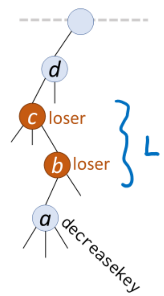
REVIEW

Amortized analysis

Φ = num.roots + 2 × num.losers


```
def decreasekey(v, k′):
    let n be the node where this value is stored
    n.key = k′
    if n violates the heap condition:
        repeat:
            p = n.parent
            remove n from p.children
            insert n into the list of roots, updating minroot if necessary
            n.loser = False
            n = p
        until p.loser == False
        if p is not a root:
            p.loser = True
```



Case I: no heap violation. $c = O(1)$ $\Delta\Phi = 0$ $c + \Delta\Phi = O(1)$.

Case II: heap violation.

1. move a to root list. $c = O(1)$ $\Delta\Phi = 1$ or $\Delta\Phi = -1$ if a was loser. $c + \Delta\Phi = O(1)$
2. move up L losers $c = O(L)$ $\Delta\Phi = +L - 2L = -L$. $c + \Delta\Phi = O(1)$
3. mark d as loser. $c = O(1)$ $\Delta\Phi = 2$ or $\Delta\Phi = 0$ if d was a root $c + \Delta\Phi = O(1)$

tot. is $O(1)$

in each cases, $O(1)$ am. cost.

Q1. What on earth does this maths mean? –

$$O(L) - L = O(1)$$

Q2. How do we come up with potential functions?

Only use $O(n) - n = O(1)$ when working with potential functions.

EXAMPLE: DYNAMIC ARRAY

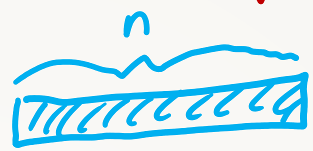
A Python list is implemented as a dynamically-sized arrays. It starts with capacity 1, and doubles its capacity whenever it becomes full.

Suppose the cost of writing an item is $O(1)$, and the cost of doubling capacity from m to $2m$ (and copying across the existing items) is $O(m)$.

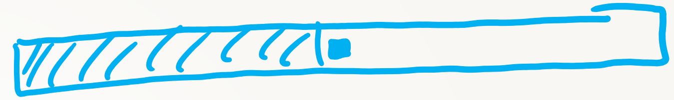
Show that the amortized cost of append is $O(1)$.

Φ = # items added since last doubling. × €

When working with Φ, there's always an arbitrary exch. rate.



$\Phi = \frac{n}{2}$ €



$\Phi = 1$ €

"cost $= O(n) + O(1)$"

really means: cost $= c_{double} + c_{app}$.

$$\left.\begin{aligned} c_{double} &\le k_1 n \\ c_{app} &\le k_2 \end{aligned}\right\} \text{for } n \text{ suff. large.}$$

$\Delta \Phi = \Phi_{after} - \Phi_{before}$

$= (1 - \frac{n}{2})$ €

$$\begin{aligned} \text{am. cost} &= \text{cost} + \Delta\Phi \\ &\le k_1 n + k_2 + (1 - \tfrac{n}{2}) € \\ &= n\left(k_1 - \tfrac{€}{2}\right) + k_2 + €. \end{aligned}$$

let's set € $= 2k_1$. Then

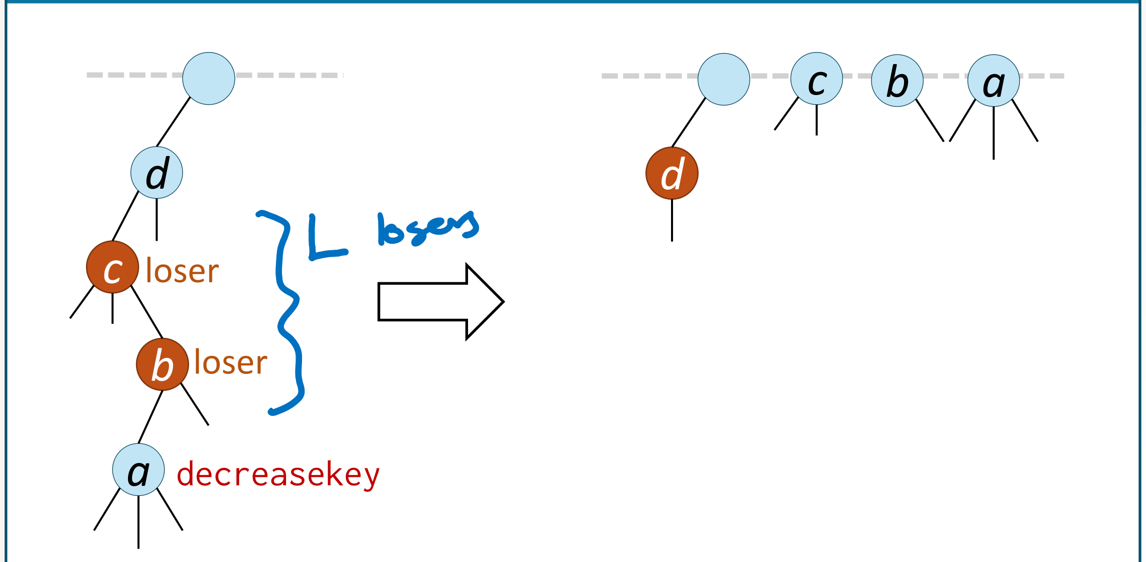
am. cost $\le k_2 + 2k_1 = O(1)$.

$$\underset{\le kn.}{O(n)} - \overset{\text{const} \times}{\downarrow} n$$

To make them cancel, set const. = k.

We should design our potential function to pay for "unbounded" costs.

FibHeap decreasekey



true cost is $O(L)$

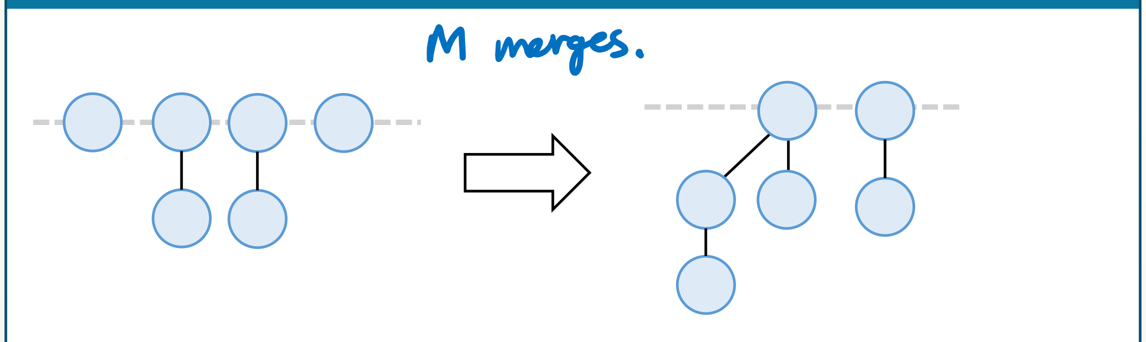
Can we make decreasekey be $O(1)$?

We'd need $\Delta\Phi = -L$.

IDEA: put credit on each loser, and release the credit when loser is moved to root.

So: $\Phi = \text{const} \times \#\text{losers}$.

FibHeap popmin / cleanup



true cost is $O(M + \log N)$ $N = \#$ items in heap.

Can we make popmin be $O(\log N)$?

Can we get $\Delta\Phi = -M$?

THINK: put credit on each tree/root. Release the credit when the tree is made a child.

So: $\Phi = \text{const} \times \#\text{trees}$.

So: $\Phi = \boxed{\text{const}_1} \times \#\text{trees} + \boxed{\text{const}_2} \times \#\text{losers}$.

Finally: look meticulously at all the operations to see if there's a tradeoff between the two constants,