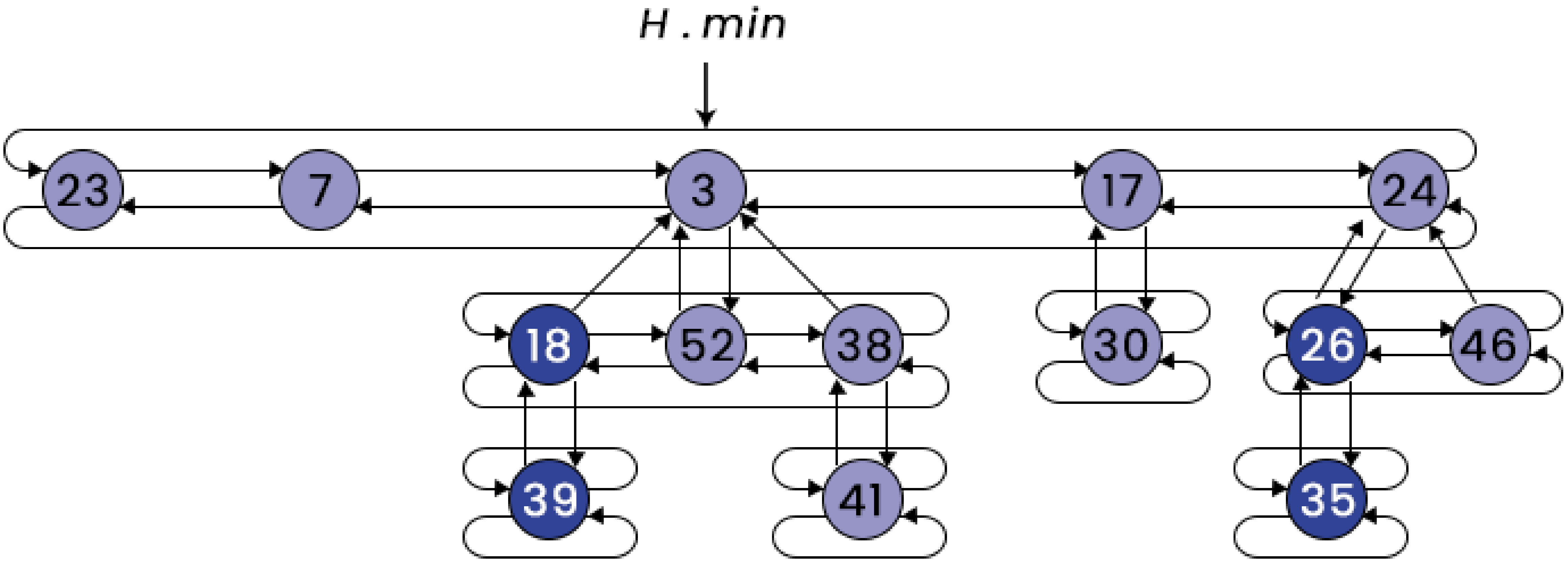



# Fibonacci Heap

By: Michael Wood, Dan Ruppin, Ben Weber

# Computational Problem

- Data Structure designed to optimize operations on Priority Queues
- Problems it was designed to solve:
  - Dijkstra’s Algorithm
  - Prim’s Algorithm

# Background Information

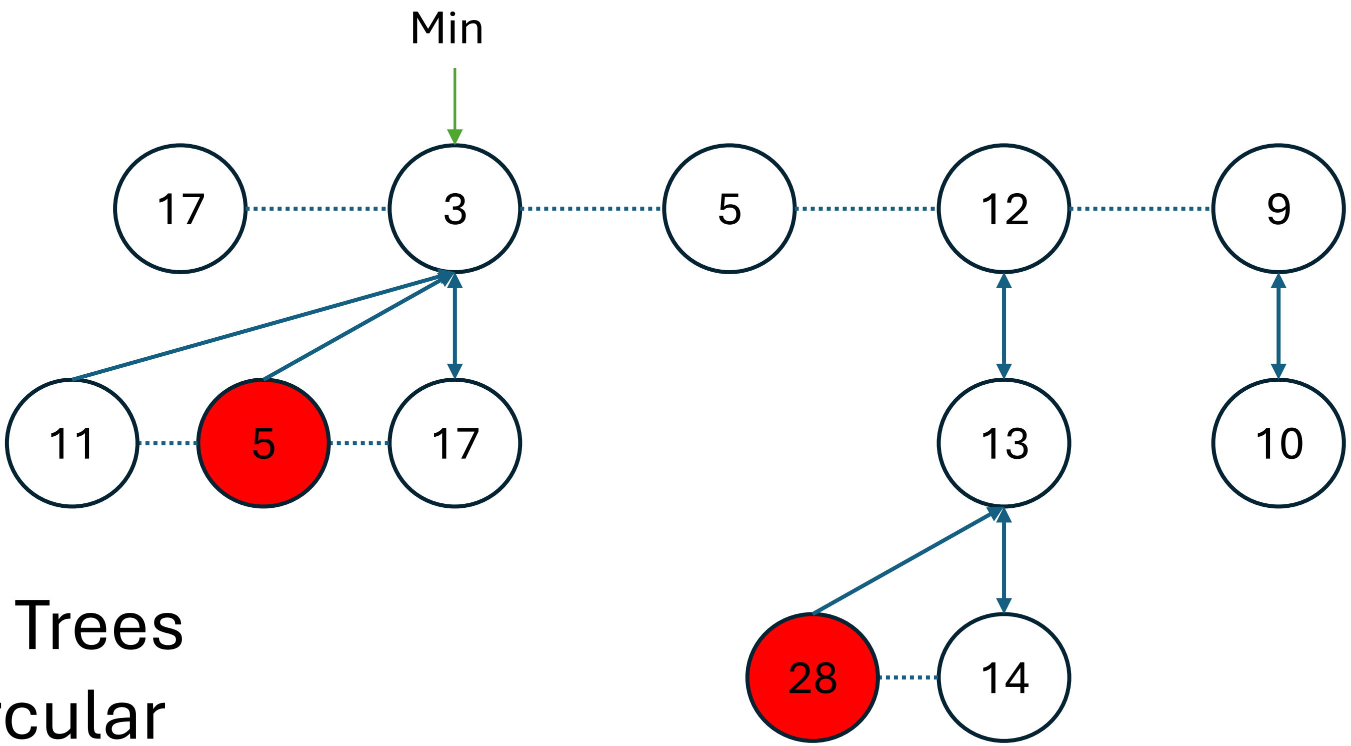
Created by Micheal Freedman & Robert Tarjan in 1984

| Function | Fibonacci Heap | Binomial Heap | Binary Heap |
|---|---|---|---|
| Insert | O(1) | O(log n) | O(log n) |
| Find-min | O(1) | O(log n) | O(1) |
| Extract-Min | O(log n) | O(log n) | O(log n) |
| Decrease-key | O(1) | O(log n) | O(log n) |
| Union | O(1) | O(log n) | O(n) |

*Amortized except for Binary heap

# Example Fibonacci Heap

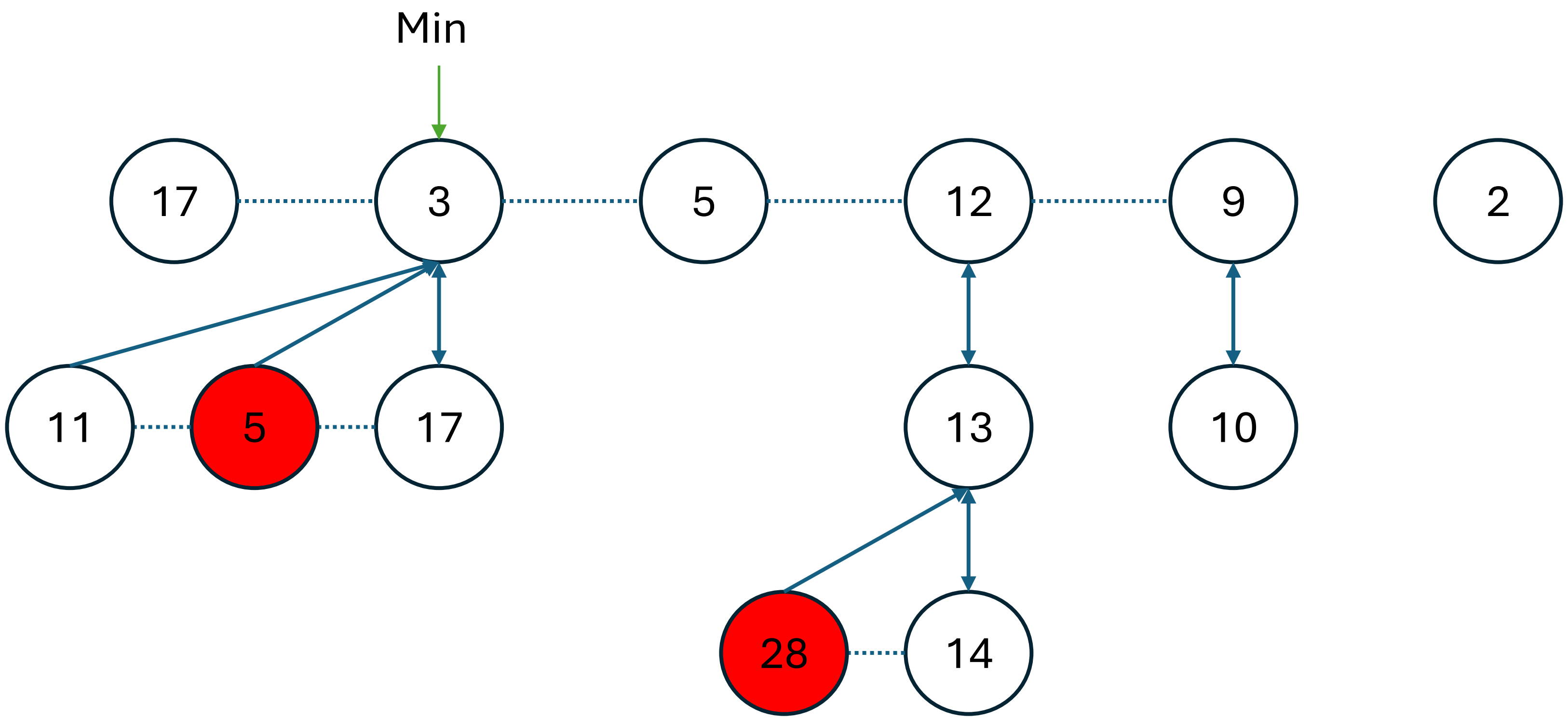
Key Operations:
- Insert
- Extract-Min
- Decrease-Key
- Union




Collection of Heap Ordered Trees
- Nodes structured in a circular doubly linked list
- Lazy Consolidation

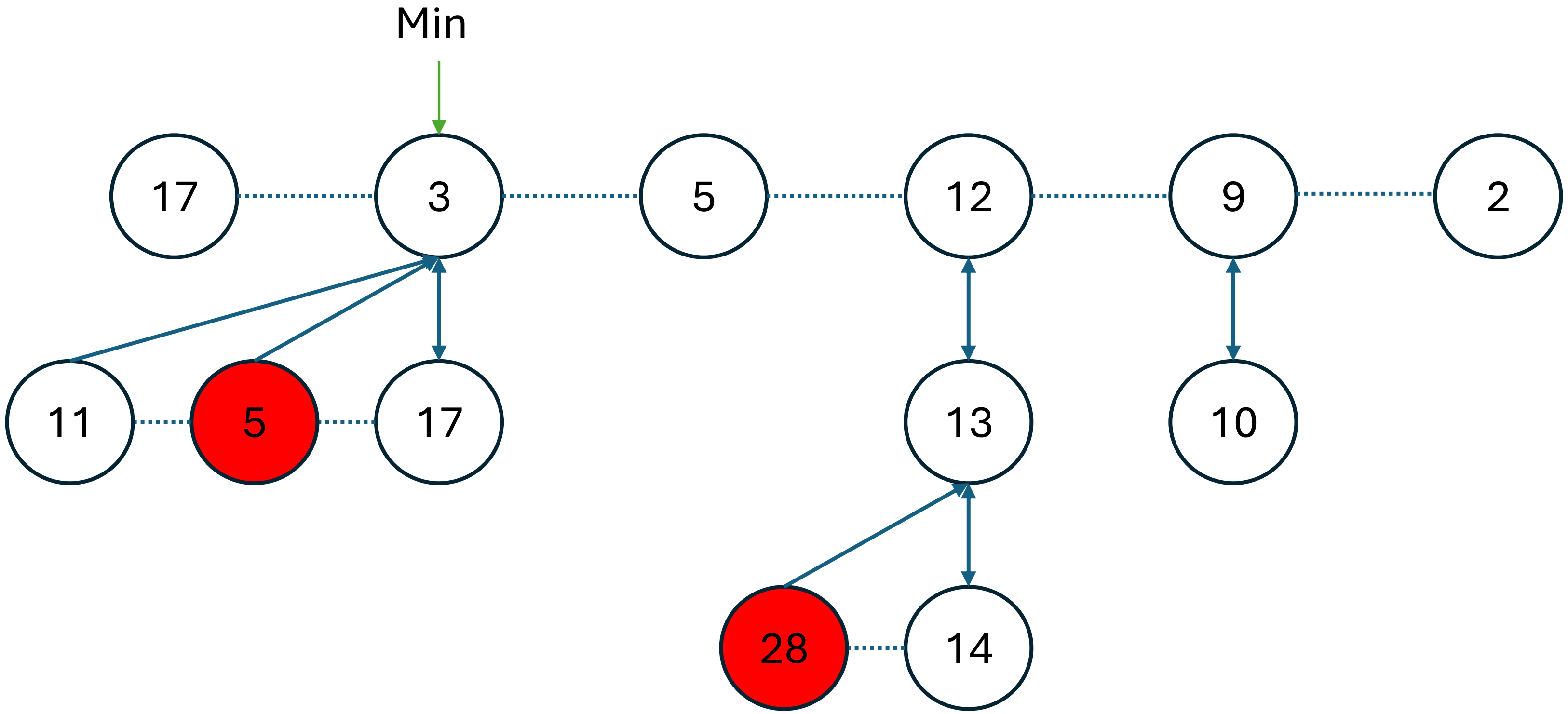
# Insert : $O(1)$

- Simply add node to the root list
- Update min if needed



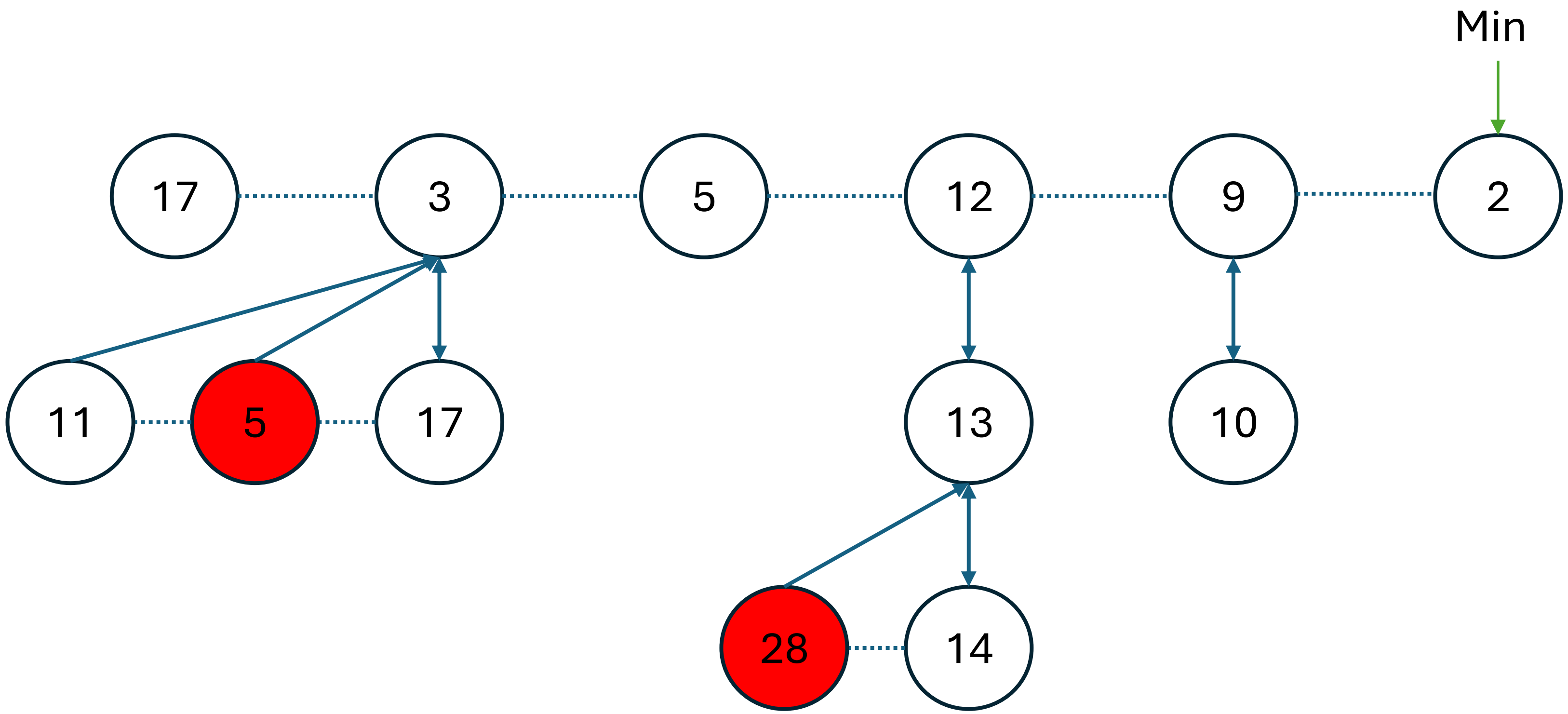
# Insert : $O(1)$

- Simply add node to the root list
- Update min if needed



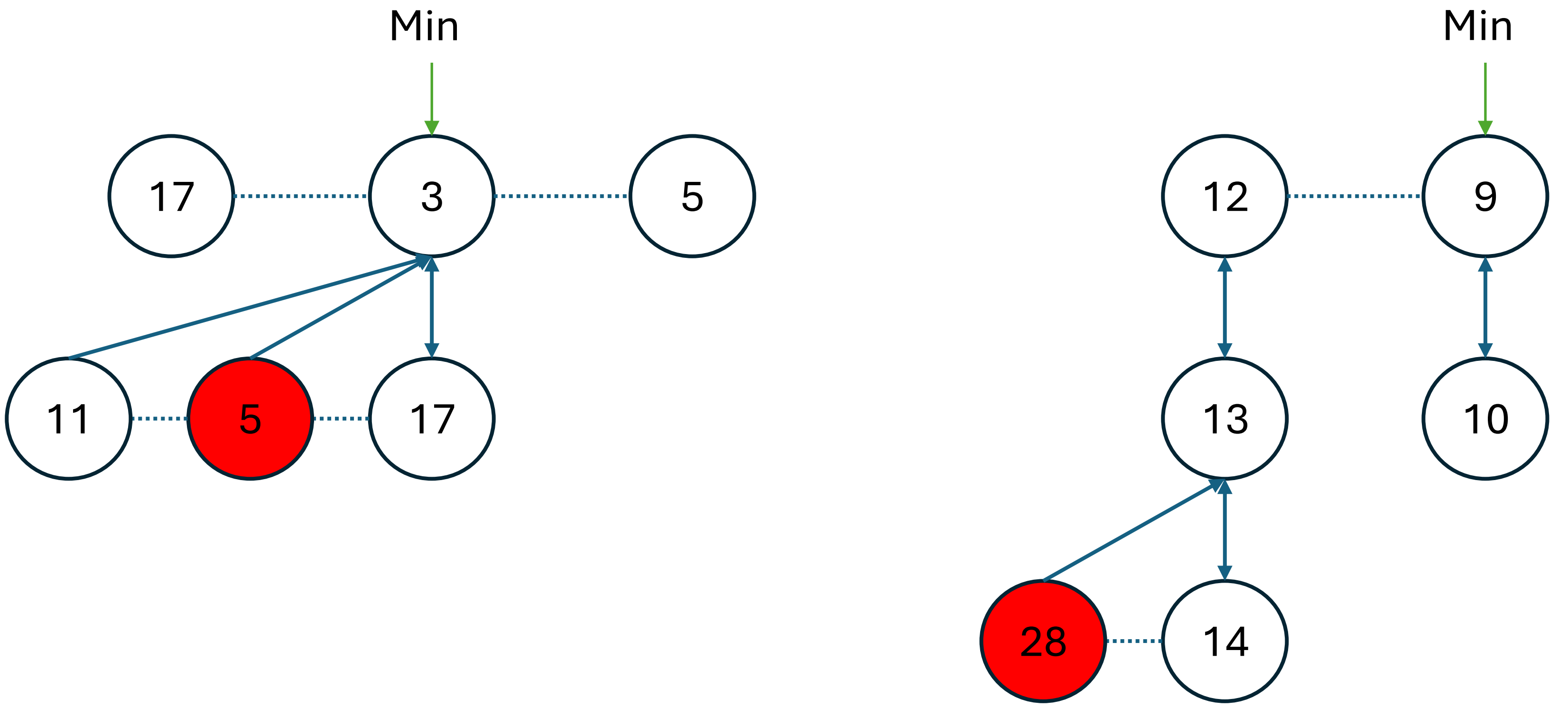
# Insert : $O(1)$

- Simply add node to the root list
- Update min if needed



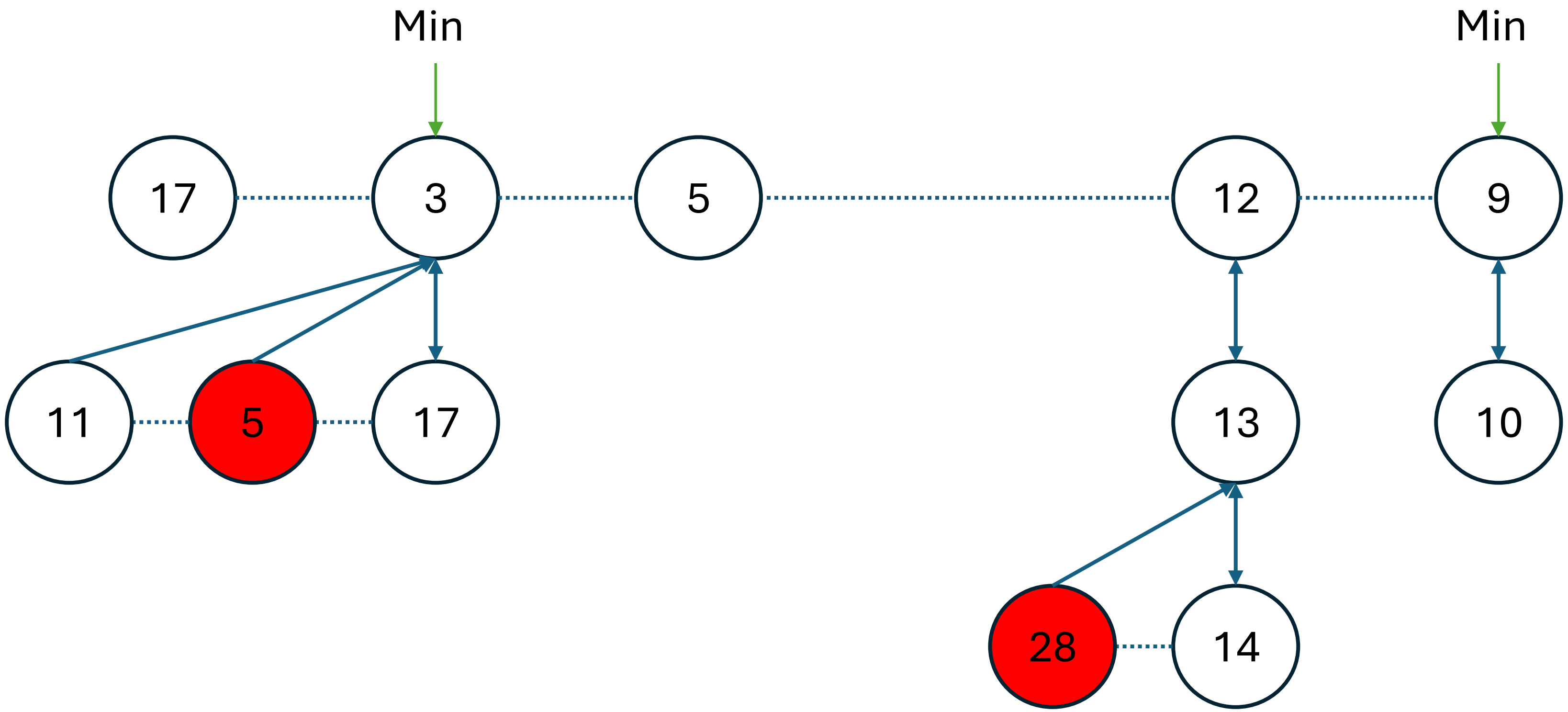
# Union : $O(1)$

- Combine root lists
- Update min



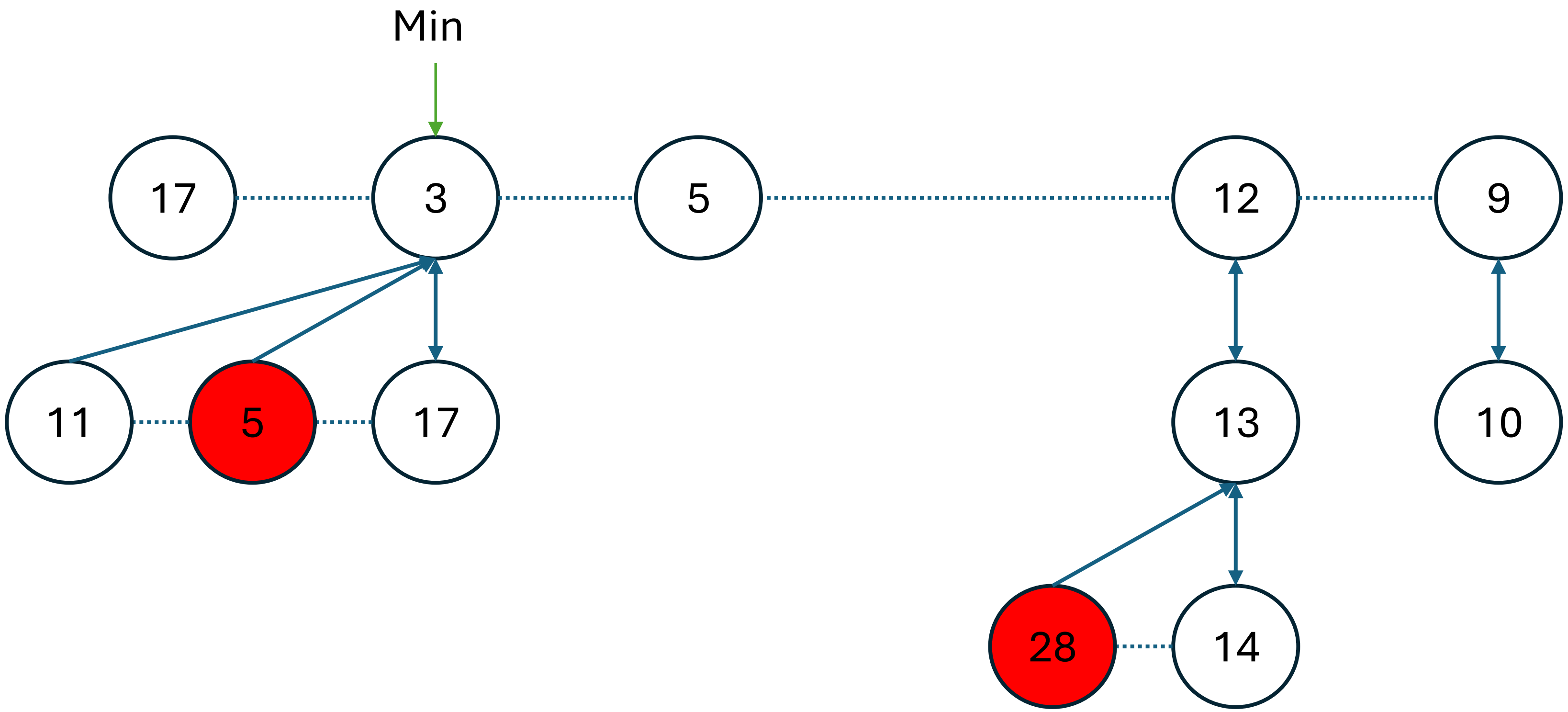
# Union : $O(1)$

- Combine root lists
- Update min



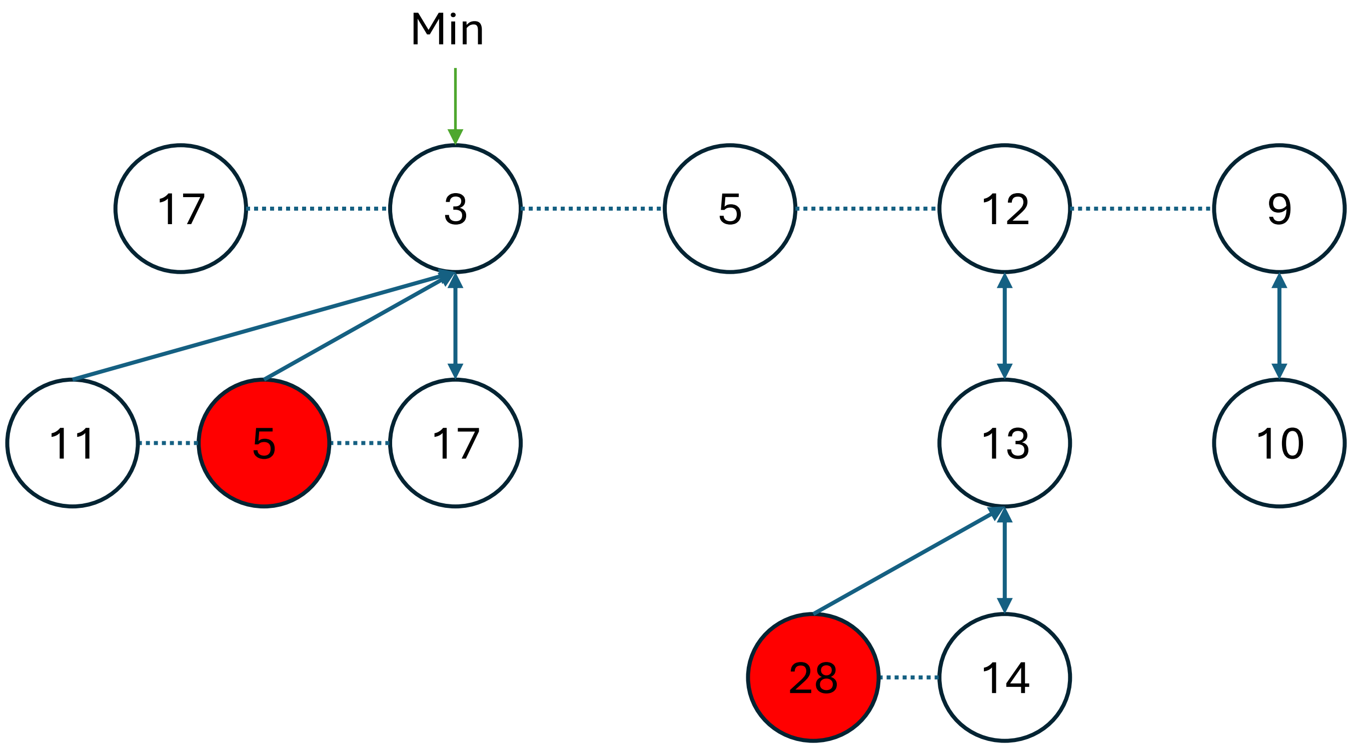
# Union : $O(1)$

- Combine root lists
- Update min



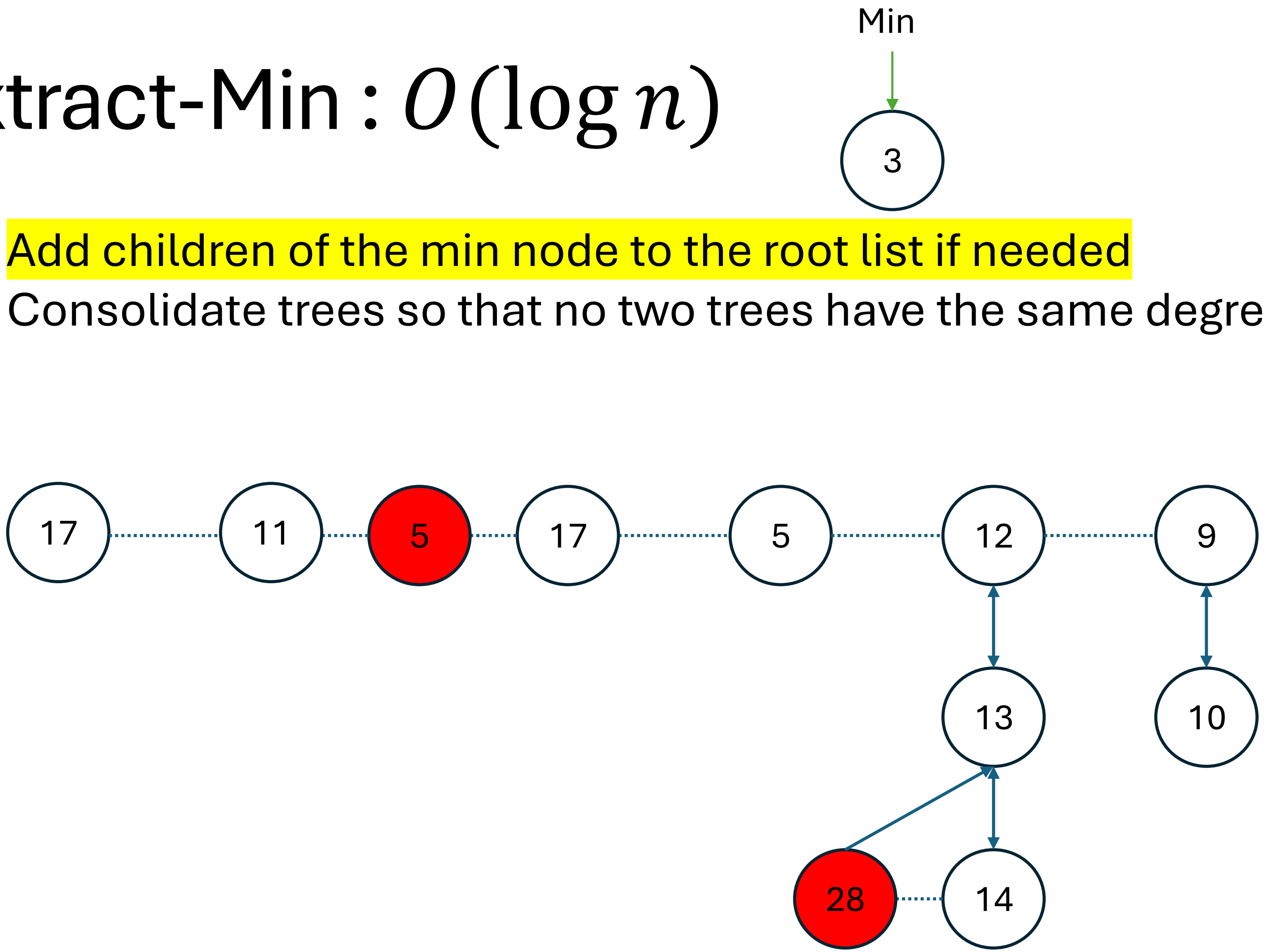
# Extract-Min : $O(\log n)$

- Add children of the min node to the root list if needed
- Consolidate trees so that no two trees have the same degree



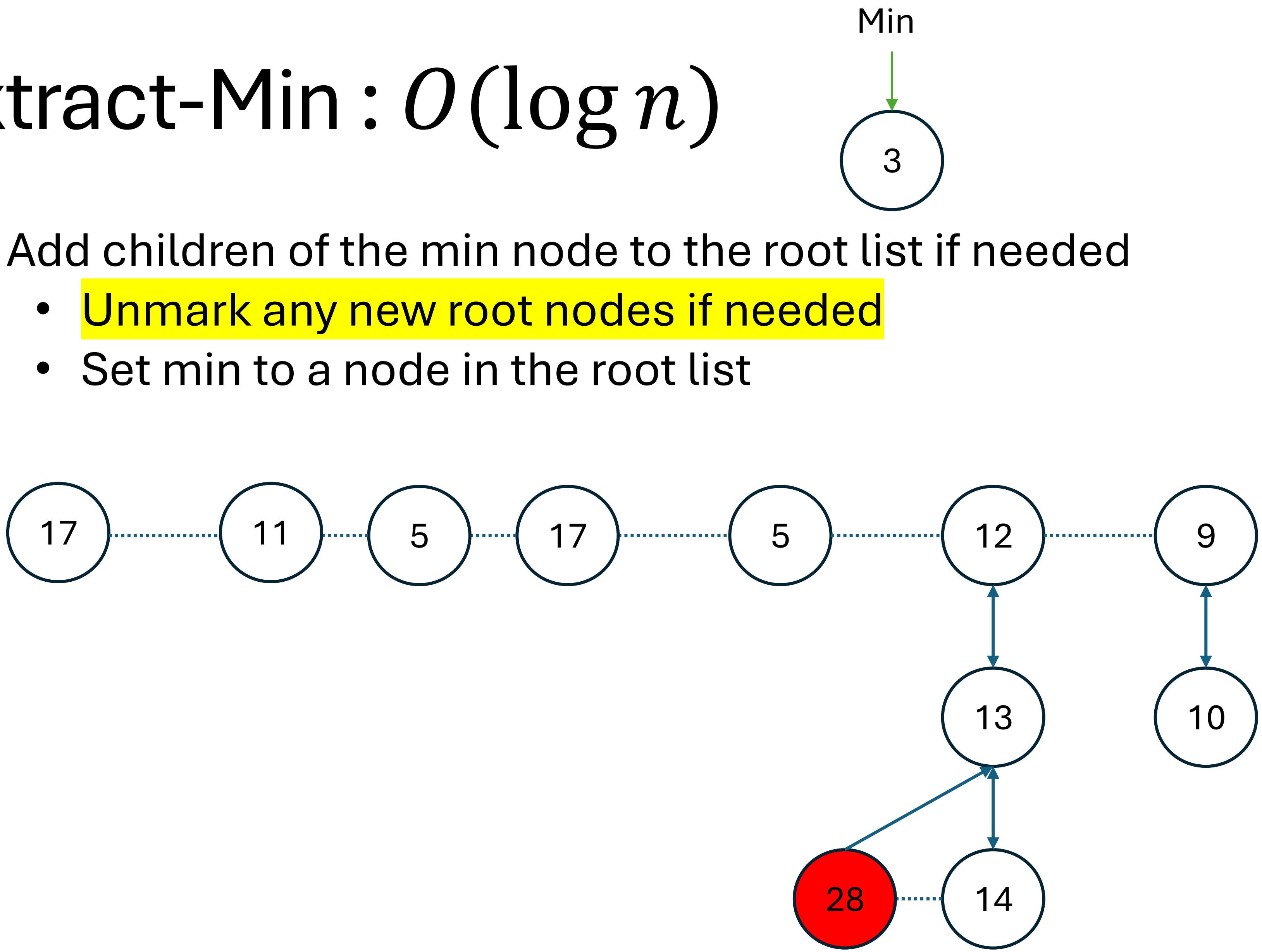
# Extract-Min : $O(\log n)$

- Add children of the min node to the root list if needed
- Consolidate trees so that no two trees have the same degree

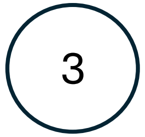


# Extract-Min : $O(\log n)$

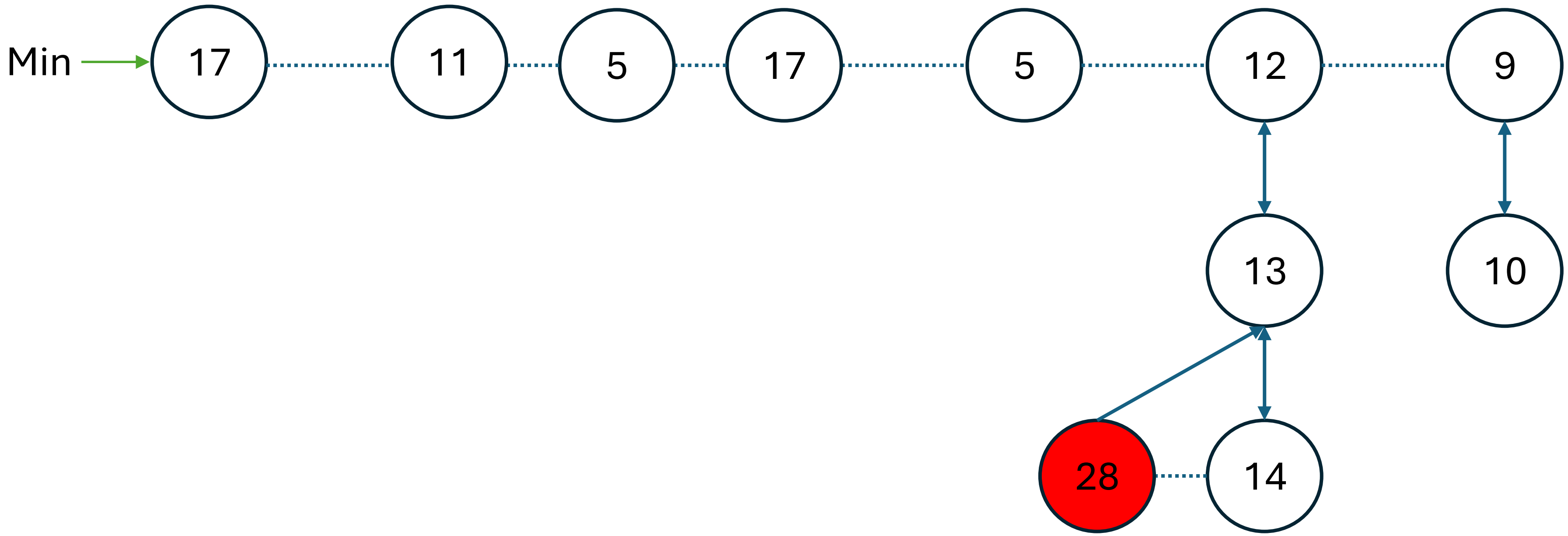
- Add children of the min node to the root list if needed
  - Unmark any new root nodes if needed
  - Set min to a node in the root list



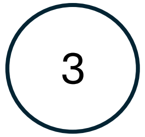
# Extract-Min : $O(\log n)$



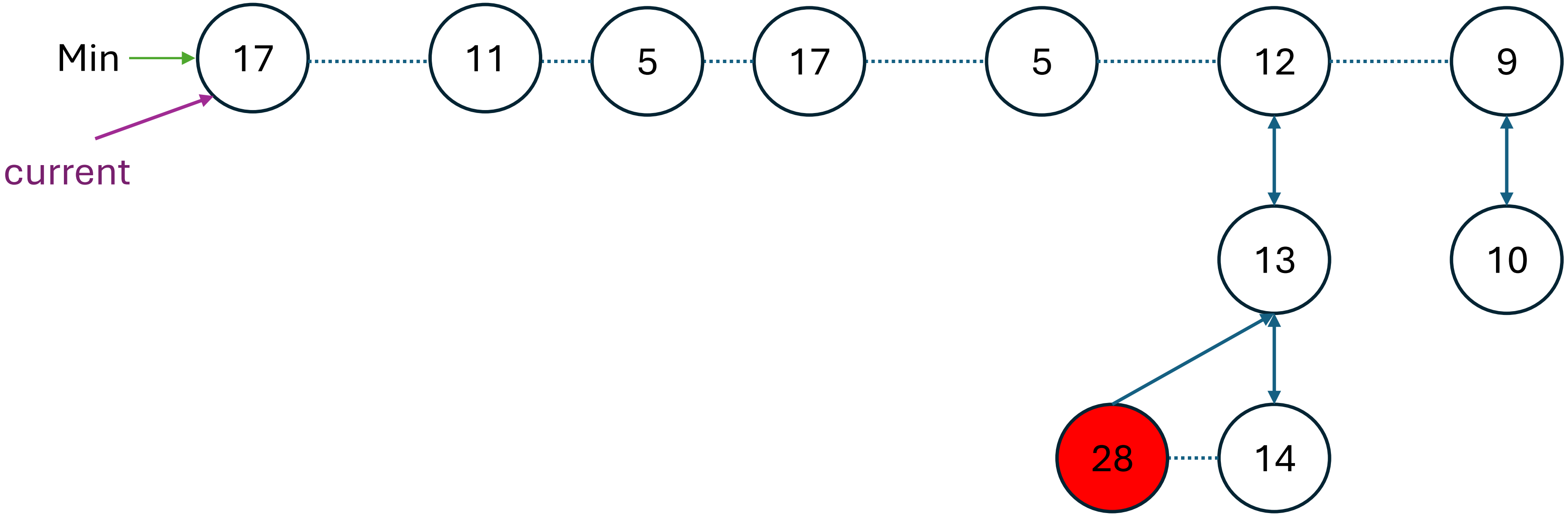

- Add children of the min node to the root list if needed
  - Unmark any new root nodes if needed
  - Set min to a node in the root list



# Extract-Min : $O(\log n)$

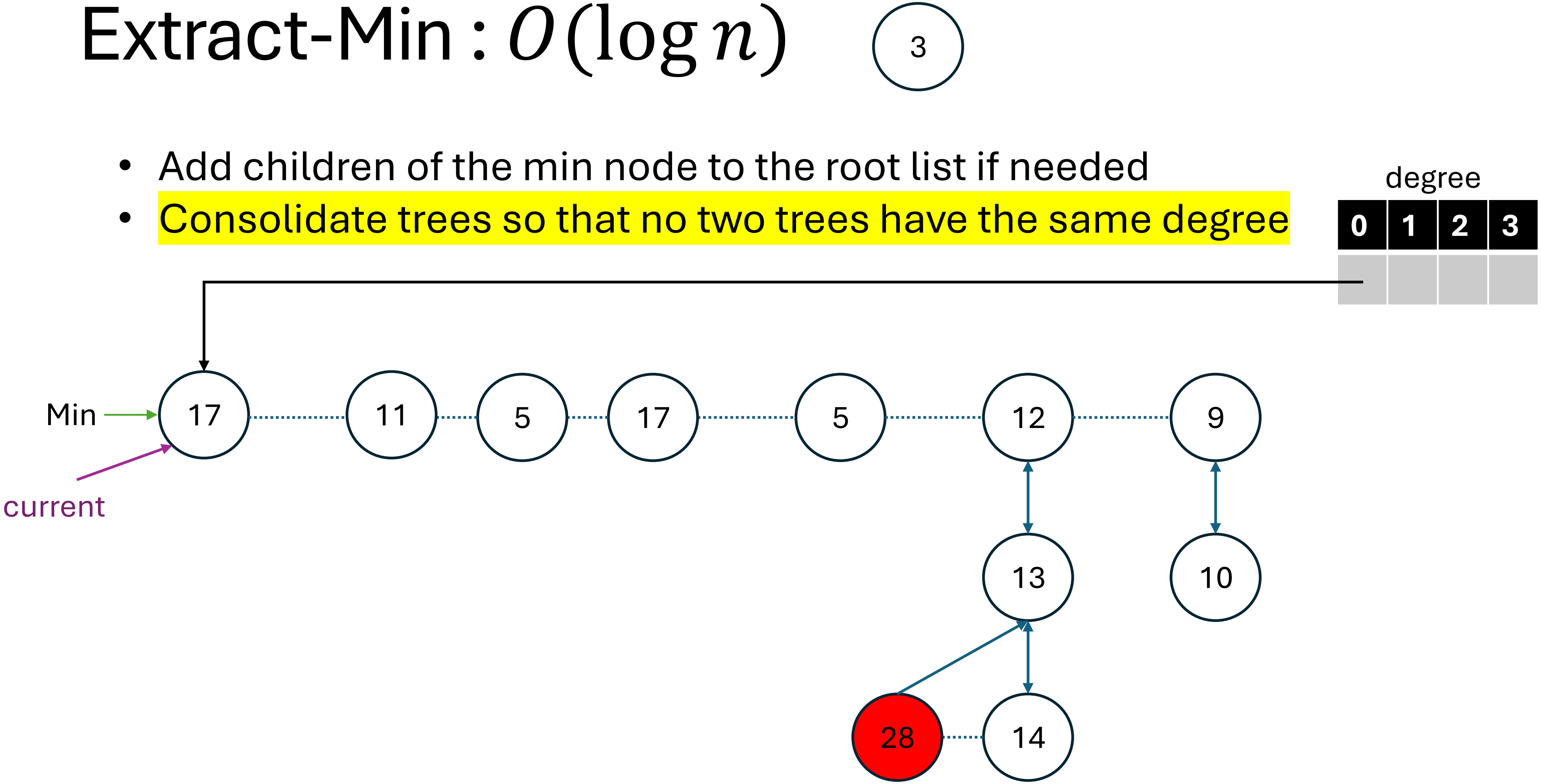



- Add children of the min node to the root list if needed
- Consolidate trees so that no two trees have the same degree



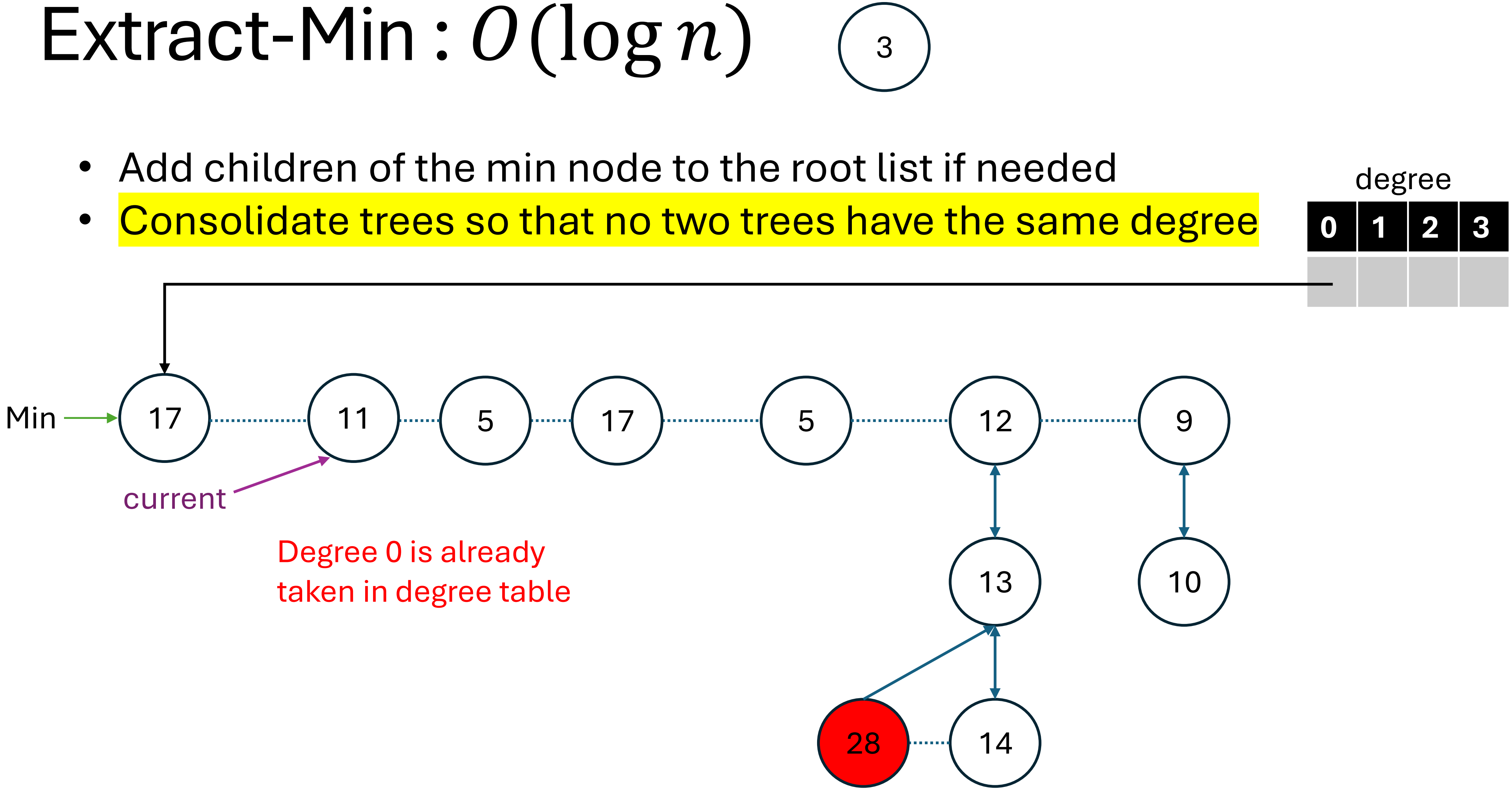
# Extract-Min : $O(\log n)$

- Add children of the min node to the root list if needed
- Consolidate trees so that no two trees have the same degree



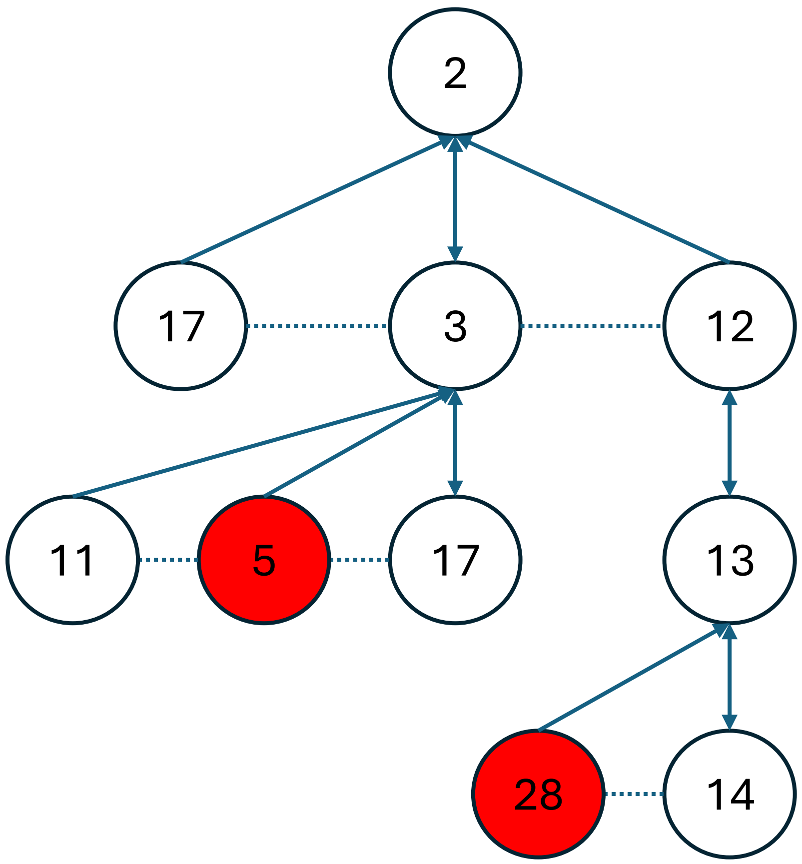
# Extract-Min : $O(\log n)$

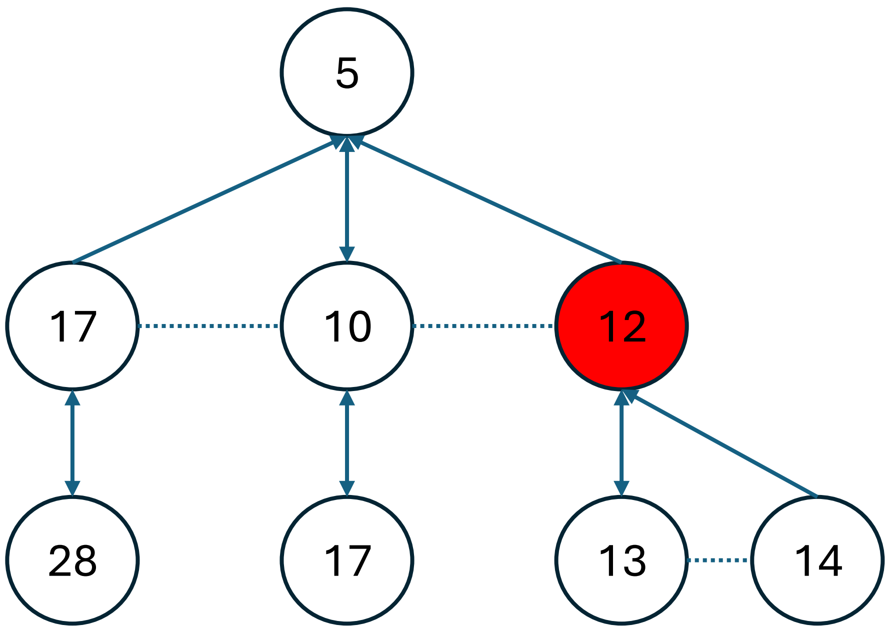
- Add children of the min node to the root list if needed
- Consolidate trees so that no two trees have the same degree



# Link

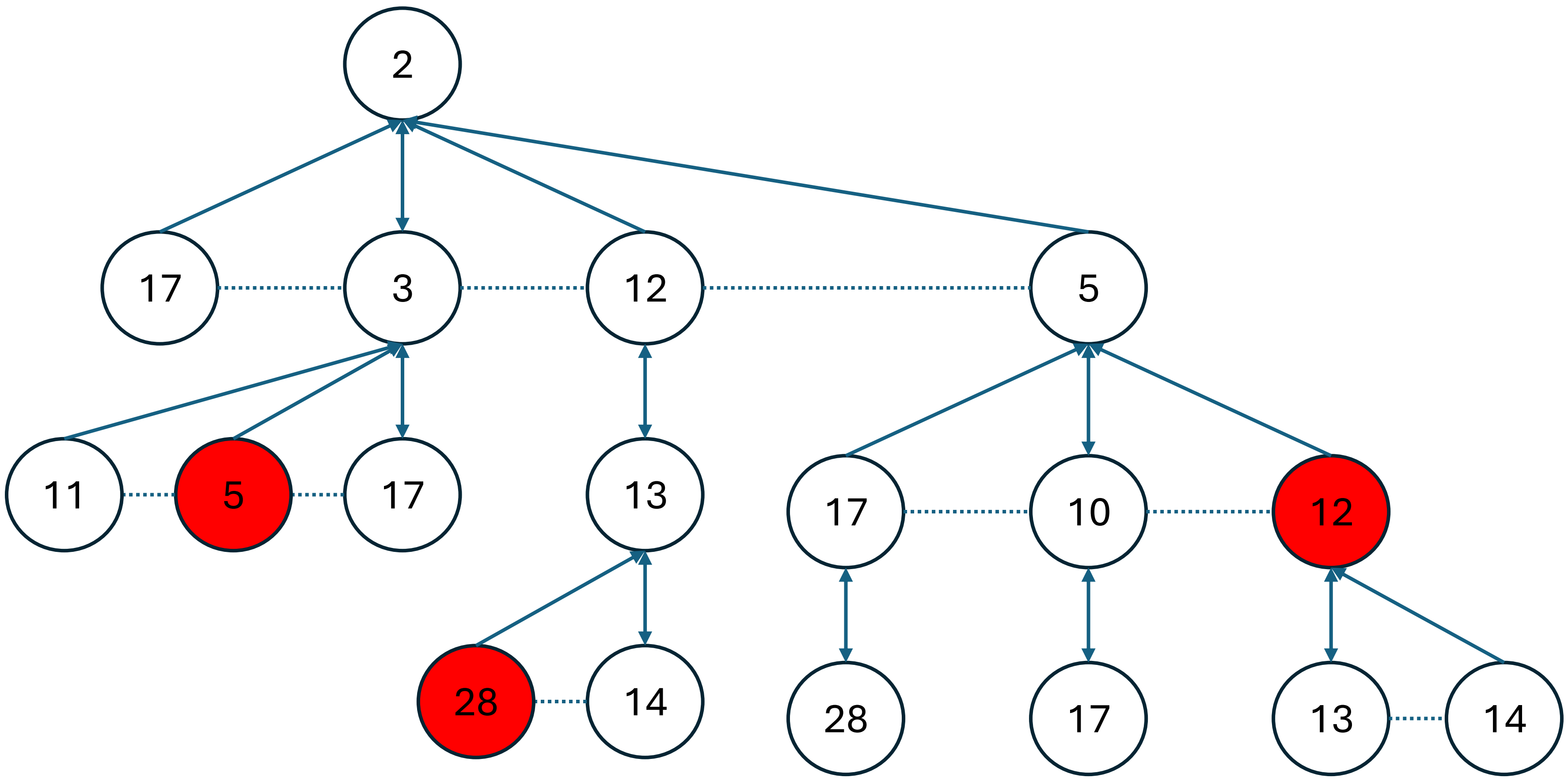
- Larger node becomes a child of the smaller node



# Link

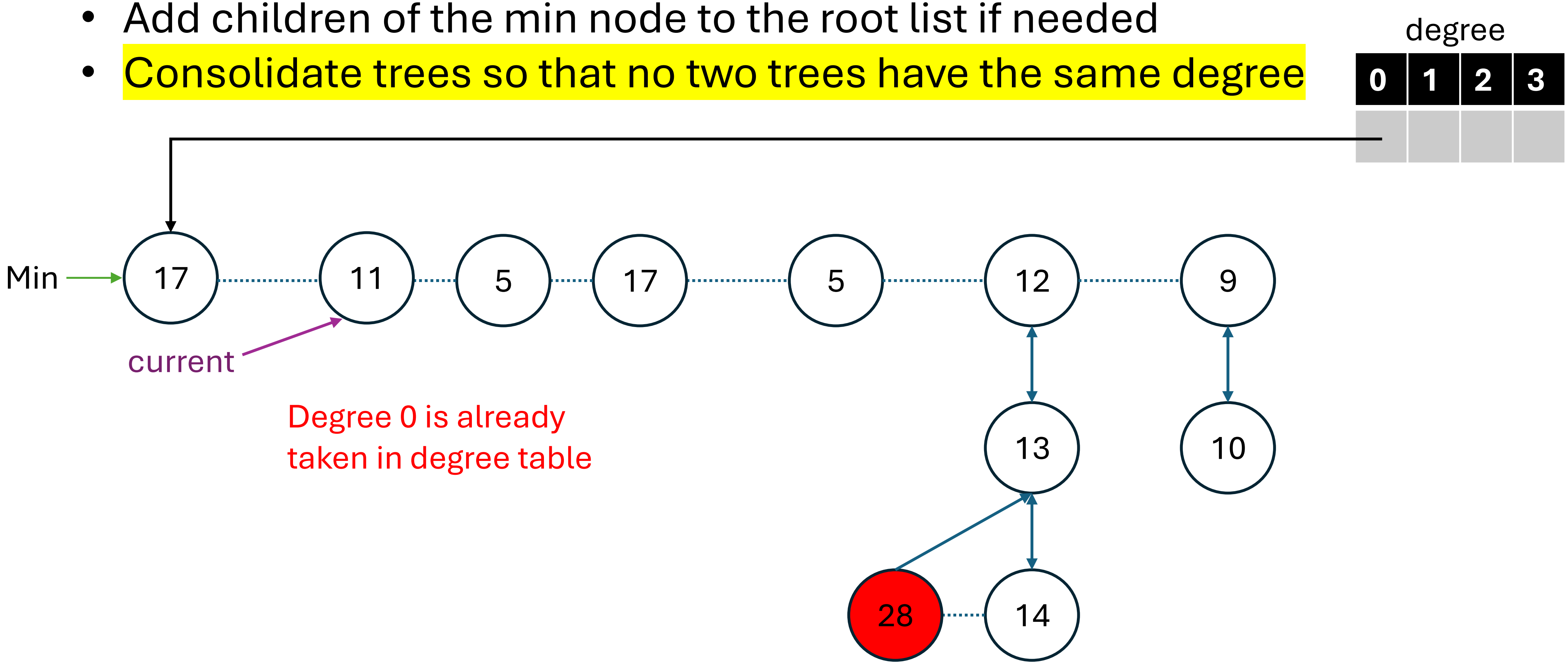
- Larger node becomes a child of the smaller node



# Extract-Min : $O(\log n)$

3

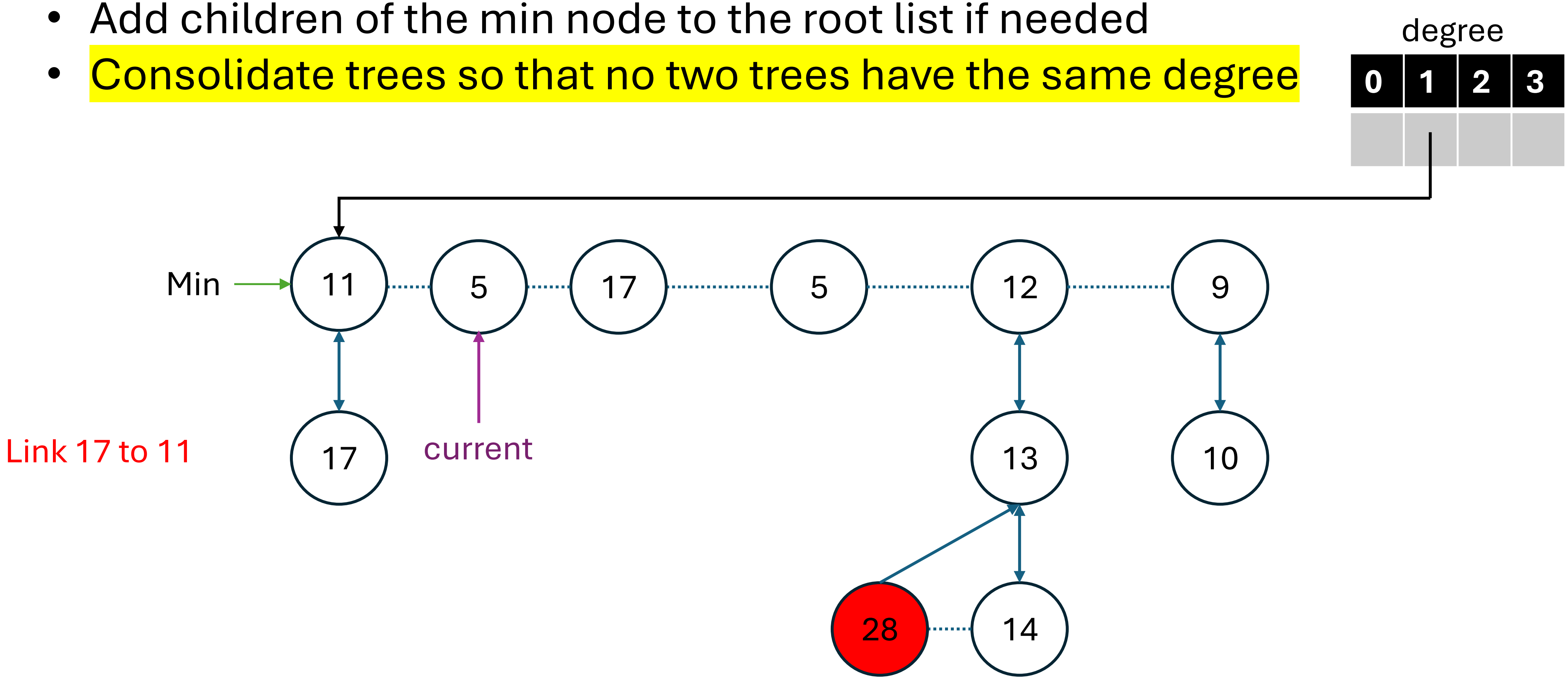
- Add children of the min node to the root list if needed
- Consolidate trees so that no two trees have the same degree



# Extract-Min : $O(\log n)$

3

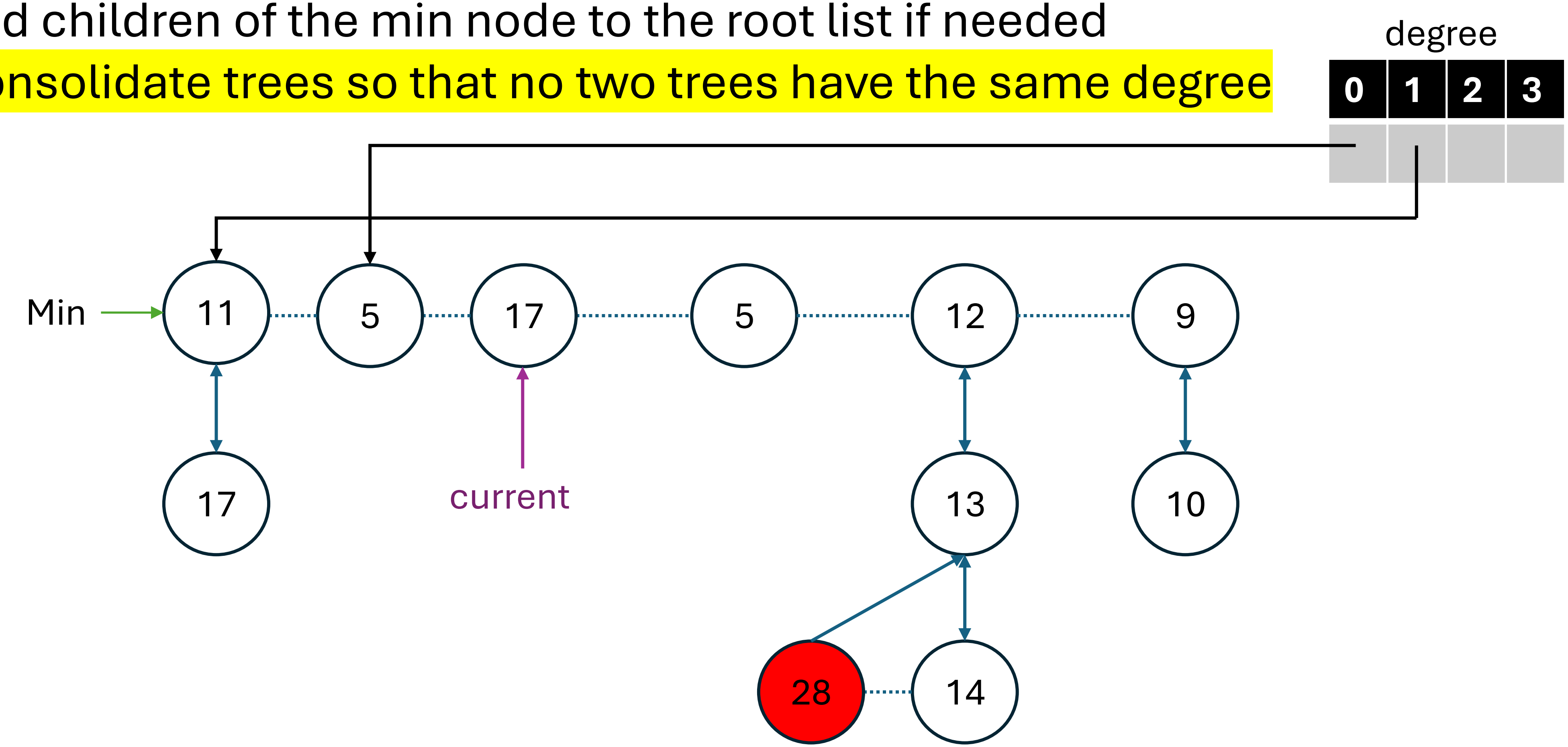
- Add children of the min node to the root list if needed
- Consolidate trees so that no two trees have the same degree



# Extract-Min : $O(\log n)$

3

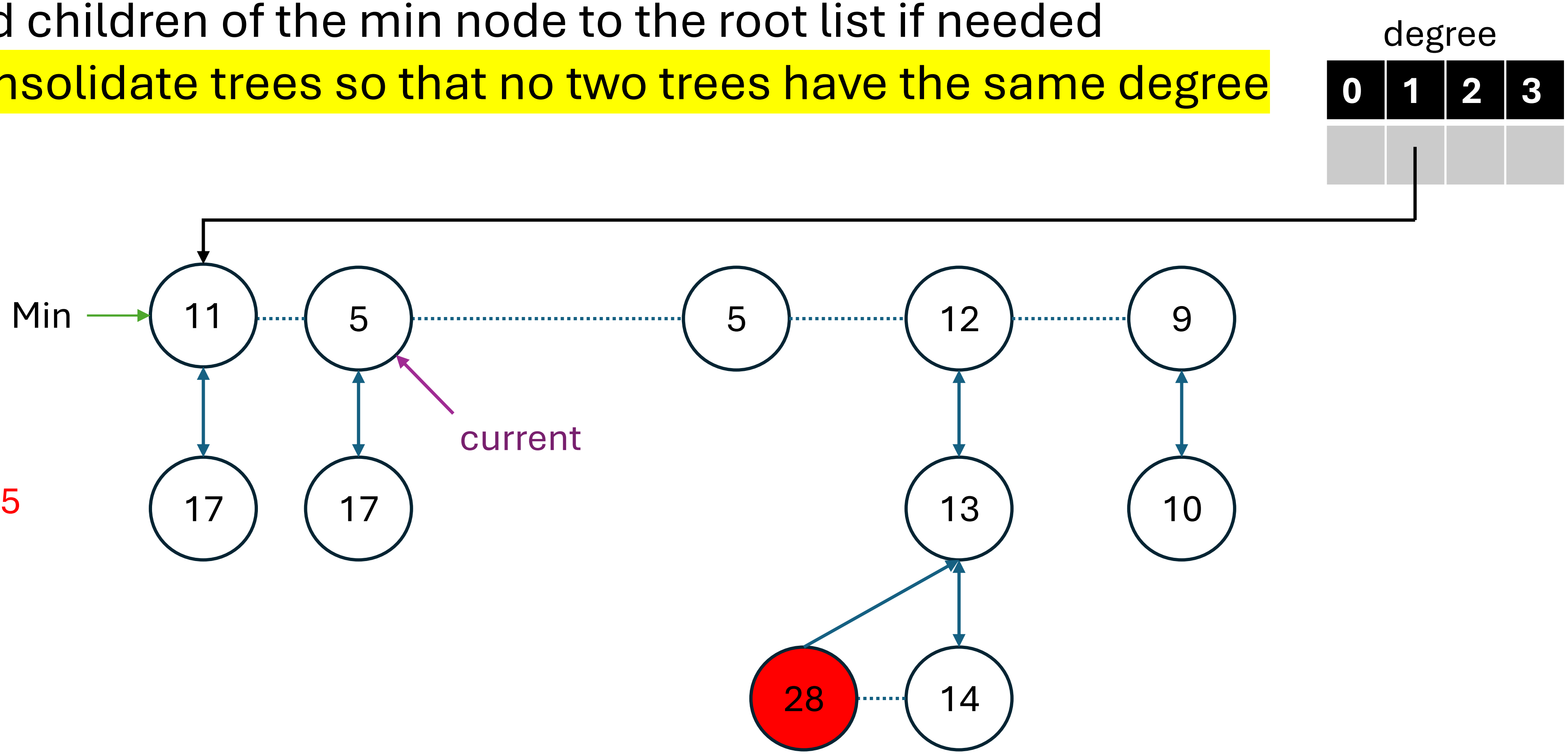
- Add children of the min node to the root list if needed
- Consolidate trees so that no two trees have the same degree



# Extract-Min : $O(\log n)$

3

- Add children of the min node to the root list if needed
- Consolidate trees so that no two trees have the same degree

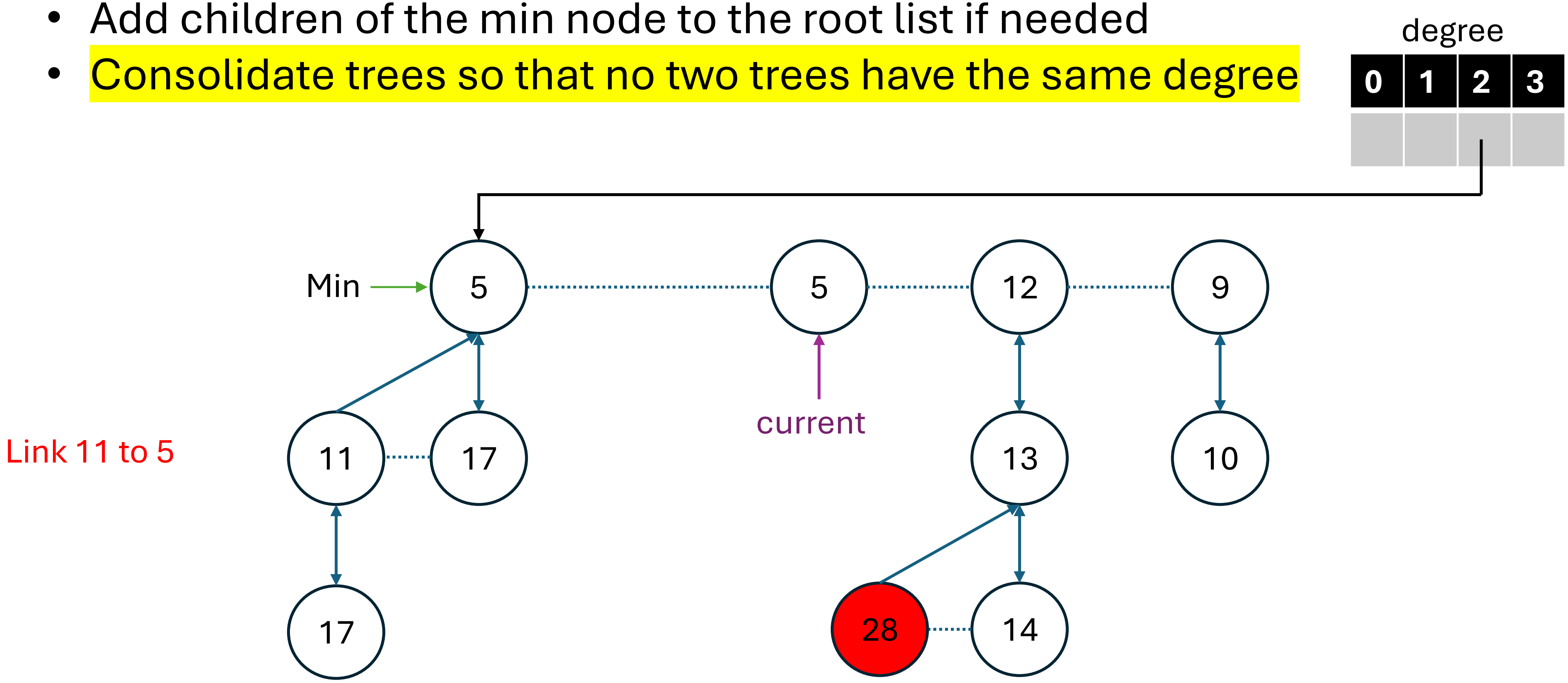



Link 17 to 5

# Extract-Min : $O(\log n)$

3

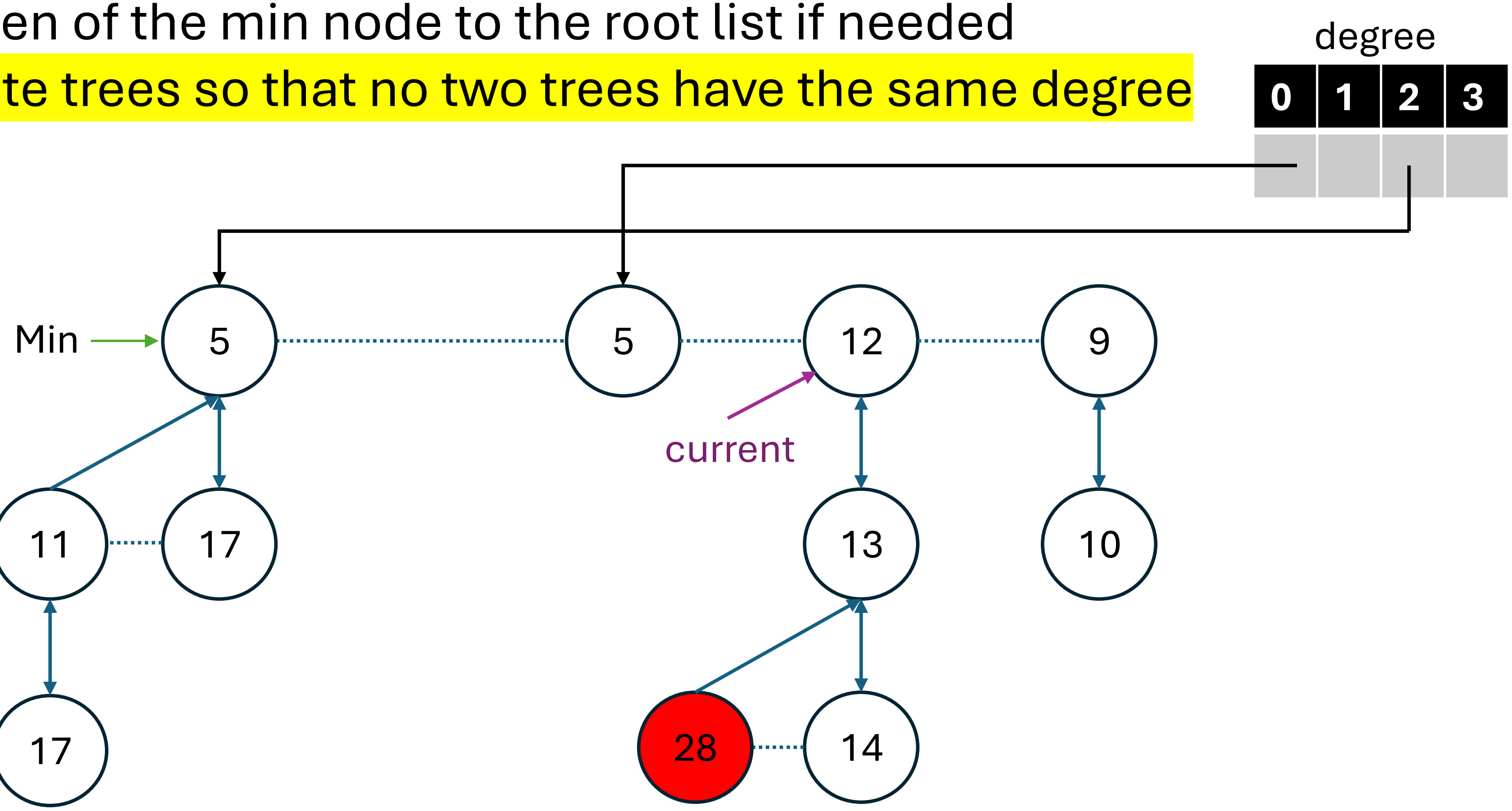
- Add children of the min node to the root list if needed
- Consolidate trees so that no two trees have the same degree



# Extract-Min : $O(\log n)$

3

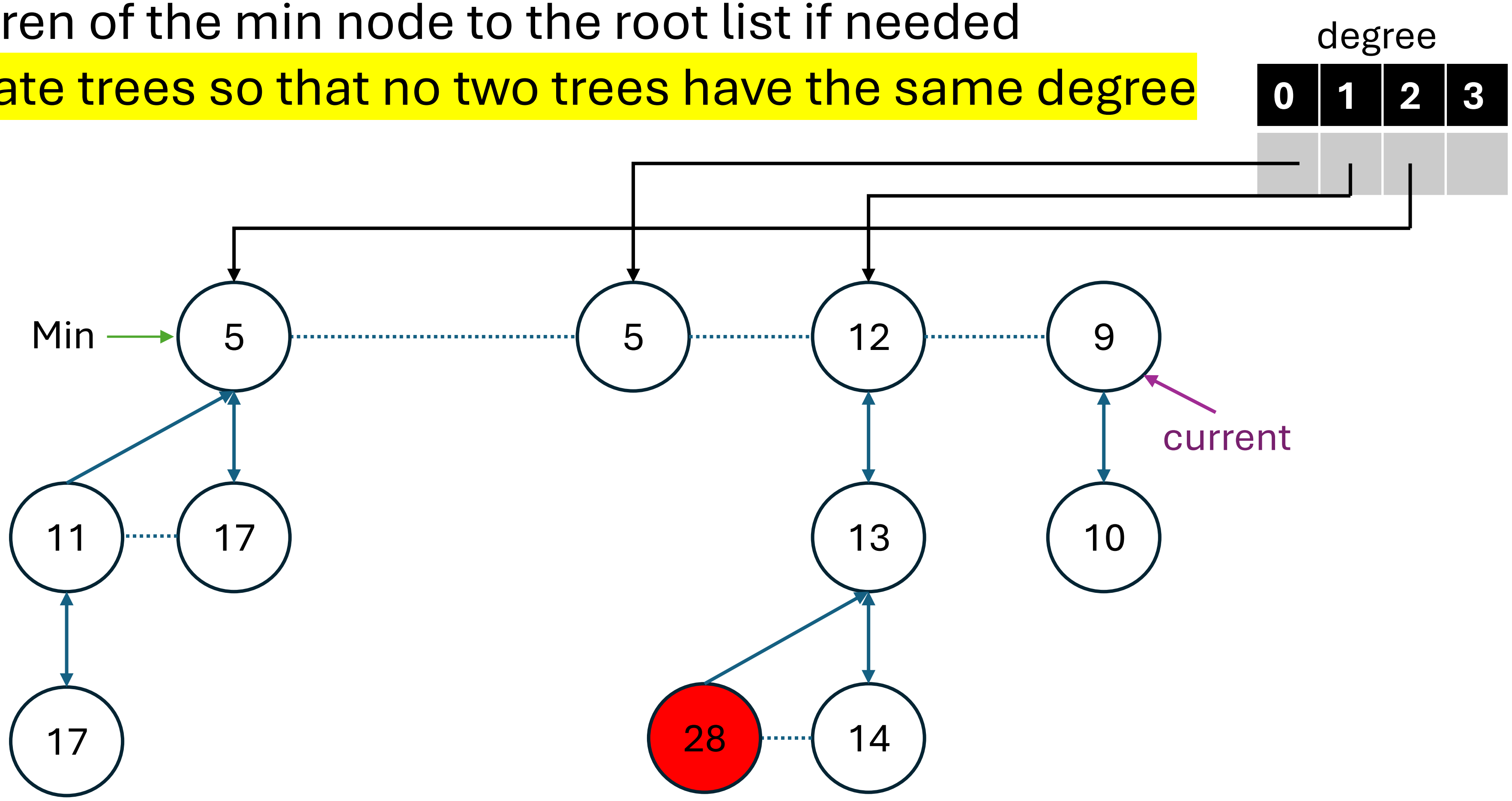
- Add children of the min node to the root list if needed
- Consolidate trees so that no two trees have the same degree



# Extract-Min : $O(\log n)$

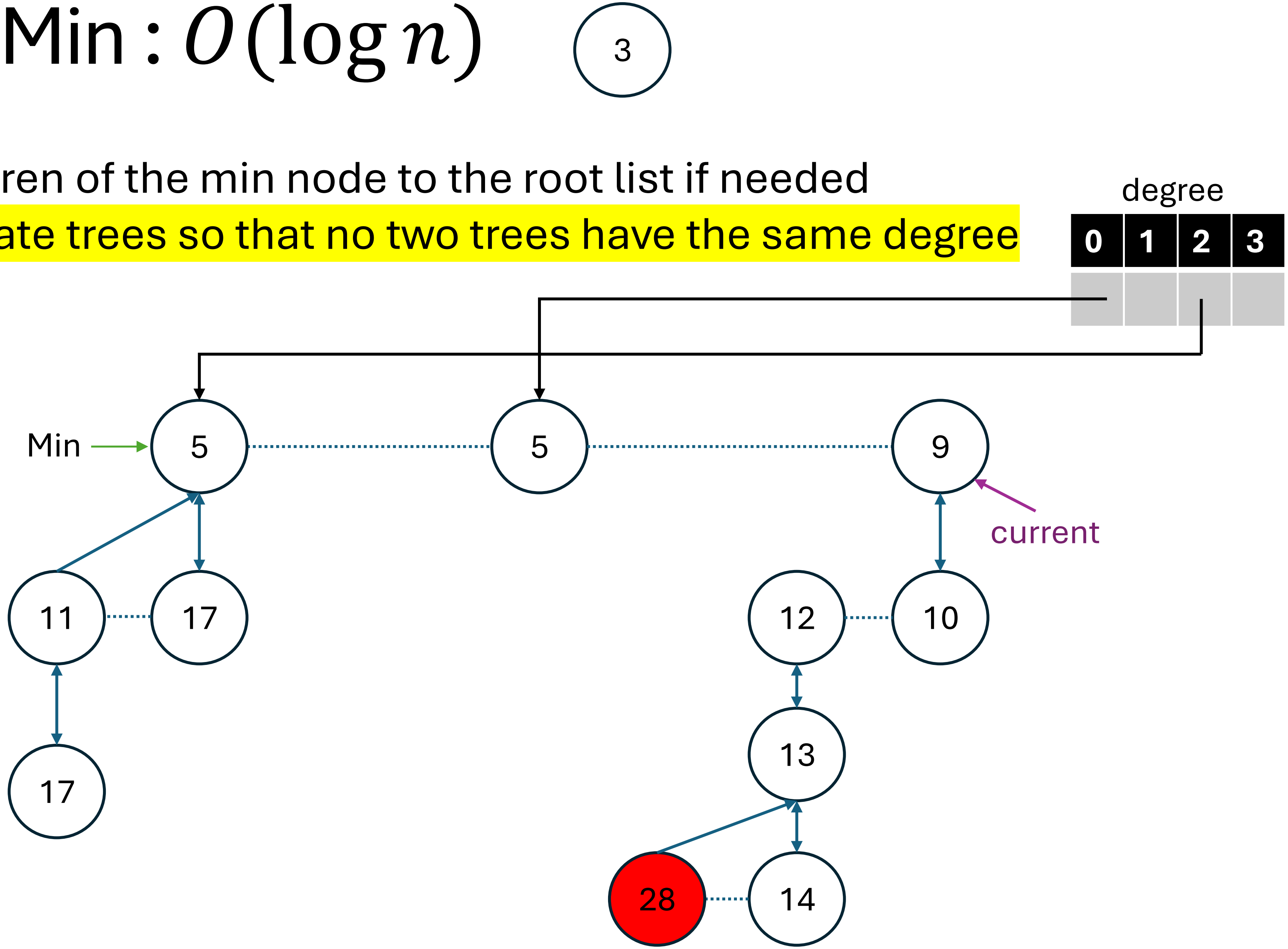
3

- Add children of the min node to the root list if needed
- Consolidate trees so that no two trees have the same degree



# Extract-Min : $O(\log n)$

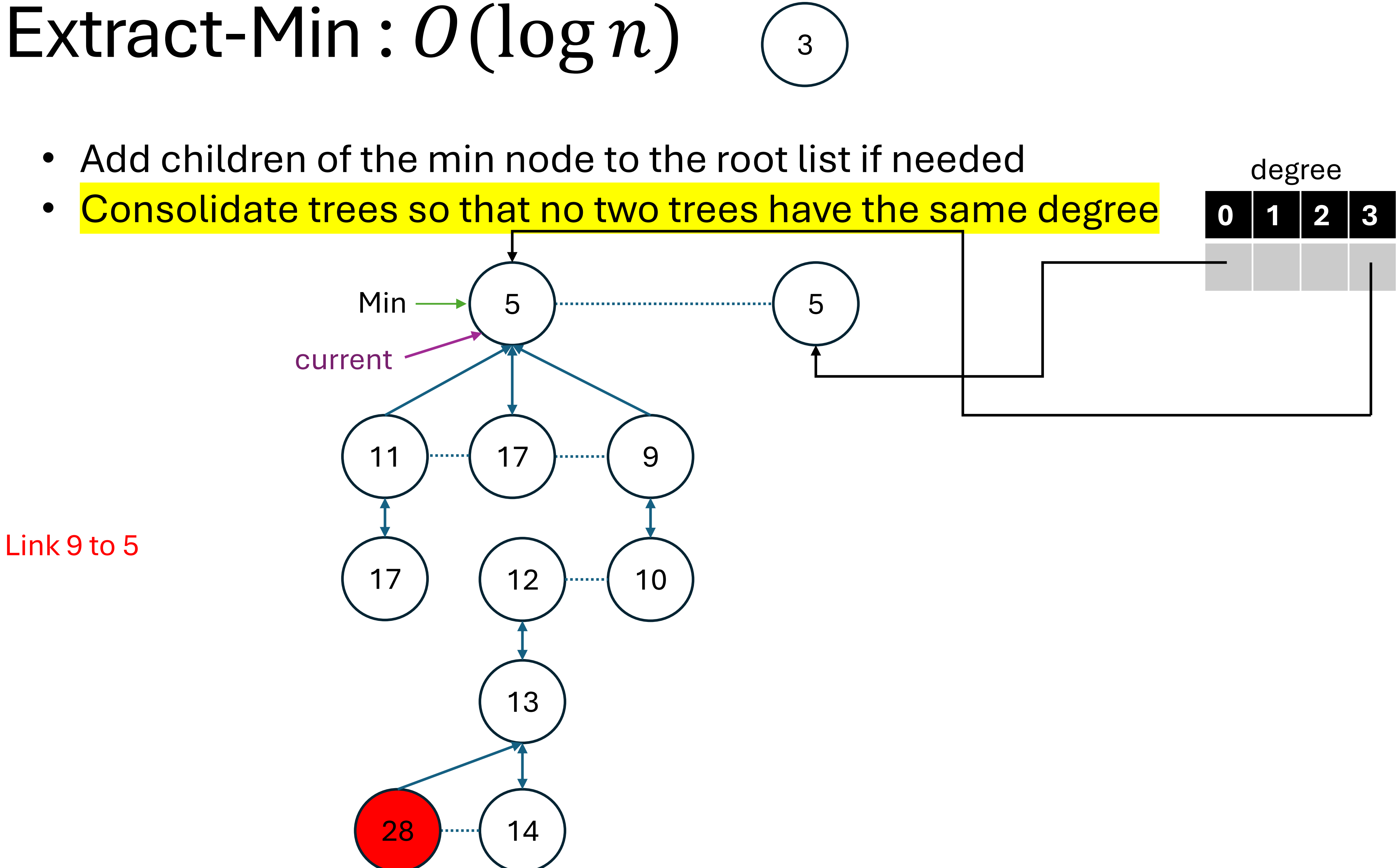
- Add children of the min node to the root list if needed
- Consolidate trees so that no two trees have the same degree




Link 12 to 9

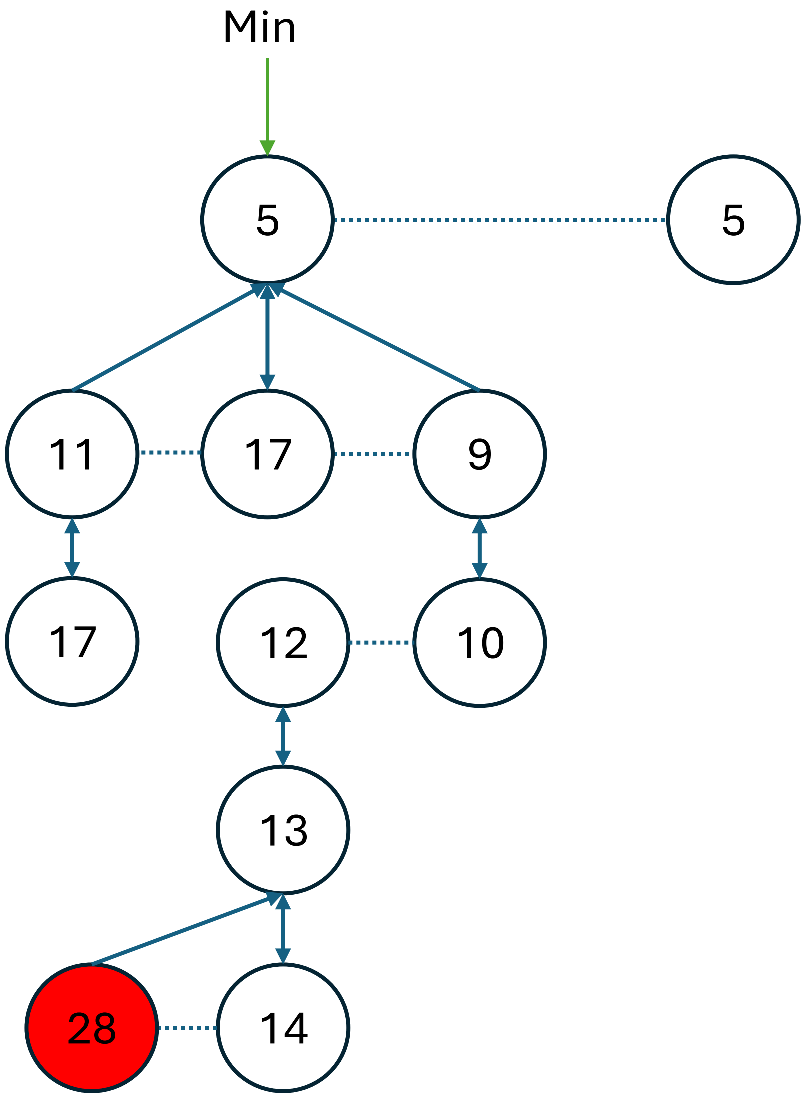
# Extract-Min : $O(\log n)$

- Add children of the min node to the root list if needed
- Consolidate trees so that no two trees have the same degree



# Extract-Min : $O(\log n)$

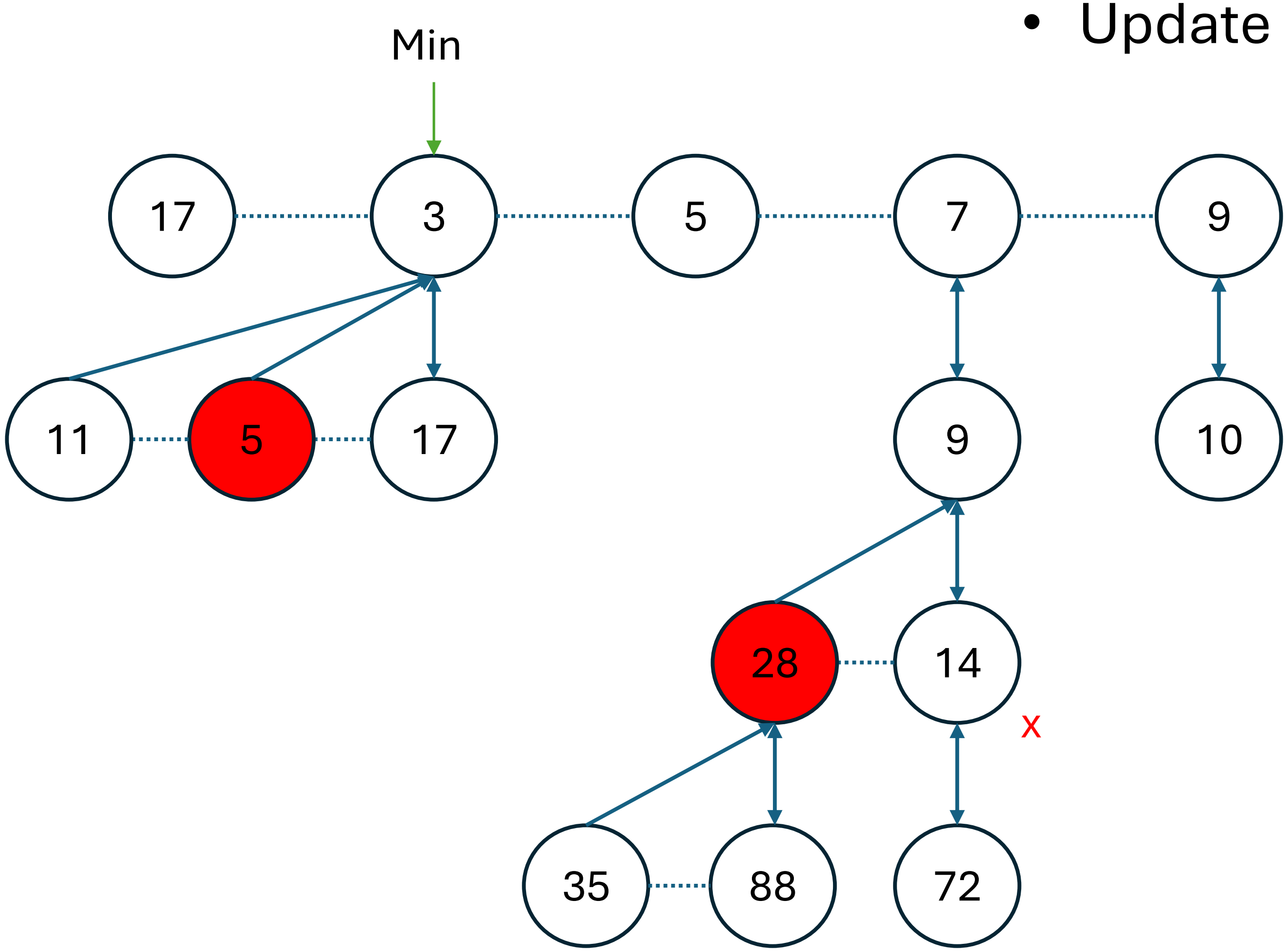
- Update min if needed




# Decrease-Key : $O(1)$

Case 1

- If heap order is not violated, just decrease key of node
- Update min if needed

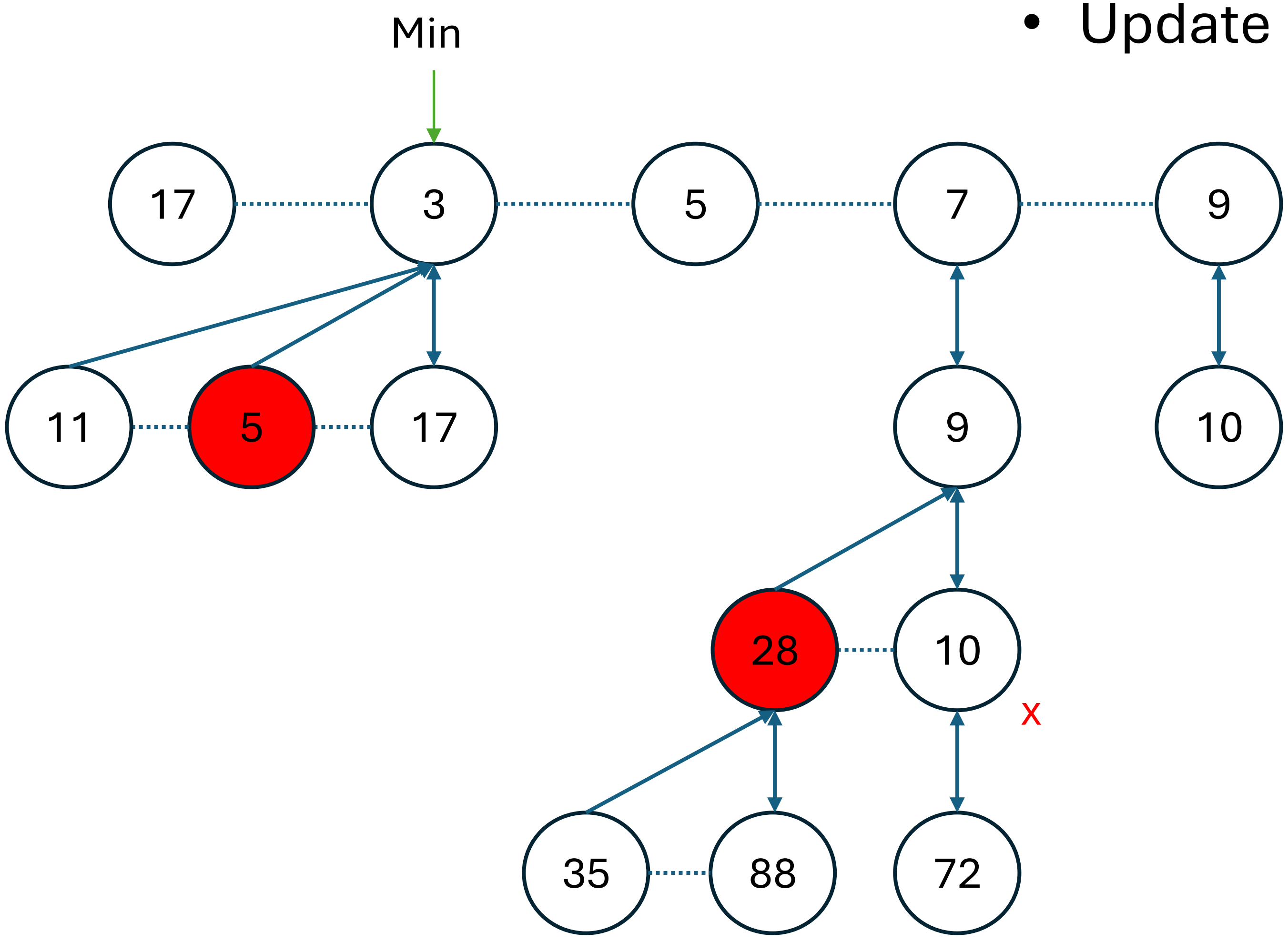



Decrease x from 14 to 10

# Decrease-Key : $O(1)$

Case 1

- If heap order is not violated, just decrease key of node
- Update min if needed

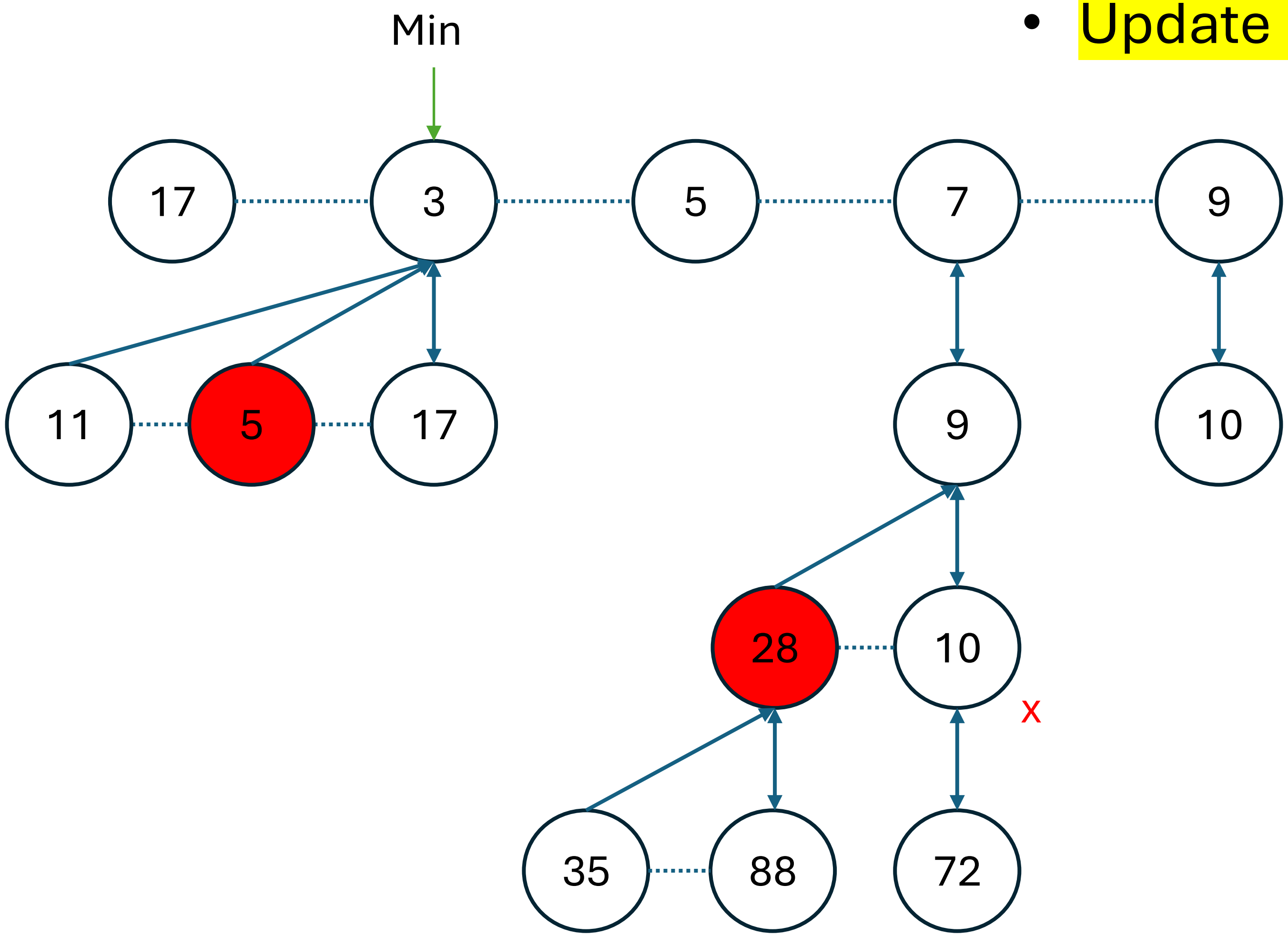



Decrease x from 14 to 10

# Decrease-Key : $O(1)$

Case 1

- If heap order is not violated, just decrease key of node
- Update min if needed

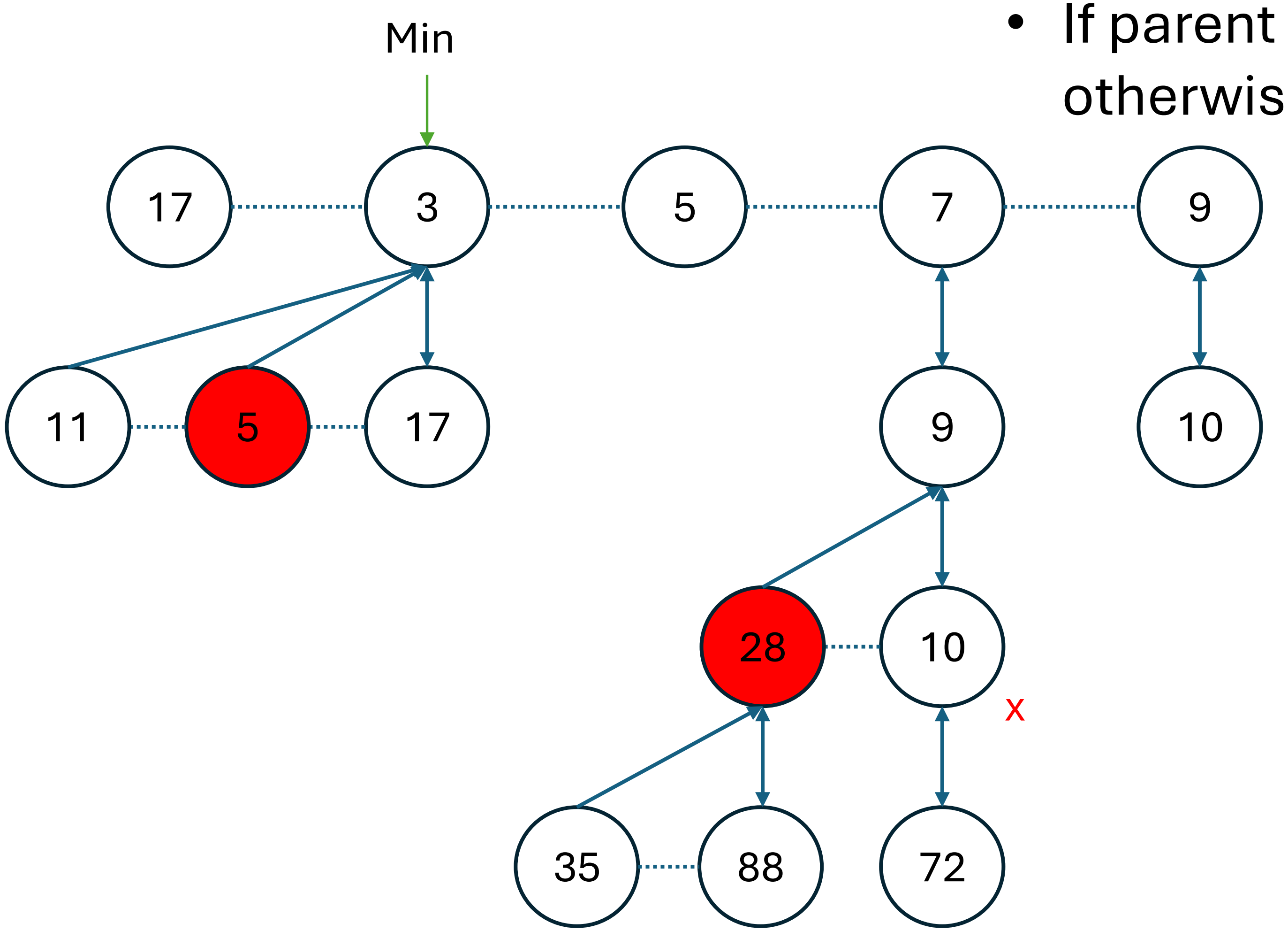



Decrease x from 14 to 10

# Decrease-Key : $O(1)$

Case 2a [Heap order violated]

- Decrease key of node
- Cut node and put in root list
- If parent of node is unmarked, mark it, otherwise perform cascading cut

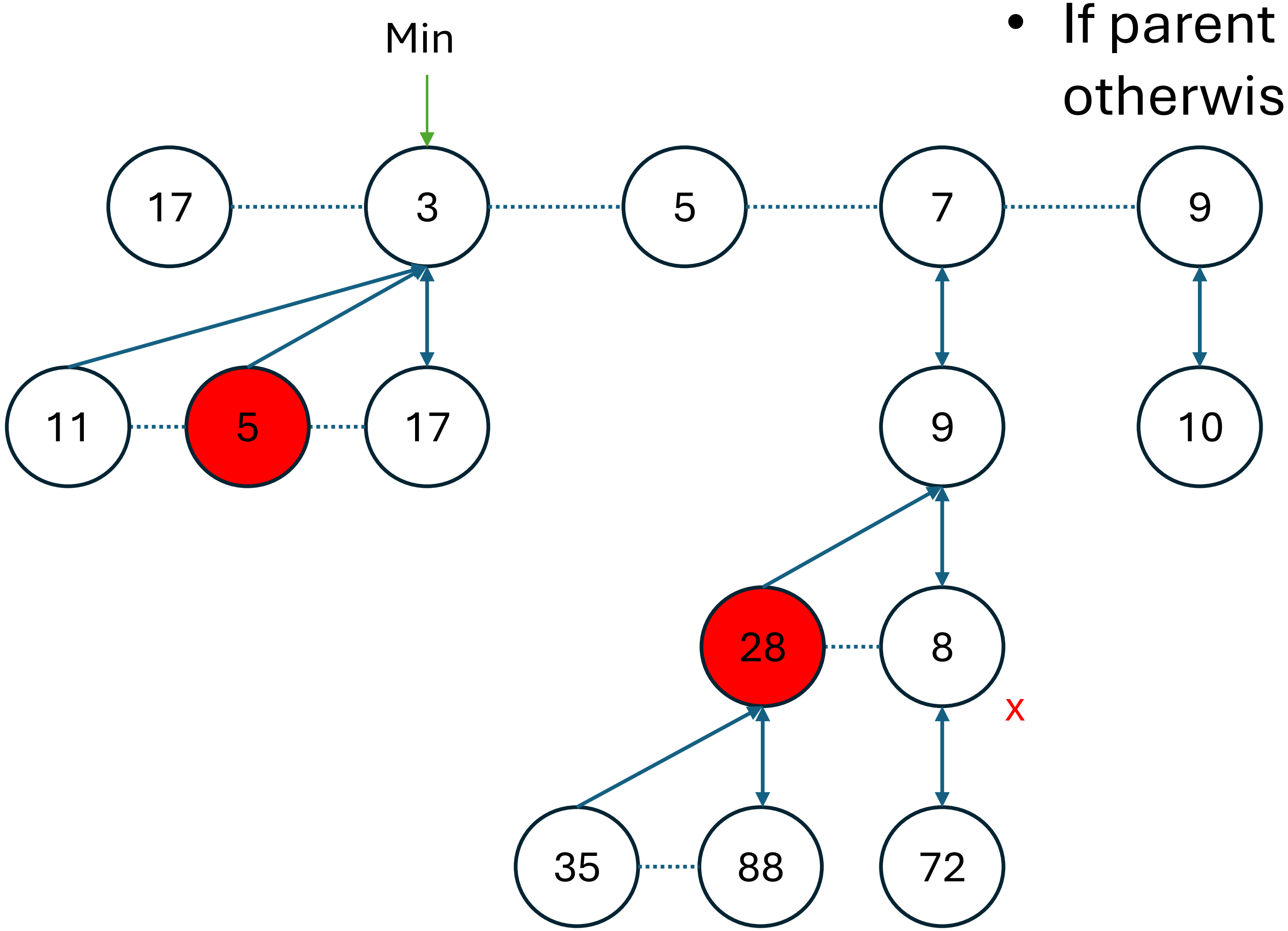



Decrease x from 10 to 8

# Decrease-Key : $O(1)$

Case 2a [Heap order violated]

- Decrease key of node
- Cut node and put in root list
- If parent of node is unmarked, mark it, otherwise perform cascading cut

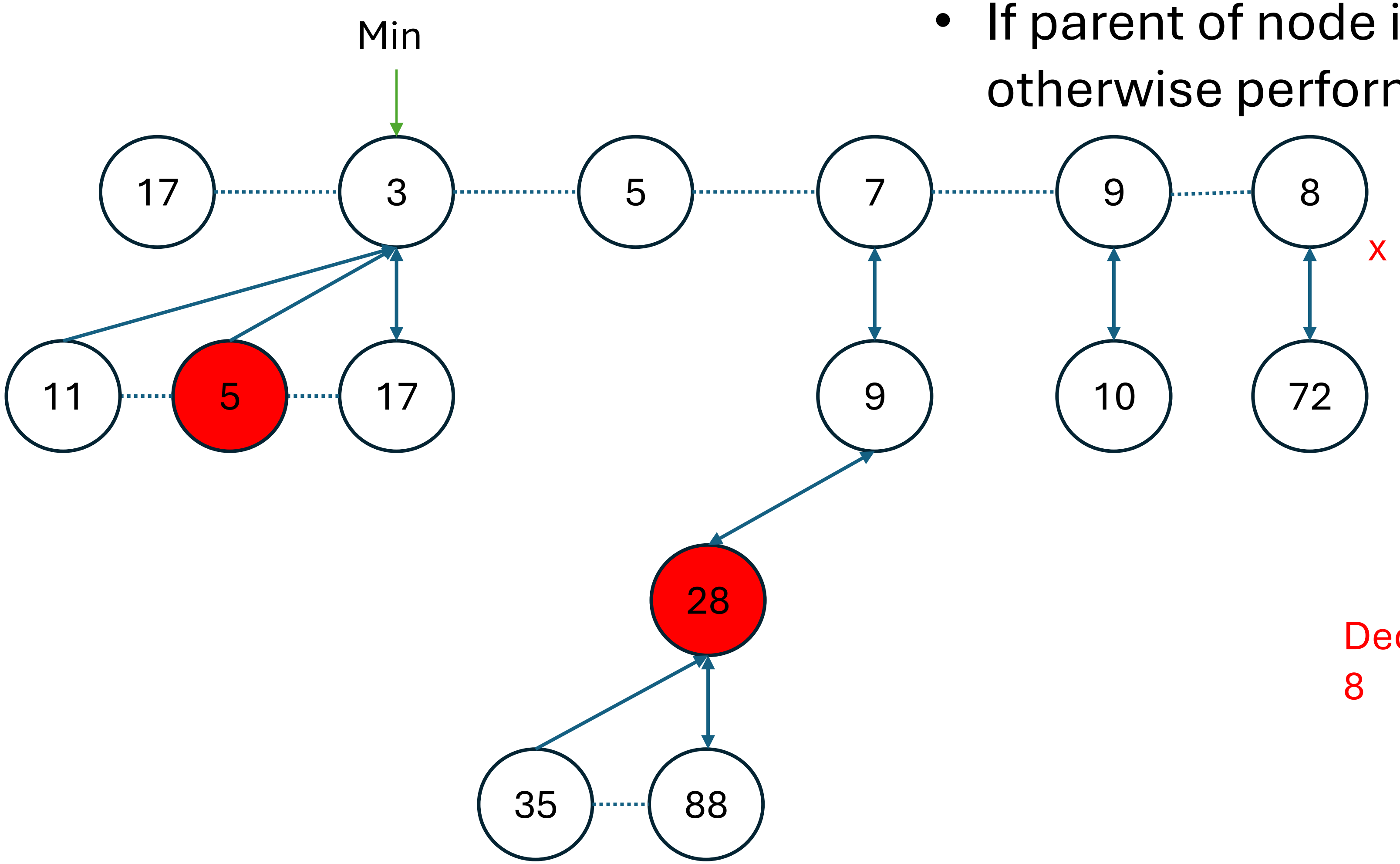



Decrease x from 10 to 8

# Decrease-Key : $O(1)$

Case 2a [Heap order violated]

- Decrease key of node
- Cut node and put in root list
- If parent of node is unmarked, mark it, otherwise perform cascading cut

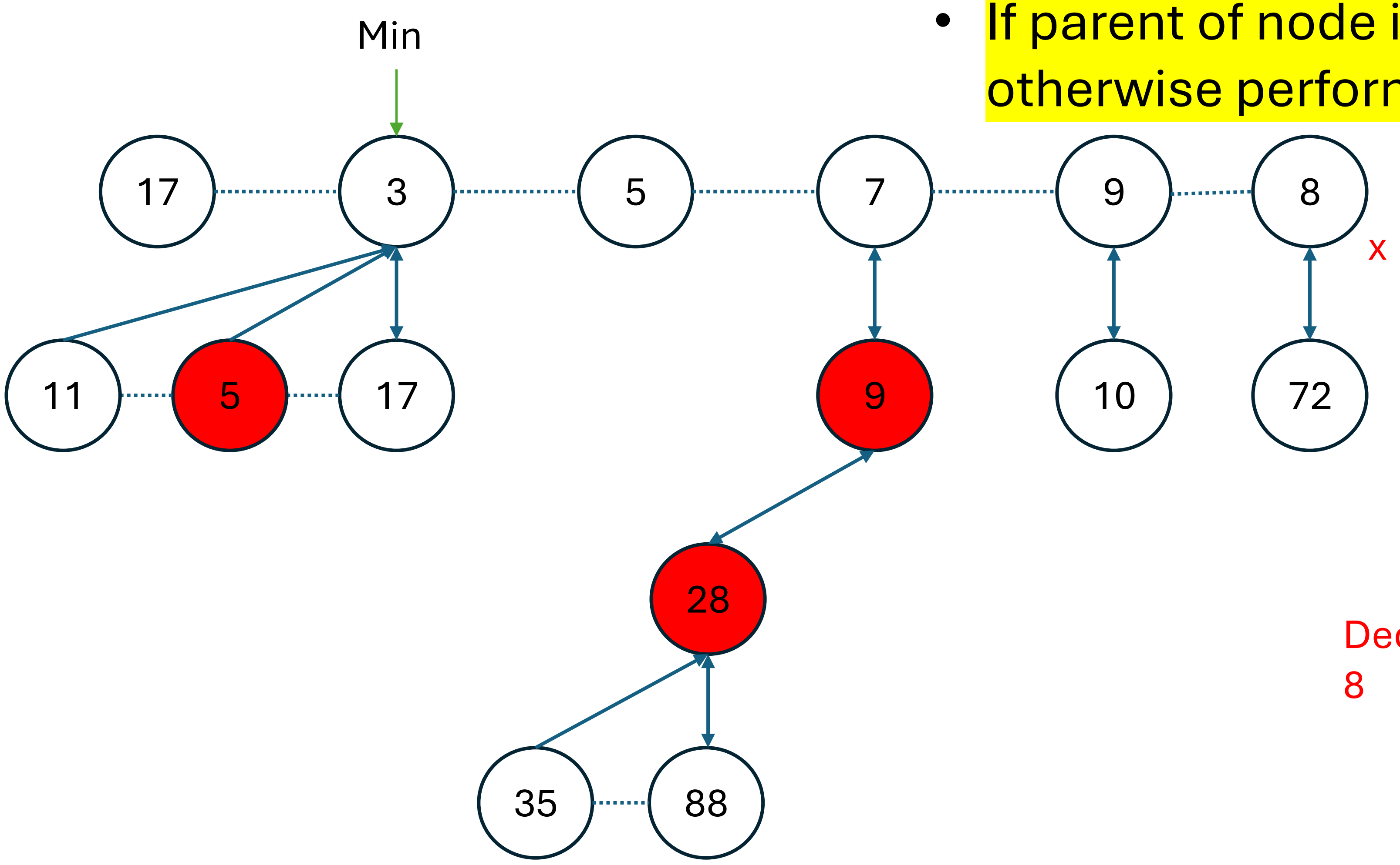



Decrease x from 10 to 8

# Decrease-Key : $O(1)$

Case 2a [Heap order violated]

- Decrease key of node
- Cut node and put in root list
- If parent of node is unmarked, mark it, otherwise perform cascading cut

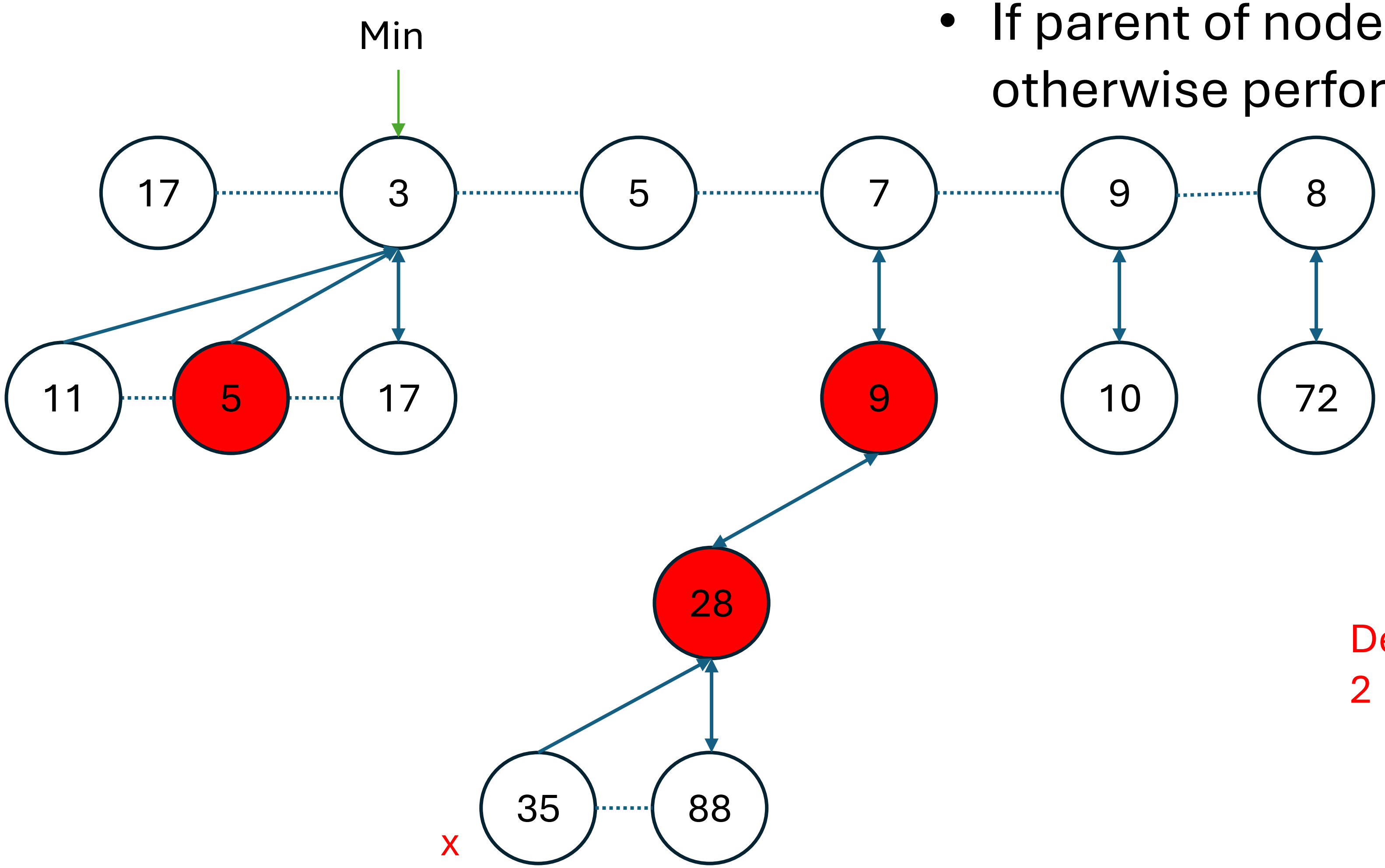



Decrease x from 10 to 8

# Decrease-Key : $O(1)$

Case 2b [Heap order violated]

- Decrease key of node
- Cut node and put in root list
- If parent of node is unmarked, mark it, otherwise perform cascading cut

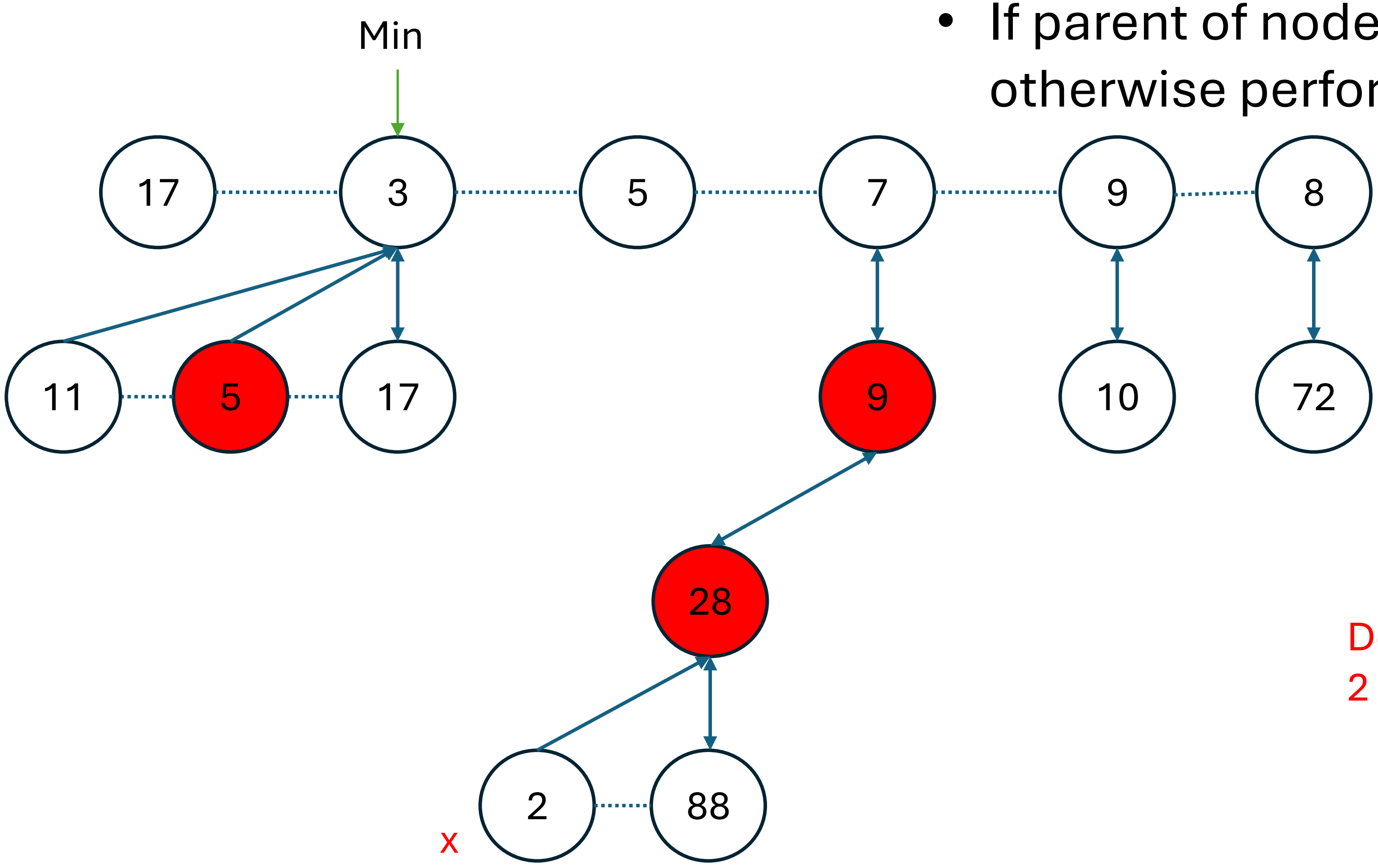



Decrease x from 35 to 2

# Decrease-Key : $O(1)$

Case 2b [Heap order violated]

- Decrease key of node
- Cut node and put in root list
- If parent of node is unmarked, mark it, otherwise perform cascading cut

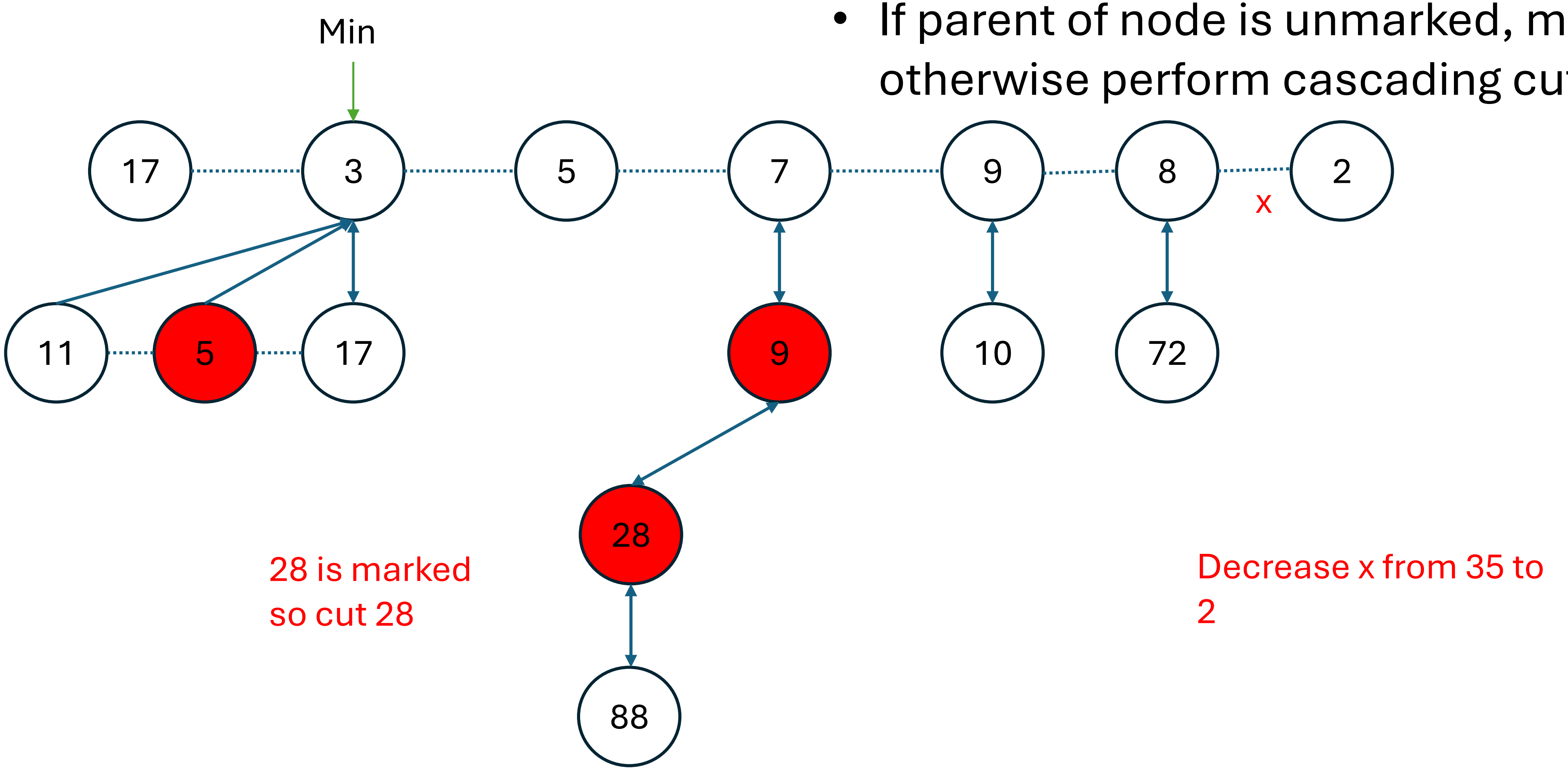



Decrease x from 35 to 2

# Decrease-Key : $O(1)$

Case 2b [Heap order violated]

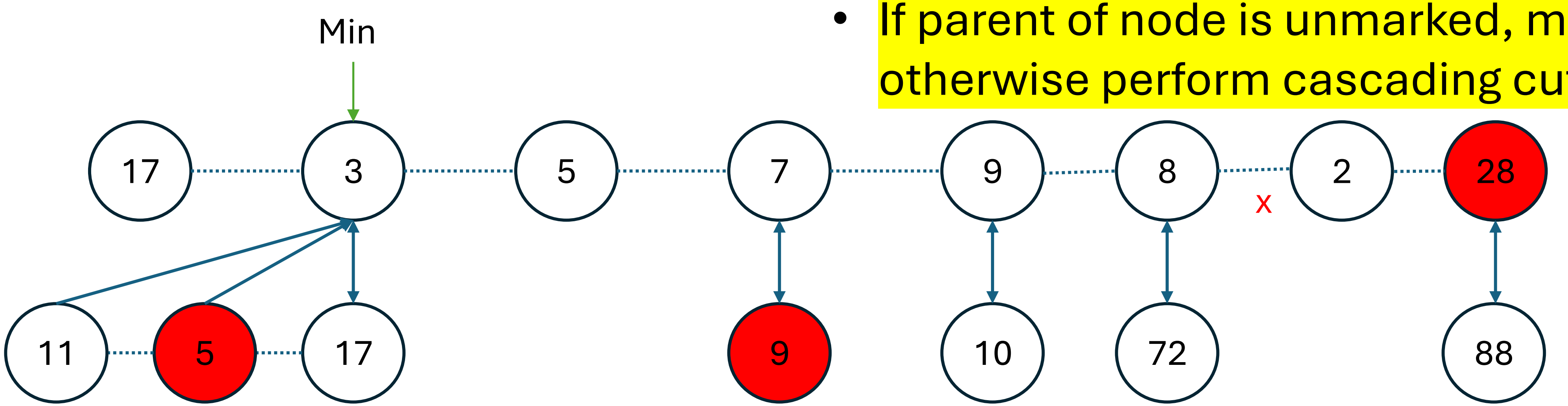
- Decrease key of node
- Cut node and put in root list
- If parent of node is unmarked, mark it, otherwise perform cascading cut



# Decrease-Key : $O(1)$

Case 2b [Heap order violated]

- Decrease key of node
- Cut node and put in root list
- If parent of node is unmarked, mark it, otherwise perform cascading cut



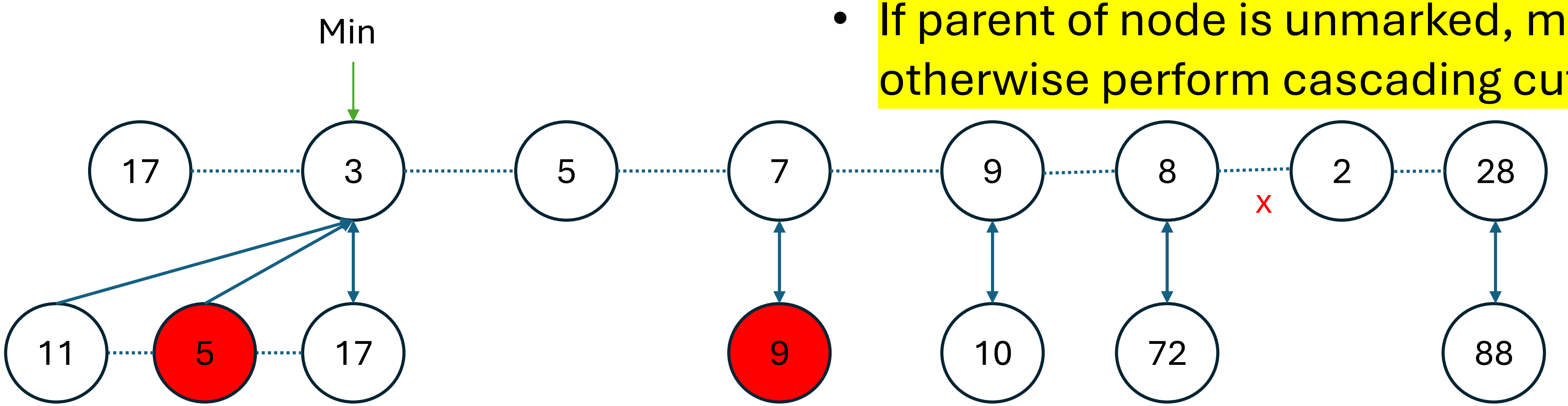

Unmark root nodes

Decrease x from 35 to 2

# Decrease-Key : $O(1)$

Case 2b [Heap order violated]

- Decrease key of node
- Cut node and put in root list
- If parent of node is unmarked, mark it, otherwise perform cascading cut



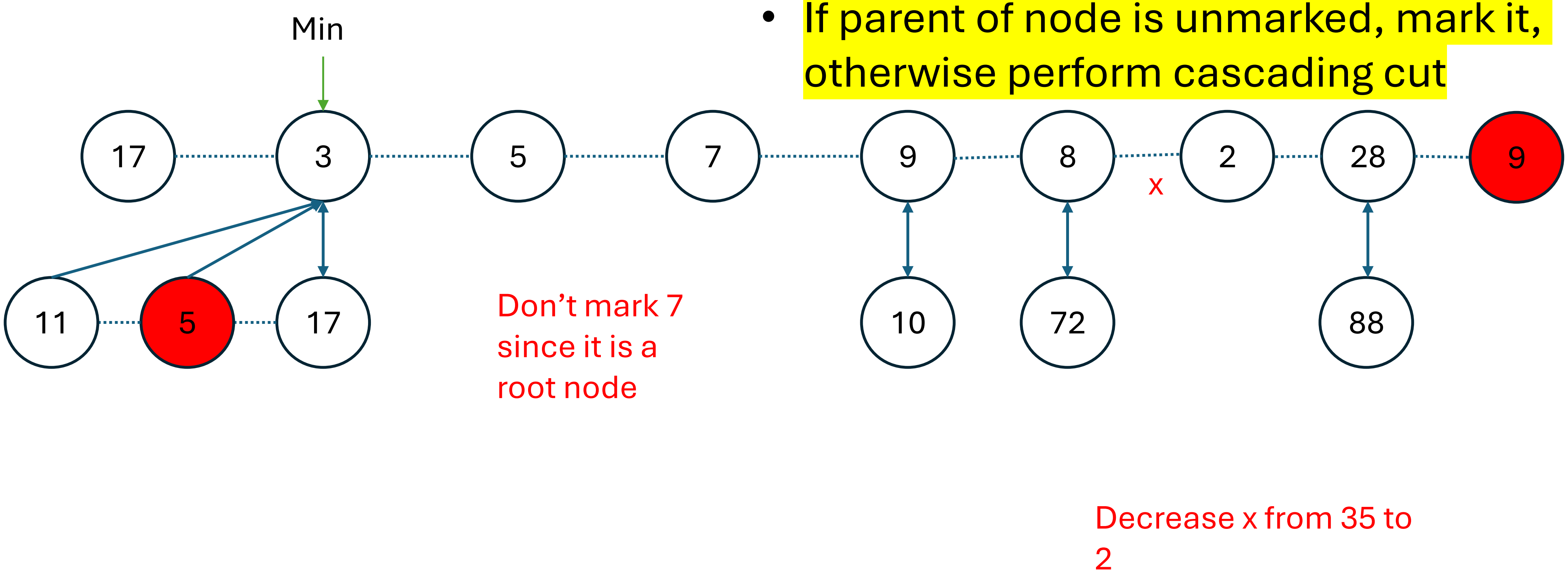

9 is marked so cut 9

Decrease x from 35 to 2

# Decrease-Key : $O(1)$

Case 2b [Heap order violated]

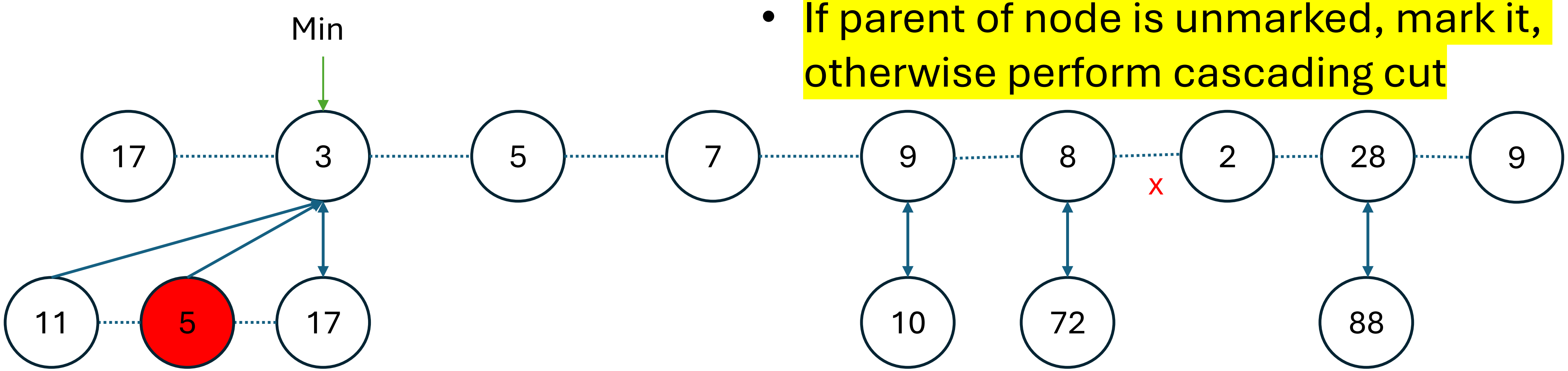
- Decrease key of node
- Cut node and put in root list
- If parent of node is unmarked, mark it, otherwise perform cascading cut



# Decrease-Key : $O(1)$

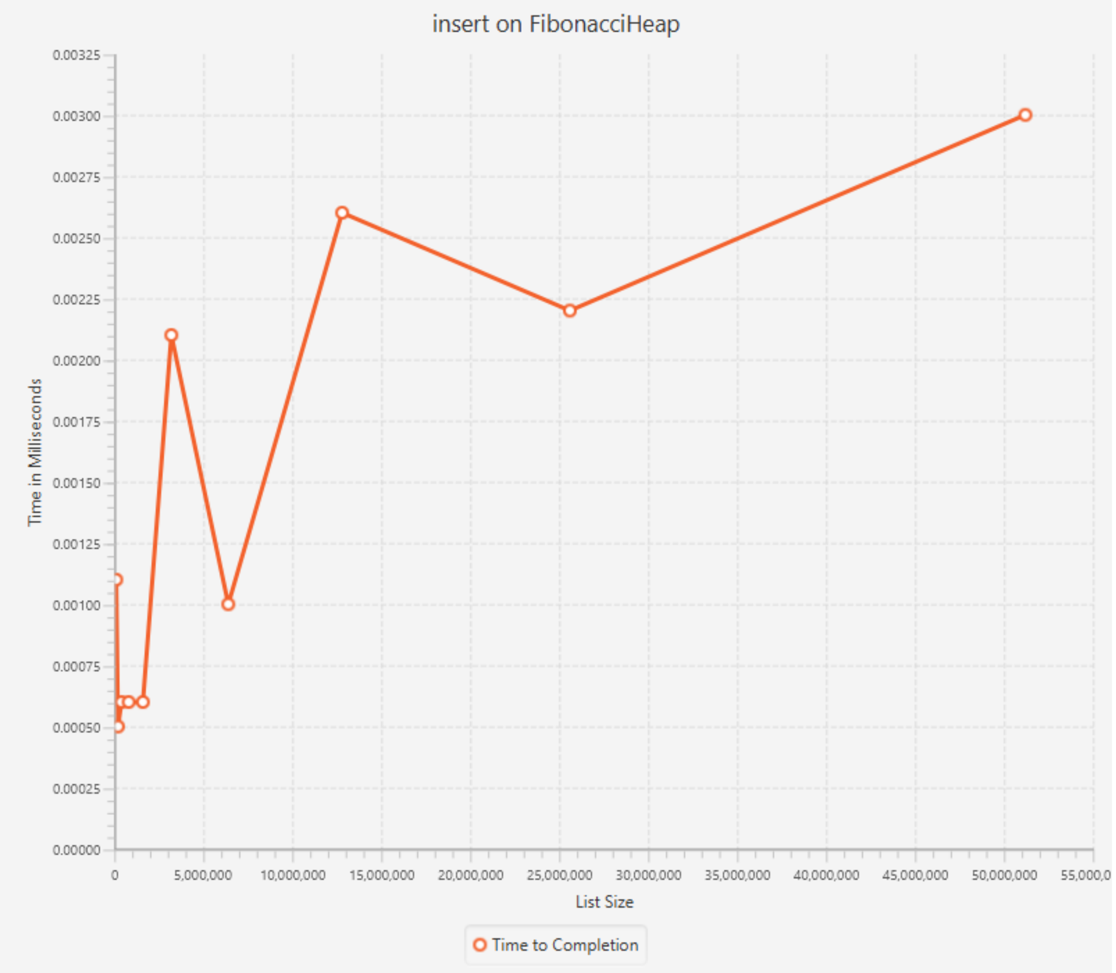
Case 2b [Heap order violated]

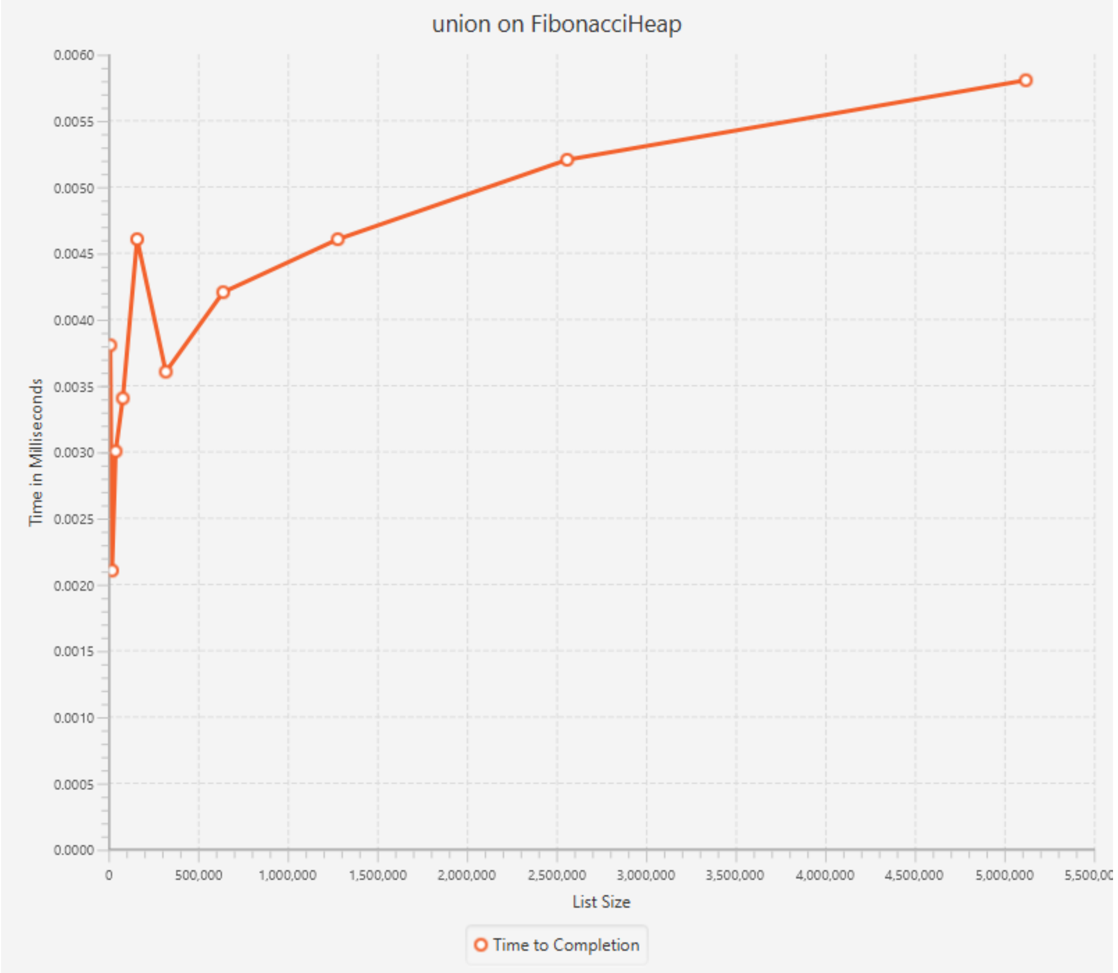
- Decrease key of node
- Cut node and put in root list
- If parent of node is unmarked, mark it, otherwise perform cascading cut

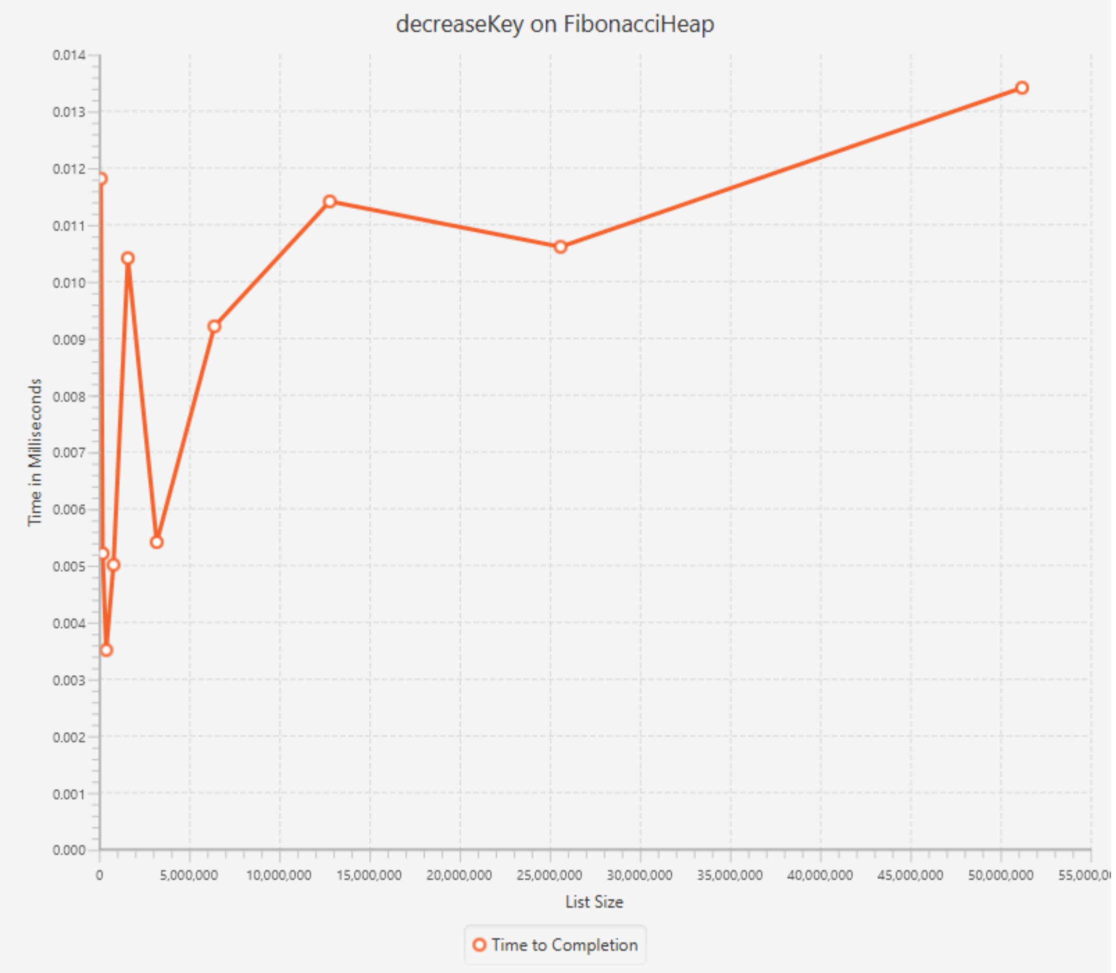


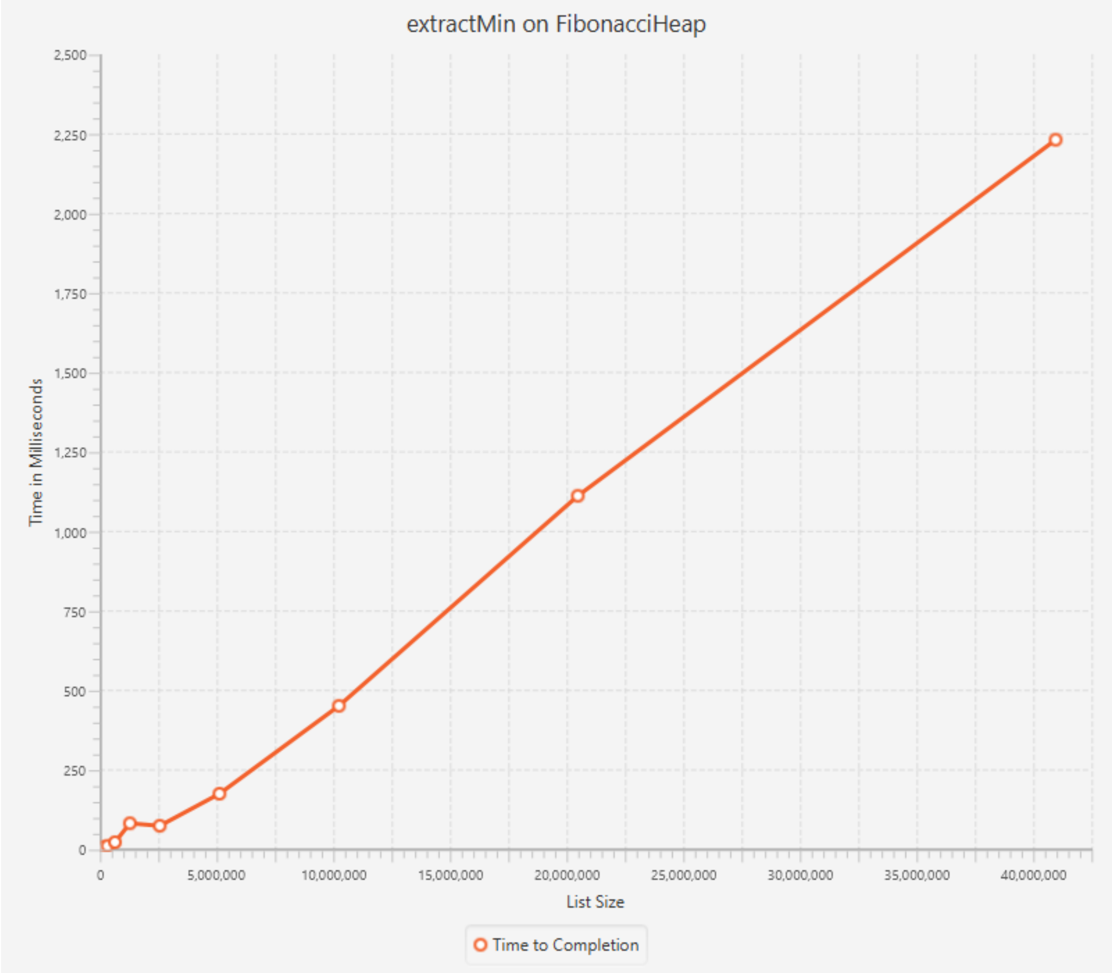

Decrease x from 35 to 2

# Demo


insert on FibonacciHeap
Time in Milliseconds
0.00325
0.00300
0.00275
0.00250
0.00225
0.00200
0.00175
0.00150
0.00125
0.00100
0.00075
0.00050
0.00025
0.00000
0
5,000,000
10,000,000
15,000,000
20,000,000
25,000,000
30,000,000
35,000,000
40,000,000
45,000,000
50,000,000
List Size
Time to Completion


union on FibonacciHeap
Time in Milliseconds
0.0060
0.0055
0.0050
0.0045
0.0040
0.0035
0.0030
0.0025
0.0020
0.0015
0.0010
0.0005
0.0000
0
500,000
1,000,000
1,500,000
2,000,000
2,500,000
3,000,000
3,500,000
4,000,000
4,500,000
5,000,000
List Size
Time to Completion


decreaseKey on FibonacciHeap
Time in Milliseconds
0.014
0.013
0.012
0.011
0.010
0.009
0.008
0.007
0.006
0.005
0.004
0.003
0.002
0.001
0.000
0
5,000,000
10,000,000
15,000,000
20,000,000
25,000,000
30,000,000
35,000,000
40,000,000
45,000,000
50,000,000
List Size
Time to Completion


extractMin on FibonacciHeap
Time in Milliseconds
2,500
2,250
2,000
1,750
1,500
1,250
1,000
750
500
250
0
0
5,000,000
10,000,000
15,000,000
20,000,000
25,000,000
30,000,000
35,000,000
40,000,000
List Size
Time to Completion

# Advantages and Disadvantages

Advantages:

- Efficient amortized time complexities for Decrease-Key, Insert, and Union
- Beneficial in graphing algorithms like Dijkstra’s and Prim’s because of many calls to Decrease-key

Disadvantages:

- Very challenging to implement and no standard implementation in Java or Python
- If an application doesn’t use decrease-key many times, binomial heaps or binary heaps may be more efficient due to reduced overhead

# Why is the maximum degree of a node $O(\log_{\phi} n)$

**1. Fibonacci Sequence and its Growth**: The Fibonacci sequence is defined as:

$$F_0 = 0, F_1 = 1, F_d = F_{d-1} + F_{d-2} \text{ for } d \geq 2$$

The Fibonacci numbers grow exponentially, and there is a closed-form expression for the Fibonacci numbers, called **Binet's formula**:

$$F_d = \frac{\phi^d - (1-\phi)^d}{\sqrt{5}} \text{ where } \phi = \frac{1+\sqrt{5}}{2} \text{ is the golden ratio}$$

For very large d, the term $(1-\phi)^d$ approaches 0, so the formula can be approximated as:

$$F_d \approx \frac{\phi^d}{\sqrt{5}}$$

# Why is the maximum degree of a node $O(\log_\phi n)$

**2. Relationship Between Degree and Fibonacci Numbers**: In a Fibonacci heap, the degree $d$ of a node is the number of children that node has. The size of the tree rooted at a node with degree $d$ is at least $F_d$, meaning that a tree with degree $d$ has at least $F_d$ nodes.

Therefore, for a Fibonacci heap with $n$ nodes, the degree $d$ of any node must satisfy:

$$F_d \leq n$$

*See Appendix B

# Why is the maximum degree of a node $O(\log_\phi n)$

**3. Finding the Maximum Degree**: To determine the maximum degree $d_{max}$ of any node in the Fibonacci heap, we want to find the largest $d$ such that $F_d \leq n$. Using the approximation $F_d \approx \frac{\phi^d}{\sqrt{5}}$, we can solve for $d$ as follows:

$$\frac{\phi^d}{\sqrt{5}} \leq n$$

Multiplying both sides by $\sqrt{5}$, we get: $\phi^d \leq n\sqrt{5}$

Taking the log of both sides, we get: $\log_\phi(\phi^d) \leq \log_\phi n\sqrt{5}$

Using properties of logs, we get: $d \log_\phi(\phi) \leq \log_\phi n + \log_\phi \sqrt{5}$

Simplifying: $$d \leq \log_\phi n + O(1)$$

Therefore the degree $d$ is bounded by: $\mathrm{d} = O(\log_\phi n)$

# How Fibonacci numbers bound node degrees

Fibonacci heaps have a special structure where the size of a subtree rooted at a node with degree $d$ is **at least** $F_d$ , the $d$-th Fibonacci number.

This property arises from the way trees are merged and the rules of consolidation in Fibonacci heaps:

- When two trees of the same degree are merged, one becomes the child of the other, and their degrees increase. (Link operation)
- The consolidation process ensures that higher-degree nodes have increasingly larger subtrees. This recursive merging leads to the **Fibonacci sequence** as the minimum number of nodes in a subtree for any given degree.

The number of nodes in the tree grows exponentially as the degree increases, following the Fibonacci sequence.

# How Fibonacci numbers bound node degrees

Now consider the entire Fibonacci heap, which contains $n$ nodes in total. No single tree in the heap can have more than $n$ nodes.

Let's denote the degree of a node in this heap by $d$. Since the subtree rooted at a node with degree $d$ must contain **at least** $F_d$ nodes, the total number of nodes in the heap $n$ provides an upper bound on $F_d$ . That is:

$$F_d \leq n$$

This inequality is the direct result of the fact that $F_d$ represents the **minimum size** of a tree of degree $d$, and the heap as a whole has at most $n$ nodes.

# How Fibonacci numbers bound node degrees

This is true because:

- Each tree in the Fibonacci heap grows according to the rules of consolidation, which guarantees that the size of a tree increases at least as fast as the Fibonacci sequence.
- The degree $d$ determines the size of the smallest possible subtree rooted at a node of that degree, which is $F_d$ .
- Since the entire heap cannot have more nodes than $n$, the largest degree $d$ must satisfy $F_d \leq n$.

# References

https://www.geeksforgeeks.org/fibonacci-heap-set-1-introduction/

https://youtu.be/UkPVvP4_OaA?si=-Jx51D9wbU0ESvbj