Honors Data Structures

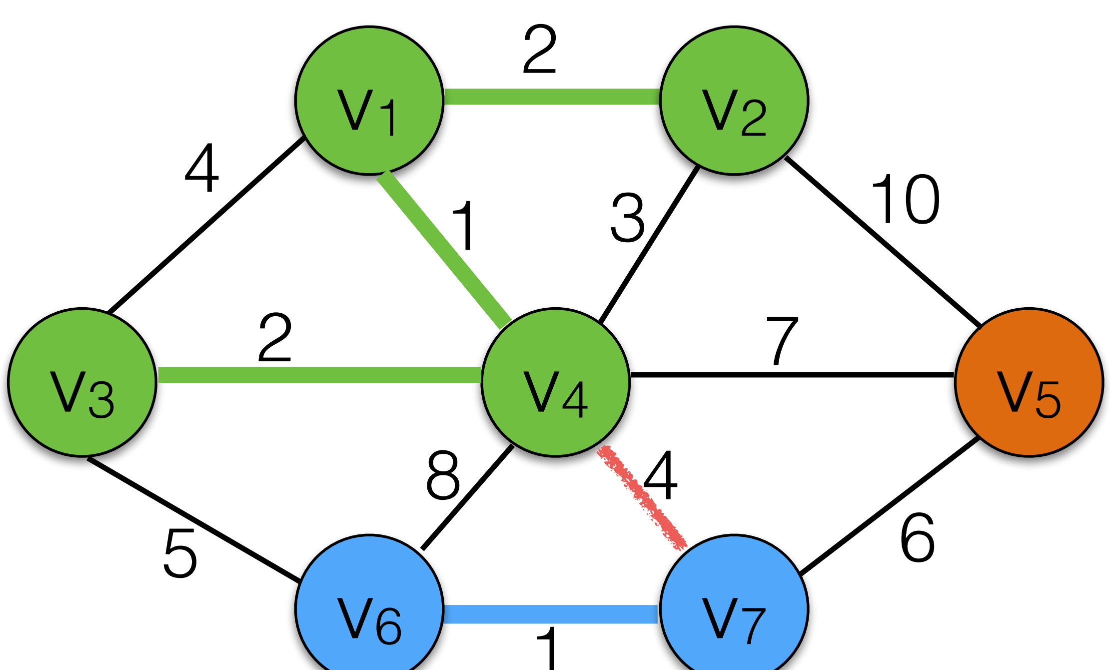
Lecture 23: Disjoint Set Data Structures

4/18/2022

Daniel Bauer

Kruskal’s Algorithm for finding MSTs

- Kruskal’s algorithm maintains a “forest” of trees.
- Initially each vertex is its own tree.
- Sort edges by weight. Then attempt to add them one-by one. Adding an edge merges two trees into a new tree.
- If an edge connects two nodes that are already in the same tree it would produce a cycle. Reject it.



Disjoint Set Data Structure

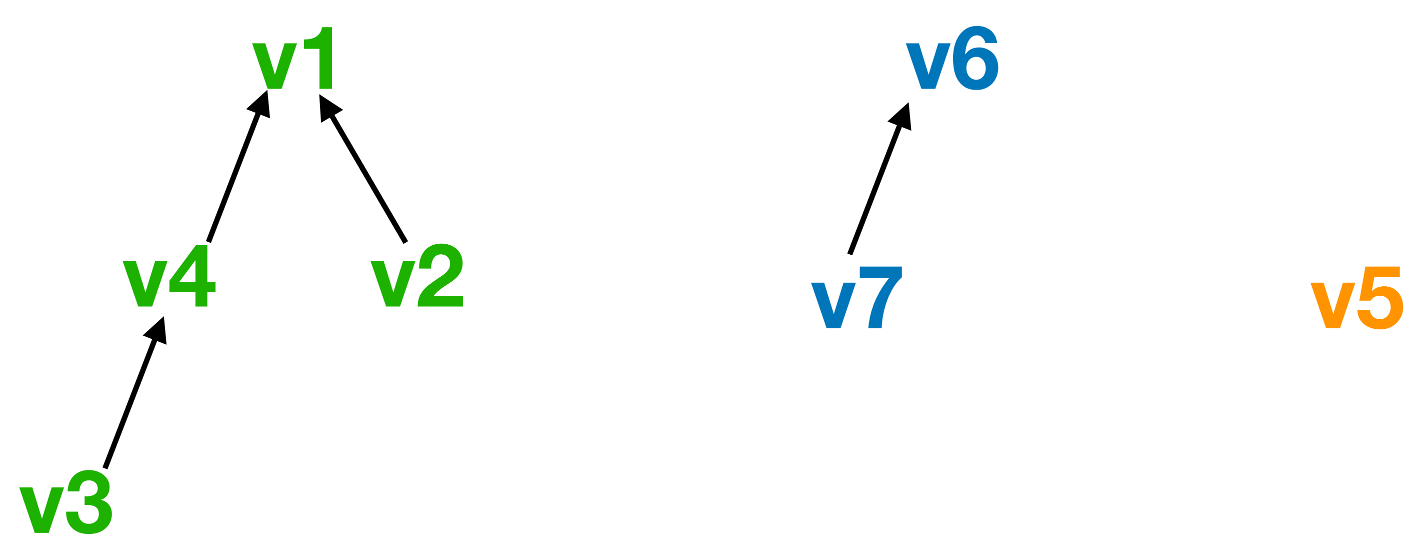
- a.k.a. Union/Find Data Structure
- Represent a collection of disjoint sets (i.e. no overlap).

{v1, v2, v3, v4}, {v6,v7}, {v5}

- Efficiently supports the following operations:
  - add a new set
  - **find** which set an item belongs to (by returning a **representative**)
  - **union/merge** two sets

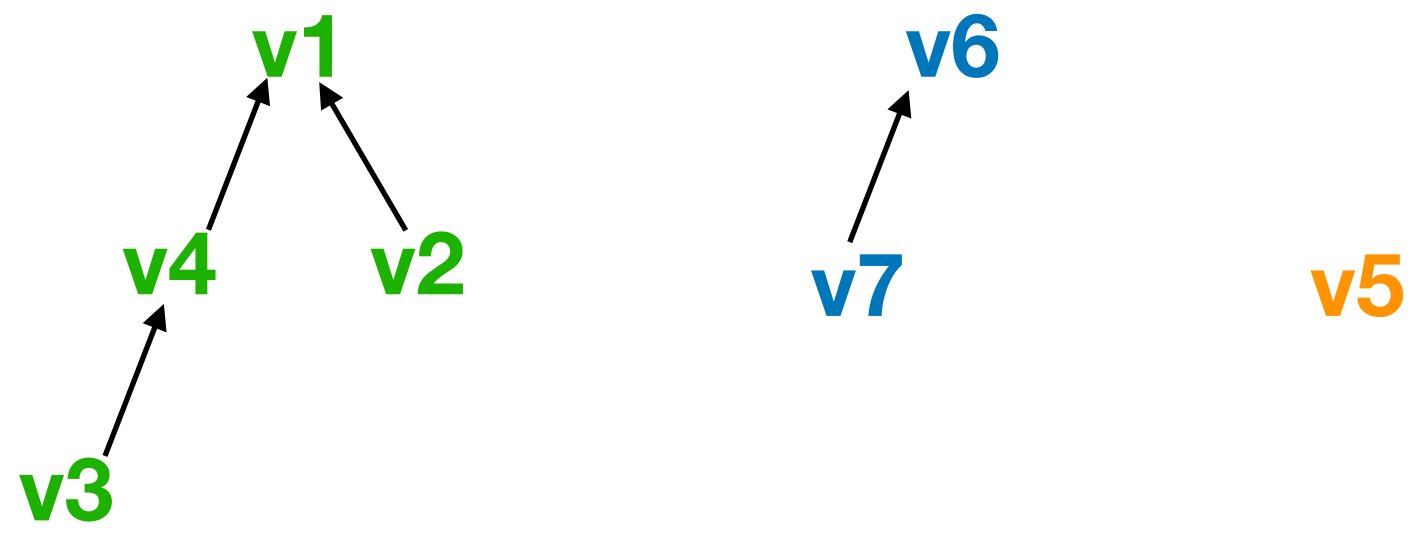
Disjoint Set Forest

- Represent the disjoint sets as a **forest.**
- Each tree represents a set. The root of each tree is the representative. Each node storing an item has a parent reference.



Find

```
public Node find(Node node){
     if (node.parent == null)
         return node;
     return find(node.parent);
}
```

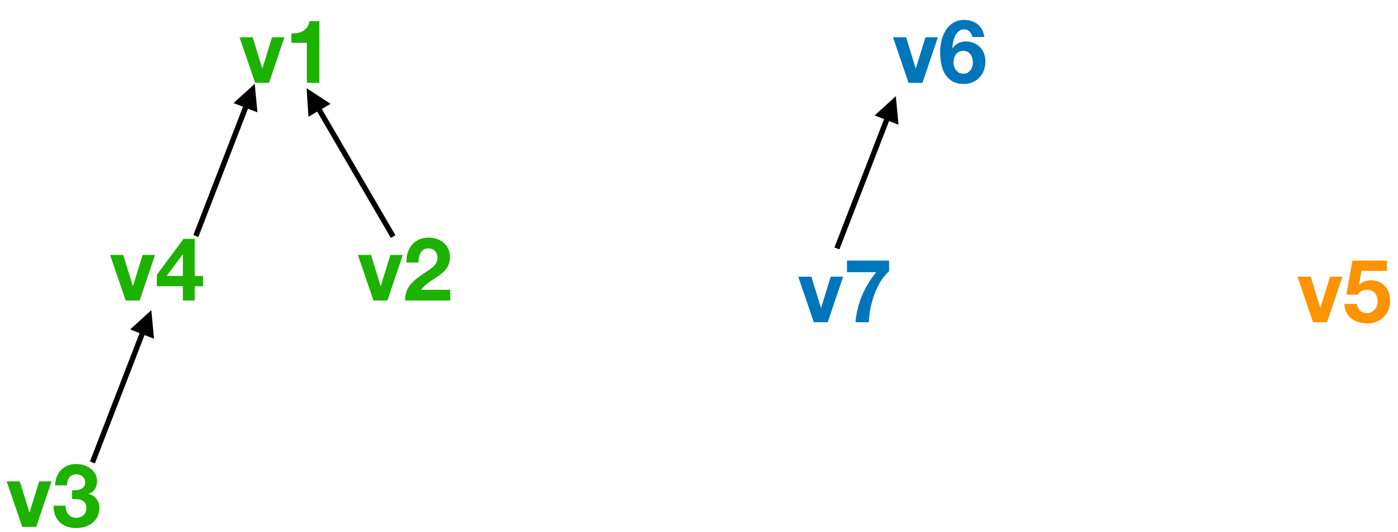


Union

```
public Node union(Node x, Node y){
    x = Find(x);
    y = Find(y);

    x.parent = y;

}
```



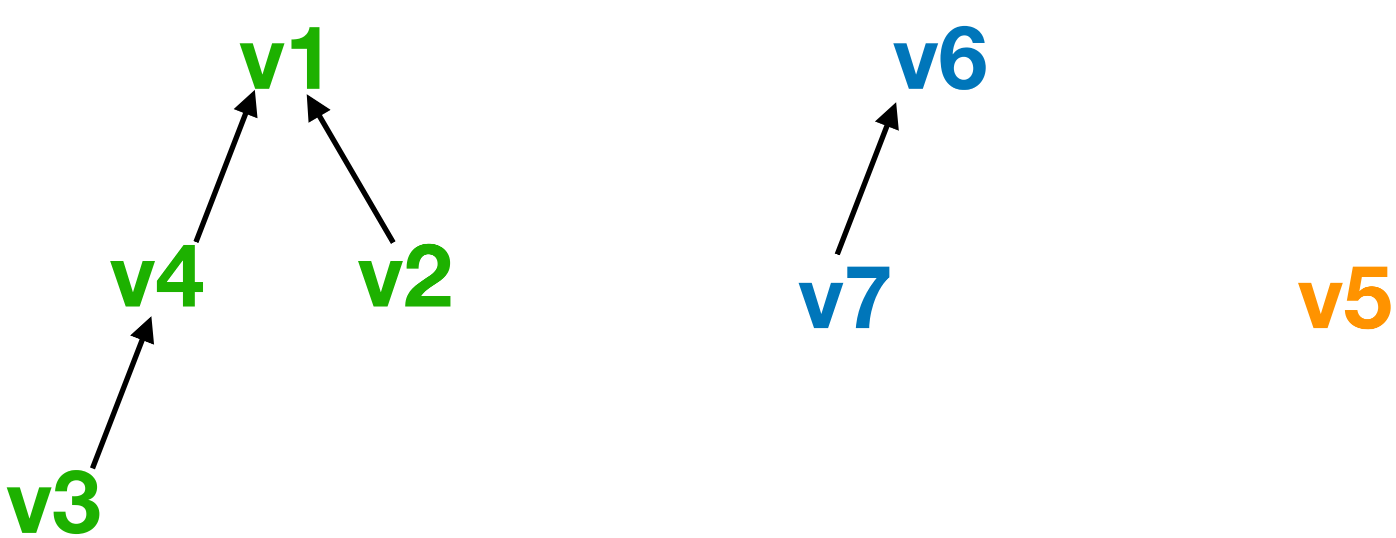
Union

```
public Node union(Node x, Node y){
     x = Find(x);
     y = Find(y);

     if (x!=y)
       x.parent = y;

}
```

union (v4, v7)

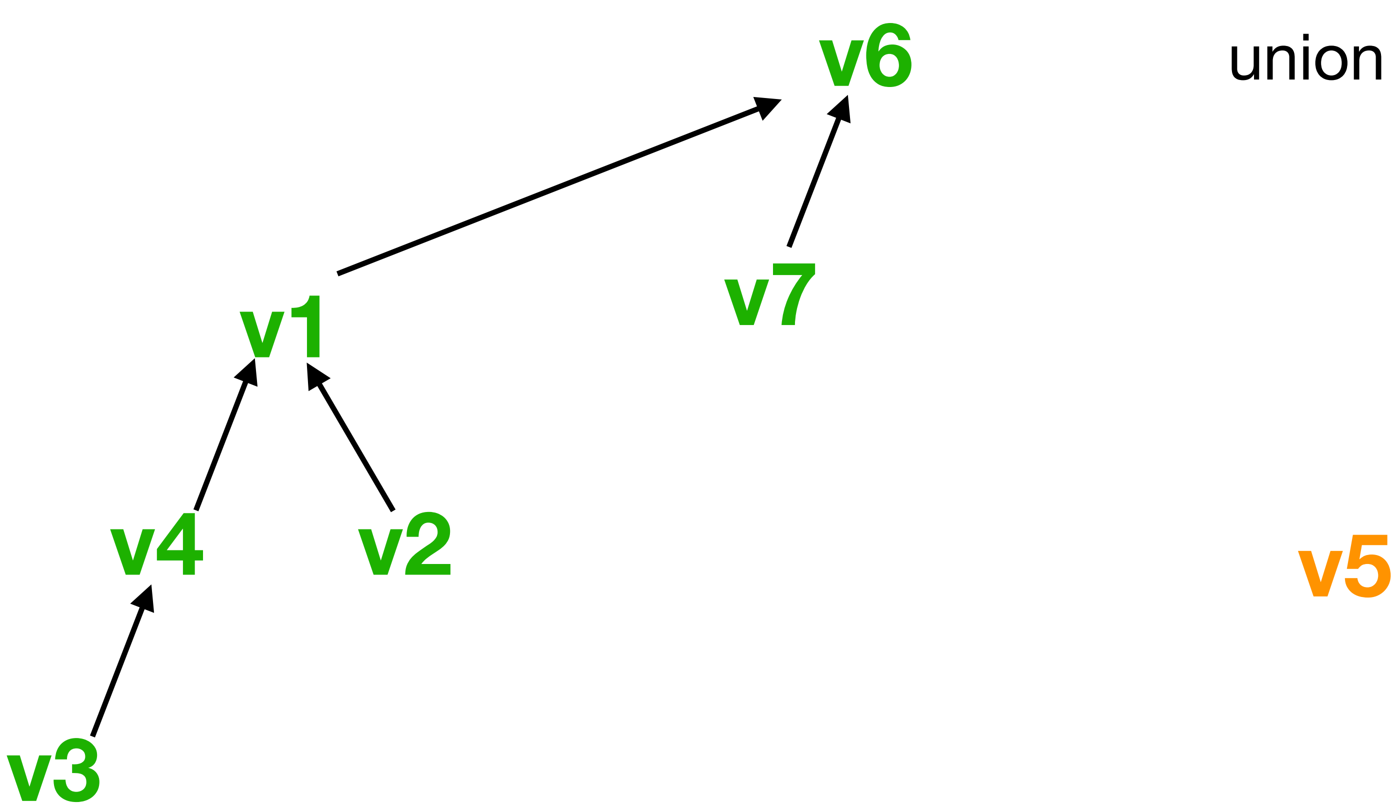


Union

```
public Node union(Node x, Node y){
    x = Find(x);
    y = Find(y);

    if (x != y)
       x.parent = y;

}
```



union (v4, v7)

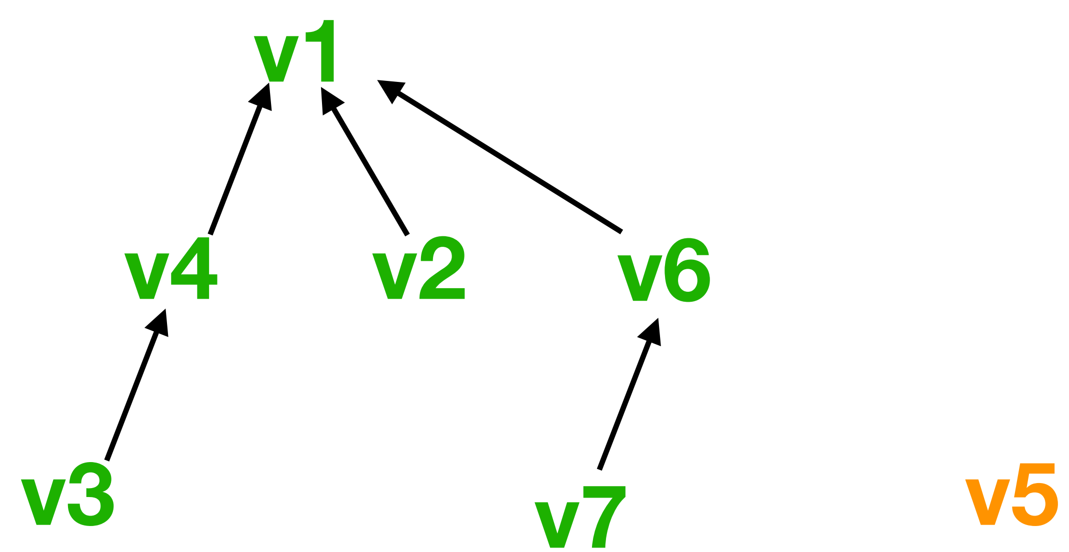
Union by Size

- Time required for find depends on the height of the trees, so we want to keep the trees shallow.
- Larger tree should become the parent.

```
public Node union(Node x, Node y){
    x = Find(x);
    y = Find(y);

    if x.size < y.size){
        Node tmp = x;
        x = y;
        y = tmp;
    }
    y.parent = x;
    x.size = x.size + y.size

}
```

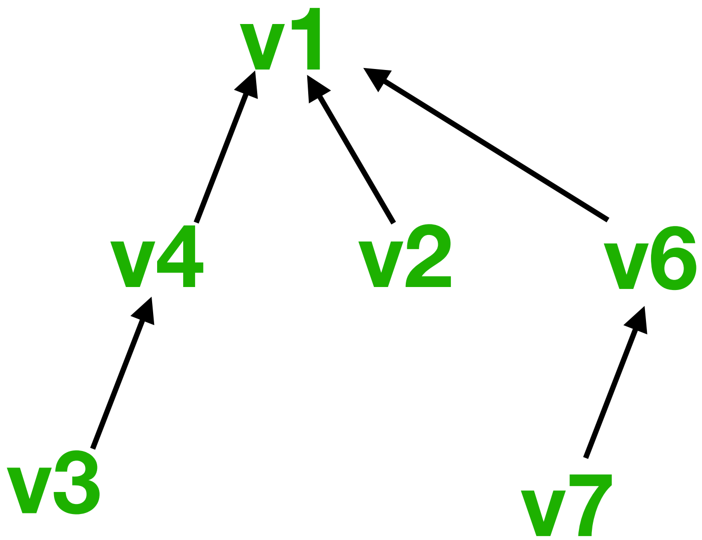


Path Compression

- We want to minimize the number of steps from each entry to the root.

- Modify find to make all nodes on the path children of the root.

```
public Node find(Node node){
     if (node.parent == null)
         return node;
     return node.parent = find(node.parent);
}
```

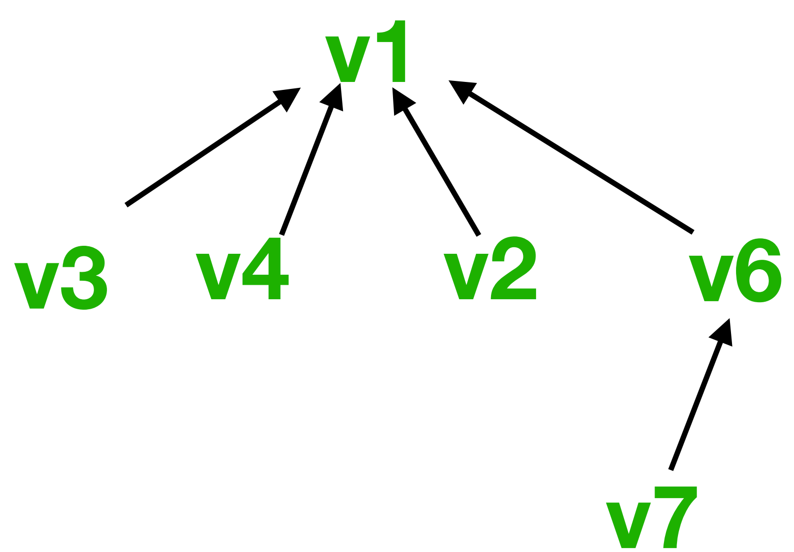


find(v3)

Path Compression

- We want to minimize the number of steps from each entry to the root.
- Modify find to make all nodes on the path children of the root.

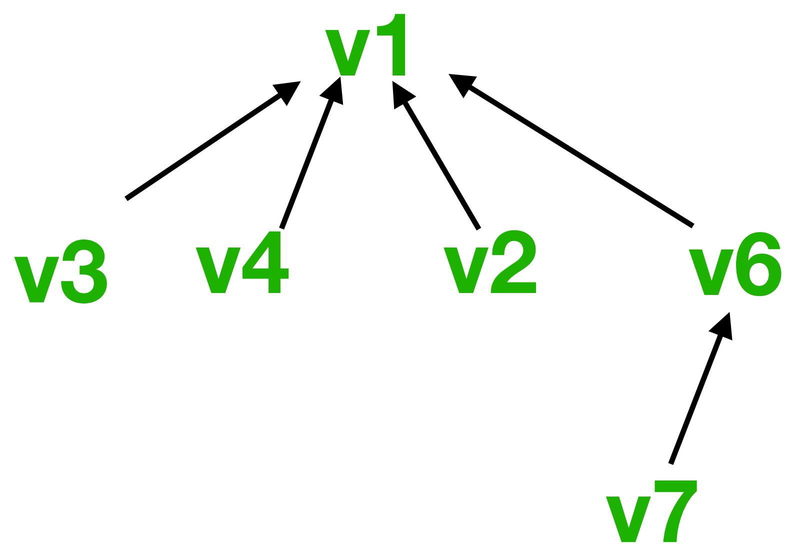
```
public Node find(Node node){
     if (node.parent == null)
         return node;
     return node.parent = find(node.parent);
}
```



Path Compression

- We want to minimize the number of steps from each entry to the root.
- Modify find to make all nodes on the path children of the root.

```
public Node find(Node node){
     if (node.parent == null)
         return node;
     return node.parent = find(node.parent);
}
```

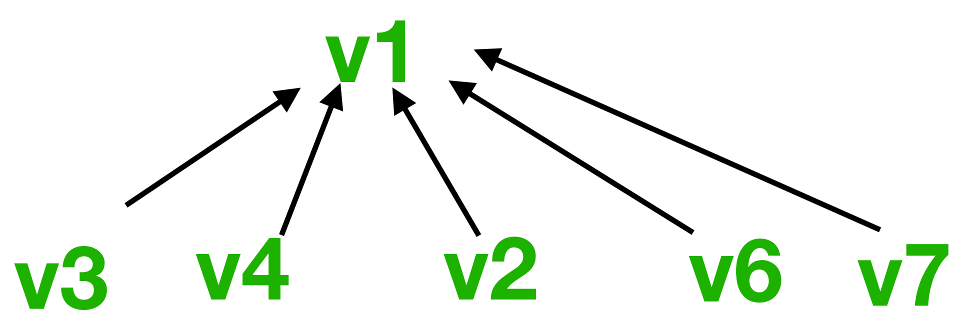


find(v7)

Path Compression

- We want to minimize the number of steps from each entry to the root.
- Modify find to make all nodes on the path children of the root.

```
public Node find(Node node){
     if (node.parent == null)
         return node;
     return node.parent = find(node.parent);
}
```

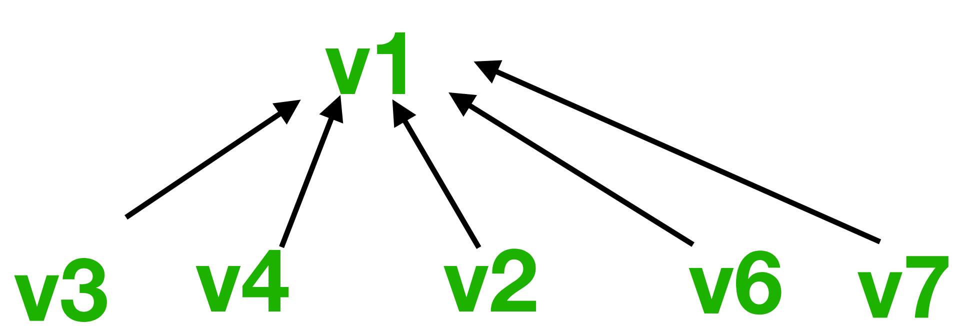


Path Compression

- We want to minimize the number of steps from each entry to the root.

- Modify find to make all nodes on the path children of the root.

```
public Node find(Node node){
     if (node.parent == null)
          return node;
     return node.parent = find(node.parent);
}
```



- With path compression and union by size, asymptotic runtime of union and find is "near constant" $O(\alpha(n))$. α is the inverse Ackerman function (practically, $\alpha(n) < 5$).