The union-find data structure
COMS20017 (Algorithms and Data)

John Lapinskas, University of Bristol

Last time...

A **union-find data structure** supports the following operations:

- `MakeUnionFind`(X): Makes a new union-find data structure containing a 1-element set $\{x\}$ for each element $x \in X$. Takes $O(|X|)$ time.
- `Union`(x, y): Merge the set containing x with the set containing y into a single set in the data structure. Takes $O(\log |X|)$ time.
- `FindSet`(x): Returns a unique identifier for the set containing x. Takes $O(\log |X|)$ time.

Set identifiers can be anything as long as they're unique.

If we implement the sets as linked lists, then `FindSet` is too slow. If we implement them as arrays, then `Union` is too slow.

We'll take the pointer structure of a linked list to make `Union` fast, but arrange it differently to make `FindSet` fast as well.

The idea

We will implement the data structure not as a set of linked lists, but as a **forest** in which the elements are vertices and the sets are **components**.

The idea

We will implement the data structure not as a set of linked lists, but as a **forest** in which the elements are vertices and the sets are **components**.

$$\texttt{MakeUnionFind}(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8);$$

The idea

We will implement the data structure not as a set of linked lists, but as a **forest** in which the elements are vertices and the sets are **components**.

$$\texttt{MakeUnionFind}(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8);$$

The idea

We will implement the data structure not as a set of linked lists, but as a **forest** in which the elements are vertices and the sets are **components**.

$$\texttt{MakeUnionFind}(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8);$$

x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8

- $\texttt{MakeUnionFind}(X)$ makes an isolated vertex for each element of X.

The idea

We will implement the data structure not as a set of linked lists, but as a **forest** in which the elements are vertices and the sets are **components**.

`FindSet`(x_3);

- `MakeUnionFind`(X) makes an isolated vertex for each element of X.

The idea

We will implement the data structure not as a set of linked lists, but as a **forest** in which the elements are vertices and the sets are **components**.

`FindSet`(x_3); Returns x_3.

- `MakeUnionFind`(X) makes an isolated vertex for each element of X.

The idea

We will implement the data structure not as a set of linked lists, but as a **forest** in which the elements are vertices and the sets are **components**.

$$\texttt{Union}(x_1, x_2);$$

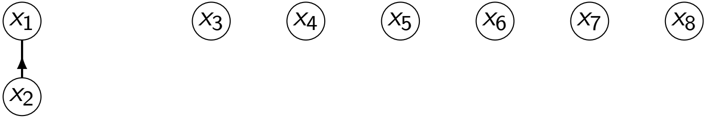
$$x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6 \quad x_7 \quad x_8$$

- $\texttt{MakeUnionFind}(X)$ makes an isolated vertex for each element of X.

The idea

We will implement the data structure not as a set of linked lists, but as a **forest** in which the elements are vertices and the sets are **components**.

`Union`(x_1, x_2);

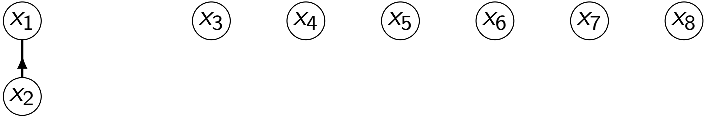


- `MakeUnionFind`(X) makes an isolated vertex for each element of X.

The idea

We will implement the data structure not as a set of linked lists, but as a **forest** in which the elements are vertices and the sets are **components**.

$\texttt{Union}(x_1, x_4);$



- $\texttt{MakeUnionFind}(X)$ makes an isolated vertex for each element of X.

The idea

We will implement the data structure not as a set of linked lists, but as a **forest** in which the elements are vertices and the sets are **components**.

$\texttt{Union}(x_1, x_4);$

- $\texttt{MakeUnionFind}(X)$ makes an isolated vertex for each element of X.

The idea

We will implement the data structure not as a set of linked lists, but as a **forest** in which the elements are vertices and the sets are **components**.

`FindSet`(x_4);

- `MakeUnionFind`(X) makes an isolated vertex for each element of X.

The idea

We will implement the data structure not as a set of linked lists, but as a **forest** in which the elements are vertices and the sets are **components**.

`FindSet`(x_4);

- `MakeUnionFind`(X) makes an isolated vertex for each element of X.

The idea

We will implement the data structure not as a set of linked lists, but as a **forest** in which the elements are vertices and the sets are **components**.

`FindSet`(x_4);

- `MakeUnionFind`(X) makes an isolated vertex for each element of X.

The idea

We will implement the data structure not as a set of linked lists, but as a **forest** in which the elements are vertices and the sets are **components**.

`FindSet`(x_4); Returns x_1.

- `MakeUnionFind`(X) makes an isolated vertex for each element of X.

The idea

We will implement the data structure not as a set of linked lists, but as a **forest** in which the elements are vertices and the sets are **components**.

`FindSet`(x_4); Returns x_1.

- `MakeUnionFind`(X) makes an isolated vertex for each element of X.
- `FindSet`(x) returns the root of x's tree as its identifier.

The idea

We will implement the data structure not as a set of linked lists, but as a **forest** in which the elements are vertices and the sets are **components**.

$\texttt{Union}(x_5, x_8)$; $\texttt{Union}(x_8, x_7)$; $\texttt{Union}(x_6, x_8)$;

- $\texttt{MakeUnionFind}(X)$ makes an isolated vertex for each element of X.
- $\texttt{FindSet}(x)$ returns the root of x's tree as its identifier.

The idea

We will implement the data structure not as a set of linked lists, but as a **forest** in which the elements are vertices and the sets are **components**.

$\texttt{Union}(x_5, x_8)$; $\texttt{Union}(x_8, x_7)$; $\texttt{Union}(x_6, x_8)$;

- $\texttt{MakeUnionFind}(X)$ makes an isolated vertex for each element of X.
- $\texttt{FindSet}(x)$ returns the root of x's tree as its identifier.

The idea

We will implement the data structure not as a set of linked lists, but as a **forest** in which the elements are vertices and the sets are **components**.

$\texttt{Union}(x_4, x_7);$

- $\texttt{MakeUnionFind}(X)$ makes an isolated vertex for each element of X.
- $\texttt{FindSet}(x)$ returns the root of x's tree as its identifier.

The idea

We will implement the data structure not as a set of linked lists, but as a **forest** in which the elements are vertices and the sets are **components**.

`Union`(x_4, x_7);

- `MakeUnionFind`(X) makes an isolated vertex for each element of X.
- `FindSet`(x) returns the root of x's tree as its identifier.
- `Union`(x_i, x_j) puts the **root** of x_i under the **root** of x_j (or vice versa).

The idea

We will implement the data structure not as a set of linked lists, but as a **forest** in which the elements are vertices and the sets are **components**.

`Union`(x_4, x_7);

- `MakeUnionFind`(X) makes an isolated vertex for each element of X.
- `FindSet`(x) returns the root of x's tree as its identifier.
- `Union`(x_i, x_j) puts the **root** of x_i under the **root** of x_j (or vice versa).

The idea

We will implement the data structure not as a set of linked lists, but as a **forest** in which the elements are vertices and the sets are **components**.

`Union`(x_4, x_7);

- `MakeUnionFind`(X) makes an isolated vertex for each element of X.
- `FindSet`(x) returns the root of x's tree as its identifier.
- `Union`(x_i, x_j) puts the **root** of x_i under the **root** of x_j (or vice versa).

The idea

We will implement the data structure not as a set of linked lists, but as a **forest** in which the elements are vertices and the sets are **components**.

`Union`(x_4, x_7);

- `MakeUnionFind`(X) makes an isolated vertex for each element of X.
- `FindSet`(x) returns the root of x's tree as its identifier.
- `Union`(x_i, x_j) puts the **root** of x_i under the **root** of x_j (or vice versa).

The idea

We will implement the data structure not as a set of linked lists, but as a **forest** in which the elements are vertices and the sets are **components**.

$\texttt{Union}(x_4, x_7);$

- $\texttt{MakeUnionFind}(X)$ makes an isolated vertex for each element of X.
- $\texttt{FindSet}(x)$ returns the root of x's tree as its identifier.
- $\texttt{Union}(x_i, x_j)$ puts the **root** of x_i under the **root** of x_j (or vice versa).

The idea

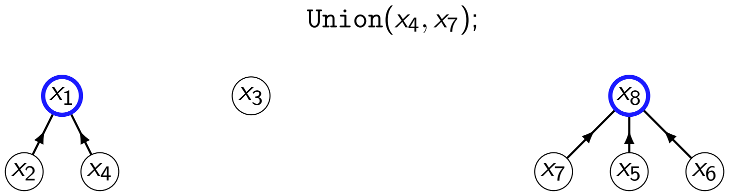
We will implement the data structure not as a set of linked lists, but as a **forest** in which the elements are vertices and the sets are **components**.

$\texttt{Union}(x_4, x_7);$

- $\texttt{MakeUnionFind}(X)$ makes an isolated vertex for each element of X.
- $\texttt{FindSet}(x)$ returns the root of x's tree as its identifier.
- $\texttt{Union}(x_i, x_j)$ puts the **root** of x_i under the **root** of x_j (or vice versa).

The idea

We will implement the data structure not as a set of linked lists, but as a **forest** in which the elements are vertices and the sets are **components**.

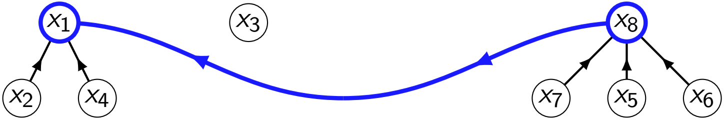


- MakeUnionFind(X) makes an isolated vertex for each element of X.
- FindSet(x) returns the root of x's tree as its identifier.
- Union(x_i, x_j) puts the **root** of x_i under the **root** of x_j (or vice versa).

The idea

We will implement the data structure not as a set of linked lists, but as a **forest** in which the elements are vertices and the sets are **components**.

$\texttt{Union}(x_4, x_7);$

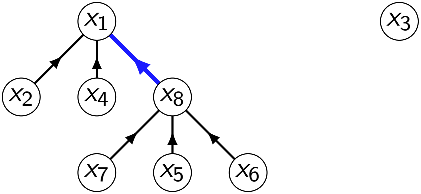


- $\texttt{MakeUnionFind}(X)$ makes an isolated vertex for each element of X.
- $\texttt{FindSet}(x)$ returns the root of x's tree as its identifier.
- $\texttt{Union}(x_i, x_j)$ puts the **root** of x_i under the **root** of x_j (or vice versa).

The idea

We will implement the data structure not as a set of linked lists, but as a **forest** in which the elements are vertices and the sets are **components**.

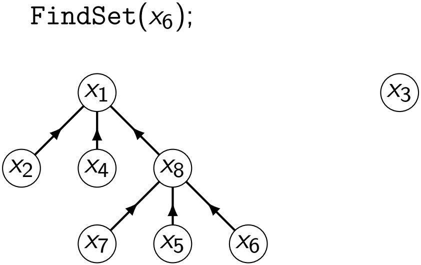
$\texttt{Union}(x_4, x_7);$



- $\texttt{MakeUnionFind}(X)$ makes an isolated vertex for each element of X.
- $\texttt{FindSet}(x)$ returns the root of x's tree as its identifier.
- $\texttt{Union}(x_i, x_j)$ puts the **root** of x_i under the **root** of x_j (or vice versa).

The idea

We will implement the data structure not as a set of linked lists, but as a **forest** in which the elements are vertices and the sets are **components**.

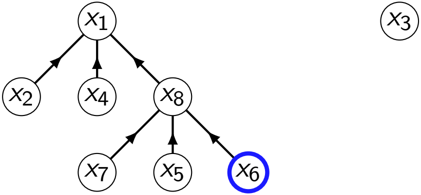


- `MakeUnionFind`(X) makes an isolated vertex for each element of X.
- `FindSet`(x) returns the root of x's tree as its identifier.
- `Union`(x_i, x_j) puts the **root** of x_i under the **root** of x_j (or vice versa).

The idea

We will implement the data structure not as a set of linked lists, but as a **forest** in which the elements are vertices and the sets are **components**.

`FindSet`(x_6);

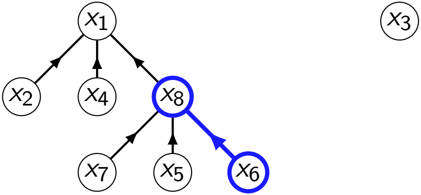


- `MakeUnionFind`(X) makes an isolated vertex for each element of X.
- `FindSet`(x) returns the root of x's tree as its identifier.
- `Union`(x_i, x_j) puts the **root** of x_i under the **root** of x_j (or vice versa).

The idea

We will implement the data structure not as a set of linked lists, but as a **forest** in which the elements are vertices and the sets are **components**.

`FindSet`(x_6);

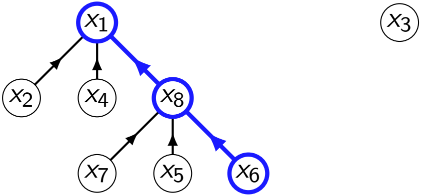


- `MakeUnionFind`(X) makes an isolated vertex for each element of X.
- `FindSet`(x) returns the root of x's tree as its identifier.
- `Union`(x_i, x_j) puts the **root** of x_i under the **root** of x_j (or vice versa).

The idea

We will implement the data structure not as a set of linked lists, but as a **forest** in which the elements are vertices and the sets are **components**.

`FindSet`(x_6);

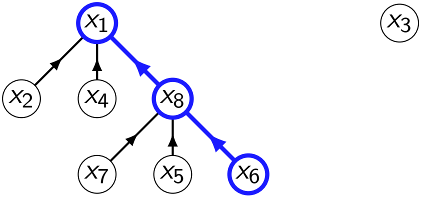


- `MakeUnionFind`(X) makes an isolated vertex for each element of X.
- `FindSet`(x) returns the root of x's tree as its identifier.
- `Union`(x_i, x_j) puts the **root** of x_i under the **root** of x_j (or vice versa).

The idea

We will implement the data structure not as a set of linked lists, but as a **forest** in which the elements are vertices and the sets are **components**.

`FindSet`(x_6); Returns x_1.

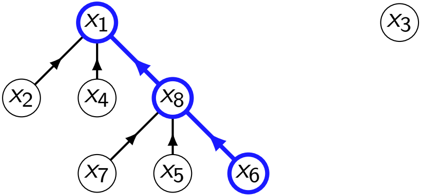


- `MakeUnionFind`(X) makes an isolated vertex for each element of X.
- `FindSet`(x) returns the root of x's tree as its identifier.
- `Union`(x_i, x_j) puts the **root** of x_i under the **root** of x_j (or vice versa).

The idea

We will implement the data structure not as a set of linked lists, but as a **forest** in which the elements are vertices and the sets are **components**.

`FindSet`(x_6); Returns x_1.



- `MakeUnionFind`(X) makes an isolated vertex for each element of X.
- `FindSet`(x) returns the root of x's tree as its identifier.
- `Union`(x_i, x_j) puts the **root** of x_i under the **root** of x_j (or vice versa).

`Union` and `FindSet` both take $\Theta(d)$ time, where d is the maximum depth of the tree components involved. How big can this be?

Forests can degenerate into linked lists

`Union` and `FindSet` both take $\Theta(d)$ time, where d is the maximum depth of the tree components involved. How big can this be?

$$x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad \cdots$$

Forests can degenerate into linked lists

`Union` and `FindSet` both take $\Theta(d)$ time, where d is the maximum depth of the tree components involved. How big can this be?

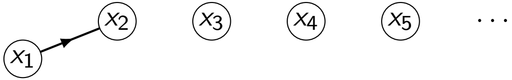
$$\texttt{Union}(x_1, x_2);$$

x_1 x_2 x_3 x_4 x_5 $\cdots$

Forests can degenerate into linked lists

`Union` and `FindSet` both take $\Theta(d)$ time, where d is the maximum depth of the tree components involved. How big can this be?

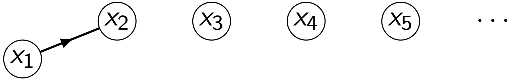
$\texttt{Union}(x_1, x_2);$



Forests can degenerate into linked lists

`Union` and `FindSet` both take $\Theta(d)$ time, where d is the maximum depth of the tree components involved. How big can this be?

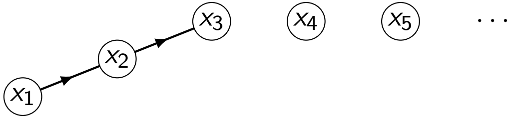
$$\texttt{Union}(x_2, x_3);$$



Forests can degenerate into linked lists

`Union` and `FindSet` both take $\Theta(d)$ time, where d is the maximum depth of the tree components involved. How big can this be?

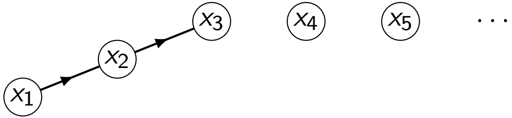
$$\texttt{Union}(x_2, x_3);$$



Forests can degenerate into linked lists

`Union` and `FindSet` both take $\Theta(d)$ time, where d is the maximum depth of the tree components involved. How big can this be?

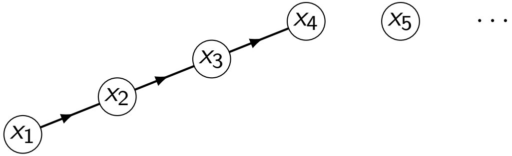
$$\texttt{Union}(x_3, x_4);$$



Forests can degenerate into linked lists

`Union` and `FindSet` both take $\Theta(d)$ time, where d is the maximum depth of the tree components involved. How big can this be?

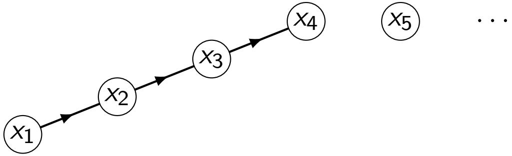
$$\texttt{Union}(x_3, x_4);$$



Forests can degenerate into linked lists

`Union` and `FindSet` both take $\Theta(d)$ time, where d is the maximum depth of the tree components involved. How big can this be?

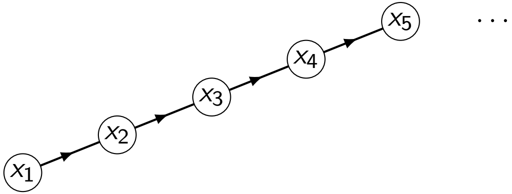
$$\texttt{Union}(x_4, x_5);$$



Forests can degenerate into linked lists

Union and FindSet both take $\Theta(d)$ time, where d is the maximum depth of the tree components involved. How big can this be?

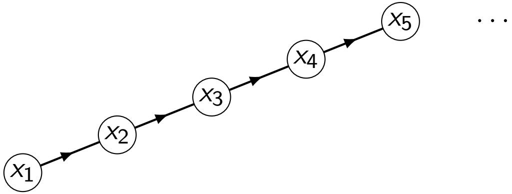
$\texttt{Union}(x_4, x_5);$



Forests can degenerate into linked lists

`Union` and `FindSet` both take $\Theta(d)$ time, where d is the maximum depth of the tree components involved. How big can this be?

$$\texttt{Union}(x_4, x_5);$$



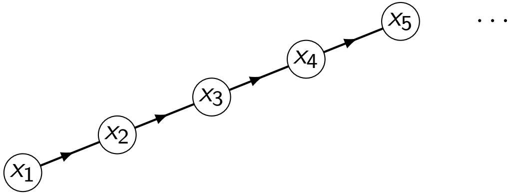
Recall $\texttt{Union}(x_i, x_j)$ puts the root of x_i under the root of x_j, or vice versa.

If we make bad choices of which root goes under which, like the above, we may have $d \in \Theta(|X|)$. How can we prevent this?

Forests can degenerate into linked lists

`Union` and `FindSet` both take $\Theta(d)$ time, where d is the maximum depth of the tree components involved. How big can this be?

$\texttt{Union}(x_4, x_5);$



Recall $\texttt{Union}(x_i, x_j)$ puts the root of x_i under the root of x_j, or vice versa.

If we make bad choices of which root goes under which, like the above, we may have $d \in \Theta(|X|)$. How can we prevent this?

Always put the tree with **lower** depth under the tree with **higher** depth! This way, d only increases if the two have equal depth.

Forests can degenerate into linked lists

`Union` and `FindSet` both take $\Theta(d)$ time, where d is the maximum depth of the tree components involved. How big can this be?

Recall $\texttt{Union}(x_i, x_j)$ puts the root of x_i under the root of x_j, or vice versa.

If we make bad choices of which root goes under which, like the above, we may have $d \in \Theta(|X|)$. How can we prevent this?

Always put the tree with **lower** depth under the tree with **higher** depth! This way, d only increases if the two have equal depth.

Forests can degenerate into linked lists

`Union` and `FindSet` both take $\Theta(d)$ time, where d is the maximum depth of the tree components involved. How big can this be?

$$\texttt{Union}(x_1, x_2);$$

x_1 x_2 x_3 x_4 x_5 $\cdots$

Recall $\texttt{Union}(x_i, x_j)$ puts the root of x_i under the root of x_j, or vice versa.

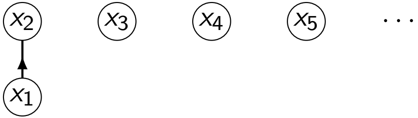
If we make bad choices of which root goes under which, like the above, we may have $d \in \Theta(|X|)$. How can we prevent this?

Always put the tree with **lower** depth under the tree with **higher** depth! This way, d only increases if the two have equal depth.

Forests can degenerate into linked lists

`Union` and `FindSet` both take $\Theta(d)$ time, where d is the maximum depth of the tree components involved. How big can this be?

$\texttt{Union}(x_1, x_2);$



Recall $\texttt{Union}(x_i, x_j)$ puts the root of x_i under the root of x_j, or vice versa.

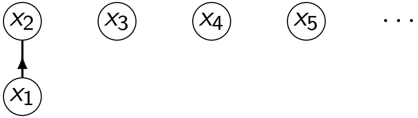
If we make bad choices of which root goes under which, like the above, we may have $d \in \Theta(|X|)$. How can we prevent this?

Always put the tree with **lower** depth under the tree with **higher** depth! This way, d only increases if the two have equal depth.

Forests can degenerate into linked lists

`Union` and `FindSet` both take $\Theta(d)$ time, where d is the maximum depth of the tree components involved. How big can this be?

$\texttt{Union}(x_2, x_3);$



Recall $\texttt{Union}(x_i, x_j)$ puts the root of x_i under the root of x_j, or vice versa.

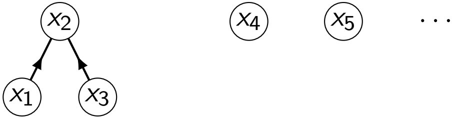
If we make bad choices of which root goes under which, like the above, we may have $d \in \Theta(|X|)$. How can we prevent this?

Always put the tree with **lower** depth under the tree with **higher** depth! This way, d only increases if the two have equal depth.

Forests can degenerate into linked lists

`Union` and `FindSet` both take $\Theta(d)$ time, where d is the maximum depth of the tree components involved. How big can this be?

$\texttt{Union}(x_2, x_3);$



Recall $\texttt{Union}(x_i, x_j)$ puts the root of x_i under the root of x_j, or vice versa.

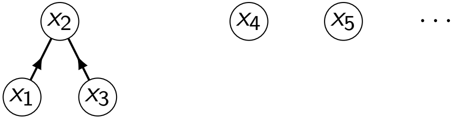
If we make bad choices of which root goes under which, like the above, we may have $d \in \Theta(|X|)$. How can we prevent this?

Always put the tree with **lower** depth under the tree with **higher** depth! This way, d only increases if the two have equal depth.

Forests can degenerate into linked lists

`Union` and `FindSet` both take $\Theta(d)$ time, where d is the maximum depth of the tree components involved. How big can this be?

$\texttt{Union}(x_3, x_4)$;



Recall $\texttt{Union}(x_i, x_j)$ puts the root of x_i under the root of x_j, or vice versa.

If we make bad choices of which root goes under which, like the above, we may have $d \in \Theta(|X|)$. How can we prevent this?

Always put the tree with **lower** depth under the tree with **higher** depth! This way, d only increases if the two have equal depth.

Forests can degenerate into linked lists

Union and FindSet both take $\Theta(d)$ time, where d is the maximum depth of the tree components involved. How big can this be?

$\texttt{Union}(x_3, x_4);$

Recall $\texttt{Union}(x_i, x_j)$ puts the root of x_i under the root of x_j, or vice versa.

If we make bad choices of which root goes under which, like the above, we may have $d \in \Theta(|X|)$. How can we prevent this?

Always put the tree with **lower** depth under the tree with **higher** depth! This way, d only increases if the two have equal depth.

Forests can degenerate into linked lists

`Union` and `FindSet` both take $\Theta(d)$ time, where d is the maximum depth of the tree components involved. How big can this be?

$\texttt{Union}(x_4, x_5);$

Recall $\texttt{Union}(x_i, x_j)$ puts the root of x_i under the root of x_j, or vice versa.

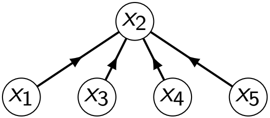
If we make bad choices of which root goes under which, like the above, we may have $d \in \Theta(|X|)$. How can we prevent this?

Always put the tree with **lower** depth under the tree with **higher** depth! This way, d only increases if the two have equal depth.

Forests can degenerate into linked lists

`Union` and `FindSet` both take $\Theta(d)$ time, where d is the maximum depth of the tree components involved. How big can this be?

$\texttt{Union}(x_4, x_5);$



...

Recall $\texttt{Union}(x_i, x_j)$ puts the root of x_i under the root of x_j, or vice versa.

If we make bad choices of which root goes under which, like the above, we may have $d \in \Theta(|X|)$. How can we prevent this?

Always put the tree with **lower** depth under the tree with **higher** depth! This way, d only increases if the two have equal depth.

Proof that $d = O(\log |X|)$

`Union`(x_1, x_2) follows pointers from x_1 and x_2 up to the roots r_1 and r_2 of their trees. If x_1's tree has lower depth than x_2's tree, then it adds r_1 as a child of r_2; if not, it adds r_2 as a child of r_1.

Proof that $d = O(\log |X|)$

`Union`(x_1, x_2) follows pointers from x_1 and x_2 up to the roots r_1 and r_2 of their trees. If x_1's tree has lower depth than x_2's tree, then it adds r_1 as a child of r_2; if not, it adds r_2 as a child of r_1.

Writing d_1 for the depth of x_1's tree, and d_2 for the depth of x_2's tree, if $d_1 < d_2$ then the depth of the new tree will be $\max\{d_2, d_1 + 1\} = d_2$.

Likewise, if $d_2 < d_1$ then the new depth will be $\max\{d_1, d_2 + 1\} = d_1$.

The depth only increases if $d_1 = d_2$.

Proof that $d = O(\log |X|)$

`Union`(x_1, x_2) follows pointers from x_1 and x_2 up to the roots r_1 and r_2 of their trees. If x_1's tree has lower depth than x_2's tree, then it adds r_1 as a child of r_2; if not, it adds r_2 as a child of r_1.

Writing d_1 for the depth of x_1's tree, and d_2 for the depth of x_2's tree, if $d_1 < d_2$ then the depth of the new tree will be $\max\{d_2, d_1 + 1\} = d_2$.

Likewise, if $d_2 < d_1$ then the new depth will be $\max\{d_1, d_2 + 1\} = d_1$.

The depth only increases if $d_1 = d_2$.

Lemma: If the data structure contains a tree of depth d, then it has at least 2^d vertices in total.

Proof that $d = O(\log |X|)$

`Union`(x_1, x_2) follows pointers from x_1 and x_2 up to the roots r_1 and r_2 of their trees. If x_1's tree has lower depth than x_2's tree, then it adds r_1 as a child of r_2; if not, it adds r_2 as a child of r_1.

Writing d_1 for the depth of x_1's tree, and d_2 for the depth of x_2's tree, if $d_1 < d_2$ then the depth of the new tree will be $\max\{d_2, d_1 + 1\} = d_2$.

Likewise, if $d_2 < d_1$ then the new depth will be $\max\{d_1, d_2 + 1\} = d_1$.

The depth only increases if $d_1 = d_2$.

Lemma: If the data structure contains a tree of depth d, then it has at least 2^d vertices in total.

Proof: By induction on d.

Proof that $d = O(\log |X|)$

`Union`(x_1, x_2) follows pointers from x_1 and x_2 up to the roots r_1 and r_2 of their trees. If x_1's tree has lower depth than x_2's tree, then it adds r_1 as a child of r_2; if not, it adds r_2 as a child of r_1.

Writing d_1 for the depth of x_1's tree, and d_2 for the depth of x_2's tree, if $d_1 < d_2$ then the depth of the new tree will be $\max\{d_2, d_1 + 1\} = d_2$.

Likewise, if $d_2 < d_1$ then the new depth will be $\max\{d_1, d_2 + 1\} = d_1$.

The depth only increases if $d_1 = d_2$.

Lemma: If the data structure contains a tree of depth d, then it has at least 2^d vertices in total.

Proof: By induction on d.

Base case: If $d = 0$, then the tree is a single vertex.

Proof that $d = O(\log|X|)$

`Union`(x_1, x_2) follows pointers from x_1 and x_2 up to the roots r_1 and r_2 of their trees. If x_1's tree has lower depth than x_2's tree, then it adds r_1 as a child of r_2; if not, it adds r_2 as a child of r_1.

Writing d_1 for the depth of x_1's tree, and d_2 for the depth of x_2's tree, if $d_1 < d_2$ then the depth of the new tree will be $\max\{d_2, d_1 + 1\} = d_2$.

Likewise, if $d_2 < d_1$ then the new depth will be $\max\{d_1, d_2 + 1\} = d_1$.

The depth only increases if $d_1 = d_2$.

Lemma: If the data structure contains a tree of depth d, then it has at least 2^d vertices in total.

Proof: By induction on d.

Base case: If $d = 0$, then the tree is a single vertex. ✓

Proof that $d = O(\log |X|)$

`Union`(x_1, x_2) follows pointers from x_1 and x_2 up to the roots r_1 and r_2 of their trees. If x_1's tree has lower depth than x_2's tree, then it adds r_1 as a child of r_2; if not, it adds r_2 as a child of r_1.

Writing d_1 for the depth of x_1's tree, and d_2 for the depth of x_2's tree, if $d_1 < d_2$ then the depth of the new tree will be $\max\{d_2, d_1 + 1\} = d_2$.

Likewise, if $d_2 < d_1$ then the new depth will be $\max\{d_1, d_2 + 1\} = d_1$.

The depth only increases if $d_1 = d_2$.

Lemma: If the data structure contains a tree of depth d, then it has at least 2^d vertices in total.

Proof: By induction on d.

Base case: If $d = 0$, then the tree is a single vertex. ✓

Inductive step: A tree of depth $d \geq 1$ must have been formed by merging two trees of depth $d - 1$, each containing 2^{d-1} vertices by the inductive hypothesis.

Proof that $d = O(\log |X|)$

`Union`(x_1, x_2) follows pointers from x_1 and x_2 up to the roots r_1 and r_2 of their trees. If x_1's tree has lower depth than x_2's tree, then it adds r_1 as a child of r_2; if not, it adds r_2 as a child of r_1.

Writing d_1 for the depth of x_1's tree, and d_2 for the depth of x_2's tree, if $d_1 < d_2$ then the depth of the new tree will be $\max\{d_2, d_1 + 1\} = d_2$.

Likewise, if $d_2 < d_1$ then the new depth will be $\max\{d_1, d_2 + 1\} = d_1$.

The depth only increases if $d_1 = d_2$.

Lemma: If the data structure contains a tree of depth d, then it has at least 2^d vertices in total.

Proof: By induction on d.

Base case: If $d = 0$, then the tree is a single vertex. ✓

Inductive step: A tree of depth $d \geq 1$ must have been formed by merging two trees of depth $d - 1$, each containing 2^{d-1} vertices by the inductive hypothesis. So the tree must contain $2 \cdot 2^{d-1} = 2^d$ vertices.

Proof that $d = O(\log |X|)$

`Union`(x_1, x_2) follows pointers from x_1 and x_2 up to the roots r_1 and r_2 of their trees. If x_1's tree has lower depth than x_2's tree, then it adds r_1 as a child of r_2; if not, it adds r_2 as a child of r_1.

Writing d_1 for the depth of x_1's tree, and d_2 for the depth of x_2's tree, if $d_1 < d_2$ then the depth of the new tree will be $\max\{d_2, d_1 + 1\} = d_2$.

Likewise, if $d_2 < d_1$ then the new depth will be $\max\{d_1, d_2 + 1\} = d_1$.

The depth only increases if $d_1 = d_2$.

Lemma: If the data structure contains a tree of depth d, then it has at least 2^d vertices in total.

Proof: By induction on d.

Base case: If $d = 0$, then the tree is a single vertex. ✓

Inductive step: A tree of depth $d \geq 1$ must have been formed by merging two trees of depth $d - 1$, each containing 2^{d-1} vertices by the inductive hypothesis. So the tree must contain $2 \cdot 2^{d-1} = 2^d$ vertices. □

Proof that $d = O(\log |X|)$

`Union`(x_1, x_2) follows pointers from x_1 and x_2 up to the roots r_1 and r_2 of their trees. If x_1's tree has lower depth than x_2's tree, then it adds r_1 as a child of r_2; if not, it adds r_2 as a child of r_1.

Writing d_1 for the depth of x_1's tree, and d_2 for the depth of x_2's tree, if $d_1 < d_2$ then the depth of the new tree will be $\max\{d_2, d_1 + 1\} = d_2$.

Likewise, if $d_2 < d_1$ then the new depth will be $\max\{d_1, d_2 + 1\} = d_1$.

The depth only increases if $d_1 = d_2$.

Lemma: If the data structure contains a tree of depth d, then it has at least 2^d vertices in total.

Proof: By induction on d.

Base case: If $d = 0$, then the tree is a single vertex. ✓

Inductive step: A tree of depth $d \geq 1$ must have been formed by merging two trees of depth $d - 1$, each containing 2^{d-1} vertices by the inductive hypothesis. So the tree must contain $2 \cdot 2^{d-1} = 2^d$ vertices. □

This means any tree with depth greater than $\log |X|$ would contain more than $2^{\log |X|} = |X|$ vertices, which is impossible! So $d \leq \log |X|$.

Summary

Overall, the operations of the union-find data structure are:

- `MakeUnionFind`(X) creates one isolated vertex for each $x \in X$.

Summary

Overall, the operations of the union-find data structure are:

- `MakeUnionFind`(X) creates one isolated vertex for each $x \in X$.
- `FindSet`(x) follows pointers from x up to the root of x's tree, which it returns as a unique identifier.

Summary

Overall, the operations of the union-find data structure are:

- `MakeUnionFind`(X) creates one isolated vertex for each $x \in X$.
- `FindSet`(x) follows pointers from x up to the root of x's tree, which it returns as a unique identifier.
- `Union`(x_1, x_2) follows pointers from x_1 and x_2 up to the roots r_1 and r_2 of their trees. If x_1's tree has lower depth than x_2's tree, then it adds r_1 as a child of r_2; otherwise, it adds r_2 as a child of r_1.

Summary

Overall, the operations of the union-find data structure are:

- `MakeUnionFind`(X) creates one isolated vertex for each $x \in X$.
- `FindSet`(x) follows pointers from x up to the root of x's tree, which it returns as a unique identifier.
- `Union`(x_1, x_2) follows pointers from x_1 and x_2 up to the roots r_1 and r_2 of their trees. If x_1's tree has lower depth than x_2's tree, then it adds r_1 as a child of r_2; otherwise, it adds r_2 as a child of r_1.

`MakeUnionFind` runs in $O(|X|)$ time. All trees in the data structure have height at most $\log|X|$, so `Union` and `FindSet` run in $O(\log|X|)$ time.

Summary

Overall, the operations of the union-find data structure are:

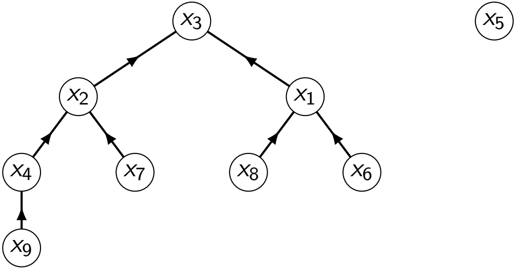
- `MakeUnionFind`(X) creates one isolated vertex for each $x \in X$.
- `FindSet`(x) follows pointers from x up to the root of x's tree, which it returns as a unique identifier.
- `Union`(x_1, x_2) follows pointers from x_1 and x_2 up to the roots r_1 and r_2 of their trees. If x_1's tree has lower depth than x_2's tree, then it adds r_1 as a child of r_2; otherwise, it adds r_2 as a child of r_1.

`MakeUnionFind` runs in $O(|X|)$ time. All trees in the data structure have height at most $\log |X|$, so `Union` and `FindSet` run in $O(\log |X|)$ time.

In particular, we can use this to implement Kruskal's algorithm and Borůvka's algorithm in $O(|E| \log |E|)$ time!

A possible improvement: Path compression

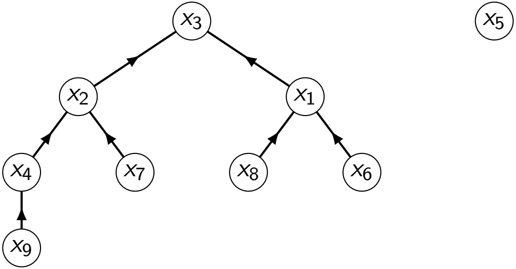
Right now, we are duplicating some work with root-finding.



A possible improvement: Path compression

Right now, we are duplicating some work with root-finding.

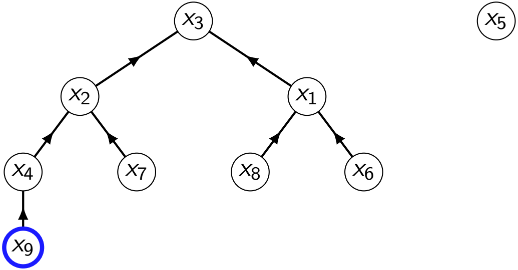
`FindSet`(x_9);



A possible improvement: Path compression

Right now, we are duplicating some work with root-finding.

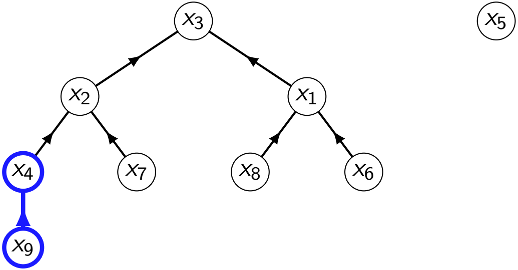
`FindSet`(x_9);



A possible improvement: Path compression

Right now, we are duplicating some work with root-finding.

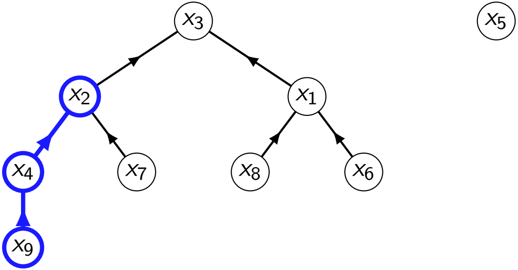
`FindSet`(x_9);



A possible improvement: Path compression

Right now, we are duplicating some work with root-finding.

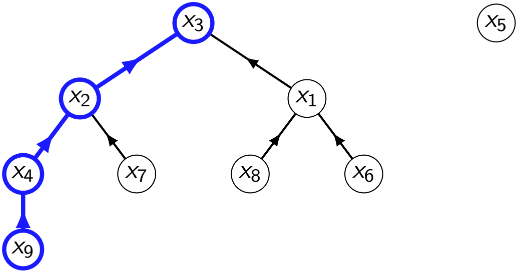
`FindSet`(x_9);



A possible improvement: Path compression

Right now, we are duplicating some work with root-finding.

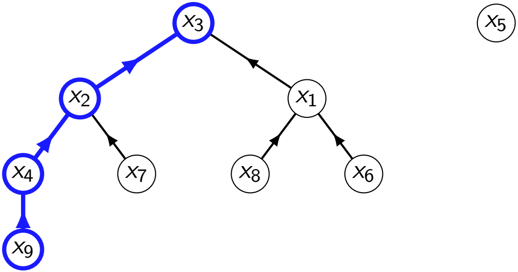
FindSet(x_9);



A possible improvement: Path compression

Right now, we are duplicating some work with root-finding.

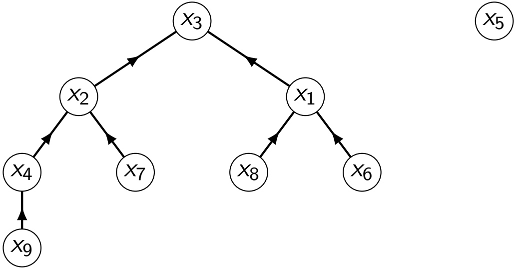
`FindSet(`x_9`);` Returns x_3.



A possible improvement: Path compression

Right now, we are duplicating some work with root-finding.

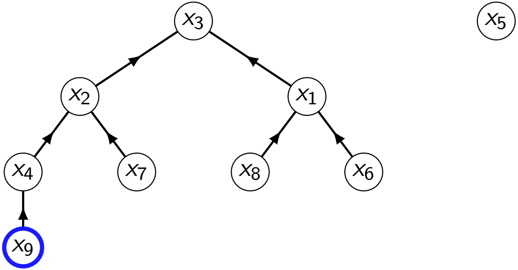
`Union`(x_9, x_5);



A possible improvement: Path compression

Right now, we are duplicating some work with root-finding.

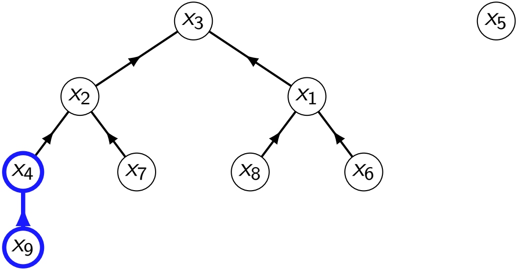
`Union`(x_9, x_5);



A possible improvement: Path compression

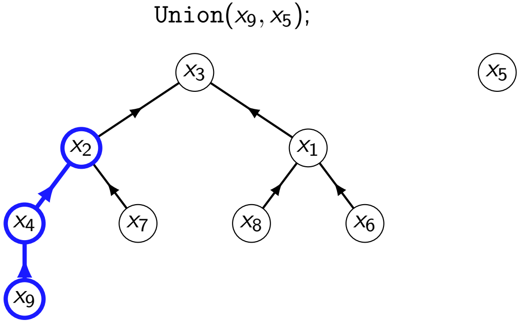
Right now, we are duplicating some work with root-finding.

$\texttt{Union}(x_9, x_5);$



A possible improvement: Path compression

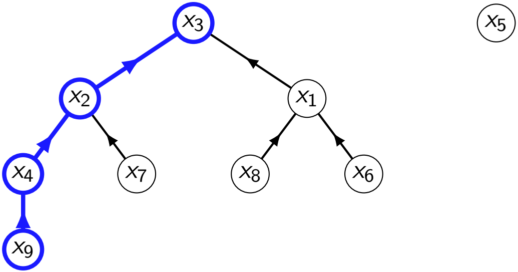
Right now, we are duplicating some work with root-finding.



A possible improvement: Path compression

Right now, we are duplicating some work with root-finding.

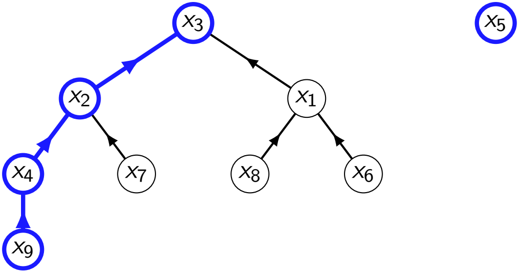
$\texttt{Union}(x_9, x_5);$



A possible improvement: Path compression

Right now, we are duplicating some work with root-finding.

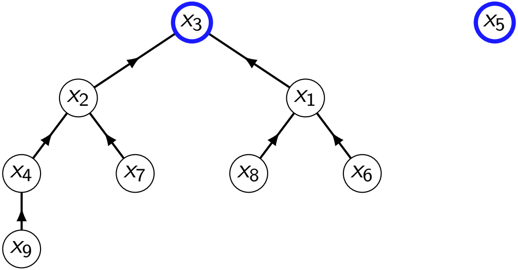
$\texttt{Union}(x_9, x_5);$



A possible improvement: Path compression

Right now, we are duplicating some work with root-finding.

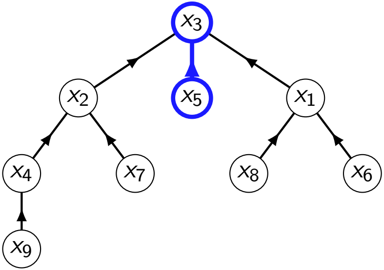
$\texttt{Union}(x_9, x_5);$



A possible improvement: Path compression

Right now, we are duplicating some work with root-finding.

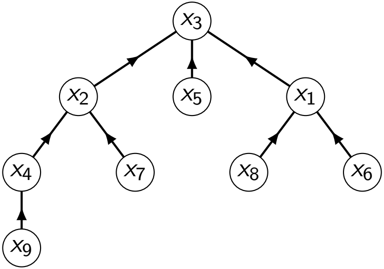
$\texttt{Union}(x_9, x_5);$



A possible improvement: Path compression

Right now, we are duplicating some work with root-finding.

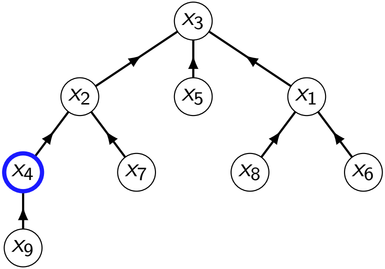
`FindSet`(x_4);



A possible improvement: Path compression

Right now, we are duplicating some work with root-finding.

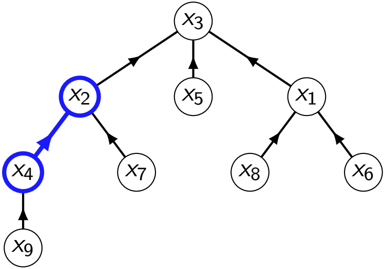
`FindSet`(x_4);



A possible improvement: Path compression

Right now, we are duplicating some work with root-finding.

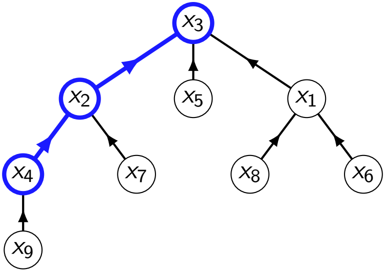
FindSet(x_4);



A possible improvement: Path compression

Right now, we are duplicating some work with root-finding.

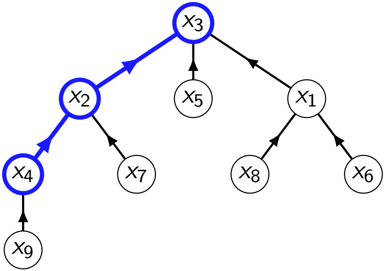
`FindSet`(x_4);



A possible improvement: Path compression

Right now, we are duplicating some work with root-finding.

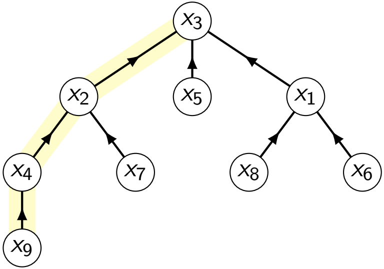
`FindSet`(x_4); Returns x_3.



A possible improvement: Path compression

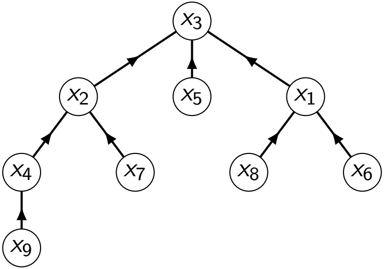
Right now, we are duplicating some work with root-finding.

We traverse these edges several times!



A possible improvement: Path compression

Right now, we are duplicating some work with root-finding.

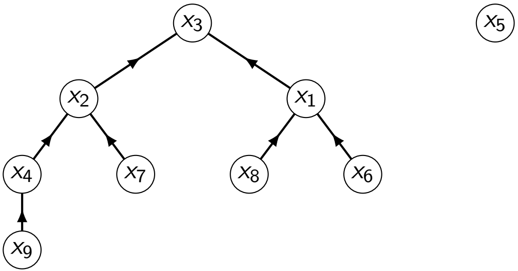


We could fix this by flattening our trees on each `Union` and `FindSet` operation, making every vertex we pass through a child of the root.

This technique is called **path compression**.

A possible improvement: Path compression

Right now, we are duplicating some work with root-finding.



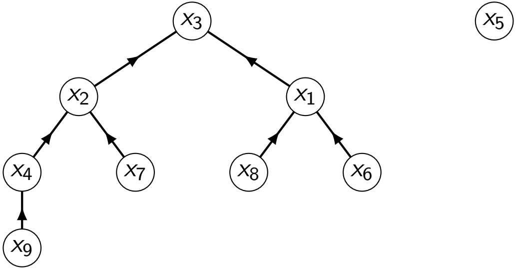
We could fix this by flattening our trees on each `Union` and `FindSet` operation, making every vertex we pass through a child of the root.

This technique is called **path compression**.

A possible improvement: Path compression

Right now, we are duplicating some work with root-finding.

FindSet(x_9);



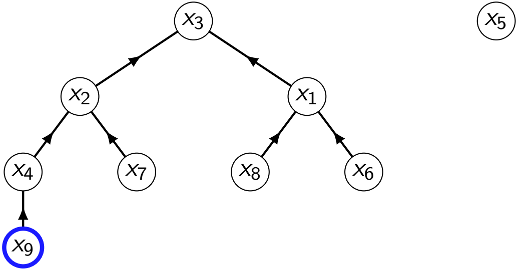
We could fix this by flattening our trees on each `Union` and `FindSet` operation, making every vertex we pass through a child of the root.

This technique is called **path compression**.

A possible improvement: Path compression

Right now, we are duplicating some work with root-finding.

`FindSet`(x_9);



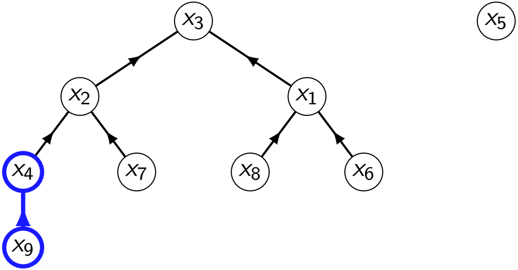
We could fix this by flattening our trees on each `Union` and `FindSet` operation, making every vertex we pass through a child of the root.

This technique is called **path compression**.

A possible improvement: Path compression

Right now, we are duplicating some work with root-finding.

`FindSet`(x_9);



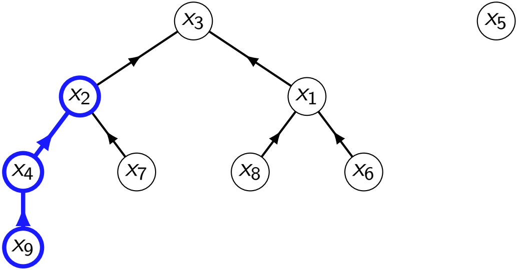
We could fix this by flattening our trees on each `Union` and `FindSet` operation, making every vertex we pass through a child of the root.

This technique is called **path compression**.

A possible improvement: Path compression

Right now, we are duplicating some work with root-finding.

`FindSet`(x_9);



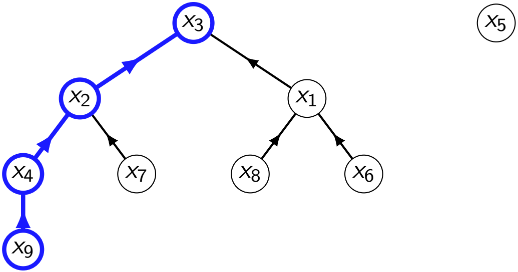
We could fix this by flattening our trees on each `Union` and `FindSet` operation, making every vertex we pass through a child of the root.

This technique is called **path compression**.

A possible improvement: Path compression

Right now, we are duplicating some work with root-finding.

`FindSet`(x_9);



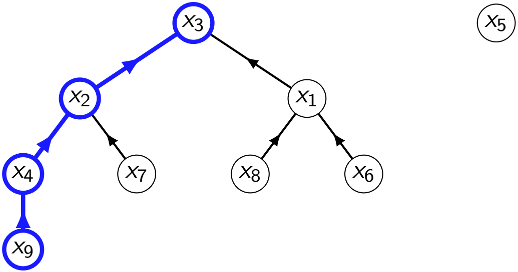
We could fix this by flattening our trees on each `Union` and `FindSet` operation, making every vertex we pass through a child of the root.

This technique is called **path compression**.

A possible improvement: Path compression

Right now, we are duplicating some work with root-finding.

FindSet(x_9); Returns x_3.



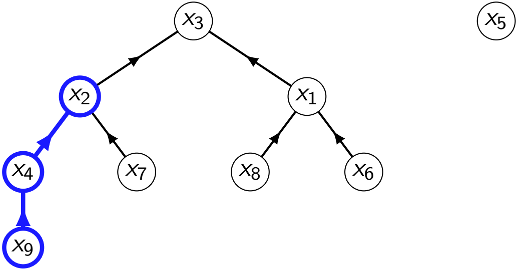
We could fix this by flattening our trees on each `Union` and `FindSet` operation, making every vertex we pass through a child of the root.

This technique is called **path compression**.

A possible improvement: Path compression

Right now, we are duplicating some work with root-finding.

`FindSet(`x_9`);` Returns x_3.



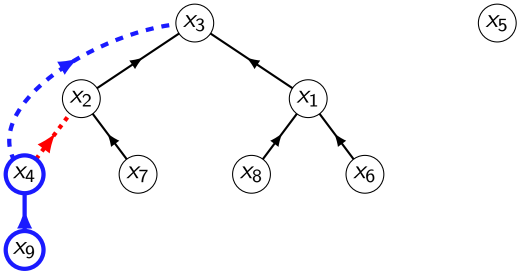
We could fix this by flattening our trees on each `Union` and `FindSet` operation, making every vertex we pass through a child of the root.

This technique is called **path compression**.

A possible improvement: Path compression

Right now, we are duplicating some work with root-finding.

`FindSet`(x_9); Returns x_3.



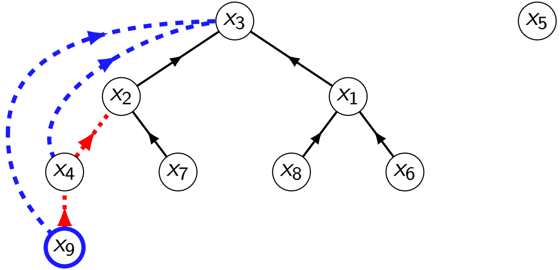
We could fix this by flattening our trees on each `Union` and `FindSet` operation, making every vertex we pass through a child of the root.

This technique is called **path compression**.

A possible improvement: Path compression

Right now, we are duplicating some work with root-finding.

FindSet(x_9);   Returns x_3.



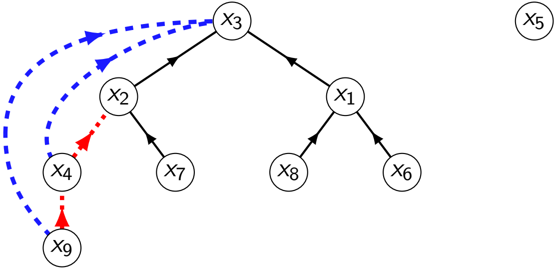
We could fix this by flattening our trees on each Union and FindSet operation, making every vertex we pass through a child of the root.

This technique is called **path compression**.

A possible improvement: Path compression

Right now, we are duplicating some work with root-finding.

FindSet(x_9);   Returns x_3.



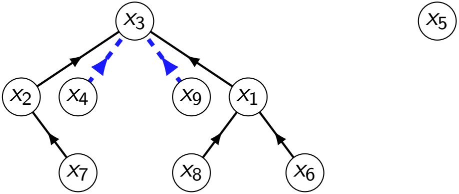
We could fix this by flattening our trees on each Union and FindSet operation, making every vertex we pass through a child of the root.

This technique is called **path compression**.

A possible improvement: Path compression

Right now, we are duplicating some work with root-finding.

`FindSet`(x_9); Returns x_3.



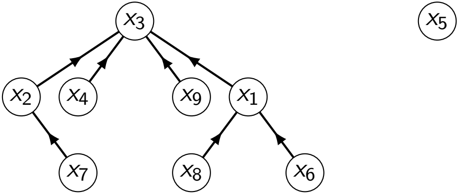
We could fix this by flattening our trees on each `Union` and `FindSet` operation, making every vertex we pass through a child of the root.

This technique is called **path compression**.

A possible improvement: Path compression

Right now, we are duplicating some work with root-finding.

$\texttt{Union}(x_9, x_5)$;



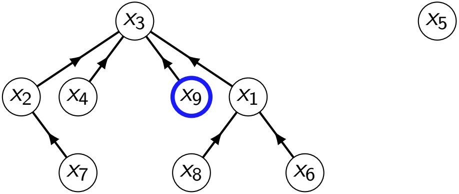
We could fix this by flattening our trees on each `Union` and `FindSet` operation, making every vertex we pass through a child of the root.

This technique is called **path compression**.

A possible improvement: Path compression

Right now, we are duplicating some work with root-finding.

`Union`(x_9, x_5);



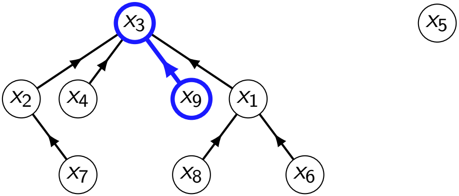
We could fix this by flattening our trees on each `Union` and `FindSet` operation, making every vertex we pass through a child of the root.

This technique is called **path compression**.

A possible improvement: Path compression

Right now, we are duplicating some work with root-finding.

`Union`(x_9, x_5);



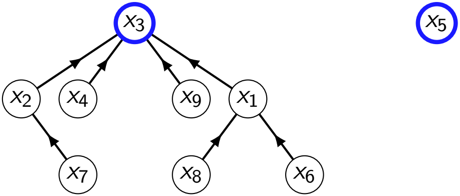
We could fix this by flattening our trees on each `Union` and `FindSet` operation, making every vertex we pass through a child of the root.

This technique is called **path compression**.

A possible improvement: Path compression

Right now, we are duplicating some work with root-finding.

$\texttt{Union}(x_9, x_5)$;



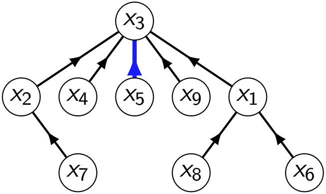
We could fix this by flattening our trees on each `Union` and `FindSet` operation, making every vertex we pass through a child of the root.

This technique is called **path compression**.

A possible improvement: Path compression

Right now, we are duplicating some work with root-finding.

`Union`(x_9, x_5);



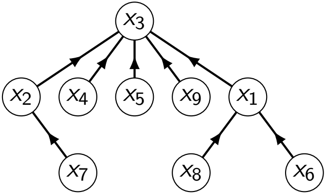
We could fix this by flattening our trees on each `Union` and `FindSet` operation, making every vertex we pass through a child of the root.

This technique is called **path compression**.

A possible improvement: Path compression

Right now, we are duplicating some work with root-finding.

`FindSet`(x_4);



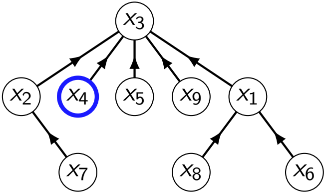
We could fix this by flattening our trees on each `Union` and `FindSet` operation, making every vertex we pass through a child of the root.

This technique is called **path compression**.

A possible improvement: Path compression

Right now, we are duplicating some work with root-finding.

`FindSet`(x_4);



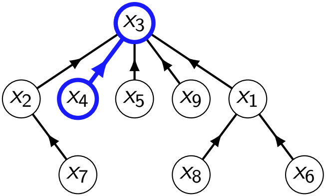
We could fix this by flattening our trees on each `Union` and `FindSet` operation, making every vertex we pass through a child of the root.

This technique is called **path compression**.

A possible improvement: Path compression

Right now, we are duplicating some work with root-finding.

`FindSet`(x_4);



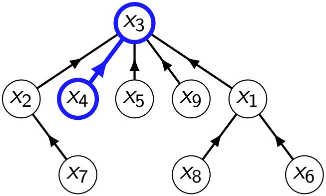
We could fix this by flattening our trees on each `Union` and `FindSet` operation, making every vertex we pass through a child of the root.

This technique is called **path compression**.

A possible improvement: Path compression

Right now, we are duplicating some work with root-finding.

`FindSet`(x_4); Returns x_3.



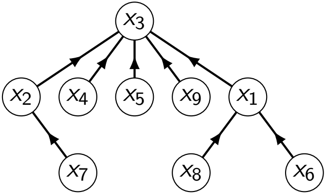
We could fix this by flattening our trees on each `Union` and `FindSet` operation, making every vertex we pass through a child of the root.

This technique is called **path compression**.

A possible improvement: Path compression

Right now, we are duplicating some work with root-finding.

`FindSet`(x_4); Returns x_3.



We could fix this by flattening our trees on each `Union` and `FindSet` operation, making every vertex we pass through a child of the root.

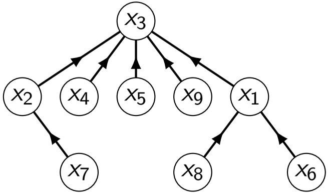
This technique is called **path compression**.

This improves the running time of n operations from $O(n \log n)$ to $O(n\alpha(n))$, where $\alpha(n)$ is the **inverse Ackermann function**: $\alpha(n) = \min\{k \colon A(k,k) \geq n\}$.

A possible improvement: Path compression

Right now, we are duplicating some work with root-finding.

`FindSet`(x_4);　　　　Returns x_3.



We could fix this by flattening our trees on each `Union` and `FindSet` operation, making every vertex we pass through a child of the root.

This technique is called **path compression**.

This improves the running time of n operations from $O(n \log n)$ to $O(n\alpha(n))$, where $\alpha(n)$ is the **inverse Ackermann function**: $\alpha(n) = \min\{k : A(k,k) \geq n\}$. In practice, we **always** have $\alpha(n) \leq 4$.