Datastrukturer til disjunkte mængder

Diskret Matematik og Algoritmer

Københavns Universitet, efterår 2021

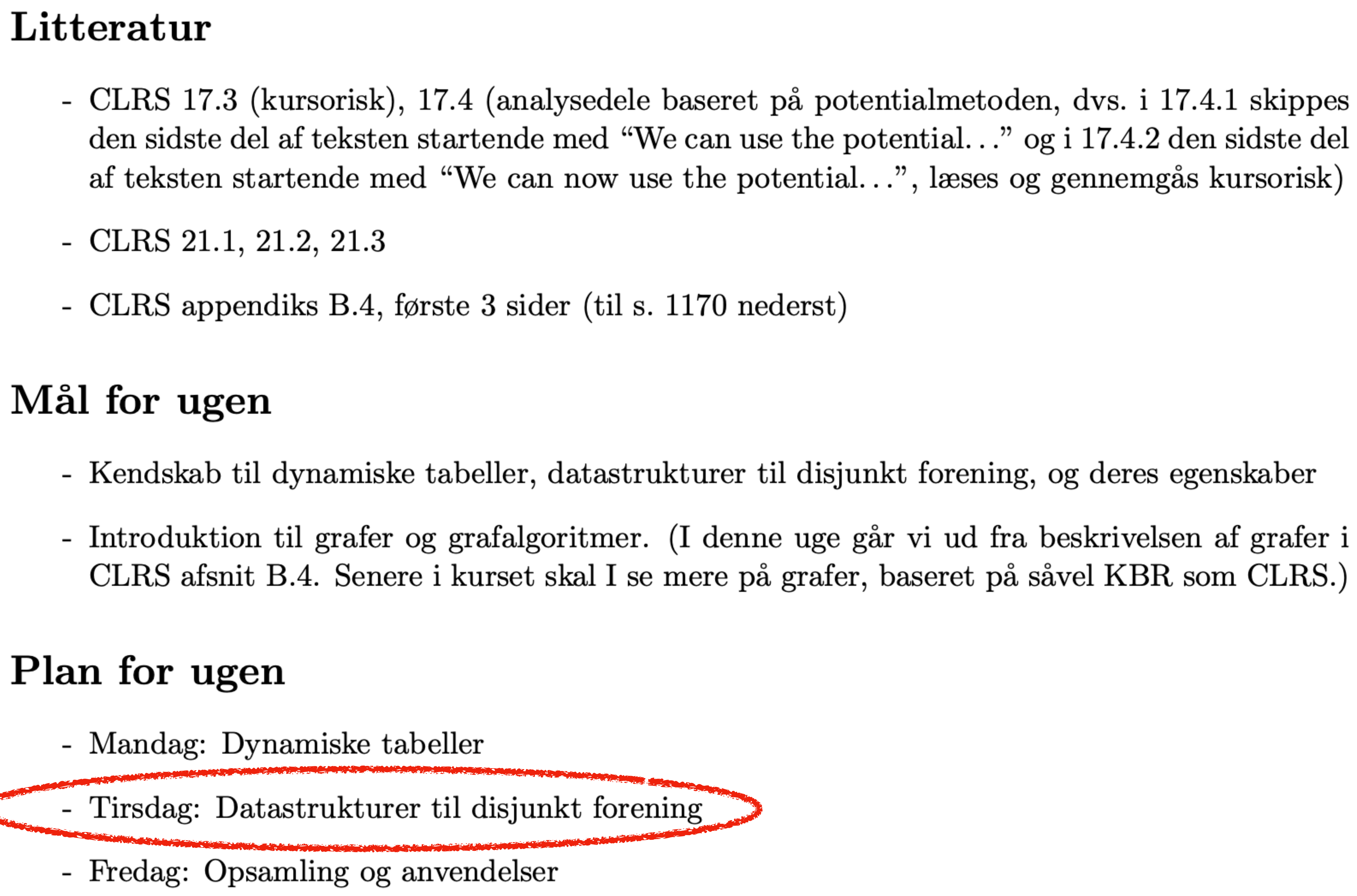
Rasmus Pagh

Slides med lyseblå baggrund er baseret på slides af Kevin Wayne

Opsummering fra mandag

- Hvis en streng skal opdateres mange gange fordi den “bygges” fra venstre til højre, er det effektivt at bruge *dynamiske strenge* (StringBuilder i F#)
- Tilsvarende kan man lave *dynamiske tabeller* med vilkårlige datatyper:
  - Understøtter `Table-Insert`, `Table-Delete` i amortiseret $O(1)$ tid
  - Kan returnere indgang nummer i i $O(1)$ tid

Hvad sker der?



Litteratur

- CLRS 17.3 (kursorisk), 17.4 (analysedele baseret på potentialmetoden, dvs. i 17.4.1 skippes den sidste del af teksten startende med "We can use the potential..." og i 17.4.2 den sidste del af teksten startende med "We can now use the potential...", læses og gennemgås kursorisk)
- CLRS 21.1, 21.2, 21.3
- CLRS appendiks B.4, første 3 sider (til s. 1170 nederst)

Mål for ugen

- Kendskab til dynamiske tabeller, datastrukturer til disjunkt forening, og deres egenskaber
- Introduktion til grafer og grafalgoritmer. (I denne uge går vi ud fra beskrivelsen af grafer i CLRS afsnit B.4. Senere i kurset skal I se mere på grafer, baseret på såvel KBR som CLRS.)

Plan for ugen

- Mandag: Dynamiske tabeller
- Tirsdag: Datastrukturer til disjunkt forening
- Fredag: Opsamling og anvendelser

Motiverende case

- Data: En liste af n Facebook venskaber

  ```
  [(Rasmus,Anna),(Rasmus,Thore),(Thore,Tony),(Anna,Thore),…]
  ```

- Spørgsmål: Hvem er forbundet med hvem, direkte eller indirekte?

Datastruktur til disjunkte mængder

Mål. Vi vil understøtte tre operationer på en samling af disjunkte mængder:

- MAKE-SET(x): skab en ny mængde, der kun indeholder elementet x.
- FIND-SET(x): returner et "kanonisk" element i mængden, der indeholder x.
- UNION(x, y): udskift mængderne der indeholder x og y med deres foreningsmængde.

Parametre:

- m = antal kald til MAKE-SET, FIND-SET, og UNION.
- n = antal elementer = antal kald til MAKE-SET.

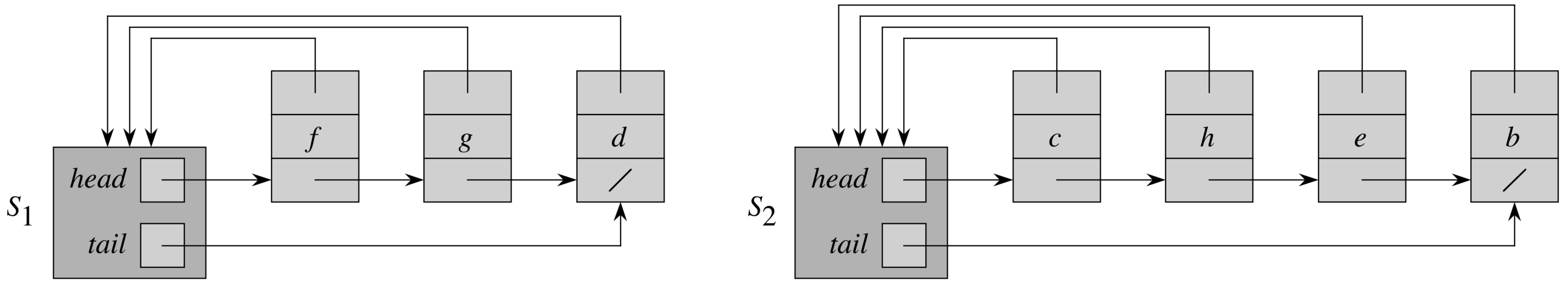
Eksempel

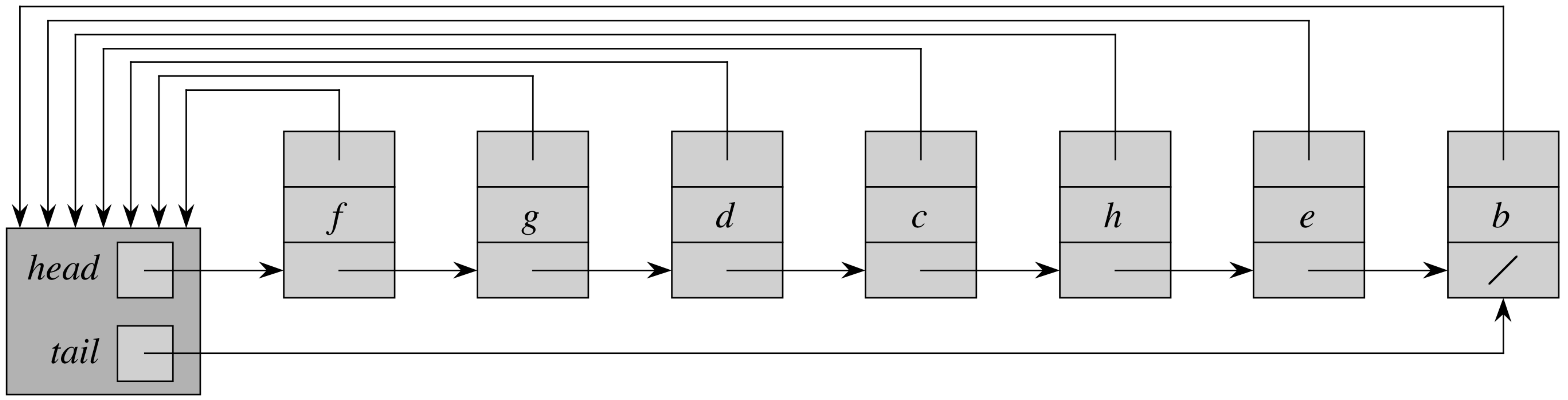
Edge processed	Collection of disjoint sets									
initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b,d)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e,g)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(a,c)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(h,i)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(a,b)	$\{a,b,c,d\}$				$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(e,f)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$
(b,c)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$

Øvelse

- Mængderne er identificeret med strenge, fx “Rasmus”, “Anna”, “Thore”,…
- Din datastruktur til disjunkte mængder understøtter kun at mængderne identificeres med tallene $0,\ldots,n-1$
- Hvordan kan vi bruge datastrukturen til at løse den motiverende case?

Hægtet liste repræsentation (CLRS 21.2)



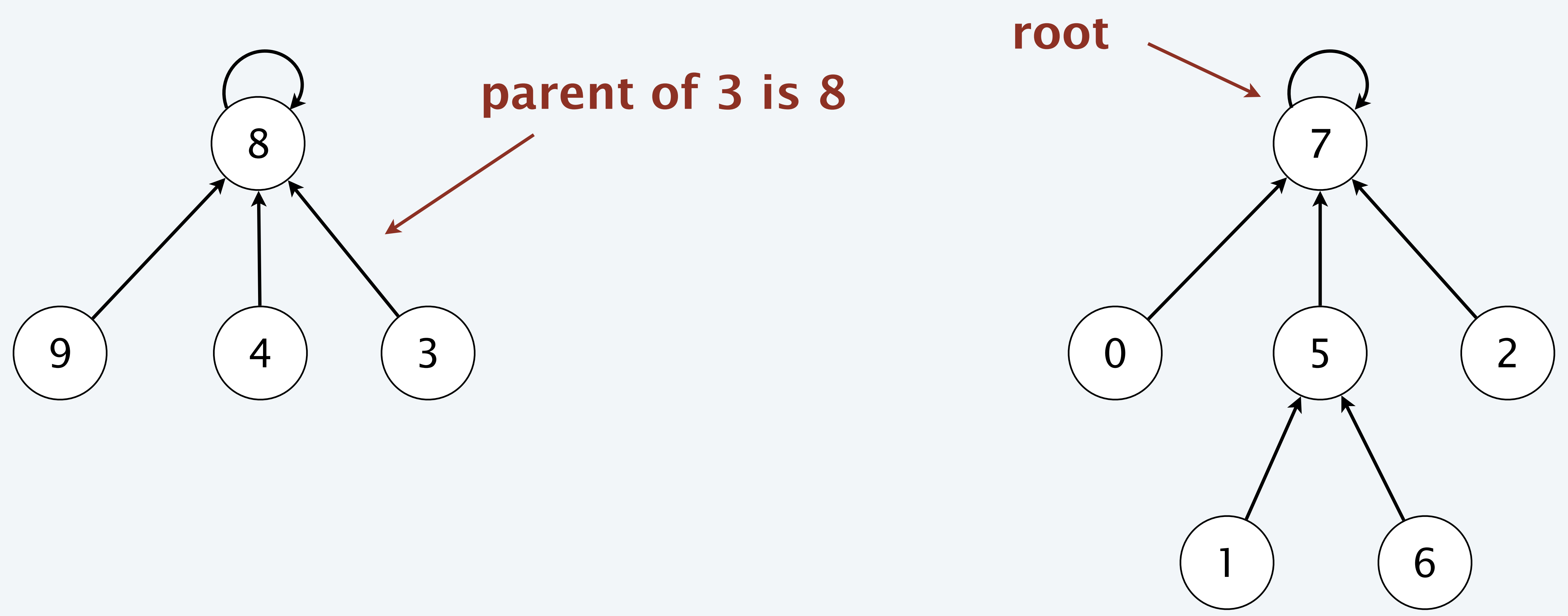


Disjoint-sets data structure

Parent-link representation. Represent each set as a tree of elements.

- Each element has an explicit parent pointer in the tree.
- The root serves as the canonical element (and points to itself).
- FIND-SET(x): find the root of the tree containing x.
- UNION(x, y): merge trees containing x and y (by making one root point to the other root).

UNION(3, 5)

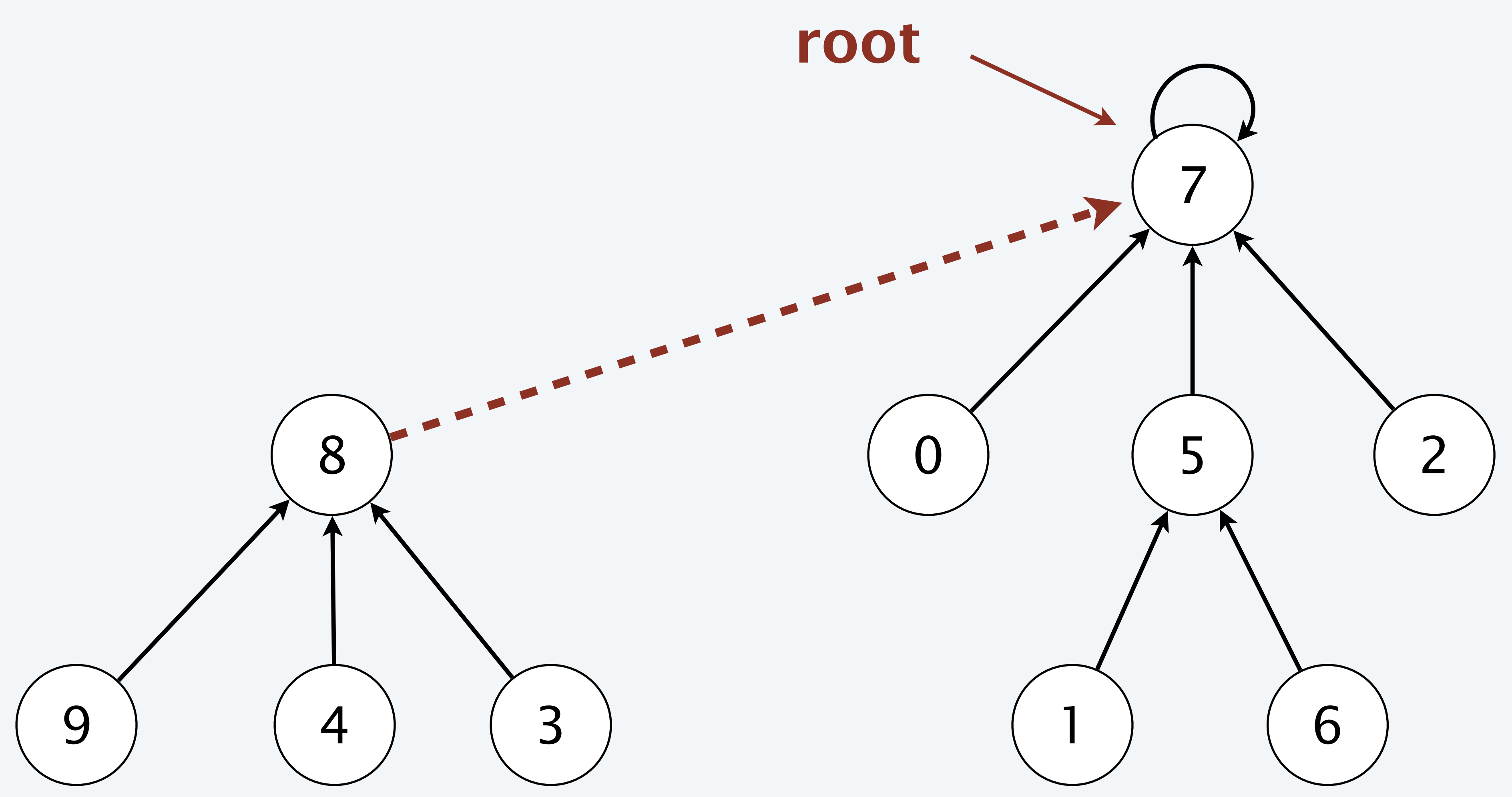


Disjoint-sets data structure

Parent-link representation. Represent each set as a tree of elements.

- Each element has an explicit parent pointer in the tree.
- The root serves as the canonical element (and points to itself).
- FIND-SET(x): find the root of the tree containing x.
- UNION(x, y): merge trees containing x and y
  (by making one root point to the other root).

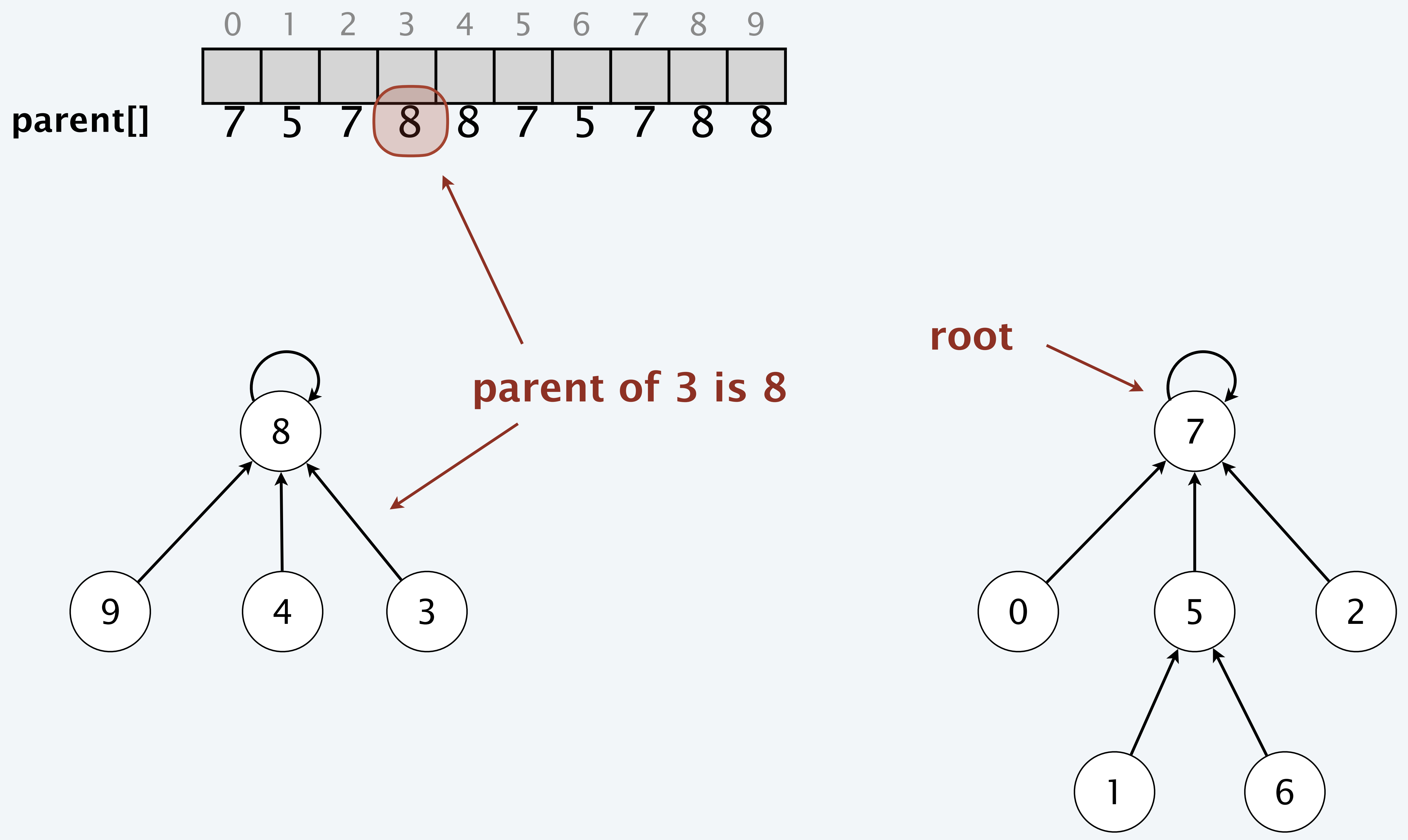
UNION(3, 5)



Disjoint-sets data structure

Array representation. Represent each set as a tree of elements.

- Allocate an array *parent*[] of length n. ← must know number of elements n a priori
- *parent*[i] = j means parent of element i is element j.

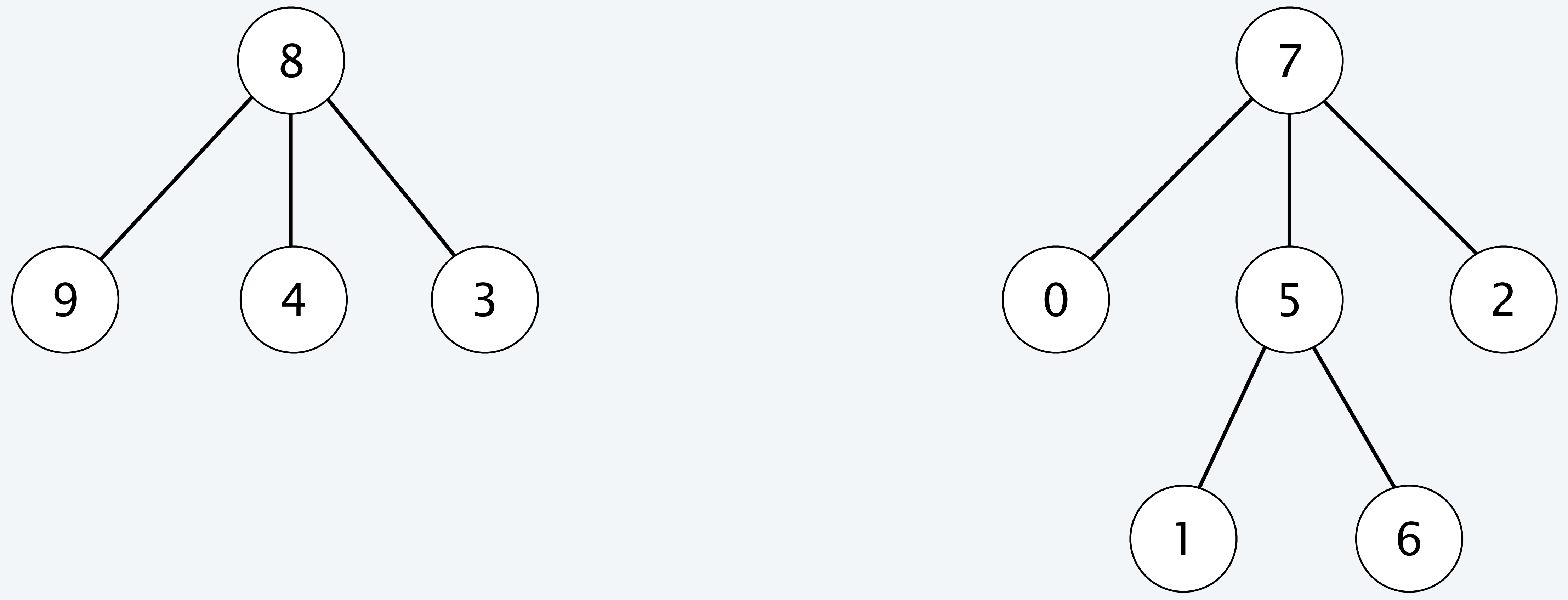


Note. For brevity, we suppress arrows and self-loops in figures.

Naïve linking

Naïve linking. Link root of first tree to root of second tree.

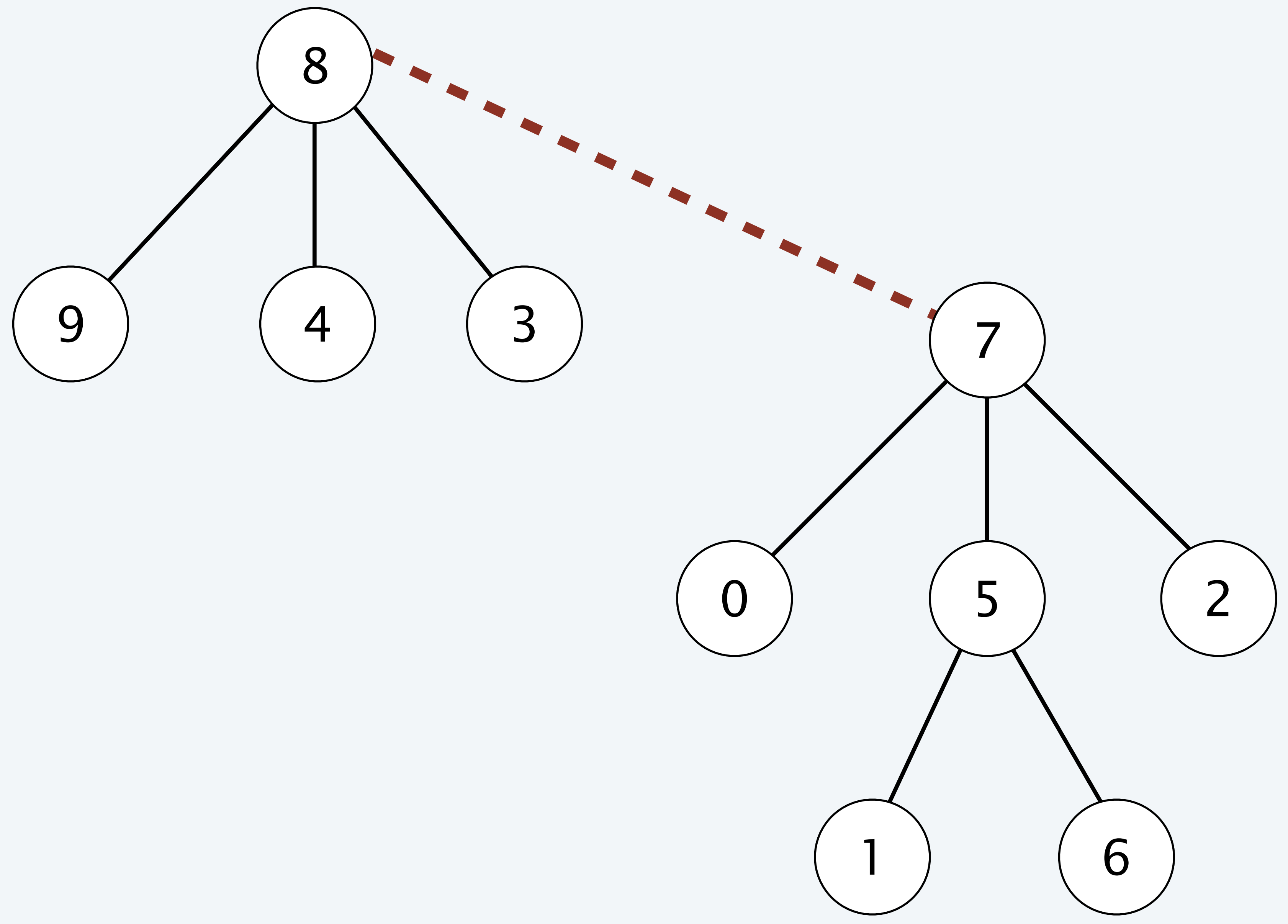
UNION(5, 3)



Naïve linking

Naïve linking. Link root of first tree to root of second tree.

UNION(5, 3)



Naïve linking

Naïve linking. Link root of first tree to root of second tree.

```
MAKE-SET($x$)
$parent[x] \leftarrow x$.
```

```
UNION($x$, $y$)
$r \leftarrow$ FIND-SET($x$).
$s \leftarrow$ FIND-SET($y$).
$parent[r] \leftarrow s$.
```

```
FIND-SET($x$)
WHILE ($x \neq parent[x]$)
   $x \leftarrow parent[x]$.
RETURN $x$.
```

Øvelse

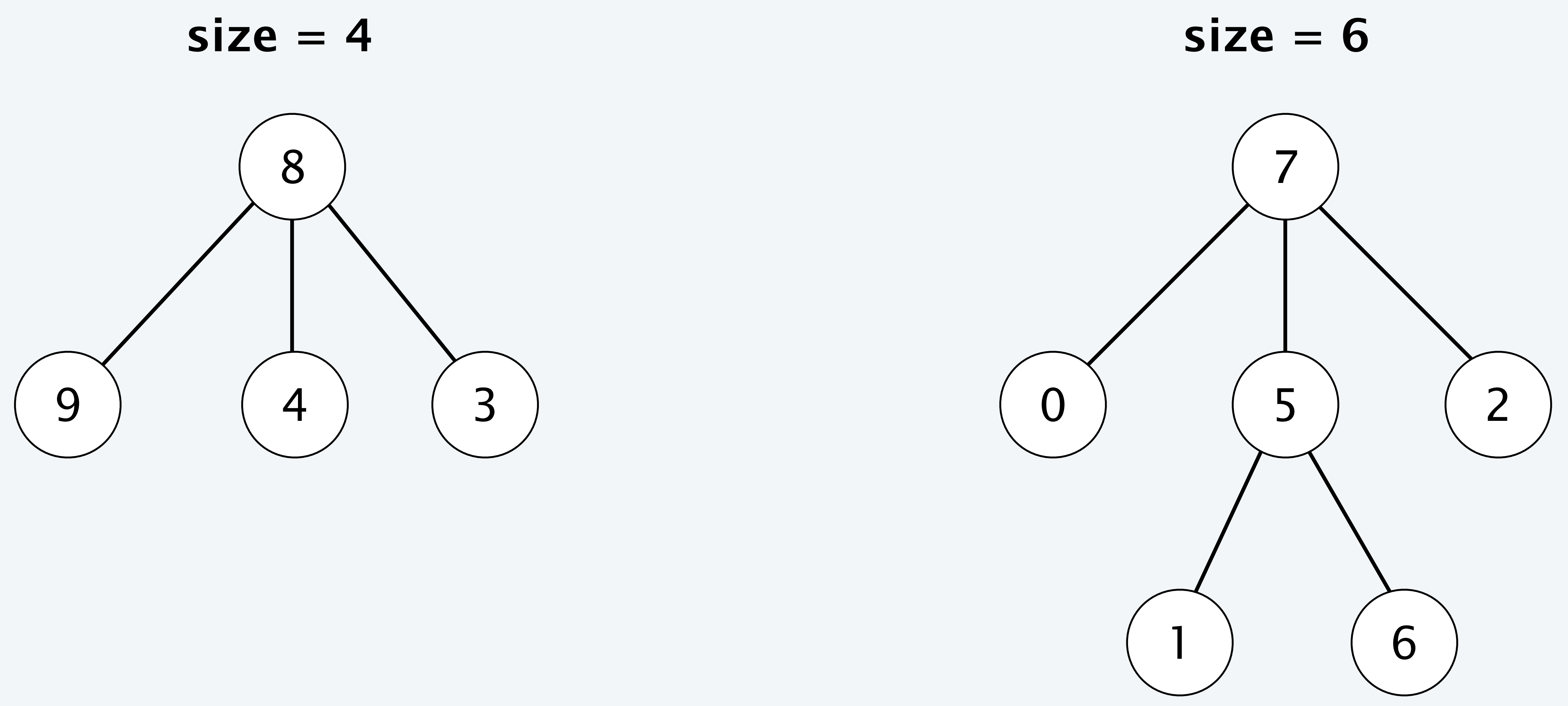
- Hvad er omkostningen af denne sekvens af operationer med naiv hægtning?

Operation	Number of objects updated
MAKE-SET(x_1)	1
MAKE-SET(x_2)	1
$\vdots$	$\vdots$
MAKE-SET(x_n)	1
UNION(x_2, x_1)	1
UNION(x_3, x_2)	2
UNION(x_4, x_3)	3
$\vdots$	$\vdots$
UNION(x_n, x_{n-1})	$n - 1$

Link-by-size (CLRS: "Weighted union heuristic")

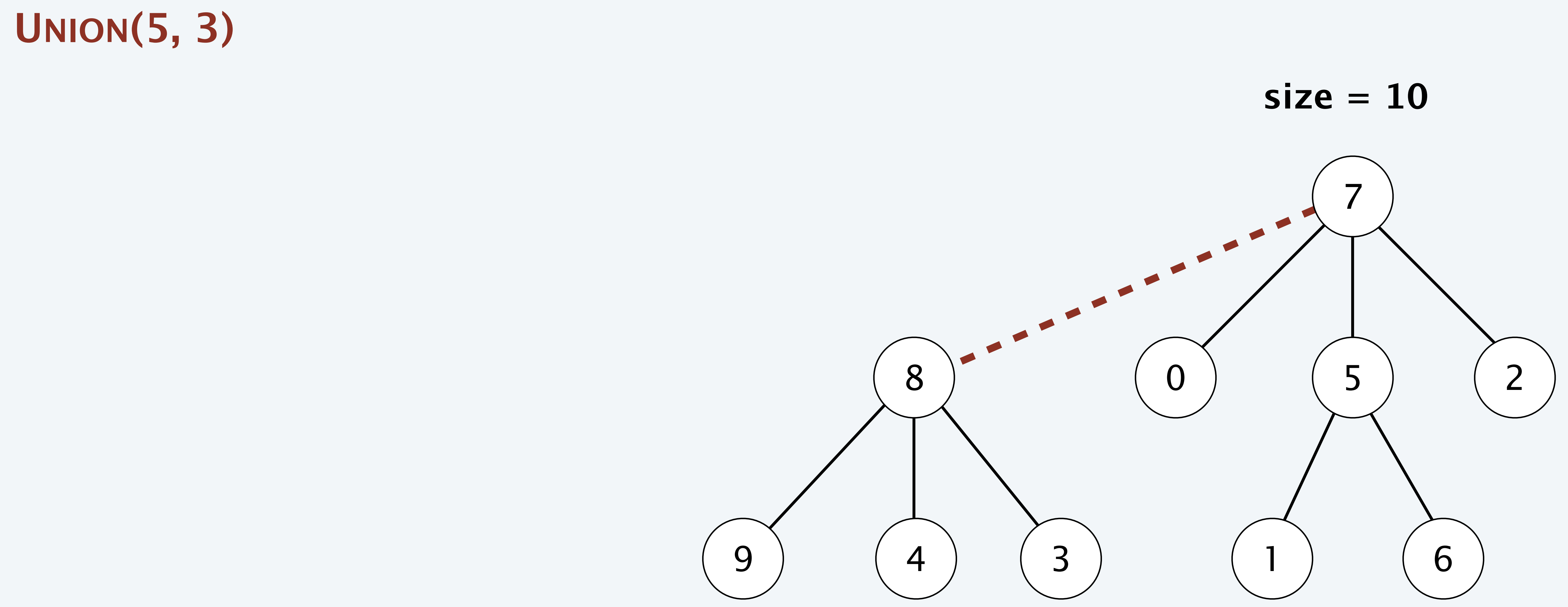
Link-by-size. Maintain a tree size (number of nodes) for each root node. Link root of smaller tree to root of larger tree (breaking ties arbitrarily).

UNION(5, 3)



Link-by-size

Link-by-size. Maintain a tree size (number of nodes) for each root node. Link root of smaller tree to root of larger tree (breaking ties arbitrarily).



Link-by-size

Link-by-size. Maintain a tree size (number of nodes) for each root node.
Link root of smaller tree to root of larger tree (breaking ties arbitrarily).

```
MAKE-SET($x$)
$parent[x] \leftarrow x$.
$size[x] \leftarrow 1$.
```

```
FIND-SET($x$)
WHILE ($x \neq parent[x]$)
    $x \leftarrow parent[x]$.
RETURN $x$.
```

```
UNION($x$, $y$)
$r \leftarrow$ FIND-SET($x$).
$s \leftarrow$ FIND-SET($y$).
IF ($r = s$) RETURN.
ELSE IF ($size[r] > size[s]$)
    $parent[s] \leftarrow r$.
    $size[r] \leftarrow size[r] + size[s]$.
ELSE
    $parent[r] \leftarrow s$.
    $size[s] \leftarrow size[r] + size[s]$.
```

link-by-size

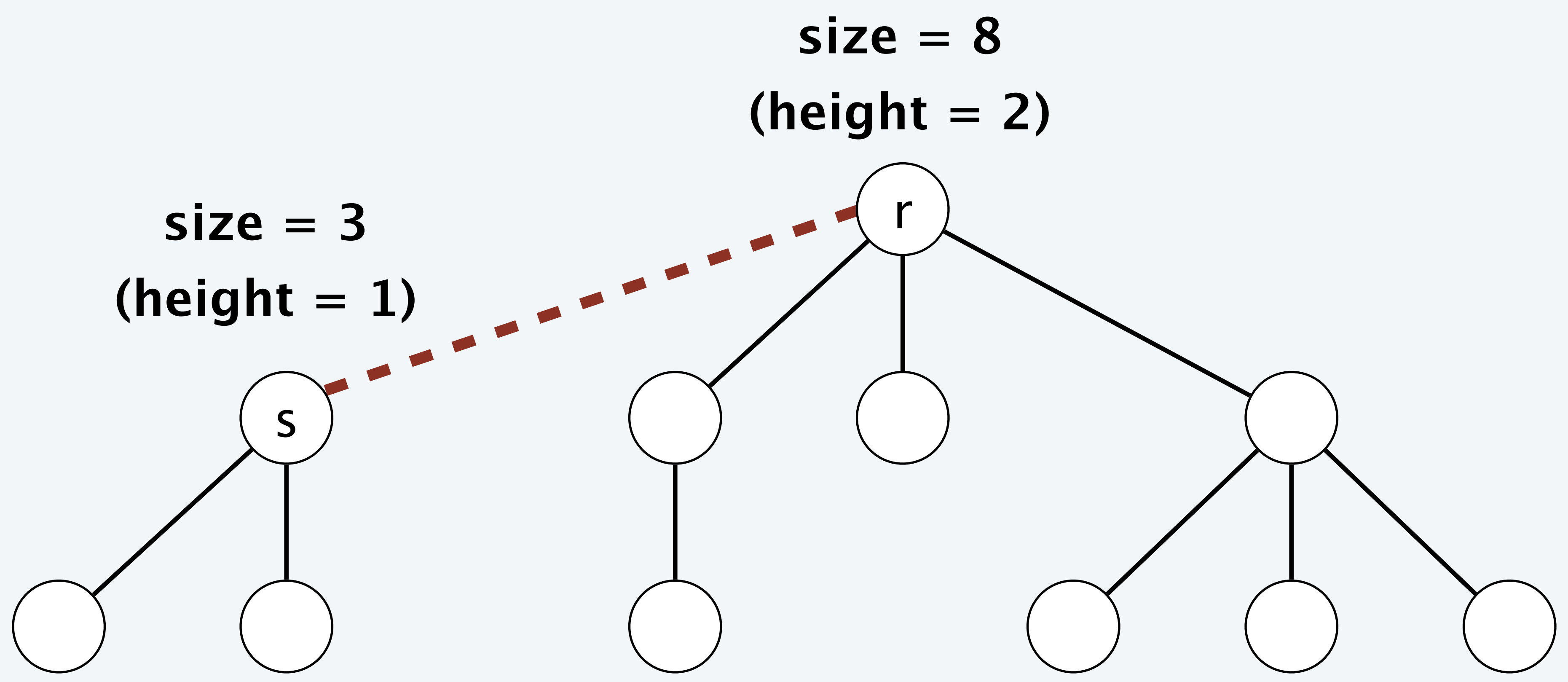
Link-by-size: analysis

Property. Using link-by-size, for every root node r : $size[r] \ge 2^{height(r)}$.

Pf. [by induction on number of links]

- Base case: singleton tree has size 1 and height 0.
- Inductive hypothesis: assume true after first i links.
- Tree rooted at r changes only when a smaller (or equal) size tree rooted at s is linked into r.
- Case 1. [$height(r) > height(s)$]

$$\begin{aligned} size'[r] &> size[r] \\ &\ge 2^{height(r)} \quad \longleftarrow \text{inductive hypothesis} \\ &= 2^{height'(r)}. \end{aligned}$$



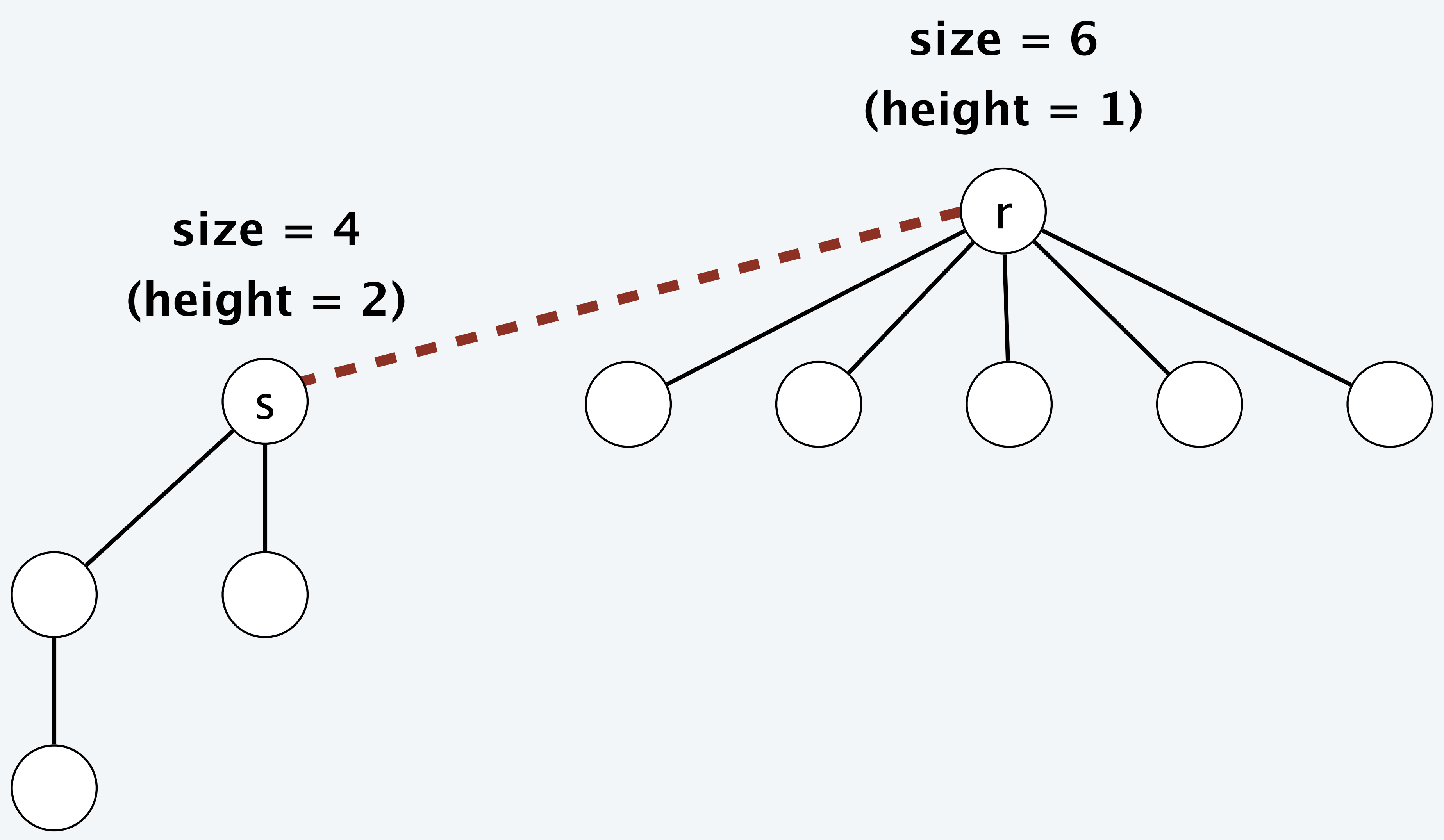
Link-by-size: analysis

Property. Using link-by-size, for every root node r : $size[r] \geq 2^{height(r)}$.

Pf. [by induction on number of links]

- Base case: singleton tree has size 1 and height 0.
- Inductive hypothesis: assume true after first i links.
- Tree rooted at r changes only when a smaller (or equal) size tree rooted at s is linked into r.
- Case 2. [$height(r) \leq height(s)$]

$$\begin{aligned} size'[r] &= size[r] + size[s] \\ &\geq 2\, size[s] && \longleftarrow \text{link-by-size} \\ &\geq 2 \cdot 2^{height(s)} && \longleftarrow \text{inductive hypothesis} \\ &= 2^{height(s)+1} \\ &= 2^{height'(r)}. \quad \blacksquare \end{aligned}$$



Link-by-size: analysis

Theorem. Using link-by-size, any UNION or FIND-SET operation takes $O(\log n)$ time in the worst case, where n is the number of elements.

Pf.

- The running time of each operation is bounded by the tree height.
- By the previous property, the height is $\le \lfloor \lg n \rfloor$. ▪

$\uparrow$

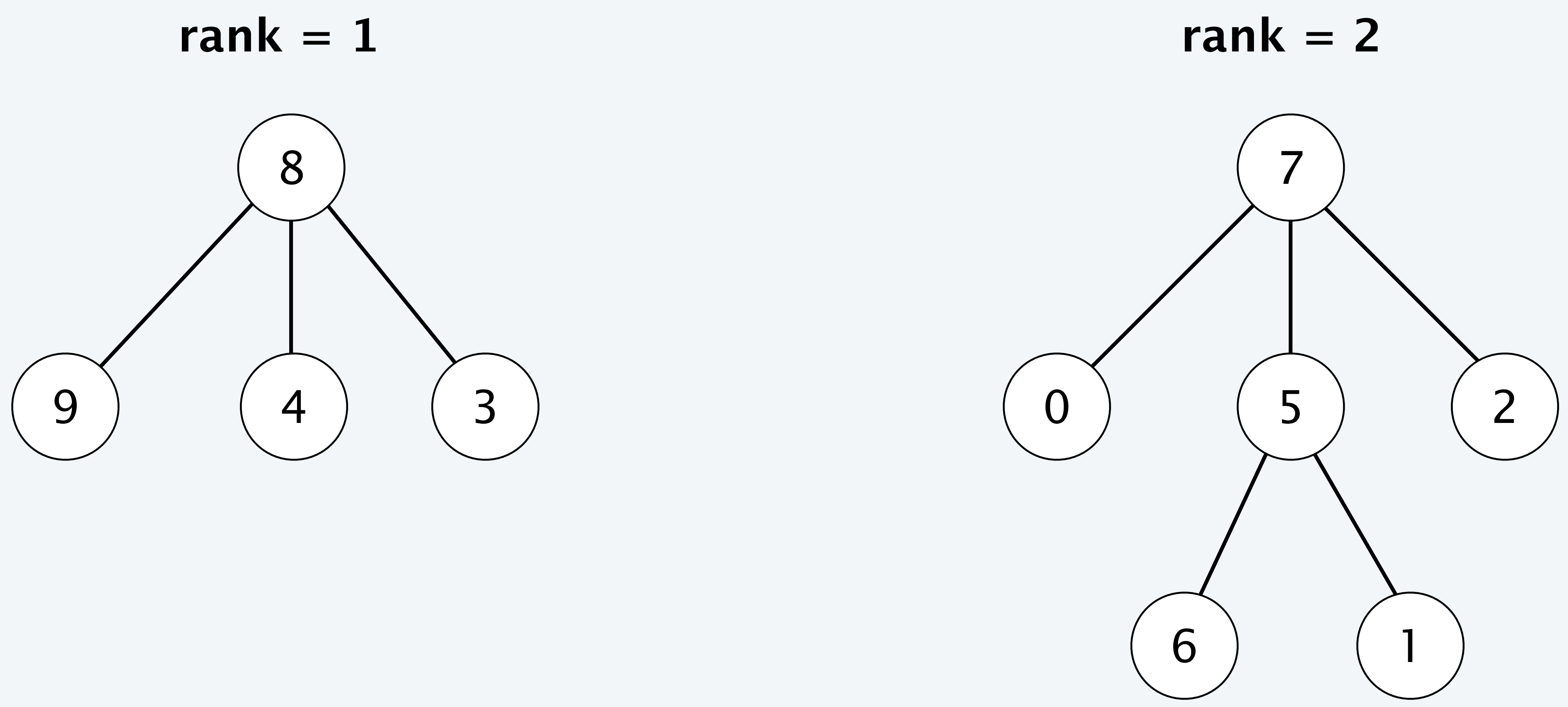
$\lg n = \log_2 n$

Note. The UNION operation takes $O(1)$ time except for its two calls to FIND-SET.

Link-by-rank

Link-by-rank. Maintain an integer rank for each node, initially 0. Link root of smaller rank to root of larger rank; if tie, increase rank of larger root by 1.

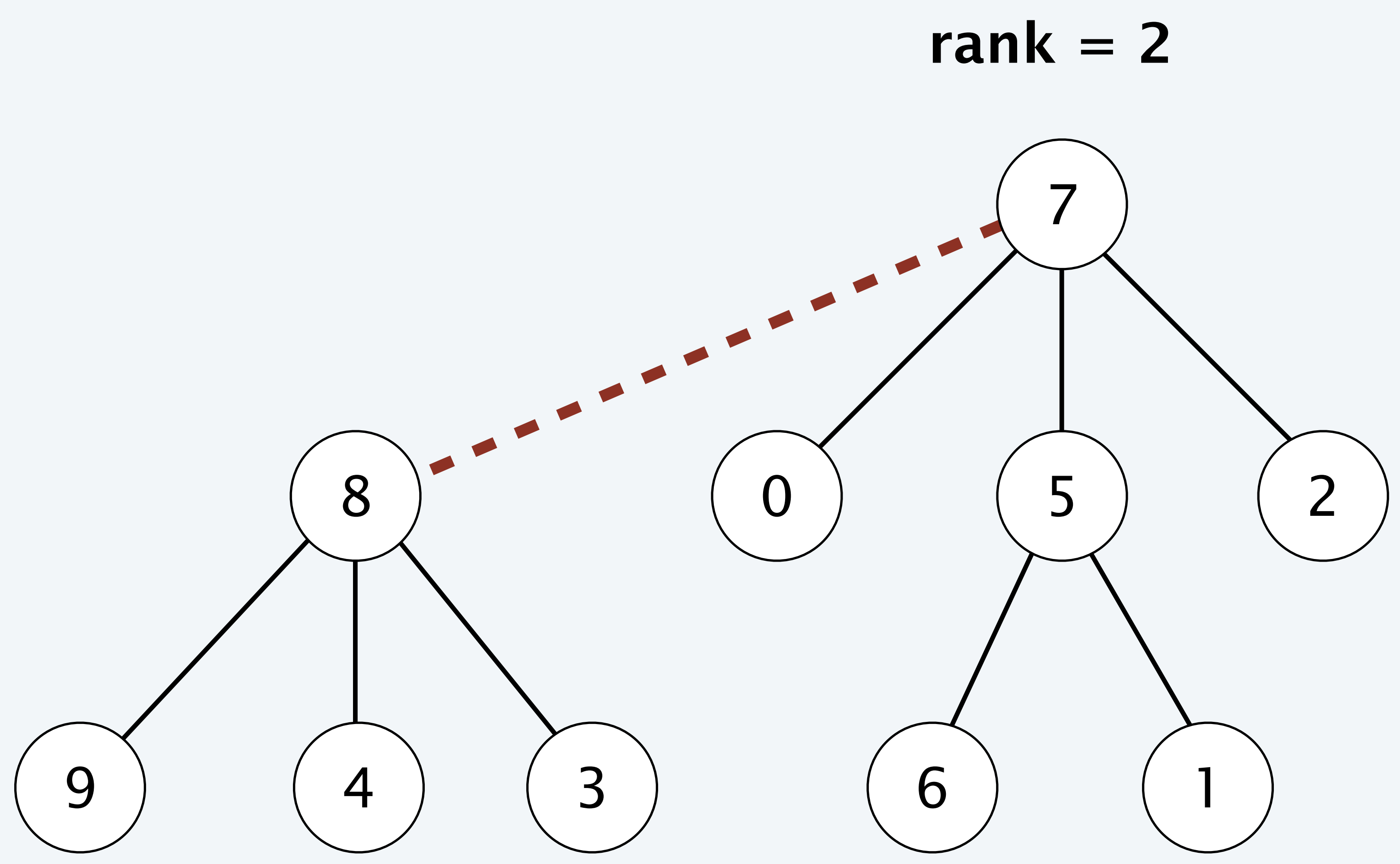
UNION(5, 3)



Note. For now, rank = height.

Link-by-rank

Link-by-rank. Maintain an integer rank for each node, initially 0. Link root of smaller rank to root of larger rank; if tie, increase rank of larger root by 1.



Note. For now, rank = height.

Link-by-rank

Link-by-rank. Maintain an integer rank for each node, initially 0. Link root of smaller rank to root of larger rank; if tie, increase rank of larger root by 1.

```
MAKE-SET($x$)
$parent[x] \leftarrow x$.
$rank[x] \leftarrow 0$.
```

```
FIND-SET($x$)
WHILE ($x \neq parent[x]$)
    $x \leftarrow parent[x]$.
RETURN $x$.
```

```
UNION($x, y$)
$r \leftarrow$ FIND-SET($x$).
$s \leftarrow$ FIND-SET($y$).
IF ($r = s$) RETURN.
ELSE IF ($rank[r] > rank[s]$)
    $parent[s] \leftarrow r$.
ELSE IF ($rank[r] < rank[s]$)
    $parent[r] \leftarrow s$.
ELSE
    $parent[r] \leftarrow s$.
    $rank[s] \leftarrow rank[s] + 1$.
```

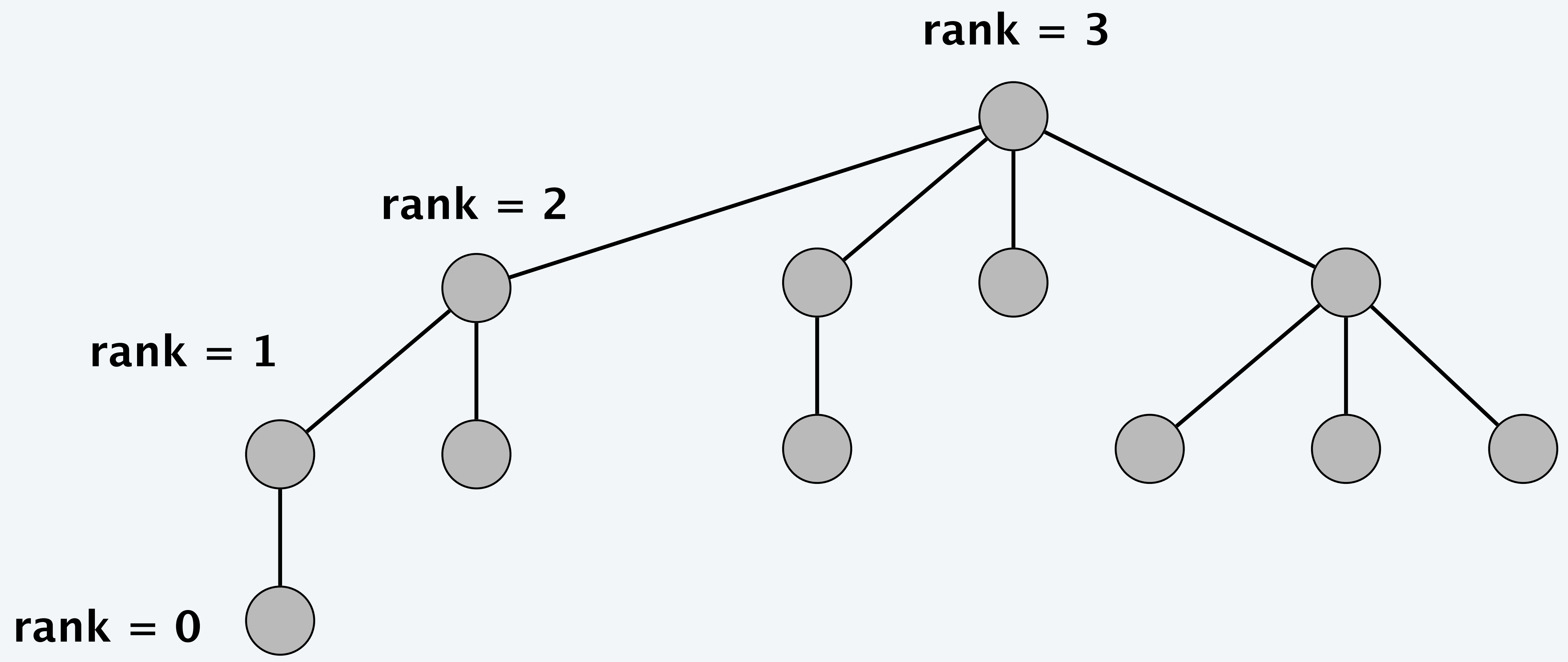
link-by-rank

Link-by-rank: properties

PROPERTY 1. If x is not a root node, then $rank[x] < rank[parent[x]]$.
Pf. A node of rank k is created only by linking two roots of rank $k-1$. ▪

PROPERTY 2. If x is not a root node, then $rank[x]$ will never change again.
Pf. Rank changes only for roots; a nonroot never becomes a root. ▪

PROPERTY 3. If $parent[x]$ changes, then $rank[parent[x]]$ strictly increases.
Pf. The parent can change only for a root, so before linking $parent[x] = x$. After x is linked-by-rank to new root r we have $rank[r] > rank[x]$. ▪



Link-by-rank: properties

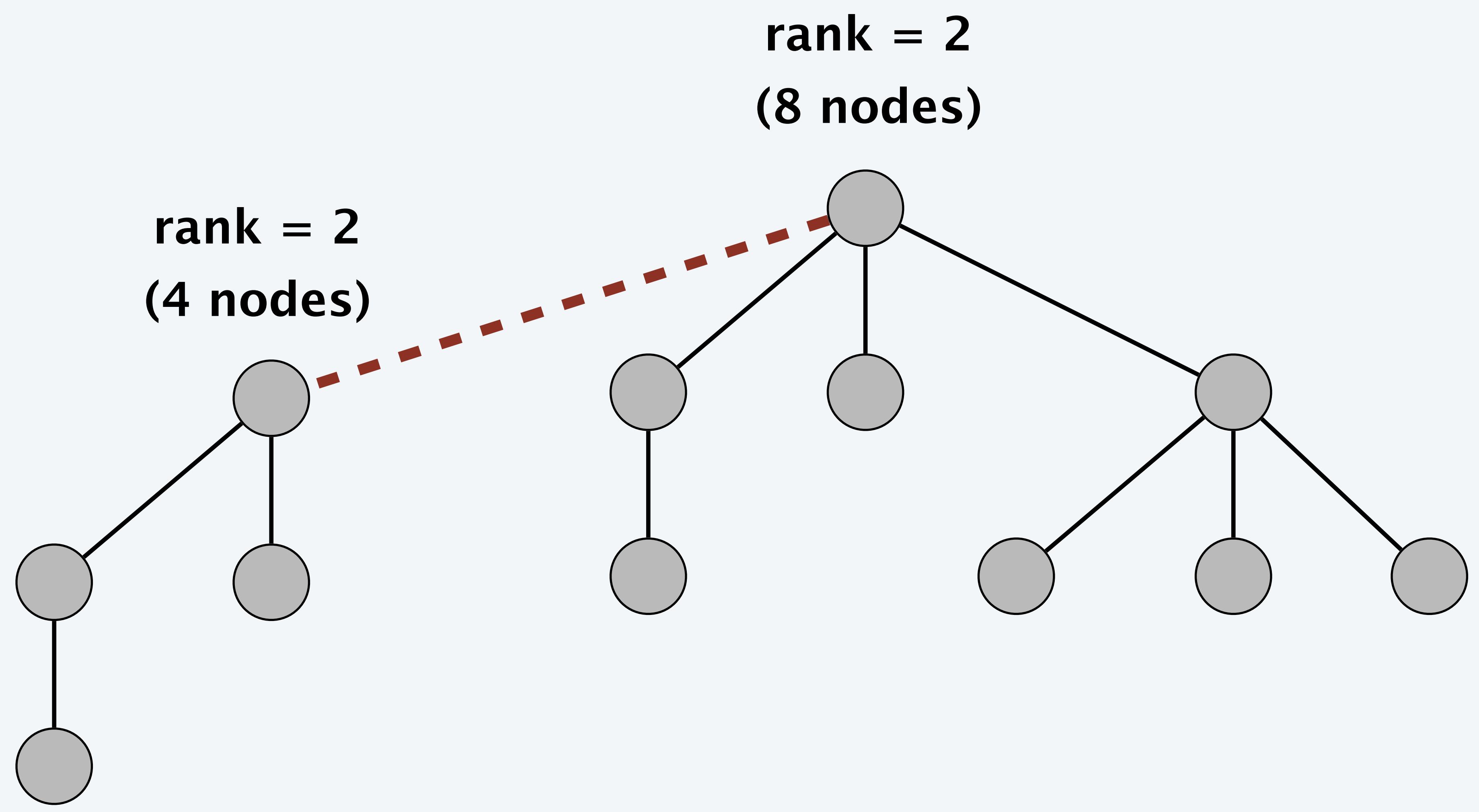
PROPERTY 4. Any root node of rank k has $\geq 2^k$ nodes in its tree.

Pf. [by induction on k]

- Base case: true for $k = 0$.
- Inductive hypothesis: assume true for $k - 1$.
- A node of rank k is created only by linking two roots of rank $k - 1$.
- By inductive hypothesis, each subtree has $\geq 2^{k-1}$ nodes
  $\Rightarrow$ resulting tree has $\geq 2^k$ nodes. ▪

PROPERTY 5. The highest rank of a node is $\leq \lfloor \lg n \rfloor$.

Pf. Immediate from PROPERTY 1 and PROPERTY 4. ▪



Link-by-rank: analysis

Theorem. Using link-by-rank, any UNION or FIND-SET operation takes $O(\log n)$ time in the worst case, where n is the number of elements.

Pf.

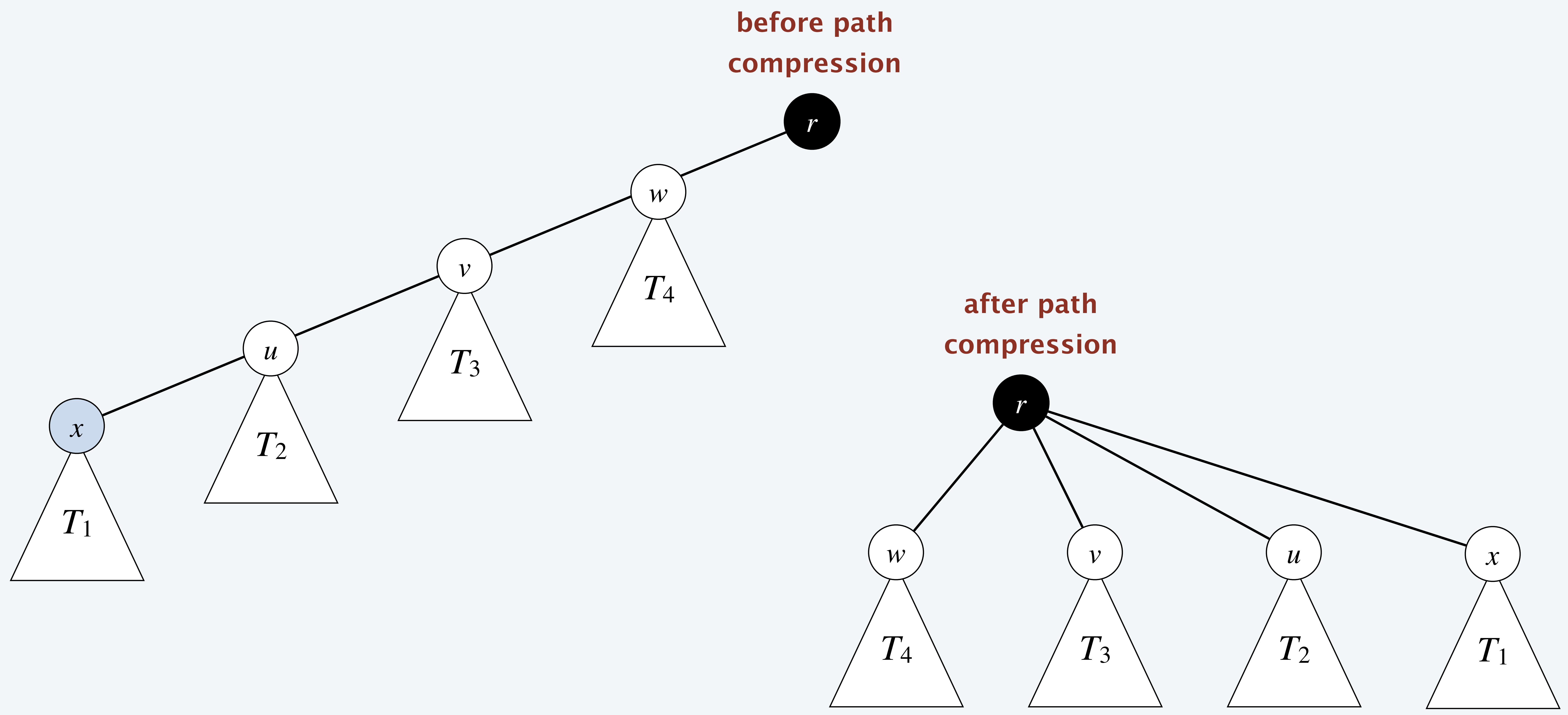
- The running time of UNION and FIND-SET is bounded by the tree height.
- By PROPERTY 5, the height is ≤ $\lfloor \lg n \rfloor$. ▪

Øvelse

- Antag at vi starter med $n = 2^i$ elementer, og vi altid laver `Union` på to mængder af samme størrelse
- Brug hægtning efter størrelse eller hægtning efter rang
- Hvad bliver højden af træet hvis vi fortsætter indtil alle elementer er i én mængde?

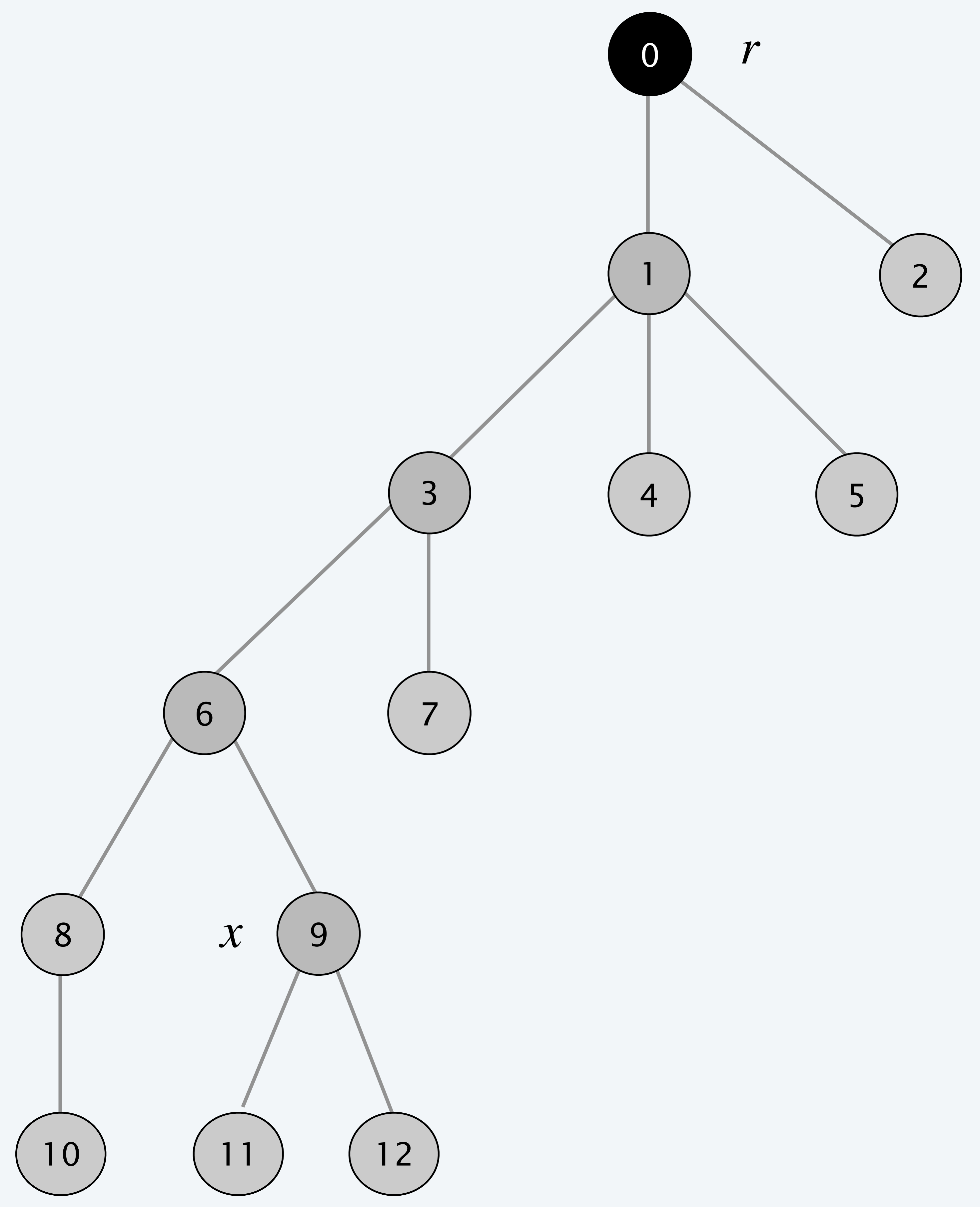
Path compression

Path compression. When finding the root r of the tree containing x, change the parent pointer of all nodes along the path to point directly to r.



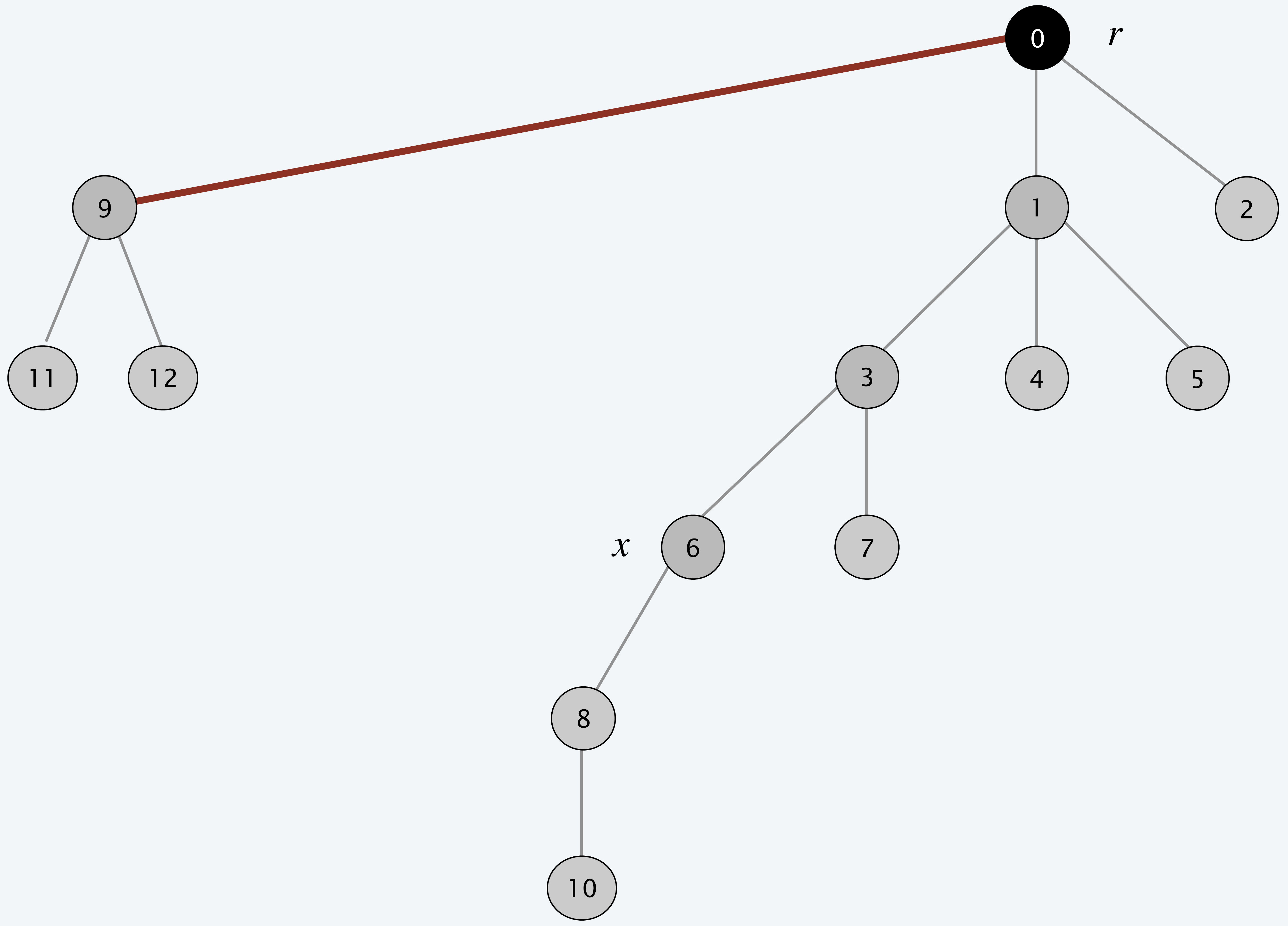
Path compression

Path compression. When finding the root r of the tree containing x, change the parent pointer of all nodes along the path to point directly to r.



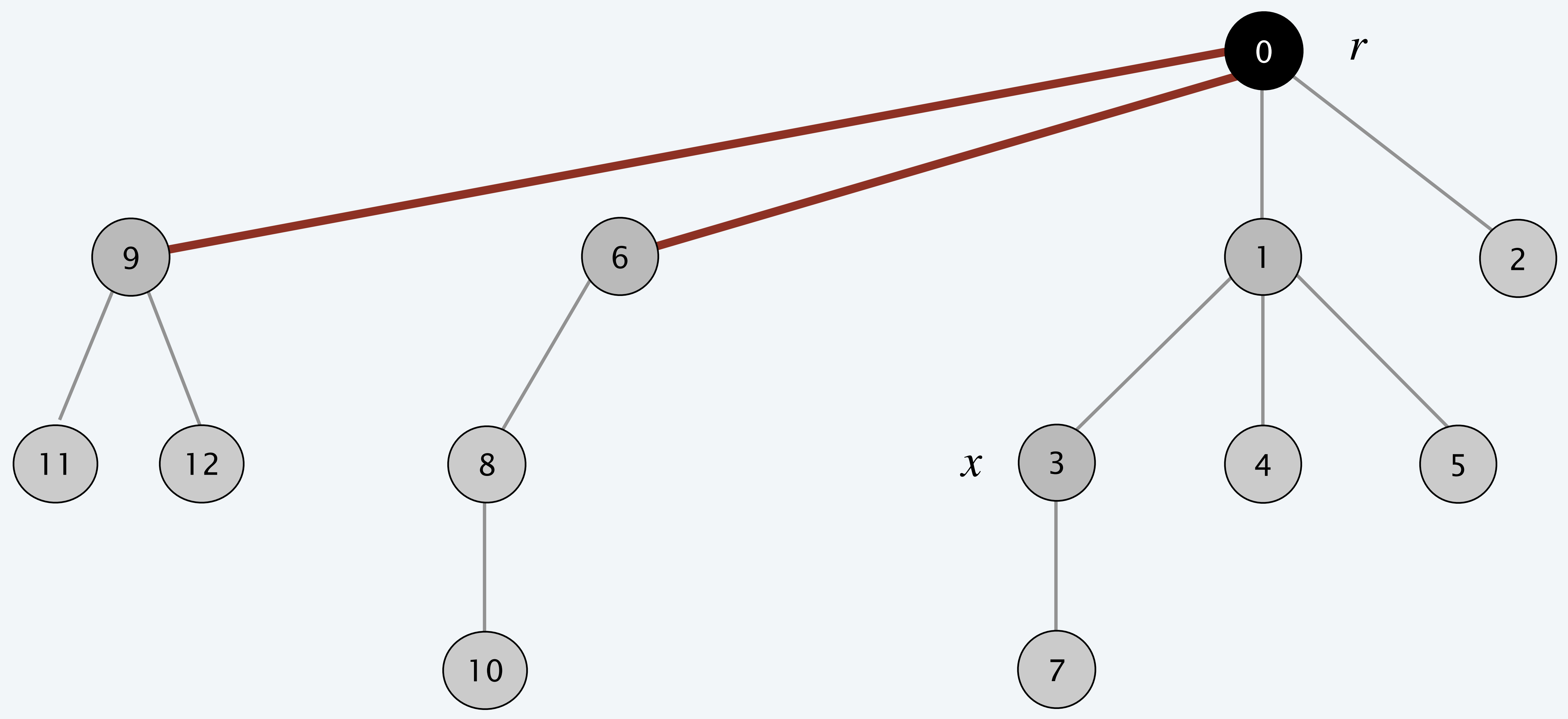
Path compression

Path compression. When finding the root r of the tree containing x, change the parent pointer of all nodes along the path to point directly to r.



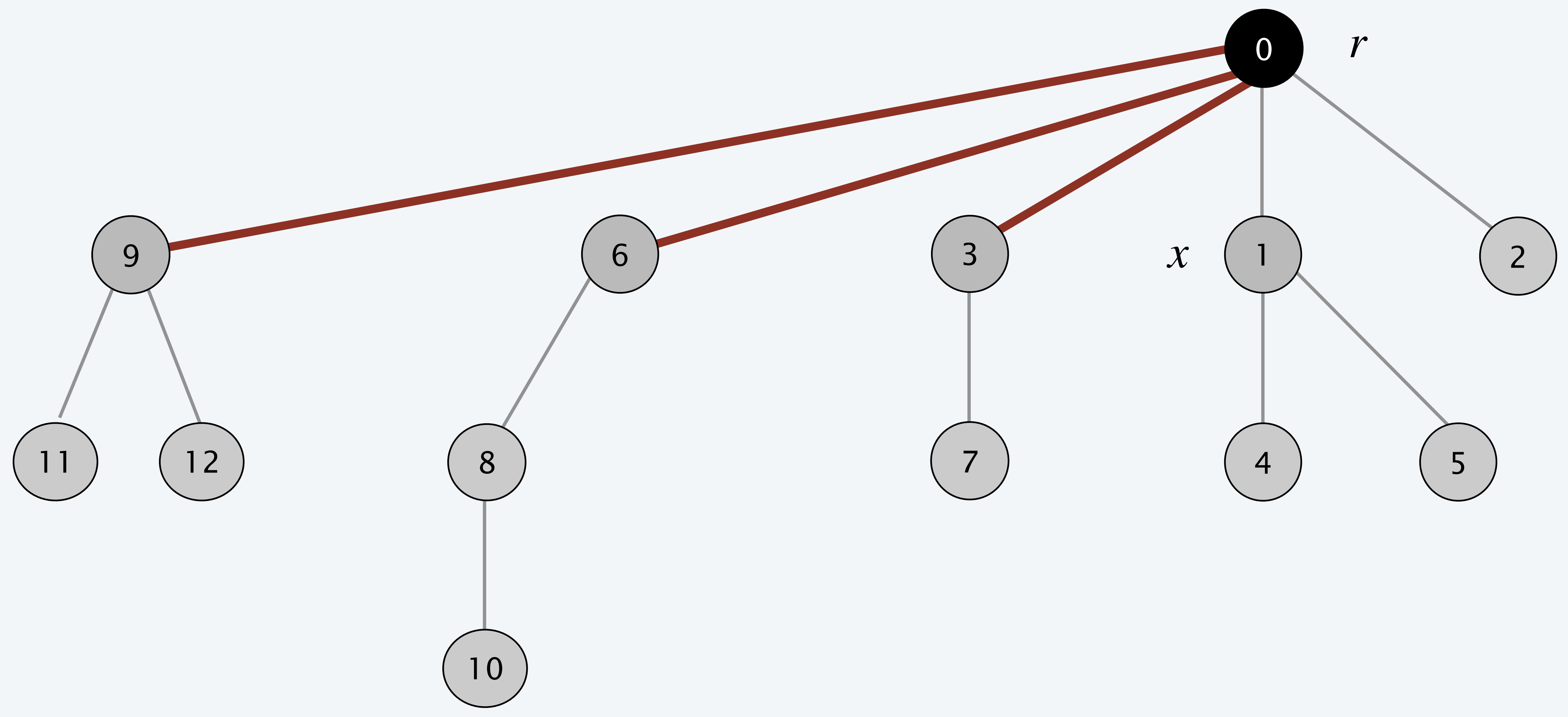
Path compression

Path compression. When finding the root r of the tree containing x, change the parent pointer of all nodes along the path to point directly to r.



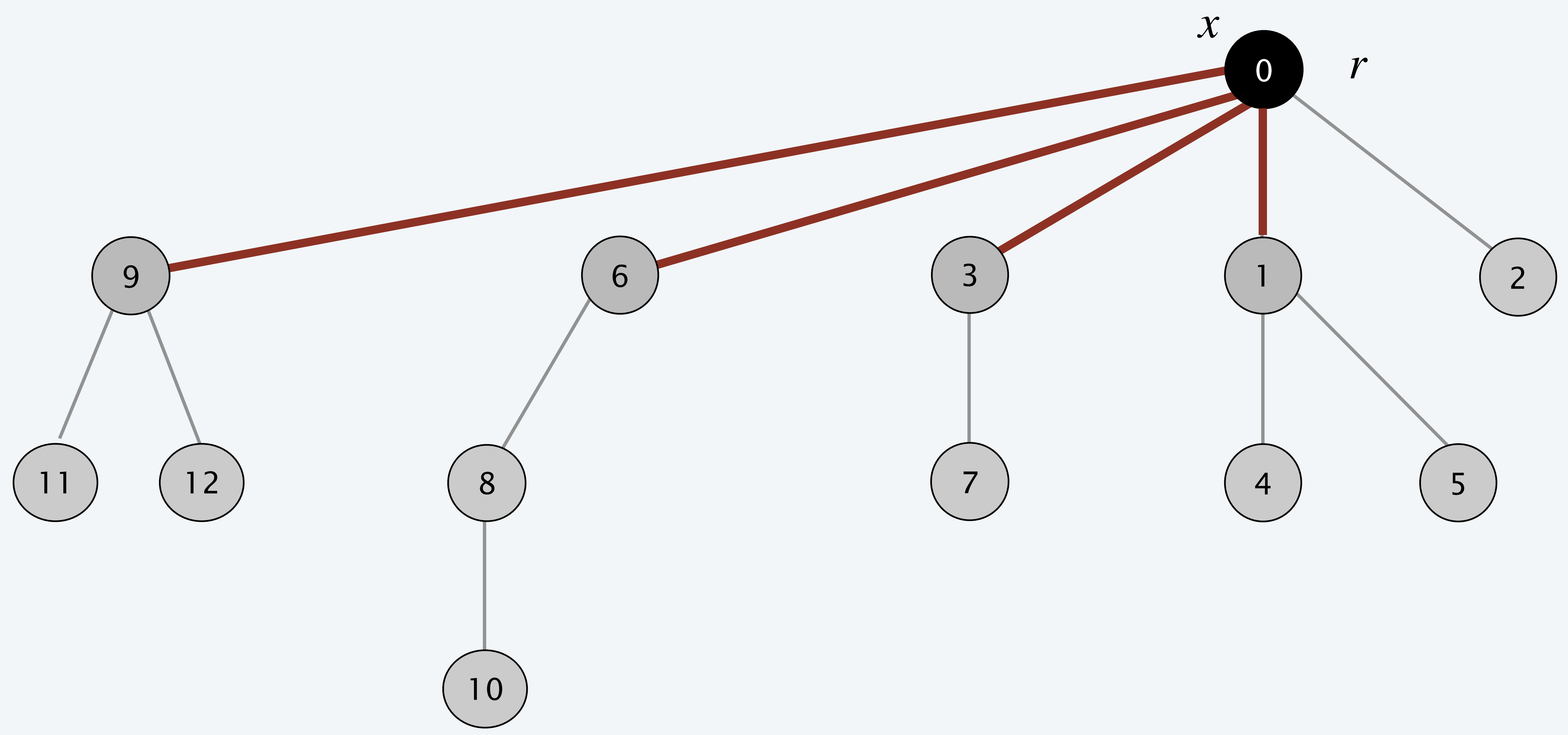
Path compression

Path compression. When finding the root r of the tree containing x, change the parent pointer of all nodes along the path to point directly to r.



Path compression

Path compression. When finding the root r of the tree containing x, change the parent pointer of all nodes along the path to point directly to r.



Path compression

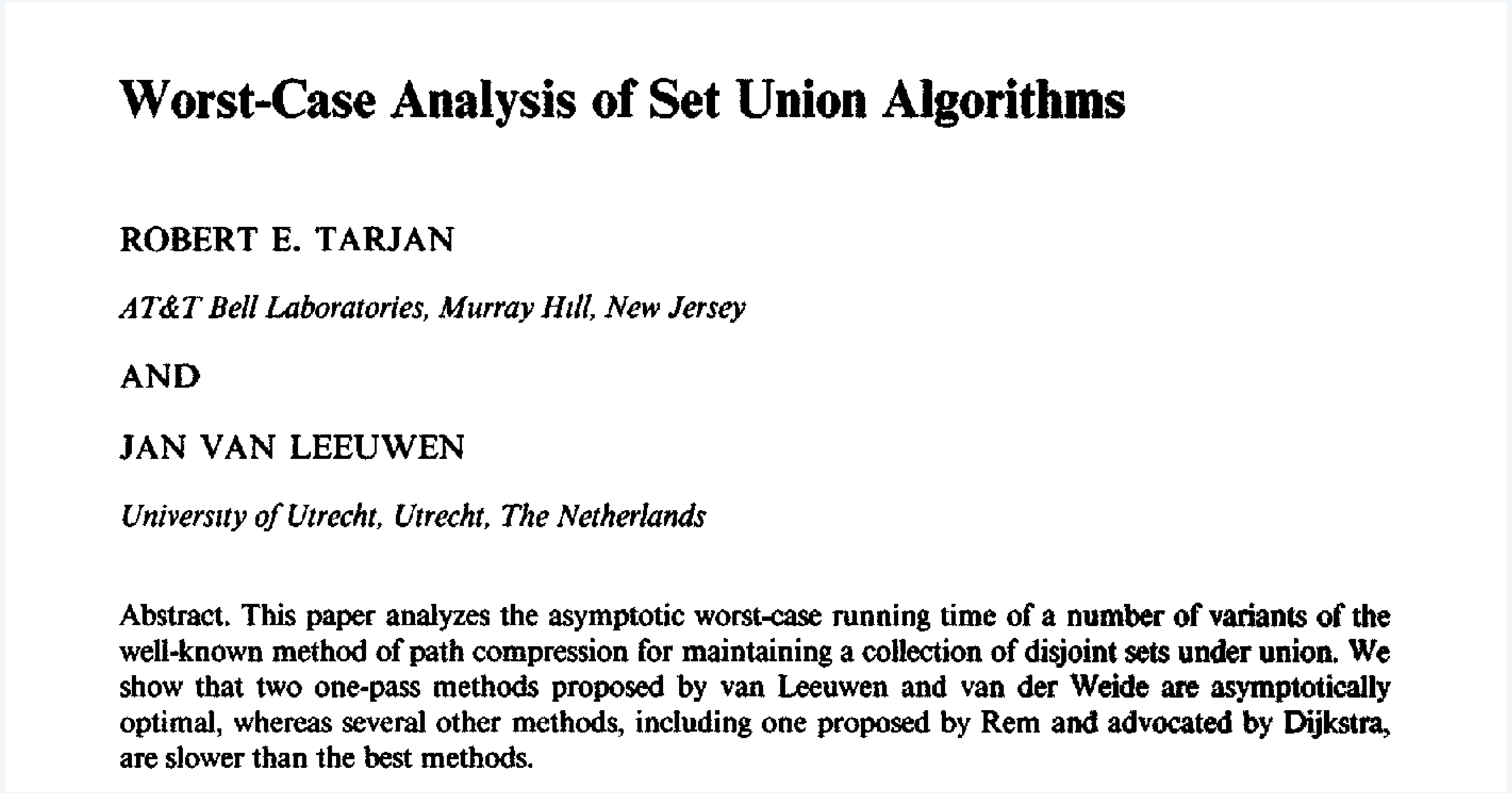
Path compression. When finding the root r of the tree containing x, change the parent pointer of all nodes along the path to point directly to r.

```
FIND-SET(x)
IF ($x \neq parent[x]$)
    $parent[x] \leftarrow$ FIND-SET($parent[x]$).
RETURN $parent[x]$.
```

this FIND-SET implementation changes the tree structure (!)

Analysis of path compression (CLRS 21.4)

Theorem. [Tarjan–van Leeuwen 1984] Starting from an empty data structure, link-by- { size, rank } combined with { path compression, path splitting, path halving } performs any intermixed sequence of $m \ge n$ MAKE-SET, UNION, and FIND operations on a set of n elements in $O(m\, \alpha(m, n))$ time.



Worst-Case Analysis of Set Union Algorithms

ROBERT E. TARJAN

AT&T Bell Laboratories, Murray Hill, New Jersey

AND

JAN VAN LEEUWEN

University of Utrecht, Utrecht, The Netherlands

Abstract. This paper analyzes the asymptotic worst-case running time of a number of variants of the well-known method of path compression for maintaining a collection of disjoint sets under union. We show that two one-pass methods proposed by van Leeuwen and van der Weide are asymptotically optimal, whereas several other methods, including one proposed by Rem and advocated by Dijkstra, are slower than the best methods.

Inverse Ackermann function

Definition.

$$\alpha_k(n) = \begin{cases} \lceil n/2 \rceil & \text{if } k = 1 \\ 0 & \text{if } n = 1 \text{ and } k \geq 2 \\ 1 + \alpha_k(\alpha_{k-1}(n)) & \text{otherwise} \end{cases}$$

Property. For every $n \geq 5$, the sequence $\alpha_1(n), \alpha_2(n), \alpha_3(n), \ldots$ converges to 3.

Ex. $[n = 9876!]$ $\alpha_1(n) \geq 10^{35163}$, $\alpha_2(n) = 116812$, $\alpha_3(n) = 6$, $\alpha_4(n) = 4$, $\alpha_5(n) = 3$.

One-parameter inverse Ackermann. $\alpha(n) = \min \{ k : \alpha_k(n) \leq 3 \}$.

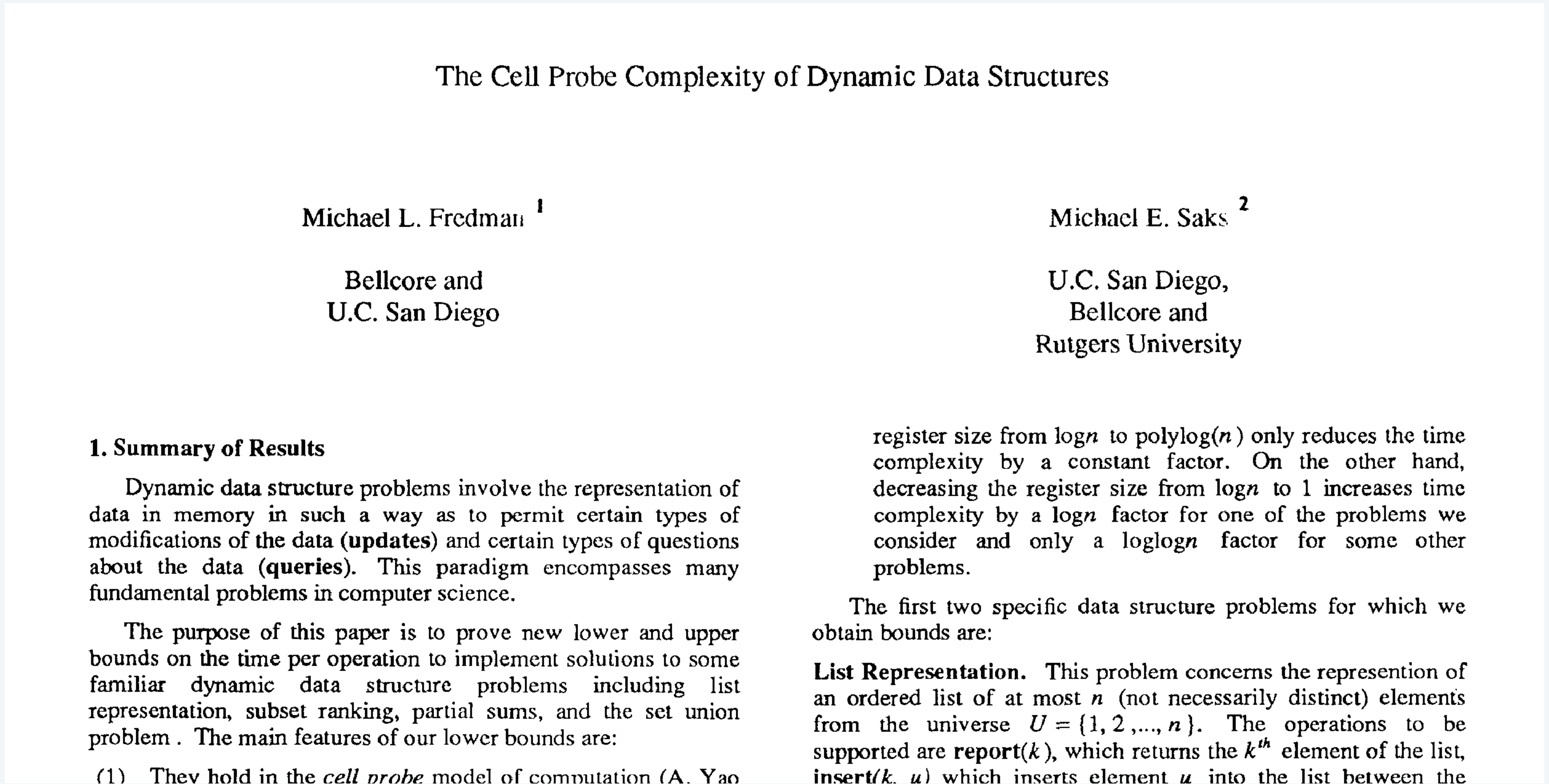
Ex. $\alpha(9876!) = 5$.

Two-parameter inverse Ackermann. $\alpha(m, n) = \min \{ k : \alpha_k(n) \leq 3 + m / n \}$.

A tight lower bound

Theorem. [Fredman–Saks 1989] In the worst case, any CELL-PROBE$(\log n)$ algorithm requires $\Omega(m\,\alpha(m, n))$ time to perform an intermixed sequence of m MAKE-SET, UNION, and FIND operations on a set of n elements.

Cell-probe model. [Yao 1981] Count only number of words of memory accessed; all other operations are free.



The Cell Probe Complexity of Dynamic Data Structures

Michael L. Fredman[1]

Bellcore and
U.C. San Diego

Michael E. Saks[2]

U.C. San Diego,
Bellcore and
Rutgers University

1. Summary of Results

Dynamic data structure problems involve the representation of data in memory in such a way as to permit certain types of modifications of the data (**updates**) and certain types of questions about the data (**queries**). This paradigm encompasses many fundamental problems in computer science.

The purpose of this paper is to prove new lower and upper bounds on the time per operation to implement solutions to some familiar dynamic data structure problems including list representation, subset ranking, partial sums, and the set union problem . The main features of our lower bounds are:

(1) They hold in the *cell probe* model of computation (A. Yao

register size from log$n$ to polylog(n) only reduces the time complexity by a constant factor. On the other hand, decreasing the register size from log$n$ to 1 increases time complexity by a log$n$ factor for one of the problems we consider and only a loglog$n$ factor for some other problems.

The first two specific data structure problems for which we obtain bounds are:

List Representation. This problem concerns the represention of an ordered list of at most n (not necessarily distinct) elements from the universe $U = \{1, 2, ..., n\}$. The operations to be supported are **report**(k), which returns the k^{th} element of the list, **insert**(k, u) which inserts element u into the list between the

Opsummering

- En samling af disjunkte mængder kan effektivt vedligeholdes under forening af mængder — med `Find-Set` og `Union` in logaritmisk tid (og bedre, amortiseret)
- Flere detaljer i CLRS 21.1-21.3 (og 21.4 🤓)

Spørg om hvad som helst

- Hvad vil I gerne vide mere om?
- Er der noget, I gerne vil have repeteret?
- Skriv i denne Google form (brug link eller QR kode)
- Jeg udvælger og svarer!

- Fredag: Opsamling og anvendelser