Complexity & Algorithms, Spring 2026

Lots of heaps

Abstract data types, "template"

+ data structure design:

high level, + focused on operations → insert, delete, ...

→ implementations

language specific

Recall: Priority Queues

An abstract data type that contains objects, each having a key.

Operations supported:

- insert(obj)
- find_min()
- delete_min()

In other words, more limited than a search tree!

Aside: or find Max, delete Max

Could implement with a BST ~>

How to implement?

With linked list:

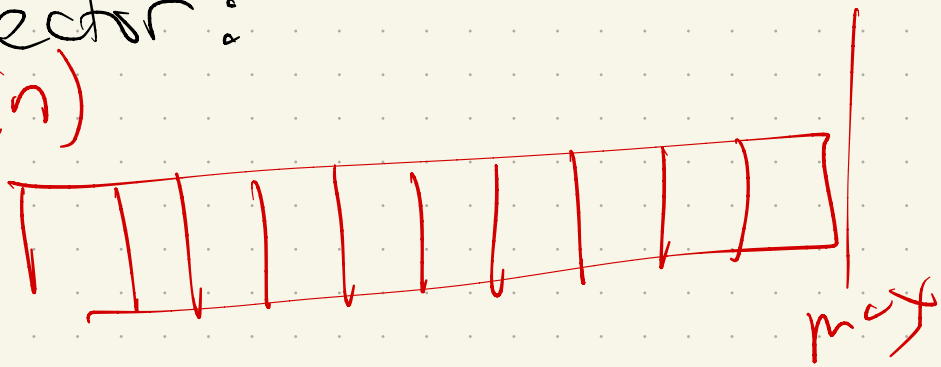
insert: easy → O(1)

getMin: sweep: ~~O(n)~~ O(1) | option: min ptr

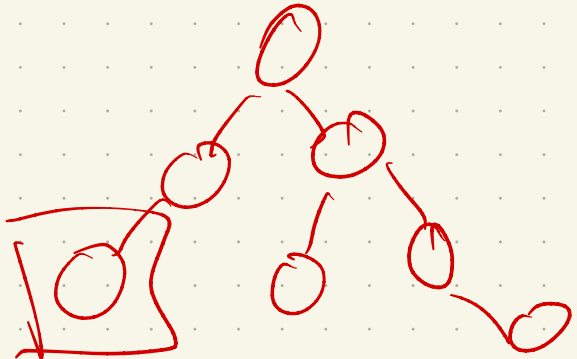
deleteMin: O(n) O(1) | min

With Vector:

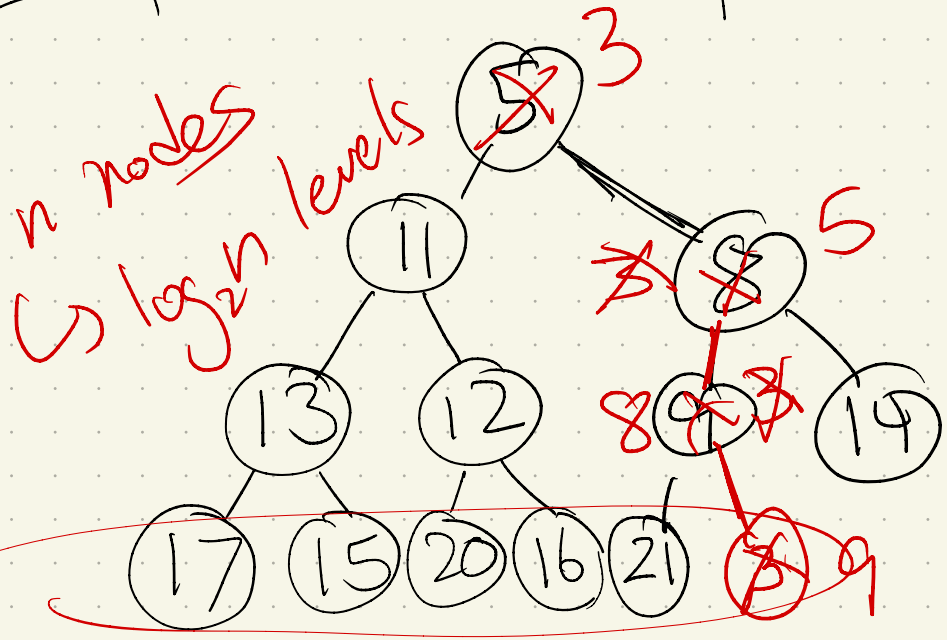
insert: O(n)



With BST: insert, delete, find: O(log n)



Heaps: one implementation, keeping "perfect" binary tree!



n nodes
↳ $\log_2 n$ levels

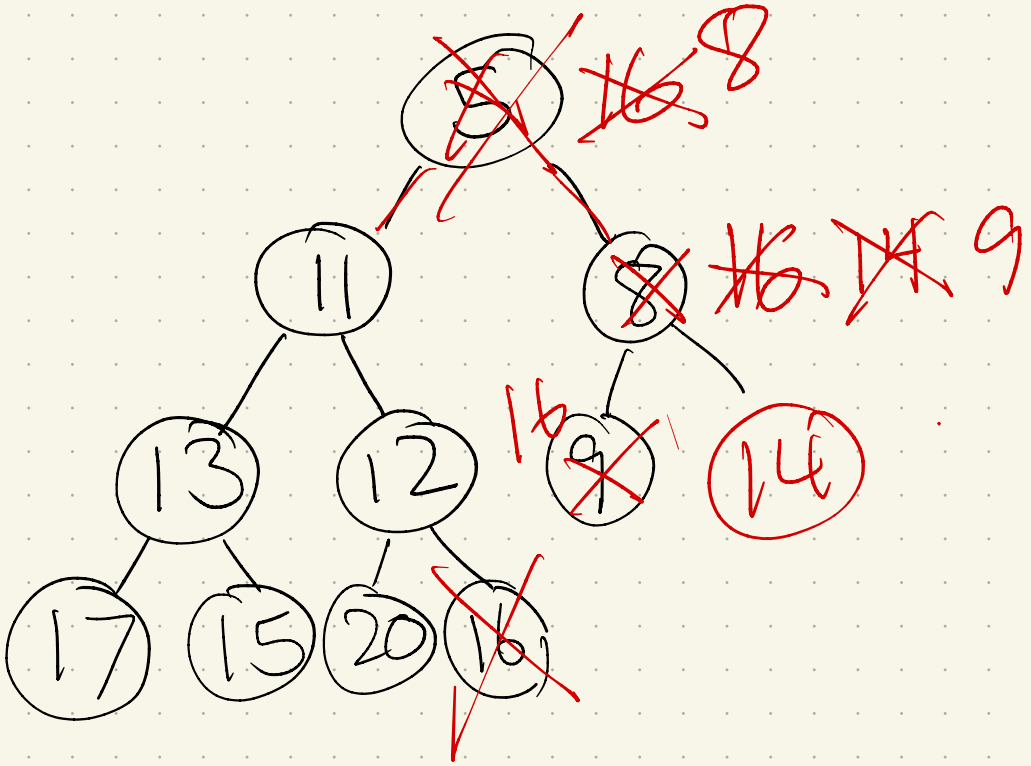
binary tree where all leaves are on same level

(-1)

$\lceil \log_2 n \rceil$

Insert (3):

adds node in "next" spot
re-order as need in loop
until it is a heap again

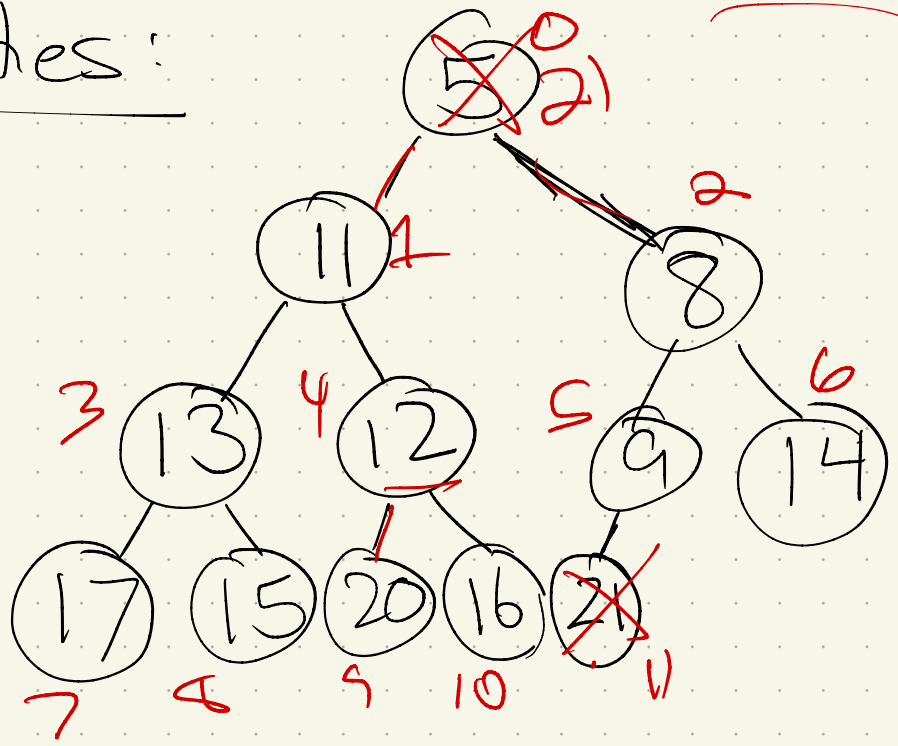


DeleteMin():

$O(\log n)$

Nice Properties:

Number vertices:



Height:

$\lceil \log_2 n \rceil$

Layout/Space:

Left(v) = $2v+1$

Right(r) = $2v+2$

parent(v) = $\lfloor \frac{v}{2} \rfloor - 1$

Adding operations:

Often a heap also supports: decreaseKey

Runtime: $O(\log_2 n)$

↳ use insert + delete "bubble" operations

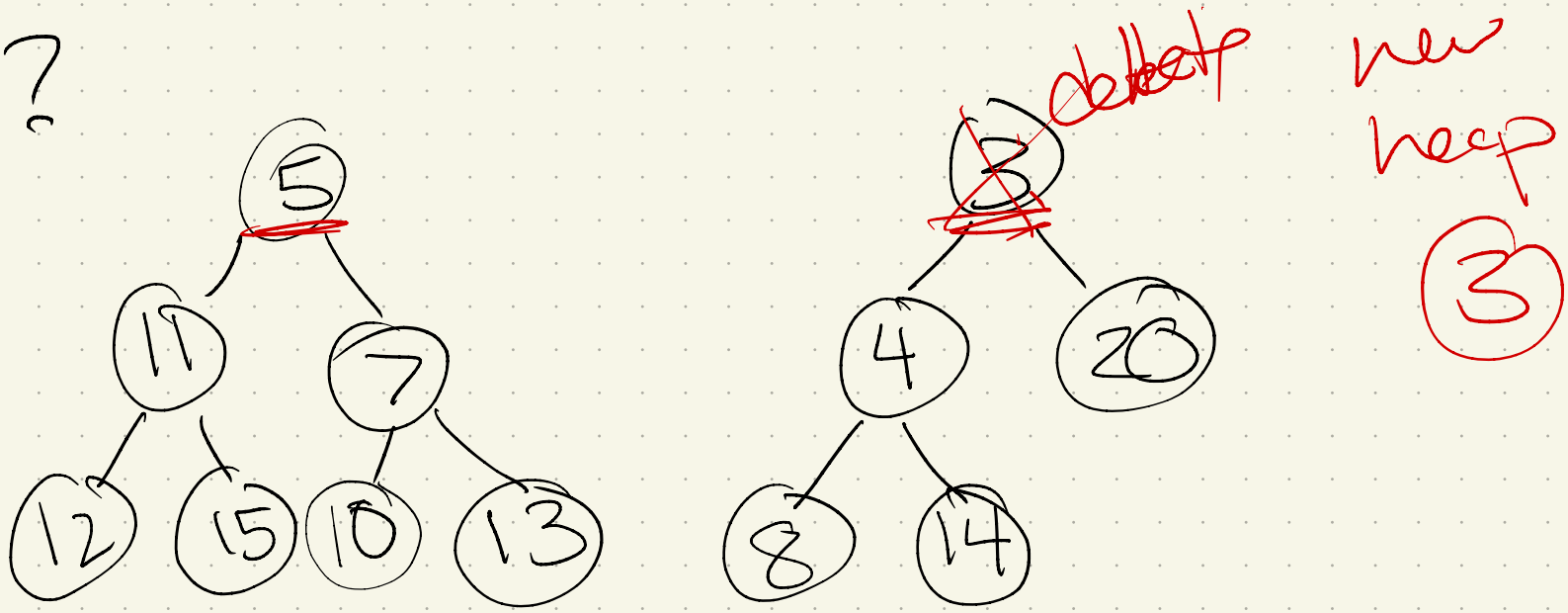
Note that this also gives us delete:

use other operations

Another: Merge(H_1, H_2):

create a new heap with all values of H_1 & H_2

How?



Runtime: $O(n \log n)$

→ $O(n)$ possible

Binomial Heap

Goal: Improve Merge

$O(n) \rightarrow O(\log n)$

at the "cost" of min

$O(1) \rightarrow O(\log n)$

[But really not!!]

Amortization: most are cheap

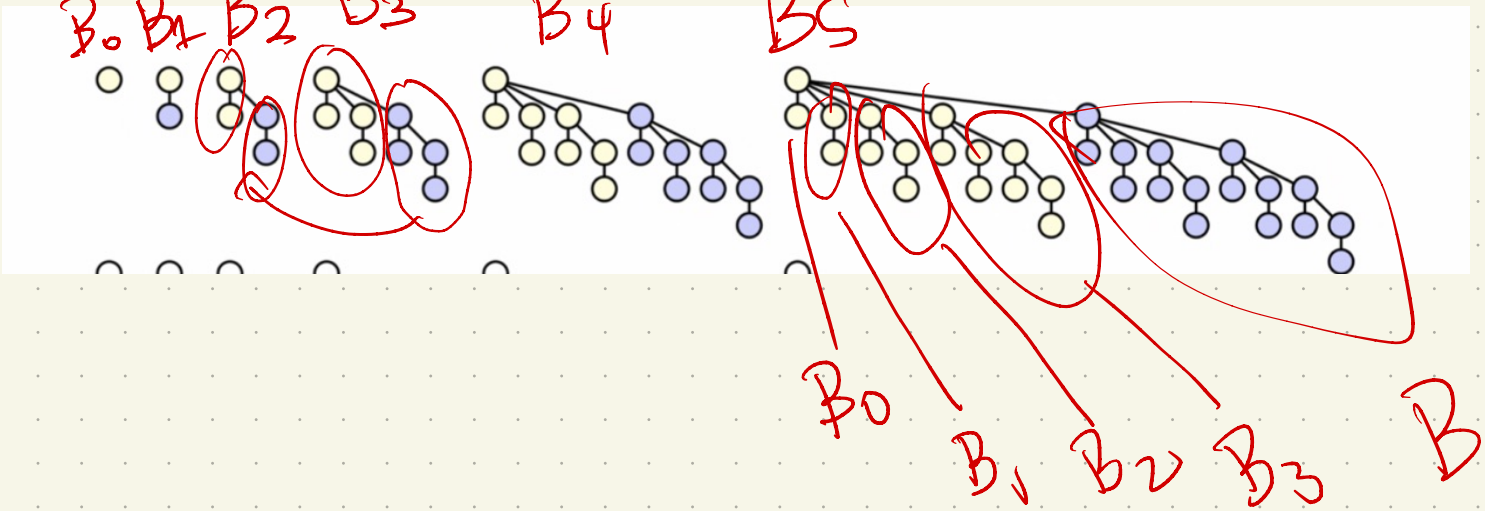
Some take longer

Dfn: A binomial tree, defined recursively:

Base case: B_0

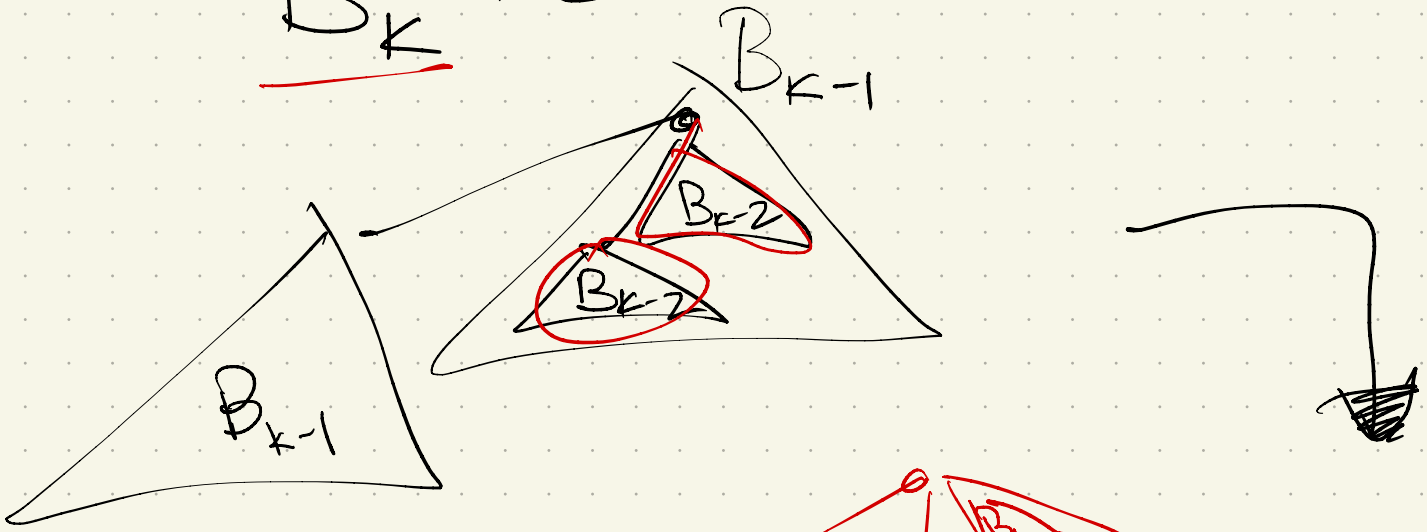
B_i: two copies of B_{i-1}, one root connected as (new) child of the other

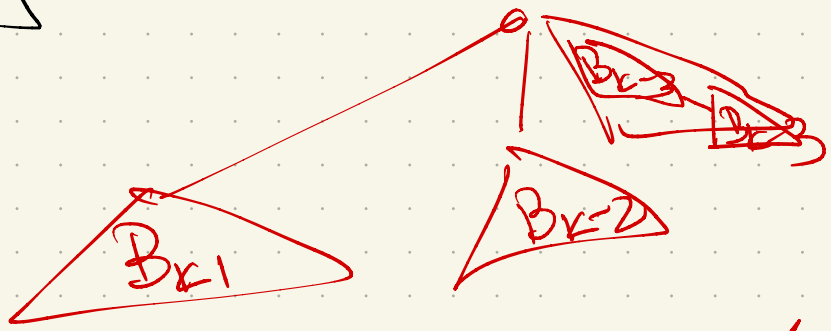
Picture:



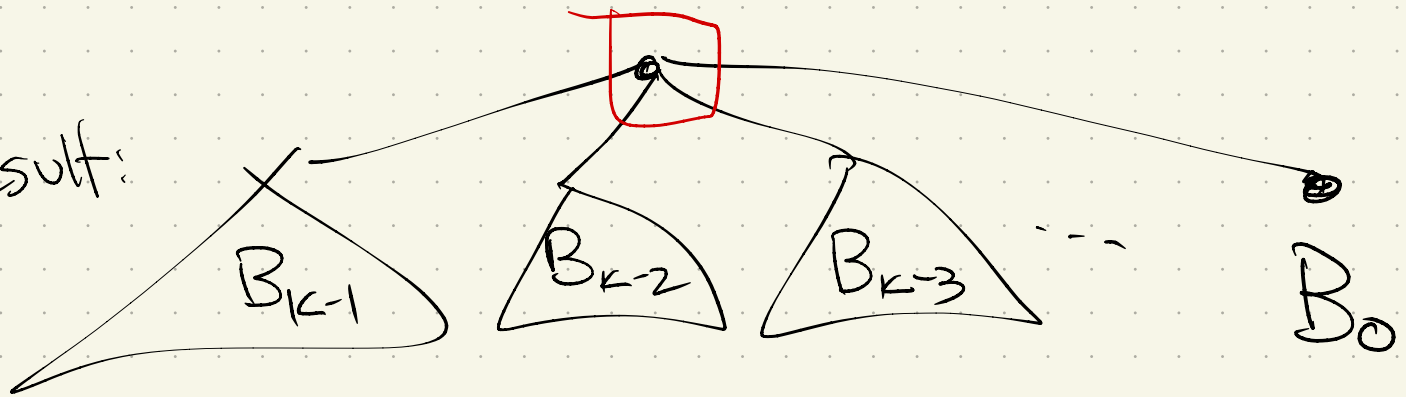
General pattern:

B_k : binomial tree of order k





End result:



HW0 question from previous semester!

4. A binomial tree of order k is defined recursively as follows:

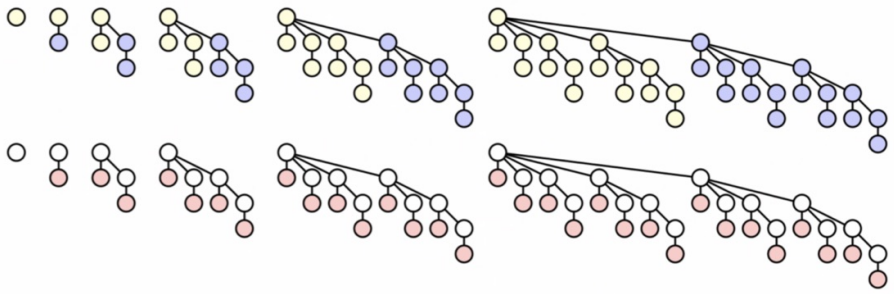
   - A binomial tree of order 0 is a single node.
   - For all $k > 0$, a binomial tree of order k consists of two binomial trees of order $k-1$, with the root of one tree connected as a new child of the root of the other. (See the figure below.)

   Prove the following claims:

   (a) For all non-negative integers k, a binomial tree of order k has exactly 2^k nodes.

   (b) For all positive integers k, attaching a leaf to every node in a binomial tree of order $k-1$ results in a binomial tree of order k.

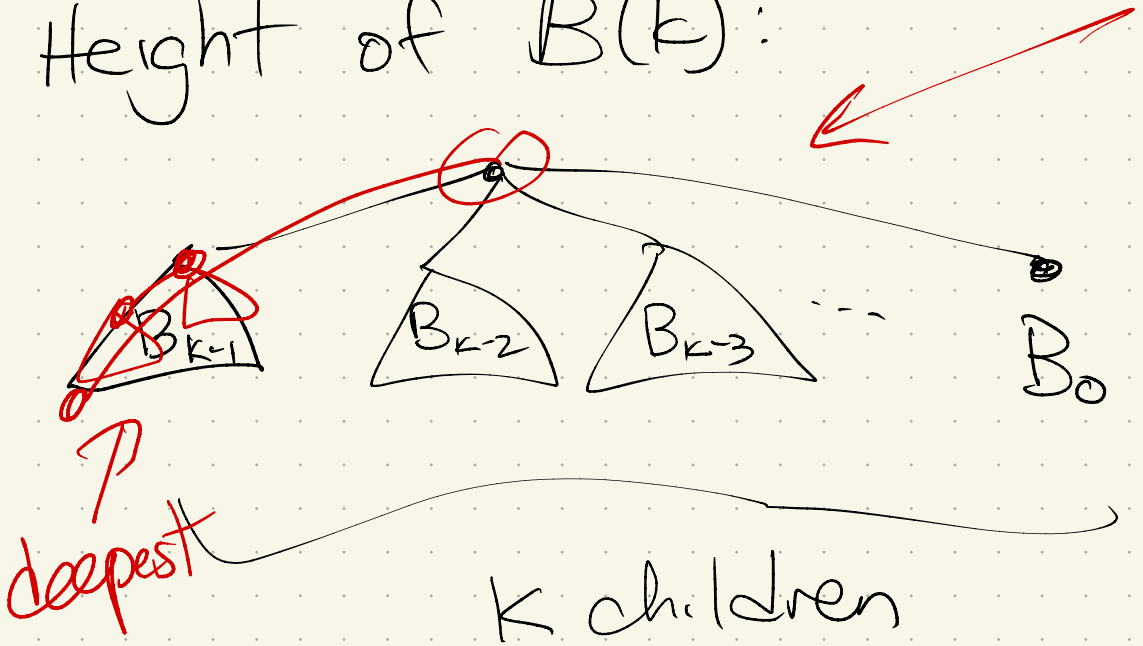
   (c) For all non-negative integers k and d, a binomial tree of order k has exactly $\binom{k}{d}$ nodes with depth d



Binomial trees of order 0 through 5.
Top row: the recursive definition. Bottom row: the property claimed in part (b).

So size = 2^k nodes in B_k

Height of $B(k)$:



which is biggest?

so $H(k) = 1 + H(k-1)$

$= 1 + 1 + H(k-2) \Rightarrow k$

in terms of # of nodes: B_k has 2^k nodes

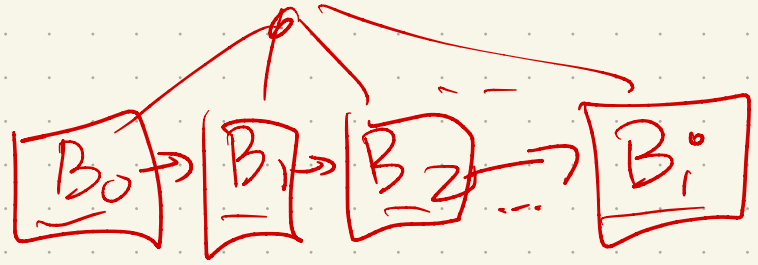
& height k

if $n = 2^k$, height $= \log_2 n$

Binomial Heap (min heap)

- Like regular heap, child > parent (for all nodes)

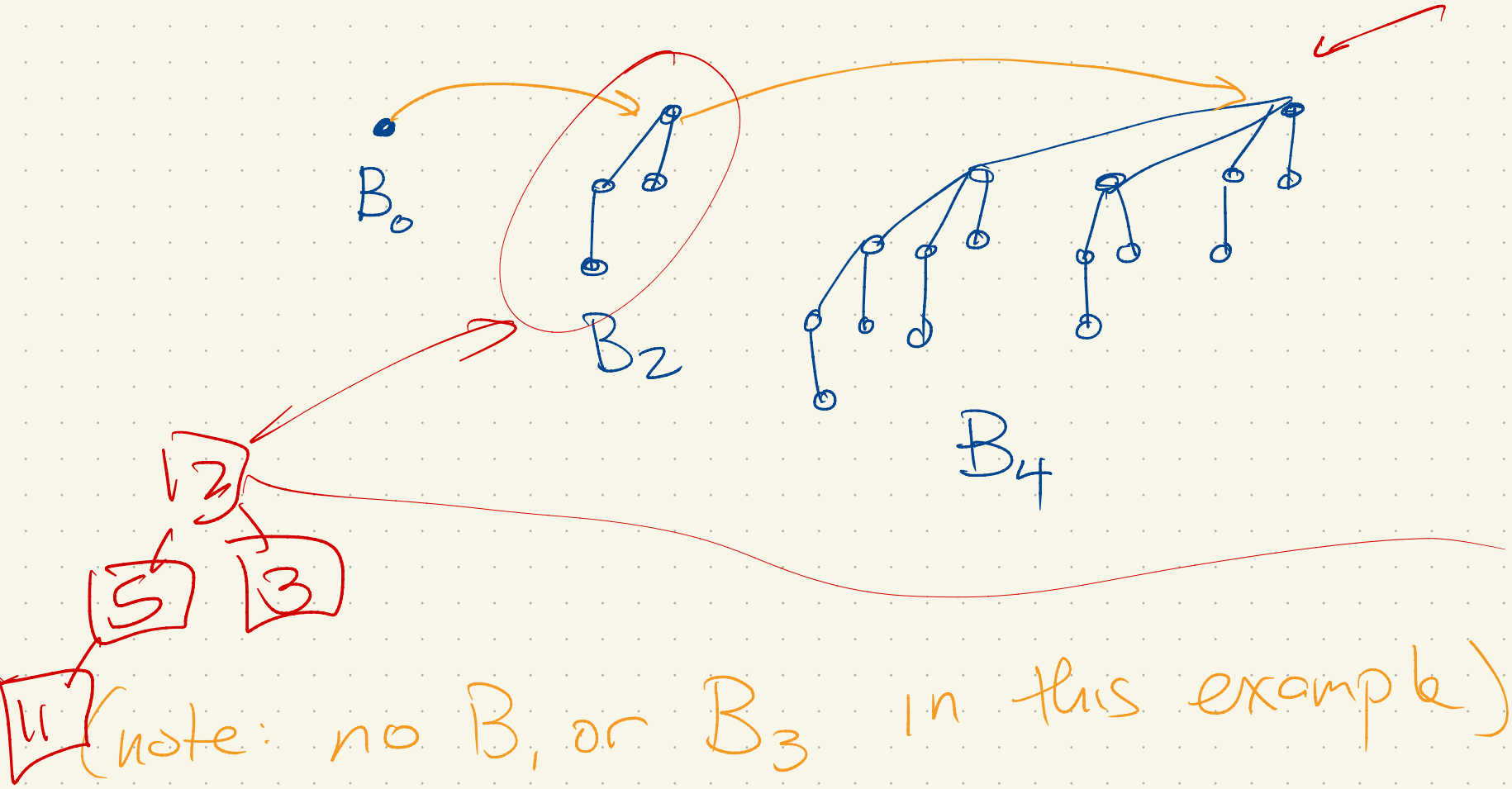
- But: this is a collection of binomial trees, with at most 1 of each size



so: one B_0
one B_1
one B_2
⋮
one B_i (some i)

Index via a linked list sorted by degree $0..i$

Ex List in orange trees in blue



(note: no B_1 or B_3 in this example)

length of list (in terms of # of nodes):

n nodes $\rightarrow$ # children $\leq \log_2 n$

2^k k

How to search:

Look at the roots & take min

Runtime: $O(\log_2 n)$ time

(1 per root)

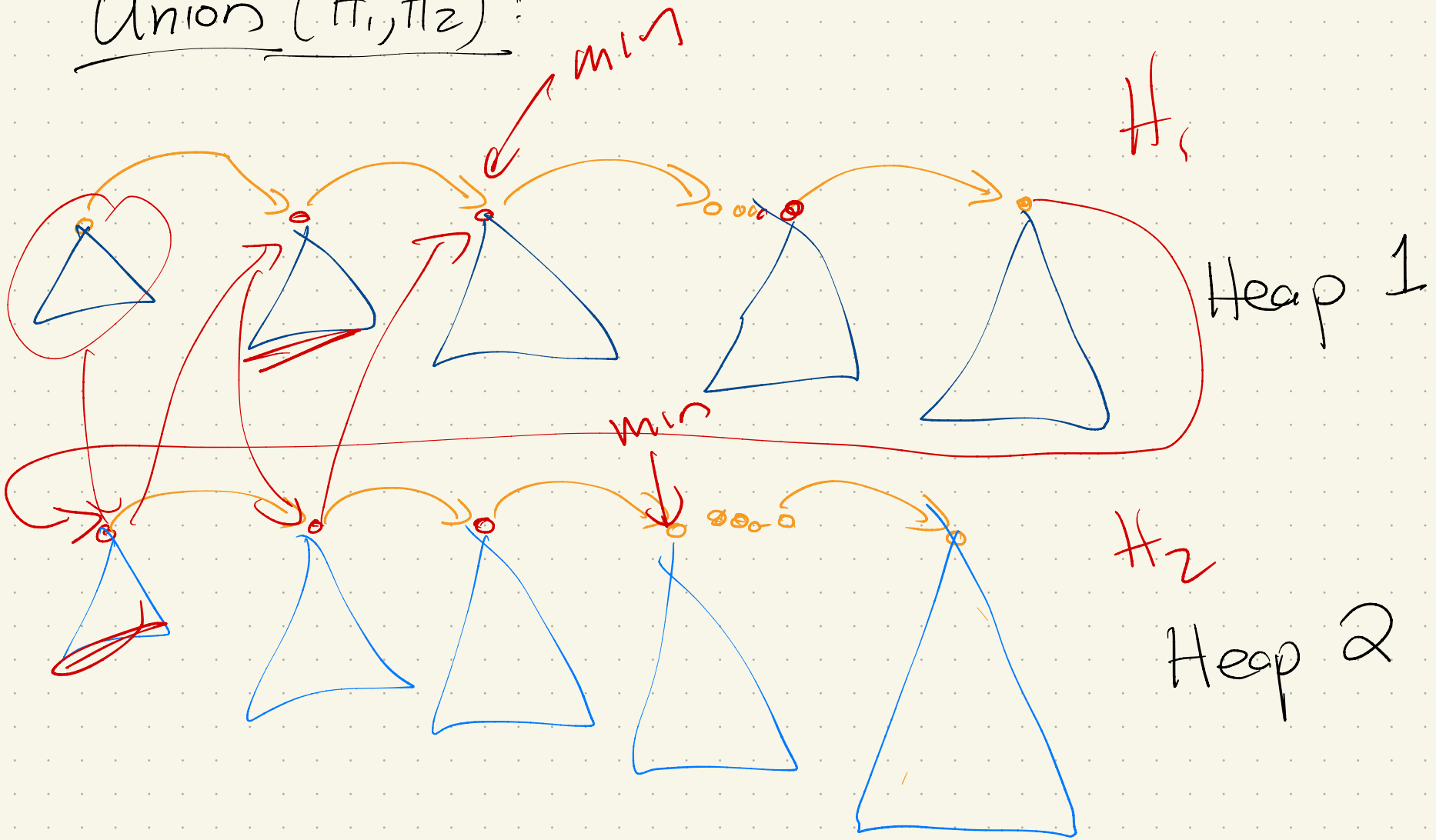
But: can keep global ptr to minimum

(& just need to update it as you go)

↳ Runtime: $O(1)$ to follow min ptr

Update?

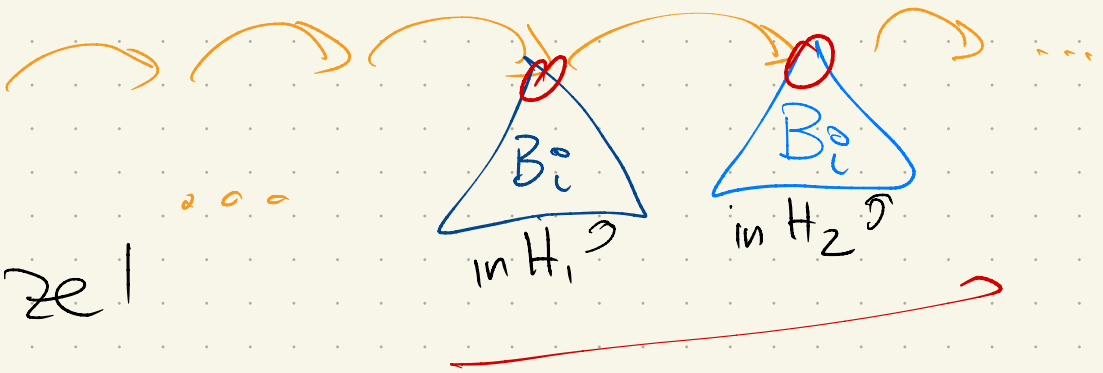
Union (H_1, H_2):



natural idea:

combine the lists of roots

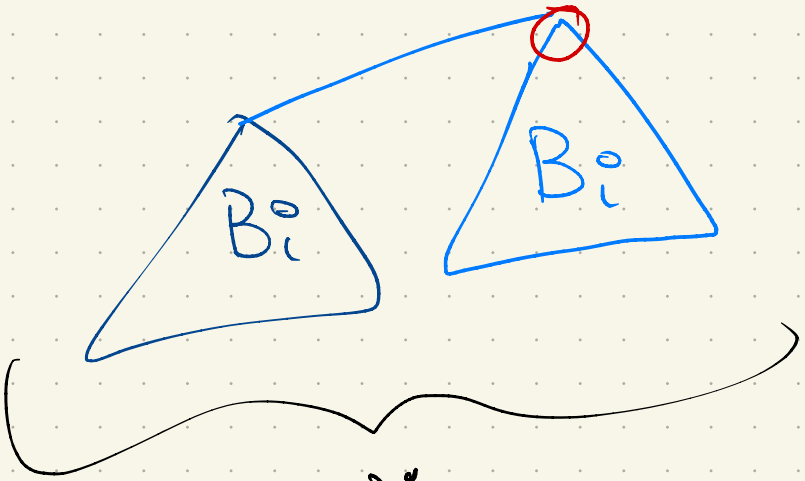
Problem: Merged list:



Could have 2 of same size!

Nice trick: combine them!

put smaller root on top



now:

B_{i+1}

More detail:

for i ~~=~~ — 0 to length of merged list:

if no nodes of degree i:

✓

if 1 node of degree i

✓

if 2 nodes of degree i

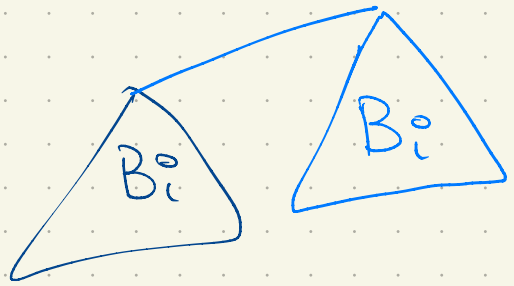
combine to new node of deg i+1

if 3 nodes of degree i

combine 2 of them into deg i+1

Runtime of merge:

each internal merge: $O(1)$ time



And min add ptr

Overall: $O(\log_2 n)$

Insert (x, H)

Create a new binomial heap (size 1)

$\Rightarrow$ • (size 1 list)

B_0

Merge with H: $O(\log_2 n)$

(keep pointer to global min)

Runtime: Worst case: $O(\log_2 n)$

But: amortized insert is $O(1)$ time!

accounting method

Why insert is faster:

Suppose we do n inserts, & consider a merge inside our loop:

If no B_0: O(1) time

If $B_0 \ldots B_i$ exist, & B_{i+1} does not:

O(i), but after there is no $B_0 - B_i$

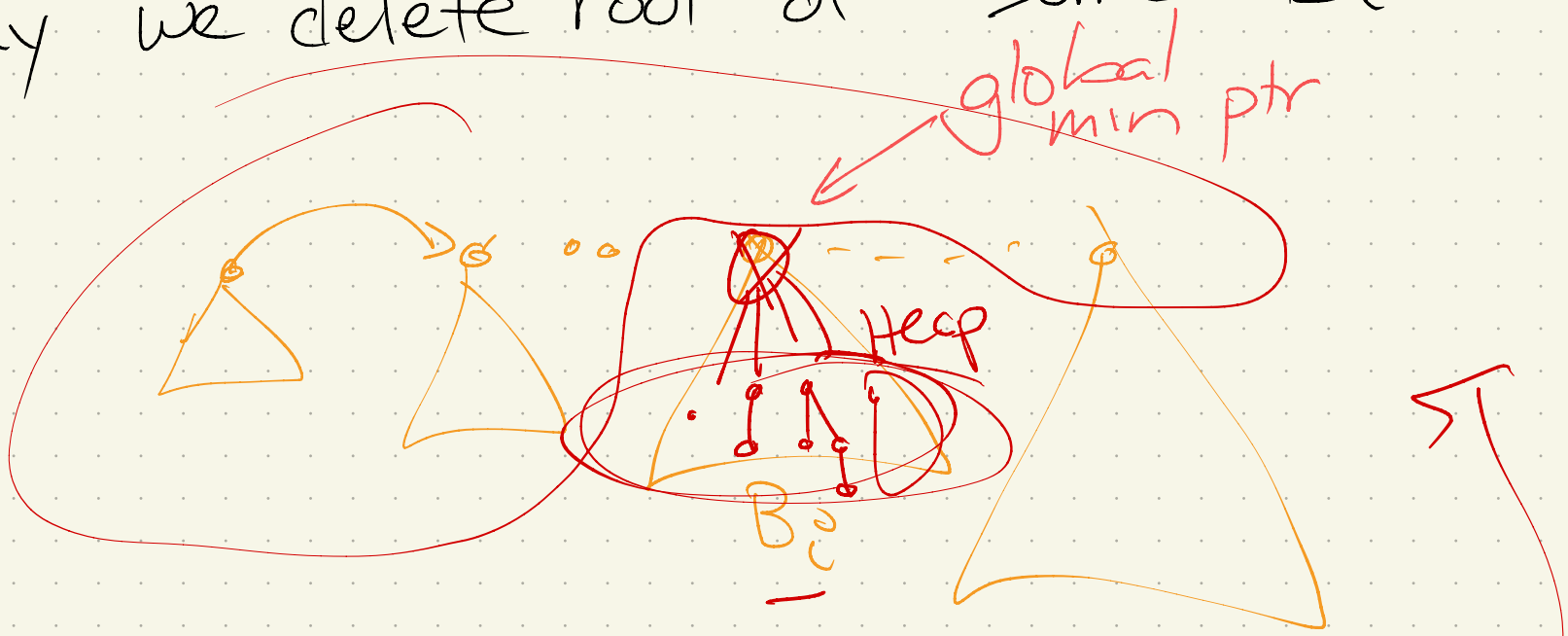
$\ldots \to ! \to : !$

Accounting method:

n inserts pay for expensive operation

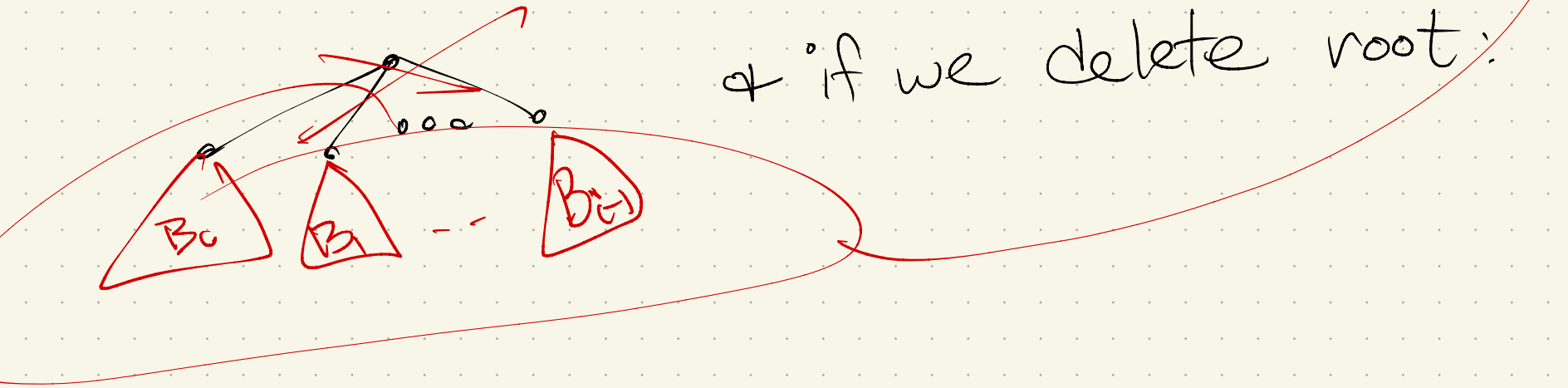
~~Extract~~ Delete Min():

Say we delete root of some B_i:

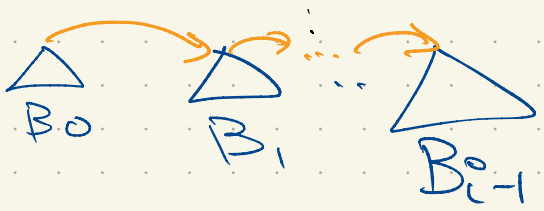


Recall: B_i is what?

& if we delete root:



So: flip children of root in B_i:



: H from children

Make a heap from these & merge with rest of the heap

Runtime: $O(\log_2 n)$

Result:

	Heap	Binomial heap
getMin	$O(1)$	~~$O(\log n)$~~ → $O(1)$ (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	$O(n)$	$O(\log_2 n)$

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

Only downsides: constants & overhead

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 any vertices that now have a better path

↳ decreaseKey

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

A first attempt:

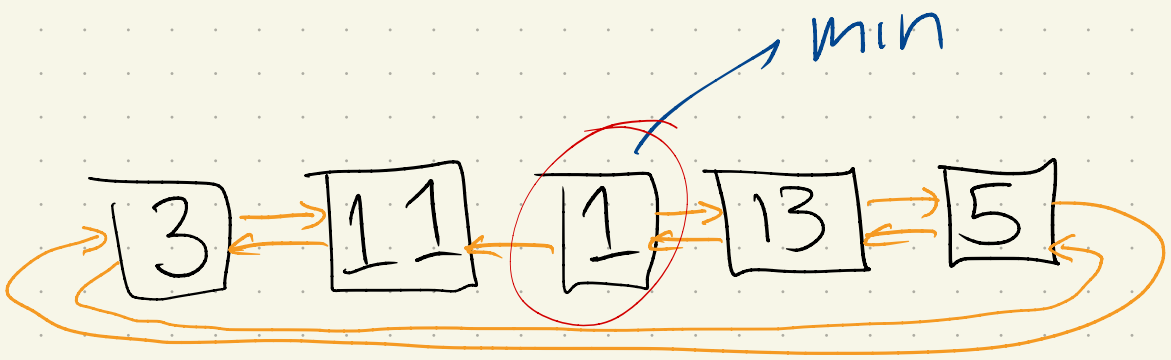
What if we just used:

- m n
- insert
- merge

Missing: decreaseKey, (deleteMin + delete)

Can we do something simple?

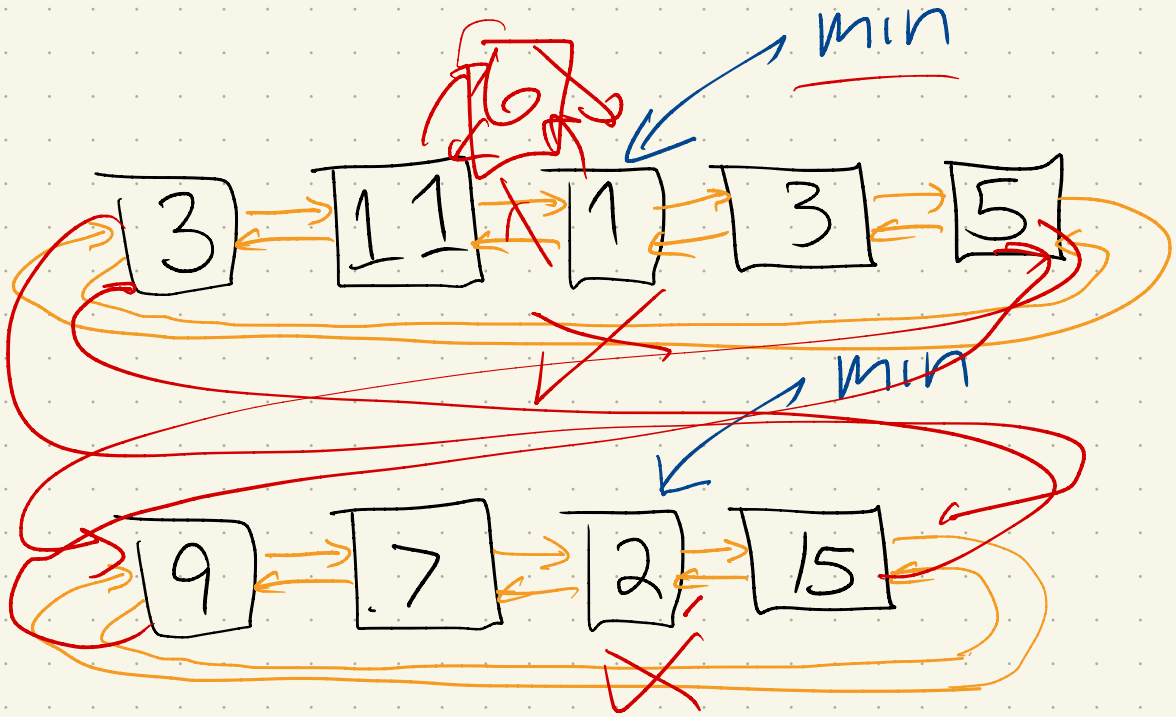
Linked List:



Min:

Insert: 6

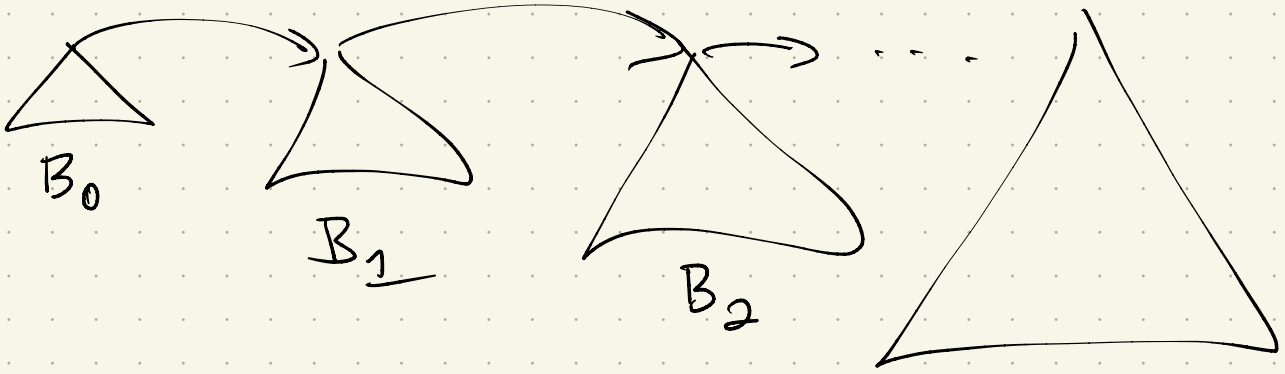
Union/Merge:



But: decrease Key?

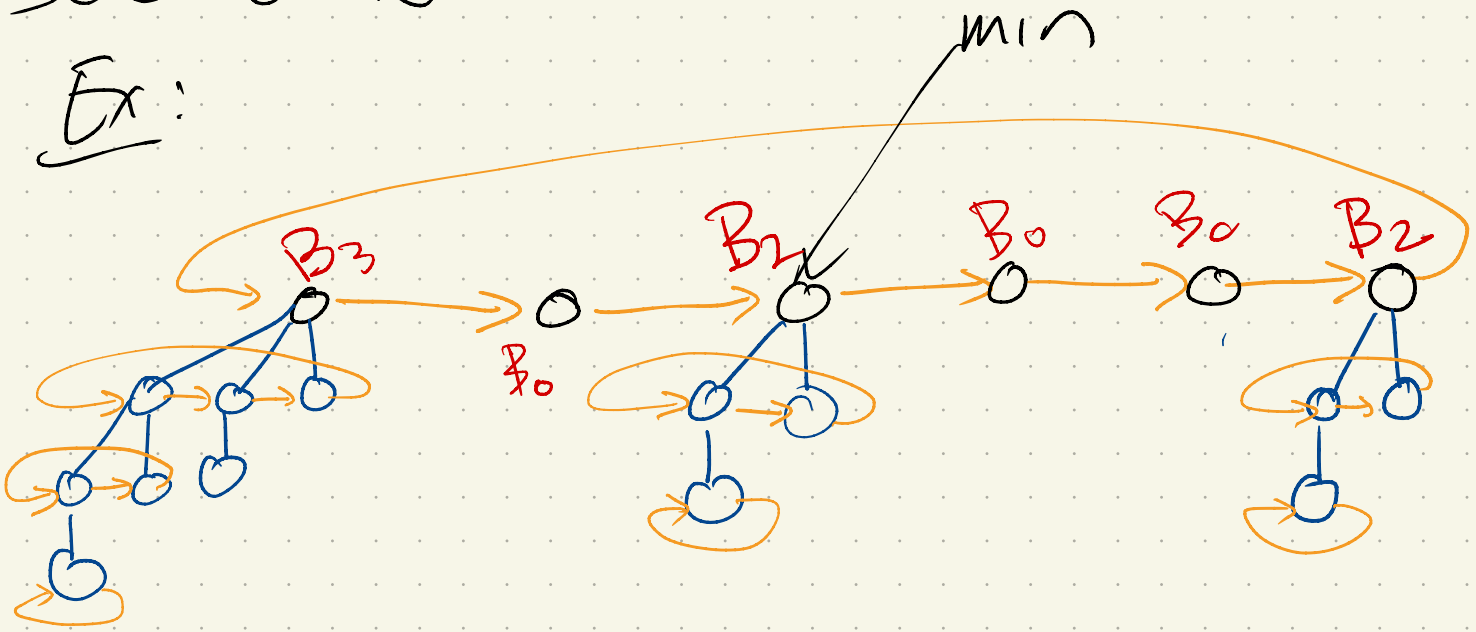
Fibonacci Heaps: "Lazy binomial heaps"

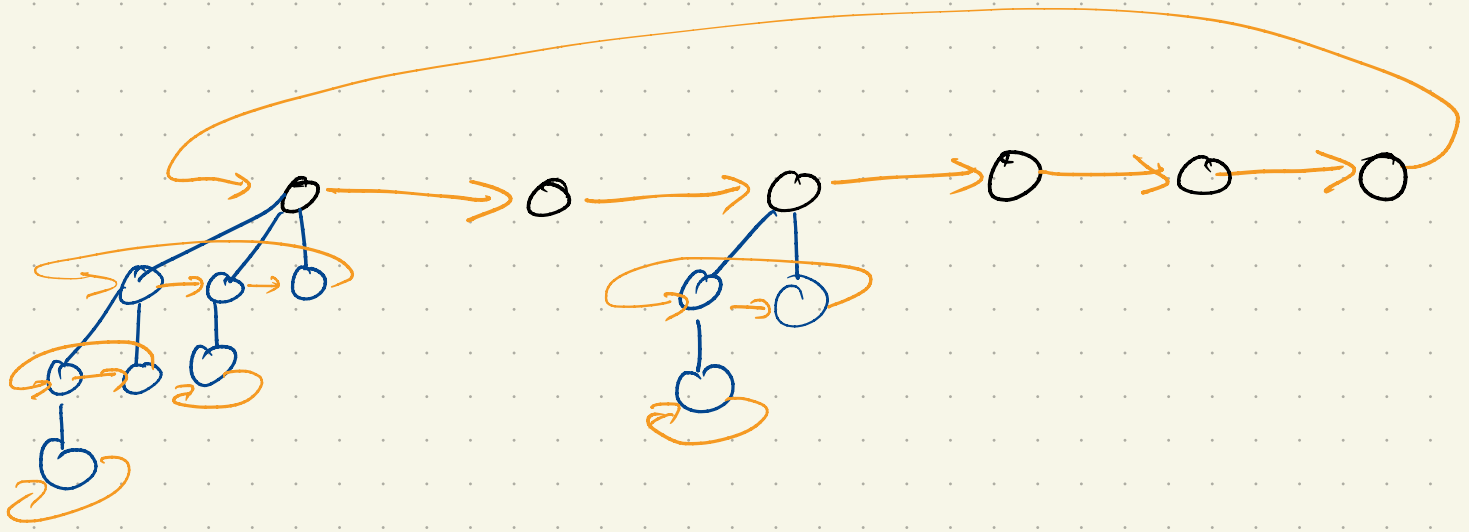
Start with a binomial heap:



Fibonacci heap:
relax the sructure, ←
all ≥1 of each in
no set order

Ex:





Functions

- insert: easy - add new B_0 $O(1)$
- getMin: follow ptr
- merge/union: $O(1)$

deleteMin & decreaseKey: a bit harder!

DeleteMin(): pay for laziness!

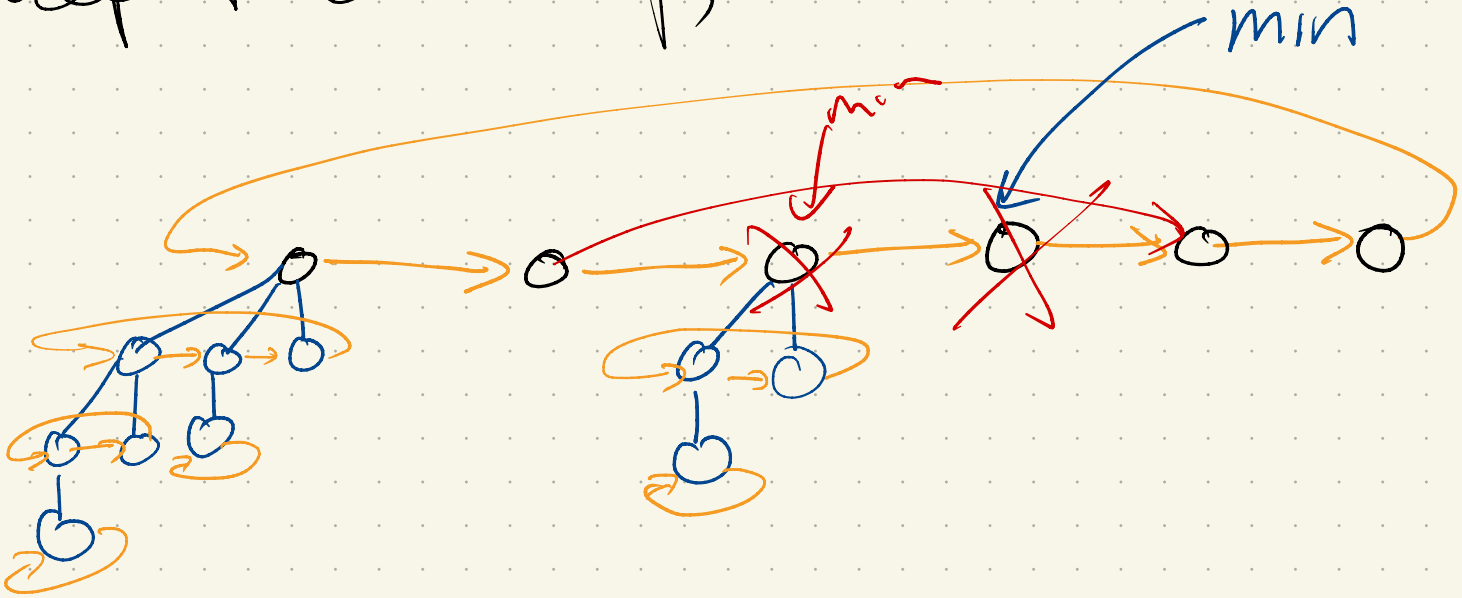
- Delete the min
- Set child's parent ptr to NULL

⇒ Now 2 Fib heaps

- Link the two lists.

(Now lots of bin. trees, unordered)

- Sweep + clean up, so ≤1 of each size

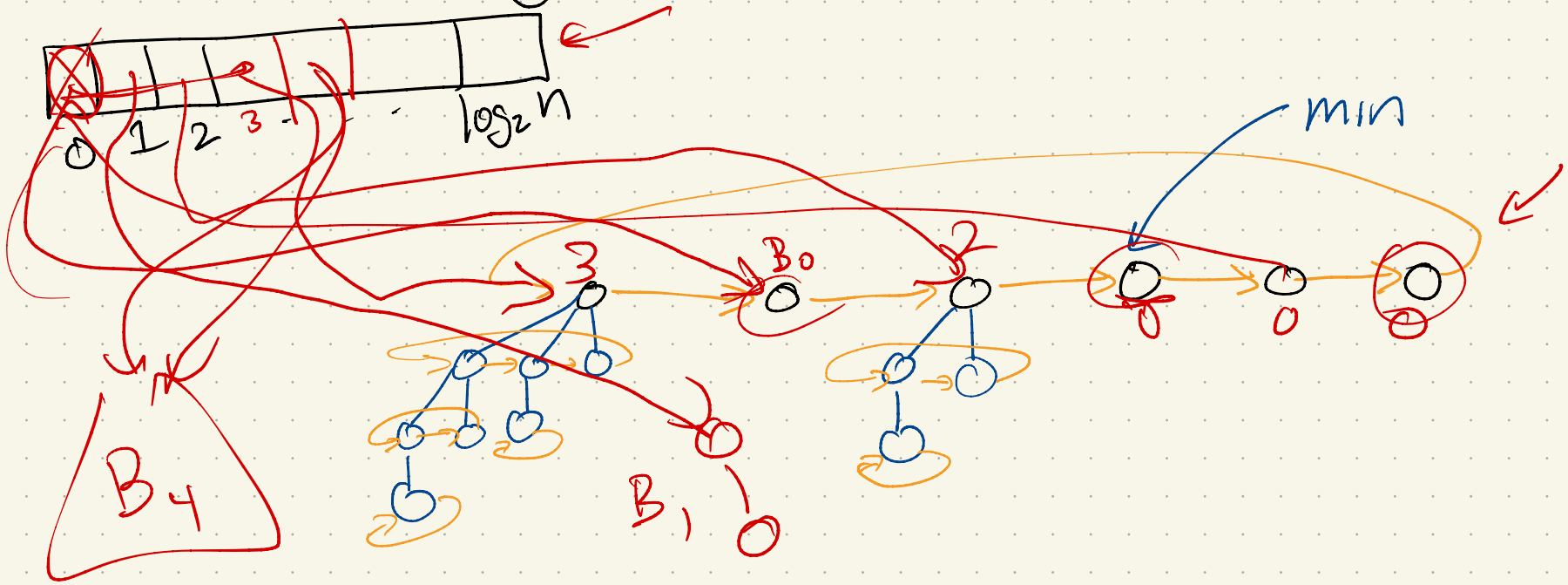


Sweep + clean up, so ≤ 1 of each size

Like union/merge, but worse!

(No ≤ 3, not in order)

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



Runtime: $O(\text{size of list before deleting}) + O(\log_2 n)$

inserts

Why?

- traverse list
- each gets merged ≤ 1 time

(then it is a child)

Amortized analysis:

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

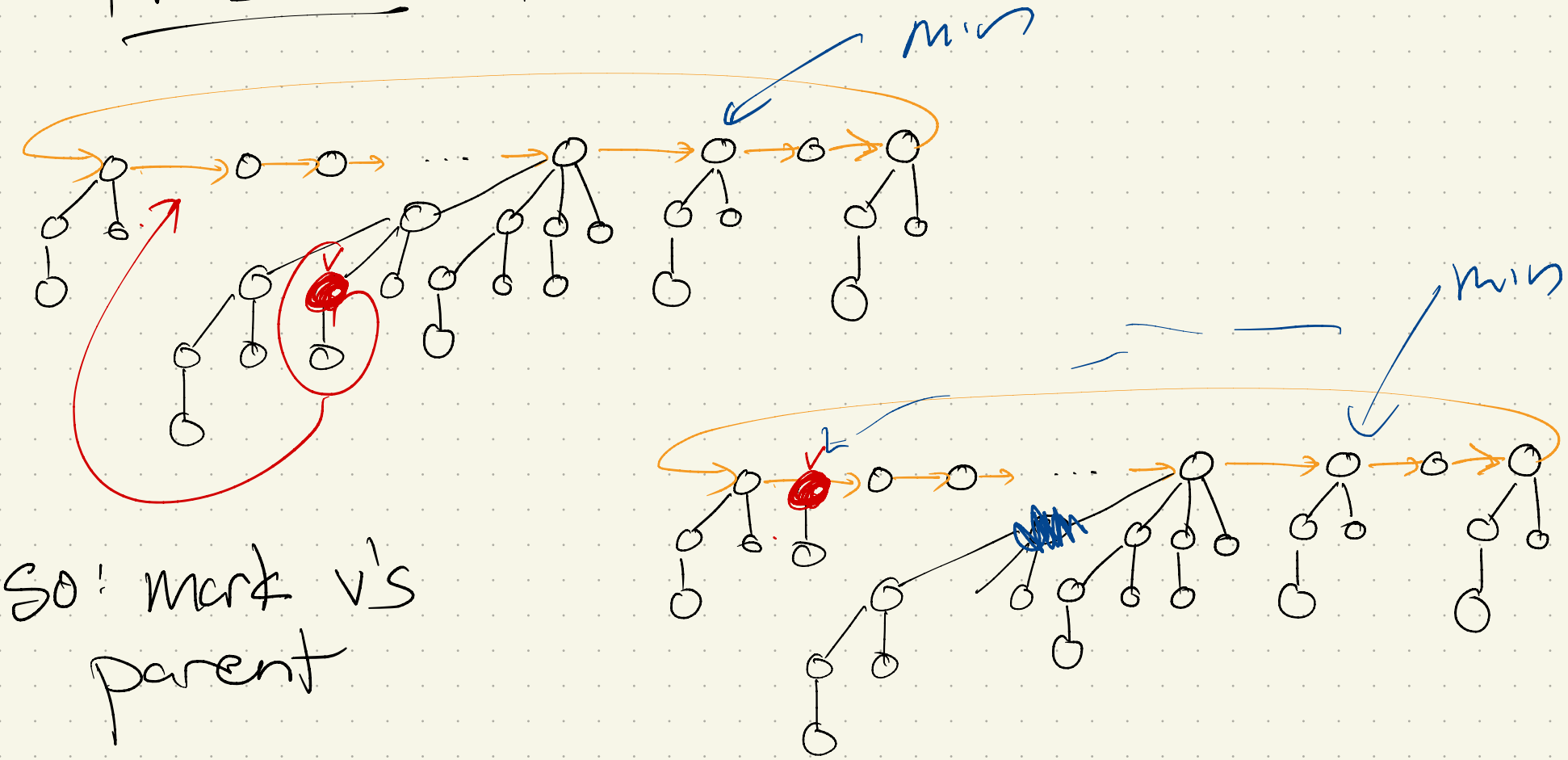
gets one longer

so → have insert pay +1

$\Rightarrow O(\log_2 n)$ amortized

DecreaseKey If v's value decreases, move it to root list (update min if needed)

Problem: Not a binomial tree!



So: mark v's parent

If already marked:

Move parent's subtrees up to root list, & unmark

Move up tree (& move as long as marked)

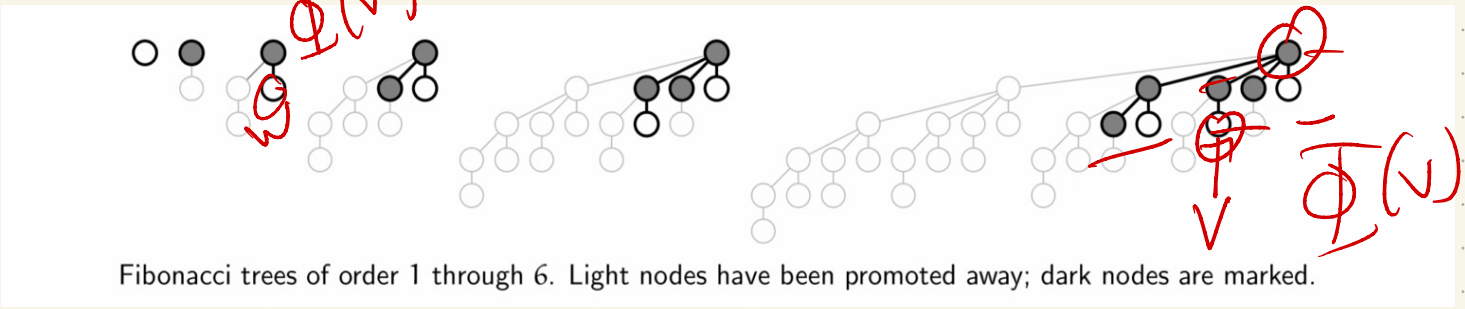
Runtime: $O(1 + \#\text{marked ancestors of } v)$

$= O(\text{depth}(v))$

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.

Most we can remove without triggering cascade of promotions.

(Note: Fibonacci #s!)

Potential function

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

When promote one, $+1$: move v up, mark ancestor

When promote k, $-k$

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

after op, overall potential goes down by $-(\Phi(v)+1)$ b/c unmark $\Phi(v)$ vertices

Result: cancel

Runtime for Fib. heaps:

Min
Insert
Merge/Union
} O(1)

DecreaseKey: O(n) worst
O(1) amortized

deleteMin: O(n) worst
O(log n) amortized

⇒ Dijkstra: works well