14.1 Introduction

🚨 New Data Structure Alert 🚨 : Disjoint Sets

People like you and I reside in our countries and live here. We can think of each country as a set and all of the people within it as elements within that set. The same person cannot live in two different countries simultaneously. What we have just modeled is a **disjoint set**.

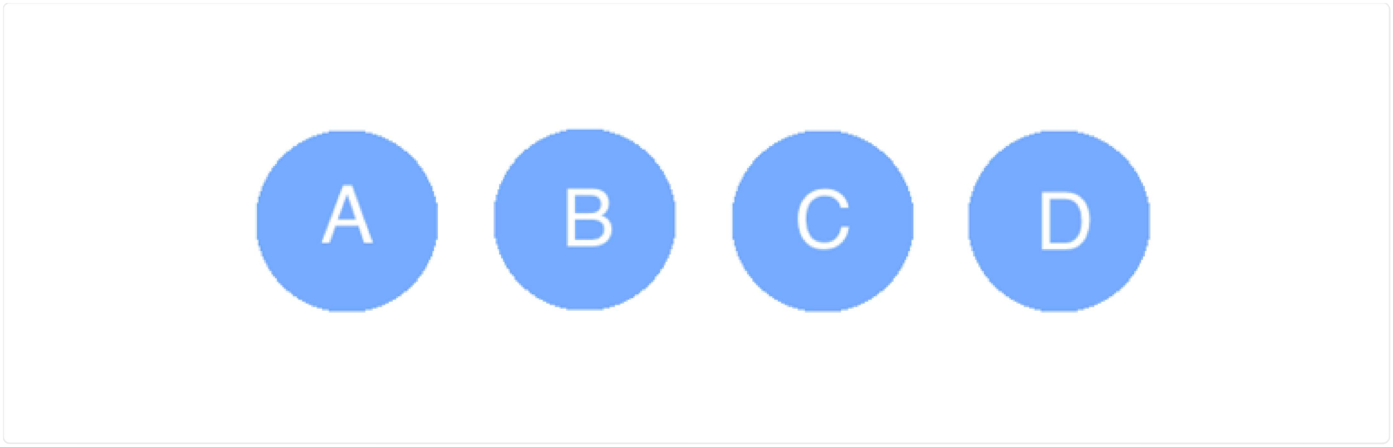
> Two sets are named *disjoint sets* if they have no elements in common. A Disjoint-Sets (or Union-Find) data structure keeps track of a fixed number of elements partitioned into a number of *disjoint sets*. The data structure has two operations:

1. `connect(x, y)` : connect `x` and `y`. Also known as `union`
2. `isConnected(x, y)`: returns true if `x` and `y` are connected (i.e. part of the same set).

Professor Hug's Explanation of an Introduction to Disjoin Sets

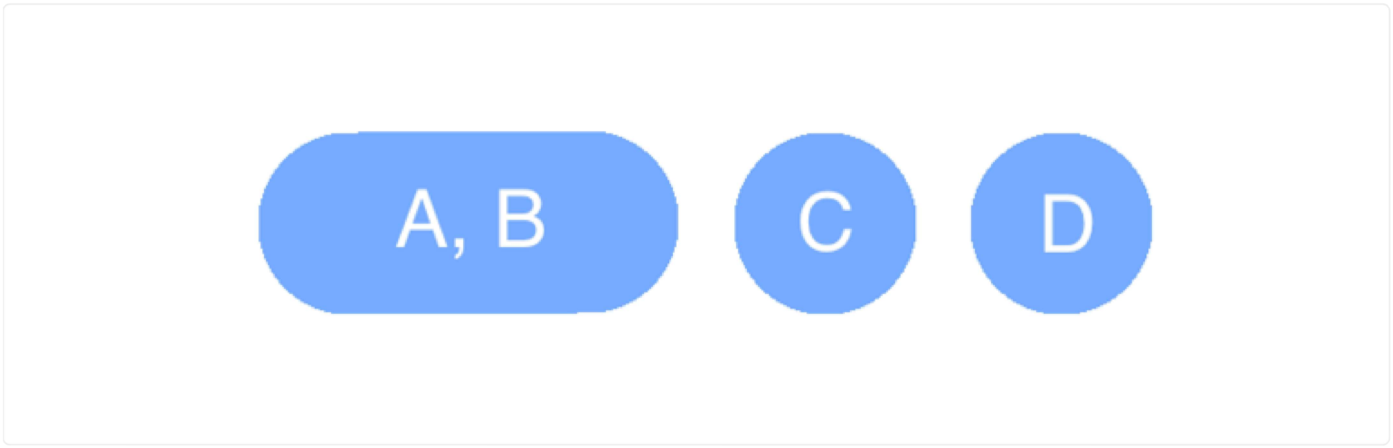
A Disjoint Sets data structure has a fixed number of elements that each start out in their own subset. By calling `connect(x, y)` for some elements `x` and `y`, we merge subsets together.

For example, say we have four elements which we'll call A, B, C, D. To start off, each element is in its own set:



{A} {B} {C} {D}

After calling `connect(A, B)`:



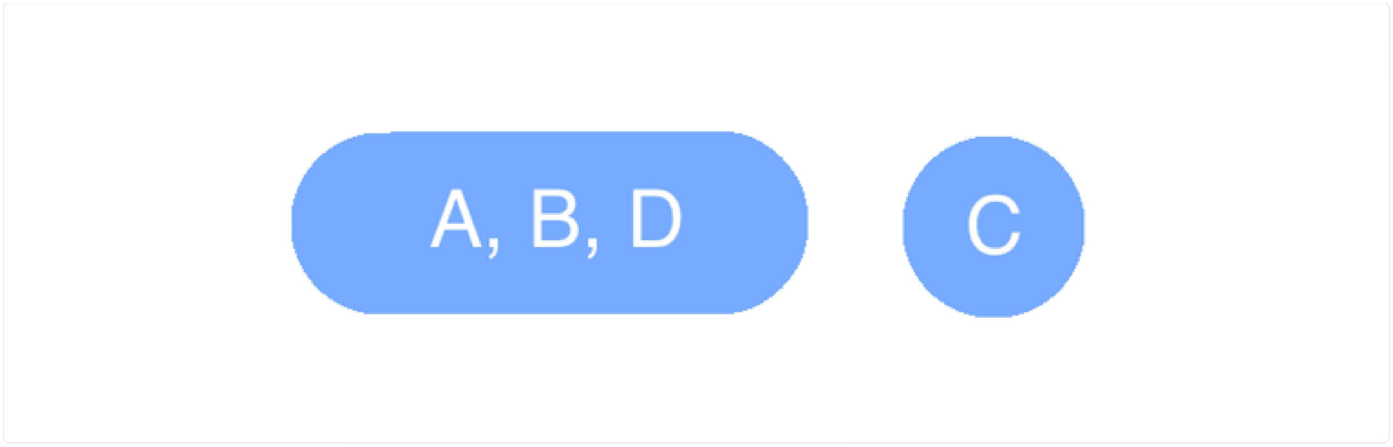
{A, B} {C} {D}

Note that the subsets A and B were merged. Let's check the output some `isConnected` calls:

```
isConnected(A, B) -> true
```

`isConnected(A, C) -> false`

After calling `connect(A, D)`:



{A, B, D} {C}

We find the set A is part of and merge it with the set D is part of, creating one big A, B, D set. C is left alone.

`isConnected(A, D) -> true`
`isConnected(A, C) -> false`

With this intuition in mind, let's formally define what our DisjointSets interface looks like. As a reminder, an **interface** determines *what* behaviors a data structure should have (but not *how* to accomplish it). In this way, any class that implements the `DisjointSets` interface knows to always include functions: `connect(int p, int q)` and `isConnected(int p, int q)` as seen below. For now, we'll only deal with sets of non-negative integers. This is not a limitation because in production we can assign integer values to anything we would like to represent.

```
public interface DisjointSets {
    /** connects two items P and Q */
    void connect(int p, int q);

    /** checks to see if two items are connected */
    boolean isConnected(int p, int q);
}
```

But how are we going to save data for these Disjoint sets to see which member belongs to it's corresponding set? What data structures are we going to use to represent this awesome data structure? In addition to learning about how to implement a fascinating data structure, this chapter will be a chance to see how an implementation of a data structure evolves. We will discuss four iterations of a Disjoint Sets design before being satisfied: *Quick Find* → *Quick Union* → *Weighted Quick Union (WQU)* → *WQU with Path Compression*. **We will see how design decisions greatly affect asymptotic runtime and code complexity.**

Previous
14. Disjoint Sets

Next
14.2 Quick Find

Last updated 1 year ago