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

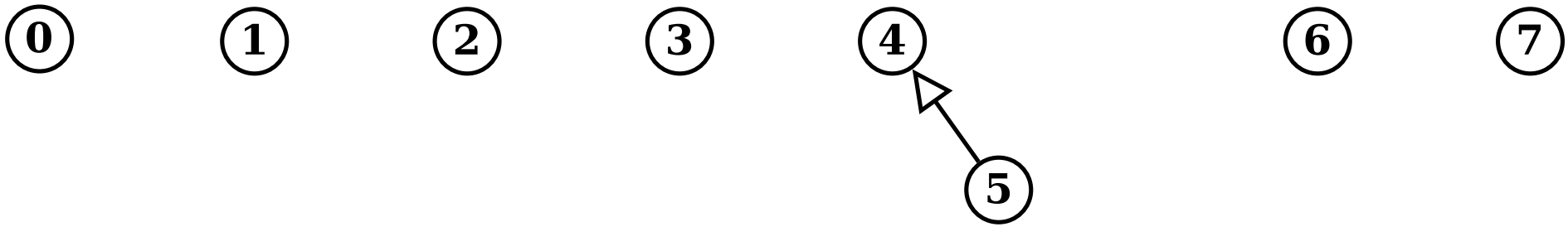
Implicit representation as an array with parent labels

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 $\{0, 1, 2, \cdots, 7\}$.
- 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 a series of **union**(.) operations

- **union**$(4, 5)$
- **union**$(6, 7)$
- **union**$(5, 7)$

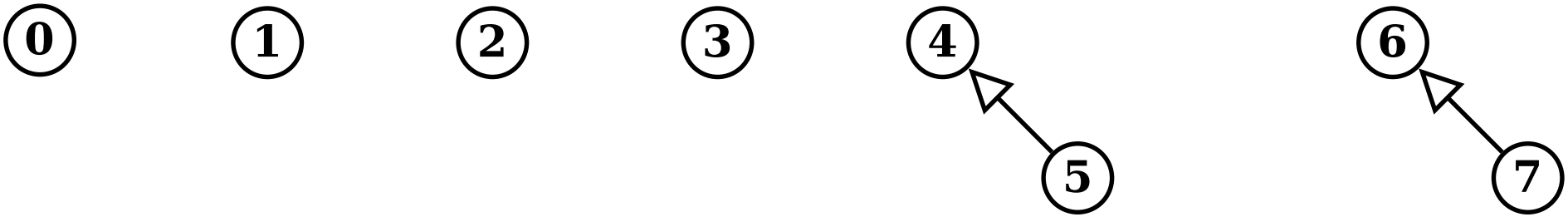


Implicit representation as an array with parent labels

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

Example: Data structure after a series of **union**(.) operations

- **union**$(4, 5)$
- **union**$(6, 7)$
- **union**$(5, 7)$

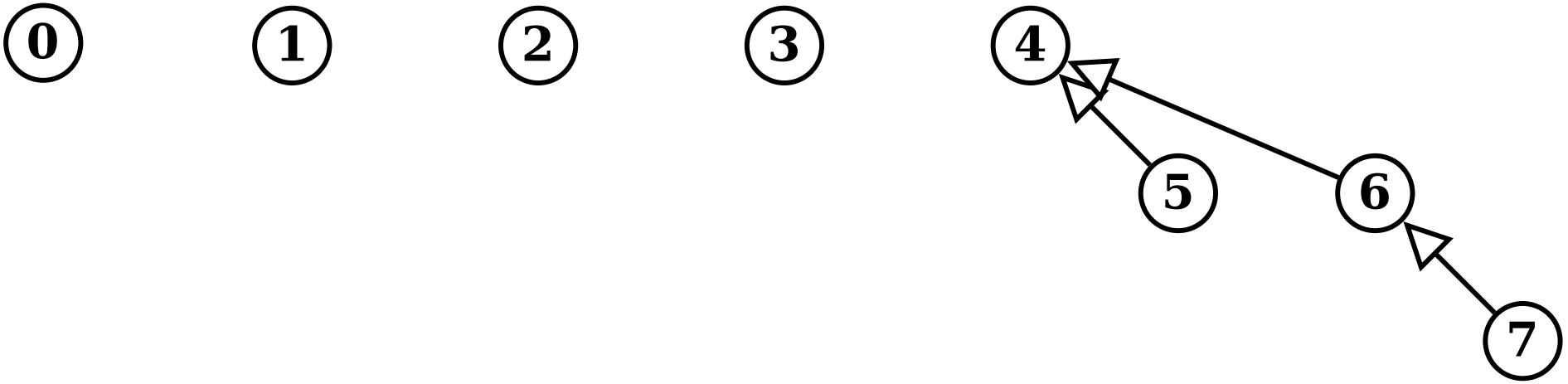


Implicit representation as an array with parent labels

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

Example: Data structure after a series of **union**(.) operations

- **union**$(4, 5)$
- **union**$(6, 7)$
- **union**$(5, 7)$



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.

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.

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 (#11-12), the subsets being merged ("union"-ed) were of the **same** size. So it did not matter.

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 (#11-12), the subsets being merged ("union"-ed) 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`**.

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 (#11-12), the subsets being merged ("union"-ed) 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

When **union**(a, b) is carried out using union-by-**size**:

- Ensure a and b are in two disjoint trees.
- Then point the root node of the tree with **smaller number of elements** to the root for the larger one.
  - If both sizes are equal, break the tie **arbitrarily**.
- The cell corresponding to the merged root in the `parent array` is updated to store the merged tree/subset size (as a negative number)

`union-by-size` example

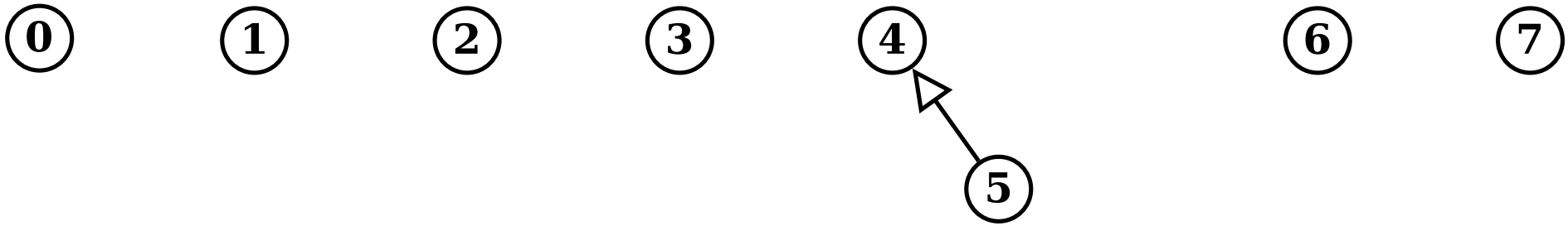
- initial state
- `union`$(4, 5)$
- `union`$(6, 7)$
- `union`$(5, 7)$
- `union`$(3, 7)$

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

union-by-size example

- initial state
- **union**$(4, 5)$
- **union**$(6, 7)$
- **union**$(5, 7)$
- **union**$(3, 7)$

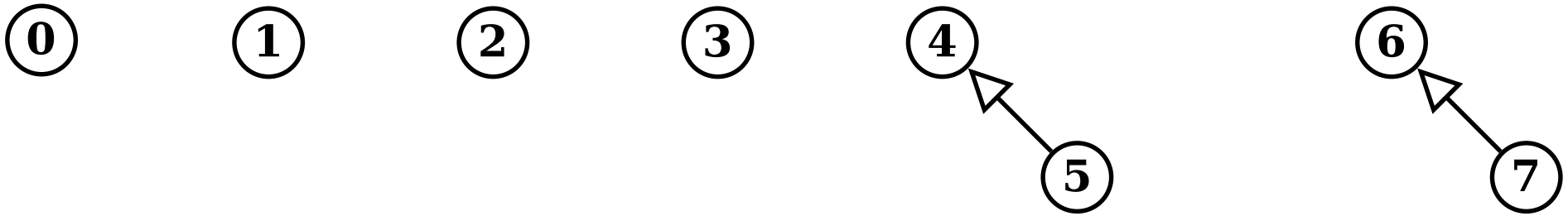


Implicit representation as an array with parent labels

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

union-by-size example

- initial state
- **union**$(4, 5)$
- **union**$(6, 7)$
- **union**$(5, 7)$
- **union**$(3, 7)$

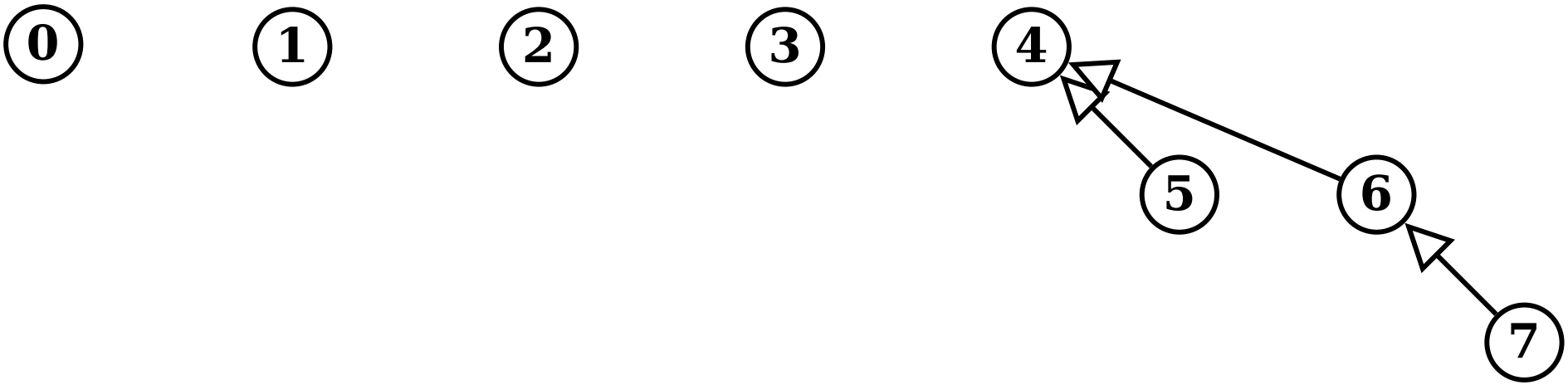


Implicit representation as an array with parent labels

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

`union-by-size` example

- initial state
- `union`$(4, 5)$
- `union`$(6, 7)$
- `union`$(5, 7)$
- `union`$(3, 7)$

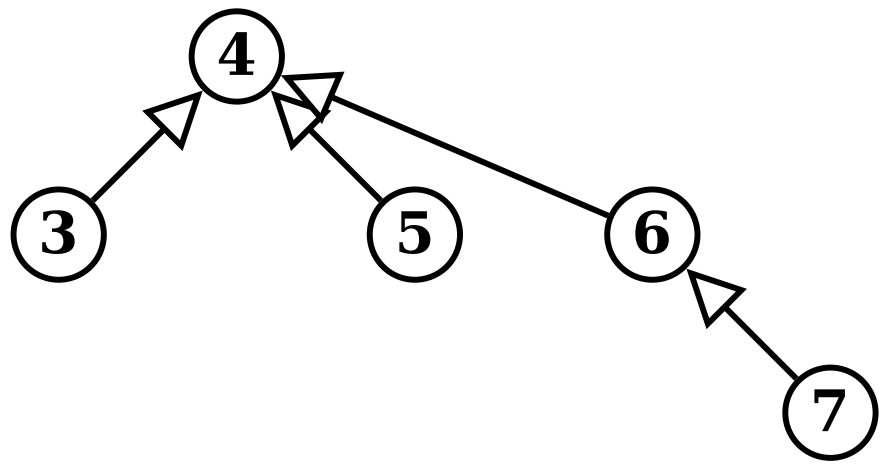


Implicit representation as an array with parent labels

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

union-by-size example

- initial state
- **union**$(4, 5)$
- **union**$(6, 7)$
- **union**$(5, 7)$
- **union**$(3, 7)$



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

Implementation of **Union-by-size**: Initialization

- At the start, initialize N **disjoint subsets** of elements.
- This involves initializing the **parent array** to all -1 values.
- Recall: In **union-by-size**, 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    }
```

Implementation of **Union-by-size**: – **find**(a)

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    }
```

Implementation of **Union-by-size**: – **union**(a, b)

Recall: 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-height (and union-by-rank)

When **union**(a, b) is carried out using union-by-**height**:

- Ensure a and b are in two disjoint trees.
- Then make the disjoint tree with **shorter height** a subtree of the taller tree.
- That is, make the root of the shorter tree point to the root of the taller one, after union.
  - If both heights are equal, break the tie arbitrarily.
- The cell corresponding to the merged root in the `parent array` is updated to store the merged tree's (**height** $+1$) as a negative number.*

A quick note on Union-by-rank

Union-by-rank is essentially identical to union-by-height, but includes an additional optimization called `path compression` to be discussed later on, on slide #24.

*Note: We store **height** $+ 1$ because **height** can be 0 (for a tree with a singleton node) and hence cannot be coded as a negative number in the parent array.

union-by-height example

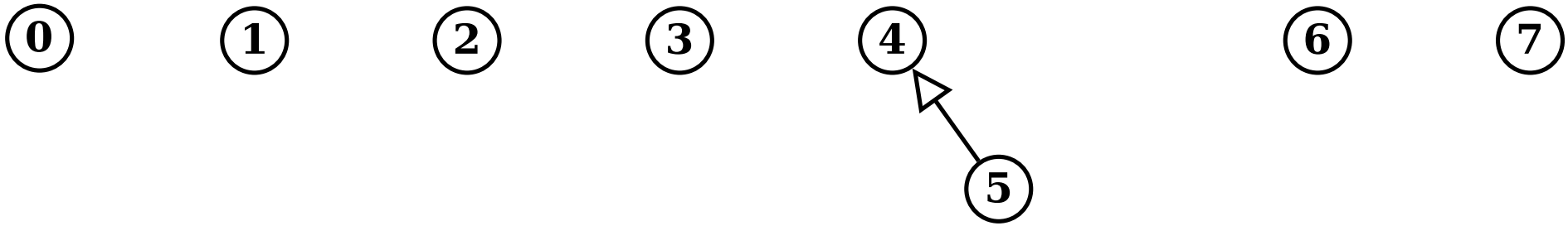
- initial state
- union$(4, 5)$
- union$(6, 7)$
- union$(5, 7)$
- union$(3, 7)$

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

union-by-height example

- initial state
- **union**$(4, 5)$
- **union**$(6, 7)$
- **union**$(5, 7)$
- **union**$(3, 7)$

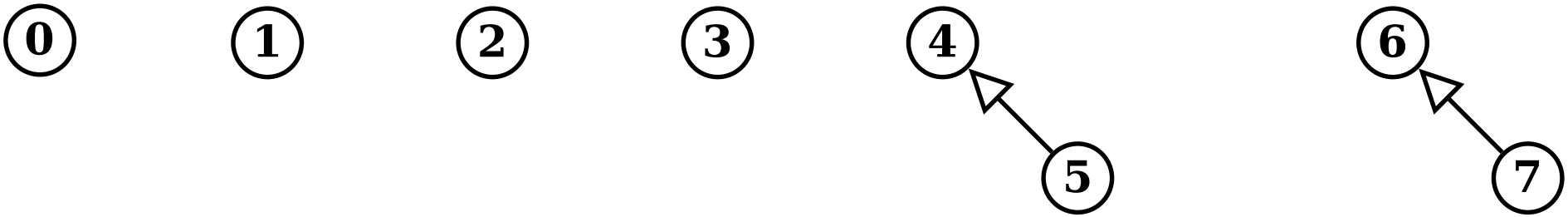


Implicit representation as an array with parent labels

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

union-by-height example

- initial state
- union$(4, 5)$
- **union**$(6, 7)$
- union$(5, 7)$
- union$(3, 7)$

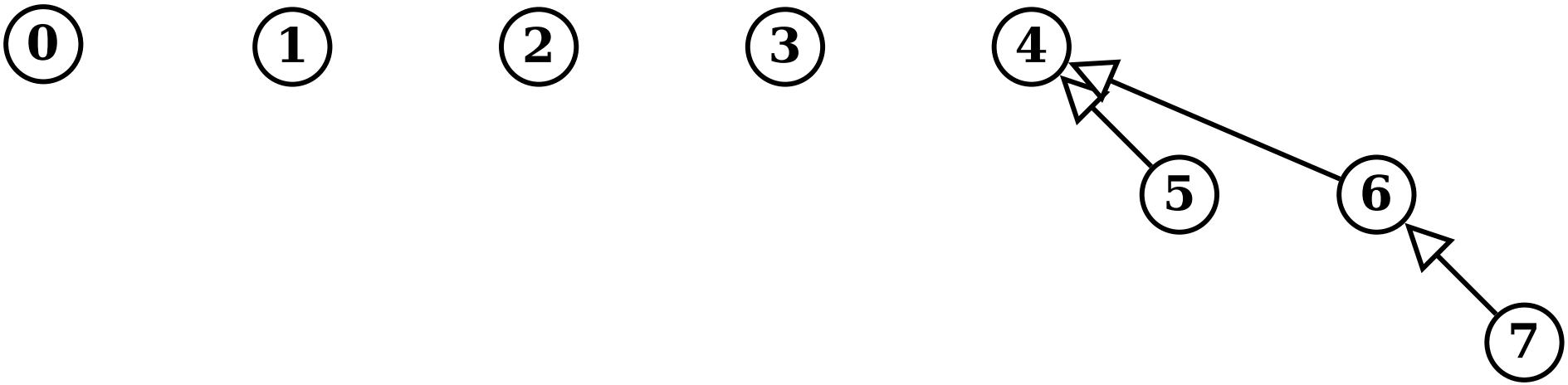


Implicit representation as an array with parent labels

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

union-by-height example

- initial state
- union$(4, 5)$
- union$(6, 7)$
- **union**$(5, 7)$
- union$(3, 7)$

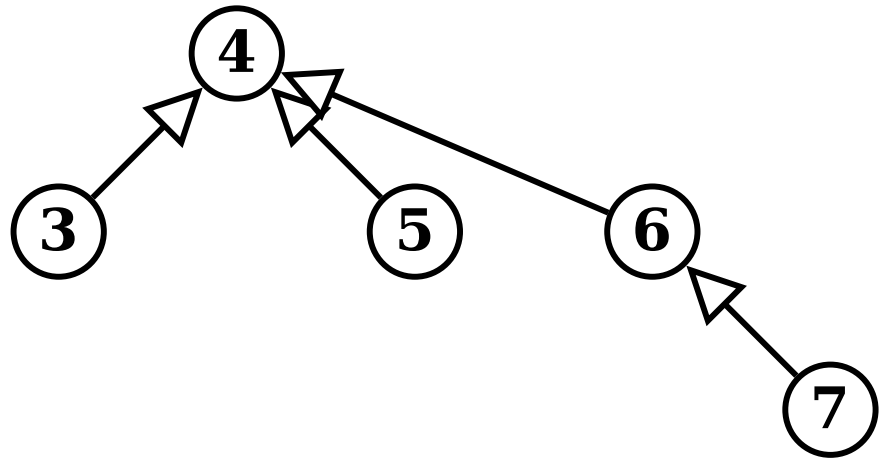


Implicit representation as an array with parent labels

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

union-by-height example

- initial state
- union$(4, 5)$
- union$(6, 7)$
- union$(5, 7)$
- union$(3, 7)$



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

Implementation of **Union-by-height**: – 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    }
```

[†] Note again, we are storing height $+ 1$ as a negative number, because height $= 0$ cannot be coded as a negative number.

Implementation of **Union-by-height**: – **find**(a)

Exactly same implementation as for **union-by-size** (see slide 17)

Implementation of **Union-by-height**: – **union**(a, b)

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-rank is same as Union-by-height but with an additional optimization implemented during **find**$(\cdot)$

- Union-by-height with a rather simple optimization during **find**$(\cdot)$operation yields union-by-rank.
- This optimization, although simple and straight-forward, has a drastic impact on the **amortized** complexity of a disjoint-set data structure:

Amortized Complexity

Amortized analysis attempts to track the complexity of performing a **sequence of operations** on a particular data structure, rather than analysing just the worst-case complexity of a single operation.

Path compression during **find**$(\cdot)$operation in union-by-rank

Path compression

When executing **find**(a), every node along the path from the '`root`' node to 'a' has its **parent index changed to point directly to the root**.

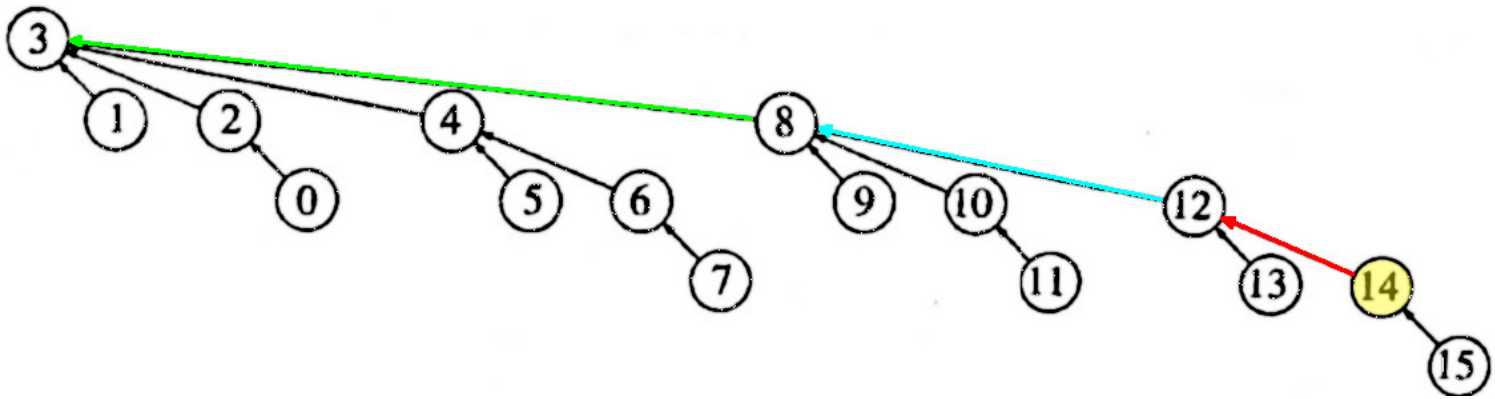
Notes:

- 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 (on slide #23).

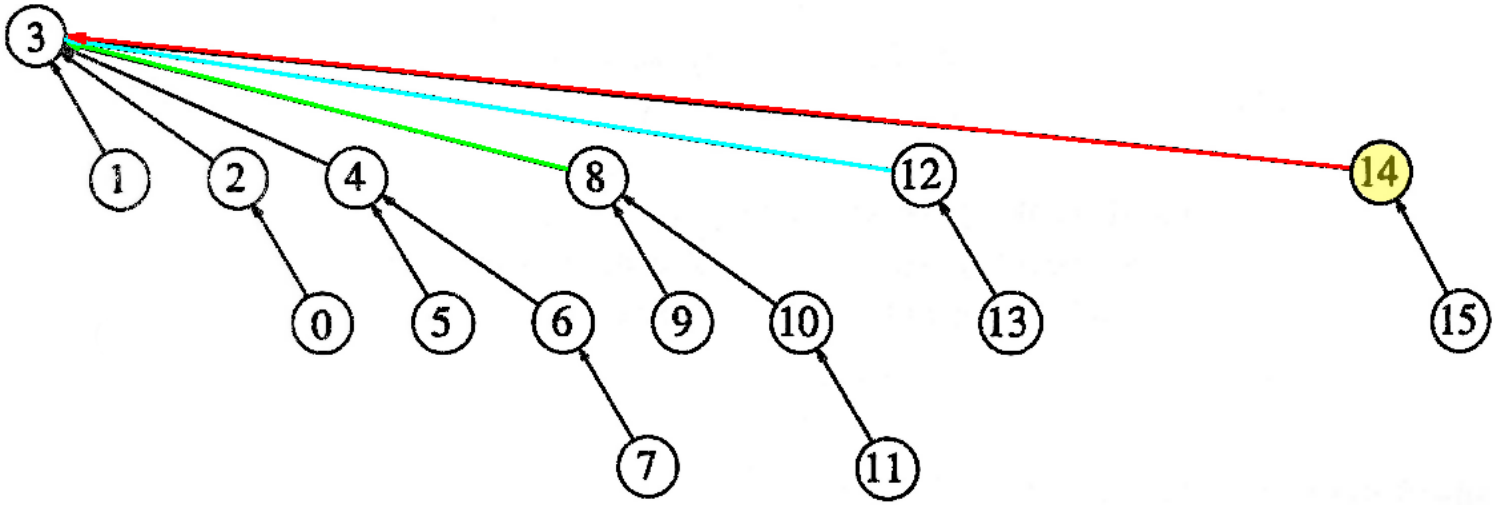
Path compression illustration

Consider this example (completely different one from the one used before).

Before **`path compression`**



Say we are running **`find`**(14) on the above state. The resultant path compressed tree 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    }
```

Complexity analysis of Disjoint set data structure using union-by-size, union-by-height and union-by-rank.

Analysis of **union-by-size**

Lemma

For any disjoint-set tree constructed using **union-by-size** with a root node (r), with its size being **size**$((r))$ and its height being **height**$((r))$, we have the following statement:

$$\textbf{size}((r)) \geq 2^{\textbf{height}((r))}$$

Proof by Induction

Base case:
When (r) is the root of a tree containing just one node (i.e., itself):

$$\textbf{size}((r)) = 1$$

$$\textbf{height}((r)) = 0$$

Substituting these values into the statement of the Lemma, the base case is true.

Analysis of **union-by-size**

Lemma

For any disjoint-set tree constructed using **union-by-size** with a root node (r), with its size being **size**$((r))$ and its height being **height**$((r))$, we have the following statement:

$$\textbf{size}((r)) \geq 2^{\textbf{height}((r))}$$

Proof by Induction

Inductive case (cont'd):

- Assume the statement is true for all disjoint-trees in the data structure after a sequence of k **union**$(\cdot, \cdot)$ operations.
- In the $(k+1)$-th **union**$(\cdot)$, let us say we are merging two disjointed trees, one rooted at (s) and other rooted at (r)
- Without loss of generality, let **size**$((r)) \geq$ **size**$((s))$)

Analysis of **union-by-size**

Lemma

For any disjoint-set tree constructed using **union-by-size** with a root node (r), with its size being **size**$((r))$ and its height being **height**$((r))$, we have the following statement:

$$\textbf{size}((r)) \geq 2^{\textbf{height}((r))}$$

Proof by Induction

Inductive case (cont'd):
Case 1: **height**$((r))$ > **height**$((s))$

$$\begin{aligned}
\text{After } (k+1)\text{-th merge:} \quad \text{new } \textbf{size}((r)) &\geq \text{old } \textbf{size}((r)) \\
&\geq 2^{\text{old } \textbf{height}((r))} \qquad \text{(inductive step)} \\
\text{But 'new } \textbf{height}((r))\text{'} = \text{'old } \textbf{height}((r))\text{'} & \\
\implies \text{new } \textbf{size}((r)) &\geq 2^{\text{new } \textbf{height}((r))}.
\end{aligned}$$

This proves the lemma for this case.

Analysis of **union-by-size**

Lemma

For any disjoint-set tree constructed using **union-by-size** with a root node r, with its size being **size**(r) and its height being **height**(r), we have the following statement:

$$\textbf{size}(r) \geq 2^{\textbf{height}(r)}$$

Proof by Induction

Inductive case (cont'd):
Case 2: **height**(r) $\leq$ **height**(s)

After $(k+1)$-th merge:

$$\begin{aligned}
\text{new } \textbf{size}(r) &= \text{old } \textbf{size}(r) + \textbf{size}(s) \\
&\geq 2 \times \textbf{size}(s) \qquad (\because \textbf{size}(r) \geq \textbf{size}(s)) \\
&\geq 2 \times 2^{\textbf{height}(s)} \qquad \text{(inductive step)} \\
&\geq 2^{\textbf{height}(s)+1} = 2^{\textbf{height}(r)}
\end{aligned}$$

This proves the lemma. ■

Analysis of **union-by-size**...cont'd.

Corollary

For a disjoint set with N elements, both **find**$(\cdot)$ and **union**$(\cdot,\cdot)$ take worst-case $O(\log_2(N))$ effort.

Proof:

$$\textbf{size}(\textcircled{r}) \geq 2^{\textbf{height}(\textcircled{r})}$$

But **size**$(\textcircled{r})$ is bounded above by N

$$\implies N \geq 2^{\textbf{height}(\textcircled{r})}$$
$$\implies \log_2 N \geq \textbf{height}(\textcircled{r})$$

- **find**$(\cdot)$in the worst case takes $O(\textbf{height}(\textcircled{r}))$, $\implies O(\log_2 N)$ effort.
- **union**$(\cdot,\cdot)$ involves 2 **find**$(\cdot)$operations $+O(1)$ effort, $\implies \log_2(N)$ effort.

■

union-by-height analysis and **union-by-rank** analysis ignoring **path compression** are the same!

Recall that **union-by-rank** and **union-by-height** mean the same thing when there is NO path compression applied in **find**$(\cdot)$operation.

Since we are ignoring path-compression for the worst-case analysis of **union-by-height**, in the remaining slides, the **rank** of any node (x) should be treated as identical to its **height** (i.e., the maximum number of hops needed from the leaf nodes to reach (x)).

Union-by-rank analysis (w/o path compression)

Lemma

Any node $\textcircled{x}$ with **rank**($\textcircled{x}$) $= k$, we have:

$$\textbf{size}(\textcircled{x}) \geq 2^k.$$

Proof

Base case:

When the tree contains only a singleton node $\textcircled{x}$, then
rank$(x) = 0$ and
size($\textcircled{x}$) $= 1$.

Thus, the statement is true for the base case.

Union-by-rank analysis (w/o path compression)

Lemma

Any node (x) with $\texttt{rank}((x)) = k$, we have:

$$\texttt{size}((x)) \geq 2^k.$$

Proof

Inductive case:

- Assume that the statement is true for all nodes up to rank $= k - 1$
- A node of rank $= k$ can only be created when merging 2 subtrees with roots of rank $= k - 1$ each.
- Inductively, each rank$=k - 1$ subtree has $\geq 2^{k-1}$ nodes.
- $\implies$ the new size of the tree $\geq 2 \times 2^{k-1} = 2^k$

Thus, the statement is also true for the general case. ■

Union-by-rank analysis (w/o **path compression**)

Corollary: Try and reason why this is true during self-study

From the Lemma on slide #32,

$$\texttt{height}(\textcircled{x}) \leq \log_2(N)$$

where N is the total number of elements in the disjoint set being merged under union-by-height/union-by-rank (w/o path compression) operations.

From this it can be shown that:

- **find**$(\cdot)$in the worst case takes $O(\texttt{height}(\textcircled{r}))$, $\implies$ $O(\log_2 N)$ effort.
- **union**$(\cdot,\cdot)$ involves 2 **find**$(\cdot)$operations $+O(1)$ effort, $\implies$ $\log_2(N)$ effort.

Union-by-rank analysis (w/ **path compression**) –not examinable

The analysis of **union-by-rank** with **path compression** has resulted in two stunning analyses:

Hopcroft-Ullman's theorem which proves that the total effort over any sequnce of m union/find operations is bounded by $O(m \log^*(N))$.

- $\log^*(N)$ is a very slowly growing **iterated logarithm** function.
- $\log^*(2^{65536}) = 5$

Tarjan's theorem which improves the upperbound to $O(m\alpha(N))$.

- $\alpha(n)$ is the **inverse Ackerman function**...
- ...which grows excruciatingly more slowly than $\log^*(n)$

Hand-scribbled proof uploaded on moodle, and I encourage you to study it.

Take home message!

Union/Find operations implementing union-by-height/rank with path compression yields an amortized $O(1)$-time for any **find**$(\cdot)$, and **union**$(\cdot,\cdot)$ operation.

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**