**Disjoint-Set Union**

Table of Contents

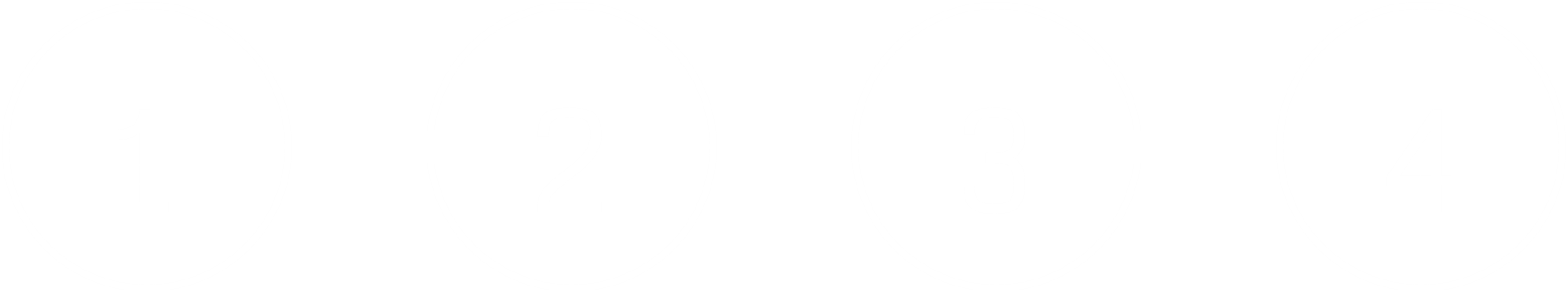
[Operations 4](#_Toc63011344)

[Inefficiencies 5](#_Toc63011345)

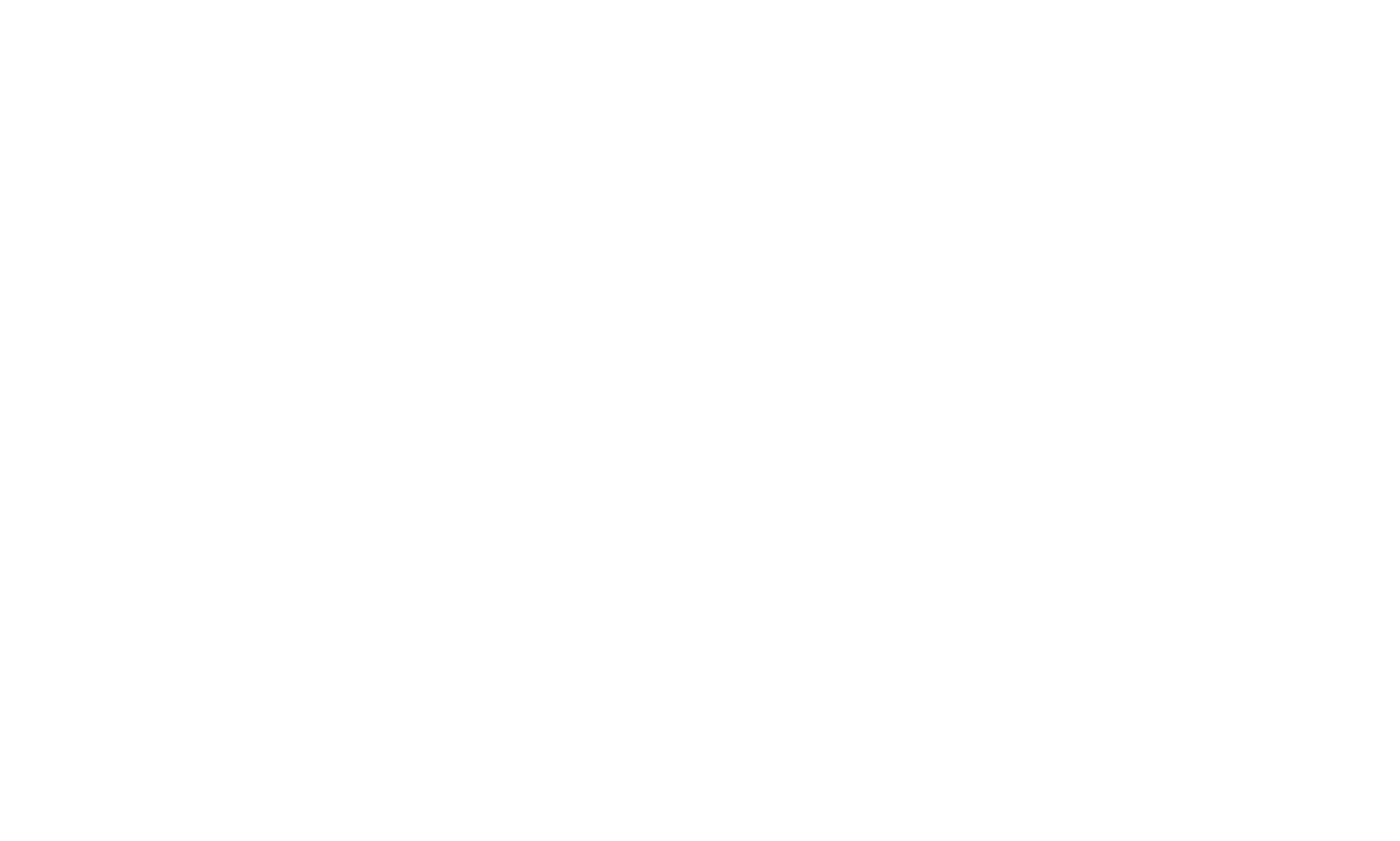
[Applications 12](#_Toc63011346)

The next algorithm we will be learning to solve MSTs is Kruskal’s algorithm. However, it uses the disjoint-set union data structure, so we need to learn that first. We will not be going into details, but will rather be gathering a brief idea of the data structure.

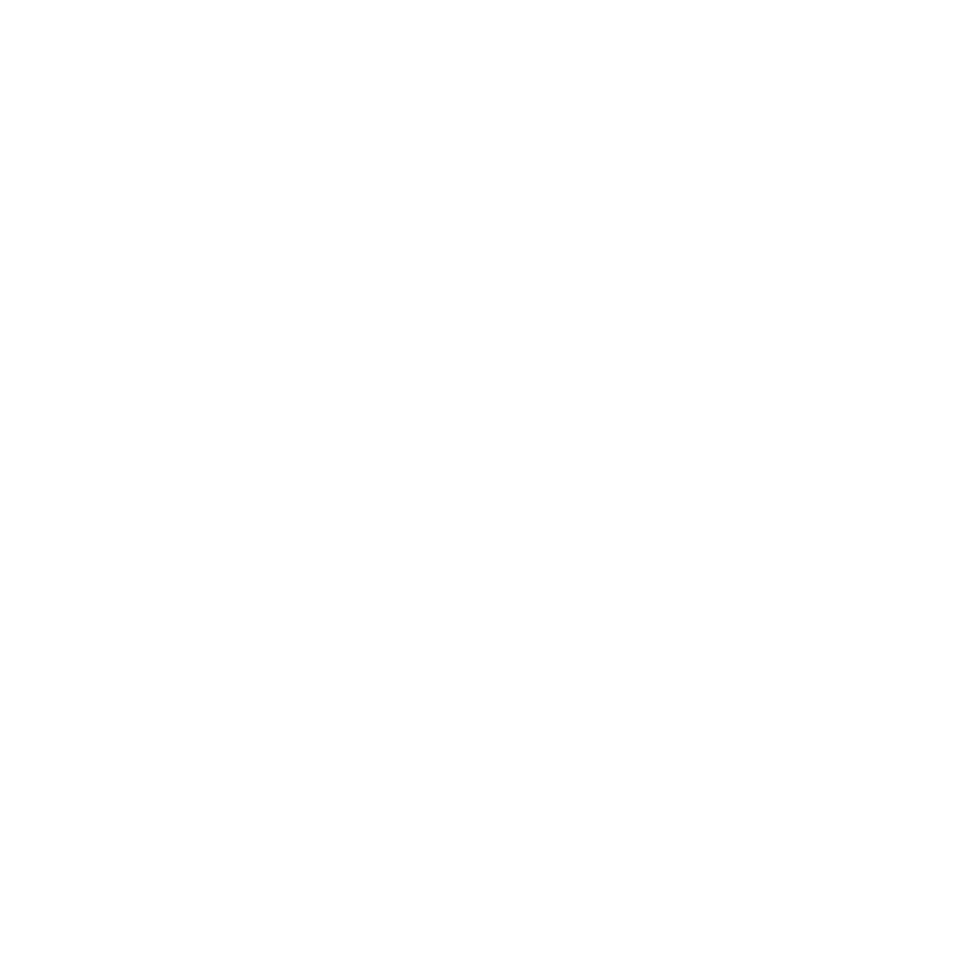
Another name for disjoint-set union (DSU) is union find. Say we have a social network and we want to keep track of groups of friends, i.e. for each user we want to keep track of who is friends with whom.



Say user 1 and 2 become friends. To show this, we draw a directed edge from 2 to 1.

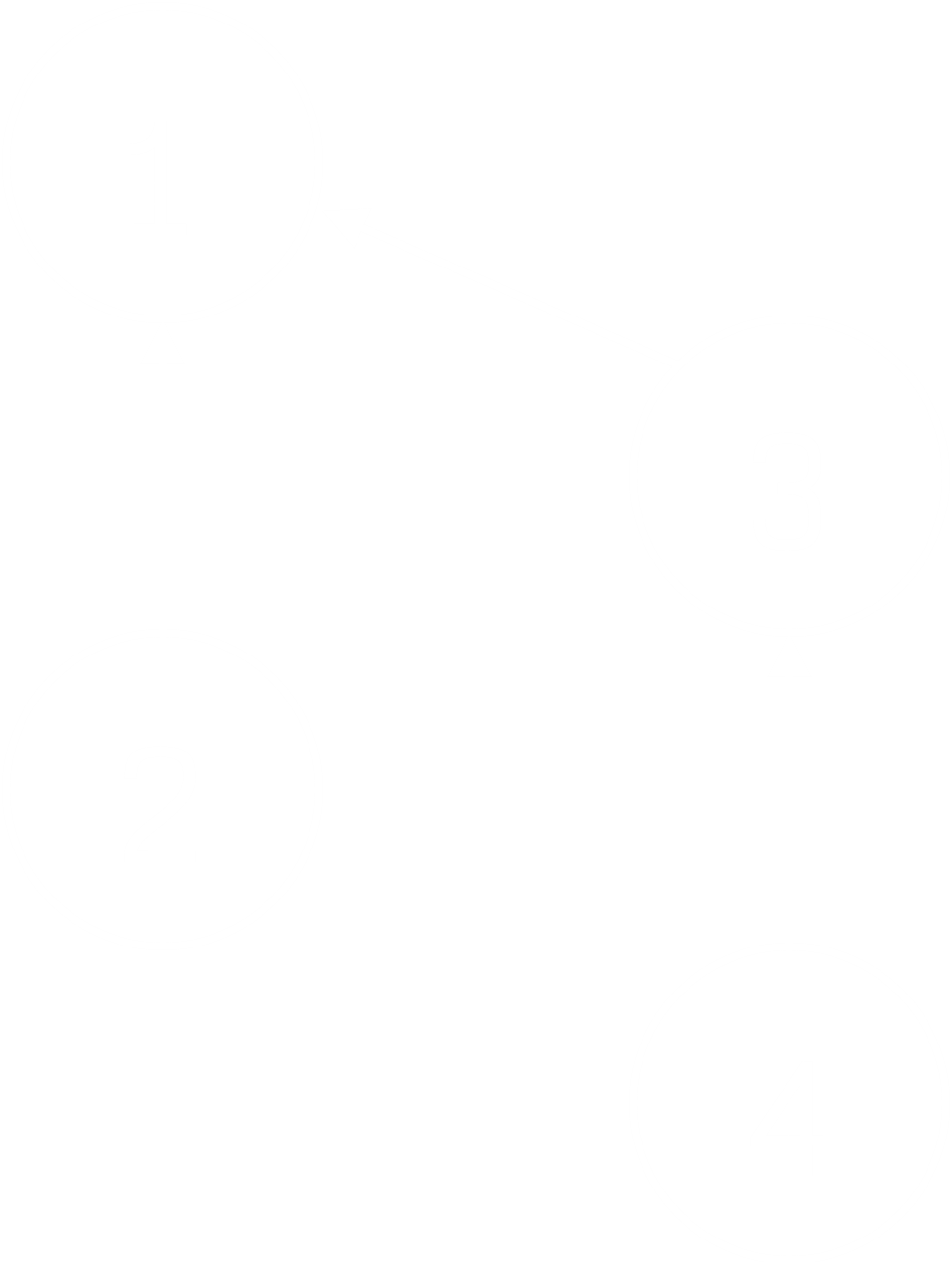


Next, 3 and 4 become friends, so we draw another directed edge.



Let’s say the rule we are following here is that we draw edges from higher values to lower values. For now, the direction does not actually matter. Imagine that we just randomly decided on this rule.

Next, say 3 and 2 become friends.



Notice now that we did not draw an edge from 3 to 2, but rather from 3 to 1. This is where DSU comes in. In DSU, we create sets and assign a representative to each set. What is happening here is that whenever two users become friends, we put them in the same set.

When 1 and 2 became friends, we put them in the same set and assigned 1 as the representative. When users from two different sets become friends, as in the case of 3 and 2, we draw a directed edge from the representative of one set to the representative of another set. This means that if 2 and 4 now become friends, there will be no change to the graph. Initially, it was like all four users were in separate sets, each being the representative for its own set.

The way we keep track of all of this in programming is by having a parent array of size . Thus, for the set where 1 and 2 are friends, parent[2] = 1. Later, when 3 and 2 became friends, we changed parent[3] to 1. The parent of the representative is itself, so parent[1] = 1.

## Operations

There are a few operations we need to cover now. Firstly, creating new users. In DSU, this is done using make\_set(v), where v is the value of the new vertex. This creates a new set and sets the parent of v to itself.

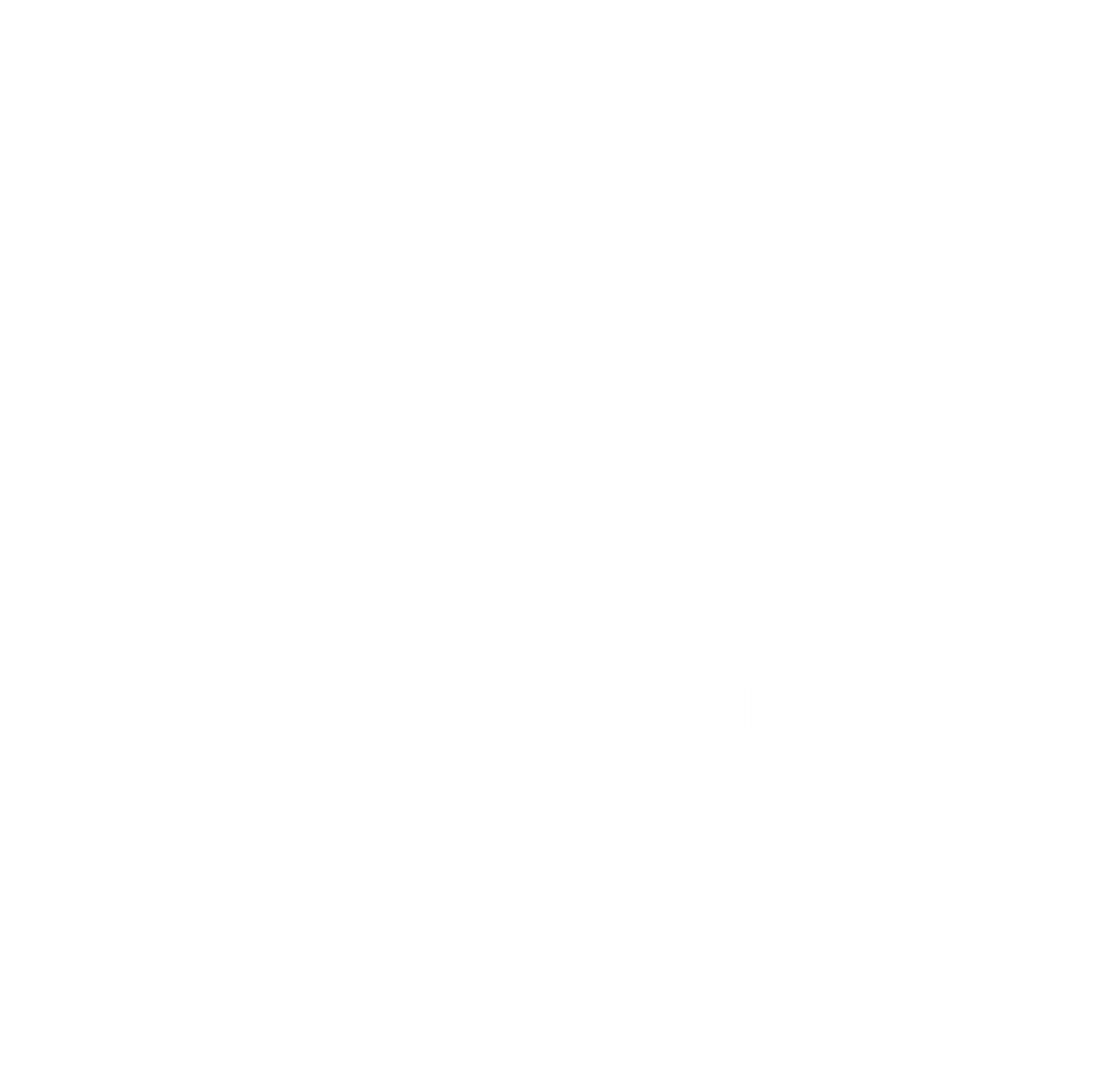
Two users are made friends using union\_set(a, b). The parent of the representative of b is set to the representative of a.

We can find the representative for the set in which a vertex lies using find\_set(v). If we ran find\_set(2) on the example above, we would get 1 in return. This is useful in situations where we want to check if two vertices are from the same set. In the previous example, if 2 and 4 wanted to become friends, we would end up trying to make an edge from 1 to 1, which is not helpful. Thus, we need to use find\_set(v) to check the representatives first. If find\_set(2) != find\_set(4), then they are part of different sets and we can make the edge.

Note that the fact that 2 and 4 cannot have an edge between them does not mean that they cannot become friends. They can become friends, but we will not make an edge between them. This is because we are only trying to keep track of friend groups. The fact that they both have the same representative already tells us that they are from the same friend group.

## Inefficiencies

Say we have elements and we connect them in a chain, with each element being connected to the element just before it.



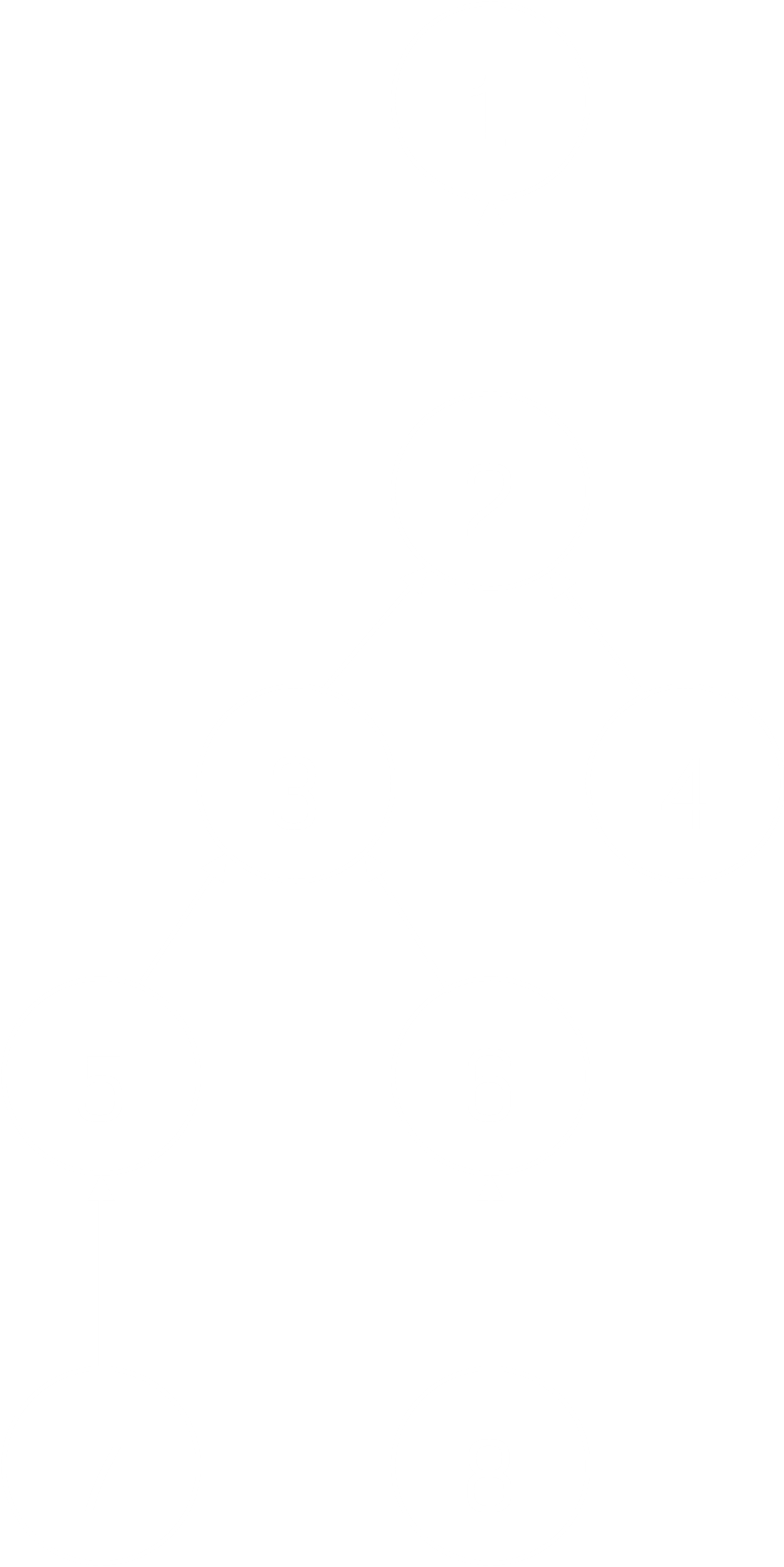
In this tree, every call to the find\_set function will have a time complexity of . We use this function inside the union\_set function as well, which means that will have the same time complexity, since everything else is trivial.

The first optimization we will make is to change the parent of each element directly to its representative. In this way, we will not have to recursively search through the entire tree, but rather just return the parent. To do this, we still have to perform a recursive call, just like before, but this time, we will change the parent on the way back. Think of this like memoizing the values. Any future calls will thus be able to use the memoized values. The code for this could look something like this:

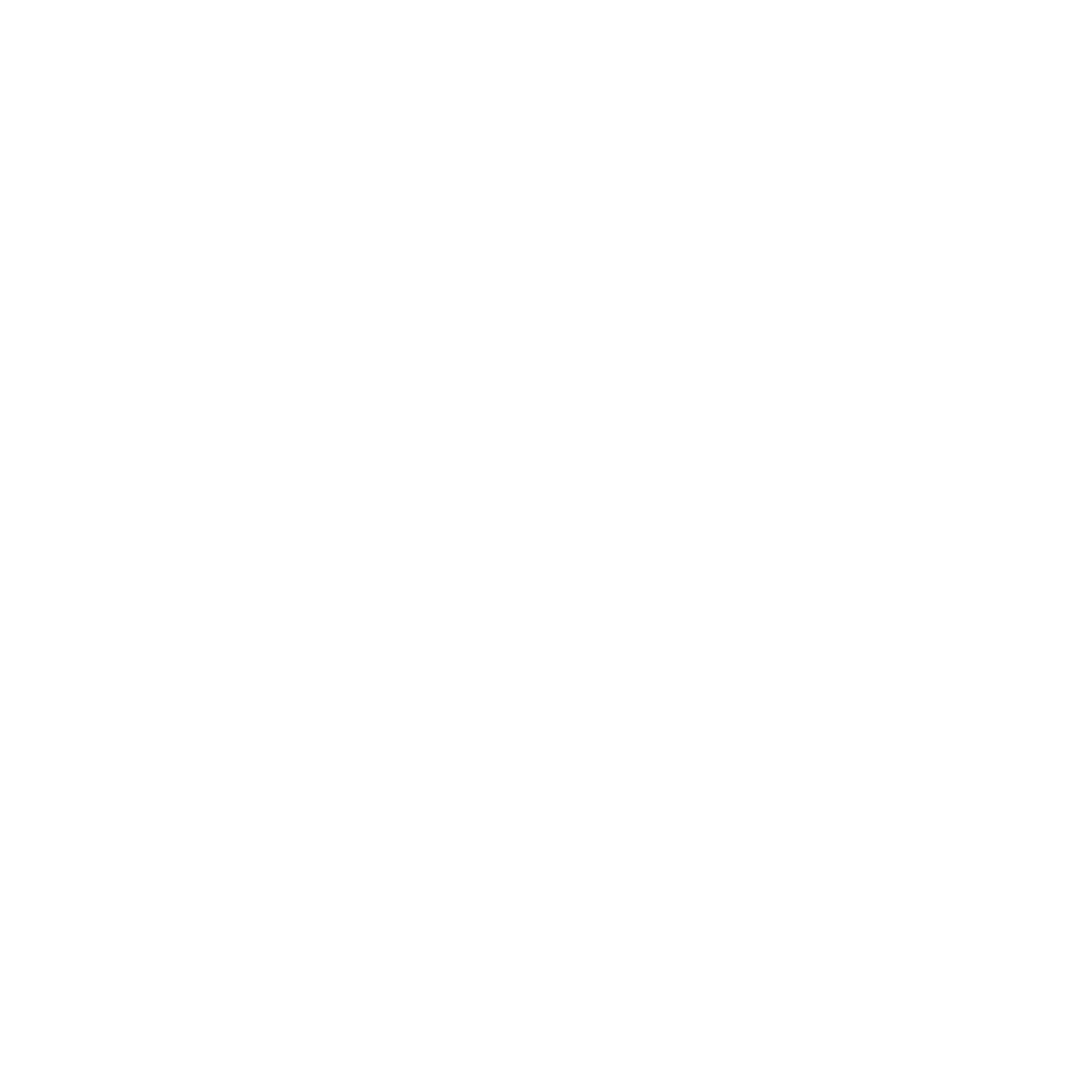
int find\_set (int v)  
{  
 if (parent[v] == v) return v; *// current element is representative* else  
 {  
 if (parent[parent[v]] != parent[v]) parent[v] = find\_set(parent[v]);  
 *// if current parent of element is not a representative,  
 // find the representative and set the parent (memoizing)* return parent[v]; *// return representative* }  
}

C++

To make this clearer, consider the graph below:



If we make a call to find\_set(7), we will find that its current parent is not a representative, so there will be a recursive call to find\_set(5). There, we will find the same thing and perform another recursive call. This will continue all the way up until we find that the parent of 2 is a representative, 1. We will return the representative. This return will be propagated through the tree and every other element that was recursively called will also have their parent set to 1. Any future calls for those elements will then immediately return the value, since they will find that the parent is a representative. The result is that the graph becomes like this:

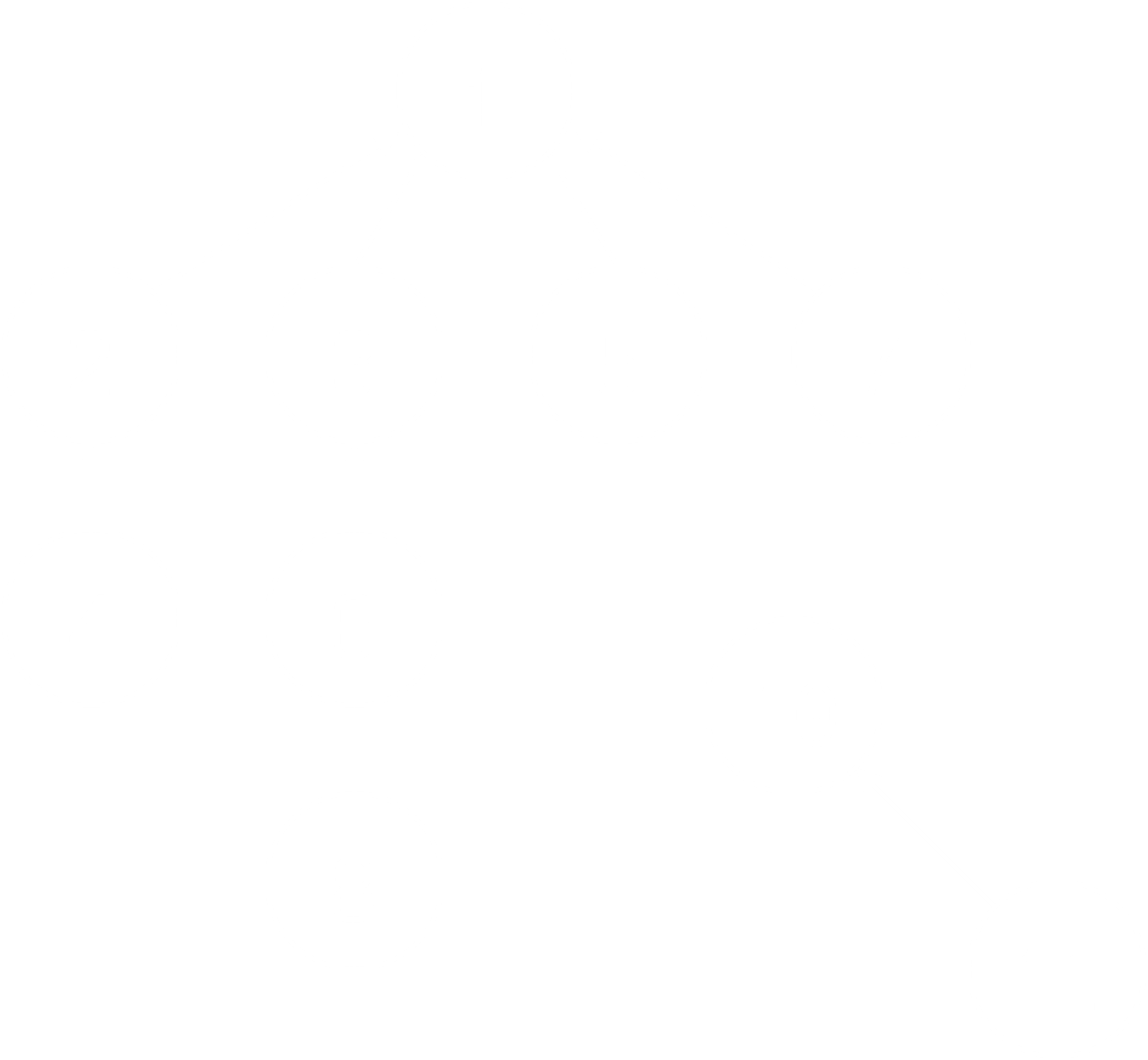


This makes no difference to our ultimate purpose, since every element is still part of the same group. We are not concerned about individual connections, just the group as a whole.

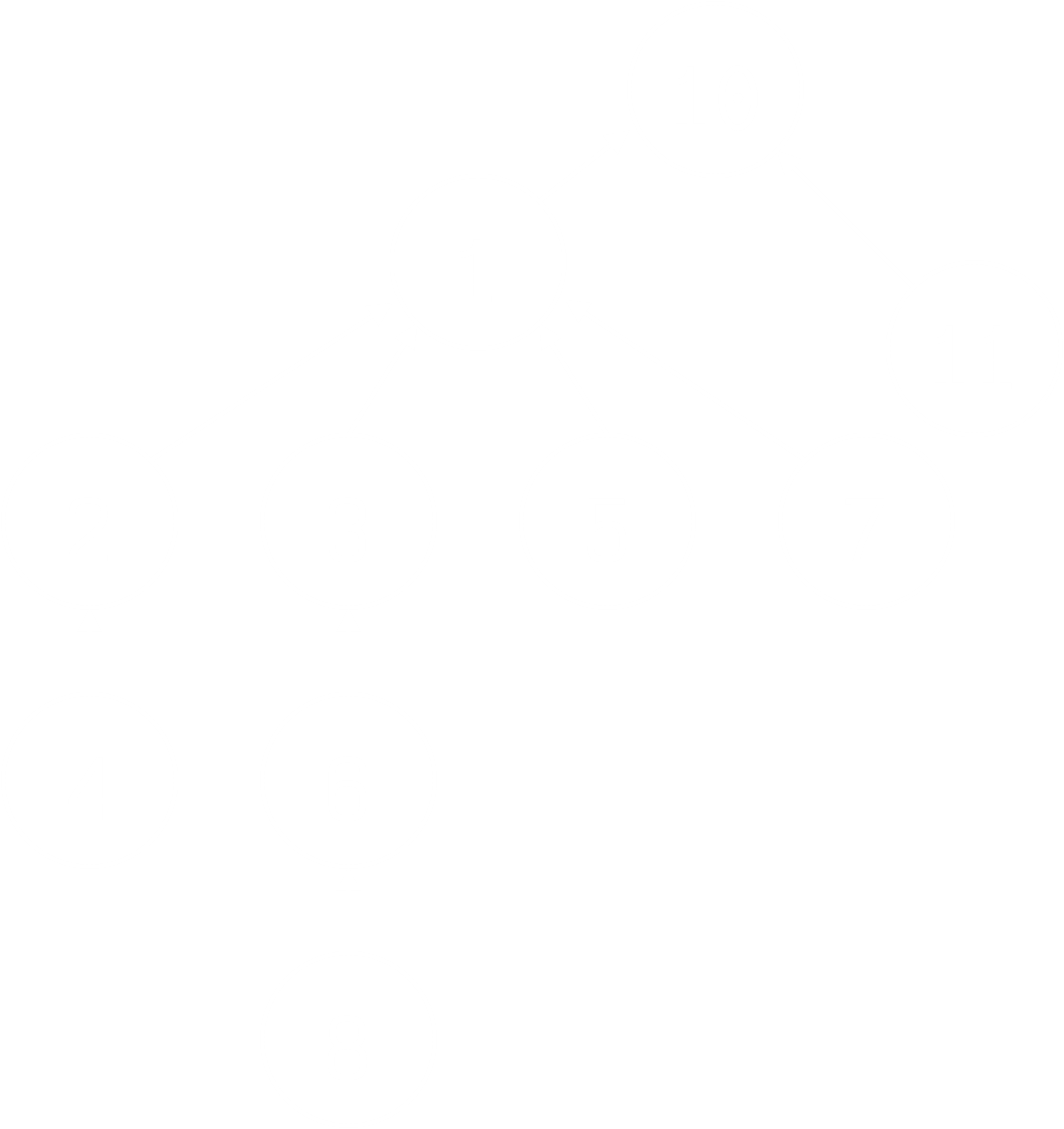
Since the first call still has a time complexity of , but future calls take constant time, on average, the time complexity becomes . We will not get into details of how this works out mathematically.

The second optimization is related to the union\_set function. So far, every time we made a connection, we just said that we would connect the representative of one set with the representative of another. However, the connections are directed, so which way should the arrow point? In the diagrams and examples, an arbitrary rule was followed, we pointed it from the larger element to the smaller one. However, this might not always be the best thing to do.

Consider this tree:



If we perform union\_set(2, 11), we will need to make an edge between 10 and 1. Following the previous arbitrary rule, the edge would be from 10 to 1. However, our main concern should be about the sizes of the trees, since having smaller sizes will help us reduce the time complexity of operations. With this in mind, say we make the edge from the representative with the larger tree to the representative of the smaller tree. This means the edge will go from 1 to 10 in this case. The result is this tree:



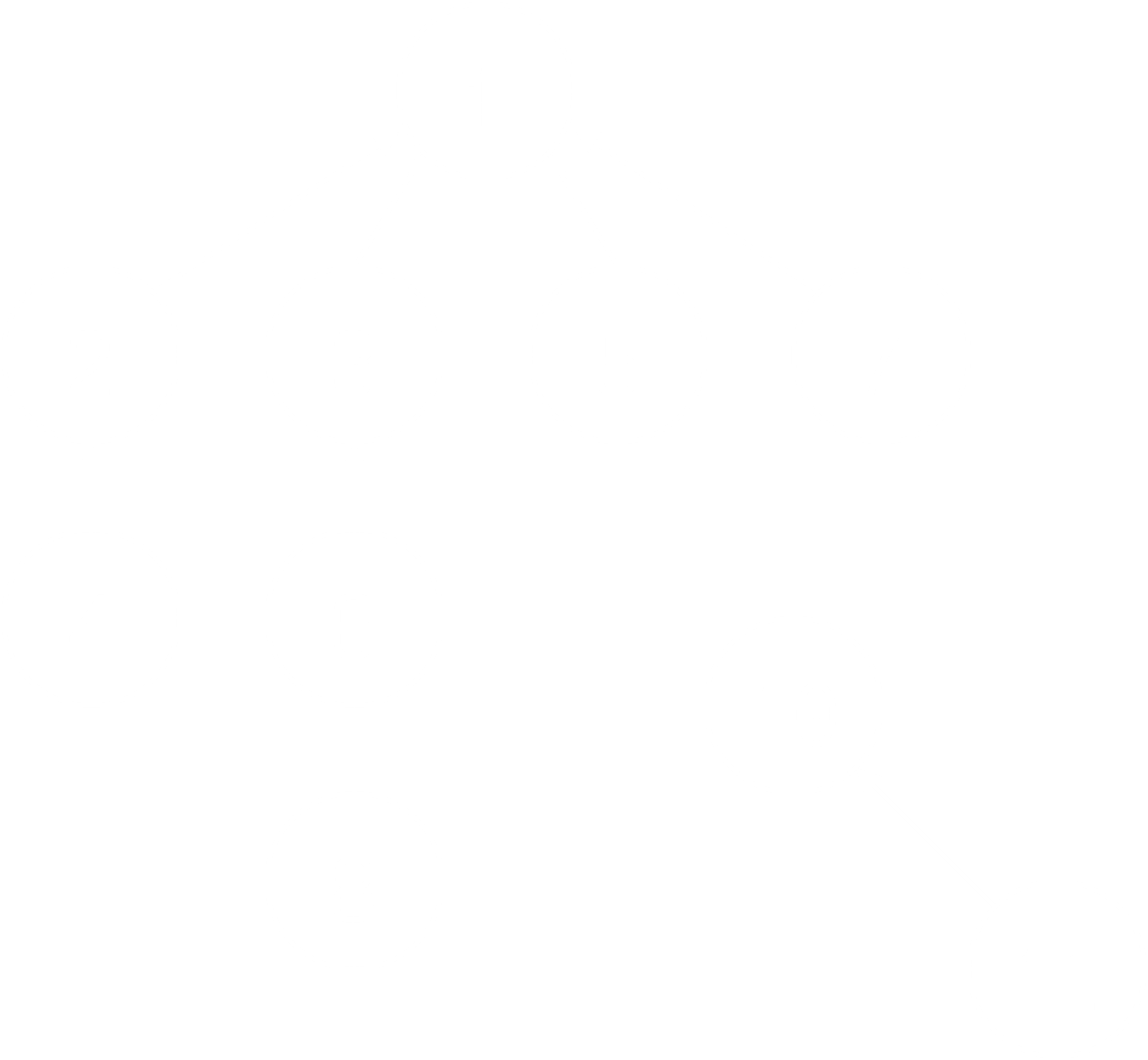
The size of this new tree can be found as size[10] += size[1]. The total size is now . Whenever we do this, we need to change the parent (which is the representative) for every node on the tree that had its parent set to 1. In this case, this means changing different values.

On the other hand, if we had made the edge from the smaller tree to the larger tree, i.e. from 10 to 1, we would end up with this tree:



The size of the resulting tree is still the same, but we only need to update the parent for elements in this case. Thus, this has far less cost. This trick is called Union by Size.

There is another approach to optimizing the union\_set function, which considers the maximum depth of the trees.



We have the same two trees, but this time, we will concentrate on the maximum depth, denoted as the rank of the representative. The rank for 1 is and the rank for 10 is .

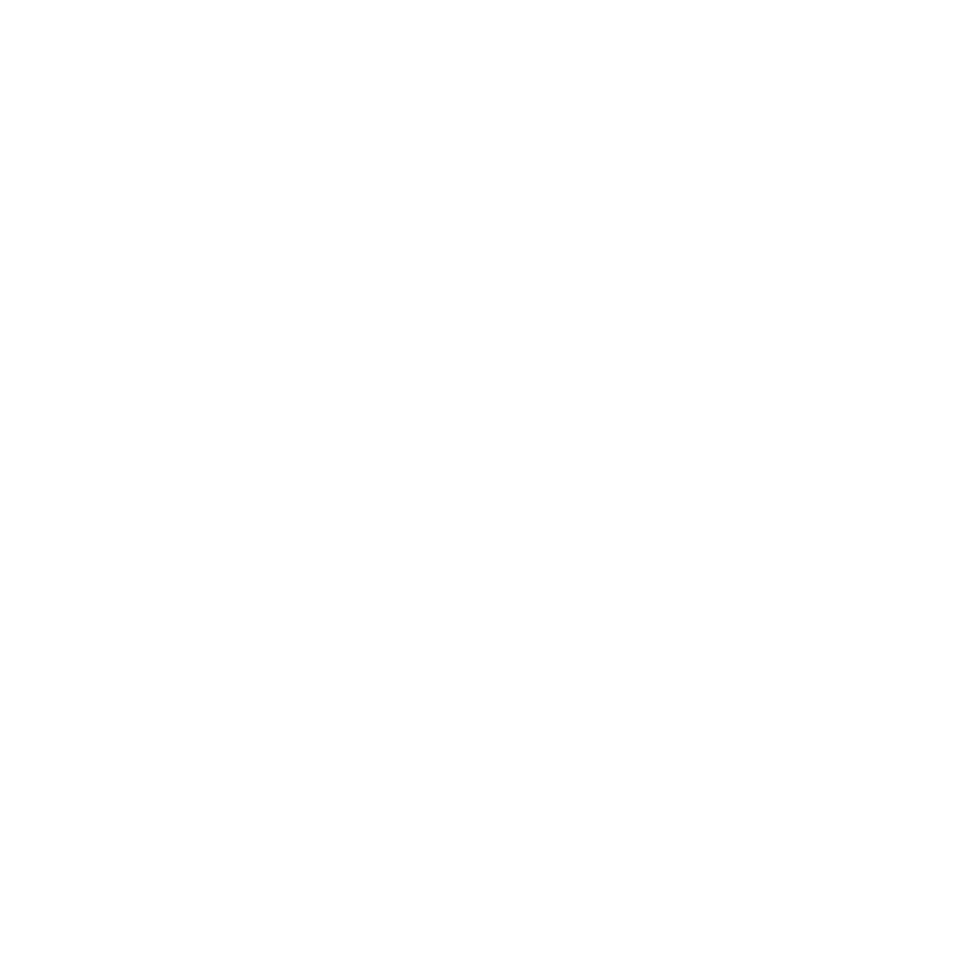
If we make an edge from the representative with the smaller rank to the representative of the larger rank, the resulting tree for our example looks like this:



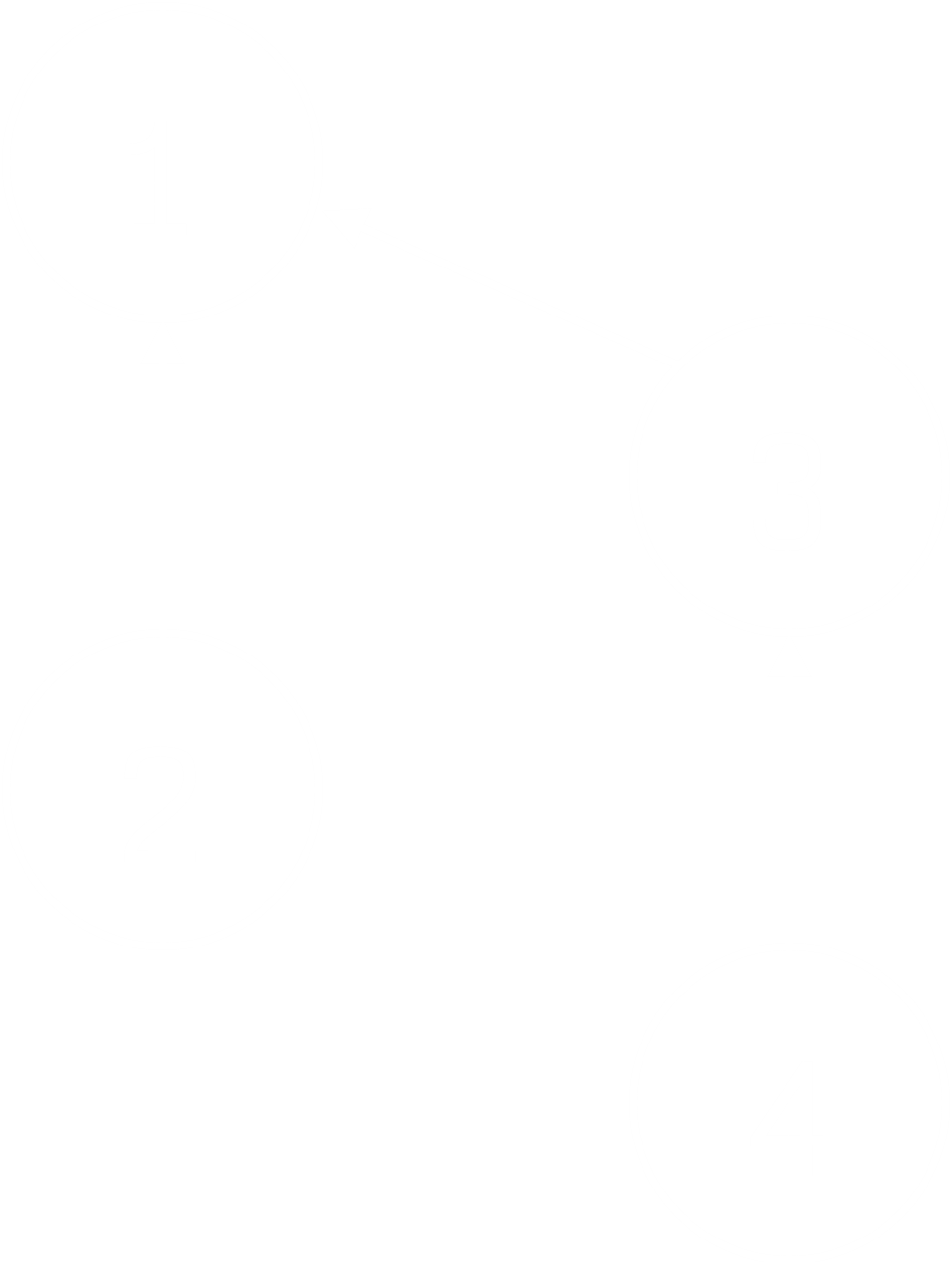
Again, this is the right way to go about things, since we just have to change the representative for elements as opposed to the changes we would have to make the other way around. This method is called Union by Rank.

Notice that the rank for the representative of the resulting tree cannot change. Since we are connecting the smaller ranked branch to the larger ranked branch, the new subtree cannot have a larger depth.

The only time the rank changes is when we connect two trees of the same rank. Since they have the same rank, we can connect either way.



The resulting tree will have a rank that is more than the previous rank, due to the new edge.



The complexities for both approaches are similar on average, which means we can use either approach. The time complexity is . is an Inverse Ackermann function. Such functions grow extremely slowly. This means that the complexity is not constant, but it is very close to being constant. We do not need to know more about this. The space complexity is .

## Applications

Applications for DSU include:

* Connected components in a graph
* Connected components in an image
* Store additional information for each set (e.g. size in Union by Size)
* Arpa’s trick (Offline RMQ)