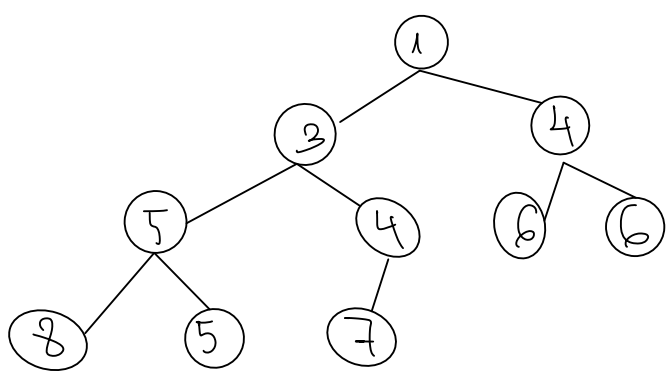
FIBONACCI HEAPS

Recall Binary Heap:

A binary heap, satisfies these 2 properties:

- Every level is full, last one from left to right.
- Heap property: No child is smaller than its parent



Costs of basic operations in binary heap:

- INSERT : $\Theta(\lg n)$
- MINIMUM: $\Theta(1)$
- EXTRACT-MIN: $\Theta(\lg n)$
- UNION : $\Theta(n)$
- DECREASE-KEY : $\Theta(\lg n)$
- DELETE : $\Theta(\lg n)$

Observations: Most of the time, binary heaps work well. However, if we need to use UNION operation or we need constant performance for all operations for INSERT and DECREASE-KEY operations, then Fibonacci Heap works better.

Costs of Fibonacci Heap operations:

- INSERT : $\Theta(1)$
- MINIMUM: $\Theta(1)$
- EXTRACT-MIN: $\Theta(\lg n)$
- UNION : $\Theta(1)$
- DECREASE-KEY : $\Theta(1)$
- DELETE : $\Theta(\lg n)$

⇒ Goal of Fibonacci Heap:

- Allow us to be lazy, i.e do as little work as possible (INSERT, UNION, DECREASE-KEY, ...) and only performs when necessary (EXTRACT-MIN)

Structure of Fibonacci Heap:

A node in Fibonacci heap is defined with these properties

```
Heap H:
    H.min
    H.n

Node x:
    x.parent
    x.child
    x.left
    x.right
    x.degree
    x.mark
```

H.min → point to min

H.n → number nodes in heap

x.parent → point to parent node

x.child → point to any of the children

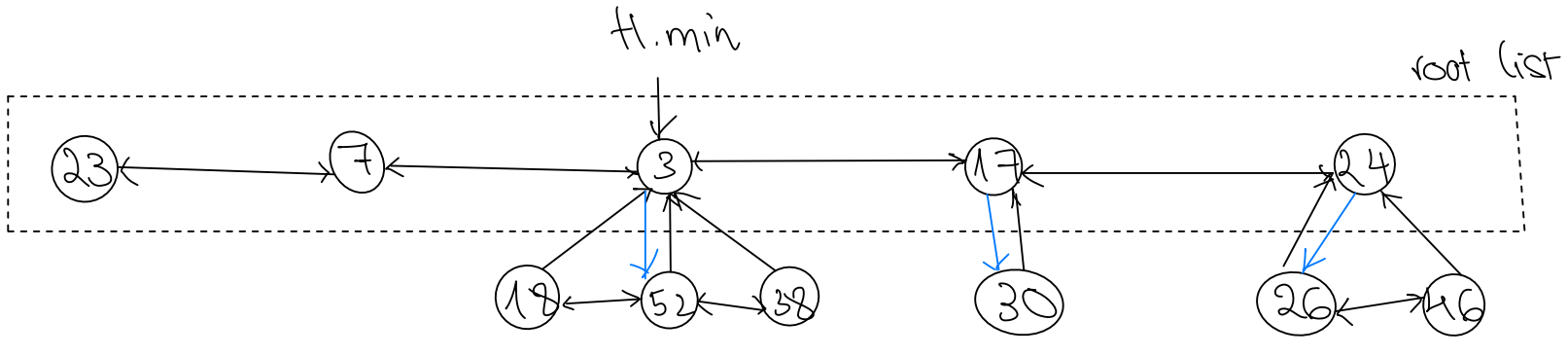
x.left, x.right → doubly linked list

x.degree → number of children

x.mark → lost child's indicator:

- marked when node is a child
- unmarked when node has no child

- Fibonacci heap has many roots that are connected together through "left" and "right" pointers as doubly linked list. All children in the same level of a root are also connected in similar manner.



Amortized Analysis Fibonacci Heap

- Potential Function:

The potential function used to analyze amortized cost of Fibonacci heap is defined as:

$$\phi(H) = t(H) + 2 \cdot m(H)$$

$t(H)$: # trees

$m(H)$: # marked nodes

Assuming we start with empty heap, the initial potential is 0. And the potential remain non negative throughout the operations

- INSERT (H, x)

- The philosophy of Fibonacci Heap is to delay the expensive work for as long as possible.
- In the case of insert, it simply add new node into the root list.
- The change in potential is therefore:

$$\begin{aligned}\Delta\phi &= \phi(H') - \phi(H) \\ &= \left[t(H') + 2 \cdot m(H')\right] - \left[t(H) + 2 \cdot m(H)\right] \\ &= t(H) + 1 + 2m(H) - t(H) - 2m(H) \\ &= 1\end{aligned}$$

The amortized cost:

$$\begin{aligned}\hat{c} &= c + \Delta\phi \\ &= O(1) + 1 \\ &= O(1)\end{aligned}$$

- UNION (H_1, H_2)
  - This operation unites H_1 and H_2, destroy both in the process. It concatenates H_1 and H_2 based on the root with smaller value
  - The change in potential is:

$$\begin{aligned}\Delta\phi &= \phi(H_{12}) - \left(\phi(H_1) + \phi(H_2)\right) \\ &= \left(t(H_{12}) - 2m(H_{12})\right) - \left(t(H_1) + 2m(H_1) + t(H_2) + 2m(H_2)\right) \\ &= 0 \quad \text{because} \begin{cases} t(H_{12}) = t(H_1) + t(H_2) \\ m(H_{12}) = m(H_1) + m(H_2) \end{cases}\end{aligned}$$

The amortized cost is:

$$\begin{aligned}\hat{c} &= c + \Delta\phi \\ &= O(1) + 0 \\ &= O(1)\end{aligned}$$

EXTRACT_MIN (H):

- This is where the delayed work is carried out, it consolidate trees in the root list.

Pseudo code:

```
EXTRACT_MIN (H):
    min_node = H.min
    if min_node = NIL:
        return min_node
    for child : min_node.children:
        add child to root_list
        child.parent := NIL
    remove min_node from root_list
    if min_node = min_node.right:
        H.min := NIL
    else:
        H.min := min_node.right
        CONSOLIDATE (H)
    H.n -= 1
    return min_node
```

empty heap

move children to root list, remove minimum node

min node is the last node in root list

set new min node to any neighbor then consolidate

```
CONSOLIDATE (H)
    init array A[0 ... D(H.n)] to nil
    for each node w in root_list:
        x, d := w, w.degree
        while A[d] ≠ NIL:
            y := A[d]
            if x.key < y.key: swap x, y
            FIB_HEAP_LINK(H, y, x)
            A[d] := NIL
            d += 1
        A[d] := x
    H.min := NIL
    for d: 0 → D(H.n):
        if A[d] = NIL: continue
        create/append to root list, update H.min
```

ensure roots have unique degree, by merging those with the same degree

build new Heap from array A of degree.

FIB_HEAP_LINK(H, y, x)

remove y from root list

make y a child of x, increase x.degree

y.mark := FALSE

- The change in potential is defined as:

$$\Delta\phi = \phi(H') - \phi(H)$$ H': heap after extract-min

$$= t(H') + 2m(H') - t(H) - 2m(H)$$

$$= (D(n) + 1) + 2m(H) - t(H) - 2m(H)$$

($D(n)+1$: unique roots; $2m(H)$: no addition marked node)

$$= D(n) + 1 - t(H) \sim O(D(n) - t(H))$$

- Amortized cost:

$$\hat{c} = c + \Delta\phi$$

$$= O(D(n) + t(H)) + O(D(n) - t(H))$$

$$= O(2D(n)) \sim O(D(n))$$

- DECREASE_KEY (H, x, k)

Pseudo Code:

```
DECREASE_KEY (H, x, k)
    if k > x.key :
        throw Error
    x.key := k
    y := x.parent
    if y ≠ NIL and x.key < y.key
        CUT ( H, x, y)
        CASCADING_CUT ( H, y)
    if x.key < H.min.key :
        H.min = x
```

```
CUT ( H, x, y)
    remove x from child list of y, decrementing y.degree
    add x to the root list of H
    x.parent := NIL
    x.mark := FALSE
```

```
CASCADING_CUT ( H, y)
    z := y.parent
    if z == NIL : return
    if y.mark == FALSE :
        y.mark := TRUE
        return
    CUT ( H, y, z)
    CASCADING_CUT ( H, z)
```

base case

- The actual cost: CUT takes only O(1), but it is a recursive func that goes up the tree and cut any node that loses its second child.
So: O(c), where c is the number of calls to the CASCADING_CUT func.

- The potential change:

$$\begin{aligned}\Delta\phi &= \phi(H') - \phi(H) \\ &= t(H') + 2m(H') - t(H) - 2m(H) \\ &= \big(t(H) + c\big) + 2\big(m(H) - (c-2)\big) - t(H) - 2m(H)\end{aligned}$$

because c new trees created

because c cuts means (c-1) cascadings

$\Rightarrow$ (c-1) unmarks

- 1 potential mark at the end

$\Rightarrow$ c - 1 - 1 = c - 2

$$= 4 - c$$

- Amortized cost:

$$\begin{aligned}\hat{c} &= c + \Delta\phi \\ &= O(c) + 4 - c \\ &= O(1)\end{aligned}$$

Bounding maximum degree D(n)

In previous section, we established that EXTRACT_MIN, and DELETE operations have amortized cost of D(n), maximum degree of any node in Fibonacci heap. Lets now prove that $D(n) \leq O(\log n)$

Lemma 19.1 degree lower bound

Let x be any node in a Fibonacci heap, and suppose that $x.degree = k$. Let $y_1, y_2, \dots, y_k$ denote the children of x in the order in which they were linked to x, from earliest to latest. Then, $y_1.degree \geq 0$ and $y_i.degree \geq i-2$ for $i = 2, 3, \dots, k$

Proof: By induction

- Base case: attach y_1 to x

  Start with a single node x, if we attach y_1 to x (through CONSOLIDATE), then $y_1.degree \geq 0$

- Inductive step: attach y_i to x for $i = 2, 3, \dots, k$

  We know that when y_i attach to x, all previous $y_1, y_2, \dots y_{i-1}$ are already attached to x, and degrees must equals to attach, so:

$$y_i.degree \geq i-1$$

But, we also know that a node can lose at most one child before it becomes a root (if it lose 2 children, it gets cut from its parent), hence:

$$y_i.\text{degree} \geqslant i - 2$$

Lemma 19.2 different expression Fibonacci number

Fibonacci numbers are defined by the recurrence:

$$F_k = \begin{cases} 0 & \text{if } k = 0 \\ 1 & \text{if } k = 1 \\ F_{k-1} + F_{k-2} & \text{if } k \geqslant 2 \end{cases}$$

But it can also be written as:

$$F_{k+2} = 1 + \sum_{i=0}^{k} F_i$$

Proof: By induction

- Base case: $k = 0$

$$F_2 = 1 + \sum_{i=0}^{0} F_i$$
$$= 1 + F_0$$

- Inductive step: Assume $F_{k+1} = 1 + \sum_{i=0}^{k-1} F_i$, and we know that:

$$F_{k+2} = F_{k+1} + F_k$$
$$= 1 + \sum_{i=0}^{k-1} + F_k$$
$$= 1 + \sum_{i=0}^{k} F_i$$

Lemma 19.3 Fibonacci number lower bound

For all integers $k \geqslant 0$, the $(k+2)^{nd}$ Fibonacci number satisfies $F_{k+2} \geqslant \phi^k$, where ϕ is the golden ratio, with value:

$$\phi = (1 + \sqrt{5})/2 = 1.61803\ldots$$

Proof: By induction

- Base case: $k = 0$

$$F_{0+2} = 1 + F_0$$
$$= 1 + 0 = \phi^0$$
$$= 1$$

$k = 1$

$$F_{1+2} = 1 + F_0 + F_1$$
$$= 2 \geqslant \phi^1$$
$$= 1.61803\ldots$$

Inductive step: $k \geq 2$, Assume $F_i \geq \phi^{i-2}$ for $i = 0, 1, 2, \dots, k-1$

$$\begin{aligned} F_{k+2} &= F_{k+1} + F_k \\ &\geq \phi^{k-1} + \phi^{k-2} \\ &= \phi^{k-2}(\phi + 1) \\ &= \phi^{k-2} \cdot \phi^2 \\ &= \phi^k \end{aligned}$$

Lemma 19.4 Fibonacci heap node size lower bound

Let x be any node in a Fibonacci heap, and let $k = x.\text{degree}$

Then:

$$\text{size}(x) \geq F_{k+2} \geq \phi^k$$

where $\phi = (1+\sqrt{5})/2$

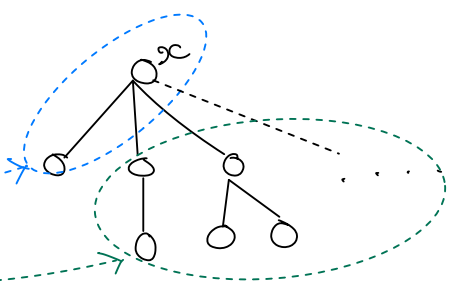
Proof:

- Let s_k denotes minimum size of any node of degree k

$$\Rightarrow \begin{cases} s_0 = 1 \\ s_1 = 2 \end{cases}$$

node x_k is at most $\text{size}(x)$ since adding more node into x cannot decrease its size:

$$\begin{aligned} \text{size}(x) &\geq s_k \\ &\geq 2 + \sum_{i=0}^{k} s_{y_i.\text{degree}} \\ &\geq 2 + \sum_{i=0}^{k} s_{i-2} \quad \langle \text{lemma 19.1} \rangle \end{aligned}$$



- Prove $s_k \geq F_{k+2}$ by induction:
  - Base case: $k=0$ and $k=1$
    - $s_0 = 1 \geq F_2 = 1$
    - $s_1 = 2 \geq F_3 = \phi$
  - Inductive step: $k \geq 2$, assume $s_i \geq F_{i+2}$ for $i = 0, 1, \dots k-1$

$$\begin{aligned} s_k &\geq 2 + \sum_{i=0}^{k} s_{i-2} \\ &\geq 2 + \sum_{i=0}^{k} F_i && \langle \text{hypothesis} \rangle \\ &= 1 + \sum_{i=0}^{k} F_i \\ &= F_{k+2} && \langle \text{lemma 19.2} \rangle \\ &\geq \phi^k && \langle \text{lemma 19.3} \rangle \end{aligned}$$

Corollary 19.5 — Rank upper bound

The maximum degree $D(n)$ of any node in an n-node Fibonacci heap is $O(\lg n)$

Proof:

Let n be number of nodes in Fibonacci heap
x be any node in the same heap

Then by Lemma 19.4, we can say that:

$$n \geq size(x) \geq \phi^k$$

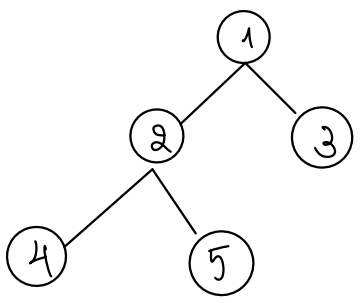
Taking log base ϕ gives us:

$$k \leq \log_\phi n$$

Therefore: the maximum degree $D(n)$ of any node is $O(\lg n)$

Reasoning through Fibonacci Heaps

We know ab Binary Heap



with 2 main properties:

1. Every level is full
2. Insert from left → right, top → bottom

And the analysis of its operations:

- GET_MIN : $O(1)$
- INSERT : $O(\lg n)$
- DECREASE_KEY : $O(\lg n)$
- EXTRACT_MIN : $O(\lg n)$ → important operation

Can easily see that in order to EXTRACT_MIN, we need to maintain this structure that is "costly" for other operations. Can we do better?

↳ Yes, with Fibonacci Heap

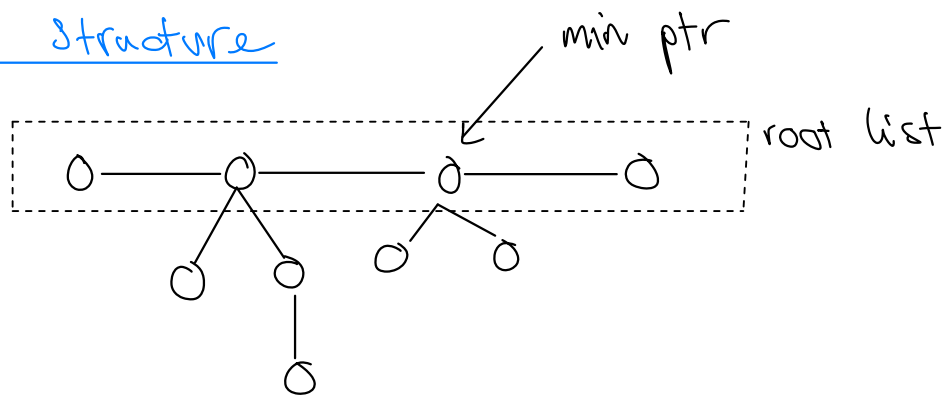
Main idea of Fibonacci Heap is: "If we can somehow be lazy in all other operations and only do the work when it matters, i.e EXTRACT_MIN, then using Amortized analysis, we can argue that we have a data structure with constant time operations"

So, if:

GET_MIN	$O(1)$
INSERT	$O(1)$
DECREASE_KEY	$O(1)$
EXTRACT_MIN	$O(\lg n)$

} ⇒ Constant amortized cost

Overall Structure



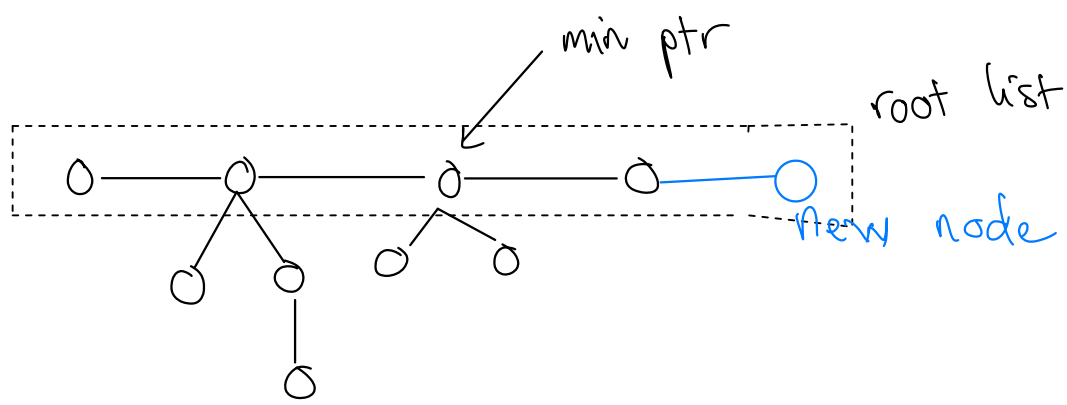
Maintain only 1 property: "Children must be larger than parent"

GET_MIN : O(1)

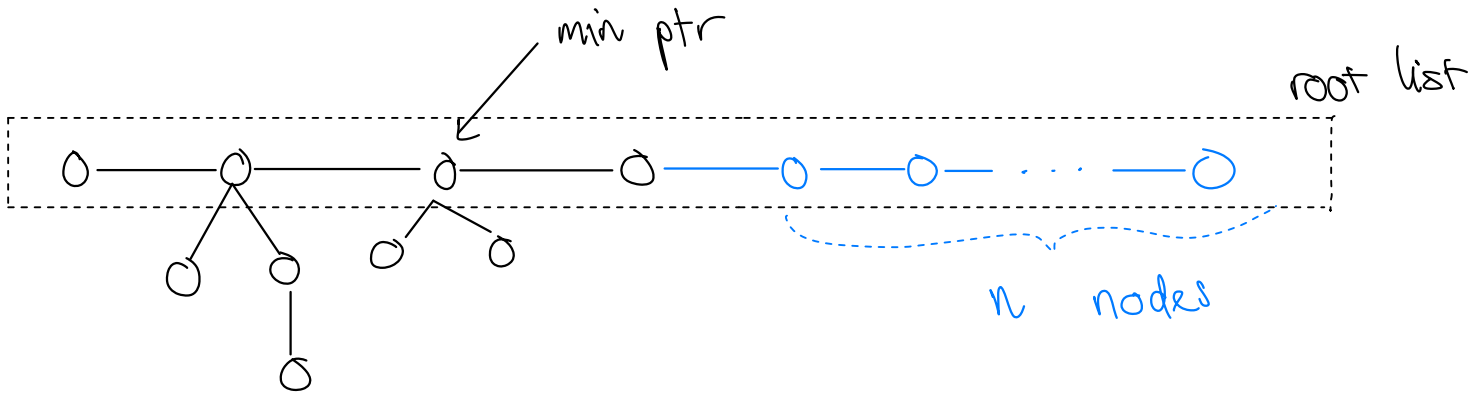
Because we maintain a "min. ptr"

INSERT : O(1)

"A new node is a new tree / root"



"If we insert n new nodes, then we get n new trees"



EXTRACT_MIN : O(lg n)

This is where we "pay" for all that n nodes we INSERT

↳ Solution here is to somehow "merge" all n nodes into 1 new tree

An abstract example of a series of INSERT(s) and EXTRACT_MIN(s)

Operations	Costs	Amortized
INSERT	1	2
INSERT	1	2
⋮ (100 operations)	⋮	⋮
INSERT	1	2
EXTRACT_MIN	100	10
EXTRACT_MIN	15	15
EXTRACT_MIN	15	15
⋮	⋮	⋮

⇢ 1st EXTRACT_MIN operation "pay" for previous INSERT s.t the subsequent EXTRACT_MIN are constant

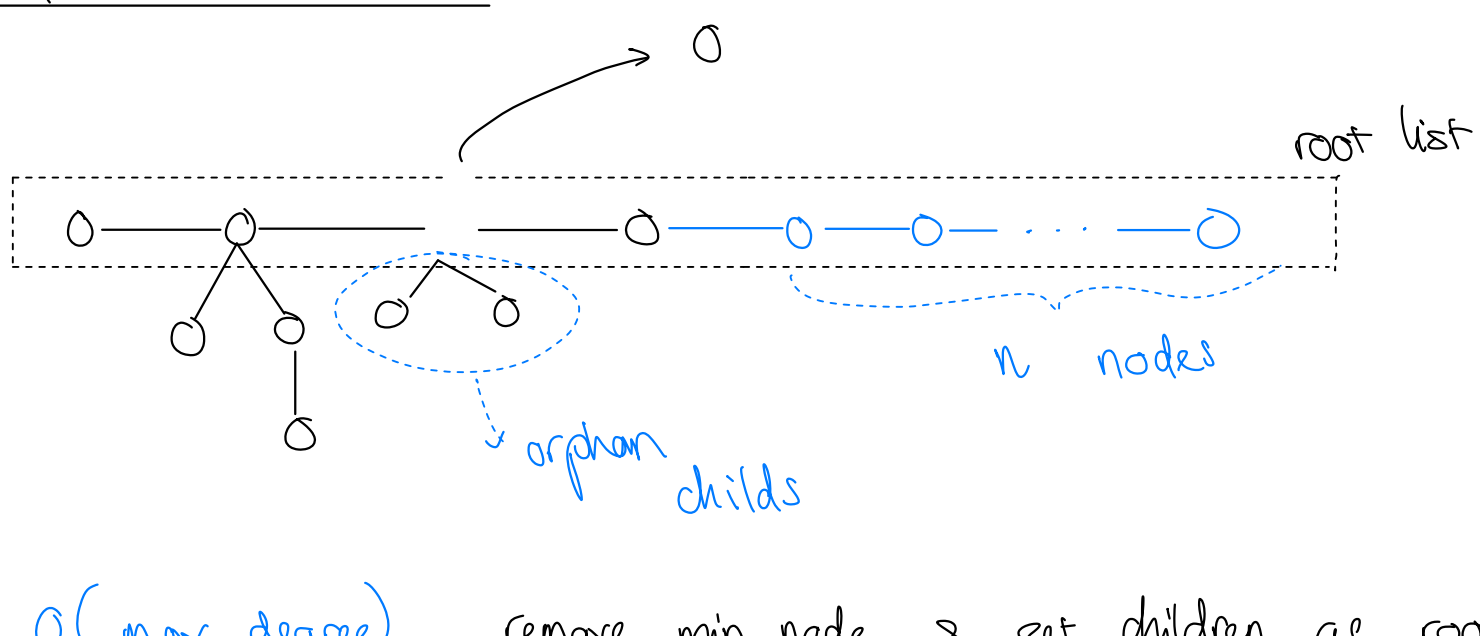
⇒ We need some EXTRA_WORK inside EXTRACT_MIN to allow the first EXTRACT_MIN to "pay" for previous INSERT(s)

⇒ Amortized for EXTRACT_MIN = O(# root after + EXTRA_WORK)

↳ The goal is that: "If we can come up a EXTRA_WORK bound that is reasonably fast, then EXTRACT_MIN will be fast"

EXTRACT_MIN procedure

1. Remove min node

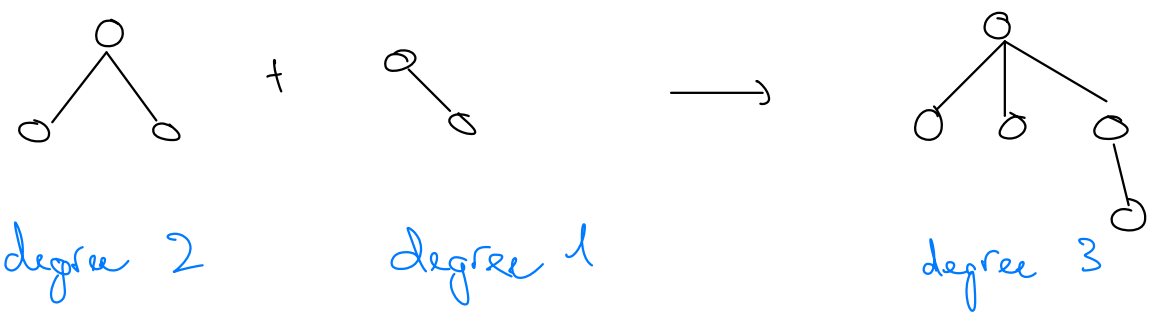


⇒ O(max degree) remove min node & set children as roots

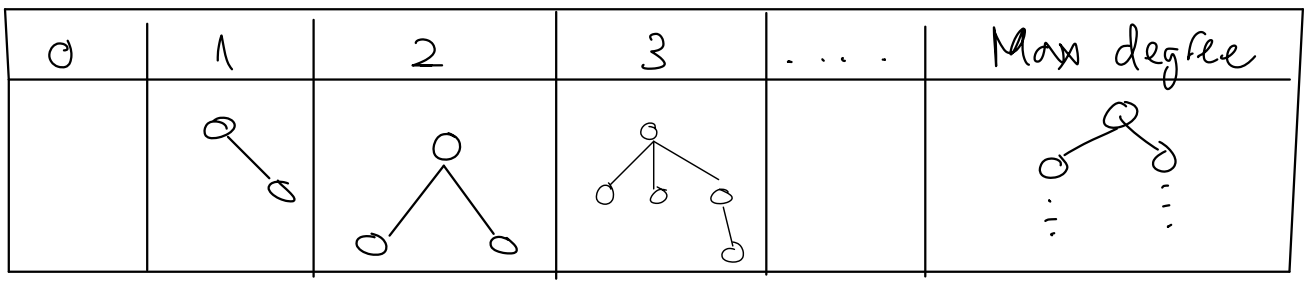
2. CLEAN_UP

"Merge trees with the same degree until the root list only contains unique degree"

Merge:-



Store in intermediate array:-



⇒ O(max degree + # roots) : for each node in root list, "slide" it through intermidiate array size Max_degree until find an empty spot.

3. Rebuild heap from intermediate array

Iterate through the array and build new heap

⇒ O(max degree)

Conclusion: EXTRACT_MIN cost is $O(max\_degree + \#roots)$

Recall from the "abstract example", our amortized cost analyzed is $O(\#roots + EXTRA\_WORK)$

$\Rightarrow O(Max\ degree) = O(EXTRA\_WORK)$

↳ Let's bound Max degree (to show that EXTRACT_MIN is fast)

Informal proof:

Consider binomial trees, which has:

degree $d \rightarrow 2^d$ nodes

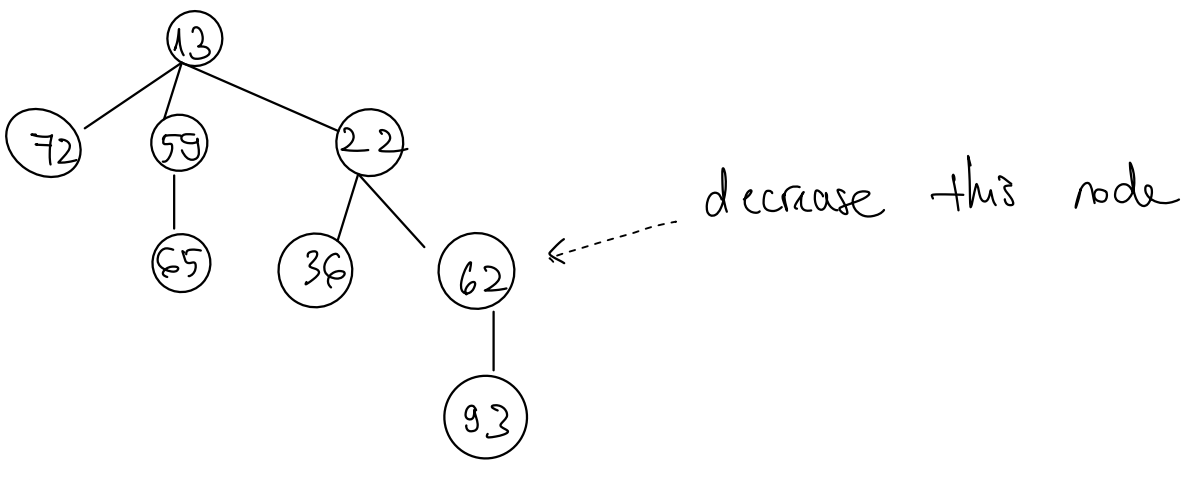
$\Leftrightarrow$ k nodes $\rightarrow$ degree $\log_2(k)$

$\Rightarrow$ Max degree of tree with n nodes is $\log_2(n)$

So far: We have shown

GET_MIN	INSERT	EXTRACT_MIN	DECREASE_KEY
$O(1)$	$O(1)$	$O(\log n)$	?

DECREASE_KEY:



There are 2 scenarios:

- Case 1: $x \geq x.parent$ i.e decrease $62 \rightarrow 30$

  $\Rightarrow$ Do nothing

- Case 2: $x < x.parent$ i.e decrease $62 \rightarrow 19$

  We do a few approaches

  - Approach 1: Bubble up (like Binary heap)

    $\Rightarrow O(\log n)$

  - Approach 2: Cut out $T(x)$ and form a new root

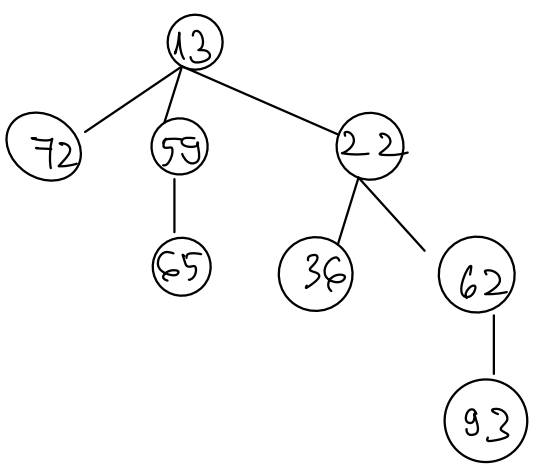
    $\Rightarrow O(1)$

    $\Rightarrow$ The goal is to be "lazy" and let EXTRACT_MIN clean up the mess, and end up with a good amortized cost (?)

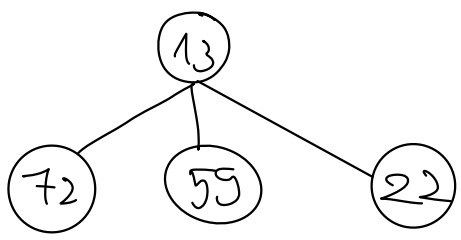
↳ Problem with Approach 2:-

If we only cut out the tree, then it is possible that we end up with a tree that is "high degree, but small # of nodes".

For example:



series of cuts



. binomial tree
. logarithmic degree

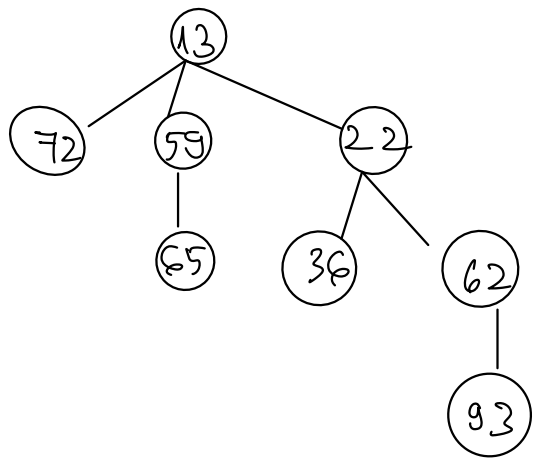
. not binomial tree
. linear degree

And recall that EXTRACT_MIN is bounded by O(max-degree)
⇒ Degree grow linearly ⇒ EXTRACT_MIN cost O(n), which destroy our proof that EXTRACT-MIN is "fast".

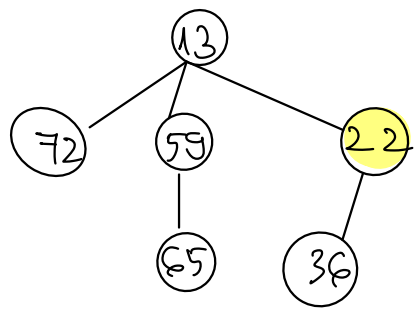
↳ Solution: Cut out at most 1 child per node

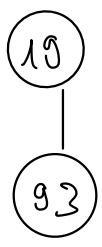
The goal is to come up with a procedure/rule that ensure we don't stray to much from the binomial tree shape
⇒ "If a node has more than one node cut out, that node is also cut out"

Example:

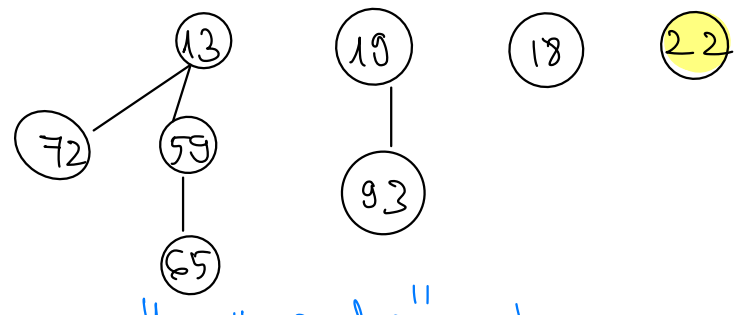


decrease 62 → 19
"mark" 22 that it has lost 1 child





decrease 36 → 18,
cut 22 (since it lost > 1 child)



⇒ Ensure there is no "large degree, small # nodes" tree

Proof : DECREASE_KEY is O(1)

- Question : If DECREASE_KEY is recursive, meaning we can DECREASE many parents on one call to DECREASE_KEY, how can we claim that DECREASE_KEY costs O(1) ?
- For k DECREASE_KEY called, cut out nodes $\leq 2k$

  Because:
  - $\leq k$ nodes are cut on k DECREASE_KEY
  - $\leq k$ additional "marked" nodes are cut

  $\Rightarrow$ Amortized cost of DECREASE_KEY is O(1)