DISJOINT SET UNION

This data structure provides the following capabilities. We are given several elements, each of which is a seperate set. A DSU will have an operation to combine any two sets, and it will be able to tell in which set a specific element is. The classical version also introduces a third operation, it can create a set from a new element.

Thus the basic interface of this data

Structure consists of only three operations:

* MAKE-SET(v) → creates a new set consisting of new element v.

* UNION-SETS(a, b) → merges the two specified sets (the set in which the element a is located, and the set in which the element b is located)

* FIND-SET(v) → returns the representative (also called leader) of the set that contains the element v. This representative is an element of its corresponding set. It is selected in each set by the data structure itself (and can change over time, namely after UNION-SETS calls). This representative can be used to check if two elements are part of the same set or not. a and b are exactly in the same set, if FIND-SET(a) == FIND-SET(b). Otherwise they are in different sets.

As described in more detail later, the data structure allows you to do each of these operations in almost O(1) time on average.

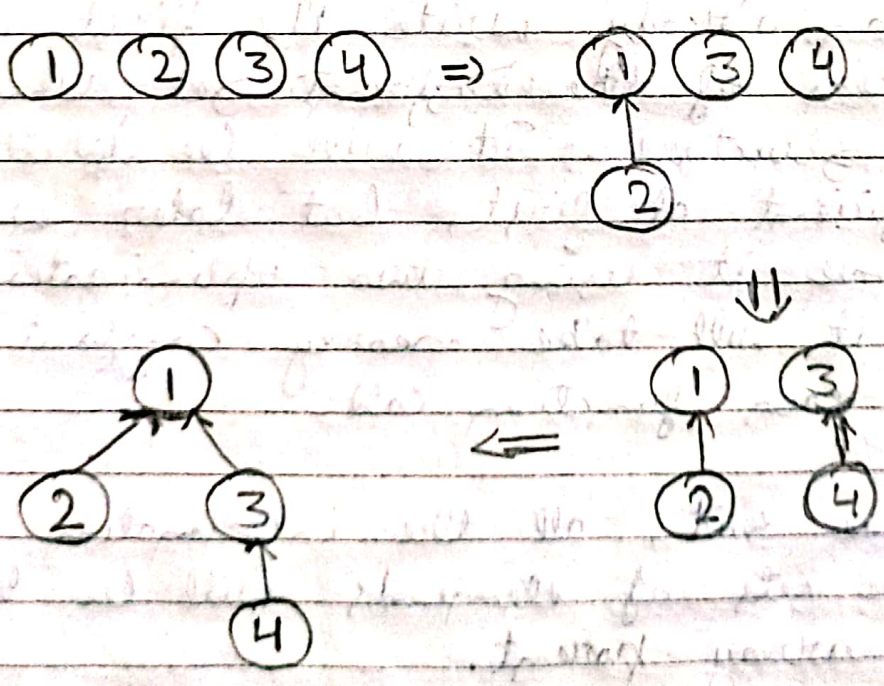
Also in one of the subsections an

alternative structure of a DSU is explained which achieves a slower average complexity of $O(\log n)$, but can be more powerful than the regular DSU structure.

BUILD AN EFFICIENT DATA STRUCTURE

We will store the sets in the form of trees : each tree will correspond to one set. And the root of the tree will be the representative / leader of the set.

In the following image, you can see the representation of such 3 trees.



In the beginning, every element starts as a single set, therefore each vertex is its own tree. Then we combine the set containing the

element 1 and the set containing the element 2. Then we combine the set containing the element 3 and the set containing the element 4. And in the last step, we combine the set containing the element 1 and the set containing the element 3.

For the implementation this ~~means~~ means that we will have to maintain an array PARENT that stores a reference to its immediate ancestor in the tree.

NAIVE IMPLEMENTATION

We can already write the first implementation of the Disjoint Set Union data structure. It will be pretty inefficient at first, but later we can improve it using two optimizations, so that it will take nearly constant time for each function call.

As we said, all the information about the sets of elements will be kept in an array parent.

To create a new set (operation MAKE_SET(v)) we simply create a tree with root in the vertex v, meaning that it is its own ancestor.

To combine two sets (operation UNION-SETS (a,b)), we first find the representative of the set in which a is located, and the representative of the set in which b is located. If the representatives are identical, that we have nothing to do, the sets are already merged. Otherwise, we can simply specify that one of the representatives in the parent of the other representative - thereby combining the two trees.

Finally the implementation of the find representative function (operation FIND-SET(v)): we simply climb the ancestors of the vertex v until we reach the root, i.e. a vertex such that the reference ~~function~~ to the ancestor leads to itself. This operation is easily implemented recursively.

CODE

```
VOID MAKE-SET(INT V)
{
    PARENT[V] = V;
}

INT FIND-SET(INT V)
{
    IF (V == PARENT[V]) RETURN V;
    RETURN FIND-SET(PARENT[V]);
}
```

```
VOID UNION-SETS (INT A, INT B)
{
    a = FIND-SET (A);
    b = FIND-SET (B);
    IF (a != b)  PARENT[b] = a;
}
```

However this implementation is inefficient. It is easy to construct an example, so that the trees degenerate into long chains. In that case each call FIND-SET(v) can take O(n) time.

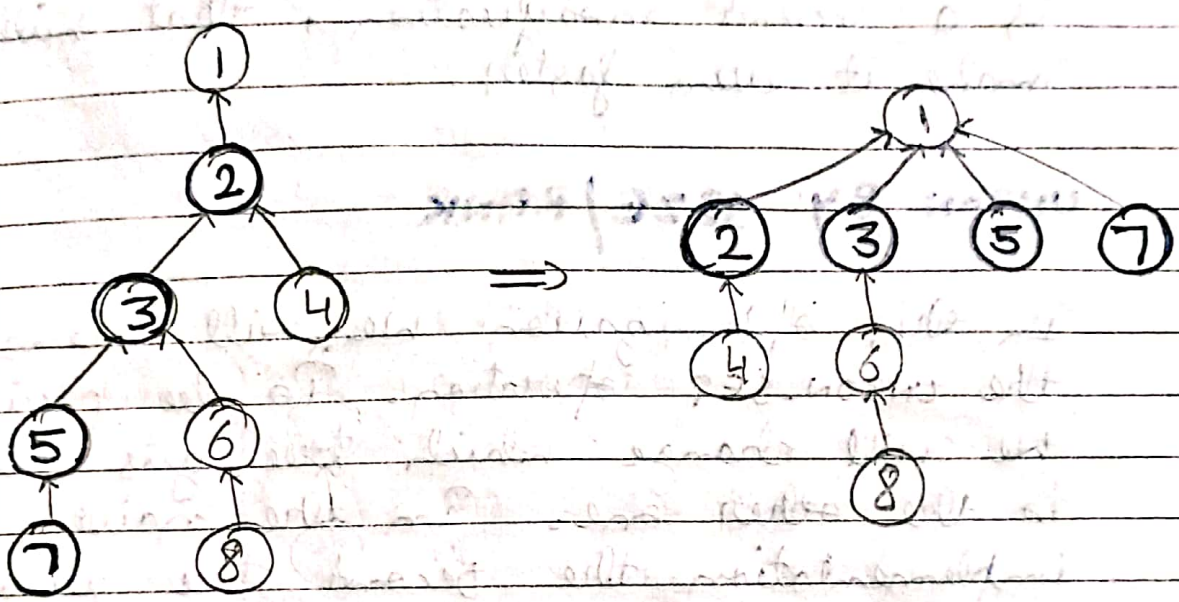
This is far away from the complexity that we want to have (nearly constant time). Therefore we will consider two optimizations that will allow to significantly accelerate the work

PATH COMPRESSION OPTIMIZATION

This optimization is designed for speeding up FIND-SET.

If we call FIND-SET(v) for some vertex v, we actually find the representative p for all vertices that we visit on the path b/w v and the actual representative p. The trick is to make the paths for all those nodes shorter, by setting the parent of each visited vertex directly to p.

You can see the operation in the following image. On the left there is a tree, and on the right side there is the compressed tree after calling FIND_SET(7), which shortens the paths for the visited nodes 7, 5, 3 and 2.



The new implementation of FIND_SET is as follows:

CODE

```
INT FIND_SET (INT V)
{
    IF ( V == PARENT[V] ) RETURN V;
    RETURN PARENT[V] = FIND_SET (PARENT[V]);
}
```

The simple implementation does what was intended: First find the representative of the set (root vertex), and then in the process of stack unwinding

the visited nodes are attached directly to the representative.

This simple modification of the operation already achieves the time complexity $O(\log n)$ per call on average. There is a second modification, that will make it even faster.

UNION BY SIZE/RANK

In this optimization we will change the UNION_SET operation. To be precise, we will change which tree gets attached to the other one. In the naive implementation the second tree always got ~~attat~~ attached to the first one. In practice that can lead to trees containing chains of length $O(n)$. With this optimization we will avoid this by choosing very carefully which tree gets attached.

There are many possible heuristics that can be used. Most popular are the following two approaches: In the first approch we use the size of the tree as rank, and in the second one we use the depth of the tree (more precisely, the upper bound on the tree depth, because the depth will get

smaller when applying path compression).

In both approches the essence of the optimization is the same: we attach the tree with the lower rank to the one with the bigger rank.

Here is the implementation of union by size:

CODE

```
VOID MAKE_SET (INT V)
{
    PARENT[V] = V;
    SIZE[V] = 1;
}

VOID UNION_SETS (INT A, INT B)
{
    a = FIND_SET(A);
    b = FIND_SET(B);
    IF (a != b)
    {
        IF (SIZE[a] < SIZE[b])
            SWAP(a, b);
        PARENT[b] = a;
        SIZE[a] += SIZE[b];
    }
}
```

And here is the implementation of

union by rank based on the depth of the trees :

```
VOID MAKE_SET(INT V)
{
    PARENT[V] = V;
    RANK[V] = 0;
}

VOID UNION_SETS(INT A, INT B)
{
    a = FIND_SET(A);
    b = FIND_SET(B);

    IF (a != b)
    {
        IF (RANK[a] < RANK[B])
            SWAP(a, b);
        PARENT[b] = a;
        IF (RANK[a] == RANK[B])
            RANK[a]++;
    }
}
```

Both optimizations are equivalent in terms of time and space complexity.

TIME COMPLEXITY

As mentioned before, if we combine both optimizations - path compression

With union by rank / size - we will reach nearly constant O time queries. It turns out, that the final amortized time complexity is $O(\alpha(n))$, where $\alpha(n)$ is the inverse Ackermann function, which grows very slowly. In fact it grows so slowly, that it doesn't exceed 4 for all reasonable n (approx $n < 10^{600}$).

Amortized O complexity is the total time per operation, evaluated over a sequence of multiple operations. The idea is to guarantee the total time of the entire sequence, while allowing single operations to be much slower then the amortized time. eg: In our case a single call might take $O(\log n)$ in the worst case, but if we do m such calls back to back we will end up with an average time of $O(\alpha(n))$.