Union by Rank - Disjoint Set

Report Issue

Disjoint Set - Union by Rank

We have implemented two kinds of “disjoint sets” so far, and they both have a concerning inefficiency. Specifically, the quick find implementation will always spend O(n) time on the union operation and in the quick union implementation, as shown in Figure 6, it is possible for all the vertices to form a line after connecting them using `union`, which results in the worst-case scenario for the `find` function. Is there any way to optimize these implementations?

Of course, there is; it is to union by rank. The word “rank” means ordering by specific criteria. Previously, for the `union` function, we always chose the root node of `x` and set it as the new root node for the other vertex. However, by choosing the parent node based on certain criteria (by rank), we can limit the maximum height of each vertex.

To be specific, the “rank” refers to the height of each vertex. When we `union` two vertices, instead of always picking the root of `x` (or `y`, it doesn't matter as long as we're consistent) as the new root node, we choose the root node of the vertex with a larger “rank”. We will merge the shorter tree under the taller tree and assign the root node of the taller tree as the root node for both vertices. In this way, we effectively avoid the possibility of connecting all vertices into a straight line. This optimization is called the “disjoint set” with union by rank.

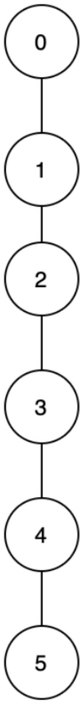


Figure 6. A line graph

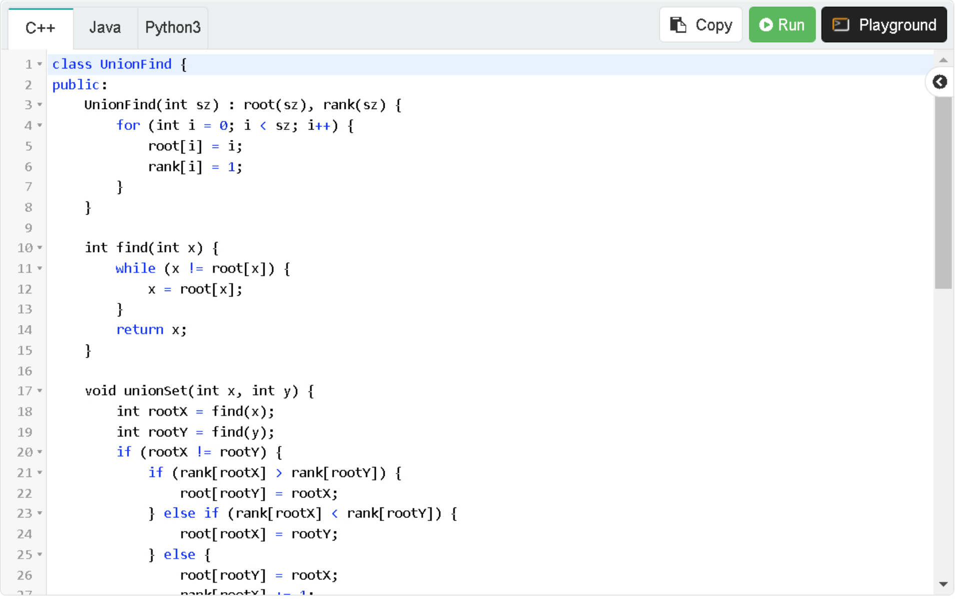
Video Explanation

In this video, you'll learn how to actually implement the “disjoint set” with union by rank.

▶ Clarifying Notes

Algorithm

Here is the sample implementation of union by rank.



```cpp
class UnionFind {
public:
    UnionFind(int sz) : root(sz), rank(sz) {
        for (int i = 0; i < sz; i++) {
            root[i] = i;
            rank[i] = 1;
        }
    }

    int find(int x) {
        while (x != root[x]) {
            x = root[x];
        }
        return x;
    }

    void unionSet(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);
        if (rootX != rootY) {
            if (rank[rootX] > rank[rootY]) {
                root[rootY] = rootX;
            } else if (rank[rootX] < rank[rootY]) {
                root[rootX] = rootY;
            } else {
                root[rootY] = rootX;
                rank[rootX] += 1;
```

Time Complexity

	Union-find Constructor	Find	Union	Connected
Time Complexity	$O(N)$	$O(\log N)$	$O(\log N)$	$O(\log N)$

Note: N is the number of vertices in the graph.

- For the `union-find` constructor, we need to create two arrays of size N each.
- For the `find` operation, in the worst-case scenario, when we repeatedly union components of equal rank, the tree height will be at most $\log(N) + 1$, so the `find` operation requires $O(\log N)$ time.
- For the `union` and `connected` operations, we also need $O(\log N)$ time since these operations are dominated by the `find` operation.

Space Complexity

We need $O(N)$ space to store the array of size N.