Disjoint Sets

Dr. Tanvir Ahmed

Set Theory Review

- Before learning about the new data structure, it is prudent to explore the concept of sets in mathematics. This way you understand the mathematical collection from which this data structure originates.
- In mathematics, sets are unordered collections without duplicates. Objects that are contained inside sets are called elements of that set.
- When writing out sets mathematically we can use the following notation (called the roster method):
- $\{apple, orange, peaches, grapes\}$

$$\{1, 2, 3, 4, 5\}$$

- Notice that the two following sets are equal under our definition:
- $\{1, 3, 2\} = \{2, 1, 3\}$
- Two sets A and B are equal if the elements in A are all contained in set B and all elements of set B are contained in set A. Also notice that each element may be contained only once in the set. We cannot have the following set:
- $\{orange, orange\}$

Set Theory Review

- There are some other useful definitions to know about sets.
- A set A is called a subset of set B if all elements in set A are also in set B. We use the following notation to denote subset:
- $A \subseteq B$
- An alternative definition of set equality is $A = B$ if and only if $A \subseteq B$ and $B \subseteq A$.
- Another piece of notation from set theory is the element of symbol. We use this to denote when one element is contained in a set. So the following is a true statement:
- $apple \in \{apple, peach, orange\}$

Set Theory Review

- **Sets also have a few handy operations:**

- Set Union
- Given two sets A and B, the union of those sets is defined as the set containing all elements in either A or B. Union is denoted $A \cup B$.

- $\{apple, orange, peach\} \cup \{apple, tomato, carrot\} = \{apple, orange, peach, tomato, carrot\}$

- Set Intersection
- Given two sets A and B, the intersection of those sets is defined as the set containing all elements in both A and B. Intersection is denoted $A \cap B$.

- $\{apple, orange, peach\} \cap \{apple, tomato, carrot\} = \{apple\}$

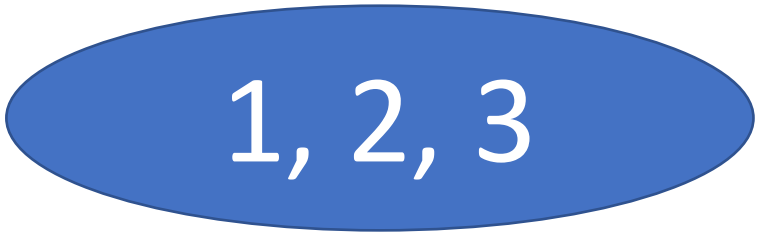
- Set Difference
- Given two sets A and B, the set difference of those sets is defined as the set containing all elements in A but not in B. Set difference is denoted $A - B$.

- $\{apple, orange, peach\} - \{apple, tomato, carrot\} = \{orange, peach\}$

- Set theory goes much deeper than this but a brief introduction to these concepts will greatly aid in your understanding of not only the data structures below but more advanced concepts based off Set Theory such as Inclusion-Exclusion, the Disjoint Sets data structure, and brute force concepts enumerating sets.

Disjoint Set

- Two or more sets with no common elements called disjoint sets. Consider, there are two sets S1 = {1, 2 , 3} and S2 = {4, 5}
- As S1 ∩ S2 = Ø Empty set (S1 and S2 have nothing in common), we can call them disjoint set



S1

S2

Use of Disjoint Set

- It can help us to keep track of the set where an element belongs to
- It is easier to check whether two given elements belong to the same subset or not.
- **This is called find operation**
- For example, if S1 = {1, 2, 3, 4} and S2 = {5, 6, 7}
- Then find(1) == find(2) will return true as both 1 and 2 belong to the same set.

Use of Disjoint Set

- It can help us to merge 2 sets in to one
- **This is called Union operation**
- For example, if S1 = {1, 2, 3, 4} and S2 = {5, 6, 7}
- Then Union will be {1, 2, 3, 4, 5, 6, 7}
- **In disjoint set,** Union (2, 6) will give you the same set as like Union (1, 5)
- , Union (2, 6) will find the set where 2 belongs to and finds the set where 6 belongs to and the do the union.
- Similarly, Union (1, 5) will do the same.

Representative/Marker/ Absolute Root

- In disjoint sets, each set is represented by an element in the set.
- That element is known as marker. Also, known as representative or absolute root.
- Here is a simple disjoint set (each of them is an individual set):
  - {1}, {2}, {3}, {4}, {5}
- clearly there can only be one marker for each of these sets.
- So, in the above list of sets, each number is the representative of its own set

Disjoint set operations

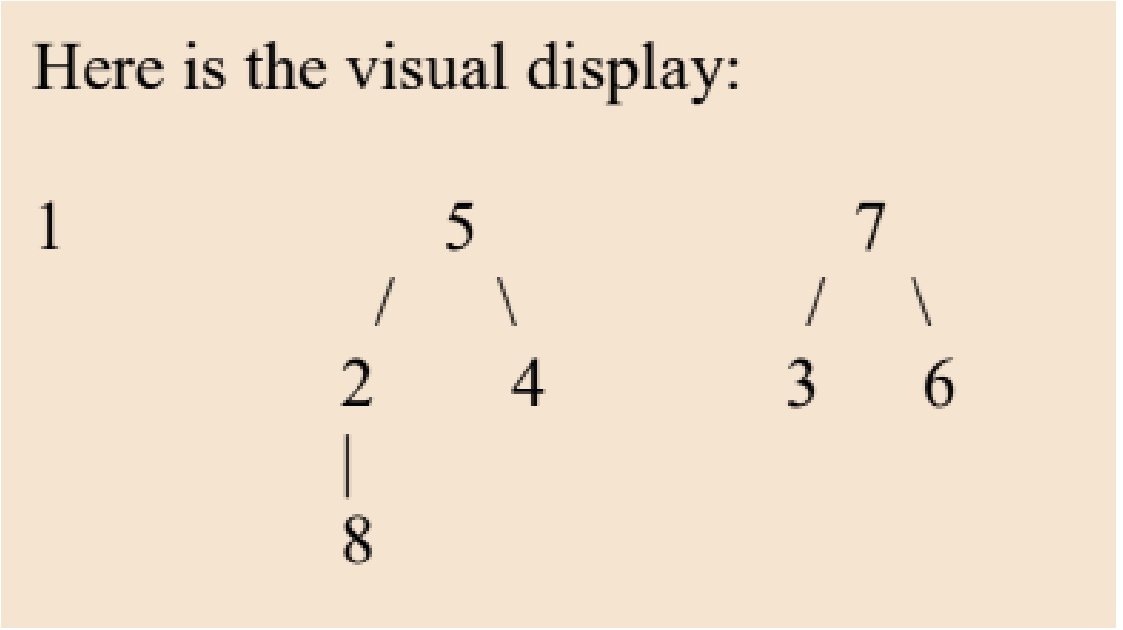
- ***Make Set****: Creates a new disjoint set whose only member is the given element.*
- ***Find****: Finds the root of a given element and returns it. It is used to check whether the elements are in the same set or not.*
- ***Union****: Forms a new set that is the union of two disjoint sets.*

Disjoint set operations

- Make Set(5) (assuming first item is 1 (not 0)):
  - {1}, {2}, {3}, {4}, {5}
- For example:
- union(1,3): would make our structure look like: {1,3}, {2}, {4}, {5}
- Here we would have to designate either 1 or 3 as the marker. Let's choose 1.
- Now consider doing these two operations: union(1,4) and union(2,5) (Assume 2 is marked.)
- Now we have: {1,3,4}, {2,5}
- Now, we can also do the findset operation. findset(3) should return 1, since 1 is the marked element in the set that contains 3

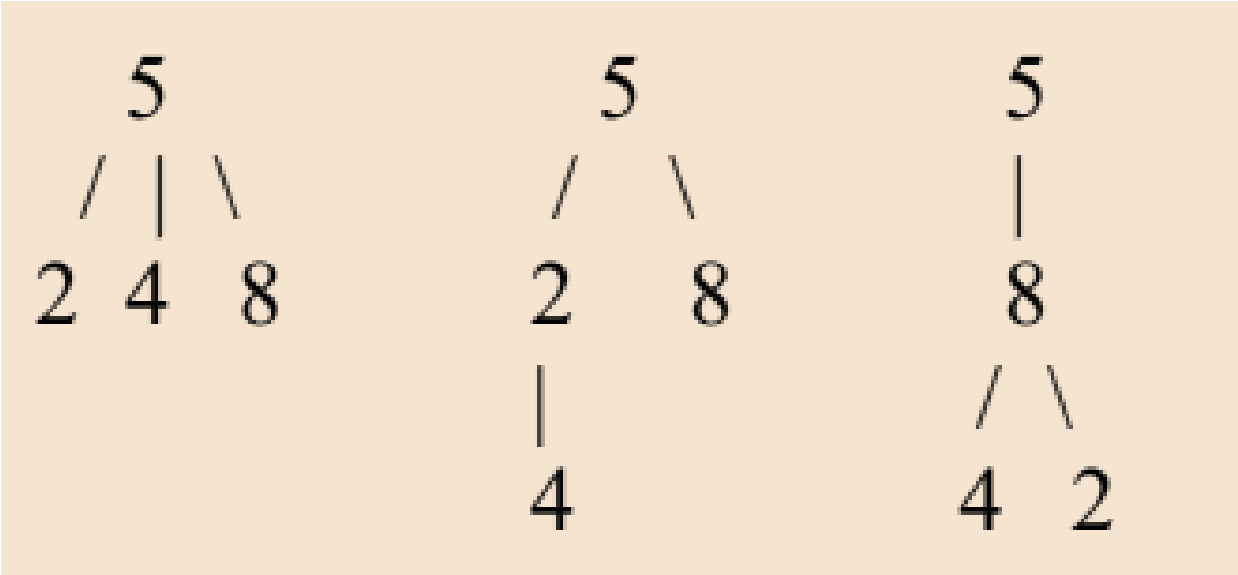
Representing disjoint sets Forest implementation

- We can represent disjoint sets as a forst (multiple trees)
- Example: {2,4,5,8}, {1}, {3,6,7} could be stored as follows:



Disjoint set implementation

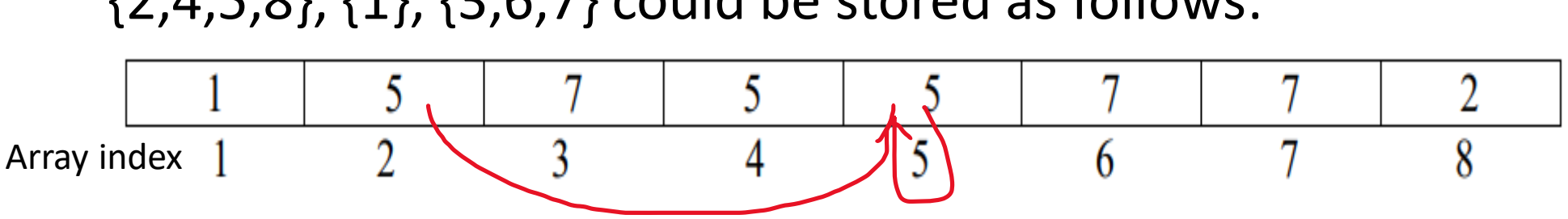
- A set within disjoint sets can be represented in several ways.
- Consider {2, 4, 5, 8} with **5 as the marked element/absolute root.**
- **The marked element's parent is the marked element itself**
- Here are a few ways that could be stored:

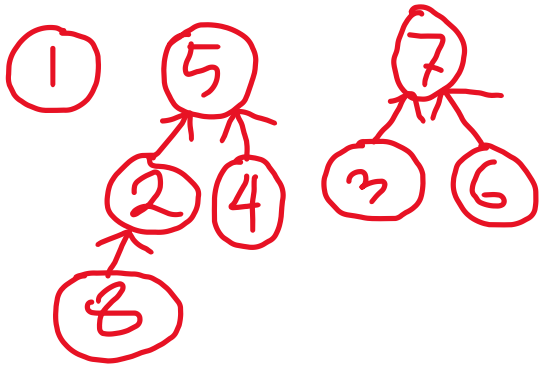


For the first tree, find (4) will give 5, and then find 5 will give 5 itself
For the second tree, find(4) will give 2, then find 2 will give 5, then find 5 will give 5 itself

Representing disjoint sets: Forest Implementation using an array

- We can actually store a disjoint set in an array. For example, the sets {2,4,5,8}, {1}, {3,6,7} could be stored as follows:

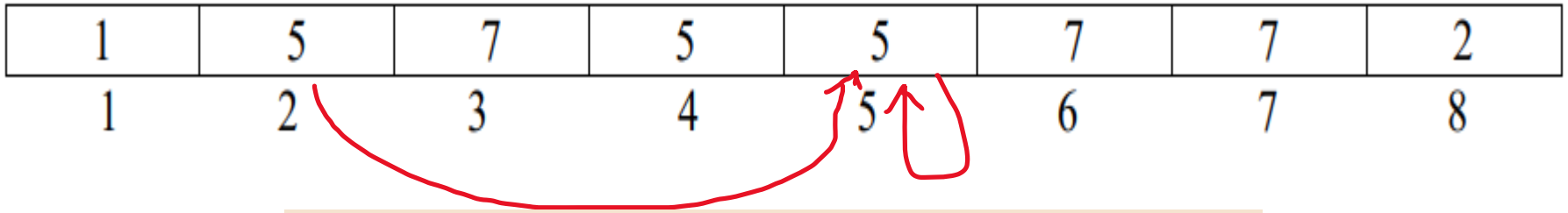




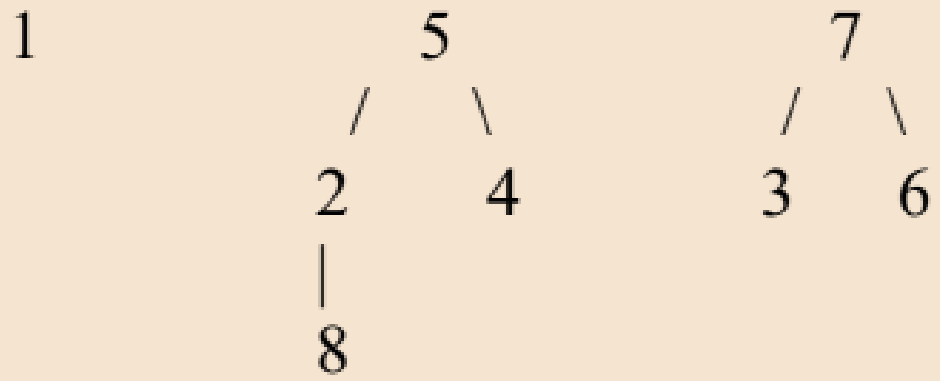
- The 5 stored in array location 2 signifies that 5 is 2's parent.
- 5 at location 5 represent, 5 is parent of itself. Thus, 5 is the representative of this set.
- Let's say we would like to find(8). (the representative of the set where 8 belongs to)
- We go to index it and find that 2 is the parent, then we go to index 2 and found 5 is the parent and then we go to index 5 and found 5 is the parent. So, 5 is the representative of the set where 8 belongs to

Representing disjoint sets Forest implementation using an array

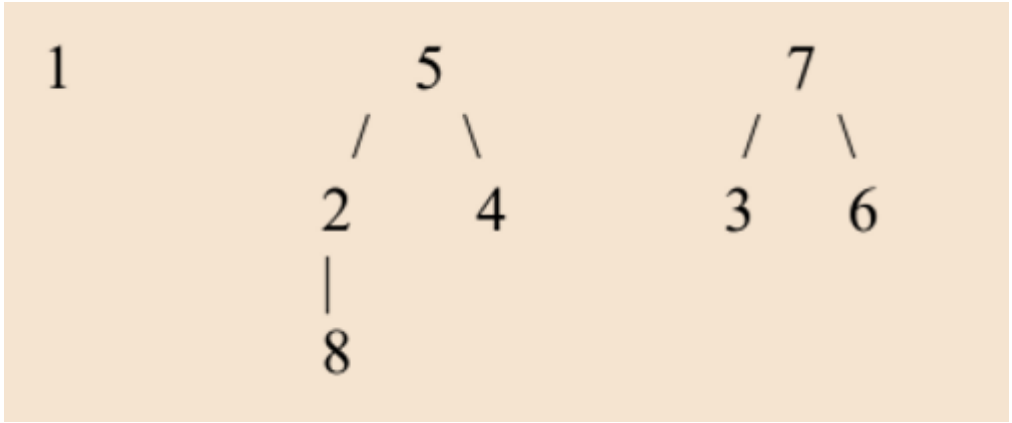
- We can actually store a disjoint set in an array. For example, the sets {2,4,5,8}, {1}, {3,6,7} could be stored as follows:



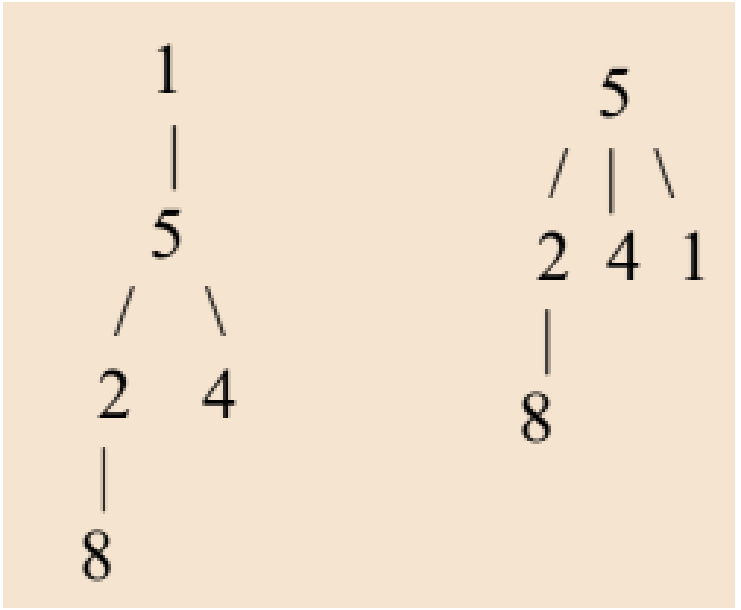
Here is the visual display:



Union Operation



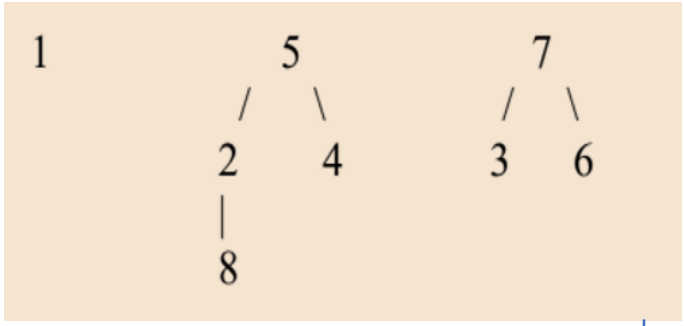
- Given two values, we must first find the markers for those two values, then merge those two trees into one. Consider union(5,1). We could do either of the following:

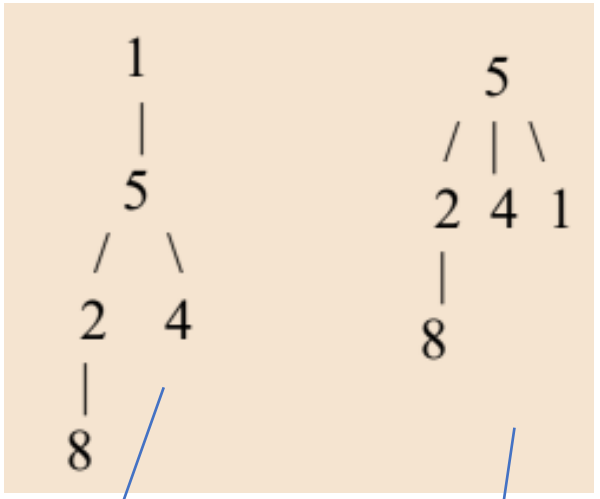


- Which of the following representation would you prefer?

Union Operation by Rank (one optimization over the regular disjoint set)

- We prefer the latter since it minimizes the height of the tree. We keep the height (aka rank in this context) with the root.
- Thus, in order to implement our disjoint sets efficiently, we must also keep track of the height of each tree, so we know how to do our merges.





- Basically, we choose which tree to merge with which based on which tree has a smaller height. If they are equal we are forced to add 1 to the height of the new tree. In this example, 1 has lower rank, so its parent is becoming 5

1	5	7	5	5	7	7	2
1	2	3	4	5	6	7	8

union(5,1).

First option

1	5	7	5	1	7	7	2
1	2	3	4	5	6	7	8

Second option

5	5	7	5	5	7	7	2
1	2	3	4	5	6	7	8

Notice how quickly we can implement that change in the array!

Run-time without any optimization

- Without optimization:
  - Find-set – Best will be O(height). Worst will be O(n)
  - Union – O(1) + 2T_findset => O(n)

- With Rank based optimization:
  - It turns out that the rank of a tree with k elements is always at most lgk.
  - Thus, the worst-case performance of a disjoint-set forest with union by rank having n elements is:
    - FIND-SET Θ(lgn)
    - UNION Θ(lgn).

Let us do a complete example

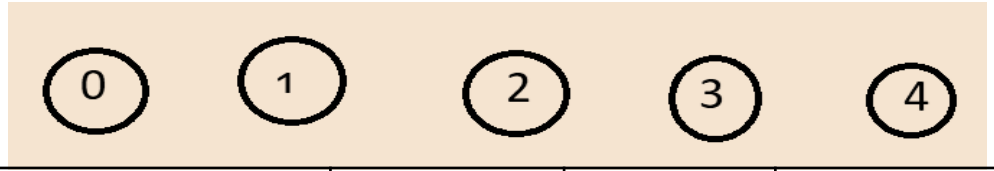
- Let’s say, n = 5, it means I would like to create disjoint sets for numbers from 0 – 4
- Makeset(5) =>So, we are initially creating 5 sets:

{0} {1} {2} {3} {4}

parent	0	1	2	3	4
Index/node	0	1	2	3	4

Let us do a complete example (1)

{0} {1} {2} {3} {4}



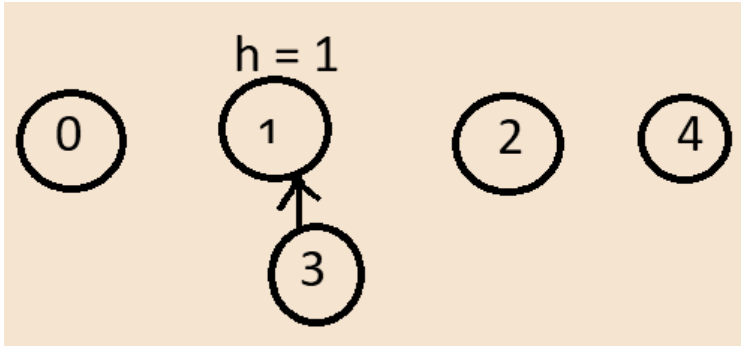
🡐 Current status of the sets and arrays

parent	0	1	2	3	4
Index/node	0	1	2	3	4

- **Process: Union(1 , 3)**
  - Find(1) = 1, find(3) = 3, both set 1 and set 3 has same height
  - So, we can make any as the parent of the other one.
  - We are making 1 as the parent of 3 (so, 1 is representing the updated set)
  - and we need to increase the height

Updated status of the sets and arrays

{0} {1, 3} {2} {4}



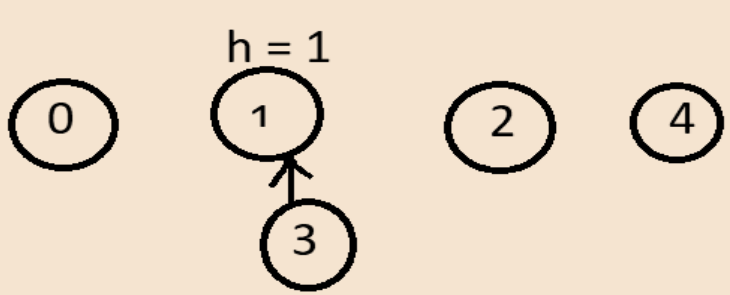
parent	**0**	**1**	**2**	**1**	**4**
Index/node	**0**	**1**	**2**	**3**	**4**

Let us do a complete example (2)

{0} {1, 3} {2} {4}

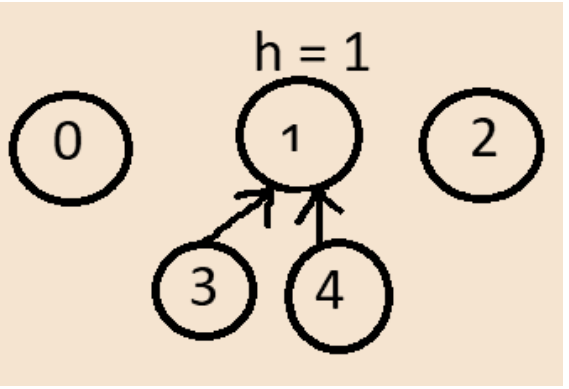
parent	0	1	2	1	4
Index/node	0	1	2	3	4

← Current status of the sets and arrays



- **Process: Union(3 , 4)**
  - Find(3) = 1, find(4) = 4, height (1) > height (4),
  - So, we are making 1 as the parent of 4 (so, 1 is representing the updated set)
  - We don't need to update height for such cases. We do it only when the heights are same

Updated status of the sets and arrays



{0} {1, 3, 4} {2}

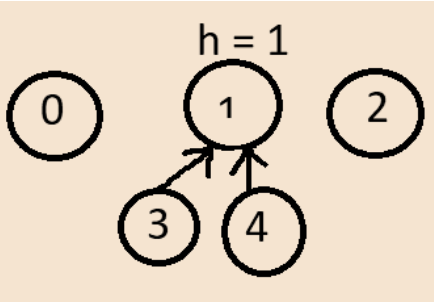
parent	0	1	2	1	1
Index/node	0	1	2	3	4

Let us do a complete example (2)

{0} {1, 3, 4} {2}

parent	0	1	2	1	1
Index/node	0	1	2	3	4

← Current status of the sets and arrays

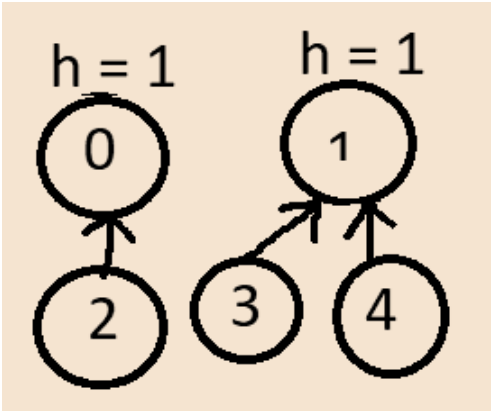


- **Process: Union(0 , 2)**
  - Find(0) = 0, find(2) = 2, height (0) == height (2),
  - So, we can make any as the parent of the other one. We are making 0 as the parent of 2 (so, 0 is representing the updated set)
  - and we need to increase the height

Updated status of the sets and arrays

{0, 2} {1, 3, 4}

parent	0	1	0	1	1
Index/node	0	1	2	3	4

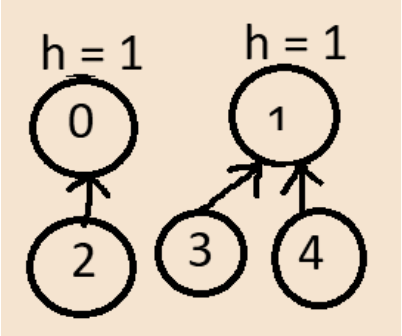


Let us do a complete example (3)

{0, 2} {1, 3, 4}

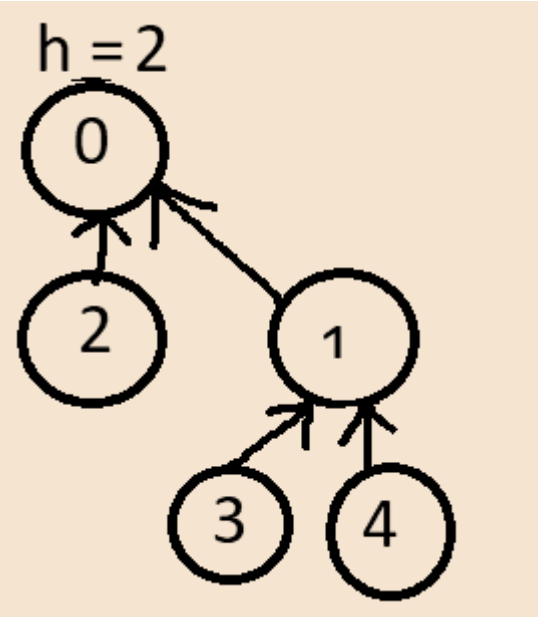
parent	0	1	0	1	1
Index/node	0	1	2	3	4

← Current status of the sets and arrays



- **Process: Union(2 , 3)**
  - Find(2) = 0, find(3) = 1, height (0) == height (1),
  - So, we can make any one (0 or 1) as the parent of the other one. We are making 0 as the parent of 1 (so, 0 is representing the updated set)
  - and we need to increase the height

Updated status of the sets and arrays



{0, 2, 1, 3, 4}

parent	0	0	0	1	1
Index/node	0	1	2	3	4

Coding Disjoint Sets

- You can implement the concept in many different ways
- Let us plan how to implement it.
- We will take n as a user input. Let's say, if n = 5, then we should initially make sets {0}, {1}, {2}, {3}, {4}, {5}
- In order to make and maintain the sets, we need an array of parents. Also, initially the height of each of the set will be 0.
- As we need to maintain height for each array parent, we can create a class for pair.
- We need something like:
- `parents = new pair[n]; //So, for a parents[i], i will represent the node, and prents[i] will represent the parent of element i`

Coding Disjoint Sets

- After getting n, we are calling the constructor to make all the sets

```
private pair[] parents;

  // Create the initial state of a disjoint set of n
elements, 0 to n-1.
  public Main(int n) {

    // All nodes start as leaf nodes.
    parents = new pair[n];
    for (int i=0; i<n; i++)
      parents[i] = new pair(i, 0); //0 is height.
parent[i]'s parent is i now
  }
```

Our Pair class

```
class pair {

  private int ID;
  private int height;

  public pair(int myNum, int myHeight) {
    ID = myNum;
    height = myHeight;
  }

  public int getHeight() {
    return height;
  }

  public int getID() {
    return ID;
  }

  public void incHeight() {
    height++;
  }

  public void setID(int newID) {
    ID = newID;
  }
}
```

Coding Disjoint Sets

- We would like to keep asking user whether the user would like to exit or do a union.
- If the user would like to do union, we will take input on what items and based on that we will do union
- During the union operation, we will need to use the find function

```
// Returns the root node of the tree storing id.
public int find(int id) {

  // Go up tree until there's no parent.
  while (id != parents[id].getID())
    id = parents[id].getID();

  return id;
}
```

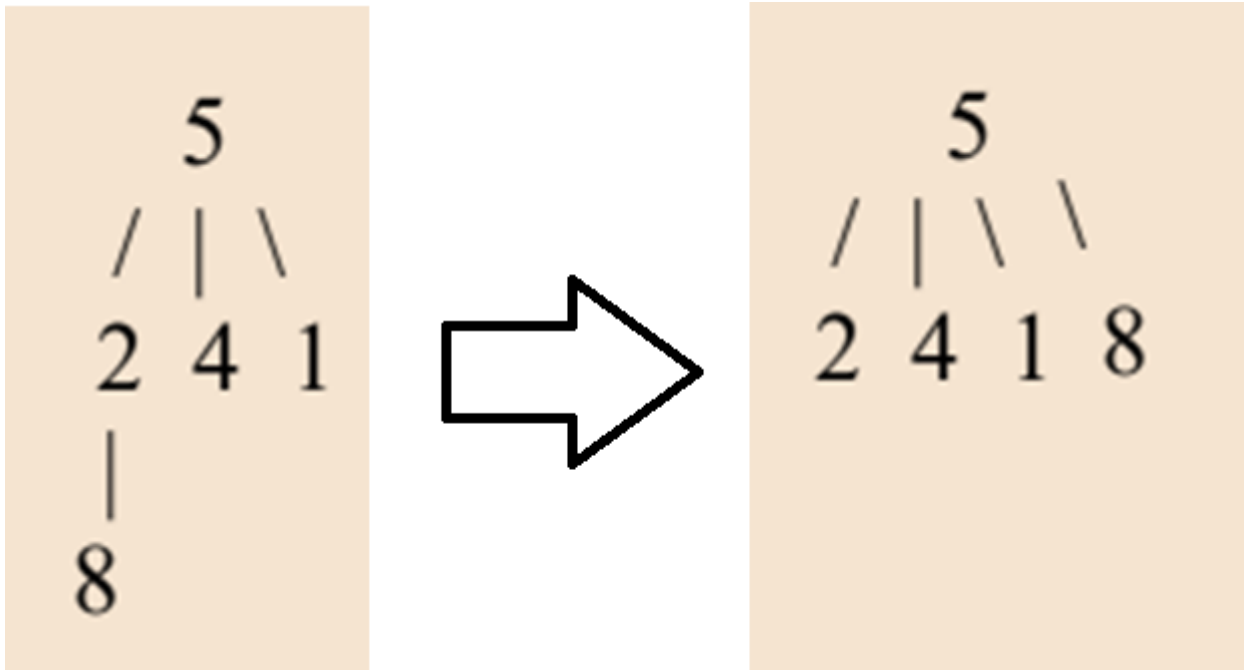
```
public boolean union(int id1, int id2) {
    // Find the parents of both nodes.
    int root1 = find(id1);
    int root2 = find(id2);
    // No union needed.
    if (root1 == root2)
      return false;
    // Attach tree 2 to tree 1
    if (parents[root1].getHeight() > parents[root2].getHeight()) {
      parents[root2].setID(root1);
    }
    // Attach tree 1 to tree 2
    else if (parents[root2].getHeight() > parents[root1].getHeight() ) {
      parents[root1].setID(root2);
    }
    // Same height case - just attach tree 2 to tree 1, adjust height.
    else {
      parents[root2].setID(root1);
      parents[root1].incHeight();
    }
    // We successfully did a union.
    return true;
  }
```

As part of displaying our array, we are also overriding toString() function

```
// Just represents this object as a list of each node's parent.
  public String toString() {

    String ans = "";
    for (int i=0; i<parents.length; i++)
    {//if root, display height
        if (i == parents[i].getID())
                ans = ans + "(" + i + ", " + parents[i].getID() + ":" +
        parents[i].getHeight()+") ";
        else
        ans = ans + "(" + i + ", " + parents[i].getID() + ") ";
    }
    return ans;
  }
```

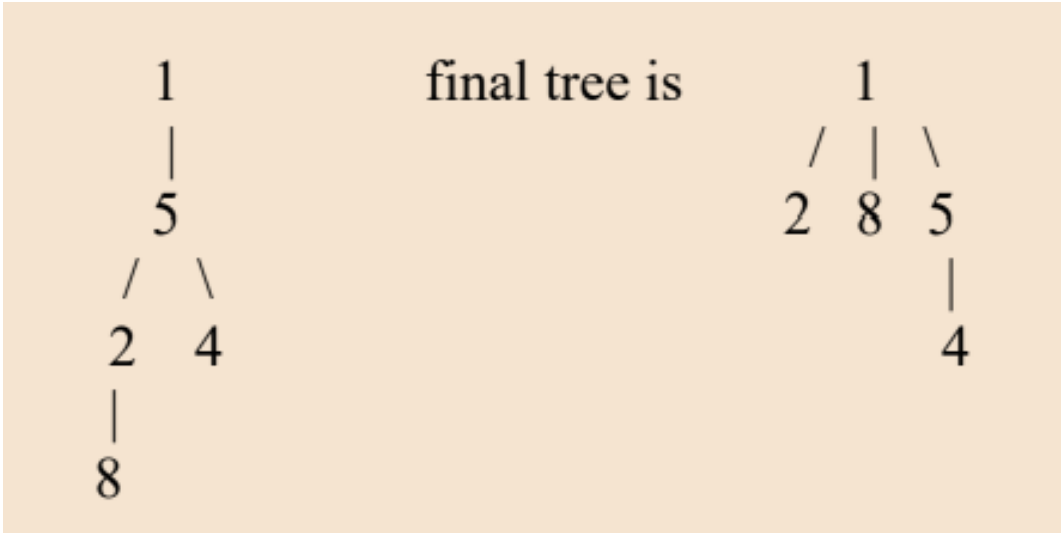
The complete code will be available in webcouses

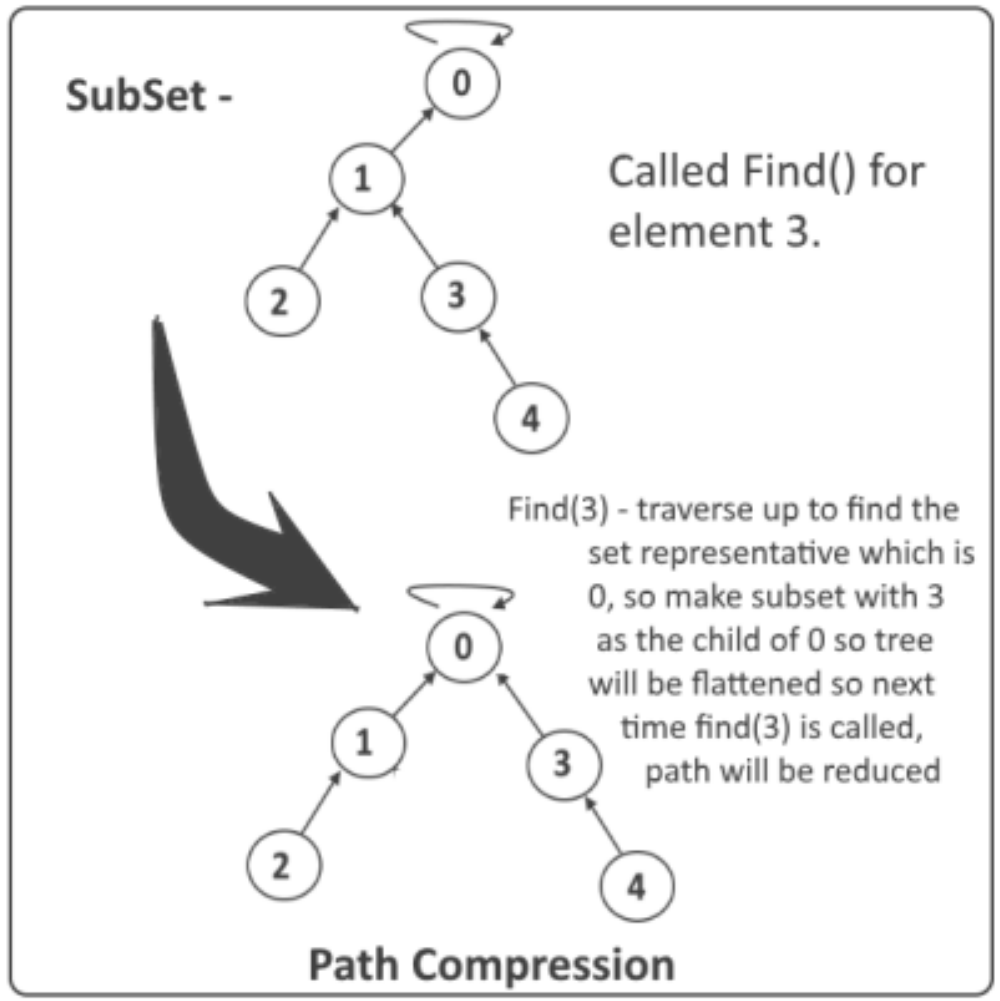
Concept of path compression (Second optimization!)



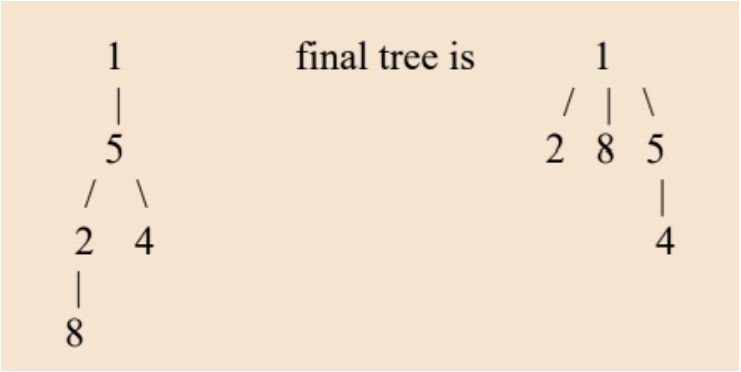
Path Compression

- Another enhancement we can add to disjoint sets is path compression.
- Every time we are forced to do a findset operation, we can directly connect each node on the path from the original node to the root.
- For example, for the set in the left side, if we need to do find(8), we will get the set in the right-side as during the find 8 process, we will have to access, 8 and 2 and both of them has the absolute root 1, we will make 1 as the parent for both 8 and 2




SubSet -
0
1
2
3
4
Called Find() for element 3.
Find(3) - traverse up to find the set representative which is 0, so make subset with 3 as the child of 0 so tree will be flattened so next time find(3) is called, path will be reduced
0
1
3
2
4
Path Compression

Path Compression



Path compression during find(8)

1	5	7	5	1	7	7	2
1	2	3	4	5	6	7	8

First, you find the root of this tree which is 1. Then you go through the path again, starting at 8, changing the parent of each of the nodes on that path to 1.

1	5	7	5	1	7	7	1
1	2	3	4	5	6	7	8

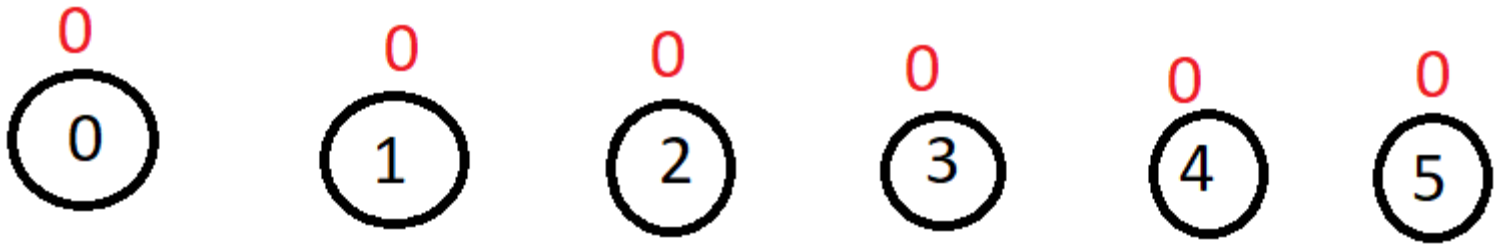
then, you take the 2 that was previously stored in index 8, and then change the value in that index to 1:

1	1	7	5	1	7	7	1
1	2	3	4	5	6	7	8

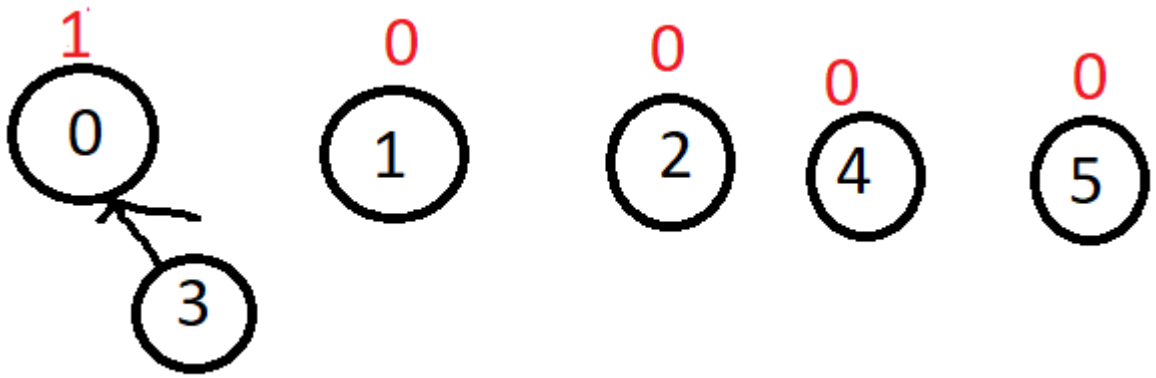
It has been shown through complicated analysis that the worst case running time of t operations is $O(t\alpha(t,n))$. Note that $\alpha(t,n) \leq 4$ for all $n \leq 10^{19728}$, so for all practical purposes on average, each operation takes constant time.

A complete example path compression

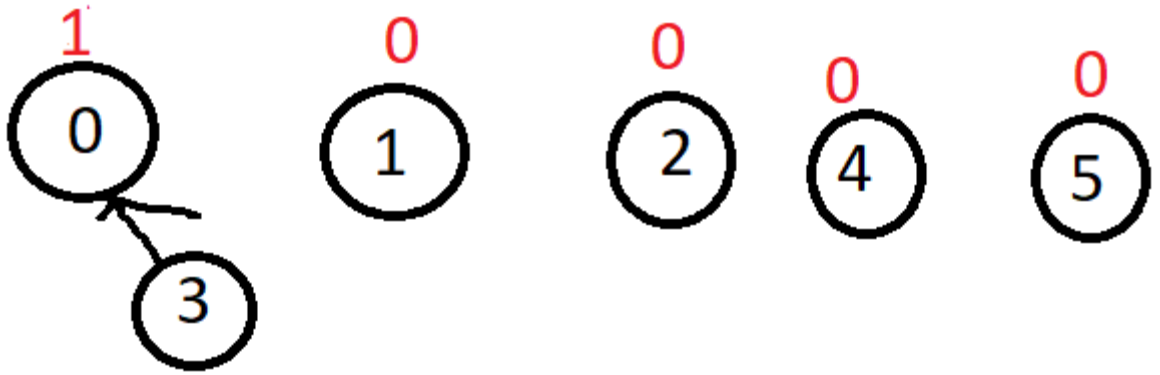
- N = 6



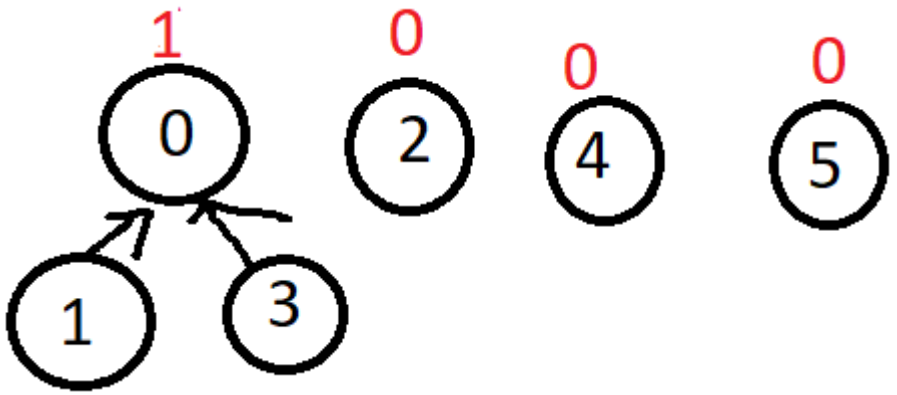
- Union(0, 3): No path compression needed during find operation



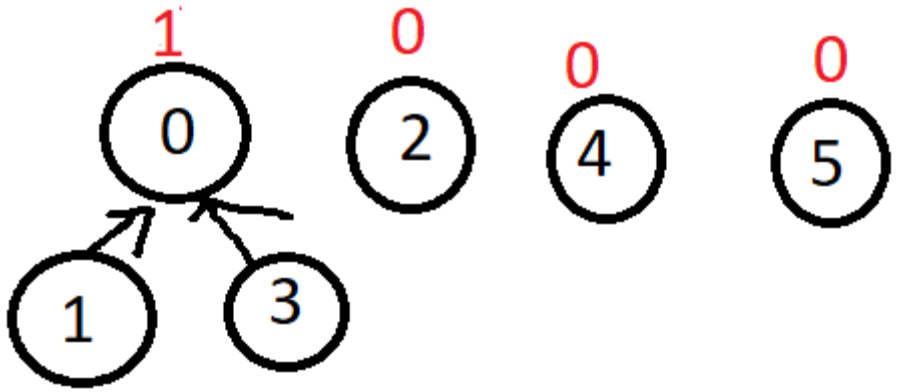
A complete example path compression



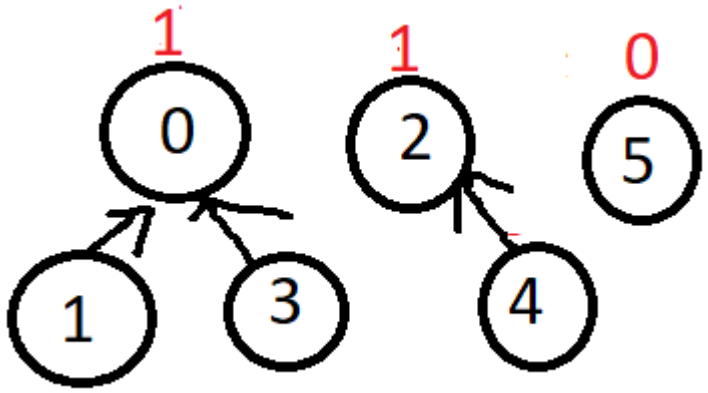
- Union(3, 1): No path compression needed during find operation



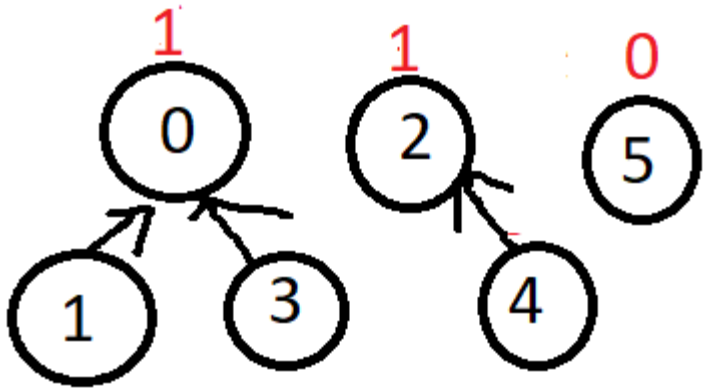
A complete example path compression



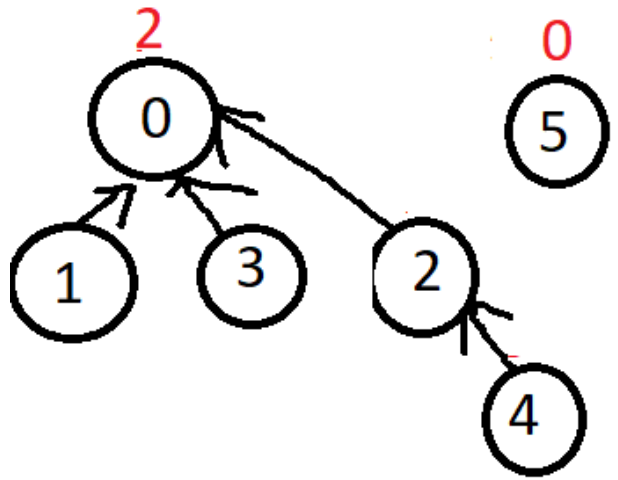
- Union(2, 4): No path compression needed during find operation



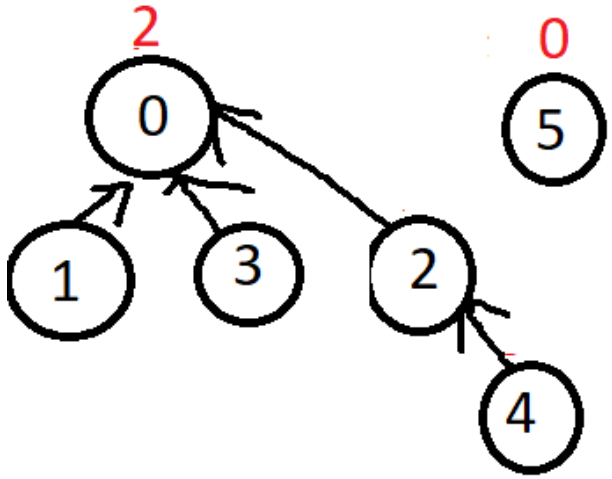
A complete example path compression



- Union(3, 4): No path compression needed during find operation

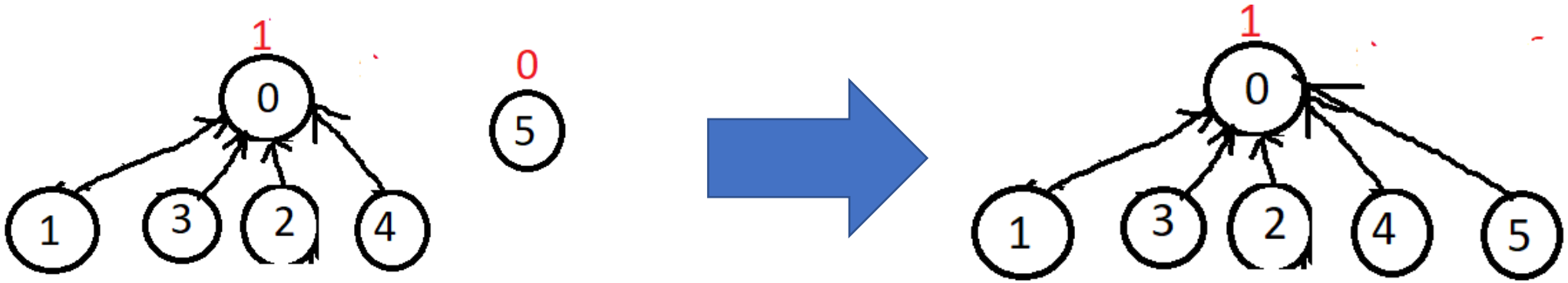


A complete example path compression



- Union(4, 5): Now while doing the find(4), there will be path compression as 4's absolute root is 0, 4's parent will become 0 instead of 2

Step 1: find(4) will update the set and return 0 and then it will perform union with find(5) = 5



How to implement path compression?

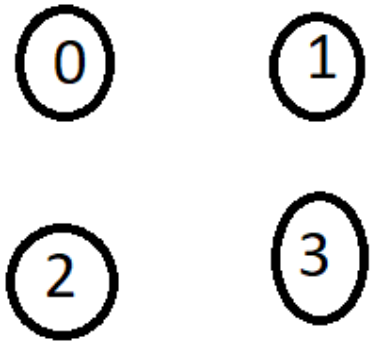
- You can just only keep the path compression optimization (without union by rank or height) and implement a code for this.
- Or, you can also combine the union by height or rank with the path compression.
- We simply update the find function so that every time your code process the find function, it will update the parent of the nodes it is visiting and make the absolute root as the parent to all of the visited nodes.
- If we need to change any parent, then we decrease the height (if the implementation also include the height optimization)

```
  // Returns the root node of the tree storing id.
  public int find(int id) {
// I am the root of the tree)
    if (id == parents[id].getID())
        return id;
    // Find my parent's root.
    int res = find(parents[id].getID());

    // if res is not my existing parent, make it parent
    if (res != parents[id].getID())
    {
        // Attach me directly to the root of my tree.
        parents[id].setID(res);

        parents[res].decHeight(); //decrease height as id is leveled up
    }
    return res;
  }
```

One common example using Disjont set is detecting cycle in an undirected graph

- Note that disjoint set can't be used to detect cycle in a directed graph
- Let us consider a graph with 4 nodes. You also have a list of edges

(0, 1)
(0, 3)
(2, 3)
(1, 2)



- Initially, we will have our disjoint sets, where each node is a set:

  {0}, {1}, {2}, {3}

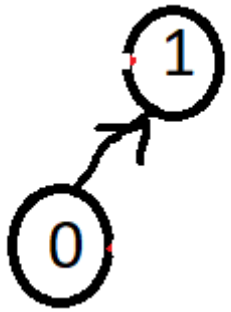
Our array will look like this

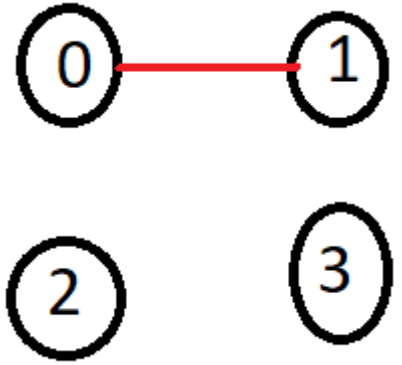
parent	**0**	**1**	**2**	**3**
Index/node	**0**	**1**	**2**	**3**

detecting cycle in a undirected graph

(0, 1)
(0, 3)
(2. 3)
(1, 2)

- Let us process the first edge (0, 1)
  - (0, 1): by using find(0) and find(1), we know that they are from different set. So, take union of them.
  - So, we have: {0, 1} {2} {3} . We are choosing 1 as the representative of the union {0, 1}



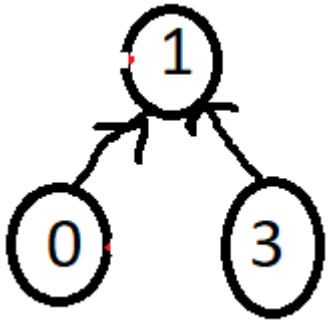


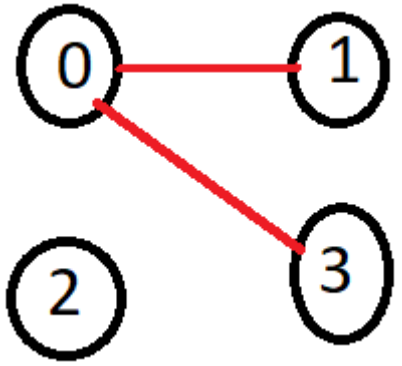
parent	1	1	2	3
Index/node	0	1	2	3

detecting cycle in a undirected graph

(0, 1)
(0, 3)
(2, 3)
(1, 2)

- Let us process the second edge (0, 3)
  - (0, 3): by using find(0) and find(3), we know that they are from different set. So, take union of them.
  - So, we have: {0, 1, 3} {2}. We are choosing 1 as the representative of the union {0, 1, 3}



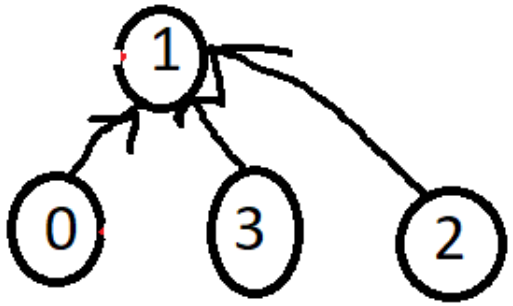


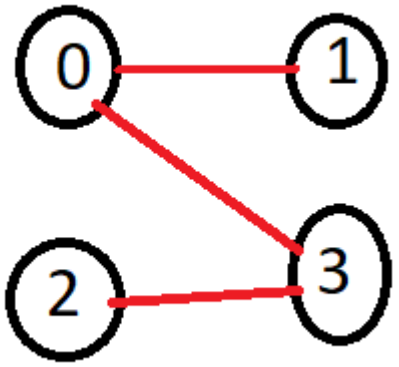
parent	**1**	**1**	**2**	**1**
Index/node	**0**	**1**	**2**	**3**

detecting cycle in a undirected graph

(0, 1)
(0, 3)
(2, 3)
(1, 2)

- Let us process the third edge (2, 3)
  - (2, 3): by using find(2) and find(3), we know that they are from different set. So, take union of them.
  - So, we have: {0, 1, 2, 3} . We are choosing 1 as the representative of the union {0, 1, 2, 3}





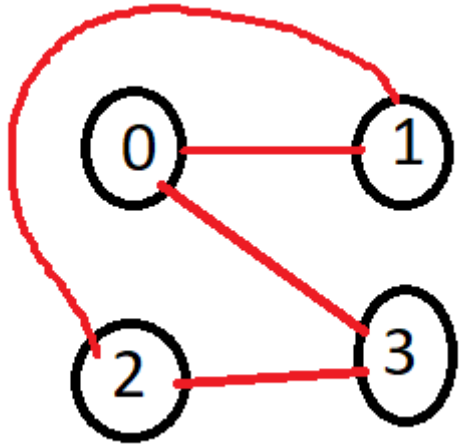
parent	**1**	**1**	**1**	**1**
Index/node	**0**	**1**	**2**	**3**

So far no cycle!
Let’s process more edges

detecting cycle in a undirected graph

- Let us process the fourth edge (1, 2)
  - (1, 2): by using find(1) and find(2) we get 1 for both cases.
  - It means they belong to the same set
  - So, if we process and add this edge to our graph, it is obviously creating a cycle!

(0, 1)
(0, 3)
(2, 3)
(1, 2)



How would you know that there is no cycle?

- If you process all edges and you could not get into a situation where both nodes in an edge belong to same set during that process!

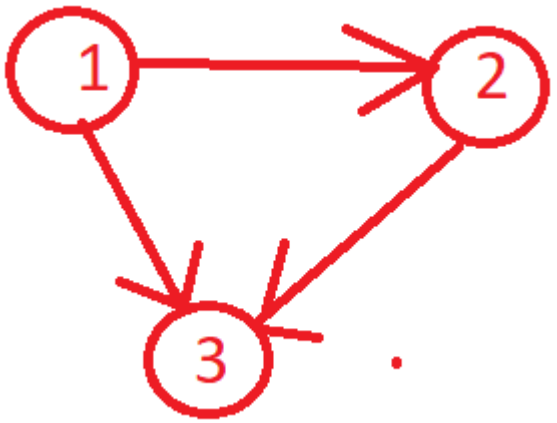
Time complexity of the cycle detection

- O(E * V)
- Where E is the total number of edges as we process all edges edge at worst case
- V is the number of vertices: for find and union worst case in a skewed set (This part also depends on the implementation of the disjoint set)

Consider writing this code!
It should not be difficult as you already have the disjoint set code.
-Just take input for the graph
- and process edge by edge
- and use the disjoint set functions in this process

Why can’t we detect cycle in a directed graph by disjoint set?

- Because set union does not follow any direction
- (A U B) does not say A to B or B to A.
- We cannot make a directional decision from set operation
- The following graph is a directed graph, but no cycle
- But, if you use disjoint set, it will detect a cycle



More example problems

You will find some more example problems on disjoint set in Arup's notes: http://www.cs.ucf.edu/~dmarino/ucf/cop3503/lectures/DisjointSets.pdf