Disjoint Sets
Warm Up

Finding a MST using Kruskal’s algorithm
Warm Up

Finding a MST using Kruskal’s algorithm
New ADT

Set ADT

**state**
- Set of elements
  - Elements must be unique!
  - No required order

**behavior**
- `create(x)` - creates a new set with a single member, x
- `add(x)` - adds x into set if it is unique, otherwise add is ignored
- `remove(x)` - removes x from set
- `size()` - returns current number of elements in set

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
Example

new()
makeSet(a)
makeSet(b)
makeSet(c)
makeSet(d)
makeSet(e)
findSet(a)  0 3
findSet(d)
union(a, c)
Example

c
new()
makeset(a)
makeset(b)
makeset(c)
makeset(d)
makeset(e)
findset(a)
findset(d)
union(a, c)
union(b, d)
**Example**

```plaintext
new()
makeSet(a)
makeSet(b)
makeSet(c)
makeSet(d)
makeSet(e)
findSet(a)
findSet(d)
union(a, c)
union(b, d)
findSet(a) == findSet(c)
findSet(a) == findSet(d)
```
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

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`

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)

- makeSet(0)
- makeSet(1)
- makeSet(2)
- makeSet(3)
- makeSet(4)
- makeSet(5)

Worst case runtime?

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

union(3, 5)

- 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

```plaintext
forest

0 -> 1
3
2
4
5

0 1 2 3 4 5
```
Implement union(x, y)

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

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

```plaintext
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)

```
CSE 373 SP 18 - KASEY CHAMPION
```

```java
TreeDisjointSet<E> 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?
- $O(n)$

Worst case runtime of union?
- $O(n)$
Improving union

Problem: Trees can be unbalanced

Solution: Union-by-rank!
- let rank(x) be a number representing the upper bound of the height of x so rank(x) >= height(x)
- Keep track of rank of all trees
- When unioning make the tree with larger rank the root
- If it’s a tie, pick one randomly and increase rank by one
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.

1. union(2, 13)
2. union(4, 12)
3. 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.

\[ \text{rank} = 3 \]

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

Does this improve the worst case runtimes?

findSet is more likely to be \( O(\log(n)) \) than \( O(n) \)
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

\[
\text{rank} = 3
\]

\[
\text{findSet}(5)
\]

\[
\text{findSet}(4)
\]

Does this improve the worst case runtimes?

\[
\text{findSet is more likely to be } O(1) \text{ than } O(\log(n))
\]
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(h)
8. union(c, e)
9. union(d, e)
10. union(a, c)
11. union(g, h)
12. union(b, f)
13. union(g, f)
14. union(b, c)
Array Representation

Like heaps, pretend the tree exists, but use an Array for actual implementation