Algorithm Theory

Chapter 5
Data Structures

Fibonacci Heaps

Fabian Kuhn

Priority Queue / Heap

- Stores (*key,data*) pairs
  - like a dictionary, but with a different set of operations

- **Initialize-Heap**: creates new empty heap
- **Is-Empty**: returns true if heap is empty
- **Insert**(*key,data*): inserts (*key,data*)-pair, returns pointer to entry
- **Get-Min**: returns (*key,data*)-pair with minimum *key*
- **Delete-Min**: deletes (and returns) minimum (*key,data*)-pair
  - has to be consistent with get-min operation
- **Decrease-Key**(*entry,newkey*): decreases *key* of *entry* to *newkey*
- **Merge**: merges two heaps into one

Implementation of Dijkstra's Algorithm

Dijkstra's Algorithm:

1. Initialize $d(s,s) = 0$ and $d(s,v) = \infty$ for all $v \neq s$
2. All nodes $v \neq s$ are unmarked

create empty priority queue Q,
add all nodes to Q with initial key $d(s,v)$

d nodes,
imates $d(s,v)$

3. Get unmarked node u which minimizes $d(s,u)$:
4. mark node u

$u \coloneqq Q$.delete_min()

vith minimum $d(s,u)$,
delete u from DS

unmarked v

5. For all $e = \{u,v\} \in E$, $d(s,v) = \min\{d(s,v), d(s,u) + w(e)\}$

For all unmarked neighbors v of u: potentially call Q.decrease_key

6. Until all nodes are marked

until Q is empty

Implementation of Prim/Jarník Algorithm

Start at node s, very similar to Dijkstra's algorithm :

1. Initialize $d(s) = 0$ and $d(v) = \infty$ for all $v \neq s$
2. All nodes $v \neq s$ are unmarked

   create empty priority queue Q,
   add all nodes to Q with initial key $d(v)$

3. Get unmarked node u which minimizes $d(u)$:
4. mark node u

   $u :=$ Q.delete_min()

   unmarked v

5. For all $e = \{u, v\} \in E$, $d(v) = \min\{d(v), w(e)\}$

   For all unmarked neighbors v of u: potentially call Q.decrease_key

6. Until all nodes are marked

   until Q is empty

Analysis

Number of priority queue operations for Dijkstra:

- Initialize-Heap: $\mathbf{1}$
- Is-Empty: $\boldsymbol{n}$
- **Insert:** $\boldsymbol{n}$
- Get-Min: $\mathbf{0}$
- **Delete-Min:** $\boldsymbol{n}$
- **Decrease-Key:** $\leq \boldsymbol{m}$
- Merge: $\mathbf{0}$

Assumption:

$n = |V|$ (number of nodes)
$m = |E|$ (number of edges)

- $m \geq n - 1$

#Decrease-Key:

Always for an unmarked neighbor v of a newly marked node u

$\Longrightarrow \leq 1$ decrease-key per edge

Can We Do Better?

- Cost of **Dijkstra** with **complete binary min-heap** implementation:

$$O(m \cdot \log n)$$

- **Binary heap:**
  insert, delete-min, and decrease-key cost $O(\log n)$

- One of the operations insert or delete-min must cost $\Omega(\log n)$:
  - Heap-Sort:
    Insert n elements into heap, then take out the minimum n times
  - (Comparison-based) sorting costs at least $\Omega(n \log n)$.

- But maybe we can improve decrease-key and one of the other two operations?

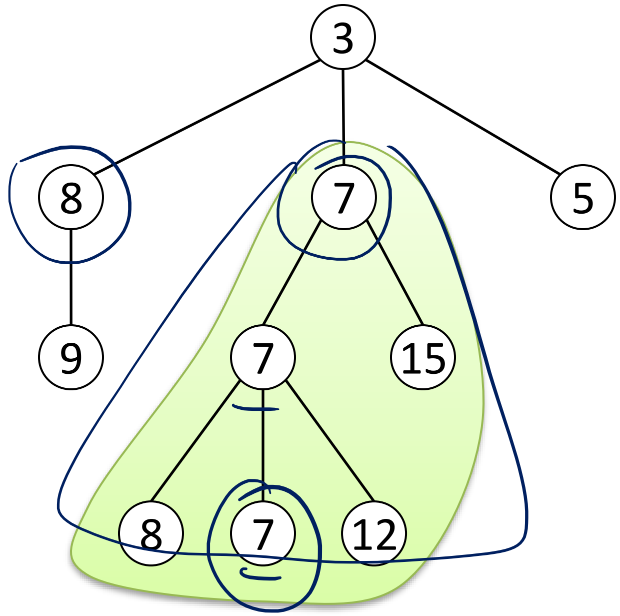
Fibonacci Heaps

Structure:

A Fibonacci heap H consists of a collection of trees satisfying the **min-heap** property.

Min-Heap Property:

Key of a node $v \leq$ keys of all nodes in any sub-tree of v



Fibonacci Heaps

Structure:

A Fibonacci heap H consists of a collection of trees satisfying the min-heap property.

Variables:

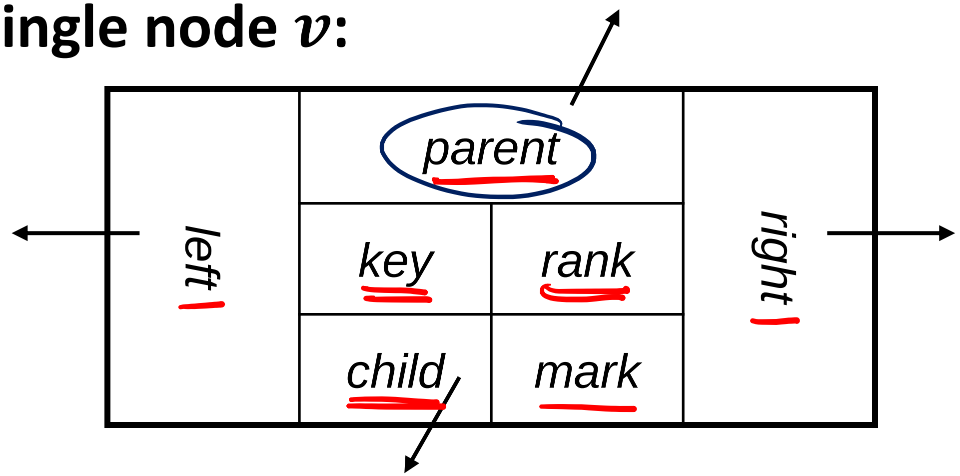
- $H.min$: root of the tree containing the (a) minimum key
- $H.rootlist$: circular, doubly linked, unordered list containing the roots of all trees
- $H.size$: number of nodes currently in H

Lazy Merging:

- To reduce the number of trees, sometimes, trees need to be merged
- Lazy merging: Do not merge as long as possible...

Trees in Fibonacci Heaps

Structure of a single node v:



- $v.child$: points to circular, doubly linked and unordered list of the children of v
- $v.left, v.right$: pointers to siblings (in doubly linked list)
- $v.mark$: will be used later…

Advantages of circular, doubly linked lists:

- Deleting an element takes constant time
- Concatenating two lists takes constant time

Example

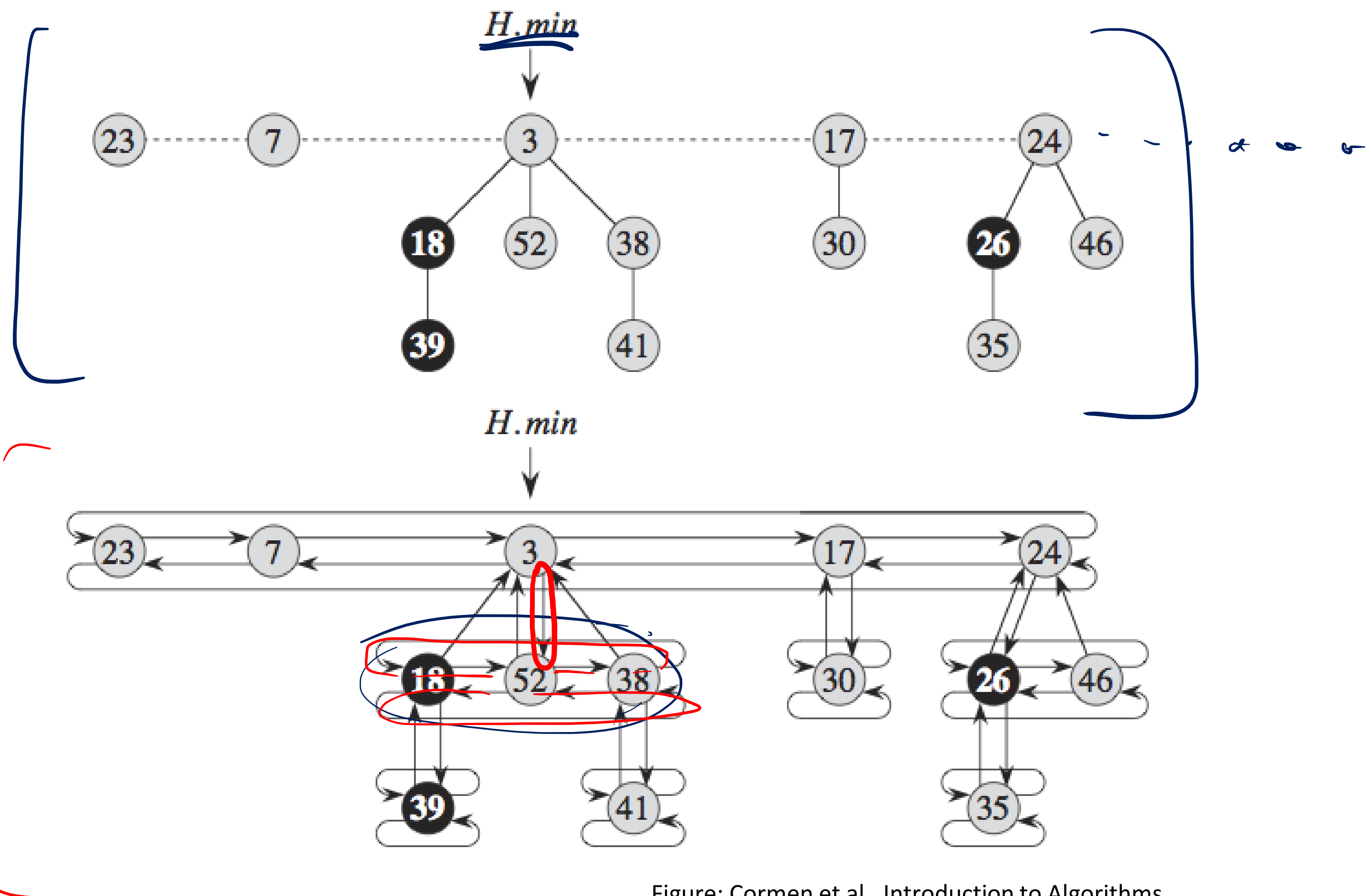


Figure: Cormen et al., Introduction to Algorithms

Simple (Lazy) Operations

Initialize-Heap H:

- $H.rootlist \coloneqq H.min \coloneqq null$

Merge heaps H and H':

- concatenate root lists
- update $H.min$

Insert element e into H:

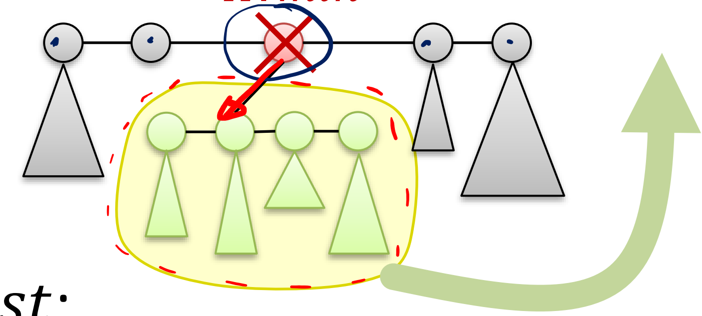
- create new one-node tree containing e → H'
  - mark of root node is set to **false**
- merge heaps H and H'

Get minimum element of H:

- return $H.min$

Operation Delete-Min

Delete the node with minimum key from H and return its element:



1. $m \coloneqq H.min;$
2. **if** $H.size > 0$ **then**
3. remove $H.min$ from $H.rootlist;$
4. add $H.min.child$ (list) to $H.rootlist$
5. $\boldsymbol{H.Consolidate();}$

// Repeatedly merge nodes with equal degree in the root list
// until degrees of nodes in the root list are distinct.
// Determine the element with minimum key

6. **return** m

Rank and Maximum Degree

Ranks of nodes, trees, heap:

Node v:

- $rank(v)$: number of children of v (degree of v)

Tree T:

- $rank(T)$: rank (degree) of root node of T

Heap H:

- $rank(H)$: maximum degree (#children) of any node in H

Assumption (n: number of nodes in H):

$$rank(H) \leq D(n)$$

- for a known function $D(n)$

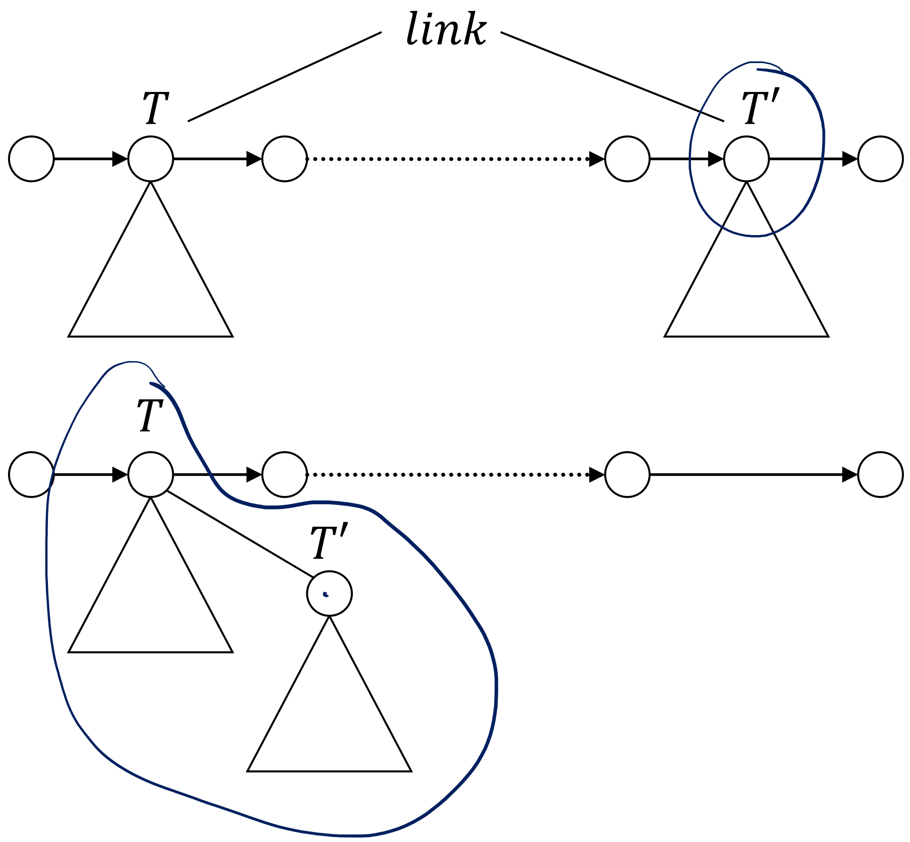
Merging Two Trees

Given: Heap-ordered trees T, T' with $rank(T) = rank(T')$

- Assume: min-key of T < min-key of T'

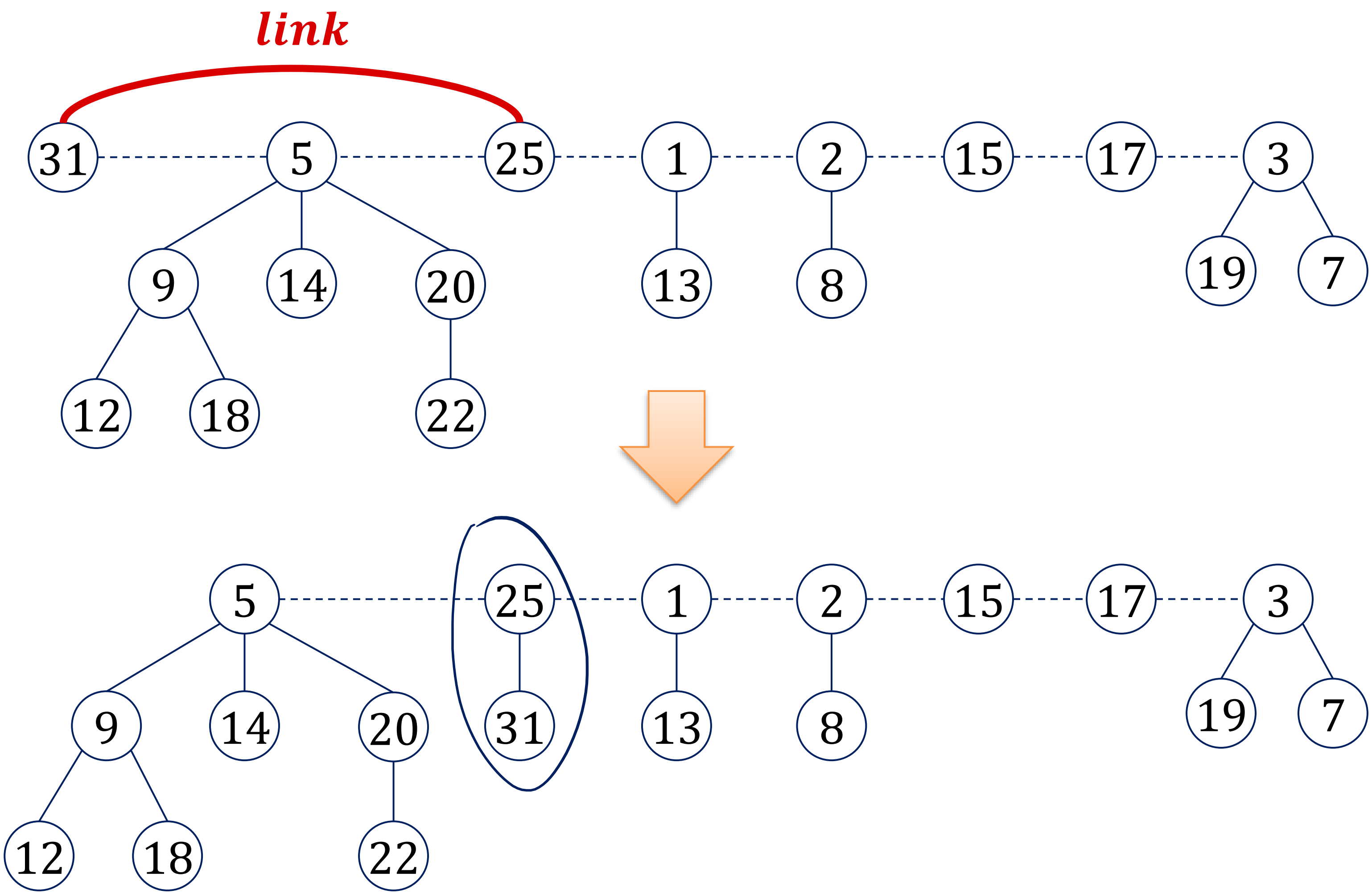
Operation $\boldsymbol{link(T, T')}$:

- Removes tree T' from root list and adds T' to child list of T

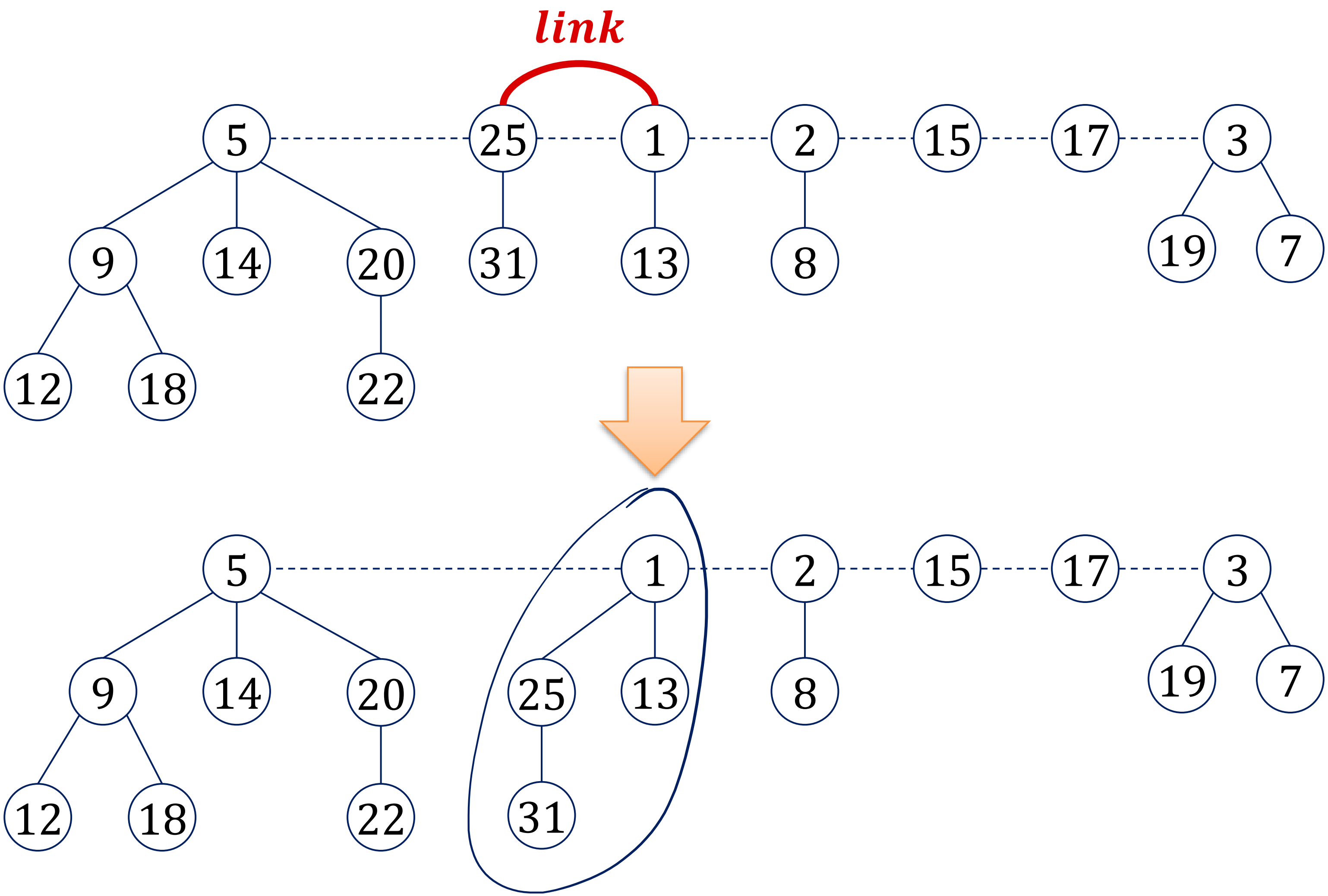


- $rank(T) \coloneqq rank(T) + 1$
- $(T'.mark = \textbf{false})$

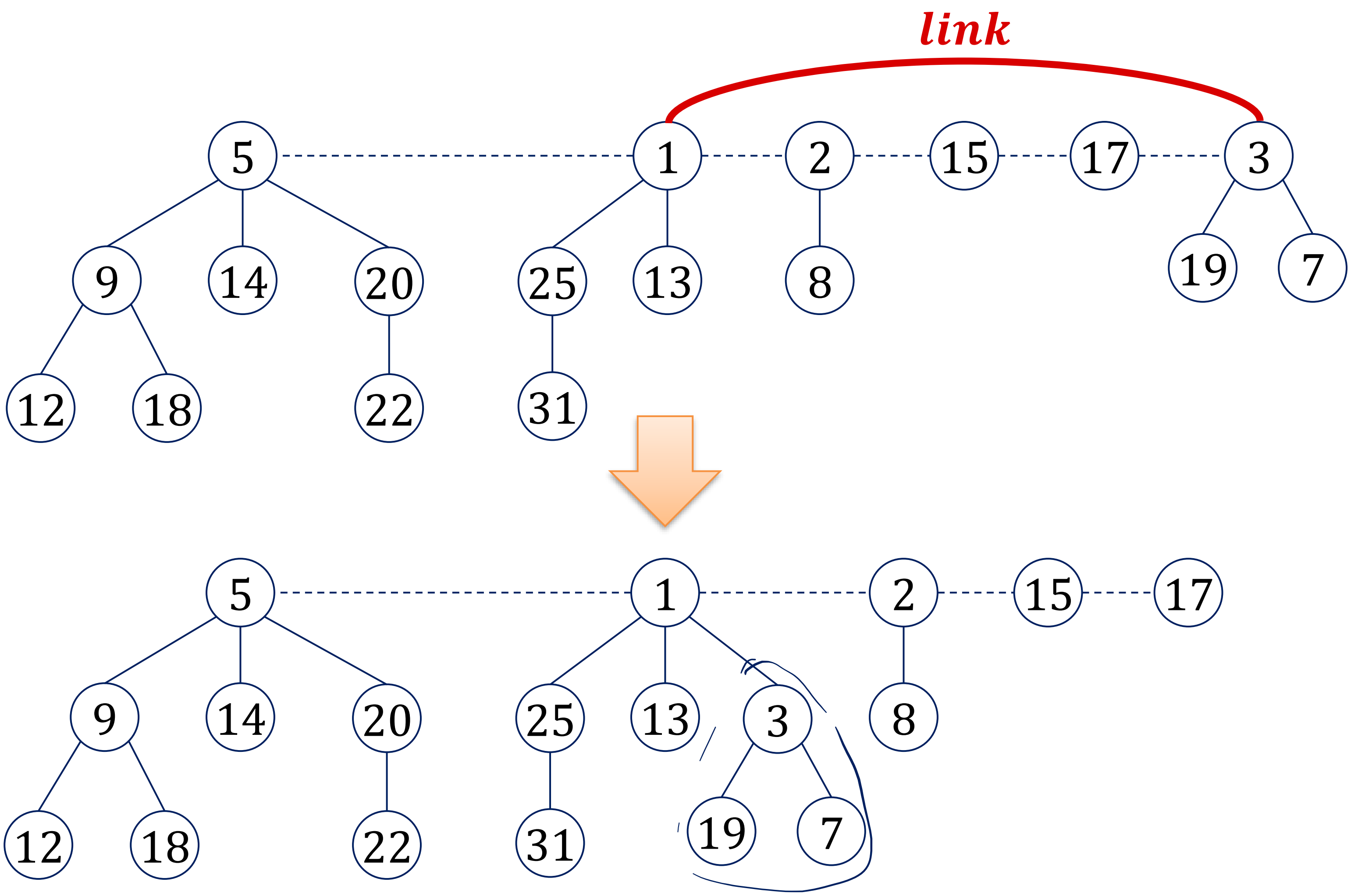
Consolidate Example



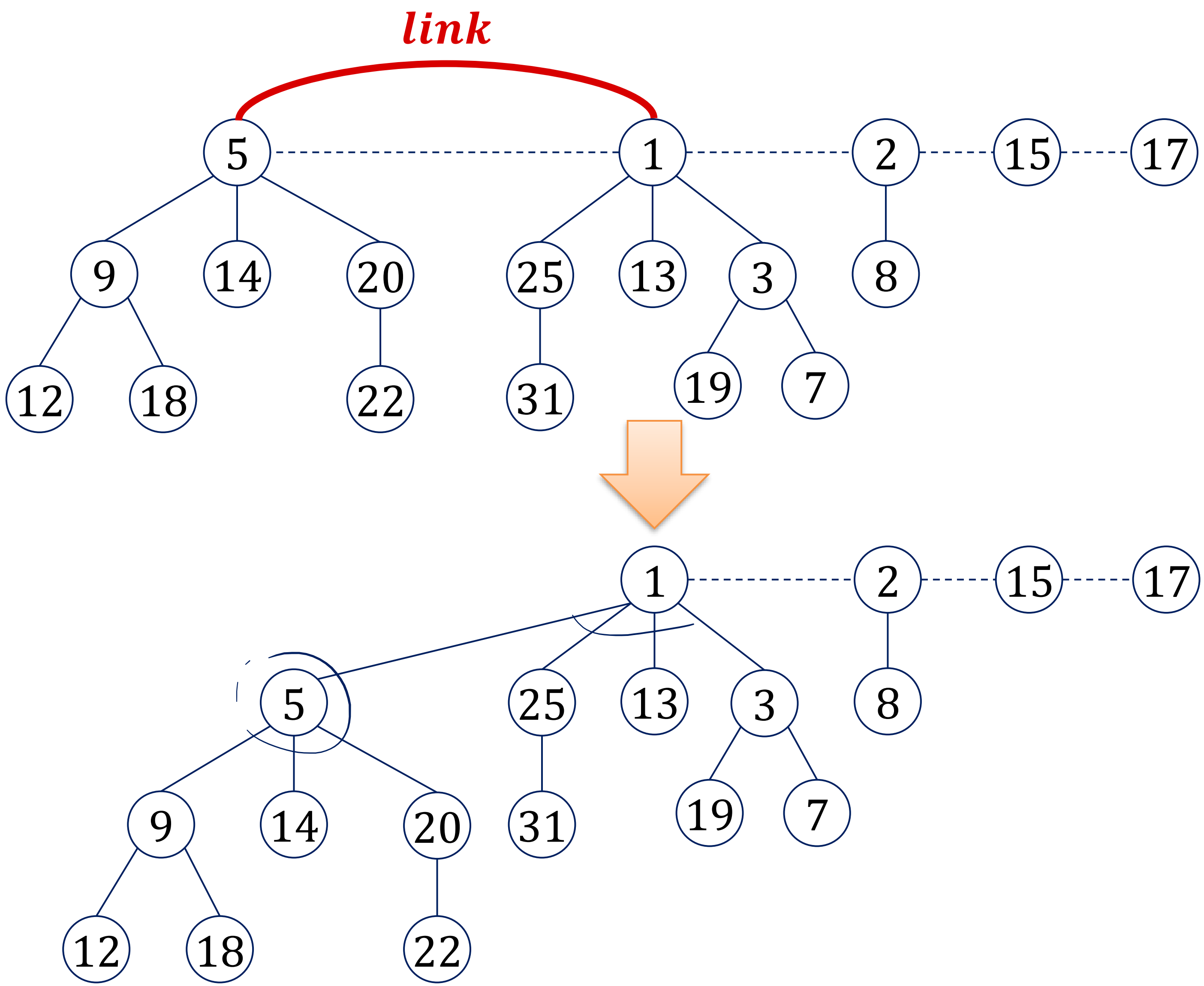
Consolidate Example



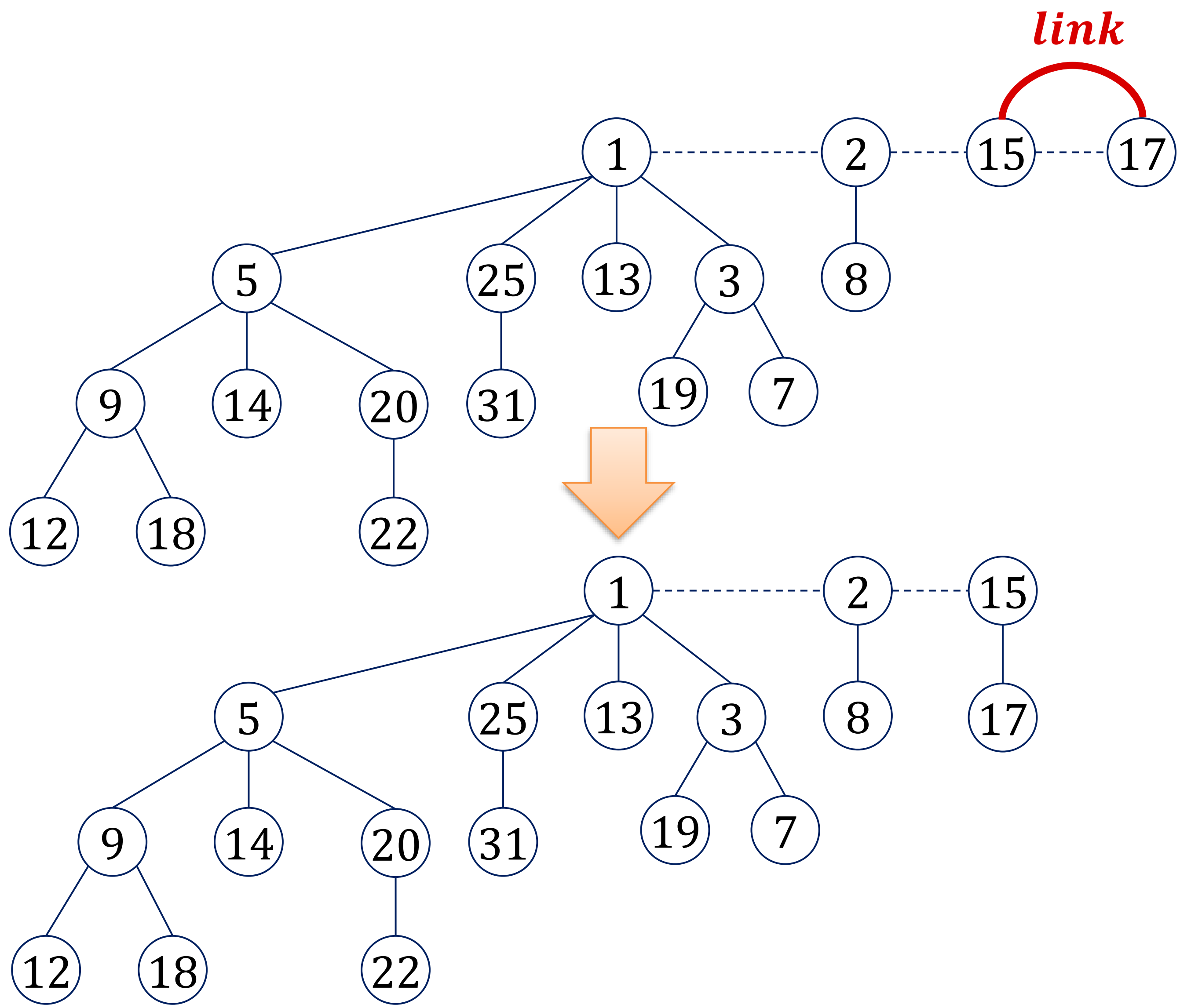
Consolidate Example



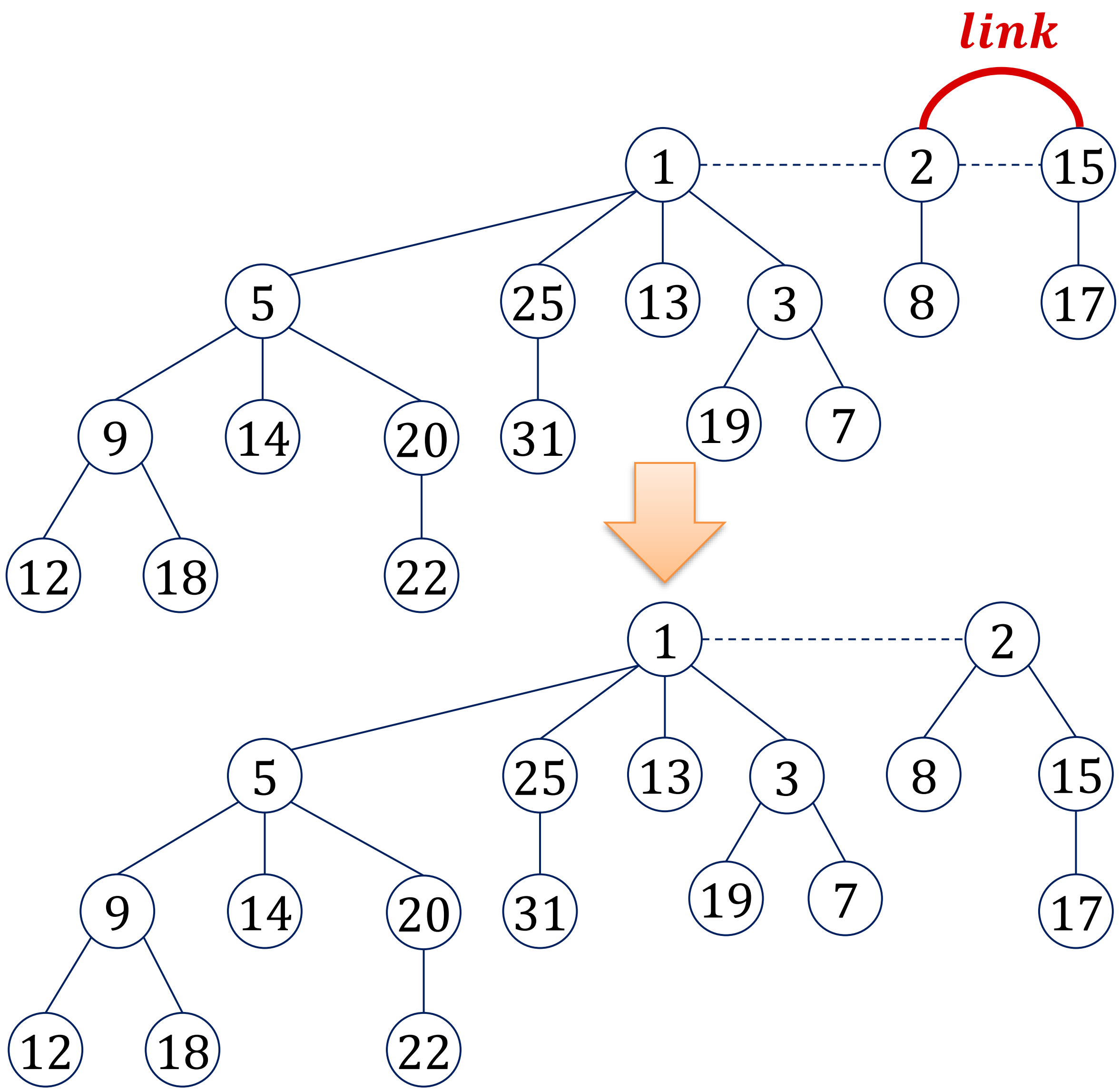
Consolidate Example



Consolidate Example



Consolidate Example



Consolidation of Root List

Array A pointing to find roots with the same rank:

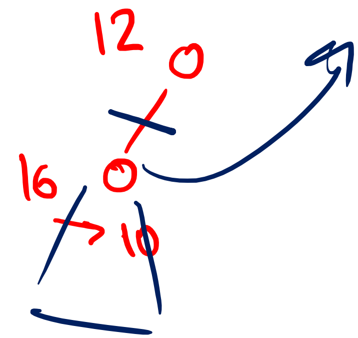
Consolidate:

```
for i := 0 to D(n) do A[i] := null
while H.rootlist ≠ null do
    T := "delete and return first element of H.rootlist"
    while A[rank(T)] ≠ null do
        T' := A[rank(T)]
        A[rank(T)] := null
        T := link(T, T')
    A[rank(T)] := T
Create new H.rootlist and H.min
```

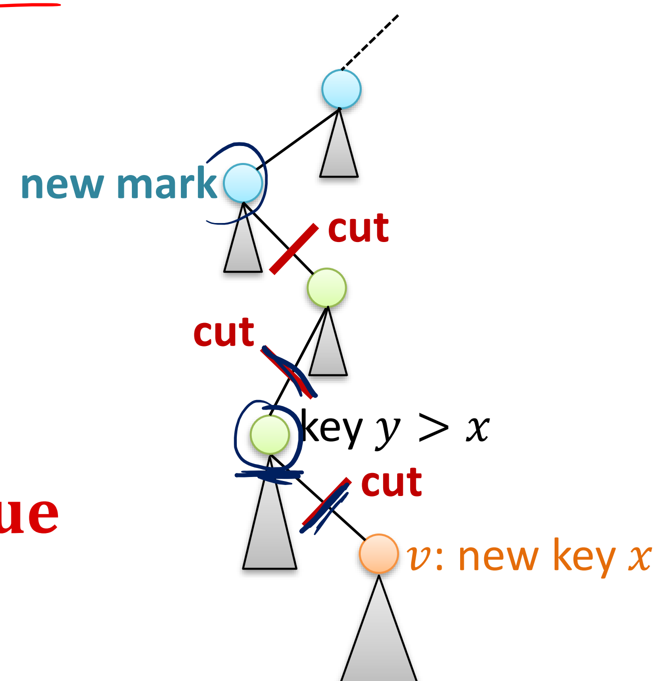
Time:
$O(|H.rootlist| + D(n))$

Operation Decrease-Key

Decrease-Key(v, x): (decrease key of node v to new value x)



if $x \geq v.key$ **then return**
$v.key := x$;
update $H.min$ to point to v if necessary
if $v \in H.rootlist \;\vee\; x \geq v.parent.key$ **then return**
repeat
    $parent := v.parent$
    $\boldsymbol{H.cut(v)}$
    $v := parent$
until $\neg(\boldsymbol{v.mark}) \;\vee\; v \in H.rootlist$
if $v \notin H.rootlist$ **then** $\boldsymbol{v.mark := \text{true}}$



Operation Cut(v)

Operation $H.cut(v)$:

- Cuts v's sub-tree from its parent and adds v to rootlist

if $v \notin H.rootlist$ **then**

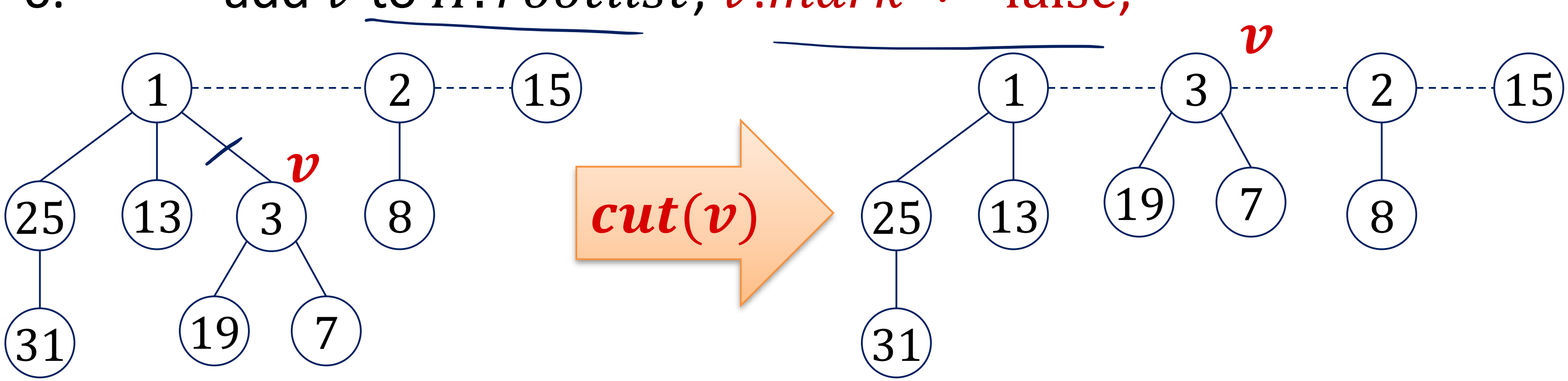
// cut the link between v and its parent

$rank(v.parent) := rank(v.parent) - 1;$

remove v from $v.parent.child$ (list)

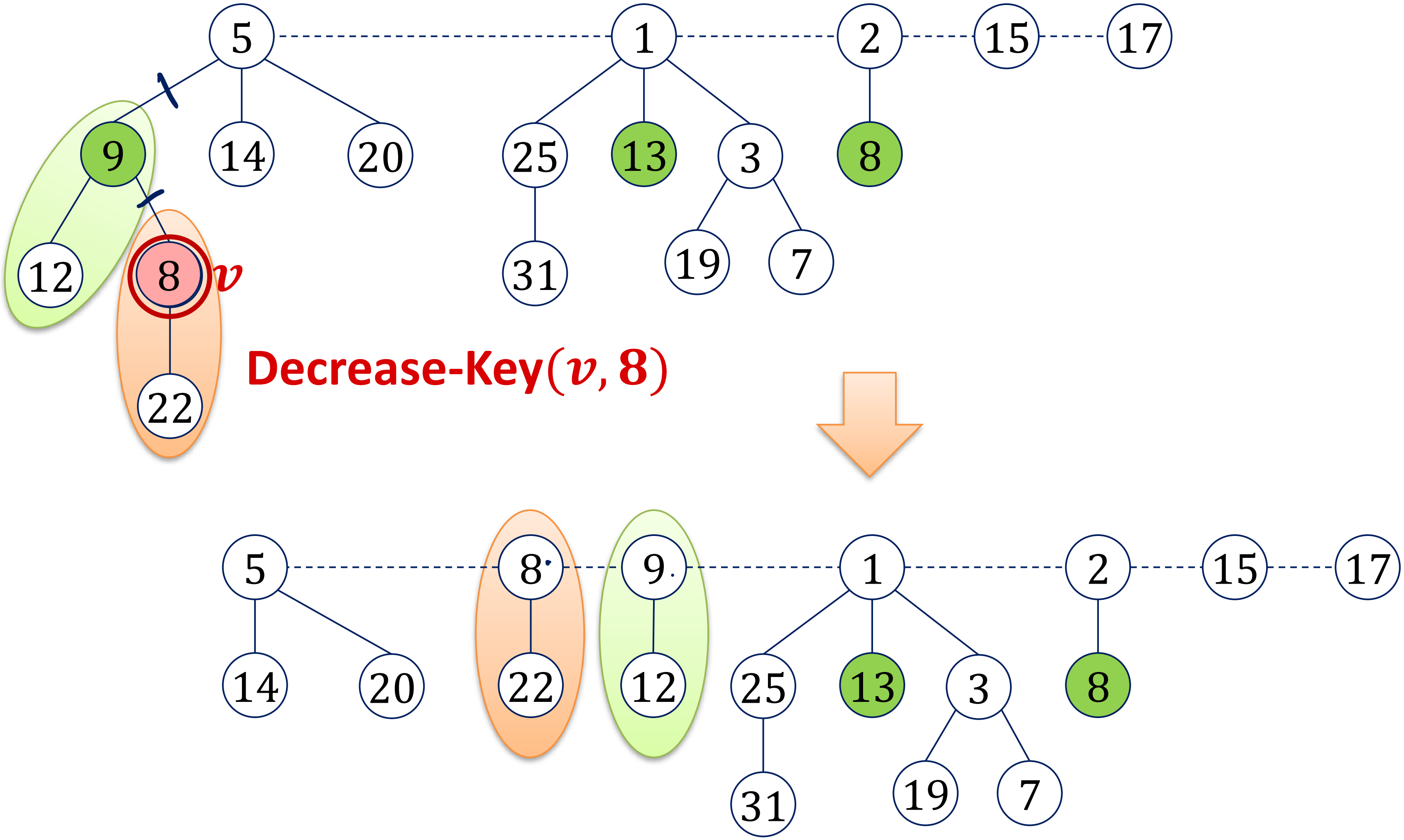
$v.parent := \text{null};$

add v to $H.rootlist$; $v.mark := \text{false};$



Decrease-Key Example

- Green nodes are marked



Fibonacci Heaps Marks

- Nodes in the root list (the tree roots) are always unmarked
  → If a node is added to the root list (insert, decrease-key), the mark of the node is set to false.

- Nodes not in the root list can only get marked when a subtree is cut in a decrease-key operation

- A node v is marked if and only if v is not in the root list and v has lost a child since v was attached to its current parent
  - a node can only change its parent by being moved to the root list

Fibonacci Heap Marks

History of a node v:

v is being linked to a node → $v.mark = \text{false}$

a child of v is cut → $v.mark \coloneqq \text{true}$

a second child of v is cut → $H.cut(v);$
$v.mark \coloneqq false$

- Hence, the boolean value $v.mark$ indicates whether node v has lost a child since the last time v was made the child of another node.
- Nodes v in the root list always have $v.mark = \text{false}$

Cost of Delete-Min & Decrease-Key

Delete-Min:

1. Delete min. root r and add $r.child$ to $H.rootlist$

   time: $O(1)$

2. Consolidate $H.rootlist$

   time: $O(\text{length of } H.rootlist + D(n))$

- Step 2 can potentially be linear in n (size of H)

Decrease-Key (at node v):

1. If new key < parent key, cut sub-tree of node v

   time: $O(1)$

2. Cascading cuts up the tree as long as nodes are marked

   time: $O(\text{number of consecutive marked nodes})$

- Step 2 can potentially be linear in n

Remark: Both operations can take $\Theta(n)$ time in the worst case!

Cost of Delete-Min & Decrease-Key

- Cost of delete-min and decrease-key can be $\Theta(n)$…
  - Seems a large price to pay to get insert in $O(1)$ time

- Maybe, the operations are efficient most of the time?
  - It seems to require a lot of operations to get a long rootlist and thus, an expensive consolidate operation
  - In each decrease-key operation, at most one node gets marked: We need a lot of decrease-key operations to get an expensive decrease-key operation

- Can we show that the average cost per operation is small?

  $\Longrightarrow$ requires **amortized analysis**

Amortized Cost of Fibonacci Heaps

- **Initialize-heap**, **is-empty**, **get-min**, **insert**, and **merge** have **worst-case** and **amortized cost $O(1)$**
- **Delete-min** has **amortized cost $O(\log n)$**
- **Decrease-key** has **amortized cost $O(1)$**
- Starting with an empty heap, any sequence of n operations with at most n_d delete-min operations has total cost (time)

$$T = O(n + n_d \log n).$$

Dijkstra: $O(m + n \log n)$

- We will now need the marks…
- Cost for Dijkstra & Prim/Jarník: $O(m + n \log n)$

Fibonacci Heaps: Marks

Cycle of a node:

1. Node v is removed from root list and linked to a node

   $\boldsymbol{v.mark = \text{false}}$

2. Child node u *of* v is cut and added to root list

   $\boldsymbol{v.mark := \text{true}}$

3. Second child of v is cut

   node v is cut as well and moved to root list

   $\boldsymbol{v.mark := \text{false}}$

The boolean value $v.mark$ indicates whether node v has lost a child since the last time v was made the child of another node.

Potential Function

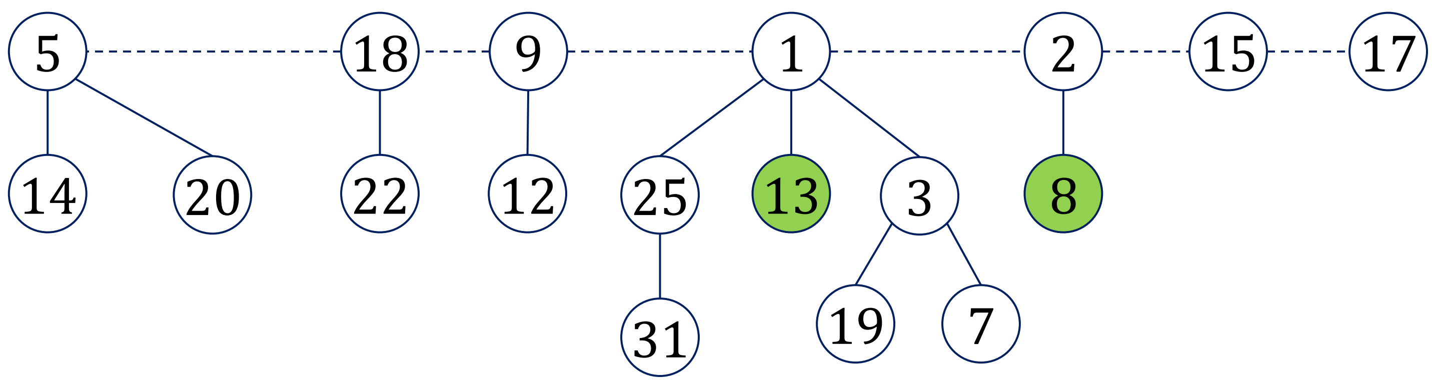
System state characterized by two parameters:

- ***R***: number of trees (length of $H.rootlist$)
- ***M***: number of marked nodes (not in the root list)

Potential function:

$$\Phi := R + 2M$$

Example:



- $R = 7, M = 2 \;\rightarrow\; \Phi = 11$

Actual Time of Operations

- Operations: ***initialize-heap***, ***is-empty***, ***insert***, ***get-min***, ***merge***

  actual time: $O(1)$

  – Normalize unit time such that

  $$t_{init}, t_{is-empty}, t_{insert}, t_{get-min}, t_{merge} \leq 1$$

- Operation ***delete-min***:
  – Actual time: $O(\text{length of } H.rootlist + D(n))$
  – Normalize unit time such that

  $$t_{del-min} \leq D(n) + \text{ length of } H.rootlist$$

- Operation **descrease-key**:
  – Actual time: $O(\text{length of path to next unmarked ancestor})$
  – Normalize unit time such that

  $$t_{decr-key} \leq \text{length of path to next unmarked ancestor}$$

Amortized Times

Assume operation i is of type:

- **initialize-heap:**
  - actual time: $t_i \leq 1$, potential: $\Phi_{i-1} = \Phi_i = 0$
  - amortized time: $a_i = t_i + \Phi_i - \Phi_{i-1} \leq 1$
- **is-empty, get-min:**
  - actual time: $t_i \leq 1$, potential: $\Phi_i = \Phi_{i-1}$ (heap doesn't change)
  - amortized time: $a_i = t_i + \Phi_i - \Phi_{i-1} \leq 1$
- **merge:**
  - Actual time: $t_i \leq 1$
  - combined potential of both heaps: $\Phi_i = \Phi_{i-1}$
  - amortized time: $a_i = t_i + \Phi_i - \Phi_{i-1} \leq 1$

Amortized Time of Insert

Assume that operation i is an *insert* operation:

- **Actual time:** $t_i \leq 1$
- **Potential function:**
  - M remains unchanged (no nodes are marked or unmarked, no marked nodes are moved to the root list)
  - R grows by 1 (one element is added to the root list)

$$M_i = M_{i-1}, \qquad R_i = R_{i-1} + 1$$
$$\Phi_i = \Phi_{i-1} + 1$$

- **Amortized time:**

$$\boldsymbol{a_i = t_i + \Phi_i - \Phi_{i-1} \leq 2}$$

Amortized Time of Delete-Min

Assume that operation i is a *delete-min* operation:

Actual time: $t_i \leq D(n) + |H.rootlist|$ ← R_{i-1}

Potential function $\Phi = R + 2M$:

- R: changes from $|H.rootlist|$ to at most $D(n) + 1$
- M: (# of marked nodes that are not in the root list)
  - Number of marks does not increase

$$M_i = M_{i-1}, \qquad R_i \leq R_{i-1} + D(n) + 1 - |H.rootlist|$$

$$\Phi_i \leq \Phi_{i-1} + D(n) + 1 - |H.rootlist|$$

$\Phi_i = R_i + 2M_i \leq \underbrace{R_{i-1} + 2M_{i-1}}_{\Phi_{i-1}} + D(n) + 1 - |H.rootlist|$

Amortized Time:

$$a_i = t_i + \Phi_i - \Phi_{i-1} \leq 2D(n) + 1$$

$D(n) + |H.rl| + D(n) + 1 - |H.rl|$

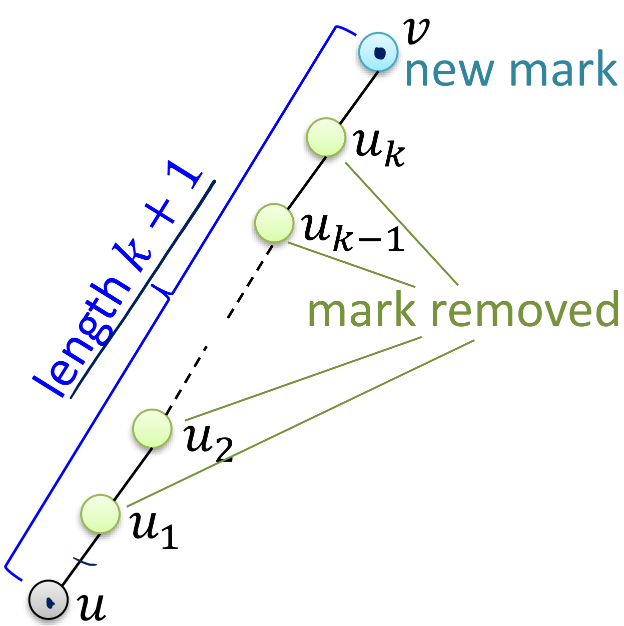
Amortized Time of Decrease-Key

Assume that operation i is a *decrease-key* operation at node u:

Actual time: $t_i \leq$ length of path to next unmarked ancestor v

Potential function $\Phi = R + 2M$: $\phi_i - \phi_{i-1} \leq k+1 + 2(-(k-1)) = -k+3$

- Assume, node u and nodes $u_1, \dots, u_k$ are moved to root list
  - $u_1, \dots, u_k$ are marked and moved to root list, v.mark is set to true



marks	root list
Removed marks: $u_1, \dots, u_k$ (and u, if u is marked) Added mark: v $M_i - M_{i-1} \leq -(k-1)$	Added to root list: $u, u_1, \dots, u_k$ $R_i - R_{i-1} = k+1$

Amortized Time of Decrease-Key

Assume that operation i is a *decrease-key* operation at node u:

Actual time: $t_i \leq$ length of path to next unmarked ancestor v

Potential function $\boldsymbol{\Phi = R + 2M}$:

- Assume, node u and nodes $u_1, \dots, u_k$ are moved to root list
  - $u_1, \dots, u_k$ are marked and moved to root list, $v.\text{mark}$ is set to true
- $\geq k$ marked nodes go to root list, ≤ 1 node gets newly marked
- R grows by $\leq k + 1$, M grows by 1 and is decreased by $\geq k$

$$R_i \leq R_{i-1} + k + 1, \qquad M_i \leq M_{i-1} + 1 - k$$
$$\Phi_i \leq \Phi_{i-1} + (k + 1) - 2(k - 1) = \Phi_{i-1} + 3 - k$$

Amortized time:

$$\boldsymbol{a_i = t_i + \Phi_i - \Phi_{i-1} \leq k + 1 + 3 - k = 4}$$

Complexities Fibonacci Heap

- **Initialize-Heap**: $\boldsymbol{O(1)}$
- **Is-Empty**: $\boldsymbol{O(1)}$
- **Insert**: $\boldsymbol{O(1)}$
- **Get-Min**: $\boldsymbol{O(1)}$
- **Delete-Min**: $\boldsymbol{O(D(n))}$
- **Decrease-Key**: $\boldsymbol{O(1)}$

amortized

- **Merge** (heaps of size m and n, $m \leq n$): $\boldsymbol{O(1)}$

- **How large can $D(n)$ get?**

Rank of Children

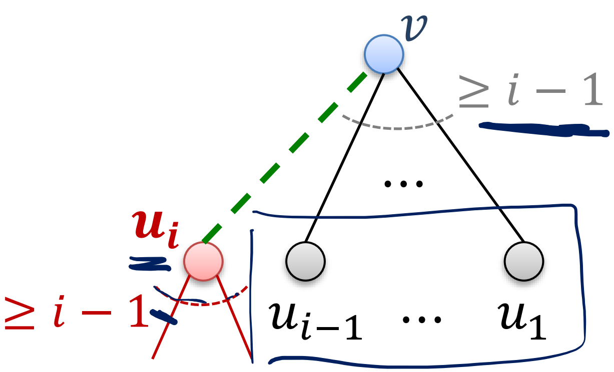
Lemma:

Consider a node v of rank k and let $u_1, \ldots, u_k$ be the children of v in the order in which they were linked to v. Then,

$$\boldsymbol{rank(u_i) \geq i - 2}.$$

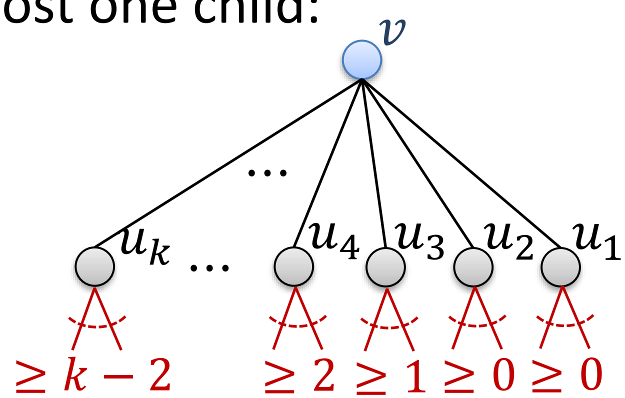
Proof:

When u_i is added, v already has children $u_1, \ldots, u_{i-1}$:



$\Longrightarrow rank(u_i) \geq i - 1$ when u_i is linked to v.

Each node can lose at most one child:



$\Longrightarrow rank(u_i) \geq i - 2$ as long as u_i is linked to v.

Size of Trees

Fibonacci Numbers:

$$F_0 = 0, \qquad F_1 = 1, \qquad \forall k \geq 2: F_k = F_{k-2} + F_{k-1}$$

Lemma:

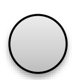
In a Fibonacci heap, the size of the sub-tree of a node v with rank k is at least F_{k+2}.

$S_k \geq F_{k+2}$

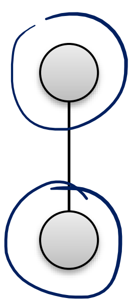
Proof:

- S_k: minimum size of the sub-tree of a node of rank k

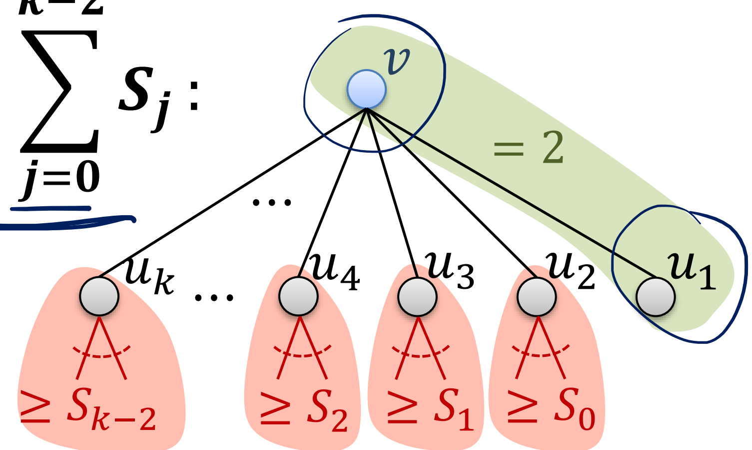
$\boldsymbol{S_0 = 1}$:



$\boldsymbol{S_1 = 2}$:



$$\boldsymbol{S_k \geq 2 + \sum_{j=0}^{k-2} S_j}:$$



Size of Trees

$$S_0 = 1, \qquad S_1 = 2, \qquad \forall k \geq 2: S_k \geq 2 + \sum_{i=0}^{k-2} S_i$$

Claim about Fibonacci numbers:

$$\forall k \geq 0: F_{k+2} = 1 + \sum_{i=0}^{k} F_i \qquad (F_0 = 0, F_1 = 1)$$

Proof of claim (by induction on k):

- **Base case ($k = 0$):** $$F_2 = 1 + \sum_{i=0}^{0} F_i = 1 + F_0 = 1$$
- **Induction step ($k > 0$):**

$$F_{k+2} = F_k + F_{k+1}$$

I.H.: $F_{k+1} = 1 + \sum_{i=0}^{k-1} F_i$

Size of Trees

$$S_0 = 1, S_1 = 2, \forall k \geq 2: S_k \geq 2 + \sum_{i=0}^{k-2} S_i, \qquad F_{k+2} = 1 + \sum_{i=0}^{k} F_i$$

Claim of lemma: $S_k \geq F_{k+2}$

Proof by induction on k:

- Base case ($k = 0, k = 1$): $S_0 \geq F_2 = 1$ $\quad S_1 \geq F_3 = 2$
- Induction step ($k > 1$):

$$S_k \geq 2 + \sum_{i=0}^{k-2} S_i \geq 2 + \sum_{i=0}^{k-2} F_{i+2} = 2 + \sum_{j=2}^{k} F_j = 1 + \sum_{j=0}^{k} F_j = F_{k+2}$$

I.H.

$j = i + 2$

$F_0 = 0, F_1 = 1$

previous claim on Fibonacci numbers

Size of Trees

Lemma:
In a Fibonacci heap, the size of the sub-tree of a node v with rank k is at least F_{k+2}.

Theorem:
The maximum rank of a node in a Fibonacci heap of size n is at most

$$\boldsymbol{D(n) = O(\log n)}.$$

Proof:

- The Fibonacci numbers grow exponentially:

$$F_k = \frac{1}{\sqrt{5}} \cdot \left(\left(\frac{1+\sqrt{5}}{2} \right)^k - \left(\frac{1-\sqrt{5}}{2} \right)^k \right)$$

- For $D(n) \geq k$, we need $n \geq F_{k+2}$ nodes.

Binary Heaps & Fibonacci Heaps

	Binary Heap	Fibonacci Heap
initialize	$O(1)$	$O(1)$
insert	$O(\log n)$	$O(1)$
get-min	$O(1)$	$O(1)$
delete-min	$O(\log n)$	$O(\log n)$ *
decrease-key	$O(\log n)$	$O(1)$ *
is-empty	$O(1)$	$O(1)$

* **amortized time**