CSE 373: Disjoint sets

Michael Lee

Wednesday, Feb 28, 2018

Review

Last time...

- **Prim's algorithm:**
  Nearly identical to Dijkstra's, except we use the distance to any already-visited node as the cost.

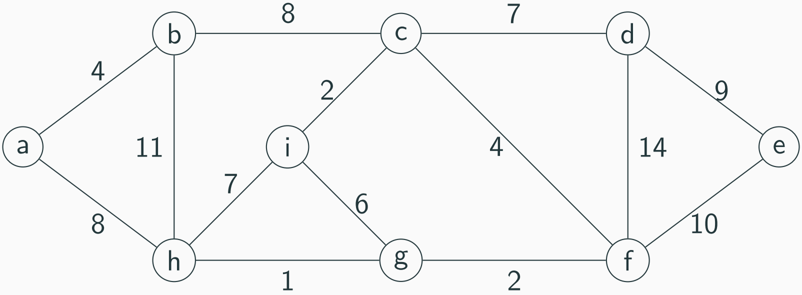
Review

Last time...

- **Prim's algorithm:**
  Nearly identical to Dijkstra's, except we use the distance to any already-visited node as the cost.
- **Kruskal's algorithm:**
  Loop over edges, from smallest to largest. Use the edge only if it doesn't introduce a cycle.

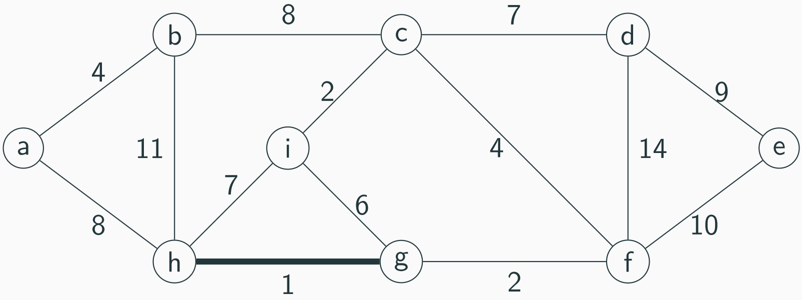
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



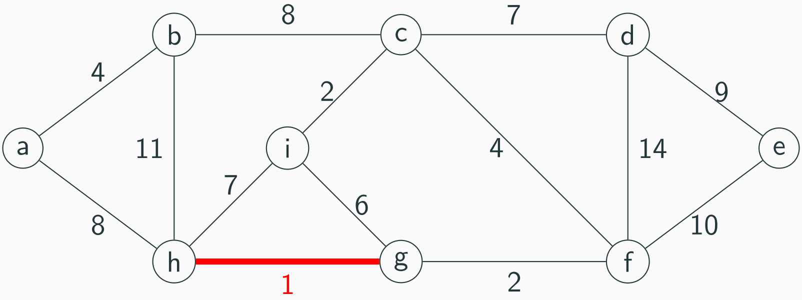
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



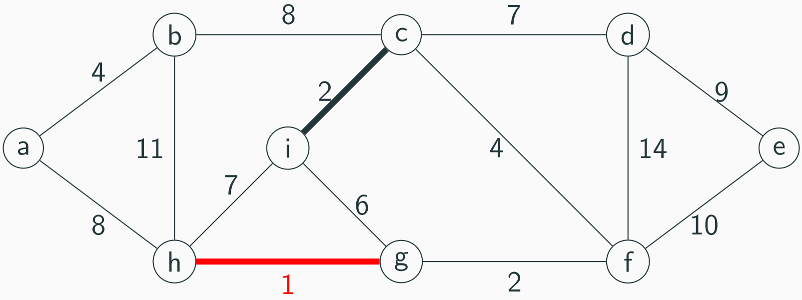
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



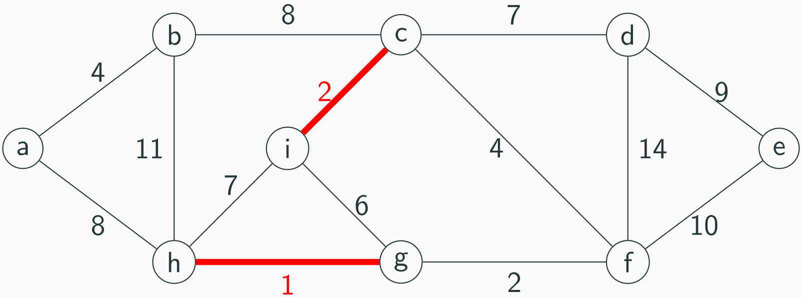
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



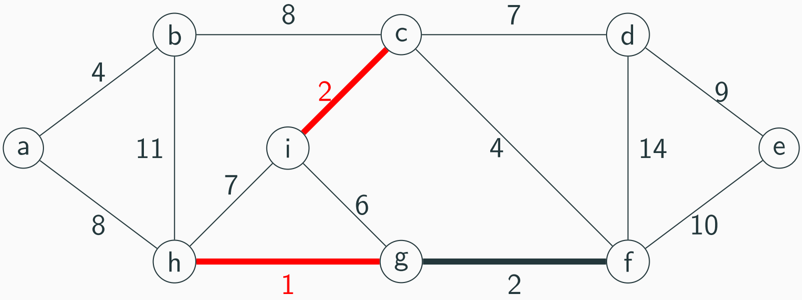
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



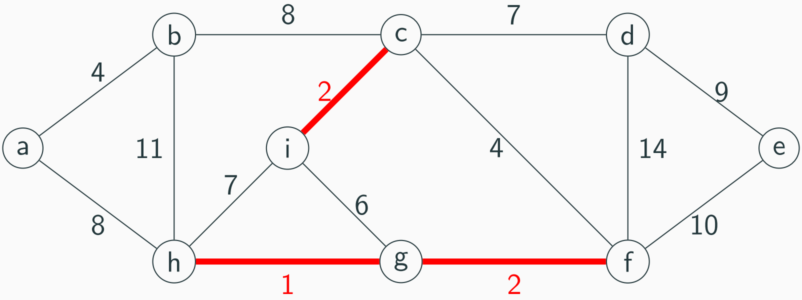
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



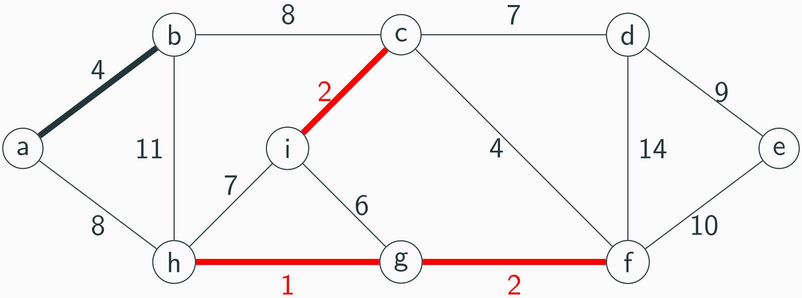
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



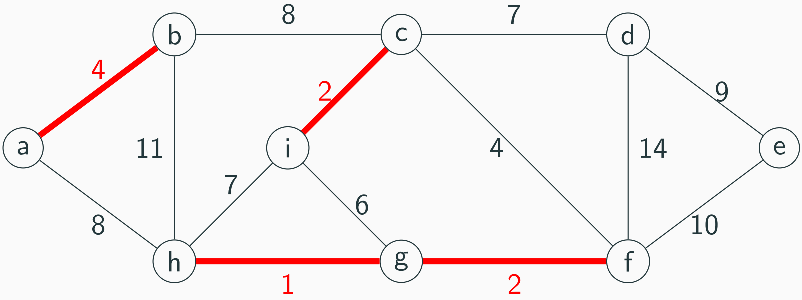
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



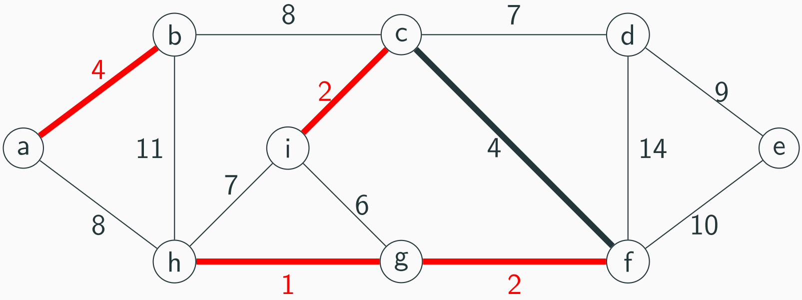
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



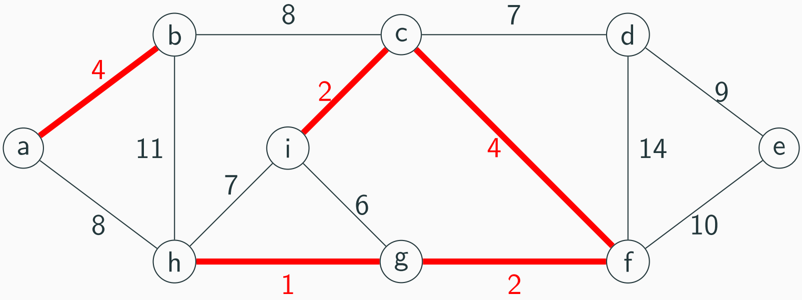
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



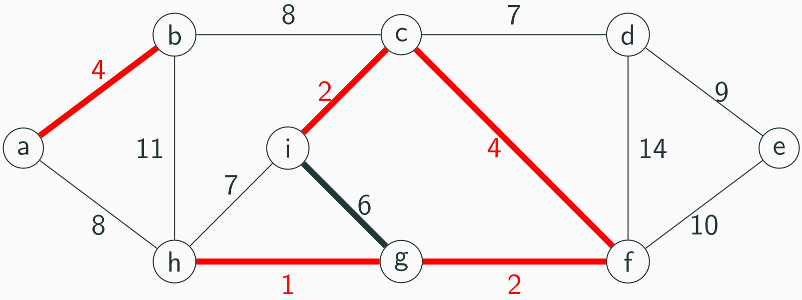
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



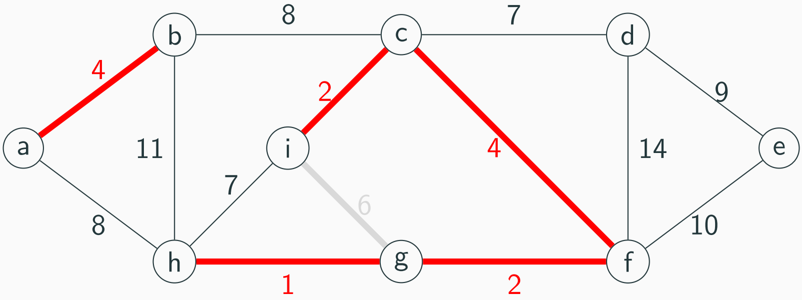
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



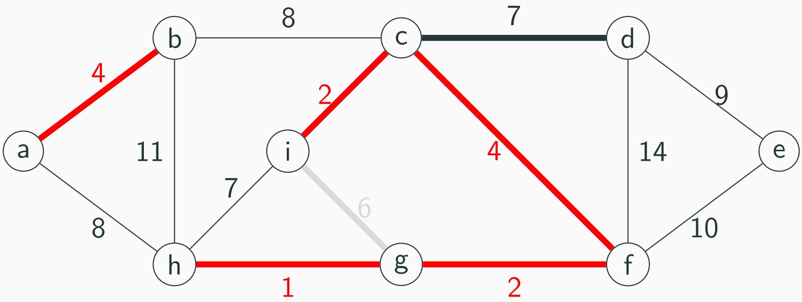
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



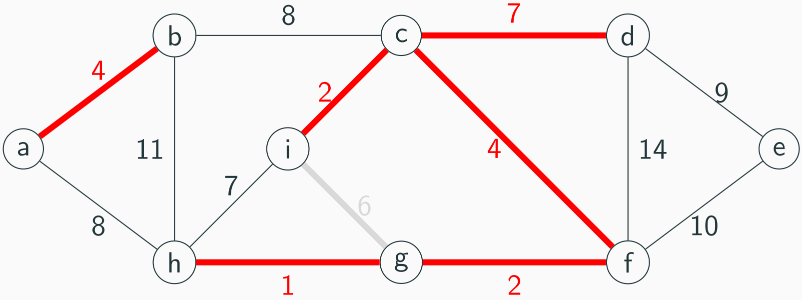
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



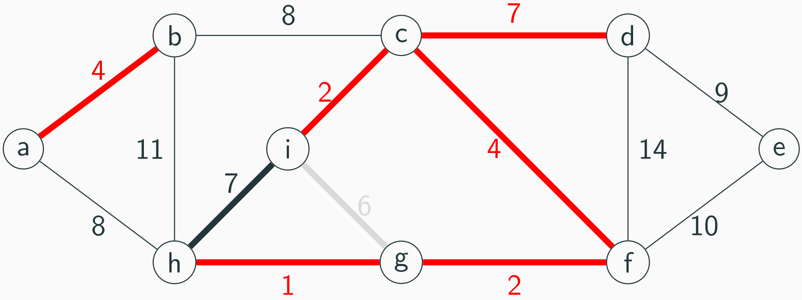
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



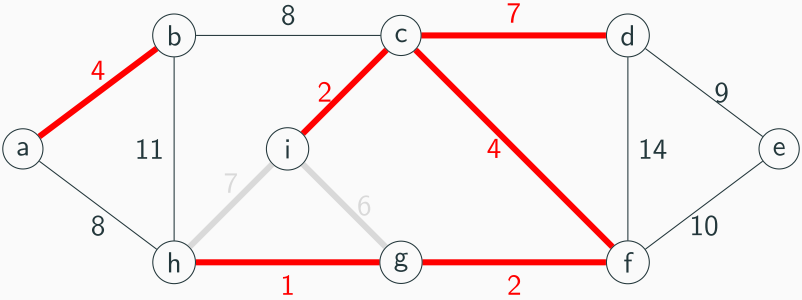
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



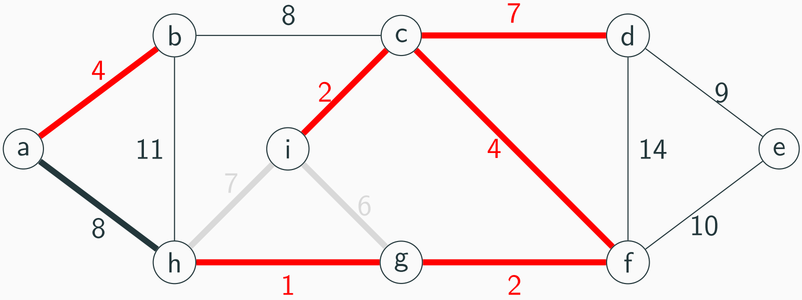
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



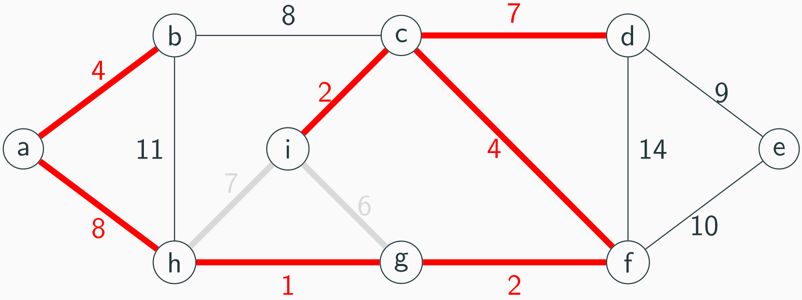
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



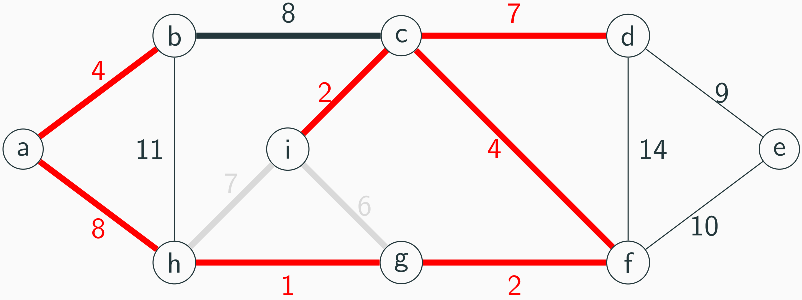
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



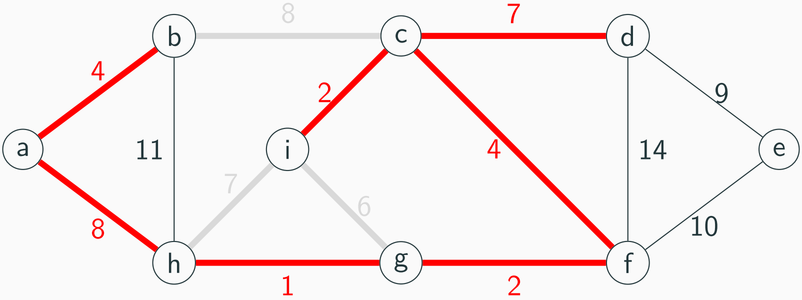
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



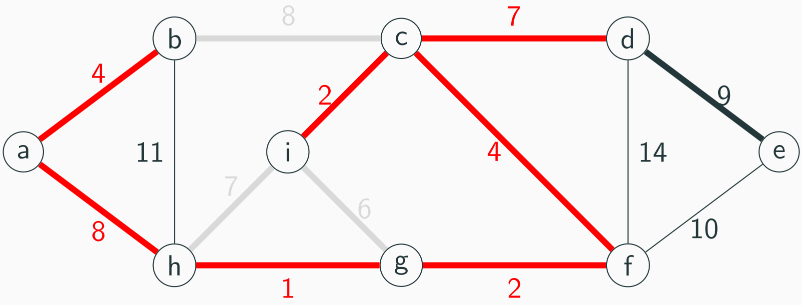
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



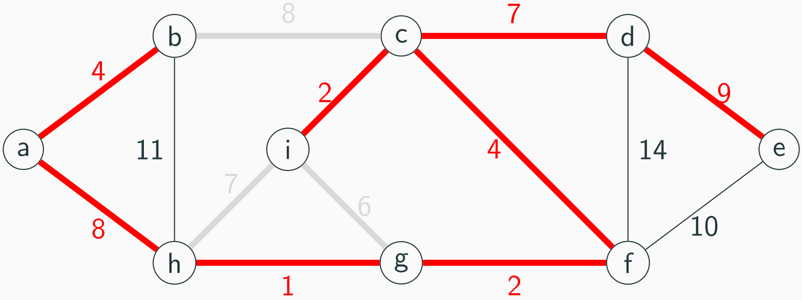
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:

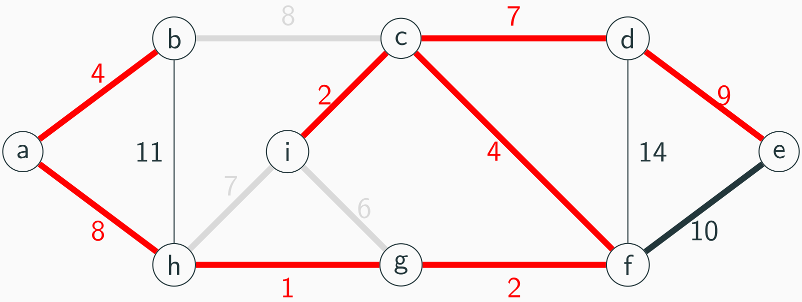


Kruskal's algorithm: example with a weighted graph

Example of the algorithm:

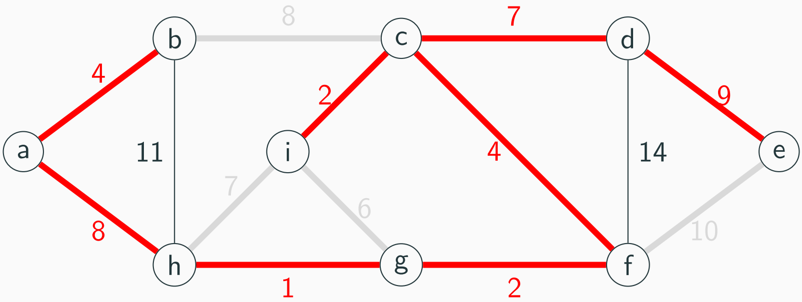


Example of the algorithm:



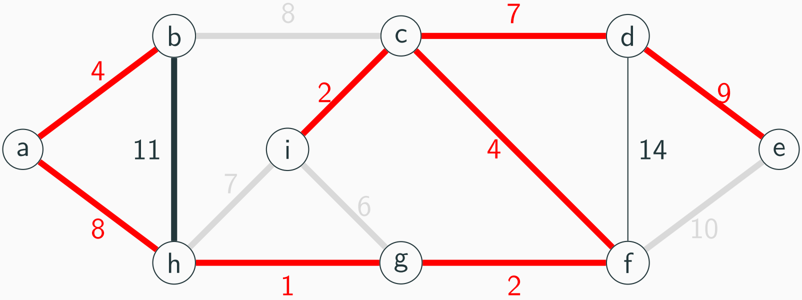
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



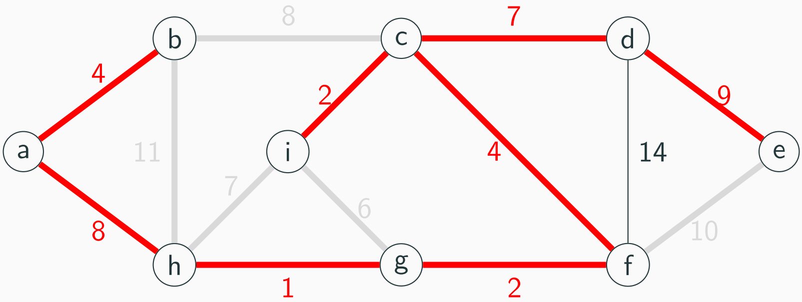
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



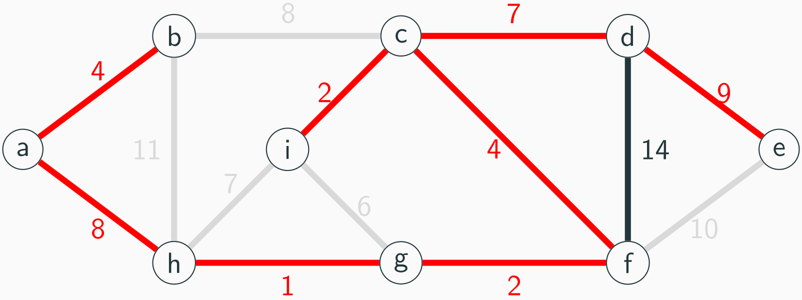
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



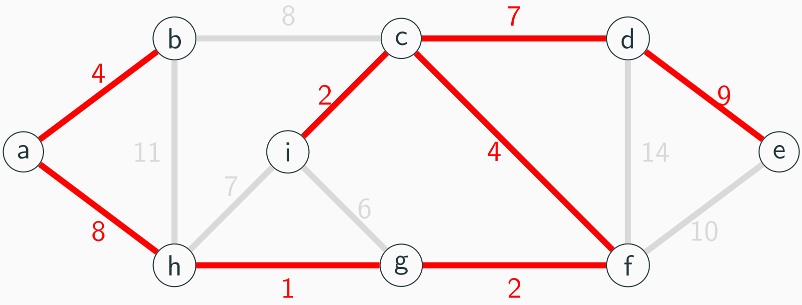
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



Kruskal's algorithm: analysis

Runtime analysis:

```
def kruskal():
    for (v : vertices):
        makeMST(v)

    sort edges in ascending order by their weight

    mst = new SomeSet<Edge>()
    for (edge : edges):
        if findMST(edge.src) != findMST(edge.dst):
            union(edge.src, edge.dst)
            mst.add(edge)

    return mst
```

Note: assume that...

- `makeMST(v)` takes $\mathcal{O}(t_m)$ time
- `findMST(v)`: takes $\mathcal{O}(t_f)$ time
- `union(u, v)`: takes $\mathcal{O}(t_u)$ time

Kruskal's algorithm: analysis

- Making the $|V|$ MSTs takes $\mathcal{O}(|V| \cdot t_m)$ time
- Sorting the edges takes $\mathcal{O}(|E| \cdot \log(|E|))$ time, assuming we use a general-purpose comparison sort
- The final loop takes $\mathcal{O}(|E| \cdot t_f + |V| \cdot t_u)$ time

Kruskal's algorithm: analysis

- Making the $|V|$ MSTs takes $\mathcal{O}(|V| \cdot t_m)$ time
- Sorting the edges takes $\mathcal{O}(|E| \cdot \log(|E|))$ time, assuming we use a general-purpose comparison sort
- The final loop takes $\mathcal{O}(|E| \cdot t_f + |V| \cdot t_u)$ time

Putting it all together:

$$\mathcal{O}(|V| \cdot t_m + |E| \cdot \log(|E|) + |E| \cdot t_f + |V| \cdot t_u)$$

The DisjointSet ADT

But wait, what exactly is t_m, t_f, and t_u? How exactly do we implement `makeMST(v)`, `findMST(v)`, and `union(u, v)`?

The DisjointSet ADT

But wait, what exactly is t_m, t_f, and t_u? How exactly do we implement `makeMST(v)`, `findMST(v)`, and `union(u, v)`?

We can do so using a new ADT called the `DisjointSet` ADT!

Interlude: What is a set?

Review: what is a set?

- A set is a “bag” of elements arranged in no particular order.

Interlude: What is a set?

Review: what is a set?

- A set is a "bag" of elements arranged in no particular order.
- A set may not contain duplicates.

Interlude: What is a set?

Review: what is a set?

- A set is a “bag” of elements arranged in no particular order.
- A set may not contain duplicates.

Interlude: What is a set?

Review: what is a set?

- A set is a "bag" of elements arranged in no particular order.
- A set may not contain duplicates.

We implemented a set in project 2: `ChainedHashSet`

Interlude: What is a set?

Review: what is a set?

- A set is a "bag" of elements arranged in no particular order.
- A set may not contain duplicates.

We implemented a set in project 2: `ChainedHashSet`

Interesting note: sets come up all the time in math.

The DisjointSet ADT

Properties of a disjoint-set data structure:

- A disjoint-set data structure maintains a collection of many different sets.

The DisjointSet ADT

Properties of a disjoint-set data structure:

- A disjoint-set data structure maintains a collection of many different sets.
- An item **may not** be contained within multiple sets.
  Each set must be *disjoint*.

The DisjointSet ADT

Properties of a disjoint-set data structure:

- A disjoint-set data structure maintains a collection of many different sets.
- An item **may not** be contained within multiple sets.
  Each set must be *disjoint*.
- Each set is associated with some *representative*.

The DisjointSet ADT

Properties of a disjoint-set data structure:

- A disjoint-set data structure maintains a collection of many different sets.
- An item **may not** be contained within multiple sets.
  Each set must be *disjoint*.
- Each set is associated with some *representative*.
  What is a representative? Any sort of unique "identifier".
  Examples:

The DisjointSet ADT

Properties of a disjoint-set data structure:

- A disjoint-set data structure maintains a collection of many different sets.
- An item **may not** be contained within multiple sets.
  Each set must be *disjoint*.
- Each set is associated with some *representative*.
  What is a representative? Any sort of unique "identifier".
  Examples:
  - We could pick some arbitrary element in the set to be the "representative"

The DisjointSet ADT

Properties of a disjoint-set data structure:

- A disjoint-set data structure maintains a collection of many different sets.
- An item **may not** be contained within multiple sets.
  Each set must be *disjoint*.
- Each set is associated with some *representative*.
  What is a representative? Any sort of unique "identifier".
  Examples:
  - We could pick some arbitrary element in the set to be the "representative"
  - We could assign each set some unique integer id.

The DisjointSet ADT

A disjoint-set has the following core operations:

- **`makeSet(x)`** – Creates a new set where the only member is x. We assign that set a representative.

The DisjointSet ADT

A disjoint-set has the following core operations:

- **`makeSet(x)`** – Creates a new set where the only member is x. We assign that set a representative.
- **`findSet(x)`** – Looks up the set containing x. Then, returns the representative of that set.

The DisjointSet ADT

A disjoint-set has the following core operations:

- **`makeSet(x)`** – Creates a new set where the only member is x. We assign that set a representative.
- **`findSet(x)`** – Looks up the set containing x. Then, returns the representative of that set.
- **`union(x, y)`** – Looks up the set containing x and the set containing y. We combine these two sets together into one. We (arbitrarily) pick one of the two representatives to be the representative of this new set.

The DisjointSet ADT

A disjoint-set has the following core operations:

- **`makeSet(x)`** – Creates a new set where the only member is x. We assign that set a representative.
- **`findSet(x)`** – Looks up the set containing x. Then, returns the representative of that set.
- **`union(x, y)`** – Looks up the set containing x and the set containing y. We combine these two sets together into one. We (arbitrarily) pick one of the two representatives to be the representative of this new set.

The DisjointSet ADT

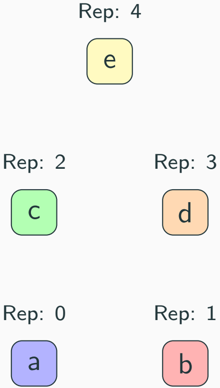
Example:

```
makeSet(a)
makeSet(b)
makeSet(c)
makeSet(d)
makeSet(e)
```

The DisjointSet ADT

Example:

```
makeSet(a)
makeSet(b)
makeSet(c)
makeSet(d)
makeSet(e)
```

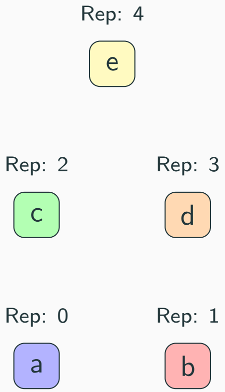


The DisjointSet ADT

Example:

```
makeSet(a)
makeSet(b)
makeSet(c)
makeSet(d)
makeSet(e)

print(findSet(a))
print(findSet(d))
```



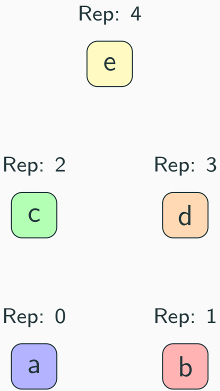
The DisjointSet ADT

Example:

```
makeSet(a)
makeSet(b)
makeSet(c)
makeSet(d)
makeSet(e)

print(findSet(a))
print(findSet(d))

union(a, c)
union(b, d)
print(findSet(a) == findSet(c))
print(findSet(a) == findSet(d))
```

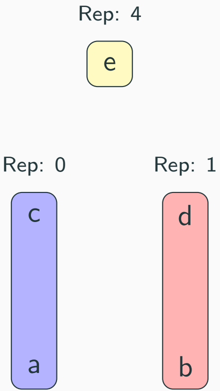


Example:

```
makeSet(a)
makeSet(b)
makeSet(c)
makeSet(d)
makeSet(e)

print(findSet(a))
print(findSet(d))

union(a, c)
union(b, d)
print(findSet(a) == findSet(c))
print(findSet(a) == findSet(d))
```



The DisjointSet ADT

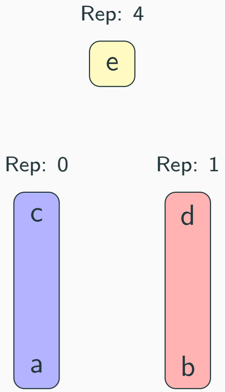
Example:

```
makeSet(a)
makeSet(b)
makeSet(c)
makeSet(d)
makeSet(e)

print(findSet(a))
print(findSet(d))

union(a, c)
union(b, d)
print(findSet(a) == findSet(c))
print(findSet(a) == findSet(d))

union(c, b)
print(findSet(a) == findSet(d))
```



The DisjointSet ADT

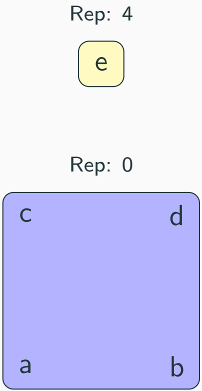
Example:

```
makeSet(a)
makeSet(b)
makeSet(c)
makeSet(d)
makeSet(e)

print(findSet(a))
print(findSet(d))

union(a, c)
union(b, d)
print(findSet(a) == findSet(c))
print(findSet(a) == findSet(d))

union(c, b)
print(findSet(a) == findSet(d))
```



The DisjointSet ADT

What operations does a disjoint-set **NOT** support?

What operations does a disjoint-set **NOT** support?

Answer: The ability to actually get the entire set.

We can *make* a set, *check* if an item is in a set, and *combine* two sets, but we don't have a built-in way of *getting* the entire set itself.

What operations does a disjoint-set **NOT** support?

Answer: The ability to actually get the entire set.

We can *make* a set, *check* if an item is in a set, and *combine* two sets, but we don't have a built-in way of *getting* the entire set itself.

Insight: The few operations we need to support, the more creative our implementation can be.

The DisjointSet ADT

What operations does a disjoint-set **NOT** support?

Answer: The ability to actually get the entire set.

We can *make* a set, *check* if an item is in a set, and *combine* two sets, but we don't have a built-in way of *getting* the entire set itself.

Insight: The few operations we need to support, the more creative our implementation can be.

(If the client really wants the sets, they can get it themselves in $\mathcal{O}(n)$ time – how?)

DisjointSet: implementation

So, how do we implement these?

DisjointSet: implementation

So, how do we implement these?

Core idea:

- We represent each set as a tree
- The disjoint-set keeps track of a “forest” of trees

DisjointSet: implementation

So, how do we implement these?

Core idea:

- We represent each set as a tree
- The disjoint-set keeps track of a "forest" of trees

Intuitions:

DisjointSet: implementation

So, how do we implement these?

Core idea:

- We represent each set as a tree
- The disjoint-set keeps track of a “forest” of trees

Intuitions:

- We want union-ing to be cheap.
  Combining two trees is cheap; we just manipulate pointers.

DisjointSet: implementation

So, how do we implement these?

Core idea:

- We represent each set as a tree
- The disjoint-set keeps track of a "forest" of trees

Intuitions:

- We want union-ing to be cheap.
  Combining two trees is cheap; we just manipulate pointers.
- We want a single "representative" per set.
  A tree has a single root!

DisjointSet: implementation

High-level overview:

- **`makeSet(x)`**: Adds a new tree (of size 1) to our "forest"
- **`findSet(x)`**: Looks up the node, then finds root of tree
- **`union(x, y)`**: Combines two trees into one

DisjointSet: implementation

Suppose we call `makeSet(...)` on 0 through 5.

DisjointSet: implementation

Suppose we call `makeSet(...)` on 0 through 5.

Each `makeSet(...)` adds a new tree to our "forest".

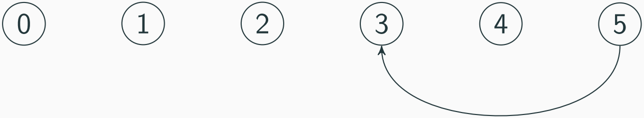
Note that right now, each tree has only one element.

DisjointSet: implementation

Suppose we call `union(3, 5)`.

DisjointSet: implementation

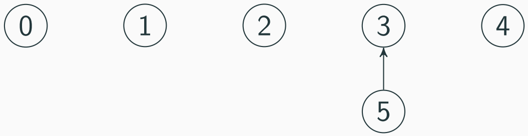
Suppose we call `union(3, 5)`.



We combine those two trees into one.

DisjointSet: implementation

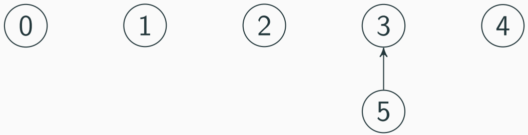
Suppose we call `union(3, 5)`.



We combine those two trees into one.

DisjointSet: implementation

Suppose we call `union(3, 5)`.

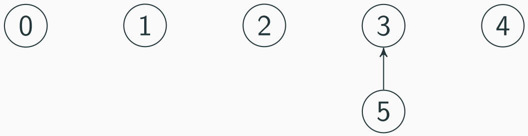


We combine those two trees into one.

Assumption: we have an $\mathcal{O}(1)$ way of getting each node.
(E.g. maintain a hashmap of numbers to node objects.)

DisjointSet: implementation

Suppose we call `union(3, 5)`.



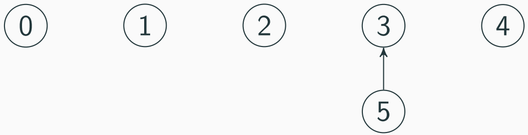
We combine those two trees into one.

Assumption: we have an $\mathcal{O}(1)$ way of getting each node.
(E.g. maintain a hashmap of numbers to node objects.)

Question: how do we implement `findSet(...)`?

DisjointSet: implementation

Suppose we call `union(3, 5)`.



We combine those two trees into one.

Assumption: we have an $\mathcal{O}(1)$ way of getting each node.
(E.g. maintain a hashmap of numbers to node objects.)

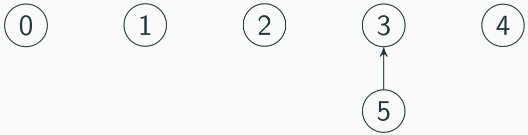
Question: how do we implement `findSet(...)`?

Once we find a node, move upwards until we're looking at root.

Then, return the root's data field.

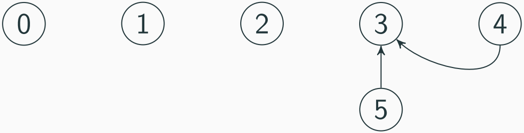
DisjointSet: implementation

Suppose we call `union(5, 4)`.



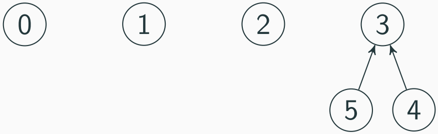
DisjointSet: implementation

Suppose we call `union(5, 4)`.



DisjointSet: implementation

Suppose we call `union(5, 4)`.

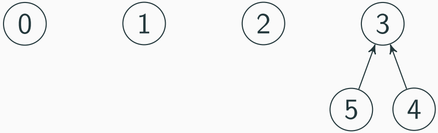


Algorithm: Find the roots of both trees and add one tree as a subchild of the other.

Which tree becomes the new root? For now, pick randomly.

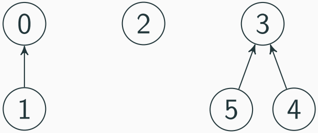
DisjointSet: implementation

Suppose we call `union(0, 1)`, then `union(2, 0)`.



DisjointSet: implementation

Suppose we call `union(0, 1)`, then `union(2, 0)`.



DisjointSet: implementation

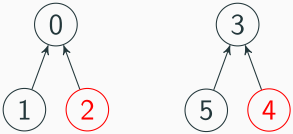
Suppose we call `union(0, 1)`, then `union(2, 0)`.

DisjointSet: implementation

Now, suppose we call `union(2, 4)`. What happens?

DisjointSet: implementation

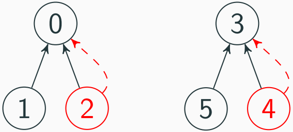
Now, suppose we call `union(2, 4)`. What happens?



Step 1: We look up 2 and 3

DisjointSet: implementation

Now, suppose we call `union(2, 4)`. What happens?



Step 2: We find the roots of 2 and 3

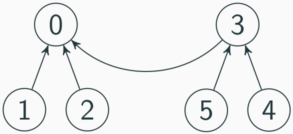
DisjointSet: implementation

Now, suppose we call `union(2, 4)`. What happens?

Step 2: We find the roots of 2 and 3

DisjointSet: implementation

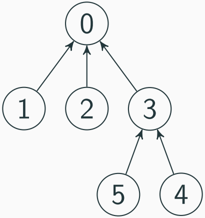
Now, suppose we call `union(2, 4)`. What happens?



Step 3: We nest one tree inside the other

DisjointSet: implementation

Now, suppose we call `union(2, 4)`. What happens?



Step 3: We nest one tree inside the other

DisjointSet: Analysis

What's the worst-case runtime of our methods?

DisjointSet: Analysis

What's the worst-case runtime of our methods?

Better question: are our trees guaranted to be balanced?

DisjointSet: Analysis

What's the worst-case runtime of our methods?

Better question: are our trees guaranted to be balanced?

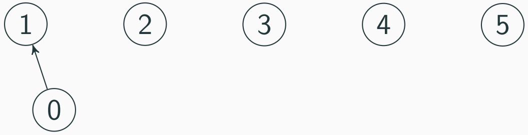
Hint: When union-ing, we pick which tree is nested randomly.
Does that guarantee we'll get a balanced tree?

DisjointSet: Analysis

The worst-case scenario:

DisjointSet: Analysis

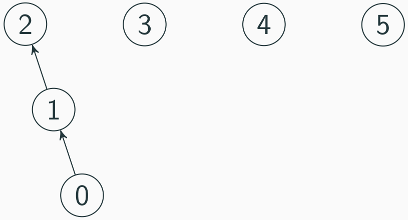
The worst-case scenario:



Possible outcome of calling `union(0, 1)`

DisjointSet: Analysis

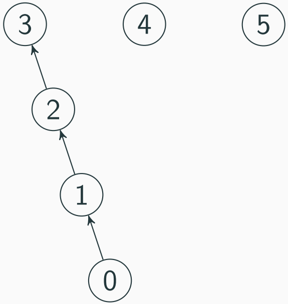
The worst-case scenario:



Possible outcome of calling `union(0, 2)`

DisjointSet: Analysis

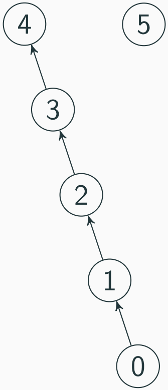
The worst-case scenario:



Possible outcome of calling `union(0, 3)`

DisjointSet: Analysis

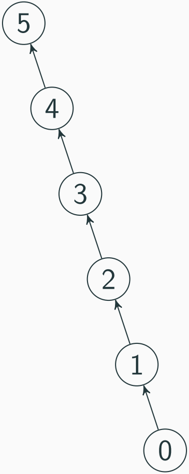
The worst-case scenario:



Possible outcome of calling `union(0, 4)`

DisjointSet: Analysis

The worst-case scenario:



Possible outcome of calling `union(0, 5)`

DisjointSet: implementation

So, what are the worst-case runtimes?

- **`makeSet(x)`**:
- **`findSet(x)`**:
- **`union(x, y)`**:

DisjointSet: implementation

So, what are the worst-case runtimes?

- **`makeSet(x)`**:
  $\mathcal{O}(1)$ – creating the tree takes constant time
- **`findSet(x)`**:
  $\mathcal{O}(n)$ – if it's a linked list, we need to traverse n elements!
- **`union(x, y)`**:
  $\mathcal{O}(n)$ – union calls `findSet(...)` on both elements

...where n is the total number of items added to the disjoint-set.

Improving DisjointSet

How can we improve disjoint sets?

Improving DisjointSet

How can we improve disjoint sets?

1. **Union-by-rank:**
   Strategy to make sure trees are balanced

Improving DisjointSet

How can we improve disjoint sets?

1. **Union-by-rank:**
   Strategy to make sure trees are balanced
2. **Path compression:**
   Hijack `findSet(x)` and make it do a little extra work to improve overall performance.

Improving DisjointSet

How can we improve disjoint sets?

1. **Union-by-rank:**
   Strategy to make sure trees are balanced
2. **Path compression:**
   Hijack `findSet(x)` and make it do a little extra work to improve overall performance.
3. **Array representation:**
   Takes advantage of cache locality, simplifies implementation, etc.

Union-by-rank

Problem: Our trees could be unbalanced

Union-by-rank

Problem: Our trees could be unbalanced

Solution:

Let rank(x) be a number representing the upper-bound of the height of x. So, rank(x) $\geq$ height(x).

Union-by-rank

Problem: Our trees could be unbalanced

Solution:

Let rank(x) be a number representing the upper-bound of the height of x. So, rank(x) $\geq$ height(x).

We then...

1. Keep track of the rank of all trees.

Union-by-rank

Problem: Our trees could be unbalanced

Solution:

Let $\text{rank}(x)$ be a number representing the upper-bound of the height of x. So, $\text{rank}(x) \geq \text{height}(x)$.

We then...

1. Keep track of the rank of all trees.
2. When unioning, make the tree with the larger rank the root!

Union-by-rank

Problem: Our trees could be unbalanced

Solution:

Let rank(x) be a number representing the upper-bound of the height of x. So, rank(x) $\geq$ height(x).

We then...

1. Keep track of the rank of all trees.
2. When unioning, make the tree with the larger rank the root!
3. If it's a tie, pick one randomly and increase the rank by one.

Union-by-rank

Problem: Our trees could be unbalanced

Solution:

Let rank(x) be a number representing the upper-bound of the height of x. So, rank(x) $\geq$ height(x).

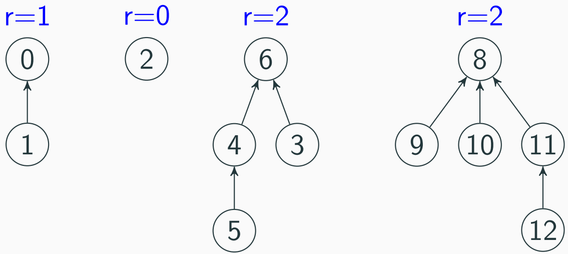
We then...

1. Keep track of the rank of all trees.
2. When unioning, make the tree with the larger rank the root!
3. If it's a tie, pick one randomly and increase the rank by one.

(Why not keep track of the height? When we look at path compression, keeping track of the height becomes more challenging.)

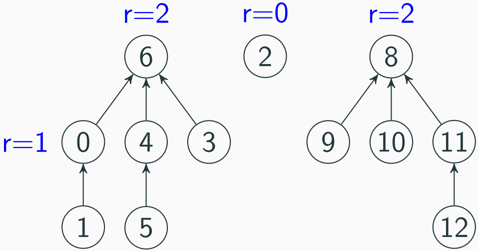
Union-by-rank

Example: Suppose we call `union(1, 5)`?



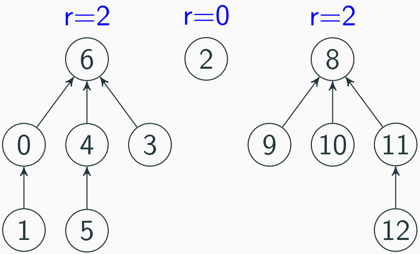
Union-by-rank

Example: Suppose we call `union(1, 5)`?



Union-by-rank

Example: Suppose we call `union(1, 5)`?

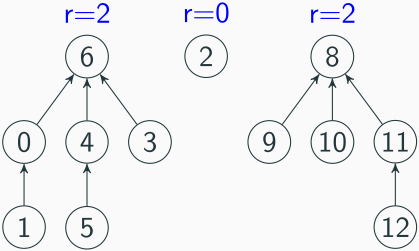


The tree with the root of “6” has the larger rank, so we make it the root.

Note: we’re not really “removing” the rank from node 0 – it’s just irrelevant, so we’re ignoring it and omitting it from the diagram to save space. We only care about the ranks at the roots.

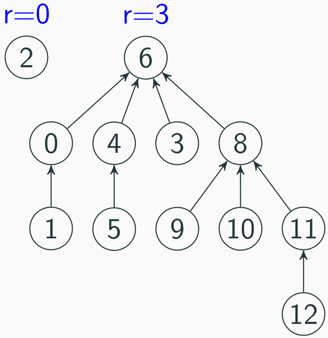
Union-by-rank

Example: Suppose we call `union(5, 11)`?



Union-by-rank

Example: Suppose we call `union(5, 11)`?



Here, there's a tie. We break the tie arbitrarily, and increment the rank of the new tree by one.

Union-by-rank

Net effect? Our trees stay relatively balanced.

So, what are the worst-case runtimes now?

- **`makeSet(x)`**:
- **`findSet(x)`**:
- **`union(x, y)`**:

Union-by-rank

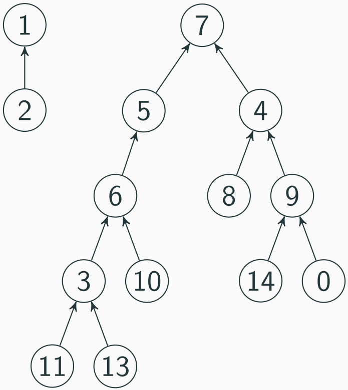
Net effect? Our trees stay relatively balanced.

So, what are the worst-case runtimes now?

- **`makeSet(x)`**:
  $\mathcal{O}(1)$ – still the same
- **`findSet(x)`**:
  $\mathcal{O}(\log(n))$ – since the tree is balanced
- **`union(x, y)`**:
  $\mathcal{O}(\log(n))$ – since union calls `findSet`

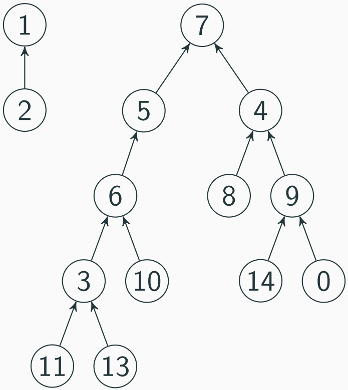
Path compression

Consider the following forest:



Path compression

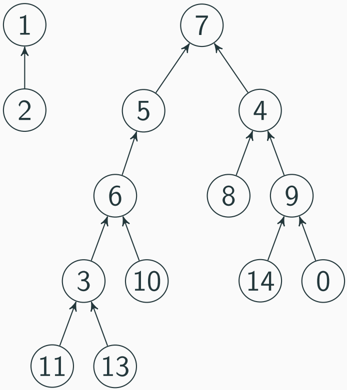
Consider the following forest:



Suppose we call `findSet(3)` a few hundred times.

Path compression

Consider the following forest:

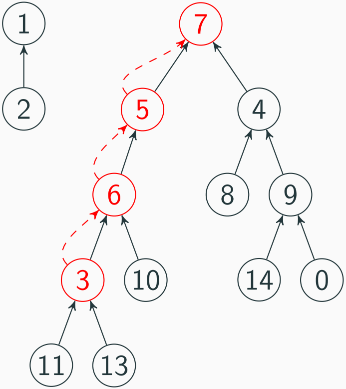


Suppose we call `findSet(3)` a few hundred times.

Why do we have to keep finding the root again and again?

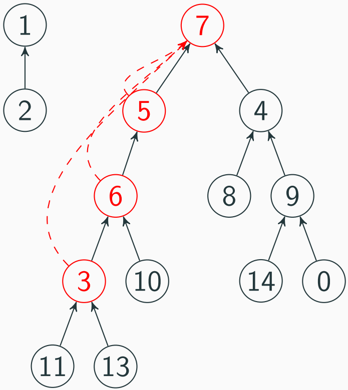
Path compression

Observation: To find root, we must also traverse these nodes:



Path compression

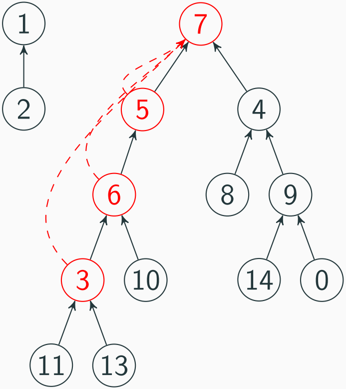
Observation: To find root, we must also traverse these nodes:



What if, next time, we could just jump straight to the root?

Path compression

Observation: To find root, we must also traverse these nodes:

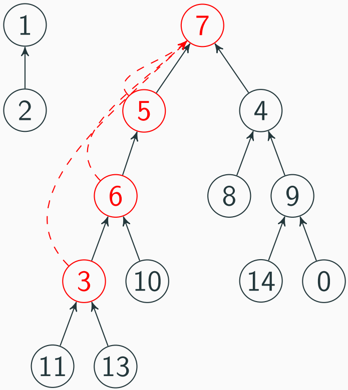


What if, next time, we could just jump straight to the root?

Same for the other nodes we visited

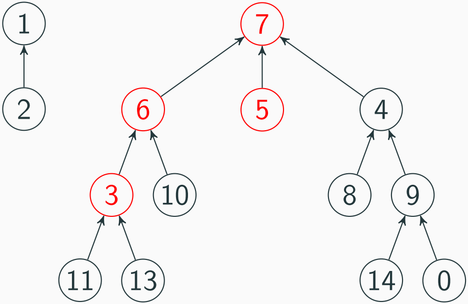
Path compression

So, let's do it!



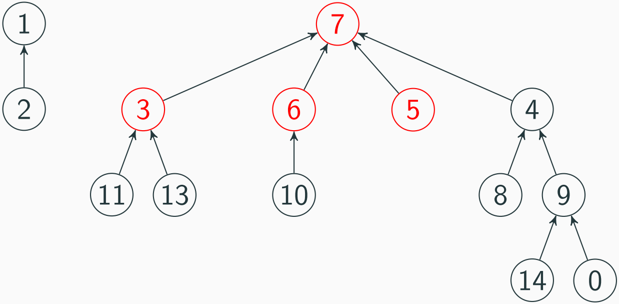
Path compression

So, let's do it!



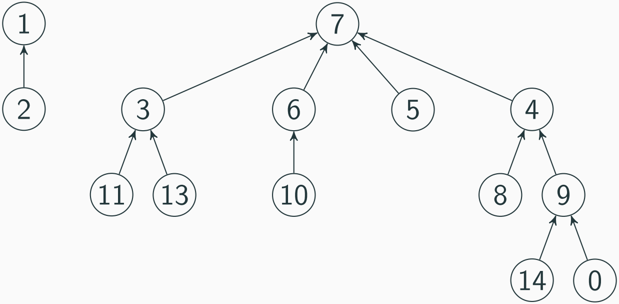
Path compression

So, let's do it!



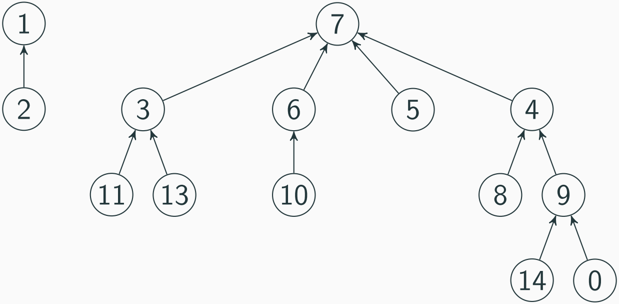
Path compression

So, let's do it!



Path compression

So, let's do it!



Now what happens if we try calling `findSet(3)`?

Path compression

One additional note: path compression changes the heights of our trees.

This means it could be the case that rank $\neq$ height.

Is this a problem?

Path compression

One additional note: path compression changes the heights of our trees.

This means it could be the case that rank $\neq$ height.

Is this a problem?

Answer: No; proof is beyond the scope of this class

Path compression: runtime

Now, what are the worst-case and best-case runtime of the following?

- **`makeSet(x)`**:
- **`findSet(x)`**:
- **`union(x, y)`**:

Path compression: runtime

Now, what are the worst-case and best-case runtime of the following?

- **`makeSet(x)`**:
  $\mathcal{O}(1)$ – still the same
- **`findSet(x)`**:
  In the best case, $\mathcal{O}(1)$, in the worst case $\mathcal{O}(\log(n))$
- **`union(x, y)`**:
  In the best case, $\mathcal{O}(1)$, in the worst case $\mathcal{O}(\log(n))$

Back to Kruskal's

Why are we doing this? To help us implement Kruskal's algorithm:

```
def kruskal():
    for (v : vertices):
        makeMST(v)

    sort edges in ascending order by their weight

    mst = new SomeSet<Edge>()
    for (edge : edges):
        if findMST(edge.src) != findMST(edge.dst):
            union(edge.src, edge.dst)
            mst.add(edge)

    return mst
```

- `makeMST(v)` takes $\mathcal{O}(t_m)$ time
- `findMST(v)`: takes $\mathcal{O}(t_f)$ time
- `union(u, v)`: takes $\mathcal{O}(t_u)$ time

Back to Kruskal's

We concluded that the runtime is:

$$\mathcal{O}\left(\underbrace{|V| \cdot t_m}_{\text{setup}} + \underbrace{|E| \cdot \log(|E|)}_{\text{sorting edges}} + \underbrace{|E| \cdot t_f + |V| \cdot t_u}_{\text{core loop}}\right)$$

Back to Kruskal's

We concluded that the runtime is:

$$\mathcal{O}\left(\underbrace{|V| \cdot t_m}_{\text{setup}} + \underbrace{|E| \cdot \log(|E|)}_{\text{sorting edges}} + \underbrace{|E| \cdot t_f + |V| \cdot t_u}_{\text{core loop}}\right)$$

Well, we just said that in the worst case:

- $t_m \in \mathcal{O}(1)$
- $t_f \in \mathcal{O}(\log(|V|))$
- $t_u \in \mathcal{O}(\log(|V|))$

Back to Kruskal's

We concluded that the runtime is:

$$\mathcal{O}\left(\underbrace{|V| \cdot t_m}_{\text{setup}} + \underbrace{|E| \cdot \log(|E|)}_{\text{sorting edges}} + \underbrace{|E| \cdot t_f + |V| \cdot t_u}_{\text{core loop}}\right)$$

Well, we just said that in the worst case:

- $t_m \in \mathcal{O}(1)$
- $t_f \in \mathcal{O}(\log(|V|))$
- $t_u \in \mathcal{O}(\log(|V|))$

So the worst-case overall runtime of Kruskal's is:

$$\mathcal{O}(|V| + |E| \cdot \log(|E|) + (|E| + |V|) \cdot \log(|V|))$$

Back to Kruskal's

Our worst-case runtime:

$$\mathcal{O}\left(|V| + |E| \cdot \log(|E|) + (|E| + |V|) \cdot \log(|V|)\right)$$

Back to Kruskal's

Our worst-case runtime:

$$\mathcal{O}\left(|V| + |E| \cdot \log(|E|) + (|E| + |V|) \cdot \log(|V|)\right)$$

One minor improvement: since our edge weights are numbers, we can likely use a *linear sort* and improve the runtime to:

$$\mathcal{O}\left(|V| + |E| + (|E| + |V|) \cdot \log(|V|)\right)$$

Back to Kruskal's

Our worst-case runtime:

$$\mathcal{O}\left(|V| + |E| \cdot \log(|E|) + (|E| + |V|) \cdot \log(|V|)\right)$$

One minor improvement: since our edge weights are numbers, we can likely use a *linear sort* and improve the runtime to:

$$\mathcal{O}\left(|V| + |E| + (|E| + |V|) \cdot \log(|V|)\right)$$

We can drop the $|V| + |E|$, since they're dominated by the last term:

$$\mathcal{O}\left((|E| + |V|) \cdot \log(|V|)\right)$$

Back to Kruskal's

Our worst-case runtime:

$$\mathcal{O}\left(|V| + |E| \cdot \log(|E|) + (|E| + |V|) \cdot \log(|V|)\right)$$

One minor improvement: since our edge weights are numbers, we can likely use a *linear sort* and improve the runtime to:

$$\mathcal{O}\left(|V| + |E| + (|E| + |V|) \cdot \log(|V|)\right)$$

We can drop the $|V| + |E|$, since they're dominated by the last term:

$$\mathcal{O}\left((|E| + |V|) \cdot \log(|V|)\right)$$

...and we're left with something that's basically the same as Prim's algorithm.

Disjoint-sets, amortized analysis

…or are we?

Disjoint-sets, amortized analysis

...or are we?

Observation: each call to `findSet(x)` improves all future calls.

How much of a difference does that make?

Disjoint-sets, amortized analysis

...or are we?

Observation: each call to `findSet(x)` improves all future calls.
How much of a difference does that make?

Interesting result:

It turns out `union` and `find` are *amortized* $\log^*(n)$.

Disjoint-sets, amortized analysis

Iterated log

The expression $\log^*(n)$ is equivalent to the number of times you need to compute $\log(x)$ to bring the value down to at most 1

Disjoint-sets, amortized analysis

Iterated log

The expression $\log^*(n)$ is equivalent to the number of times you need to compute $\log(x)$ to bring the value down to at most 1

Example:

- $\log^*(2) = \log(2) = 1$
- $\log^*(4) = \log(\log(4)) = 2$
- $\log^*(8) = \log(\log(\log(8))) = 3$
- $\log^*(65536) = \log^*(2^{2^{2^{2}}}) = 4$
- $\log^*(2^{65536}) = \ldots = 5$

A big number

What is 2^{65536}?

$2^{65536} =$

2003529930406846464979072351560255750447825475569751419
2650169737108940595563114530895061308809333481010382343429072
6318182294938211881266886950636476154702916504187191635158796
6347219442930927982084309104855990570159318959639524863372367
2030029169695921561087649488892540908059114570376752085002066
7156370236612635974714480711177481588091413574272096719015183
6282560618091458852699826141425030123391108273603843767876449
0432059603791244909057075603140350761625624760318637931264847
0374378295497561377098160461441330869211810248595915238019533
1030292162800160568670105651646750568038741529463842244845292
5373614425336143737290883037946012747249584148649159306472520
1515569392262818069165079638106413227530726714399815850881129
2628901134237782705567421080070065283963322155077831214288551

A big number

4376370598692891375715374000198639433246489005254310662966916
5243419174691389632476560289415199775477703138064781342309596
1909606545913008901888875880847336259560654448885014473357060
5881709016210849971452956834406197969056546981363116205357936
9791403236328496233046421066136200220175787851857409162050489
7117818204001872829399434461862243280098373237649318147898481
1945271300744022076568091037620399920349202390662626449190916
7985461515778839060397720759279378852241294301017458086862263
3692847258514030396155585643303854506886522131148136384083847
7826379045960718687672850976347127198889068047824323039471865
0525660978150729861141430305816927924971409161059417185352275
8875044775922183011587807019755357222414000195481020056617735
8978149953232520858975346354700778669040642901676380816174055
0405117670093673202804549339027992491867306539931640720492238
4748152806191669009338057321208163507076343516698696250209690

A big number

6340696503084422585596703927186946115851379338647569974856867
00798239606043934788850861649260304945061743412365828352144806
7266768418070837548622114082365798029612000274413244384324023
3125740354501935242877643088023285085588608996277445816468085
7875115807014743763867976955049991643998284357290415378143438
8473034842619033888414940313661398542576355771053355802066221
8557706008255128889333222643628198483861323957067619140963853
3832374343758830859233722284644287996245605476932428998432652
6773783731732880632107532112386806046747084280511664887090847
7029120816110491255559832236624486855665140268464120969498259
0565519216188104341226838996283071654868525536914850299539675
5039549383718534059000961874894739928804324963731657538036735
8671017578399481847179849824694806053208199606618343401247609
6639519778021441199752546704080608499344178256285092726523709
8986515394621930046073645079262129759176982938923670151709920

A big number

...I got tired of copying and pasting, but we're not even a fourth of the way through.

A big number

...I got tired of copying and pasting, but we're not even a fourth of the way through.

Punchline? $\log^*(n) \leq 5$, for basically any reasonable value of n.

A big number

...I got tired of copying and pasting, but we're not even a fourth of the way through.

Punchline? $\log^*(n) \leq 5$, for basically any reasonable value of n.

Runtime of Kruskal? $\mathcal{O}\left((|E| + |V|)\log^*(|V|)\right) \approx \mathcal{O}\left(|E| + |V|\right)$

Inverse of the Ackerman function

But wait!

Somebody then came along and proved that find and union are amortized $\mathcal{O}(\alpha(n))$ – the inverse of the Ackermann function.

This grows even more slowly then $\log^*(n)$!