Fibonacci Heap

Data Structures and Algorithms

Priyanshu Kumar Vasim Ali

Indian Institute of Information Technology, Bhubaneswar

November 20, 2025

Contents

Introduction

Before learning the **Fibonacci Heap**, it is important to understand the basic idea of a **heap**. A heap is a special tree-based data structure that helps us quickly find the smallest or largest value.

There are two fundamental types of heaps:

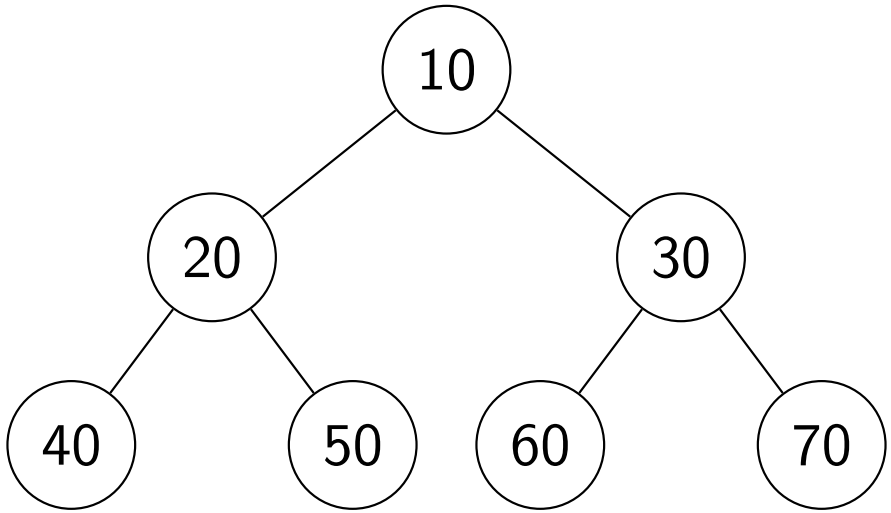
1. **Min-Heap**
2. **Max-Heap**

Min-Heap

Min-Heap is a binary tree–based structure where:

- The **root contains the smallest element**.
- Every parent node is **less than or equal to** its children.
- The tree is a **complete binary tree**.

Example Min-Heap:

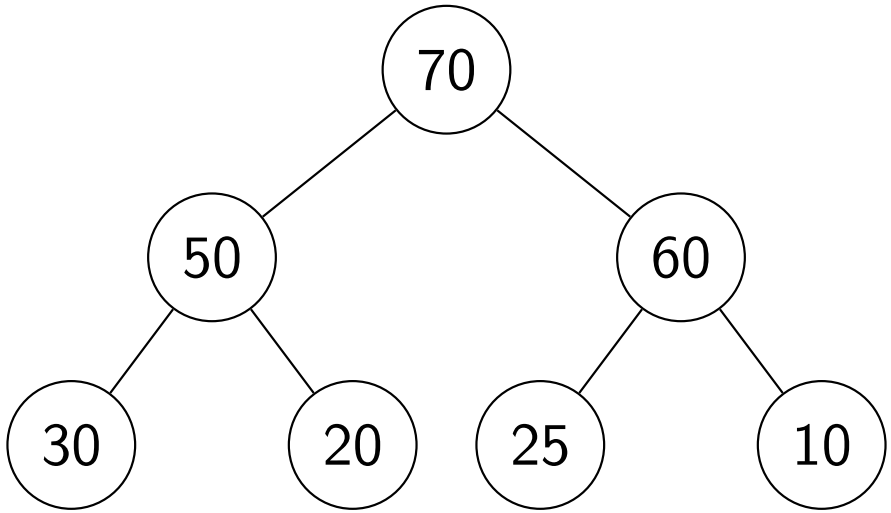


Max-Heap

Max-Heap is a binary tree–based structure where:

- The **root contains the largest element**.
- Every parent node is **greater than or equal to** its children.
- The tree is a **complete binary tree**.

Example Max-Heap:



Introduction to Fibonacci Heap

A **Fibonacci Heap** is a collection of trees that follow the **min-heap property**, meaning the key of every child is greater than or equal to the key of its parent. Thus, the **minimum element** is always at the root of one of the trees.

Key characteristics:

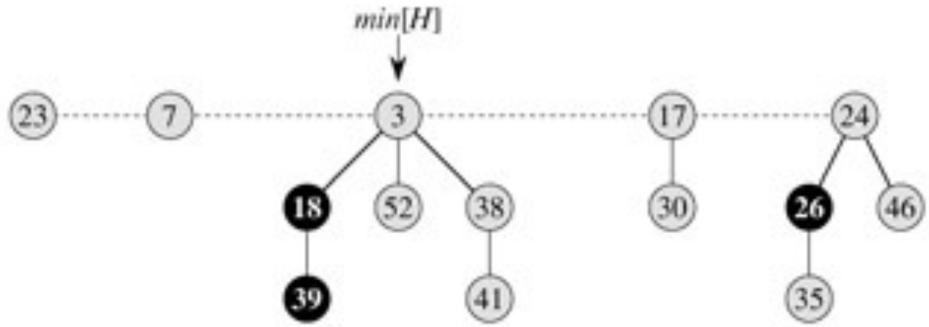
- More **flexible** than binomial heaps; tree shapes are not fixed.
- Supports **lazy operations** — work is postponed until required.
- **Merge (Union)** of heaps is very fast: done by simply concatenating tree lists.
- **Decrease-Key** may cut a node and make it a new root (used for efficiency).
- Consolidation happens during **delete-min**, not after every insert.
- Degree of a node is at most $O(\log n)$ due to Fibonacci number structure.

Fibonacci Heaps: Structure

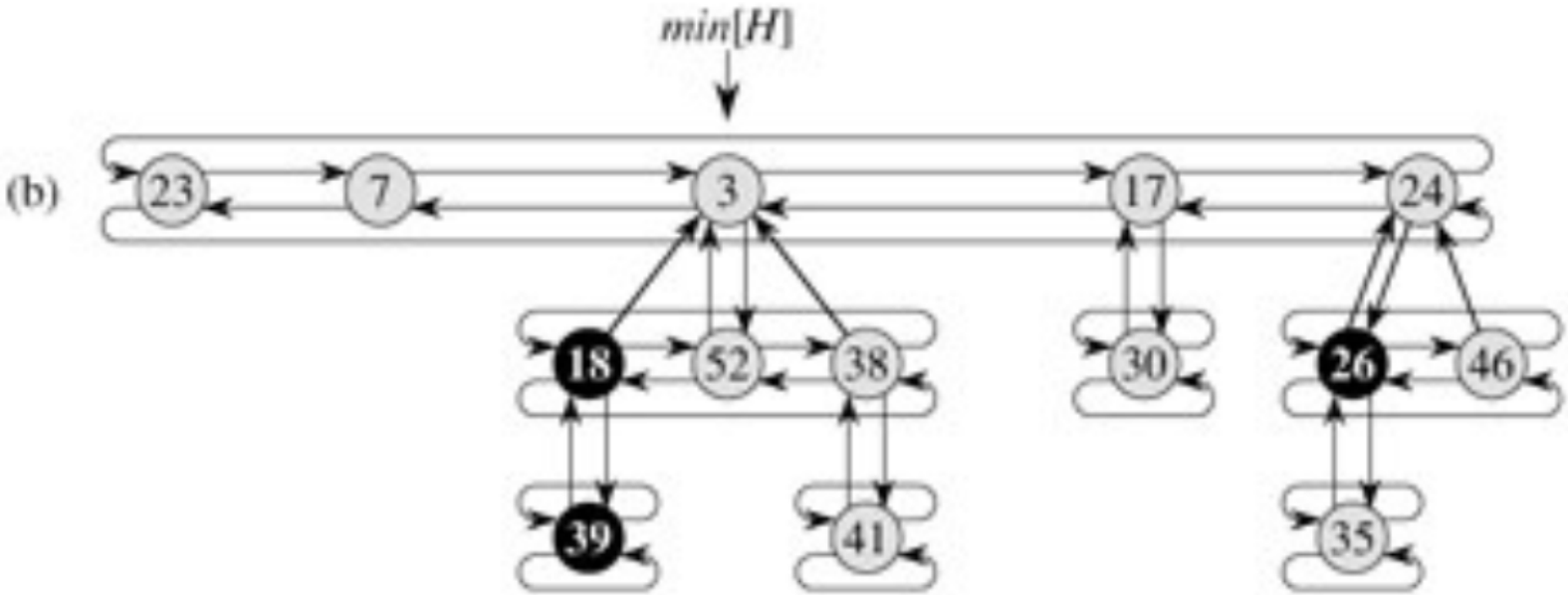
Fibonacci heap:

- Set of **heap-ordered trees**.
- Maintain pointer to the **minimum element**.
- Set of **marked nodes**.

parent pointer	
data	
degree	
left sibling	right sibling
mark	
any child	



Fibonacci Heaps: Structure



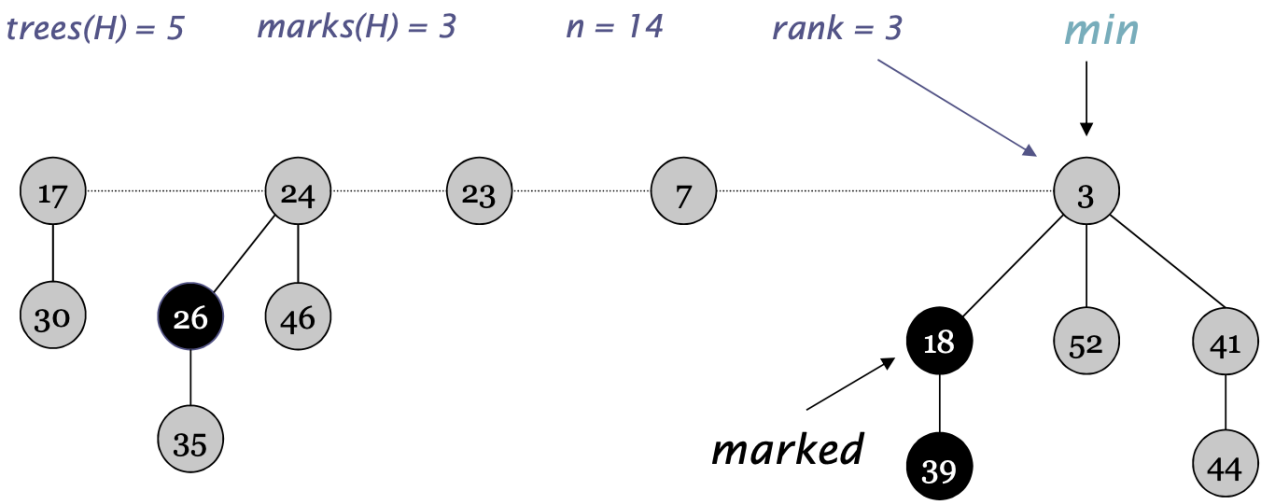
Doubly Circular Linked List

Dark nodes represent marked nodes.

Fibonacci Heaps: Structure

Basic Terms

- $n(H)$ = number of nodes in the heap.
- degree(x) = number of children of node x.
- $t(H)$ = number of trees in the root list.
- $m(H)$ = number of marked nodes in the heap.



Operations on Fibonacci Heap

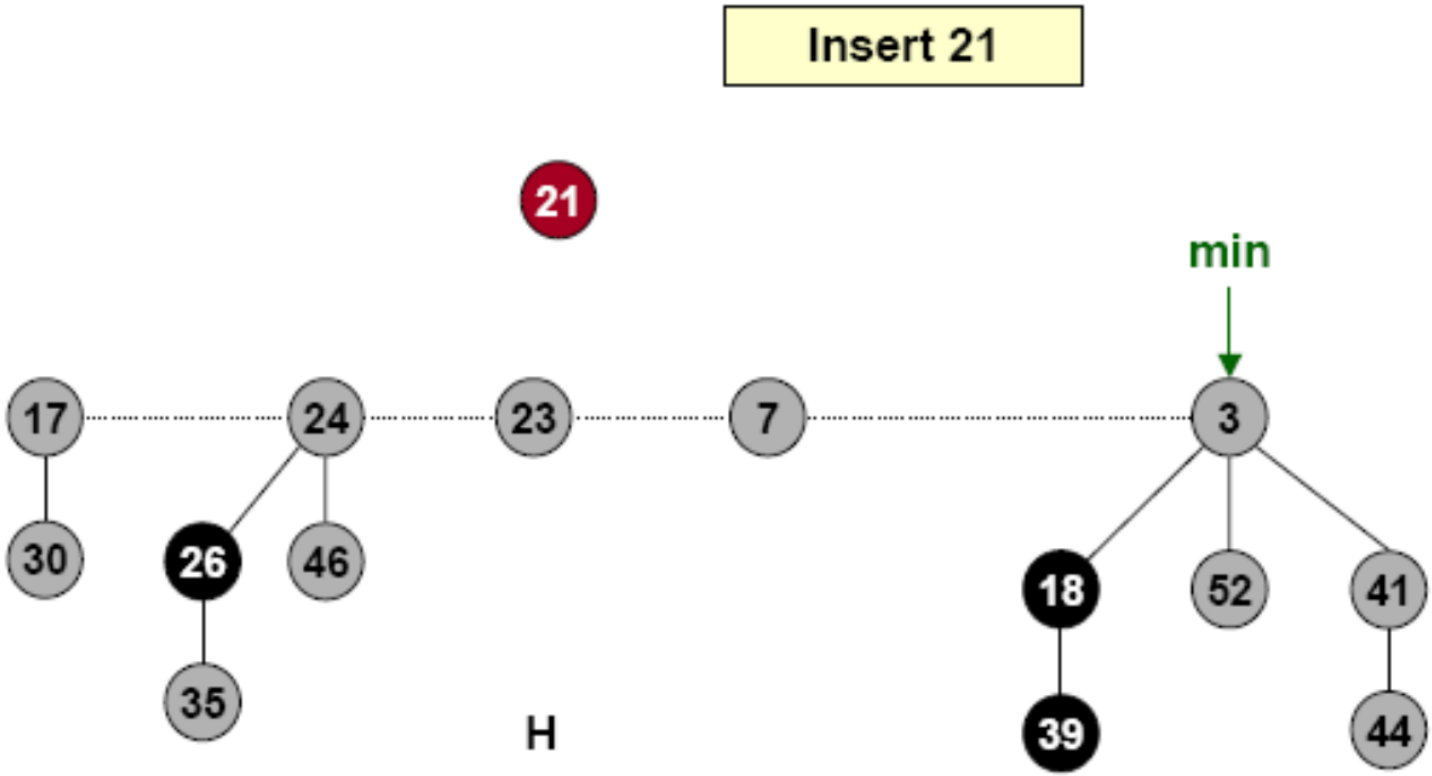
The main operations supported by a Fibonacci Heap are:

- **Insert** — Insert a new key into the heap.
- **Union (Merge)** — Combine two heaps by concatenating their root lists.
- **Delete-Min (Extract-Min)** — Remove the minimum element and consolidate the trees.
- **Decrease-Key** — Reduce the value of a key and perform cascading cuts if required.
- **Delete** — Remove a node by decreasing its key to $-\infty$ and then performing delete-min.

FIB-HEAP-INSERT(H, x)

Operation: Insert a Node into a Fibonacci Heap

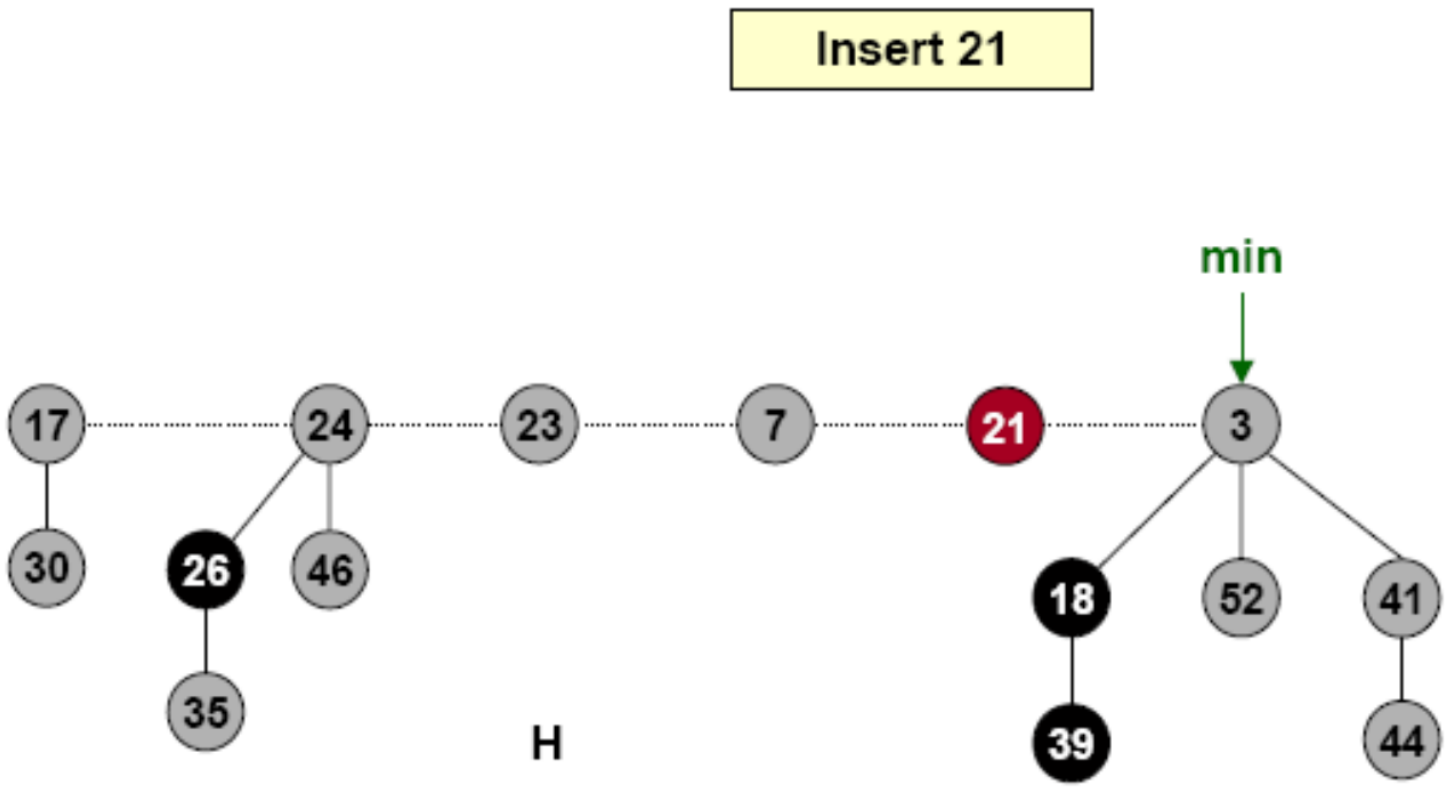
- Create a new singleton tree
- Insert the node into the root list (to the left of min pointer)
- Update the min pointer if needed



FIB-HEAP-INSERT(H, x)

Operation: Insert a Node into a Fibonacci Heap

- Create a new singleton tree
- Insert the node into the root list (to the left of min pointer)
- Update the min pointer if needed



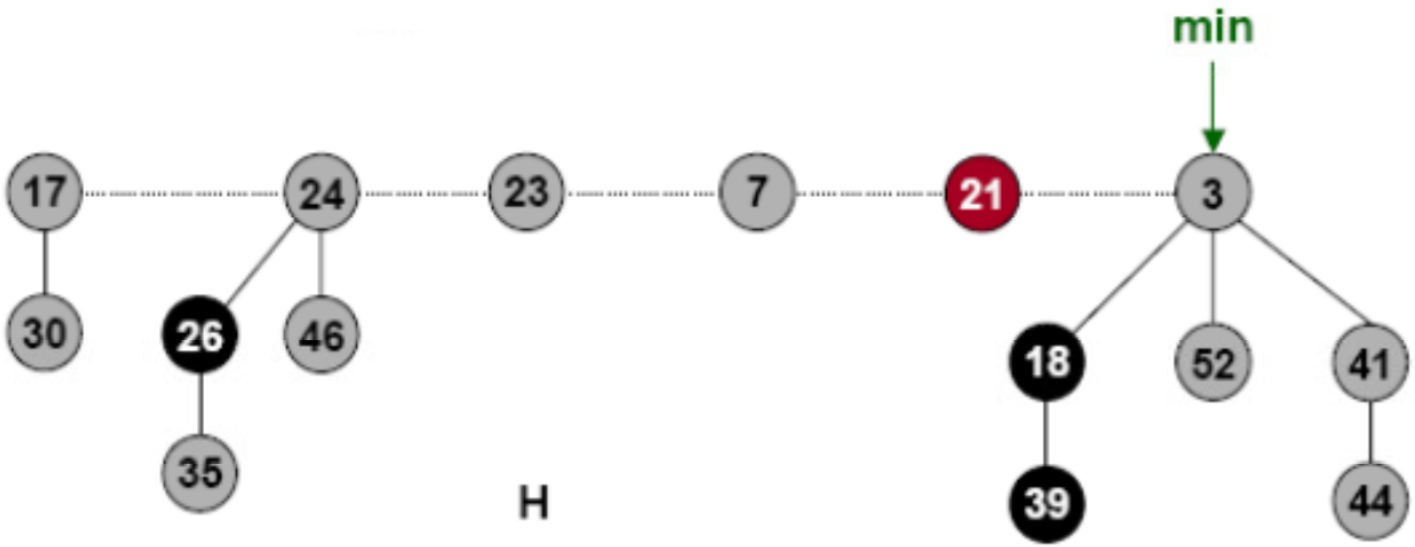
FIB-HEAP-INSERT(H, x)

Running Time:

- Increase in potential function is:

$$((t(H) + 1) + 2m(H)) - (t(H) + 2m(H)) = 1$$

- Since actual cost is $O(1)$,
- The amortized cost is $O(1) + 1 = O(1)$



Insert Operation: Time Complexity

Insert(x) in a Fibonacci Heap works by simply:

- Creating a new node.
- Adding it directly to the **root list**.
- Updating the **min pointer** if needed.

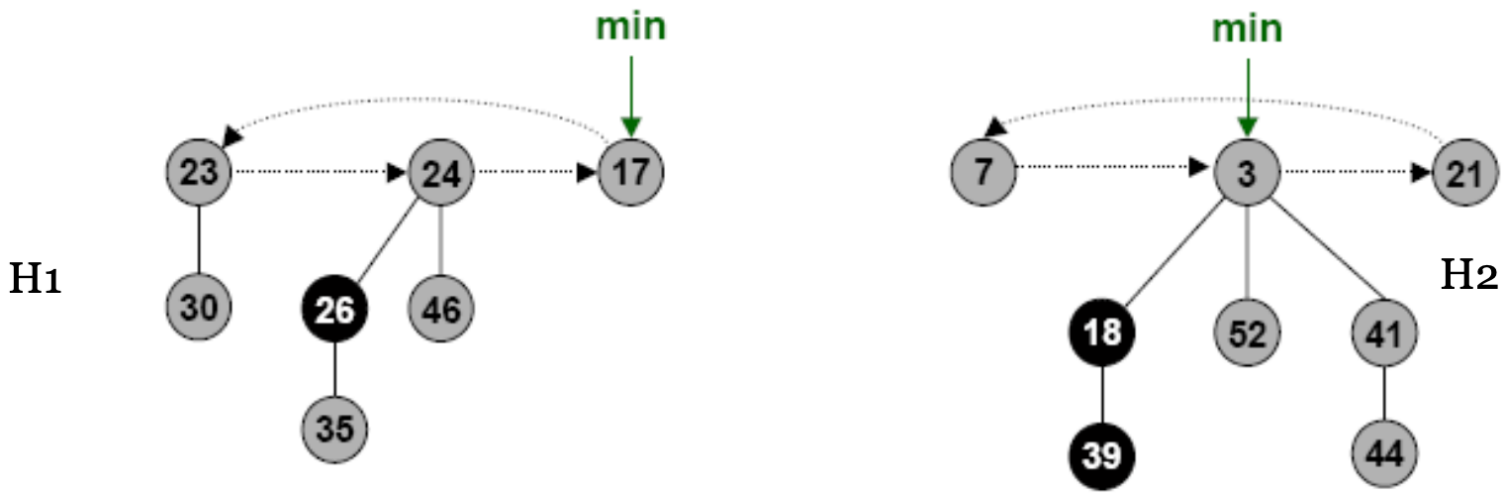
Time Complexity:

$$\boxed{O(1) \text{ amortized}}$$

FIB-HEAP-UNION

Union Operation:

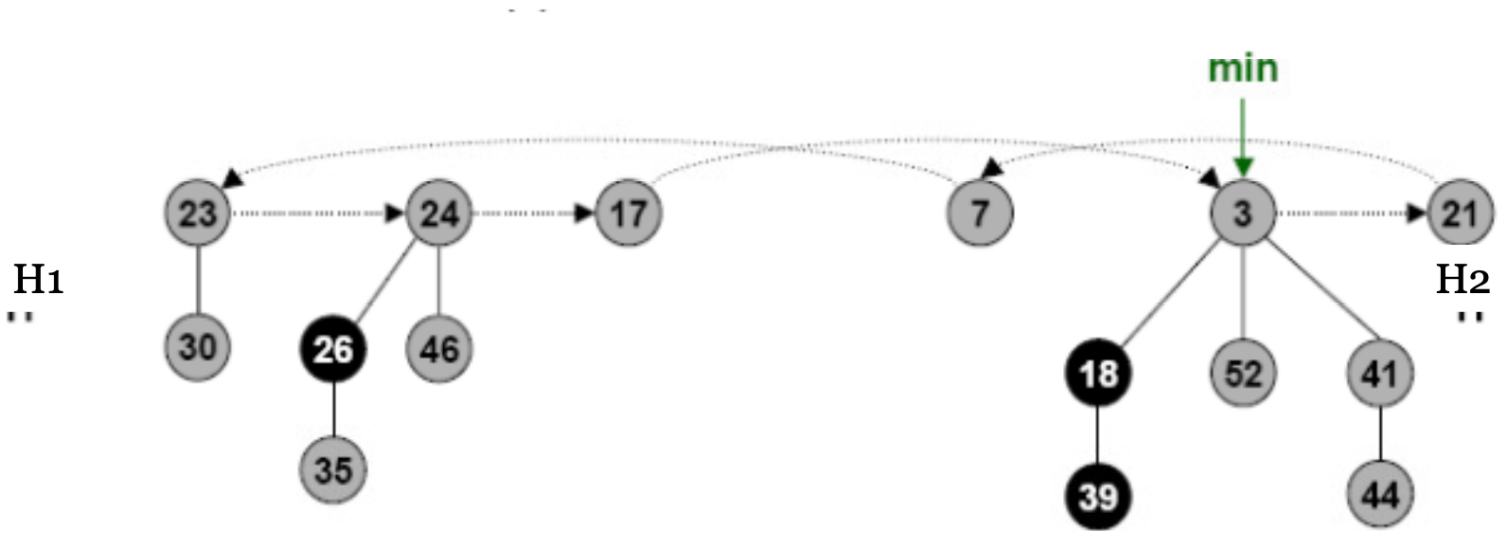
- The UNION operation joins two Fibonacci heaps.
- The root lists of both heaps are concatenated.
- Root lists are maintained as **circular doubly linked lists**.
- Update the min pointer to the smaller of the two minimum keys.
- This operation takes $O(1)$ actual time.



FIB-HEAP-UNION

How to Connect Two Heaps:

- Take the root list of the first heap.
- Take the root list of the second heap.
- Join (concatenate) both root lists together.
- This creates one big circular doubly linked list.
- After connecting, update the minimum pointer.



FIB-HEAP-UNION: Time Complexity

FIB-HEAP-UNION(H_1, H_2) performs:

- Concatenation of the two root lists.
- Updating the pointer to the minimum element.

Time Complexity:

$$O(1)$$

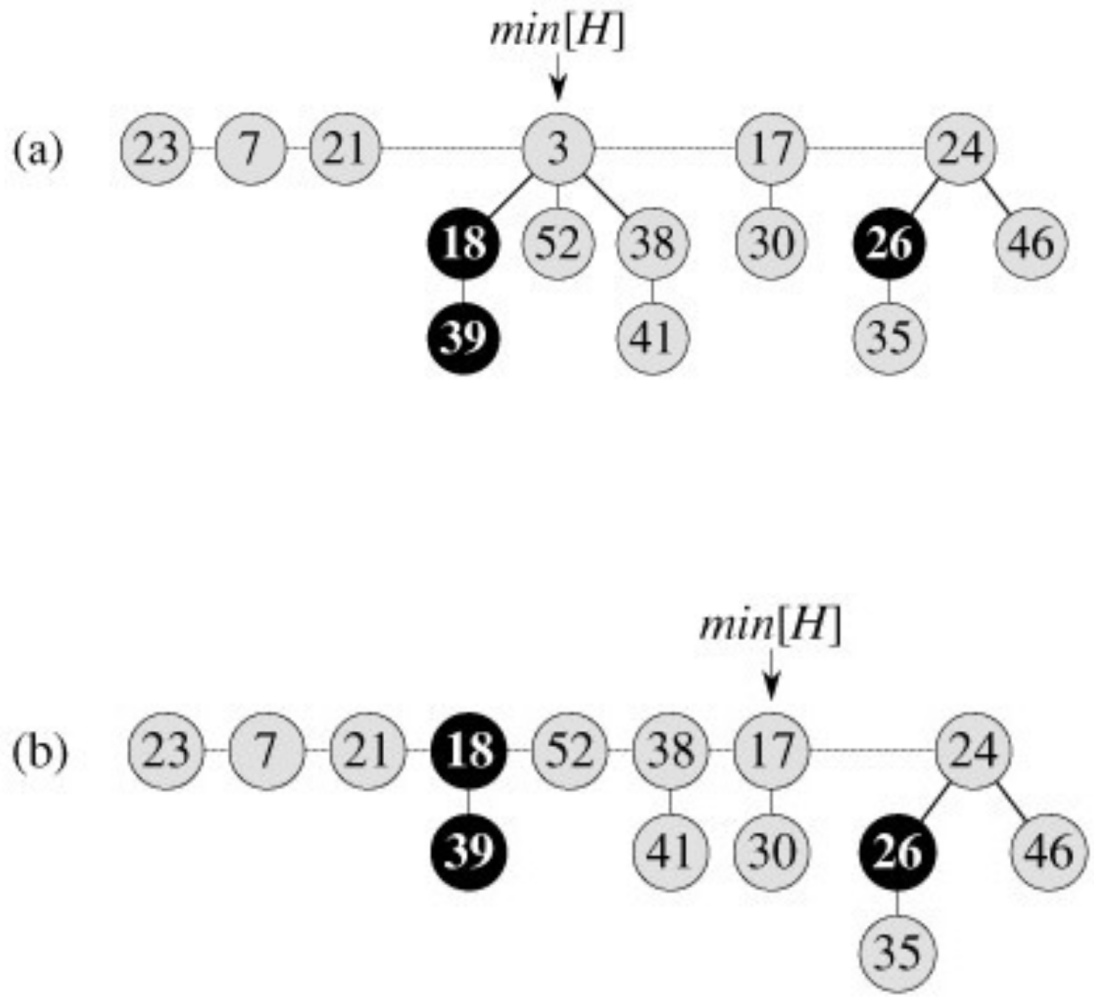
(Union is a lazy operation; no consolidation is done here.)

FIB-HEAP-DELETE-MIN

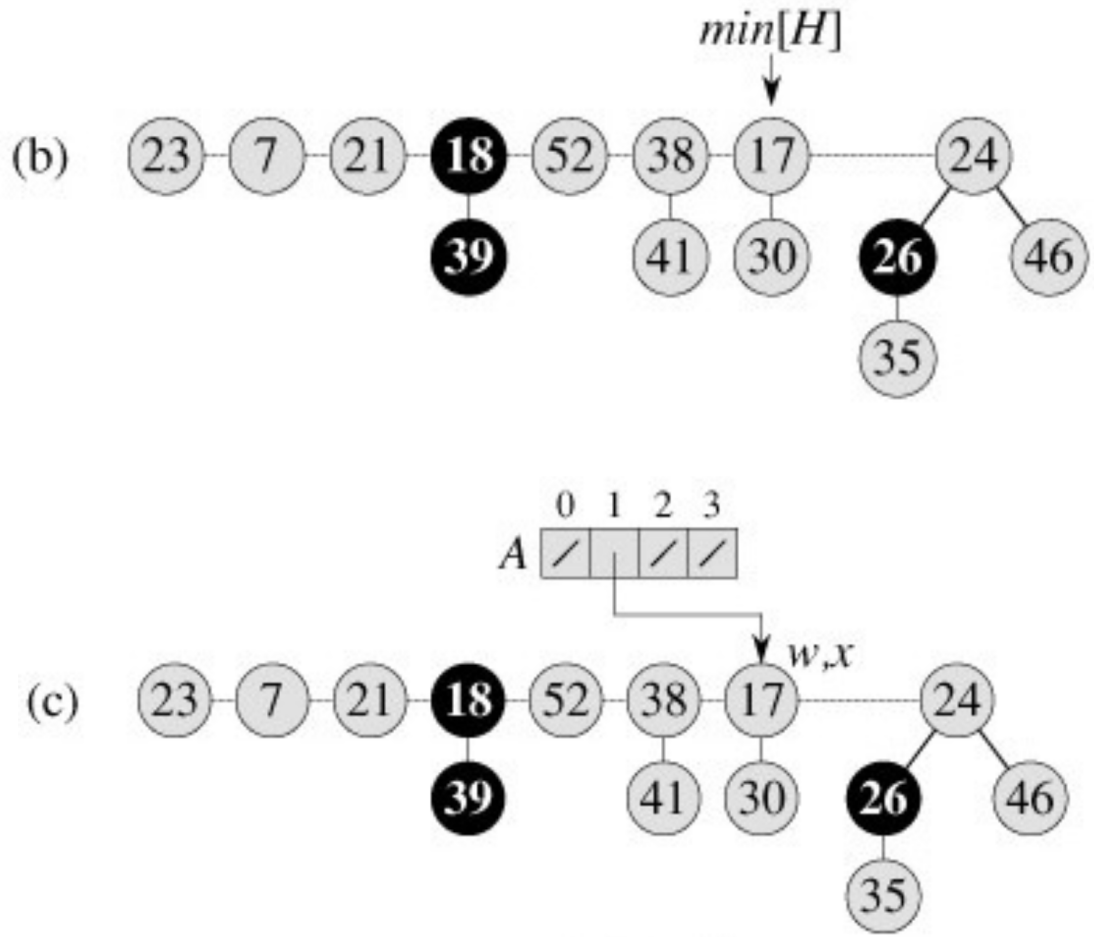
How Delete-Min Works:

- Find the minimum node in the heap.
- Remove the minimum node.
- Take all children of the removed node.
- Add those children directly to the root list.
- Now many trees may have the same degree.
- To fix this, we will perform **consolidation** in the next step.

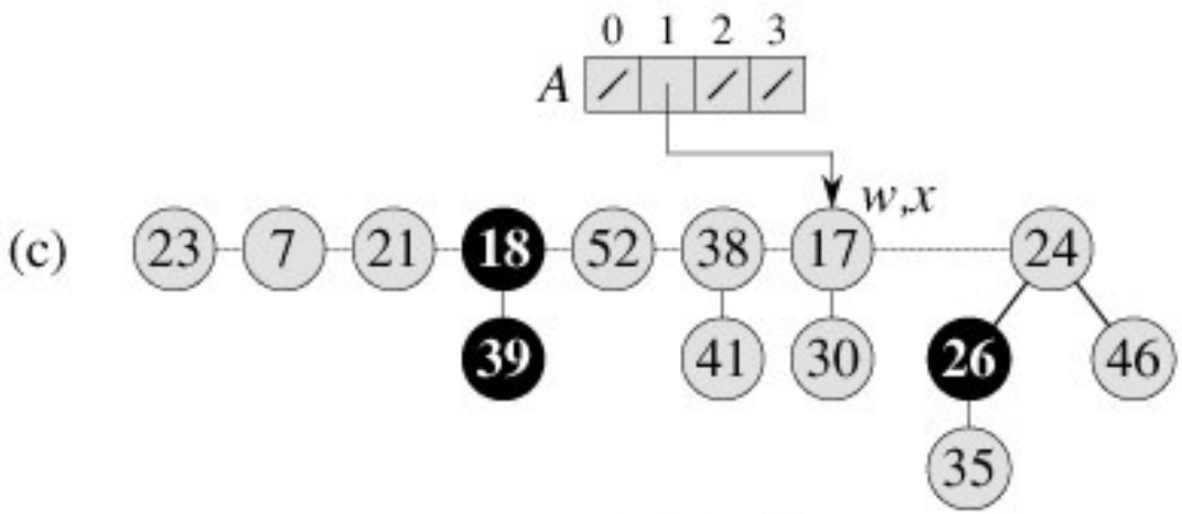
FIB-HEAP-DELETE-MIN

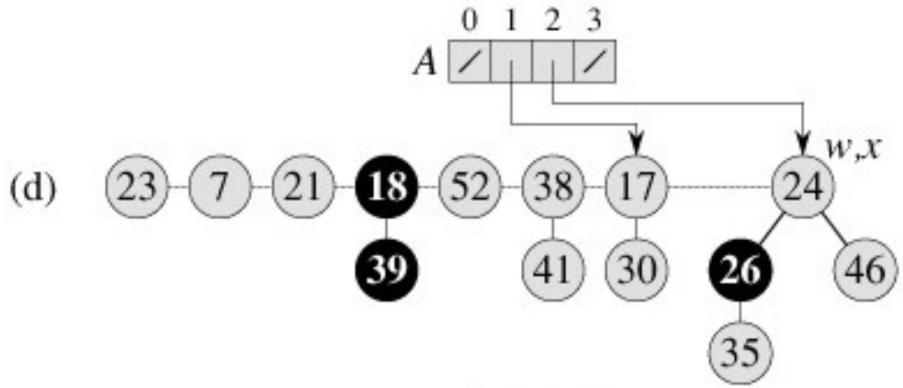


FIB-HEAP-DELETE-MIN

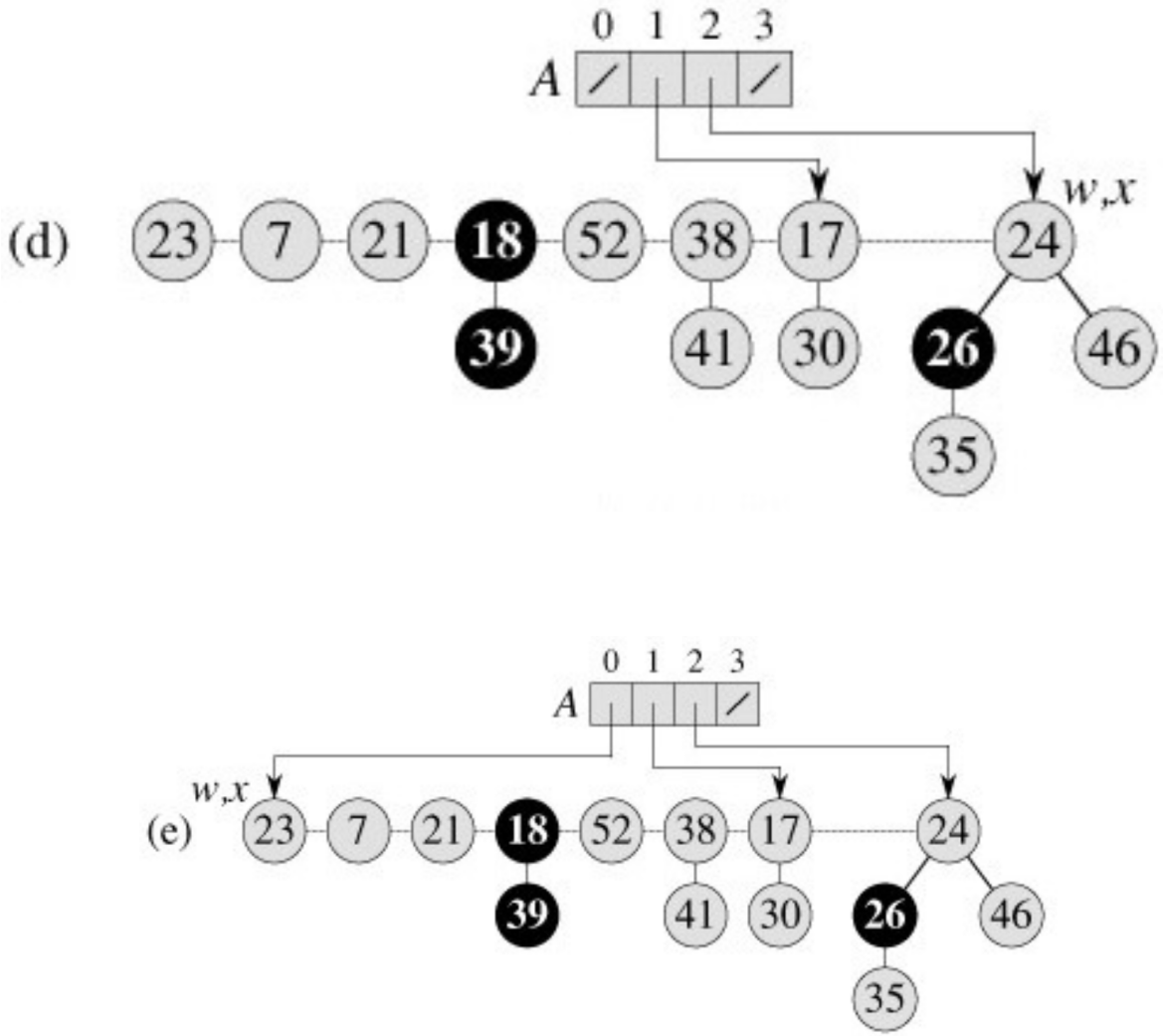


FIB-HEAP-DELETE-MIN

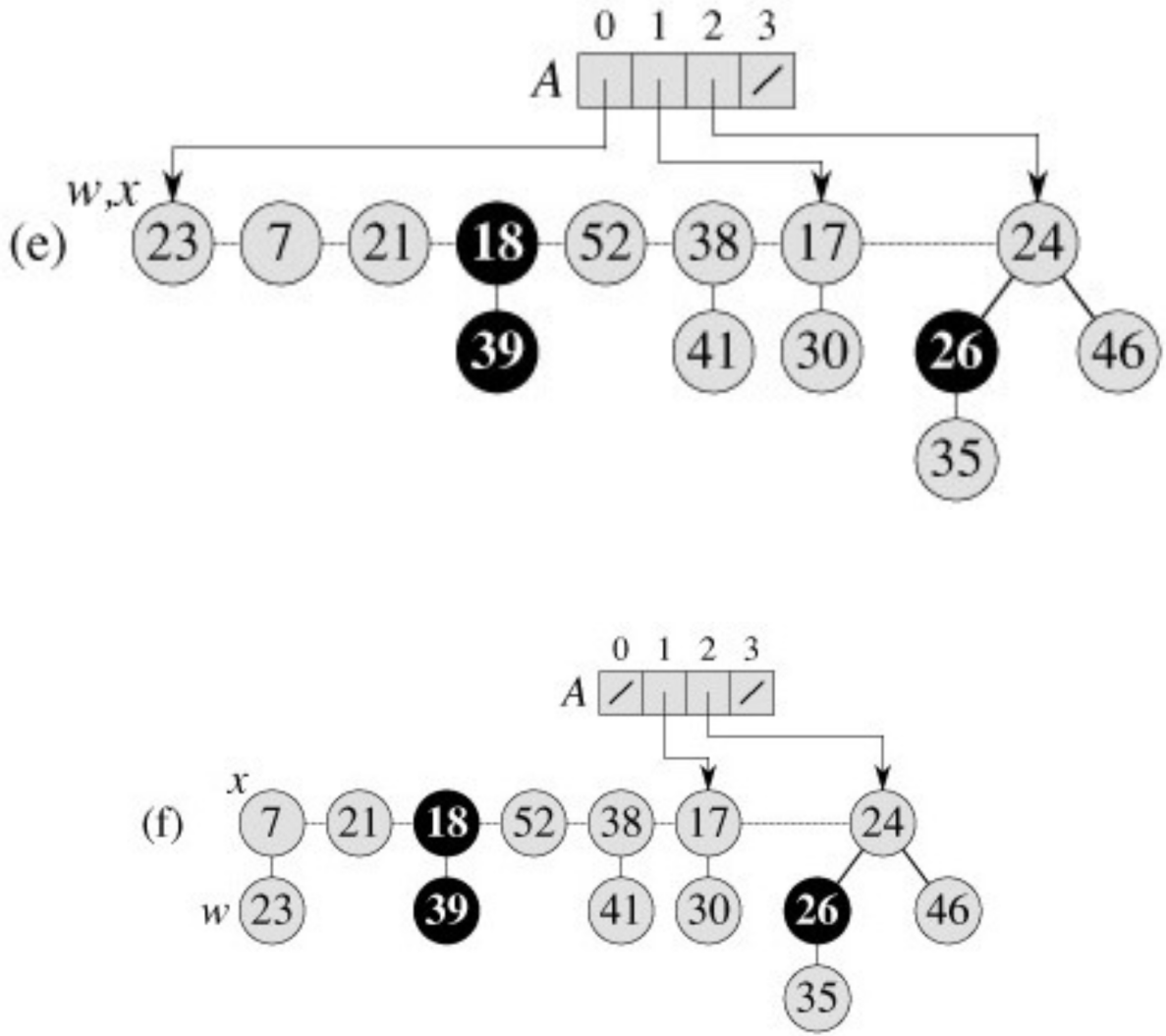




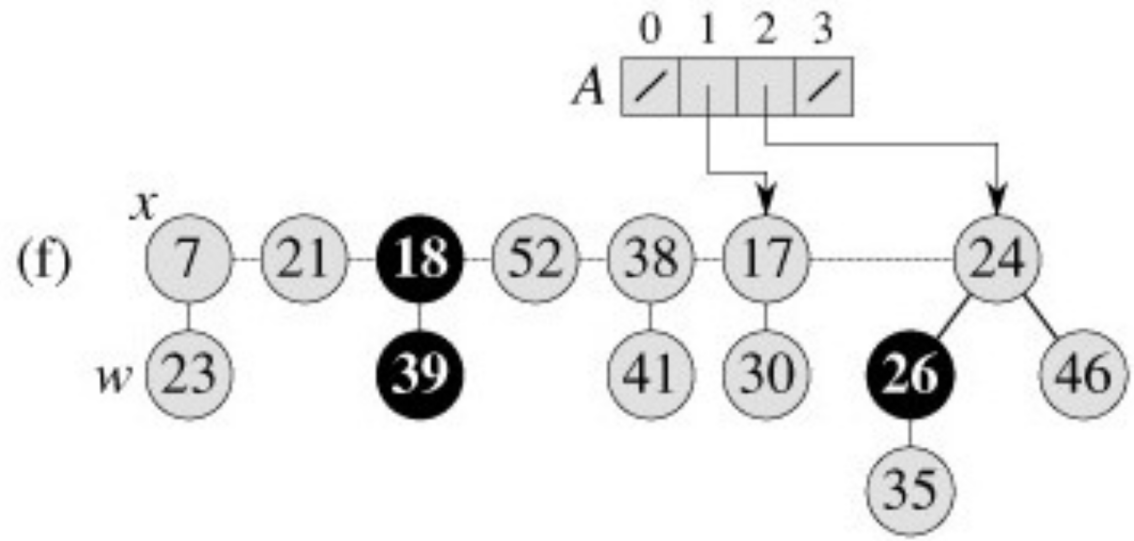
FIB-HEAP-DELETE-MIN

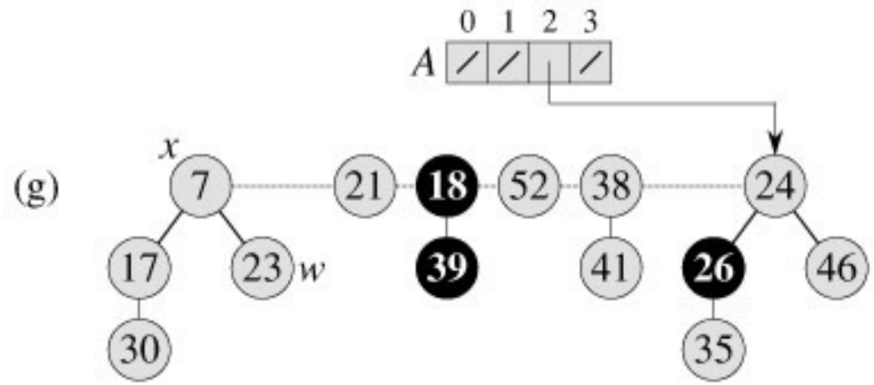


FIB-HEAP-DELETE-MIN

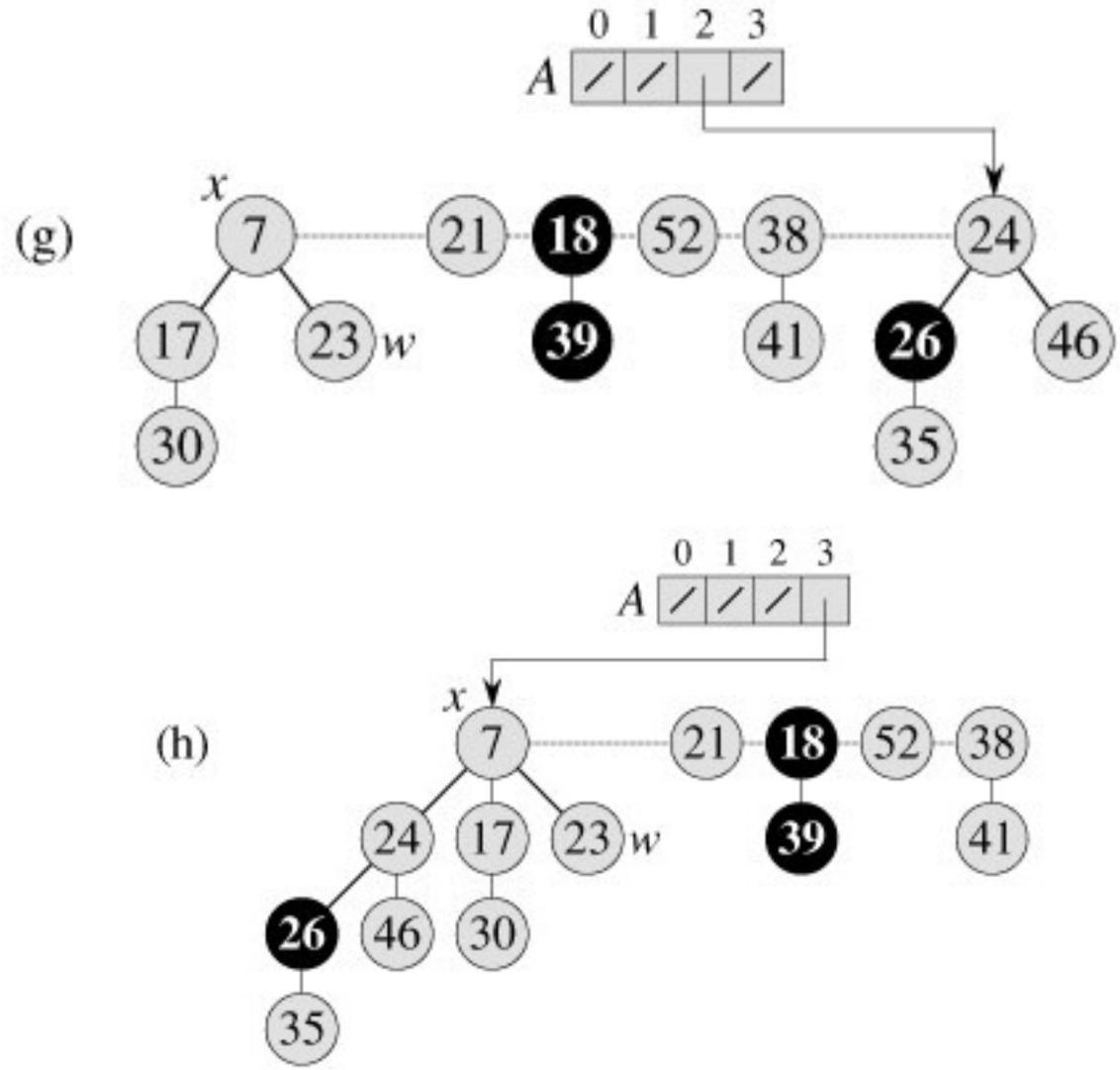


FIB-HEAP-DELETE-MIN

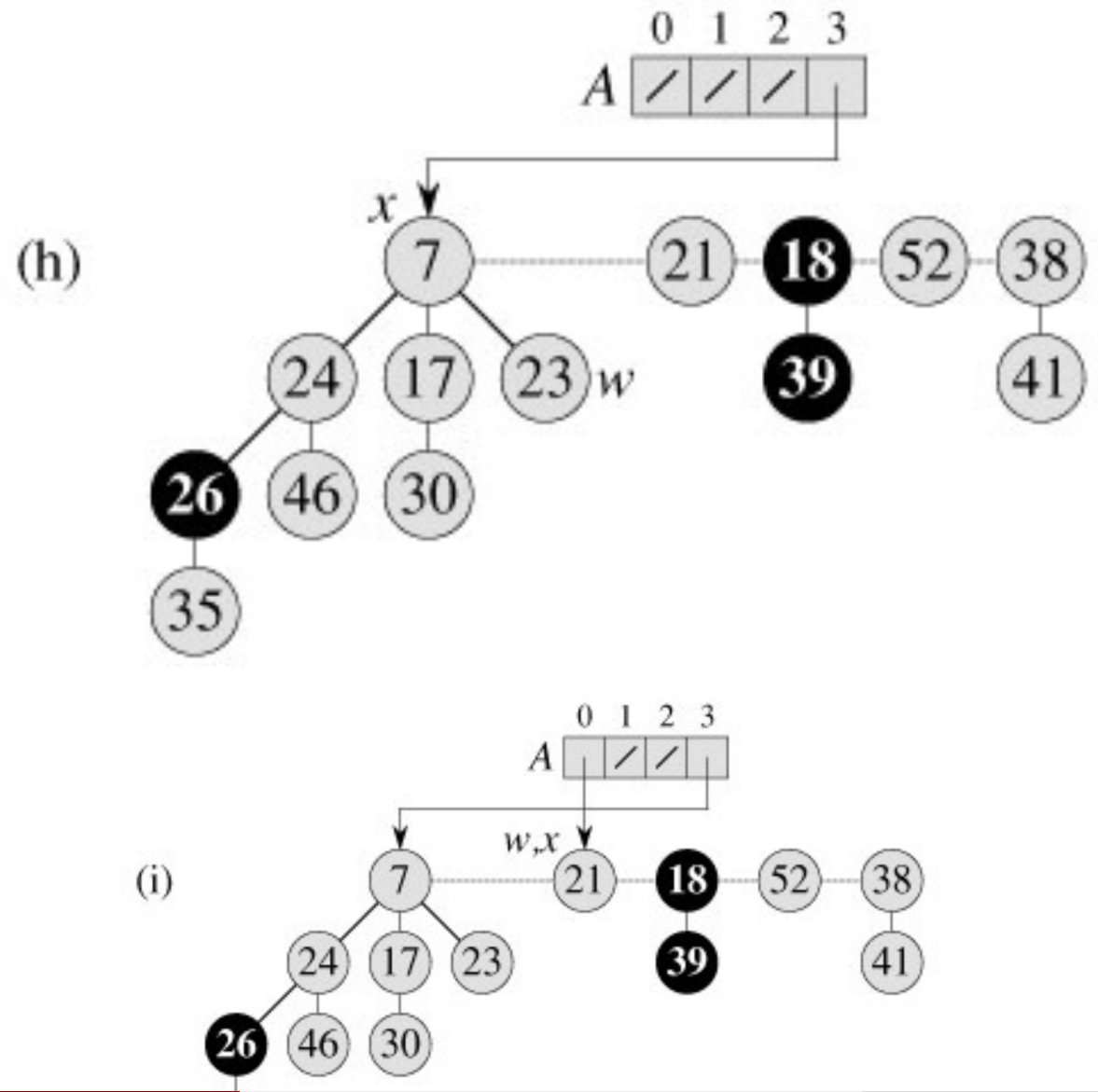




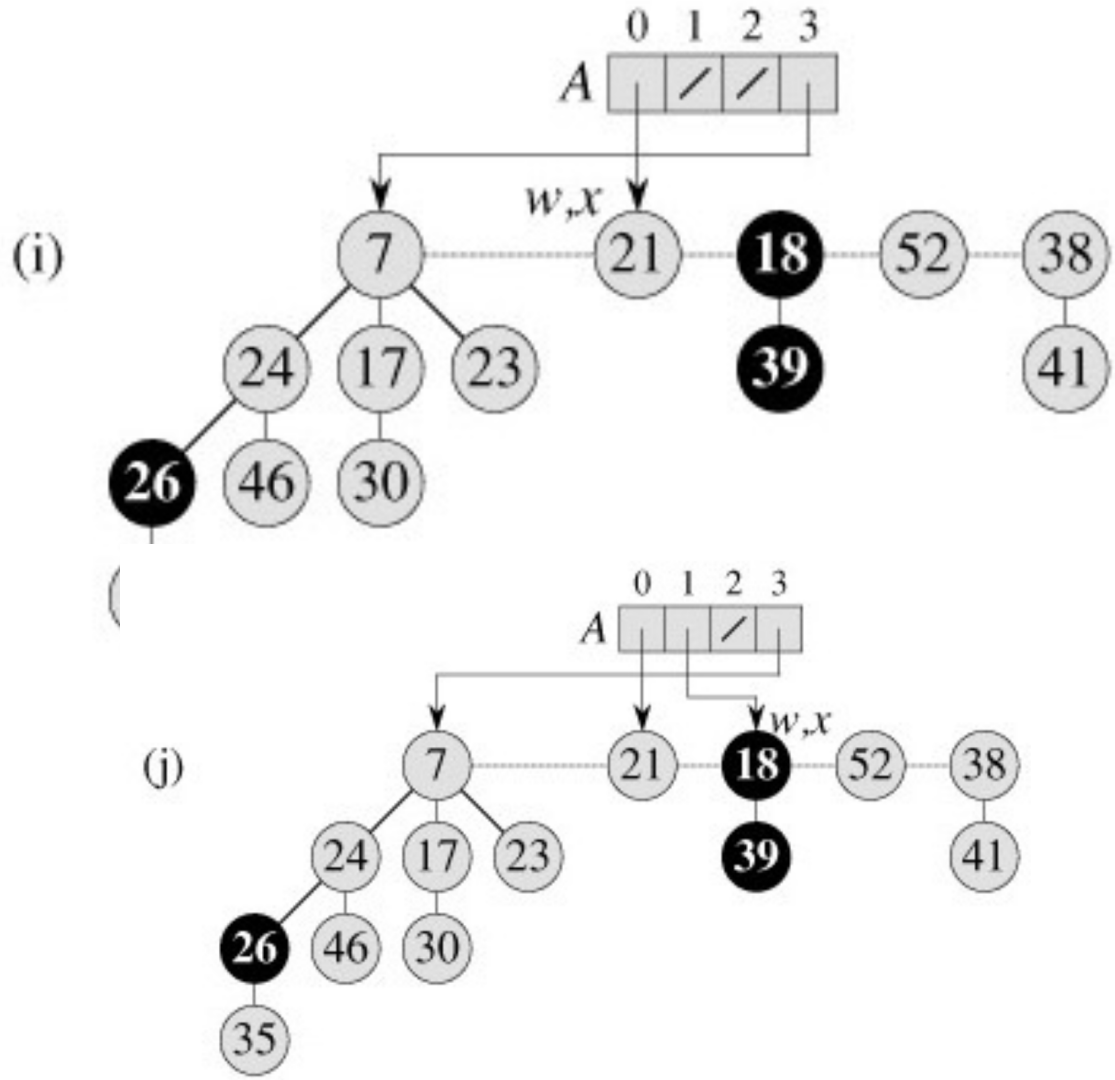
FIB-HEAP-DELETE-MIN



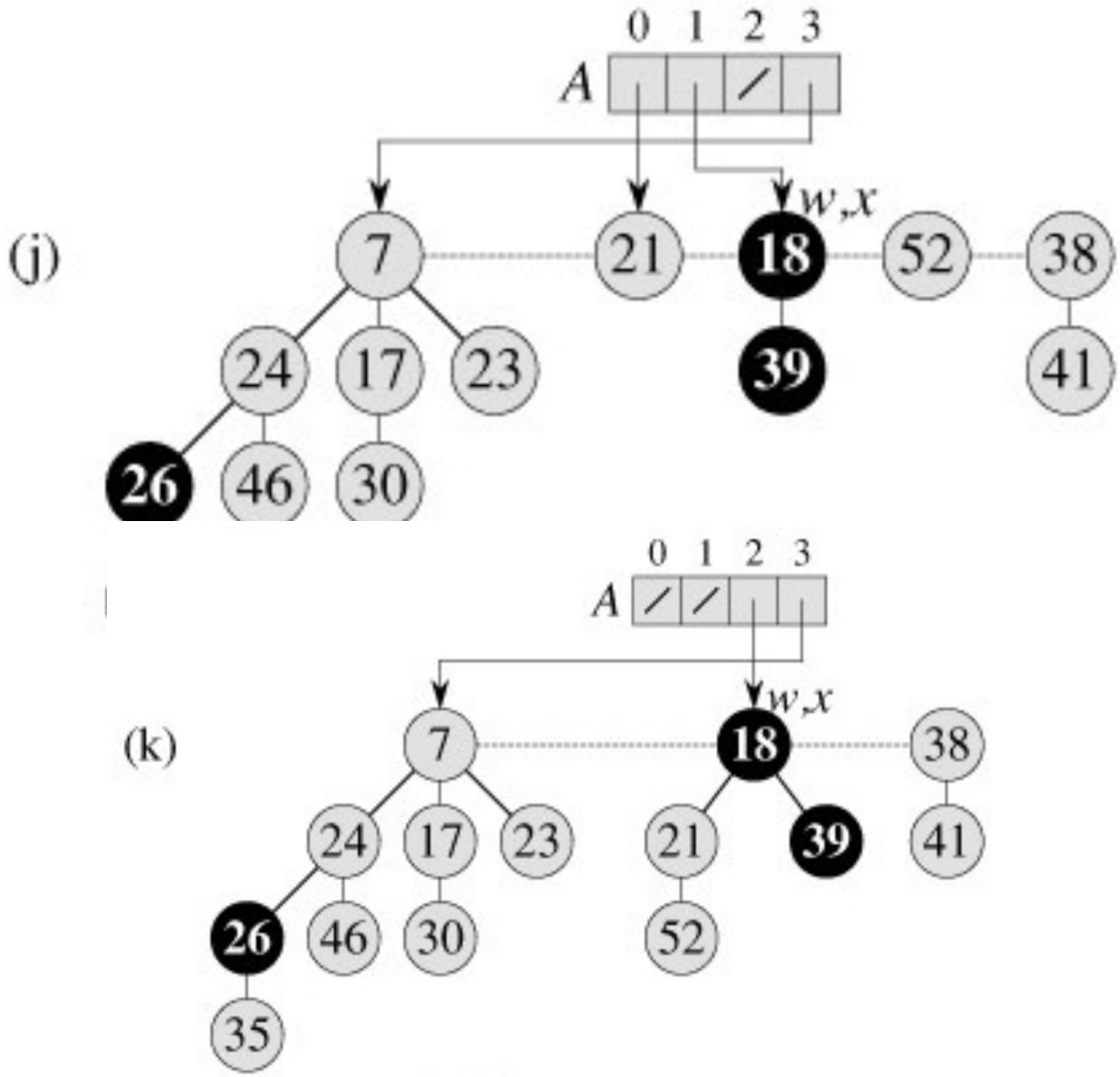
FIB-HEAP-DELETE-MIN



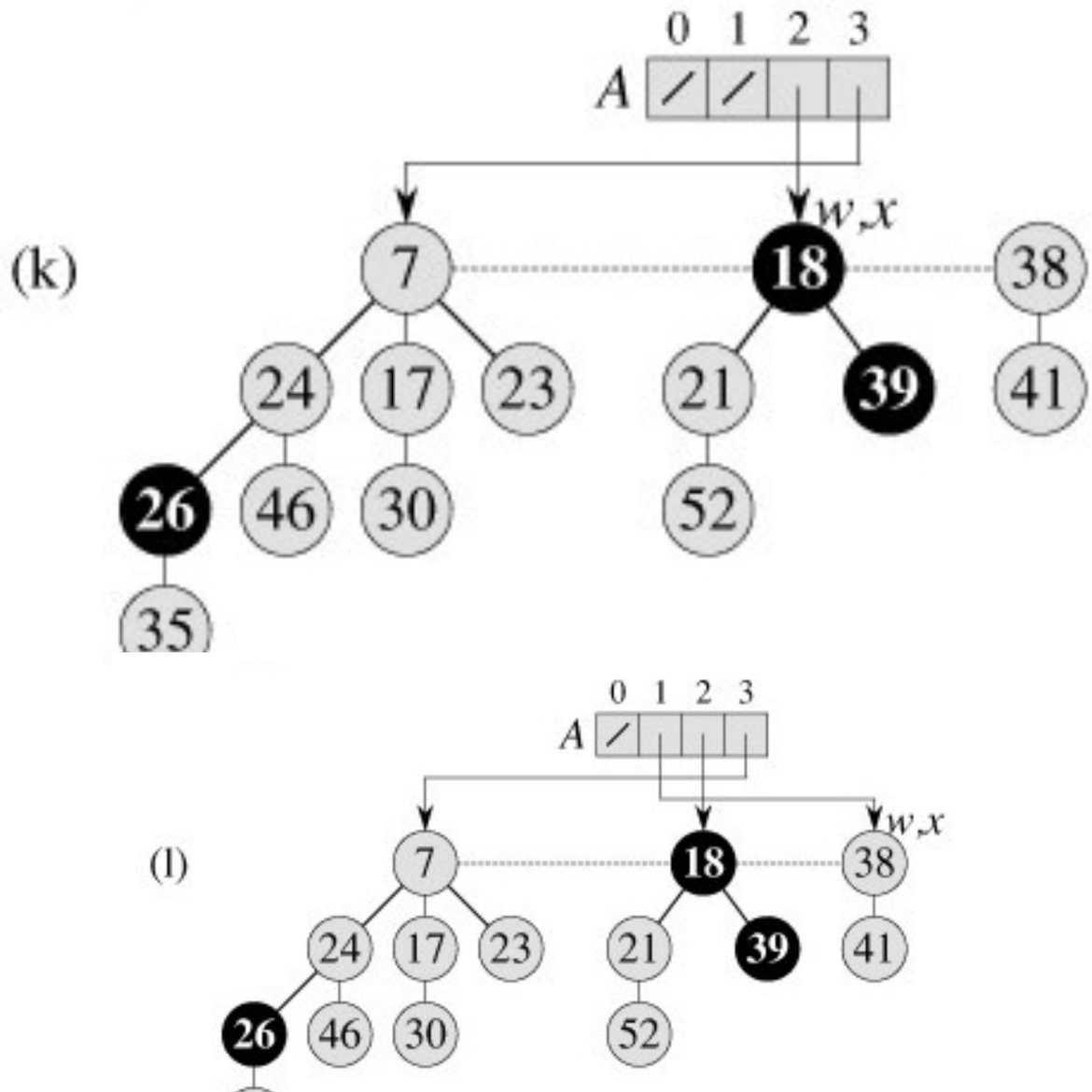
FIB-HEAP-DELETE-MIN



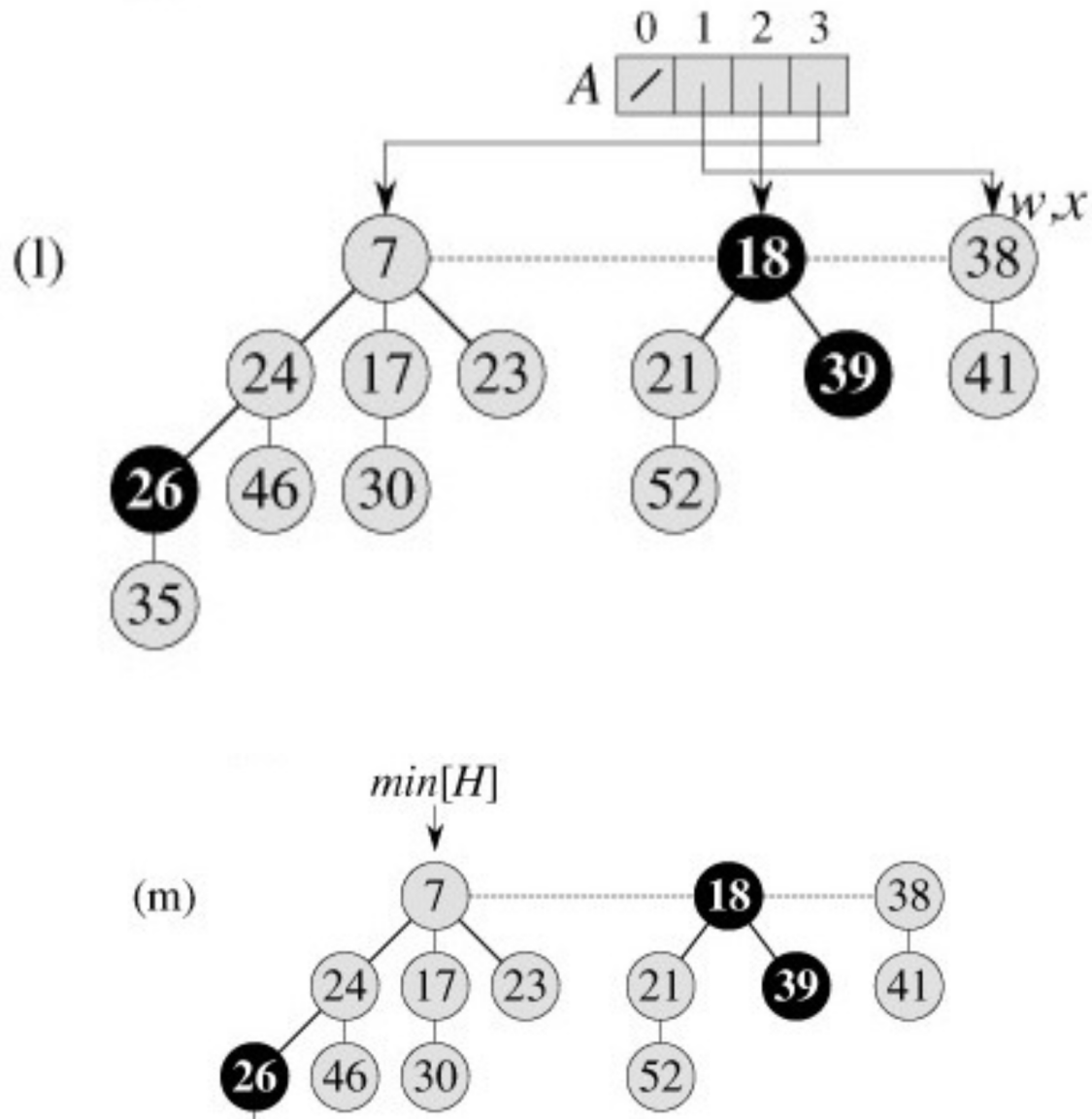
FIB-HEAP-DELETE-MIN



FIB-HEAP-DELETE-MIN



FIB-HEAP-DELETE-MIN



FIB-HEAP-DELETE-MIN: Time Complexity

FIB-HEAP-DELETE-MIN(H) performs:

- Remove the minimum node from the root list.
- Add all its children to the root list.
- Consolidate the trees to ensure no two roots have the same degree.

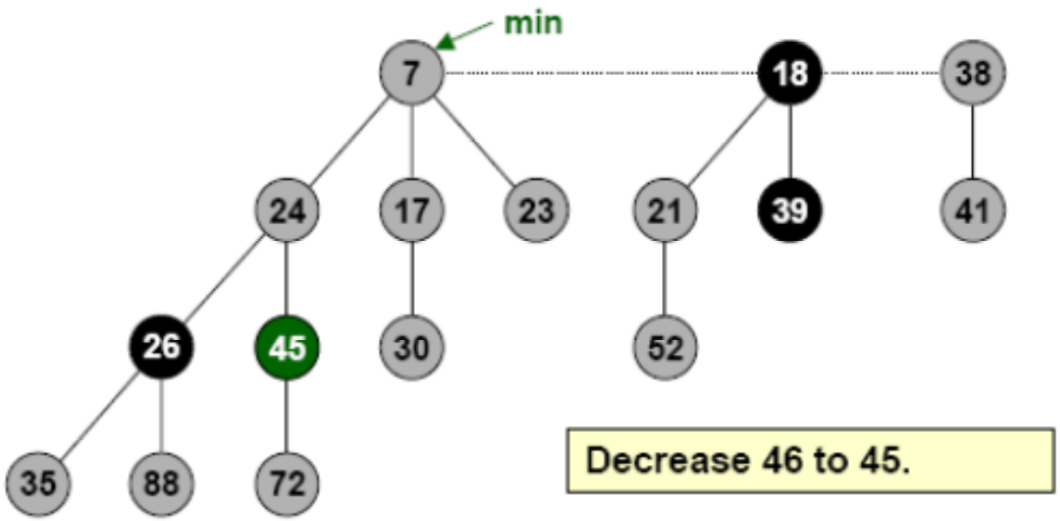
Time Complexity:

$O(\log n)$ (amortized)

(Consolidation dominates the cost by linking trees of equal degree.)

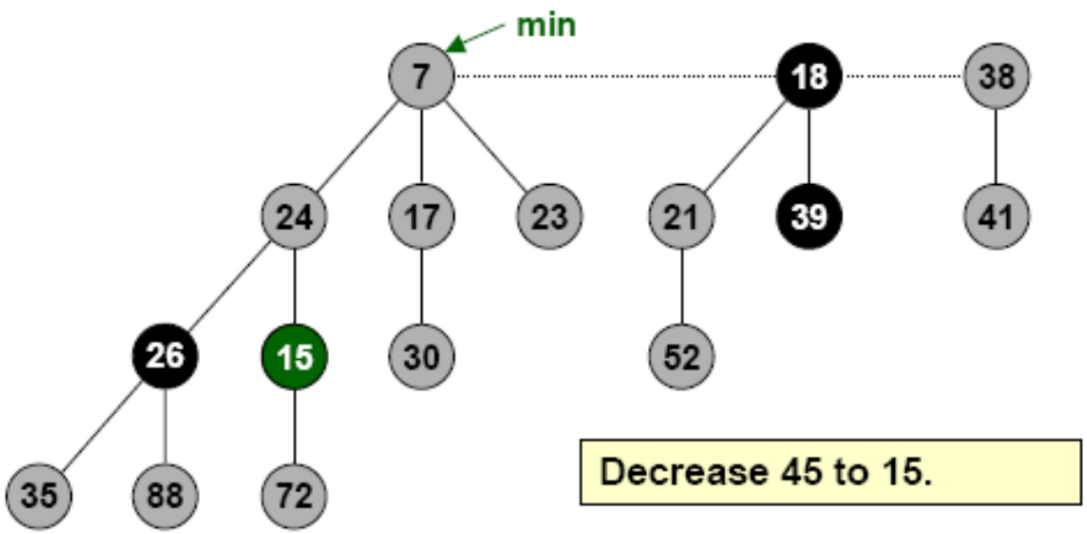
Decrease Key

Case 0



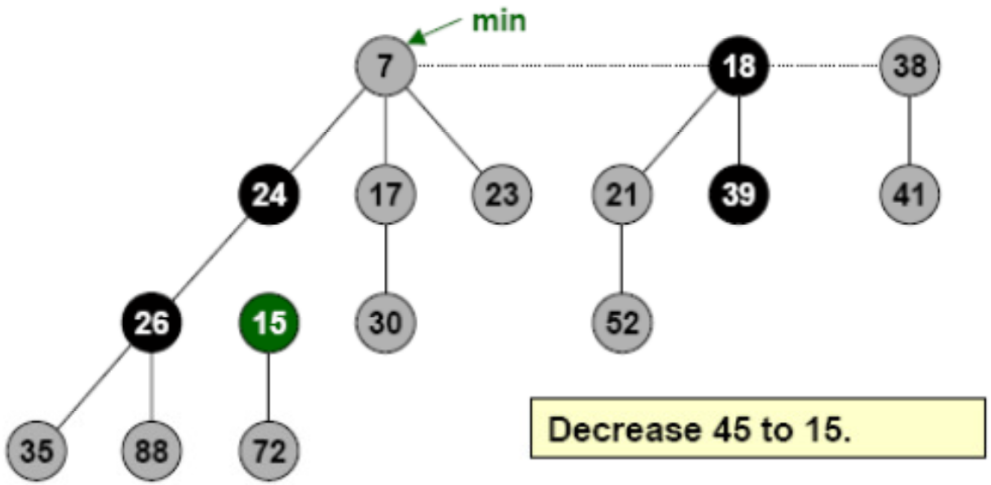
Decrease Key

Case 1



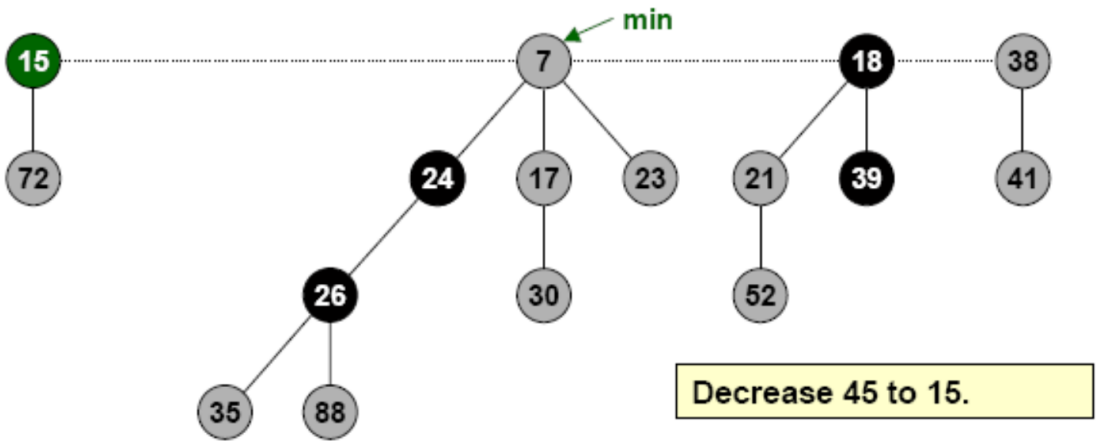
Decrease Key

Case 1



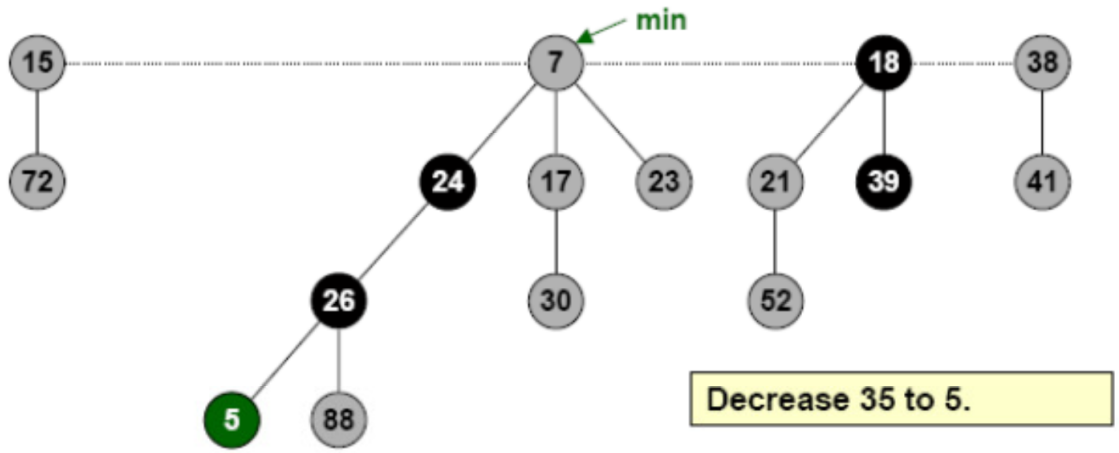
Decrease Key

Case 1

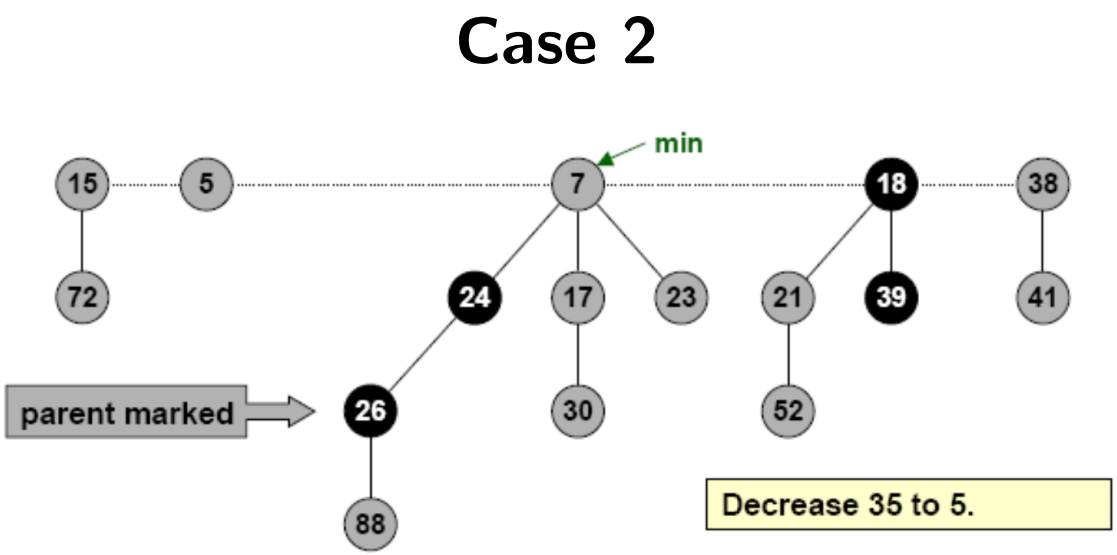


Decrease Key

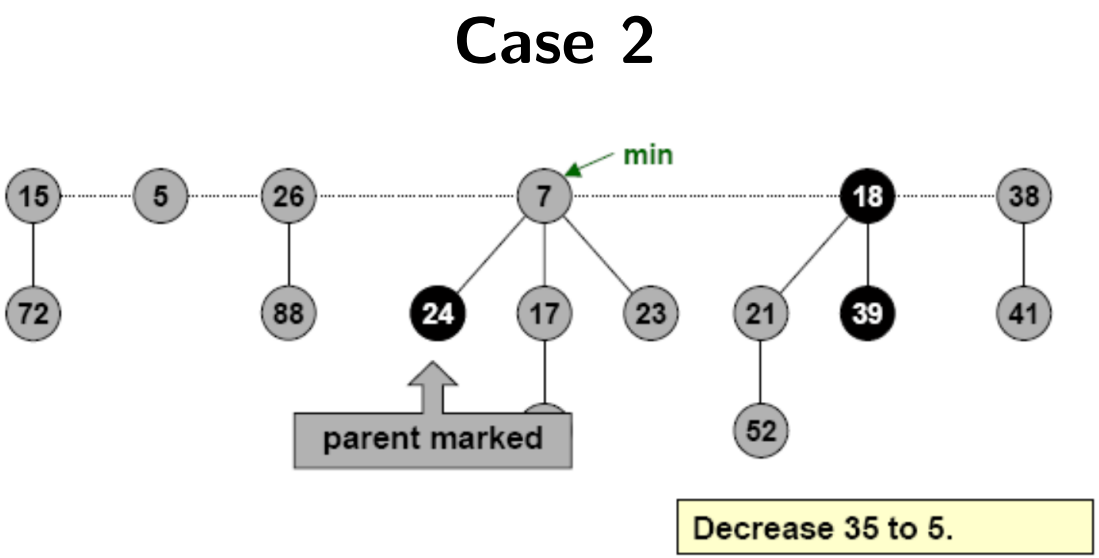
Case 2



Decrease Key

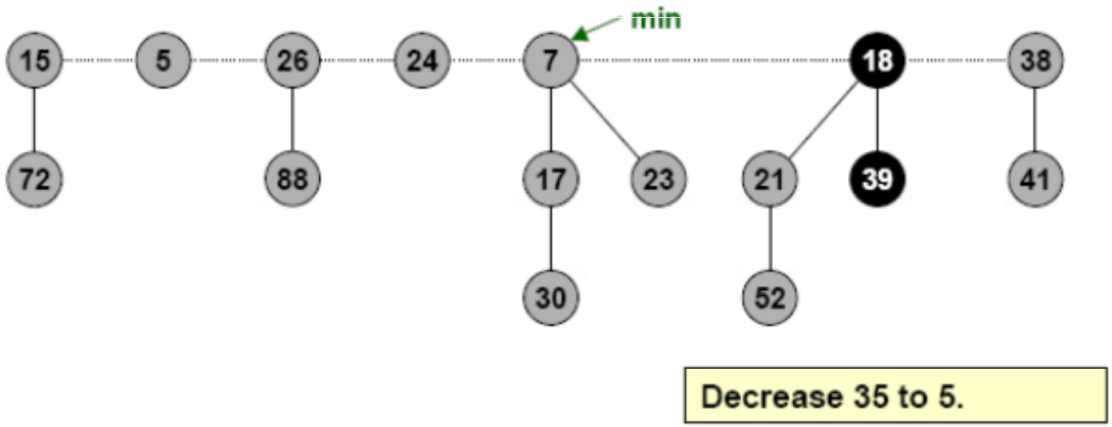


Decrease Key



Decrease Key

Case 2



FIB-HEAP-DECREASE-KEY: Time Complexity

FIB-HEAP-DECREASE-KEY(H, x, k) performs:

- Updates the key of node x to a smaller value.
- If the heap-order property is violated:
  - Cuts x from its parent and moves it to the root list.
  - Performs cascading cuts on ancestors if needed.

Time Complexity:

$O(1)$ (amortized)

(Cascading cuts allow the operation to remain constant time.)

Delete Operation

The **Delete** operation removes a specific node x from a Fibonacci Heap.

- Step 1: Apply **Decrease-Key(x, $-\infty$)** – This makes x become the minimum element.
- Step 2: Perform **Extract-Min()** – This removes x from the heap.

Thus, Delete(x) is implemented using Decrease-Key + Extract-Min.

FIB-HEAP-DELETE: Time Complexity

FIB-HEAP-DELETE(H, x) removes a specific node x by:

- Performing **Decrease-Key(x, $-\infty$)** to move x to the root list.
- Applying **Delete-Min()** to remove it from the heap.

Time Complexity:

$$O(\log n)$$

(Dominated by the cost of Delete-Min after decreasing the key.)

Algorithms That Use Fibonacci Heap

Why Fibonacci Heap Is Used in Algorithms?

- A Fibonacci Heap supports very fast operations.
- Especially useful when an algorithm needs:
  - Many **Decrease-Key** operations
  - Many **Insert** operations
  - Very fast **Union** of heaps
- These features help improve the running time of major graph algorithms.

Major Algorithms That Use Fibonacci Heap

Most Common Algorithms:

1. **Dijkstra's Shortest Path Algorithm**
   - Using Fibonacci Heap makes it faster:

$$O(E + V \log V)$$

   - Because Decrease-Key is $O(1)$ amortized.
2. **Prim's Minimum Spanning Tree Algorithm**
   - Many decrease-key operations $\rightarrow$ Fib Heap gives best performance.
3. **Kruskal's Algorithm (Optimized versions)**
   - Priority queue for edges becomes faster.

Other Uses and Summary

Other Situations Where Fibonacci Heap Helps:

- **Mergeable Heap Operations**
  - UNION operation is only $O(1)$.
  - Useful in algorithms that frequently merge priority queues.
- **Network Optimization Algorithms**
  - Many routing and flow algorithms use Fib Heap for efficiency.
- **Overall Summary**
  - Fibonacci Heaps make algorithms faster
  - Especially when many Decrease-Key operations are needed
  - Great for large graphs and heavy priority queue usage

Time Complexity Analysis of Fibonacci Heap

Time Complexity of Fibonacci Heap Operations

Operation	Amortized Time
Make-Heap	$O(1)$
Insert	$O(1)$
Minimum	$O(1)$
Union (Concatenate)	$O(1)$
Decrease-Key	$O(1)$
Delete	$O(\log n)$
Extract-Min (Delete-Min)	$O(\log n)$

Why Fibonacci Heap Is Fast?

Reason for Low Amortized Time:

- Fibonacci Heap delays expensive operations.
- Structures are kept loose and are fixed only when needed.
- This gives:
  - Insert $\rightarrow O(1)$
  - Union $\rightarrow O(1)$
  - Decrease-Key $\rightarrow O(1)$ (very important in graph algorithms)
- Only Extract-Min and Delete take $O(\log n)$.
- Consolidation step controls the number of trees.

Time Complexity Summary

Key Points to Remember:

- Fibonacci Heap is best when the algorithm needs many **Decrease-Key** operations.
- Extract-Min is the only costly operation:

$$O(\log n)$$

- All other major operations are:

$$O(1) \text{ amortized}$$

- This is why algorithms like Dijkstra and Prim become faster.

Comparative Advantages of Fibonacci Heap

Comparison of Heap Data Structures

Operation	**Binary Heap**	**Binomial Heap**	**Fibonacci Heap**
Make-Heap	$O(1)$	$O(1)$	$O(1)$
Insert	$O(\log n)$	$O(1)$	$O(1)$ amortized
Find-Min	$O(1)$	$O(1)$	$O(1)$
Union	$O(n)$	$O(\log n)$	$O(1)$
Extract-Min	$O(\log n)$	$O(\log n)$	$O(\log n)$ amortized
Decrease-Key	$O(\log n)$	$O(\log n)$	$O(1)$ amortized
Delete	$O(\log n)$	$O(\log n)$	$O(\log n)$ amortized

Applications of Fibonacci Heap

Where Fibonacci Heaps Are Used

- **Graph Algorithms**
  - Dijkstra's Shortest Path Algorithm (fast because of $O(1)$ Decrease-Key)
  - Prim's Minimum Spanning Tree Algorithm (benefits from efficient Decrease-Key)
  - Johnson's Algorithm for All-Pairs Shortest Paths
- **Network Optimization**
  - Fast routing and path discovery
  - Used in telecom and transport networks
- **Advanced Priority Queue Applications**
  - Event-driven simulation
  - Task scheduling
  - CPU scheduling systems
- **Data Structure Research**
  - Used to study amortized algorithms
  - Helps compare theoretical heap models

Conclusion

Summary of Fibonacci Heap

- Fibonacci Heap is an advanced priority queue supporting very fast amortized operations.
- Its main strength comes from:
  - Flexible tree structure
  - Lazy operations
  - Efficient **Decrease-Key** and **Union** operations
- Compared to Binary and Binomial Heaps, Fibonacci Heap offers:
  - Better amortized performance
  - Ideal use in graph algorithms with frequent Decrease-Key calls
- Widely used in:
  - Dijkstra's Algorithm
  - Prim's Algorithm
  - Advanced network routing systems
- Although complex to implement, its theoretical efficiency makes it important in algorithm design.