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: $G=(V,E)$, $s \in V$

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

   initialize, insert all nodes v with init. key ∞

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

   get-min

4. mark node u

   delete-min

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

   for all neighbors of u : decrease-key

6. Until all nodes are marked   is-empty

Analysis

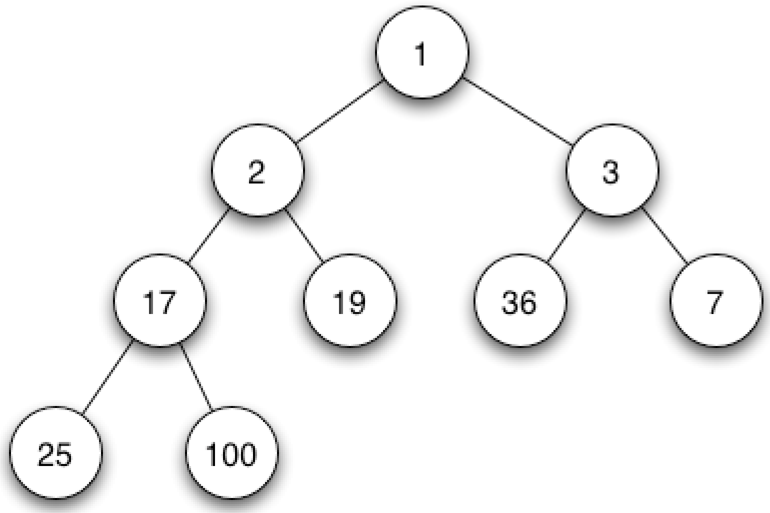
Number of priority queue operations for Dijkstra:

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

Priority Queue Implementation

Implementation as min-heap:

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



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

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…

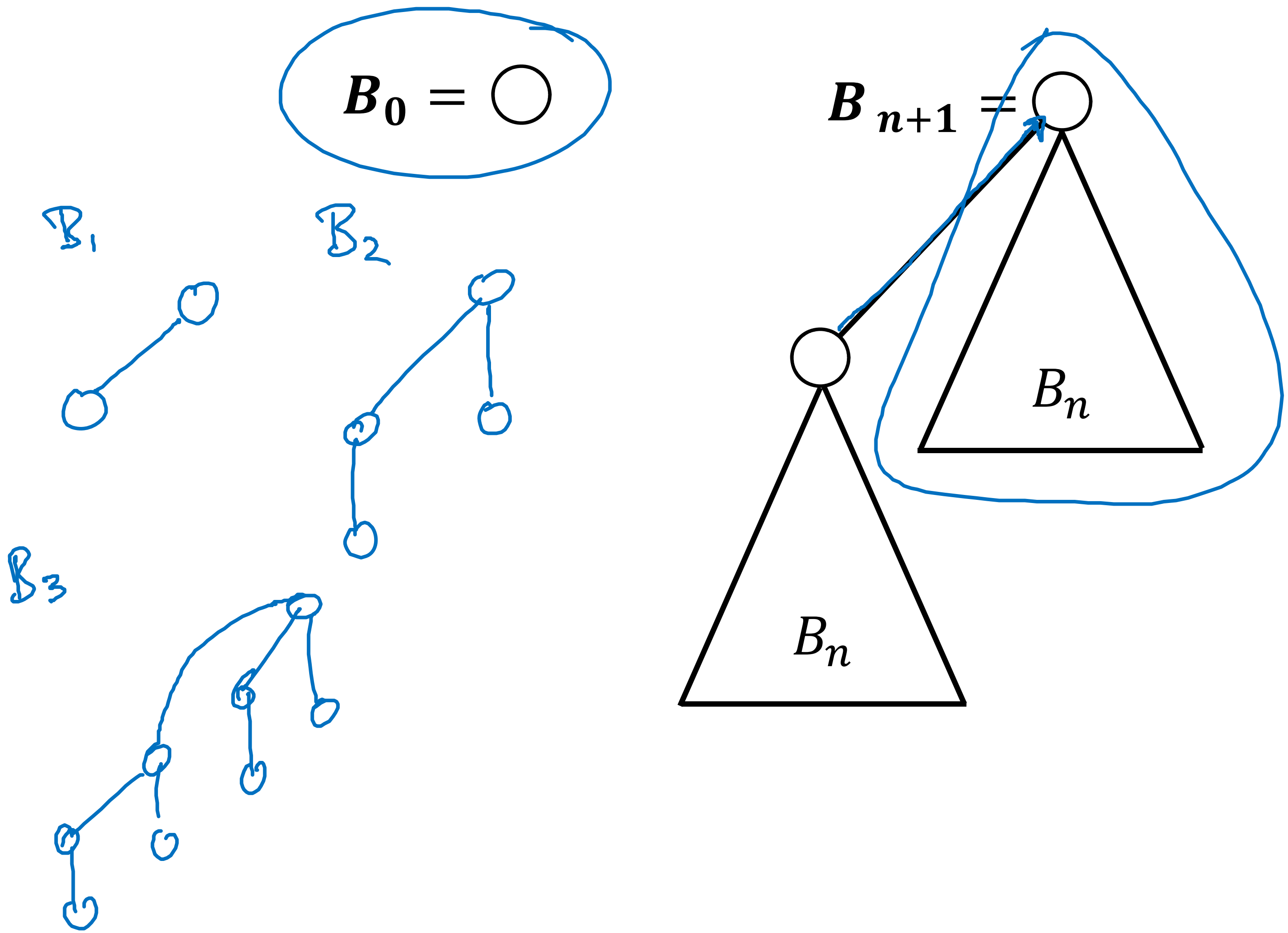
- Cost of merging two heaps is expensive

- We will get there in two steps:

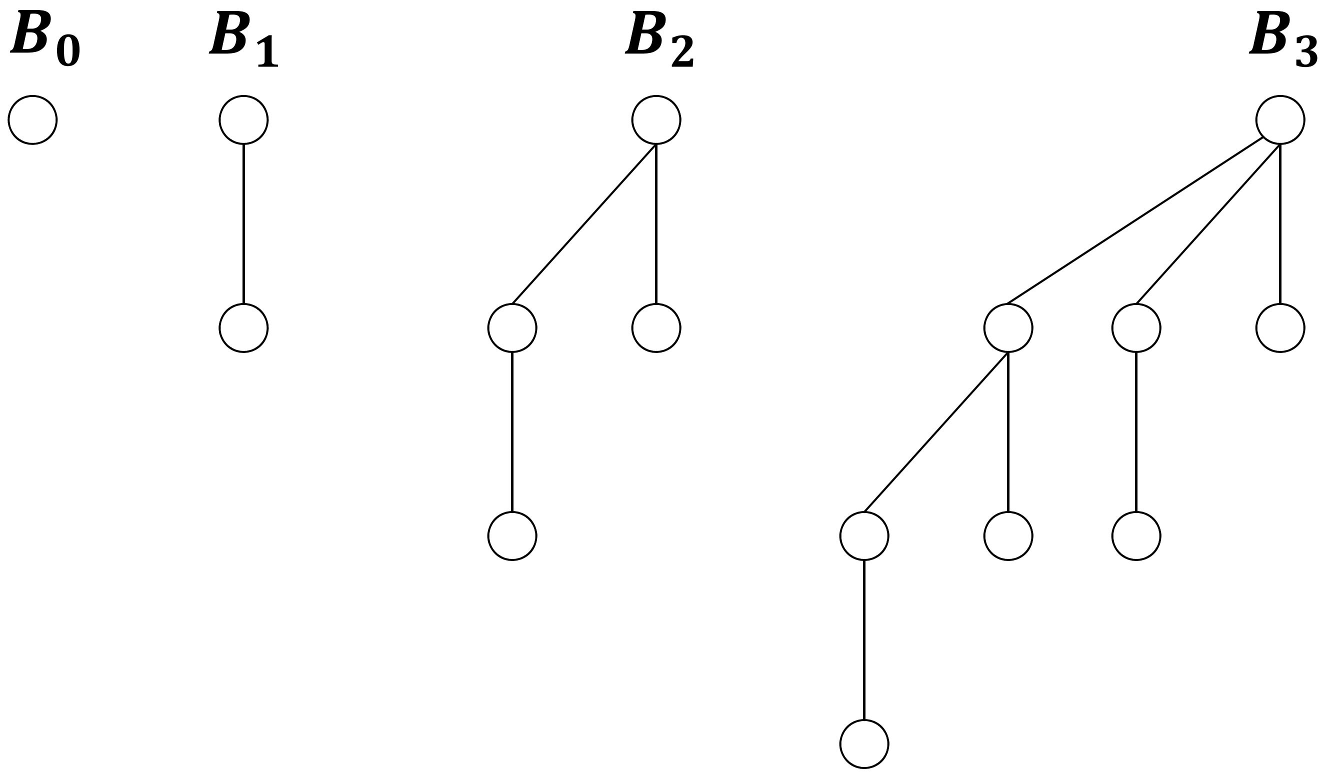
Binomial heap → Fibonacci heap

Definition: Binomial Tree

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



Binomial Trees



Properties

1. Tree B_n has 2^n nodes

induction on n : n = 0 ✓

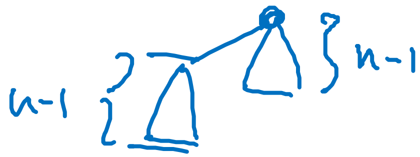
step:

$|B_n| = 2|B_{n-1}|$

2. Height of tree B_n is n

n = 0 ✓

step:

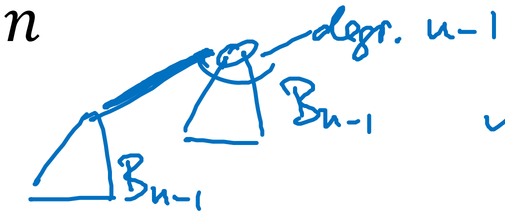


✓

3. Root degree of B_n is n

n = 0 ✓

step:



✓

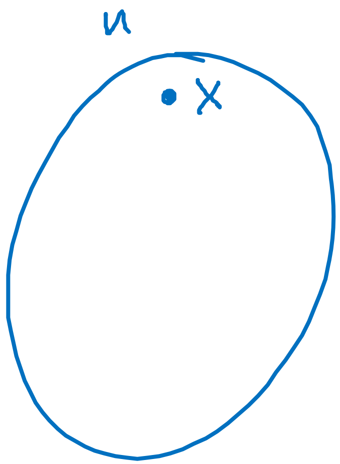
4. In B_n, there are exactly $\binom{n}{i}$ nodes at depth i

Binomial Coefficients

- Binomial coefficient:

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

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



subsets that contain x

$\binom{n-1}{k-1}$

subsets without x

$\binom{n-1}{k}$

Pascal triangle:

1

1 1

1 2 1

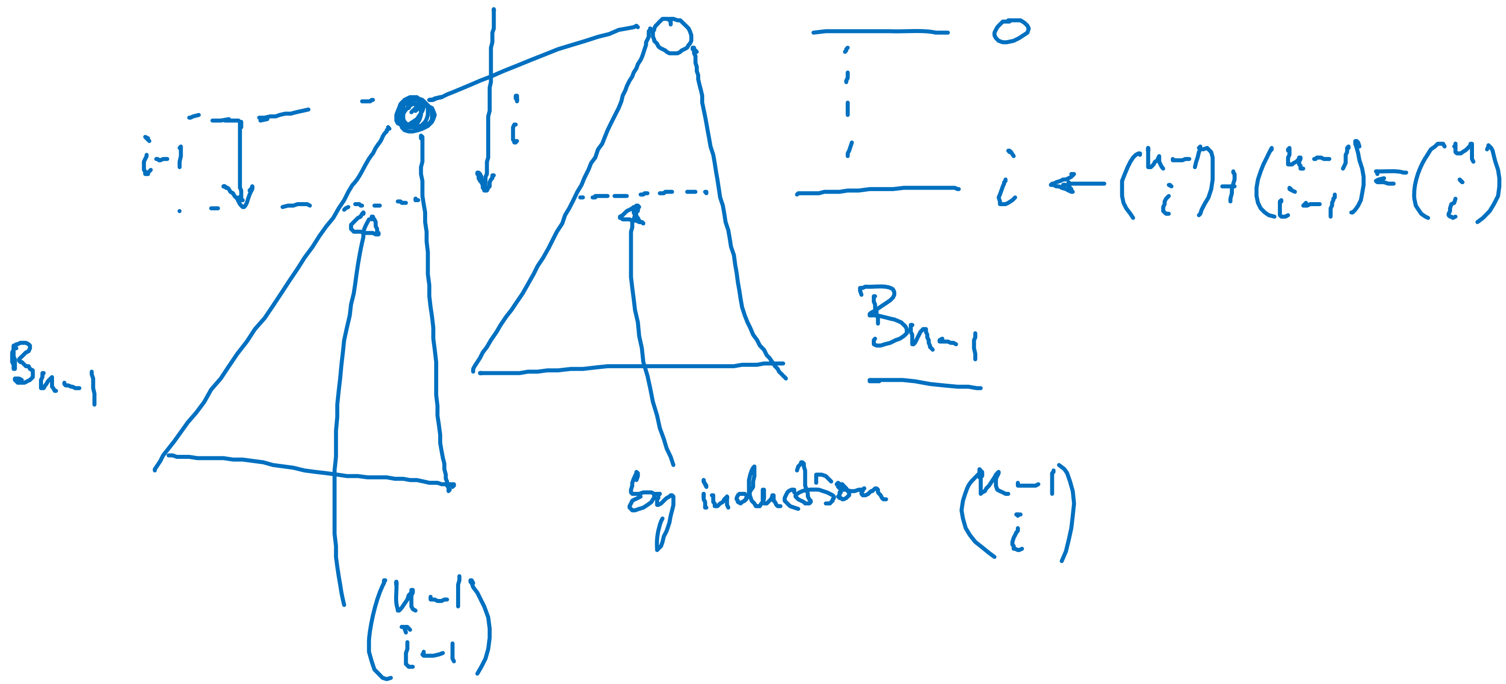
1 3 3 1

1 4 6 4 1

1 5 10 10 5 1

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

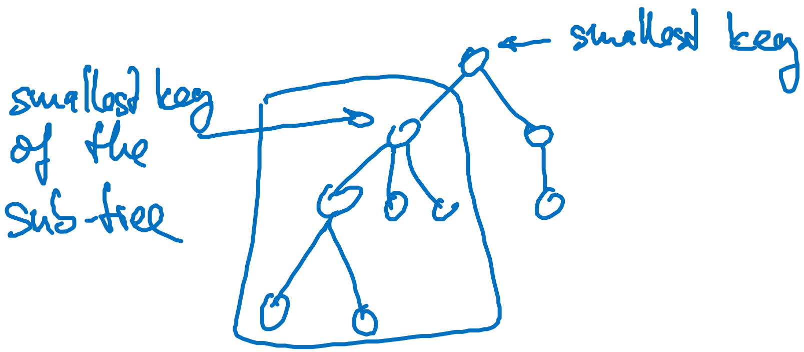
- Keys are stored in nodes of binomial trees of different order

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

  $n = 21 = 2^4 + 2^2 + 2^0 = (10101)_2$ (B_4, B_2, B_0) $|B_i| = 2^i$

- **Min-Heap Property:**

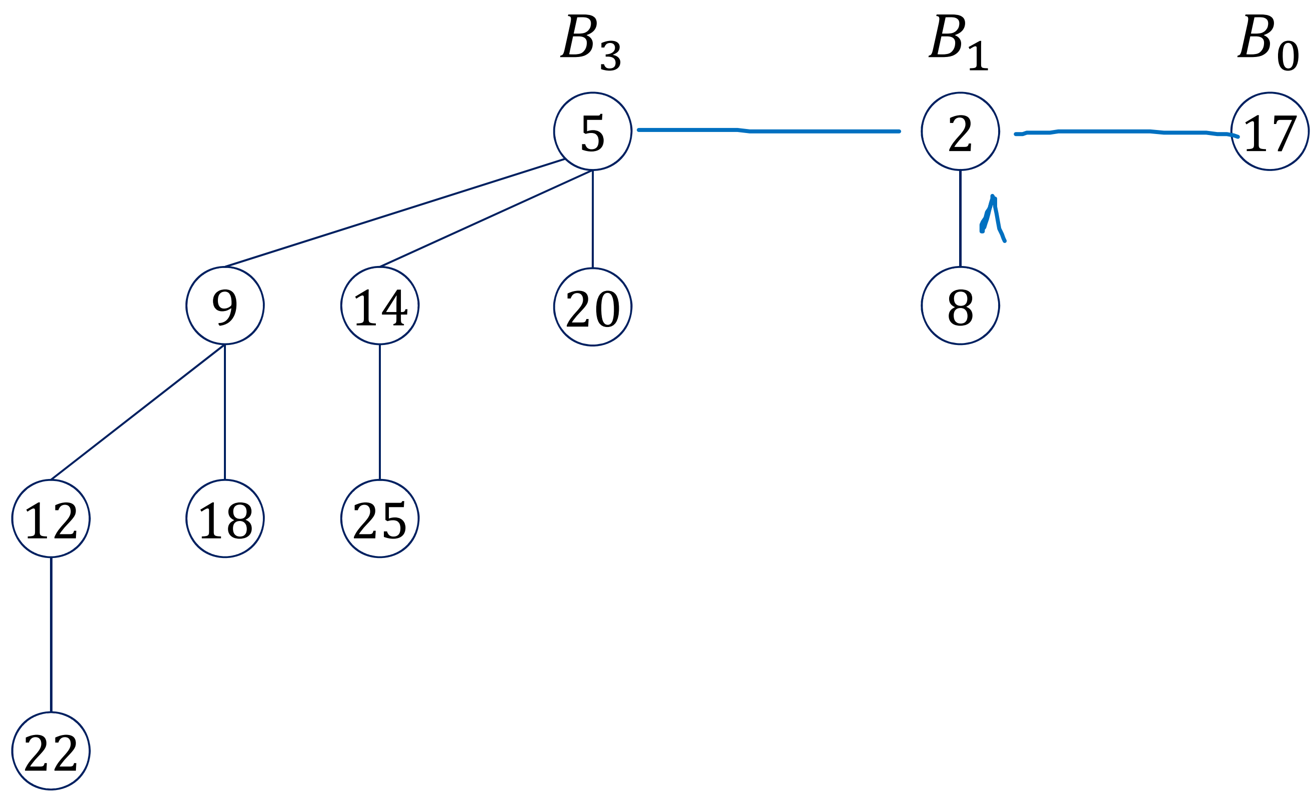
  Key of node $v \leq$ keys of all nodes in sub-tree of v



Example

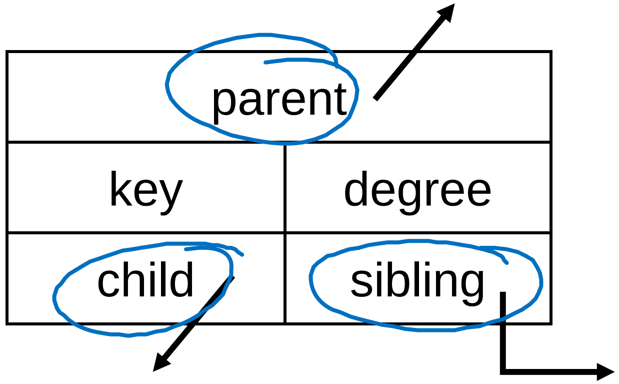
- 10 keys: $\{2, 5, 8, 9, 12, 14, 17, 18, 20, 22, 25\}$

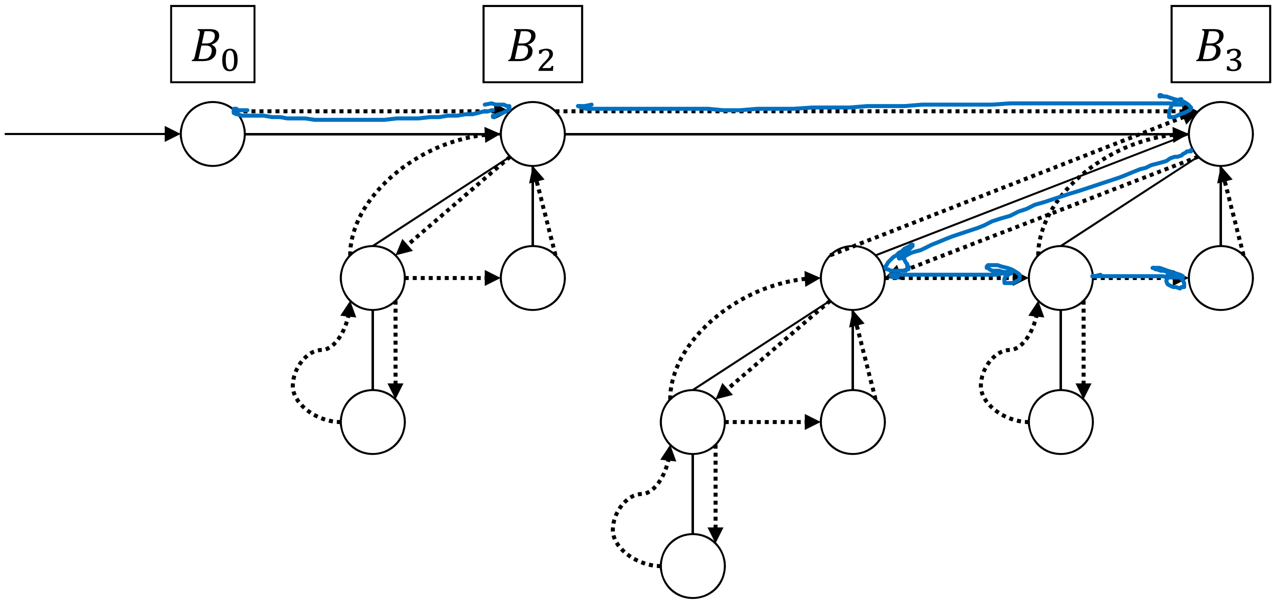
- Binary representation of n: $(11)_2 = 1011$
  → trees B_0, B_1, and B_3 present



Child-Sibling Representation

Structure of a node:



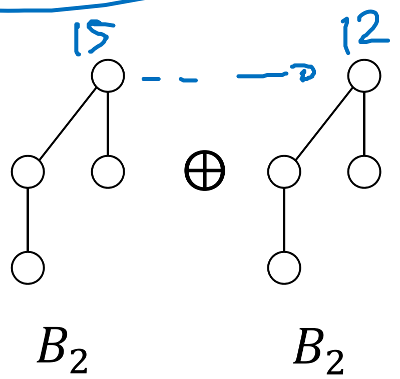


Link Operation

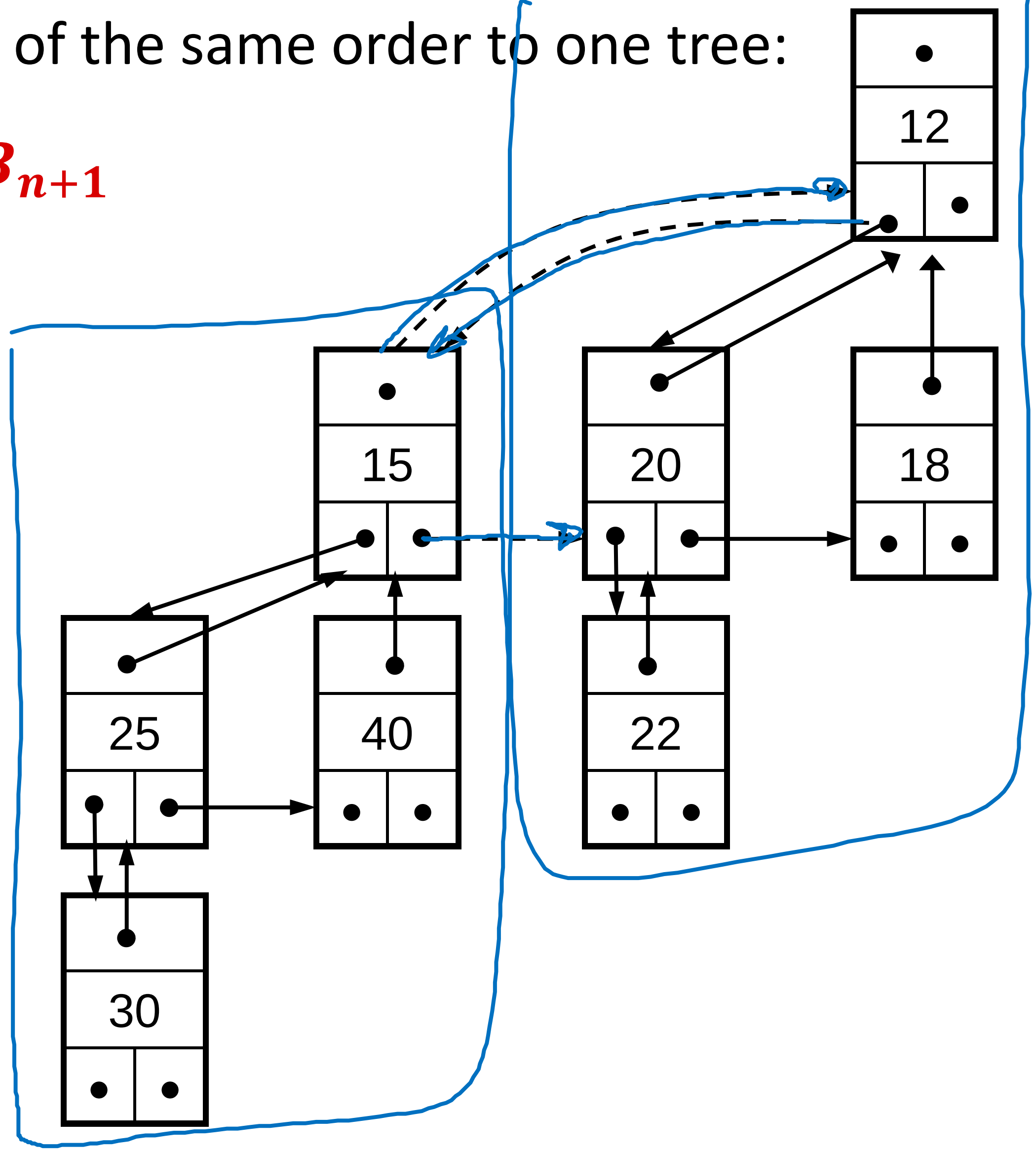
- Unite two binomial trees of the same order to one tree:

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

- Time: $O(1)$



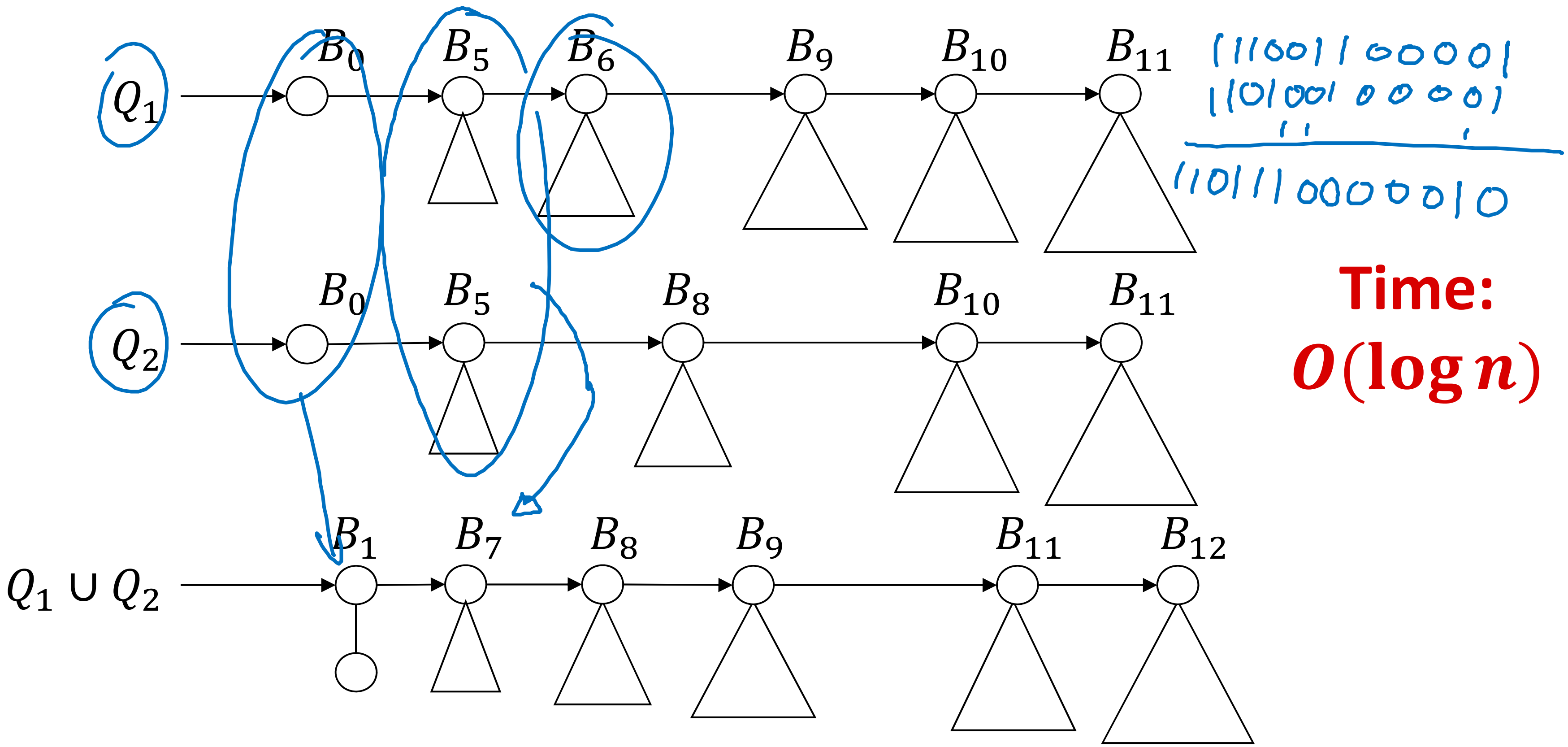
=



Merge Operation

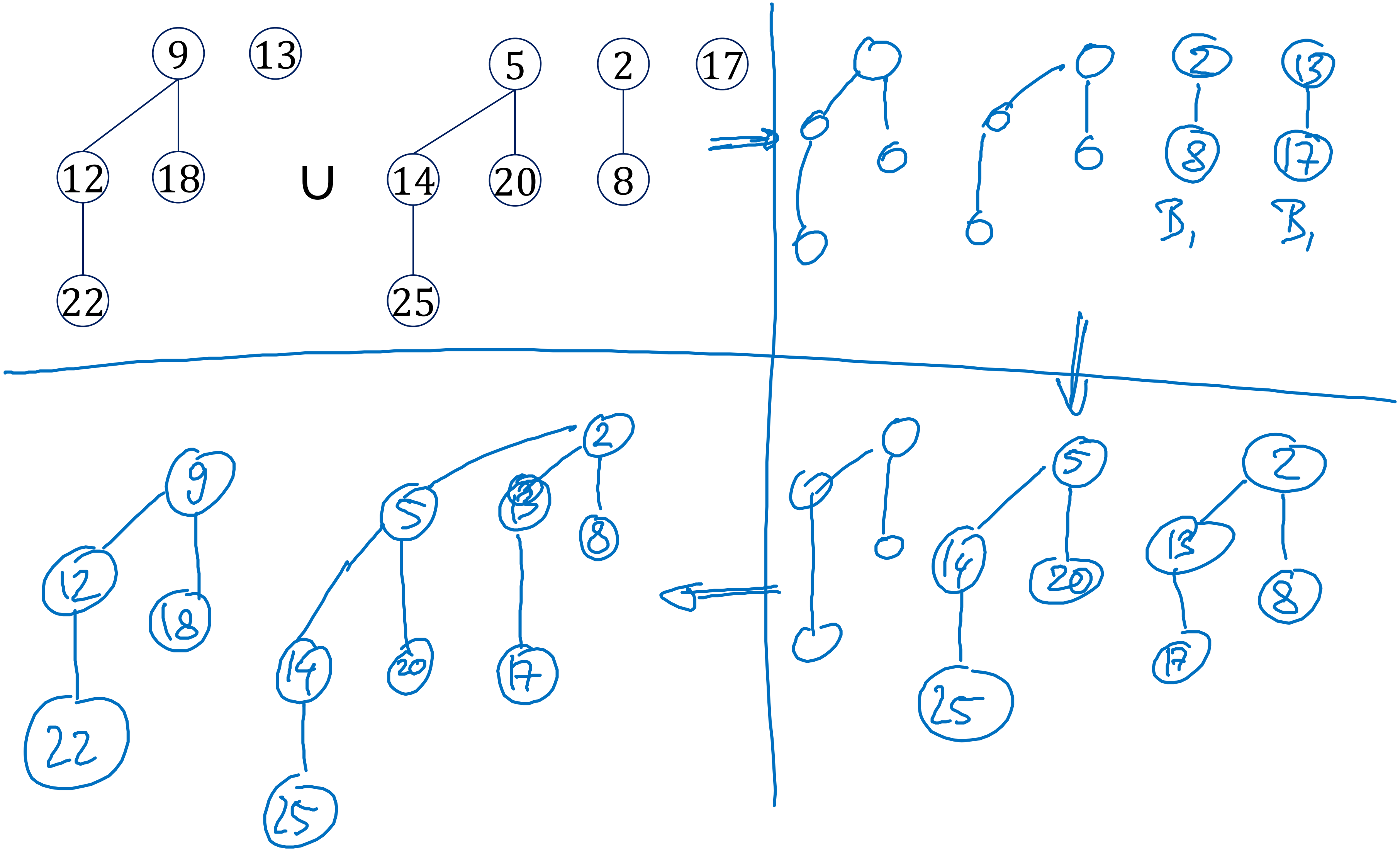
Merging two binomial heaps:

- **For $\boldsymbol{i = 0, 1, \dots, \log n}$:**
  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

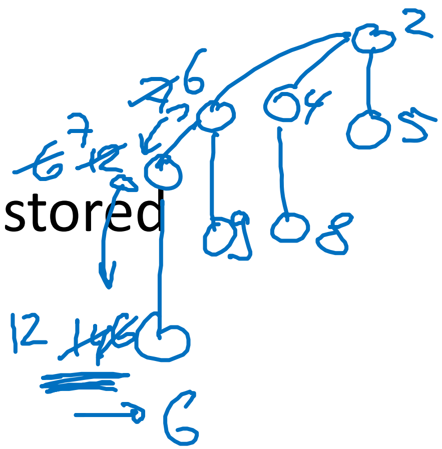


Operations

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 ← $O(1)$ time
2. Merge Q and Q' ← $O(\log n)$ time

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

Operations

Delete-Min Operation:

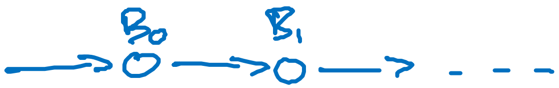
need to delete

1. Find tree B_i with minimum root r

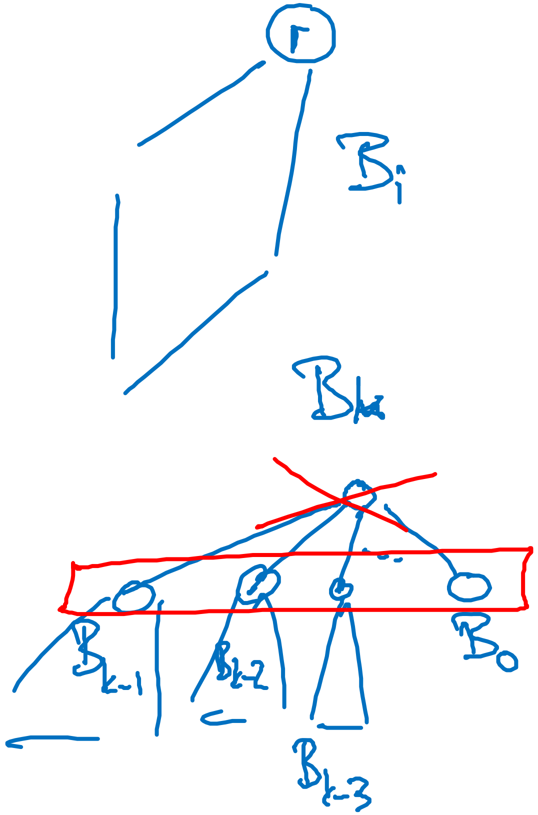
   get-min $O(1)$

2. Remove B_i from queue Q → queue Q'

3. Children of r form new queue Q''

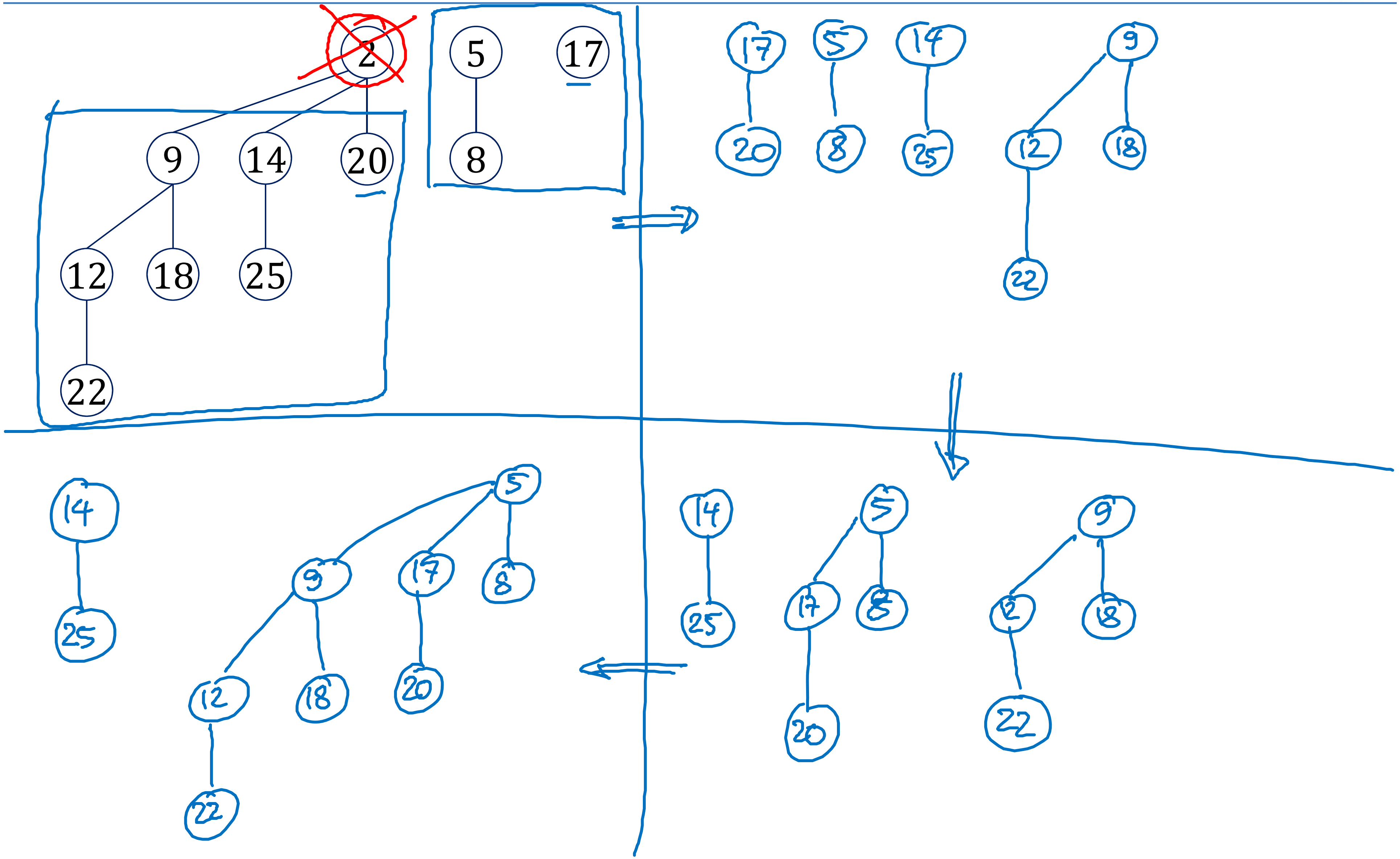


4. Merge queues Q' and Q''



- **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, $m \leq n$): $\boldsymbol{O(\log n)}$

Dijkstra is still $O(|E| \log |V|)$

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

$2^k - 1 \leftrightarrow 2^k$

Fibonacci Heaps

Lacy-merge variant of binomial heaps:

- Do not merge trees as long as possible…

Structure:

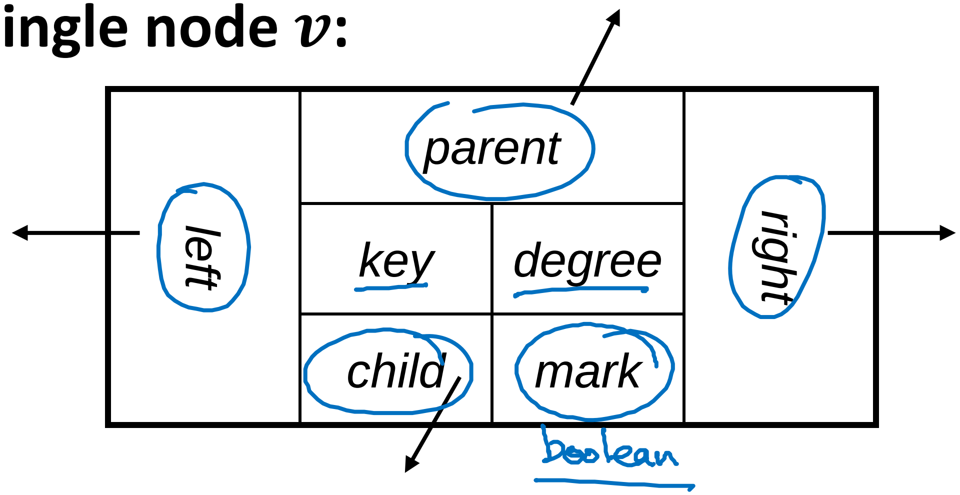
A Fibonacci heap H consists of a collection of trees satisfying the min-heap property.

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

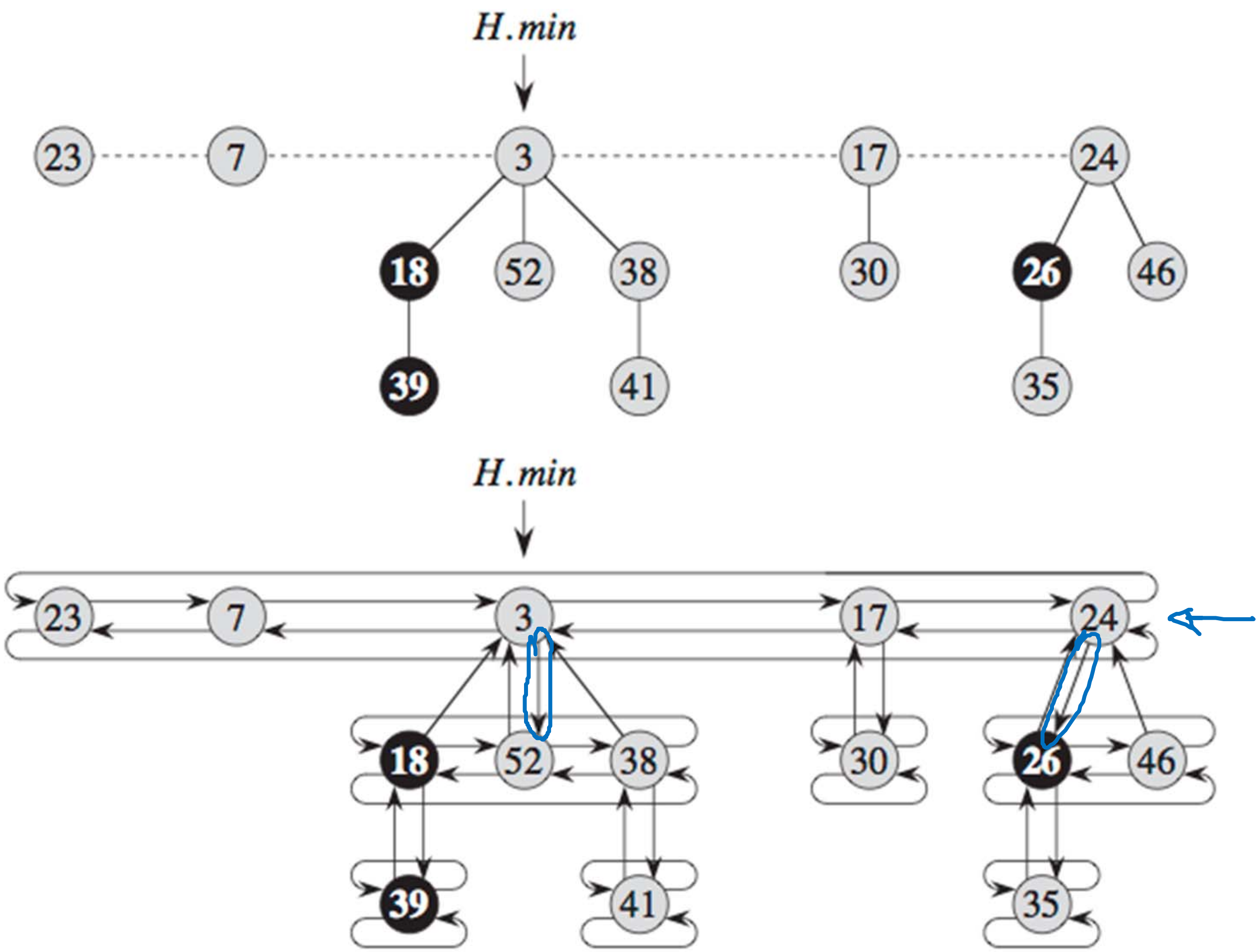


Figure: Cormen et al., Introduction to Algorithms

Simple (Lazy) Operations

Initialize-Heap H:

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

Merge heaps H and H':

- concatenate root lists
- update $H.min$

$O(1)$ time

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 $H.rootlist$; delete min
4. add $H.min.child$ (list) to $H.rootlist$ merge 2 heaps
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 $D(n)$

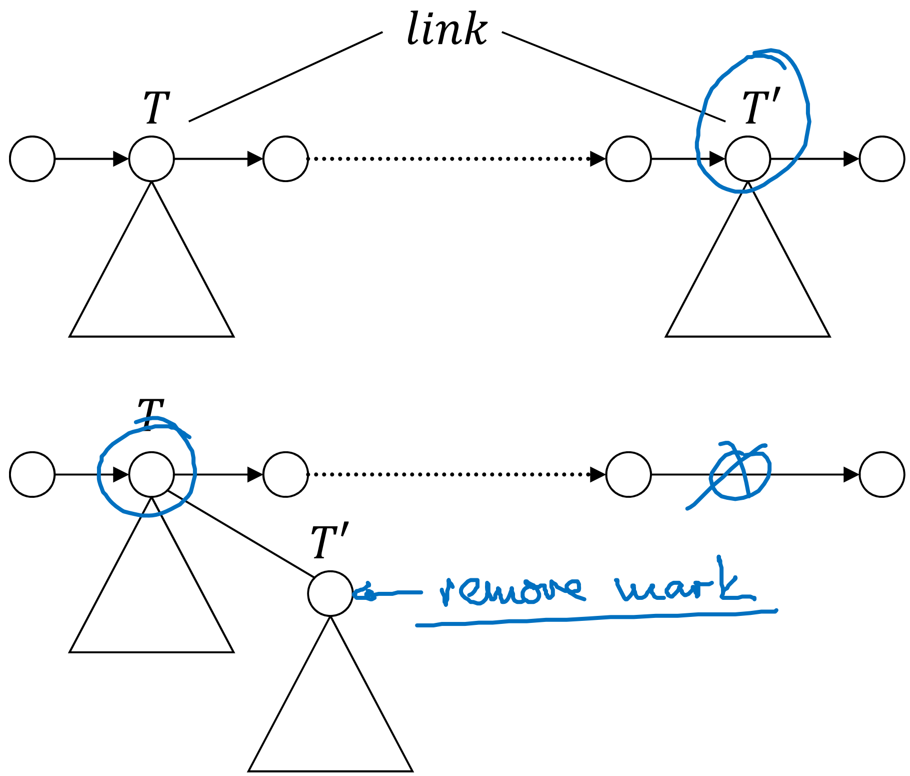
Merging Two Trees

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



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

root of T'

Consolidation of Root List

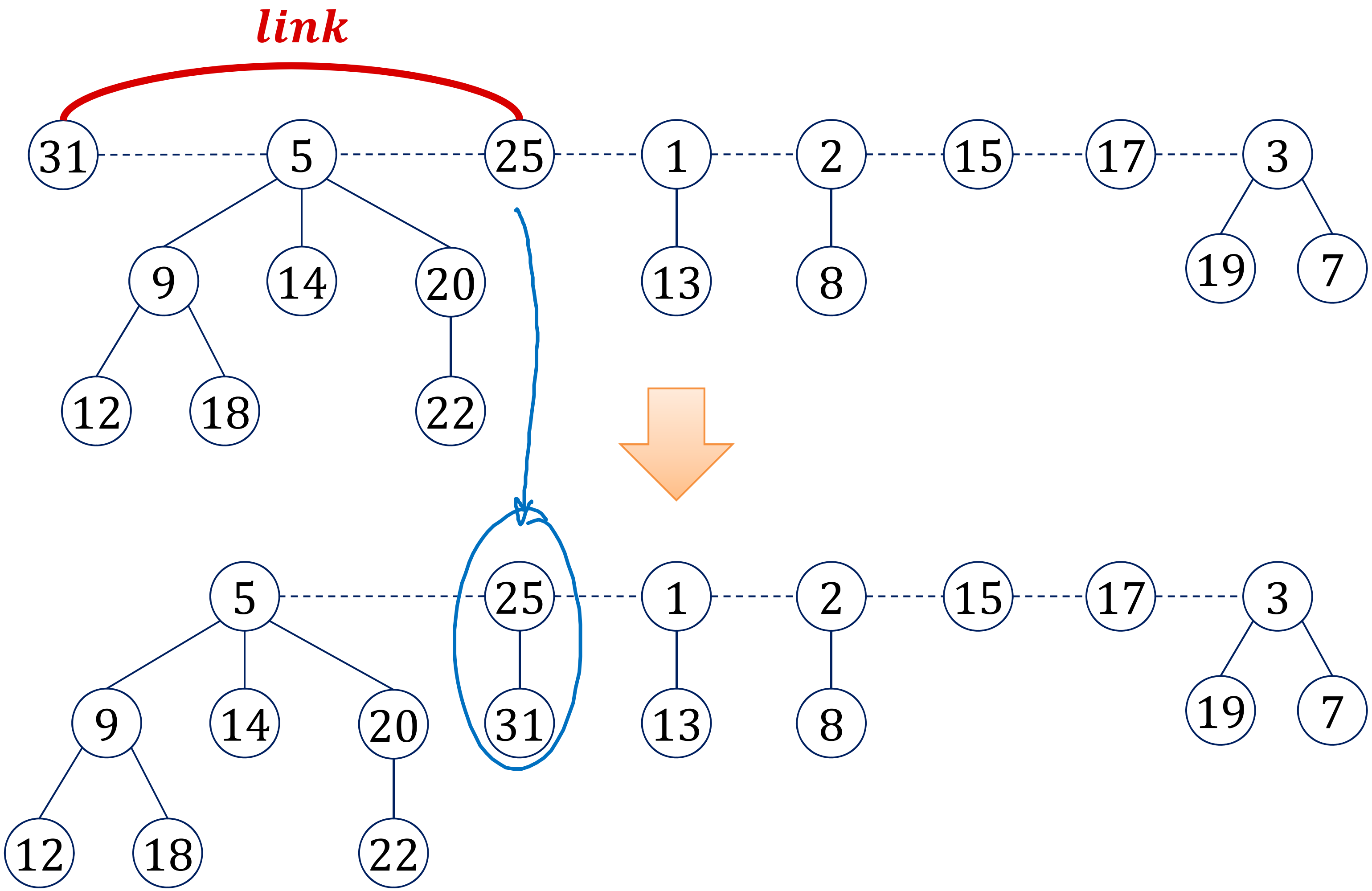
Array A pointing to find roots with the same rank:

Consolidate:

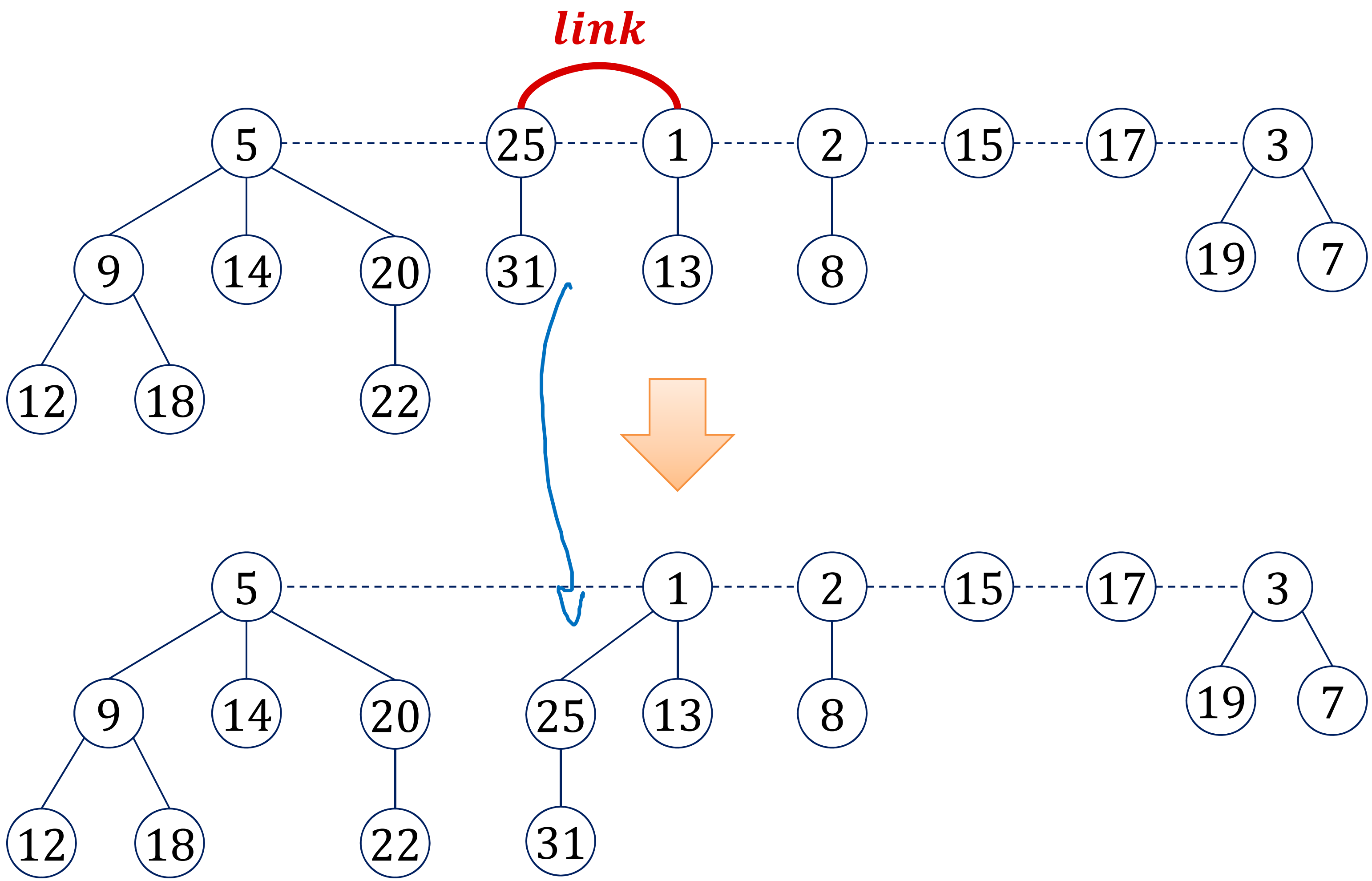
1. **for** $i := 0$ **to** $D(n)$ **do** $A[i] :=$ null;
2. **while** $H.rootlist \neq$ null **do**
3. $\quad T :=$ "delete and return first element of $H.rootlist$"
4. $\quad$ **while** $A[rank(T)] \neq$ null **do**
5. $\quad\quad T' := A[rank(T)];$
6. $\quad\quad A[rank(T)] := null;$
7. $\quad\quad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

Time:
$\boldsymbol{O(|H.rootlist| + D(n))}$

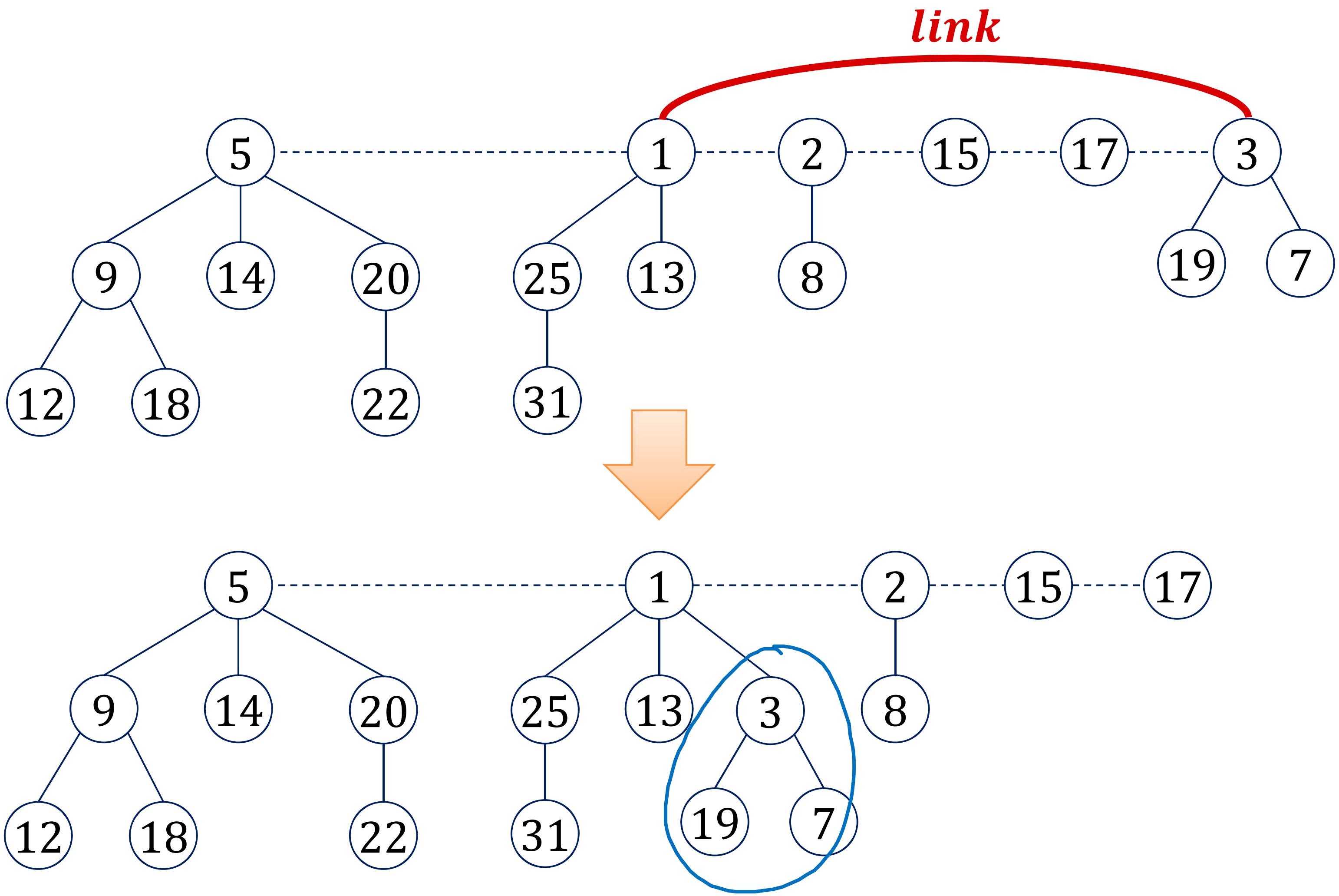
Consolidate Example



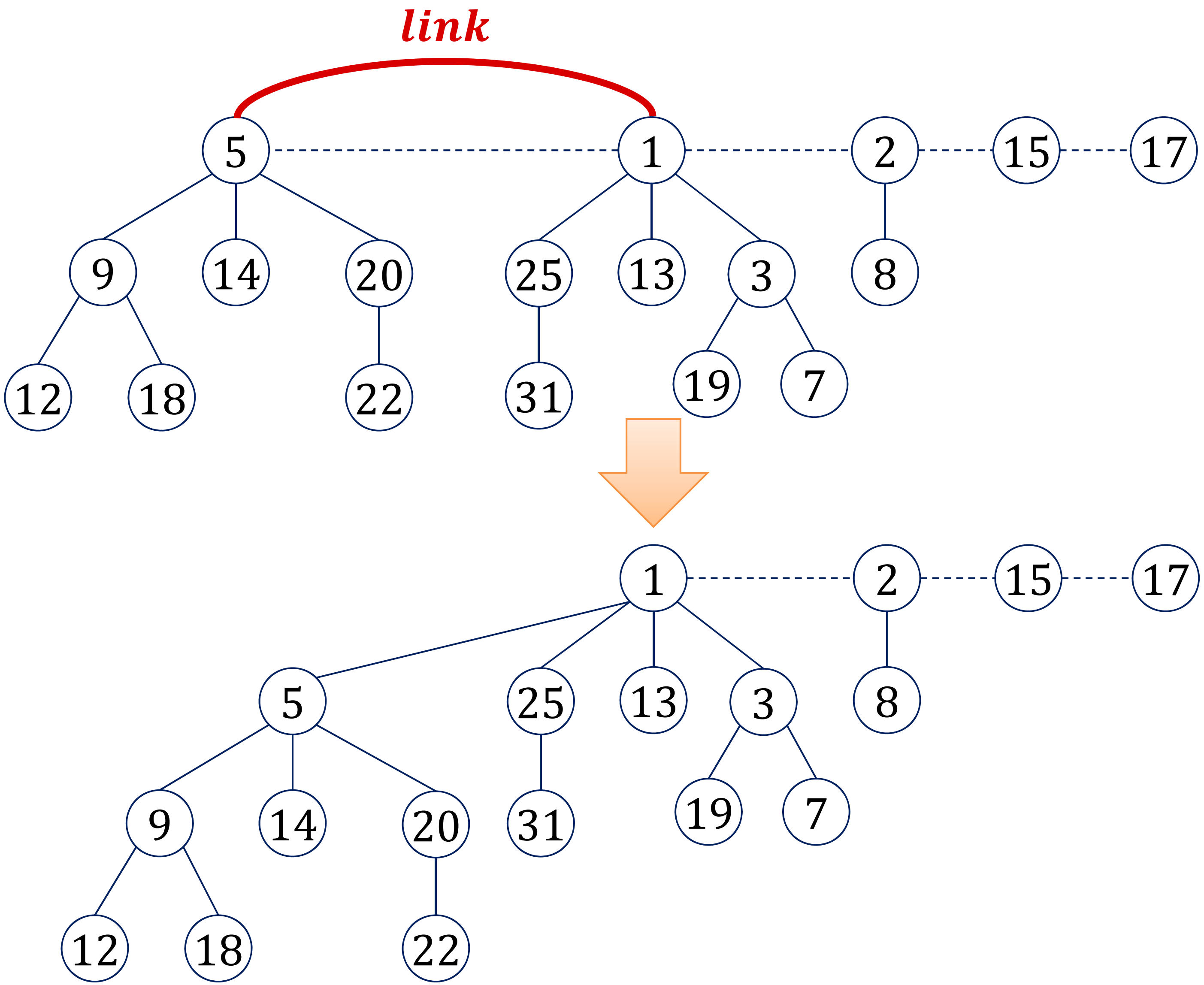
Consolidate Example



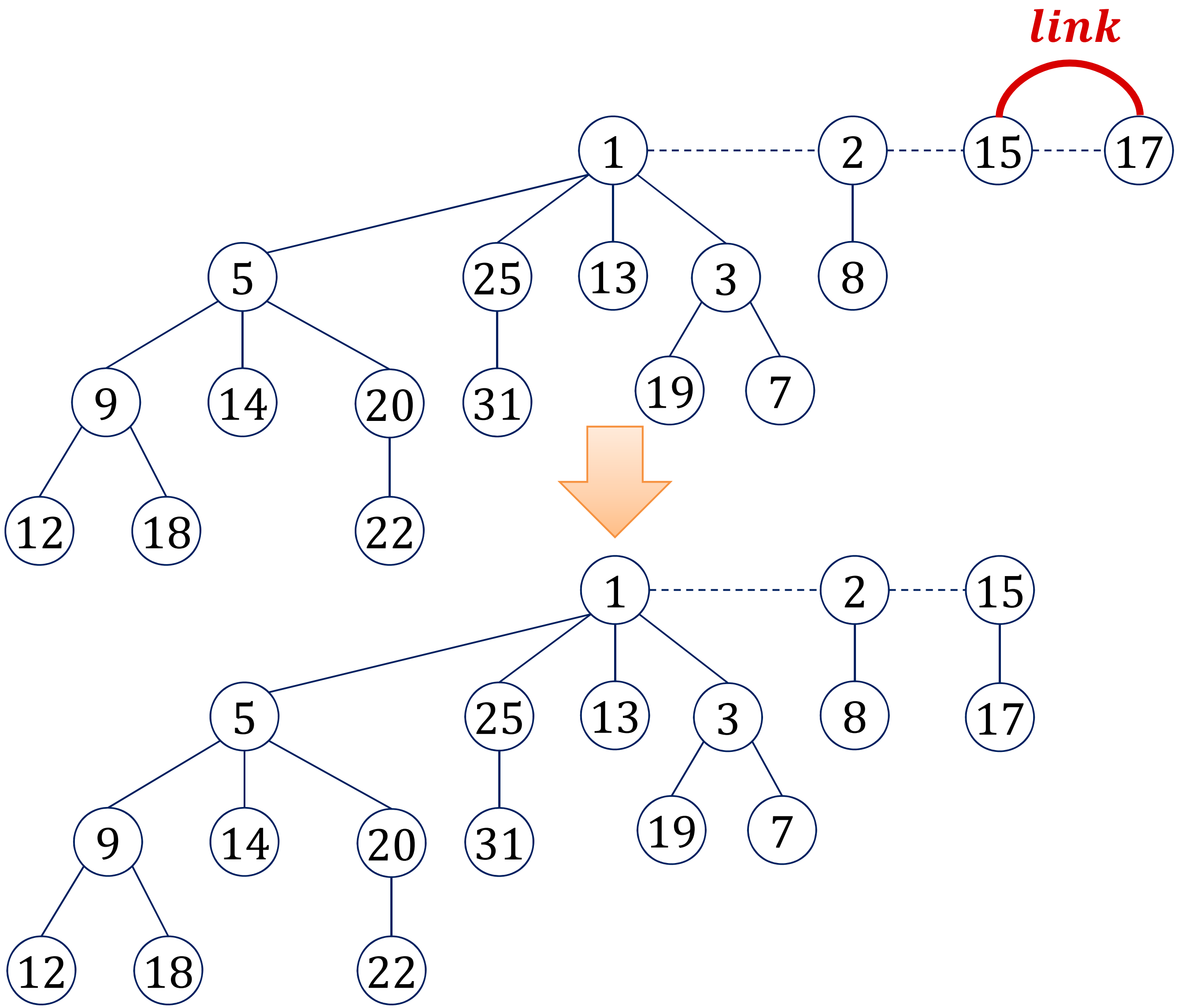
Consolidate Example



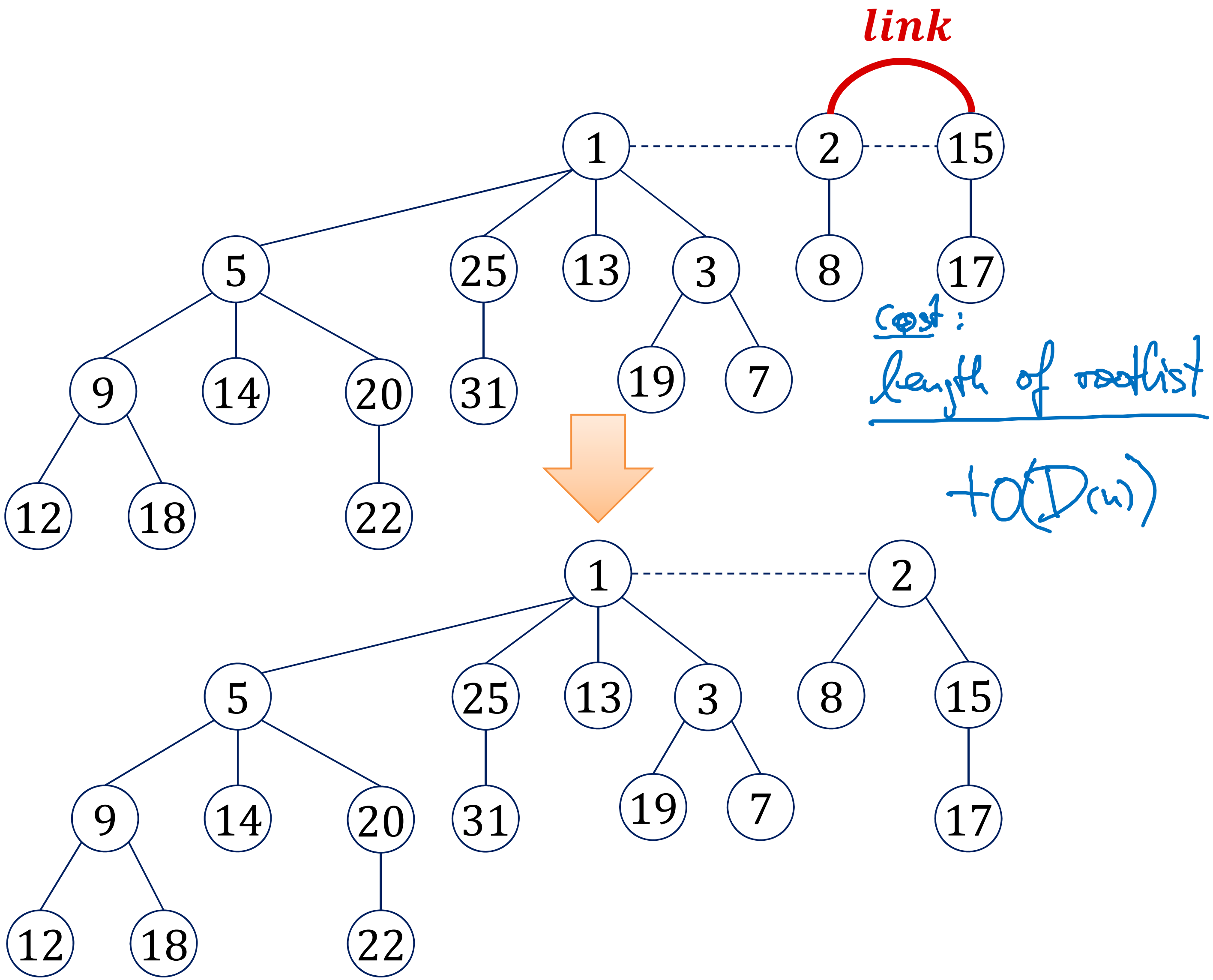
Consolidate Example



Consolidate Example



Consolidate Example



Operation Decrease-Key

Decrease-Key$(\boldsymbol{v}, \boldsymbol{x})$**:** (decrease key of node v to new value x)

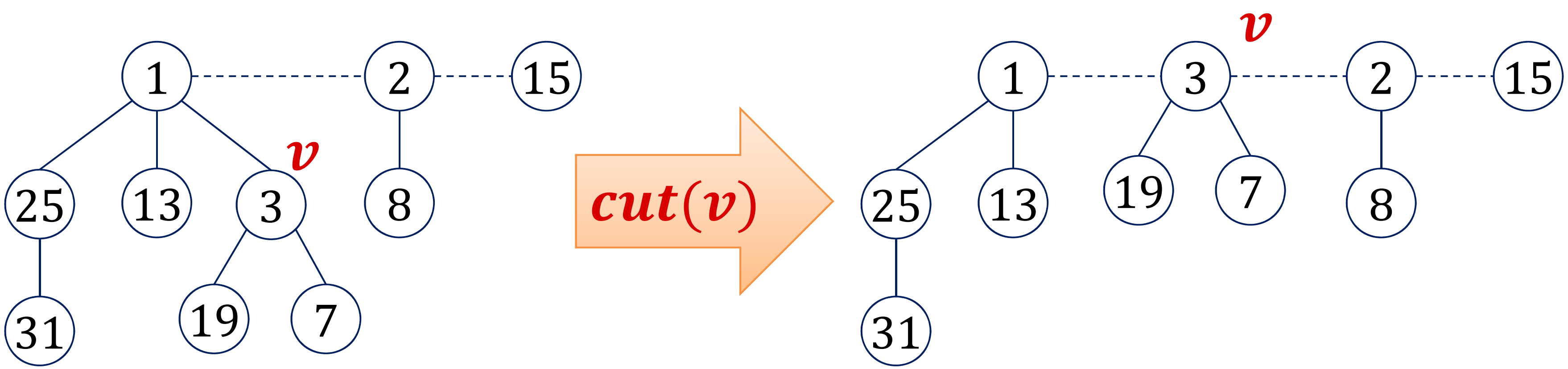
1. **if** $x \geq v.key$ **then return**;
2. $v.key := x$; update $H.min$;
3. **if** $v \in H.rootlist \lor x \geq v.parent.key$ **then return**
4. **repeat**
5. $\quad parent := v.parent$;
6. $\quad \boldsymbol{H.cut(v)}$;
7. $\quad v := parent$;
8. **until** $\neg(\boldsymbol{v.mark}) \lor v \in H.rootlist$;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

Operation Cut(v)

Operation $H.cut(v)$:

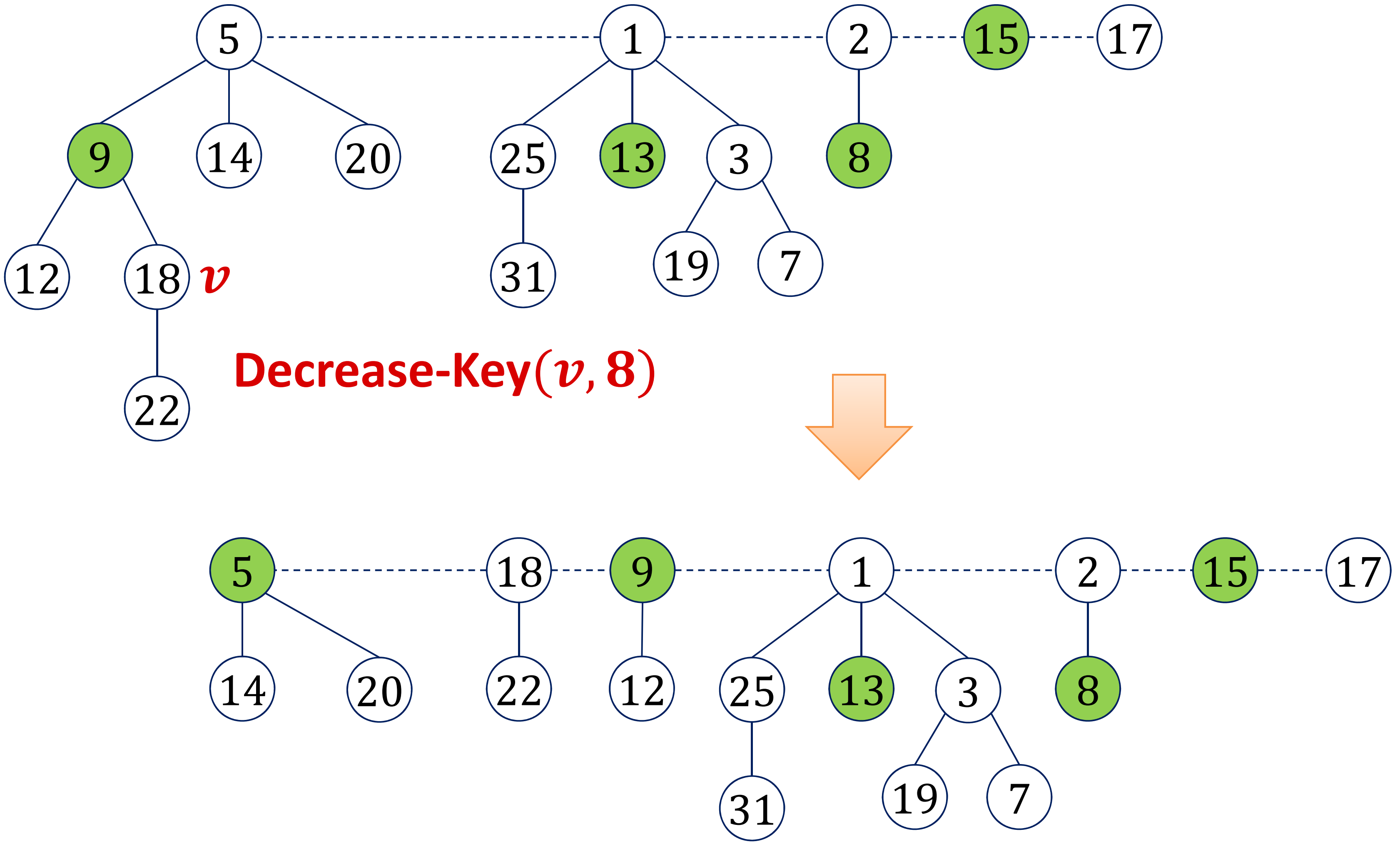
- Cuts v's sub-tree from its parent and adds v to rootlist

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



Decrease-Key Example

- Green nodes are marked



Fibonacci Heap Marks

History of a node v:

v is being linked to a node ⇨ $\boldsymbol{v.mark \coloneqq \mathbf{false}}$

a child of v is cut ⇨ $\boldsymbol{v.mark \coloneqq \mathbf{true}}$

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: $O(\text{length of } H.rootlist)$

- 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	00001	1
2	00010	2
3	00011	1
4	00100	3
5	00101	1
6	00110	2
7	00111	1
8	01000	4
9	01001	1
10	01010	2
11	01011	1
12	01100	3
13	01101	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 $\boldsymbol{\Phi}: \mathcal{S} \to \mathbb{R}_{\geq 0}$

- **Operation $\boldsymbol{i}$:**
  - $\boldsymbol{t_i}$: actual cost of operation i
  - $\boldsymbol{S_i}$: state after execution of operation i (S_0: initial state)
  - $\boldsymbol{\Phi_i} \coloneqq \Phi(S_i)$: potential after exec. of operation i
  - $\boldsymbol{a_i}$: amortized cost of operation i:

$$\boldsymbol{a_i \coloneqq t_i + \Phi_i - \Phi_{i-1}}$$

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

- **Delete-min** has **amortized cost $\boldsymbol{O(\log n)}$**
- **Decrease-key** has **amortized cost $\boldsymbol{O(1)}$**

- Starting with an empty heap, any sequence of n operations with at most n_d delete-min operations has total cost (time)

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

- Cost for Dijkstra: $O(|E| + |V| \log |V|)$