The Fibonacci heap ruins my life

A couple of Sundays ago, I wrote an implementation of Dijkstra's algorithm in Clojure. The core algorithm came to twenty-five lines. I banged out the code as I sat in a coffee shop with some other people from the Recurse Center. I ran my program on a data set that has two-hundred nodes in a densely interconnected graph. The program produced best paths from a start node to all other nodes in the graph in about 200 milliseconds.

I closed my laptop, finished my peanut butter, banana and honey sandwich, said goodbye to my friends and spent the rest of the afternoon wandering around the Lower East Side in the dusty sunlight.

By Monday evening, my life had begun falling apart.

Dijkstra's algorithm is a way to find the shortest route from one node to another in a graph. If the cities in Britain were the nodes and the roads were the connections between the nodes, Dijkstra could be used to plan the shortest route from London to Edinburgh. And plan is the key word. The algorithm does reconnaissance. It does not go on a road trip.

Imagine you are the algorithm. You examine all the cities directly connected to London. That is, all the cities connected to London by a single stretch of road. You record the distance between London and each city. You note that you have now explored London. You focus on the city that is the closest to London by road. Let us say Brighton. You examine each of the cities directly connected to Brighton. You ignore London because you have already explored it. For each city, you record the distance from London to Brighton to the city. You note that you have explored Brighton. You now focus on the next unexplored city nearest to London. You continue like this until your destination comes up as the next city to focus on. When this happens, you have found the shortest distance to Edinburgh.

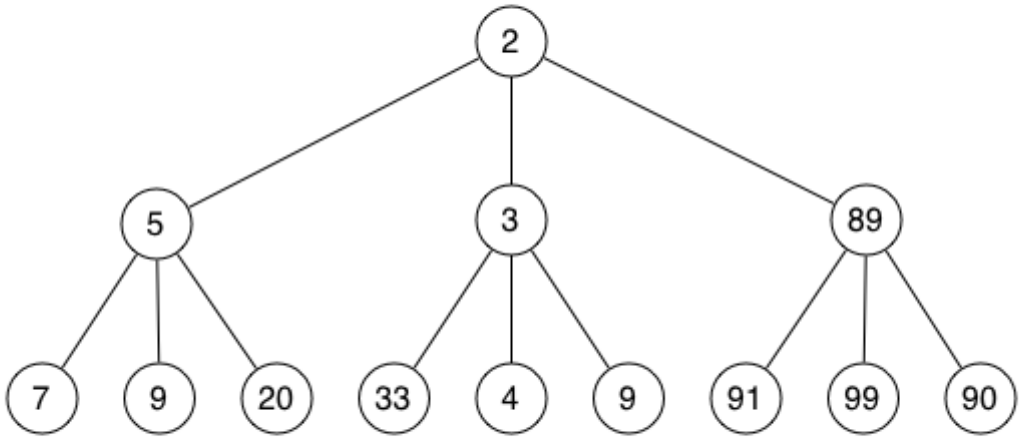
On Monday, at the Recurse Center morning check-in meeting, I blithely reported that I was going to spend the next couple of days implementing a Fibonacci heap in Clojure.

I had read on the Dijkstra Wikipedia article that two men had reduced the running time of Dijkstra by inventing the Fibonacci heap and using it for node selection. Dijkstra's algorithm still did the route planning. The Fibonacci heap just helped out by quickly finding the city to explore next.

This search is a time-consuming procedure. You must go through the list of all the unexplored cities in Britain and find the one for which you have noted the shortest distance from London. When I had implemented Dijkstra in the coffee shop, my code had just gone through the whole list and returned the one with the shortest distance. This was slow.

The Fibonacci heap solves this problem in a different way. It orders the cities by their distance from London. Therefore, when you want to get the next city to explore, you just take the first one.

To explain, I must set aside the cities and roads metaphor. To my great regret, I cannot take up a new metaphor as a replacement. Heaps are not quite like family trees, nor quite like green trees that grow. So, instead, a figure:



This is an ordinary heap. The Fibonacci heap is a little more complicated, but the idea is the same. Each circle - each node - has zero or more child nodes. Further, each node includes a numerical annotation. This is its sorting value, or key. The nodes are connected into a tree where the one with the lowest key is at the root.

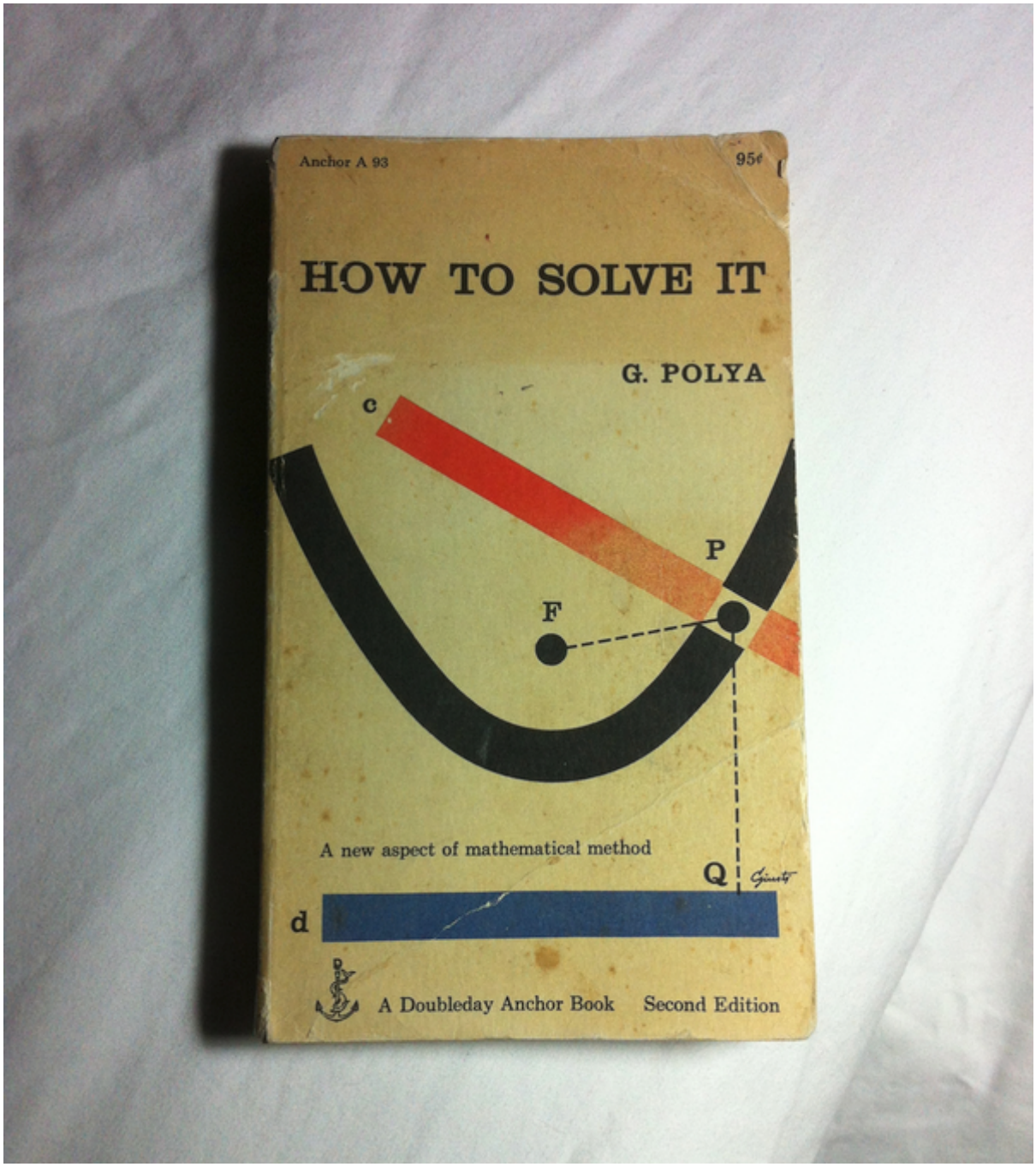
I started by producing diagrams showing examples of the core Fibonacci heap operations being applied to imaginary Fibonacci heaps. With the help of multiple reads through the Wikipedia article, I drew out each operation.

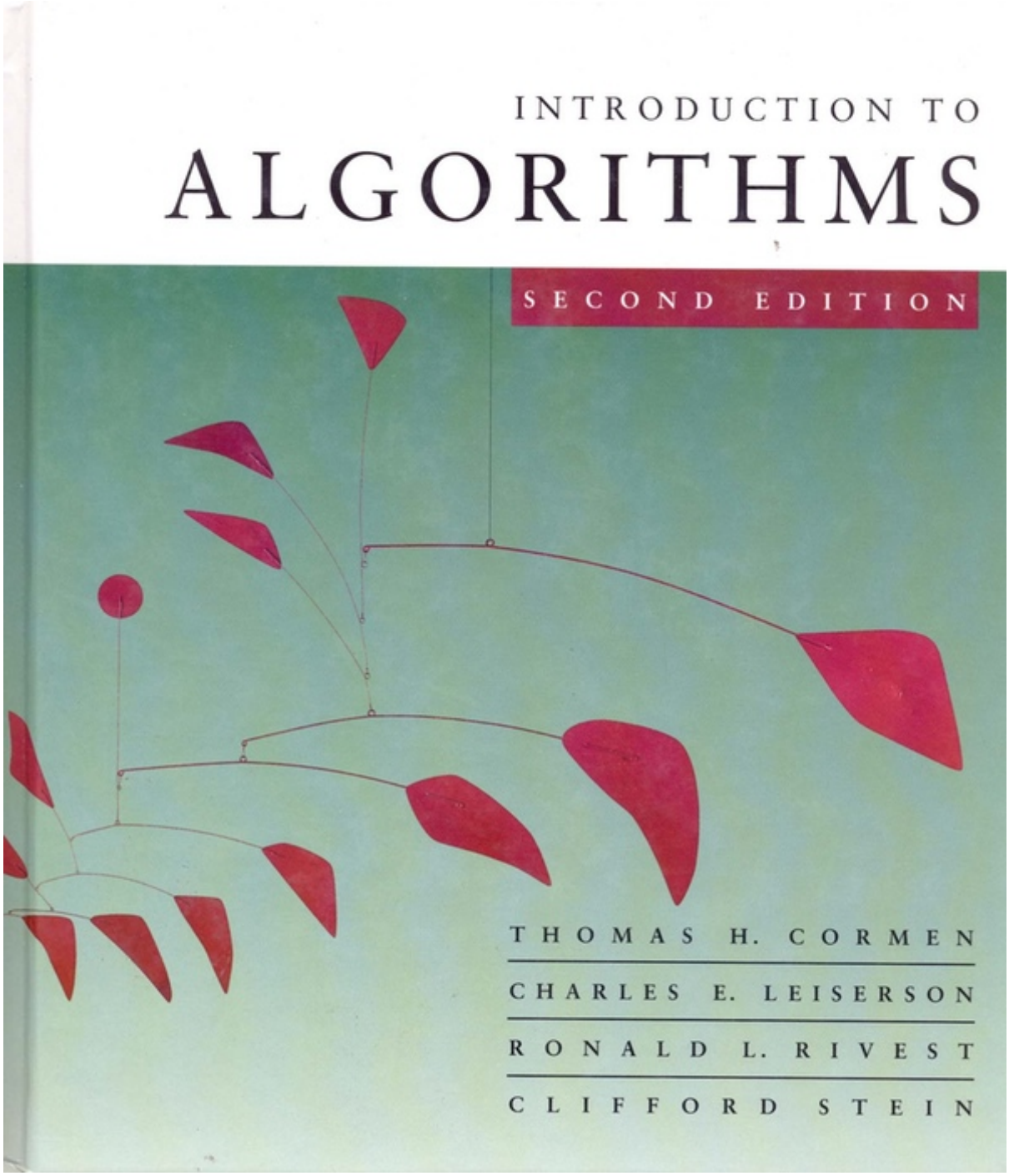
By the end of Monday, I had a forest of trees drawn out. I had mastered the algorithm and I could confidently explain it to Vera and Pepijn.

That night, as I was lying in bed, waiting to fall asleep, I realised that I was thinking about Fibonacci heaps. My downfall had begun.

I came into the Recurse Center on Tuesday and spent another day just writing in my notebook. This time, I produced rough blocks of pseudocode for the core algorithms.

In the morning, I had taken the G train to the C train to school. I'd been reading How to Solve It, a classic mathematical text that lays out an informal method for solving any problem. I had also been half thinking about Introduction to Algorithms, an exploration of algorithms for manipulating lists, trees, heaps and graphs.





It is one of the most glorious feelings to look back on a period and realise that it is suffused with what you were working on at the time. To realise that your work seeped into your journeys, your conversations, your relationships. I don't mean that you looked up at the trees rattling in the wind and saw upended Fibonacci heaps. It's much simpler than that. You were thinking about your work as you stood on the pinchingly hot pavement outside The Sunburnt Cow, as you discussed with your dad how the brush strokes for the sandstone in Roadway with Underpass somehow suggest the sparklyness of the sun. Your work was just gently present with you, providing a back story, or an invisible friend.

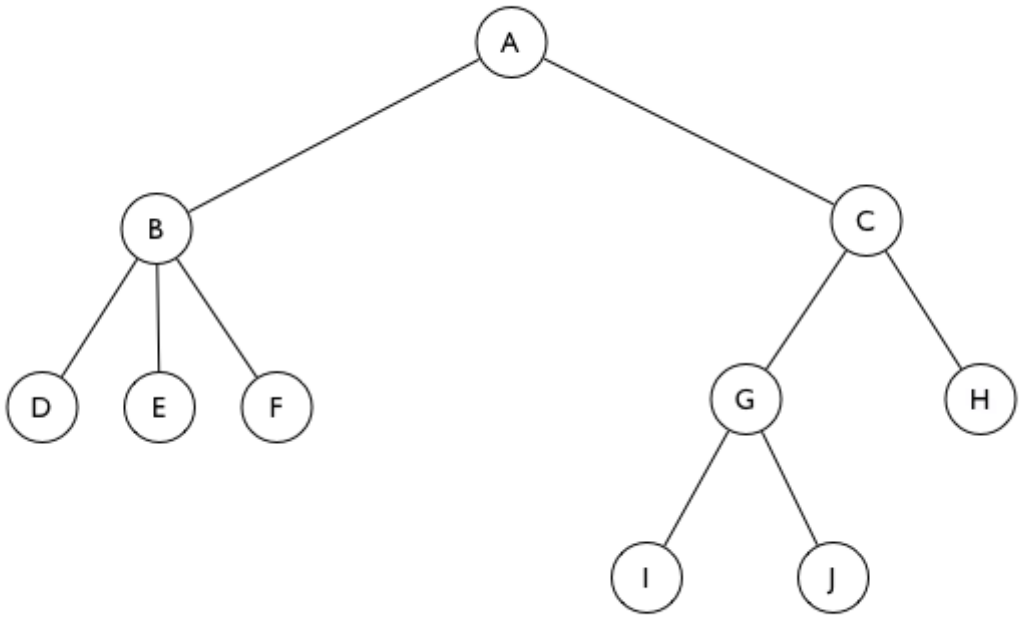
Thus, the next two weeks were steeped in maths and heaps. I thought about them in the shower. I thought about them on the G train. I thought about them in this bar that I and some other Recursers ended up in following an attempt to go to a bossa nova night after I had been disabused of the notion that bossa nova is drum-heavy salsa-ish music by a few minutes plugged into James's iPod as we waited for the train and had discovered that it is actually much lighter and more deft and sounds like a skittering barbershop quartet.

I spoke about this interleaving of code and life in my talk on Pistol Slut at JSConf US in 2011. But, this time, I worried about the code. I started sleeping badly and fretting about how long things were taking to get finished. Instead of the nasty problems being spaced out over months, they were compressed into a couple of weeks. Instead of the code being a side project, I was working on it full time. Instead of writing in JavaScript, a relatively simple language, I was writing in Clojure, tearing dense, richly meaningful lines of code out of my brain.

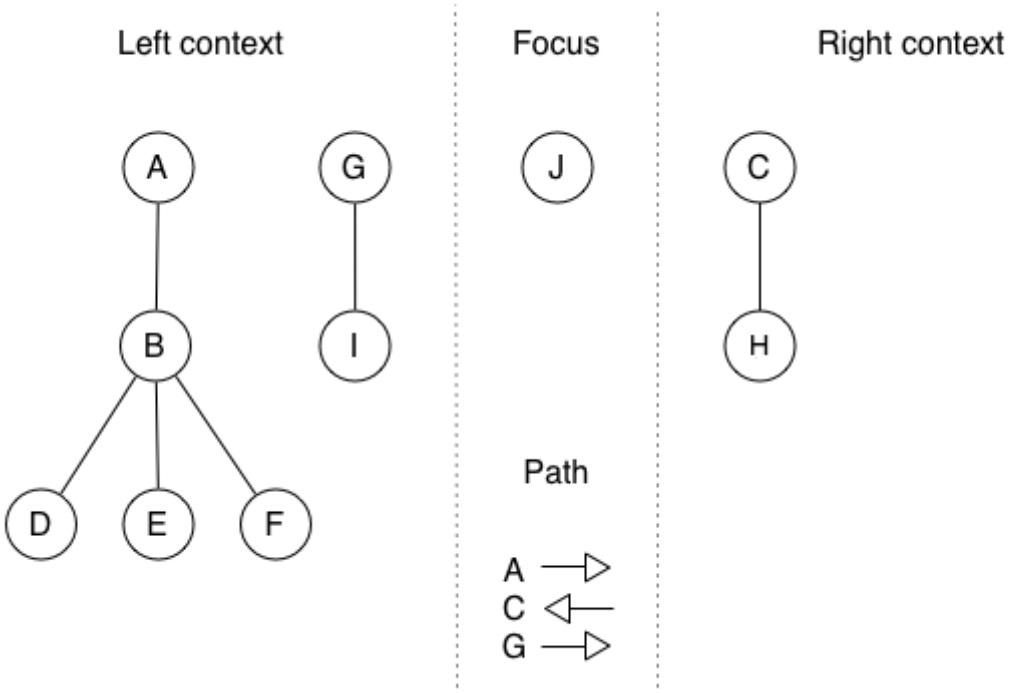
I began writing code. Immediately, I discovered that tree structures are more complicated in languages like Clojure that have immutable state. This is because changing a node requires rebuilding large parts of the tree. Imagine you want to give a new child to a node that is somewhere far down the tree. You must duplicate all

the nodes and branches that comprise the ancestory of the parent node. This is time consuming and takes up a lot of memory.

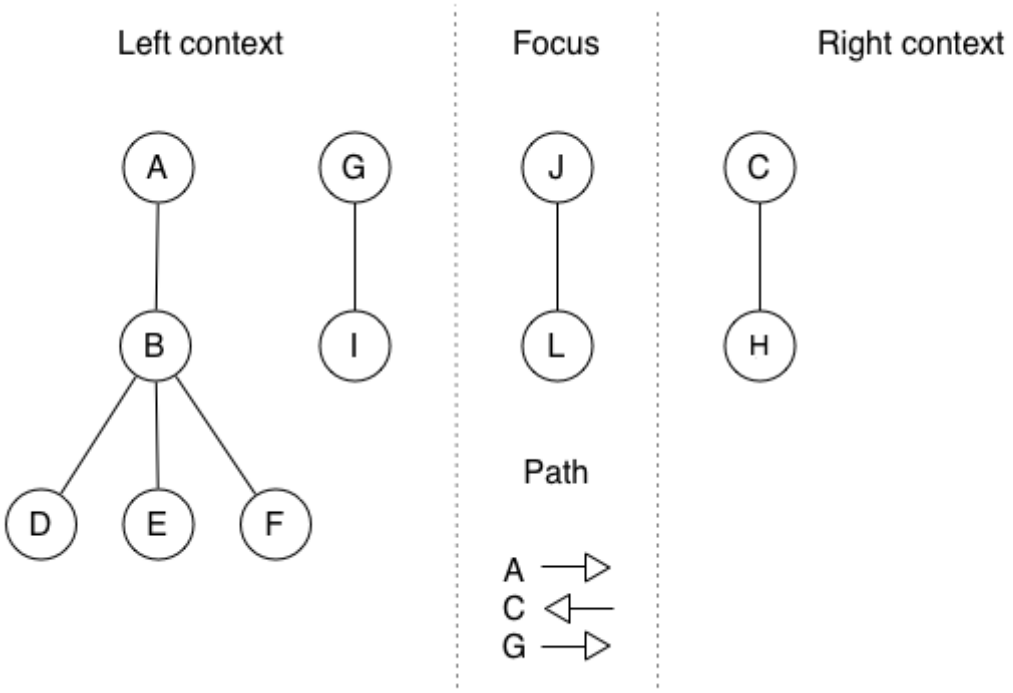
Pepijn pointed me towards a solution: a concept called the zipper. This structure expresses a tree as the node you are currently focused on, and the rest of the tree as viewed from the current node. For example, for this tree:



a zipper focused on node J would look like this:



Recall that expectant mother node. Now that the tree is modelled as a zipper, giving birth is much easier. I am going to create a new zipper that is an amalgamation of the reusable parts of the pre-child zipper and the new pieces I must create to represent the modification. The time and memory required depend on how much of the original I can use. In this case, I can reuse the path and the left and right contexts. So, the only new thing I must create is a new focus that consists of the new child, L, attached to its parent, J:

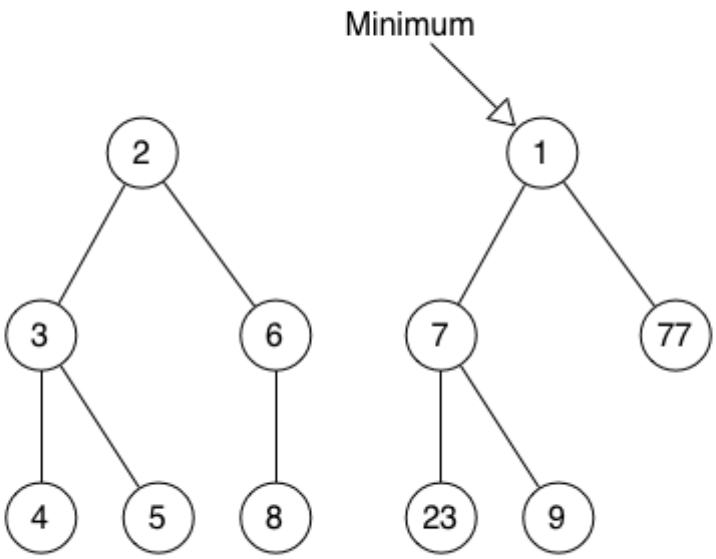


If you want to learn more about zippers, I recommend reading this excellent article, from which I adapted the previous example.

On with the story.

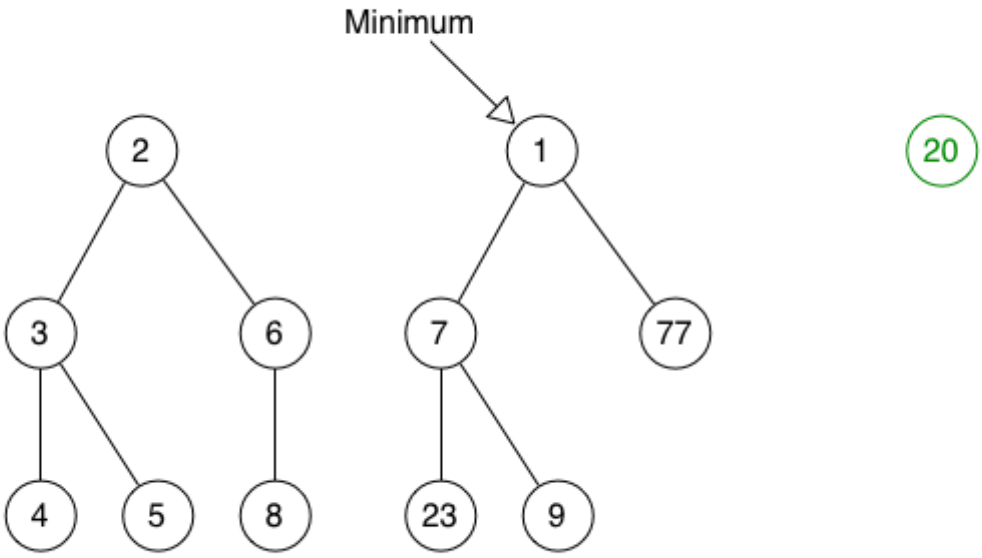
Vera and I spent the next two days buried in code. By the time the Recurse Center demo day came, we had nothing finished that we could show. We could only talk about the way the Fibonacci heap algorithm works.

Vera explained that a Fibonacci heap is not one heap (tree), but a list of self-contained sub-heaps (trees). She said there is a minimum pointer which indicates the sub-heap root node with the smallest sorting value. I can't remember if she used a metaphor of cities on a map, and if she explained that, in this metaphor, the keys on the nodes are the distances of the nodes from the starting city.



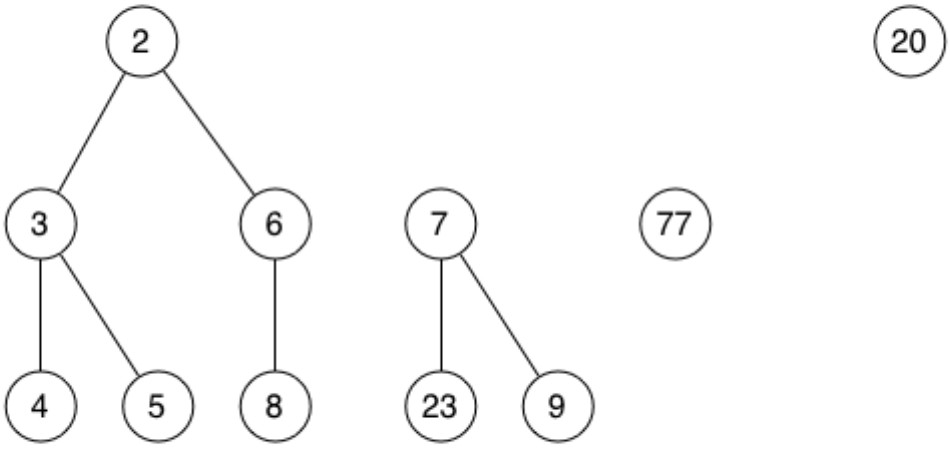
She explained the core Fibonacci heap operations.

She said that merge is the way that a new node is added to the Fibonacci heap. The node is inserted as the root of a new sub-heap in the list of sub-heaps. If the new node's sorting value is smaller than the node currently indicated by the minimum pointer, the pointer is updated.

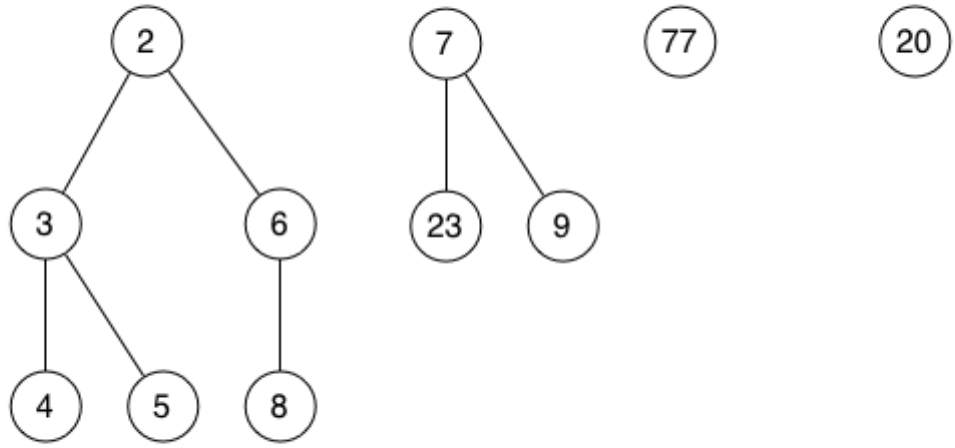


She said that find min takes the pointer to the smallest root node, follows it and returns the node itself.

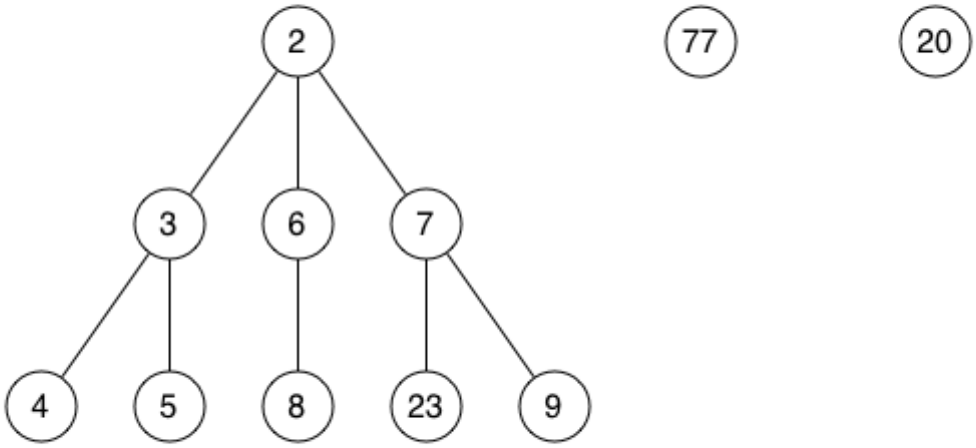
She said that extract min takes the pointer to the smallest root node, follows it and removes the node from the sub-heap to which it belongs.

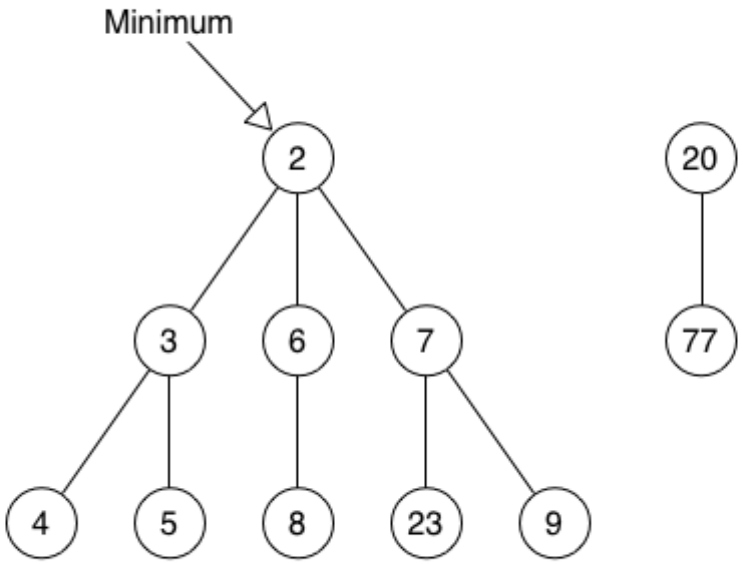


She showed how extract min tidies up by going through the former children of the node and making each one into a new stand-alone heap in the list of sub-heaps.

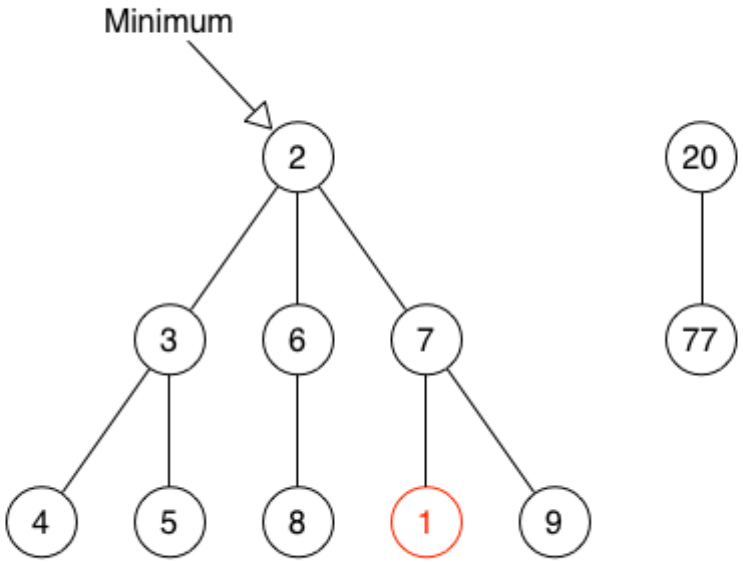


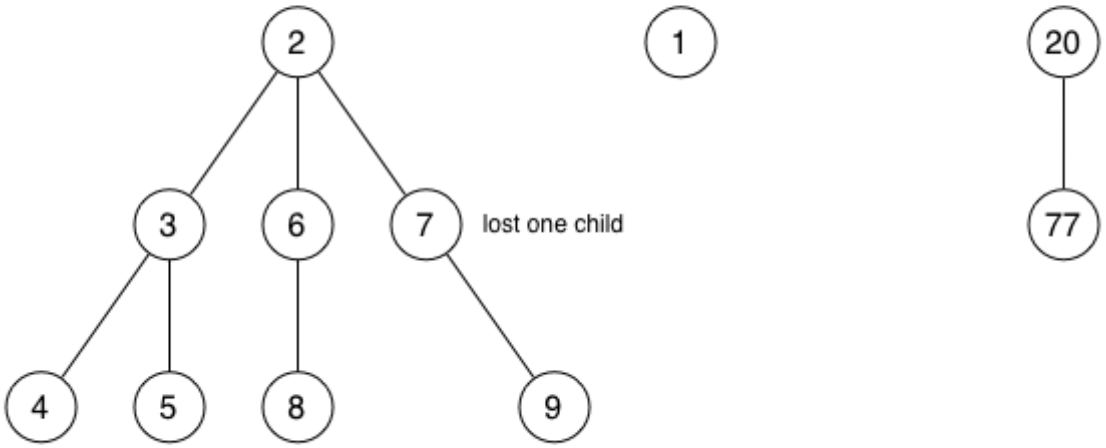
She showed how extract min tidies up further by consolidating the list of sub-heaps, combining pairs of sub-heaps that have the same number of direct descendents.



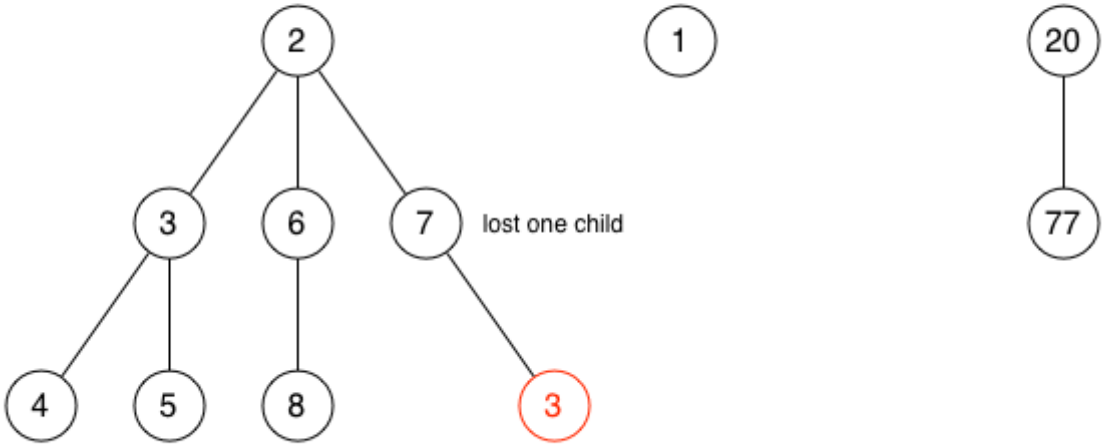


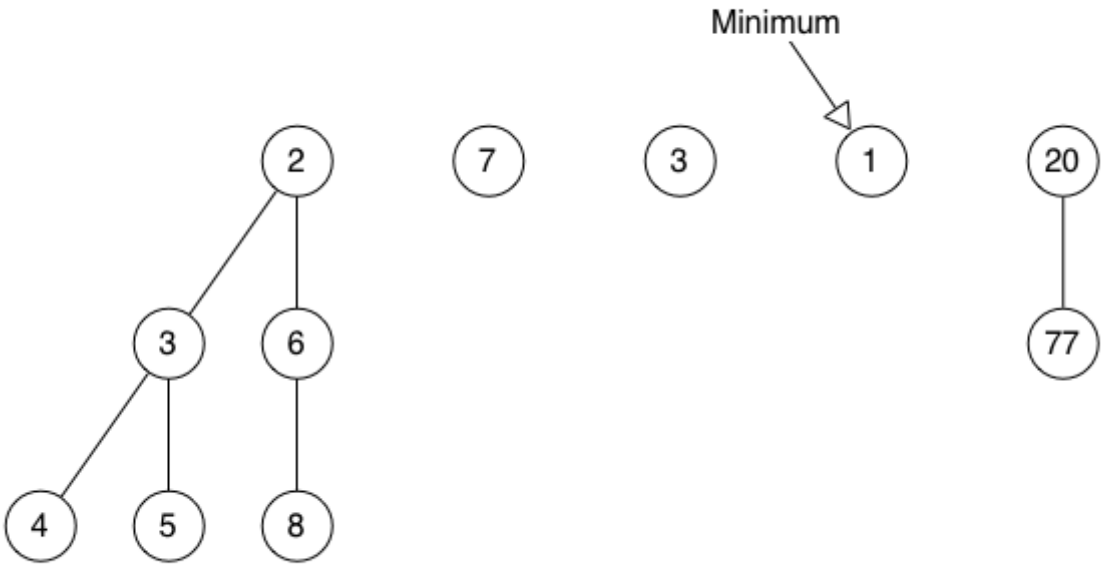
She explained how the decrease key operation works. She said that the first step is simply reducing the sorting value of a node. She said that, if this reduction makes the key smaller than the key of the node's parent, the node must be removed from the sub-heap and made into the root of a new sub-heap. She also said that the minimum pointer might need to be updated to point at a new minimum root node.





She explained that, if any node loses more than one child to a decrease key operation, it, too, must be made into a new sub-heap.





The day following, a Sunday, I spent the afternoon sat in a coffee shop in a beam of sunlight, finding some respite from the turmoil. I wrote code that was isolated from the main problem. It produces the Fibonacci heap corresponding to a terse definition of a tree structure. This made writing tests much easier, because it made it easier to produce Fibonacci heaps that reflect a scenario to be tested.

On Monday and Tuesday of week two, Vera and I dived back into the chaos and implemented decrease key, thus completing the Fibonacci heap. We worked on Saturday - demo day - to wire it into my implementation of Dijkstra's algorithm. Slowly, a horrible realisation dawned upon us. A Fibonacci heap written in a language with immutable state is, as far as we could tell then, and as far as I can tell now, incompatible with Dijkstra's algorithm. In fact, it is incompatible with any algorithm that relies on a data structure that is distinct from the Fibonacci heap, but that also shares the same information. Pepijn has written on his blog about a generalised version of this difficulty, naming it the double index problem.

In Dijkstra with Fibonacci heap, there are two data structures. First, the Fibonacci heap. Second, the graph of nodes that tells you which node is connected to which (fuck it we're going back to the map metaphor) city. Examining the neighbour of the city you are currently focusing on may neccessitate decreasing the best route distance you have stored for the neighbour. But, this is not so easy. When you get the list of neighbouring cities, you are looking at the graph data structure, which means you have graph nodes in your hands. Though these graph nodes represent the same cities as the nodes in the Fibonacci heap, they are distinct entities inside the computer. Thus, to call decrease key (distance) on a city (heap node), you must

get hold of the node that represents the city in the heap. This is a time-consuming operation that involves searching through the heap.

I'm going to stage a little mock Q&A between an imaginary version of you and the actual version of me.

You: Why is this not a problem in languages with mutable values?

Me: Because the nodes in the graph can just contain a pointer to their counterpart in the Fibonacci heap.

You: What is a pointer?

Me: It's like an arrow that gives you a direct, quick route to a computer's representation of something.

You: Why could you not use use such pointers?

Me: We could, but they would not have the desired effect.

You: Why?

Me: Because, in an immutable language, when you change a piece of data, you get a whole new copy back.

You: Gosh, it's annoying how you won't go into any detail unless I ask. Why is the whole new copy thing a problem?

Me: Sorry. It's a problem because when you update a node with a new distance, you copy that node as part of the change. Other pieces of data may have pointers to the node, but they now only have pointers to the old version. In a mutable language, you would have just updated a shared piece of data and not copied anything. Thus, all pointers to that data would have remained valid.

You: I see.

We implemented the find operation in the Fibonacci heap. This operation is given a node and finds it in the Fibonacci heap. This operation is antithetical to the point of the Fibonacci heap. When we used it, this operation was so time consuming that the original version of Dikstra's algorithm ran twice as fast as the version with the Fibonacci heap.

Showing these results at the Recurse Center demo day was kind of disappointing and kind of funny. However, both these feelings were tempered - no, obliterated - by the warm, relieved afterglow of finishing the Fibonacci heap.
