# Lecture 9: Disjoint Sets / Union-Find

Michael Dinitz

September 25, 2025

601.433/633 Introduction to Algorithms

# Introduction

Informal: Universe of elements, want to maintain *disjoint sets*.

Slightly more formally:

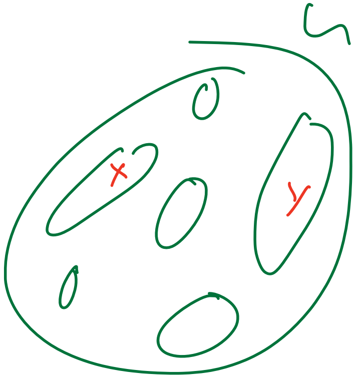
- Make-Set($x$): create a new set containing just $x$ (i.e., $\{x\}$)
- Union($x, y$): Replace set containing $x$ ($S$) and set containing $y$ ($T$) with single set $S \cup T$
- Find($x$): Return *representative* of set containing $x$

# Introduction

Informal: Universe of elements, want to maintain *disjoint sets*.

Slightly more formally:

- Make-Set($x$): create a new set containing just $x$ (i.e., $\{x\}$)
- Union($x, y$): Replace set containing $x$ ($S$) and set containing $y$ ($T$) with single set $S \cup T$
- Find($x$): Return *representative* of set containing $x$




Rules: every set has a *unique* representative.

- If $x$ and $y$ are in same set, Find($x$) = Find($y$)
- If $x$ and $y$ are in different sets, then Find($x$) $\neq$ Find($y$)
- Make-Set($x$): cannot be called on the same $x$ twice

# Introduction

Informal: Universe of elements, want to maintain *disjoint sets*.

Slightly more formally:

- Make-Set($x$): create a new set containing just $x$ (i.e., $\{x\}$)
- Union($x, y$): Replace set containing $x$ ($S$) and set containing $y$ ($T$) with single set $S \cup T$
- Find($x$): Return *representative* of set containing $x$

Rules: every set has a *unique* representative.

- If $x$ and $y$ are in same set, Find($x$) = Find($y$)
- If $x$ and $y$ are in different sets, then Find($x$) $\neq$ Find($y$)
- Make-Set($x$): cannot be called on the same $x$ twice

Note: disjoint (and partition) by construction!

# Introduction (II)

We'll see a few ways of doing this, from efficient to very efficient.
CLRS: extremely efficient

# Introduction (II)

We'll see a few ways of doing this, from efficient to very efficient.
CLRS: extremely efficient

Nice thing about Union-Find: don't hit a limit to improvement for a very long time!

# Introduction (II)

We'll see a few ways of doing this, from efficient to very efficient.
CLRS: extremely efficient

Nice thing about Union-Find: don't hit a limit to improvement for a very long time!

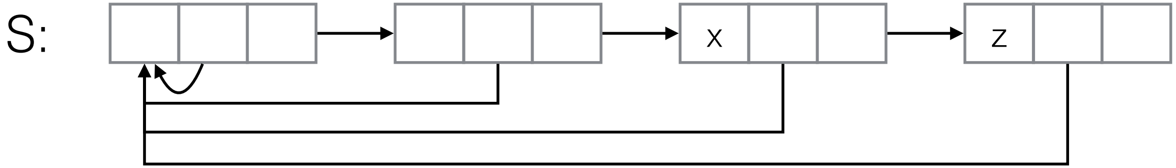
Notation and Notes:

- $m$ operations total
- $n$ of which are Make-Sets (so $n$ elements)
- Assume have pointer/access to elements we care about (like last class)

# First Approach: Lists

Linked list for each set.

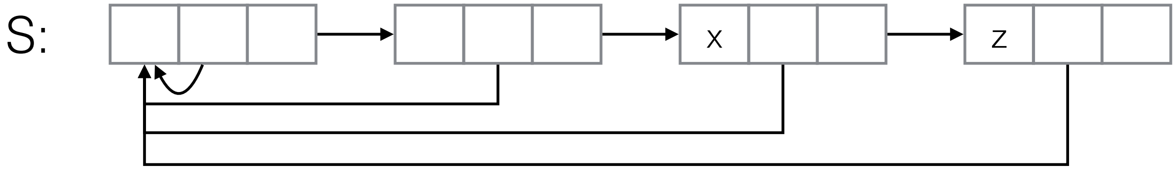
- Representative of set is head (first element on list)
- Each element has pointer to head and to next element, so stored as triple: (element, head, next)



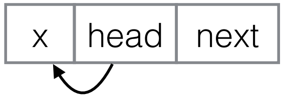
# First Approach: Lists

Linked list for each set.

- Representative of set is head (first element on list)
- Each element has pointer to head and to next element, so stored as triple: (element, head, next)



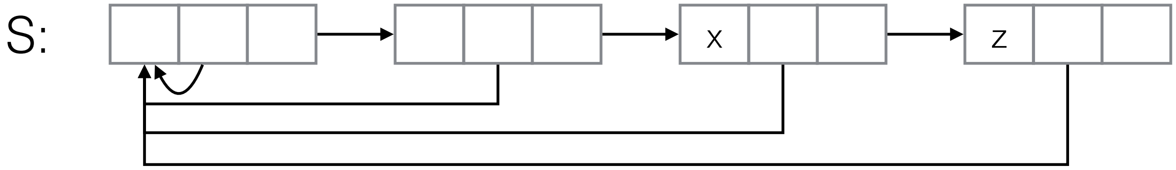

Make-Set($x$):



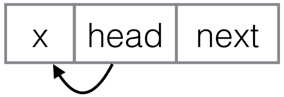
# First Approach: Lists

Linked list for each set.

- Representative of set is head (first element on list)
- Each element has pointer to head and to next element, so stored as triple: (element, head, next)

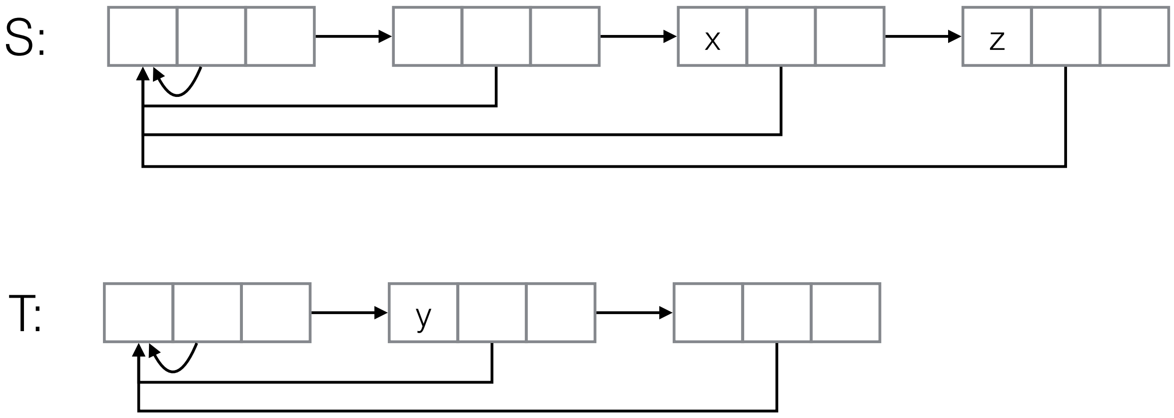



Make-Set($\boldsymbol{x}$):

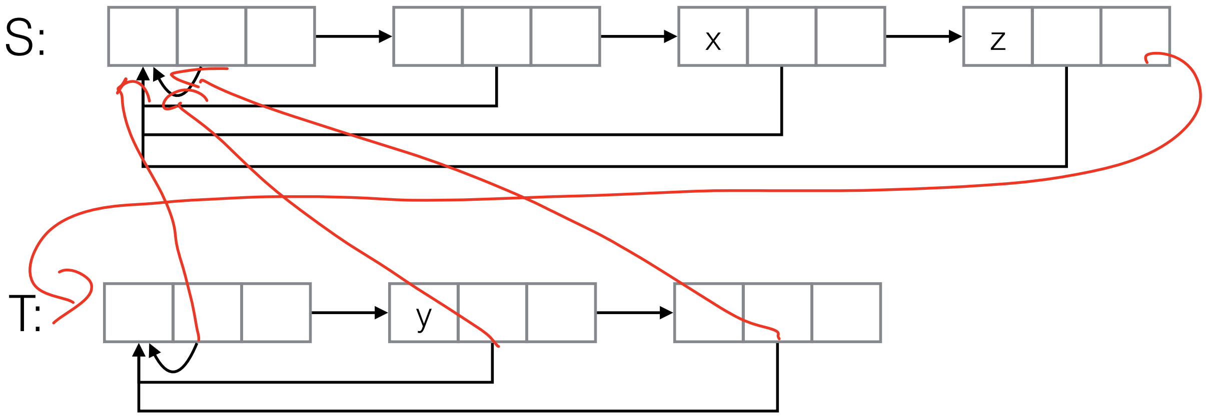



Find($\boldsymbol{x}$): return $\boldsymbol{x} \rightarrow$ head

# Union($x, y$)



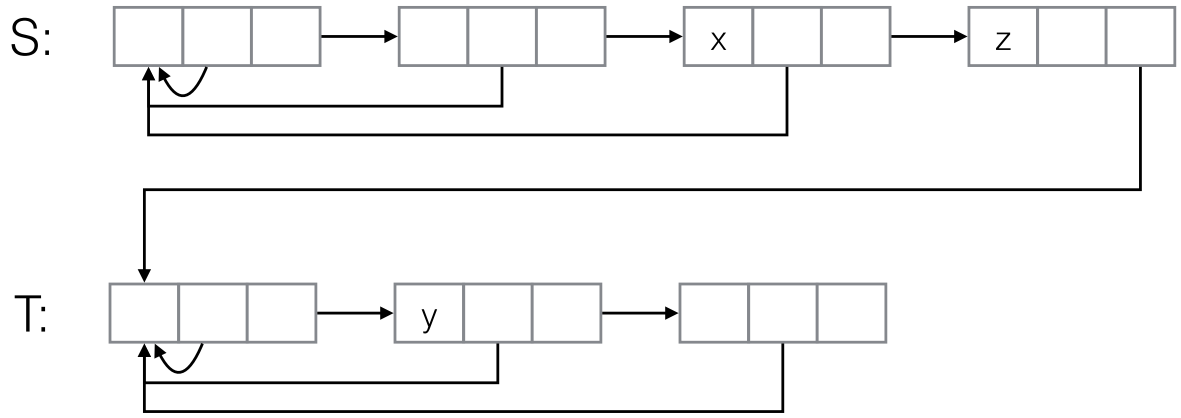
# Union($\boldsymbol{x}, \boldsymbol{y}$)



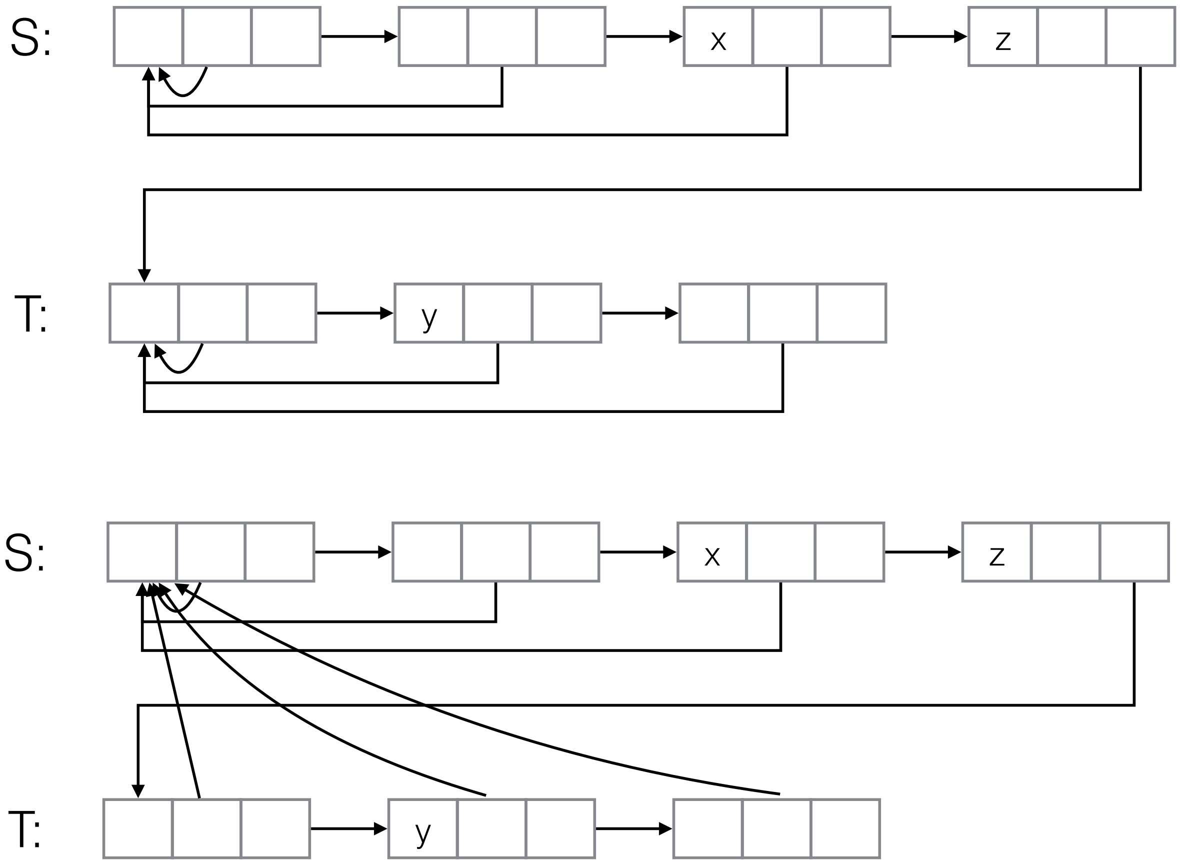

Obvious approach:

- Walk down $\boldsymbol{S}$ to final element $\boldsymbol{z}$ (starting from $\boldsymbol{x}$)
- Set $\boldsymbol{z} \rightarrow \text{next} = \boldsymbol{y} \rightarrow \text{head}$
- Walk down $\boldsymbol{T}$, set every elements head pointer to $\boldsymbol{x} \rightarrow \text{head}$

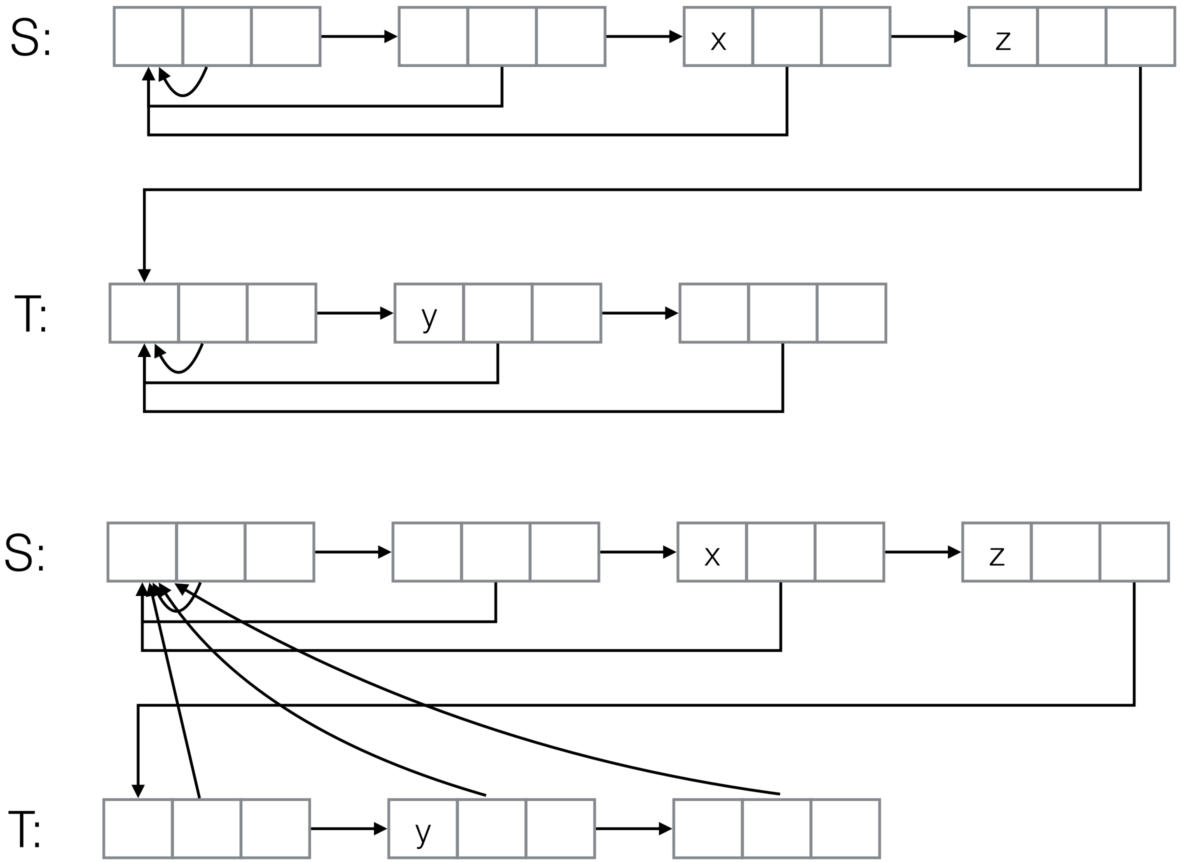
# Union($x, y$)



# Union($\boldsymbol{x}, \boldsymbol{y}$)

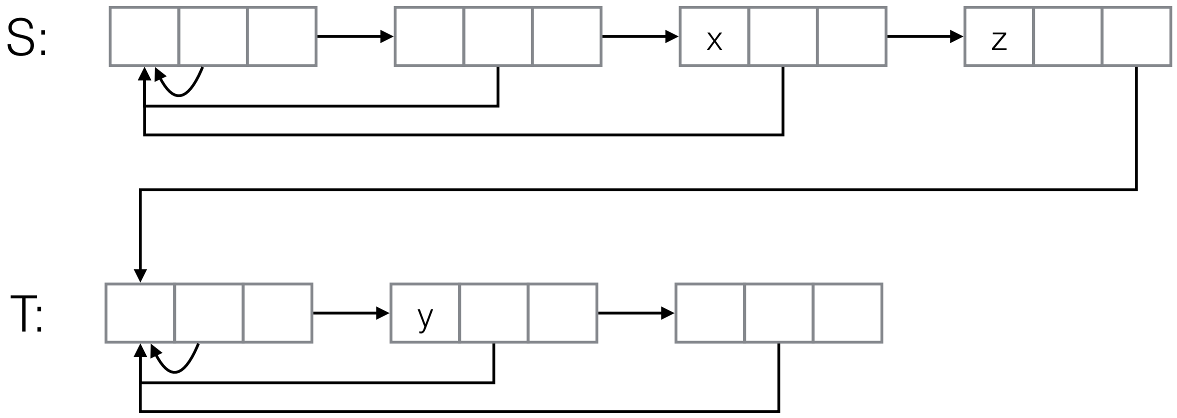


# Union($x, y$)

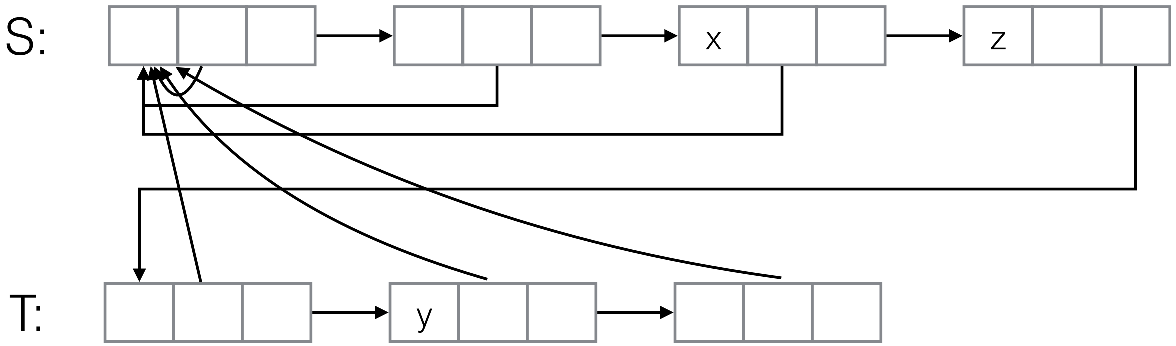



Running time:

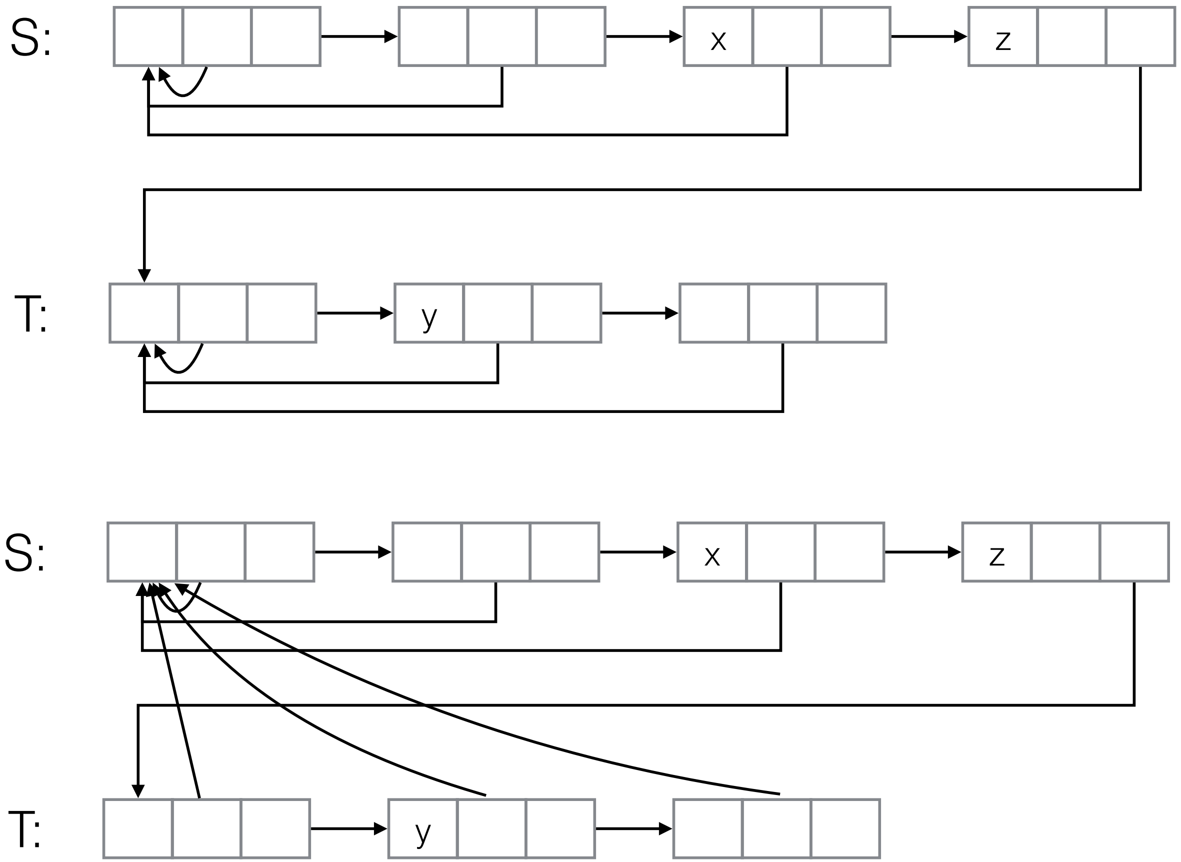
# Union($x, y$)




Running time: $O(|S| + |T|)$



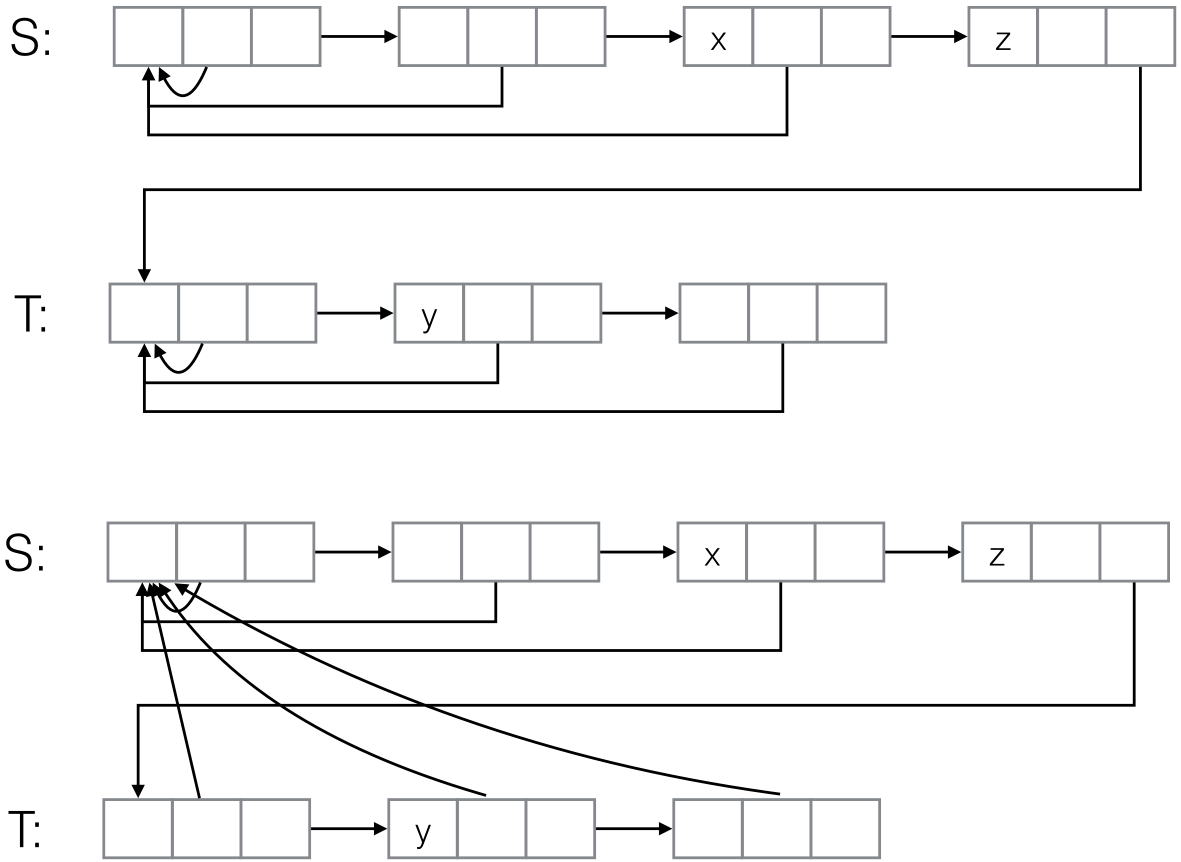
# Union($x, y$)




Running time: $O(|S| + |T|)$

- $|S|$ to walk down $S$ to final element
- $|T|$ to walk down $T$ resetting head pointers

# Union($x, y$)




Running time: $O(|S| + |T|)$

- $|S|$ to walk down $S$ to final element
- $|T|$ to walk down $T$ resetting head pointers

Since $|S|, |T|$ could be $\Theta(n)$, can only say $O(n)$ for Unions

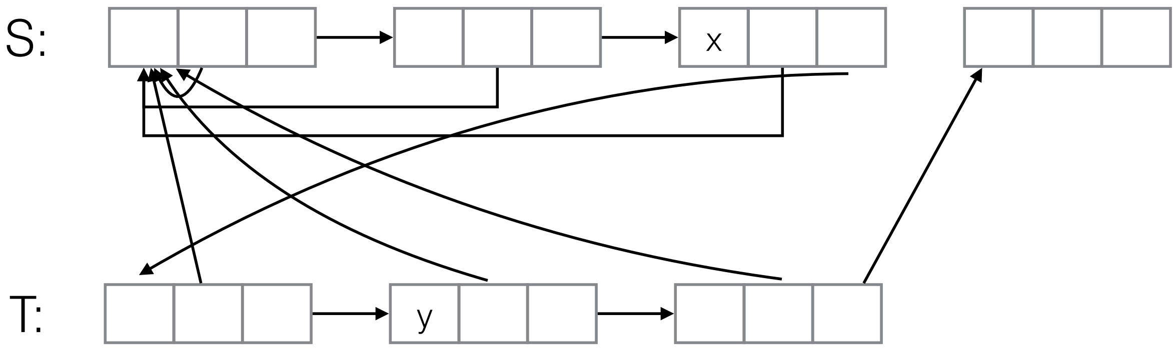
# Improved Union($x, y$)

Observation: don't need to preserve ordering inside the Union!

# Improved Union($\boldsymbol{x}, \boldsymbol{y}$)

Observation: don't need to preserve ordering inside the Union!

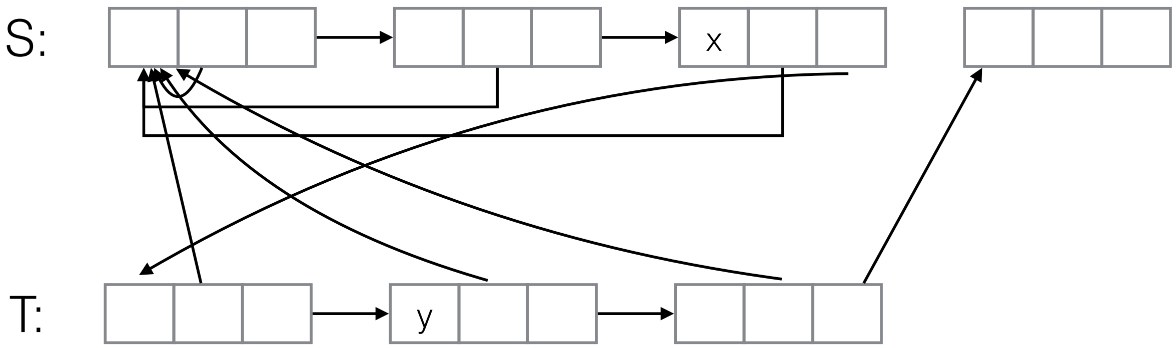
- Splice $\boldsymbol{T}$ into $\boldsymbol{S}$ right after $\boldsymbol{x}$



# Improved Union($x, y$)

Observation: don't need to preserve ordering inside the Union!

- Splice $T$ into $S$ right after $x$

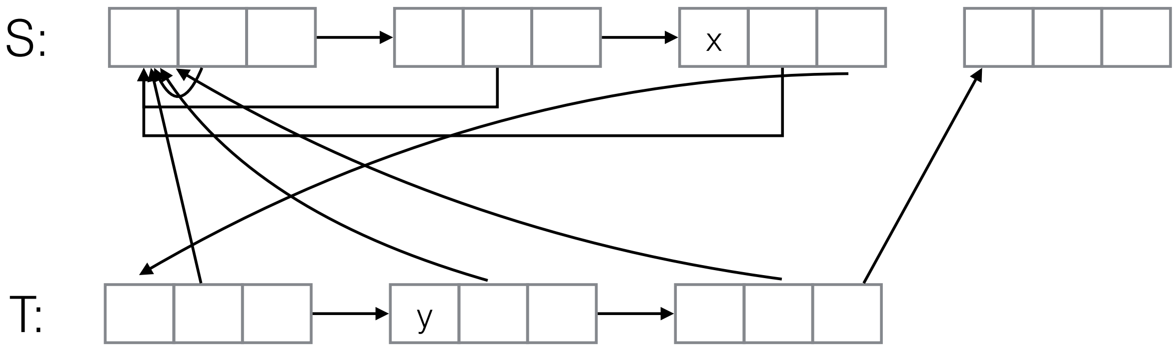



Running time:

# Improved Union($x, y$)

Observation: don't need to preserve ordering inside the Union!

- Splice $T$ into $S$ right after $x$

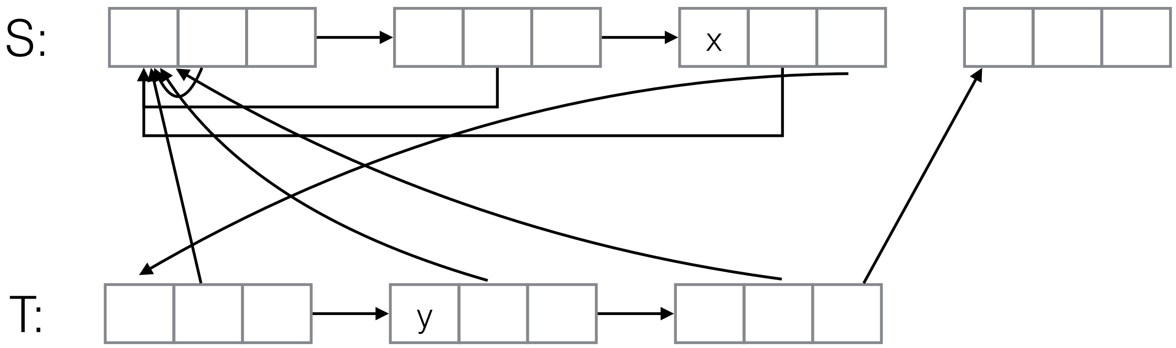



Running time: $O(|T|)$

# Improved Union($x, y$)

Observation: don't need to preserve ordering inside the Union!

- Splice $T$ into $S$ right after $x$




Running time: $O(|T|)$

- Still can't say anything better than $O(n)$

# Even more improved Union($\boldsymbol{x}, \boldsymbol{y}$)

Observation: Why splice $\boldsymbol{T}$ into $\boldsymbol{S}$? Could also splice $\boldsymbol{S}$ into $\boldsymbol{T}$.

- Time $\boldsymbol{O}(|\boldsymbol{S}|)$

# Even more improved Union($x, y$)

Observation: Why splice $T$ into $S$? Could also splice $S$ into $T$.

- Time $O(|S|)$

Splice smaller into bigger!

- Store size of set in head node.
- Splice smaller into bigger: time $O(\min(|S|, |T|))$
- *Still* only $O(n)$. But now can make stronger amortized guarantee!

# Even more improved Union($x, y$)

Observation: Why splice $T$ into $S$? Could also splice $S$ into $T$.

- Time $O(|S|)$

Splice smaller into bigger!

- Store size of set in head node.
- Splice smaller into bigger: time $O(\min(|S|, |T|))$
- *Still* only $O(n)$. But now can make stronger amortized guarantee!

**Theorem**

*The amortized cost of Find and Union is $O(1)$, and the amortized cost of Make-Set is $O(\log n)$.*

**Corollary**

*The total running time is $O(m + n \log n)$.*

# Amortized Analysis of List Algorithm

Banking/accounting argument: bank for every element

- When an element is created (via Make-Set), add $\log n$ tokens to its bank
- Find does not affect banks
- When doing Union, take token from bank of each element in smaller set.

# Amortized Analysis of List Algorithm

Banking/accounting argument: bank for every element

- When an element is created (via Make-Set), add $\log n$ tokens to its bank
- Find does not affect banks
- When doing Union, take token from bank of each element in smaller set.

Obvious: initially, total bank is $0$ (no elements).

# Amortized Analysis of List Algorithm

Banking/accounting argument: bank for every element

- When an element is created (via Make-Set), add $\log n$ tokens to its bank
- Find does not affect banks
- When doing Union, take token from bank of each element in smaller set.

Obvious: initially, total bank is $0$ (no elements).

**Lemma**

*No bank is ever negative.*

# Amortized Analysis of List Algorithm

Banking/accounting argument: bank for every element

- When an element is created (via Make-Set), add $\log n$ tokens to its bank
- Find does not affect banks
- When doing Union, take token from bank of each element in smaller set.

Obvious: initially, total bank is $\mathbf{0}$ (no elements).

## Lemma

*No bank is ever negative.*

## Proof.

Fix element $e$. Starts with $\log n$ tokens. When do we remove a token?

# Amortized Analysis of List Algorithm

Banking/accounting argument: bank for every element

- When an element is created (via Make-Set), add $\log n$ tokens to its bank
- Find does not affect banks
- When doing Union, take token from bank of each element in smaller set.

Obvious: initially, total bank is $0$ (no elements).

**Lemma**

*No bank is ever negative.*

**Proof.**

Fix element $e$. Starts with $\log n$ tokens. When do we remove a token?

- When in smaller set of a Union.

# Amortized Analysis of List Algorithm

Banking/accounting argument: bank for every element

- When an element is created (via Make-Set), add $\log n$ tokens to its bank
- Find does not affect banks
- When doing Union, take token from bank of each element in smaller set.

Obvious: initially, total bank is $0$ (no elements).

**Lemma**

*No bank is ever negative.*

**Proof.**

Fix element $e$. Starts with $\log n$ tokens. When do we remove a token?

- When in smaller set of a Union.
- Size of set containing $e$ at least doubles!

# Amortized Analysis of List Algorithm

Banking/accounting argument: bank for every element

- When an element is created (via Make-Set), add $\log n$ tokens to its bank
- Find does not affect banks
- When doing Union, take token from bank of each element in smaller set.

Obvious: initially, total bank is $0$ (no elements).

**Lemma**

*No bank is ever negative.*

**Proof.**

Fix element $e$. Starts with $\log n$ tokens. When do we remove a token?

- When in smaller set of a Union.
- Size of set containing $e$ at least doubles!
- Can only happen at most $\log n$ times. $\square$

# Amortized Analysis of List Algorithm (cont'd)

Make-Set:

- True cost: $O(1)$
- Change in banks: $\log n$

$\implies$ Amortized cost: $O(1) + O(\log n) = O(\log n)$

# Amortized Analysis of List Algorithm (cont'd)

Make-Set:

- True cost: $O(1)$
- Change in banks: $\log n$

$\Longrightarrow$ Amortized cost: $O(1) + O(\log n) = O(\log n)$

Find:

- True cost: $O(1)$
- Change in banks: $0$

$\Longrightarrow$ Amortized cost: $O(1) + 0 = O(1)$

# Amortized Analysis of List Algorithm (cont'd)

Make-Set:

- True cost: $O(1)$
- Change in banks: $\log n$

$\Longrightarrow$ Amortized cost: $O(1) + O(\log n) = O(\log n)$

Find:

- True cost: $O(1)$
- Change in banks: $0$

$\Longrightarrow$ Amortized cost: $O(1) + 0 = O(1)$

Union:

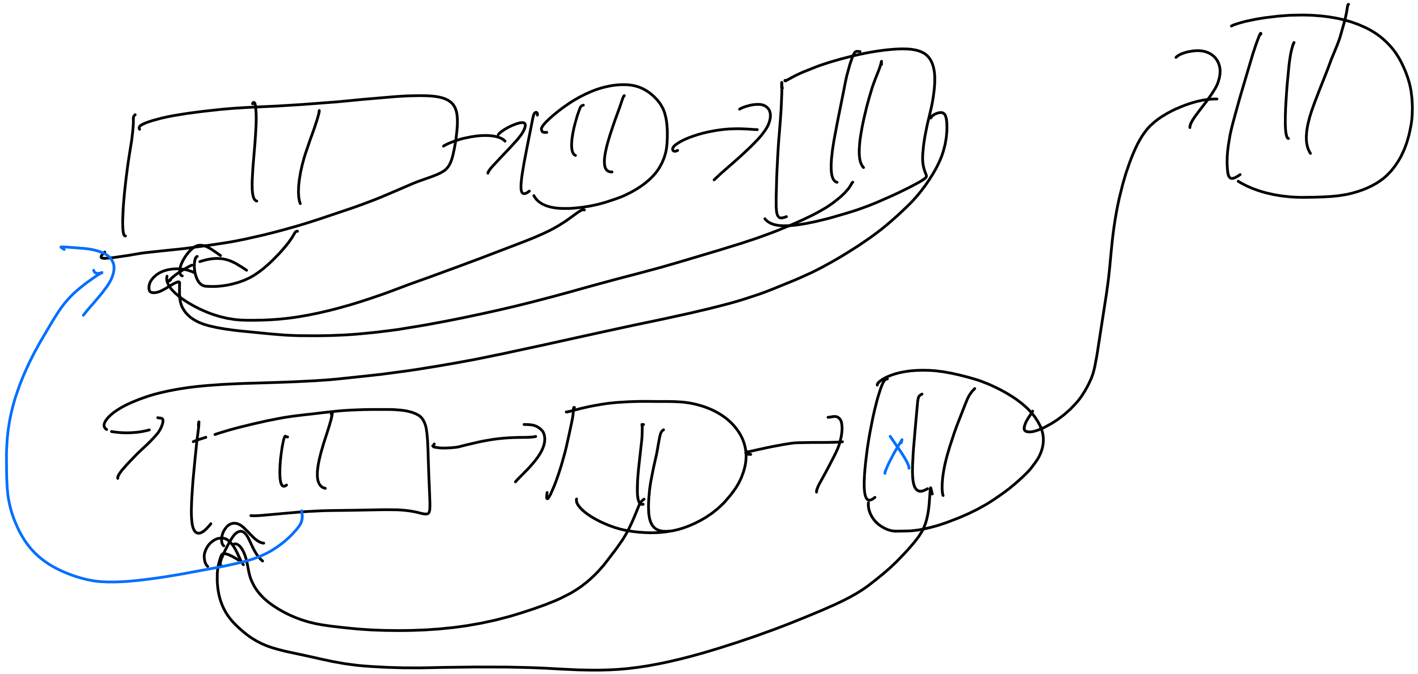
- True cost: $\min(|S|, |T|)$
- Change in banks: $-\min(|S|, |T|)$

$\Longrightarrow$ Amortized cost: $\min(|S|, |T|) - \min(|S|, |T|) = 0 = O(1)$.

# Even Better

Starting idea: want to make Unions faster, willing to make Finds a little slower.

- Slow part of Union: updating all head pointers in smaller list.
- Don't do it!



# Even Better

Starting idea: want to make Unions faster, willing to make Finds a little slower.

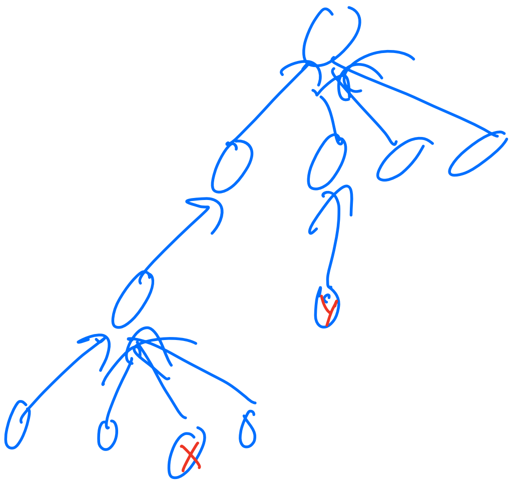
- Slow part of Union: updating all head pointers in smaller list.
- Don't do it!
- Results in trees rather than lists (can drop next pointer)

# Even Better

Starting idea: want to make Unions faster, willing to make Finds a little slower.

- Slow part of Union: updating all head pointers in smaller list.
- Don't do it!
- Results in trees rather than lists (can drop next pointer)

Finds slow: need to walk up tree




# Even Better

Starting idea: want to make Unions faster, willing to make Finds a little slower.

- Slow part of Union: updating all head pointers in smaller list.
- Don't do it!
- Results in trees rather than lists (can drop next pointer)

Finds slow: need to walk up tree

- Use *this* time to "update head" pointers: on Find($\boldsymbol{x}$), change pointers of $\boldsymbol{x}$ and all ancestors to point to root
- *Path Compression*

# Even Better

Starting idea: want to make Unions faster, willing to make Finds a little slower.

- Slow part of Union: updating all head pointers in smaller list.
- Don't do it!
- Results in trees rather than lists (can drop next pointer)

Finds slow: need to walk up tree

- Use *this* time to "update head" pointers: on Find($\boldsymbol{x}$), change pointers of $\boldsymbol{x}$ and all ancestors to point to root
- *Path Compression*

Idea 2: *Union By Rank*

- Size of set was important for lists, less important for trees.
- Choose which set to splice into which by *rank* of trees (related to height)

# Main Result

**Theorem**

*When using Path Compression and Union By Rank, total time at most* $O(m \log^* n)$.

$\log^*$: iterated $\log_2$.

- $\log^* n = \#$ times apply $\log_2$ until get to $1$

# Main Result

**Theorem**

*When using Path Compression and Union By Rank, total time at most $O(m \log^* n)$.*

$\log^*$: iterated $\log_2$.

- $\log^* n$ = # times apply $\log_2$ until get to $1$
- $\log^*(2^{65536}) = 1 + \log^*(65536) = 2 + \log^*(16) = 3 + \log^*(4) = 4 + \log^*(2) = 5$

# Main Result

**Theorem**

*When using Path Compression and Union By Rank, total time at most* $O(m \log^* n)$.

$\log^*$: iterated $\log_2$.

- $\log^* n = \#$ times apply $\log_2$ until get to $1$
- $\log^*(2^{65536}) = 1 + \log^*(65536) = 2 + \log^*(16) = 3 + \log^*(4) = 4 + \log^*(2) = 5$
- Basically $\log^* n$ always $\leq 5$.

# Main Result

**Theorem**

*When using Path Compression and Union By Rank, total time at most $O(m \log^* n)$.*

$\log^*$: iterated $\log_2$.

- $\log^* n = \#$ times apply $\log_2$ until get to $1$
- $\log^*(2^{65536}) = 1 + \log^*(65536) = 2 + \log^*(16) = 3 + \log^*(4) = 4 + \log^*(2) = 5$
- Basically $\log^* n$ always $\leq 5$.

Stronger theorem: total time at most $O(m \cdot \alpha(m, n))$.

- $\alpha(m, n)$: inverse Ackermann function. Grows even slower than $\log^*$.
- See CLRS for details

# Formal Procedures: Make-Set and Find

Make-Set($\boldsymbol{x}$): Set $\boldsymbol{x} \rightarrow \boldsymbol{rank} = \mathbf{0}$ and $\boldsymbol{x} \rightarrow \boldsymbol{parent} = \boldsymbol{x}$

- Running time: $\boldsymbol{O(1)}$.

# Formal Procedures: Make-Set and Find

Make-Set($x$): Set $x \to \boldsymbol{rank} = \mathbf{0}$ and $x \to \boldsymbol{parent} = x$

- Running time: $\boldsymbol{O(1)}$.

Find($x$): Walk from $x$ to root, and return root. Set parent pointers of $x$ and all ancestors to root.

- If $x \to \boldsymbol{parent} = x$ then return $x$
- $x \to \boldsymbol{parent} = \boldsymbol{Find}(x \to \boldsymbol{parent})$
- Return $x \to \boldsymbol{parent}$

# Formal Procedures: Make-Set and Find

Make-Set($x$): Set $x \to \boldsymbol{rank} = \mathbf{0}$ and $x \to \boldsymbol{parent} = x$

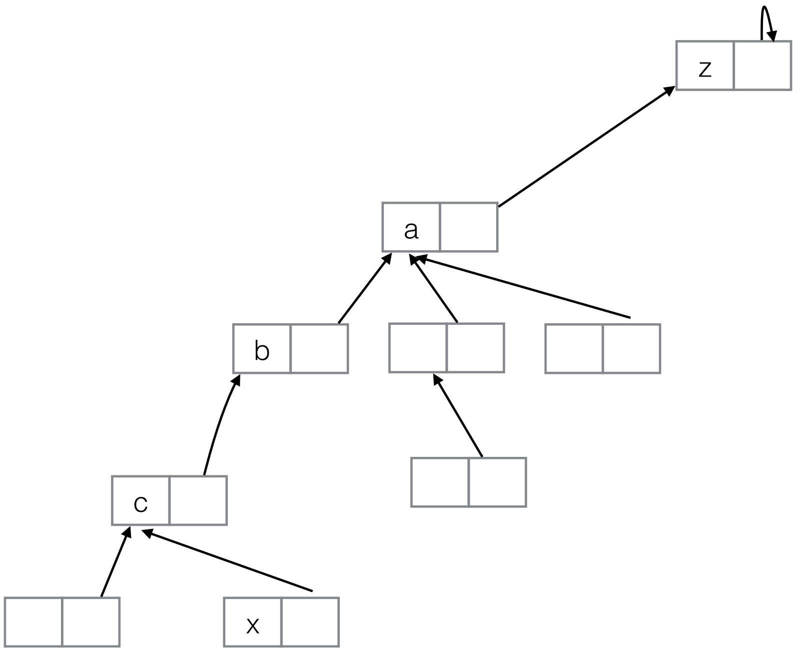
- Running time: $O(1)$.

Find($x$): Walk from $x$ to root, and return root. Set parent pointers of $x$ and all ancestors to root.

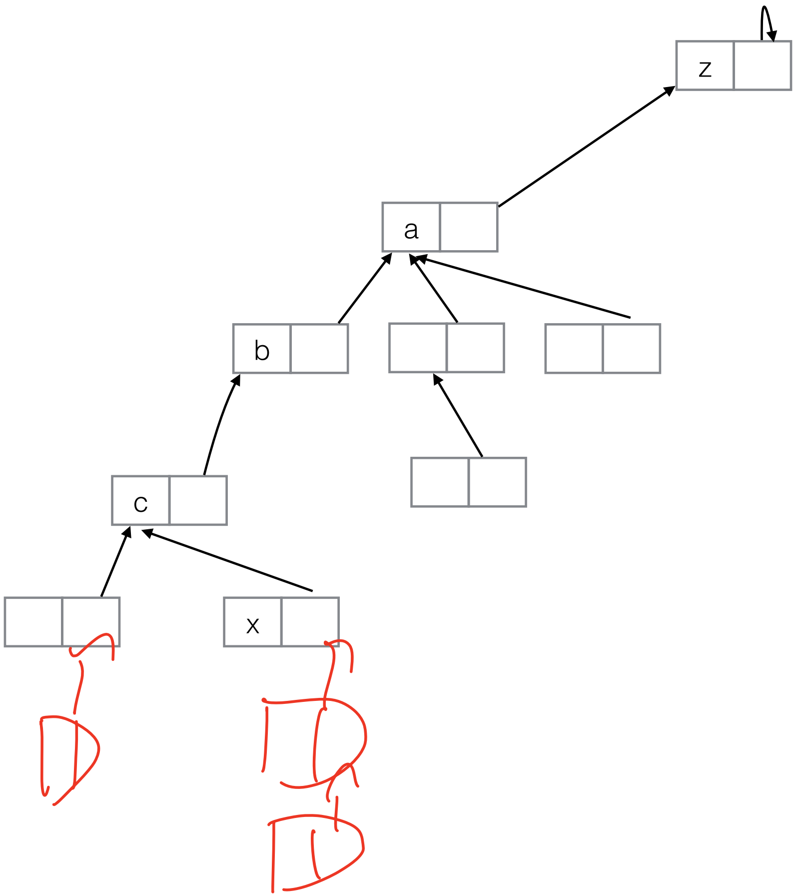
- If $x \to \boldsymbol{parent} = x$ then return $x$
- $x \to \boldsymbol{parent} = \boldsymbol{Find}(x \to \boldsymbol{parent})$
- Return $x \to \boldsymbol{parent}$

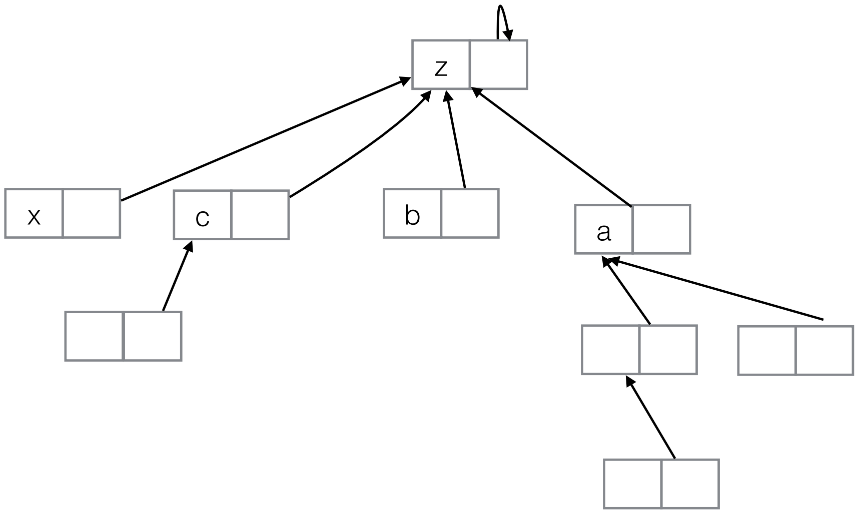
Running time of Find: depth of $x$ (distance to root)

# Find example



# Find example

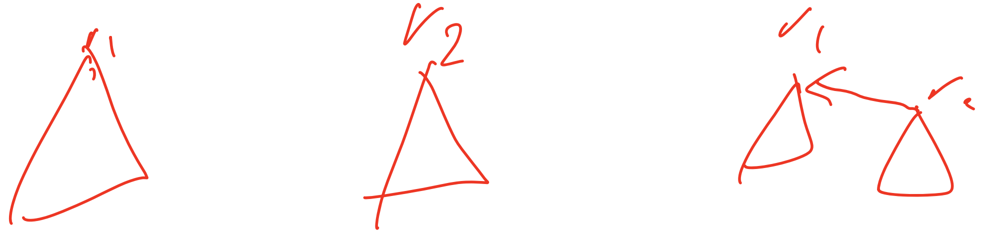


# Formal Procedure: Union

Link($\boldsymbol{r_1}, \boldsymbol{r_2}$): Only applied to root nodes

- If $\boldsymbol{r_1} \to \boldsymbol{rank} > \boldsymbol{r_2} \to \boldsymbol{rank}$, set $\boldsymbol{r_2} \to \boldsymbol{parent} = \boldsymbol{r_1}$
- If $\boldsymbol{r_2} \to \boldsymbol{rank} > \boldsymbol{r_1} \to \boldsymbol{rank}$, set $\boldsymbol{r_1} \to \boldsymbol{parent} = \boldsymbol{r_2}$
- If $\boldsymbol{r_1} \to \boldsymbol{rank} = \boldsymbol{r_2} \to \boldsymbol{rank}$, set $\boldsymbol{r_2} \to \boldsymbol{parent} = \boldsymbol{r_1}$ and increment $\boldsymbol{r_1} \to \boldsymbol{rank}$.



# Formal Procedure: Union

Link($r_1, r_2$): Only applied to root nodes

- If $r_1 \to \textit{\textbf{rank}} > r_2 \to \textit{\textbf{rank}}$, set $r_2 \to \textit{\textbf{parent}} = r_1$
- If $r_2 \to \textit{\textbf{rank}} > r_1 \to \textit{\textbf{rank}}$, set $r_1 \to \textit{\textbf{parent}} = r_2$
- If $r_1 \to \textit{\textbf{rank}} = r_2 \to \textit{\textbf{rank}}$, set $r_2 \to \textit{\textbf{parent}} = r_1$ and increment $r_1 \to \textit{\textbf{rank}}$.

Running time of Link: $O(1)$

# Formal Procedure: Union

Link($\boldsymbol{r_1}, \boldsymbol{r_2}$): Only applied to root nodes

- If $\boldsymbol{r_1 \to rank} > \boldsymbol{r_2 \to rank}$, set $\boldsymbol{r_2 \to parent} = \boldsymbol{r_1}$
- If $\boldsymbol{r_2 \to rank} > \boldsymbol{r_1 \to rank}$, set $\boldsymbol{r_1 \to parent} = \boldsymbol{r_2}$
- If $\boldsymbol{r_1 \to rank} = \boldsymbol{r_2 \to rank}$, set $\boldsymbol{r_2 \to parent} = \boldsymbol{r_1}$ and increment $\boldsymbol{r_1 \to rank}$.

Running time of Link: $\boldsymbol{O(1)}$

Union($\boldsymbol{x}, \boldsymbol{y}$): Link(Find($\boldsymbol{x}$), Find($\boldsymbol{y}$))

# Formal Procedure: Union

Link($r_1, r_2$): Only applied to root nodes

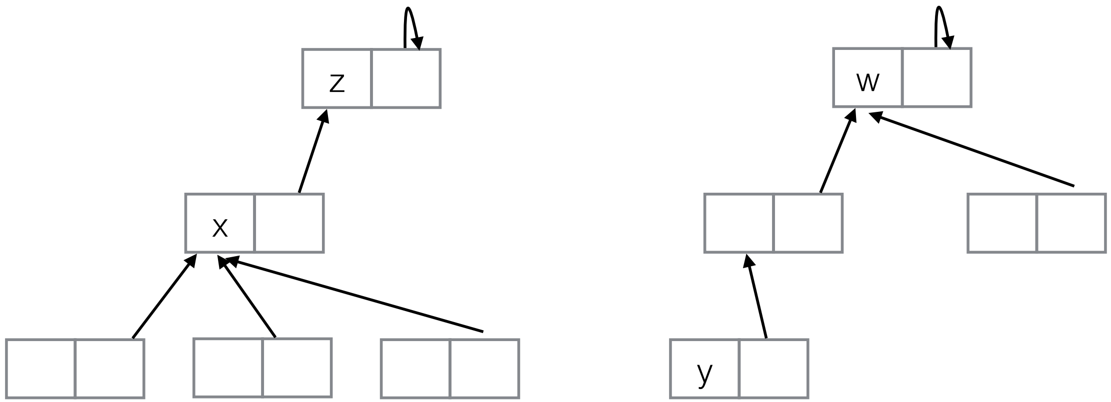
- If $r_1 \to rank > r_2 \to rank$, set $r_2 \to parent = r_1$
- If $r_2 \to rank > r_1 \to rank$, set $r_1 \to parent = r_2$
- If $r_1 \to rank = r_2 \to rank$, set $r_2 \to parent = r_1$ and increment $r_1 \to rank$.

Running time of Link: $O(1)$

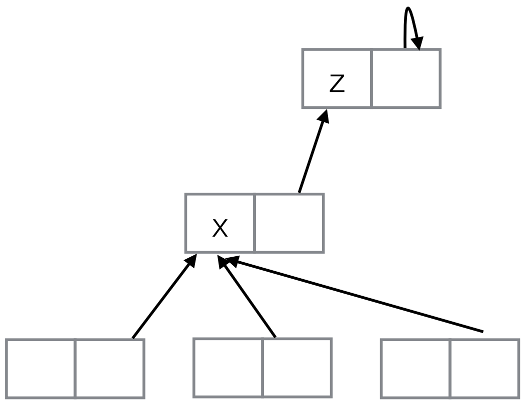
Union($x, y$): Link(Find($x$), Find($y$))

- Running time: depth($x$) + depth($y$)

# Union example

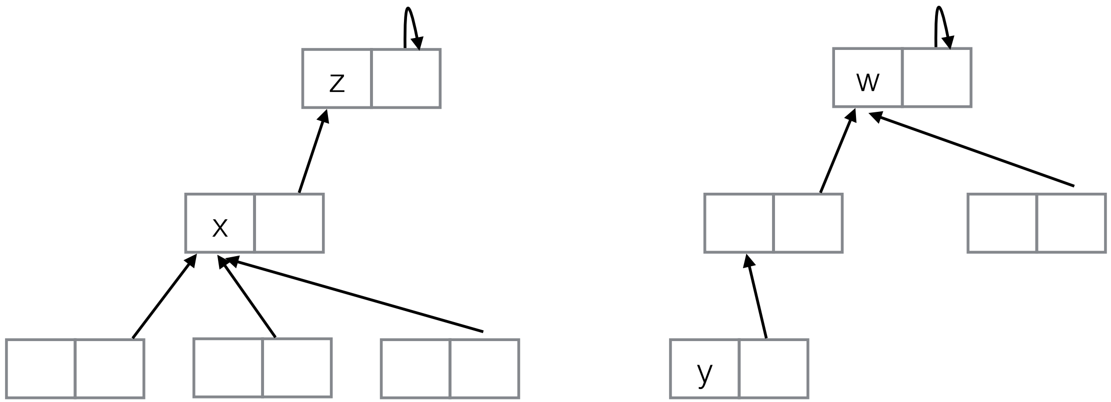


# Union example

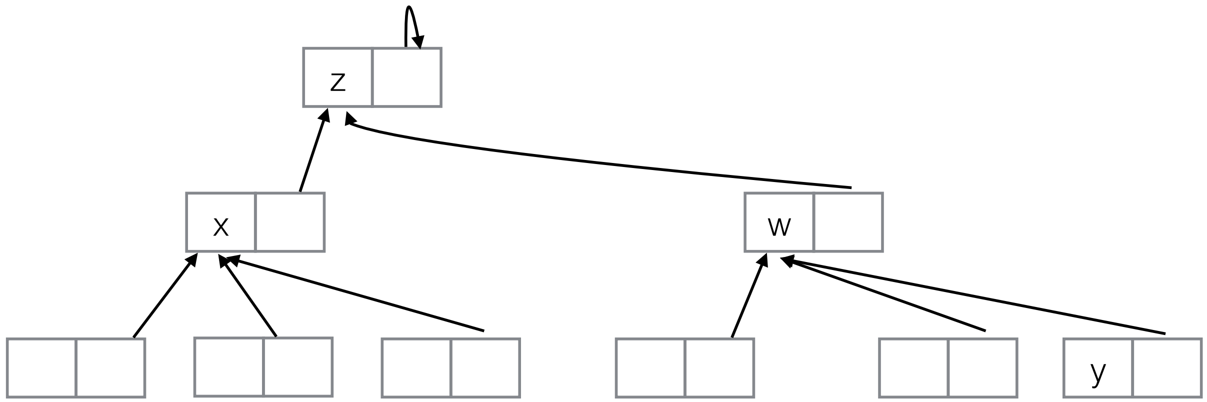



If $z \rightarrow \textbf{\textit{rank}} \geq w \rightarrow \textbf{\textit{rank}}$

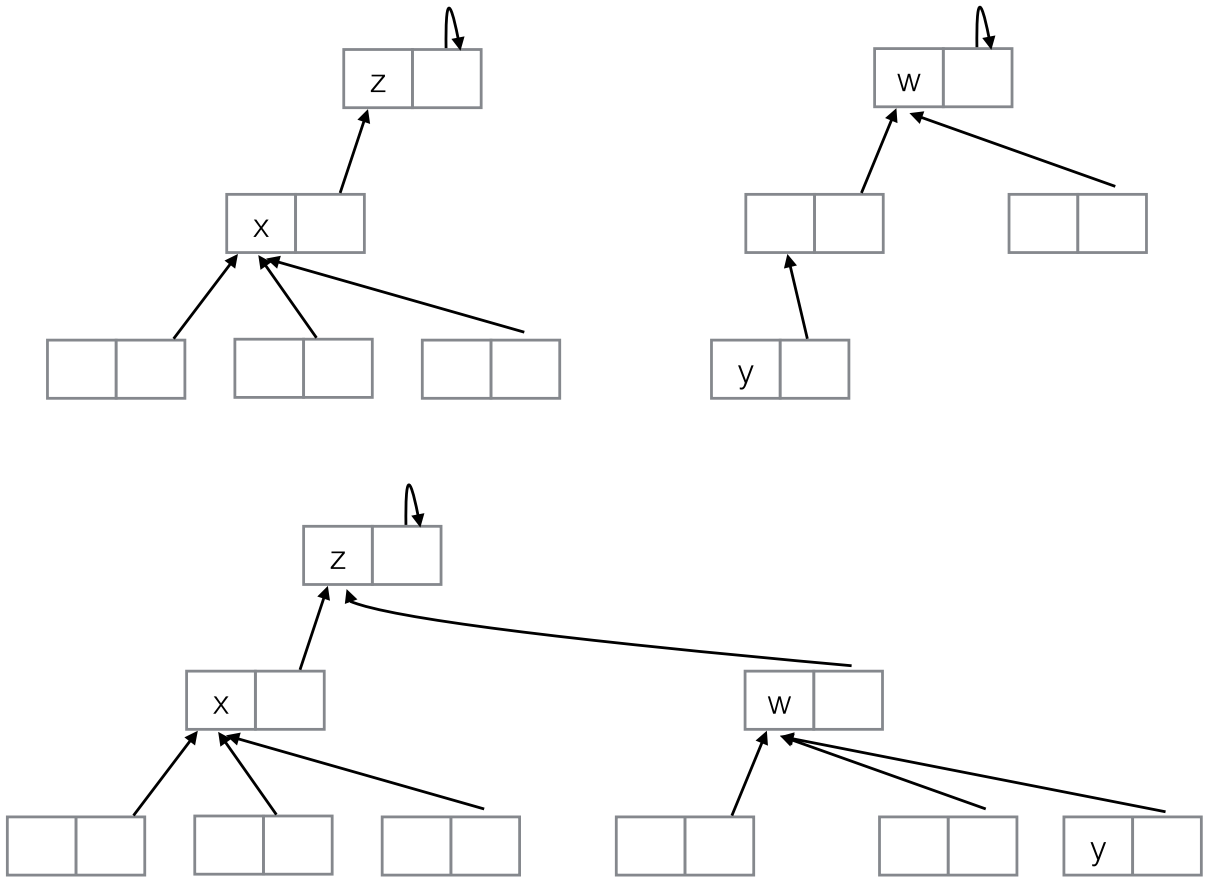
# Union example




If $\boldsymbol{z \to rank \geq w \to rank}$



# Union example




If $z \to \textit{rank} \geq w \to \textit{rank}$

If $z \to \textit{rank} = w \to \textit{rank}$, then $(z \to \textit{rank})++$

# Properties of Ranks

1. If $\boldsymbol{x}$ not a root, then $(\boldsymbol{x} \rightarrow \boldsymbol{rank}) < (\boldsymbol{x} \rightarrow \boldsymbol{parent} \rightarrow \boldsymbol{rank})$
2. When doing path compression, if parent of $\boldsymbol{x}$ changes, new parent has rank strictly larger than old parent
3. $\boldsymbol{x} \rightarrow \boldsymbol{rank}$ can change only if $\boldsymbol{x}$ a root, and once $\boldsymbol{x}$ is a non-root it never becomes a root again.

# Properties of Ranks

1. If $\boldsymbol{x}$ not a root, then $(\boldsymbol{x} \to \boldsymbol{rank}) < (\boldsymbol{x} \to \boldsymbol{parent} \to \boldsymbol{rank})$
2. When doing path compression, if parent of $\boldsymbol{x}$ changes, new parent has rank strictly larger than old parent
3. $\boldsymbol{x} \to \boldsymbol{rank}$ can change only if $\boldsymbol{x}$ a root, and once $\boldsymbol{x}$ is a non-root it never becomes a root again.
4. When $\boldsymbol{x}$ first reaches rank $\boldsymbol{r}$, there are at least $\boldsymbol{2^r}$ nodes in tree rooted at $\boldsymbol{x}$.

# Properties of Ranks

1. If $x$ not a root, then $(x \to \textit{rank}) < (x \to \textit{parent} \to \textit{rank})$
2. When doing path compression, if parent of $x$ changes, new parent has rank strictly larger than old parent
3. $x \to \textit{rank}$ can change only if $x$ a root, and once $x$ is a non-root it never becomes a root again.
4. When $x$ first reaches rank $r$, there are at least $2^r$ nodes in tree rooted at $x$.

**Proof of Property 4.**

Induction. Base case: $r = 0$.

# Properties of Ranks

1. If $x$ not a root, then $(x \to \textit{rank}) < (x \to \textit{parent} \to \textit{rank})$
2. When doing path compression, if parent of $x$ changes, new parent has rank strictly larger than old parent
3. $x \to \textit{rank}$ can change only if $x$ a root, and once $x$ is a non-root it never becomes a root again.
4. When $x$ first reaches rank $r$, there are at least $2^r$ nodes in tree rooted at $x$.

## Proof of Property 4.

Induction. Base case: $r = 0$. ✓

# Properties of Ranks

1. If $\boldsymbol{x}$ not a root, then $(\boldsymbol{x} \to \boldsymbol{rank}) < (\boldsymbol{x} \to \boldsymbol{parent} \to \boldsymbol{rank})$
2. When doing path compression, if parent of $\boldsymbol{x}$ changes, new parent has rank strictly larger than old parent
3. $\boldsymbol{x} \to \boldsymbol{rank}$ can change only if $\boldsymbol{x}$ a root, and once $\boldsymbol{x}$ is a non-root it never becomes a root again.
4. When $\boldsymbol{x}$ first reaches rank $\boldsymbol{r}$, there are at least $\mathbf{2}^{\boldsymbol{r}}$ nodes in tree rooted at $\boldsymbol{x}$.

**Proof of Property 4.**

Induction. Base case: $\boldsymbol{r} = \mathbf{0}$. ✓

Inductive case: Suppose true for $\boldsymbol{r} - \mathbf{1}$.

# Properties of Ranks

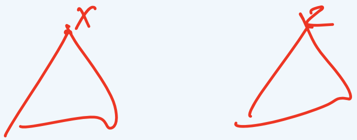
1. If $x$ not a root, then $(x \to \textbf{\textit{rank}}) < (x \to \textbf{\textit{parent}} \to \textbf{\textit{rank}})$
2. When doing path compression, if parent of $x$ changes, new parent has rank strictly larger than old parent
3. $x \to \textbf{\textit{rank}}$ can change only if $x$ a root, and once $x$ is a non-root it never becomes a root again.
4. When $x$ first reaches rank $r$, there are at least $2^r$ nodes in tree rooted at $x$.

## Proof of Property 4.

Induction. Base case: $r = 0$. ✓

Inductive case: Suppose true for $r - 1$.

When $x$ first gets rank $r$, must be because $x$ had rank $r - 1$ (and was root), unioned with another set with root $z$ of rank $r - 1$.



# Properties of Ranks

1. If $\boldsymbol{x}$ not a root, then $(\boldsymbol{x} \to \boldsymbol{rank}) < (\boldsymbol{x} \to \boldsymbol{parent} \to \boldsymbol{rank})$
2. When doing path compression, if parent of $\boldsymbol{x}$ changes, new parent has rank strictly larger than old parent
3. $\boldsymbol{x} \to \boldsymbol{rank}$ can change only if $\boldsymbol{x}$ a root, and once $\boldsymbol{x}$ is a non-root it never becomes a root again.
4. When $\boldsymbol{x}$ first reaches rank $\boldsymbol{r}$, there are at least $\mathbf{2}^{\boldsymbol{r}}$ nodes in tree rooted at $\boldsymbol{x}$.

## Proof of Property 4.

Induction. Base case: $\boldsymbol{r} = \mathbf{0}$. ✓

Inductive case: Suppose true for $\boldsymbol{r} - \mathbf{1}$.

When $\boldsymbol{x}$ first gets rank $\boldsymbol{r}$, must be because $\boldsymbol{x}$ had rank $\boldsymbol{r} - \mathbf{1}$ (and was root), unioned with another set with root $\boldsymbol{z}$ of rank $\boldsymbol{r} - \mathbf{1}$.

$\Longrightarrow$ By induction, at least $\mathbf{2}^{\boldsymbol{r}-\mathbf{1}}$ nodes in each tree

# Properties of Ranks

1. If $x$ not a root, then $(x \to \textit{rank}) < (x \to \textit{parent} \to \textit{rank})$
2. When doing path compression, if parent of $x$ changes, new parent has rank strictly larger than old parent
3. $x \to \textit{rank}$ can change only if $x$ a root, and once $x$ is a non-root it never becomes a root again.
4. When $x$ first reaches rank $r$, there are at least $2^r$ nodes in tree rooted at $x$.

## Proof of Property 4.

Induction. Base case: $r = 0$. ✓

Inductive case: Suppose true for $r - 1$.

When $x$ first gets rank $r$, must be because $x$ had rank $r - 1$ (and was root), unioned with another set with root $z$ of rank $r - 1$.

$\Longrightarrow$ By induction, at least $2^{r-1}$ nodes in each tree

$\Longrightarrow$ At least $2^{r-1} + 2^{r-1} = 2^r$ nodes in combined tree. □

# Nodes of rank $r$

## Lemma

*There are at most $n/2^r$ nodes of rank at least $r$.*

## Proof.

Let $x$ node of rank at least $r$. Let $S_x$ be descendants of $x$ when it first got rank $r$.
$\implies |S_x| \geq 2^r$ by property 4.

# Nodes of rank $r$

## Lemma

*There are at most $n/2^r$ nodes of rank at least $r$.*

## Proof.

Let $x$ node of rank at least $r$. Let $S_x$ be descendants of $x$ when it first got rank $r$.
$\implies |S_x| \geq 2^r$ by property 4.

Let $z$ some other node of rank $\geq r$. Without loss of generality, suppose $x$ got rank $r$ before $z$. Consider some $e \in S_x$. Then $e$ can't be in $S_z$ (already in tree with rank $\geq r$). So $S_x \cap S_z = \emptyset$.

# Nodes of rank $r$

## Lemma

*There are at most $n/2^r$ nodes of rank at least $r$.*

## Proof.

Let $x$ node of rank at least $r$. Let $S_x$ be descendants of $x$ when it first got rank $r$.
$\implies |S_x| \geq 2^r$ by property 4.

Let $z$ some other node of rank $\geq r$. Without loss of generality, suppose $x$ got rank $r$ before $z$. Consider some $e \in S_x$. Then $e$ can't be in $S_z$ (already in tree with rank $\geq r$). So $S_x \cap S_z = \emptyset$.

$\implies$ At most $n/2^r$ nodes of rank $\geq r$. $\square$

# Main Result I

**Theorem**

*When using Path Compression and Union By Rank, total time at most* $O(m \log^* n)$.

# Main Result I

**Theorem**

*When using Path Compression and Union By Rank, total time at most* $O(m \log^* n)$.

$m$ operations total. Analyze each type separately:

- Make-Set: $O(1)$ time each
- Union: two Find operations, plus $O(1)$ other work.
- Find($x$): proportional to depth of $x$. Count number of parent pointers followed, call this the time.

# Main Result I

**Theorem**

*When using Path Compression and Union By Rank, total time at most $O(m \log^* n)$.*

$m$ operations total. Analyze each type separately:

- Make-Set: $O(1)$ time each
- Union: two Find operations, plus $O(1)$ other work.
- Find($x$): proportional to depth of $x$. Count number of parent pointers followed, call this the time.

So at most $2m$ Finds, want to bound total # parent pointers followed.

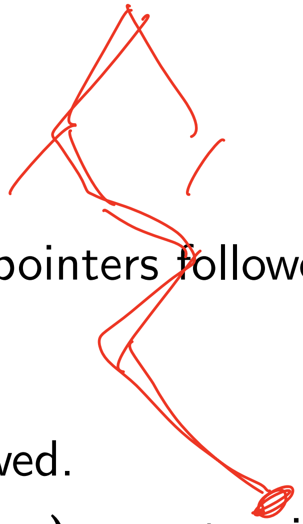
# Main Result I

**Theorem**

*When using Path Compression and Union By Rank, total time at most* $O(m \log^* n)$.

$m$ operations total. Analyze each type separately:

- Make-Set: $O(1)$ time each
- Union: two Find operations, plus $O(1)$ other work.
- Find($x$): proportional to depth of $x$. Count number of parent pointers followed, call this the time.



So at most $2m$ Finds, want to bound total # parent pointers followed.

- At most one parent pointer to root per Find $\implies$ at most $O(m)$ parent pointers to roots.
- So only need to worry about parent pointers to non-roots.

# Main Result II: Buckets

Put elements in buckets according to rank (only in analysis).

Notation: $\boldsymbol{2 \uparrow i}$ denote a tower of $\boldsymbol{i}$ **2**'s

- $\boldsymbol{2 \uparrow 1 = 2, \quad 2 \uparrow 2 = 2^2 = 4, \quad 2 \uparrow 3 = 2^{2^2} = 2^4 = 16, \quad 2 \uparrow 4 = 2^{2^{2^2}} = 2^{16} = 65536}$
- $\boldsymbol{\log^*(2 \uparrow i) = i}$

# Main Result II: Buckets

Put elements in buckets according to rank (only in analysis).

Notation: $2 \uparrow i$ denote a tower of $i$ 2's

- $2 \uparrow 1 = 2, \quad 2 \uparrow 2 = 2^2 = 4, \quad 2 \uparrow 3 = 2^{2^2} = 2^4 = 16, \quad 2 \uparrow 4 = 2^{2^{2^2}} = 2^{16} = 65536$
- $\log^*(2 \uparrow i) = i$

$B(i)$ (Bucket $i$): All elements of rank at least $2 \uparrow (i-1)$, at most $(2 \uparrow i) - 1$

- Bucket $0$: nodes with rank $0$
- Bucket $1$: rank at least $1$, at most $1$
- Bucket $2$: rank at least $2$, at most $3$
- Bucket $3$: rank at least $4$, at most $15$
- Bucket $4$: rank at least $16$, at most $65535$

# Main Result II: Buckets

Put elements in buckets according to rank (only in analysis).

Notation: $\mathbf{2 \uparrow \boldsymbol{i}}$ denote a tower of $\boldsymbol{i}$ $\mathbf{2}$'s

- $2 \uparrow 1 = 2, \quad 2 \uparrow 2 = 2^2 = 4, \quad 2 \uparrow 3 = 2^{2^2} = 2^4 = 16, \quad 2 \uparrow 4 = 2^{2^{2^2}} = 2^{16} = 65536$
- $\log^*(2 \uparrow i) = i$

$B(i)$ (Bucket $i$): All elements of rank at least $2 \uparrow (i-1)$, at most $(2 \uparrow i) - 1$

- Bucket $0$: nodes with rank $0$
- Bucket $1$: rank at least $1$, at most $1$
- Bucket $2$: rank at least $2$, at most $3$
- Bucket $3$: rank at least $4$, at most $15$
- Bucket $4$: rank at least $16$, at most $65535$
- At most $\log^* n$ buckets.

# Main Result II: Buckets

Put elements in buckets according to rank (only in analysis).

Notation: $2 \uparrow i$ denote a tower of $i$ 2's

- $2 \uparrow 1 = 2, \quad 2 \uparrow 2 = 2^2 = 4, \quad 2 \uparrow 3 = 2^{2^2} = 2^4 = 16, \quad 2 \uparrow 4 = 2^{2^{2^2}} = 2^{16} = 65536$
- $\log^*(2 \uparrow i) = i$

$B(i)$ (Bucket $i$): All elements of rank at least $2 \uparrow (i-1)$, at most $(2 \uparrow i) - 1$

- Bucket $0$: nodes with rank $0$
- Bucket $1$: rank at least $1$, at most $1$
- Bucket $2$: rank at least $2$, at most $3$
- Bucket $3$: rank at least $4$, at most $15$
- Bucket $4$: rank at least $16$, at most $65535$
- At most $\log^* n$ buckets.

From Lemma: at most $n/(2^{2 \uparrow (i-1)}) = n/(2 \uparrow i)$ elements in bucket $i$.

# Main Result III

Want to bound total # parent pointers (to non-roots) followed over all $\leq \boldsymbol{2m}$ Finds.

# Main Result III

Want to bound total # parent pointers (to non-roots) followed over all $\leq \boldsymbol{2m}$ Finds.

Type 1: Parent pointers that cross buckets

- $\leq \log^* n$ buckets $\implies \leq \log^* n$ per Find $\implies \leq 2m\log^* n = O(m\log^* n)$ total

# Main Result III

Want to bound total # parent pointers (to non-roots) followed over all $\leq 2m$ Finds.

Type 1: Parent pointers that cross buckets

- $\leq \log^* n$ buckets $\implies \leq \log^* n$ per Find $\implies \leq 2m \log^* n = O(m \log^* n)$ total

Type 2: Parent pointers that do not cross buckets

- For each $x$, let $\alpha(x) = \#$ times follow parent point from $x$ to parent in same bucket, not root. Want to show $\sum_x \alpha(x) \leq O(m \log^* n)$.
- Since $x$ not root when following pointers, always has same rank

# Main Result III

Want to bound total # parent pointers (to non-roots) followed over all $\leq 2m$ Finds.

Type 1: Parent pointers that cross buckets

- $\leq \log^* n$ buckets $\implies \leq \log^* n$ per Find $\implies \leq 2m \log^* n = O(m \log^* n)$ total

Type 2: Parent pointers that do not cross buckets

- For each $x$, let $\alpha(x) = \#$ times follow parent point from $x$ to parent in same bucket, not root. Want to show $\sum_x \alpha(x) \leq O(m \log^* n)$.
- Since $x$ not root when following pointers, always has same rank
- Whenever $x$'s pointer followed, gets new parent (path compression)
  $\implies$ rank of parent goes up by at least $1$ (properties of rank)
  $\implies$ happens at most $2 \uparrow i$ times if $x$ in bucket $i$
  $\implies \alpha(x) \leq 2 \uparrow i$.

# Main Result III

Want to bound total # parent pointers (to non-roots) followed over all $\leq 2m$ Finds.

Type 1: Parent pointers that cross buckets

- $\leq \log^* n$ buckets $\implies \leq \log^* n$ per Find $\implies \leq 2m\log^* n = O(m\log^* n)$ total

Type 2: Parent pointers that do not cross buckets

- For each $x$, let $\alpha(x) = \#$ times follow parent point from $x$ to parent in same bucket, not root. Want to show $\sum_x \alpha(x) \leq O(m\log^* n)$.
- Since $x$ not root when following pointers, always has same rank
- Whenever $x$'s pointer followed, gets new parent (path compression)

  $\implies$ rank of parent goes up by at least $1$ (properties of rank)

  $\implies$ happens at most $2 \uparrow i$ times if $x$ in bucket $i$

  $\implies$ $\alpha(x) \leq 2 \uparrow i$.

$$\sum_x \alpha(x) = \sum_{i=0}^{O(\log^* n)} \sum_{x \in B(i)} \alpha(x) \leq \sum_{i=0}^{O(\log^* n)} \sum_{x \in B(i)} (2 \uparrow i) \leq \sum_{i=0}^{O(\log^* n)} \frac{n}{2 \uparrow i}(2 \uparrow i) = O(n\log^* n)$$

$$\leq O(m\log^* n)$$