Disjoint set/Union find

CPMSoc Programming Term 2

Outline

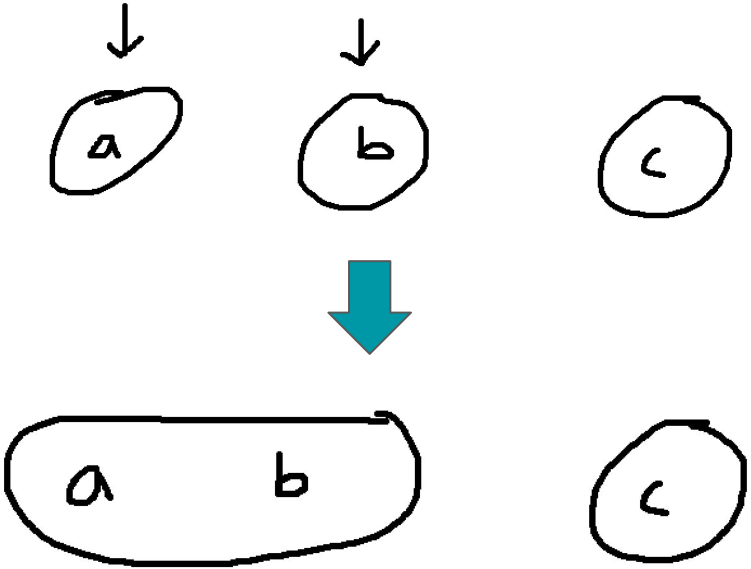
1. The Problem
2. The Data Structure
3. The Implementation
4. The Optimizations
    a. Path compression
    b. Union by size
5. The Applications
    a. Kruskal's algorithm

The Problem

You have: a list of elements, each in their own set.

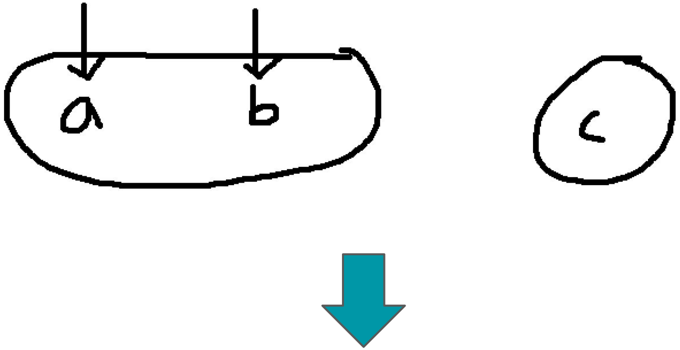
The Problem (merging)

You can: merge any two **sets** together.

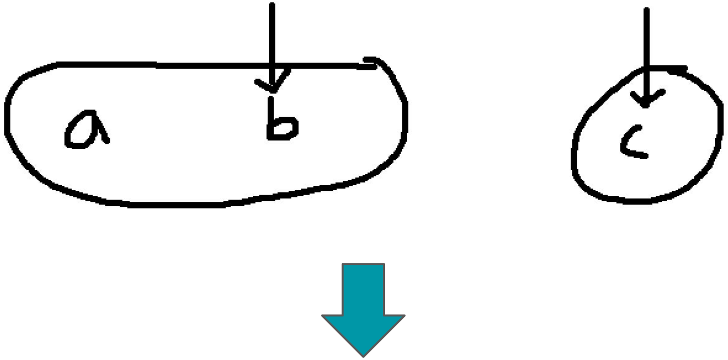


The Problem (commonality)

You can: check whether two **elements** belong to the **same set**.



Yes



No

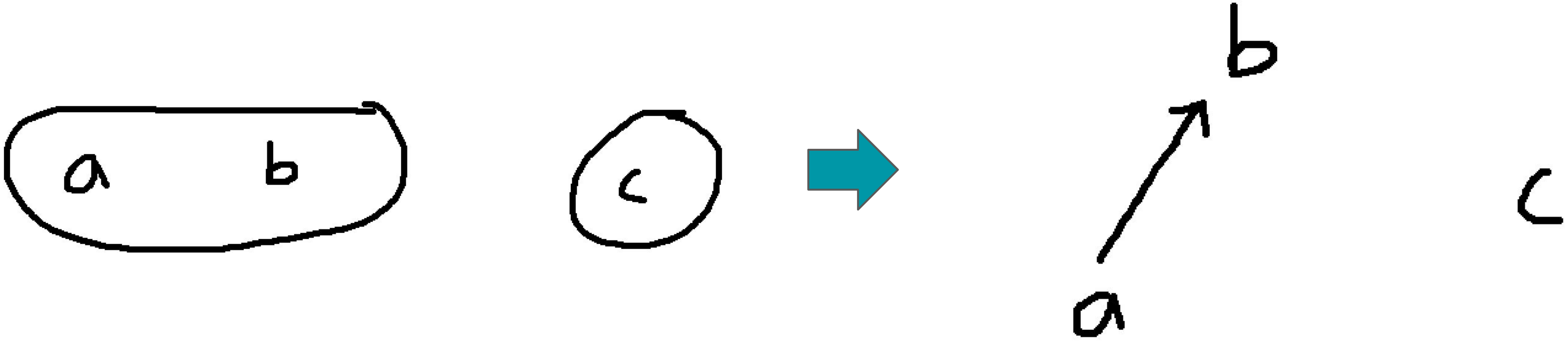
The Problem

How can we do these two operations efficiently?

The Data Structure

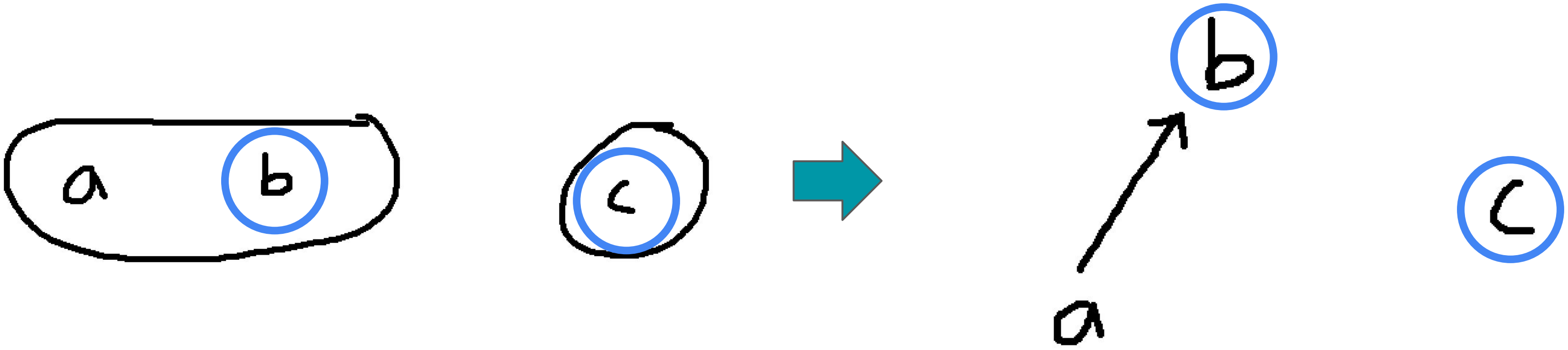
Let's represent the sets as a forest of trees.

Two elements belong to the same set if they have the same ancestor or root.



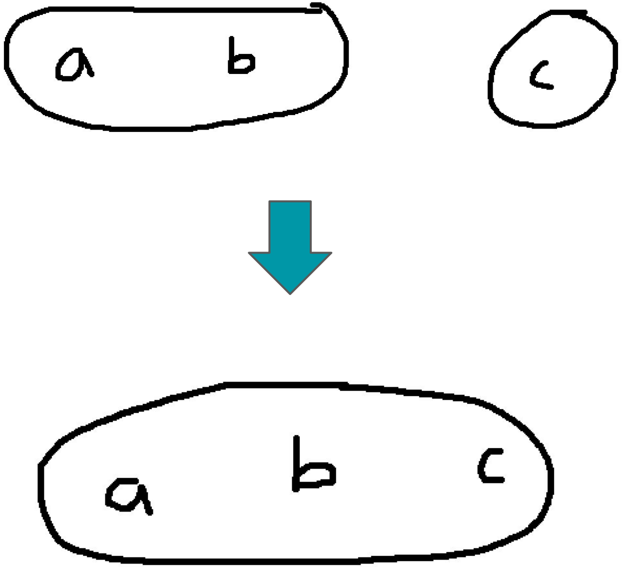
The Data Structure

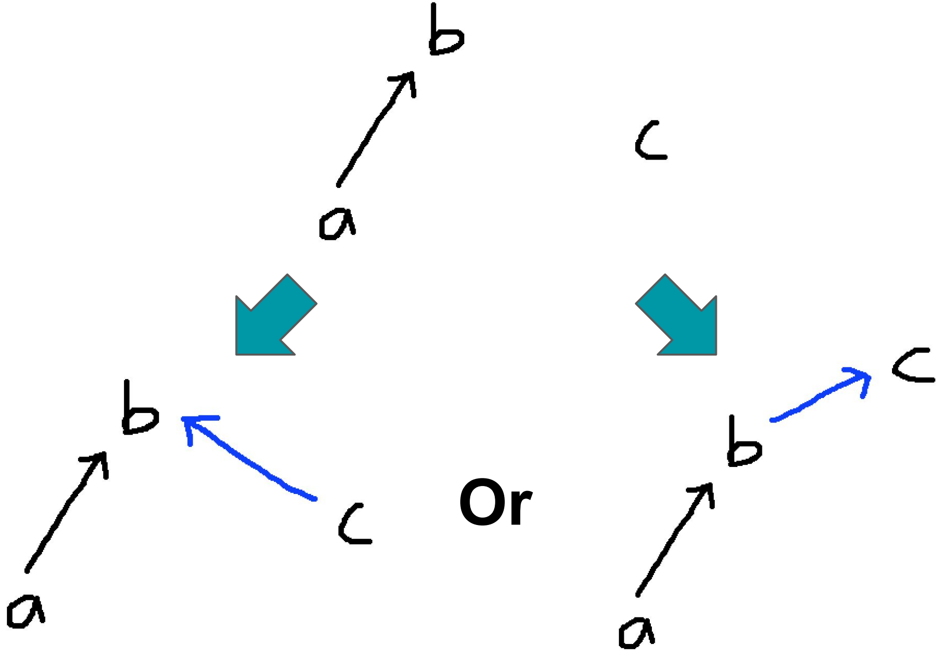
We call the root of a tree the **representative element** of a set.



The Data Structure

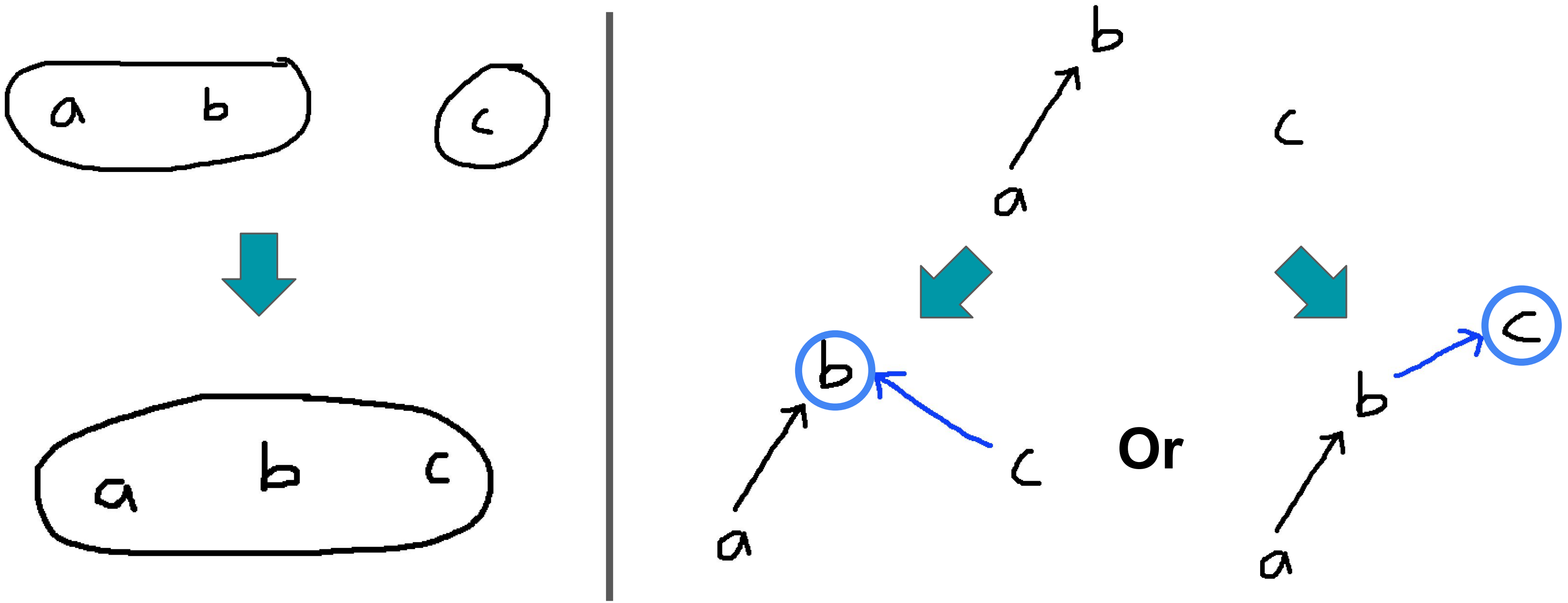
To merge two sets, we point one of the **representatives** into the other.





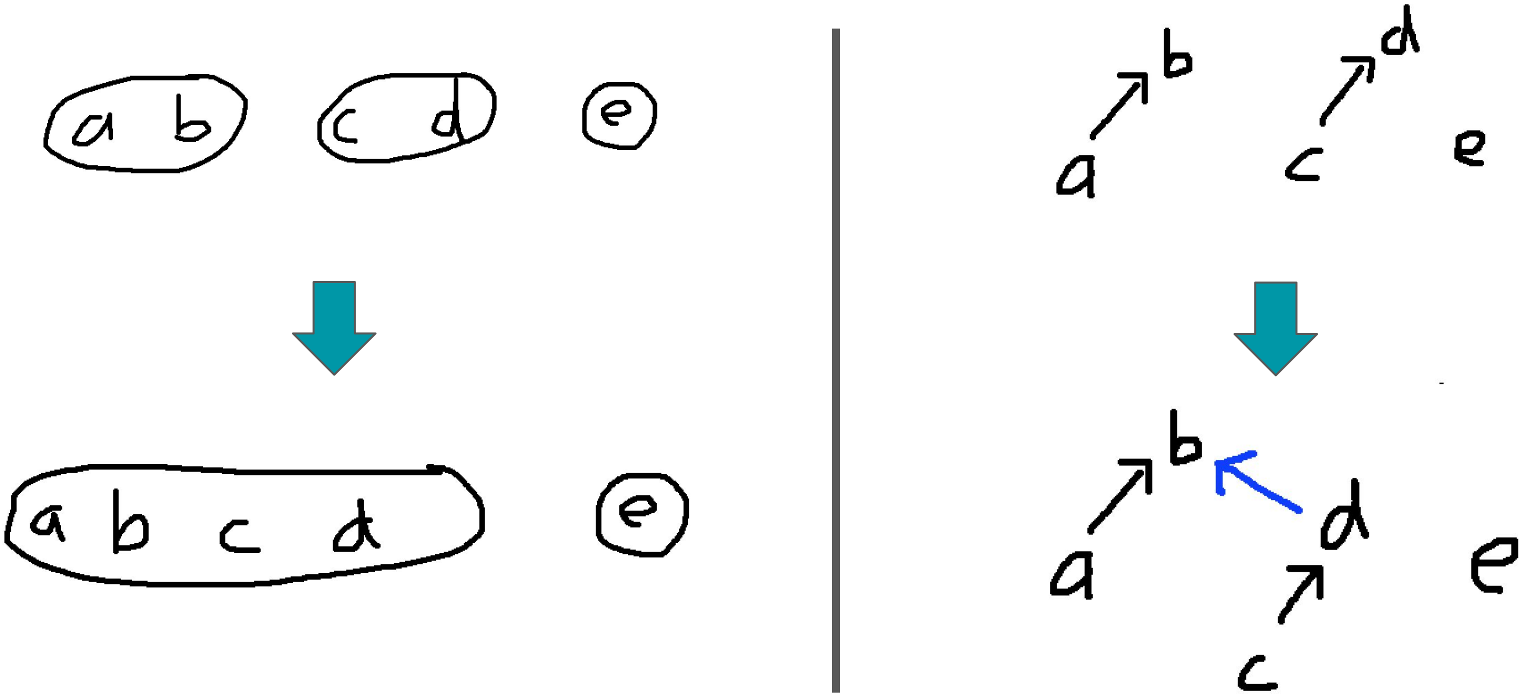
The Data Structure

Now either **b** or **c** is the new representative.



The Data Structure

A more complicated example.



The Implementation (find)

Store the parent of an element using a map/dictionary. Our elements are strings.

find gets the **representative** of the set an element is in.

(C++ idiom to check if a key is in a map.)

Return the representative of the set the parent is in.

Otherwise, this element is already a representative.

```
struct DisjointSet {
    map<string, string> parents;

    string find(string x) {
        if (parents.count(x)) {
            string parent = parents[x];
            return find(parent);
        } else {
            return x;
        }
    }
```

The Implementation (commonality)

Two elements are in the same set if they have the same representative.

```
bool in_same_set(string a, string b) {
    return find(a) == find(b);
}
```

The Implementation (union)

(We call it **merge** because **union** is a keyword in C++.)

merge combines the sets of two elements together.

We **must only merge** if they are **not** already in the same set.

Change the parent of one of the representatives.

```
void merge(string a, string b) {
    if (!in_same_set(a, b)) {
        string a_root = find(a);
        string b_root = find(b);
        parents[a_root] = b_root;
    }
}
```

The Implementation

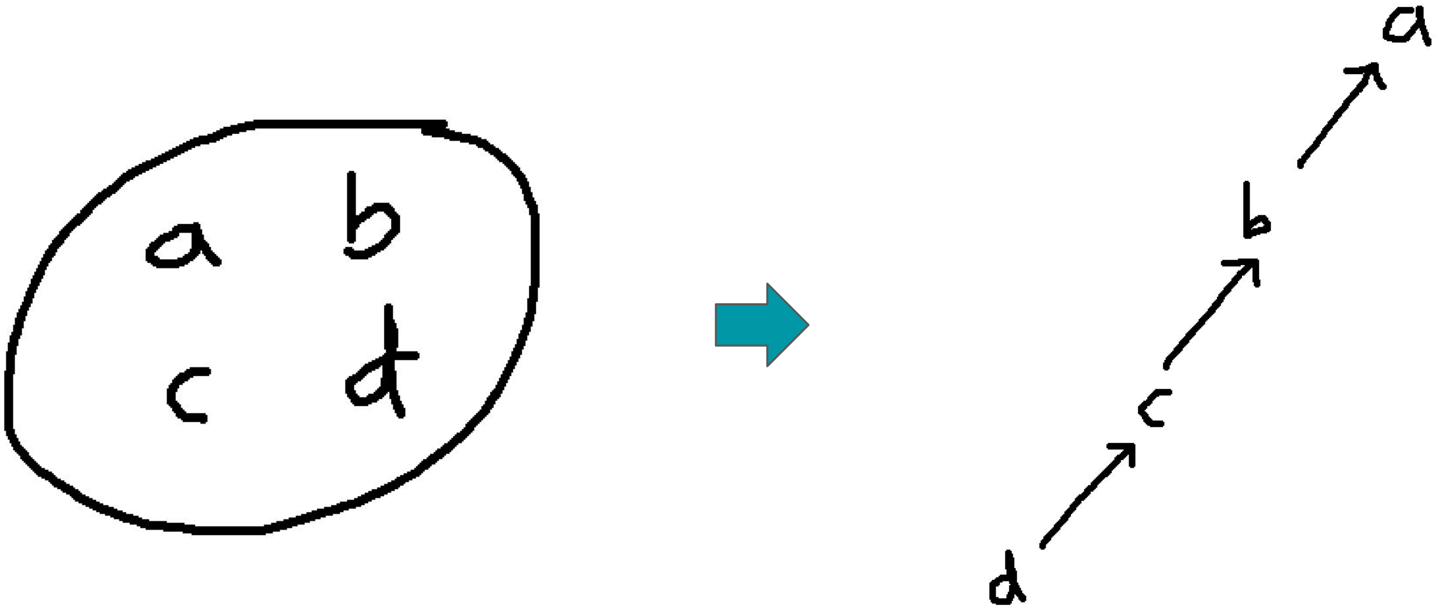
```cpp
int main() {
    DisjointSet s;
    s.merge("a", "b");
    cout << "a =? b: " << s.in_same_set("a", "b") << endl;
    cout << "b =? c: " << s.in_same_set("b", "c") << endl;
    s.merge("a", "c");
    cout << "b =? c: " << s.in_same_set("b", "c") << endl;
}
```

```
→ ./a.out
a =? b: 1
b =? c: 0
b =? c: 1
```

(**1** means **true**)

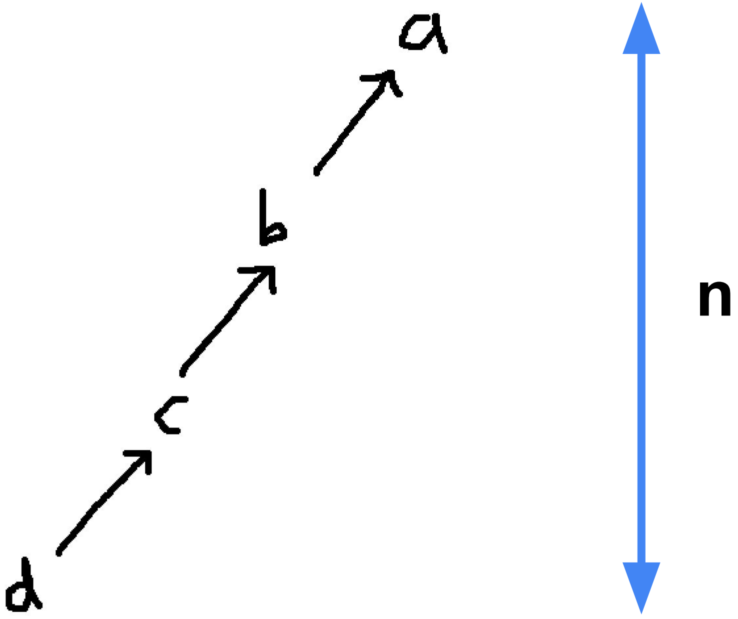
The Optimizations (pathological case)

Depending on how we **merge**, we may end up with this kind of “tree”:



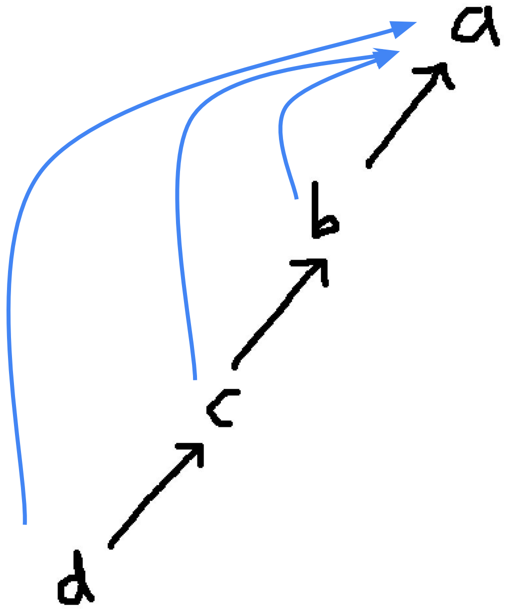
The Optimizations (pathological case)

It takes **linear time** to check if **a** and **d** are in the same set.



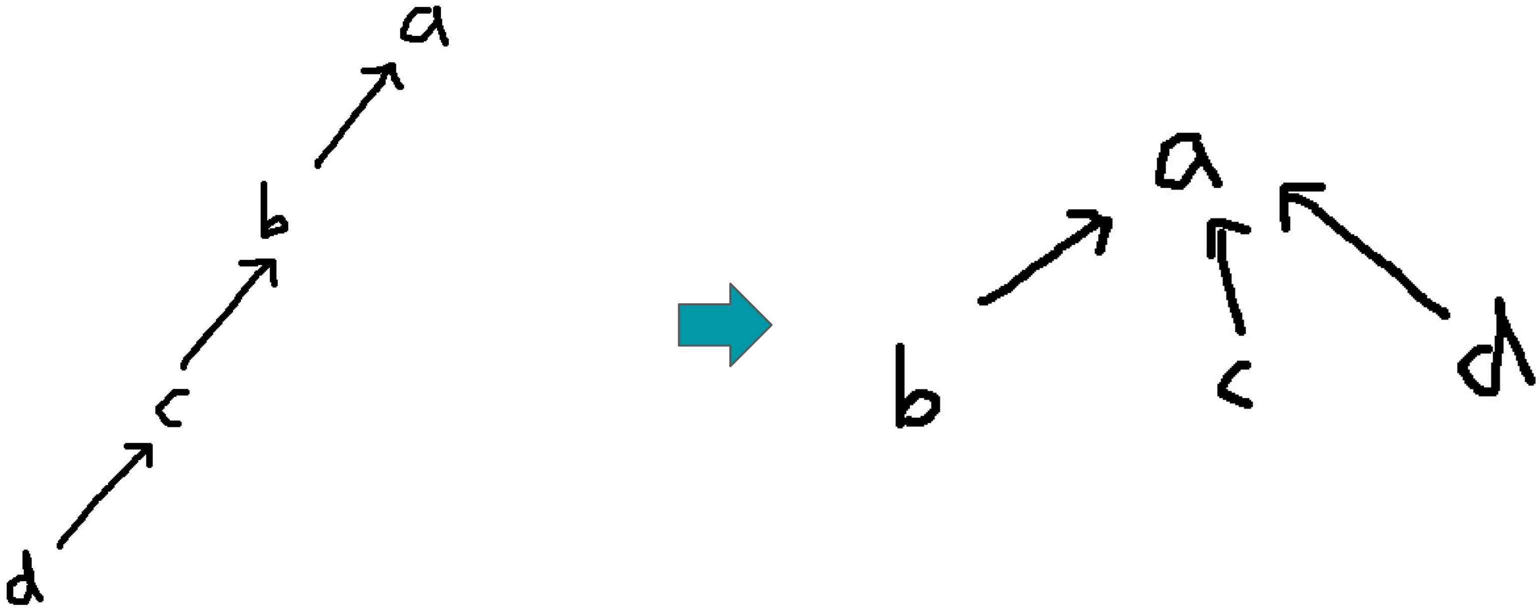
The Optimizations (path compression)

When we find the **representative** for **d**, we know that the **representative for all its ancestors** are the same.



The Optimizations (path compression)

So let's flatten this path!



The Optimizations (path compression)

Change this element's parent to the representative.

```
string find(string x) {
    if (parents.count(x)) {
        string parent = parents[x];
        string representative = find(parent);
        parents[x] = representative;
        return representative;
    } else {
        return x;
    }
}
```

The Optimizations (path compression)

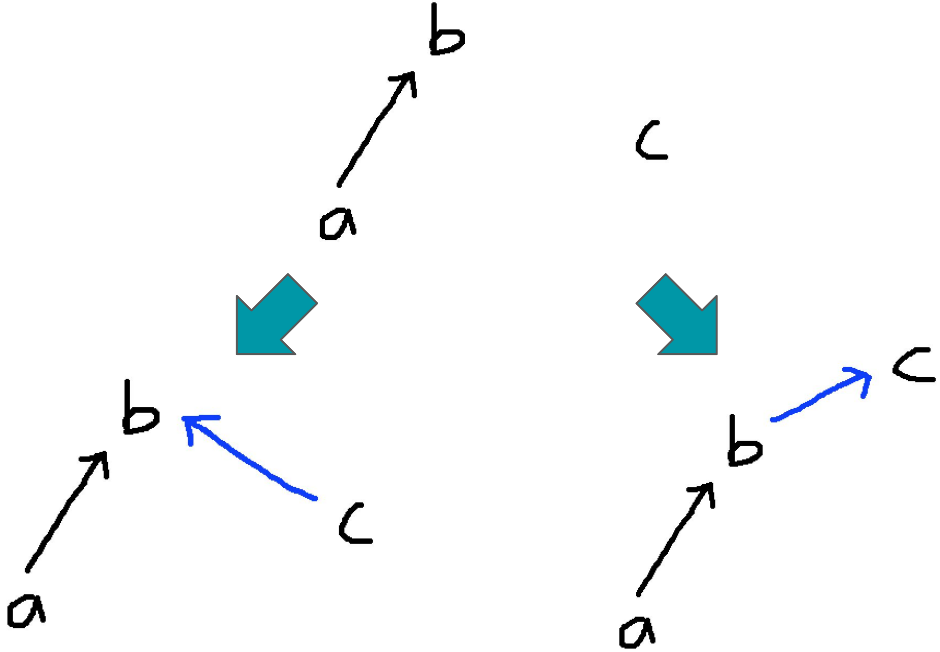
What's the time complexity of this new data structure?

It now takes **log n** time on average (amortized) for **find**.

Proof: hard

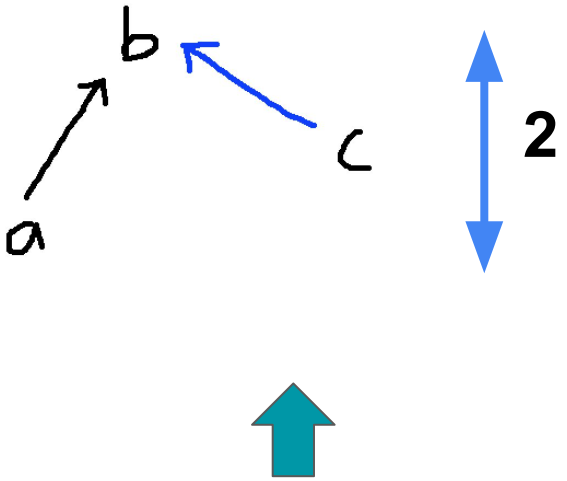
The Optimizations (union by size)

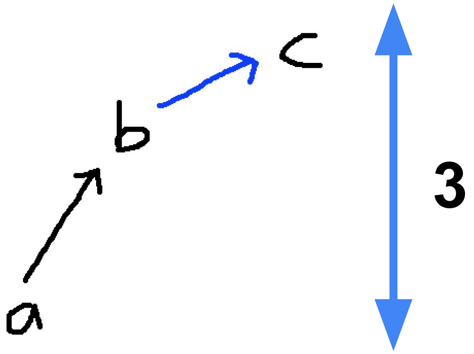
When we **merge** these two sets, which resulting tree is better?



The Optimizations (union by size)

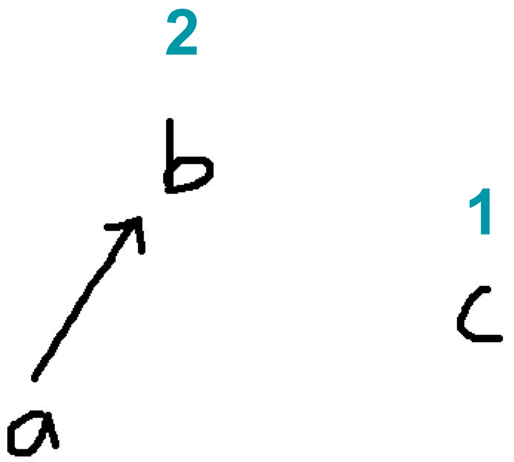
When we **merge** these two sets, which resulting tree is better?





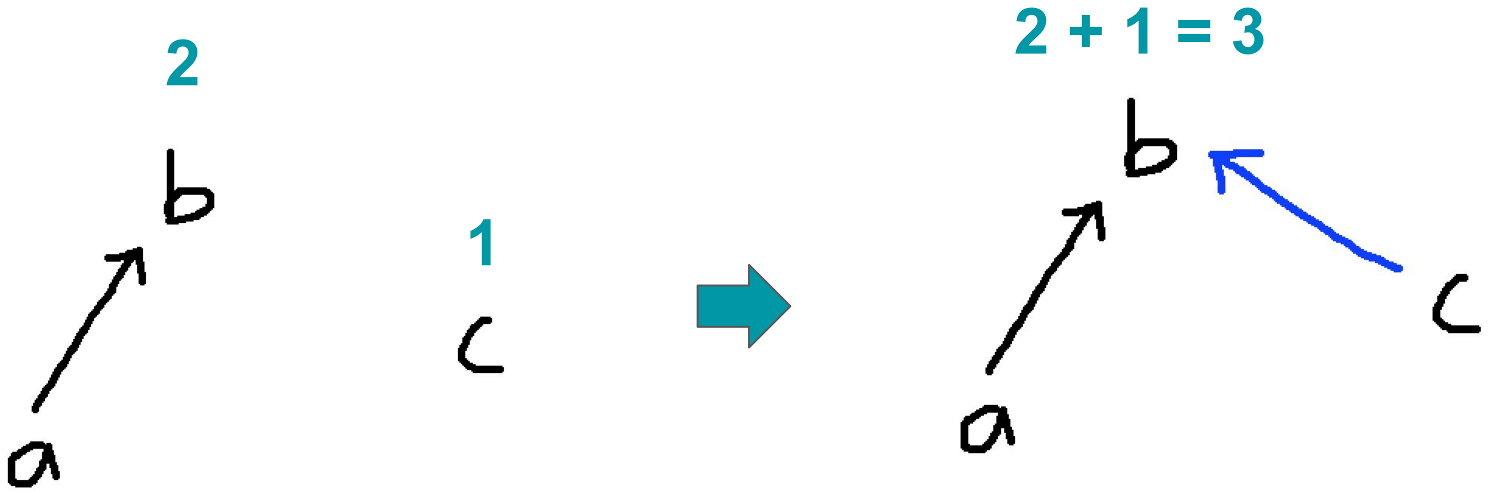
The Optimizations (union by size)

Let's store the **size** of each set in its representative.



The Optimizations (union by size)

We always point the **smaller set's representative** into the larger one's.



The Optimizations (union by size)

Store the size of each set by its representative element.

```
struct DisjointSet {
    map<string, string> parents;
    map<string, int> sizes;
```

Initialize the size of a set if it doesn't exist.

Point the smaller set's representative into the larger one's and update the set sizes.

```
void merge(string a, string b) {
    if (!sizes.count(a)) sizes[a] = 1;
    if (!sizes.count(b)) sizes[b] = 1;
    if (!in_same_set(a, b)) {
        string a_root = find(a);
        string b_root = find(b);
        if (sizes[b_root] < sizes[a_root]) {
            parents[a_root] = b_root;
            sizes[a_root] += sizes[b_root];
        } else {
            parents[b_root] = a_root;
            sizes[b_root] += sizes[a_root];
        }
    }
}
```

The Optimizations (union by size)

What's the time complexity of this new data structure?

It also takes **log n** time (in the worst case) for **find**.

Proof: in the worst case, it's a balanced binary tree.

The Optimizations

What if we combine the two optimizations?

- Path compression
- Union by size

What's the time complexity of this new data structure?

The Optimizations

What if we combine the two optimizations?

- Path compression
- Union by size

It takes **inverse Ackermann time** (practically constant) for **find**.

The Optimizations

What if we combine the two optimizations?

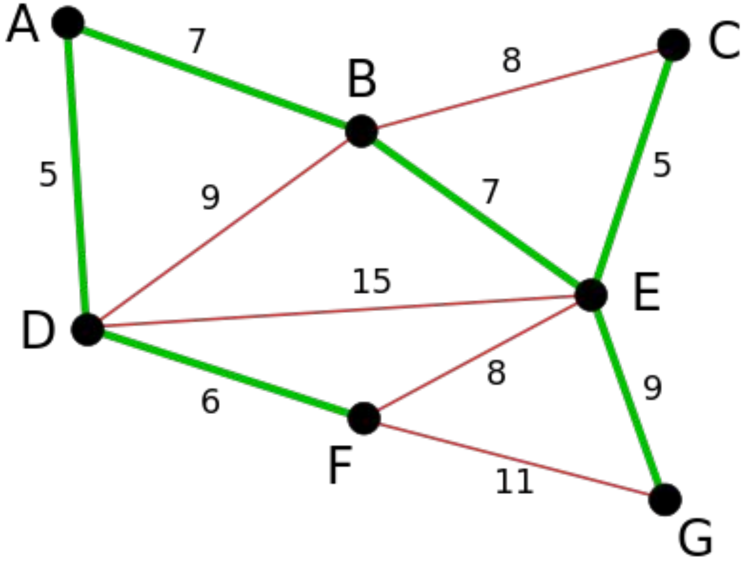
- Path compression
- Union by size

It takes **inverse Ackermann time** (practically constant) for **find**.

Proof:

The Applications (Kruskal's algorithm)

Kruskal's Algorithm finds a **minimum spanning tree** (tree connecting all nodes with the lowest total weight) on a graph.



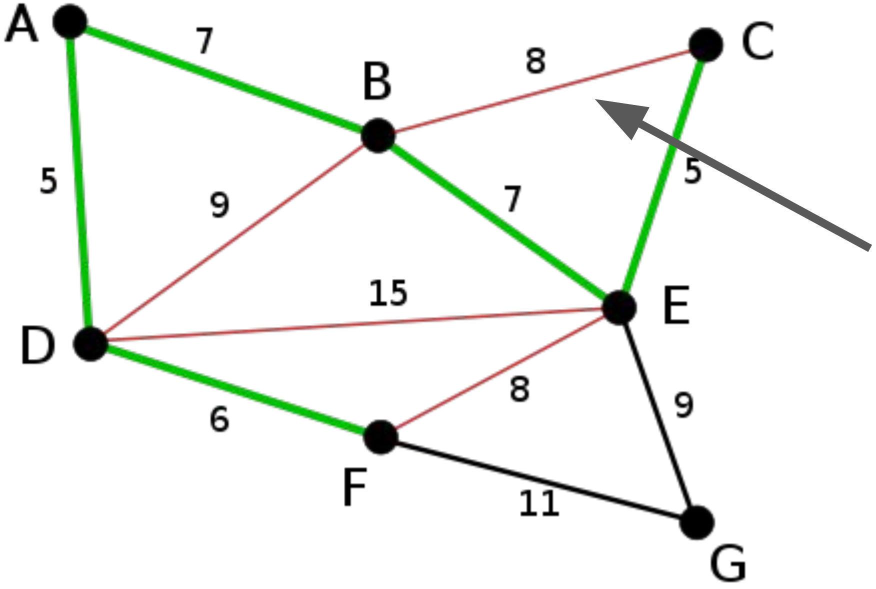
Source: Wikipedia

The Applications (Kruskal’s algorithm)

How it works:

1. Sort all edges by lowest weight first
2. For each edge:
    a. Check if the two nodes of the edge are connected
    b. If not, add the edge to the tree

The Applications (Kruskal's algorithm)



This is the next shortest edge but we don't add it because nodes **B** and **C** are already connected (through **E**).

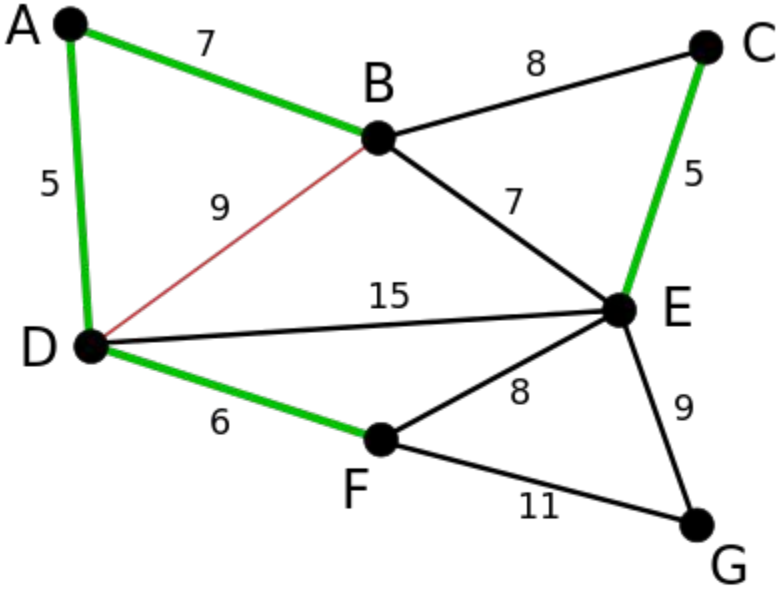
The Applications (Kruskal's algorithm)

How do we quickly check if two nodes are connected?

With a disjoint set!

Two nodes are connected if they are in the same set.

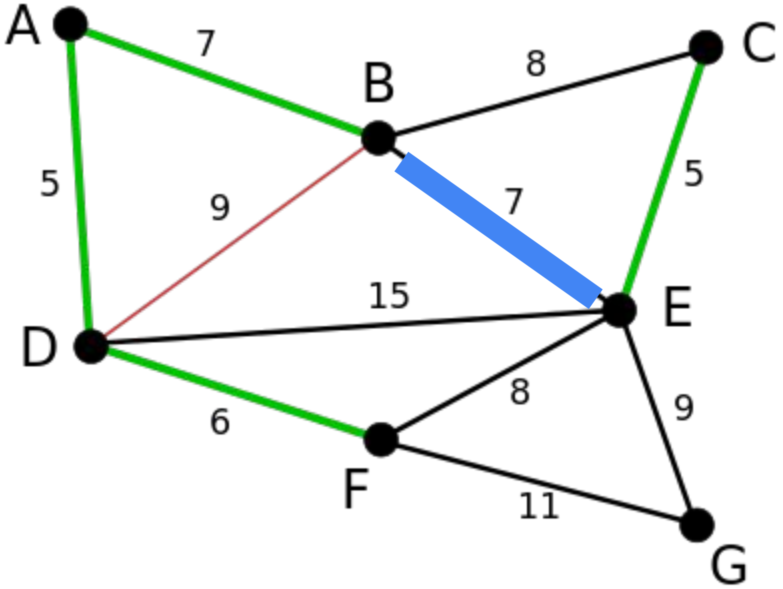
The Applications (Kruskal's algorithm)



Sets:

- {A, B, D, F}
- {C, E}

The Applications (Kruskal’s algorithm)



Sets:
- {A, B, D, F}
- {C, E}

Sets:
- {A, B, D, F, C, E}

The Applications (Kruskal's algorithm)

Sort edges by lowest weight first.

Add edge to tree only if the nodes aren't already connected.

```
// weight, start, end.
using Edge = tuple<int, string, string>;

vector<Edge> kruskals(vector<Edge> edges) {
    DisjointSet s;
    vector<Edge> tree;

    sort(edges.begin(), edges.end());
    for (Edge edge : edges) {
        int weight;
        string a, b;
        tie(weight, a, b) = edge;

        if (!s.in_same_set(a, b)) {
            s.merge(a, b);
            tree.push_back(edge);
        }
    }

    return tree;
}
```

The End

Resources

- Problems: Minimum spanning tree
    - https://www.hackerrank.com/challenges/kruskalmstrsub/problem
    - https://orac2.info/problem/aiio08trains/
    - https://orac2.info/problem/aiio13basmas/
- Problems: Disjoint set
    - https://dmoj.ca/problem/coci10c7p5
- Applications:
    - Kruskal’s algorithm
    - Hindley-Milner type inference

https://forms.gle/n1xKtBaxQp69fAsH6