Resolving a Question about Randomized Fibonacci Heaps in the Negative

John Peebles[*]

December 14, 2013

Abstract

In this project, we study a randomized variant of Fibonacci heaps where instead of using mark bits, one flips coins in order to determine whether to cascade bringing nodes into the root list. Although it seems intuitive that such heaps should have the same expected performance as standard Fibonacci heaps—and Karger has conjectured as such—the only previous work was an $O(\log^2 s)$ upper bound using a special potential function where s is the number of operations on the heap requested so far. We first match this bound using an (apparently) different proof that reduces the analysis to the non-randomized case with high probability via a union bound.

We then prove that randomized Fibonacci heaps, as originally defined, perform worse asymptotically then standard Fibonacci heaps. Specifically, while they match the amortized time bounds for every operation except delete-min, they can take $O(\sqrt{n})$ amortized time for a single delete-min operation if an exponentially large (in n) number of heap operations are requested.

A small modification can be used to bypass this issue, but we then show that even this modification fails by giving a request sequence for randomized Fibonacci heaps with $\omega(\log s)$ expected average cost per delete-min.

1 Introduction

A *Fibonacci heap* is a type of data structure which maintains a priority queue. It supports the following operations with the given amortized time bounds:

operation	amortized runtime
insert	$O(1)$
decrease-key	$O(1)$
merge	$O(1)$
delete-min	$O(\log n)$

[*]jpeebles@mit.edu

where n is the number of elements in the data structure.

Fibonacci heaps were developed by Fredman and Tarjan in [FT87]. In their paper, they leave the open question of whether there is a "version of F-heaps that achieves the same amortized time bounds, but requires neither maintaining ranks nor cascading cut." While there are at least nine other data-structures that support essentially this set of operations in the same amortized time bounds, it is arguable that none of them is really a "version of F-heaps." (See [Cha13] for a good list of these data-structures.)

One potential route to achieving this is via the use of randomization in the data structure. A reasonable starting place is the elimination of the need to keep mark bits by flipping coins instead of looking at mark bits.

Karger has conjectured that the following randomized variant of Fibonacci heaps meets the same time bounds as standard Fibonaccie heaps [Kar13]. Recall that when a Fibonacci heap performs a decrease-key operation, it moves a node to the root list, checks its parent's mark bit, and recurses on the parent if the mark bit is set. In Karger's randomized variant, the only difference is that we no longer use mark bits. When we need to decide whether to continue by moving the parent to the root list, we flip a coin instead of using mark bits. (The probability on heads is simply 1/2.)

We will refer to the non-randomized version of Fibonacci heaps developed by Fredman-Tarjan as *standard Fibonacci heaps* and the randomized version proposed by Karger as *randomized Fibonacci heaps*. When context is clear, we may refer to either one simply as a *Fibonacci heap*, *F-heap*, or *heap*.

While there is a considerable amount of prior *effort* on this problem—given that virtually every CS theory student to pass through MIT in over a decade has seen it—there is little in the way of prior results, probably because the problem is impossible. All we are aware of is a result by Eric Price and/or David Karger which we have not seen. This result apparently shows that randomized Fibonacci heaps obtain $O(\log^2 s)$ expected amortized time—where s is the number of operations requested so far—for delete-min and $O(1)$ expected amortized time for all other operations. We are told that this proof uses a special potential function designed for randomized Fibonacci heaps.

In Section 2, we prove the same upper bound using only the standard potential function used in the analysis of standard Fibonacci heaps. Our analysis uses a union bound to reduce the case of non-randomized Fibonacci heaps with high probability, giving a rather straightforward analysis.

In Section 3, we prove that randomized Fibonacci heaps—as they are defined by Karger—do not have the same expected amortized time bounds as Fibonacci heaps. We do so by illustrating a request sequence in which the expected average cost of a delete-min is $O(\sqrt{n})$.

In Section 4 show how to modify the heap get around this lower bound.

In Section 5, we give another lower bound which shows that neither the original version of Karger's randomized Fibonacci heaps, nor the modified version, achieves the same expected amortized asymptotic performance as standard Fibonacci heaps.

Finally, we give possible directions for future work in Section 6.

2 An $O(\log^2 s)$ Upper Bound[1]

In this section, we prove that the expected amortized cost of a delete-min in randomized Fibonacci heaps is $O(\log^2 s)$ and that the expected amortized cost of all other operations is $O(1)$. To do this, we first upper-bound the probability that a node has lost a lot of its children since the last time it was in the root list.

Say a non-root node v in a Fibonacci heap is *missing* a child if the child was removed from the node in the course of a decrease-key operation since the last time v was in the root list.

Lemma 1. Suppose we have an empty randomized Fibonacci heap and we intend to perform s operations on it which will result in a heap of size n. Then the probability that every non-root node in the resulting heap is missing more than k children is at least $1 - ns2^{-k}$.

Proof. The proof idea is as follows. We can think of the request sequence as being generated by an adversary who wishes to make a bunch of non-root nodes that are each missing at least k of their children. In order to do this, the adversary will inevitably be required to cascade decrease-key operations up to the children it wants to remove. However, any decrease-key operation that gets rid of a child also has probability 1/2 of sending the parent to the root list as well. However, the adversary gets many tries to remove k children from a node, potentially up to one per delete-min operation. The probability of any try succeeding is 2^{-k} and the adversary gets at most s tries on each of the n nodes. A union bound over all tries on all nodes gives a probability of success for the adversary of no more than $ns2^{-k}$.

More formally, for any node v in the heap, let $p_{v,t}$ denote the probability that v just became a non-root node during operation t, [2] never returns to the root list after operation t, and is eventually missing at least k of its children after operation t.

A necessary condition for this event is that k of v's children get cascaded to the root after time t, but the cascade does not continue on to v. The probability of this necessary condition is at most 2^{-k}. Thus, $p_{v,t} \leq 2^{-k}$.

Then the probability p_v that node v is missing at least k children after s operations is $\sum_{i=1}^{s} p_{v,i} \leq s2^{-k}$. [3]

Taking a union bound over all nodes gives an upper bound of adversary success of $ns2^{-k}$.

$\square$

As a corollary, we get the following:

[1]The proof in this section is joint work with classmate Jerry Li from earlier in the semester. This is my own writeup of it, though.

[2]IE., v is a non-root node after operation t and either (1) there was no operation before operation t or (2)v was a root node after operation $t-1$.

[3]Note that an easy to make error here would be to think that one can replace the sum with a max statement in the above, justified by the intuition that we could condition on the last time node v becomes a non-root. However, the probability of the adversary succeeding is not independent of the event we are conditioning on, so we can't just replace the sum with a max. One can also see this by noting that doing so would ignore the fact that the adversary gets multiple attempts to remove k children. A final way of seeing that doing so doesn't work is that the result we could obtain using it would violate the lower bound in the next section.

Corollary 1. With probability at least $1 - 1/n$, no node in the heap described in the above lemma is missing more than $k = 2 \log n + \log s \leq 3 \log s$ children.

We also use the following lemma which was essentially proved on a homework assignment.[4]

Lemma 2. Suppose a rooted tree on n nodes has the property that no non-root node in the heap is missing more than k children. Then the root has $O(k \log n)$ children. □

Now we can prove the main theorem of this section.

Theorem 1. A randomized Fibonacci heap has expected amortized cost $O(1)$ for all operations except delete-min and $O(\log s \log n) \leq O(\log^2 s)$ for delete-min.

Proof. We use the same potential function as for standard Fibonacci heaps in which we take the number of elements in the root list. (The standard Fibonacci heap potential function also includes the number of mark bits added in, but this number is 0 for us because we use no mark bits.)

The insert and merge operations are implemented exactly the same way as in standard Fibonacci heaps, so they take the same $O(1)$ amortized time under the same potential function.

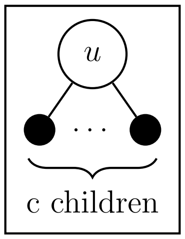
Note that the number of times we cascade in a decrease-key is $O(1)$, so the expected amortized cost of this operation is $O(1)$: $O(1)$ real plus $O(1)$ change in potential for each cascade.

Finally, the expected amortized cost of delete-min is simply $O(\log n)$ plus the number of children of the minimum node. By Lemma 2, this number is $k \log n$ where k is a bound on how many children are missing from any non-root node in the tree. If $k < 3 \log s$, this cost is $O(\log s \log n)$.

Furthermore, the probability that $k \geq 3 \log s$ is no more than $1/n$ by Corollary 1. So, the contribution of this case to the expected cost is $O(1)$. Thus, the total expected amortized cost of this operation is $O(\log s \log n) \leq O(\log^2 s)$. □

3 An $\Omega(\sqrt{n})$ Lower Bound

Lemma 3. For every c, u and $\epsilon > 0$, there exists a (very long) sequence of operations which results in the following heap with probability at least $1 - \epsilon$:



[4]The target audience is supposed to be members of this class, so I assume reproving it isn't necessary.

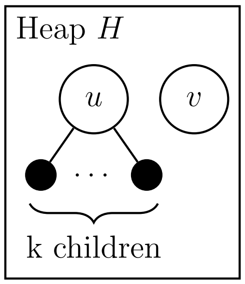
Proof. We proceed by induction on c.

Base Case: For $c = 0$, simply start with an empty heap and insert u. This results in the desired heap with probability 1.

Induction Hypothesis: Suppose that there exists a sequence of operations that can construct the heap shown with any particular desired root node value and $c = k$ children with probability at least $1 - \epsilon$ for every $\epsilon > 0$.

Now suppose consider the case of $c = k + 1$ children. We want to construct a heap like the one shown with root node u and $k+1$ children, succeeding with probability at least $1-\epsilon$ for any particular $\epsilon > 0$.

We do this as follows. Start by using the induction hypothesis to get a heap H which consists of a single root node u with k children and no other descendants, succeeding with probability at least $\sqrt{1-\epsilon}$. Now insert a node v with $v > u$. This results in the heap shown below (with probability at least $\sqrt{1-\epsilon}$).



Consider the following procedure which we will apply a larger number of times.

1. Add $2^k - 1$ nodes $s_1, \ldots, s_{2^k-1}$ such that $u < v < s_1 < s_2 < \ldots < s_{2^k-1}$.

2. Add a node t smaller than all other nodes in the heap and perform a delete-min. (This results in t being removed and the rest of the nodes being consolidated.)

3. For all $1 \leq i \leq 2^k - 1$, decrease the key of s_i to be minimum in the heap and delete-min, removing it. The order is arbitrary.

Given a heap H as shown in Figure 2, if we apply this procedure over and over again, the state of H after any particular application of the procedure is given by a *Markov Process* shown by the flowchart in Figure 1.

A more detailed step-by-step version of the flowchart is given in Figure 2.

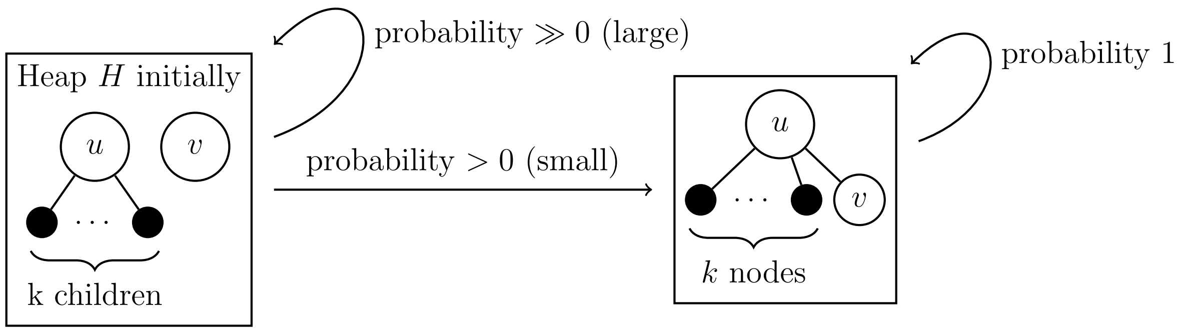


Figure 1: High-Level description of how one iteration of our procedure works. Each box represents a state of the heap and each arrow represents the probability of going from one state to the other after applying steps 1–3 once. After a large number of applications, we will get stuck in the state on the right with high probability.

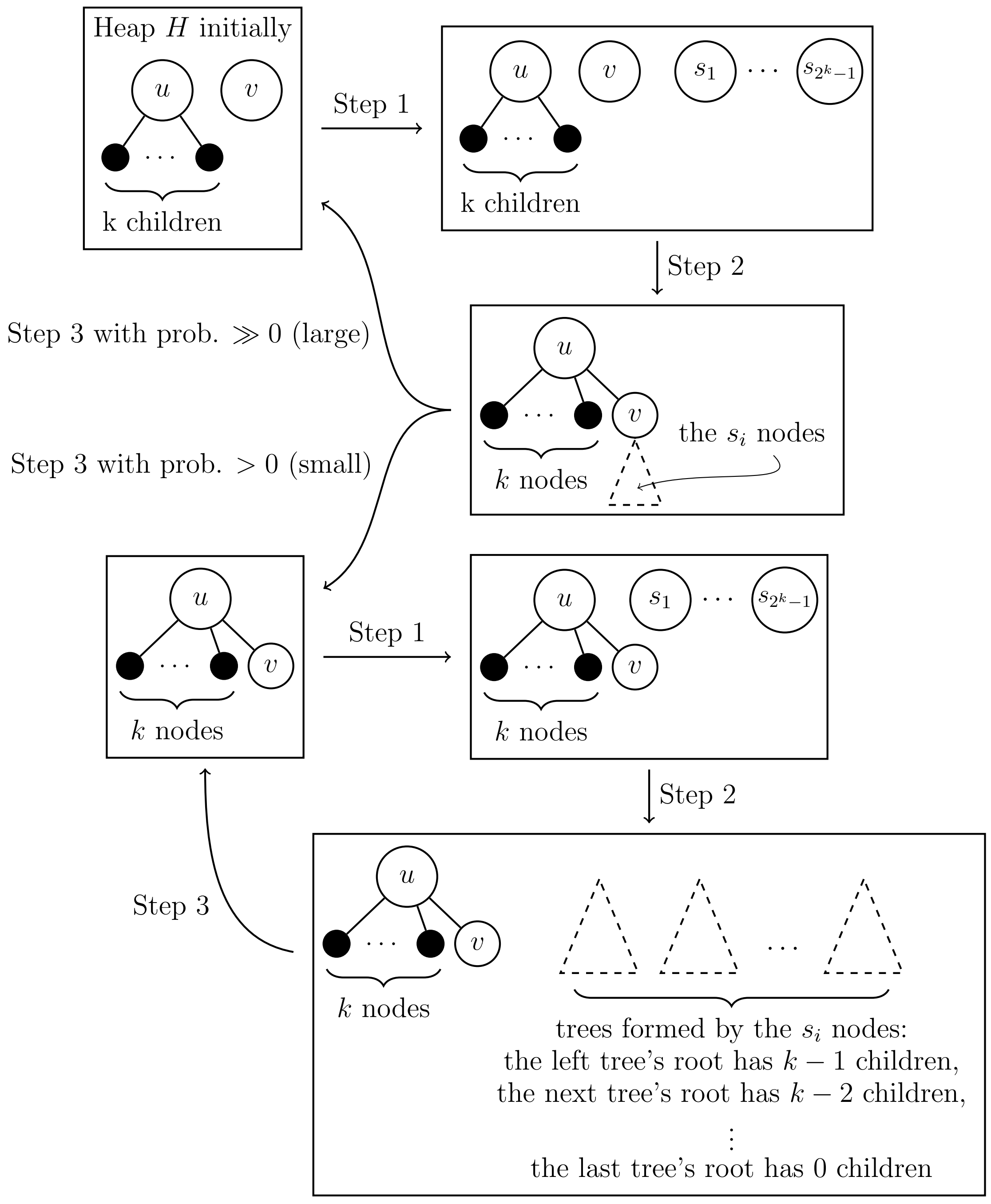

Heap H initially
u
v
k children
Step 1
u
v
s_1
$\cdots$
s_{2^k-1}
k children
Step 2
Step 3 with prob. $\gg 0$ (large)
Step 3 with prob. > 0 (small)
u
v
k nodes
the s_i nodes
u
v
k nodes
Step 1
u
s_1
$\cdots$
s_{2^k-1}
v
k nodes
Step 2
Step 3
u
v
k nodes
trees formed by the s_i nodes:
the left tree's root has $k-1$ children,
the next tree's root has $k-2$ children,
$\vdots$
the last tree's root has 0 children

Figure 2: Detailed description of how one iteration of our procedure works. If only one arrow is shown, the probability of going from the state it starts at to the state it ends at is 1. Triangles denote trees/subtrees.

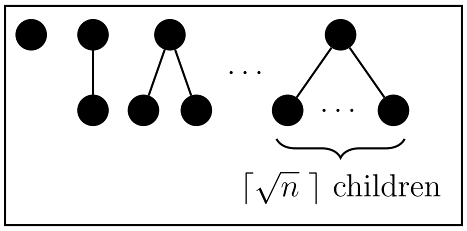
Notice that H always has a positive probability of gaining a single extra child (and no extra descendants), resulting in the heap we are trying to create. Furthermore, once H enters this state, it will never leave. As such, if we apply the procedure a sufficiently large number of times—and provided H had the structure shown in Figure 2— we can construct a sequence of steps that gives the desired resulting H with probability arbitrarily high. Fix a number r of repetitions such that this probability is at least $\sqrt{1-\epsilon}$.

Then consider the initial sequence of steps from the induction hypothesis, plus the step of adding v, plus the r repetitions of the procedure described above. The probability that this sequence of steps succeeds is at least $1-\epsilon$, as desired.

Thus, by induction, the result holds for all c. □

Theorem 2. There exists a request sequence such that any randomized Fibonacci Heaps takes expected time $\Omega(\sqrt{n})$ on average per request where n is the size of the Fibonacci heap.

Proof. Using Lemma 1, construct $\lceil\sqrt{n}\rceil$ Fibonacci heaps where the ith heap has i children, each with independent success probability at least $(1/2)^{1/\lceil\sqrt{n}\rceil}$. Now merge them all together. With probability at least $(1/2)$, the resulting heap is now as shown below.



Now perform the following procedure a very large number of times:

1. Add two elements t_1, t_2 smaller than every element in the heap with $t_1 < t_2$.

2. Delete-min twice.

If our earlier steps give us the heap shown earlier, repeatedly applying this procedure yields the cycle of states shown in Figure 3. It is clear that the last delete-min operation in the procedure takes $\Theta(\sqrt{n})$ time. Furthermore, the size of the Fibonacci heap when this operation is requested is $1+\cdots+(\lceil\sqrt{n}\rceil+1)=\Theta(n)$, so the runtime is big-Theta of the square root of the actual heap size. By applying this procedure a very large number of times, we can make the average time per request over the entire sequence be $\Theta(\sqrt{n})$.

However, this analysis was assuming we were able to get the heap shown in Figure 5 initially. This happens with probability at least $(1/2)$, so the average expected cost of each operation is still at least $\Omega(\sqrt{n})$.

□

Note that one doesn't actually need to make use of the merge operation to construct a request sequence proving the above theorem. The only place they are used is to get the heap into an initial bad state as shown at the beginning of the proof. Instead, we can construct all of the trees that comprise this state in the same heap, largest first. This guarantees with probability at least 1/2 that we get the heap into the same state.

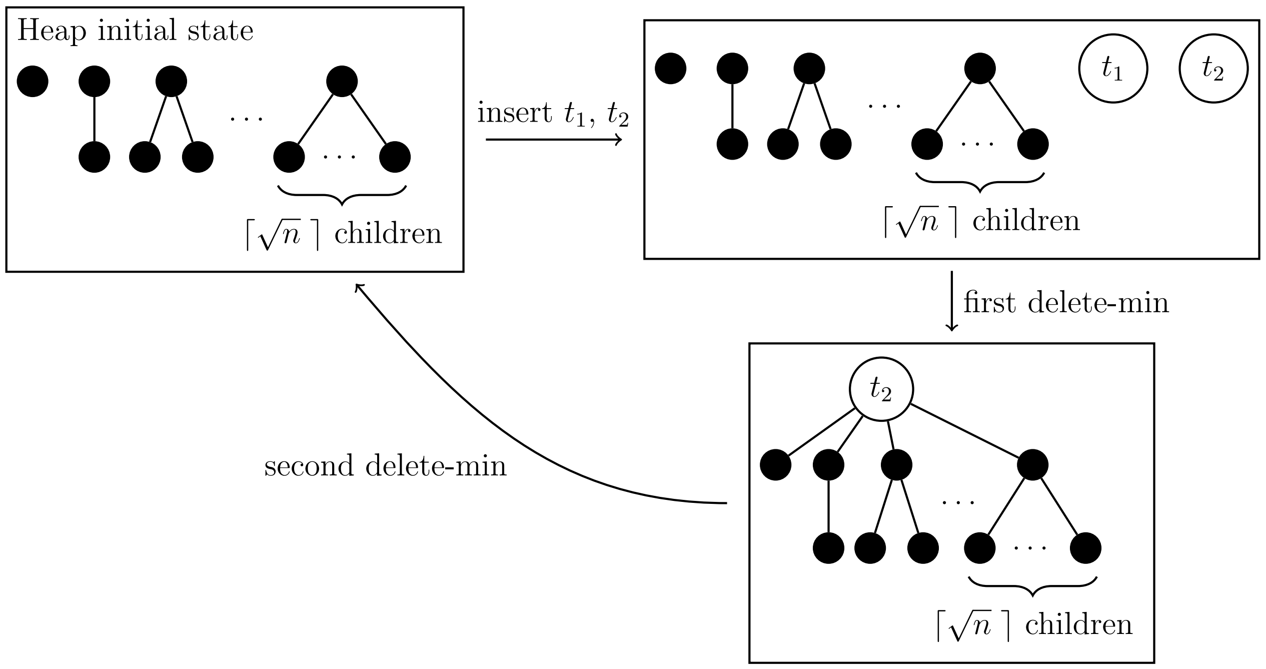

Heap initial state
$\lceil\sqrt{n}\,\rceil$ children
insert t_1, t_2
t_1
t_2
$\lceil\sqrt{n}\,\rceil$ children
first delete-min
t_2
$\lceil\sqrt{n}\,\rceil$ children
second delete-min

Figure 3: Creating many expensive delete-min operations.

4 Replacing s with n

The dependence on the number of operations s is in the runtime of randomized Fibonacci heaps rather annoying because it prevents randomized Fibonacci heaps from attaining the same time bounds as standard ones. However, there is a simple modification we can make to the heap to remove any dependence of the runtime on s.

Specifically, we can rebuild the heap periodically. Here, rebuild means we traverse the heap in some arbitrary order and insert all elements into an empty heap, then discard the old heap. This takes $O(n)$ time. If we keep a counter of the number of operations since the last rebuild and we rebuild the heap any time this counter gets to n, the amortized cost of rebuilds is free because we can charge the rebuild cost to the operations since the last rebuild. Furthermore, this ensures that the number of operations since the heap was rebuilt is never more than n, replacing s with n in any statements about the runtime.

For example, Theorem 1 proves an upper bound of $\log^2 n$ on the expected amortized runtime of delete-min for these modified heaps. Theorem 2 no longer applies to these heaps because the heap will get rebuilt long before we can force it into a bad state.

5 Closing the Problem Back Up: an $\omega(\log s)$ lower bound

Note that all figures in this section are hand-drawn (due to their complexity) and are attached after the end of the document.

The considerations in the preceding section raise the question of whether the modified randomized Fibonacci heaps have the same time bounds in expectation as standard Fibonacci heaps. In this section, we resolve this question in the negative by showing that unmodified randomized Fibonacci heaps (hereafter referred to simply as randomized Fibonacci heaps) can be made to take $\omega(\log s)$ expected amortized time per delete-min. This implies an $\omega(\log n)$ lower bound on modified randomized Fibonacci heaps.

Our approach is to get the heap into essentially same bad state we construct in Theorem 2. Specifically, we want it to be in a state of the form shown in Figure 4.

Figure 4: See attached.

Our construction will make use of other request sequences as subroutines. We will be very careful to control the exact internal state of the heap throughout this process. To this end, we will construct our subroutines so that they have a completely *predictable effect* on the heap, even though they may invoke heap operations that employ randomization. In other words, we require of our subroutines that if we know the exact starting state of the heap and that the subroutine succeeded, we should be able to infer the exact ending state of the heap.

We will refer to a subroutine that turn a heap into the state shown in Figure 4 as an *evil subroutine for rank m.*

We now describe our first subroutine which we construct.

Lemma 4 (*Shifting Subroutine*)**.** For every constant j, there exists a subroutine with predictable effect given by the diagram below which fails with probability at most p and uses

$-\log_e p \cdot \text{poly}(m) \cdot 2^{m/j}$ heap operations.[5] Furthermore, it does not interfere with any root nodes of rank greater than i.

Figure 5: See attached.

Proof. To start off, decrease-key on u so that it is the smallest element in the heap. Then add an element t_2 such that $u < t_2 <$ everything else.

We perform the following procedure $y = -\log_e p \cdot 2^{m/j}$ times:

1. Insert an element t_1 such that $t_1 < u < t_2 <$ everything else. Then perform a delete-min operation.
2. Set $w = \min(\lfloor m/j \rfloor, i)$. For the w children of t_2 having the largest number of children of their own, decrease-key on them to send them to the root list.

See Figure 6 for a flowchart showing the effects of these operations. We fail to get the heap into the desired state with probability at most $(1 - 2^{-m/j})^y \leq p$. □

The above subroutine allows us to increase the rank of one node in the root list, at the cost of creating some "gaps": first, it is missing a bunch of low-rank nodes; second, it has no node of rank one less than the largest rank. We show that a subroutine for rebuilding the low rank nodes is the only other thing we need to obtain an evil subroutine.

Lemma 5. Fix j. Suppose we are given an evil subroutine[6] for rank $m - \lfloor m/j \rfloor$ of predictable effect which does not interfere with any root nodes of rank greater than $m - \lfloor m/j \rfloor$. Also suppose that this subroutine uses x operations to obtain failure probability $p_{\text{evil sub}}$.

Let $p_{\text{shift sub}}$ be the probability of success of the shifting subroutine with our given value of j.

Then there exists an evil subroutine for rank m of predictable effect which does not interfere with any root nodes of rank greater than m. Furthermore, this subroutine has failure probability at most $m^2(p_{\text{evil sub}} + p_{\text{shift sub}})$. It uses at most $O(m^2(x - \log_e p_{\text{shift sub}} \cdot \text{poly}(m) \cdot 2^{m/j}))$ heap operations.

Proof. We start by applying our shifting subroutine to increase the rank of the highest rank node, then our evil subroutine to rebuild the low rank nodes. This allows us to apply our shifting subroutine again to increase the rank of the second-highest rank node without disturbing the highest-rank node. Continuing in this fashion, we can iterate a total of m times[7] to obtain a subroutine with effect given by the diagram below.

Figure 6: See attached.

[5]Note that poly(m) means some polynomial in m.

[6]Note that a subroutine for rebuilding low-rank nodes is just an evil subroutine with a smaller value of m.

[7]We'll also need to insert a node of after finishing iterating, but that's easy.

Iterating the above subroutine itself m times[8] gives us the state shown in Figure 4; ie., we have an evil subroutine. By a union bound, this new evil subroutine will have a probability of failure of $m^2(p_{\text{evil sub}}+p_{\text{shift sub}})$. Also, it need only use $O(m^2(x-\log_e p_{\text{shift sub}}\cdot\text{poly}(m)\cdot 2^{m/j}))$ heap operations.[9]

This completes the proof. □

So if we have an evil subroutine, we can construct another one. It turns out that the above lemma will allow us to actually construct a *better* one. However, we need an evil subroutine to start off with in the first place, which we now give.

Lemma 6. *For every m, there exists an evil subroutine for rank m with predictable effect which uses $O(2^m)$ heap operations and does not affect any root nodes of rank greater than m. It has failure probability 0.*

Proof. If we add 2^{m+1} nodes smaller than any other nodes in the heap and delete-min, we get a heap in the desired state. However, if we only did this, we wouldn't be able to keep track of where all the nodes go, so it wouldn't be a *predictable* subroutine. We can get around this pretty easily, however.

Here is a sketch. When inserting nodes, we frequently stop the insertion process to force a consolidation by inserting a dummy and doing a delete-min operation. If we do this frequently enough (say, after every insertion of a real node), then there will be at most one pair of nodes that can consolidate, so we can predict exactly what will happen. □

We can now prove our main theorem which immediately implies the desired $\omega(\log s)$ lower bound.

Theorem 3. *Fix any $j \geq 1$. There exists a request sequence for randomized Fibonacci heaps in which the expected average cost of delete-min is $\frac{j}{4}\log s$ where s is the sufficiently large length of the request sequence.*

Proof. We construct a request sequence which, with probability at least 1/2, gets the heap into the state shown in Figure 4 using $s = \text{poly}(m)\cdot 2^{m/j}$ heap operations. This gives $m \geq \frac{j}{2}\log s$ for sufficiently large m. The same method used in Theorem 2 can then be applied to make the expected average cost of the delete-min operations at least $m/4$, while only increasing the length of the request sequence by a factor of two.[10] This will give the desired $\omega(\log s)$ lower bound.

Thus, we simply need to construct an evil subroutine with 1/2, gets the heap into the state shown in Figure 4 using $s = \text{poly}(m)\cdot 2^{m/j}$ heap operations.

We start with the evil subroutine obtained in Lemma 6. We then apply Lemma 5 a total of $j-1$ times where the ith application (indexed at $i=1$) has $p_{\text{shift sub}} = \frac{1}{m^{2(j-i)}2^{j-i}}$ and the j in the lemma set to i.

By a union bound, the probability of failure is at most $\frac{2^{j-2}m^{2(j-1)}}{m^{2(j-1)}2^{j-1}} = \frac{1}{2}$.

[8]We obtain the initial state of a heap with one node by simply inserting a node into the heap.

[9]The factor m^2 in the number of heap operations gets absorbed into the poly(m) term.

[10]The expression $m/4$ comes from the fact that we lose a factor of two using the method of Theorem 2 and a factor of 2 from the fact that this only potentially works with probability 1/2.

By Lemma 5 and with some calculation, one can show that the number of heap operations used is $\mathrm{poly}(m) \cdot 2^{m/j}$. For example, if $j = 1$, we get $\mathrm{poly}(m)2^m$. For $j = 2$, we get

$$O(2^{m/2}) - \log_e \left(\frac{1}{m^{2(2-1)}2^{2-1}} \right) \cdot \mathrm{poly}(m) \cdot 2^{m/2} = \mathrm{poly}(m) \cdot 2^{m/2}.$$

For $j = 3$, we get

$$\mathrm{poly}(m) \cdot 2^{(2/3)m/2} + \mathrm{poly}(m) \cdot 2^{m/3} = \mathrm{poly}(m) \cdot 2^{m/3},$$

etcetera.

This completes the proof. □

6 Directions for Future Work

One simple direction for further work would be to get an exact asymptotic characterization of the expected amortized cost of delete-min operations for randomized Fibonacci heaps. Our work shows it is $\omega(\log s)$ and $O(\log^2 s)$, but which is it closer to? We suspect the lower bound could probably be tightened further through slightly more careful analysis, at least to the point where it gives some explicit function which is $\omega(\log s)$ as the lower bound.

Another direction would be to try and make additional modifications to randomized Fibonacci heaps beyond those in Section 4 in order to obtain the same performance as standard Fibonacci heaps. Here is one idea along these lines. Suppose that instead of cascading up the tree towards the root during a decrease-key, we also cascade down. That is, we also perform a separate cascade starting at the most recently added child of the current node and flip a coin. If it is heads, we cascade the child to the root and recurse on the child's most recently added child.

It seems like this modification would critically break most of the bad request sequences discussed herein.

Another possible, but less well-defined modification would be to simply have the heap split up a node whenever it gets too many children.

References

[Cha13] Timothy M. Chan. Quake heaps: A simple alternative to fibonacci heaps. In Andrej Brodnik, Alejandro Lpez-Ortiz, Venkatesh Raman, and Alfredo Viola, editors, *Space-Efficient Data Structures, Streams, and Algorithms*, number 8066 in Lecture Notes in Computer Science, pages 27–32. Springer Berlin Heidelberg, January 2013.

[FT87] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, July 1987.

[Kar13] David Karger. Advanced algorithms homework 1, problem 4. September 2013.

Figure 4. A very bad state. Triangles denote (possibly empty) subtrees.

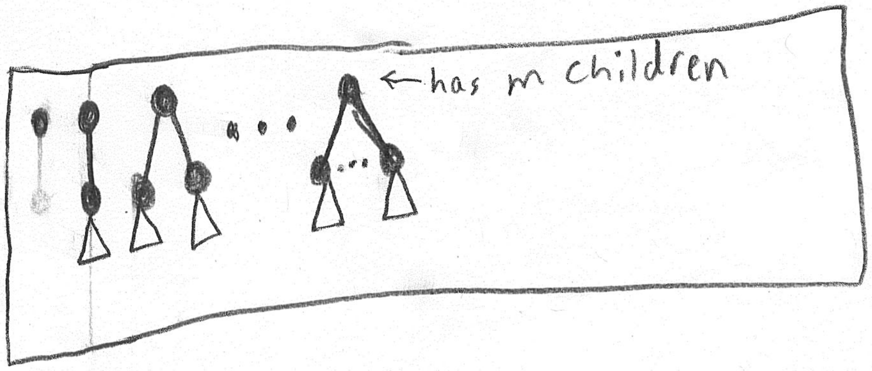


Figure 5. Let $w = \lfloor m/i \rfloor$.

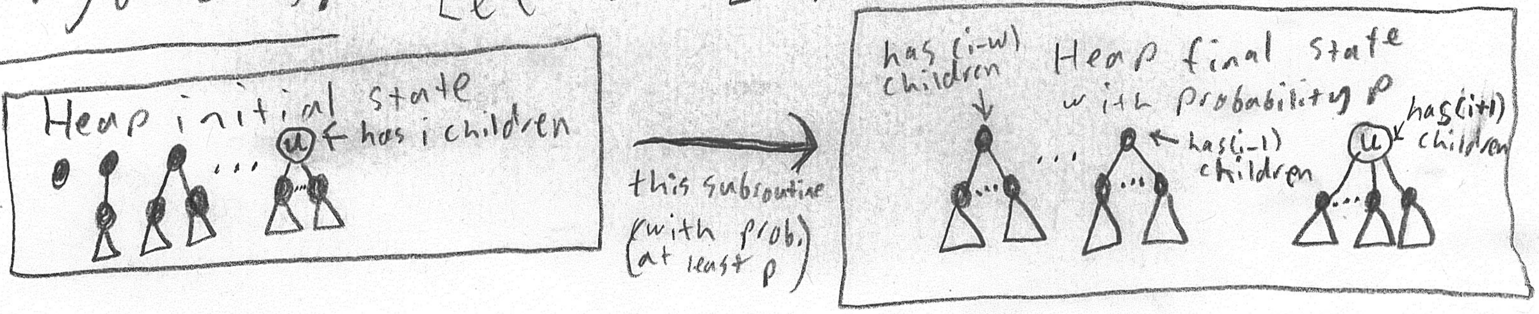


Figure 6. Illustrated Proof of Lemma 4. Triangles represent subtrees.

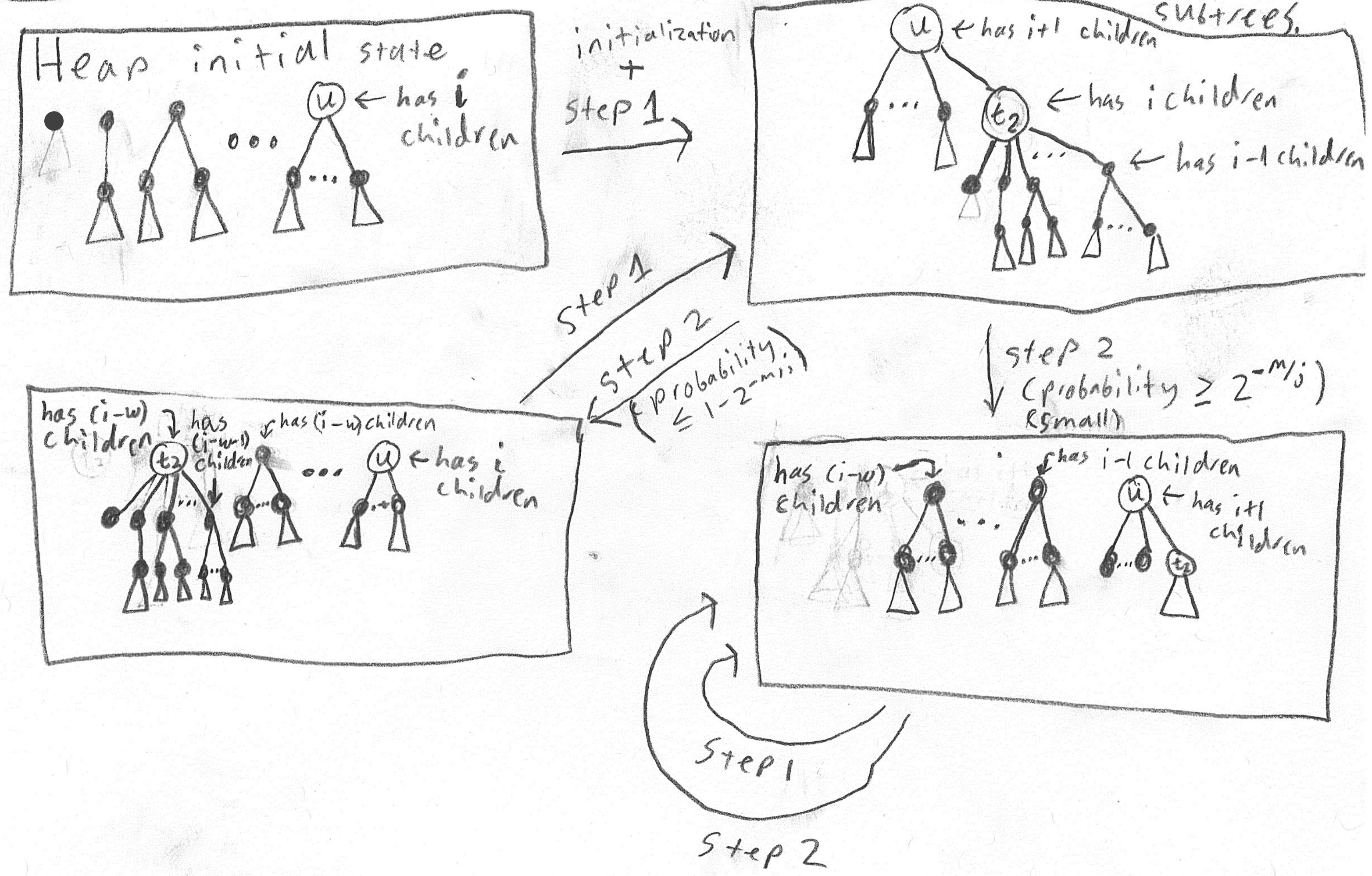
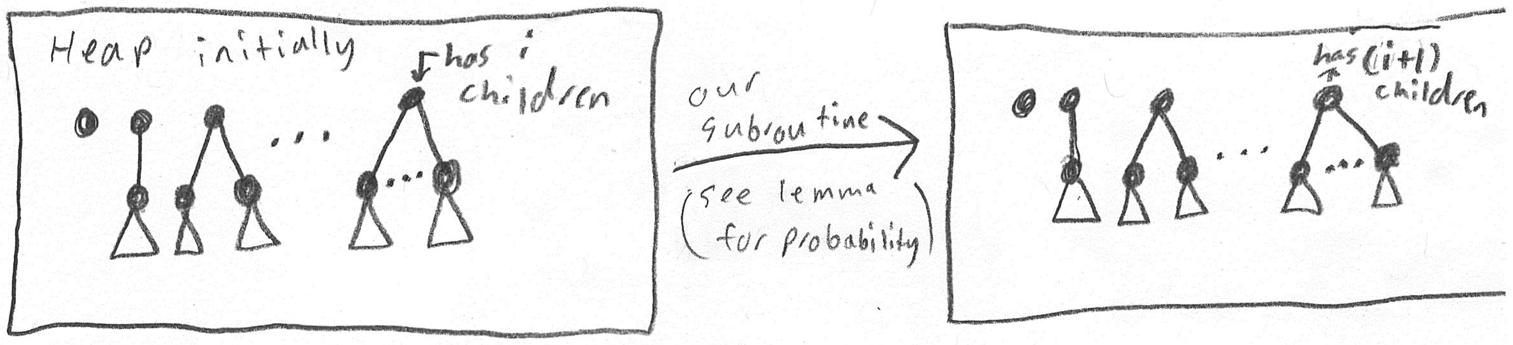


Figure 7.


Heap initially
has i children
our subroutine
(see lemma for probability)
has (i+1) children