

## Study guide, 5/5-5/7

1. We looked at *leftist heaps* and *binomial heaps*, which support the `meld` operation, and contrasted them with the classic *binary heaps* that are taught in undergrad data structures classes. The binary heap consists of a complete binary tree, where all levels are full, except possibly for the last, which is left-justified.

Tarjan mentions a  $d$ -ary generalization of binary heaps, and claims on page 36 that it takes  $O(\log_d n)$  for `insert` and  $O(d \log_d n)$  for `extract-min`. Before reading about how this is accomplished, think about how you would generalize binary heaps, now that somebody has suggested the idea of a  $d$ -ary generalization where binary heaps are 2-ary heaps. Then try to derive what time bounds you get, now that  $d$  is a variable, and see if it matches these bounds.

If you don't succeed, look in the text for hints. If you do, glance through the text to verify that you are on the same wavelength.

2. Review Prim's algorithm for minimum spanning tree in Cormen. Figure out where the operation of `reduce-key` operation on a heap is needed. Which takes up more of the  $O(m \log n)$  time of Prim's when the graph is dense: the `reduce-key` operations or the `extract-min` operations? This is the **bottleneck**.

If you have a means of trading off the costs of these two operations so that their total cost is equal, you may be able to lower the asymptotic cost of the bottleneck without going to far and raising the cost of calls to the other operation above the new bottleneck.

Your analysis above of  $d$ -ary heaps provides just such a tradeoff! Where should you set  $d$ ? You should try to figure this out, then peek at the bottom of page 77 for the answer. Notice the surprising result that it gives when the graph is *dense*, which we'll define to be when  $m = n^{1+\epsilon}$  for  $\epsilon > 0$ .

This trick of trading off one large cost for another smaller cost is called *balancing*, and it deserves to be considered a generic algorithm-design trick.

3. Here's a puzzler involving balancing. Some of you have seen this one from CS420.

You have a ladder with  $n$  rungs and you have two identical jars. You want to figure out the highest rung from which you can drop the jars without breaking them. You can break both jars in the process. Because you are lazy, you want to minimize the number of times you drop a jar.

Show that you can get an  $O(\sqrt{n})$  bound on the number of drops. Use this to get even better bounds if you have three jars, four jars. After  $\log_2 n$  jars more jars don't help; you're doing the equivalent of binary search at that point.

We will see another clever example of balancing in Tarjan's treatment of minimum spanning trees.

4. Where should you set  $d$  if  $m = \Theta(n^2)$ , for instance, when the graph is complete? What surprising result does this give about the value of a heap for Prim's algorithm in this case?

5. Look at the *red rule* and the *blue rule* on page 71. You are familiar with the blue rule from Cormen (and probably from a data structures course). Figure out what you know from those courses that's being nicknamed the the blue rule here.

The red rule is one that Karen suggested in class on 4/30. It is proved with the same type of greedy proof technique that is taught in classes like CS420, which almost always use minimum spanning trees as an illustration.

Tarjan doesn't address what time bound you can get for the red rule, but think about how you could get a competitive bound using linking/cutting trees. It's a variant on Problem 2 of this week's homework.

6. Read up to the pseudocode of Section 6.3, referring to Boruvka's coloring step on page 73 for comparison. Let me save you some time by summarizing the pseudocode. Set up a queue of trees, which is initially a queue of  $n$  trees, each with one vertex. While there is more than one tree, remove the front tree  $T_1$ , find a minimum-weight edge  $e$  from a vertex of that tree to a vertex outside of  $T_1$ . Call the tree that contains this vertex  $T_2$ . Remove  $T_1$  and  $T_2$  from the queue, turn them into a single tree by adding  $e$  to them, and insert the new tree to the back of the queue.

The pseudocode uses meldable heaps to find  $e$ . Instead of this, consider the totally naive approach of finding  $e$  by exhaustively searching the adjacency lists of all vertices of  $T_1$ . Identifying that an edge goes to a different tree requires that the vertices be labeled with an identifier that tells their tree. Therefore, we must say how to update these labels when  $T_1$  and  $T_2$  are merged. You may be tempted to cleverly relabel the vertices of the smaller tree to update these labels. Let's be more naive than that for a moment, and relabel all the vertices of both  $T_1$  and  $T_2$  with a new identifier for the new tree.

Figure out why, in spite of our lack of clumsiness, we get an  $O(m \log n)$  time bound. This is as good as what Cormen claims for Prim's and Kruskal's, which use clever data structures!

Once you have understood these issues, you will be in a position to see why, if we can get as good a bound without even trying, we can get a better bound by applying some clever data structures, and the balancing trick.