Lecture #10 Advanced Data Structure (2)

Algorithm

JBNU Spring 2021

Jinhong Jung

In Previous Lecture

Red-black tree (used in std::map)

- Why do we need red-black tree?
  - ⇒ For a worst case, BST has $O(n)$ height, but RBT guarantees $O(\log n)$ height
- Definition and properties
  - BST where each node is colored by either RED or **BLACK**
    - P1) **BLACK** root node
    - P2) All **BLACK** leaf (or NIL) nodes
    - P3) No two consecutive RED nodes
    - P4) Consistent black height
- Insert and remove
  - Do BST's corresponding operation and re-arrange violated area using re-colorring and rotation case by case

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

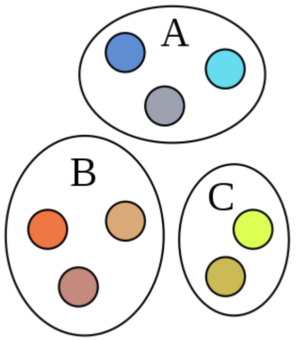
How to improve efficiency?

Analysis of disjoint set

Disjoint Set

What is disjoint set?

- A **set** is used to contain unique objects
- Consider we have multiple sets, and they are not overlapping, i.e., each intersection is empty
- **Disjoint set** is a data structure managing such **non-overlapping sets**



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

Disjoint set consists of the following 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
  - We do not need to consider intersect operation in disjoint set
  - Due to the operations, it's also known as union-find
  - We cover only a basic version of disjoint set in this lecture
    - It does not have remove operation of an element (not easy)

Outline

Definition of disjoint set

Disjoint set using non-binary tree

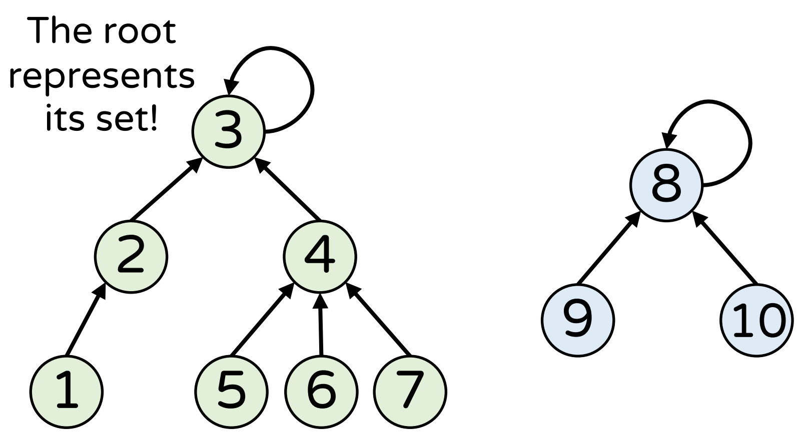
How to improve efficiency?

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 in the parent pointer tree
  - The root node points to itself (self-looped node)
- This data structure manages multiple sets (= forest)
- This tree is represented by an 1D array called p[ ]



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

For positive integer elements

Node's key

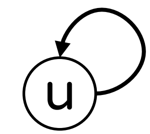
Index	1	2	3	4	5	6	7	8	9	10
Value	2	3	3	3	4	4	4	8	8	8

Parent

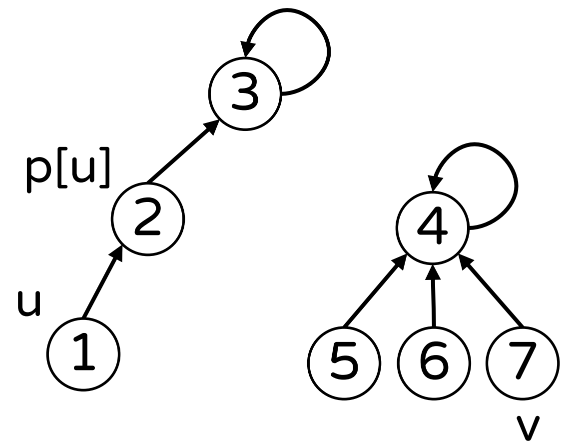
p[u]: parent of node u

Main Operations

Main operations of disjoint set



- make-set(u)
  - Given an element u, make it one disjoint set
    - 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 Disjoint Set

Space complexity of disjoint set [forest model]

- It takes $\Theta(n)$ space

Time complexity of each operation

- Efficiency of the basic implementation hinges completely on the height of the tree
  - make-set(u) takes $\Theta(1)$ time
  - find-set(u) takes $\Theta(h_u)$ time
    - Where 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 (or unbalanced)
  - 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?

Disjoint set can be improved in terms of efficiency

- 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

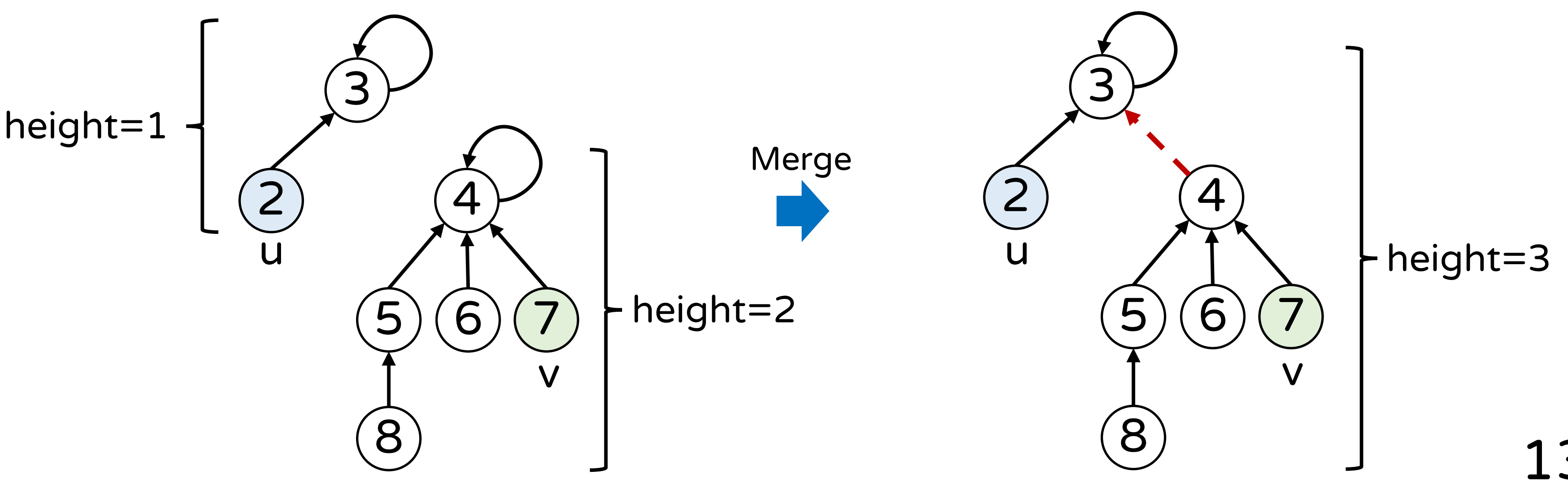
Union By Rank

When does the tree's height increase?

- It increases while we merge two disjoint sets – union(u, v)
- Let S_u denote the set containing u
- Merging S_v and S_u results in a tree of height as

$$\max\{\text{height}[S_u], \text{height}[S_v] + 1\}$$

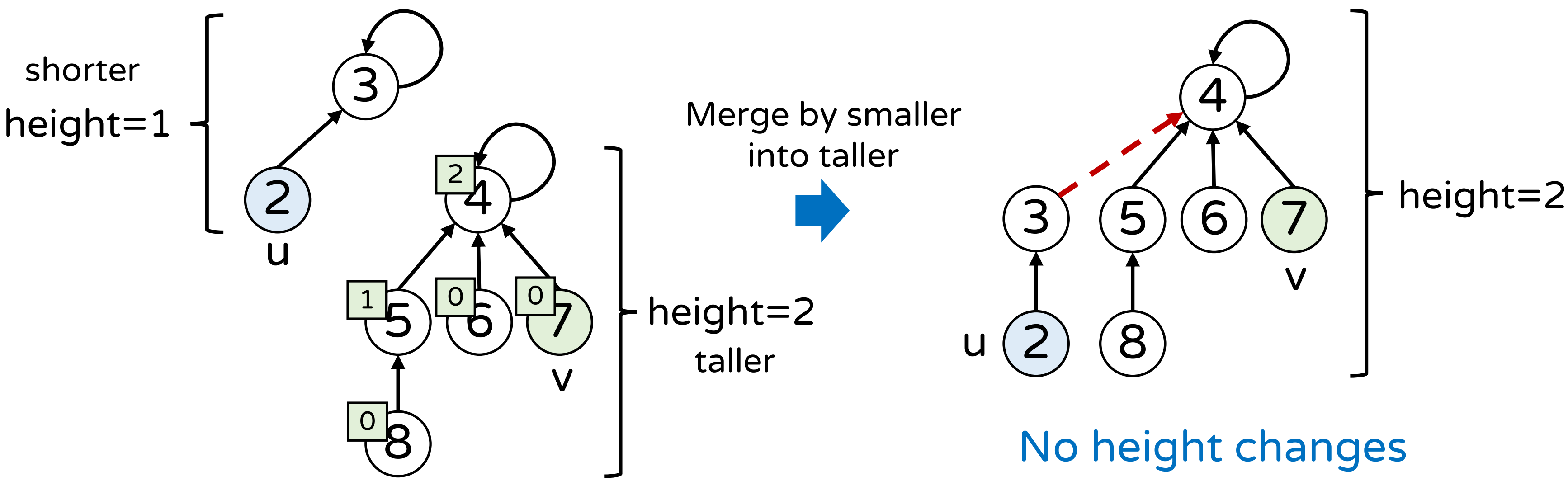
  - $\text{height}[S_v] + 1$ means the tree of v is added below the root of the tree of u



Union By Rank

Smaller into taller strategy

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

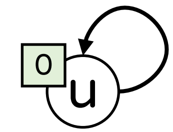


- To check the tree's height quickly, let's store a variable for each node, called rank
    - The rank of node u is the (upper bound) height of the sub-tree rooted at node u

Union By Rank

Main operations of disjoint set

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



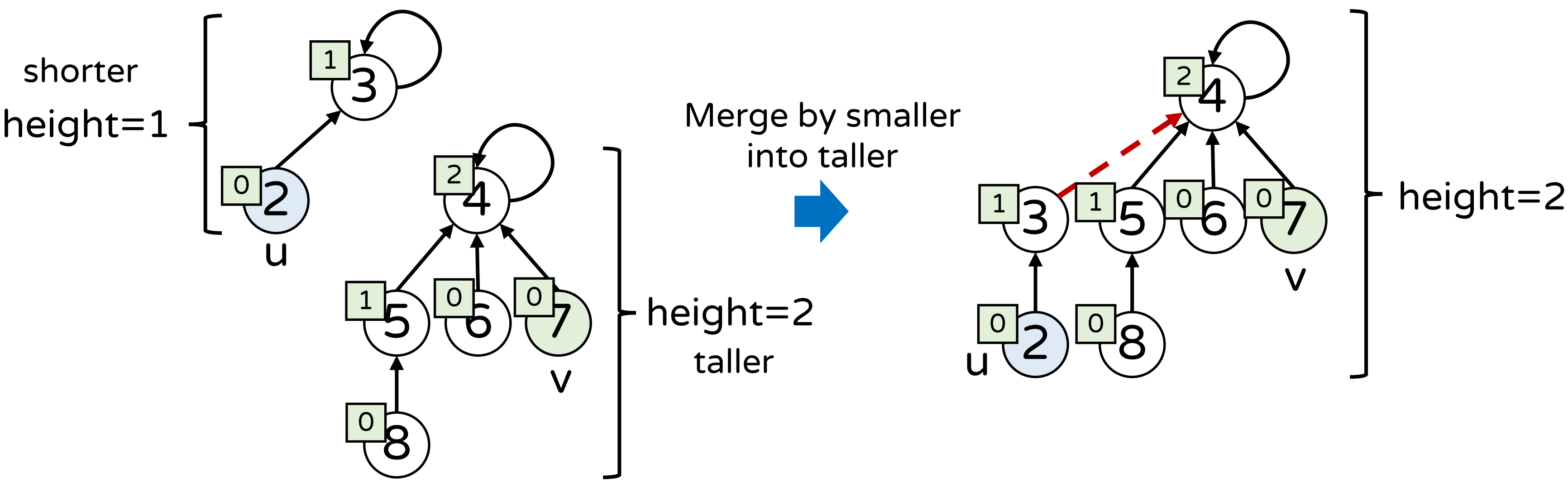
- make-set(u)
  - Given an element u, make it one disjoint set
- union(u, v)
  - Merge the set having u and the set having v by smaller into larger strategy

```
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
```

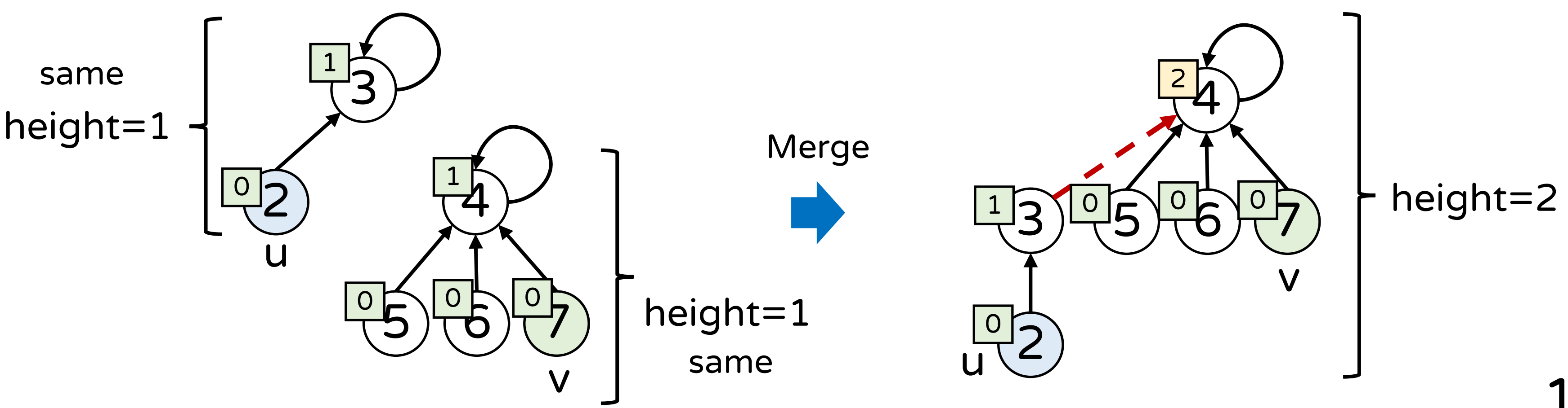
Union By Rank

Examples

No rank (height) changes



Rank changes

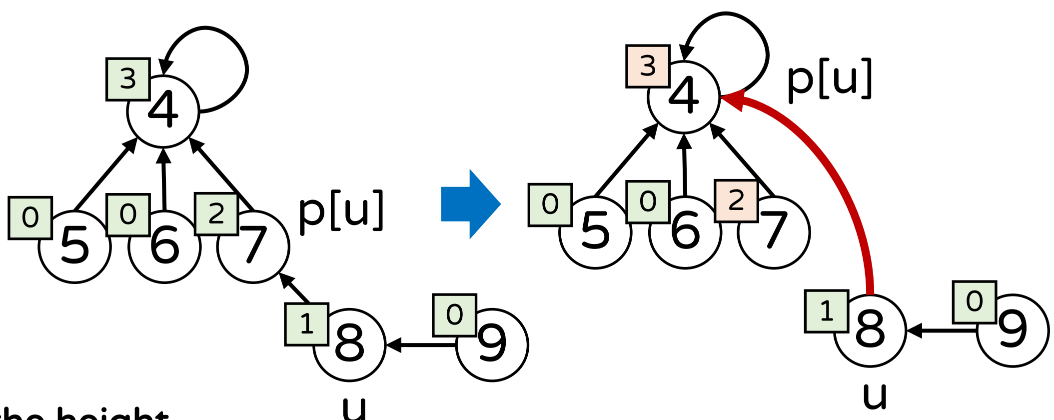


Path Compression

When does the tree's height increase?

- Even if 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 in find-set, let's re-assign parent pointers to make each node we pass a direct child of the root

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



Note that the ranks are not updated!
That's why 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

- **Proof by induction**
  - 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 resulting 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 + Union By Rank

Using only union by rank

- Time complexity of each operation
  - make-set(u) takes $\Theta(1)$ time
  - find-set(u) takes $\Theta(\log n)$ time
  - union(u, v) takes $\Theta(\log n)$ time
- (Amortized) Analysis based 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
  - 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 + Path Compression

Using union by rank + path compression

- (Amortized) Analysis based 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
  - Then, the total complexity is $O(m \log^* n)$ (proof is out-of-scope)
    - $\log^* n = \min\{k \mid \underbrace{\log \log \cdots \log n}_{k} \leq 1\}$ (repeatedly apply $\log()$ to n, k times)
    - $\log^* n$ is very small for extremely large n (e.g., $\log^* 2^{65536} = 5$)
  - The result means that after m operations, it takes $O(m)$ time for a worst case (with a practical input size n)
    - On average, each operation takes $O(1)$ time!
    - Disjoint-set with 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
  - This lecture does not cover a version having remove operation
- 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

Dynamic programming

- Concept and motivation

- Basic problems

Thank You