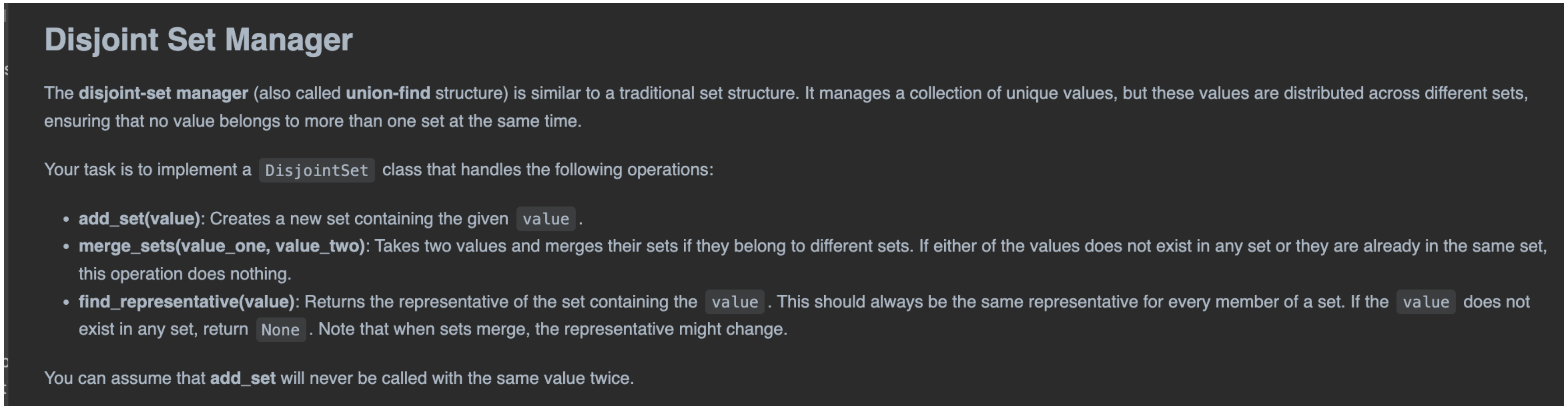
Union-Find Algorithm Explanation

Problem Definition: Disjoint Set



Disjoint Set Manager

The **disjoint-set manager** (also called **union-find** structure) is similar to a traditional set structure. It manages a collection of unique values, but these values are distributed across different sets, ensuring that no value belongs to more than one set at the same time.

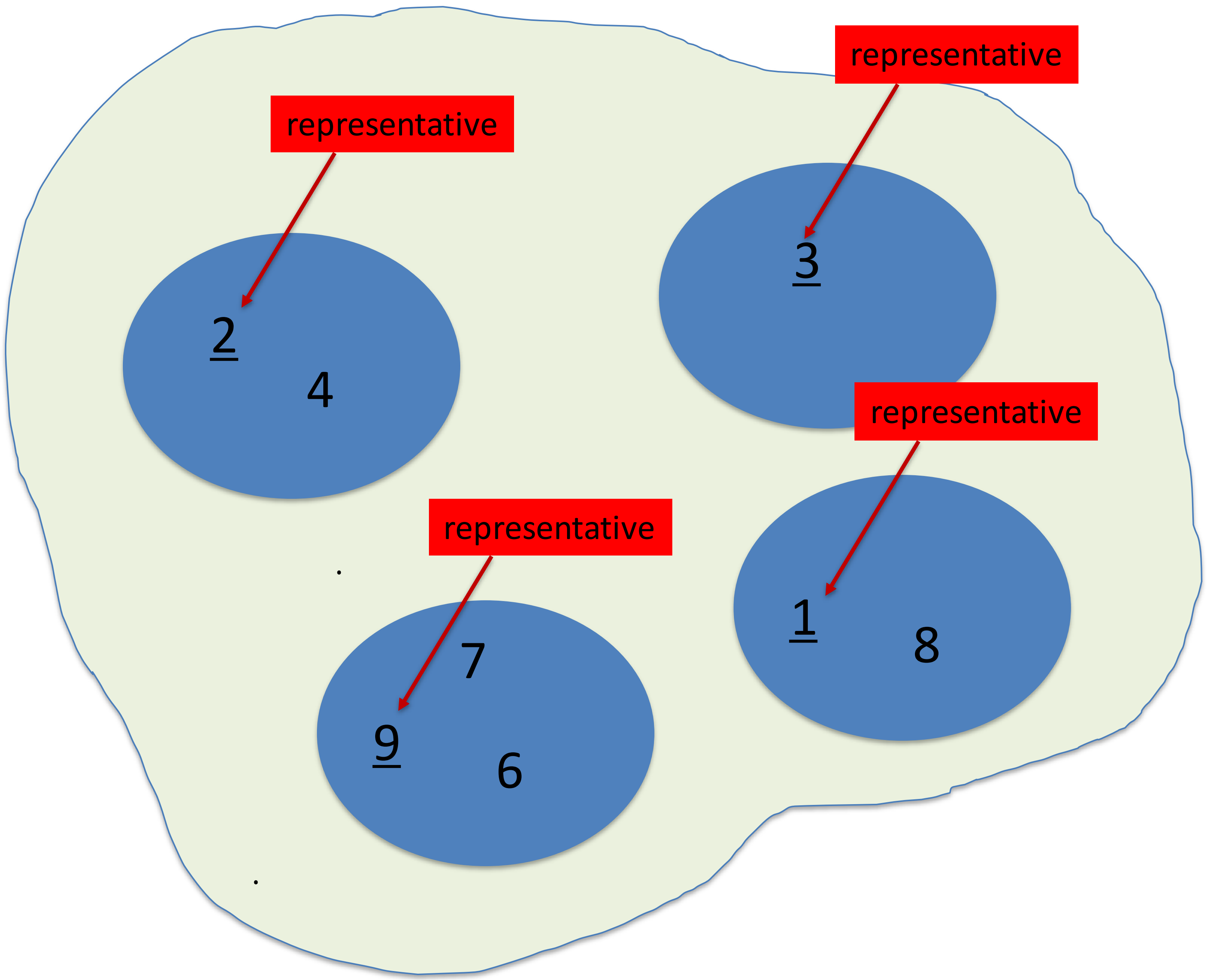
Your task is to implement a `DisjointSet` class that handles the following operations:

- **add_set(value)**: Creates a new set containing the given `value`.
- **merge_sets(value_one, value_two)**: Takes two values and merges their sets if they belong to different sets. If either of the values does not exist in any set or they are already in the same set, this operation does nothing.
- **find_representative(value)**: Returns the representative of the set containing the `value`. This should always be the same representative for every member of a set. If the `value` does not exist in any set, return `None`. Note that when sets merge, the representative might change.

You can assume that **add_set** will never be called with the same value twice.

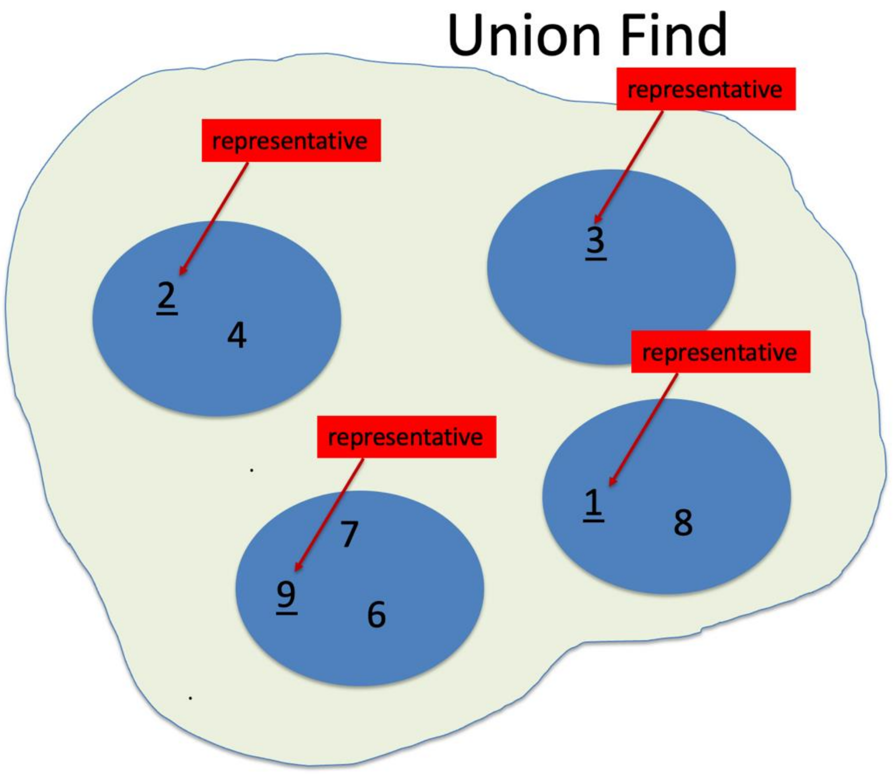
Disjoint sets: contains unique values these values are distributed amongst a variety of distinct disjoint sets, meaning that
no set can have duplicate values,
and no two sets can contain the same value

Union Find

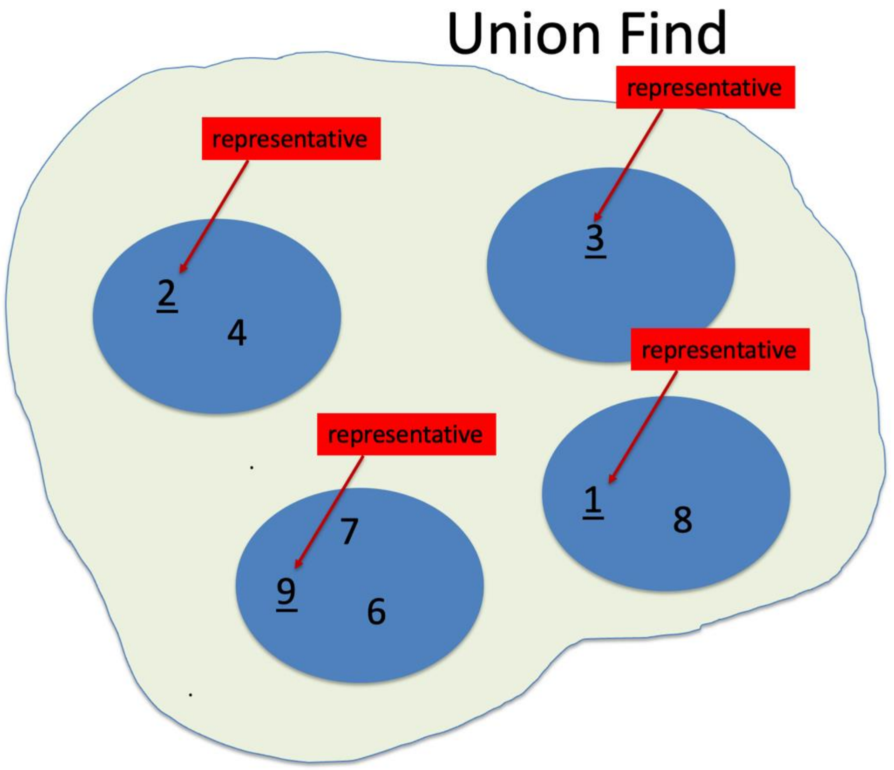

representative
representative
representative
representative
2
4
3
1
8
7
9
6

add_set

- It is to create one of these subsets.

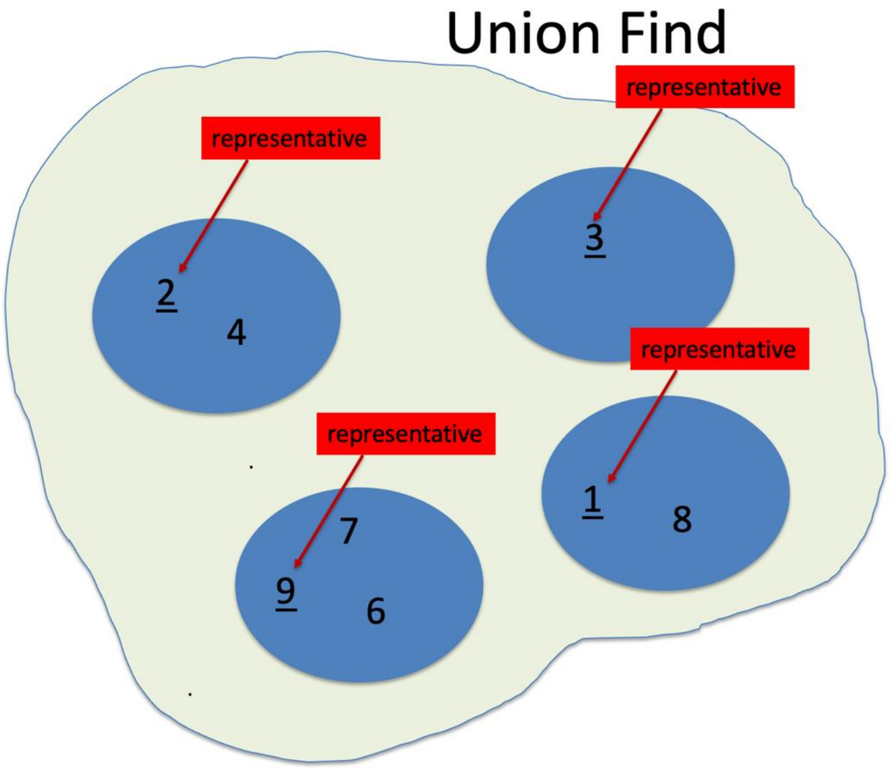


find_representative



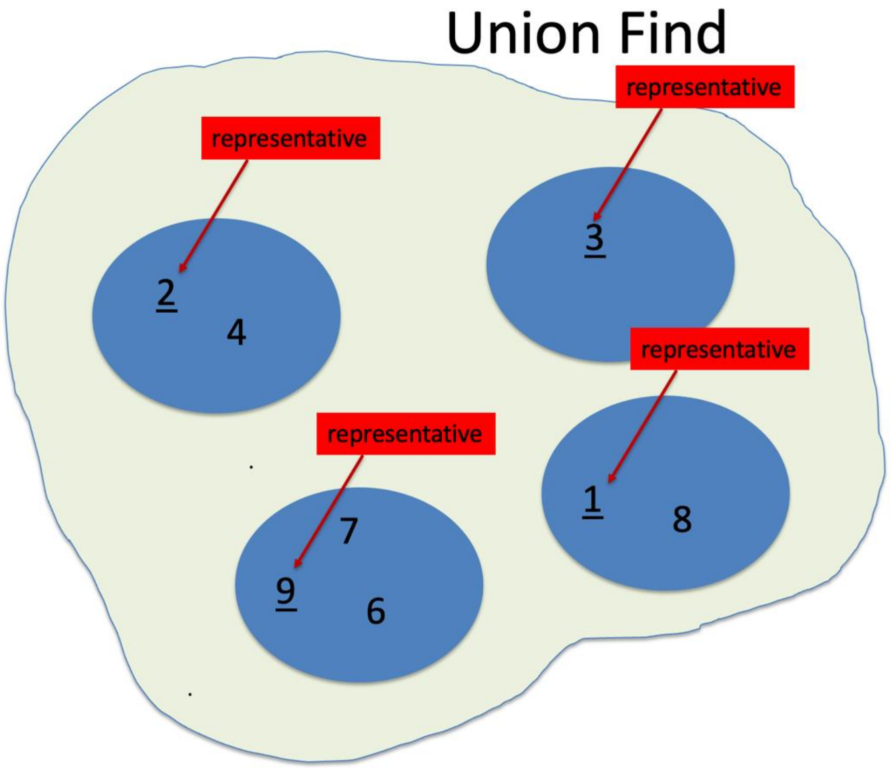
It takes a value and figures out which one of the subsets it is a part of ?
And it returns a representative of that subset
For example: if we find 3, that is part of a subset and 3 is its representative , so we would return 3.
find_rep(7) should return 9.
find_rep(8) should return 1.

find_representative



It takes a value and figures out which one of the subsets it is a part of ? And it returns a representative of that subset
For example: if we find 8, that is part of a subset and 1 is its representative , so we would return 1.

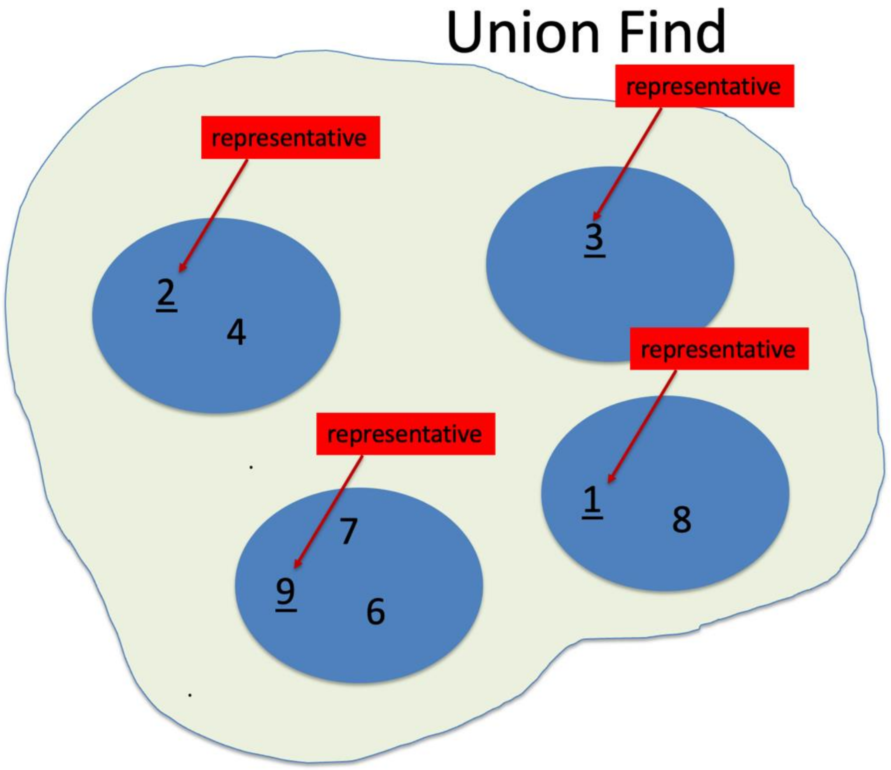
find_representative

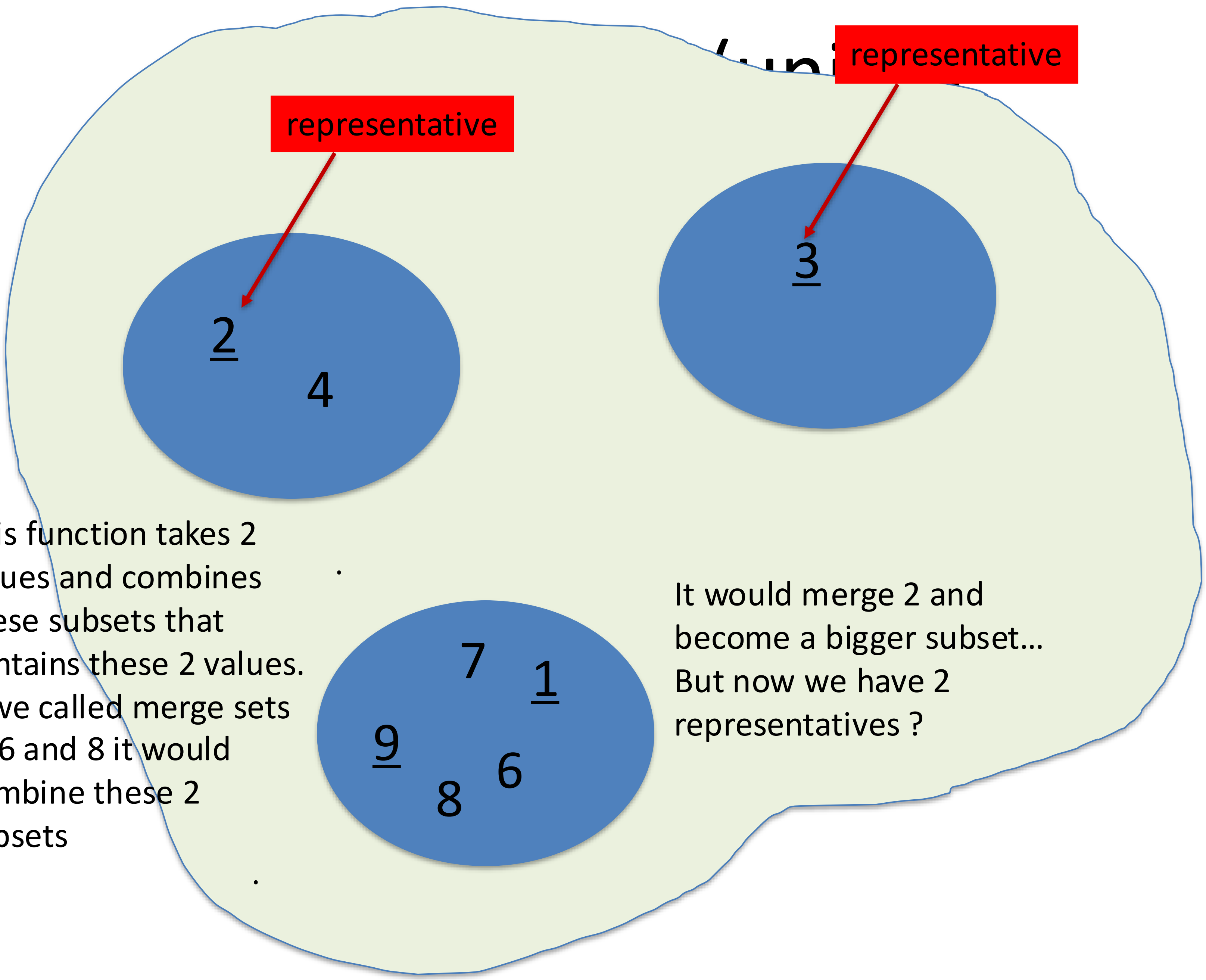


It takes a value and figures out which one of the subsets it is a part of ? And it returns a representative of that subset
For example: if we find 1, that is part of a subset and 1 is its representative , so we would return 1.

merge_sets(union)

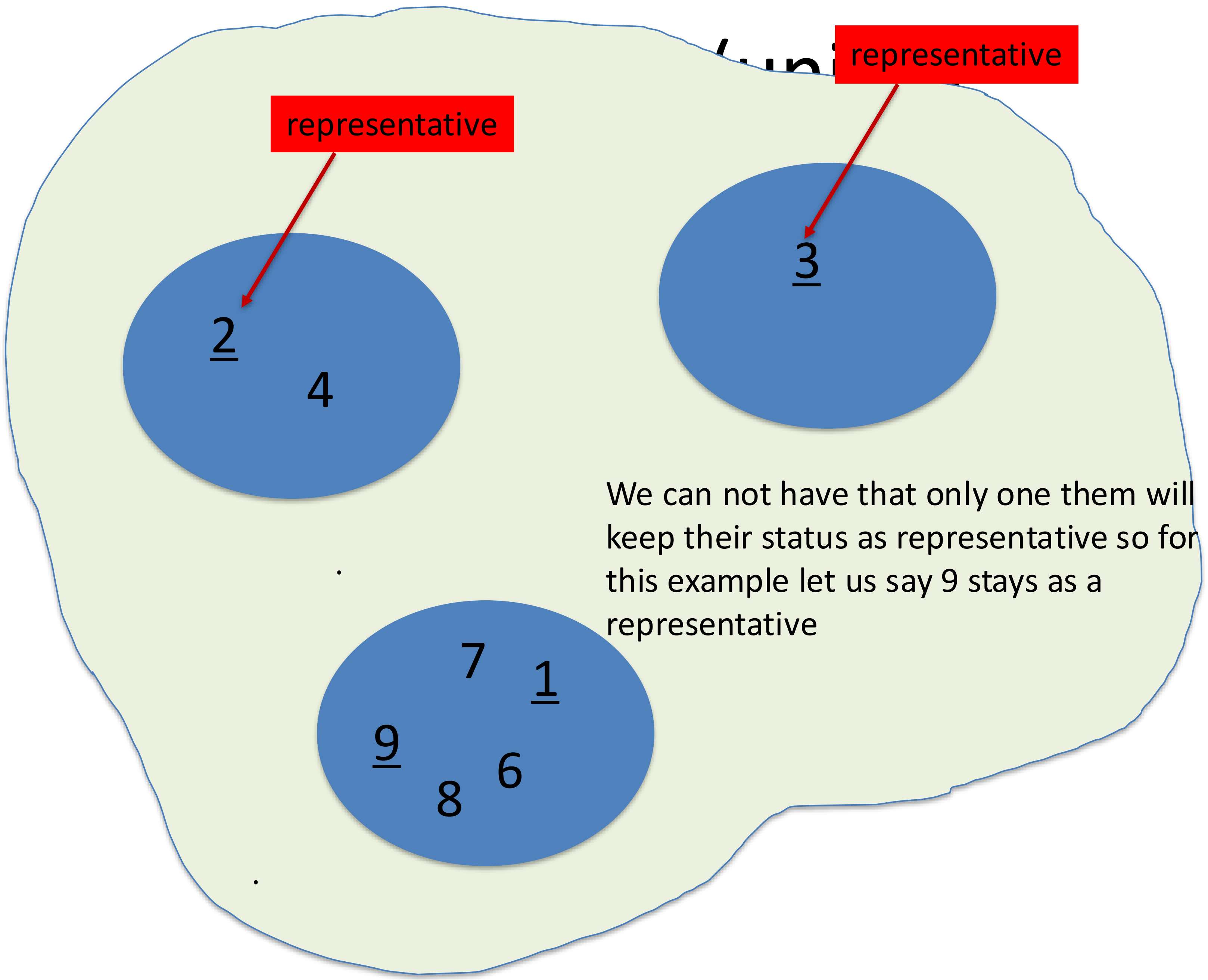
This function takes 2 values and combines these subsets that contains these 2 values.
If we called merge sets of 6 and 8 it would combine these 2 subsets



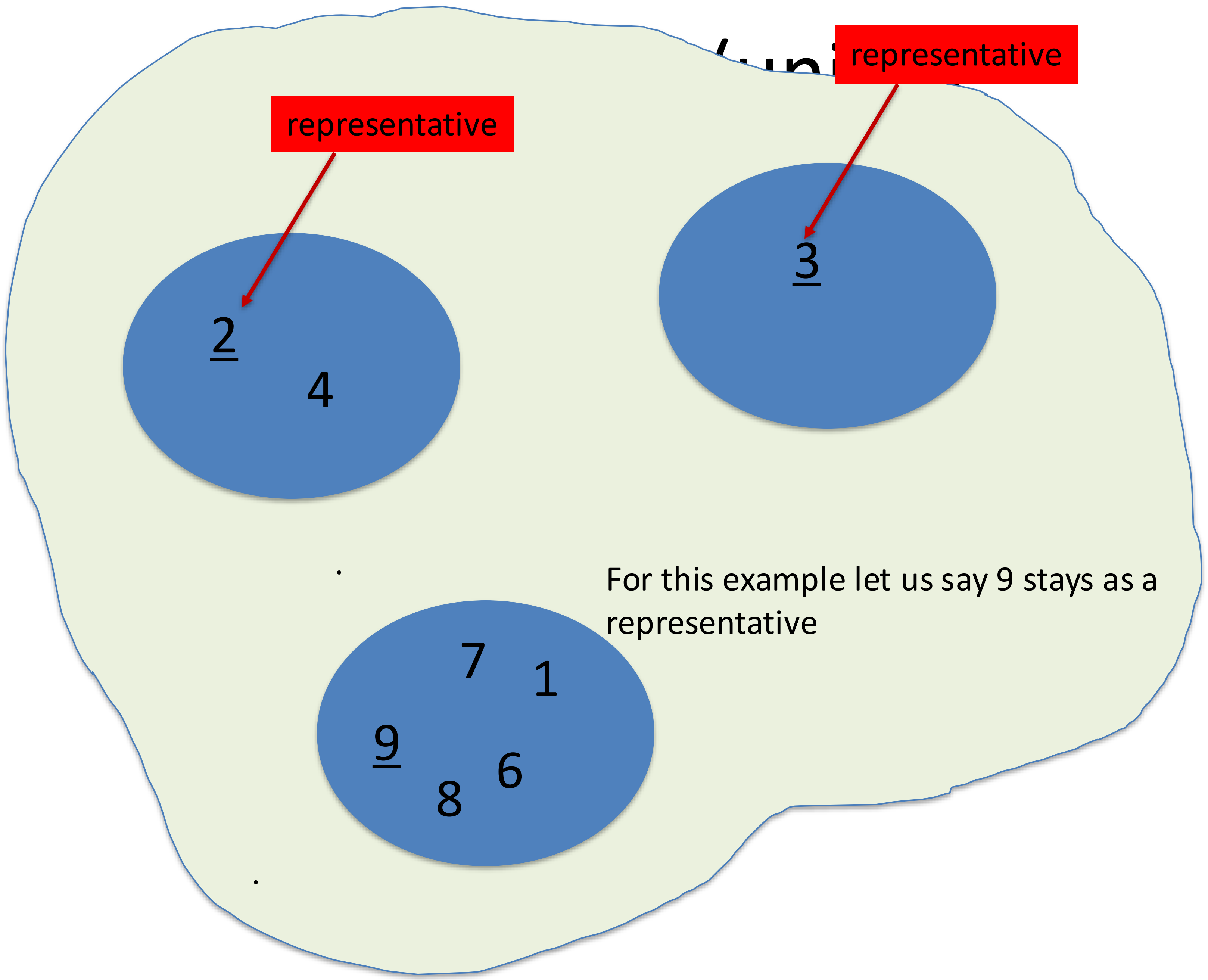


This function takes 2 values and combines these subsets that contains these 2 values. If we called merge sets of 6 and 8 it would combine these 2 subsets

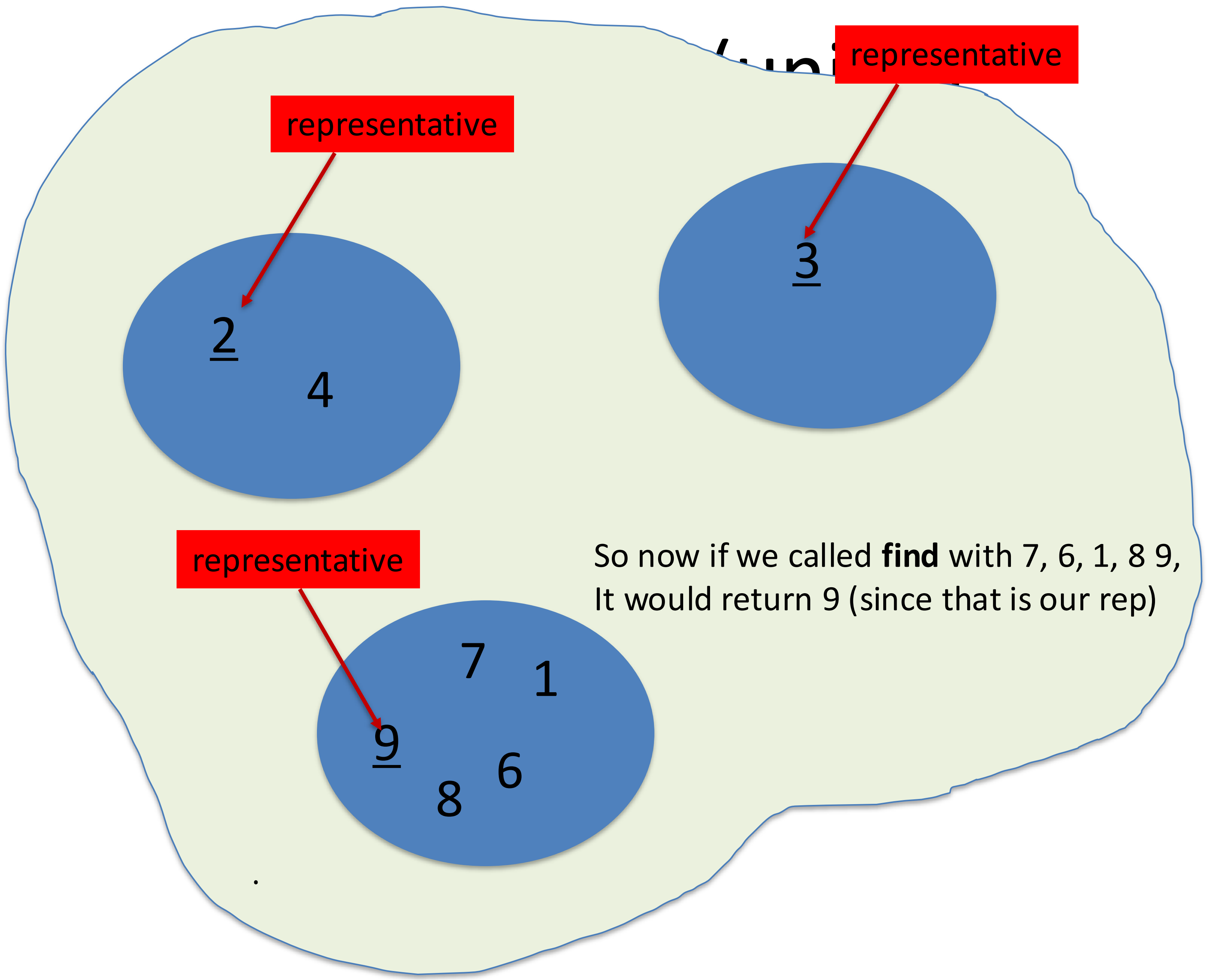
It would merge 2 and become a bigger subset… But now we have 2 representatives ?



We can not have that only one them will keep their status as representative so for this example let us say 9 stays as a representative



For this example let us say 9 stays as a representative



So now if we called **find** with 7, 6, 1, 8 9,
It would return 9 (since that is our rep)

How to implement the merge(union) function?

- We traditionally solve this like a tree data structure.
- We would have 3 trees
- And the root of each trees is going to be be the **representative**
- All the other nodes

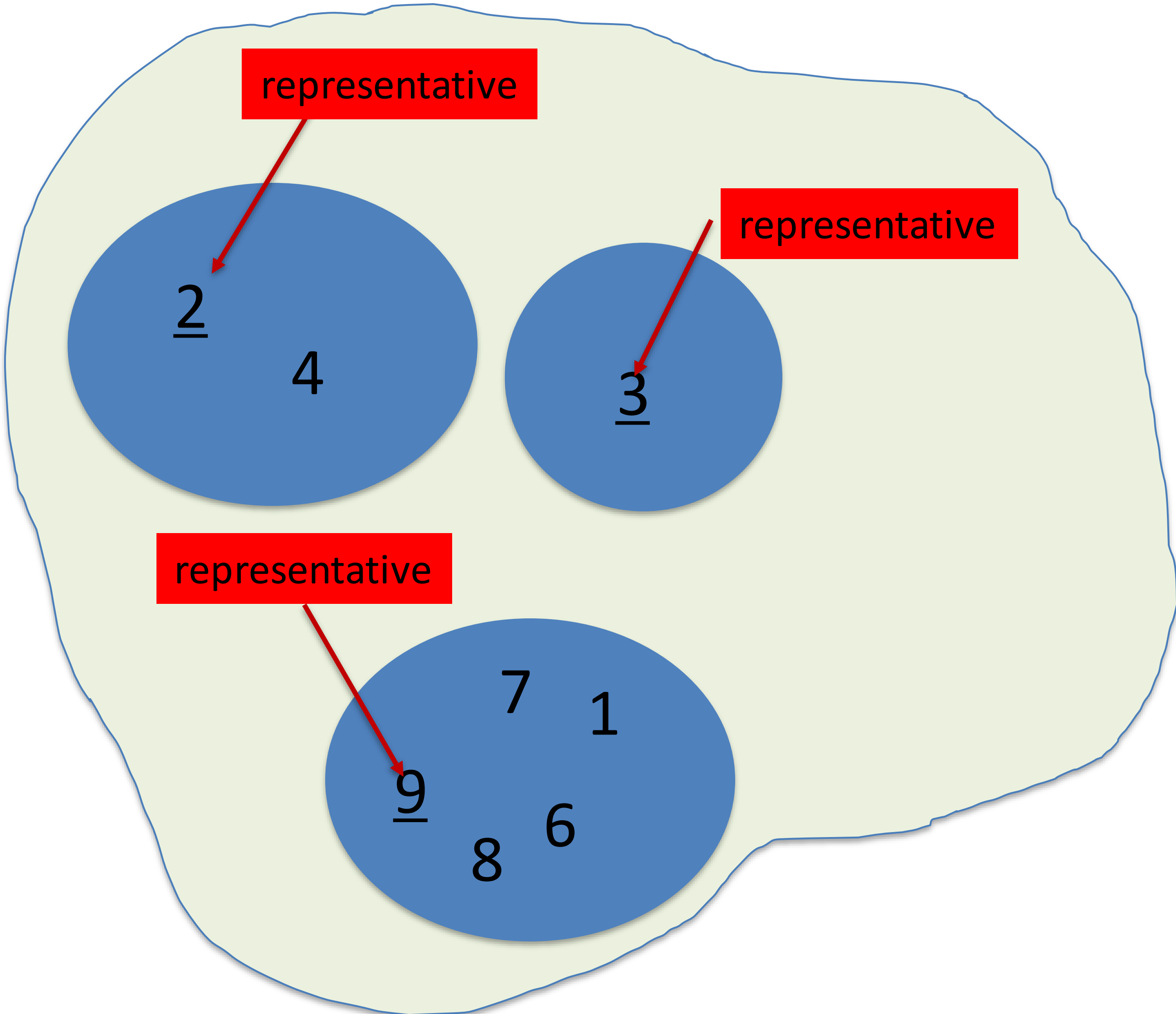
Will be underneath of this root node and we will also a have a reference to the parent.

So for the tree of set (2,4)

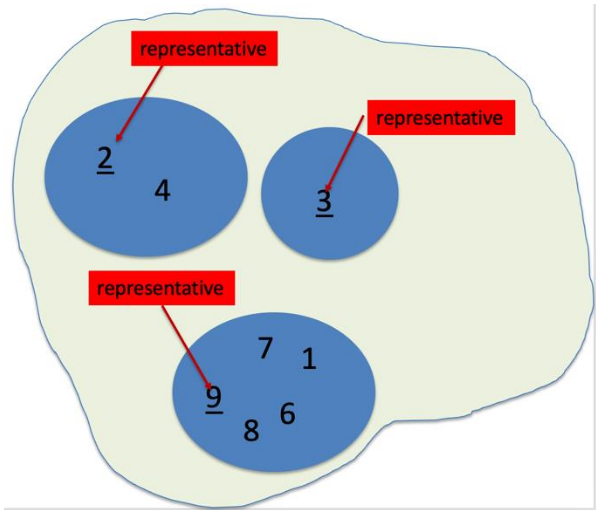
4 will be a node with a reference to its parent which is 2.

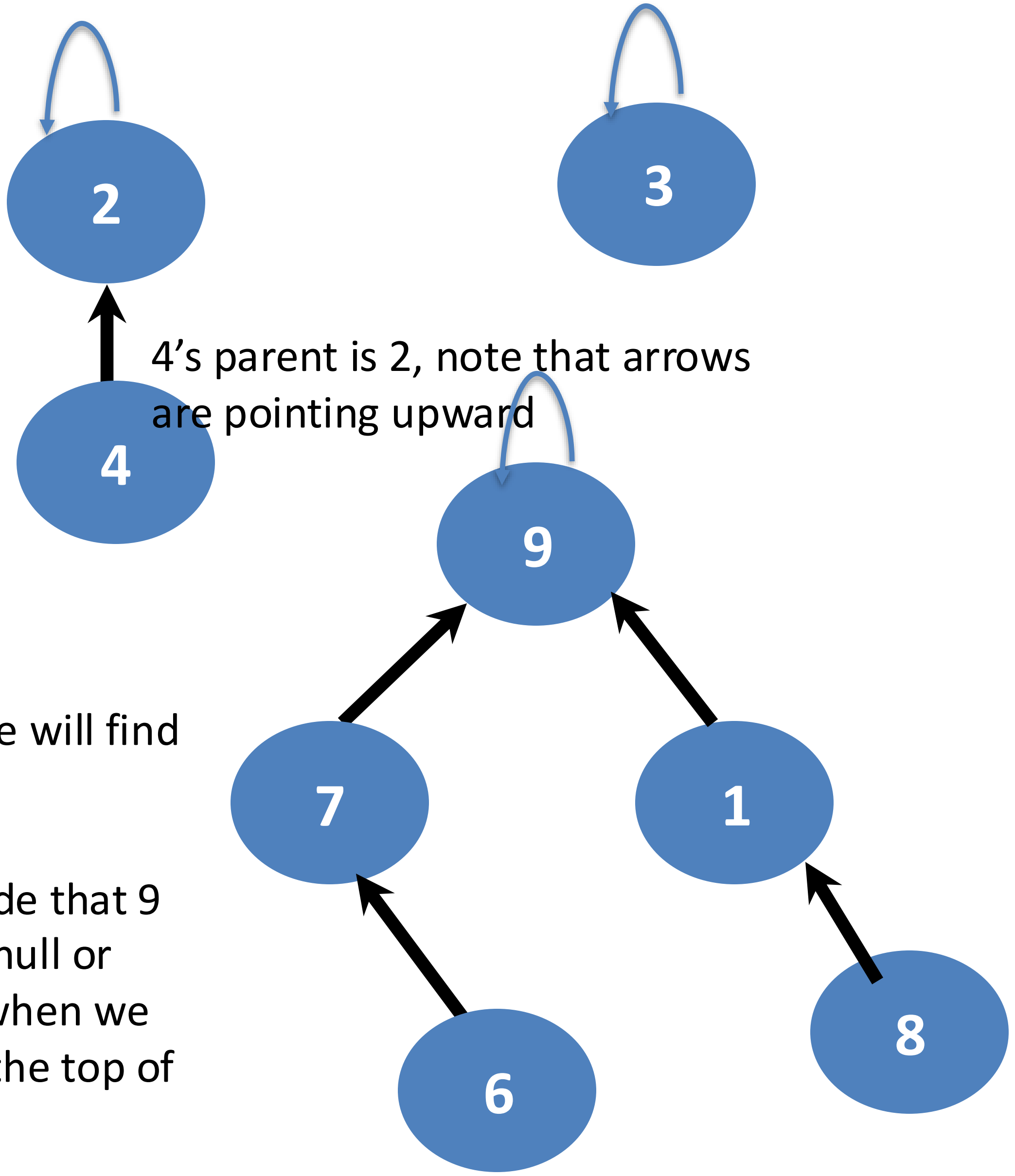
So when we invoke find(4) it will return its parent which is 2.

Another example: find(8) will return 9.



So let’s try to create these trees





4’s parent is 2, note that arrows are pointing upward

We build a binary tree.
For example if we are looking for 6, we will find it is parent 7 and
7 is not the root node we will find 9.
 can represent he root node in our code that 9 is the root by checking it its parent is null or just add a self pointer, we can check when we get to the self pointer we must be at the top of the tree.

FIRST SOLUTION ATTEMPT AND ITS RUN TIME ANALYSIS

Let’s look at the function calls

```
add_set(3)
add_set(6)
find_rep(6)
merge_sets(3,6)
find_rep(6)
find_rep(3)
add_set(2)
merge_sets(2,6)
```

Note that we do not need an actual tree with nodes instead all we have to do is keep track of each value what its parent is and then we have all the information to go up to the tree.

Let’s look at the function calls

```
           value:parent
parents : {              }
```

add_set(3)
add_set(6)
find_rep(6)
merge_sets(3,6)
find_rep(6)
find_rep(3)
add_set(2)
merge_sets(2,6)

We need to create a **value:parent** dictionary that stores the parents

Let’s look at the function calls

parents : { value:parent }

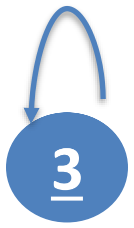
add_set(3)
add_set(6)
find_rep(6)
merge_sets(3,6)
find_rep(6)
find_rep(3)
add_set(2)
merge_sets(2,6)

We need to create a value that stores the parents

Let’s look at the function calls

value:parent

parents : { 3:3 }

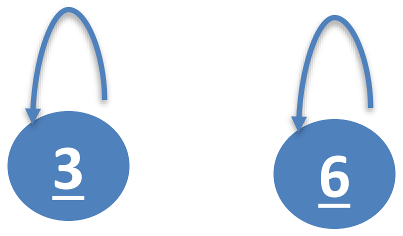


add_set(3)
add_set(6)
find_rep(6)
merge_sets(3,6)
find_rep(6)
find_rep(3)
add_set(2)
merge_sets(2,6)

We create node 3 , value is 3, parent is 3, so we add its self pointer

Let’s look at the function calls

value:parent
parents : { 3:3 , 6:6 }

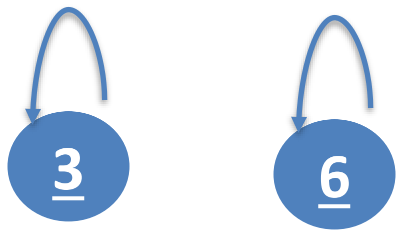


add_set(3)
add_set(6)
find_rep(6)
merge_sets(3,6)
find_rep(6)
find_rep(3)
add_set(2)
merge_sets(2,6)

We create node 6 , value is 6, parent is 6, so we add its self pointer

Let's look at the function calls

value:parent
parents : { 3:3 , 6:6 }

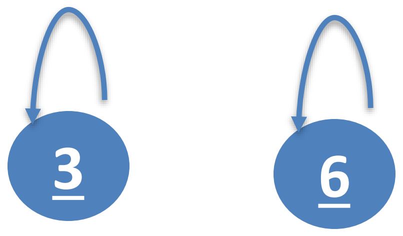


add_set(3)
add_set(6)
find_rep(6)
merge_sets(3,6)
find_rep(6)
find_rep(3)
add_set(2)
merge_sets(2,6)

Next we call find_rep(6) 6 is the root node its parent is 6, we return 6

Let’s look at the function calls

parents : { 3:3 , 6:6 }

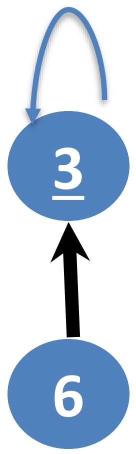


add_set(3)
add_set(6)
find_rep(6)
merge_sets(3,6)
find_rep(6)
find_rep(3)
add_set(2)
merge_sets(2,6)

Next we merge 3 and 6 . We want 3 and 6 to be in same sets.
Which to pick as rep?
Either is fine. Let’s pick 3.

Let’s look at the function calls

value:parent
parents : { 3:3 , 6:3 }

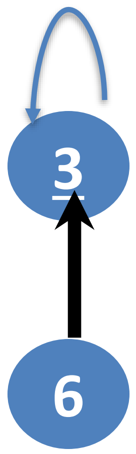


add_set(3)
add_set(6)
find_rep(6)
merge_sets(3,6)
find_rep(6)
find_rep(3)
add_set(2)
merge_sets(2,6)

Next we merge 3 and 6 . We want 3 and 6 to be in same sets. Which to pick as rep? Either is fine. Let’s pick 3. Value 6 now has a parent 3.

Let’s look at the function calls

value:parent
parents : { 3:3 , 6:3 }



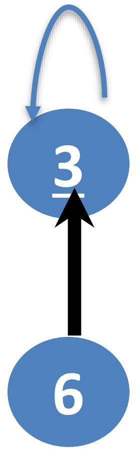
add_set(3)
add_set(6)
find_rep(6)
merge_sets(3,6)
find_rep(6)
find_rep(3)
add_set(2)
merge_sets(2,6)

find_rep(6) should return 3.
How?
We start at 6, we look at its parent and go there, its parent is 3, and 3 is the root node, we return 3

Let’s look at the function calls

value:parent

parents : { 3:3 , 6:3 }

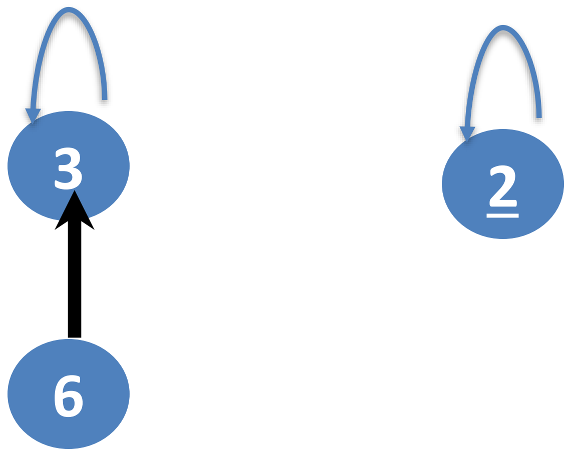


add_set(3)
add_set(6)
find_rep(6)
merge_sets(3,6)
find_rep(6)
find_rep(3)
add_set(2)
merge_sets(2,6)

find_rep(3) should return 3.

Let's look at the function calls

value:parent
parents : { 3:3 , 6:3 , 2:2 }

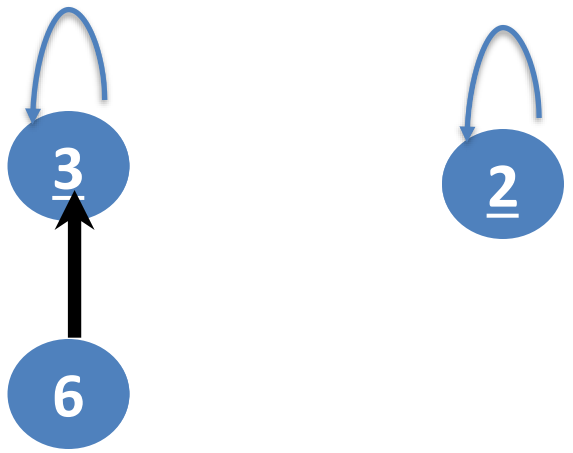


add_set(3)
add_set(6)
find_rep(6)
merge_sets(3,6)
find_rep(6)
find_rep(3)
add_set(2)
merge_sets(2,6)

Add set 2

Let’s look at the function calls

value:parent
parents : { 3:3 , 6:3 , 2:2 }

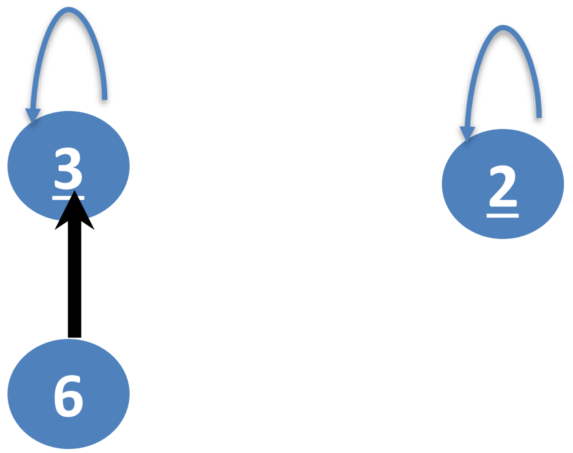


add_set(3)
add_set(6)
find_rep(6)
merge_sets(3,6)
find_rep(6)
find_rep(3)
add_set(2)
merge_sets(2,6)

merge_sets(2,6):we have multiple options, we can add the tree 3 to tree 2 as its child, or we can add the tree 2 to tree 3 as its child, we can add 2 to be the child of 6.

Let’s look at the function calls

value:parent
parents : { 3:3 , 6:3 , 2:2 }

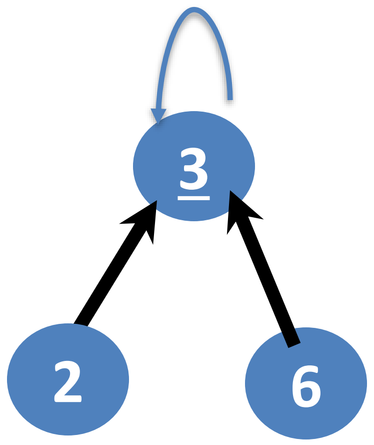


add_set(3)
add_set(6)
find_rep(6)
merge_sets(3,6)
find_rep(6)
find_rep(3)
add_set(2)
merge_sets(2,6)

merge_sets(2,6): we do not care too much so for the sake of the argument let’s take 2 and make it a child of 3

Let’s look at the function calls

value:parent
parents : { 3:3 , 6:3 , 2:3 }



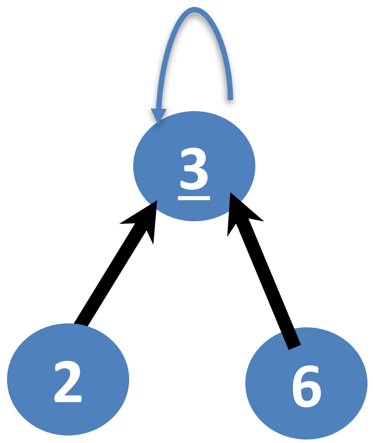
add_set(3)
add_set(6)
find_rep(6)
merge_sets(3,6)
find_rep(6)
find_rep(3)
add_set(2)
merge_sets(2,6)

merge_sets(2,6): we do not care too much so for the sake of the argument let’s take 2 and make it a child of 3

Let’s look at the function calls

As far as the time complexity, let’s talk about how it works:

value:parent
parents : { 3:3 , 6:3 , 2:3 }



add_set : it does not do too much except adding to parents,
runtime:O(1),
Aux space:O(1)

find_rep :
runtime:? , aux space:?

merge_sets :

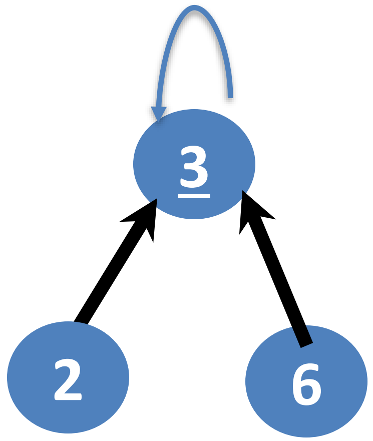
runtime:?, space: ?

add_set(3)
add_set(6)
find_rep(6)
merge_sets(3,6)
find_rep(6)
find_rep(3)
add_set(2)
merge_sets(2,6)

Let's look at the function calls

As far as the time complexity, let's talk about how it works:

value:parent
parents : { 3:3 , 6:3 , 2:3 }



add_set : runtime:O(1), aux space:O(1)

find_rep : has to go through the tree, looks for the tree, so the big O is whatever the height of the tree is.
We need to figure out the worst case for the height of the tree.
Remember when we said we really do not care how we position them, well that might be an issue where we can get a degenerate tree.
runtime:O(n), aux space:O(1)

merge_sets :

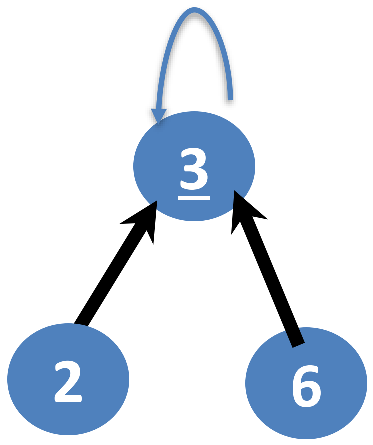
runtime:?, aux space: ?

add_set(3)
add_set(6)
find_rep(6)
merge_sets(3,6)
find_rep(6)
find_rep(3)
add_set(2)
merge_sets(2,6)

Let's look at the function calls

As far as the time complexity, let's talk about how it works:

value:parent
parents : { 3:3 , 6:3 , 2:3 }



add_set : runtime:O(1), aux space:O(1) , adds most one value at time

find_rep : Worst case runtime:O(n), aux space:O(1)

merge_sets : At first this function's run time appears to be constant, but not really, we have few things to do in merge_sets.
Which subsets are they in?
so that means merge_sets is going to call **find_rep.**
For example: merge_sets(2,6) must call find_rep(2), find_rep(6)
That means union takes the same amount time as find_rep

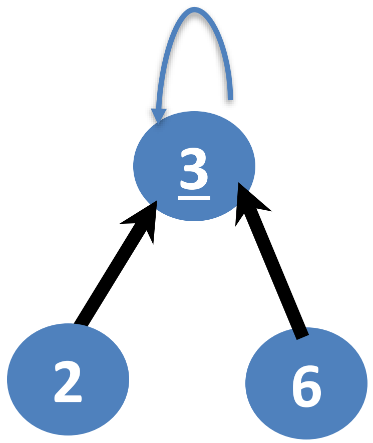
runtime: O(n), aux space: O(1)

add_set(3)
add_set(6)
find_rep(6)
merge_sets(3,6)
find_rep(6)
find_rep(3)
add_set(2)
merge_sets(2,6)

Let's look at the function calls

As far as the time complexity, let's talk about how it works:

value:parent
parents : { 3:3 , 6:3 , 2:3 }



add_set : runtime:O(1), aux space:O(1) , adds most one value at time

find_rep : Worst case runtime:O(n), aux space:O(1)

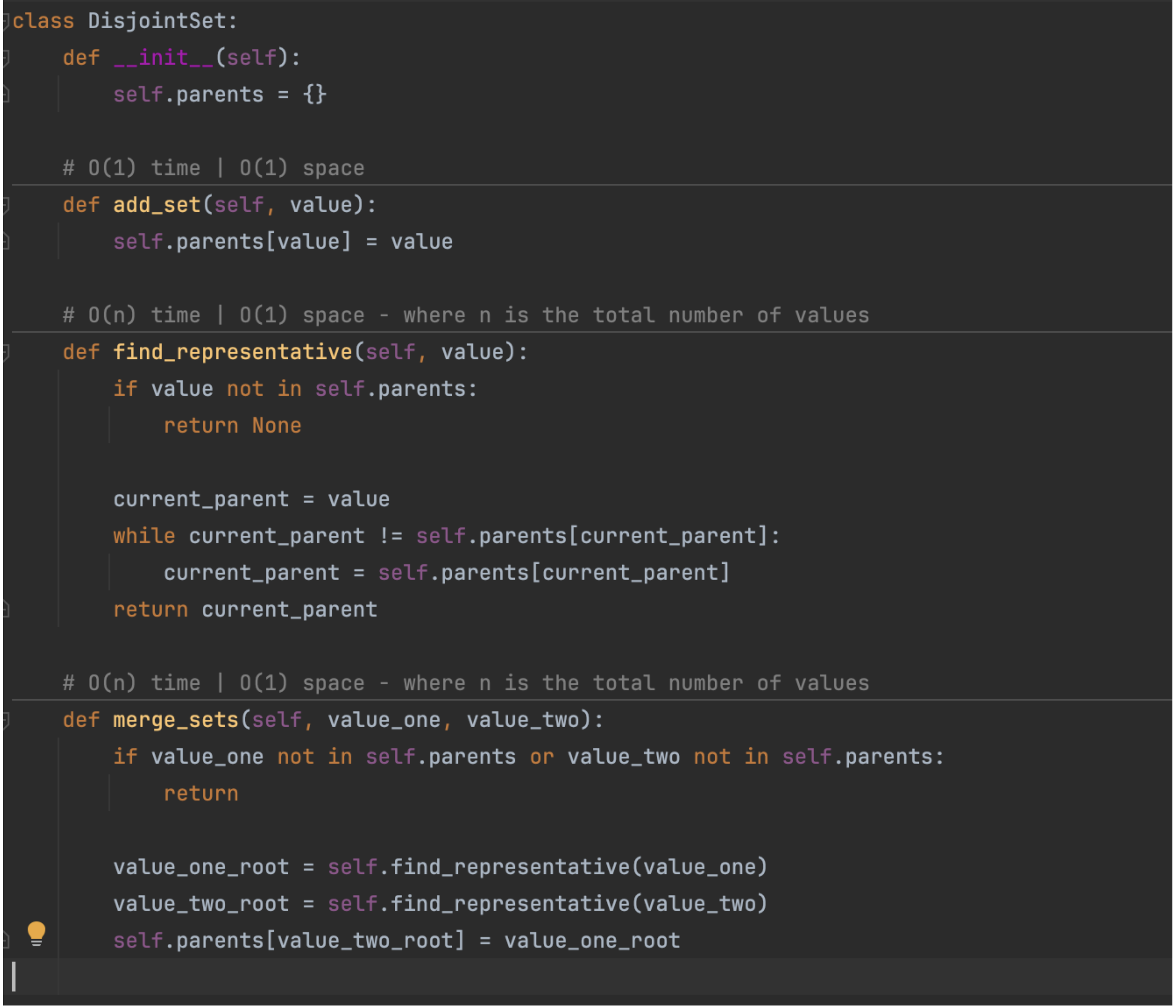
add_set(3)
add_set(6)
find_rep(6)
merge_sets(3,6)
find_rep(6)
find_rep(3)
add_set(2)
merge_sets(2,6)

merge_sets : At first this function's run time appears to be constant, but not really, we have few things to do in merge_sets.
Which subsets are they in?so that means merge_sets is going to call find_rep.
For example: merge_sets(2,6) must call find_rep(2), find_rep(6)
That means union takes the same amount time as find_rep

runtime: O(n), aux space: O(1)

Keep in mind that the entire class takes O(n) space for the parents dictionary for storing the value:parent but none of the functions individually use aux space,

IMPLEMENTING FIRST SOLUTION



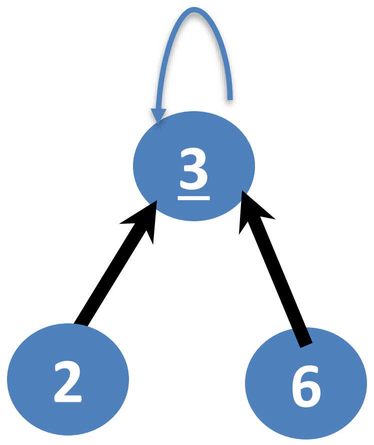
```python
class DisjointSet:
    def __init__(self):
        self.parents = {}

    # O(1) time | O(1) space
    def add_set(self, value):
        self.parents[value] = value

    # O(n) time | O(1) space - where n is the total number of values
    def find_representative(self, value):
        if value not in self.parents:
            return None

        current_parent = value
        while current_parent != self.parents[current_parent]:
            current_parent = self.parents[current_parent]
        return current_parent

    # O(n) time | O(1) space - where n is the total number of values
    def merge_sets(self, value_one, value_two):
        if value_one not in self.parents or value_two not in self.parents:
            return

        value_one_root = self.find_representative(value_one)
        value_two_root = self.find_representative(value_two)
        self.parents[value_two_root] = value_one_root
```

SECOND SOLUTION ATTEMPT

What can we do to optimize this solution a little better ?

Second solution

value:parent
parents : { 3:3 , 6:3 , 2:3 }



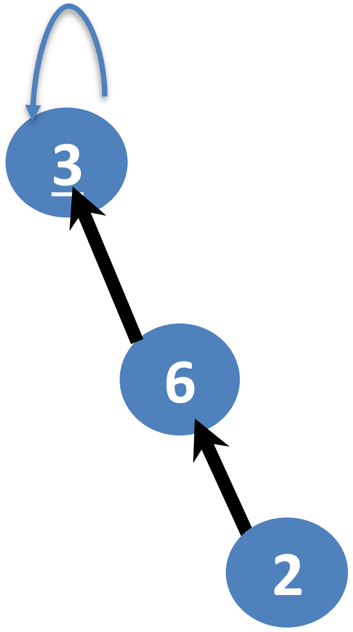
add_set(3)
add_set(6)
find_rep(6)
merge_sets(3,6)
find_rep(6)
find_rep(3)
add_set(2)
merge_sets(2,6)

We want our structure to use the tree but we want to **minimize the height** of our tree to avoid O(n) run time.
We want our nodes to be as close to the parent as possible.

Second solution

value:parent

parents : { 3:3 , 6:3 , 2:3 }



add_set(3)

add_set(6)

find_rep(6)

merge_sets(3,6)

find_rep(6)

find_rep(3)

add_set(2)

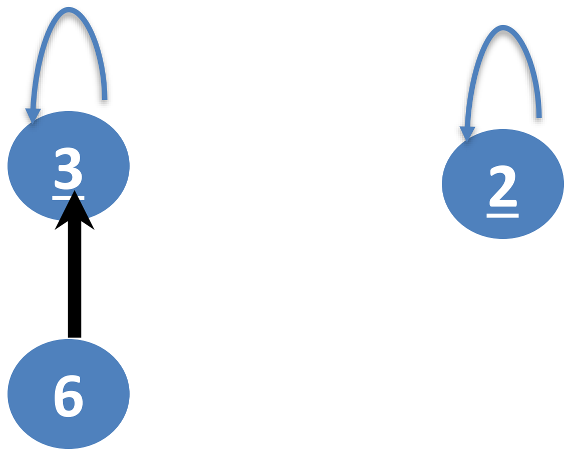
merge_sets(2,6)

We want our structure to use the tree but we want to minimize the height of our tree to avoi
O(n) run time.
We want our nodes to be as close to the parent as possible.
But with our first solution we saw that we could end up with a tree like above.
And this would take linear time to get through the height of the tree. NOT OPTIMAL !

What would be the most optimal way to merge these 2 sets

value:parent

parents : { 3:3 , 6:3 , 2:2 }

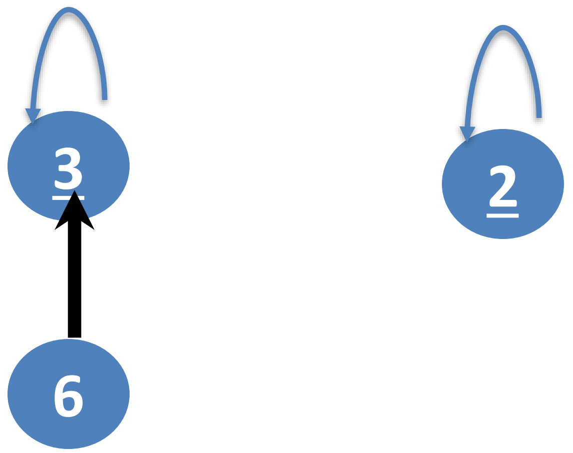


add_set(3)
add_set(6)
find_rep(6)
merge_sets(3,6)
find_rep(6)
find_rep(3)
add_set(2)
merge_sets(2,6)

What would be the most optimal way to merge these 2 sets

value:parent

parents : { 3:3 , 6:3 , 2:2 }



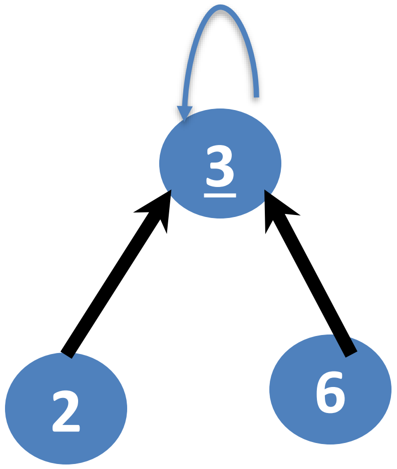
add_set(3)
add_set(6)
find_rep(6)
merge_sets(3,6)
find_rep(6)
find_rep(3)
add_set(2)
merge_sets(2,6)

We would like to avoid to have a tree with a height of 3, this could be where we decide that 2 becomes a child of 6 height=2, or 3 becomes a child of 2 again the height =2.

What would be the most optimal way to merge these 2 sets

value:parent

parents : { 3:3 , 6:3 , 2:2 }

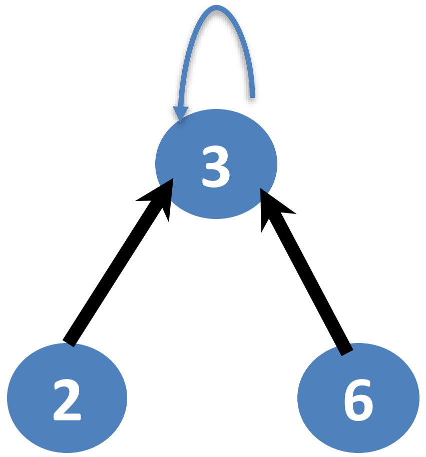


add_set(3)
add_set(6)
find_rep(6)
merge_sets(3,6)
find_rep(6)
find_rep(3)
add_set(2)
merge_sets(2,6)

We want node 2 to become a child of 3.

What would be the most optimal way to merge these 2 sets

value:parent
parents : { 3:3 , 6:3 , 2:2 }

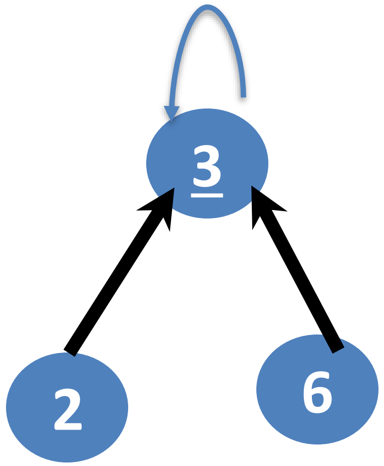


add_set(3)
add_set(6)
find_rep(6)
merge_sets(3,6)
find_rep(6)
find_rep(3)
add_set(2)
merge_sets(2,6)

We want node 2 to become a child of 3.
In other words:
We would like the **smaller of the trees to become a child of the larger of the trees root node.**

What would be the most optimal way to merge these 2 sets

value:parent
parents : { 3:3 , 6:3 , 2:2 }



add_set(3)
add_set(6)
find_rep(6)
merge_sets(3,6)
find_rep(6)
find_rep(3)
add_set(2)
merge_sets(2,6)

We want node 2 to become a child of 3.
In other words: We would like the smaller of the trees to become a child of the larger of the trees root node. So we would like 2 to be directly related to 3, meaning become the child of root node 3.

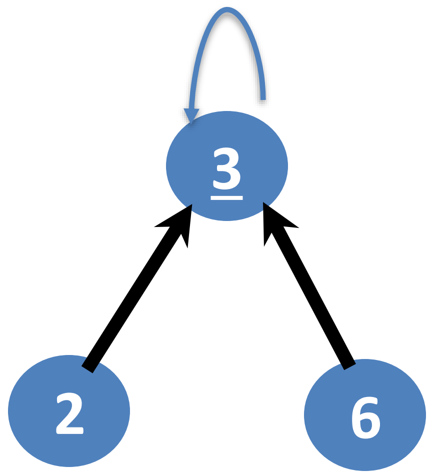
If node 2 is a child of 6, it would be farther away from the root node.

What would be the most optimal way to merge these 2 sets

value:parent

parents : { 3:3 , 6:3 , 2:2 }

```
add_set(3)
add_set(6)
find_rep(6)
merge_sets(3,6)
find_rep(6)
find_rep(3)
add_set(2)
merge_sets(2,6)
```



How do we make sure the smaller tree becomes the child of the larger tree? In other words **smaller tree is added to the root node of the larger tree.**

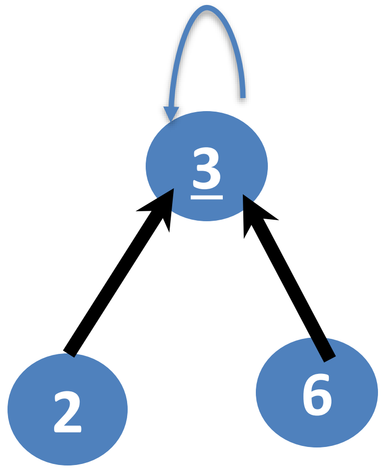
How do we write an algorithm that will do just that ?

What would be the most optimal way to merge these 2 sets

value:parent

parents : { 3:3 , 6:3 , 2:2 }

add_set(3)
add_set(6)
find_rep(6)
merge_sets(3,6)
find_rep(6)
find_rep(3)
add_set(2)
merge_sets(2,6)



How do we write an algorithm that will do just that ?
Keep track of the height ?
Or maybe we can add another dictionary and name it as **rank** ?

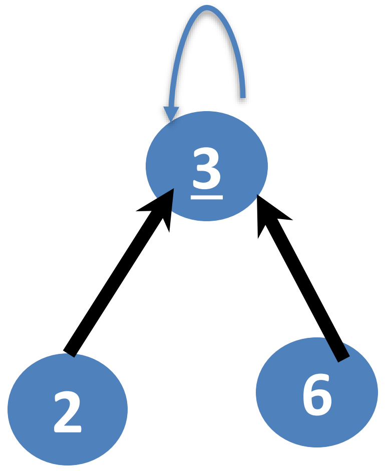
What would be the most optimal way to merge these 2 sets

value:parent

parents : { 3:3 , 6:3 , 2:2 }

rank : { , , , }

add_set(3)
add_set(6)
find_rep(6)
merge_sets(3,6)
find_rep(6)
find_rep(3)
add_set(2)
merge_sets(2,6)



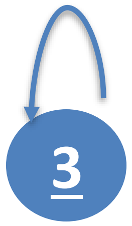
How do we write an algorithm that will do just that ?
Keep track of the height ?
Or maybe we can add another dictionary and name it as rank ?

What would be the most optimal way to merge these 2 sets

value:parent

parents : { 3:3 }

rank : { 3:0 }

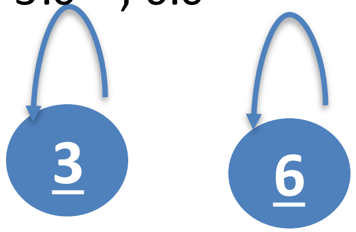


add_set(3)
add_set(6)
find_rep(6)
merge_sets(3,6)
find_rep(6)
find_rep(3)
add_set(2)
merge_sets(2,6)

What would be the most optimal way to merge these 2 sets

value:parent
parents : { 3:3 , 6:6 }

value:rank
rank : { 3:0 , 6:0 }



add_set(3)
add_set(6)
find_rep(6)
merge_sets(3,6)
find_rep(6)
find_rep(3)
add_set(2)
merge_sets(2,6)

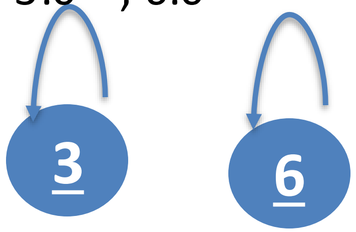
What would be the most optimal way to merge these 2 sets

value:parent

parents : { 3:3 , 6:6 }

value:rank

rank : { 3:0 , 6:0 }



add_set(3)

add_set(6)

find_rep(6)

merge_sets(3,6)

find_rep(6)

find_rep(3)

add_set(2)

merge_sets(2,6)

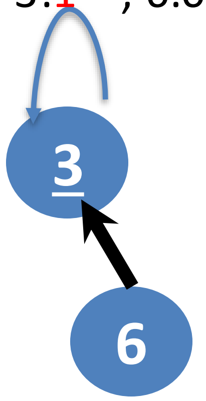
What would be the most optimal way to merge these 2 sets

value:parent

parents : { 3:3 , 6:3 }

value:rank

rank : { 3:1 , 6:0 }



add_set(3)

add_set(6)

find_rep(6)

merge_sets(3,6)

find_rep(6)

find_rep(3)

add_set(2)

merge_sets(2,6)

merge_sets call find_rep for 3 and 6 finding out that they are their own parents, also finds their ranks which are 0 , **they are the same rank** , it doesn't matter which one will be the child let's say 6 will be child of 3.
And now we know that node 3 has more values below it so we must increase its rank, by adding 1 to 3's rank.

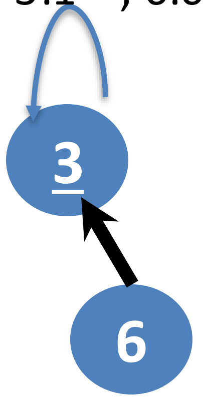
What would be the most optimal way to merge these 2 sets

value:parent

parents : { 3:3 , 6:3 }

value:rank

rank : { 3:1 , 6:0 }



add_set(3)

add_set(6)

find_rep(6)

merge_sets(3,6)

find_rep(6)

find_rep(3)

add_set(2)

merge_sets(2,6)

Next we evaluate the find function for 6, 6 has a parent of 3 , 3 is a parent of itself it should return 3.
Same for find_rep(3), return 3.

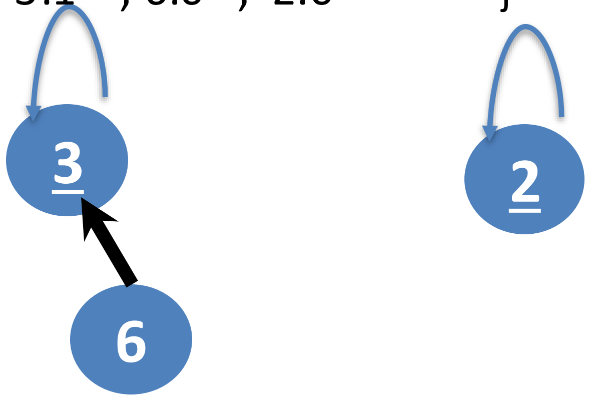
What would be the most optimal way to merge these 2 sets

value:parent

parents : { 3:3 , 6:3 , 2:2 }

value:rank

rank : { 3:1 , 6:0 , 2:0 }



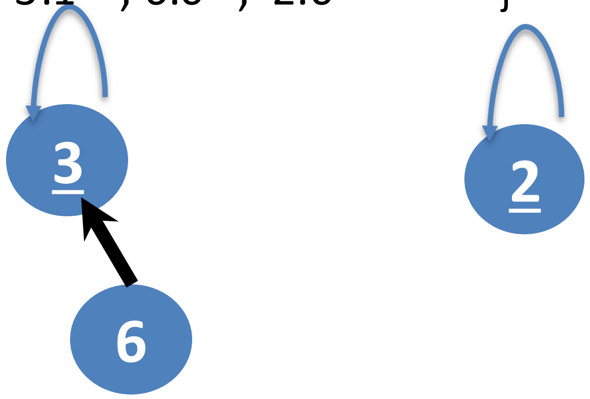
add_set(3)
add_set(6)
find_rep(6)
merge_sets(3,6)
find_rep(6)
find_rep(3)
add_set(2)
merge_sets(2,6)

Next add_set(2)

What would be the most optimal way to merge these 2 sets

value:parent
parents : { 3:3 , 6:3 , 2:2 }

value:rank
rank : { 3:1 , 6:0 , 2:0 }



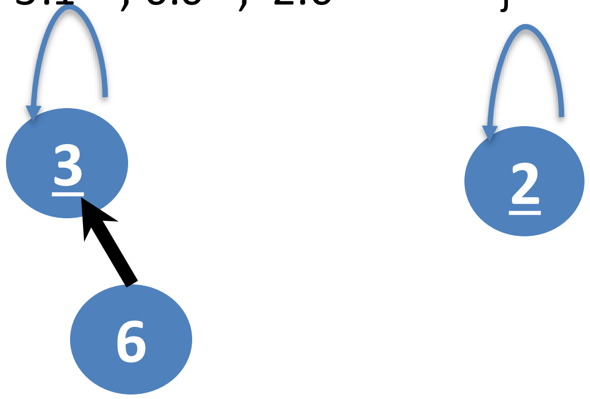
add_set(3)
add_set(6)
find_rep(6)
merge_sets(3,6)
find_rep(6)
find_rep(3)
add_set(2)
merge_sets(2,6)

Next:merge_sets(2,6):
the root node for 2 is 2,
but the root node for 6 is not 6 it is 3. 3 is the self pointer as a parent so 3 is 6's root node. So we stop at 3 for node 6.

What would be the most optimal way to merge these 2 sets

value:parent
parents : { 3:3 , 6:3 , 2:2 }

value:rank
rank : { 3:1 , 6:0 , 2:0 }



add_set(3)
add_set(6)
find_rep(6)
merge_sets(3,6)
find_rep(6)
find_rep(3)
add_set(2)
merge_sets(2,6)

Next:merge_sets(2,6):
the root node for 2 is 2,
but the root node for 6 is not 6 it is 3. 3 is the self pointer as a parent so 3 is 6's root node. So we stop at 3 for node 6.
Essentially we are merging 3 root node with 2 root node.

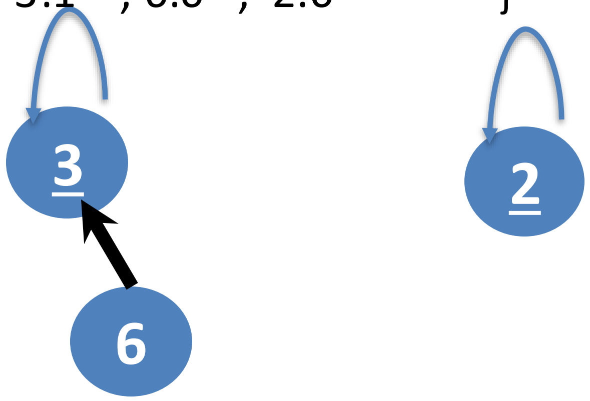
What would be the most optimal way to merge these 2 sets

value:parent

parents : { 3:3 , 6:3 , 2:2 }

value:rank

rank : { 3:1 , 6:0 , 2:0 }



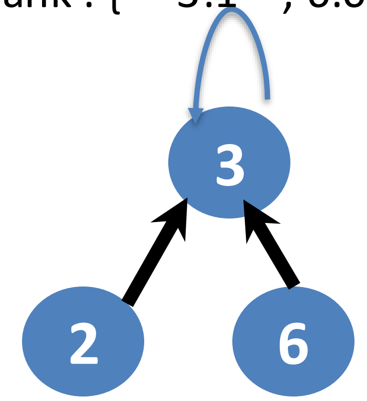
add_set(3)
add_set(6)
find_rep(6)
merge_sets(3,6)
find_rep(6)
find_rep(3)
add_set(2)
merge_sets(2,6)

Next:merge_sets(2,6):
the root node for 2 is 2,
but the root node for 6 is not 6 it is 3. 3 is the self pointer as a parent so 3 is 6's root node. So we stop at 3 for node 6.
Essentially we are merging 3 root node with 2 root node.
But Node 3 has a higher rank than the node 2. This means node 2 is smaller than node 3 in terms of ranking. This means we can move the node 2 under the node 3 without increasing the height of this Node 3 tree.

What would be the most optimal way to merge these 2 sets

value:parent
parents : { 3:3 , 6:3 , 2:3 }

value:rank
rank : { 3:1 , 6:0 , 2:0 }



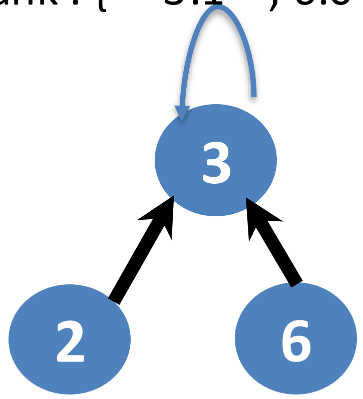
add_set(3)
add_set(6)
find_rep(6)
merge_sets(3,6)
find_rep(6)
find_rep(3)
add_set(2)
merge_sets(2,6)

Next:merge_sets(2,6):
the root node for 2 is 2,
but the root node for 6 is not 6 it is 3. 3 is the self pointer as a parent so 3 is 6’s root node. So we stop at 3 for node 6.
Essentially we are merging 3 root node with 2 root node.
But Node 3 has a higher rank than the node 2. This means node 2 is smaller than node 3 in terms of ranking. This means we can move the node 2 under the node 3 without increasing the height of this Node 3 tree.

What would be the most optimal way to merge these 2 sets

value:parent
parents : { 3:3 , 6:3 , 2:3 }

value:rank
rank : { 3:1 , 6:0 , 2:0 }



add_set(3)
add_set(6)
find_rep(6)
merge_sets(3,6)
find_rep(6)
find_rep(3)
add_set(2)
merge_sets(2,6)

Node 3 has a higher rank than the node 2.
This means **node 2 is smaller than node 3 in terms of ranking**.
This means we can move the node 2 under the node 3 without increasing the height of this Node 3 tree.
We update 2's parent but we do not change the rank of 3 since the height of the tree did not change. (We just added smaller rank tree to node 3 and it will not change the height of the tree.)

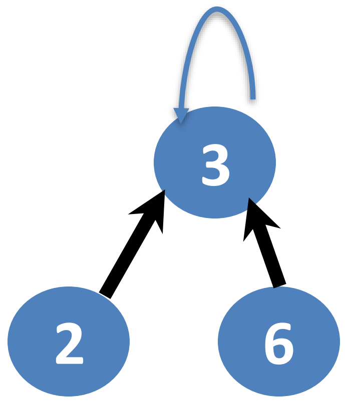
So that is it for merge_sets() by rank !

parents : { 3:3 , 6:3 , 2:3 }

value:parent

rank : { 3:1 , 6:0 , 2:0 }

value:rank



add_set: it does not change
runtime:O(1),
Aux space:O(1)

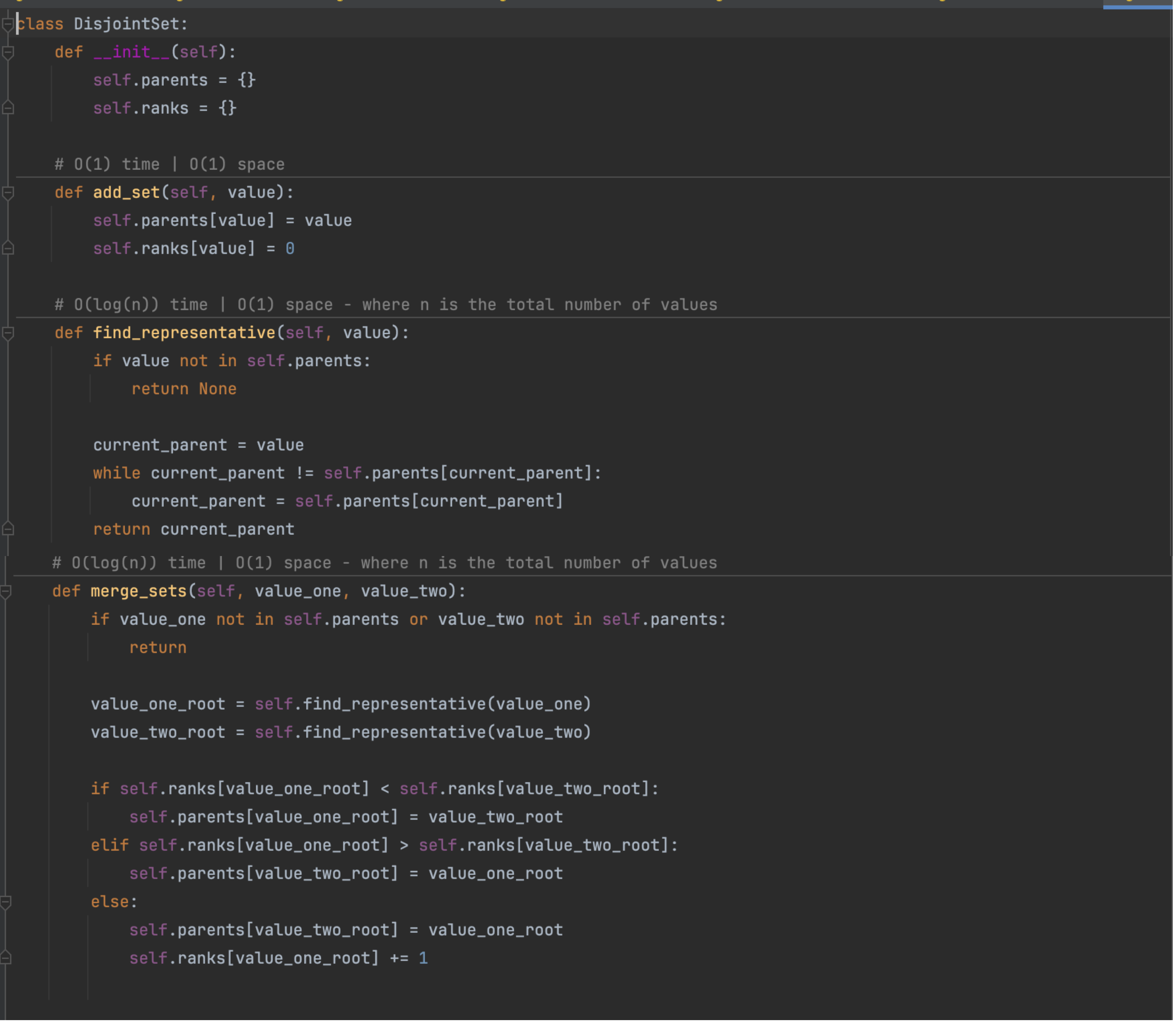
find_rep :
runtime: O(logn) , we know the tree will be balanced, height is log(n)aux space:?

merge_sets :

runtime: O(logn)

add_set(3)
add_set(6)
find_rep(6)
merge_sets(3,6)
find_rep(6)
find_rep(3)
add_set(2)
merge_sets(2,6)

IMPLEMENTING SECOND SOLUTION



```python
class DisjointSet:
    def __init__(self):
        self.parents = {}
        self.ranks = {}

    # O(1) time | O(1) space
    def add_set(self, value):
        self.parents[value] = value
        self.ranks[value] = 0

    # O(log(n)) time | O(1) space - where n is the total number of values
    def find_representative(self, value):
        if value not in self.parents:
            return None

        current_parent = value
        while current_parent != self.parents[current_parent]:
            current_parent = self.parents[current_parent]
        return current_parent

    # O(log(n)) time | O(1) space - where n is the total number of values
    def merge_sets(self, value_one, value_two):
        if value_one not in self.parents or value_two not in self.parents:
            return

        value_one_root = self.find_representative(value_one)
        value_two_root = self.find_representative(value_two)

        if self.ranks[value_one_root] < self.ranks[value_two_root]:
            self.parents[value_one_root] = value_two_root
        elif self.ranks[value_one_root] > self.ranks[value_two_root]:
            self.parents[value_two_root] = value_one_root
        else:
            self.parents[value_two_root] = value_one_root
            self.ranks[value_one_root] += 1
```

THIRD SOLUTION

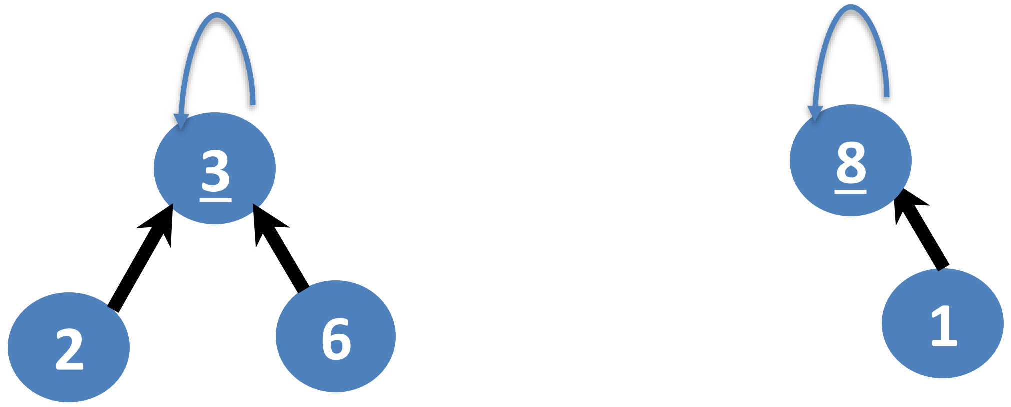
So that is it for merge_sets() by rank !

parents : { 3:3 , 6:3 , 2:3, 8:8 , 1:8 }

value:parent

rank : { 3:1 , 6:0 , 2:0, 8:1, 1:0 }

value:rank



What if we wanted to merge_sets(3 and 8) ?

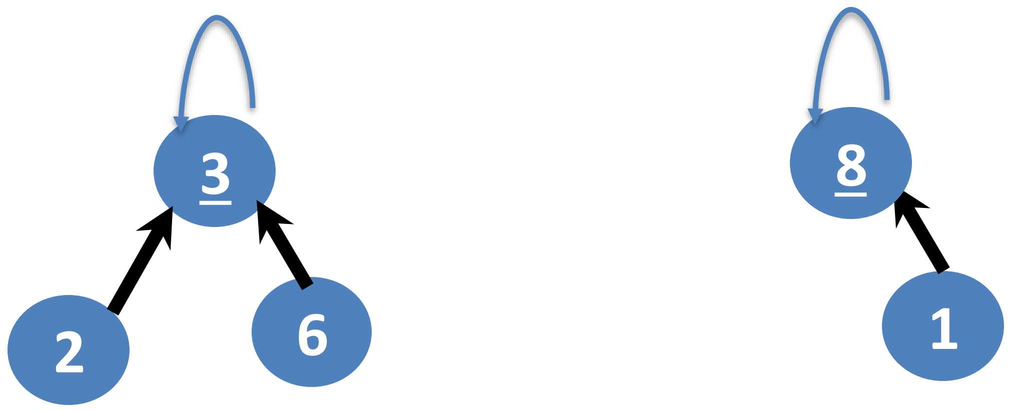
So that is it for merge_sets() by rank !

parents : { 3:3 , 6:3 , 2:3, 8:8, 1:8 , }

value:parent

rank : { 3:1 , 6:0 , 2:0, 8:1, 1:0 , }

value:rank



What if we wanted to merge_sets(3 and 8) ?

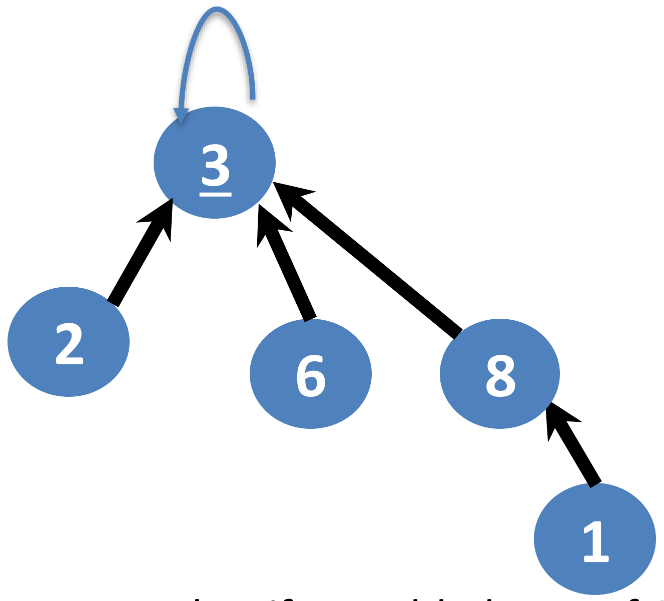
So that is it for merge_sets() by rank !

parents : { 3:3 , 6:3 , 2:3, 8:3 1:8 , }

value:parent

rank : { 3:2 , 6:0 , 2:0, 8:1, 1:0 , }

value:rank



What if we added tree of 8 to 3.
then we update the parents dictionary and also update 3's rank since we merged 2 trees with the same rank we need to increment 3's ranking to 2

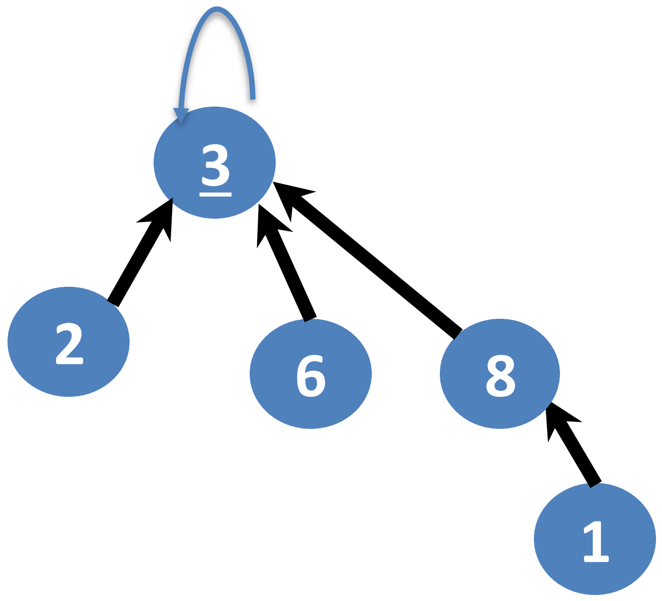
So that is it for merge_sets() by rank !

parents : {   3:3   , 6:3   , 2:3,   8:3   1:8 ,   }

value:parent

rank : {   3:2   , 6:0 , 2:0,   8:1,   1:0 ,   }

value:rank



So if we invoke find_rep(1) we would look at 8 , then its parent 3 and return 3

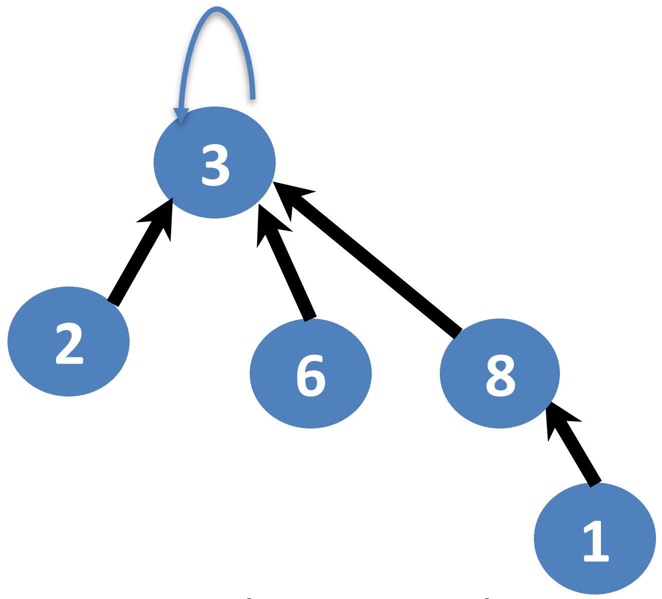
So that is it for merge_sets() by rank !

parents : { 3:3 , 6:3 , 2:3, 8:3 1:8 , }

value:parent

rank : { 3:2 , 6:0 , 2:0, 8:1, 1:0 , }

value:rank



But the intuition here is very simple, there is no reason for this 1 to be here rooted as 8. ?

The 1's parent could easily be 3. that could create one less step for us.

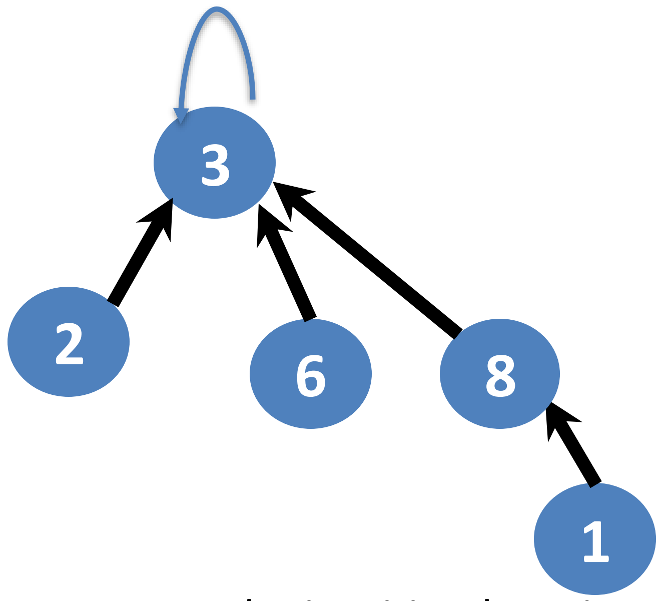
So that is it for merge_sets() by rank !

parents : {   3:3   , 6:3   , 2:3, 8:3   1:8 ,   }

value:parent

rank : {   3:2   , 6:0 , 2:0,   8:1,   1:0 ,   }

value:rank

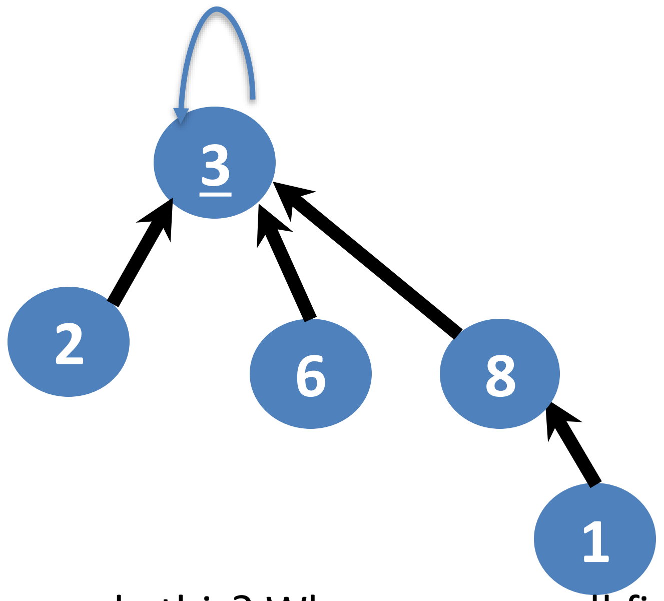


But the intuition here is very simple, there is no reason for this 1 to be here rooted as 8. ?
The 1's parent could easily be 3. that could create one less step for us.
How do we do this?

So that is it for merge_sets() by rank !

parents : {   3:3   , 6:3   , 2:3, 8:3   1:8 ,   }
value:parent

rank : {   3:2   , 6:0 , 2:0,   8:1,   1:0 ,   }
value:rank



How do we do this? Whenever we call find_rep , as we go up to tree, we can just update the parents of every node we see to be that root node.
When we call find_rep(1) we will see that 1's parent (8) is not the root node, what we can do is as we are recursing up to this tree we can update these pointers.
We can say there is no point for 1 to have 8 as its parent.

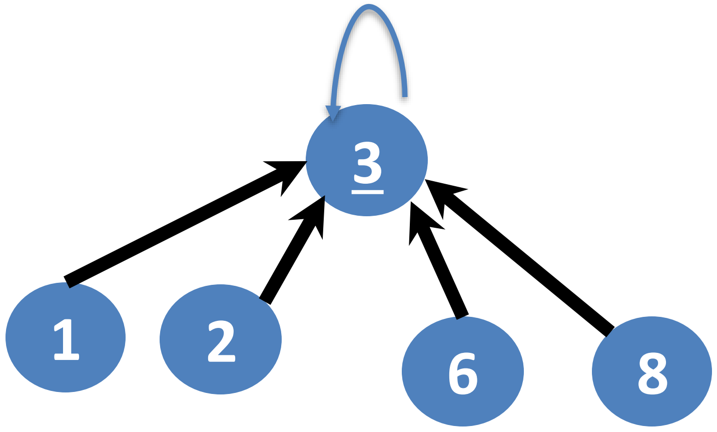
So that is it for merge_sets() by rank !

parents : { 3:3 , 6:3 , 2:3, 8:3 1:3 , }

value:parent

rank : { 3:2 , 6:0 , 2:0, 8:1, 1:0 , }

value:rank



We can say there is no point for 1 to have 8 as its parent.
As we are recursing up the tree we are just going to update node 1 to have a parent of 3.
And update parent's map as well

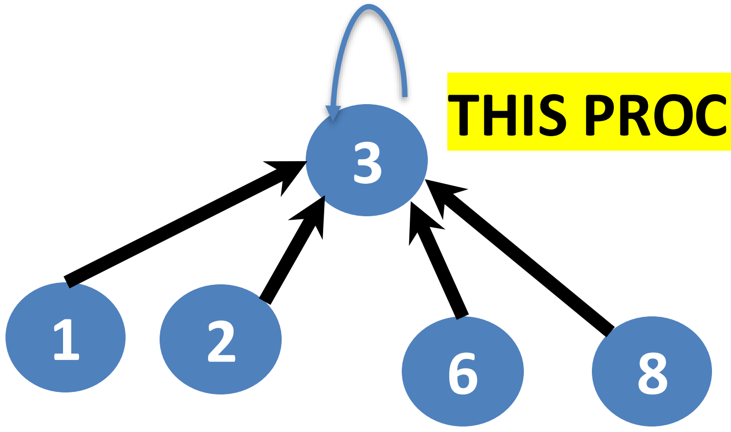
So that is it for merge_sets() by rank !

parents : {   3:3   , 6:3   , 2:3, 8:3   1:3 ,   }

value:parent

rank : {   3:2   , 6:0 , 2:0,   8:1,   1:0 ,   }

value:rank



THIS PROCESS IS KNOWN AS PATH COMPRESSION!

We can say there is no point for 1 to have 8 as its parent.
As we are recursing up the tree we are just going to update node 1 to have a parent of 3. And update parent's map as well

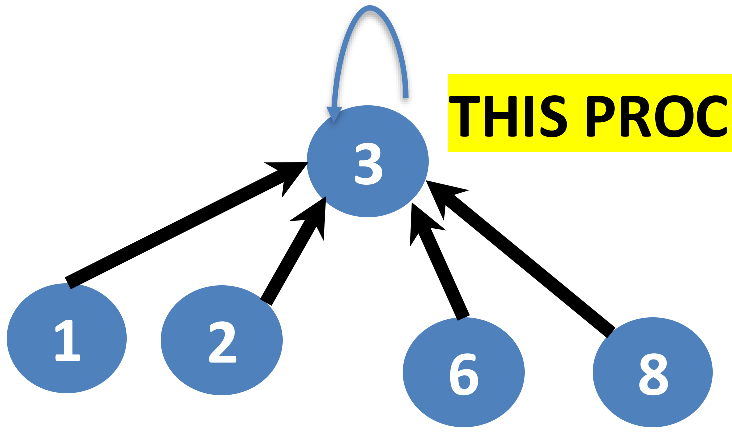
So that is it for merge_sets() by rank !

parents : {   3:3   , 6:3   , 2:3,   8:3   1:3 ,   }

value:parent

rank : {   3:2   , 6:0   , 2:0,   8:1,   1:0 ,   }

value:rank



THIS PROCESS IS KNOWN AS PATH COMPRESSION!

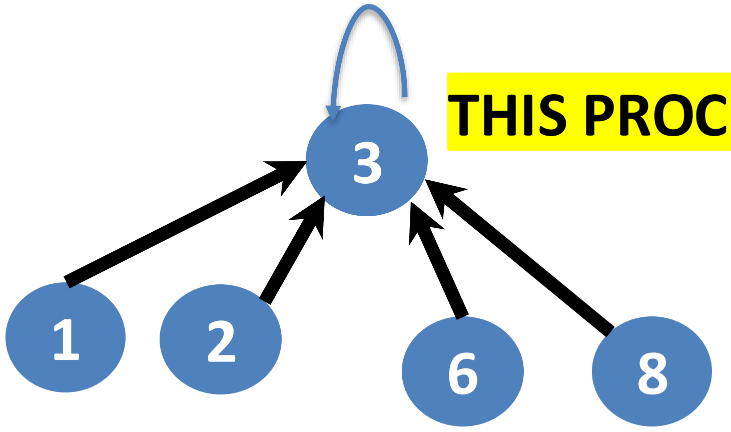
So that is it for merge_sets() by rank !

parents : { 3:3 , 6:3 , 2:3, 8:3 1:3 , }

value:parent

rank : { 3:2 , 6:0 , 2:0, 8:1, 1:0 , }

value:rank



THIS PROCESS IS KNOWN AS PATH COMPRESSION!

If we keep the paths compressed, which means if we keep the paths minimum to the bottom of the tree to the top of the tree than we can keep the time complexity of find_rep and merge_sets down!

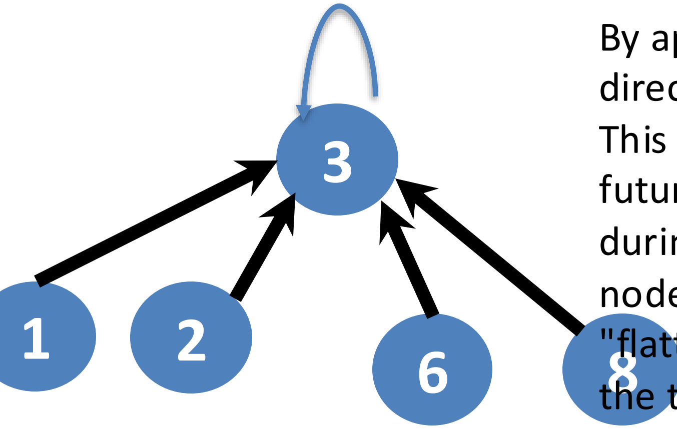
So that is it for merge_sets() by rank !

parents : { 3:3 , 6:3 , 2:3, 8:3 1:3 , }

value:parent

rank : { 3:2 , 6:0 , 2:0, 8:1, 1:0 , }

value:rank



By applying path compression, we ensure that each node points directly to the root of the set it belongs to.
This reduces the number of steps needed to traverse the tree in future operations. The main idea is that as we traverse up the tree during a `find_rep` (find representative) operation, we update each node along the path to point directly to the root, effectively "flattening" the structure. This optimization significantly decreases the time complexity of both `find_rep` and `merge_sets` operations.

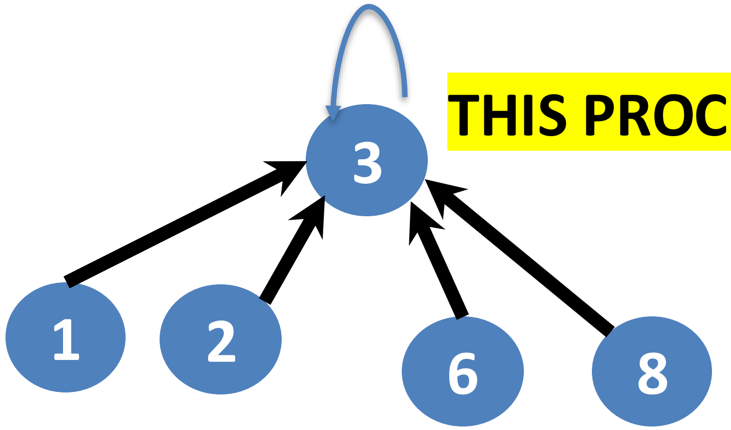
It's also important to note that, after applying path compression, the rank or depth of the tree may no longer correspond to the actual height of the tree, but it will still provide a useful structure for efficient merging.

Example: When recursing up the tree, if node 1 originally pointed to node 2, and node 2 pointed to node 3, after path compression, node 1 will directly point to node 3, making future queries faster.

So that is it for merge_sets() by rank !

parents : { 3:3 , 6:3 , 2:3, 8:3 1:3 , }
value:parent

rank : { 3:2 , 6:0 , 2:0, 8:1, 1:0 , }
value:rank



THIS PROCESS IS KNOWN AS PATH COMPRESSION!

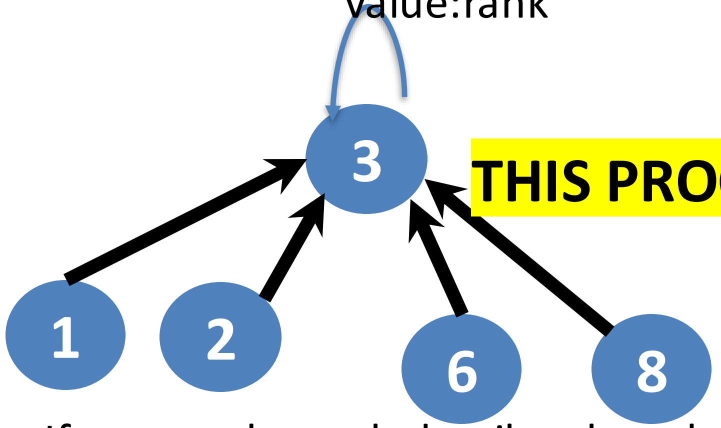
If we keep the paths compressed, which means if we keep the paths minimum to the bottom of the tree to the top of the tree than we can keep the time complexity of find_rep and merge_sets down!
It is also worth noting that the rank no longer will be directly related to height.

Idea: As we are recursing up the tree we are just going to update node say 1 to have a parent of say 3. And update parent’s map in process

So that is it for merge_sets() by rank !

parents : { 3:3 , 6:3 , 2:3, 8:3 1:3 , }
value:parent

rank : { 3:2 , 6:0 , 2:0, 8:1, 1:0 , }
value:rank



THIS PROCESS IS KNOWN AS PATH COMPRESSION!

If we use the path details ; then the time complexity for find_rep become O(α(n))
Where α is inverse Ackerman's function. We will not go in details ,
It is more mathematical may be better suited in 400 level and its proof is not likely to come up in an interview.
Note that this inverse Ackerman's function is always going to return a very small number.
In fact, it will never return a number larger than 4 for any reasonable input.
This means we can say that find_rep function is approximately O(1) for any reasonably sized input.
This makes sense as we are trying to make sure the tree never has a much larger height than 1. Height is constant at 1, run time is roughly O(1).

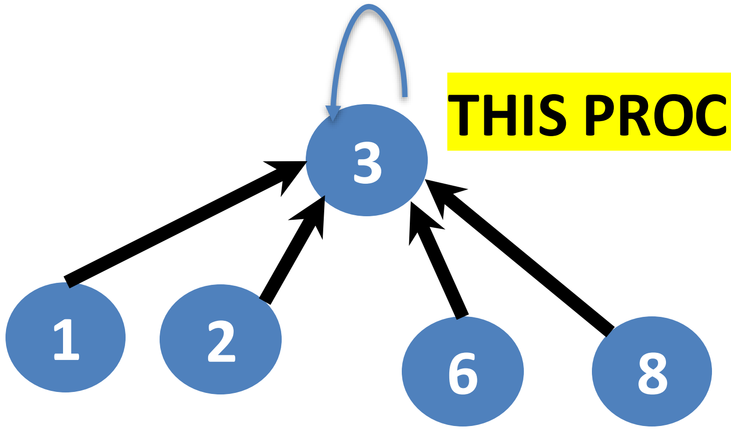
So that is it for merge_sets() by rank !

parents : { 3:3 , 6:3 , 2:3, 8:3 1:3 , }

value:parent

rank : { 3:2 , 6:0 , 2:0, 8:1, 1:0 , }

value:rank



THIS PROCESS IS KNOWN AS PATH COMPRESSION!

Same time time complexity applies for merge_sets is roughly O(1).