Saurav Prateek's

Disjoint Sets

Data Structure

How does it achieve **Superlinear** runtime complexity?

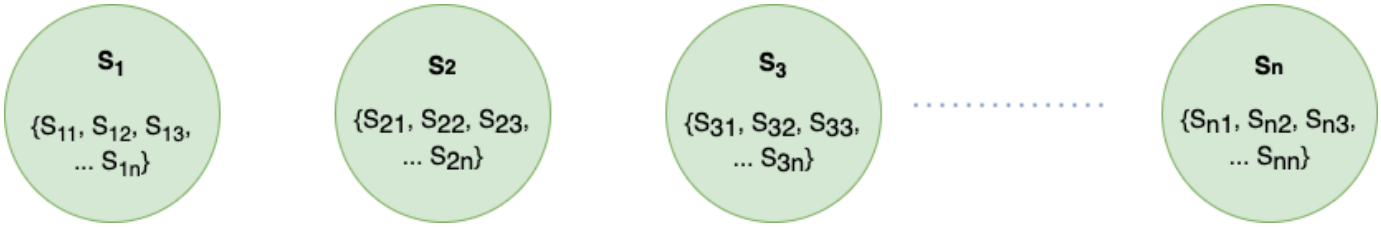
Check out my entire collection of handbooks here:
sauravprateek.me/swe-handbooks

Table of Contents

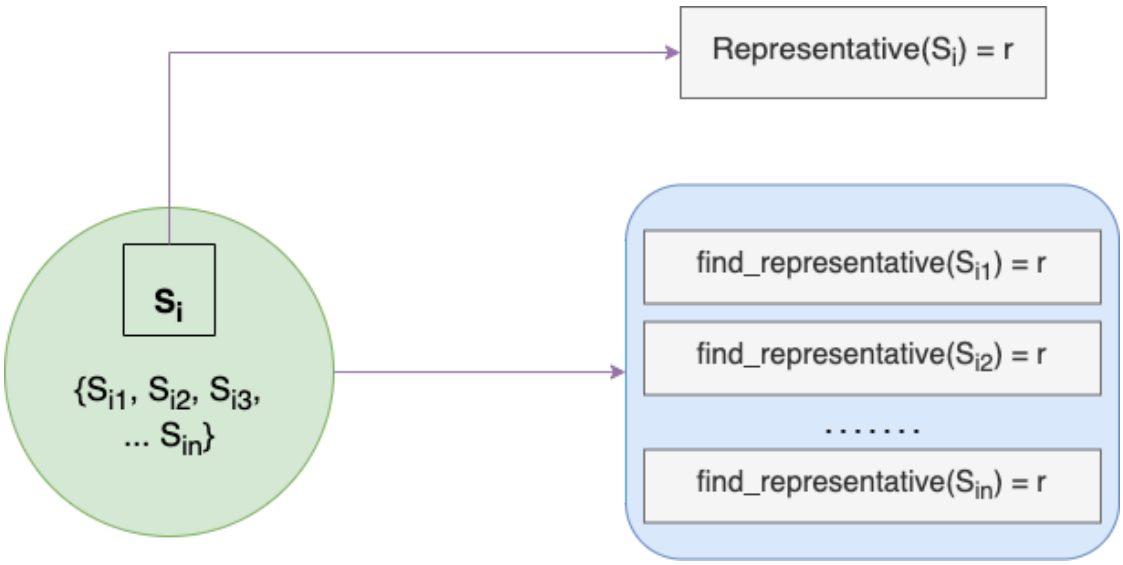
Introduction

A **Disjoint Set** data structure can be called a collection of **Sets** which are **disjoint** in nature. By the term disjoint, we mean if an element is a part of a set then it can never be a part of any other sets in the collection.

$$\text{Disjoint Set} = \{S_1, S_2, S_3, \ldots., S_n\}$$



Every set has a **Representative**. The representative of a set is the one element which represents the entire set. Hence, if we ask for the representative from any element in the set, they all should be able to return the one correct representative.



In the above image we can see a Set $\{S_i\}$ having **n** elements $\{S_{i1}, S_{i2}, S_{i3}, ..., S_{in}\}$ with a representative element, say **r**. All the elements of the set are able to return the representative of their set.

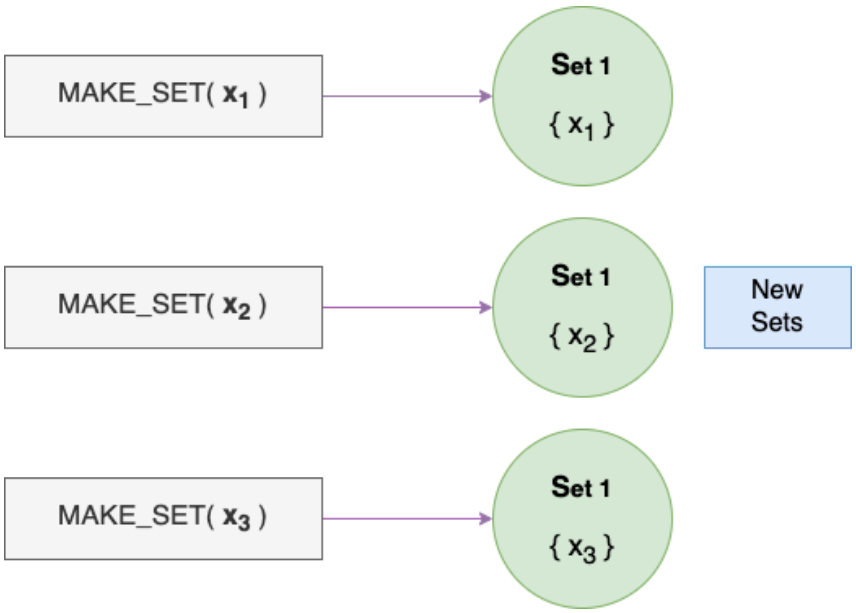
$$\begin{aligned} \texttt{find\_representative}(S_{i1}) &= r \\ \texttt{find\_representative}(S_{i2}) &= r \\ &\dots \\ \texttt{find\_representative}(S_{in}) &= r \end{aligned}$$

Operations supported by Disjoint Set

A **Disjoint Set** data structure is capable of supporting the following operations:

MAKE_SET(X)

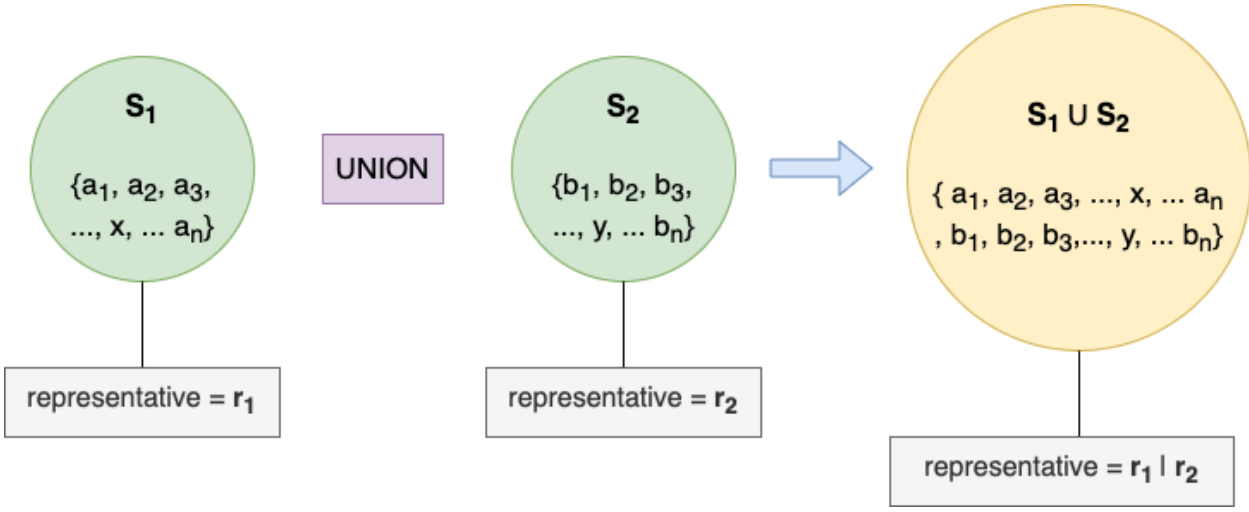
The operation creates a new set whose only member (and representative) is **x**. This is the initial operation during the creation of the **Disjoint Set** data structure.



UNION(X, Y)

The operation combines the two sets which contain the element **x** and **y** respectively, given that they are disjoint. By disjoint we mean that none of the elements of $\mathbf{S_1}$ ($x \in S_1$) should be a part of $\mathbf{S_2}$ ($y \in S_2$) and vice-versa.

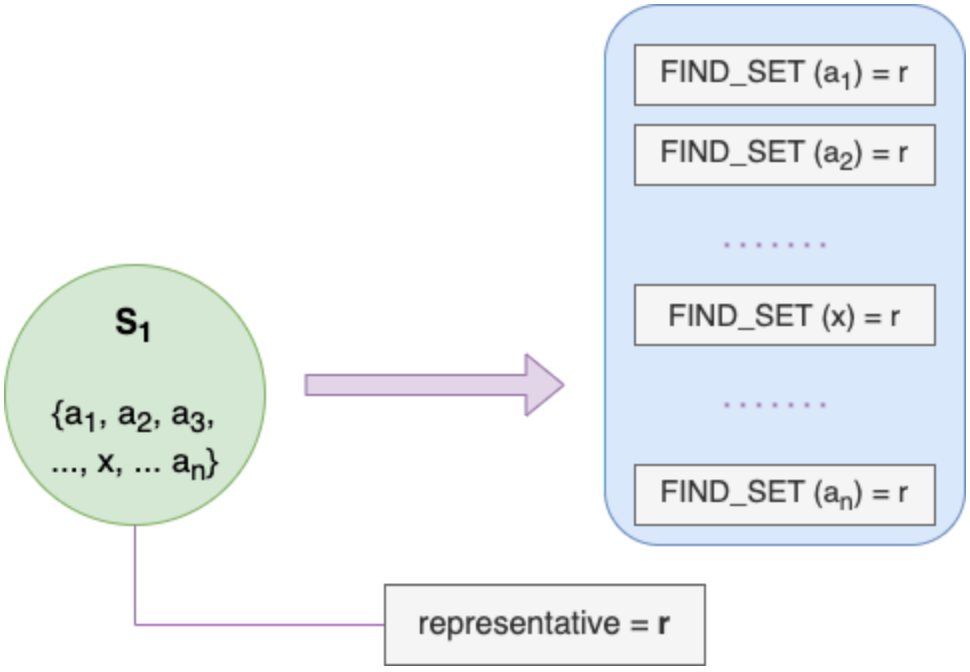
If the two sets are disjoint, then one of the representatives of the two sets is made the representative of the union set.



FIND_SET(X)

The operation returns the representative of the Set that contains element **x**.

$$\texttt{FIND\_SET}(S_1(x \in S_1)) = r$$



In this handbook we will focus on how we can implement a **Disjoint Set** data structure that can perform the above series of operations in a **Super-Linear** run-time.

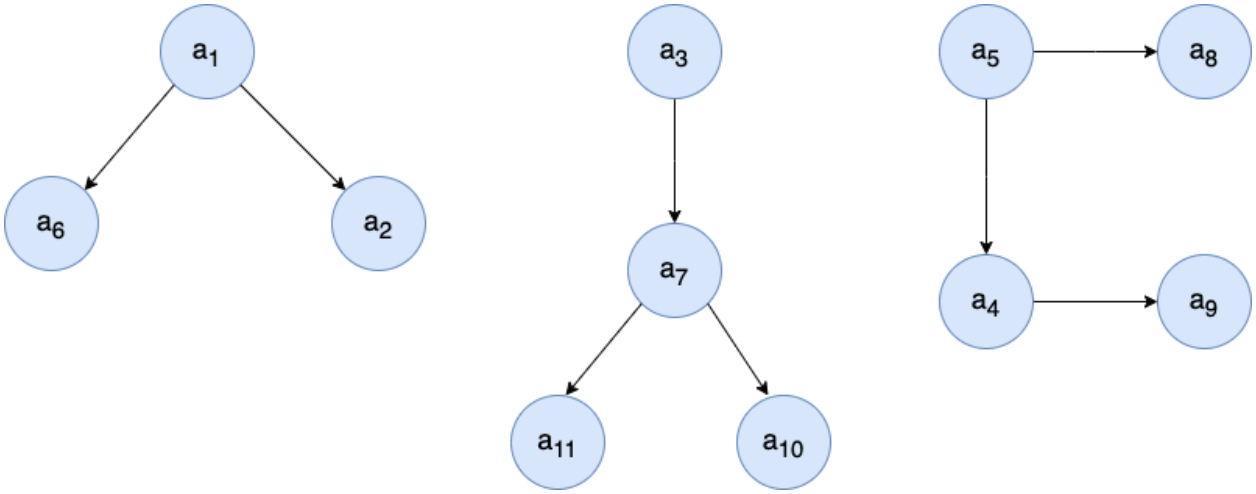
Connected Components

One of the major applications of a **Disjoint Set** data structure is to determine whether two elements belong to the same component or if two elements are connected to each other or not.

```
IS_CONNECTED(x, y) = True | False
```

Let's take an example of an undirected graph **G** where **G.V** is the set of vertices and **G.E** is the set of edges.

The graph looks like this.



Now let's add the edges to the nodes in the above graph one-by-one.

Adding Edges	Nodes
	a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_{10}, a_{11}
(a_1 - a_2)	{a_1, a_2}, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_{10}, a_{11}
(a_1 - a_6)	{a_1, a_2, a_6}, a_3, a_4, a_5, a_7, a_8, a_9, a_{10}, a_{11}
(a_3 - a_7)	{a_1, a_2, a_6}, {a_3, a_7}, a_4, a_5, a_8, a_9, a_{10}, a_{11}
(a_7 - a_{11})	{a_1, a_2, a_6}, {a_3, a_7, a_{11}}, a_4, a_5, a_8, a_9, a_{10}
(a_7 - a_{10})	{a_1, a_2, a_6}, {a_3, a_7, a_{11}, a_{10}}, a_4, a_5, a_8, a_9
(a_5 - a_8)	{a_1, a_2, a_6}, {a_3, a_7, a_{11}, a_{10}}, a_4, {a_5, a_8}, a_9
(a_5 - a_4)	{a_1, a_2, a_6}, {a_3, a_7, a_{11}, a_{10}}, {a_4, a_5, a_8}, a_9
(a_4 - a_9)	{a_1, a_2, a_6}, {a_3, a_7, a_{11}, a_{10}}, {a_4, a_5, a_8, a_9}

In the above procedure we started from the position where there were no edges in the graph and every node was an individual disjoint component/set. This is similar to the **MAKE_SET(x)** method of the **Disjoint Set** discussed earlier. Gradually we

started introducing edges one-by-one and started adding nodes together. This process is similar to the **UNION(x, y)** method of the disjoint set discussed earlier.

We can lay out the algorithm of the entire approach somewhat like this.

Step 1: Introducing the individual Nodes in the graph

In this step we introduce the nodes into the graph.

```
INITIALIZE_NODE (G.V):
    FOR each vertex V ∈ G.V:
        MAKE_SET(V)
```

Step 2: Adding the Edges to the previously added nodes

In this step we add the edges to the Graph one by one.

```
CONNECT_NODES (G.E):
    FOR each edge (u, v) ∈ G.E:
        UNION(u, v)
```

Now, since we have performed the initial steps through the operations discussed in Disjoint Set, we can finally check if two nodes belong to some component or root.

Let **u** and **v** be two nodes of Graph **G**.

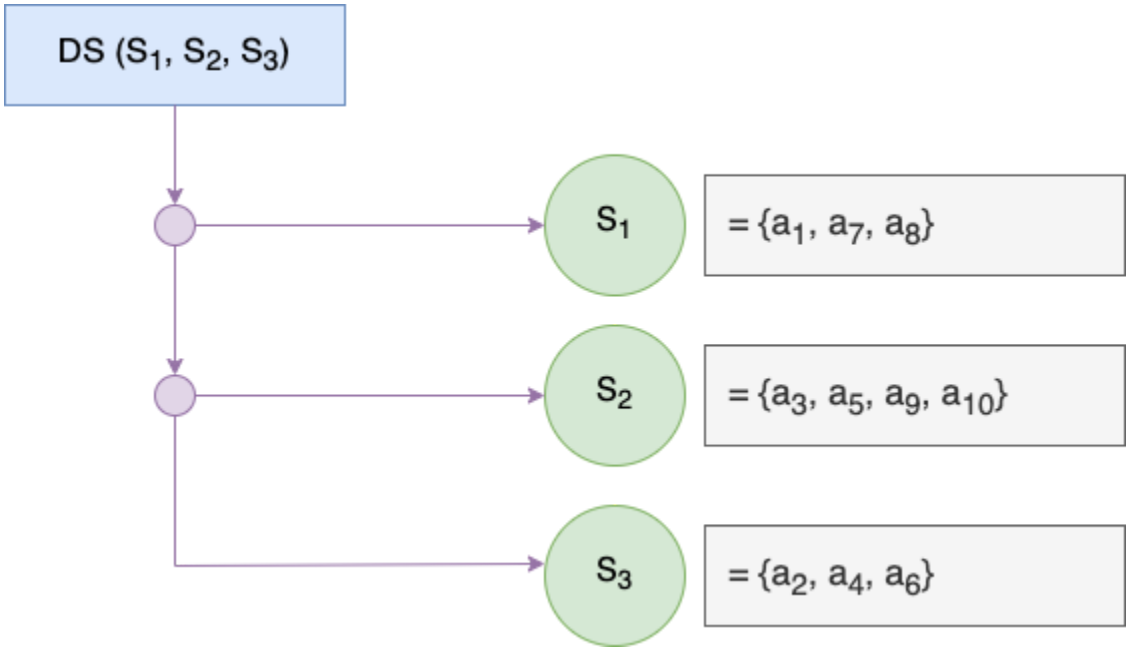
```
IS_CONNECTED (u, v):
    r1 <- FIND_SET(u)
    r2 <- FIND_SET(v)

    return r1 == r2
```

Two nodes are considered to be from the same component, if they both have the same set representative.

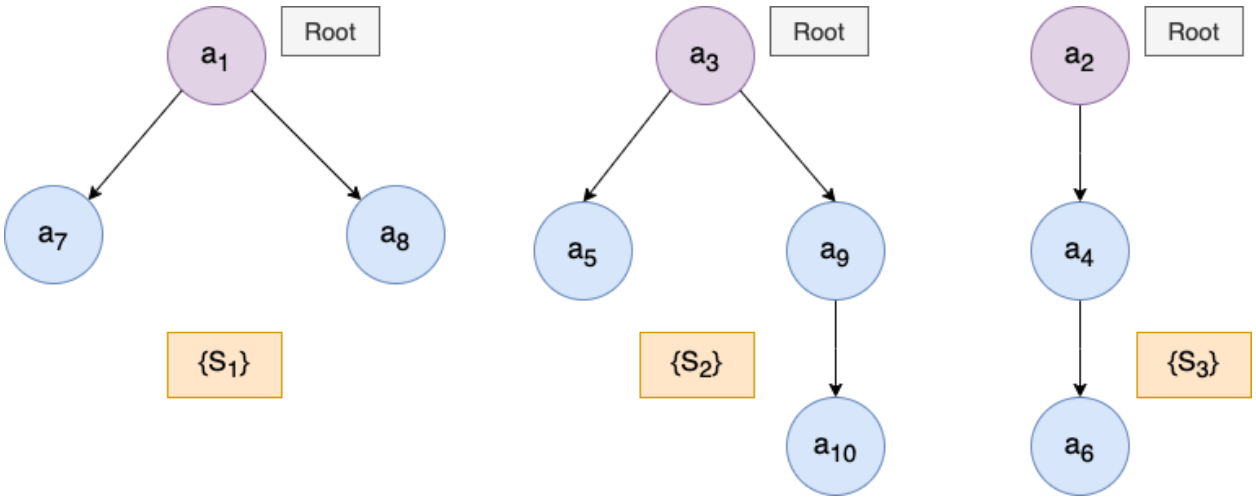
Implementing Disjoint Set

In the optimal implementation of the Disjoint Set we will consider each set as a **tree** with each node containing one member. The entire disjoint set will then become a collection of trees and hence called **Disjoint Set Forest**.

Let's take an example of the Disjoint Set below.



In our implementation, the above Disjoint-Set can be represented as:



Remember we discussed the three operations of a Disjoint-Set data structure earlier? Let's revisit those operations again.

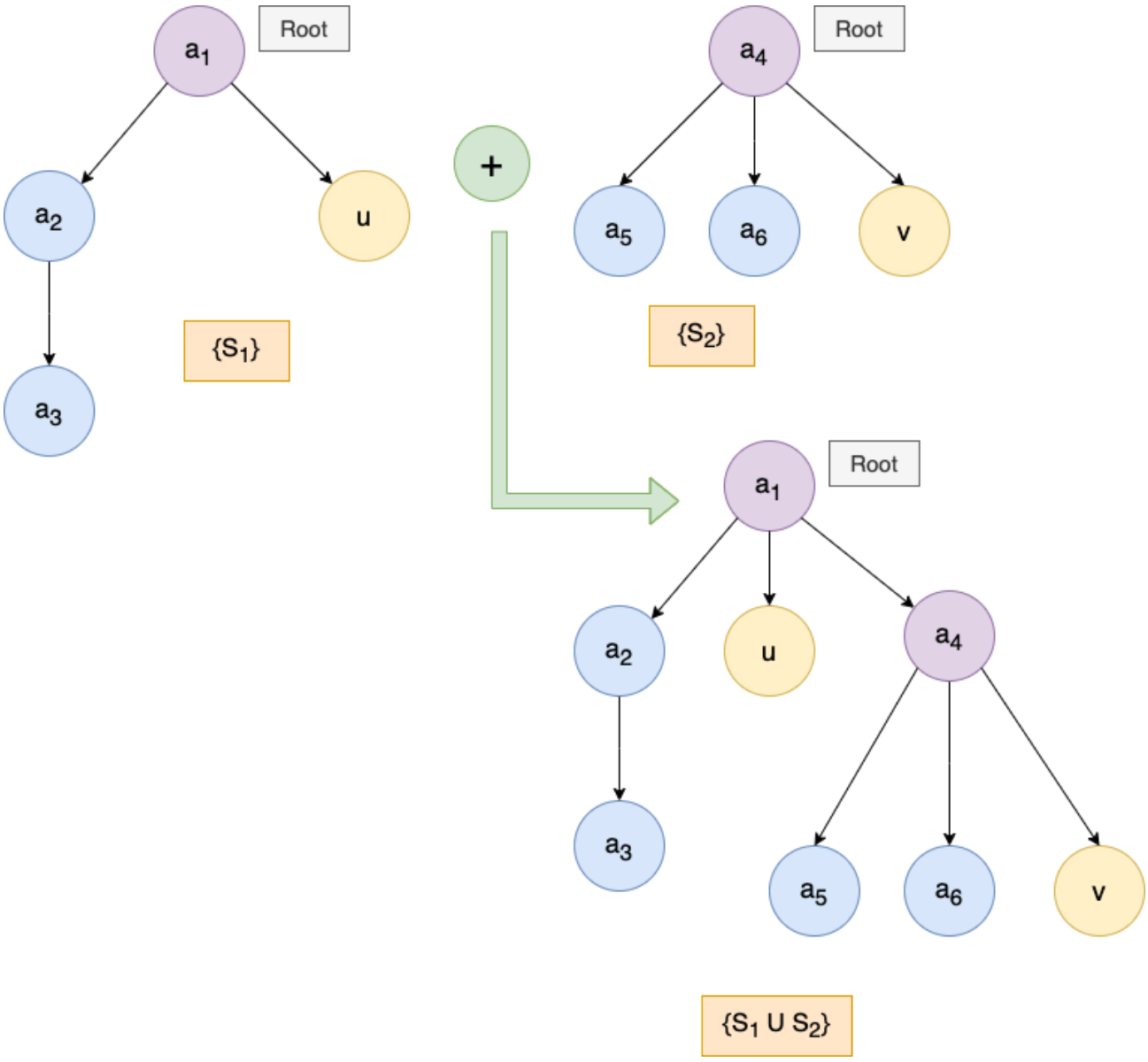
MAKE_SET(x)

In this operation we will simply create a tree with one node i.e. **x**.

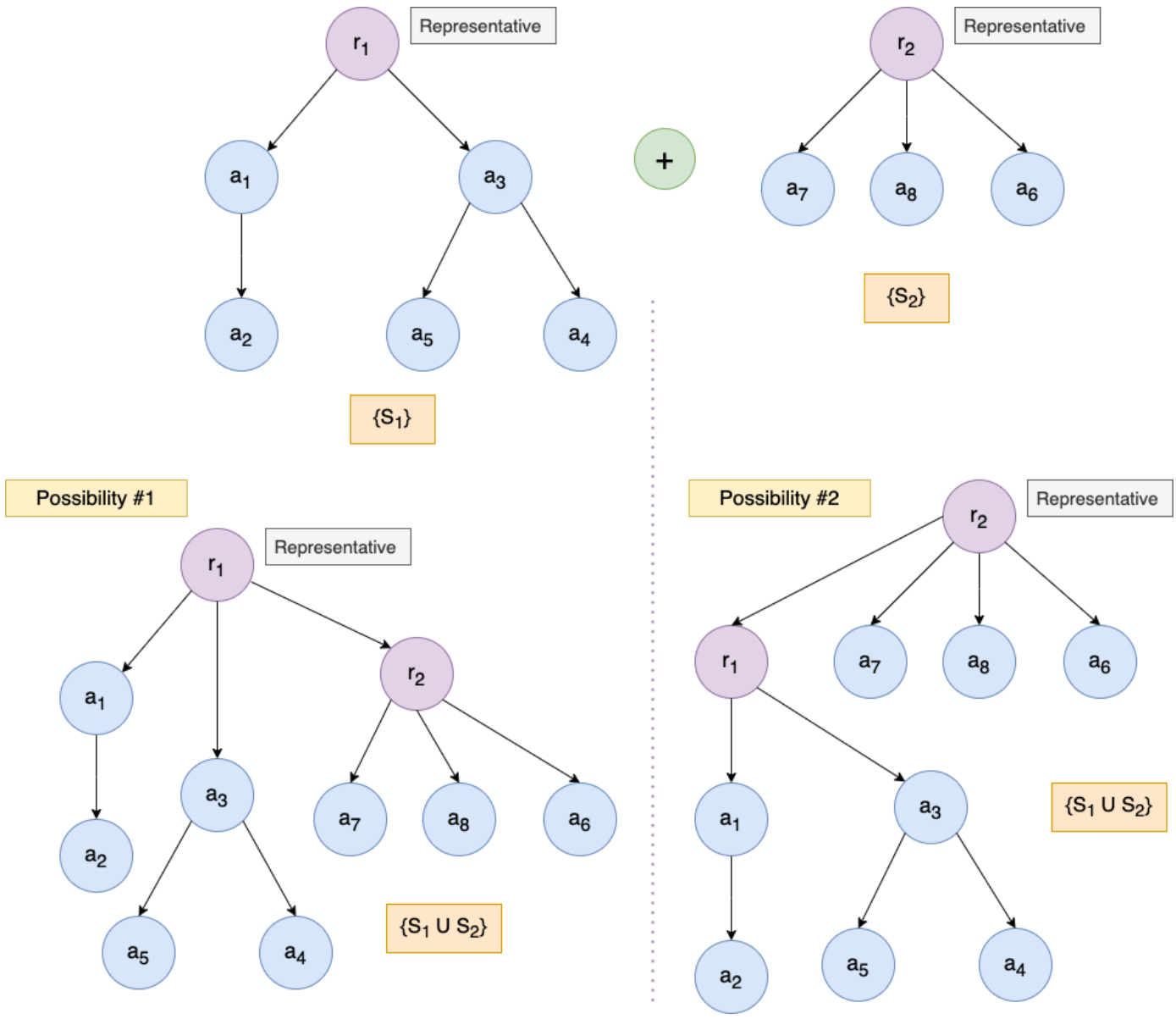
UNION(u, v)

In this operation we will put one tree as a subtree of the other tree. Since each tree is a Disjoint Set, this is equivalent to the union of two sets.

Suppose **u** is a part of set $\mathbf{S_1}$ ($u \in S_1$) and **v** is a part of set $\mathbf{S_2}$ then the union operation will look somewhat like this.

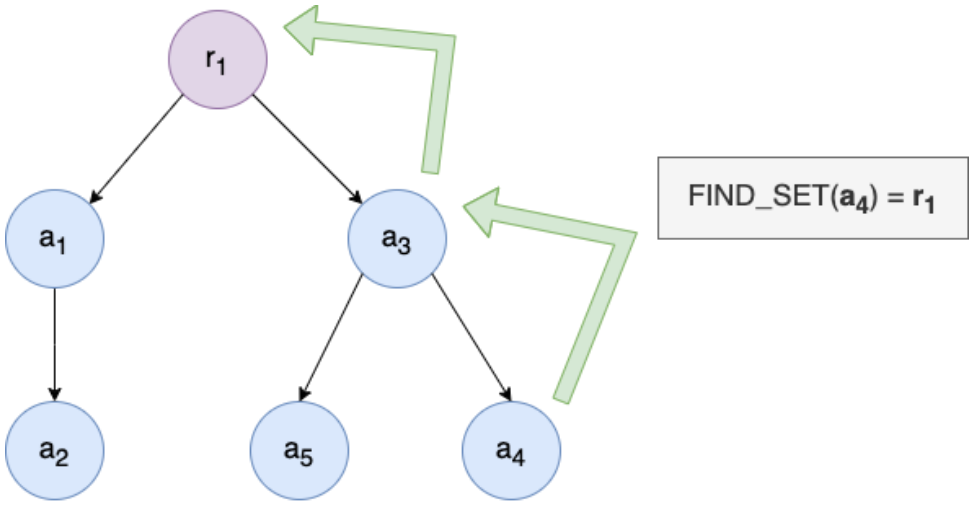


Every **root** of the tree is the representative of that set. Hence after union one of the two representatives becomes the final representative of the unioned set. This is factually correct since we put one tree as a subtree to another.



FIND_SET(x)

We simply move from the node towards the root of the tree and finally return the **root** which is equivalent to the representative of the set.

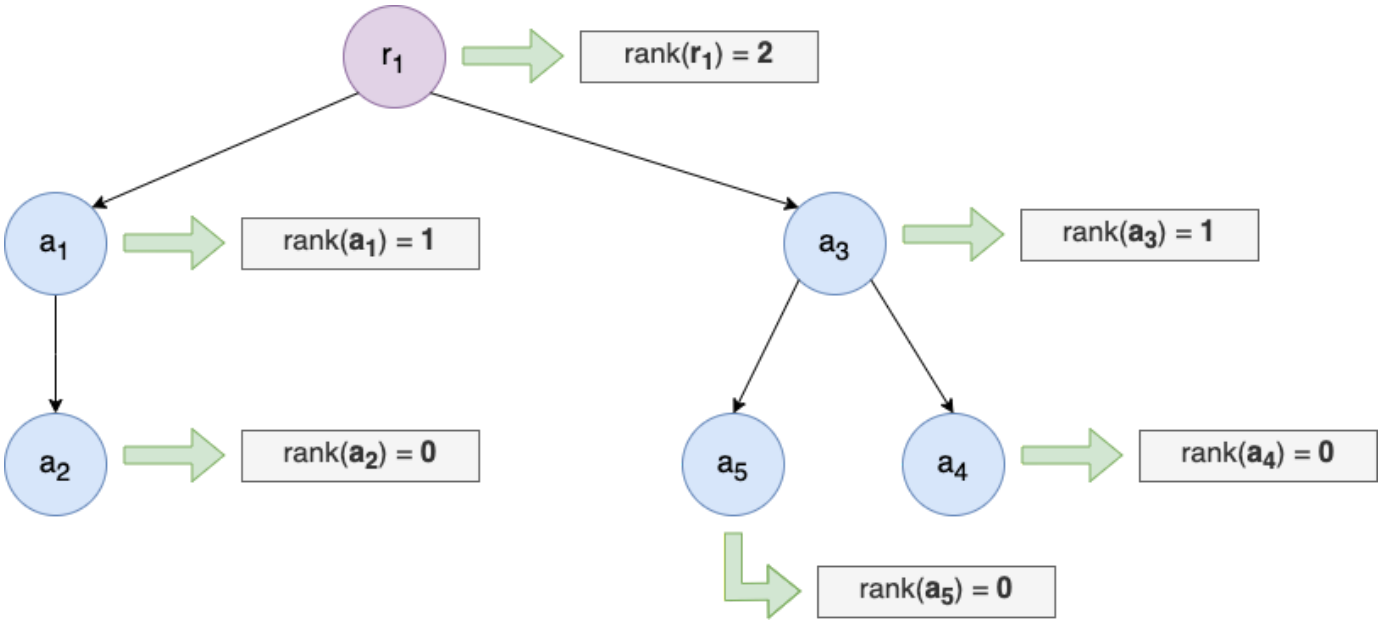


Code Implementation of the Disjoint Set forest

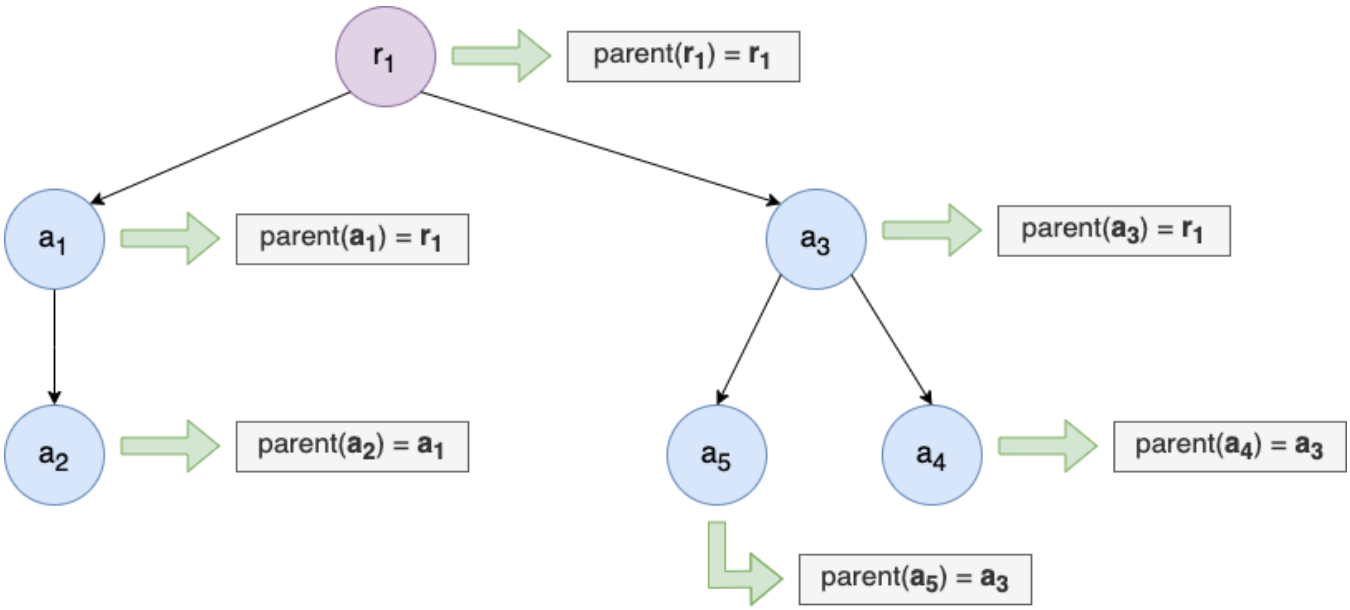
In the previous section we discussed how to implement a Disjoint Set data structure through a collection of **trees** or **forest**. Let's discuss how the code implementation looks like for the same.

The Disjoint Set looks like this.

- int[] rank: We associate each element in the disjoint-set with a rank which is described as the upper-bound on the height of that element in the tree.



- int[] parent: The parent array stores the parent of every element in the tree. The root node is parent to itself.



MAKE_SET(x) implementation

In the operation **MAKE_SET(x)** implementation we initialize the **rank** and **parent** of every element.

```
MAKE_SET (x):
     parent[x] <- x
     rank[x] <- 0
```

UNION(u, v) implementation

In this operation we union the two sets which include **u** and **v** respectively, if they are disjoint. The sets are union according to their ranks. The tree with smaller rank becomes the **subtree** of the other tree (with larger rank).

```
UNION (u, v):
     parent_u <- parent[u]
     parent_v <- parent[v]
```

```
    IF parent_u != parent_v
        IF rank[parent_u] > rank[parent_v]
            parent[v] <- parent_u
        ELSE
            parent[u] <- parent_v
            IF rank[parent_u] == rank[parent_v]
                rank[parent_v]++
```

FIND_SET(x) implementation

In this operation we backtrack to get the root of the tree.

```
FIND_SET(x)
    IF x != parent[x]
        return FIND_SET(parent[x])

    return x
```

The above implementation is a simple recursive method to get the root of the tree.

This implementation has a runtime of **O(M.log(N))** where **M** is the number of operations and **N** is the number of elements in the Disjoint Set.

Let's look at the runtime complexity for each operation.

1. MAKE_SET(x): It has a constant runtime complexity of **O(1)**. Can be easily computed by looking at the method implementation.

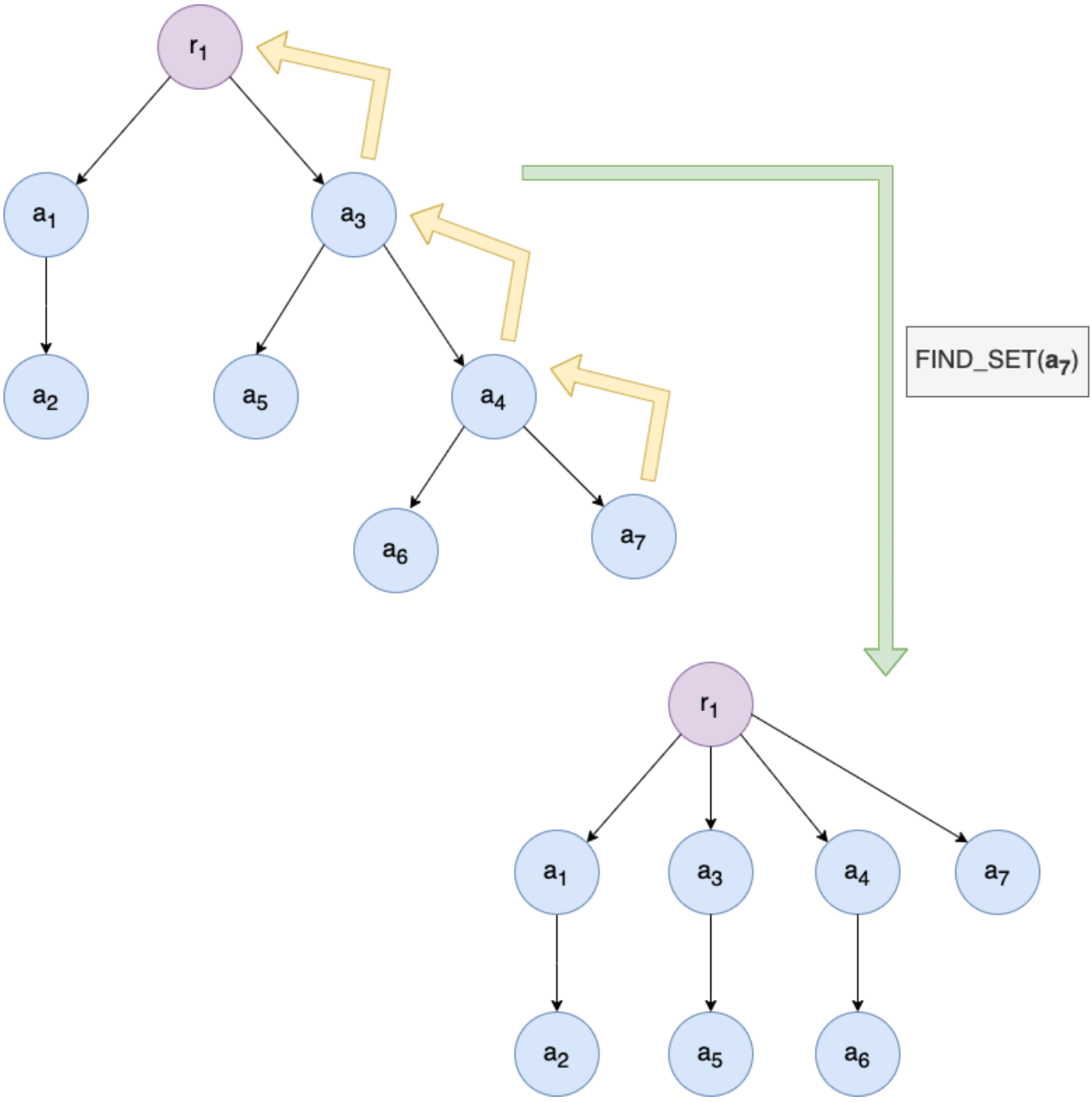
2. FIND_SET(x): It has a runtime complexity of **O(logN)** where **N** is the number of elements in the tree. Since the tree is un-skewed, we can get a **logarithmic** complexity as the height of the tree will be around **log(N)**.

3. UNION(u, v): The operation uses **FIND_SET** operation to determine the representative to two disjoint sets. Hence, it will also have a runtime complexity of **O(logN)**.

Path Compression and Super Linear runtime

Path Compression is a technique in which we modify the existing **FIND_SET(x)** method so that every node in the path from **x** to the **root** of the tree points directly to the root node.

The process looks like this.



The technique is called Path Compression because we can observe that we have compressed the path between the intermediate nodes (from $\mathbf{a_7}$ to $\mathbf{r_1}$) and the root node.

This improves the runtime complexity of the **FIND_SET(x)** operation, since we can directly reach the root-node in one hop from the intermediate nodes.

The implementation of the path compression in **FIND_SET** operation can be described as:

```
FIND_SET(x)
     IF x != parent[x]
          parent[x] <- FIND_SET(parent[x])

     return parent[x]
```

It has been proved that the runtime complexity of **FIND_SET(x)** operation is reduced from **log(N)** to **α(N)** after the implementation of Path Compression. **α** is a very slowly growing function also known as the **Inverse Ackermann** function and **α(N)** <= 4 for almost all the possible values of **N**.

Hence, the entire complexity of the series of **M** disjoint set operations with **N** elements becomes **O(M.α(N))**.

Since, **O(α(N)) <= 4**

We can safely assume that: **O(M.α(N))** ~ **O(M)**

Hence, we can say that the overall runtime of a Disjoint Set data structure operation after path compression is almost **linear** or strictly **superlinear**.

Thank You! ❤️

Hope this handbook helped you in clearing out the concept of **Disjoint Set** data structure and how it achieves the **Superlinear** complexity.

Follow me on Linkedin
Do subscribe to my engineering newsletter "Systems That Scale"