

# Disjoint Sets Representation, Counting Binary Trees and Selection Trees

Joseph Chuang-Chieh Lin (林莊傑)

Department of Computer Science & Engineering,  
National Taiwan Ocean University

Fall 2024



# Outline

- 1 Set Representation
- 2 Counting Binary Trees
- 3 Selection Trees

# Outline

- 1 Set Representation
- 2 Counting Binary Trees
- 3 Selection Trees

# Introduction

- In this class, we study the use of trees in the representation of sets.
- For simplicity, we assume that the elements of the sets are  $0, 1, 2, \dots, n - 1$ .
- We also assume that the sets being represented are pairwise disjoint.

# Introduction

- In this class, we study the use of trees in the representation of sets.
- For simplicity, we assume that the elements of the sets are  $0, 1, 2, \dots, n - 1$ .
- We also assume that the sets being represented are pairwise disjoint.
  - If  $S_i$  and  $S_j$  are two disjoint sets, then  $S_i \cap S_j = \emptyset$ , that is, no element that is in both  $S_i$  and  $S_j$ .

# Set Representation (1/2)

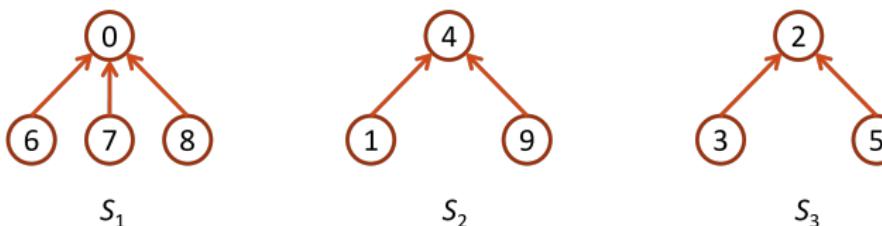
- Suppose that we have  $S_1 = \{0, 6, 7, 8\}$ ,  $S_2 = \{1, 4, 9\}$  and  $S_3 = \{2, 3, 5\}$ .

# Set Representation (1/2)

- Suppose that we have  $S_1 = \{0, 6, 7, 8\}$ ,  $S_2 = \{1, 4, 9\}$  and  $S_3 = \{2, 3, 5\}$ .
- The following figure illustrates one possible representation for these sets.

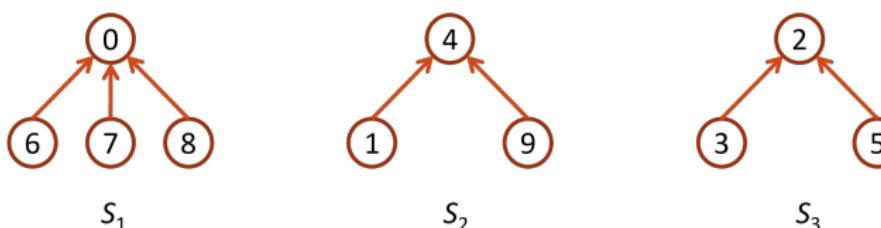
# Set Representation (1/2)

- Suppose that we have  $S_1 = \{0, 6, 7, 8\}$ ,  $S_2 = \{1, 4, 9\}$  and  $S_3 = \{2, 3, 5\}$ .
- The following figure illustrates one possible representation for these sets.



# Set Representation (1/2)

- Suppose that we have  $S_1 = \{0, 6, 7, 8\}$ ,  $S_2 = \{1, 4, 9\}$  and  $S_3 = \{2, 3, 5\}$ .
- The following figure illustrates one possible representation for these sets.



- **Note:** for each set, we have linked the nodes from the children to the parent.

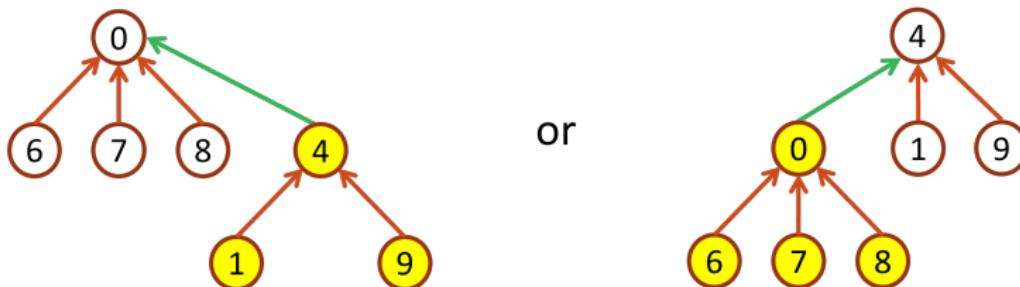
# Set Representation (2/2)

- The operations that we wish to perform on these sets are:

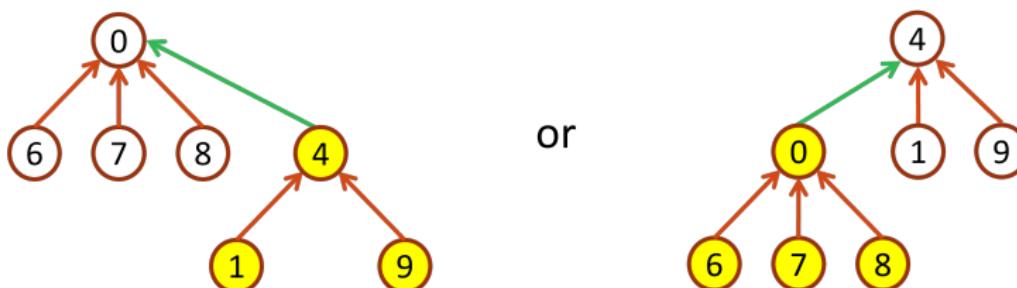
## Set Representation (2/2)

- The operations that we wish to perform on these sets are:
  - **Disjoint set union:** If  $S_i$  and  $S_j$  are two disjoint sets, then their union  $S_i \cup S_j = \{x \mid x \in S_i \text{ or } x \in S_j\}$ .
  - **Find(i):** find the set containing the element  $i$ .

# Possible Representations of the Union of Two Sets



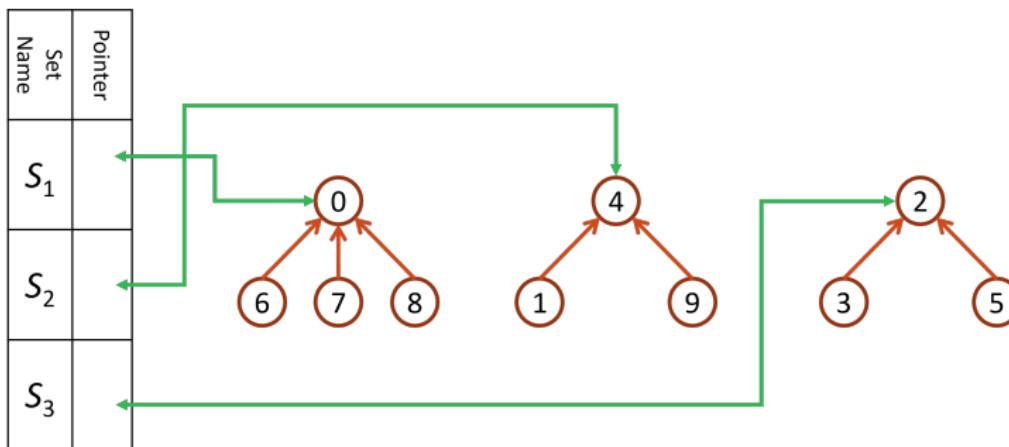
# Possible Representations of the Union of Two Sets



- Since we have linked the nodes from the children to parent, we simply make one of the trees a subtree of the other.

# Find() Operation

- We can find which set an element is in by following the parent links to the root and then returning the pointer to the set name.



# Array Representation of Sets

- We identify the sets **by the roots** of the trees representing them.
- We can use the node's number as the index in our simplified example.
- This means that each node needs only one field: the index of its parents, to link to its parent,

$i$	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
parent	-1	4	-1	2	-1	2	[0]	[0]	[0]	4

- **Note:** The root nodes have a parent of -1.

# Union and Find Operations

We can now find element  $i$  by simply following the **parent values starting at  $i$**  and continuing until we reach a negative parent value.

- For example, to **find** 5, we start at 5, and then move to 5's parent, 2. Since node 2 has a negative parent value, we have reached the root.

$i$	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
parent	-1	4	-1	2	-1	2	[0]	[0]	[0]	4

# Union and Find Operations

We can now find element  $i$  by simply following the **parent values starting at  $i$**  and continuing until we reach a negative parent value.

- For example, to **find** 5, we start at 5, and then move to 5's parent, 2. Since node 2 has a negative parent value, we have reached the root.

$i$	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
parent	-1	4	-1	2	-1	2	[0]	[0]	[0]	4

- To **union** two trees with root  $i$  and  $j$ , we can simply set  $\text{parent}[i] = j$ .



# Initial Attempt for the Union-Find Functions

```
int simpleFind (int i) {
    for (; parent[i] >= 0; i = parent[i])
        ;
    return i;
}

void simpleUnion (int i, int j) {
    parent [i] = j;
}
```

$i$	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
parent	-1	4	-1	2	-1	2	0	0	0	4



# Analysis of the Functions

- Consider the following case:

# Analysis of the Functions

- Consider the following case:
  - We start with  $p$  elements, each in a set of its own, that is,  $S_i = \{i\}$ .

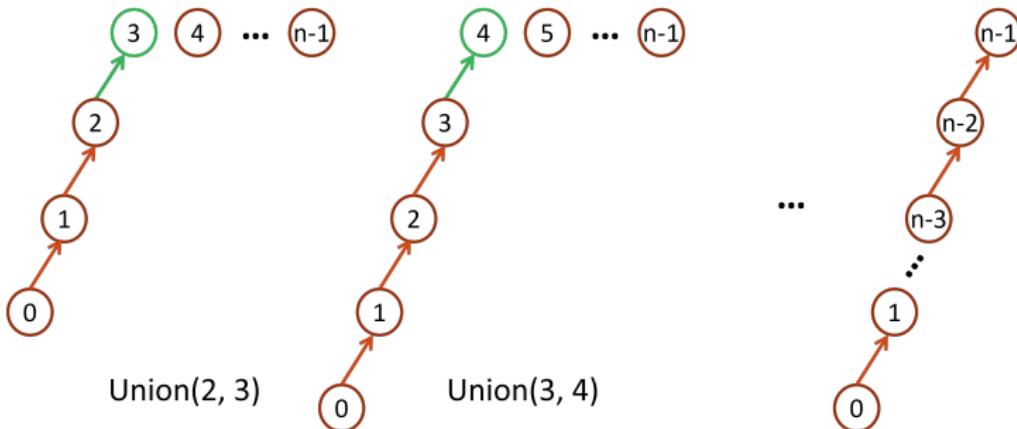
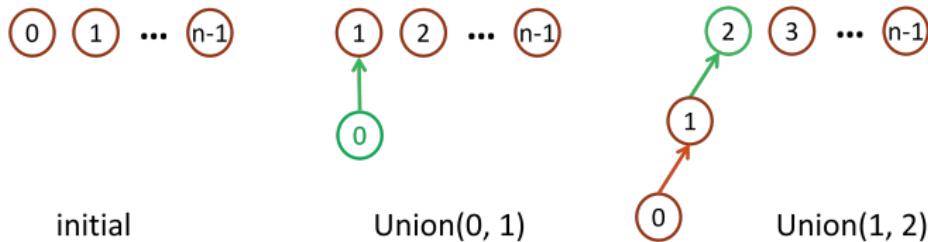
# Analysis of the Functions

- Consider the following case:
  - We start with  $p$  elements, each in a set of its own, that is,  $S_i = \{i\}$ .
- The following sequence of union-find operations produces the **degenerate tree** (退化樹):

# Analysis of the Functions

- Consider the following case:
    - We start with  $p$  elements, each in a set of its own, that is,  $S_i = \{i\}$ .
  - The following sequence of union-find operations produces the **degenerate tree** (退化樹):
    - ① union(0,1), find(0).
    - ② union(1,2), find(0).
    - ③ :
    - ④ union(n-2, n-1), find(0).
- ▷ The time complexity of **union operations** is  $O(n)$ .
- ▷ The time complexity of **find operations** is  $\sum_{i=2}^n i = O(n^2)$ .





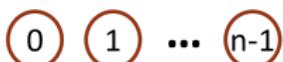
# Weighting Rule for Union comes to the Rescue!

## Weighting Rule for union( $i, j$ )

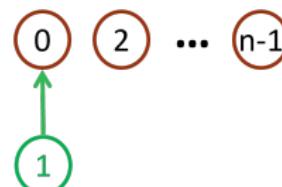
- If the number of nodes in tree  $i$  is less than the number in tree  $j$ , make  $j$  the parent of  $i$
- If the number of nodes in tree  $i$  is greater than the number in tree  $j$ , make  $i$  the parent of  $j$ .
- **Note:** If  $i$  is a root node, we set parent[ $i$ ] to be the negative number of nodes in that tree.

$i$	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
parent	-4	4	-3	2	-3	2	[0]	[0]	[0]	4

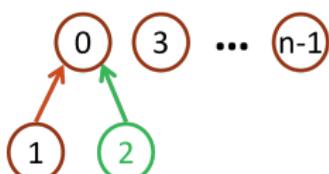
# Example of Using the Weight Rule



initial

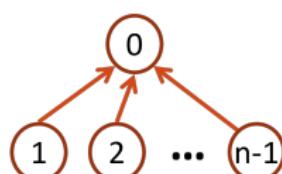


Union(0, 1)



Union(0, 2)

...



Union(0, n-1)

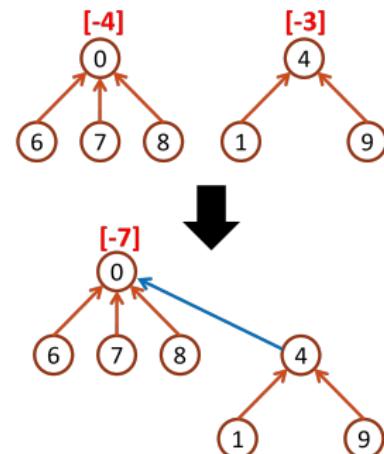
# The Code of Union Function Using Weighting Rule

```

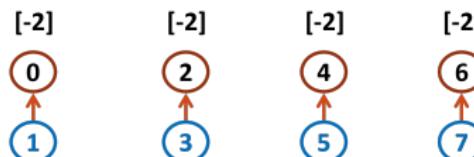
void weightedUnion(int i, int j) {
/* union the sets with roots i and j,
   i != j, using the weighting rule.
   parent [i] = -count [i]
   and parent [j] = -count[j] */

    int temp = parent[i] + parent[j];
    if (parent[i] > parent[j]) {
        parent[i] = j; /*make j the new root */
        parent[j] = temp;
    } else {
        parent[j] = i; /*make i the new root */
        parent[i] = temp;
    }
}

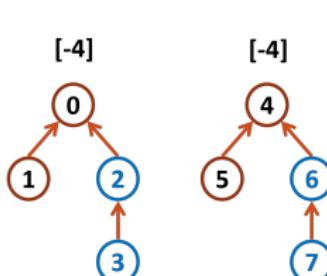
```



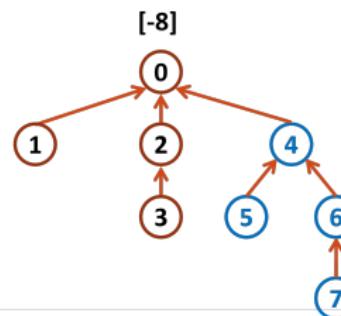
# Trees Achieving the Worst Case



(b) Height-2 trees following Union(0, 1), (2, 3), (4, 5) and (6, 7)



(c) Height-3 trees following Union(0, 2) and (4, 6)



(d) Height-4 tree following Union(0, 4)

# Another Rule: Collapsing Rule

## Collapsing Rule for $\text{union}(i, j)$

- If  $j$  is a node on the path from  $i$  to its root and  $\text{parent}[i] \neq \text{root}(i)$ , then set  $\text{parent}[j]$  to  $\text{root}(i)$ .
- Consider the previous example, and process the following eight  $\text{find}()$ :

$\overbrace{\text{find}(7), \text{find}(7), \dots, \text{find}(7)}^{8 \text{ times}}$ .

- The SimpleFind() needs  $3 \times 8 = 24$  moves.

# Another Rule: Collapsing Rule

## Collapsing Rule for union( $i, j$ )

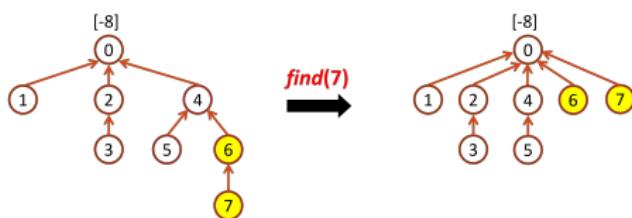
- If  $j$  is a node on the path from  $i$  to its root and  $\text{parent}[i] \neq \text{root}(i)$ , then set  $\text{parent}[j]$  to  $\text{root}(i)$ .
- Consider the previous example, and process the following eight `find()`:

$\overbrace{\text{find}(7), \text{find}(7), \dots, \text{find}(7)}^{8 \text{ times}}$ .

- The `SimpleFind()` needs  $3 \times 8 = 24$  moves.
- The `CollapsingFind()` needs  $3 + 3 + 7 = 13$  moves.
  - first `find(7)`: 3 moves.
  - reset 3 links: 3 moves.
  - remaining 7 finds: 7 moves.



# Collapsing Rule (contd.)



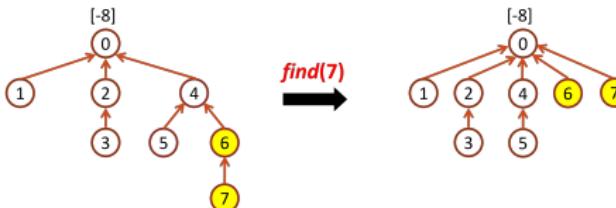
- When `collapsingFind` is used, the first `find(7)` requires going up three links and then resetting two links.
- Note:** Even though only two parent links need to be reset, `collapsingFind` will actually reset three (the parent of 4 is reset to 0).

# The Code for the Collapsing Rule

```

int collapsingFind (int i) {
    /* find the root of the tree containing element i.
       Use the collapsing rule to collapse all nodes
       from i to root */
    int root, trail, lead;
    for (root=i; parent[root]>=0; root=parent[root])
        ;
    for (trail=i; trail != root; trail=lead) {
        lead = parent[trail];
        parent[trail] = root;
    }
    return root;
}

```



Consider  $i = 7$ :

$\text{root} = 0$  (after the 1st for-loop)

$\text{trail} = 7$

$\text{lead} = \text{parent}[7] = 6$

$\text{parent}[\text{trail}] = \text{parent}[7] = 0$

$\text{trail} = 6$

$\text{lead} = \text{parent}[6] = 4$

$\text{parent}[6] = 0$

$\text{trail} = 4$

$\text{lead} = \text{parent}[4] = 0$

$\text{parent}[4] = 0$

$\text{trail} = 0$

# Outline

- 1 Set Representation
- 2 Counting Binary Trees
- 3 Selection Trees

# Counting Binary Trees

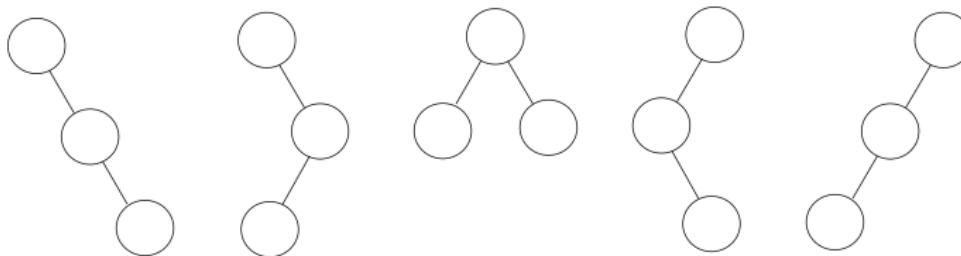
- Consider the following three disparate problems:
  - The number of distinct binary trees having  $n$  nodes.
  - The number of distinct permutations of the numbers from 1 to  $n$  obtainable by a **stack**.
  - The number of distinct ways of multiplying  $n + 1$  matrices.

# Counting Binary Trees

- Consider the following three disparate problems:
  - ➊ The number of distinct binary trees having  $n$  nodes.
  - ➋ The number of distinct permutations of the numbers from 1 to  $n$  obtainable by a **stack**.
  - ➌ The number of distinct ways of multiplying  $n + 1$  matrices.
- Amazingly, **these problems have the same solution!**

# Problem One

- The number of distinct binary trees having  $n$  nodes.



- ★ Example of  $n = 3$ .

## Problem Two

- The number of distinct permutations of the numbers from 1 to  $n$  obtainable by a stack.
  - ① push 1 → pop → push 2 → pop → push 3 → pop ⇒ 123.
  - ② push 1 → pop → push 2 → push 3 → pop → pop ⇒ 132.
  - ③ push 1 → push 2 → push 3 → pop → pop → pop ⇒ 321.
  - ④ push 1 → push 2 → pop → pop → push 3 → pop ⇒ 213.
  - ⑤ push 1 → push 2 → pop → push 3 → pop → pop ⇒ 231.

★ Example of  $n = 3$ .

# Outline

- 1 Set Representation
- 2 Counting Binary Trees
- 3 Selection Trees

# Discussions