

**Disclaimer:** *These are rough notes, with some exercises.*

Preparation for this lecture was mostly from Karger's advanced algorithms scribe notes.

## 6.1 Shortest path, and heaps.

**Question:** Give a graph  $G = (V, E)$ , and a node  $s$ , how do you compute the shortest path from  $s$  to all the other nodes.

Start by setting a label,  $d(\cdot)$  to 0 for  $s$  and  $\infty$  for all other nodes. Place 0 in a priority queue whose priority is the value of  $d(\cdot)$ .

While the queue is not empty: remove a node from the queue set the value of each neighbor  $v$  of  $u$ , to

$$d(v) = \min(d(u) + c(u, v), d(v)).$$

**Question:** For MST?

Change the setting of  $d(v)$  to the following

$$d(v) = \min(c(u, v), d(v)).$$

**Question:** How do you implement the priority queue?

A heap.

**Question:** What is the heap property for a tree?

The parent should have a lower value for the key than all of its children.

**Question:** What operations does the queue need to support.

Decrease-key, delete-min.

**Question:** What is the running time for a binary tree implementation of these operations?

$O(\log n)$  for both.

**Question:** What is the running time of the SP algorithm.

$n$  times the running time of delete-min,  $m$  times the running time of Decrease-key.

**Question:** Degree  $d$  implementation.

$O(\log_d n)$  for Decrease-key.  $O(d \log dn)$  for delete-min.

**Question: What is the best tradeoff?**

$d = m/n$ , Running time becomes  $O(m \log_{m/n} n)$ .

**Question: Can we get a better heap?**

Maybe?

## 6.2 Fibonacci heaps?

**Question: Amortized analysis of Data Structure, what is it?**

Some operations may be expensive, but they are cheap “on average”; that is over a sequence of  $n$  operations, the total cost is bounded.

**Question: Fibonacci heap times?**

$O(m + n \log n)$  time for  $m$  decrease-keys and  $n$  inserts, delete-mins.

**Question: How is this carried out?**

Often, a “potential” function is associated with the state of the data structure. And each operation is charged for the average amount of work, contributes an average amount of work to the potential, and perhaps uses up some perhaps arbitrary amount of potential. The key is that when it uses a lot of potential there must have been a lot there from previous operations.

**Question: Huh?**

Huh?

**Question: The fibonacci heap consists of a list of heap ordered trees. Where is the minimum element of each heap?**

At its root.

**Question: Add an element?**

Add to root list.

**Question: Find and delete minimum element?**

Keep pointer to minimum root element. Find is easy. Delete it, put children in root list.

**Question: We make the rank of a node be its number of children. We only allow one root of each rank in the list. What is an upper bound on the number of nodes in list?**

Maximum rank of any node.

**Question: How do you maintain this property when delete min, insert?**

Merge nodes (with same number of children) in list. One points to the other. Swap elts, as necessary.

**Question: So far, how many children for a degree  $d$  node?**

Well,  $d$  children. One has  $d - 1$  children, one has  $d - 2$  children and so on.

## Exercise 1: Show that at depth $i$ , one has $\binom{d}{i}$ descendants

Thus, this has been called the binomial tree.

**Question: The maximum rank?**

The number of nodes in a degree  $d$ , tree is  $\sum_i \binom{d}{i} = 2^d$ . Thus,  $d < \log n$ .

**Question: Decrease key?**

$O(\log n)$  the old way.

**Question: Another way?**

Cut the node out, make it a root. Can still find the min, the merging with equal rank node.

**Question: How long is the time?**

Constant for the cut. But then the merging may percolate.

**Question: On average how much do you pay over cuts. What?**

For each decrease-key, we allocate some potential (say 10 bucks) whenever we create a new rank  $d$  tree by a cut (or by an insert or by a delete min where all the children are cut.) We assume the invariant that each bucket in the top level data structure has banked 10 bucks. Now a merge takes two rank  $d$  trees, and creates one rank  $d + 1$  tree. We charge 10 bucks for this operation, the tree we created still has 10 bucks associated with it. Thus, we maintain the invariant that each bucket has 10 bucks associated with it.

**Question: What are our total deposits?**

Moreover, the total deposits is 10 bucks times the number of decrease-key operations + 10 bucks times the number of inserts + 10 bucks times the number of new trees created by delete mins. This amounts to  $O(m + n + n \times \max \text{rank})$ .

**Question: Are the total deposits an upper bound on work?**

Yes.

**Question: Problem?**

Trees may no longer contain many children. This may lead to rank growing to much larger than  $O(\log n)$

**Question: What do we do?**

If parent loses a second child, cut parent, too. (Here I made a joke, about losing children that is perhaps better to not be committed to the written word.)

For each node, maintain a mark, if parent is marked cut it as well. Unmark it.

**Question: How much time?**

The number of "unmarks."

**Question: Remember potential for state of the data structure. A mark is cheap, doing extra work, unmarks a node. What should we associate potential to?**

The marked nodes. We can add potential (perhaps 10 bucks) when we mark and take out when we unmark (and pay for cuts.)

**Question: How does this help the degree argument?**

Well, the nodes have degree within one of their rank.

**Question: Are we done?**

We need to show the rank remains large. In particular, we can have the following property.

**Lemma 6.1** *The  $i$ th child added to a node has rank at least  $i - 2$ .*

**Question: Why?**

We prove this by noting that it had degree at least  $i - 1$  when it was merged, since we only merge equal rank nodes. Furthermore, it can only lose one unit of degree, before it is cut itself.

**Question: Let  $S_i$  be the number of descendants degree  $i$  node. How many nodes in  $S_0, S_1$  or  $S_k$ ?**

We will assume that one is one's own descendant.  $S_0 = 1, S_1 \geq 2$ .

$$S_k \geq \sum_{i=0}^{k-2} S_i.$$

**Question: What is this at least?**

The  $k$ th fibonacci number! That is,  $\geq 1.5^k$ . Thus,  $k = O(\log n)$ .

**Question: Can we do better?**

No. Could sort is less than  $O(n \log n)$  comparisons.

**Question: Can we do better?**

Yes, maybe don't only use comparisons.

## 6.3 Van Emde de Boas queue

**Question: Say the maximum number is  $C$ . Is there an algorithm for doing shortest path priority queue operations in  $n + m + C$ ?**

Sure for shortest path delete is always on a bigger number, have an array of size  $C$ . Put element in bucket corresponding to value. Scan to find min. Total scanning is  $C$ . (OK for shortest path, not necessarily for MST.)

**Question: Say the maximum number is  $C$ . Is there an algorithm for doing priority queue operations in  $(n + m)\sqrt{C}$ ?**

Sure, have an array of size  $\sqrt{C}$ . For an element, look at its higher half of its bits. That specifies one of  $\sqrt{C}$  buckets. In that bucket, put another array of size  $\sqrt{C}$  and place the element in the bucket according to the lower half of its bits.

For decrease-key, delete item. Re-insert. For delete-min, scan to find next. Total scanning is  $\sqrt{C} + n\sqrt{C}$  since one needs to scan all of the high level buckets and the non-empty low level buckets.

**Question: Can we use recursion to do better? Let's try one more level.**

We can continue to divide the low level buckets, and get,  $\sqrt{C} + nC^{1/4}$ . This seems stuck. Because, we need to scan the whole thing.

**Question: Must we scan to find lowest high level bucket?**

Not necessarily. We can maintain a data structure for finding that as well.

So, now we will maintain the following. An array of size  $\sqrt{C}$  (to be indexed by the high level word of an element) containing queues on low level word of an element, and a queue on the high order half-word with range up to  $\sqrt{C}$ . We also maintain a pointer on the minimum element.

Well, for delete-min, looks ok. We have a pointer to the minimum element bucket. Find the min on the data structure for that bucket, delete it. If this queue is now empty, delete it from higher level data structure and find the new min for the higher level data structure.

**Question: Running time?**

Let  $b$  be  $\log C$ . It takes  $O(1)$  to find the correct lower level datastructure. If the lower level data structure will become empty, it takes  $O(1)$  time to delete the element (why?), and then we have to delete an item in the higher level data structure, which takes  $T(b/2)$  time. On the other hand if the lower level data structure is not empty, we don't have to touch the higher one, and we only pay for an operation on the lower one, i.e.,  $T(b/2)$ . Either way, we get the following recurrence.

$$T(b) = T(b/2) + O(1)$$

This solves to  $O(\log b) = O(\log \log C)$ .

**Question: That worried look on my face in class?**

I must admit I got worried in class. I made the terrible mistake of unrolling a recurrence and confusing myself. But, perhaps a word or two. Here the deletion in the case the queue gets empty means the minimum element is directly pointed to, and is not stored recursively. And to check if the rest is empty does not need to invoke anything recursively (its min is empty.) This to me is suprising. It seems to just be putting off the pain by doing a "special" case for one item. But it does work. Who would have thought?

**Exercise 2: Describe how to implement the operation, Find(x), in the van Emde Boas queue.**