Adv. Data Structures

Fibonacci Heaps

Fibonacci Heap: the motivation

Recall: Heaps support operations:

	Heap	Binomial heap :)
getMin	$O(1)$	~~$O(\log n)$~~ $\to O(1)$ (w/pointer)*
insert	$O(\log_2 n)$	$O(\log_2 n)$ & $O(1)$ amortized if n inserts
removeMin	$O(\log_2 n)$	$O(\log_2 n)$
decreaseKey	$O(\log_2 n)$	$O(\log_2 n)$
delete	$O(\log_2 n)$	$O(\log_2 n)$
union/merge	$O(n)$	$O(\log_2 n)$

(*adds overhead to others, but only $O(1)$)

However, there is a major algorithm that needs decreaseKey often: graphs!

Ex: Dijkstra on a graph $G=(V,E)$

Loop:

Keeps a current best distance to each vertex (initially $s=0$, other $v=\infty$)

Pops min off heap, & updates all vertices that now have a better path

decreaseKey! (lots of times)

n items on heap

n^2 changes

So a new goal:
Improve decreaseKey, by whatever means necessary!

A first attempt:

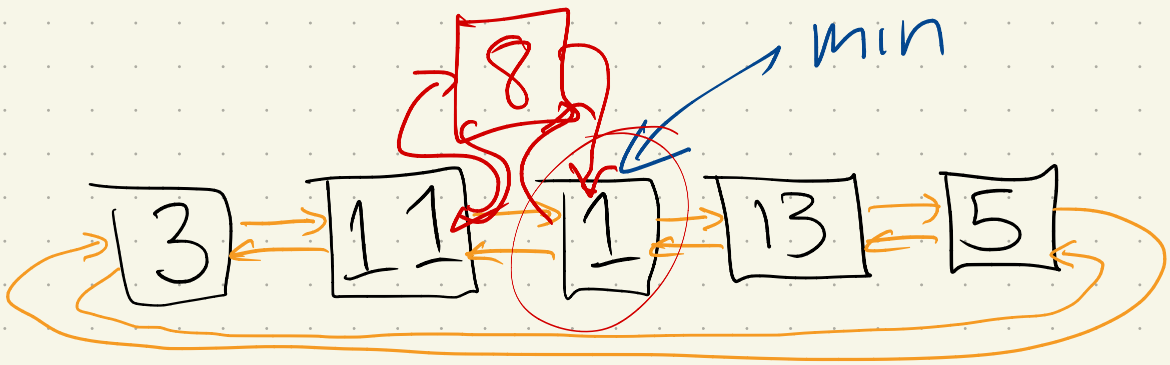
What if we just used:
- min
- insert
- merge

were $O(\log n)$ — $O(1)$ or $\log_2 n$

(Missing: decreaseKey, deleteMin + delete)

Could we use something simpler than heaps?

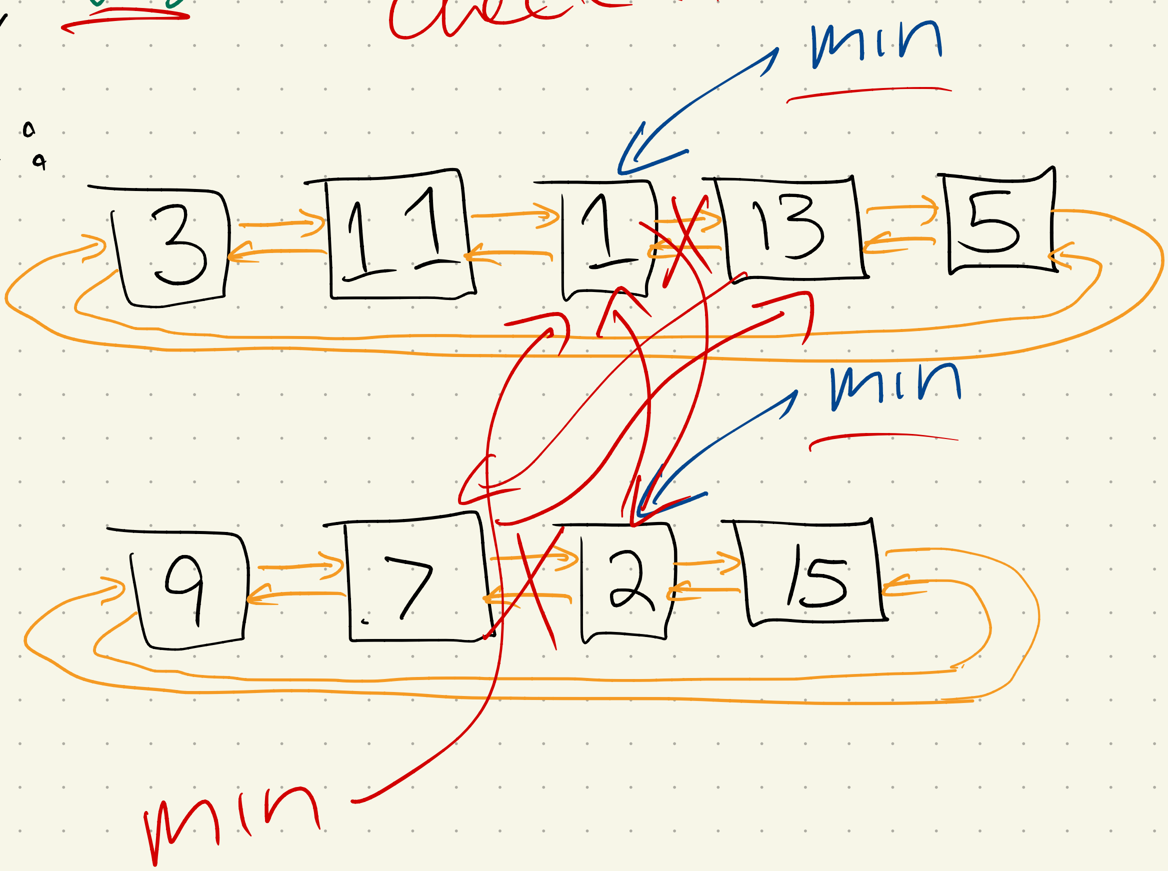
Yes! $O(1)$



Min: O(1) – follow global ptr

Insert (8): splice into list
check if new min

Union/Merge: O(1)



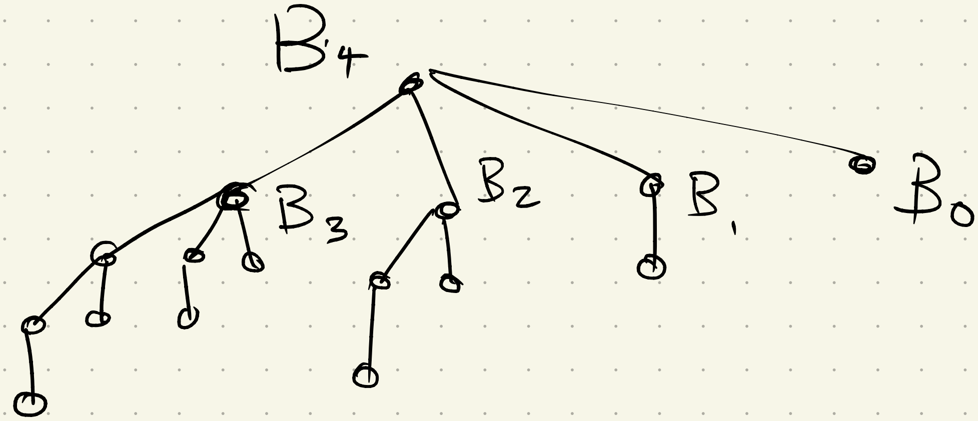
But: I want decreaseKey!

Recall: Binomial heap

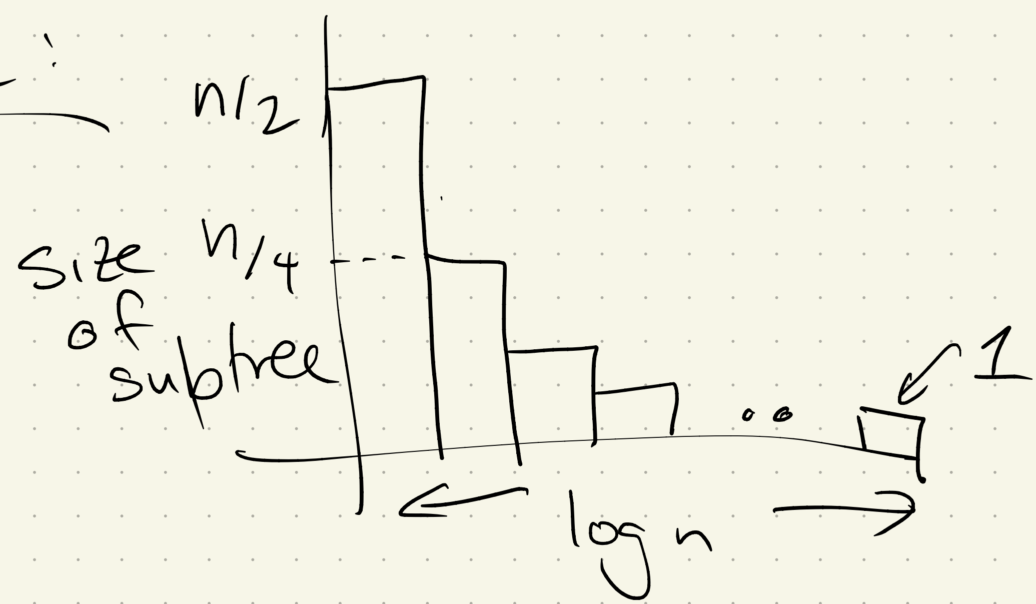
list of binomial trees
(≤ 1 of each order)

Ex

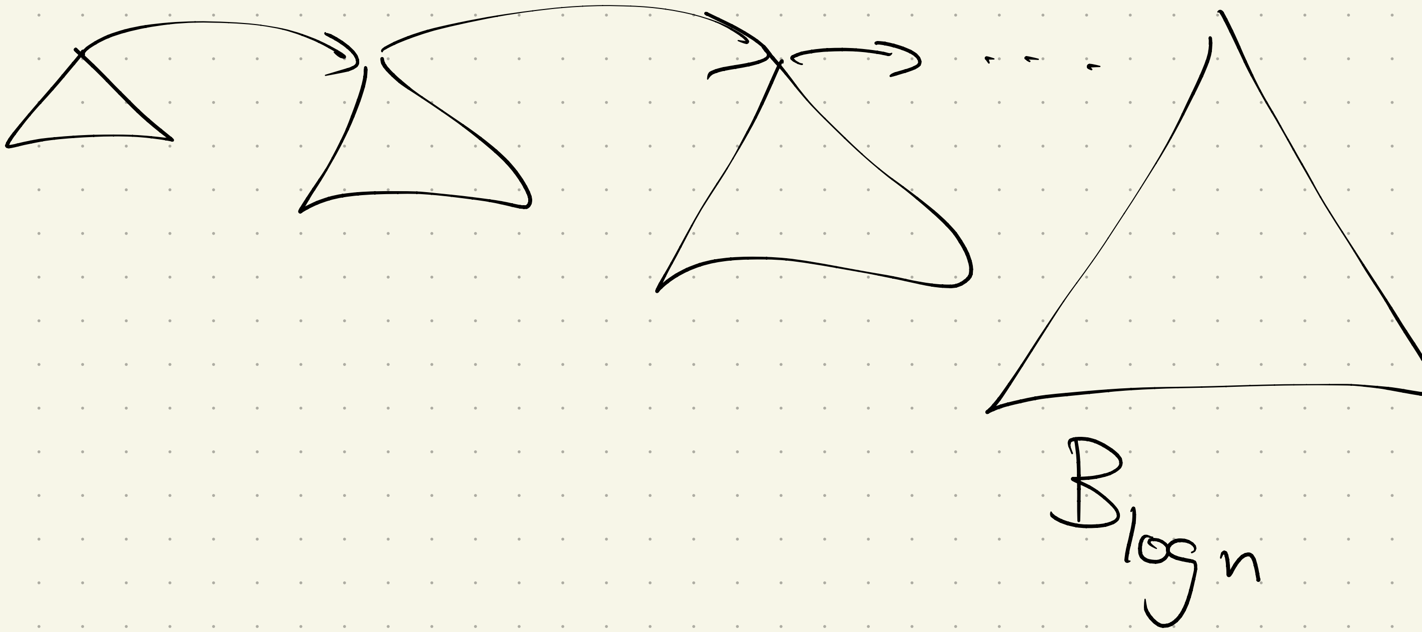
binomial tree of order 4:



Note:



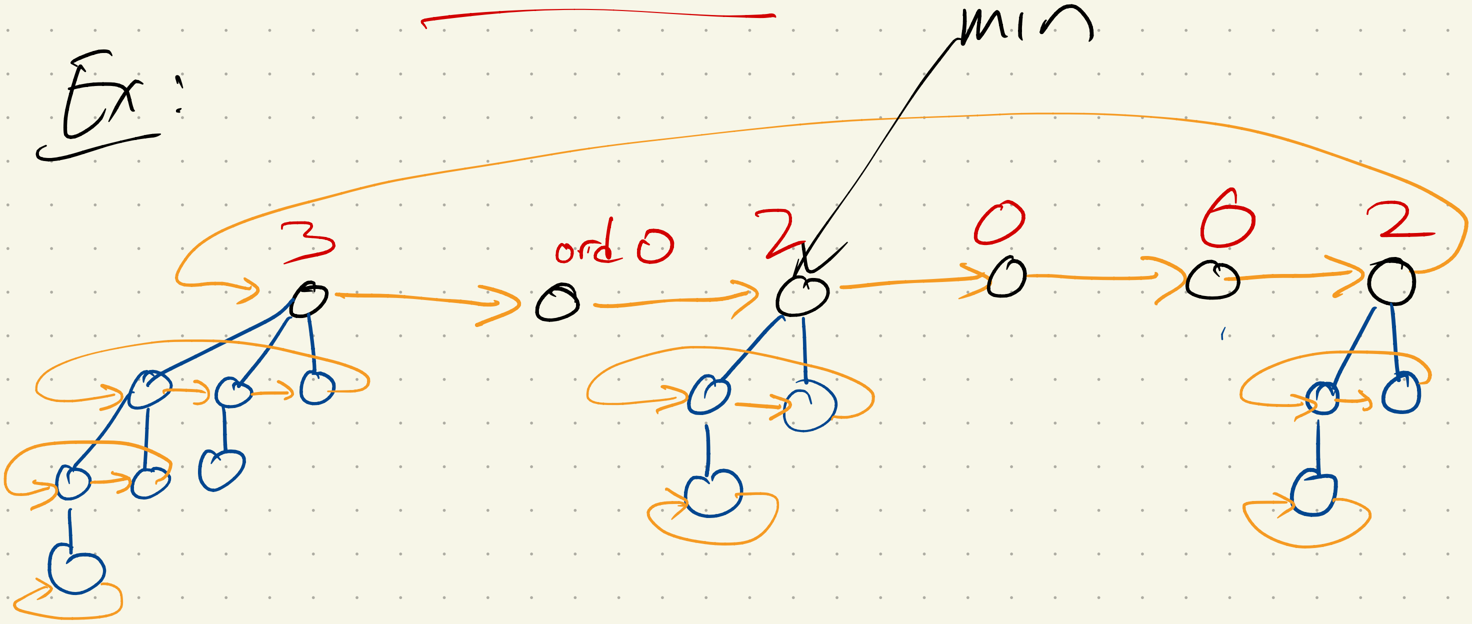
A binomial heap:

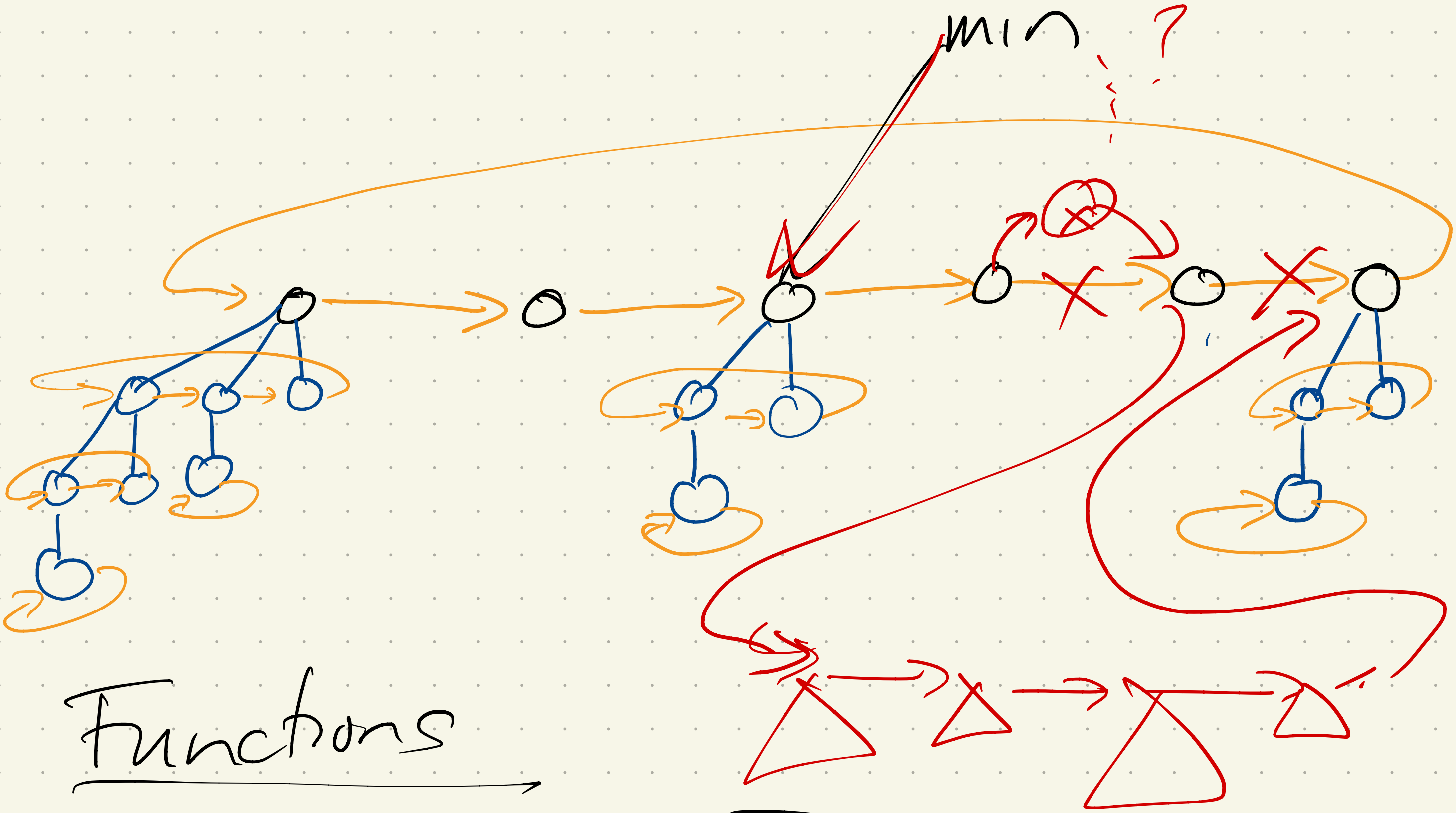


Fibonacci heap:

relax the structure, & allow >1 of each in no set order

Ex:





Functions

- insert (x)
- getMin
- merge/union

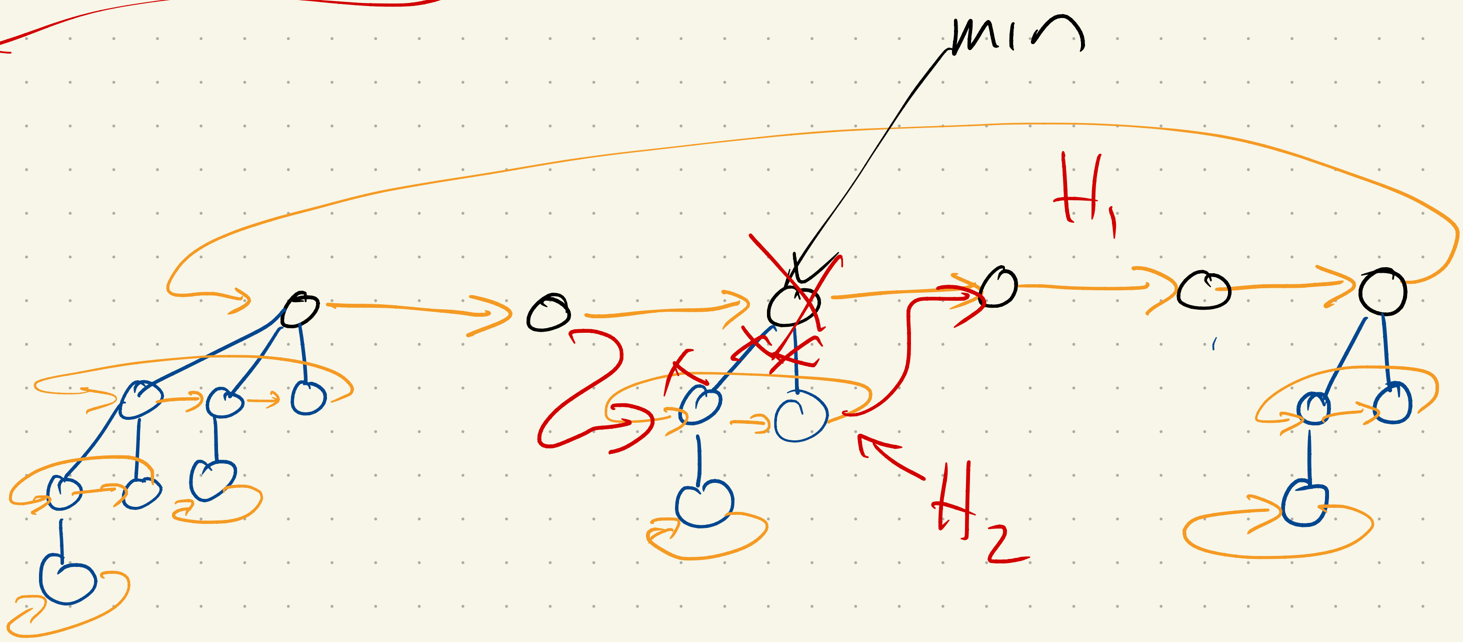
easy!

Bit harder:

- deleteMin
- decreaseKey

avoid/beat $\rightsquigarrow$ $O(n)$

DeleteMin(): pay for laziness!



- Delete the min
- Set child's parent ptr to NULL

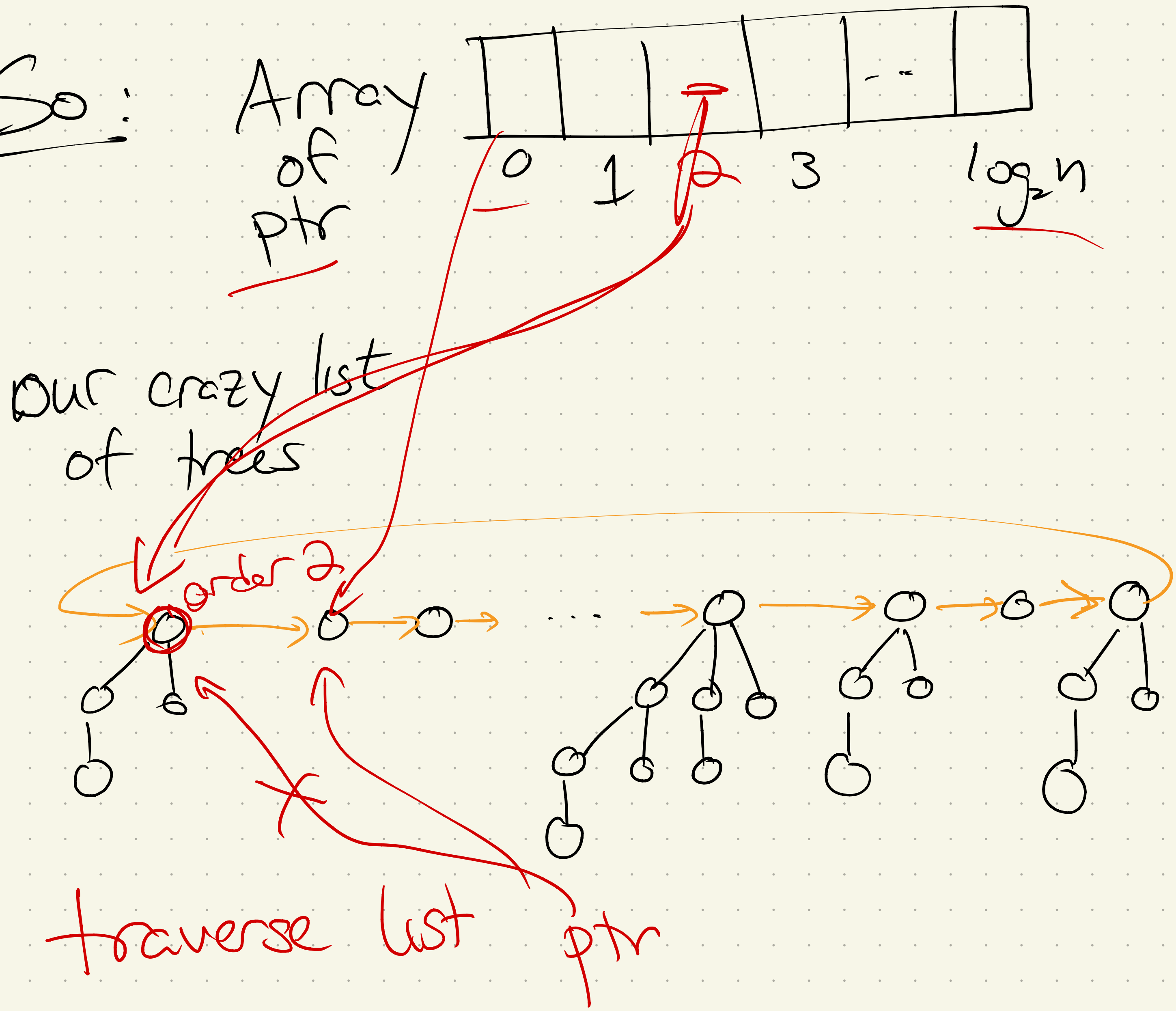
  ⇒ Now 2 Fib. heaps
- Link the two lists,

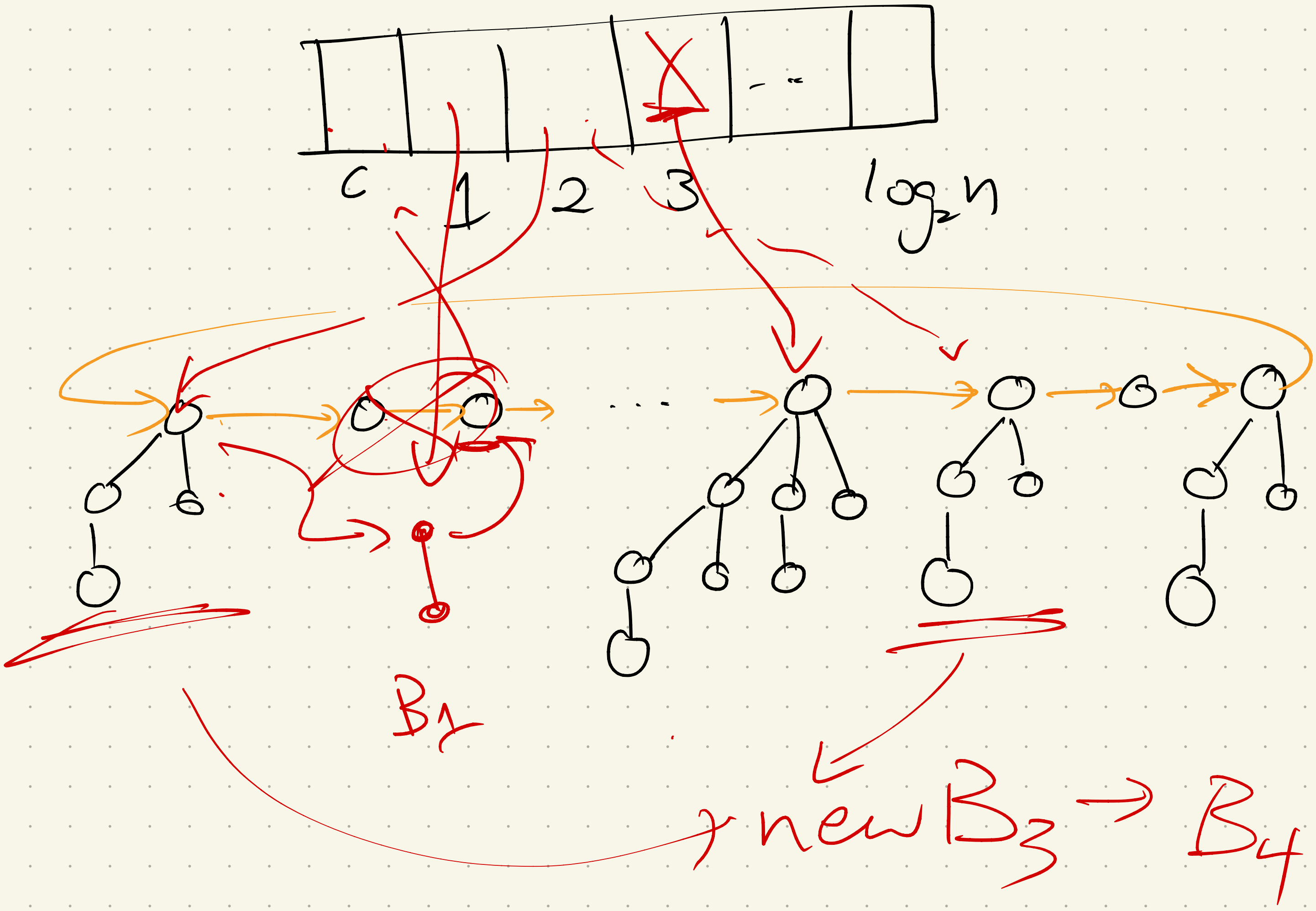
  (Now lots of bin. trees, unordered)
- Sweep + clean up, so ≤1 of each size → ☞

Sweep + clean up, so
≤ 1 of each size

Like union/merge, but worse!
(No ≤ 3, not in order)

So: Array of ptr





- traverse root list (& update min if needed)
- if conflict, merge & update

12x

Runtime: $O(\text{size of list before delete}) + O(\log n)$

Why?

- Traverse list

- Merge some # of times

each list item could get merged once

each node can be at most depth $\log_2 n$

Amortized analysis:

- root list size starts = 0 & reset to $\log_2 n$ after deleteMin call.
- with each insert: gets worse

so → have insert pay +1

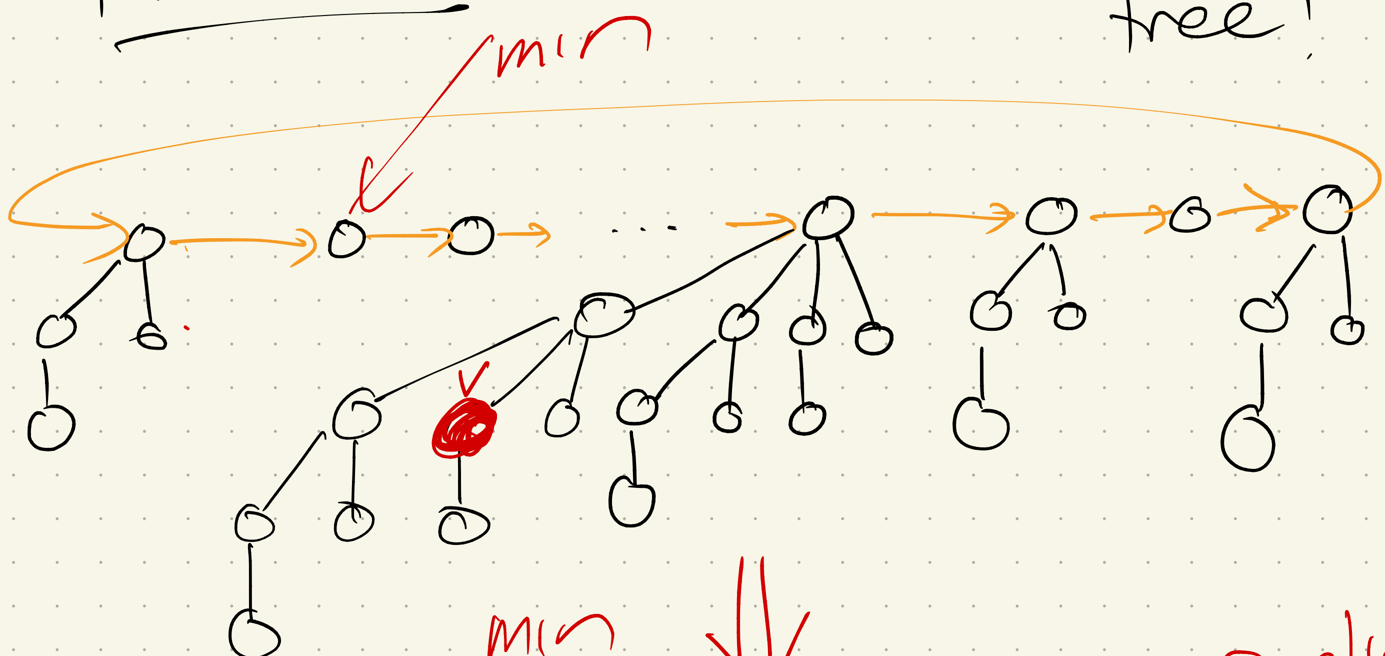
⟹ $O(\log_2 n)$ amortized

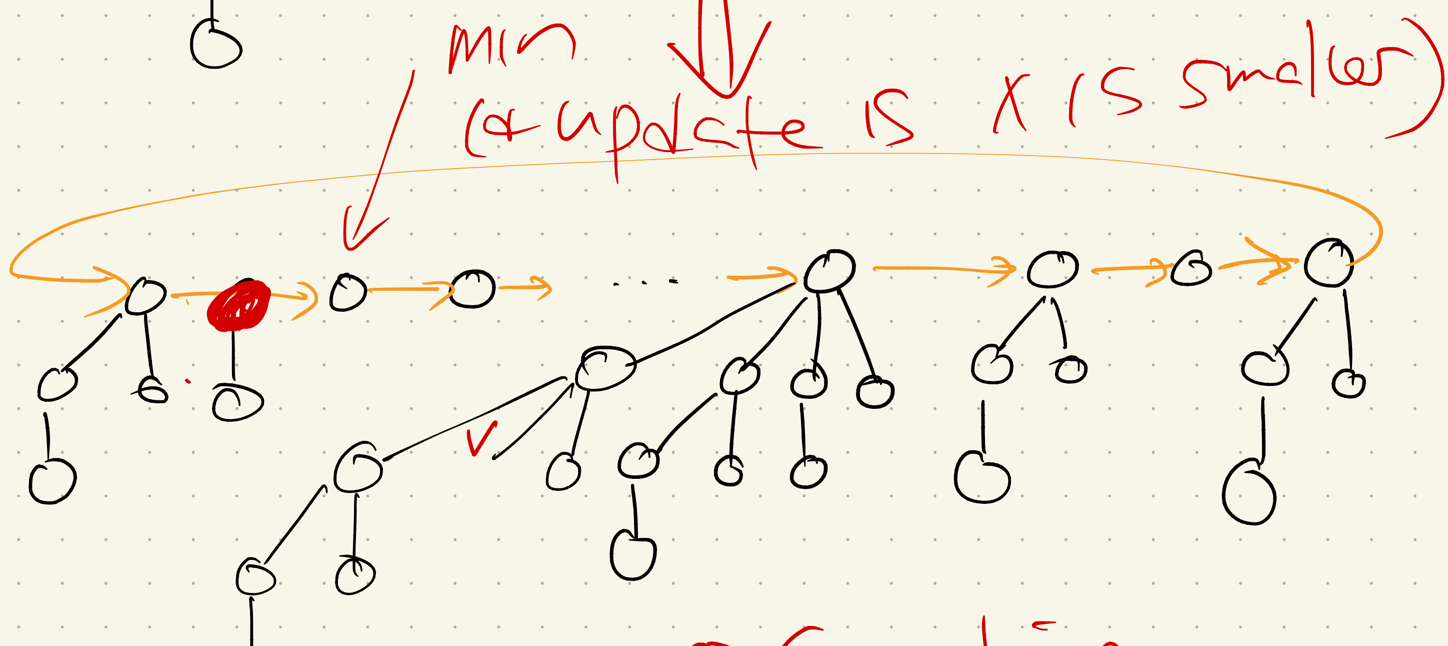
(Details in posted refs)

decreaseKey fcn:

If v's value decreases, move it to root list.

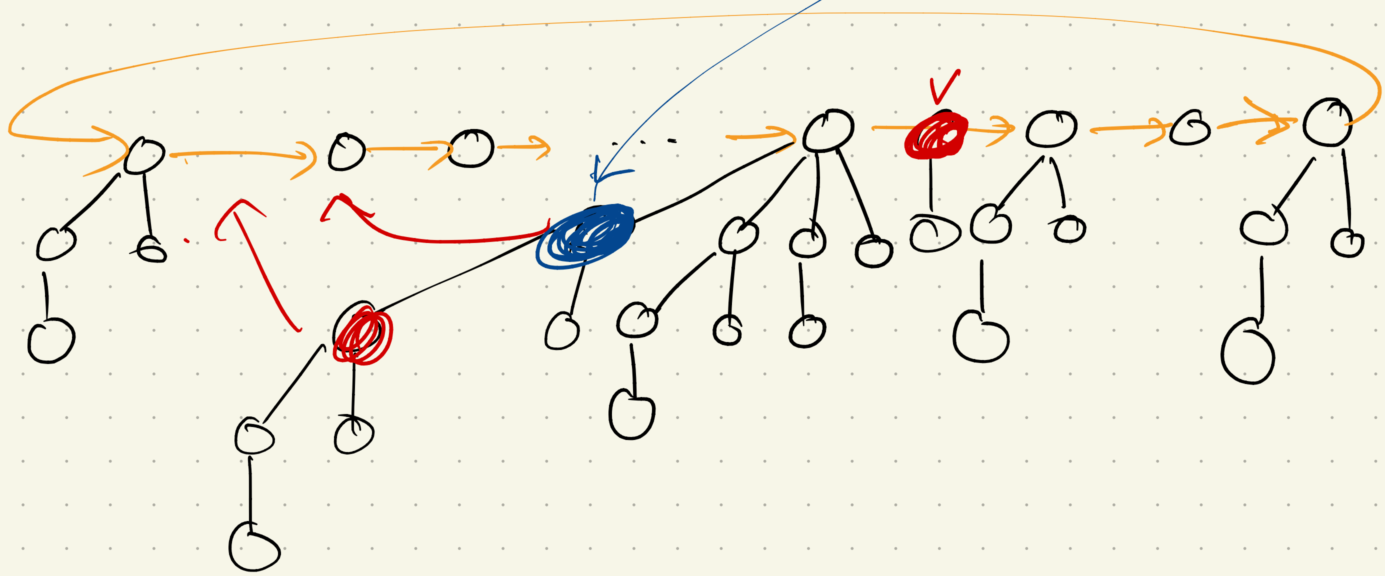
Problem: Not a binomial tree!





O(1) time

So: If parent of v is unmarked, mark it



If marked, move parent's subtree to root & unmark

Move up tree (& move as long as marked)

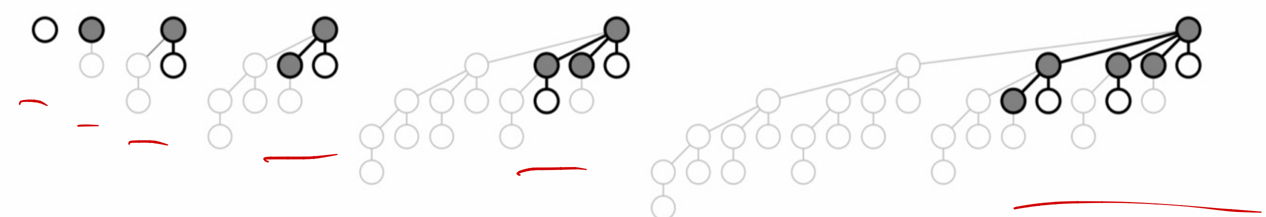
Runtime: O(1 + (# marked ancestors in a row))

= O(depth(v))

Problem: Not a binomial tree list!

If it were, depth $\approx \log n$

Here, parts could be missing:



Fibonacci trees of order 1 through 6. Light nodes have been promoted away; dark nodes are marked.

"Worst case"

Most we can remove without triggering cascade of promotions.

(Note: Fibonacci #'s!)

Potential function $\Phi(v) =$ # marked consecutive ancestors of v

When promote one, +1

When promote k, −k

time for op = $t + \Phi(v)$

after op, overall potential goes down by $-(\Phi(v)+1)$

$\boxed{\Rightarrow O(1) \text{ amortized cost}}$

(again, see notes for careful proof)

Result:

$$\left.\begin{array}{l}\text{Min}\\ \text{Insert}\\ \text{Merge/Union}\end{array}\right\} O(1)$$

DecreaseKey : $O(n)$ wort
$O(1)$ amortized

deleteMin : $O(n)$ worst
$O(\log n)$ amortize