Lecture 9: Disjoint Sets / Union-Find

Michael Dinitz

September 29, 2020
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 **∪ **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, \(\text{Find}(x) = \text{Find}(y)\)
- If \(x\) and \(y\) are in different sets, then \(\text{Find}(x) \neq \text{Find}(y)\)
- **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
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!
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)

```
S:
```

```
Make-Set(x): Find(x): return x → head
```
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: [\[\] \[\] x z]
```

```
Make-Set(x):
```

```
x 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):

Find(x): return x → head
Union($x, y$)

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

Running time: $O(S + T)$

- Walk down $S$ to final element
- Walk down $T$ resetting head pointers

Since $S$ and $T$ could be $O(n)$, can only say $O(n)$ for Unions
Union($x, y$)

Running time: $O(S + T)$

- To walk down $S$ to final element
- To walk down $T$ and reset head pointers

Since $S$, $T$ could be $\mathcal{O}(n)$, can only say $O(n)$ for Unions
Union($x, y$)

Running time:

$O(S + T)$

Since $S$, $T$ could be $\mathcal{O}(n)$, can only say $O(n)$ for Unions.
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 $x$

Running time: $O(T)$

Still can’t say anything better than $O(n)$
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 \(x\)

![Diagram](image)

Running time: \(O(|T|)\)

- Still can’t say anything better than \(O(n)\)
Even more improved $\text{Union}(x, y)$

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 Make-Set, 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 Make-Set, 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 Make-Set, 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 Make-Set, 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 Make-Set, 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 Make-Set, 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: \(\log n\)

\[\Rightarrow \text{Amortized cost: } O(1) + O(\log n) = O(\log n)\]

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

\[\Rightarrow \text{Amortized cost: } O(1) + 0 = O(1)\]

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

\[\Rightarrow \text{Amortized cost: } \min(|S|, |T|) - \min(|S|, |T|) = 0 = O(1)\]
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(*x*), change pointers of *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

$\log^*$: iterated $\log_2$. Stronger theorem: total time at most $O(m \log^* n)$. $\Rightarrow$ inverse Ackermann function. Grows even slower than $\log^*$. See CLRS for details.
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

<table>
<thead>
<tr>
<th>Theorem</th>
</tr>
</thead>
<tbody>
<tr>
<td>When using Path Compression and Union By Rank, total time at most ( O(m \log^* n) ).</td>
</tr>
</tbody>
</table>

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

- $\log^* n = \#\text{ times apply } \log_2 \text{ until get to 1}$
- $\log^*(2^{65536}) = 1 + \log^*(65536) = 2 + \log^*(16) = 3 + \log^*(4) = 4 + \log^*(2) = 5$
- 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(x): Set \( x \rightarrow \text{rank} = 0 \) and \( x \rightarrow \text{parent} = x \)
- 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 \( x \rightarrow \text{parent} = x \) then return \( x \).
- Return \( x \rightarrow \text{parent} \).
Formal Procedures: Make-Set and Find

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

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

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

- If $\mathbf{x} \rightarrow \text{parent} = \mathbf{x}$ then return $\mathbf{x}$
- $\mathbf{x} \rightarrow \text{parent} = \text{Find}(\mathbf{x} \rightarrow \text{parent})$
- Return $\mathbf{x} \rightarrow \text{parent}$
Formal Procedures: Make-Set and Find

Make-Set(x): Set $x \rightarrow \text{rank} = 0$ and $x \rightarrow \text{parent} = x$
  - 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 $x \rightarrow \text{parent} = x$ then return x
  - $x \rightarrow \text{parent} = \text{Find}(x \rightarrow \text{parent})$
  - Return $x \rightarrow \text{parent}$

Running time of Find: depth of x (distance to root)
Find example
Formal Procedure: Union

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

- If \( r_1 \rightarrow \text{rank} > r_2 \rightarrow \text{rank} \), set \( r_2 \rightarrow \text{parent} = r_1 \)
- If \( r_2 \rightarrow \text{rank} > r_1 \rightarrow \text{rank} \), set \( r_1 \rightarrow \text{parent} = r_2 \)
- If \( r_1 \rightarrow \text{rank} = r_2 \rightarrow \text{rank} \), set \( r_2 \rightarrow \text{parent} = r_1 \) and increment \( r_1 \rightarrow \text{rank} \).

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

\textbf{Union}(x, y): \text{Link} (\text{Find}(x), \text{Find}(y))

\textbf{Running time}: depth(x) + depth(y)
Formal Procedure: Union

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

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

Running time of Link: $O(1)$
Formal Procedure: Union

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

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

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

Running time of Link: $O(1)$

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

- Running time: $\text{depth}(x) + \text{depth}(y)$
Union example

If \( z \rightarrow \text{rank} \geq w \rightarrow \text{rank} \), then \( (z \rightarrow \text{rank})++ \).
Union example

If $z \rightarrow \text{rank} \geq w \rightarrow \text{rank}$, then $(z \rightarrow \text{rank})++$.
Union example

If $z \rightarrow \text{rank} \geq w \rightarrow \text{rank}$, then $(z \rightarrow \text{rank})++
Union example

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

If $z \rightarrow \text{rank} = w \rightarrow \text{rank}$, then $(z \rightarrow \text{rank})++$
Properties of Ranks

1. If \( x \) not a root, then \((x \rightarrow \text{rank}) < (x \rightarrow \text{parent} \rightarrow \text{rank})\)

2. When doing path compression, if parent of \( x \) changes, new parent has rank strictly larger than old parent

3. \( x \rightarrow \text{rank} \) can change only if \( x \) a root, and once \( x \) is a non-root it never becomes a root again.

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 \( r - 1 \).

\( \Rightarrow \) By induction, at least \( 2^{r-1} \) nodes in each tree

\( \Rightarrow \) At least \( 2^{r-1} + 2^{r-1} = 2^r \) nodes in combined tree.
Properties of Ranks

1. If \( x \) not a root, then \( (x \rightarrow \text{rank}) < (x \rightarrow \text{parent} \rightarrow \text{rank}) \)

2. When doing path compression, if parent of \( x \) changes, new parent has rank strictly larger than old parent

3. \( x \rightarrow \text{rank} \) 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 $(x \rightarrow \text{rank}) < (x \rightarrow \text{parent} \rightarrow \text{rank})$
2. When doing path compression, if parent of $x$ changes, new parent has rank strictly larger than old parent
3. $x \rightarrow \text{rank}$ 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: $r = 0$. 

\[ \text{At least } 2^0 = 1 \text{ node in tree rooted at } x. \]
Properties of Ranks

1. If \( x \) not a root, then \((x \to \text{rank}) < (x \to \text{parent} \to \text{rank})\)

2. When doing path compression, if parent of \( x \) changes, new parent has rank strictly larger than old parent

3. \( x \to \text{rank} \) 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 $(x \rightarrow \text{rank}) < (x \rightarrow \text{parent} \rightarrow \text{rank})$

2. When doing path compression, if parent of $x$ changes, new parent has rank strictly larger than old parent

3. $x \rightarrow \text{rank}$ 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 $r - 1$. ➡️

By induction, at least $2^{r-1}$ nodes in each tree ➡️
At least $2^{r-1} + 2^{r-1} = 2^r$ nodes in combined tree.
Properties of Ranks

1. If $x$ not a root, then $(x \rightarrow \text{rank}) < (x \rightarrow \text{parent} \rightarrow \text{rank})$

2. When doing path compression, if parent of $x$ changes, new parent has rank strictly larger than old parent.

3. $x \rightarrow \text{rank}$ 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 $r - 1$. 

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

$\Rightarrow$ At least $2^{r-1} + 2^{r-1} = 2^r$ nodes in combined tree.
Properties of Ranks

1. If \( x \) not a root, then \((x \rightarrow \text{rank}) < (x \rightarrow \text{parent} \rightarrow \text{rank})\)
2. When doing path compression, if parent of \( x \) changes, new parent has rank strictly larger than old parent
3. \( x \rightarrow \text{rank} \) 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: \( r = 0 \). \( \checkmark \)
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
Properties of Ranks

1. If \( x \) not a root, then \((x \rightarrow \text{rank}) < (x \rightarrow \text{parent} \rightarrow \text{rank})\)

2. When doing path compression, if parent of \( x \) changes, new parent has rank strictly larger than old parent

3. \( x \rightarrow \text{rank} \) 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 $r$. 
\[ \implies |S_x| \geq 2^r \text{ by property 4.} \]

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 $S_x \cap S_z = \emptyset$. 
\[ \implies \text{At most } n/2^r \text{ nodes of rank } \geq r. \]
Main Result I

**Theorem**

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

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 $\Rightarrow$ at most $O(m)$ parent pointers to roots.
  - So only need to worry about parent pointers to non-roots.
Main Result 1

**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: $2 \uparrow i$ denote a tower of $i$ 2’s

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

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

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

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

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

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

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 $\leq 2m$ 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

\[ \leq \log^* n \text{ buckets} \implies \leq \log^* n \text{ per Find} \implies \leq 2m \log^* n = O(m \log^* n) \text{ total} \]

Type 2: Parent pointers that do not cross buckets

\[ \text{For each } x, \text{ let } \alpha(x) = \# \text{ times follow parent point from } x \text{ to parent in same bucket, not root. Want to show } \sum_x \alpha(x) \leq O(m \log^* n). \]

\[ \text{Since } x \text{ not root when following pointers, always has same rank} \]

\[ \text{Whenever } x \text{'s pointer followed, gets new parent (path compression)} \]

\[ \implies \text{rank of parent goes up by at least } 1 \text{ (properties of rank)} \]

\[ \implies \text{happens at most } 2 \uparrow i \text{ times if } x \text{ in bucket } i \]

\[ \implies \alpha(x) \leq 2 \uparrow i. \]
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)
  \[ \implies \text{rank of parent goes up by at least } 1 \] (properties of rank)
  \[ \implies \text{happens at most } 2 \uparrow i \text{ times if } x \text{ in bucket } i \]
  \[ \implies \alpha(x) \leq 2 \uparrow i. \]

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

\[ \leq O(m \log^* n), \]