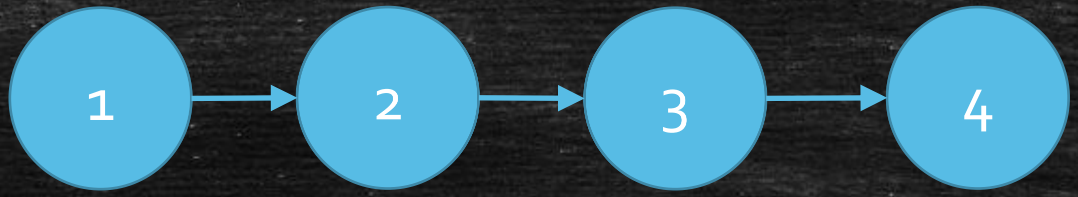
Shortest Path

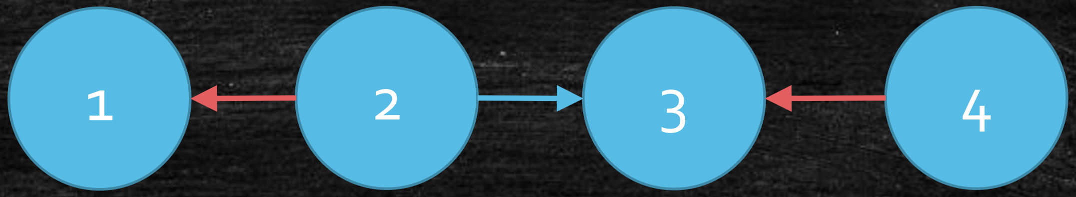
BFS and Dijkstra

What is path?

- Today we discuss directed graphs!
- 1 to 4 Path



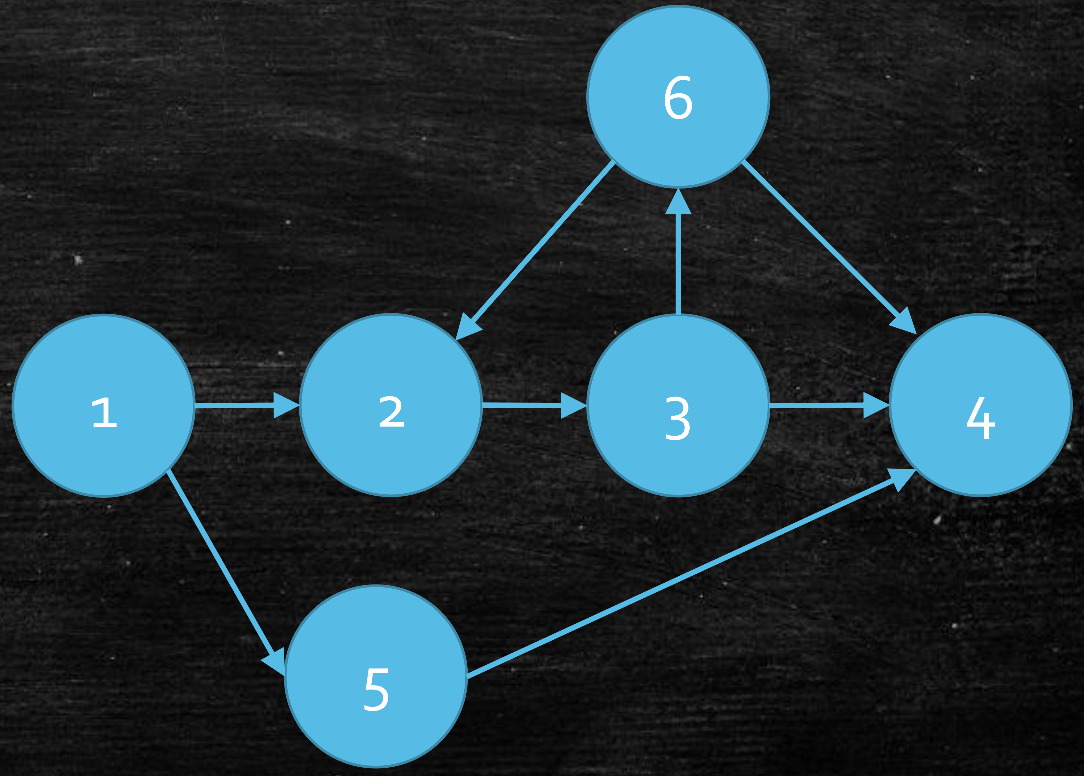
- Not a 1 to 4 path



- Length: the number of arcs in the path.

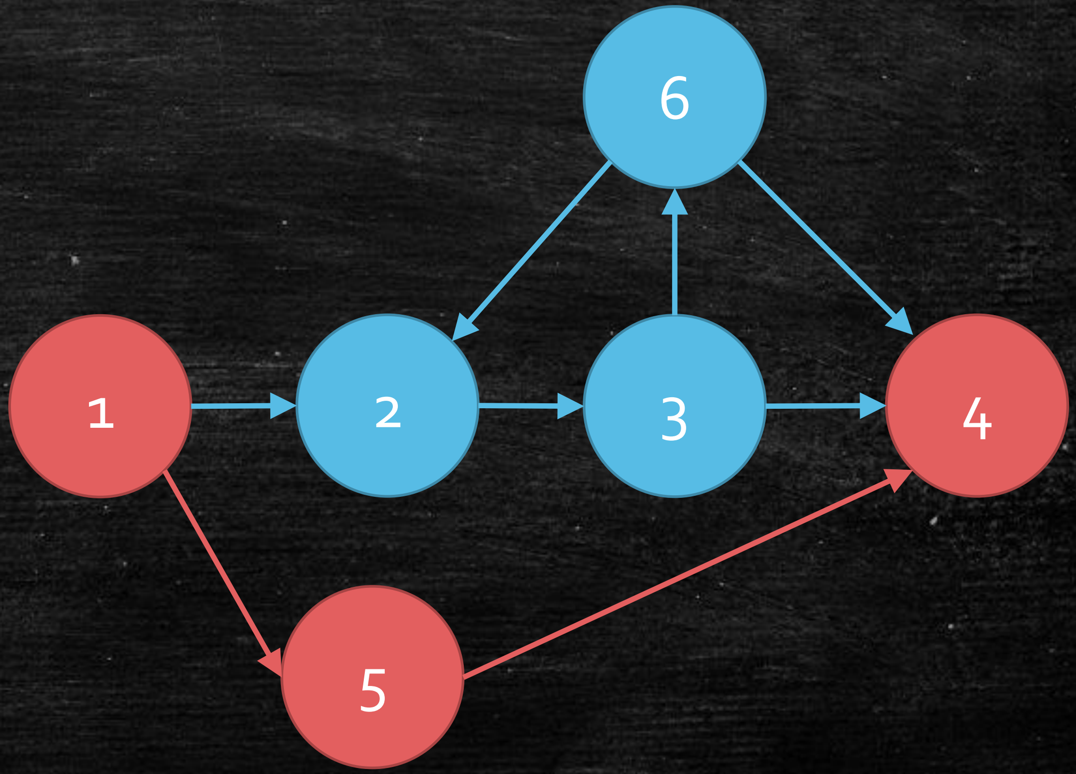
Vertices Distance

- How to define distance?
- $d(u, v)$: the length of shortest path from u to v.



Vertices Distance

- How to define distance?
- $d(u, v)$: the length of shortest path from u to v.
- $d(1,4) = 2$

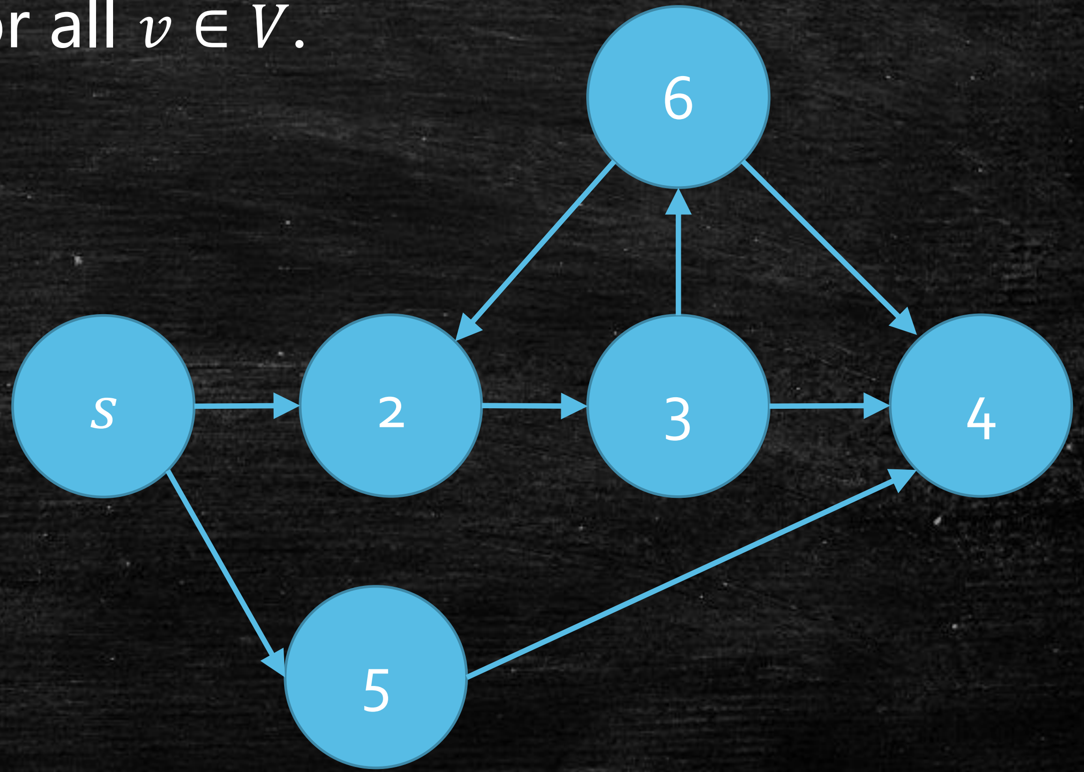


Single-Source Shortest Path Problems

- **Input:** A directed graph $G(V,E)$, represented by an Adjacent List, and a source vertex s.
- **Output:** Distance $d(s,v)$, for all $v \in V$.

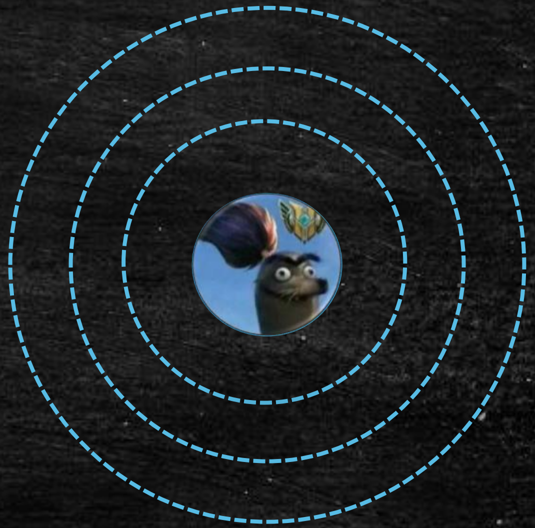
Single-Source Shortest Path Problems

- **Input:** A directed graph $G(V,E)$, represented by an Adjacent List, and a source vertex s.
- **Output:** Distance $d(s,v)$, for all $v \in V$.



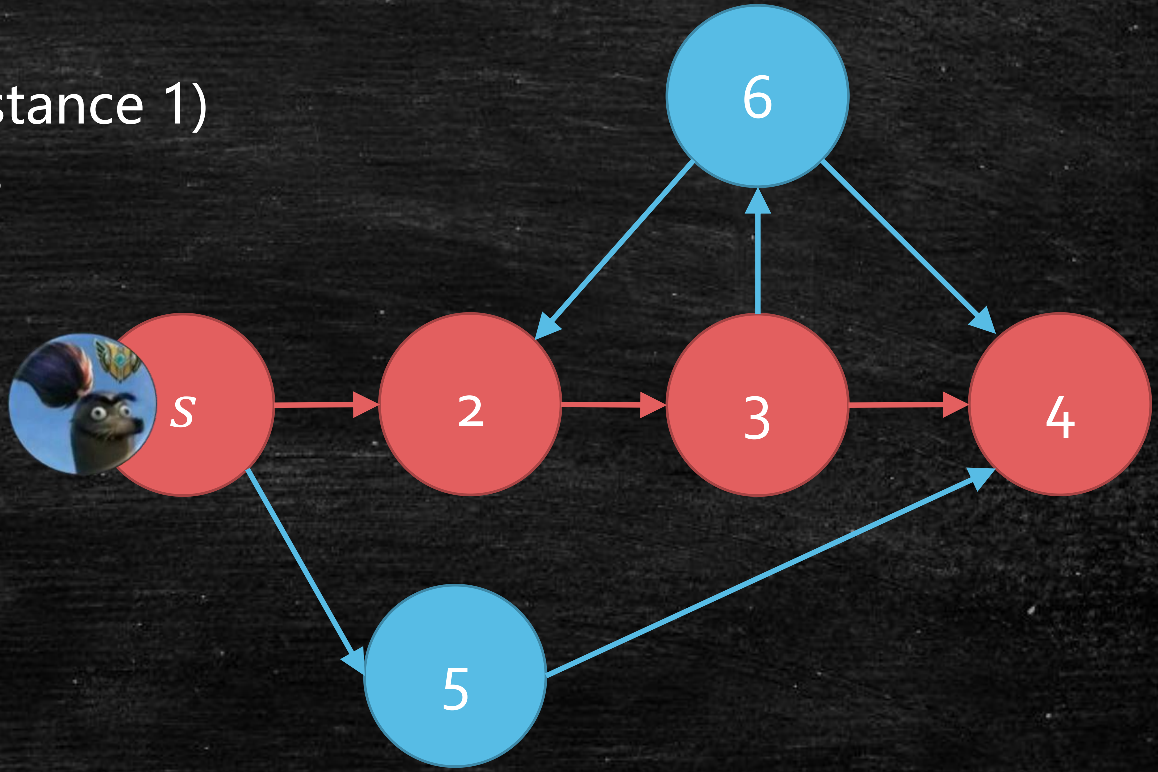
Key Idea

- **Input:** A directed graph $G(V,E)$, represented by an Adjacent List, and a source vertex s.
- **Output:** Distance $d(s, v)$, for all $v \in V$.
- Idea
  - Walk from s
  - Keep walking
  - Walk 1 step: Arrive distance 1 vertices
  - Walk 2 steps: Arrive distance 2 vertices
  - Walk 3 steps; Arrive distance 3 vertices
  - ……



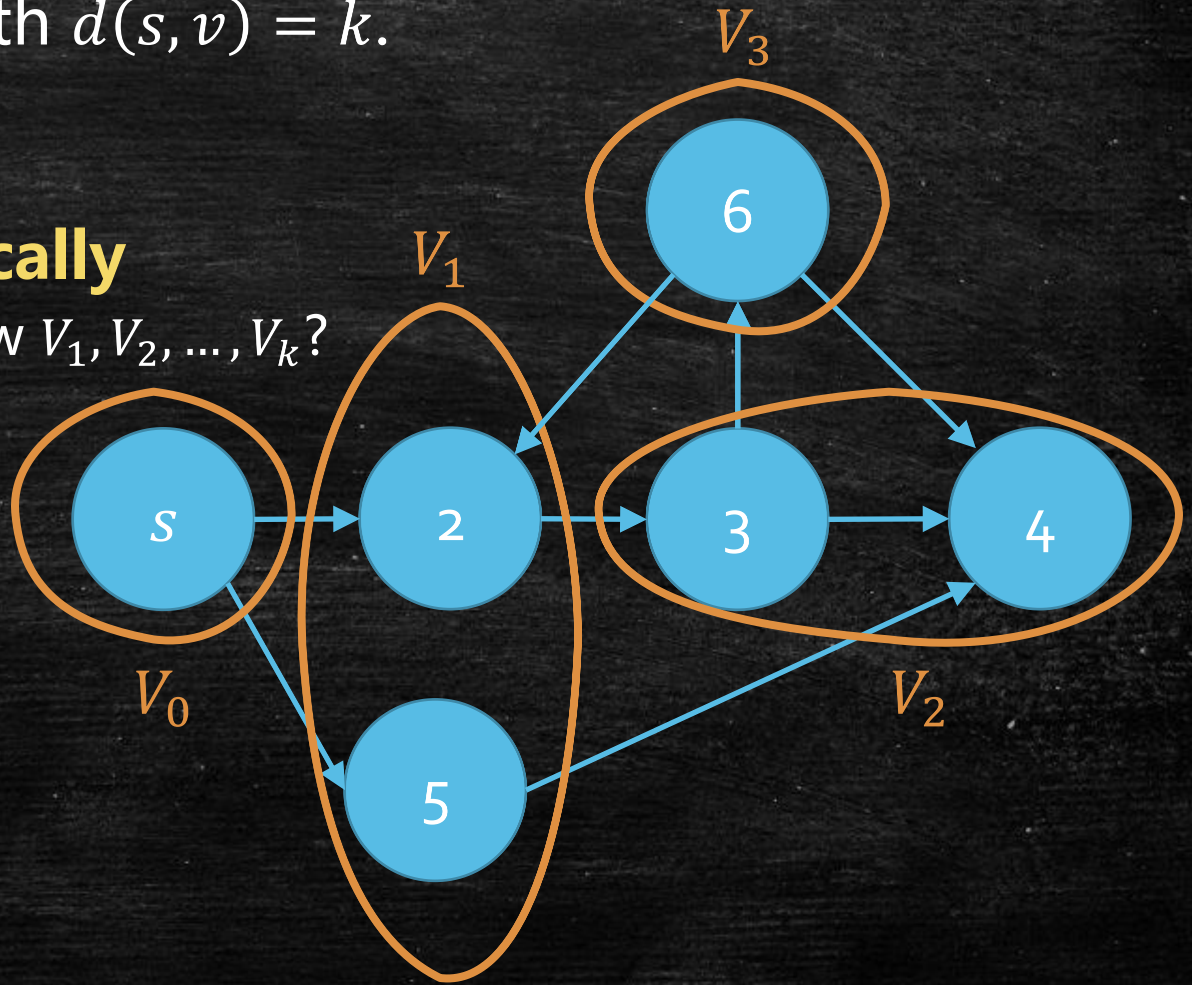
Can DFS help us?

- DFS after 4 explorations.
- Problems:
  - Vertex 5 not visited (only distance 1)
  - Arrive vertex 4 with length 3



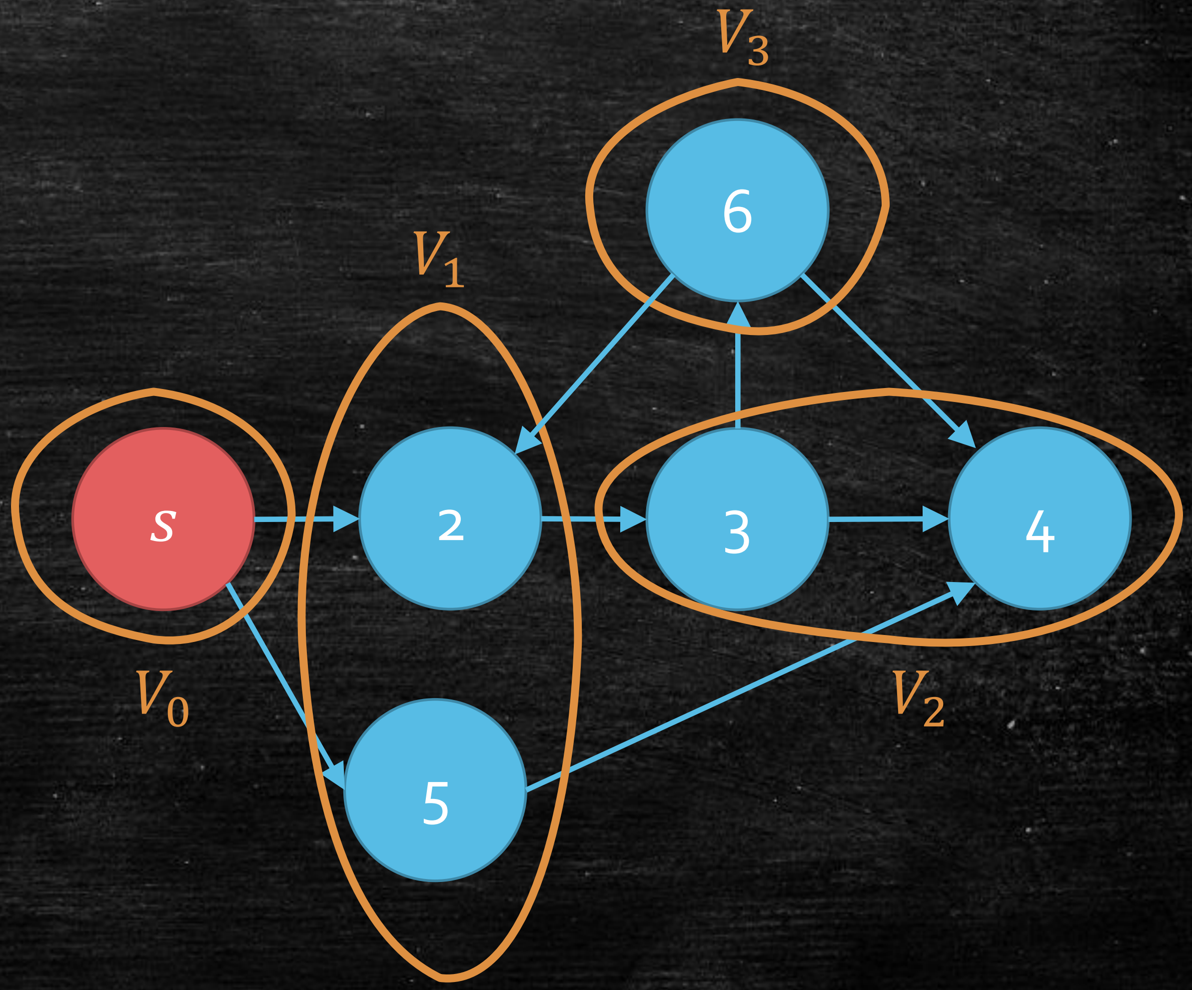
How to Implement the Idea?

- V_k: the set of vertices v with $d(s,v) = k$.
- $V_0 = \{s\}$
- Design algorithm **Analytically**
  - Can we know V_{k+1}, if we know $V_1, V_2, \ldots, V_k$?
  - Yes!
  - $v \in V_{k+1}$ if and only if
    - $u \in V_k$ and (u, v) exists
    - $v \notin V_l,\ \forall l \leq k$.



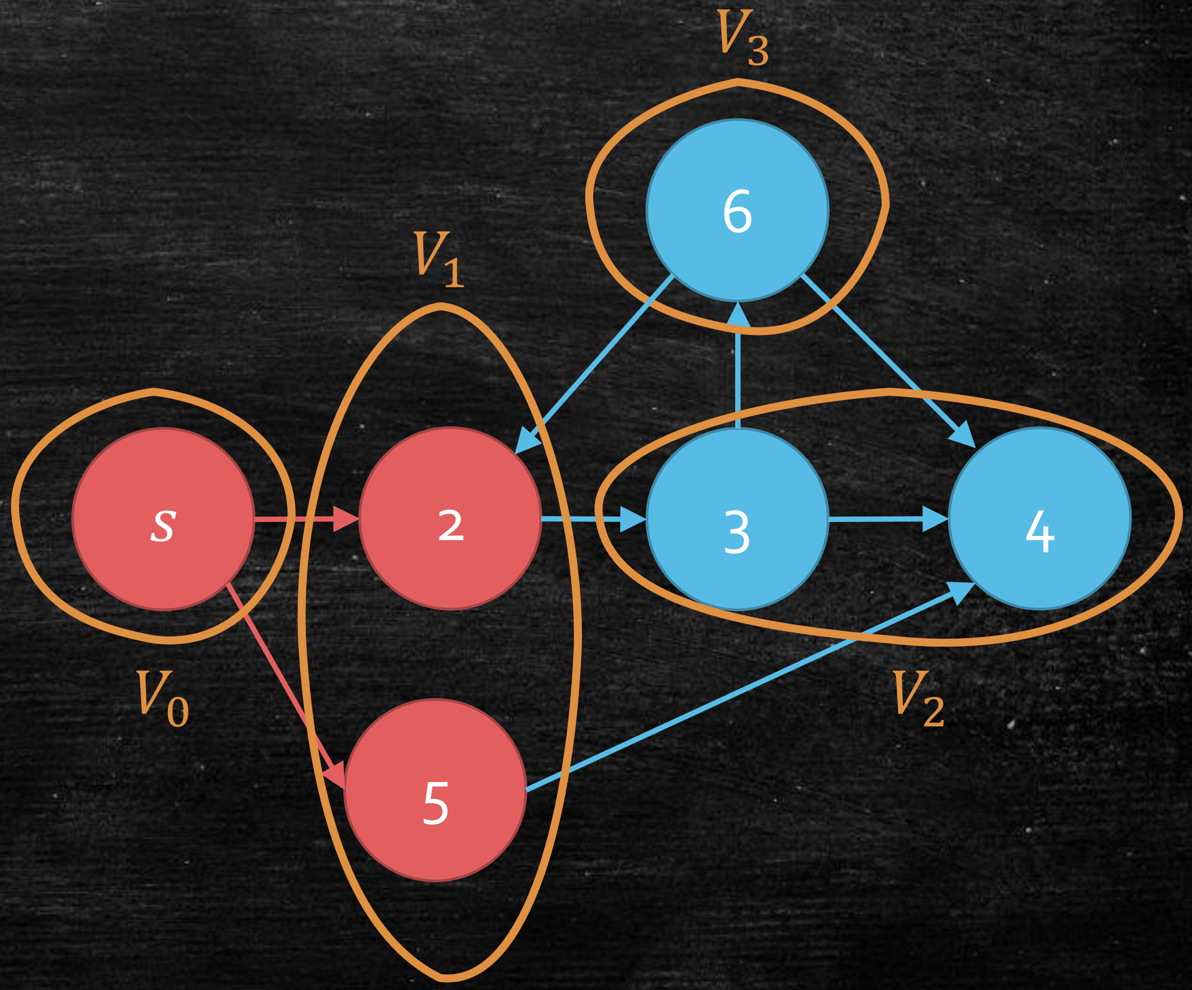
Breadth-First Search (BFS)

- A water frontier.
  - Explore s



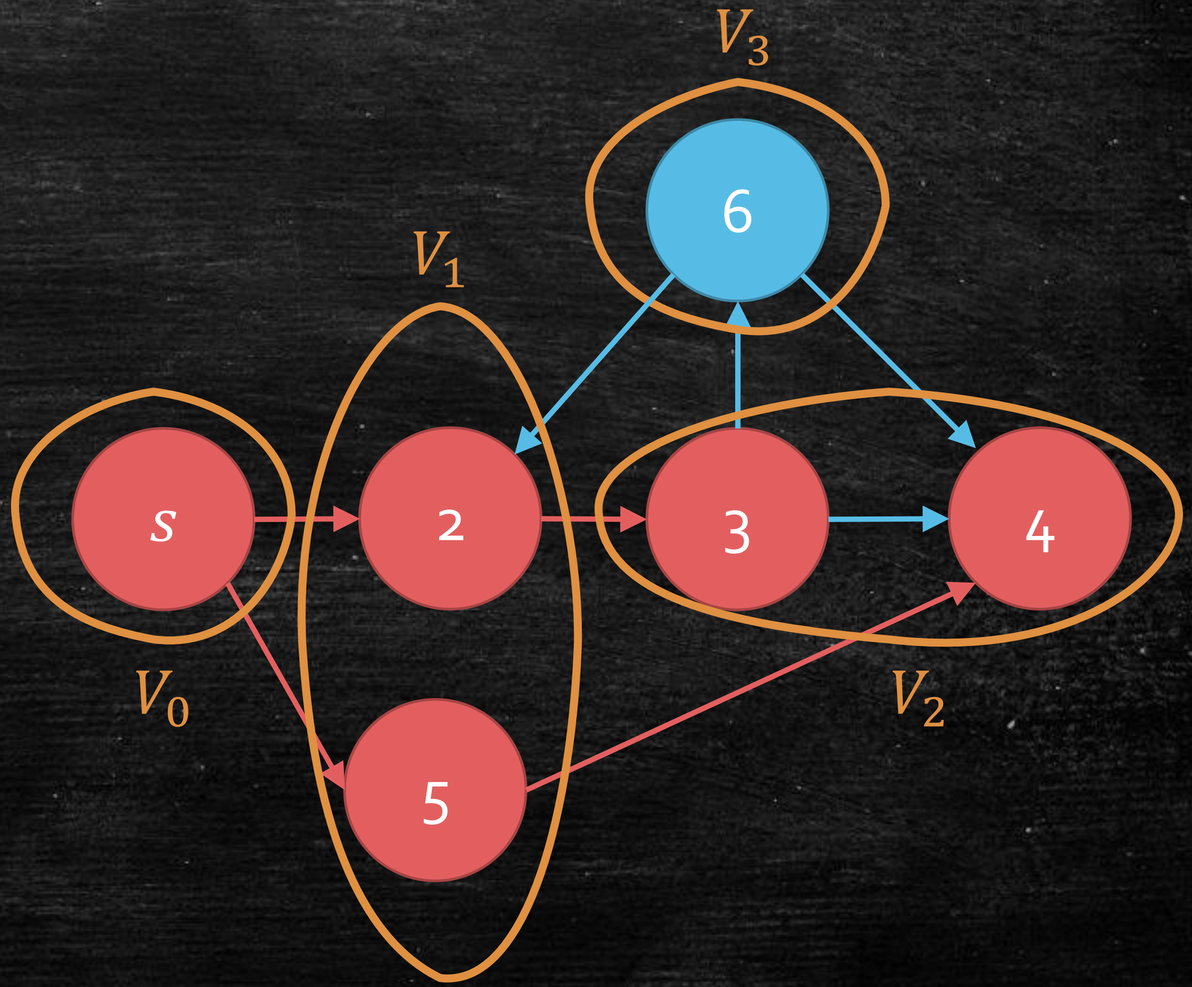
Breadth-First Search (BFS)

- A water frontier.
  - Explore s
  - Explore V_1



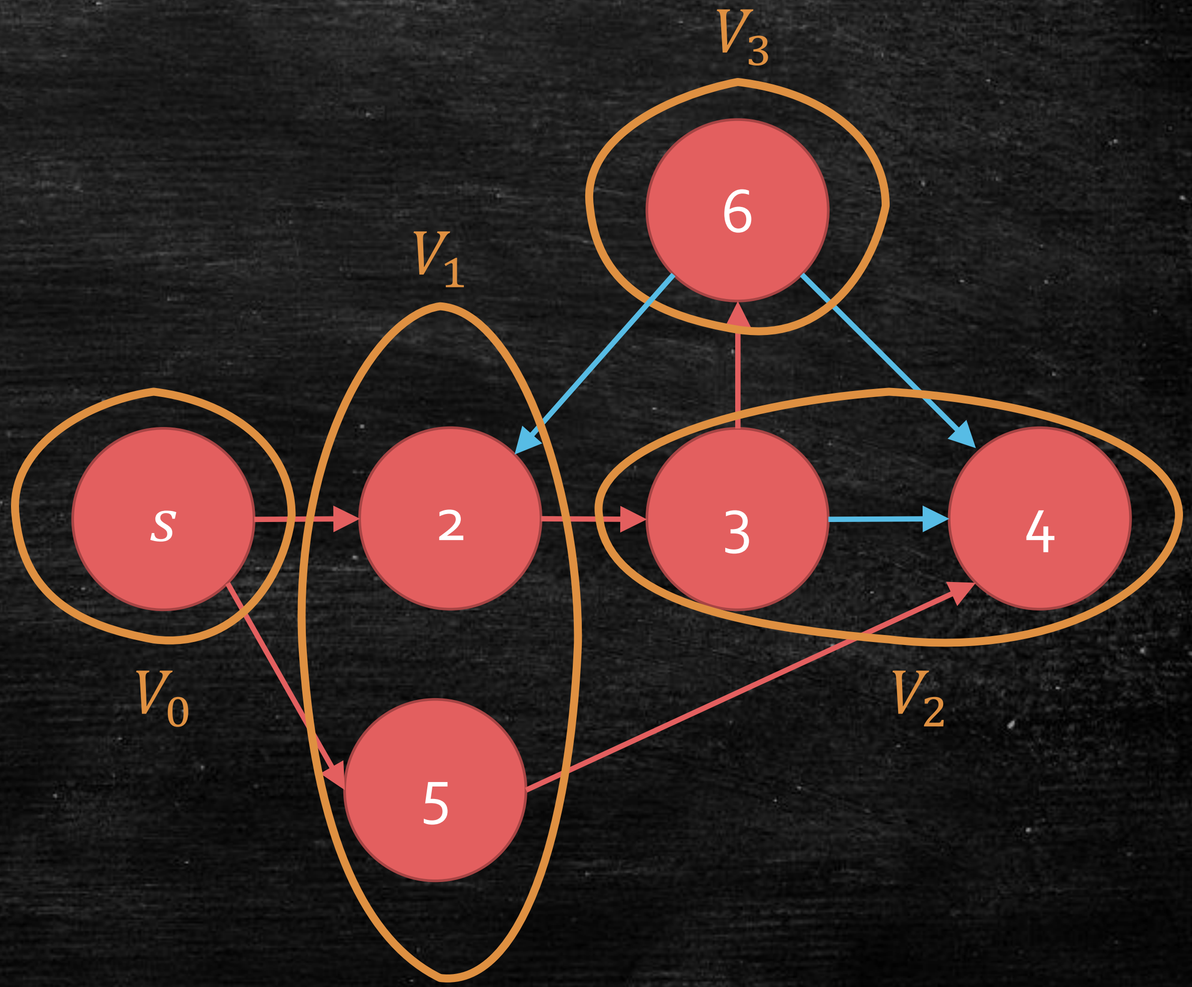
Breadth-First Search (BFS)

- A water frontier.
  - Explore s
  - Explore V_1
  - Explore V_2



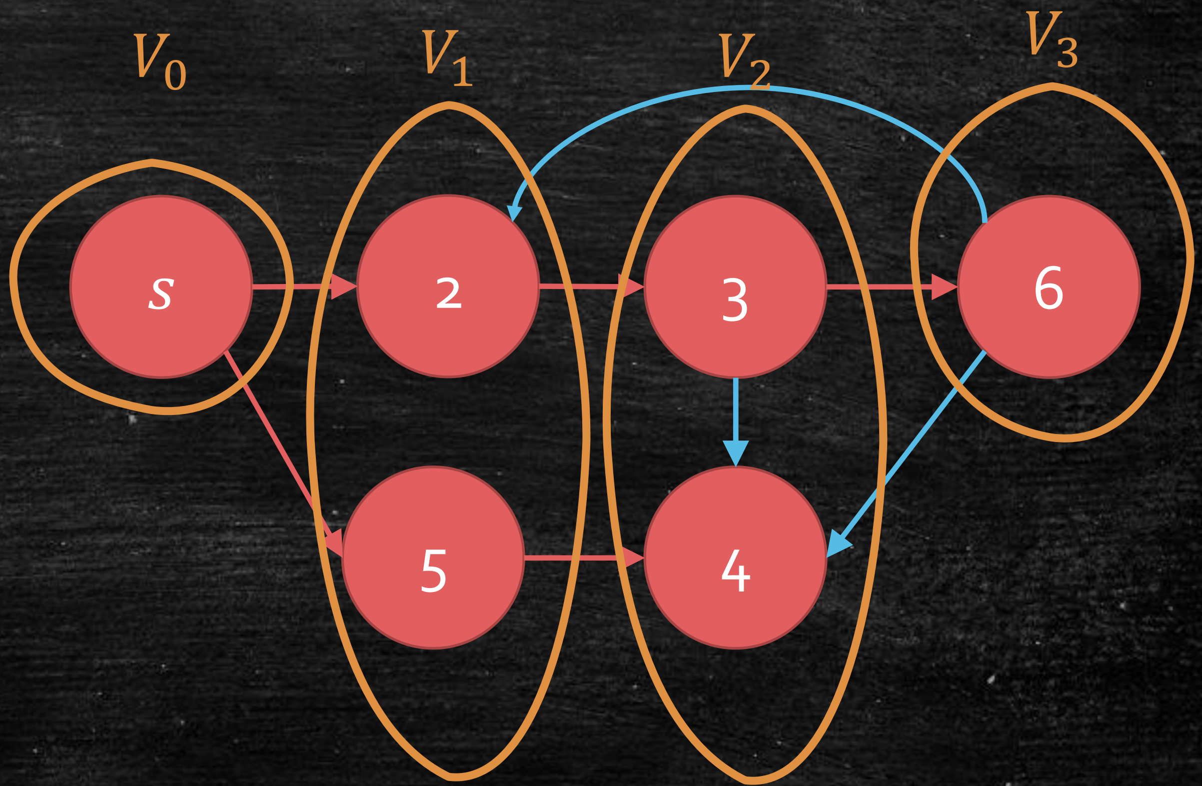
Breadth-First Search (BFS)

- A water frontier.
  - Explore s
  - Explore V_1
  - Explore V_2
  - …



BFS Tree

- A water frontier.
  - Explore s
  - Explore V_1
  - Explore V_2
  - …
- The layer of the vertex
- = The distance from s



How to program?

Breadth First Search

Function bfs(G, s)

  for each v $\in V$ $marked[v] \leftarrow [0]$

  $i \leftarrow 0$ (layer counter)

  $V_0 \leftarrow \{s\}$

  while V_i is not empty

    for each $u \in V_i$

      for each $(u, v) \in E$

        if $marked[v] = false$

          $marked[v] \leftarrow true$

          Add v into V_{i+1}

    $i \leftarrow i + 1$

Running Time? $\boldsymbol{O(|V| + |E|)}$

Charge to marked vertices.

Charge to edges from V_i.

Charge to edges from V_i.

Charge to unmarked vertices.

Output Path?

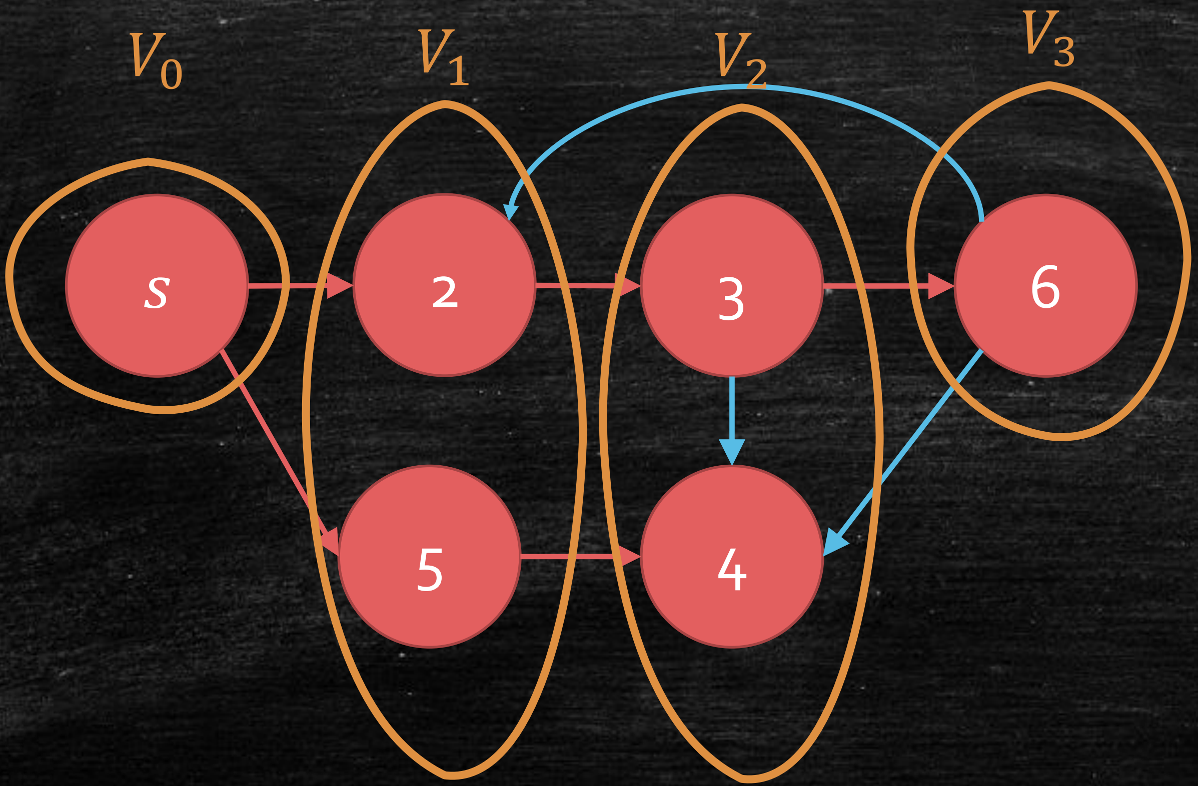
- What if we want to output the shortest path?
- Solution
  - Maintain an array $pre[v]$ means to record the vertex that explores v.

Breadth First Search

Function bfs(G, s)
    for each $\mathrm{v} \in V$ $marked[v] \leftarrow [0]$
    $i \leftarrow 0$ (layer counter)
    $V_0 \leftarrow \{s\}$
    while V_i is not empty
        for each $u \in V_i$
            for each $(u, v) \in E$
                if $marked[v] = false$
                    $marked[v] \leftarrow true$
                    Add v into V_{i+1}
                    $pre[v] \leftarrow u$
        $i \leftarrow i + 1$

Usage of $pre[v]$

- $pre[6] = 3, pre[3] = 2, pre[2] = s.$



DFS vs BFS

	DFS	BFS
Detecting Cycles	YES	NO
Topological Ordering	YES	NO
Finding CCs	YES	YES
Finding SCCs	YES	NO
Shortest Path	NO	YES

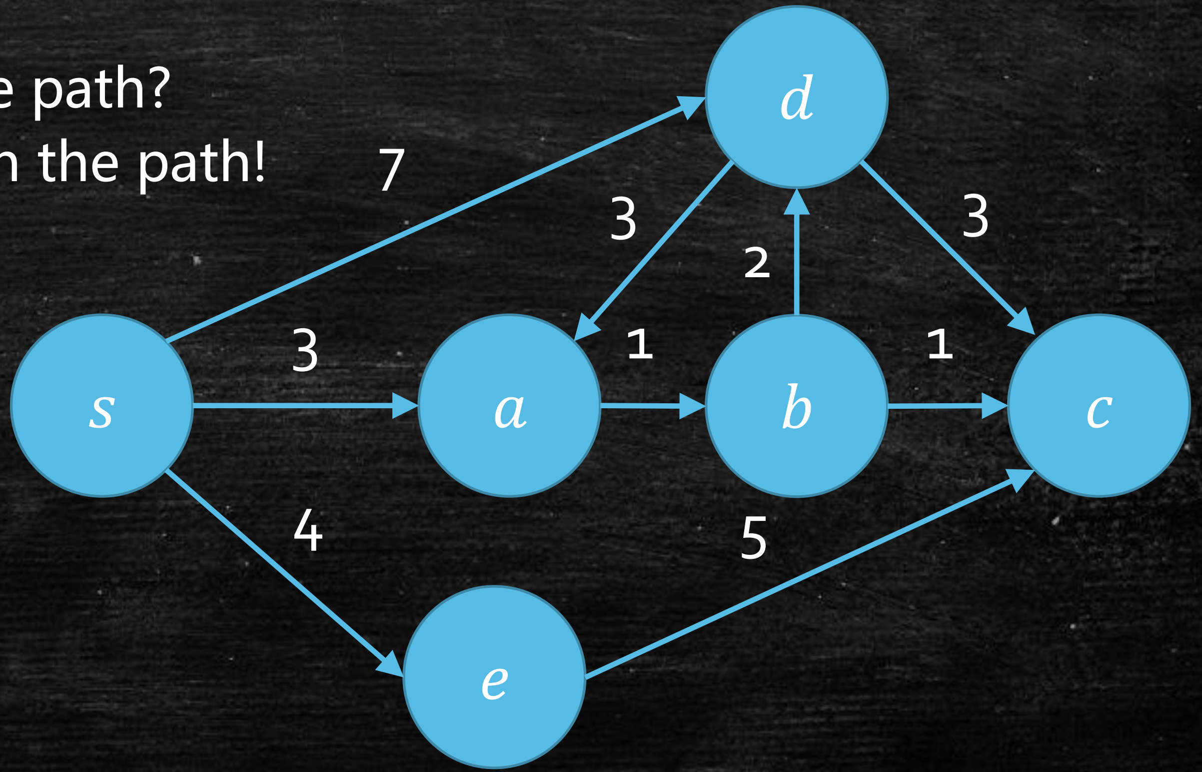
- Hard to distinguish cross edge and back edges in BFS
- Finish time is meaningless in BFS
- *We are discussing the pure DFS and BFS order.

What if edges have length?

Dijkstra Algorithm

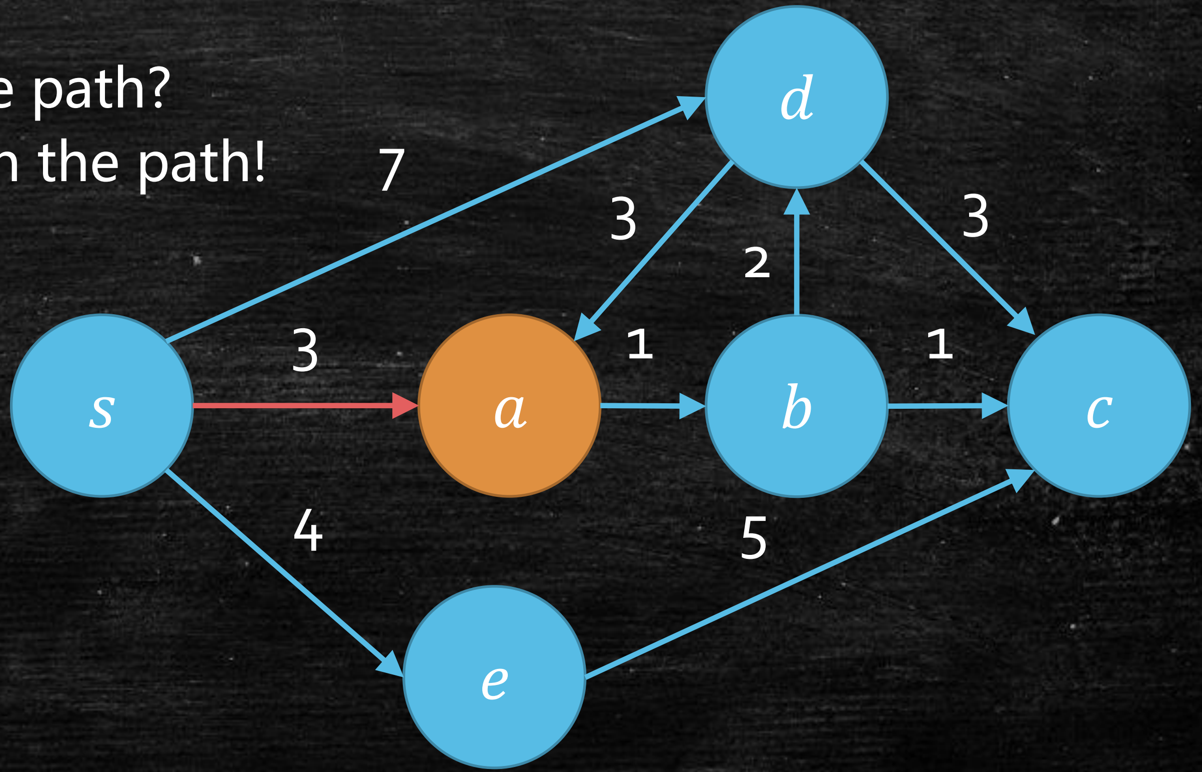
Single-Source Shortest Paths with Weights!

- New Input!
  - Weight/Distance: $w(u, v) \geq 0$ for each edge (u, v)
- **New Length of Path**
  - The number of edges in the path?
  - The **sum** of edges' length in the path!
  - Length $s \rightarrow e \rightarrow c = 9$
  - Length $s \rightarrow a \rightarrow b \rightarrow c = 5$



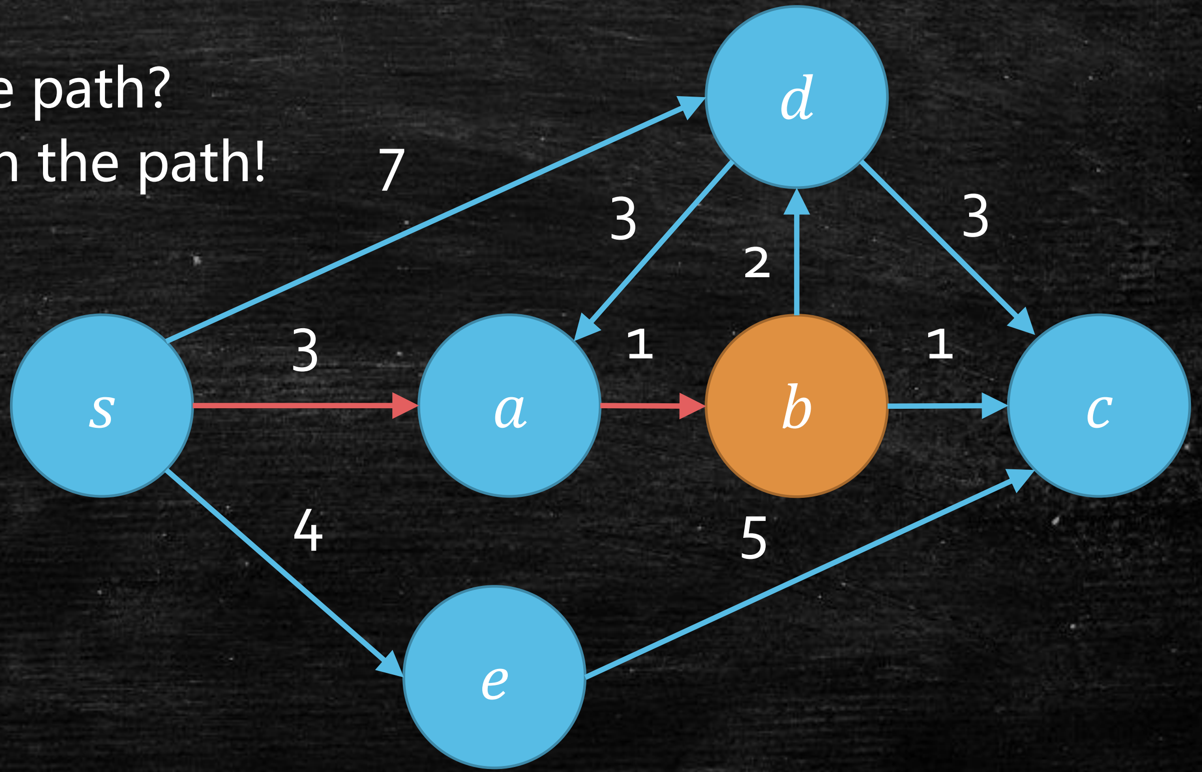
Single-Source Shortest Paths with Weights!

- New Input!
  - Weight/Distance: $w(u, v) \geq 0$ for each edge (u, v)
- **New Length of Path**
  - The number of edges in the path?
  - The **sum** of edges' length in the path!
  - Length $s \rightarrow e \rightarrow c = 9$
  - Length $s \rightarrow a \rightarrow b \rightarrow c = 5$



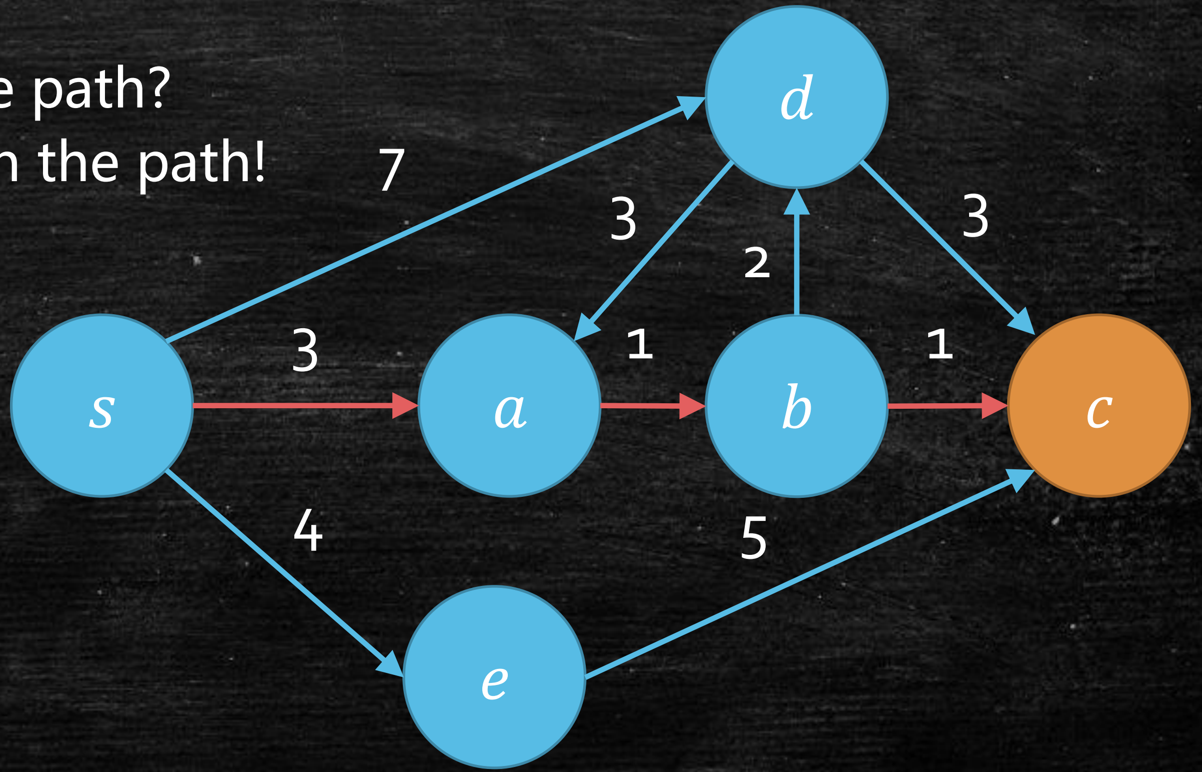
Single-Source Shortest Paths with Weights!

- New Input!
  - Weight/Distance: $w(u, v) \geq 0$ for each edge (u, v)
- **New Length of Path**
  - The number of edges in the path?
  - The **sum** of edges' length in the path!
  - Length $s \to e \to c = 9$
  - Length $s \to a \to b \to c = 5$



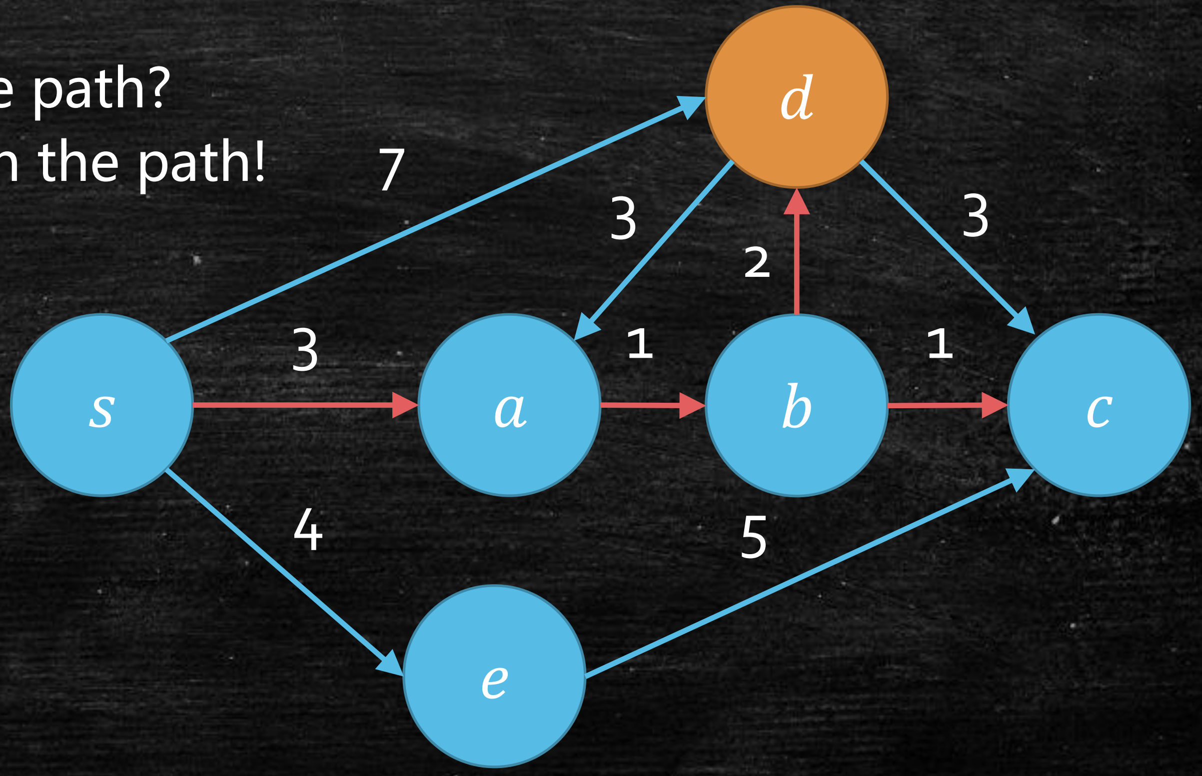
Single-Source Shortest Paths with Weights!

- New Input!
  - Weight/Distance: $w(u,v) \geq 0$ for each edge (u,v)
- **New Length of Path**
  - The number of edges in the path?
  - The **sum** of edges' length in the path!
  - Length $s \to e \to c = 9$
  - Length $s \to a \to b \to c = 5$



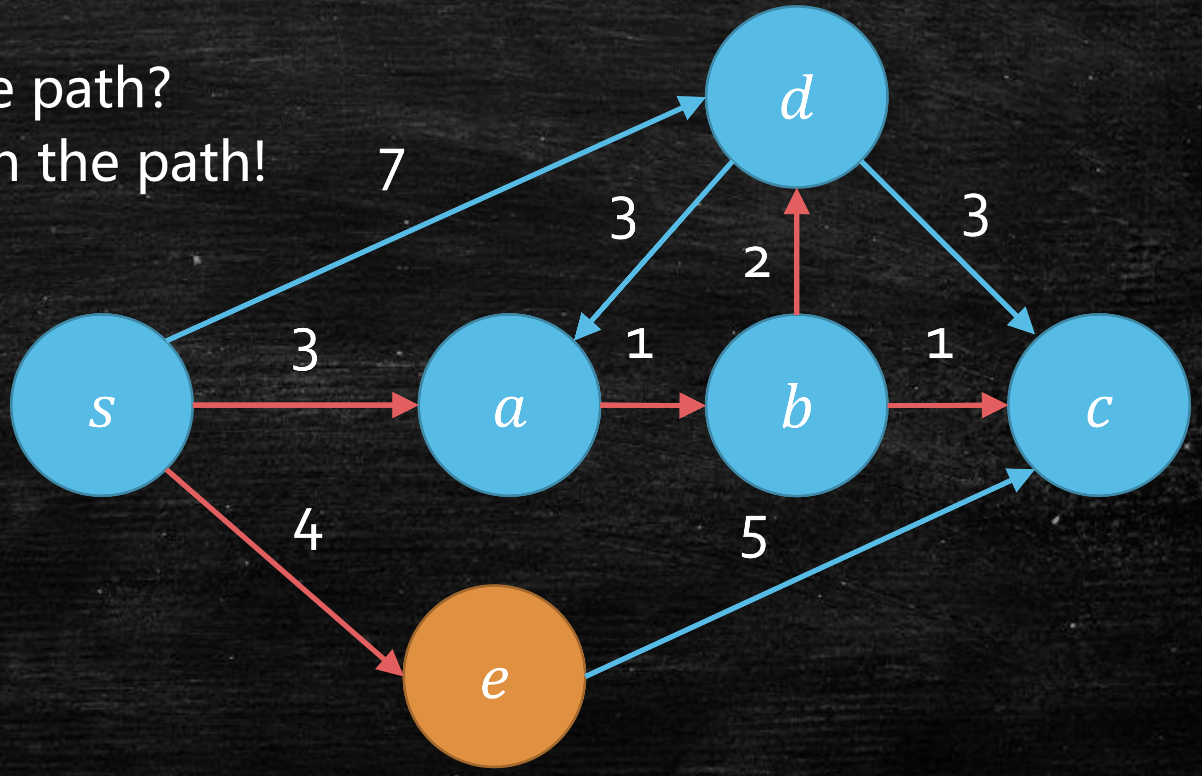
Single-Source Shortest Paths with Weights!

- New Input!
  - Weight/Distance: $w(u,v) \geq 0$ for each edge (u,v)
- **New Length of Path**
  - The number of edges in the path?
  - The **sum** of edges' length in the path!
  - Length $s \to e \to c = 9$
  - Length $s \to a \to b \to c = 5$



Single-Source Shortest Paths with Weights!

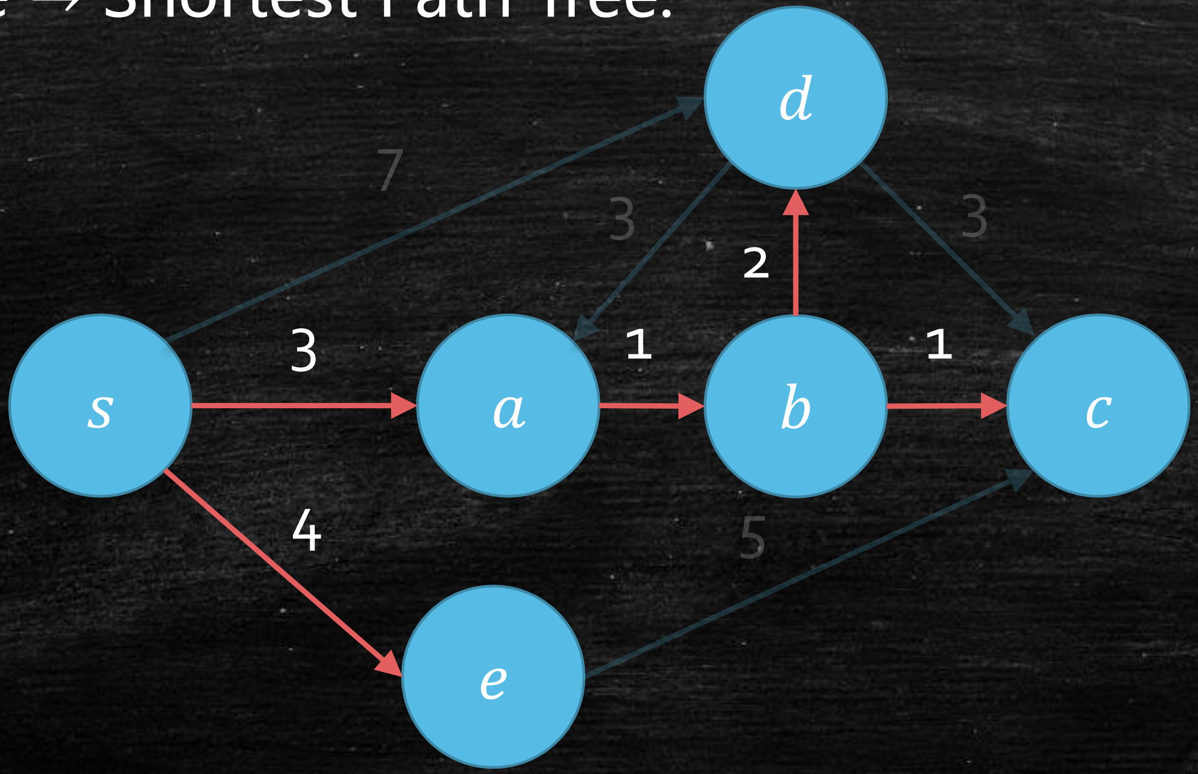
- New Input!
  - Weight/Distance: $w(u, v) \geq 0$ for each edge (u, v)
- **New Length of Path**
  - The number of edges in the path?
  - The **sum** of edges' length in the path!
  - Length $s \rightarrow e \rightarrow c = 9$
  - Length $s \rightarrow a \rightarrow b \rightarrow c = 5$



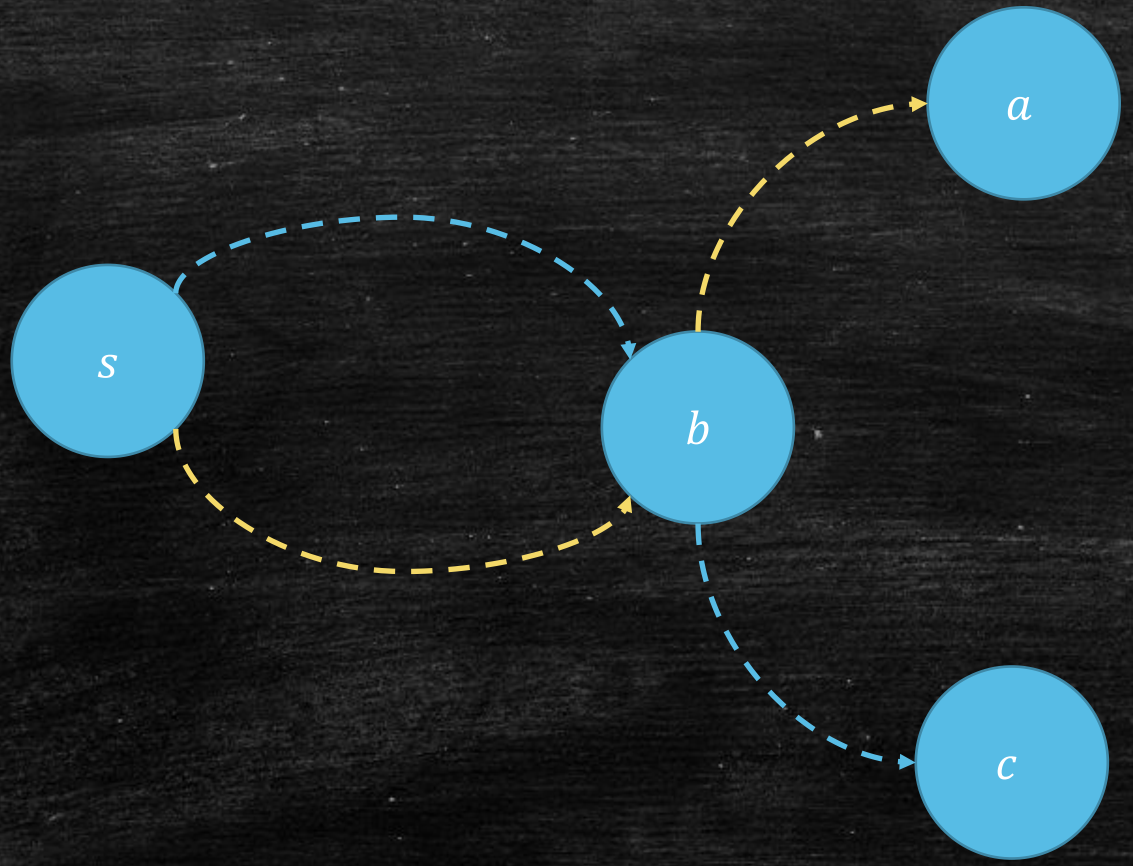
Can we still use BFS?

Rough Observation

- The union of shortest paths forms a tree
  - **Shortest Path Tree.**
- BFS tree → Shortest Path Tree.



What if we have more than one indegree?



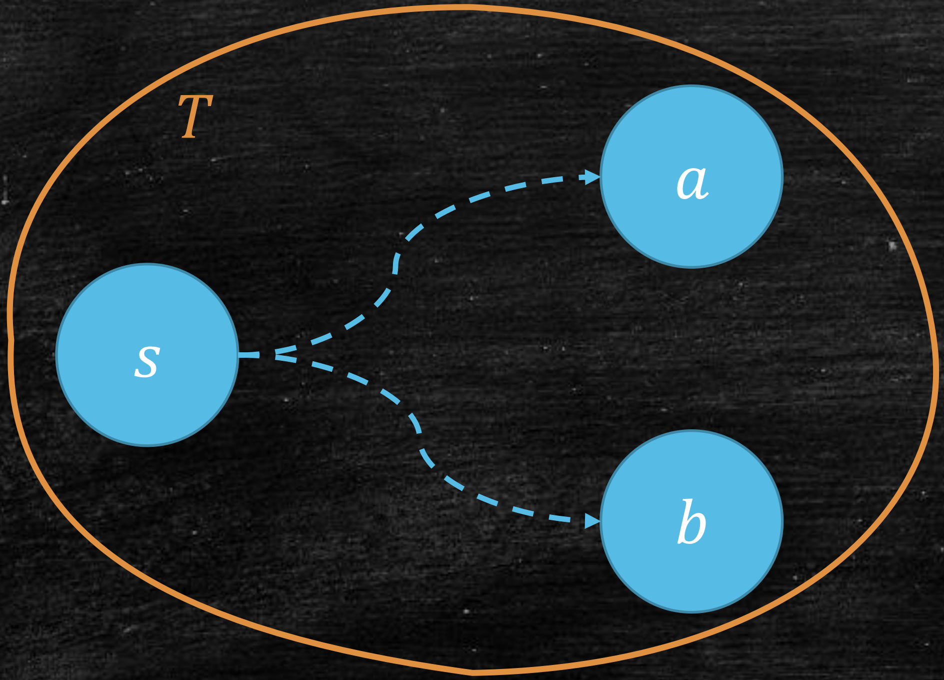
Prove it forms a shortest path tree and find it!

Prove it by a construction!

- Question:
  - Does it exist a **Shortest Path Tree**?
  - Prove it by an inductive construction!
- **Shortest Path Tree (SPT)**
  - $v \in T$, $s \to v$ path in T is the shortest path in G.
- Start point
  - $\{s\}$ is a SPT.
- Next
  - Can we always explore current SPT to a larger one until all vertices are included?

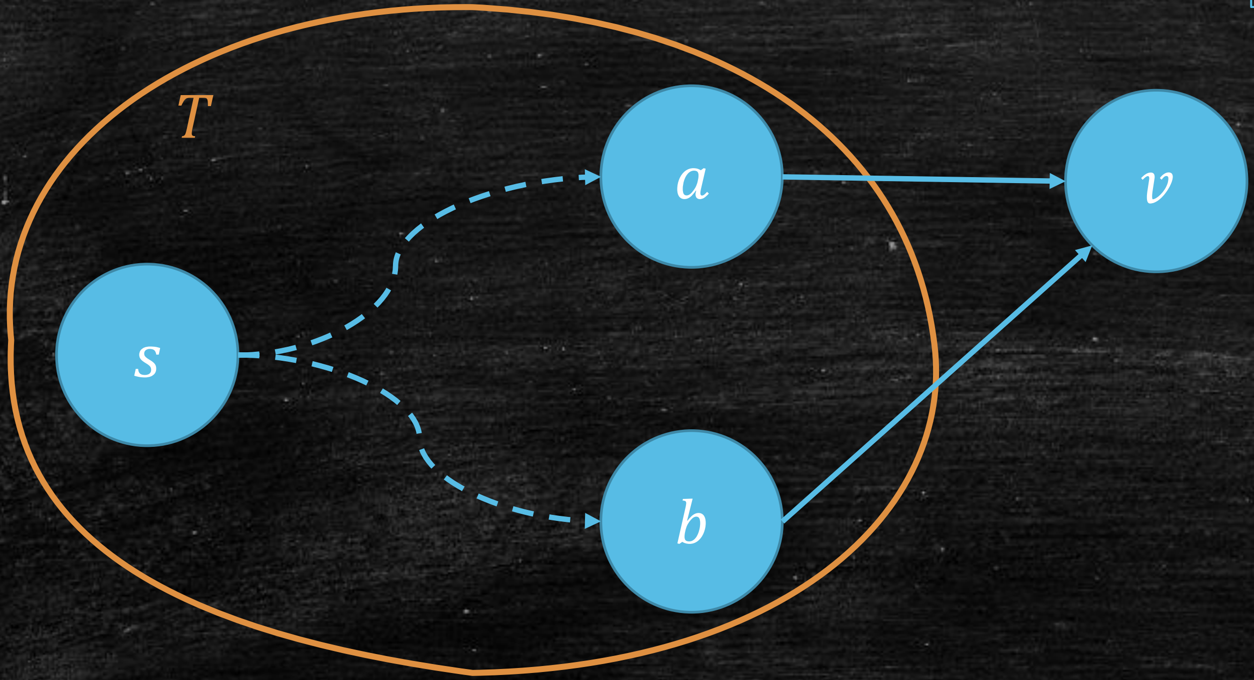
Key Task

- **Given:** a small SPT (not contains all the vertices)
- **Want:** a larger SPT



Key Task: Vertex Exploring

- Can we explore v into T?

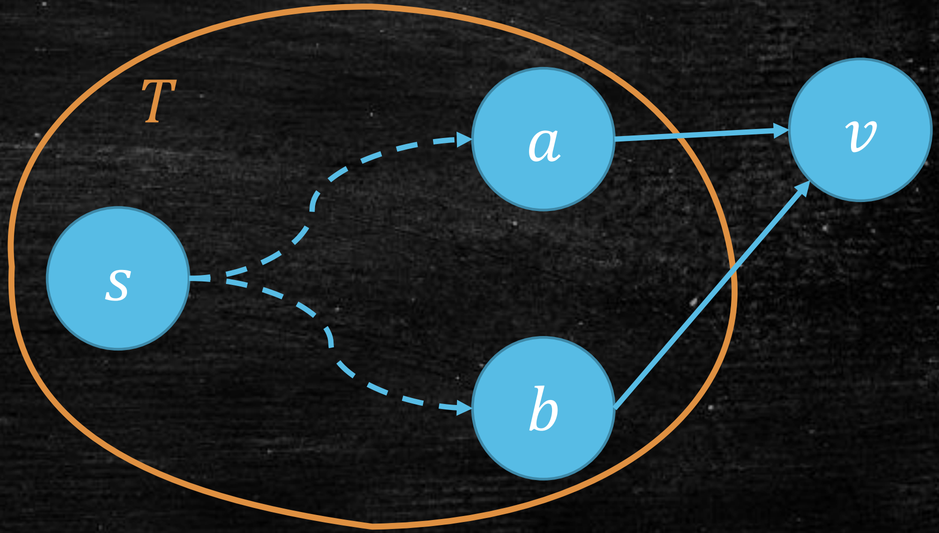


- **Given:** a small SPT (not contains all the vertices)
- **Want:** a larger SPT

Key Task: Vertex Exploring

- Property of the current $\boldsymbol{T}$
  - True distance: $dist(u)$
  - Tree distance: $dist_T(u)$ only allows to go through T.
  - Basic property: $dist_T(u)=dist(u)$ if $u \in T$
- We want to join v into T!
- Where should we put v?

- **Given:** a small SPT (not contains all the vertices)
- **Want:** a larger SPT
- Can we explore v into T?

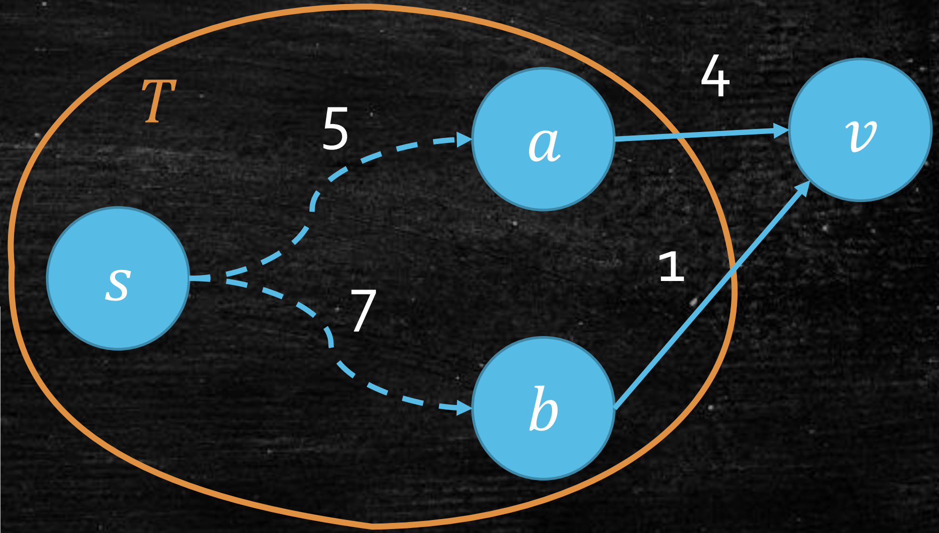


Key Task

- Property of the current T
  - True distance: $dist(u)$
  - Tree distance: $dist_T(u)$ only allows to go through T.
  - Basic property: $dist_T(u)=dist(u)$ if $u \in T$
- We want to join v into T!
- Where should we put v?
- $dist_T(v) = \min_{u \in T}\{dist_T(u) + d(u,v)\}$

- $s \to a \to v = 9$
- $s \to b \to v = 8$
- $dist_T(v) = 8$

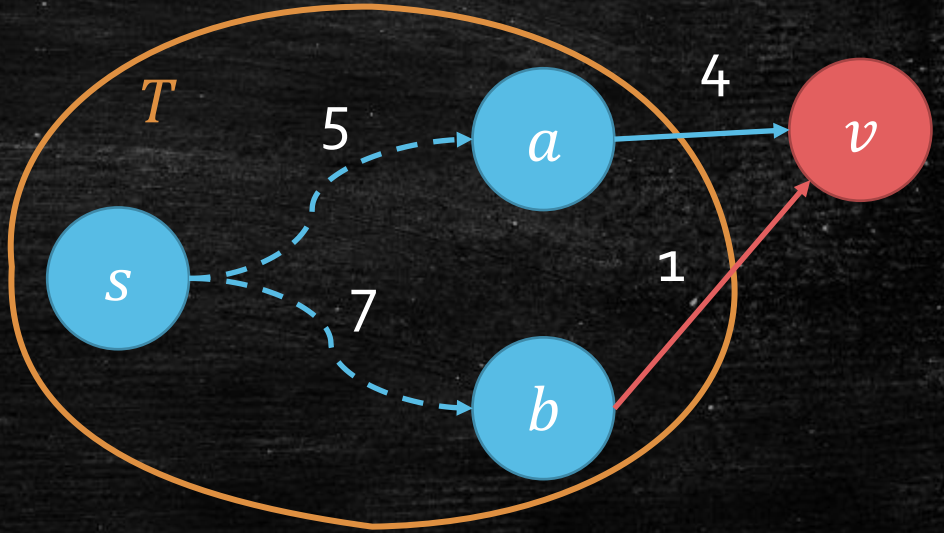
- **Given:** a small SPT (not contains all the vertices)
- **Want:** a larger SPT
- Can we explore v into T?



Key Task

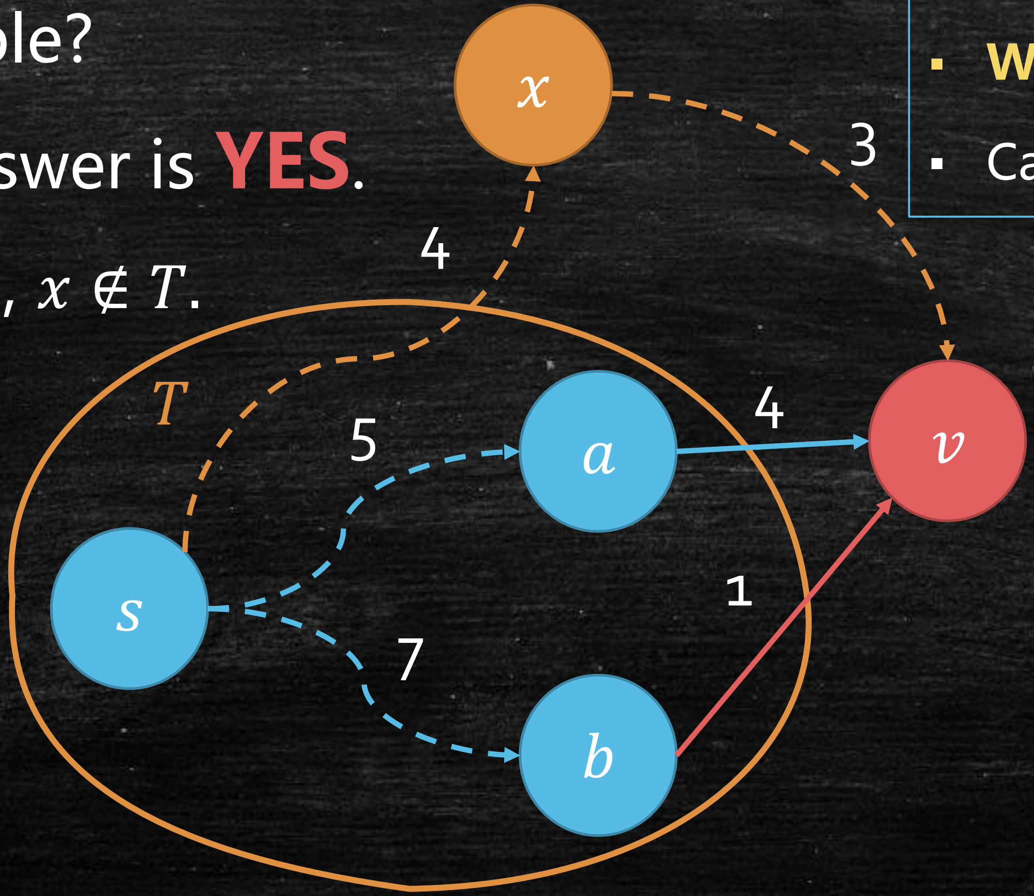
- Try to explore v into T
- Naturally, we should connect it to $\underset{u \in T}{\operatorname{argmin}}\, dist_T(u) + d(u, v)$
- Is that still an SPT?
  - Need to keep: Shortest T-path is the shortest in G.
  - All the other vertices except v is ok
  - Tree distance of v: $dist_T(v)$
  - **Key challenge**: Does $dist_T(v) = dist(v)$?

- **Given:** a small SPT (not contains all the vertices)
- **Want:** a larger SPT
- Can we explore v into T?



Prove $dist_T(v) \le dist(v)$

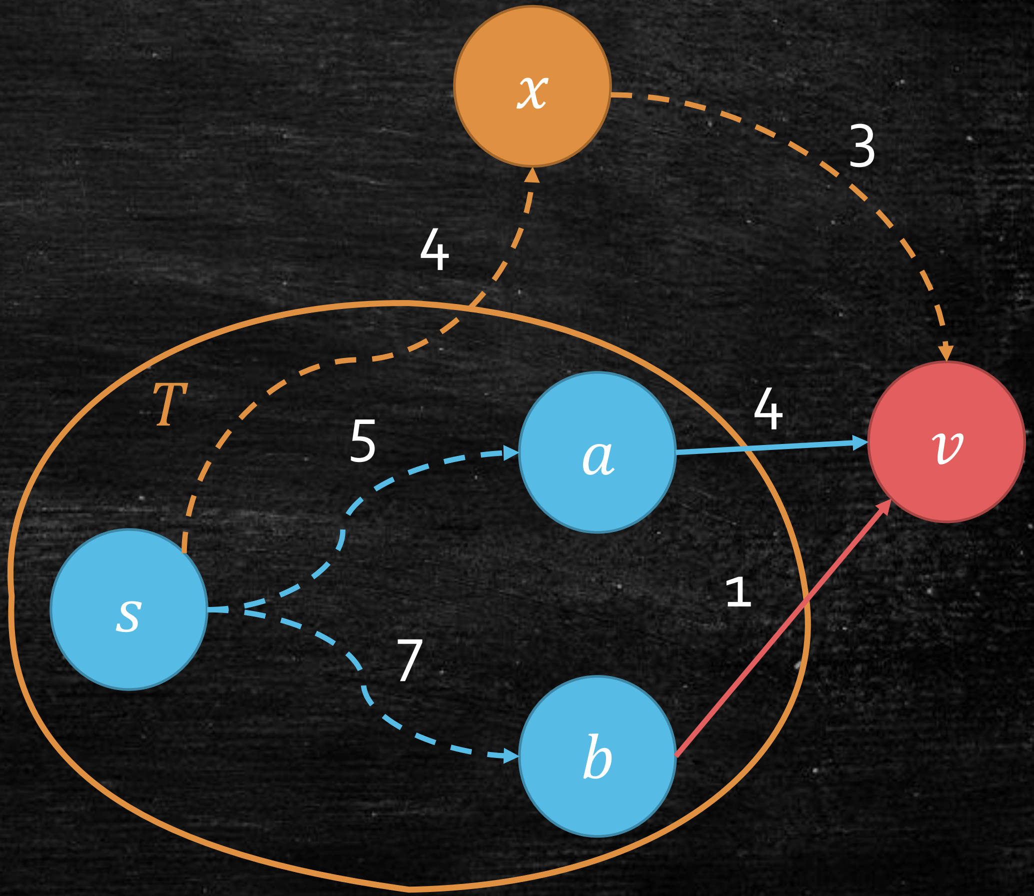
- Assume $dist_T(v) > dist(v)$
- Is that possible?
- Sorry, the answer is **YES**.
- $s \to x \to v = 7$, $x \notin T$.



- **Given:** a small SPT (not contains all the vertices)
- **Want:** a larger SPT
- Can we explore v into T?

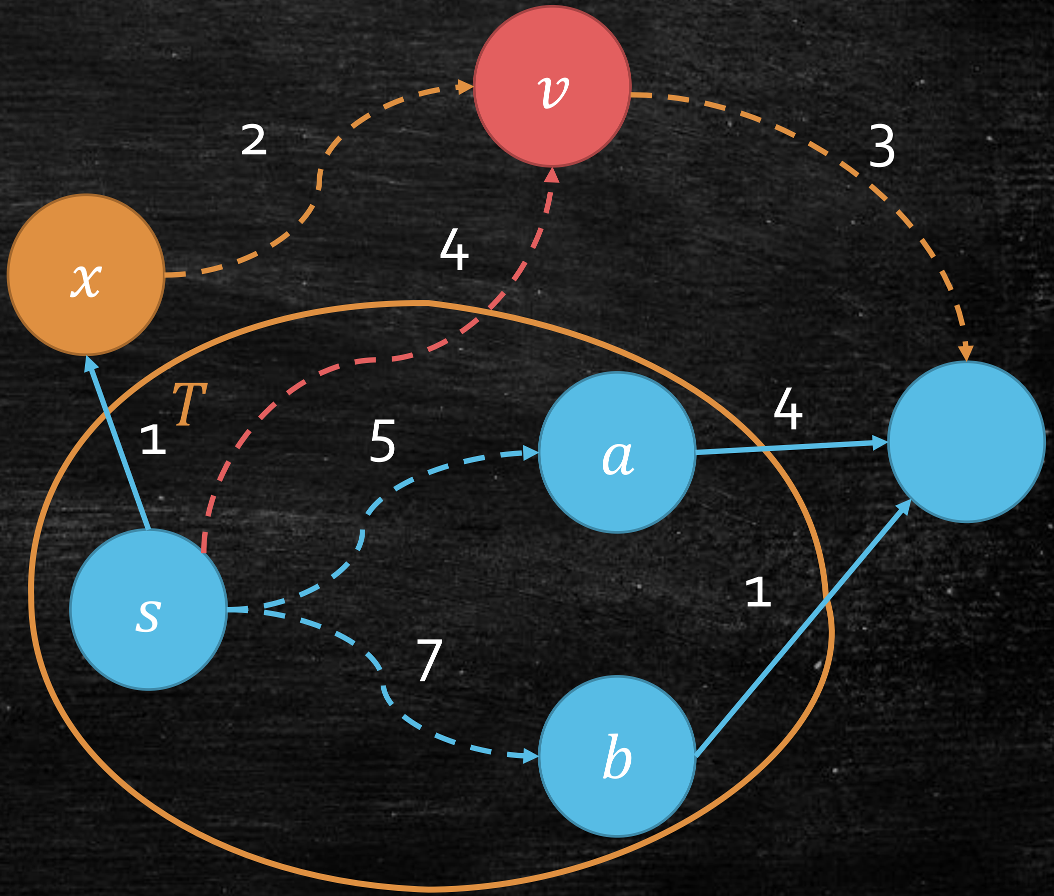
How to handle it?

- Recall BFS idea
- Each time, we explore a closest vertex.
- What happens now?
- $\boldsymbol{x}$ is a closer vertex than $\boldsymbol{v}$.
- Why not explore $\boldsymbol{x}$?
- Formalize: Choose the vertex v with **smallest** $dist_T(v)$!



Prove $dist_T(v) \le dist(v)$ **AGAIN**!

- Try to explore v with **smallest** $dist_T(v)$ into T
- We should connect it to $\underset{u \in T}{\operatorname{argmin}}\, dist_T(u) + d(u, v)$
- Assume $dist_T(v) > dist(v)$
- $x \notin T,\ s \to x \to v < dist_T(v)$
- $dist_T(x)$ is a part of $s \to x \to v$
- $dist_T(x) < dist_T(v)$
- **Contradiction!**



Yah! Success

- **Given:** a small SPT (not contains all the vertices)
- **Want:** a larger SPT
- Can we explore v into T?
- Yes!
- We can find $v = \underset{v \in T}{\operatorname{argmin}}\, dist_T(v)$ to explore!
- Finally, we can get SPT that contains all vertices!
  - Assume s can arrive all vertices

So, we also have a construction for SPT.

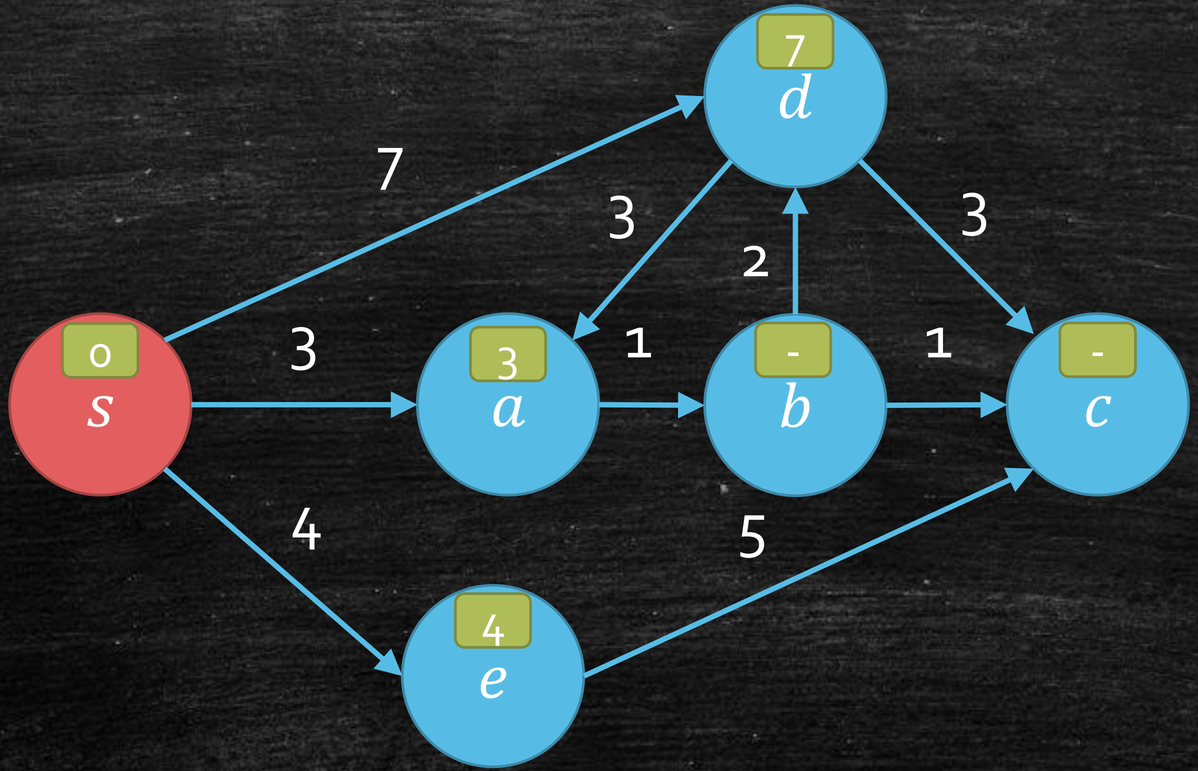
We also have an algorithm!

Dijkstra Algorithm

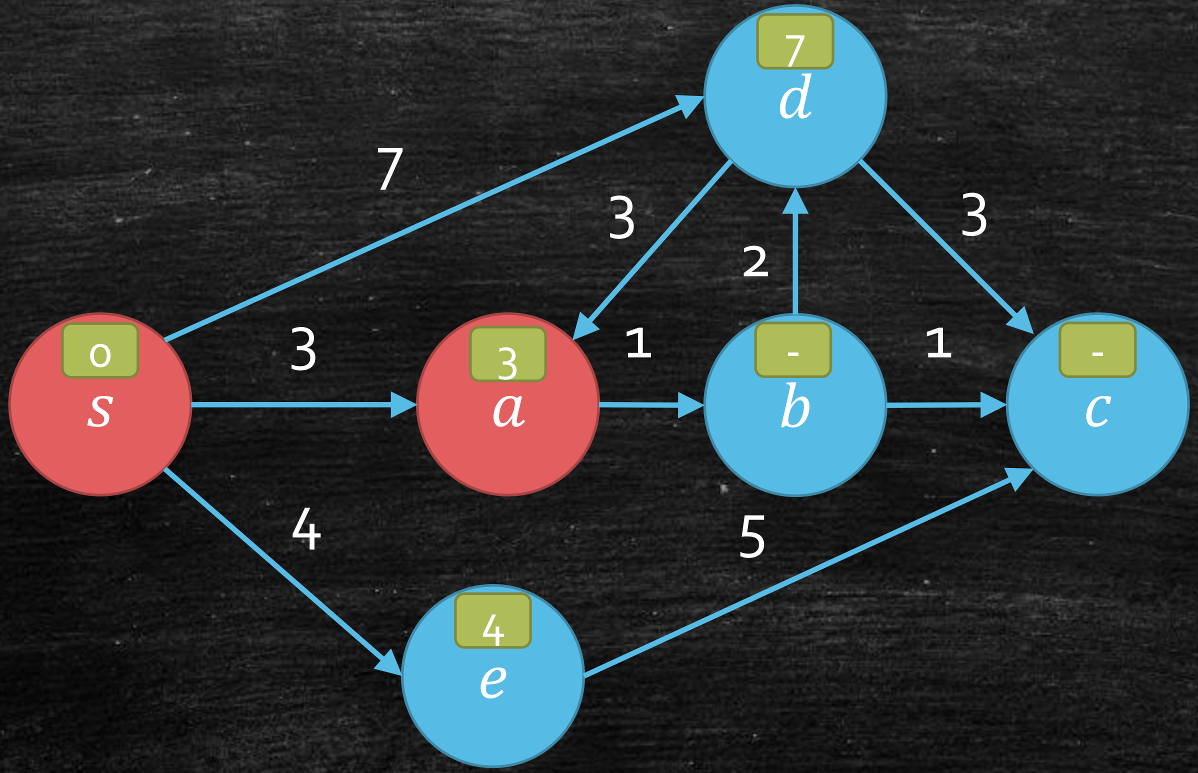
Dijkstra$(\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E}), \boldsymbol{s})$

1. **Initialize**
   - $T = \{s\}$,
   - $tdist[s] = 0,\ \ tdist[v] \leftarrow \infty$ for all v other than s.
   - $tdist[v] \leftarrow w(s, v)$ for all $(s, v) \in E$.
2. **Explore**
   - Find $v \notin T$ with smallest $tdist[v]$.
   - $T \leftarrow T + \{v\}$
3. **Update $\boldsymbol{tdist[u]}$**
   - $tdist[u] = \min\{tdist[u], tdist[v] + w(v, u)\}$ for all $(v, u) \in E$

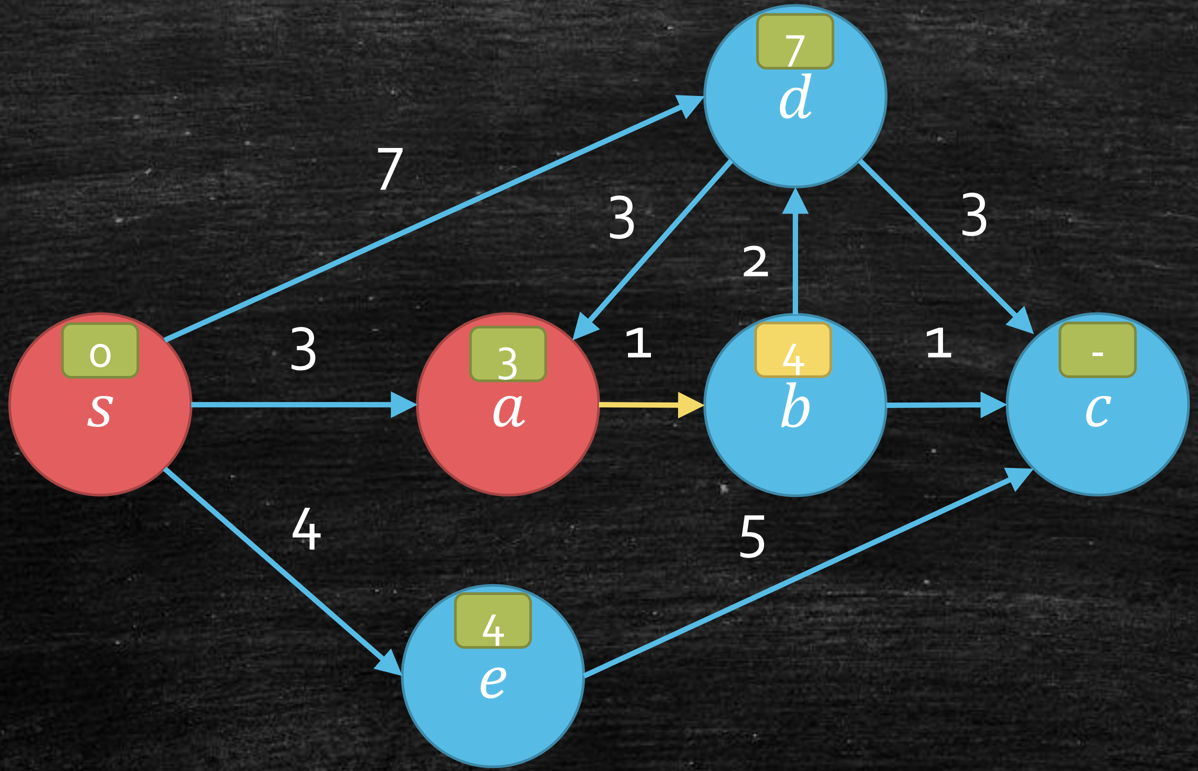
Sample Run



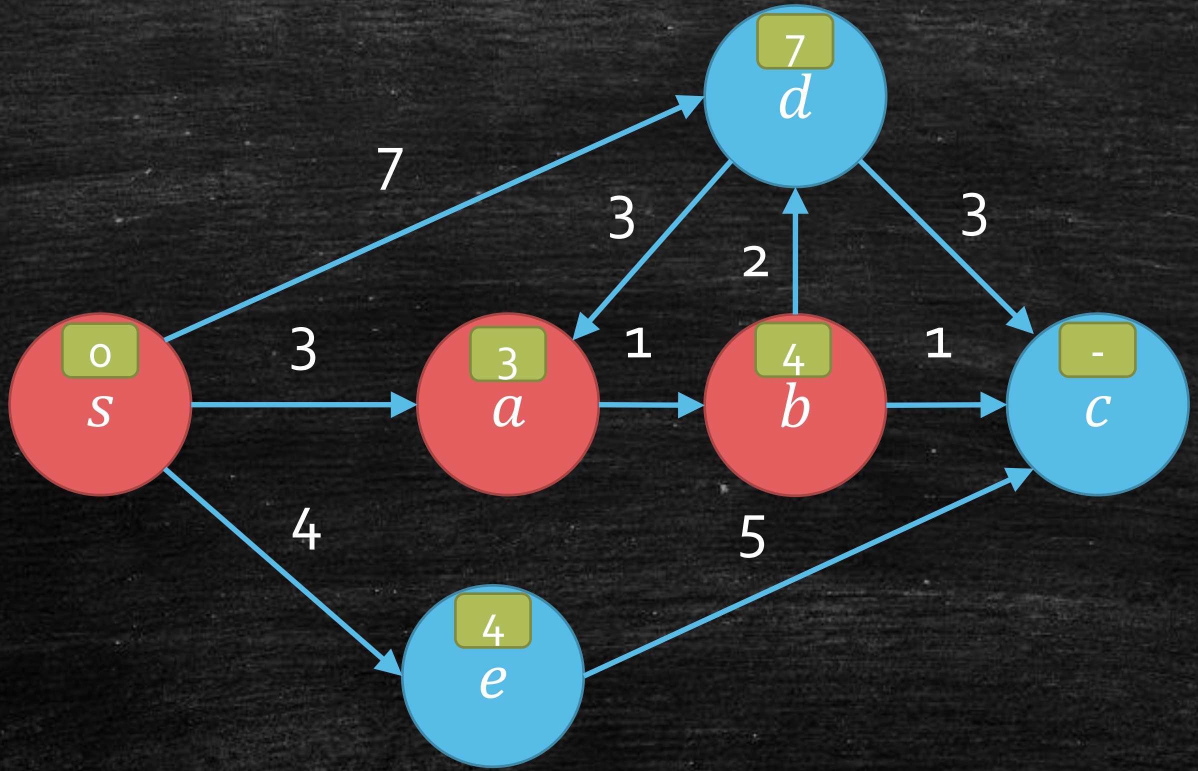
Sample Run



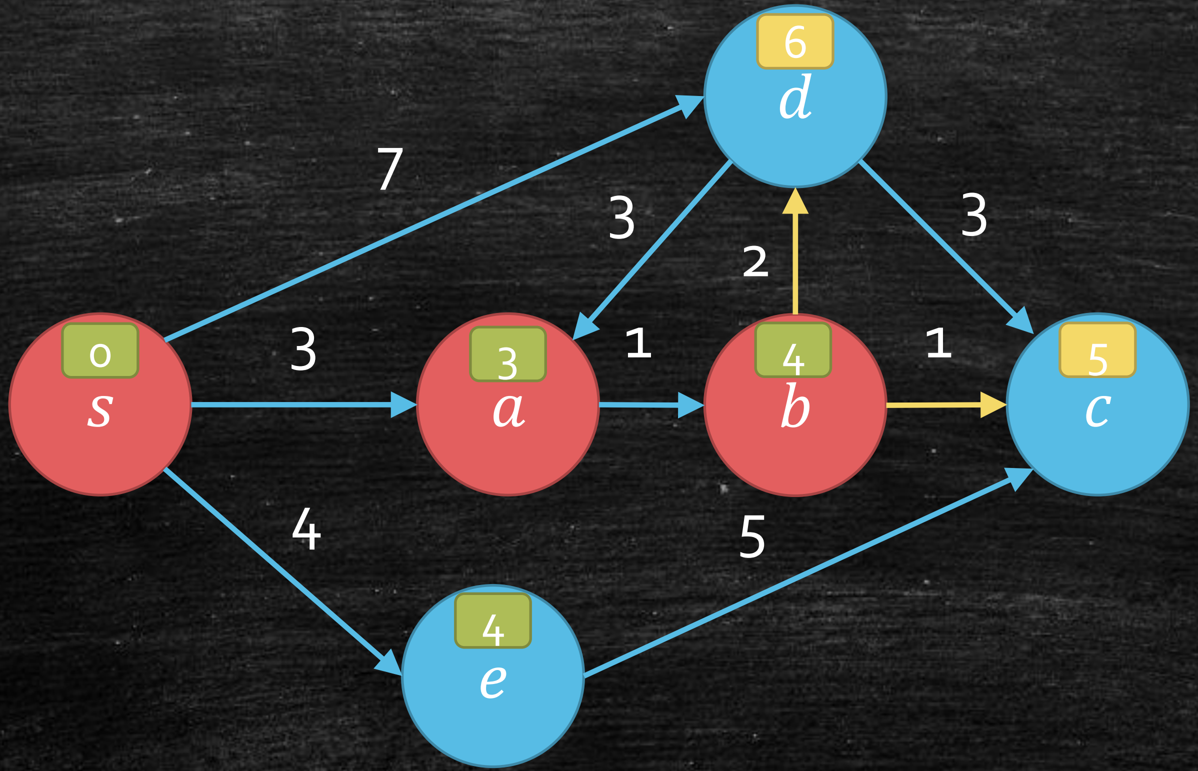
Sample Run



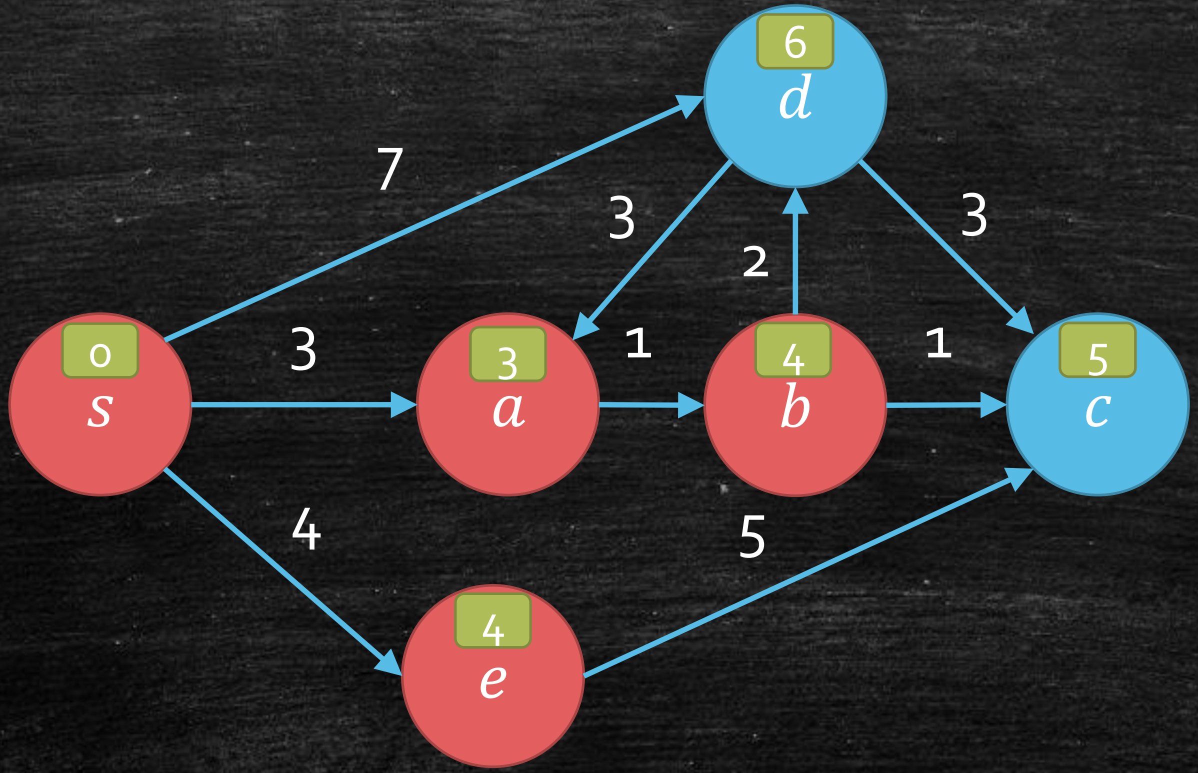
Sample Run



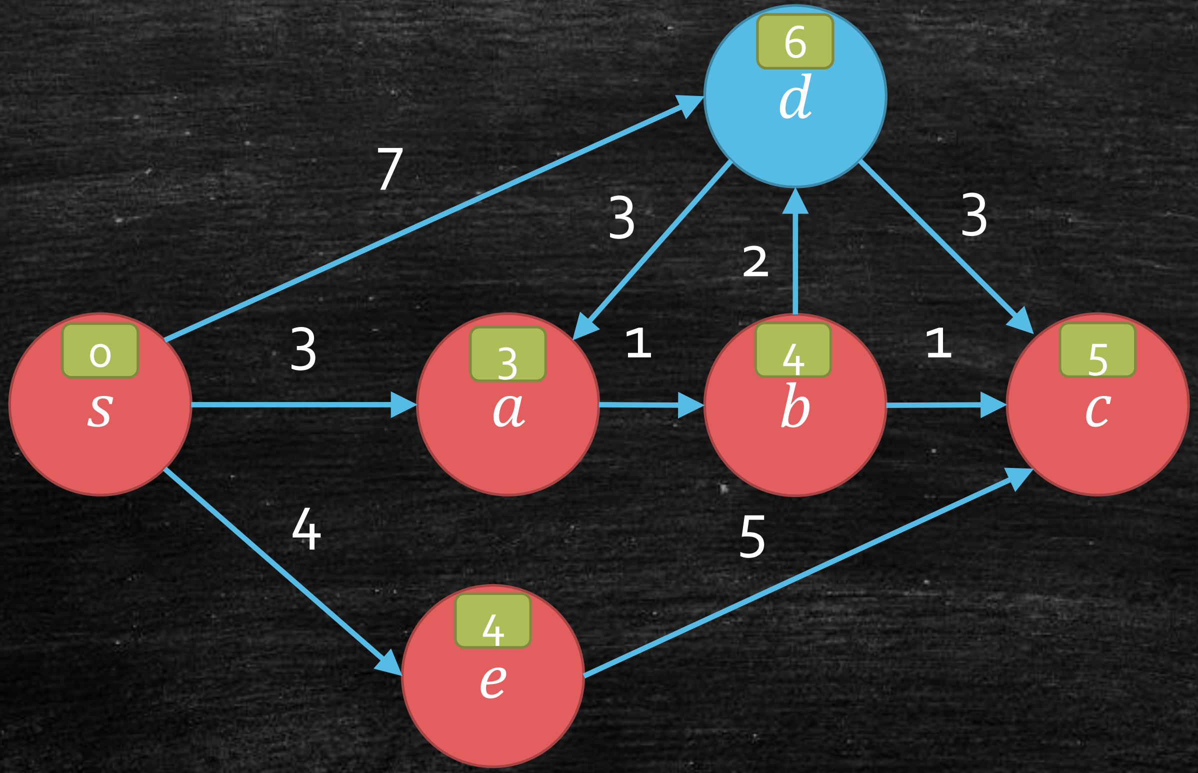
Sample Run



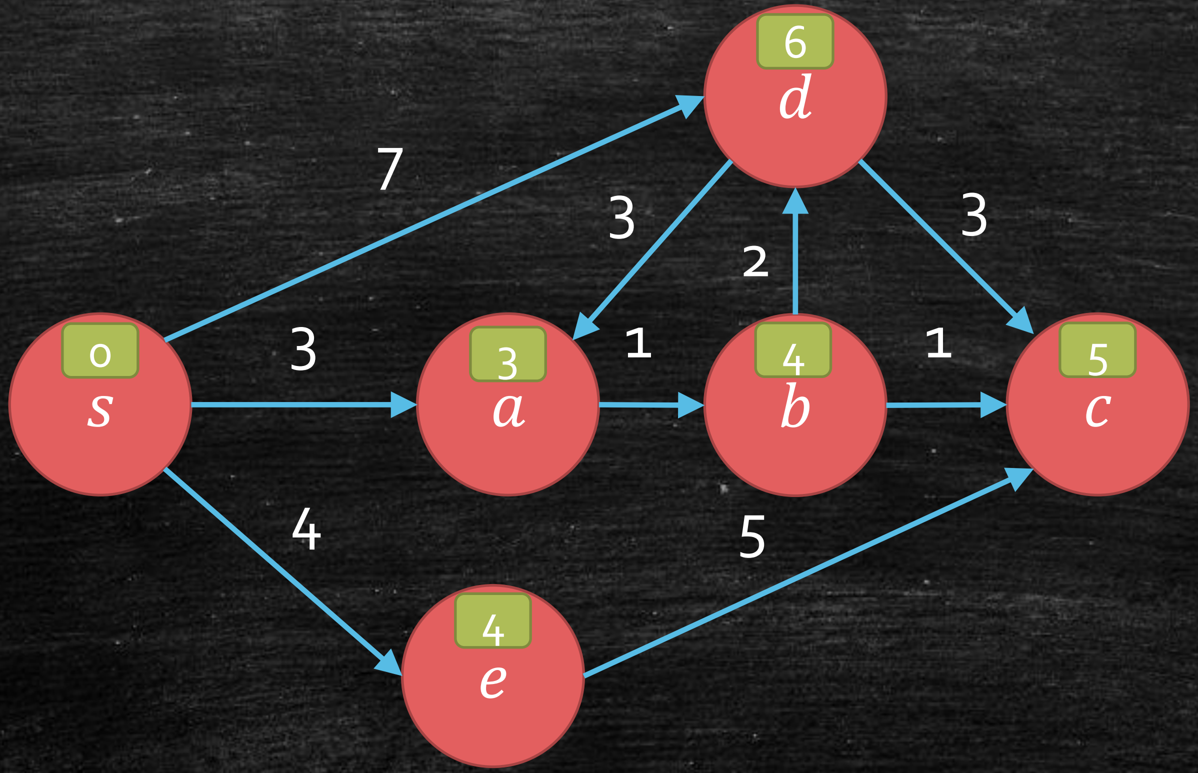
Sample Run



Sample Run



Sample Run



Output a path?

Dijkstra$(G = (V, E), s)$

1. **Initialize**
   - $T \leftarrow \{s\}$
   - $tdist[v] \leftarrow w(s, v),\ pre[v] \leftarrow s$ for all $(s, v) \in E$.
2. **Explore**
   - Find $v \notin T$ with smallest $tdist[v]$.
   - $T \leftarrow T + \{v\}$
3. **Update $tdist[u]$**
   - $tdist[u] = \min\{tdist[u], tdist[v] + w(v, u)\}$ for all $(v, u) \in E$.
   - If $tdist[u]$ is updated, then $pre[u] \leftarrow v$.

Time Complexity

Dijkstra$(G = (V, E), s)$

1. **Initialize**
   - $T \leftarrow \{s\}$
   - $tdist[v] \leftarrow w(s,v)$, $pre[v] \leftarrow s$ for all $(s,v) \in E$.
2. **Explore**
   - Find $v \notin T$ with smallest $tdist[v]$.
   - $T \leftarrow T + \{v\}$
3. **Update $tdist[u]$**
   - $tdist[u] = \min\{tdist[u], tdist[v] + w(v,u)\}$ for all $(v,u) \in E$.
   - If $tdist[u]$ is updated, then $pre[u] \leftarrow v$.

$|V|$ rounds

$|E|$ rounds

$|E|$ rounds

Time Complexity: Conclusion

- Find Min
  - $|V|$ rounds
- Update
  - |E| rounds
- If we use simple array, then
  - First round find min: $|V| - 1$
  - Second round find min: $|V| - 2$
  - …
  - Find min totally: $O(|V|^2)$
  - Each update: $O(1)$
  - Update totally: $O(|E|)$
  - Algorithm totally: $\boldsymbol{O(|V|^2 + |E|)}$

Improve Dijkstra by Heap!

- Find Min
  - $|V|$ rounds
- Update
  - |E| rounds
- What about heap?

	Pop Min	Insert	Update Key	Merge
Binary Heap	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
d-nary Heap	$O(d\log_d n)$	$O(\log_d n)$	$O(\log_d n)$	$O(n)$
Binomial Heap	$O(\log n)$	$O(1)$	$O(\log n)$	$O(\log n)$
Fibonacci	$O(\log n)$	$O(1)$	$O(1)$	$O(1)$

Only Decreasing

Improve Dijkstra by Heap!

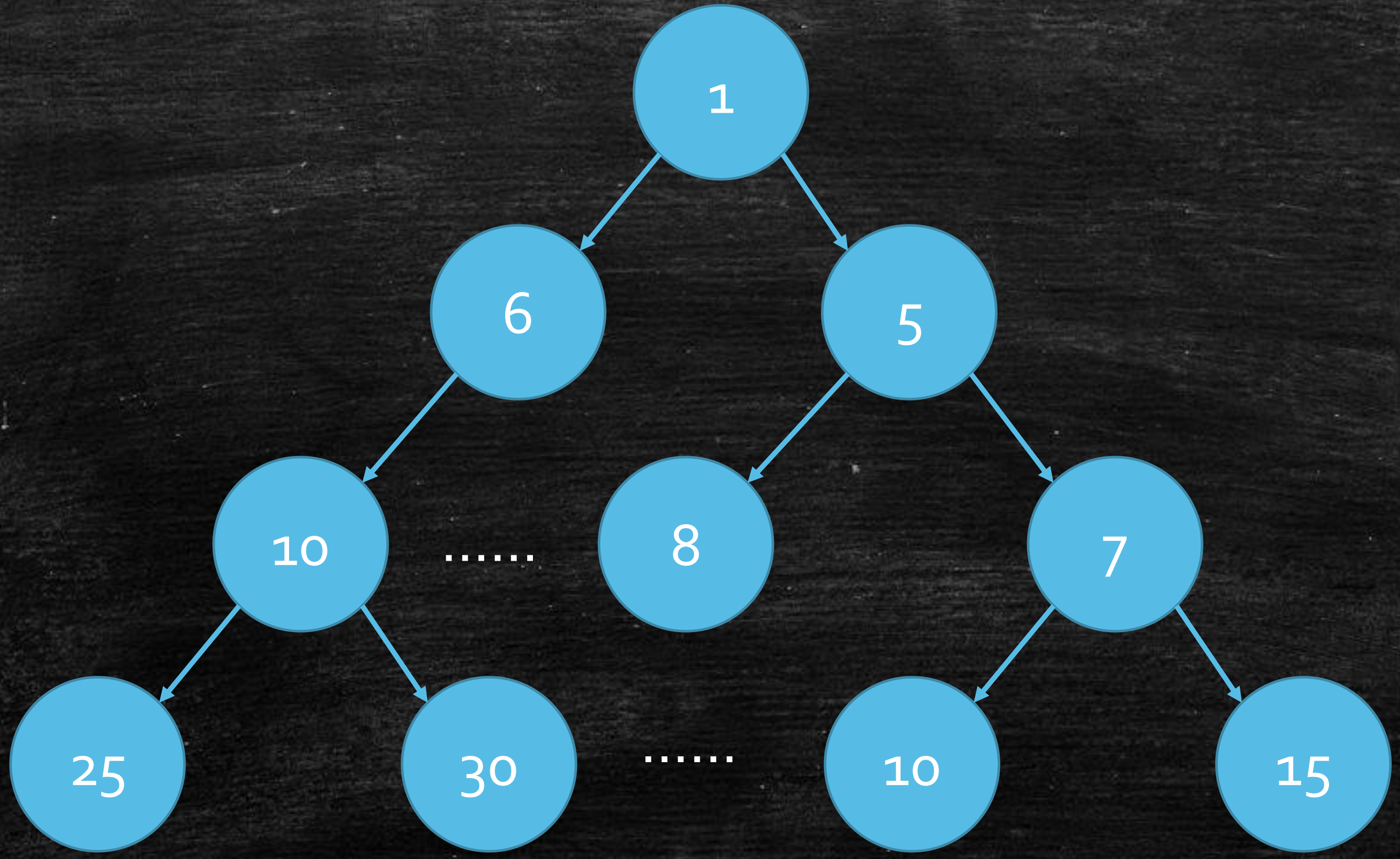
Find Min: $|V|$ rounds
Update: |E| rounds

Array: $O(|V|^2 + |E|)$

- Binary Heap
  - Find Min: $O(|V| \log |V|)$
  - Update: $O(|E| \log |V|)$
  - Totally: $\boldsymbol{O((|V| + |E|) \log |V|)}$
- d-nary Heap
  - Find Min: $O(|V| d \log_d |V|)$
  - Update: $O(|E| \log_d |V|)$
  - Set $d = |E|/|V|$
  - Totally: $\boldsymbol{O(|E| \log_{|E|/|V|} |V|)}$
- Fibonacci Heap
  - Find Min: $O(|V| \log |V|)$
  - Update: $O(|E|)$
  - Totally: $\boldsymbol{O(|E| + |V| \log |V|)}$

	Pop Min	Insert	Update Key	Merge
Binary Heap	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
d-nary Heap	$O(d \log_d n)$	$O(\log_d n)$	$O(\log_d n)$	$O(n)$
Binomial Heap	$O(\log n)$	$O(1)$	$O(\log n)$	$O(\log n)$
Fibonacci	$O(\log n)$	$O(1)$	$O(1)$	$O(1)$

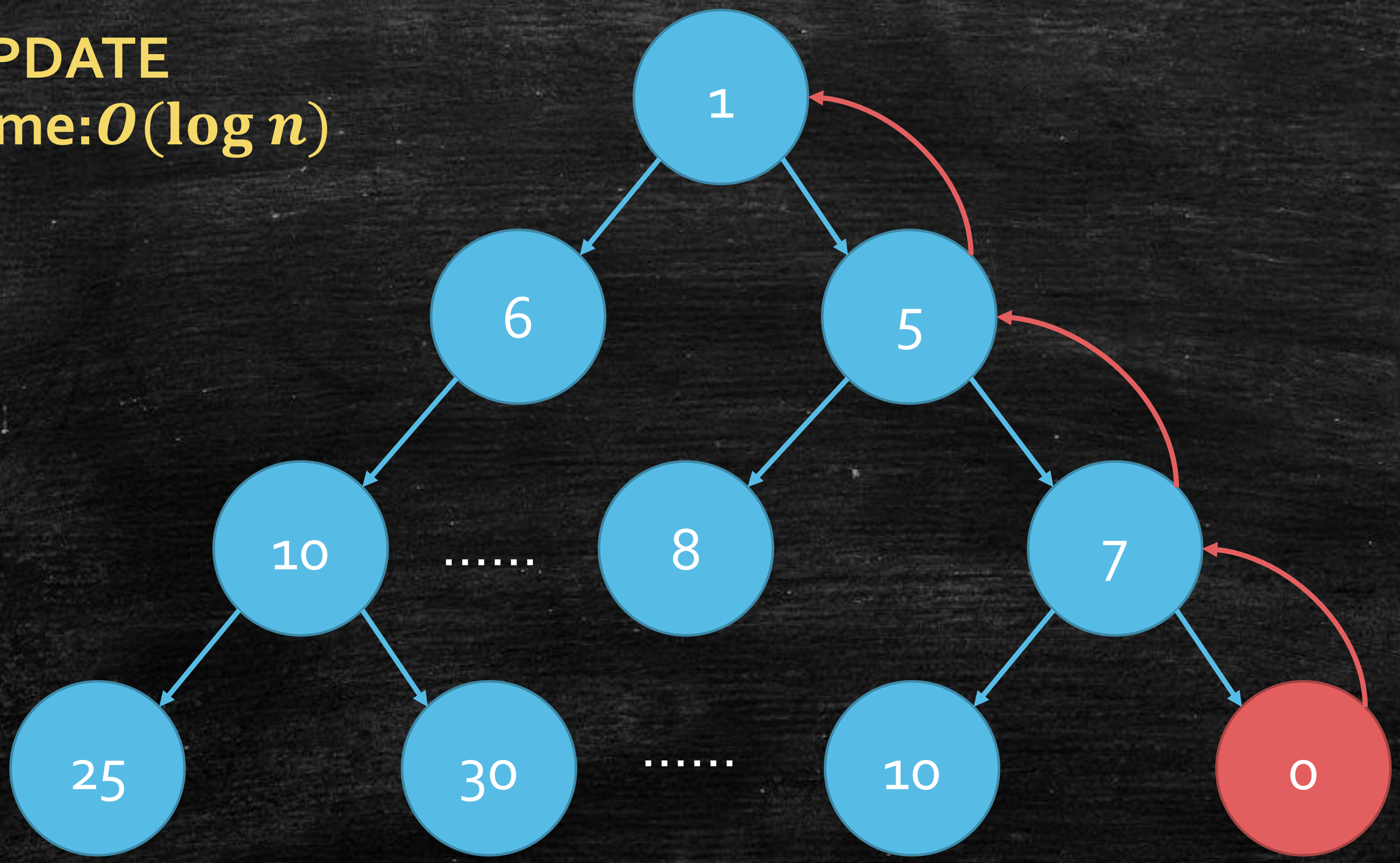
Quick Review (or Preview?): Binary Heap



Let us only discuss POPMIN and Update

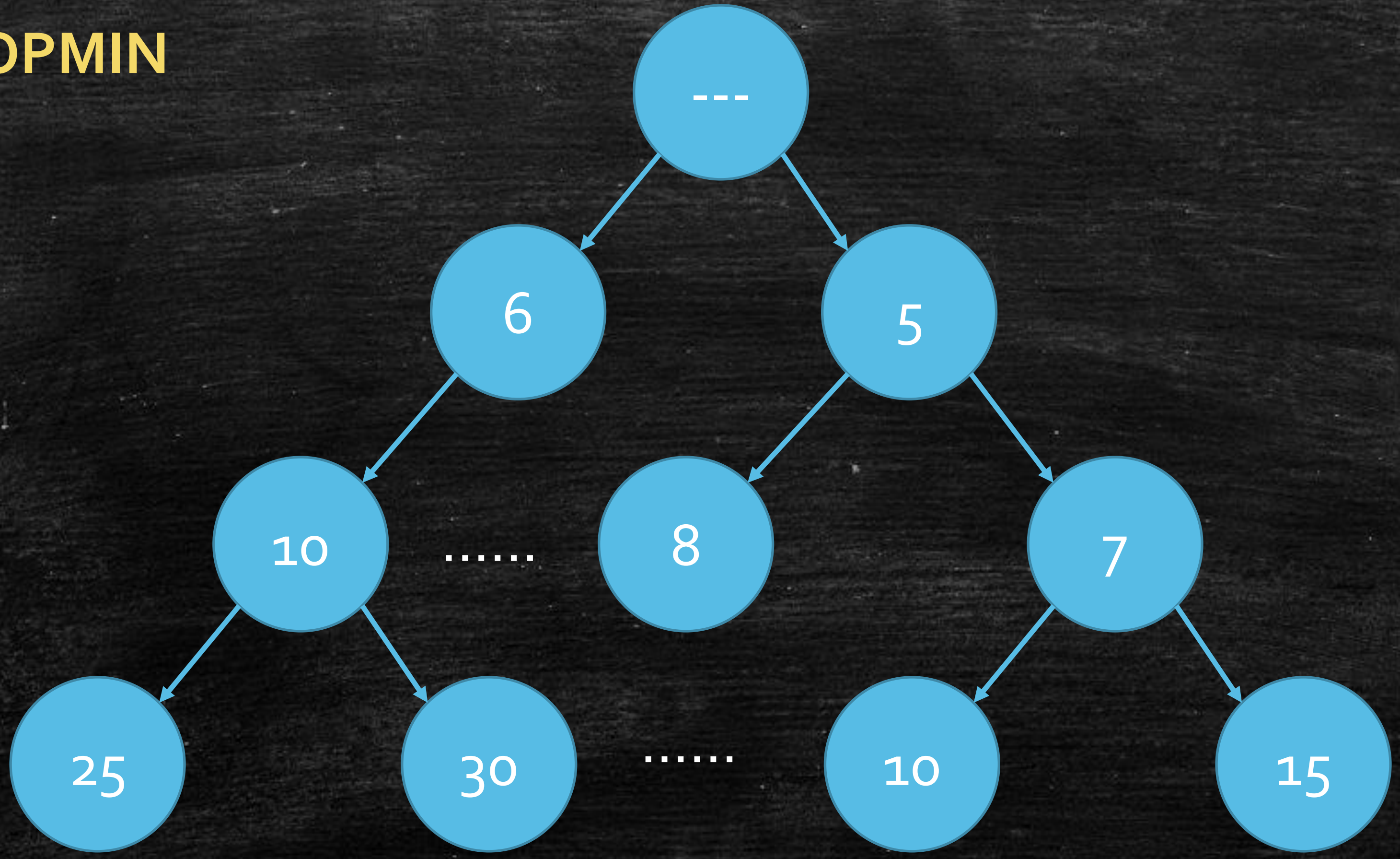
Quick Review (or Preview?): Binary Heap

UPDATE
Time: $O(\log n)$



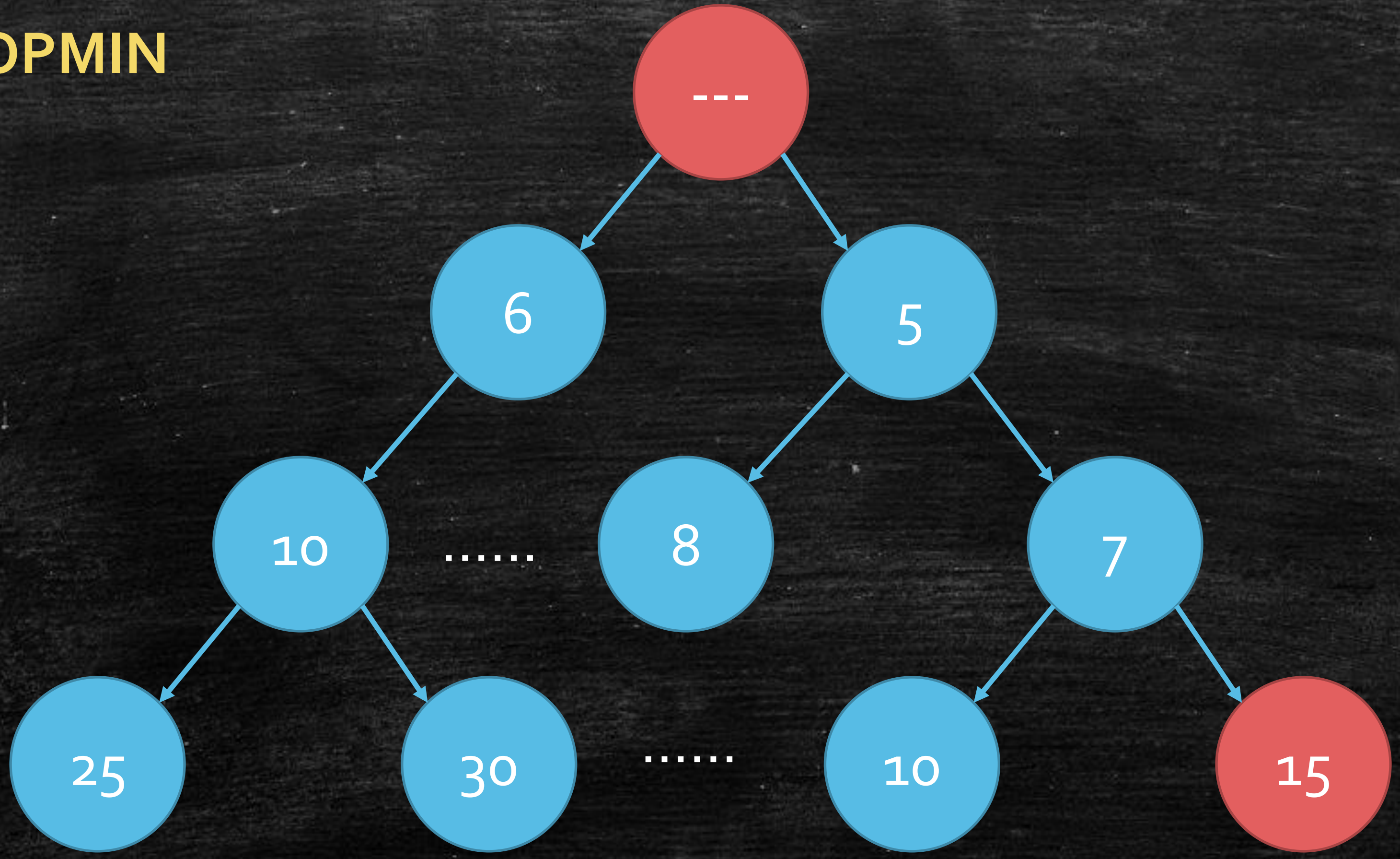
Quick Review (or Preview?): Binary Heap

POPMIN



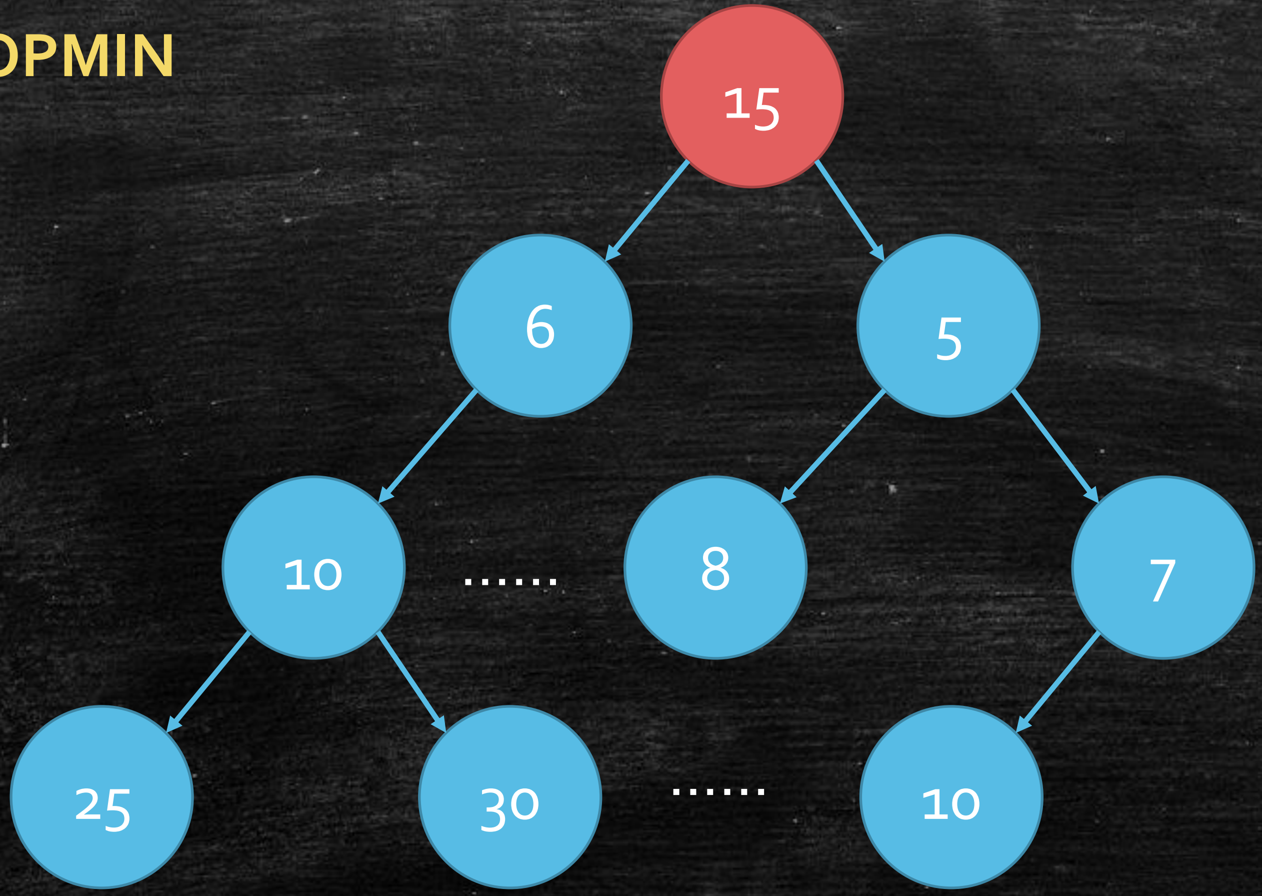
Quick Review (or Preview?): Binary Heap

POPMIN



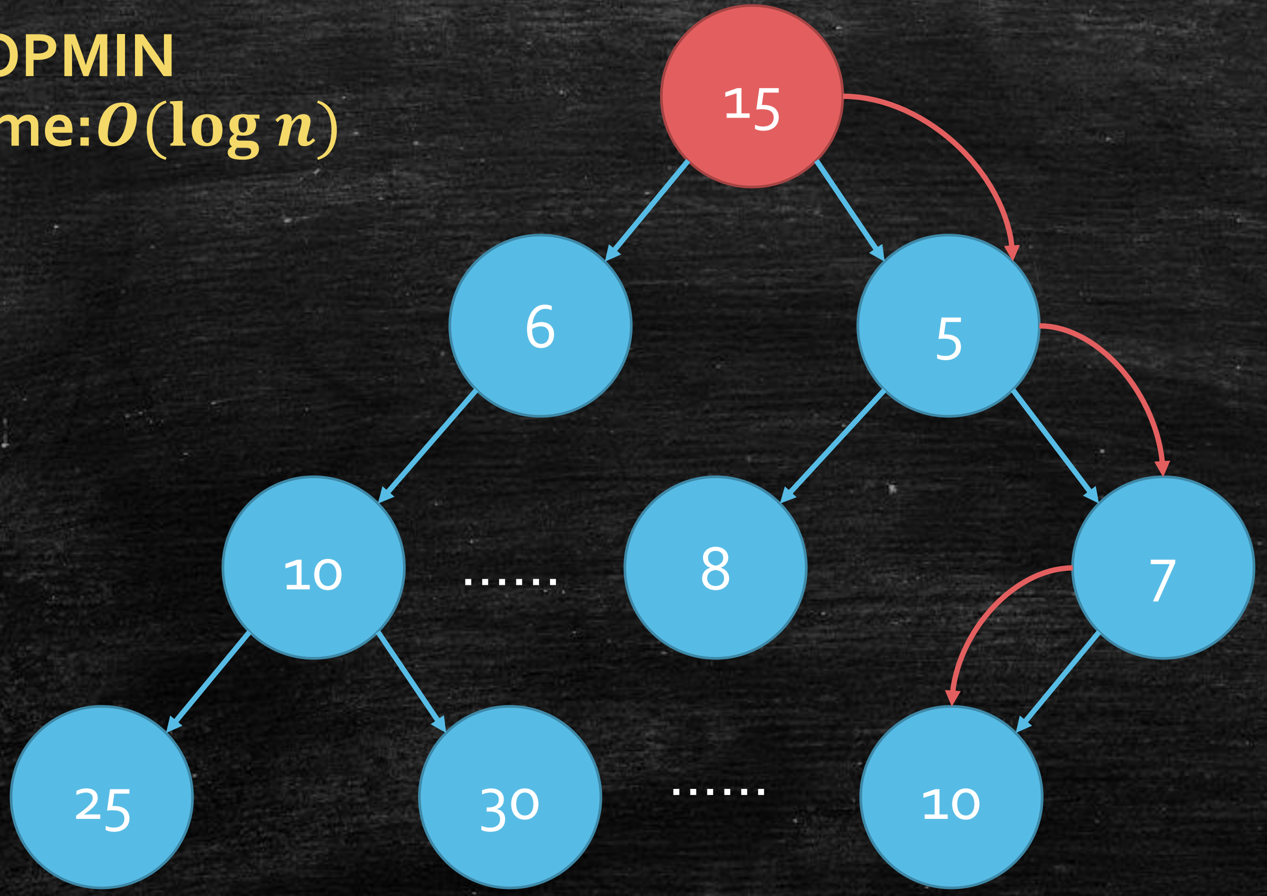
Quick Review (or Preview?): Binary Heap

POPMIN



Quick Review (or Preview?): Binary Heap

POPMIN
Time: $O(\log n)$

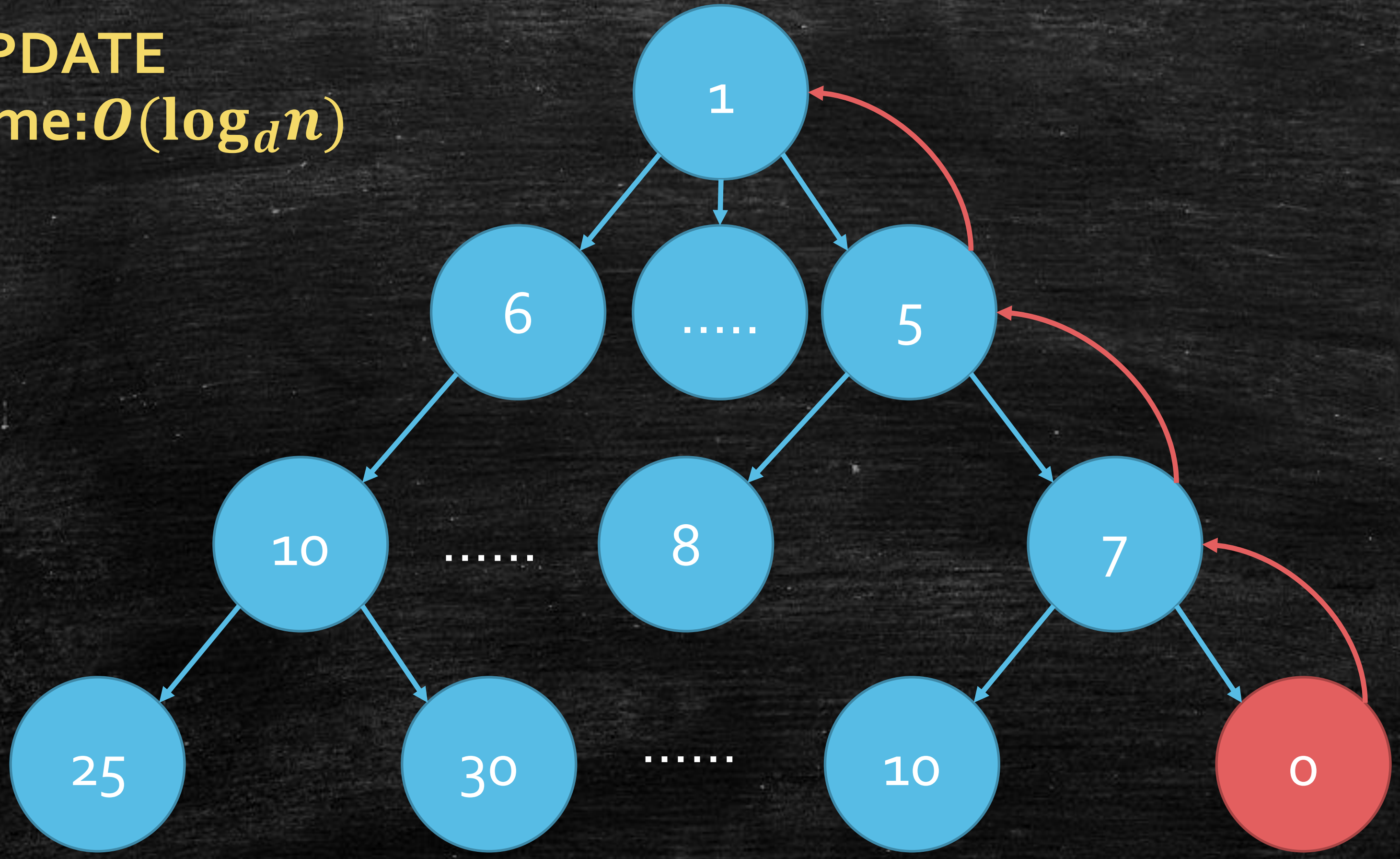


Why the two operations are good?

It keep the tree balanced!

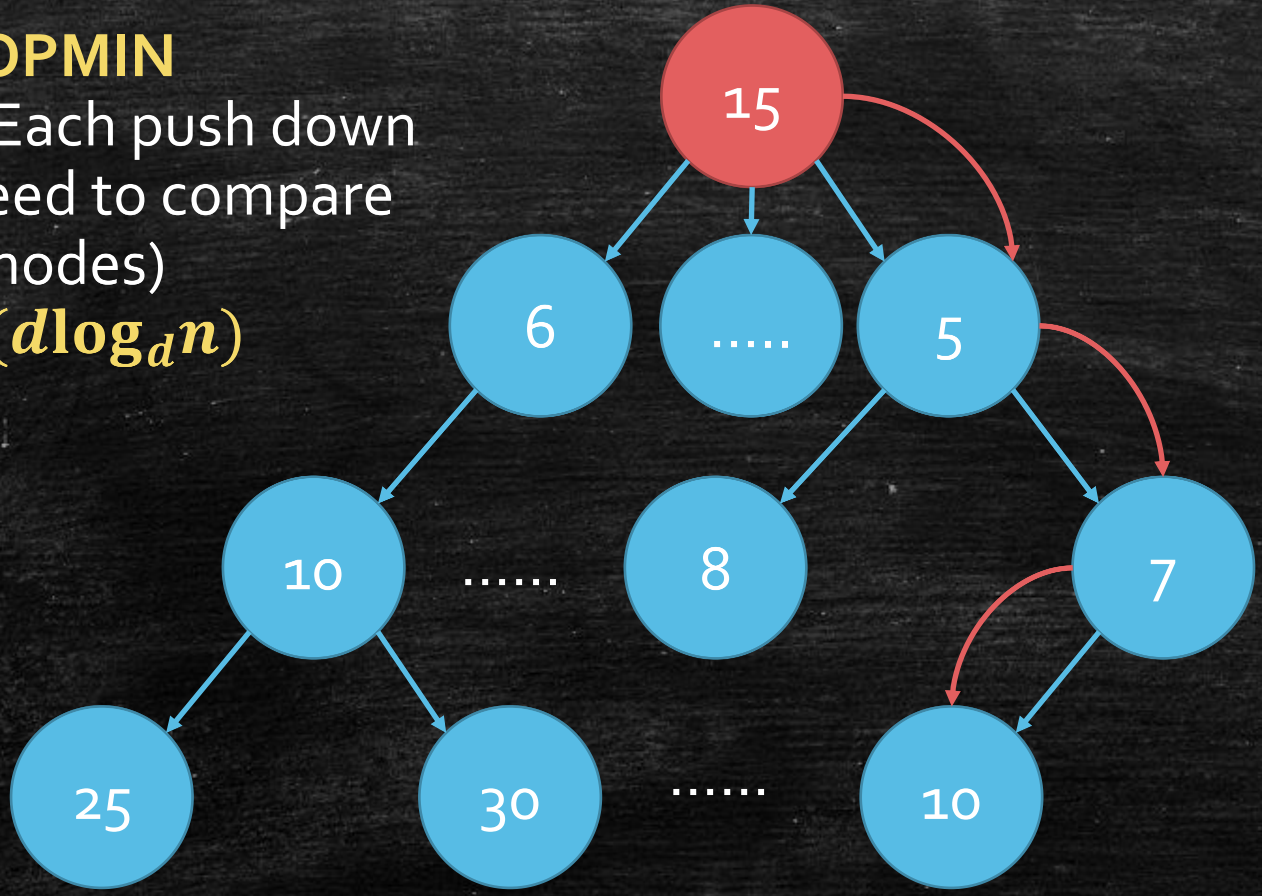
Quick Review (or Preview?): d-nary Heap

UPDATE
Time: $O(\log_d n)$



Quick Review (or Preview?): d-nary Heap

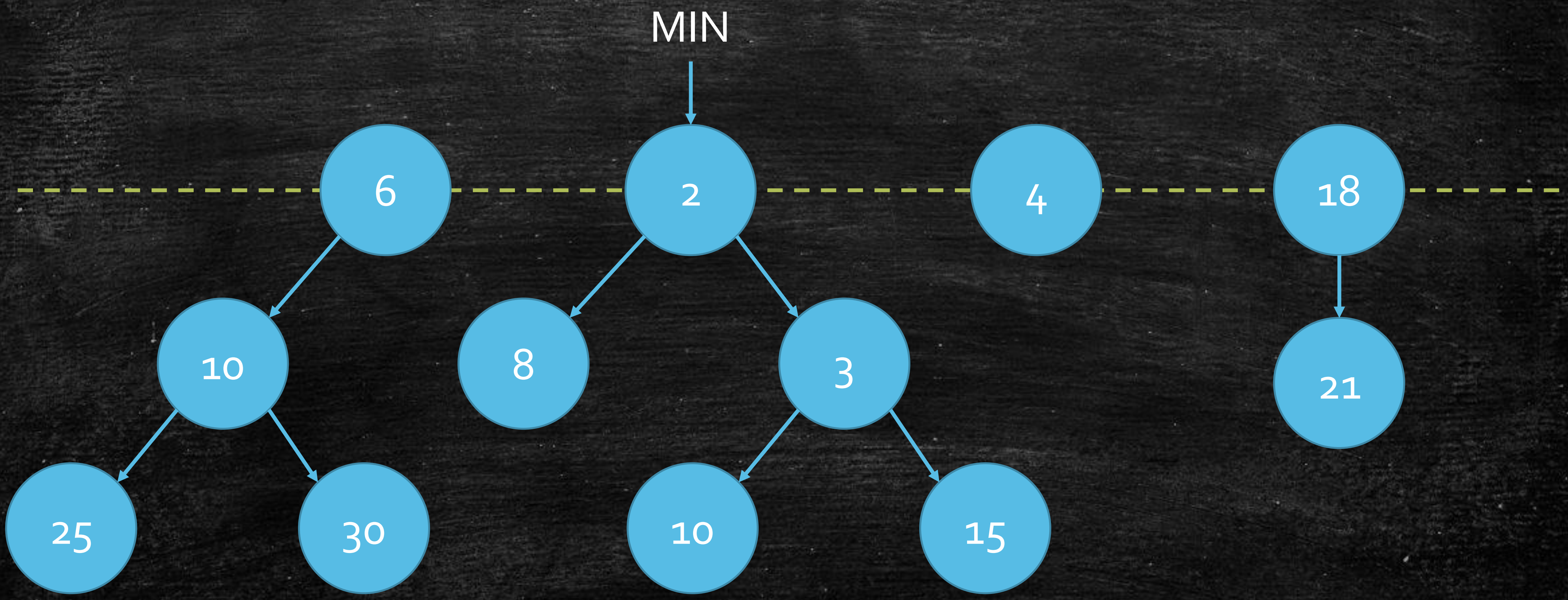
POPMIN
(*Each push down Need to compare d nodes)
$\boldsymbol{O(d\log_d n)}$



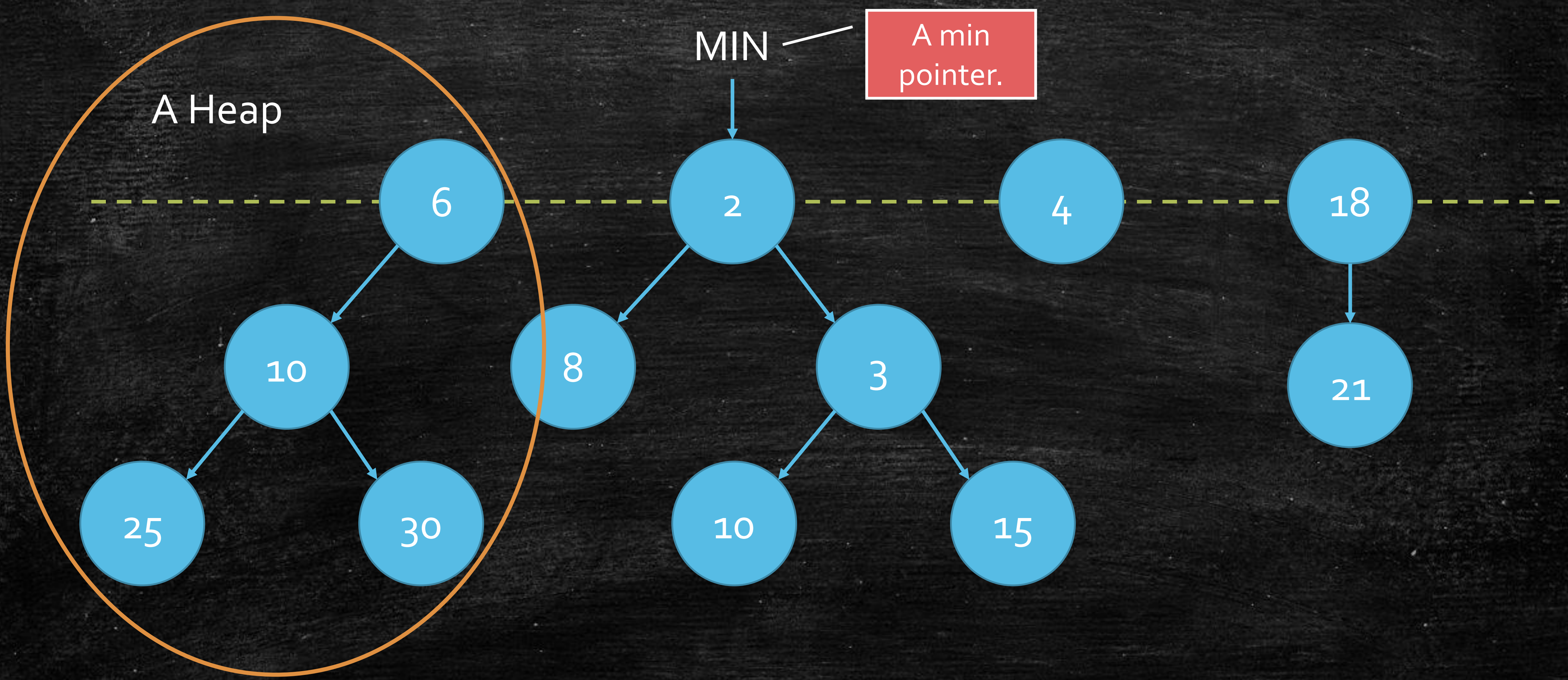
Quick Review (or Preview?): Fibonacci Heap

	Pop Min	Insert	Update Key	Merge
Binary Heap	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
d-nary Heap	$O(d\log_d n)$	$O(\log_d n)$	$O(\log_d n)$	$O(n)$
Binomial Heap	$O(\log n)$	$O(1)$	$O(\log n)$	$O(\log n)$
Fibonacci	$O(\log n)$	$O(1)$	$O(1)$	$O(1)$

Quick Review (or Preview?): Fibonacci Heap

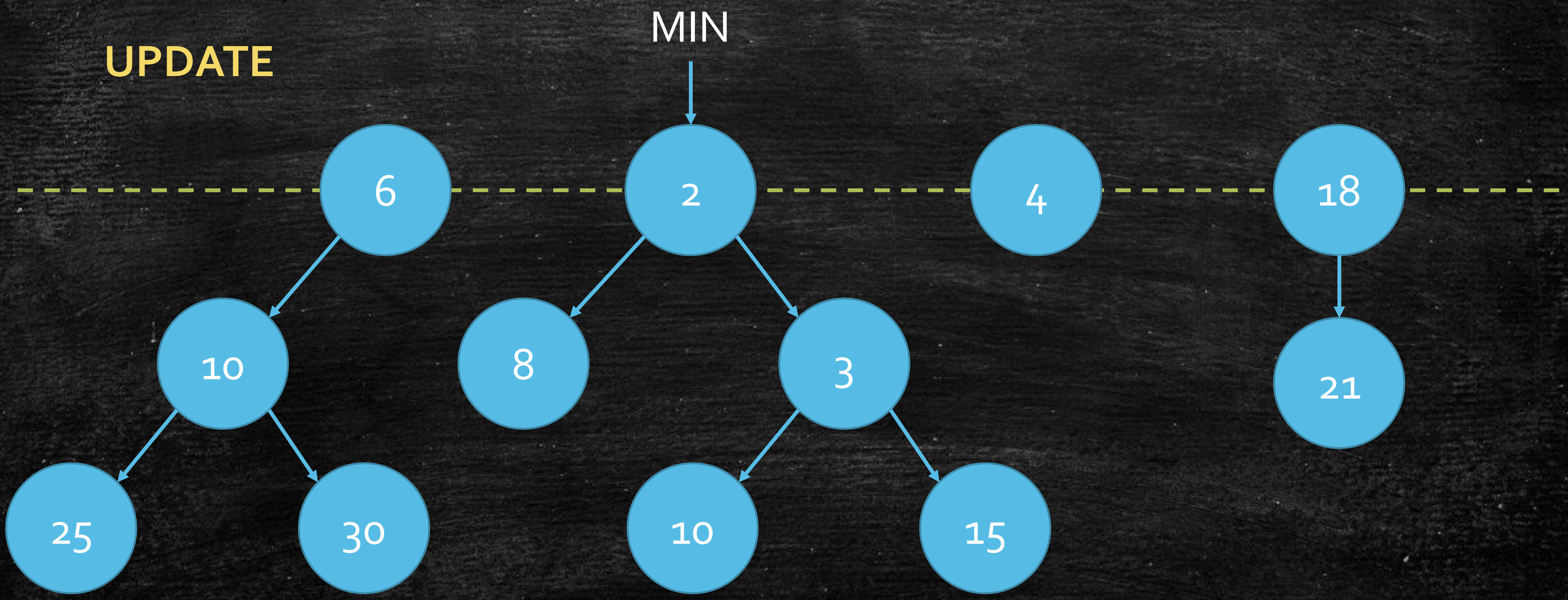


Quick Review (or Preview?): Fibonacci Heap

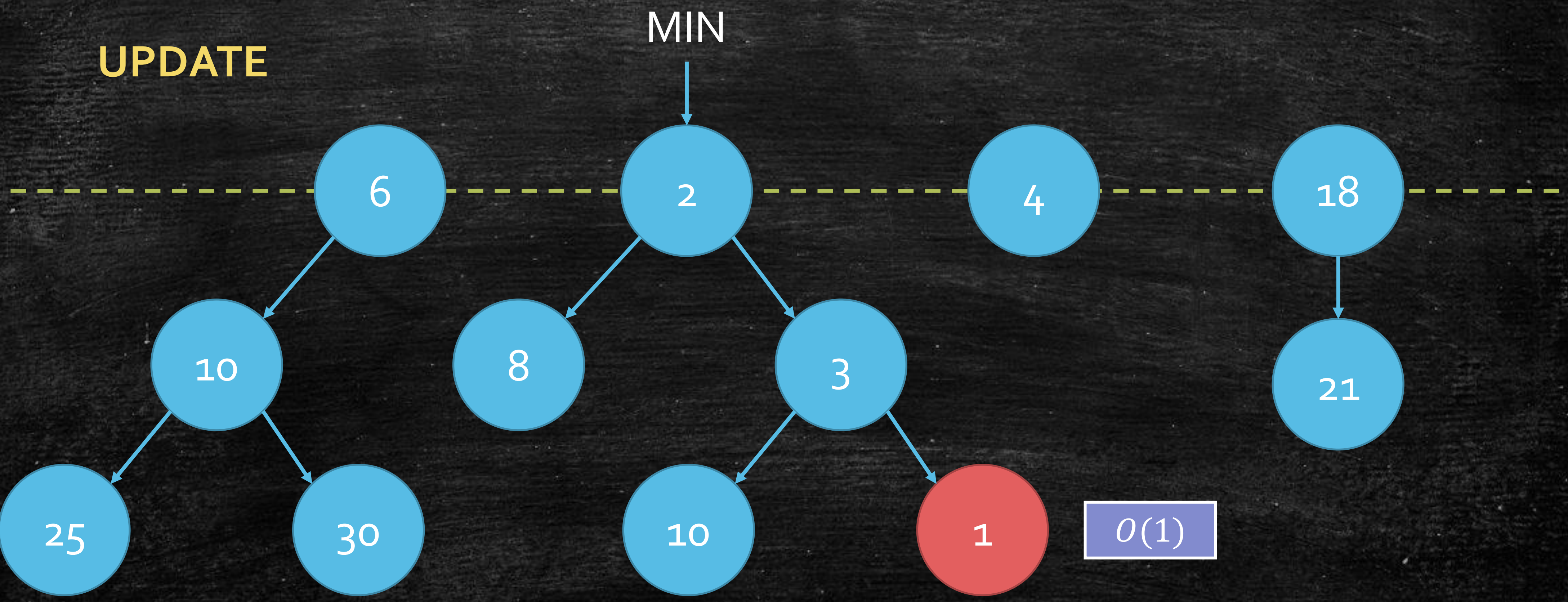


A magic Idea of UPDATE!

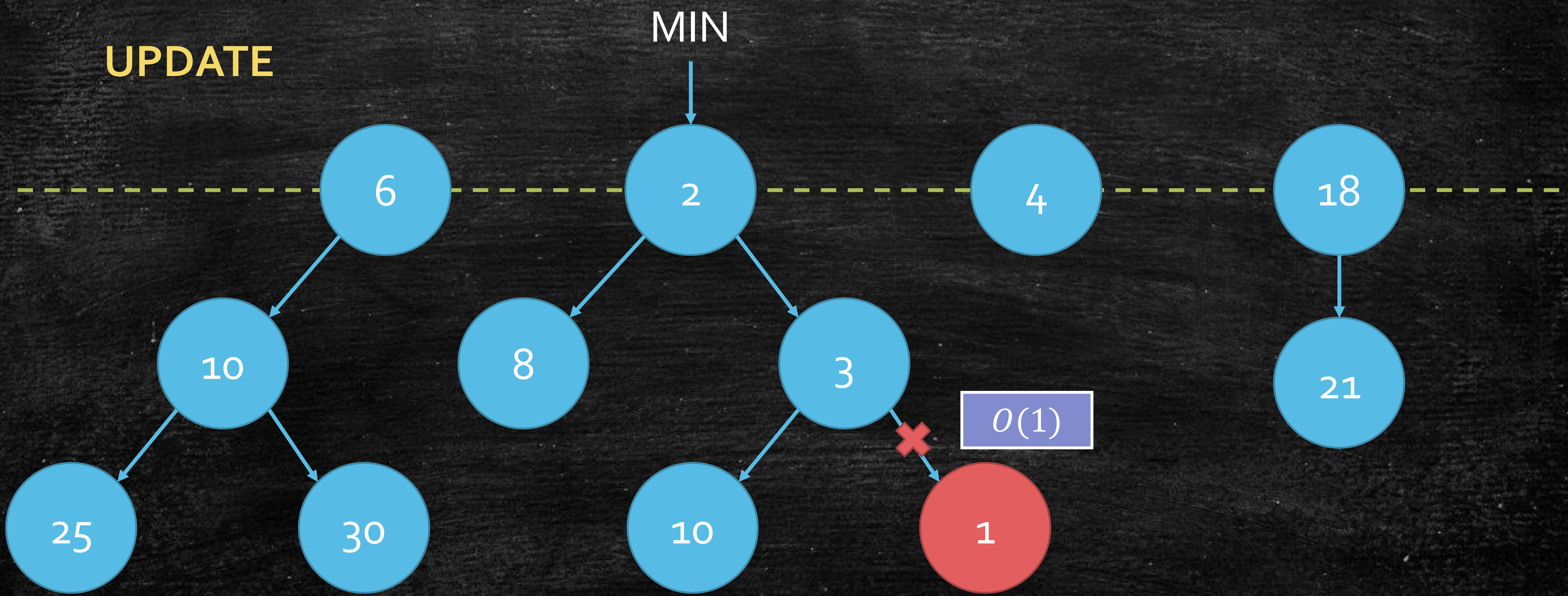
Quick Review (or Preview?): Fibonacci Heap



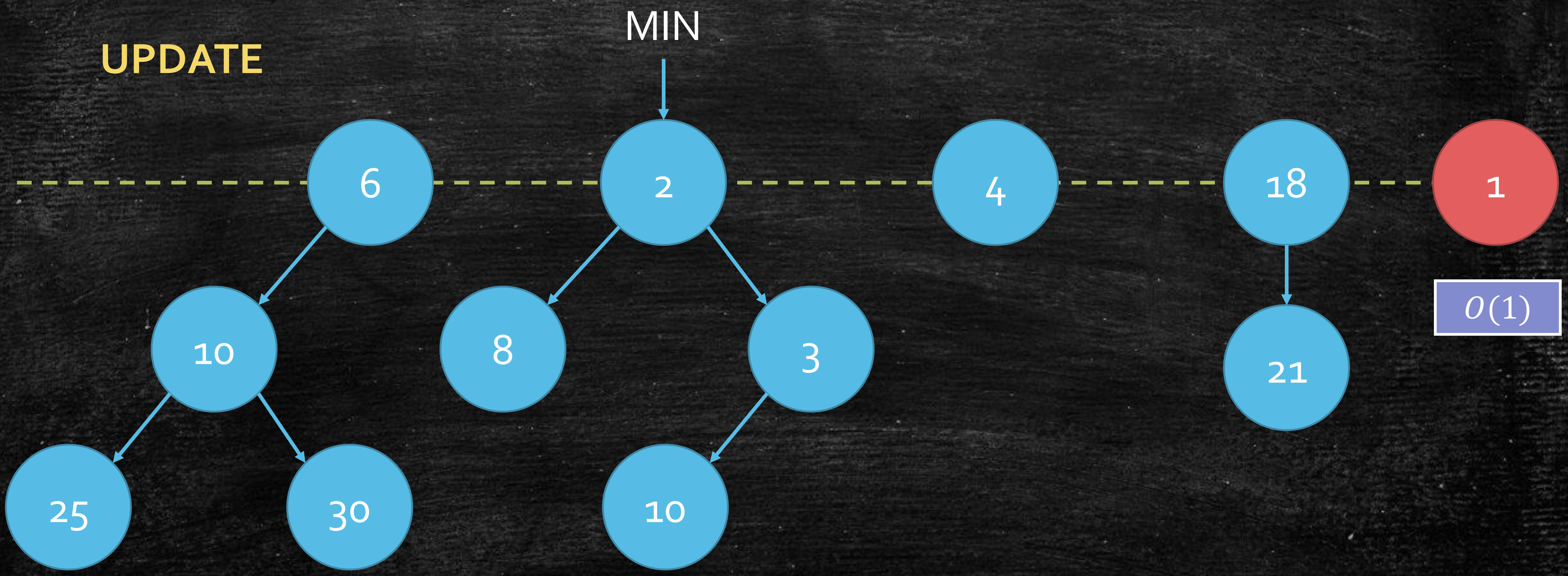
Quick Review (or Preview?): Fibonacci Heap



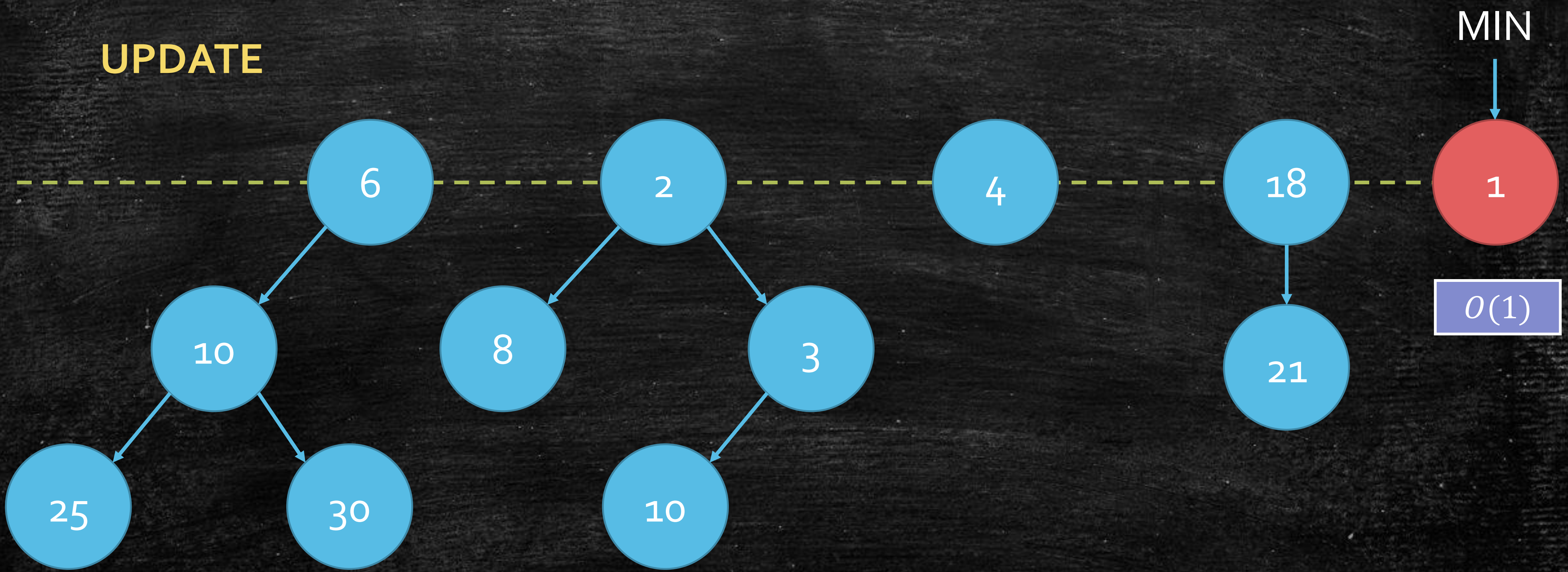
Quick Review (or Preview?): Fibonacci Heap



Quick Review (or Preview?): Fibonacci Heap

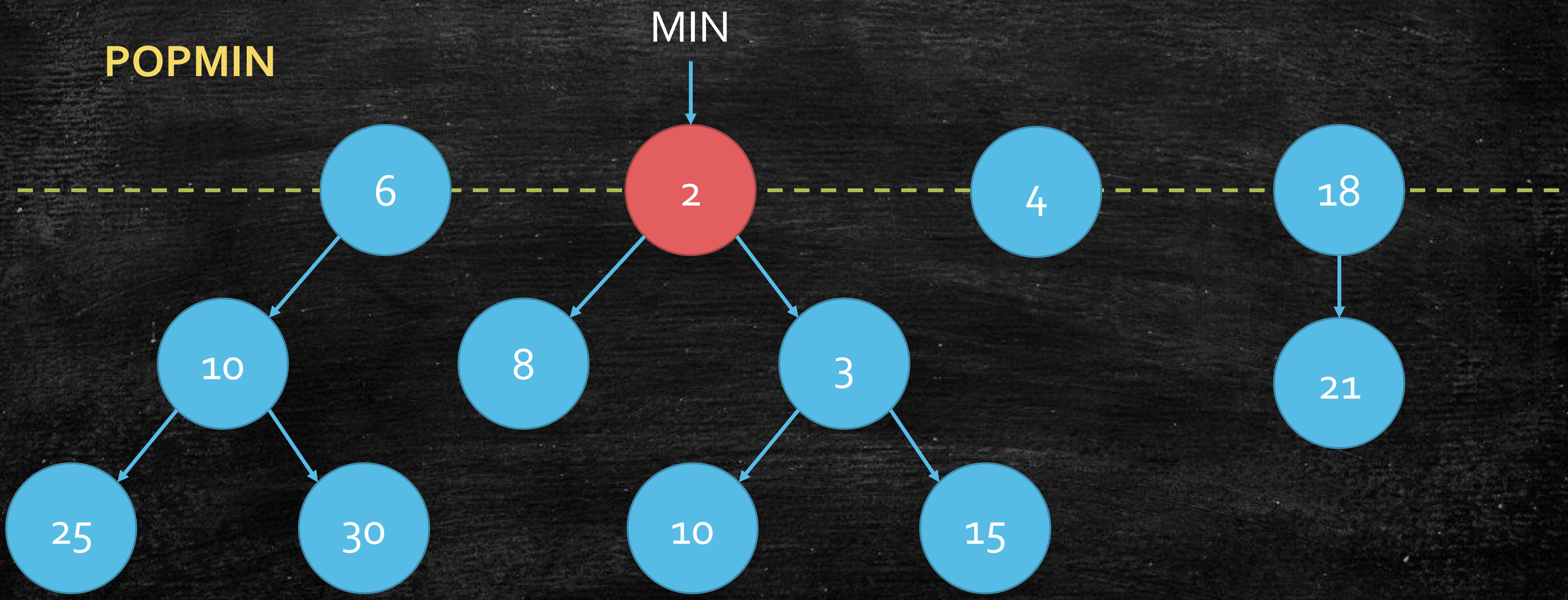


Quick Review (or Preview?): Fibonacci Heap

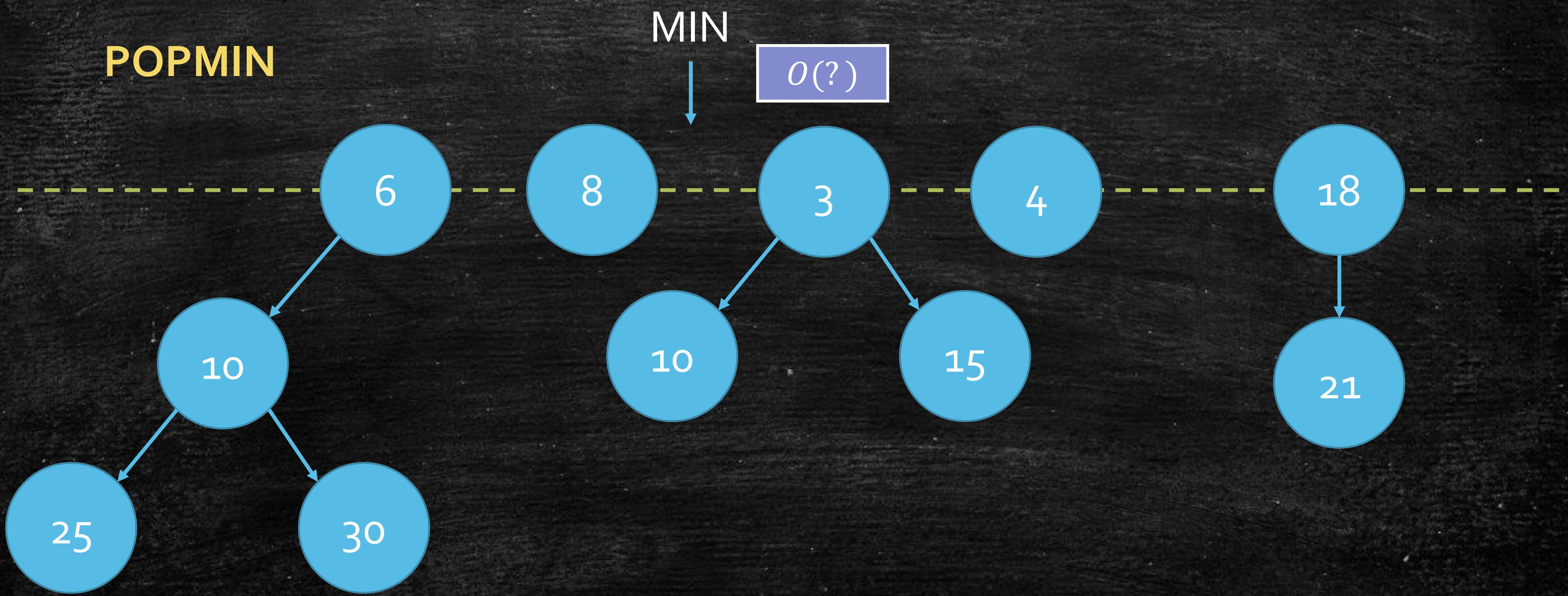


Quick Review (or Preview?): Fibonacci Heap

POPMIN

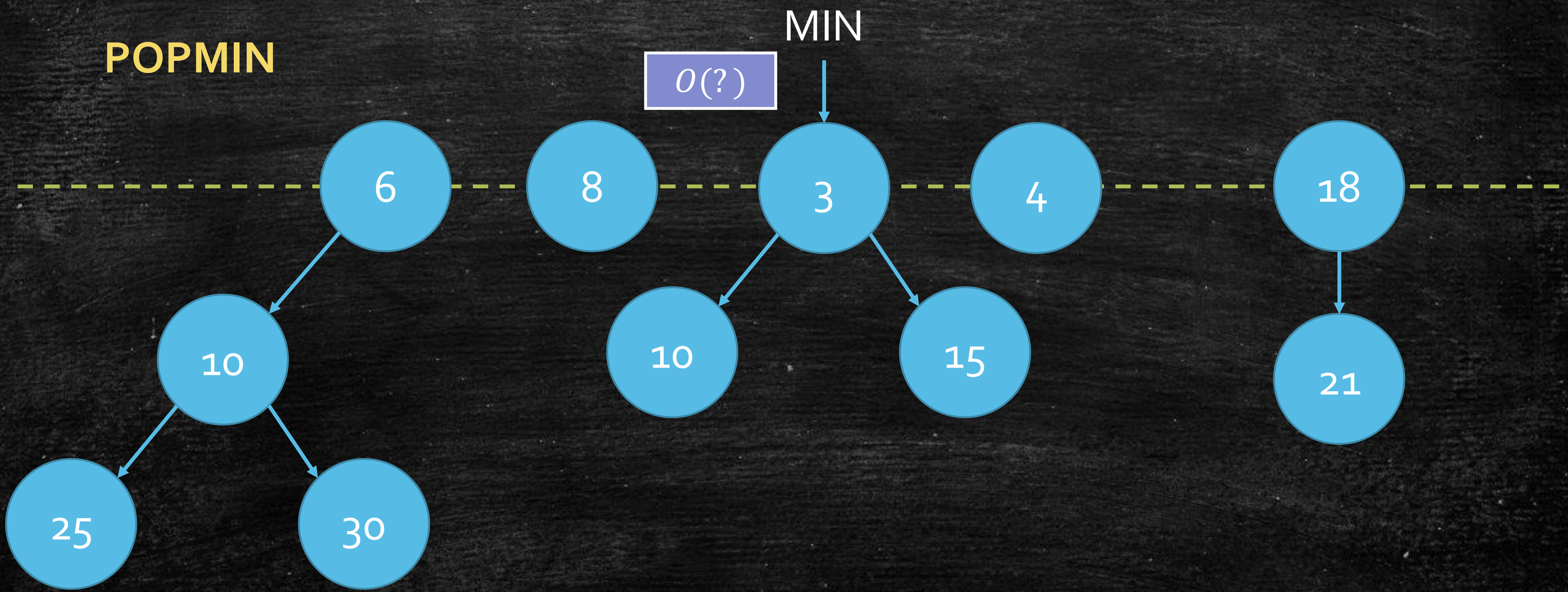


Quick Review (or Preview?): Fibonacci Heap



Quick Review (or Preview?): Fibonacci Heap

POPMIN



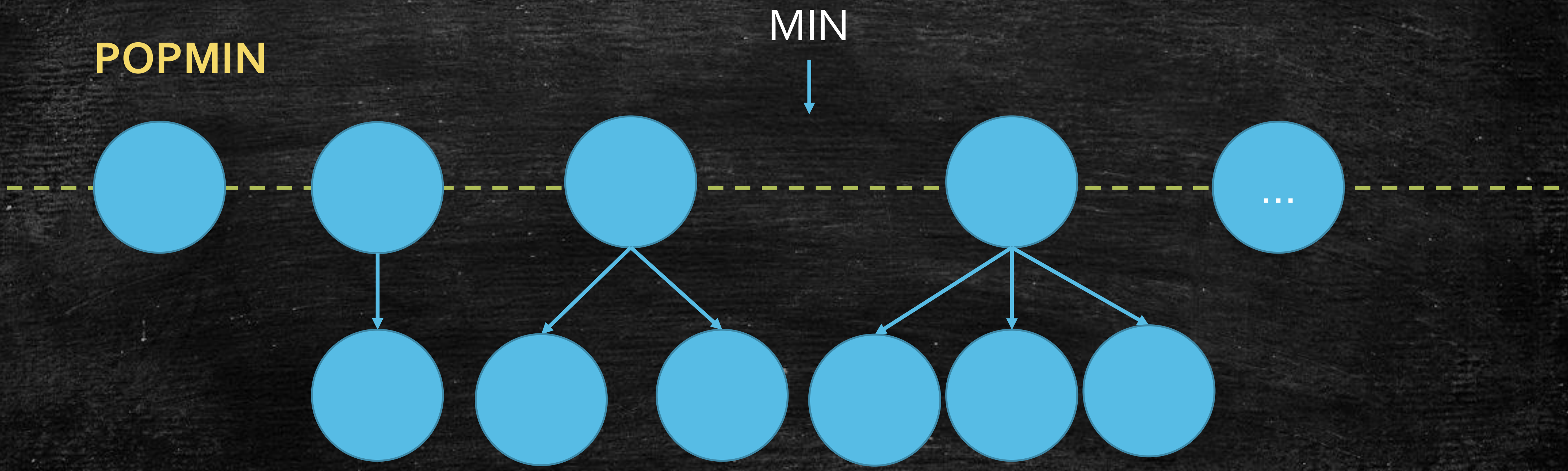
Still many problems!

- Update seems good: $\boldsymbol{O(1)}$
- Pop Min need to compare all the roots?
- Running Time of POPMIN
  - t^-: the root number before POPMIN.
  - D: The max degree of all root.
  - It needs $O(t^- + D)$.
- It can be very bad: $\Omega(\boldsymbol{n})$!

Two Tasks: How to make POPMIN fast?

- Task 1: Bound D: max degree.
- Task 2: Bound t^-.
- Property we want to maintain:
- Each degree at most has one root!
    - 1 root with degree 0, 1 root with degree 1......
  - Bound Largest degree → Bound the number of roots!

Is it enough?



Degree k root is size $k+1$, number of roots = largest degree = $\sqrt{n}$.
It is not enough.

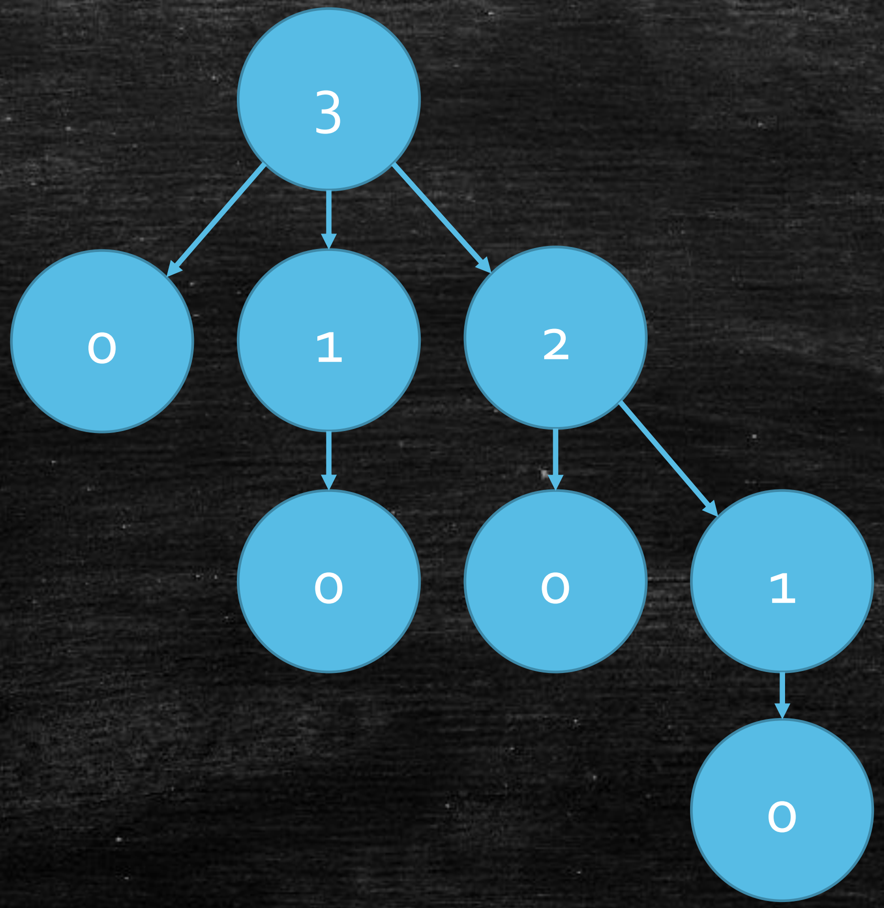
How to make a degree k tree large?

- We want the degree property recursively holds!
- The children of every vertex have the degree property
  - Each degree at most has one root!

How to bound max degree?

- Make the tree heavy!
- We want a claim: a degree k root has at least 2^k descendants.

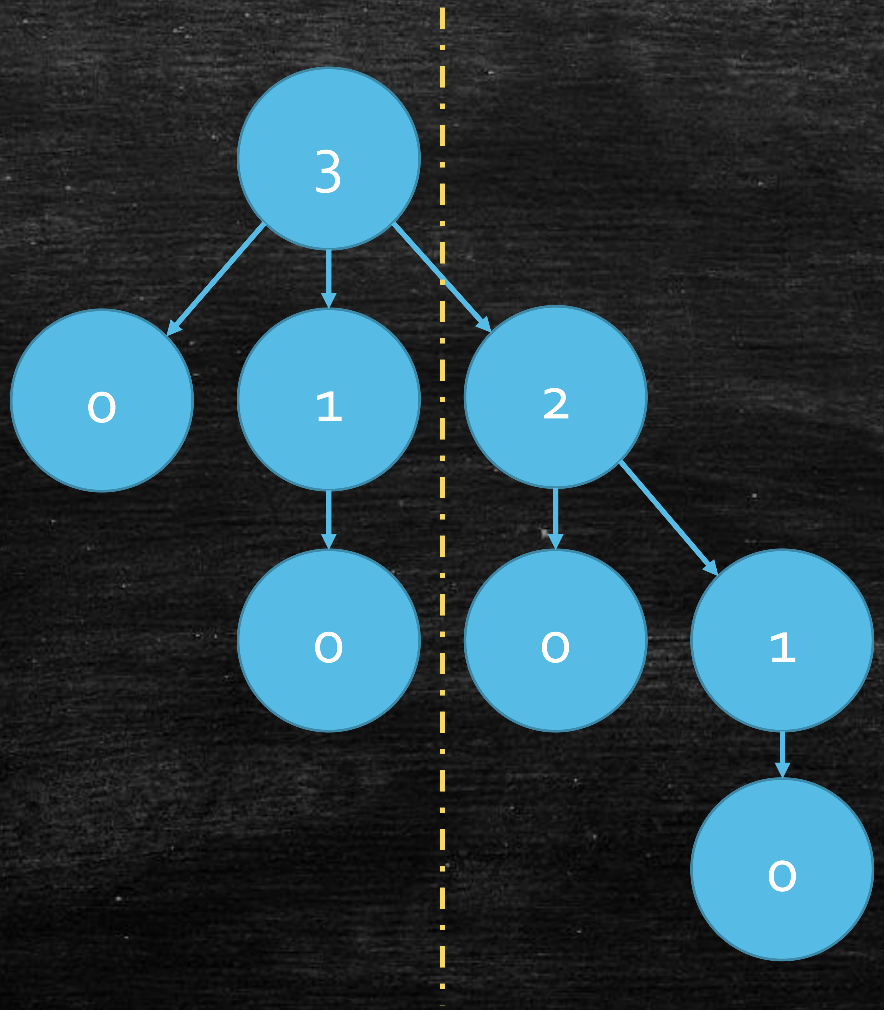
Build a Good Tree (Recall Binomial Heap)



What is the result now?

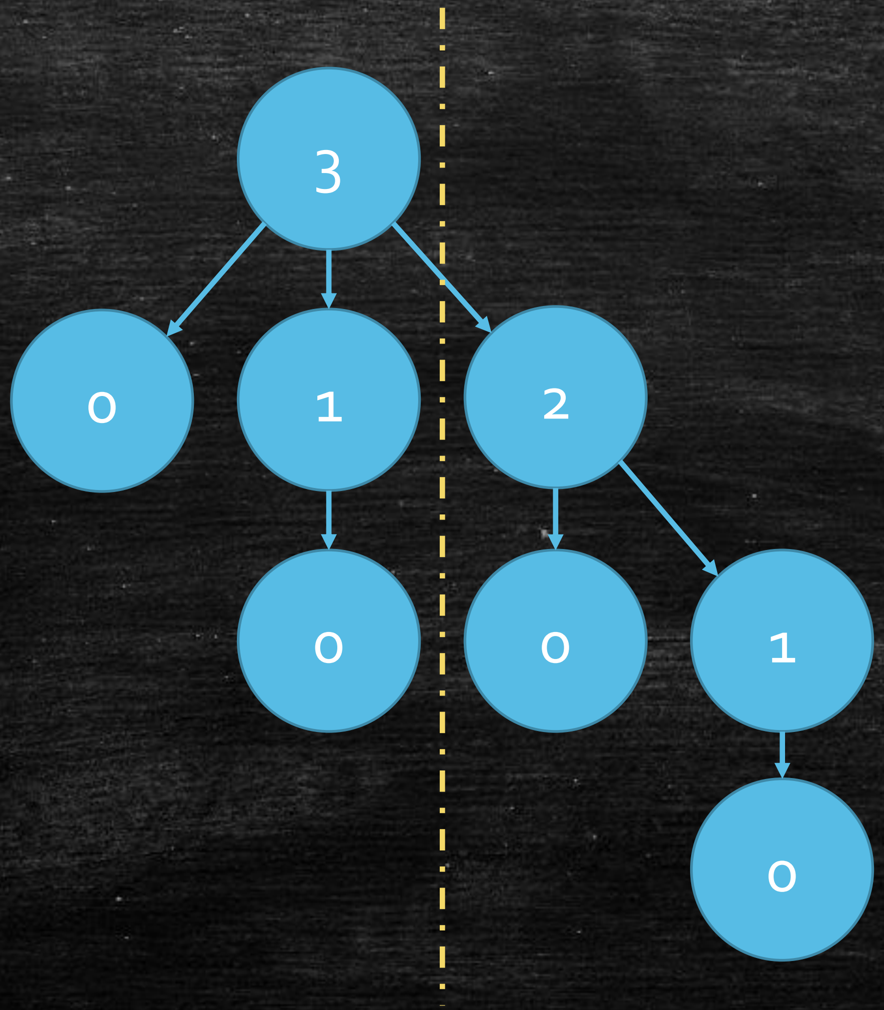
- Assume all trees are **good** in the Fibonacci Heap.
- A degree k **good** tree has $d(k)$ nodes

Build a Good Tree (Recall Binomial Heap)



$$d(0) = 1$$
$$d(1) = 2$$
$$d(k) = \sum_{i=0}^{k-1} d(i) + 1 = 2^{\mathrm{k}}$$

Build a Good Tree (Recall Binomial Heap)



$d(0) = 1$
$d(1) = 2$
$$d(k) = \sum_{i=0}^{k-1} d(i) + 1 = 2^{\mathrm{k}}$$

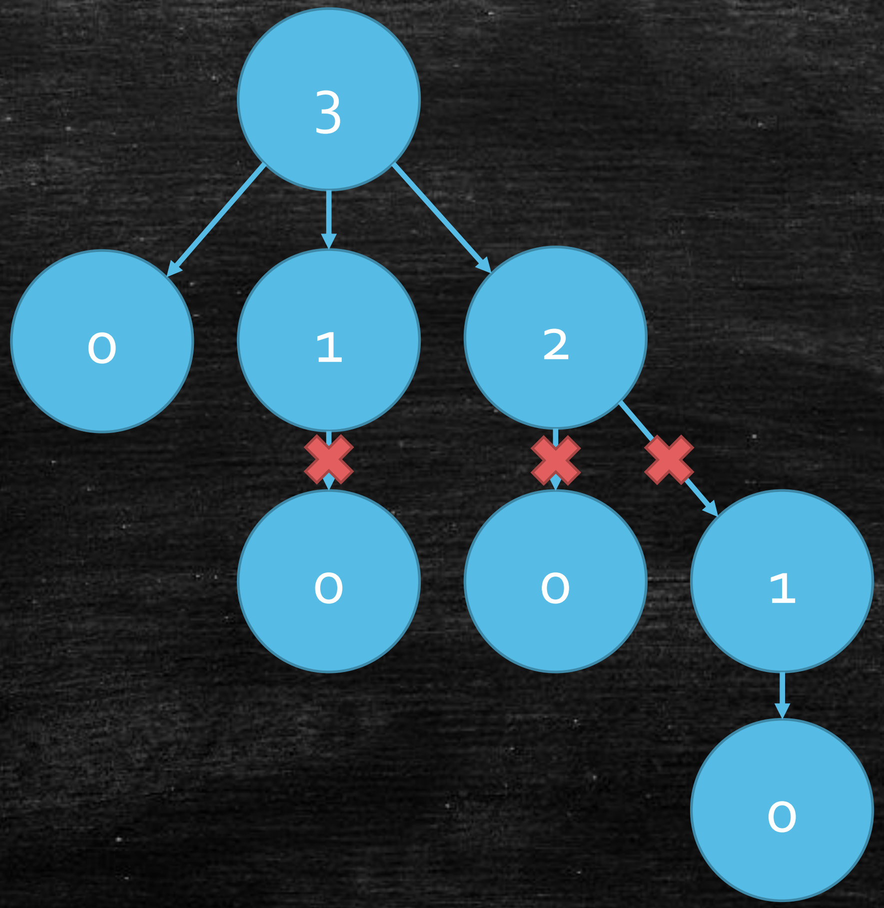
$d(D) = 2^D \leq n$
$\rightarrow D \leq \log n\,!$

But what is the problem?

Cut may break the property.

The good tree may be broken!

UPDATE

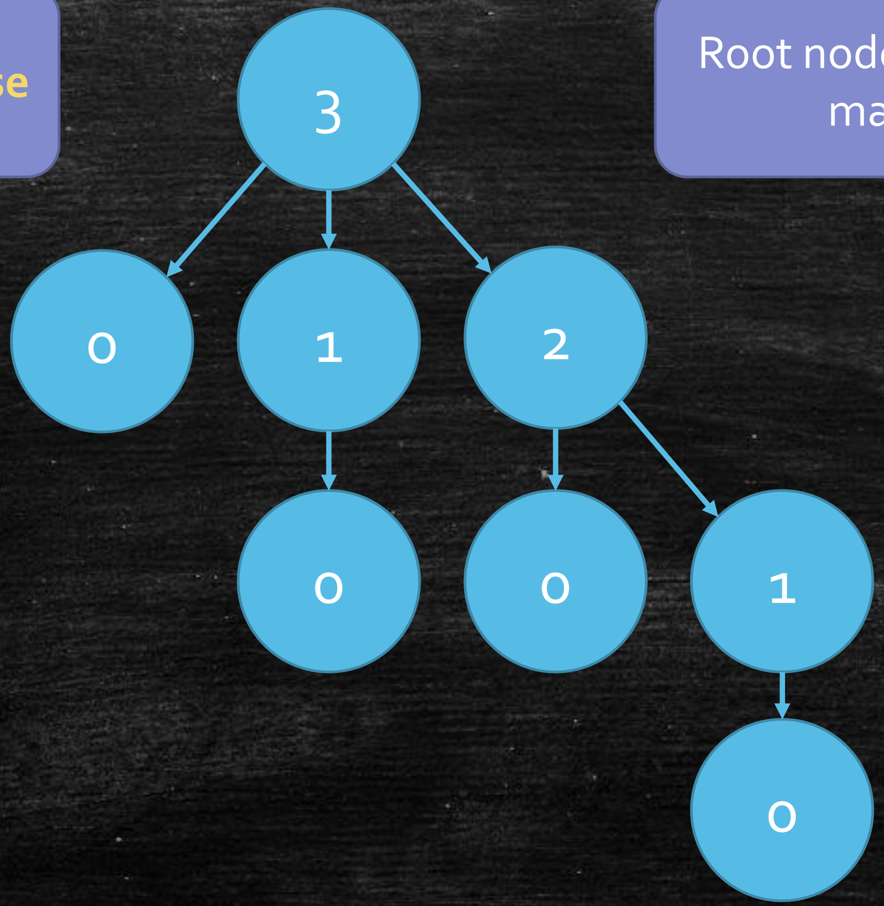


Solution!

- We do not want it to be broken too much!
- Design a rule, to maintain a **slightly weaker** property.

Build a Good Tree (Recall Binomial Heap)

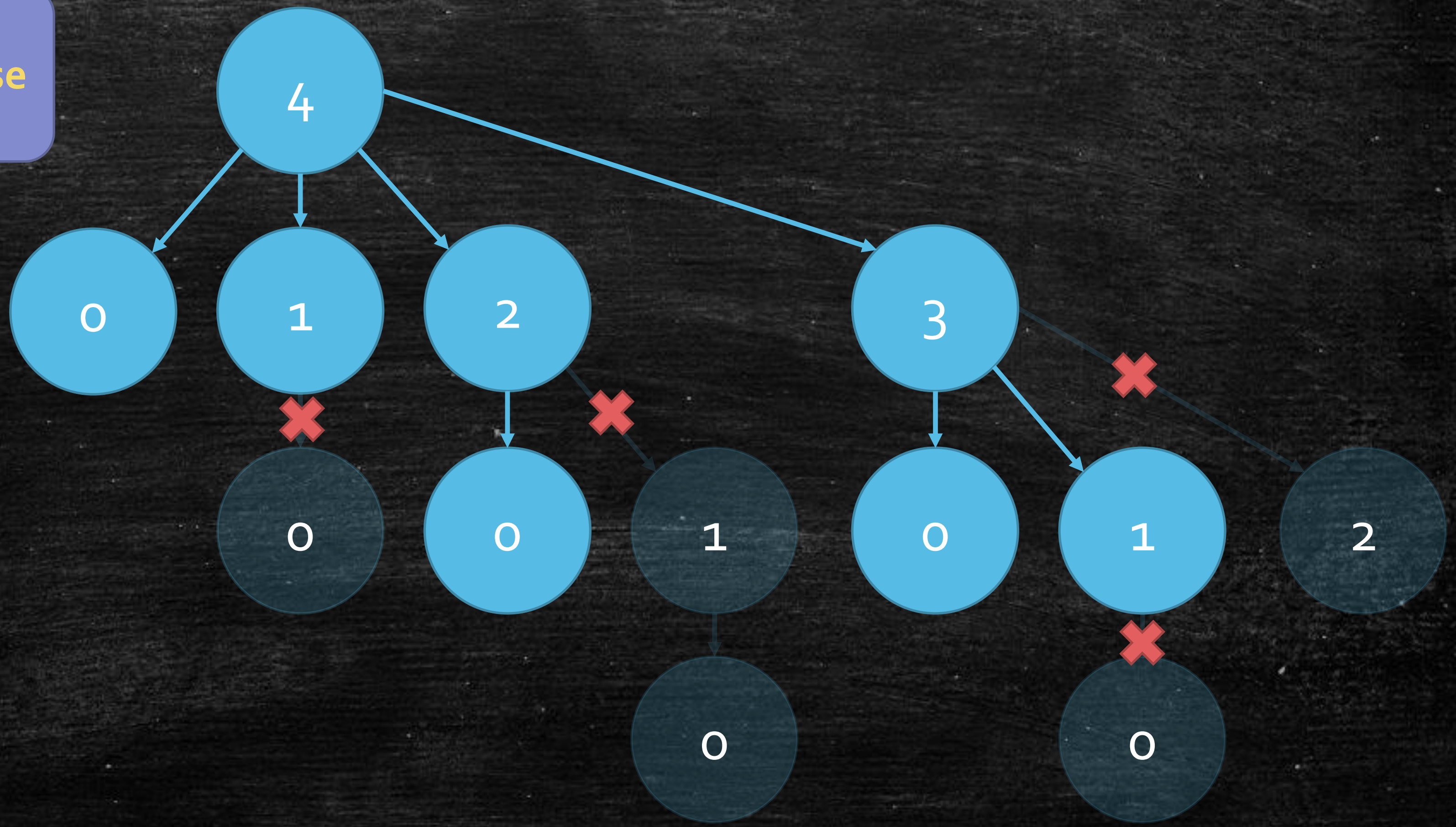
We only allow each non-root node to **lose one child.**



Root nodes does not matter.

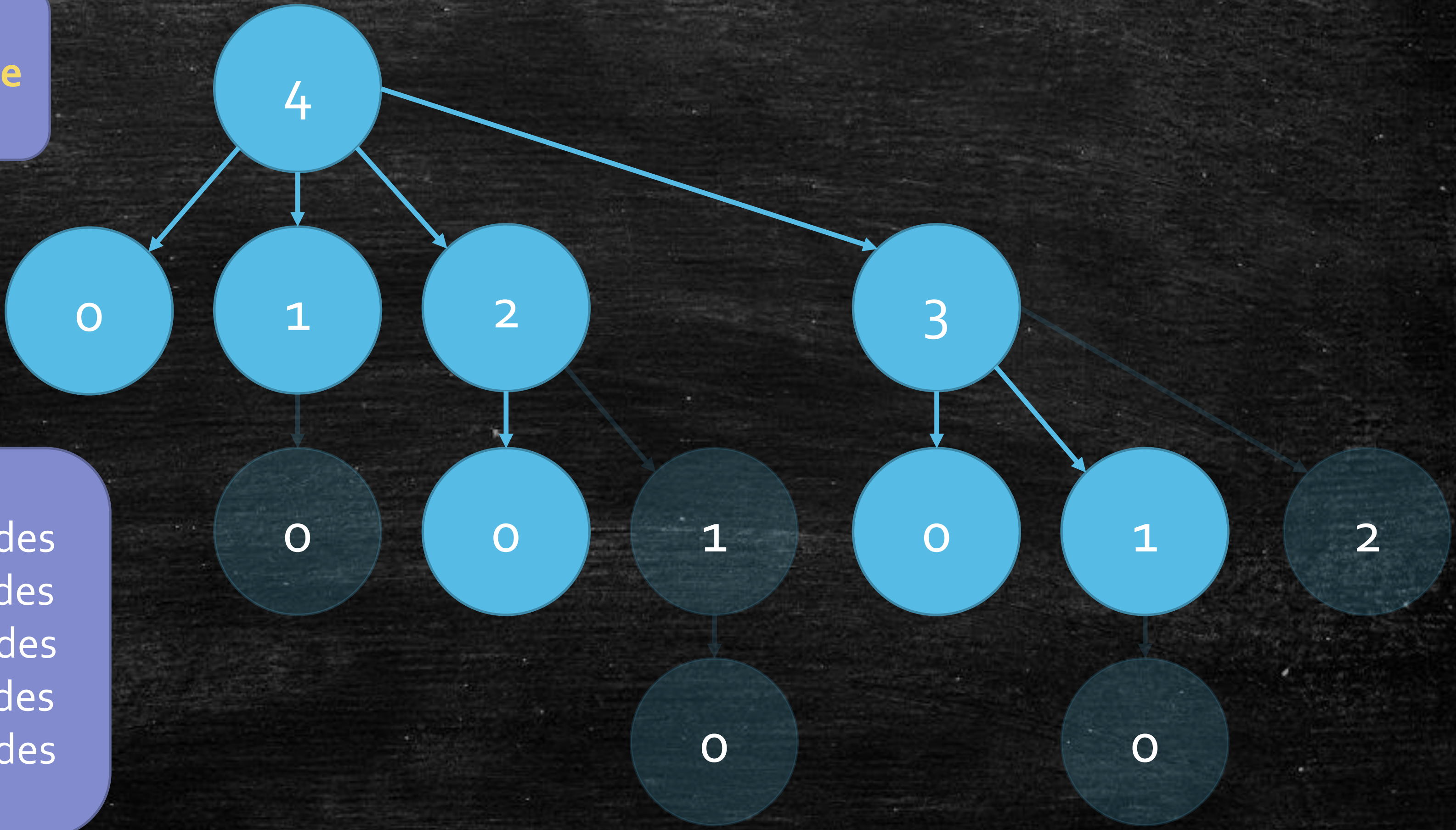
Maximum Broken tree

We only allow each non-root node to **lose one child.**



Maximum Broken tree

We only allow each non-root node to **lose one child.**



Degree 0 subtree: 1 nodes
Degree 1 subtree: 1 nodes
Degree 2 subtree: 2 nodes
Degree 3 subtree: 3 nodes
Degree 4 subtree: 5 nodes

A New Good Tree

- Each vertex in the tree with original degree k.
  - Has at least $k-1$ children, only lose one from k.
  - Children's degree 1,2,3…$k-1$, only lose one of them.
- New good definition:
  - All non-root vertex can at most lose one child.

Conclusion

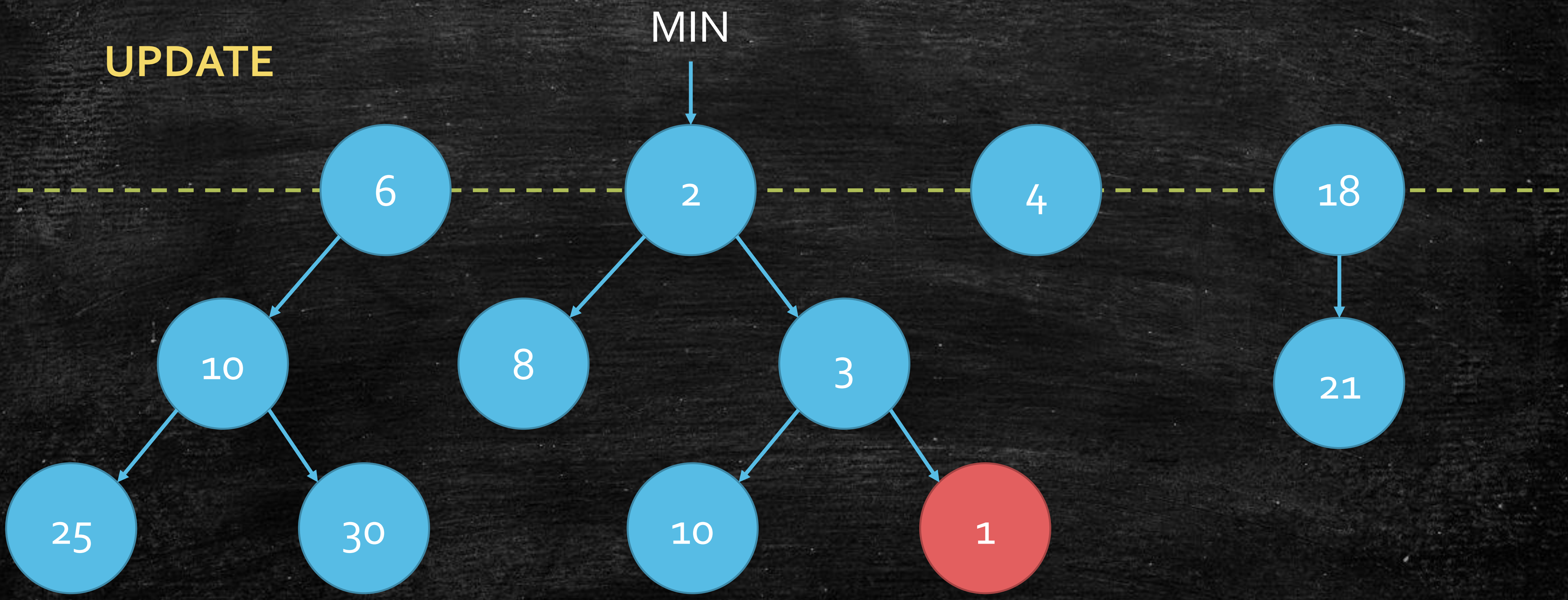
- Suppose the tree is good.
- Degree k root contains
  - A subtree of original degree 0.
  - A subtree of original degree 1.
  - ....
  - A subtree of original degree $k-1$.
- At least $F(k)$ nodes
- $F(k) = \sum_{i=1}^{k} fib[i] = O(C^k)$
- Max degree D is at most $O(\log n)$.

How to maintain this property?

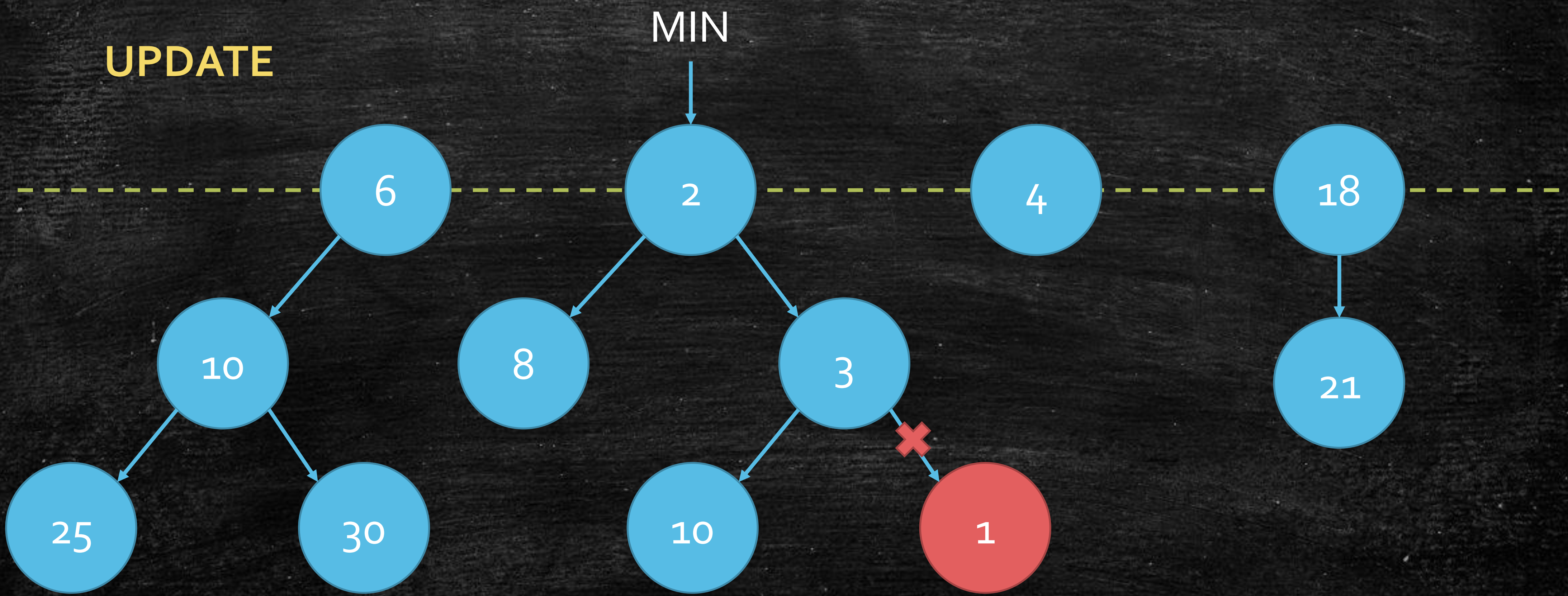
Cascading Cut

First Time Update

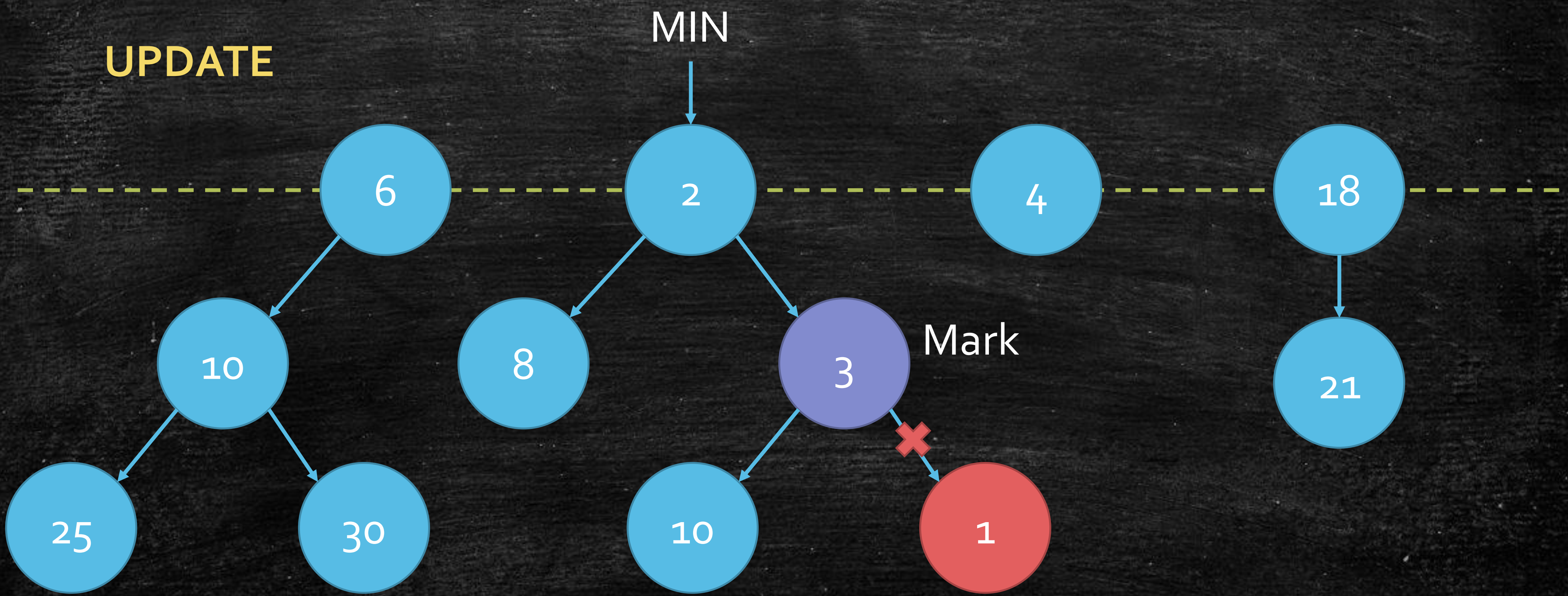
Fibonacci Heap: Cascading Cut



Fibonacci Heap: Cascading Cut



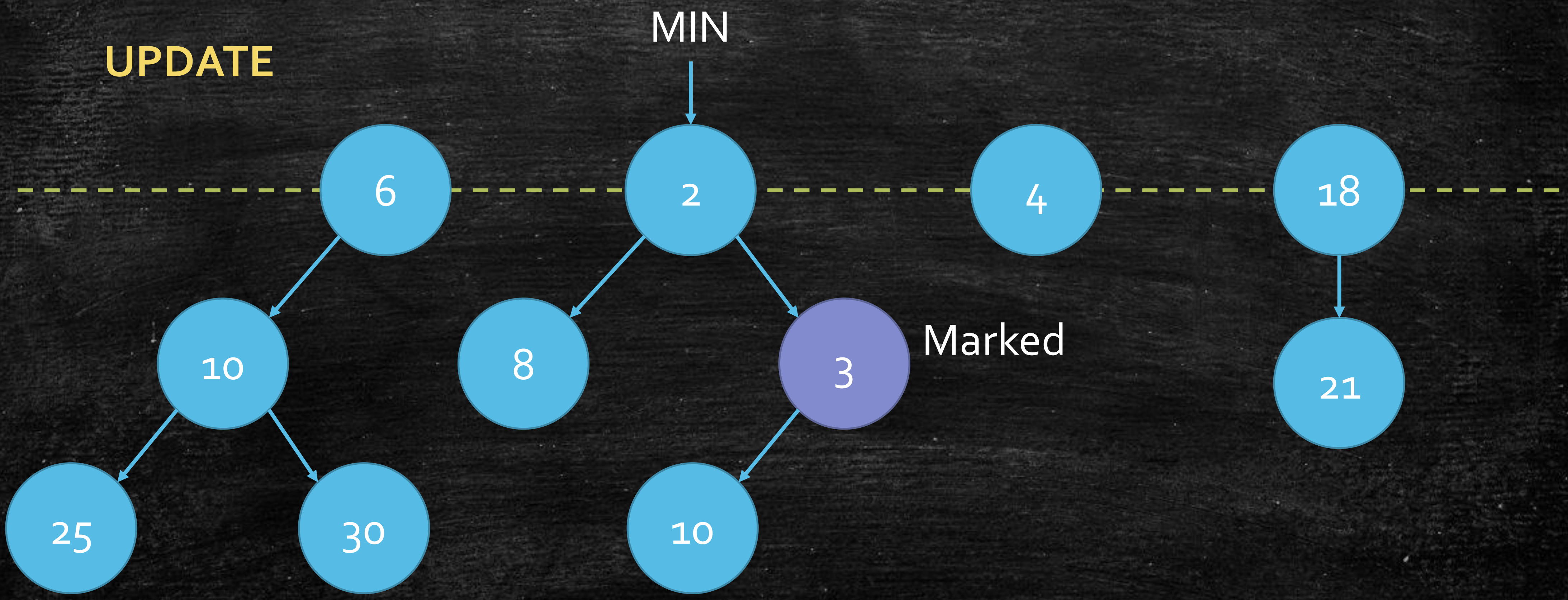
Fibonacci Heap: Cascading Cut



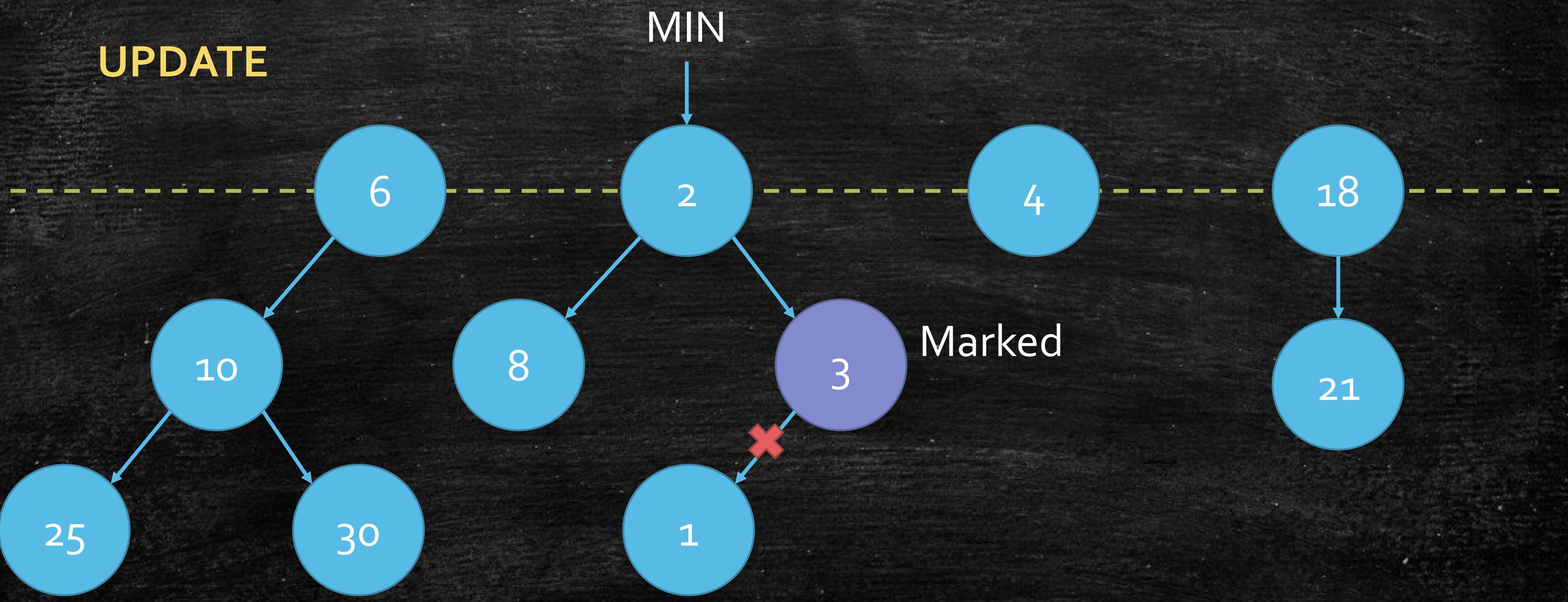
Second Time Update

Fibonacci Heap: Cascading Cut

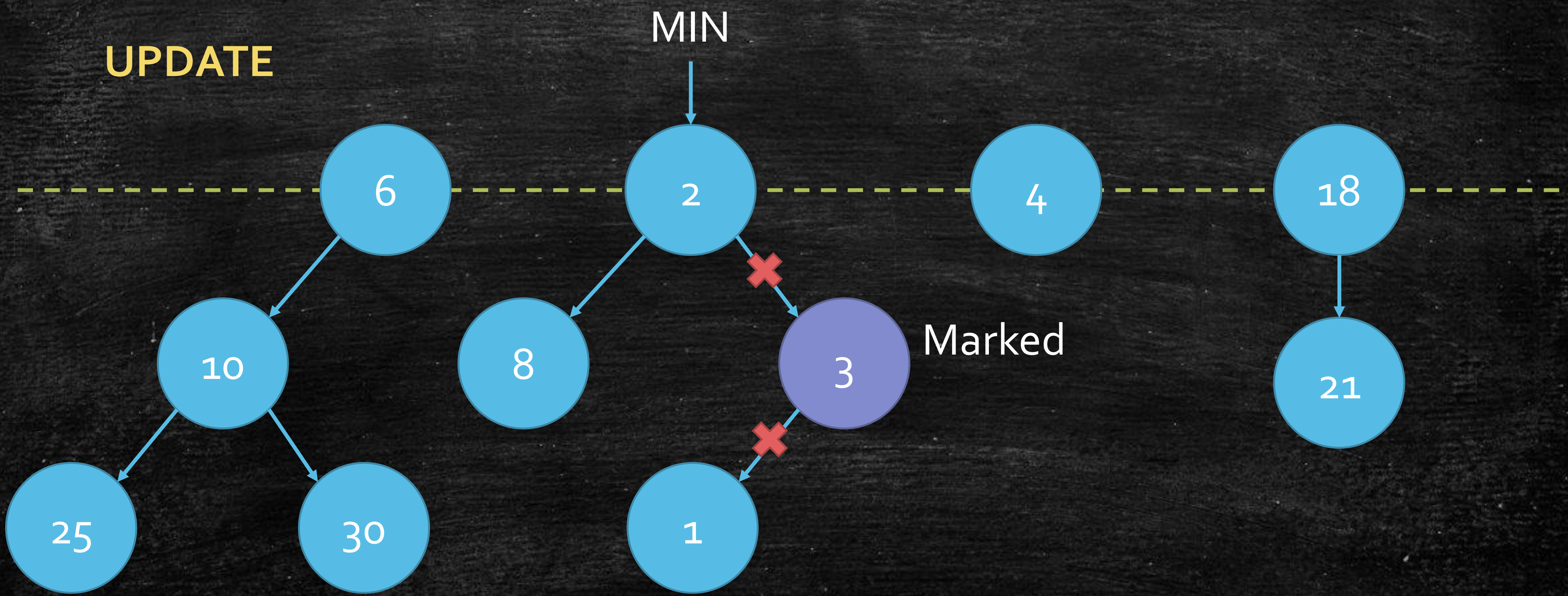
UPDATE



Fibonacci Heap: Cascading Cut



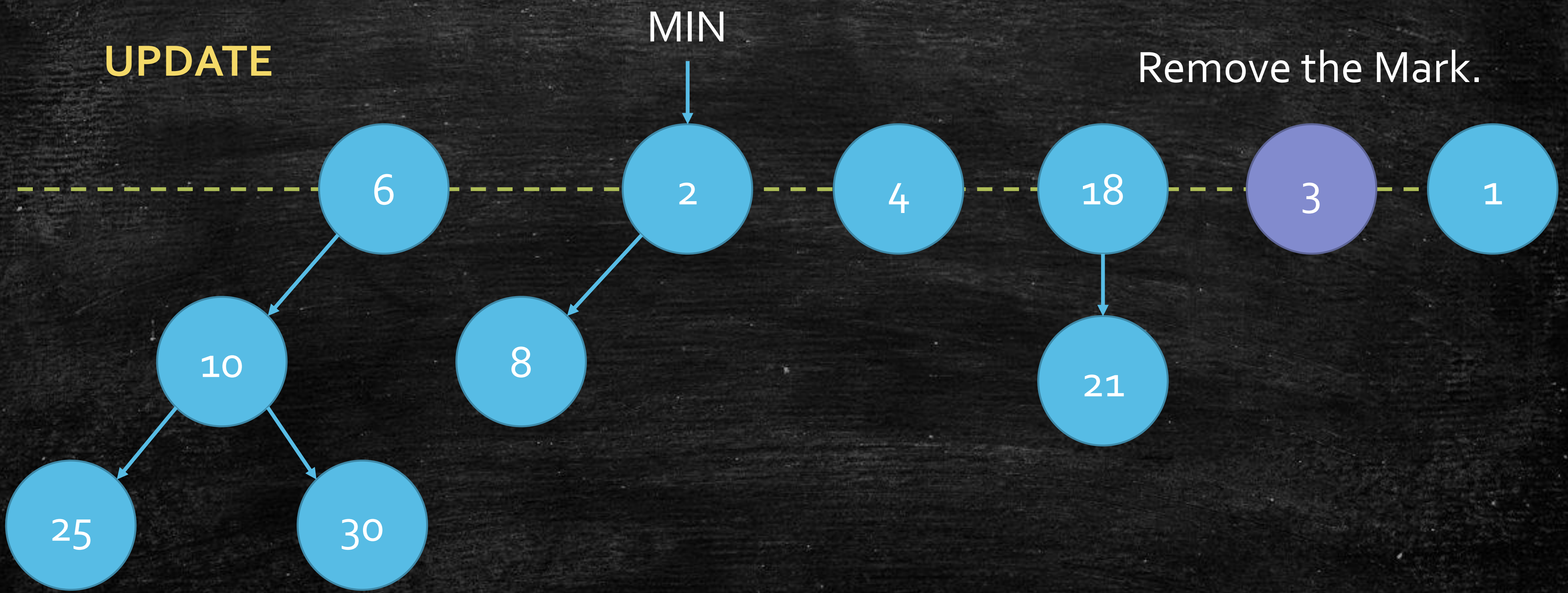
Fibonacci Heap: Cascading Cut



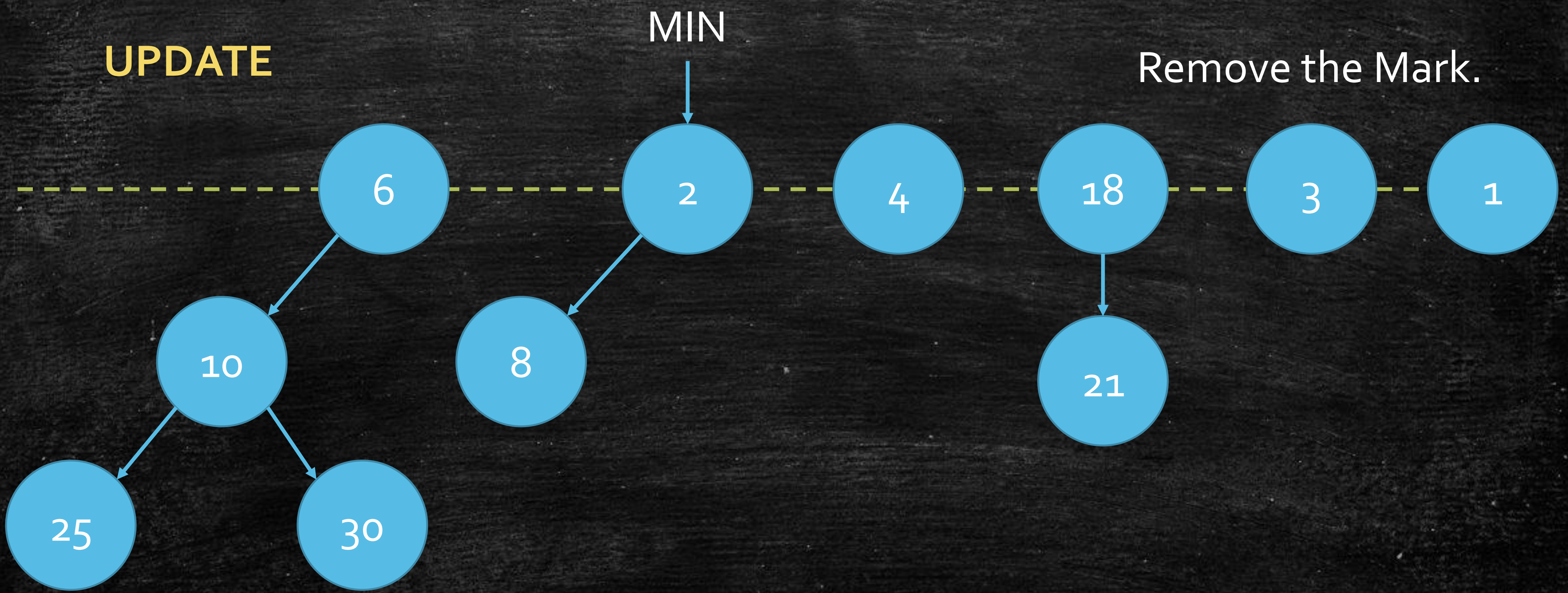
Fibonacci Heap: Cascading Cut

UPDATE

Remove the Mark.



Fibonacci Heap: Cascading Cut



Every tree keep the property!

But what is the problem now?

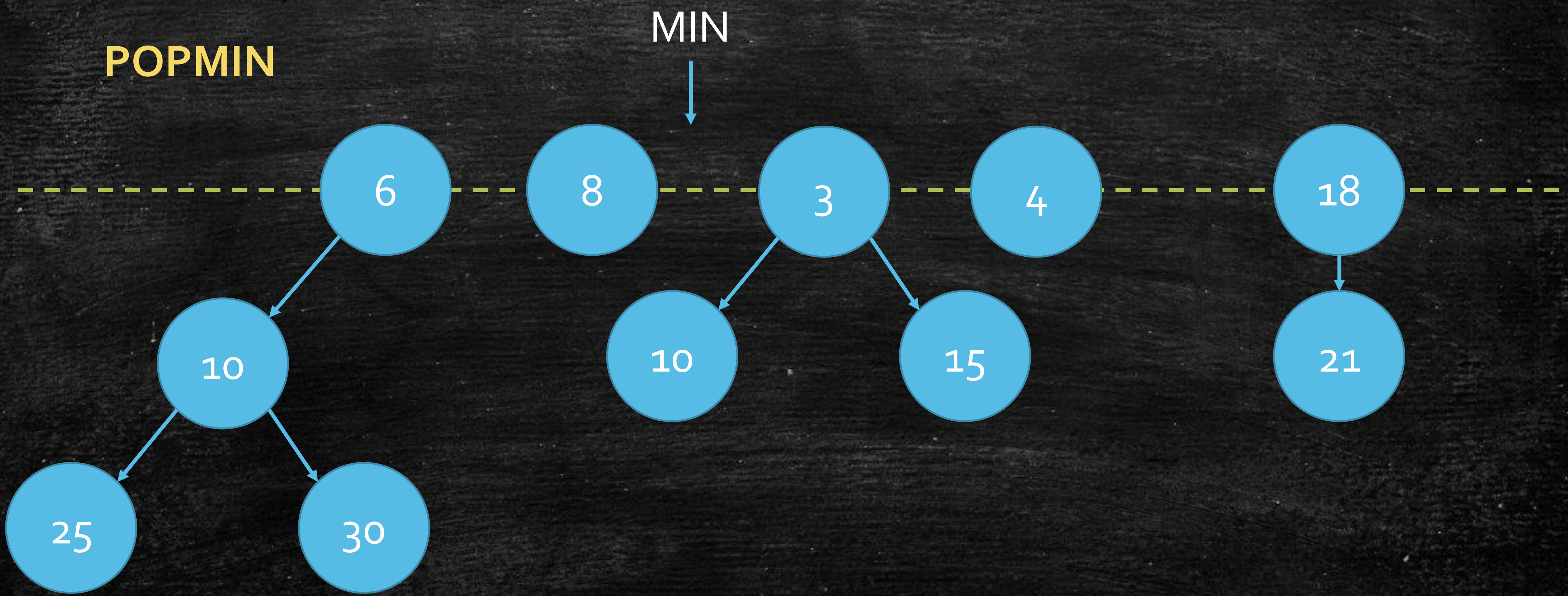
The problem

- We create so many roots on the green line!
- Yes, we have bounded D.
- However, we have not bounded the root number t.
- Cascading Cut breaks the property
  - One degree one root on the green line.
- Cost of POPMIN is still
  - $O(t + D)$
  - Maybe large

Do some good things for the future.

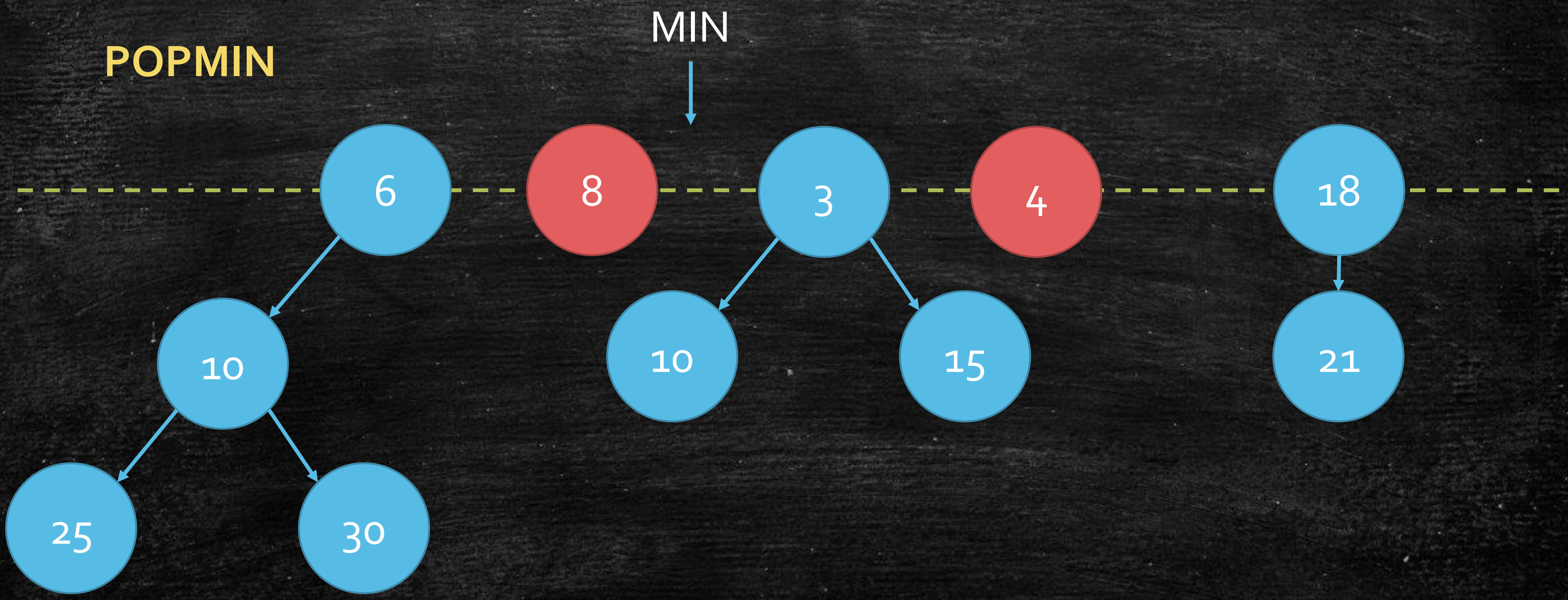
- If POPMIN is **slow**, why not do some **good** things for the future?
- Next: an $O(t)$ time **Merge subroutine**, that decrease the number of roots.
- Next time, we do not have so many roots.

Before move the MIN pointer: Merge

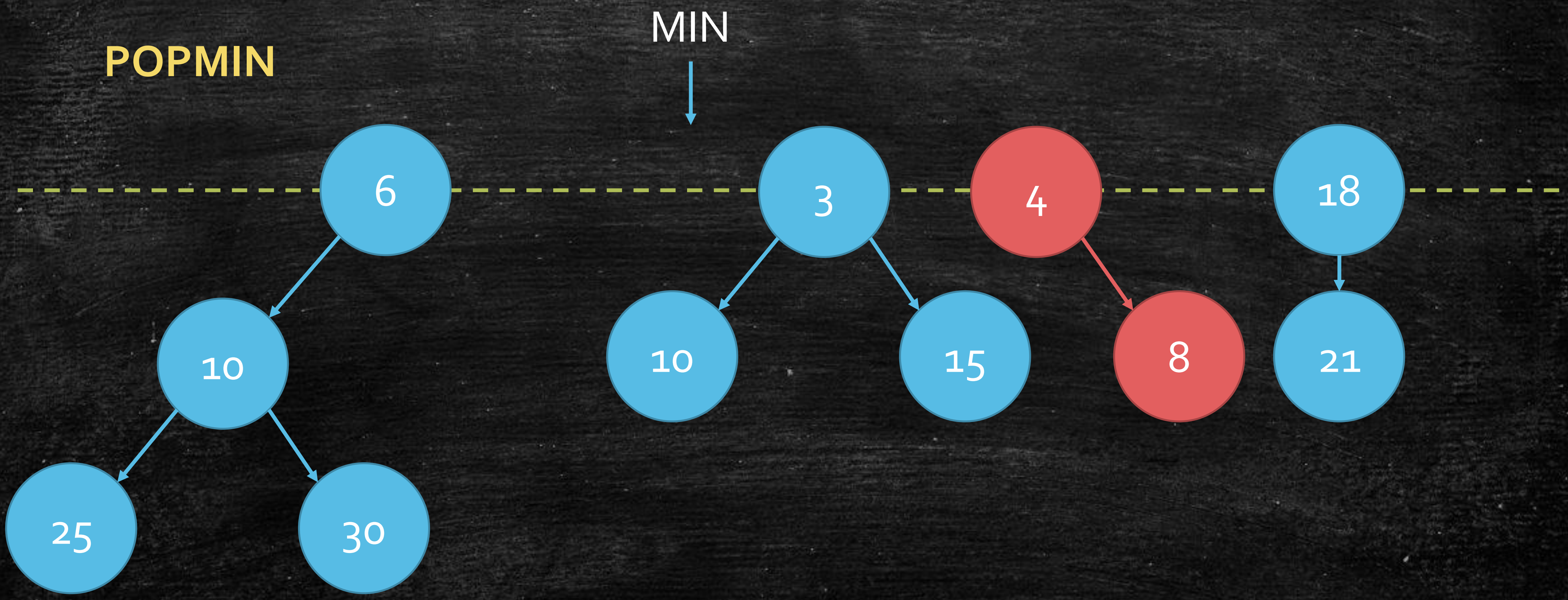


Before move the MIN pointer: Merge

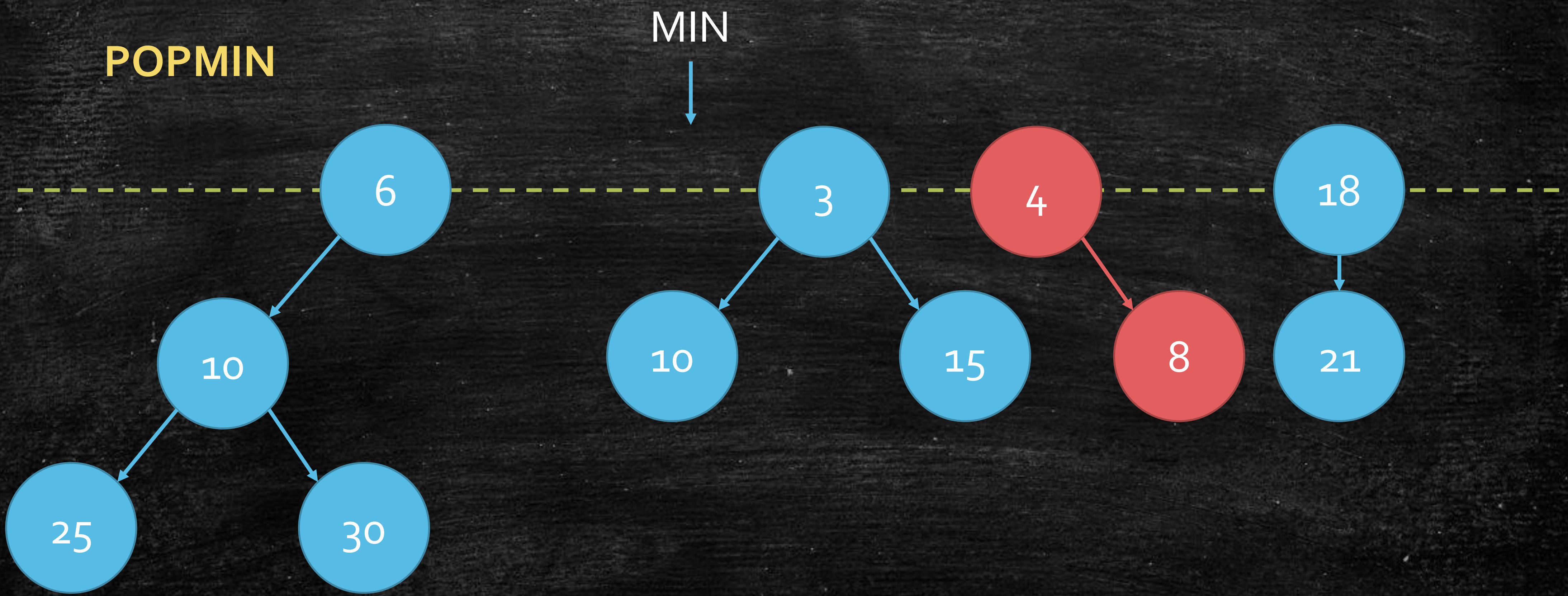
POPMIN



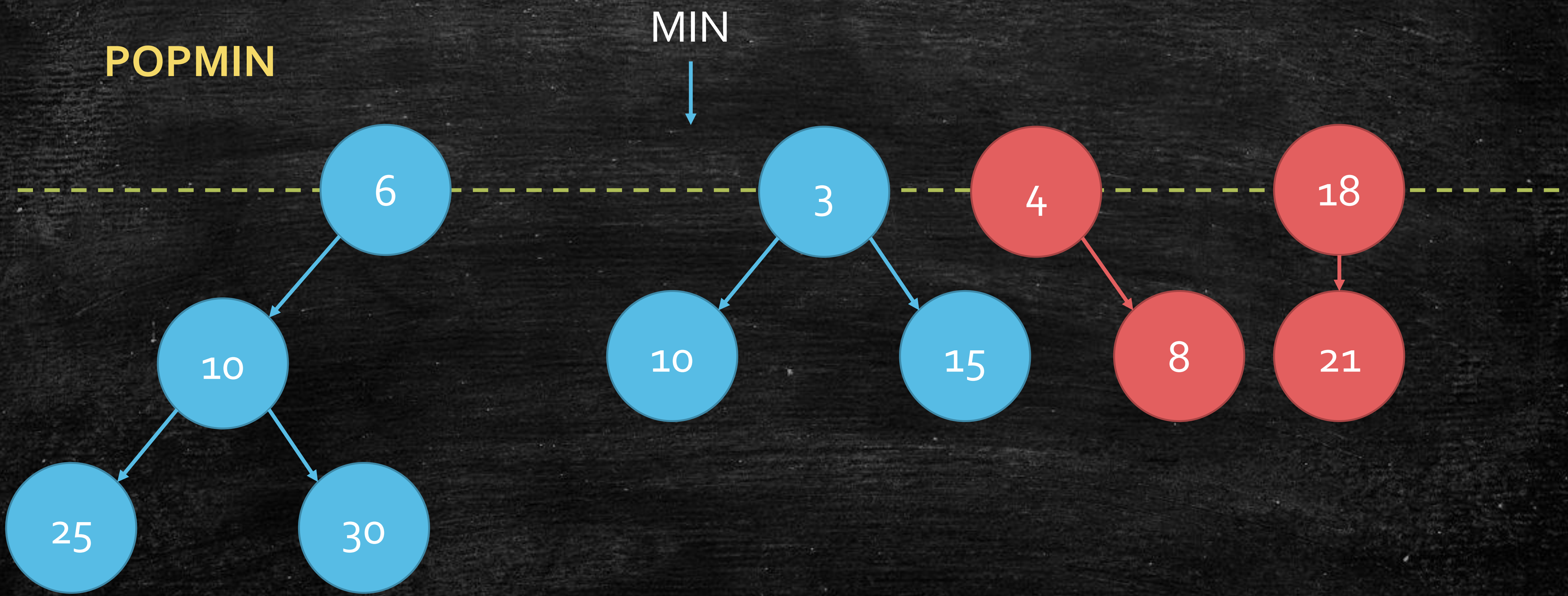
Before move the MIN pointer: Merge



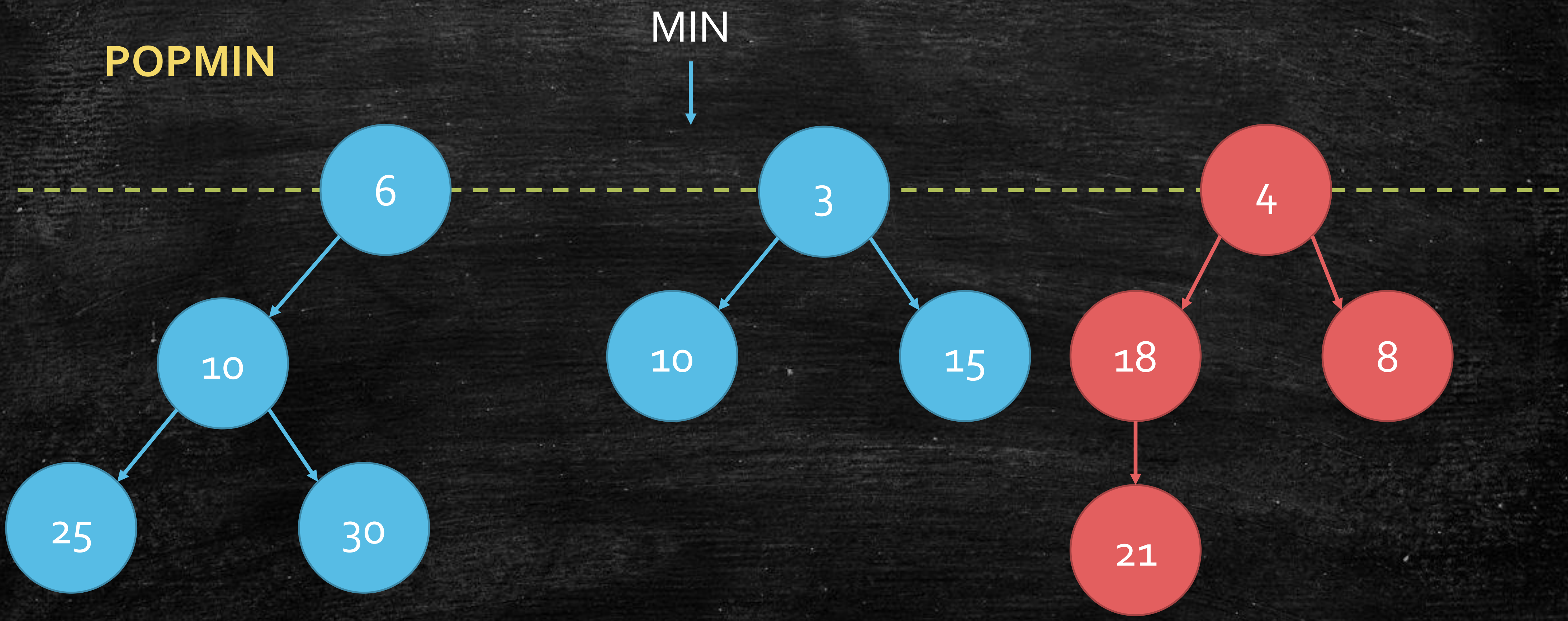
Before move the MIN pointer: Merge



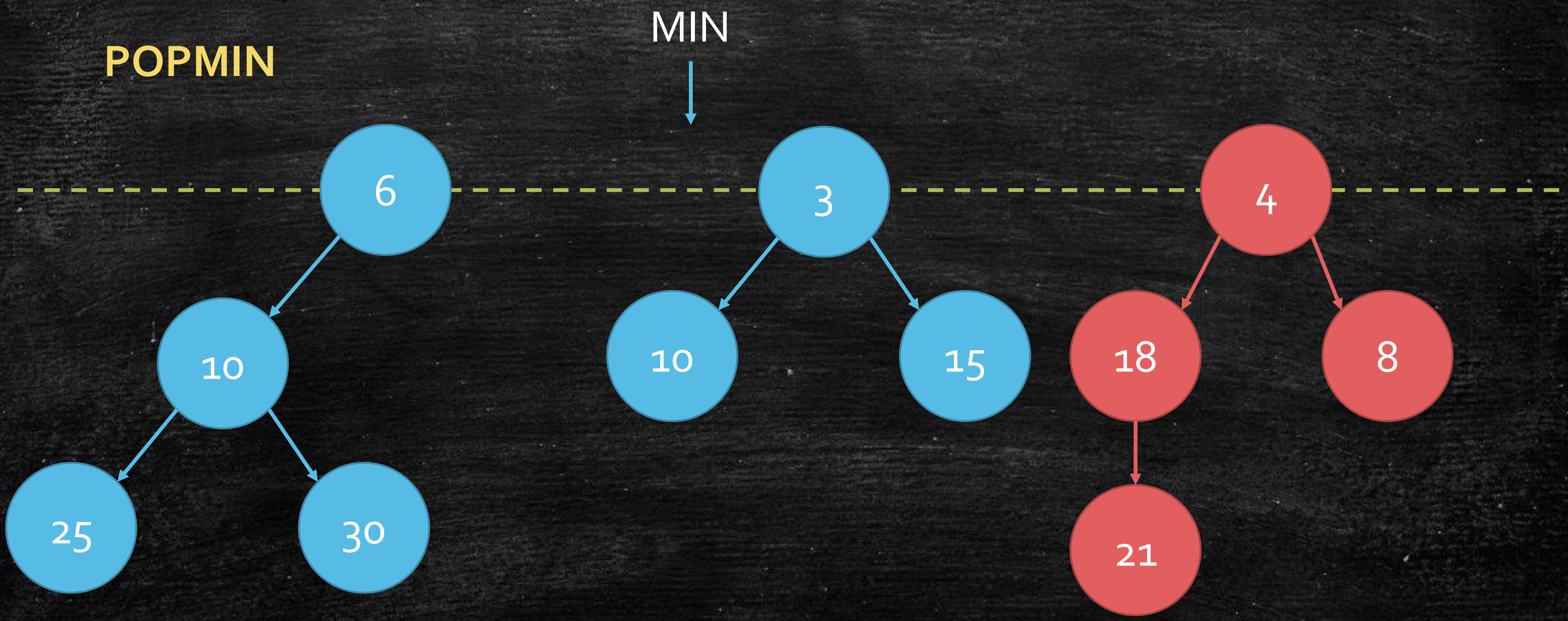
Before move the MIN pointer: Merge



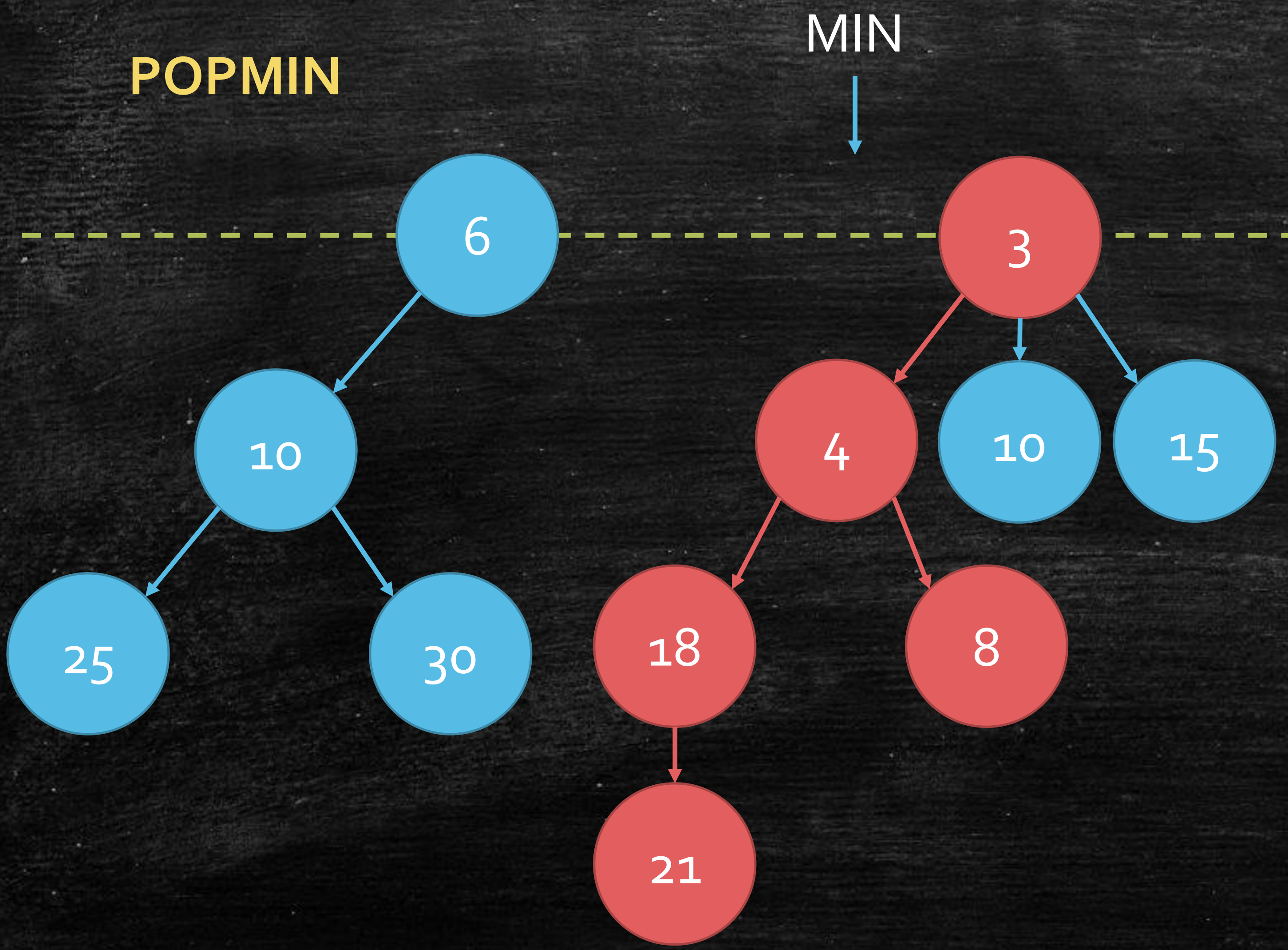
Before move the MIN pointer: Merge



Before move the MIN pointer: Merge

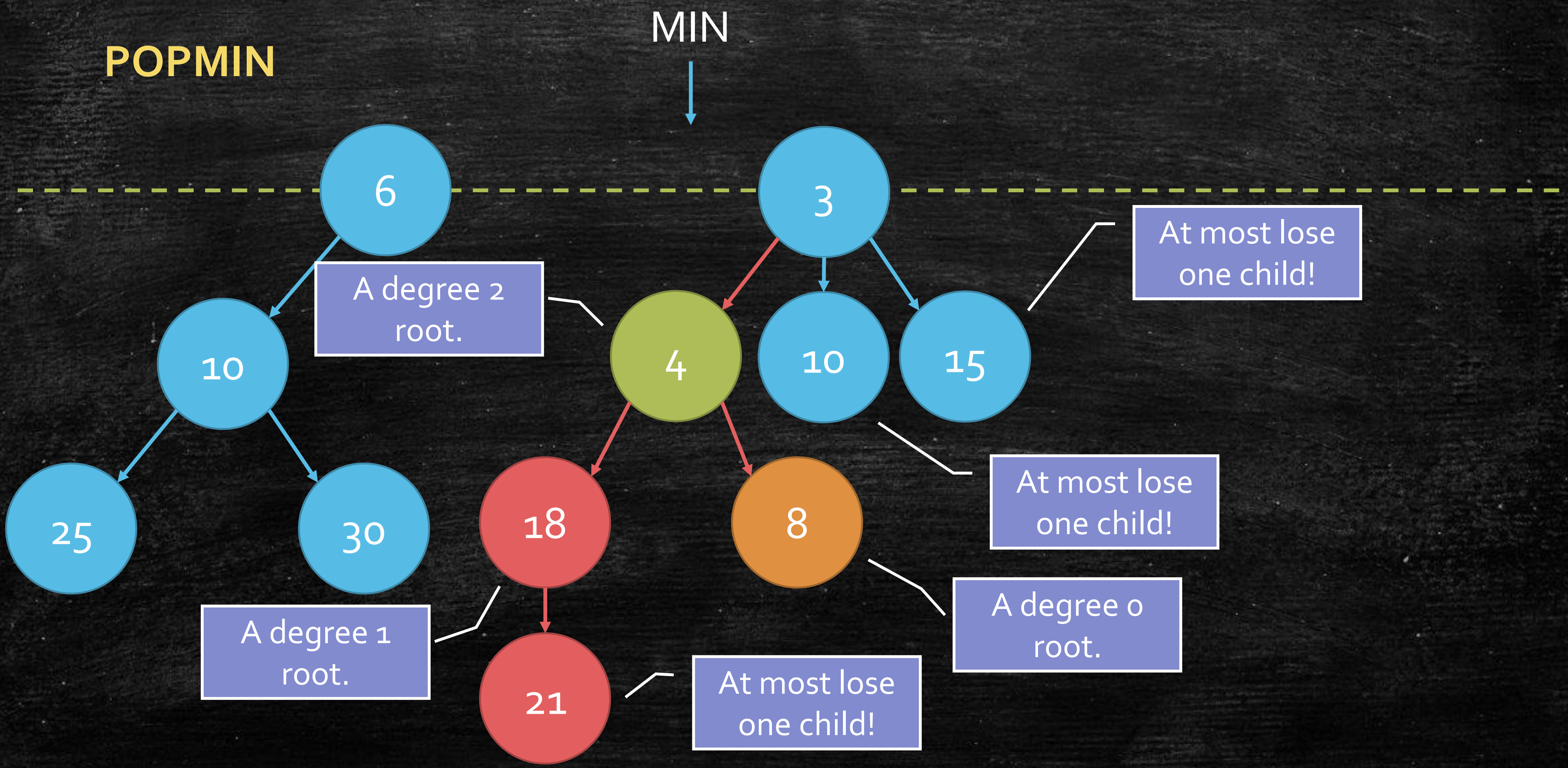


Before move the MIN pointer: Merge



Is the merged tree good?

Before move the MIN pointer: Merge



Conclusion of Merge

- After **Merge**
- We have the degree property!
  - One degree one root.
  - $D \leq \log n$

Running Time

If we focus on a specific operation.

Time Complexity: Update

- Original cut: 1
- Cascading cut: < #marked nodes it go though (m')
  - We will unmark them.
- Time: $\boldsymbol{O(m')}$

Time Complexity: POPMIN

- Totally: $\boldsymbol{O(t^{-} + D)}$
- Bad thing: t^{-} can be n!

Amortized Analysis

Recall that we have do some **good** things for the **future**!

What is amortized analysis?

- We want consider the total cost of k **arbitrary** operations.
  - $p_1, p_2, p_3 \dots .$
- We do not mean k ~~**random**~~ operations.
- $C(p_i)$: The real cost of Operation p_i.
- Total cost $C(p_1) + C(p_2) + C(p_3) + \cdots + C(p_k)$
- Assume we have two type P_1, P_2.
- $\hat{C}(P)$: Amortized cost of a type P cost.
- $C(p_1) + C(p_2) + C(p_3) + \cdots + C(p_k) \leq k_1\hat{C}(P_1) + k_2\hat{C}(P_2)$

Amortized Analysis: Potential Function

- Some operation may have **small C** make **later operation** bad.
- Define Φ to represent the state of the problem, $\Phi_0 = 0$.
- Let it pay something for the future, so we let $\hat{C} = C + \delta \cdot \Delta\Phi$.
- Φ is a function to evaluate current state.
- $\sum \hat{C} = \sum C + \delta \cdot \sum \Delta\Phi = \sum C + \delta \cdot \Phi$
- $\sum \hat{C} \geq \sum C$ if $\Phi \geq 0$.

A chosen constant.

A chosen constant.

Consider the example of Stack.

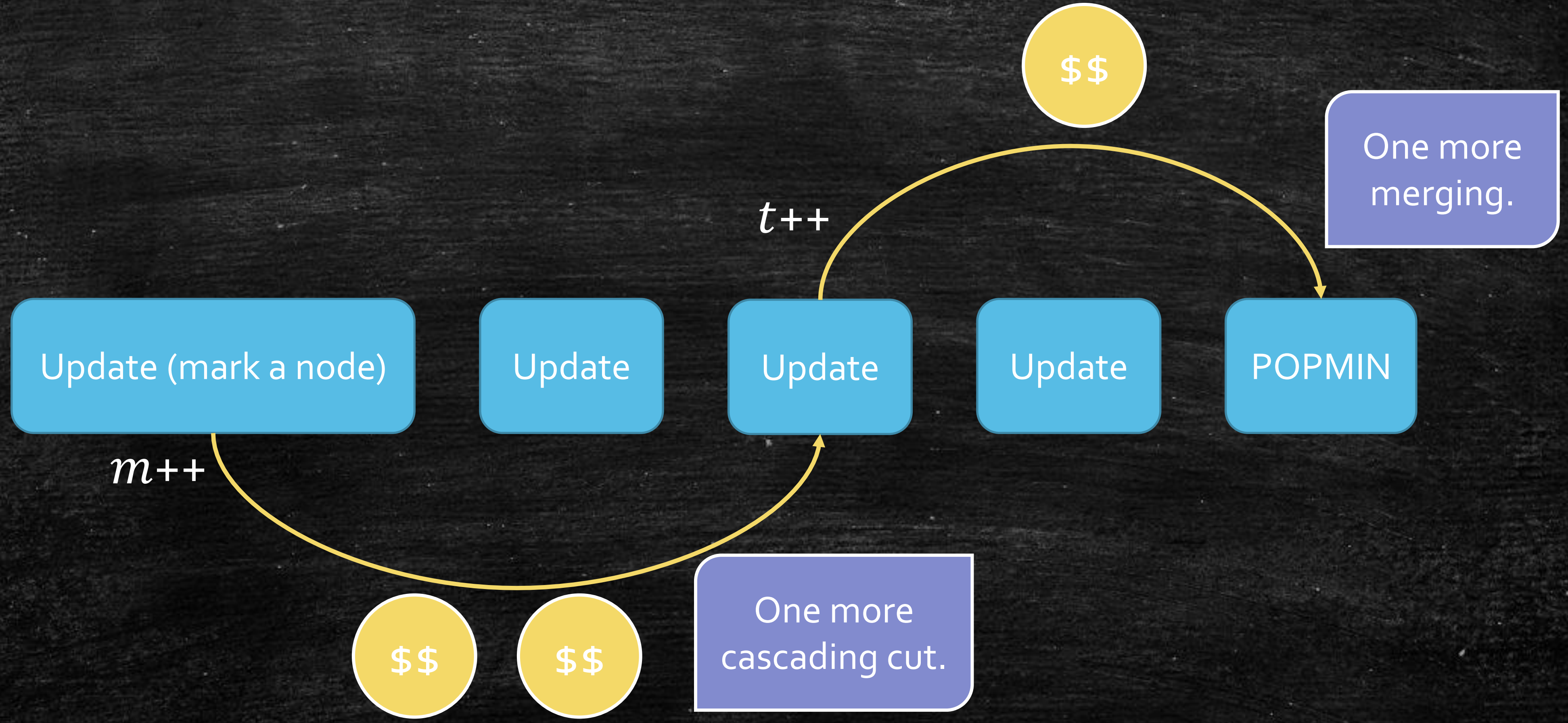
Amortized Analysis: Stack

- Operations
  - Pop all elements one by one.
  - Push one element.
- Potential Function
  - $\Phi = \#\text{elements}$
- **Push**
  - $C = \boldsymbol{O(1)}$
  - $\hat{C} = O(1) + \delta \cdot 1 = \boldsymbol{O(1)}$
- **Pop**
  - $C = \boldsymbol{O(k)}$
  - $\hat{C} = O(k) + \delta \cdot (-k) = \boldsymbol{O(1)}$

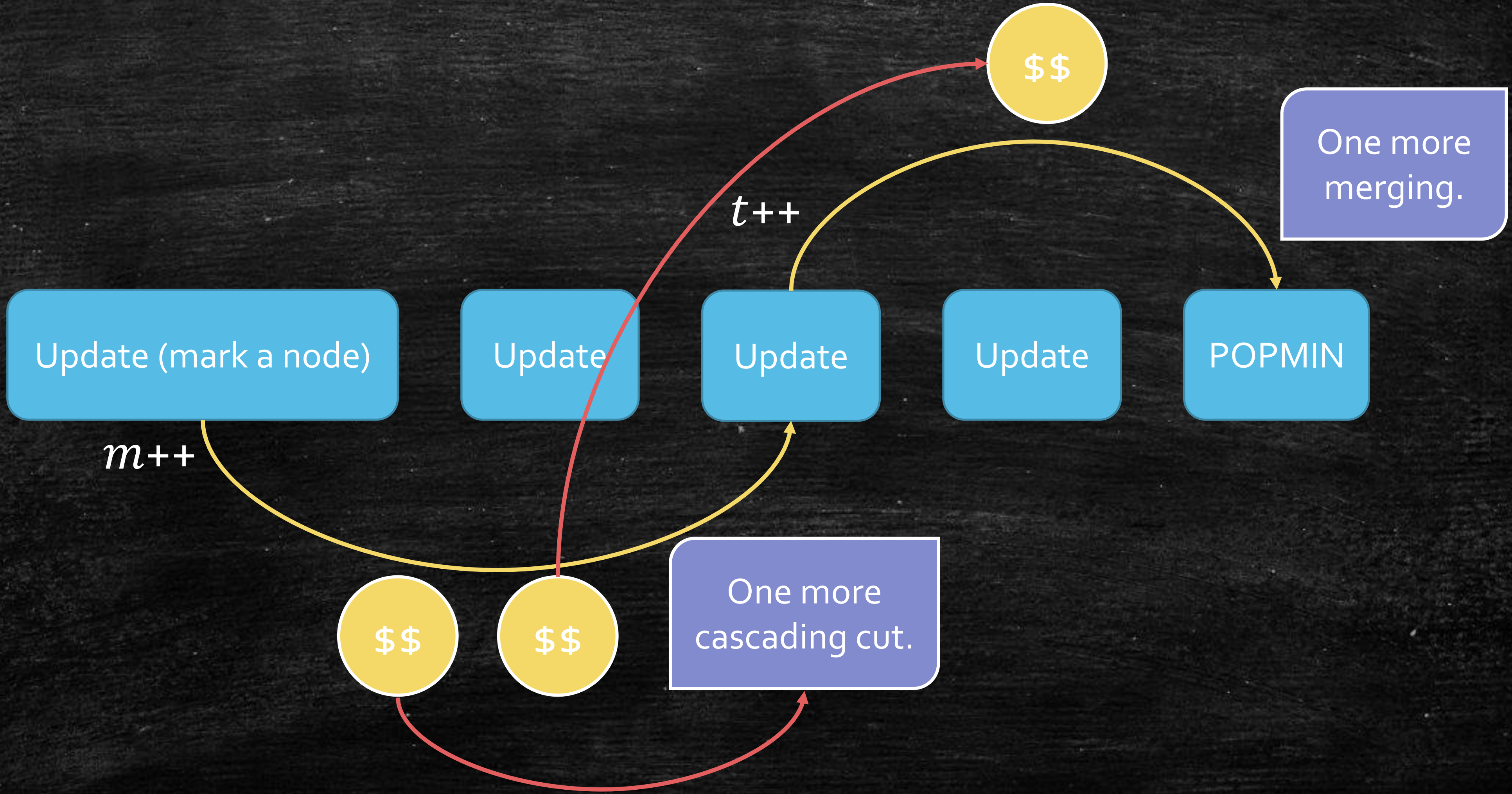
Amortized Analysis: Fibonacci Heap

- **Update:** $O(m')$
- **Pop Min:** $O(t^- + 2D)$
- What is bad?
  - #marked nodes
  - #roots
- Potential Function: $\boldsymbol{\Phi = t + 2m}$
- Why we need $2m$?
- m has two bad things
  - One more cascading cut!
  - One potential root at merging!

How we pay for the future



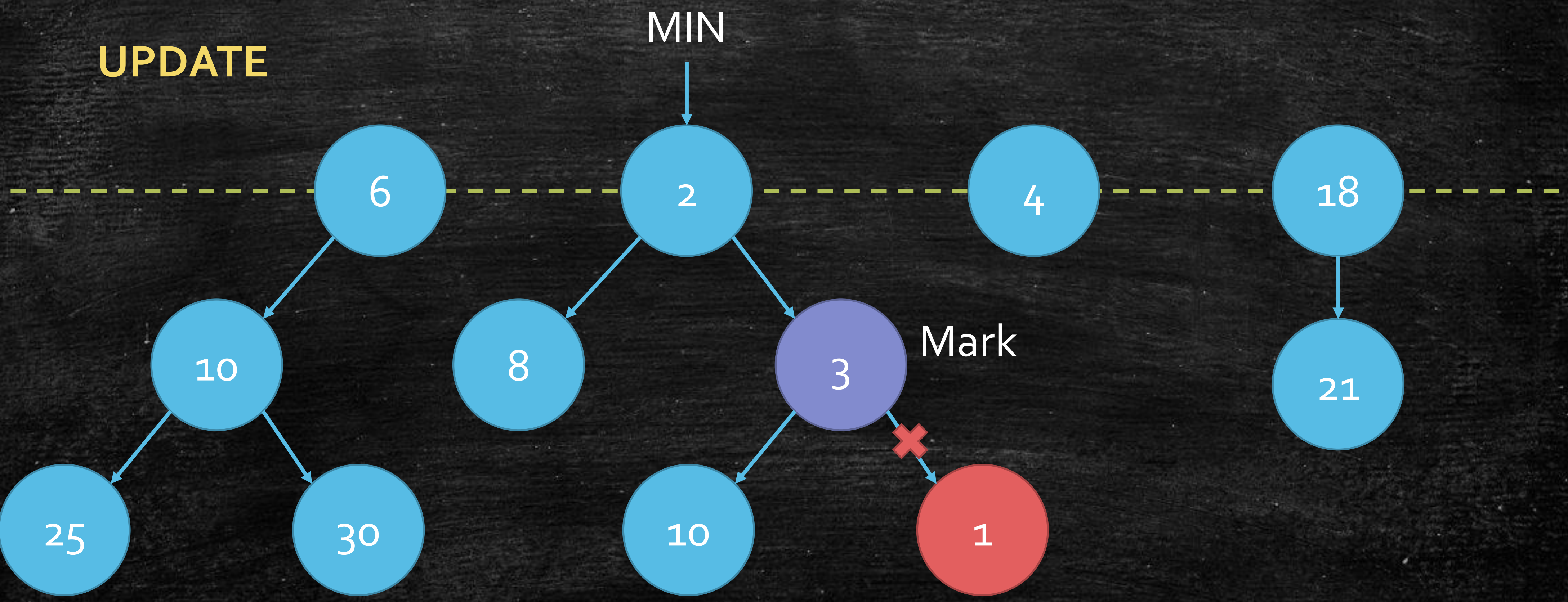
How we pay for the future



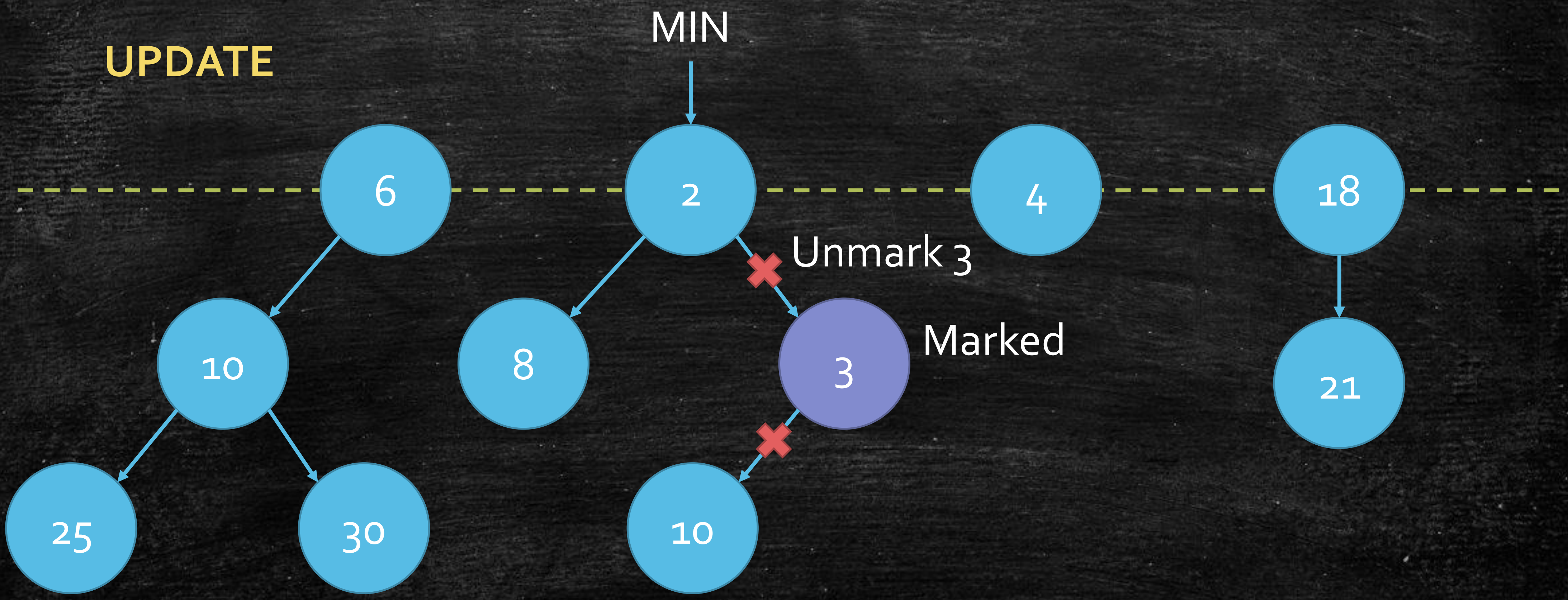
Let us analyze the amortized cost!

Fibonacci Heap: Cascading Cut

UPDATE



Fibonacci Heap: Cascading Cut



Amortized Analysis: Fibonacci Heap

- **Update:** $O(m')$
- **Pop Min:** $O(t^{-} + D)$
- Potential Function: $\boldsymbol{\Phi = t + 2m}$
- **Update**
  - $\#CC$ cascading cuts, remove $\#CC$ mark, add $\#CC$ roots.
  - one basic cut, one more mark, add one root.
  - $C = O(\#CC + 1)$
  - $\Delta t = \#CC + 1$
  - $\Delta m = -\#CC + 1$
  - $\hat{C} = O(\#\mathrm{CC} + 1) + \delta \cdot \Delta\Phi = \mathrm{O}(\#\mathrm{CC} + 1) + \delta \cdot (-\#\mathrm{CC} + 3) = \boldsymbol{O(1)}$

Amortized Analysis: Fibonacci Heap

- **Update:** $O(m')$
- **Pop Min:** $O(t^- + D)$
- Potential Function: $\boldsymbol{\Phi = t + 2m}$
- **Update**
  - $\hat{C} = \boldsymbol{O(1)}$
- **Pop Min**
  - $C = O(t^- + D)$
  - $\hat{C} = O(t^- + D) + \delta \cdot \Delta t \leq O(t^- + 2D) + \delta \cdot (D - t^-) = \boldsymbol{O(D)} = \boldsymbol{O(\log n)}$
  - Recall
    - We have $t^- + D$ roots before merging, and at most D roots after merging.

Conclusion

Dijkstra + Fibonacci Heap = $O(|E| + |V| \log|V|)$

Today's goal

- Learn **Dijkstra**
  - Why it is **correct?**
  - How to **design** algorithm if you are Dijkstra?
  - How to use **Heap** to improve Dijkstra?
  - How to use **Data Structures** to improve **Algorithms**?
- Learn **Amortized Analysis**