Kruskal’s Algorithm Implementation

KruskalMST(Graph G)
initialize each vertex to be an independent 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
  }
}

KruskalMST(Graph G)
foreach (V : vertices) {
  makeMST(v); +?
}
sort edges in ascending order by weight
foreach(edge (u, v)){
  if(findMST(v) is not in findMST(u)){
    union(u, v) +?
  }
}

How many times will we call union?
V – 1
-> +Vunion + EfindMST
### New ADT

#### Set ADT

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

**Count of Elements**

**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

**Count of Sets**

**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)
findset(d)
union(a, c)
Example

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

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)
Implementation

Disjoint-Set ADT

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

**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<E>> 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)

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)
Implement union(x, y)

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

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

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

```java
TreeDisjointSet<E>

state
Collection<Forest> 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?
\( O(n) \)

Worst case runtime of union?
\( O(n) \)
Improving union

**Problem:** Trees can be unbalanced

**Solution:** Union-by-rank!
- let \( \text{rank}(x) \) be a number representing the upper bound of the height of \( x \) so \( \text{rank}(x) \geq \text{height}(x) \)
- 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.

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 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.

`findSet(5)`
`findSet(4)`

Does this improve the worst case runtimes?

`findSet` is more likely to be $O(1)$ 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)
## Optimized Disjoint Set Runtime

<table>
<thead>
<tr>
<th>Function</th>
<th>Without Optimizations</th>
<th>With Optimizations</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>makeSet(x)</code></td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td><code>findSet(x)</code></td>
<td>$O(n)$</td>
<td><strong>Best case: $O(1)$ Worst case: $O(\log n)$</strong></td>
</tr>
<tr>
<td><code>union(x, y)</code></td>
<td>$O(n)$</td>
<td><strong>Best case: $O(1)$ Worst case: $O(\log n)$</strong></td>
</tr>
</tbody>
</table>
Worksheet question 1

1: function Kruskal(Graph G)
2:     initialize each vertex to be a component
3:     sort all edges by weight
4:     for each edge (u, v) in sorted order do
5:         if u and v are in different components then
6:             add edge (u,v) to the MST
7:             update u and v to be in the same component
8:         end if
9:     end for
10: end function
Worksheet question 1

1: function Kruskal(Graph G)
2: initialize a disjoint set; call makeSet() on each vertex
3: sort all edges by weight
4: for each edge (u, v) in sorted order do
5:   if findSet(u) ≠ findSet(v) then
6:     add edge (u,v) to the MST
7:     union(u, v)
8:   end if
9: end for
10: end function
Implementation

Use Nodes?

In modern Java (assuming 64-bit JDK) each object takes about 32 bytes
- int field takes 4 bytes
- Pointer takes 8 bytes
- Overhead ~ 16 bytes
- Adds up to 28, but we must partition in multiples of 8 => 32 bytes

Use arrays instead!
- Make index of the array be the vertex number
  - Either directly to store ints or representationally
  - We implement makeSet(x) so that we choose the representative
- Make element in the array the index of the parent
Array implementation

rank = 0

rank = 3

rank = 3
Array implementation

rank = 0

rank = 3

rank = 3

Store (rank * -1) - 1
Example

Consider the following disjoint set. Assume that (from left) the first tree has rank 3, the second has rank 0, the third has rank 1, and the last tree has rank 1.

Write the array representation of this disjoint set in the array below.

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
</table>
```
Example

Consider the following disjoint set. Assume that (from left) the first tree has rank 3, the second has rank 0, the third has rank 1, and the last tree has rank 1.

Write the array representation of this disjoint set in the array below.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>2</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>-2</td>
<td>-4</td>
<td>3</td>
<td>3</td>
<td>-1</td>
<td>4</td>
<td>4</td>
<td>14</td>
<td>-2</td>
<td>14</td>
</tr>
</tbody>
</table>
Array method implementation

\textbf{makeSet}(x)

add new value to array with a rank of -1

\textbf{findSet}(x)

Jump into array at index/value you’re looking for, jump to parent based on element at that index, continue until you hit negative number

\textbf{union}(x, y)

\textbf{findSet}(x) and \textbf{findSet}(y) to decide who has larger rank, update element to represent new parent as appropriate
Graph Review

Graph Definitions/Vocabulary
- Vertices, Edges
- Directed/undirected
- Weighted
- Etc...

Graph Traversals
- Breadth First Search
- Depth First Search

Finding Shortest Path
- Dijkstra’s

Topological Sort, Strongly connected components

Minimum Spanning Trees
- Primm’s
- Kruskal’s

Disjoint Sets
- Implementing Kruskal’s