Chapter 4
Data Structures

Algorithm Theory
WS 2012/13

Fabian Kuhn

Examples

Dictionary:

- Operations: insert(*key,value*), delete(*key*), find(*key*)
- Implementations:
  - Linked list: all operations take $O(n)$ time (n: size of data structure)
  - Balanced binary tree: all operations take $O(\log n)$ time
  - Hash table: all operations take $O(1)$ times (with some assumptions)

Stack (LIFO Queue):

- Operations: push, pull
- Linked list: $O(1)$ for both operations

(FIFO) Queue:

- Operations: enqueue, dequeue
- Linked list: $O(1)$ time for both operations

Here: **Priority Queues (heaps)**, **Union-Find data structure**

Dijkstra's Algorithm

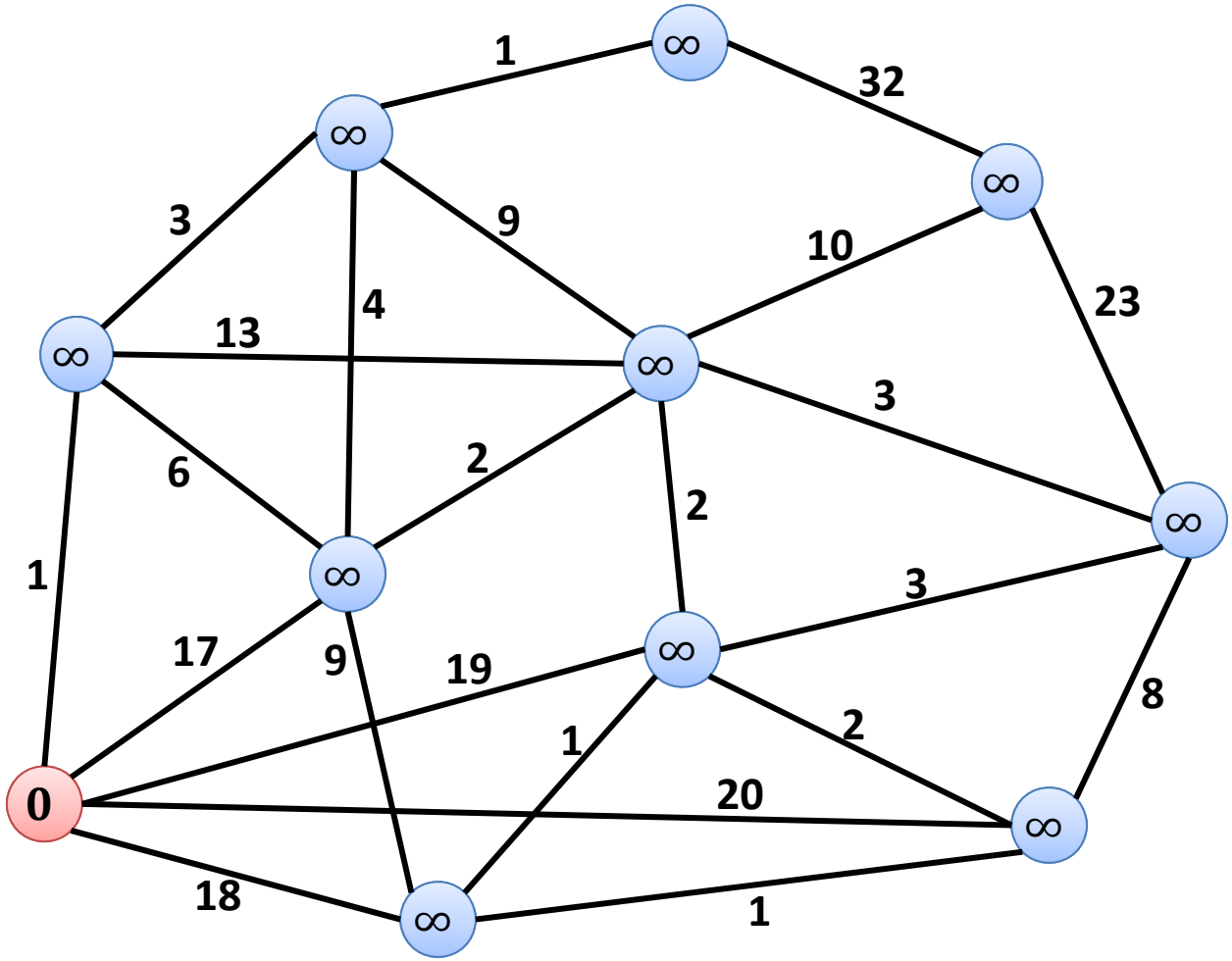
Single-Source Shortest Path Problem:

- **Given:** graph $G = (V, E)$ with edge weights $w(e) \geq 0$ for $e \in E$
  source node $s \in V$
- **Goal:** compute shortest paths from s to all $v \in V$

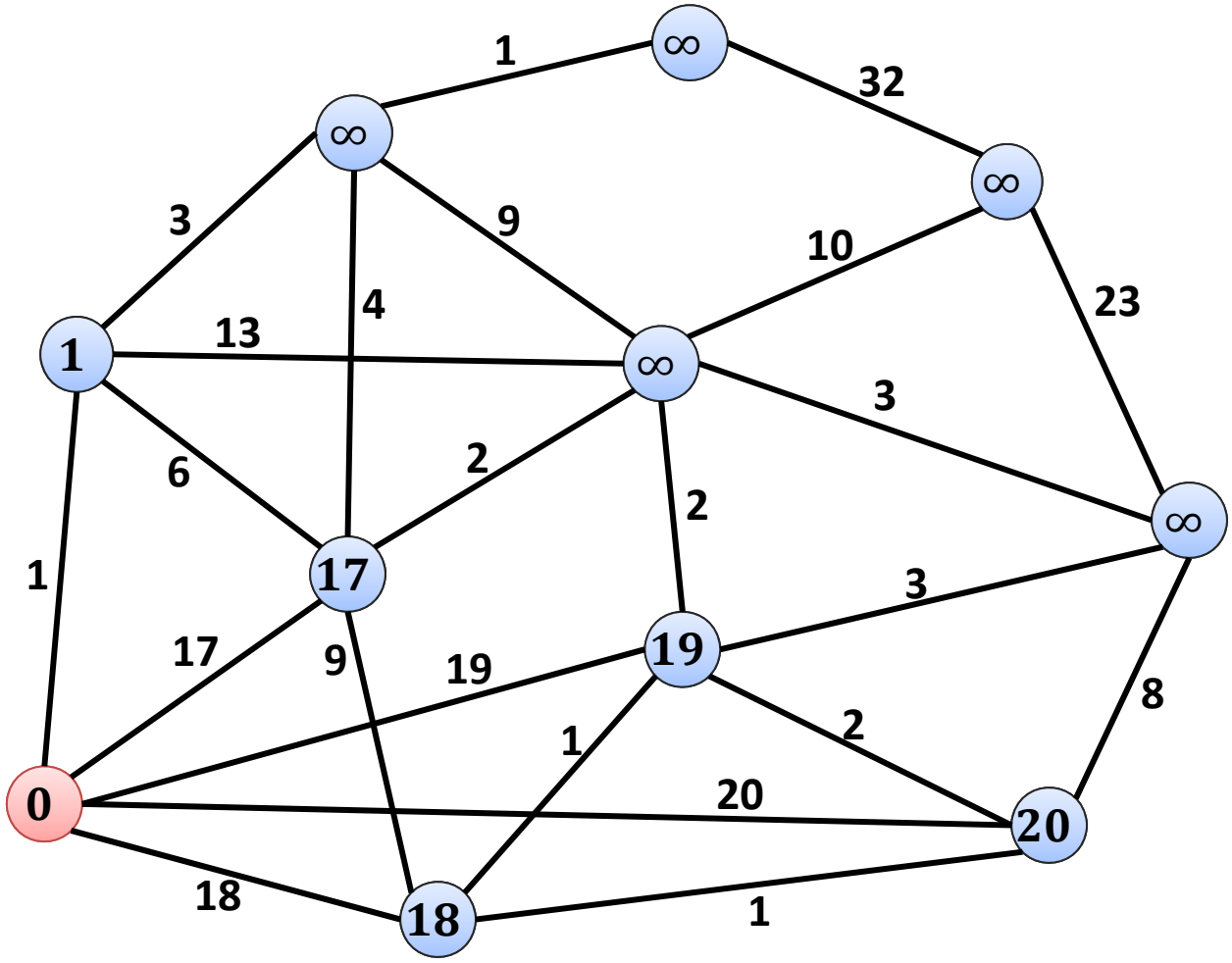
Dijkstra's Algorithm:

1. Initialize $d(s,s) = 0$ and $d(s,v) = \infty$ for all $v \neq s$
2. All nodes are unmarked
3. Get unmarked node u which minimizes $d(s,u)$:
4. For all $e = \{u,v\} \in E$, $d(s,v) = \min\{d(s,v), d(s,u) + w(e)\}$
5. mark node u
6. Until all nodes are marked

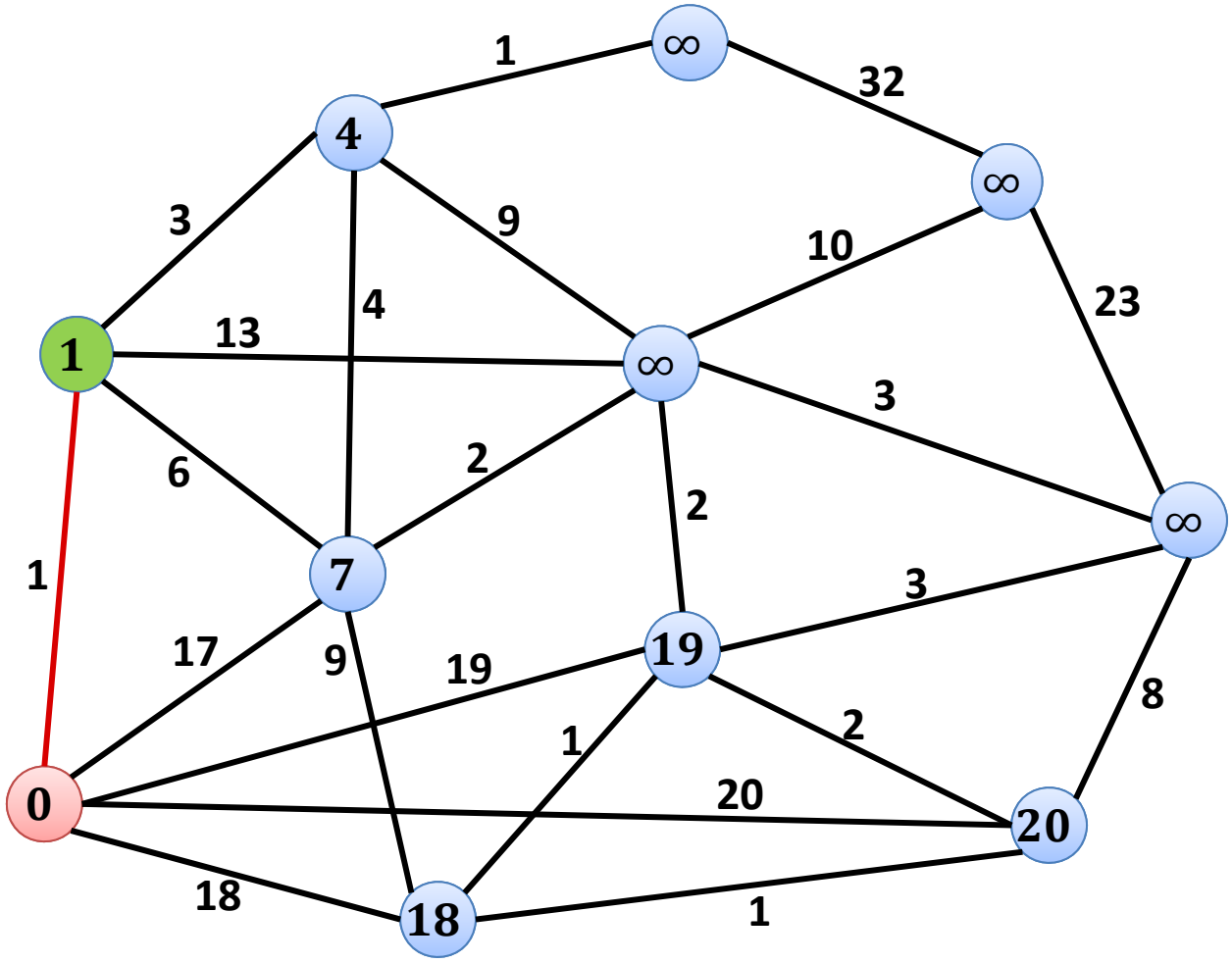
Example



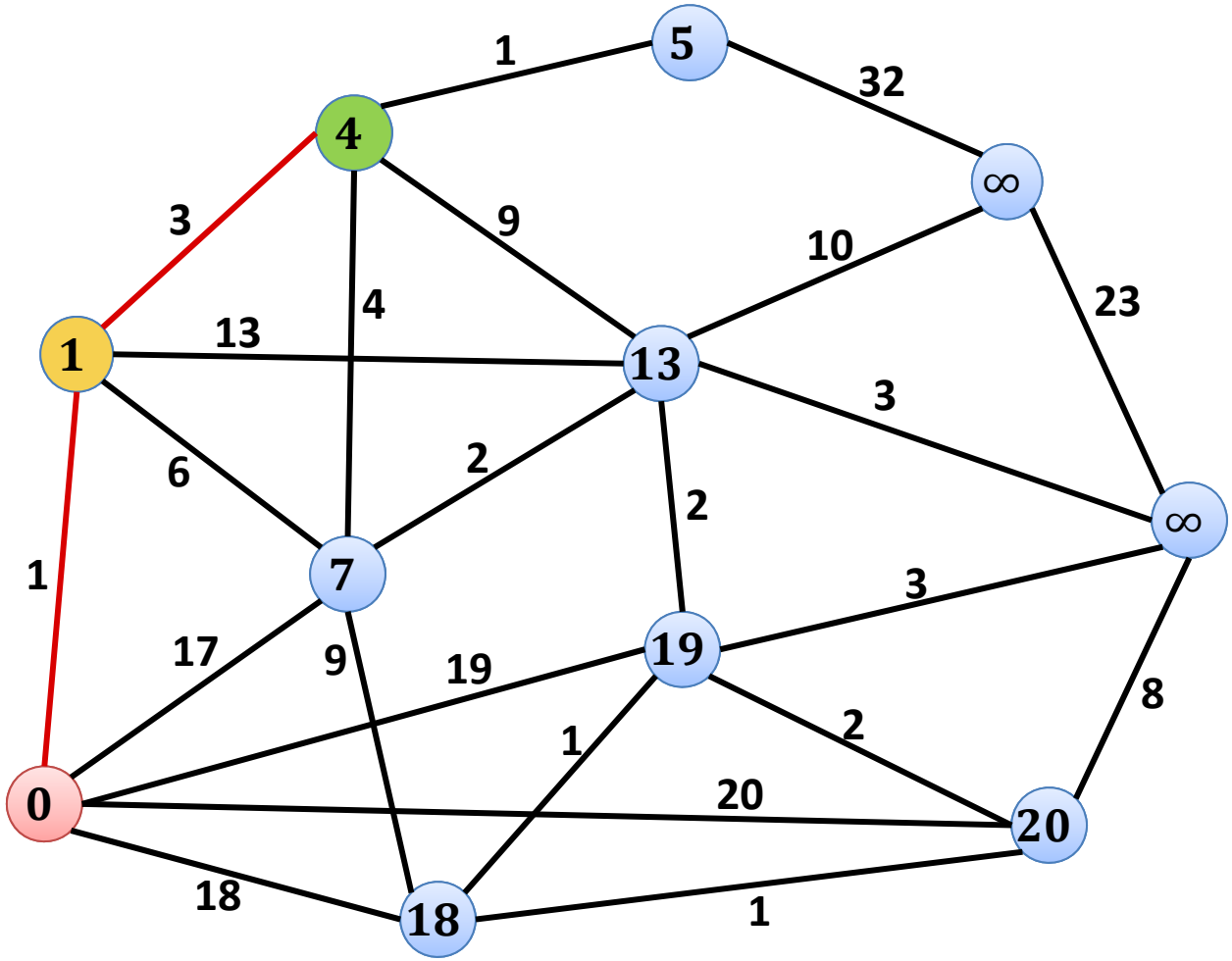
Example



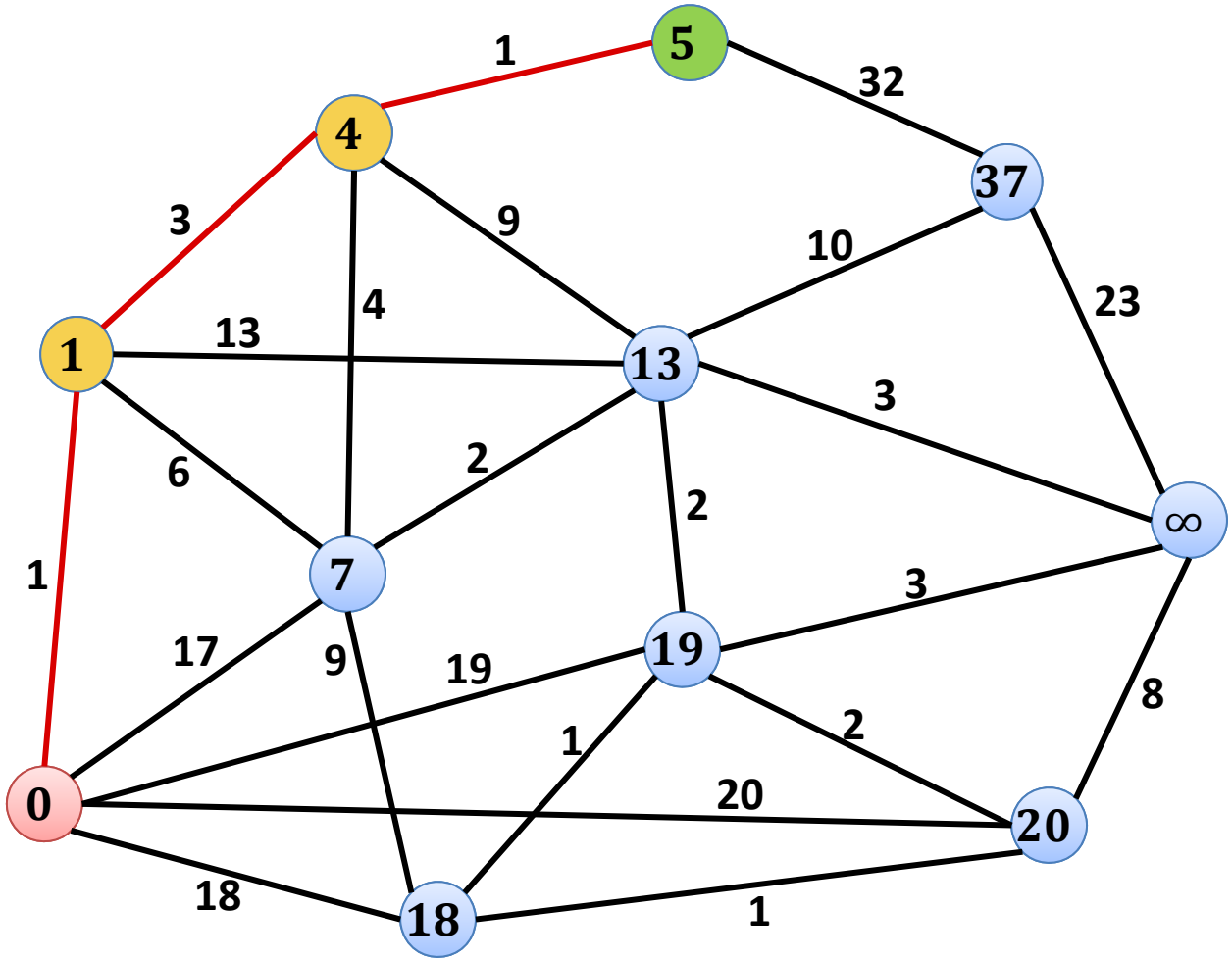
Example



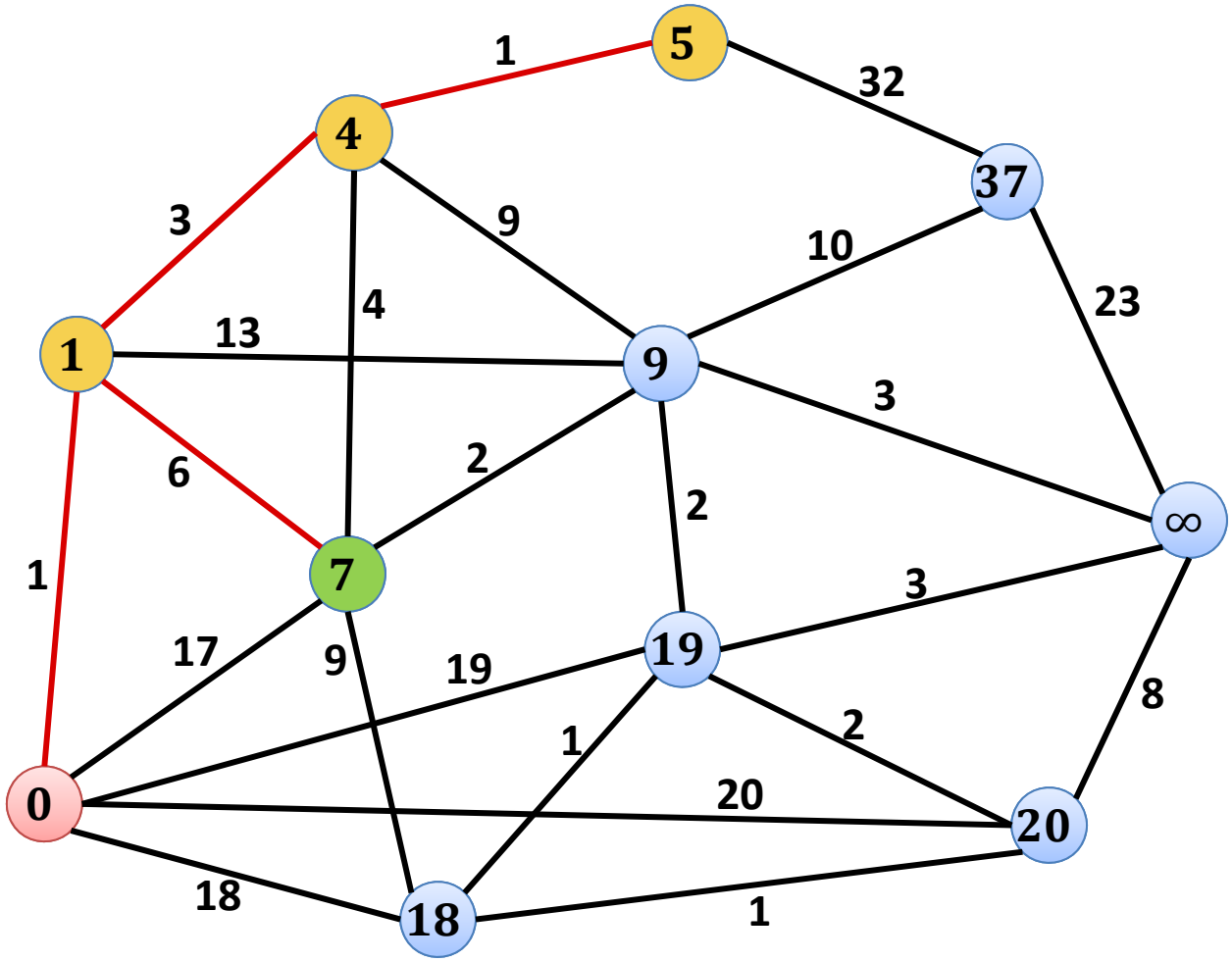
Example



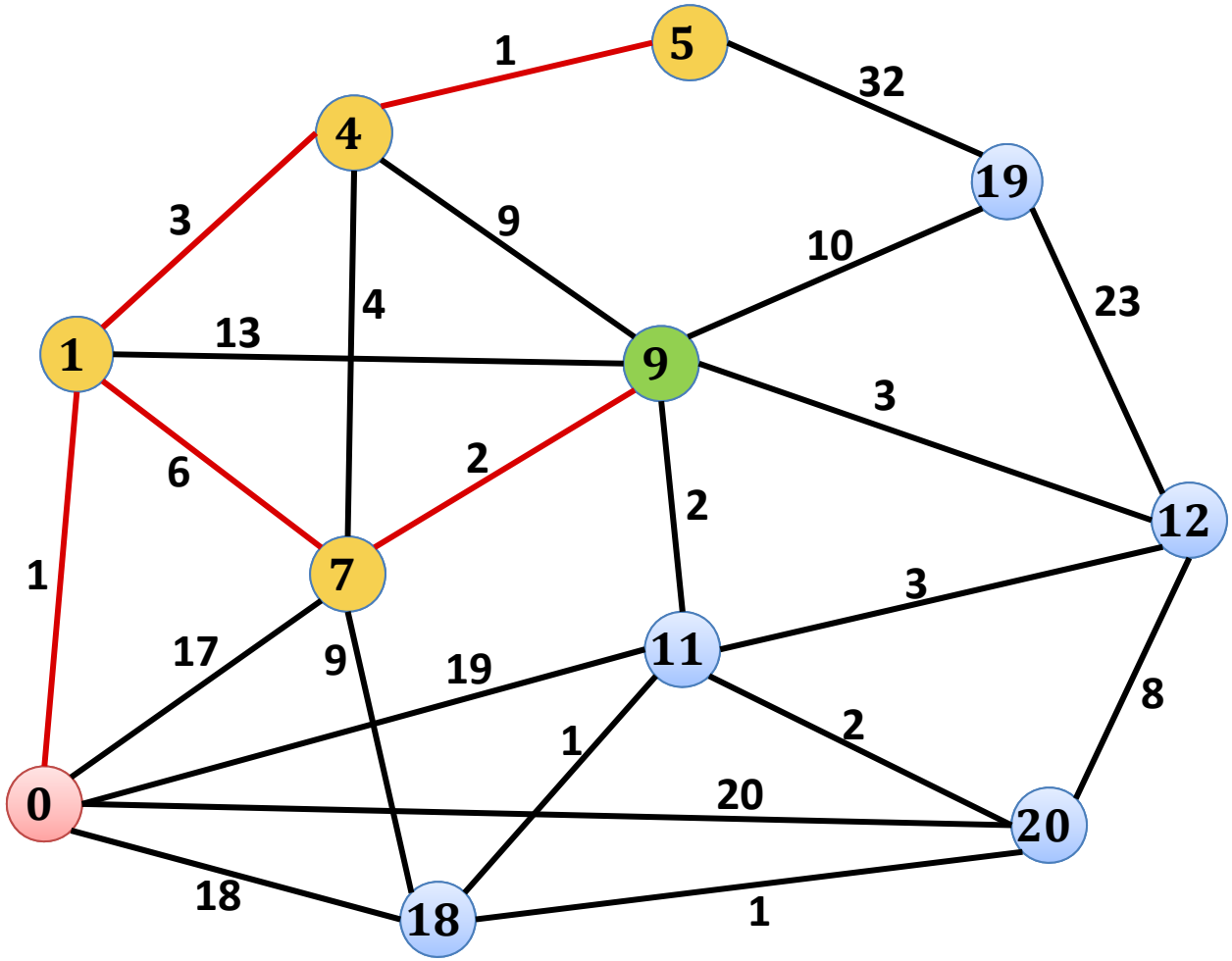
Example



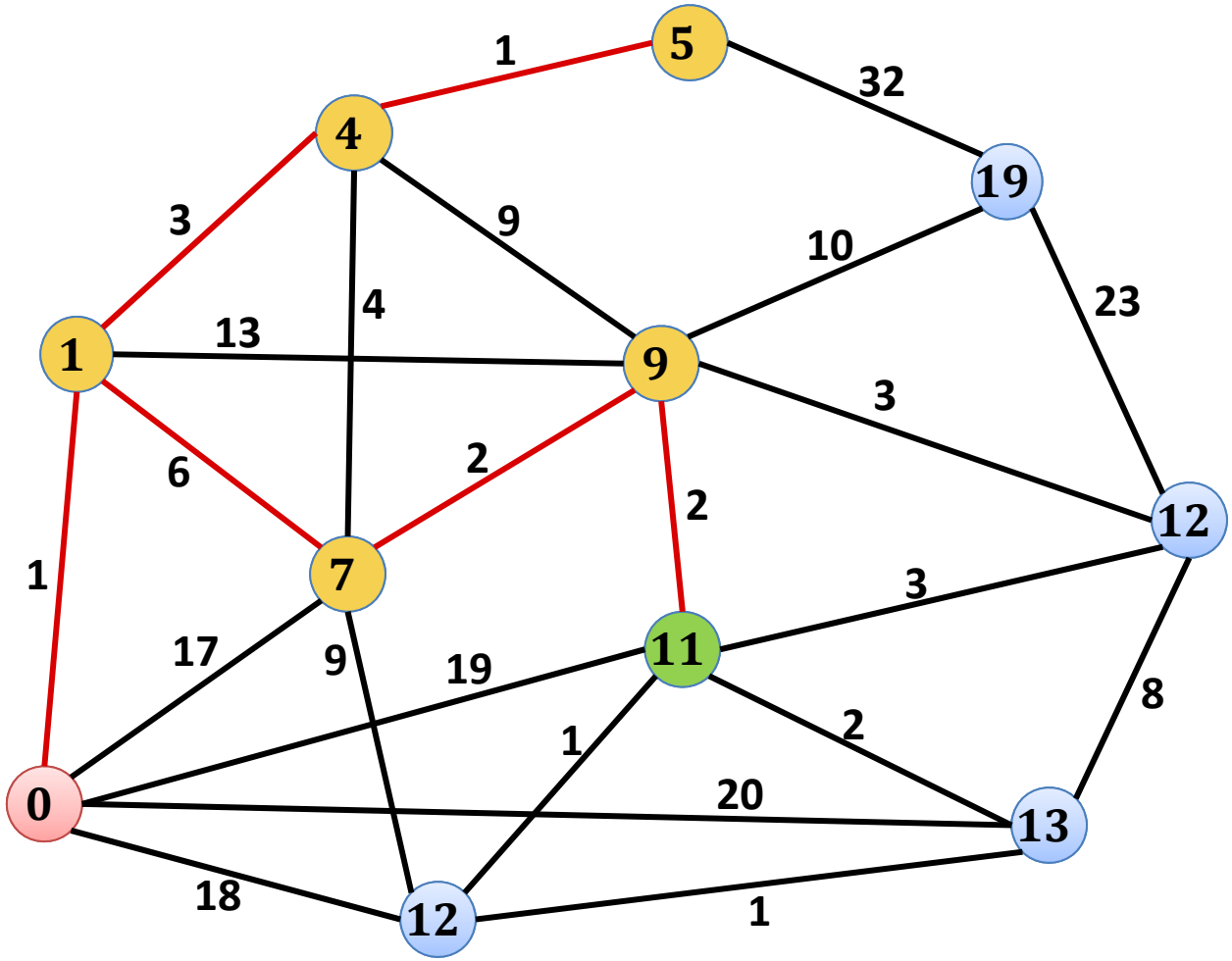
Example



Example



Example



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 are unmarked

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

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

5. mark node u

6. Until all nodes are marked

Priority Queue / Heap

- Stores (*key,data*) pairs (like dictionary)
- But, 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 minimum (*key,data*)-pair
- **Decrease-Key**(*entry,newkey*): decreases *key* of *entry* to *newkey*
- **Merge**: merges two heaps into one

Implementation of Dijkstra's Algorithm

Store nodes in a priority queue, use $d(s,v)$ as keys:

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

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

4. mark node u

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

6. Until all nodes are marked

Analysis

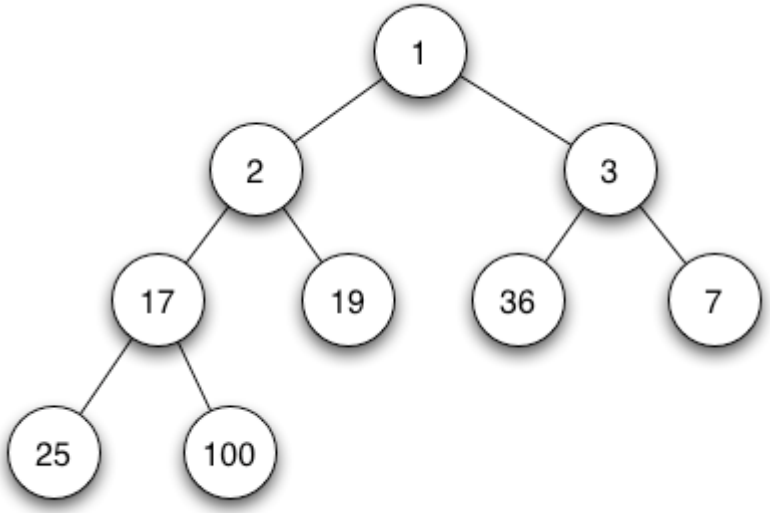
Number of priority queue operations for Dijkstra:

- **Initialize-Heap**: $\mathbf{1}$
- **Is-Empty**: $|V|$
- **Insert**: $|V|$
- **Get-Min**: $|V|$
- **Delete-Min**: $|V|$
- **Decrease-Key**: $|E|$
- **Merge**: $\mathbf{0}$

Priority Queue Implementation

Implementation as min-heap:

→ complete binary tree,
e.g., stored in an array



- **Initialize-Heap**: $\boldsymbol{O(1)}$
- **Is-Empty**: $\boldsymbol{O(1)}$
- **Insert**: $\boldsymbol{O(\log n)}$
- **Get-Min**: $\boldsymbol{O(1)}$
- **Delete-Min**: $\boldsymbol{O(\log n)}$
- **Decrease-Key**: $\boldsymbol{O(\log n)}$
- **Merge** (heaps of size m and n, $m \leq n$): $\boldsymbol{O(m \log n)}$

Better Implementation

- Can we do better?
- Cost of Dijkstra with complete binary min-heap implementation:

$$O(|E|\log|V|)$$

- Can be improved if we can make decrease-key cheaper…
- Cost of merging two heaps is expensive
- We will get there in two steps:

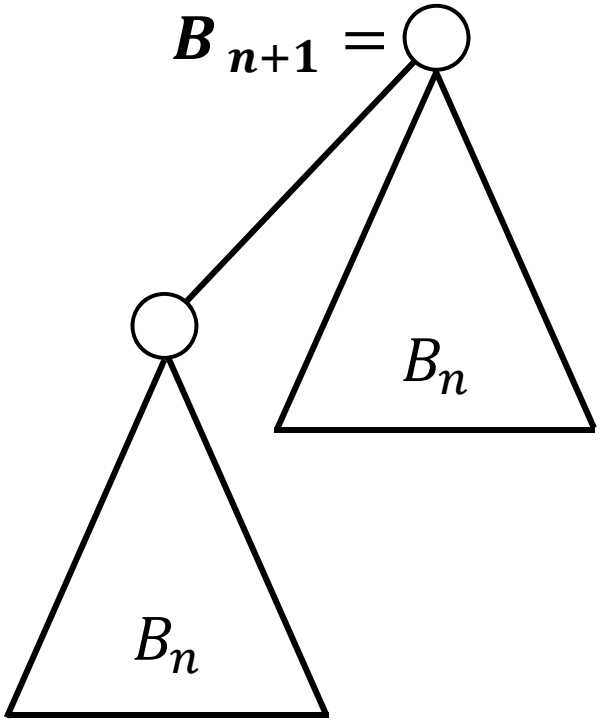
  Binomial heap → Fibonacci heap

Definition: Binomial Tree

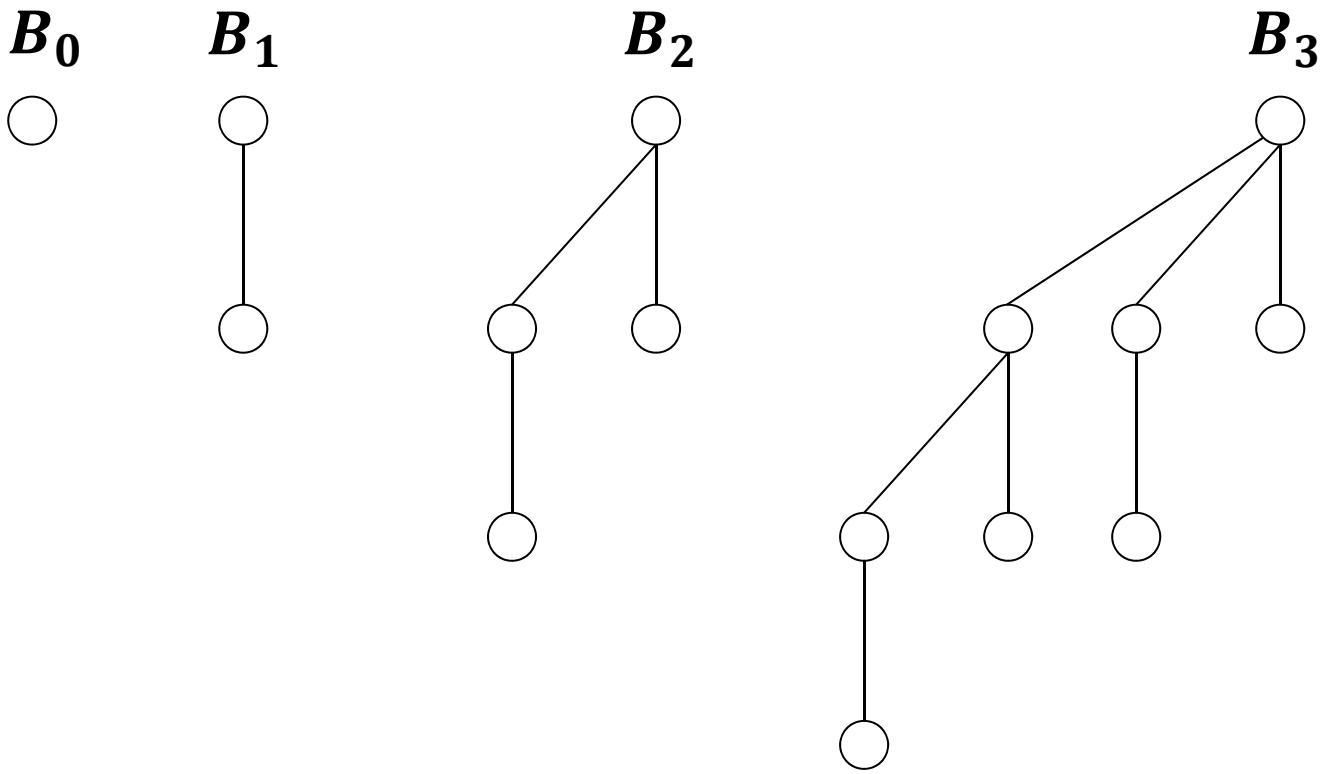
Binomial tree B_n of order n $(n \geq 0)$:

$\boldsymbol{B_0} =$ ○

$\boldsymbol{B_{n+1}} =$



Binomial Trees



Properties

1. Tree B_n has 2^n nodes

2. Height of tree B_n is n

3. Root degree of B_n is n

4. In B_n, there are exactly $\binom{n}{i}$ nodes at depth i

Binomial Coefficients

- Binomial coefficient:

$$\binom{n}{k} : \#\text{ of } k-\text{element}-\text{subsets of a set of size } n$$

- Property: $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$

Pascal triangle:

$$\begin{array}{ccccccccccc}
 & & & & & 1 & & & & & \\
 & & & & 1 & & 1 & & & & \\
 & & & 1 & & 2 & & 1 & & & \\
 & & 1 & & 3 & & 3 & & 1 & & \\
 & 1 & & 4 & & 6 & & 4 & & 1 & \\
1 & & 5 & & 10 & & 10 & & 5 & & 1
\end{array}$$

Number of Nodes at Depth i in B_n

Claim: In B_n, there are exactly $\binom{n}{i}$ nodes at depth i

Binomial Heap

- Keys are stored in nodes of binomial trees of different order

  n nodes: there is a binomial tree B_i or order i iff
  bit i of base-2 representation of n is 1.

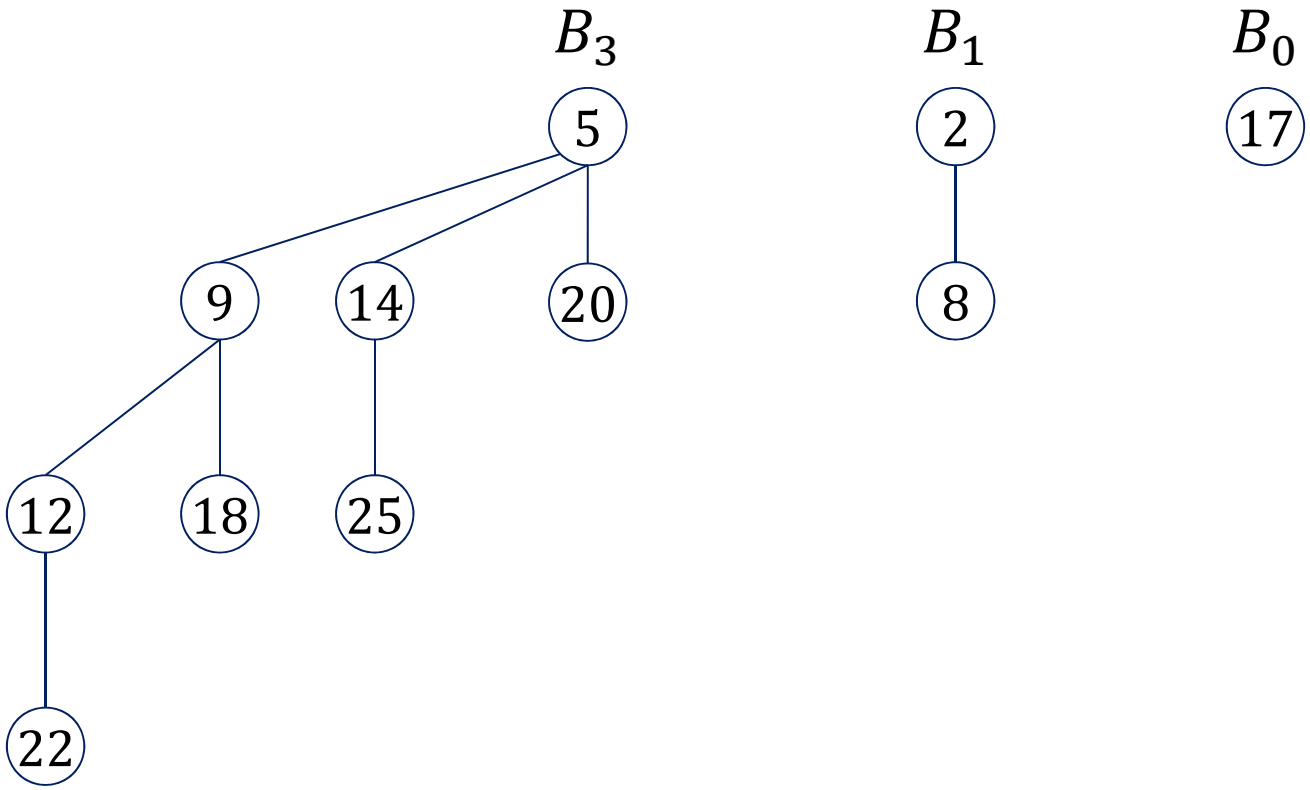
- **Min-Heap Property:**

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

Example

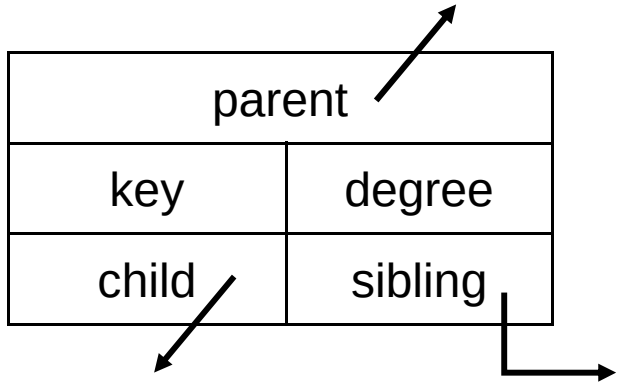
- 10 keys: $\{2, 5, 8, 9, 12, 14, 17, 18, 20, 22, 25\}$

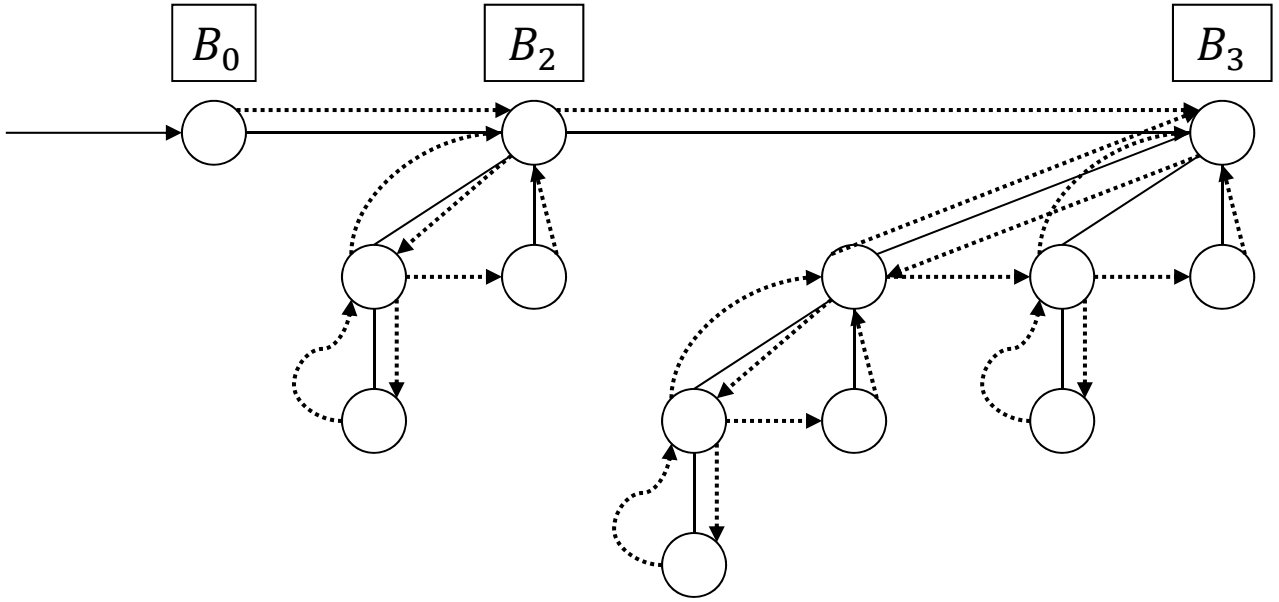
- Binary representation of n: $(11)_2 = 1011$
  → trees B_0, B_1, and B_3 present



Child-Sibling Representation

Structure of a node:



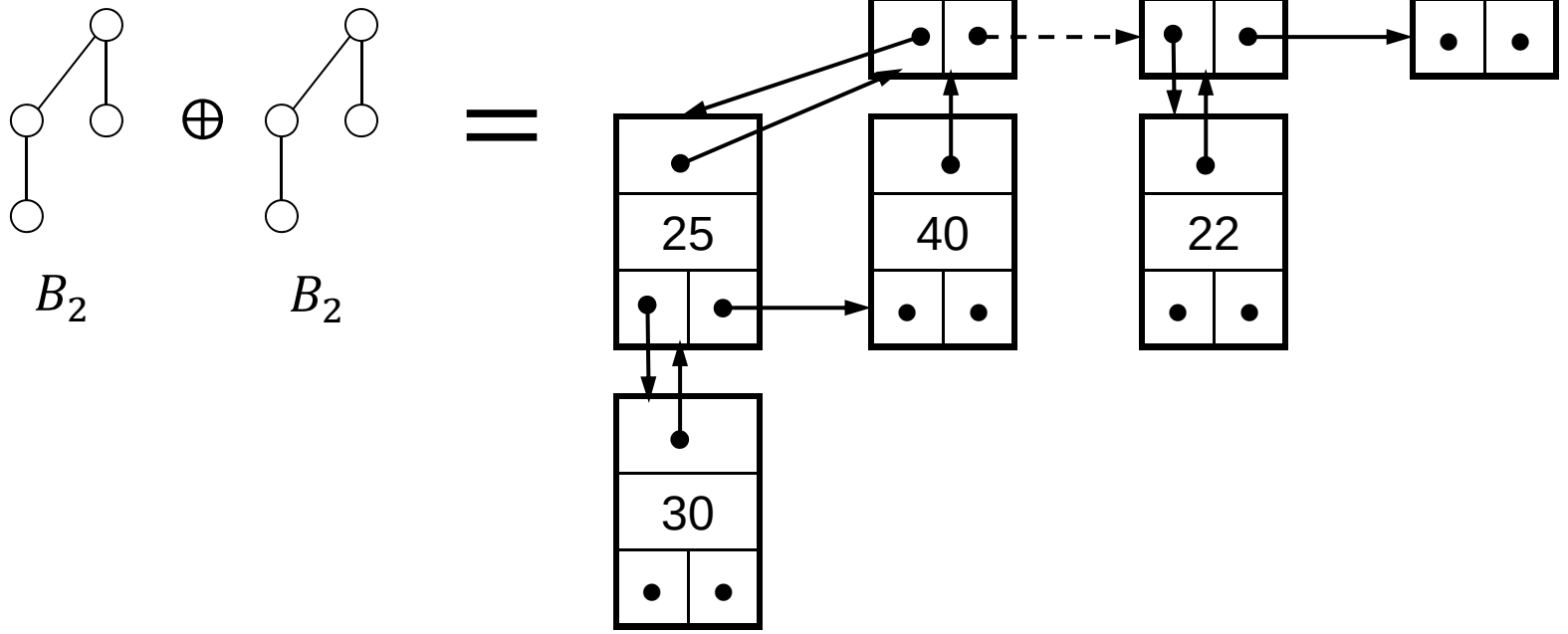


Link Operation

- Unite two binomial trees of the same order to one tree:

$$B_n \oplus B_n \Rightarrow B_{n+1}$$

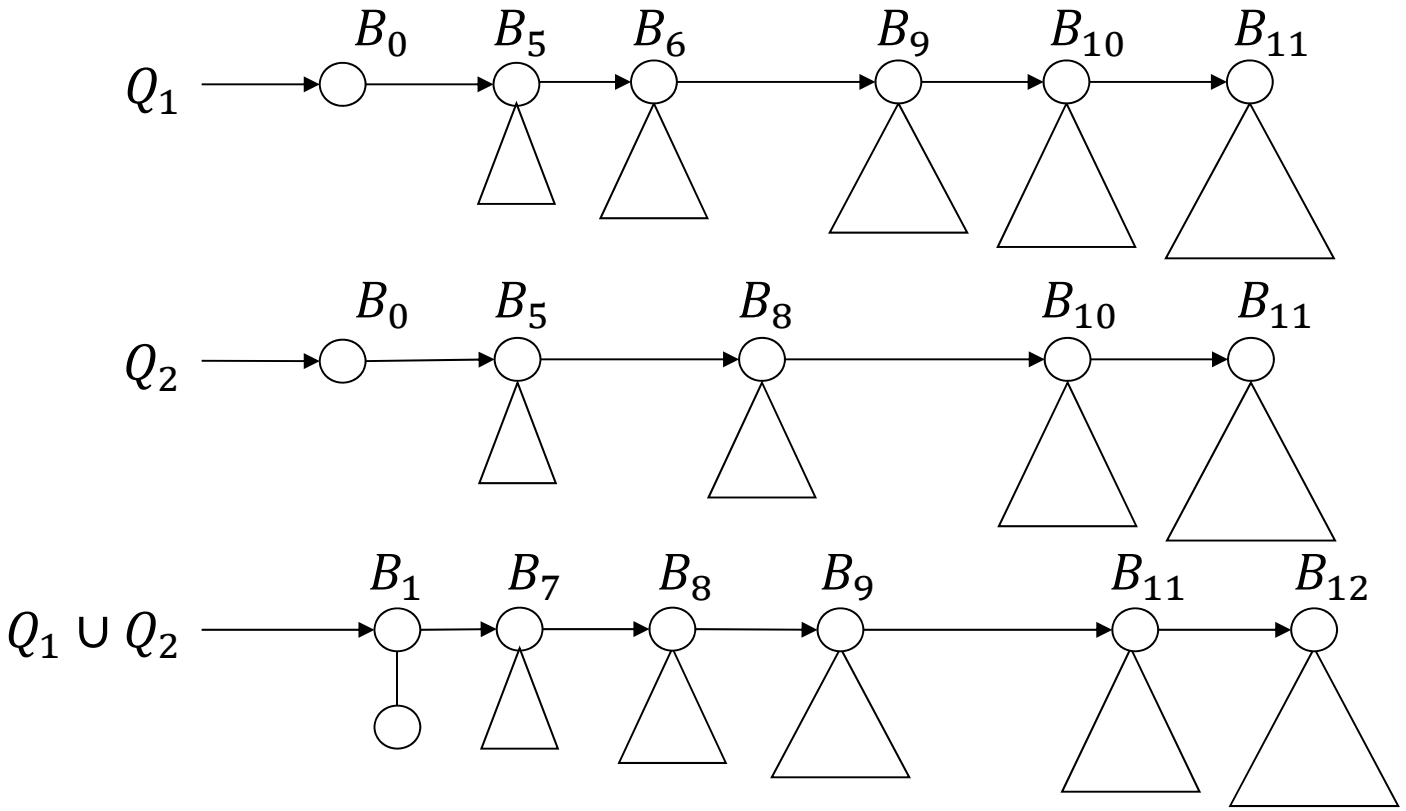
- Time: $O(1)$



Merge Operation

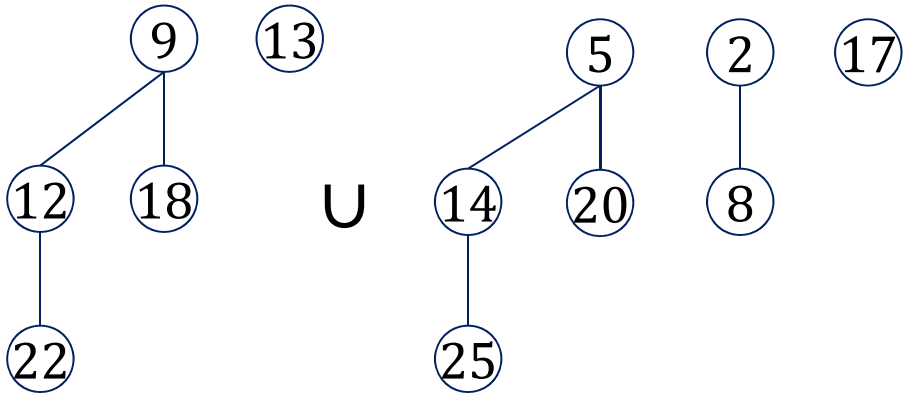
Merging two binomial heaps:

- **For $i = 0, 1, \dots, \log n$:**
  If there are 2 or 3 binomial trees B_i: apply link operation to merge 2 trees into one binomial tree B_{i+1}



Time:
$O(\log n)$

Example



Operations

Initialize: create empty list of trees

Get minimum of queue: time $O(1)$ (if we maintain a pointer)

Decrease-key at node v:

- Set *key* of node v to new key
- Swap with parent until min-heap property is restored
- Time: $O(\log n)$

Insert *key* x into queue Q:

1. Create queue Q' of size 1 containing only x
2. Merge Q and Q'

- Time for insert: $O(\log n)$

Operations

Delete-Min Operation:

1. Find tree B_i with minimum root r

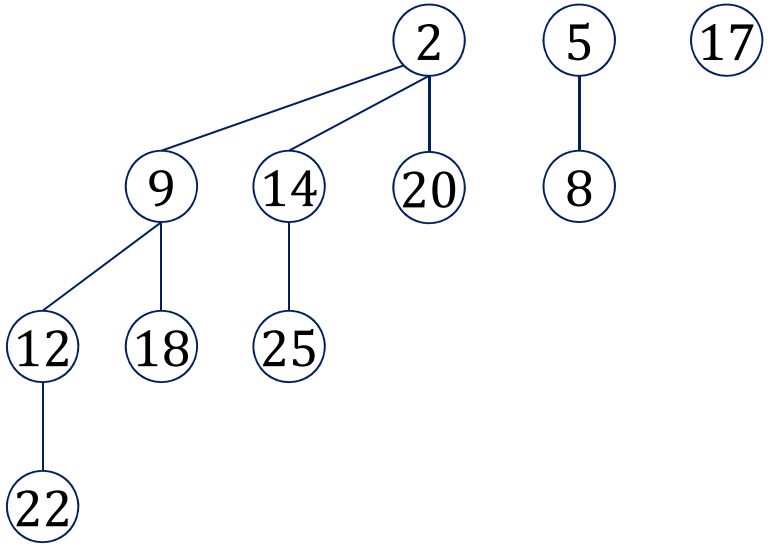
2. Remove B_i from queue Q → queue Q'

3. Children of r form new queue Q''

4. Merge queues Q' and Q''

- **Overall time: $O(\log n)$**

Delete-Min Example



Complexities Binomial Heap

- **Initialize-Heap**: $\boldsymbol{O(1)}$
- **Is-Empty**: $\boldsymbol{O(1)}$
- **Insert**: $\boldsymbol{O(\log n)}$
- **Get-Min**: $\boldsymbol{O(1)}$
- **Delete-Min**: $\boldsymbol{O(\log n)}$
- **Decrease-Key**: $\boldsymbol{O(\log n)}$
- **Merge** (heaps of size m and n, $m \leq n$): $\boldsymbol{O(\log n)}$

Can We Do Better?

- Binomial 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?

- Structure of binomial heap is not flexible:
  - Simplifies analysis, allows to get strong worst-case bounds
  - **But**, operations almost inherently need at least logarithmic time

Fibonacci Heaps

Lacy-merge variant of binomial heaps:

- Do not merge trees as long as possible…

Structure:

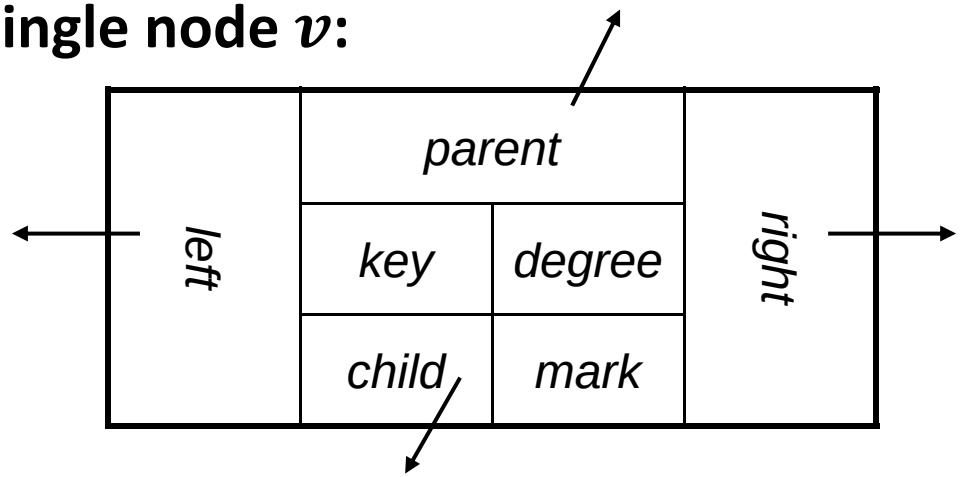
A Fibonacci heap H consists of a collection of trees satisfying the min-heap property.

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

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

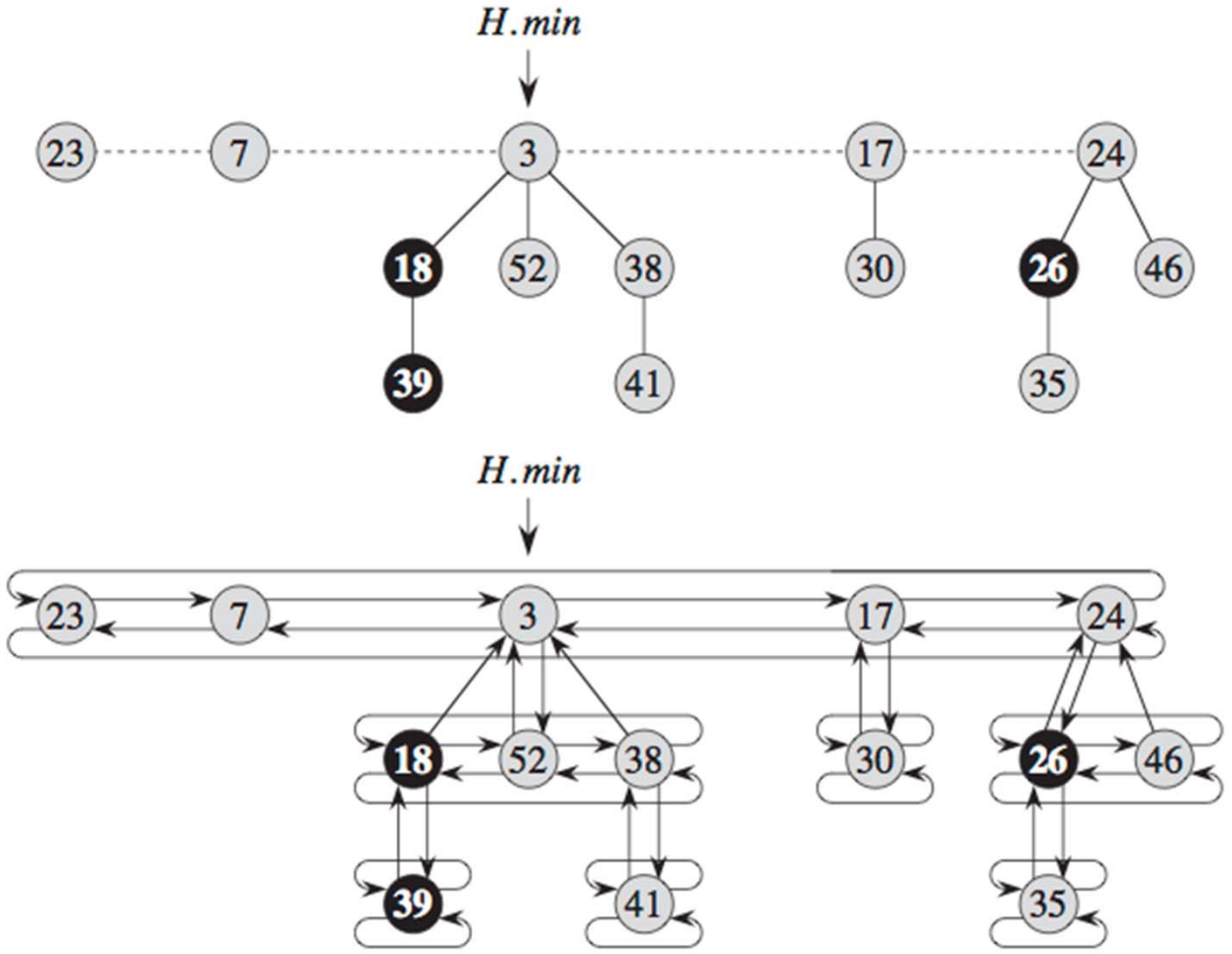


Figure: Cormen et al., Introduction to Algorithms

Simple (Lazy) Operations

Initialize-Heap H:

- $H.rootlist := H.min := null$

Merge heaps H and H':

- concatenate root lists
- update $H.min$

Insert element e into H:

- create new one-node tree containing e → H'
- 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. $m := H.min$;
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:

- $rank(v)$: degree of v

Tree T:

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

Heap H:

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

Assumption (n: number of nodes in H):

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

  - for a known function $D(n)$

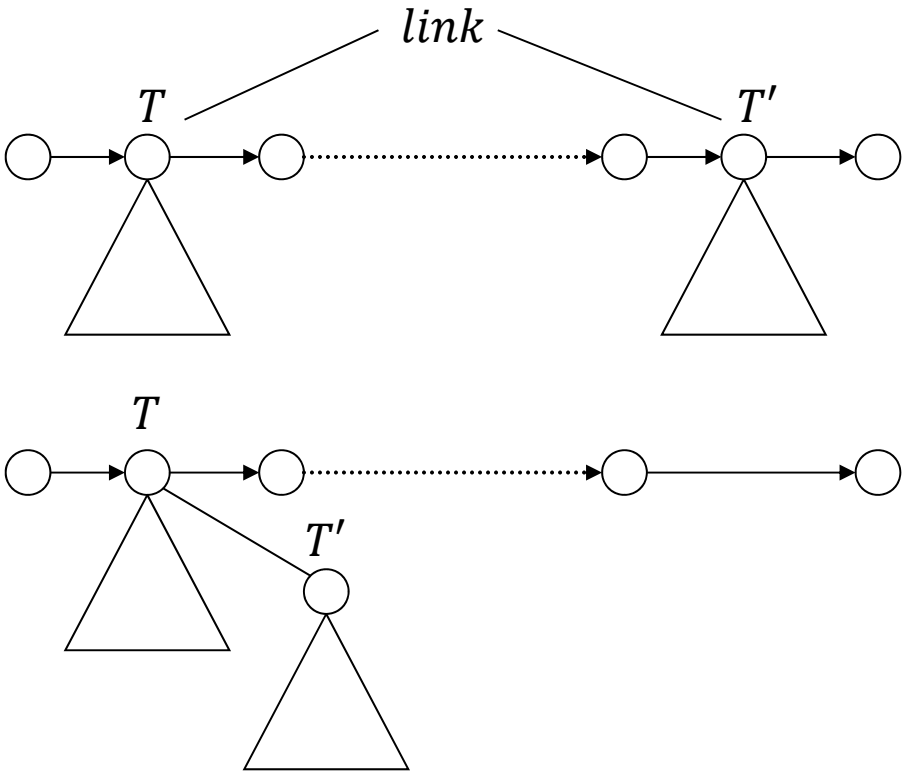
Merging Two Trees

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

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

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

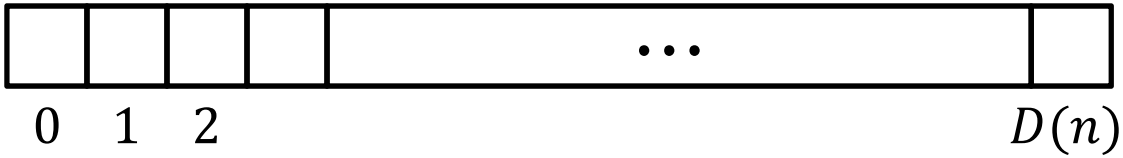
- Removes tree T' from root list and adds T' to child list of T



- $rank(T) := rank(T) + 1$
- $T'.mark :=$ **false**

Consolidation of Root List

Array A pointing to find roots with the same rank:

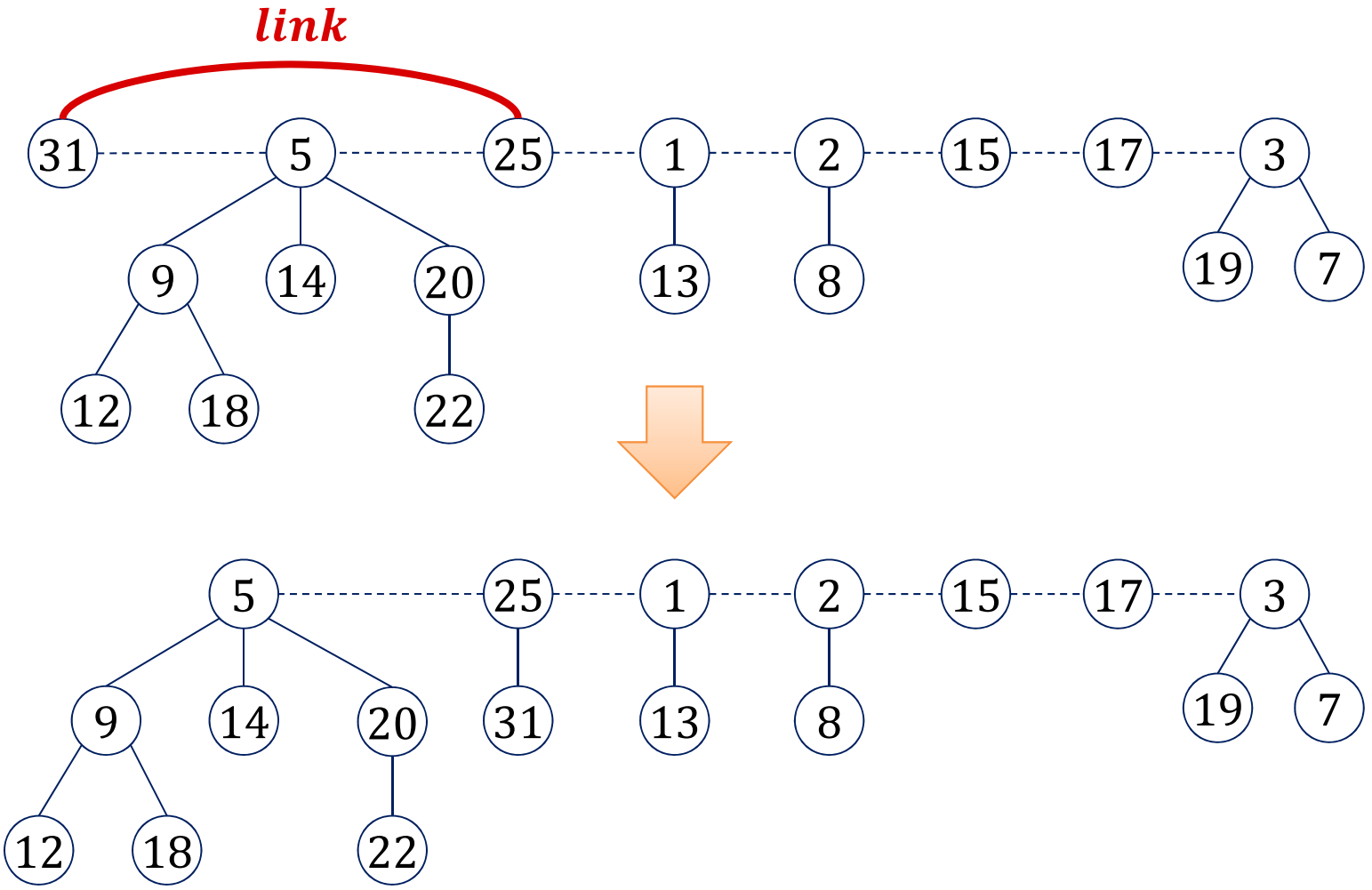


Consolidate:

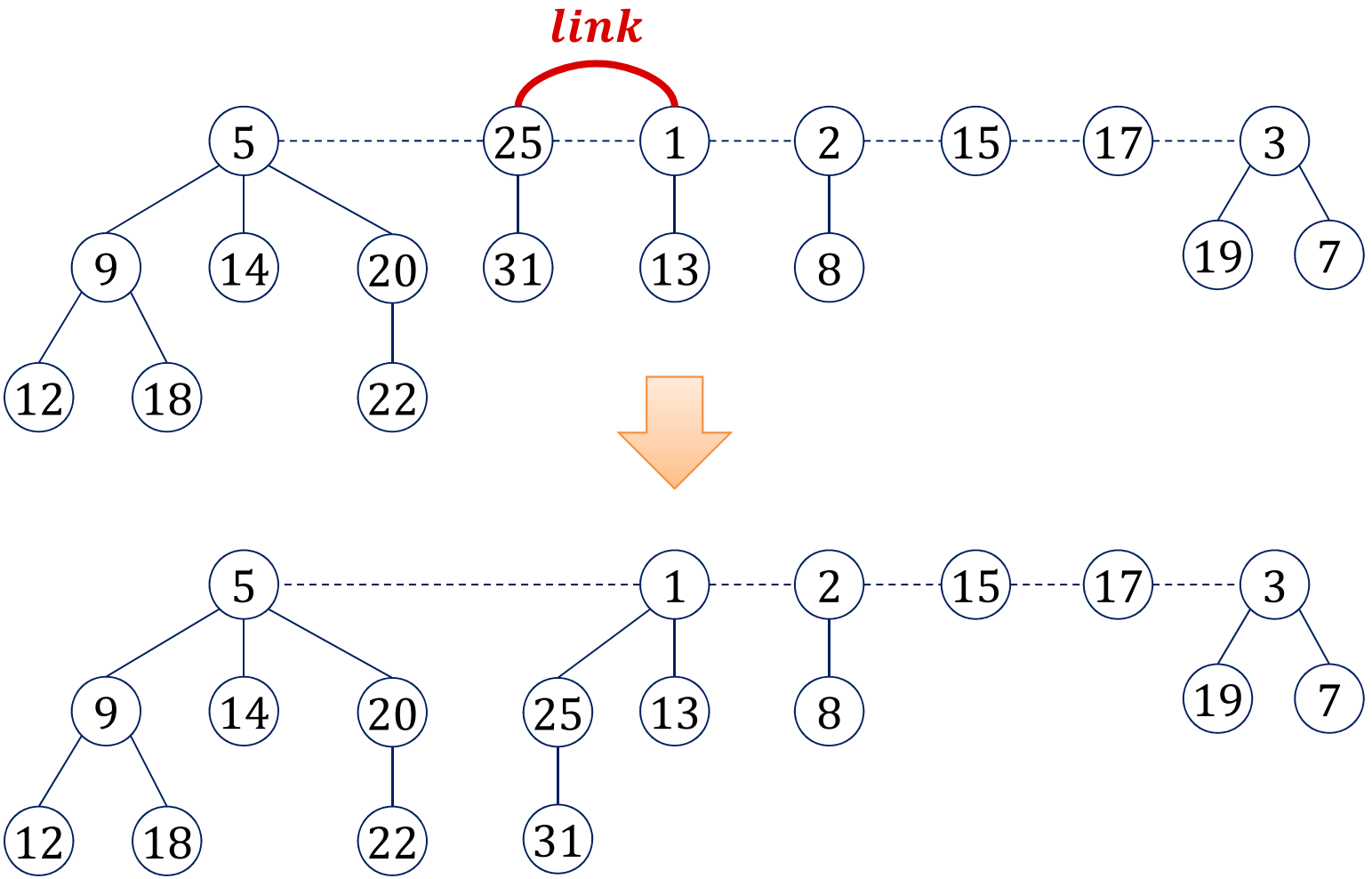
1. **for** $i := 0$ **to** $D(n)$ **do** $A[i] := \text{null}$;
2. **while** $H.rootlist \neq \text{null}$ **do**
3. $\quad T :=$ "delete and return first element of $H.rootlist$"
4. $\quad$ **while** $A[rank(T)] \neq \text{null}$ **do**
5. $\quad\quad T' := A[rank(T)]$;
6. $\quad\quad A[rank(T)] := null$;
7. $\quad\quad T := link(T, T')$
8. $\quad A[rank(T)] := T$
9. Create new $H.rootlist$ and $H.min$

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

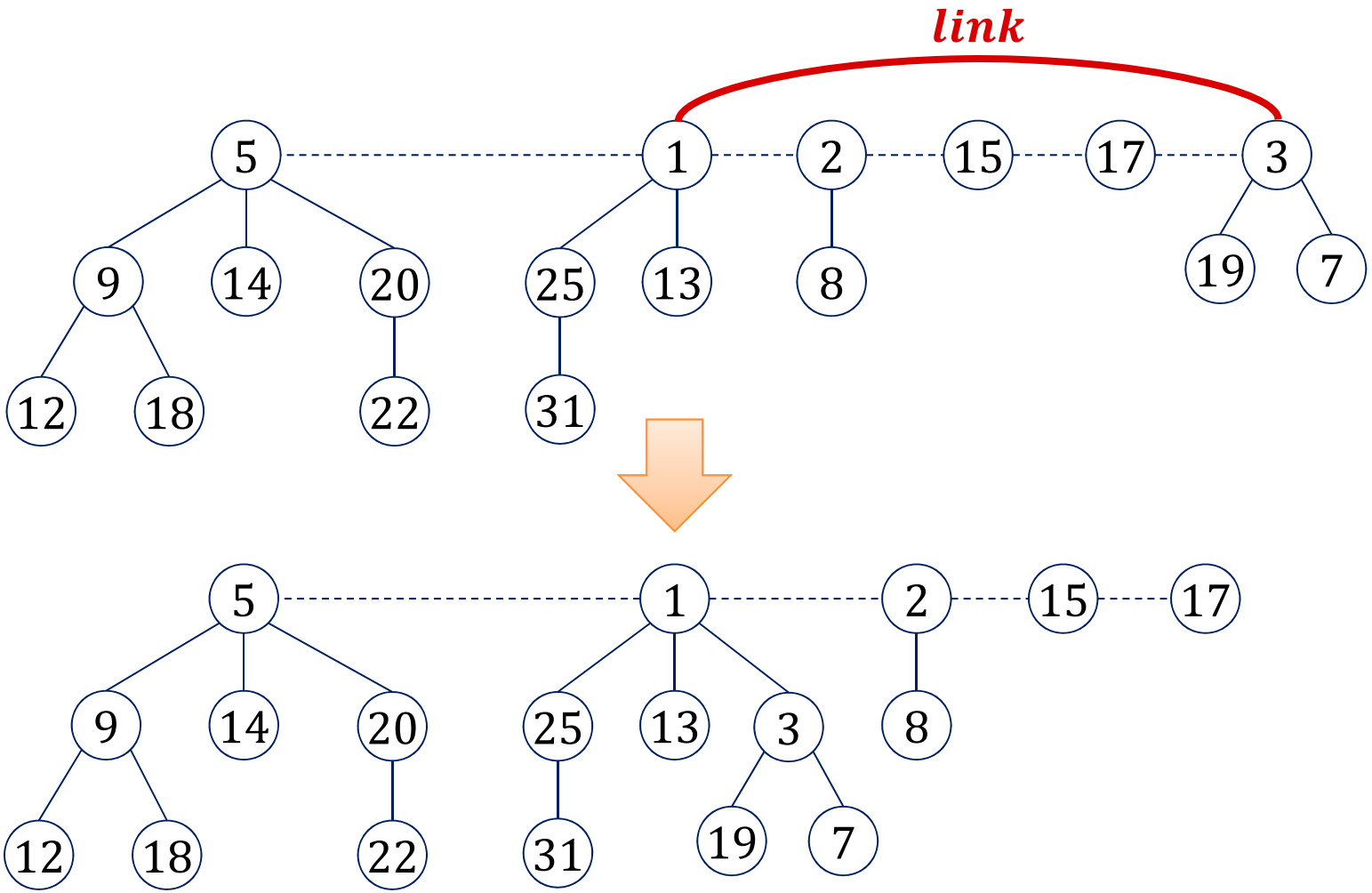
Consolidate Example



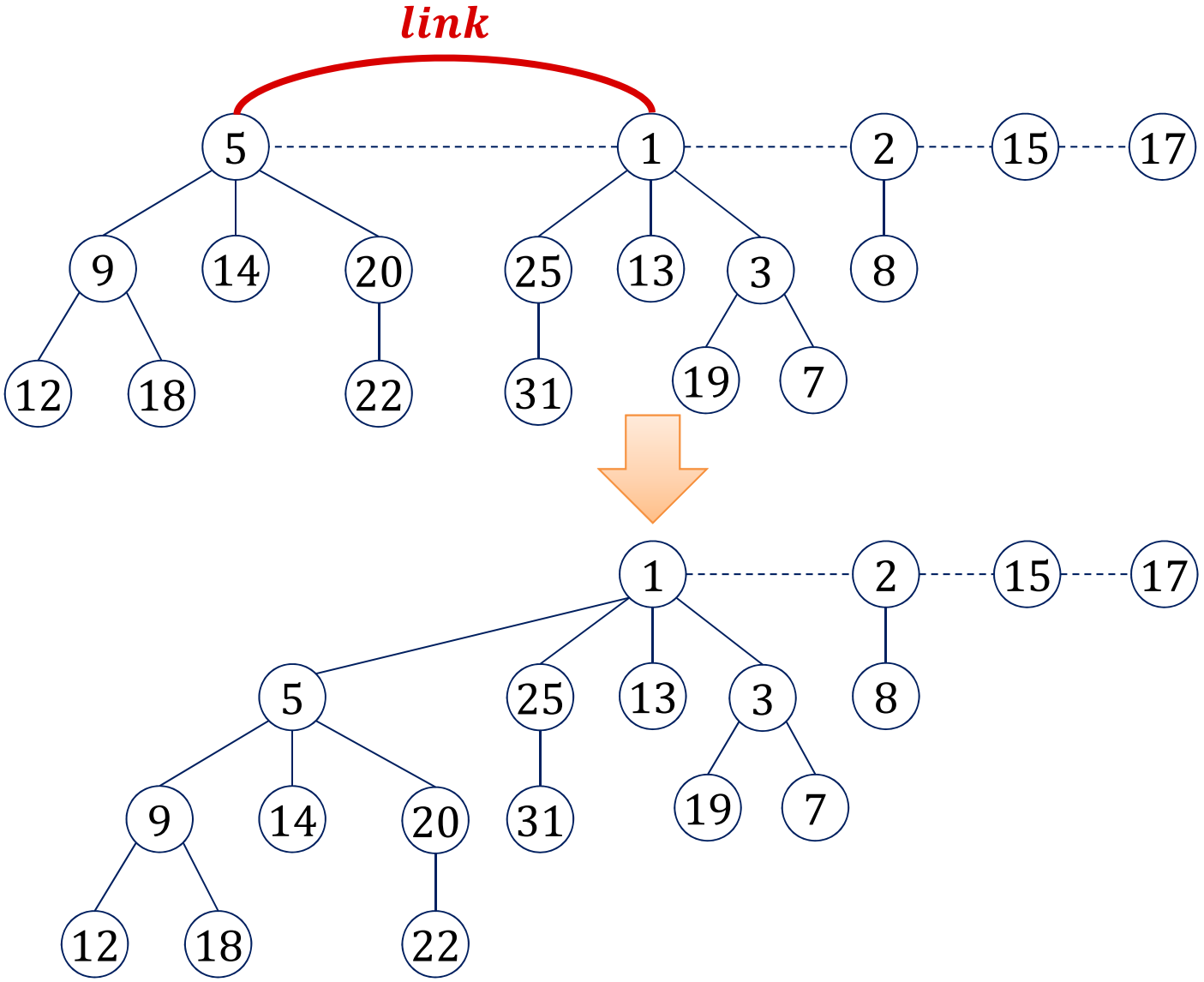
Consolidate Example



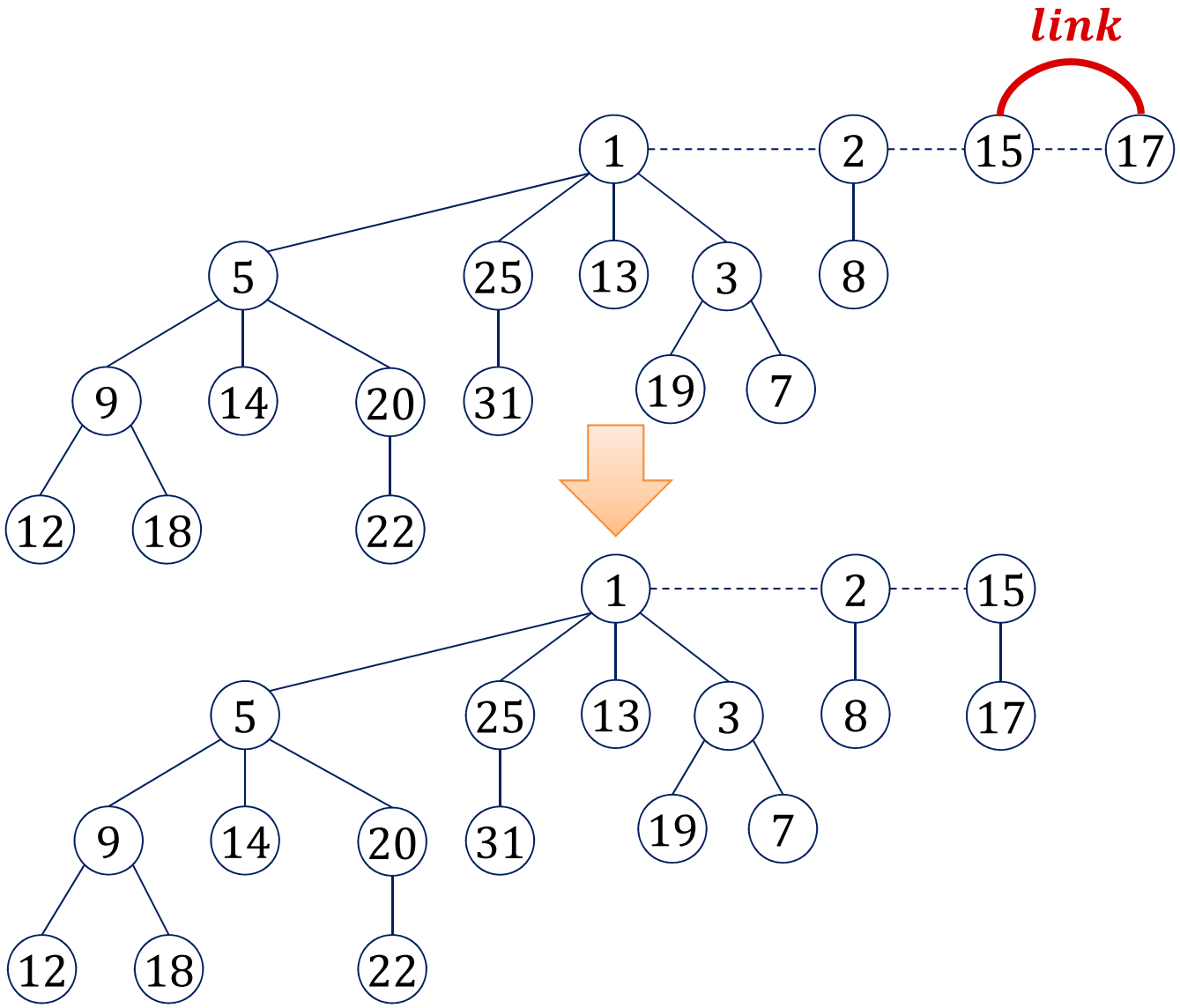
Consolidate Example



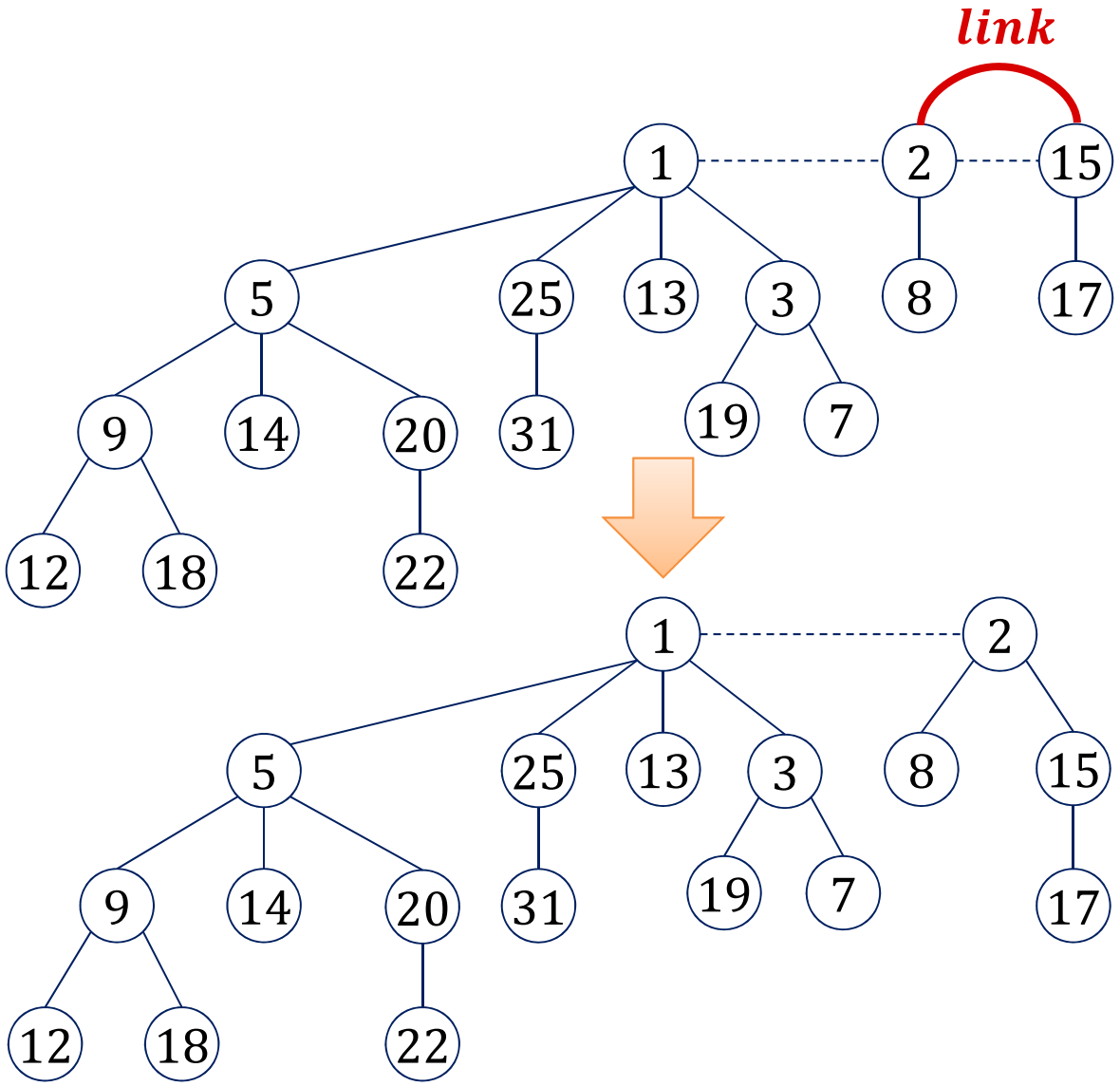
Consolidate Example



Consolidate Example



Consolidate Example



Operation Decrease-Key

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

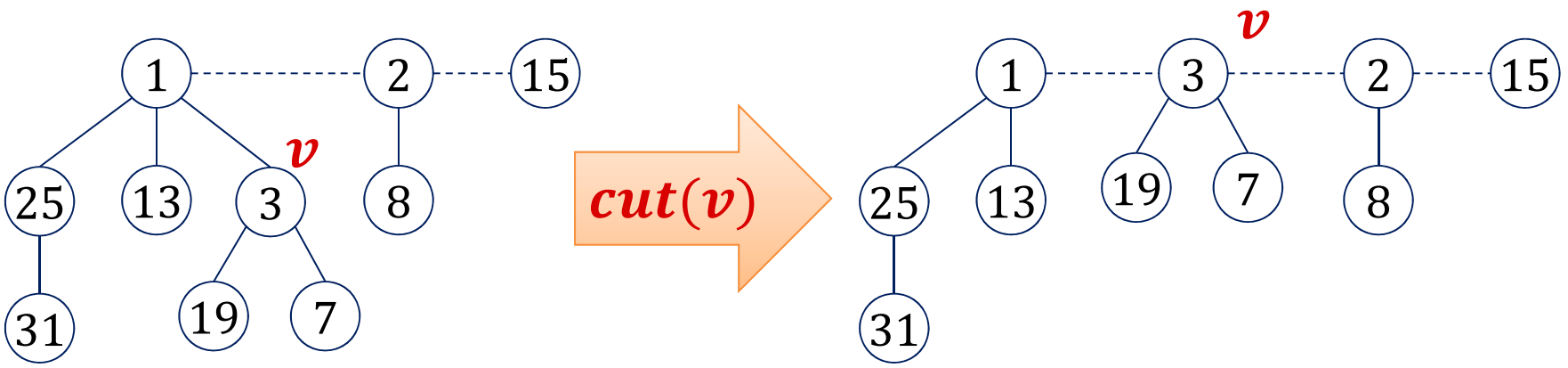
1. **if** $x \geq v.key$ **then return**;
2. $v.key := x$; update $H.min$;
3. **if** $v \in H.rootlist \vee x \geq v.parent.key$ **then return**
4. **repeat**
5. $\quad parent := v.parent$;
6. $\quad \boldsymbol{H.cut(v)}$;
7. $\quad v := parent$;
8. **until** $\neg(\boldsymbol{v.mark}) \vee v \in H.rootlist$;
9. **if** $v \notin H.rootlist$ **then** $\boldsymbol{v.mark := \text{true}}$;

Operation Cut(v)

Operation $H.cut(v)$:

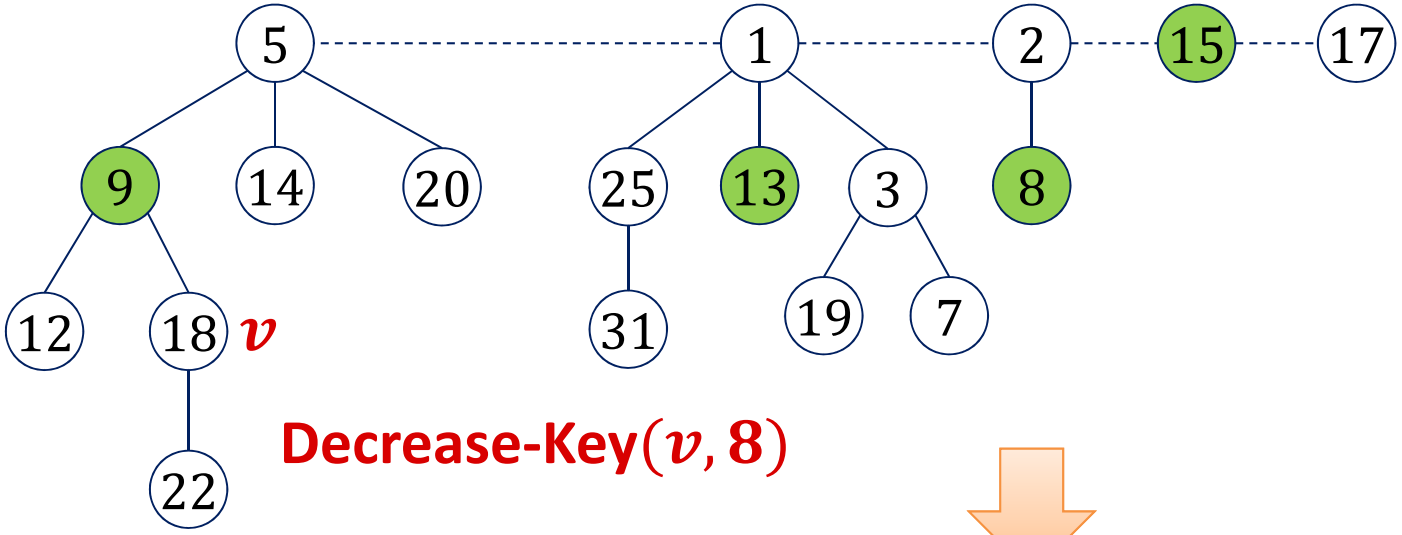
- Cuts v's sub-tree from its parent and adds v to rootlist

1. **if** $v \notin H.rootlist$ **then**
2. // cut the link between v and its parent
3. $rank(v.parent) := rank(v.parent) - 1;$
4. remove v from $v.parent.child$ (list)
5. $v.parent := \text{null};$
6. add v to $H.rootlist$

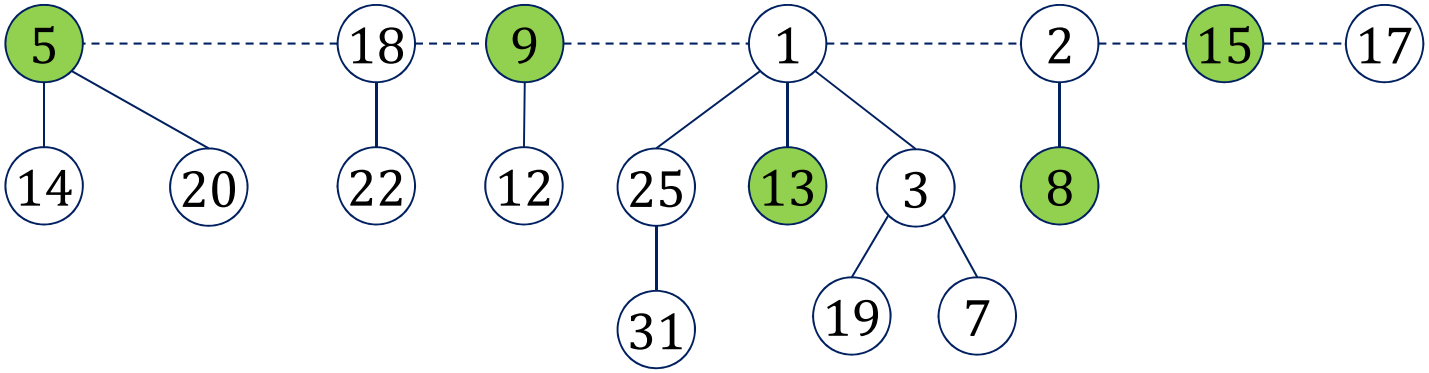


Decrease-Key Example

- Green nodes are marked



Decrease-Key$(v, 8)$



Fibonacci Heap Marks

History of a node v:

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

a child of v is cut ⇒ $\boldsymbol{v.mark := \text{true}}$

a second child of v is cut ⇒ $\boldsymbol{H.cut(v)}$

- 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.

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

Exercises: 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 and merge 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?

- We can → requires **amortized analysis**

Amortization

- Consider sequence $o_1, o_2, \dots, o_n$ of n operations (typically performed on some data structure D)

- $\boldsymbol{t_i}$: execution time of operation o_i
- $\boldsymbol{T := t_1 + t_2 + \cdots + t_n}$: total execution time

- The execution time of a single operation might

  vary within a large range (e.g., $t_i \in [1, O(i)]$)

- The worst case overall execution time might still be small

  → average execution time per operation might be small in the worst case, even if single operations can be expensive

Analysis of Algorithms

- Best case
- Worst case
- Average case
- Amortized worst case

What it the average cost of an operation in a worst case sequence of operations?

Example: Binary Counter

Incrementing a binary counter: determine the bit flip cost:

Operation	Counter Value	Cost
	00000	
1	0000**1**	1
2	000**10**	2
3	0001**1**	1
4	00**100**	3
5	0010**1**	1
6	0011**0**	2
7	0011**1**	1
8	0**1000**	4
9	0100**1**	1
10	0101**0**	2
11	0101**1**	1
12	01**100**	3
13	0110**1**	1

Accounting Method

Observation:

- Each increment flips exactly one 0 into a 1

$$00100\mathbf{0}1111 \Longrightarrow 00100\mathbf{1}0000$$

Idea:

- Have a bank account (with initial amount 0)
- Paying x to the bank account costs x
- Take "money" from account to pay for expensive operations

Applied to binary counter:

- Flip from 0 to 1: pay 1 to bank account (cost: 2)
- Flip from 1 to 0: take 1 from bank account (cost: 0)
- Amount on bank account = number of ones

  → We always have enough "money" to pay!

Accounting Method

Op.	Counter	Cost	To Bank	From Bank	Net Cost	Credit
	0 0 0 0 0					
1	0 0 0 0 **1**	1				
2	0 0 0 **1 0**	2				
3	0 0 0 1 **1**	1				
4	0 0 **1 0 0**	3				
5	0 0 1 0 **1**	1				
6	0 0 1 **1 0**	2				
7	0 0 1 1 **1**	1				
8	0 **1 0 0 0**	4				
9	0 1 0 0 **1**	1				
10	0 1 0 **1 0**	2				

Potential Function Method

- Most generic and elegant way to do amortized analysis!
  - But, also more abstract than the others…
- State of data structure / system: $S \in \mathcal{S}$ (state space)

  Potential function $\Phi: \mathcal{S} \to \mathbb{R}_{\geq 0}$
- **Operation i:**
  - t_i: actual cost of operation i
  - S_i: state after execution of operation i (S_0: initial state)
  - $\Phi_i := \Phi(S_i)$: potential after exec. of operation i
  - a_i: amortized cost of operation i:

$$a_i := t_i + \Phi_i - \Phi_{i-1}$$

Potential Function Method

Operation i:

actual cost: t_i amortized cost: $a_i = t_i + \Phi_i - \Phi_{i-1}$

Overall cost:

$$T := \sum_{i=1}^{n} t_i = \left(\sum_{i}^{n} a_i\right) + \Phi_0 - \Phi_n$$

Binary Counter: Potential Method

- **Potential function:**

  Φ: number of ones in current counter

- Clearly, $\Phi_0 = 0$ and $\Phi_i \geq 0$ for all $i \geq 0$

- Actual cost t_i:
  - 1 flip from 0 to 1
  - $t_i - 1$ flips from 1 to 0

- Potential difference: $\Phi_i - \Phi_{i-1} = 1 - (t_i - 1) = 2 - t_i$

- Amortized cost: $a_i = t_i + \Phi_i - \Phi_{i-1} = 2$

Back to Fibonacci Heaps

- Worst-case cost of a single delete-min or decrease-key operation is $\Omega(n)$
- Can we prove a small worst-case amortized cost for delete-min and decrease-key operations?

Remark:

- Data structure that allows operations $O_1, \dots, O_k$
- We say that operation O_p has amortized cost a_p if for every execution the total time is

$$T \leq \sum_{p=1}^{k} n_p \cdot a_p \,,$$

where n_p is the number of operations of type O_p

Amortized Cost of Fibonacci Heaps

- **Initialize-heap**, **is-empty**, **get-min**, **insert**, and **merge** have **worst-case 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)

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

- We will now need the marks…
- Cost for Dijkstra: $O(|E| + |V| \log |V|)$

Fibonacci Heaps: Marks

Cycle of a node:

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

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

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

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

3. Second child of v is cut

node v is cut as well and moved to root list

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

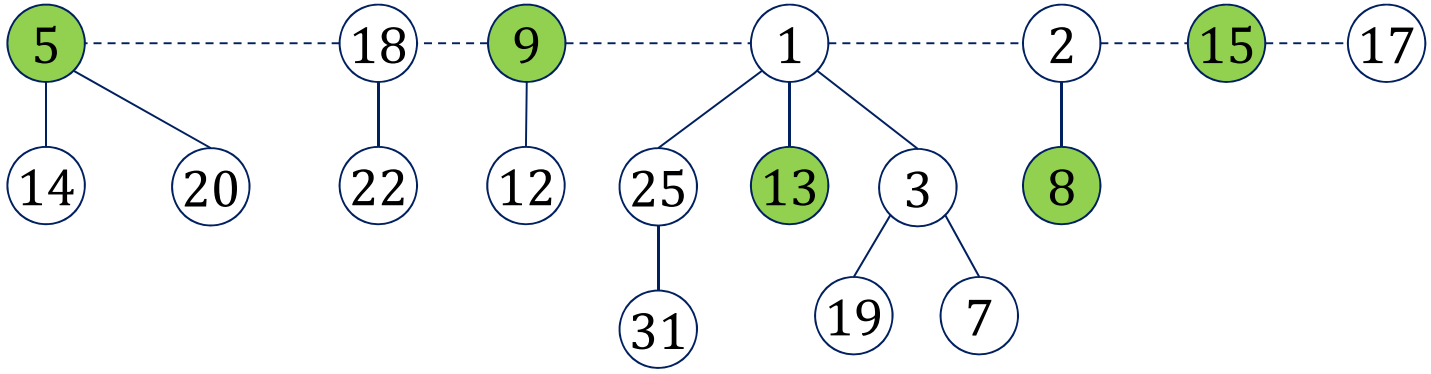
System state characterized by two parameters:

- $\boldsymbol{R}$**:** number of trees (length of $H.rootlist$)
- $\boldsymbol{M}$**:** number of marked nodes that are not in the root list

Potential function:

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

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: $t_i \leq D(n) + |H.rootlist|$

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

- R: changes from $H.rootlist$ to at most $D(n)$
- M: (# of marked nodes that are not in the root list)
  - no new marks
  - if node v is moved away from root list, $v.mark$ is set to false
    → value of M does not change!

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

Amortized Time:

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

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 $\Phi = R + 2M$:

- 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**: $\boldsymbol{O(1)}$

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

- **How large can $\boldsymbol{D(n)}$ get?**

Rank of Children

Lemma:

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

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

Proof:

Size of Trees

Fibonacci Numbers:

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

Lemma:

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

Proof:

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

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:

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

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: $S_k \geq F_{k+2}$

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.

Summary: Binomial and Fibonacci Heaps

	Binomial 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	$O(\log n)$	$O(1)$ *
merge	$O(\log n)$	$O(1)$
is-empty	$O(1)$	$O(1)$

* amortized time

Minimum Spanning Trees

Prim Algorithm:

1. Start with any node v (v is the initial component)
2. In each step:
   Grow the current component by adding the minimum weight edge e connecting the current component with any other node

Kruskal Algorithm:

1. Start with an empty edge set
2. In each step:
   Add minimum weight edge e such that e does not close a cycle

Implementation of Prim 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 are unmarked

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

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

5. mark node u

6. Until all nodes are marked

Implementation of Prim Algorithm

Implementation with Fibonacci heap:

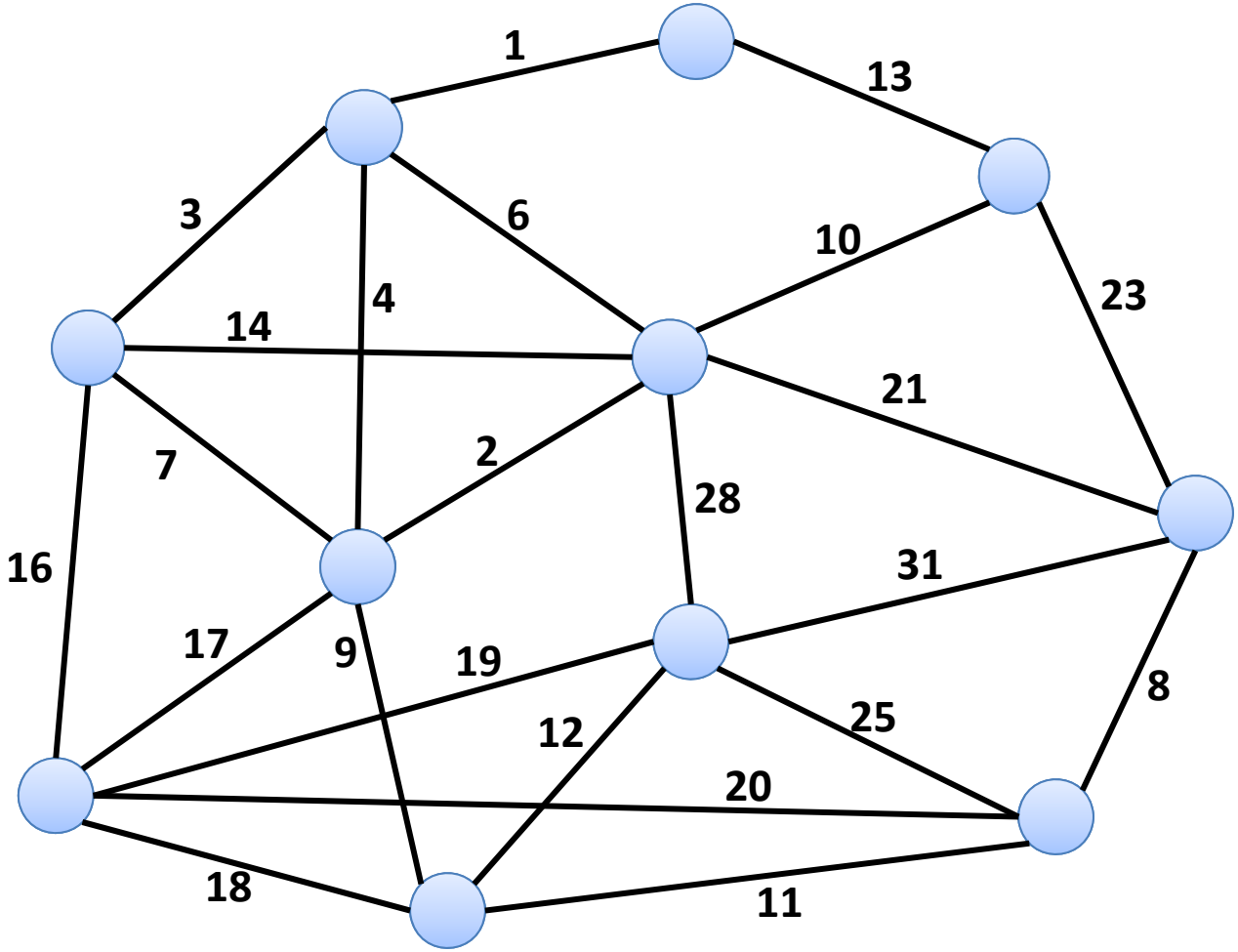
- Analysis identical to the analysis of Dijkstra's algorithm:

  $O(n)$ insert and delete-min operations

  $O(m)$ decrease-key operations

- Running time: $\boldsymbol{O(m + n \log n)}$

Kruskal Algorithm



1. Start with an empty edge set
2. In each step: Add minimum weight edge e such that e does *not* close a cycle

Implementation of Kruskal Algorithm

1. Go through edges in order of increasing weights

2. For each edge e:

   if e does not close a cycle then

      add e to the current solution

Union-Find Data Structure

Also known as **Disjoint-Set Data Structure**…

Manages partition of a set of elements

- set of disjoint sets

Operations:

- $\mathbf{make\_set}(x)$**:** create a new set that only contains element x
- $\mathbf{find}(x)$**:** return the set containing x
- $\mathbf{union}(x, y)$**:** merge the two sets containing x and y

Implementation of Kruskal Algorithm

1. Initialization:
   For each node v: $\mathrm{make\_set}(v)$

2. Go through edges in order of increasing weights:
   Sort edges by edge weight

3. For each edge $e = \{u, v\}$:

   if $\mathbf{find}(u) \neq \mathbf{find}(v)$ then

   add e to the current solution

   $\mathbf{union}(u, v)$

Managing Connected Components

- Union-find data structure can be used more generally to manage the connected components of a graph

  … if edges are added incrementally

- $\text{make\_set}(v)$ for every node v
- $\text{find}(v)$ returns component containing v
- $\text{union}(u, v)$ merges the components of u and v
  (when an edge is added between the components)

- Can also be used to manage biconnected components

Basic Implementation Properties

Representation of sets:

- Every set S of the partition is identified with a representative, by one of its members $x \in S$

Operations:

- $\text{make\_set}(x)$: x is the representative of the new set $\{x\}$
- $\text{find}(x)$: return representative of set S_x containing x
- $\text{union}(x, y)$: unites the sets S_x and S_y containing x and y and returns the new representative of $S_x \cup S_y$

Observations

Throughout the discussion of union-find:

- n: total number of make_set operations
- m: total number of operations (make_set, find, and union)

Clearly:

- $m \geq n$
- There are at most $n - 1$ union operations

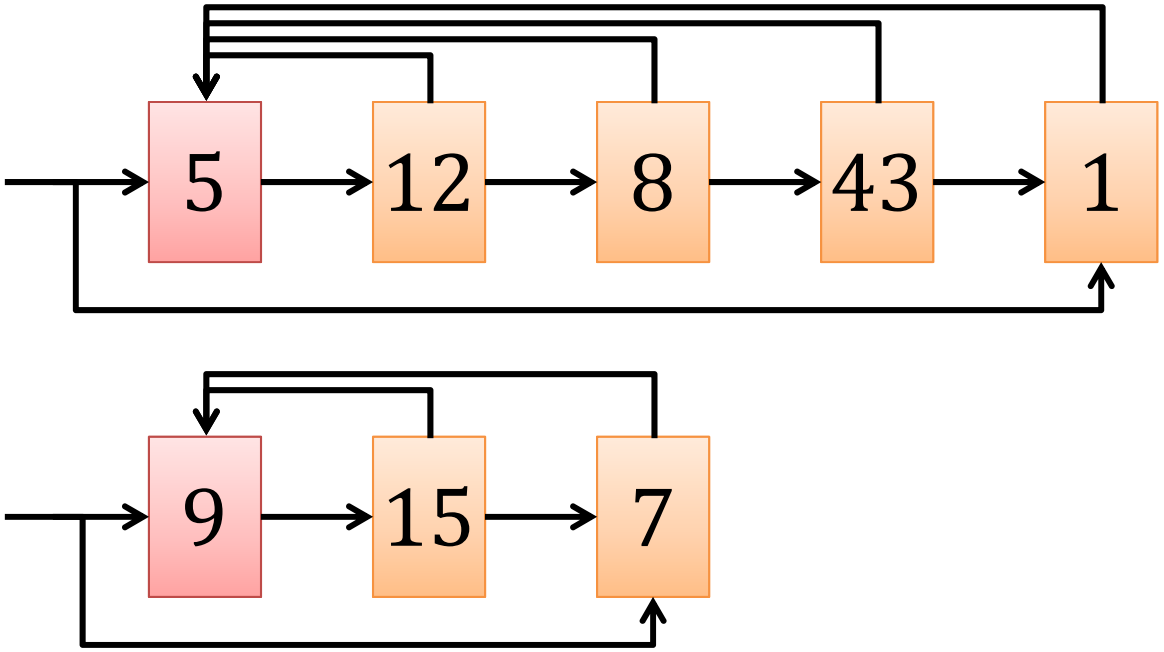
Remark:

- We assume that the n make_set operations are the first n operations
  - Does not really matter…

Linked List Implementation

Each set is implemented as a linked list:

- representative: first list element (all nodes point to first elem.)
  in addition: pointer to first and last element



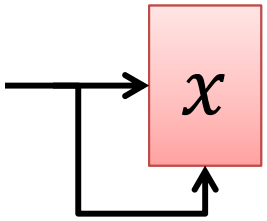
- sets: $\{1,5,8,12,43\}, \{7,9,15\}$; representatives: $5, 9$

Linked List Implementation

make_set(x):

- Create list with one element:

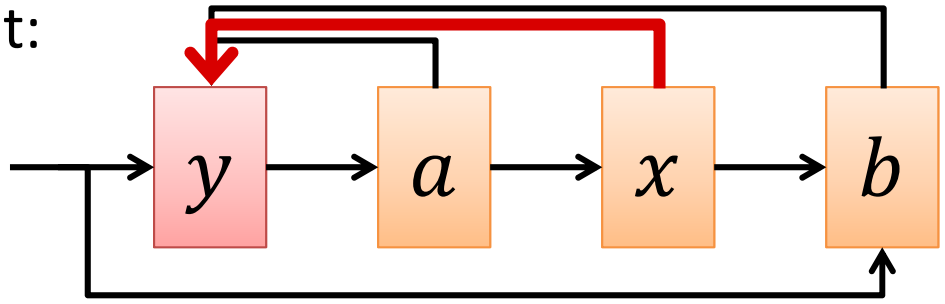
  time: $O(1)$



find(x):

- Return first list element:

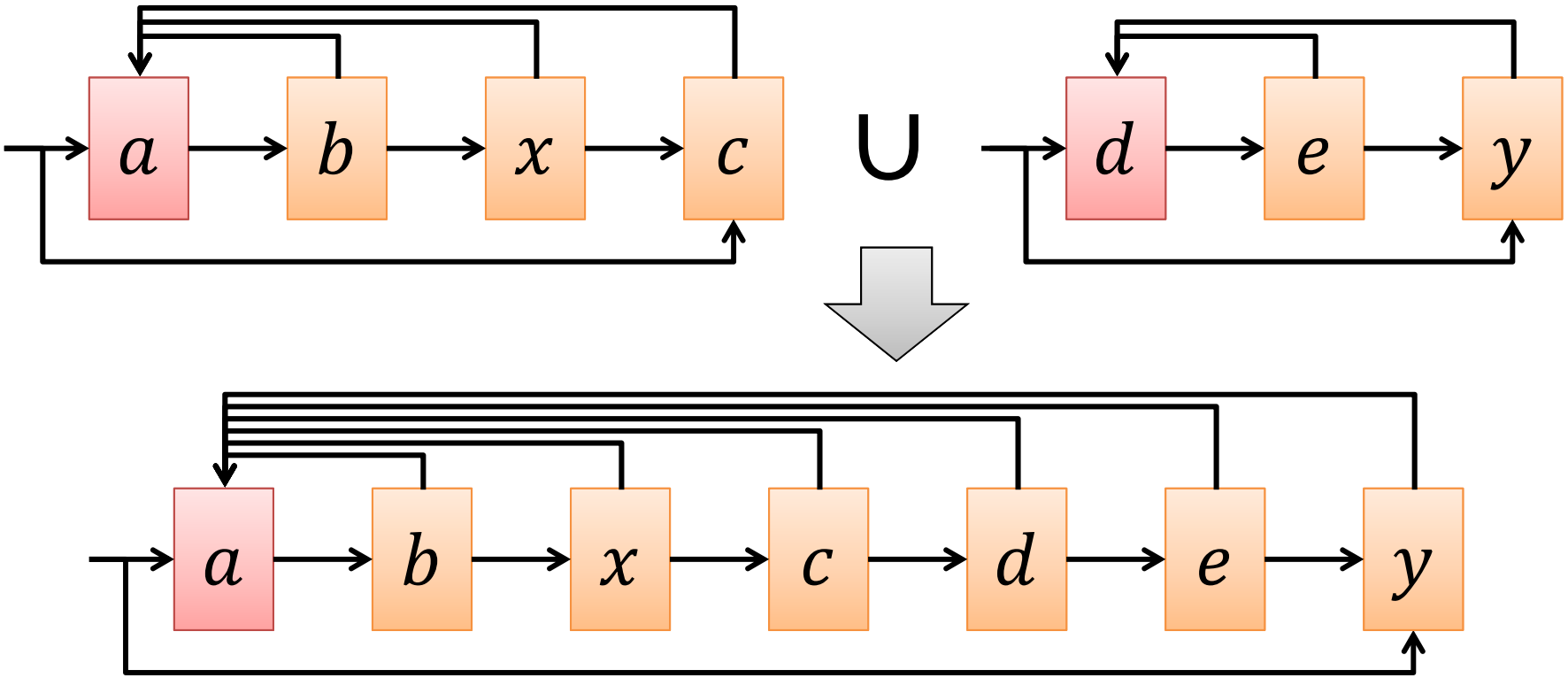
  time: $O(1)$



Linked List Implementation

union(x, y):

- Append list of y to list of x:



Time: O(length of list of y)

Cost of Union (Linked List Implementation)

Total cost for $n-1$ union operations can be $\Theta(n^2)$:

- $\text{make\_set}(x_1), \text{make\_set}(x_2), \dots, \text{make\_set}(x_n),$ $\text{union}(x_{n-1}, x_n), \text{union}(x_{n-2}, x_{n-1}), \dots, \text{union}(x_1, x_2)$

Weighted-Union Heuristic

- In a bad execution, average cost per union can be $\Theta(n)$
- Problem: The longer list is always appended to the shorter one

Idea:

- In each union operation, append shorter list to longer one!

Cost for union of sets S_x and S_y: $O\left(\min\{|S_x|, |S_y|\}\right)$

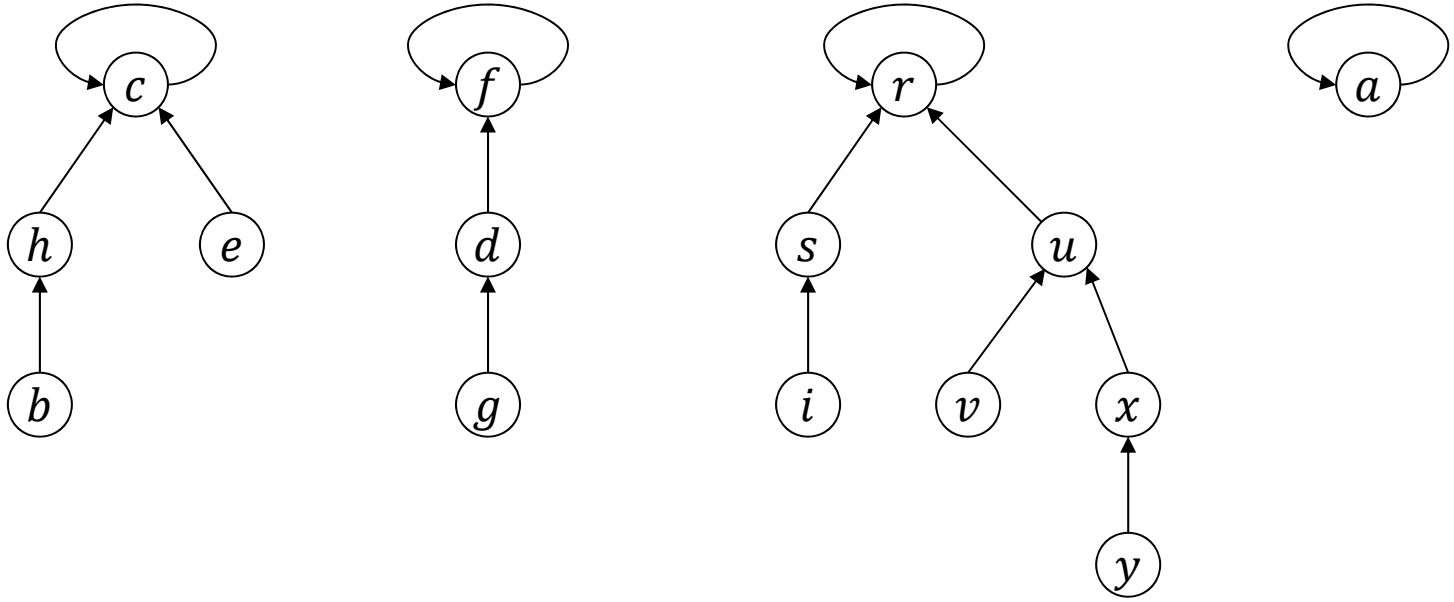
Theorem: The overall cost of m operations of which at most n are make_set operations is $\boldsymbol{O(m + n \log n)}$.

Weighted-Union Heuristic

Theorem: The overall cost of m operations of which at most n are make_set operations is $\boldsymbol{O(m + n \log n)}$.

Proof:

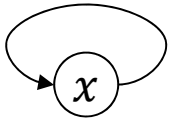
Disjoint-Set Forests



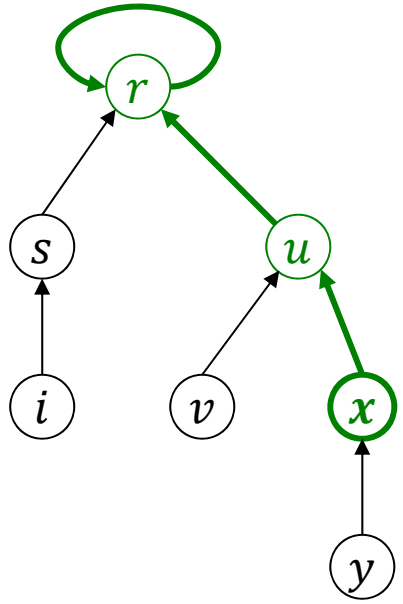
- Represent each set by a tree
- Representative of a set is the root of the tree

Disjoint-Set Forests

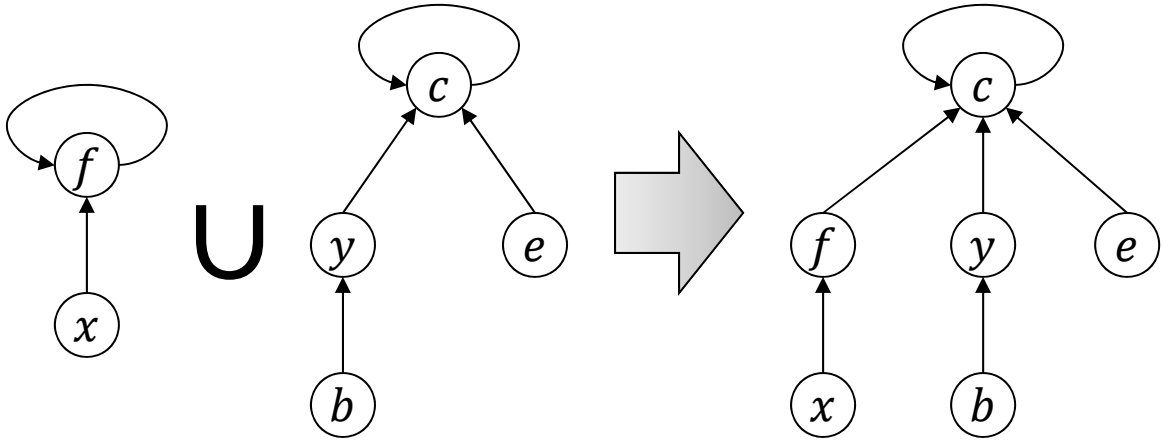
make_set(x): create new one-node tree



find(x): follow parent point to root
(parent pointer to itself)



union(x, y): attach tree of x to tree of y



Bad Sequence

Bad sequence leads to tree(s) of depth $\Theta(n)$

- $\text{make\_set}(x_1), \text{make\_set}(x_2), \dots, \text{make\_set}(x_n),$ $\text{union}(x_1, x_2), \text{union}(x_1, x_3), \dots, \text{union}(x_1, x_n)$

Union-By-Size Heuristic

Union of sets S_1 and S_2:

- Root of trees representing S_1 and S_2: r_1 and r_2
- W.l.o.g., assume that $|S_1| \geq |S_2|$
- Root of $S_1 \cup S_2$: r_1 (r_2 is attached to r_1 as a new child)

Theorem: If the union-by-rank heuristic is used, the **worst-case cost of a find-operation is $O(\log n)$**

Proof:

Union-Find Algorithms

Recall: m operations, n of the operations are make_set-operations

Linked List with Weighted Union Heuristic:

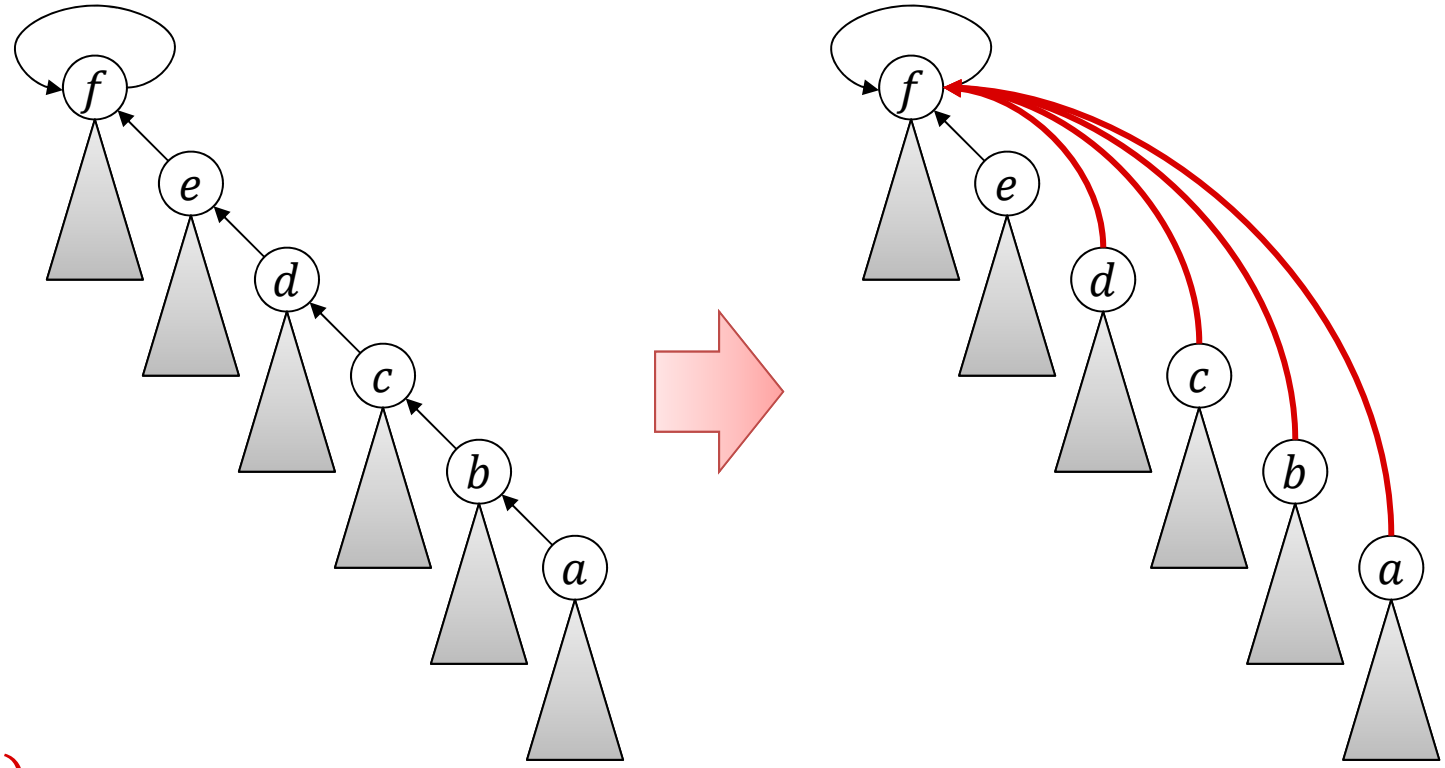
- make_set: worst-case cost $O(1)$
- find : worst-case cost $O(1)$
- union : amortized worst-case cost $O(\log n)$

Disjoint-Set Forest with Union-By-Size Heuristic:

- make_set: worst-case cost $O(1)$
- find : worst-case cost $O(\log n)$
- union : worst-case cost $O(\log n)$

Can we make this faster?

Path Compression During Find Operation



find(a):

1. **if** $a \neq a.parent$ **then**
2. $\quad a.parent := \text{find}(a.parent)$
3. **return** $a.parent$

Complexity With Path Compression

When using only path compression (without union-by-rank):

m: total number of operations

- f of which are find-operations
- n of which are make_set-operations
  → at most $n-1$ are union-operations

Total cost: $\mathbf{O\left(n + f \cdot \left\lceil \log_{2+f/n} n \right\rceil\right) = O\left(m + f \cdot \log_{2+m/n} n\right)}$

Union-By-Size and Path Compression

Theorem:

Using the combined union-by-size and path compression heuristic, the running time of m disjoint-set (union-find) operations on n elements (at most n make_set-operations) is

$$\boldsymbol{\Theta\big(m \cdot \alpha(m,n)\big)},$$

Where $\alpha(m,n)$ is the inverse of the Ackermann function.

Ackermann Function and its Inverse

Ackermann Function:

For $k, \ell \geq 1$,

$$A(k,\ell) := \begin{cases} 2^{\ell}, & \text{if } k = 1, \ell \geq 1 \\ A(k-1,2), & \text{if } k > 1, \ell = 1 \\ A\big(k-1, A(k,\ell-1)\big), & \text{if } k > 1, \ell > 1 \end{cases}$$

Inverse of Ackermann Function:

$$\alpha(m,n) := \min\{k \geq 1 \mid A(k, \lfloor m/n \rfloor) > \log_2 n\}$$

Inverse of Ackermann Function

- $\alpha(m,n) := \min\{k \geq 1 \mid A(k, \lfloor m/n \rfloor) > \log_2 n\}$

  $m \geq n \Rightarrow A(k, \lfloor m/n \rfloor) \geq A(k,1) \Rightarrow \alpha(m,n) \leq \min\{k \geq 1 | A(k,1) > \log n\}$

- $A(1,\ell) = 2^\ell, \quad A(k,1) = A(k-1,2),$
  $A(k,\ell) = A\big(k-1, A(k,\ell-1)\big)$