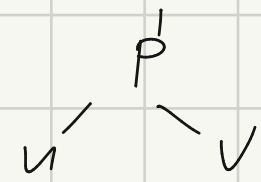
Binary heap

1. heap property



p<u & p<v.

2. complete binary tree

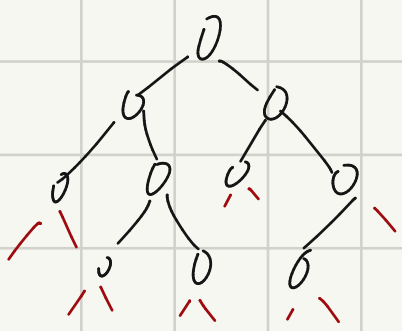
findmin	O(1)	
deletemin	O(logn)	
insert	O(logn)	
given pointers.	decreasekey	O(logn)
	delete	O(logn)

makeheap O(n)

merge: O(n) ⟶ O(logn) (Goal)

Leftist heap ⇒ complete binary tree.

1. heap property.
2. binary tree with leftist property.



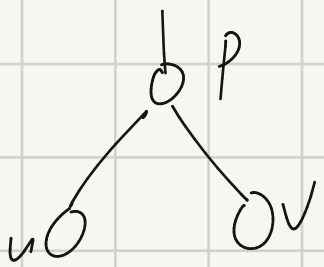
from u to null, npl(u) = shortest path length

from each u

the right decending path from u to null

is the shortest from u to null

(one of)



$npl(p) = \min\{npl(u), npl(v)\} + 1$

if leftist $npl(u) \geq npl(v)$

Lemma

For a leftist heap with n nodes, its right path has at most $\log_2(n+1)$ nodes

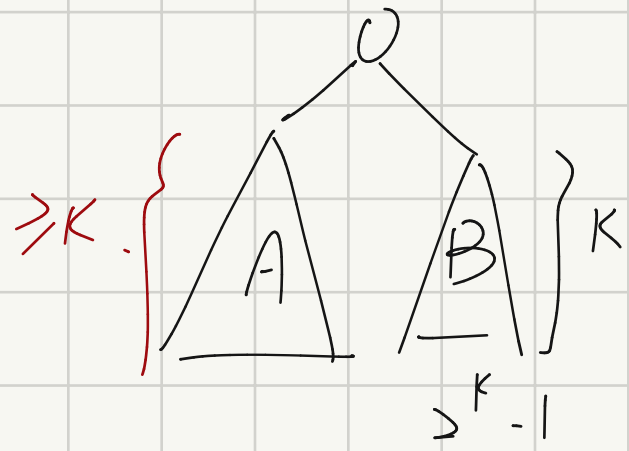
$\Uparrow\Downarrow$

nodes in right path: r # nodes in leftist heap $\geq 2^r - 1$

base: $r=1$ $2^1 - 1 = 1$

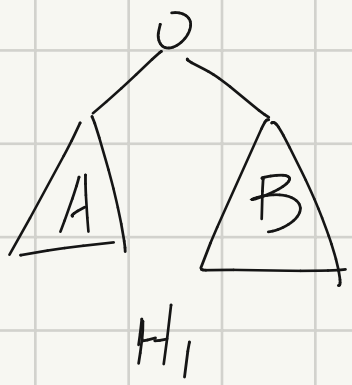
inductive hypothesis $r=k$ $\geq 2^k - 1$.

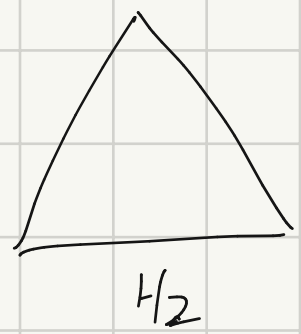
" " step $r = k+1$



$\geq 2(2^k - 1) + 1 = 2^{k+1} - 1$

merge:





$H_1.root < H_2.root$

$\Rightarrow$ swap children

if $npl(A.root) < npl(B'.root)$

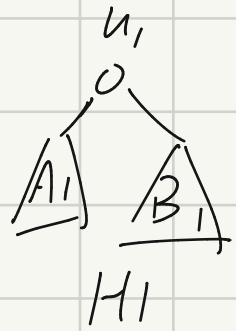
$B' = merge(B, H_2)$

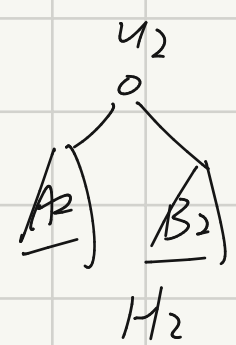
(recursive)

merge (H_1, H_2)

1. if $H_1 == null$

   return H_2.





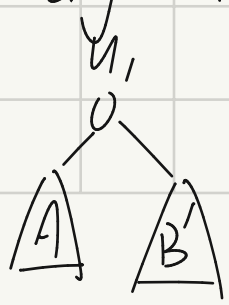
2. if $H_2 == null$

   return H_1

3. if $H_1.Key < H_2.Key$

4. $B' = merge(B_1, H_2)$

5. $H =$



if npl (A) ≥ npl (B')

6 update npl (u_1)
7 return H
8 else // u_1.key > u_2.key.
similar

levels of recursion ≤ $r_1 + r_2$ → # nodes on the right path of H_i

time per level: $O(1)$

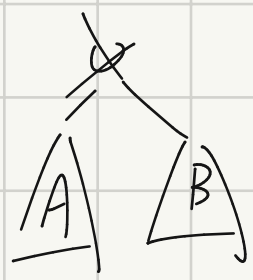
total time: $O(r_1+r_2)\cdot O(1) = O(\log n_1 + \log n_2)$

$= O(\log n)$

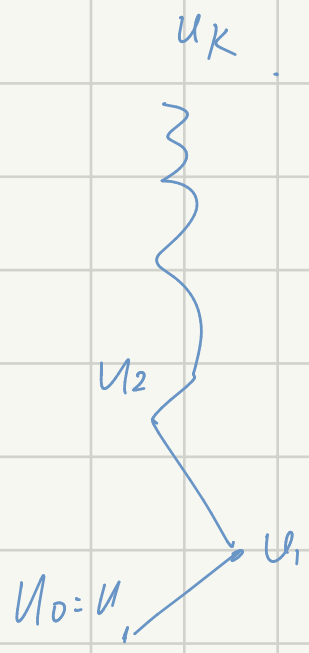
$n = n_1 + n_2$

then Insertion: Merge (H_1, and a node).

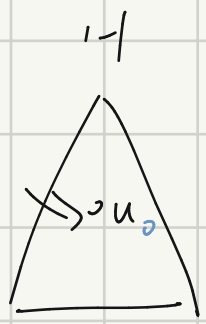
Deletemin



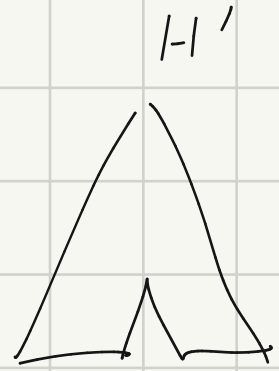
merge (A, B)



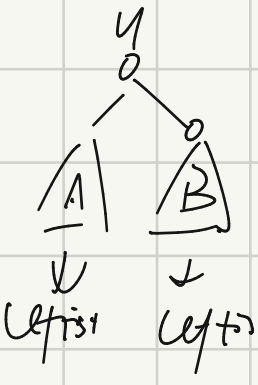
delete:



=

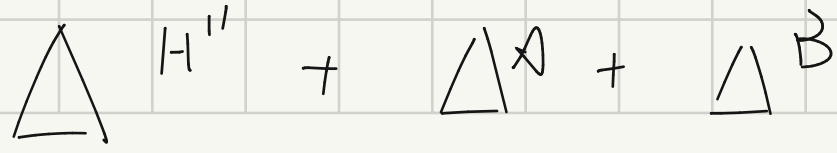


+

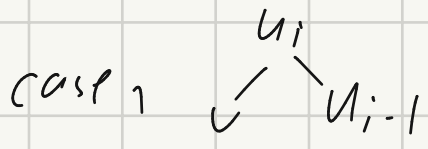


↓ fix the leftist property

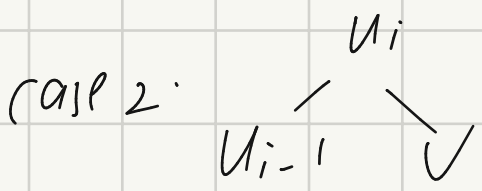
=



for $i=1, \dots k$ $\quad u_i$: ancestors of u_0 (nodes influenced)



update $\quad npl(u_i) = \min\{npl(u_1), npl(u_{i-1})\} + 1$



case 2.1 $\quad npl(u_{i-1}) \geq npl(v)$

break.

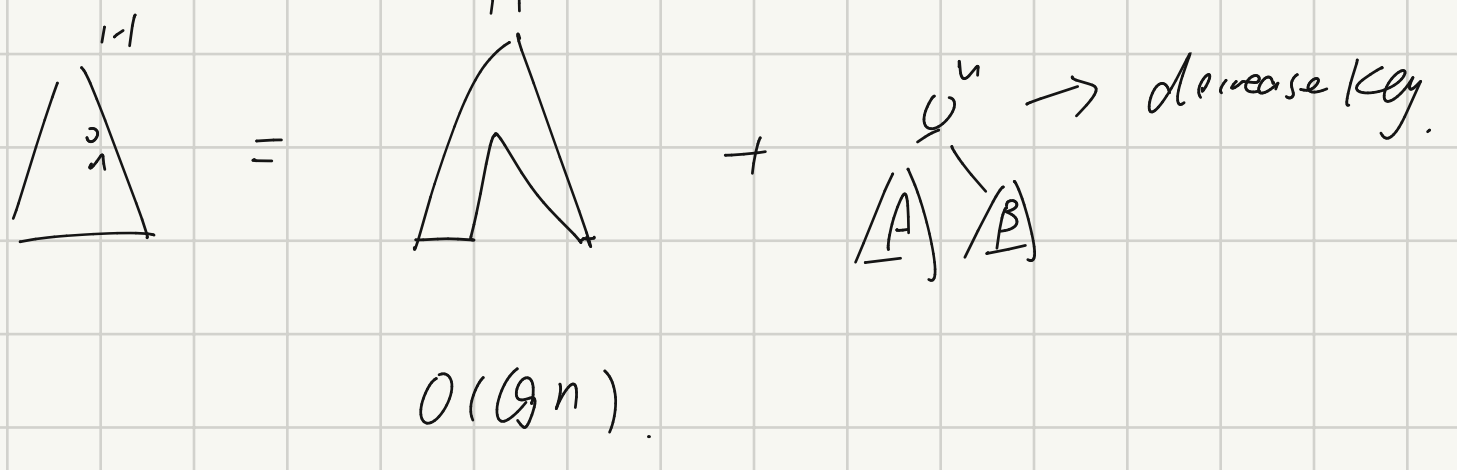
case 2.2 $\quad npl(u_{i-1}) \leq npl(v)$

swapchildren (u_i) ⇒ swap children of u_i

update $npl(u_i)$

$O(\lg n)$

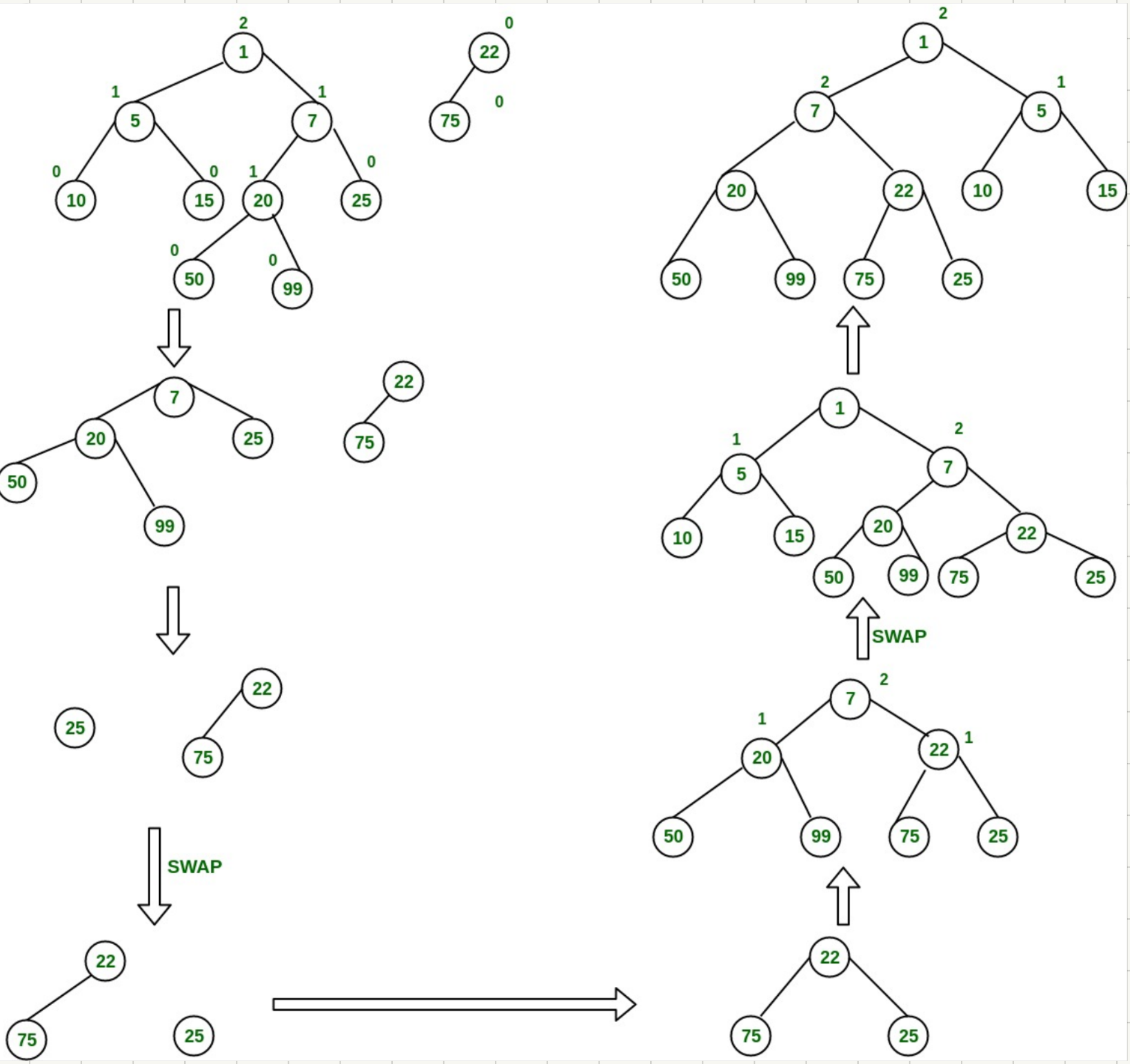
decreaseKey:



$u \to$ decrease key.

$O(\lg n)$.

make-heap: $O(n)$: the same as ordinary heap


2
1
0
22
1
1
5
7
0
75
0
0
0
1
0
10
15
20
25
0
0
50
99
7
22
20
25
75
50
99
22
75
25
SWAP
SWAP
1
2
1
7
5
1
2
20
22
10
15
50
99
75
25
10
15
50
99
75
25
1
2
7
1
1
20
22
50
99
75
25
22
75
25

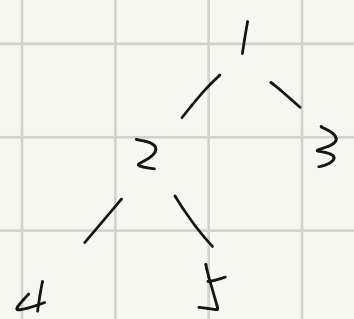
skewed heap : make : nlogn

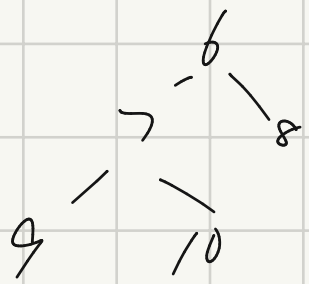
self - adjusting version of leftist heap.

① swap children after merge without judging

② if null, swap child for all nodes on the right path except for the lowest one.

H delete ②, still O(logn).





insertion
deletemin
> via merge

delete
decreasekey
> not support

→ insert and deletemin O(logn)

potential function:

Given a node u ∈ H

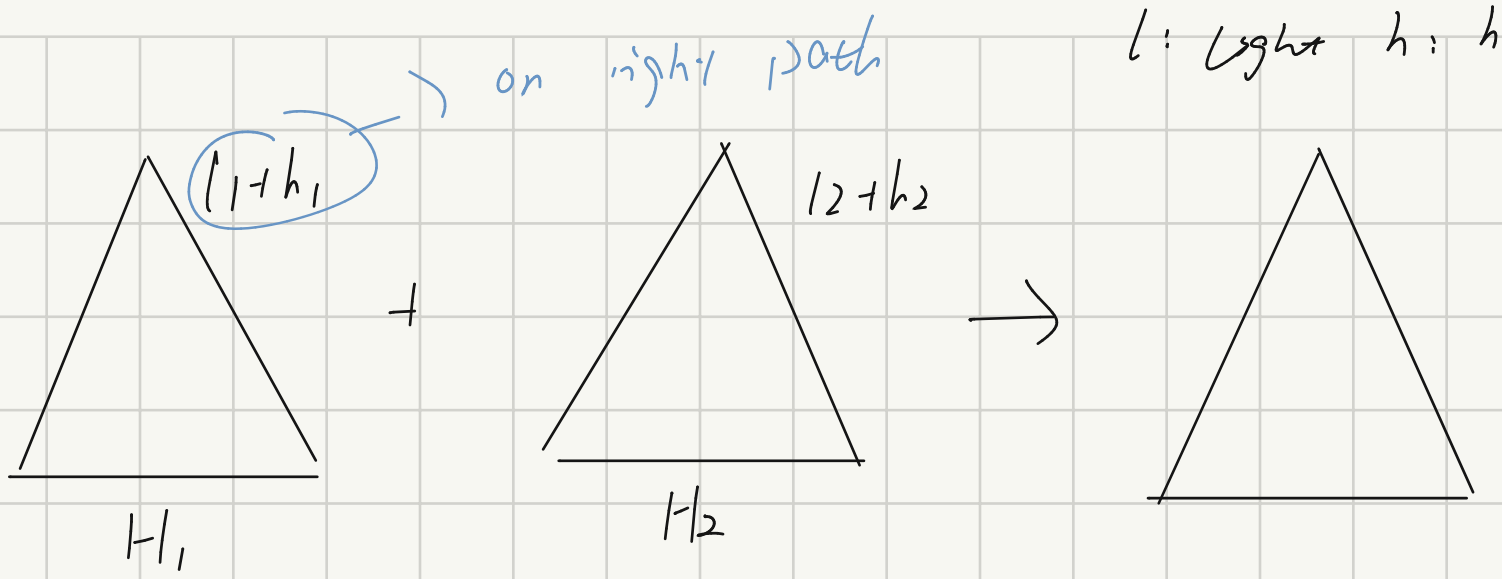
u is heavy if size (right subtree of u) ≥ size (left subtree of u)

light otherwise.

$\overline{\Phi}(H)$ = # heavy nodes in H

$\overline{\Phi}$ (empty) = 0 H-1 $\overline{\Phi}(H) \geq 0$

merge



actual cost = $O((l_1 + l_2 + h_1 + h_2)$

$\overline{\Phi}(H_1) + \overline{\Phi}(H_2) = h_1 + h_2 + h$ (heavy nodes not on right path

$\overline{\Phi}(H) \leq h + l_1 + l_2$ (because all heavy nodes turn to light, but not all

↳ remain unchanged, because they're not swapped light turn to heavy)

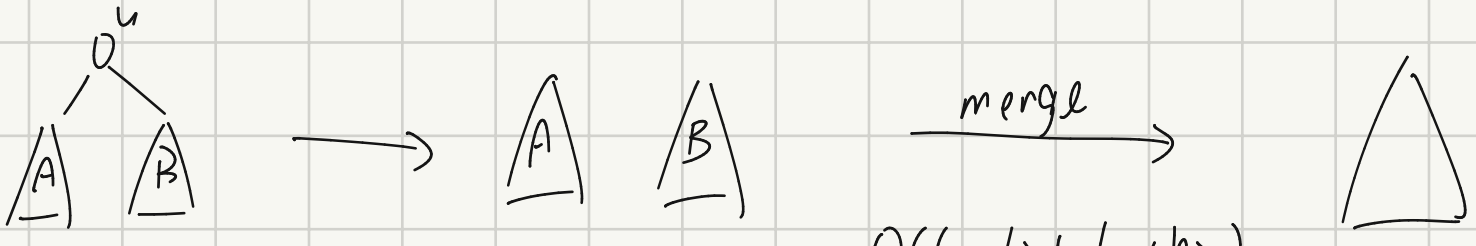
$\Delta\overline{\Phi} \leq l_1 + l_2 - h_1 - h_2$

ammorlized cost = actual cost + $\Delta\overline{\Phi}$

$= O(l_1 + l_2)$

$= O(\lg n_1 + \lg n_2) = O(\lg n)$

Delmin



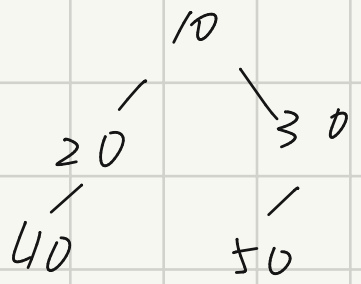
Actual: $O(1)$ $O((l_1 + l_2 + h_1 + h_2)$

$\Delta\overline{\Phi}$: ≤ 0 $O(l_1 + l_2 - h_1 - h_2)$

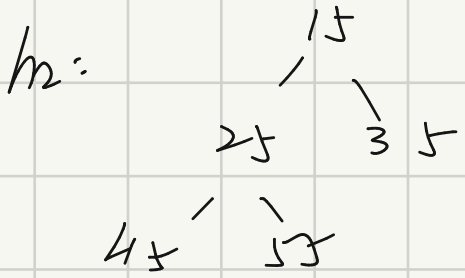
∴ amortized = $O(l_1 + l_2) = O(\lg n)$

eg. Merge (h_1, h_2)

h_1:

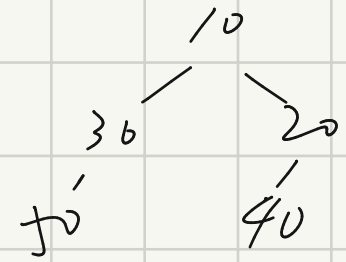


h_2:



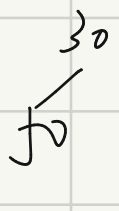
①: 1. 10 < 15 => with swap h_1 and h_2.

2. swap h_1:

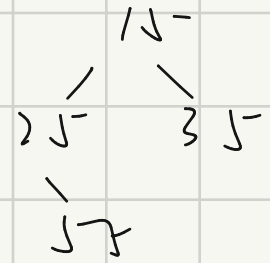


3. Merge (h_1 -> left child, h_2)

②. h_1':

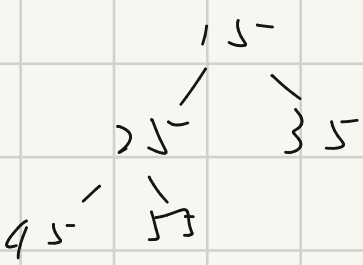


h_2':

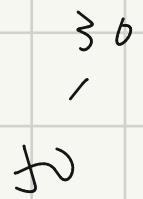


1. 30 > 15 => swap h_1' and h_2'

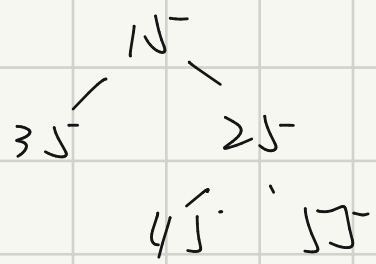
=> h_1'



h_2'



2. swap h_1'



3. Merge (h_1' -> left child, h_2')

③ : h_1'' : 35 h_2'' 30 / \ 40

1. 35 > 30 ∴ swap h_1'' and h_2''

h_1'' 30 / \ 40 h_2'' 35

2. swap h_1''

30 / \ 40

3. Merge (h_1'' -> left child, h_2'')

④ h_1'' NULL h_2'' 35

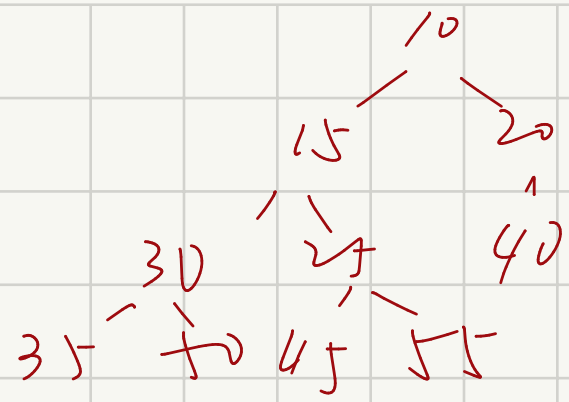
∴ => 35

then return back.

=> ③' 30 / \ 35 40

=> ②' 15 / \ 30 25 / \ / \ 35 40 40 55

=> ①'


10
15
20
30
25
40
35
45
55