Hollow Heaps

CS166 Final Project Checkpoint

Martin, Eric Mark
Department of Computer Science
Stanford University
ericmarkmartin@stanford.edu

Murphy, Dana
Department of Computer Science
Stanford University
dkm0713@stanford.edu

Perez, Luis Antonio
Department of Computer Science
Stanford University
luis0@stanford.edu

2019–05–16

Abstract

We provide an overview of hollow heaps, a *simpler* amortized data structure that achieves the same run-time as traditional Fibonacci heaps. We also provide detailed answers to the theoretical differences between hollow heap variants (one-parent, two-parent, one-root, multi-root, etc.), as well as the an insightful analysis of the process of hollowing nodes and its affect on run-time.

Lastly, in this paper, we present the novel idea of node 'resurrection', and analyze its properties.

1 Back to Basics

1.1 Introducing Priority Heaps

Priority heaps are a widely used abstract data structure, with multiple differing implementations with different run-time guarantees. In our discussion, we restrict our definition of priority-heaps in general, and hollow-heaps specifically, to the following commonly used operations:

- `MakeHeap`$\rightarrow Q$ - constructs a new, empty heap Q
- `Meld`$(h_1, h_2) \rightarrow h_3$: Returns a new heap h_3 containing all items from heaps h_1 and h_2.
- `Insert`$(Q, x, k) \rightarrow \bar{x}$ - insert an item x with key k into heap Q and return a reference $\bar{x}$ to x.
- `DeleteMin`$(Q) \rightarrow k$ - remove the minimum item from the heap Q and return the key k associated with it
- `DecreaseKey`$(Q, \bar{x}, k') \rightarrow$ `None` - given a handle $\bar{x}$, change the key of item x contained in the heap Q to be k'. It is guaranteed that $k' \leq k$, the original key.

With the above operations, implementations of priority heaps can be used for widely popular algorithms, such as Prim's algorithm for finding the MST of a graph, as well ad Dijkstra's shortest path algorithm. As such, finding an efficient way to perform the above operations can have a wide impact.

For the remaining of this paper, we assume that reader are familiar with heaps, specifically Fibonacci heaps.

2 Introducing Hollow Heaps

Hollow heaps are an alternative implementation of priority heaps, and can be thought of as a a lazy version of Fibonacci heaps. Their main claim to fame is that they provide the similar amortized efficiency as the classical Fibonacci heap, but are *simpler* to implement. They rely on lazy deletions and re-insertions to perform `DecreaseKey` operations. There are two critical aspects to hollow heaps that give them this simplicity while maintaining good amortized run-times.

1. **Lazy Deletion**: This is in-fact where Hollow Heaps get their name. Rather than immediately remove a node from the tree when the element is deleted or modified (e.g, due to an `DeleteMin` or `DecreateKey` operation, nodes are instead retained but "hollowed" (made empty). These hollow nodes are eventually removed in bulk, while maintaining balance in the data structure.
2. **Directed Cyclic Graphs**: Hollow Heaps can readily be extended to two-parent versions, making the tree (or trees) into DAGs instead. This representation does not fundamentally change the data structure, but does delay the re-arranging of children when a new hollow node is created.

3 Hollow Heaps in Detail

Conceptually, it'll be helpful if we introduce Hollow Heaps in steps. These steps are simple modifications to Fibonacci heaps which aim to simplify the structure while maintaining the same efficiency. There are a total of three modifications, each presented below in detail.

To begin, consider at first Hollow heaps to be the same as Fibonacci heaps.

3.0.1 What stays the same?

Most of the operations stay the same as a traditional Fibonacci heap, with the biggest difference occuring in the `DecreaseKey` and `DeleteMin` operations.

3.0.2 Replacing Cascading Cuts with Hollow Nodes

Fibonacci heaps are amortized efficient primarily due to their use of cascading cuts to prevent the forest of trees from becoming too unbalanced. Each `DecreaseKey` operation introduces imperfections into the tree, but by making sure that the cut leads to trees that are about the same size, Fibonacci heaps can maintain a relatively ordered forest, leading to fast `Insert`'s and `DeleteMin` operations.

Hollow heaps instead introduce the concept of *hollow* nodes. In normal Fibonacci heaps, each node represents an item (in one extreme case, the keys themselves can be the values). However, Hollow heaps require an extra level of indirection, a node can either be **full** or **hollow**. **Full** nodes are nodes with a key which point to a stored value, while **hollow** nodes are nodes with a key which have no value.

Note that this relaxation on the one-node, one-value invariant is not free. Hollow heaps must necessarily be *exogenous* rather than *endogenous* – nodes must *hold* items rather than *be* items.

Now, when decreasing the key of an existing node n, rather than cutting the tree rooted at n and then performing a set of cascading cuts as is the case with Fibonacci heaps, simply create a new node which is a duplicate of n, and move all subtrees rooted at n except the two largest subtrees (the ones with rank $r-1$ and $r-2$) to be children of the newly created node. The rank of the newly created node is $r-2$.

This process maintains the invariant that a node of rank r has at least $F_{r+3}-1$ nodes, both full and hollow. Note that this further implies that the rank of a node in a hollow heap of N nodes is at most $\log_\phi N$, which means that if our multi-root hollow heap has N nodes. The critical aspect of the data structure at this point is maintaining the following invariant:

4 Insightful Questions Answered

4.1 Question 1

There are several different ways to represent hollow heaps (two-parent, one-parent, etc.). What are the advantages and disadvantages of each?

Generally speaking, there are a few major parameters that can be tweaked for hollow heaps.

4.1.1 One-root vs multi-root

The main difference between these two variants is the existence of *unranked* links within the tree. In the multi-root version, we maintain a forest of trees while in the one-root version, we allow the linking of these trees' root through unranked links.

This linking essentially allows use to save the results of comparisons that would normally be undertaken after extracting the minimum value from a tree. This gives us the advantage of theoretically increasing performance by a constant factor. We also hypothesize that the implementation of a single-root heap might be more cache-friendly, since more children nodes are linked to each other. However, this would ultimately depend on the implementation.

However, the disadvantage of this single-root heap is that it adds complexity to the algorithm and implementation of the data structure. While it's possible to maintain the ranked/unranked links without any additional state, it's nonetheless a conceptual complexity added to the data structure.

4.1.2 One-parent vs two-parent

Another interesting variant of hollow heaps, and a vastly more drashtic change, is the move from tree (or single tree) to a forest of DAGs (or single DAG).

The motivation for this modification to the data structure is to reduce the movement of children during `DecreaseKey` operations. Instead of moving children from the newly hollowed node u to a new node v, we instead make v an additional parent of u. This elimination of child movement not only decreases the number of operations that must be performed during a `DecreaseKey` step, but we also post-pone when the children become roots. To see this, note that when a hollow node is destroyed, all children become roots (or are themselves destroyed). In the one-parent version, this happens when the single parent is destroyed. However, in the two-parent version, this process will only occur when both parents are destroyed, thereby delaying it.

However, the two-parent hollow heap does have some disadvantages. It is a more complex data structure, and it's analysis for run-time complexity relies on a more complex proof which requires a transformation from two-parent heaps to one-parent heaps. Lastly, the rebuilding aspect of hollow heaps, which is what decreases the amortized run-time from $\log N$ to $\log n$ is not described in the paper, and it is still an open question whether such a rebuilding can be performed on two-parent trees.

4.2 Question 2

Why are hollow nodes constrained to have exactly two children of the indicated ranks? What happens if we relax this requirement (allow hollow nodes to have an unbounded number of children) or tighten it (force hollow nodes to have exactly one child?)

4.2.1 Hollow heaps with two children

When a node is made hollow, the children of that node are moved such that they maintain the rank invariant, which states that "A node u of rank r has exactly r children, of ranks $0, 1, ..., r-1$, 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 $r-1$." This means that hollow nodes can have at most two children. One consequence of this is that the rank bound of a node in hollow heap of N nodes (for either multi-root hollow heaps or one-root hollow heaps) is at most $\log_\phi N$.

These properties lead to an amortized cost that is as efficient as a Fibonacci heap. To see this, we can create a potential function for the amortized cost of operations in a hollow heap, such that the potential is the number of full nodes that are not a child of another full node. The amortized cost of an operation to be the number of links plus the increase in potential. The cost per insert is one (one new root), zero for find-min and meld, And three for decrease-key (one new root and at most two full children of a newly hollowed node). The delete function becomes the cost of every other function, plus the cost of removing an item. By the rank bound given above, removing an item is at most an increase of $\log_\phi N$ in potential. Therefore, the amortized cost of delete is $\log_\phi N$. Our final amortized runtimes are $O(1)$ for everything except delete, which is $O(\log_\phi N)$.

Interestingly, hollow nodes can have any fixed number of children $j \geq 2$ and still have that same asymptotic efficiency as Fibonacci heaps. This is because as j increases, the rank bound decreases by a constant factor. One consequence of this is that, because the amortized cost of decrease-key is one new root plus at most the number of children of a new hollow node, the cost of decrease-key will increase linearly with j with an amortized cost of $O(j + 1)$.

However, any other variant in the number of children for hollow nodes will be less efficient than Fibonacci heaps. Generally speaking, this is because for any other variant, we can provide a series of operations where the amortized cost of decrease-key is not constant. We will explore two of these variants, hollow nodes with unbounded children and hollow nodes with only one child, in more detail.

(Note that these examples are specific to 1-parent, 1-root hollow heaps. However, a similar argument can be applied to multi-root and 2-parent hollow heaps.)

4.2.2 Hollow heaps with unbounded children

First, we will consider the case of hollow nodes having unbounded children. This is a specific instance of a class of variants where hollow nodes can have at most $f(r)$ children, where f is a positive non-decreasing function that tends to infinity. In this case, $f(r) = r$. To do this, we create a 'B_ℓ' binomial tree (for some arbitrary ℓ) by doing $2^\ell + 1$ insertions of items with increasing key value. This will result in a tree with one root with 2ℓ children of rank 0. We then delete the minimum value; all the links will be ranked, and will create a copy of B_ℓ, and every node which is a root of the copy will have a rank of j.

Next, do ℓ decrease-key operations on ℓ children of the root in B_ℓ (keeping the new key greater than the root so it doesn't replace the root). This will make the original ℓ children hollow and create ℓ new children. Insert a new item with a minimum key, such that it will be the new root. Finally, perform a delete-minimum, deleting the root and its ℓ hollow children, resulting in the children of the hollow nodes being linked (we will adversarially link newly inserted links with grandchildren of the deleted node when possible).

Repeat this process 2^ℓ times. The total number of operations is $O(2^\ell \ell)$. The total amortized runtime is $O(2^\ell \ell^2)$ for $O(2^\ell \ell)$ operations, with decrease-key being lower-bounded by $\Omega(\ell)$, which is greater than the $O(1)$ amortized cost of a Fibonacci heap. Thus, hollow heaps with unbounded children for hollow nodes is not as efficient as a 2-children version of hollow heaps.

4.2.3 Hollow heaps with one child

Next, we will consider the case of restricting the number of children in hollow nodes to one child. In this case, we create a tree, T_ℓ, that has a full root and full children of ranks 0 to $\ell - 1$. We can then repeatedly insert an element and delete the minimum, taking $\Omega(\ell)$ time.

Next, suppose we have a tree $T_\ell(i)$, which is the same as a T_ℓ tree except that the ith child has a rank 0. We can convert this tree into a $T_\ell(i-1)$ tree. To do so, insert an item larger than the root but smaller than all other items. Delete the minimum, making the newly inserted node the root of the tree. All children of the old root with a rank less than i will become children of a new tree of root x, where x has a rank i and is a child of the new root. All children with a rank greater than i will become children of the new root. Next, perform a decrease-key on all of the full descendants of x, starting from the leaves and working upwards, making sure each new key is still greater than that of the root. Each new node has a rank 1 less than the rank it came from, except the node that was already rank 0, meaning the root gets two new children of rank 0 and 1 new child for the ranks 1 to $i - 1$. This is equivalent to a $T_\ell(i-1)$ tree.

We can additionally convert T_ℓ into a copy of $T_{\ell+1}(\ell+1)$ by inserting a new non-minimum item. Combining this fact with the fact above, we can show that we can create a copy of T_ℓ in $O(\ell^3)$ operations (ℓ^3 insertions plus one delete-min). These operations take a total of $\Omega(\ell^4)$ work, which is greater than the $O(1)$ amortized work per operation of a Fibonacci heap. Thus, hollow heaps with one child per hollow node is not as efficient as a 2-children version of hollow heaps.

5 Interesting Extension Proposal

The analogy between hallowing within hallow heaps and lazy deletion within structures that support it (such as RAVEL trees) is clear. Both involve the removal of data from the set contained by the structure without the removal of the structural housing of that data, but rather the marking of it as inactive. As such, we found the lack of mention of this connection in this paper introducing them odd. For our interesting component, we propose an investigation of this connection, primarily focusing on the possibility of re-adding a previously deleted element to a hallow heap via "unhollowing."

Some implications of such an operation are immediately obvious. For one, unhollow (full?) nodes have invariants dictating the quantity and contents of their children—invariants which are relaxed and violated when a node hallowed. Re-filing in a node is immediately problematic because it creates invariant violations in the node that is filled. Additionally, it seems as though an adversarially chosen sequence of operations could abuse the re-filling in of nodes to create a particularly inefficiently structured hallow heap.

It seems like these structural violations might be small individually. If we could find a way to clean up these invariant breaches occasionally or on the invocation of some other heap routines, the additional cost imposed by the structural violations created by filling in nodes might amortize away.

One of the greatest shortcomings of hallow heaps is that operations pay costs on the number of hallow nodes even though they contain no actual information in the data structure. If we are able to find a way to refill nodes without paying enormous runtime costs, it could help mitigate this weakness by quickly eliminating these troublesome hallow nodes, converting them into useful full nodes that do not add dead weight to the hallow heap. Even if refilling in nodes is not viable, there are other interesting implications of viewing hallow nodes as lazily deleted nodes, such as using the proportion of hallow nodes in the heap as a trigger for rebuilding it.

Viewing hallow nodes through this lens also has the potential to invite new ways of deriving and interpreting results in the hallow heap, such as its asymptotic run time complexity bounds.