Prepared by: [Arun Konagurthu]

FIT3155: Advanced Algorithms and Data Structures

Week 4: **The disjoint-set data structure**

Faculty of Information Technology, Monash University

What is covered in this lecture?

The disjoint-set data structure and its analysis

Source material and recommended reading

- Mark Weiss, Data Structures and Algorithm Analysis (Chapter 8).

Equivalence relationship

- A **relation** $\otimes$ is defined over members/elements of some set S.
- For any pair of elements (a, b) from this set S:
  - $a \otimes b$ results in a true or false answer.

What is an equivalence relation

An **equivalence relation** is a relation $\otimes$ that satisfies:

reflexive property: $a \otimes a$ for all a in set S

symmetric property: $a \otimes b$ implies $b \otimes a$ for all a, b in S

transitive property: $a \otimes b$ and $b \otimes c$ implies $a \otimes c$ for all a, b, c in S

Equivalence class

- An **equivalence class** of an element $a \in S$ defines a **subset** of elements from S, where all elements in that subset are **related** to a.
- Every element of S belongs to exactly one equivalence class (subset).
- To check if two elements a and b are related, we only have to check if they are in the same equivalence class.

Basic disjoint-set data structure

Disjoint set data structure supports two basic operations:

find(a): This returns the name/label of the subset (i.e. equivalence class) containing the element a in the set S.

- Note: the name/label of the subset itself is **arbitrary**.
- All that really matters is this: For two elements a and b to be related, we should check if **find**(a)==**find**(b).

union(a, b): Merge the two (disjoint) subsets containing a and b in S.

- In practice, this is implemented as **union**(**find**(a), **find**(b)).

The input to this data structure is initially a collection of N elements, that are treated to be disjoint (no relation) with each other. Using **find**$(\cdot)$ and **union**$(\cdot, \cdot)$ operations, the relations are dynamically checked and (new relations) established.

Some applications of Disjoint set data structure

- Kruskal's algorithm
- Keeping track of connected components of a graph
- Computing Lowest Common Ancestor in trees
- Checking equivalence of finite state automata
- Hindley-Milner polymorphic type inference
- etc.

RECALL FROM FIT2004?

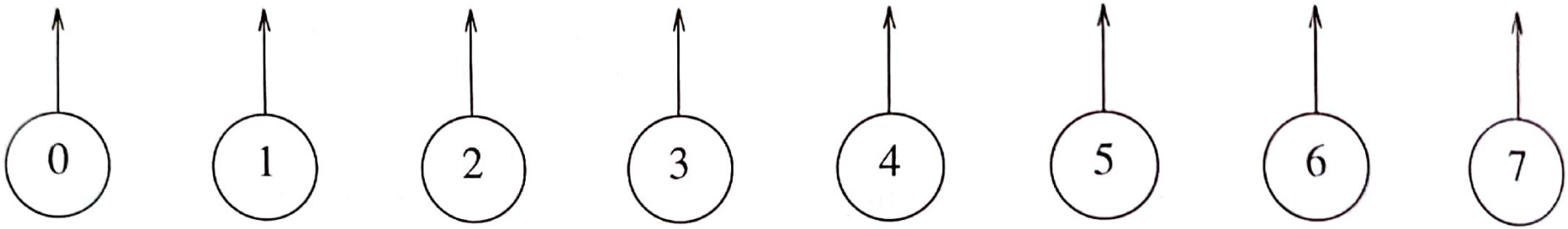
- Kruskal's algorithm introduced a basic implementation of disjoint-set data structure. – Revise, if forgotten!
- This involved maintaining the disjoint data-structure using:
  1. an **array of linked-lists** to support **union**(a, b).
  2. a **membership array** to support **find**(a).

RECALL FROM FIT2004? – continued

Using the implementation on previous slide:

- **find**(a) operation can be achieved via array access in $O(1)$-time.
- **union**(a, b) operation can be achieved in $O(N)$-time, because it requires:
  - appending two linked lists (each denoting a subset being merged)
  - change **membership** array for elements in the **smaller** of the two subsets, so that they are now merged.

New disjoint set data structure using just an array

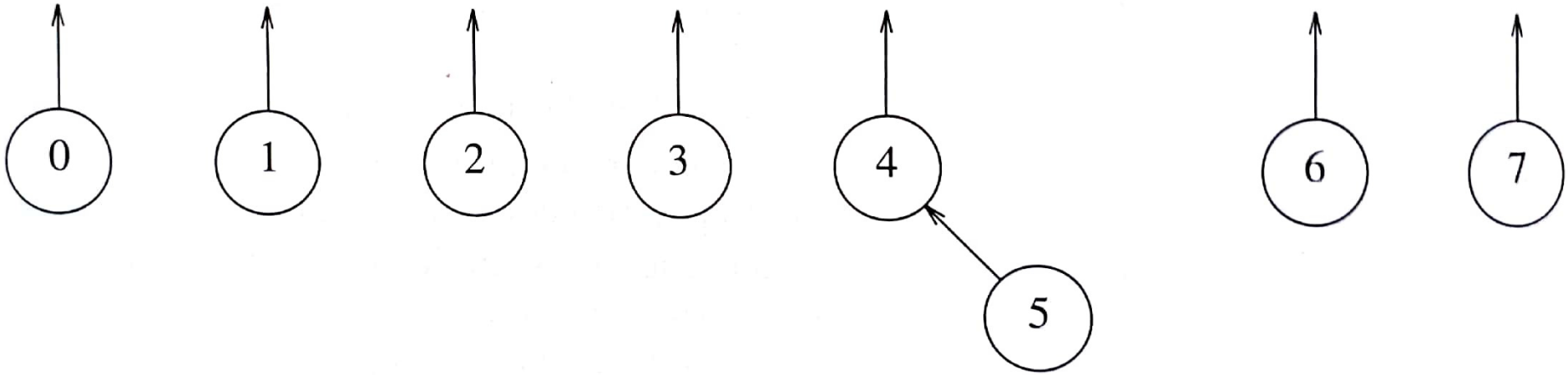


Implicit representation as an array with parent labels

array index	0	1	2	3	4	5	6	7
parent array	-1	-1	-1	-1	-1	-1	-1	-1

- The above example shows $N = 8$ disjoint elements initially.
- These are numbered $\{0, 1, 2, \cdots, 7\}$.
- The arrows in the figure represent some imaginary parent.
- A simple **parent array** is used to capture this information.
- In the **parent array**, the imaginary parent is denoted as -1.
- In general, each element in the array points to its parent.

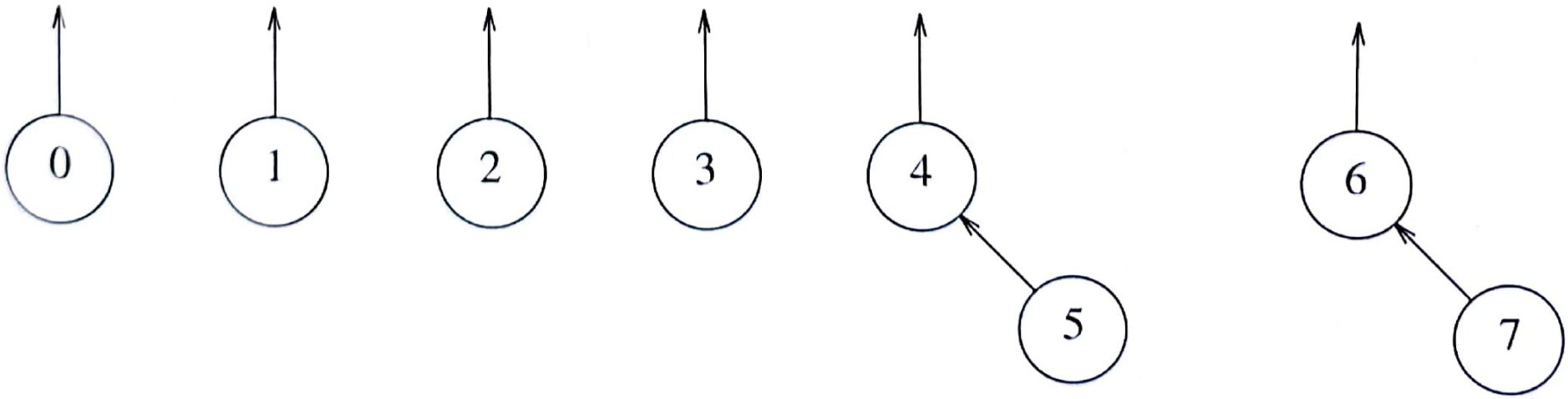
Example: Data structure after **union**$(4, 5)$ operation



Implicit representation as an array with parent labels

array index	0	1	2	3	4	5	6	7
parent array	-1	-1	-1	-1	-1	4	-1	-1

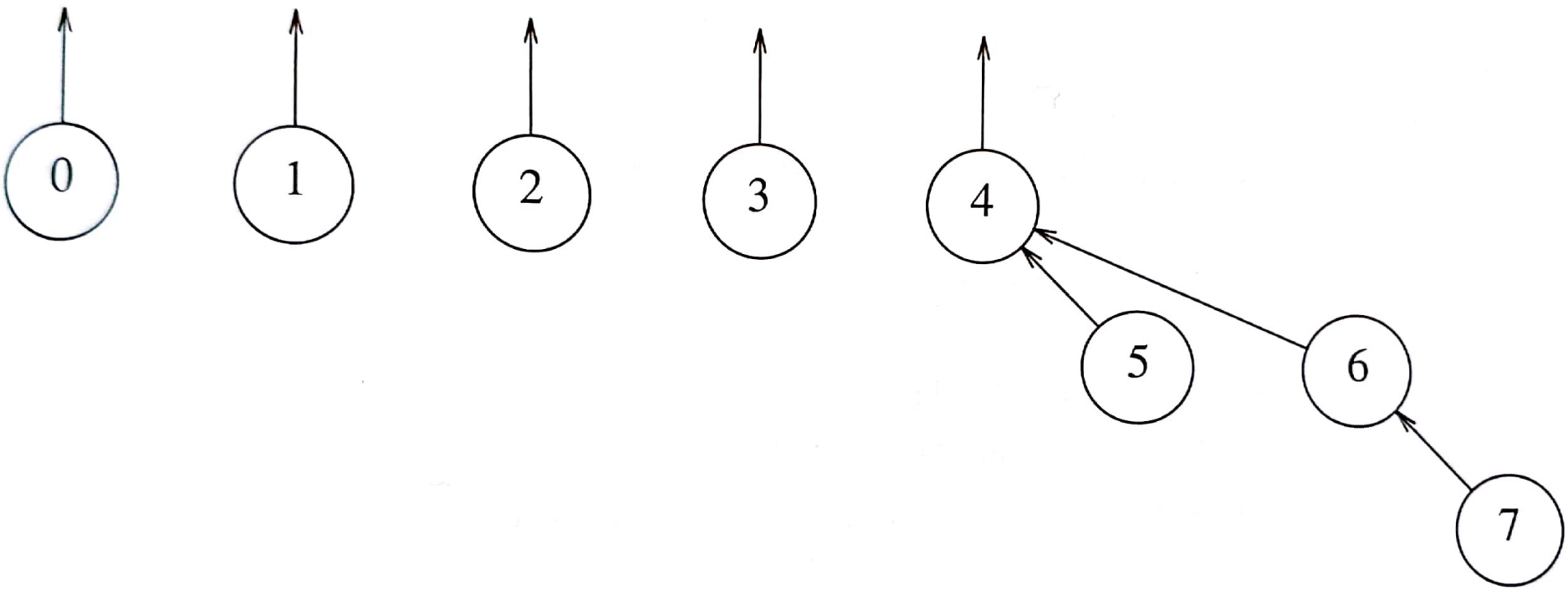
Example: Data structure after **union**$(6, 7)$ operation



Implicit representation as an array with parent labels

array index	0	1	2	3	4	5	6	7
parent array	-1	-1	-1	-1	-1	4	-1	6

Example: Data structure after **union**$(4, 6)$ operation



Implicit representation as an array with parent labels

array index	0	1	2	3	4	5	6	7
parent array	-1	-1	-1	-1	-1	4	4	6

Smart Union algorithms – Motivation

- `union`$(\cdot)$ in the earlier examples performed a union by making the second tree (subset) a subtree of the first.
- This was really an arbitrary choice.
- In fact, in the examples on slides 11-14, the subsets being merged ("unioned") were of the **same** size. So it did not matter.
- But in general, a smarter way would be to make the **smaller** tree (in terms of the number of elements in it) the subtree of the larger tree. This is called `union-by-size`.
- An even smarter approach would be to make the **shorter**/shallower tree (in tree height) the subtree of the **taller**/deeper tree. This is called `union-by-height`.

Union by size

Union by size – initialization

- At the start, initialize N **disjoint subsets** of elements.
- This involves initializing the **parent array** to all -1 values.
- In **union-by-size**, in general, the cell in the **parent array**, corresponding to any (disjoint) tree's root node, stores the **size of the tree** as a **negative number**.

Initialization

```
1    InitSet(N) {
2        for (a = 0 to N-1) {
3            Make_disjoint_set(a)
4        }
5    }
6
7    Make_disjoint_set(x) {
8        parent[x] = -1
9    }
```

Union by size – **find**(a) implementation

Search until the **root** of the tree (containing 'a') is reached; return the root's label/index.

find(a)

```
1    find(a) {
2       // find root of the tree containing 'a'
3       while (parent[a] >= 0) { // note: while parent[a] not negative
4            a = parent[a]
5        }
6        return a  // 'a' now indexes the root node of the subtree/subset
7    }
```

Union by size – **union**(a, b) implementation

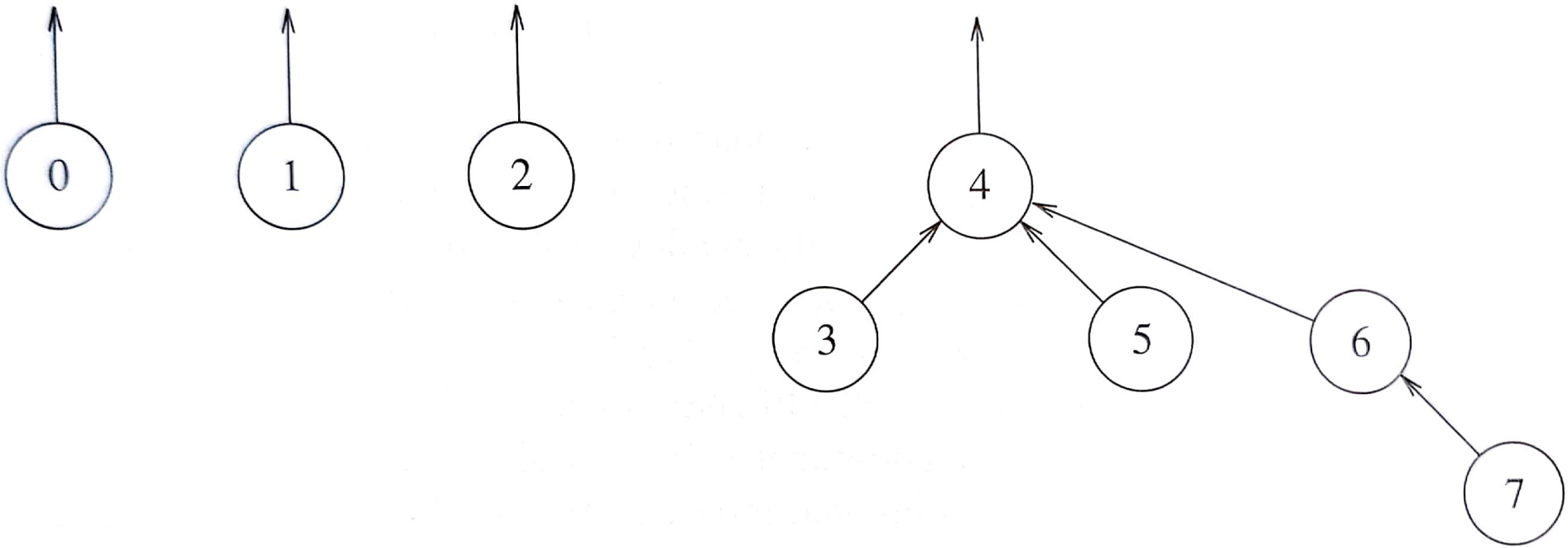
When executing **union**(a, b), ensure a and b are in two disjoint trees. Then link the root node of the **smaller** (in size) tree to the root for the larger one. If both sizes are equal, break the tie **arbitrarily**. Update the merged tree size (stored as a negative number)

union(a, b) – by size

```
1 union(a,b) {
2     root_a = find(a) // find root of tree containing 'a'
3     root_b = find(b) // find root of tree containing 'b'
4     if (root_a == root_b) return  // 'a' and 'b' in the same tree
5
6     size_a = -parent[root_a]
7     size_b = -parent[root_b]
8
9     if (size_a > size_b) {
10        parent[root_b] = root_a   // link smaller tree's root to larger
11        parent[root_a] = -(size_a+size_b) // update size
12    }
13    else // if (size_b >= size_a)
14        parent[root_a] = root_b
15        parent[root_b] = -(size_a+size_b)
16    }
17 }
```

Union by size Example

Refer to slides 11-14. Now redo the same set of unions, this time using union-by-size, and on top of them, do a union (by-size) of **union**$(3, 7)$. We get to this state/representation:



Implicit representation as an array with parent labels

array index	0	1	2	3	4	5	6	7
parent array	-1	-1	-1	4	-5	4	4	6

Union by height (or rank)

Union by height (or rank) – initialization

In union by height/rank, the cell in the **parent array** corresponding to the root element of any disjoint tree/set, **now** stores the **height of the tree** coded as a negative number.*

Initialization

```
1    InitSet(N) {
2        for (a = 0 to N-1) {
3            Make_disjoint_set(a)
4        }
5    }
6
7    Make_disjoint_set(x) {
8        parent[x] = -1
9    }
```

*Strictly speaking, this is storing height $+ 1$ as a negative number, because height $= 0$ cannot be coded as a negative number.

Union by height – **find**(a) implementation

Exactly same as for union by size (see slide 18)

Union by height (or rank) – **union**(a, b) implementation

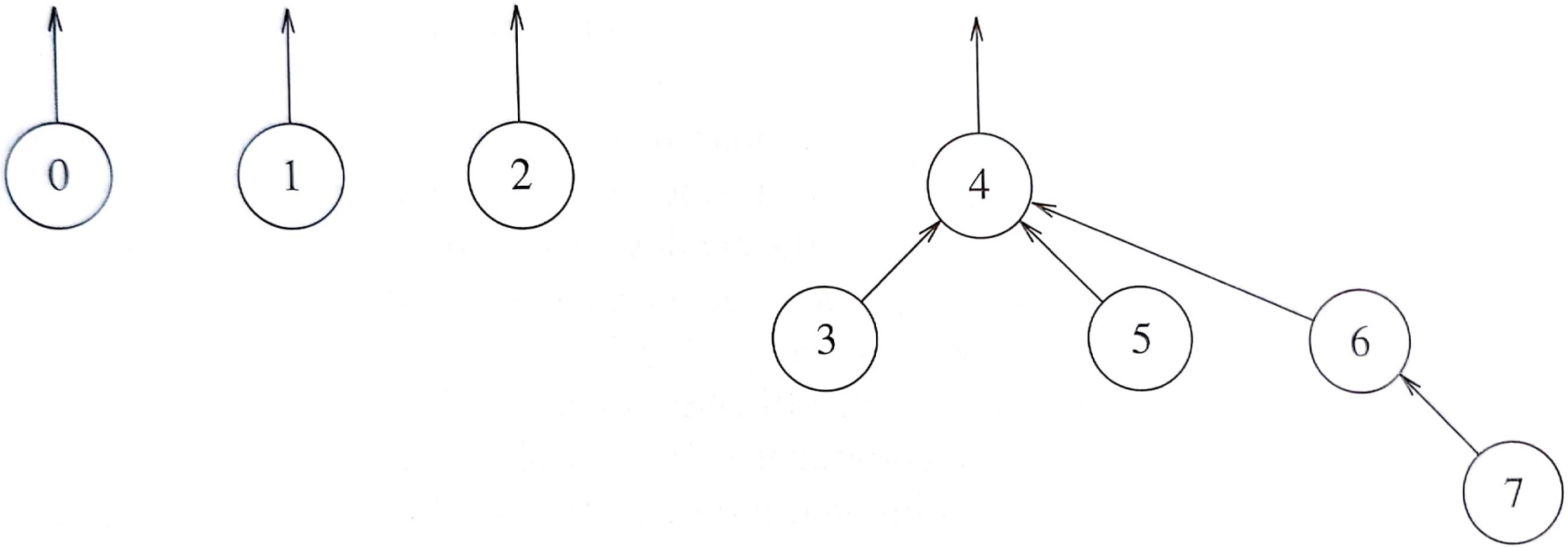
When executing **union**(a, b), ensure a and b are in two disjoint trees. Then link the root node of the shorter (in **height**) tree to the root of the taller tree. If both heights are equal, break the tie arbitrarily. Notice below how the logic of update to the height of the merged tree (stored at the root as a negative number) now is differs (compared to union-by-size).

union(a, b)

```
1 union(a,b) {
2     root_a = find(a) // find root of tree containing 'a'
3     root_b = find(b) // find root of tree containing 'b'
4     if (root_a == root_b) return  // 'a' and 'b' in the same tree
5
6     height_a = -parent[root_a] // height of tree containing 'a'
7     height_b = -parent[root_b] // height of tree containing 'b'
8     if (height_a > height_b) {
9         parent[root_b] = root_a // link shorter tree's root to taller
10    }
11    else if (height_b > height_a) {
12        parent[root_a] = root_b
13    }
14    else { // if (height_a == height_b)
15        parent[root_a] = root_b
16        parent[root_b] = -(height_b+1) // update to height
17    }
18 }
```

Union by height (or rank) – Example

Refer to slides 11-14. Now redo the same set of unions this time using union-by-**height**, and on top of them, do a union (by-**height**) of **union**$(3, 7)$. We get to this state/representation:



Implicit representation as an array with parent labels

array index	0	1	2	3	4	5	6	7
parent array	-1	-1	-1	4	-3	4	4	6

Additional optimization

Path Compression (performed during **find**$(\cdot)$operation)

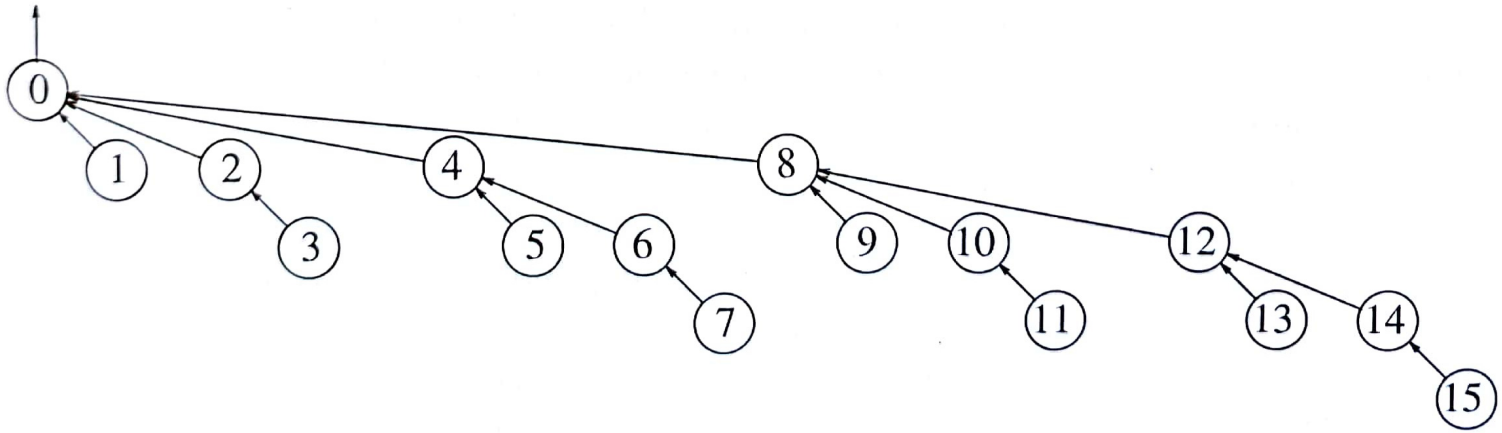
Path Compression during **find**$(\cdot)$operation

- Path compression is performed during **find**$(\cdot)$operation.
- It is independent of the strategy used for **union**$(\cdot, \cdot)$ operation.
- But for subsequent discussion, assume we are using **union**$(\cdot, \cdot)$ using union by height/rank.
- When executing **find**(a), for any a, every node along the path from the 'root' node to 'a' has its **parent index change to point to the root**.

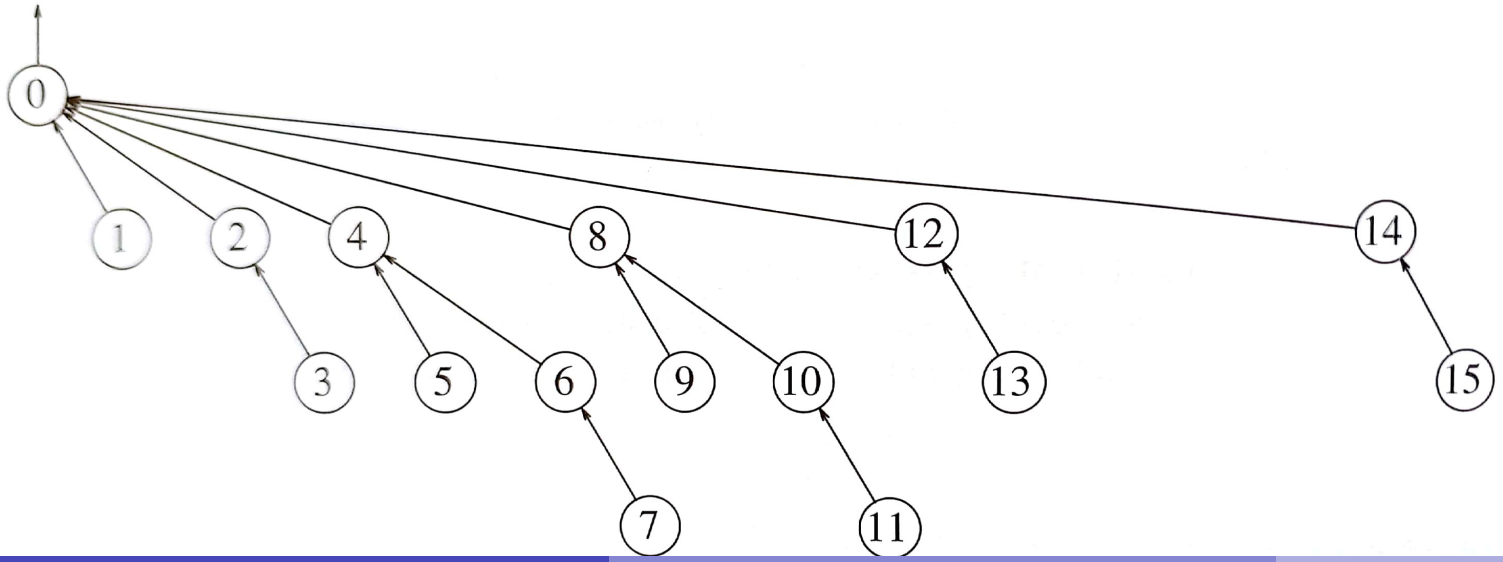
Path compression illustration

Consider this example (note completely different from those before).

Before **`path compression`**



Say we are running **`find`**(14) on the above state. The resultant path compressed graph will be:



Path compression based **find**(a) implementation

After finding the root of the tree containing 'a', change the parent pointer of all nodes along the path to point directly to the root.

find(a) – implementing path compression

```
1    find(a) {
2        // find root of the tree containing 'a'
3        if (parent[a] < 0)  { // root is reached
4            return a
5        }
6        else {
7            return parent[a] = find(parent[a])
8        }
9    }
```

Algorithmic analysis of Disjoint set data structure

To be discussed during the lecture (pen on paper); take notes. Analysis part will be uploaded on Moodle as a separate document.

Other variants of path compression –not examinable!

path splitting: Every node along the path points to its **grandparent**

path halving: Every alternate node along the path points to its **grandparent**