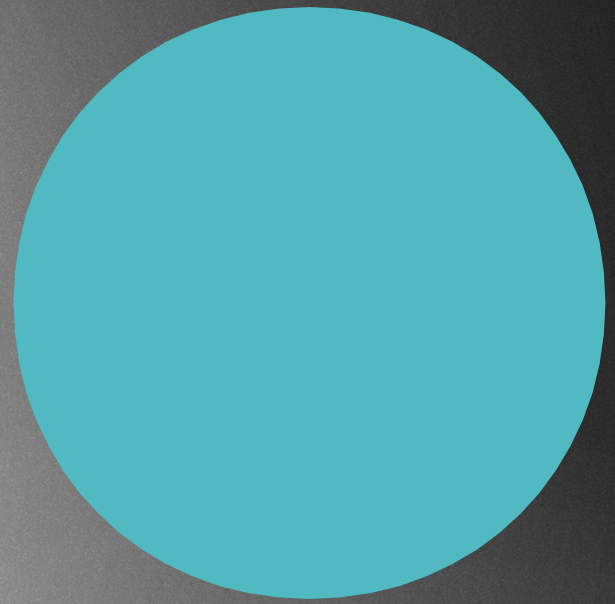
DISJOINT SET



Union find data structure

Disjoint sets

- Also known as union-find data structures
- Data structure to keep track of a set of elements partitioned into a number of disjoint (non everlapping) subsets
- Three main operations: **union** and **find** and **makeSet**
- Disjoint sets can be represented with the help of linked lists but usually we implement it as a tree like structure
- In Kruskal algorithm it will be useful: with disjoint sets we can decide in approximately O(1) time whether two vertexes are in the same set or not

makeSet

```
function makeSet(x)
    x.parent = x
```

So the make sets operation is quite easy to implement
~ we set the parent of the given node to be itself

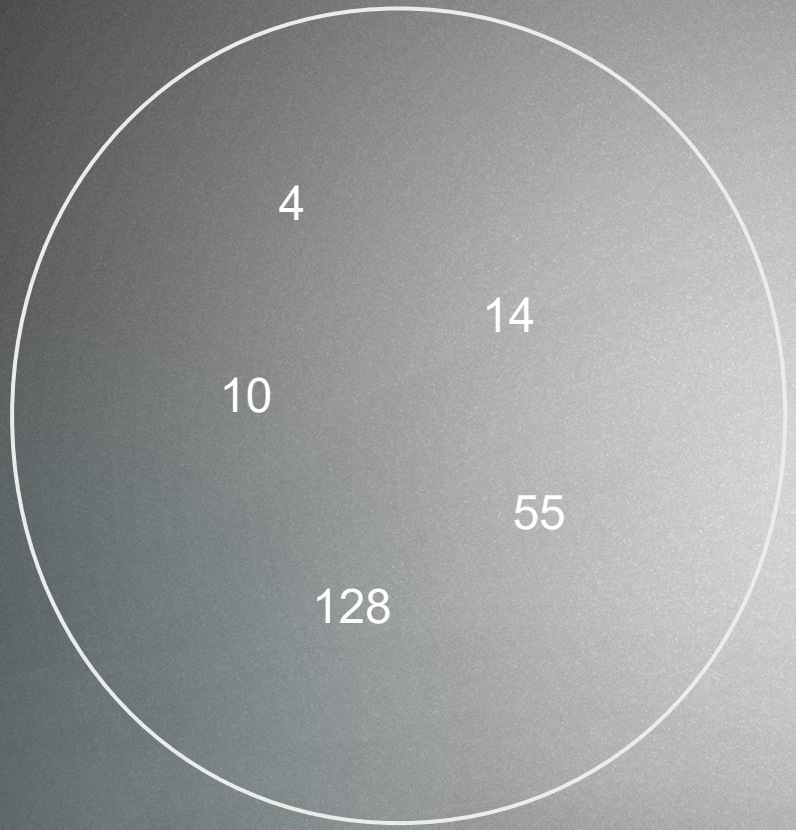
Basically we create a distinct set to all the items/nodes

find

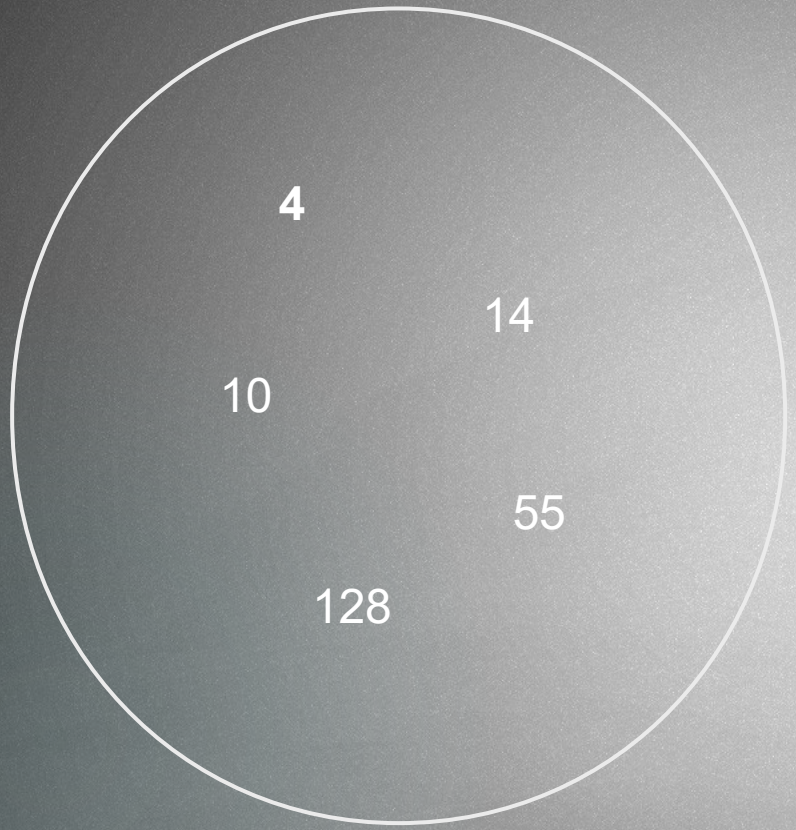
```
function find(x)
        if x.parent == x
                return x
        else
                return find(x.parent)
```

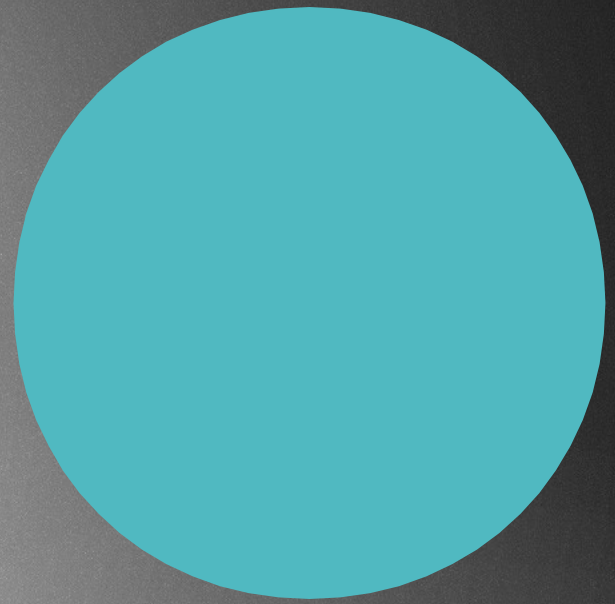
Several items can belong to the same set → we usually represent the set with one of its items „representative of the set”

When we search for an item with find() then the operation is going to return with the representative

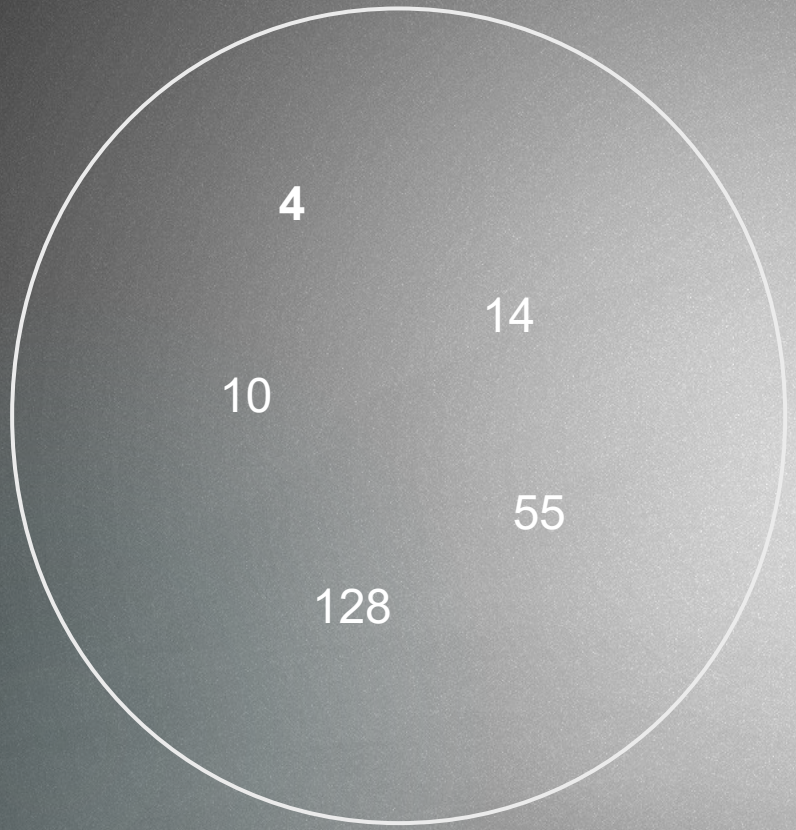

4
14
10
55
128

Representative: 4

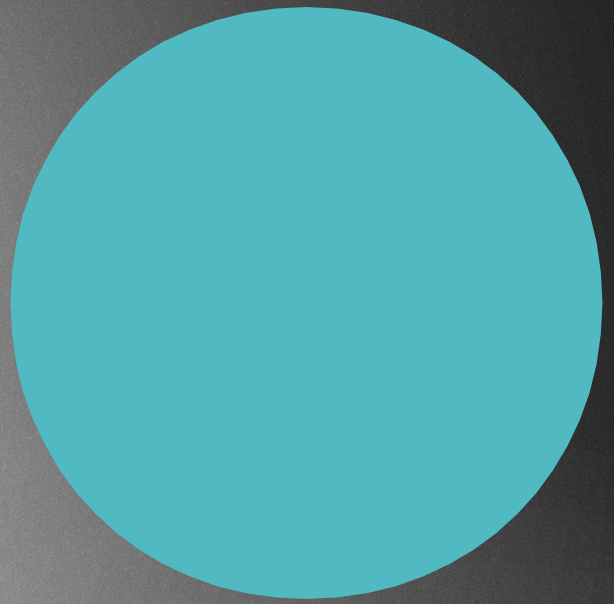




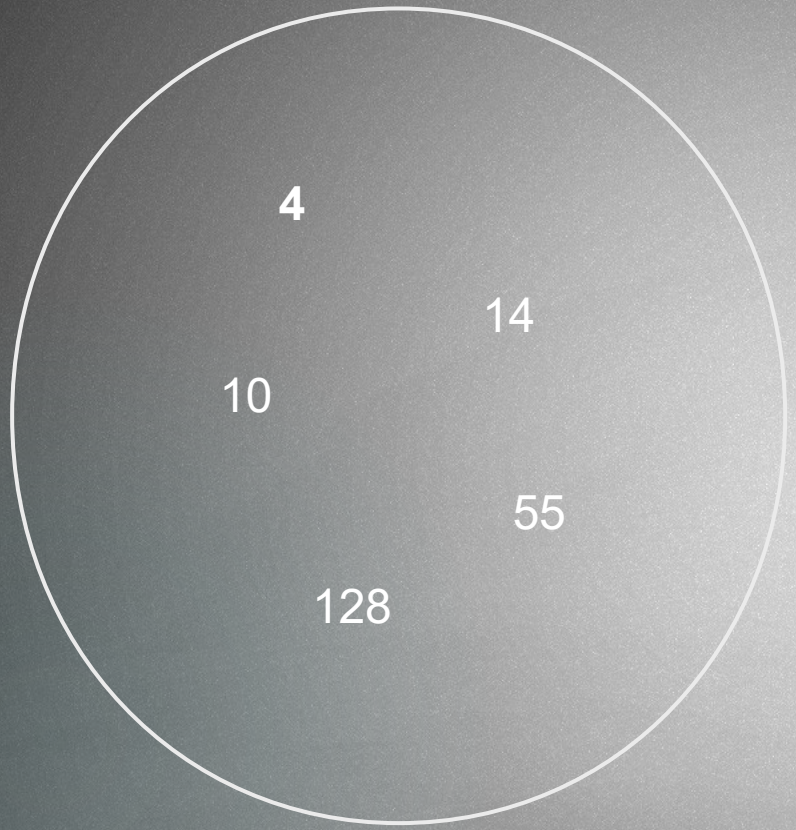
Representative: 4



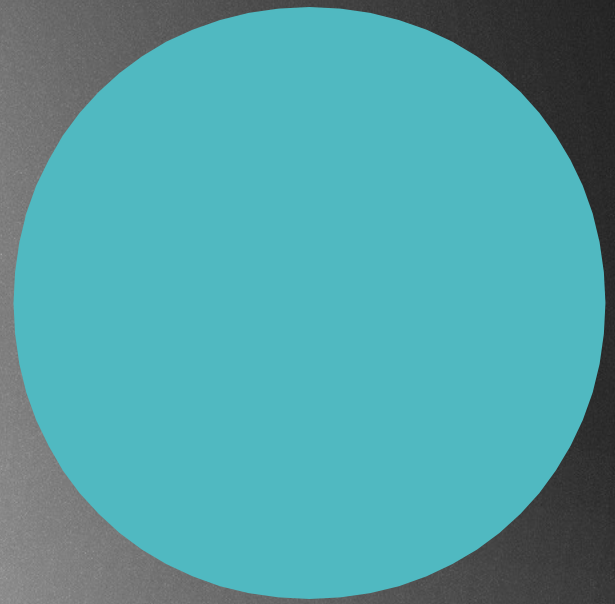
find(4) =



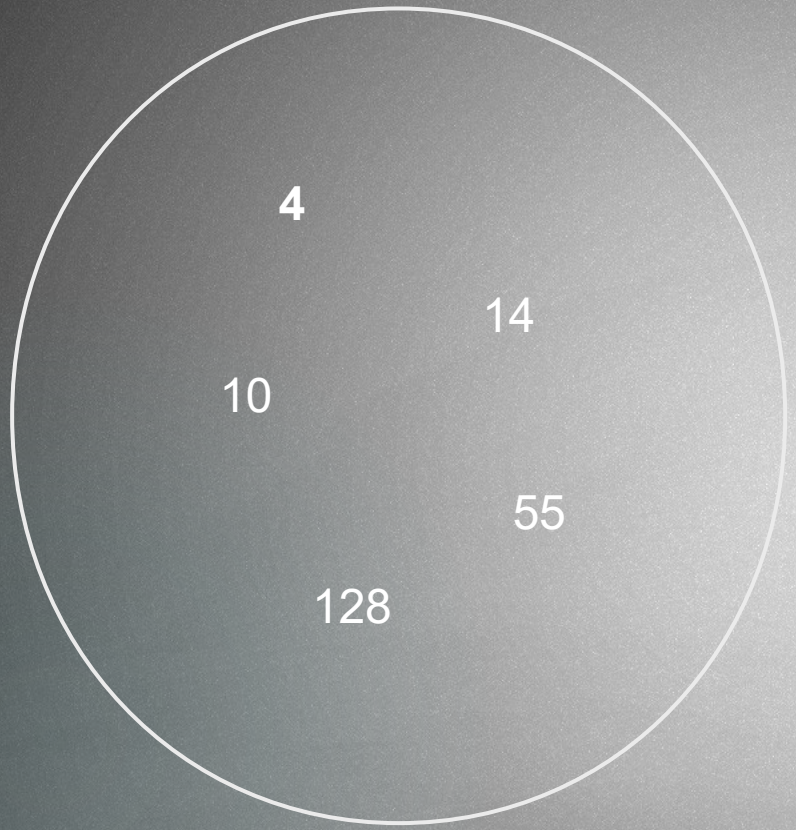
Representative: 4



find(4) = 4

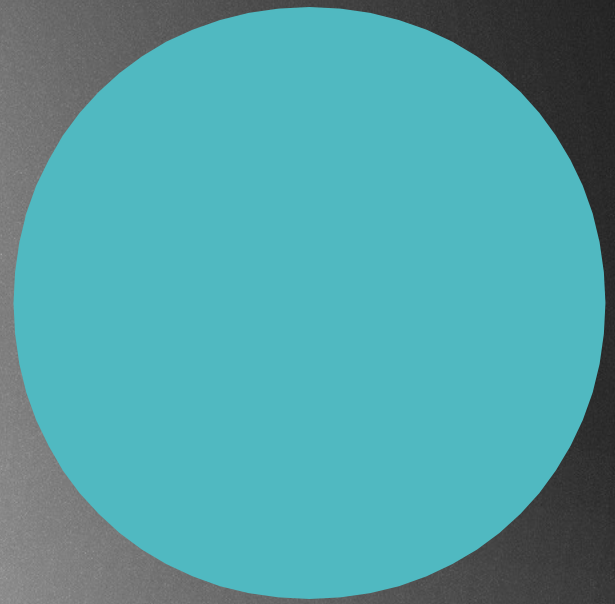


Representative: 4

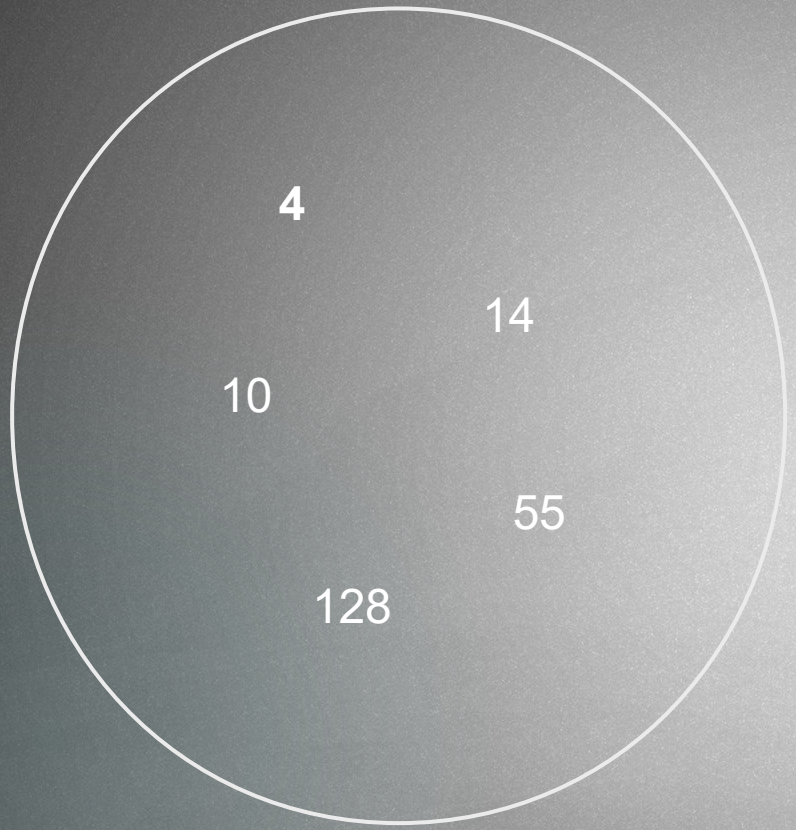


find(4) = 4

find(10) =

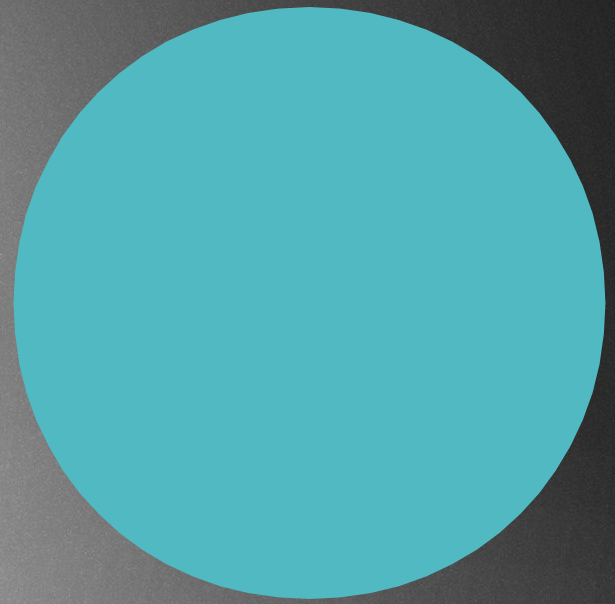


Representative: 4

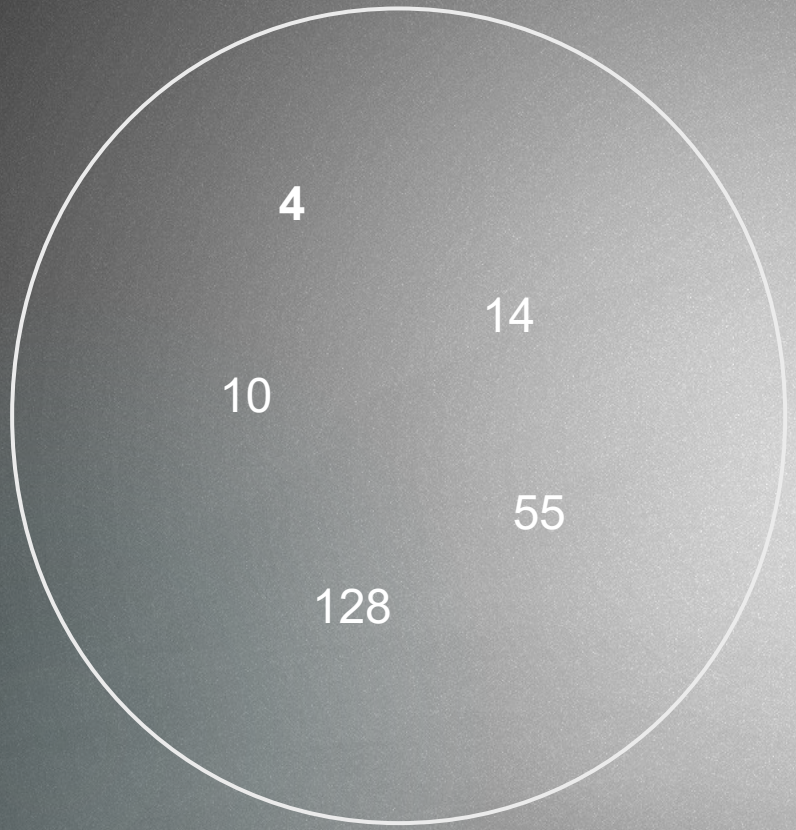


find(4) = 4

find(10) = 4



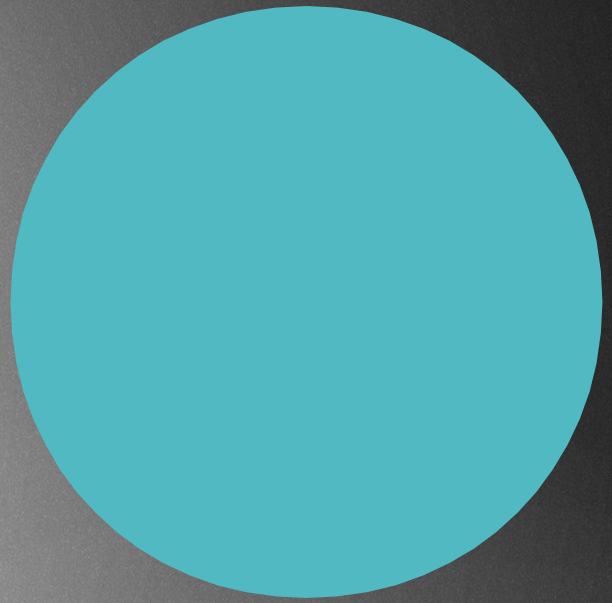
Representative: 4



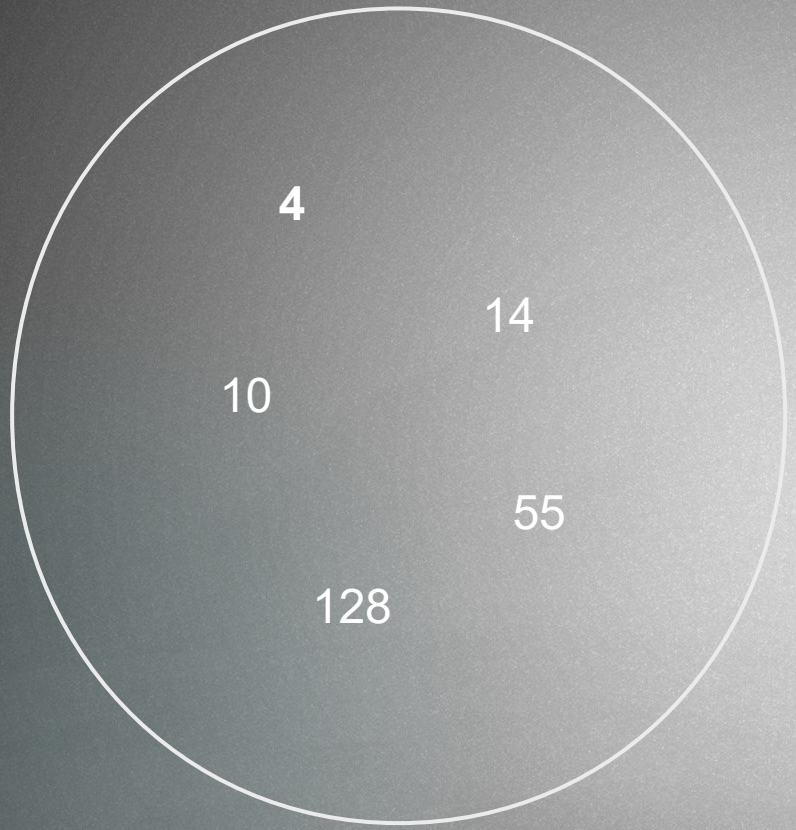
find(4) = 4

find(10) = 4

find(55) =



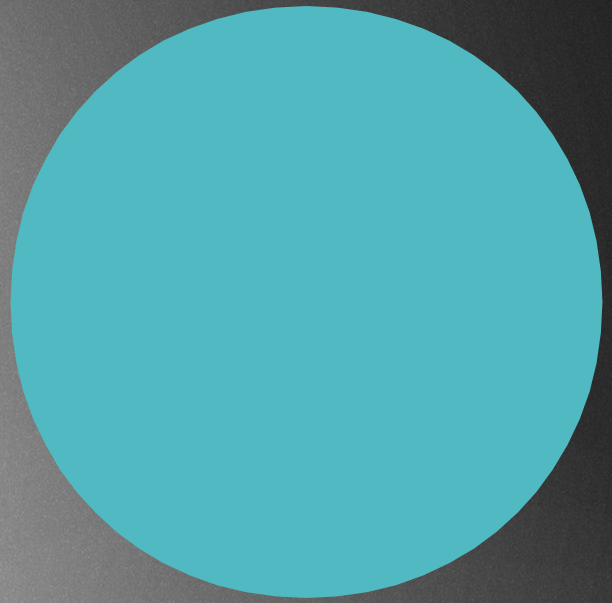
Representative: 4



find(4) = 4

find(10) = 4

find(55) = 4



union

```
function union(x,y)
    xRoot = find(x)
    yRoot = find(y)

    xRoot.parent = yRoot
```

The union operation is merge two disjoint sets together by connecting them according to the representatives

PROBLEM: this tree like structure can become unbalanced

union

```
function union(x,y)
    xRoot = find(x)
    yRoot = find(y)

    xRoot.parent = yRoot
```

The union operation is merge two disjoint sets together by connecting them according to the representatives

PROBLEM: this tree like structure can become unbalanced

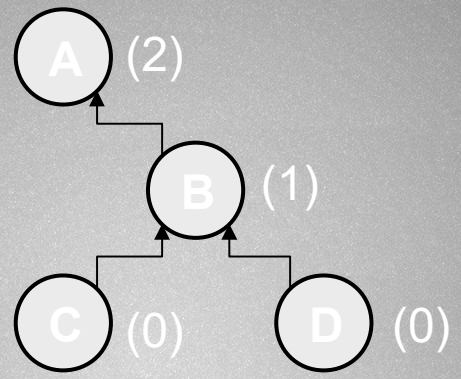
1.) union by rank → always attach the smaller tree to the root of the larger one

The tree will become more balanced: faster !!!

2.) path compression → flattening the structure of the tree

We set every visited node to be connected to the root directly !!!

rank basically the depth of the tree



The rank of the set is equal to the rank of the representative // ~ the root node

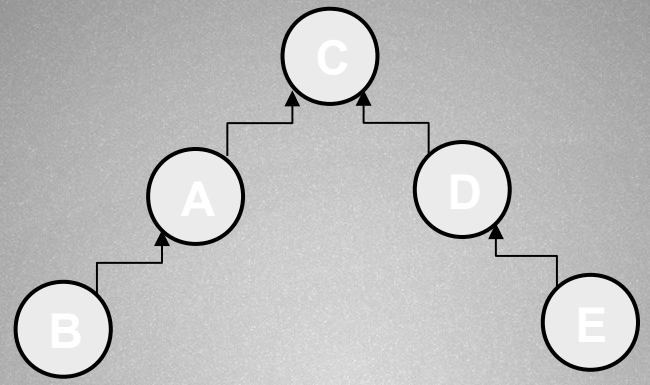
We attack the smaller tree to the larger one → it means we attach the tree with smaller rank to the tree with higher rank !!!

rank basically the depth of the tree

The rank of the set is equal to the rank of the representative // ~ the root node

We attack the smaller tree to the larger one → it means we attack the tree with smallest rank to the tree with highest rank !!!

rank basically the depth of the tree

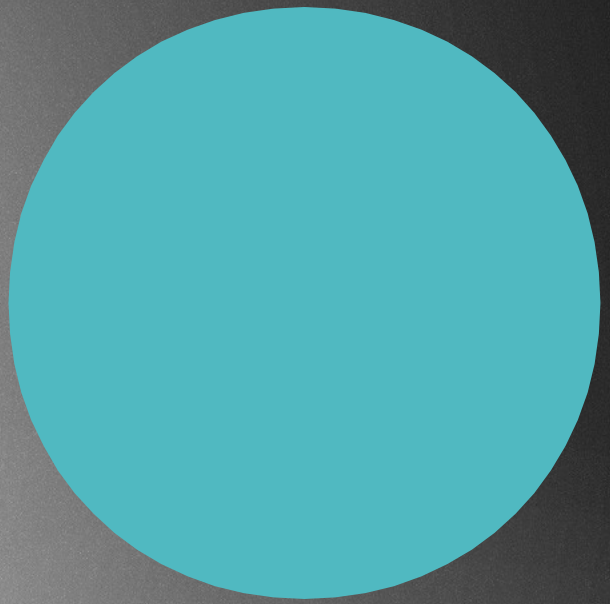


The rank of the set is equal to the rank of the representative // ~ the root node

We attack the smaller tree to the larger one → it means we attack the tree with smallest rank to the tree with highest rank !!!

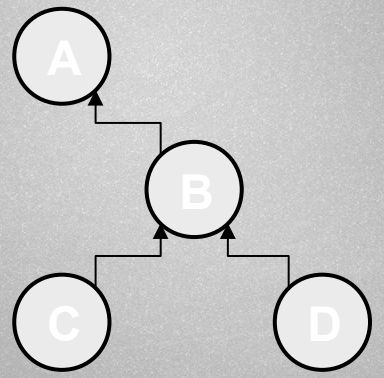
pathCompression

```
function find(x)
    if x.parent != x
        x.parent = find(x.parent)
    return x.parent
```



pathCompression

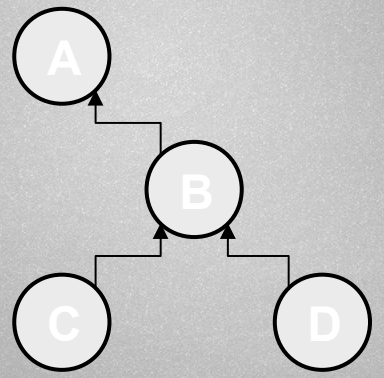
```
function find (x)
        if x.parent != x
                x.parent = find (x.parent)
        return x.parent
```



pathCompression

```
function find (x)
        if x.parent != x
                x.parent = find (x.parent)
        return x.parent
```

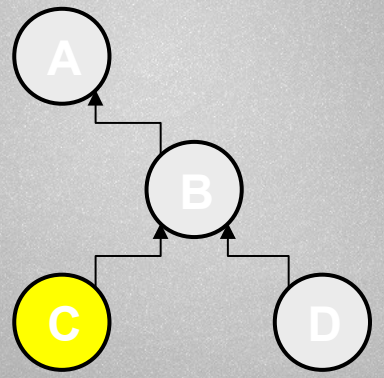
find(C)



pathCompression

```
function find (x)
        if x.parent != x
                x.parent = find (x.parent)
        return x.parent
```

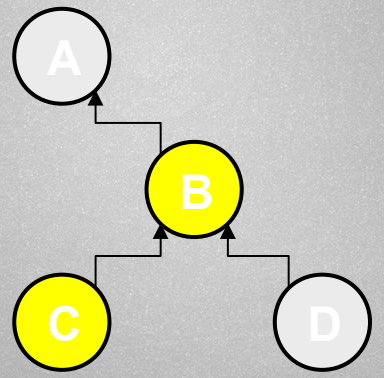
find(C)



pathCompression

```
function find (x)
        if x.parent != x
                x.parent = find (x.parent)
        return x.parent
```

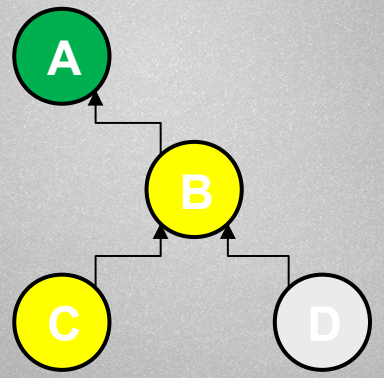
find(C)



pathCompression

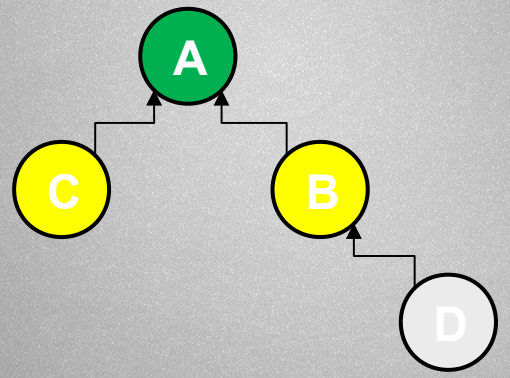
```
function find (x)
        if x.parent != x
                x.parent = find (x.parent)
        return x.parent
```

find(C)



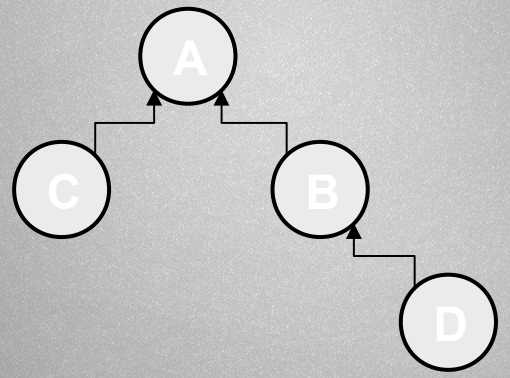
pathCompression

```
function find (x)
        if x.parent != x
                x.parent = find (x.parent)
        return x.parent
```



pathCompression

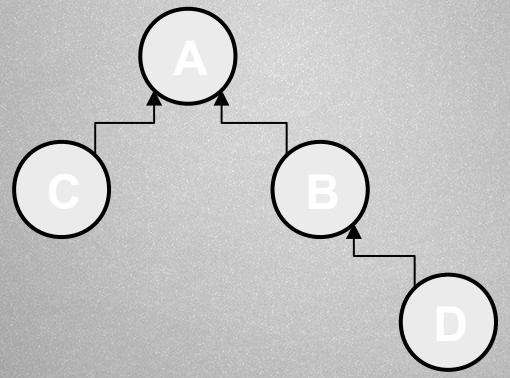
```
function find (x)
        if x.parent != x
                x.parent = find (x.parent)
        return x.parent
```



pathCompression

```
function find (x)
        if x.parent != x
                x.parent = find (x.parent)
        return x.parent
```

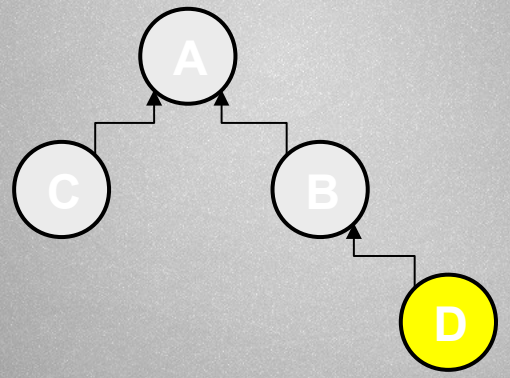
find(D)



pathCompression

```
function find (x)
        if x.parent != x
                x.parent = find (x.parent)
        return x.parent
```

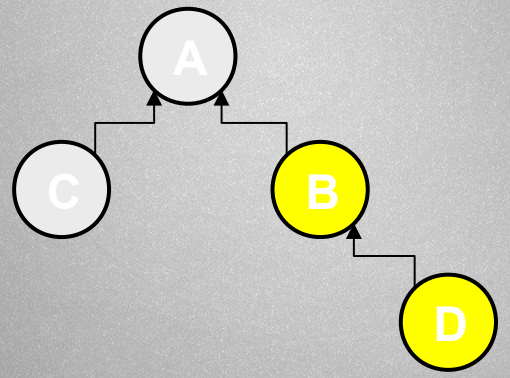
find(D)



pathCompression

```
function find (x)
        if x.parent != x
                x.parent = find (x.parent)
        return x.parent
```

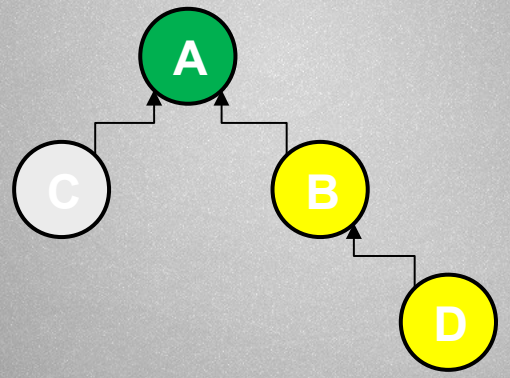
find(D)



pathCompression

```
function find (x)
        if x.parent != x
                x.parent = find (x.parent)
        return x.parent
```

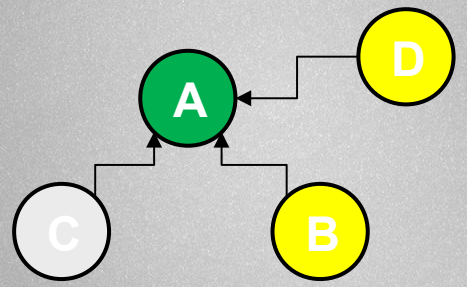
find(D)



pathCompression

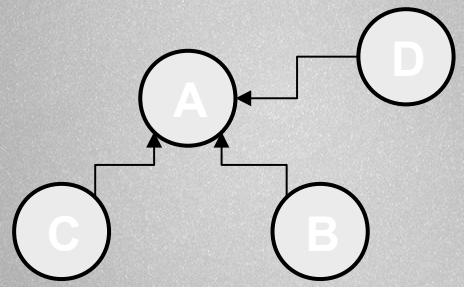
```
function find (x)
        if x.parent != x
                x.parent = find (x.parent)
        return x.parent
```

find(D)



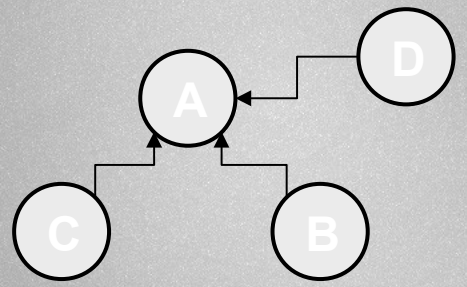
pathCompression

```
function find (x)
        if x.parent != x
                x.parent = find(x.parent)
        return x.parent
```



pathCompression

```
function find (x)
    if x.parent != x
        x.parent = find (x.parent)
    return x.parent
```



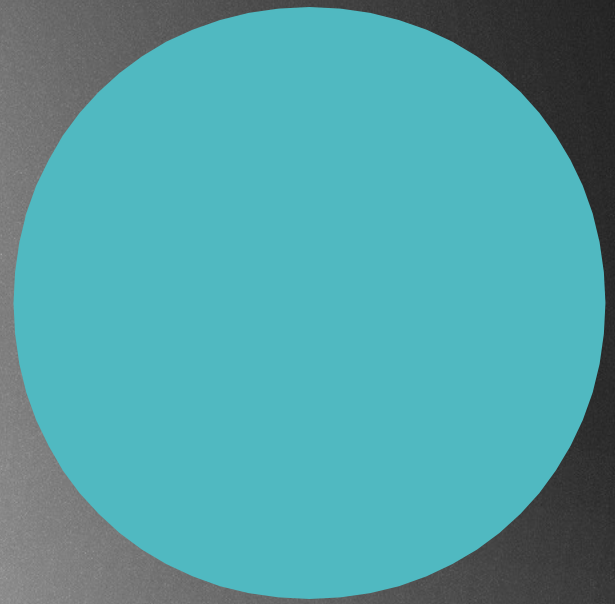
Why is it good? The next time we want to find(C) or find(D) it will take O(1) time because they are the direct neighbour of the representative !!!

~ the algorithm will be faster because of the „path compression”

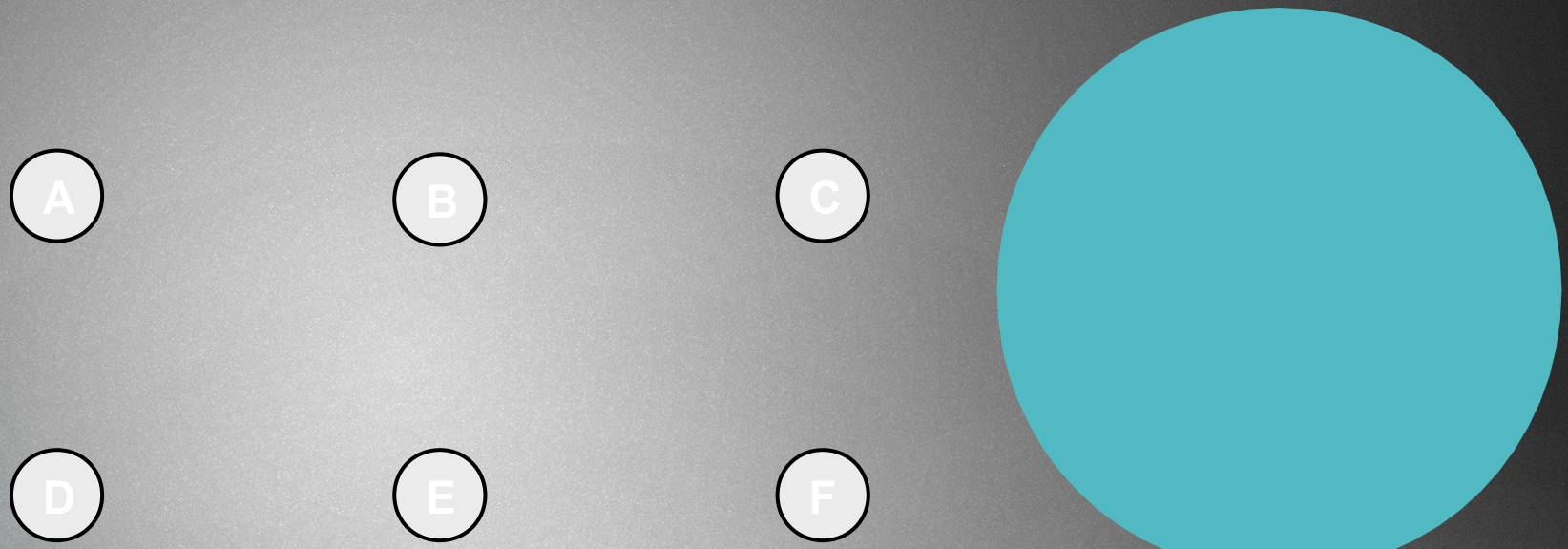
Applications

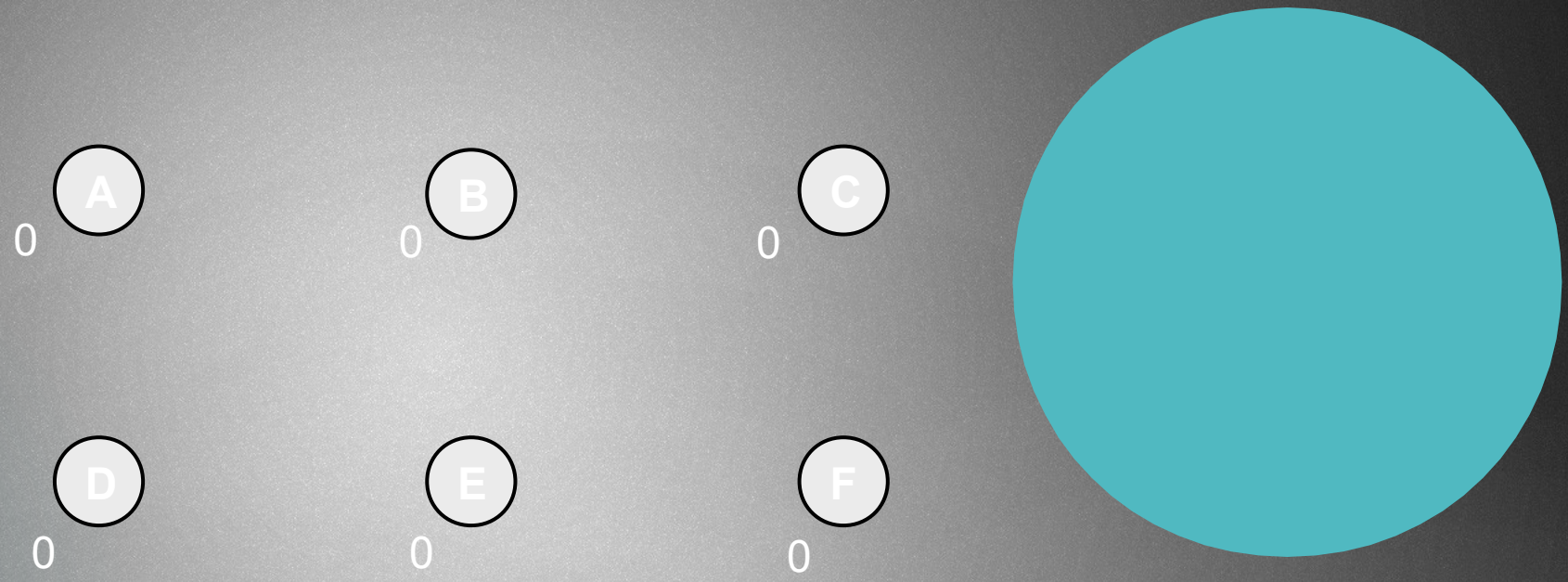
- It is used mostly in Kruskal-algorithm implementation
- We have to check whether adding a given edge to the MST would form a cycle or not
- For checking this → union-find data structure is extremely helpful
- We can check whether a cycle is present → in asymtotically O(1) constant time complexity !!!

DISJOINT SET



Union find data structure


A
B
C
D
E
F

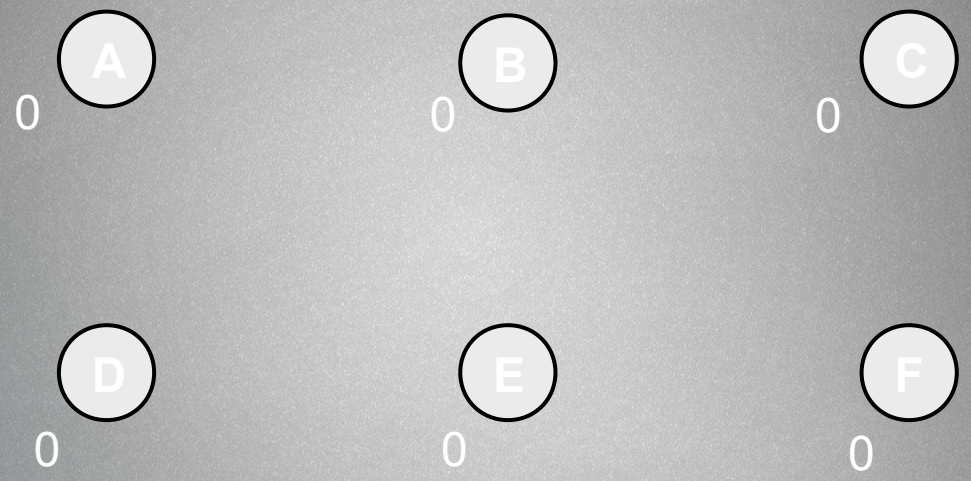

A
0
B
0
C
0
D
0
E
0
F
0

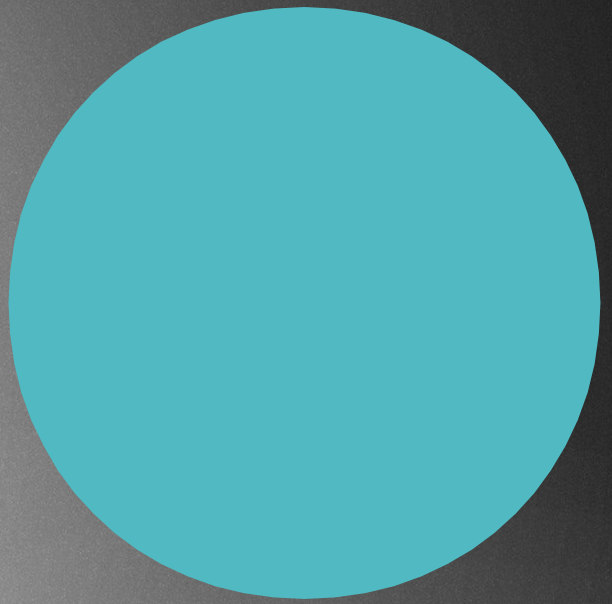
merge(A,B)

We make the set with lower rank to be the child of the set with higher rank

~ it keeps the depth of the tree as low as possible !!!

+ we have to operate with the representatives always

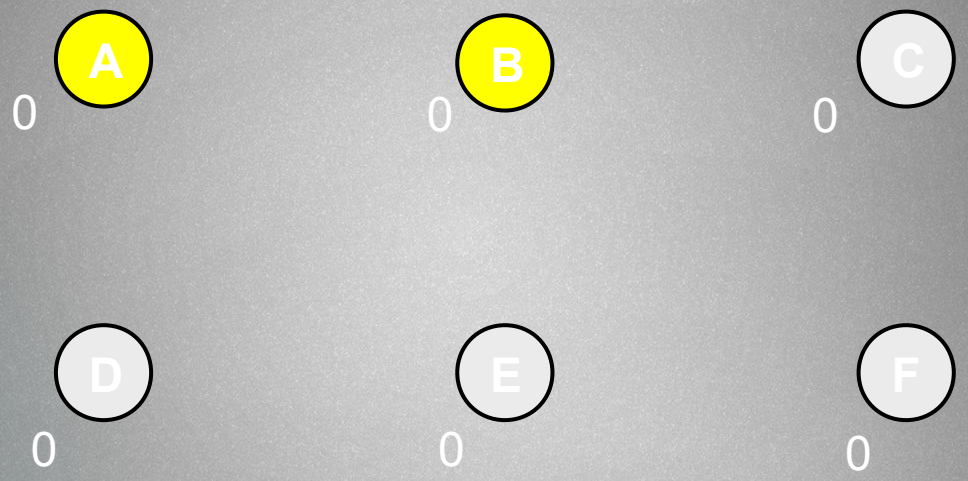


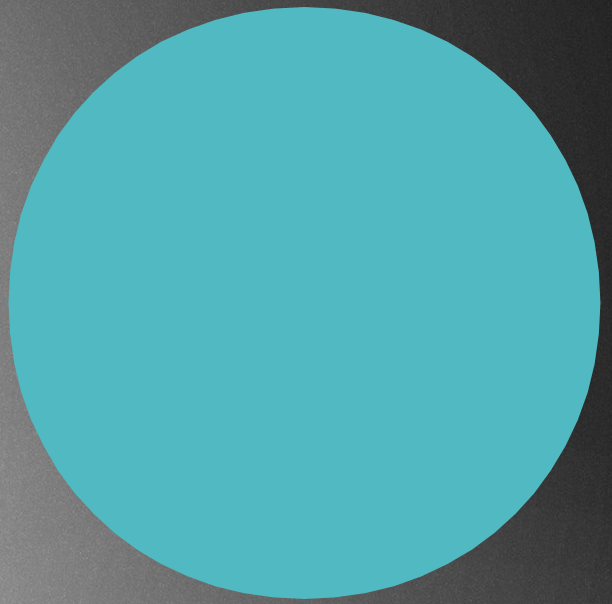


merge(A,B)

We make the set with lower rank to be the child of the set with higher rank

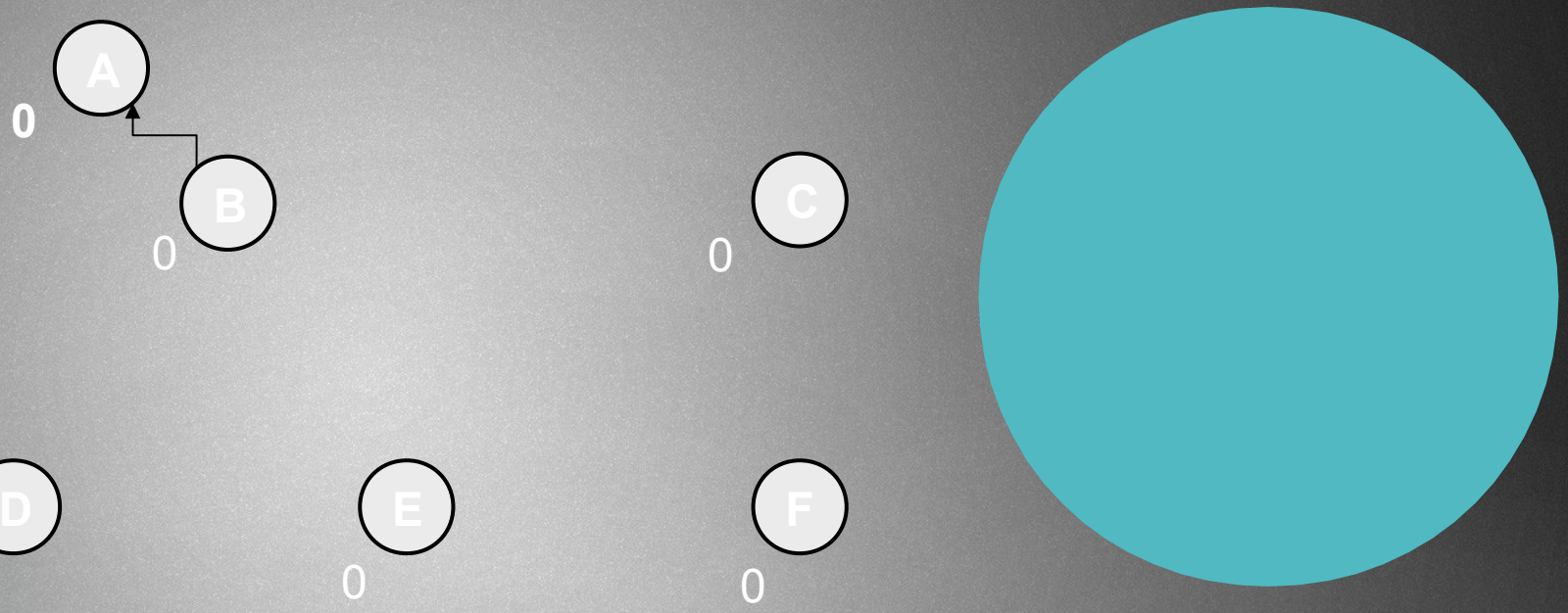
~ it keeps the depth of the tree as low as possible !!!



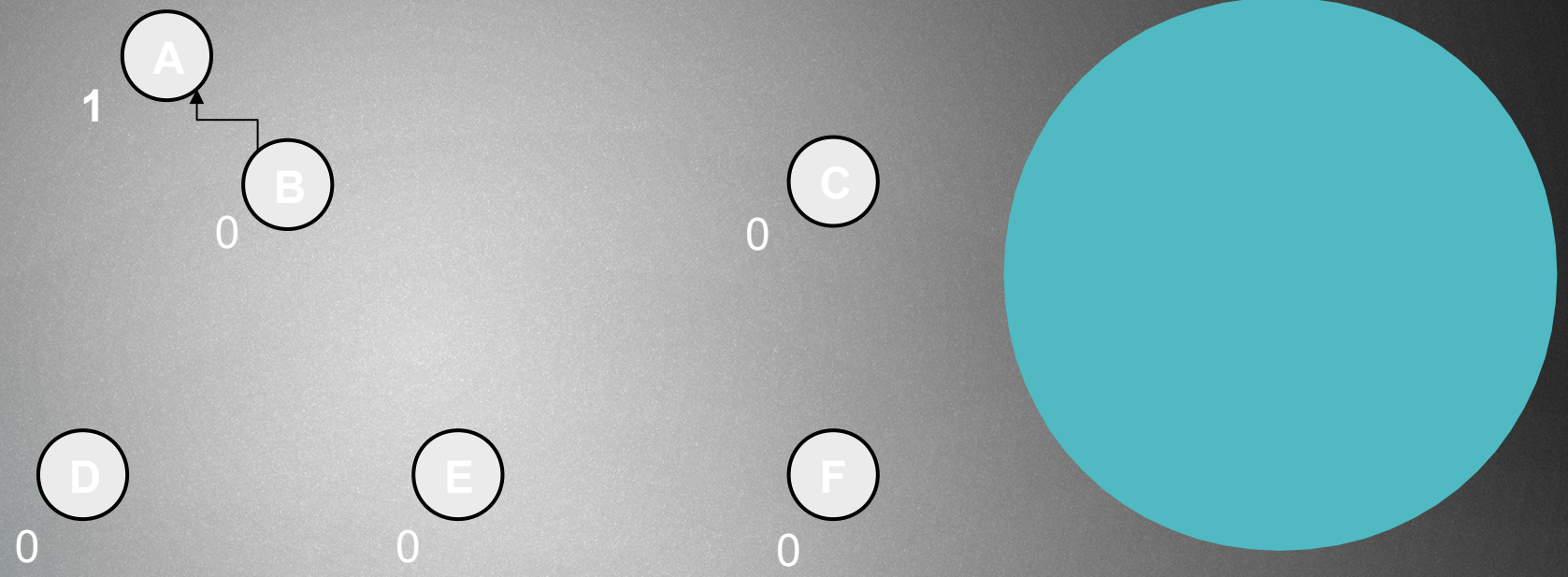


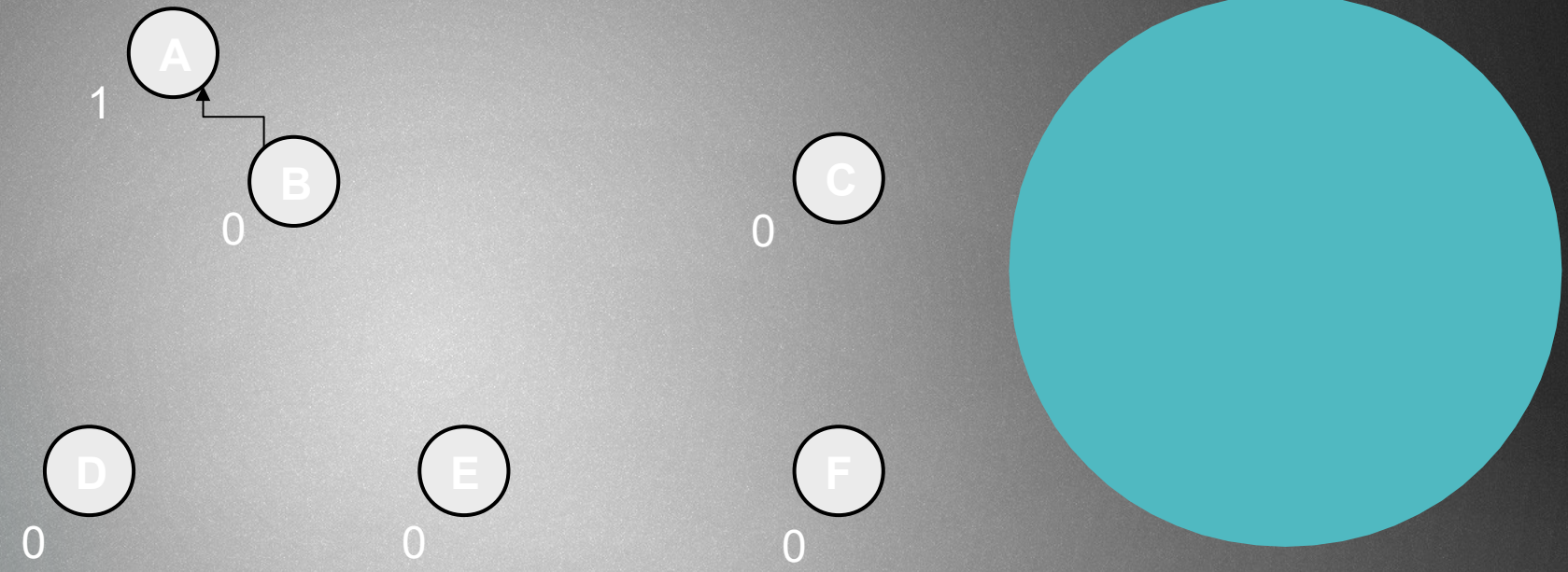
merge(A,B)

We increment the rank ONLY if the rank parameters were the same before the merge operation

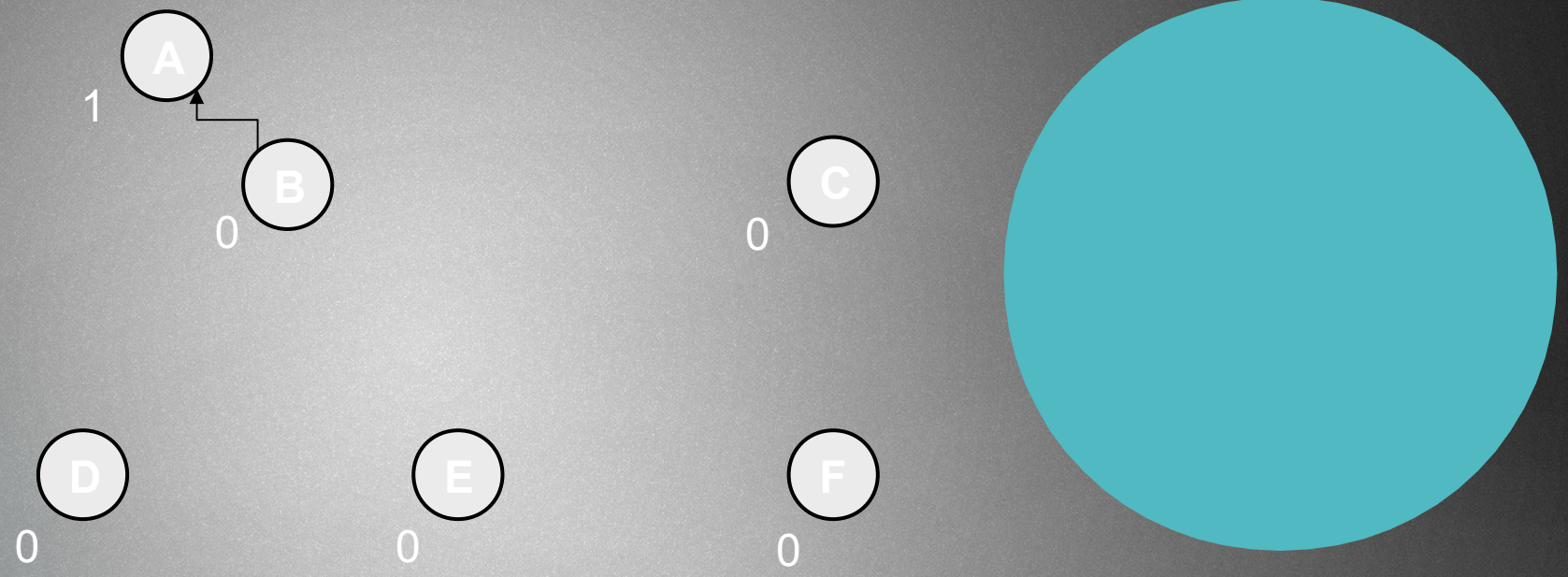


merge(A,B)



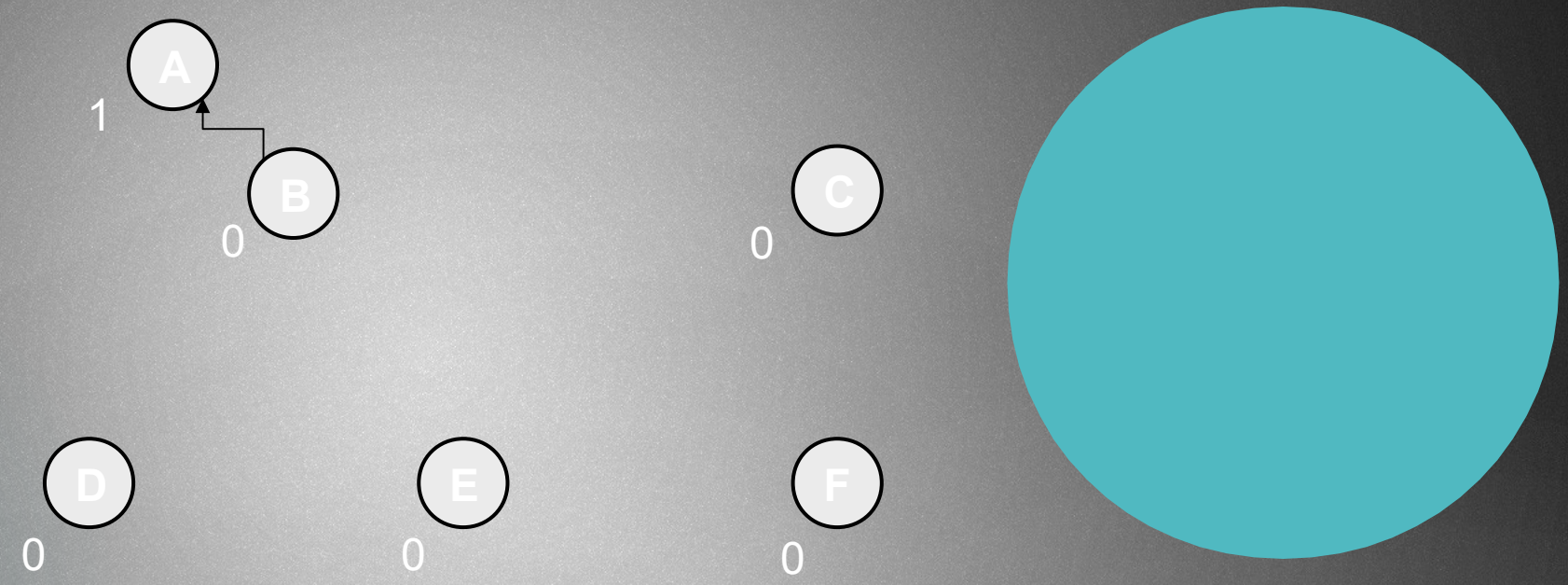

A
1
B
0
C
0
D
0
E
0
F
0

merge(B,C)



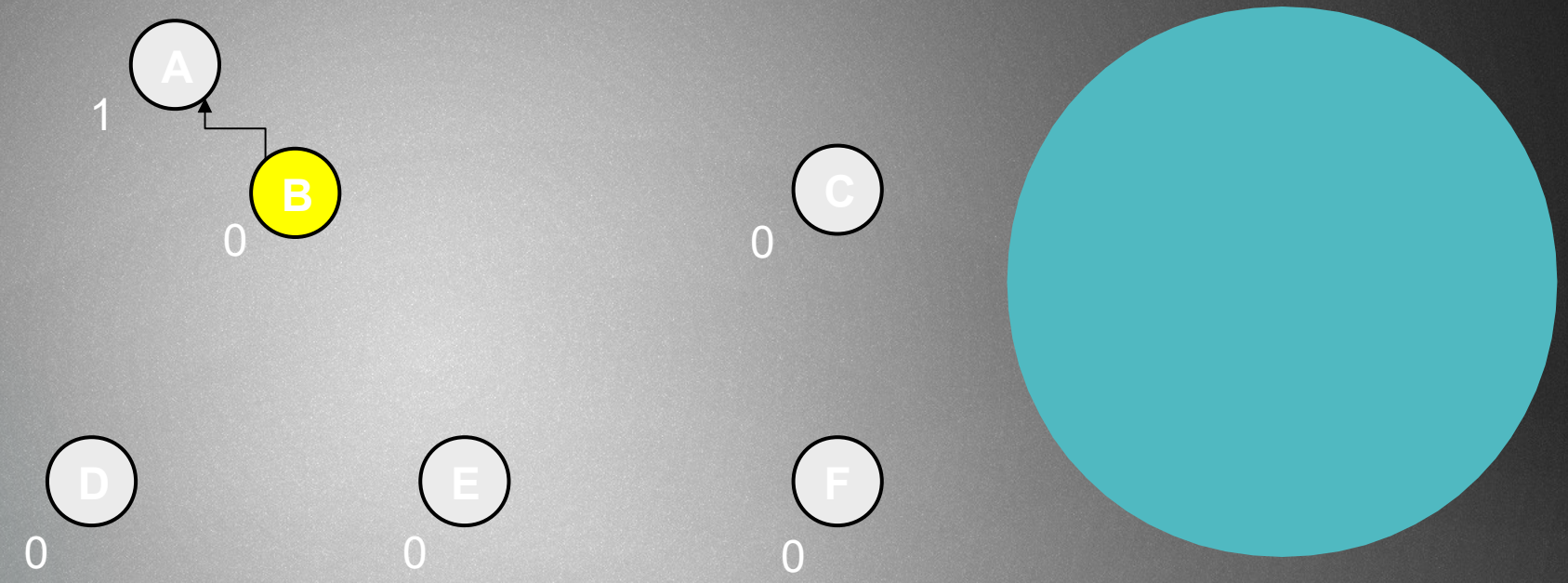
merge(B,C)

First we have to find the representatives in both sets: and merge them together !!!



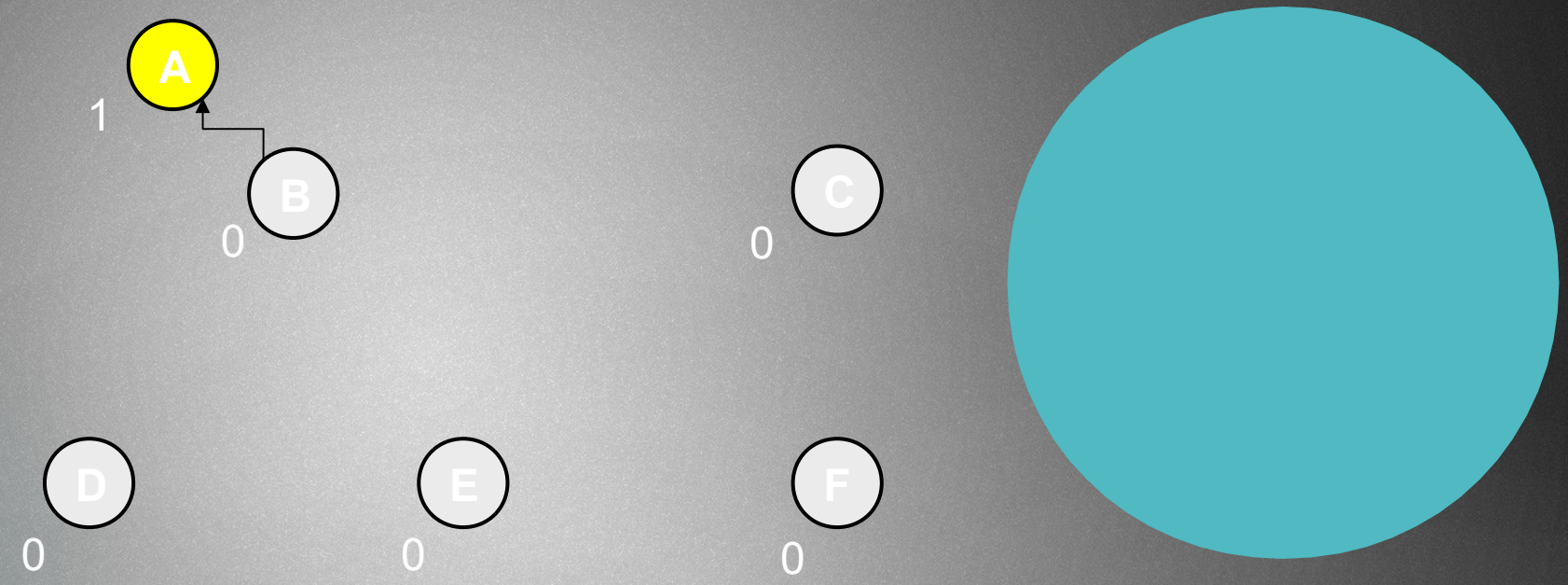
merge(B,C)

First we have to find the representatives in both sets: and merge them together !!!



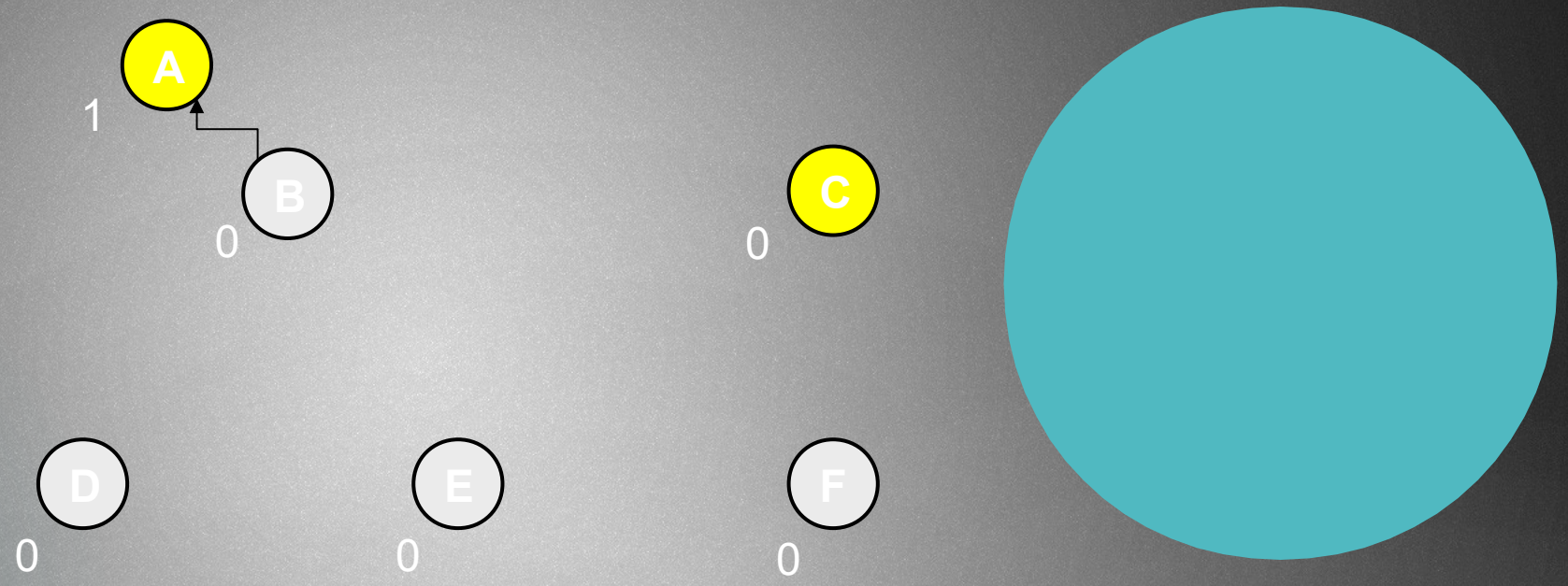
merge(B,C)

First we have to find the representatives in both sets: and merge them together !!!



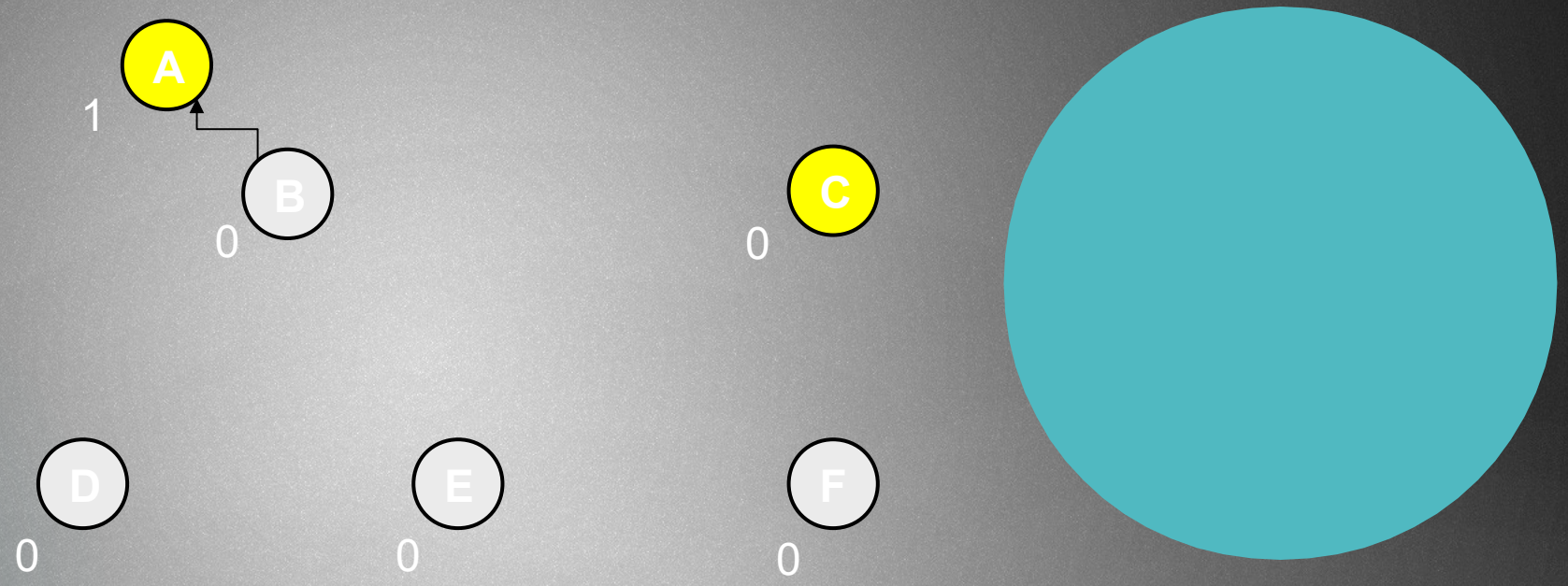
merge(B,C)

First we have to find the representatives in both sets: and merge them together !!!



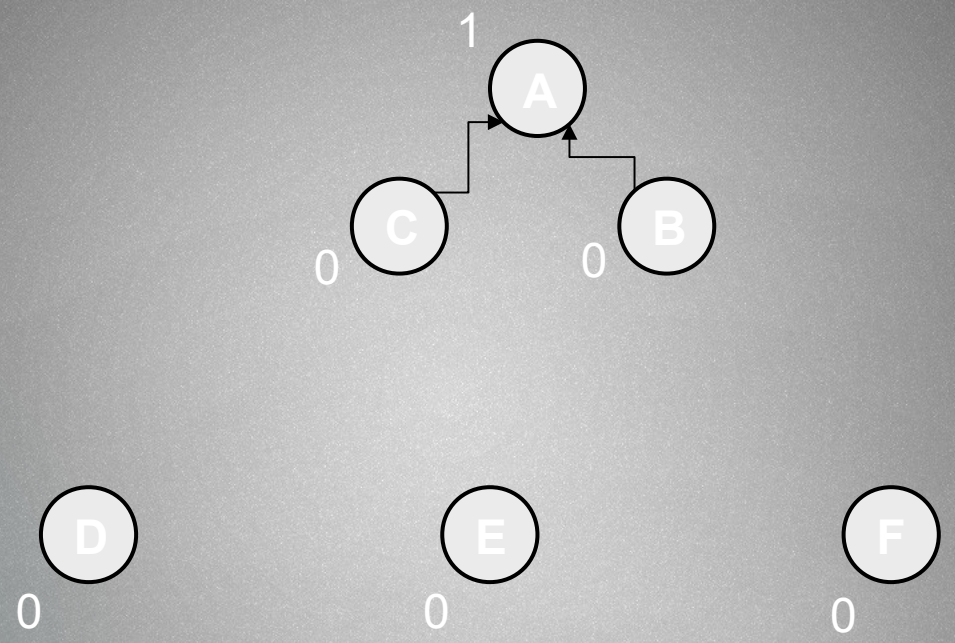
merge(B,C)

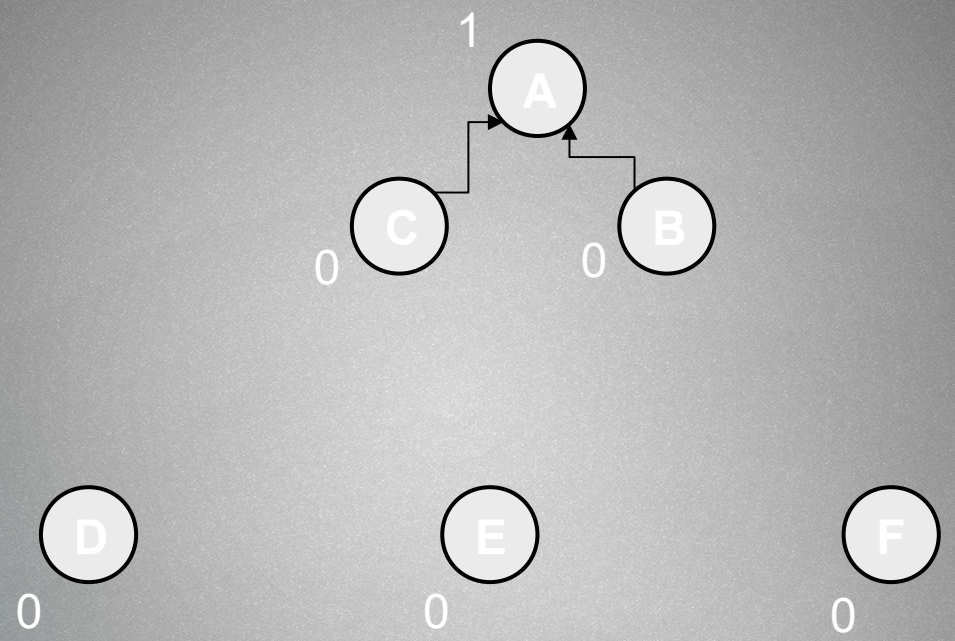
First we have to find the representatives in both sets: and merge them together !!!



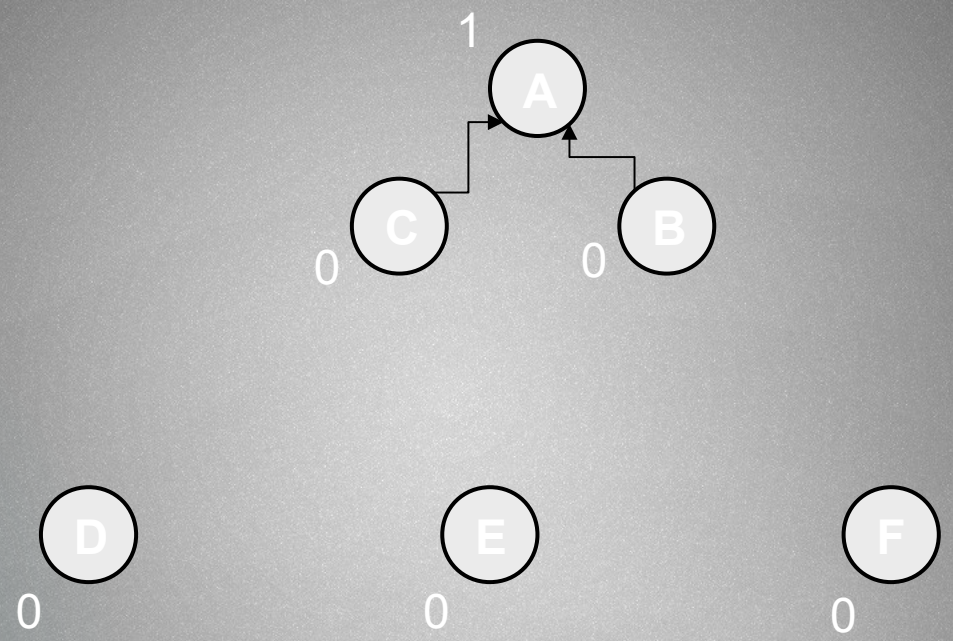
merge(B,C)

First we have to find the representatives in both sets: and merge them together !!!

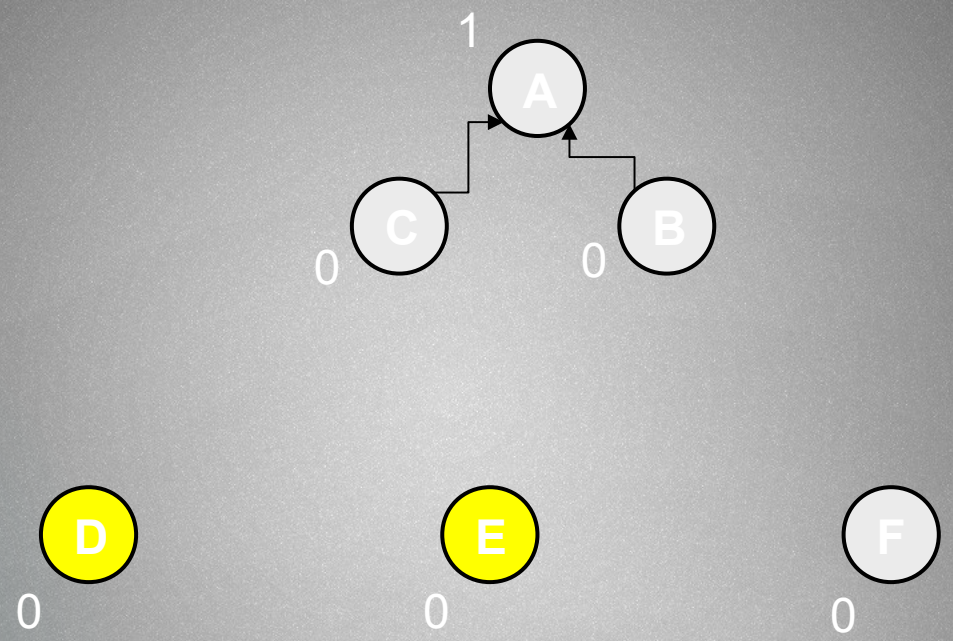


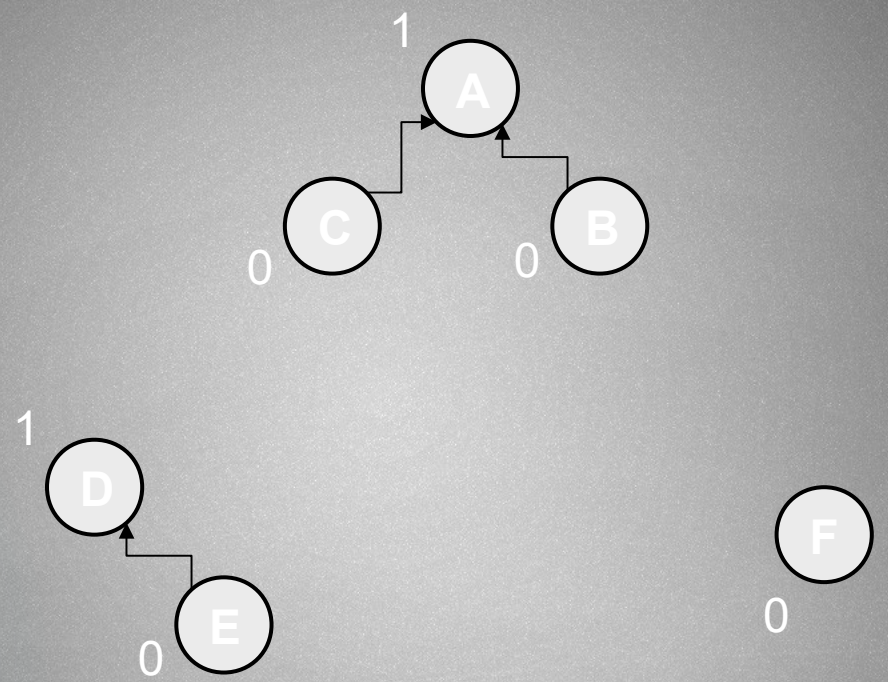

1
A
C
B
0
0
D
E
F
0
0
0

merge(D,E)

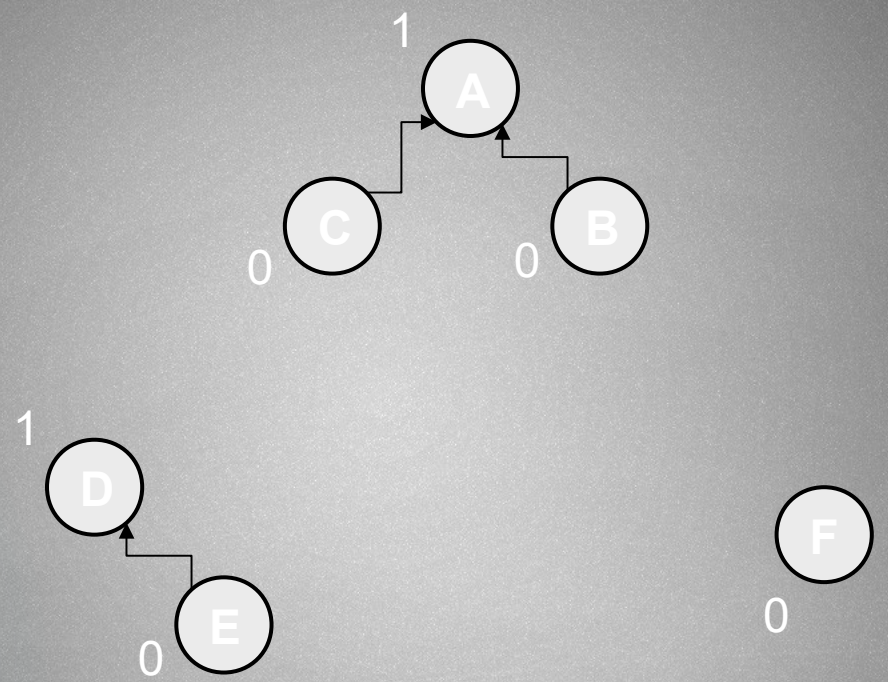


merge(D,E)

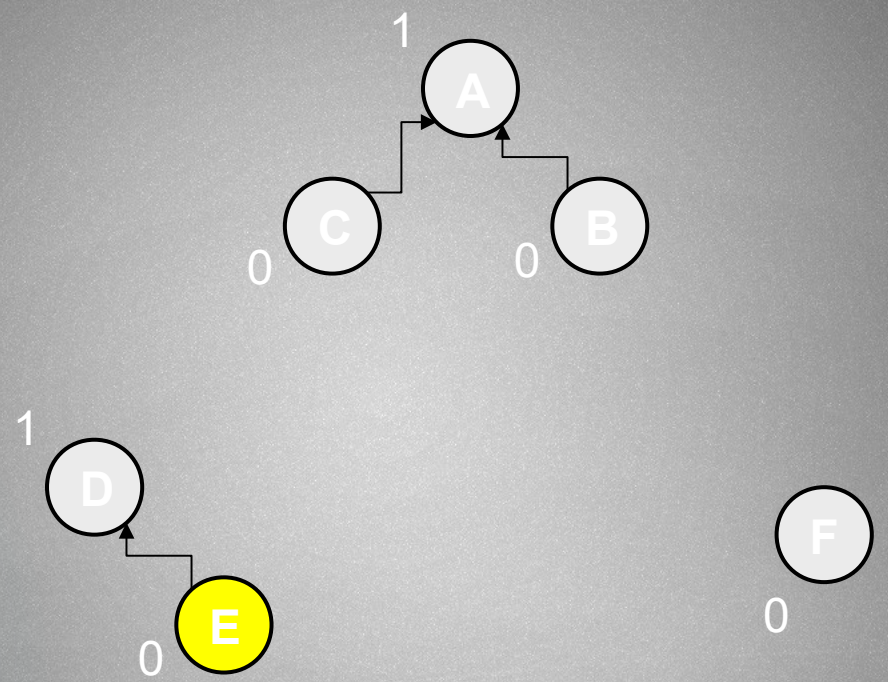



1
A
C
B
0
0
1
D
E
0
F
0

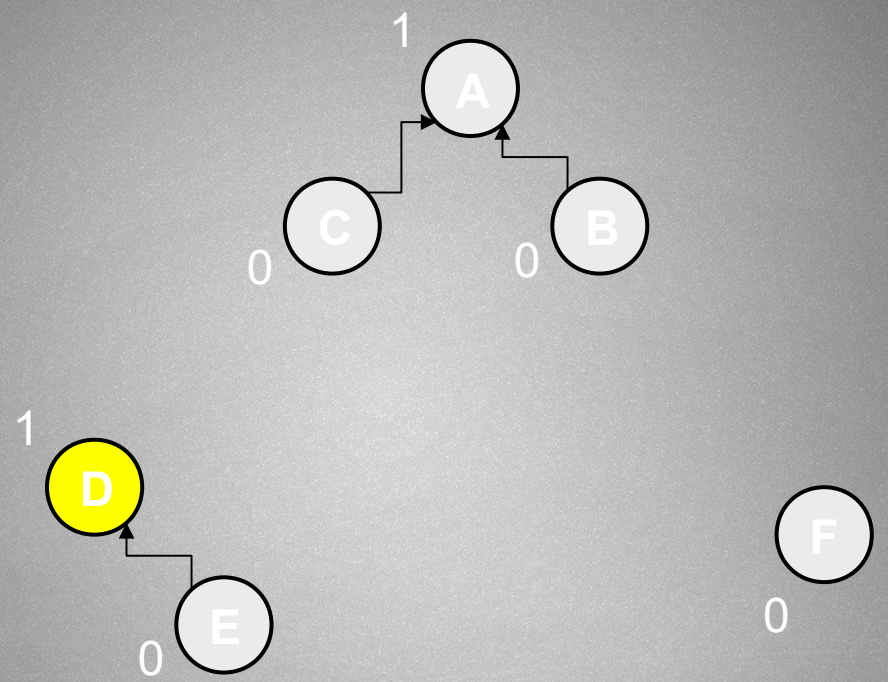
merge(E,C)



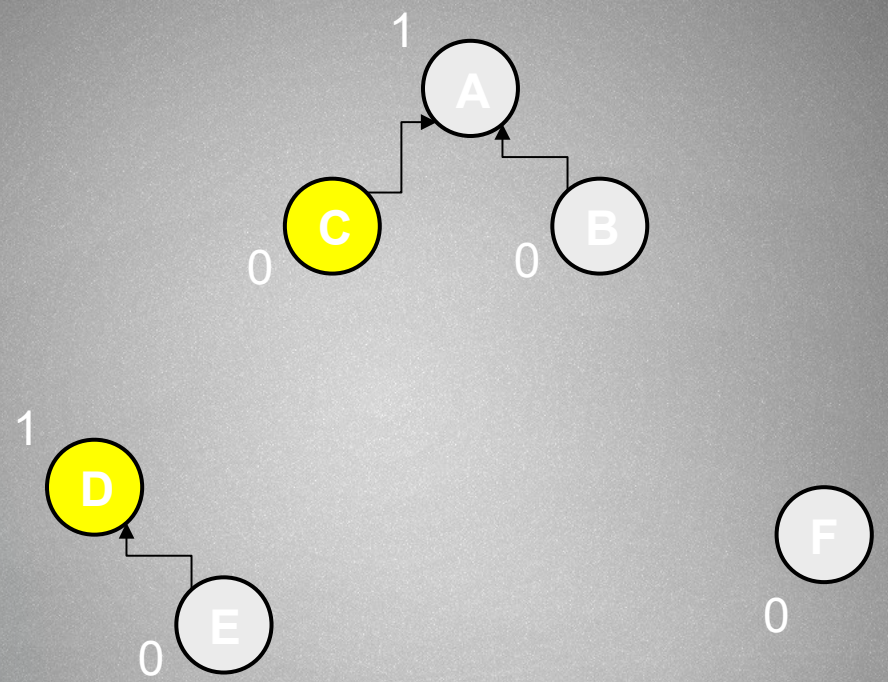
merge(E,C)



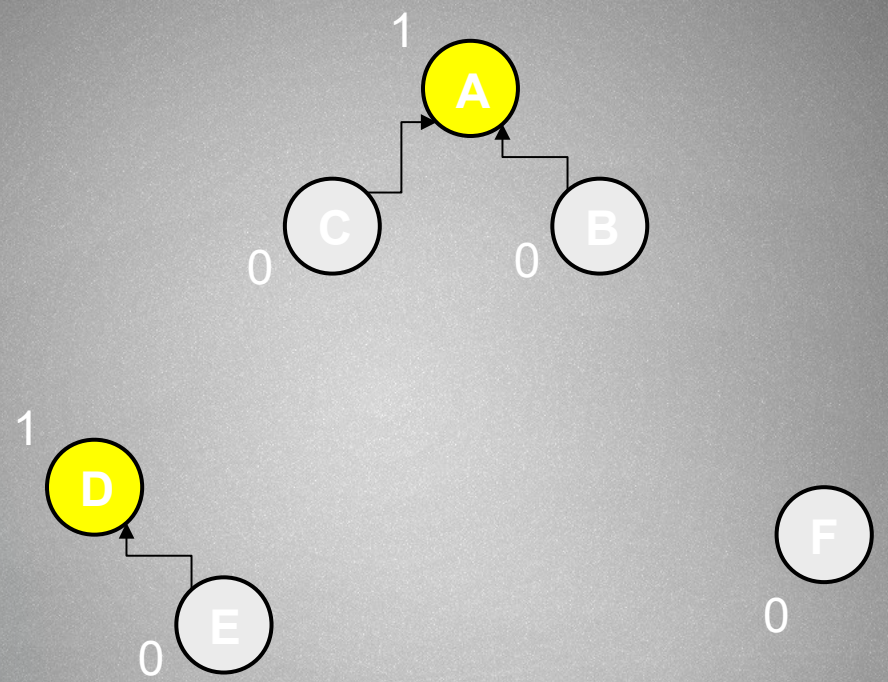
merge(E,C)



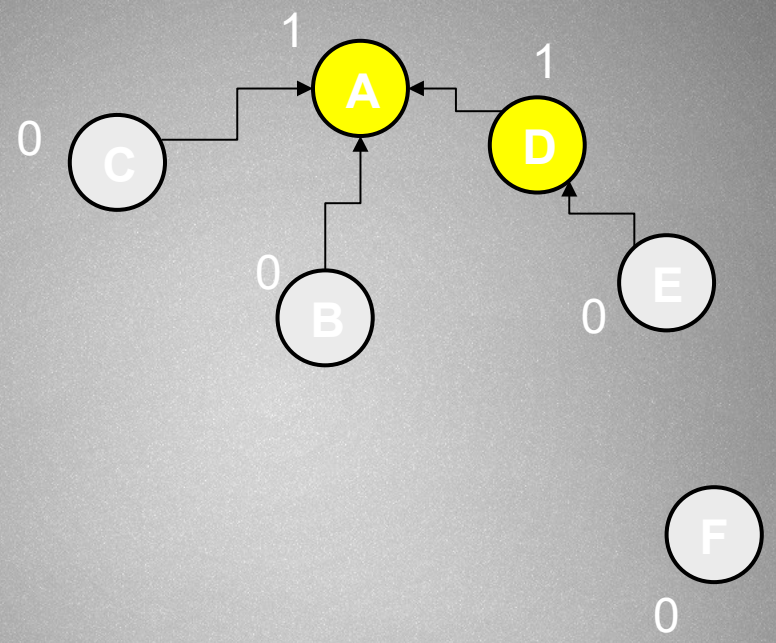
merge(E,C)



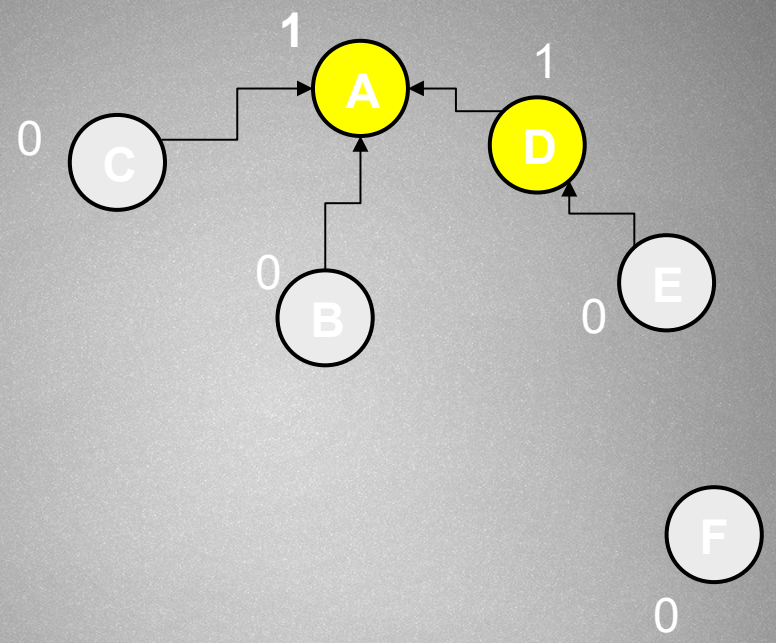
merge(E,C)



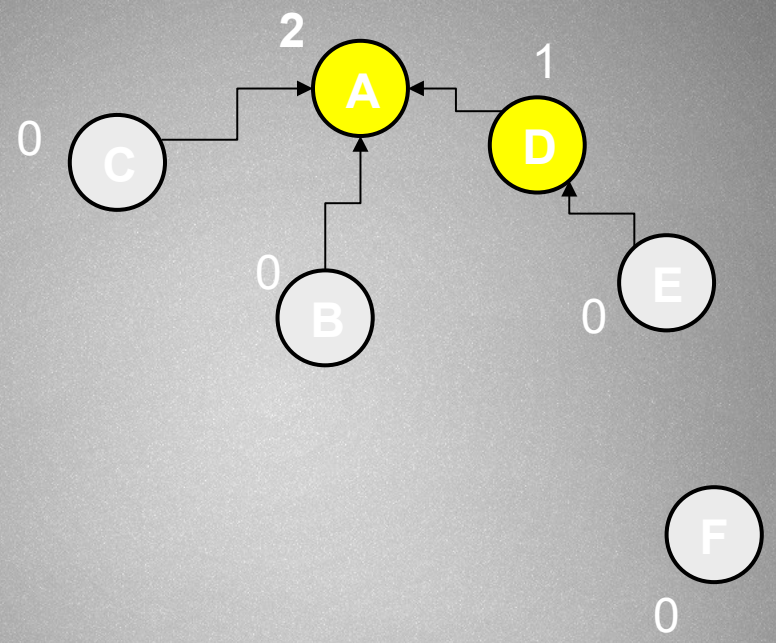
merge(E,C)

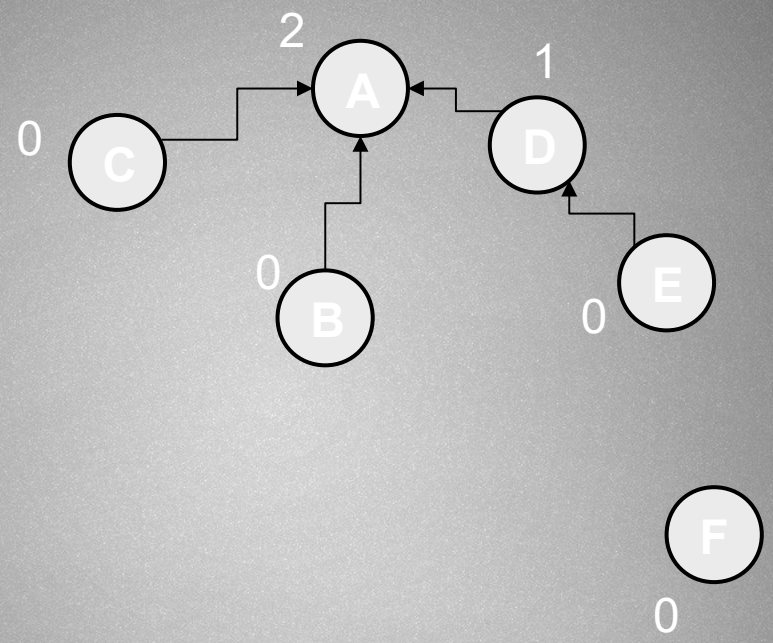


merge(E,C)

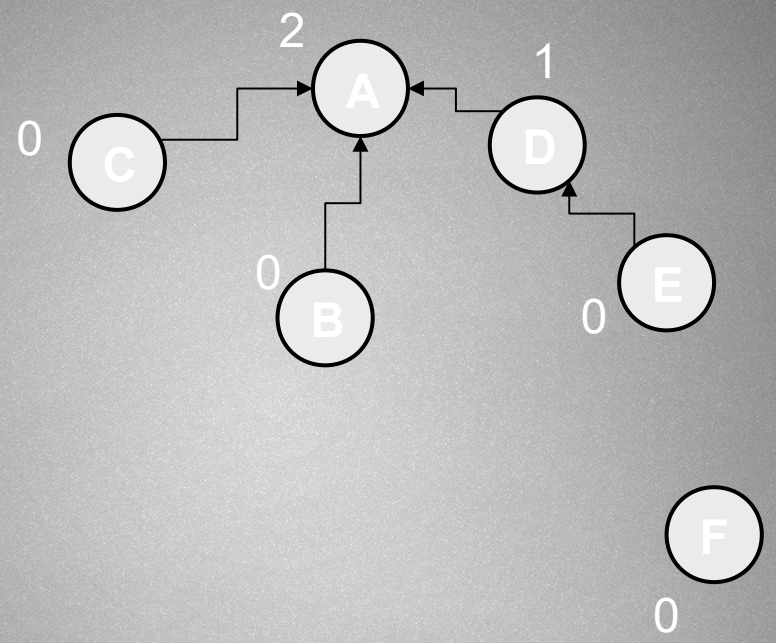


merge(E,C)

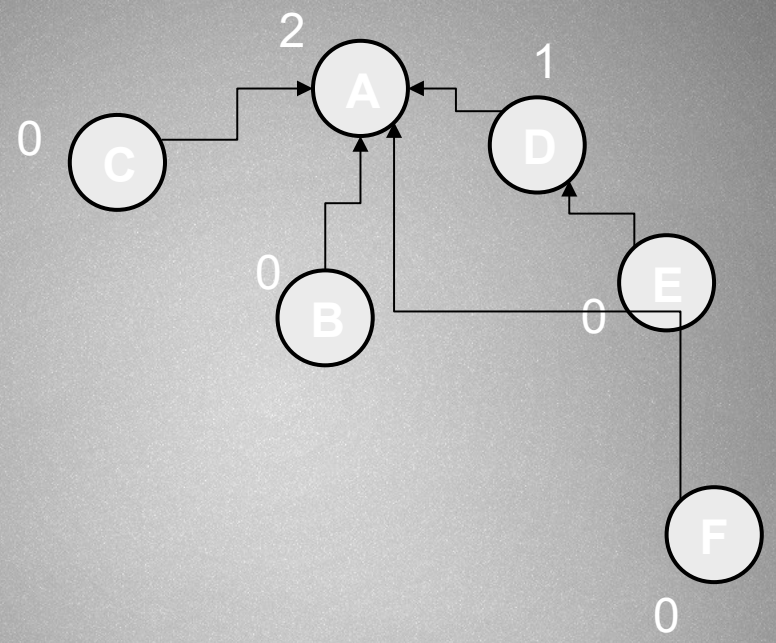



2
A
1
0
C
D
0
B
0
E
F
0

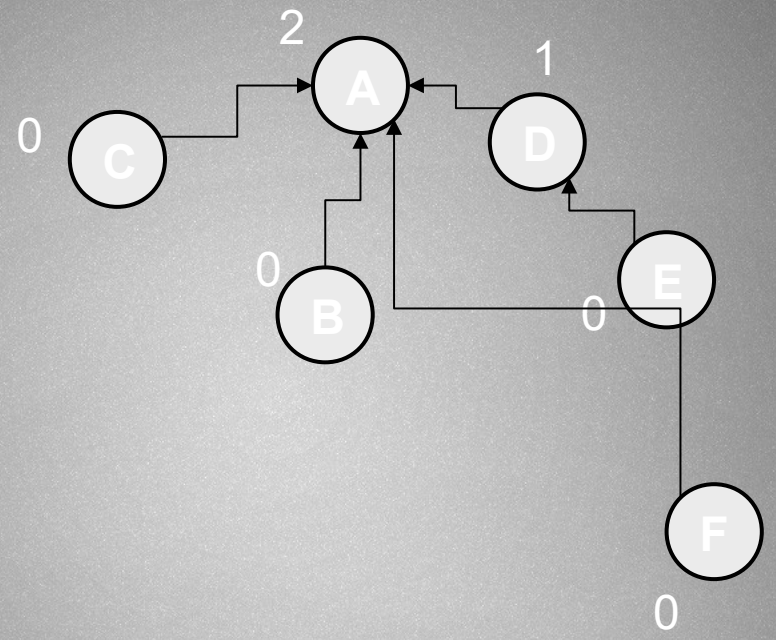
merge(A,F)



merge(A,F)



Because of the path compression → all the nodes will connect to the representative directly. Finding the representative takes **O(1)** for every node !!!



Because of the path compression → all the nodes will connect to the representative directly. Finding the representative takes **O(1)** for every node !!!