CS11921 - **Fall** 2023

Algorithm
Design & Analysis

Data Structures for Disjoint Sets

Ibrahim Albluwi

A Union-Find Data Structure

Problem. Given n items, each in a singleton set, build a data structure to support the following operations:

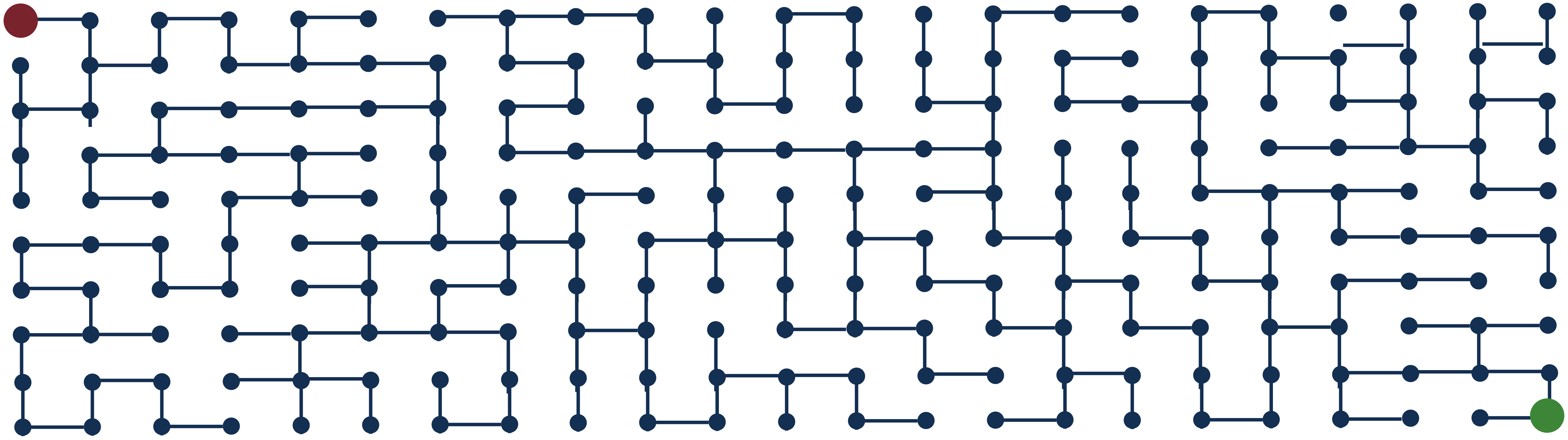
`UNION``(p, q)`	Merge the set containing **`p`** and the set containing **`q`** into one set.
`FIND``(p)`	Identify which set item **`p`** belongs to.
`CONNECTED``(p, q)`	Check if **`p`** and **`q`** belong to the same set.

A Union-Find **Data Structure**

Problem. Given n items, each in a singleton set, build a data structure to support the following operations:

`UNION(p, q)`	Merge the set containing **p** and the set containing **q** into one set.
`FIND(p)`	Identify which set item **p** belongs to.
`CONNECTED(p, q)`	Check if **p** and **q** belong to the same set.

Motivation. A basic data structure used in many applications.



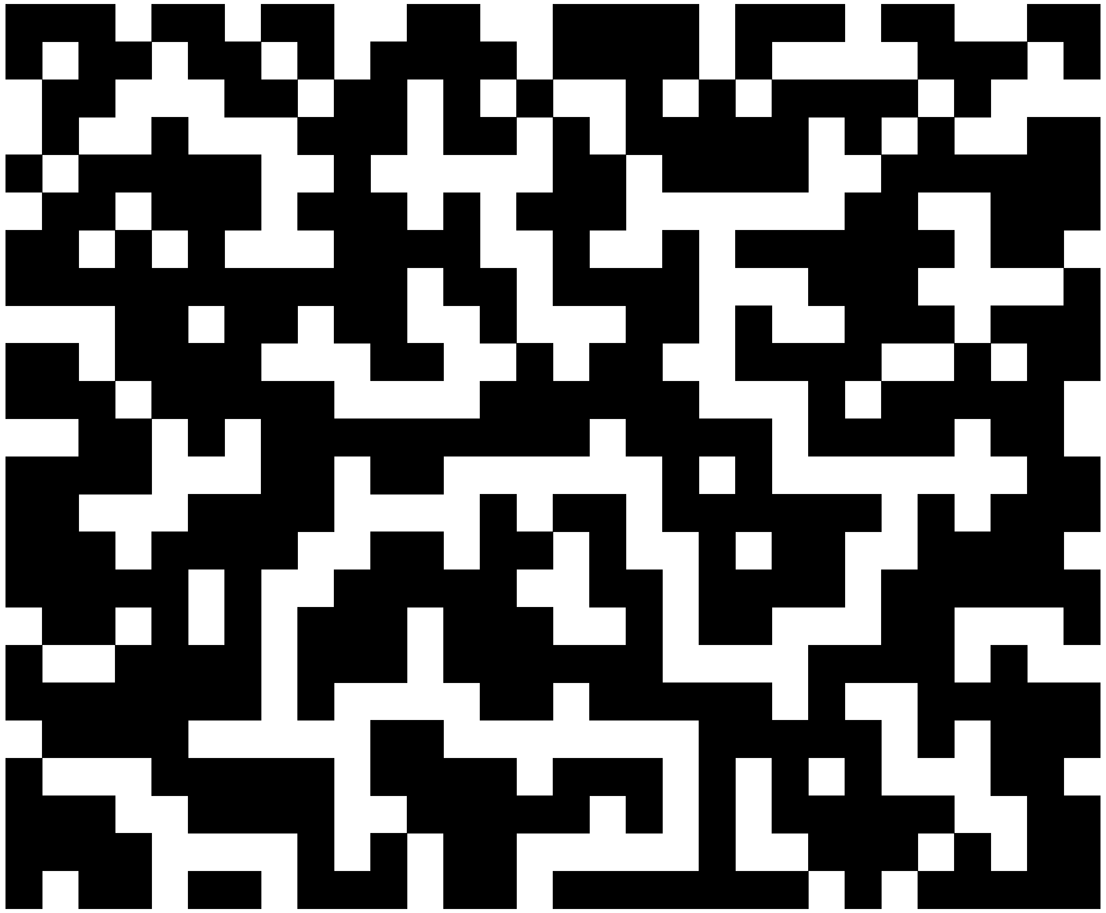
Is the **red** node connected to the **green** node?
Are all the nodes connected? Is there a cycle?

A Union-Find **Data Structure**

Problem. Given *n* items, each in a singleton set, build a data structure to support the following operations:

`UNION``(p, q)`	Merge the set containing **`p`** and the set containing **`q`** into one set.
`FIND``(p)`	Identify which set item **`p`** belongs to.
`CONNECTED``(p, q)`	Check if **`p`** and **`q`** belong to the same set.

Motivation. A basic data structure used in many applications.



Does this plate conduct electricity?

(black = conductive material
white = insulating material)

A Simple Implementation: Array-based Quick-Find

- Store a unique **`id`** for each set in an array.
- Initially, every element is in a singleton set.
- **`FIND`**`(p)` returns the `id` of `p`.
- **`UNION`**`(p, q)` changes the `id` of all the elements in the set of `q` to the `id` of the set of `p`.

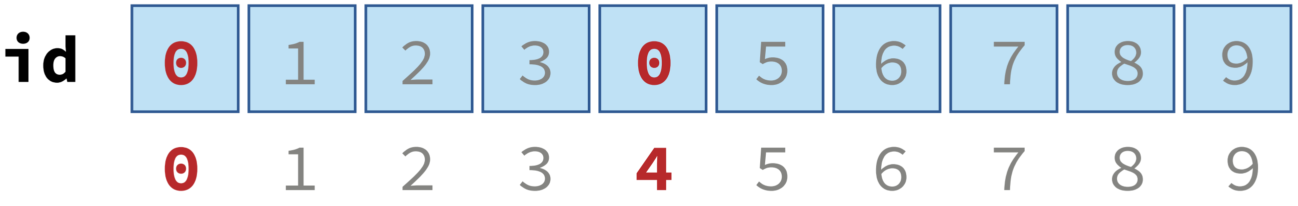
Example.

id	0	1	2	3	4	5	6	7	8	9
	0	1	2	3	4	5	6	7	8	9

A Simple Implementation: Array-based Quick-Find

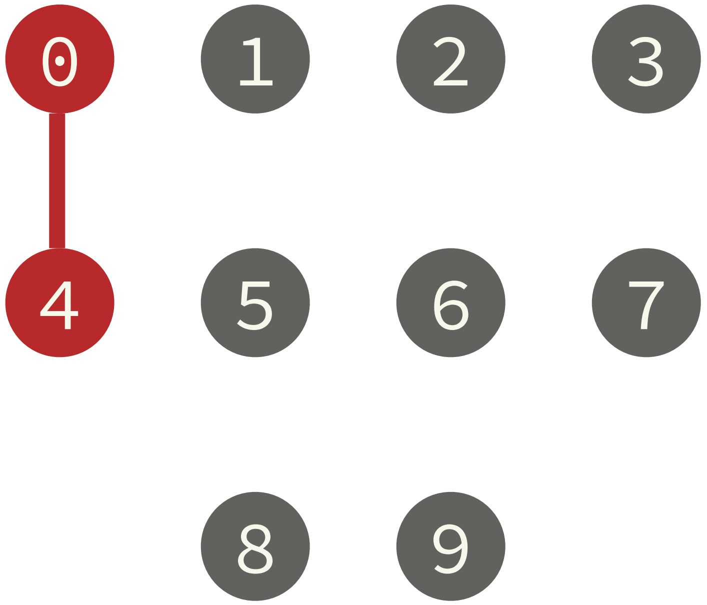
- Store a unique **`id`** for each set in an array.
- Initially, every element is in a singleton set.
- **`FIND`**`(p)` returns the `id` of `p`.
- **`UNION`**`(p, q)` changes the `id` of all the elements in the set of `q` to the `id` of the set of `p`.

Example.



id	**0**	1	2	3	**0**	5	6	7	8	9
	0	1	2	3	**4**	5	6	7	8	9

- **`UNION`**`(0, 4)`



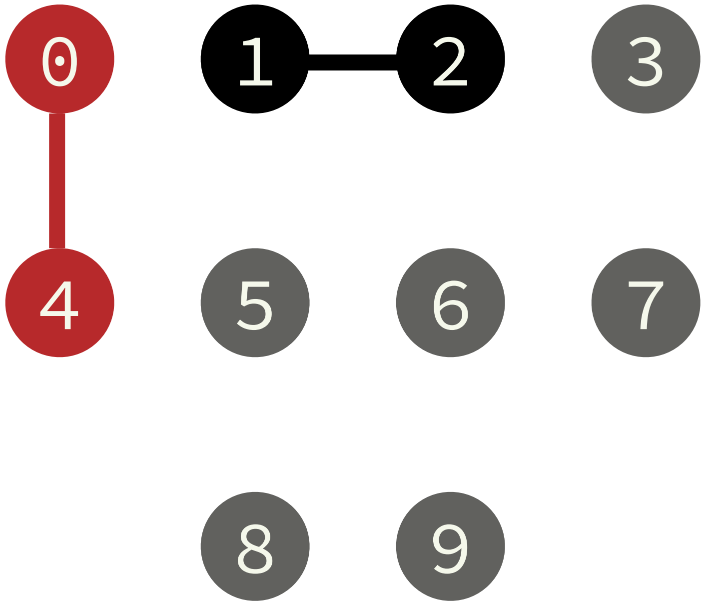
A Simple Implementation: Array-based Quick-Find

- Store a unique **`id`** for each set in an array.
- Initially, every element is in a singleton set.
- **`FIND`**`(p)` returns the `id` of `p`.
- **`UNION`**`(p, q)` changes the `id` of all the elements in the set of `q` to the `id` of the set of `p`.

Example.

id	0	1	1	3	0	5	6	7	8	9
	0	1	2	3	4	5	6	7	8	9

- `UNION(0, 4)`
- **`UNION`**`(1, 2)`



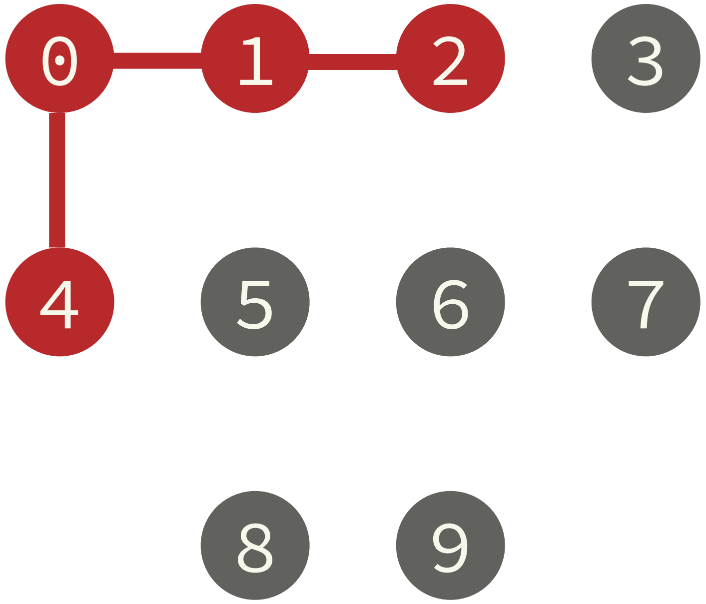
A Simple Implementation: Array-based Quick-Find

- Store a unique **`id`** for each set in an array.
- Initially, every element is in a singleton set.
- **`FIND`**`(p)` returns the `id` of `p`.
- **`UNION`**`(p, q)` changes the `id` of all the elements in the set of `q` to the `id` of the set of `p`.

Example.

id	**0**	**0**	**0**	3	**0**	5	6	7	8	9
	0	**1**	**2**	3	**4**	5	6	7	8	9

- `UNION(0, 4)`
- `UNION(1, 2)`
- **`UNION`**`(0, 1)`



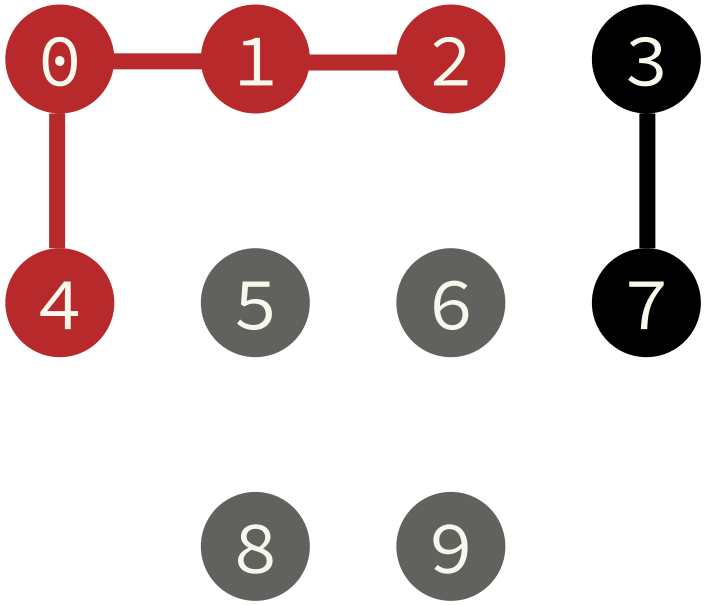
A Simple Implementation: Array-based Quick-Find

- Store a unique **`id`** for each set in an array.
- Initially, every element is in a singleton set.
- **`FIND`**`(p)` returns the `id` of `p`.
- **`UNION`**`(p, q)` changes the `id` of all the elements in the set of `q` to the `id` of the set of `p`.

Example.

id	0	0	0	3	0	5	6	3	8	9
	0	1	2	3	4	5	6	7	8	9

- **`UNION`**`(0, 4)`
- **`UNION`**`(1, 2)`
- **`UNION`**`(0, 1)`
- **`UNION`**`(3, 7)`



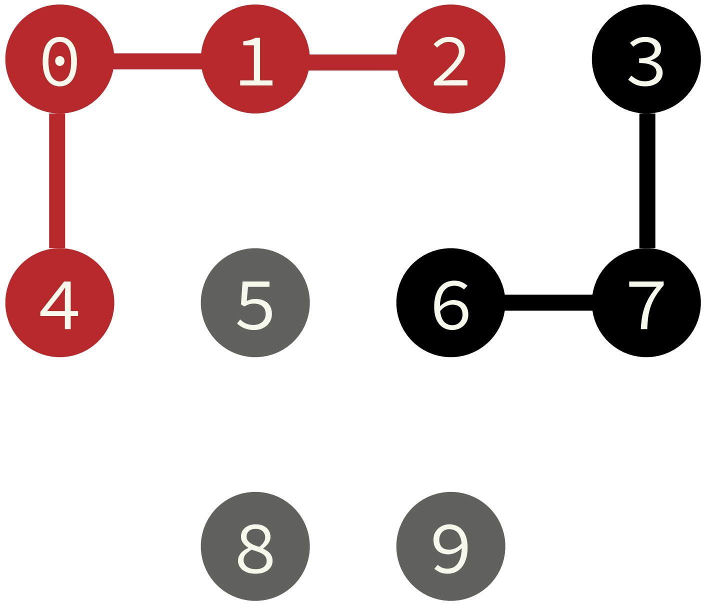
A Simple Implementation: Array-based Quick-Find

- Store a unique **`id`** for each set in an array.
- Initially, every element is in a singleton set.
- **`FIND`**`(p)` returns the `id` of `p`.
- **`UNION`**`(p, q)` changes the `id` of all the elements in the set of `q` to the `id` of the set of `p`.

Example.

id	0	0	0	6	0	5	6	6	8	9
	0	1	2	3	4	5	6	7	8	9

- `UNION(0, 4)`
- `UNION(1, 2)`
- `UNION(0, 1)`
- `UNION(3, 7)`
- **`UNION(6, 7)`**



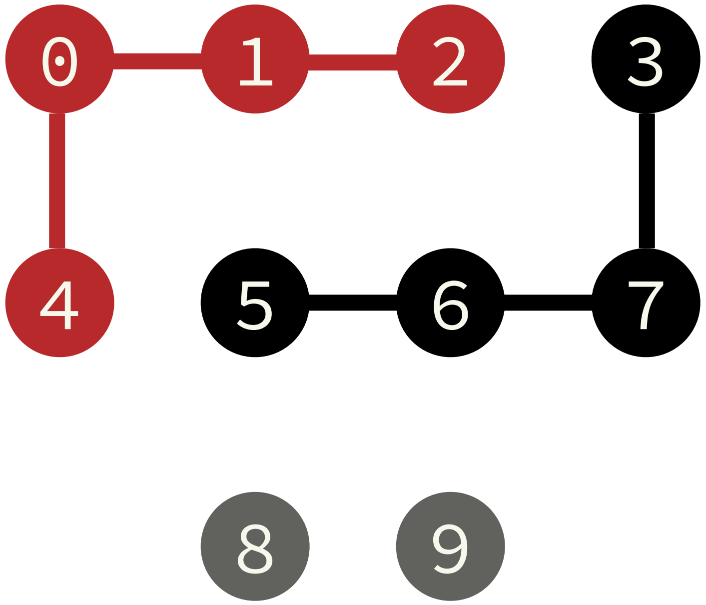
A Simple Implementation: Array-based Quick-Find

- Store a unique **`id`** for each set in an array.
- Initially, every element is in a singleton set.
- **`FIND`**`(p)` returns the `id` of `p`.
- **`UNION`**`(p, q)` changes the `id` of all the elements in the set of `q` to the `id` of the set of `p`.

Example.

id	0	0	0	5	0	5	5	5	8	9
	0	1	2	3	4	5	6	7	8	9

- `UNION(0, 4)`
- `UNION(1, 2)`
- `UNION(0, 1)`
- `UNION(3, 7)`
- `UNION(6, 7)`
- **`UNION(5, 6)`**



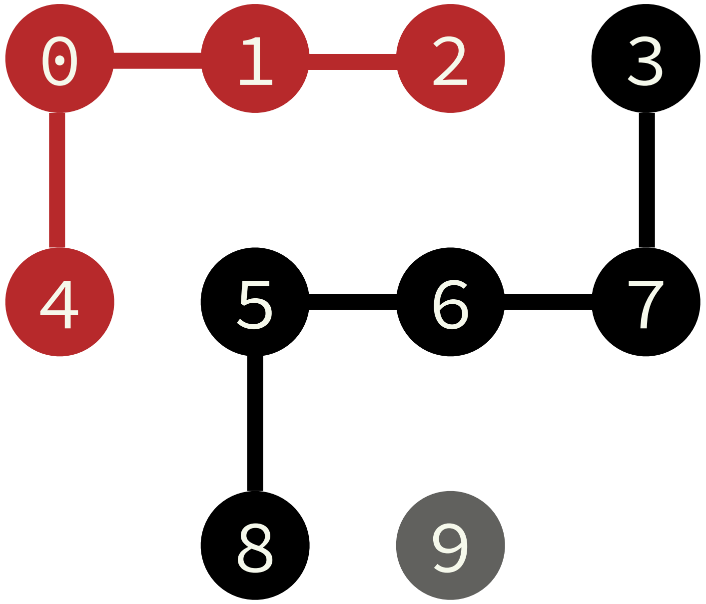
A Simple Implementation: Array-based Quick-Find

- Store a unique **`id`** for each set in an array.
- Initially, every element is in a singleton set.
- **`FIND`**`(p)` returns the `id` of `p`.
- **`UNION`**`(p, q)` changes the `id` of all the elements in the set of `q` to the `id` of the set of `p`.

Example.

id	0	0	0	8	0	8	8	8	8	9
	0	1	2	3	4	5	6	7	8	9

- `UNION(0, 4)`
- `UNION(1, 2)`
- `UNION(0, 1)`
- `UNION(3, 7)`
- `UNION(6, 7)`
- `UNION(5, 6)`
- **`UNION(8, 5)`**



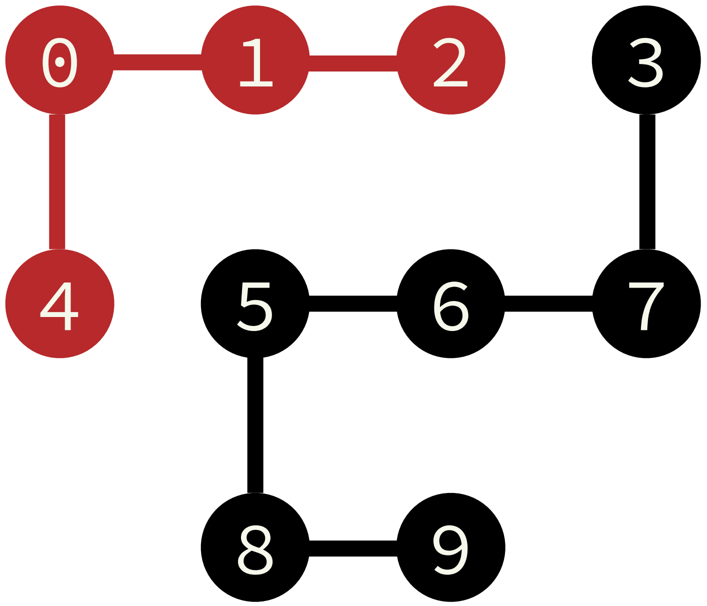
A Simple Implementation: Array-based Quick-Find

- Store a unique **`id`** for each set in an array.
- Initially, every element is in a singleton set.
- **`FIND`**`(p)` returns the `id` of `p`.
- **`UNION`**`(p, q)` changes the `id` of all the elements in the set of `q` to the `id` of the set of `p`.

Example.

	0	1	2	3	4	5	6	7	8	9
id	0	0	0	8	0	8	8	8	8	8

- `UNION(0, 4)`
- `UNION(1, 2)`
- `UNION(0, 1)`
- `UNION(3, 7)`
- `UNION(6, 7)`
- `UNION(5, 6)`
- `UNION(8, 5)`
- **`UNION`**`(8, 9)`

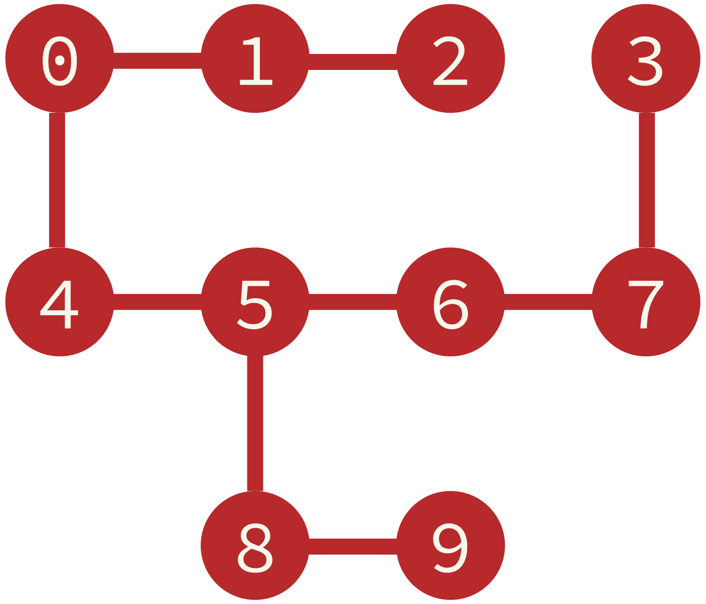


A Simple Implementation: Array-based Quick-Find

- Store a unique **`id`** for each set in an array.
- Initially, every element is in a singleton set.
- **`FIND`**`(p)` returns the `id` of `p`.
- **`UNION`**`(p, q)` changes the `id` of all the elements in the set of `q` to the `id` of the set of `p`.

Example.

id	0	0	0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6	7	8	9



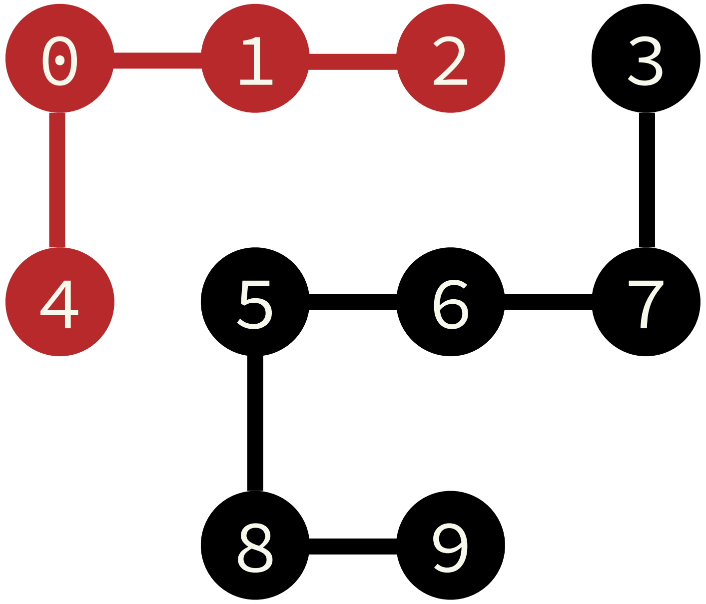
- `UNION(0, 4)`
- `UNION(1, 2)`
- `UNION(0, 1)`
- `UNION(3, 7)`
- `UNION(6, 7)`
- `UNION(5, 6)`
- `UNION(8, 5)`
- `UNION(8, 9)`
- `UNION(4, 5)`

A Simple Implementation: Array-based Quick-Find

- Store a unique **`id`** for each set in an array.
- Initially, every element is in a singleton set.
- **`FIND`**`(p)` returns the `id` of `p`.
- **`UNION`**`(p, q)` changes the `id` of all the elements in the set of `q` to the `id` of the set of `p`.

Example.

id	0	0	0	8	0	8	8	8	8	8
	0	1	2	3	4	5	6	7	8	9



`UNION``(p, q)`

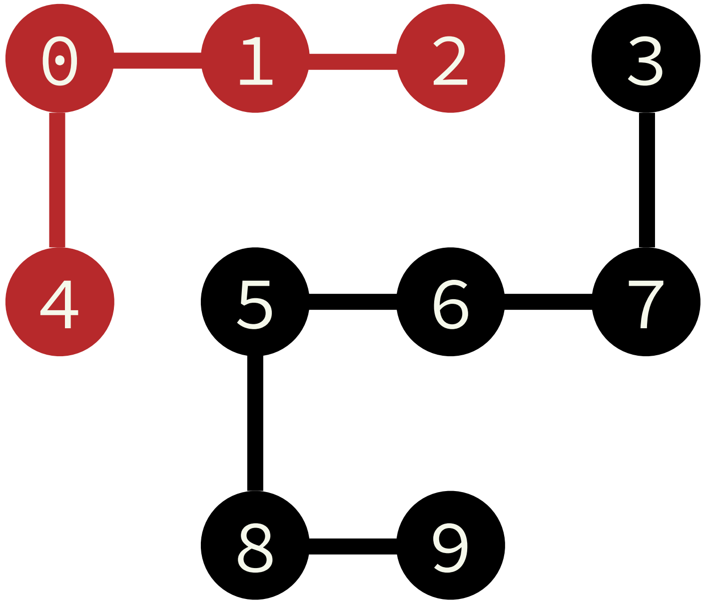
```
if (id[p] == id[q])
    return
```

A Simple Implementation: Array-based Quick-Find

- Store a unique **`id`** for each set in an array.
- Initially, every element is in a singleton set.
- **`FIND`**`(p)` returns the `id` of `p`.
- **`UNION`**`(p, q)` changes the `id` of all the elements in the set of `q` to the `id` of the set of `p`.

Example.

id	0	0	0	8	0	8	8	8	8	8
	0	1	2	3	4	5	6	7	8	9



`UNION``(p, q)`

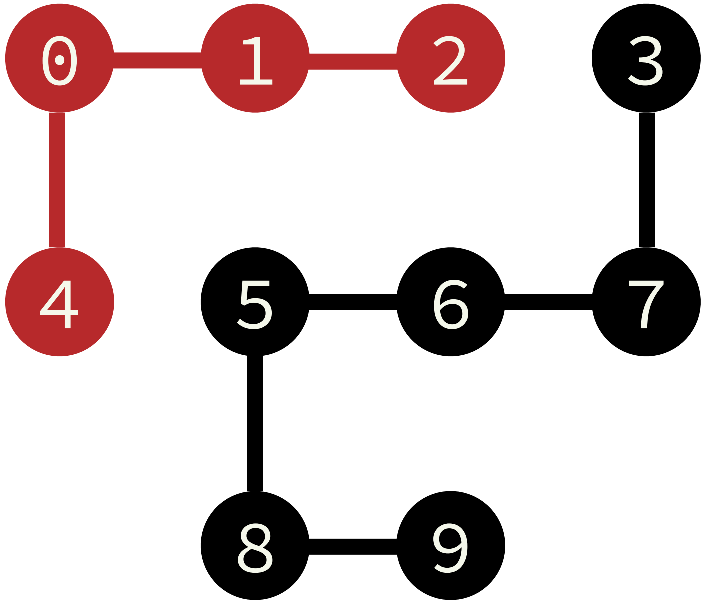
```
if (id[p] == id[q])
   return


for (i = 0 to n-1)
```

A Simple Implementation: Array-based Quick-Find

- Store a unique **`id`** for each set in an array.
- Initially, every element is in a singleton set.
- **`FIND`**`(p)` returns the `id` of `p`.
- **`UNION`**`(p, q)` changes the `id` of all the elements in the set of `q` to the `id` of the set of `p`.

Example.

id	0	0	0	8	0	8	8	8	8	8
	0	1	2	3	4	5	6	7	8	9



UNION(p, q)

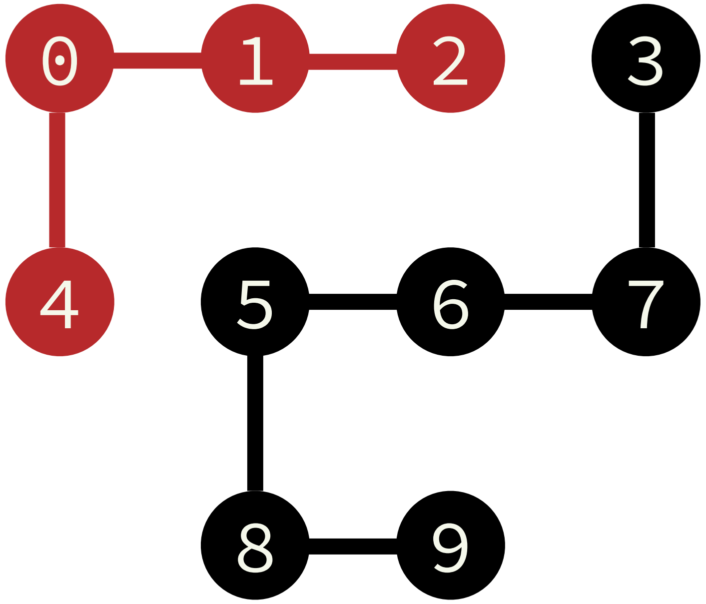
```
if (id[p] == id[q])
    return


for (i = 0 to n-1)
    if (id[i] == id[q])
        id[i] = id[p]
```

A Simple Implementation: Array-based Quick-Find

- Store a unique **`id`** for each set in an array.
- Initially, every element is in a singleton set.
- **`FIND`**`(p)` returns the `id` of `p`.
- **`UNION`**`(p, q)` changes the `id` of all the elements in the set of `q` to the `id` of the set of `p`.

Example.

id	0	0	0	8	0	8	8	8	8	8
	0	1	2	3	4	5	6	7	8	9



`UNION(p, q)`

```
if (id[p] == id[q])
   return


for (i = 0 to n-1)
   if (id[i] == id[q])
       id[i] = id[p]
```

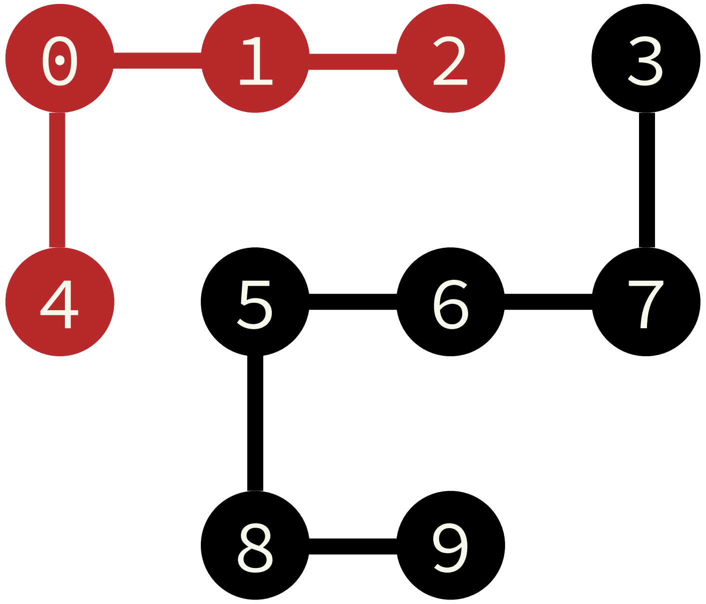
Buggy Code?

A Simple Implementation: Array-based Quick-Find

- Store a unique **`id`** for each set in an array.
- Initially, every element is in a singleton set.
- **`FIND`**`(p)` returns the `id` of `p`.
- **`UNION`**`(p, q)` changes the `id` of all the elements in the set of `q` to the `id` of the set of `p`.

Example.

id	0	0	0	8	0	8	8	8	8	8
	0	1	2	3	4	5	6	7	8	9



`UNION``(p, q)`

```
if (id[p] == id[q])
   return

id_q = id[q]
for (i = 0 to n-1)
   if (id[i] == id_q)
       id[i] = id[p]
```

A Simple Implementation: Array-based Quick-Find

- Store a unique **`id`** for each set in an array.
- Initially, every element is in a singleton set.
- **`FIND`**`(p)` returns the `id` of `p`.
- **`UNION`**`(p, q)` changes the `id` of all the elements in the set of `q` to the `id` of the set of `p`.

Running Time.

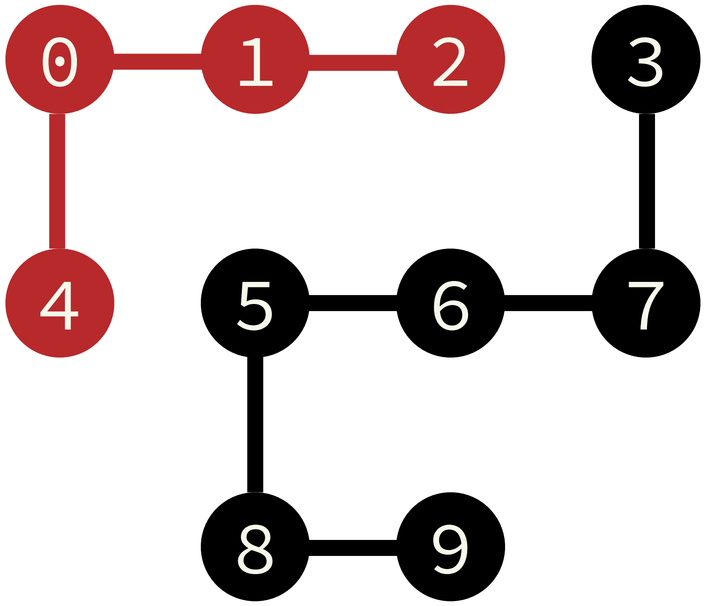
`FIND:` $\Theta(1)$

`UNION:` $\Theta(n)$

Cost Model. Number of array accesses

Example.

id	0	0	0	8	0	8	8	8	8	8
	0	1	2	3	4	5	6	7	8	9



`UNION(p, q)`

```
if (id[p] == id[q])
   return

id_q = id[q]
for (i = 0 to n-1)
   if (id[i] == id_q)
       id[i] = id[p]
```

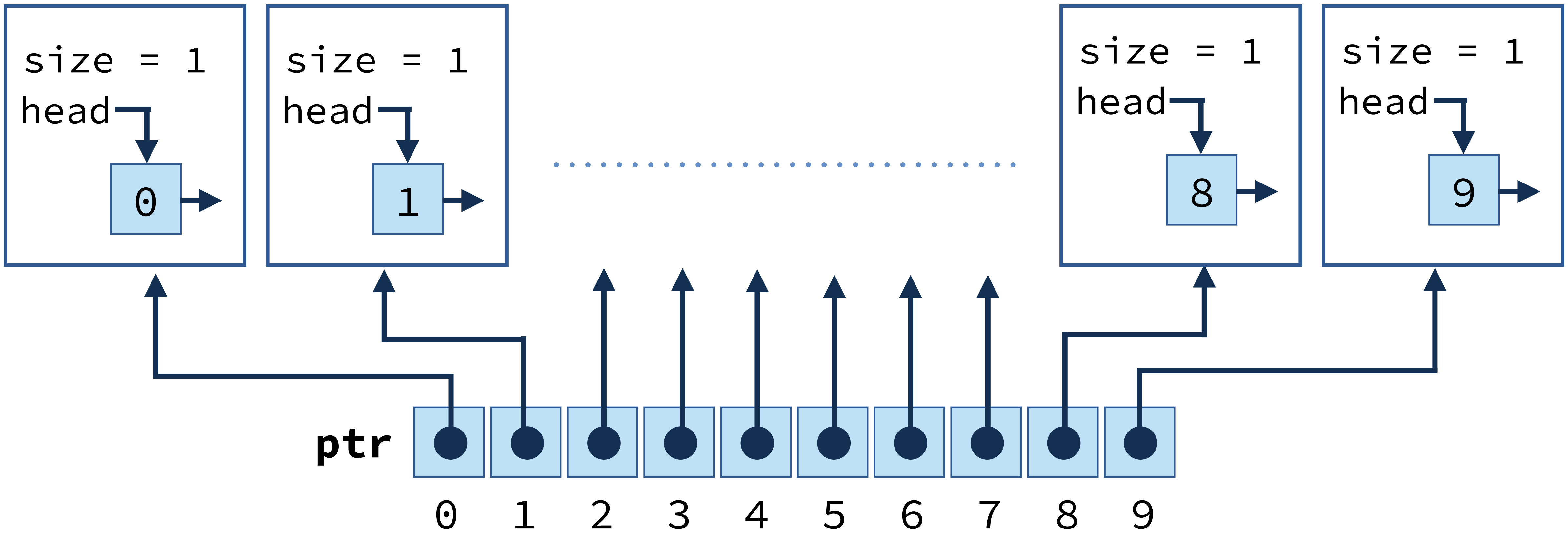
Improvement Attempt 1: Link-List-Based Quick-Find

Idea. Each set is a linked linked list.
Rationale. Iterate only over the elements of the smaller set when merging sets.

Improvement Attempt 1: Link-List-Based Quick-Find

Idea. Each set is a linked linked list.
Rationale. Iterate only over the elements of the smaller set when merging sets.

Example. Each element is in a singleton set.

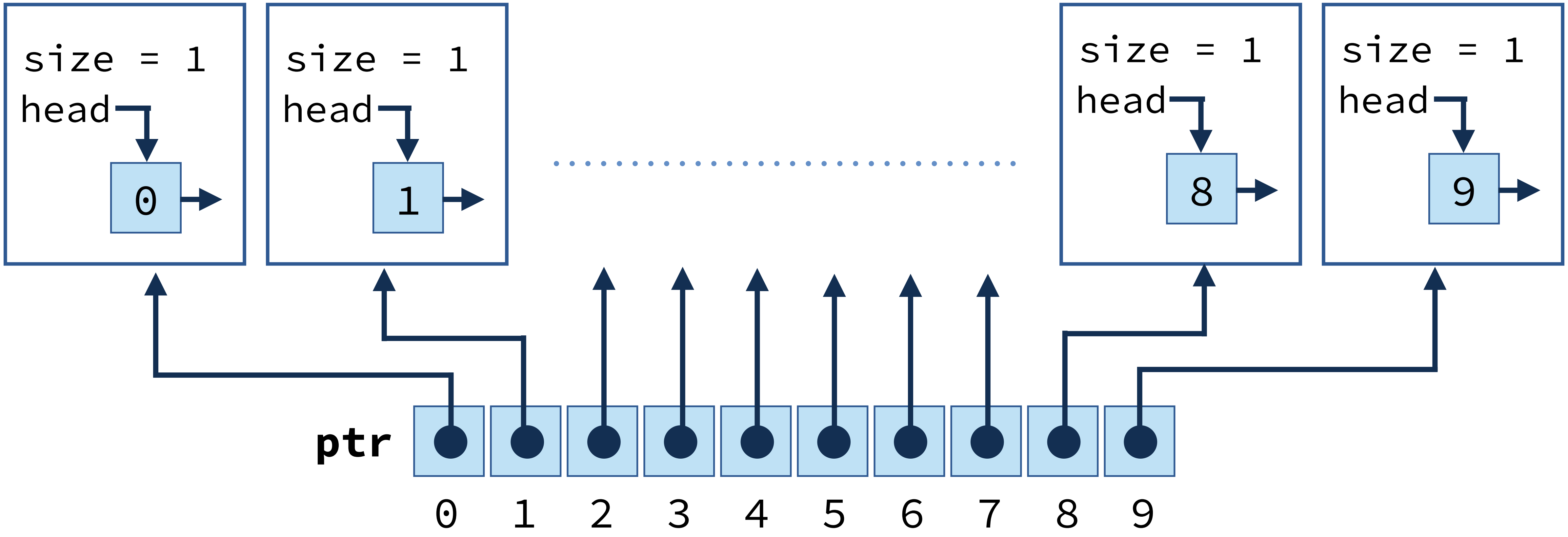


Improvement Attempt 1: Link-List-Based Quick-Find

Idea. Each set is a linked linked list.
Rationale. Iterate only over the elements of the smaller set when merging sets.

- **`FIND`**`(p)` Returns `ptr[p].head`
- **`UNION`**`(p, q)` Merges the two linked lists of `p` and `q`.

Example. Each element is in a singleton set.

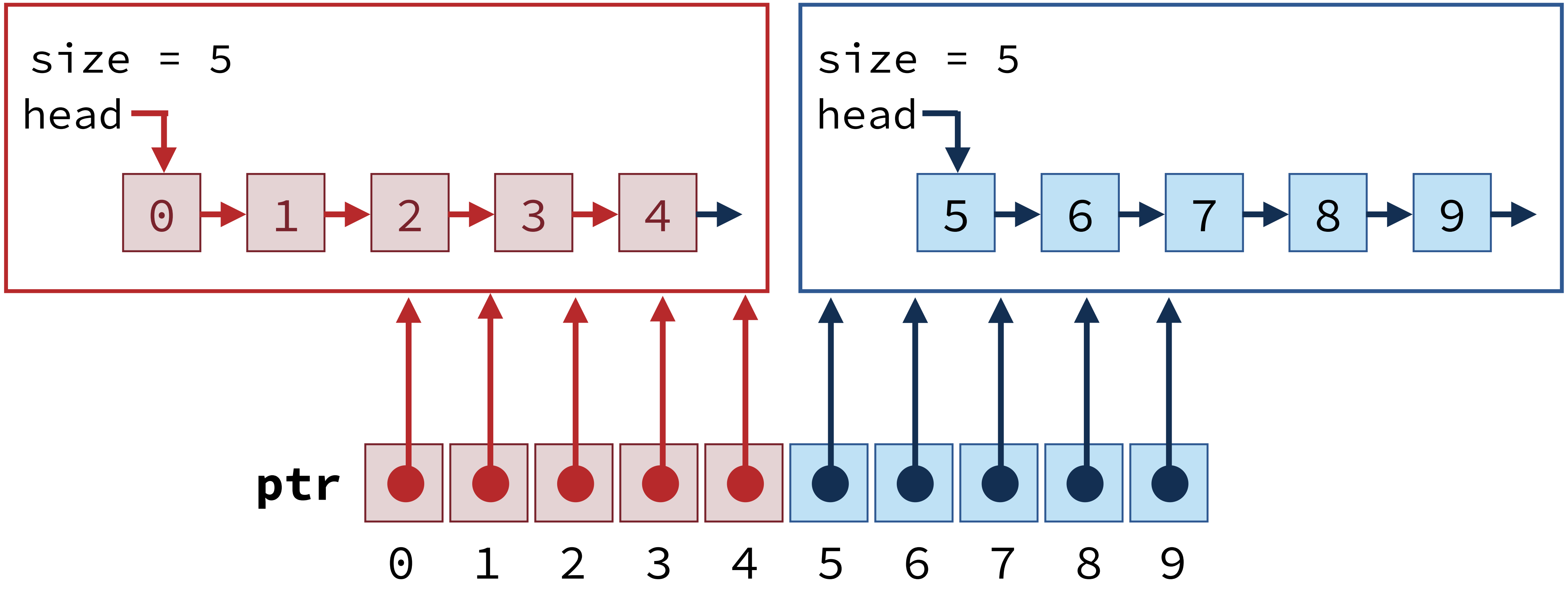


Improvement Attempt 1: Link-List-Based Quick-Find

Idea. Each set is a linked linked list.
Rationale. Iterate only over the elements of the smaller set when merging sets.

- **`FIND`**`(p)` Returns `ptr[p].head`
- **`UNION`**`(p, q)` Merges the two linked lists of `p` and `q`.

Example. Elements `0–4` are in one set and `5–9` are in another set.

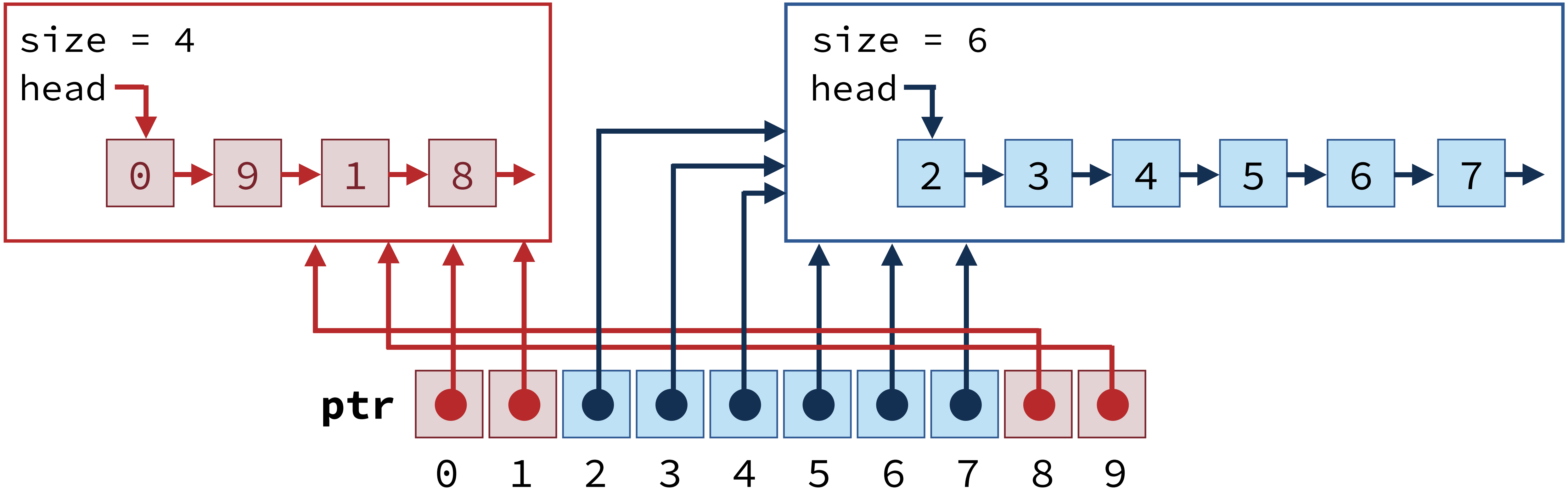


Improvement Attempt 1: Link-List-Based Quick-Find

Idea. Each set is a linked linked list.
Rationale. Iterate only over the elements of the smaller set when merging sets.

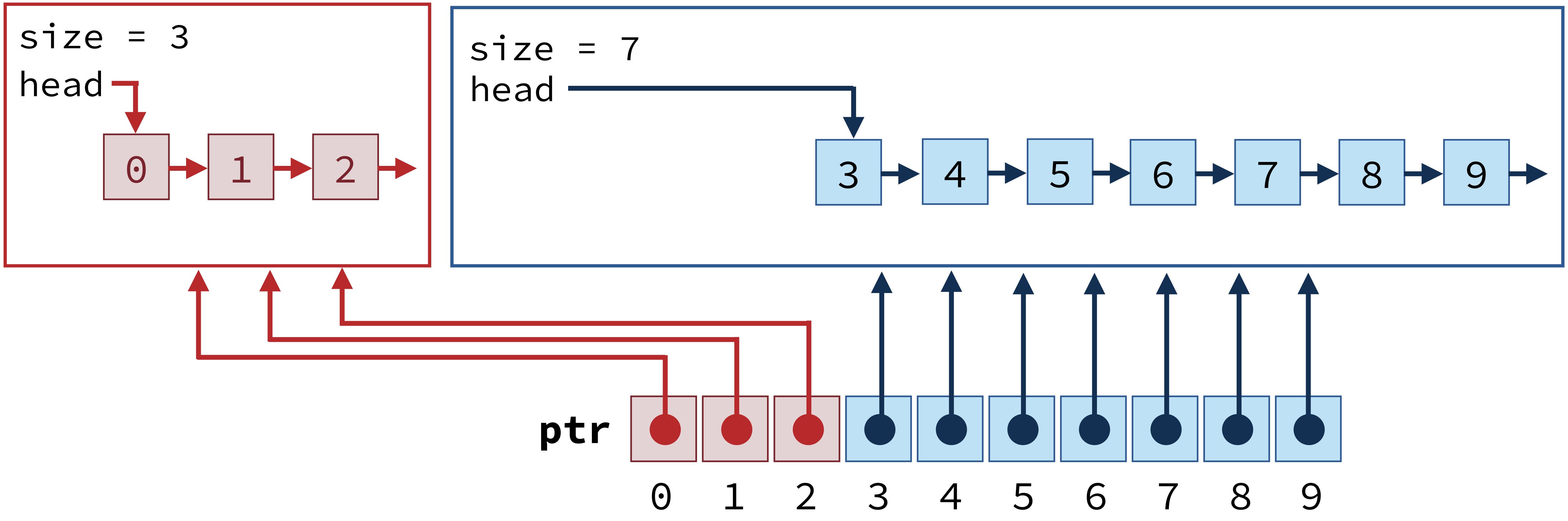
- **`FIND`**`(p)` Returns `ptr[p].head`
- **`UNION`**`(p, q)` Merges the two linked lists of `p` and `q`.

Example. Two sets: `{0,1,9,8}` and `{2,3,4,5,6,7}`.



Link-List-Based Quick-Find: **`UNION`** Example

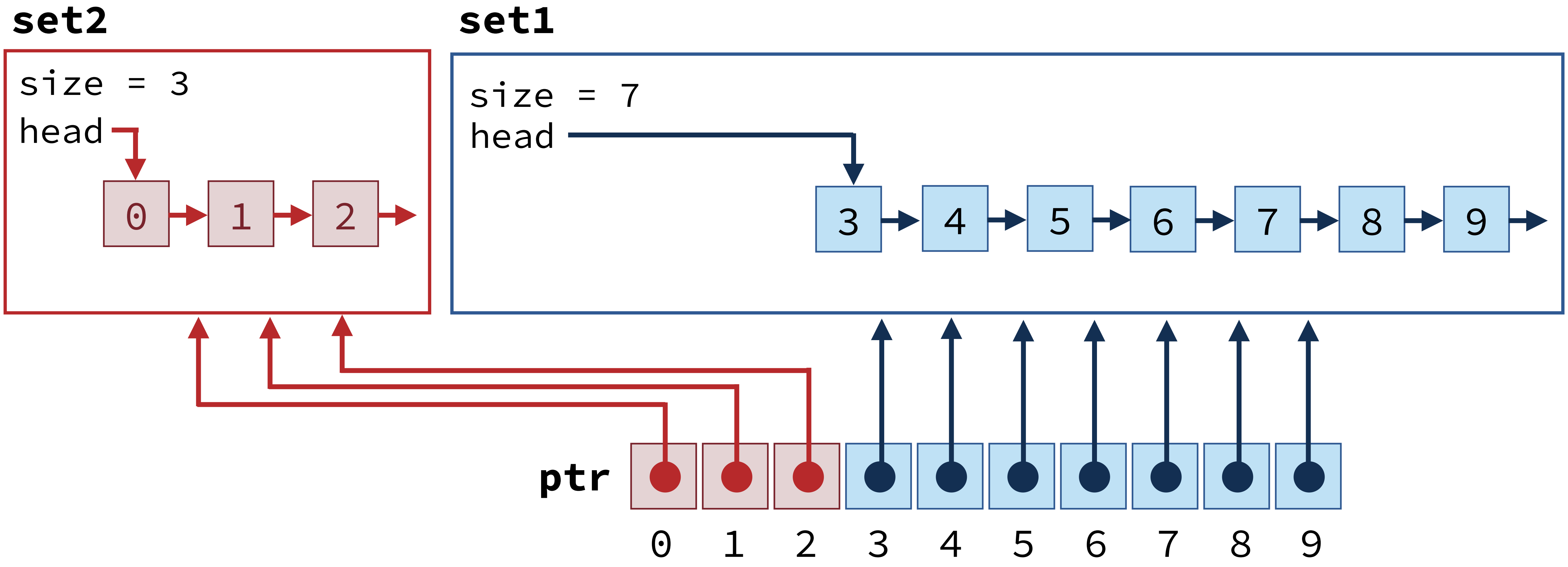
Example. **`UNION`**`(1, 7)`



Link-List-Based Quick-Find: `UNION` Example

Example. **`UNION`**`(1, 7)`

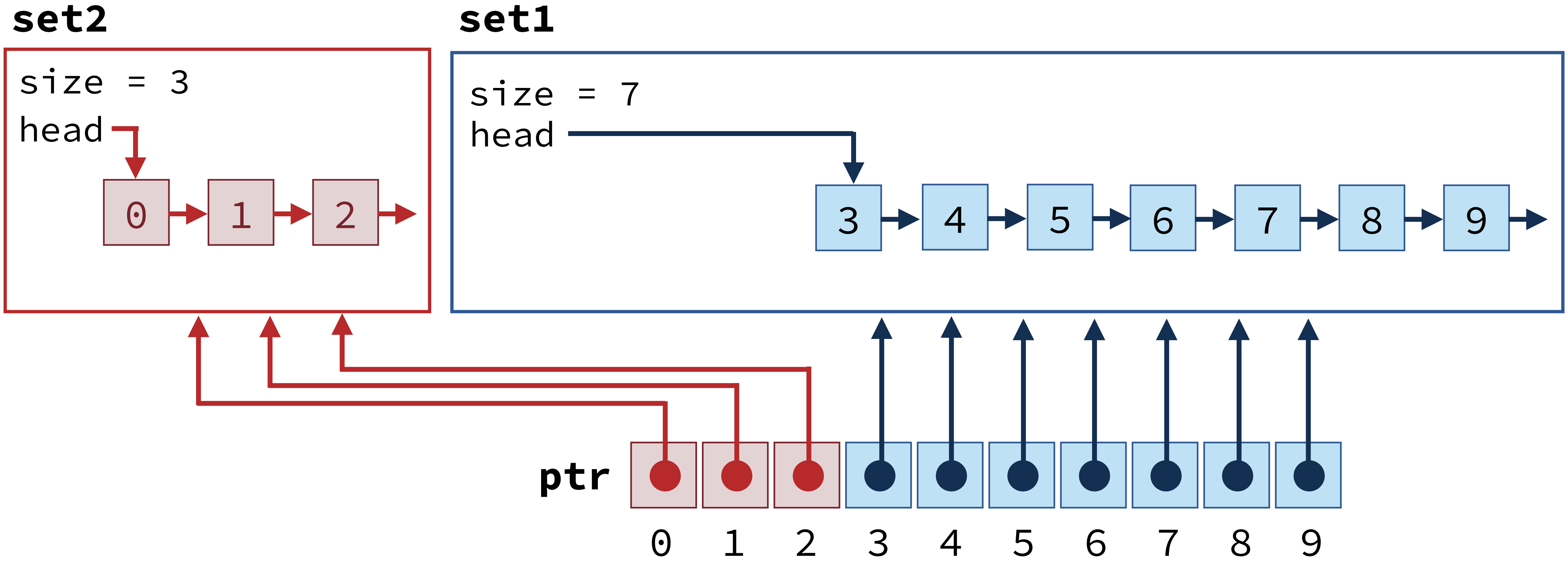
1. Make `set2` point at the smaller set and `set1` at the larger set.



Link-List-Based Quick-Find: `UNION` Example

Example. **`UNION(1, 7)`**

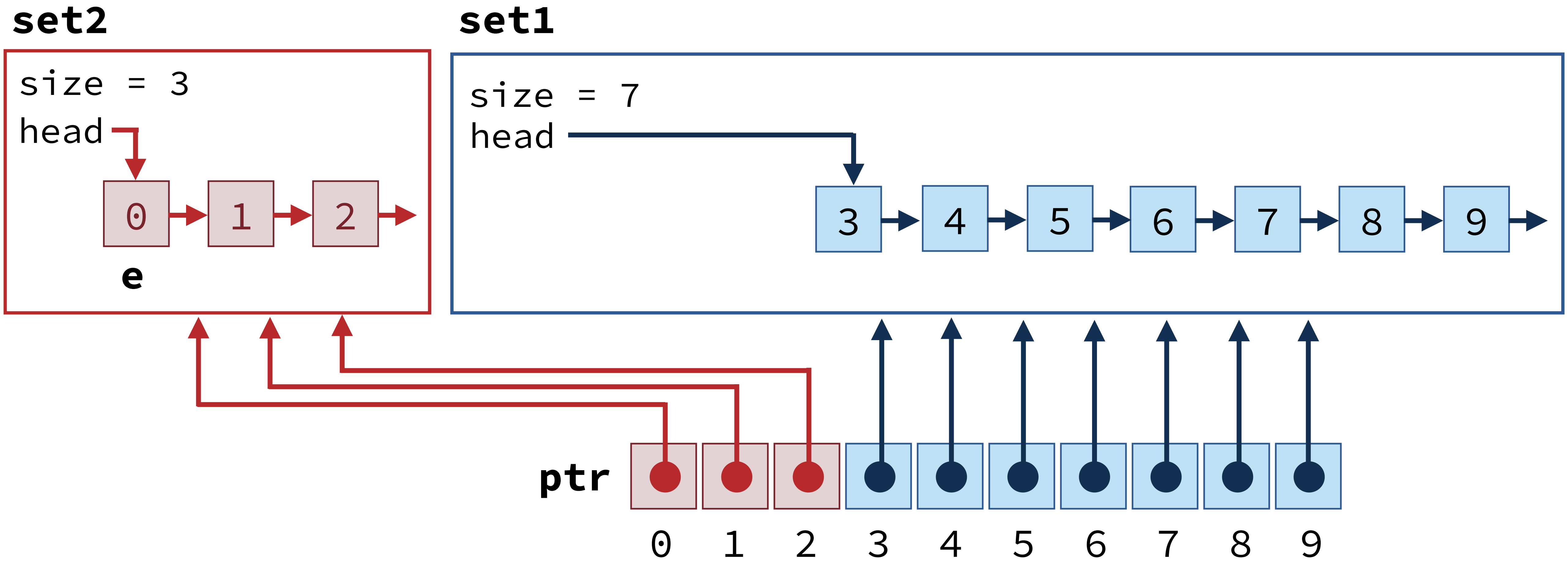
1. Make `set2` point at the smaller set and `set1` at the larger set.
2. For each element `e` in `set2:`
   `ptr[e] = set1`
   Move node `e` to `set1` and increment `set1.size`



Link-List-Based Quick-Find: `UNION` Example

Example. `UNION(1, 7)`

```
1. Make set2 point at the smaller set and set1 at the larger set.
2. For each element e in set2:
       ptr[e] = set1
       Move node e to set1 and increment set1.size
```



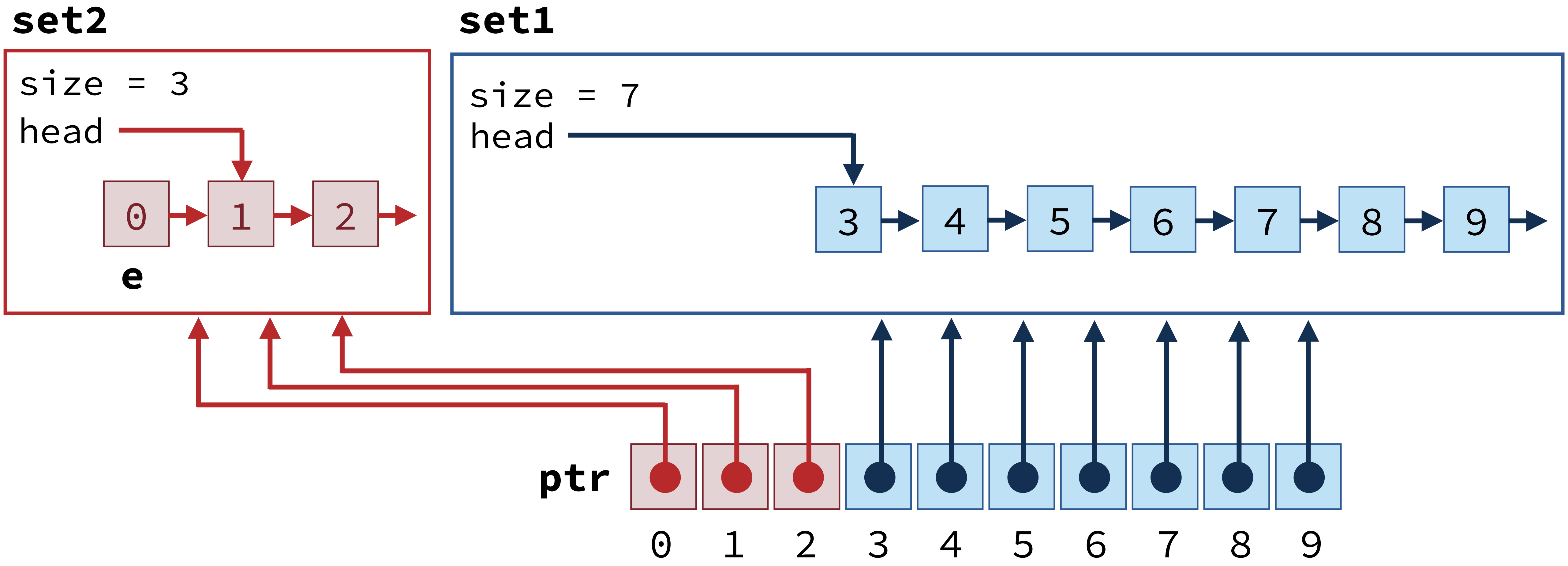
Link-List-Based Quick-Find: `UNION` Example

Example. **`UNION`**`(1, 7)`

1. Make `set2` point at the smaller set and `set1` at the larger set.
2. For each element `e` in `set2:`

   `ptr[e] = set1`

   Move node `e` to `set1` and increment `set1.size`



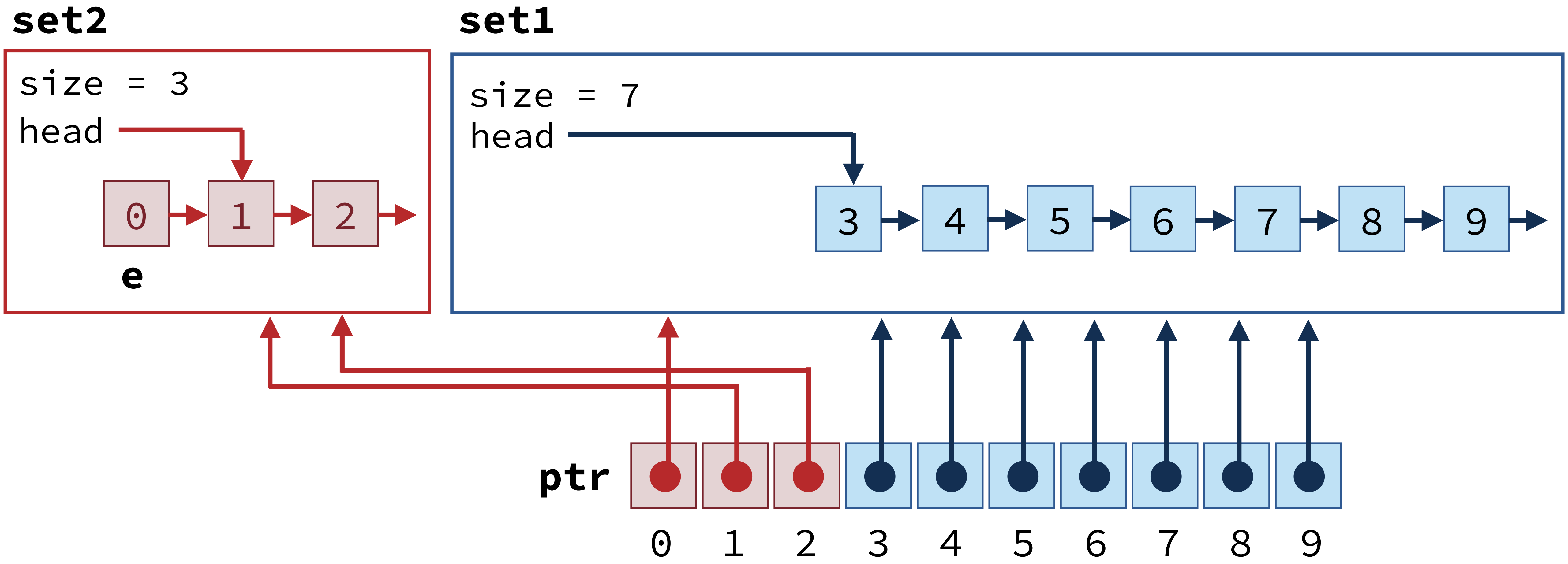
Link-List-Based Quick-Find: `UNION` Example

Example. **`UNION`**`(1, 7)`

1. Make `set2` point at the smaller set and `set1` at the larger set.
2. For each element `e` in `set2:`

   `ptr[e] = set1`

   Move node `e` to `set1` and increment `set1.size`

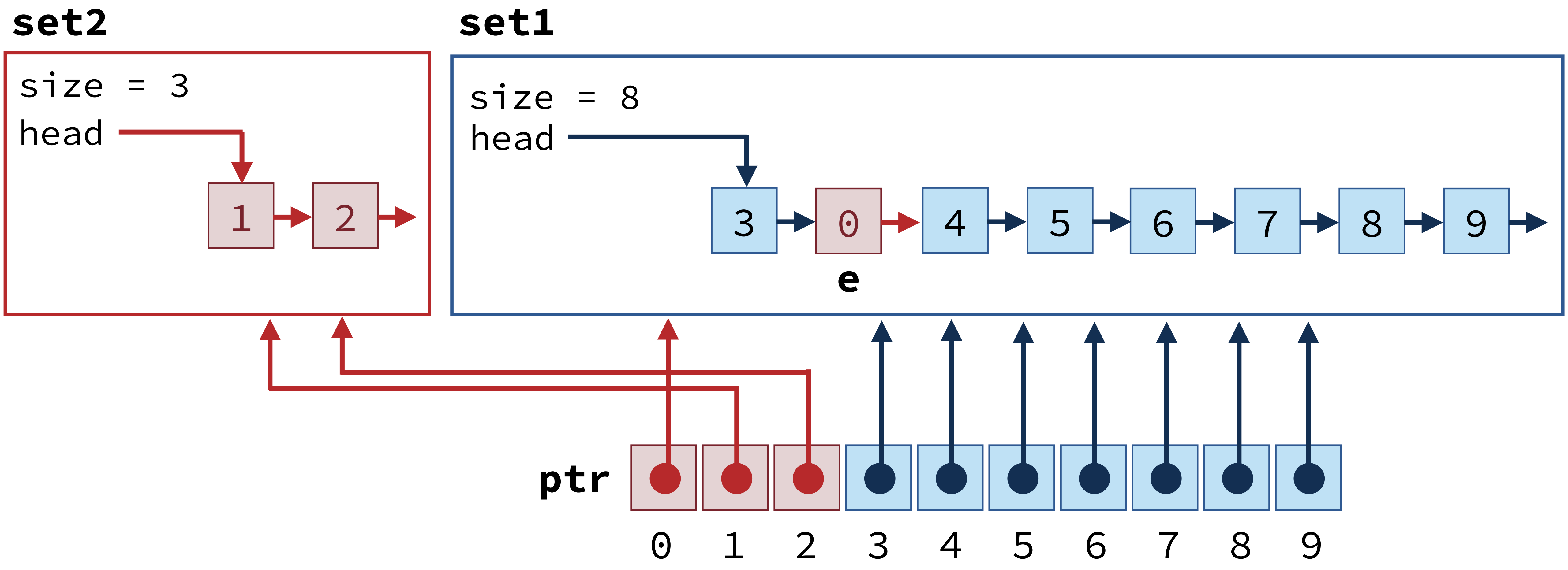


Link-List-Based Quick-Find: `UNION` Example

Example. **`UNION`**`(1, 7)`

1. Make `set2` point at the smaller set and `set1` at the larger set.
2. For each element `e` in `set2:`

```
    ptr[e] = set1
    Move node e to set1 and increment set1.size
```

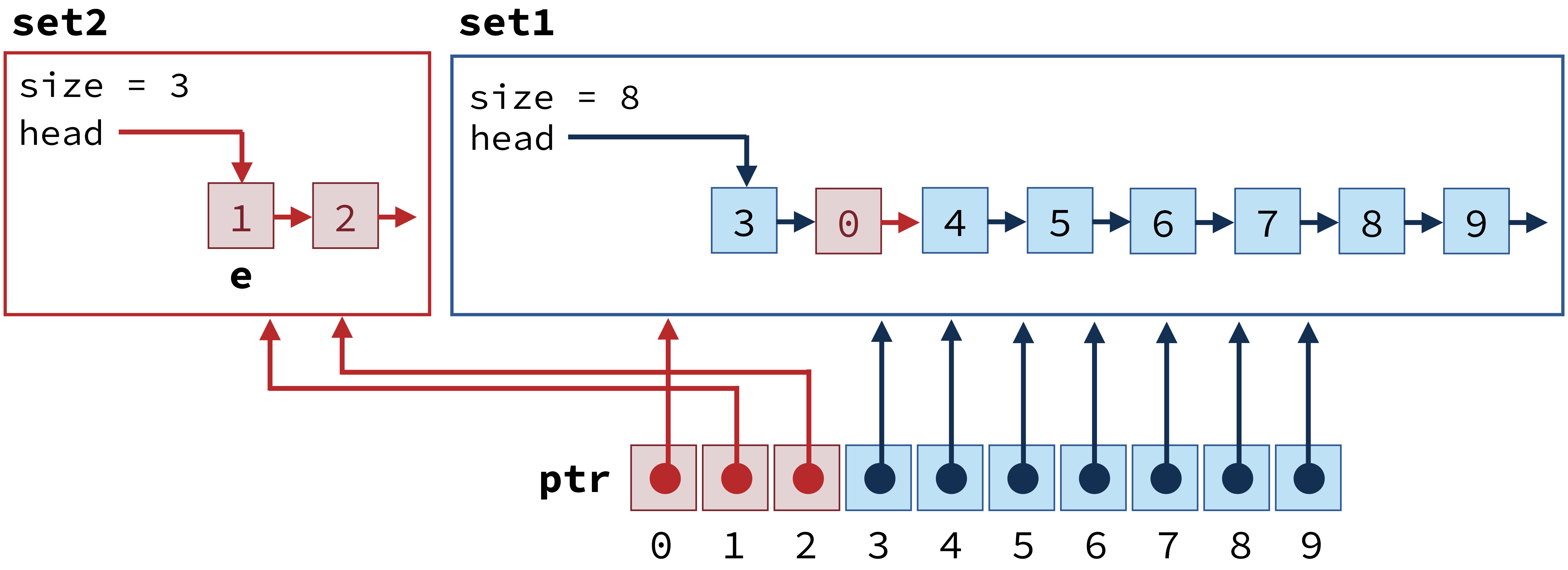


Link-List-Based Quick-Find: `UNION` Example

Example. **`UNION`**`(1, 7)`

1. Make `set2` point at the smaller set and `set1` at the larger set.
2. For each element `e` in `set2:`

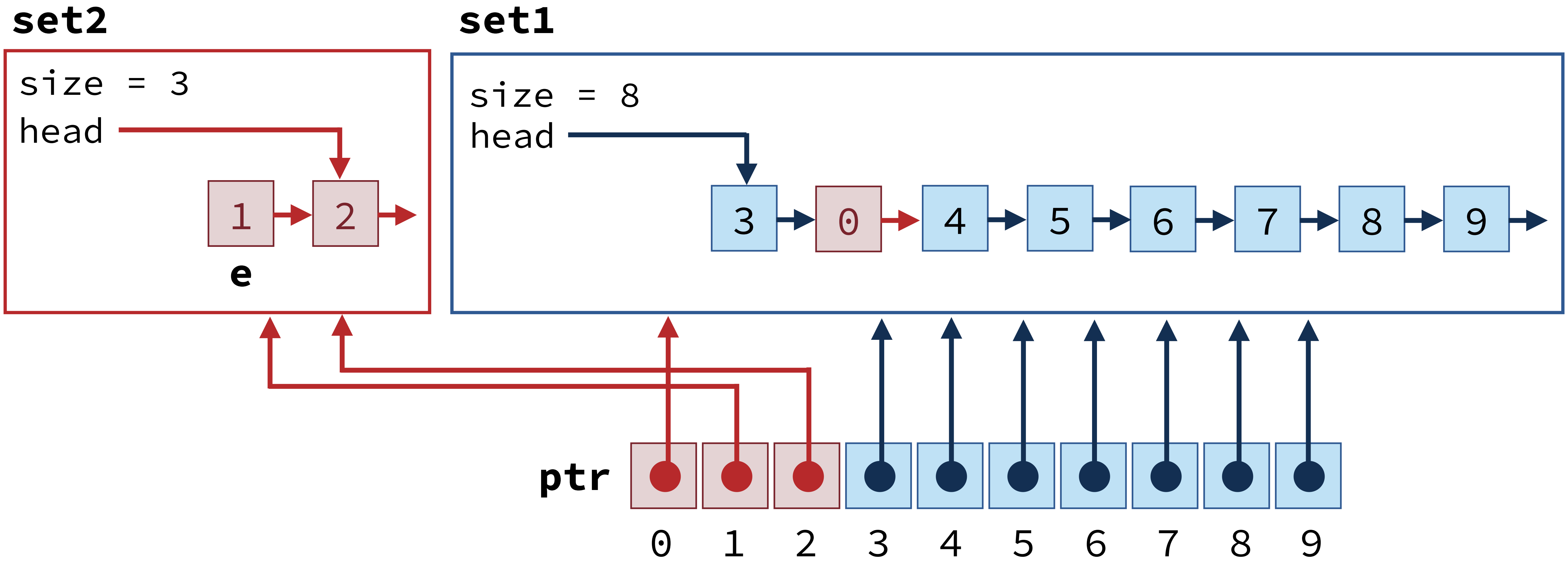
```
    ptr[e] = set1
    Move node e to set1 and increment set1.size
```



Link-List-Based Quick-Find: `UNION` Example

Example. **`UNION`**`(1, 7)`

```
1. Make set2 point at the smaller set and set1 at the larger set.
2. For each element e in set2:
       ptr[e] = set1
       Move node e to set1 and increment set1.size
```

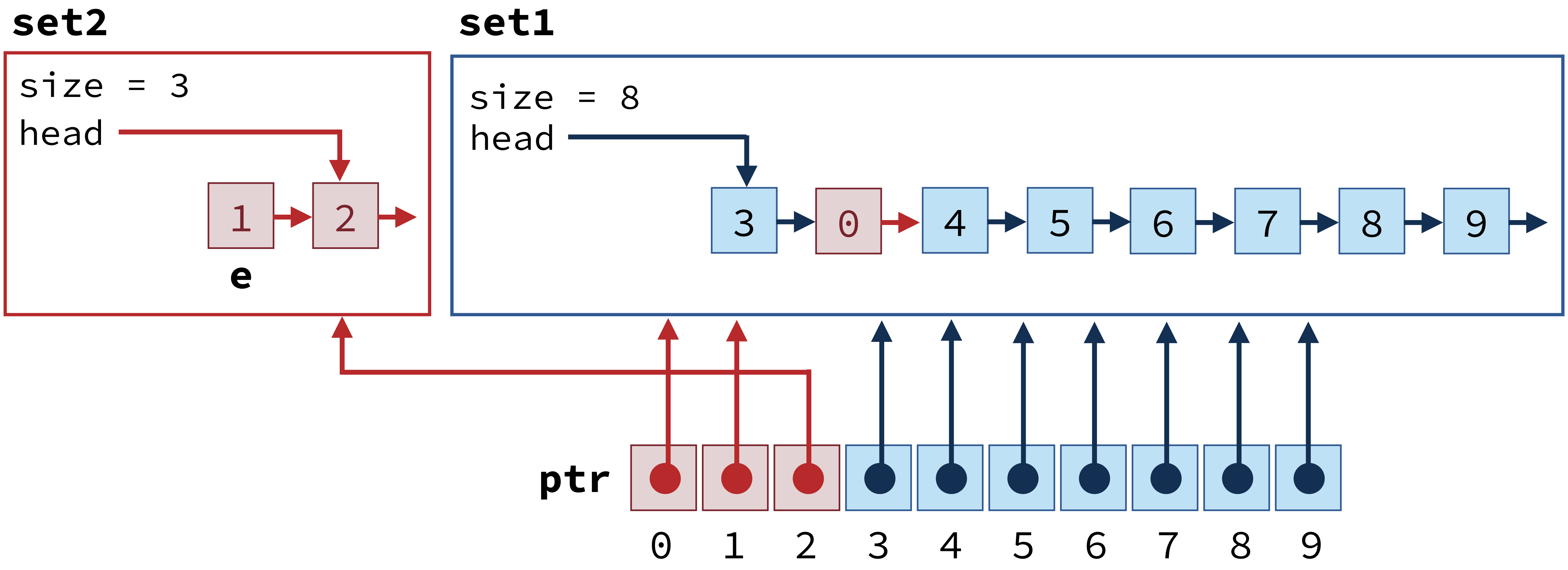


Link-List-Based Quick-Find: `UNION` Example

Example. **`UNION`**`(1, 7)`

1. Make `set2` point at the smaller set and `set1` at the larger set.
2. For each element `e` in `set2:`

```
    ptr[e] = set1
    Move node e to set1 and increment set1.size
```

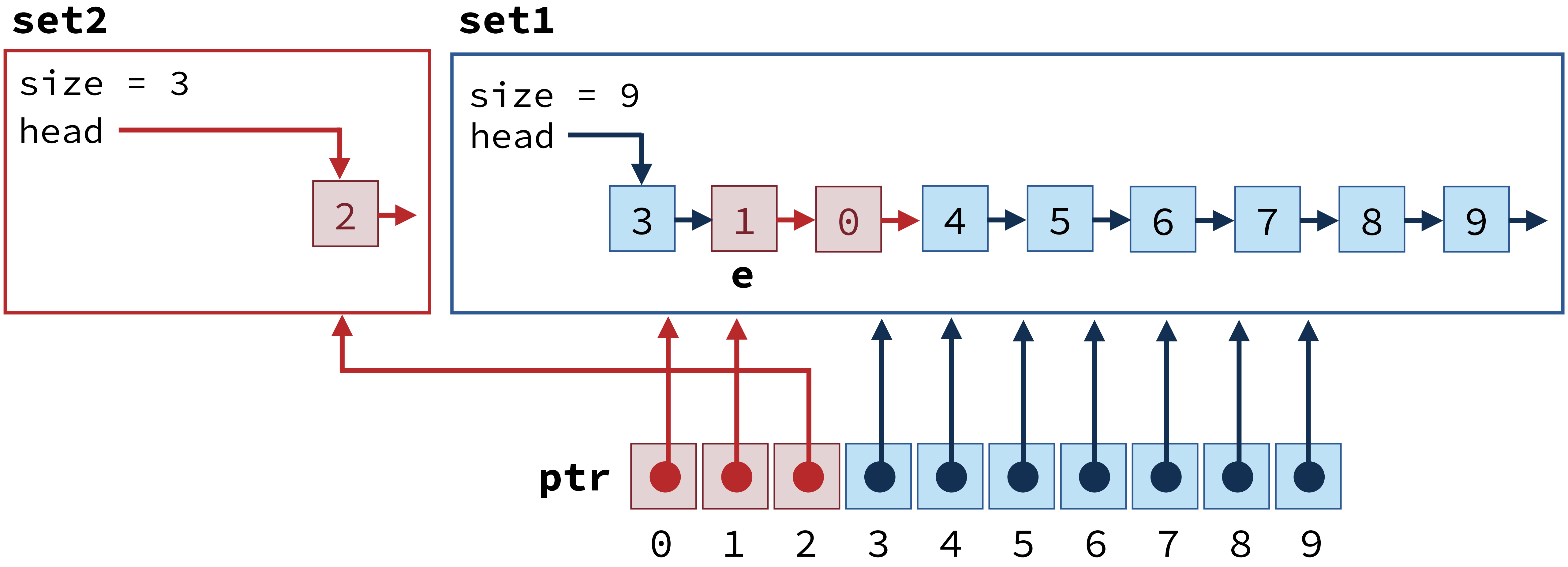


Link-List-Based Quick-Find: `UNION` Example

Example. **`UNION(1, 7)`**

1. Make `set2` point at the smaller set and `set1` at the larger set.
2. For each element `e` in `set2`:

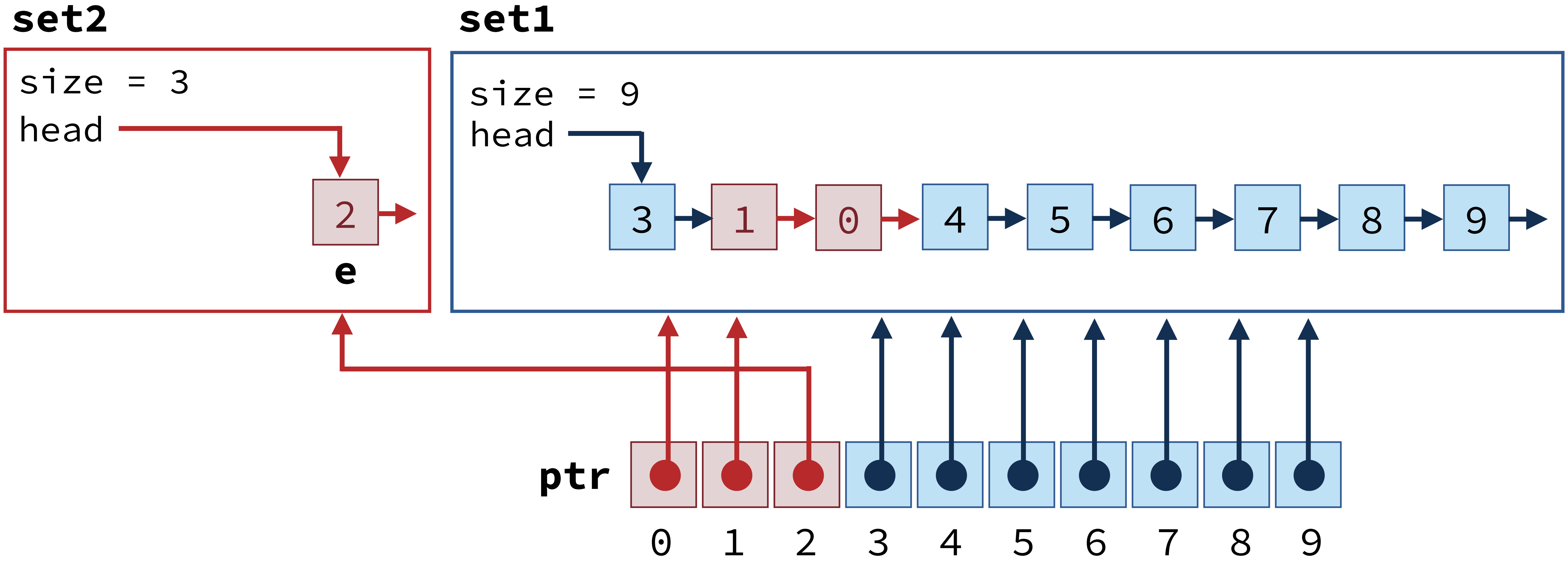
```
    ptr[e] = set1
    Move node e to set1 and increment set1.size
```



Link-List-Based Quick-Find: `UNION` Example

Example. **`UNION`**`(1, 7)`

```
1. Make set2 point at the smaller set and set1 at the larger set.
2. For each element e in set2:
       ptr[e] = set1
       Move node e to set1 and increment set1.size
```

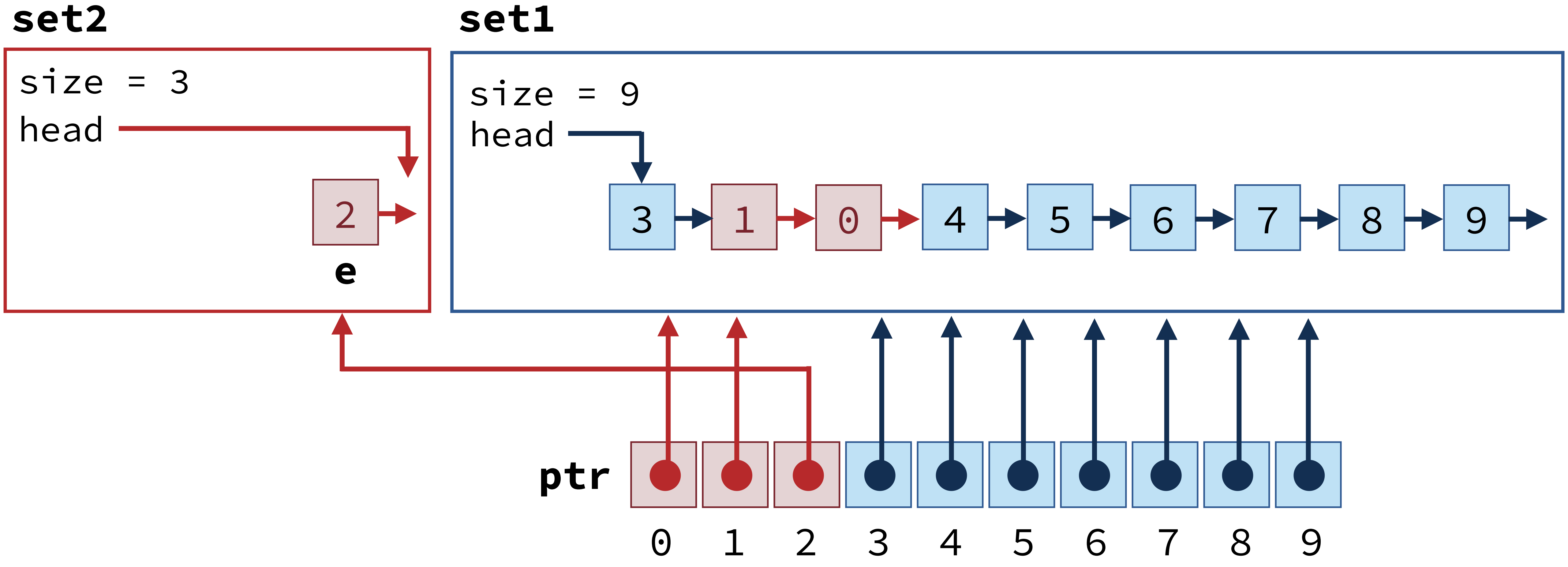


Link-List-Based Quick-Find: `UNION` Example

Example. **`UNION`**`(1, 7)`

1. Make `set2` point at the smaller set and `set1` at the larger set.
2. For each element `e` in `set2:`

```
    ptr[e] = set1
    Move node e to set1 and increment set1.size
```



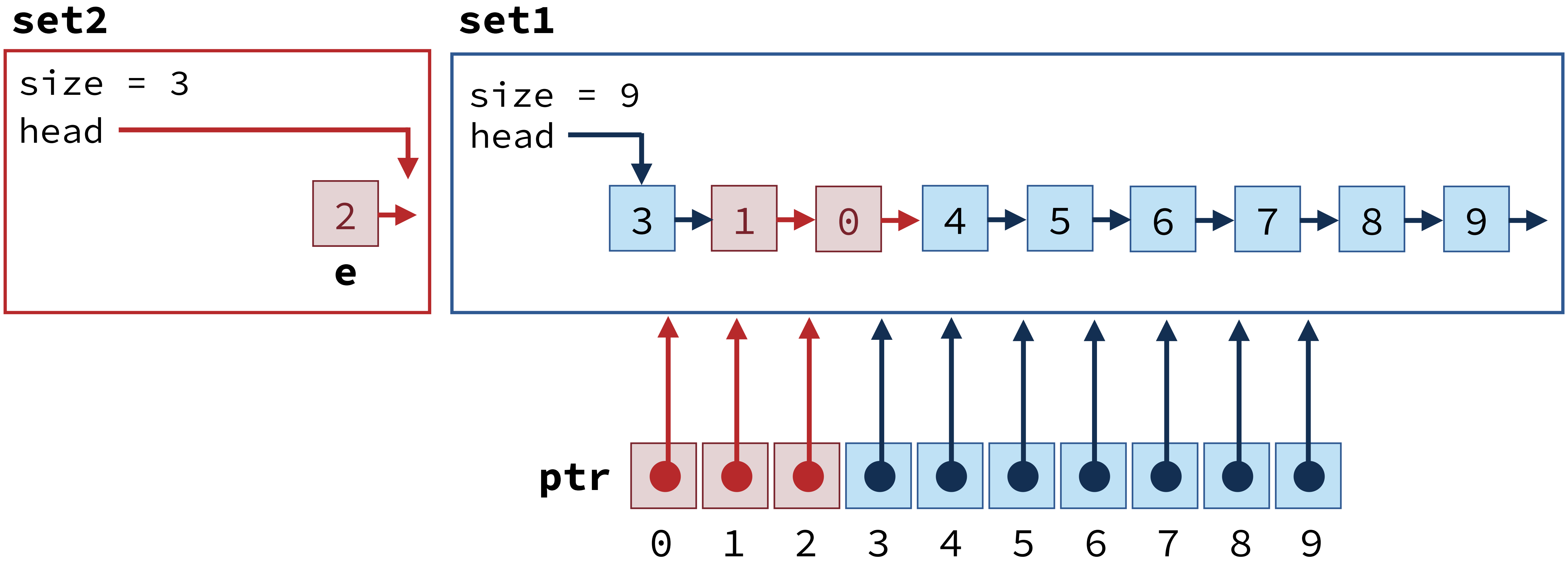
Link-List-Based Quick-Find: `UNION` Example

Example. **`UNION(1, 7)`**

1. Make `set2` point at the smaller set and `set1` at the larger set.
2. For each element `e` in `set2:`

   `ptr[e] = set1`

   Move node `e` to `set1` and increment `set1.size`



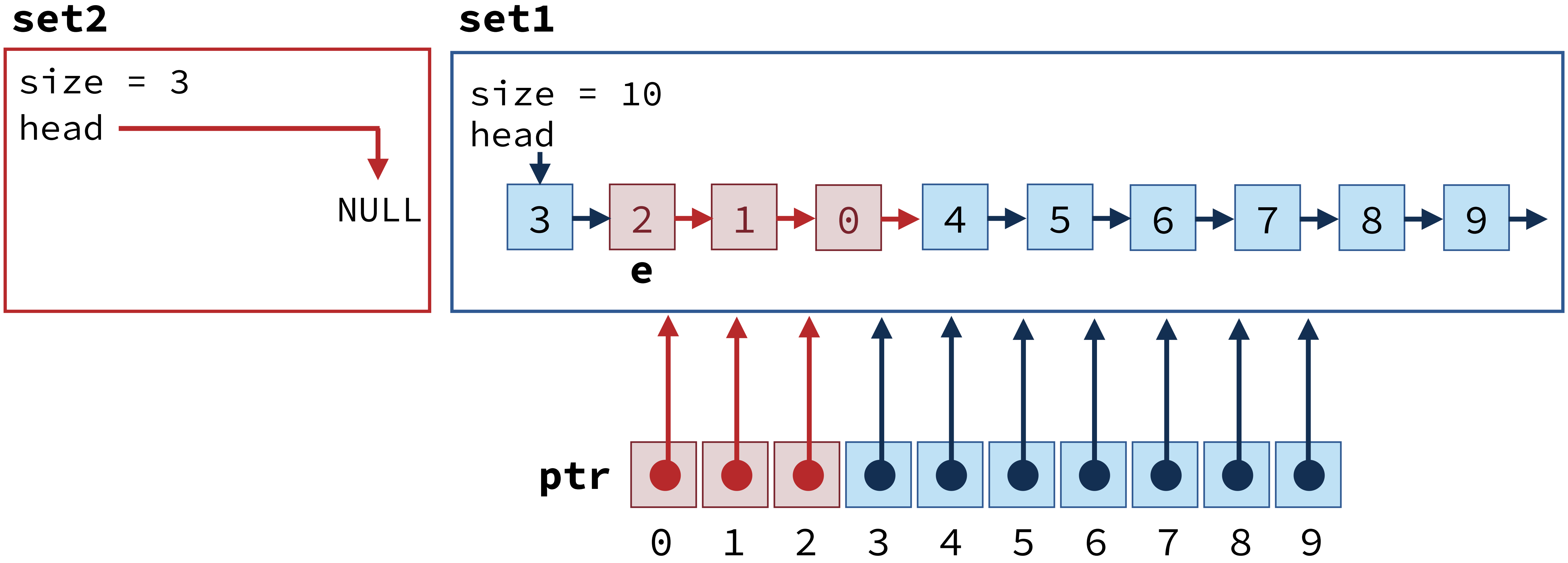
Link-List-Based Quick-Find: `UNION` Example

Example. **`UNION`**`(1, 7)`

1. Make `set2` point at the smaller set and `set1` at the larger set.
2. For each element `e` in `set2:`

   `ptr[e] = set1`

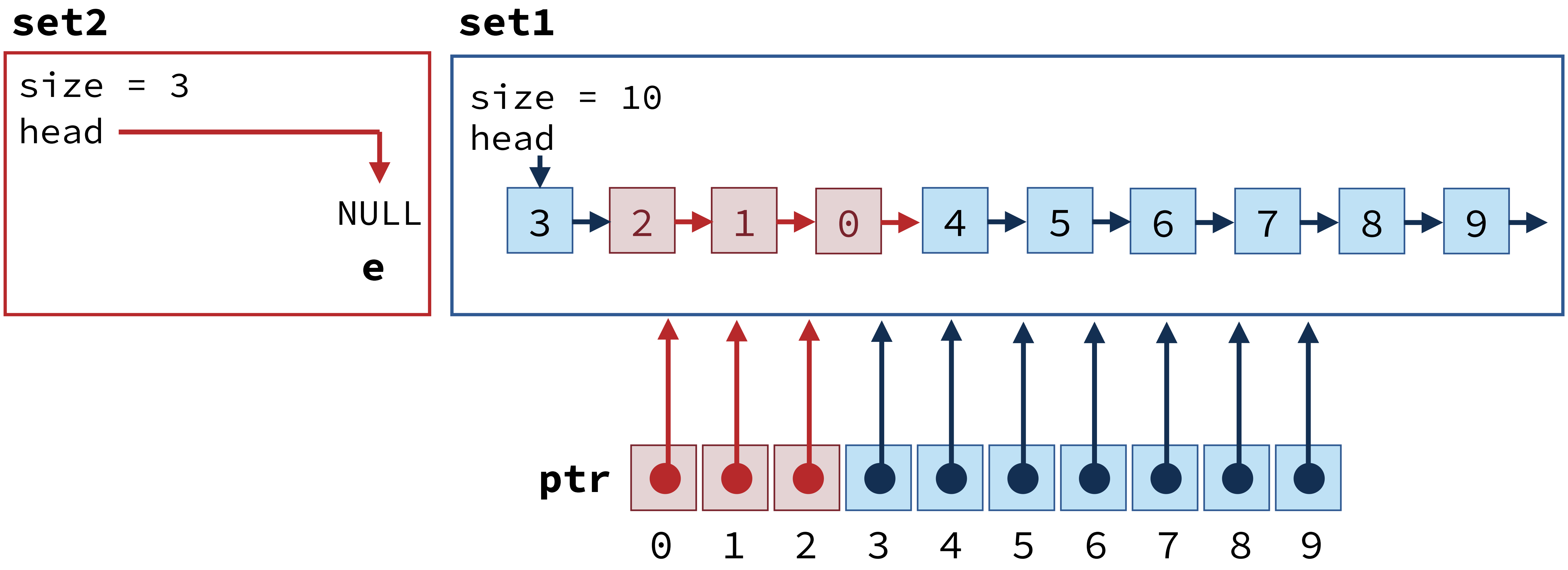
   Move node `e` to `set1` and increment `set1.size`



Link-List-Based Quick-Find: `UNION` Example

Example. **`UNION`**`(1, 7)`

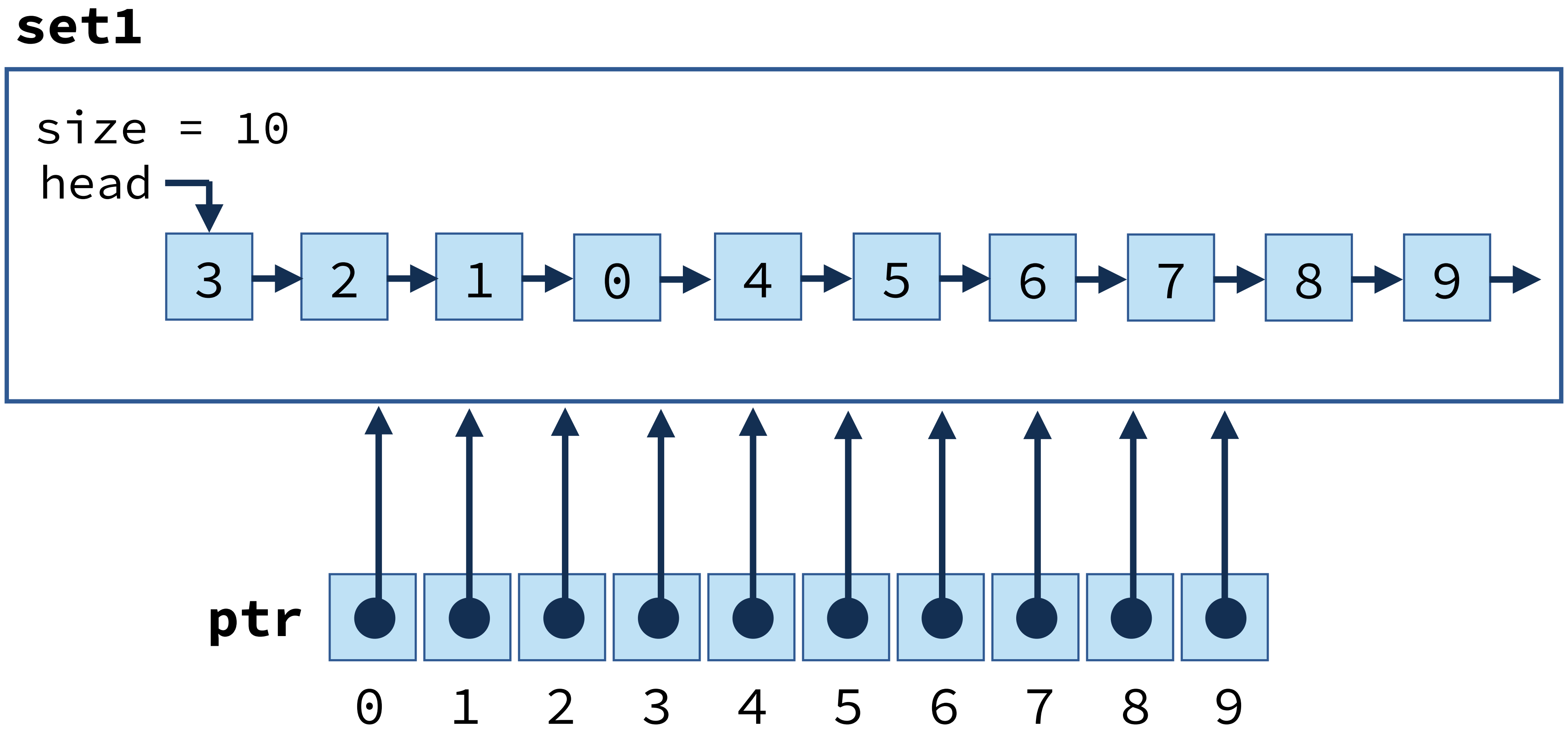
```
1. Make set2 point at the smaller set and set1 at the larger set.
2. For each element e in set2:
       ptr[e] = set1
       Move node e to set1 and increment set1.size
```



Link-List-Based Quick-Find: `UNION` Example

Example. **`UNION`**`(1, 7)`

```
1. Make set2 point at the smaller set and set1 at the larger set.
2. For each element e in set2:
       ptr[e] = set1
       Move node e to set1 and increment set1.size
3. Remove set2.
```



Link-List-Based Quick-Find: `UNION` Code

```
UNION(p, q)
  if (FIND(p) == FIND(q)) return
```

Link-List-Based Quick-Find: UNION Code

```
UNION(p, q)
  if (FIND(p) == FIND(q)) return

  LARGE= ptr[p], SMALL = ptr[q]
  if (LARGE.size < SMALL.size)
     SWAP(LARGE, SMALL)
```

Link-List-Based Quick-Find: UNION Code

UNION(p, q)

```
if (FIND(p) == FIND(q)) return

LARGE= ptr[p], SMALL = ptr[q]
if (LARGE.size < SMALL.size)
   SWAP(LARGE, SMALL)


// Add into LARGE every element from SMALL
e = SMALL.head
while (e != NULL)
    ptr[e.val]       = LARGE

    SMALL.head       = SMALL.head.next
    e.next           = LARGE.head.next
    LARGE.head.next = e
    LARGE.size     += 1

    e                = SMALL.head

delete SMALL
```

Link-List-Based Quick-Find: UNION Code

```
UNION(p, q)
  if (FIND(p) == FIND(q)) return

  LARGE= ptr[p], SMALL = ptr[q]
  if (LARGE.size < SMALL.size)
     SWAP(LARGE, SMALL)


  // Add into LARGE every element from SMALL
  e = SMALL.head
  while (e != NULL)
      ptr[e.val]        = LARGE

      SMALL.head        = SMALL.head.next
      e.next            = LARGE.head.next
      LARGE.head.next = e
      LARGE.size       += 1

      e                 = SMALL.head

  delete SMALL
```

Link-List-Based Quick-Find: UNION Code

UNION(p, q)

```
if (FIND(p) == FIND(q)) return

LARGE= ptr[p], SMALL = ptr[q]
if (LARGE.size < SMALL.size)
   SWAP(LARGE, SMALL)


// Add into LARGE every element from SMALL
e = SMALL.head
while (e != NULL)
    ptr[e.val]      = LARGE

    SMALL.head      = SMALL.head.next
    e.next          = LARGE.head.next
    LARGE.head.next = e
    LARGE.size     += 1

    e               = SMALL.head

delete SMALL
```

Worst Case Running Time.

FIND:
UNION:

Cost Model. Number of pointer updates or reads

Link-List-Based Quick-Find: **UNION** Code

```
UNION(p, q)

if (FIND(p) == FIND(q)) return

LARGE= ptr[p], SMALL = ptr[q]
if (LARGE.size < SMALL.size)
   SWAP(LARGE, SMALL)


// Add into LARGE every element from SMALL
e = SMALL.head
while (e != NULL)
    ptr[e.val]      = LARGE

    SMALL.head      = SMALL.head.next
    e.next          = LARGE.head.next
    LARGE.head.next = e
    LARGE.size     += 1

    e               = SMALL.head

delete SMALL
```

Worst Case Running Time.

FIND: $\Theta(1)$

UNION: $\Theta(\min(size_1, size_2))$

Cost Model. Number of pointer updates or reads

Quiz # 1

What is the total running time of a sequence of n `UNION` operations performed on n singleton sets?

Cost Model. Count the number of *pointer reads or updates* .

Choose the *best* answer.

A. $O(n^2)$

B. $O(n \log n)$

C. $O(n)$

D. I can see where this is going …

Quiz # 1

What is the total running time of a sequence of n **`UNION`** operations performed on n singleton sets?

Cost Model. Count the number of *pointer reads or updates* .

Choose the *best* answer.

A. $O(n^2)$ correct but too pessimistic!

😁 $O(n \log n)$ correct and tight bound! … *why*?

C. $O(n)$ incorrect! Showing a counterexample is easy.

D. I can see where this is going …

Link-List-Based Quick-Find: Running Time Proof

Observation 1. `ptr[e]` changes during a **`UNION`** operation only if **`e`** is in the smaller set.

Observation 2. If `ptr[e]` changes because of a **`UNION`** operation, **`e`** becomes in a set whose size is at least double the size it was in before the **`UNION`** operation.

Link-List-Based Quick-Find: Running Time Proof

Observation 1. `ptr[e]` changes during a **`UNION`** operation only if **`e`** is in the smaller set.

Observation 2. If `ptr[e]` changes because of a **`UNION`** operation, **`e`** becomes in a set whose size is at least double the size it was in before the **`UNION`** operation.

Proposition. `ptr[e]` cannot change more than $\log_2(n)$ times during a sequence of n **`UNION`** operations.

Link-List-Based Quick-Find: Running Time Proof

Observation 1. `ptr[e]` changes during a **`UNION`** operation only if **`e`** is in the smaller set.

Observation 2. If `ptr[e]` changes because of a **`UNION`** operation, **`e`** becomes in a set whose size is at least double the size it was in before the **`UNION`** operation.

Proposition. `ptr[e]` cannot change more than $\log_2(n)$ times during a sequence of n **`UNION`** operations.

Proof. Assume for the sake of contradiction that `ptr[e]` changed **more** than $\log_2(n)$ times during the n **`UNION`** operations.

Link-List-Based Quick-Find: Running Time Proof

Observation 1. `ptr[e]` changes during a **`UNION`** operation only if **`e`** is in the smaller set.

Observation 2. If `ptr[e]` changes because of a **`UNION`** operation, **`e`** becomes in a set whose size is at least double the size it was in before the **`UNION`** operation.

Proposition. `ptr[e]` cannot change more than $\log_2(n)$ times during a sequence of n **`UNION`** operations.

Proof. Assume for the sake of contradiction that `ptr[e]` changed **more** than $\log_2(n)$ times during the n **`UNION`** operations.

From observation 2, this means that the size of the set containing `e` at least doubled more than $\log_2(n)$ times, which implies that e is in a set whose size is $> 2^{\log_2(n)} > n$, which is impossible because there are only n elements.

Link-List-Based Quick-Find: Running Time Proof

Observation 1. `ptr[e]` changes during a **`UNION`** operation only if **`e`** is in the smaller set.

Observation 2. If `ptr[e]` changes because of a **`UNION`** operation, **`e`** becomes in a set whose size is at least double the size it was in before the **`UNION`** operation.

Proposition. `ptr[e]` cannot change more than $\log_2(n)$ times during a sequence of n **`UNION`** operations.

Proof. Assume for the sake of contradiction that `ptr[e]` changed **more** than $\log_2(n)$ times during the n **`UNION`** operations.

From observation 2, this means that the size of the set containing `e` at least doubled more than $\log_2(n)$ times, which implies that e is in a set whose size is $> 2^{\log_2(n)} > n$, which is impossible because there are only n elements.

Proposition. **`UNION`** runs in $O(\log n)$ amortized time.

Link-List-Based Quick-Find: Running Time Proof

Observation 1. `ptr[e]` changes during a **`UNION`** operation only if **`e`** is in the smaller set.

Observation 2. If `ptr[e]` changes because of a **`UNION`** operation, **`e`** becomes in a set whose size is at least double the size it was in before the **`UNION`** operation.

Proposition. `ptr[e]` cannot change more than $\log_2(n)$ times during a sequence of n **`UNION`** operations.

Proof. Assume for the sake of contradiction that `ptr[e]` changed **more** than $\log_2(n)$ times during the n **`UNION`** operations.

From observation 2, this means that the size of the set containing `e` at least doubled more than $\log_2(n)$ times, which implies that e is in a set whose size is $> 2^{\log_2(n)} > n$, which is impossible because there are only n elements.

Proposition. **`UNION`** runs in $O(\log n)$ amortized time.

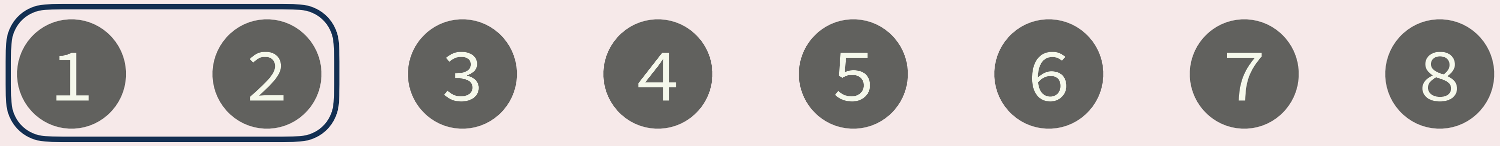
Proof. In a sequence of n **`UNION`** operations, no pointer can change more than $\log_2(n)$ times in total. Hence, the total is $O(n \log n)$ and each **`UNION`** operation costs $O(\log_2 n)$ on average.

Quiz # 2

Provide a sequence of `UNION` operations that leads to a running time of $\Theta(n)$ and another one that leads to a running time of $\Theta(n \log n)$.

Quiz # 2

Provide a sequence of `UNION` operations that leads to a running time of $\Theta(n)$ and another one that leads to a running time of $\Theta(n \log n)$.



Quiz # 2

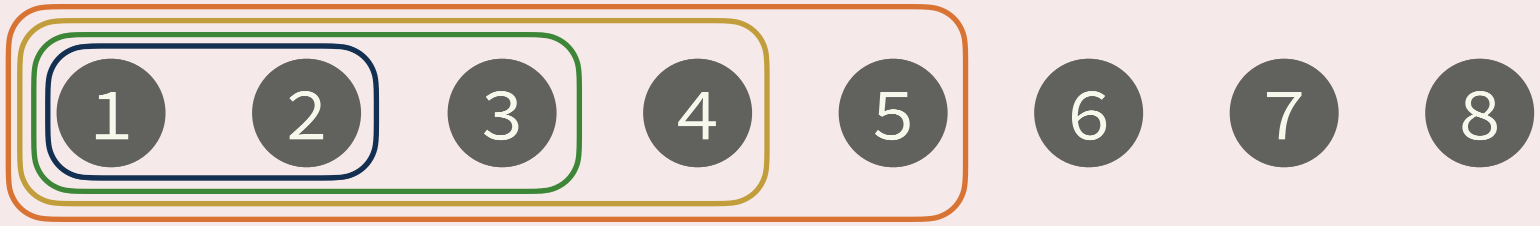
Provide a sequence of **`UNION`** operations that leads to a running time of $\Theta(n)$ and another one that leads to a running time of $\Theta(n \log n)$.

Quiz # 2

Provide a sequence of `UNION` operations that leads to a running time of $\Theta(n)$ and another one that leads to a running time of $\Theta(n \log n)$.

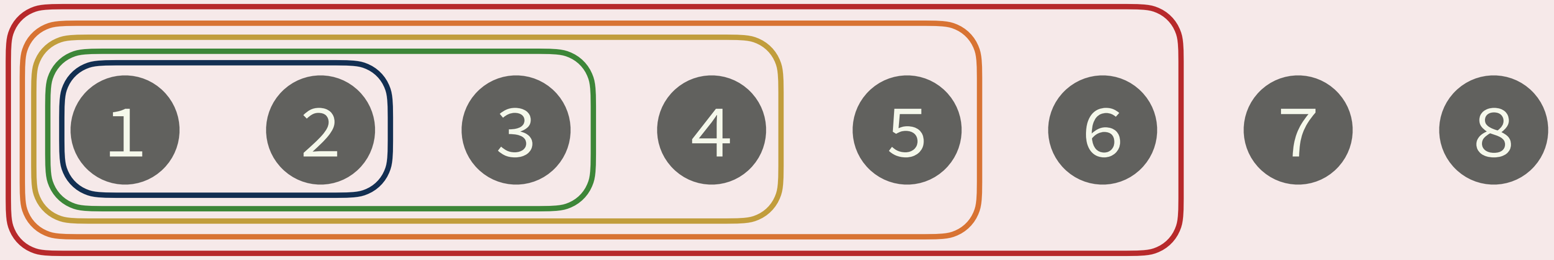
Quiz # 2

Provide a sequence of **`UNION`** operations that leads to a running time of $\Theta(n)$ and another one that leads to a running time of $\Theta(n \log n)$.



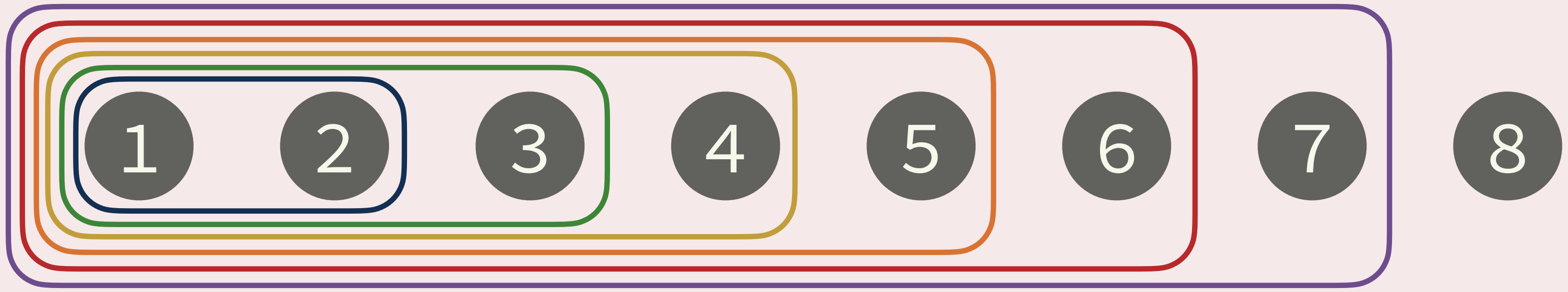
Quiz # 2

Provide a sequence of **`UNION`** operations that leads to a running time of $\Theta(n)$ and another one that leads to a running time of $\Theta(n \log n)$.



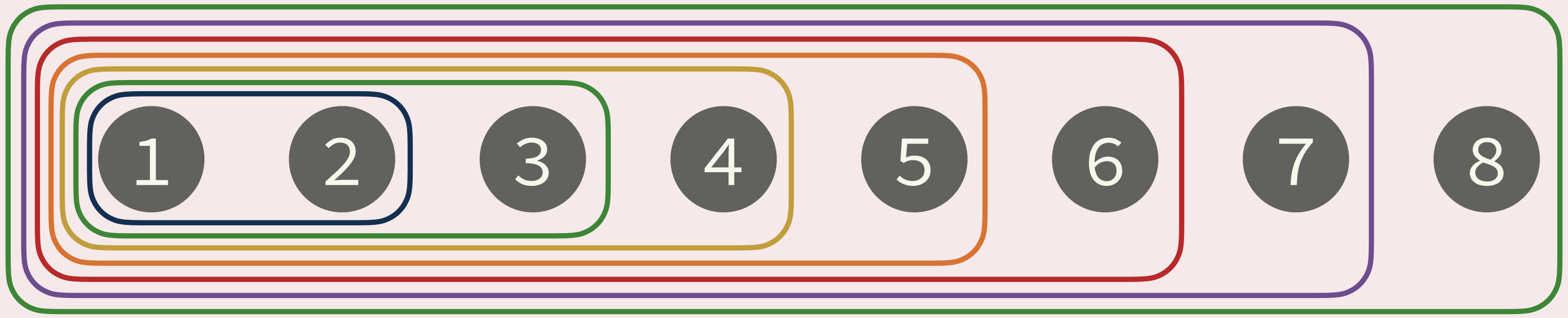
Quiz # 2

Provide a sequence of **`UNION`** operations that leads to a running time of $\Theta(n)$ and another one that leads to a running time of $\Theta(n \log n)$.



Quiz # 2

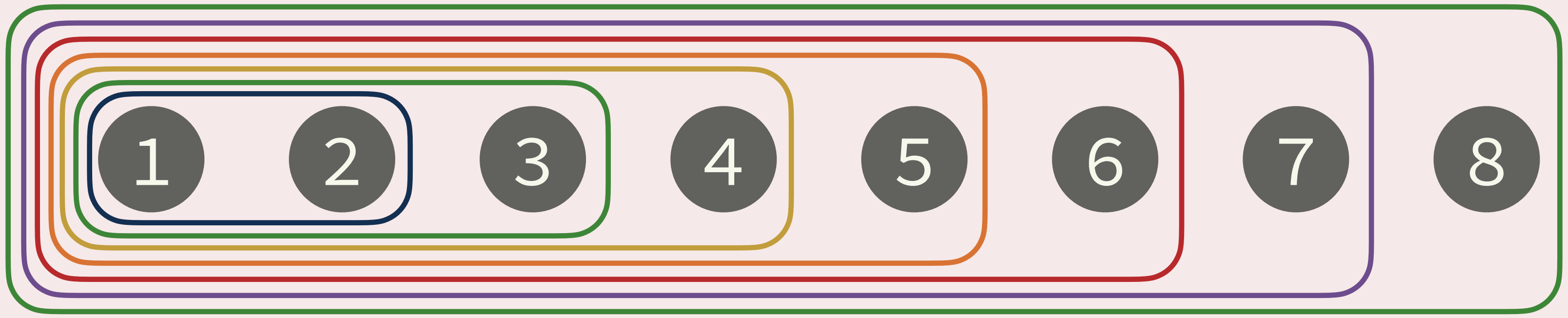
Provide a sequence of **`UNION`** operations that leads to a running time of $\Theta(n)$ and another one that leads to a running time of $\Theta(n \log n)$.



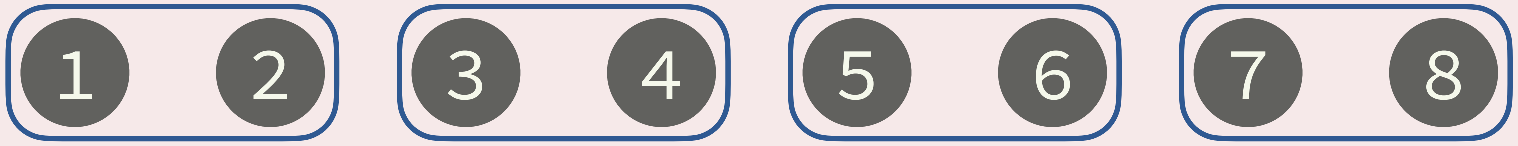
$$(n-1) \times 1 = O(n)$$

Quiz # 2

Provide a sequence of `UNION` operations that leads to a running time of $\Theta(n)$ and another one that leads to a running time of $\Theta(n \log n)$.



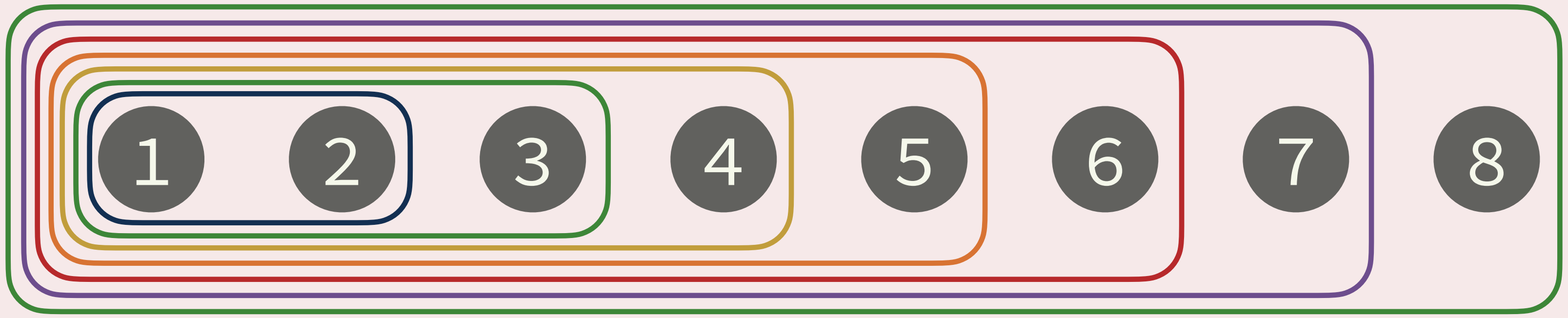
$(n - 1) \times 1 = O(n)$



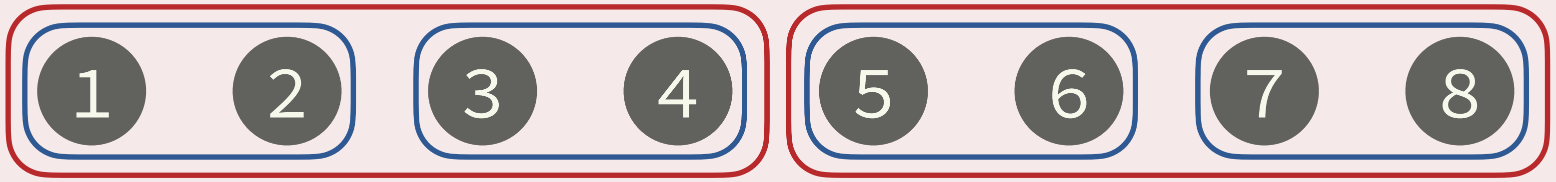
$$\frac{n}{2^1} \times 2^0$$

Quiz # 2

Provide a sequence of **`UNION`** operations that leads to a running time of $\Theta(n)$ and another one that leads to a running time of $\Theta(n \log n)$.



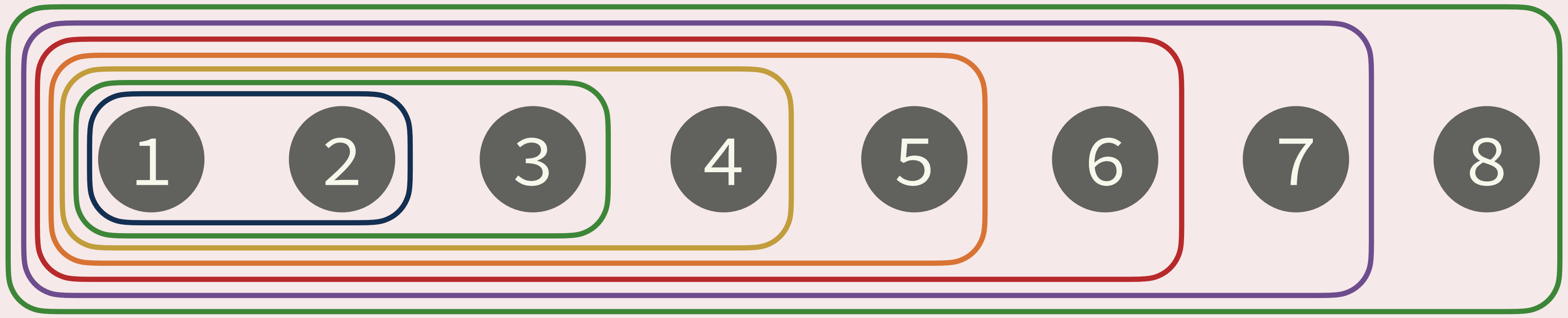
$$(n-1) \times 1 = O(n)$$



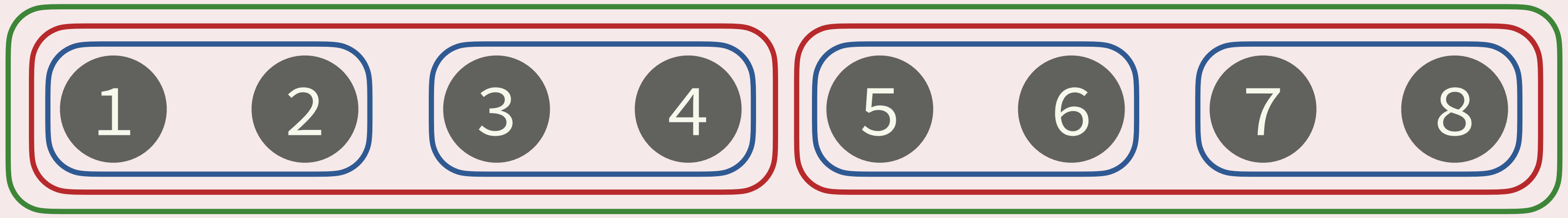
$$\frac{n}{2^1} \times 2^0 \quad + \quad \frac{n}{2^2} \times 2^1$$

Quiz # 2

Provide a sequence of **`UNION`** operations that leads to a running time of $\Theta(n)$ and another one that leads to a running time of $\Theta(n \log n)$.



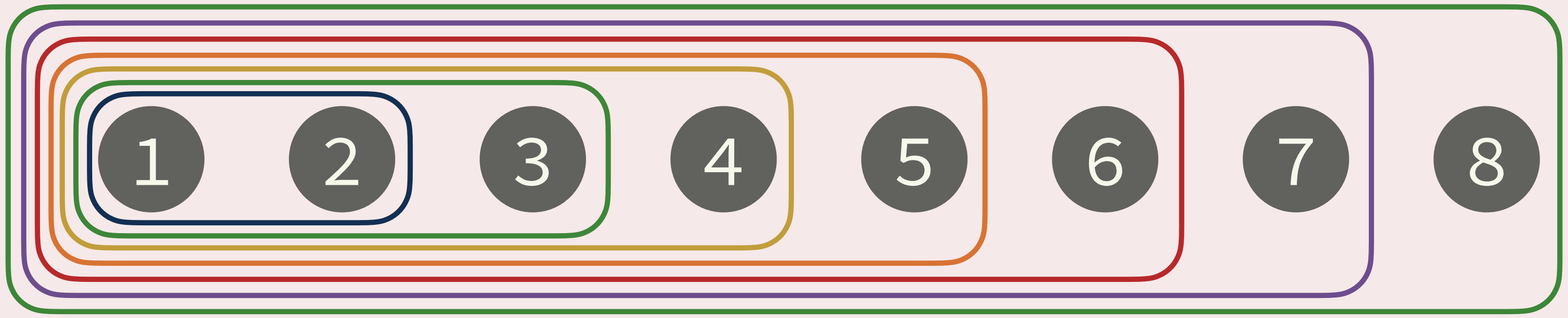
$$(n-1) \times 1 = O(n)$$



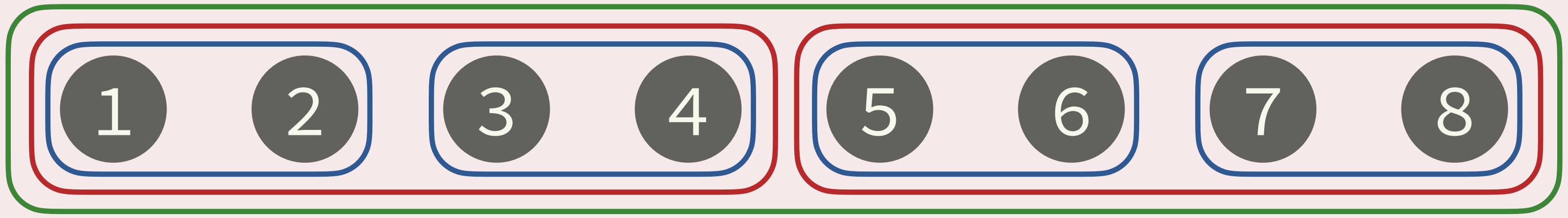
$$\frac{n}{2^1} \times 2^0 \quad + \quad \frac{n}{2^2} \times 2^1 \quad + \quad \frac{n}{2^3} \times 2^2$$

Quiz # 2

Provide a sequence of **`UNION`** operations that leads to a running time of $\Theta(n)$ and another one that leads to a running time of $\Theta(n \log n)$.



$$(n-1) \times 1 = O(n)$$



$$\frac{n}{2^1} \times 2^0 + \frac{n}{2^2} \times 2^1 + \frac{n}{2^3} \times 2^2 = \frac{n}{2} \times \log_2(n)$$

Quick-Find: Running Time Summary

	Array-based Quick-Find	Linked-List-based Quick-Find
`FIND`	$O(1)$	$O(1)$
`UNION`	$O(n)$	$O(n)$
Sequence of n `UNION` operations:		

Quick-Find: Running Time Summary

	Array-based Quick-Find	Linked-List-based Quick-Find
`FIND`	$O(1)$	$O(1)$
`UNION`	$O(n)$	$O(n)$
Sequence of n `UNION` operations:	$O(n^2)$	$O(n \log n)$

Quick-Find: Running Time Summary

	Array-based Quick-Find	Linked-List-based Quick-Find
`FIND`	$O(1)$	$O(1)$
`UNION`	$O(n)$	$O(n)$
Sequence of n **`UNION`** operations:	$O(n^2)$	$O(n \log n)$

🤔

Can we do better?

Improvement Attempt 2: Quick-Union

Idea. Each set has a *canonical element* (root, representative or leader for the set.)

- **`FIND`**`(p)` Returns the root of the set of `p`.
- **`UNION`**`(p, q)` Change the root of the set of `q` to be the result of **`FIND`**`(p)`.

Initially. Every element `e` is in a singleton set whose root is `e` itself.

Example.

`parent``[]`

0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9

- **`UNION`**`(0, 3)`
- **`UNION`**`(3, 1)`
- **`UNION`**`(4, 1)`
- **`UNION`**`(7, 5)`
- **`UNION`**`(1, 5)`
- **`UNION`**`(2, 7)`
- **`UNION`**`(0, 9)`
- **`UNION`**`(6, 1)`

Improvement Attempt 2: Quick-Union

Idea. Each set has a *canonical element* (root, representative or leader for the set.)

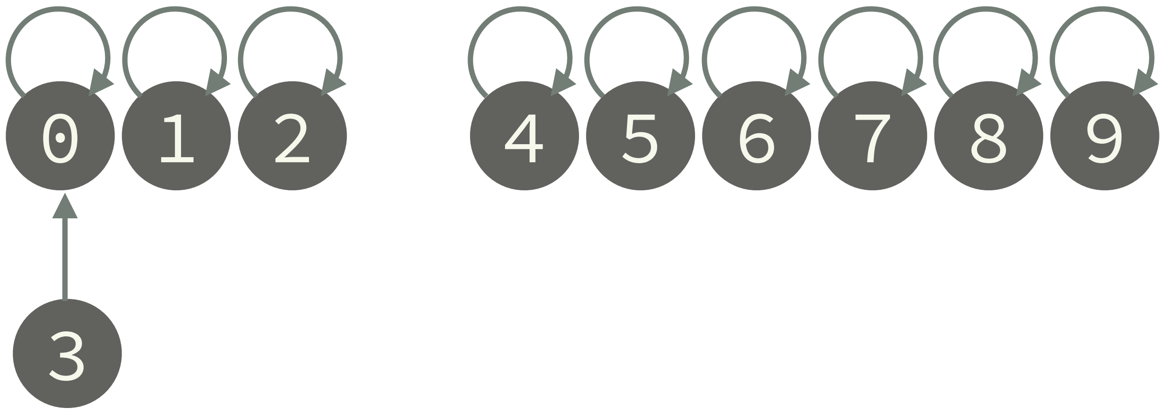
- **`FIND`**`(p)` Returns the root of the set of `p`.
- **`UNION`**`(p, q)` Change the root of the set of `q` to be the result of **`FIND`**`(p)`.

Initially. Every element `e` is in a singleton set whose root is `e` itself.

Example.

`parent``[]`

0	1	2	**0**	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9



- **`UNION`**`(0, 3)`
- `UNION(3, 1)`
- `UNION(4, 1)`
- `UNION(7, 5)`
- `UNION(1, 5)`
- `UNION(2, 7)`
- `UNION(0, 9)`
- `UNION(6, 1)`

Improvement Attempt 2: Quick-Union

Idea. Each set has a *canonical element* (root, representative or leader for the set.)

- `FIND(p)` Returns the root of the set of `p`.
- `UNION(p, q)` Change the root of the set of `q` to be the result of `FIND(p)`.

Initially. Every element `e` is in a singleton set whose root is `e` itself.

Example.

parent[]

0	0	2	0	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9

- `UNION(0, 3)`
- `UNION(3, 1)`
- `UNION(4, 1)`
- `UNION(7, 5)`
- `UNION(1, 5)`
- `UNION(2, 7)`
- `UNION(0, 9)`
- `UNION(6, 1)`

Improvement Attempt 2: Quick-Union

Idea. Each set has a *canonical element* (root, representative or leader for the set.)

- **`FIND`**`(p)` Returns the root of the set of `p`.
- **`UNION`**`(p, q)` Change the root of the set of `q` to be the result of **`FIND`**`(p)`.

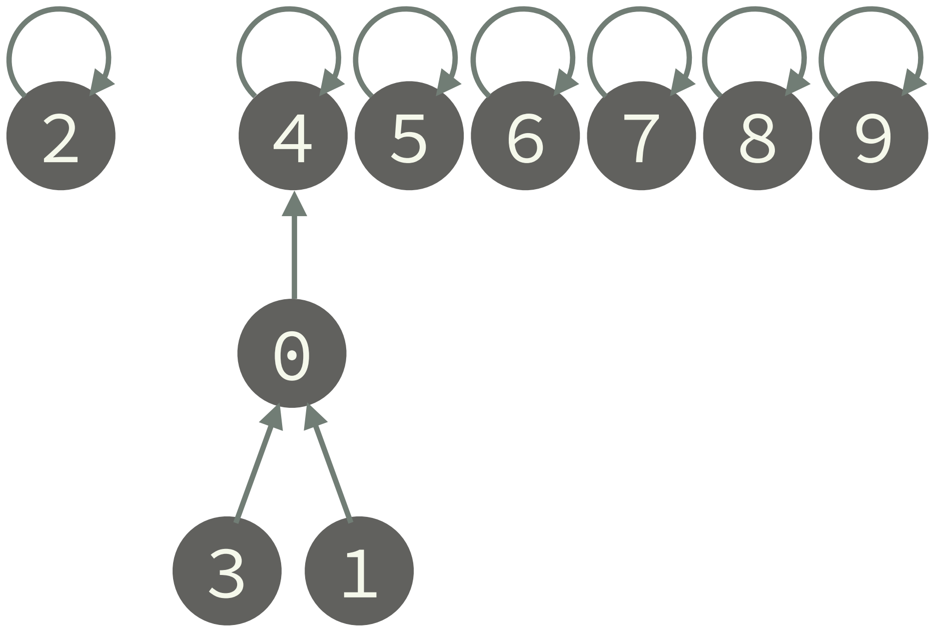
Initially. Every element `e` is in a singleton set whose root is `e` itself.

Example.

`parent[]`

4	0	2	0	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9

- `UNION(0, 3)`
- `UNION(3, 1)`
- **`UNION(4, 1)`**
- `UNION(7, 5)`
- `UNION(1, 5)`
- `UNION(2, 7)`
- `UNION(0, 9)`
- `UNION(6, 1)`



Improvement Attempt 2: Quick-Union

Idea. Each set has a *canonical element* (root, representative or leader for the set.)

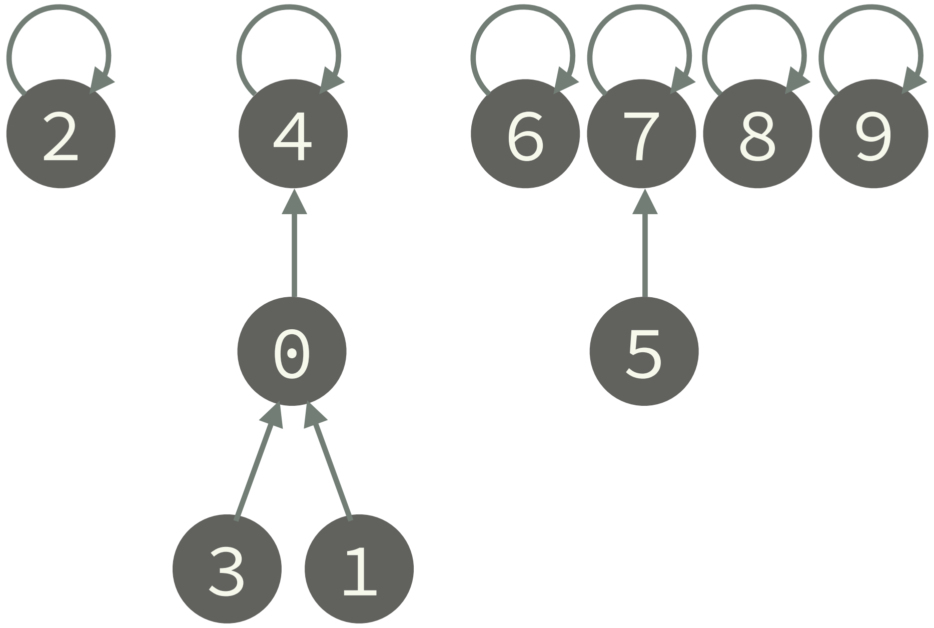
- **FIND**(p) Returns the root of the set of p.
- **UNION**(p, q) Change the root of the set of q to be the result of **FIND**(p).

Initially. Every element e is in a singleton set whose root is e itself.

Example.

parent[]

4	0	2	0	4	7	6	7	8	9
0	1	2	3	4	5	6	7	8	9



- UNION(0, 3)
- UNION(3, 1)
- UNION(4, 1)
- **UNION(7, 5)**
- UNION(1, 5)
- UNION(2, 7)
- UNION(0, 9)
- UNION(6, 1)

Improvement Attempt 2: Quick-Union

Idea. Each set has a *canonical element* (root, representative or leader for the set.)

- **`FIND`**`(p)` Returns the root of the set of `p`.
- **`UNION`**`(p, q)` Change the root of the set of `q` to be the result of **`FIND`**`(p)`.

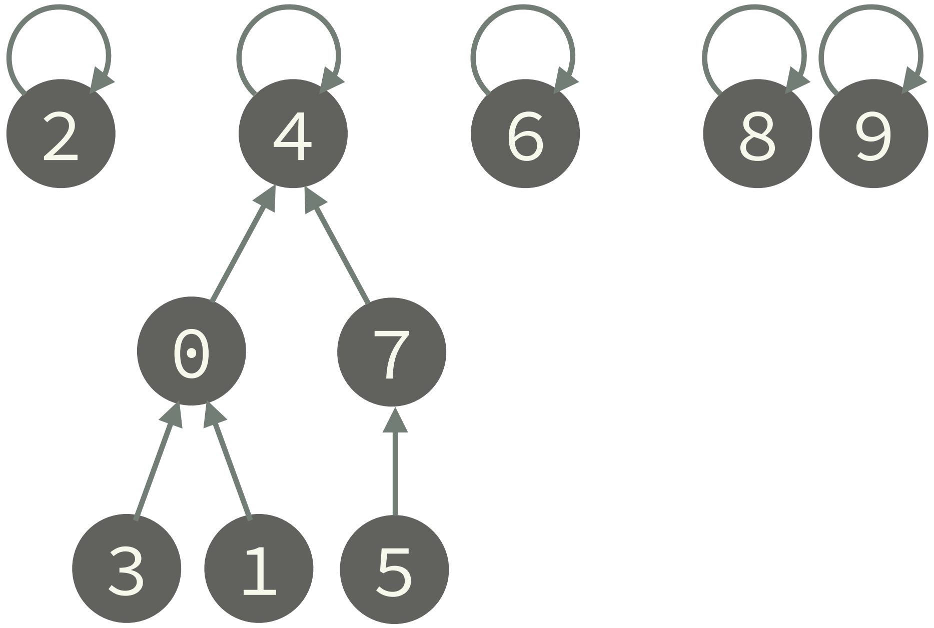
Initially. Every element `e` is in a singleton set whose root is `e` itself.

Example.

`parent``[]`

4	0	2	0	4	7	6	**4**	8	9
0	1	2	3	4	5	6	7	8	9

- `UNION(0, 3)`
- `UNION(3, 1)`
- `UNION(4, 1)`
- `UNION(7, 5)`
- **`UNION(1, 5)`**
- `UNION(2, 7)`
- `UNION(0, 9)`
- `UNION(6, 1)`



Improvement Attempt 2: Quick-Union

Idea. Each set has a *canonical element* (root, representative or leader for the set.)

- `FIND(p)` Returns the root of the set of `p`.
- `UNION(p, q)` Change the root of the set of `q` to be the result of `FIND(p)`.

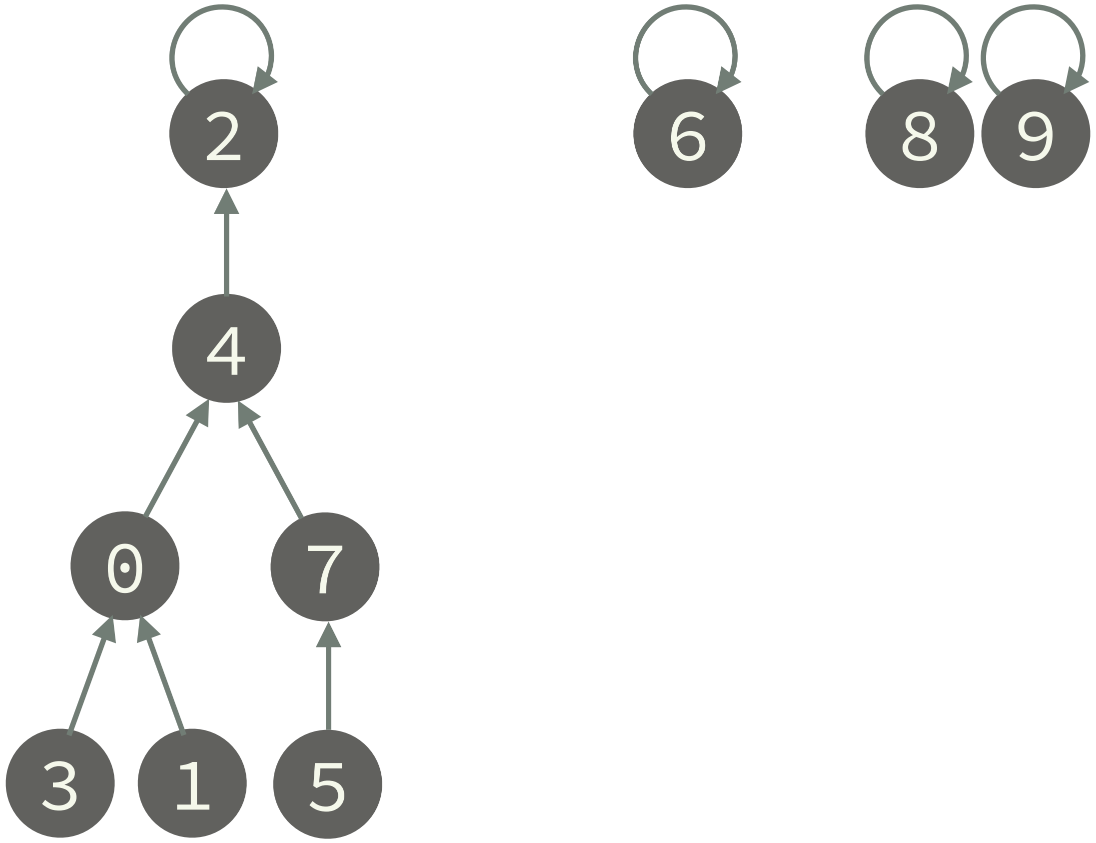
Initially. Every element `e` is in a singleton set whose root is `e` itself.

Example.

parent[]

4	0	2	0	2	7	6	4	8	9
0	1	2	3	4	5	6	7	8	9

- `UNION(0, 3)`
- `UNION(3, 1)`
- `UNION(4, 1)`
- `UNION(7, 5)`
- `UNION(1, 5)`
- **`UNION(2, 7)`**
- `UNION(0, 9)`
- `UNION(6, 1)`



Improvement Attempt 2: Quick-Union

Idea. Each set has a *canonical element* (root, representative or leader for the set.)

- **`FIND`**`(p)` Returns the root of the set of `p`.
- **`UNION`**`(p, q)` Change the root of the set of `q` to be the result of **`FIND`**`(p)`.

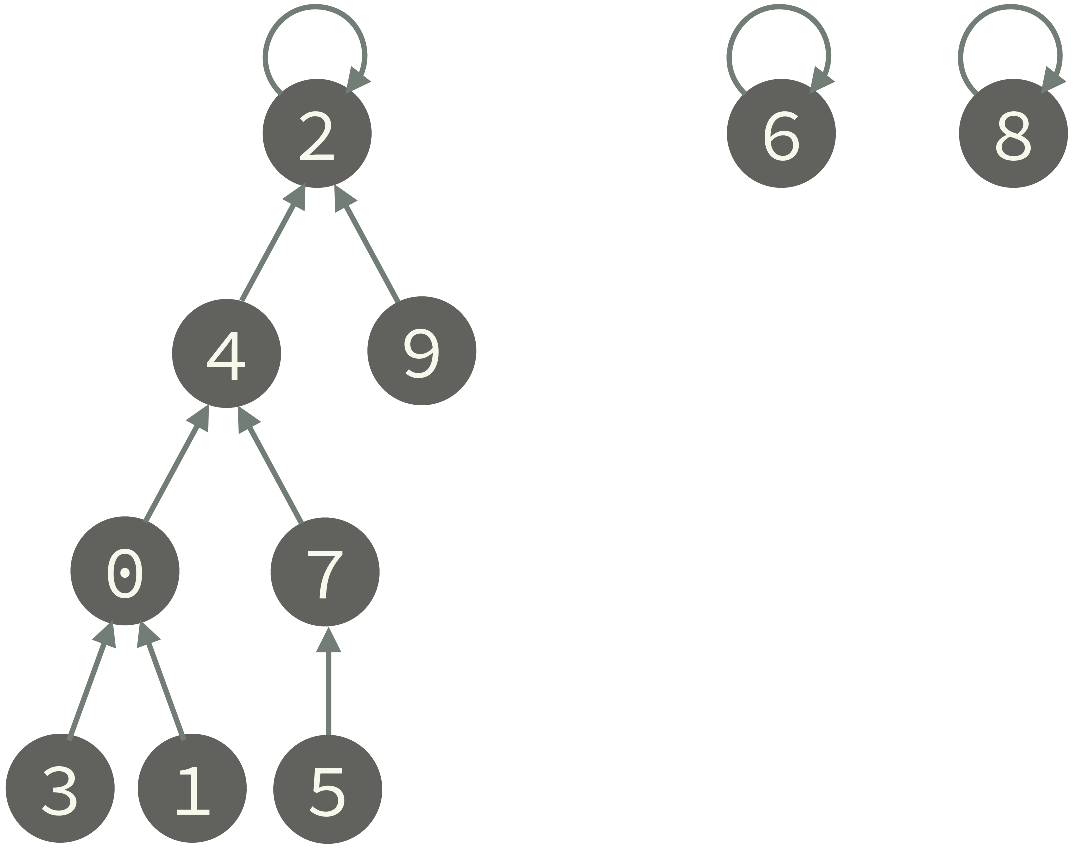
Initially. Every element `e` is in a singleton set whose root is `e` itself.

Example.

`parent``[]`

4	0	2	0	2	7	6	4	8	**2**
0	1	2	3	4	5	6	7	8	9

- `UNION(0, 3)`
- `UNION(3, 1)`
- `UNION(4, 1)`
- `UNION(7, 5)`
- `UNION(1, 5)`
- `UNION(2, 7)`
- **`UNION(0, 9)`**
- `UNION(6, 1)`



Improvement Attempt 2: Quick-Union

Idea. Each set has a *canonical element* (root, representative or leader for the set.)

- **`FIND`**`(p)` Returns the root of the set of `p`.
- **`UNION`**`(p, q)` Change the root of the set of `q` to be the result of **`FIND`**`(p)`.

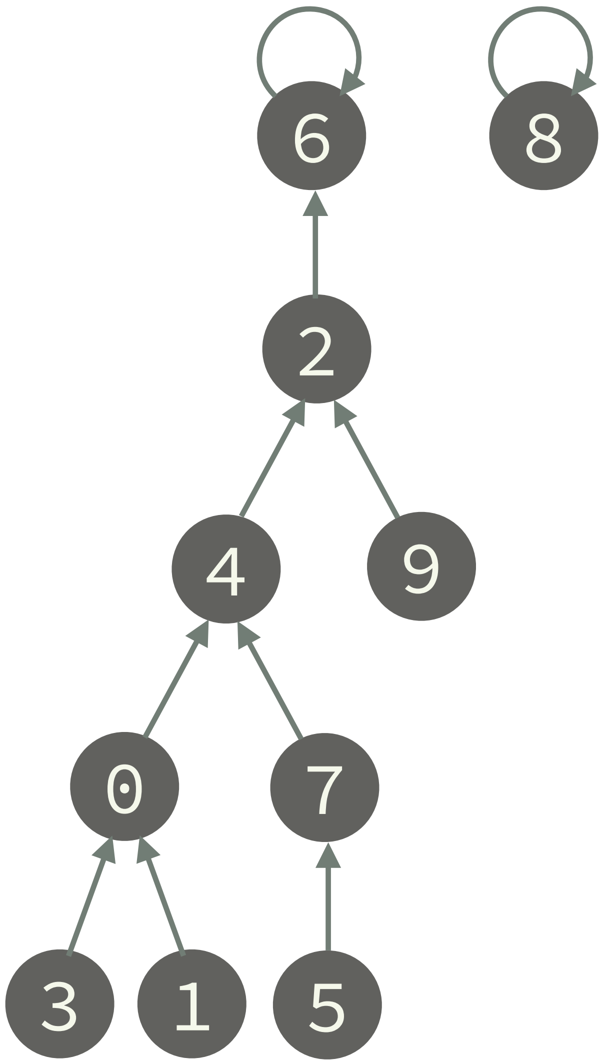
Initially. Every element `e` is in a singleton set whose root is `e` itself.

Example.

`parent``[]`

4	6	2	0	2	7	6	4	8	2
0	1	2	3	4	5	6	7	8	9

- `UNION(0, 3)`
- `UNION(3, 1)`
- `UNION(4, 1)`
- `UNION(7, 5)`
- `UNION(1, 5)`
- `UNION(2, 7)`
- `UNION(0, 9)`
- **`UNION`**`(6, 1)`



Improvement Attempt 2: Quick-Union

Idea. Each set has a *canonical element* (root, representative or leader for the set.)

- `FIND(p)` Returns the root of the set of `p`.
- `UNION(p, q)` Change the root of the set of `q` to be the result of `FIND(p)`.

Initially. Every element `e` is in a singleton set whose root is `e` itself.

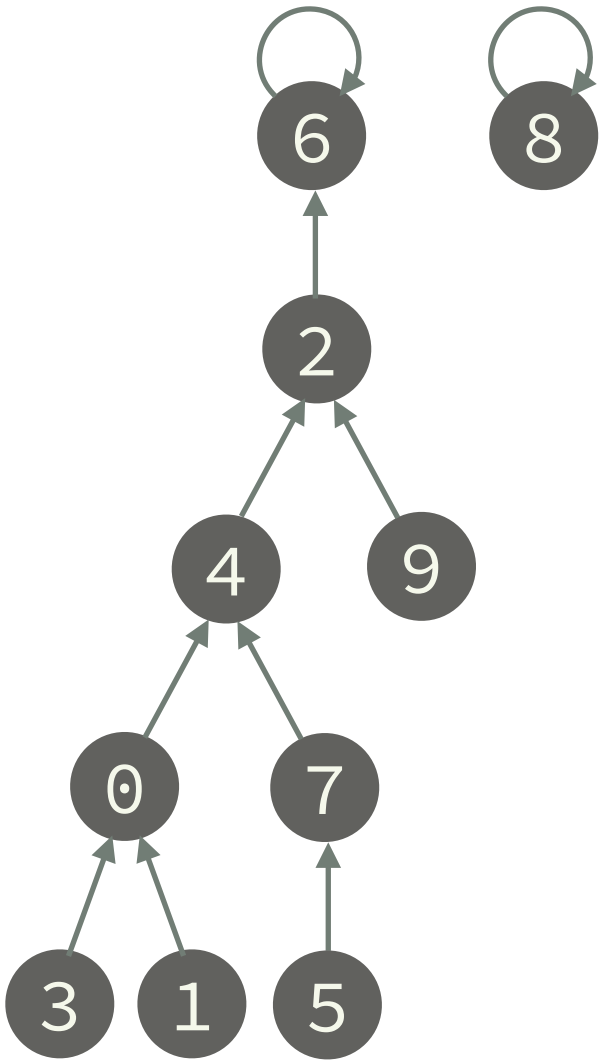
Example.

parent[]

4	6	2	0	2	7	6	4	8	2
0	1	2	3	4	5	6	7	8	9

UNION(p, q)

```
root1 = FIND(p)
root2 = FIND(q)

if (root1 != root2)
   parent[root2] = root1
```



Improvement Attempt 2: Quick-Union

Idea. Each set has a *canonical element* (root, representative or leader for the set.)

- **`FIND`**`(p)` Returns the root of the set of `p`.
- **`UNION`**`(p, q)` Change the root of the set of `q` to be the result of **`FIND`**`(p)`.

Initially. Every element `e` is in a singleton set whose root is `e` itself.

Example.

parent[]

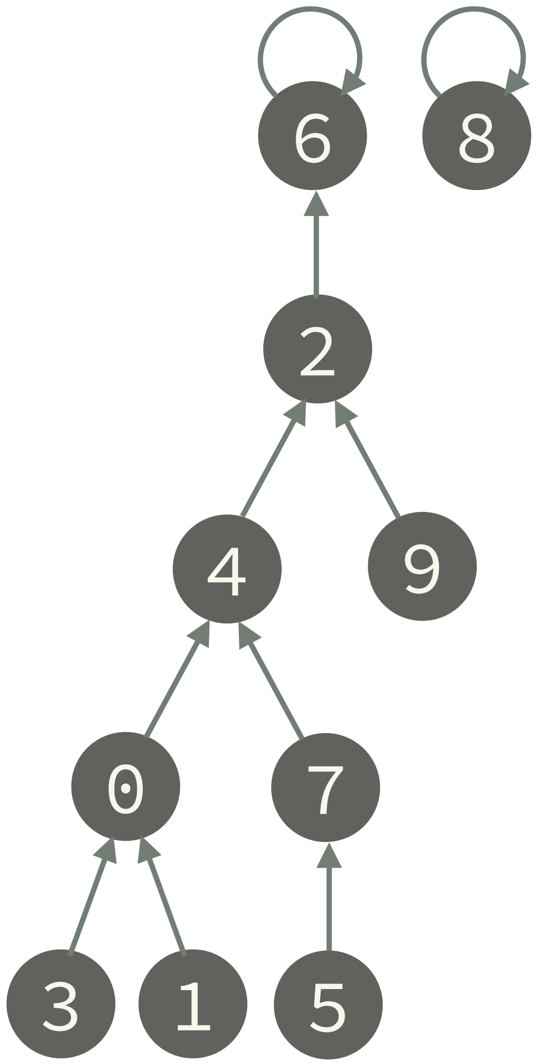
4	6	2	0	2	7	6	4	8	2
0	1	2	3	4	5	6	7	8	9

UNION(p, q)

```
root1 = FIND(p)
root2 = FIND(q)

if (root1 != root2)
   parent[root2] = root1
```

FIND(p)

```
while (parent[p] != p)
    p = parent[p]

return p
```



Improvement Attempt 2: Quick-Union

Idea. Each set has a *canonical element* (root, representative or leader for the set.)

- **`FIND`**`(p)` Returns the root of the set of `p`.
- **`UNION`**`(p, q)` Change the root of the set of `q` to be the result of **`FIND`**`(p)`.

Initially. Every element `e` is in a singleton set whose root is `e` itself.

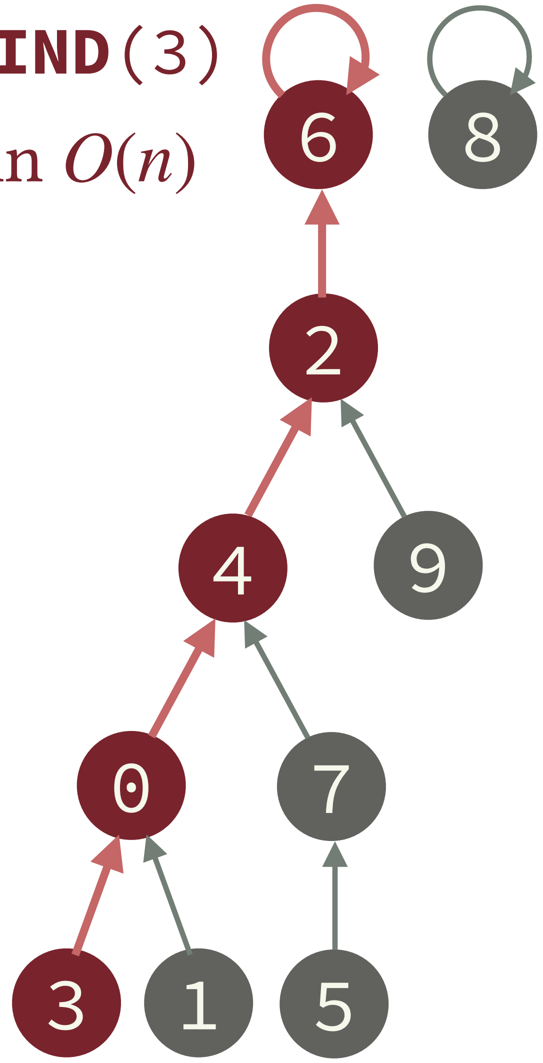
Example.

`parent``[]`

0	1	2	3	4	5	6	7	8	9
4	6	2	0	2	7	6	4	8	2

`FIND``(3)`

Runs in $O(n)$



`UNION``(p, q)`

```
root1 = FIND(p)
root2 = FIND(q)

if (root1 != root2)
   parent[root2] = root1
```

`FIND``(p)`

```
while (parent[p] != p)
    p = parent[p]

return p
```

Improvement Attempt 2: Quick-Union

Idea. Each set has a *canonical element* (root, representative or leader for the set.)

- **FIND**(p) Returns the root of the set of p.
- **UNION**(p, q) Change the root of the set of q to be the result of **FIND**(p).

Initially. Every element `e` is in a singleton set whose root is `e` itself.

Example.

parent[]

4	6	2	0	2	7	6	4	8	2
0	1	2	3	4	5	6	7	8	9

UNION(p, q)

```
root1 = FIND(p)
root2 = FIND(q)

if (root1 != root2)
   parent[root2] = root1
```

FIND(p)

```
while (parent[p] != p)
    p = parent[p]

return p
```

Worst Case Running Time.

FIND: $O(n)$
UNION: $O(n)$

Cost Model. Number of array accesses.

Improvement Attempt 2.1: Weighted Quick-Union (by size)

Idea. Attach the smaller tree to the larger tree.

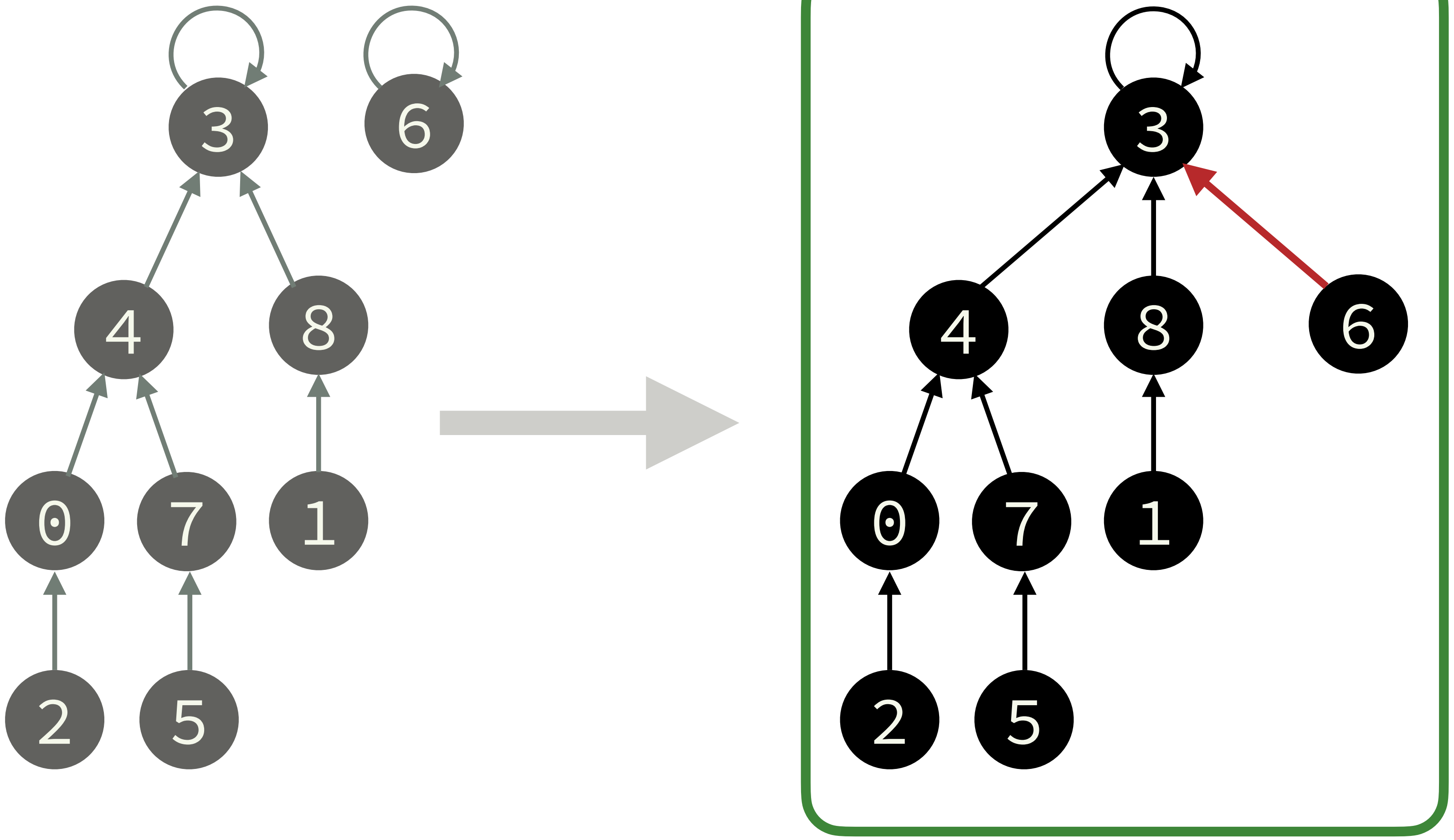
Rationale. Reduce the likelihood of long chains.

Improvement Attempt 2.1: Weighted Quick-Union (by size)

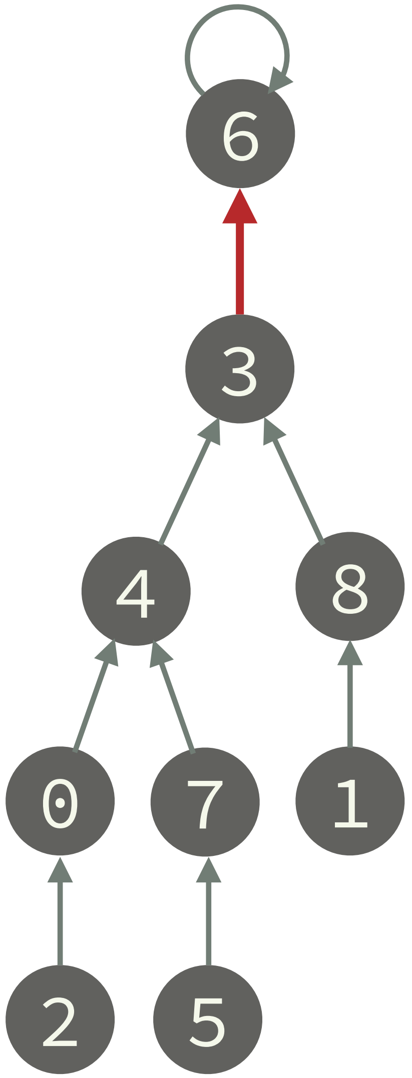
Idea. Attach the smaller tree to the larger tree.

Rationale. Reduce the likelihood of long chains.

Example. **`UNION`**`(6, 2)` attaches 6 ⟶ 3 not 3 ⟶ 6.



did not increase the height



increased the height

Improvement Attempt 2.1: Weighted Quick-Union (by size)

Idea. Attach the smaller tree to the larger tree.

Rationale. Reduce the likelihood of long chains.

Modification. Add an array to record the size of the tree rooted at each element.

parent	0	1	2	3	4	5	6	7	8
size	1	1	1	1	1	1	1	1	1
	0	1	2	3	4	5	6	7	8

Improvement Attempt 2.1: Weighted Quick-Union (by size)

Idea. Attach the smaller tree to the larger tree.

Rationale. Reduce the likelihood of long chains.

Modification. Add an array to record the size of the tree rooted at each element.

parent	0	1	2	3	4	5	6	7	8
size	1	1	1	1	1	1	1	1	1
	0	1	2	3	4	5	6	7	8

UNION(p, q)

```
r1 = FIND(p)
r2 = FIND(q)

if (r1 == r2) return
```

Improvement Attempt 2.1: Weighted Quick-Union (by size)

Idea. Attach the smaller tree to the larger tree.

Rationale. Reduce the likelihood of long chains.

Modification. Add an array to record the size of the tree rooted at each element.

parent	0	1	2	3	4	5	6	7	8
size	1	1	1	1	1	1	1	1	1
	0	1	2	3	4	5	6	7	8

UNION(p, q)

```
r1 = FIND(p)
r2 = FIND(q)

if (r1 == r2) return

if (size[r1] < size[r2])
    SWAP(r1, r2)
```

Improvement Attempt 2.1: Weighted Quick-Union (by size)

Idea. Attach the smaller tree to the larger tree.

Rationale. Reduce the likelihood of long chains.

Modification. Add an array to record the size of the tree rooted at each element.

parent	0	1	2	3	4	5	6	7	8
size	1	1	1	1	1	1	1	1	1
	0	1	2	3	4	5	6	7	8

```
UNION(p, q)

r1 = FIND(p)
r2 = FIND(q)

if (r1 == r2) return

if (size[r1] < size[r2])
    SWAP(r1, r2)

parent[r2] = r1
size[r1]  += size[r2]
```

```
UNION(0, 1)
UNION(2, 1)
UNION(4, 3)
UNION(2, 3)
```

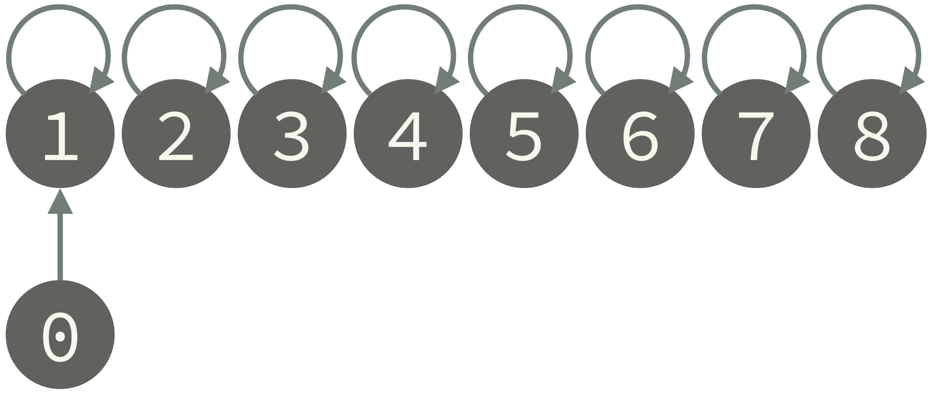
Improvement Attempt 2.1: Weighted Quick-Union (by size)

Idea. Attach the smaller tree to the larger tree.

Rationale. Reduce the likelihood of long chains.

Modification. Add an array to record the size of the tree rooted at each element.

parent	1	1	2	3	4	5	6	7	8
size	1	2	1	1	1	1	1	1	1
	0	1	2	3	4	5	6	7	8



UNION(p, q)

```
r1 = FIND(p)
r2 = FIND(q)

if (r1 == r2) return

if (size[r1] < size[r2])
    SWAP(r1, r2)

parent[r2] = r1
size[r1]  += size[r2]
```

```
UNION(0, 1)
UNION(2, 1)
UNION(4, 3)
UNION(2, 3)
```

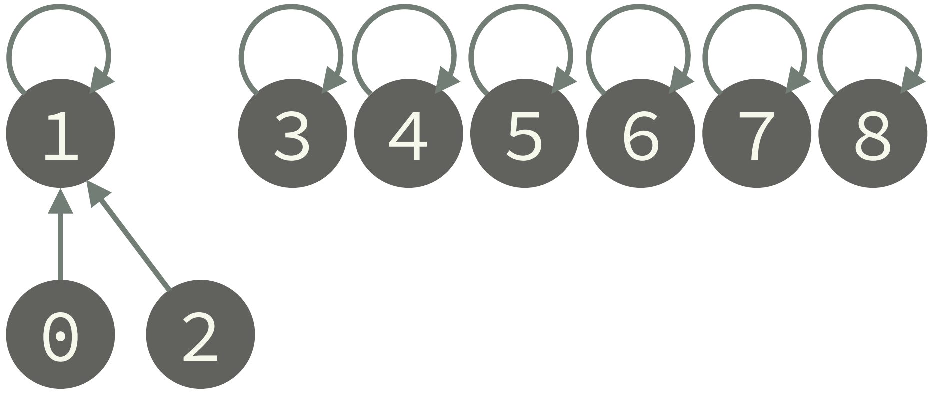
Improvement Attempt 2.1: Weighted Quick-Union (by size)

Idea. Attach the smaller tree to the larger tree.

Rationale. Reduce the likelihood of long chains.

Modification. Add an array to record the size of the tree rooted at each element.

	0	1	2	3	4	5	6	7	8
parent	1	1	**1**	3	4	5	6	7	8
size	1	**3**	1	1	1	1	1	1	1



UNION(p, q)

```
r1 = FIND(p)
r2 = FIND(q)

if (r1 == r2) return

if (size[r1] < size[r2])
    SWAP(r1, r2)

parent[r2] = r1
size[r1]  += size[r2]
```

```
UNION(0, 1)
UNION(2, 1)
UNION(4, 3)
UNION(2, 3)
```

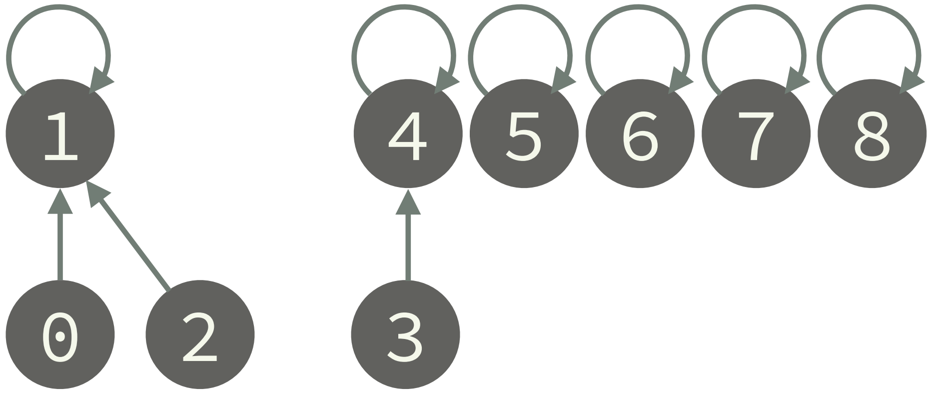
Improvement Attempt 2.1: Weighted Quick-Union (by size)

Idea. Attach the smaller tree to the larger tree.

Rationale. Reduce the likelihood of long chains.

Modification. Add an array to record the size of the tree rooted at each element.

parent	1	1	1	**4**	4	5	6	7	8
size	1	3	1	1	**2**	1	1	1	1
	0	1	2	3	4	5	6	7	8



```
UNION(p, q)

r1 = FIND(p)
r2 = FIND(q)

if (r1 == r2) return

if (size[r1] < size[r2])
    SWAP(r1, r2)

parent[r2] = r1
size[r1]  += size[r2]
```

UNION(0, 1)
UNION(2, 1)
UNION(4, 3)
UNION(2, 3)

Improvement Attempt 2.1: Weighted Quick-Union (by size)

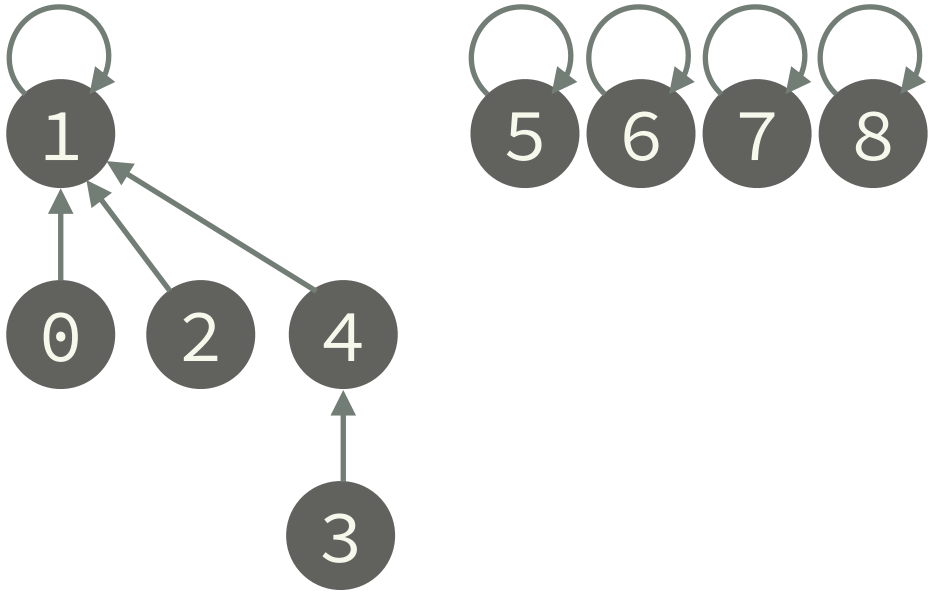
Idea. Attach the smaller tree to the larger tree.

Rationale. Reduce the likelihood of long chains.

Modification. Add an array to record the size of the tree rooted at each element.

	0	1	2	3	4	5	6	7	8
parent	1	1	1	4	**1**	5	6	7	8
size	1	**5**	1	1	2	1	1	1	1

```
UNION(p, q)

r1 = FIND(p)
r2 = FIND(q)

if (r1 == r2) return

if (size[r1] < size[r2])
    SWAP(r1, r2)

parent[r2] = r1
size[r1]  += size[r2]
```



```
UNION(0, 1)
UNION(2, 1)
UNION(4, 3)
UNION(2, 3)
```

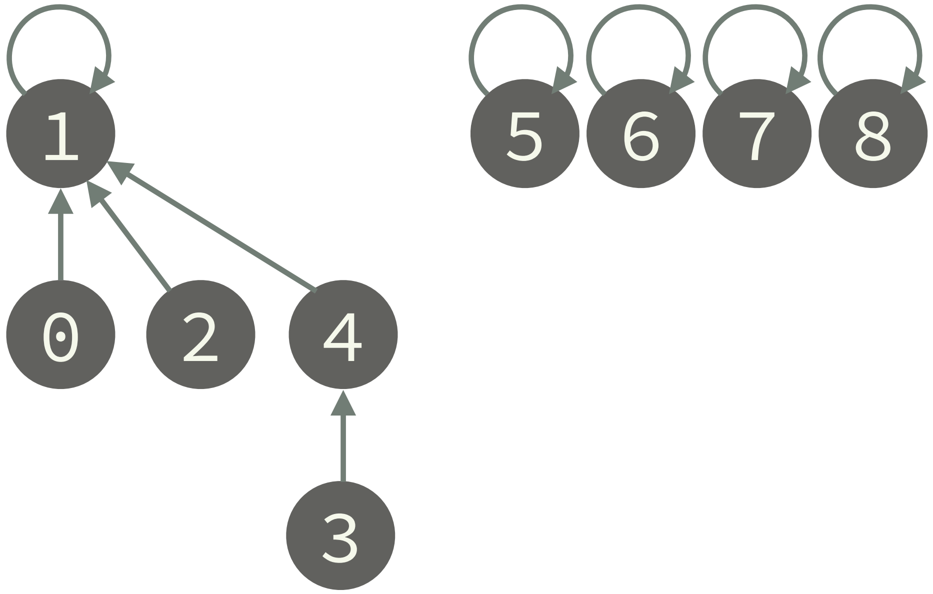
Improvement Attempt 2.1: Weighted Quick-Union (by size)

Idea. Attach the smaller tree to the larger tree.

Rationale. Reduce the likelihood of long chains.

Modification. Add an array to record the size of the tree rooted at each element.

parent	1	1	1	4	**1**	5	6	7	8
size	1	**5**	1	1	2	1	1	1	1
	0	1	2	3	4	5	6	7	8



UNION(p, q)

```
r1 = FIND(p)
r2 = FIND(q)

if (r1 == r2) return

if (size[r1] < size[r2])
    SWAP(r1, r2)

parent[r2] = r1
size[r1]  += size[r2]
```

Worst Case Running Time.

FIND:
UNION:

Cost Model. Number of array accesses.

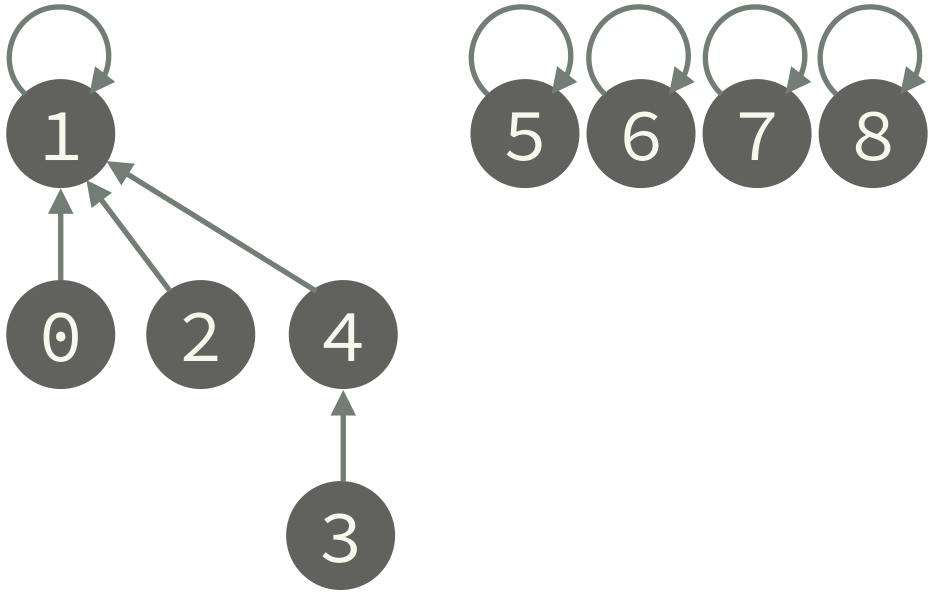
Improvement Attempt 2.1: Weighted Quick-Union (by size)

Idea. Attach the smaller tree to the larger tree.

Rationale. Reduce the likelihood of long chains.

Modification. Add an array to record the size of the tree rooted at each element.

parent	1	1	1	4	1	5	6	7	8
size	1	5	1	1	2	1	1	1	1
	0	1	2	3	4	5	6	7	8



UNION(p, q)

```
r1 = FIND(p)
r2 = FIND(q)

if (r1 == r2) return
if (size[r1] < size[r2])
    SWAP(r1, r2)

parent[r2] = r1
size[r1]  += size[r2]
```

Worst Case Running Time.

FIND: $O(\log n)$

UNION: $O(\log n)$ 😁

Cost Model. Number of array accesses.

Weighted Quick-Union (by size)**:** Running Time

Proposition. The depth of any node in a tree of size $\boldsymbol{K}$ built using a sequence of weighted quick-union operations (by size) is $\leq \log_2 K$.

Weighted Quick-Union (by size): Running Time

Proposition. The depth of any node in a tree of size ***K*** built using a sequence of weighted quick-union operations (by size) is $\leq \log_2 K$.

Proof By Induction.

Base Case. A tree of size $K = 1$ has one node at depth $0 = \log_2 1 = \log_2 K$.

Weighted Quick-Union (by size): Running Time

Proposition. The depth of any node in a tree of size $\boldsymbol{K}$ built using a sequence of weighted quick-union operations (by size) is $\leq \log_2 K$.

Proof By Induction.

Base Case. A tree of size $K = 1$ has one node at depth $0 = \log_2 1 = \log_2 K$.

Induction Step. Let the proposition be true for every tree of size $i < K$.
Consider two trees of sizes $M > 0$ and $N > 0$, where $M \leq N$ and $N + M = K$.

Weighted Quick-Union (by size): Running Time

Proposition. The depth of any node in a tree of size $\boldsymbol{K}$ built using a sequence of weighted quick-union operations (by size) is $\leq \log_2 K$.

Proof By Induction.

Base Case. A tree of size $K = 1$ has one node at depth $0 = \log_2 1 = \log_2 K$.

Induction Step. Let the proposition be true for every tree of size $i < K$.
Consider two trees of sizes $M > 0$ and $N > 0$, where $M \leq N$ and $N + M = K$.

By the induction hypothesis, the maximum depth in the *smaller* tree is $\leq \log_2 M$
and the maximum depth in the *larger* tree is $\leq \log_2 N$.

Weighted Quick-Union (by size): Running Time

Proposition. The depth of any node in a tree of size $\boldsymbol{K}$ built using a sequence of weighted quick-union operations (by size) is $\leq \log_2 K$.

Proof By Induction.

Base Case. A tree of size $K = 1$ has one node at depth $0 = \log_2 1 = \log_2 K$.

Induction Step. Let the proposition be true for every tree of size $i < K$.
Consider two trees of sizes $M > 0$ and $N > 0$, where $M \leq N$ and $N + M = K$.

By the induction hypothesis, the maximum depth in the *smaller* tree is $\leq \log_2 M$
and the maximum depth in the *larger* tree is $\leq \log_2 N$.

- The **`UNION`** operation does not affect the depths of the N nodes in the larger subtree.

Weighted Quick-Union (by size): Running Time

Proposition. The depth of any node in a tree of size $\boldsymbol{K}$ built using a sequence of weighted quick-union operations (by size) is $\leq \log_2 K$.

Proof By Induction.

Base Case. A tree of size $K = 1$ has one node at depth $0 = \log_2 1 = \log_2 K$.

Induction Step. Let the proposition be true for every tree of size $i < K$.
Consider two trees of sizes $M > 0$ and $N > 0$, where $M \leq N$ and $N + M = K$.

By the induction hypothesis, the maximum depth in the *smaller* tree is $\leq \log_2 M$
and the maximum depth in the *larger* tree is $\leq \log_2 N$.

- The **`UNION`** operation does not affect the depths of the N nodes in the larger subtree.
- The **`UNION`** operation increases the depth of the M nodes in the smaller subtree by 1.

Weighted Quick-Union (by size): Running Time

Proposition. The depth of any node in a tree of size $\boldsymbol{K}$ built using a sequence of weighted quick-union operations (by size) is $\leq \log_2 K$.

Proof By Induction.

Base Case. A tree of size $K = 1$ has one node at depth $0 = \log_2 1 = \log_2 K$.

Induction Step. Let the proposition be true for every tree of size $i < K$.
Consider two trees of sizes $M > 0$ and $N > 0$, where $M \leq N$ and $N + M = K$.

By the induction hypothesis, the maximum depth in the *smaller* tree is $\leq \log_2 M$
and the maximum depth in the *larger* tree is $\leq \log_2 N$.

- The **`UNION`** operation does not affect the depths of the N nodes in the larger subtree.
- The **`UNION`** operation increases the depth of the M nodes in the smaller subtree by 1.
  This makes the maximum depth in that subtree $\leq \log_2(M) + 1$.

Weighted Quick-Union (by size): Running Time

Proposition. The depth of any node in a tree of size K built using a sequence of weighted quick-union operations (by size) is $\leq \log_2 K$.

Proof By Induction.

Base Case. A tree of size $K = 1$ has one node at depth $0 = \log_2 1 = \log_2 K$.

Induction Step. Let the proposition be true for every tree of size $i < K$.
Consider two trees of sizes $M > 0$ and $N > 0$, where $M \leq N$ and $N + M = K$.

By the induction hypothesis, the maximum depth in the *smaller* tree is $\leq \log_2 M$
and the maximum depth in the *larger* tree is $\leq \log_2 N$.

- The `UNION` operation does not affect the depths of the N nodes in the larger subtree.
- The `UNION` operation increases the depth of the M nodes in the smaller subtree by 1.
  This makes the maximum depth in that subtree $\leq \log_2(M) + 1$.
  However, this is fine, because: $\leq \log_2(M) + \log_2(2) \leq \log_2(2M)$

Weighted Quick-Union (by size): Running Time

Proposition. The depth of any node in a tree of size $\boldsymbol{K}$ built using a sequence of weighted quick-union operations (by size) is $\leq \log_2 K$.

Proof By Induction.

Base Case. A tree of size $K = 1$ has one node at depth $0 = \log_2 1 = \log_2 K$.

Induction Step. Let the proposition be true for every tree of size $i < K$.
Consider two trees of sizes $M > 0$ and $N > 0$, where $M \leq N$ and $N + M = K$.

By the induction hypothesis, the maximum depth in the *smaller* tree is $\leq \log_2 M$
and the maximum depth in the *larger* tree is $\leq \log_2 N$.

- The **`UNION`** operation does not affect the depths of the N nodes in the larger subtree.
- The **`UNION`** operation increases the depth of the M nodes in the smaller subtree by 1.
  This makes the maximum depth in that subtree $\leq \log_2(M) + 1$.
  However, this is fine, because: $\leq \log_2(M) + \log_2(2) \leq \log_2(2M)$
  $\leq \log_2(N + M) \leq \log_2(K)$ ■

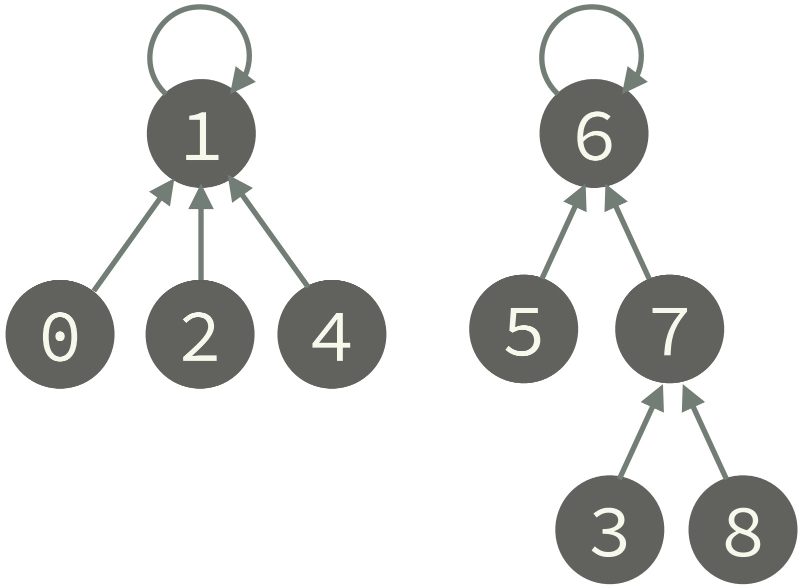
Improvement Attempt 2.2: Weighted Quick-Union (by height)

Idea. Attach the shorter tree to the longer tree.

Rationale. Isn't the height of the tree what we want to optimize?

Modification. Add an array to record the height of the tree rooted at each element.

	0	1	2	3	4	5	6	7	8
parent	1	1	1	7	1	6	6	6	7
height	0	1	0	0	0	0	2	1	0



Improvement Attempt 2.2: Weighted Quick-Union (by height)

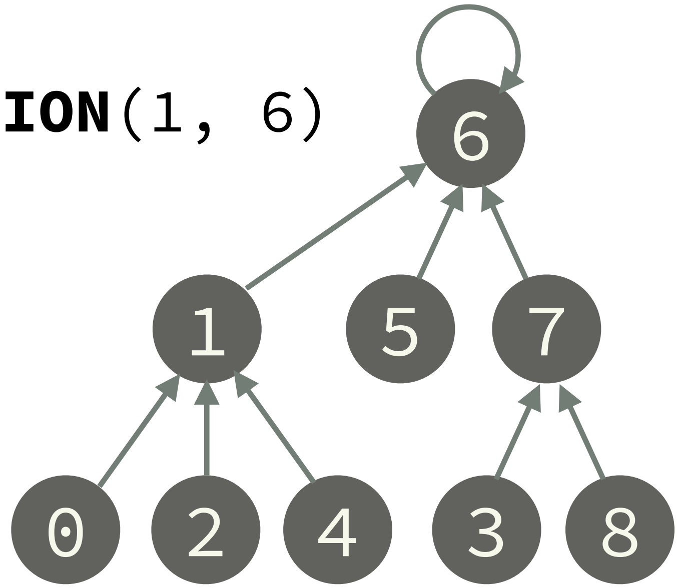
Idea. Attach the shorter tree to the longer tree.

Rationale. Isn't the height of the tree what we want to optimize?

Modification. Add an array to record the height of the tree rooted at each element.

	0	1	2	3	4	5	6	7	8
parent	1	1	1	7	1	6	6	6	7
height	0	**1**	0	0	0	0	**2**	1	0

`UNION(1, 6)`



Improvement Attempt 2.2: Weighted Quick-Union (by height)

Idea. Attach the shorter tree to the longer tree.

Rationale. Isn't the height of the tree what we want to optimize?

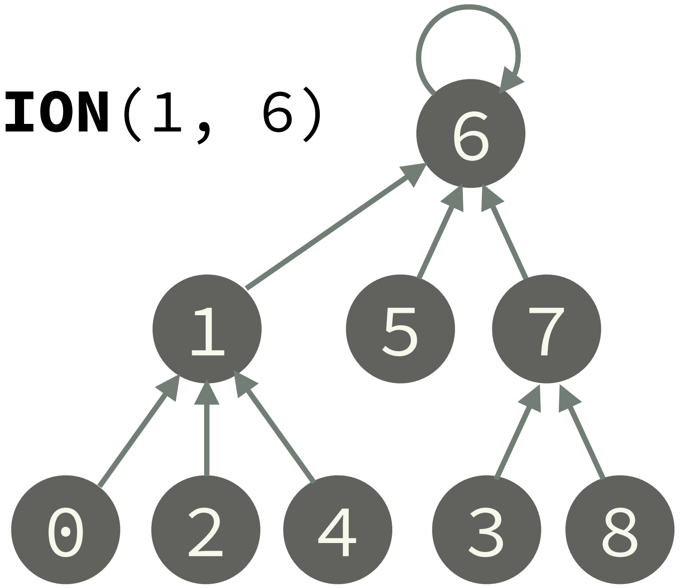
Modification. Add an array to record the height of the tree rooted at each element.

parent	1	1	1	7	1	6	6	6	7
height	0	**1**	0	0	0	0	**2**	1	0
	0	1	2	3	4	5	6	7	8

UNION(p, q)

```
r1 = FIND(p)
r2 = FIND(q)
if (r1 == r2) return
```

UNION(1, 6)



Improvement Attempt 2.2: Weighted Quick-Union (by height)

Idea. Attach the shorter tree to the longer tree.

Rationale. Isn't the height of the tree what we want to optimize?

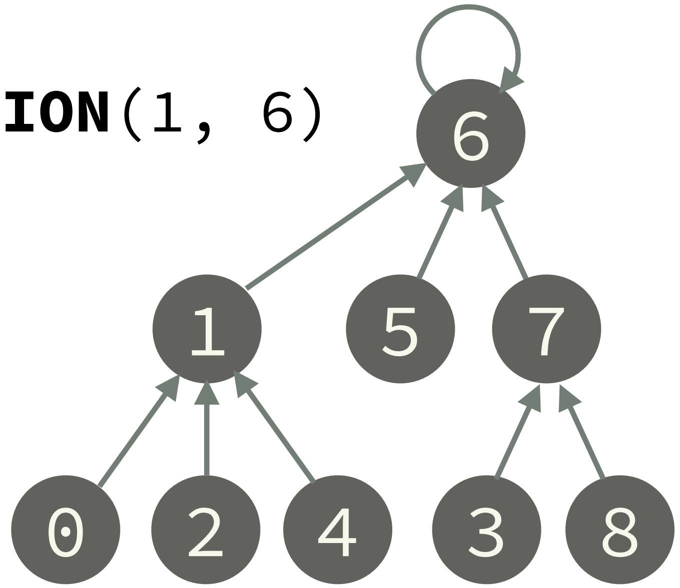
Modification. Add an array to record the height of the tree rooted at each element.

parent	1	1	1	7	1	6	6	6	7
height	0	1	0	0	0	0	2	1	0
	0	1	2	3	4	5	6	7	8

UNION(p, q)

```
r1 = FIND(p)
r2 = FIND(q)
if (r1 == r2) return

if (height[r1] < height[r2])
   SWAP(r1, r2)
parent[r2] = r1
```

UNION(1, 6)



Improvement Attempt 2.2: Weighted Quick-Union (by height)

Idea. Attach the shorter tree to the longer tree.

Rationale. Isn't the height of the tree what we want to optimize?

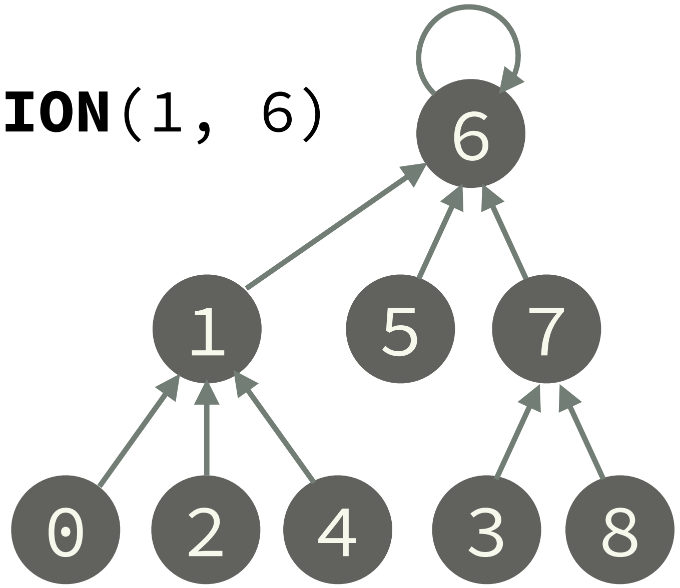
Modification. Add an array to record the height of the tree rooted at each element.

parent	1	1	1	7	1	6	6	6	7
height	0	**1**	0	0	0	0	**2**	1	0
	0	1	2	3	4	5	6	7	8

UNION(p, q)

```
r1 = FIND(p)
r2 = FIND(q)
if (r1 == r2) return

if (height[r1] < height[r2])
   SWAP(r1, r2)
parent[r2] = r1

if (height[r1] == height[r2])
   height[r1] += 1
```

UNION(1, 6)



Improvement Attempt 2.2: Weighted Quick-Union (by height)

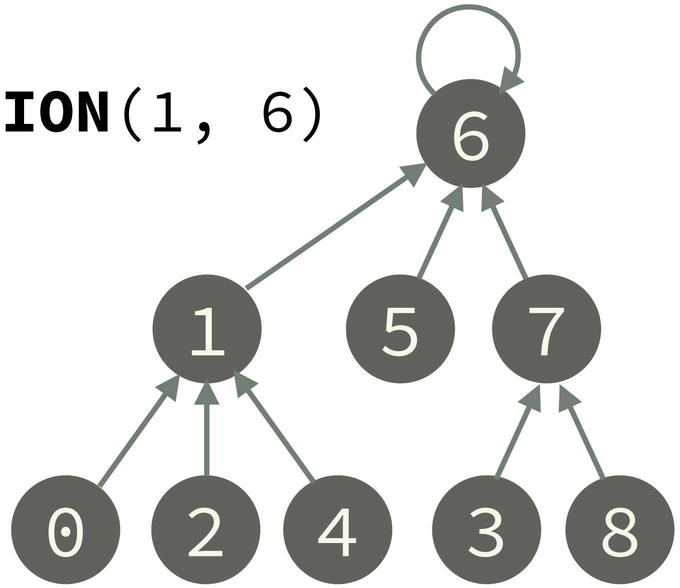
Idea. Attach the shorter tree to the longer tree.

Rationale. Isn't the height of the tree what we want to optimize?

Modification. Add an array to record the height of the tree rooted at each element.

parent	1	1	1	7	1	6	6	6	7
height	0	1	0	0	0	0	2	1	0
	0	1	2	3	4	5	6	7	8

UNION(1, 6)



```
UNION(p, q)

r1 = FIND(p)
r2 = FIND(q)
if (r1 == r2) return

if (height[r1] < height[r2])
   SWAP(r1, r2)
parent[r2] = r1

if (height[r1] == height[r2])
   height[r1] += 1
```

Important. Height changes **only** when two trees of the same height are merged.

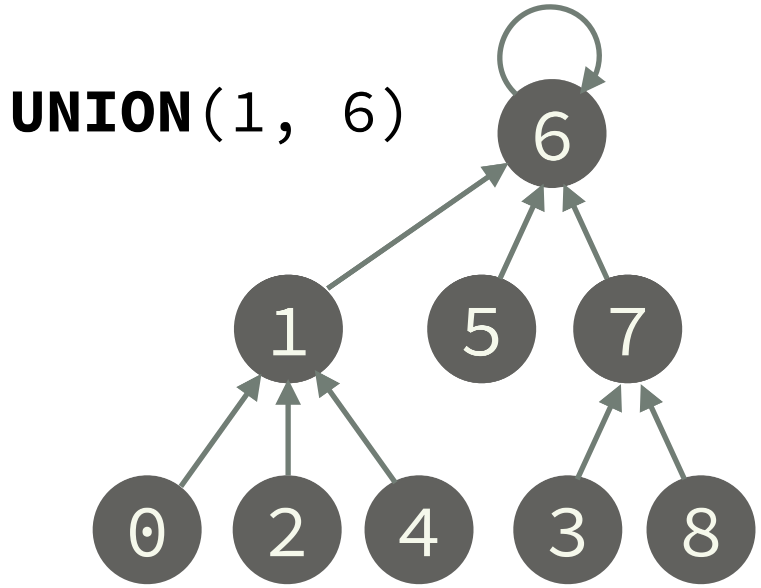
Improvement Attempt 2.2: Weighted Quick-Union (by height)

Idea. Attach the shorter tree to the longer tree.

Rationale. Isn't the height of the tree what we want to optimize?

Modification. Add an array to record the height of the tree rooted at each element.

parent	1	1	1	7	1	6	6	6	7
height	0	**1**	0	0	0	0	**2**	1	0
	0	1	2	3	4	5	6	7	8



UNION(p, q)

```
r1 = FIND(p)
r2 = FIND(q)
if (r1 == r2) return

if (height[r1] < height[r2])
   SWAP(r1, r2)
parent[r2] = r1

if (height[r1] == height[r2])
   height[r1] += 1
```

Worst Case Running Time.

FIND: $O(\log n)$
UNION: $O(\log n)$

Cost Model. Number of array accesses.

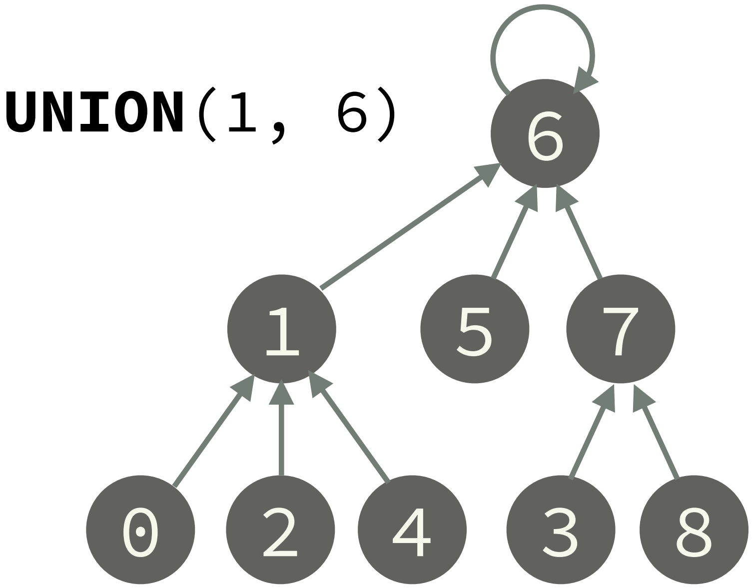
Improvement Attempt 2.2: Weighted Quick-Union (by height)

Idea. Attach the shorter tree to the longer tree.

Rationale. Isn't the height of the tree what we want to optimize?

Modification. Add an array to record the height of the tree rooted at each element.

parent	1	1	1	7	1	6	6	6	7
height	0	**1**	0	0	0	0	**2**	1	0
	0	1	2	3	4	5	6	7	8



UNION(p, q)

```
r1 = FIND(p)
r2 = FIND(q)
if (r1 == r2) return

if (height[r1] < height[r2])
   SWAP(r1, r2)
parent[r2] = r1

if (height[r1] == height[r2])
   height[r1] += 1
```

Worst Case Running Time.

FIND: $O(\log n)$
UNION: $O(\log n)$

Cost Model. Number of array accesses.

I see a proof coming ...

Weighted Quick-Union (by height): Running Time

Proposition 1. Any tree of height $\boldsymbol{H}$ built using a sequence of weighted quick-union operations (by height) has $\geq 2^H$ nodes.

Weighted Quick-Union (by height): Running Time

Proposition 1. Any tree of height $\boldsymbol{H}$ built using a sequence of weighted quick-union operations (by height) has $\geq 2^H$ nodes.

Proof By Induction.

Base Case. A tree of height $h = 0$ has ≥ 1 nodes $(1 = 2^0 = 2^h)$.

Weighted Quick-Union (by height): Running Time

Proposition 1. Any tree of height $\boldsymbol{H}$ built using a sequence of weighted quick-union operations (by height) has $\geq 2^H$ nodes.

Proof By Induction.

Base Case. A tree of height $h = 0$ has ≥ 1 nodes $(1 = 2^0 = 2^h)$.

Induction Step. Assume the proposition is true for every tree of height $h < H$.

Weighted Quick-Union (by height): Running Time

Proposition 1. Any tree of height $\boldsymbol{H}$ built using a sequence of weighted quick-union operations (by height) has $\geq 2^H$ nodes.

Proof By Induction.

Base Case. A tree of height $h = 0$ has ≥ 1 nodes $(1 = 2^0 = 2^h)$.

Induction Step. Assume the proposition is true for every tree of height $h < H$.

Consider a tree $\boldsymbol{T}$ of height $\boldsymbol{H}$ created from merging two trees of height $H - 1$ each.

Weighted Quick-Union (by height): Running Time

Proposition 1. Any tree of height $\boldsymbol{H}$ built using a sequence of weighted quick-union operations (by height) has $\geq 2^H$ nodes.

Proof By Induction.

Base Case. A tree of height $h = 0$ has ≥ 1 nodes $(1 = 2^0 = 2^h)$.

Induction Step. Assume the proposition is true for every tree of height $h < H$.

Consider a tree $\boldsymbol{T}$ of height $\boldsymbol{H}$ created from merging two trees of height $H - 1$ each. From the inductive hypothesis, each of the trees has $\geq 2^{H-1}$ nodes and $\boldsymbol{T}$ has $\geq 2^{H-1} + 2^{H-1} \geq 2^H$ nodes. ■

Weighted Quick-Union (by height): Running Time

Proposition 1. Any tree of height $\boldsymbol{H}$ built using a sequence of weighted quick-union operations (by height) has $\geq 2^H$ nodes.

Proof By Induction.

Base Case. A tree of height $h = 0$ has ≥ 1 nodes $(1 = 2^0 = 2^h)$.

Induction Step. Assume the proposition is true for every tree of height $h < H$.

Consider a tree $\boldsymbol{T}$ of height $\boldsymbol{H}$ created from merging two trees of height $H - 1$ each. From the inductive hypothesis, each of the trees has $\geq 2^{H-1}$ nodes and $\boldsymbol{T}$ has $\geq 2^{H-1} + 2^{H-1} \geq 2^H$ nodes. ■

Proposition 2. A tree of $\boldsymbol{N}$ nodes built using a sequence of weighted quick-union operations (by height) cannot have a height $> \log_2 N$.

Weighted Quick-Union (by height): Running Time

Proposition 1. Any tree of height $\boldsymbol{H}$ built using a sequence of weighted quick-union operations (by height) has $\geq 2^H$ nodes.

Proof By Induction.

Base Case. A tree of height $h = 0$ has ≥ 1 nodes $(1 = 2^0 = 2^h)$.

Induction Step. Assume the proposition is true for every tree of height $h < H$.

Consider a tree $\boldsymbol{T}$ of height $\boldsymbol{H}$ created from merging two trees of height $H - 1$ each. From the inductive hypothesis, each of the trees has $\geq 2^{H-1}$ nodes and $\boldsymbol{T}$ has $\geq 2^{H-1} + 2^{H-1} \geq 2^H$ nodes. ■

Proposition 2. A tree of $\boldsymbol{N}$ nodes built using a sequence of weighted quick-union operations (by height) cannot have a height $> \log_2 N$.

Proof. From the proof of proposition 1, a tree of height $h > \log_2 N$ has more than $2^{\log_2 N} > N$ nodes. This is a contradiction, as there are only N nodes in the tree! ■

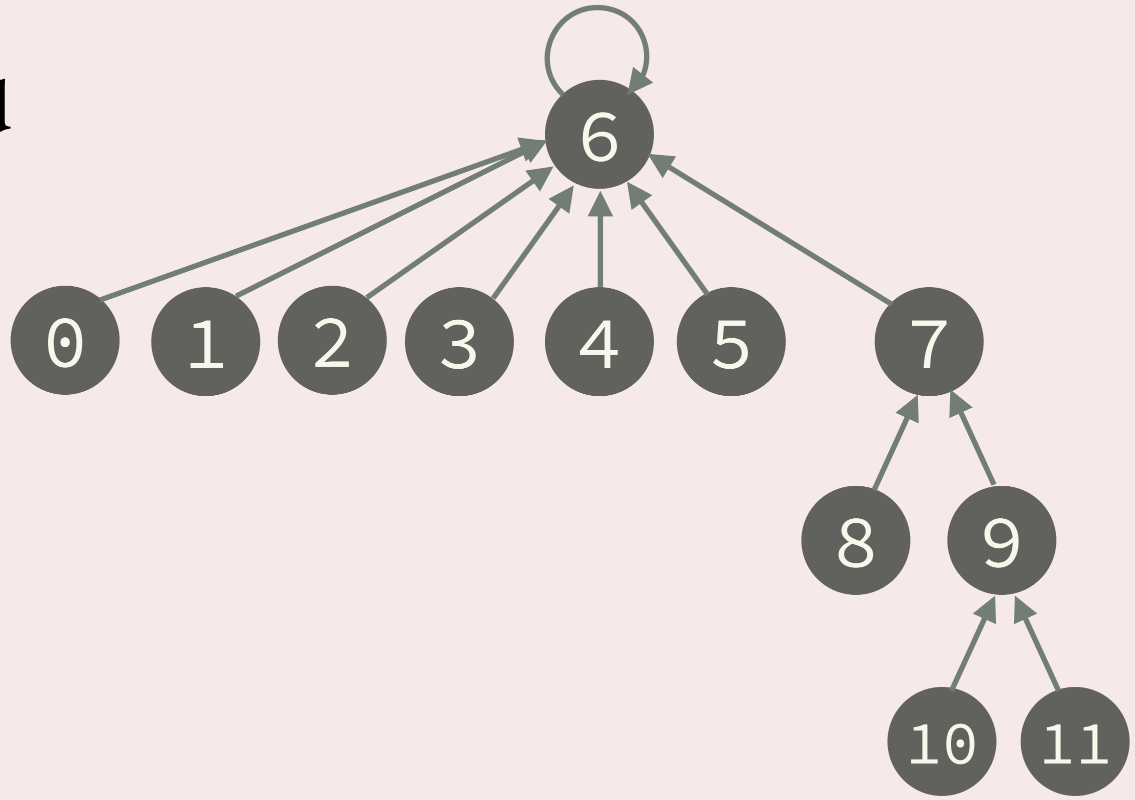
Quiz # 3

Draw a tree that can be the result of weighted quick-union-by-size but can't be the result of weighted quick-union-by-height.

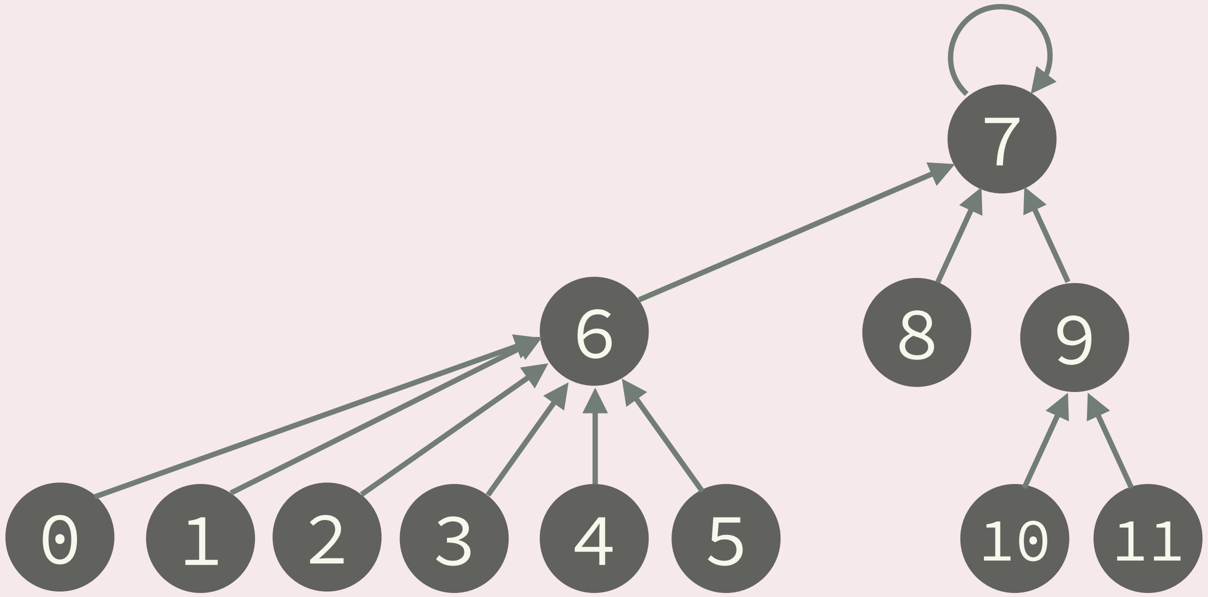
Draw a tree that can be the result of weighted quick-union-by-height but can't be the result of weighted quick-union-by-size.

Quiz # 3

Draw a tree that can be the result of weighted quick-union-by-size but can't be the result of weighted quick-union-by-height.



Draw a tree that can be the result of weighted quick-union-by-height but can't be the result of weighted quick-union-by-size.



Union-Find: Running Time Summary

	Quick-**Find** (array)	Quick-**Find** (linked-list)
`FIND`	$O(1)$	$O(1)$
`UNION`	$O(n)$	$O(n)$
Sequence of n `UNION` operations:	$O(n^2)$	$O(n \log n)$

Union-Find: Running Time Summary

	Quick-**Find** (array)	Quick-**Find** (linked-list)	Quick-**Union**
`FIND`	$O(1)$	$O(1)$	$O(n)$
`UNION`	$O(n)$	$O(n)$	$O(n)$
Sequence of n `UNION` operations:	$O(n^2)$	$O(n \log n)$	$O(n^2)$

Union-Find: Running Time Summary

	Quick-**Find** (array)	Quick-**Find** (linked-list)	Quick-**Union**	Weighted Quick-**Union**
`FIND`	$O(1)$	$O(1)$	$O(n)$	$O(\log n)$
`UNION`	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$
Sequence of n `UNION` operations:	$O(n^2)$	$O(n \log n)$	$O(n^2)$	$O(n \log n)$

Union-Find: Running Time Summary

	Quick-**Find** (array)	Quick-**Find** (linked-list)	Quick-**Union**	Weighted Quick-**Union**
FIND	$O(1)$	$O(1)$	$O(n)$	$O(\log n)$
UNION	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$
Sequence of n **UNION** operations:	$O(n^2)$	$O(n \log n)$	$O(n^2)$	$O(n \log n)$

🤔

Can we do better?

Improvement Attempt 3: Path Compression

Idea. When `FIND` is called, attach directly to the root every node visited by `FIND`.

Rationale. Make use of work done anyway to speed up future calls to `FIND`.

Improvement Attempt 3: Path Compression

Idea. When **`FIND`** is called, attach directly to the root every node visited by **`FIND`**.

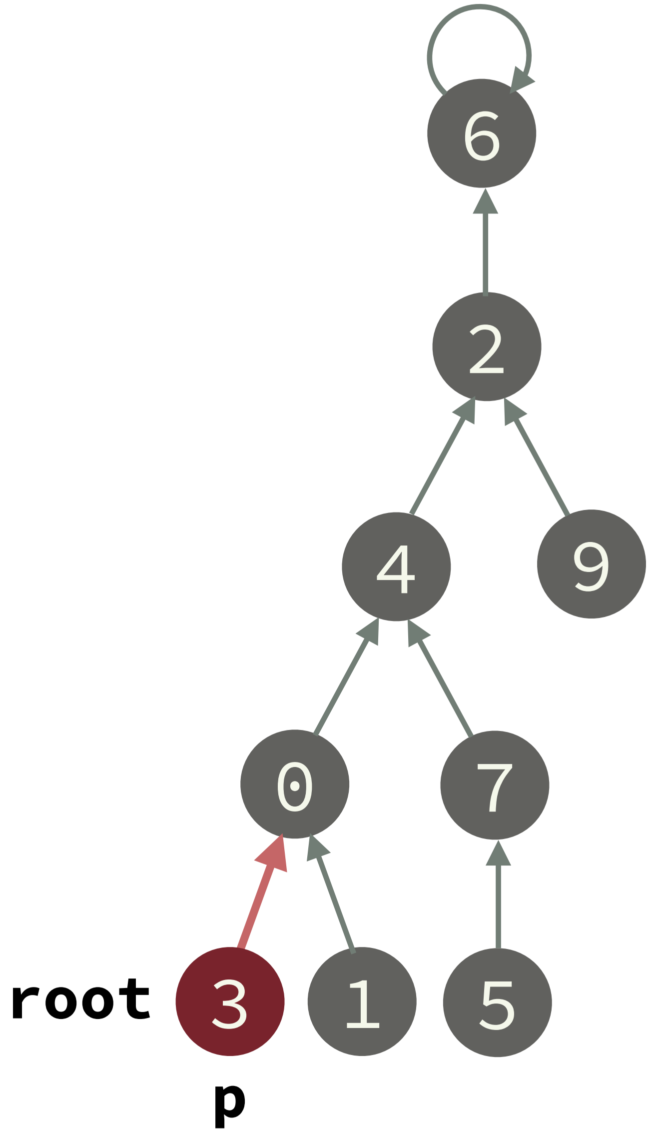
Rationale. Make use of work done anyway to speed up future calls to **`FIND`**.

FIND(p)

```
root = p
while (parent[root] != root)
    root = parent[root]
```

1. get the root as usual

Example. **`FIND`**`(3)`



Improvement Attempt 3: Path Compression

Idea. When **`FIND`** is called, attach directly to the root every node visited by **`FIND`**.

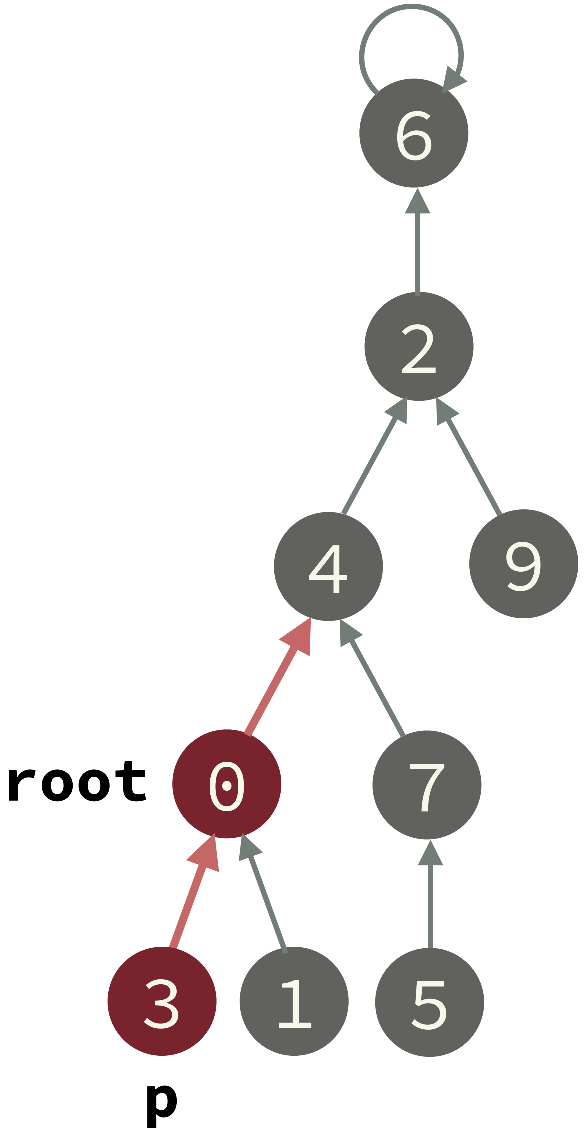
Rationale. Make use of work done anyway to speed up future calls to **`FIND`**.

`FIND``(p)`

```
root = p
while (parent[root] != root)
    root = parent[root]
```

1. get the root as usual

Example. **`FIND`**`(3)`



Improvement Attempt 3: Path Compression

Idea. When **`FIND`** is called, attach directly to the root every node visited by **`FIND`**.

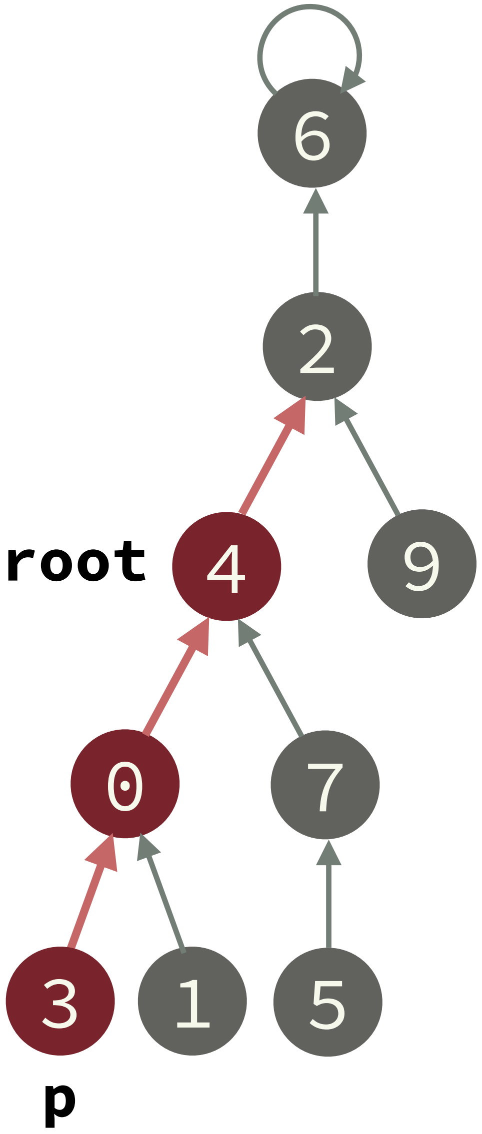
Rationale. Make use of work done anyway to speed up future calls to **`FIND`**.

`FIND(p)`

```
root = p
while (parent[root] != root)
    root = parent[root]
```

1. get the root as usual

Example. **`FIND`**`(3)`



Improvement Attempt 3: Path Compression

Idea. When **FIND** is called, attach directly to the root every node visited by **FIND**.

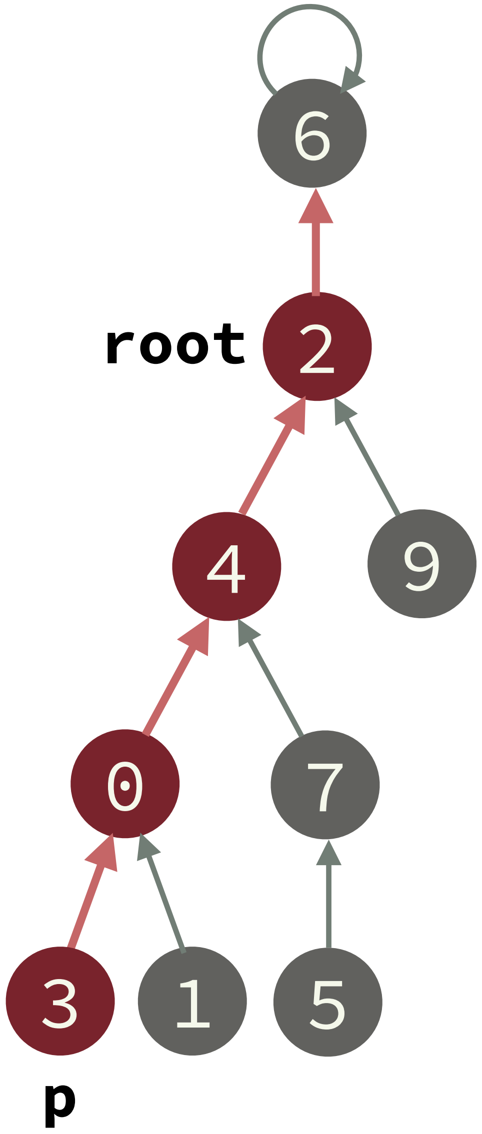
Rationale. Make use of work done anyway to speed up future calls to **FIND**.

FIND(p)

```
root = p
while (parent[root] != root)
    root = parent[root]
```

1. get the root as usual

Example. **FIND**(3)



Improvement Attempt 3: Path Compression

Idea. When **`FIND`** is called, attach directly to the root every node visited by **`FIND`**.

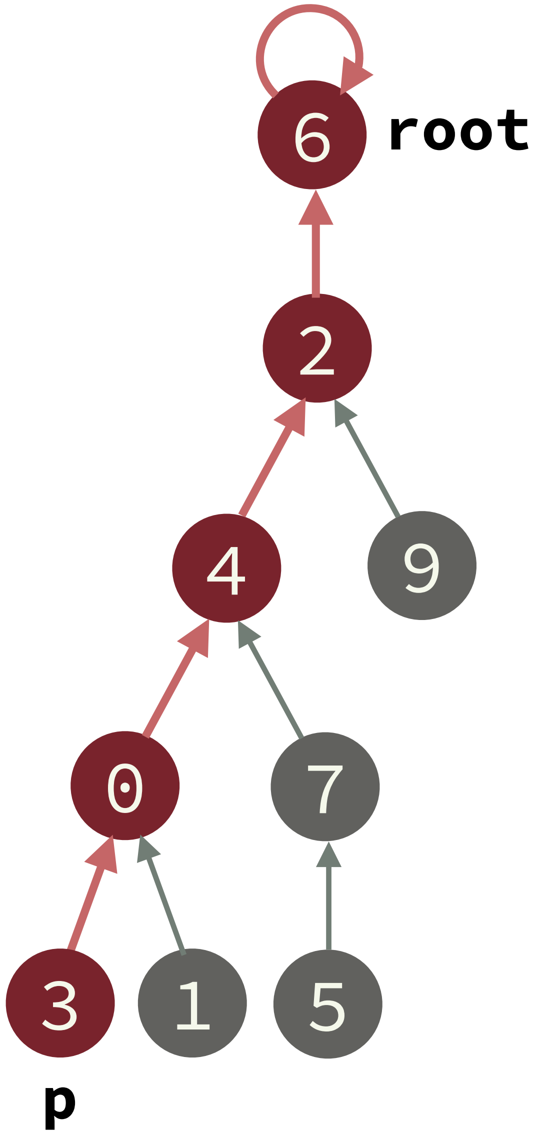
Rationale. Make use of work done anyway to speed up future calls to **`FIND`**.

`FIND(p)`

```
root = p
while (parent[root] != root)
   root = parent[root]
```

1. get the root as usual

Example. **`FIND(3)`**



Improvement Attempt 3: Path Compression

Idea. When **`FIND`** is called, attach directly to the root every node visited by **`FIND`**.

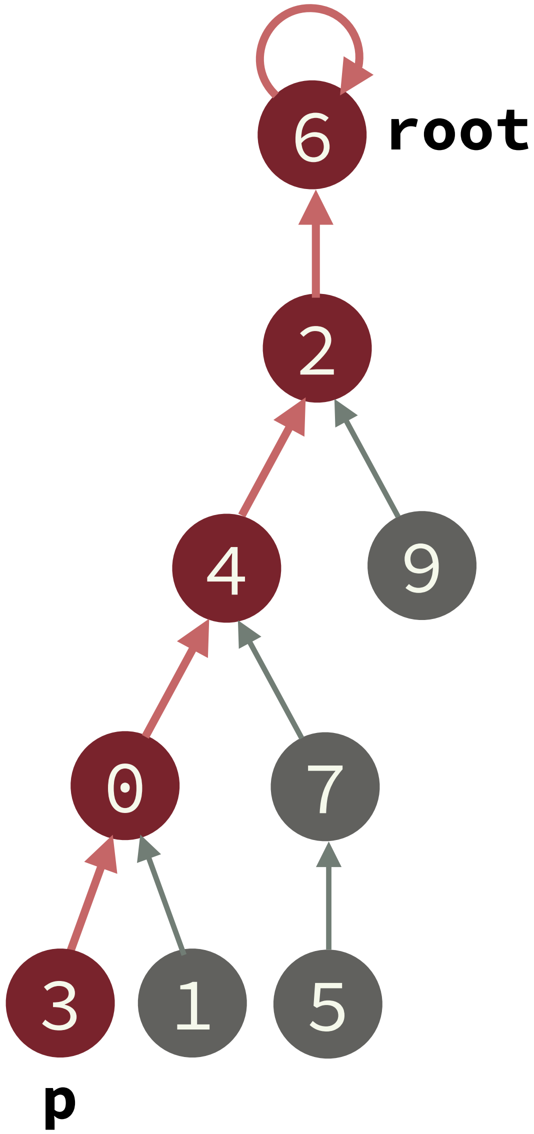
Rationale. Make use of work done anyway to speed up future calls to **`FIND`**.

`FIND(p)`

```
root = p
while (parent[root] != root)
    root = parent[root]

while (p != root)
    next = parent[p]
    parent[p] = root
    p = next

return p
```

2. link every node on the **`FIND`** path with the root

Example. **`FIND(3)`**



Improvement Attempt 3: Path Compression

Idea. When **FIND** is called, attach directly to the root every node visited by **FIND**.

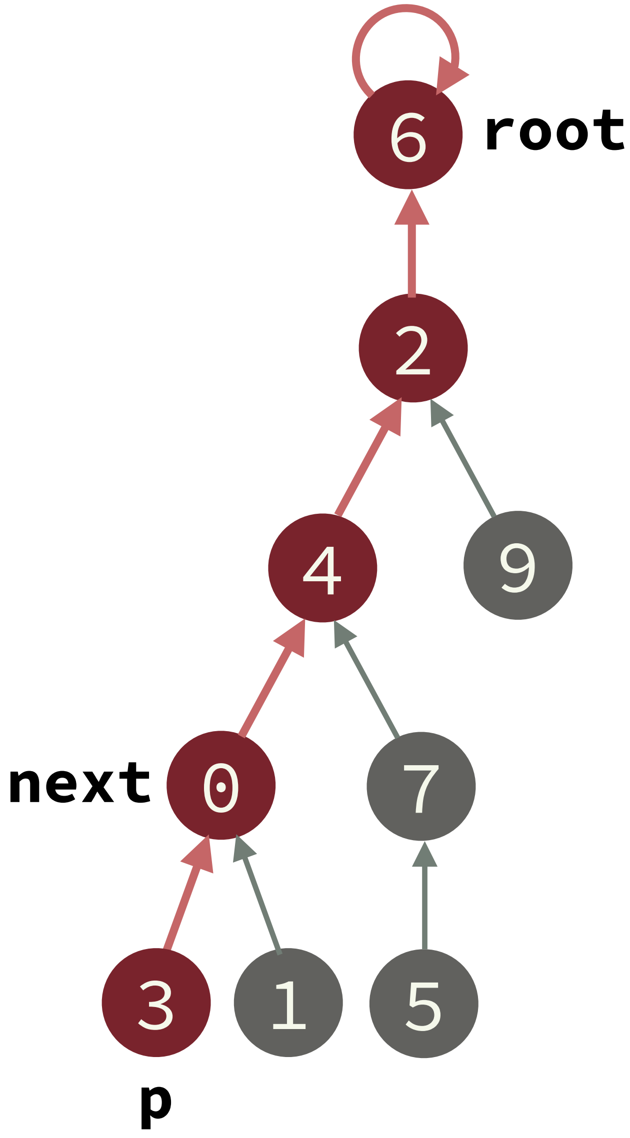
Rationale. Make use of work done anyway to speed up future calls to **FIND**.

FIND(p)

```
root = p
while (parent[root] != root)
    root = parent[root]

while (p != root)
    next = parent[p]
    parent[p] = root
    p = next

return p
```

2. link every node on the **FIND** path with the root

Example. **FIND(3)**



Improvement Attempt 3: Path Compression

Idea. When **`FIND`** is called, attach directly to the root every node visited by **`FIND`**.

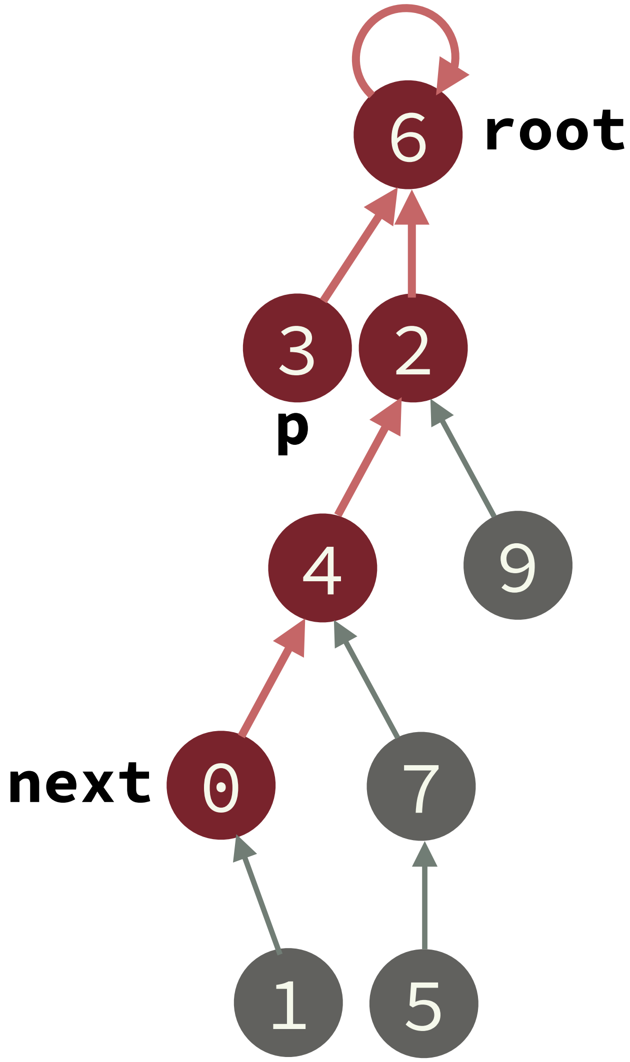
Rationale. Make use of work done anyway to speed up future calls to **`FIND`**.

`FIND``(p)`

```
root = p
while (parent[root] != root)
    root = parent[root]

while (p != root)
    next = parent[p]
    parent[p] = root
    p = next

return p
```

2. link every node on the **`FIND`** path with the root

Example. **`FIND`**`(3)`



Improvement Attempt 3: Path Compression

Idea. When **FIND** is called, attach directly to the root every node visited by **FIND**.

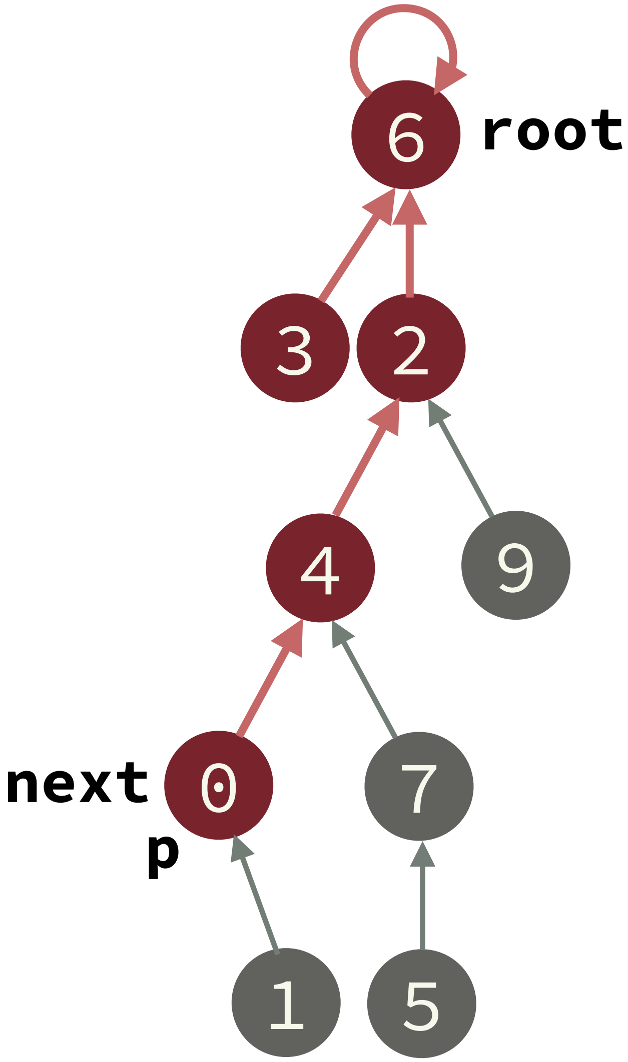
Rationale. Make use of work done anyway to speed up future calls to **FIND**.

FIND(p)

```
root = p
while (parent[root] != root)
   root = parent[root]

while (p != root)
   next = parent[p]
   parent[p] = root
   p = next

return p
```

2. link every node on the **FIND** path with the root

Example. **FIND**(3)



Improvement Attempt 3: Path Compression

Idea. When **`FIND`** is called, attach directly to the root every node visited by **`FIND`**.

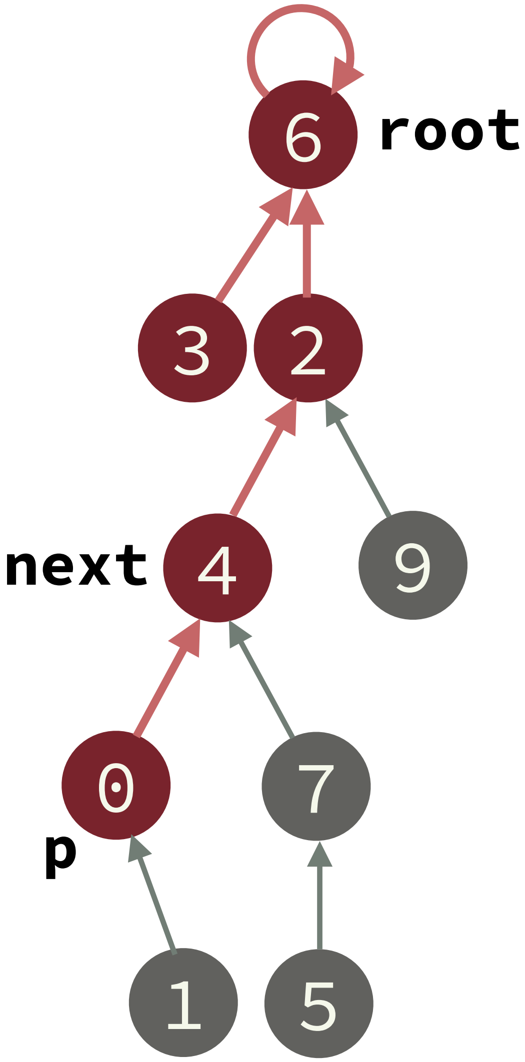
Rationale. Make use of work done anyway to speed up future calls to **`FIND`**.

`FIND(p)`

```
root = p
while (parent[root] != root)
    root = parent[root]

while (p != root)
    next = parent[p]
    parent[p] = root
    p = next

return p
```

2. link every node on the **`FIND`** path with the root

Example. **`FIND(3)`**



Improvement Attempt 3: Path Compression

Idea. When **FIND** is called, attach directly to the root every node visited by **FIND**.

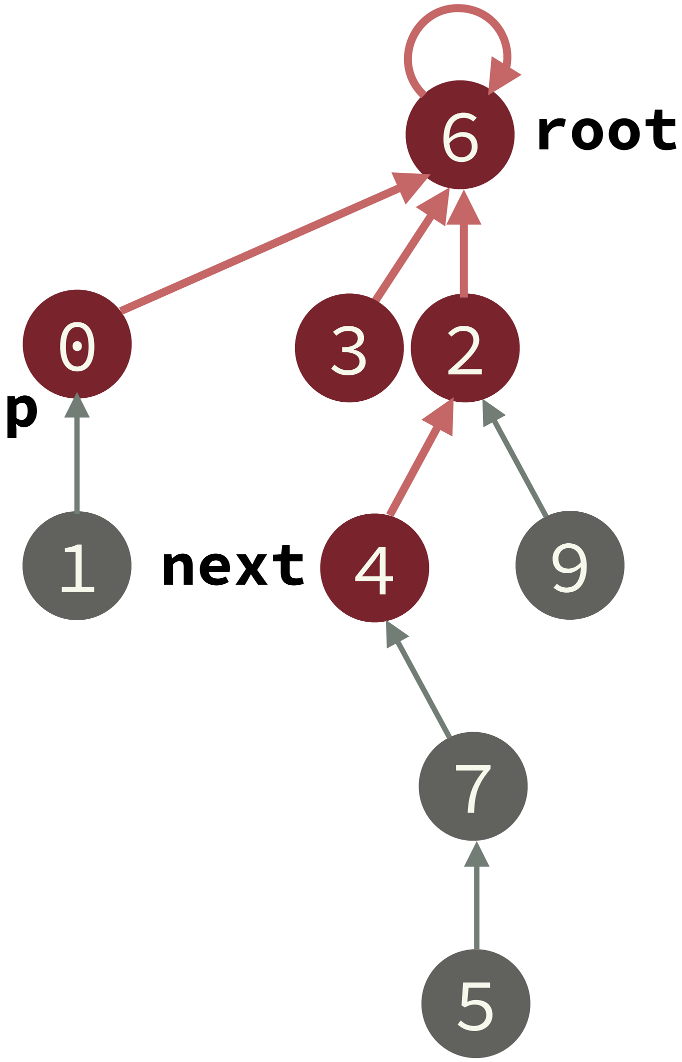
Rationale. Make use of work done anyway to speed up future calls to **FIND**.

FIND(p)

```
root = p
while (parent[root] != root)
   root = parent[root]

while (p != root)
   next = parent[p]
   parent[p] = root
   p = next

return p
```

2. link every node on the **FIND** path with the root

Example. **FIND**(3)



Improvement Attempt 3: Path Compression

Idea. When **`FIND`** is called, attach directly to the root every node visited by **`FIND`**.

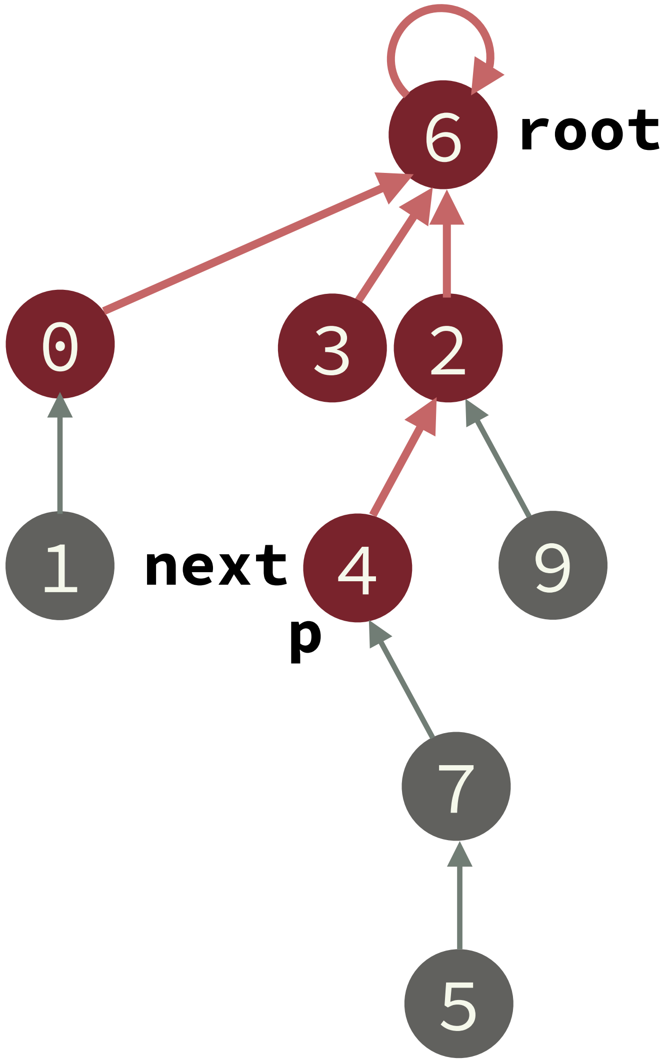
Rationale. Make use of work done anyway to speed up future calls to **`FIND`**.

`FIND(p)`

```
root = p
while (parent[root] != root)
    root = parent[root]

while (p != root)
    next = parent[p]
    parent[p] = root
    p = next

return p
```

2. link every node on the **`FIND`** path with the root

Example. **`FIND(3)`**



Improvement Attempt 3: Path Compression

Idea. When **FIND** is called, attach directly to the root every node visited by **FIND**.

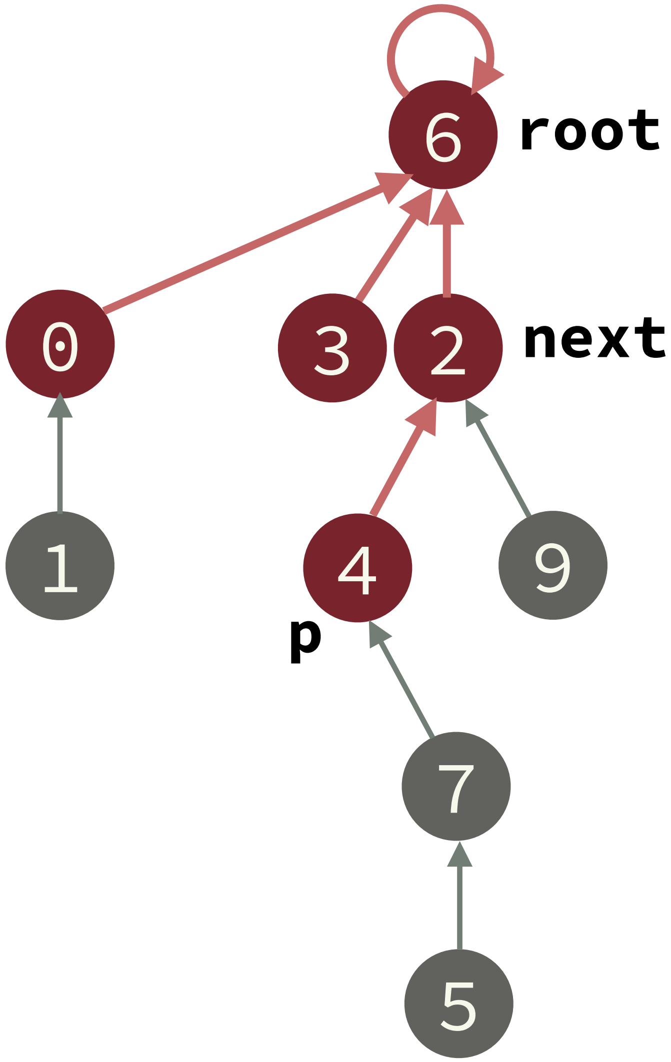
Rationale. Make use of work done anyway to speed up future calls to **FIND**.

FIND(p)

```
root = p
while (parent[root] != root)
    root = parent[root]

while (p != root)
    next = parent[p]
    parent[p] = root
    p = next

return p
```

2. link every node on the **FIND** path with the root

Example. **FIND**(3)



Improvement Attempt 3: Path Compression

Idea. When **`FIND`** is called, attach directly to the root every node visited by **`FIND`**.

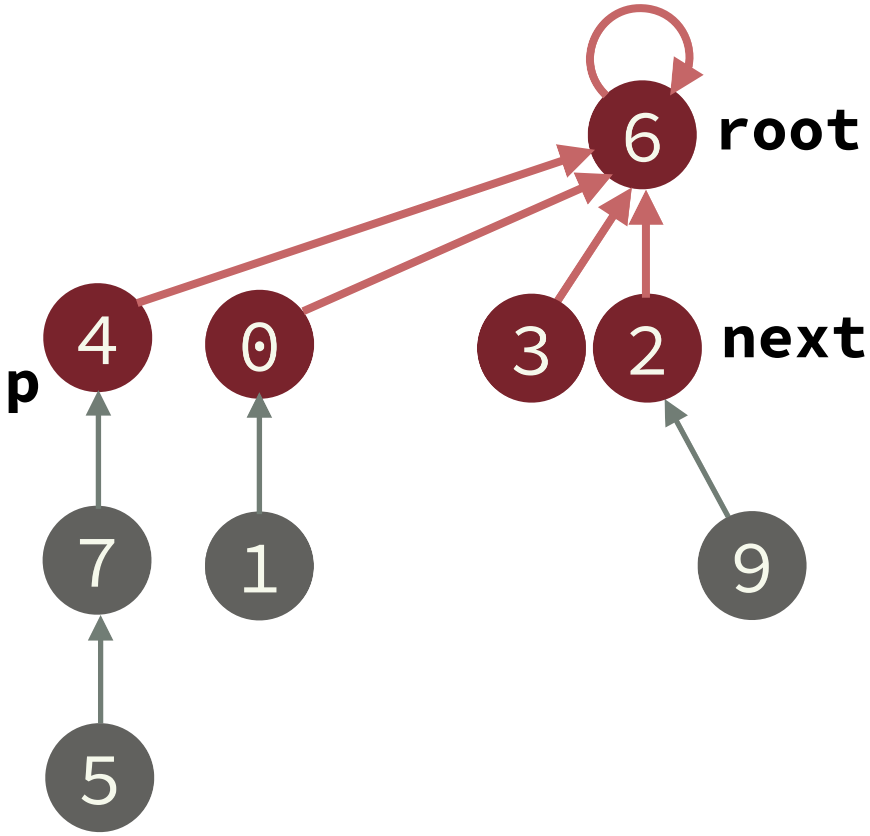
Rationale. Make use of work done anyway to speed up future calls to **`FIND`**.

`FIND(p)`

```
root = p
while (parent[root] != root)
   root = parent[root]

while (p != root)
   next = parent[p]
   parent[p] = root
   p = next

return p
```

2. link every node on the
`FIND` path with the root

Example. **`FIND(3)`**



Improvement Attempt 3: Path Compression

Idea. When **`FIND`** is called, attach directly to the root every node visited by **`FIND`**.

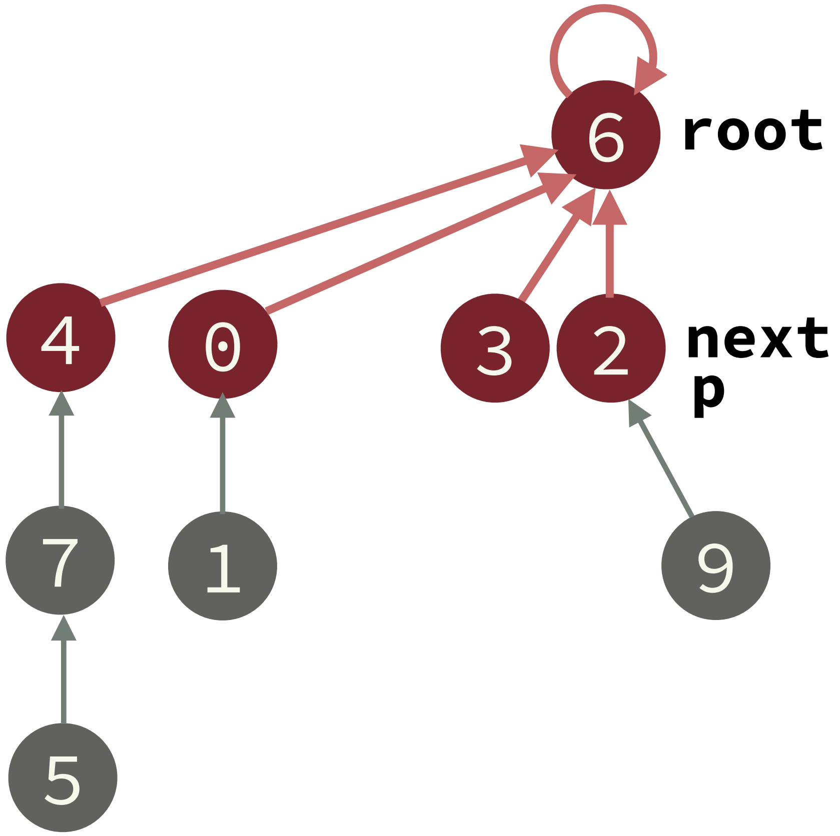
Rationale. Make use of work done anyway to speed up future calls to **`FIND`**.

`FIND(p)`

```
root = p
while (parent[root] != root)
    root = parent[root]

while (p != root)
    next = parent[p]
    parent[p] = root
    p = next

return p
```

2. link every node on the **`FIND`** path with the root

Example. **`FIND(3)`**



Improvement Attempt 3: Path Compression

Idea. When **FIND** is called, attach directly to the root every node visited by **FIND**.

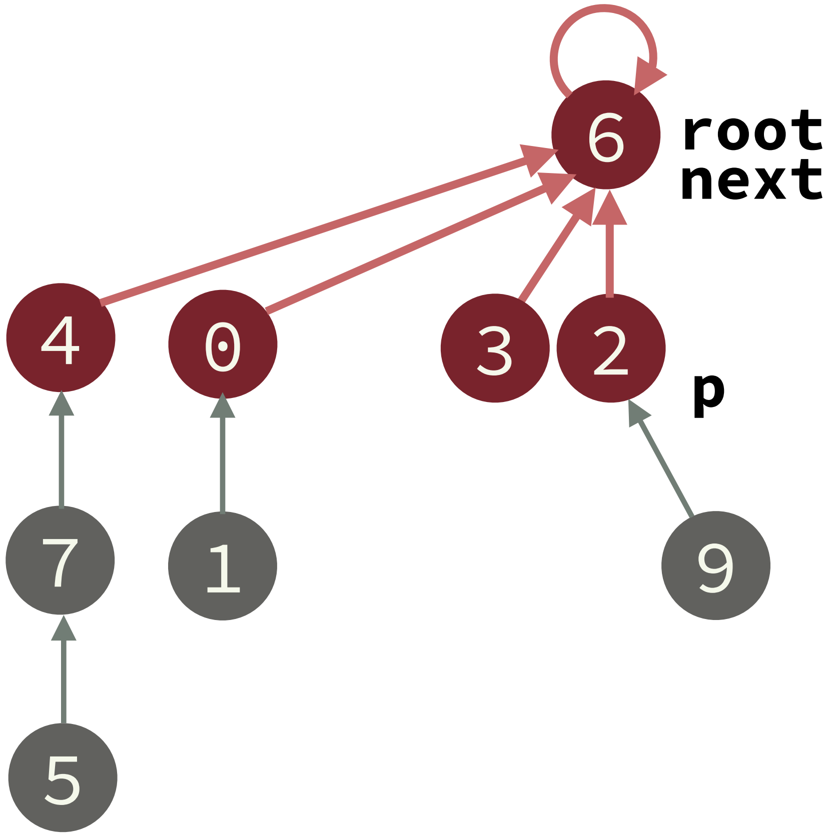
Rationale. Make use of work done anyway to speed up future calls to **FIND**.

FIND(p)

```
root = p
while (parent[root] != root)
    root = parent[root]

while (p != root)
    next = parent[p]
    parent[p] = root
    p = next

return p
```

2. link every node on the **FIND** path with the root

Example. **FIND(3)**



Improvement Attempt 3: Path Compression

Idea. When **FIND** is called, attach directly to the root every node visited by **FIND**.

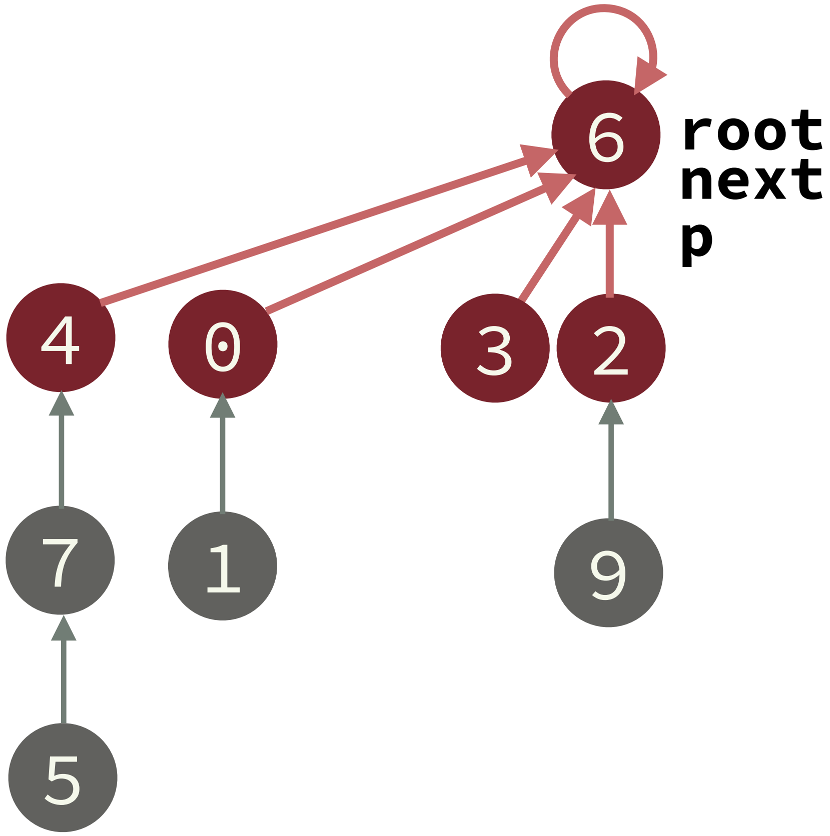
Rationale. Make use of work done anyway to speed up future calls to **FIND**.

FIND(p)

```
root = p
while (parent[root] != root)
    root = parent[root]

while (p != root)
    next = parent[p]
    parent[p] = root
    p = next

return p
```

2. link every node on the **FIND** path with the root

Example. **FIND**(3)



Improvement Attempt 3: Path Compression

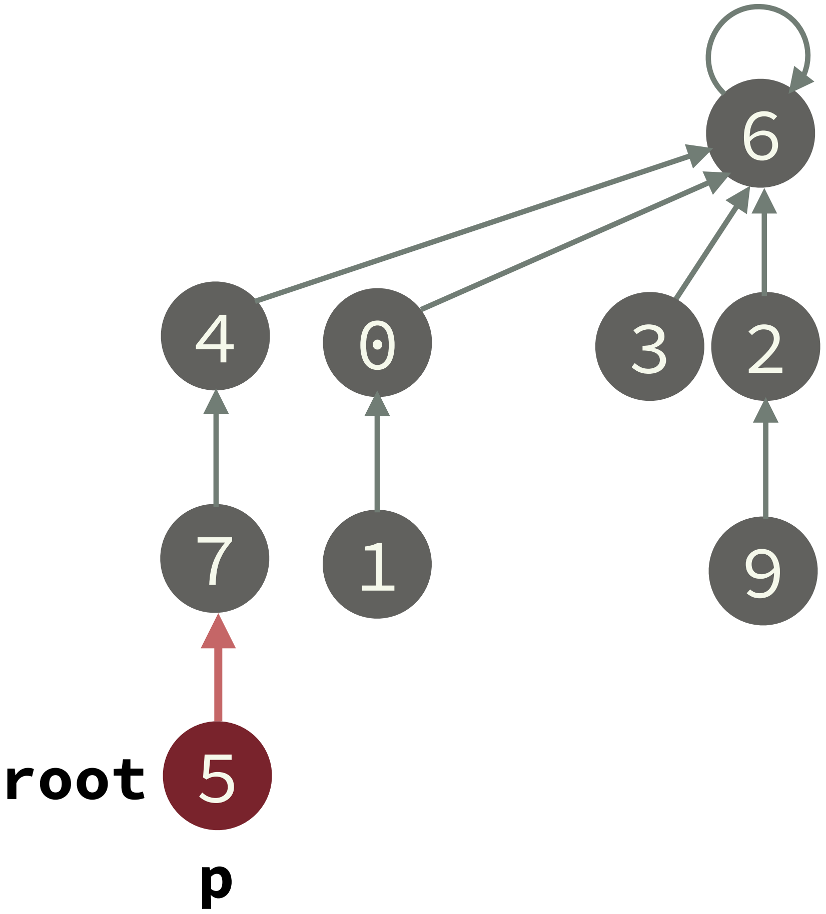
Idea. When **`FIND`** is called, attach directly to the root every node visited by **`FIND`**.

Rationale. Make use of work done anyway to speed up future calls to **`FIND`**.

`FIND(p)`

```
root = p
while (parent[root] != root)
    root = parent[root]

while (p != root)
    next = parent[p]
    parent[p] = root
    p = next

return p
```

Example. **`FIND(5)`**



Improvement Attempt 3: Path Compression

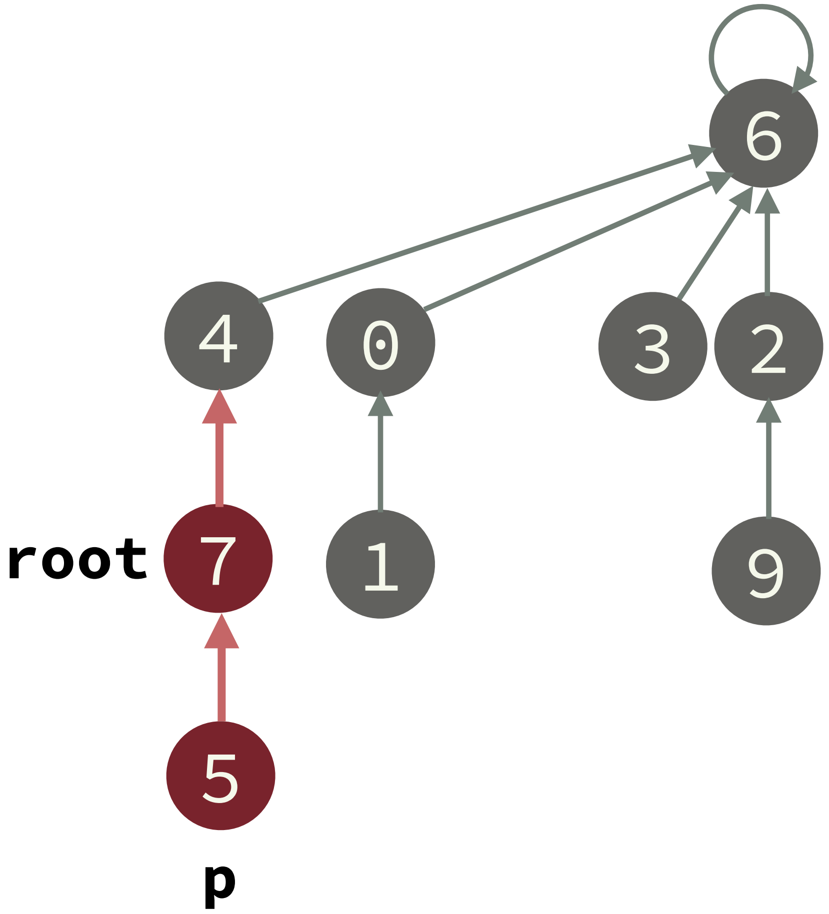
Idea. When **FIND** is called, attach directly to the root every node visited by **FIND**.

Rationale. Make use of work done anyway to speed up future calls to **FIND**.

FIND(p)

```
root = p
while (parent[root] != root)
    root = parent[root]

while (p != root)
    next = parent[p]
    parent[p] = root
    p = next

return p
```

Example. **FIND(5)**



Improvement Attempt 3: Path Compression

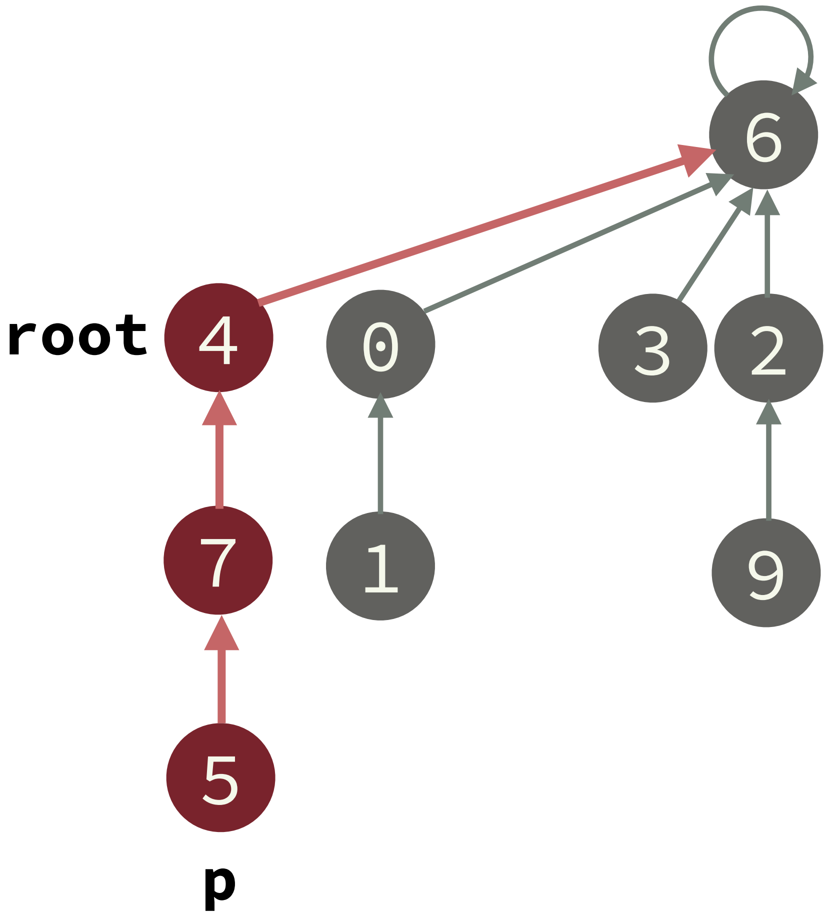
Idea. When **`FIND`** is called, attach directly to the root every node visited by **`FIND`**.

Rationale. Make use of work done anyway to speed up future calls to **`FIND`**.

`FIND(p)`

```
root = p
while (parent[root] != root)
    root = parent[root]

while (p != root)
    next = parent[p]
    parent[p] = root
    p = next

return p
```

Example. **`FIND(5)`**



Improvement Attempt 3: Path Compression

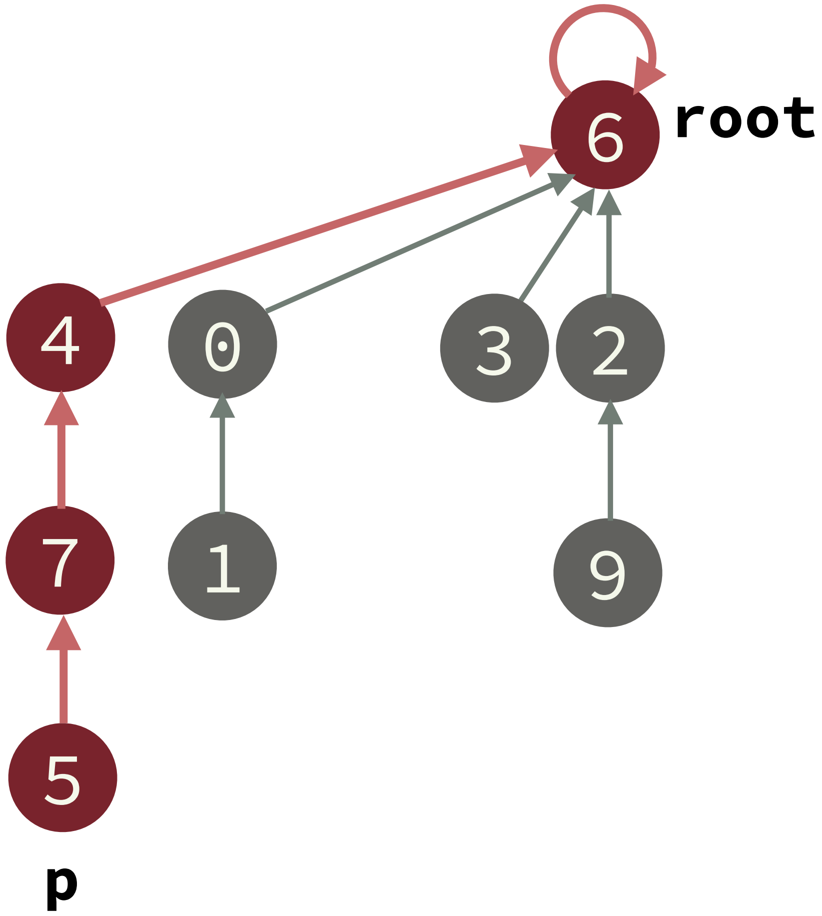
Idea. When **`FIND`** is called, attach directly to the root every node visited by **`FIND`**.

Rationale. Make use of work done anyway to speed up future calls to **`FIND`**.

`FIND(p)`

```
root = p
while (parent[root] != root)
    root = parent[root]

while (p != root)
    next = parent[p]
    parent[p] = root
    p = next

return p
```

Example. **`FIND(5)`**



Improvement Attempt 3: Path Compression

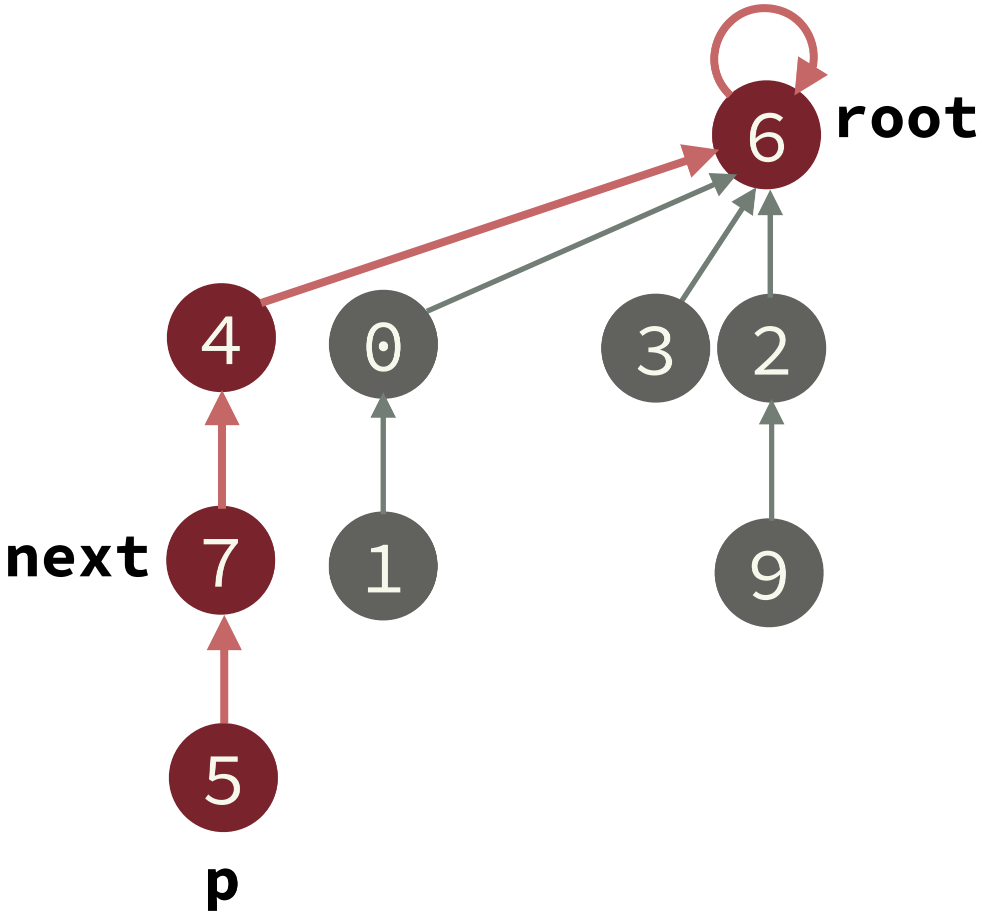
Idea. When **`FIND`** is called, attach directly to the root every node visited by **`FIND`**.

Rationale. Make use of work done anyway to speed up future calls to **`FIND`**.

`FIND(p)`

```
root = p
while (parent[root] != root)
    root = parent[root]

while (p != root)
    next = parent[p]
    parent[p] = root
    p = next

return p
```

Example. **`FIND(5)`**



Improvement Attempt 3: Path Compression

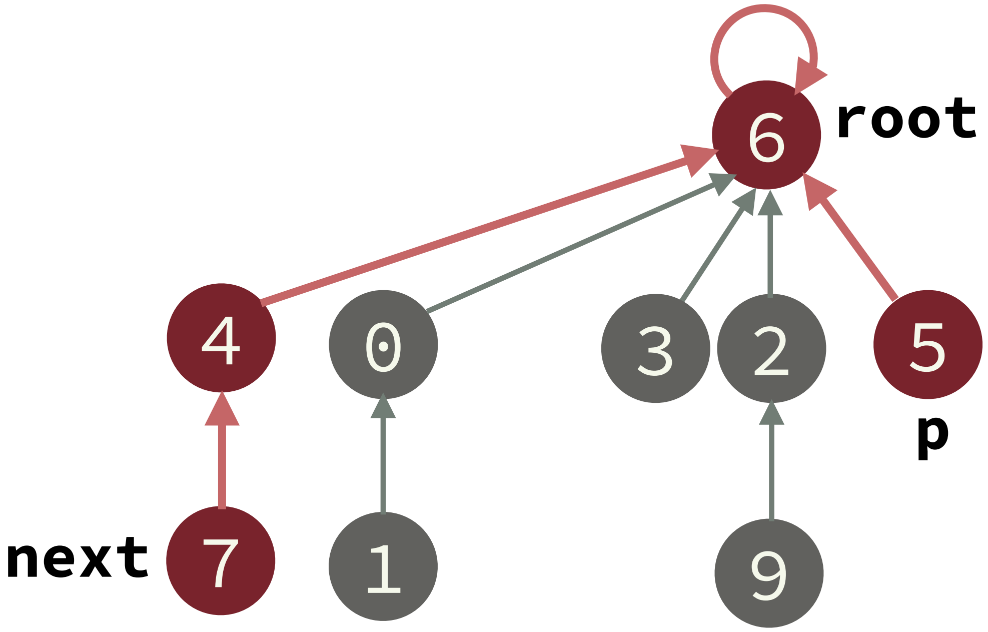
Idea. When **`FIND`** is called, attach directly to the root every node visited by **`FIND`**.

Rationale. Make use of work done anyway to speed up future calls to **`FIND`**.

`FIND(p)`

```
root = p
while (parent[root] != root)
    root = parent[root]

while (p != root)
    next = parent[p]
    parent[p] = root
    p = next

return p
```

Example. **`FIND`**`(5)`



Improvement Attempt 3: Path Compression

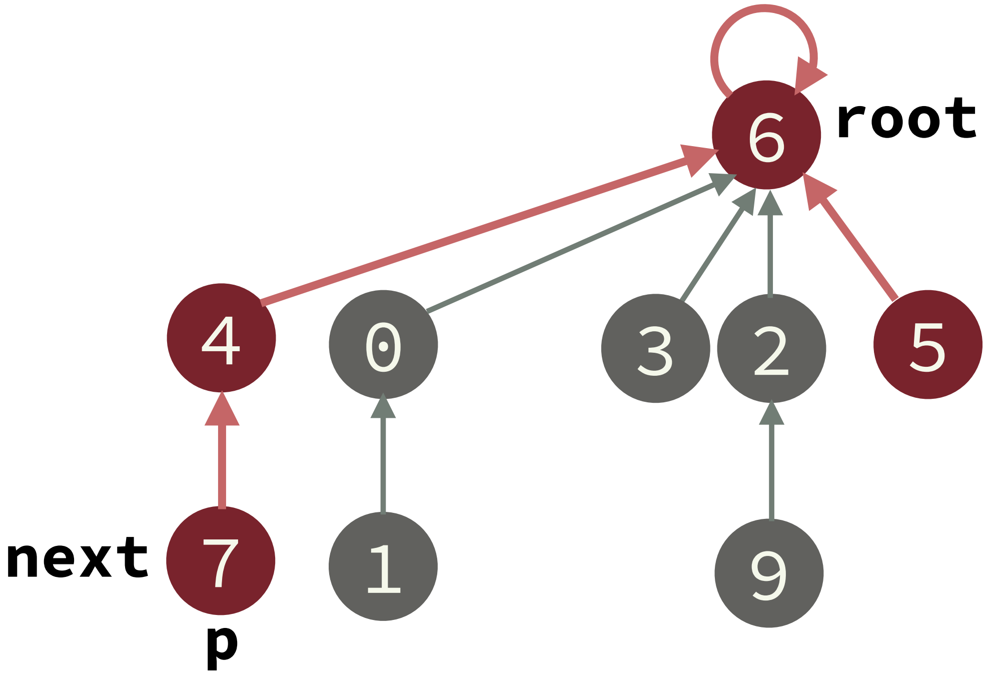
Idea. When **`FIND`** is called, attach directly to the root every node visited by **`FIND`**.

Rationale. Make use of work done anyway to speed up future calls to **`FIND`**.

`FIND(p)`

```
root = p
while (parent[root] != root)
    root = parent[root]

while (p != root)
    next = parent[p]
    parent[p] = root
    p = next

return p
```

Example. **`FIND(5)`**



Improvement Attempt 3: Path Compression

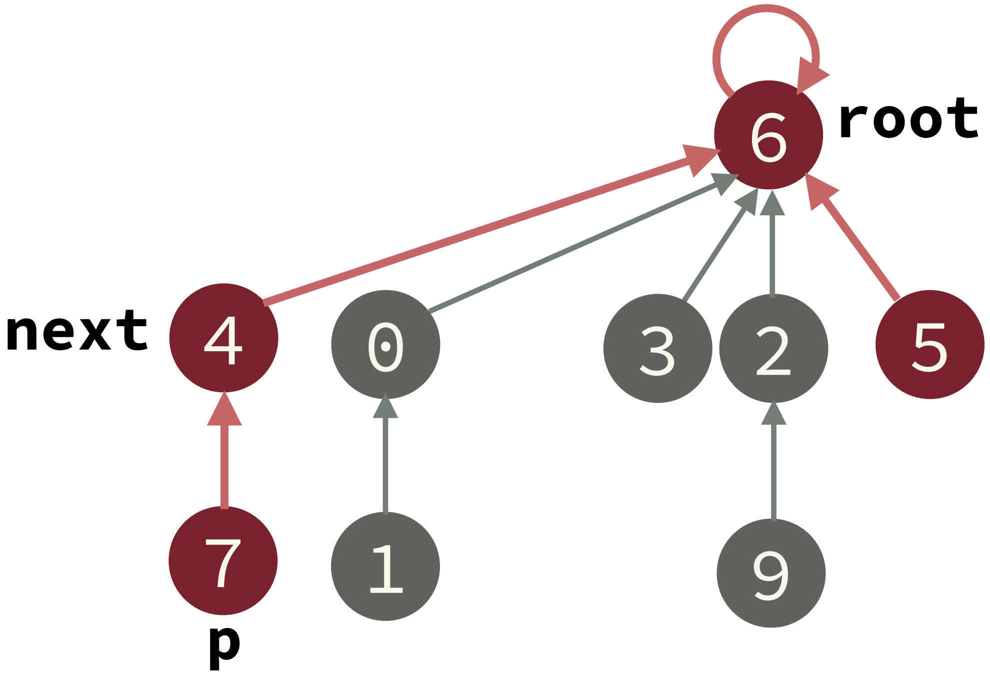
Idea. When **`FIND`** is called, attach directly to the root every node visited by **`FIND`**.

Rationale. Make use of work done anyway to speed up future calls to **`FIND`**.

`FIND(p)`

```
root = p
while (parent[root] != root)
    root = parent[root]

while (p != root)
    next = parent[p]
    parent[p] = root
    p = next

return p
```

Example. **`FIND(5)`**



Improvement Attempt 3: Path Compression

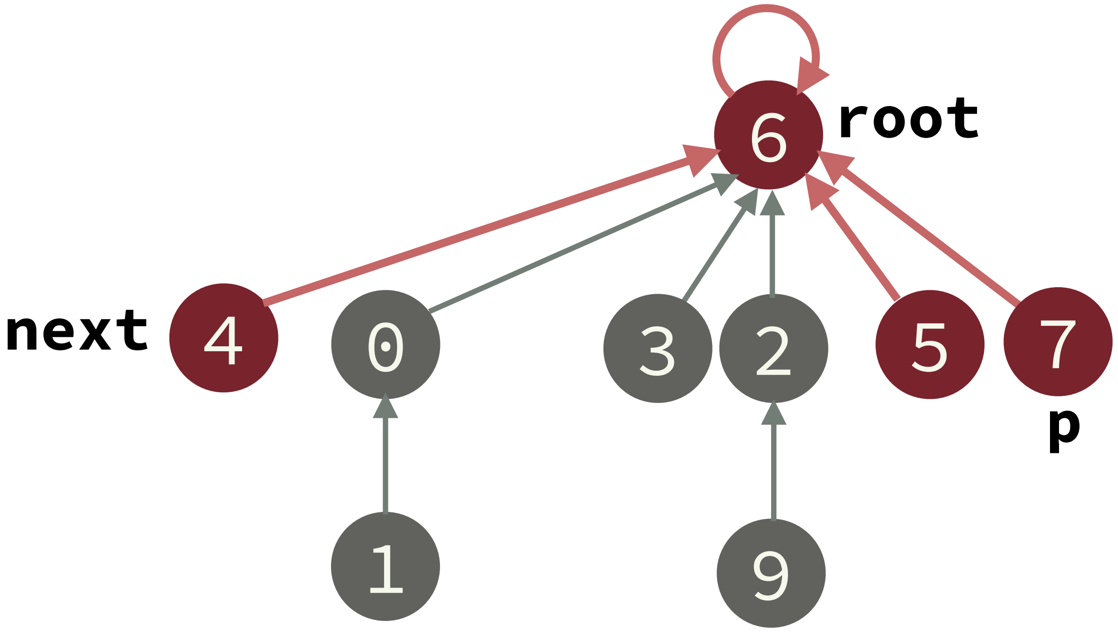
Idea. When **`FIND`** is called, attach directly to the root every node visited by **`FIND`**.

Rationale. Make use of work done anyway to speed up future calls to **`FIND`**.

`FIND(p)`

```
root = p
while (parent[root] != root)
    root = parent[root]

while (p != root)
    next = parent[p]
    parent[p] = root
    p = next

return p
```

Example. **`FIND(5)`**



Improvement Attempt 3: Path Compression

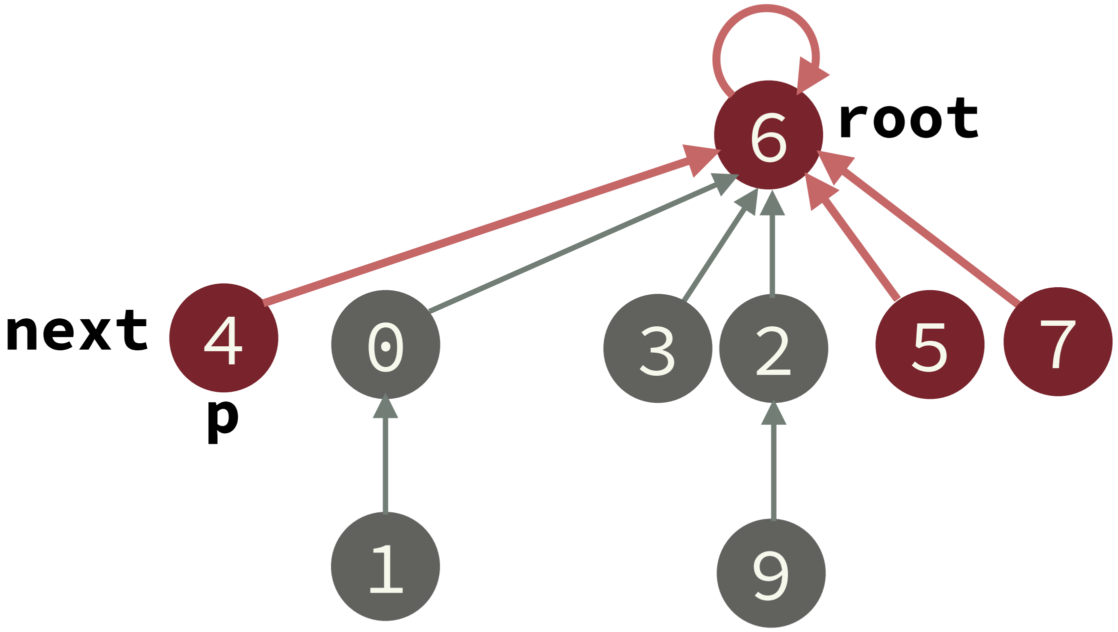
Idea. When **`FIND`** is called, attach directly to the root every node visited by **`FIND`**.

Rationale. Make use of work done anyway to speed up future calls to **`FIND`**.

`FIND(p)`

```
root = p
while (parent[root] != root)
    root = parent[root]

while (p != root)
    next = parent[p]
    parent[p] = root
    p = next

return p
```

Example. **`FIND(5)`**



Improvement Attempt 3: Path Compression

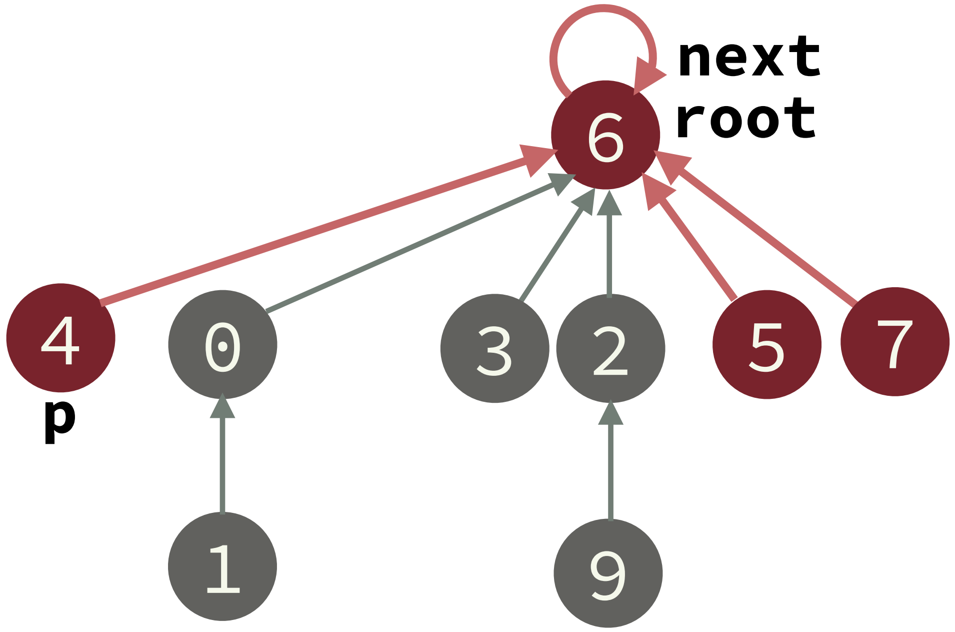
Idea. When **`FIND`** is called, attach directly to the root every node visited by **`FIND`**.

Rationale. Make use of work done anyway to speed up future calls to **`FIND`**.

`FIND(p)`

```
root = p
while (parent[root] != root)
   root = parent[root]

while (p != root)
   next = parent[p]
   parent[p] = root
   p = next

return p
```

Example. **`FIND(5)`**



Improvement Attempt 3: Path Compression

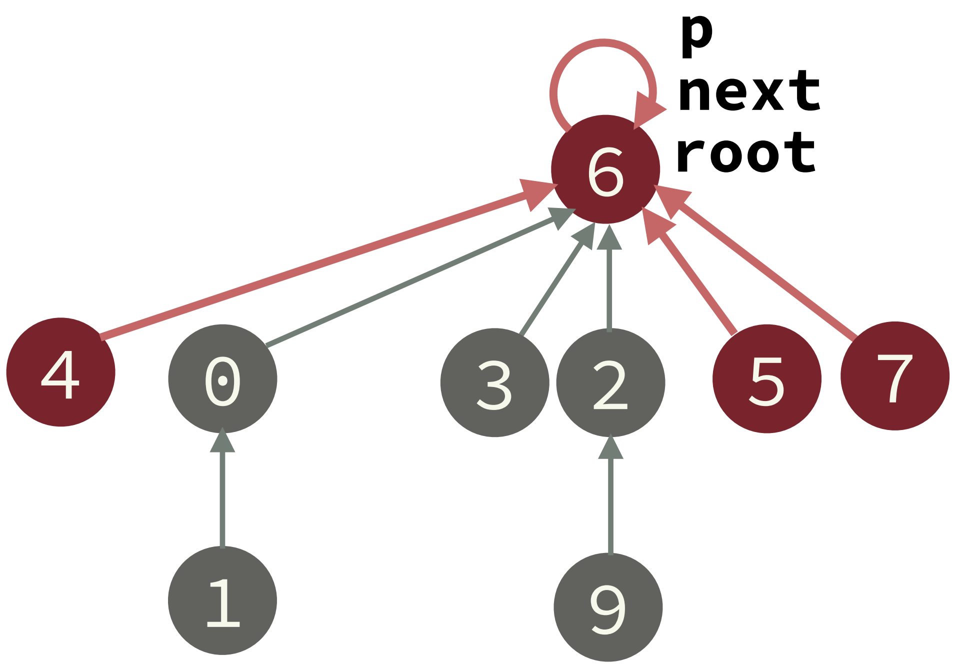
Idea. When **`FIND`** is called, attach directly to the root every node visited by **`FIND`**.

Rationale. Make use of work done anyway to speed up future calls to **`FIND`**.

`FIND(p)`

```
root = p
while (parent[root] != root)
    root = parent[root]

while (p != root)
    next = parent[p]
    parent[p] = root
    p = next

return p
```

Example. **`FIND(5)`**



🎉 The more expensive **`FIND`** operations are performed, the flatter the tree becomes!

Improvement Attempt 3: Path Compression

Theorem. A sequence of n **`UNION`** and **`FIND`** operations on a set of n singleton sets runs in $O(n \cdot \alpha(n))$.

Improvement Attempt 3: Path Compression

Theorem. A sequence of n **`UNION`** and **`FIND`** operations on a set of n singleton sets runs in $O(n \cdot \alpha(n))$.

Inverse Ackermann function
an extremely slowly growing function

😵 $\alpha(n) \leq 3$ for any remotely imaginable value of n

Improvement Attempt 3: Path Compression

Theorem. A sequence of n `UNION` and `FIND` operations on a set of n singleton sets runs in $O(n \cdot \alpha(n))$.

Hence, the running time of `UNION` and `FIND` is $O(\alpha(n))$ amortized.

Improvement Attempt 3: Path Compression

Theorem. A sequence of n **`UNION`** and **`FIND`** operations on a set of n singleton sets runs in $O(n \cdot \alpha(n))$.

Hence, the running time of **`UNION`** and **`FIND`** is $O(\alpha(n))$ amortized.

> Note. Although $\alpha(n) \leq 3$ for any remotely imaginable value of n, it is monotonically increasing and is eventually larger than any constant. Therefore, the running time is not $O(1)$ in theory but can be considered $O(1)$ in practice.

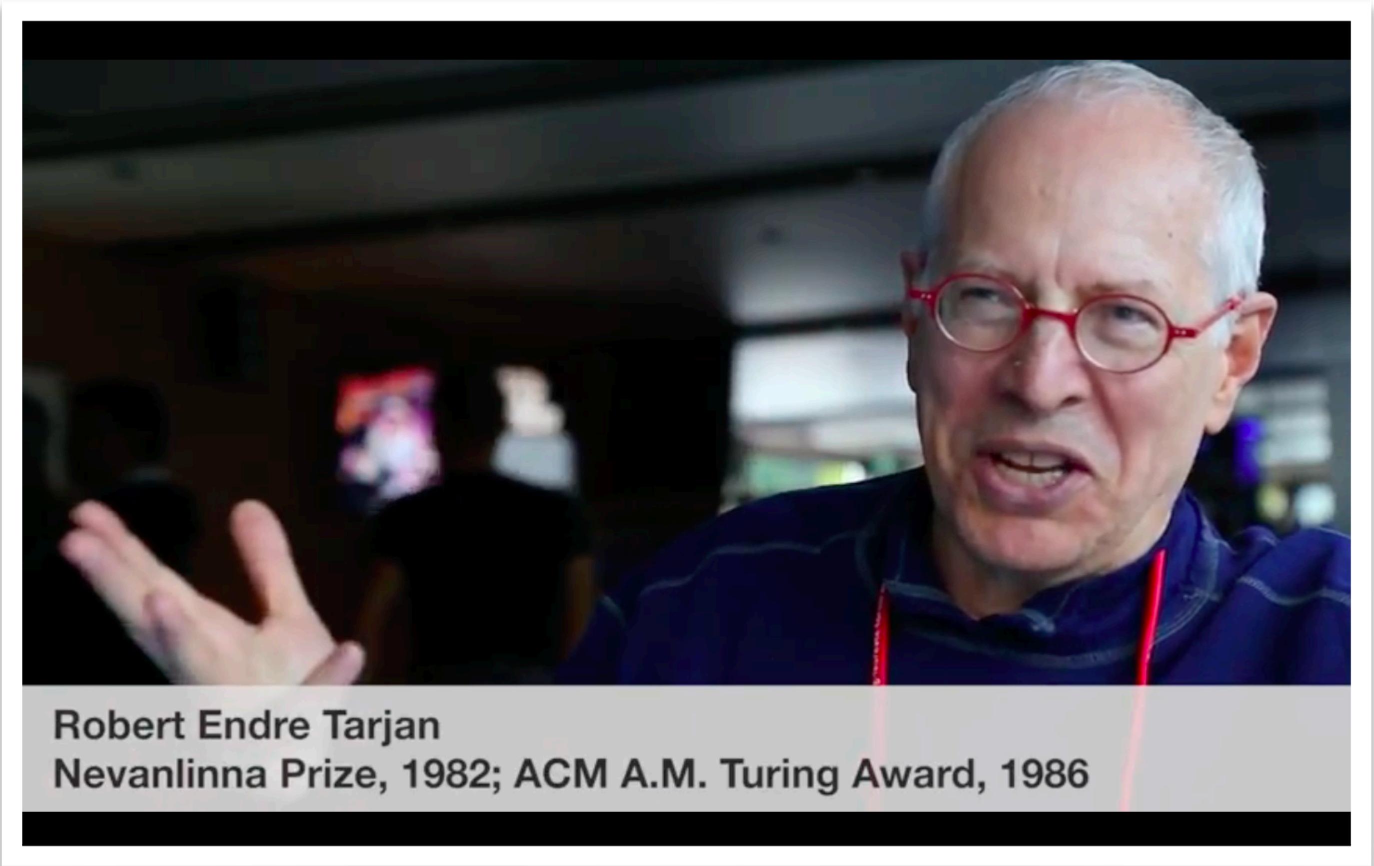
Improvement Attempt 3: Path Compression

Theorem. A sequence of n **`UNION`** and **`FIND`** operations on a set of n singleton sets runs in $O(n \cdot \alpha(n))$.

Hence, the running time of **`UNION`** and **`FIND`** is $O(\alpha(n))$ amortized.

> Note. Although $\alpha(n) \leq 3$ for any remotely imaginable value of n, it is monotonically increasing and is eventually larger than any constant. Therefore, the running time is not $O(1)$ in theory but can be considered $O(1)$ in practice.

Proof. Ask Robert Tarjan.



Improvement Attempt 3: Path Compression

Theorem. A sequence of n **`UNION`** and **`FIND`** operations on a set of n singleton sets runs in $O(n \cdot \alpha(n))$.

Hence, the running time of **`UNION`** and **`FIND`** is $O(\alpha(n))$ amortized.

> Note. Although $\alpha(n) \leq 3$ for any remotely imaginable value of n, it is monotonically increasing and is eventually larger than any constant. Therefore, the running time is not $O(1)$ in theory but can be considered $O(1)$ in practice.

Proof. Ask Robert Tarjan.

Can we do better? No.
$O(\alpha(n))$ is optimal.

Improvement Attempt 3: Path Compression

Theorem. A sequence of n **`UNION`** and **`FIND`** operations on a set of n singleton sets runs in $O(n \cdot \alpha(n))$.

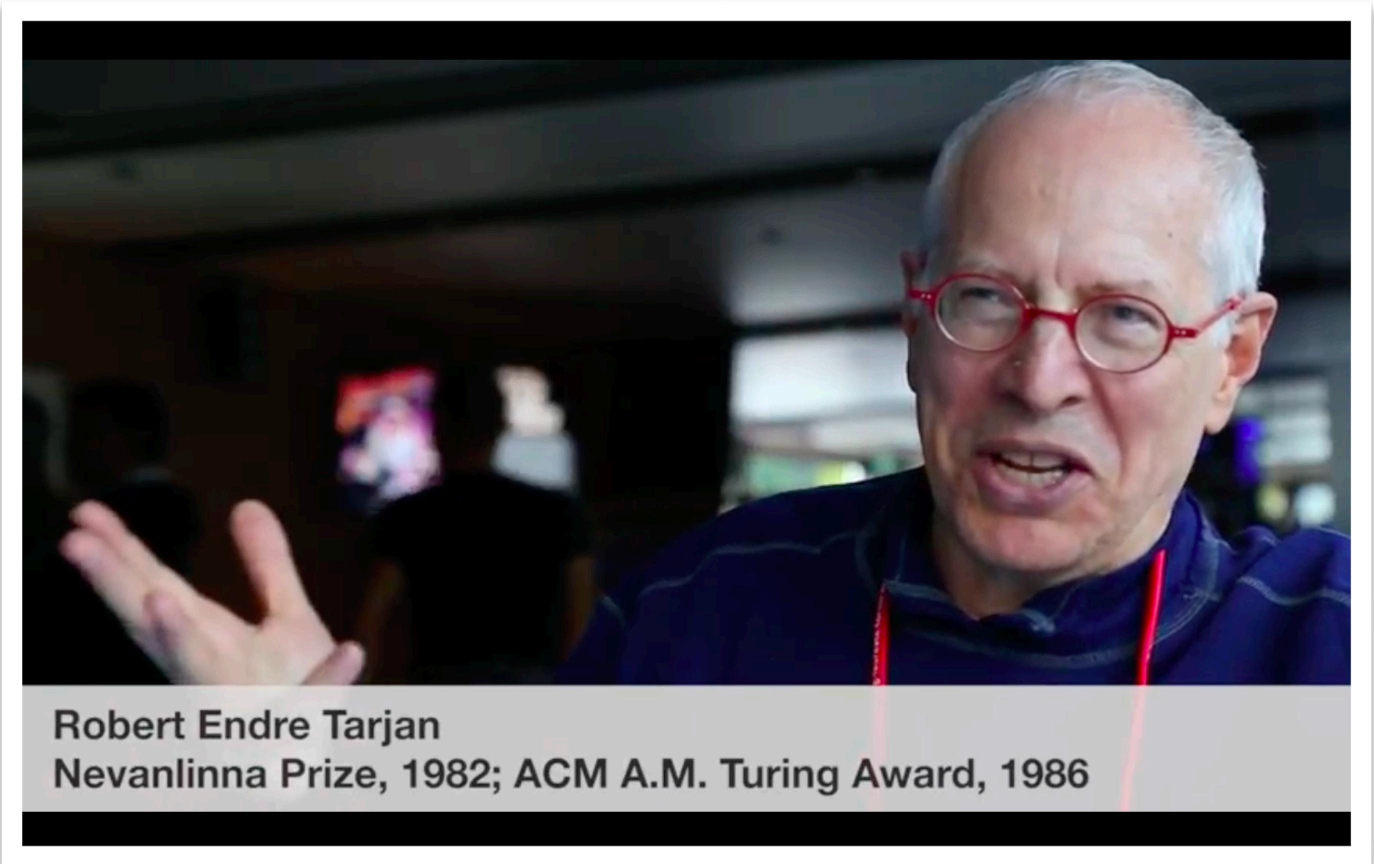
Hence, the running time of **`UNION`** and **`FIND`** is $O(\alpha(n))$ amortized.

> Note. Although $\alpha(n) \leq 3$ for any remotely imaginable value of n, it is monotonically increasing and is eventually larger than any constant. Therefore, the running time is not $O(1)$ in theory but can be considered $O(1)$ in practice.

Proof. Ask Robert Tarjan.

Can we do better? No.
$O(\alpha(n))$ is optimal.

Proof. Ask Robert Tarjan.



Robert Endre Tarjan
Nevanlinna Prize, 1982; ACM A.M. Turing Award, 1986

Improvement Attempt 3: Path Compression

Theorem. A sequence of n `UNION` and `FIND` operations on a set of n singleton sets runs in $O(n \cdot \alpha(n))$.

Hence, the running time of `UNION` and `FIND` is $O(\alpha(n))$ amortized.

> Note. Although $\alpha(n) \leq 3$ for any remotely imaginable value of n, it is monotonically increasing and is eventually larger than any constant. Therefore, the running time is not $O(1)$ in theory but can be considered $O(1)$ in practice.

Proof. Ask Robert Tarjan.

Can we do better? No.
$O(\alpha(n))$ is optimal.

Proof. Ask Robert Tarjan.

Other Methods. Many!

E.g. Assign random indices to the elements and use them instead of the size in weighted quick-union (by size).

Result. Almost same performance!

Improvement Attempt 3: Path Compression

Theorem. A sequence of n **`UNION`** and **`FIND`** operations on a set of n singleton sets runs in $O(n \cdot \alpha(n))$.

Hence, the running time of **`UNION`** and **`FIND`** is $O(\alpha(n))$ amortized.

> Note. Although $\alpha(n) \leq 3$ for any remotely imaginable value of n, it is monotonically increasing and is eventually larger than any constant. Therefore, the running time is not $O(1)$ in theory but can be considered $O(1)$ in practice.

Proof. Ask Robert Tarjan.

Can we do better? No.
$O(\alpha(n))$ is optimal.

Proof. Ask Robert Tarjan.

Other Methods. Many!

E.g. Assign random indices to the elements and use them instead of the size in weighted quick-union (by size).

Result. Almost same performance!

Proof. Ask Robert Tarjan.

Improvement Attempt 3: Path Compression

Theorem. A sequence of n **`UNION`** and **`FIND`** operations on a set of n singleton sets runs in $O(n \cdot \alpha(n))$.

Hence, the running time of **`UNION`** and **`FIND`** is $O(\alpha(n))$ amortized.

> Note. Although $\alpha(n) \leq 3$ for any remotely imaginable value of n, it is monotonically increasing and is eventually larger than any constant. Therefore, the running time is not $O(1)$ in theory but can be considered $O(1)$ in practice.

Proof. Ask Robert Tarjan.

Can we do better? No.
$O(\alpha(n))$ is optimal.

Proof. Ask Robert Tarjan.

Other Optimizations.

Get rid of the `height[]` array in weighted quick-union.

How?

Other Methods. Many!

E.g. Assign random indices to the elements and use them instead of the size in weighted quick-union (by size).

Result. Almost same performance!

Proof. Ask Robert Tarjan.