Lecture #13
Disjoint Set

Algorithm

JBNU

Jinhong Jung

In This Lecture

❑ Advanced data structure - disjoint set

- What is the disjoint set?

- How to represent and implement disjoint sets?
  - Basic version of disjoint set

- How to improve efficiency?
  - Union by rank
  - Path compression

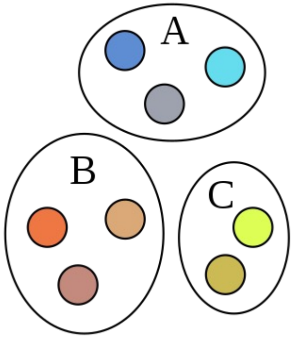
Outline

- Definition of disjoint set
- Disjoint set using non-binary tree
- How to improve efficiency?
- Analysis of disjoint set

Disjoint Set

❑ What is disjoint set?

- **Disjoint set** is a data structure managing non-overlapping **sets**
  - A **set** is used to contain unique objects.
  - Each intersection of two sets are empty.



Disjoint sets

❑ Applications

- Used when we need to manage multiple partitions or groups in a problem
  - Connected components in a graph
  - Minimum spanning tree in a graph (Kruskal's algorithm)

Main Operations

- make-set(u)
  - Create a new set containing only given element u
- find-set(u)
  - Return the set containing given element u
- union(u, v)
  - Merge (or union) the set having u and the set having v
- Notes
  - No need to consider intersect operation in disjoint set
  - Due to the operations, it's also known as union-find.

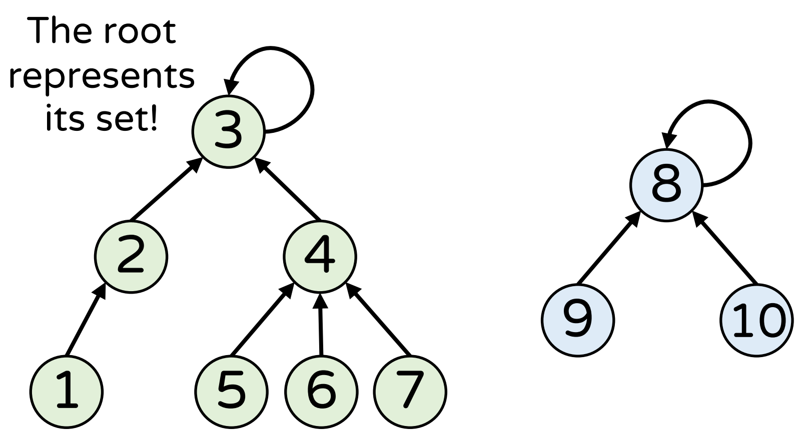
Outline

- ❑ Definition of disjoint set
- ❑ **Disjoint set using non-binary tree**
- ❑ How to improve efficiency?
- ❑ Analysis of disjoint set

How To Represent Disjoint Set

❑ A disjoint set is represented by a non-binary tree.

- Unlike normal trees, we use **parent pointer tree.**
  - A child points to its parent, and the root points to itself (self-looped).
- Each tree represents a set (i.e., forest = multiple sets)
- This tree is implemented by an 1D array called `p`.
  - Assume an element in the set is a positive integer.



{1, 2, 3, 4, 5, 6, 7}   {8, 9, 10}

Node's key

Index	1	2	3	4	5	6	7	8	9	10
Value	2	3	3	3	4	4	4	8	8	8

Parent

p[u]: parent of node u

Main Operations of Basic Version

❑ make-set(u)

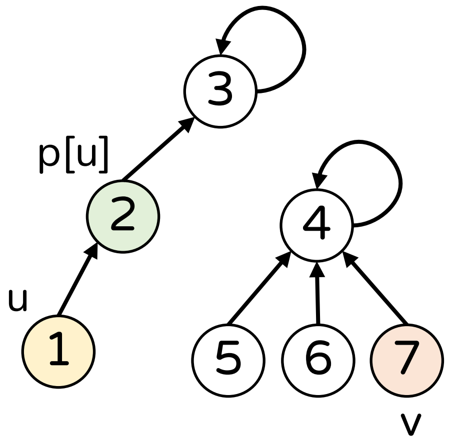
- It's implemented as u's parent points to u

❑ find-set(u)

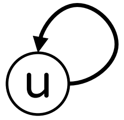
- Return the root of the set containing given u
- Recursively walk up from u to the root

❑ union(u, v)

- Merge the set having u and the set having v
  - Let the root of one set point to the root of other set




```
def make-set(u):
    p[u] ← u
```




```
def find-set(u):
    if u is p[u]:  # if self-looped, it's root
        return u
    else:
        return find-set(p[u])  # go up one level
```

```
def union(u, v):
    p[find-set(v)] ← find-set(u)
    # v's root points to u's root
```

Analysis of Basic Version

❑ Space complexity

- It takes $\Theta(n)$ space because of the array p.

❑ Time complexity

- make-set(u) takes $\Theta(1)$ time.
- find-set(u) takes $\Theta(h_u)$ time.
  - h_u is the height of the tree having u
- union(u, v) takes $h_v + h_u + c$ time.

❑ For a worst case, find-set(u) takes $\Theta(n)$ time.

- When the tree of n nodes becomes degenerate
- Can we improve this even for such a worst case?

Outline

- ❑ Definition of disjoint set
- ❑ Disjoint set using non-binary tree
- ❑ **How to improve efficiency?**
- ❑ Analysis of disjoint set

How To Improve Efficiency?

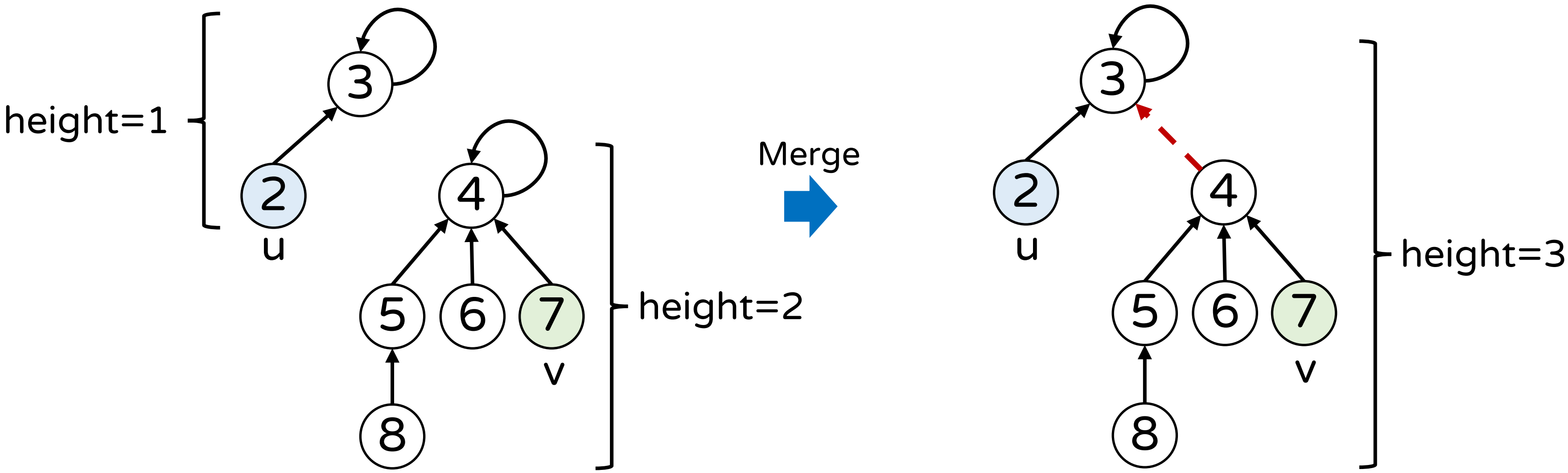
- The efficiency of disjoint set can be improved
  - By reducing the height of each tree
  - Because main operations totally depends on the tree height

- Two techniques can be used for the purpose
  - Union by rank
    - Idea: smaller tree is merged into taller tree in union
  - Path compression
    - Idea: flatten the tree while walking up to the root in find-set

Union By Rank (1)

❑ When does the tree's height increase?

- It increases while we merge two disjoint sets – union(u, v)
- Suppose the right tree is merged into the left one.

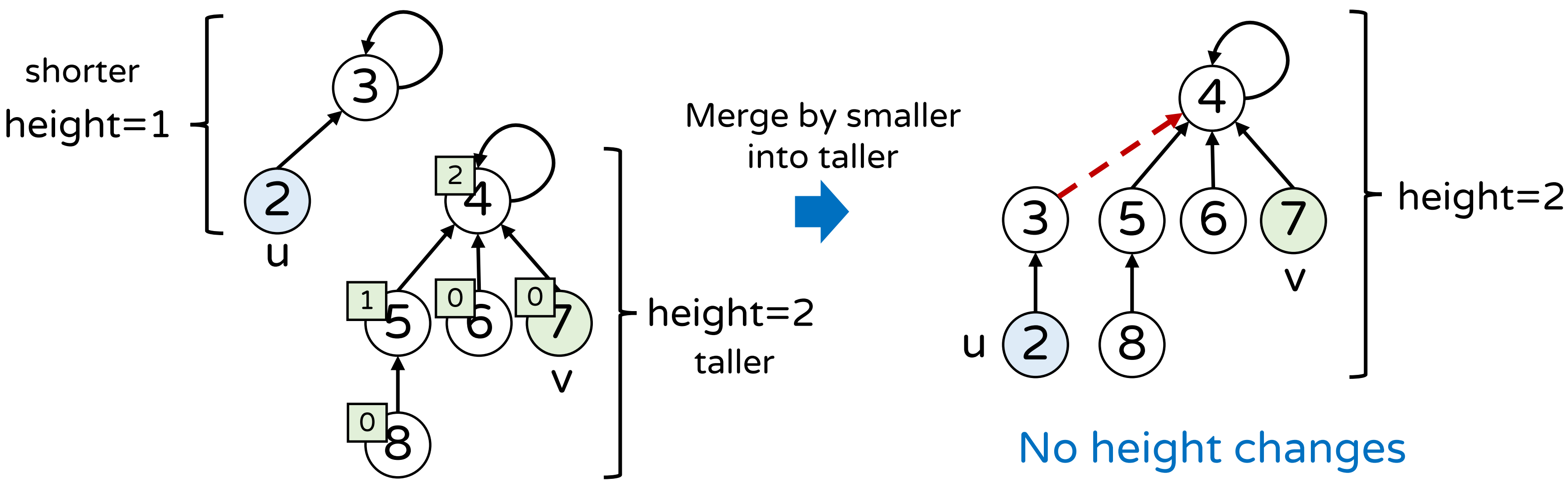


- What if the left tree is merged into the right one?
  - Then, the height doesn't change.

Union By Rank (2)

❑ Smaller into taller strategy

- Let's merge the shorter tree into the taller tree



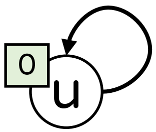
- To check the tree's height quickly, let's store a variable for each node, called rank.

Union By Rank (3)

❑ make-set(u)

- Make one disjoint set of u

```
def make-set(u):
    p[u] ← u
    rank[u] ← 0
```



❑ union(u, v)

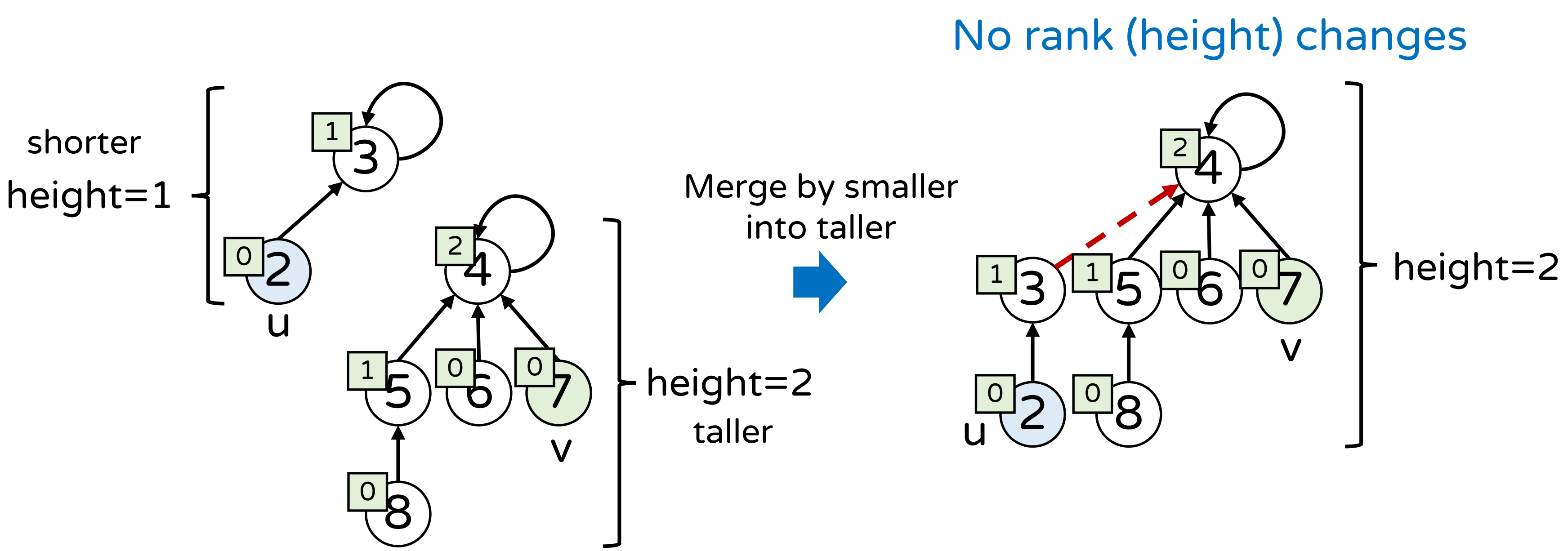
- Merge the set having u and the set having v by smaller into larger strategy

```
def union(u, v):
    u_r ← find-set(u) # u_r is the root node of the set having u
    v_r ← find-set(v)
    if rank[u_r] > rank[v_r]: # the tree of u_r is taller than that of v_r
        p[v_r] ← u_r  # the tree of v_r is merged into that of u_r
    else:
        p[u_r] ← v_r
        if rank[u_r] == rank[v_r]: # the tree of u_r has the same height as that of v_r
            rank[v_r] ← rank[v_r] + 1 # the resulting height increases by 1
```

Examples (1)

❑ When the height does not change after merge

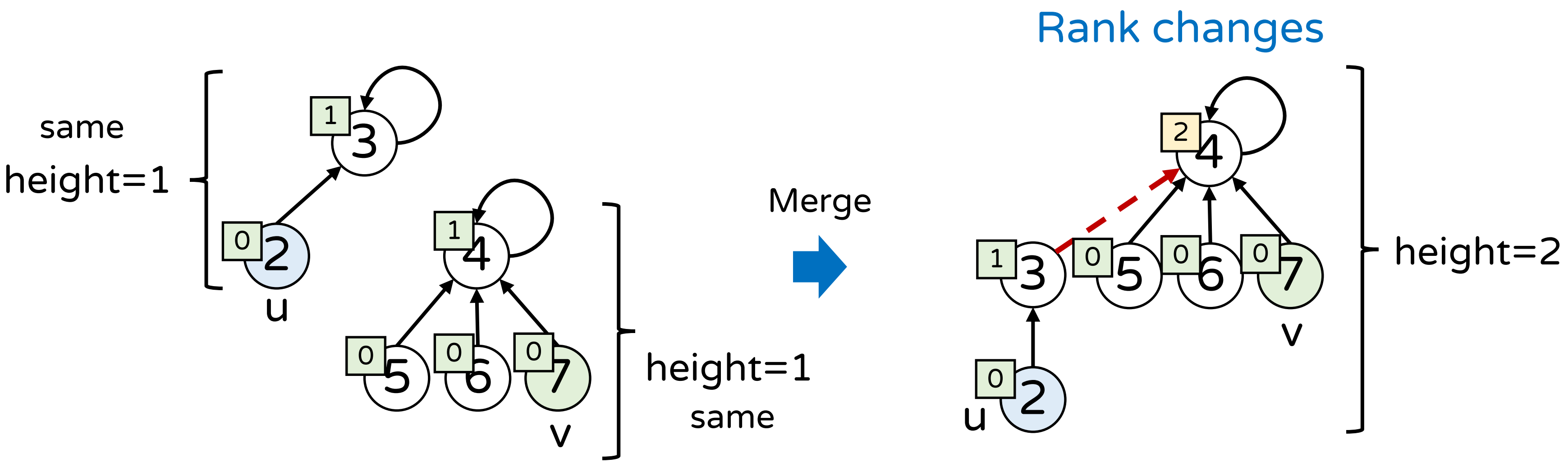
- If their heights are different, the height of the merged tree don’t change.



Examples (2)

When the height changes

- If their heights are the same, the height of the merged tree increases by 1.



How To Improve Efficiency?

- The efficiency of disjoint set can be improved
  - By reducing the height of each tree
  - Because main operations totally depends on the tree height

- Two techniques can be used for the purpose
  - Union by rank
    - Idea: smaller tree is merged into taller tree in union
  - ⇨ Path compression
    - Idea: flatten the tree while walking up to the root in find-set

Path Compression

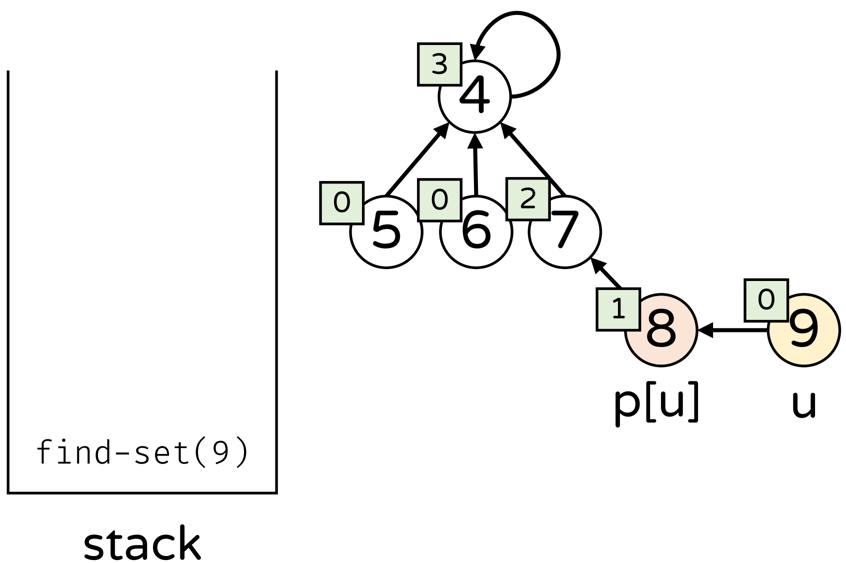
- **Even though we use union-by-rank, the tree's height can increase during the union operation**
  - When the height of the sets to be merged is the same
  - Where else can we reduce the tree's height?

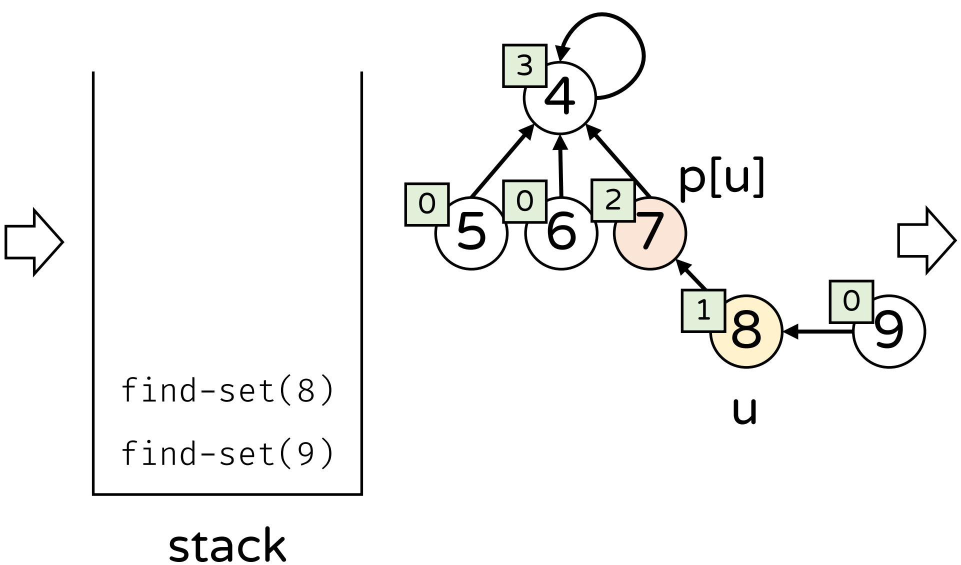
- **Path compression's idea: Let's flatten the tree**
  - Every time we walk up the tree during find-set, let's re-assign parent pointers to make each node we pass a direct child of the root

Examples (1)

❑ Flatten the tree during the find-set operation!

```
def find-set(u):
    if p[u] != u:
        p[u] ← find-set(p[u])
    return p[u]
```

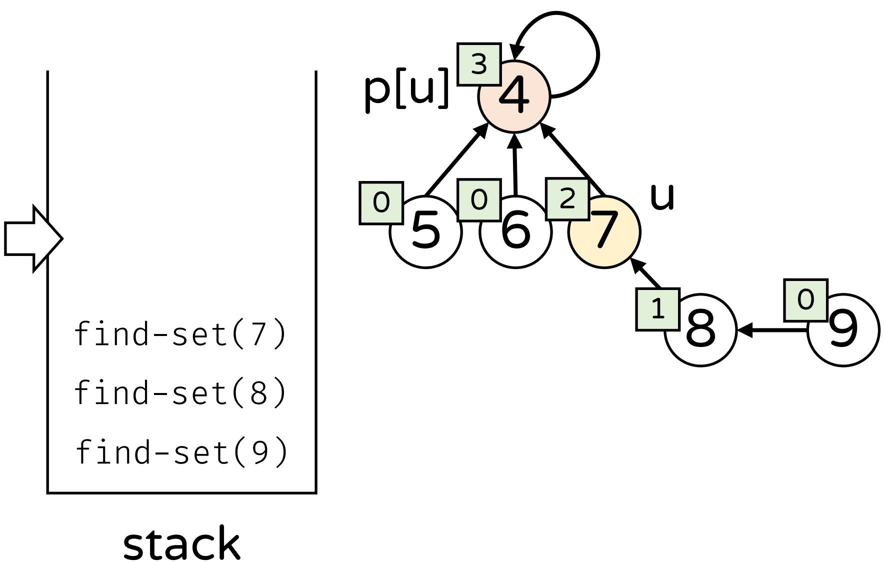


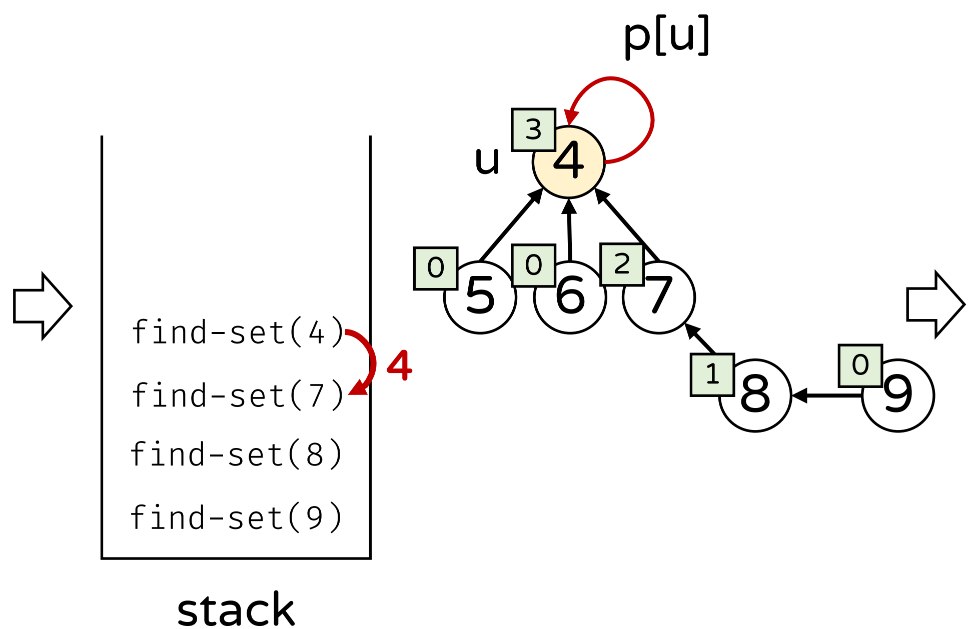


Examples (2)

- Flatten the tree during the find-set operation!

```
def find-set(u):
    if p[u] != u:
        p[u] ← find-set(p[u])
    return p[u]
```

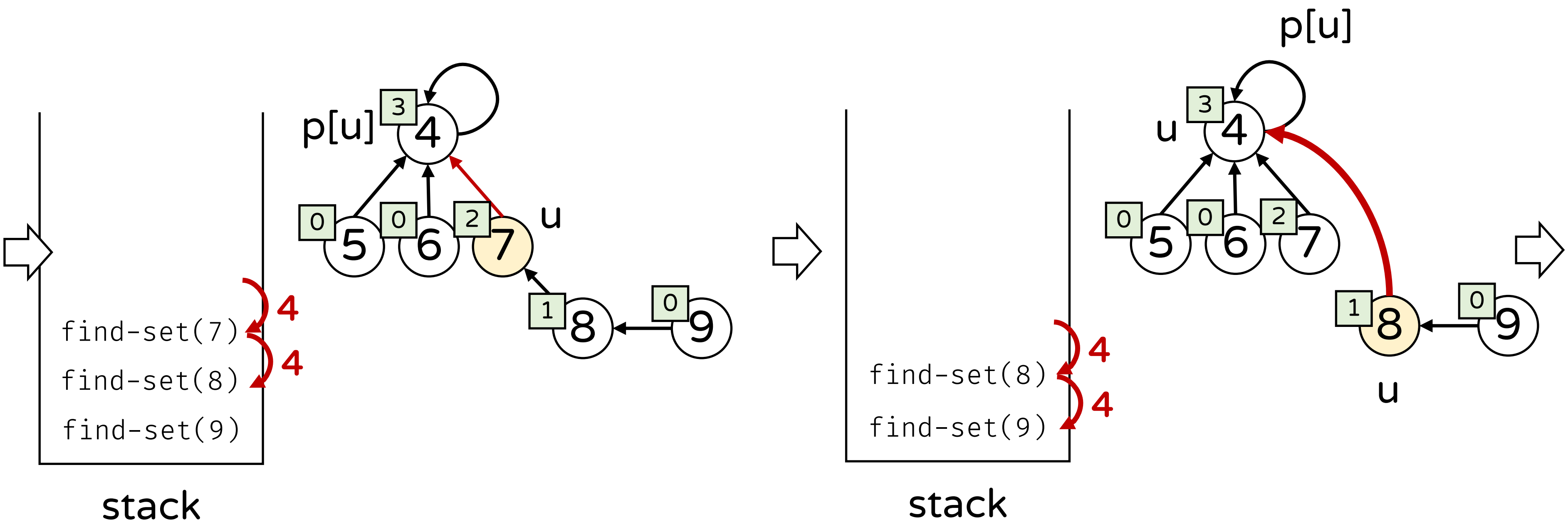




Examples (3)

❑ Flatten the tree during the find-set operation!

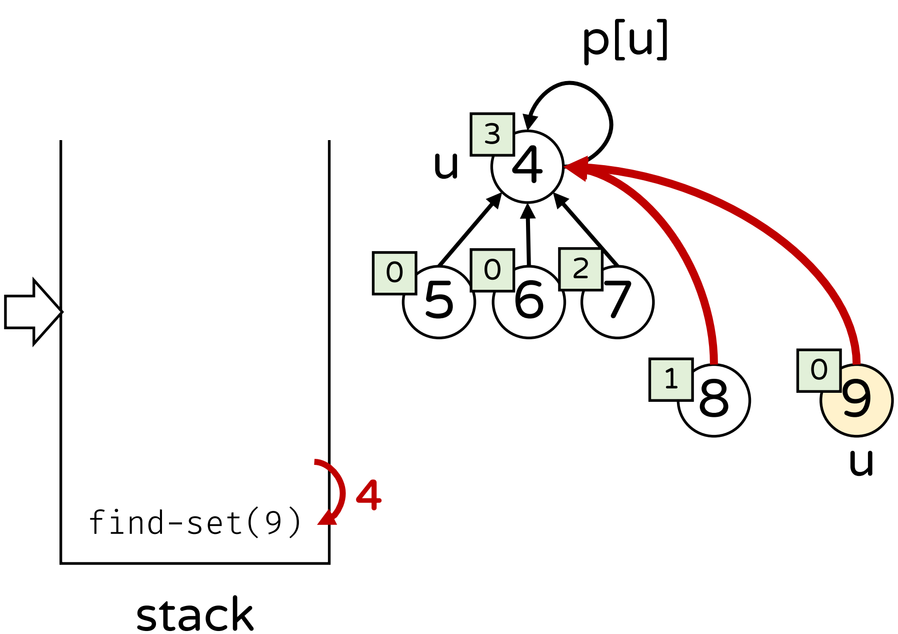
```
def find-set(u):
    if p[u] != u:
        p[u] ← find-set(p[u])
    return p[u]
```

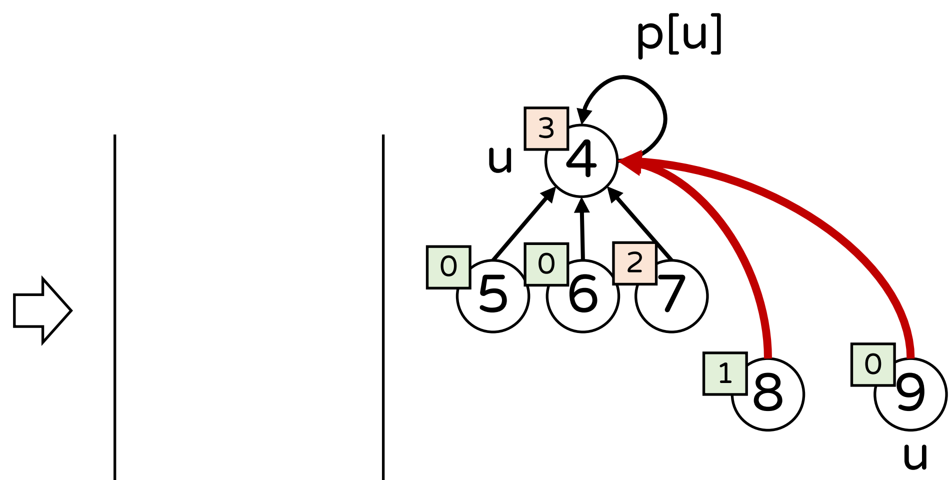


Examples (4)

❑ Flatten the tree during the find-set operation!

```
def find-set(u):
    if p[u] != u:
        p[u] ← find-set(p[u])
    return p[u]
```





The ranks are not updated!
Thus, a rank is the upper bound of the height.

Outline

- ❑ Definition of disjoint set
- ❑ Disjoint set using non-binary tree
- ❑ How to improve efficiency?
- ❑ Analysis of disjoint set

Analysis of Union By Rank

- **Claim:** using union by rank, # of elements in a set represented by a root having rank k is at least 2^k.
  - Base case: If rank = 0, $2^0 = 1$ element in the set.
  - Inductive step
    - Assume the claim holds for rank r; then, is it true for rank $r + 1$.
    - The rank becomes $r + 1$ when both ranks of two sets are r.
    - By the assumption, each set has at least 2^r elements.
    - Thus, the merged set of rank $r + 1$ has at least $2^r + 2^r = 2^{r+1}$ elements.
- **Claim:** using union by rank, if the set has n nodes, then the root of the set for has $O(\log n)$ rank.
  - Let k be the root's rank; $n \geq 2^k \Leftrightarrow k \leq \log_2 n = O(\log n)$
    - The height of the tree $\leq$ rank $k \leq \log_2 n$

Analysis of Union By Rank

❑ Time complexity of basic version + union-by-rank

- make-set(u) takes $O(1)$ time.
- find-set(u) takes $O(\log n)$ time.
- union(u, v) takes $O(\log n)$ time.

❑ (Amortized) Analysis in a sequence of operations

- Among m operations consisting of make-set, find-set, and union, let n be the number of make-set operations.
- Then, the total complexity is $O(m \log n)$.
  - Because after n make-set operations, there are n nodes; thus, the height of a tree cannot exceeds $O(\log n)$.
  - Thus, m times of the above operations takes $O(m \log n)$

Analysis of Path Compression

❑ (Amortized) Analysis in on a sequence of operations

- Among m operations consisting of make-set, find-set, and union, let n be the number of make-set operations.
- The total complexity is $O(m \log^* n)$ (proof is out-of-scope)
  - $\log^* n = \min\{k \mid \underbrace{\log \log \cdots \log}_{k} n \leq 1\}$ (repeatedly apply $\log()$ to n, k times)
  - $\log^* n$ is very small for extremely large n (e.g., $\log^* 2^{65536} = 5$).
- $\Rightarrow$ After m operations, it takes $O(m)$ time for a worst case.
  - On average, each operation takes $O(1)$ time!
- **Disjoint-set with union-by-rank and path-compression supports very fast operations.**

What You Need To Know

❑ Disjoint set (a.k.a. union-find)

- Data structure managing such non-overlapping sets
- **Main operations**: make-set, find-set, and union
- Represented by a non-binary **parent pointer tree**
  - For positive integer elements, 1D-array is enough for the purpose
- Disjoint set is improved by
  - **Union by rank**: smaller into taller strategy
  - **Path compression**: flatten the tree while walking up to the root
- Disjoint set with both techniques is very fast
  - By amortized analysis, each operation takes $O(1)$ time!

In Next Lecture

- **Minimum spanning tree on a graph**
  - Prim's algorithm
  - Kruskal's algorithm

- **Should review graph representation & basic graph searches**
  - See the graph section in data structure
    - Adjacency matrix
    - Adjacency list
    - Depth first search
    - Breadth first search

Thank You