Lecture 9: Disjoint Sets / Union-Find

Michael Dinitz

September 28, 2021

601.433/633 Introduction to Algorithms

Introduction

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

Slightly more formally:

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

Introduction

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

Slightly more formally:

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

Rules: every set has a *unique* representative.

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

Introduction

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

Slightly more formally:

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

Rules: every set has a *unique* representative.

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

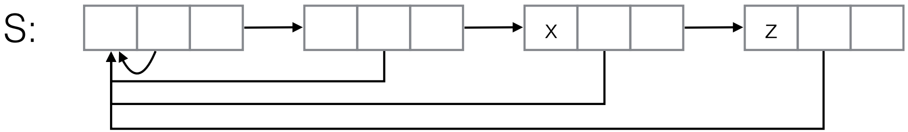
Notation and Notes:

- **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.

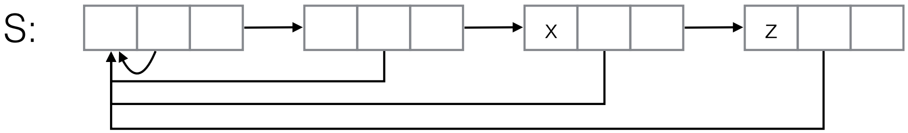
- Representative of set is head (first element on list)
- Each element has pointer to head and to next element, so stored as triple: (element, head, next)



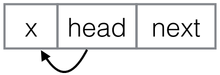
First Approach: Lists

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)



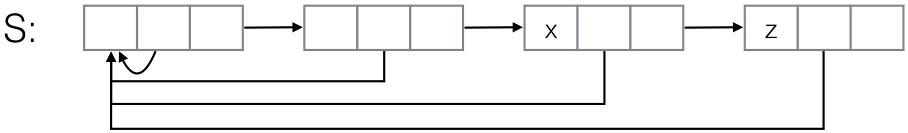
Make-Set($\mathbf{x}$):



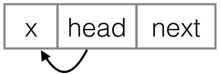
First Approach: Lists

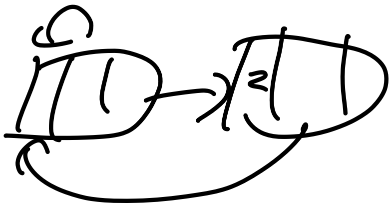
Linked list for each set.

- 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)



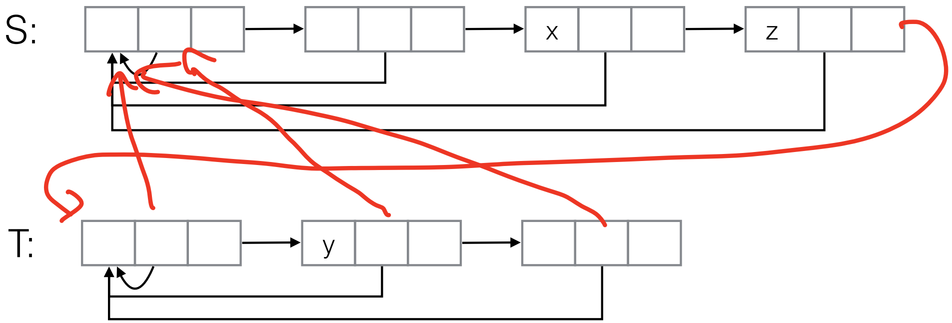
Make-Set(**x**):





Find(**x**): return **x** → head

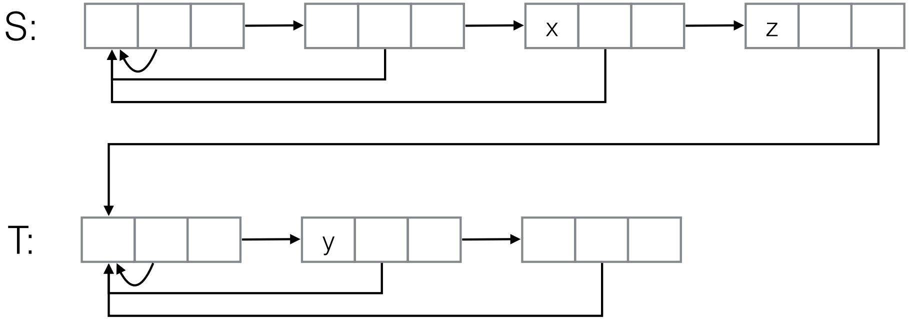
Union($\mathbf{x}, \mathbf{y}$)



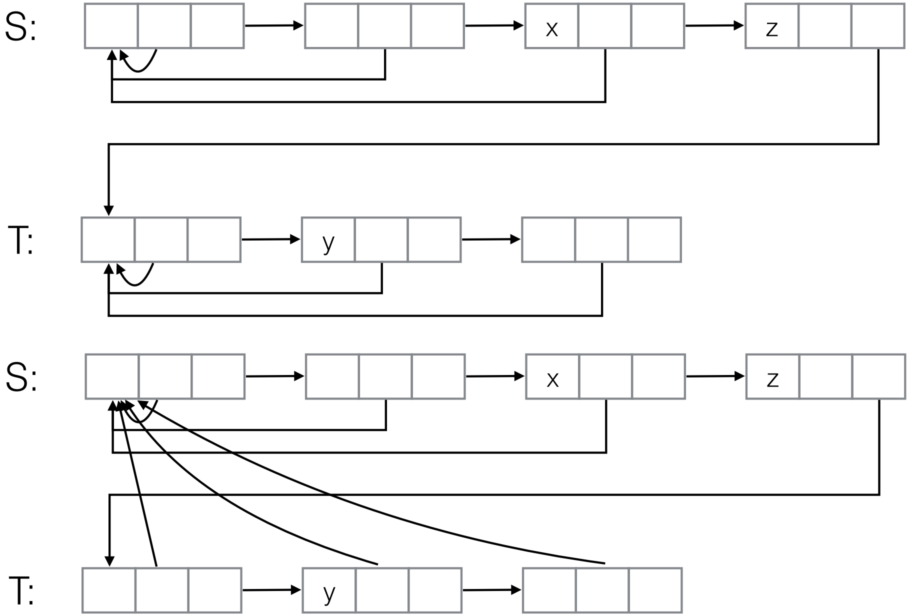
Obvious approach:

- Walk down **S** to final element **z** (starting from **x**)
- Set **z** → next = **y** → head
- Walk down **T**, set every elements head pointer to **x** → head

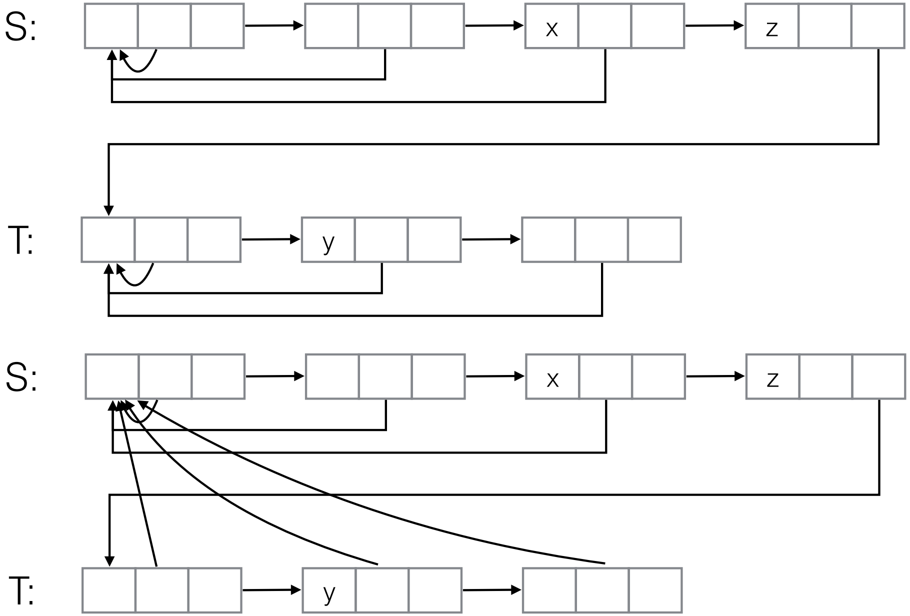
Union($\mathbf{x}, \mathbf{y}$)



Union($\mathbf{x}, \mathbf{y}$)

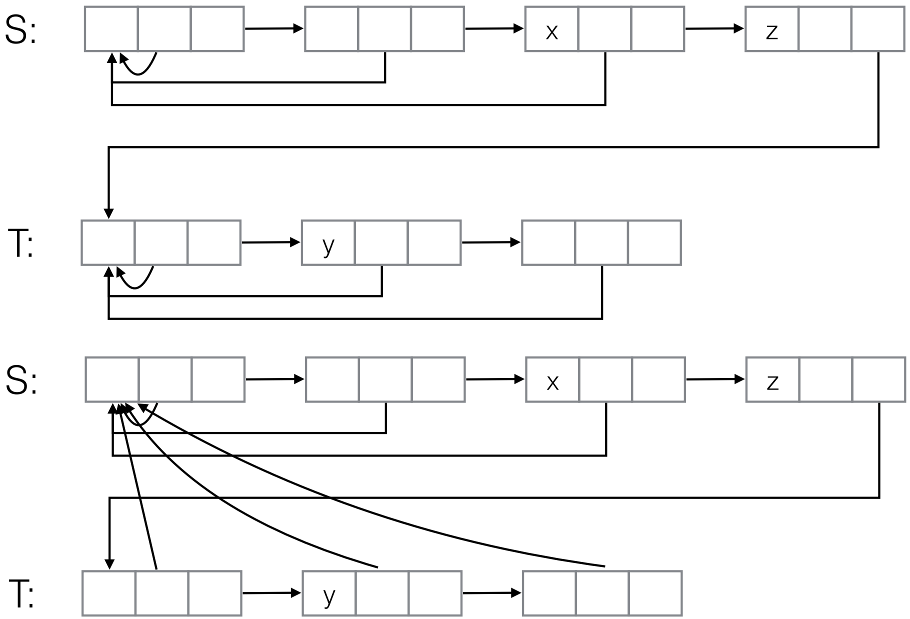


Union(**x**, **y**)



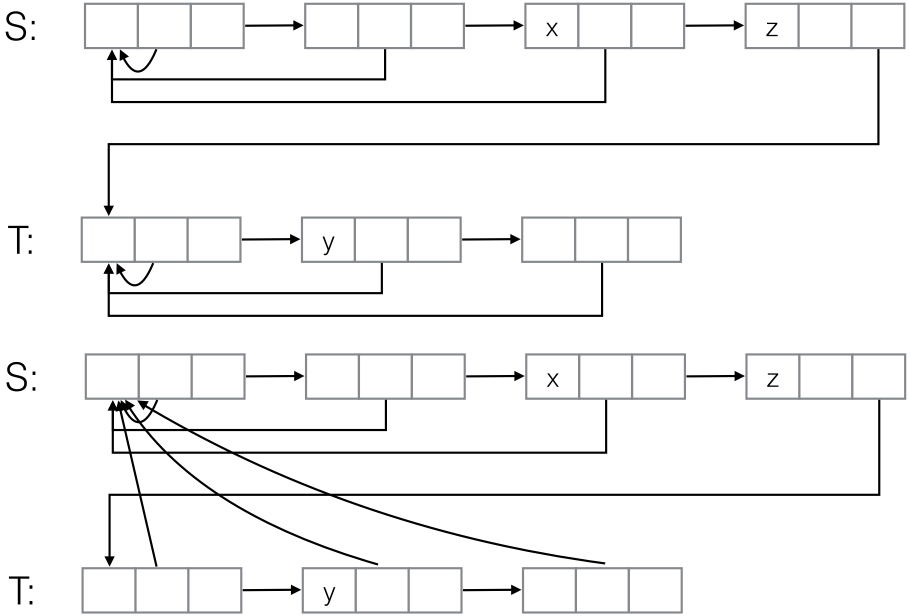
Running time:

Union(x, y)



Running time: $\mathbf{O}(|\mathbf{S}| + |\mathbf{T}|)$

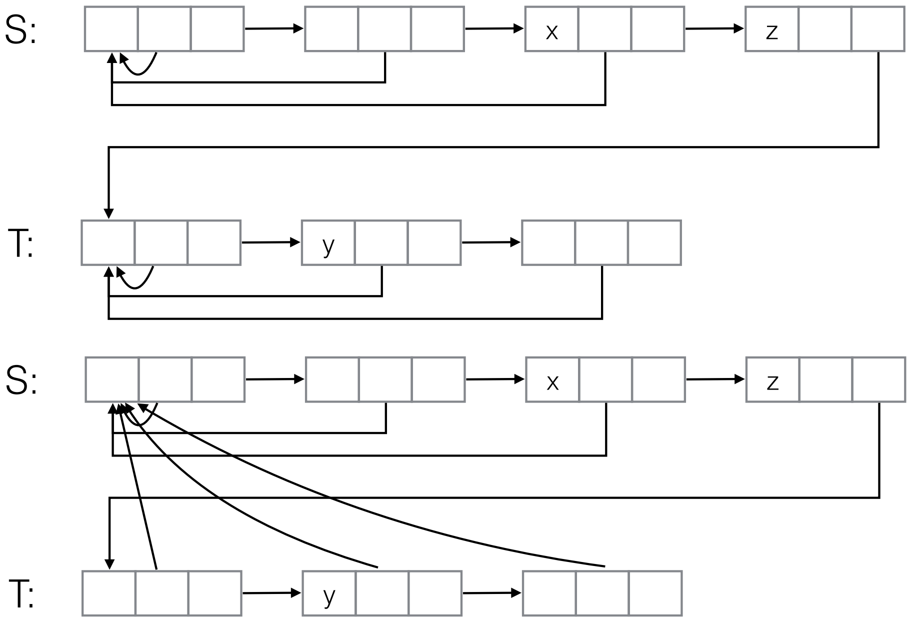
Union($\mathbf{x}, \mathbf{y}$)



Running time: $\mathbf{O}(|\mathbf{S}| + |\mathbf{T}|)$

- $|\mathbf{S}|$ to walk down $\mathbf{S}$ to final element
- $|\mathbf{T}|$ to walk down $\mathbf{T}$ resetting head pointers

Union($\mathbf{x}, \mathbf{y}$)



Running time: $\mathbf{O(|S| + |T|)}$

- $|\mathbf{S}|$ to walk down $\mathbf{S}$ to final element
- $|\mathbf{T}|$ to walk down $\mathbf{T}$ resetting head pointers

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

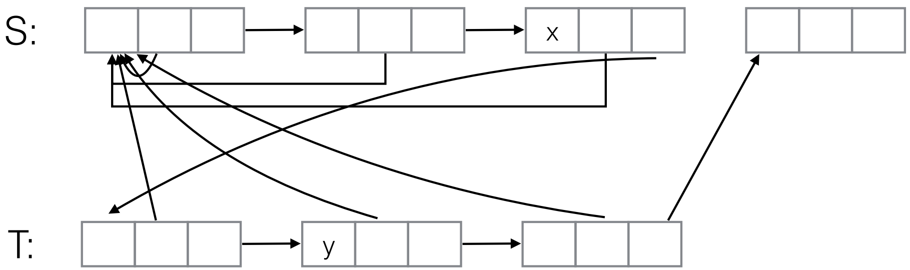
Improved Union($\mathbf{x}, \mathbf{y}$)

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

Improved Union($\mathbf{x}, \mathbf{y}$)

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

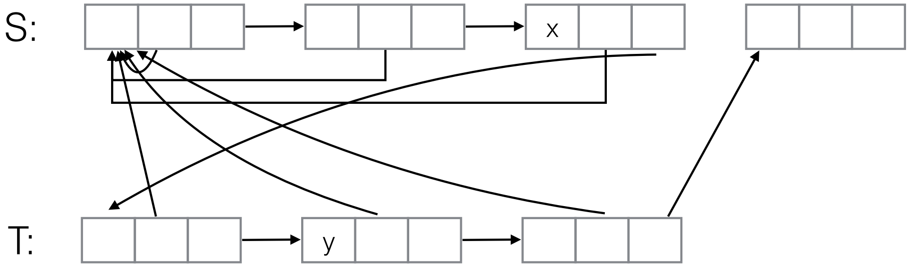
- Splice **T** into **S** right after **x**



Improved Union($\mathbf{x}, \mathbf{y}$)

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

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

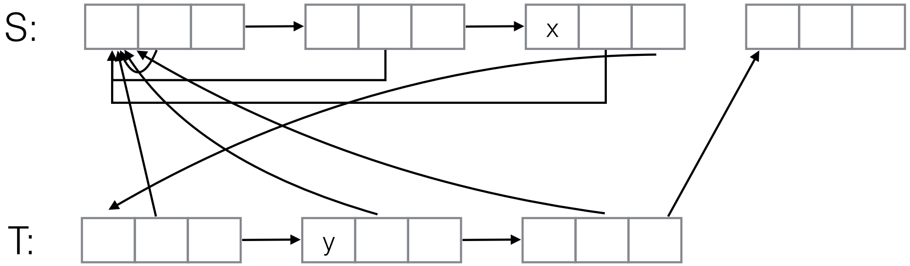


Running time:

Improved Union($\mathbf{x}, \mathbf{y}$)

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

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

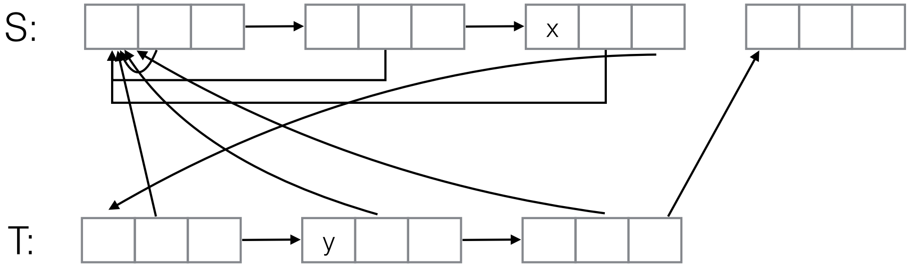


Running time: $\mathbf{O(|T|)}$

Improved Union($\mathbf{x}, \mathbf{y}$)

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

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



Running time: $\mathbf{O(|T|)}$

- Still can't say anything better than $\mathbf{O(n)}$

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

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

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

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

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

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

Splice smaller into bigger!

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

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

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

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

Splice smaller into bigger!

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

Theorem

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

Corollary

The total running time is $\mathbf{O(m + n \log n)}$.

Amortized Analysis of List Algorithm

Banking/accounting argument: bank for every element

- When an element is created (via Make-Set), add $\mathbf{\log n}$ 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 $\mathbf{0}$ (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.

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: $O(1) + O(\log n) = O(\log n)$

Find:

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

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

Union:

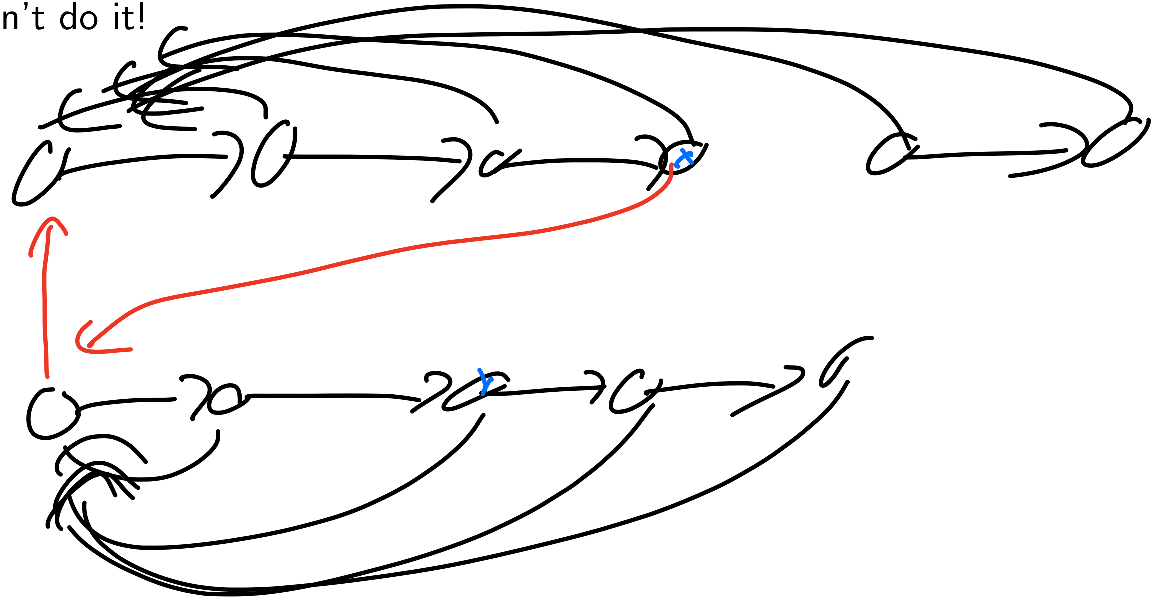
- True cost: $\min(|S|, |T|)$
- Change in banks: $-\min(|S|, |T|)$

$\implies$ 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.

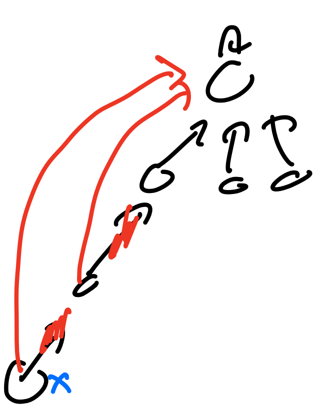
- Slow part of Union: updating all head pointers in smaller list.
- Don't do it!
- Results in trees rather than lists (can drop next pointer)

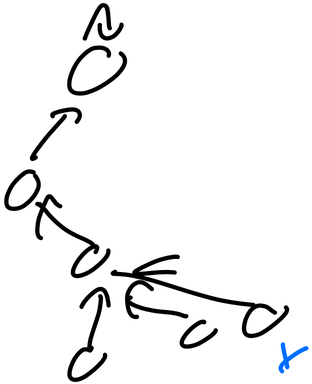
Even Better

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(**x**), change pointers of **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($\mathbf{x}$), change pointers of $\mathbf{x}$ 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 $\mathbf{O(m \log^* n)}$.

$\mathbf{\log^*}$: iterated $\mathbf{\log_2}$.

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

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 ≤ 5.

Main Result

Theorem

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

$\mathbf{\log^*}$: iterated $\mathbf{\log_2}$.

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

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

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

Formal Procedures: Make-Set and Find

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

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

Formal Procedures: Make-Set and Find

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

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

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

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

Formal Procedures: Make-Set and Find

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

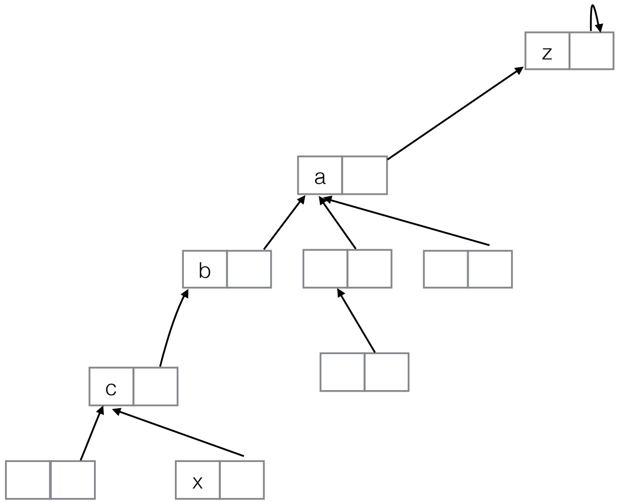
- Running time: $\mathbf{O(1)}$.

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

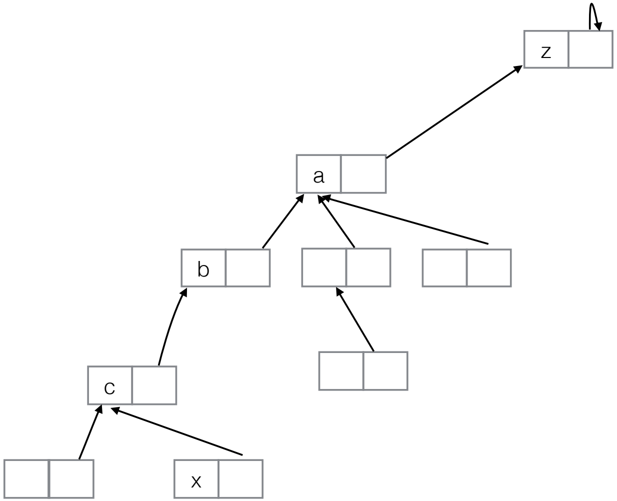
- If $\mathbf{x} \to \mathbf{parent} = \mathbf{x}$ then return $\mathbf{x}$
- $\mathbf{x} \to \mathbf{parent} = \mathbf{Find(x \to parent)}$
- Return $\mathbf{x} \to \mathbf{parent}$

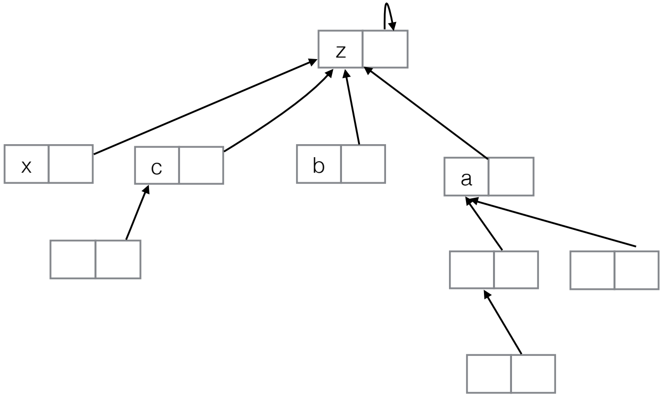
Running time of Find: depth of $\mathbf{x}$ (distance to root)

Find example



Find example





Formal Procedure: Union

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

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

Formal Procedure: Union

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

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

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

Formal Procedure: Union

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

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

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

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

Formal Procedure: Union

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

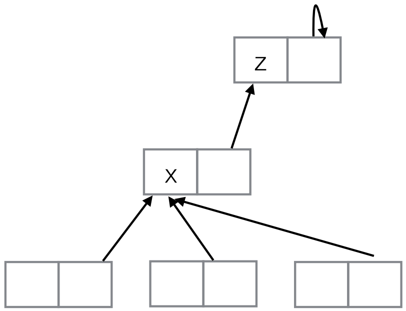
- If $\mathbf{r_1} \rightarrow \mathbf{rank} > \mathbf{r_2} \rightarrow \mathbf{rank}$, set $\mathbf{r_2} \rightarrow \mathbf{parent} = \mathbf{r_1}$
- If $\mathbf{r_2} \rightarrow \mathbf{rank} > \mathbf{r_1} \rightarrow \mathbf{rank}$, set $\mathbf{r_1} \rightarrow \mathbf{parent} = \mathbf{r_2}$
-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

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

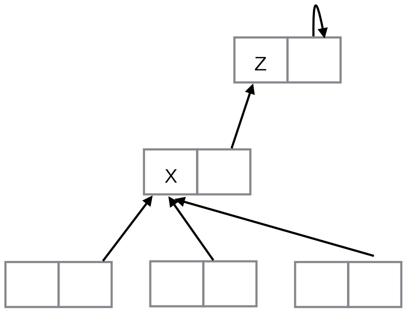
Union($\mathbf{x, y}$): Link(Find($\mathbf{x}$), Find($\mathbf{y}$))

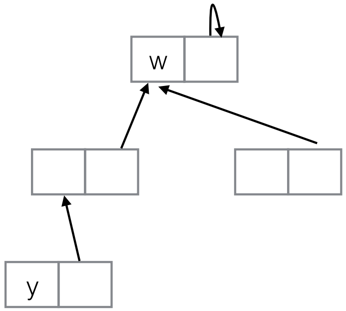
- Running time: depth($\mathbf{x}$) + depth($\mathbf{y}$)

Union example



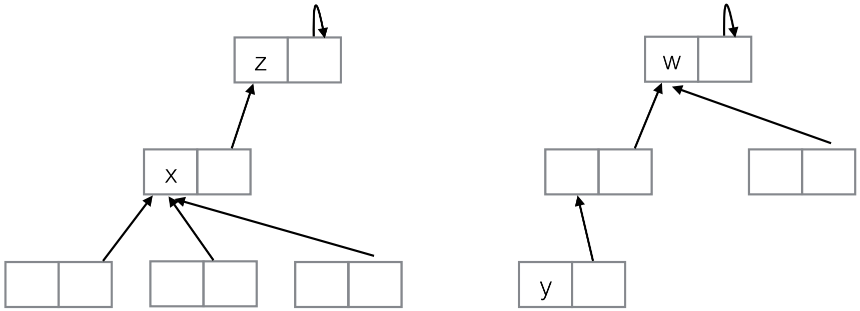
Union example



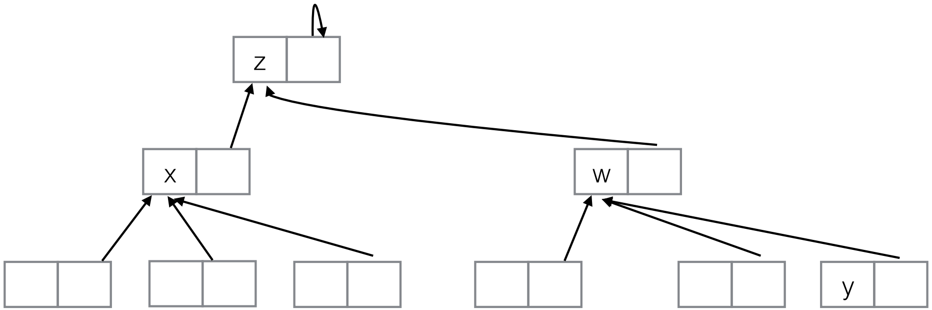


If $\mathbf{z} \to \mathbf{rank} \geq \mathbf{w} \to \mathbf{rank}$

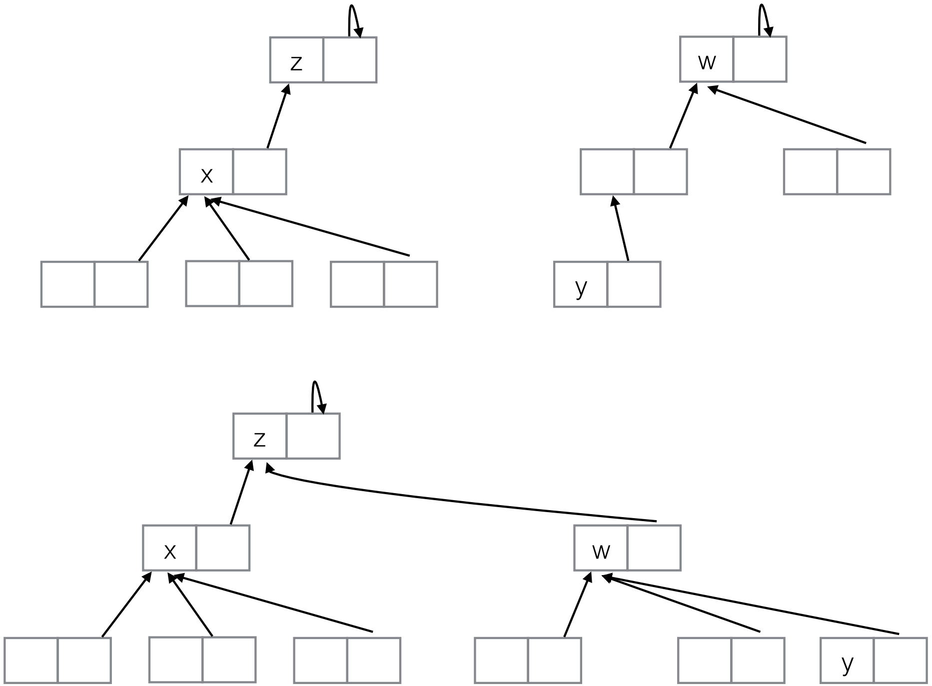
Union example



If **z → rank ≥ w → rank**



Union example



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

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

Properties of Ranks

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

Properties of Ranks

1. If **x** not a root, then $(\mathbf{x} \rightarrow \mathbf{rank}) < (\mathbf{x} \rightarrow \mathbf{parent} \rightarrow \mathbf{rank})$
2. When doing path compression, if parent of **x** changes, new parent has rank strictly larger than old parent
3. $\mathbf{x} \rightarrow \mathbf{rank}$ 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 $\mathbf{2^r}$ nodes in tree rooted at **x**.

Properties of Ranks

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

Proof of Property 4.

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

Properties of Ranks

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

Proof of Property 4.

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

Properties of Ranks

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

Proof of Property 4.

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

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

Properties of Ranks

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

Proof of Property 4.

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

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

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

Properties of Ranks

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

Proof of Property 4.

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

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

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

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

Properties of Ranks

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

Proof of Property 4.

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

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

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

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

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

Nodes of rank $\mathbf{r}$

Lemma

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

Proof.

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

Nodes of rank $\mathbf{r}$

Lemma

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

Proof.

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

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

Nodes of rank $\mathbf{r}$

Lemma

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

Proof.

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

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

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

Main Result I

Theorem

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

Main Result I

Theorem

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

$\mathbf{m}$ operations total. Analyze each type separately:

- Make-Set: $\mathbf{O(1)}$ time each
- Union: two Find operations, plus $\mathbf{O(1)}$ other work.
- Find($\mathbf{x}$): proportional to depth of $\mathbf{x}$. 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 $\mathbf{O(m \log^* n)}$.

$\mathbf{m}$ operations total. Analyze each type separately:

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

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

Main Result I

Theorem

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

$\mathbf{m}$ operations total. Analyze each type separately:

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

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

- At most one parent pointer to root per Find $\implies$ at most $\mathbf{O(m)}$ 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: $\mathbf{2 \uparrow i}$ denote a tower of $\mathbf{i}$ $\mathbf{2}$'s

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

Main Result II: Buckets

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

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

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

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

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

Main Result II: Buckets

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

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

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

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

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

Main Result II: Buckets

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

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

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

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

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

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

Main Result III

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

Main Result III

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

Type 1: Parent pointers that cross buckets

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

Main Result III

Want to bound total # parent pointers (to non-roots) followed over all $\leq \mathbf{2m}$ 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 $\mathbf{x}$, let $\alpha(\mathbf{x})$ = # times follow parent point from $\mathbf{x}$ to parent in same bucket, not root. Want to show $\sum_x \alpha(x) \leq O(m \log^* n)$.
- Since $\mathbf{x}$ not root when following pointers, always has same rank

Main Result III

Want to bound total # parent pointers (to non-roots) followed over all $\leq \mathbf{2m}$ 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 $\mathbf{x}$, let $\alpha(x)$ = # times follow parent point from $\mathbf{x}$ to parent in same bucket, not root. Want to show $\sum_x \alpha(x) \leq O(m\log^* n)$.
- Since $\mathbf{x}$ not root when following pointers, always has same rank
- Whenever $\mathbf{x}$'s pointer followed, gets new parent (path compression)

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

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

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

Main Result III

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

Type 1: Parent pointers that cross buckets

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

Type 2: Parent pointers that do not cross buckets

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

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

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

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

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

$$\leq \mathbf{O(m\log^* n)},$$