# A new ADT

We need a new ADT!

## Disjoint-Sets (aka Union-Find) ADT

### state

- **Family of Sets**
  - **sets are disjoint:** No element appears in more than one set
  - No required order (neither within sets, nor between sets)
  - Each set has a name (usually one of its elements)

### behavior

- **makeSet(value)** – creates a new set where the only member is the value. Picks a name
- **findSet(value)** – looks up the name of the set containing value, returns the name of that set
- **union(x, y)** – looks up set containing x and set containing y, combines two sets into one. All of the values of one set are added to the other, and the now empty set goes away. Chooses a name for combined set
A better idea

Here’s a better idea:

We need to be able to combine things easily.
  - Pointer based data structures are better at that.

But given a value, we need to be able to find the right set.
  - Sounds like we need a dictionary somewhere

And we need to be able to find a certain element (“the representative”) within a set quickly.
  - Trees are good at that (better than linked lists at least)
The Real Implementation

**Disjoint-Set ADT**

**state**
- Set of Sets
  - **Disjoint**: Elements must be unique across sets
  - No required order
  - Each set has representative

**behavior**
- `makeSet(x)` – creates a new set within the disjoint set where the only member is x. Picks representative for set
- `findSet(x)` – looks up the set containing element x, returns representative of that set
- `union(x, y)` – looks up set containing x and set containing y, combines two sets into one. Picks new representative for resulting set

**UpTreeDisjointSet<E>**

**state**
- `Collection<TreeSet> forest`
- `Dictionary<NodeValues, NodeLocations> nodeInventory`

**behavior**
- `makeSet(x)`– create a new tree of size 1 and add to our forest
- `findSet(x)`– locates node with x and moves up tree to find root
- `union(x, y)`– append tree with y as a child of tree with x

**TreeSet<E>**

**state**
- `SetNode overallRoot`

**behavior**
- `TreeSet(x)`
- `add(x)`
- `remove(x, y)`
- `getRep()`– returns data of overallRoot

**SetNode<E>**

**state**
- `E data`
- `Collection<SetNode> children`

**behavior**
- `SetNode(x)`
- `addChild(x)`
- `removeChild(x, y)`
Implement `makeSet(x)`

Worst case runtime? Just like with graphs, we’re going to assume we have control over the dictionary keys and just say we’ll always have $\Theta(1)$ dictionary behavior.

$O(1)$
Implement union(x, y)

union(3, 5)

TreeDisjointSet<E>

state
Collection<TreeSet> forest
Dictionary<NodeValues, NodeLocations> nodeInventory

behavior
makeSet(x) - create a new tree of size 1 and add to our forest
findSet(x) - locates node with x and moves up tree to find root
union(x, y) - append tree with y as a child of tree with x
Implement union(x, y)

union(3, 5)
union(2, 1)

TreeDisjointSet<E>

state
Collection<TreeSet> forest
Dictionary<NodeValues, NodeLocations> nodeInventory

behavior
makeSet(x) - create a new tree of size 1 and add to our forest
findSet(x) - locates node with x and moves up tree to find root
union(x, y) - append tree with y as a child of tree with x
Implement union(x, y)

union(3, 5)
union(2, 1)
union(2, 5)
Implement `union(x, y)`

- `union(3, 5)`
- `union(2, 1)`
- `union(2, 5)`

#### TreeDisjointSet<T>

**State**
- `Collection<TreeSet>` forest
- `Dictionary<NodeValues, NodeLocations>` nodeInventory

**Behavior**
- `makeSet(x)` - create a new tree of size 1 and add to our forest
- `findSet(x)` - locates node with x and moves up tree to find root
- `union(x, y)` - append tree with y as a child of tree with x
Implement `findSet(x)`

`findSet(0)`
`findSet(3)`
`findSet(5)`

Worst case runtime of `findSet`?
\( \Theta(n) \)

Worst case runtime of `union`?
\( \Theta(n) \) – `union` has to call `find`!

**TreeDisjointSet<\(E\)>**

**state**
- `Collection<TreeSet>` `forest`
- `Dictionary<NodeValues, NodeLocations>` `nodeInventory`

**behavior**
- `makeSet(x)` - create a new tree of size 1 and add to our forest
- `findSet(x)` - locates node with `x` and moves up tree to find root
- `union(x, y)` - append tree with `y` as a child of tree with `x`
Improving union

Problem: Trees can be unbalanced

Solution: Union-by-rank!
- rank is a lot like height (it’s not quite height, for reasons we’ll see soon)
- Keep track of rank of all trees
- makeSet creates a tree of rank 0.
- When unioning make the tree with larger rank the root. New rank is larger of two merged ranks.
- If it’s a tie, pick one to be root arbitrarily and increase rank by one.
Practice

Given the following disjoint-set what would be the result of the following calls on union if we add the “union-by-rank” optimization. Draw the forest at each stage with corresponding ranks for each tree.

union(2, 13)
union(4, 12)
union(2, 8)
Practice

Given the following disjoint-set what would be the result of the following calls on union if we add the “union-by-rank” optimization. Draw the forest at each stage with corresponding ranks for each tree.

union(2, 13)
Practice

Given the following disjoint-set what would be the result of the following calls on union if we add the “union-by-rank” optimization. Draw the forest at each stage with corresponding ranks for each tree.

union(2, 13)
union(4, 12)
Practice

Given the following disjoint-set what would be the result of the following calls on union if we add the “union-by-rank” optimization. Draw the forest at each stage with corresponding ranks for each tree.

union(2, 13)
union(4, 12)
union(2, 8)
Practice

Given the following disjoint-set what would be the result of the following calls on union if we add the “union-by-rank” optimization. Draw the forest at each stage with corresponding ranks for each tree.

union(2, 13)
union(4, 12)
union(2, 8)

Does this improve the worst case runtimes?

findSet is $\Theta(\log(n))$ now, not $\Theta(n)$!
Improving `findSet()`

**Problem:** Every time we call `findSet()` you must traverse all the levels of the tree to find representative

**Solution:** Path Compression
- Collapse tree into fewer levels by updating parent pointer of each node you visit
- Whenever you call `findSet()` update each node you touch’s parent pointer to point directly to `overallRoot`

`findSet(5)`

`findSet(4)`

Does this improve the worst case runtimes?

Not the worst-case, but...in-practice it makes a big difference.
Example

Using the union-by-rank and path-compression optimized implementations of disjoint-sets draw the resulting forest caused by these calls:

1. makeSet(a)
2. makeSet(b)
3. makeSet(c)
4. makeSet(d)
5. makeSet(e)
6. makeSet(f)
7. makeSet(g)
8. makeSet(h)
9. union(c, e)
10. union(d, e)
11. union(a, c)
12. union(g, h)
13. union(b, f)
14. union(g, f)
15. union(b, c)
Using the union-by-rank and path-compression optimized implementations of disjoint-sets draw the resulting forest caused by these calls:

1. makeSet(a)
2. makeSet(b)
3. makeSet(c)
4. makeSet(d)
5. makeSet(e)
6. makeSet(f)
7. makeSet(g)
8. makeSet(h)
9. union(c, e)
10. union(d, e)
11. union(a, c)
12. union(g, h)
13. union(b, f)
14. union(g, f)
15. union(b, c)
16. union(g, a)
Subtleties of Path Compression

Path compression is an optimization written into the `findSet` code.

It does not appear directly in the `union` code.
- It’s not worth it; you’d have to rewrite the entire `findSet` code inside `union` to make it happen.

But `union` does make two `findSet` calls,
- So path compression will happen when you do a `union` call, just indirectly.
# Optimized Up-trees Runtimes

<table>
<thead>
<tr>
<th></th>
<th>makeSet</th>
<th>findSet</th>
<th>Union</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worst-Case</td>
<td>$\Theta(1)$</td>
<td>$\Theta(\log n)$</td>
<td>$\Theta(\log n)$</td>
</tr>
<tr>
<td>Best-Case</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>In-Practice</td>
<td>$\Theta(1)$</td>
<td>$O(\log^* n)$</td>
<td>$O(\log^* n)$</td>
</tr>
</tbody>
</table>

Hey why are some of those $O()$ not $\Theta()$? And...wait what’s that * above the log?
\( \log^* (n) \)

\( \log^* (n) \) is the “iterated logarithm”

It answers the question “how many times do I have to take the log of this to get a number at most 1?”

E.g. \( \log^* (16) = 3 \)

\( \log(16) = 4 \) \hspace{10mm} \( \log(4) = 2 \) \hspace{10mm} \( \log(2) = 1 \).

\( \log^* n \) grows ridiculously slowly.

\( \log^* (10^{80}) = 5 \).

\( 10^{80} \) is the number of atoms in the observable universe. For all practical purposes these operations are constant time.

But they aren’t \( O(1) \).
Optimized Up-tree Runtimes

$\log^* n$ isn’t tight – that’s why those $\Theta()$ bounds became $O()$ bounds.

There is a tight bound. It’s a function that grows even slower than $\log^* n$
- Google “inverse Ackerman function”
Kruskal’s Algorithm

KruskalMST(Graph G)
initialize each vertex to be its own component
sort the edges by weight
foreach(edge (u, v) in sorted order){
    if(u and v are in different components){
        add (u,v) to the MST
        Update u and v to be in the same component
    }
}
What’s the running time of Kruskal’s?

For MST algorithms, assume that \( m \) dominates \( n \)
(if it doesn’t, there is no spanning tree to find)

\[
\text{KruskalMST(Graph G)} \\
\text{initialize new DisjointSets DS} \\
\text{for}(v : G.\text{vertices}) \{ \text{DS.makeSet}(v) \} \\
\text{sort the edges by weight} \quad \Theta(\log m) \\
\text{foreach(edge (u, v) in sorted order)} \{ \\
\quad \text{if}(\text{DS.findSet}(u) \neq \text{DS.findSet}(v)) \{ \\
\quad \quad \text{add } (u, v) \text{ to the MST} \\
\quad \quad \text{DS.union}(u, v) \quad n \text{ calls, do we have to worry} \\
\quad \quad \text{about the } \log n \text{ worst case?} \\
\quad \} \\
\}
\]

Intuition: We could make the \( \log n \) running time happen once...but not really more than that.
Since we’re counting total operations, we’re actually going to see the “in-practice” behavior

Whether we hit worst-case or not: \( \Theta(m \log m) \) is dominating term.
Running Time Notes

Intuition: We could make the bad case happen once...but not really more than that. Since we’re counting total operations, we’re actually going to see “in-practice” behavior.

This kind of statement is “amortized analysis”
- It’s also the math behind why we always double the size of array-based data structures.

Some people write the running time as $\Theta(m \log n)$ instead of $\Theta(m \log m)$
They’re assuming the graph doesn’t have any multi-edges.
- I.e. there’s at most one edge between any pair of vertices.
And they just think $\Theta(m \log n)$ looks better (even though it’s just a constant factor)