
Kruskal's MST and Find-Union Data Structure

CS4102

Tom Horton and Mark Floryan



Topics

Topics in this slide-deck:

- ▶ **Motivating Problem: Minimum Spanning Trees**
 - ▶ This is a graph problem, and you've seen it
- ▶ **One solution**
 - ▶ Kruskal's Algorithm (Uses a find-union structure)
- ▶ **Define and design the find-union to support Kruskal's Algorithm**
 - ▶ Will require some clever implementation details

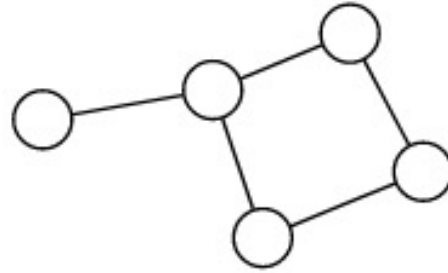
Minimum Spanning Trees

Spanning Tree

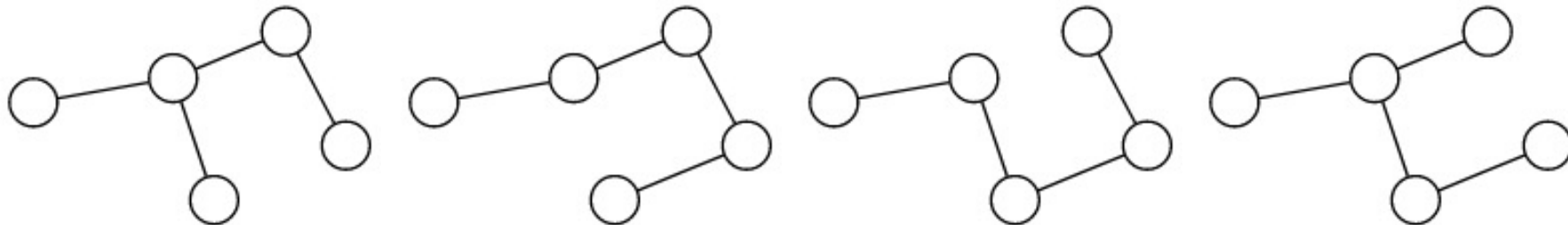
- ▶ A **spanning tree** of a graph G is a subgraph of G that contains every vertex in G and is also a **tree** (i.e., it has no cycles)
 - ▶ All connected graphs have spanning tree(s)
 - ▶ All spanning trees have the same number of nodes (all of them)
 - ▶ You can construct a spanning tree by arbitrarily remove edges from cycles

Spanning Tree: Example

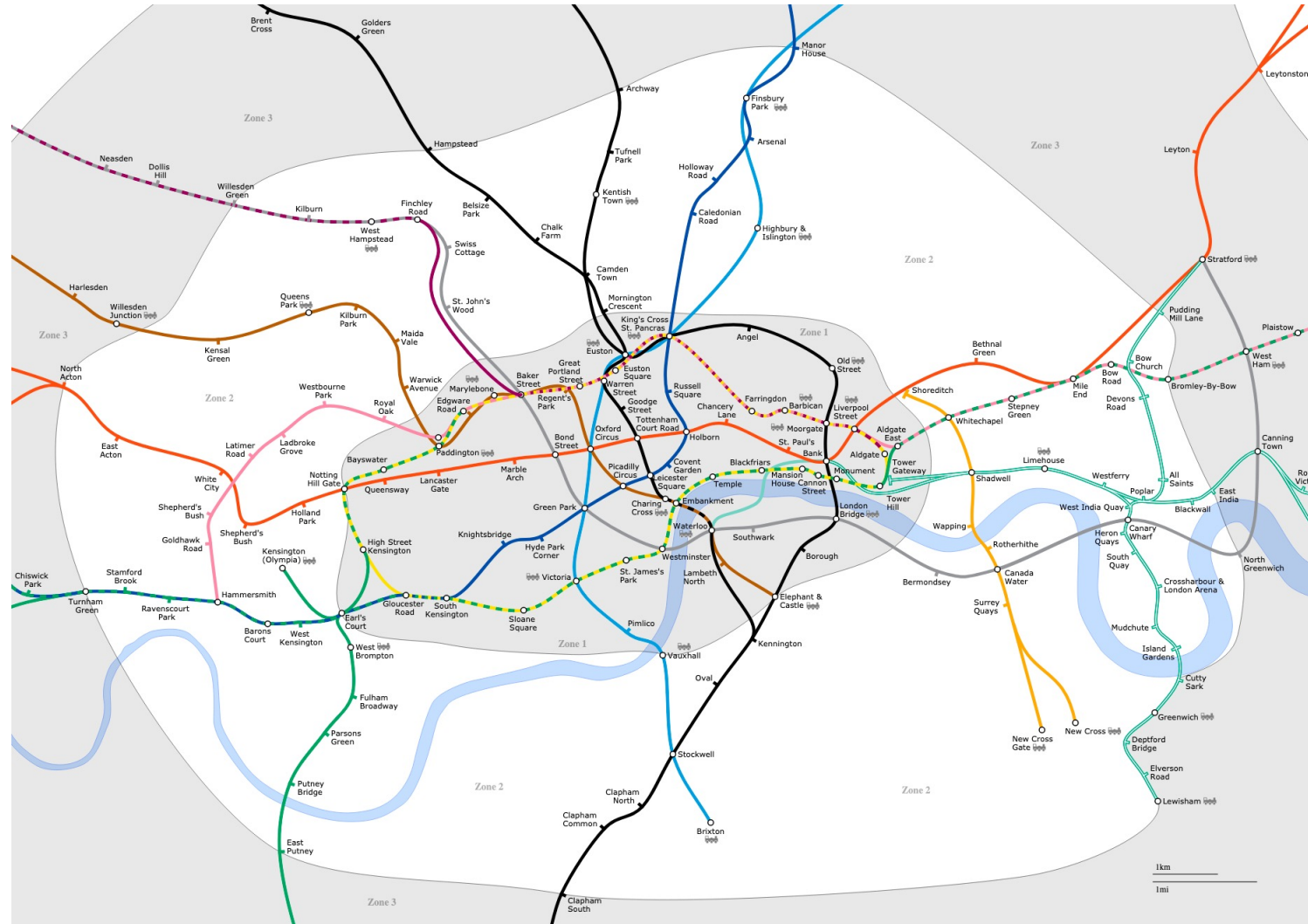
- ▶ Original Graph:



- ▶ Possible spanning trees:



Spanning Tree: Example (almost)



Minimum Spanning Tree

- ▶ Just constructing any spanning tree is simple
- ▶ Suppose edges have costs!
 - ▶ Cost of building tracks between two stations
 - ▶ Length of wire between boxes in a house
- ▶ Each spanning tree has a different total cost (sum of edges included in tree)
- ▶ The ***Minimum Spanning Tree*** is the spanning tree with lowest overall cost

Minimum Spanning Tree

- ▶ Given a connected and undirected graph $G=(V, E)$
- ▶ Find a graph $G' = (V, E')$ such that:
 - ▶ E' is a subset of E
 - ▶ $|E'| = |V| - 1$
 - ▶ G' is connected (assuming G was connected)
 - ▶ Sum of cost of edges in E' is minimum
- ▶ G' is then the minimum spanning tree

Kruskal's Algorithm

Kruskal's MST Algorithm

- ▶ Prim's approach:

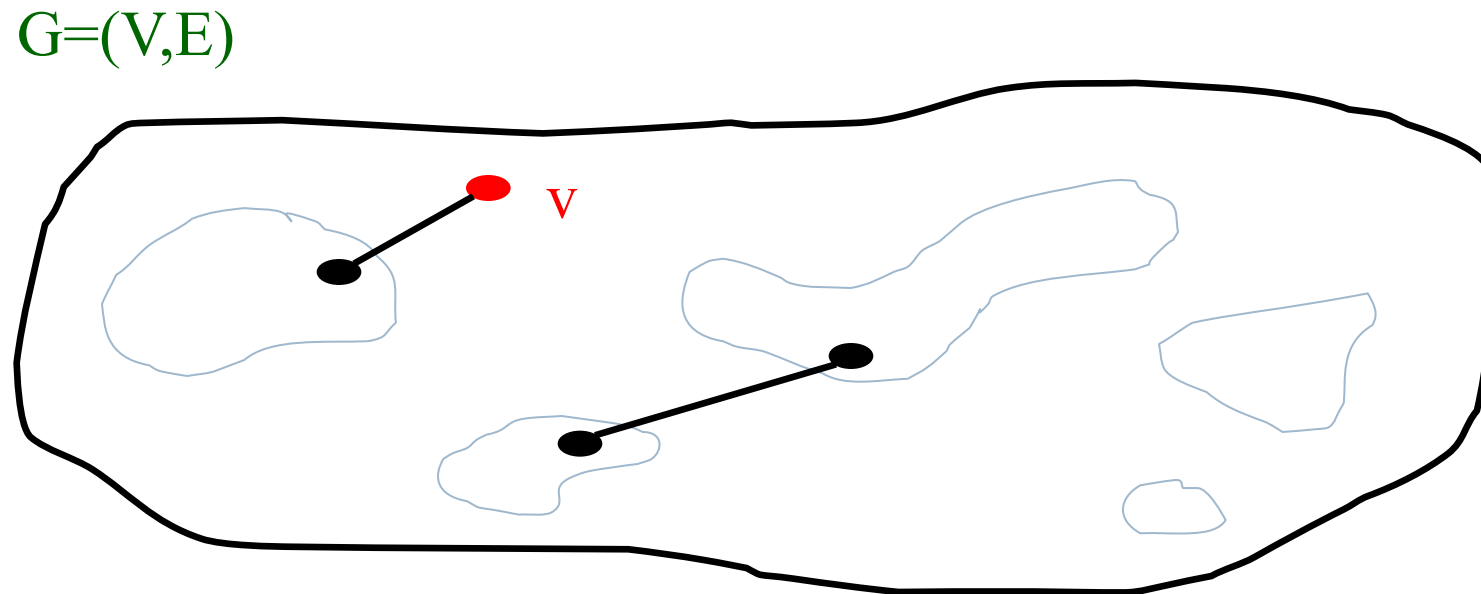
- ▶ Build one tree. Make the one tree bigger and as good as it can be.

- ▶ Kruskal's approach

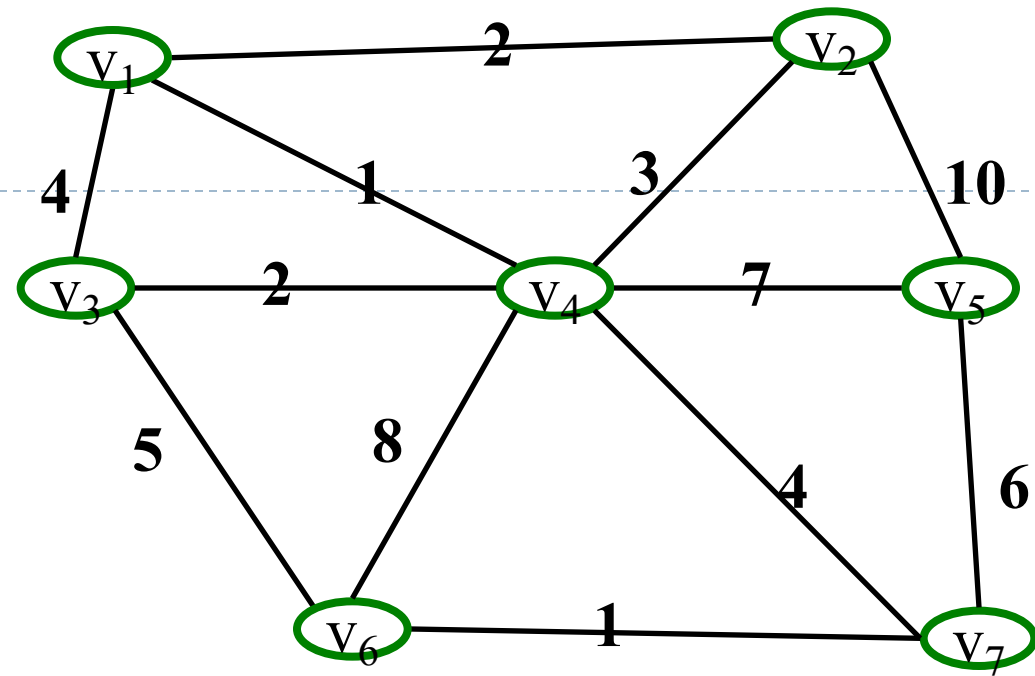
- ▶ Choose the best edge possible: smallest weight
- ▶ Not one tree – maintain a forest!
- ▶ Each edge added will connect two trees.
Can't form a cycle in a tree!
- ▶ After adding $n-1$ edges, you have one tree, the MST

Kruskal's MST Algorithm

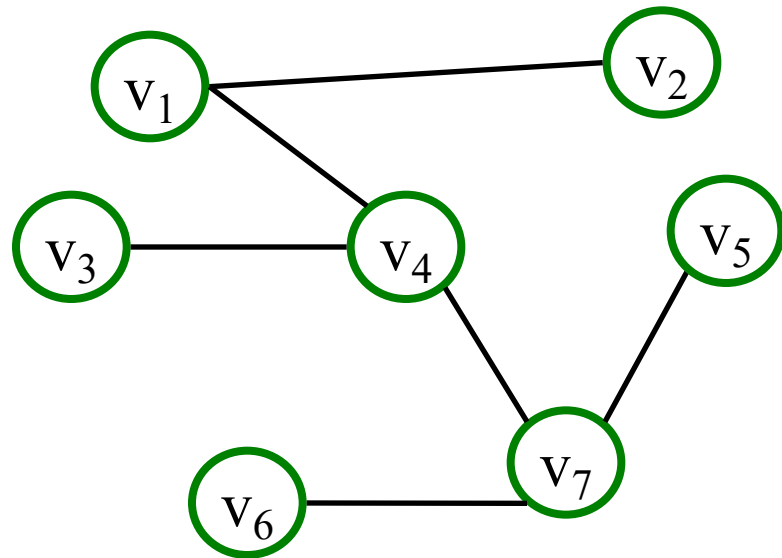
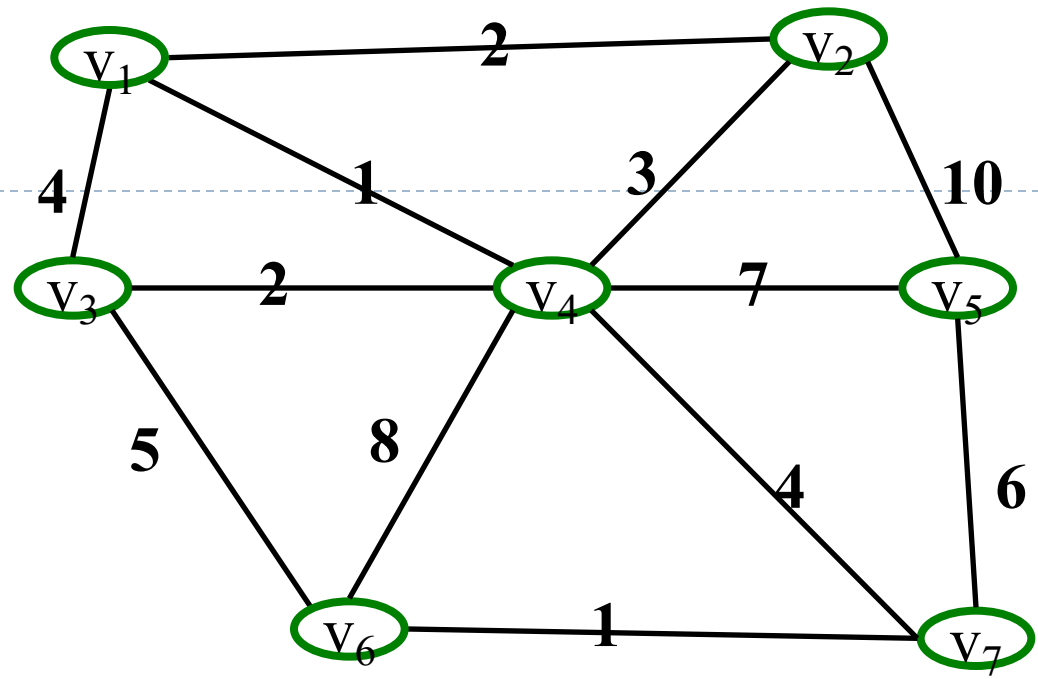
- ▶ Idea: Grow a **forest** out of edges that do not create a cycle. Pick an edge with the smallest weight.



MST



MST



Kruskal code

```
void Graph::kruskal() {  
    int edgesAccepted = 0;  
    DisjSet s(NUM_VERTICES);
```

```
    while (edgesAccepted < NUM_VERTICES - 1) {  
        e = smallest weight edge not deleted yet;
```

```
        // edge e = (u, v)  
        uset = s.find(u);
```

```
        vset = s.find(v);
```

```
        if (uset != vset) {  
            edgesAccepted++;
```

```
            s.unionSets(uset, vset);
```

```
        }
```

```
    }
```

```
}
```

$|E|$ heap ops



$2|E|$ finds



$|V|$ unions



Runtime of Kruskal's

- ▶ Every edge is placed on priority queue once and removed once
 - ▶ $\Theta(E * \log(E)) = \Theta(E * \log(V))$
- ▶ For each edge you do 2 set finds and one set union.
 - ▶ Let $f(V)$ be time of find, and $u(V)$ be time of union.
 - ▶ $\Theta(E * (2f(V) + u(V)))$
 - ▶ If find and union are linear time, then $\Theta(E * (2V + V)) = \Theta(E * V) = O(V^3)$
- ▶ Overall: $\Theta(E * \log(V) + E * V) = \Theta(E * V) = \mathbf{O(V^3)}$ //Assumes find and union linear time

Strategy for Kruskal's

- ▶ **EL = sorted set of edges ascending by weight**
- ▶ **Foreach edge e in EL**
 - ▶ $T1 = \text{tree for head}(e)$
 - ▶ $T2 = \text{tree for tail}(e)$
 - ▶ **If ($T1 \neq T2$)**
 - ▶ add e to the output (the MST)
 - ▶ Combine trees $T1$ and $T2$
- ▶ **Seems simple, no?**
 - ▶ But, how do you keep track of what trees a node is in?
 - ▶ Trees are sets. Need to find $\text{findset}(v)$ and “union” two sets

Find-Union Data Structure

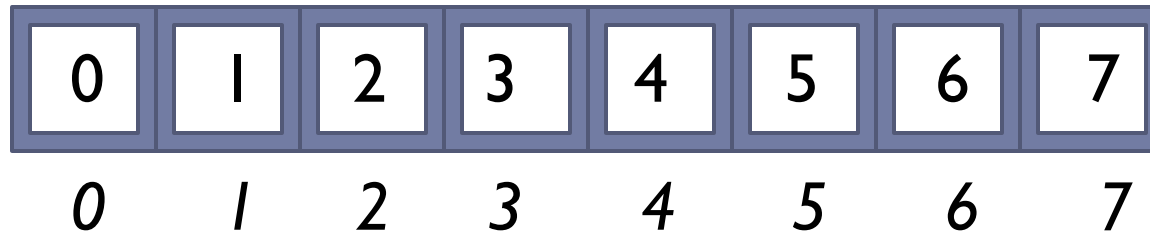
Union/Find and Disjoint Sets

- ▶ Needs to support the following operations

- ▶ `void makeSet(int n)` *//construct n independent sets*
- ▶ `int findSet(int i)` *//given i, which set does i belong to?*
- ▶ `void union(int i, int j)` *//merge sets containing i and j*

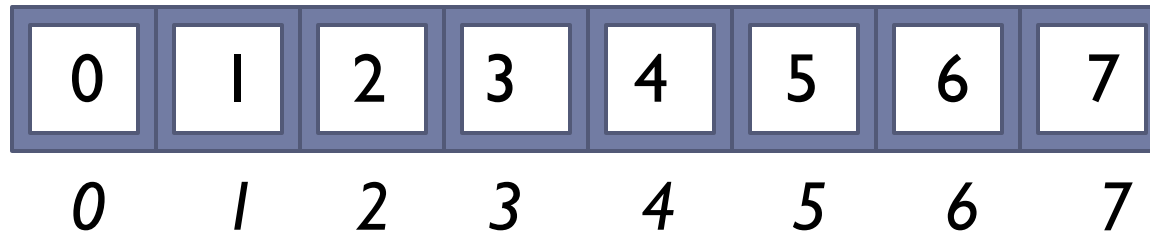
Union/Find and Disjoint Sets

- ▶ Needs to support the following operations
 - ▶ `void makeSet(int n)` //construct n independent sets
- ▶ Solution:
 - ▶ Store as array of size n. Each location stores label for that set.



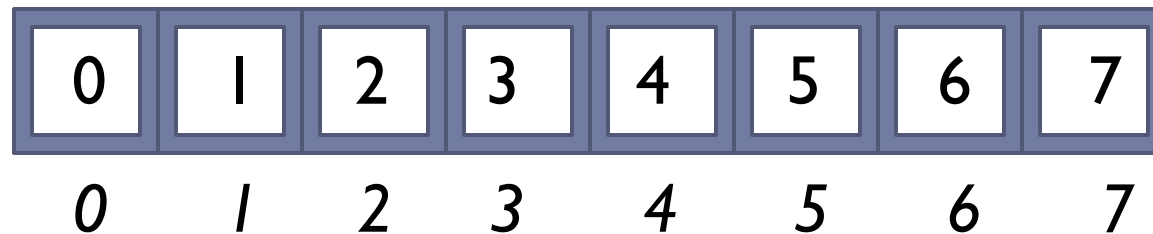
Union/Find and Disjoint Sets

- ▶ Needs to support the following operations
 - ▶ `int findSet(int i)` //given i, which set does i belong to?
- ▶ Solution: Trace around array until we find place where index and contents match
 - ▶ Start at index i and repeat:
 - ▶ If `a[i] == i` then return i
 - ▶ Else set `i = a[i]`



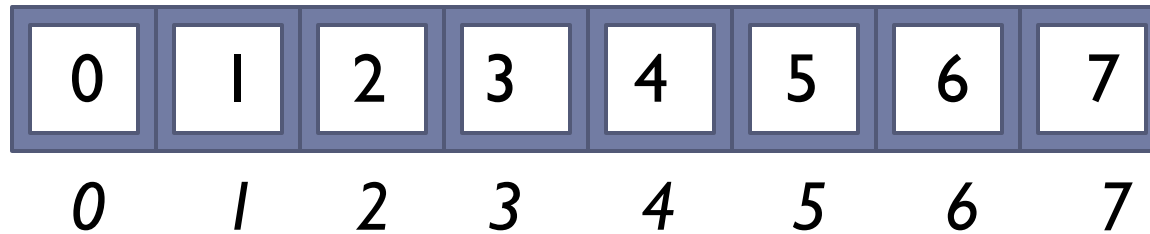
Union/Find and Disjoint Sets

- ▶ Needs to support the following operations
 - ▶ `void union(int i, int j)` //merge sets i and j
- ▶ Solution: find label for each set (call `find()` method), then set one label to point to other
 - ▶ `Label1 = find(i); Label2 = find(j)`
 - ▶ `a[Label1] = Label2` //OR `a[Label2] = Label1`



Union/Find and Disjoint Sets

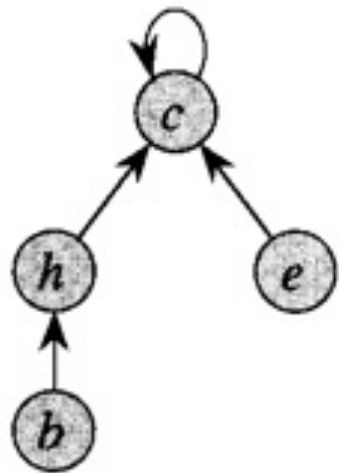
- ▶ **Example:**
 - ▶ merge(4,5)
 - ▶ merge(6,7)
 - ▶ merge(1,2)
 - ▶ merge(5,6)
 - ▶ find(1); find(4); find(6)



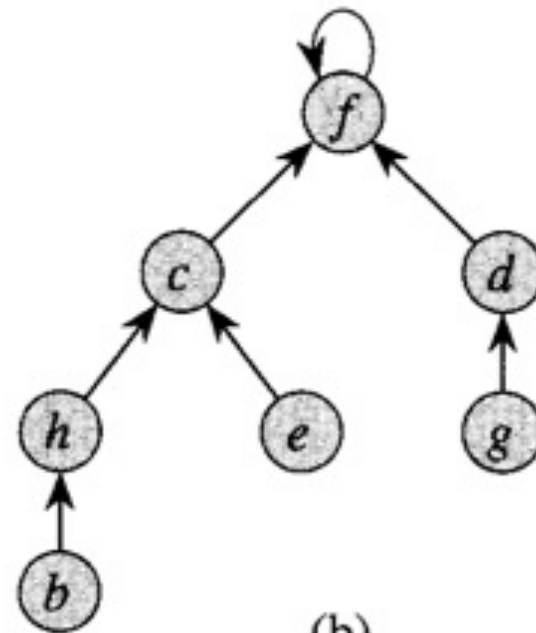
Union/Find and Disjoint Sets

- ▶ Time-complexity if where n is size of array?
- ▶ `makeSet()`
 - ▶ Linear: just create array and fill it with values
- ▶ `find()`
 - ▶ Linear if have to trace a long way to get to label
 - ▶ Constant if lucky and label given as input
- ▶ `union()`
 - ▶ Constant to change the label BUT...
 - ▶ Could be linear to find the two labels first.

Optimization 1: Union by rank



(a)



(b)

Optimization 1: Union by rank

- ▶ Easy to implement!!

MAKE-SET(x)

```
1  $x.p = x$   
2  $x.rank = 0$ 
```

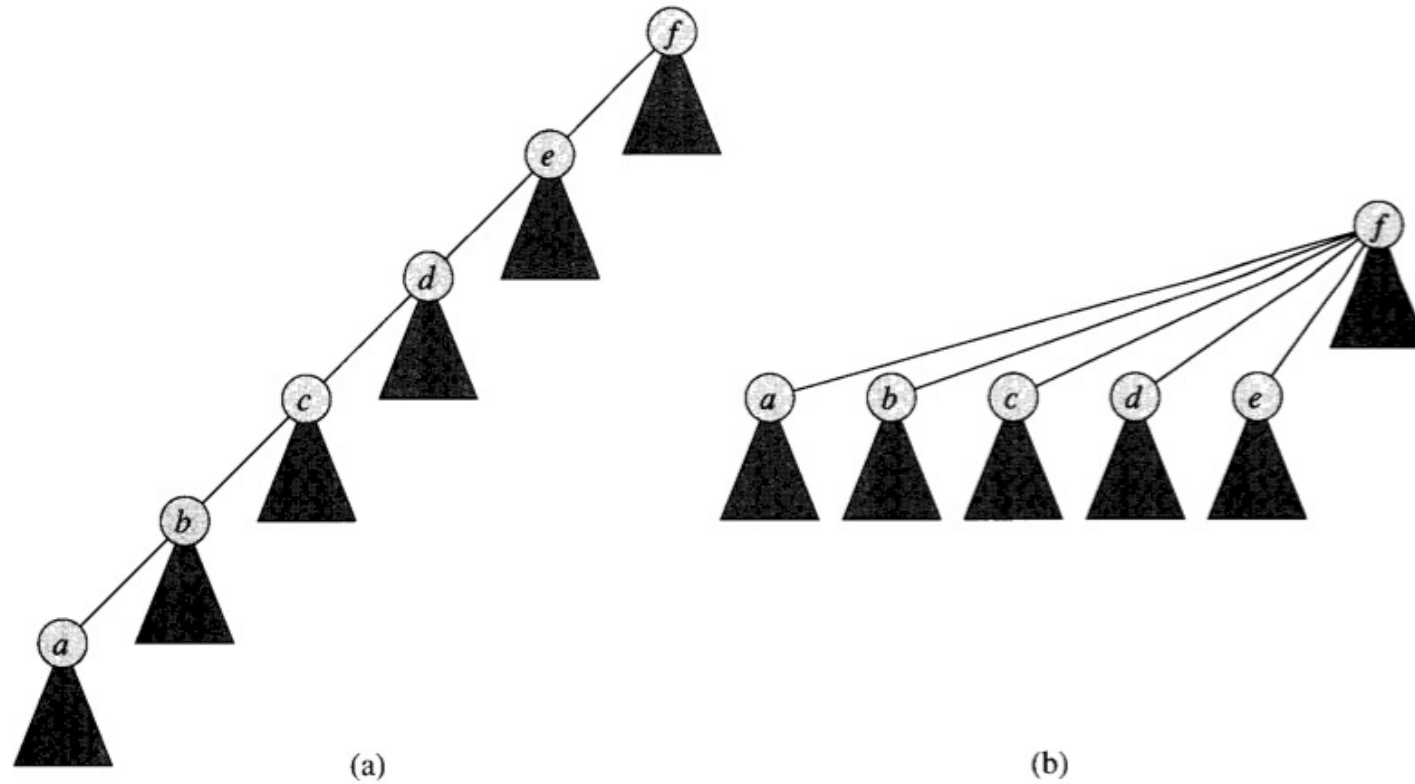
UNION(x, y)

```
1 LINK(FIND-SET( $x$ ), FIND-SET( $y$ ))
```

LINK(x, y)

```
1 if  $x.rank > y.rank$   
2      $y.p = x$   
3 else  $x.p = y$   
4     if  $x.rank == y.rank$   
5          $y.rank = y.rank + 1$ 
```

Optimization 2: Path Compression



Optimization 2: Path Compression

- ▶ Also easy to implement

FIND-SET(x)

1 **if** $x \neq x.p$

2 $x.p = \text{FIND-SET}(x.p)$

3 **return** $x.p$

Complexity for Kruskal's

- ▶ Union-by-rank and path compression yields m operations in $\Theta(m * \alpha(n))$
 - ▶ where $\alpha(n)$ a VERY slowly growing function. (See textbook for details)
 - ▶ m is the number of times you run the operation. So constant time, for each operation
- ▶ So Kruskal's overall:
 - ▶ $\Theta(E * \log(V) + E * 1) = \Theta(E * \log(V))$ //now the heap is slowest part

Summary

What did we learn?

- ▶ **Minimum Spanning Trees**
 - ▶ Review!
- ▶ **Kruskal's Algorithm**
 - ▶ Review again!
- ▶ **Find-union**
 - ▶ How to implement
 - ▶ How to optimize
 - ▶ How it affects runtime of Kruskal's algorithm.