Introduction to Union Find

Let’s go over the Union Find pattern, its real-world applications, and some problems we can solve with it.

We'll cover the following...

About the pattern

The **union find** pattern is used to group elements into sets based on a specified property. Each set is nonoverlapping, that is, it contains unique elements that are not present in any other set. The pattern uses a disjoint set data structure, such as an array, to keep track of which set each element belongs to.

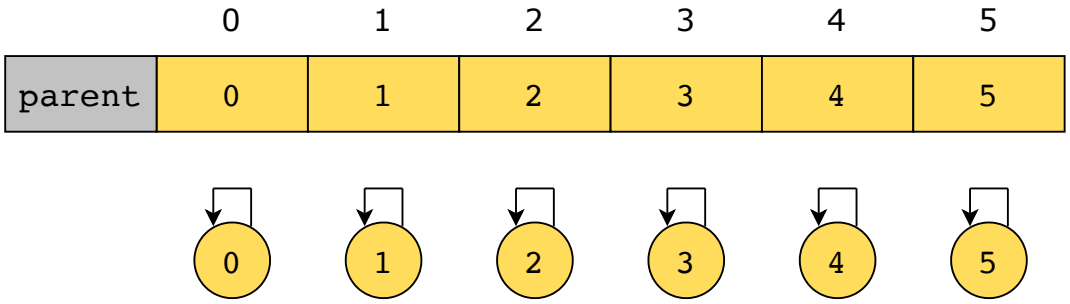
Each set forms a tree data structure and has a representative element that resides at the root of the tree. Every element in this tree maintains a pointer to its parent. The representative’s parent pointer points to itself. If we pick any element in a set and follow its parent pointers, we’ll always reach the set representative.

The pattern is composed of two operations performed on the disjoint data structure:

- **find(v):** Finds the representative of the set that contains v.
- **union(v1, v2):** Merges the sets containing v1 and v2 into one.

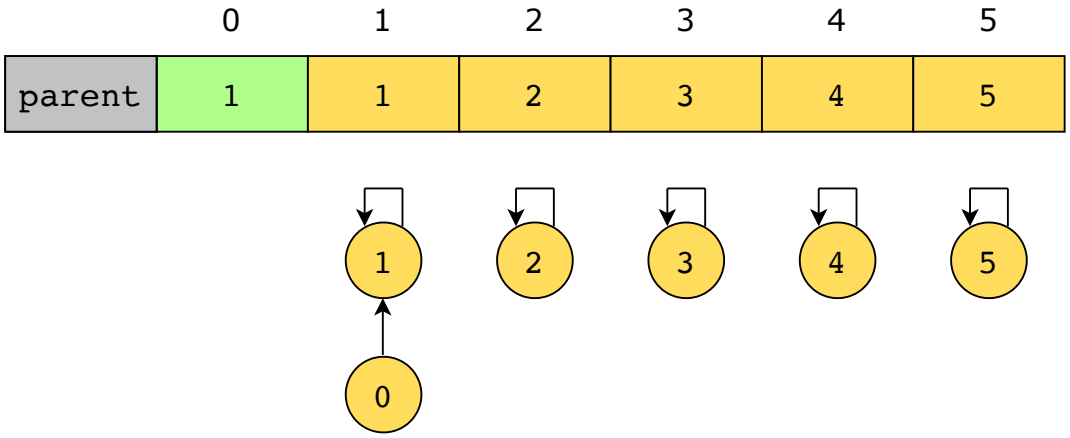
Here is how the union find pattern uses these operations to form and extract information from sets of elements:

We will demonstrate how Union Find works without using union by rank or path compression. The **union(v1, v2)** method merges the tree containing v1 into the set containing v2. The **find(v)** method finds the representative of the tree containing v.

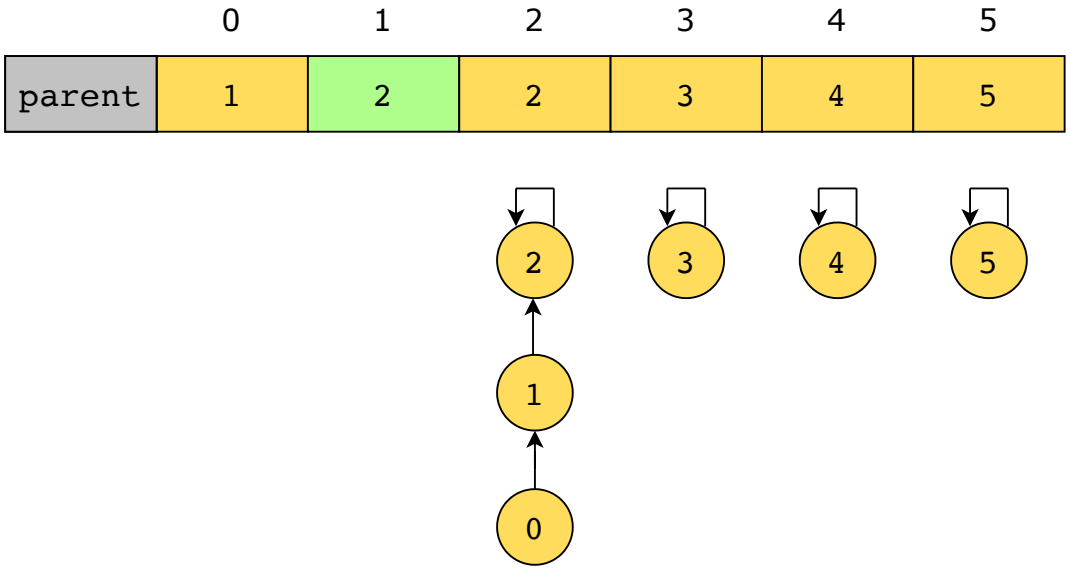


1 of 7

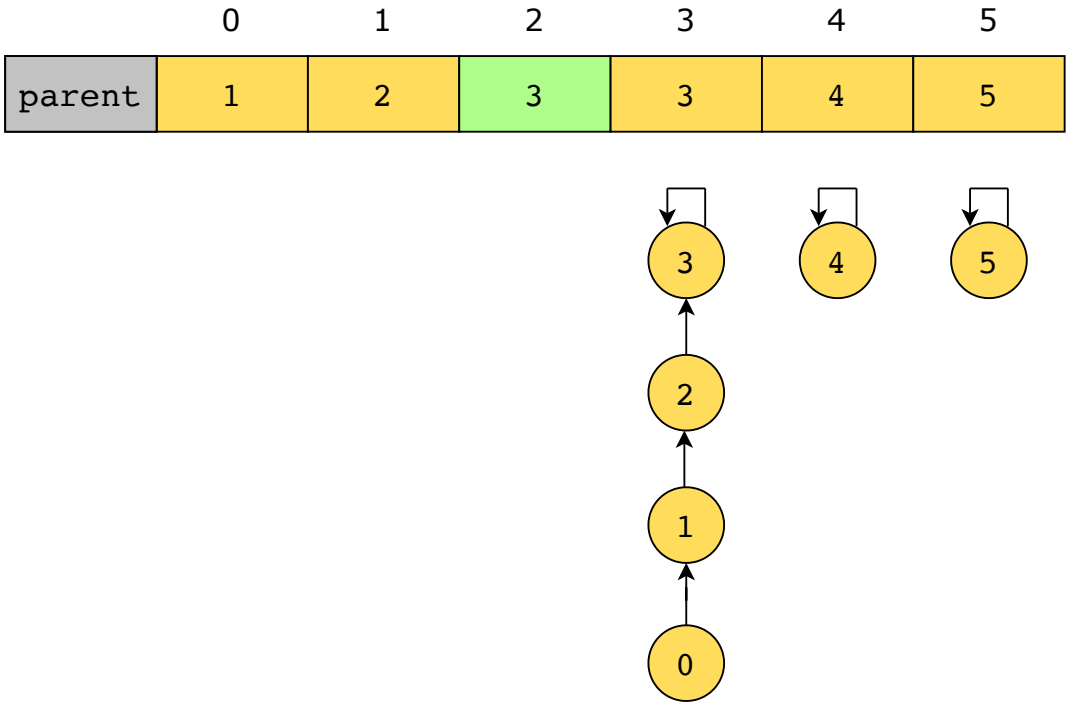
union(0, 1): Search for the representatives of 0 and 1. Merge the tree containing 0 with the tree containing 1. Change the representative of 0 to the representative of 1.



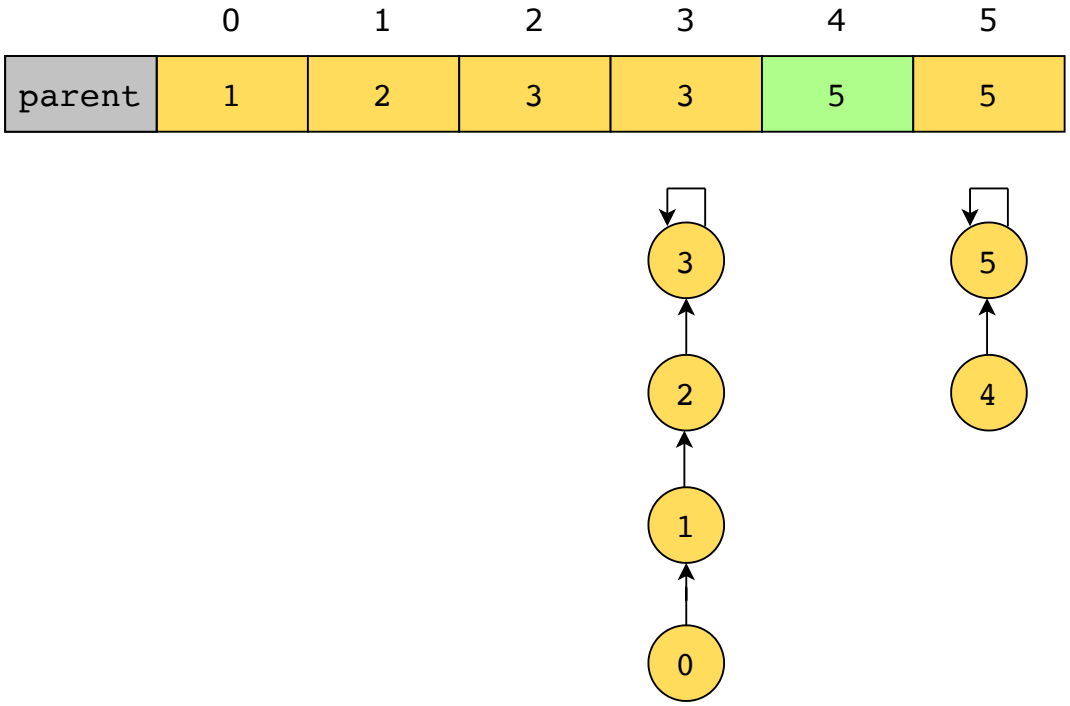
union(1, 2): Search for the representatives of 1 and 2. Merge the tree containing 1 with the tree containing. Change the representative of 1 to the representative of 2.



union(2, 3): Search for the representatives of 2 and 3. Merge the tree containing 2 with the tree containing 3. Change the representative of 2 to the representative of 3.

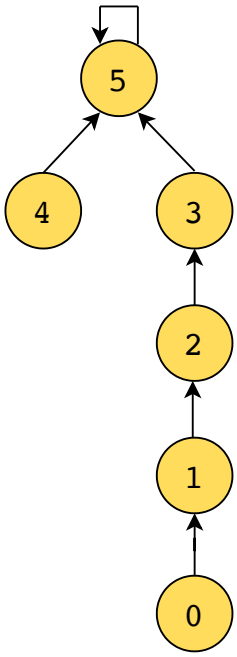


union(4, 5): Search for the representatives of 4 and 5. Merge the tree containing 4 with the tree containing 5. Change the representative of 4 to the representative of 5.



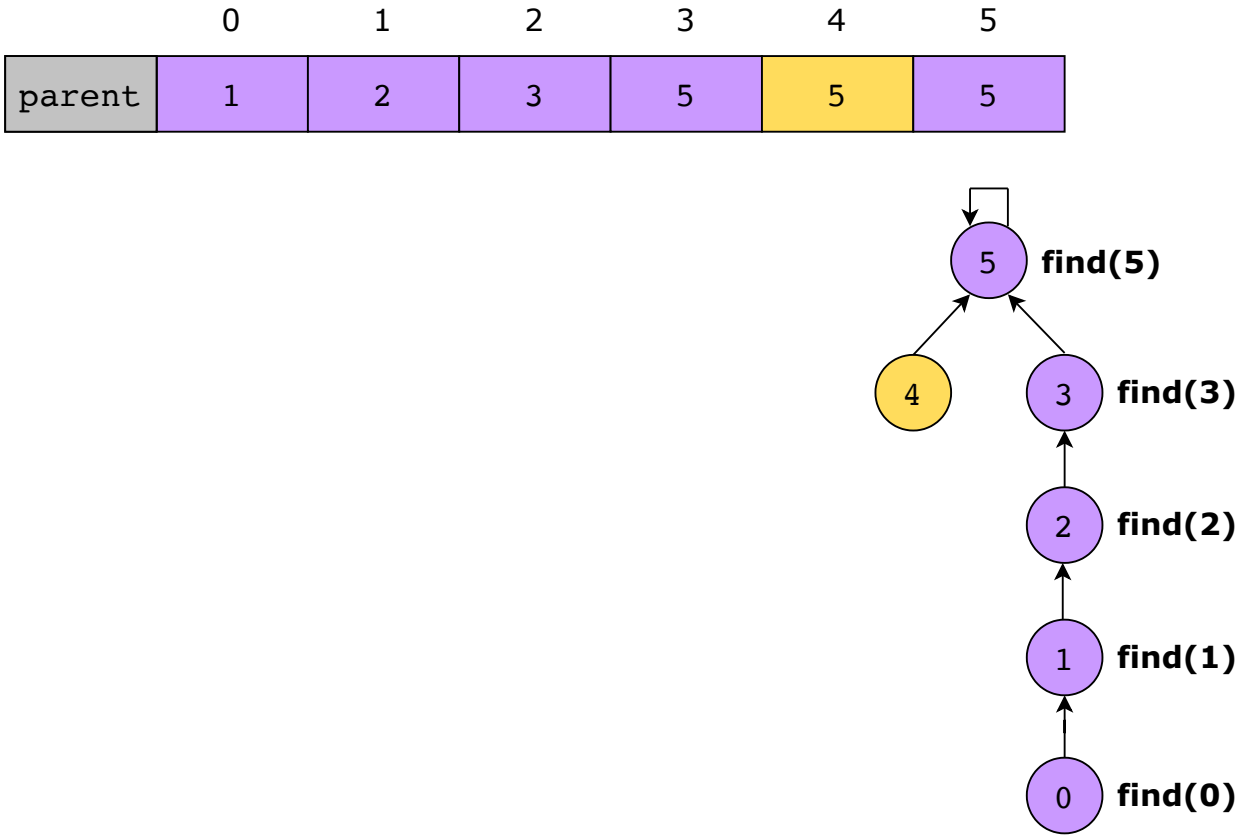
union(3, 5): Search for the representatives of 3 and 5. Merge the tree containing 3 with the tree containing 5. Change the representative of 3 to the representative of 5.

	0	1	2	3	4	5
parent	1	2	3	5	5	5



6 of 7

find(0): Find the representative of the tree containing 0. Five find operations are needed for this. We find the parent of the 0, then the parent of 1 and so on until we reach an element, which is a parent of itself.



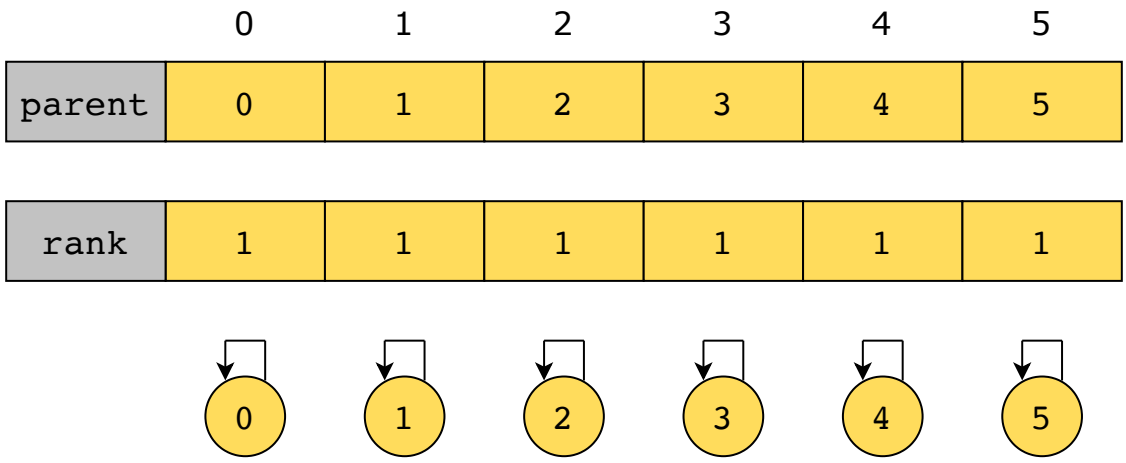
7 of 7

> **Note:** In the above illustration, we skipped the `union(3,4)` function to demonstrate how trees of heights greater than 1 can be merged.

For now, the worst-case time complexity of this approach is $O(n)$ because we might have to traverse the entire tree to find the representative of an element.

We can improve the union find pattern through an optimization called union by rank. We maintain a rank for each tree. The larger the size of the tree, the greater the rank. The idea is that when merging two trees with the union method, we always attach a tree of a lower rank to one with a higher rank. This ensures that when two trees are merged, each element in this merged tree has the shortest possible path to the root.

We will demonstrate how Union Find by rank optimizes the algorithm by assigning a rank to each tree. The rank of each tree is initially 1.



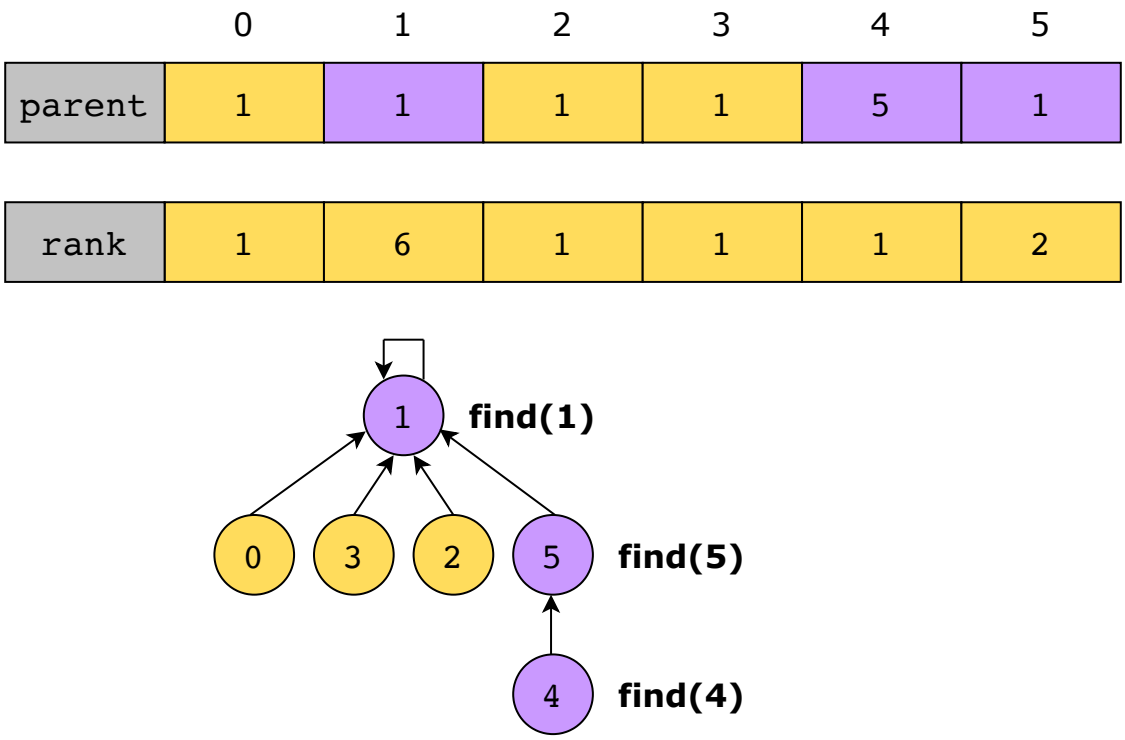
1 of 7

This brings the time complexity down to $O(\log(n))$ in the worst case.

Another optimization to the union find pattern is path compression, where on each find operation on a node of a tree, we update the parent of that node to point directly to the root. This reduces the length of the path of that node to the root, ensuring we don't have to travel all the intermediate nodes on future find operations.

The following illustration shows how path compression works:

To demonstrate the use of path compression, suppose we needed to find the parent of 4. We would first do a **find(4)** operation, in which it would take three find operations to get to its representative.



1 of 2

Both these optimizations used together bring down the worst-case time complexity to lower than $O(\log(n))$. The amortized time complexity becomes $O(\alpha(n))$ which is a small constant. Here, α is the inverse Ackerman function. This is faster than any naive approach to merging subsets, which would take at least $O(n)$ time if not more.

> **Note:** The time complexities explained above were for a single union find operation. When we perform m union find operations, the time complexity becomes $O(m(\alpha(n)))$.

Examples

The following examples illustrate some problems that can be solved with this approach:

1. **Longest consecutive sequence:** Find the longest consecutive sequence of integers in an unsorted array.

Make a disjoint-set union array by assigning a unique index to each unique element in the array.

array	78	2	32	4	1	3

DSU	0	1	2	3	4	5

1 of 3

2. **Successor with delete:** Delete numbers and find a successor to a number in a stream of integers from 1 to n.

Make a disjoint-set union array containing values from 0 to n. The **delete(x)** method deletes x from the stream. This can be mimicked using the **union(x, x+1)** method which makes x the child of x+1. The **successor(x)** method finds the number immediately to the right of x . This can be mimicked by the **find(x+1)** method which finds the root of x+1 that is still a part of the stream of integers.

`n = 5`

DSU	0	1	2	3	4	5

delete(2) = **union(2, 3)**

DSU	0	1	3	3	4	5

successor(1) = **find(2)** = 3

DSU	0	1	3	3	4	5