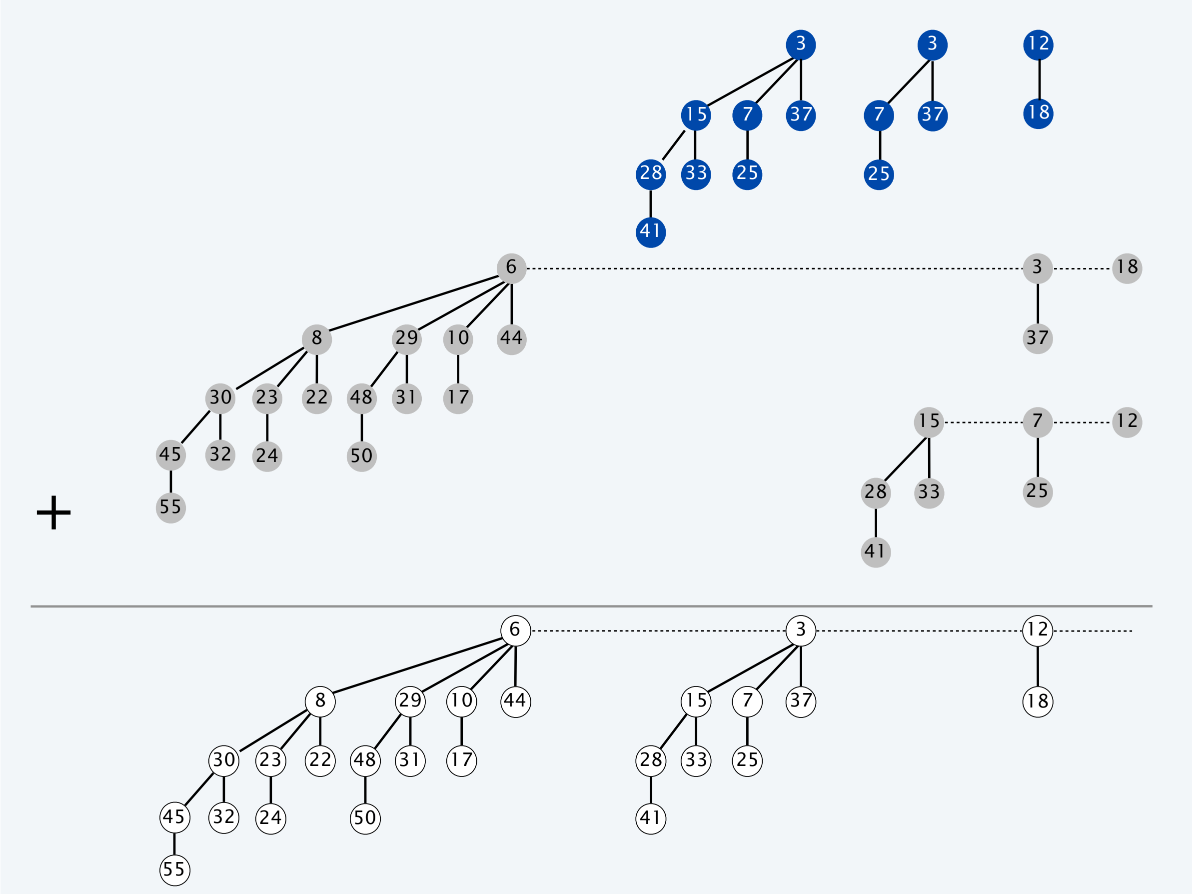
Lecture 10: Disjoint Sets / Union-Find

Jessica Sorrell

September 25, 2025
601.433/633 Introduction to Algorithms
Slides by Mike Dinitz

Meld(H_1, H_2): General Case

(Almost) just like binary addition!



Insert(H, x)

Use Meld:

- Create new heap H' with one B_0 consisting of just x
- Meld(H, H')

Correctness: Obvious

Insert(H, x)

Use Meld:

- Create new heap H' with one B_0 consisting of just x
- Meld(H, H')

Correctness: Obvious

Running Time:

- Worst case: $O(\log n)$ (via Meld)

Insert(H, x)

Use Meld:

- Create new heap H' with one B_0 consisting of just x
- Meld(H, H')

Correctness: Obvious

Running Time:

- Worst case: $O(\log n)$ (via Meld)
- Amortized:
  - Like incrementing a binary counter!

Insert(H, x)

Use Meld:

- Create new heap H' with one B_0 consisting of just x
- Meld(H, H')

Correctness: Obvious

Running Time:

- Worst case: $O(\log n)$ (via Meld)
- Amortized:
  - Like incrementing a binary counter!
  - If we link k trees, potential goes down by $k - 1$
  - Cost = # links plus 1 (for making new heap)
  - Amortized cost = $k + 1 + \Delta\Phi = k + 1 - (k - 1) = 2 = O(1)$

Extract-Min($\boldsymbol{H}$)

Use Meld again!

- $\boldsymbol{O(\log n)}$ to Find-Min: one of the roots.
- Delete and return this root: tree turns into a new heap!
- Meld with original heap (minus the tree)

Correctness: Obvious

Extract-Min(H)

Use Meld again!

- $O(\log n)$ to Find-Min: one of the roots.
- Delete and return this root: tree turns into a new heap!
- Meld with original heap (minus the tree)

Correctness: Obvious

Running Time:

- Worst-Case: $O(\log n)$ from creating new heap, Meld
- Amortized:
  - Potential can go up! But by at most $\log n$
  - Amortized time at most $O(\log n) + \log n = O(\log n)$

Introduction to Union-Find

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

For $x, y \in S$

Find(x) $\stackrel{?}{=}$ Find(y)

Introduction to Union-Find

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

Slightly more formally:

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

Rules: every set has a *unique* representative.

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

Introduction to Union-Find

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

Slightly more formally:

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

Rules: every set has a *unique* representative.

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

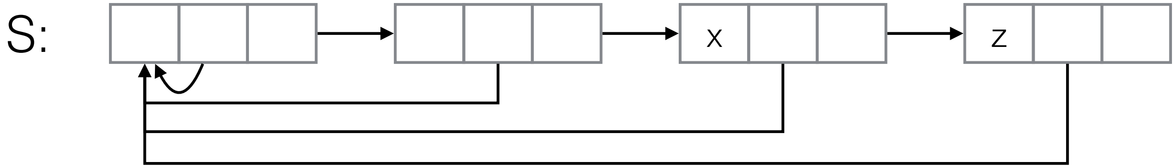
Notation and Notes:

- $\boldsymbol{m}$ operations total
- $\boldsymbol{n}$ of which are Make-Sets (so $\boldsymbol{n}$ 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)

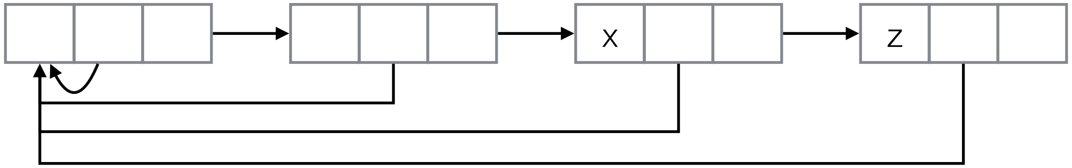


First Approach: Lists

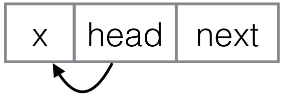
Linked list for each set.

- 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)

S:



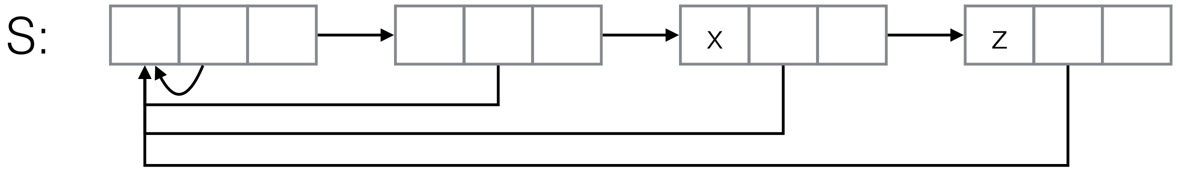
Make-Set(x):



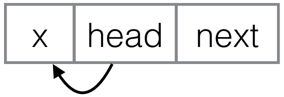
First Approach: Lists

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**)**

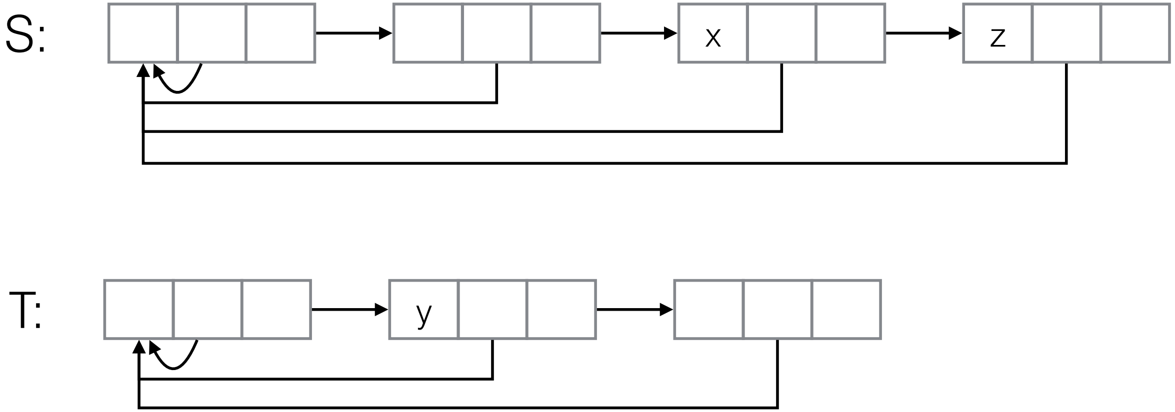


Make-Set(***x***):

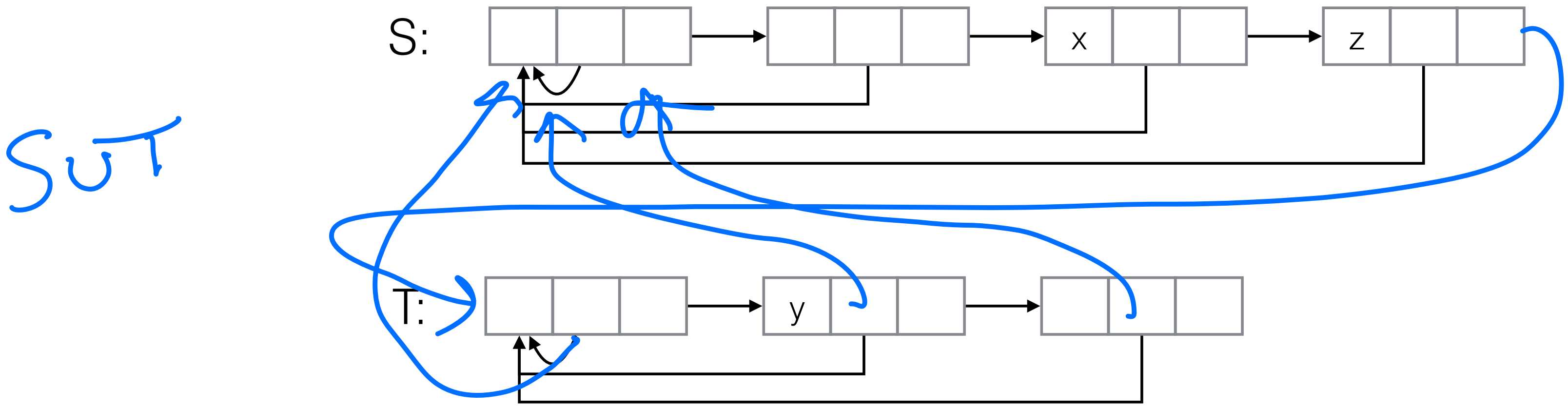


Find(***x***): return ***x*** → head

Union(x, y)



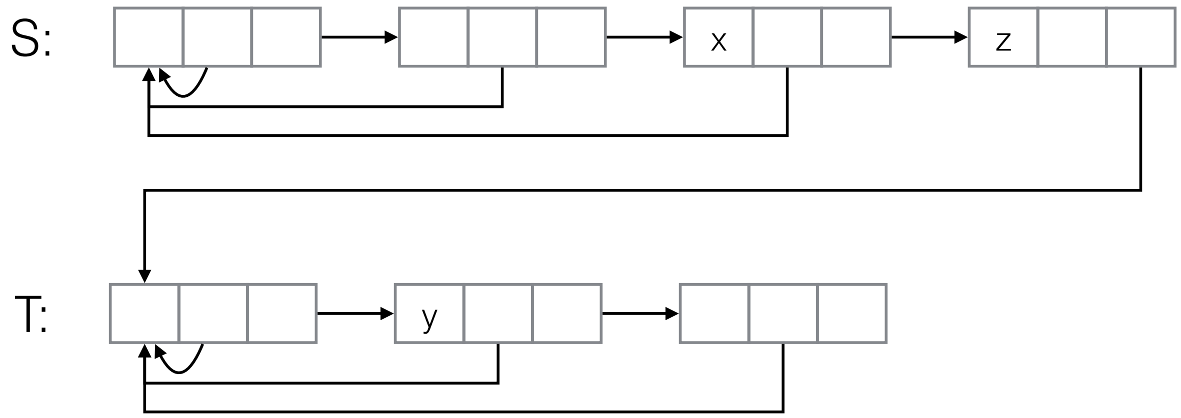
Union(x, y)



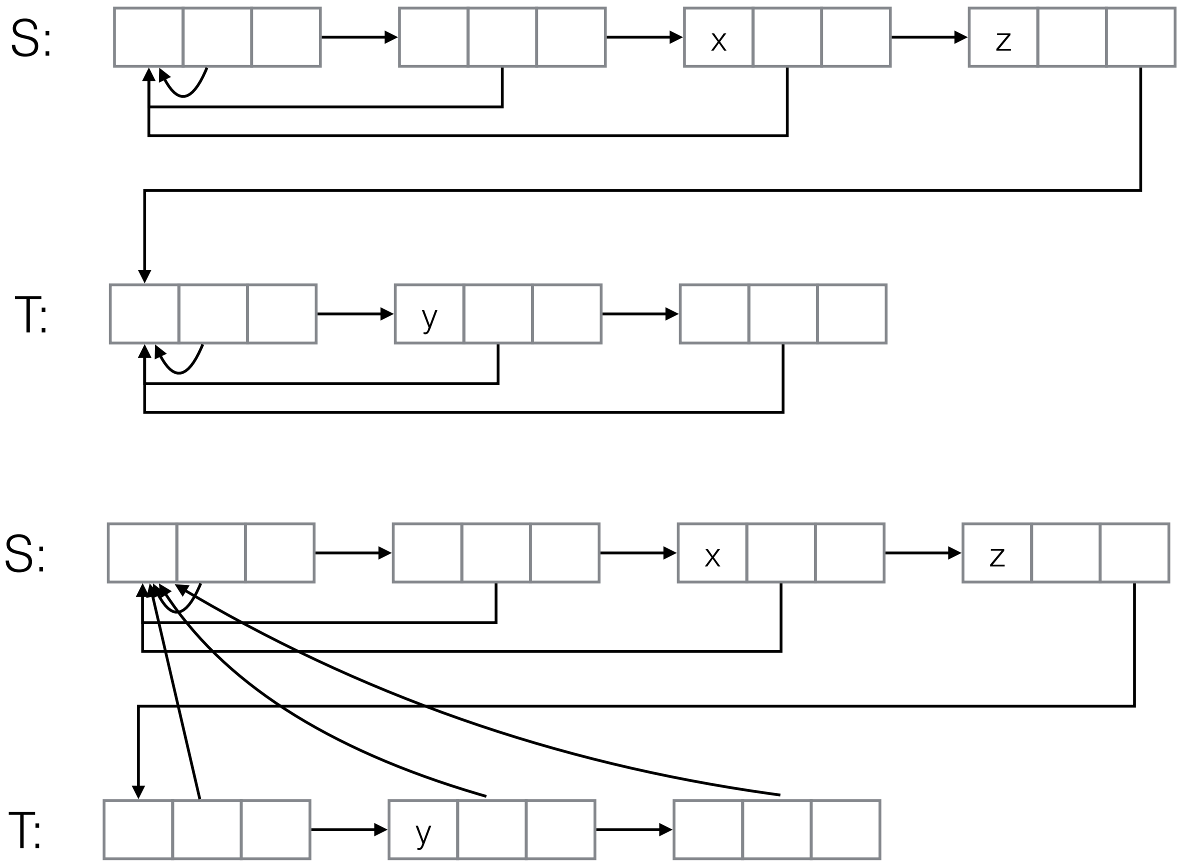
Obvious approach:

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

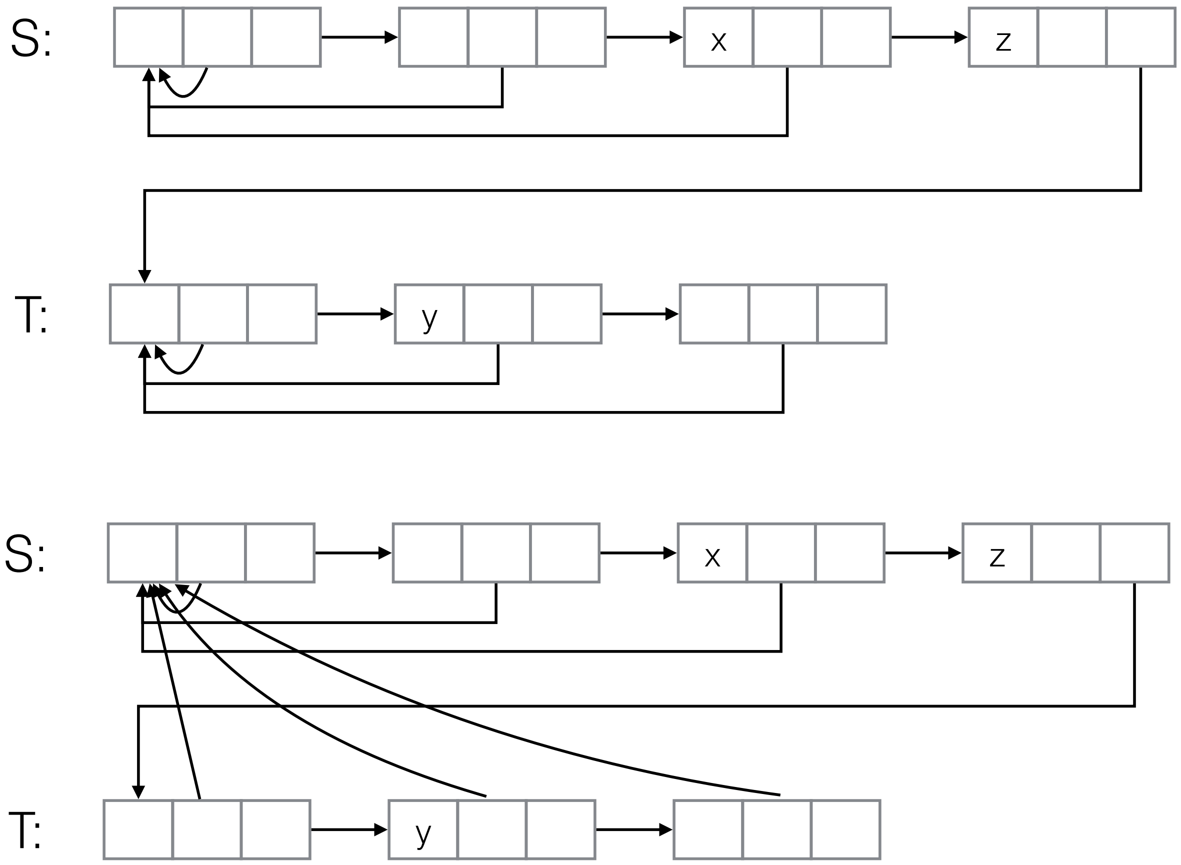
Union($\boldsymbol{x}, \boldsymbol{y}$)



Union(x, y)

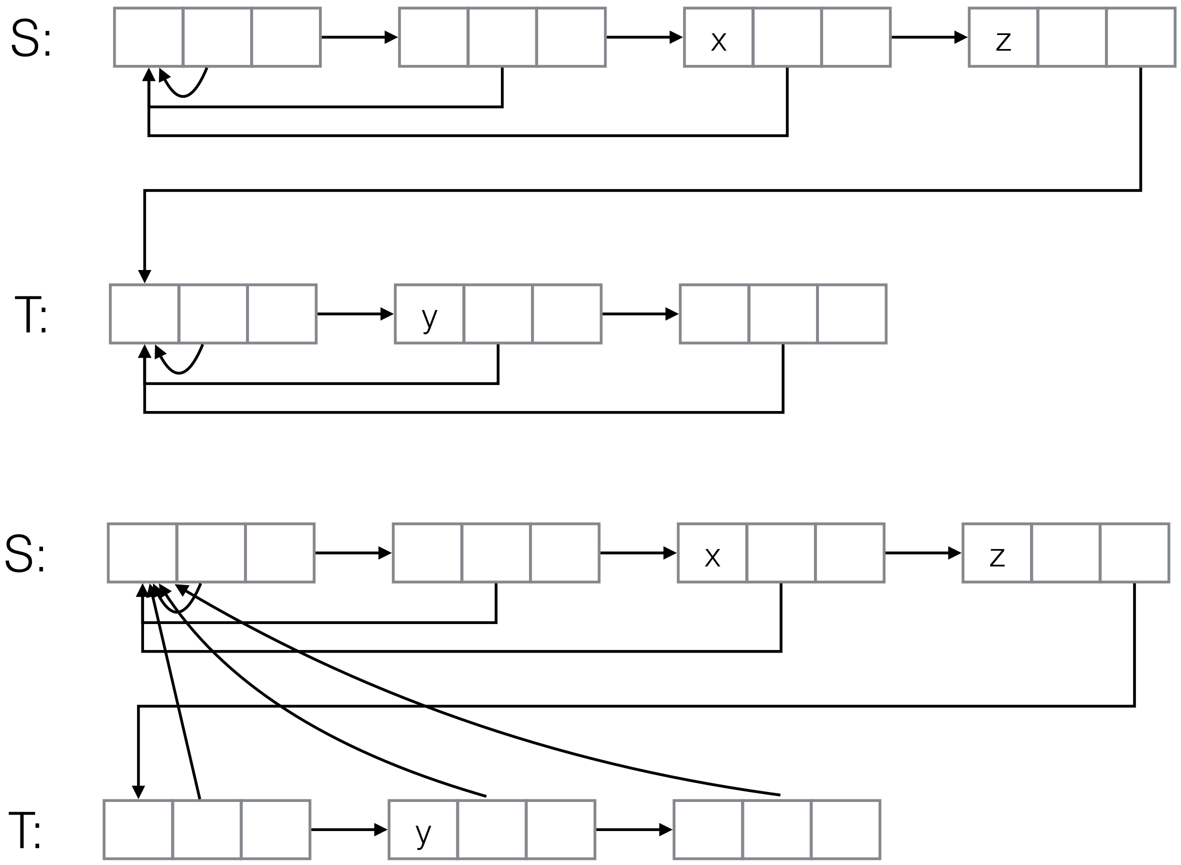


Union(x, y)



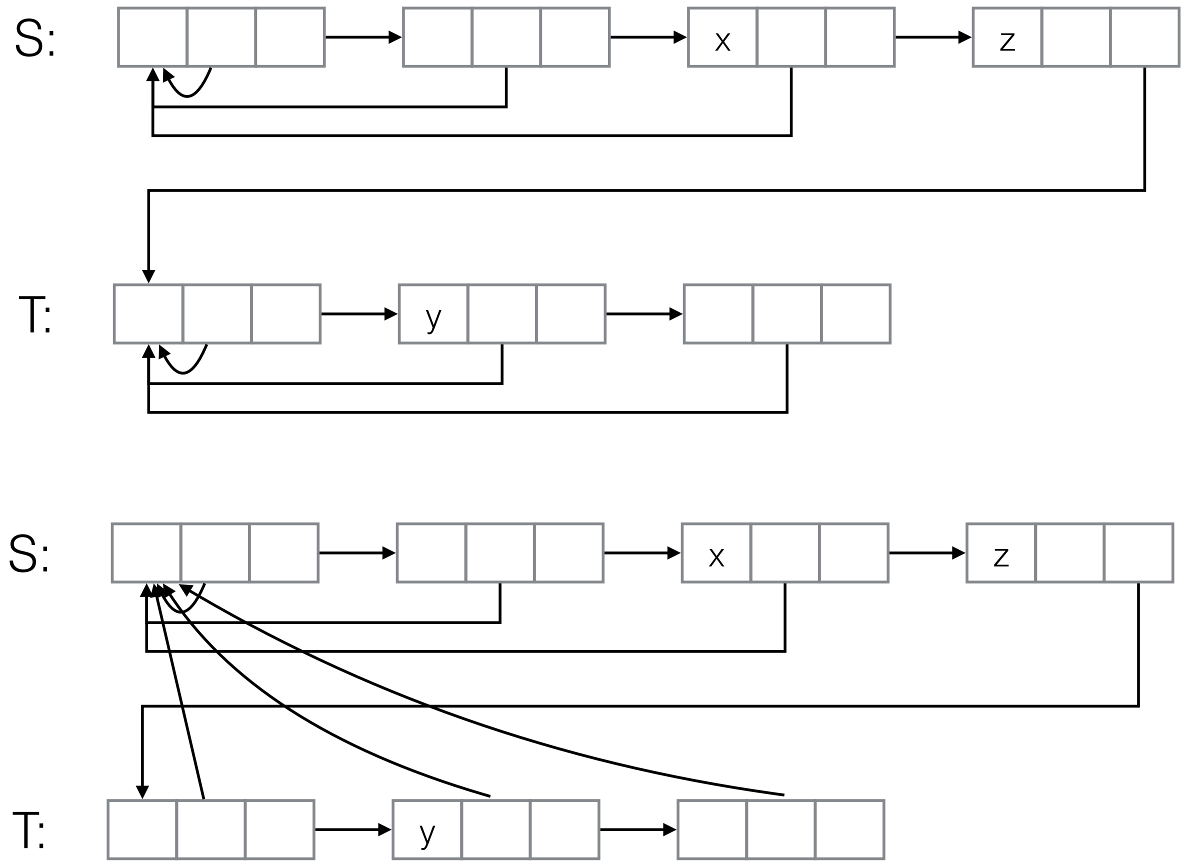
Running time:

Union(x, y)



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

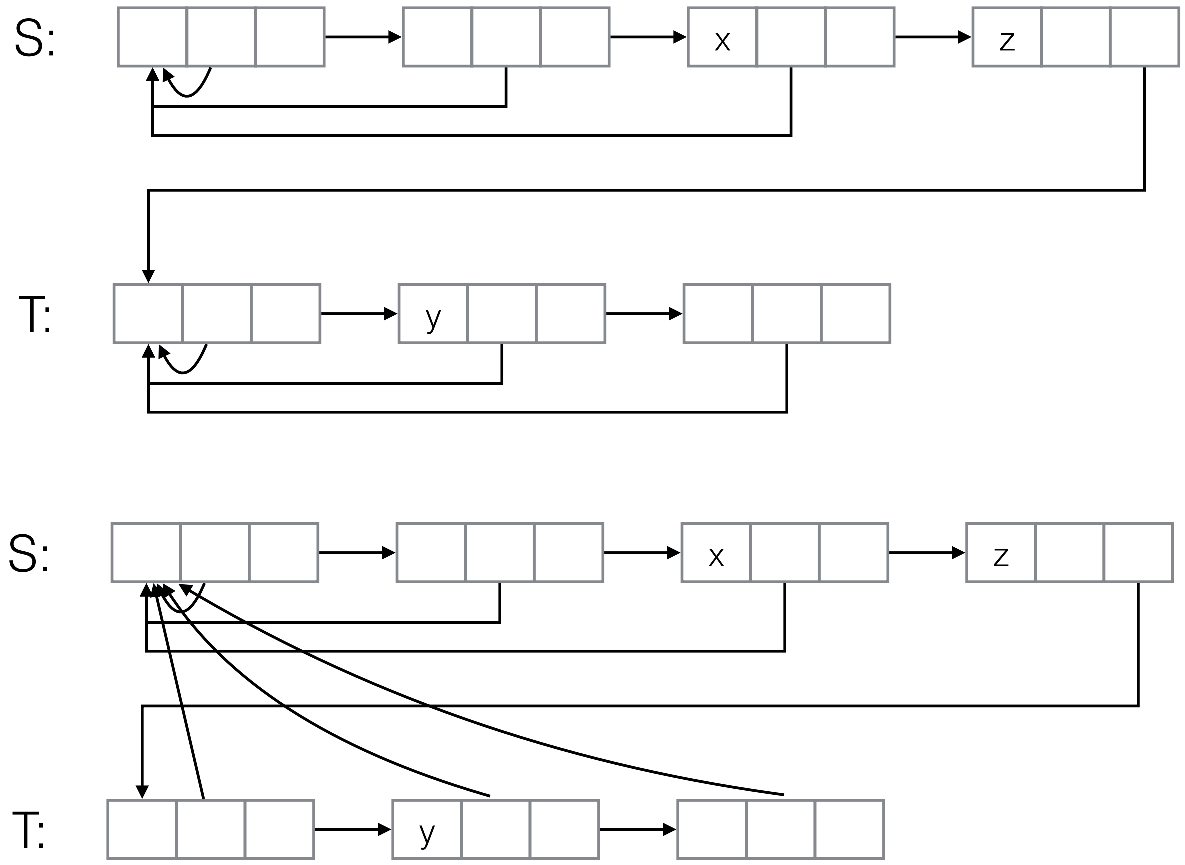
Union(x, y)



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

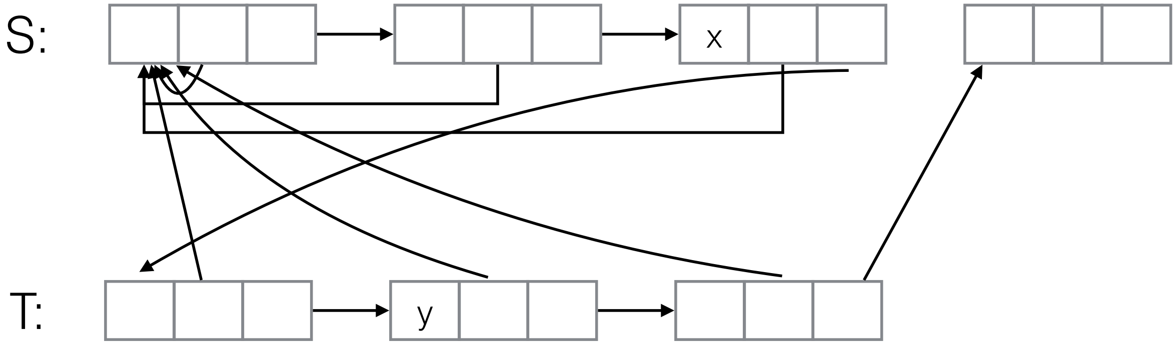
Improved Union(x, y)

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

Improved Union(x, y)

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

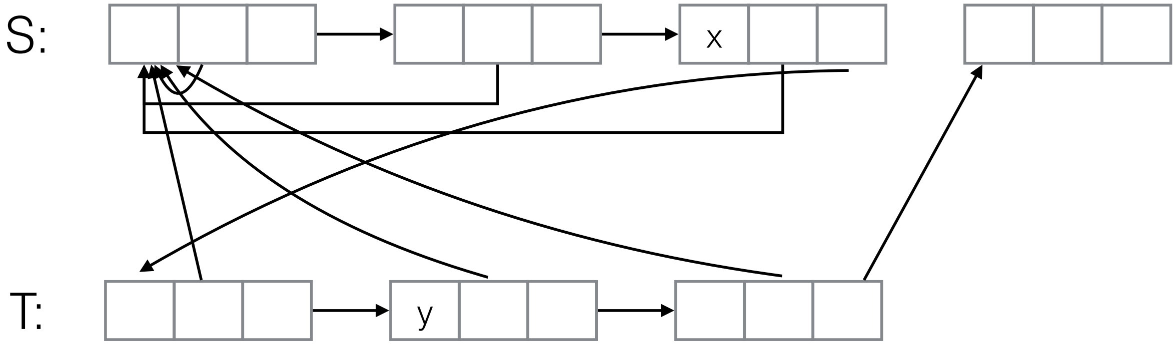
- Splice T into S right after x



Improved Union(x, y)

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

- Splice T into S right after x

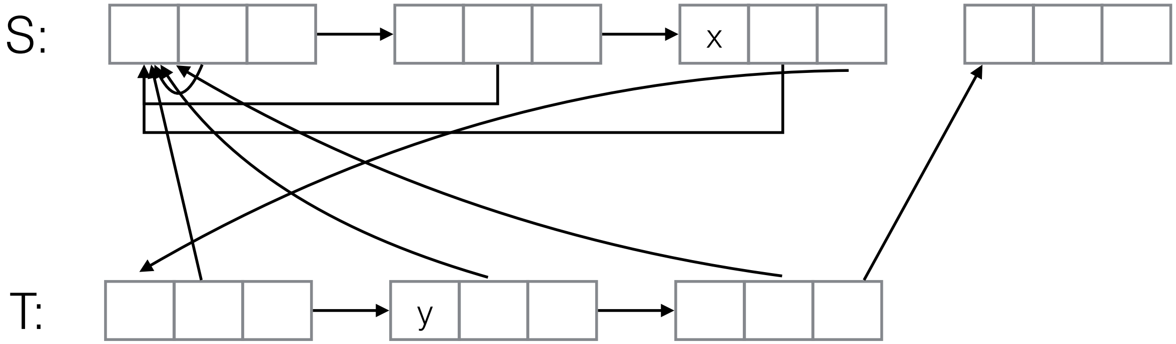


Running time:

Improved Union(x, y)

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

- Splice T into S right after x

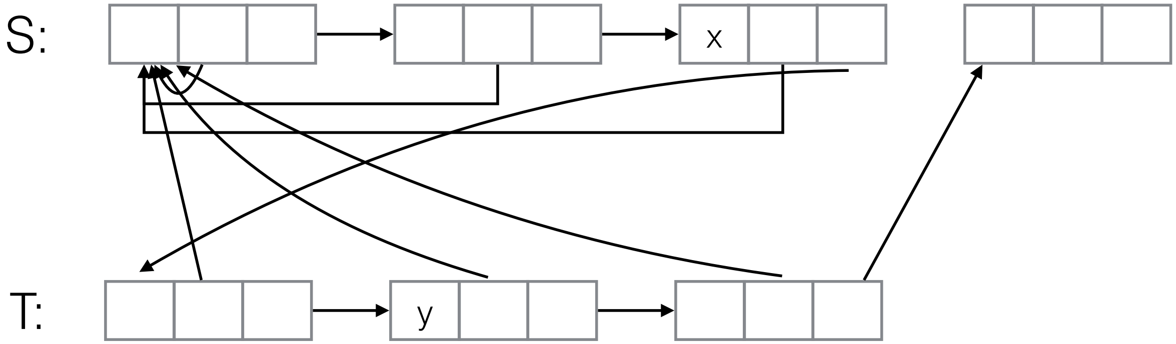


Running time: $O(|T|)$

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(x, y)

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

- Time $O(|S|)$

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

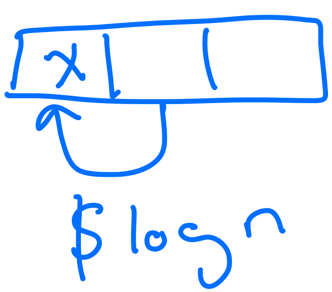
The total running time is $O(m + n \log n)$.

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.

MakeSet(x) :



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 $\mathbf{0}$ (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 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.

□

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: $\log n$

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

$O(n \log n + m)$

Find:

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

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

$O(m \log^* n)$

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

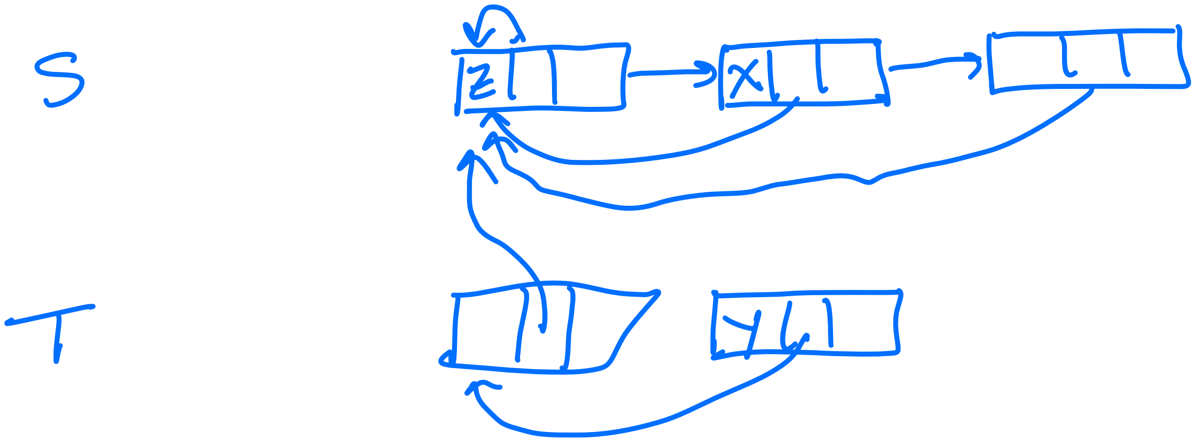
Starting idea: want to make Unions faster, willing to make Finds a little slower.

- 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

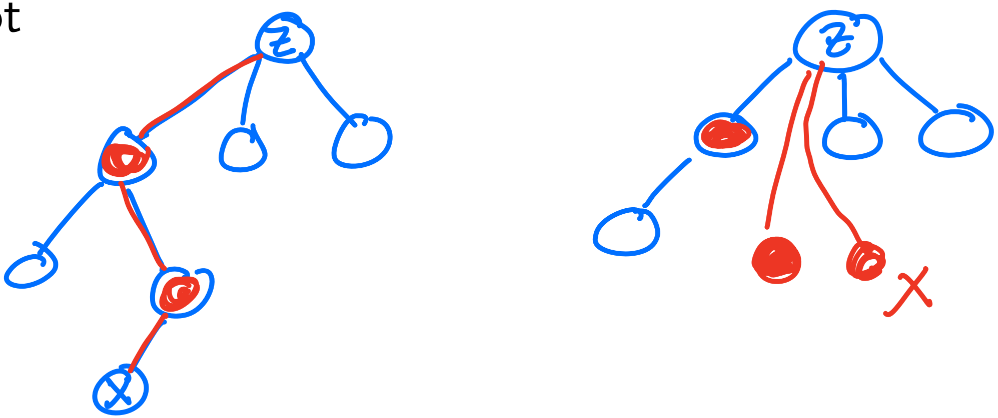
Even Better

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($\boldsymbol{x}$), 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 ≤ 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.

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(x): Set $x \rightarrow rank = 0$ and $x \rightarrow parent = x$

- Running time: $O(1)$.

Formal Procedures: Make-Set and Find

Make-Set(x): Set $x \rightarrow rank = 0$ and $x \rightarrow parent = x$

- 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 $x \rightarrow parent = x$ then return x
- $x \rightarrow parent = Find(x \rightarrow parent)$
- Return $x \rightarrow parent$

Formal Procedures: Make-Set and Find

Make-Set(x): Set $x \rightarrow rank = 0$ and $x \rightarrow parent = x$

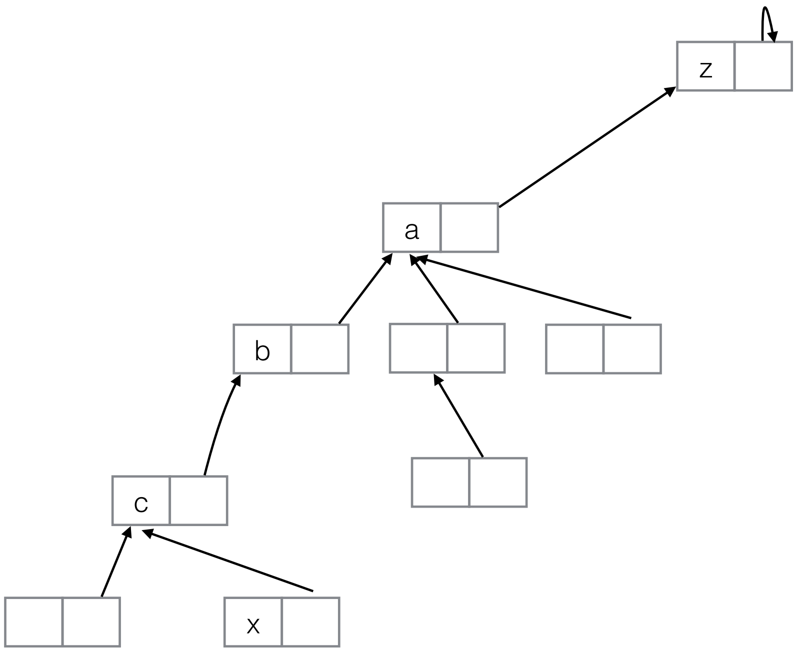
- Running time: $O(1)$.

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

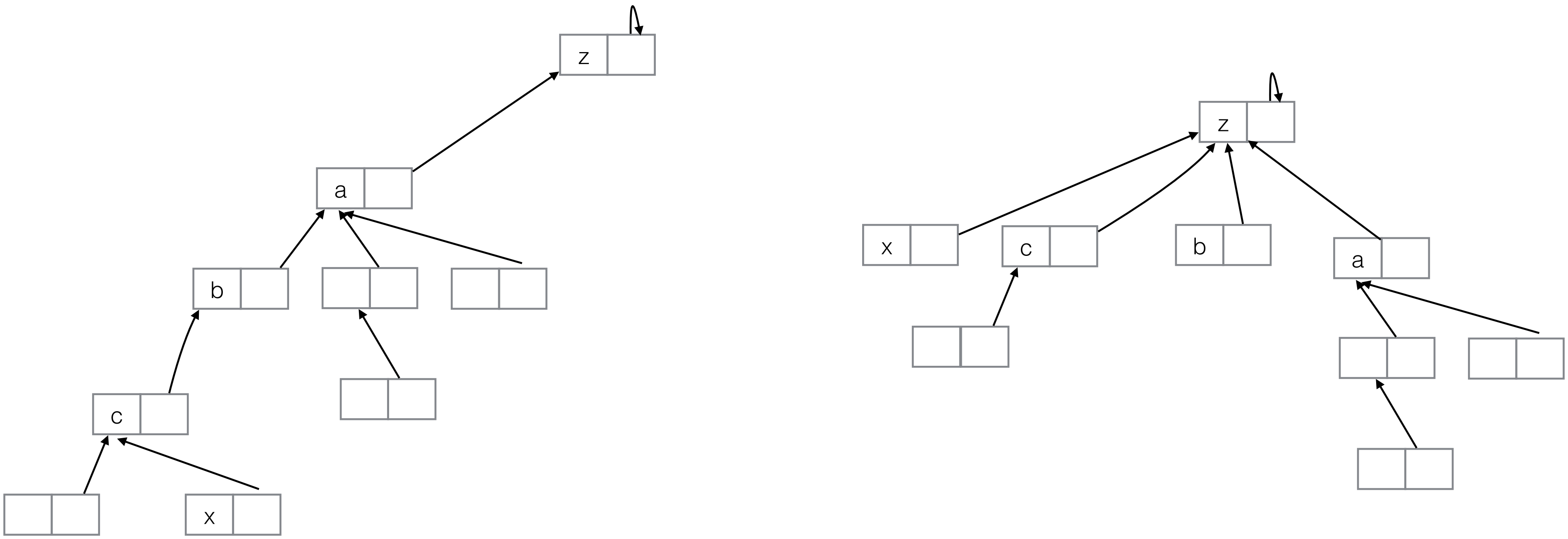
- If $x \rightarrow parent = x$ then return x
- $x \rightarrow parent = Find(x \rightarrow parent)$
- Return $x \rightarrow parent$

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

Find example



Find example



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 $r_1 \to \textit{\textbf{rank}}$.

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: $O(1)$

Formal Procedure: Union

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

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

Running time of Link: $O(1)$

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

Formal Procedure: Union

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

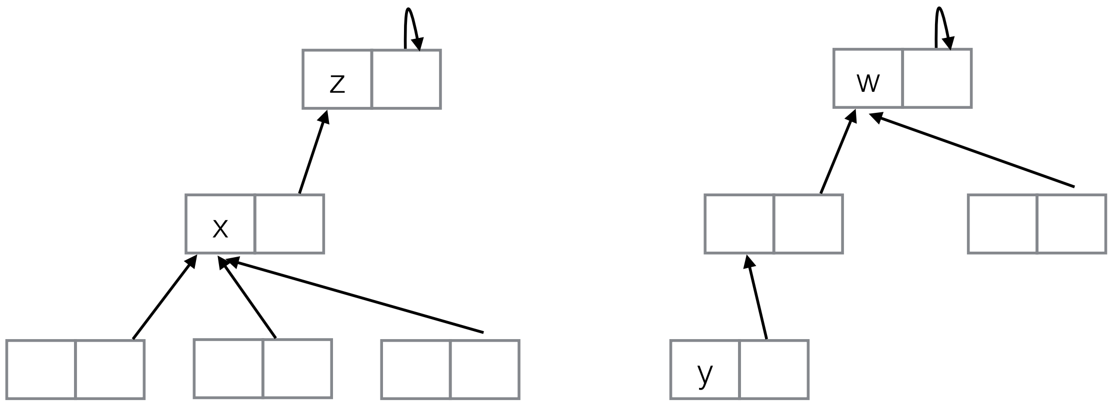
- If $r_1 \to rank > r_2 \to rank$, set $r_2 \to parent = r_1$
- If $r_2 \to rank > r_1 \to rank$, set $r_1 \to parent = r_2$
- If $r_1 \to rank = r_2 \to rank$, set $r_2 \to parent = r_1$ and increment $r_1 \to rank$.

Running time of Link: $O(1)$

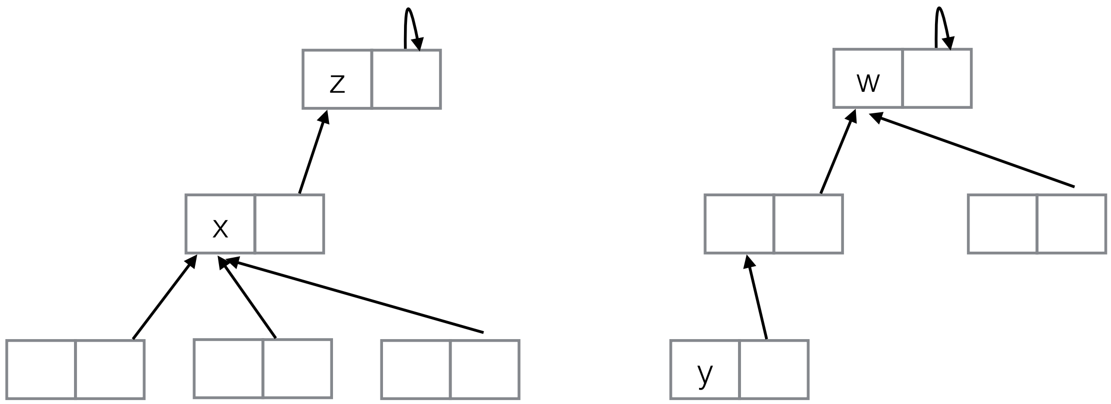
Union(x, y): Link(Find(x), Find(y))

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

Union example

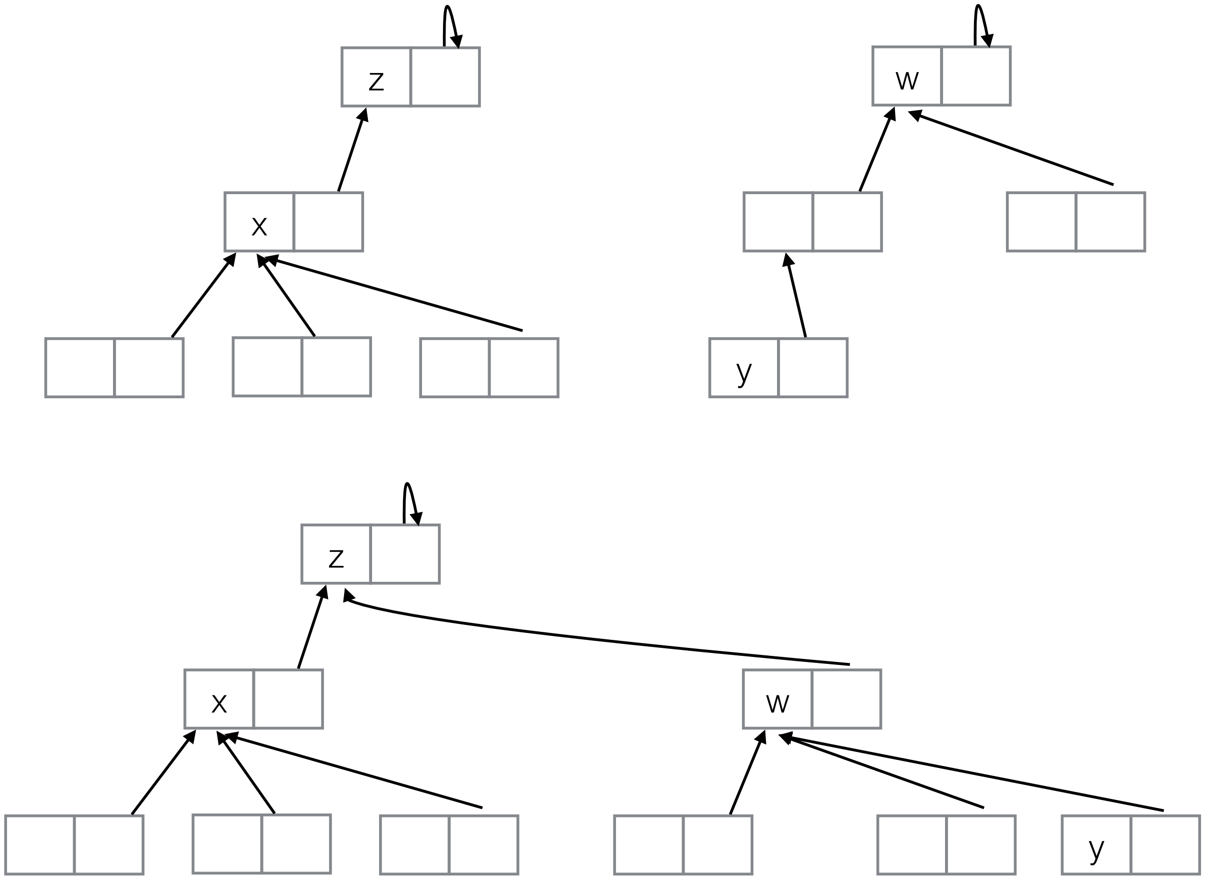


Union example



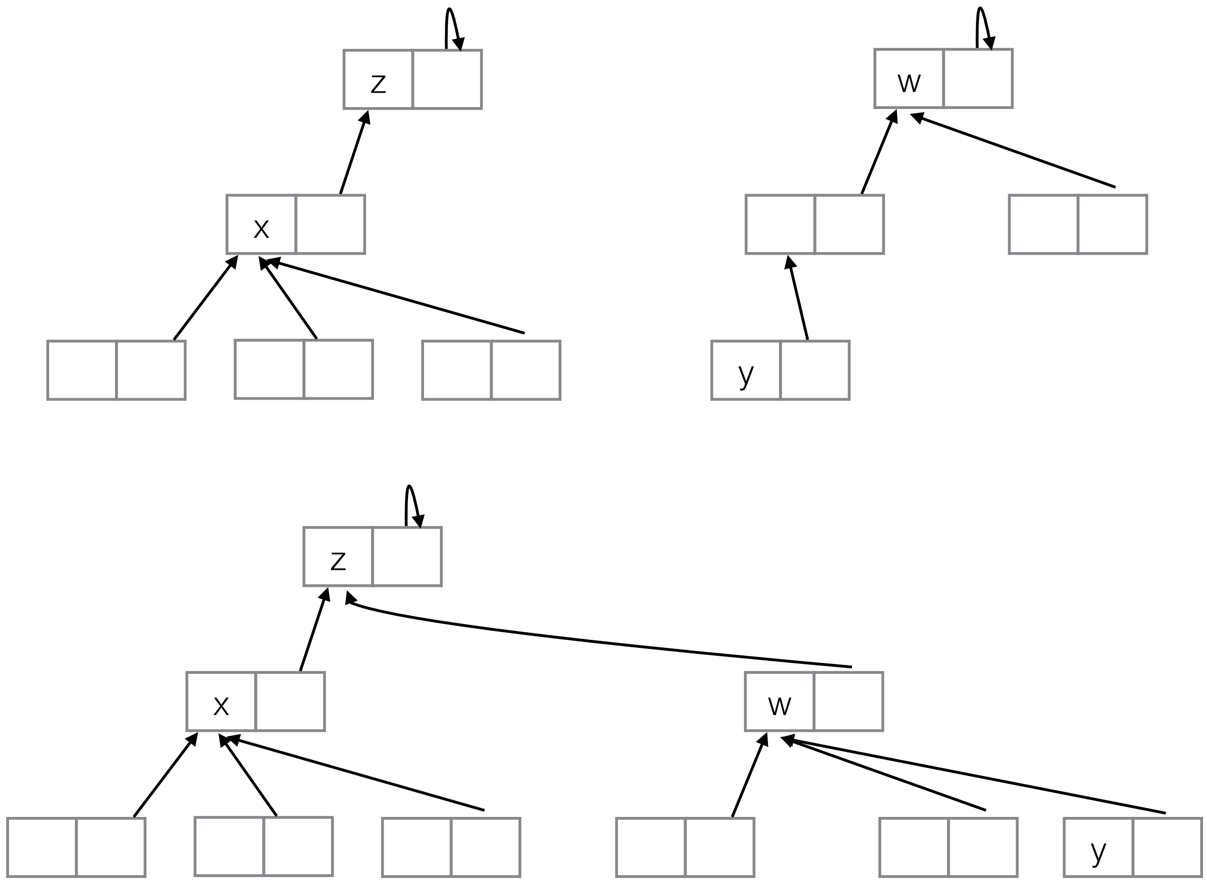
If $\boldsymbol{z \rightarrow rank \geq w \rightarrow rank}$

Union example



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

Union example



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

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

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 x not a root, then $(x \to rank) < (x \to parent \to rank)$
2. When doing path compression, if parent of x changes, new parent has rank strictly larger than old parent
3. $x \to rank$ 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.

Properties of Ranks

1. If x not a root, then $(x \to rank) < (x \to parent \to rank)$
2. When doing path compression, if parent of x changes, new parent has rank strictly larger than old parent
3. $x \to rank$ 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 $(x \to \textbf{rank}) < (x \to \textbf{parent} \to \textbf{rank})$
2. When doing path compression, if parent of x changes, new parent has rank strictly larger than old parent
3. $x \to \textbf{rank}$ 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 $(x \to rank) < (x \to parent \to rank)$
2. When doing path compression, if parent of x changes, new parent has rank strictly larger than old parent
3. $x \to rank$ 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 $r - 1$.

Properties of Ranks

1. If x not a root, then $(x \rightarrow rank) < (x \rightarrow parent \rightarrow rank)$
2. When doing path compression, if parent of x changes, new parent has rank strictly larger than old parent
3. $x \rightarrow rank$ 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 x not a root, then $(x \to rank) < (x \to parent \to rank)$
2. When doing path compression, if parent of x changes, new parent has rank strictly larger than old parent
3. $x \to rank$ 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

Properties of Ranks

1. If x not a root, then $(x \to rank) < (x \to parent \to rank)$
2. When doing path compression, if parent of x changes, new parent has rank strictly larger than old parent
3. $x \to rank$ 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 $r - 1$.

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

$\implies$ 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 $\geq r$. □

Main Result I

Theorem

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

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

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

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

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

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

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

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

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

- 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 $\mathbf{\log^* \boldsymbol{n}}$ 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 $\leq 2m$ 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

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

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

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