Priority Queues & Disjoint Union

"Average" analysis of algorithms

(without sacrificing worst-case robustness)

1. Amortized analysis ("average" in hindsight)

2. Randomized algorithms ("average" over random bits)

Both are very practical, different perspectives

Shortest Paths

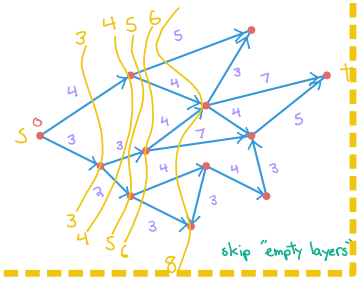
Dijkstra($G = (V, E), \ell : E \to \mathbb{R}_{>0}, s \in V$)

/ An implementation of Dijkstra's algorithm where a priority queue keeps track of the unlabeled vertex of minimum tentative distance. */*

1. Set tentative(s) = 0 and tentative(v) = $+\infty$ for all $v \neq s$.
2. Let Q be a priority queue / heap over V with priority decreasing in tentative(v).
3. While Q is not empty
   A. Remove the vertex v of minimum tentative(v) from Q.
   B. Set distance(v) = tentative(v).
   C. For each outgoing edge (v, w):
      1. Set

$$\text{tentative}(w) = \min\{\text{tentative}(w), \text{distance}(v) + \ell(v, w)\}$$

      and decrease w's priority in Q accordingly.
4. Return the distance labels.



Minimum Spanning Tree

search-MST($G = (V, E), w : E \to \mathbb{R}$)

/* *Also known as Prim's algorithm.* */

1. Set $T \leftarrow \emptyset$ and $S \leftarrow \{v\}$ for an arbitrary vertex v.

/* *We maintain the invariant that S is the connected component of v in T.* */

2. While T is not a spanning tree:

   /* *We can accelerate the following steps with a Fibonacci heap.* */

   A. Let e be the minimum weight edge in $\partial(S)$.

   B. Set $T \leftarrow T + e$, $S \leftarrow S \cup e$.

3. Return T.

Shortest Paths

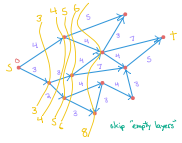
`Dijkstra`$(G=(V,E), \ell : E \to \mathbb{R}_{>0}, s \in V)$

/ An implementation of Dijkstra's algorithm where a priority queue keeps track of the unlabeled vertex of minimum tentative distance. */*

1. Set `tentative`$(s) = 0$ and `tentative`$(v) = +\infty$ for all $v \neq s$.
2. Let Q be a priority queue / heap over V with priority decreasing in `tentative`(v).
3. While Q is not empty
   A. Remove the vertex v of minimum `tentative`(v) from Q.
   B. Set `distance`(v) = `tentative`(v).
   C. For each outgoing edge (v,w):
      1. Set

         `tentative`$(w) = \min\{$`tentative`(w), `distance`$(v) + \ell(v,w)\}$

         and decrease w's priority in Q accordingly.



4. Return the distance labels.

Minimum Spanning Tree

`search-MST`$(G=(V,E), w : E \to \mathbb{R})$

/ Also known as Prim's algorithm. */*

1. Set $T \leftarrow \emptyset$ and $S \leftarrow \{v\}$ for an arbitrary vertex v.

/ We maintain the invariant that S is the connected component of v in T. */*

2. While T is not a spanning tree:

   / We can accelerate the following steps with a Fibonacci heap. */*

   A. Let e be the minimum weight edge in $\partial(S)$.
   B. Set $T \leftarrow T + e$, $S \leftarrow S \cup e$.

3. Return T.

Fibonacci Heaps

AMORTIZED

1. `insert`(k,p): inserts an key k with priority p. — O(1)
2. `decrease`(k,p): Let k be a key in the heap. If p is less than the current priority of k, decrease the priority to p. — O(1)
3. `remove-min`$()$: removes and returns the key value pair (k,p) with the minimum priority p. — O(log n)

Fibonacci Heaps

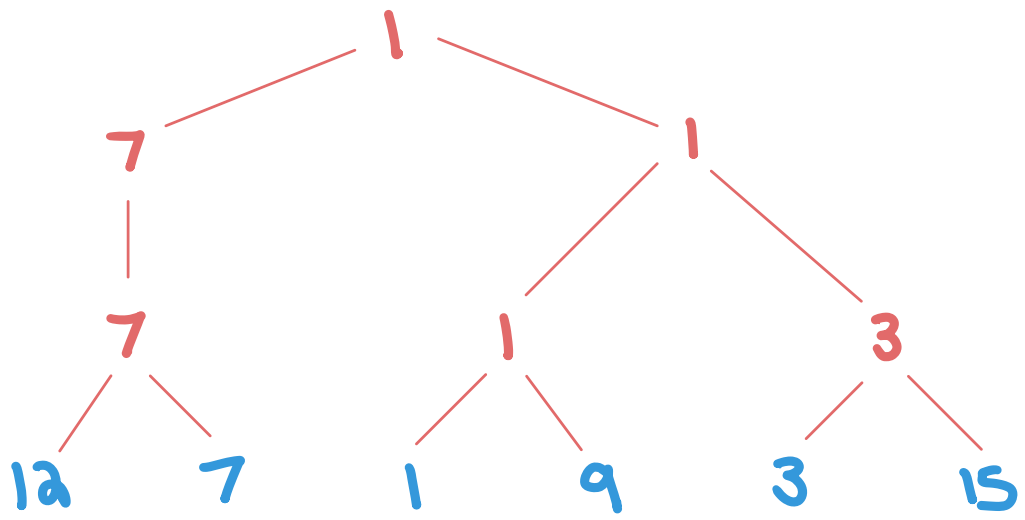
AMORTIZED

1. `insert`(k, p): inserts an key k with priority p. — O(1)
2. `decrease`(k, p): Let k be a key in the heap. If p is less than the current priority of k, decrease the priority to p. — O(1)
3. `remove-min()`: removes and returns the key value pair (k, p) with the minimum priority p. — O(log n)

Ideas?

We will follow a simpler construction...

Tournament tree



(internal)
Nodes
- store min over subtree
- have 1 or 2 children

Leaves:
- store elements
- same depth

Operations: link & cut
link: trees together
cut: subtree

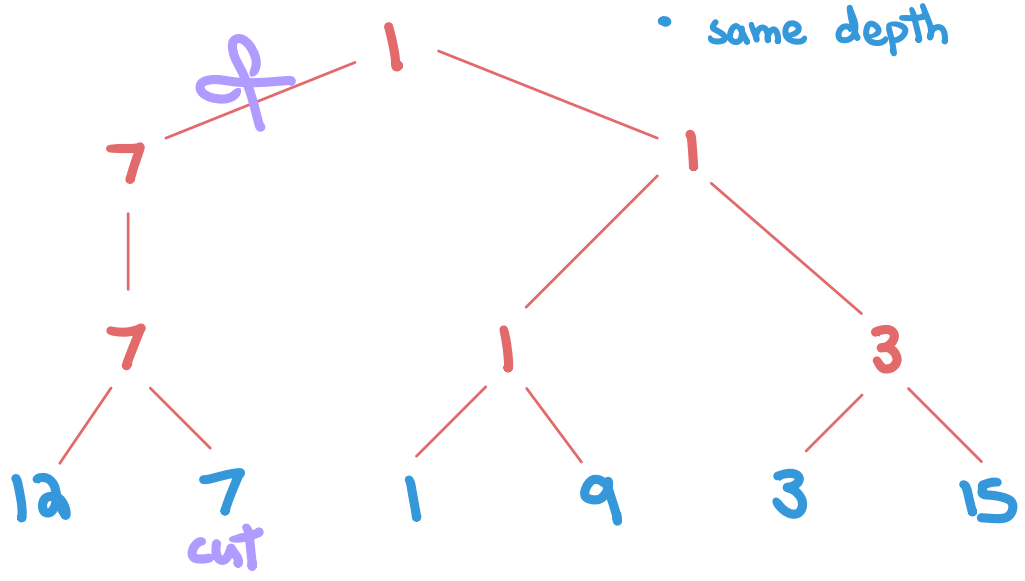
Tournament tree

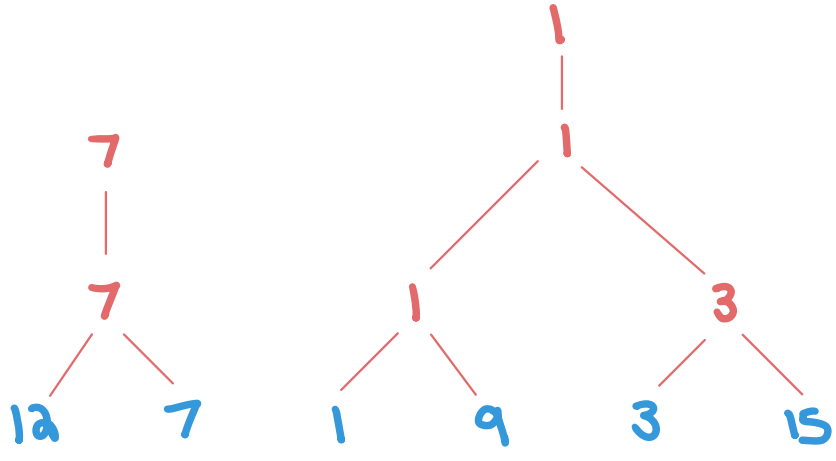
(internal) Nodes
- store min over subtree
- have 1 or 2 children

Leaves:
- store elements
- same depth

Cut

For "top node" of element delete parent edge

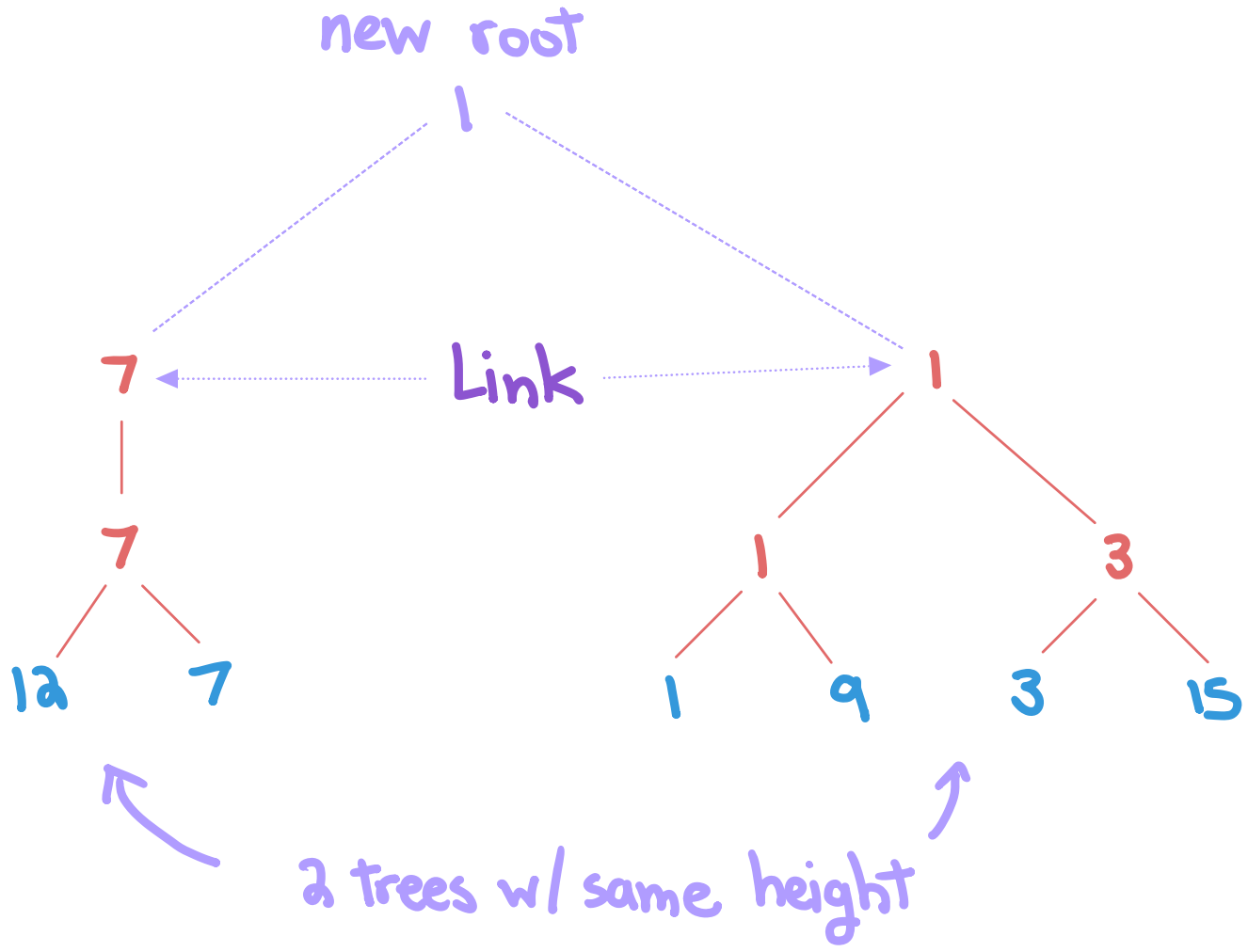




Tournament tree

(internal) Nodes
- store min over subtree
- have 1 or 2 children

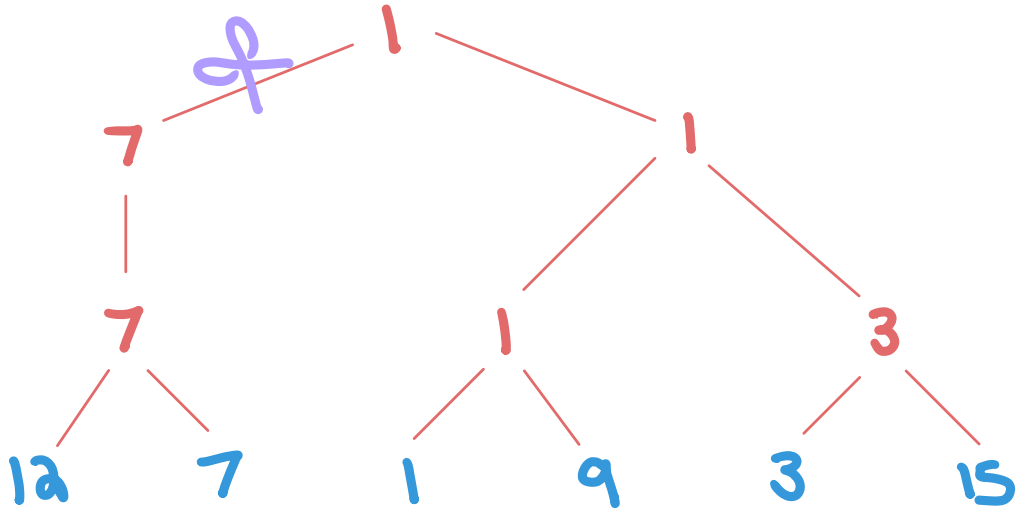
Leaves:
- store elements
- same depth



Tournament tree

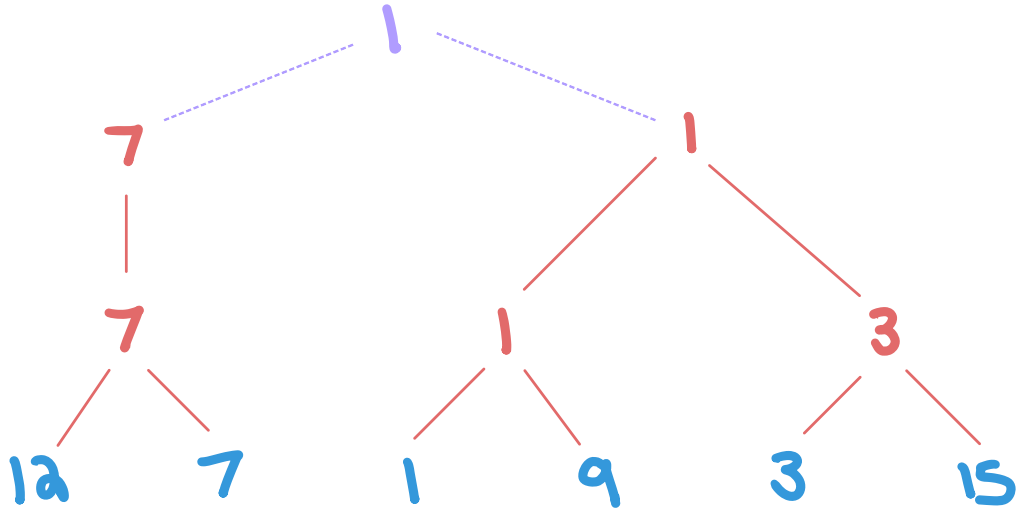
(internal) Nodes
- store min over subtree
- have 1 or 2 children

Leaves:
- store elements
- same depth



Cut

For "top node" of element
delete parent edge



Link

2 trees w/ same height
combine w/ new node

"Heap" = Collection of tournament trees:

Rules:

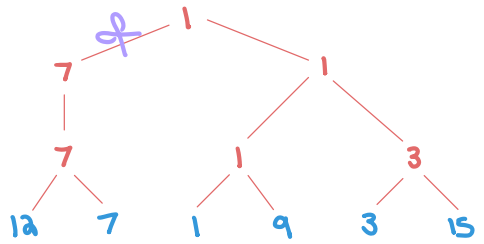
1. All tree heights distinct
2. Max height $O(\log n)$

(internal) Nodes

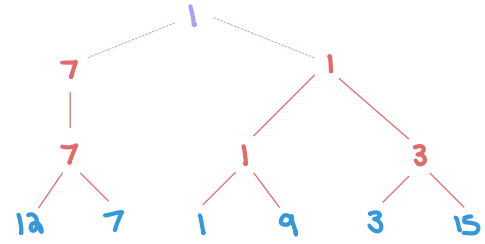
- store min over subtree
- have 1 or 2 children

Leaves

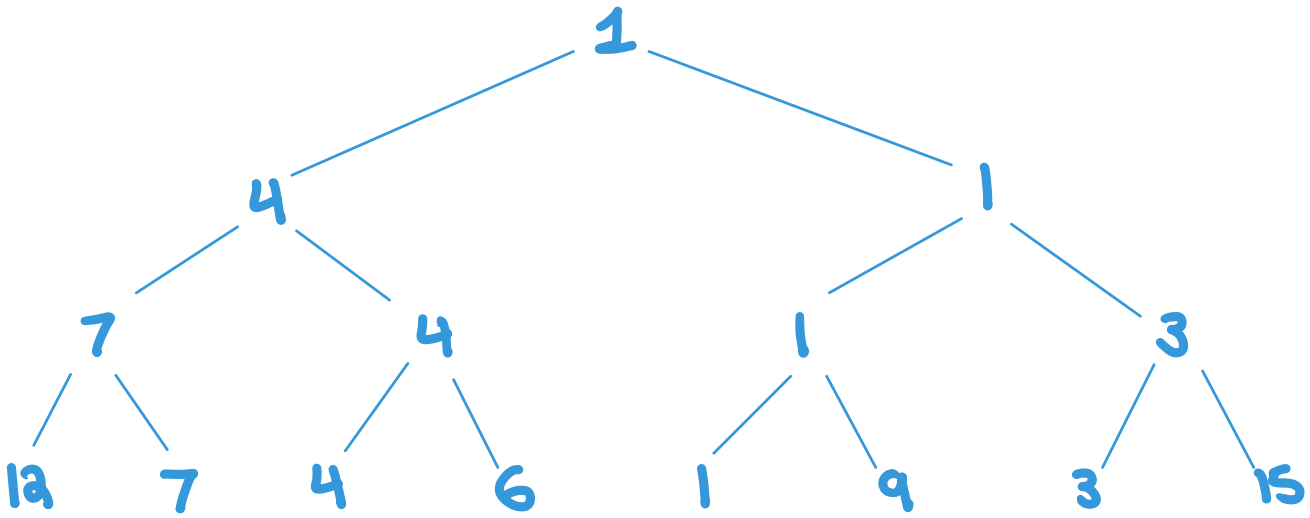
- store elements
- same depth

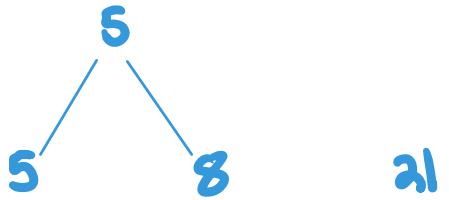


Cut

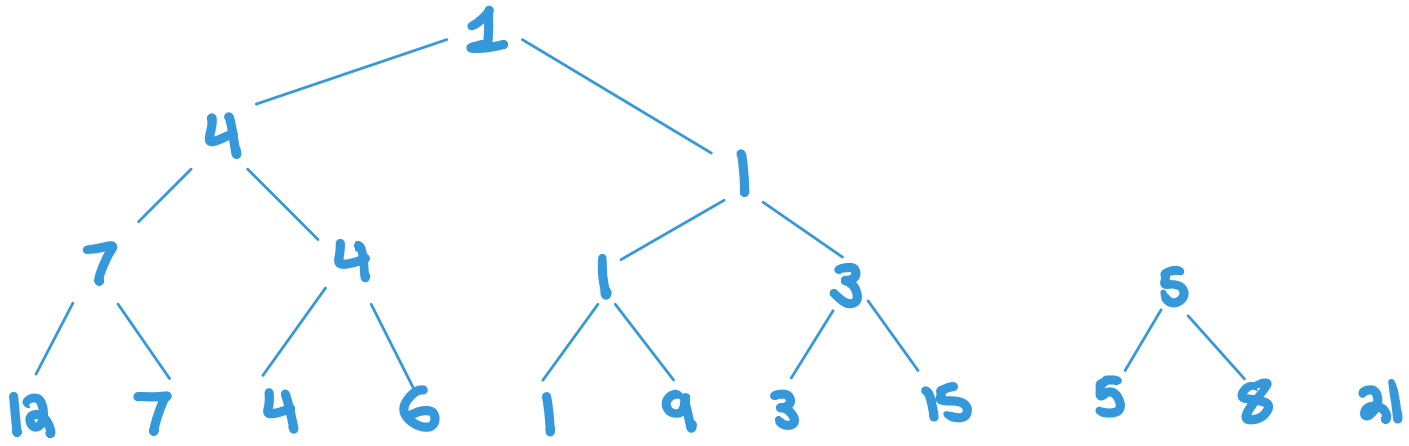


Link





"Heap" = Collection of tournament trees:



Rules:

① All tree heights distinct

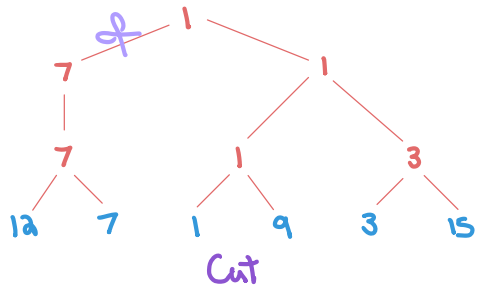
② Max height $O(\log n)$

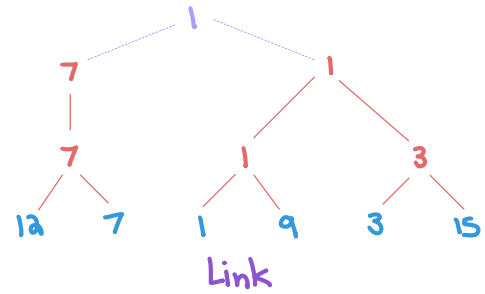
(internal) Nodes

- store min over subtree
- have 1 or 2 children

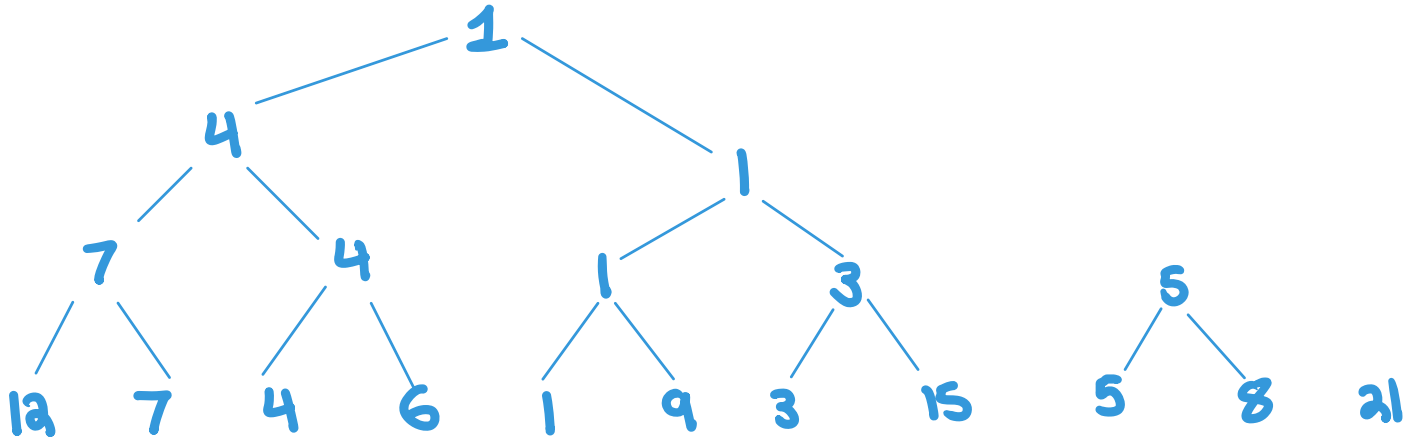
Leaves

- store elements
- same depth



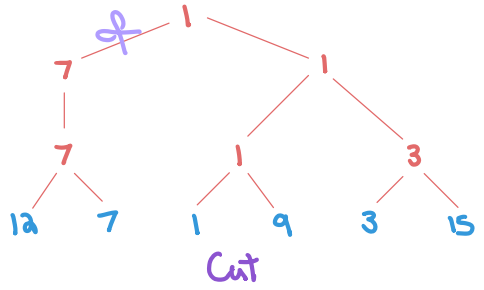


"Heap" = Collection of tournament trees:

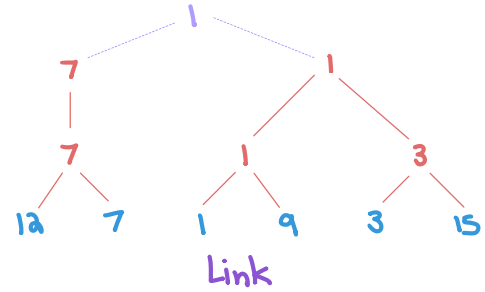


(internal) Nodes
- store min over subtree
- have 1 or 2 children

Leaves
- store elements
- same depth



Cut



Link

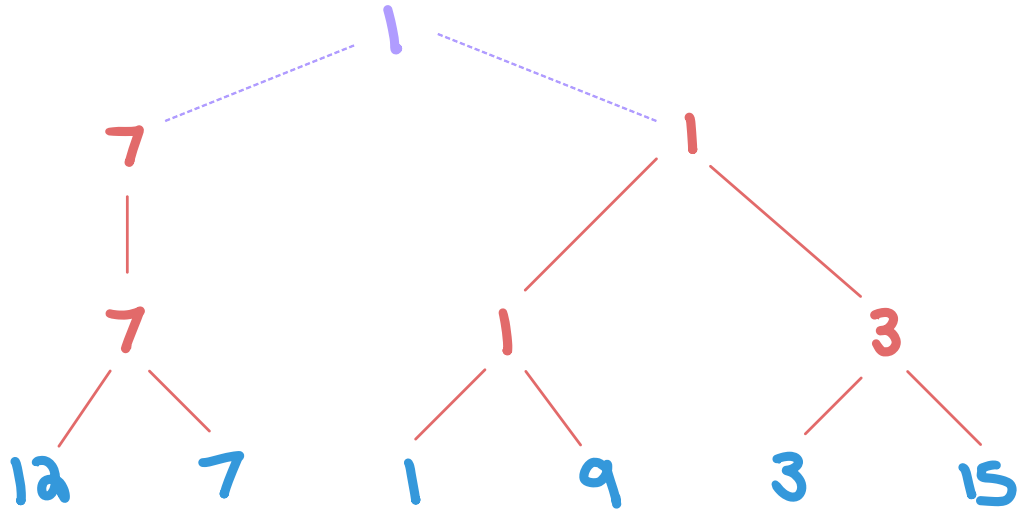
Rules:
1. All tree heights distinct
2. Max height $O(\log n)$

Maintaining:

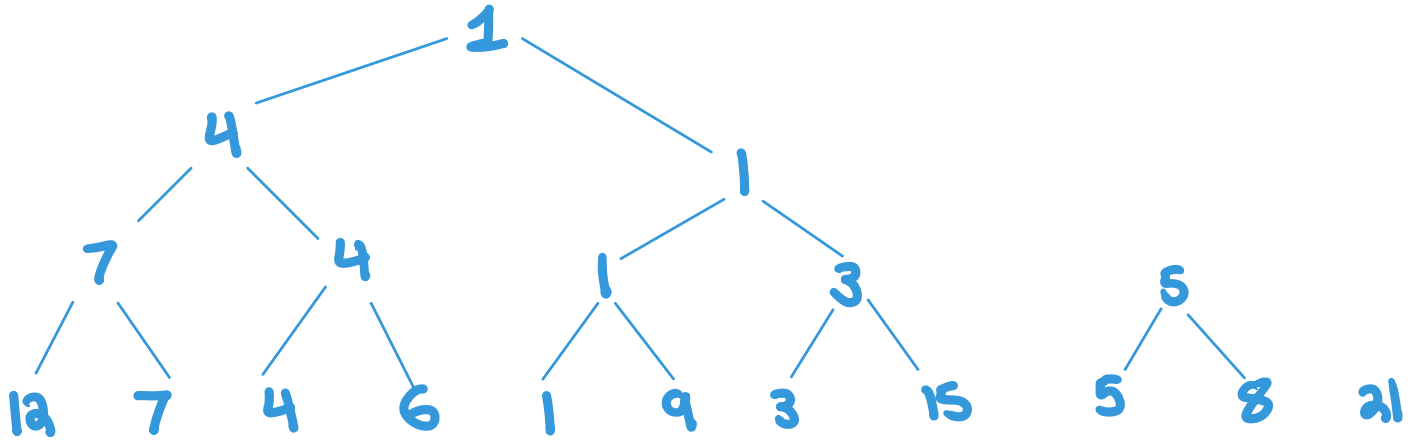
If 2 trees w/ same height

⇓

link!

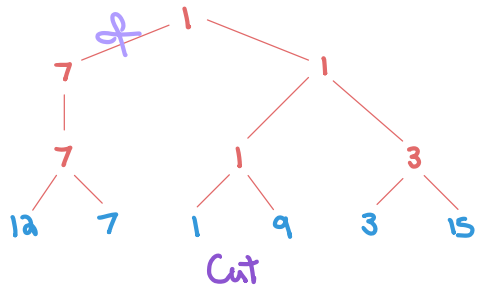


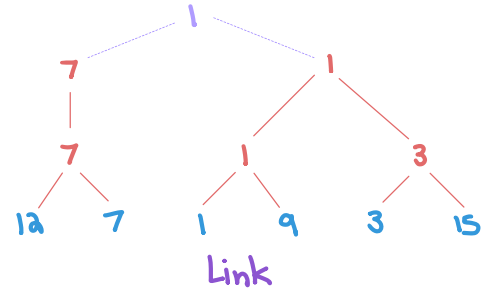
"Heap" = Collection of tournament trees:



(internal) Nodes
- store min over subtree
- have 1 or 2 children

Leaves
- store elements
- same depth





Rules: ① All tree heights distinct

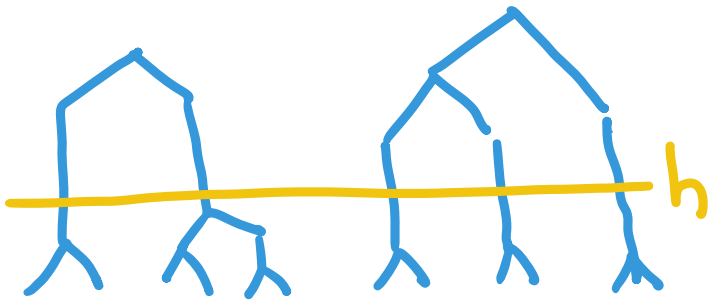
② Max height $O(\log n)$

Maintaining

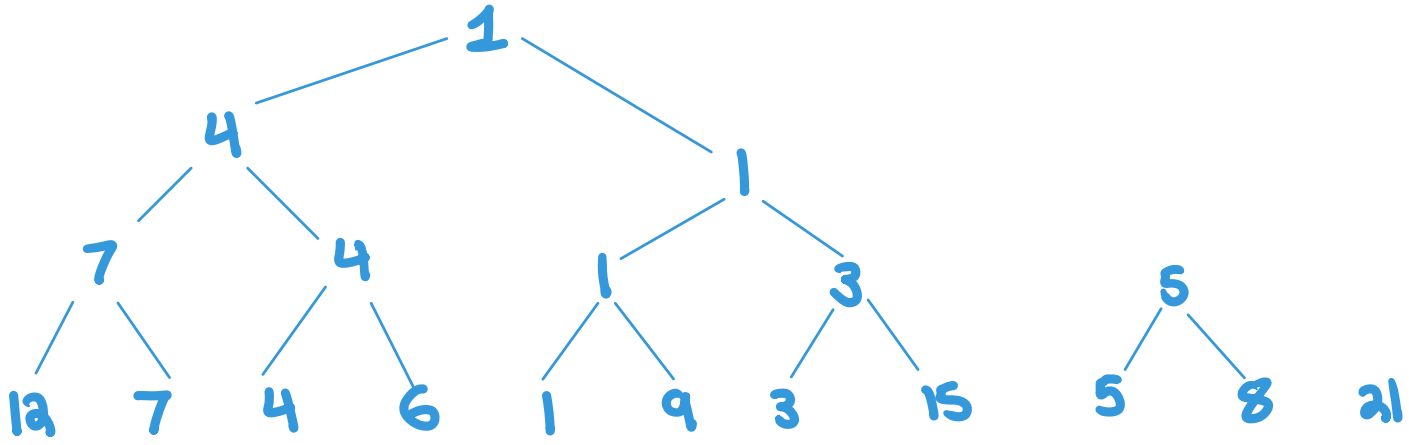
Risk: too many unary nodes

If $\left(\begin{smallmatrix}\#\text{ unary nodes}\\ \text{at height } h\end{smallmatrix}\right) \geq \frac{1}{2}\left(\begin{smallmatrix}\#\text{ nodes}\\ \text{at height } h\end{smallmatrix}\right)$:

delete all nodes w/ height $\geq h$

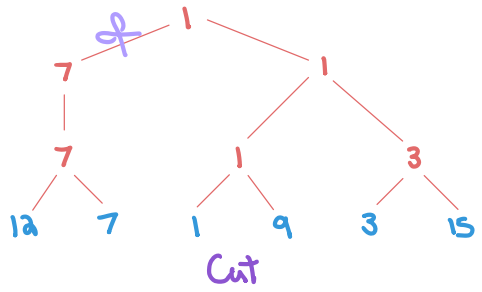


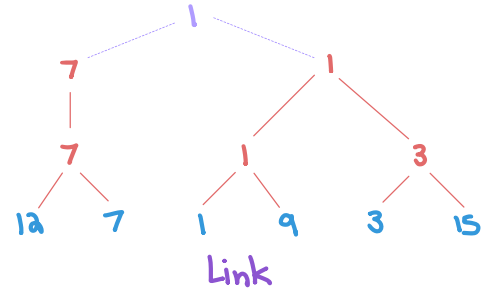
"Heap" = Collection of tournament trees:



(internal) Nodes
- store min over subtree
- have 1 or 2 children

Leaves
- store elements
- same depth





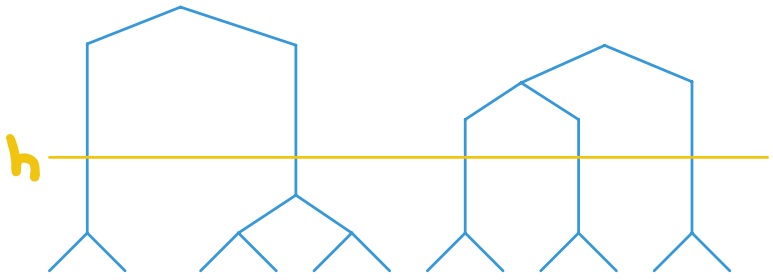
Rules: ① All tree heights distinct
② Max height $O(\log n)$

Maintaining

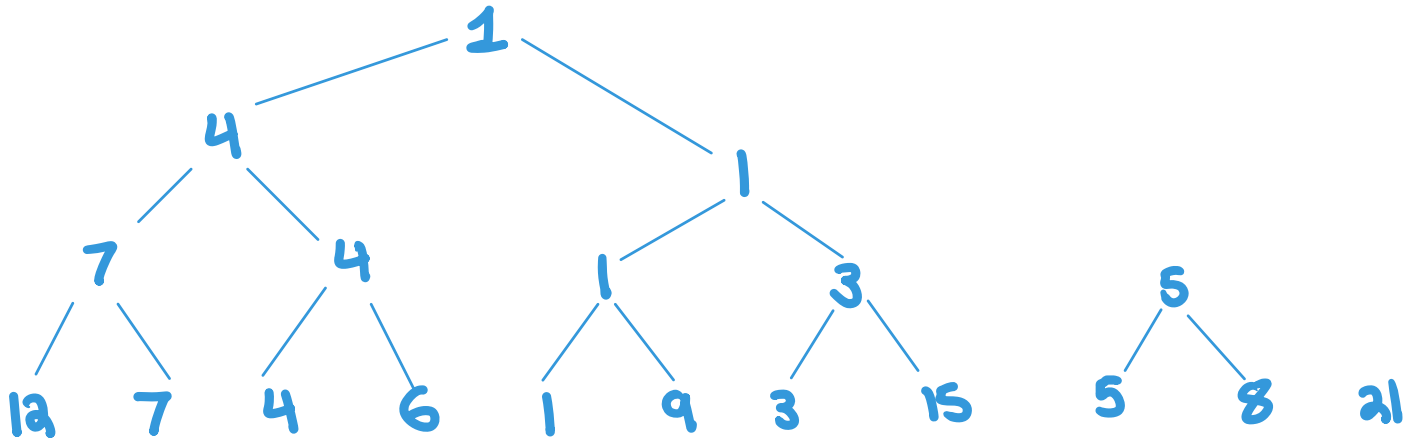
Risk: many unary nodes

If $\left(\begin{smallmatrix}\#\text{ unary nodes}\\ \text{at height } h\end{smallmatrix}\right) \geq \frac{1}{2}\left(\begin{smallmatrix}\#\text{ nodes}\\ \text{at height } h\end{smallmatrix}\right)$:

delete all nodes w/ height $\geq h$

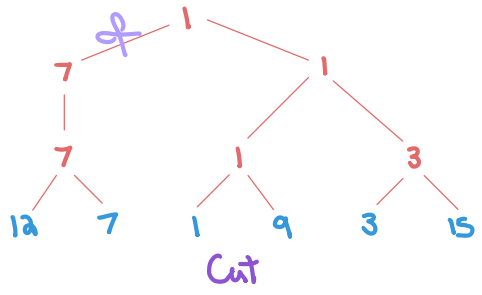


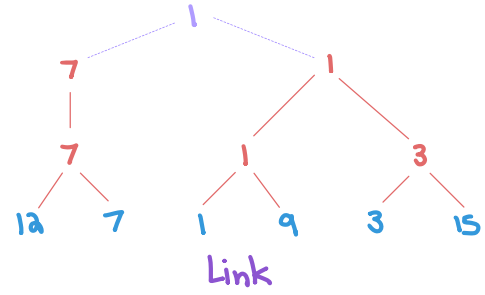
"Heap" = Collection of tournament trees:



(internal) Nodes
- store min over subtree
- have 1 or 2 children

Leaves
- store elements
- same depth





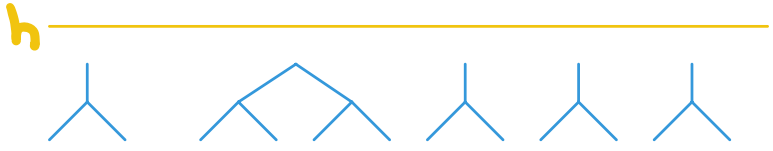
Rules: ① All tree heights distinct
② Max height $O(\log n)$

Maintaining

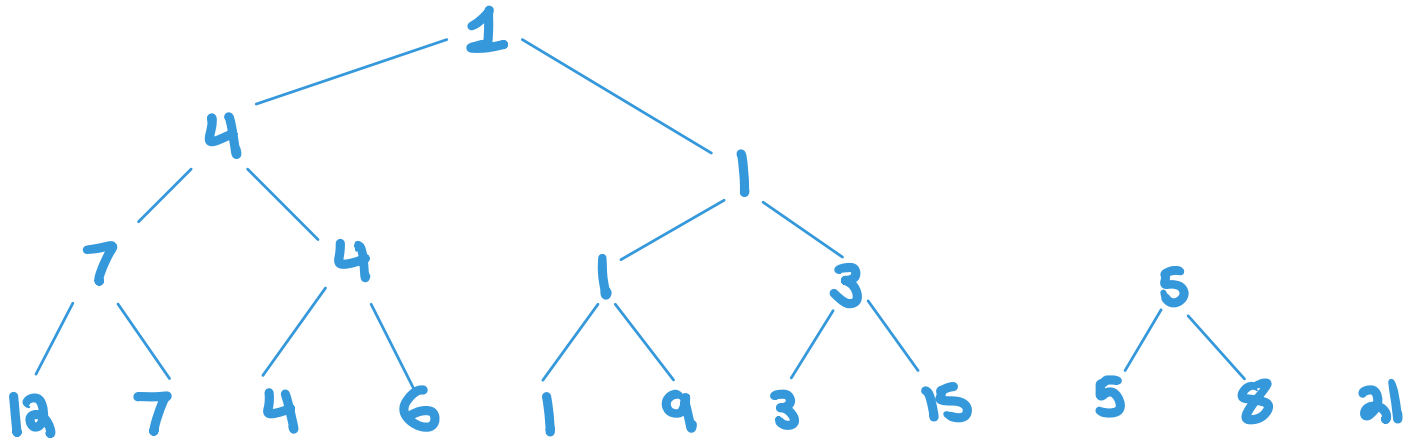
Risk: many unary nodes

If $\left(\begin{smallmatrix}\#\text{ unary nodes}\\ \text{at height } h\end{smallmatrix}\right) \leq \frac{1}{2}\left(\begin{smallmatrix}\#\text{ nodes}\\ \text{at height } h\end{smallmatrix}\right)$

for all h, then max height $= O(\log n)$

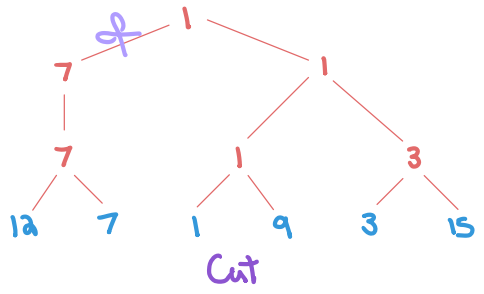


"Heap" = Collection of tournament trees:

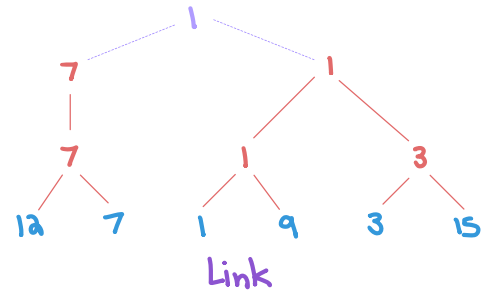


(internal) Nodes
- store min over subtree
- have 1 or 2 children

Leaves
- store elements
- same depth



Cut

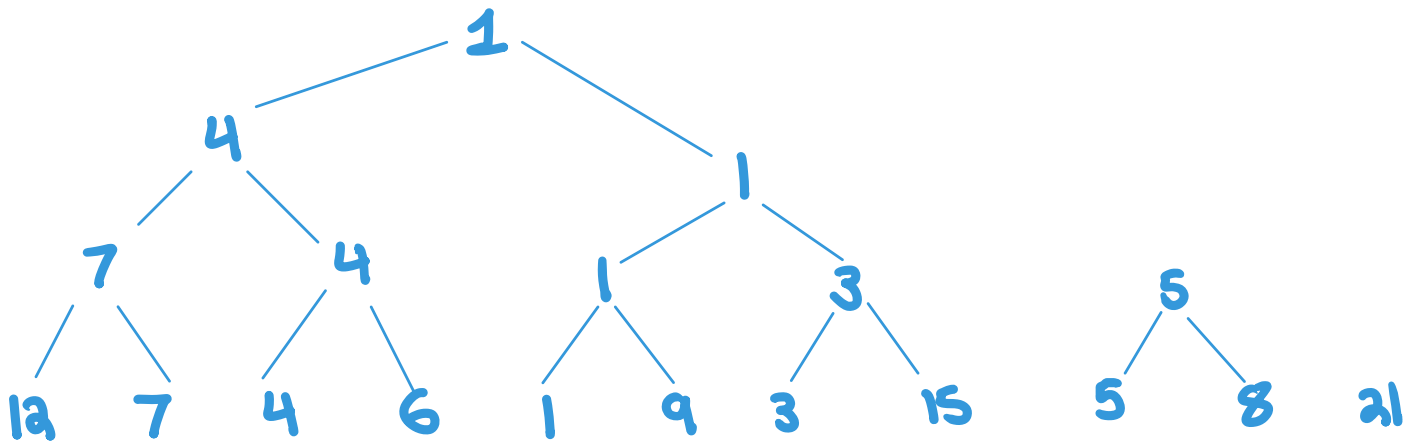


Link

Rules: ① All tree heights distinct
② Max height $O(\log n)$

Paying for housekeeping

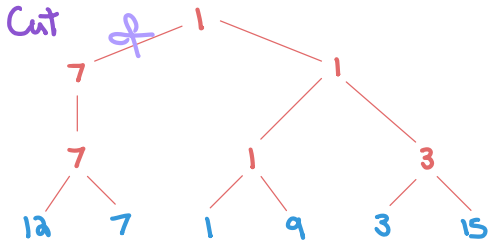
$$\Phi = \underset{\text{constant TBD}}{C_t}(\#\text{ trees}) + \underset{\text{constant TBD}}{C_u}(\#\text{ unary nodes}) + \underset{\text{constant TBD}}{C_n}(\#\text{nodes})$$



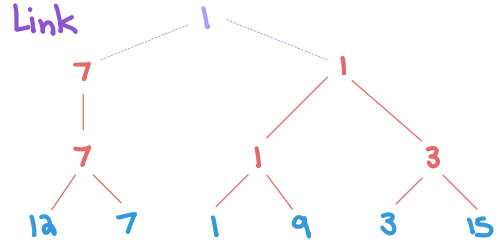
(internal) Nodes
- store min over subtree
- have 1 or 2 children

Leaves
- store elements
- same depth

Cut

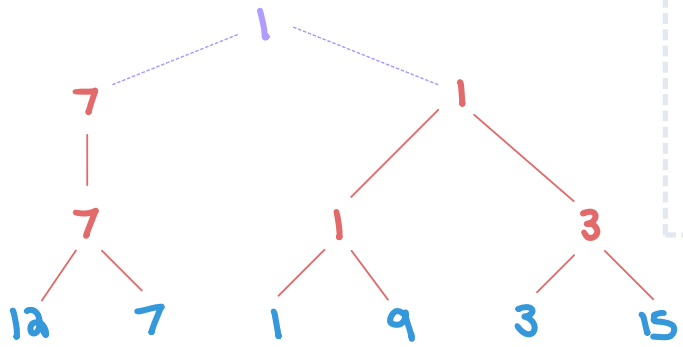


Link



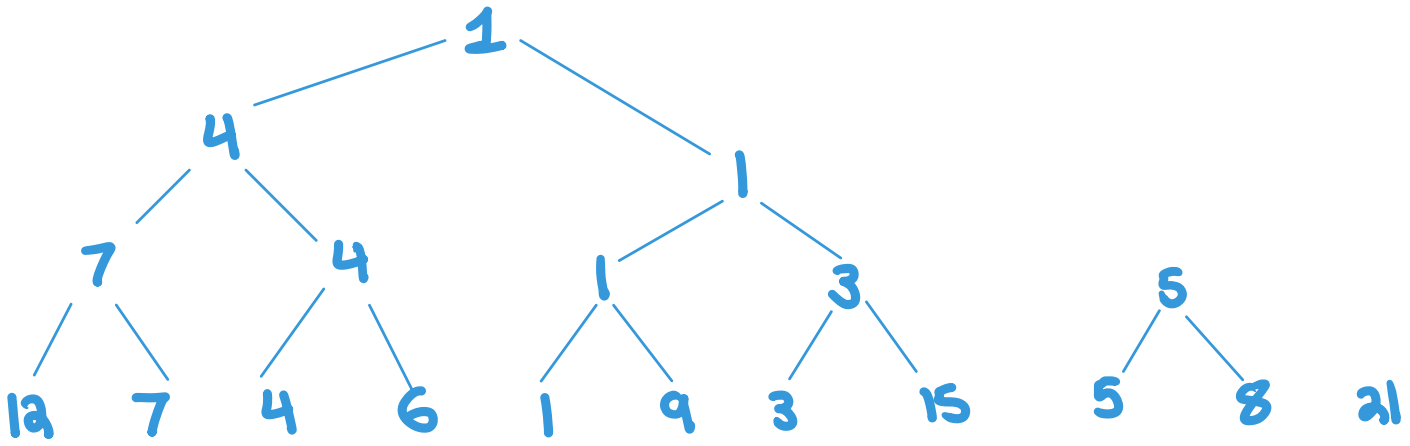
Rules:
① All tree heights distinct
② Max height $O(\log n)$

Maintaining:
2 trees w/ same height
⇒ link!



$$\Phi = \underset{\text{constant TBD}}{C_t}(\#\text{ trees}) + \underset{\text{constant TBD}}{C_u}(\#\text{ unary nodes}) + \underset{\text{constant TBD}}{C_n}(\#\text{nodes})$$

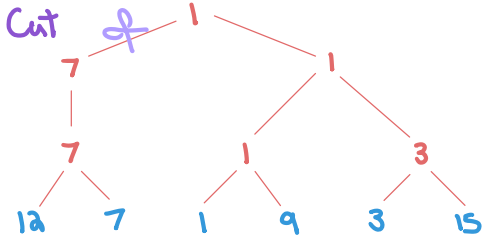
pays for link



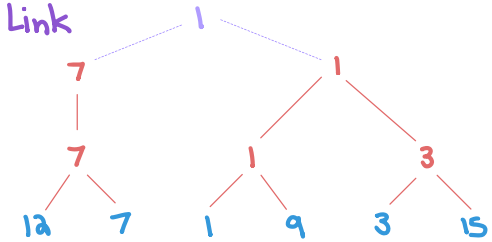
(internal) Nodes
- store min over subtree
- have 1 or 2 children

Leaves
- store elements
- same depth

Cut



Link

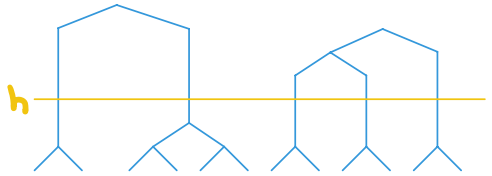


Rules: ① All tree heights distinct

② Max height $O(\log n)$

Maintaining

Risk: many unary nodes



$$\left(\begin{array}{c}\#\text{ unary nodes} \\ \text{at height } h\end{array}\right) \geq \frac{1}{2}\left(\begin{array}{c}\#\text{ nodes} \\ \text{at height } h\end{array}\right)$$

$\Rightarrow$ delete all nodes w/ height $\geq h$

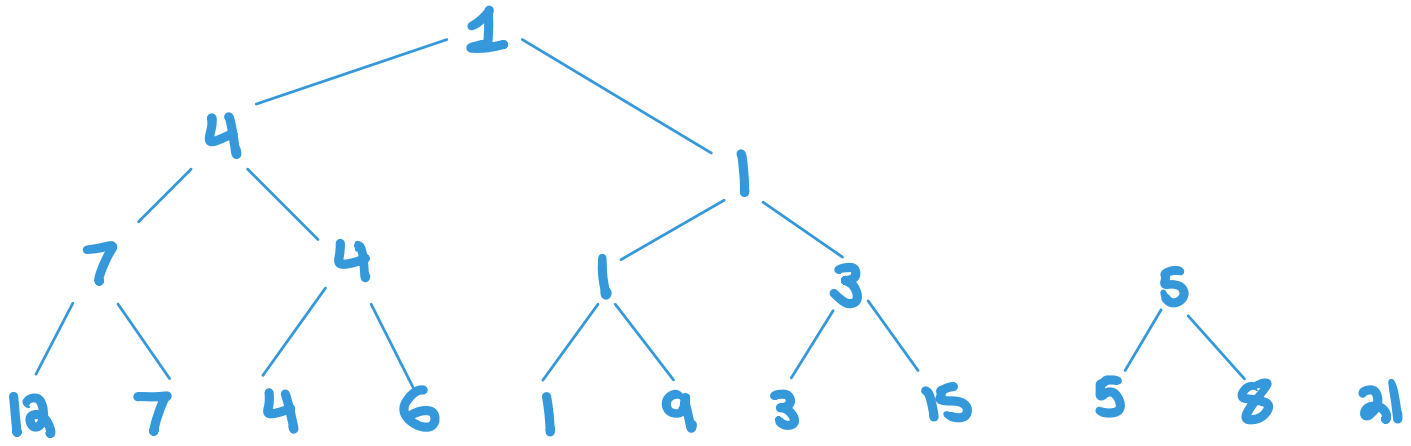
$$\Phi = \underbrace{c_t}_{\text{constant TBD}}(\#\text{ trees}) + \underbrace{c_u}_{\text{constant TBD}}(\#\text{ unary nodes}) + \underbrace{c_n}_{\text{constant TBD}}(\#\text{nodes})$$

pays for link

pays for new trees if $c_u > 2c_t$

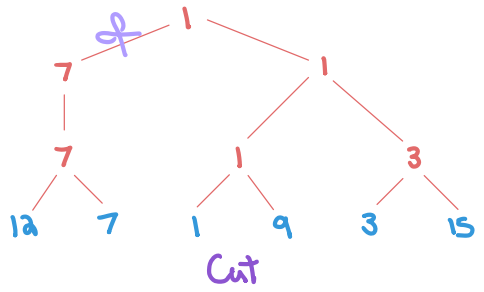
pays for deletion

"Heap" = Collection of tournament trees:

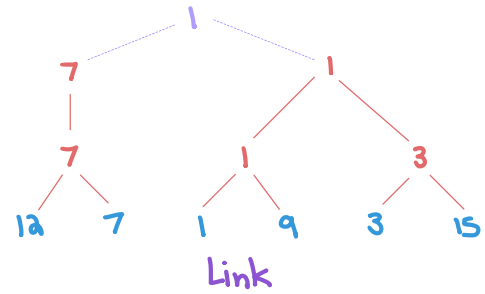


(internal) Nodes
- store min over subtree
- have 1 or 2 children

Leaves
- store elements
- same depth



Cut



Link

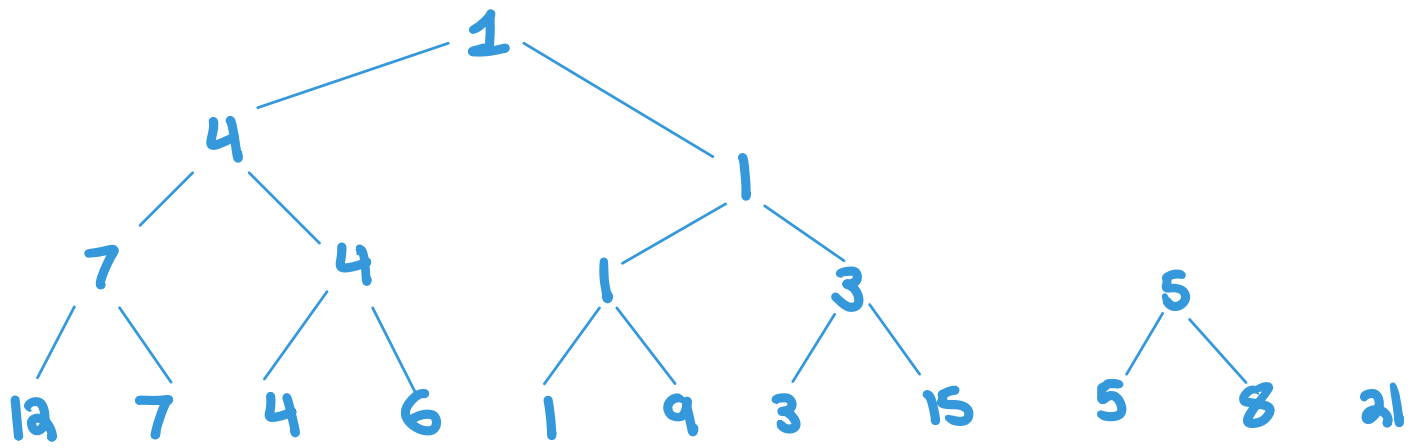
Rules: ① All tree heights distinct
② Max height $O(\log n)$

Paying for housekeeping

$$\Phi = \underbrace{C_t}_{\text{constant TBD}}(\#\text{ trees}) + \underbrace{C_u}_{\text{constant TBD}}(\#\text{ unary nodes}) + \underbrace{C_n}_{\text{constant TBD}}(\#\text{nodes})$$

Φ => "free maintenance"

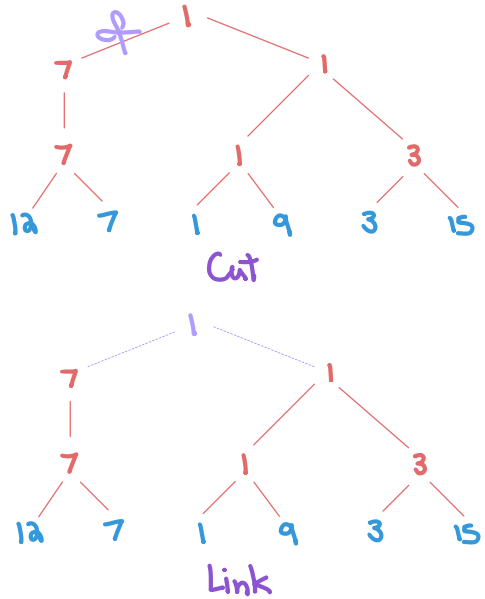
$\Phi = C_t(\#\text{trees}) + C_u(\#\text{unary nodes}) + C_n(\#\text{nodes})$



① All tree heights distinct
② Max height $O(\log n)$
} paid by Φ

(internal) Nodes
- store min over subtree
- have 1 or 2 children

Leaves
- store elements
- same depth



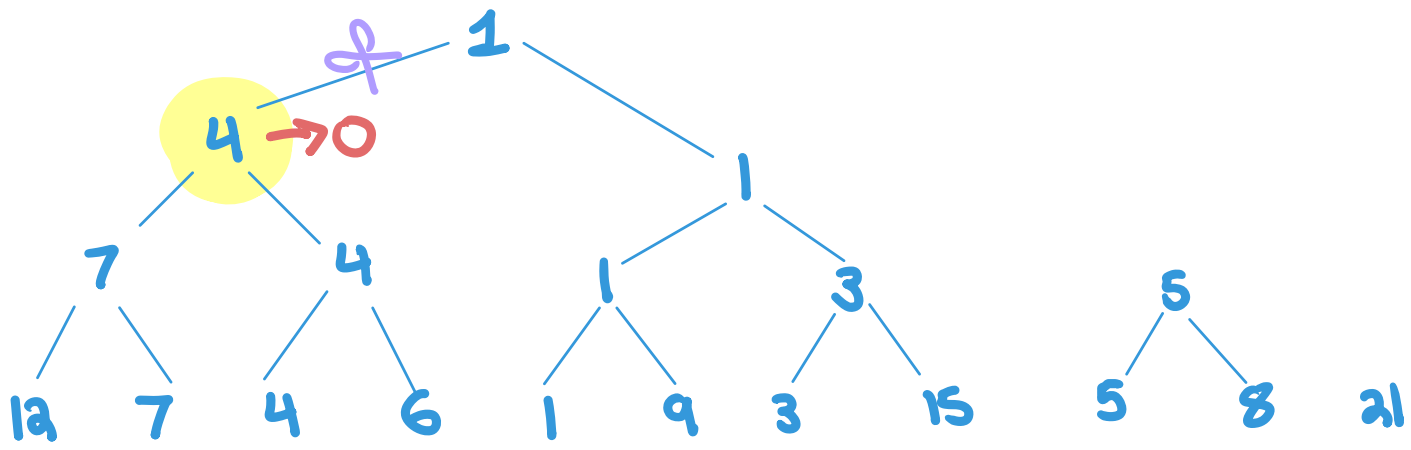
1. `insert`(k, p): inserts an key k with priority p.

1. make singleton for (k,p)

(and cleanup)

O(1) work (excluding cleanup)
+ O(1) increase in Φ
O(1) amortized time

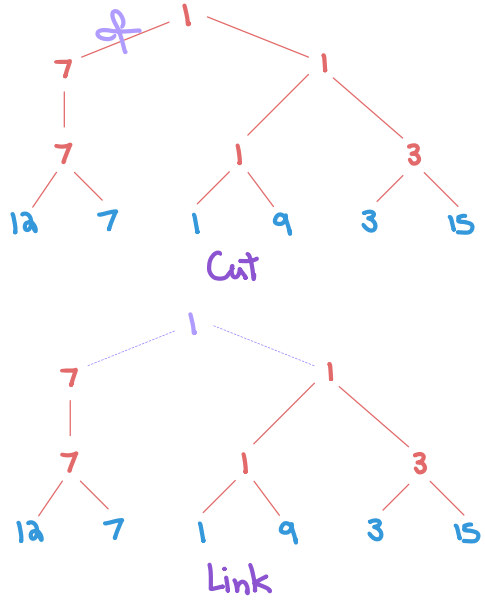
$\Phi = C_t(\#\text{trees}) + C_u(\#\text{unary nodes}) + C_n(\#\text{nodes})$



① All tree heights distinct
② Max height $O(\log n)$
} paid by Φ

(internal) Nodes
- store min over subtree
- have 1 or 2 children

Leaves
- store elements
- same depth

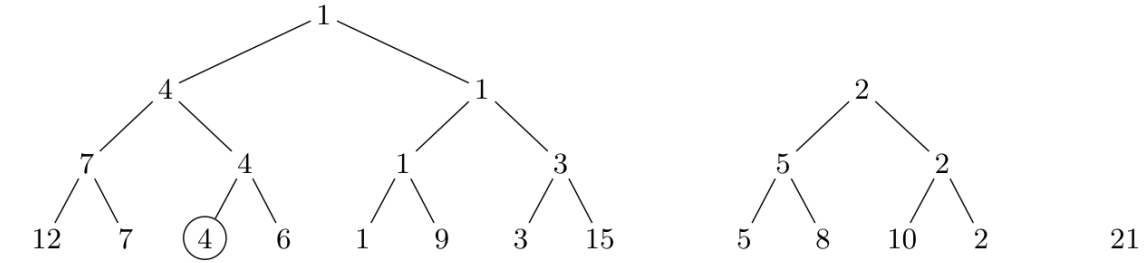


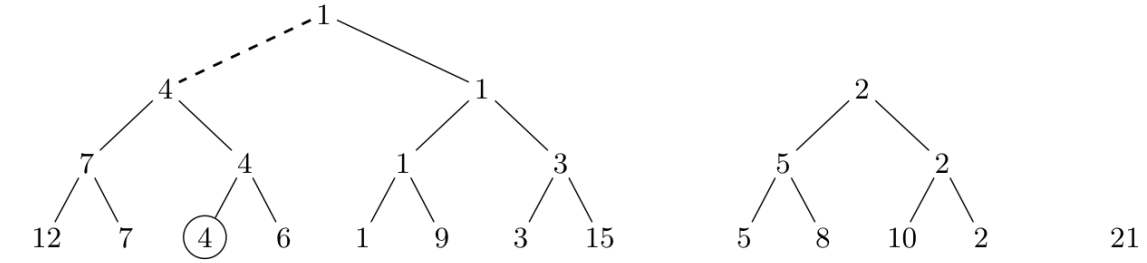
2. decrease(k, p): Let k be a key in the heap. If p is less than the current priority of k, decrease the priority to p.

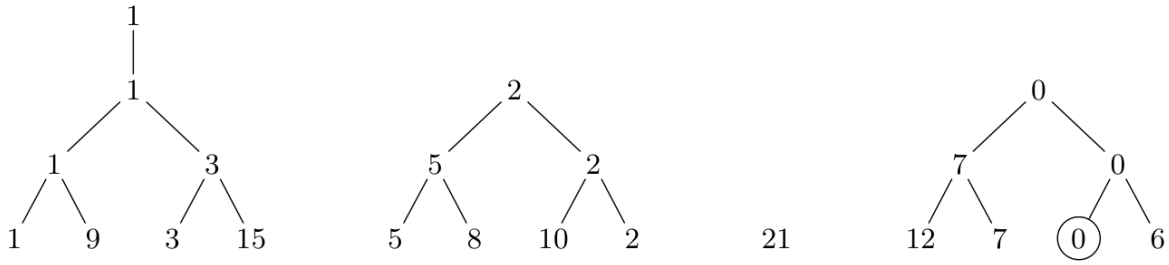
1. cut parent of "top node" of element (making it root)
2. decrease value

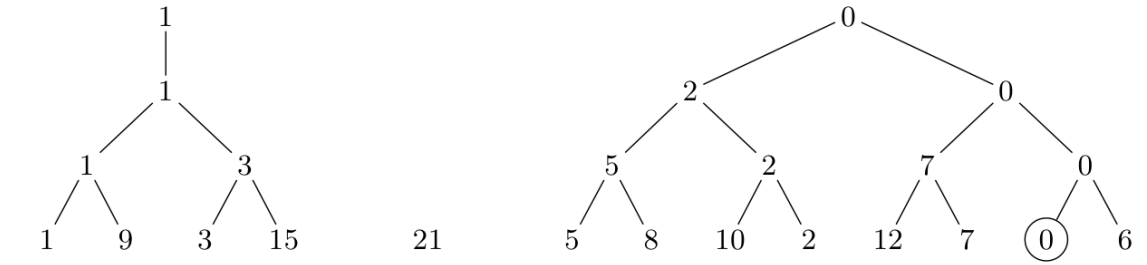
(and cleanup)

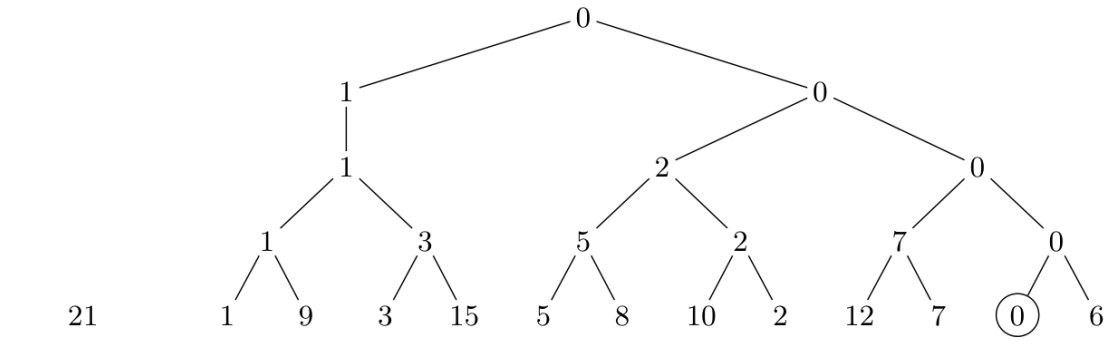
$O(1)$ work (excluding cleanup)
$+ O(1)$ increase in Φ
$O(1)$ amortized time


1
4 1
7 4 1 3
12 7 4 6 1 9 3 15
2
5 2
5 8 10 2
21

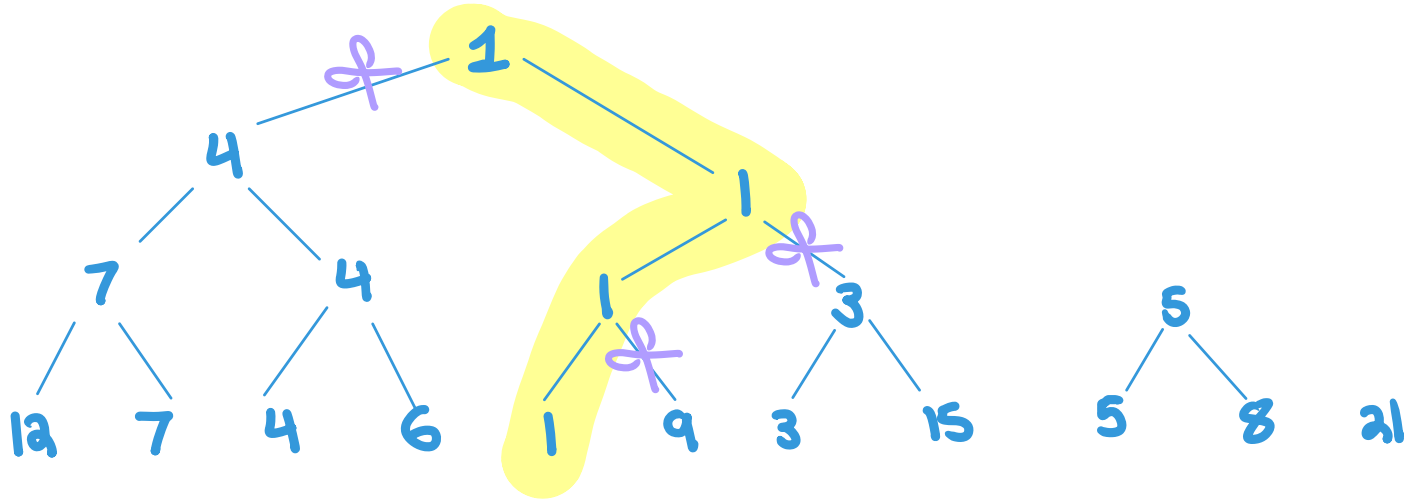

1
4 1
7 4 1 3
12 7 4 6 1 9 3 15
2
5 2
5 8 10 2
21


1
1
1 3
1 9 3 15
2
5 2
5 8 10 2
21
0
7 0
12 7 0 6


1
1
1 3
1 9 3 15
21
0
2 0
5 2 7 0
5 8 10 2 12 7 0 6


0
1 0
1 2 0
1 3 5 2 7 0
21
1 9 3 15 5 8 10 2 12 7 0 6

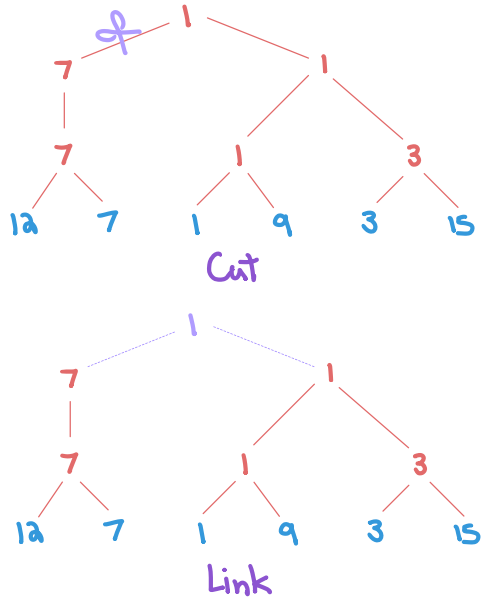
$\Phi = C_t(\#\text{trees}) + C_u(\#\text{unary nodes}) + C_n(\#\text{nodes})$



① All tree heights distinct
② Max height $O(\log n)$
} paid by Φ

(internal) Nodes
- store min over subtree
- have 1 or 2 children

Leaves
- store elements
- same depth



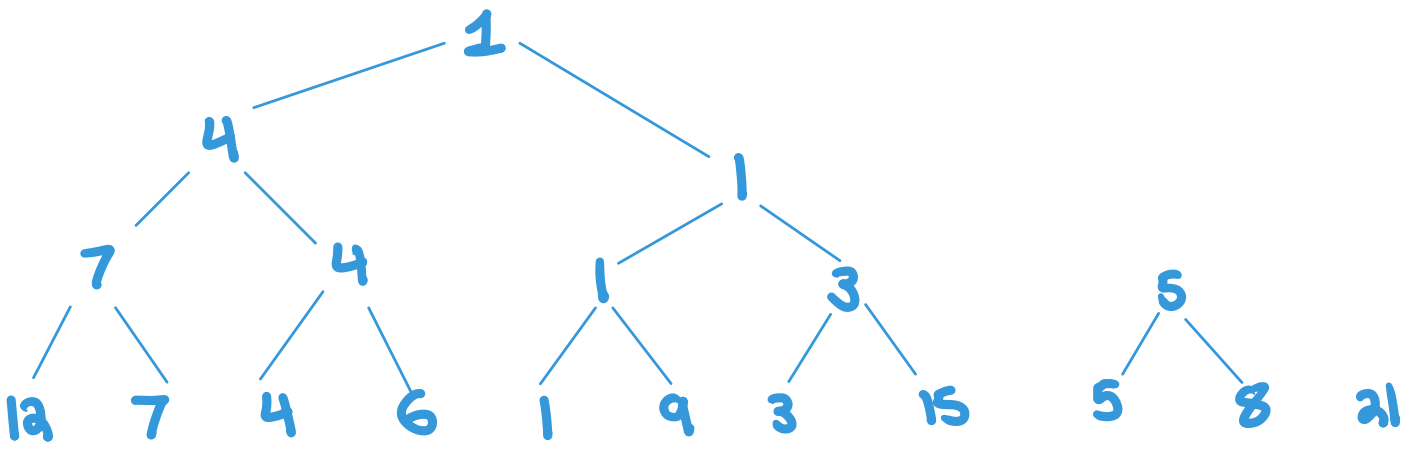
3. `remove-min()`: removes and returns the key value pair (k, p) with the minimum priority p.

1. find min by scanning trees
2. cut all subtrees

(and cleanup)

$O(\log n)$ work (excluding cleanup)
$+ O(\log n)$ increase in Φ
$O(\log n)$ amortized time

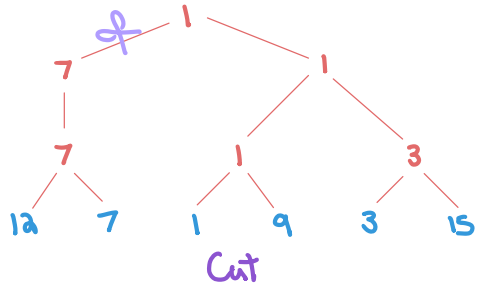
$\Phi = C_t(\#\text{trees}) + C_u(\#\text{unary nodes}) + C_n(\#\text{nodes})$

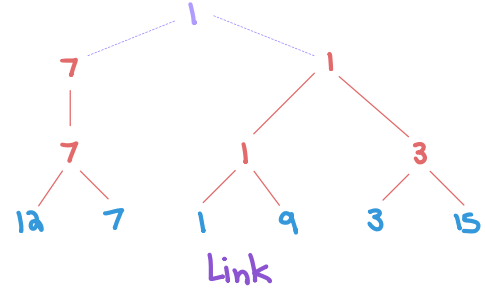


① All tree heights distinct
② Max height $O(\log n)$
} paid by Φ

(internal) Nodes
- store min over subtree
- have 1 or 2 children

Leaves
- store elements
- same depth





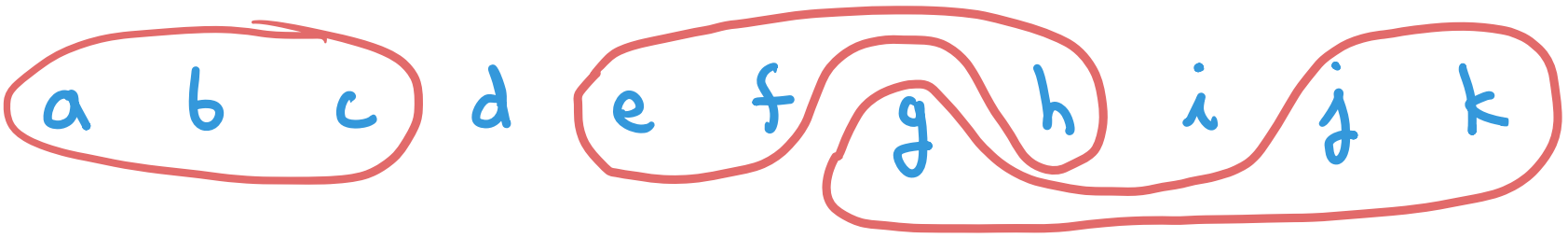
Amortized

$O(1)$ 1. `insert`(k, p): inserts an key k with priority p.

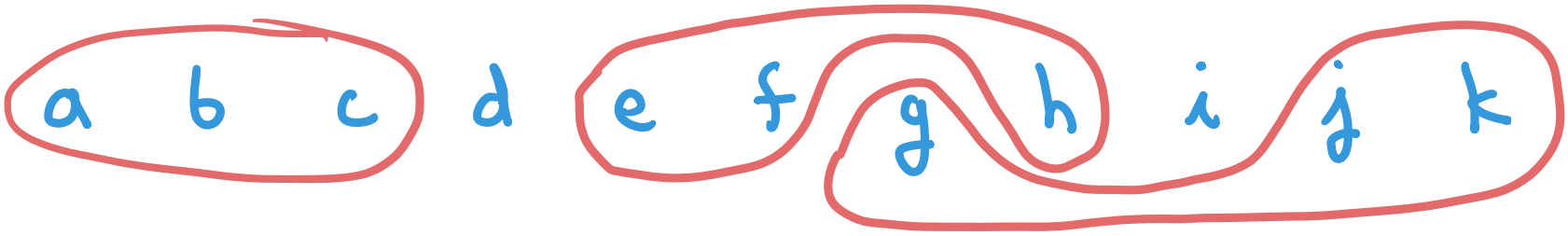
$O(1)$ 2. `decrease`(k, p): Let k be a key in the heap. If p is less than the current priority of k, decrease the priority to p.

$O(\log n)$ 3. `remove-min()`: removes and returns the key value pair (k, p) with the minimum priority p.

Disjoint Union



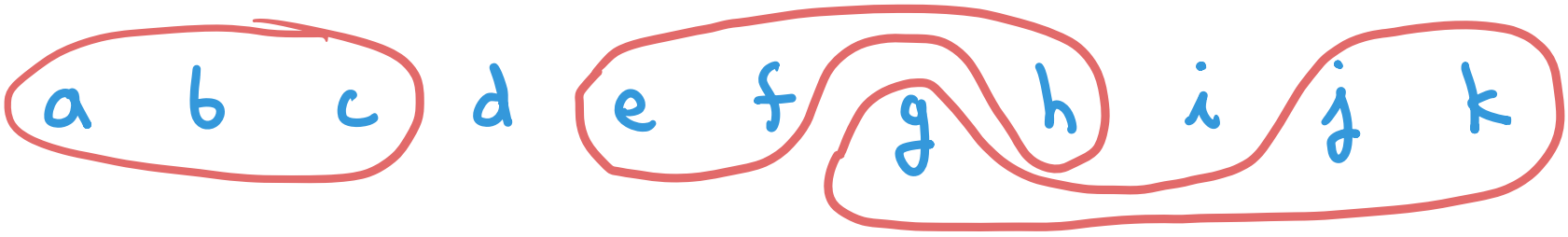
Disjoint Union



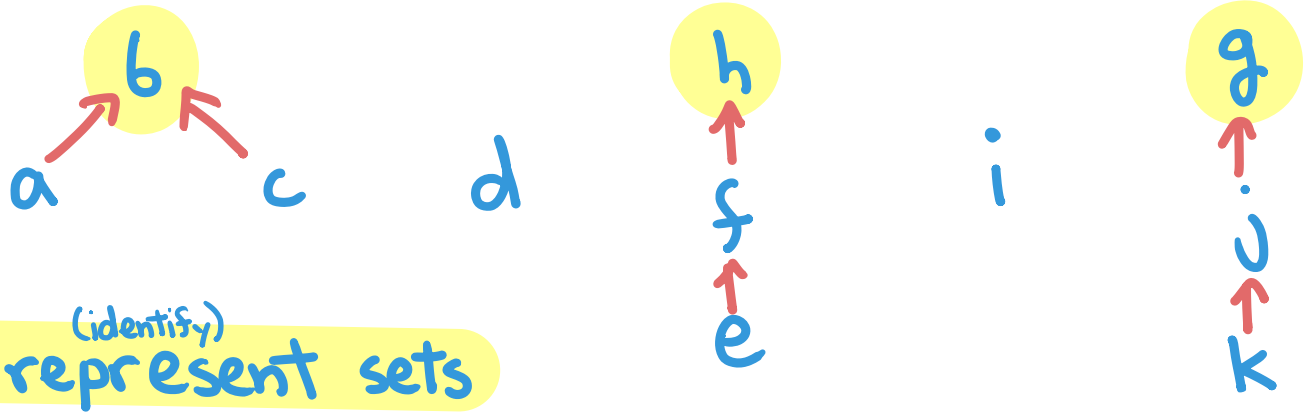
- `new-set`(x): Create a new singleton set containing x.
- `union`(x,y): Merge the set containing x with the set containing y.
- `same-set`(x,y): Return `true` if x and y are in the same set; else return `false`.

Disjoint Union

- `new-set`(x): Create a new singleton set containing x.
- `union`(x, y): Merge the set containing x with the set containing y.
- `same-set`(x, y): Return `true` if x and y are in the same set; else return `false`.



Sets = rooted trees



roots represent (identify) sets

`find`(x)

1. Follow parent pointers from x and return the root.

// O(height)

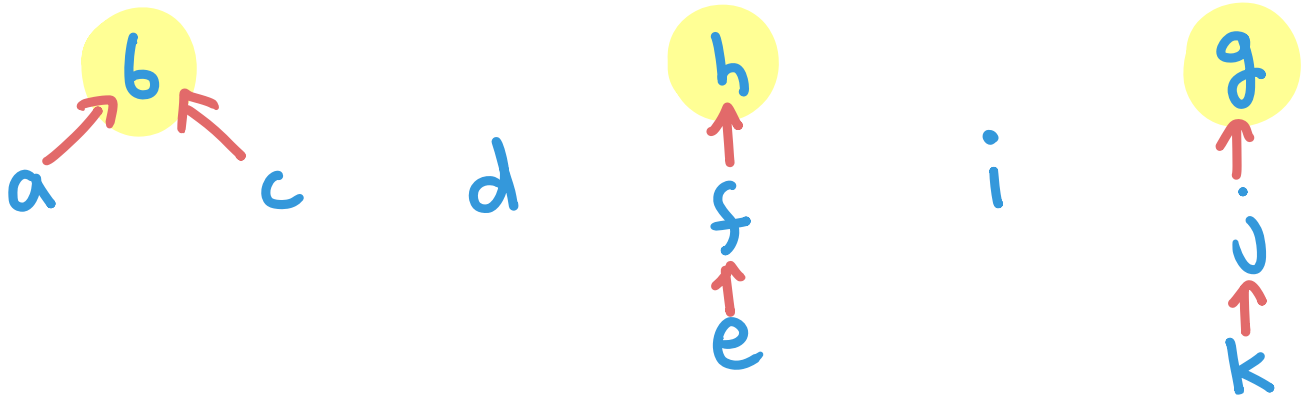
`union`(x, y)

1. Let $a \leftarrow$ `find`(x) and $b \leftarrow$ `find`(y).

2. make one a child of the other ← which?

Sets = rooted trees

roots represent sets



`find`(x)

1. Follow parent pointers from x and return the root. // O(height)

Union by rank

(singletons have rank 0)

`union`(x, y)

1. Let $a \leftarrow$ `find`(x) and $b \leftarrow$ `find`(y).
2. If a.`rank` $> b$.`rank` then make b a child of a.
3. Else if b.`rank` $> a$.`rank` then make a a child of b.
4. Otherwise make b a child of a and increment the rank of a.

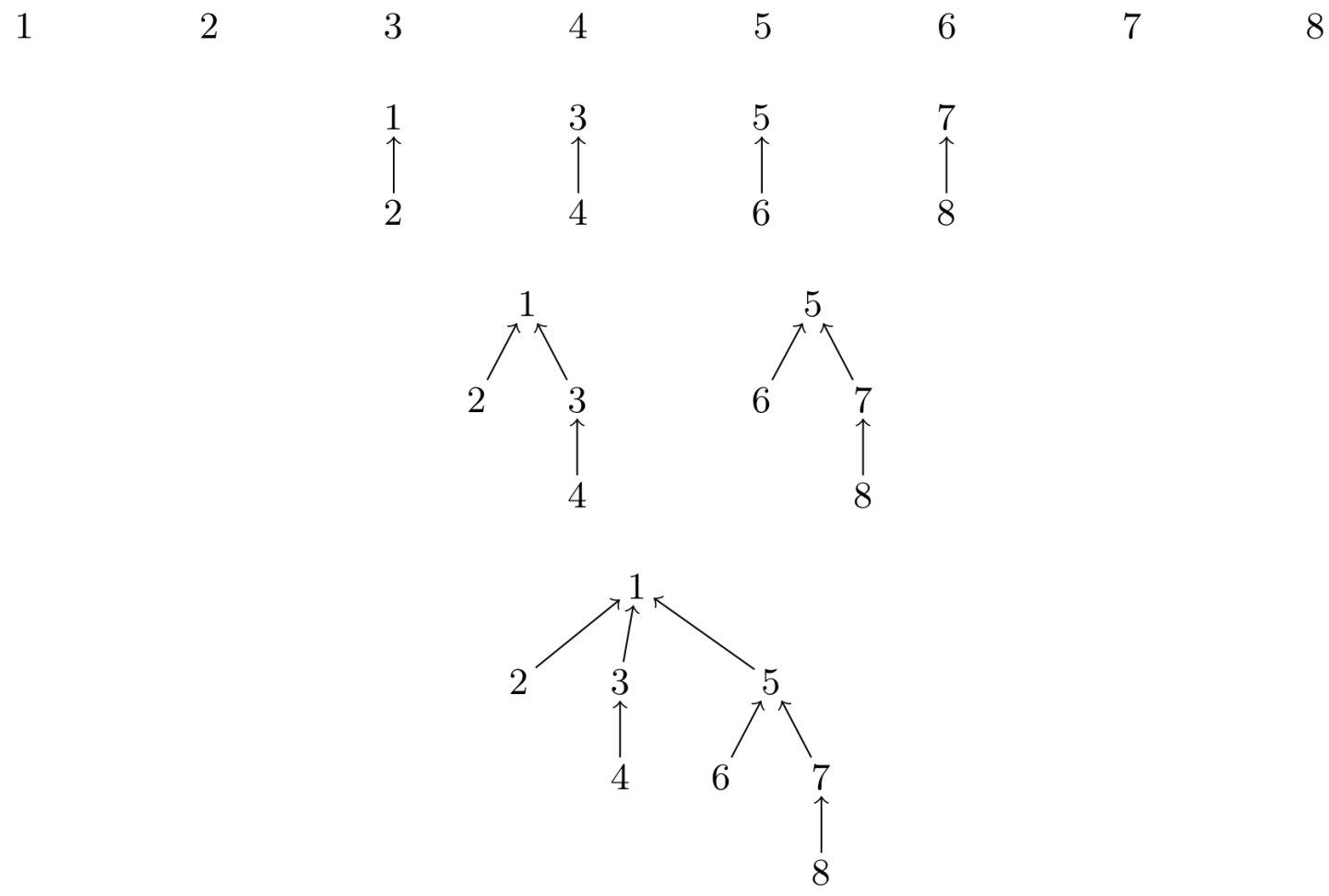
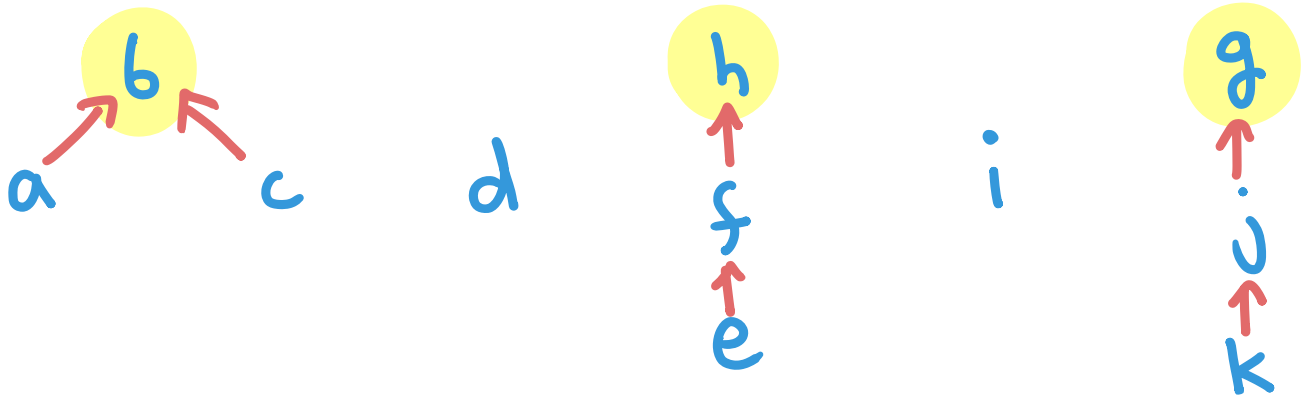


Figure 23.2: Building a tree by union by rank. Eight singletons are merged in three rounds.

Sets = rooted trees

roots represent sets



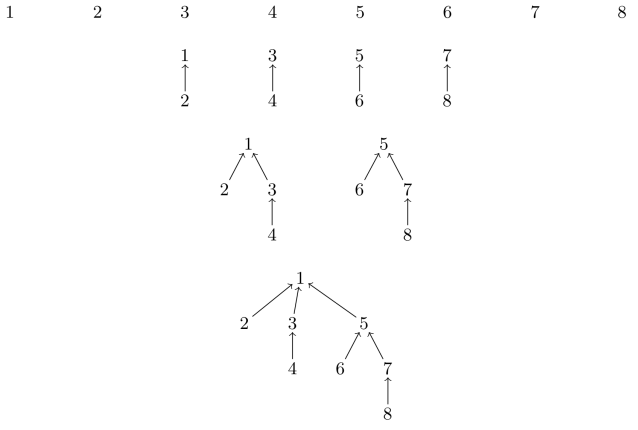
`find`(x)

1. Follow parent pointers from x and return the root.

// O(height)

Union by rank

(singletons have rank 0)



`union`(x, y)

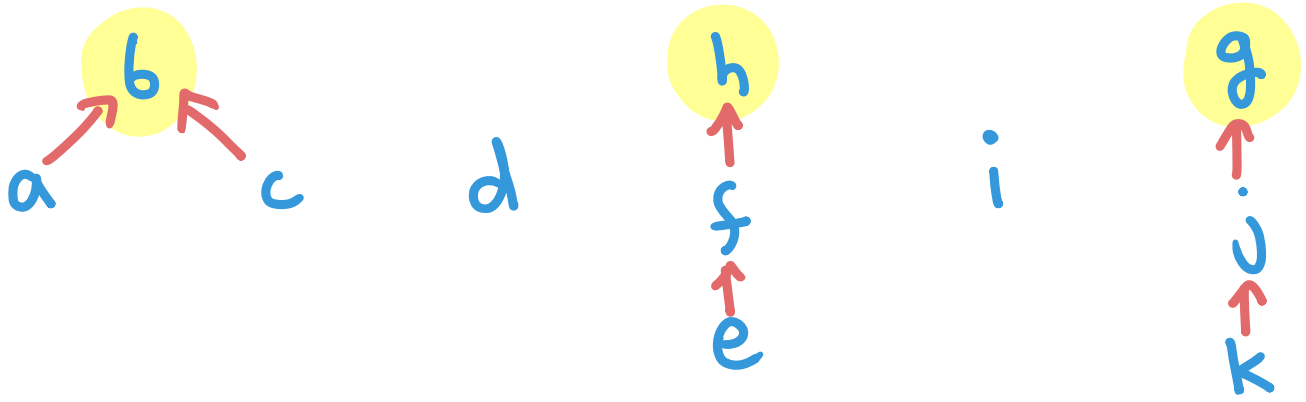
1. Let $a \leftarrow$ `find`(x) and $b \leftarrow$ `find`(y).
2. If a.`rank` $>$ b.`rank` then make b a child of a.
3. Else if b.`rank` $>$ a.`rank` then make a a child of b.
4. Otherwise make b a child of a and increment the rank of a.

Lemma: tree w/ rank (@ root) k has $\geq 2^k$ nodes

why?

Sets = rooted trees

roots represent sets

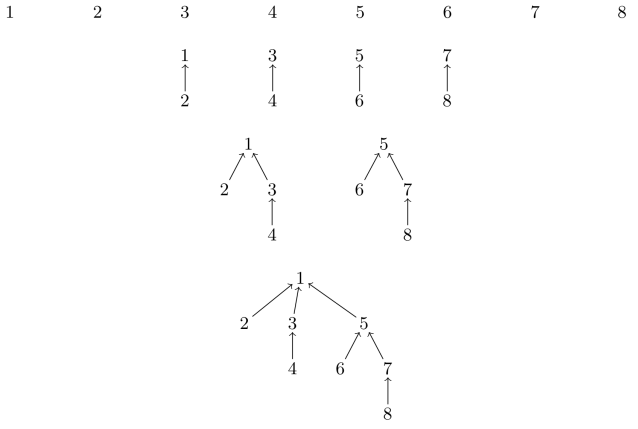


`find`(x)

1. Follow parent pointers from x and return the root. // O(height)

Union by rank

(singletons have rank 0)



`union`(x, y)

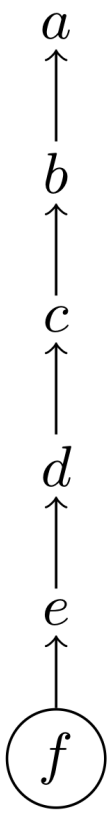
1. Let $a \leftarrow$ `find`(x) and $b \leftarrow$ `find`(y).
2. If a.`rank` $>$ b.`rank` then make b a child of a.
3. Else if b.`rank` $>$ a.`rank` then make a a child of b.
4. Otherwise make b a child of a and increment the rank of a.

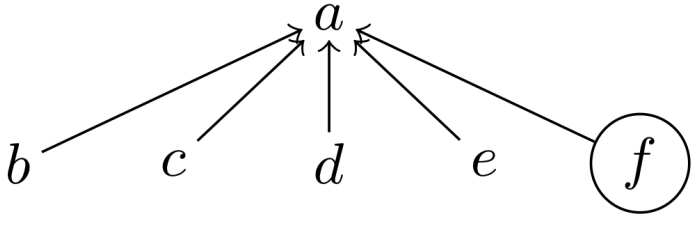
Lemma: tree w/ rank (@ root) k has $\geq 2^k$ nodes

=> max height O(log n) => O(log n) time.

Better?

Path Compression



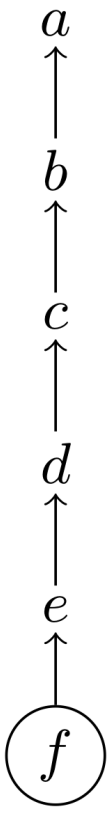


find(x)

/* *Follow parent pointers from x to its root, compressing the x-to-root path along the way.* */

1. If x.parent is nil then return x.
2. x.parent $\leftarrow$ find(x.parent).
3. Return x.parent.

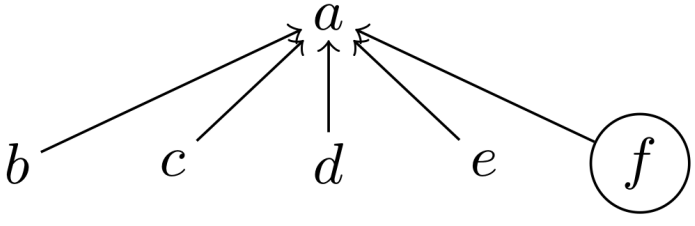
Path Compression

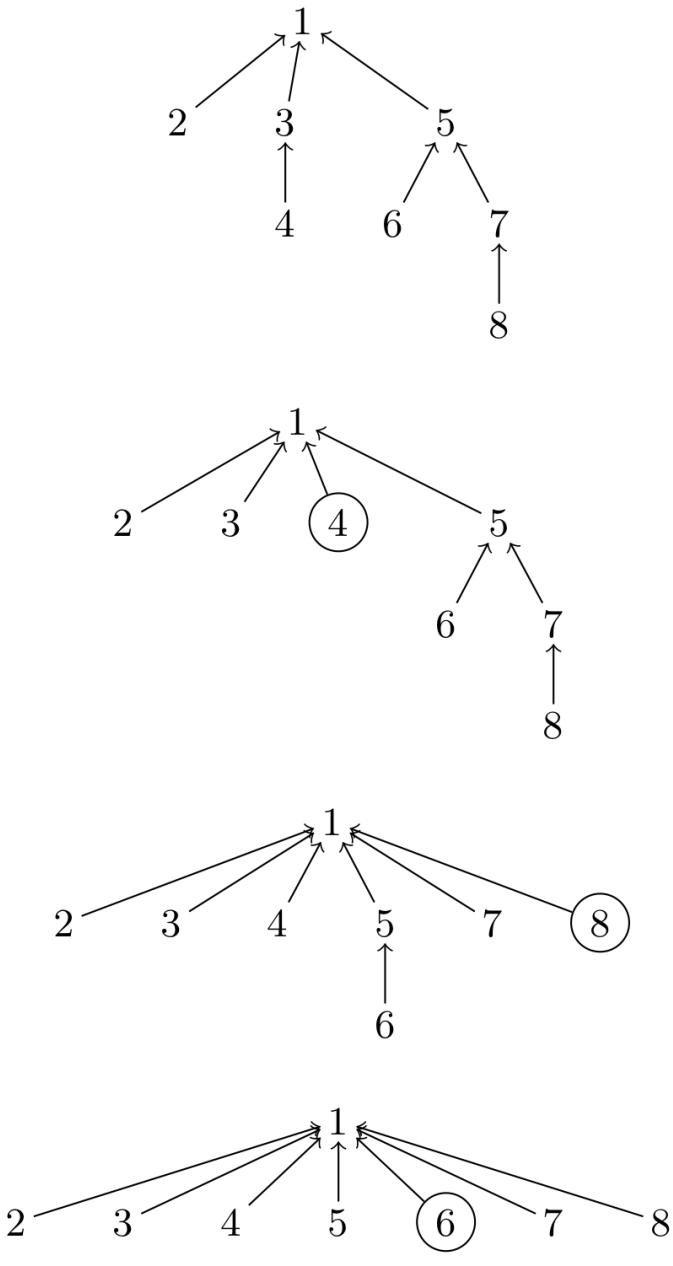


`find`(x)

/ Follow parent pointers from x to its root, compressing the x-to-root path along the way. */*

1. If x.`parent` is `nil` then return x.
2. x.`parent` $\leftarrow$ `find`(x.`parent`).
3. Return x.`parent`.





Flattening a set over three calls to `find`.

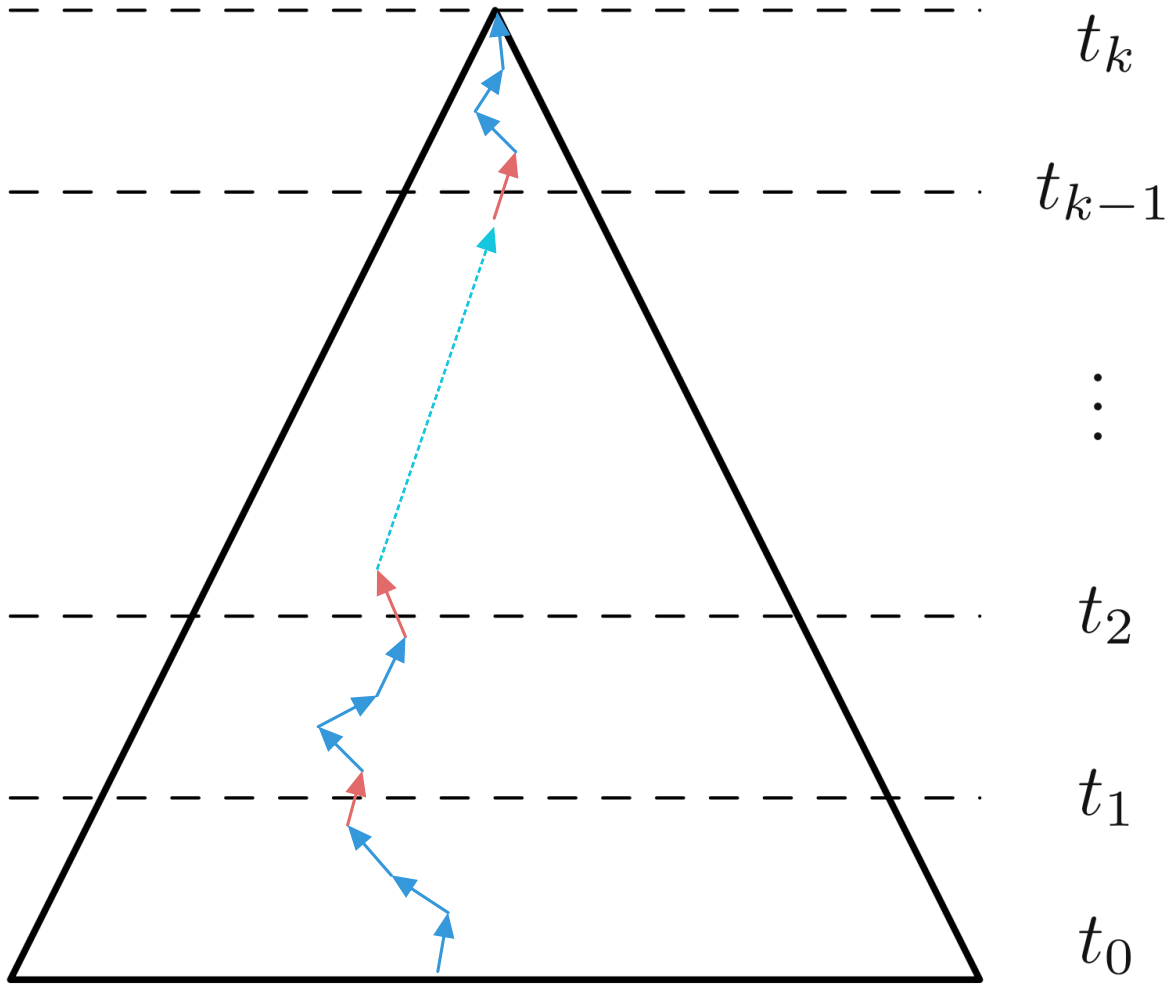
Analysis divide ranks into thresholds

$$0 = t_0 < t_1 < \cdots < t_k = \lfloor \log n \rfloor$$

"big hops" across thresholds

how many? k

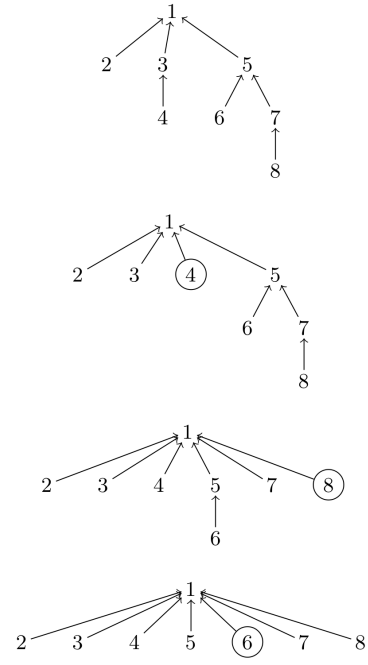
"small hops" between thresholds



`find`(x)

/ Follow parent pointers from x to its root, compressing the x-to-root path along the way. */*

1. If x.`parent` is `nil` then return x.
2. x.`parent` $\leftarrow$ `find`(x.`parent`).
3. Return x.`parent`.

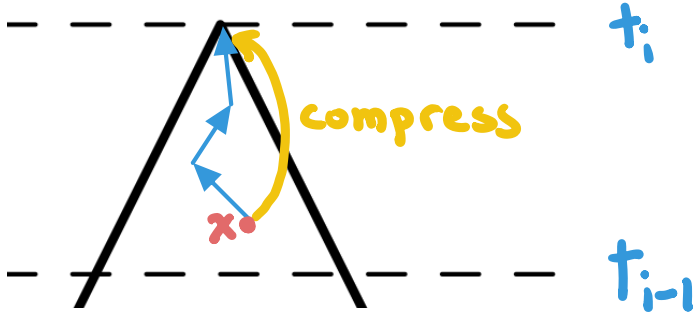


Flattening a set over three calls to `find`.

Analysis divide ranks into thresholds

$$0 = t_0 < t_1 < \cdots < t_k = \lfloor \log n \rfloor$$

"small hops" between thresholds



For node x w/ $t_{i-1} \le \text{rank}(x) < t_i$

$$\Phi_x = \max\{0, t_i - \text{rank}(\text{parent}(x))\}$$

pays for small compressions

$$(\#^{\text{nodes}}\, x \text{ w/ } t_{i-1} \le \text{rank}(x) < t_i) \le \sum_{s=t_{i-1}}^{t_i} \frac{n}{2^s} = O\left(\frac{n}{2^{t_{i-1}}}\right)$$

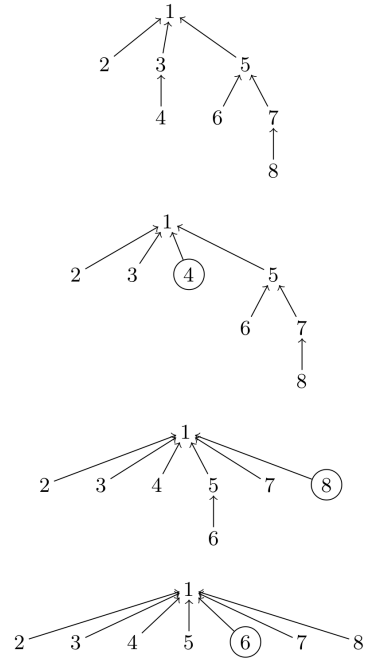
Total potential over $[t_{i-1}, t_i)$:

$$O\left(\frac{n}{2^{t_{i-1}}} \cdot t_i\right) = O(n) \quad \text{for } t_i = 2^{t_{i-1}}$$

`find`(x)

/ Follow parent pointers from x to its root, compressing the x-to-root path along the way. */*

1. If x.`parent` is `nil` then return x.
2. x.`parent` $\leftarrow$ `find`(x.`parent`).
3. Return x.`parent`.



Flattening a set over three calls to `find`.

Analysis divide ranks into thresholds

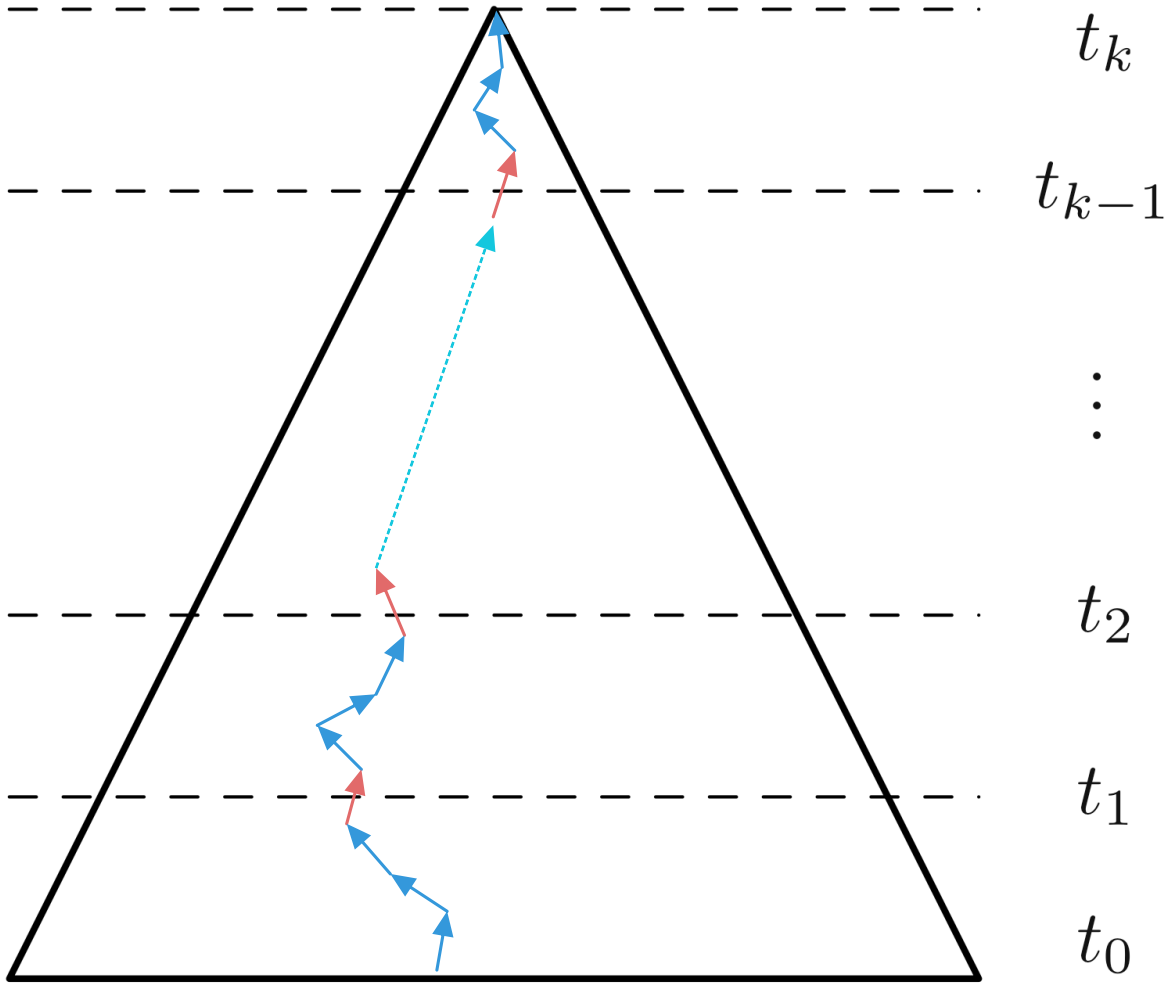
$$0 = t_0 < t_1 < \cdots < t_k = \lfloor \log n \rfloor$$

$t_i = 2^{t_{i-1}} \Rightarrow k = \log^* n$

"big hops" across thresholds
how many?
k

"small hops" between thresholds

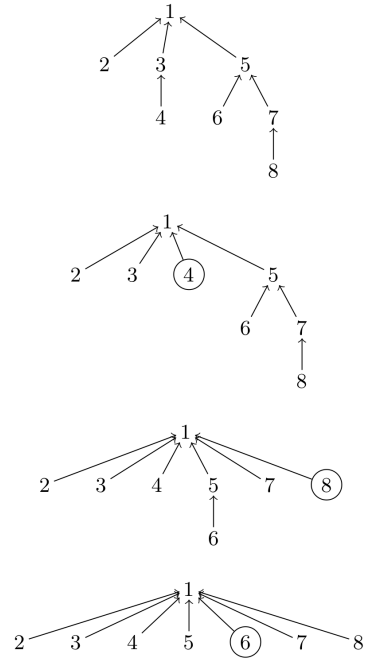
O(nk) total work for $t_i = 2^{t_{i-1}}$



`find`(x)

/ Follow parent pointers from x to its root, compressing the x-to-root path along the way. */*

1. If x.`parent` is `nil` then return x.
2. x.`parent` $\leftarrow$ `find`(x.`parent`).
3. Return x.`parent`.



Flattening a set over three calls to `find`.

Analysis divide ranks into thresholds

$$0 = t_0 < t_1 < \cdots < t_k = \lfloor \log n \rfloor$$

$t_i = 2^{t_{i-1}} \Rightarrow k = \log^* n$

"big hops across thre

how man k

$\log^*(n)$

$\log(\log(\log(\sim(\log(n))\sim))) \leq 1$

applied $\log^*(n)$ times

"small hops" between thresholds

$O(nk)$ total work for $t_i = 2^{t_{i-1}}$

t_k

t_{k-1}

t_2

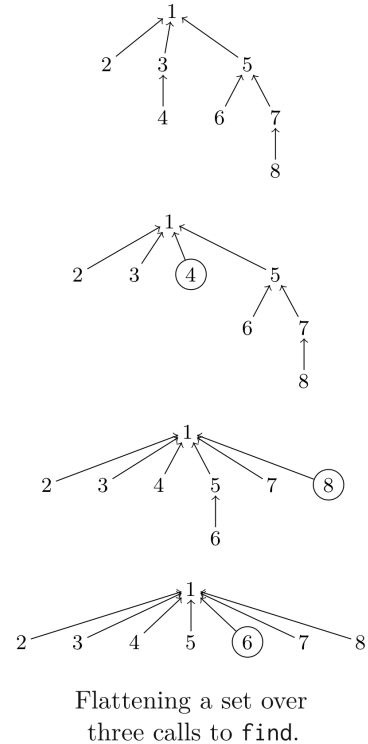
t_1

t_0

`find(`x`)`

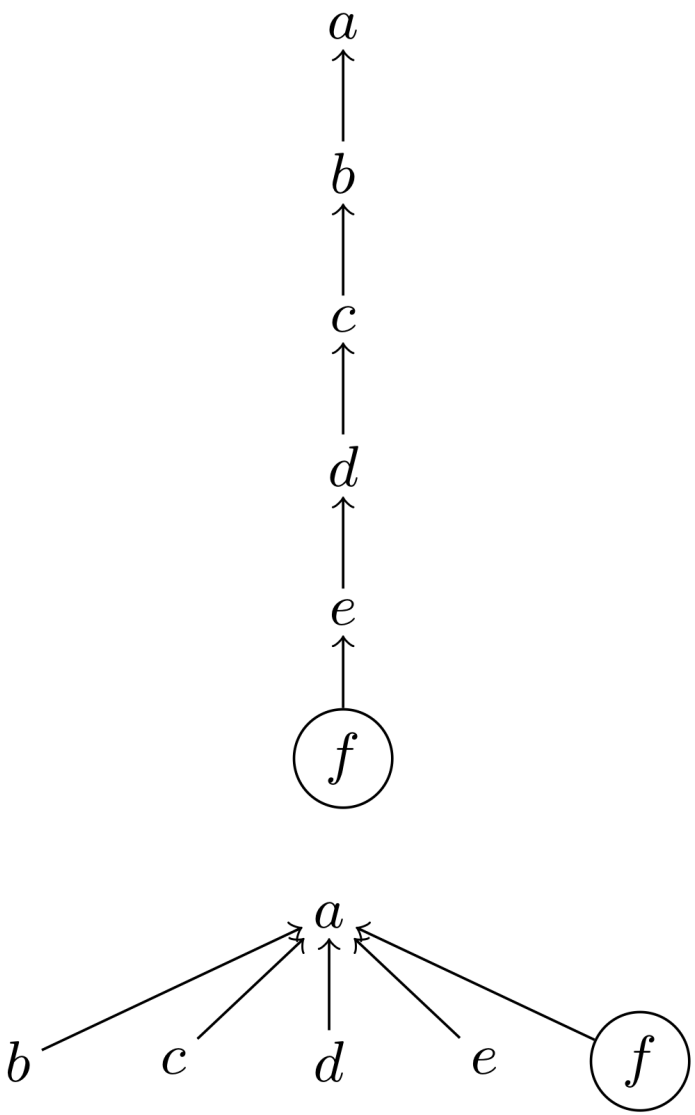
/ Follow parent pointers from x to its root, compressing the x-to-root path along the way. */*

1. If x.`parent` is `nil` then return x.
2. x.`parent` $\leftarrow$ `find(`x.`parent)`.
3. Return x.`parent`.



Flattening a set over three calls to `find`.

Path Compression



find(x)

/* *Follow parent pointers from x to its root, compressing the x-to-root path along the way.* */

1. If x.parent is nil then return x.
2. x.parent $\leftarrow$ find(x.parent).
3. Return x.parent.

$O(\log^* n)$ amortized for union / find

Better?

Yes, w/ better thresholds/potential

Fix node x w/ rank r

"Hierarchical grid of thresholds"

scale 0	$r+1$	$r+2$	$r+3$	⋯	$2r-1$
scale 1	2^r	$2^2 r$	$2^3 r$	⋯	$2^{r-1} r$
scale 2	$2^{2^r} r$	$2^{2^{2^r}}(2^{2^r} r)$	$2^{2^{2^{2^r}}(2^{2^{2^r}} r)}$	⋯	$2^{2^{\cdot^{\cdot^{\cdot}(\sim)}}}(\sim)$
⋮					
scale k	$f_k(r) = f_{k-1}^{(r)}(r)$	$f_k^{(2)}(r)$	$f_k^{(3)}(r)$	⋯	$f_k^{(r-1)}(r)$

Pattern

functions per scale

generate thresholds at scale

scale 0: $f_0(t) = t+1$

scale 1: $f_1(t) = f_0^{(t)}(t) = \overbrace{f_0(f_0(\sim(f_0(t))\sim))}^{\text{applied } t \text{ times}}$

⋮

scale k: $f_k(t) = f_{k-1}^{(t)}(t) = \underbrace{f_{k-1}(f_{k-1}(\sim(f_{k-1}(t))\sim))}_{\text{applied } t \text{ times}}$

each entry represents interval (from this threshold to next) containing rank(parent(x))

* "th

"Hierarchical grid of thresholds"

scale 0	$r+1$	$r+2$	$r+3$	⋯	$2r-1$
scale 1	2^r	$2^2 r$	$2^3 r$	⋯	$2^{r-1} r$
scale 2	$2^r r$	$2^{2^r}(2^r r)$	$2^{2^{2^r}(2^r r)}(2^{2^r}(2^r r))$	⋯	$2^{2^{\cdot^{\cdot^{\cdot}(\sim)}}(\sim)}$
⋮					
scale k	$f_k(r) = f_{k-1}^{(r)}(r)$	$f_k^{(2)}(r)$	$f_k^{(3)}(r)$	⋯	$f_k^{(r-1)}(r)$

"level" $\ell(x)=2$ →

"index" $i(x)=3$ ↑

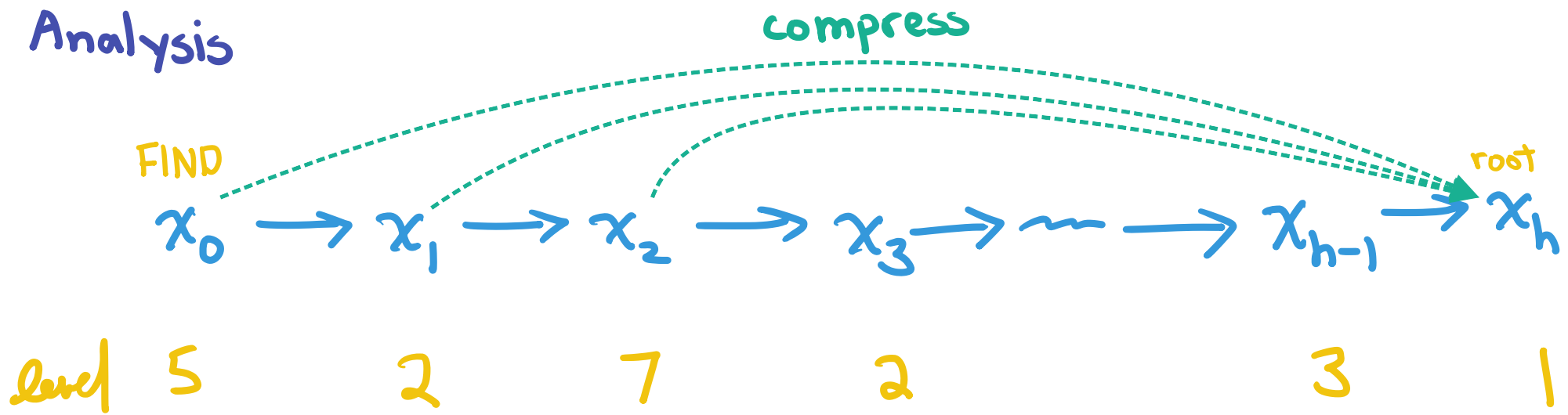
$$\Phi_x = (L - \ell(x))\,\mathrm{rank}(x) - i(x)$$

level

index

Max level +1

Analysis



x_i "big" if $\ell(x_i) > \ell(x_j)$ $\forall j > i$ how many big elements?

x_i "small" if $\ell(x_i) < \ell(x_j)$ for some $j < i$

"Hierarchical grid of thresholds"

scale 0	$r+1$	$r+2$	$r+3$	~	$2r-1$
scale 1	2^r	$2^2 r$	$2^3 r$	~	$2^{r-1} r$
scale 2	$2^r r$	$2^{2^r}(2^r r)$	$2^{2^{2^r}(2^r r)}(2^{2^r}(2^r r))$	~	$2^{2^{\cdot^{\cdot^{\cdot}(\cdot)}}(\cdot)}(\sim)$
⋮					
scale k	$f_k(r) = f_{k-1}^{(r)}(r)$	$f_k^{(2)}(r)$	$f_k^{(3)}(r)$	~	$f_k^{(r-1)}(r)$

"level" $\ell(x)=2$ → scale 2

"index" $i(x)=3$ → $f_k^{(3)}(r)$

$$\Phi_x = (L - \ell(x)) \operatorname{rank}(x) - i(x)$$

level: $L - \ell(x)$; index: $i(x)$

Chapter 23

Priority queues and disjoint union

23.1 Priority queues

The humble heap stores key-value pairs and supports two operations.

- `insert(`k`,`v`)`: insert a new key-value pair (k, v).
- `remove-min()`: remove and return the pair (k, v) with minimum value v.

The standard binary heap supports both operations in $O(\log n)$ time. Among other applications, heaps can sort n values in $O(n \log n)$ time.

We have seen two basic graph algorithms, for shortest paths and for MST, that use heaps. Both algorithms build out trees starting from a single vertex and use heaps to identify the next closest vertex to add. In addition to calling each of the heap operations above once for every vertex, they also call the following operation for every edge in the graph.

- `decrease(`k`,`v`)`: decrease the value of an existing key k to v.

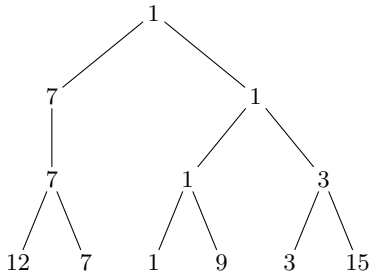
A *priority queue* is a heap that also supports `decrease`.

A binary heap supports `decrease` in $O(\log n)$ time. Surprisingly, Fredman and Tarjan [FT87] showed that `decrease` can be implemented in $O(1)$ amortized time, with a data structure called the *Fibonacci heap*.

Theorem 23.1 ([FT87])**.** *There exists a data structure supporting* `insert`*,* `decrease`*, and* `remove-min` *in* $O(1)$*,* $O(1)$*, and* $O(\log n)$ *amortized time, respectively.*

We follow a simpler construction from [Cha13].

Tournament trees. A tournament tree is a rooted tree where every node has one or two children, and all leaves are at the same depth. All elements are stored at the leaves. Each internal node stores the minimum element in its subtree. The *top node* of an element is the highest node storing it.



Two tournament trees of the same height h can be combined to form a single tournament tree of height $h+1$, by making them children of a new root. This is called *linking*.

If x is a non-root top node, we can *cut* the subtree rooted at x to form its own tournament tree. The parent becomes unary, and the rest of the tree remains a tournament tree. (Why?)

Maintaining tournament trees. Our heap is a collection of tournament trees satisfying two invariants.

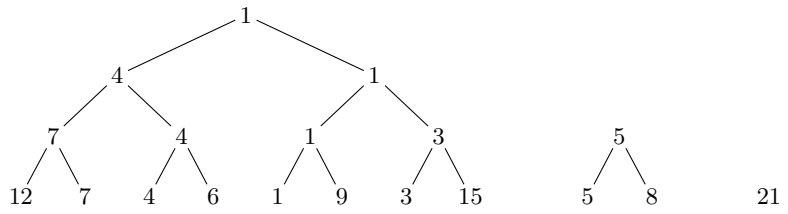
(a) All tree heights are distinct.

(b) Every tree has height $O(\log n)$.

To pay for this housekeeping, consider the potential

$$\Phi = C_t(\#\text{ trees}) + C_s(\#\text{ unary nodes}) + C_n(\#\text{ nodes}),$$

for suitable positive constants C_t, C_s, C_n.



If two trees have the same height h, we link them. This adds one new root and decreases the number of trees by 1. For C_t a constant factor larger than C_n, the drop in Φ pays for the linking.

Cuts create unary nodes. Too many of them near the top make the tree taller than it needs to be. Whenever at least half the nodes at some height h are unary, we delete every node at height h and above. This leaves a collection of trees of height

$h - 1$. The number of new trees is at most a constant factor larger than the number of deleted unary nodes at height h. For C_s a constant factor larger than C_t, the drop in unary nodes pays for the new trees. The drop in the number of nodes pays for the deletions.

The cleanup is free. Linking ensures that all tree heights are distinct. Rebuilding when there are too many unary nodes keeps the heights bounded by $O(\log n)$. There are only $O(\log n)$ trees.

Priority queue operations. `insert` creates a singleton tree. `decrease` cuts at the top node of the decreased element and updates its value. Both take constant time and increase the potential by at most a constant, so the amortized cost of each is $O(1)$.

`remove-min` scans the roots to find the minimum, say at key k, and deletes the root-to-leaf path of nodes containing k. This creates $O(\log n)$ new trees. The time and potential increase are both $O(\log n)$, so the amortized cost is $O(\log n)$.

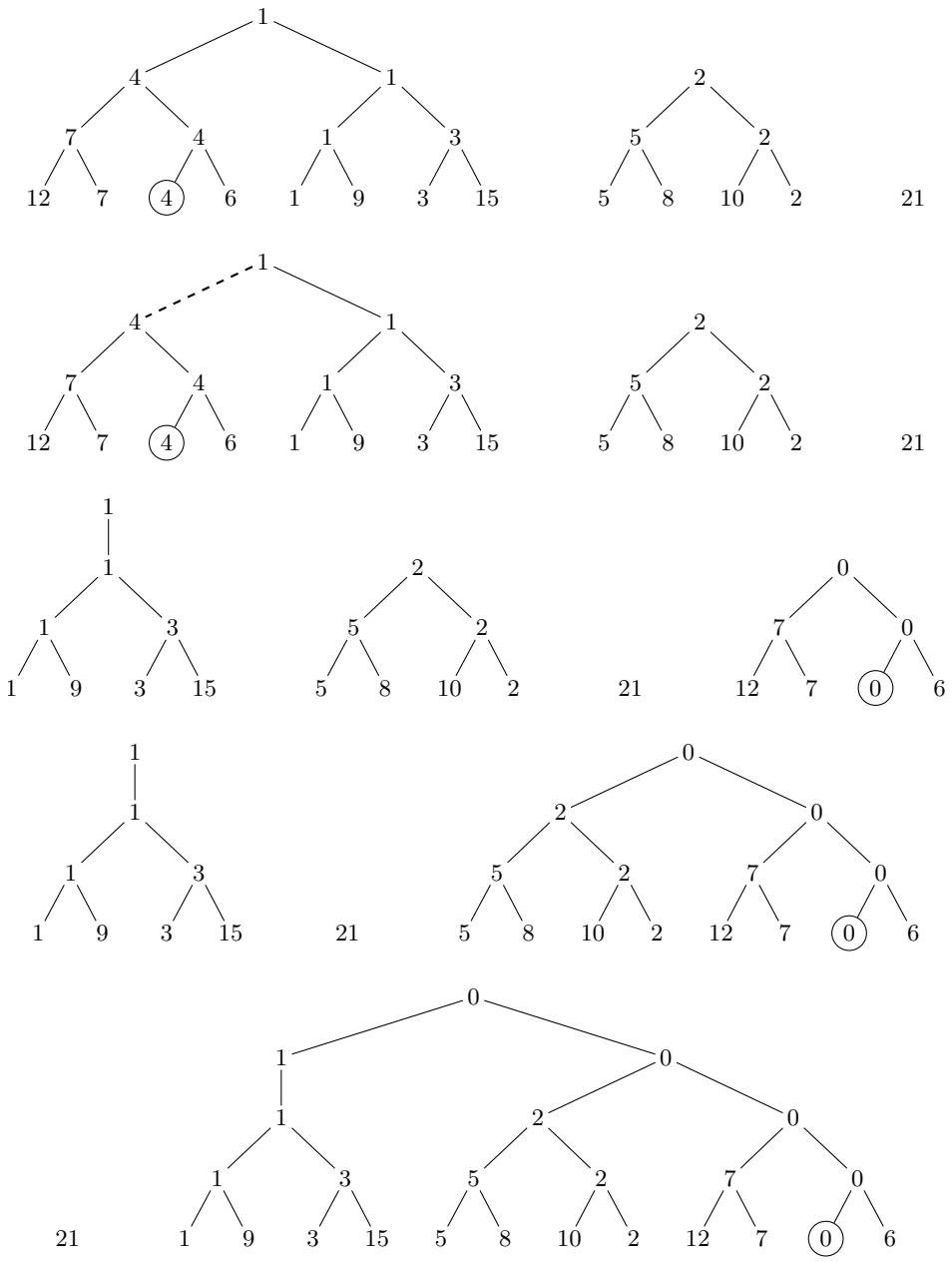


Figure 23.1: Decreasing element 4 to 0, and consolidating.

23.2 Disjoint union

The greedy algorithm processes the edges in increasing order of weight, taking any edge whose endpoints lie in distinct components. A *disjoint-union* data structure tracks these components as they merge.

A disjoint-union data structure over a set of n elements supports the following operations.

- `new-set(`x`)`: Create a new singleton set containing x.
- `union(`x`,`y`)`: Merge the set containing x with the set containing y.
- `same-set(`x`,`y`)`: Return `true` if x and y are in the same set; else return `false`.

Represent each set as a rooted tree. Each element stores a parent pointer. The root has no parent and serves as the *representative* of the set.

The basic operation is `find(`x`)`, which returns the representative of the set containing x.

`find(`x`)`

1. Follow parent pointers from x and return the root.

Then `same-set(`x`,`y`)` checks whether `find(`x`)` = `find(`y`)`.

To implement `union`, we need a rule for linking two roots. Each node stores a rank that upper-bounds the height of the subtree rooted there. Initially every node has rank 0. When linking two roots, the lower-rank root becomes a child of the higher-rank root. If the ranks are equal, either root can become the parent, and its rank increases by 1.

`union(`x`,`y`)`

1. Let $a \leftarrow$ `find(`x`)` and $b \leftarrow$ `find(`y`)`.
2. If a`.rank` $>$ b`.rank` then make b a child of a.
3. Else if b`.rank` $>$ a`.rank` then make a a child of b.
4. Otherwise make b a child of a and increment the rank of a.

A tree whose root has rank k contains at least 2^k nodes. (Why?) So every rank is at most $\log n$. Since rank upper-bounds height, every tree has height $O(\log n)$. Accordingly, `find` takes $O(\log n)$ time. `new-set` takes constant time. `union` is just two calls to `find`.

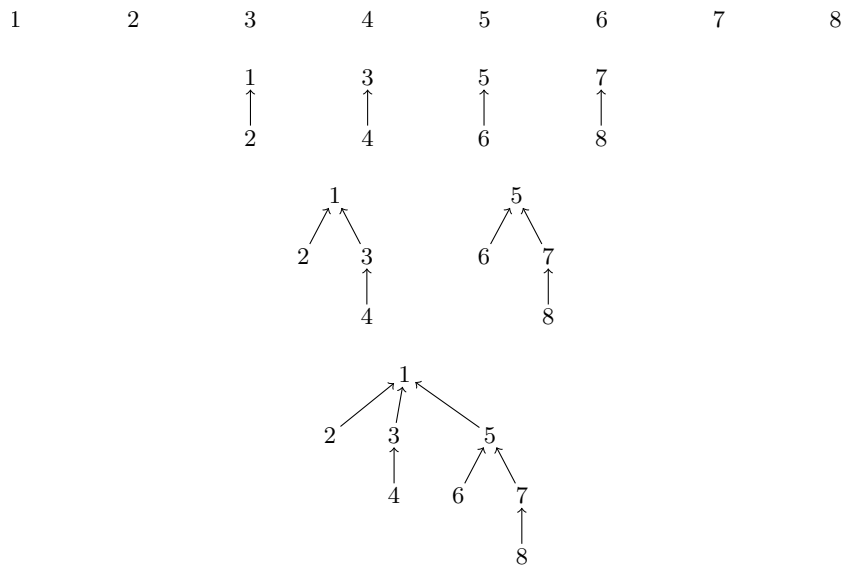
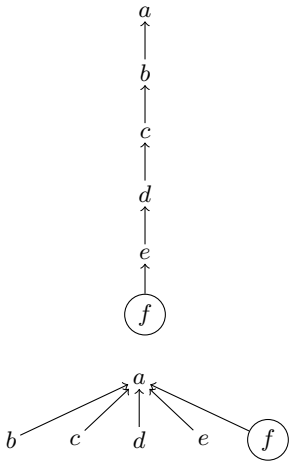


Figure 23.2: Building a tree by union by rank. Eight singletons are merged in three rounds.

Path compression. `find(`x`)` traverses the path from x to its root. Once the root is known, there is no reason to leave that path in place. Redirect x to the root, and while at it redirect the other nodes on the path to the root. Use the work of traversing the path to flatten it. This is called *path compression*. The next time we call `find` on any of these nodes, it returns in constant time.



`find(`x`)`

`/*` *Follow parent pointers from x to its root, compressing the x-to-root path along the way.* `*/`

1. If x`.parent` is `nil` then return x.
2. x`.parent` $\leftarrow$ `find(`x`.parent)`.
3. Return x`.parent`.

The cost of each `find` is proportional to the number of parent pointers followed. So the analysis is a count of pointer traversals.

We continue to use rank-based linking for `union`. Ranks never change, and they increase strictly from child to parent.

There are at most $n/2^i$ nodes of rank i. (Why?)

To $\log^* n$... All ranks lie in the interval $[0, \lfloor \log n \rfloor]$. Choose thresholds

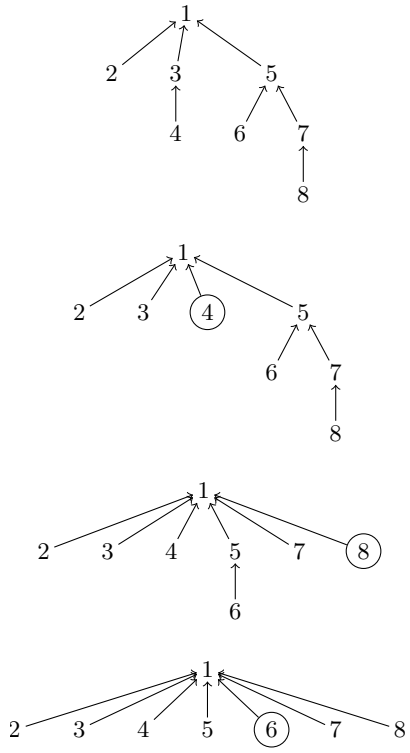
$$0 = t_0 < t_1 < \cdots < t_k = \lfloor \log n \rfloor.$$

These determine intervals

$$T_1 = [t_0, t_1),\ T_2 = [t_1, t_2),\ \ldots,\ T_{k-1} = [t_{k-2}, t_{k-1})$$
$$T_k = [t_{k-1}, t_k]$$

Each interval T_i has $\sum_{j \in T_i} n/2^j = O(n/2^{t_{i-1}})$ nodes.

Each node lies in a particular interval by its rank. Call a parent pointer "small" if it is contained in an interval, and "big" if it crosses intervals. There are at most k big pointers along any `find`, because each big pointer crosses to a higher interval.



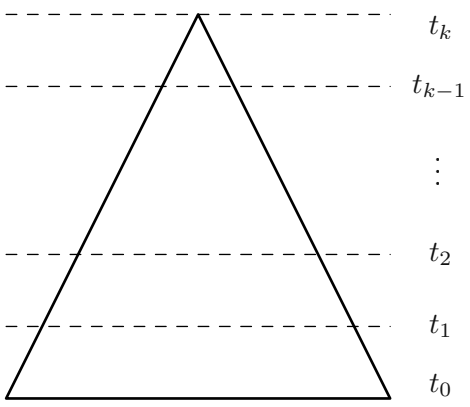
Flattening a set over three calls to `find`.

Path compression helps us amortize the total number of small parent pointers over all `find` operations. Suppose a node x follows a small parent pointer. If the parent is the root we charge that pointer following to the `find` operation. Otherwise, path compression will reset x's parent to some node with strictly higher rank. In particular, if node x is in interval T_i, then x will have a small parent pointer reset t_i times before its pointer becomes big.

For each non-root node x in interval T_i, define

$$\phi(x) = \max\{0, t_i - \text{rank}(\texttt{parent}(x))\}.$$

Let Φ_i be the sum of $\phi(x)$ over all non-root nodes x in T_i, and let $\Phi = \sum_{i=1}^{k} \Phi_i$ be the total potential over all intervals. Each small pointer in a `find` operation, except the last one, decreases Φ by 1, so Φ pays for the small pointers.



When a union makes a root x in interval T_i into a child, it creates at most t_i units of new potential. Each node becomes a child at most once. Since interval T_i contains only

$$O\Big(n/2^{t_{i-1}}\Big)$$

nodes, the total potential ever introduced in interval T_i is

$$O\Big(\frac{t_i}{2^{t_{i-1}}}n\Big),$$

hence

$$\text{total potential ever introduced} = O\bigg(\sum_{i=1}^{k}\frac{t_i}{2^{t_{i-1}}}n\bigg).$$

Let $t_i = \min\{2^{t_{i-1}}, \lfloor \log n \rfloor\}$. Then total potential ever introduced $= O(kn)$, and accounting for both big and small pointers, the total cost of all operations is $O(k(m+n))$. Choose k just large enough that $t_k = \lfloor \log n \rfloor$. Each threshold t_i exponentiates the previous one. So for $k = \log^* n$, $2^{t_{k-1}} \geq \log(n)$, and $t_k = \lfloor \log n \rfloor$. In conclusion, the total cost of all operations is $O((m+n)\log^* n)$.

And beyond! The $\log^* n$ proof used one fixed partition of the entire range of ranks. A sharper analysis gives each node its own thresholds, generated recursively from its rank.

Fix a non-root node x of rank $r \geq 2$. We measure $\text{rank}(\texttt{parent}(x))$ against a hierarchy of scales starting from r. The first few look like

$$\begin{array}{llll} r+1, & r+2, & r+3, & \dots, \quad 2r-1, \\ 2r, & 2^2 r, & 2^3 r, & \dots, \quad 2^{r-1}r, \\ 2^r r, & 2^{2^r r}(2^r r), & 2^{2^{2^r r}(2^r r)}(2^{2^r r}(2^r r)), & \dots, \quad 2^{2^{\cdot^{\cdot^{\cdot}}(\cdots)}}(\cdots), \end{array}$$

By the third scale the formulas are already unwieldy. The pattern is clearer recursively. Define functions $f_0, f_1, f_2, \dots$ of an input t by

$$f_0(t) = t+1,$$
$$f_{k+1}(t) = f_k^{(t)}(t).$$

Scale i consists of the $r-1$ thresholds obtained by repeatedly applying f_i to r.

$$\begin{array}{lllll} \text{scale } 0: & f_0(r) = r+1, & f_0^{(2)}(r) = r+2, & \dots, & f_0^{(r-1)}(r) = 2r-1, \\ \text{scale } 1: & f_1(r) = f_0^{(r)}(r) = 2r, & f_1^{(2)}(r) = 2^2 r, & \dots, & f_1^{(r-1)}(r) = 2^{r-1}r, \\ \text{scale } 2: & f_2(r) = f_1^{(r)}(r) = 2^r r, & f_2^{(2)}(r), & \dots, & f_2^{(r-1)}(r), \\ & \vdots & \vdots & \ddots & \vdots \\ \text{scale } k: & f_k(r) = f_{k-1}^{(r)}(r), & f_k^{(2)}(r), & \dots, & f_k^{(r-1)}(r). \end{array}$$

Each function f_k is increasing.

These thresholds divide the possible values of $\text{rank}(\texttt{parent}(x))$ into intervals. For $\ell \geq 0$ and $1 \leq i \leq r-1$, let

$$I_{\ell,i} = [f_\ell^{(i)}(r), f_\ell^{(i+1)}(r)).$$

The intervals in each scale are increasing, and the last interval in one scale ends where the first interval in the next begins. So for a fixed node x, there are unique values $\ell(x) \geq 0$ and $1 \leq \iota(x) \leq r-1$ such that

$$\text{rank}(\texttt{parent}(x)) \in I_{\ell(x),\iota(x)}.$$

The *level* $\ell(x)$ identifies the scale, and the *index* $\iota(x)$ identifies the interval inside that scale.

Each time path compression gives x a new parent, the parent's rank increases. The level can only increase, and the index can only increase within a fixed level.

A potential function. Let $L \in \mathbb{N}$ be any upper bound on the level of every node of rank at least 2. For each non-root node x of rank at least 2, define

$$\phi(x) = (L - \ell(x)) \,\text{rank}(x) - \iota(x).$$

Let Φ be the sum of $\phi(x)$ over all such nodes.

If x has rank r and level k, then

$$\text{rank}(\texttt{parent}(x)) < f_{k+1}(r) = f_k^{(r)}(r),$$

so $\iota(x) \in \{1, 2, \ldots, r-1\}$. It follows that

$$0 \leq \phi(x) < L \,\text{rank}(x).$$

Whenever a union makes a root into a child, it creates at most $L\,\text{rank}(x)$ units of new potential. Each node becomes a child at most once, so the total potential ever introduced is at most

$$L \sum_x \text{rank}(x) \leq L \sum_{i \geq 0} i \cdot \frac{n}{2^i} = O(nL).$$

`find`. Consider one call to `find`, tracing a path $x_0, x_1, \ldots, x_h$ to the root.

Any traversed node of rank 0 or 1 is charged directly to the operation. There can be at most one node of each rank on the path, so this costs only $O(1)$.

Now look at traversed nodes of rank at least 2. Call such a node x_i *big* if its level is larger than the level of every such node above it on the path. Call it *small* otherwise.

Big nodes are charged directly to the operation. Their levels strictly decrease, so there are at most L of them.

Now let x_i be a small node. Then some node x_j lies above it on the path with $\ell(x_j) \geq \ell(x_i)$. We claim that path compression then decreases $\phi(x_i)$ by at least 1.

We have

$$\operatorname{rank}(x_h) \overset{\text{(a)}}{\geq} \operatorname{rank}(x_{j+1}) \overset{\text{(b)}}{\geq} f_{\ell(x_j)}(\operatorname{rank}(x_j)) \overset{\text{(c)}}{\geq} f_{\ell(x_i)}(\operatorname{rank}(x_j)).$$

Here (a) is by monotonicity of ranks, (b) is by definition of level, and (c) is because higher scales start later.

Also

$$\operatorname{rank}(x_j) \overset{\text{(d)}}{\geq} \operatorname{rank}(x_{i+1}) \overset{\text{(e)}}{\geq} f_{\ell(x_i)}^{(\iota(x_i))}(\operatorname{rank}(x_i)).$$

Here (d) is by monotonicity of ranks and (e) is by definition of index.

Combining these inequalities and using monotonicity of $f_{\ell(x_i)}$,

$$\operatorname{rank}(x_h) \geq f_{\ell(x_i)}\Big(f_{\ell(x_i)}^{(\iota(x_i))}(\operatorname{rank}(x_i))\Big) = f_{\ell(x_i)}^{(\iota(x_i)+1)}(\operatorname{rank}(x_i)).$$

After compression, the parent of x_i becomes x_h. So the new parent rank of x_i lies in a later interval than before. Thus either the level of x_i rises, or the level stays equal and the index rises by at least 1. In either case $\phi(x_i)$ drops by at least 1.

Therefore every traversed node is paid for either by a direct charge or by a unit drop in potential. There are only $O(L)$ direct charges. Each call to `find` has amortized cost $O(L)$.

Since `new-set` takes constant time and `union` is just two calls to `find`, any sequence of m operations on n elements takes total time $O((m+n)L)$.

Choosing L. Since ranks are bounded above by $\log n$ and each f_k is increasing, take

$$L = \min\{k \geq 0 : f_k(2) > \log n\}.$$

Then every node of rank at least 2 has level at most $L-1$. The first few values come from

$$\begin{aligned}
f_0(2) &= 3,\\
f_1(2) &= f_0^{(2)}(2) = 4,\\
f_2(2) &= f_1^{(2)}(2) = 8,\\
f_3(2) &= f_2^{(2)}(2) = f_2(8) = 2048,\\
f_4(2) &= f_3^{(2)}(2) = f_3(2048) = f_2^{(2048)}(2048).
\end{aligned}$$

Any standard inverse of such an *Ackermann-style hierarchy* differs only by a constant and is called an *inverse Ackermann function*, written $\alpha(n)$. So disjoint union with path compression takes

$$O((m+n)\alpha(n))$$

amortized time.

Theorem 23.2 ([Tar75])**.** *There exists a data structure supporting* `union` *and* `find` *over n elements in $\alpha(n)$ amortized time, where $\alpha(n)$ is the inverse Ackermann function.*