HEAPS:	REGULAR	BINOMIAL
REPORT MIN:	$O(1)$	←
EXTRACT MIN:	$O(\log n)$	←
INSERT:	$O(\log n)$ $O(1)$ amortized	←
DECREASE KEY:	$O(\log n)$	←
DELETE:	$O(\log n)$	←
MERGE/UNION:	$O(n)$	$O(\log n)$

HEAPS:	REGULAR	BINOMIAL	QUAKE
REPORT MIN:	$O(1)$	←	←
EXTRACT MIN:	$O(\log n)$	←	$O(n)$ $O(\log n)$ amortized
INSERT:	$O(\log n)$ $O(1)$ amortized	←	$O(1)$
DECREASE KEY:	$O(\log n)$	←	$O(1)$
DELETE:	$O(\log n)$	←	$O(n)$ $O(\log n)$ amortized
MERGE/UNION:	$O(n)$	$O(\log n)$	$O(1)$

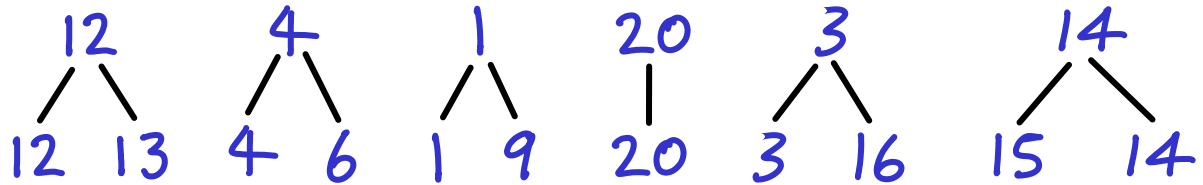
TOURNAMENT TREES

- ALL DATA STORED IN LEAVES AT LOWEST LEVEL

12 13 4 6 1 9 20 3 16 15 14

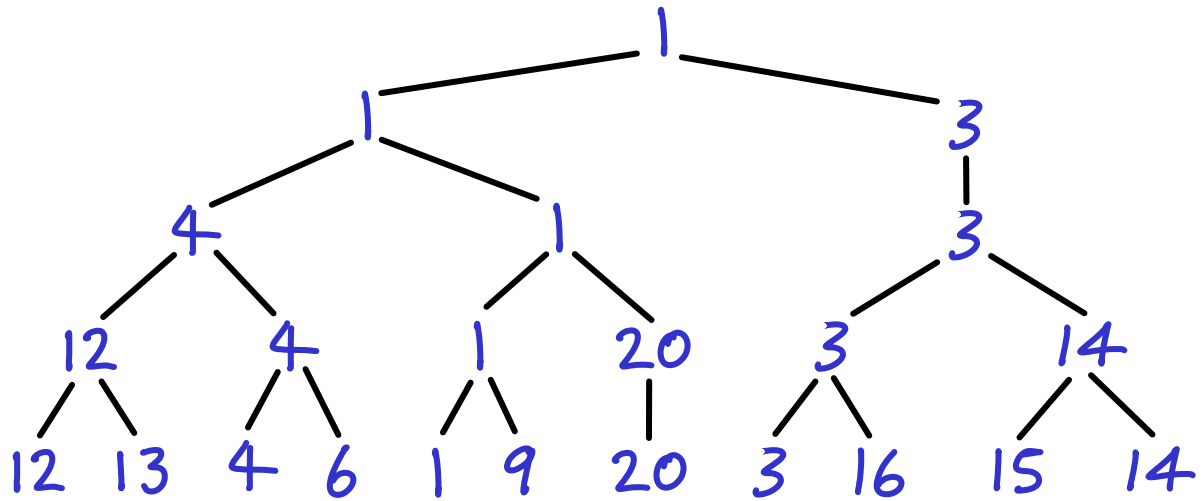
TOURNAMENT TREES

- ALL DATA STORED IN LEAVES AT LOWEST LEVEL
- PARENT NODE = MIN{CHILDREN}



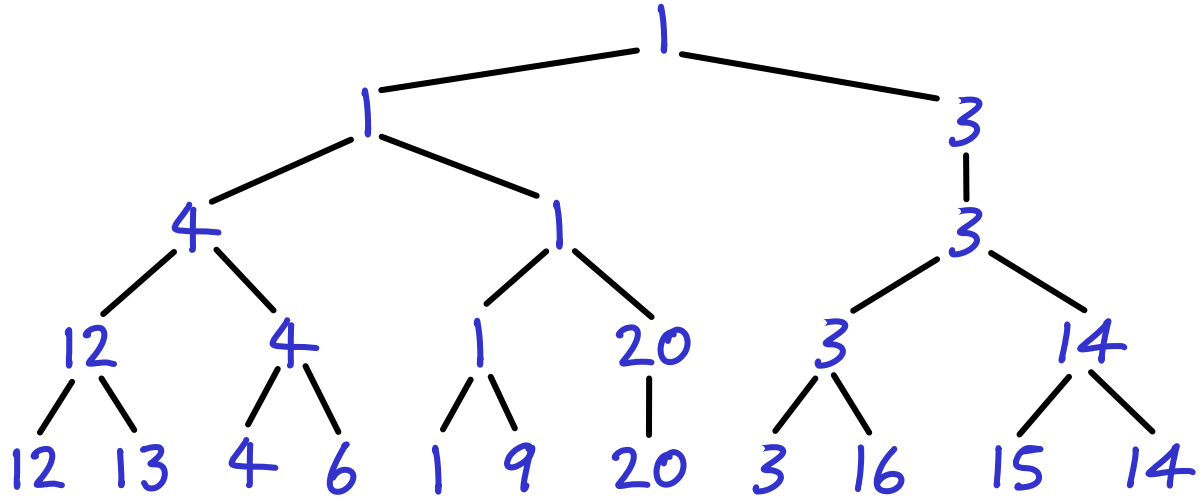
TOURNAMENT TREES

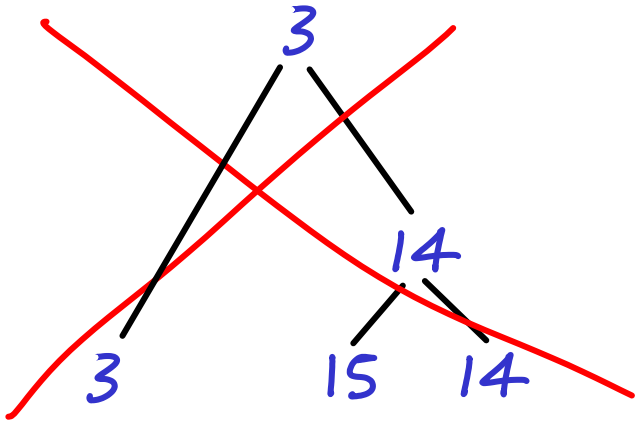
- ALL DATA STORED IN LEAVES AT LOWEST LEVEL
- PARENT NODE = MIN{CHILDREN}
- EVERY NODE HAS 1 OR 2 CHILDREN



TOURNAMENT TREES

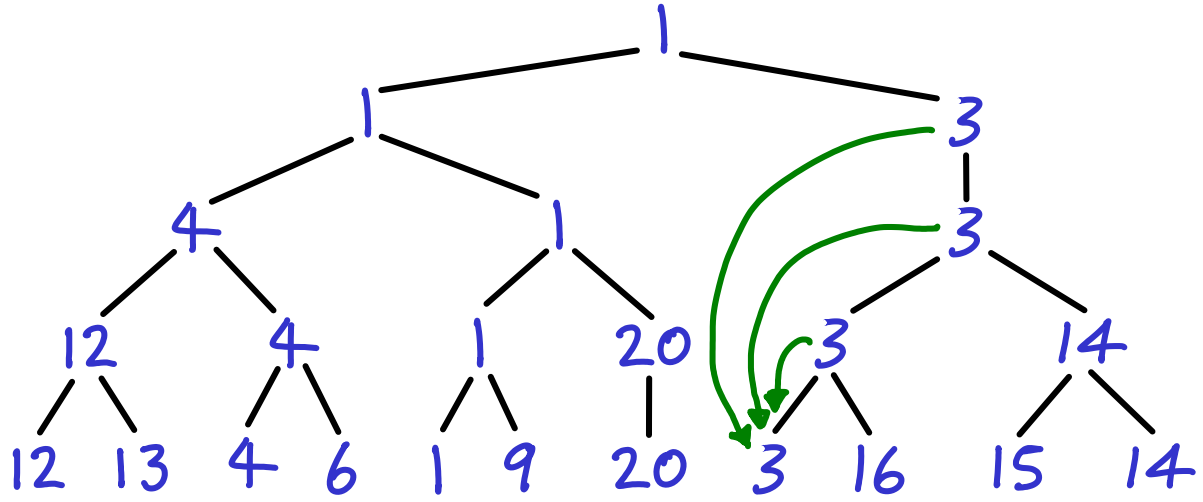
- ALL DATA STORED IN LEAVES AT LOWEST LEVEL
  (EDGES CAN'T SKIP LEVELS)
- PARENT NODE = MIN{CHILDREN}
- EVERY NODE HAS 1 OR 2 CHILDREN

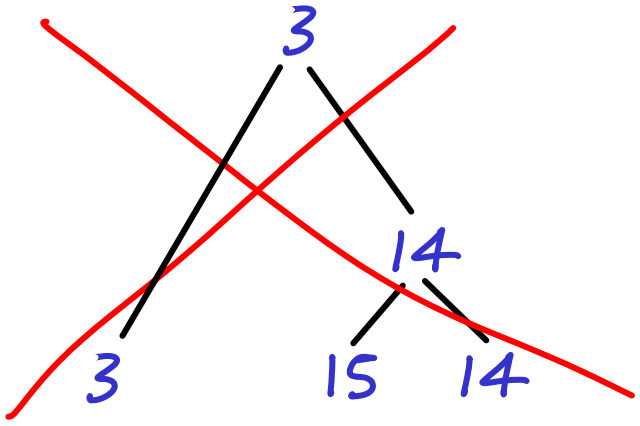




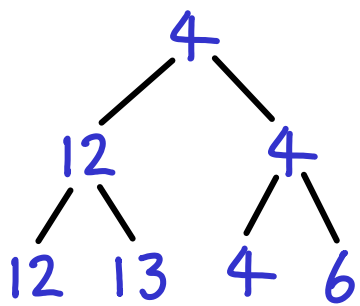
TOURNAMENT TREES

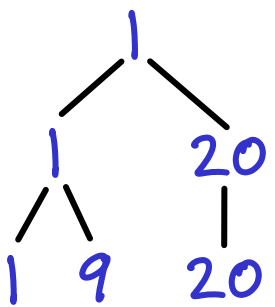
- ALL DATA STORED IN LEAVES AT LOWEST LEVEL
  (EDGES CAN'T SKIP LEVELS)
- PARENT NODE = MIN{CHILDREN} (CLONE = use pointer)
- EVERY NODE HAS 1 OR 2 CHILDREN

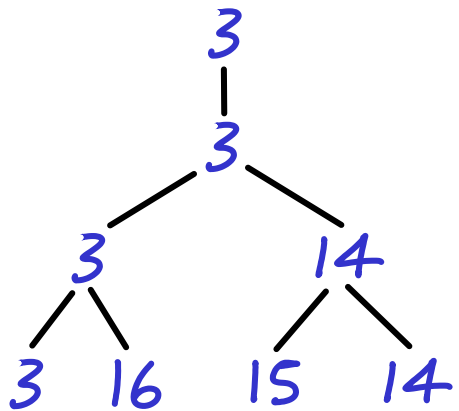




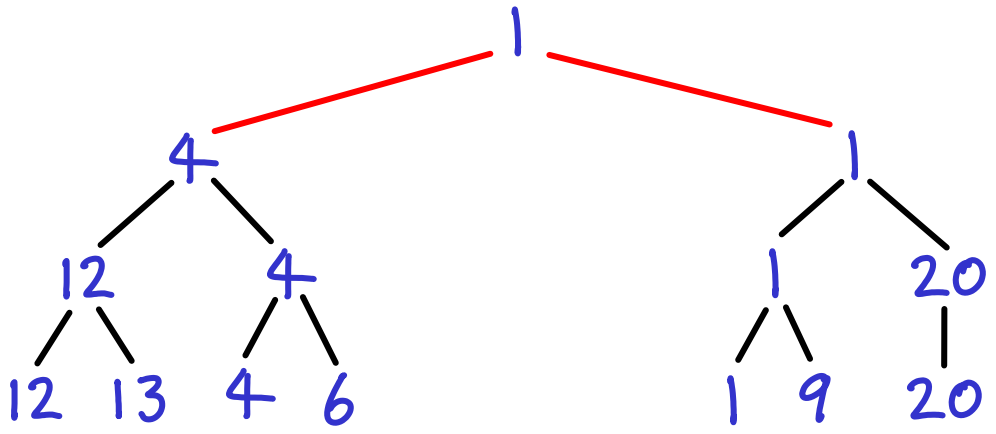
LINK (2 trees) : assumes same height

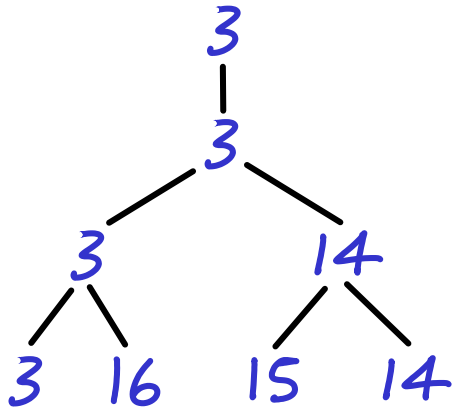




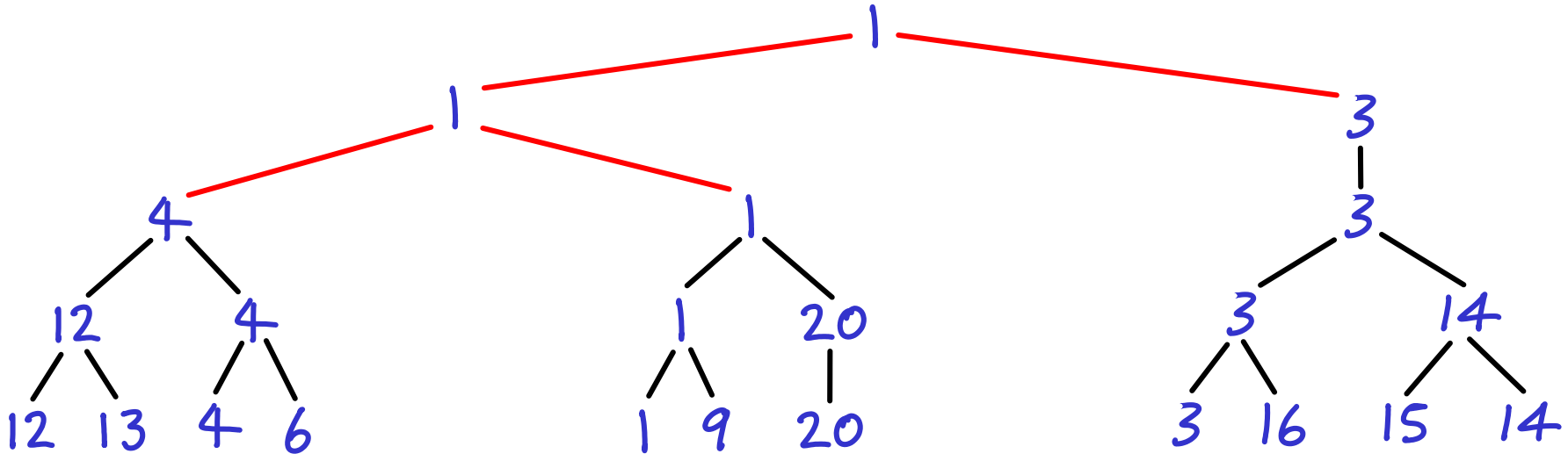


LINK (2 trees) : assumes same height → Make new root & clone



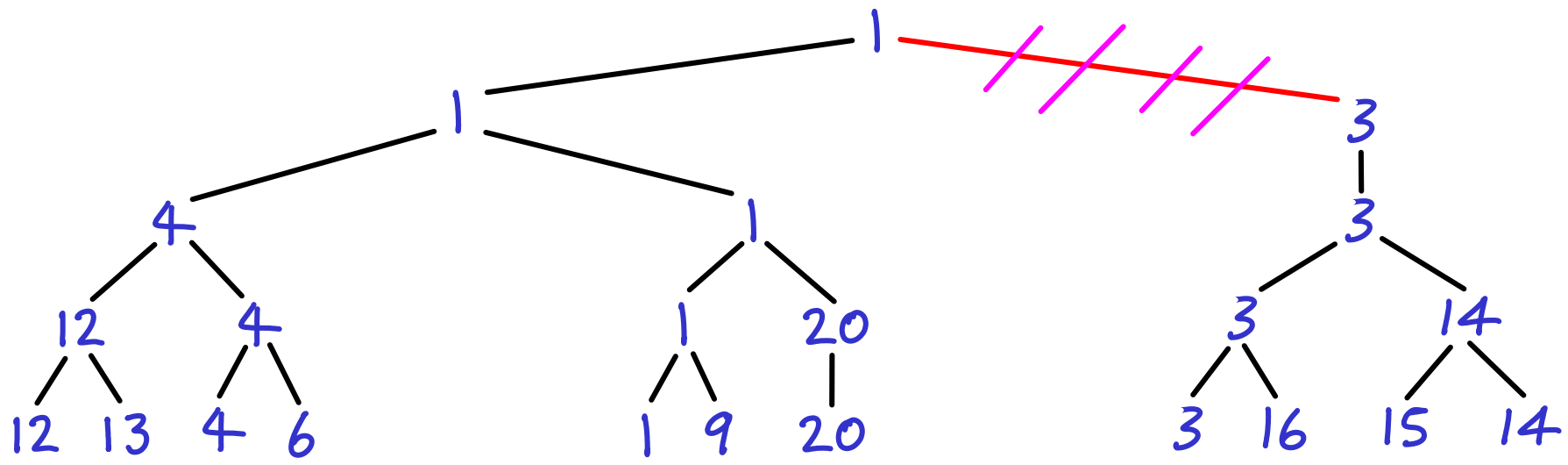


LINK (2 trees) : assumes same height → Make new root & clone



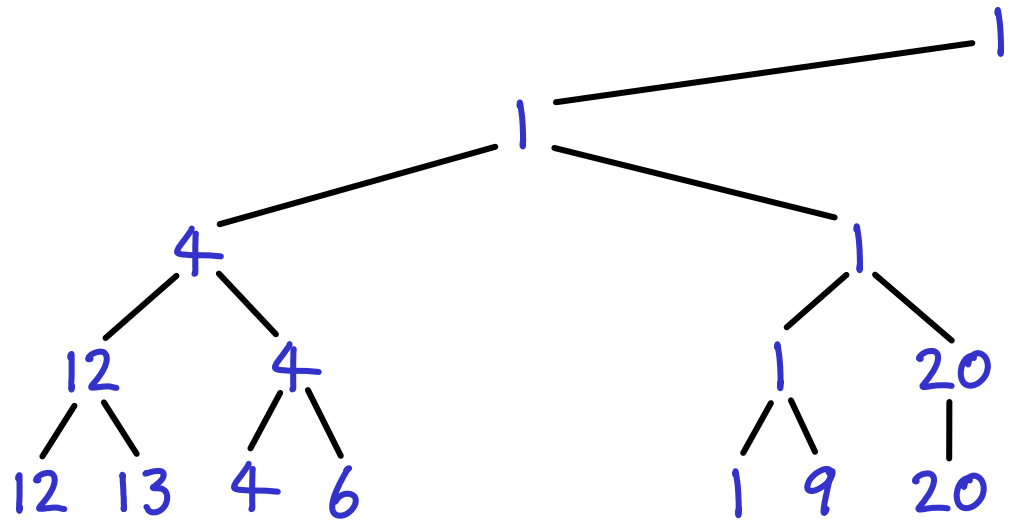
LINK (2 trees) : assumes same height → Make new root & clone

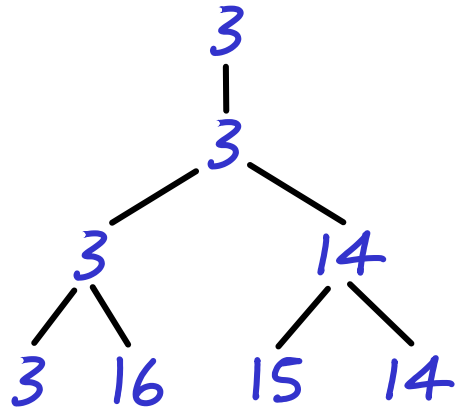
CUT (at an edge)



LINK (2 trees) : assumes same height → Make new root & clone

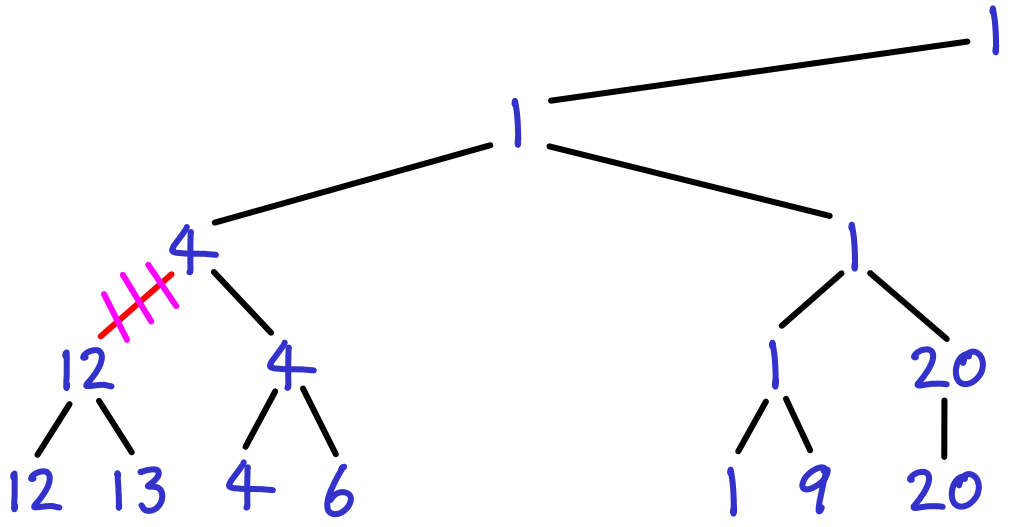
CUT (at an edge) : → Trivial : 2 new trees

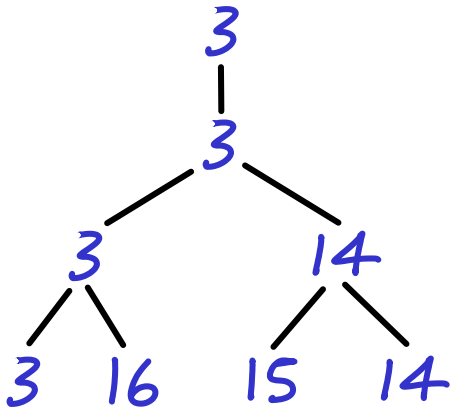




LINK (2 trees) : assumes same height → Make new root & clone

CUT (at an edge) : → Trivial : 2 new trees

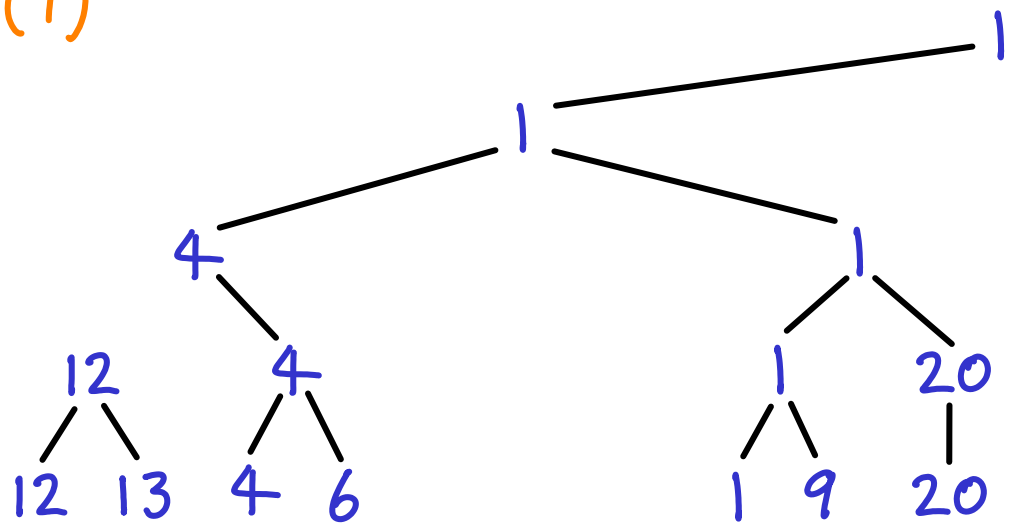


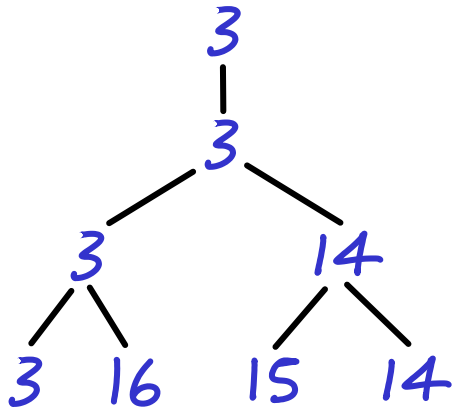


LINK (2 trees) : assumes same height → Make new root & clone

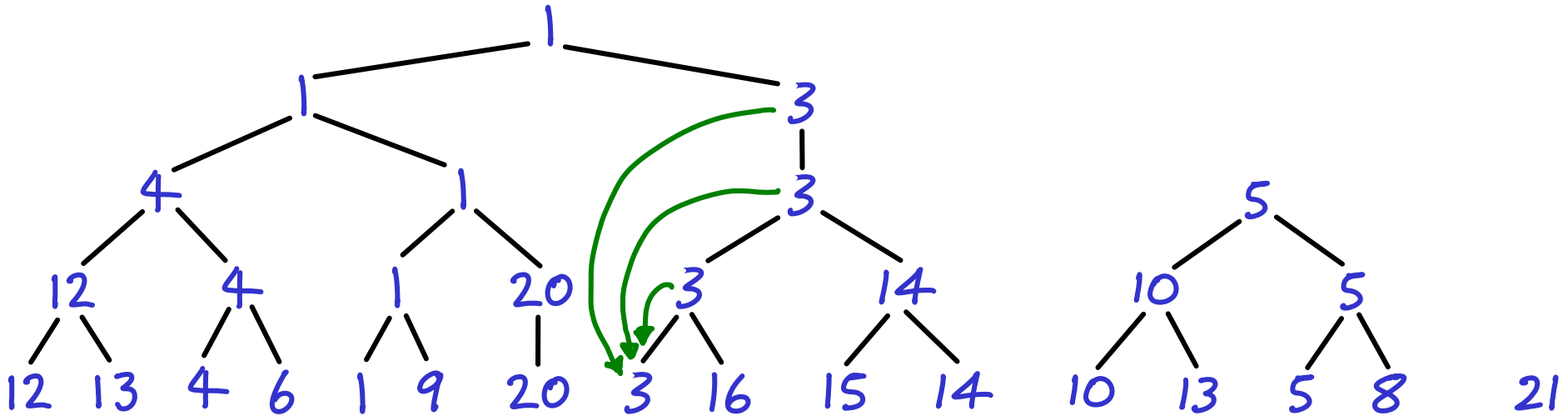
CUT (at an edge) : → Trivial : 2 new trees

O(1)





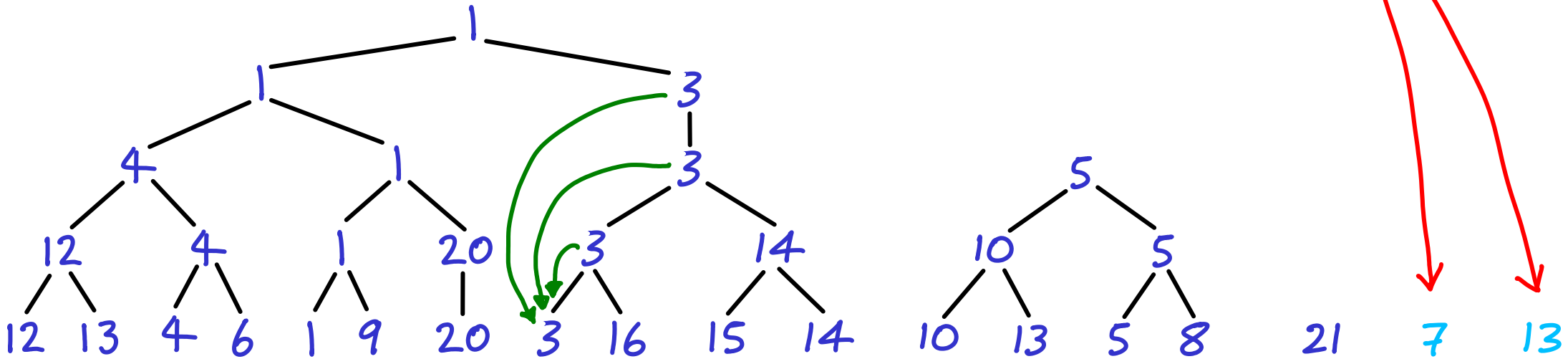
QUAKE HEAP ~ collection of TOURNAMENT TREES with distinct heights



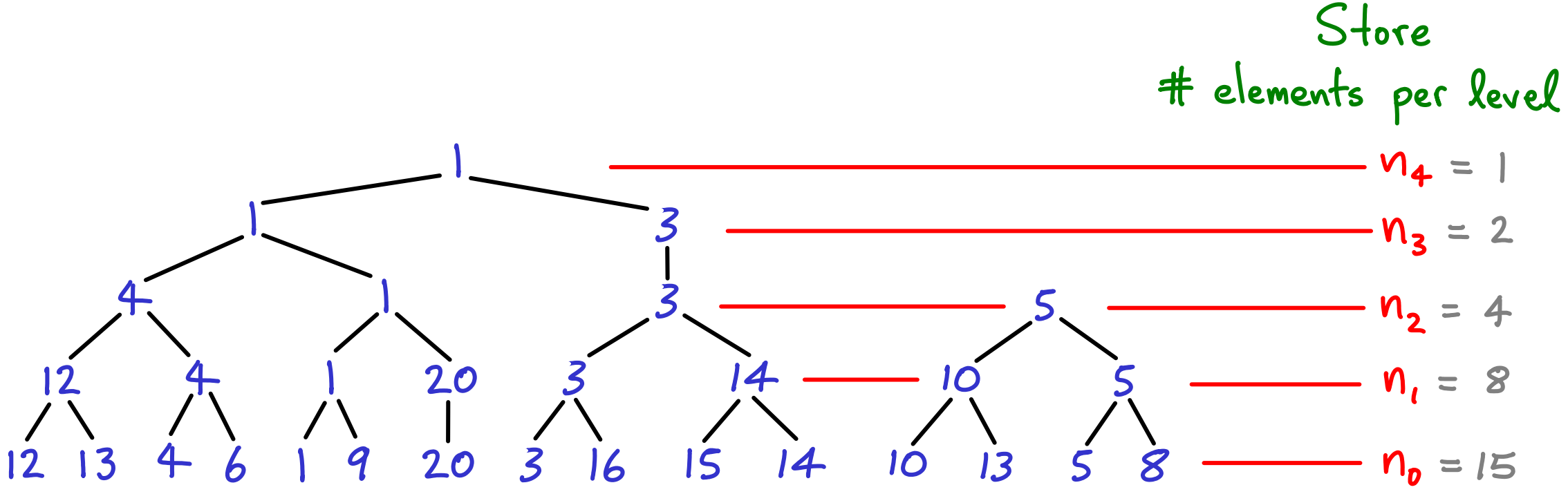
QUAKE HEAP ~ collection of TOURNAMENT TREES

with distinct heights ... most of the time

LAZY INSERT

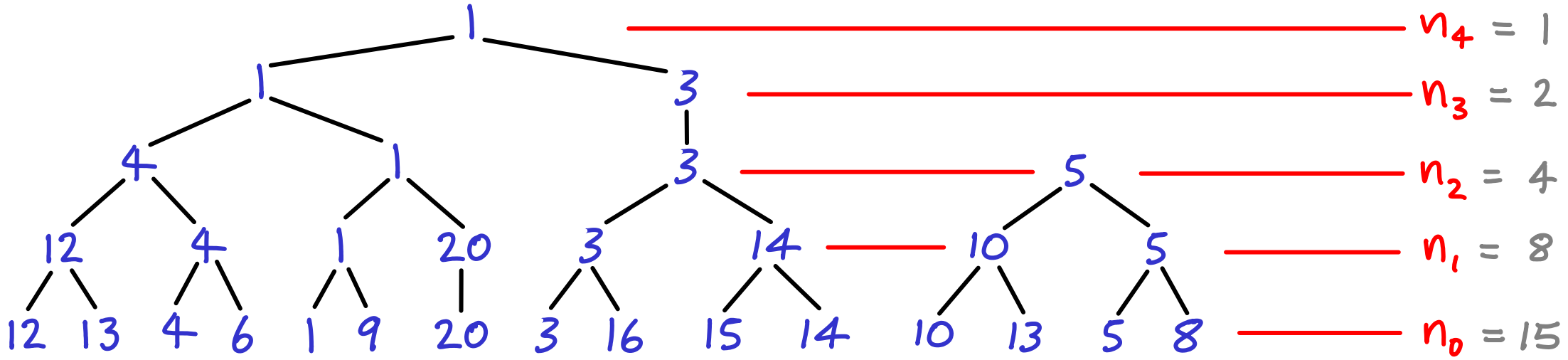


QUAKE HEAP ~ collection of TOURNAMENT TREES with distinct heights ... most of the time



QUAKE HEAP ~ collection of TOURNAMENT TREES with distinct heights ... most of the time

Enforce $\frac{n_{i+1}}{n_i} \leq \alpha \quad \left(\frac{1}{2} < \alpha < 1\right)$

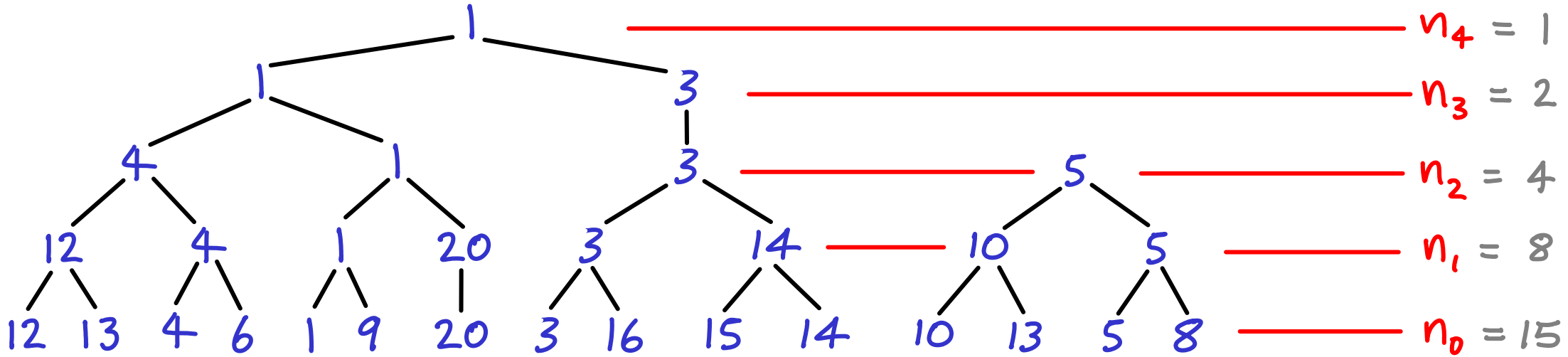


QUAKE HEAP ~ collection of TOURNAMENT TREES with distinct heights ... most of the time

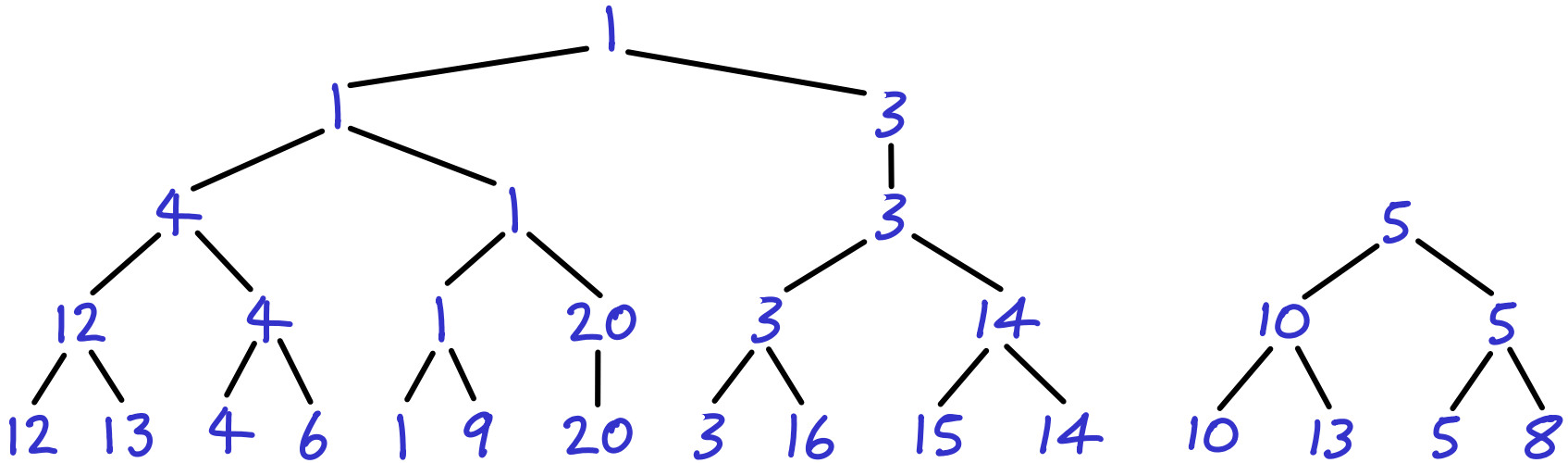
Enforce $\frac{n_{i+1}}{n_i} \leq \alpha \quad \left(\frac{1}{2} < \alpha < 1\right)$

We will use $\alpha = \frac{3}{4}$

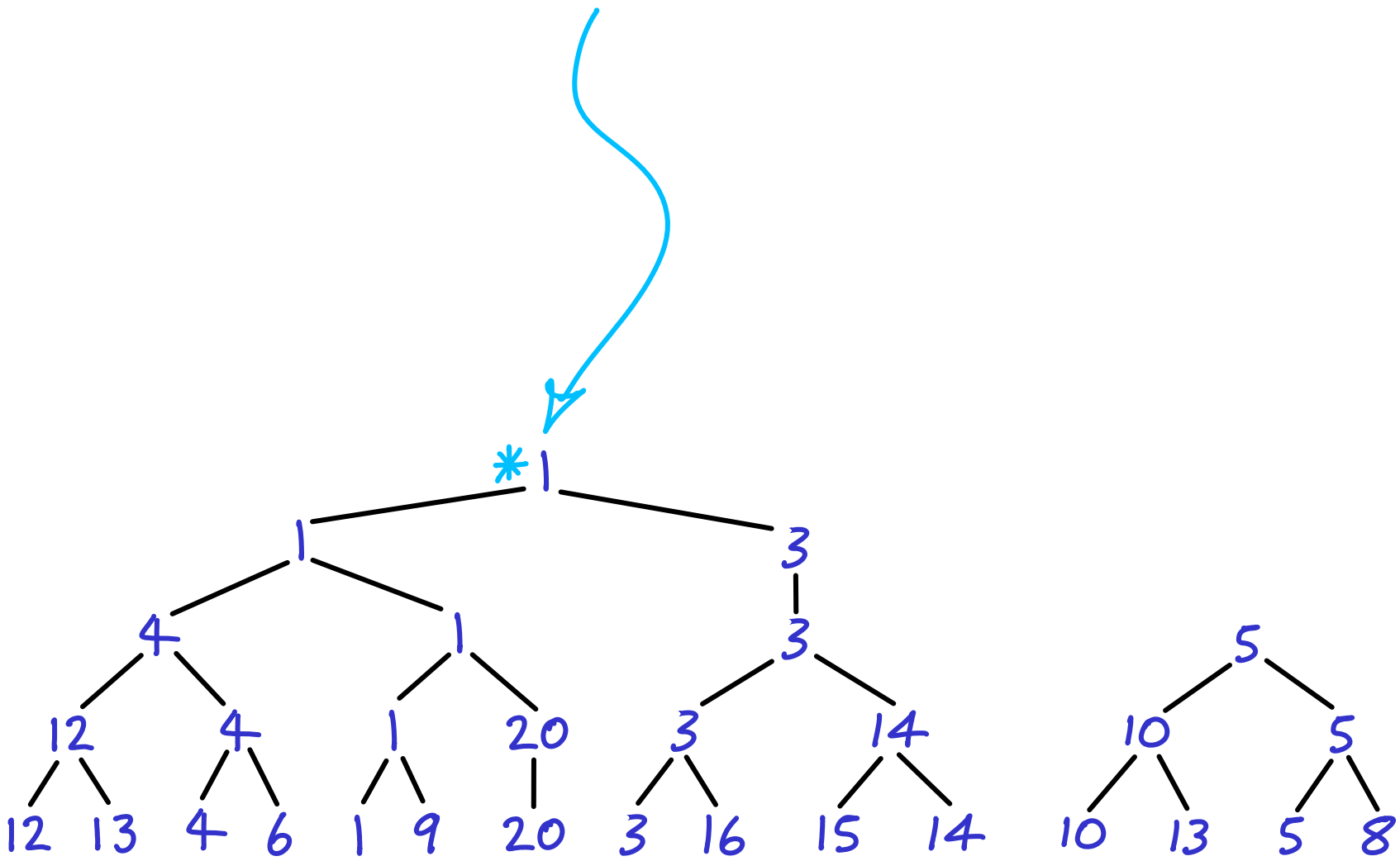
MAX HEIGHT $\leq \log_{4/3} n_0$



REPORT MIN: ?

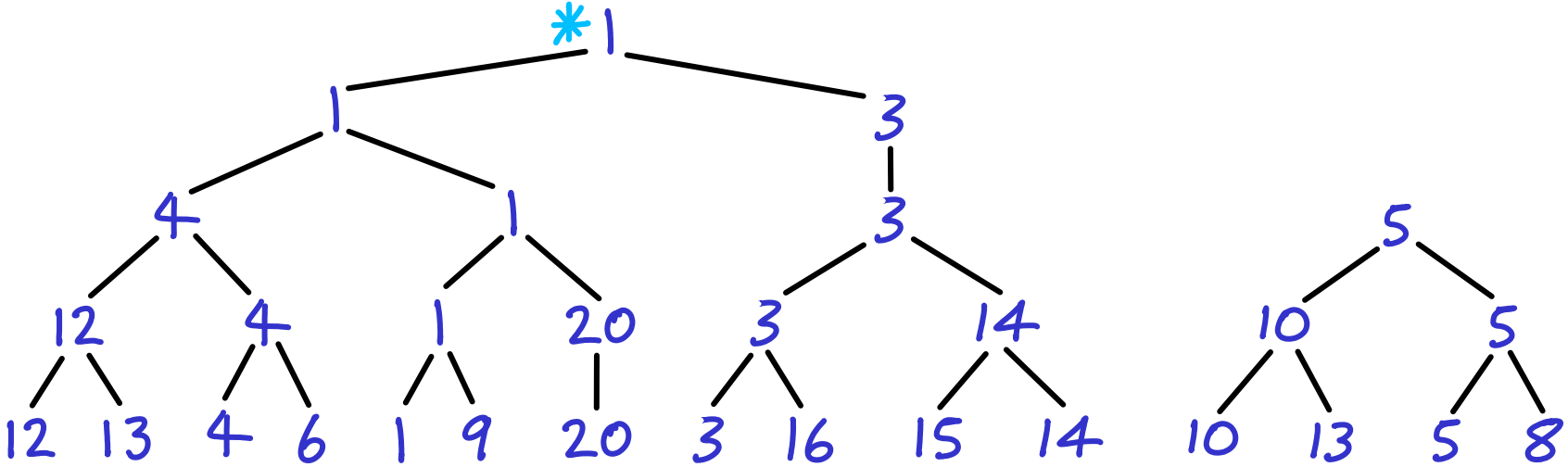


REPORT MIN: KEEP POINTER O(1)



REPORT MIN: KEEP POINTER O(1)

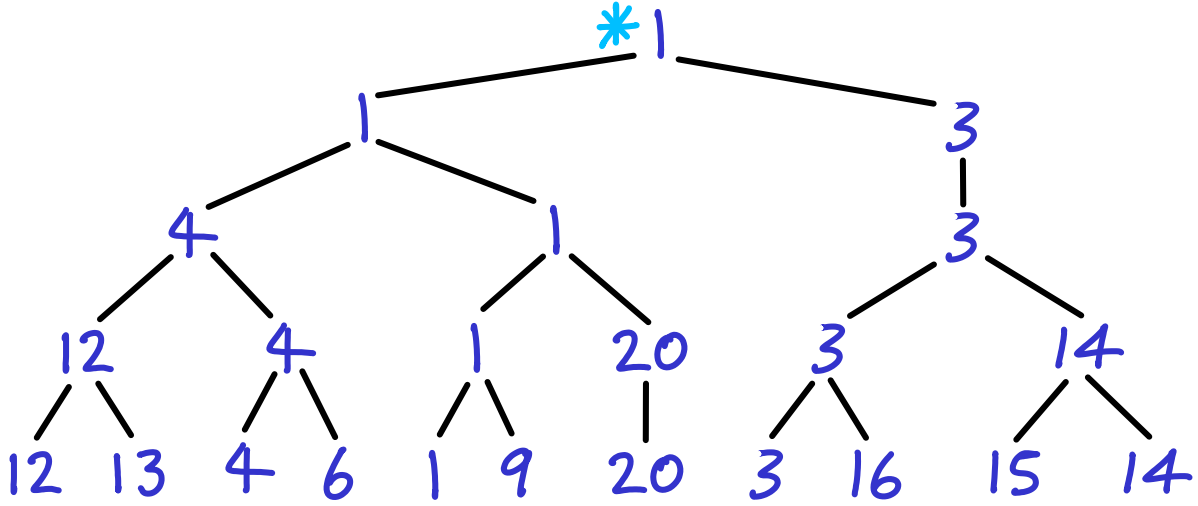
INSERT: ?

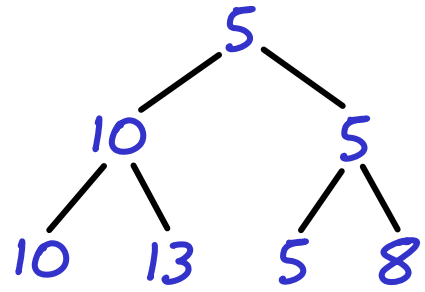


REPORT MIN: KEEP POINTER $O(1)$

INSERT: increment n_0 $O(1)$

$$\frac{n_{i+1}}{n_i} \leq \frac{3}{4}$$





$n_4 = 1$

$n_3 = 2$

$n_2 = 3$

$n_1 = 8$

21

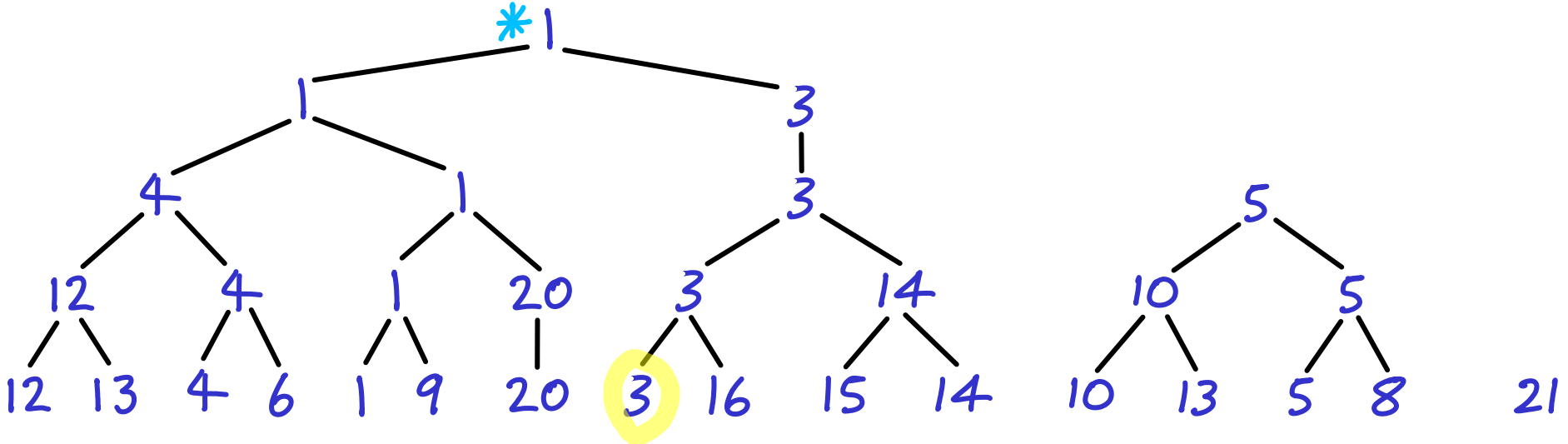
$n_0 = 16$

REPORT MIN: KEEP POINTER $O(1)$

INSERT: increment n_0 $O(1)$

DECREASE KEY: ?

$$\frac{n_{i+1}}{n_i} \leq \frac{3}{4}$$



21

$n_4 = 1$

$n_3 = 2$

$n_2 = 3$

$n_1 = 8$

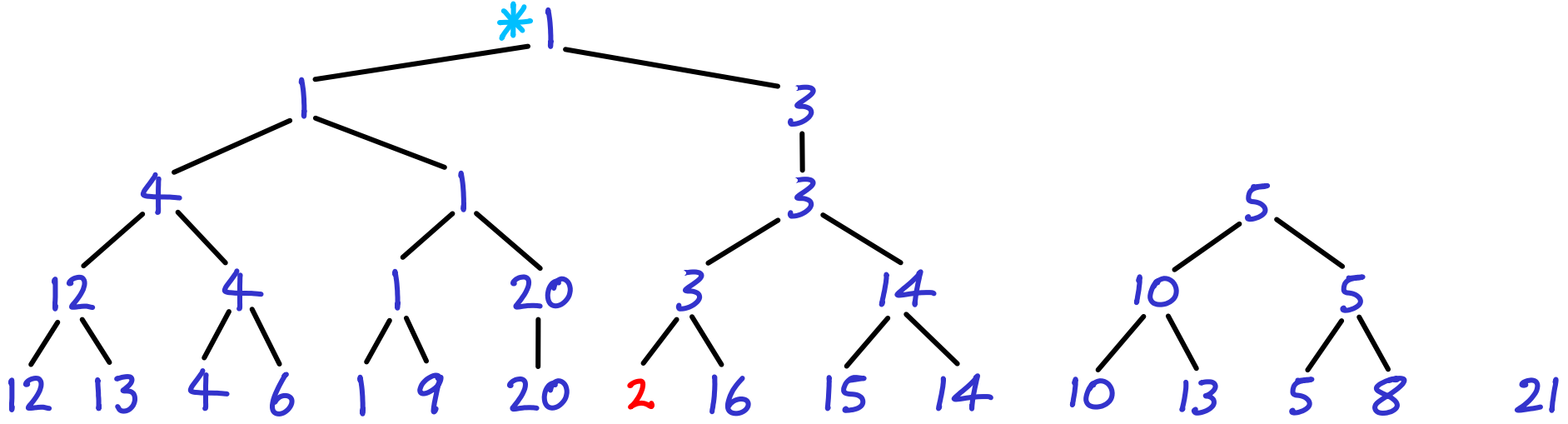
$n_0 = 16$

REPORT MIN: KEEP POINTER $O(1)$

INSERT: increment n_0 $O(1)$

DECREASE KEY:

$$\frac{n_{i+1}}{n_i} \leq \frac{3}{4}$$



$n_4 = 1$

$n_3 = 2$

$n_2 = 3$

$n_1 = 8$

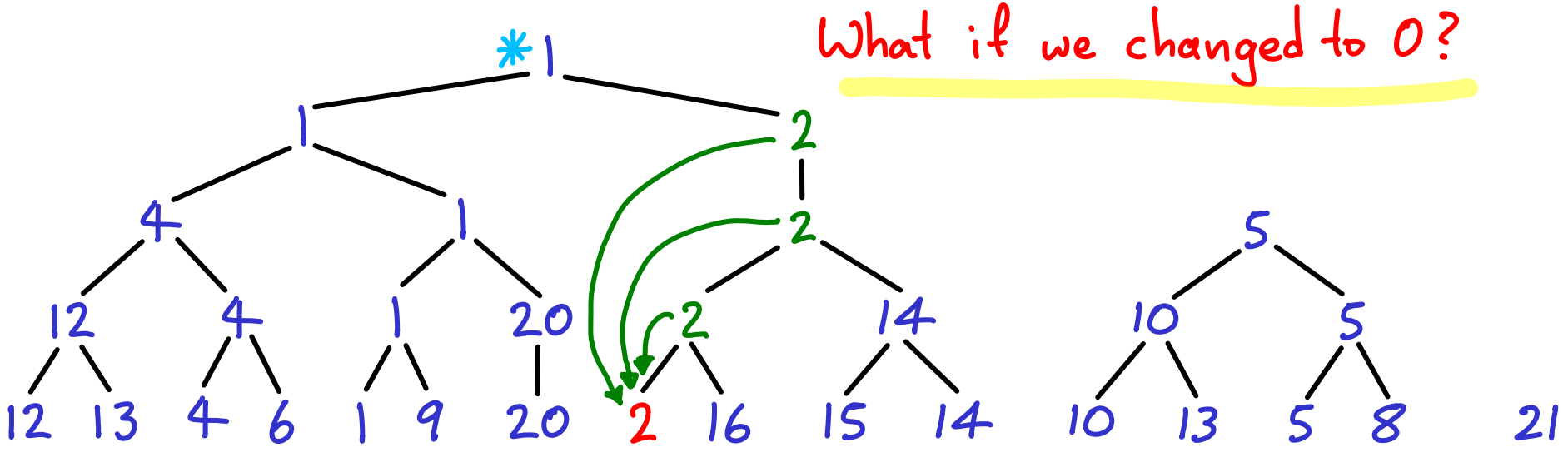
$n_0 = 16$

REPORT MIN: KEEP POINTER $O(1)$

INSERT: increment n_0 $O(1)$

DECREASE KEY: clones reflect changed leaf $O(1)$ ...

$\frac{n_{i+1}}{n_i} \leq \frac{3}{4}$



What if we changed to 0?

$n_4 = 1$

$n_3 = 2$

$n_2 = 3$

$n_1 = 8$

$n_0 = 16$

REPORT MIN: KEEP POINTER $O(1)$

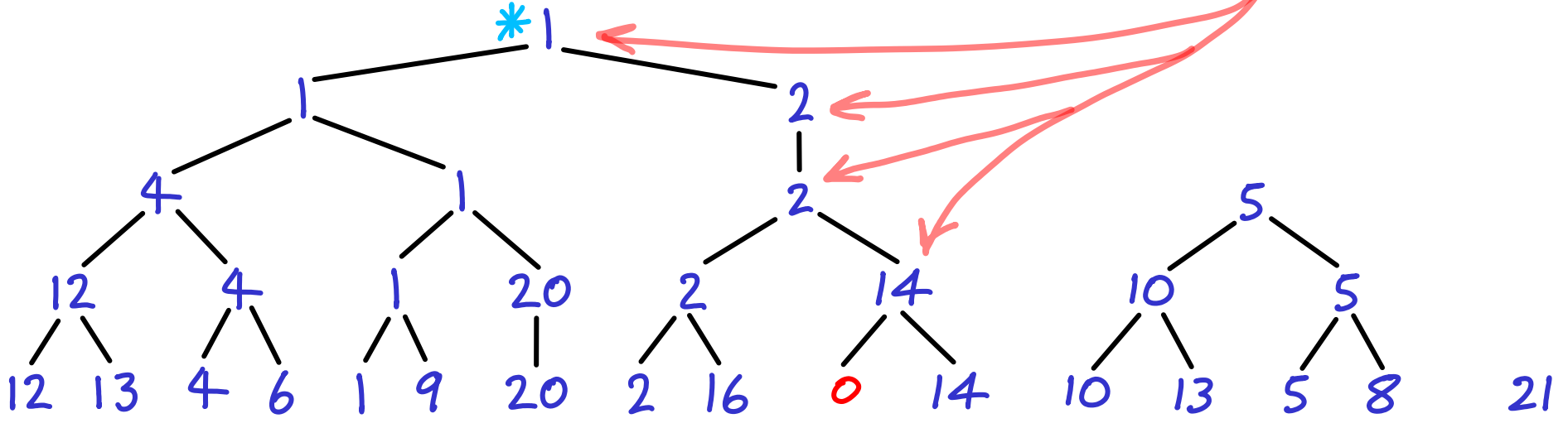
$\frac{n_{i+1}}{n_i} \leq \frac{3}{4}$

INSERT: increment n_0 $O(1)$

DECREASE KEY: clones reflect changed leaf.

too expensive to update up to root

↳ $O(\log n)$



$n_4 = 1$

$n_3 = 2$

$n_2 = 3$

$n_1 = 8$

$n_0 = 16$

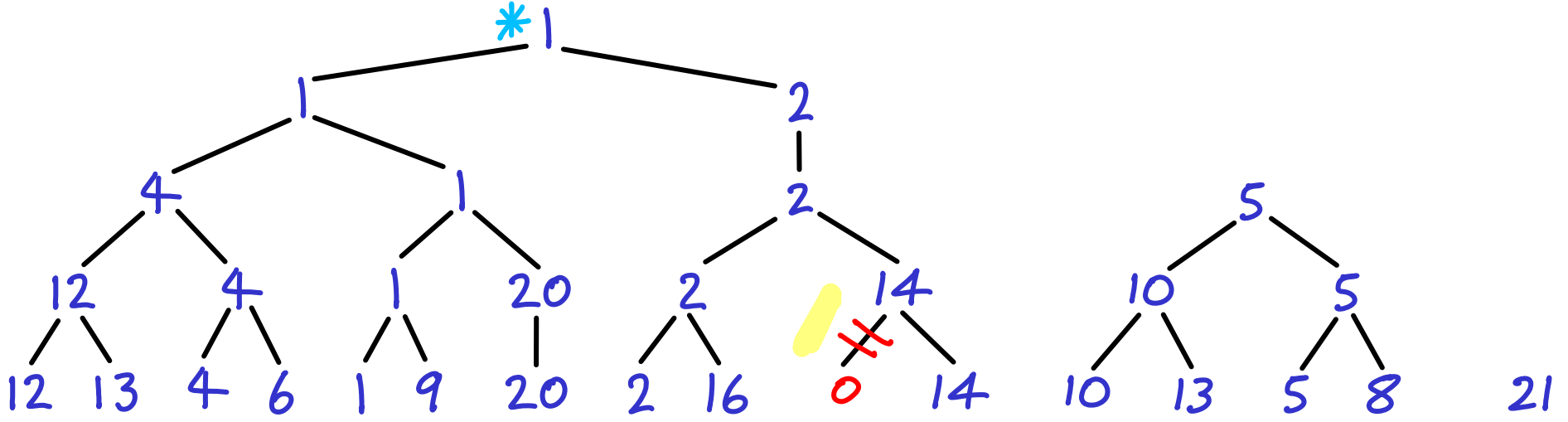
REPORT MIN: KEEP POINTER $O(1)$

$\frac{n_{i+1}}{n_i} \leq \frac{3}{4}$

INSERT: increment n_0 $O(1)$

DECREASE KEY: clones reflect changed leaf. Just CUT

too expensive to update up to root



$n_4 = 1$

$n_3 = 2$

$n_2 = 3$

$n_1 = 8$

$n_0 = 16$

REPORT MIN: KEEP POINTER $O(1)$

INSERT: increment n_0 $O(1)$

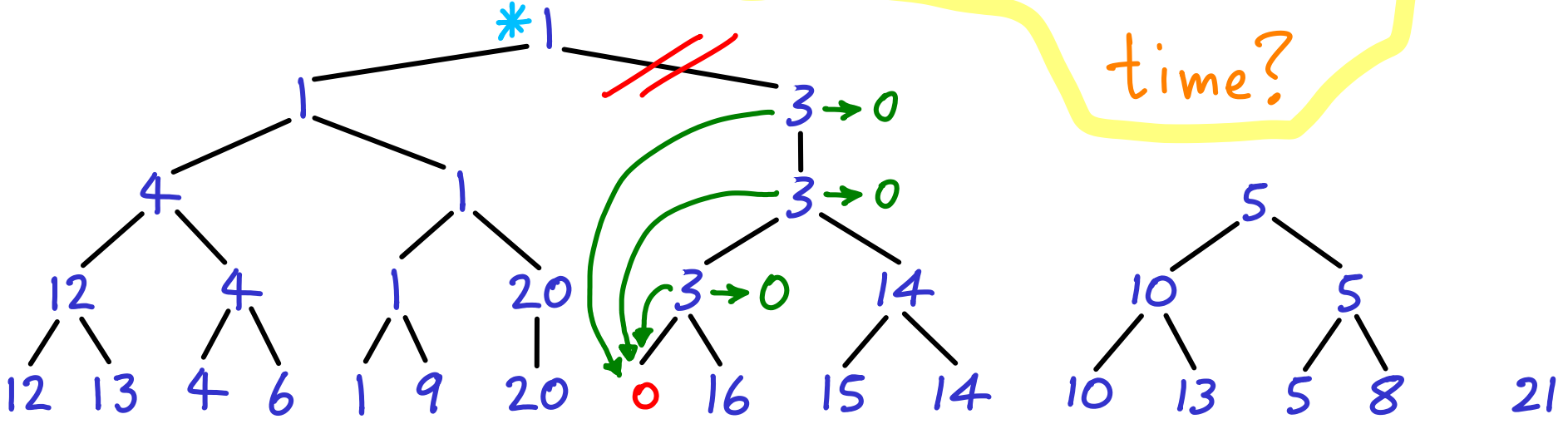
DECREASE KEY: clones reflect changed leaf.

too expensive to update up to root

CUT above highest clone

time?

$\frac{n_{i+1}}{n_i} \leq \frac{3}{4}$



$n_4 = 1$

$n_3 = 2$

$n_2 = 3$

$n_1 = 8$

$n_0 = 16$

REPORT MIN: KEEP POINTER $O(1)$

INSERT: increment n_0 $O(1)$

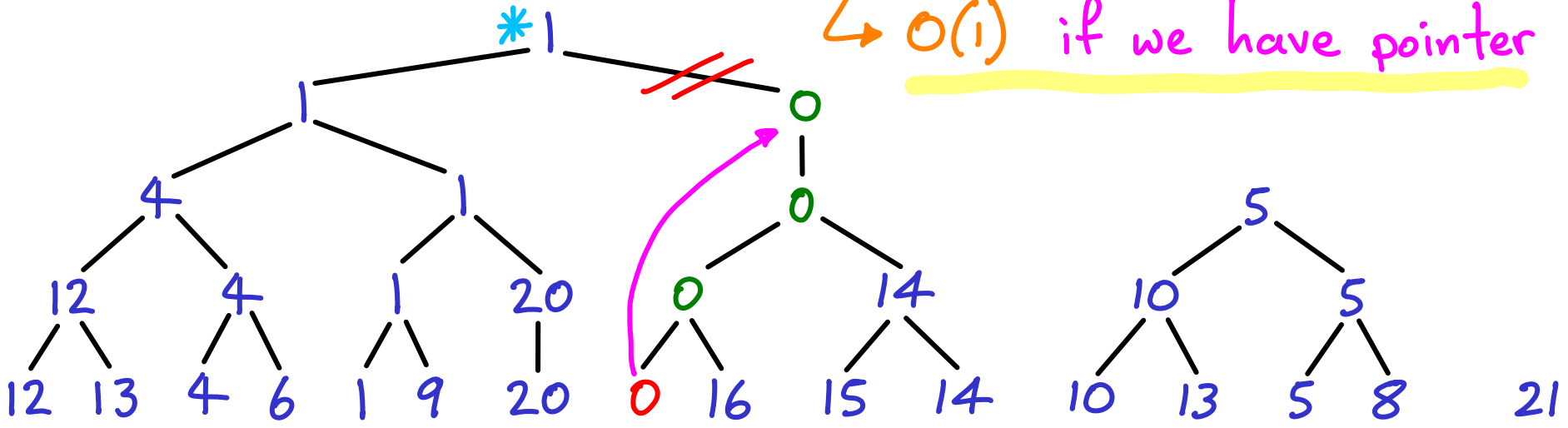
DECREASE KEY: clones reflect changed leaf.

too expensive to update up to root

CUT above highest clone

↳ $O(1)$ if we have pointer

$$\frac{n_{i+1}}{n_i} \leq \frac{3}{4}$$



$n_4 = 1$

$n_3 = 2$

$n_2 = 3$

$n_1 = 8$

$n_0 = 16$

REPORT MIN: KEEP POINTER $O(1)$

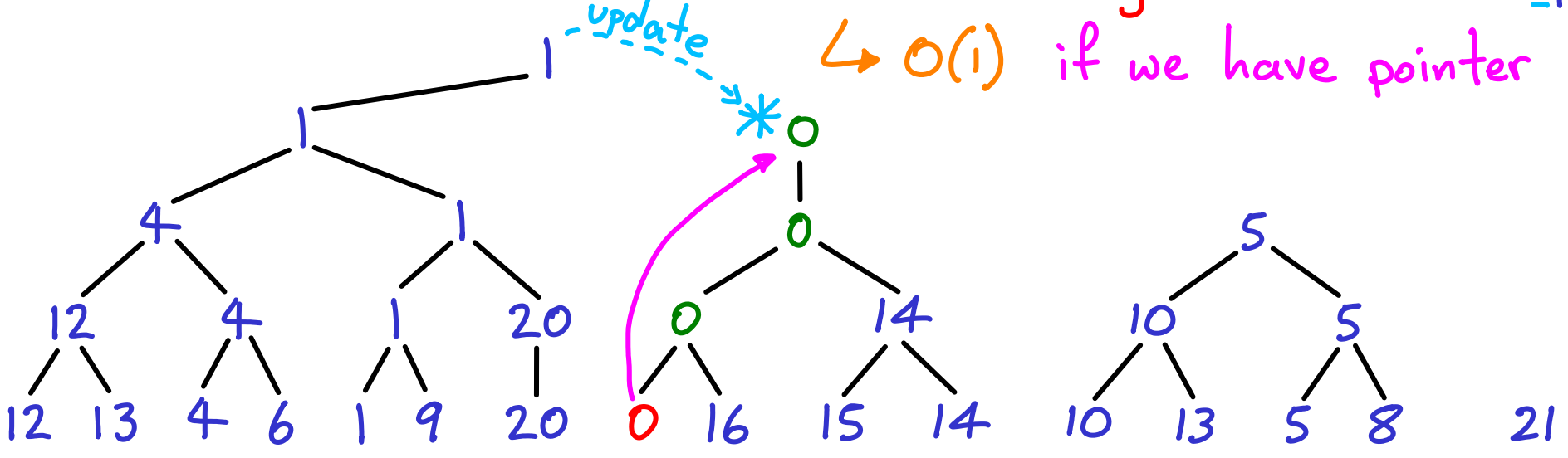
INSERT: increment n_0 $O(1)$

DECREASE KEY:
- clones reflect changed leaf.
- too expensive to update up to root

CUT above highest clone & update MIN*

↳ $O(1)$ if we have pointer

$$\frac{n_{i+1}}{n_i} \leq \frac{3}{4}$$



$n_4 = 1$

$n_3 = 2$

$n_2 = 3$

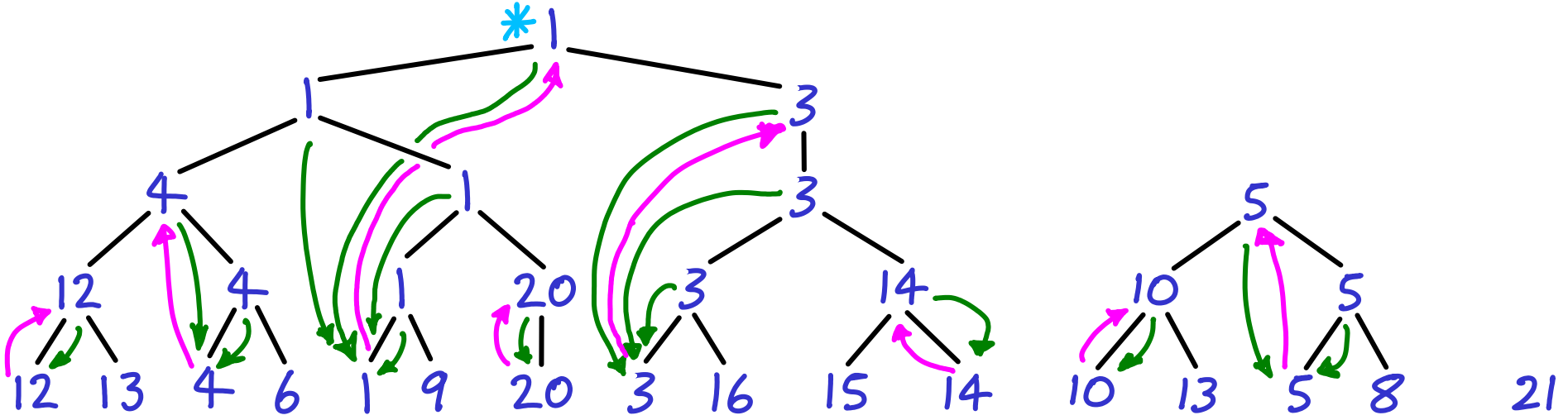
$n_1 = 8$

$n_0 = 16$

REPORT MIN, INSERT: $O(1)$

DECREASE KEY: CUT above highest clone $O(1)$

Storing: clone pointers, leaf-to-highest-clone, global min *, level counts ↘



$\frac{n_{i+1}}{n_i} \leq \frac{3}{4}$

$n_4 = 1$

$n_3 = 2$

$n_2 = 3$

$n_1 = 8$

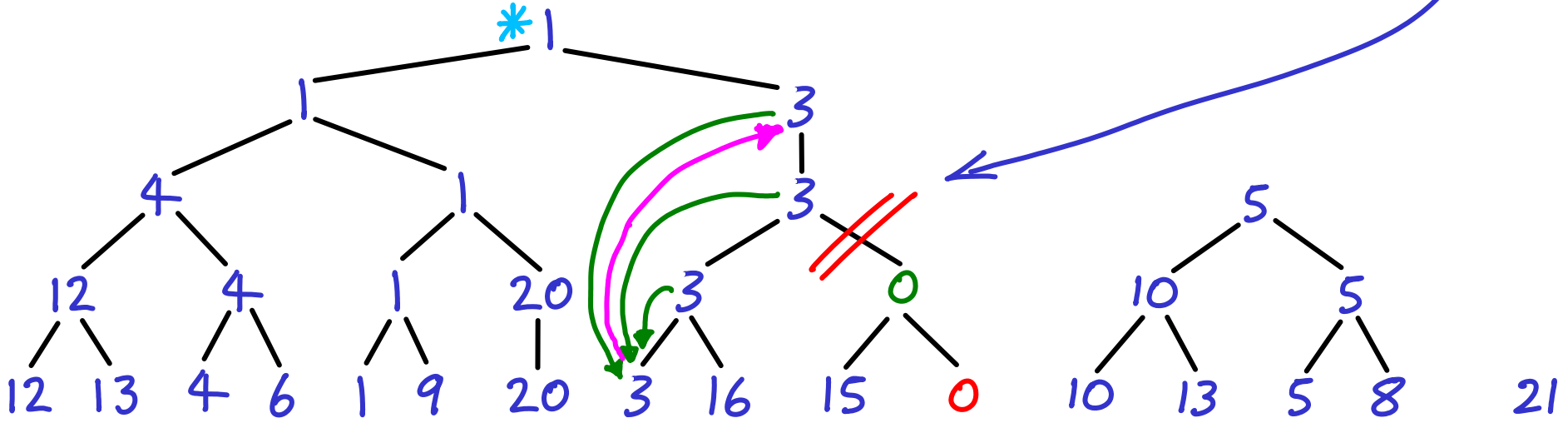
$n_0 = 16$

REPORT MIN, INSERT: $O(1)$

DECREASE KEY: CUT above highest clone $O(1)$

Storing: clone pointers, leaf-to-highest-clone, global min*, level counts

Also want distinct tree heights; so far we are extra lazy



$$\frac{n_{i+1}}{n_i} \leq \frac{3}{4}$$

$n_4 = 1$

$n_3 = 2$

$n_2 = 3$

$n_1 = 8$

$n_0 = 16$

REPORT MIN, INSERT: $O(1)$

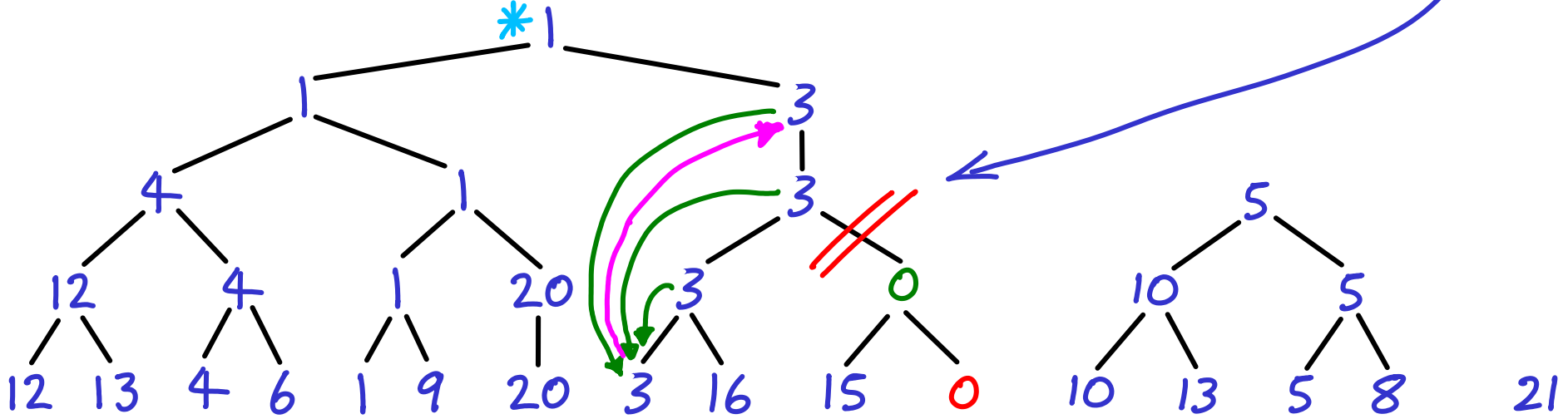
DECREASE KEY: CUT above highest clone $O(1)$

Storing: clone pointers, leaf-to-highest-clone, global min*, level counts

Also want distinct tree heights; so far we are extra lazy

UNION: be even more lazy: $O(1)$

↳ keep level counts separate for now, otherwise $O(\log n)$.



$\frac{n_{i+1}}{n_i} \leq \frac{3}{4}$

$n_4 = 1$

$n_3 = 2$

$n_2 = 3$

$n_1 = 8$

$n_0 = 16$

REPORT MIN, INSERT, DECREASE KEY, UNION: $O(1)$

So far we could have used a ...

REPORT MIN, INSERT, DECREASE KEY, UNION: $O(1)$

So far we could have used a linked list

REPORT MIN, INSERT, DECREASE KEY, UNION: $O(1)$

So far we could have used a linked list

12 13 4 6 1* 9 20 3 16 15 14 10 13 5 8 21

We need to DELETE (& EXTRACT-MIN = FIND-MIN + DELETE)

in $O(n)$ but $O(\log n)$ amortized (Pay for being lazy)

REPORT MIN, INSERT, DECREASE KEY, UNION: $O(1)$

So far we could have used a linked list

12 13 4 6 1* 9 20 3 16 15 14 10 13 5 8 21

We need to DELETE (& EXTRACT-MIN = FIND-MIN + DELETE)

in $O(n)$ but $O(\log n)$ amortized (Pay for being lazy)

We will also need to deal with separate level counts.

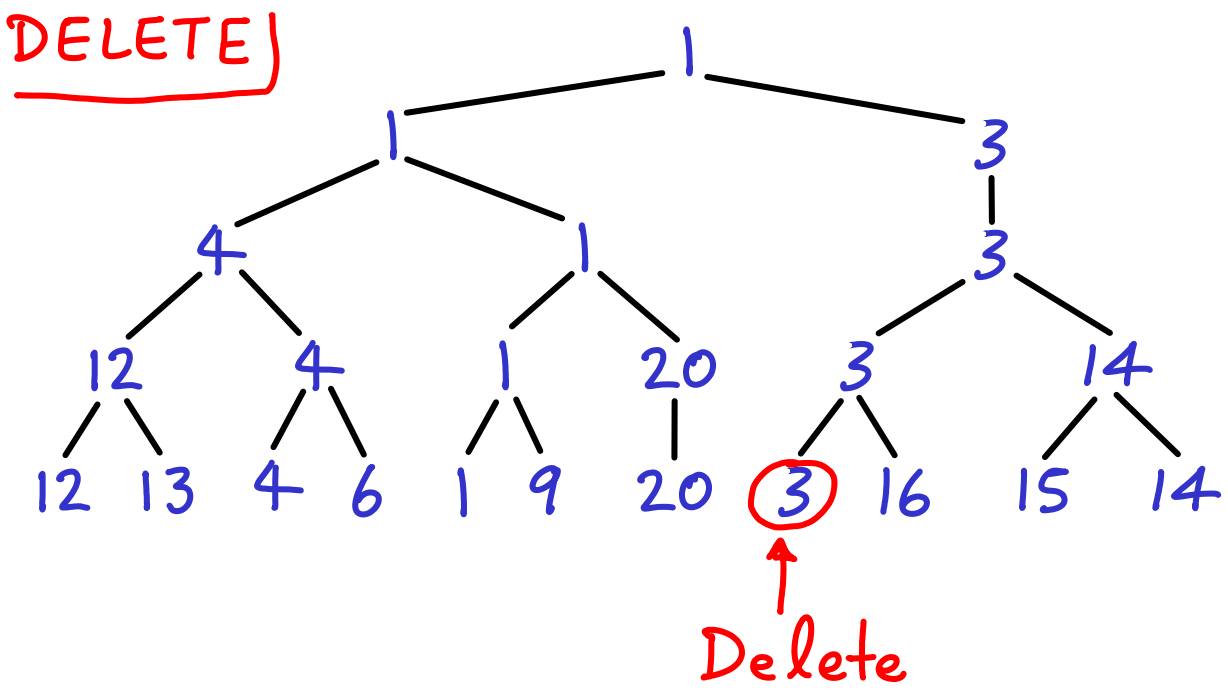
↳ (only if UNION is needed)

Next:

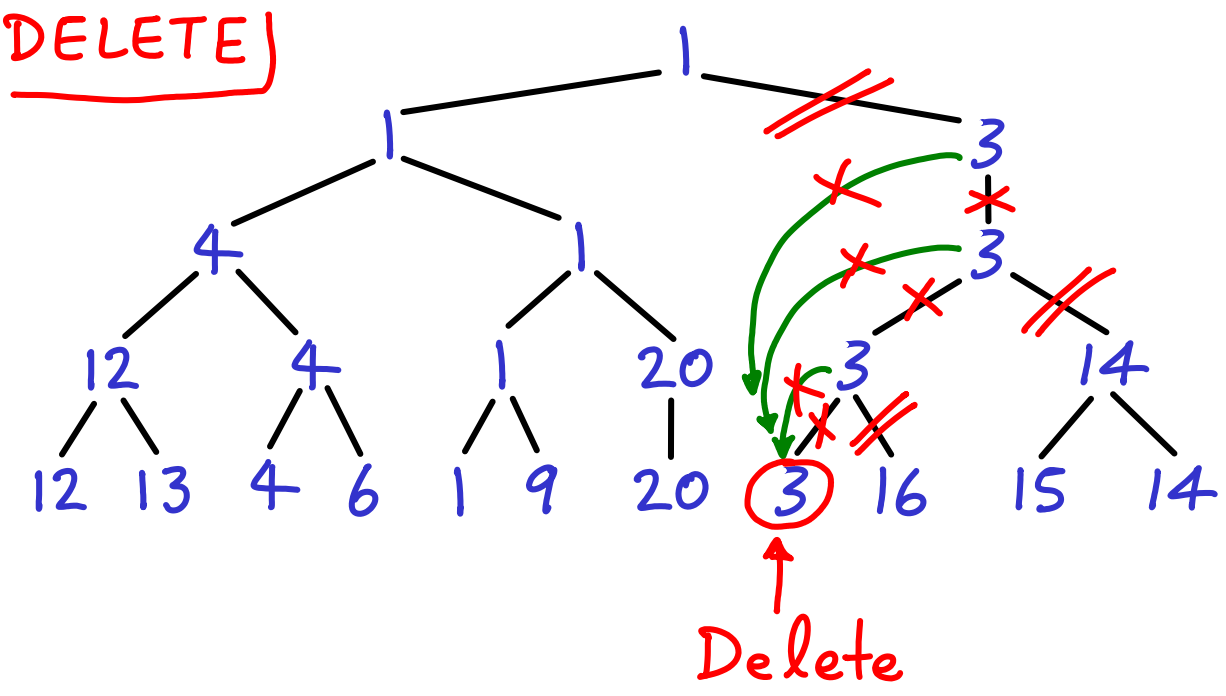
How to get DELETE O(log n) amortized

assuming no UNION

DELETE

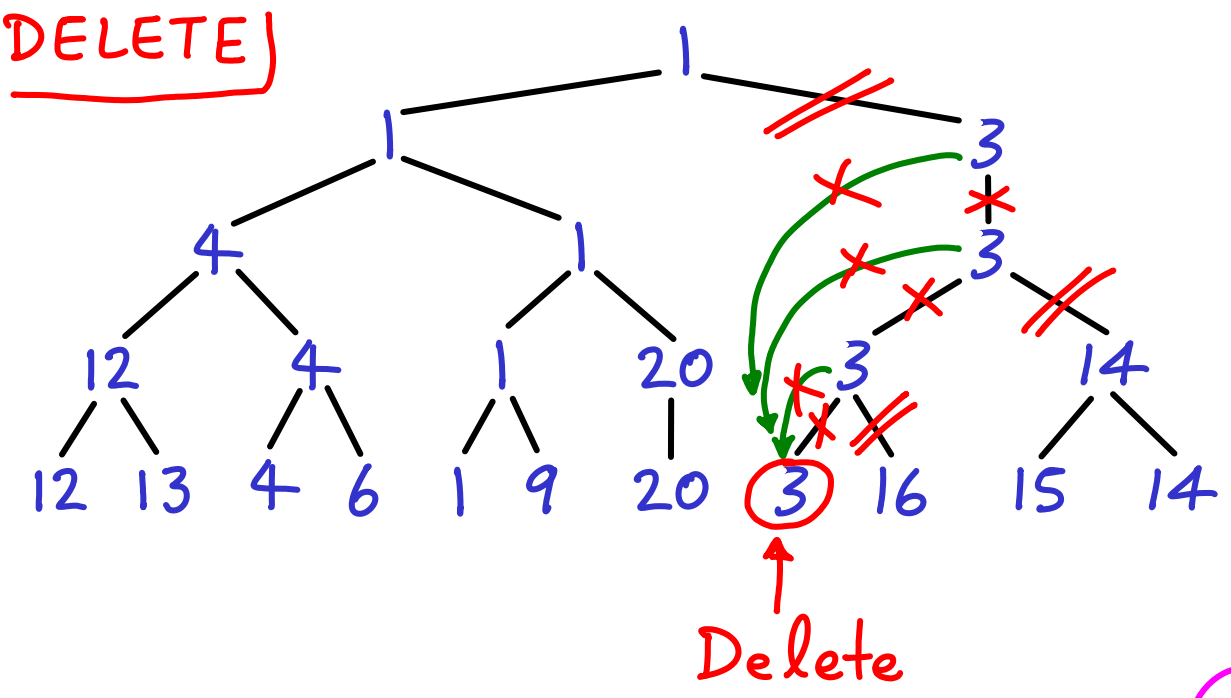


DELETE



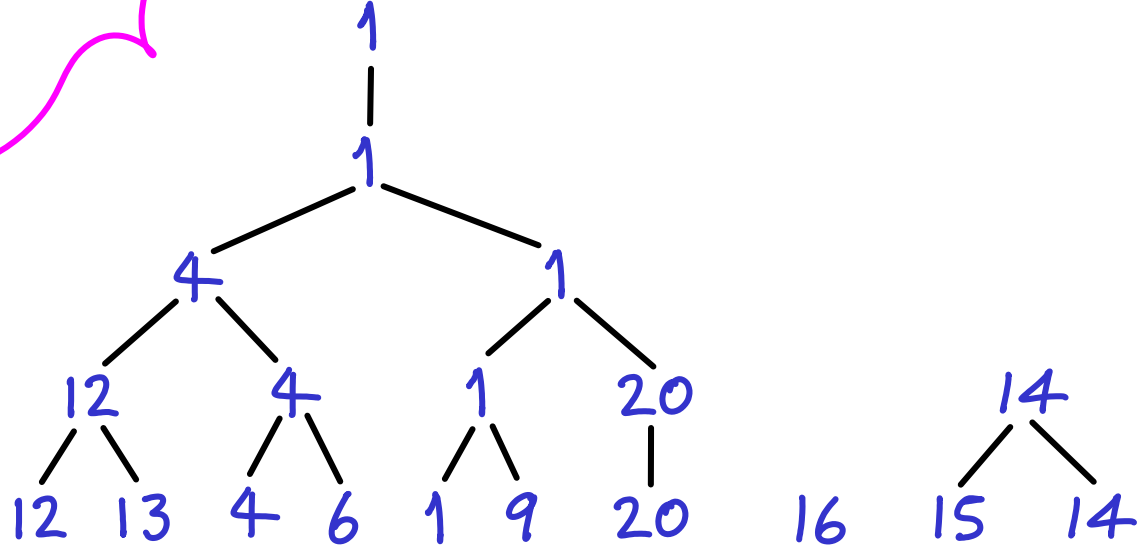
Walk up from leaf,
delete clones & CUT

DELETE

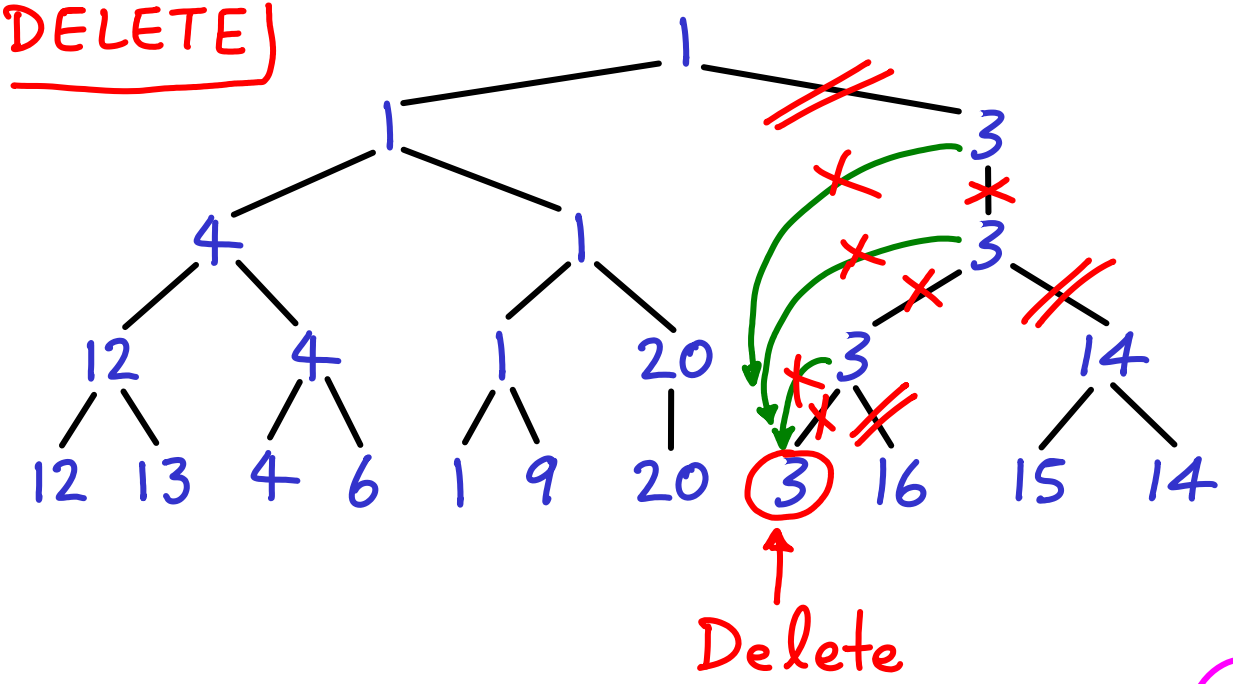


Walk up from leaf, delete clones & CUT

Also decrement n_i

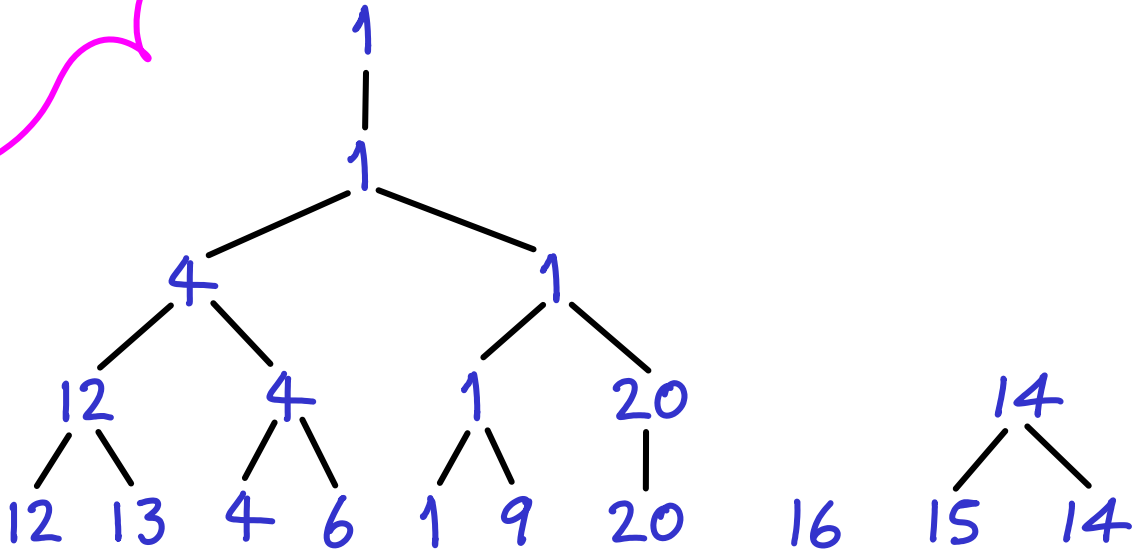


DELETE(



Walk up from leaf,
delete clones & CUT

Also decrement n_i

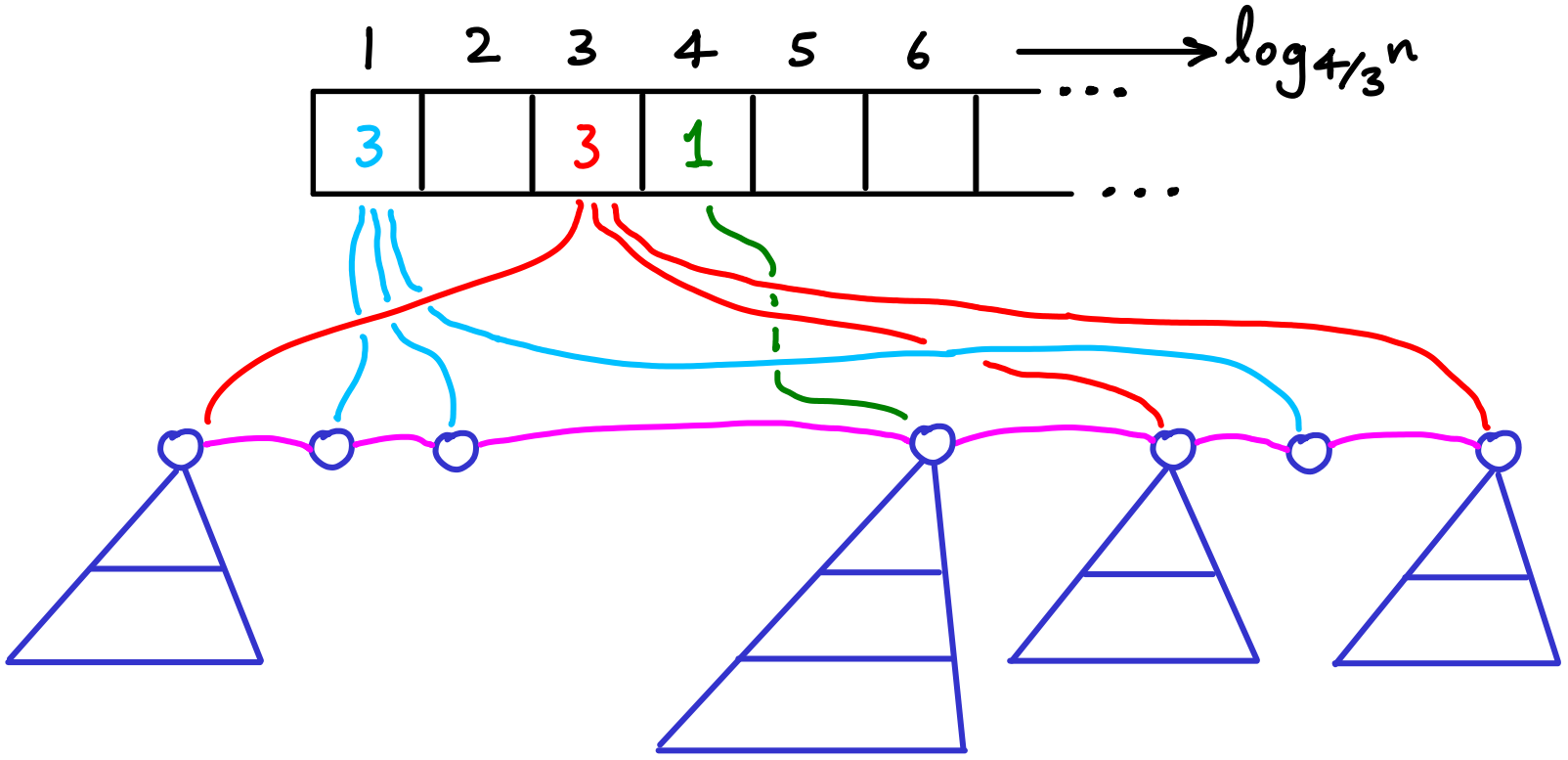


because of α:

- height = O(log n)
- time = O(log n)
- # new trees = O(log n)

Make sure ≤1 tournament tree of each height.

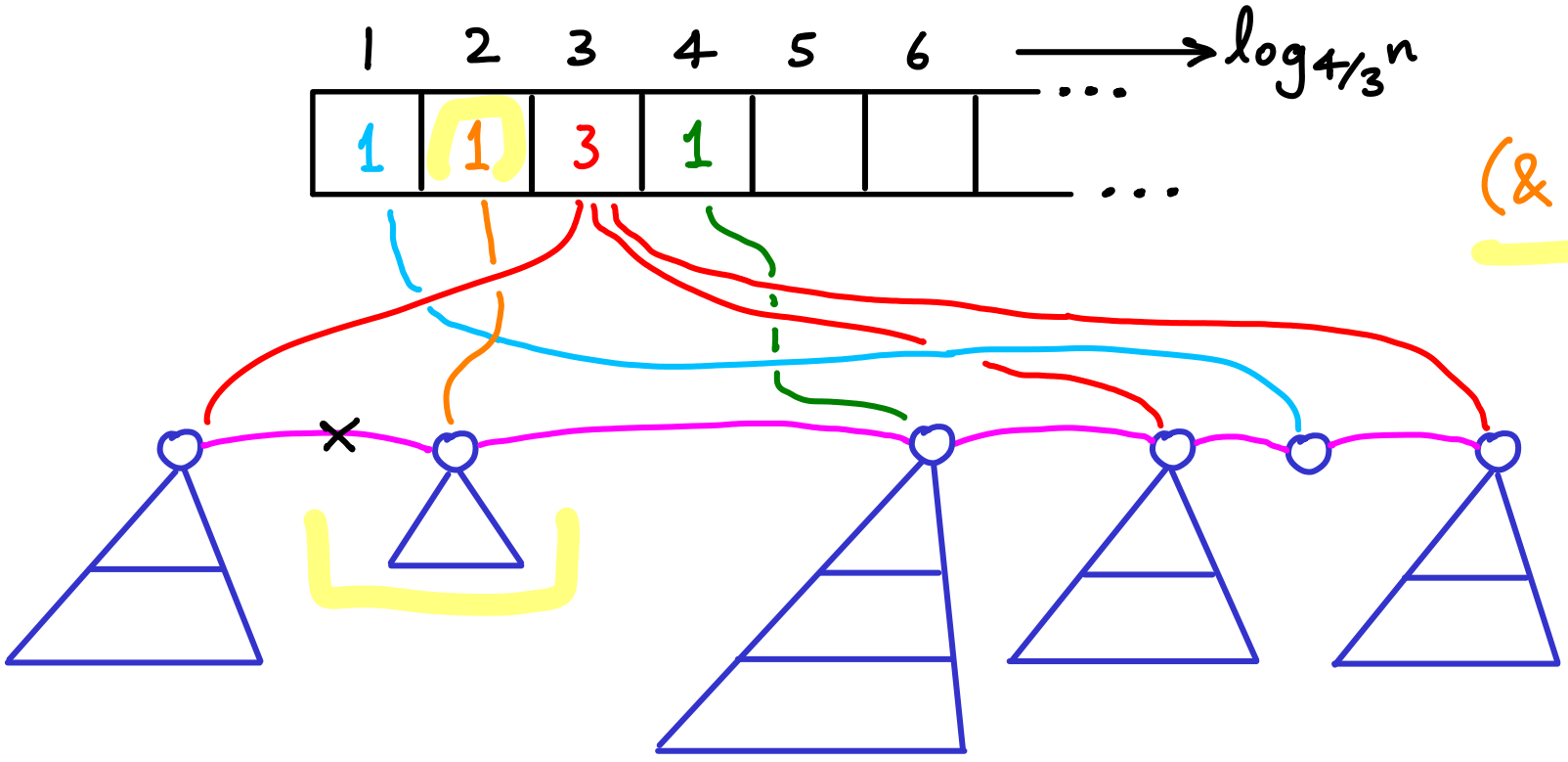
↳ LINK trees if equal height Every tree knows its height; easy to maintain



Make sure ≤1 tournament tree of each height.

↳ LINK trees if equal height

Every tree knows its height; easy to maintain

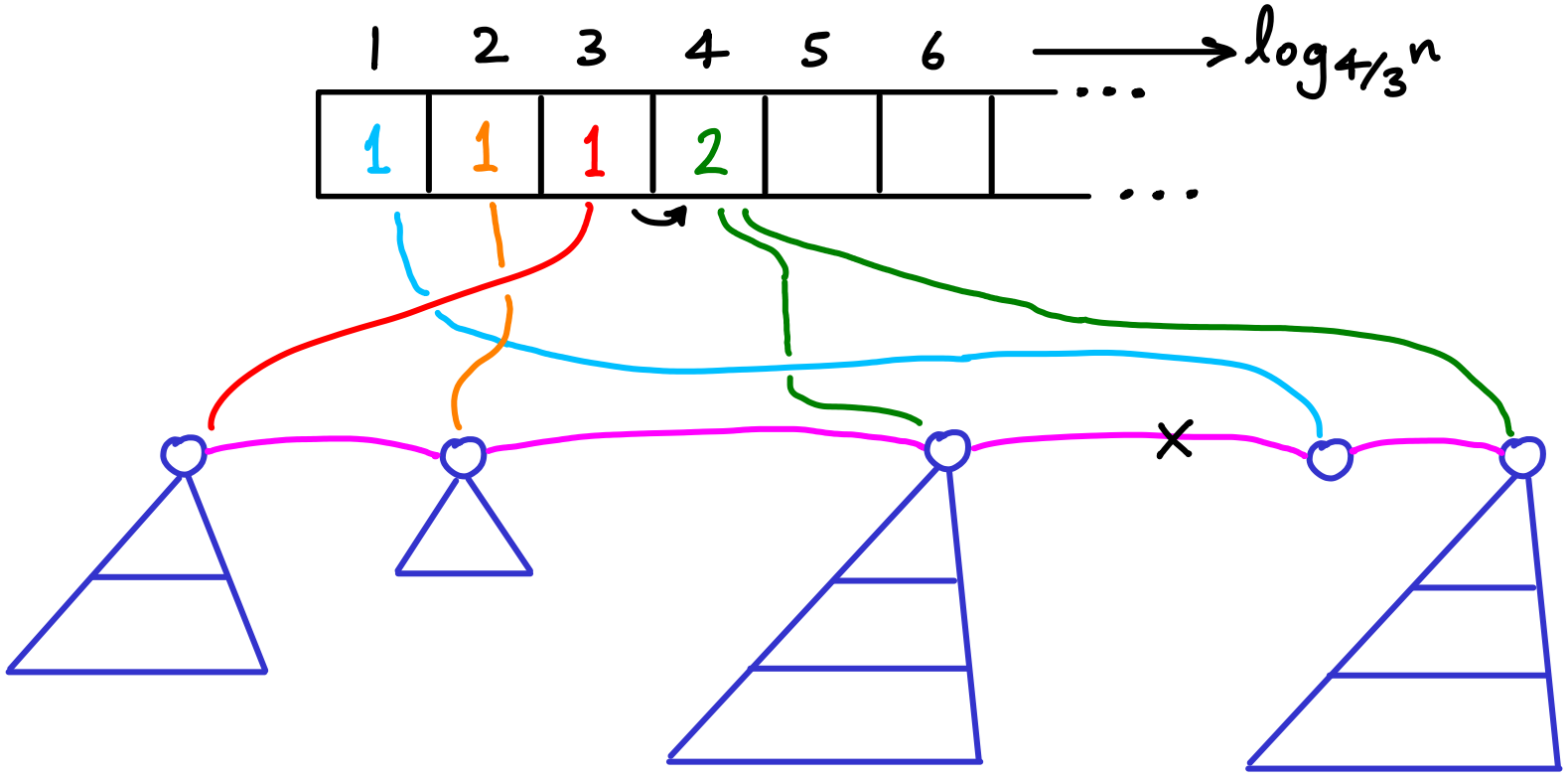


(& increment n_i)

Make sure ≤1 tournament tree of each height.

↳ LINK trees if equal height

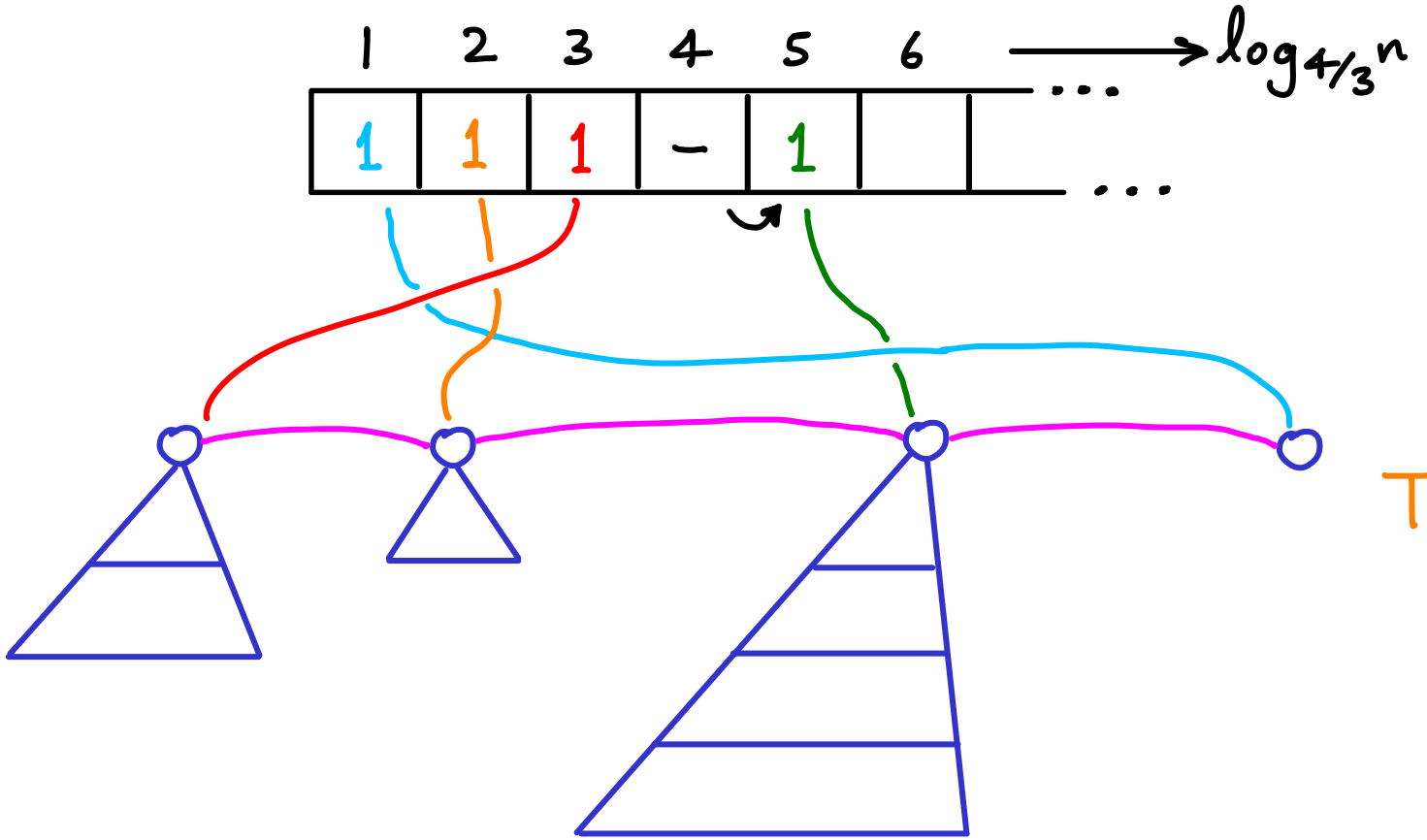
Every tree knows its height;
easy to maintain



Make sure ≤1 tournament tree of each height.

↳ LINK trees if equal height

Every tree knows its height;
easy to maintain

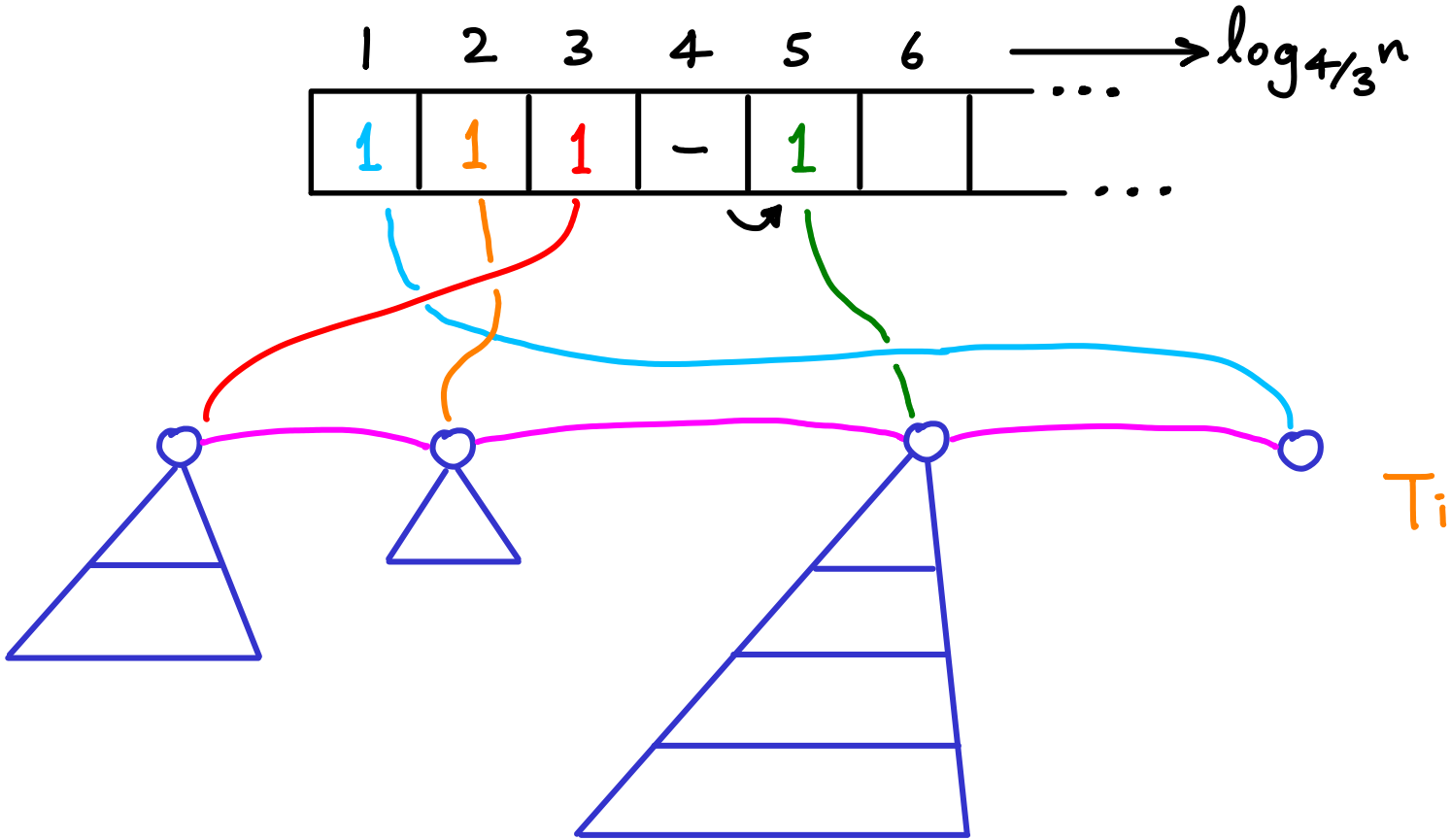


Time: ?

Make sure ≤1 tournament tree of each height.

↳ LINK trees if equal height

Every tree knows its height;
easy to maintain



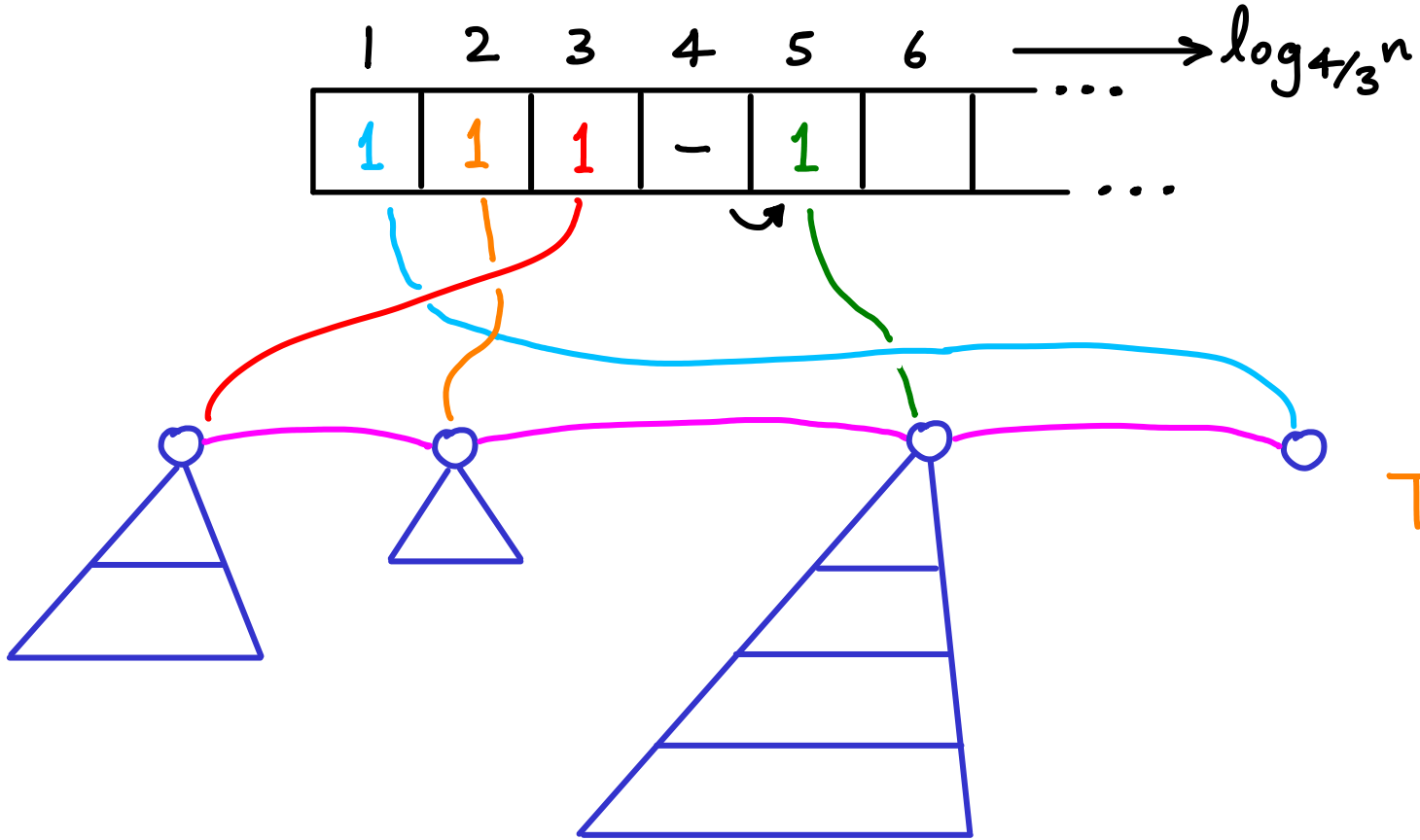
build

Time: $O(\log n) + \ldots$

Make sure ≤1 tournament tree of each height.

↳ LINK trees if equal height

Every tree knows its height;
easy to maintain

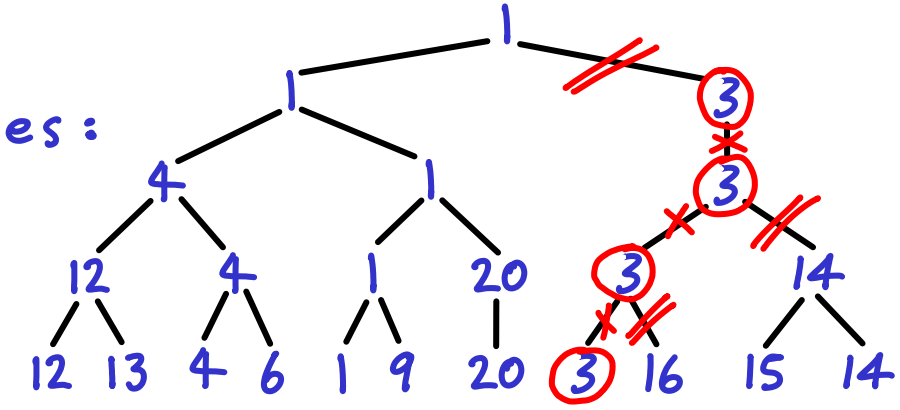


Time: $O(\log n)$ +

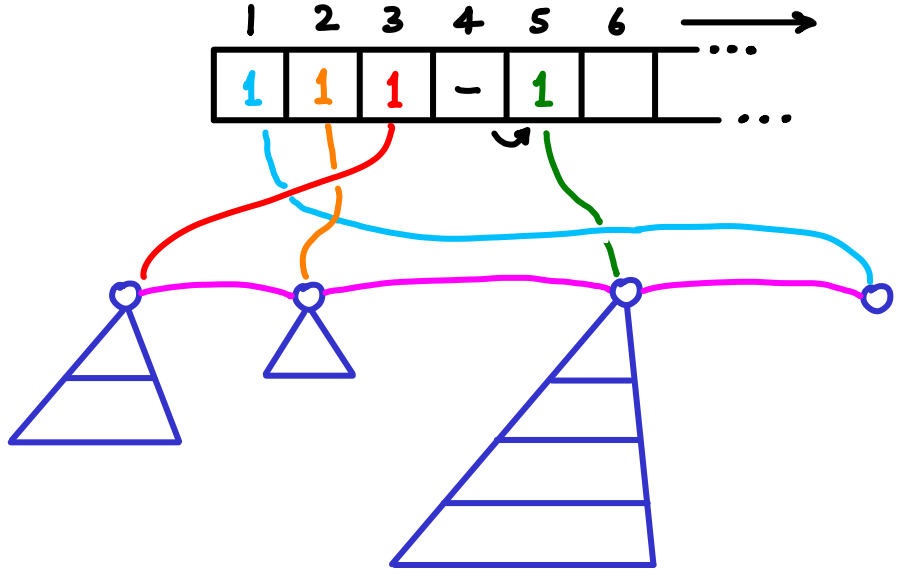
+ $O(\#\text{trees}) = O(n)$

(each step reduces #trees)

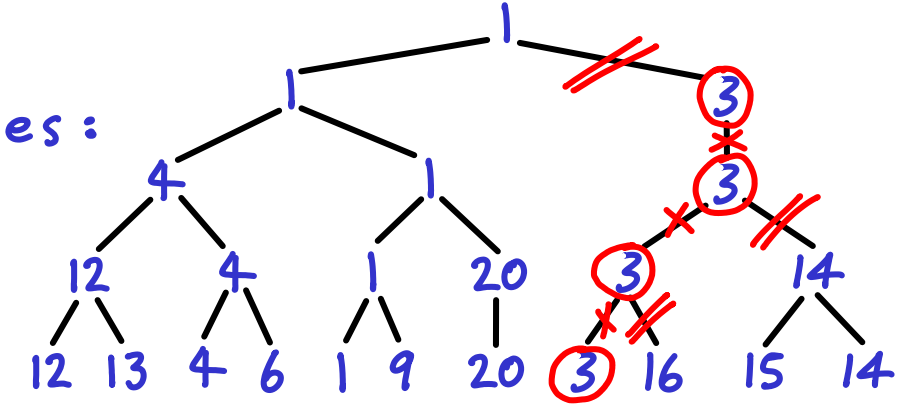
So far,
DELETE does:



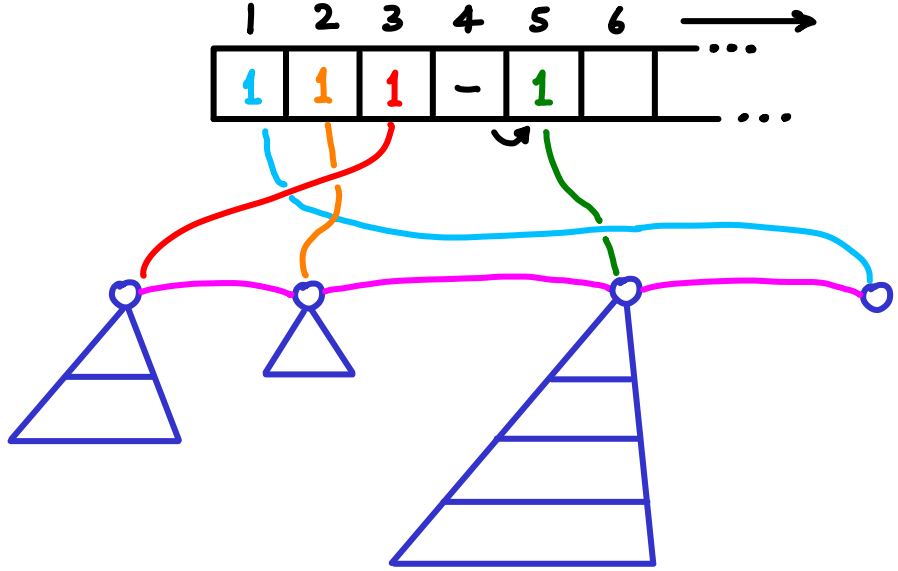
&



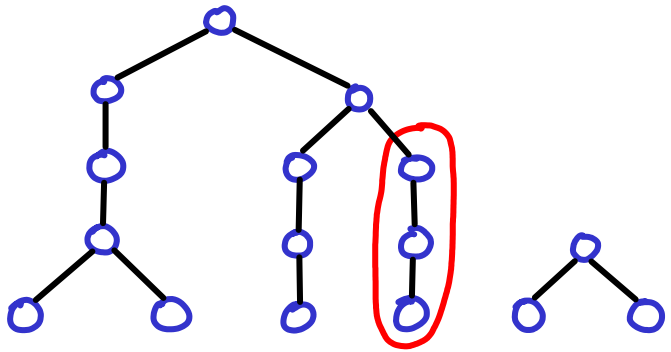
So far, DELETE does:



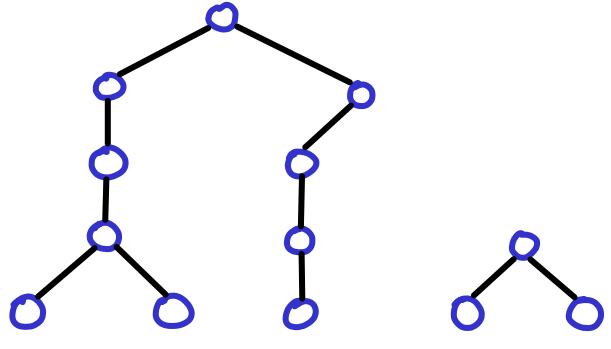
&



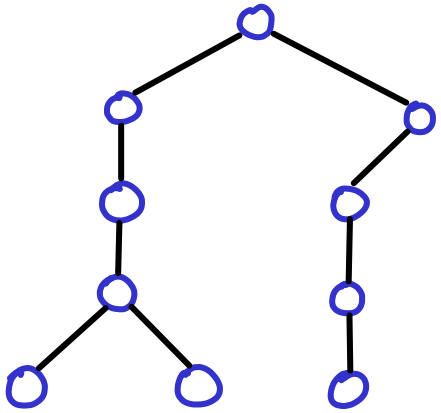
But DELETE can cause an α violation

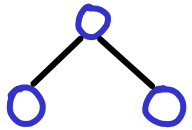


2/1 ✓
3/2 ✓
4/3 ✓
6/4 ✓



2/1 ✓
2/2 ✗
3/2 ✓
5/3 ✓

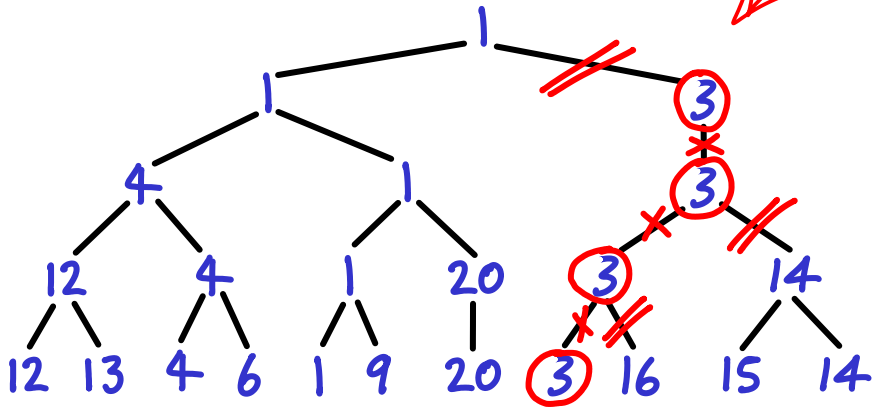




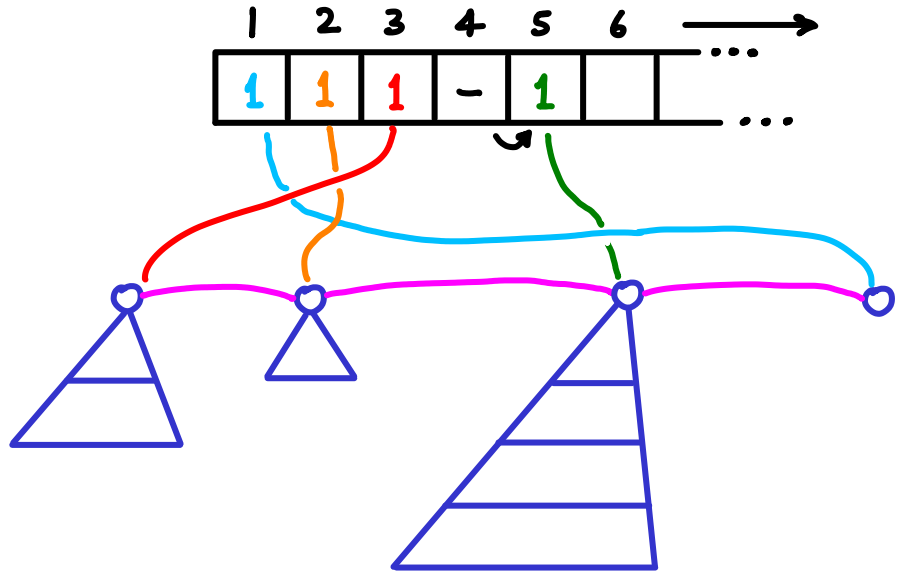
2/1 ✓
2/2 ✗
3/2 ✓
5/3 ✓

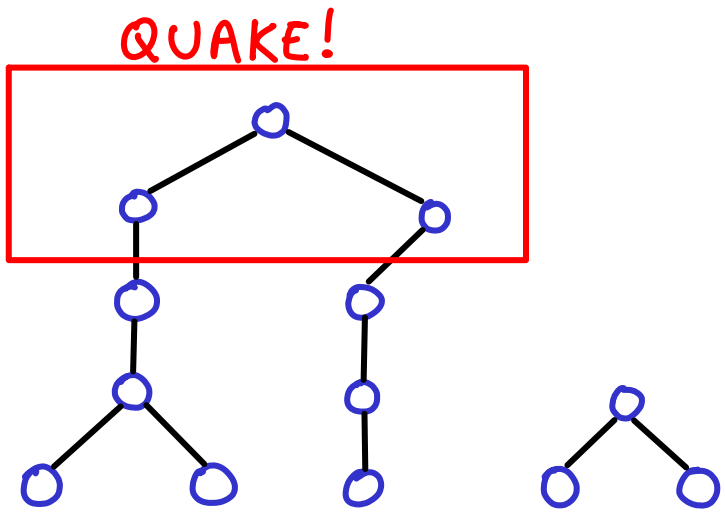
Detect lowest α-violation while walking up during deletion.

& when incrementing α_i



&



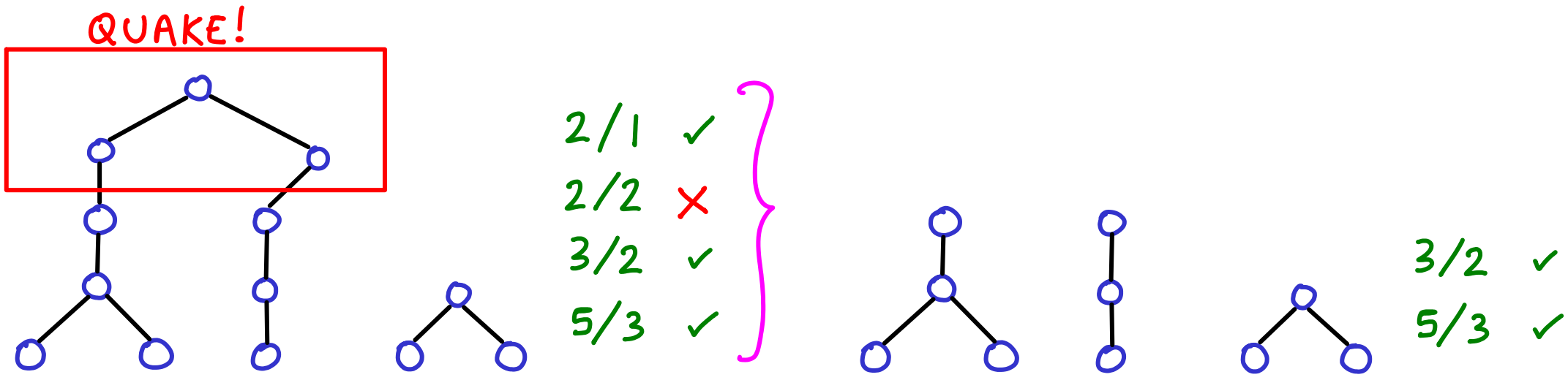


2/1 ✓

2/2 ✗

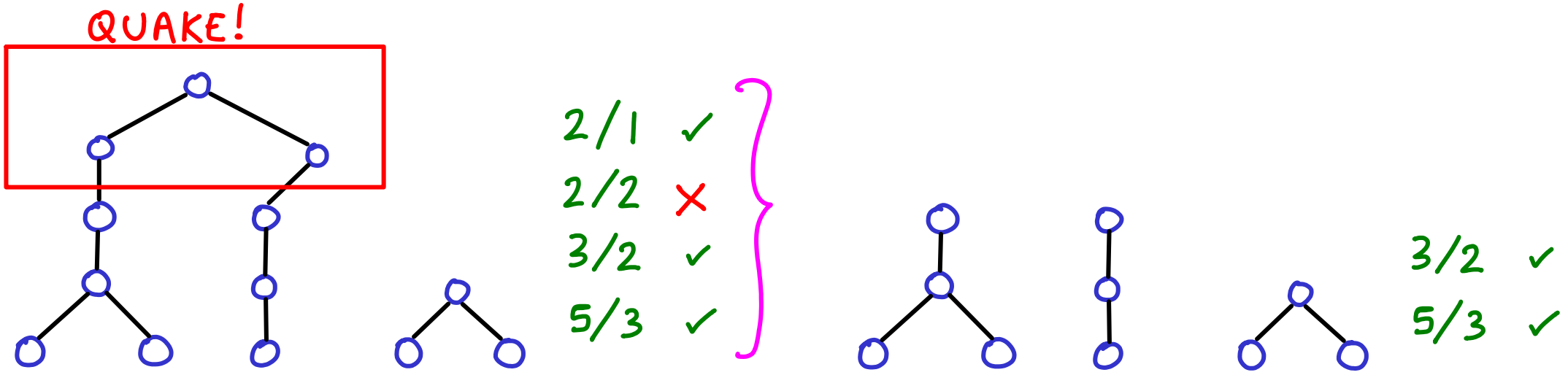
3/2 ✓

5/3 ✓



QUAKE = Remove all nodes above lowest α-violation

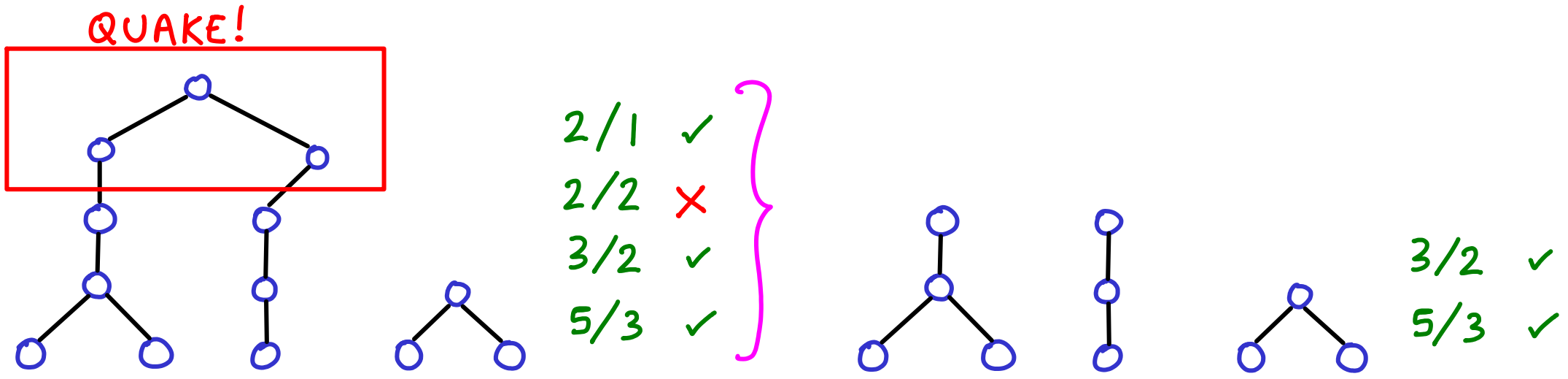
↳ in all trees



QUAKE = Remove <u>all</u> nodes above lowest α-violation (level v)

↳ in all trees

↳ Cost = $O(\sum_{i=v}^{top} n_i)$

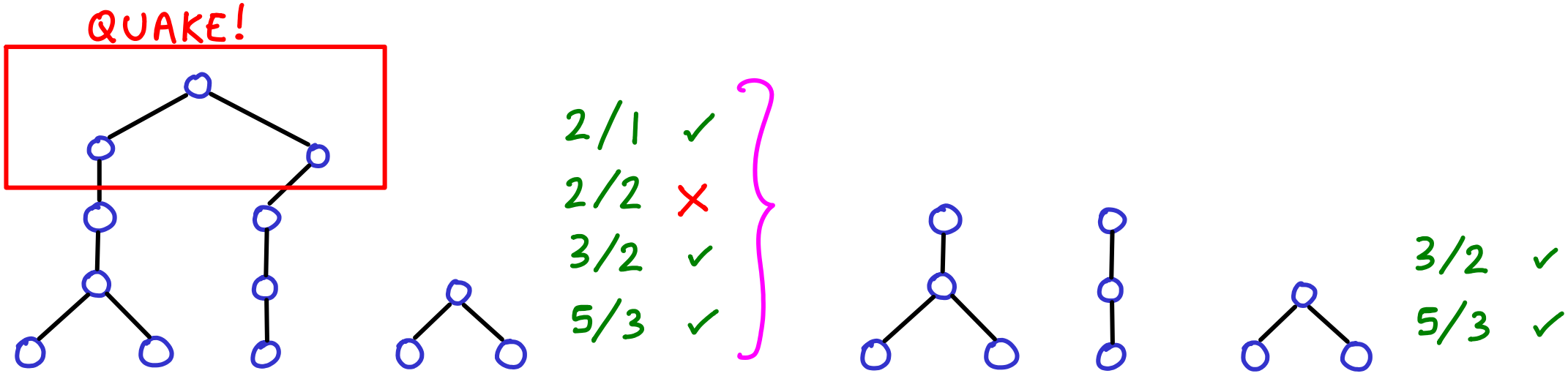


QUAKE = Remove all nodes above lowest α-violation (level v)

↳ in all trees

↳ Cost = $O(\sum_{i=v}^{top} n_i) = O(n)$

but we will amortize



QUAKE = Remove all nodes above lowest α-violation (level v)

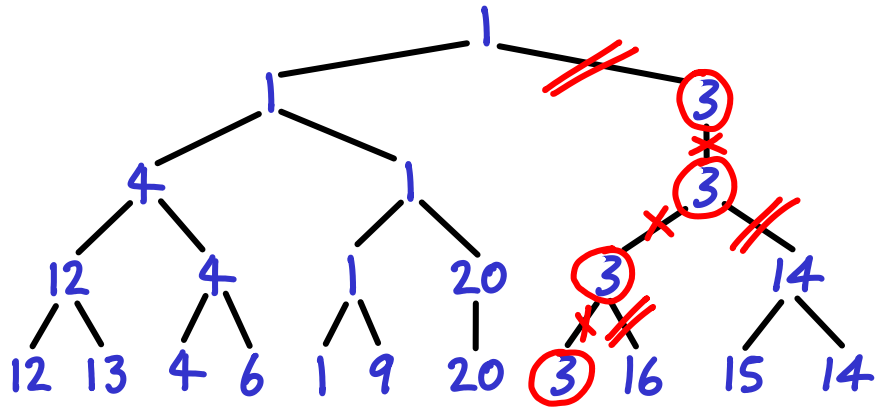
↳ in all trees

↳ Cost = $O(\sum_{i=v}^{top} n_i) = O(n)$

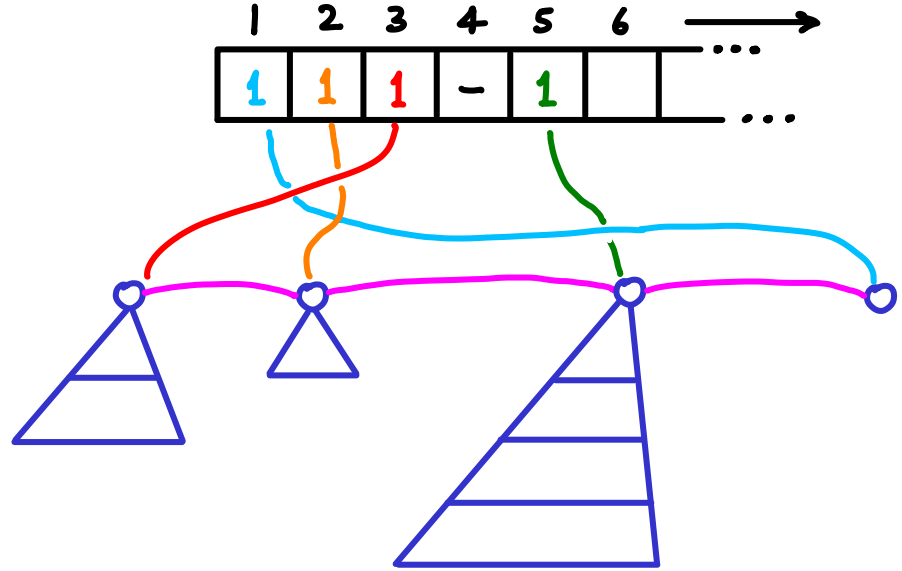
but we will amortize

↳ Now α is ok ☺ but we have more trees ∴ → bad for next DELETE

DELETE does:



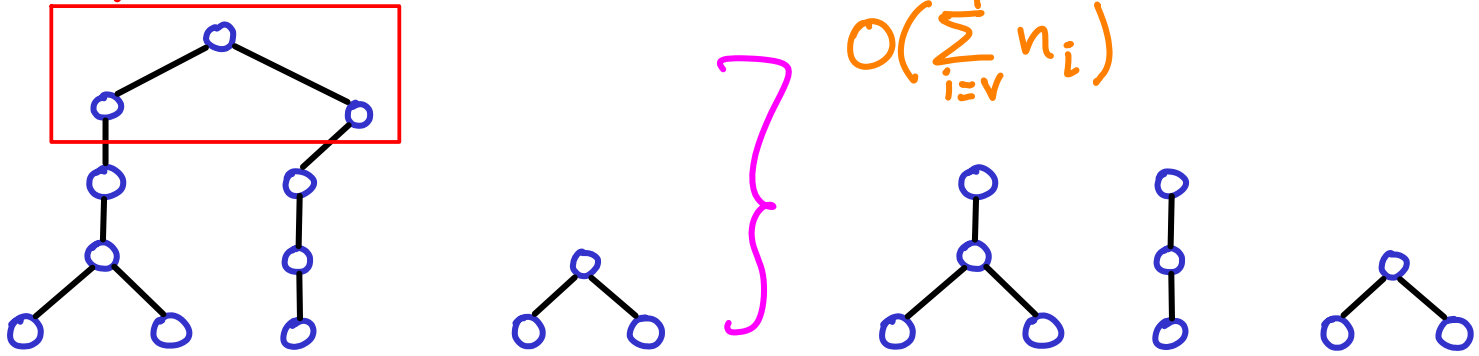
(1) O(logn) CUTS



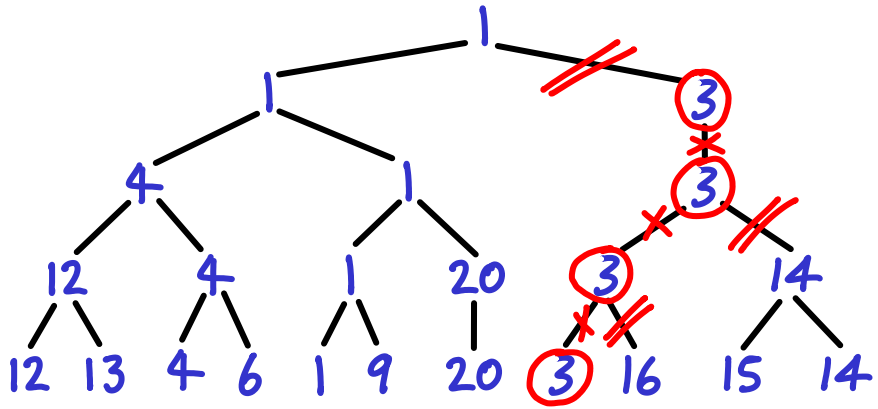
(2) O(#trees) LINKS

+ O(logn) work

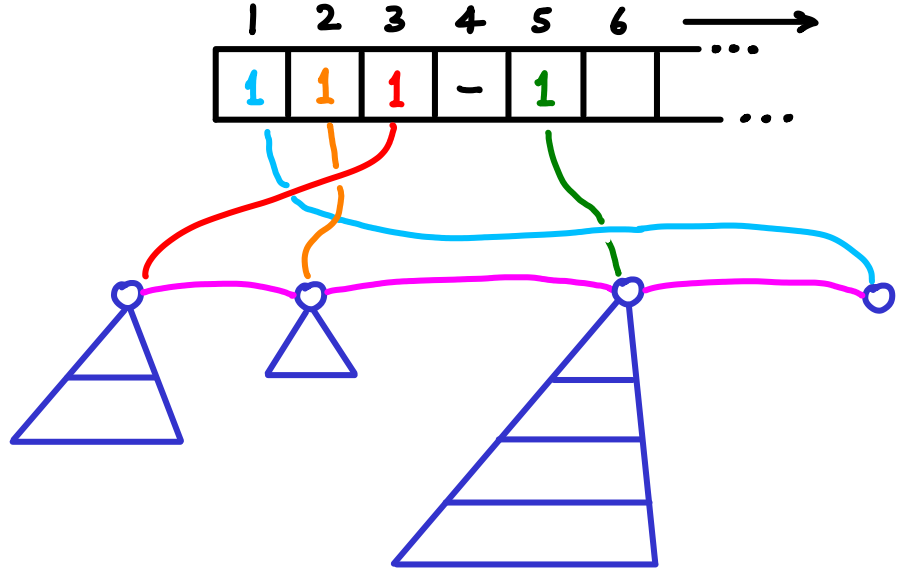
(3) QUAKE



DELETE does:

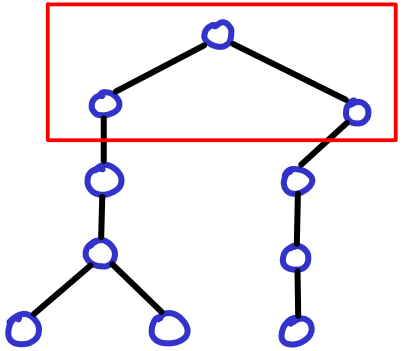


(1) $O(\log n)$ CUTS

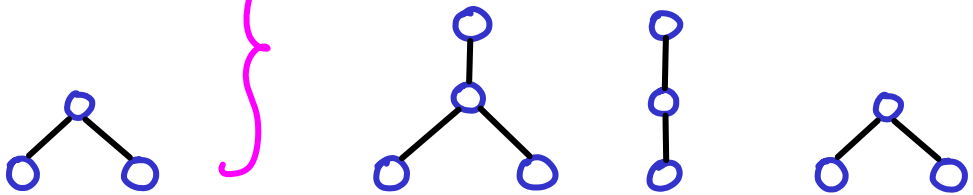


(2) $O(\#\text{trees})$ LINKS

(3) QUAKE



$O(\sum_{i=v}^{top} n_i)$

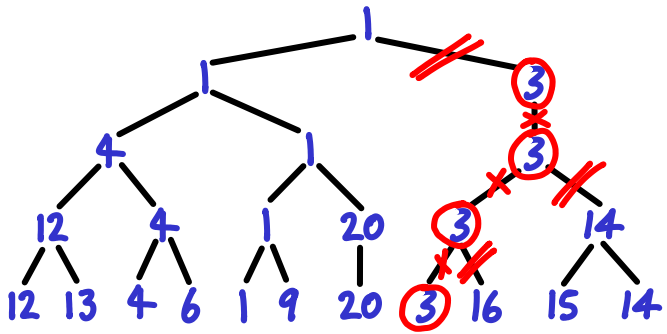


AMORTIZE

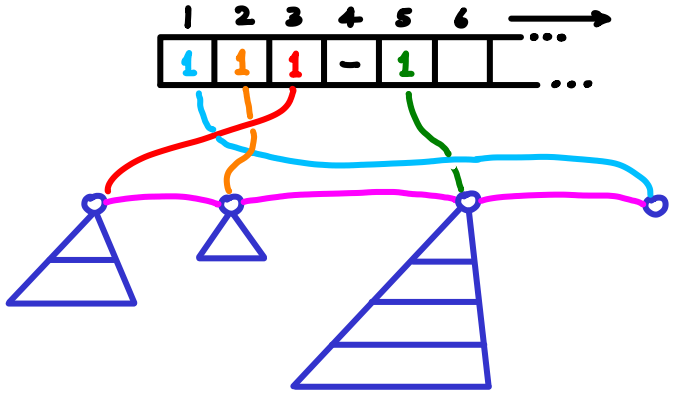
need to deal with:

↳ #trees

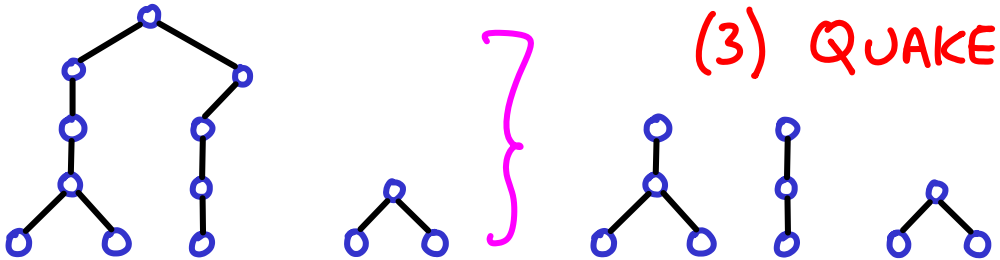
↳ Σn_i



(1) CUT



(2) LINK

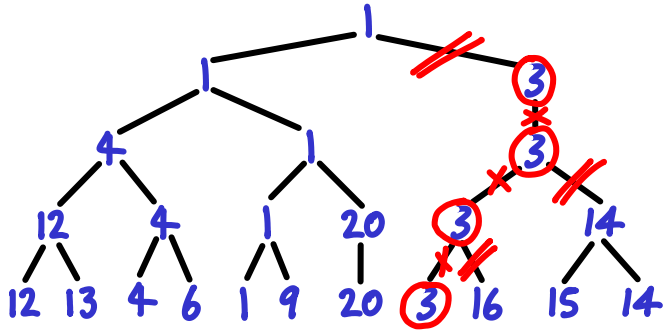


(3) QUAKE

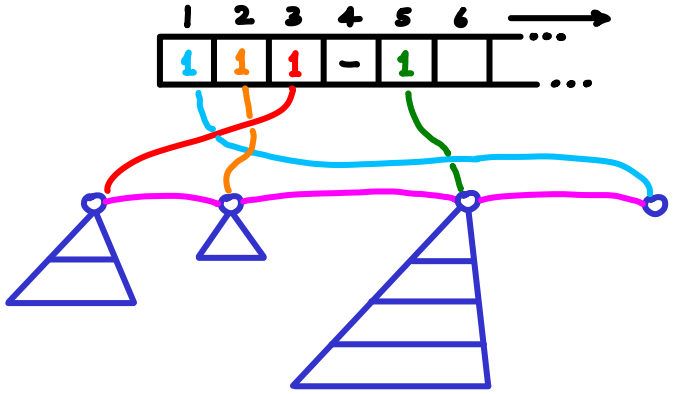
AMORTIZE $\Phi = N$?

N = #nodes in Quake heap $\neq n$ (#keys)

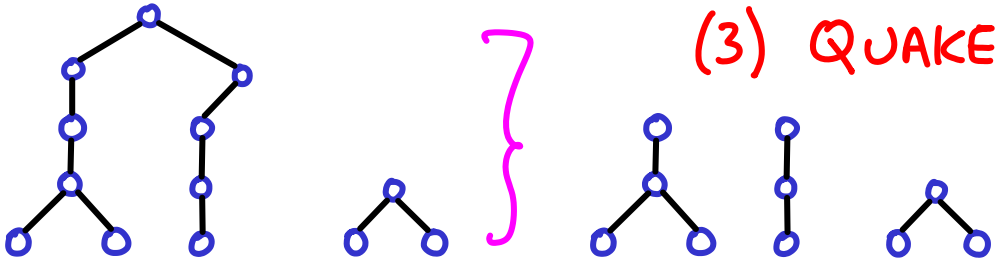
try to offset this



(1) CUT



(2) LINK



(3) QUAKE

AMORTIZE

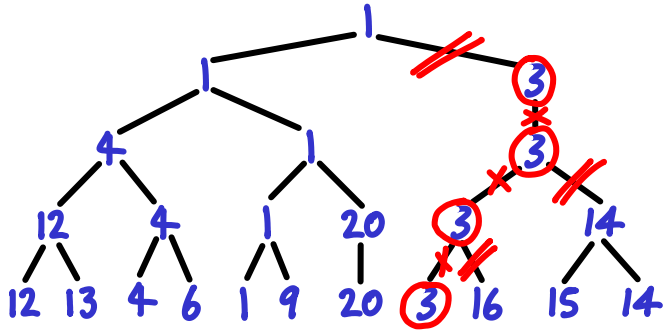
$\Phi = N$?

N = #nodes in Quake heap $\neq n$ (#keys)

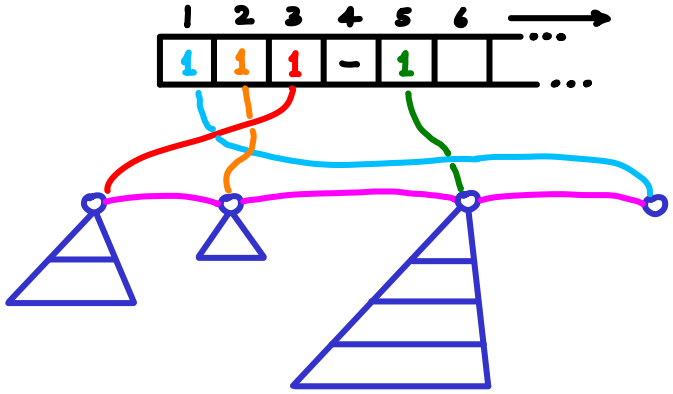
$$\Delta\Phi = -\sum_{i=v}^{top} n_i$$

$$\hat{c} = c + \Delta\Phi = 0$$

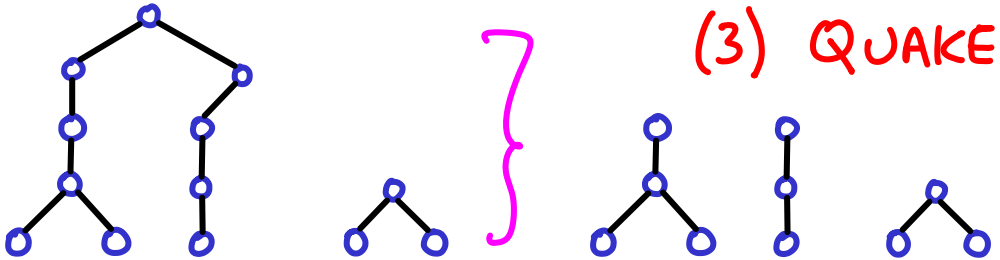
✓



(1) CUT



(2) LINK



(3) QUAKE

AMORTIZE $\Phi = N$?

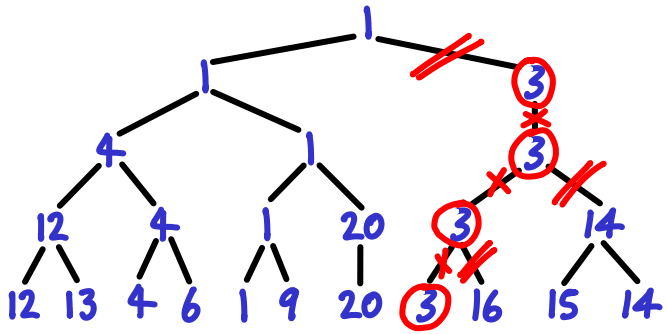
N = #nodes in Quake heap $\neq n$ (#keys)

$\Delta\Phi = ?$

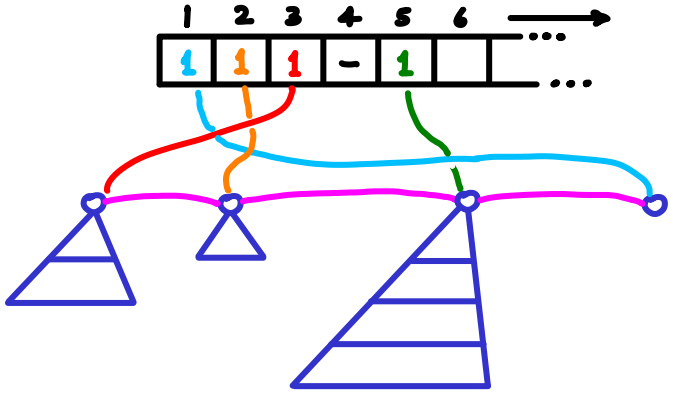
$c = O(\#trees) + O(\log n)$

$\Delta\Phi = -\sum_{i=v}^{top} n_i$

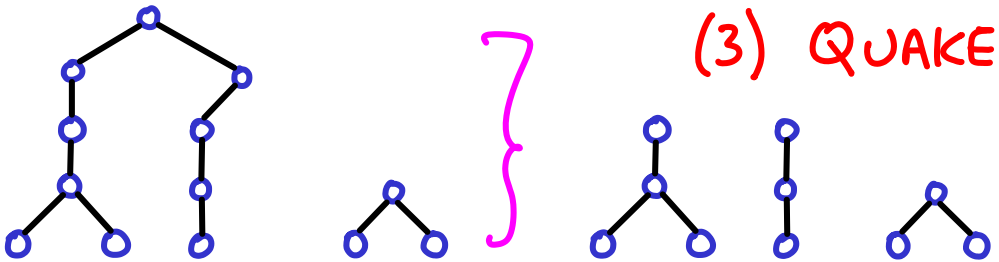
$\hat{c} = c + \Delta\Phi = 0$ ✓



(1) CUT



(2) LINK



(3) QUAKE

AMORTIZE

$\Phi = N$?

N = #nodes in Quake heap $\neq n$ (#keys)

$\Delta\Phi \geq 0$ ☹

$c = O(\#trees) + O(\log n)$

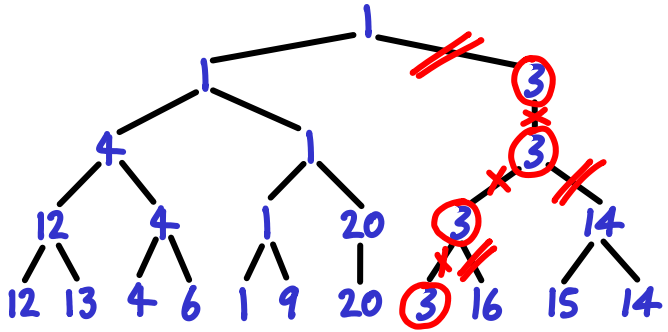
Actually,

$\Delta\Phi = \Delta N \leq \#trees$

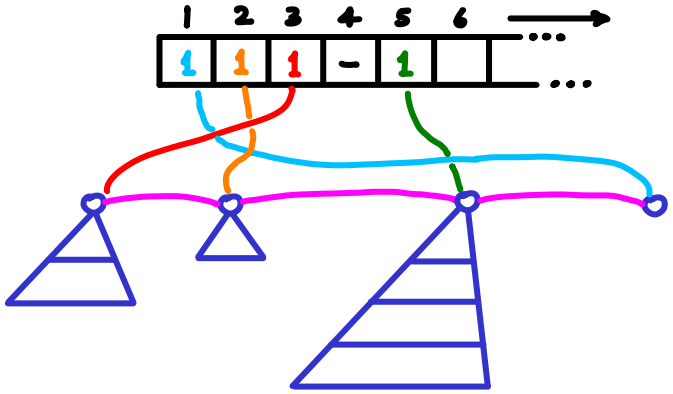
Each LINK creates one node

$$\Delta\Phi = -\sum_{i=v}^{top} n_i$$

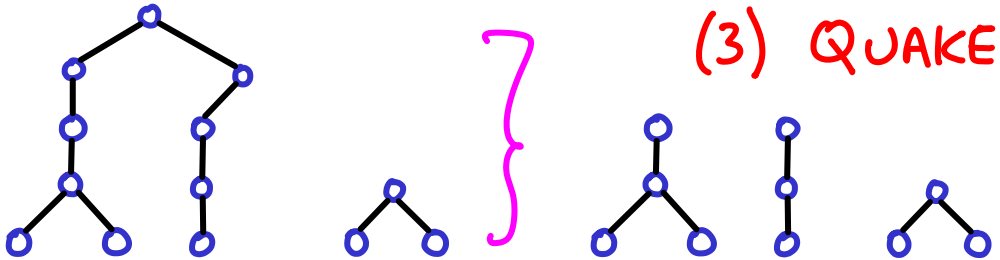
$\hat{c} = c + \Delta\Phi = 0$ ✓



(1) CUT



(2) LINK

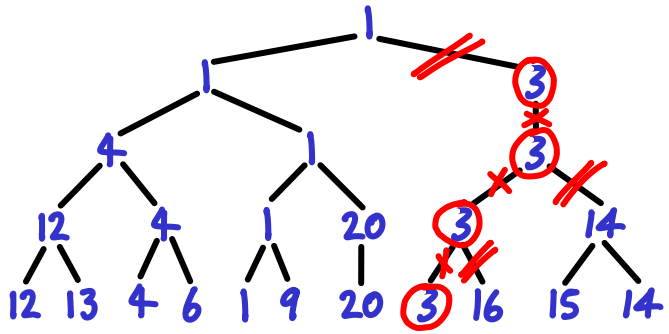


(3) QUAKE

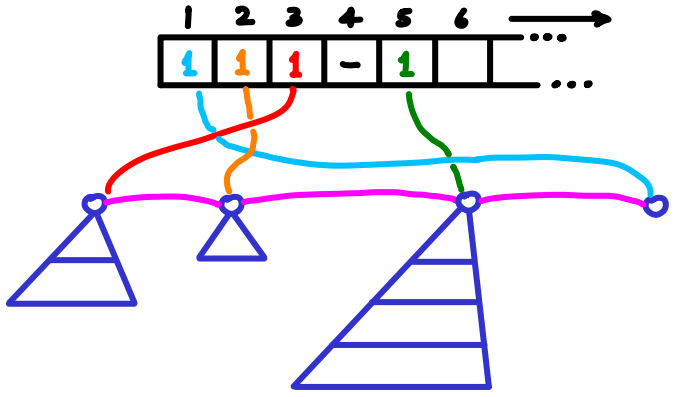
AMORTIZE $\Phi = N + T$?

N = # nodes in Quake heap $\neq n$ (# keys)

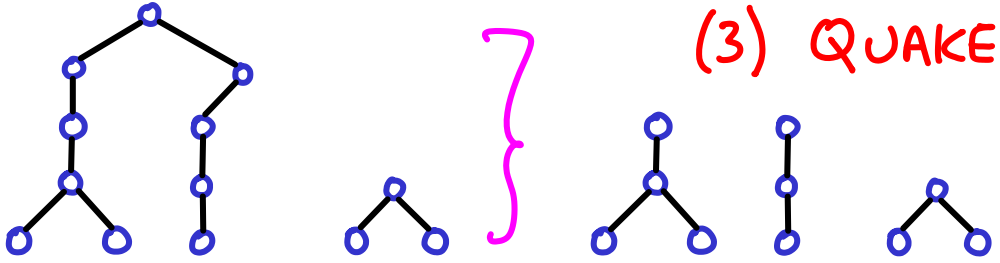
T = # trees



(1) CUT



(2) LINK



(3) QUAKE

AMORTIZE

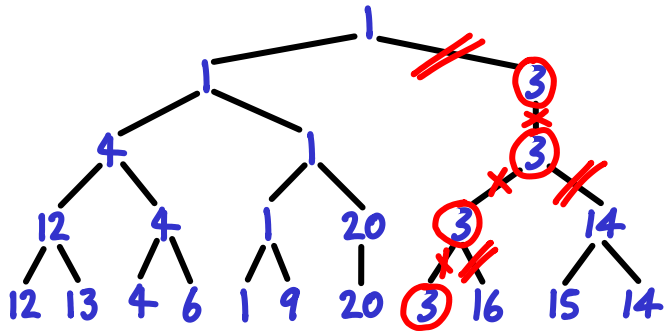
$\Phi = N + T$?

N = # nodes in Quake heap $\neq n$ (# keys)

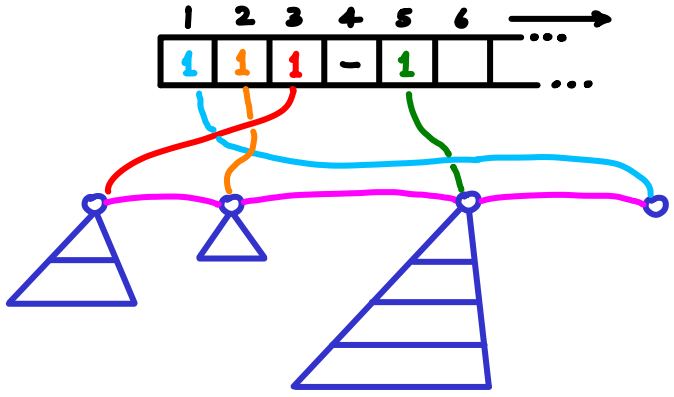
T = # trees

$c_j = O(T_{j-1}) + O(\log n)$

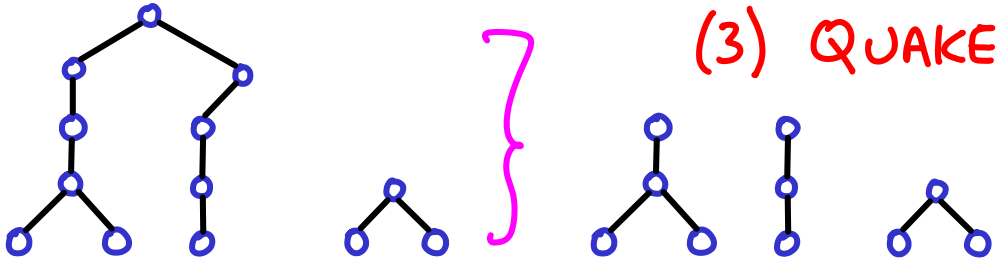
j = iteration # (operation)



(1) CUT



(2) LINK



(3) QUAKE

AMORTIZE

$\Phi = N + T$?

N = # nodes in Quake heap $\neq n$ (# keys)

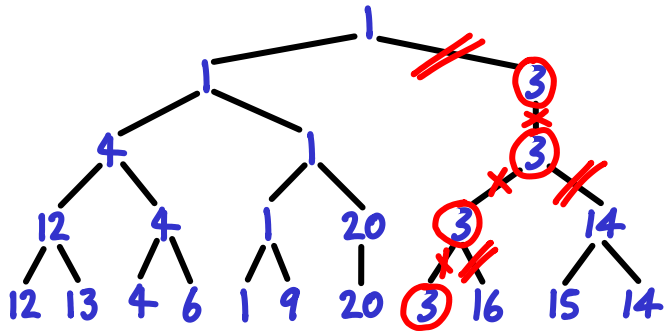
T = # trees

$$\Delta\Phi_j \leq \overbrace{T_{j-1}}^{\Delta N} + \overbrace{(\log n - T_{j-1})}^{\Delta T}$$

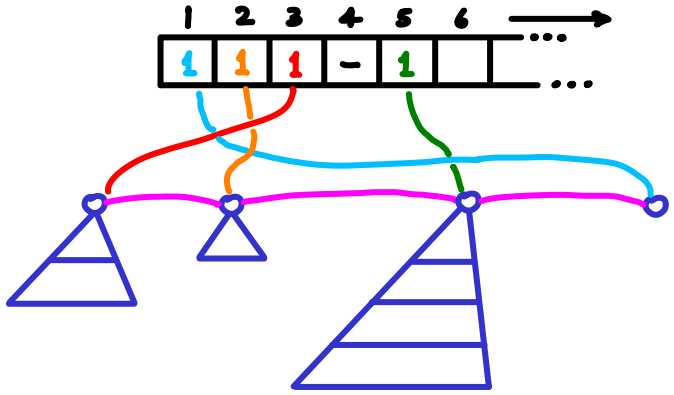
☹

$$c_j = O(T_{j-1}) + O(\log n)$$

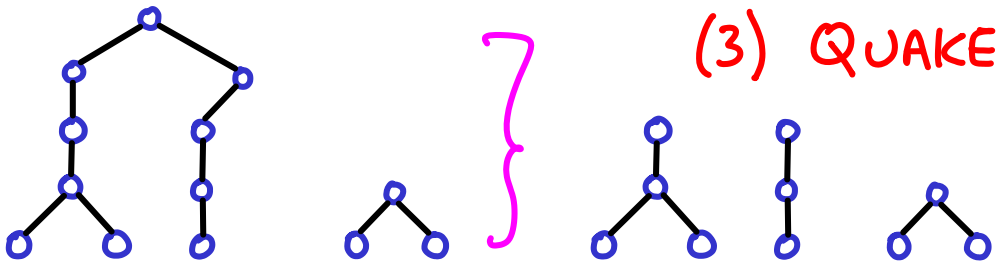
j = iteration #



(1) CUT



(2) LINK



(3) QUAKE

AMORTIZE $\Phi = N + T$?

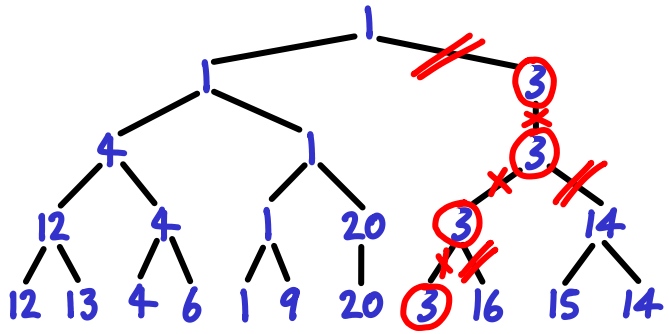
N = # nodes in Quake heap $\neq n$ (# keys)

T = # trees multiplied by some constant, c

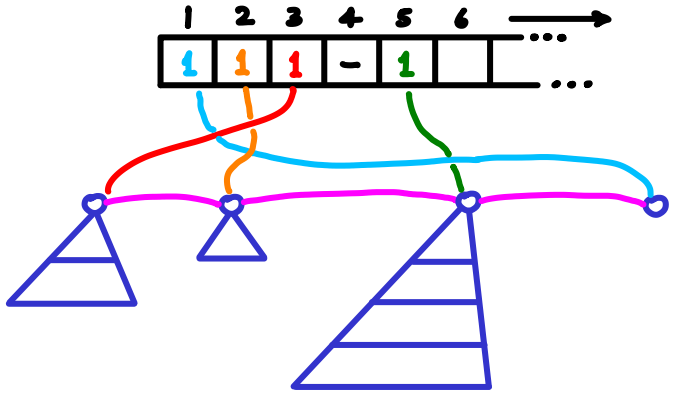
$$\Delta\Phi_j \leq \overbrace{T_{j-1}}^{\Delta N} + \overbrace{c(\log n - T_{j-1})}^{\Delta T}$$

$$c_j = O(T_{j-1}) + O(\log n)$$

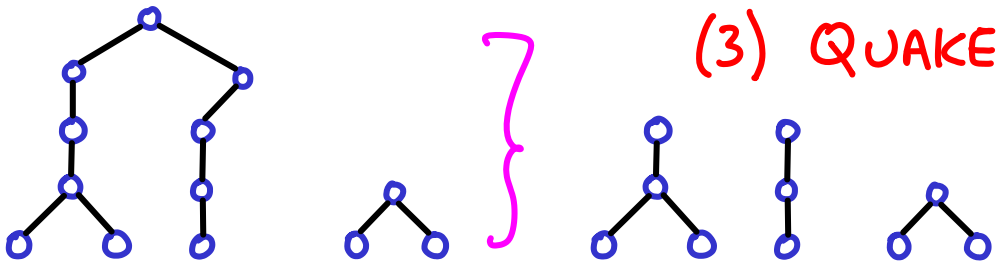
j = iteration #



(1) CUT



(2) LINK



(3) QUAKE

AMORTIZE $\Phi = N + T$?

N = # nodes in Quake heap $\neq n$ (# keys)

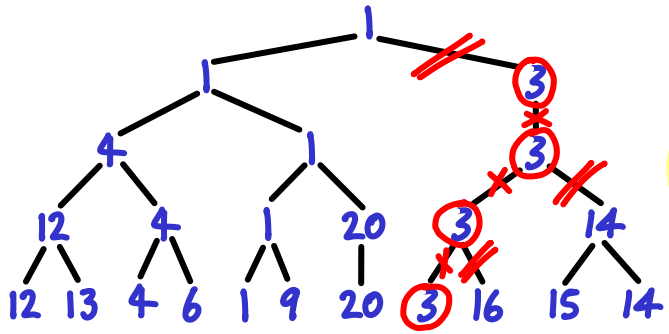
T = # trees multiplied by some constant, c

$$\Delta\Phi_j \leq \overbrace{T_{j-1}}^{\Delta N} + \overbrace{c(\log n - T_{j-1})}^{\Delta T}$$

$$c_j = O(T_{j-1}) + O(\log n)$$

$$\hat{c}_j = O(\log n) \checkmark$$

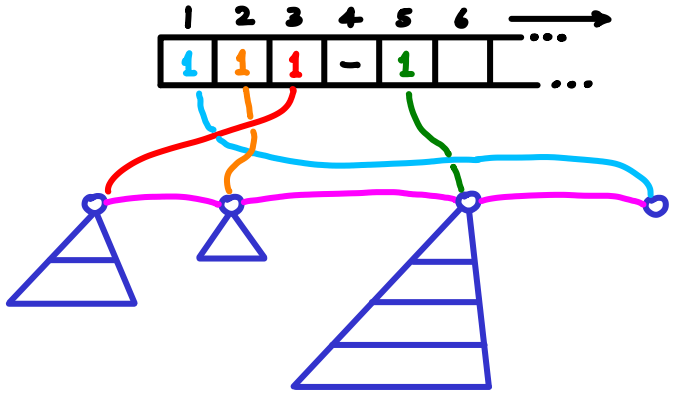
j = iteration #



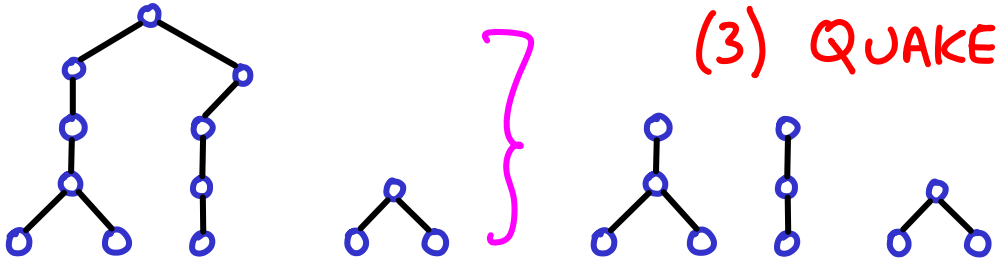
(1) CUT

$\Delta T = O(\log n)$

$\Delta N < 0$ ✓



(2) LINK



(3) QUAKE

AMORTIZE

$\Phi = N + T$?

N = # nodes in Quake heap $\neq n$ (# keys)

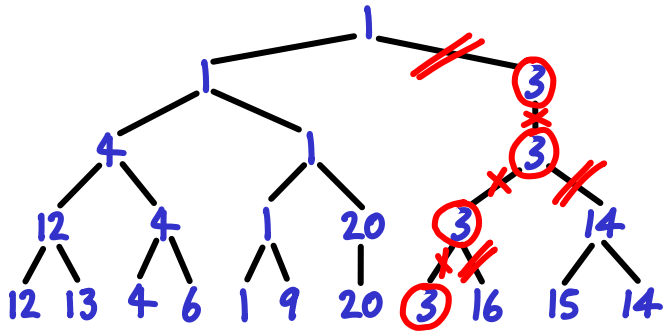
T = # trees multiplied by some constant, c

$$\Delta\Phi_j \leq \overbrace{T_{j-1}}^{\Delta N} + \overbrace{c(\log n - T_{j-1})}^{\Delta T}$$

$$c_j = O(T_{j-1}) + O(\log n)$$

$$\hat{c}_j = O(\log n)$$ ✓

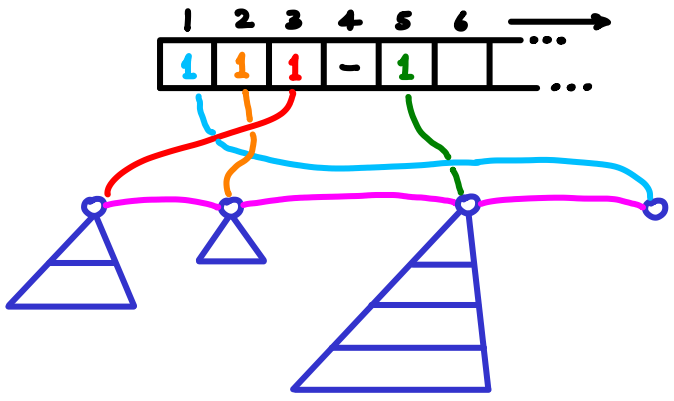
j = iteration #



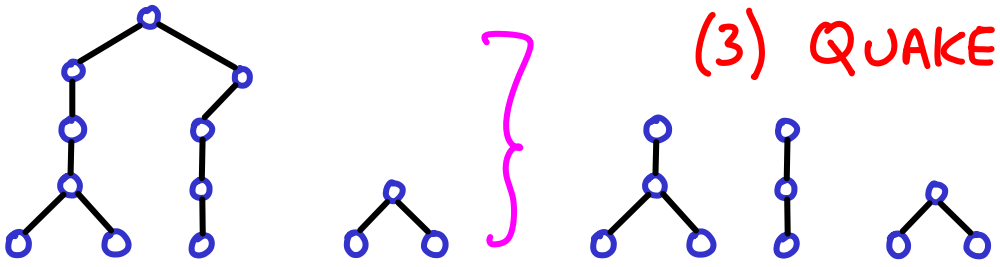
(1) CUT

$\Delta T = O(\log n)$

$\Delta N < 0$ ✓



(2) LINK



(3) QUAKE

AMORTIZE

$\Phi = N + T$?

N = # nodes in Quake heap $\neq n$ (# keys)

T = # trees multiplied by some constant, c

$$\Delta\Phi_j \leq \overbrace{T_{j-1}}^{\Delta N} + \overbrace{c(\log n - T_{j-1})}^{\Delta T}$$

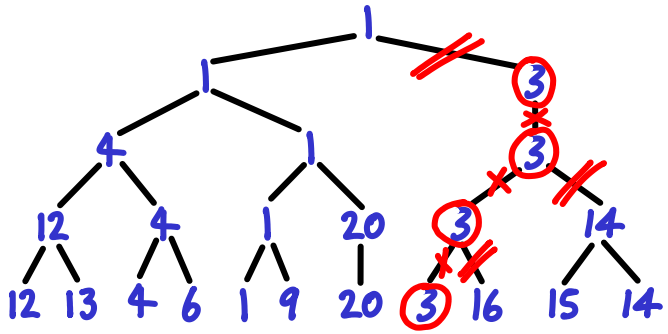
$$c_j = O(T_{j-1}) + O(\log n)$$

$$\hat{c}_j = O(\log n)$$ ✓

j = iteration #

$$\Delta\Phi = -\sum_{i=v}^{top} n_i + c \cdot \Delta T = ?$$

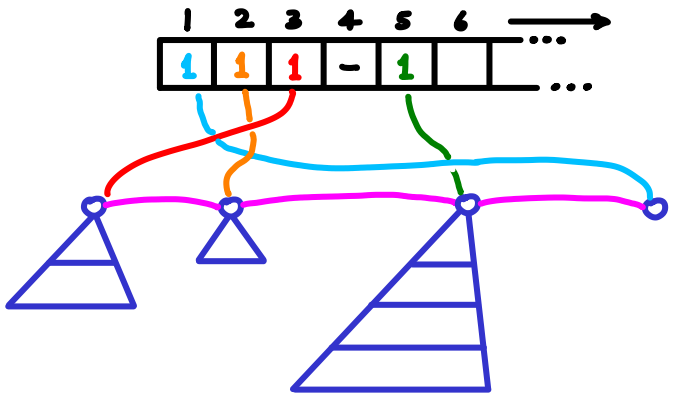
i = level #



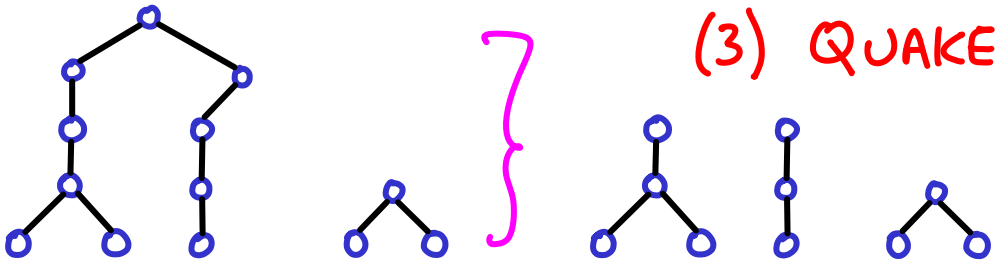
(1) CUT

$\Delta T = O(\log n)$

$\Delta N < 0$ ✓



(2) LINK



(3) QUAKE

AMORTIZE

$\Phi = N + T$?

N = # nodes in Quake heap $\neq n$ (# keys)

T = # trees multiplied by some constant, c

$$\Delta\Phi_j \leq \overbrace{T_{j-1}}^{\Delta N} + \overbrace{c(\log n - T_{j-1})}^{\Delta T}$$

$$c_j = O(T_{j-1}) + O(\log n)$$

$$\hat{c}_j = O(\log n)$$ ✓

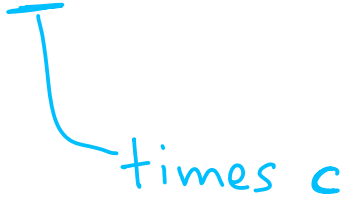
j = iteration #

$$\Delta\Phi = -\sum_{i=v}^{top} n_i + c \cdot \Delta T = ?$$

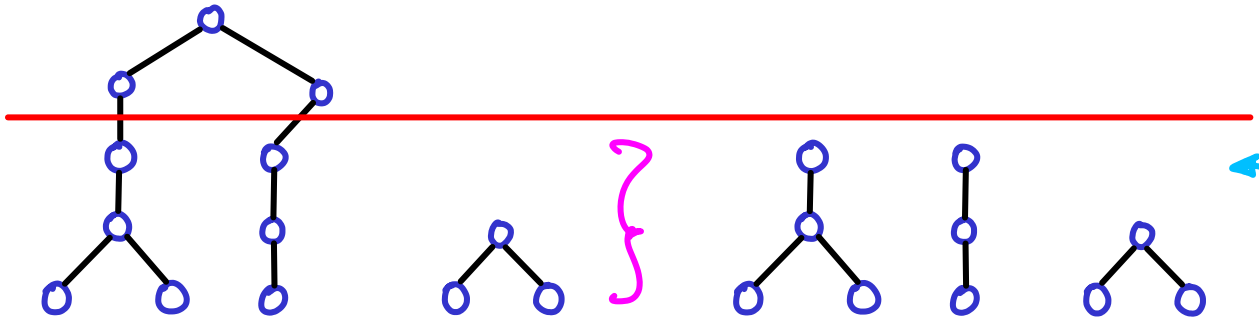
i = level #

$$\hat{c} = c + \Delta\Phi = \cancel{\sum_{i=v}^{top} n_i} - \cancel{\sum_{i=v}^{top} n_i} + c \cdot \Delta T = \text{☹}$$

QUAKE : $\Phi = N + T$ doesn't work: We're stuck with ΔT



QUAKE : $\Phi = N + T$ doesn't work: We're stuck with ΔT

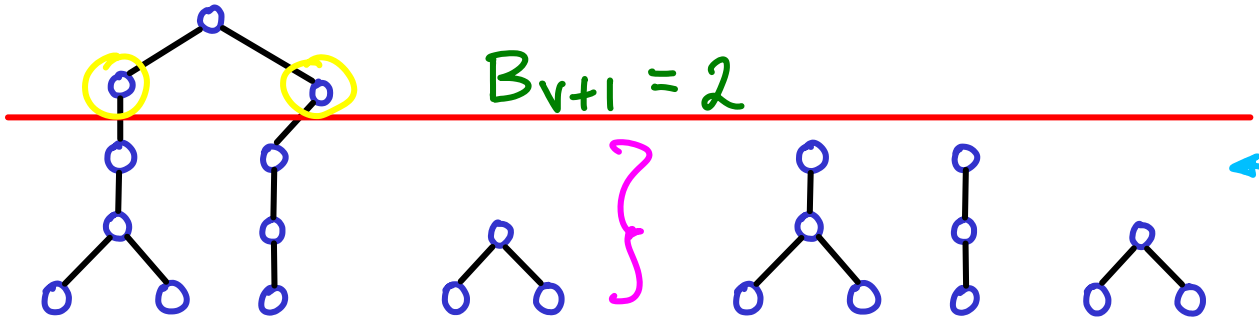


$\Delta T \leq n_v$
= # nodes at quake level

$n_v = n_3 = 2$

$\Delta T = 1$

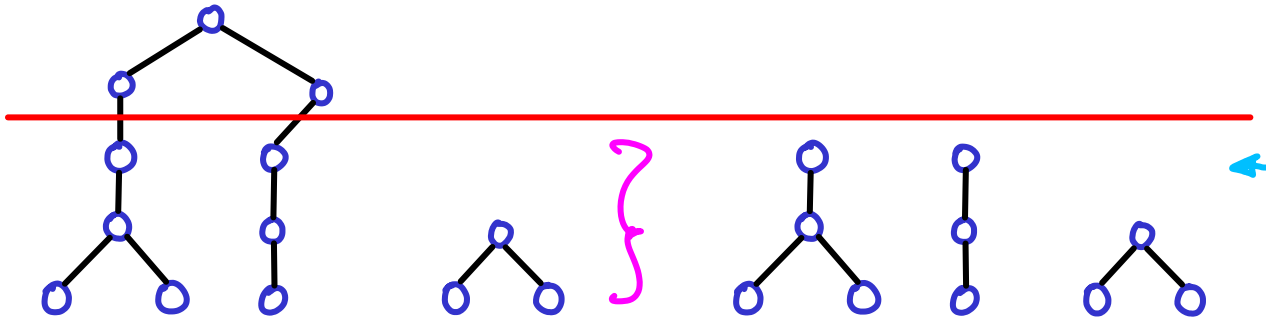
QUAKE : $\Phi = N + T$ doesn't work: We're stuck with ΔT



$\Delta T \leq n_v$
= # nodes at quake level

B_{v+1} = # nodes at level v+1 with only 1 child

QUAKE : $\Phi = N + T$ doesn't work: We're stuck with ΔT



$\Delta T \leq n_v$

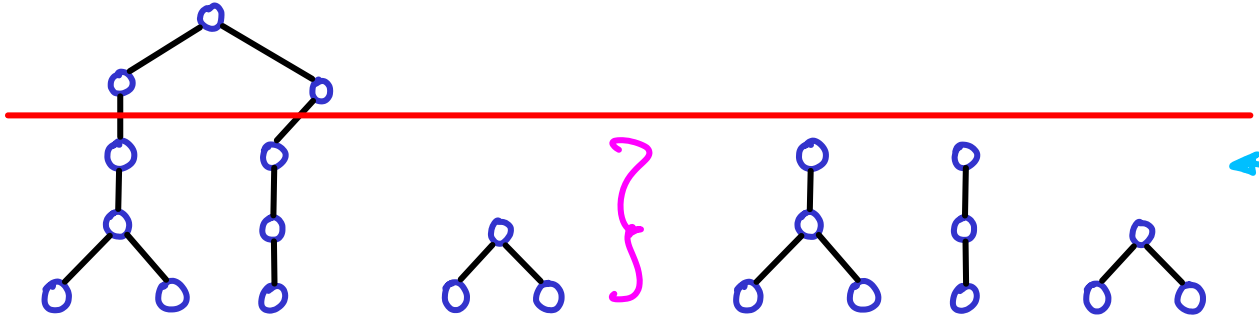
= # nodes at quake level

$n_v \geq 2n_{v+1} - B_{v+1}$

B_{v+1} = # nodes at level $v+1$ with only 1 child

Equal if every node in n_v has a parent

QUAKE : $\Phi = N + T$ doesn't work: We're stuck with ΔT



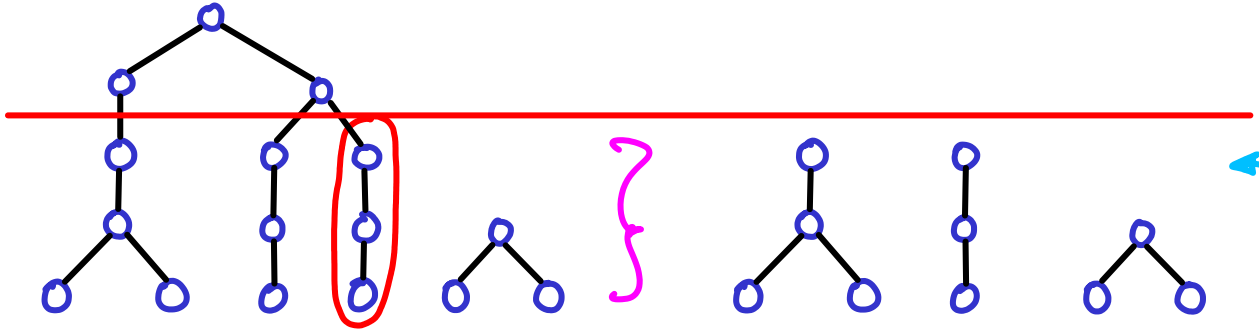
$\Delta T \leq n_v$

= # nodes at quake level

$n_v \geq 2n_{v+1} - B_{v+1}$

B_{v+1} = # nodes at level $v+1$ with only 1 child $\geq 2n_{v+1} - n_v$

QUAKE : $\Phi = N + T$ doesn't work: We're stuck with ΔT



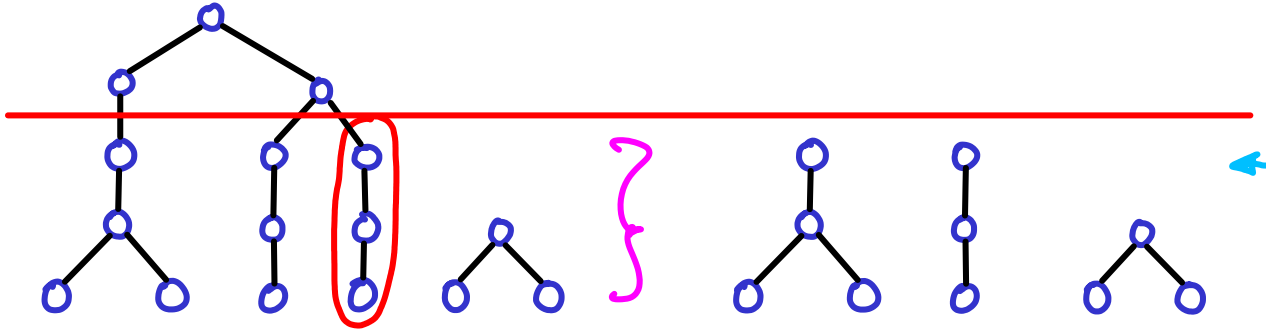
$\Delta T \leq n_v$
= # nodes at quake level

$n_v \geq 2n_{v+1} - B_{v+1}$

B_{v+1} = # nodes at level $v+1$ with only 1 child $\geq 2n_{v+1} - n_v$

violation $\Rightarrow$ $\frac{n_{v+1}}{n_v} > \frac{3}{4}$

QUAKE : $\Phi = N + T$ doesn't work: We're stuck with ΔT



$\Delta T \leq n_v$

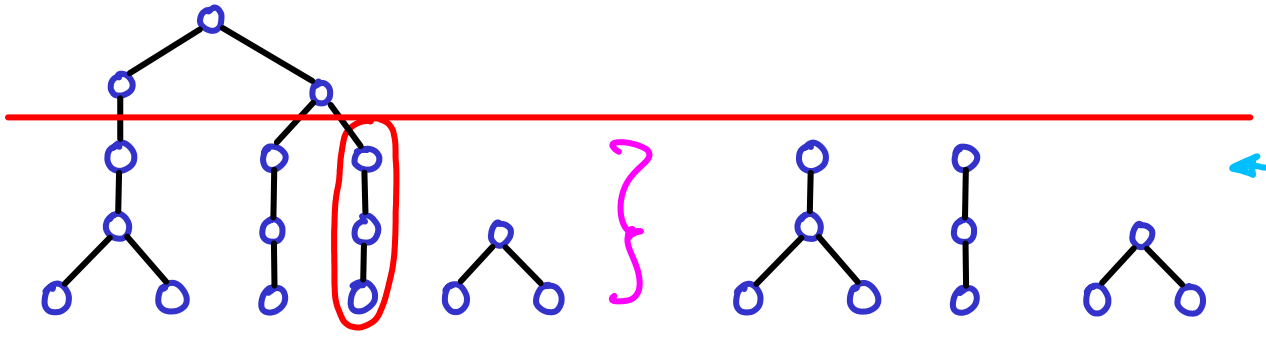
$= \#$ nodes at quake level

$n_v \geq 2n_{v+1} - B_{v+1}$

$B_{v+1} = \#$ nodes at level $v+1$ with only 1 child $\geq 2n_{v+1} - n_v > 2 \cdot \frac{3}{4} n_v - n_v$

violation $\Rightarrow \frac{n_{v+1}}{n_v} > \frac{3}{4}$

QUAKE : $\Phi = N + T$ doesn't work: We're stuck with ΔT



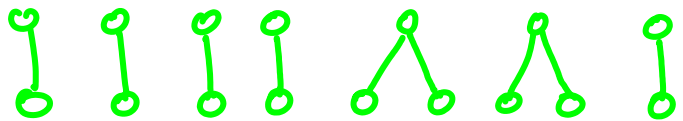
$\Delta T \leq n_v$

= # nodes at quake level

$n_v \geq 2n_{v+1} - B_{v+1}$

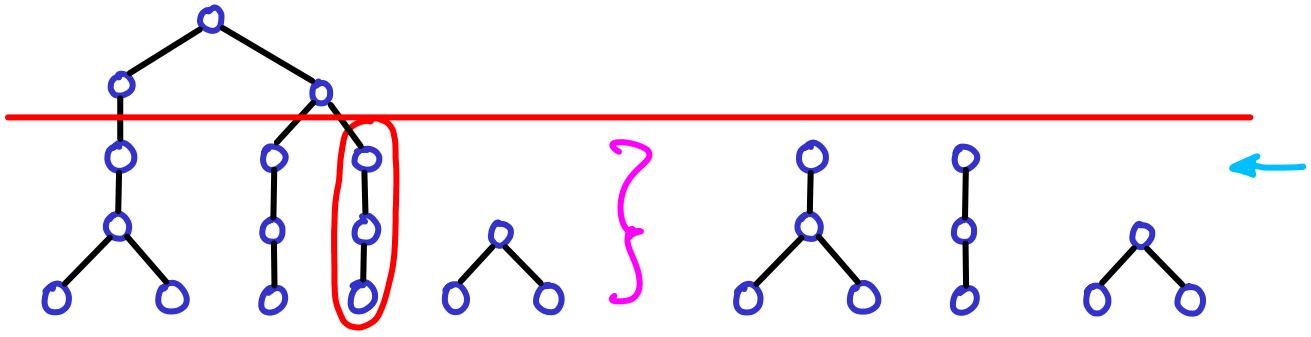
B_{v+1} = # nodes at level $v+1$ with only 1 child $\geq 2n_{v+1} - n_v > 2 \cdot \frac{3}{4} n_v - n_v$

$B_{v+1} > \frac{1}{2} n_v$



violation $\Rightarrow$ $\frac{n_{v+1}}{n_v} > \frac{3}{4}$

QUAKE : $\Phi = N + T$ doesn't work: We're stuck with ΔT



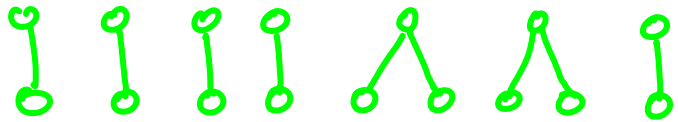
$\Delta T \leq n_v$

= # nodes at quake level

$n_v \geq 2n_{v+1} - B_{v+1}$

B_{v+1} = # nodes at level v+1 with only 1 child $\geq 2n_{v+1} - n_v > 2 \cdot \frac{3}{4} n_v - n_v$

$B_{v+1} > \frac{1}{2} n_v$

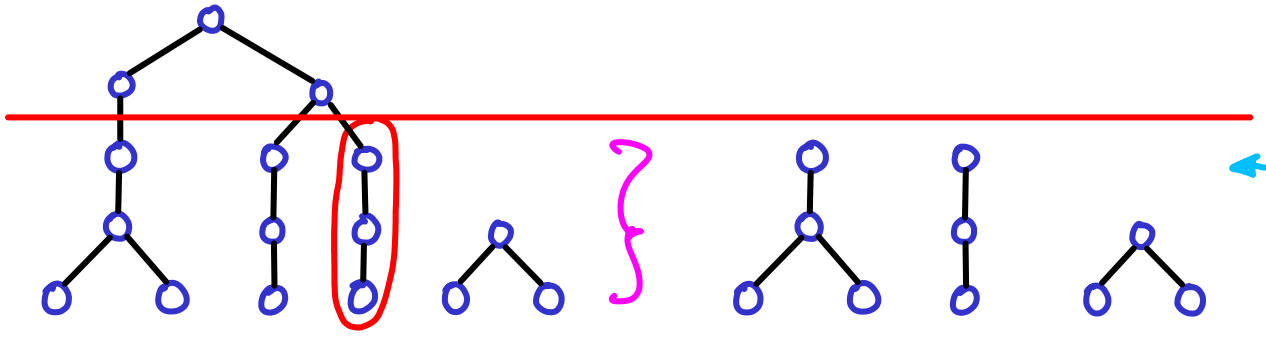


violation $\Rightarrow \frac{n_{v+1}}{n_v} > \frac{3}{4}$

With the quake, # B-nodes goes down by at least B_{v+1}

→ Any node with only 1 child, in all levels

QUAKE : $\Phi = N + T$ doesn't work: We're stuck with ΔT



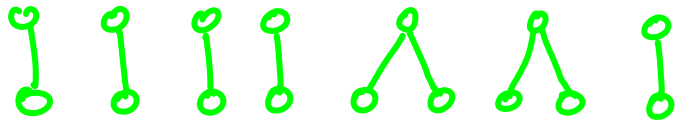
$\Delta T \leq n_v$

= # nodes at quake level

$n_v \geq 2n_{v+1} - B_{v+1}$

B_{v+1} = # nodes at level $v+1$ with only 1 child $\geq 2n_{v+1} - n_v > 2 \cdot \frac{3}{4} n_v - n_v$

$B_{v+1} > \frac{1}{2} n_v$

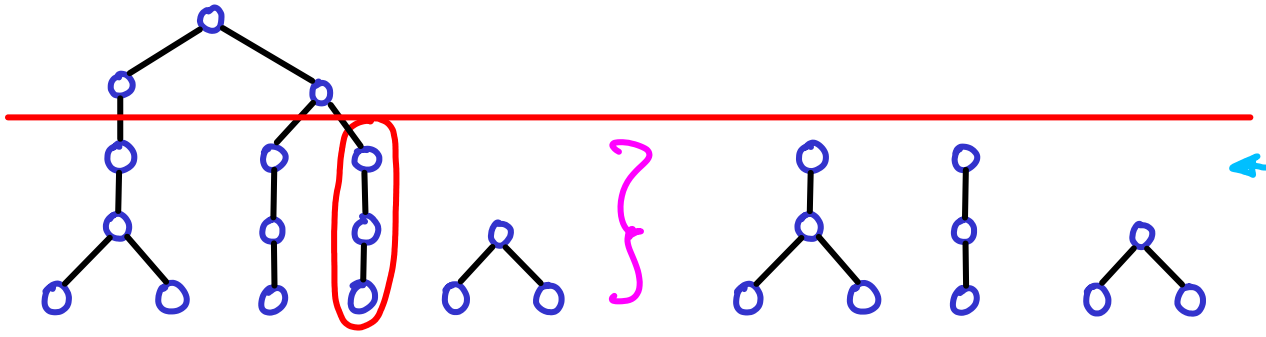


violation $\Rightarrow$ $\frac{n_{v+1}}{n_v} > \frac{3}{4}$

With the quake, # B-nodes goes down by at least B_{v+1}

$\Delta B \leq - B_{v+1}$

QUAKE : $\Phi = N + T$ doesn't work: We're stuck with ΔT



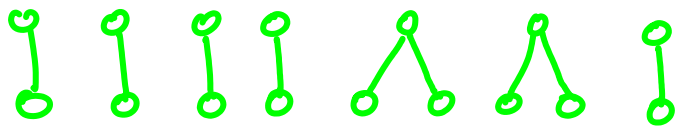
$\Delta T \leq n_v$

= # nodes at quake level

$n_v \geq 2n_{v+1} - B_{v+1}$

B_{v+1} = # nodes at level $v+1$ with only 1 child $\geq 2n_{v+1} - n_v > 2 \cdot \frac{3}{4} n_v - n_v$

$B_{v+1} > \frac{1}{2} n_v$

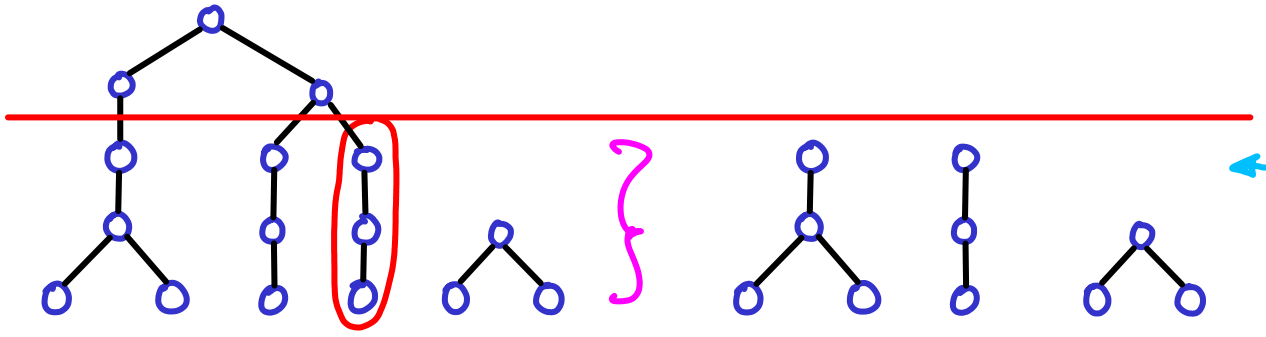


violation $\Rightarrow \frac{n_{v+1}}{n_v} > \frac{3}{4}$

With the quake, # B-nodes goes down by at least B_{v+1}

$\Delta B \leq - B_{v+1} < -\frac{1}{2} n_v$

QUAKE : $\Phi = N + T$ doesn't work: We're stuck with ΔT



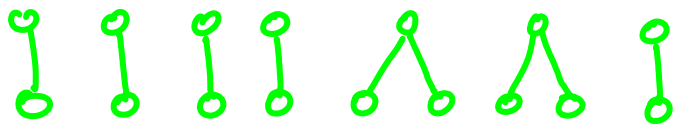
$\Delta T \leq n_v$

= # nodes at quake level

$n_v \geq 2n_{v+1} - B_{v+1}$

B_{v+1} = # nodes at level $v+1$ with only 1 child $\geq 2n_{v+1} - n_v > 2 \cdot \frac{3}{4} n_v - n_v$

$B_{v+1} > \frac{1}{2} n_v$

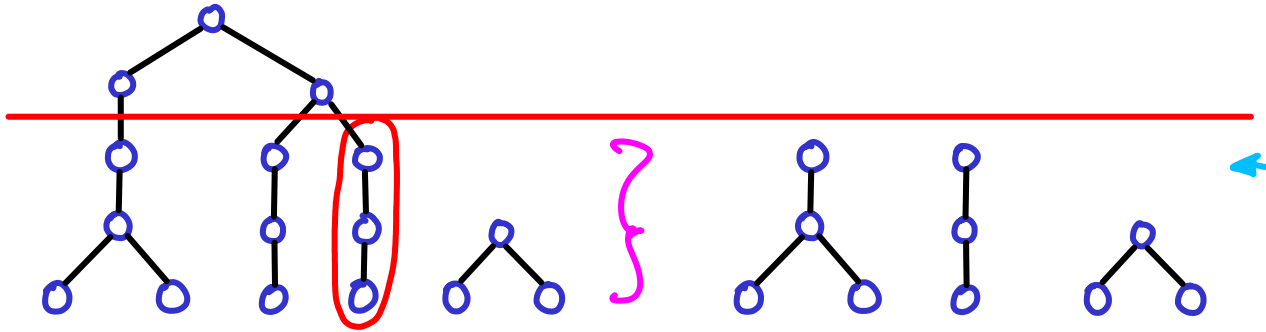


violation $\Rightarrow$ $\frac{n_{v+1}}{n_v} > \frac{3}{4}$

With the quake, # B-nodes goes down by at least B_{v+1}

$2\Delta B \leq -2B_{v+1} < -\frac{1}{2} n_v$

QUAKE : $\Phi = N + T$ doesn't work: We're stuck with ΔT



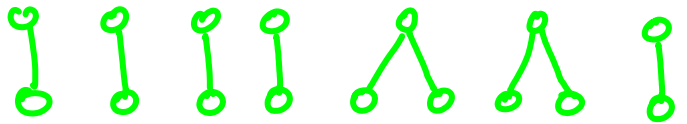
$\Delta T \leq n_v$

$= \#$ nodes at quake level

$n_v \geq 2n_{v+1} - B_{v+1}$

$B_{v+1} = \#$ nodes at level $v+1$ with only 1 child $\geq 2n_{v+1} - n_v > 2 \cdot \frac{3}{4} n_v - n_v$

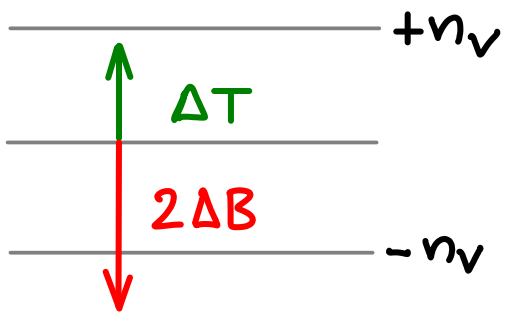
$B_{v+1} > \frac{1}{2} n_v$



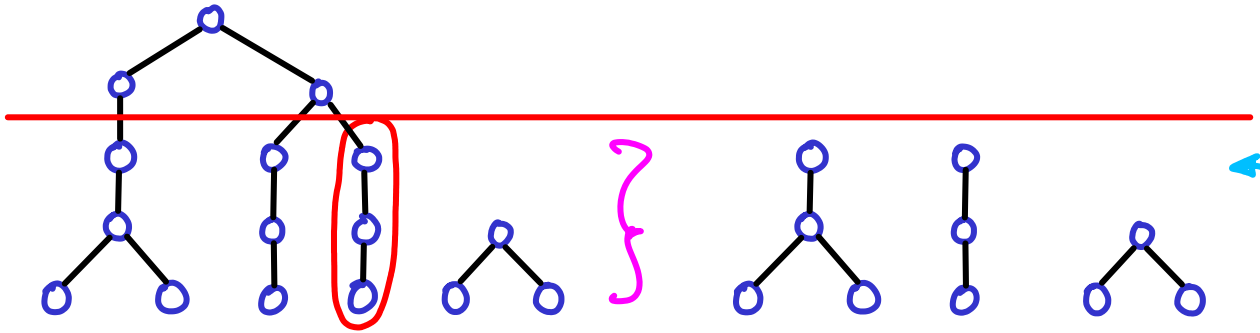
violation $\Rightarrow \frac{n_{v+1}}{n_v} > \frac{3}{4}$

With the quake, #B-nodes goes down by at least B_{v+1}

$2\Delta B \leq -2B_{v+1} < -\frac{1}{2} n_v \Rightarrow \Delta T + 2\Delta B < 0$



QUAKE : $\Phi = N + T$ doesn't work: We're stuck with ΔT

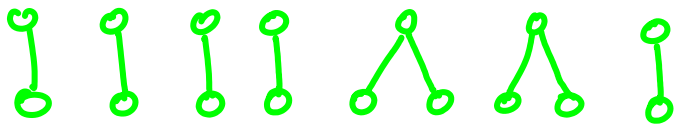


$\Delta T \leq n_v$
= # nodes at quake level

$n_v \geq 2n_{v+1} - B_{v+1}$

B_{v+1} = # nodes at level v+1 with only 1 child $\geq 2n_{v+1} - n_v > 2 \cdot \frac{3}{4} n_v - n_v$

$B_{v+1} > \frac{1}{2} n_v$

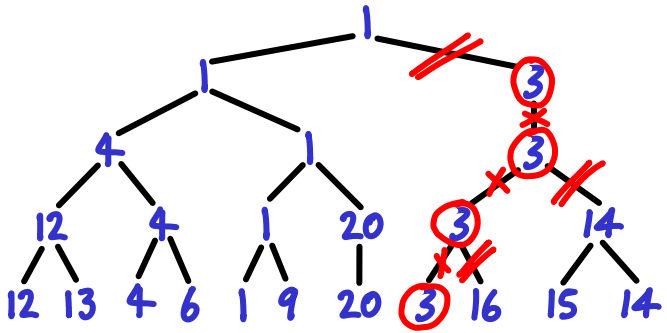


violation $\Rightarrow$ $\frac{n_{v+1}}{n_v} > \frac{3}{4}$

With the quake, # B-nodes goes down by at least B_{v+1}

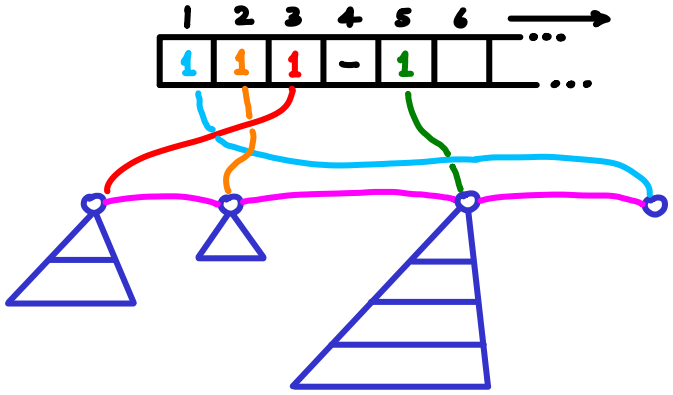
$2\Delta B \leq -2B_{v+1} < -\frac{1}{2} n_v \Rightarrow \Delta T + 2\Delta B < 0 \Rightarrow$ Try $\Phi = N + T + 2B$

Whatever ΔT is, when it is "expensive" ΔB changes by ~just as much, negatively

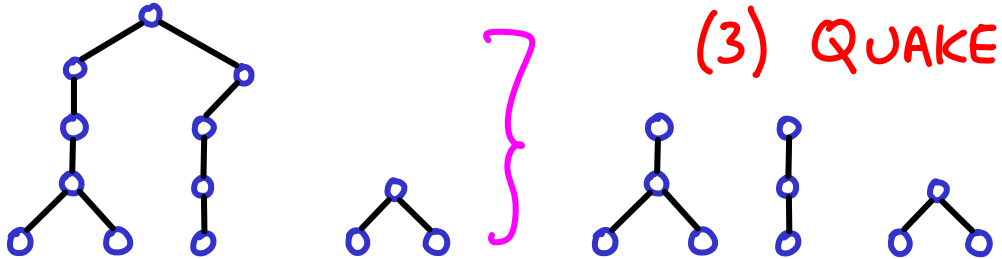


DELETE

(1) CUT



(2) LINK



(3) QUAKE

AMORTIZE

$\Phi = N + c \cdot T + c \cdot 2B$

$c = O(\log n)$ $\quad \Delta\Phi \leq 0 + O(\log n) \underline{+ 2c}$

$\hat{c} = O(\log n)$ ✓

$c = O(T_{i-1}) + O(\log n)$ $\quad \Delta\Phi \leq T_{i-1} + c(\log n - T_{i-1}) \underline{+ 0}$

$\hat{c} = O(\log n)$ ✓

$c = \sum_{i=v}^{top} n_i$ (actually $O(\Sigma)$: easy to handle)

$2\Delta B < -\Delta T$

$\Delta\Phi \leq -\sum_{i=0}^{v} n_i + c\Delta T - c\Delta T$ ✓

c

$\Delta\Phi = \Delta N + \Delta T + 2\Delta B$

ignore c.

It's just a detail.

REPORT MIN: $O(1)$

INSERT: $O(1)$

DECREASE KEY: $O(1)$

CUT above highest clone

UNION: be lazy: $O(1)$

	c	$\Delta\Phi = \Delta N + \Delta T + 2\Delta B$
REPORT MIN:	$O(1)$	0
INSERT:	$O(1)$	
DECREASE KEY: CUT above highest clone	$O(1)$	
UNION: be lazy:	$O(1)$	

	c	$\Delta\Phi = \Delta N + \Delta T + 2\Delta B$
REPORT MIN:	$O(1)$	0
INSERT:	$O(1)$	$+1 \quad +1 \quad +0$
DECREASE KEY: CUT above highest clone	$O(1)$	
UNION: be lazy:	$O(1)$	

	c	$\Delta\Phi = \Delta N + \Delta T + 2\Delta B$
REPORT MIN:	$O(1)$	0
INSERT:	$O(1)$	$+1 \quad +1 \quad +0$
DECREASE KEY:	$O(1)$	$0 \quad +1 \quad +2$
CUT above highest clone		
UNION: be lazy:	$O(1)$	

	c	$\Delta\Phi = \Delta N + \Delta T + 2\Delta B$
REPORT MIN:	$O(1)$	0
INSERT:	$O(1)$	$+1 \quad +1 \quad +0$
DECREASE KEY:	$O(1)$	$0 \quad +1 \quad +2$
CUT above highest clone		
UNION: be lazy:	$O(1)$	0

□ ?

	c	$\Delta\Phi = \Delta N + \Delta T + 2\Delta B$
REPORT MIN:	$O(1)$	0
INSERT:	$O(1)$	$+1 \quad +1 \quad +0$
DECREASE KEY:	$O(1)$	$0 \quad +1 \quad +2$

CUT above highest clone

UNION: be lazy: $O(1)$ 0

No longer have one list of n_i

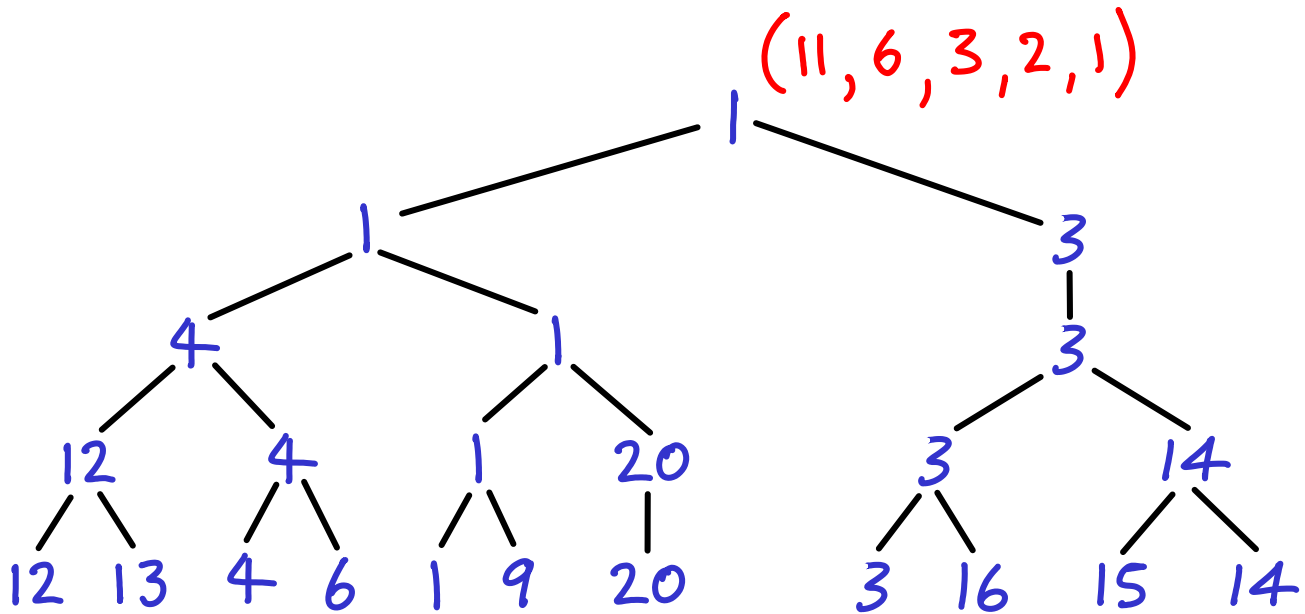
□ ?

Having multiple lists of level counts is a problem.
It can slow down detection of alpha-violations.

Next we show one way of merging all the lists during DELETE.
(This is not in Chan's paper)

Suppose we have one tournament tree, X.

At the root we store a linked list of n_i

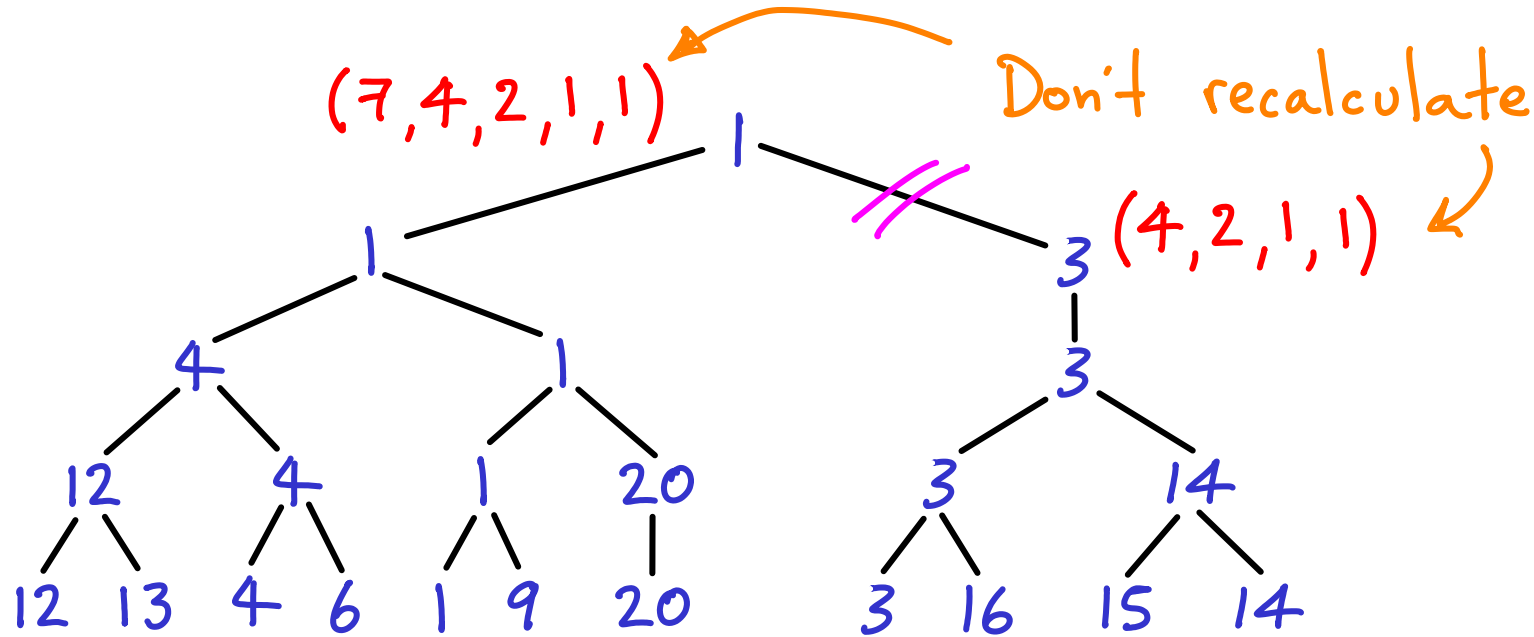


Suppose we have one tournament tree, X.

At the root we store a linked list of n_i

If we CUT anywhere, let X keep its list,

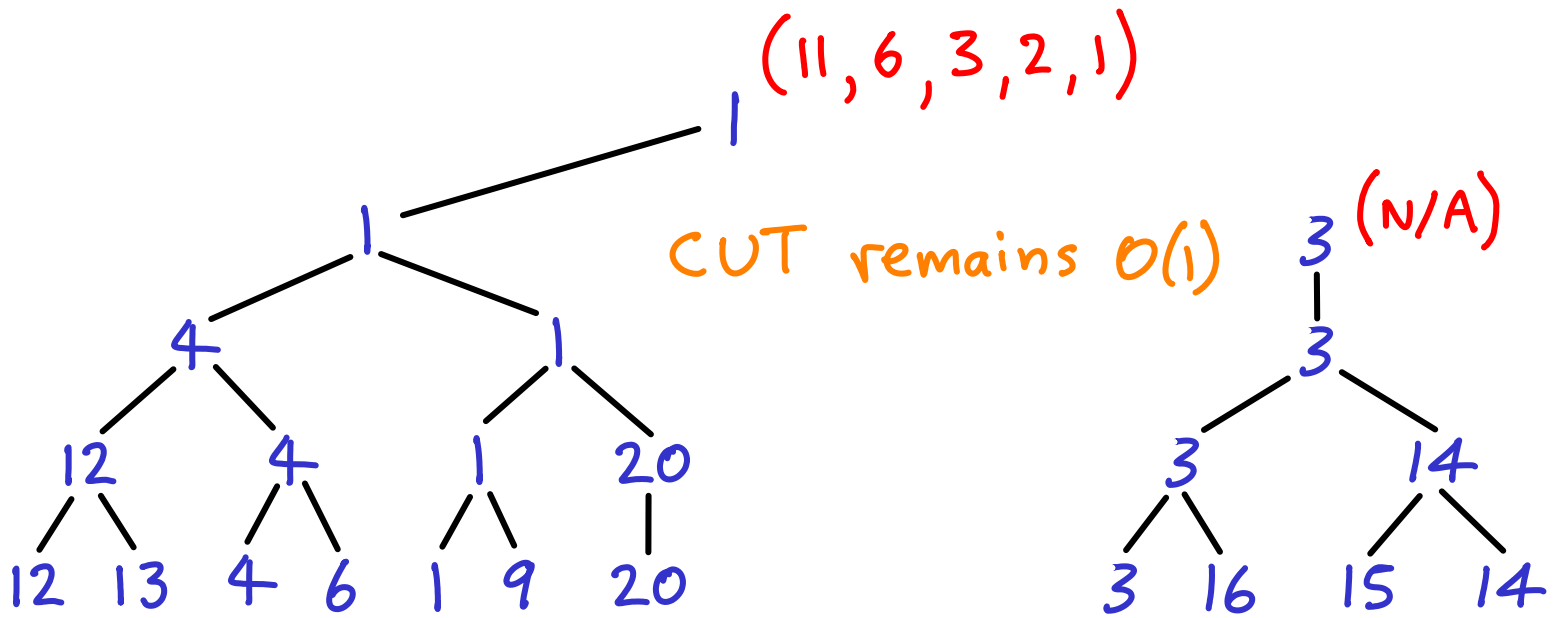
to keep representing the cut subtree.



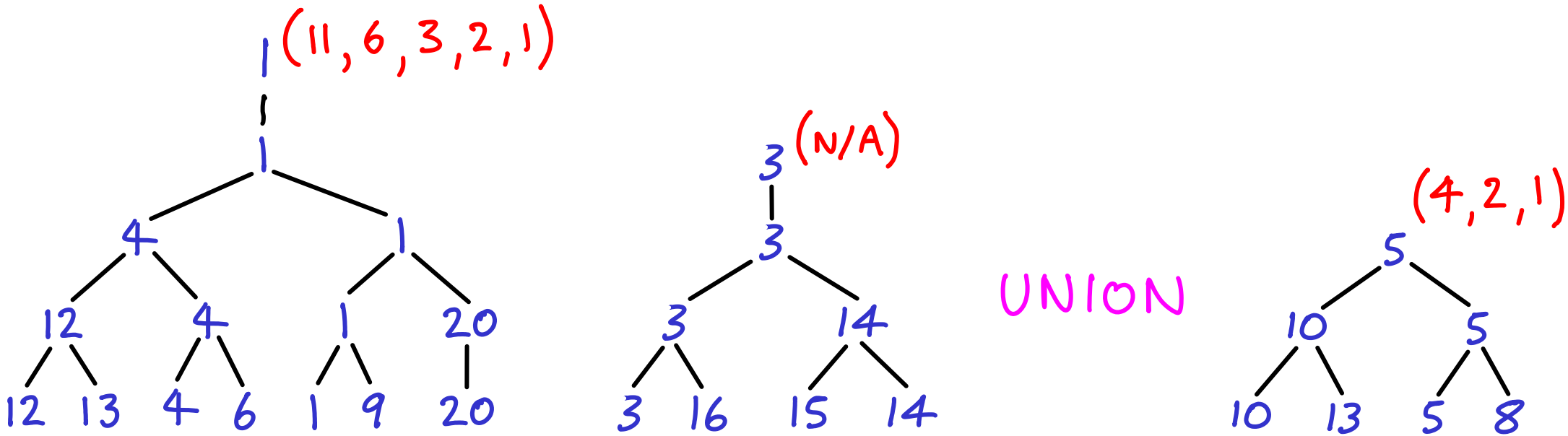
Suppose we have one tournament tree, X.

At the root we store a linked list of n_i

If we CUT anywhere, let X keep its list,
to keep representing the cut subtree.

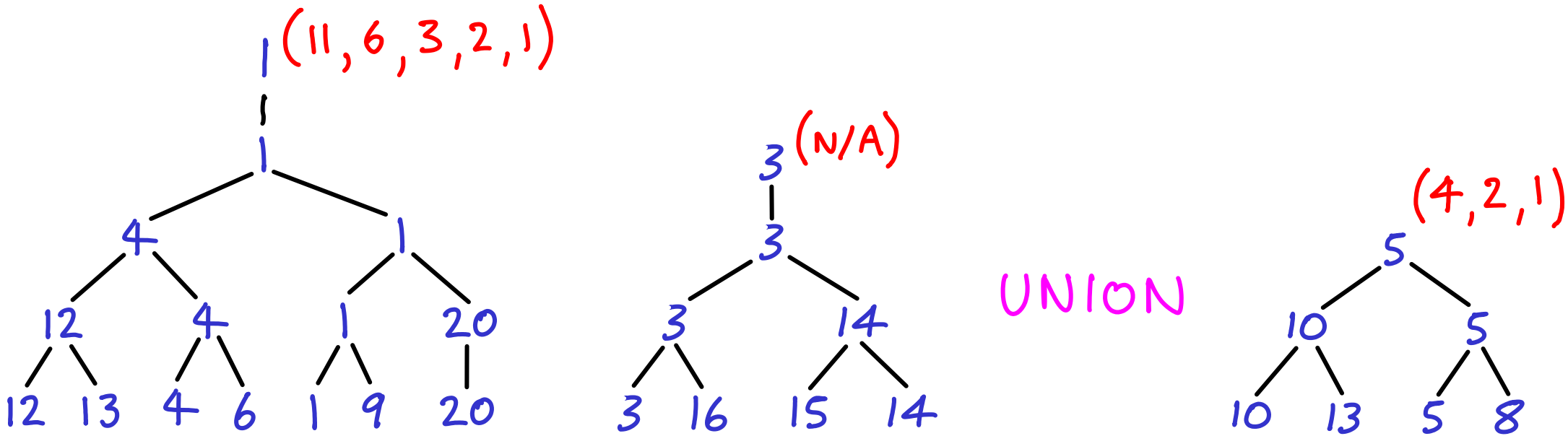


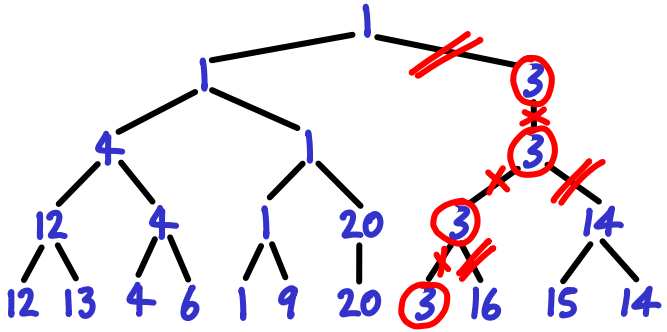
So by cutting or by UNION (or INSERT) we could have multiple lists.



So by cutting or by UNION (or INSERT) we could have multiple lists.

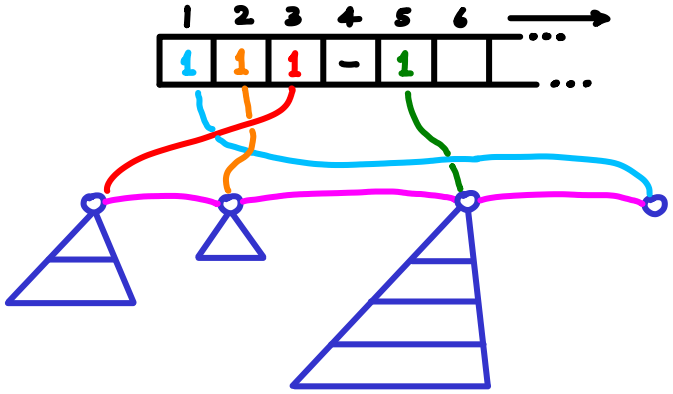
We only care when we need to check for α-violation: DELETE



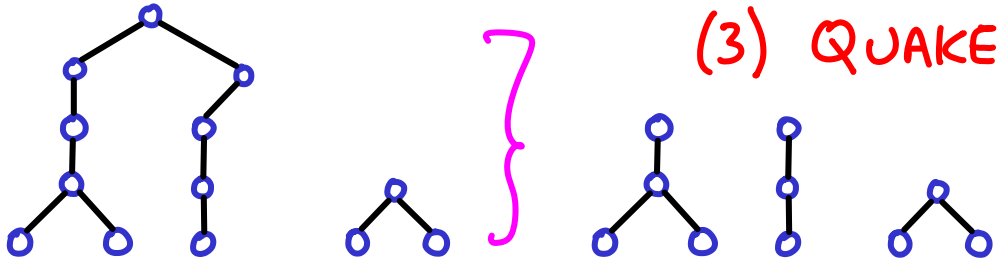


DELETE

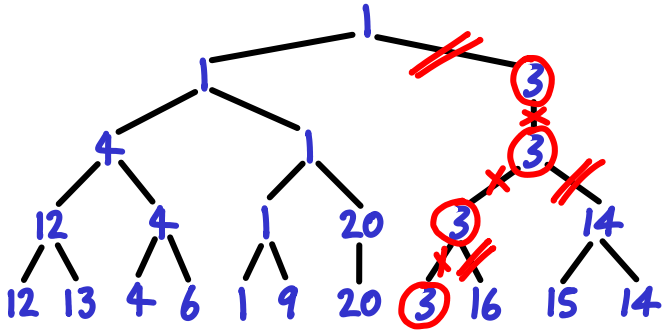
(1) CUT



(2) LINK

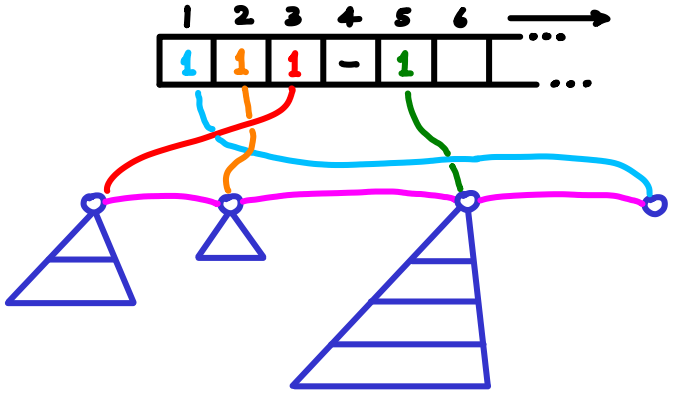


(3) QUAKE

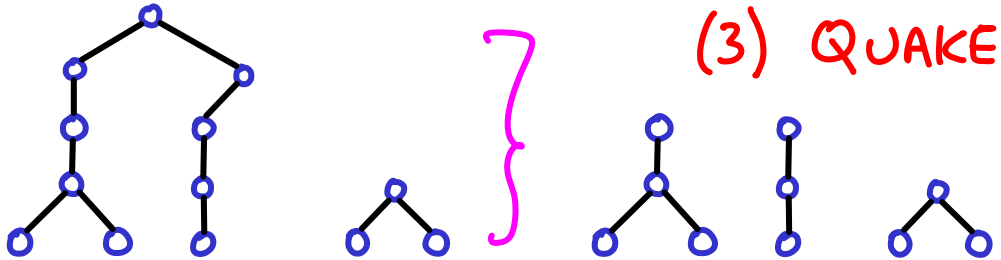


DELETE

(1) CUT



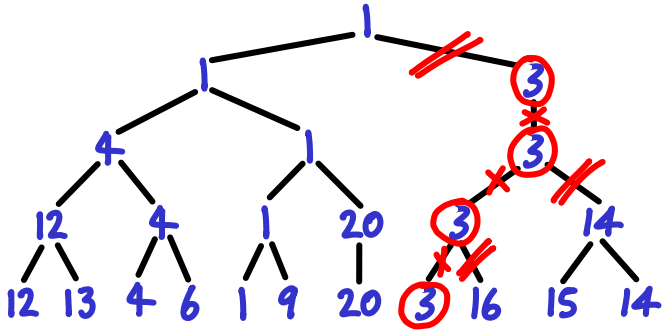
(2) LINK



(3) QUAKE

Keep track of how many levels lost one node. (store one level number)

No effect on time. Still O(logn)

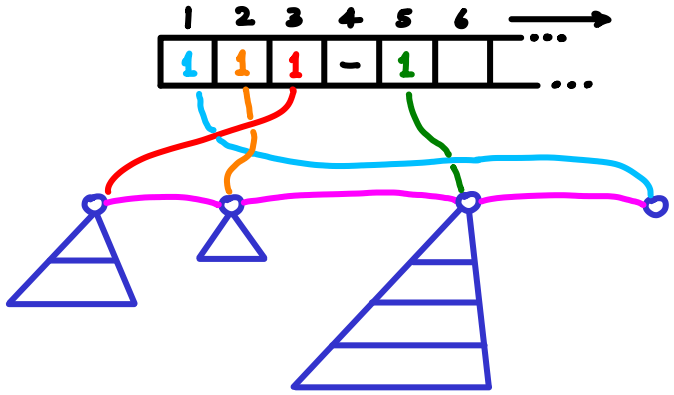


DELETE

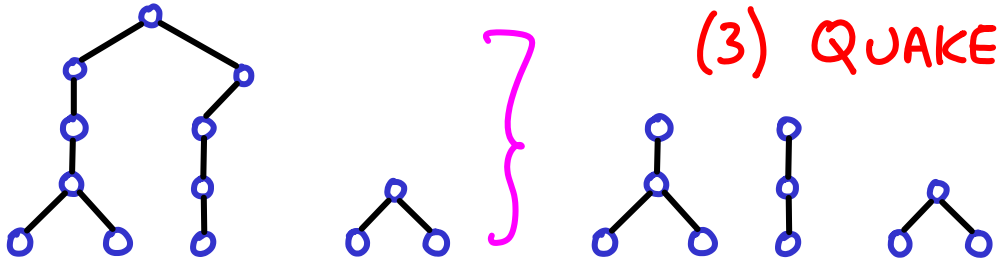
(1) CUT

} Keep track of how many levels lost one node. (store one level number)

No effect on time. Still O(logn)



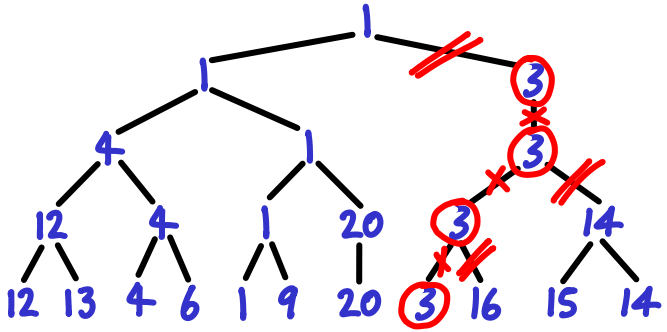
(2) LINK



(3) QUAKE

} As before

(Delete parts of lists corresponding to Quaked levels: cheap)

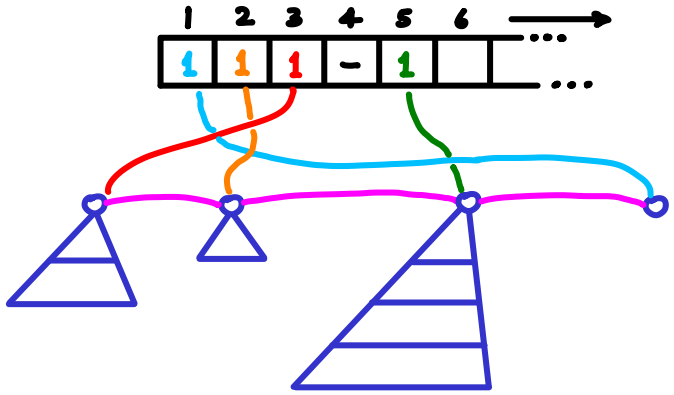


DELETE

(1) CUT

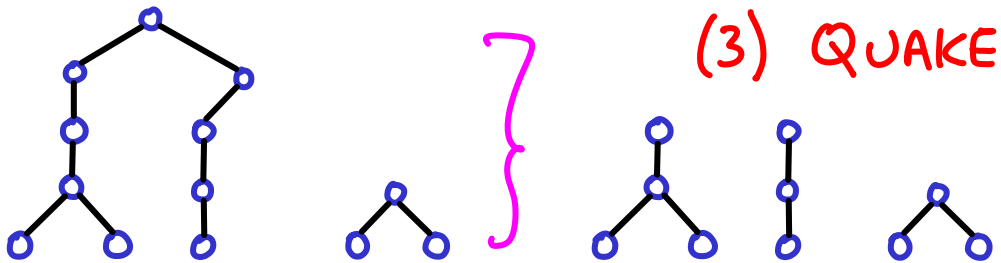
Keep track of how many levels lost one node. (store one level number)

No effect on time. Still O(logn)



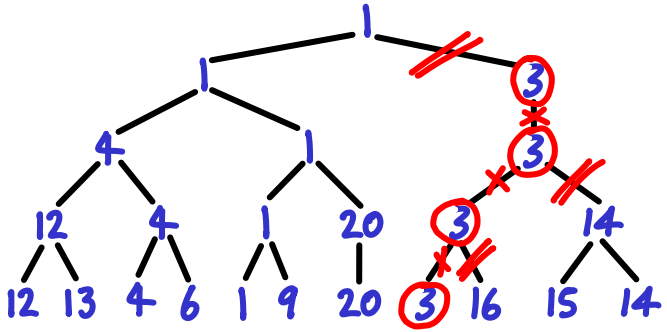
(2) LINK

For every LINK, merge lists if applicable.



(3) QUAKE

As before

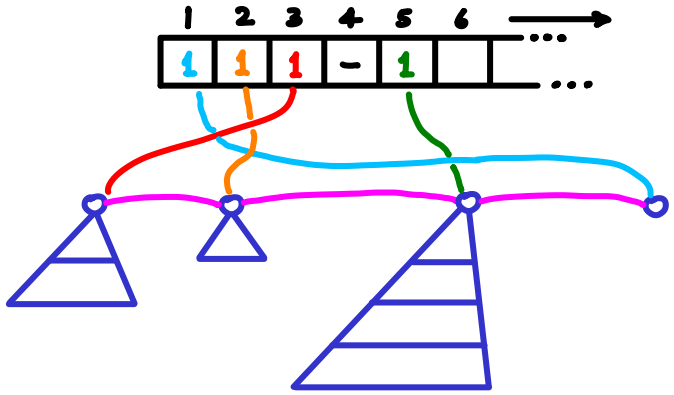


DELETE

(1) CUT

Keep track of how many levels lost one node. (store one level number)

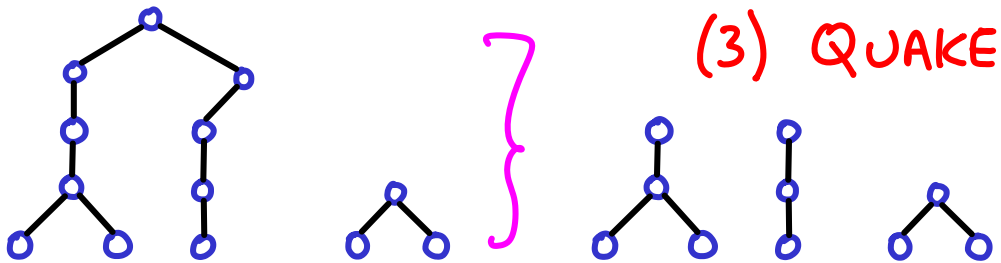
No effect on time. Still O(logn)



(2) LINK

For every LINK, merge lists if applicable.

Then merge O(logn) remaining lists.

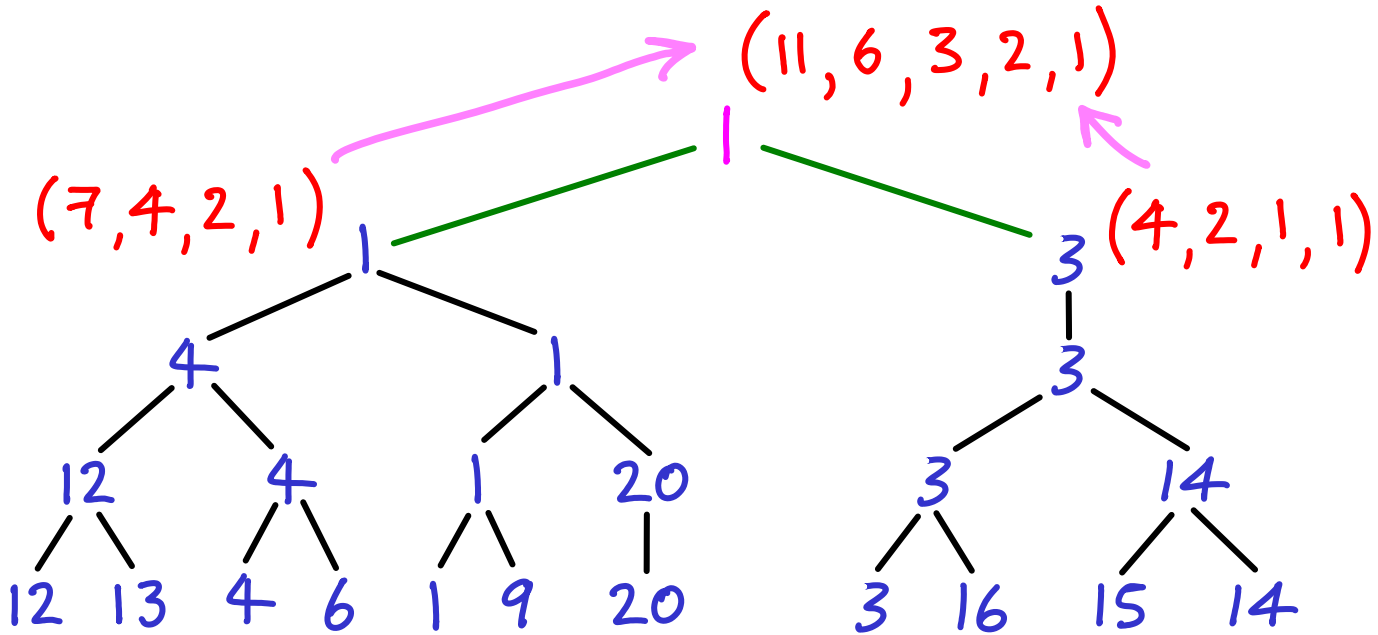


(3) QUAKE

As before

LINK cost ~ height of new node.

LINK ⟷ create new node

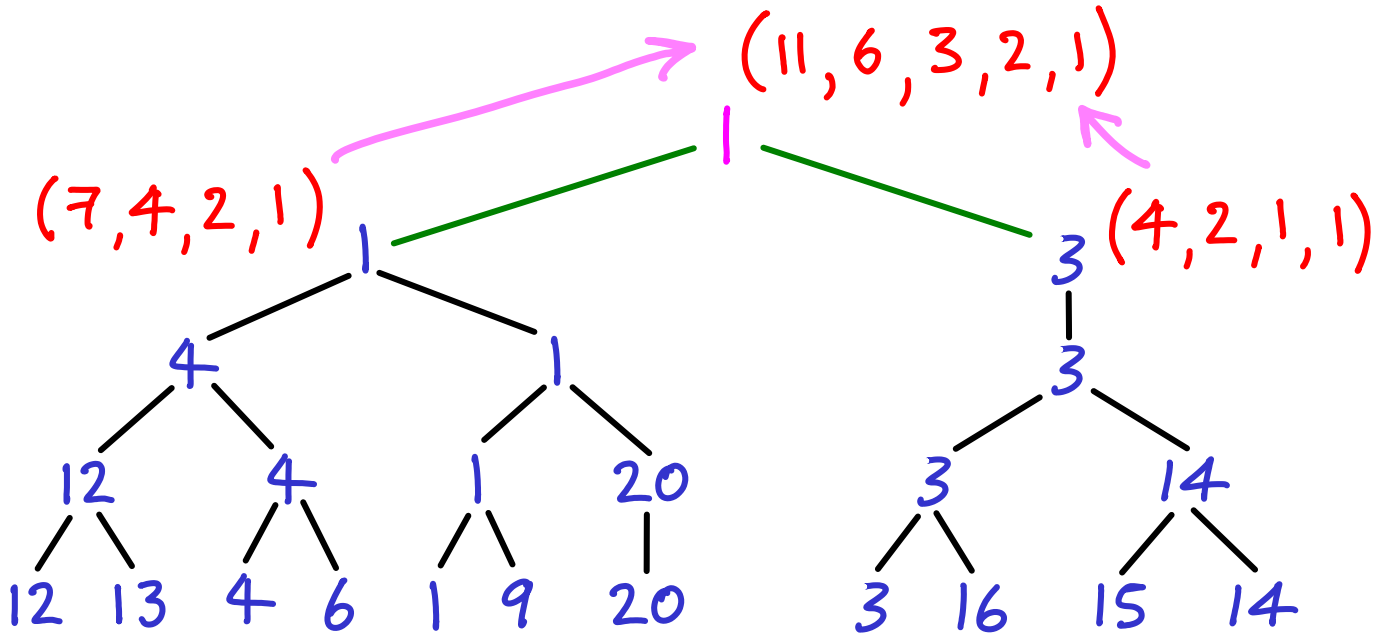


LINK cost ~ height of new node.

LINK $\longleftrightarrow$ create new node

↳ Total cost of <u>all</u> LINK operations = $\sum_{\text{all } x} \text{height}(x) = O(n)$

like binary heap build

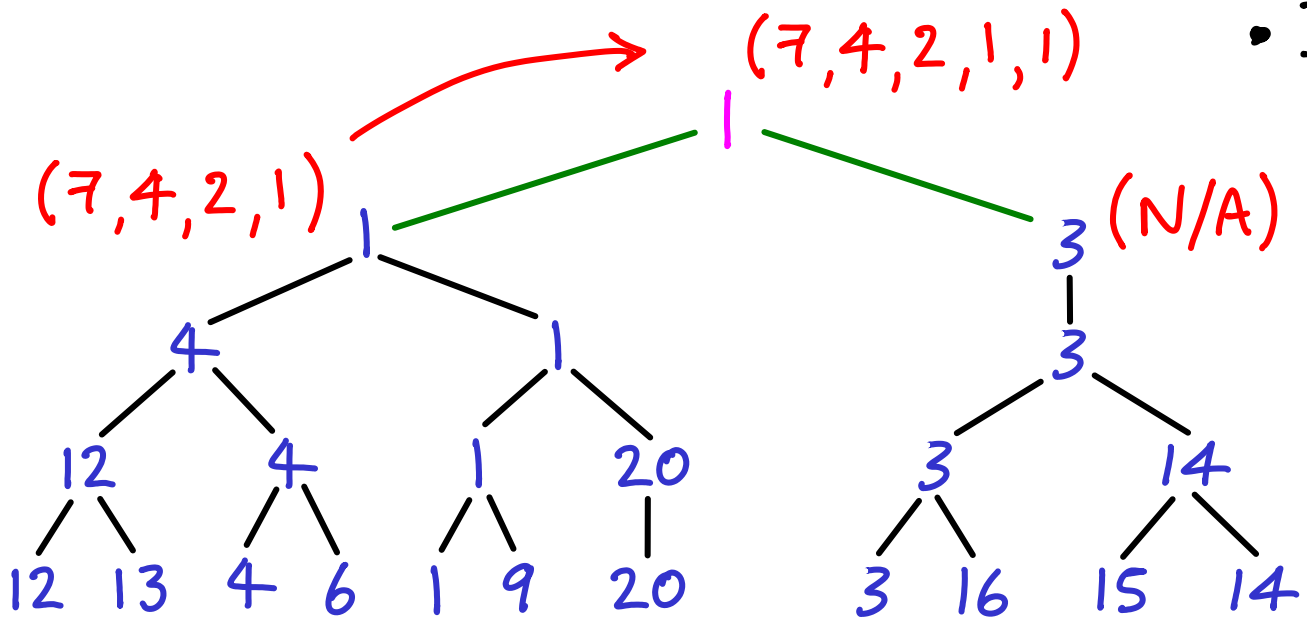


LINK cost ~ height of new node.

LINK $\longleftrightarrow$ create new node

↳ Total cost of all LINK operations = $\sum_{\text{all } x} \text{height}(x) = O(n)$

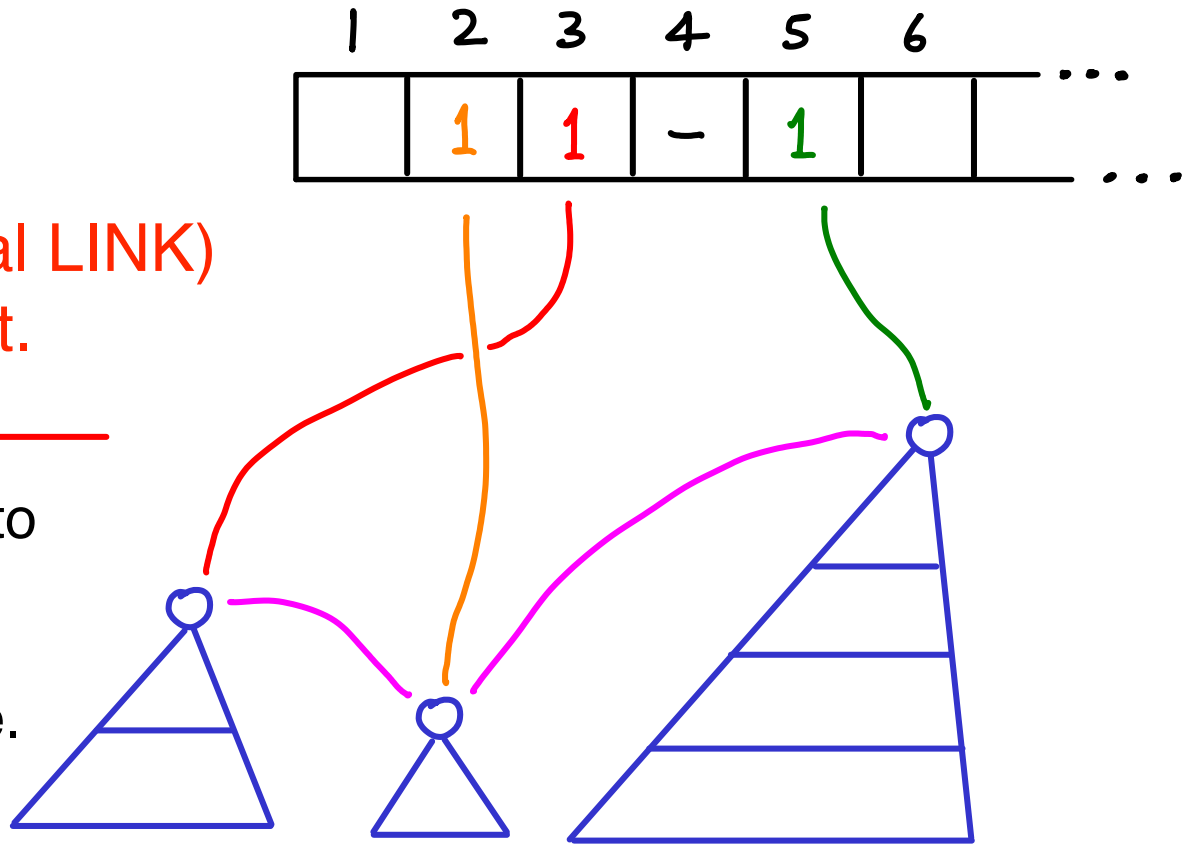
like binary heap build



- In fact if one tree has no list then no merge, just append 1.
- If neither has a list, +1 in a temp array for the new root.

When we are left with < logn trees,
merge their lists as well,
from smallest to largest. (No actual LINK)
Each root can still pay for its height.

At the end of DELETE, we still have up to logn trees, but now all but one of them will have “N/A” as a count list, so in the future they will not contribute to a merge.



The result is that we now have one count list.

We maintain the invariant:
over all operations of a Quake heap, every node pays for its height, at most.

Summary of merging the count lists during DELETE:

In step 1 of DELETE: Record level, L, of topmost deleted node.

In step 2: Merge count lists during LINK. Also store certain increments in
an array of size logn, for new nodes that were generated without actual merging of lists.

In step 3: Delete parts of lists in Quaked levels.
This could affect L: it could be updated (reduced) to the highest surviving level.

Finally merge lists of trees with distinct heights, to get one list.
Then merge this with the array of step 2, and decrement every count L and all levels below.

Can vary α and implement in other ways (see paper) } project?

Exercise: Analyze Quake heap with accounting method.