Lecture Notes on Fibonacci Heaps — CS 670

David Kempe

September 23, 2022

Fibonacci Heaps are a data structure for an asymptotically faster implementation of Prim's MST algorithm, and Dijkstra's Shortest Paths algorithm. The data structure is designed to contain the nodes we are currently considering for addition to the growing root component. Hence, we need it to support the following operations:

1. `insert`(v, h): Insert an element v into the structure h (together with an associated value). In Prim's and Dijkstra's algorithm, this element will be a node to which we have just discovered the first edge.
2. `delete-min`(h): Find the element with the smallest associated value in the structure h, return it, and remove it from h. This is done when we choose the next node to include in the root component in Dijkstra's or Prim's algorithm.
3. `decrease`(v, Δ): Decrease the value associated with node v by Δ. This becomes necessary if we just discovered a new (and cheaper) way of adding the node v. At that point, v is *not* yet in the component including the root, but it may just have become a more attractive candidate for inclusion in the next step.

In Dijkstra and Prim, each node is inserted and deleted once, hence we have a total of n `insert` and `delete-min` operations. A `decrease` is always the result of looking at an edge, so there will be at most m `decrease` operations. Our goal is to have the total running time be $O(m + n \log n)$. This would require us to implement `decrease` to run in $O(1)$ time, and `insert` and `delete-min` in $O(\log n)$.

The standard way of achieving such runtime bounds would be to ensure that every single one of those operations never takes more than $O(1)$ resp. $O(\log n)$ steps. However, we do not know how to achieve that. Every now and then, an operation will have to be expensive. What we would like to show, then, is that expensive operations are rare, so that their effect "averages out". This idea is called "amortized analysis", and we will see it in action below.

1 Cleaning up a Floor

Let us first start with a simple (albeit slightly contrived) example to illustrate the idea of amortized analysis. Suppose we have two operations, `drop` and `clean-up`(t). The `drop` operation drops a piece of trash on the floor (and takes one unit of time), while `clean-up`(t) cleans t pieces of trash from the floor (we assume that t is no more than the actual number of pieces on the floor). `clean-up`(t) takes one time unit of time for each of the t pieces, plus one extra unit to carry all of them to the trashcan, so it takes $t + 1$ steps. We also assume that we start with a clean floor.

Now, if we look at a `clean-up` operation in isolation, it may take very long — there may be a lot of trash on the floor, and t may be large. But if that is so, we must have executed a lot of (fast) `drop` operations previously, to accumulate that much trash. So how long can a sequence of d `drop` and c `clean-up` operations take in total?

Let's first look at a simpler version: we drop d items, and then we clean all of them up. The dropping takes time 1 each, and the cleanup time $d + 1$. So it turns out that a single cleanup is expensive, but on

average, each operation costs $\frac{1}{d+1} \cdot (d \cdot 1 + 1 \cdot (d+1)) = \frac{2d+1}{d+1} \leq 2$ steps. Another way to think about it is that we can attribute each extra step of cleaning up to one (easy) step of dropping something. A `clean-up` operation cannot be expensive unless we had a lot of (cheap) `drop` operations preceding it. If we took one step of `clean-up`, and arbitrarily claimed that it belonged to a preceding `drop` step, the total (and average) would not change. So our goal would be to somehow spread out the cost for `clean-up` evenly among the cheap `drop` operations.

How do we formalize this intuition, gained from one special case, into a more general analysis? Suppose that instead of doing all this work ourselves, we paid someone to do it. For each `drop` operation, we paid 1, and for each `clean-up`(t) $t+1$ dollars. Since we know that each item may potentially have to be cleaned up, we could pay him 2 for the dropping instead. Then, when time comes around to cleaning up, all the picking up is already pre-paid, and we only have to pay 1 for the trip to the trashcan. So the total cost for a sequence of d `drop` and c `clean-up` operations is at most $2d + c$ dollars.

Exactly the same kind of accounting works if we do the work ourselves: we pretend that the `drop` operation takes 2 time steps, to account for any subsequent `clean-up` we may do on this item. In the terminology we will use later, we perform two units of work for `drop`, and attach one *credit* to the item. The credit later pays for the picking up part. So the total number of steps taken is at most $2d + c$. Of course, some of these steps may be unused: for instance, if we never execute `clean-up`, the total time for d drops was actually d, whereas we counted $2d$. But at least, we get an upper bound of $O(d)$.

2 Binomial Heaps

We will obtain our desired Fibonacci Heaps by starting with a somewhat simpler data structure, called *Binomial Heaps*, and then modifying them. Binomial Heaps will support the `insert` and `delete-min` operations, and we will later add the `decrease` functionality.

2.1 Binomial Trees

Binomial Heaps are collections of *Binomial Trees*[1]. Binomial Trees are denoted by B_k, and defined inductively as follows: B_0 is the tree consisting of a single node, and no edges. B_{k+1} is the tree consisting of a *root node* r of degree $k+1$. One of its children is the root of a B_0, one is the root of a B_1, one the root of a B_2, and so forth, up to B_k. For a root r, its *rank* $\text{rank}(r)$ is its number of children. The rank $\text{rank}(T)$ of a tree T is the rank of the root node, so that $\text{rank}(B_k) = k$.

It is not too difficult to observe that there is an alternate definition of binomial heaps leading to the same outcome. B_{k+1} can be obtained by taking two B_k trees, and connecting their roots with an edge. This leads us to two simple, but very useful, observations.

Proposition 1 *Two trees B_k can be merged into one B_{k+1} in constant time. We call this the* `merge` *operation.*

Proof. All we need to do is designate one as the child, and have its root point to the other root as a parent, as well as adding a pointer from the parent to its newly acquired child. ■

Proposition 2 *A B_k tree contains exactly 2^k nodes.*

Proof. By induction on k. ■

[1] This is not the same as a *binary* tree.

2.2 Binomial Heaps — a basic version

A *Binomial Heap* h is a collection of Binomial trees, such that for each rank k, at most one tree B_k is contained in h. In addition, and very crucially, we require that each of the trees in h satisfy the *heap property*: this means that the value associated with a node v is no larger than the value associated with any of its children. While we will relax (in other words, violate) many other properties, we will always strictly maintain the heap property.

We will support three operations on Binomial Heaps:

1. `insert`(v, h): insert an item, as before.
2. `delete-min`(h): return and delete the minimum item in the tree.
3. `meld`(h, h'): take two heaps and combine them into one. This is used as a subroutine for the implementation of the other operations.

Our goal is to implement the `insert` operation in time $O(1)$, and the `delete-min` operation in time $O(\log n)$.

2.2.1 `meld`

Let us start with the `meld`(h, h') operation. It would be easiest to implement if we could just take all the trees from h and h', and combine them into one new heap. However, that may violate the property that the resulting heap may contain at most one tree of each rank, as both h and h' may contain trees of the same rank k. When h and h' do contain trees of the same rank k, we can apply the `merge` operation mentioned above: make one of them a child of the other, thus generating a tree of rank $k+1$. Of course, we make the one with larger value the child of the other, in order to maintain the heap property.

We iterate this while there are still trees of the same rank (including the newly generated ones). How many `merge` operations does this take? Each `merge` takes two trees, and replaces them with one, so the number of trees total decreases by 1. Hence, if we start with $|h| + |h'|$ trees, it takes at most that many steps. This brings us to the next question: how large can $|h|$ (or $|h'|$) be? This is answered by the following lemma:

Lemma 3 *A Binomial Heap (over an n-element universe) contains at most $O(\log n)$ trees.*

Proof. A Binomial Heap contains at most one tree of rank 0, one of rank 1, one of rank 2, ... What is the highest rank any tree could have? No tree can contain more than n nodes, as there is only a total of n elements to insert into trees. Because a tree of rank k contains 2^k nodes (by Proposition 2), a tree of rank at least $1 + \log n$ would contain $2^{1+\log n} = 2n$ nodes, more than there are. So there are no trees of rank exceeding $\log n$, so the total number of trees in a heap is at most $1 + \log n = O(\log n)$. ∎

Now, as both heaps have at most $O(\log n)$ trees, the `meld` operation uses at most $O(\log n)$ `merge` steps. If we start with rank 0, and look at the trees in increasing order of rank, it can be seen that the implementation actually does take $O(\log n)$ time.

2.2.2 `insert`

The basic implementation of the insert operation is quite simple. To insert a new element v, we just create a B_0 tree containing only v. Unluckily, the binomial heap h may already contain a B_0, so we can't just add the new one — it would violate the condition that we have at most one tree of each rank. Formally, we create a new heap h', consisting of just the new B_0 tree, and meld it with h.

The creation of a new tree and heap takes $O(1)$ time. In the worst case, the subsequent `meld` operation takes $O(\log n)$, so the implementation takes $O(\log n)$.

2.2.3 `delete-min`

The `delete-min`(h) operation is supposed to find and delete the element with the smallest associated value. In order to find it, it suffices to check the roots of all trees in h. That is because each tree satisfies the heap condition, so each tree's smallest element is in the tree's root. The smallest overall is then the smallest element among all the candidates in the roots. By Lemma 3, there are at most $O(\log n)$ trees in the heap, so the minimum can be found in $O(\log n)$ steps.

After that, we remove the root that contained the minimum. In so doing, we generate a new heap h', consisting of all the subtrees that were rooted at that node. If the tree was a B_k, we generate a B_0, a B_1, and so forth, up to a B_{k-1}. In order to reinstate the heap property afterwards, we need to meld the new heap h' with the old heap h. The melding takes at most $O(\log n)$ steps as well. So the total time for `delete-min` is also $O(\log n)$.

2.3 Improving the running time

While we achieved our goal of a running time of $O(\log n)$ for `delete-min`, we fell short of our goal of $O(1)$ for `insert`. The main reason is that we invoked the `meld` operation as a subroutine, which takes $O(\log n)$ steps. Really, there isn't that much we can do to speed up `meld` beyond that, as even checking if any merges are necessary takes $\Omega(\log n)$ steps. So we will have to refrain from invoking `meld` as part of `insert`. If we don't invoke `meld`, we actually achieve the running time of $O(1)$ for `insert`.

But wait! Then, as we generate new trees and add them to the heap, there may be multiple trees of the same rank. That in turn will make `delete-min` take more time, as it needs to check the roots of all trees to find the minimum. To keep the search for the minimum element within $O(\log n)$, we need to ensure that before the search starts, we reinstate the property that there is at most one tree of each rank. In other words, we will need to call a variant of the `meld` operation before doing the search. We call it `clean-up`(h): it takes a "violated binomial heap" — one in which there may be multiple trees of the same rank — and, in the same way as `meld`, merges trees of the same rank to reduce the number of trees.

So how long does this new `clean-up` operation take? It certainly depends on the number of trees in the "violated heap" h: if there are $|h|$ trees before the `clean-up` operation, it may take time $\Omega(h)$. In other words, if we had a lot of `insert` operations (say, $n/2$), generating $n/2$ 1-node trees, then `clean-up` will take $n/2$ steps. So in the worst case, the running time of `clean-up` will be quite bad. However, as with our previous example of `clean-up` in Section 1, such a bad running time must be caused by a lot of cheap operations preceding the `clean-up` operation, so the average time per operation is not that bad. In other words, it is time to look for an amortization argument.

Here, the argument is not quite as easy, as we don't just drop or pick up items: trees are merged, and fall apart again, and so forth. But the basic idea is the same. Here, we will do extra work (or pay extra money) during some operations, to ensure that each tree has a credit. This credit can then be used when two trees are merged, to pay for the cost of the `merge` operation. We will call this credit the *merge credit*, and design our analysis so that it ensures the following key invariant:

> (*) At each time, each tree has a merge credit (which is associated with its root).

We need to verify how to maintain this invariant across all operations that we do on the data structure:

1. `insert`: when a new tree is created via an `insert` operation, we just do one more unit of work, which will be stored as a credit with the new tree. The amount of work done is still $O(1)$ (we don't execute `meld` here any more).

2. `merge`: the `merge` operation takes two trees B_k, and combines them into one B_{k+1}. By the invariant, both trees have a credit before the operation. We use one of the credits to pay for the `merge`, while the second one stays with the resulting B_{k+1}. Hence, we can actually execute `merge` for free (it is already paid for), and the invariant holds afterwards.

3. `delete-min`: the only other way in which new trees are generated is by a `delete-min` operation. This removes the root of some B_k, and generates k new trees $B_0, B_1, \ldots, B_{k-1}$. We can ensure that each of these new trees has a merge credit by doing one extra unit of work for each of them, a total of $O(\log n)$ extra work. As `delete-min` was allowed to take time $O(\log n)$ anyway, this does not make it asymptotically slower.

Hence, we have given an accounting scheme that maintains the credit invariant (*). In this accounting scheme, `insert` takes amortized time $O(1)$, `merge` can be done for free, and `delete-min` takes time $O(\log n)$. Using the free `merge` implementation, `meld` runs in time $O(\log n)$, regardless of how many trees are in the "violated heap". (Notice that the $\log n$ is necessary just to check if any `merge` operations need to be done.) After the `meld` operation, `delete-min` runs in time $O(\log n)$. So, overall, the amortized time for `insert` is $O(1)$, and for `delete-min`, it is $O(\log n)$.

To iterate once more the important point about amortized analysis, this does not mean that any one of these operations takes that little time, but it does mean that a sequence of c `insert` and d `delete-min` operations takes time $O(c + d \log n)$.

3 Fibonacci Heaps

To get a data structure that supports all the operations we want for Prim's and Dijkstra's algorithm, we still need to add a `decrease` operation that runs in (amortized) time $O(1)$. Recall that `decrease`(v, Δ) decreases the value associated with node v by Δ.

3.1 Implementation of `decrease`

A first approach would be simply to go ahead and change the value. But that may violate the really crucial heap property: by decreasing the value for v, its value may now be smaller than its parent's. A first approach to remedy that would be to swap v with its parent. But then, its new value may also be smaller than its grandparent's, so a further swap is necessary, and so forth. In the extreme case, it may need to be swapped all the way to the root. Since the height of a tree may be as much as $\Omega(\log n)$, this could take $\Omega(\log n)$ steps, which is more than the $O(1)$ we are aiming for. Also, it doesn't seem like an amortized analysis will help us here, as the sequence of decreases (i.e., Δ values) may be such that all swaps will really be necessary.

An alternate approach is the following: when v's value decreases, we remove it from the tree, together with the entire subtree rooted at v (so we end up with two trees afterwards). Since v is now the root of its own tree, it certainly won't violate the heap property. However, something else may go wrong now. If we have a tree B_k with root r, and all of r's children lose their children in this way, the resulting tree has only $k + 1$ nodes in the end. In other words, Proposition 2 does not hold any more, and it was a crucial part in proving Lemma 3. In fact, a heap can contain trees up to rank $\Omega(n)$ in that case, in which case `delete-min` would take time $\Omega(n)$ just to search all of the roots. In order to keep the running time of `delete-min` small enough ($O(\log n)$), we need to make sure that there are no trees of rank exceeding $O(\log n)$. And a good way to ensure that is to make sure that a tree of rank k contains at least c^k nodes, for some constant $c > 1$.

The upshot of the preceding paragraph is that we can't just delete subtrees arbitrarily, but rather have to stop at some point. The specific rule we use is the following: if we decrease the value for some node v, we let p be the parent of v. We always delete v and its subtree, making them a new tree. If no other child of p has been removed, then that's all we do. On the other hand, if p has already lost a child, then we also remove p itself and its remaining subtree now. Of course, p may be the second child of its parent p' that is removed. In that case, we also remove p', and so forth.

3.2 Analysis of `decrease`

How long does the new `decrease` operation take? In the worst case, we will remove all the ancestors of the node v, which would mean that we do $\Omega(\log n)$ removals, and create as many new trees. So the new `decrease` implementation still could take time $\Omega(\log n)$. So what have we gained over the first idea?

The advantage is that in order for `decrease` to cut a lot of trees, each of the nodes along the way has to already have lost one of its children. And the removal of the first child is cheap. So in order to have an expensive operation, of cost c, we first had to execute c cheap operations of cost $O(1)$. So it's time for another amortization argument.

We introduce a second type of credit, a tree removal credit. When a node v is decreased (and removed) as the first child of its parent p, we spend an additional unit of work to give p a removal credit. Later, when another child v' of p is decreased and removed, we use the credit to pay for the removal of p itself. If the parent p' of p is also removed, then it too must have a removal credit to pay for the removal, so we will be all set. That way, we actually implement the `decrease` operation in amortized $O(1)$ time.

One little subtlety: the `decrease` operation, by removing a subtree, can generate an additional new tree. That tree also needs a merge credit, to maintain the invariant (*). Again, whenever we actually remove a tree, we can do two more units of work, to give the new tree its merge credit, and have a merge credit ready for the parent if it becomes removed. This still leaves the amortized time for the `decrease` operation at $O(1)$.

3.3 Size of a tree in a Fibonacci Heap

Recall again our goal with the "delete one child but not two" rule: the deletion was necessary to maintain the heap property — on the other hand, we did not want to delete too many nodes from a tree, as we still wanted a tree of rank k to have c^k nodes, for some constant $c > 1$. The reason for wanting this property is so that the searching part of `delete-min` should take only $O(\log n)$ steps.

So did we succeed? What is the smallest number of nodes that could be in a tree of rank k? Let us call this number S_k, and try to derive a characterization. $S_0 = 1$, as a node with no children still is one node. $S_1 = 2$, as a root r with one child v still is a tree of size 2; v had no children to begin with, and has not lost any. Now let us look at a root r with k children. Label the children $v_0, v_1, \ldots, v_{k-1}$, by non-decreasing rank. Because the `meld` operation ensures that there is at most one child of each rank i, prior to removals, each node v_i had rank at least i. (The rank could be larger, because some ranks may not have been represented.) Because each v_i had at most one child removed, its rank is at least $i-1$, except for v_0, which still has rank at least 0. Thus, by definition, the subtree rooted at v_i must still contain at least S_{i-1} nodes for $i > 0$, and $S_0 = 1$ node for $i = 0$. Thus, the total number of nodes in a tree rooted at r is at least

$$S_k \quad = \quad 1 + S_0 + S_0 + S_1 + S_2 + \ldots + S_{k-3} + S_{k-2}$$

(1 for the node r itself). A completely identical argument about a node r of rank $k-1$ gives

$$S_{k-1} \quad = \quad 1 + S_0 + S_0 + S_1 + S_2 + \ldots + S_{k-4} + S_{k-3}$$

By subtracting the second equation from the first, we find that $S_k - S_{k-1} = S_{k-2}$, or $S_k = S_{k-1} + S_{k-2}$. Except for the fact that we start with $S_1 = 2$ instead of $S_1 = 1$, this is exactly the sequence of Fibonacci numbers.

It still remains to show that the Fibonacci numbers actually grow exponentially fast in k. To get a good guess as to what the constant c would be, assume for now that we are more ambitious, and want to show that $S_k = c^k$ for some $c > 1$ (whereas in reality, all we want is that $S_k \geq c^k$). What would that constant c be? If it is to satisfy the recurrence $S_k = S_{k-1} + S_{k-2}$, it certainly would have to satisfy $c^k = c^{k-1} + c^{k-2}$. Now, dividing this by c^{k-2} gives the necessary condition that $c^2 = c + 1$. By applying the formula that we learned in middle school about solving quadratic equations, we find that the only positive candidate would be the *golden ratio* $c = \frac{1+\sqrt{5}}{2}$. So let us try and show that for this particular c, we have that $S_k \geq c^k$ (they are not equal, as the value S_1 at $k = 1$ is too big).

Lemma 4 *$S_k \geq c^k$, where $c = \frac{1+\sqrt{5}}{2}$ is the solution to the equation $c^2 = c + 1$.*

Proof. We prove this claim by induction on k. For $k = 0$, we have that $S_0 = 1 = c^0$. For $k = 1$, this does not hold with equality: $S_1 = 2 \geq \frac{1+\sqrt{5}}{2} = c^1$.

For the inductive step to $k+1$, we know that $S_{k+1} = S_k + S_{k-1}$. By induction hypothesis (applied to k and $k-1$), $S_k \geq c^k$ and $S_{k-1} \geq c^{k-1}$. Thus,

$$S_{k+1} \quad \geq \quad c^k + c^{k-1} \quad = \quad c^{k-1} \cdot (c+1) \quad = \quad c^{k-1} \cdot c^2 \quad = \quad c^{k+1}.$$

Hence, the claim also holds for $k+1$. ■

The lemma thus shows that the number of nodes in any one tree in the Fibonacci Heap is still exponential in the tree's rank. Hence, the ranks can only range from 0 to $O(\log n)$, and after the `clean-up` operation, the `delete-min` operation only needs to check $O(\log n)$ roots, and hence terminates in $O(\log n)$ steps.

Putting all of this together, Fibonacci Heaps are a data structure that support `insert` in time $O(1)$, `delete-min` in amortized time $O(\log n)$, and `decrease` in amortized time $O(1)$. In other words, a sequence of a `insert`, b `delete-min` and c `decrease` operations always takes time $O(a + c + b \log n)$. In particular, the operations in Prim's or Dijkstra's algorithm take time $O(m + n + n \log n) = O(m + n \log n)$.