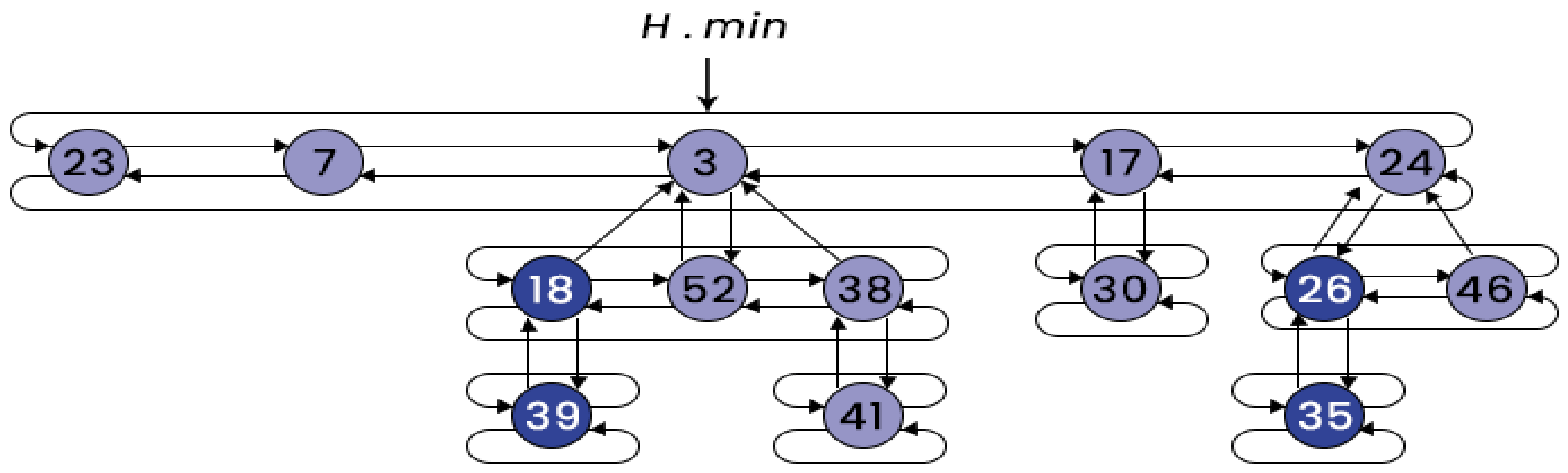
Fibonacci Heaps



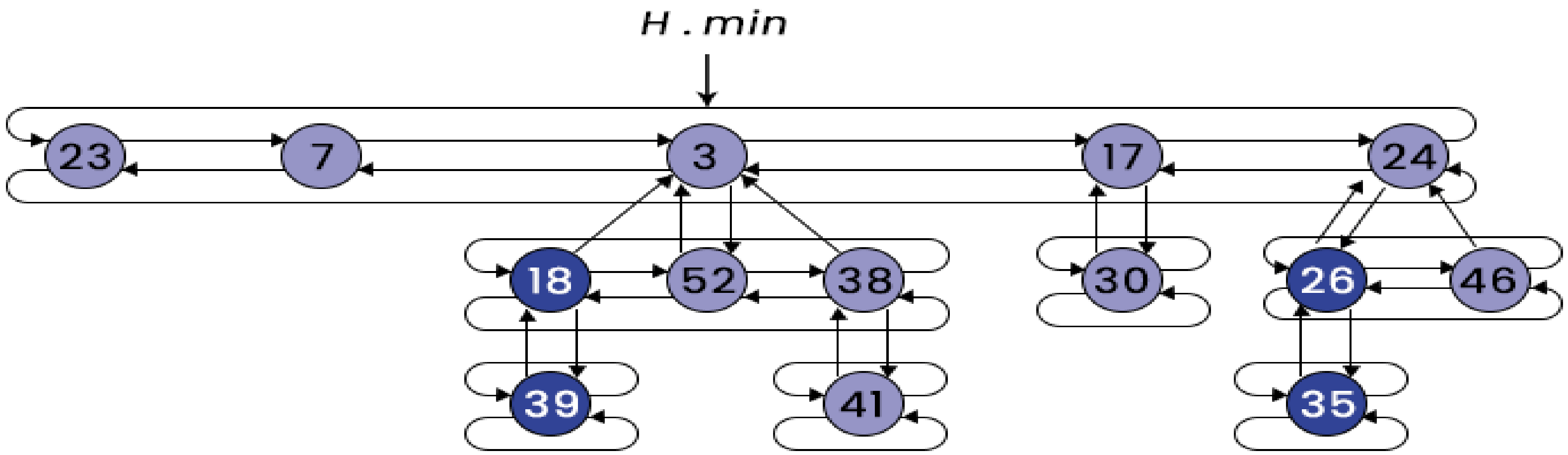
Dr. Bibhudatta Sahoo

# CS215, Department of CSE, NIT Rourkela

Email: bdsahu@nitrkl.ac.in, 9937324437, 2462358

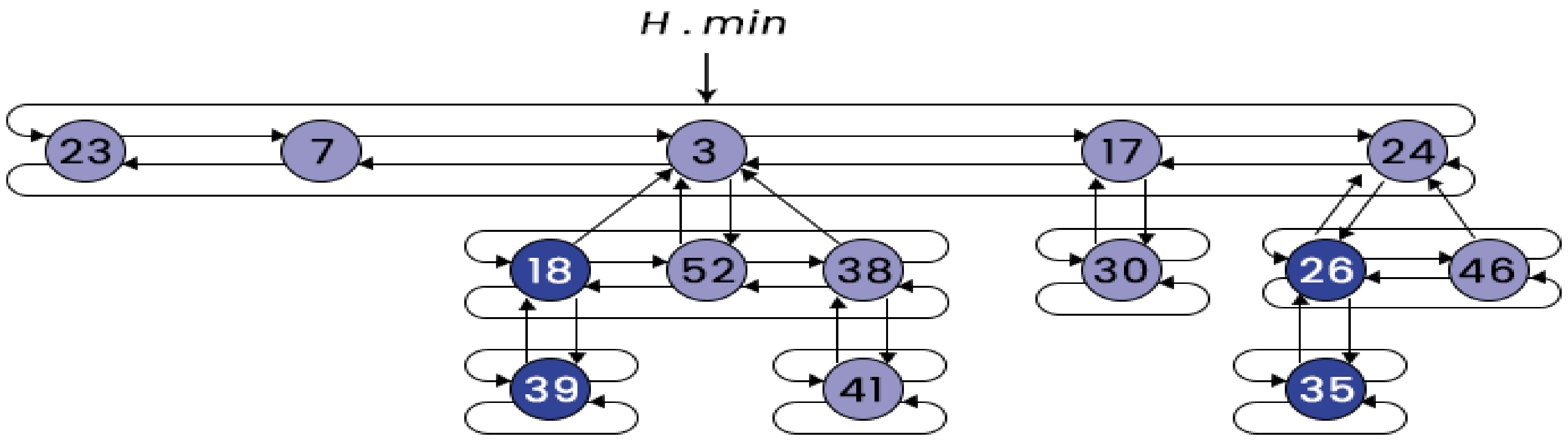
Fibonacci Heap

- A **Fibonacci heap** is a data structure for **priority queue** operations, consisting of a collection of heap-ordered trees(Min-Heap or Max-Heap).
- It has a faster **amortized running time** for many operations compared to other heap types like the binary heap or binomial heap.
- The Fibonacci heap was introduced by Michael L. Fredman and Robert E. Tarjan in 1984, and its main applications are in the fields of **network optimization** and **graph algorithms**, such as Dijkstra's shortest path algorithm.



Fibonacci Heap

- The Fibonacci heap is a type of data structure that can be used to implement a **priority queue**.
- The key feature of this type of heap is its efficient merging of elements, which allows for fast and efficient processing of elements.
- Fibonacci heaps provide an amortized O(1) time complexity for operations such as insertions, deletions, and finding the minimum element, making them highly efficient in comparison to other types of data structures for priority queues.



Comparison between Fibonacci heaps and Binomial heaps

1. Structure

Fibonacci Heap:

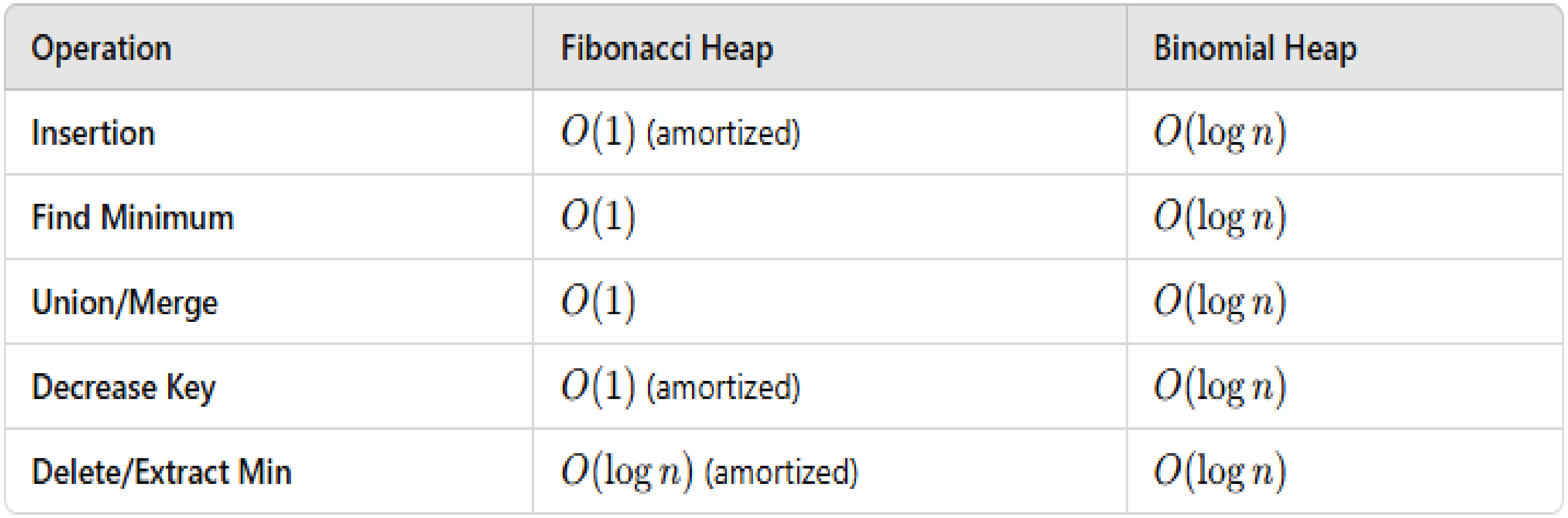
1. Made up of a collection of trees, each of which is a minimum-heap.
2. The trees in a Fibonacci heap are not required to be binomial trees, and they have a more relaxed structure, allowing nodes to have any number of children.
3. Fibonacci heaps also maintain a "marked" property for nodes that helps in reducing the time complexity of certain operations.

Binomial Heap:

1. Binomial Heap consists of a collection of trees, but each tree is a binomial tree, and the heap is structured as a collection of binomial trees of varying orders (powers of two).
2. Each tree in a binomial heap is ordered, with the smallest element at the root, following a strict structure that resembles binary counting.

Comparison between Fibonacci heaps and Binomial heaps

2. Key Operations & Time Complexity



Operation	Fibonacci Heap	Binomial Heap
Insertion	$O(1)$ (amortized)	$O(\log n)$
Find Minimum	$O(1)$	$O(\log n)$
Union/Merge	$O(1)$	$O(\log n)$
Decrease Key	$O(1)$ (amortized)	$O(\log n)$
Delete/Extract Min	$O(\log n)$ (amortized)	$O(\log n)$

Comparison between Fibonacci heaps and Binomial heaps

3. Amortized vs. Worst-Case Performance

- **Fibonacci Heap**: Uses amortized analysis, which means that some operations, while not always constant time, average out to O(1) across multiple operations. This is achieved through the relaxed structure, which allows faster insertion and decrease-key operations.
- **Binomial Heap**: Operations have worst-case time complexities because it maintains stricter structural rules. Therefore, every operation has a guaranteed upper time limit without amortization.

4. Use Cases & Efficiency

- **Fibonacci Heap**: More efficient for applications where a large number of **decrease-key** and **merge** operations are expected, such as Dijkstra's shortest path algorithm or Prim's minimum spanning tree algorithm, due to its O(1) amortized decrease-key performance.
- **Binomial Heap**: Generally simpler and better when decrease-key operations are infrequent, or for use in applications that prefer strict structure over amortized performance, such as simpler implementations of priority queues where worst-case guarantees are preferable.

Comparison between Fibonacci Heaps and Binomial Heaps

5. Practicality & Complexity

- **Fibonacci Heap**: Complex to implement due to the amortized analysis and marking rules but theoretically very efficient.
- **Binomial Heap**: Easier to implement and manage due to its strict structure, and often preferred for scenarios requiring predictable time bounds.

Key Characteristics of Fibonacci Heaps

1. **Tree Structure:** A Fibonacci Heap is a collection of min-heap-ordered trees, meaning each tree satisfies the min-heap property (each parent node has a key smaller than or equal to its children's keys).
2. **Minimum Node:** The heap maintains a pointer to the node with the minimum key, which allows for constant-time retrieval of the minimum element.
3. **Lazy Merging:** Trees are combined lazily, meaning trees are not immediately restructured after each operation. Instead, restructuring happens only when necessary (like during delete_min), which allows for more efficient amortized operation times.
4. **Marked Nodes:** Nodes keep track of their "marked" state. If a child node loses two or more of its children, it is cut and promoted to the root list. This marking process is part of what helps Fibonacci Heaps achieve their amortized time complexity.

Properties of the Fibonacci Heap

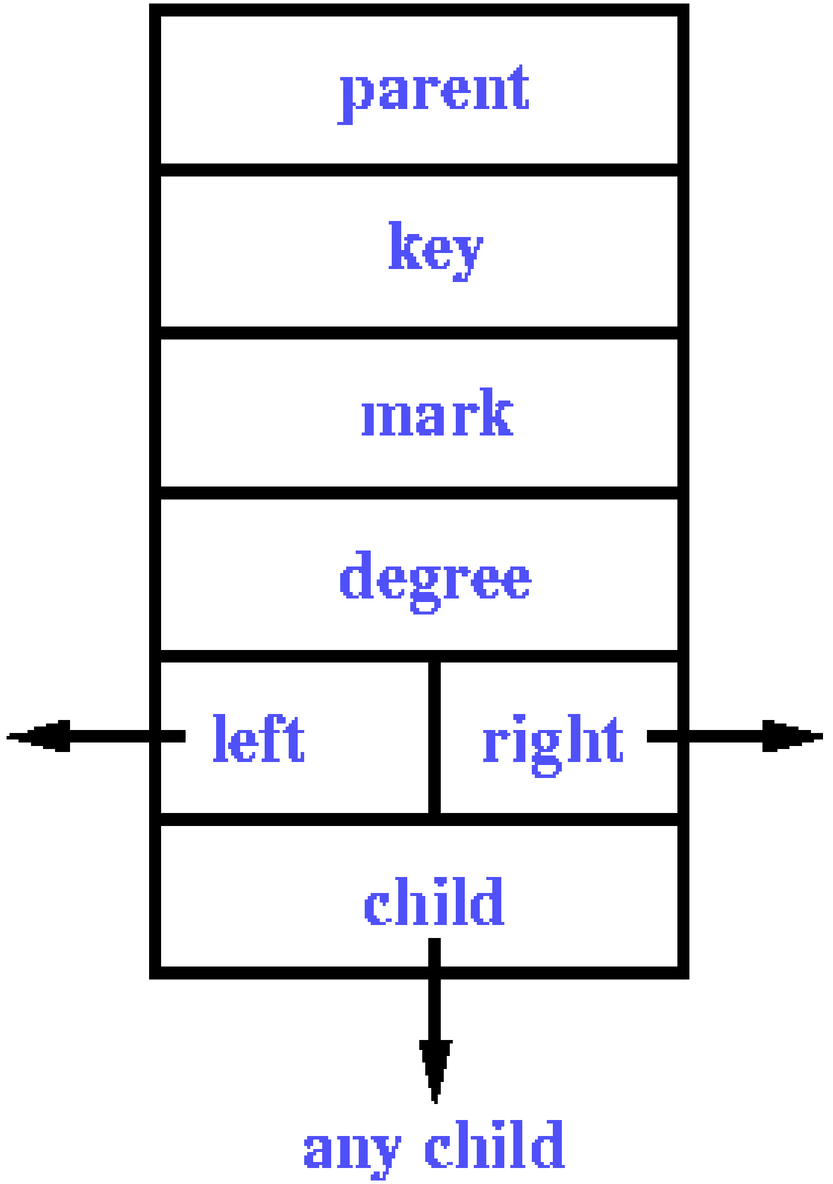
1. **Min-heap property:** Each node has a key that is less than or equal to its children.
2. **Structure:** A collection of trees that are min heaps, where a tree is either a single node or a collection of min heaps that are all rooted at a single node.
3. **Amortized time:** O(1) time for inserting a node and O(log n) time for deleting the minimum node.
4. **Linking:** When two trees of the same rank are combined, they are linked by making one tree a subtree of the other.
5. **Consolidation:** The process of combining trees of the same rank during a delete operation.
6. **Rank:** Number of children a node has.
7. **Marking:** A node can be marked if it has lost a child since the last time it was made the child of another node.

Memory Representation of the Nodes

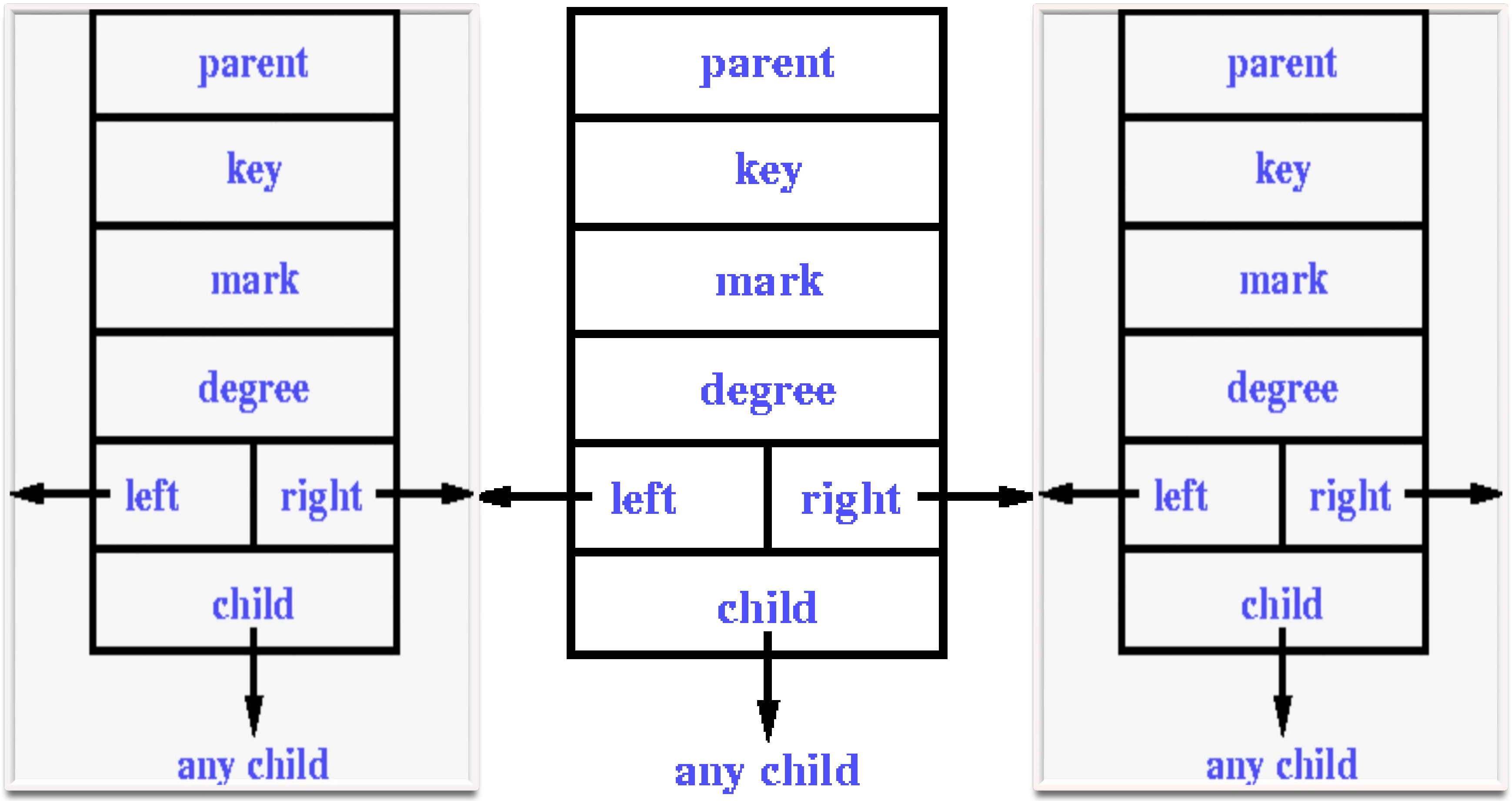
- In a Fibonacci heap, each node in the heap contains the following fields:
- **Key:** the value of the node.
- **Degree:** the number of children the node has.
- **Mark:** a flag that indicates whether the node has lost a child since it was added to the root list.
- **Parent:** a pointer to the parent node.
- **Child:** a pointer to the first child node.
- **Left:** a pointer to the node to the left of the current node in the doubly linked list.
- **Right:** a pointer to the node to the right of the current node in the doubly linked list.

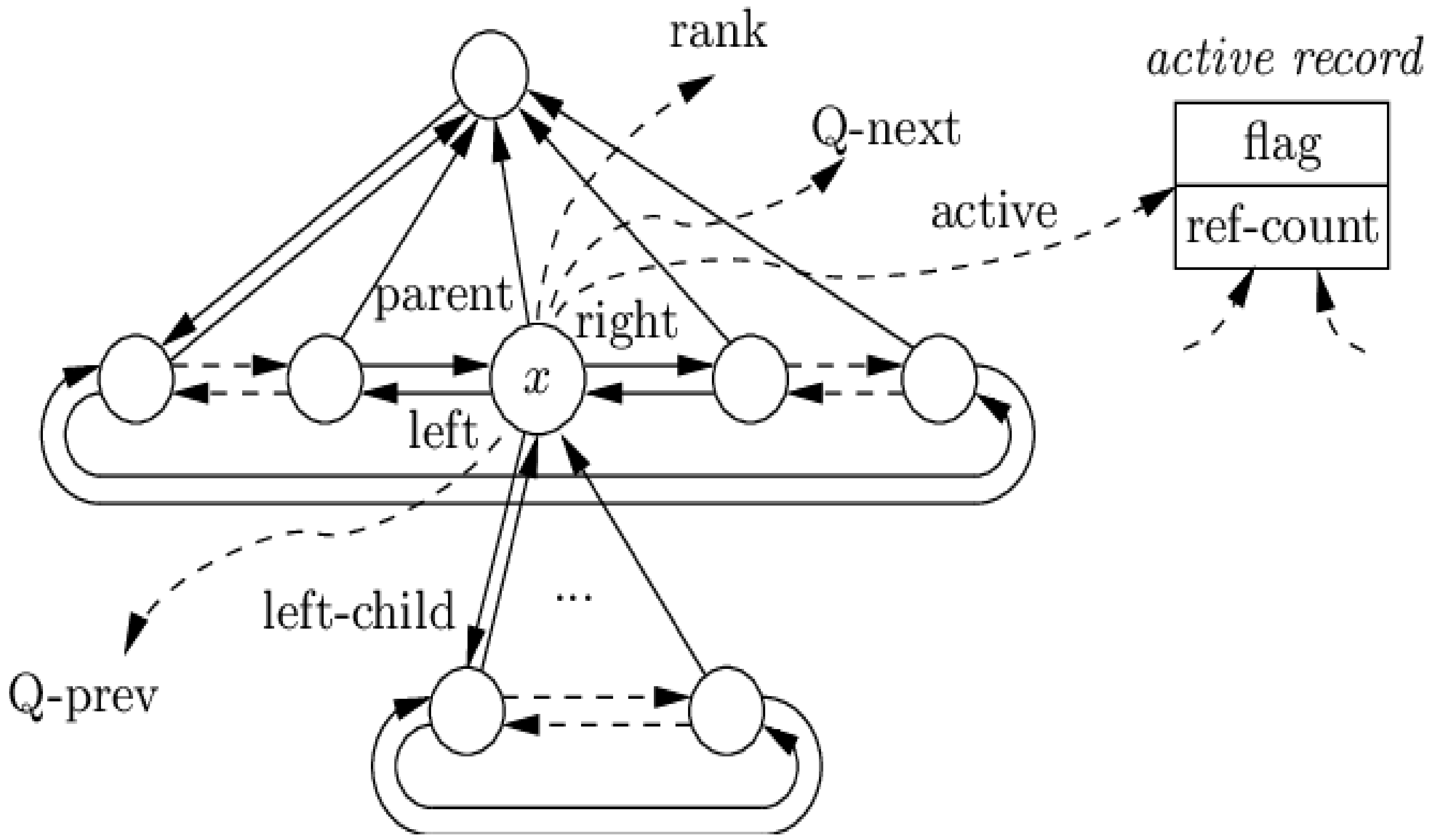
Memory Representation of the Nodes

- The field ``mark'' is True if the node has lost a child since the node became a child of another node.
- The field ``degree'' contains the number of children of this node.
- The structure contains a doubly-linked list of sibling nodes.



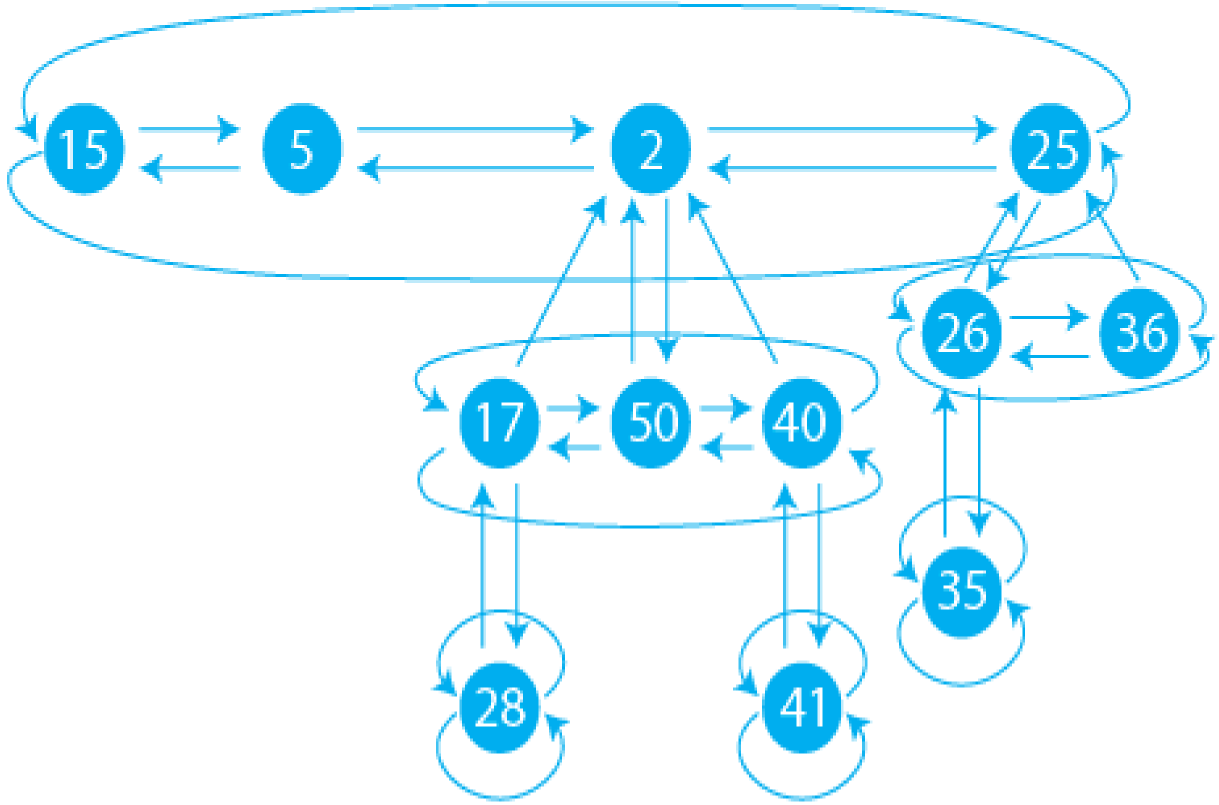
Memory Representation of the Nodes




rank
active record
flag
ref-count
Q-next
active
parent
right
x
left
left-child
...
Q-prev

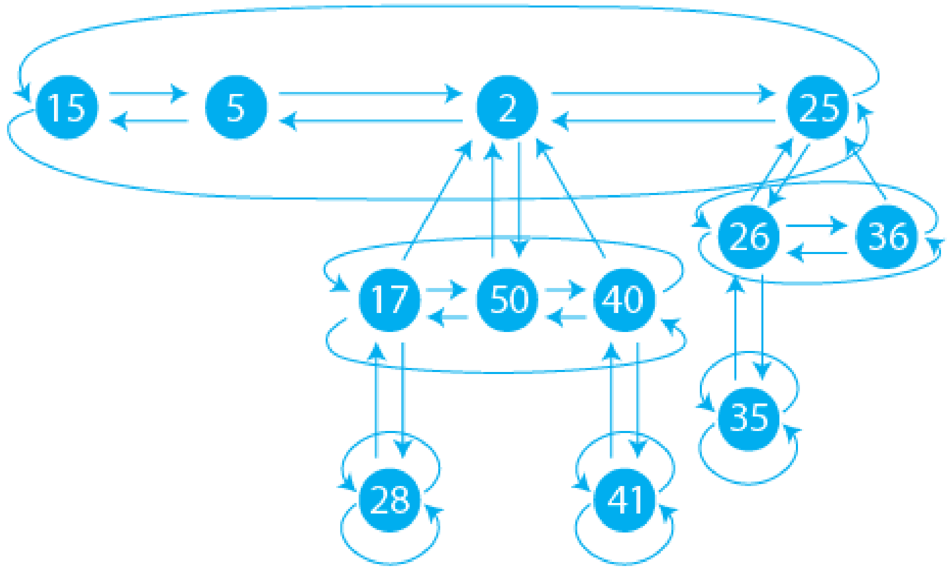
Fibonacci Heaps

The nodes in a Fibonacci heap are organized as a collection of trees, with each tree having a root node that is linked in a **circular doubly linked list** with the other roots there are two main advantages of using a circular doubly linked list.

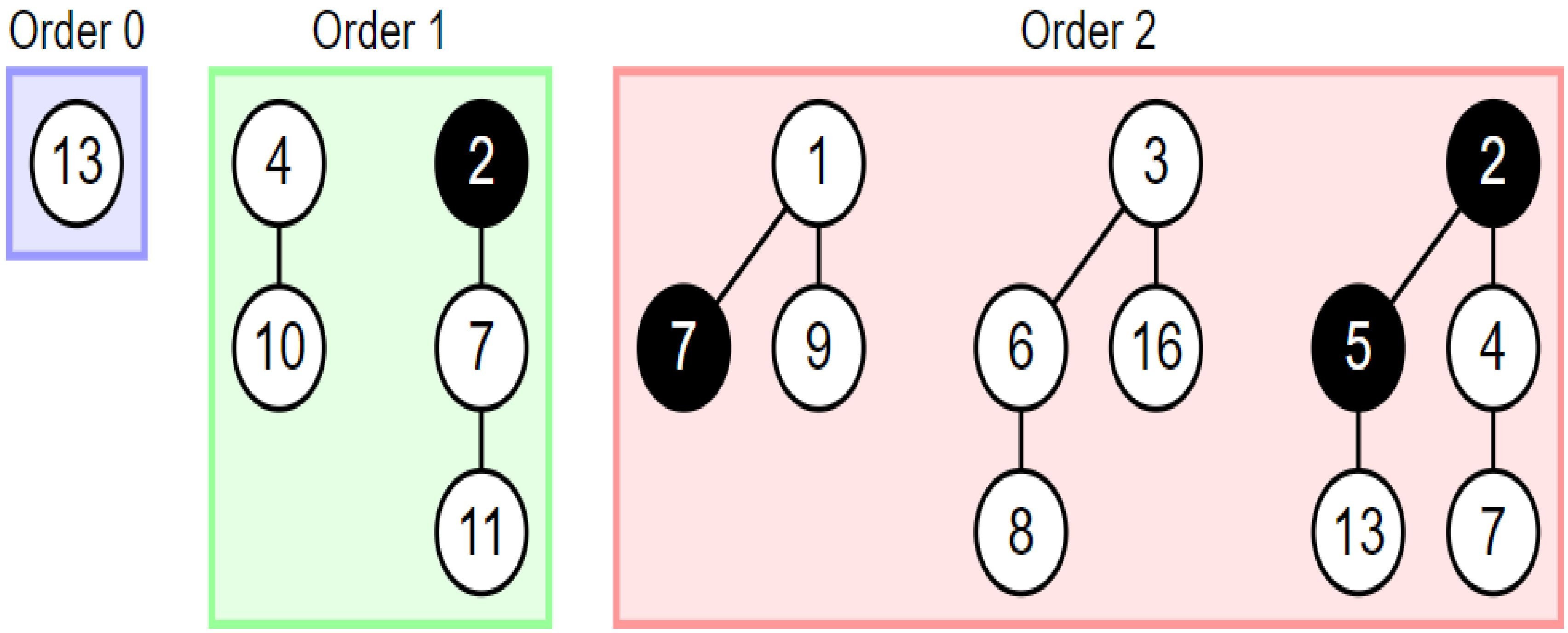


Structure Example

- A Fibonacci Heap may look something like this:
- The root list contains multiple trees, each having a minimum node.
- Each node has a pointer to its parent and children, creating a tree structure.
- Nodes in each tree satisfy the min-heap property.
- Marked nodes indicate those that have lost children, and if they lose a second child, they are moved to the root list.



Trees of order 0, 1 and 2 (the black nodes are 'marked')



Time complexity

Operation	Description	Complexity
Decrease key	Decreases an existing key to some value	$\Theta(1)$*
Delete	Deletes a node given a reference to it	$O(\log n)$*
Extract minimum	Removes and returns the minimum value given a reference to it	$O(\log n)$*
Find minimum	Returns the minimum value	$\Theta(1)$
Insert	Inserts a new value	$\Theta(1)$
Union	Combine the heap with another to form a valid Fibonacci heap	$\Theta(1)$

* *Amortised*

Properties:

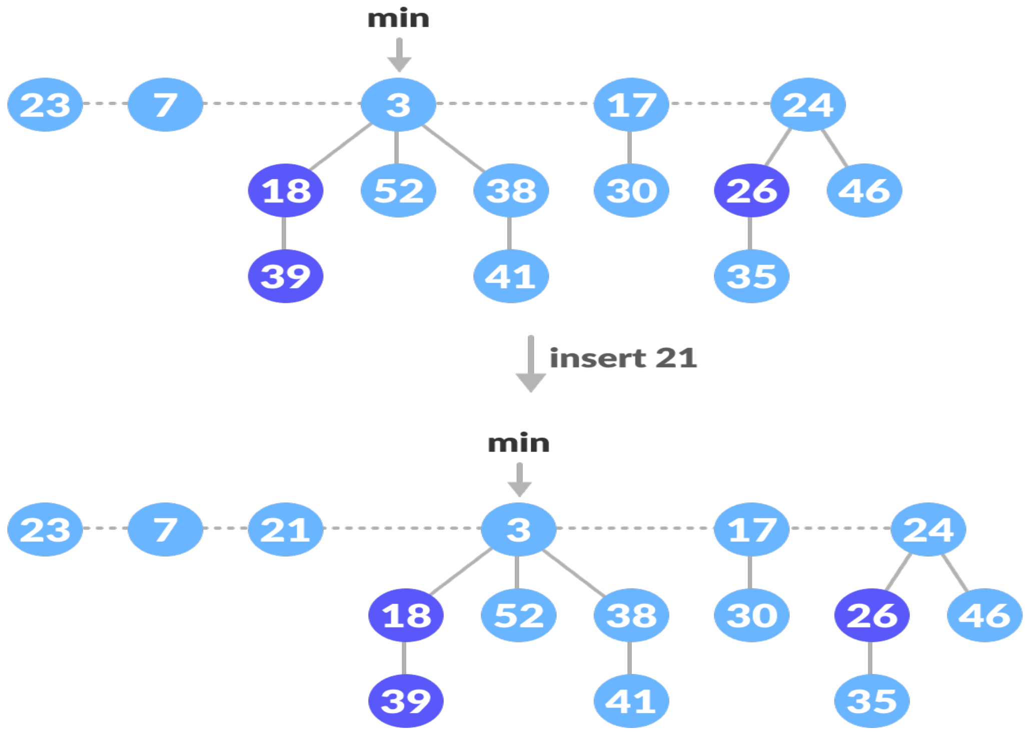
- **Lazy Approach**: Many operations are delayed as long as possible. For example, node removal and consolidation of trees of the same rank (degree).
- **Collection of Trees**: A Fibonacci heap is a collection of trees that are rooted but not necessarily binomial trees. These trees adhere to the min-heap property.
- **Minimum Node Pointer**: A pointer to the node with the minimum key value is maintained explicitly.
- **Marked Nodes**: Nodes can be "marked" as an indication that they have lost a child since the last time they were made the child of another node.

Operations and their Amortized Time Complexities:

1. **Insertion** (insert): $O(1)$ - A new tree of one node is created and added to the root list.
2. **Find Minimum** (find-min): $O(1)$ - Directly obtained from the minimum pointer.
3. **Union** (union): $O(1)$ - Joins two Fibonacci heaps into a single heap by concatenating their root lists.
4. **Extract Minimum** (extract-min): $O(\log n)$ amortized - This involves removing the min node, adding its children to the root list, and then consolidating trees of the same rank.
5. **Decrease Key** (decrease-key): $O(1)$ amortized - This operation can cause a cascading cut in the heap to maintain the heap property.
6. **Delete** (delete): $O(\log n)$ amortized - This is achieved by decreasing the key of the node to negative infinity and then extracting the minimum.

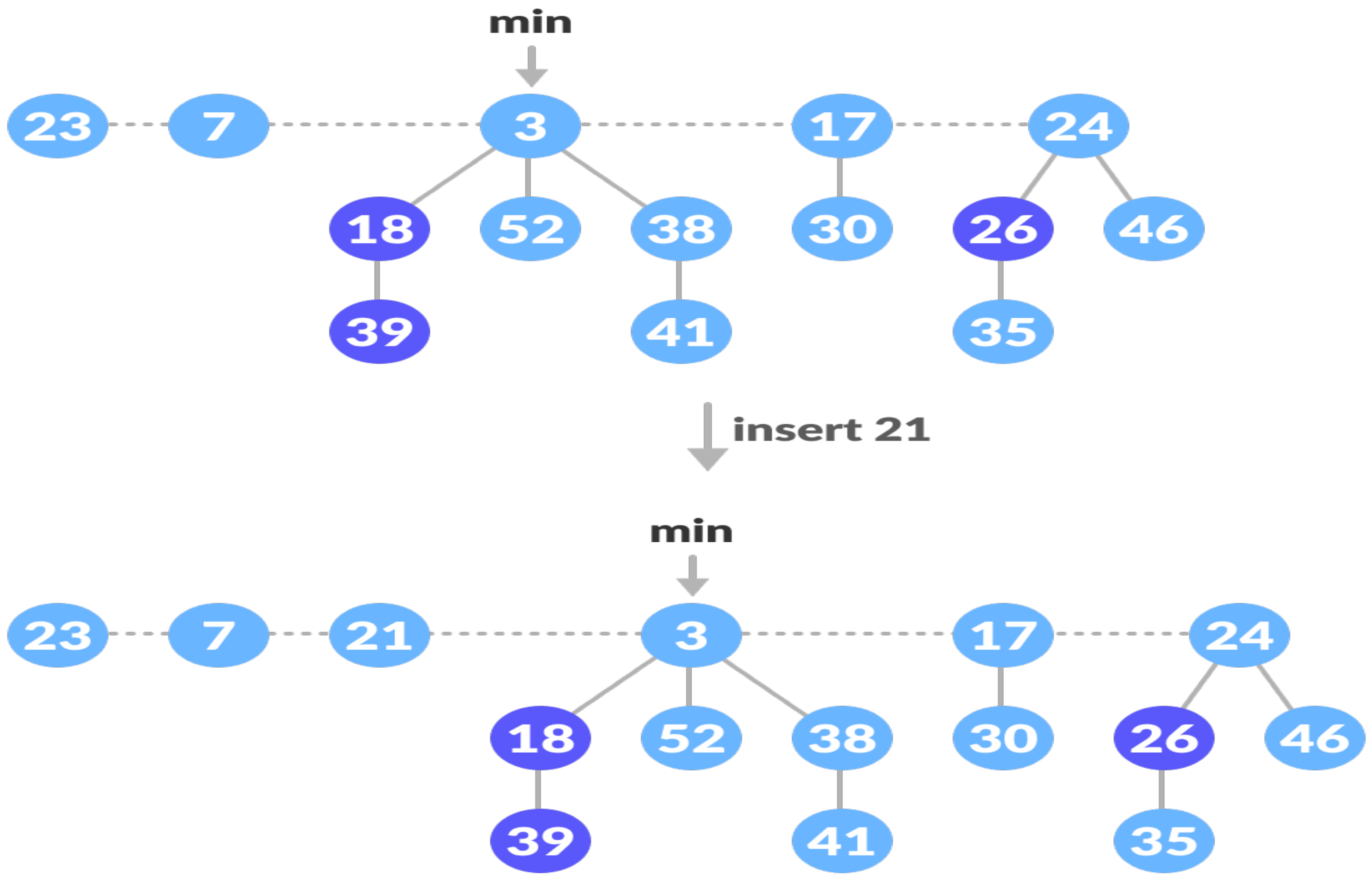
Inserting a node into an already existing heap

1. Create a new node for the element.
2. Check if the heap is empty.
3. If the heap is empty, set the new node as a root node and mark it **min**.
4. Else, insert the node into the root list and update **min**.



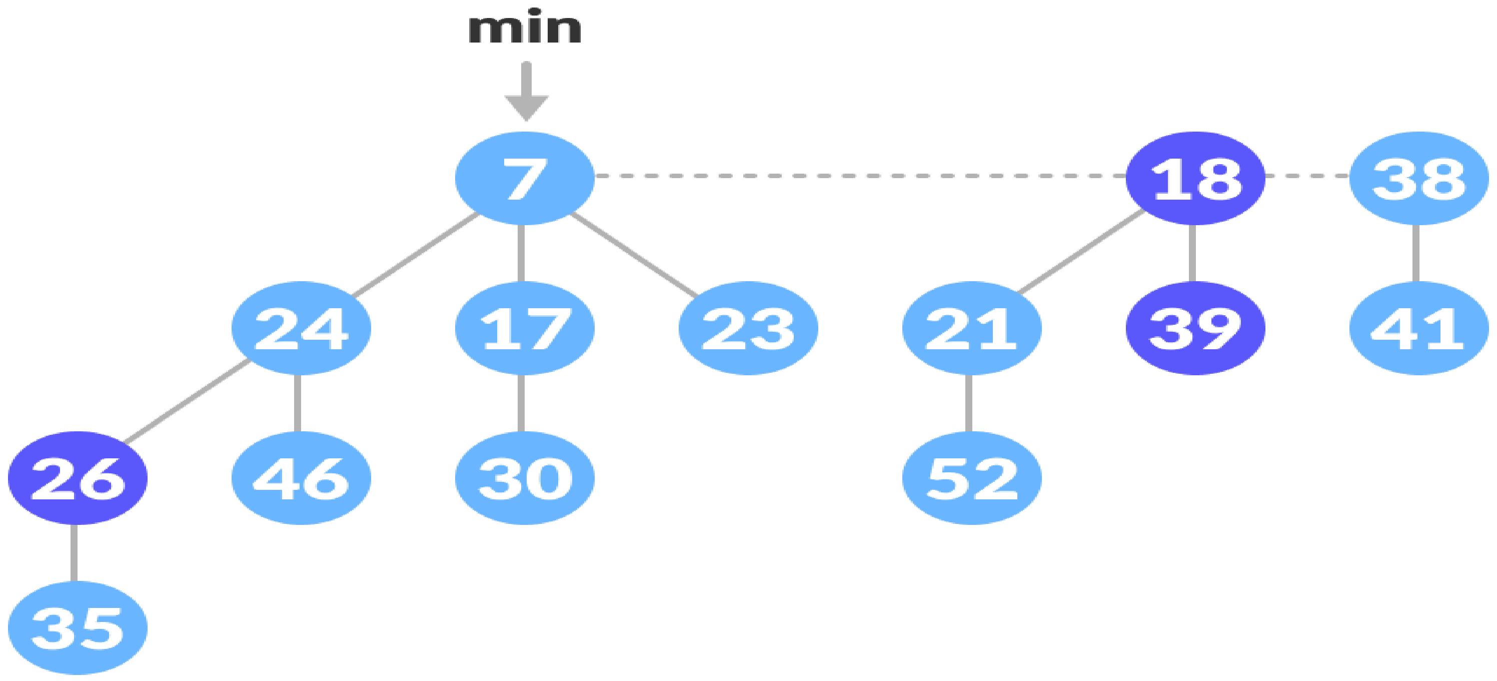
Inserting a node into an already existing heap

- The minimum element is always given by the **min** pointer.



Find Minimum

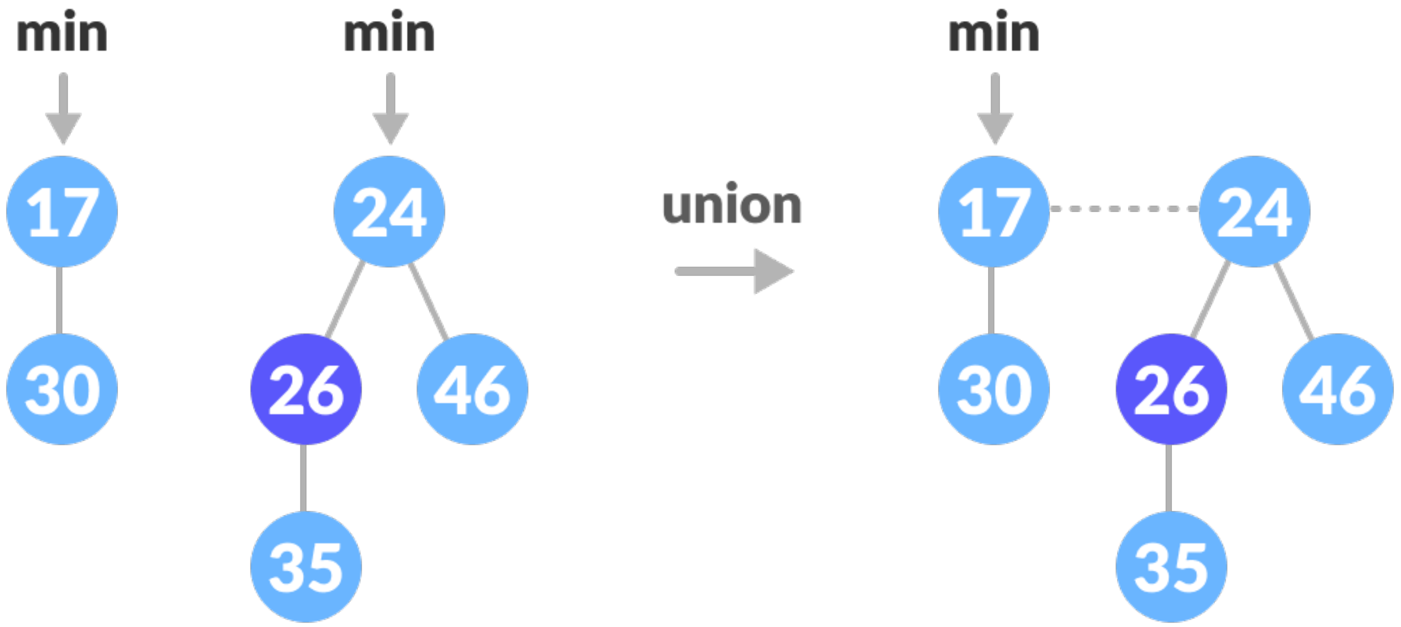
- **Find Minimum** (find-min): $O(1)$ - Directly obtained from the minimum pointer.



Union of two fibonacci heaps

- Union of two fibonacci heaps consists of following steps:

1. Concatenate the roots of both the heaps.
2. Update **min** by selecting a minimum key from the new root lists.



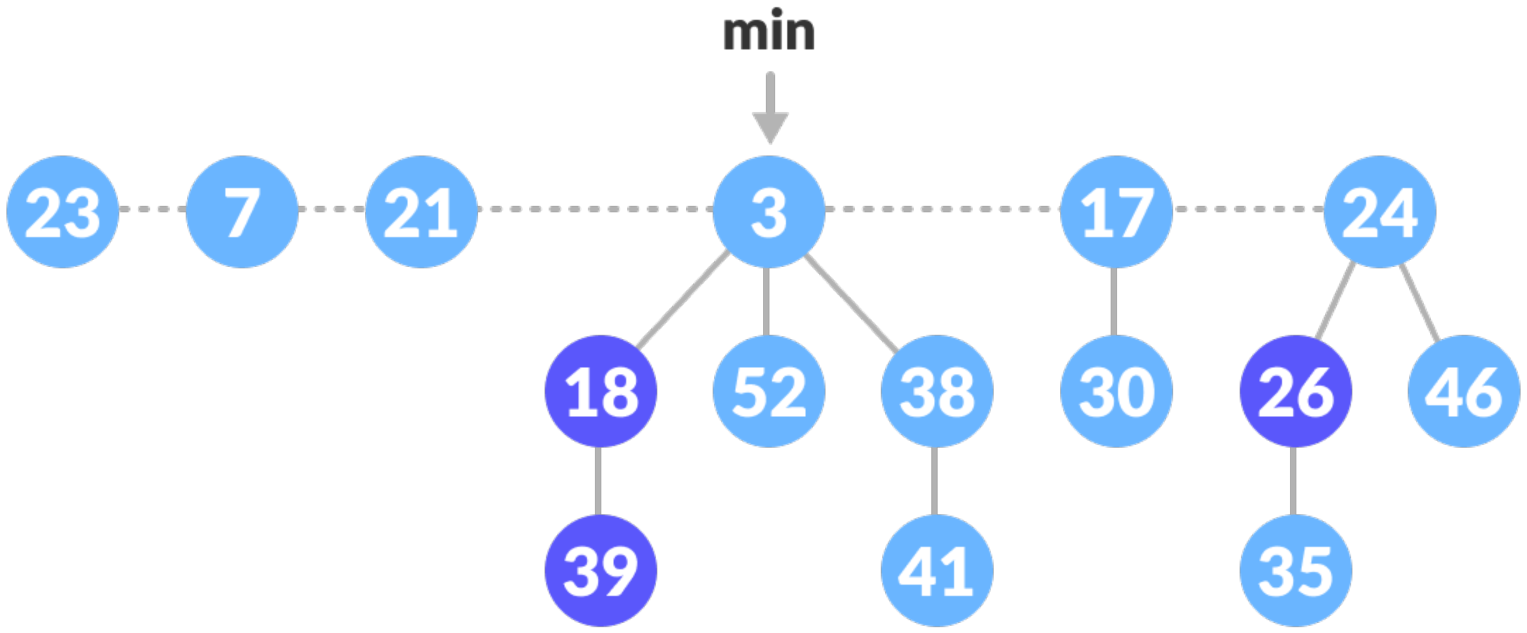
Extract-min

In this operation on a fibonacci heap, the node with minimum value is removed from the heap and the tree is re-adjusted.

Basic Steps

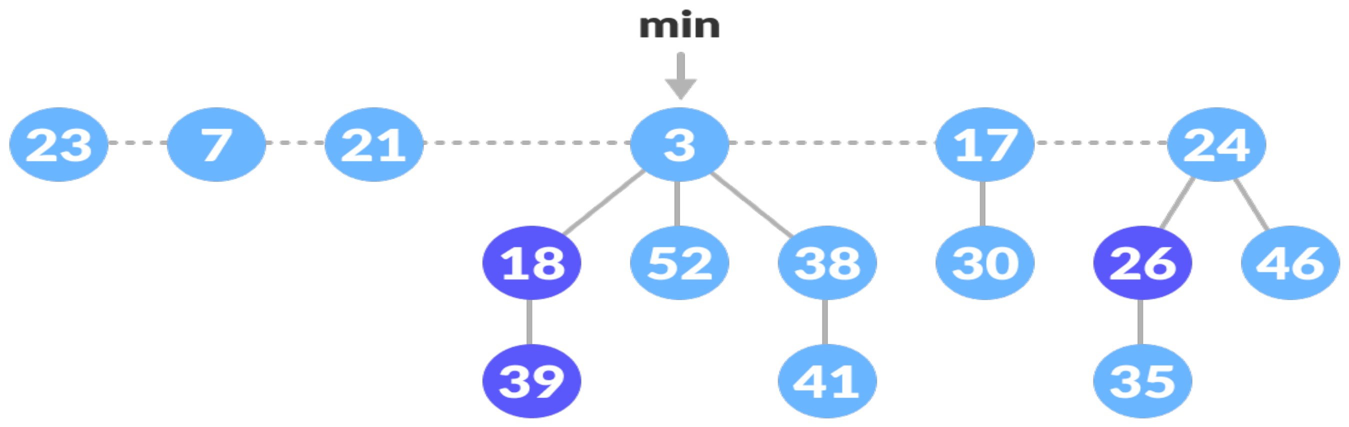
1. Delete the min node.
2. Set the min-pointer to the next root in the root list.
3. Create an array of size equal to the maximum degree of the trees in the heap before deletion.
4. Do the following (steps 5-7) until there are no multiple roots with the same degree.
5. Map the degree of current root (min-pointer) to the degree in the array.
6. Map the degree of next root to the degree in array.
7. If there are more than two mappings for the same degree, then apply union operation to those roots such that the min-heap property is maintained (i.e. the minimum is at the root).

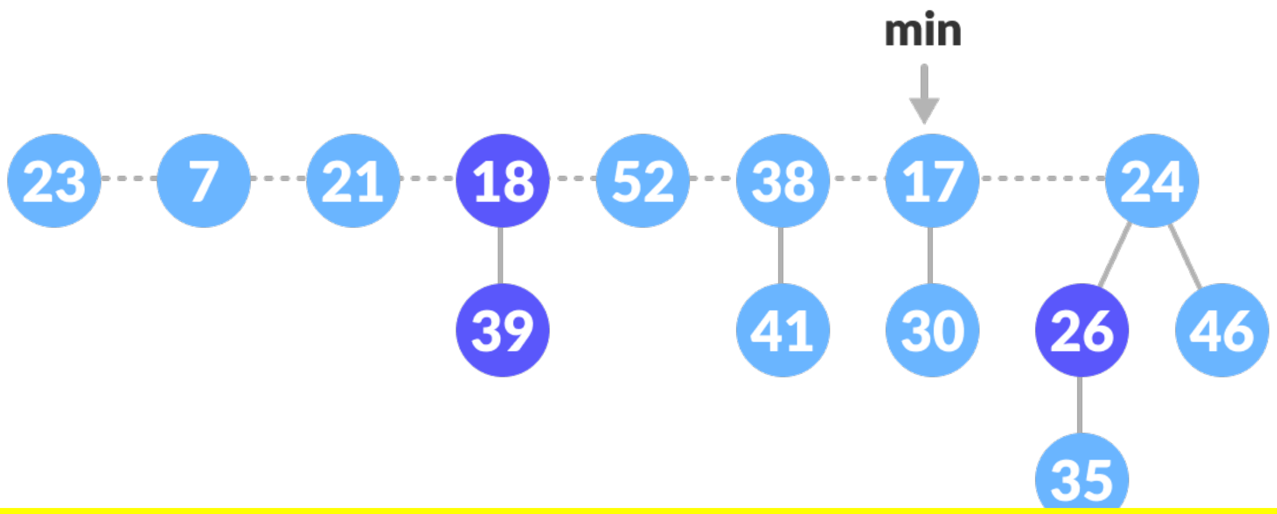
Extract-min operation on the heap



- Delete the **min node**, add all its child nodes to the root list and set the min-pointer to the next root in the root list.

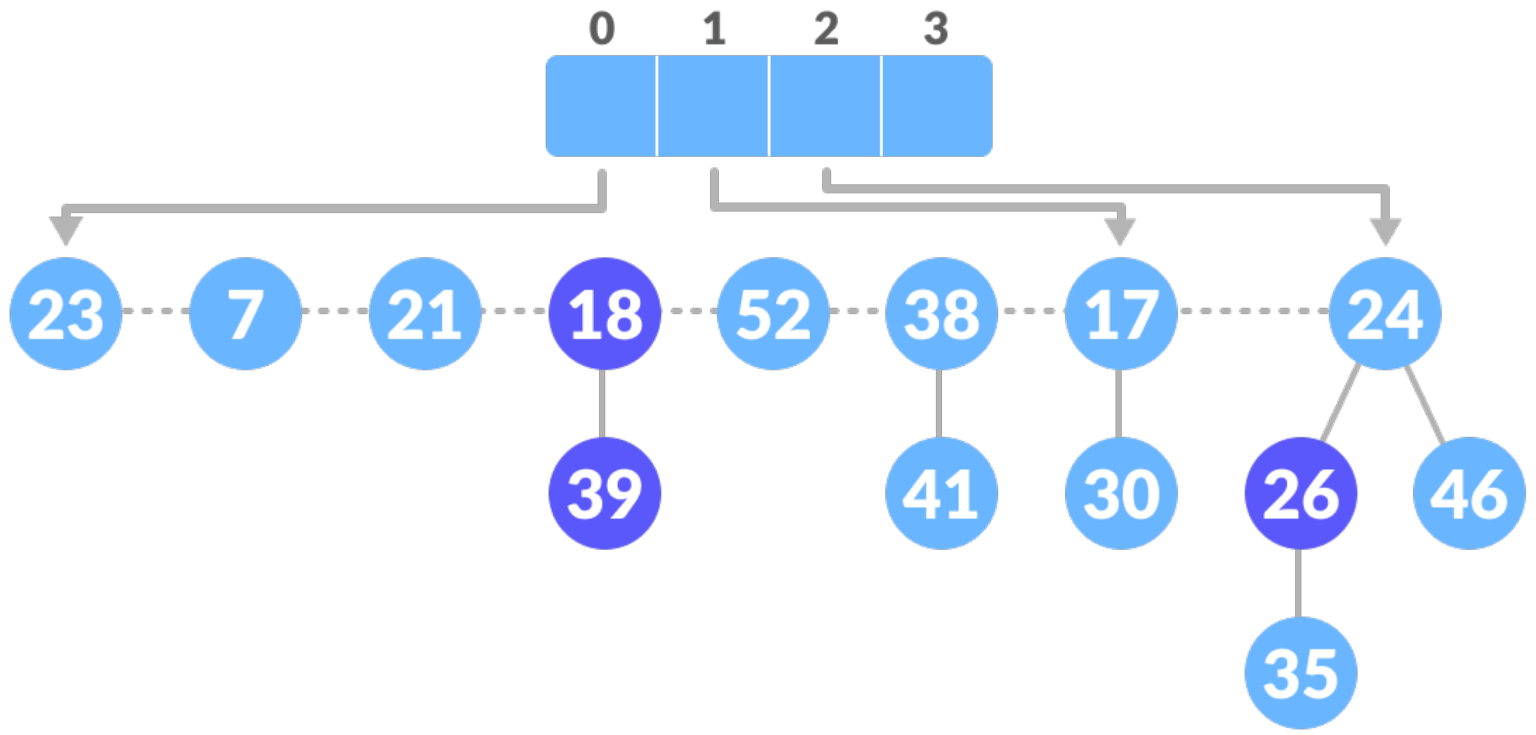
Extract-min operation on the heap





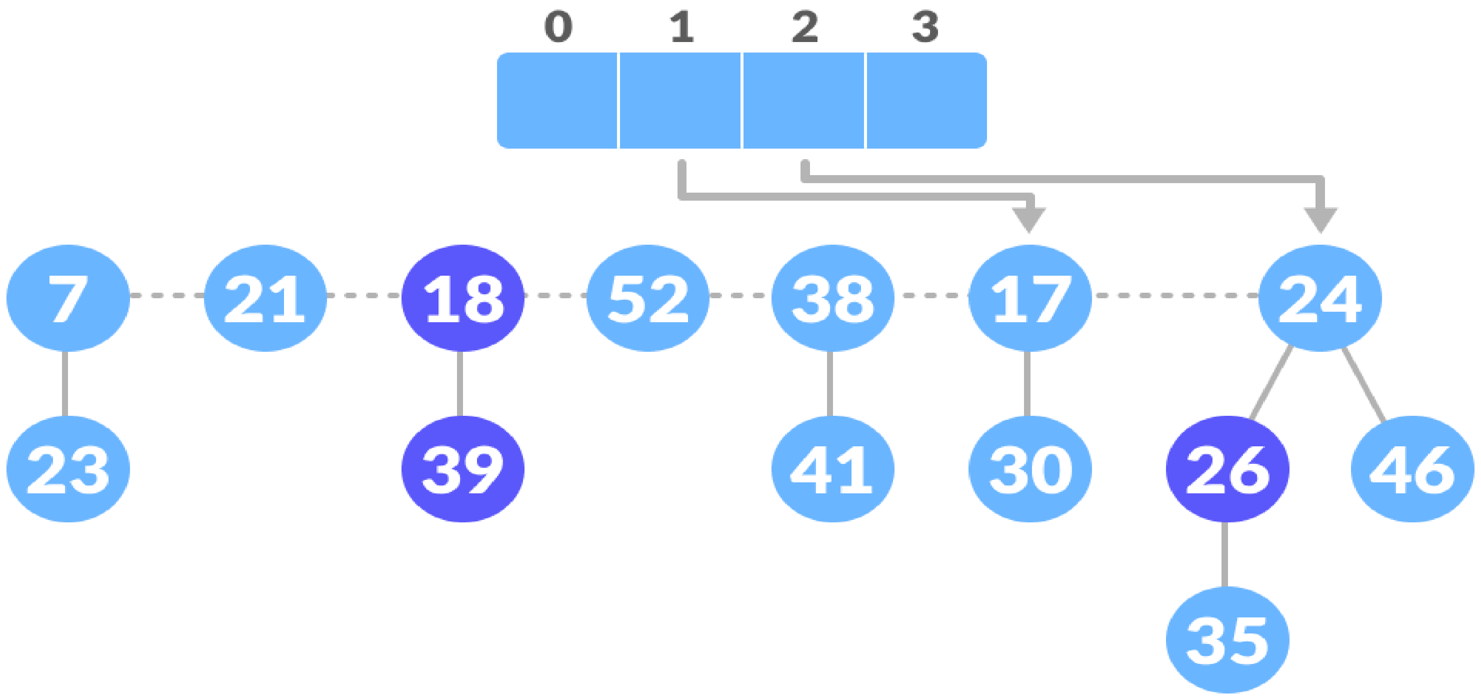
The maximum degree in the tree is 3. Create an array of size 4 and map degree of the next roots with the array.

Extract-min operation on the heap



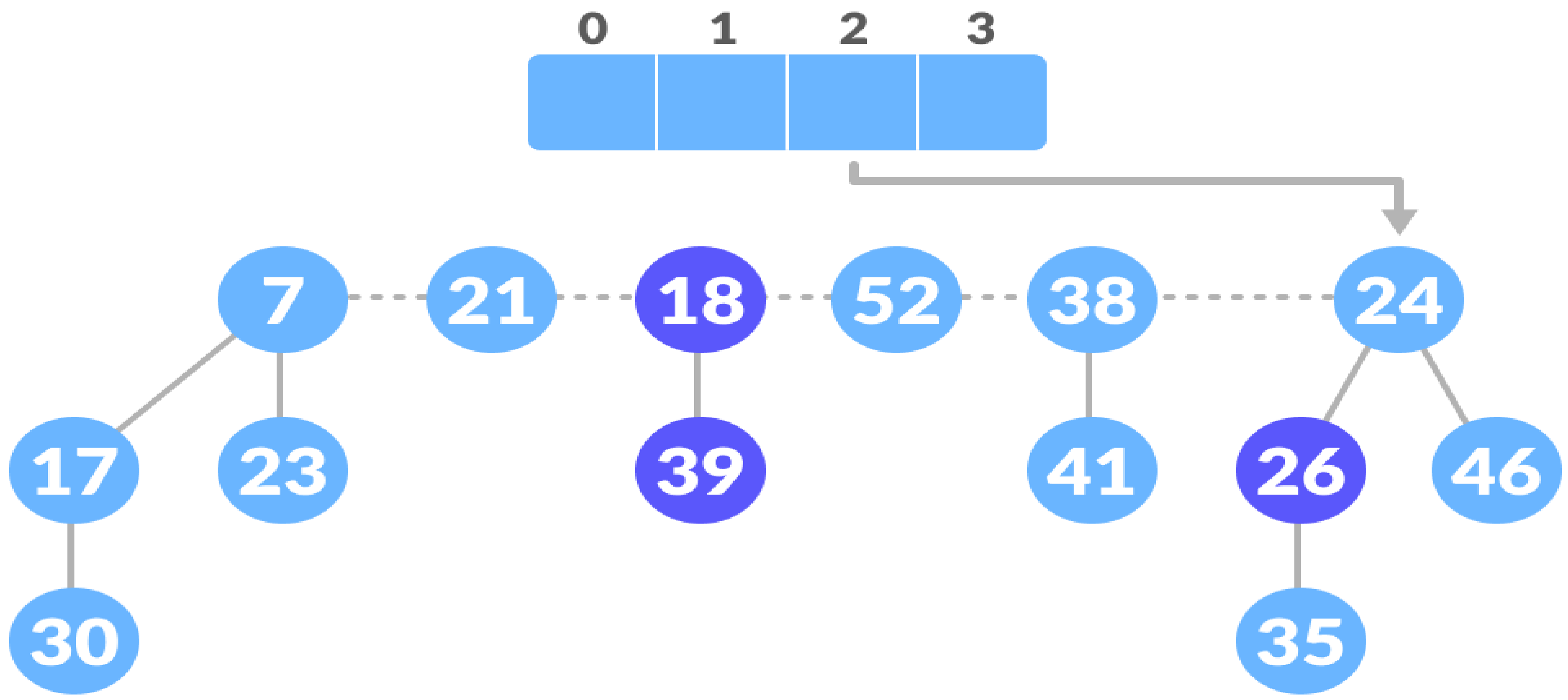
Here, 23 and 7 have the same degrees, so unite them.

Extract-min operation on the heap



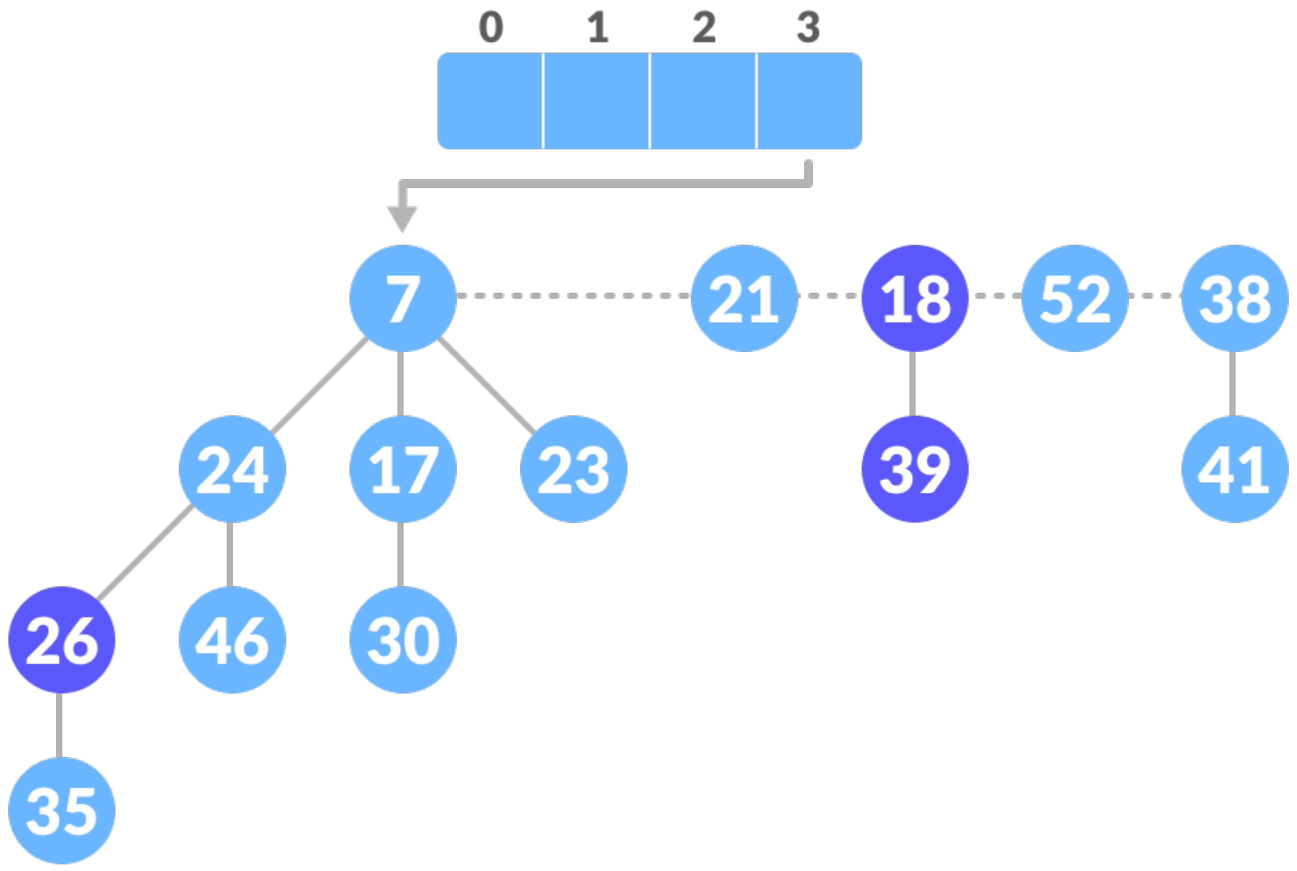
Again, 7 and 17 have the same degrees, so unite them as well

Extract-min operation on the heap



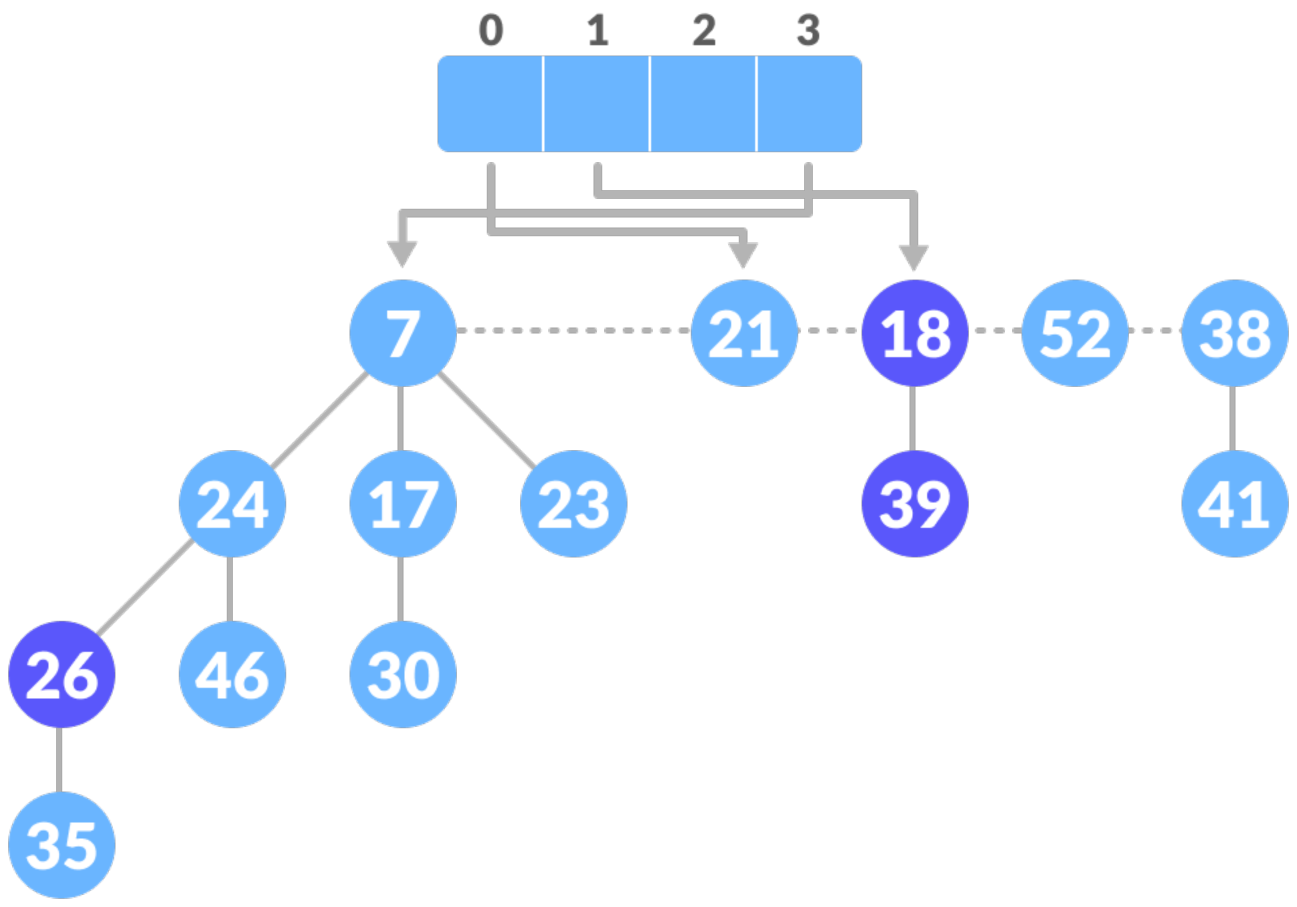
Again 7 and 24 have the same degree, so unite them

Extract-min operation on the heap



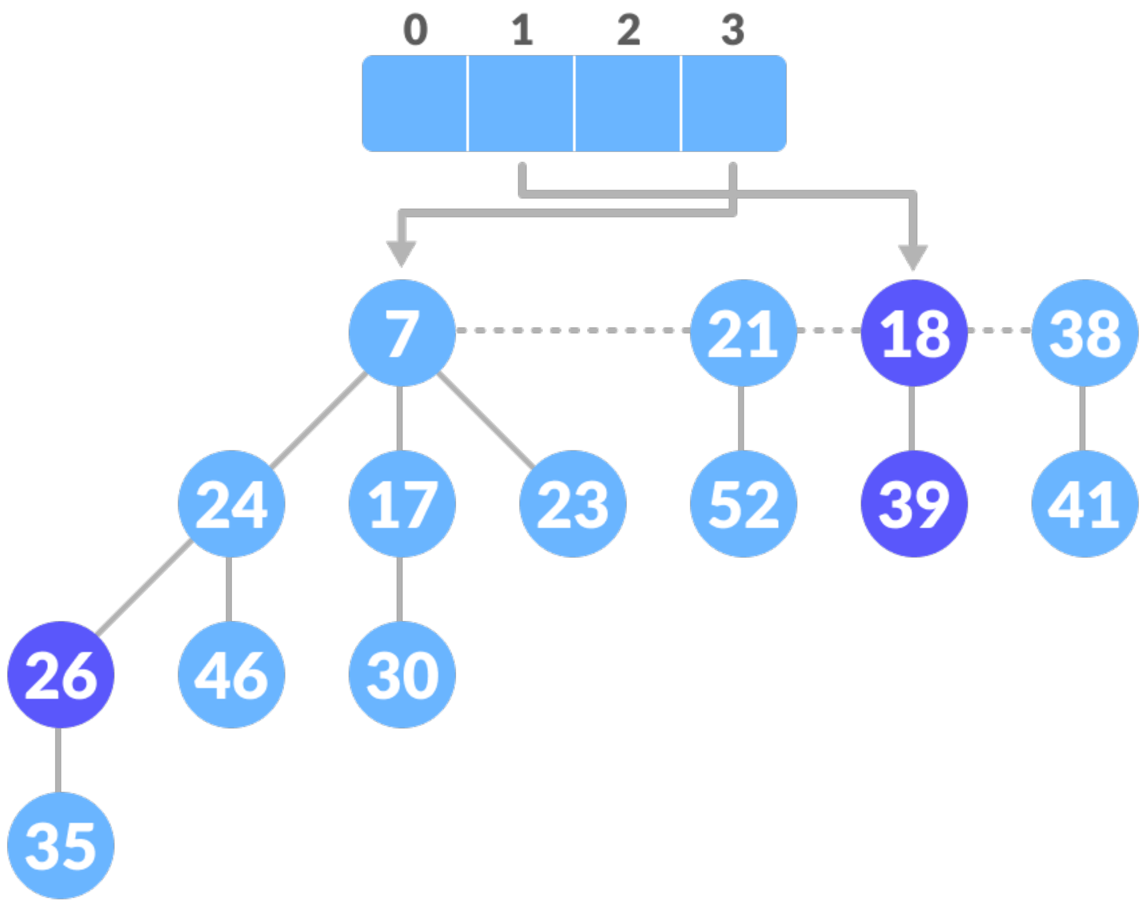
Map the next nodes.

Extract-min operation on the heap

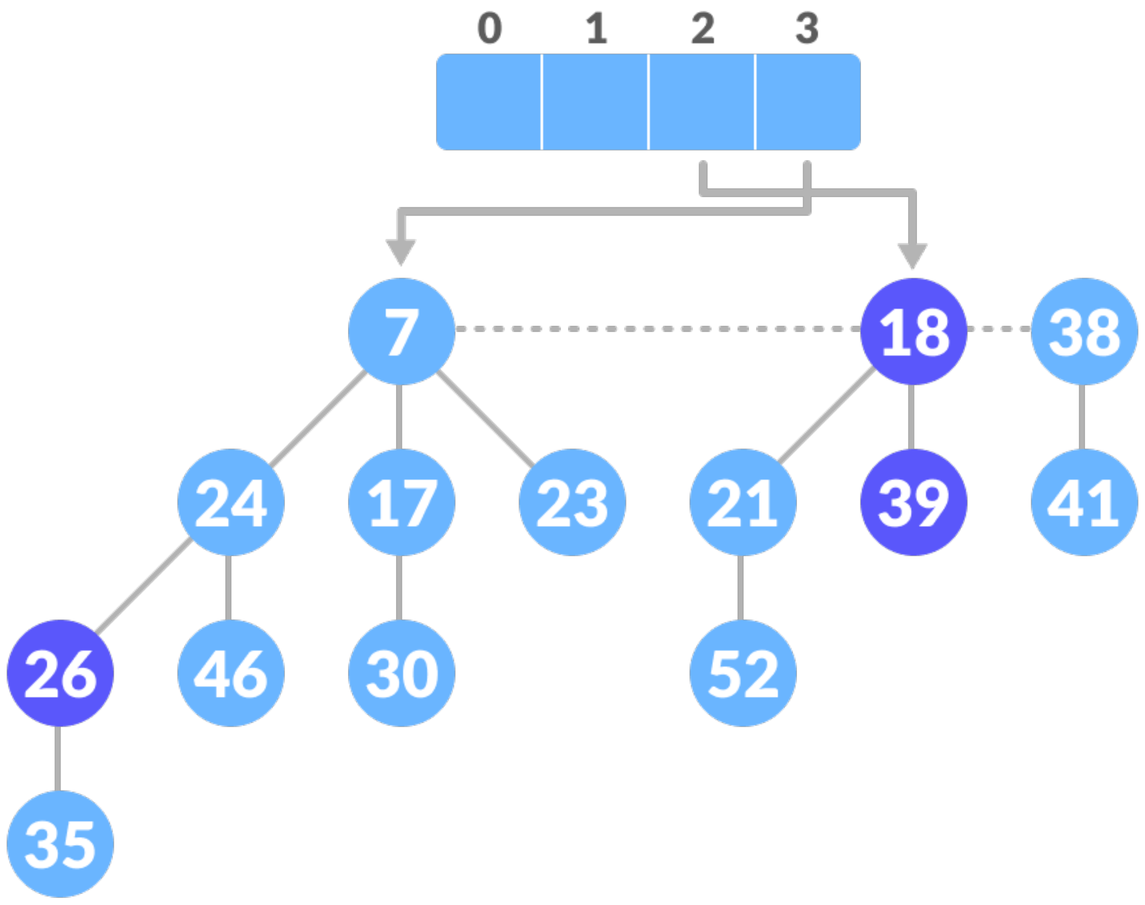


Again, 52 and 21 have the same degree, so unite them

Extract-min operation on the heap

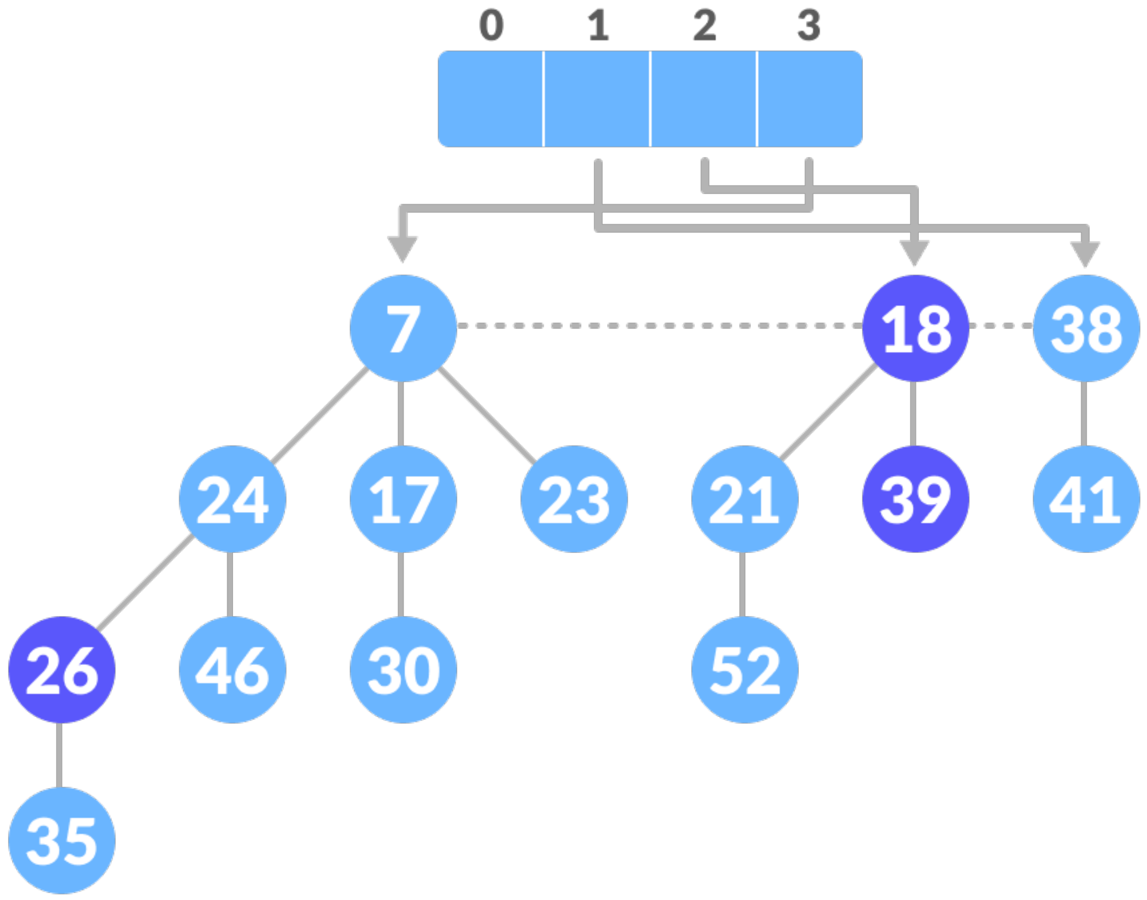


Extract-min operation on the heap

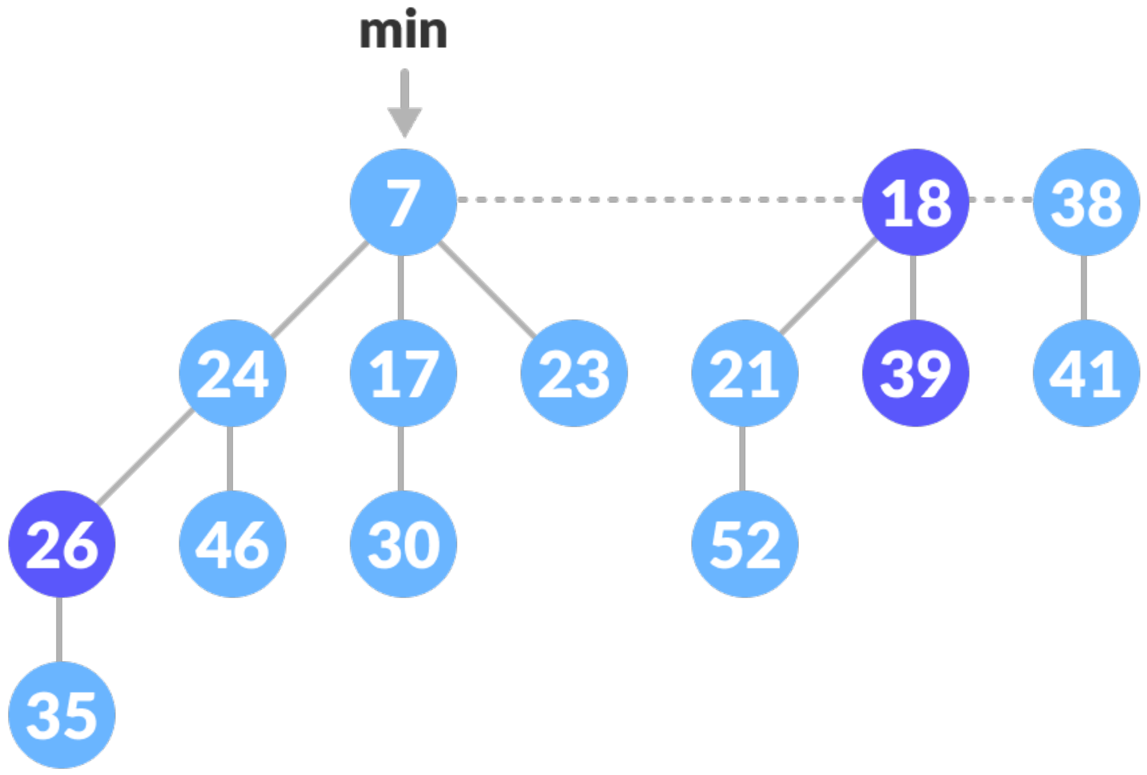


Similarly, unite 21 and 18.

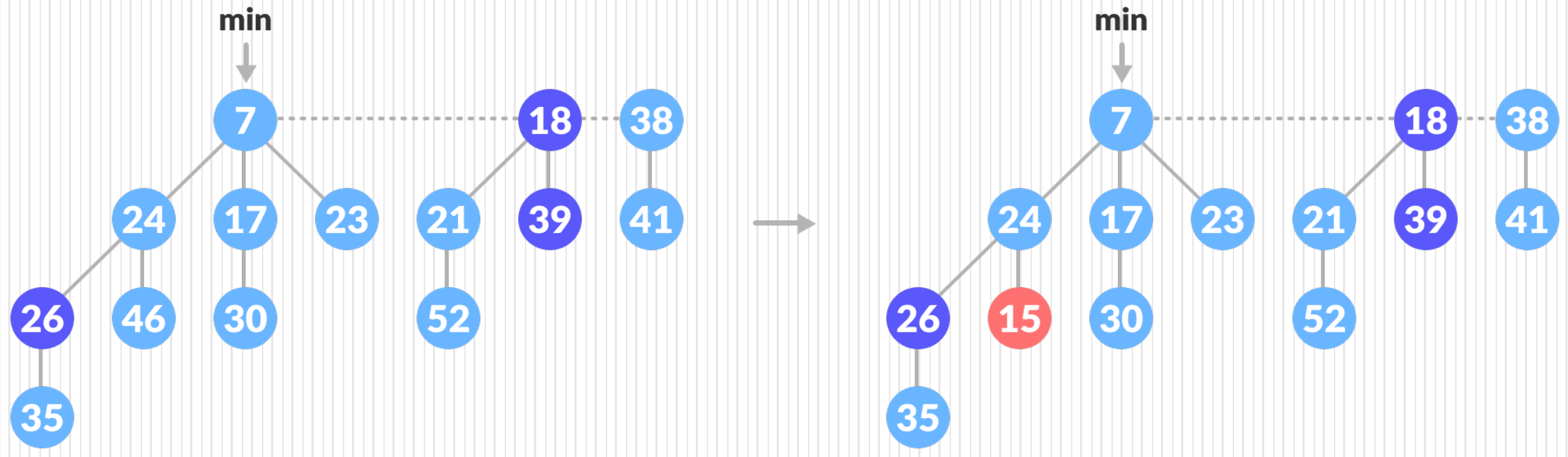
Extract-min operation on the heap



Extract-min operation on the heap: The final heap



Decrease Key and Delete Node Operations on a Fibonacci Heap



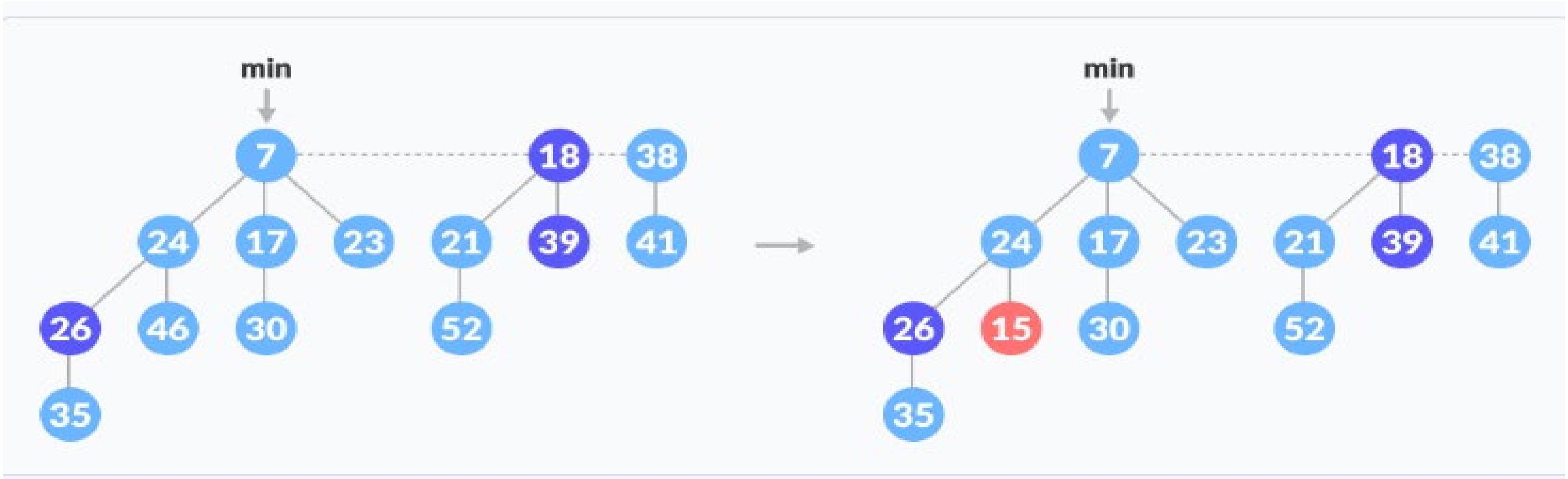
Decrease Key operation in a Fibonacci Heap

- The **Decrease Key** operation in a Fibonacci Heap is a key component for maintaining the heap's efficiency, particularly in graph algorithms like Dijkstra's shortest path, where reducing key values is frequent.

Algorithm: Decrease Key in a Fibonacci Heap

Input: A Fibonacci heap H, a node x within H, and a new key k, where k is less than or equal to the current key of x.

Output: Updated Fibonacci heap with the key of x decreased to k.



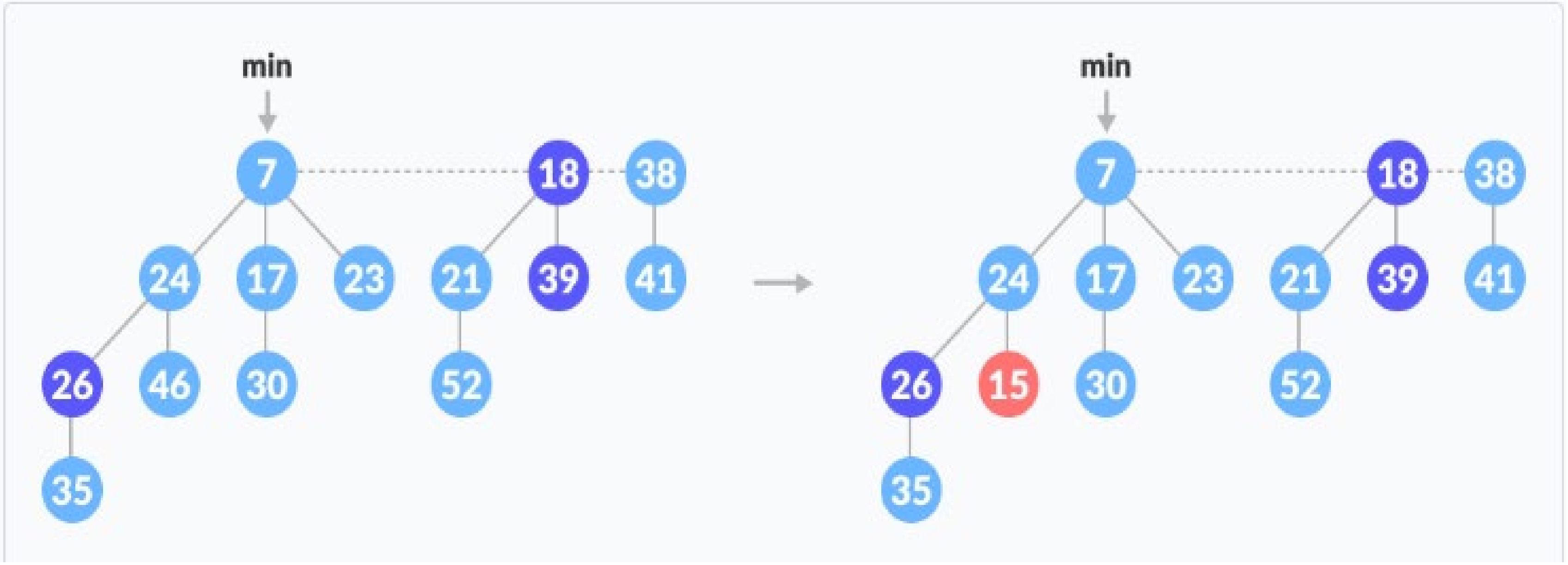
Decrease 46 to 15

Decrease-Key Operation (decrease-key):

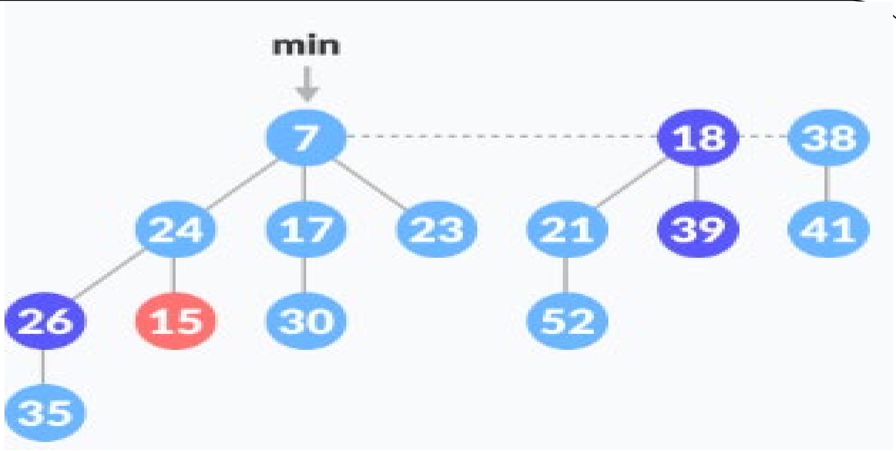
1. **Update the Node's Key**: Decrease the value of the specified node to the new value. It's assumed the new value is indeed less than the node's current value.
2. **Check Heap Property**: Compare the updated node's value with its parent. If the value of the node is now less than its parent, we've violated the heap property.
3. **Cut the Node**: If the heap property is violated:
   - Remove (or "cut") the node from its parent and add it to the root list.
   - Mark the parent if it's not already marked. If the parent is already marked, then cut the parent as well and add it to the root list, then continue this process up the tree. This is called a "cascading cut."
4. **Update Minimum Pointer**: If the decreased key is now the smallest in the heap, update the minimum pointer to point to this node.

Decrease Key Example

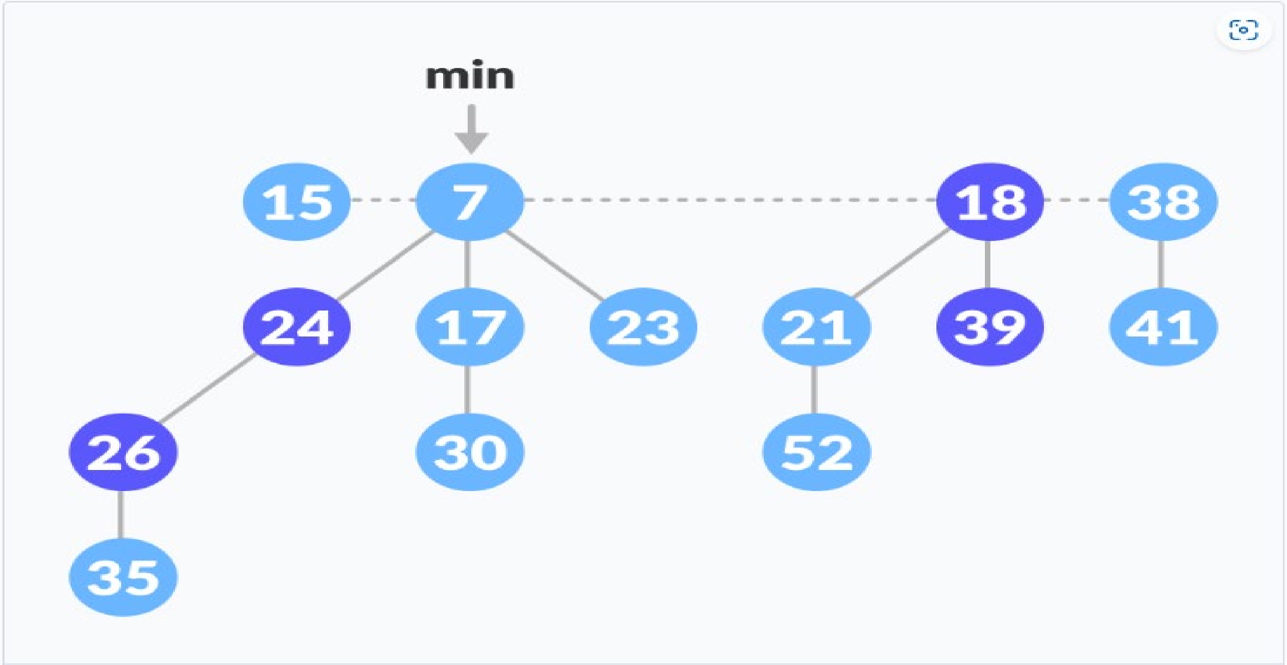
1. Decrease the value 46 to 15.



Decrease Key Example



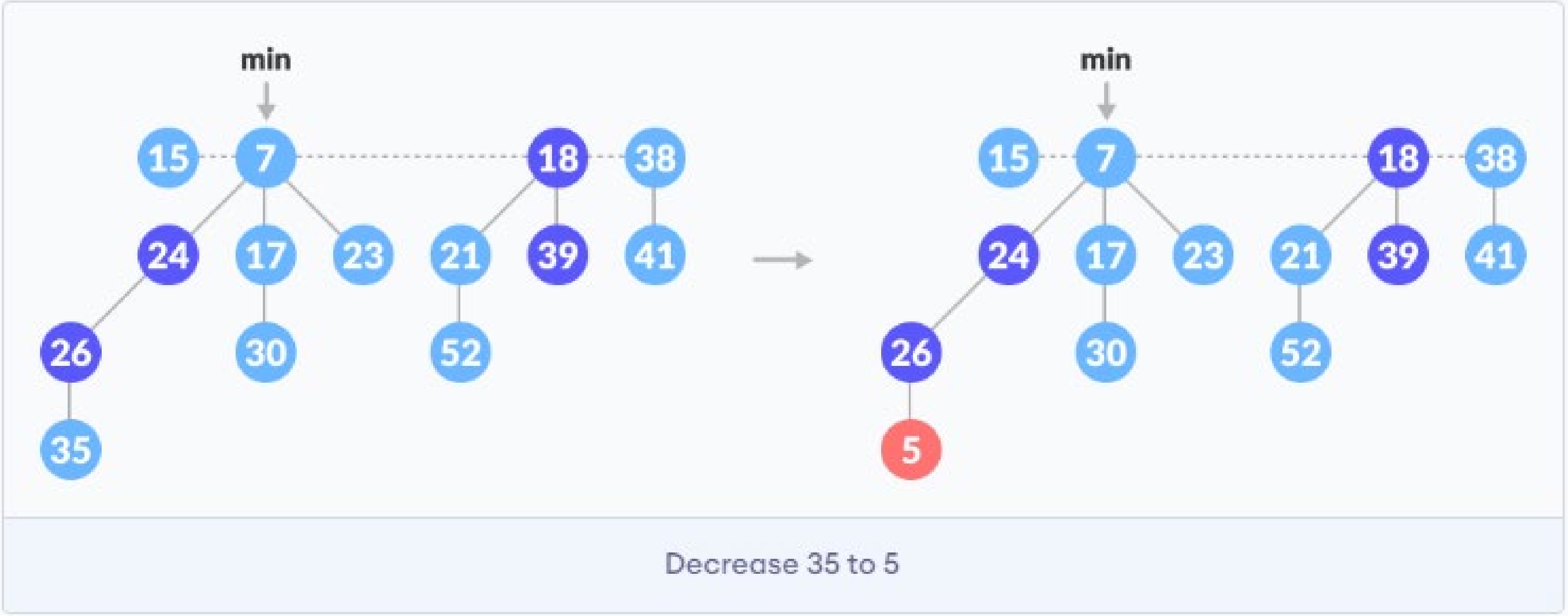
2. **Cut part:** Since `24 ≠ nill` and `15 < its parent`, cut it and add it to the root list.
Cascading-Cut part: mark 24.



Add 15 to root list and mark 24

Example: Decreasing 35 to 5

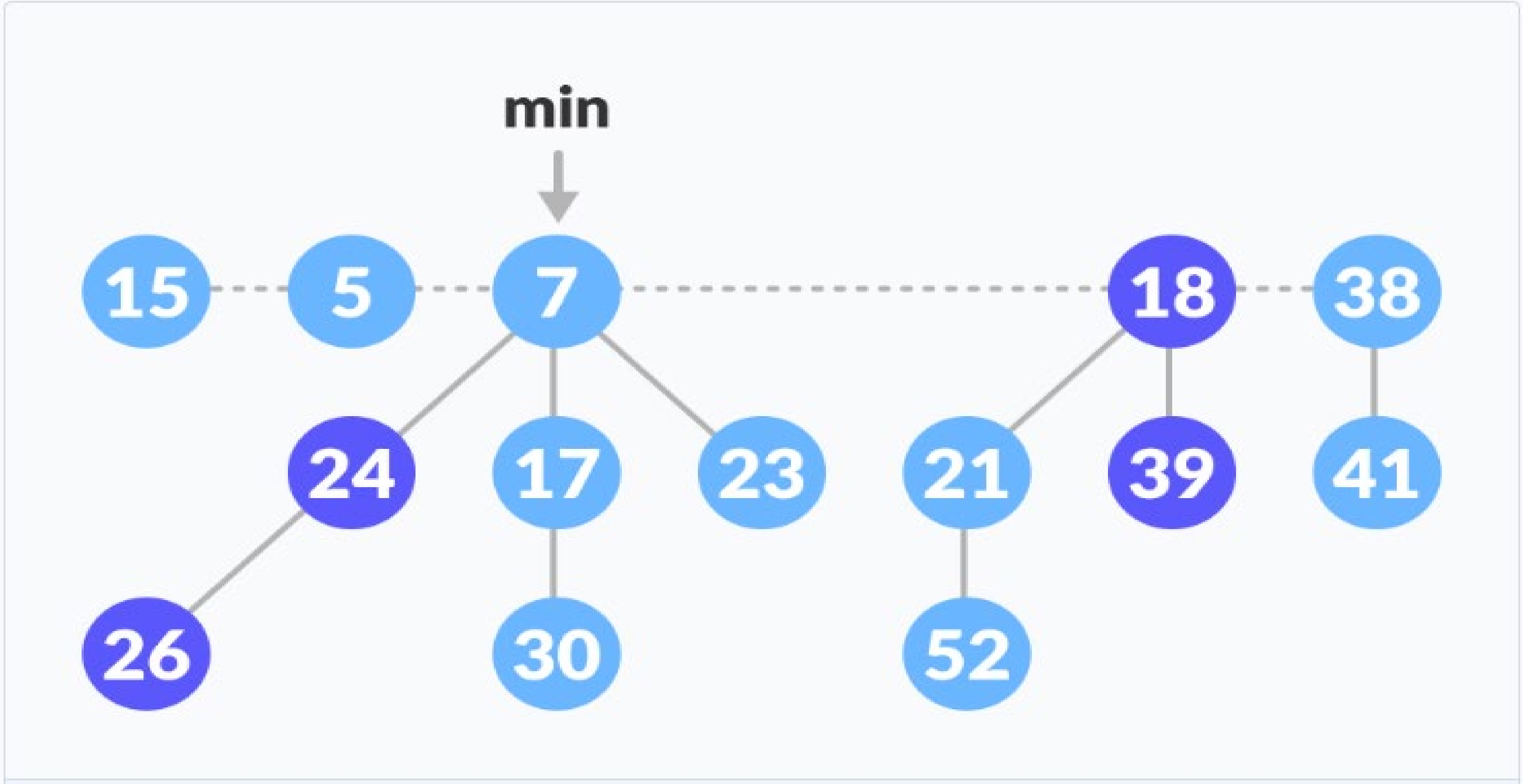
1. Decrease the value 35 to 5.



Decrease 35 to 5

Decrease Key Example

2. Cut part: Since `26 ≠ nill` and `5<its parent`, cut it and add it to the root list.

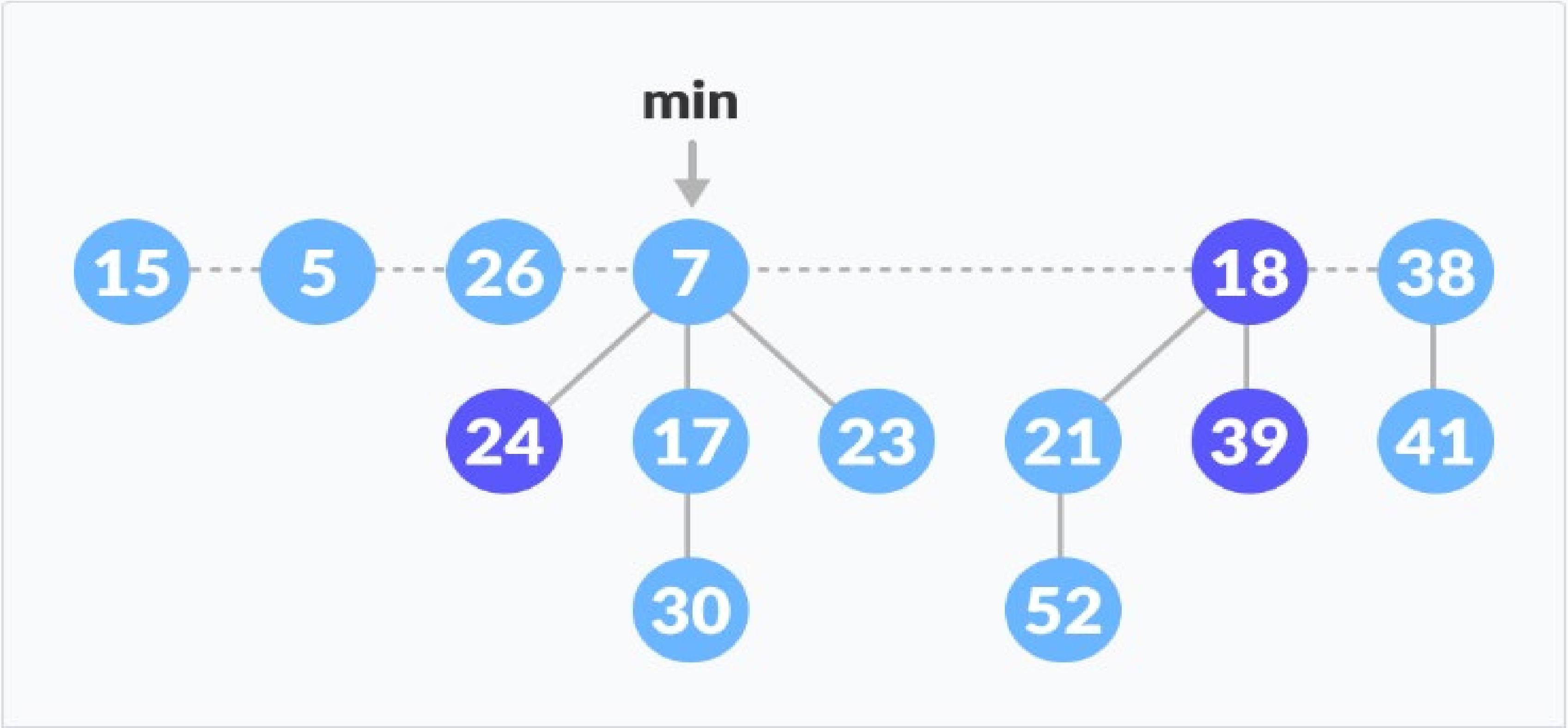


Cut 5 and add it to root list

Decrease Key Example

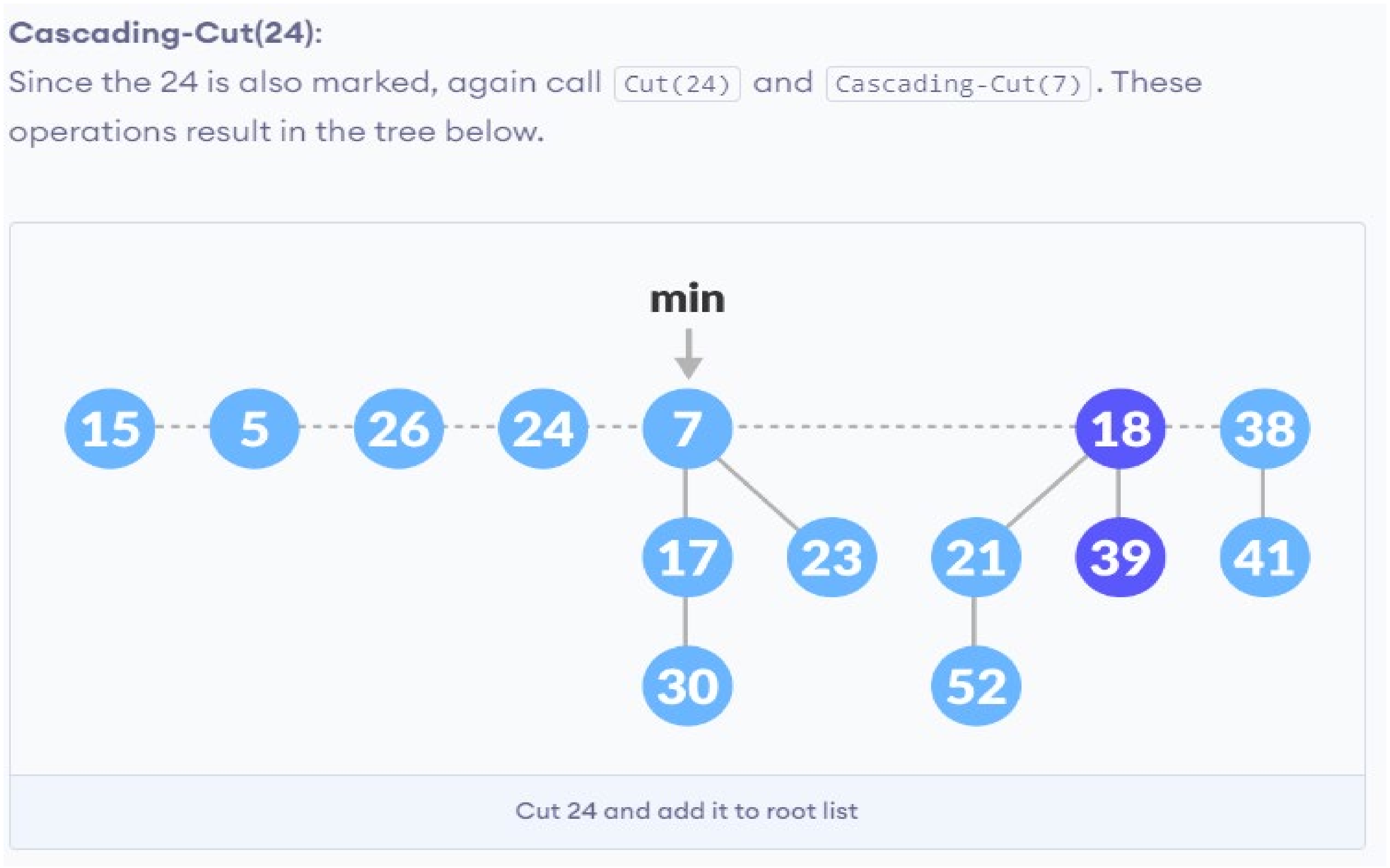
3. Cascading-Cut part: Since 26 is marked, the flow goes to `Cut` and `Cascading-Cut`.

Cut(26): Cut 26 and add it to the root list and mark it as false.



Cut 26 and add it to root list

Decrease Key Example



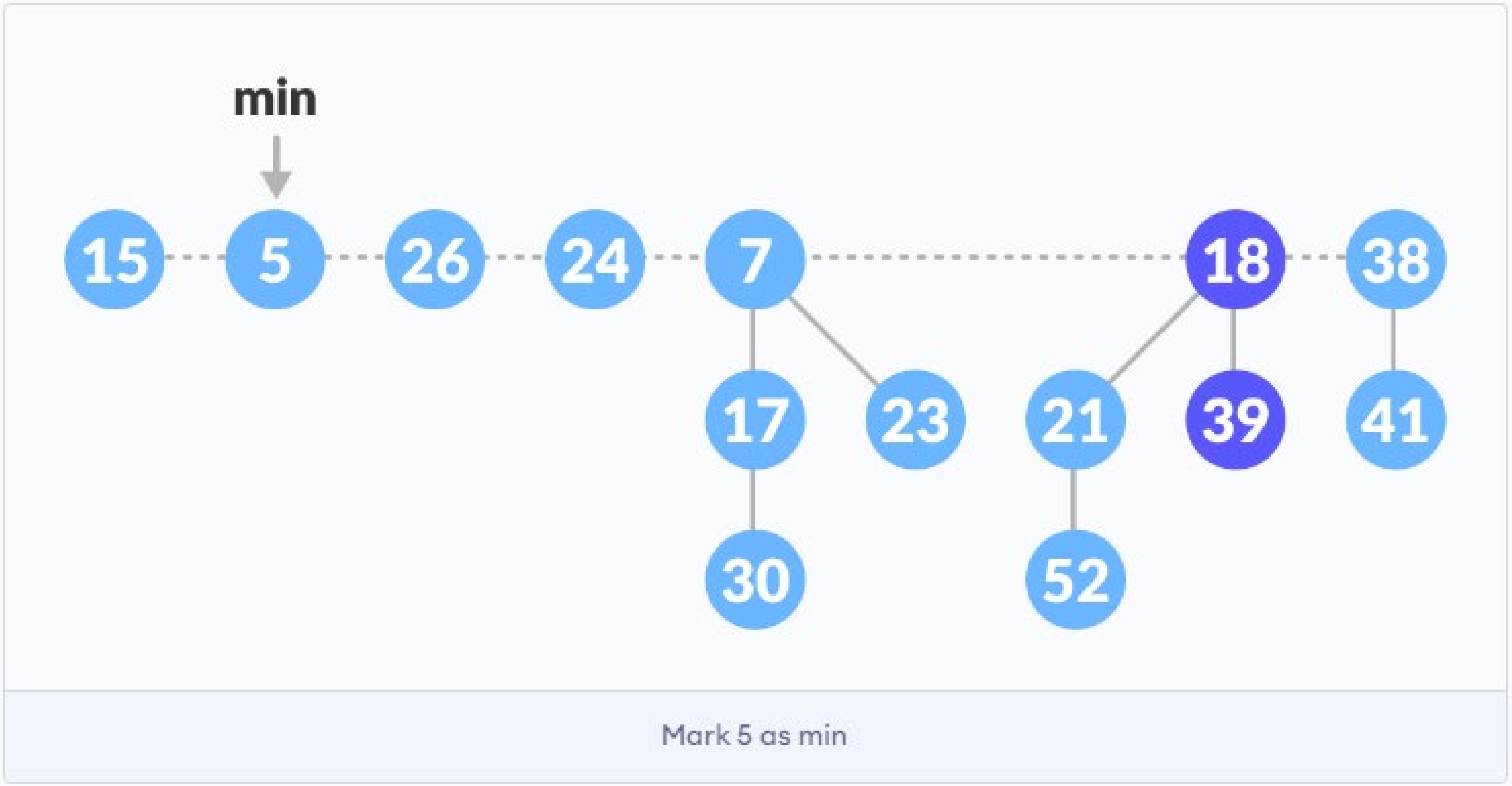
Cascading-Cut(24):

Since the 24 is also marked, again call `Cut(24)` and `Cascading-Cut(7)`. These operations result in the tree below.

Cut 24 and add it to root list

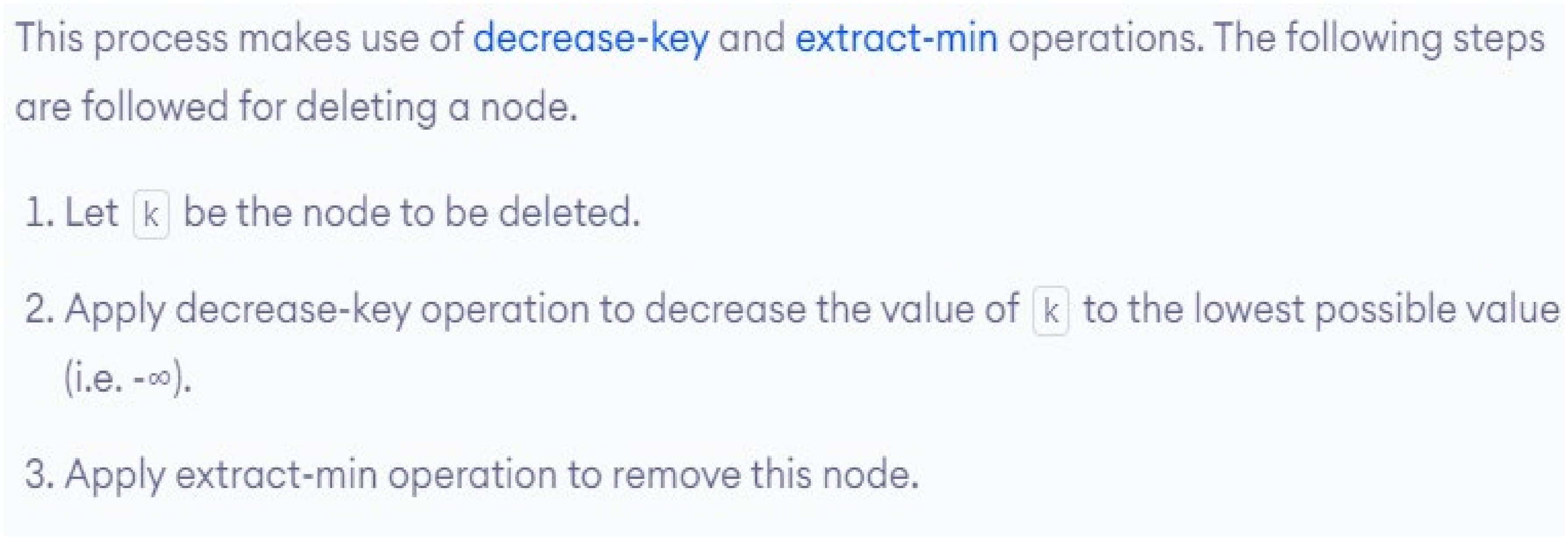
Decrease Key Example

4. Since `5 < 7`, mark 5 as min.



Mark 5 as min

Deleting a Node



This process makes use of decrease-key and extract-min operations. The following steps are followed for deleting a node.

1. Let `k` be the node to be deleted.
2. Apply decrease-key operation to decrease the value of `k` to the lowest possible value (i.e. -∞).
3. Apply extract-min operation to remove this node.

Question

You have a Fibonacci heap, initially empty, and the following sequence of operations is performed:

- Insert(10)
- Insert(2)
- Insert(8)
- Insert(5)
- Decrease-Key(node with value 8, new value 1)
- Extract-Min

Illustrate the state of the Fibonacci heap after each operation and identify the minimum value at each step.

Summary

- A Fibonacci Heap is a specialized data structure for priority queue operations, designed to perform **better than binary and binomial heaps in some cases**.
- A Fibonacci heap is a tree based data structure which consists of a collection of trees with **min heap** or **max heap** property.
- The operations are more efficient in terms of time complexity than those of its similar data structures like binomial heap and binary heap.
- It consists of a collection of trees that follow the minimum-heap property, where the key of each node is greater than or equal to the key of its parent.
- Fibonacci Heaps are especially efficient for operations that reduce key values, making them suitable for algorithms like Dijkstra's shortest path and Prim's minimum spanning tree.

Problem 1: Maximum Value Extraction

Question: Given a Fibonacci heap, describe an algorithm to extract the maximum value from it.

Solution Outline: Fibonacci heaps are designed to efficiently extract the minimum value, not the maximum. To get the maximum, you'd have to traverse all the nodes.

1. Perform a linear search across the root list to find the maximum value.
2. Use the cut operation to remove this node (and add it to the root list if it's not already there).
3. Use the extract-min operation on this node, treating it as the minimum (since we've identified it as the maximum).
4. If required, perform consolidation and update the minimum pointer.

Problem 2: Combining Heaps

Question: Given two Fibonacci heaps, H1 and H2, where every element in H1 is greater than every element in H2, describe an efficient algorithm to combine them.

Solution Outline:

- Use the union operation to combine H1 and H2.
- Since every element in H1 is greater than every element in H2, the minimum pointer of the resulting heap should point to the minimum pointer of H2.

Problem 3: Node Frequency

Question: Given a Fibonacci heap, design an algorithm to determine the frequency of nodes having k children.

Solution Outline:

- Traverse each node in the heap.
- For each node, count its children. If the count is k, increment your frequency counter.
- Return the frequency counter at the end.

Problem 4: Fibonacci Heap Sort

Question: Describe a method to sort a list of n numbers using a Fibonacci heap.

Solution Outline:

- Insert all numbers into a Fibonacci heap.
- Repeatedly call extract-min until the heap is empty, appending each extracted value to a list.
- The list now contains the numbers in sorted order.
- Remember, while this method would work, it isn't the most efficient way to sort numbers. The primary strength of Fibonacci heaps lies in their amortized time complexities for specific operations, making them more suitable for algorithms like Dijkstra's or Prim's, rather than for sorting.

Problem 5: Node Removal

Question: Design an algorithm to remove an arbitrary node from a Fibonacci heap.

Solution Outline:

- Use the decrease-key operation to decrease the value of the node to negative infinity (or a value you know is less than the minimum).
- Use extract-min to remove the node.
- Ensure any potential structural issues in the heap are resolved post-removal (like consolidating trees of the same rank).

Problem 6: Find an Element

- Given a Fibonacci heap, design an algorithm to check if an element with value x exists in the heap.
- **Solution Outline**: Since Fibonacci heaps don't support direct lookup operations, you'd need to perform a traversal.

1. Implement a depth-first traversal of the heap.
2. For each node, compare its value with x.
3. If a match is found, return True; otherwise, continue the search.

Problem 7: Merge K Sorted Lists

- You have k sorted lists with a total of n elements. Describe an algorithm to merge these lists into one sorted list using a Fibonacci heap.
- **Solution Outline**:

1. Insert the first element of each list into the Fibonacci heap.
2. Extract the minimum from the heap and append it to the result list.
3. Insert the next element from the list which provided the extracted element into the heap.
4. Repeat steps 2 and 3 until the heap is empty.

Problem 8: Fibonacci Heap Depth

- Design an algorithm to find the maximum depth (or height) of a Fibonacci heap.
- **Solution Outline**:
- Traverse each tree in the root list of the heap.
- For each tree, perform a depth-first traversal, keeping track of the depth.
- Return the maximum depth encountered.

Problem 9: Decrease All by K

- Given a Fibonacci heap and an integer k, describe an efficient method to decrease the value of all elements in the heap by k.
- **Solution Outline**:
- For each node in the root list, decrease its value by k.
- Traverse and adjust any node that now violates the heap property by moving it to the root list and marking its parent if necessary.
- If any nodes in the root list have the same degree after these adjustments, consolidate them.
-

Problem 8: Fibonacci Heap Depth

- Design an algorithm to find the maximum depth (or height) of a Fibonacci heap.
- **Solution Outline**:
- Traverse each tree in the root list of the heap.
- For each tree, perform a depth-first traversal, keeping track of the depth.
- Return the maximum depth encountered.

Problem 9: Decrease All by K

- Given a Fibonacci heap and an integer k, describe an efficient method to decrease the value of all elements in the heap by k.
- **Solution Outline**:
- For each node in the root list, decrease its value by k.
- Traverse and adjust any node that now violates the heap property by moving it to the root list and marking its parent if necessary.
- If any nodes in the root list have the same degree after these adjustments, consolidate them.
-

Problem 10: Compare Heaps

Question: Given two Fibonacci heaps, H1 and H2, design an algorithm to check if they contain the exact same set of elements (not necessarily in the same structure, but the same values).

Solution Outline:

- Use extract-min repeatedly on both heaps to get the elements in ascending order.
- Compare the extracted elements from both heaps at each step.
- If all extracted elements match and both heaps are emptied at the same time, they contain the same set of elements.
- Note that this method destructively compares the heaps (i.e., empties them). To avoid this, you could copy the heaps before comparison or devise another method that doesn't rely on extract-min.

Advantages & Disadvantages

Advantages:

1. **Faster Amortized Time**: For operations like decrease-key and delete, Fibonacci heaps provide better amortized time complexities compared to other heap structures.
2. **Applications in Graph Algorithms**: The improved efficiencies of these operations make Fibonacci heaps suitable for advanced algorithms like Dijkstra's and Prim's.

Disadvantages:

1. **Complex Implementation**: The data structure itself is more complex to implement than binary or binomial heaps.
2. **Slower Worst-case Time**: While the amortized time for many operations is excellent, the worst-case time can be less efficient than simpler structures.
3. **Memory Overhead**: Maintaining the structure requires more memory overhead than simpler heaps.

Application of Fibonacci Heap

- **Graph algorithms:** It is used in algorithms such as Dijkstra's shortest path algorithm and Prim's algorithm for finding minimum spanning trees.
- **Network flow algorithms:** Fibonacci heap is used in algorithms such as the Edmonds-Karp algorithm for finding maximum flow in a network.
- **Heap sort:** It is used in heap sort algorithms as a more efficient alternative to binary heaps.
- **Mathematical optimization:** Fibonacci heap is used in optimization problems to efficiently find the minimum or maximum element in a set.
- **Task scheduling:** It is used in scheduling algorithms for efficient resource allocation and task prioritization.
- **Computational geometry:** It is used in geometric algorithms to efficiently compute intersections, closest pairs, and convex hulls.

Thank you for your attention

References

- http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap21.htm
- https://www.prepbytes.com/blog/heap/fibonacci-heap/
- https://www.programiz.com/dsa/fibonacci-heap

Exercises

- Show the Fibonacci heap that results from calling FIB-HEAP-EXTRACT-MIN on the Fibonacci heap shown in Figure 21.3(m).
- Prove that Lemma 20.1 holds for unordered binomial trees, but with property 4' in place of property 4.
- Show that if only the mergeable-heap operations are supported, the maximum degree *D(n)* in an *n*-node Fibonacci heap is at most 1g *n*.
- Professor McGee has devised a new data structure based on Fibonacci heaps. A McGee heap has the same structure as a Fibonacci heap and supports the mergeable-heap operations. The implementations of the operations are the same as for Fibonacci heaps, except that insertion and union perform consolidation as their last step. What are the worst-case running time of operations on McGee heaps? How novel is the professor's data structure?