. *naïve-k* is identical to *naïve*-0 except that nodes created by *decrease-key* may not have rank 0. The construction for *naïve-k* deals with this issue by inserting new nodes with rank 0 that serve the function of nodes created by *decrease-key* for *naïve*-0. The additional nodes with non-zero rank are linked such that they don't affect the construction.

We build an initial B_n as before. Then we do n *decrease-key* operations on the items in the children of the root, followed by $n+1$ *insert* operations of items with keys greater than that of the root, followed by one *delete-min* operation, and repeat these operations 2^n 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_n 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_n, each round of $2n+2$ subsequent operations maintains the invariant that the tree consists of a copy of B_n with additional children of its root, whose descendants are all the nodes added by *decrease-key* operations.

The analysis is the same as for *eager*-0, i.e., $f(r) = r$. The total number of operations is $m = O(n2^n)$ and the desired time is $O(n2^n) = O(m)$. The total time for the operations is however $\Theta(n^2 2^n) = \Theta(mn)$. Thus, the construction shows that *naïve-k* for any value of k takes at least logarithmic amortized time per *decrease-key*.

8.2 Constructions for *lazy*-r, *eager*-r, *lazy*-$(r-1)$, and *eager*-$(r-1)$

The idea is to construct a tree T_n with a full root, having full children of ranks $0, 1, \ldots, n-1$, and in which all other nodes, if any, are hollow. Then we can repeatedly do an *insert* followed by a *delete-min*, each round taking $\Omega(n)$ time. One set of bad examples suffices for *lazy*-r and *eager*-r, another for *lazy*-$(r-1)$ and *eager*-$(r-1)$.

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

Consider *lazy*-r and *eager*-r. Given a copy of T_n in which the root has rank n, we can build a copy of T_{n+1} in which the root has rank $n+1$ 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 n, inclusive, and two full children of rank 0. In particular one of these children is the old root which has rank n. The *delete-min* produces a copy of T_{n+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_n for an arbitrary value of n in $O(n^2)$ operations. These operations followed by n^2 rounds of an *insert* followed by a *delete-min* form a sequence of $m = O(n^2)$ operations that take $\Omega(n^3) = \Omega(m^{3/2})$ time.

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

Given a copy of $T_n(i)$ with $i > 0$, we can build a copy of $T_n(i-1)$ 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 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. Unlike the situation shown in the figure, the descendants of

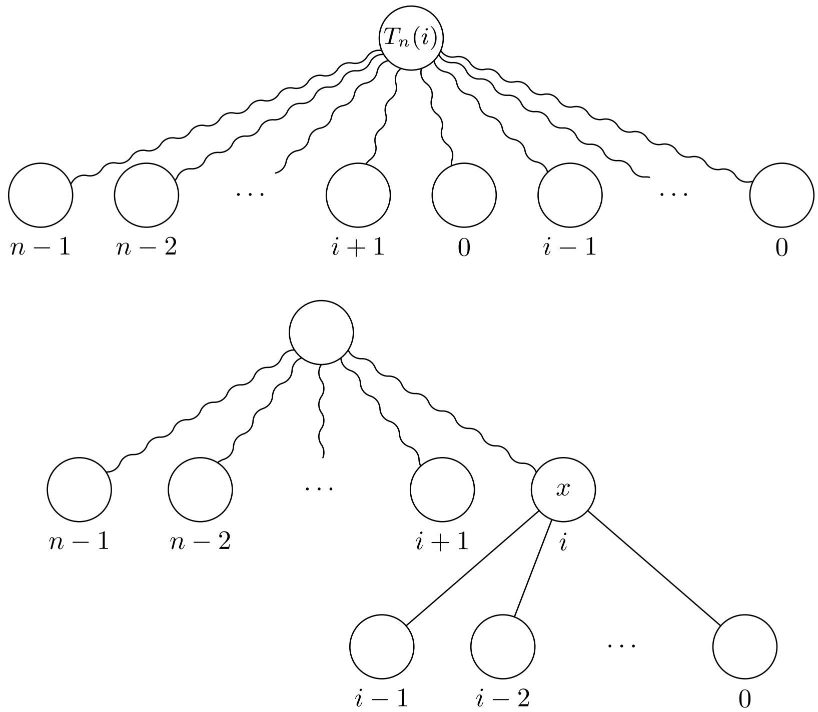

Figure 9: The construction for *lazy*-$(r-1)$ and *eager*-$(r-1)$. Only full nodes are shown. Solid and squigly lines denote ranked and unranked links, respectively. Ranks are shown beneath nodes. (Top) The tree $T_n(i)$. (Bottom) The tree obtained from $T_n(i)$ by inserting an item and performing a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

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 $T_n(i-1)$, with some extra hollow nodes, which we ignore. We can convert a copy of $T_n(0) = T_n$ into a copy of $T_{n+1}(n+1)$ 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_n in $m = O(n^3)$ operations. These operations followed by n^3 repetitions of an *insert* followed by a *delete-min* take a total of $\Omega(n^4) = \Omega(m^{4/3})$ time but the desired time is $O(n^3 \log n) = O(m \log m)$.

8.3 Construction for *lazy*-k

Finally, we consider *lazy*-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 *lazy*-0 and then show how to generalize the construction to all choices of k in the small regime.

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

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

We build $Q_0, Q_1, \ldots, Q_n$ 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 *eager*-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, such that the root remains the same throughout the sequence of operations. We first 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 with 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. 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 nodes in the *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.

Suppose we are given a copy of R_n. Let x_j, for $j = 0, 1, \ldots, n$, be the root of the copy of S_n. In particular, x_n is the root of R_n. We can do an expensive round of operations that produces a new copy of R_n as follows: Do *decrease-key* operations on x_j for $j = 0, 1, \ldots, n-1$, producing a second parent y_j for each x_j. Make all the new keys larger than that of x_n and smaller than those of all children of x_n; among them, make the key of y_{n-1} the smallest. Next, insert a new item with key greater than that of y_{n-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 $j = 0, \ldots, n-1$. Once the hollow nodes are deleted, the remaining roots are z, the y_j, and the roots of $n - i$ copies of B_i for $i = 0, 1, \ldots, n-1$. 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 $S_0, S_1, \ldots, S_n$ (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_n. The round of operations consists of one *insert*, n *decrease-key* operations, and one *delete-min* and takes $O(n^2)$ time.

The number of nodes in R_n is $O(2^n)$, as is the number of operations needed to build it and the time these operations take. If we then do 2^n rounds of operations as above, the total number of operations is $m = O(n2^n)$. The operations take $\Theta(n^2 2^n) = \Theta(mn)$ time, whereas the desired time is $O(m)$.

An extension of the same construction shows the inefficiency of *lazy*-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 $k/f(\sqrt{k}) + \sqrt{k}$. So the extended construction does at most $n^2/f(\sqrt{n}) + n^{3/2}$ *decrease-key* operations per round, and the amortized time per decrease-key is $\Omega(f(\sqrt{n})) = \omega(1)$, assuming that the amortized time per *delete* is $O(n)$ and that of *make-heap* and *insert* is $O(1)$.

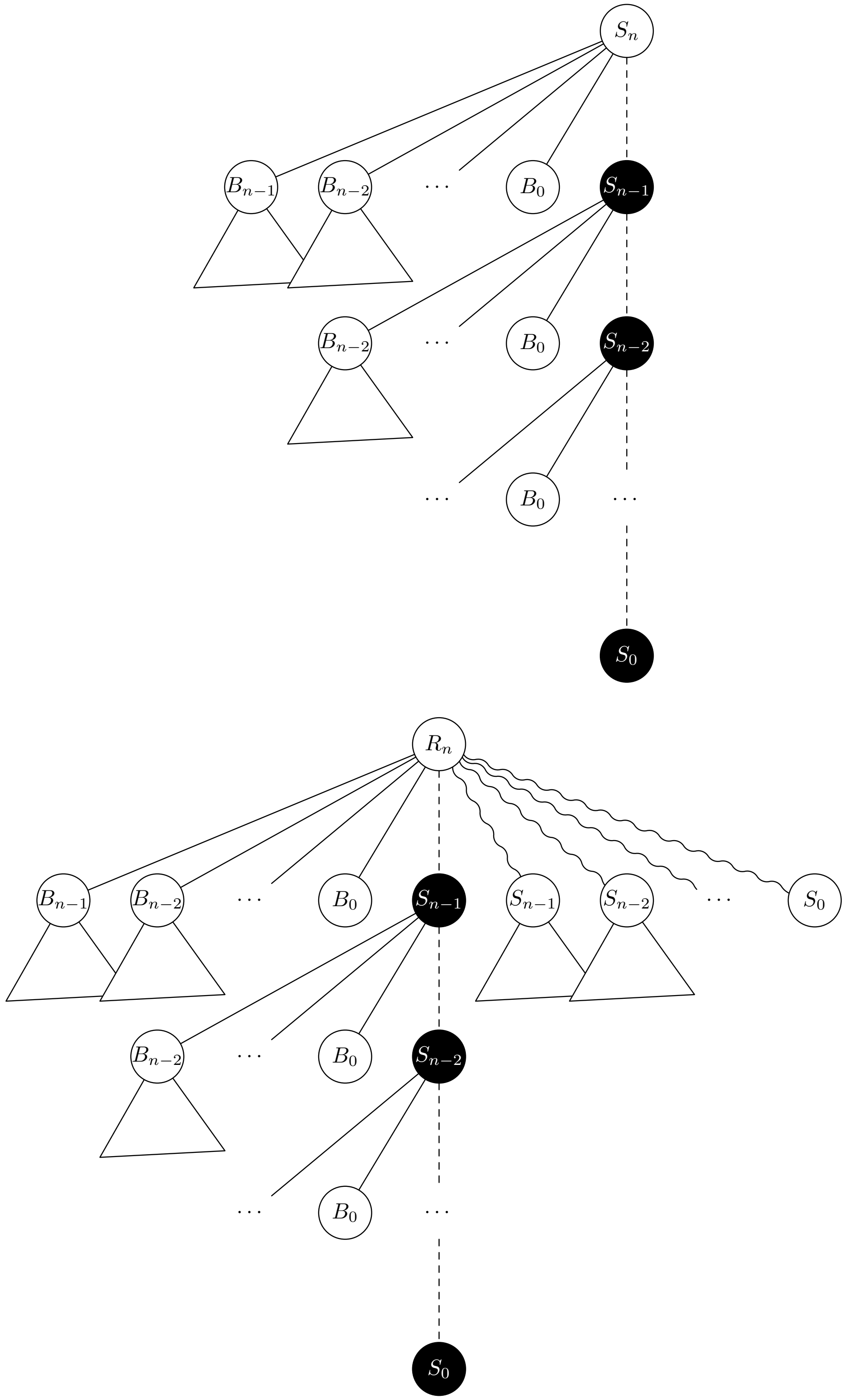

Figure 10: The trees S_n (top) and R_n (bottom). Every node is labelled by the type of subtree it is the root of. The triangles symbolize such subtrees. Black nodes are hollow. Solid and squigly lines denote ranked and unranked links, respectively. Dashed lines denote unranked links made by one node becoming the second parent of another node.

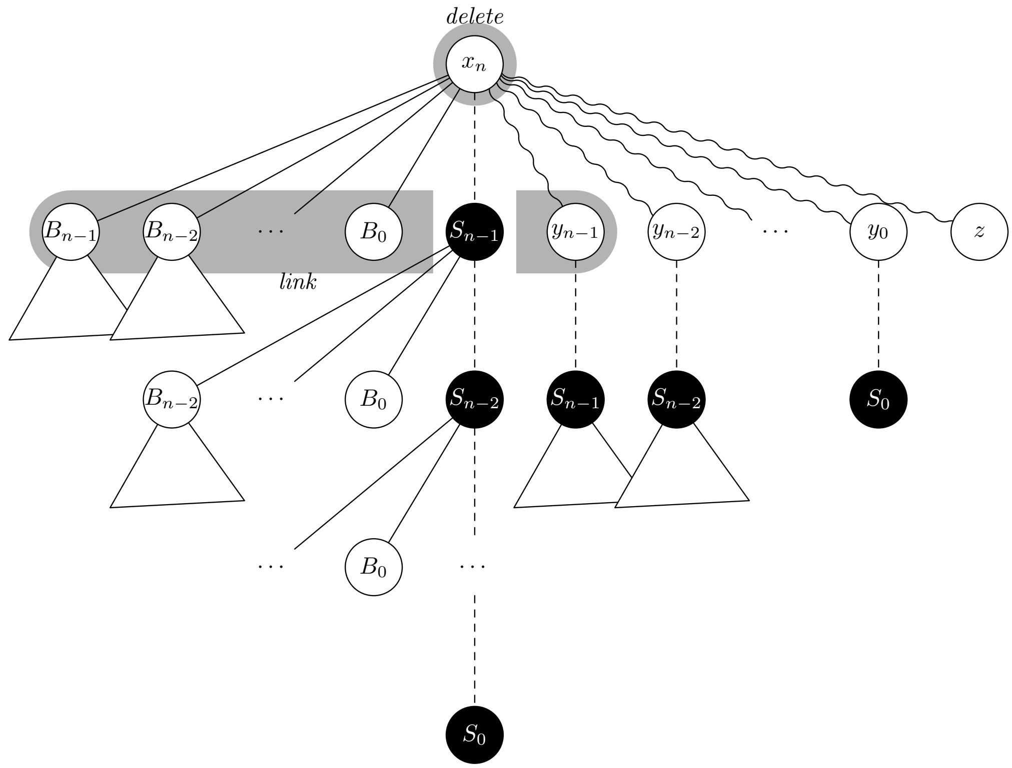

Figure 11: The tree obtained from R_n by performing n *decrease-key* operations and one *insert*. Every node is either labelled by its name or the type of subtree it is the root of. The triangles symbolize such subtrees. Black nodes are hollow. Solid and squigly lines denote ranked and unranked links, respectively. Dashed lines denote unranked links made by one node becoming the second parent of another node. Unranked links from children of y_i, for $i = 0, \ldots, n-1$, to x_n have been omitted.

9 Remarks

We have presented the hollow heap, a new data structure with the same amortized efficiency as Fibonacci heaps but with several advantages, including fewer pointers per node and $O(1)$-time *decrease-key* operations in the worst case. Hollow heaps obtain their efficiency by careful use of lazy deletion.

There are several ways to change the implementation of hollow heaps and related data structures that we mention below. Whether any of these changes is useful is a question for experiments.

The implementation of eager hollow heaps in Section 7 maintains sibling lists so that ranked children always precede unranked children, and ranked children are in decreasing order by rank. Instead of maintaining one list of children per node, one could maintain separate lists of ranked and unranked children. This costs one extra pointer per node but may make unranked links slightly faster.

It is not necessary to do as many ranked links as possible. It suffices to find a maximum matching of roots of the same rank, link the matched roots by ranked links, and link the remaining roots by unranked links. See [11].

As in Fibonacci heaps, it suffices to do ranked links only. A heap is represented by a set of trees instead of a single tree, with a pointer to a root of minimum key. When deleting the item in a root, ranked links are done until no more are possible; then the remaining roots are scanned to find one of minimum key. As mentioned in Section 7, using this idea in eager hollow heaps allows one to represent each set of children by a non-circular singly linked list, which simplifies linking.

An extension of this idea is to do all the links in *find-min* instead of in *delete*. Again a heap is represented by a set of trees rather than a single tree, but with no pointer to a root of minimum key. A delete operation merely deletes the appropriate item, making the node holding it hollow. A find-min deletes hollow roots and does links until there is only one root, a full one, or does ranked links until none is possible.

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

[10] 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.

[11] B. Haeupler, S. Sen, and R.E. Tarjan. Rank-pairing heaps. pages 659–670, 2009.

[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] 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.

[15] H. Kaplan, R. E. Tarjan, and U. Zwick. Fibonacci heaps revisited. Technical report, 2014. arXiv:1407.5750.

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