HANYANG UNIVERSITY
DIVISION OF COMPUTER SCIENCE

Data Structures

박영준 교수님

Lab7: Disjoint Set

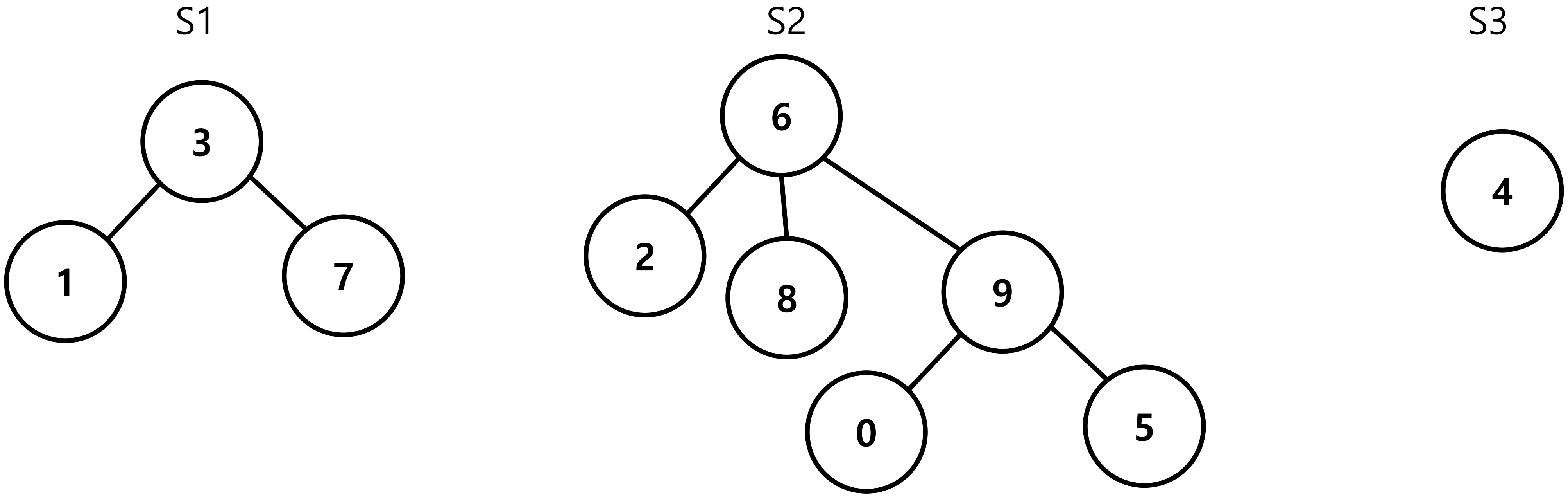
Disjoint Set Data Structure

- Data structure that tracks a set of elements partitioned into a number of disjoint subsets.

Disjoint Set Data Structure

- ex) Set S1, S2 and S3 are disjoint each other
  - S1 = {1, 3, 7}
  - S2 = {0, 2, 5, 6, 8, 9}
  - S3 = {4}

Disjoint Set Data Structure



*arr =

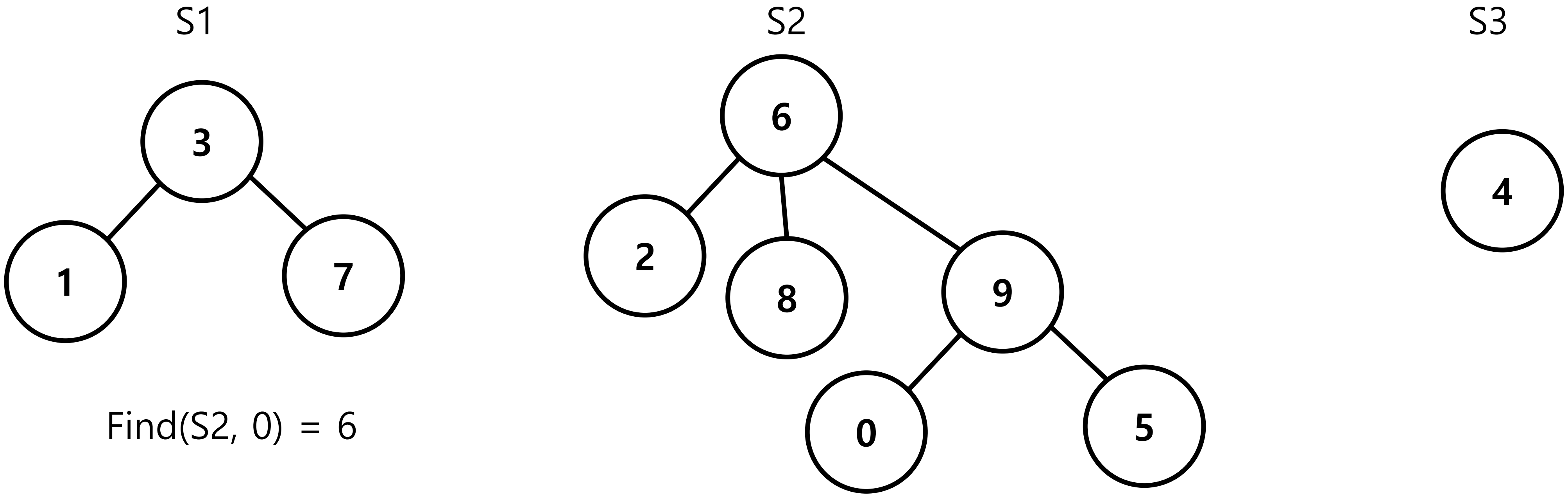
Node	0	1	2	3	4	5	6	7	8	9
Parent	9	3	6	-2	-1	9	-3	3	6	6

Disjoint Set Data Structure ADT

- DATATYPE Find(DATATYPE *arr, DATATYPE Node);
  - Find root of Node
  - Return root

Disjoint Set Data Structure

Find



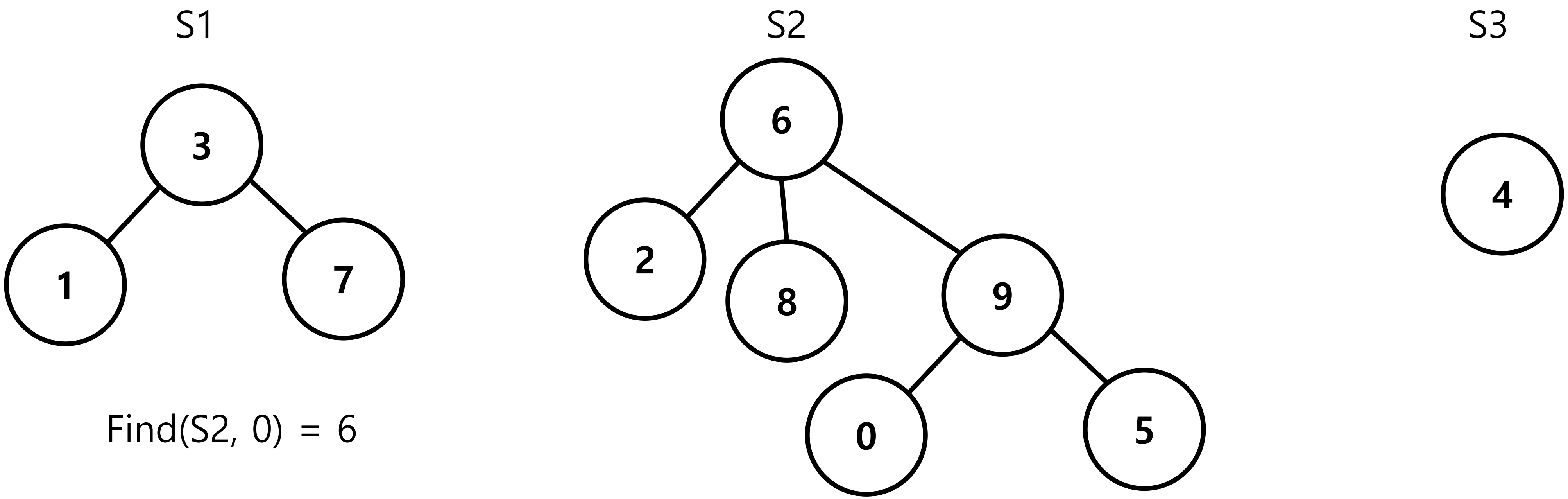
Find(S2, 0) = 6

*arr =

Node	0	1	2	3	4	5	6	7	8	9
Parent	9	3	6	-2	-1	9	-3	3	6	6

Disjoint Set Data Structure

Find



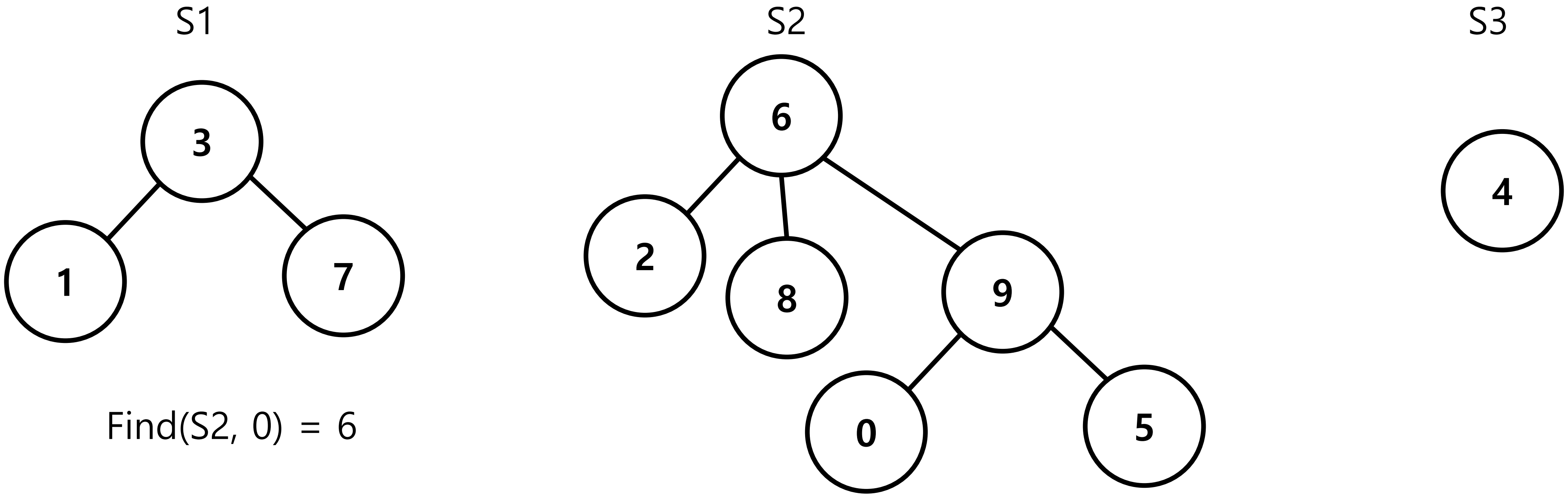
Find(S2, 0) = 6

*arr =

Node	0	1	2	3	4	5	6	7	8	9
Parent	9	3	6	-2	-1	9	-3	3	6	6

Disjoint Set Data Structure

Find



Find(S2, 0) = 6

*arr =

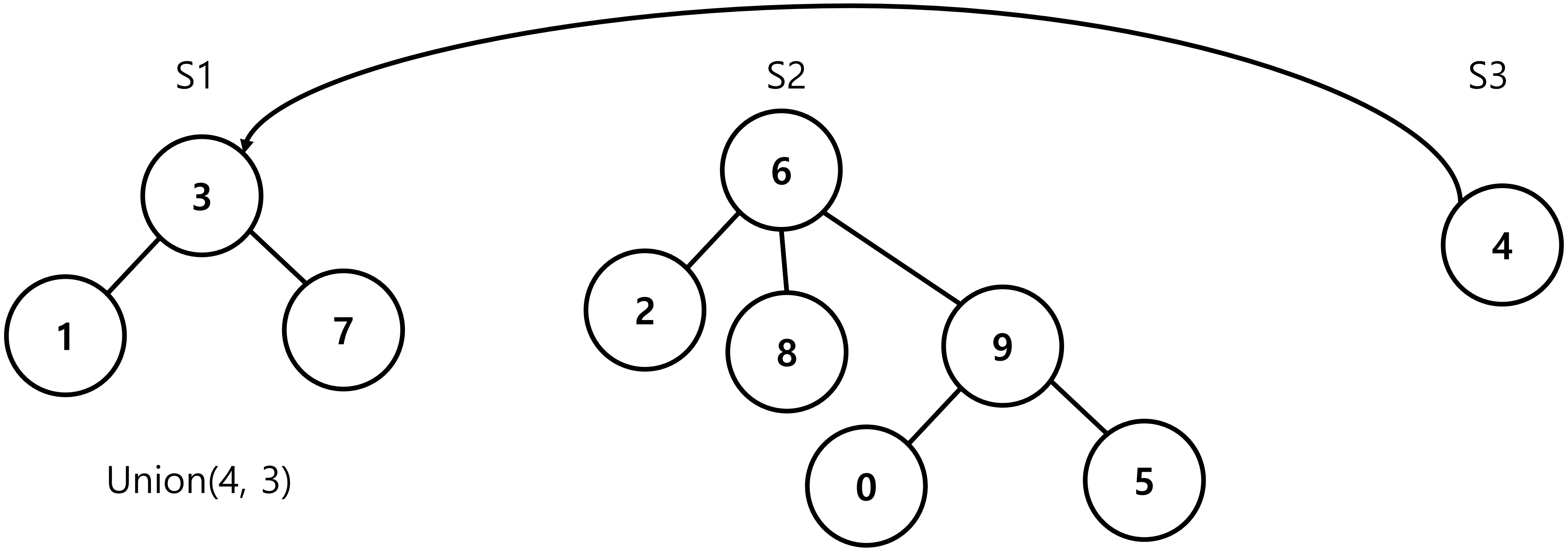
Node	0	1	2	3	4	5	6	7	8	9
Parent	9	3	6	-2	-1	9	-3	3	6	6

Disjoint Set Data Structure ADT

- void Union(DATATYPE *arr, DATATYPE Root1, DATATYPE Root2);
  - Merge trees of Root1 and Root2

Disjoint Set Data Structure

Union



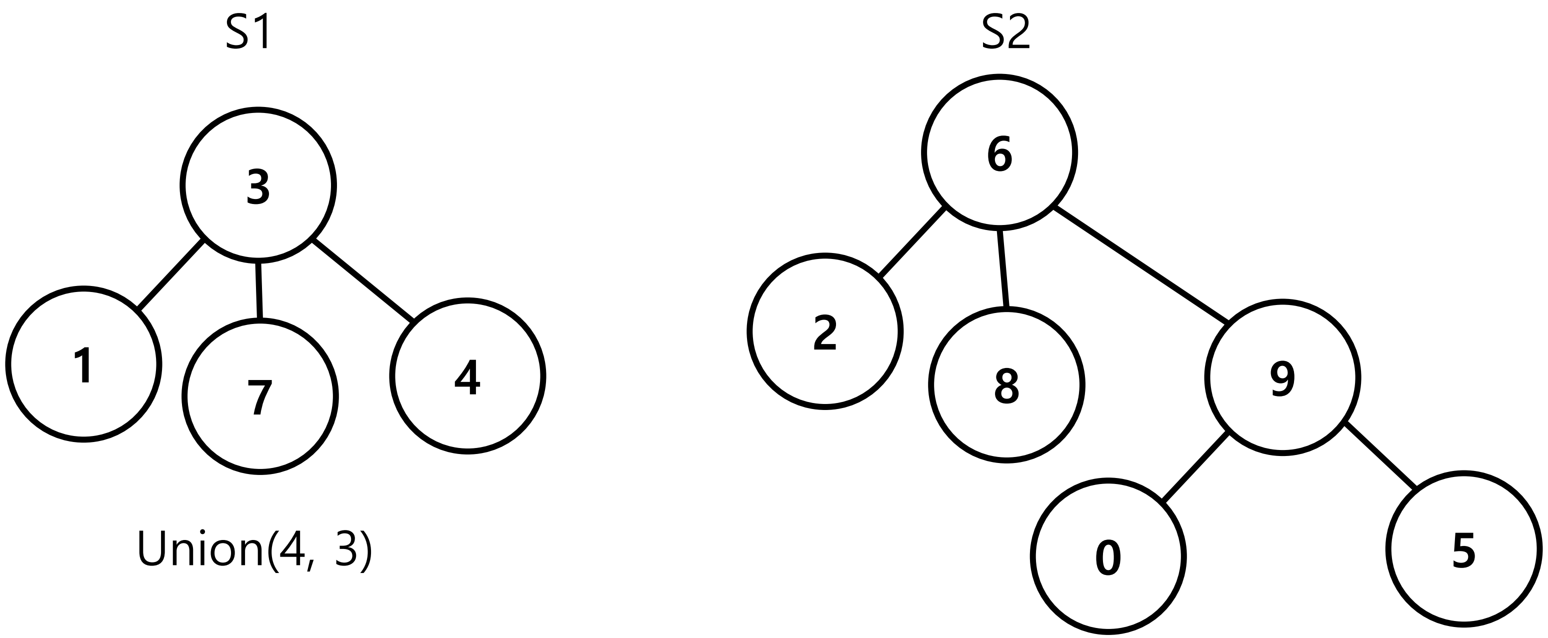
Union(4, 3)

*arr =

Node	0	1	2	3	4	5	6	7	8	9
Parent	9	3	6	-2	-1	9	-3	3	6	6

Disjoint Set Data Structure

Union



Union(4, 3)

*arr =

Node	0	1	2	3	4	5	6	7	8	9
Parent	9	3	6	-2	3	9	-3	3	6	6

Disjoint Set Data Structure ADT

- void PrintArr(DATATYPE *arr);
  - Print arr

Disjoint Set Data Structure ADT

```c
#include <stdio.h>

#define LIST_LEN 10

typedef int DATATYPE;

void Union(DATATYPE *arr, DATATYPE Root1, DATATYPE Root2);
DATATYPE Find(DATATYPE *arr, DATATYPE Node);

void PrintArr(DATATYPE *arr);
```

```c
void Union(DATATYPE *arr, DATATYPE Root1, DATATYPE Root2)
{
    if(arr[Root1] < arr[Root2])
    {
        arr[Root2] = Root1;
    }
    else
    {
        if(arr[Root1] == arr[Root2])
        {
            arr[Root2]--;
        }
        arr[Root1] = Root2;
    }
}

DATATYPE Find(DATATYPE *arr, DATATYPE Node)
{
    if(arr[Node] >= 0)
    {
        return arr[Node] = Find(arr, arr[Node]);
    }
    else
    {
        return Node;
    }
}

void PrintArr(DATATYPE *arr)
{
    for(int i = 0; i < LIST_LEN; i++)
    {
        printf("[%d] : %d\n", i, arr[i]);
    }
//  printf("\n");
}
```

Disjoint Set Data Structure ADT

```c
int main(int argc, char *argv[])
{

    DATATYPE *arr = (DATATYPE*)malloc(sizeof(DATATYPE) * LIST_LEN);
    //index become node id, elements become parent pointer
    //make 3 disjoint set
    //S1 = {1, 3, 7}, 2
    //S2 = {0, 2, 5, 6, 8, 9}, 3
    //S3 = {4}, 1
    arr[0] = 9;
    arr[1] = 3;
    arr[2] = 6;
    arr[3] = -2;
    arr[4] = -1;
    arr[5] = 9;
    arr[6] = -3;
    arr[7] = 3;
    arr[8] = 6;
    arr[9] = 6;

    PrintArr(arr);

    printf("\n");

    int temp1, temp2, temp3;
    int num;
    //find root of 7
    num = 7;
    temp1 = Find(arr, num);
    printf("Parent of Node %d : %d\n", num, temp1);

    //find root of 0
    num = 0;
    temp2 = Find(arr, num);
    printf("Parent of Node %d : %d\n", num, temp2);

    //find root of 4
    num = 4;
    temp3 = Find(arr, num);
    printf("Parent of Node %d : %d\n", num, temp3);

    printf("\n");

    //Union disjoint sets
    printf("Union 3, 4\n");
    Union(arr, 3, 4);
    PrintArr(arr);

    printf("\n");

    printf("Union 4, 6\n");
    Union(arr, 4, 6);
    PrintArr(arr);


    return 0;
}
```

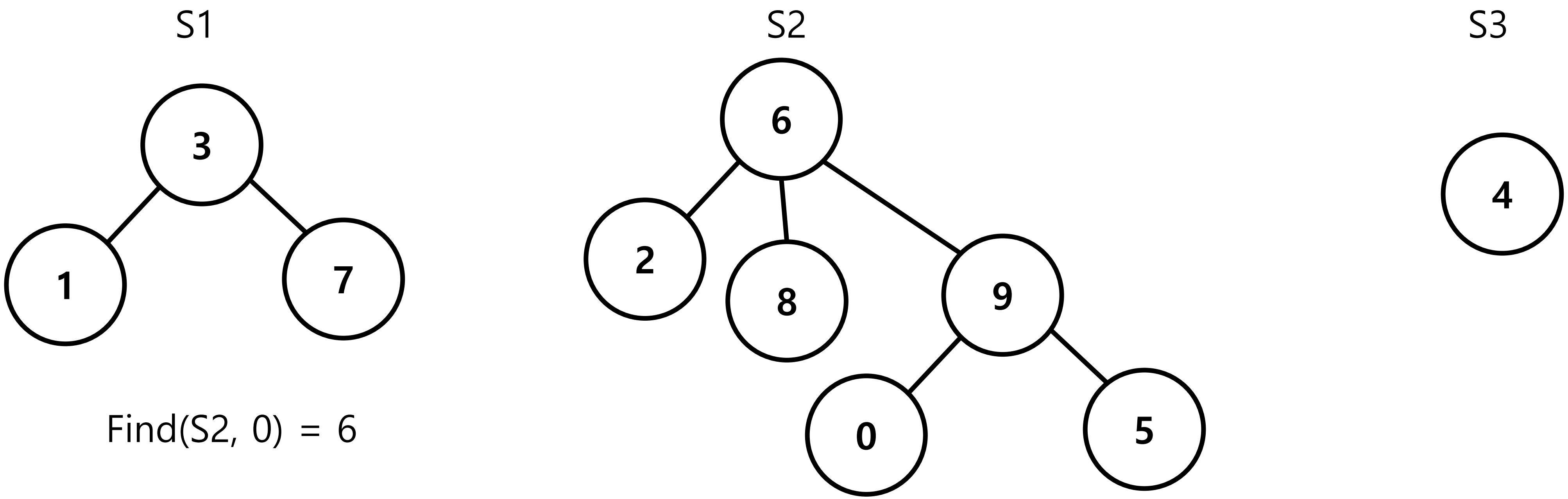
Disjoint Set Data Structure ADT

Path Compression

- DATATYPE Find(DATATYPE *arr, DATATYPE Node);
  - Find root of Node
  - Change parent of Node to its root
  - Return root

Disjoint Set Data Structure

Find – Path Compression



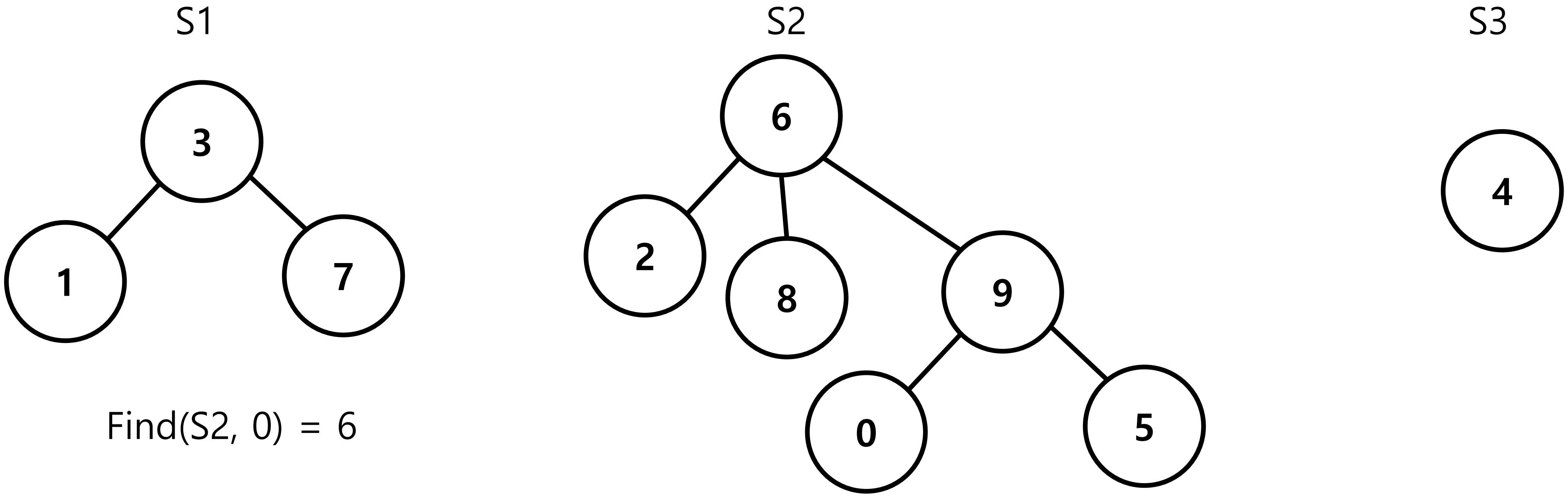
Find(S2, 0) = 6

*arr =

Node	0	1	2	3	4	5	6	7	8	9
Parent	9	3	6	-2	-1	9	-3	3	6	6

Disjoint Set Data Structure

Find



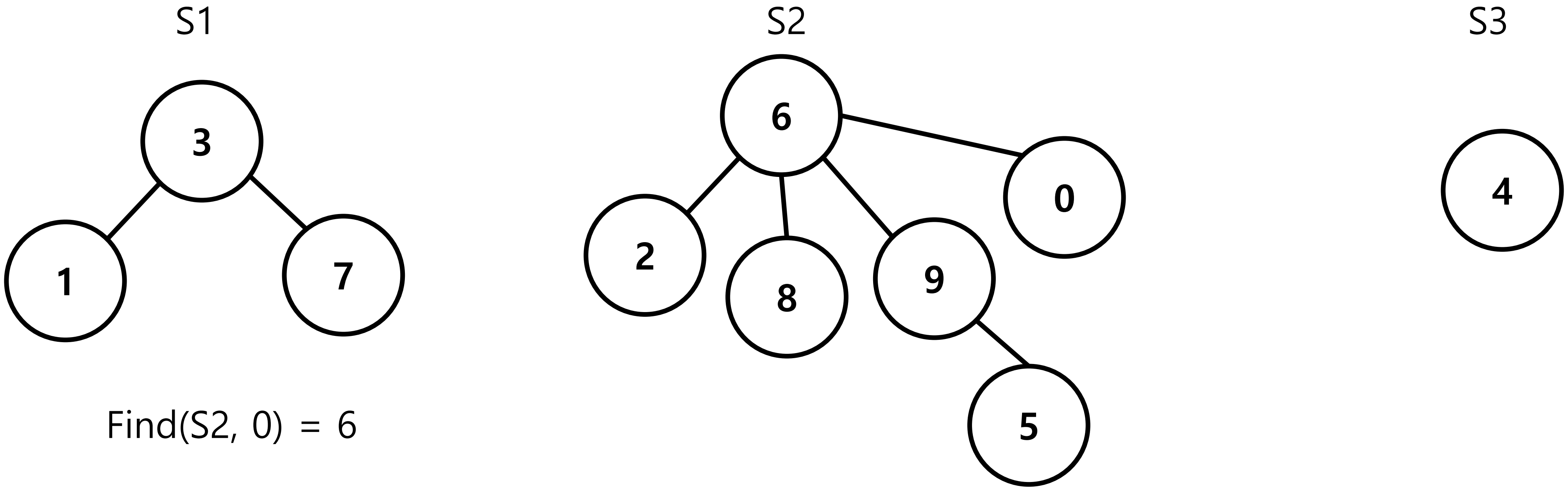
Find(S2, 0) = 6

*arr =

Node	0	1	2	3	4	5	6	7	8	9
Parent	9	3	6	-2	-1	9	-3	3	6	6

Disjoint Set Data Structure

Find



Find(S2, 0) = 6

*arr =

Node	0	1	2	3	4	5	6	7	8	9
Parent	6	3	6	-2	-1	9	-3	3	6	6

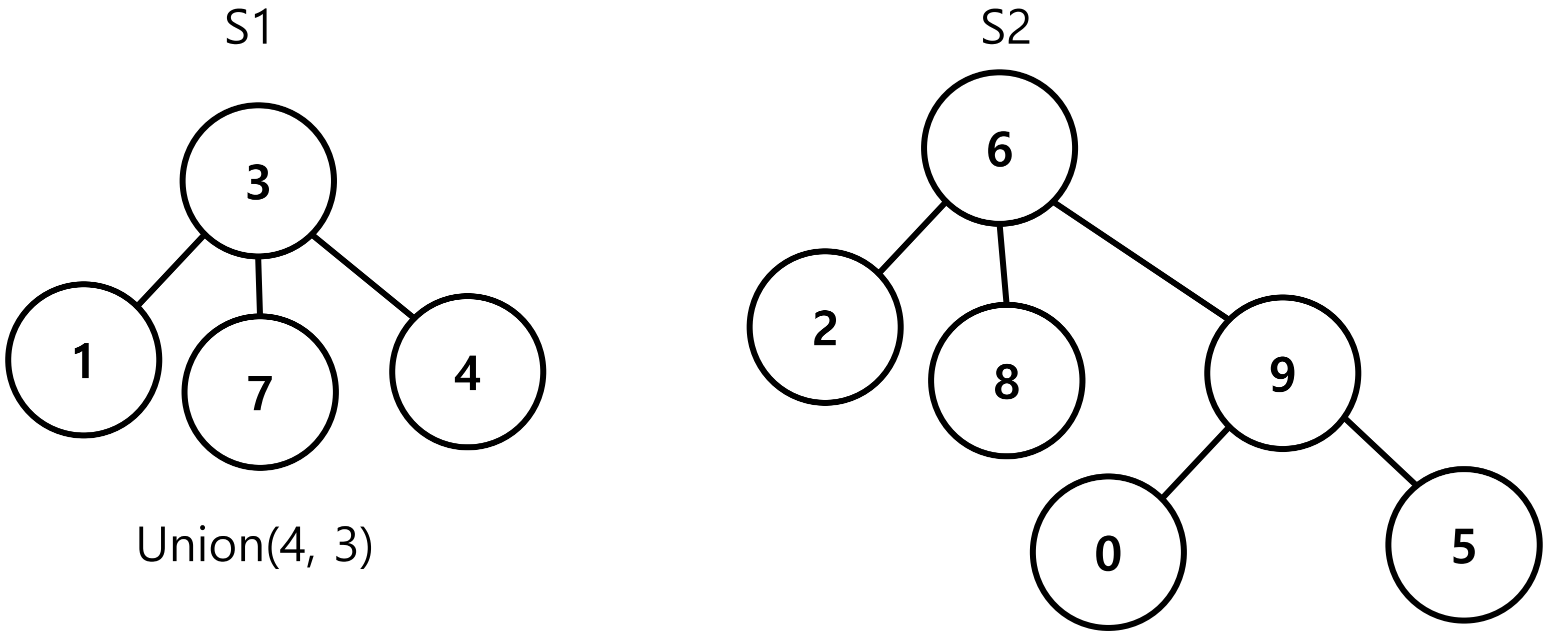
Disjoint Set Data Structure ADT

Path Compression

- void Union(DATATYPE *arr, DATATYPE Root1, DATATYPE Root2);
  - Merge trees of Root1 and Root2
  - Consider the height of the tree to be minimal
  - if height_of_tree1 < height_of_tree2
       merge tree1 into tree2
    else
       merge tree2 into tree1

Disjoint Set Data Structure

Union – Path Compression



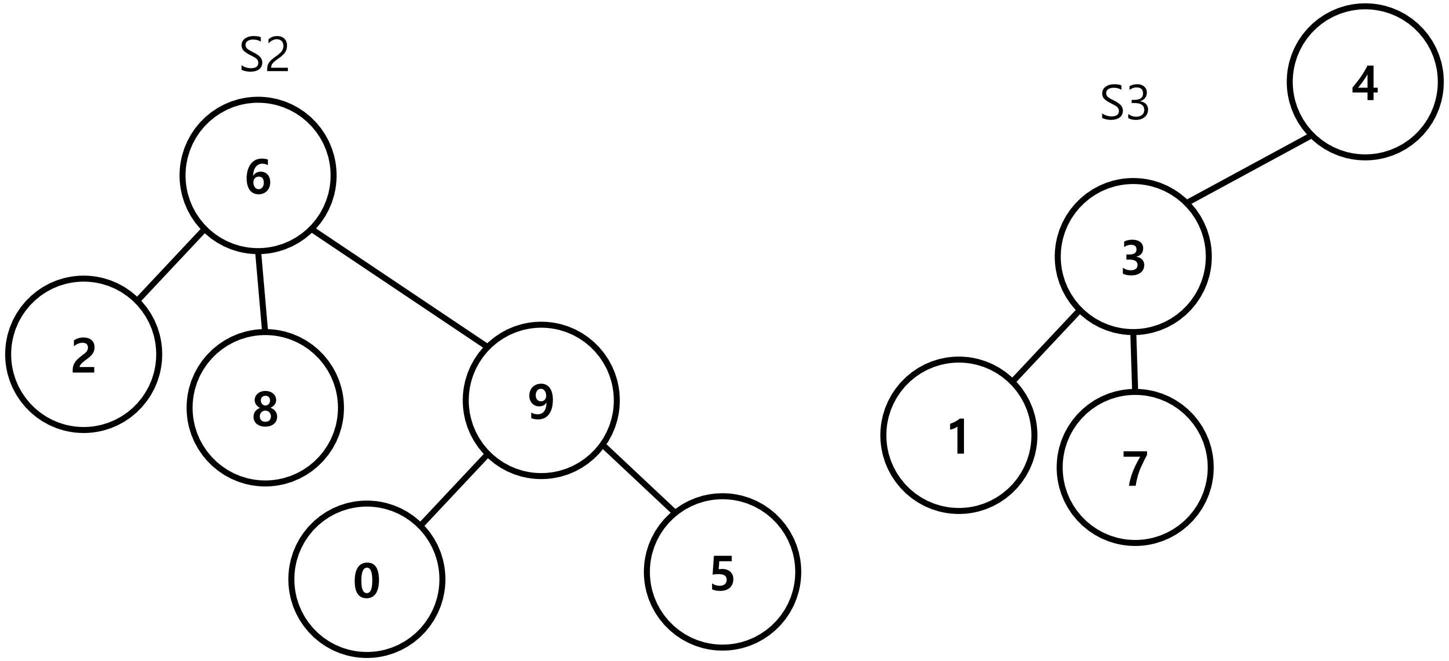
Union(4, 3)

*arr =

Node	0	1	2	3	4	5	6	7	8	9
Parent	9	3	6	-2	3	9	-3	3	6	6

Disjoint Set Data Structure

Union



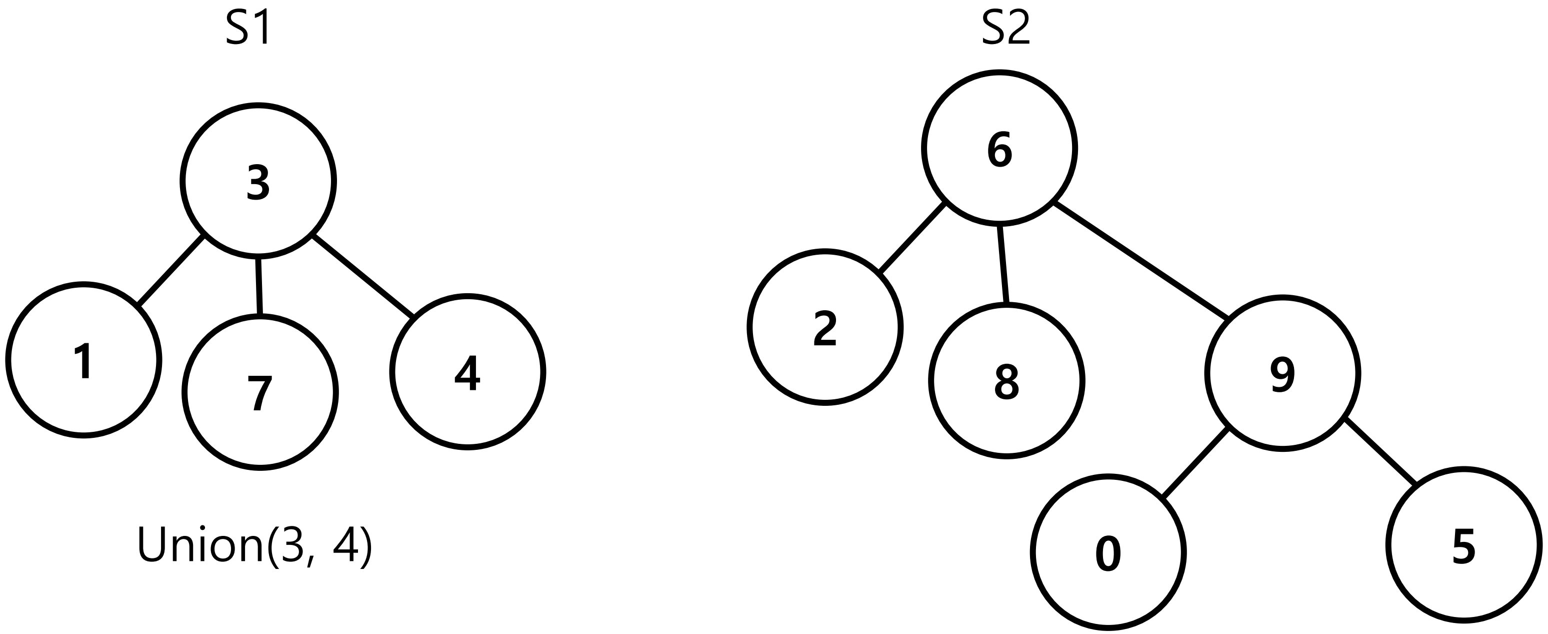
Union(3, 4)

*arr =

Node	0	1	2	3	4	5	6	7	8	9
Parent	9	3	6	4	-4	9	-3	3	6	6

Disjoint Set Data Structure

Union – Path Compression



Union(3, 4)

*arr =

Node	0	1	2	3	4	5	6	7	8	9
Parent	9	3	6	-2	3	9	-3	3	6	6

Disjoint Set Data Structure ADT

Path Compression

```c
#include <stdio.h>

#define LIST_LEN 10

typedef int DATATYPE;

void Union(DATATYPE *arr, DATATYPE Root1, DATATYPE Root2);
DATATYPE Find(DATATYPE *arr, DATATYPE Node);

void PrintArr(DATATYPE *arr);
```

```c
void Union(DATATYPE *arr, DATATYPE Root1, DATATYPE Root2)
{
    if(arr[Root1] < arr[Root2])
    {
        arr[Root2] = Root1;
    }
    else
    {
        if(arr[Root1] == arr[Root2])
        {
            arr[Root1]--;
        }
        arr[Root1] = Root2;
    }
}

DATATYPE Find(DATATYPE *arr, DATATYPE Node)
{
    if(arr[Node] >= 0)
    {
        return arr[Node] = Find(arr, arr[Node]);
    }
    else
    {
        return Node;
    }
}

void PrintArr(DATATYPE *arr)
{
    for(int i = 0; i < LIST_LEN; i++)
    {
        printf("[%d] : %d\n", i, arr[i]);
    }
//  printf("\n");
}
```

Disjoint Set Data Structure

Path Compression

```c
int main(int argc, char *argv[])
{

    DATATYPE *arr = (DATATYPE*)malloc(sizeof(DATATYPE) * LIST_LEN);
    //index become node id, elements become parent pointer
    //make 3 disjoint set
    //S1 = {1, 3, 7}, 2
    //S2 = {0, 2, 5, 6, 8, 9}, 3
    //S3 = {4}, 1
    arr[0] = 9;
    arr[1] = 3;
    arr[2] = 6;
    arr[3] = -2;
    arr[4] = -1;
    arr[5] = 9;
    arr[6] = -3;
    arr[7] = 3;
    arr[8] = 6;
    arr[9] = 6;

    PrintArr(arr);

    printf("\n");

    int temp1, temp2, temp3;
    int num;
    //find root of 7
    num = 7;
    temp1 = Find(arr, num);
    printf("Parent of Node %d : %d\n", num, temp1);

    //find root of 0
    num = 0;
    temp2 = Find(arr, num);
    printf("Parent of Node %d : %d\n", num, temp2);

    //find root of 4
    num = 4;
    temp3 = Find(arr, num);
    printf("Parent of Node %d : %d\n", num, temp3);
    printf("\n");

    PrintArr(arr);
    printf("\n");

    //Union disjoint sets
    printf("Union 3, 4\n");
    Union(arr, 3, 4);
    PrintArr(arr);

    printf("\n");

    printf("Union 3, 6\n");
    Union(arr, 3, 6);
    PrintArr(arr);

    printf("\n");

    //path compression
    for(int i = 0; i < LIST_LEN; i++)
    {
        Find(arr, i);
    }
    PrintArr(arr);

    return 0;
}
```

Lab7:Disjoint Set Data Structure

Using Binary Tree and Linked List

- Submit on GitLab
- Disjoint set data structure using binary tree and linked lists
- Create Lab7 directory on your own GitLab project
- Submit file : source_code(c only, run on linux)
- Filename : StudentID_lab7.c
- Input file : no

Lab7:Disjoint Set Data Structure

Using Binary Tree and Linked List

- Note :
  - Write codes in given file
  - Do not change other codes in given file(including main)
  - Complete blank functions : Insert, Search, RetTotalHeight,
    MakeAddressMapWithSubTree, Find, Union
  - Output should be same as the example given

Lab7:Disjoint Set Data Structure

Using Binary Tree and Linked List

```
donghyeonkim@donghyeonkim-System-Product-Name:~/Hanyang/classes_2020/DS/week7/DisjointSetWithTree$ ./a.out
Insert 3
Insert 7
Insert 1

Print S1

            7

3

            1

Print Address Map
Node 1, Parent 3
Node 7, Parnet 3
Node 3, Parnet -1
```

Lab7:Disjoint Set Data Structure

Using Binary Tree and Linked List

```
Insert 10
Insert 8
Insert 2
Rotate LL
Insert 9
Insert 6
Insert 5
Rotate RL

Print S2

                10

                                9

8

                                6

                5

                                2


Print Address Map
Node 2, Parent 5
Node 6, Parnet 5
Node 5, Parnet 8
Node 9, Parnet 10
Node 10, Parnet 8
Node 8, Parnet -2
```

Lab7:Disjoint Set Data Structure

Using Binary Tree and Linked List

```
Insert 4

Print S3

4

Print Address Map
Node 4, Parent 0

Find roots
Node 1, Root 3
Node 2, Root 8
Node 3, Root 3
Node 4, Root 4
Node 5, Root 8
Node 6, Root 8
Node 7, Root 3
Node 8, Root 8
Node 9, Root 8
Node 10, Root 8
```

Lab7:Disjoint Set Data Structure

Using Binary Tree and Linked List

```
Union

            10

                        9

8

                                    7

                        6

            5

                                    3

                        2

                                    1

Node 1, Parent 2
Node 3, Parnet 2
Node 2, Parnet 5
Node 7, Parnet 6
Node 6, Parnet 5
Node 5, Parnet 8
Node 9, Parnet 10
Node 10, Parnet 8
Node 8, Parnet -3
```

Lab7:Disjoint Set Data Structure

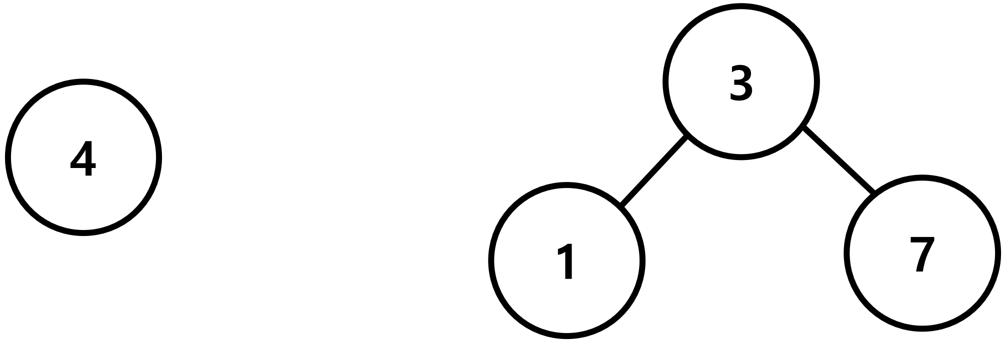
Using Binary Tree and Linked List

```
Rotate LL

                        10

                                    9

            8

                                    7

                        6

5

                                    4

                        3

            2

                        1

Node 1, Parent 2
Node 4, Parnet 3
Node 3, Parnet 2
Node 2, Parnet 5
Node 7, Parnet 6
Node 6, Parnet 8
Node 9, Parnet 10
Node 10, Parnet 8
Node 8, Parnet 5
Node 5, Parnet -3
```

Lab7:Disjoint Set Data Structure

Using Binary Tree and Linked List

- Given file based on AVL and linked list using dummy node
- AVL is used for disjoint set, linked list is used for address map
- Nodes are not including 0
- Integet <= 0 means height in address map
  - ex) 0 – only root exist, -1 – root and child exists



Lab7:Disjoint Set Data Structure

Using Binary Tree and Linked List

- void Insert(BTTree *Tree, DATATYPE Data, DATATYPE Parent);
  - Insert Data and Parent into linked list in Tree
  - No sorting
  - Using dummy node
  - Insert new node at the end of the list

Lab7:Disjoint Set Data Structure

Using Binary Tree and Linked List

- DATATYPE Search(BTTree *Tree, DATATYPE *Data);
  - Find Data(Target) in the linked list and return its parent
  - Use PosHead and PosNext

- int RetTotalHeight(BTTree *Tree);
  - Return height of tree
  - Use RetHeight
  - Used in MakeAddressMap

Lab7:Disjoint Set Data Structure

Using Binary Tree and Linked List

- DATATYPE MakeAddressMapWithSubTree(BTTree *Tree, BTNode *Node);
  - Make address map
  - Call itself recursively
  - Returned value of MakeAddressMapWithSubTree is child of Node
  - Insert Node(Data) and its parent into linked list
  - Used in MakeAddressMap

Lab7:Disjoint Set Data Structure

Using Binary Tree and Linked List

- DATATYPE Find(BTTree *Tree, DATATYPE *Data);
    - Find root of Data in tree
    - No path compression
    - Use Search and recursive call of itself

Lab7:Disjoint Set Data Structure

Using Binary Tree and Linked List

- int Union(BTTree *Tree1, BTTree *Tree2);
  - Union Tree1 and Tree2
  - Consider the height of the tree to be minimal
  - Find each height from the linked list (using Search)
  - Compare heights and insert all nodes in lower height tree to higher height tree
  - When searching all nodes in tree, use PosHead and PosNext
  - When inserting nodes into other tree, use InseartBST
  - After all nodes inserted, remake address map of tree
  - Return 1 when tree2 inserted into tree1, and return 2 when tree1 inserted into tree2