This material takes 1:15

1 Heaps

Shortest path/MST motivation. Discuss Prim/Dijkstra algorithm.
Note: lots more decrease-key than delete.
Response: *balancing*

- trade off costs of operations
- making different parts equal time.

d-heaps:

- $m \log_d n + nd \log_d n$.
- set $d = m/n$
- $O(m \log_{m/n} n)$

1.1 Fibonacci Heaps

Fredman-Tarjan, JACM 34(3) 1987.
`http://www.acm.org/pubs/citations/journals/jacm/1987-34-3/p596-fredman/`
Key principles:

- Lazy: don't work till you must
- If you must work, use your work to "simplify" data structure too
- force user (adversary) to spend lots of time to make you work
- analysis via potential function measuring "complexity" of structure.
  - user has to do lots of insertions to raise potential,
  - so you can spread cost of complex ops over many insertions.
  - potential says how much work *adversary* has done that you haven't had to deal with yet.
  - You can charge your work against that work.
- another perspective: procrastinate. if you don't do the work, might never need to.
- Why the name? Wait and see.

Lazy approach:

- During insertion, do minimum, i.e. nothing.
- For first delete-min, cost is n

- So, amortized cost 1.
- Problem with second and further delete mins
- n delete mins cost n^2—means amortized n

Use your work to simplify

- As do comparisons, remember outcomes
- point from loser to winner
- creates "heap ordered tree" (HOT)
- might not be full or balanced, but heap ordered
- now you take out the root, so get set of HOTs
- next time, min is among roots of HOTs—less work to find
- eg, if build perfect binary tree, just need to check 2 children
- problem: can't control tree shapes
- problem: may get star, next delete min loses all useful info
- problem: additional insertions add more things to inspect next time

Summary/Goals

- Maintain set of HOTs, with pointer to min root
- Formalize notion that scan through inserted items is "paid for" by consolidation
- Devise mechanism so not too many additional trees added by removal of min

Heap ordered trees implementation

- definition
- represent using left-child, parent, and sibling pointers (both directions)
- keep double linked list of HOTs
- and a pointer to min HOT root.
- so in constant time, can find min
- in constant time, can add item
- in contant time, can link two HOT lists (Fibonacci heaps are *mergeable* in constant time)

- time to delete-min equal to number of roots, and simplifies struct.

Solution to 'star' problem: use heap-ordered trees, but keep degree small!

- method: ensure that any node has descendant count exponential in degree.
- So max degree $O(\log n)$ for n items
- how?
  - bucket HOTs by degree (maintain degree in node)
  - only link HOTs of same degree
  - same "union by rank" idea as union find
  - start at smallest bucket; link pairs till < 2 left. next bucket.
- lemma: if only link heaps of same degree, than any degree-d heap has 2^d nodes.
- creates "binomial trees" (draw)
- "Binomial heaps" do this aggressively—when delete items, split up trees to preserve exact tree shapes.

Insert solution idea: adversary has to do many insertions to make consolidation expensive.

- analysis: **potential function** ϕ equal to number of roots.
  - $\text{amortized}_i = \text{real}_i + \phi_i - \phi_{i-1}$
  - then $\sum a_i = \sum r_i + \phi_n - \phi_0$
  - upper bounds real cost if $\phi_n \geq \phi_0$.
  - sufficient that $\phi_n \geq 0$ and ϕ_0 fixed
- insertion real cost 1, potential cost 1. total 2.
- deletion: take r roots and add c children, then do $r + c$ scan work.
- r roots at start, $\log n$ roots at end. So, $r - \log n$ potential decrease
- so, total work $O(c + \log n) = O(\log n)$

Result: constant insert, $O(\log n)$ amortized delete (plus $O(1)$ insert).
What about decrease-key?

- basically easy: cut off node from parent, make root.
- constant time decrease-key
- problem: may violate exponential-in-degree property
- "saving private ryan"

- fix: if a node loses more than one child, cut it from parent, make it a root (adds 1 to root potential—ok).
- implement using "mark bit" in node if has lost 1 child (clear when becomes root)
- may cause "cascading cut" until reach unmarked node
- why 2 children? We'll see.

Analysis: must show

- cascading cuts "free"
- tree size is exponential in degree

Second potential function: number of mark bits.

- if cascading cut hits r nodes, clears r mark bits
- adds 1 mark bit where stops
- amortized cost: $O(1)$ per decrease key
- note: if cut without marking, couldn't pay for cascade!
  - this is binomial heaps approach. may do same $O(\log n)$ consolidation and cutting over and over.
- Wait, problem
  - new root per cut
  - adds to first potential function
  - and thus to amortized cost
  - fix: double new potential function.
  - use one unit to pay for cut, one to pay for increase in 1st potential
  - so, doesn't harm first potential function analysis

Analysis of tree size:

- node x. consider *current* children in order were added.
- claim: i^{th} remaining child (in addition order) has degree at least $i - 2$
- proof:
  - Let y be i^{th} added child
  - When added, the $i-1$ items preceding it in the add-order were already there
  - i.e., x had degree $\geq i - 1$

  - So i^{th} child y had (same) degree $\geq i - 1$
  - y could lose only 1 child before getting cut

- let S_k be minimum number of descendants (inc self) of degree k node. Deduce $S_0 = 1$, $S_1 = 2$, and

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

- to upper bound, solve equality

$$\begin{aligned} S_k &= \sum_{i=0}^{k-2} S_i \\ S_k - S_{k-1} &= S_{k-2} \end{aligned}$$

- deduce $S_k \geq F_{k+2}$ fibonacci numbers
- reason for name
- we know $F_k \geq \phi^k$

Practical?

- non-amortized versions with same bounds exist (good for realtime apps).
- Constants not that bad
- ie fib heaps reduces comparisons on moderate sized problems
- but, regular heaps are in an array
- fib heaps use lots of pointer manipulations
- lose locality of reference, mess up cache.
- work on "implicit heaps" that don't use pointers

1.2 Minimum Spanning Tree

minimum spanning tree (and shortest path) easy in $O(m + n \log n)$.
More sophisticated MST:

- why $n \log n$? Because deleting from size-n heap
- idea: keep heap small to reduce cost.
  - choose a parameter k
  - run prim till region has k neighbors
  - set aside and start over elsewhere.

  - heap size bounded by k, delete by $\log k$
  - "contract" regions (a la Kruskal) and start over.

Formal:

- phase starts with t vertices.
- set $k = 2^{2m/t}$.
- unmark all vertices and repeat following
  - choose unmarked vertex
  - Prim until attach to marked vertex or heap reaches size k
  - ie k edges incident on current region
  - mark all vertices in region
- contract graph in $O(m)$ time and repeat

Analysis:

- time for phase: m decrease keys, t delete-mins from size-k heaps, so $O(m + t \log k) = O(m)$.
- number of phases:
  - At end of phase, each compressed vertex "owns" k edges (one or both endpoints)
  - so next number of vertices $t' \leq 2m/k$
  - so $k' = 2^{2m/t'} \geq 2^k$
  - when reach $k = n$, done (last pass)
  - number of phases: $\beta(m,n) = \min\{i \mid \log^{(i)} n \leq 2m/n\} \leq \log^* n$.

Remarks:

- subsequently improved to $O(m \log \beta(m,n))$ using edge packets
- chazelle recently improved to $O(m\alpha(n) \log \alpha(n))$ using "error-prone heaps"
- ramachandran gave optimal algorithm (runtime not clear)
- randomization gives linear.
- fails for Dijkstra