Disjoint Sets Representation

Joseph Chuang-Chieh Lin (林莊傑)

Department of Computer Science & Engineering,
National Taiwan Ocean University

Fall 2025

Outline

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 $0, 1, 2, \ldots, n-1$.
- 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 $0, 1, 2, \ldots, n-1$.
- 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)

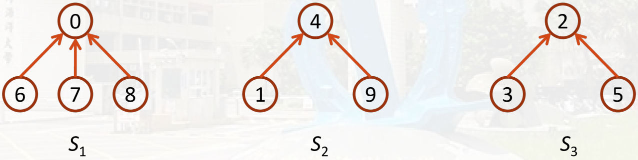
- Suppose that we have $S_1 = \{0, 6, 7, 8\}$, $S_2 = \{1, 4, 9\}$ and $S_3 = \{2, 3, 5\}$.

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.

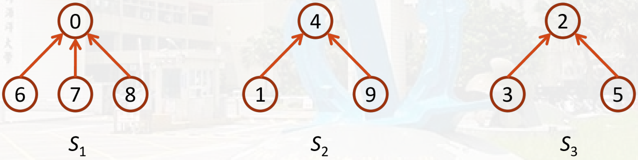
Set Representation (1/2)

- 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.

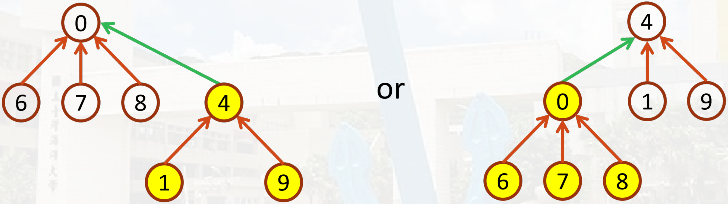
Set Representation (2/2)

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

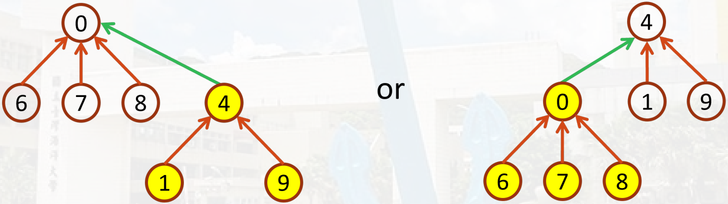
Set Representation (2/2)

- 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



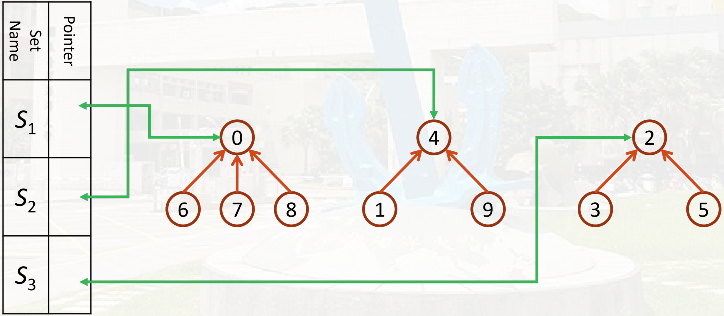
Possible Representations of the Union of Two Sets



- 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 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

Initial Attempt for the Union-Find Functions

```
int simpleFind (int i) {
// try the while-loop instead
    while (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, $S_i = \{i\}$.
  - Initially, parrent$[i] = -1$, for $0 \le i < p$.

Analysis of the Functions

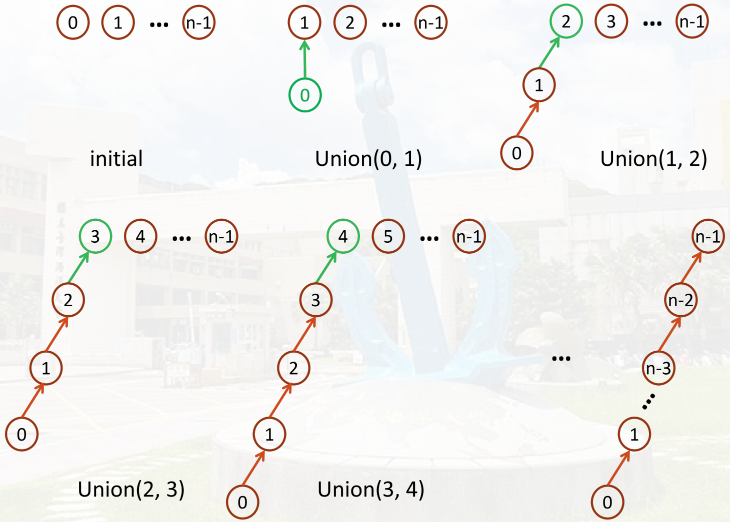
- Consider the following case:
  - We start with p elements, each in a set of its own, that is, $S_i = \{i\}$.
  - Initially, parrent$[i] = -1$, for $0 \leq i < p$.
- The following sequence of union-find operations produces the degenerate tree (退化樹):

Analysis of the Functions

- Consider the following case:
  - We start with p elements, each in a set of its own, that is, $S_i = \{i\}$.
  - Initially, parrent$[i] = -1$, for $0 \leq i < p$.
- The following sequence of union-find operations produces the degenerate tree (退化樹):
  - `union(0,1)`, `find(0)`.
  - `union(1,2)`, `find(0)`.
  - $\vdots$
  - `union(n-2, n-1)`, `find(0)`.

▷ The time complexity of union operations is $O(n)$.

▷ The time complexity of find operations is $\sum_{i=2}^{n} i = O(n^2)$.


0
1
...
n-1
initial
1
2
...
n-1
0
Union(0, 1)
2
3
...
n-1
1
0
Union(1, 2)
3
4
...
n-1
2
1
0
Union(2, 3)
4
5
...
n-1
3
2
1
0
Union(3, 4)
...
n-1
n-2
n-3
1
0

Weighting Rule for Union comes to the Rescue!

Goal: Avoiding the creation of degenerate trees.

Weighting Rule for Union comes to the Rescue!

Goal: Avoiding the creation of degenerate trees.

Weighting Rule for union(i, j)

- If the number of nodes in tree i is less than the number of nodes in tree j, make j the parent of i;
- If the number of nodes in tree i is greater than the number of nodes in tree j, make i the parent of j.

Weighting Rule for Union comes to the Rescue!

Goal: Avoiding the creation of degenerate trees.

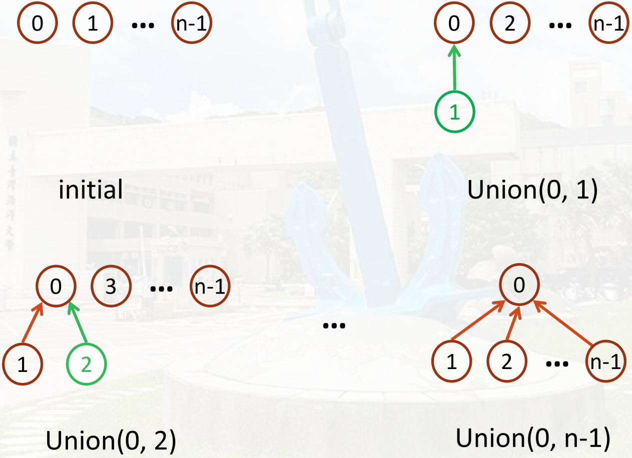
Weighting Rule for union(i, j)

- If the number of nodes in tree i is less than the number of nodes in tree j, make j the parent of i;
- If the number of nodes in tree i is greater than the number of nodes 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.

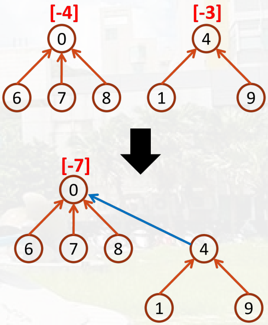
i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
parent	-4	4	-3	2	-3	2	0	0	0	4

Example of Using the Weight Rule

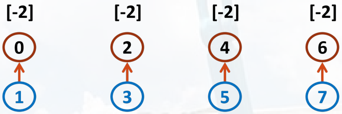


The Code of Union Function Using Weighting Rule

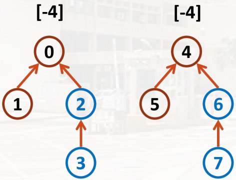
```
void weightedUnion(int i, int j) {
/* union the sets with roots i and j,
   i != j, using the weighting rule.
   parent[i] = -count[i]
   and parent[j] = -count[j] */

    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;
    }
}
```



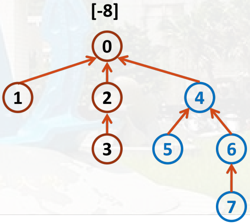
Trees Achieving the Worst Case



(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)

Worst depth of the tree by the Weighted Union

Lemma

Let T be a tree with n nodes created by `weightedUnion`, then no node in T has level greater than $\lfloor \lg n \rfloor + 1$.

- Proof by induction on n (Exercise).

Another Rule: Collapsing Rule

Collapsing Rule for union(i, j)

- If j is a node on the path from i to its root and parent$[i] \neq$ root(i), then set parent$[j]$ to root(i).

- Consider the previous example, and process the following eight `find()`:

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

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

Another Rule: Collapsing Rule

Collapsing Rule for union(i, j)

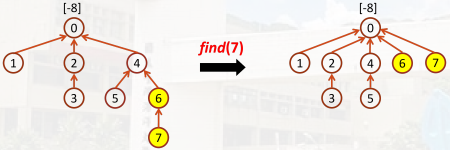
- If j is a node on the path from i to its root and parent$[i] \neq$ root(i), then set parent$[j]$ to root(i).

- Consider the previous example, and process the following eight `find()`:

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

- 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

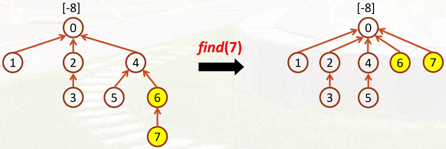
```
int collapsingFind (int i) {
/* find the root of the tree containing element i.
   Use the collapsing rule to collapse all nodes
   from i to root */
    int root, trail, lead;
    for (root=i; parent[root]>=0; root=parent[root])
        ;
    for (trail=i; trail != root; trail=lead) {
        lead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

Consider $i = 7$:

root $= 0$ (after the 1st for-loop)

trail $= 7$
lead $=$ parent[7] $= 6$
parent[trail] $=$ parent[7] $= 0$
trail $= 6$
lead $=$ parent[6] $= 4$
parent[6] $= 0$
trail $= 4$
lead $=$ parent[4] $= 0$
parent[4] $= 0$

trail $= 0$



Discussions