# Greedy

# What is Greedy?

Follows the “looks good” strategy.

# Recap the Graph Algorithm

- DFS (walking in a maze)
  - If we can explore, then explore.
  - If we can not explore, backtrack.
  - Do not re-visit a vertex.
  - Applications
    - Cycle
    - Topological
    - SCC

# Recap the Graph Algorithm

- BFS (waterfront)
  - 1 step from $r$
  - 2 steps from $r$
  - …
  - Application
    - Shortest Path

# Recap the Graph Algorithm

- Dijkstra (a generalized BFS)
  - Explore $s$.
  - Explore the closet vertex from $s$.
  - Explore the second closest vertex from $s$.
  - …
  - We can use Fibonacci heap to improve it.
- Bellman-Ford

# Are they Greedy?

# Do we have any other Greedy?

# Examples

- Finding Shortest Path
  - Dijkstra.
- Finishing homework
  - Keep finishing the one with the **closest** deadline.

# Is that optimal?

# Formalize the problem

- **Input:** $n$ homework, each homework $j$ has a size $s_j$, and a deadline $d_j$.
- **Output:** output a time schedule of doing homework!

# Algorithm

- Greedy
  - Keep finishing the homework with the **closest** deadline.
- Prove it is optimal.
- What is optimal?
- Claim: If we can not finish all the homework by the greedy order, then no one can finish all the homework on time.

Discussion

# Proof

- Claim: If we can not finish all the homework by the greedy order, then no one can finish all the homework on time.
- Proof:
  - If there exist $i$, finished later than $d_i$, what do we have?

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 2 | 2 | 2 | … | $i$ | $i$ | $i$ |

$d_i$

A possible schedule

# Proof

- Claim: If we can not finish all the homework by the greedy order, then no one can finish all the homework on time.
- Proof:
  - If there exist $i$, finished later than $d_i$, what do we have?

$\boxed{\sum_{j \le i} s_i \ge d_i}$ → No one can not finish all the homework (1~$i$) before $d_i$.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 2 | 2 | 2 | … | $i$ | $i$ | $i$ |

$d_i$

A possible schedule

Greedy:$\forall j < i, d_j < d_i$ → Whatever order, the last homework (1~$i$) can not be on time

# Minimum Spanning Tree

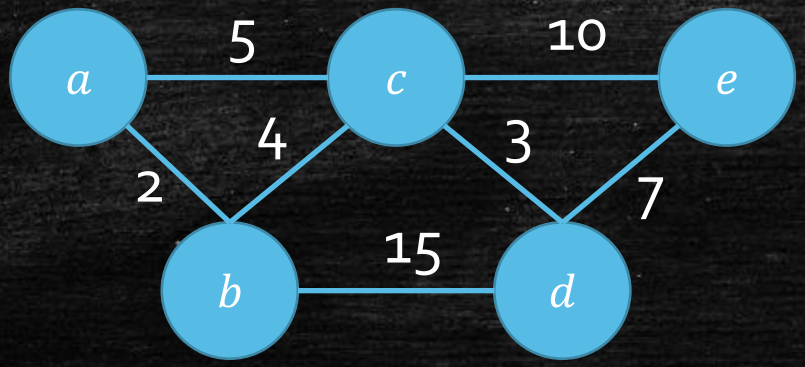
Prime & Kruskal

# Spanning Tree

- **Input:** Given a connected undirected graph $G = (V, E)$
- **Output:** A spanning tree of $G$ is, i.e., a subset of edges that forms a tree and contains all the vertices in $G$.
- Applications
  - Building a network, connecting all hubs via minimum number of cables.
- Solutions
  - BFS, DFS.

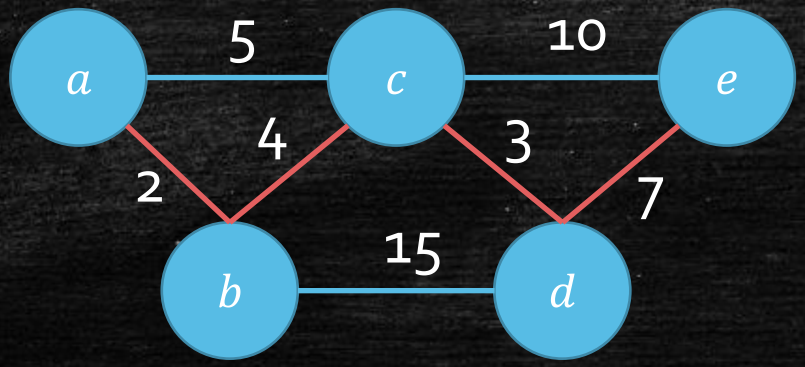
# Minimum Spanning Tree

- **Input:** Given a connected undirected graph $G = (V, E)$, and a weight function $w(e)$ for each $e \in E$.
- **Output:** A spanning tree of $G$ is, i.e., a subset of edges, with minimized total weight.
- Applications
  - Building a network, connecting all hubs via minimum number of cables.



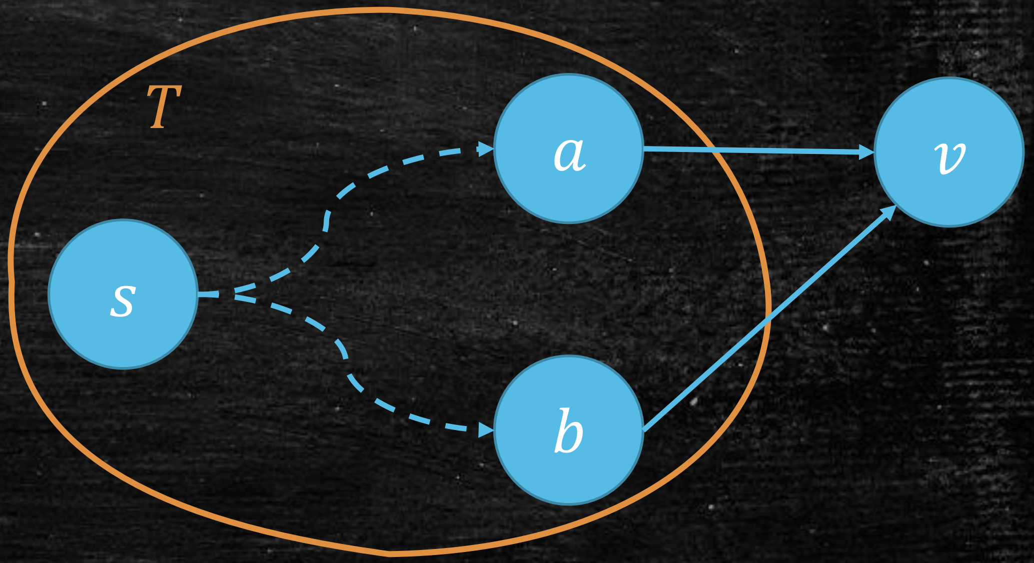
# Minimum Spanning Tree

- **Input:** Given a connected undirected graph $G = (V, E)$, and a weight function $w(e)$ for each $e \in E$.
- **Output:** A spanning tree of $G$ is, i.e., a subset of edges, with minimized total weight.
- Applications
  - Building a network, connecting all hubs via minimum number of cables.



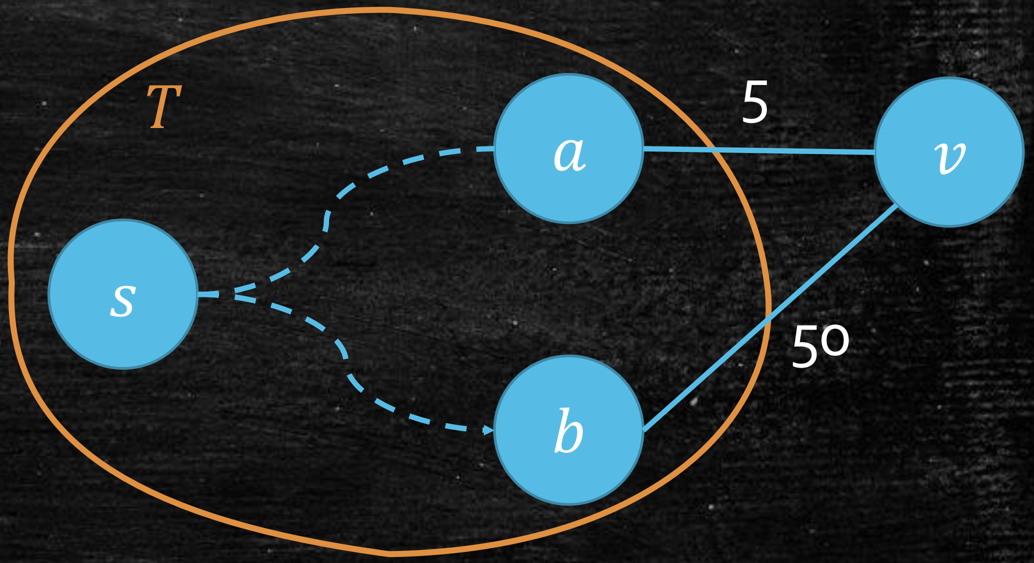
# Dijkstra's growing idea

- Given a small SPT,
- Choose a proper vertex $v$ to find a larger SPT.
- New Plan for MST:
- Given a small MST,
- choose a proper vertex $v$ to find a larger MST.



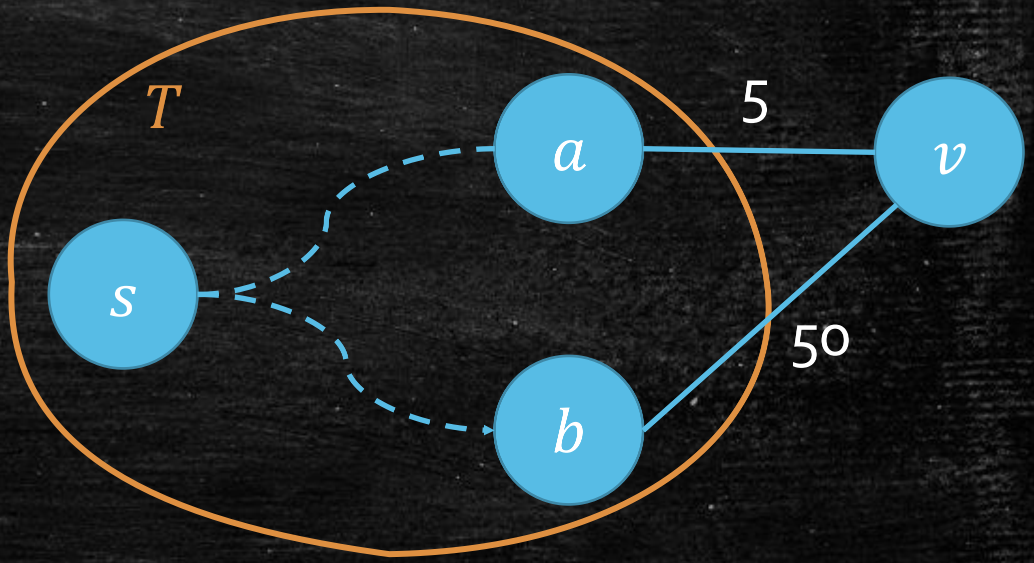
# Prim's growing idea

- Given a small MST,
- Choose a proper vertex $v$ to find a larger MST.
- Which $v$ is good?
- Dijkstra: $v$ with smallest $T$-distance to $s$.
- Now: $v$ with smallest cost!



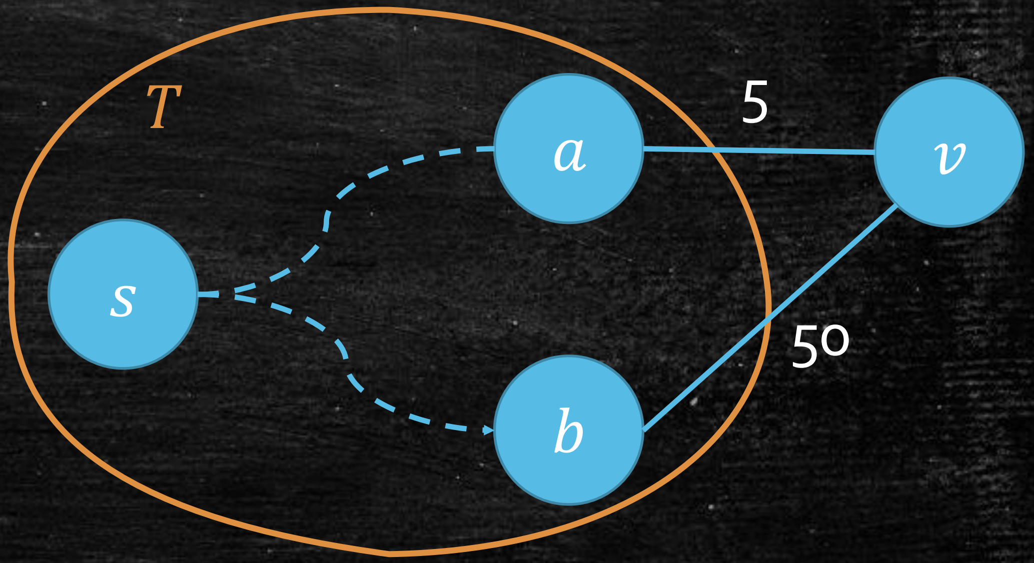
# Prim's growing idea

- Given a small MST,
- Choose a proper vertex $v$ to find a larger MST.
- Grow $v$ with smallest cost!
- Is it correct?
- Challenge:
  - How to define small MST



# How to define small MST?

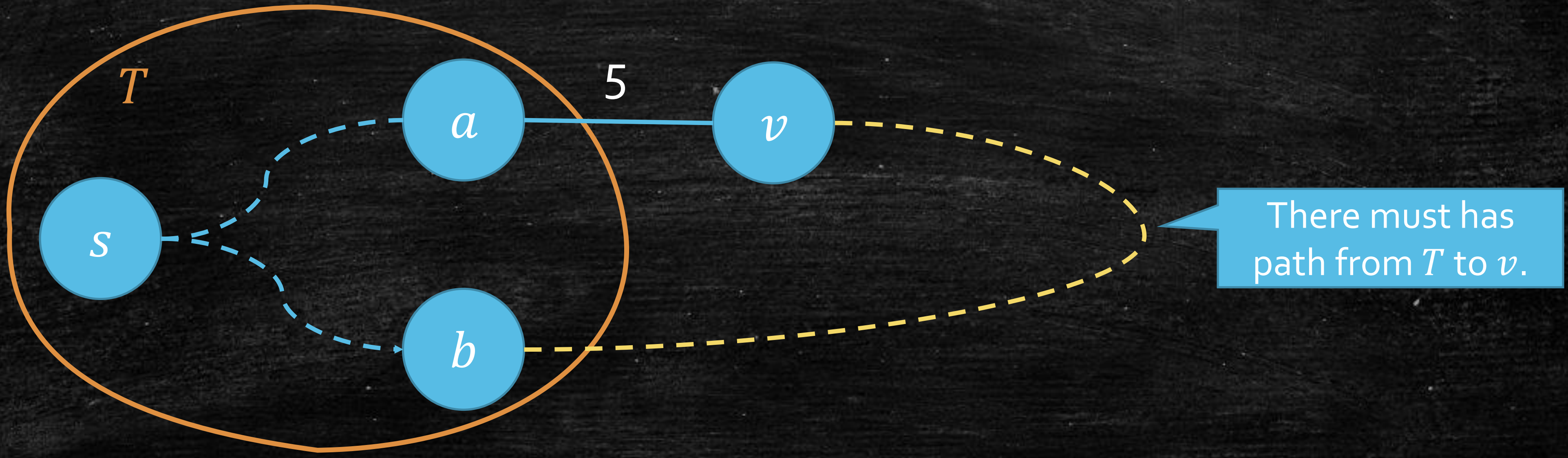
- $T = (V', E')$ is a small MST if it is an MST for $V'$.
- Problem
  - Does it suffice to say those edges are small?
  - Each two-vertex subgraph are small MST.
- A better choice:
- $T$ is a P-MST (Partial MST) if it is a part of a complete MST for $G$.



# Correctness of Prim's Growing idea

- Prove by contradiction again!
- Let's say $T^*$ is a complete MST that contains $T$ We assume $\forall T^*, (a, v) \notin T^*$.

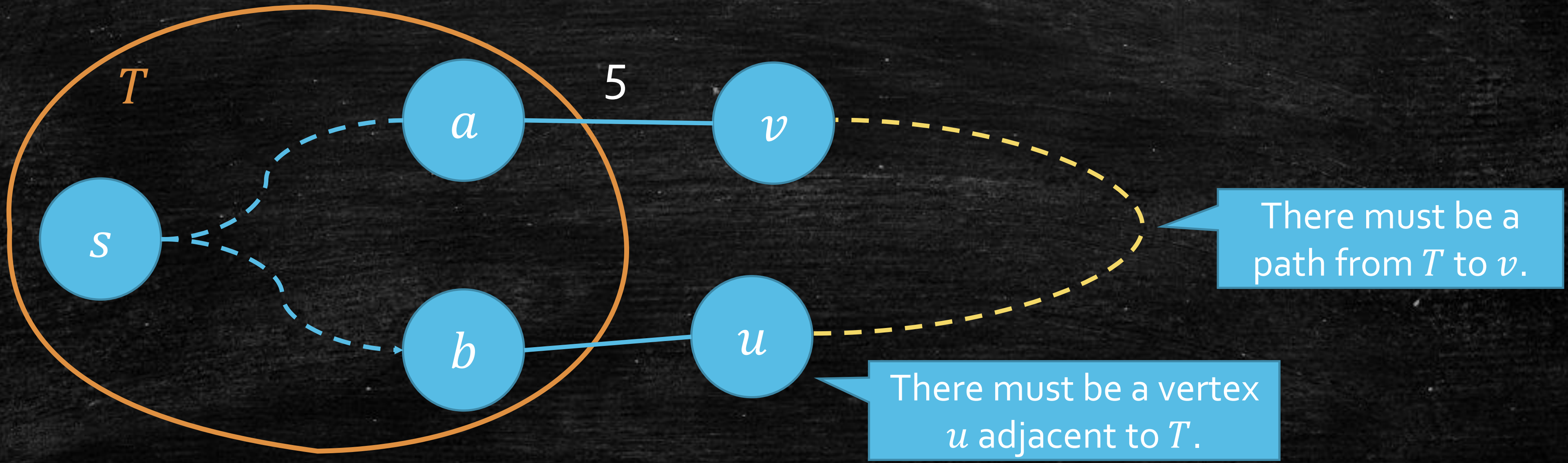
- **Given:** a small P-MST $T$.
- **Want:** a larger P-MST.
- Can we explore $v$ (smallest cost) into $T$?



# Correctness of Prim's Growing idea

- Let's say $T^*$ is a complete MST that contains $T$ We assume $\forall T^*$ , $(a, v) \notin T^*$.

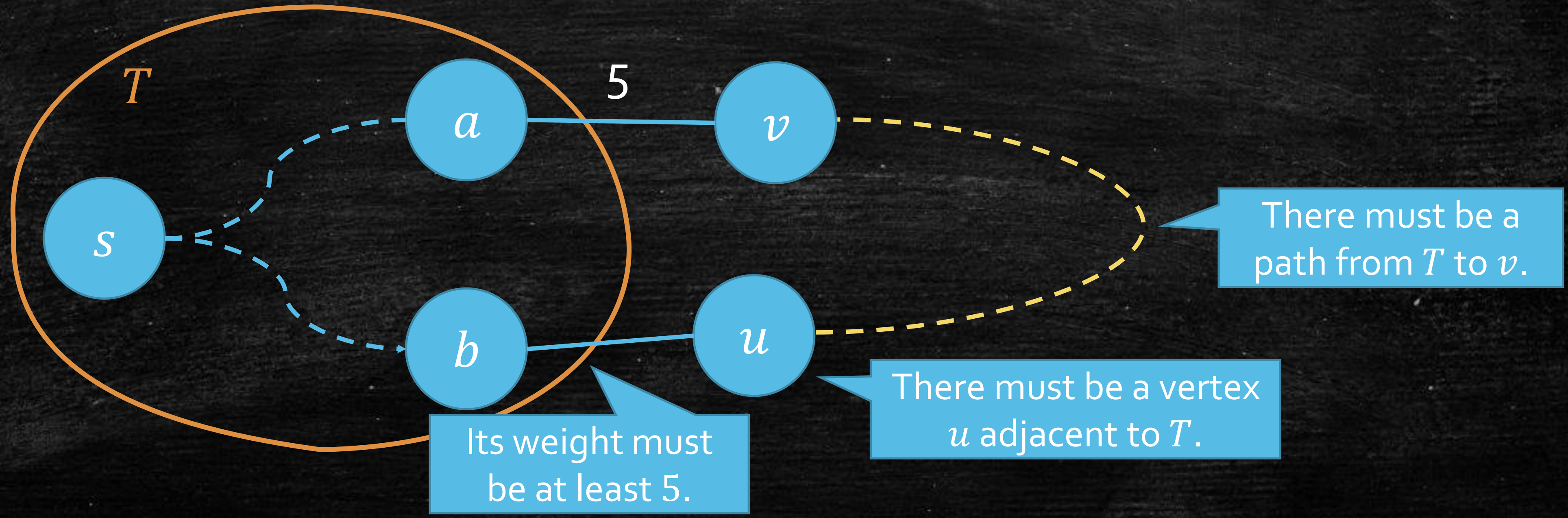
- **Given:** a small P-MST $T$.
- **Want:** a larger P-MST.
- Can we explore $v$ (smallest cost) into $T$?



# Correctness of Prim's Growing idea

- Let's say $T^*$ is a complete MST that contains $T$ We assume $\forall T^*$ , $(a, v) \notin T^*$.

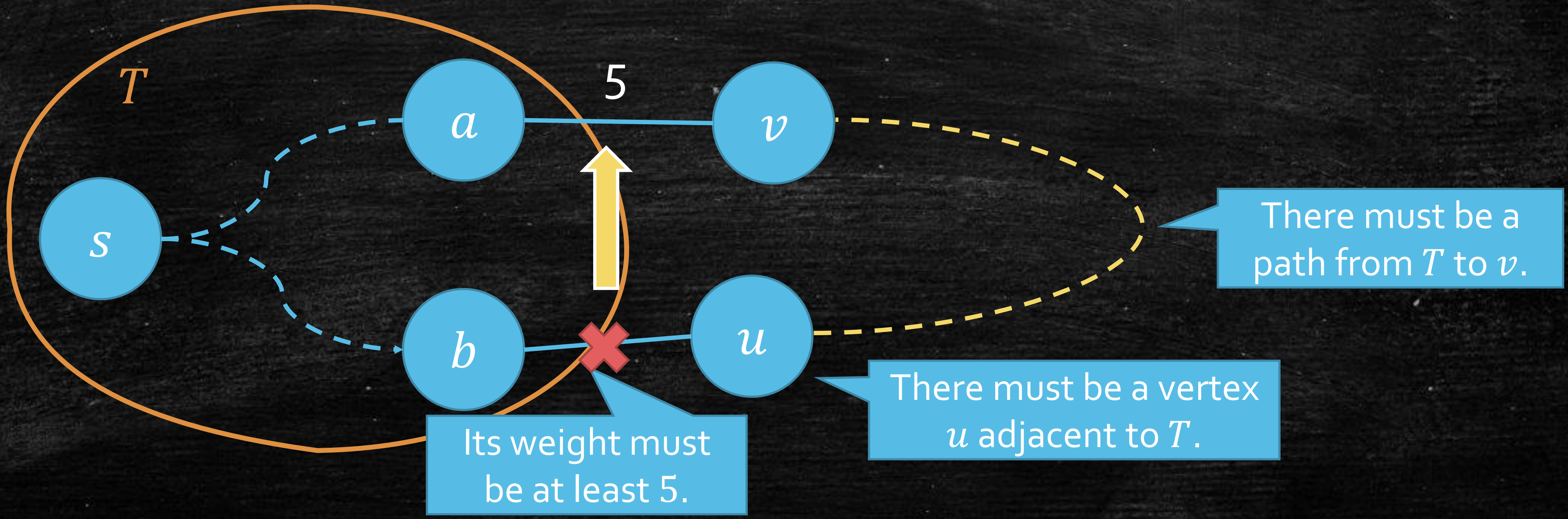
- **Given:** a small P-MST $T$.
- **Want:** a larger P-MST.
- Can we explore $v$ (smallest cost) into $T$?



# Correctness of Prim's Growing idea

- Let's say $T^*$ is a complete MST that contains $T$ We assume $\forall T^*$ , $(a, v) \notin T^*$.

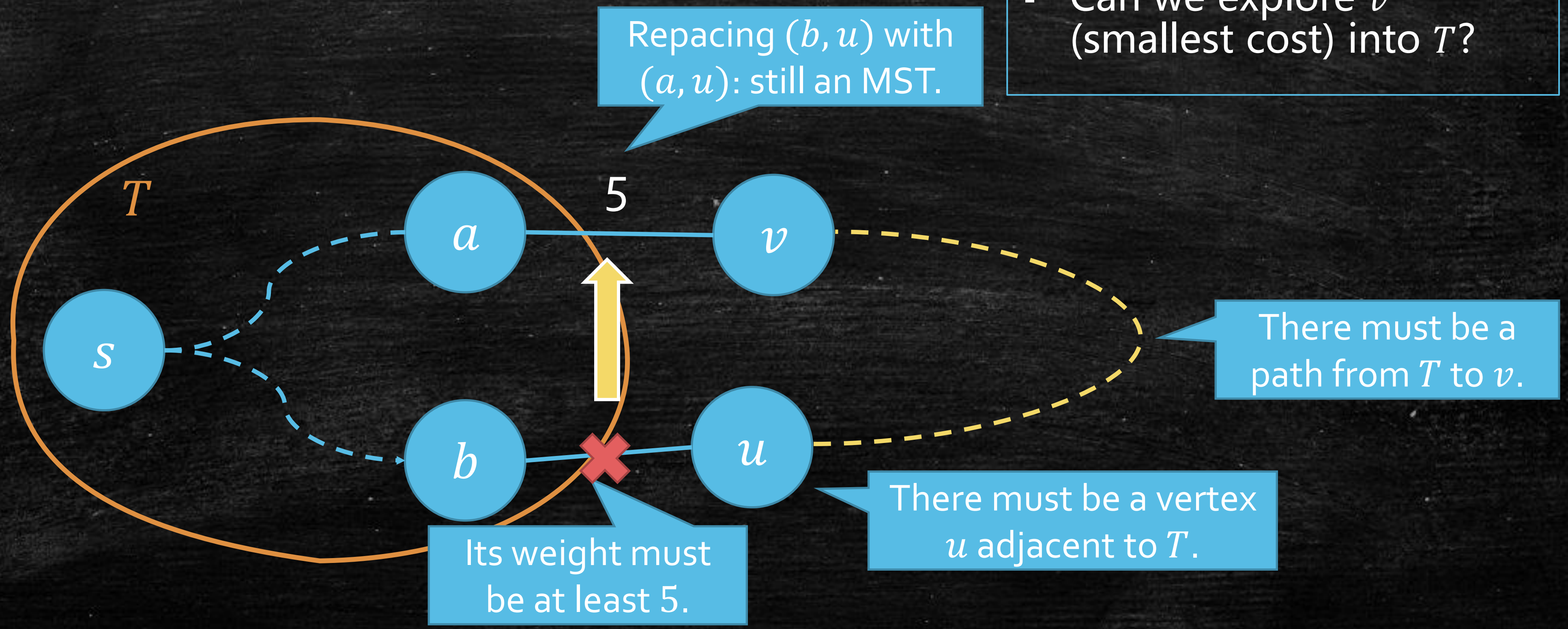
- **Given:** a small P-MST $T$.
- **Want:** a larger P-MST.
- Can we explore $v$ (smallest cost) into $T$?



# Correctness of Prim's Growing idea

- Let's say $T^*$ is a complete MST that contains $T$ We assume $\forall T^*$ , $(a, v) \notin T^*$.

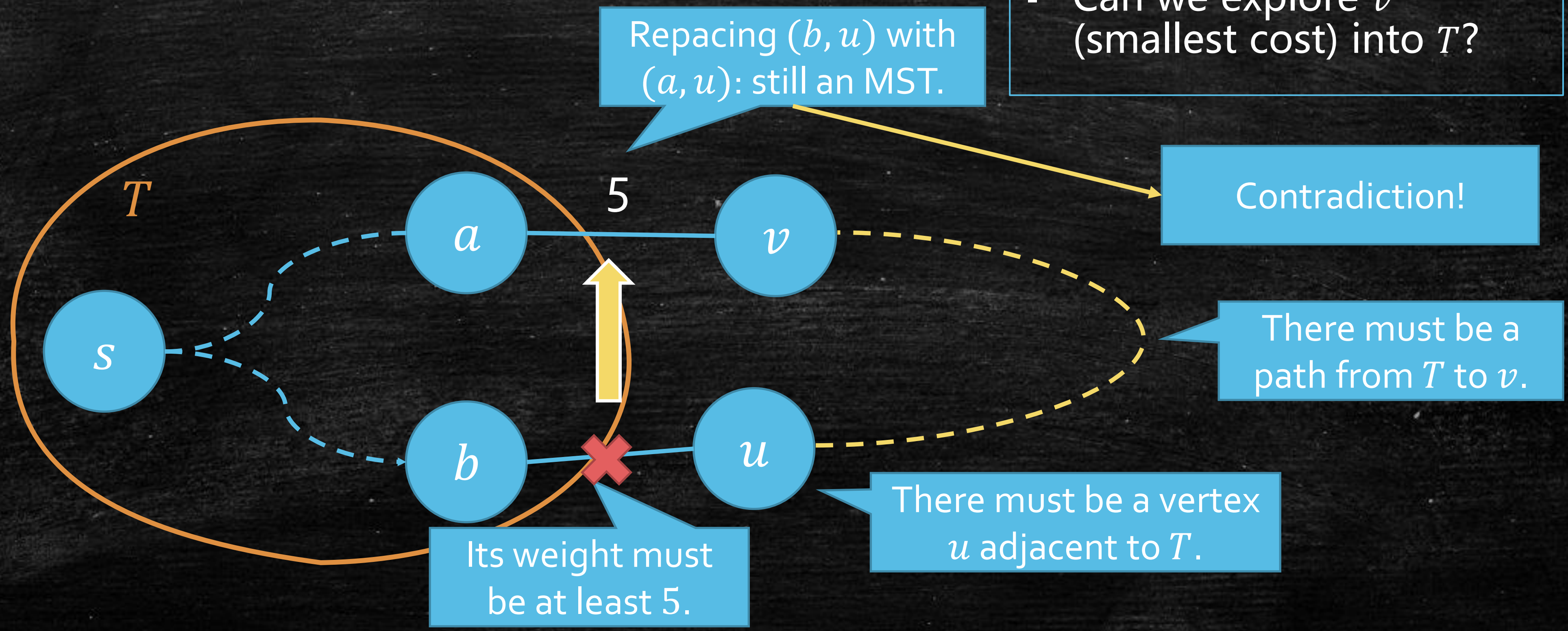
- **Given:** a small P-MST $T$.
- **Want:** a larger P-MST.
- Can we explore $v$ (smallest cost) into $T$?



# Correctness of Prim's Growing idea

- Let's say $T^*$ is a complete MST that contains $T$ We assume $\forall T^*,\ (a, v) \notin T^*$.

- **Given:** a small P-MST $T$.
- **Want:** a larger P-MST.
- Can we explore $v$ (smallest cost) into $T$?



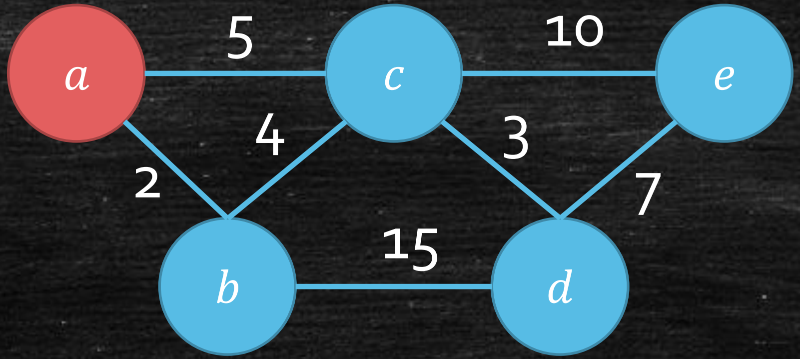
# Does negative weight matters?

# Prim Algorithm [Jarník '30, Prim '57, Dijkstra '59]

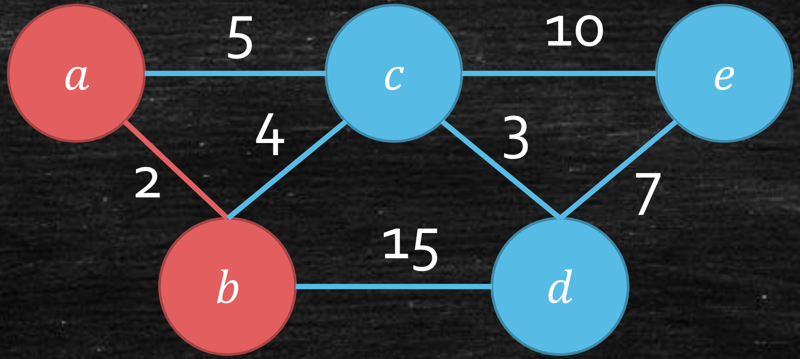
**Prim**$(\boldsymbol{G = (V, E)})$

1. **Initialize**
   - $T \leftarrow \{\,\}, S \leftarrow \{s\}$ ; #s is an arbitrary vertex.
   - $cost[s] = 0$, $cost[v] \leftarrow \infty$ for all $v$ other than $s$.
   - $cost[v] \leftarrow w(s, v), pre[v] = s$ for all $(s, v) \in E$.
2. **Explore**
   - Find $v \notin S$ with smallest $cost[v]$.
   - $S \leftarrow S + \{v\}; T \leftarrow T + \{(pre[v], v)\}$
3. **Update $\boldsymbol{cost[u]}$**
   - $cost[u] = \min\{cost[u], w(v, u)\}$ for all $(v, u) \in E$
   - If $cost[u]$ is updated, then $pre[u] = v$.

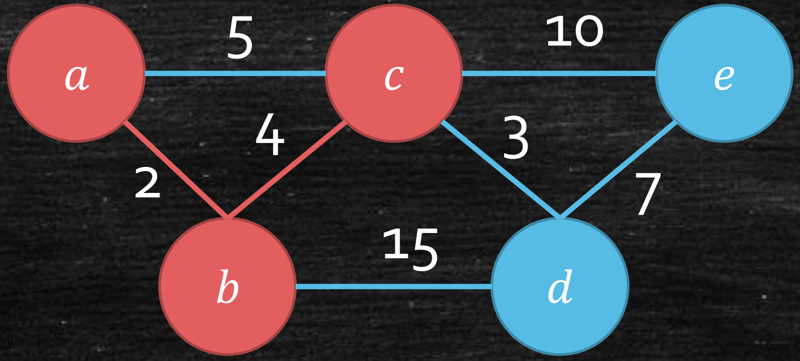
# Sample Run



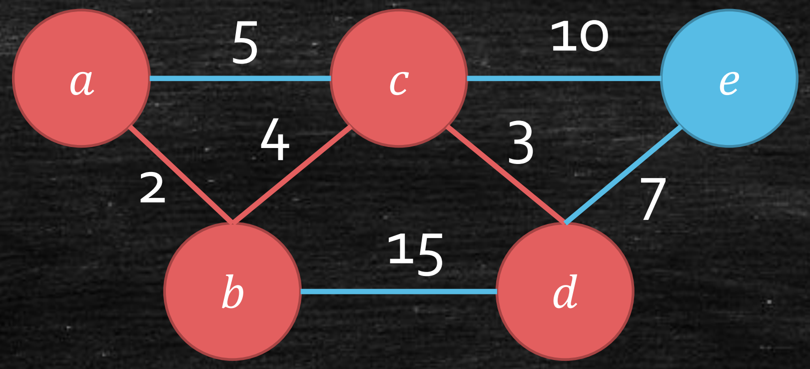
# Sample Run



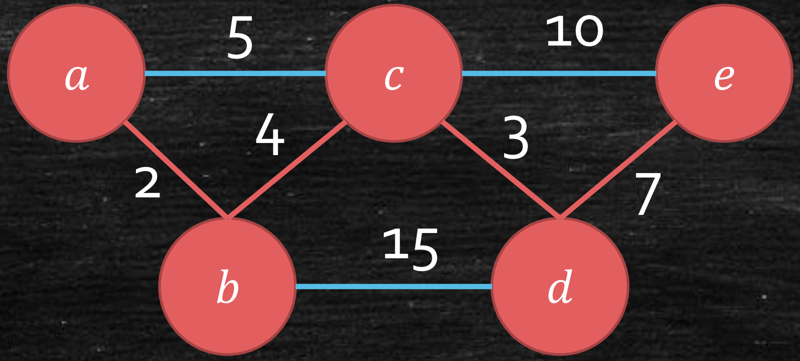
# Sample Run



# Sample Run



# Sample Run



# Running Time

- I believe you know how to analyze it:
- We have
  - $|E|$ rounds Update.
  - $|V|$ rounds PopMin.
- We can do it in $O(|E| + |V| \log|V|)$.
- Fibonacci Heap again!

# Kruskal Algorithm [Kruskal 1956]

- Another Greedy!

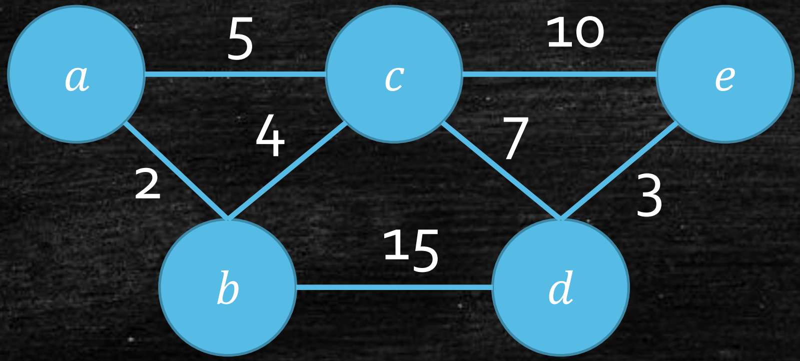
**Kruskal**$(G = (V, E))$

- Sort the edge set $E$ to ascending order.
- For each $e \in E$ in ascending order
  - If e do not create a **cycle**, then choose it.

# Kruskal Algorithm

**Kruskal**$(G = (V, E))$

- Sort the edge set $E$ to ascending order.
- For each $e \in E$ in ascending order
  - If e do not create a **cycle**, then choose it.

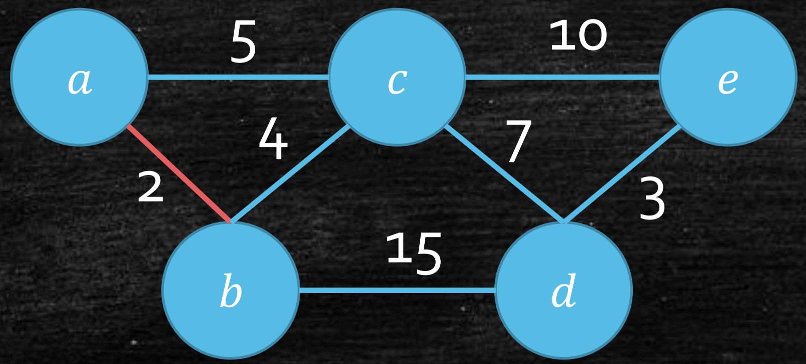



| | | | | | | |
|---|---|---|---|---|---|---|
| 2 | 3 | 4 | 5 | 7 | 10 | 15 |

# Kruskal Algorithm

**Kruskal**$(G = (V, E))$

- Sort the edge set $E$ to ascending order.
- For each $e \in E$ in ascending order
  - If e do not create a **cycle**, then choose it.

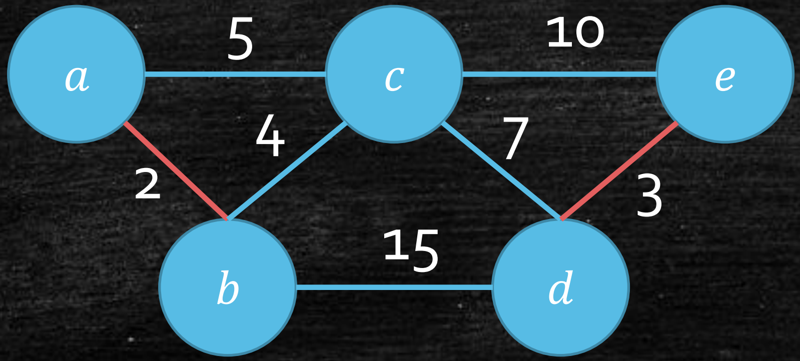



| | | | | | | |
|---|---|---|---|---|---|---|
| 2 | 3 | 4 | 5 | 7 | 10 | 15 |

# Kruskal Algorithm

**Kruskal**$(\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E}))$

- Sort the edge set $E$ to ascending order.
- For each $e \in E$ in ascending order
  - If e do not create a **cycle**, then choose it.

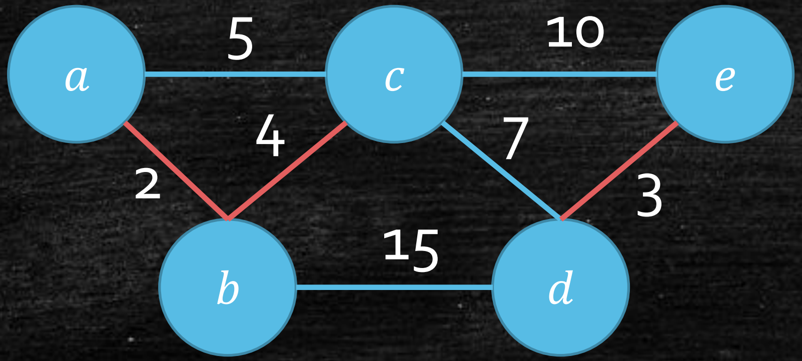



| | | | | | | |
|---|---|---|---|---|---|---|
| 2 | 3 | 4 | 5 | 7 | 10 | 15 |

# Kruskal Algorithm

**Kruskal**$(G = (V, E))$

- Sort the edge set $E$ to ascending order.
- For each $e \in E$ in ascending order
  - If e do not create a **cycle**, then choose it.

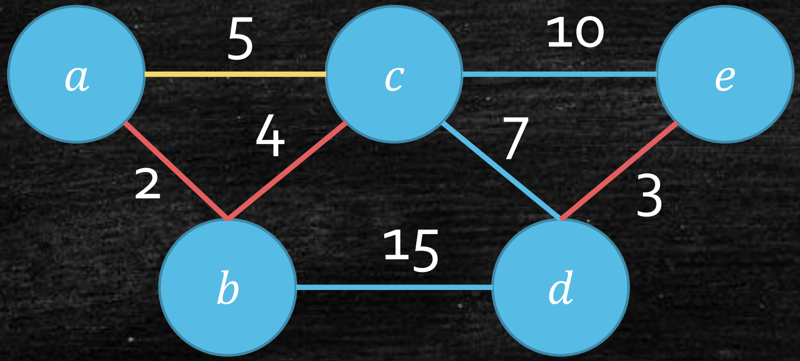



| | | | | | | |
|---|---|---|---|---|---|---|
| 2 | 3 | 4 | 5 | 7 | 10 | 15 |

# Kruskal Algorithm

**Kruskal**$(G = (V, E))$

- Sort the edge set $E$ to ascending order.
- For each $e \in E$ in ascending order
  - If e do not create a **cycle**, then choose it.

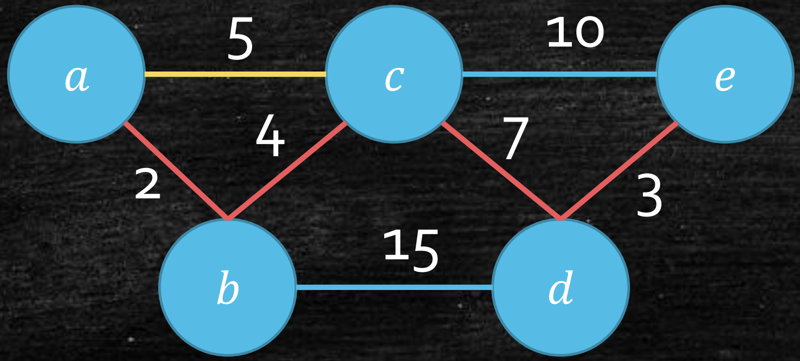



| | | | | | | |
|---|---|---|---|---|---|---|
| 2 | 3 | 4 | 5 | 7 | 10 | 15 |

# Kruskal Algorithm

**Kruskal**$(G = (V, E))$

- Sort the edge set $E$ to ascending order.
- For each $e \in E$ in ascending order
  - If e do not create a **cycle**, then choose it.

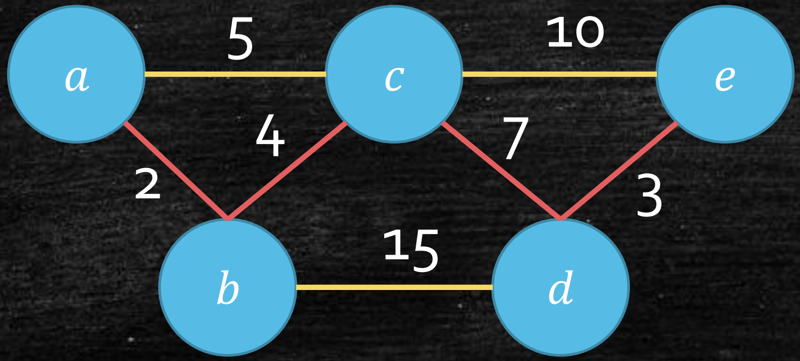



| | | | | | | |
|---|---|---|---|---|---|---|
| 2 | 3 | 4 | 5 | 7 | 10 | 15 |

# Kruskal Algorithm

**Kruskal**$(G = (V, E))$

- Sort the edge set $E$ to ascending order.
- For each $e \in E$ in ascending order
  - If e do not create a **cycle**, then choose it.

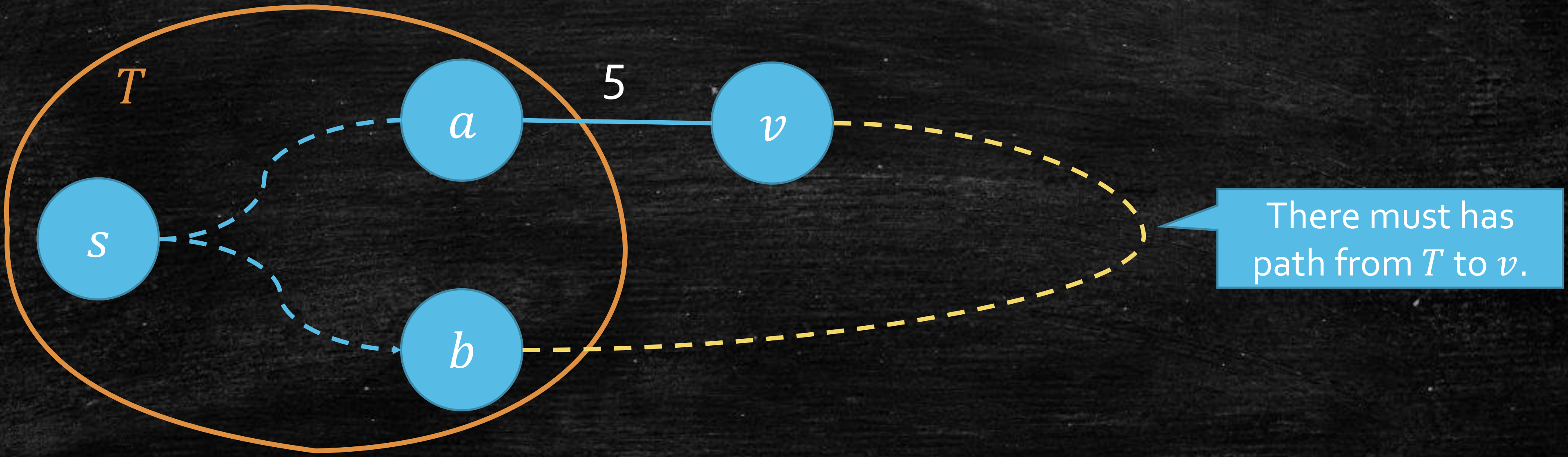



| | | | | | | |
|---|---|---|---|---|---|---|
| 2 | 3 | 4 | 5 | 7 | 10 | 15 |

# Correctness of Prim's Growing idea

- Let's say $T^*$ is the complete MST that contains $T$, and suppose $(\mathrm{a}, \mathrm{v}) \notin T^*$.

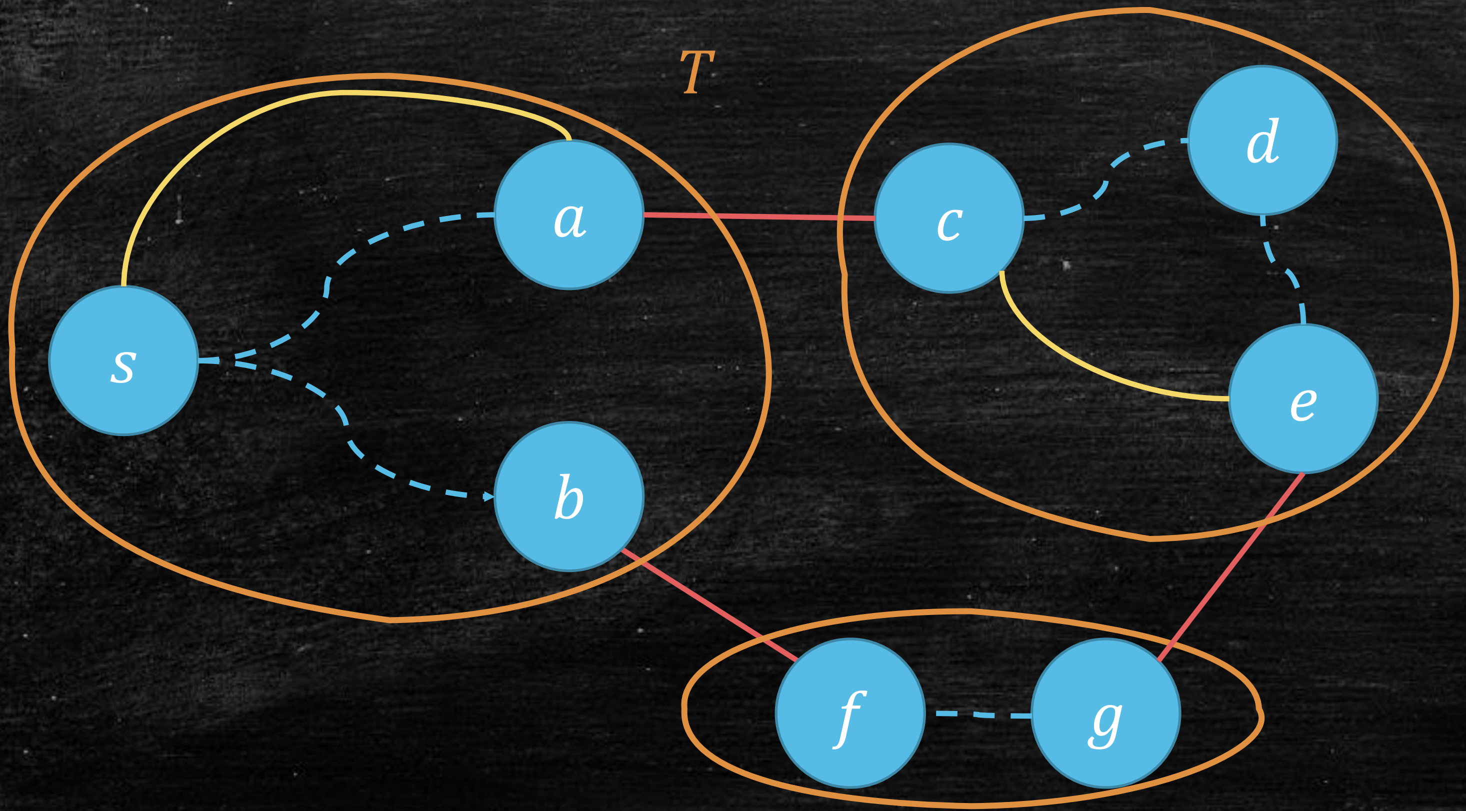
- **Given:** a small P-MST $T$.
- **Want:** a larger P-MST.
- Can we explore $v$ (smallest cost) into $T$?



# Correctness of Kruskal's Growing idea

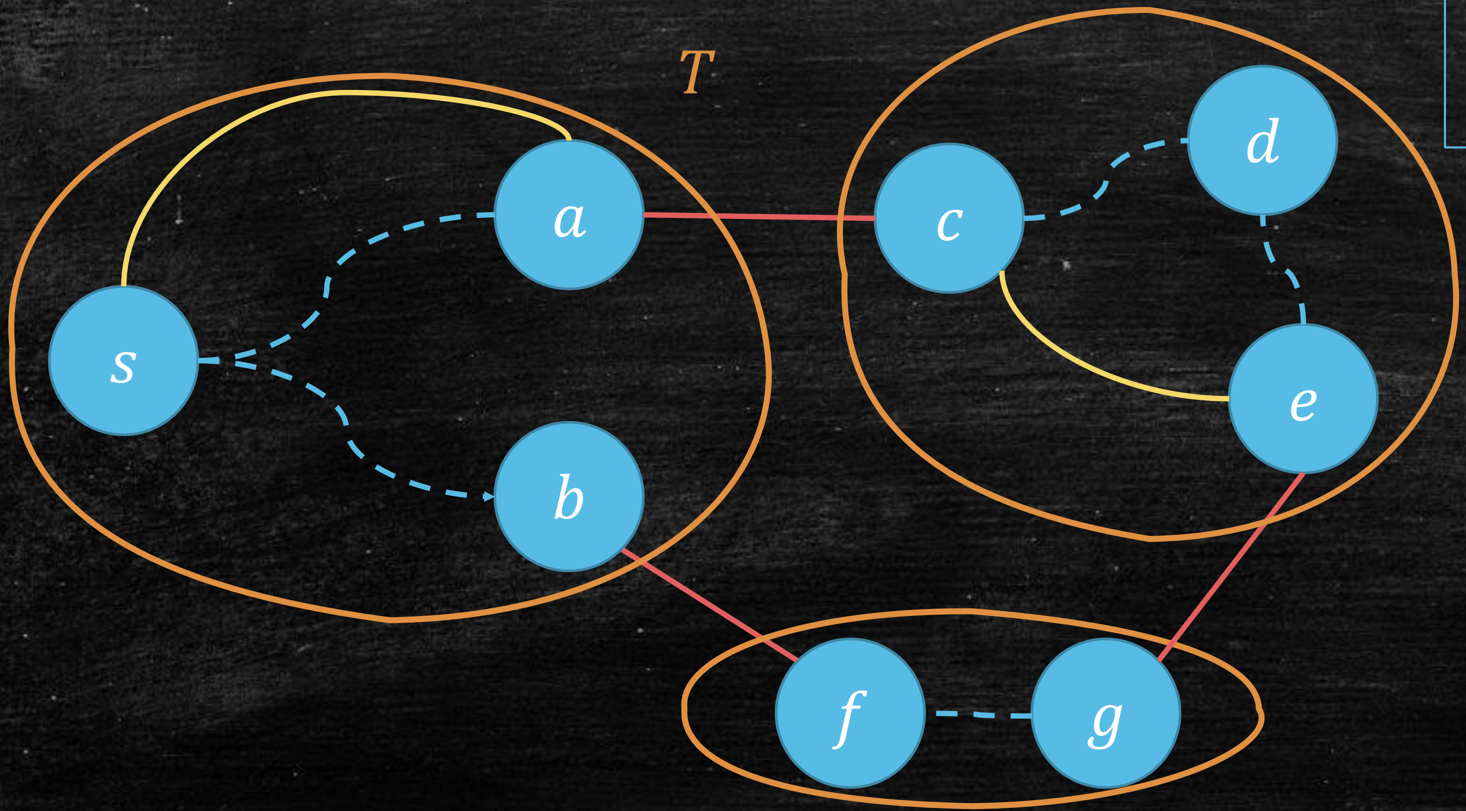
- Let's say $T^*$ is the complete MST that contains $T$, and suppose $(\mathrm{a}, \mathrm{v}) \notin T^*$.

- **Given:** a small P-MST $T$.
- **Want:** a larger P-MST.



# Correctness of Kruskal's Growing idea

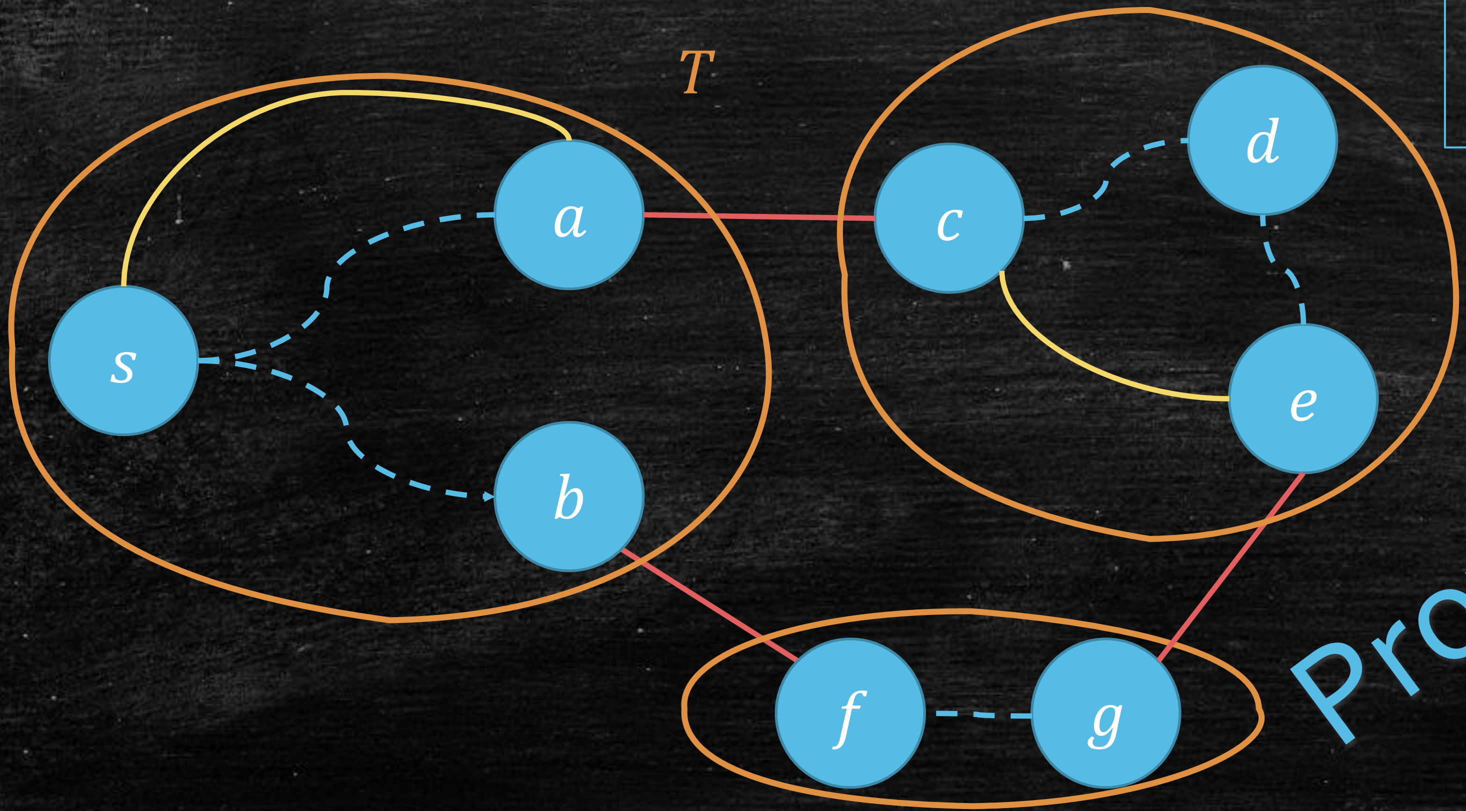
- Let's say $T^*$ is the complete MST that contains $T$, and suppose $(\mathrm{a}, \mathrm{v}) \notin T^*$.




- **Given:** a small P-MST $T$.
- **Want:** a larger P-MST.
- Add the smallest red edge get a larger P-MST.

# Correctness of Kruskal's Growing idea

- Let's say $T^*$ is the complete MST that contains $T$, and suppose $(\mathrm{a}, \mathrm{v}) \notin T^*$.




- **Given:** a small P-MST $T$.
- **Want:** a larger P-MST.
- Add the smallest red edge get a larger P-MST

Prove by yourself!

# Running Time

**Kruskal**$(G = (V, E))$

- Sort the edge set $E$ to ascending order.
- For each $e \in E$ in ascending order
  - If e do not create a **cycle**, then choose it.

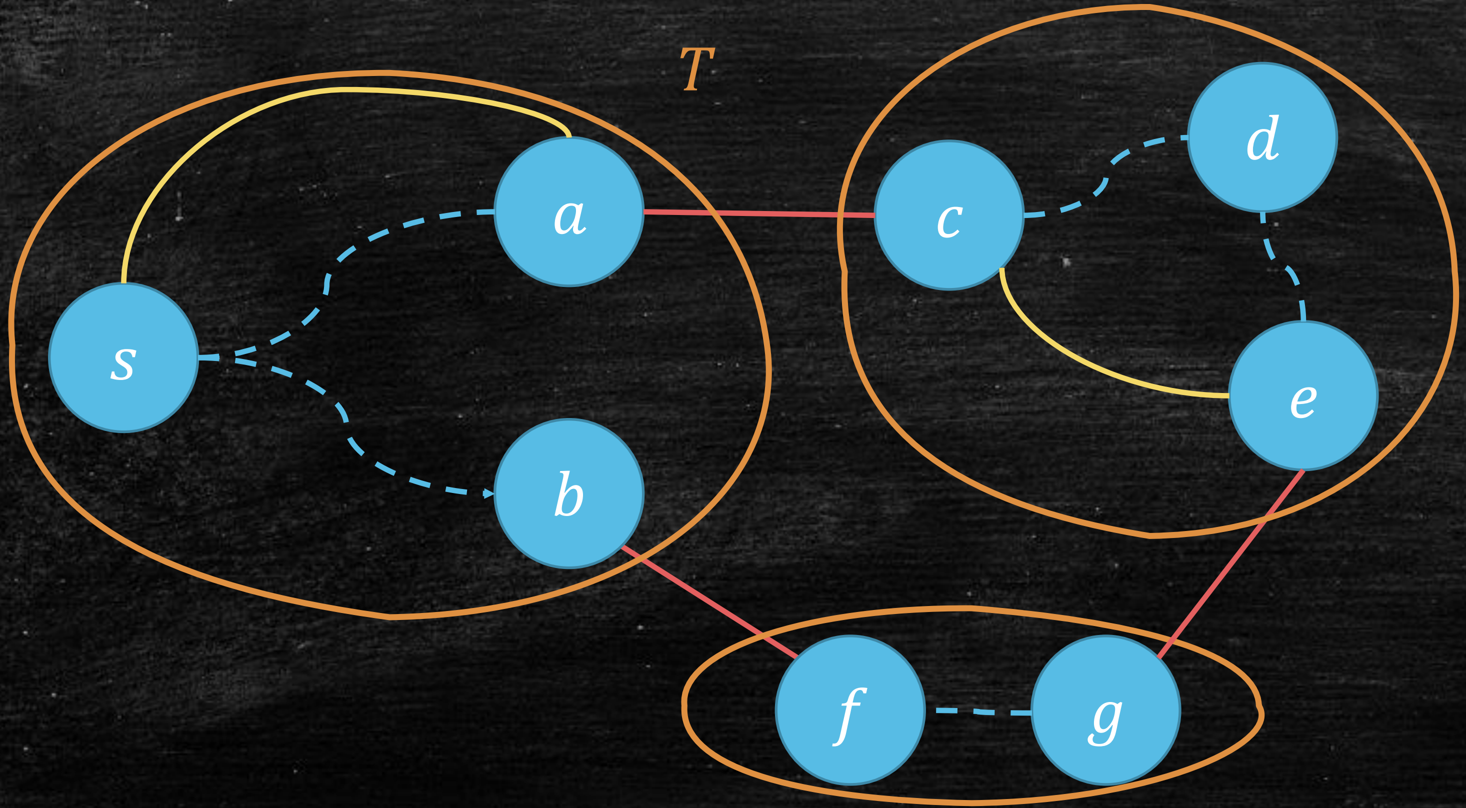
- $O(|E| \log|E|) = O(|E| \log|V|)$ for sorting.
- $|E|$ round: check cycle!

# Recall DFS

- When an edge is a back edge (to marked vertices),
- It forms a cycle.

# During Kruskal

- Cycle: When an edge connect the same group vertices.
- Change: $Marked[v] \rightarrow Group[v]$.



# Kruskal (refine)

**Kruskal**$(\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E}))$

- Sort the edge set $E$ to ascending order.
- For each $(u, v) \in E$ in ascending order
  - If $group(u)! = group(v)$
    - Choose $(u, v)$.
    - $union(group(u), group(v))$

# Running Time: Kruskal (refine)

**Kruskal**$(\boldsymbol{G = (V, E)})$

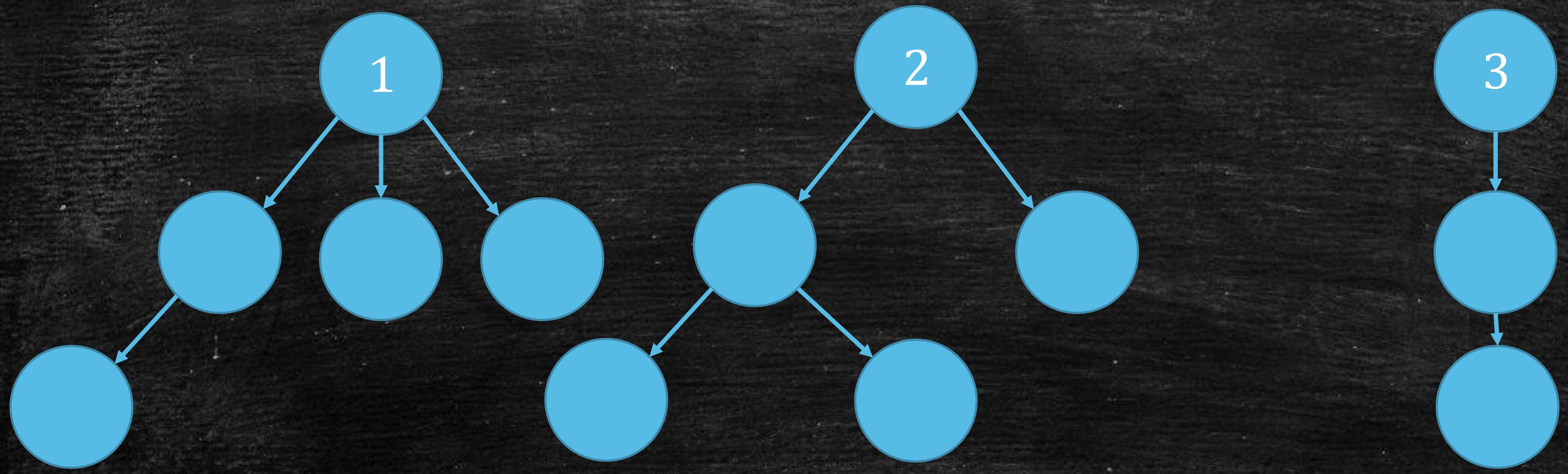
- Sort the edge set $E$ to ascending order.
- For each $(u, v) \in E$ in ascending order
  - If $group(u)! = group(v)$
    - Choose $(u, v)$.
    - $union(group(u), group(v))$

- $O(|E|\log|E|)$ for sorting.
- $2|E|$ round: check group
- $|V|$ round: union group

# Union-Find Set

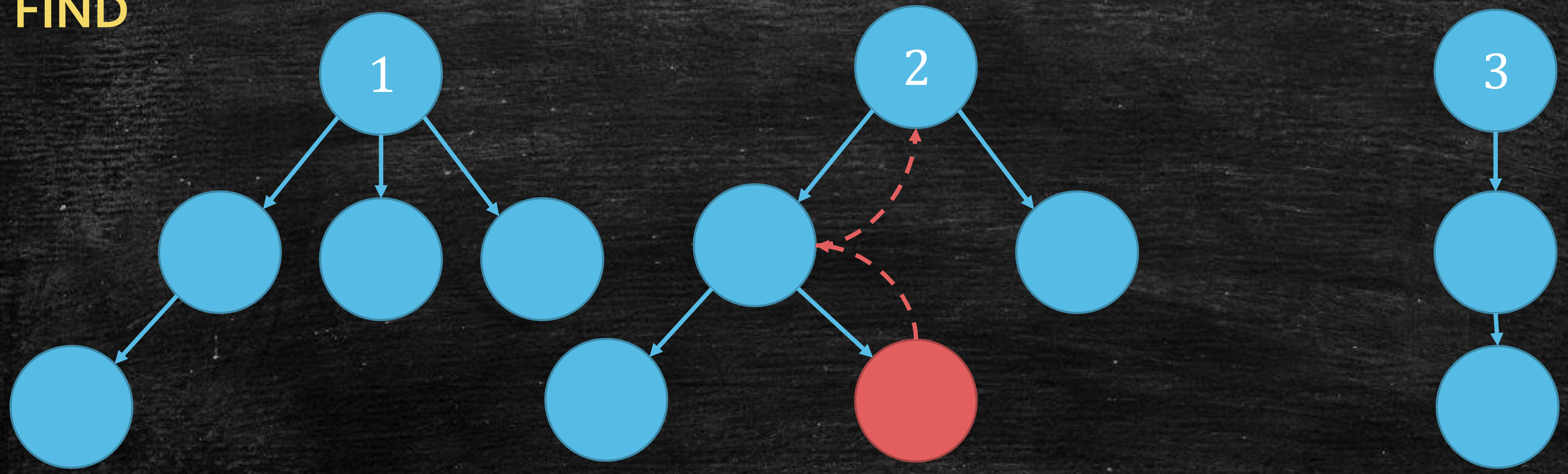
- Recall Union-Find Set
  - Find: $O(\log n)$
  - Union: $O(1)$
- Kruskal
  - $O(|E|\log|E|)$ for sorting.
  - $2|E|$ round: check group
  - $|V|$ round: union group
  - $O(|E|\log|E|) = O(|E|\log|V|)$
- Prime
  - $O(|E| + |V|\log|V|)$

# Review Union-Find Set



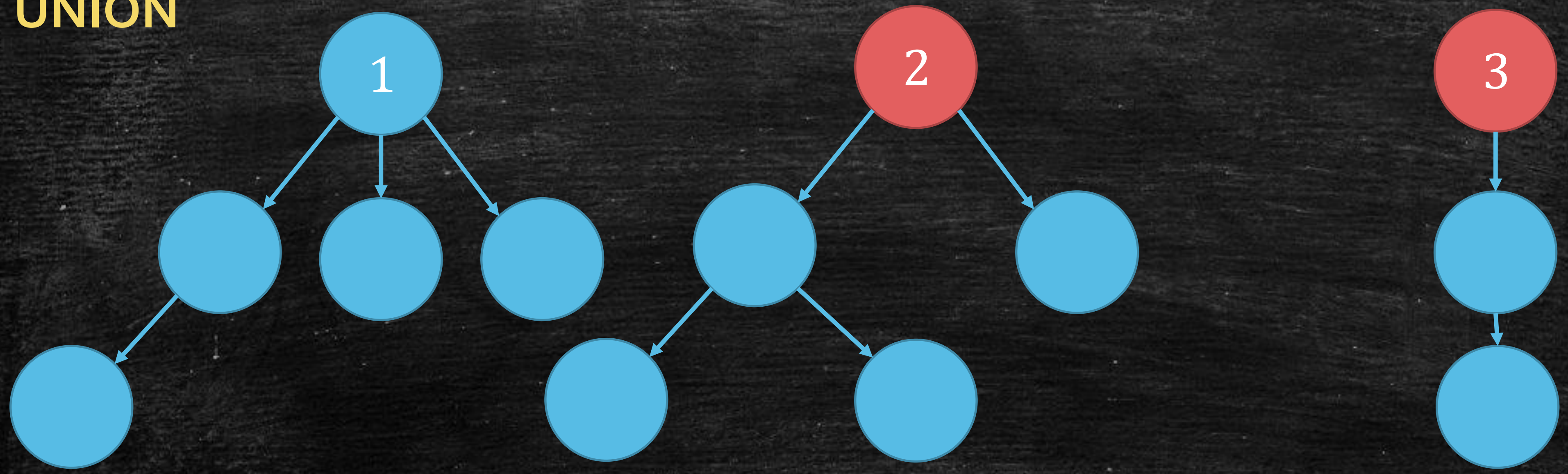
# Review Union-Find Set

FIND



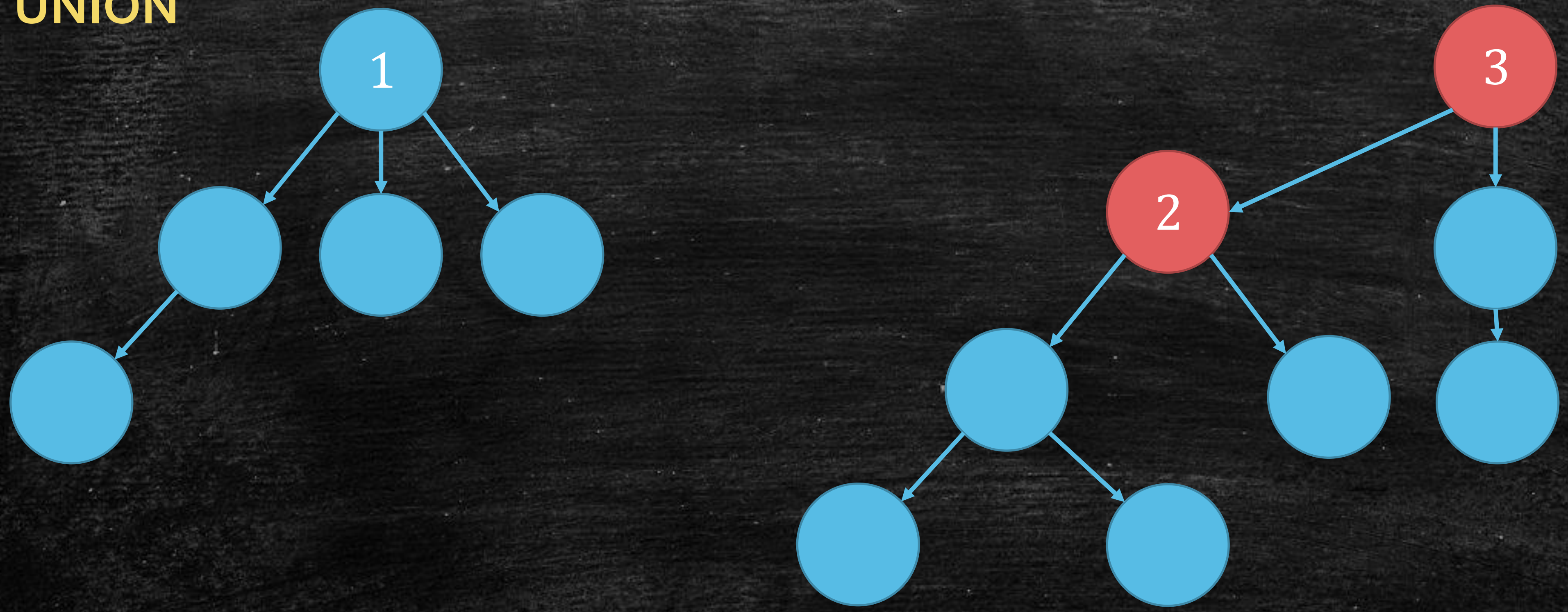
# Review Union-Find Set

UNION



# Review Union-Find Set

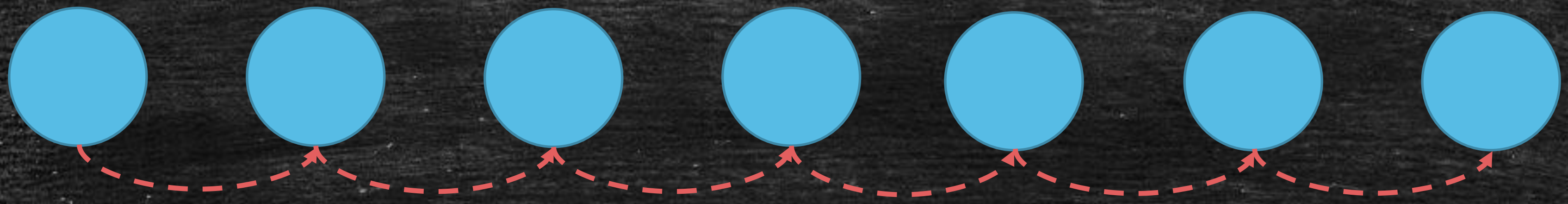
UNION



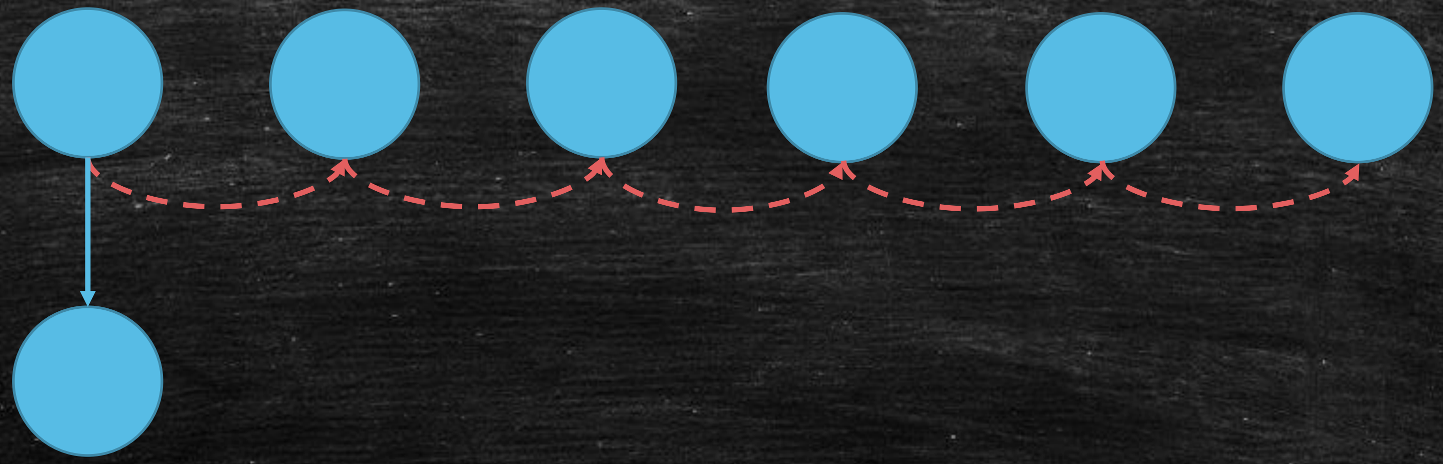
# Time Complexity

- Find
  - $O(\max\{Tree\ height\})$
- Union
  - $O(1)$

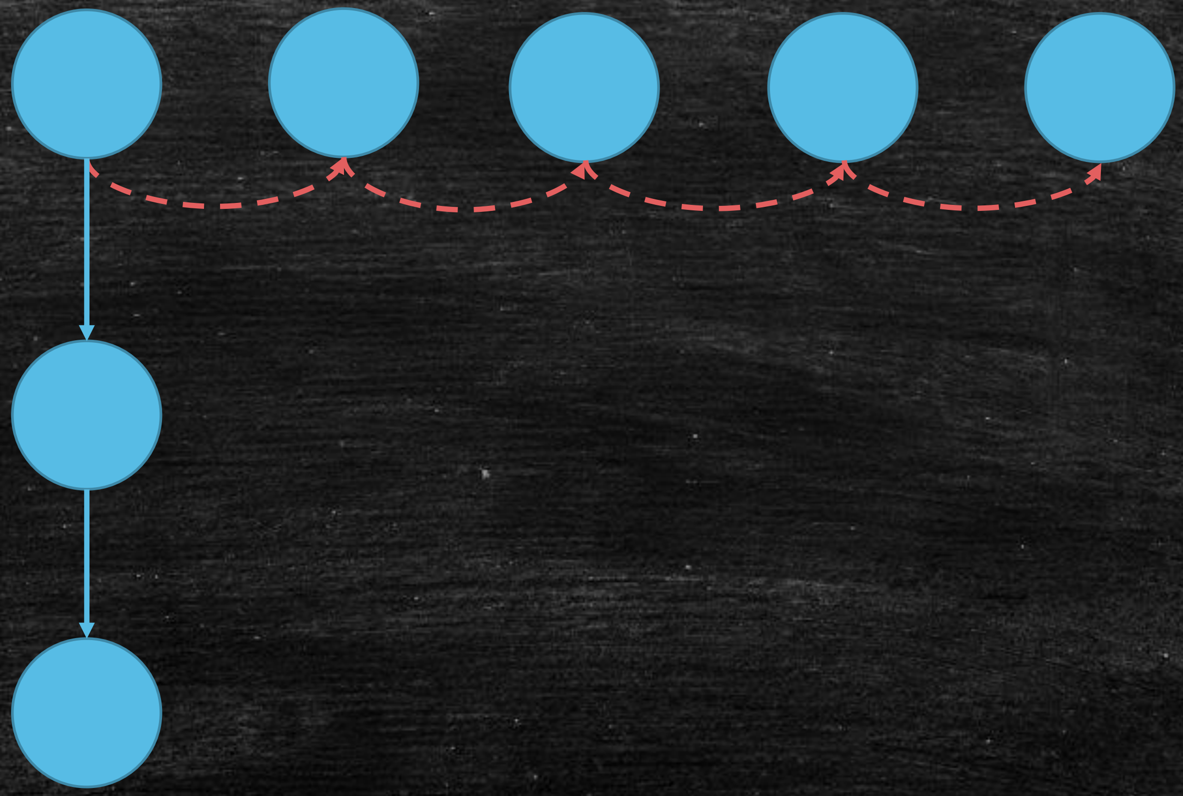
# A bad Case



# A bad Case

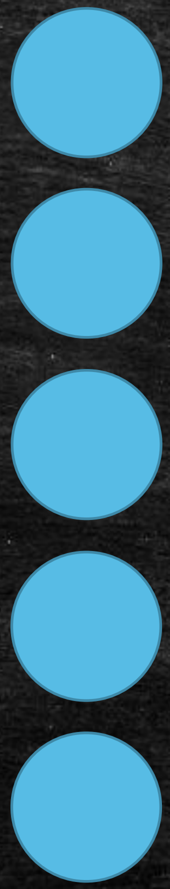


# A bad Case



# A bad Case

O(n) tree height



# How to improve

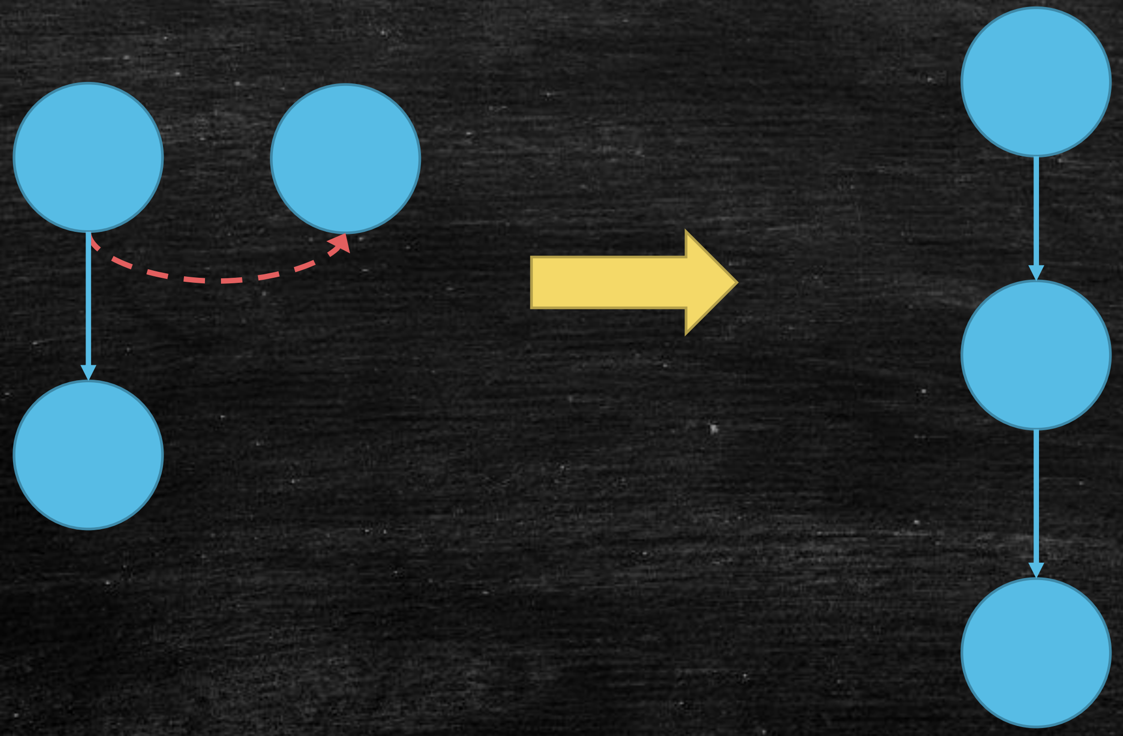
- Find
  - $O(\max\{Tree\ height\})$
  - $O(n)$!
- Union
  - $O(1)$
- To Do
  - Reduce Tree Height

# Amortized?

- The cost we pay is $1 + 2 + 3 + 4 + \cdots + n$
- The amortized cost is still $O(n)$.

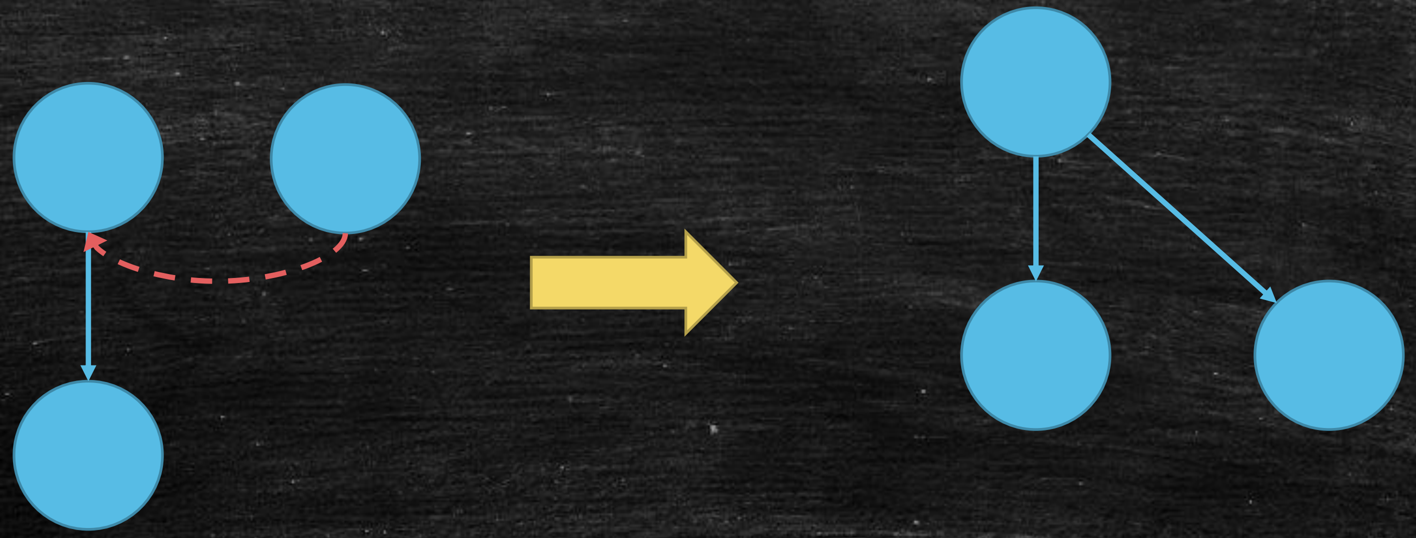
# Intuition

BAD



# Intuition

GOOD



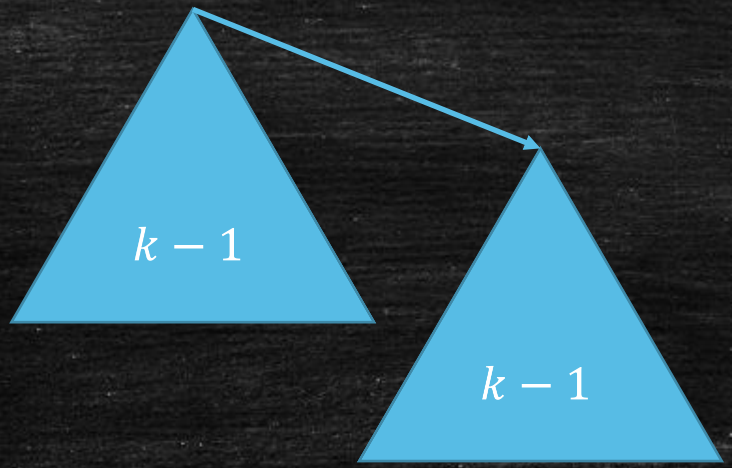
We should merge to a same root!
We should merge short tree to high tree!

# Implement

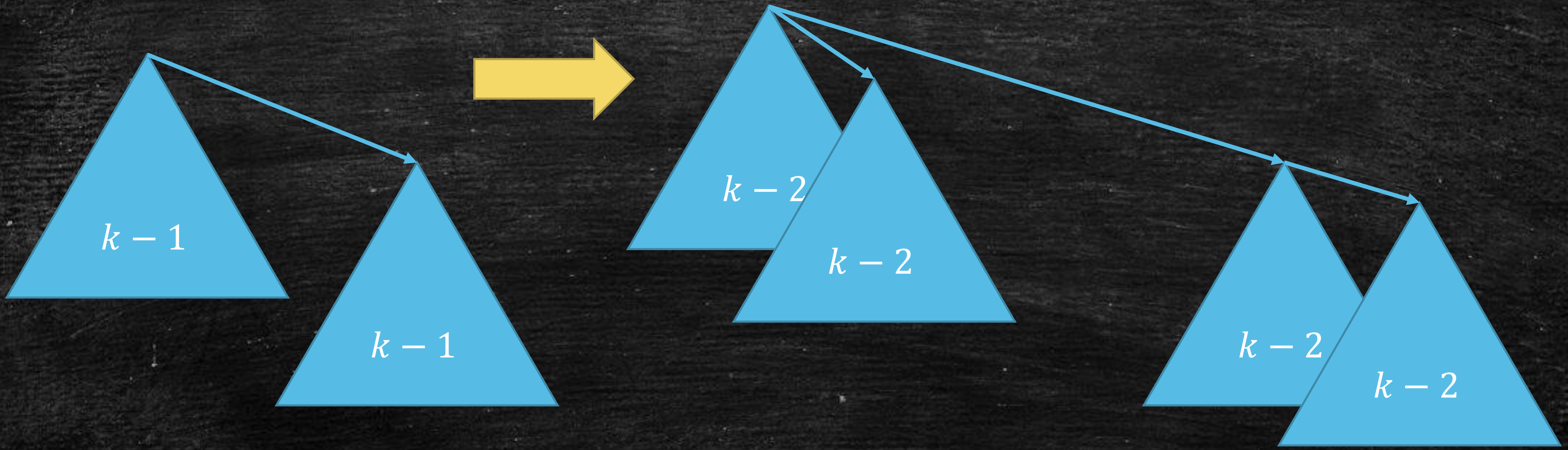
- Record Tree's Height (also called rank).
- $rank[v]$: the rank of tree rooted at $v$.
- Union: $u$ and $v$.
  - Rooted at $u$: if $rank[u] \geq rank[v]$
  - Rooted at $v$: if $rank[u] < rank[v]$
  - Update $rank[u]{+}{+}$: if $rank[u] = rank[v]$
- We make it hard to build a large rank tree!

# How to build a rank $k$ tree?

# How to build a rank $k$ tree?



# How to build a rank $k$ tree?




We should at least use **$2^k$ nodes!**

# Max tree height

- Build a rank $k$ tree: We should at least use $2^k$ **nodes!**
- What is the max tree height (rank)? $\boldsymbol{O(\log n)}$
- Find
  - $O(\max\{Tree\ height\})$
  - $\boldsymbol{O(\log n)}$!
- Union (rank based)
  - $O(1)$

# Union-Find Set

- Recall Union-Find Set
  - Find: $O(\log n)$
  - Union: $O(1)$
- Kruskal
  - $O(|E|\log|E|)$ for sorting.
  - $2|E|$ round: check group
  - $|V|$ round: union group
  - $O(|E|\log|E|) = O(|E|\log|V|)$
- Prime
  - $O(|E| + |V|\log|V|)$

# Can we do better for MST?

- $m = |E|!$
- Karger-Klein Tarjan (1995)
  – $O(m)$ randomized algorithm.
- Chazelle (2000)
  – $O(m \cdot \alpha(n))$ deterministic algorithm.
  – $\alpha(n)$ is the inverse Ackermann function $\alpha(9876!) \leq 5$.
  – Ackermann function: $A(4,4) \approx 2^{2^{2^{2^{16}}}}$.
- Pettie-Ramachandran (2002)
  – O(optimal #comparison to determine solution)
  – We know #comparison $= \Omega(n) = O(m \cdot \alpha(n))$

# Can we do better

- Kruskal
  - $O(|E|\log|E|)$ for sorting.
  - $2|E|$ round: check group
  - $|V|$ round: union group
  - $O(|E|\log|E|) = O(|E|\log|V|)$
- There are two bottlenecks
  - Sorting
  - Union and Find.
- If we do not need to sort, can we do better?

# Can we do better for Union-Find Set?

Have you heard **Path Compression**?

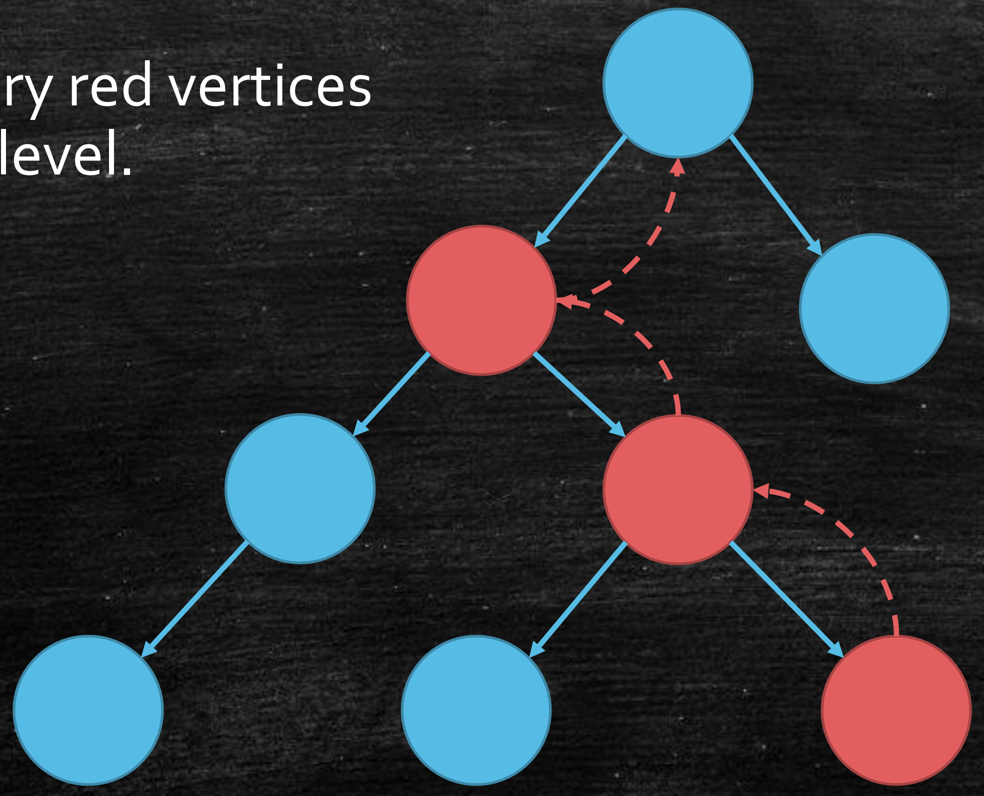
# Path Compression

Why not do some good things for the future?

# Path Compression

## FIND

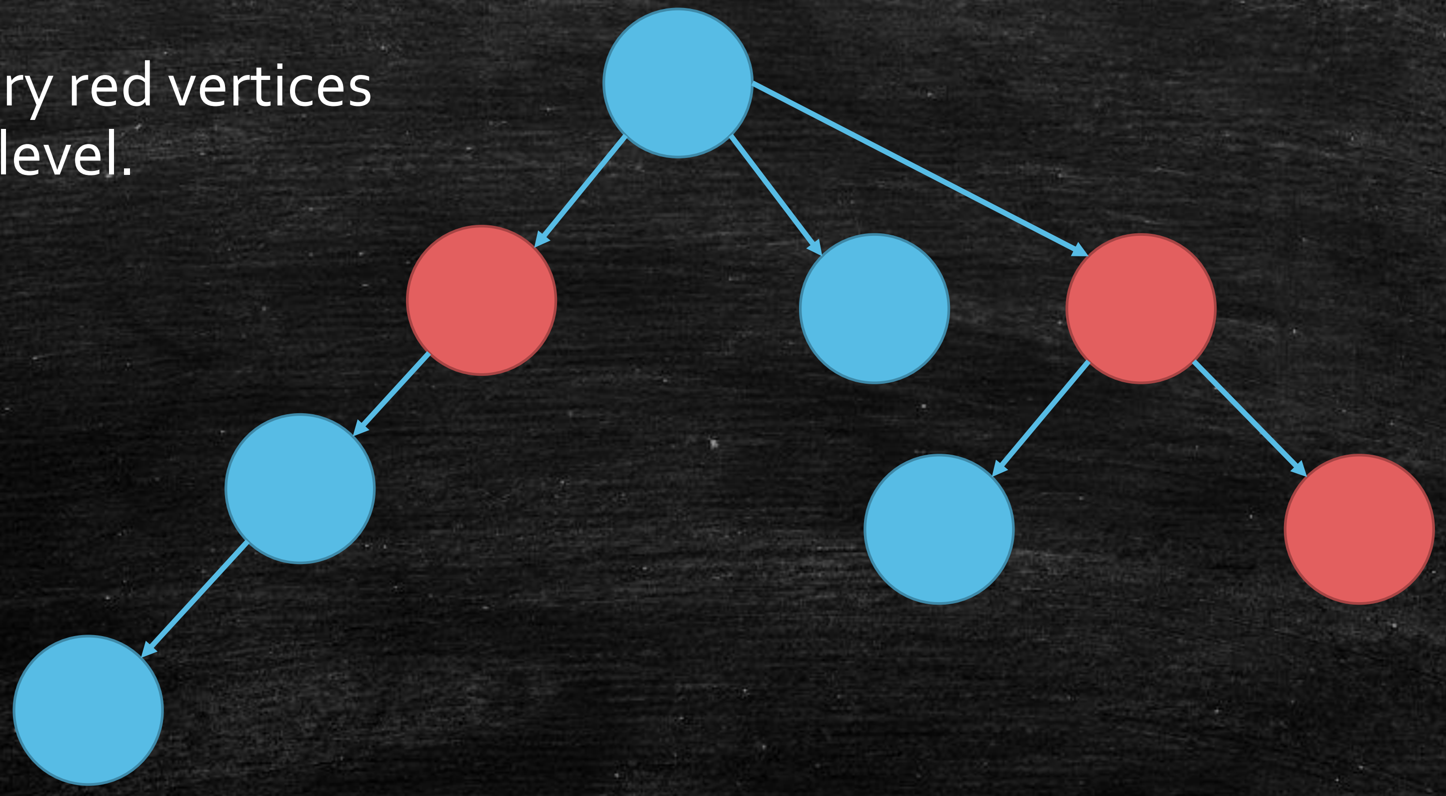
We put every red vertices
to the first level.



# Path Compression

**FIND**

We put every red vertices to the first level.

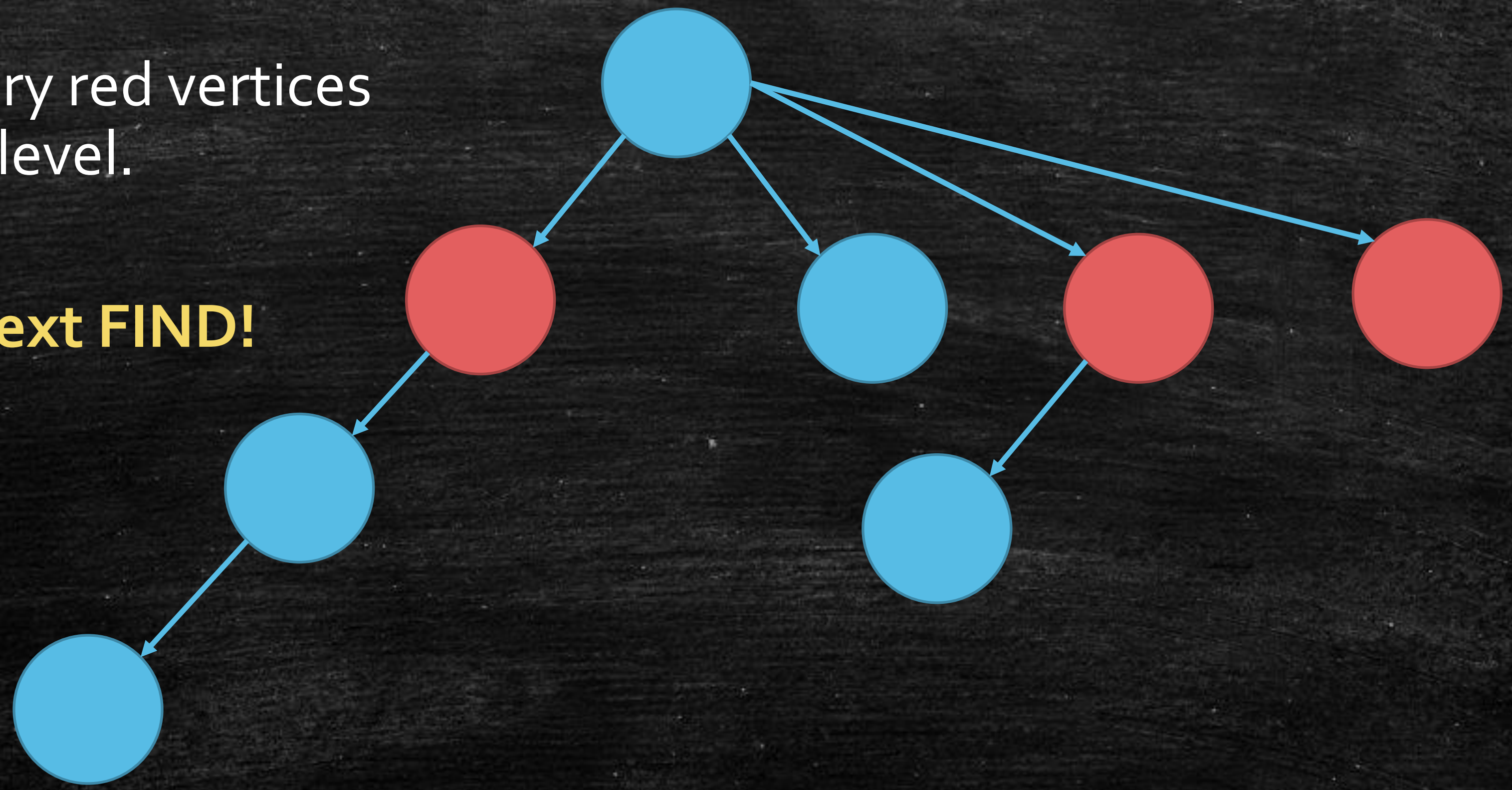


# Path Compression

**FIND**

We put every red vertices to the first level.

**Good for next FIND!**



# Does it useful in analysis?

Amortized Analysis

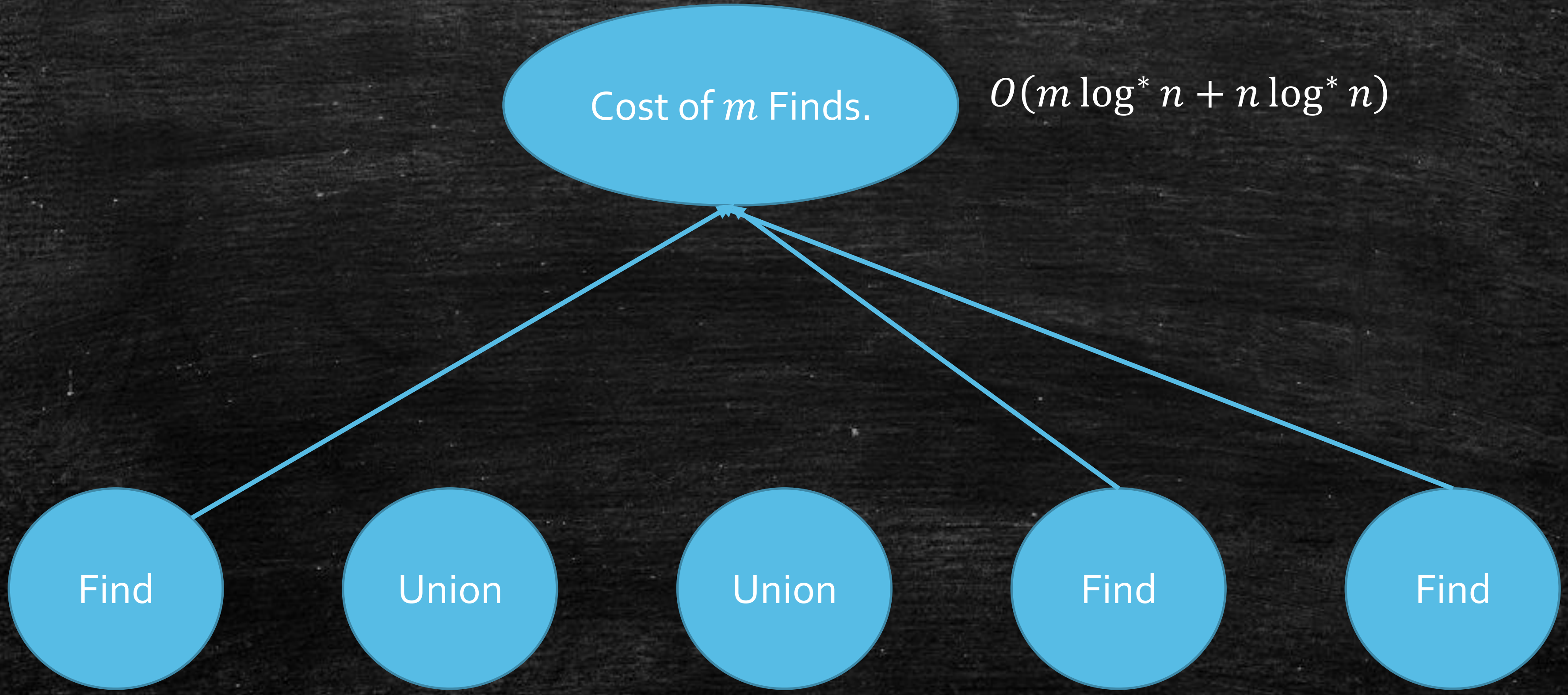
# Time Complexity

- Find (Path Compression)
  - $\hat{C} = O(\log^* n)$ [Hopcroft & Ullman 1973]
  - $\log^*(2^{2^{2^{2^2}}}) = \log^*(2^{65536}) = 5$
  - $\hat{C} = O(\alpha(n))$ [Tarjan 1975]
  - $\alpha(n)$ is the inverse Ackermann function $\alpha(9876!) = 5$.
- Union (rank based)
  - $O(1)$
- They hold when #Finds $\geq n$.

# Rank Based Union + Find with Path Compression

- It is still an amortized analysis
- We prove:
  - Any $m$ find operations totally cost $O(m \log^* n + n \log^* n)$.
- The $n \log^* n$ cost does not increase by $m$
- We can say the amortized cost of **Find** is $O(\log^* n)$ for each operation.
- We may view $n \log^n n$ as a base cost.
- It does not matter when $m$ becomes large.

# The Big Picture




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

# We use pure charging argument today!

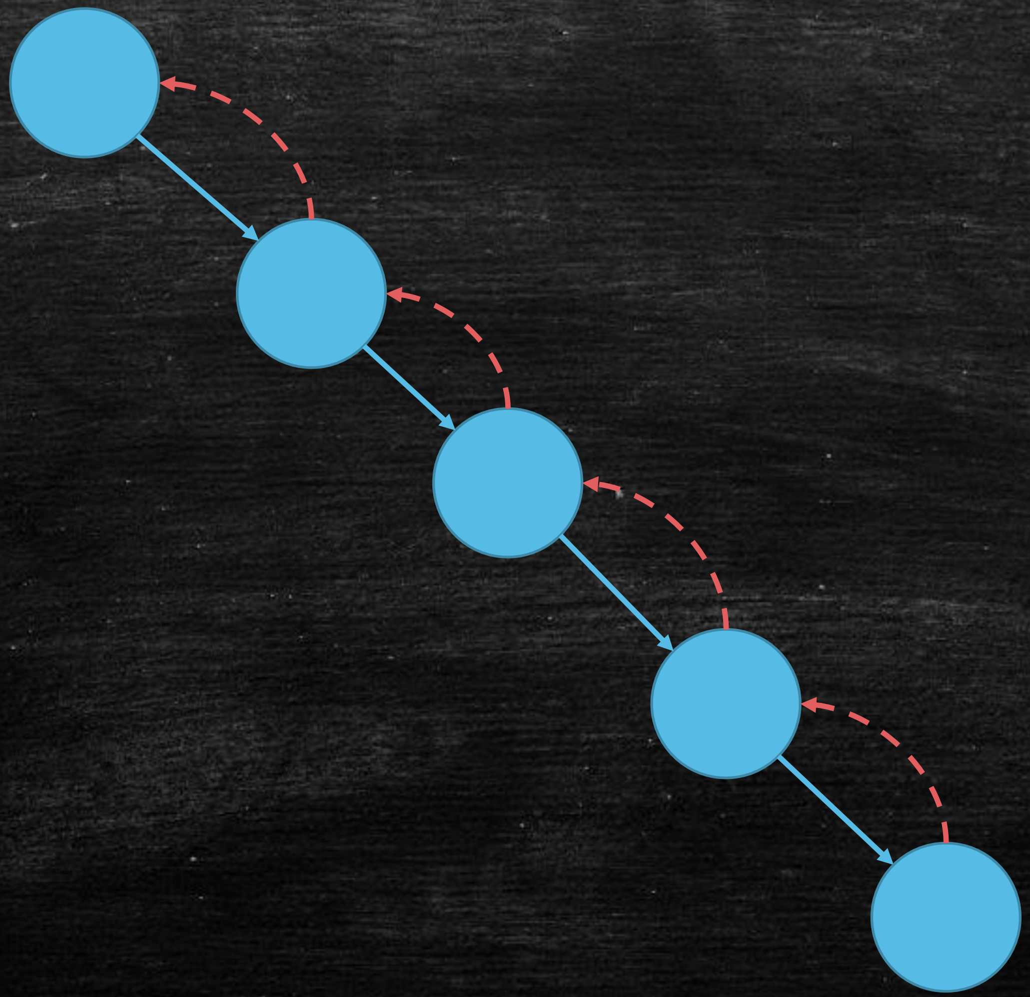
No potential functions this time.

# Basic Charging

# Find

**FIND**

Cost=number of **red edges.**

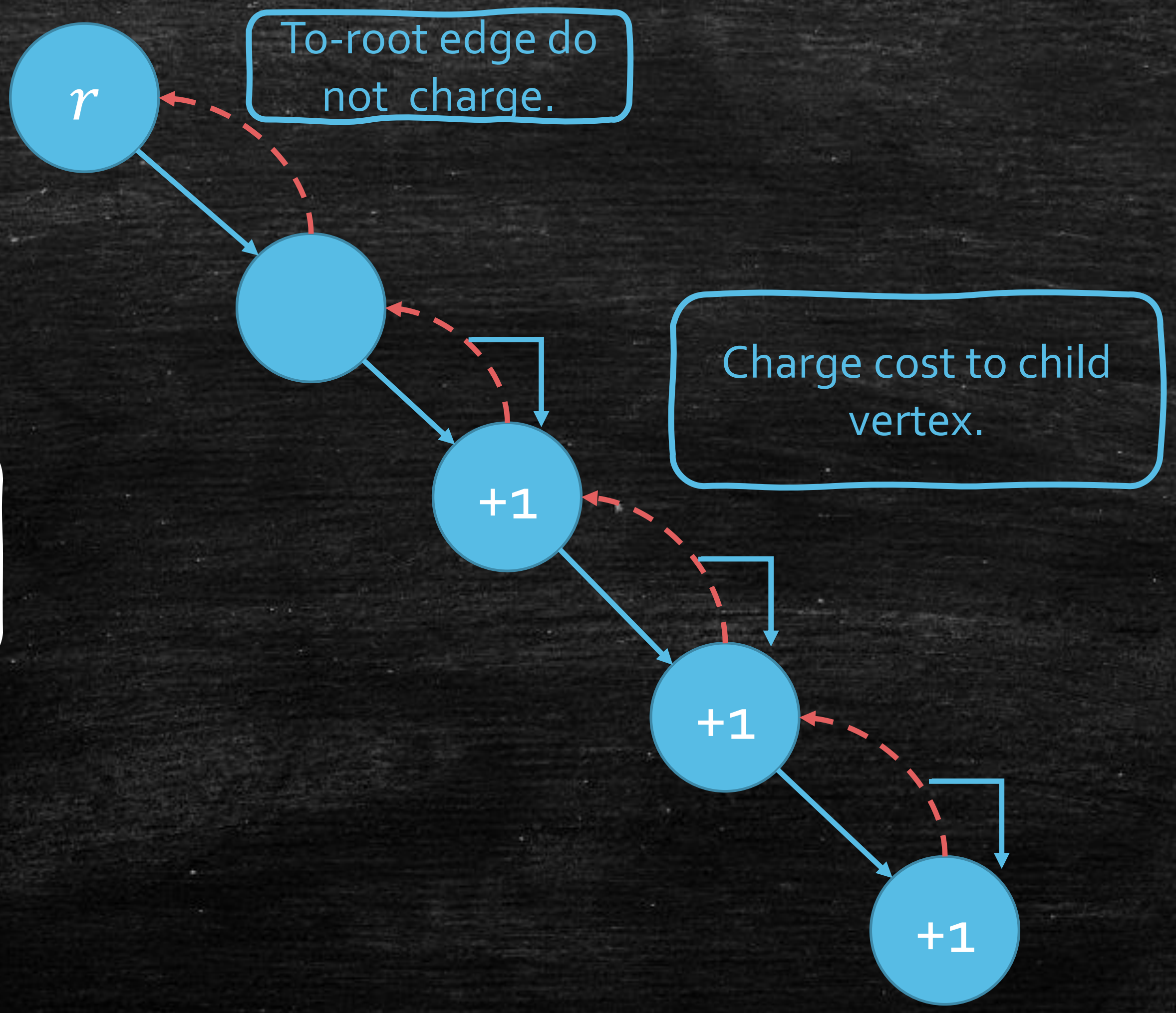


# Key Idea: Charge Cost to Vertices

**FIND**

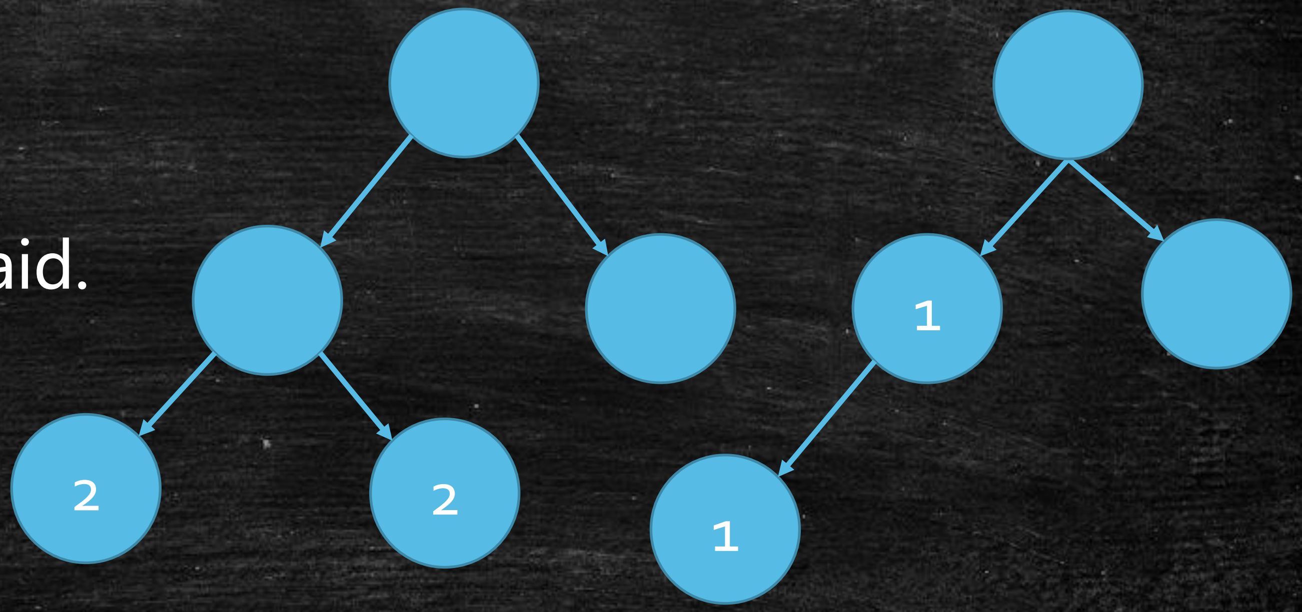
Cost=number of **red edges.**

$Cost(Find)$
- $O(1)$
- Charged Cost.

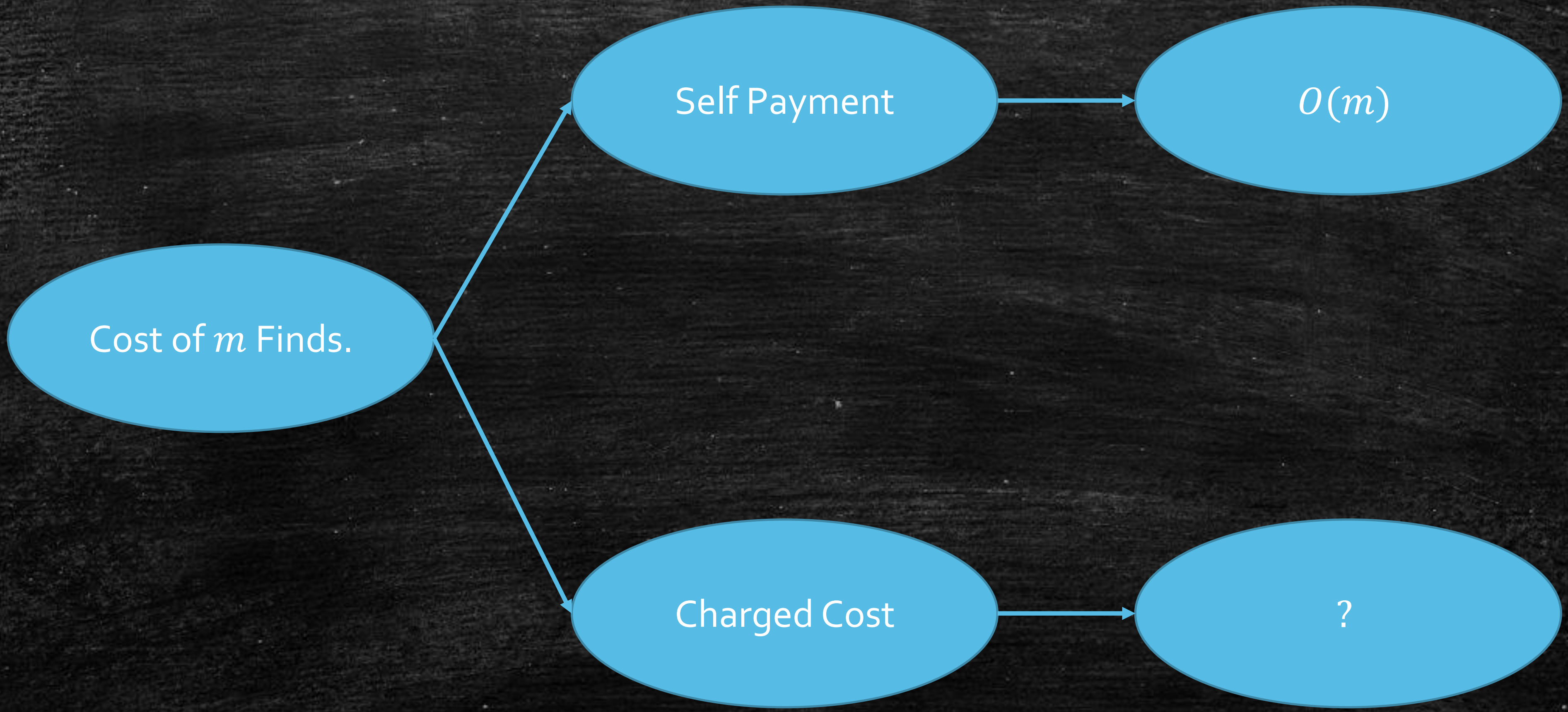


# Total cost of $m$ Finds.

- Total cost of $m$ Finds.
  - Self Payment: $O(m)$
  - Charging Cost: $\sum_v C(v)$
- $C(v)$: The cost $v$ has paid.



# The Big Picture



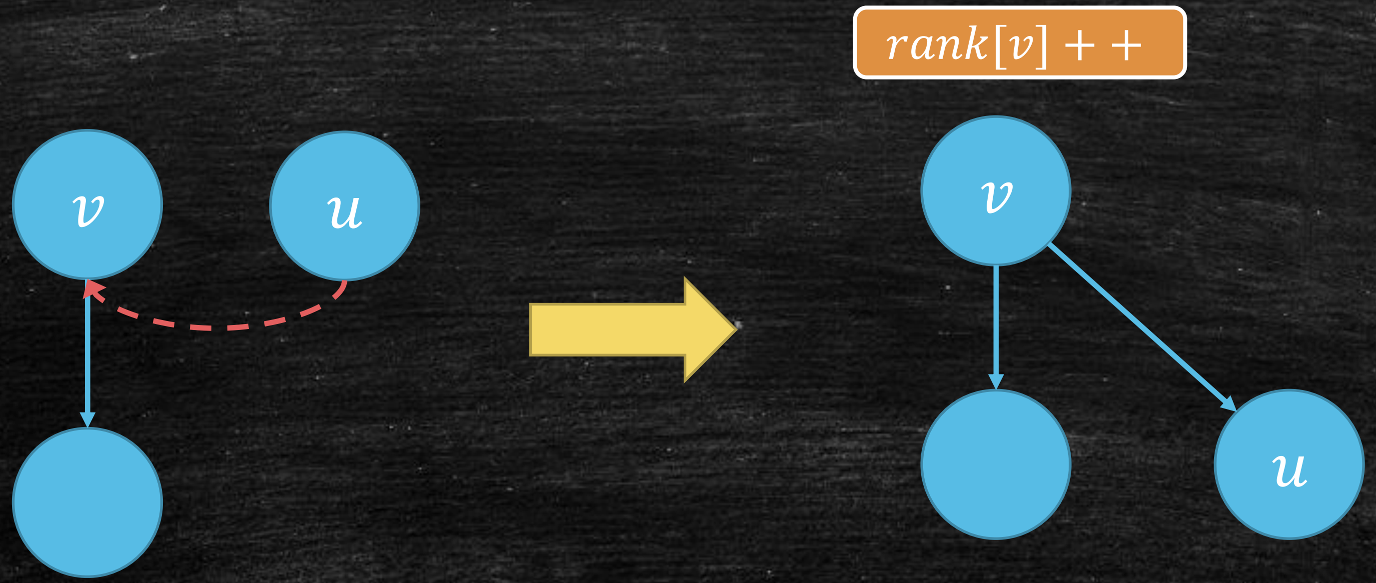
# How much each vertex will be charge?

# What is the rank now?

We do not update rank when we do path compression.

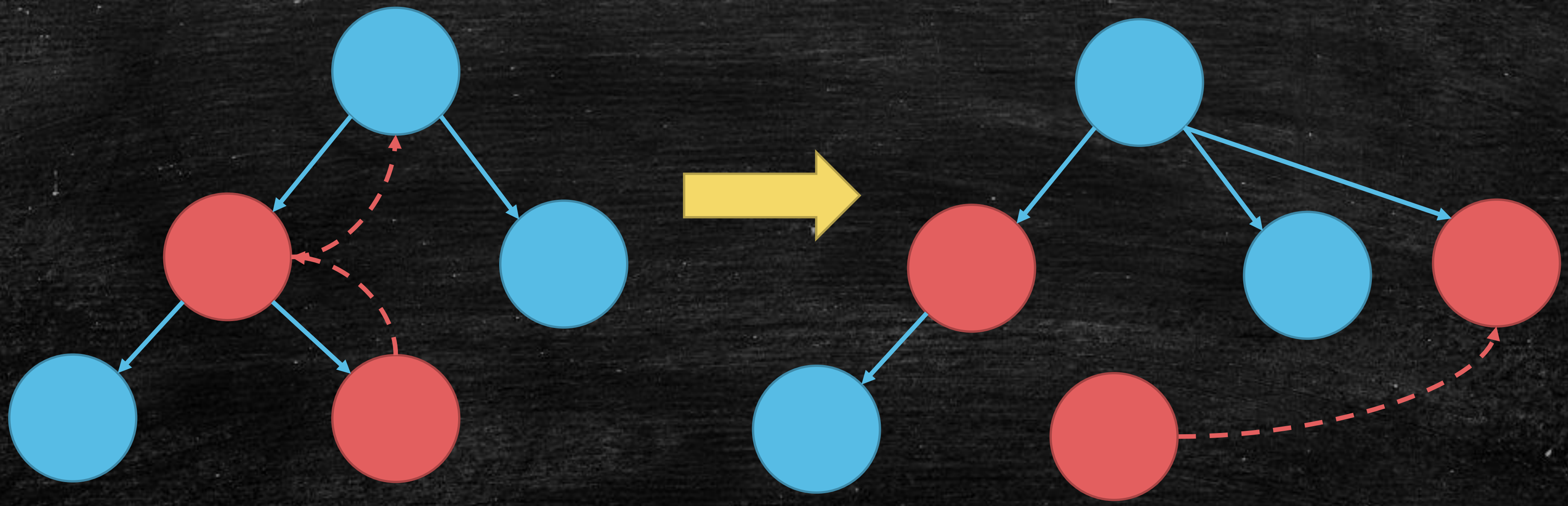
# When do we update $rank[v]$?

- We update rank when we merge two tree.



# When do we update $rank[v]$?

- We do not update any rank in path compression.



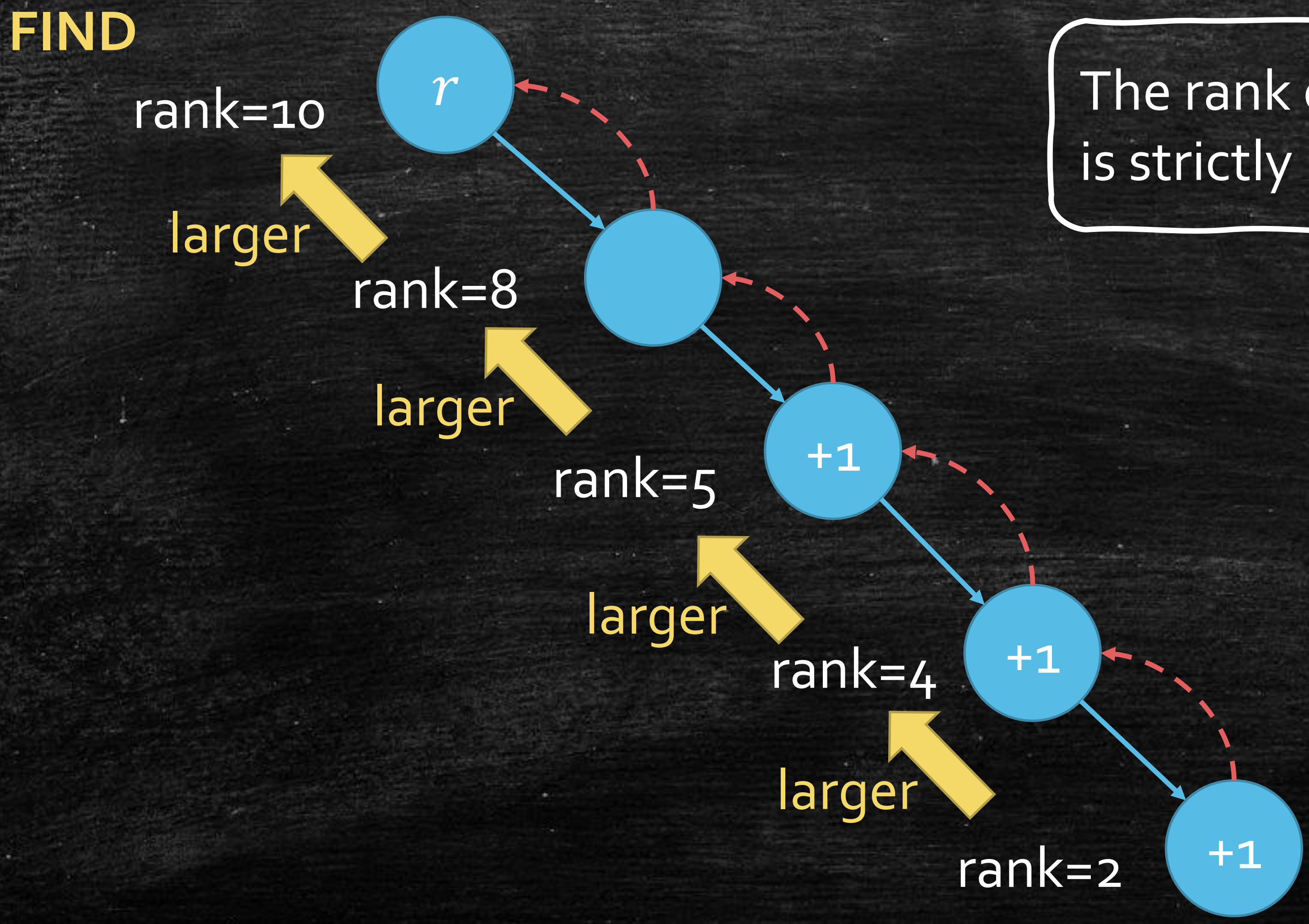
# Some Difference

- $rank[v] \neq height[v]$
  - Because we may change the height in path compression.
- $rank[v] \geq height[v]$.
  - Because we only decrease tree's height.

# We still have good properties!

- **Lemma 1.** Parent's rank is strictly larger than the child.
- Proof
  - We only merge small rank to large rank。
  - If we merge two same root, the new root's rank will $+1$.
  - Path compression only make parent's rank **larger**!
- **Lemma 2.** The tree of $v$ has at least $2^{rank[v]}$ vertices.
- Proof
  - It is because we union by rank.
  - Path compression never remove some vertices from a tree.
  - (*remark) It may remove vertices from a subtree.

# Key Fact in FIND




The rank on the path is strictly increasing.

# Do Some Amazing Analysis!

# Group Vertices

- Group vertices by rank (final)
  - Group 1: $k_1 = 0$
  - Group 2: $k_2 = 1$
  - Group $i$: $k_i = 2^{k_{i-1}}$

$k$

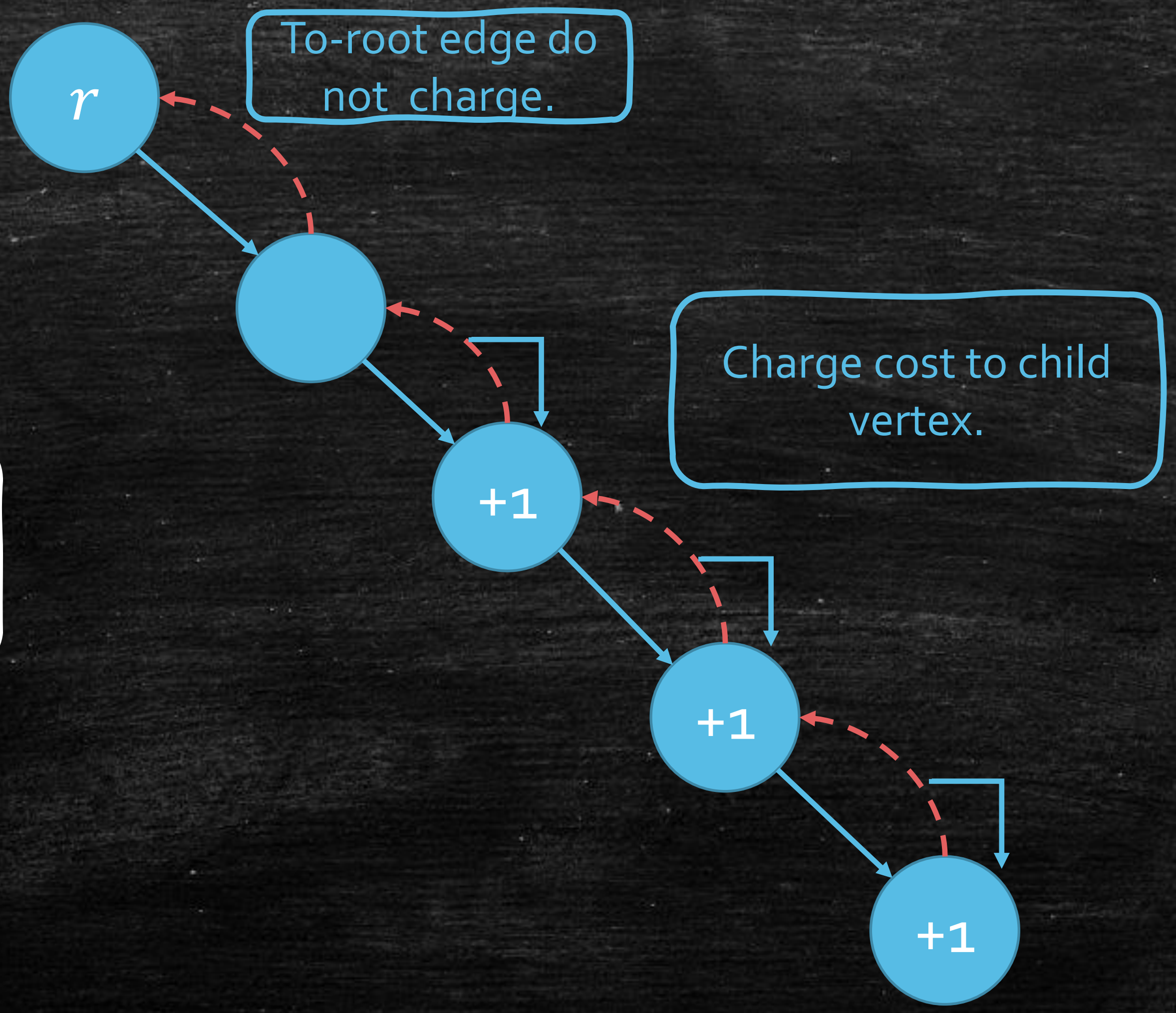
| | | | | | |
|---|---|---|---|---|---|
| 1…1 | 2…2 | 3…4 | 5…16 | 17…65536 | $k$+1… $2^k$ |

# Move back to charging.

**FIND**

Cost=number of **red edges.**

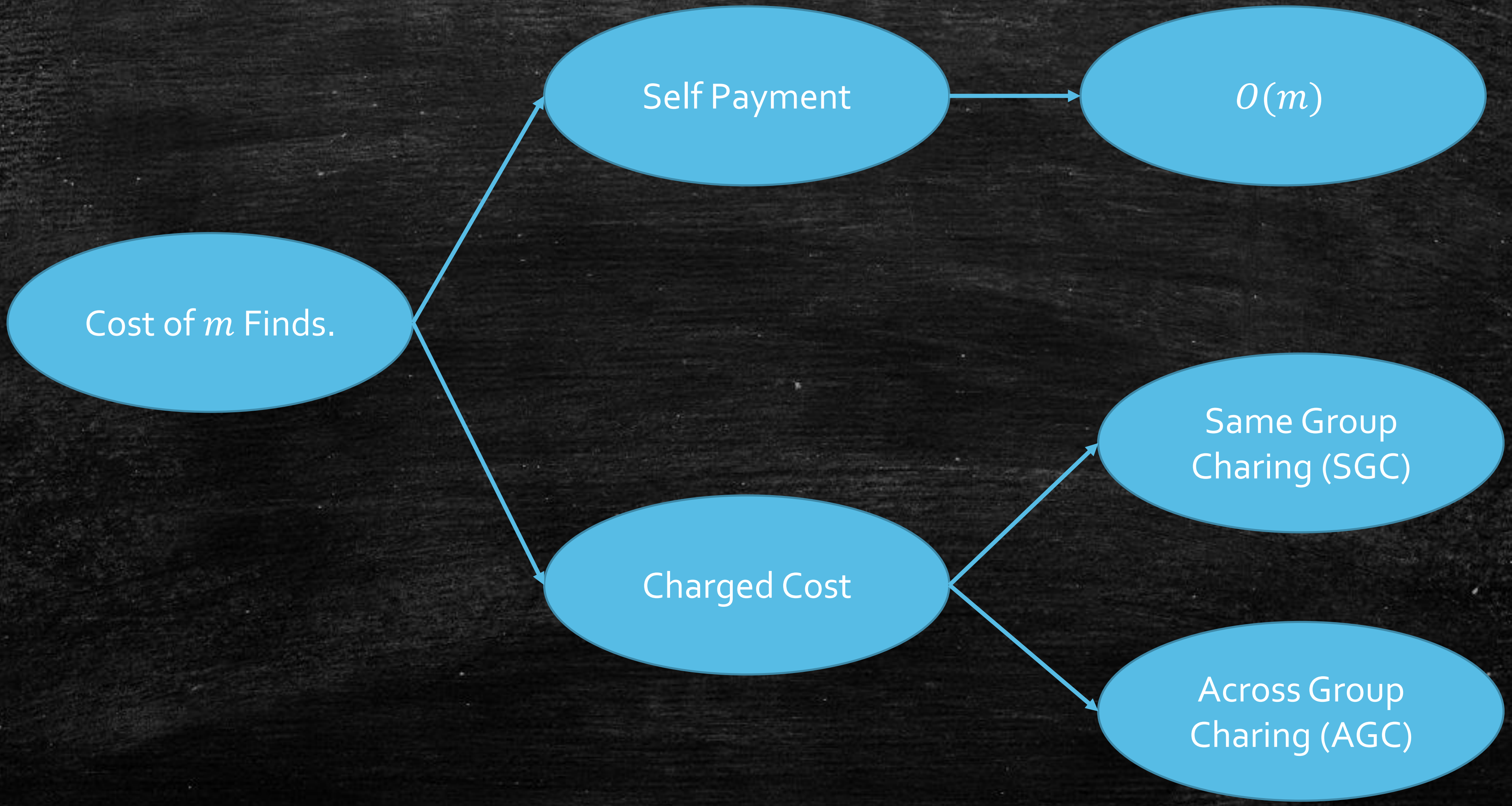
$Cost(Find)$
- $O(1)$
- Charged Cost.



# Different kind of charging.

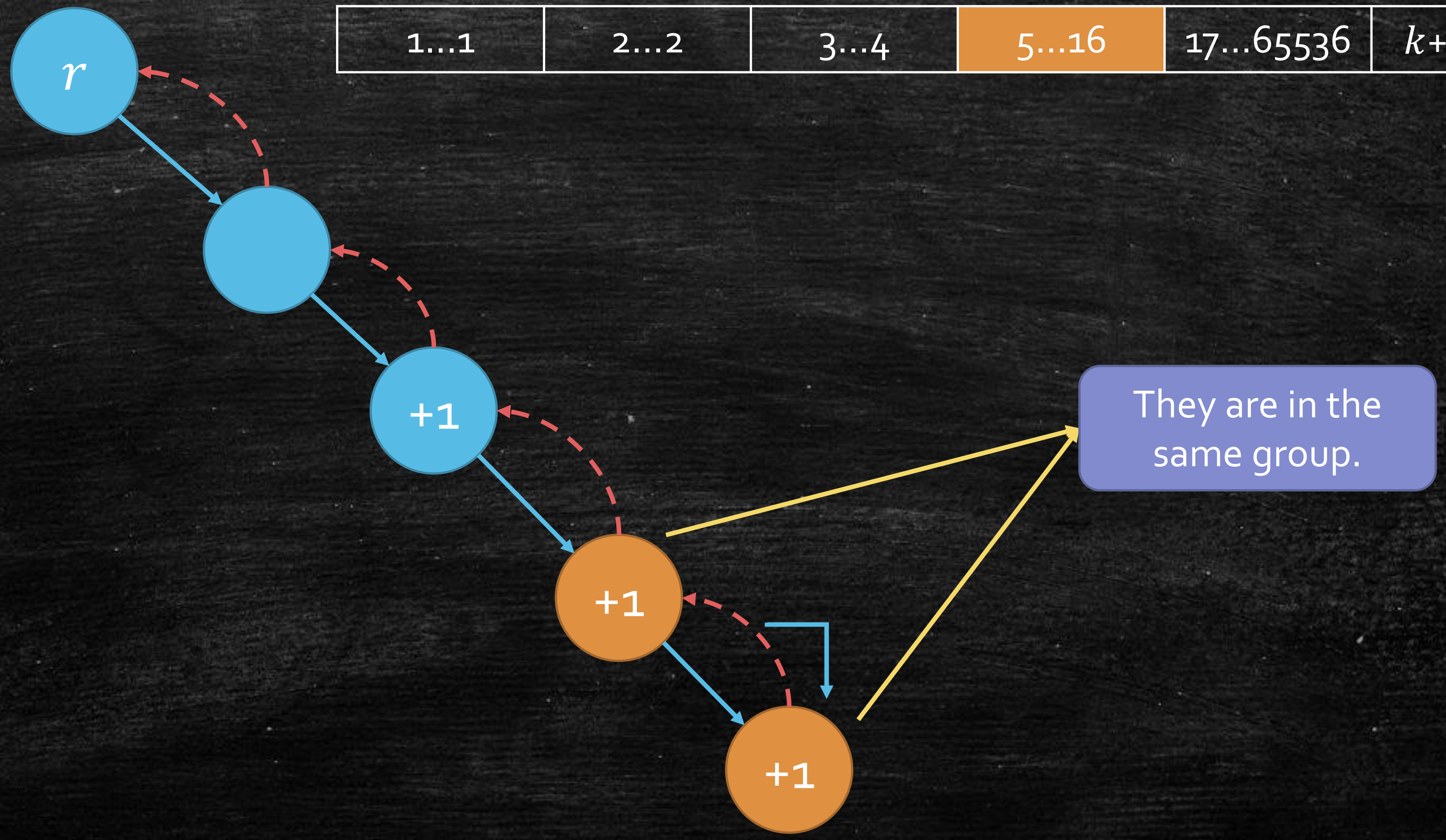
- Two kind of charging
  - Same Group Charging (SGC)
  - Across Group Charing (AGC)

# The Big Picture



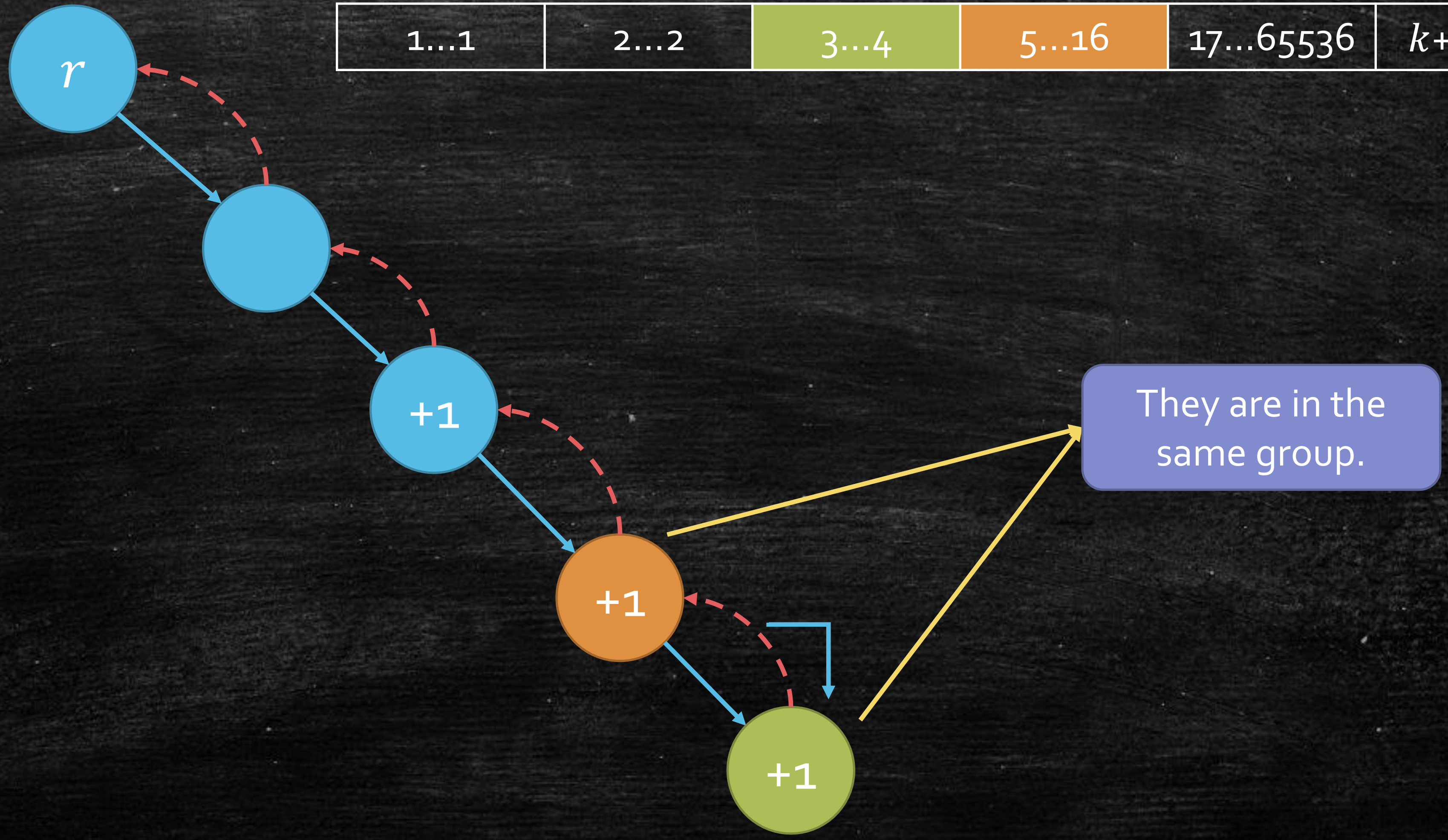
# Same Group Charging (SGC)

| | | | | | |
|---|---|---|---|---|---|
| 1…1 | 2…2 | 3…4 | 5…16 | 17…65536 | $k$+1… $2^k$ |



# Across Group Charing (AGC)

| | | | | | |
|---|---|---|---|---|---|
| 1…1 | 2…2 | 3…4 | 5…16 | 17…65536 | $k$+1… $2^k$ |



# Group Number

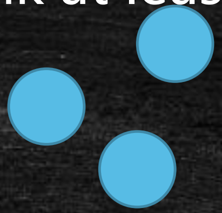
- **Lemma 3.** We have at most $\log^* n$ groups.
- Proof
  - The largest rank is at most $\log n$.
  - The last group is $[k = \log n, 2^k]$.
  - How many groups before $[k = \log n, 2^k]$
  - $k_i = 2^{k_{i-1}} \to k_{i-1} = \log k_i$
  - $k_{\log^* n} = \log n$

| | | | | | |
|---|---|---|---|---|---|
| 1…1 | 2…2 | 3…4 | 5…16 | 17…65536 | $k$+1… $2^k$ |

$\log^* n$

# Vertices in a group

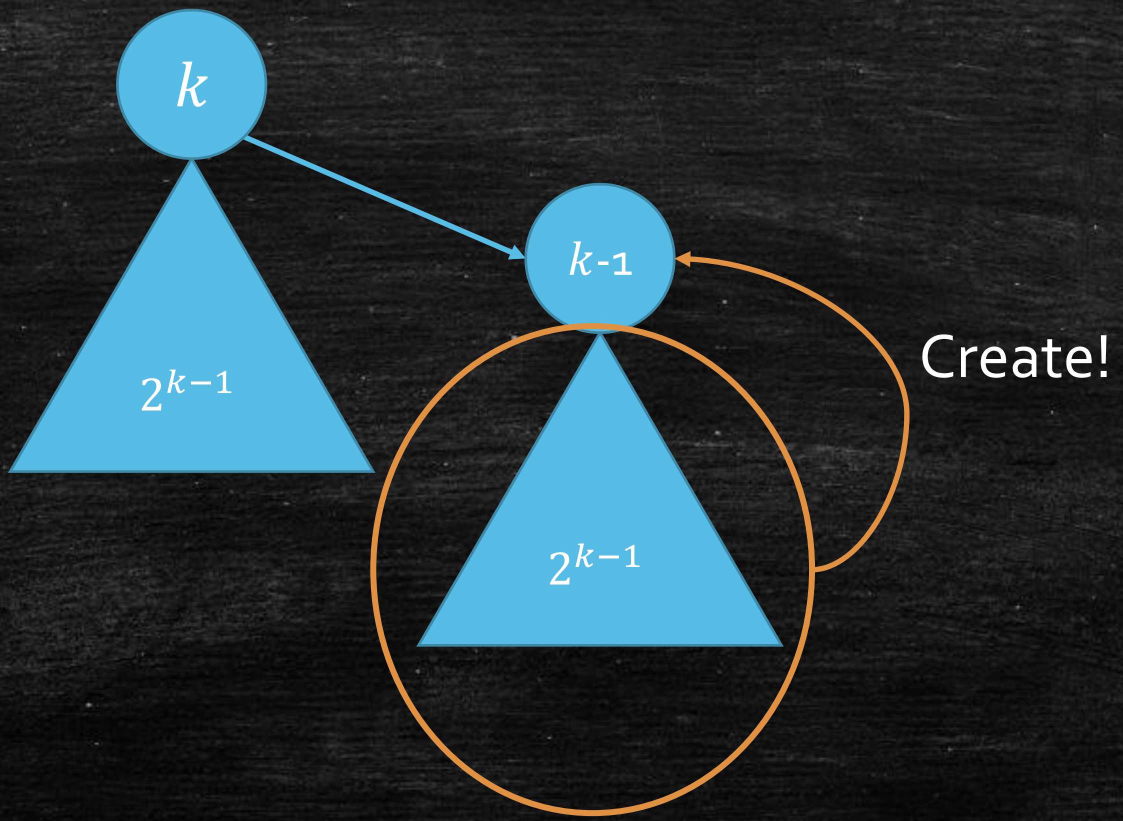
- **Lemma 4.** Group $[k+1, 2^k]$ at most have $n/2^k$ vertices.
- A wrong proof
  - All vertices inside has rank at least $k$.
  - **By Lemma 2.** The tree of $v$ has at least $2^{rank[v]}$ vertices.
  - Creating $v$ need $2^{rank[v]}$ vertices.
  - $n$ vertices can only create $n/2^k$ vertices with rank at least $k$.



| | | | | | |
|---|---|---|---|---|---|
| 1…1 | 2…2 | 3…4 | 5…16 | 17…65536 | $k$+1… $2^k$ |

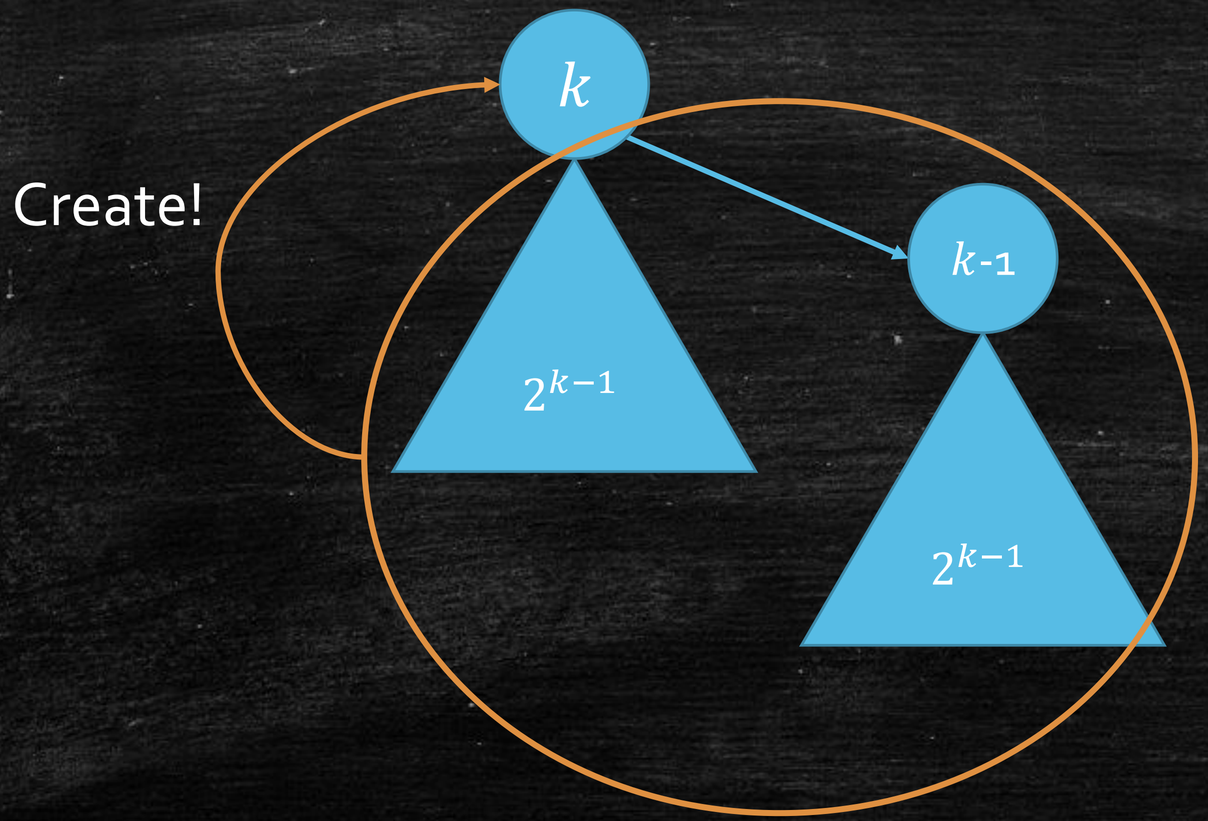
# Why it is wrong

- We may double count some vertices!



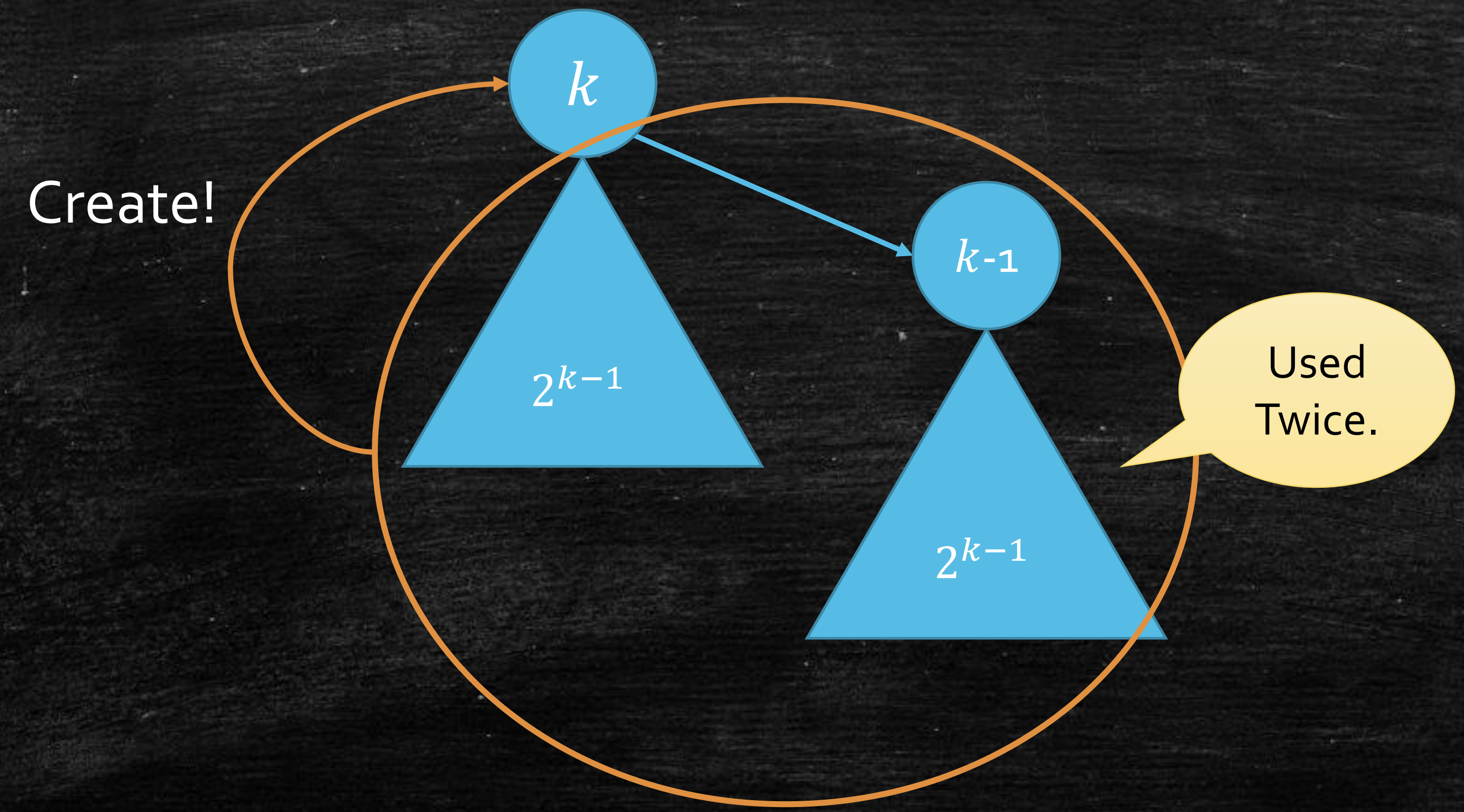
# Why it is wrong

- We may double count some vertices!



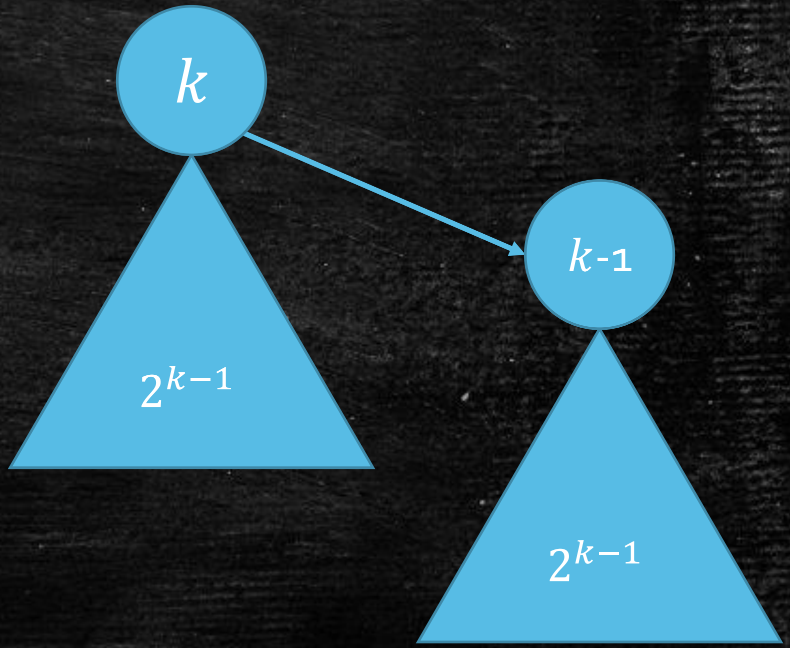
# Why it is wrong

- We may double count some vertices!



# The property we have

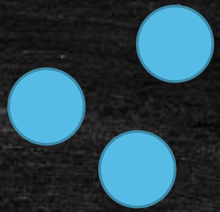
- **Fact.** The number of vertices of **exactly** rank $k$ is at most $n/2^k$.
- Proof
  - Key Fact: a vertex can only used to create **one rank $k$ root**!



# A Correct Proof

- **Lemma 4.** Group $[k+1, 2^k]$ at most have $n/2^k$ vertices.
- A Correct Proof
  - The Number of vertices in the group is at most

$$\frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \cdots \frac{n}{2^{2^k}} \le \frac{n}{2^k}$$



| | | | | | |
|---|---|---|---|---|---|
| 1…1 | 2…2 | 3…4 | 5…16 | 17…65536 | $k$+1… $2^k$ |

# Conclusion

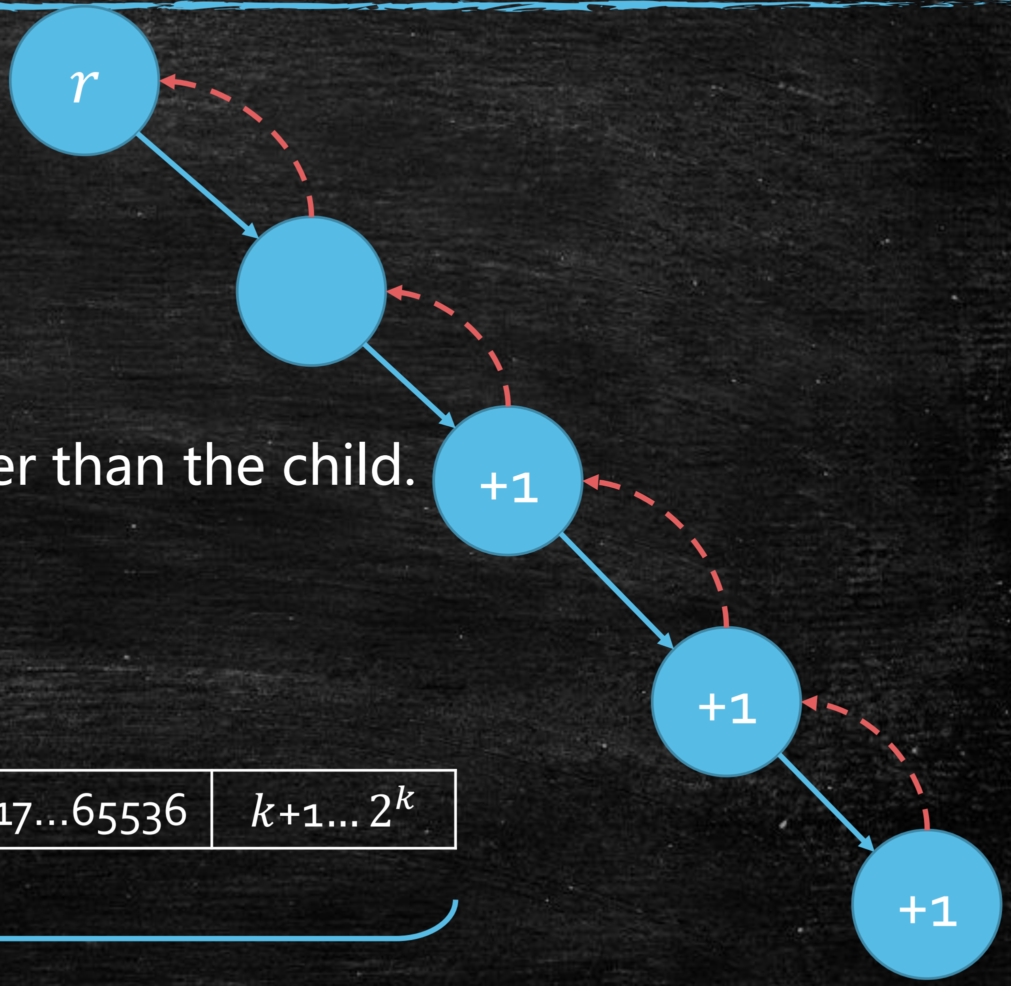
- Grouping
  - **Definition**: $v$ is in group $i$ iff $rank[v] \in [k_i + 1, 2^{k_i}]$.
  - **Lemma 3**: We have at most $\log^* n$ groups.
  - **Lemma 4**: Group $[k + 1, 2^k]$ at most have $n/2^k$ vertices.

| | | | | | |
|---|---|---|---|---|---|
| 1…1 | 2…2 | 3…4 | 5…16 | 17…65536 | $k$+1… $2^k$ |

$\log^* n$

# Across Group Charing (AGC)

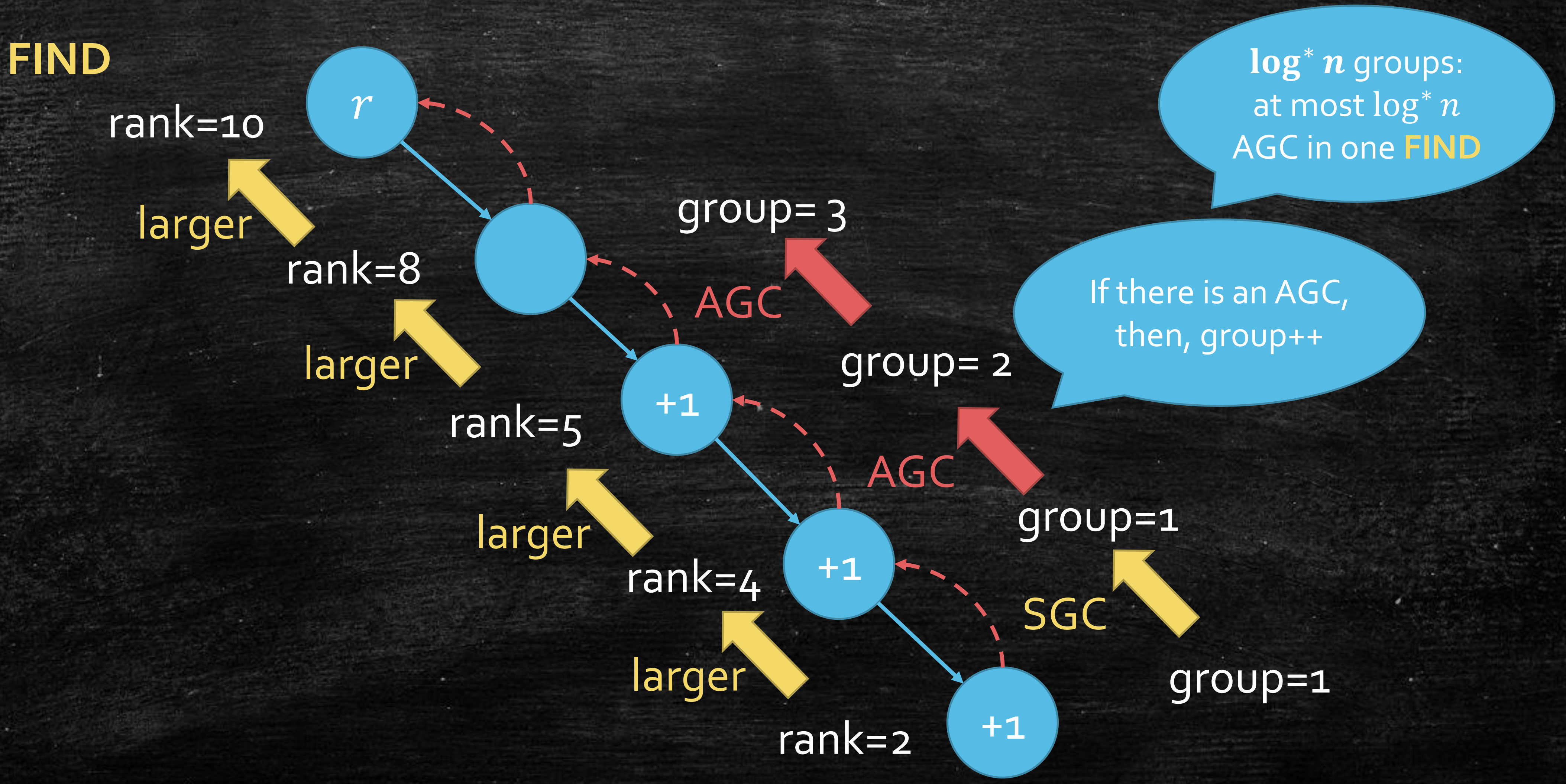
- Each Find incurs:
  - At most $\log^* n$ AGC.
- Why?
  - The rank is increasing!
  - **Lemma 1.** Parent's rank is strictly larger than the child.



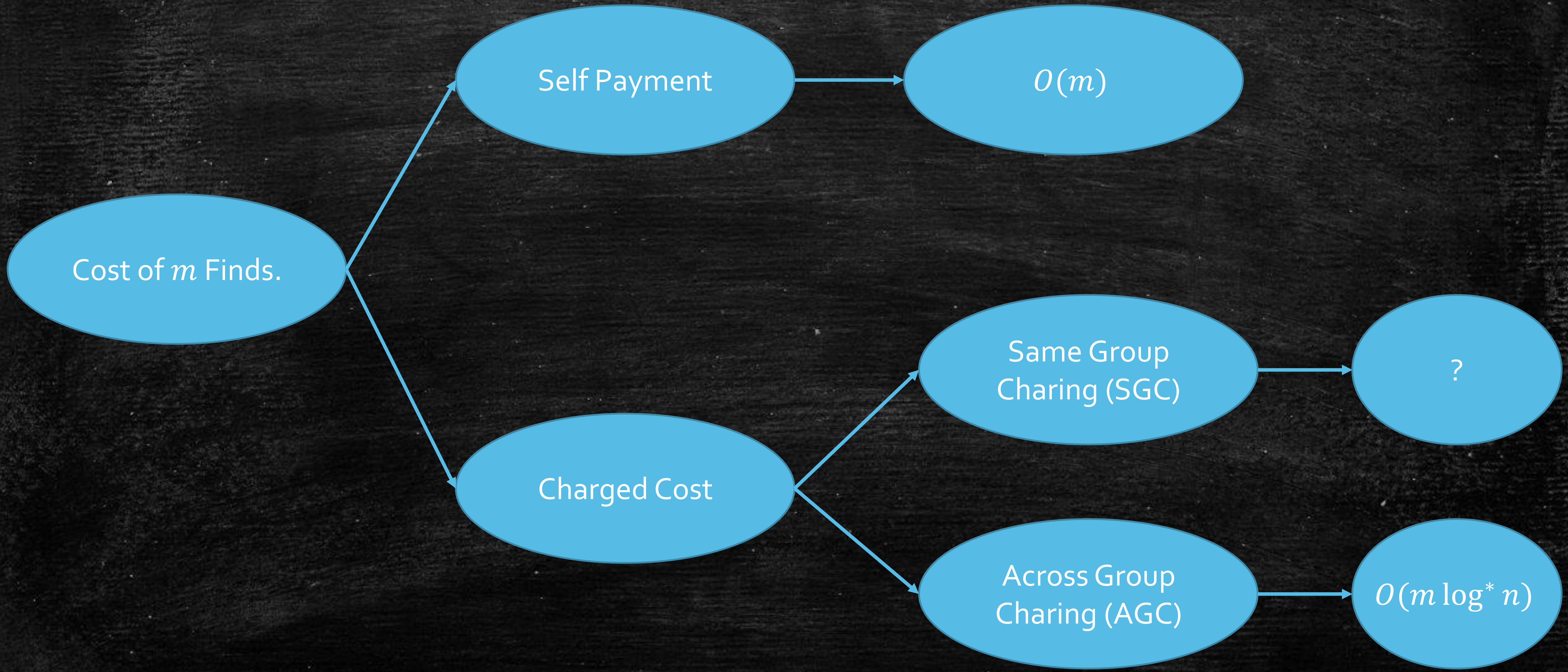

| | | | | | |
|---|---|---|---|---|---|
| 1…1 | 2…2 | 3…4 | 5…16 | 17…65536 | $k$+1… $2^k$ |

$\log^* n$

# Cost of Across Group Charing (AGC)



# The Big Picture



# What about SGC

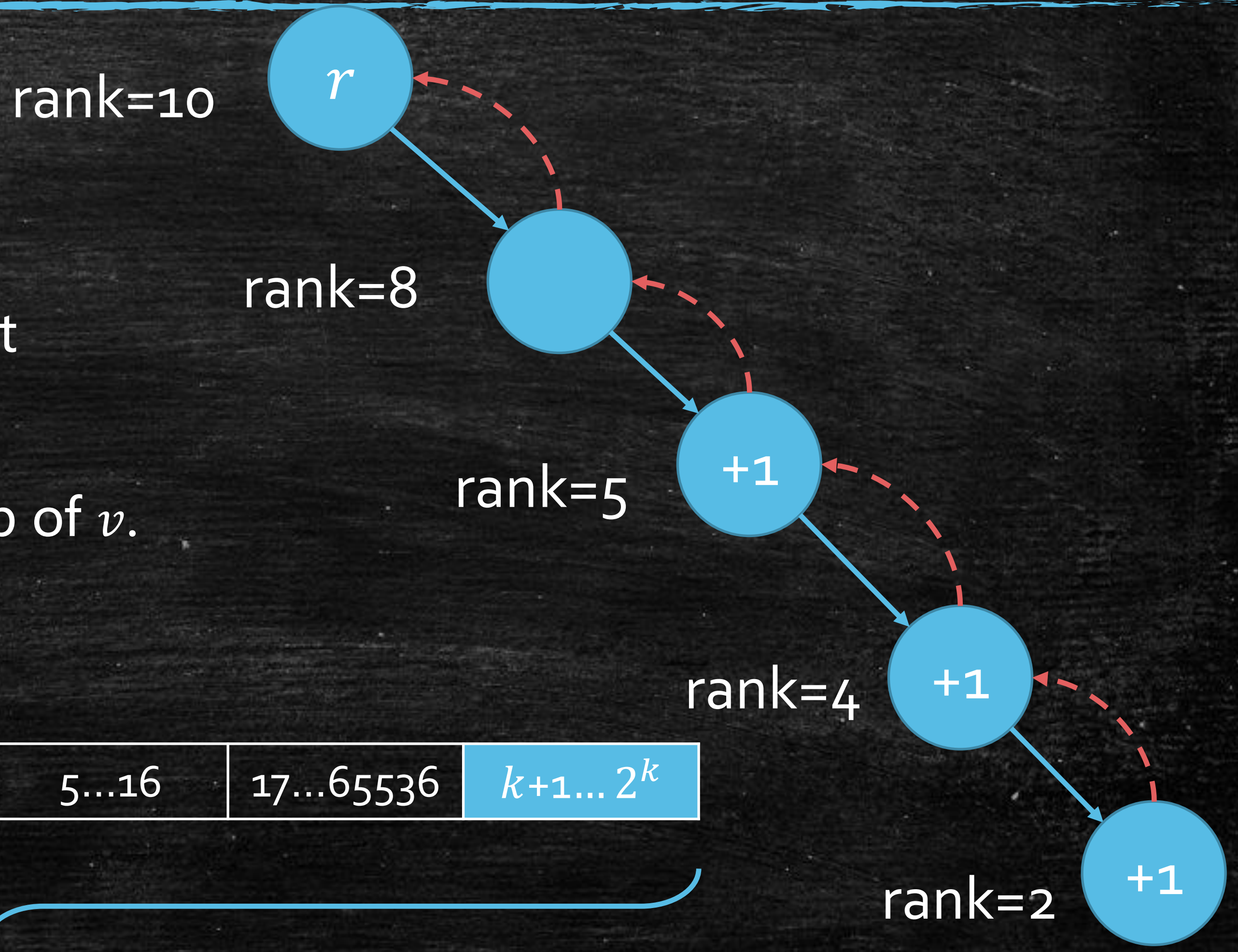
We considers each $v$ but not each Find.

# Bound Same Group Charing (SGC)

- Consider all SGC for a vertex $v$.
- We want to say each $v$ can not have so many SGC.

# Bound Same Group Charing (SGC)

- After $m$ FIND
  - $v$ have many SGC
    - By $(u1, v), (u_2, v)$ … …
  - $u_1, u_2$, … is each SGC's parent
- They are SGC
  - $u_1, u_2$ … is in the same group of $v$.

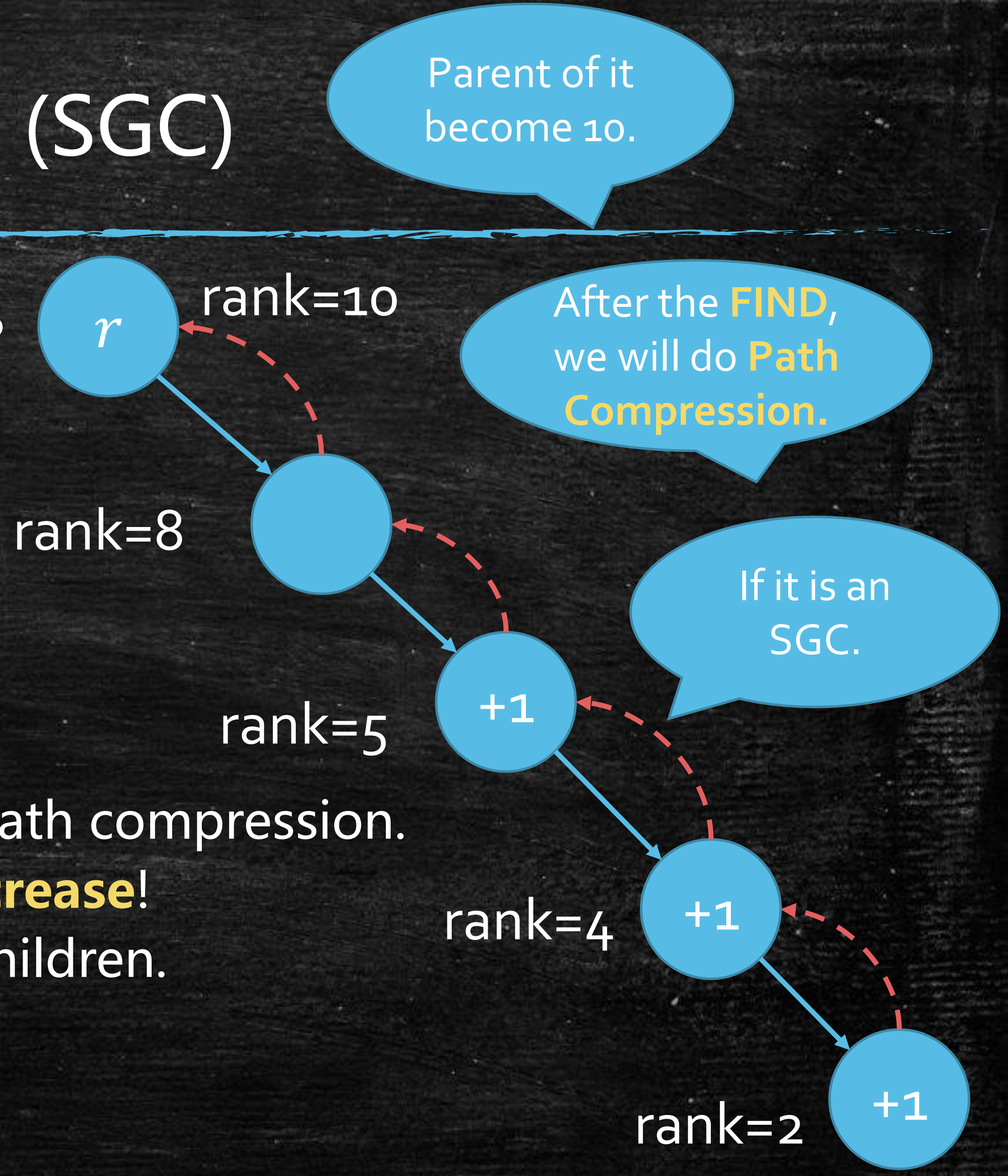



| | | | | | |
|---|---|---|---|---|---|
| 1…1 | 2…2 | 3…4 | 5…16 | 17…65536 | $k$+1… $2^k$ |

$\log^* n$

# Bound Same Group Charing (SGC)

- What are the properties of $u_1, u_2$ ... ?
- Can $rank[u_x] = rank[u_y]$?
- No!
- Path Compression
  - $rank[u_3] > rank[u_2] > rank[u_1] > rank[v]$.
  - Every time we make SGC, we also do a path compression.
  - Parent of $v$ become $r$ → Parent's rank **increase**!
  - That is why we do not charge to root's children.
- At most $2^k - (k+1) < 2^k$ SGC for $v$.

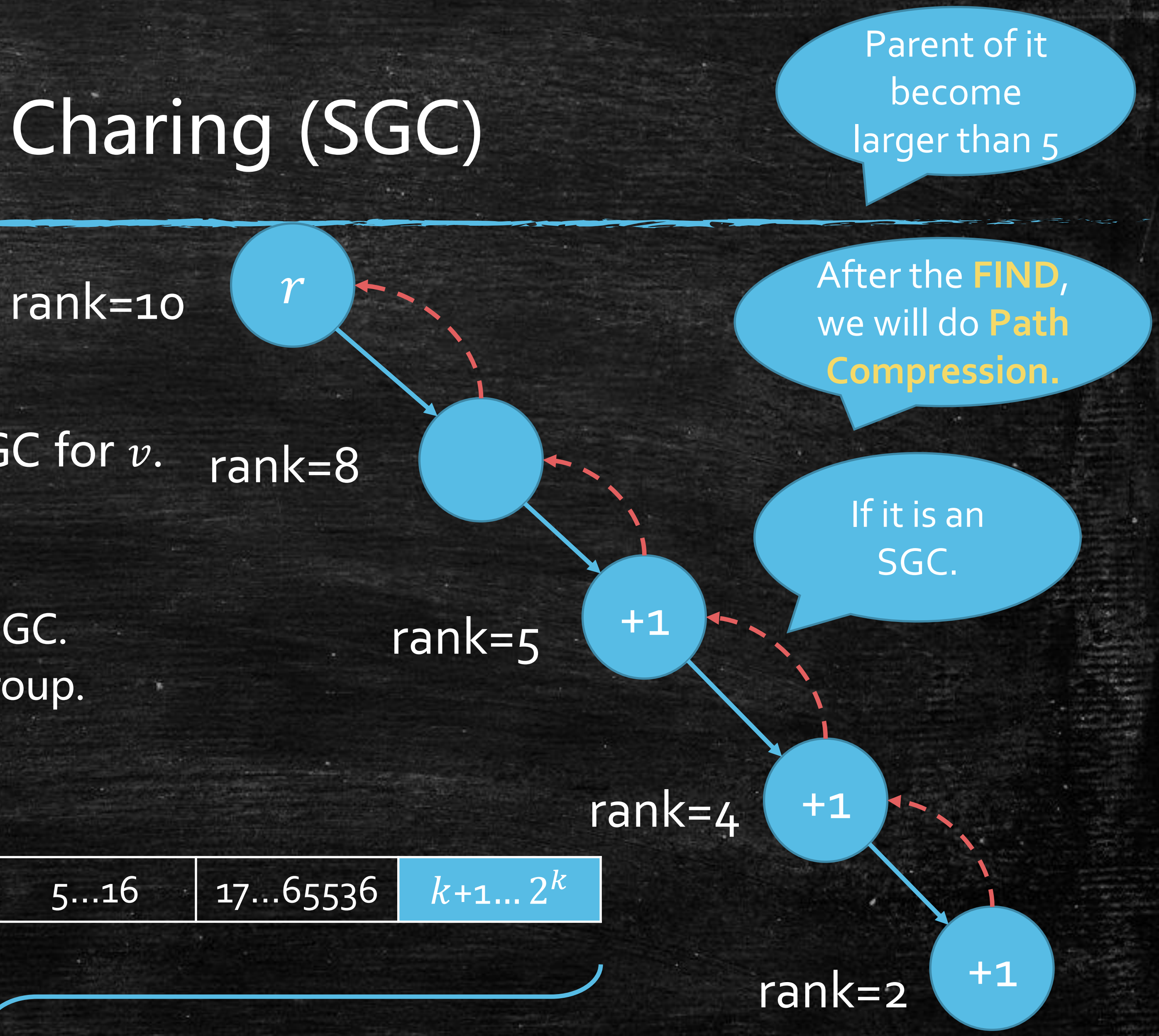


# Bound Same Group Charing (SGC)

- After $m$ FIND
  - $v$ can be SGC many times
  - At most $2^{\mathrm{k}} - (\mathrm{k}+1) < 2^{\mathrm{k}}$ SGC for $v$.
  - In a group $[k+1 \ldots 2^k]$
    - **Lemma 4:** $n/2^k$ vertices.
    - Each vertex has at most $2^k$ SGC.
    - Totally at most $n$ SGC in a group.
  - Totally: $n \log^* n$ SGC
    - **Lemma 3:** $\log^* n$ groups.

| | | | | | |
|---|---|---|---|---|---|
| 1…1 | 2…2 | 3…4 | 5…16 | 17…65536 | $k$+1… $2^k$ |

$$\underbrace{\hspace{10em}}_{\log^* n}$$



# Can you answer this question?

- Question
  - Yes, if a vertex is at a group $[k+1, 2^k]$, we at most have $2^{\mathrm{k}}$ SGC.
  - But, if the vertex's rank increase, it will change a group.
  - Why not we have more SGC for this vertex?

# Bound Total cost

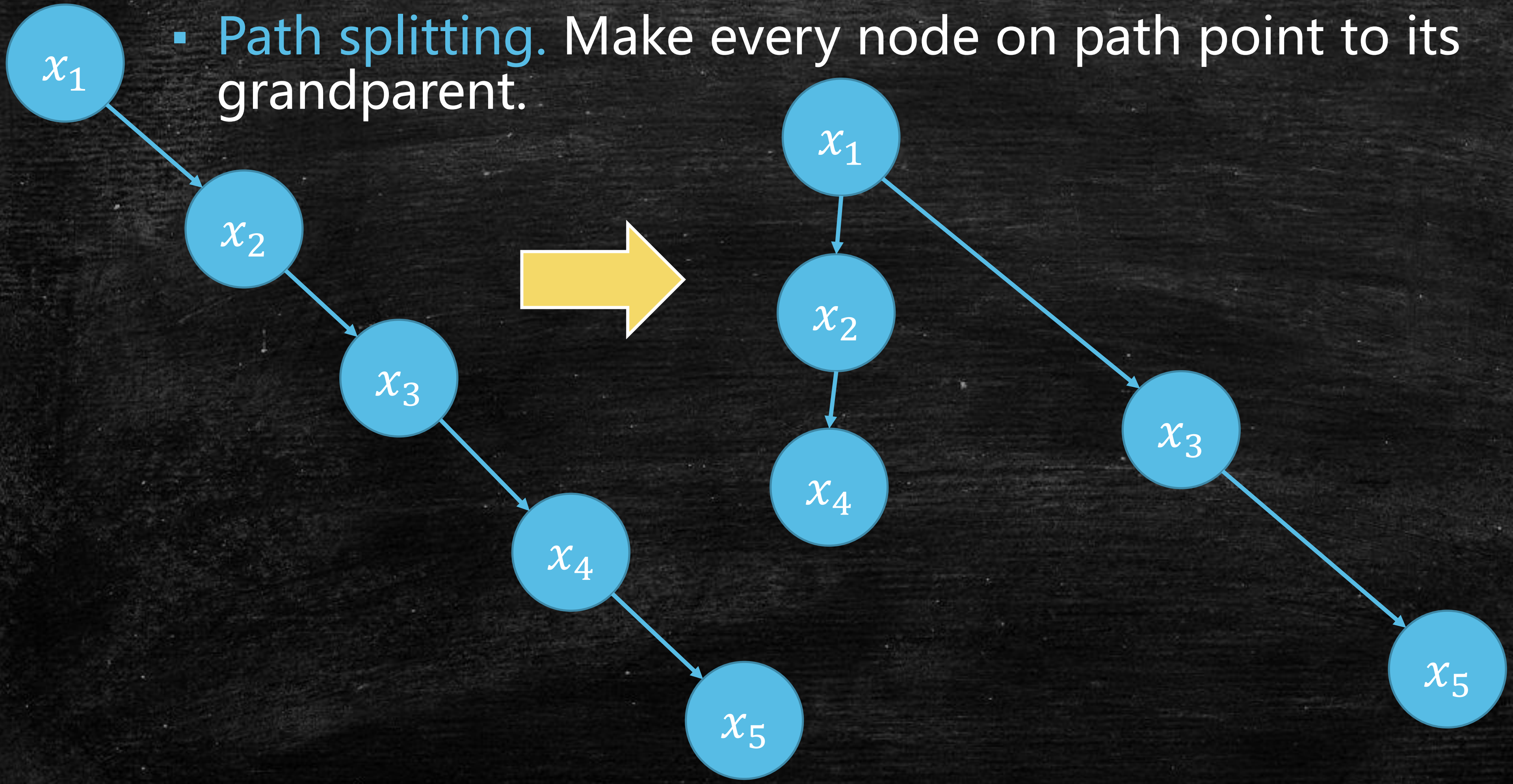
- Total Cost of $m$ FIND
  - $O(m)(to\ root)$
  - Total AGC
    - $m \cdot \log^* n$
  - Total SGC
    - $n \cdot \log^* n$
  - Total : $O(m \log^* n + n \log^* n)$

# Other union Method

- Union By Size v.s. Union By Rank
- Using Union By Size with Path Compression
- [Tarjan 1975] Any $m \geq n$ Find Operations and $n - 1$ Union operations cost $O(\alpha(m,n))$ time.
- Union by random
  - Give a rank for each vertex uniformly random in $[0,1]$.

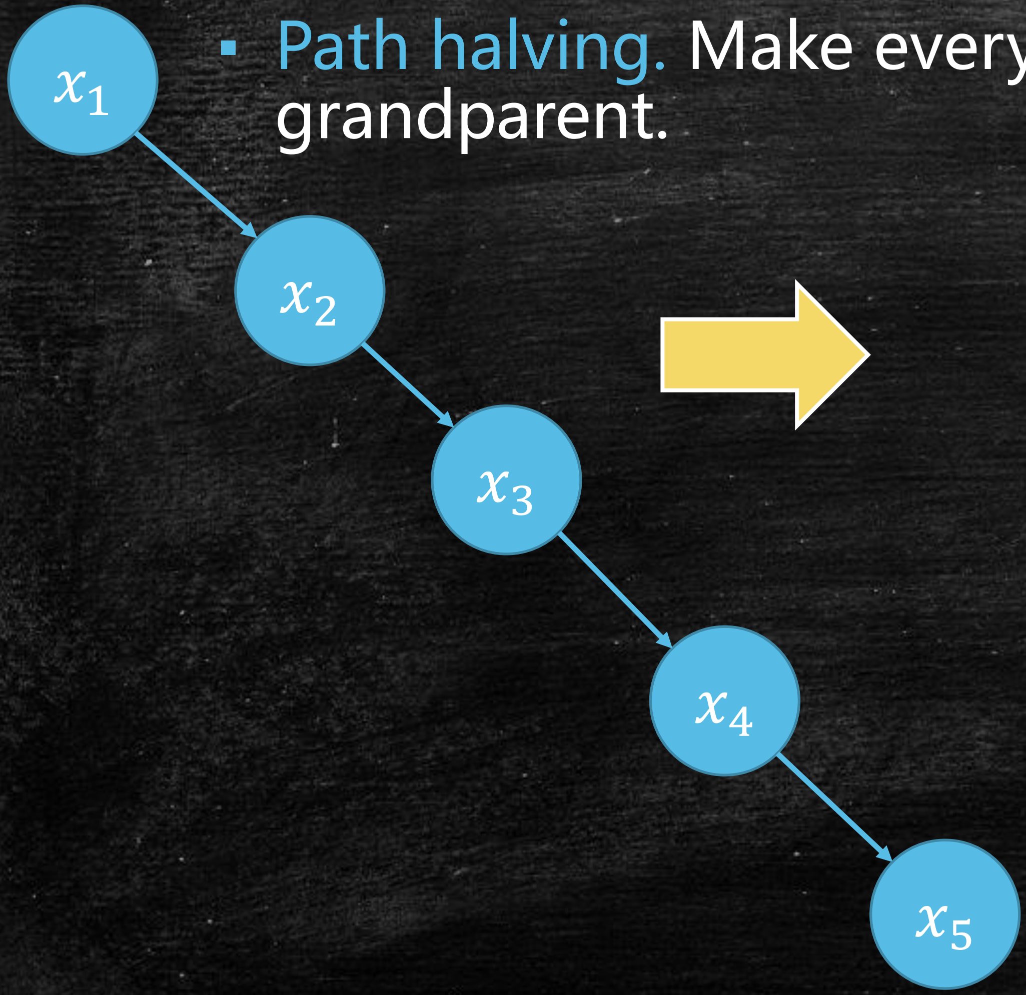
# Other Path compression

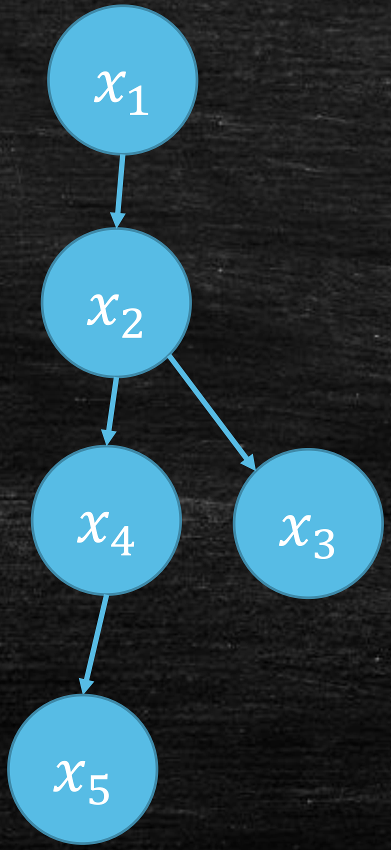
- Path splitting. Make every node on path point to its grandparent.



# Other Path compression

- Path halving. Make every even node on path point to its grandparent.



# All of them are somehow equivalent!

- **Theorem.** [Tarjan-van Leeuwen 1984]

Union by {rank,size} with {path compression, path splitting, path halving} perform $m \geq n$ Find Operations and $n-1$ Union Operations in $O(\alpha(m,n))$ time.

# Today's goal

- Learn what is **Greedy**!
- Learn to use **Greedy** to finish homework!
- Learn Prim and Kruskal!
- Again, how to use **Data Structure** to improve **Algorithms**.
- Review Union-Find Set!
- Learn another **Amortized Analysis**!