Union-Find (Disjoint Set Union)

Efficient Maintenance of Disjoint Sets

Joseph Chuang-Chieh Lin (林莊傑)

Department of Computer Science & Engineering,
National Taiwan Ocean University

Fall 2025

Reference

- Lecture Notes of CS6820 2022 (Cornell University)
- Robert Endre Tarjan: Efficiency of a Good But Not Linear Set Union Algorithm. *Journal of the ACM*. Vol. 22(2) (1975) 215–225. [DOI LINK]
- R. E. Tarjan on analyzing the “union-find” data structure [YouTube]

Outline

1. Motivation & Abstract Data Type
2. Two Key Heuristics
3. Implementation
4. Why $\alpha(n)$? A Glimpse of the Analysis

Why Union-Find?

- Many graph algorithms need to maintain a partition of elements into disjoint sets.
- Example: **Kruskal's algorithm** and **Borůvka/Sollin's algorithm** for Minimum Spanning Tree (MST)
  - Cycle detection.
  - When scanning edges in nondecreasing weight:
    Add edge (u, v) iff u and v are currently in different connected components.
- Union-Find supports this pattern in (almost) constant amortized time.

Disjoint-Set (Union-Find) ADT

Operations

- **`find(v)`**: return a canonical representative of the set containing v.
- **`union(u,v)`**: merge the two sets containing u and v.

Disjoint-Set (Union-Find) ADT

Operations

- **`find(v)`**: return a canonical representative of the set containing v.
- **`union(u,v)`**: merge the two sets containing u and v.

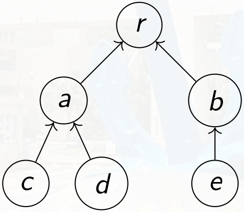
- Same-set query:

$$u \text{ and } v \text{ are in the same set} \iff \texttt{find}(u) = \texttt{find}(v).$$

- Initially: n singleton sets.

Representing Each Set as a Rooted Tree

- Each element x stores a pointer `parent[x]`.
- The root is the canonical representative (for the set).



`find(x)` = follow parent pointers to the root.

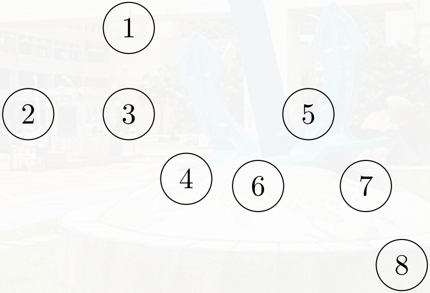
Naïve Implementation: Worst-Case Can Be Bad

- If we always attach one root under the other arbitrarily, the tree can become a chain.

`find(5)` traverses 4 edges
Worst-case: $\Theta(n)$

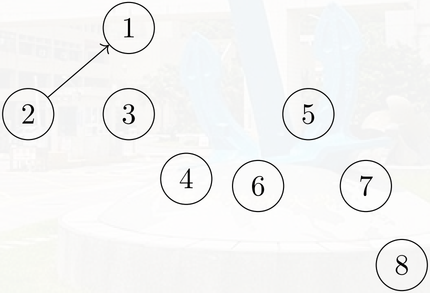
A Small Example: Unions Build a Forest

- Start with singletons $\{1\}, \{2\}, \ldots, \{8\}$.
- Perform unions: `union(1,2)`, `union(3,4)`, `union(5,6)`, `union(7,8)`, `union(1,3)`, `union(5,7)`, `union(1,5)`.



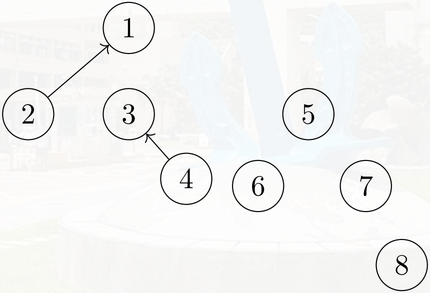
A Small Example: Unions Build a Forest

- Start with singletons $\{1\}, \{2\}, \ldots, \{8\}$.
- Perform unions: `union(1,2)`, `union(3,4)`, `union(5,6)`, `union(7,8)`, `union(1,3)`, `union(5,7)`, `union(1,5)`.



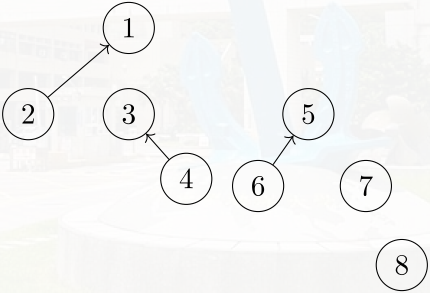
A Small Example: Unions Build a Forest

- Start with singletons $\{1\}, \{2\}, \ldots, \{8\}$.
- Perform unions: `union(1,2)`, `union(3,4)`, `union(5,6)`, `union(7,8)`, `union(1,3)`, `union(5,7)`, `union(1,5)`.



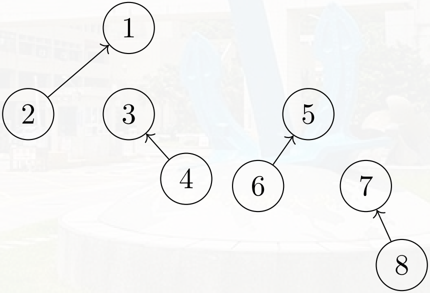
A Small Example: Unions Build a Forest

- Start with singletons $\{1\}, \{2\}, \ldots, \{8\}$.
- Perform unions: `union(1,2)`, `union(3,4)`, `union(5,6)`, `union(7,8)`, `union(1,3)`, `union(5,7)`, `union(1,5)`.



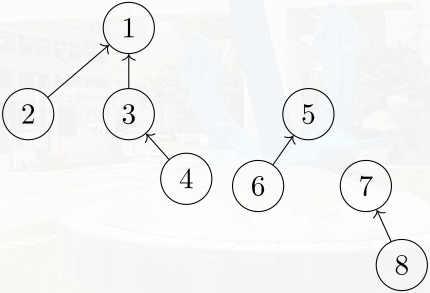
A Small Example: Unions Build a Forest

- Start with singletons $\{1\}, \{2\}, \ldots, \{8\}$.
- Perform unions: `union(1,2)`, `union(3,4)`, `union(5,6)`, `union(7,8)`, `union(1,3)`, `union(5,7)`, `union(1,5)`.



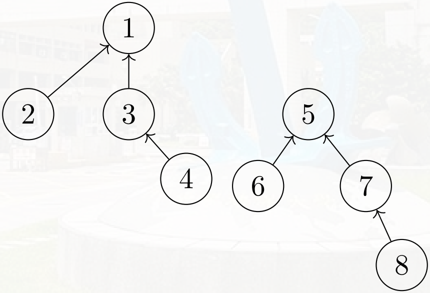
A Small Example: Unions Build a Forest

- Start with singletons $\{1\}, \{2\}, \ldots, \{8\}$.
- Perform unions: `union(1,2)`, `union(3,4)`, `union(5,6)`, `union(7,8)`, `union(1,3)`, `union(5,7)`, `union(1,5)`.



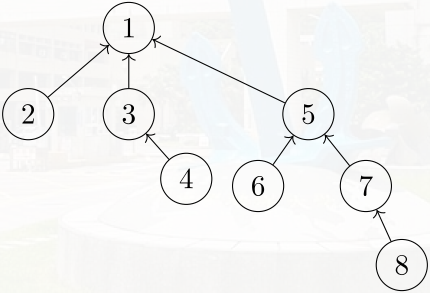
A Small Example: Unions Build a Forest

- Start with singletons $\{1\}, \{2\}, \ldots, \{8\}$.
- Perform unions: `union(1,2)`, `union(3,4)`, `union(5,6)`, `union(7,8)`, `union(1,3)`, `union(5,7)`, `union(1,5)`.



A Small Example: Unions Build a Forest

- Start with singletons $\{1\}, \{2\}, \ldots, \{8\}$.
- Perform unions: `union(1,2)`, `union(3,4)`, `union(5,6)`, `union(7,8)`, `union(1,3)`, `union(5,7)`, `union(1,5)`.



Cost Model

- `union(u,v)`:
  - Usually does two `find` operations, then one pointer update.
  - Time dominated by the two `find` calls.
- `find(u)`:
  - Time proportional to the length of the path from u to the root.

Goal

Support a sequence of m operations on n elements in **near-linear** total time.

Outline

1 Motivation & Abstract Data Type

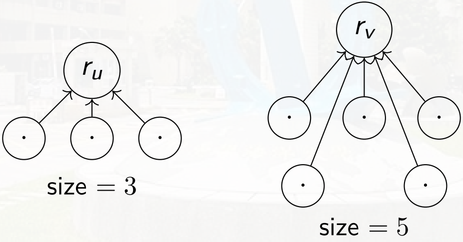
2 Two Key Heuristics

3 Implementation

4 Why $\alpha(n)$? A Glimpse of the Analysis

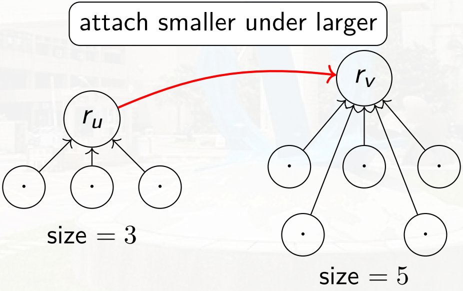
Heuristic #1: Union by Size (Smaller $\rightarrow$ Larger)

- Maintain `size[root]` $=$ number of nodes in that tree.
- On `union(u,v)`:
  - Find roots r_u, r_v.
  - Make the root of the smaller tree point to the root of the larger tree.



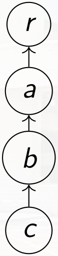
Heuristic #1: Union by Size (Smaller → Larger)

- Maintain `size[root]` = number of nodes in that tree.
- On `union(u,v)`:
  - Find roots r_u, r_v.
  - Make the root of the smaller tree point to the root of the larger tree.



Heuristic #2: Path Compression (During `find`)

- After finding the root r for a query node x:
  - traverse the path again and set every visited node's parent directly to r.
- This makes future `find` operations faster.

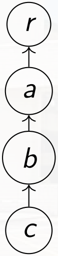


`find(c)` follows 3 edges

Before

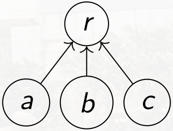
Heuristic #2: Path Compression (During `find`)

- After finding the root r for a query node x:
  - traverse the path again and set every visited node's parent directly to r.
- This makes future `find` operations faster.



`find(c)` follows 3 edges

Before



path compressed

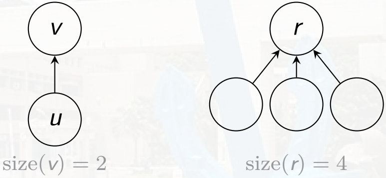
After

Key Observations (w/ vs. w/o Path Compression)

- Consider a fixed sequence σ of m operations on n elements.
- With or without path compression:
  - the partition into sets at each time is the same,
  - the roots (representatives) are the same.
- The difference: path compression can later make a node a non-descendant of a former ancestor.

Example

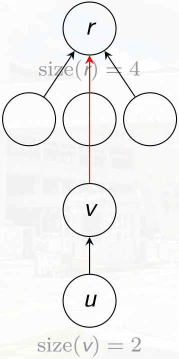
Time t: after `union(u,v)` (and r already has a larger set)



Now u is a descendant of v.

Example

Later: after `union(v,r)` by size (no compression yet)

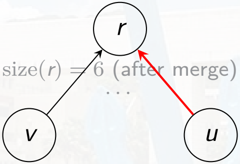


since $2 < 4$, attach v under r

Path is $u \to v \to r$, so u is still a descendant of v.

Example

After `find(u)` with path compression



u now bypasses v

Now u is *not* a descendant of v (even though it was earlier).

What We Get from These Two Heuristics

Performance Guarantee (informal)

A sequence of m `union`/`find` operations takes

$$O\big((m+n)\,\alpha(n)\big)$$

time, where $\alpha(n)$ is the inverse Ackermann function.

What We Get from These Two Heuristics

Performance Guarantee (informal)

A sequence of m `union`/`find` operations takes

$$O\big((m+n)\,\alpha(n)\big)$$

time, where $\alpha(n)$ is the inverse Ackermann function.

- $\alpha(n)$ grows extremely slowly that it can be effectively viewed as a constant for all realistic inputs.

What We Get from These Two Heuristics

Performance Guarantee (informal)

A sequence of m `union`/`find` operations takes

$$O\big((m+n)\,\alpha(n)\big)$$

time, where $\alpha(n)$ is the inverse Ackermann function.

- $\alpha(n)$ grows extremely slowly that it can be effectively viewed as a constant for all realistic inputs.
- Why and how does this exotic function appear in the analysis?

Implementation: Two Arrays

- `parent[x]`: parent pointer; roots satisfy `parent[r] = r`.
- `rank`[x] (a kind of “weight” of x): maintained only for roots.

Invariants

- Each set is represented by exactly one rooted tree.
- Roots are the canonical representatives.

C++ style Reference Implementation

```
struct DSU {
  vector<int> parent, rank;
  DSU(int n=0) { init(n); }
  void init(int n){
    parent.resize(n);
    rank.assign(n,1);
    for(int i=0; i<n; i++) parent[i]=i;
  }
  int find(int x) {
    if(parent[x] == x) return x;
    return parent[x] = find(parent[x]); // path compression
  }
  bool union(int a,int b){ // weighted union
    a = find(a); b = find(b);
    if(a == b) return false;
    if(rank[a] < rank[b]) swap(a,b);  // union by size
    parent[b] = a;
    rank[a] += rank[b];
    return true;
  }
};
```

Practical Notes

- Union-by-rank is a common variant:
  - maintain an upper bound on height instead of exact subtree size,
  - update only when ranks tie.
- **Rule of thumb**: always combine one of (`union by size/rank`) **and** `path compression`.

Outline

1 Motivation & Abstract Data Type

2 Two Key Heuristics

3 Implementation

4 Why $\alpha(n)$? A Glimpse of the Analysis

Ackermann-Type Growth

- The analysis uses a hierarchy of rapidly growing functions $\{A_k\}$.
- One convenient definition:

$$A_0(x) = x + 1, \qquad A_{k+1}(x) = A_k^{(x)}(x),$$

where $A_k^{(i)} := \underbrace{A_k \circ A_k \cdots \circ A_k}_{i \text{ times}}$ and $A_k^0 :=$ identity function.

Ackermann-Type Growth

- The analysis uses a hierarchy of rapidly growing functions $\{A_k\}$.
- One convenient definition:

$$A_0(x) = x + 1, \qquad A_{k+1}(x) = A_k^{(x)}(x),$$

where $A_k^{(i)} := \underbrace{A_k \circ A_k \cdots \circ A_k}_{i \text{ times}}$ and $A_k^0 :=$ identity function.

Intuition

As k increases, $A_k(x)$ grows astronomically fast (much faster than exponentials/towers).

Concrete Examples (Small k)

For $x \geq 2$, the first few levels behave like:

$$
\begin{aligned}
A_0(x) &= x + 1, \\
A_1(x) &= 2x, \\
A_2(x) &= x \cdot 2^x \quad \text{(roughly exponential)}, \\
A_3(x) &= A_2^x(x) \geq \underbrace{2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}}_{x} \text{(a tower of 2s of height } x\text{)}.
\end{aligned}
$$

- $A_4(2) \geq \underbrace{2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}}_{2048}$.
- This is why the inverse function grows extremely slowly.

Ackermann Function and Its Inverse

Definitions [Ackermann and the Inverse Ackermann]

$$A(k) := A_k(2), \qquad \alpha(n) = \min\{k \mid A(k) \geq n\}.$$

For example, $\alpha(2048) = 3$ ($\because A_3(2) = 2048$), $\alpha(2^{65536}) = 4$.

Ackermann Function and Its Inverse

Definitions [Ackermann and the Inverse Ackermann]

$$A(k) := A_k(2), \qquad \alpha(n) = \min\{k \mid A(k) \geq n\}.$$

For example, $\alpha(2048) = 3$ ($\because A_3(2) = 2048$), $\alpha(2^{65536}) = 4$.

- $\alpha(n)$ is so small that in practice it behaves like a constant.
- The theoretical bound for Union-Find becomes **near-linear**.

High-Level Statement of the Main Result

- Let σ be a sequence of m `union` and `find` operations.

Theorem (Tarjan [JACM 1975])

Starting from n singleton sets, any sequence of m `union` and `find` operations takes

$$O\big((m+n)\,\alpha(n)\big)$$

time when using union-by-rank and path compression.

High-Level Statement of the Main Result

- Let σ be a sequence of m `union` and `find` operations.

Theorem (Tarjan [JACM 1975])

Starting from n singleton sets, any sequence of m `union` and `find` operations takes

$$O\big((m+n)\,\alpha(n)\big)$$

time when using union-by-rank and path compression.

- Next: the proof idea uses ranks and an amortized charging argument.

Rank of a Node

- Consider executing the same operation sequence without path compression.
- Let $T_m(u)$ be the subtree rooted at u in the final forest.
  - $T_t(u)$: the subtree rooted at u at time t in the execution of a sequence of m `union` and `find` instructions **without** path compressions.

Definition (Rank): a quantity that survives path compression

$$\mathrm{rank}(u) = 2 + \mathrm{height}\left(T_m(u)\right).$$

$\mathrm{height}(T)$: the length of the longest path from leaves to the root.

Rank of a Node

- Consider executing the same operation sequence without path compression.
- Let $T_m(u)$ be the subtree rooted at u in the final forest.
  - $T_t(u)$: the subtree rooted at u at time t in the execution of a sequence of m `union` and `find` instructions **without** path compressions.

Definition (Rank): a quantity that survives path compression

$$\mathrm{rank}(u) = 2 + \mathrm{height}\left(T_m(u)\right).$$

$\mathrm{height}(T)$: the length of the longest path from leaves to the root.

- With union-by-rank, ranks are bounded and monotone along parent pointers.

A Key Balancing Lemma (Union-by-Rank)

Lemma (Size vs. Height)

If we always merge the smaller tree into the larger, then for any time t and root u,

$$|T_t(u)| \geq 2^{\mathrm{height}(T_t(u))}.$$

A Key Balancing Lemma (Union-by-Rank)

Lemma (Size vs. Height)

If we always merge the smaller tree into the larger, then for any time t and root u,

$$|T_t(u)| \geq 2^{\mathrm{height}(T_t(u))}.$$

- Prove by induction on t.

A Key Balancing Lemma (Union-by-Rank)

Lemma (Size vs. Height)

If we always merge the smaller tree into the larger, then for any time t and root u,

$$|T_t(u)| \geq 2^{\mathrm{height}(T_t(u))}.$$

- Prove by induction on t.
- Intuition: each time the height increases by 1, the subtree size at least doubles.

A Key Balancing Lemma (Union-by-Rank)

Lemma (Size vs. Height)

If we always merge the smaller tree into the larger, then for any time t and root u,

$$|T_t(u)| \geq 2^{\mathrm{height}(T_t(u))}.$$

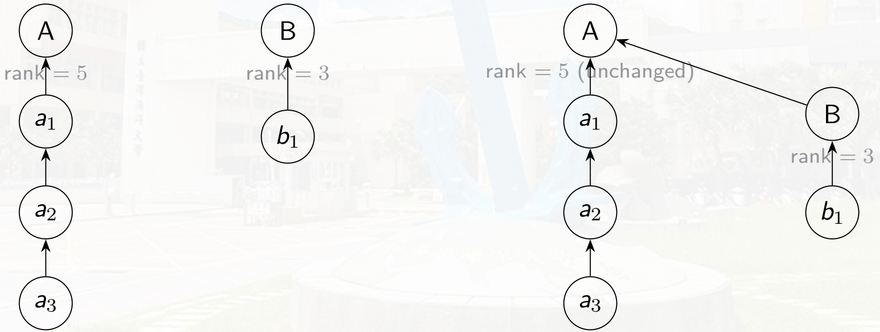
- Prove by induction on t.
- Intuition: each time the height increases by 1, the subtree size at least doubles.
- Consequence: the maximum rank is $\lfloor \lg n \rfloor + 2 = O(\log n)$.

Note: Effect on the rank after union operation (1/2)

Example 1: unequal heights (no rank increase)

Before union

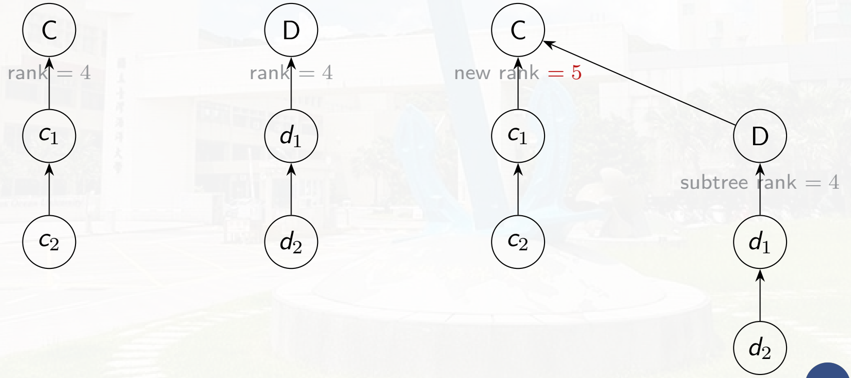
After union



Note: Effect on the rank after union operation (2/2)

Example 2: equal heights (rank increases)

Before union　　　　　　After union



From Ranks to $\alpha(n)$ (Rough Sketch (1/2))

- Path compression changes parents, so we need to track how fast parent ranks can grow.
- Define levels using the Ackermann hierarchy:

$$\ell(u) = \max\left\{k \;\middle|\; \mathrm{rank}(\mathrm{parent}(u)) \geq A_k(\mathrm{rank}(u))\right\}.$$

- During path compression, $\ell(u)$ can only increase.

From Ranks to $\alpha(n)$ (Rough Sketch (1/2))

- Path compression changes parents, so we need to track how fast parent ranks can grow.
- Define levels using the Ackermann hierarchy:

$$\ell(u) = \max\left\{ k \;\middle|\; \mathrm{rank}(\mathrm{parent}(u)) \geq A_k(\mathrm{rank}(u)) \right\}.$$

- During path compression, $\ell(u)$ can only increase.
    - Path compression changes $\mathrm{parent}(u)$ to a higher ancestor (larger rank), so $\ell(u)$ is monotone.

From Ranks to $\alpha(n)$ (Rough Sketch (1/2))

- Path compression changes parents, so we need to track how fast parent ranks can grow.
- Define levels using the Ackermann hierarchy:

$$\ell(u) = \max\left\{ k \;\middle|\; \mathrm{rank}(\mathrm{parent}(u)) \geq A_k(\mathrm{rank}(u)) \right\}.$$

- During path compression, $\ell(u)$ can only increase.
  - Path compression changes $\mathrm{parent}(u)$ to a higher ancestor (larger rank), so $\ell(u)$ is monotone.

From Ranks to $\alpha(n)$ (Rough Sketch (1/2))

- Path compression changes parents, so we need to track how fast parent ranks can grow.
- Define levels using the Ackermann hierarchy:

$$\ell(u) = \max\Big\{k \;\Big|\; \mathrm{rank}(\mathrm{parent}(u)) \geq A_k(\mathrm{rank}(u))\Big\}.$$

- During path compression, $\ell(u)$ can only increase.
  - Path compression changes $\mathrm{parent}(u)$ to a higher ancestor (larger rank), so $\ell(u)$ is monotone.

From Ranks to $\alpha(n)$ (Rough Sketch (2/2))

$$\ell(u) = \max\left\{k \;\middle|\; \mathrm{rank}(\mathrm{parent}(u)) \geq A_k(\mathrm{rank}(u))\right\}.$$

Charging idea (just bookkeeping for the analysis, not in the code)

When a `find` traverses a node u, we charge its $O(1)$ cost as follows:

- **Node-charge (same-level step):** if the traversal is "same-level" for u (e.g., $\ell(\mathrm{parent}(u)) = \ell(u)$), then we charge it to the node u.
- **Find-charge (level-increasing step):** if the traversal is "level-increasing" (e.g., $\ell(\mathrm{parent}(u)) > \ell(u)$), then we charge it to the `find` itself.

From Ranks to $\alpha(n)$ (Rough Sketch (2/2))

$$\ell(u) = \max\Big\{k \;\Big|\; \mathrm{rank}(\mathrm{parent}(u)) \ge A_k(\mathrm{rank}(u))\Big\}.$$

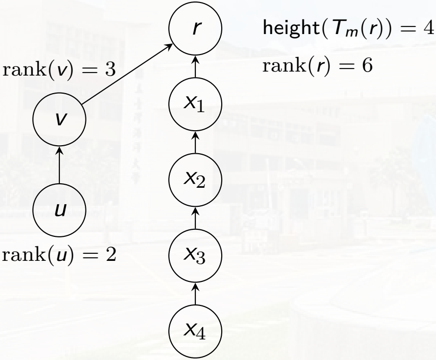
Charging idea (just bookkeeping for the analysis, not in the code)

When a `find` traverses a node u, we charge its $O(1)$ cost as follows:

- **Node-charge (same-level step):** if the traversal is "same-level" for u (e.g., $\ell(\mathrm{parent}(u)) = \ell(u)$), then we charge it to the node u.
  - After compression, $\mathrm{rank}(\mathrm{parent}(u))$ strictly increases, so u makes progress (either $\ell(u)$ increases, or it advances within the same level via the auxiliary index).
- **Find-charge (level-increasing step):** if the traversal is "level-increasing" (e.g., $\ell(\mathrm{parent}(u)) > \ell(u)$), then we charge it to the `find` itself.

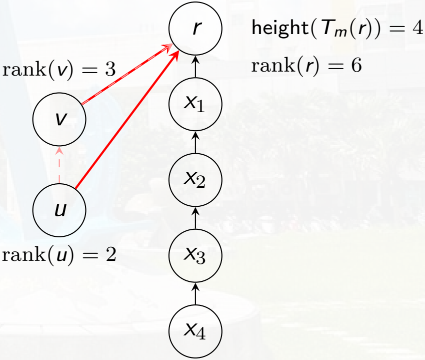
Example

Before path compression



$\text{rank}(\text{parent}(u)) = \text{rank}(v) = 3$

After `find(u)` with compression



$\text{rank}(\text{parent}(u)) = \text{rank}(r) = 6$

Key distinction: what we *maintain* vs. what we *analyze*

- **What we maintain (in the data structure):**
  - We store a rank value that is updated *only* by `union` (and *never* recomputed during `find`).
  - Hence ranks evolve as if path compression did not exist. They remain valid monotone labels that reflect the `union`-history.
- **What we analyze (running time):**
  - We analyze the *actual* execution with path compression.
  - Ignoring compression gives a valid but loose upper bound:

$$\texttt{find} = O(\log n), \qquad \text{total} = O(m \log n).$$

  - The sharper bound $O\big((m+n)\alpha(n)\big)$ requires using the effect of compression:
    - Parent pointers $\rightarrow$ higher-rank ancestors;
    - and the ranks serve as a progress measure in the amortized analysis.

Remark (connections to the Ackermann)

- If u has a parent, then

$$\mathrm{rank}(\mathrm{parent}(u)) \geq \mathrm{rank}(u) + 1 = A_0(\mathrm{rank}(u)).$$

- For $n \geq 5$, the maximum value $\ell(u)$ can take on is $\alpha(n) - 1$, since for $\ell(u) = k$,

$$\begin{aligned} n &> \lfloor \lg n \rfloor + 2 \\ &\geq \mathrm{rank}(\mathrm{parent}(u)) \\ &\geq A_k(\mathrm{rank}(u)) \\ &\geq A_k(2), \end{aligned}$$

therefore, $\alpha(n) > k = \ell(u)$.

Remark: levels $\leftrightarrow$ ranks

- Because path compression changes parents, we track how large the parent's rank is compared to the node's rank on an Ackermann scale.

Level

$$\ell(u) = \max\left\{ k \;\middle|\; \mathrm{rank}(\mathrm{parent}(u)) \;\geq\; A_k(\mathrm{rank}(u)) \right\}.$$

- **Intuition:** $\ell(u)$ is the number of "Ackermann jumps" by which the parent rank dominates the child rank.

Observation of Monotonicity: Levels only go up

- During a `find`, path compression sets $\mathrm{parent}(u)$ to an ancestor (often the root), whose rank is at least the old parent's rank.

Observation of Monotonicity: Levels only go up

- During a `find`, path compression sets $\mathrm{parent}(u)$ to an ancestor (often the root), whose rank is at least the old parent's rank.
- Therefore $\mathrm{rank}(\mathrm{parent}(u))$ is nondecreasing over time.

Observation of Monotonicity: Levels only go up

- During a `find`, path compression sets $\text{parent}(u)$ to an ancestor (often the root), whose rank is at least the old parent's rank.
- Therefore $\text{rank}(\text{parent}(u))$ is nondecreasing over time.
- Since $\ell(u)$ depends on whether $\text{rank}(\text{parent}(u))$ crosses thresholds $A_k(\text{rank}(u))$, we get:

Level monotonicity

$\ell(u)$ never decreases as operations proceed.

Observation of Monotonicity: Levels only go up

- During a `find`, path compression sets $\mathrm{parent}(u)$ to an ancestor (often the root), whose rank is at least the old parent's rank.
- Therefore $\mathrm{rank}(\mathrm{parent}(u))$ is nondecreasing over time.
- Since $\ell(u)$ depends on whether $\mathrm{rank}(\mathrm{parent}(u))$ crosses thresholds $A_k(\mathrm{rank}(u))$, we get:

Level monotonicity

$\ell(u)$ never decreases as operations proceed.

- Consequence: each node's level can increase at most $\alpha(n)$ times.

Charging scheme for the cost of `find`

Consider one `find(x)` that traverses nodes on the path $x = u_0, u_1, \ldots, u_t = \text{root}$.

$$\text{Total time} \;\le\; O\big(\underbrace{\#\text{node-charges}}_{\le n\alpha(n)} + \underbrace{\#\text{operation-charges}}_{\le m\alpha(n)}\big).$$

Bounding operation-charges: $\leq m\alpha(n)$

- An operation-charge happens only when $\ell(\mathrm{parent}(u)) > \ell(u)$.
- But $\ell(u)$ is monotone and bounded:

$$0 \leq \ell(u) \leq \alpha(n).$$

- Hence each `find` can be charged at most $\alpha(n)$ times.

Conclusion

$\#$operation-charges $\leq m\,\alpha(n)$.

Bounding node-charges

Node-charges are assigned to "same-level" traversal steps (e.g., $\ell(\mathrm{parent}(u)) = \ell(u)$ at the time the edge is traversed).

- Fix a node u that is visited by `find`.
- After compression, $\mathrm{rank}(\mathrm{parent}(u))$ strictly increases, so u makes progress.
- Important: $\ell(u)$ may stay the same;
  - We will keep track a refined state of the progress of u.
- This state can advance at most $O(\alpha(n))$ times per node.

Claim

#node-charges in all finds $= O(n\alpha(n))$.

On the number of charges to the node

To formalize the previous slide, analyses often add an index within a level.

Index within level $k = \ell(u)$

Let $A_k^{(j)}$ be the j-fold iterate of A_k. Define

$$i(u) := \max\left\{ j \;\middle|\; \mathrm{rank}(\mathrm{parent}(u)) \geq A_k^{(j)}(\mathrm{rank}(u)) \right\}.$$

- **Note:** larger $i(u)$ means the parent's rank is not just above A_k but above many repeated applications of A_k.
- If $\ell(u)$ does not increase, path compression still tends to increase $\mathrm{rank}(\mathrm{parent}(u))$, so $i(u)$ increases.

Wait! Why do we iterate A_k?

- By definition of ℓ, at time t we have the level-k threshold:

$$\mathrm{rank}(\mathrm{parent}(x)) \;\geq\; A_k(\mathrm{rank}(x)).$$

Wait! Why do we iterate A_k?

- By definition of ℓ, at time t we have the level-k threshold:

$$\mathrm{rank}(\mathrm{parent}(x)) \;\geq\; A_k(\mathrm{rank}(x)).$$

- Path compression changes `parent` pointers (e.g., $x \to v$, $v = \mathrm{parent}(x)$), so

$$\mathrm{rank}(\mathrm{parent}(x))$$

  may *increase over time* even if $\ell(x)$ (the level) stays the same.

Wait! Why do we iterate A_k?

- By definition of ℓ, at time t we have the level-k threshold:

$$\mathrm{rank}(\mathrm{parent}(x)) \;\geq\; A_k(\mathrm{rank}(x)).$$

- Path compression changes `parent` pointers (e.g., $x \to v$, $v = \mathrm{parent}(x)$), so

$$\mathrm{rank}(\mathrm{parent}(x))$$

  may *increase over time* even if $\ell(x)$ (the level) stays the same.
- Therefore, we need a *finer progress measure within level k*.

Index within a fixed level: Using iterates A_k^i

- Consider an *iterate index* $i \geq 1$: $\mathrm{rank}(\mathrm{parent}(x)) \geq A_k^i(\mathrm{rank}(x))$, where A_k^i denotes the i-fold iterate of A_k.
- It is an *analytic ruler* that counts how far $\mathrm{rank}(\mathrm{parent}(x))$ has advanced *within the same level* k.
- If a charge to x occurs at time t, then after compression (time $t+1$) the new parent becomes a later vertex v on the find path, and

$$\mathrm{rank}(v) \geq A_k\big(\mathrm{rank}(\mathrm{parent}(x))\big) \geq A_k\big(A_k^i(\mathrm{rank}(x))\big) = A_k^{i+1}(\mathrm{rank}(x)).$$

- Since v is the new parent of x, this implies

$$\mathrm{rank}(\mathrm{parent}(x)) \geq A_k^{i+1}(\mathrm{rank}(x)).$$

- Thus, each such event increases the *index* i by at least 1.

i is bounded before the level must increase

- While $\ell(x) = k$ remains fixed, the index i can increase only finitely many times.
- After at most $\mathrm{rank}(x)$ such increments, the notes obtain

$$\mathrm{rank}(\mathrm{parent}(x)) \;\geq\; A_k^{\mathrm{rank}(x)}(\mathrm{rank}(x)) \;=\; A_{k+1}(\mathrm{rank}(x)).$$

i is bounded before the level must increase

- While $\ell(x) = k$ remains fixed, the index i can increase only finitely many times.
- After at most $\mathrm{rank}(x)$ such increments, the notes obtain

$$\mathrm{rank}(\mathrm{parent}(x)) \;\geq\; A_k^{\mathrm{rank}(x)}(\mathrm{rank}(x)) \;=\; A_{k+1}(\mathrm{rank}(x)).$$

- Crossing the next threshold forces the level to increase:

$$\ell(x) \;\geq\; k+1 \qquad \text{(equivalently, } \ell(x) \text{ must increase).}$$

i is bounded before the level must increase

- While $\ell(x) = k$ remains fixed, the index i can increase only finitely many times.
- After at most $\mathrm{rank}(x)$ such increments, the notes obtain

$$\mathrm{rank}(\mathrm{parent}(x)) \;\geq\; A_k^{\mathrm{rank}(x)}(\mathrm{rank}(x)) \;=\; A_{k+1}(\mathrm{rank}(x)).$$

- Crossing the next threshold forces the level to increase:

$$\ell(x) \;\geq\; k+1 \qquad \text{(equivalently, } \ell(x) \text{ must increase).}$$

- Therefore, at most $\mathrm{rank}(x)\alpha(n)$ such charges against x.

On the number of nodes of the same rank

Lemma

For any integer r,

$$\left|\{\, u \mid \mathrm{rank}(u) = r \,\}\right| \;\le\; \frac{n}{2^{\,r-2}}.$$

On the number of nodes of the same rank

Lemma

For any integer r,

$$\left|\{\, u \mid \mathrm{rank}(u) = r \,\}\right| \;\le\; \frac{n}{2^{\,r-2}}.$$

Proof sketch

- If $\mathrm{rank}(u) = \mathrm{rank}(v) = r$, then the subtrees $T_m(u)$ and $T_m(v)$ are disjoint.
- Hence, we have the union of these subtrees has size $\left|\bigcup_{\mathrm{rank}(u)=r} T_m(u)\right| = \sum_{\mathrm{rank}(u)=r} |T_m(u)| \le n$.
- Also, we have known that every node of rank r satisfies $|T_m(u)| \ge 2^{\,r-2}$, so

$$\sum_{\mathrm{rank}(u)=r} |T_m(u)| \;\ge\; \sum_{\mathrm{rank}(u)=r} 2^{\,r-2} \;=\; \left|\{u : \mathrm{rank}(u) = r\}\right| \cdot 2^{\,r-2}.$$

- Rearranging yields $\left|\{u : \mathrm{rank}(u) = r\}\right| \;\le\; \dfrac{n}{2^{\,r-2}}$.

Sum up the charges against nodes

- At most $\mathrm{rank}(x)\alpha(n)$ charges against a node x. So, there are at most
$$r\alpha(n)\frac{n}{2^{r-2}} = n\alpha(n)\frac{r}{2^{r-2}}$$
charges against nodes of rank r.
- Summing over all values of r we obtain the following bound on all charges to all vertices
$$\sum_{r=0}^{\infty} n\alpha(n)\,\frac{r}{2^{\,r-2}} \;=\; n\alpha(n)\cdot\sum_{r=0}^{\infty}\frac{r}{2^{\,r-2}} \;=\; 8n\alpha(n).$$

Putting it together: near-linear time

- Node-charges: at most $n\alpha(n)$ total.
- Operation-charges: at most $m\alpha(n)$ total.

Conclusion (Tarjan)

Starting from n singletons, any sequence of m union/find operations with union-by-rank (or size) plus path compression runs in

$$O\big((m+n)\alpha(n)\big) \text{ time.}$$

- Practically, $\alpha(n) \leq 4$ for any realistic n, so it behaves like "almost constant amortized time."

Conclusion of the Amortized Analysis

- Total charge to `find` operations: $O(m\,\alpha(n))$.
- Total charge to nodes over the entire computation: $O(n\,\alpha(n))$.

Therefore

Total time for m operations is $O((m+n)\alpha(n))$.

- This is why Union-Find is considered almost linear time.