ADVANCED DATA STRUCTURES

24CS2255F

CO2: Priority Queues & Heaps

1. Implementation of Priority Queues

A priority queue is a data structure in which each element is associated with a priority, and elements are served according to their priority rather than the order in which they were inserted. An element with higher priority is processed before an element with lower priority. If two elements have the same priority, they are usually processed in first-in–first-out (FIFO) order.

There are two common types of priority queues: max-priority queue, where the element with the highest value or priority is removed first, and min-priority queue, where the element with the lowest value or priority is removed first. Priority queues support basic operations such as insert, delete (remove highest/lowest priority element), and peek (view the highest-priority element).

Priority queues are commonly implemented using heaps, which allow insertion and deletion operations to be performed efficiently in O(log n) time. They are widely used in real-world applications such as CPU scheduling, Dijkstra's shortest path algorithm, job scheduling, event-driven simulations, and network packet routing.

a priority queue is an efficient data structure used when elements must be processed based on importance rather than arrival order.

Why priority queue:

A priority queue is used because in many real-world and computing situations not all tasks are equally important, and they must be processed based on priority rather than arrival time. It is needed to efficiently manage and retrieve the most important element at any moment. For example, in CPU scheduling, urgent processes must be executed before normal ones; in Dijkstra's algorithm, the node with the smallest distance must always be selected next; and in emergency systems, critical tasks must be handled first. Using a normal queue (FIFO) cannot support this behavior. Priority queues also provide better performance. With heap-based implementation, insertion and deletion take only O(log n) time, which is much faster than repeatedly searching through a list. This makes priority queues suitable for large datasets and time-critical applications. In short, a priority queue is used to ensure correct ordering based on importance, improve efficiency, and support priority-based decision making in algorithms and systems.

A priority queue can be implemented using arrays, linked lists, or heaps. In an unsorted array, insertion is fast but deletion is slow. In a sorted array or linked list, insertion takes more time but

deletion is efficient. The most efficient and commonly used implementation is using a heap, where both insertion and deletion operations take logarithmic time. Python provides heap support through the heapq module, making heap-based priority queues practical and efficient for real-world applications.

Ways to implement a Priority Queue:

1. Using Unsorted Array / List
2. Using Sorted Array / List
3. Using Heap (Binary Heap – Min Heap / Max Heap)
4. Using Linked List
5. Using Balanced Binary Search Tree (BST / AVL / Red-Black Tree)
6. Using Deque (Double Ended Queue – limited cases)

1. Using Unsorted Array / List

In a **priority queue**, each element has a priority and elements are processed based on priority rather than insertion order. When a **priority queue is implemented using an unsorted array or list**, elements are inserted without maintaining any order. The **highest priority element is searched and removed during deletion**.

- **Insertion is fast** because elements are simply appended.
- **Deletion is slow** because the entire list must be searched to find the highest priority element.

This method is simple to implement and suitable when **insert operations are more frequent than deletions**.

Algorithm: Priority Queue using Unsorted List (Max-Priority Queue)

Insert Operation

1. Add the element at the end of the list.

Delete Operation

1. Search the list to find the element with the highest priority.
2. Remove that element from the list.
3. Return the removed element.

Algorithm Steps (General)

1. **Start**
2. Initialize an empty list PQ.
3. To insert an element, append it to PQ.
4. To delete:
    - Find the maximum element in PQ.
    - Remove it.
5. **Stop**

A priority queue implemented using an unsorted array stores elements without maintaining any order. Insertion is performed in constant time by appending elements to the list, while deletion requires scanning the entire list to find and remove the highest priority element. Although this approach is easy to implement, it is inefficient for frequent deletions due to linear search time. Therefore, this method is suitable only when insert operations dominate over delete operations.

Example:

Input

Number of elements = 10

Elements: 15 40 10 60 25 55 5 30 45 20

(Assume **larger value = higher priority**)

Step-by-Step Insertion (Unsorted List)

Step	Inserted Element	Priority Queue Contents
1	15	[15]
2	40	[15, 40]
3	10	[15, 40, 10]
4	60	[15, 40, 10, 60]
5	25	[15, 40, 10, 60, 25]
6	55	[15, 40, 10, 60, 25, 55]
7	5	[15, 40, 10, 60, 25, 55, 5]
8	30	[15, 40, 10, 60, 25, 55, 5, 30]
9	45	[15, 40, 10, 60, 25, 55, 5, 30, 45]
10	20	[15, 40, 10, 60, 25, 55, 5, 30, 45, 20]

Step-by-Step Deletion Table

Delete Highest Priority Elements One by One

Deletion Step	Queue Before Deletion	Highest Priority Element	Queue After Deletion
1	[15, 40, 10, 60, 25, 55, 5, 30, 45, 20]	**60**	[15, 40, 10, 25, 55, 5, 30, 45, 20]
2	[15, 40, 10, 25, 55, 5, 30, 45, 20]	**55**	[15, 40, 10, 25, 5, 30, 45, 20]
3	[15, 40, 10, 25, 5, 30, 45, 20]	**45**	[15, 40, 10, 25, 5, 30, 20]
4	[15, 40, 10, 25, 5, 30, 20]	**40**	[15, 10, 25, 5, 30, 20]

5	[15, 10, 25, 5, 30, 20]	**30**	[15, 10, 25, 5, 20]
6	[15, 10, 25, 5, 20]	**25**	[15, 10, 5, 20]
7	[15, 10, 5, 20]	**20**	[15, 10, 5]
8	[15, 10, 5]	**15**	[10, 5]
9	[10, 5]	**10**	[5]
10	[5]	**5**	[ ]

Important Observation:

- At **each deletion**, the entire list is scanned to find the **maximum element**.
- Deletion order is:
  60 → 55 → 45 → 40 → 30 → 25 → 20 → 15 → 10 → 5

Key Points

- **Insertion:** O(1)
- **Deletion:** O(n)
- Data structure used: **Unsorted Array/List**
- Priority rule: **Larger value = higher priority**

Python Code:

```
class PriorityQueue:
    def __init__(self):
        self.q = []

    def insert(self, element):
        self.q.append(element)

    def delete(self):
        if not self.q:
            print("Queue is empty")
            return None
        highest = max(self.q)
        self.q.remove(highest)
        return highest

    def display(self):
        print("Current Queue:", self.q)


# Driver code
pq = PriorityQueue()

# Input size of elements
```

```
size = int(input("Enter the number of elements: "))

# Input elements from user
elements = []
for i in range(size):
    elem = int(input(f"Enter element {i+1}: "))
    elements.append(elem)

# Insert elements into the priority queue
for e in elements:
    pq.insert(e)

# Delete two elements to show functionality
print("Deleted element:", pq.delete())
print("Deleted element:", pq.delete())

# Display remaining elements
pq.display()
```

2. Using Sorted Array / List

A Priority Queue using a Sorted Array / List is a linear data structure where elements are maintained in a sorted order based on their priority. In this implementation, the highest-priority element is always at the front of the list, making it very easy and fast to remove it. When a new element is inserted, the list is traversed to find the correct position so that the descending order is maintained, ensuring that the largest (or highest-priority) element is always accessible. This approach provides a simple way to maintain priorities without using complex data structures like heaps.

The main advantage of this method is that deletion of the highest-priority element takes constant time (O(1)), since it is always at the front of the list. However, the insertion operation can be slower, with a time complexity of O(n) in the worst case, because the new element may need to be placed somewhere in the middle or end of the list. Overall, the sorted array/list implementation is effective for situations where deletions are more frequent than insertions, and it provides a clear, step-by-step ordering of elements according to priority, which is easy to visualize and implement.

Example:

Input Size=10
elements = [15, 40, 10, 60, 25, 55, 5, 30, 45, 20]
We'll create a Priority Queue using a Sorted Array / List, show step-by-step insertion and deletion tables, write the algorithm, and provide a Python code version that can take input from the user.

Step 1: Insertion Table (Descending Order)

Step	Insert Element	Queue State (Descending)	Description
1	15	[15]	Queue is empty, insert 15
2	40	[40, 15]	Insert 40 at correct position
3	10	[40, 15, 10]	Insert 10 at correct position

4	60	[60, 40, 15, 10]	Insert 60 at front (highest)
5	25	[60, 40, 25, 15, 10]	Insert 25 at correct position
6	55	[60, 55, 40, 25, 15, 10]	Insert 55 at correct position
7	5	[60, 55, 40, 25, 15, 10, 5]	Insert 5 at end
8	30	[60, 55, 40, 30, 25, 15, 10, 5]	Insert 30 at correct position
9	45	[60, 55, 45, 40, 30, 25, 15, 10, 5]	Insert 45 at correct position
10	20	[60, 55, 45, 40, 30, 25, 20, 15, 10, 5]	Insert 20 at correct position

---

Step 2: Deletion Table

Step	Delete Operation	Queue State After Deletion	Deleted Element	Description
1	Delete highest	[55, 45, 40, 30, 25, 20, 15, 10, 5]	60	Remove first element
2	Delete highest	[45, 40, 30, 25, 20, 15, 10, 5]	55	Remove first element again

Remaining queue: [45, 40, 30, 25, 20, 15, 10, 5]

Algorithm:

Algorithm Steps

1. Initialize Queue

- Create an empty list pq to store elements.

2. Insert Element (element)

1. If pq is empty, append element.
2. Else, traverse pq from start:
    - Find the first element smaller than element.
    - Insert element at that position.
3. If no smaller element is found, append element at the end.

Logic: Keep the list **sorted in descending order** so the **largest element is always at the front**.
Time Complexity: O(n) for insertion.

3. Delete Highest-Priority Element

1. If pq is empty, print "Queue is empty" and return None.
2. Else, remove the first element of pq.
3. Return the removed element.

Logic: The **first element is always the highest-priority**.
Time Complexity: O(1) for deletion.

4. Display Queue

- Print the current pq list.

Logic: Shows the current queue state.

Python Code:

```
class PriorityQueueSorted:
    def __init__(self):
        self.q = []

    def insert(self, element):
        # Insert element in sorted order (descending)
        if not self.q:
            self.q.append(element)
        else:
            inserted = False
            for i in range(len(self.q)):
                if element > self.q[i]:
                    self.q.insert(i, element)
                    inserted = True
                    break
            if not inserted:
                self.q.append(element)

    def delete(self):
        if not self.q:
            print("Queue is empty")
            return None
        # Highest priority element is at front
        return self.q.pop(0)

    def display(self):
        print("Current Queue:", self.q)


# Driver code
pq = PriorityQueueSorted()

# Input number of elements
size = int(input("Enter the number of elements: "))

# Input elements
for i in range(size):
    elem = int(input(f"Enter element {i+1}: "))
    pq.insert(elem)
    print(f"After insertion of {elem}: {pq.q}")  # Step by step insertion

# Delete two highest-priority elements
print("Deleted element:", pq.delete())
print("Deleted element:", pq.delete())

# Display remaining queue
pq.display()
```

3 Priority Queue Using Heap (Binary Heap – Min Heap / Max Heap)

A Priority Queue is an abstract data type where each element is associated with a priority, and elements are removed based on priority rather than insertion order. A Binary Heap is the most efficient and commonly used data structure to implement a priority queue. A binary heap is a complete binary tree that satisfies the heap property. There are two types of binary heaps:

- **Max Heap: Parent node ≥ its children → highest priority at the root**
- **Min Heap: Parent node ≤ its children → lowest priority at the root**

In heap-based priority queues, the root element is always the highest (or lowest) priority, enabling efficient access and deletion.

Heap Properties

1. Complete Binary Tree
   - All levels are completely filled except possibly the last.
2. Heap Order Property
   - Max Heap: Parent ≥ children
   - Min Heap: Parent ≤ children
3. Array Representation
   - Parent index = $\lfloor (i-1)/2 \rfloor$
   - Left child = $2i + 1$
   - Right child = $2i + 2$

Max Heap

Algorithm: Max Heap (Priority Queue)

Insert Operation (Max Heap)

1. Insert the new element at the end of the heap.
2. Set i to the index of the inserted element.
3. While i > 0 and heap[parent(i)] < heap[i]:
   - Swap heap[i] and heap[parent(i)]
   - Update i = parent(i)
4. Stop when heap property is satisfied.

Logic:
Move the element upward until parent is larger.

Delete Operation (Max Heap)

1. If heap is empty, print “Queue is empty”.
2. Store the root element (maximum).
3. Replace root with the last element.
4. Remove the last element.
5. Heapify down from root:
    - Compare with left and right children
    - Swap with the largest child
6. Repeat until heap property is restored.
7. Return the deleted element.

Example -1 Tree representation:

A **Max Heap** is a **complete binary tree** where:

- Each parent node ≥ its children
- Maximum element is always at the **root**

Given Elements (Insertion Order)
[15, 40, 10, 60, 25, 55, 5, 30, 45, 20]

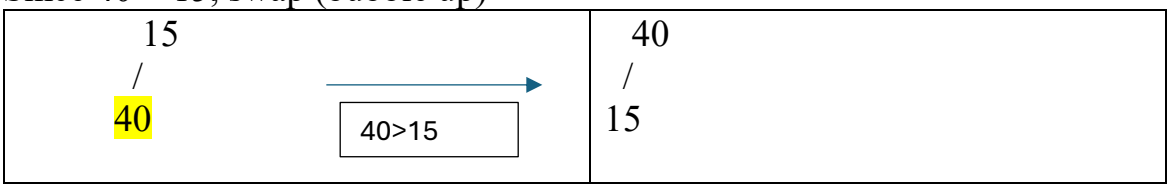
Step 1: Insert 15

- Heap is empty
- 15 becomes the root

15
Heap array: [15]

Step 2: Insert 40

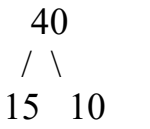
- Insert at next available position (left child of 15)
- Since 40 > 15, swap (bubble up)



Heap array: [40, 15]

Step 3: Insert 10

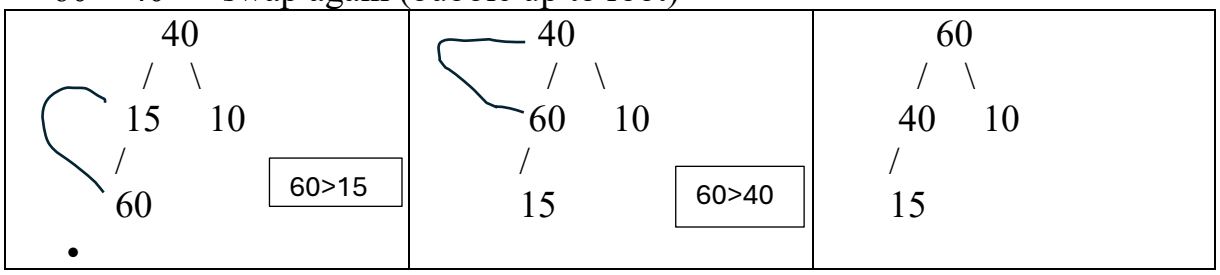
- Insert as right child of 40
- 10 < 40, no swap



Heap array: [40, 15, 10]

Step 4: Insert 60

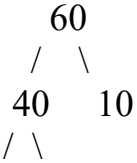
- Insert as left child of 15
- 60 > 15 → swap
- 60 > 40 → swap again (bubble up to root)



Heap array: [60, 40, 10, 15]

Step 5: Insert 25

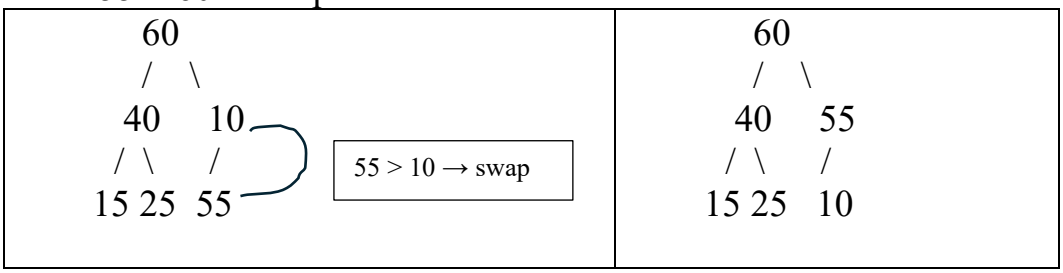
- Insert as right child of 40
- 25 < 40, heap property satisfied



Heap array: [60, 40, 10, 15, 25]

Step 6: Insert 55

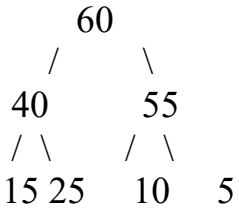
- Insert as left child of 10
- 55 > 10 → swap
- 55 < 60 → stop



Heap array: [60, 40, 55, 15, 25, 10]

Step 7: Insert 5

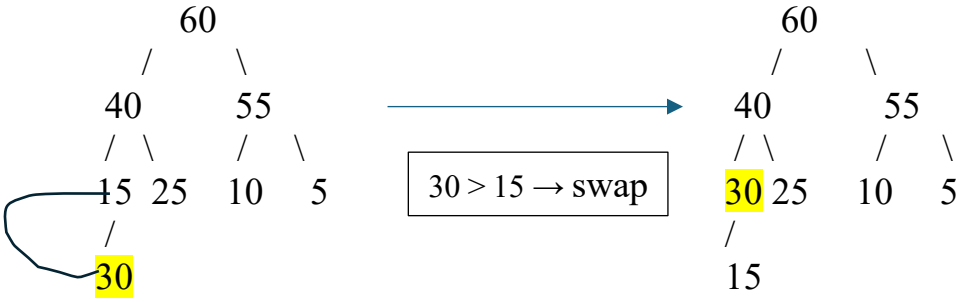
- Insert as right child of 55
- 5 < 55, no swap



Heap array: [60, 40, 55, 15, 25, 10, 5]

Step 8: Insert 30

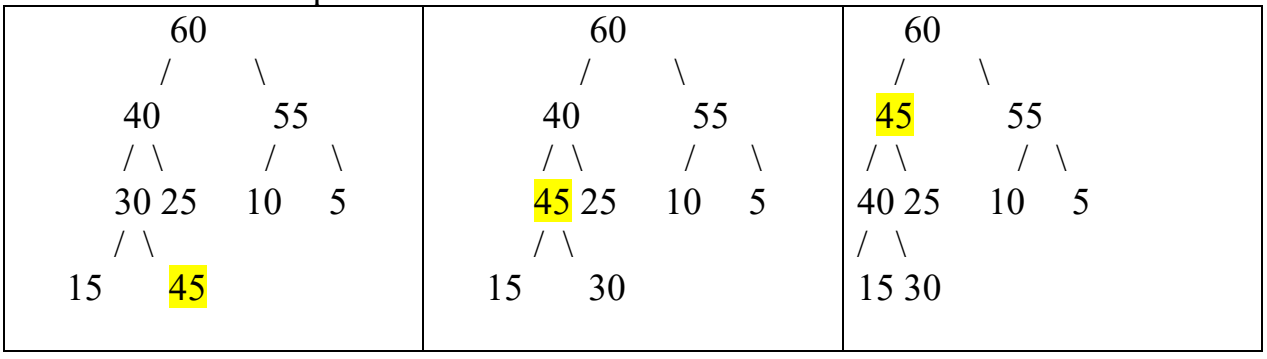
- Insert as left child of 15
- 30 > 15 → swap
- 30 < 40 → stop



Heap array: [60, 40, 55, 30, 25, 10, 5, 15]

Step 9: Insert 45

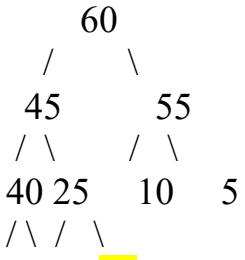
- Insert as right child of 30
- 45 > 30 → swap
- 45 > 40 → swap
- 45 < 60 → stop



Heap array: [60, 45, 55, 40, 25, 10, 5, 15, 30]

Step 10: Insert 20

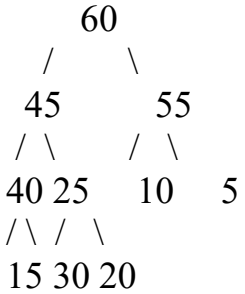
- Insert as left child of 25
- 20 < 25, no swap



Heap array:
[60, 45, 55, 40, 25, 10, 5, 15, 30, 20]

Final Max Heap

Tree Representation



Heap Array Representation

[60, 45, 55, 40, 25, 10, 5, 15, 30, 20]

Explanation:

Algorithm Control Flow

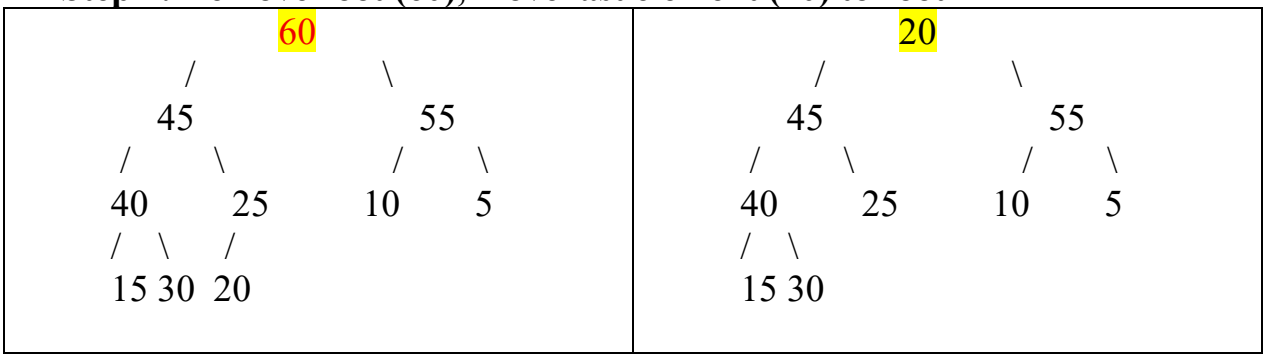
Step	Inserted Element	i	parent(i)	Comparison	Action	Tree (Before)	Tree (After)
1	15	0	—	—	Insert root	—	15
2	40	1	0	15 < 40	Swap	15 / 40	40 / 15
3	10	2	0	40 > 10	Stop	40 / \ 15 10	Same

4	60	3	1	15 < 60	Swap	40 / \ 15 10 / 60	40 / \ 60 10 / 15
5	60	1	0	40 < 60	Swap	40 / \ 60 10	60 / \ 40 10
6	25	4	1	40 > 25	Stop	60 / \ 40 10 / \ 15 25	Same
7	55	5	2	10 < 55	Swap	60 / \ 40 10 / \ / 15 25 55	60 / \ 40 55 / \ / 15 25 10
8	5	6	2	55 > 5	Stop	60 / \ 40 55 / \ / \ 15 25 10 5	Same
9	30	7	3	15 < 30	Swap	60 / \ 40 55 / \ / \ 15 25 10 5 / 30	60 / \ 40 55 / \ / \ 30 25 10 5 / 15
10	45	8	3	30 < 45	Swap	60 / \ 40 55 / \ / \ 30 25 10 5 / \ 15 45	60 / \ 40 55 / \ / \ 45 25 10 5 / \ 15 30
11	45	3	1	40 < 45	Swap	60 / \ 40 55	60 / \ 45 55
12	20	9	4	25 > 20	Stop	60 / \ 45 55 / \ / \ 40 25 10 5 / \ / 15 30 20	Same

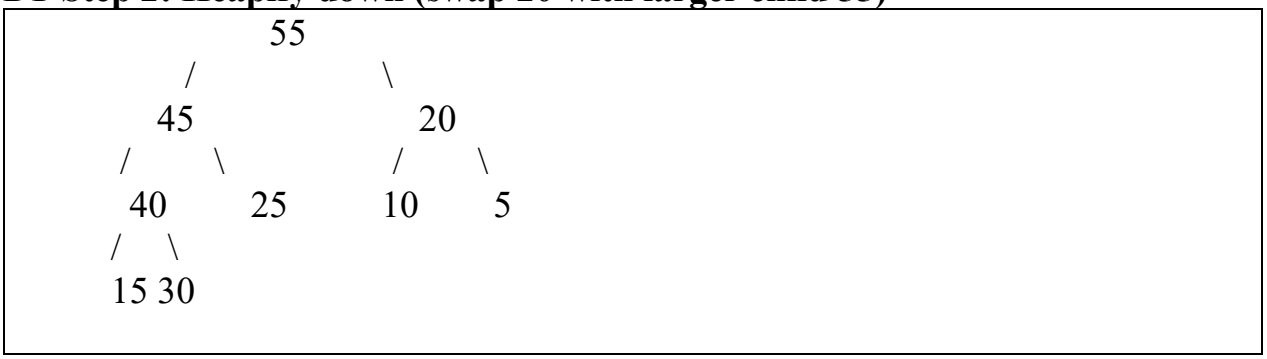
B) Max Heap – Step-by-step DELETION (2 deletions)

Deletion 1: Delete 60

D1-Step 1: Remove root (60), move last element (20) to root



D1-Step 2: Heapify down (swap 20 with larger child 55)

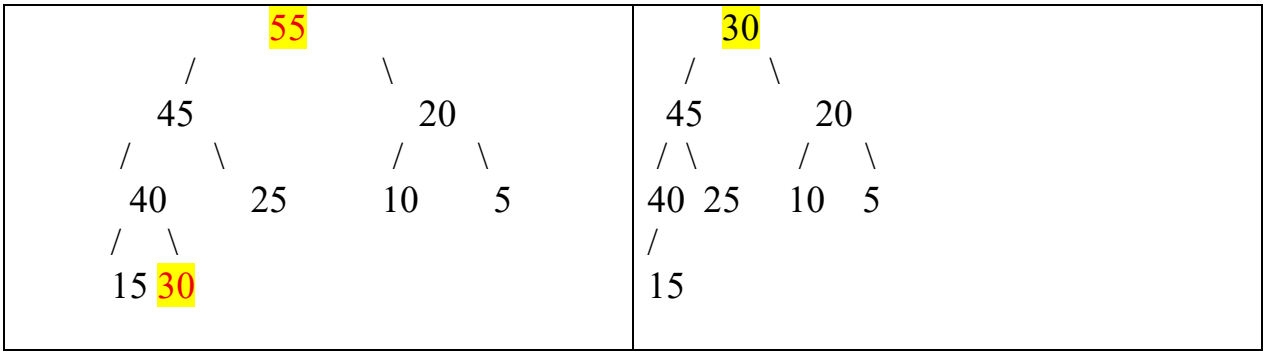


Heap after deleting 60

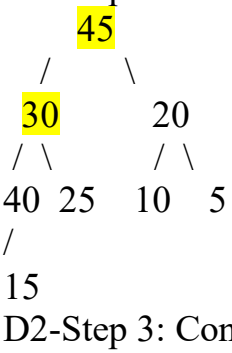
Heap array: 55,45,20,40,25,10,5,15,30

Deletion 2: Delete 55

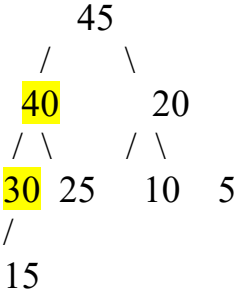
D2-Step 1: Remove root (55), move last element (30) to root



D2-Step 2: Heapify down (swap 30 with larger child 45)



D2-Step 3: Continue heapify (swap 30 with larger child 40)



Heap after deleting 55

Heap array: [45, 40, 20, 30, 25, 10, 5, 15]

Python code:

```
import heapq
class MaxHeap:
    def __init__(self):
        self.heap = []

    def insert(self, value):
        heapq.heappush(self.heap, -value)

    def delete(self):
        if not self.heap:
            print("Heap is empty")
            return None
        return -heapq.heappop(self.heap)

    def display(self):
        print("Max Heap:", [-x for x in self.heap])


# Driver code
h = MaxHeap()
elements = [15, 40, 10, 60, 25, 55, 5, 30, 45, 20]
```

```
for e in elements:
    h.insert(e)

print("Deleted:", h.delete())
print("Deleted:", h.delete())
h.display()
```

Min Heap

Algorithm: Min Heap (Priority Queue)
Insert Operation (Min Heap)

1. Insert the new element at the end of the heap.
2. Set i to the index of the inserted element.
3. While i > 0 and heap[parent(i)] > heap[i]:
    - Swap heap[i] and heap[parent(i)]
    - Update i = parent(i)
4. Stop when heap property is satisfied.

Logic:
Move the element upward until parent is smaller.

Delete Operation (Min Heap)

1. If heap is empty, print “Queue is empty”.
2. Store the root element (minimum).
3. Replace root with the last element.
4. Remove the last element.
5. Heapify down from root:
    - Compare with left and right children
    - Swap with the smallest child
6. Repeat until heap property is restored.
7. Return the deleted element.

Logic:
Move the element downward to restore min heap order.

Example:

Elements (Insertion Order)

[15, 40, 10, 60, 25, 55, 5, 30, 45, 20]

Step 1: Insert 15

15

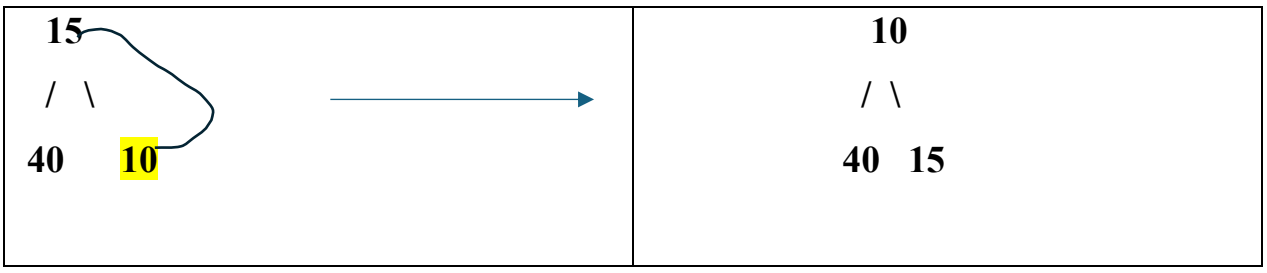
Step 2: Insert 40

15

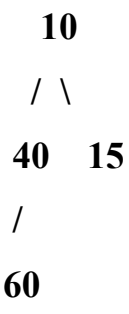
/

40

Step 3: Insert 10 (10 bubbles up)

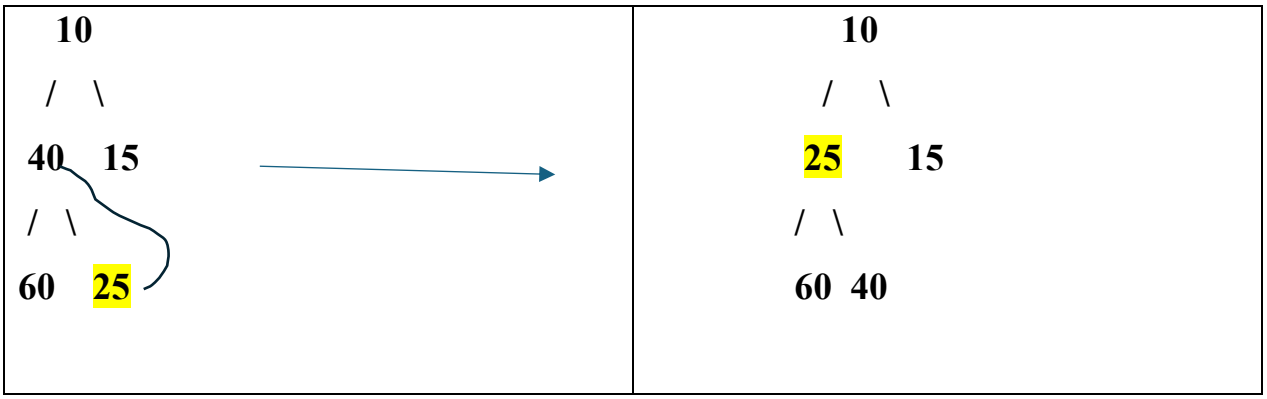


Step 4: Insert 60

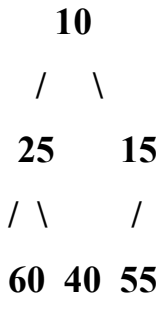


Step 5: Insert 25

(25 bubbles above 40)

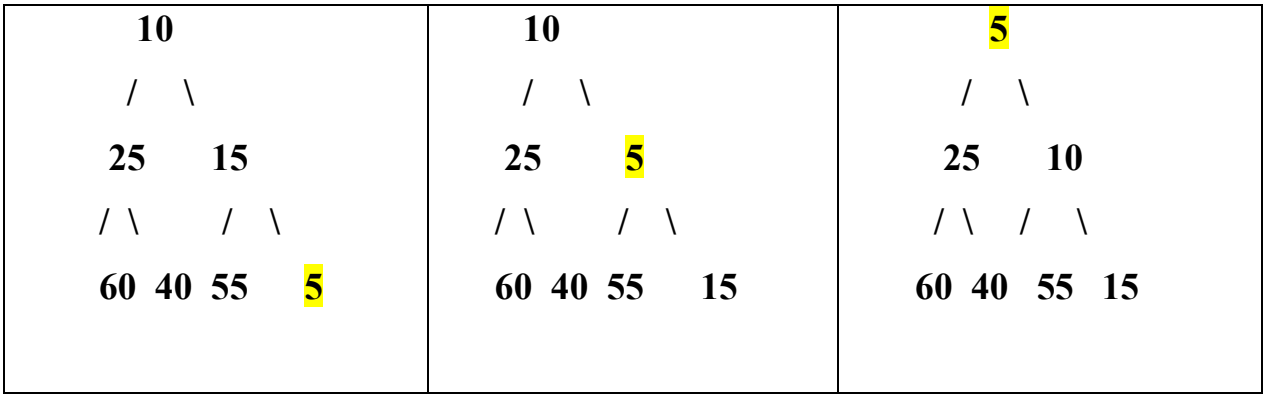


Step 6: Insert 55



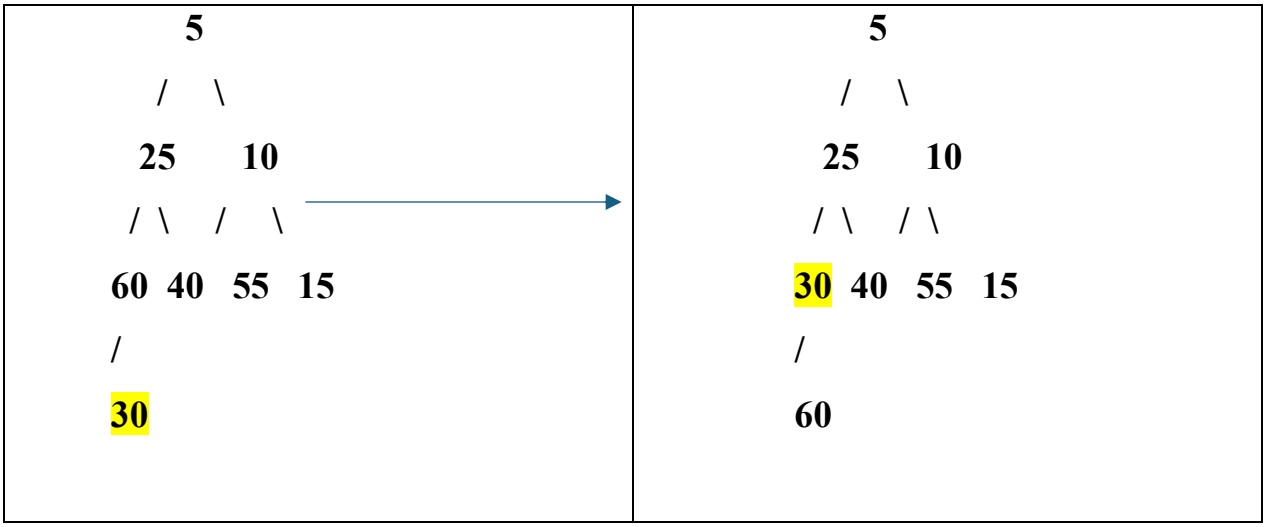
Step 7: Insert 5

(5 bubbles up to root)

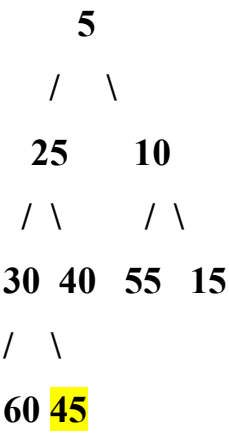


Step 8: Insert 30

(30 bubbles above 60)

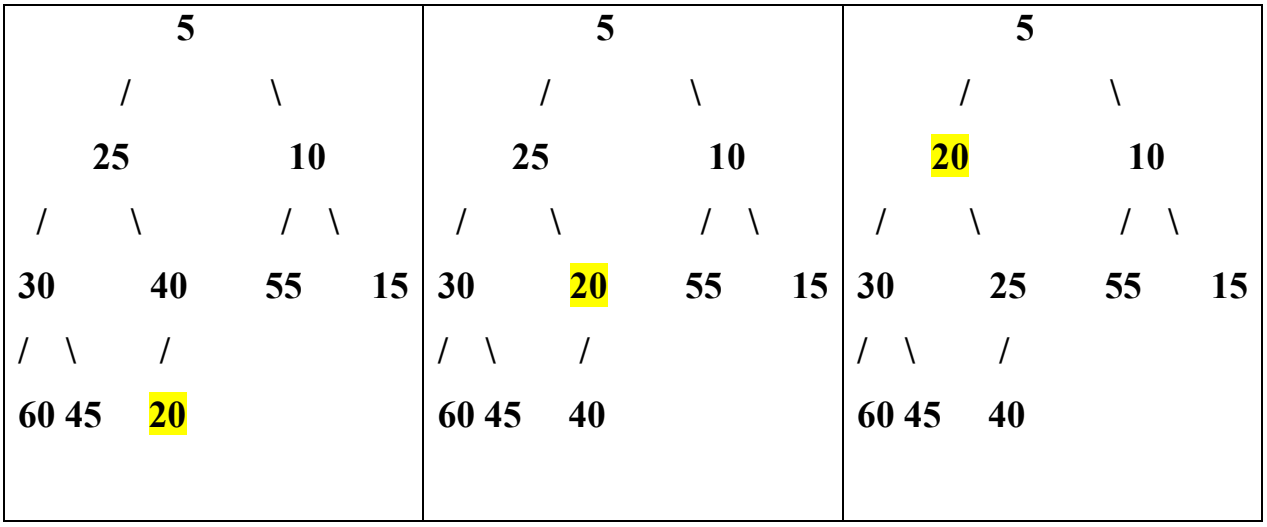


Step 9: Insert 45

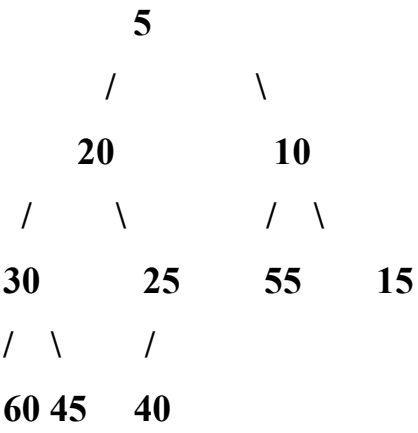


Step 10: Insert 20

(20 bubbles above 25)



Final Min Heap after all insertions



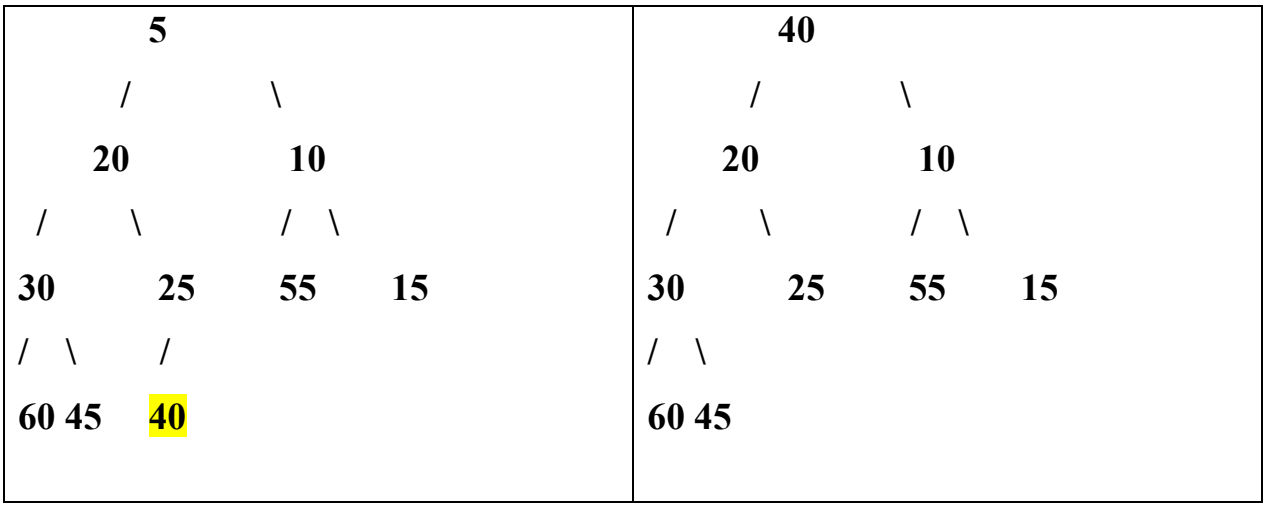
Heap array:

[5, 20, 10, 30, 25, 55, 15, 60, 45, 40]

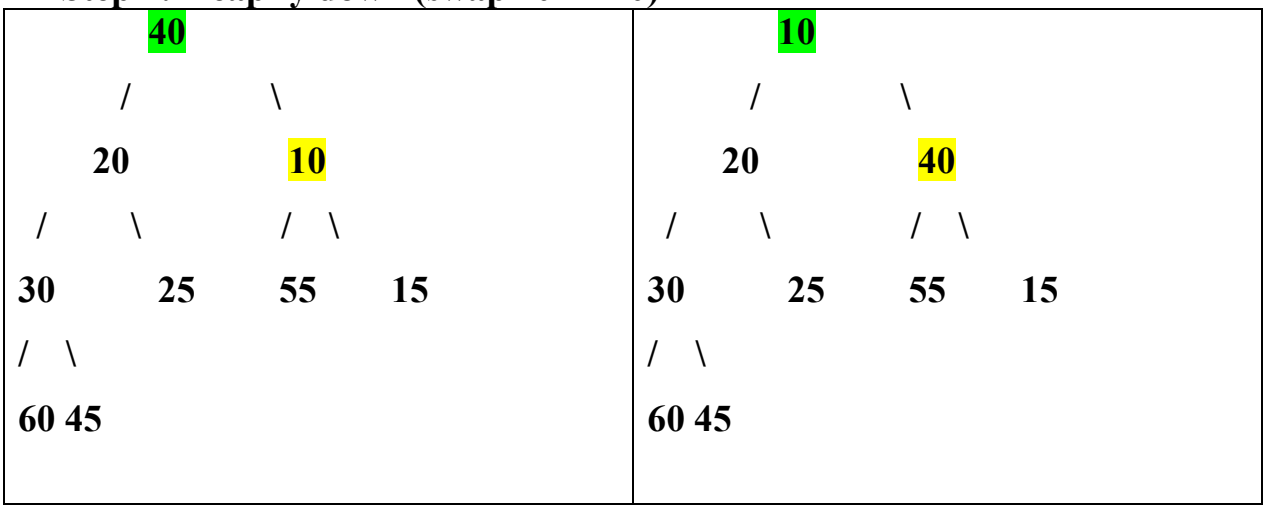
DELETION

Deletion 1: Delete 5 (Root)

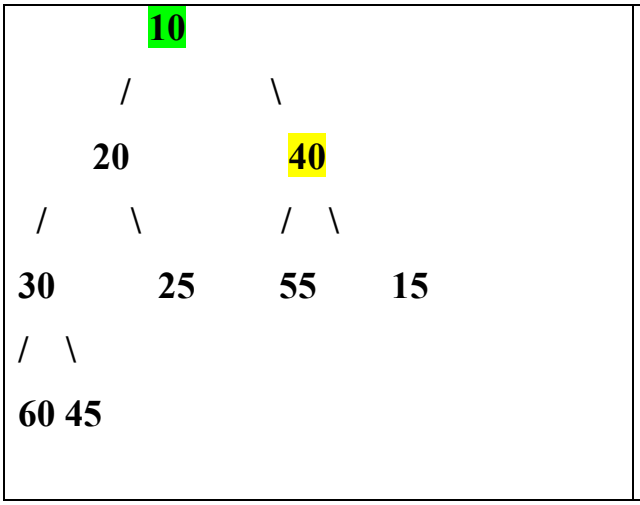
D1-Step 1: Replace root with last element (40)

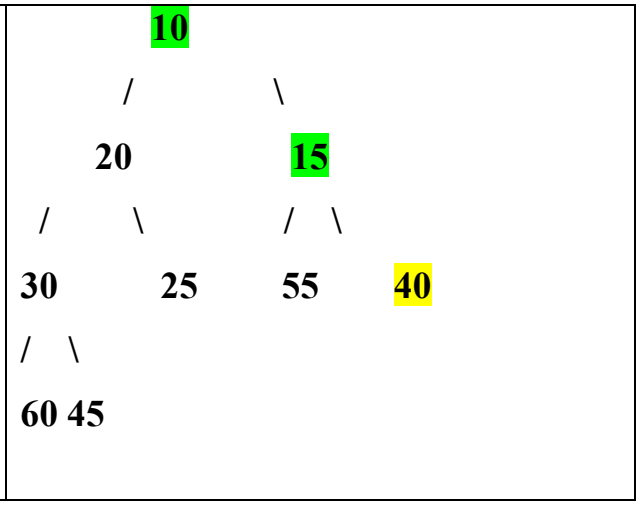


D1-Step 2: Heapify down (swap 40 ↔ 10)



D1-Step 3: Heapify down (swap 40 ↔ 15)

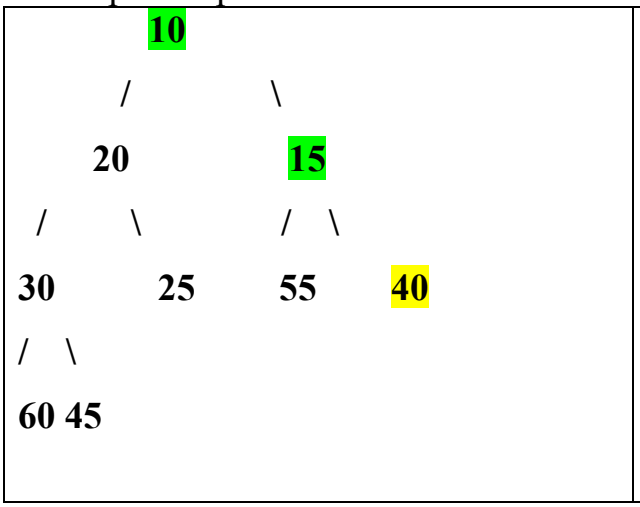


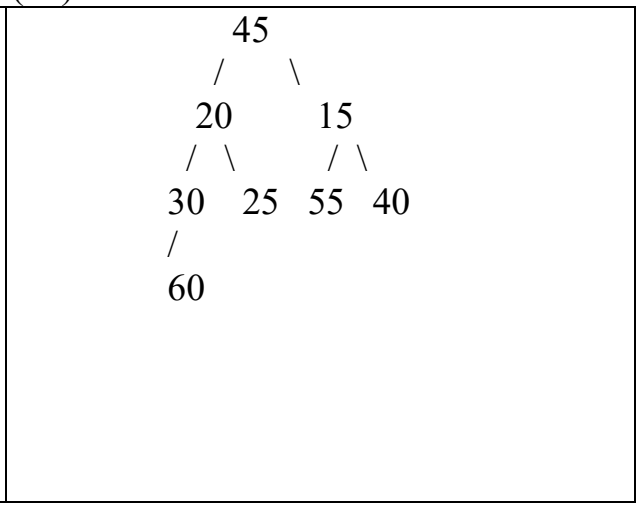


Heap after deleting 5
[10, 20, 15, 30, 25, 55, 40, 60, 45]

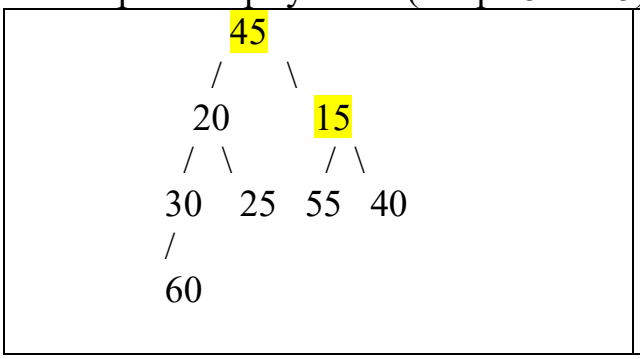
Deletion 2: Delete 10

D2-Step 1: Replace root with last element (45)

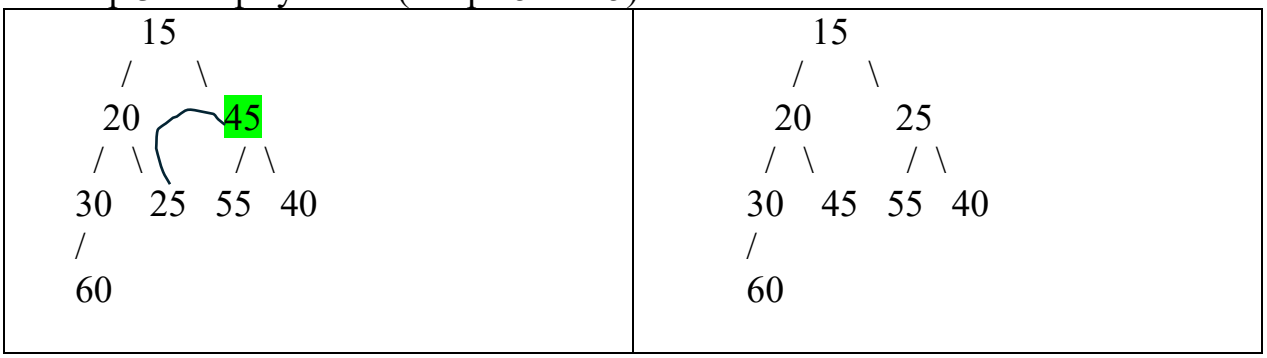




D2-Step 2: Heapify down (swap 45 ↔ 15)



D2-Step 3: Heapify down (swap 45 ↔ 25)



Final Min Heap
[15, 20, 25, 30, 45, 55, 40, 60]

Example :2 Array Represention

Elements:

[15, 40, 10, 60, 25, 55, 5, 30, 45, 20]

Step	Inserted	Heap Array
1	15	[15]
2	40	[15, 40]
3	10	[10, 40, 15]
4	60	[10, 40, 15, 60]
5	25	[10, 25, 15, 60, 40]
6	55	[10, 25, 15, 60, 40, 55]
7	5	[5, 25, 10, 60, 40, 55, 15]
8	30	[5, 25, 10, 30, 40, 55, 15, 60]
9	45	[5, 25, 10, 30, 40, 55, 15, 60, 45]
10	20	[5, 20, 10, 30, 25, 55, 15, 60, 45, 40]

Deletion:

Initial Min Heap Array

[5, 20, 10, 30, 25, 55, 15, 60, 45, 40]

Delete Highest Priority (Min Element)

Step	Operation	Heap After Operation
1	Delete root (5)	[40, 20, 10, 30, 25, 55, 15, 60, 45]
2	Heapify down – swap 40 ↔ 10	[10, 20, 40, 30, 25, 55, 15, 60, 45]
3	Heapify down – swap 40 ↔ 15	[10, 20, 15, 30, 25, 55, 40, 60, 45]

After 1st Deletion

[10, 20, 15, 30, 25, 55, 40, 60, 45]

Second Deletion

Step	Operation	Heap After Operation
1	Delete root (10)	[45, 20, 15, 30, 25, 55, 40, 60]
2	Heapify down – swap 45 ↔ 15	[15, 20, 45, 30, 25, 55, 40, 60]
3	Heapify down – swap 45 ↔ 25	[15, 20, 25, 30, 45, 55, 40, 60]

Final Min Heap

[15, 20, 25, 30, 45, 55, 40, 60]

Python code:

```
import heapq

class MinHeap:
    def __init__(self):
        self.heap = []

    def insert(self, value):
        heapq.heappush(self.heap, value)

    def delete(self):
```

```
        if not self.heap:
            print("Heap is empty")
            return None
        return heapq.heappop(self.heap)

    def display(self):
        print("Min Heap:", self.heap)


# Driver code
h = MinHeap()
elements = [15, 40, 10, 60, 25, 55, 5, 30, 45, 20]

for e in elements:
    h.insert(e)

print("Deleted:", h.delete())
print("Deleted:", h.delete())
h.display()
```

Differences:

Feature	Max Heap	Min Heap
Root node	Maximum element	Minimum element
Priority	Largest value	Smallest value
Parent relation	Parent ≥ Children	Parent ≤ Children
Common usage	Scheduling, Max-PQ	Dijkstra, Prim

2. Double Ended Priority Queues

A **Double Ended Priority Queue (DEPQ)** is an advanced type of priority queue that allows **efficient access, insertion, and deletion of both the minimum and maximum elements**. Unlike a standard priority queue, which supports deletion of either the minimum or the maximum, a DEPQ maintains **both extremes simultaneously**, making it ideal for applications that require continuous monitoring of both the largest and smallest elements. DEPQs are commonly implemented using **Min–Max Heaps**, where the root contains the minimum element, the largest elements are located at max levels, and levels alternate between min and max to maintain the heap property.

DEPQs support operations like insert(), deleteMin(), deleteMax(), getMin(), and getMax() efficiently. The **Min–Max Heap** structure ensures that these operations can be performed in **O(log n)** time for insertion and deletion, and **O(1)** time for accessing the minimum and maximum. This structure is widely used in **simulations, scheduling, and real-time systems**, where both extremes of the dataset need to be processed quickly.

Algorithm DEPQ_Insert_Delete:

Input: Element x to insert, or deleteMin / deleteMax operation

1. If inserting:

a. Insert x at next available leaf (complete binary tree)
b. Determine node level (even = MIN, odd = MAX)
c. If MIN level:
  i. If x > parent, swap and bubble-up in MAX levels
  ii. Else, bubble-up in MIN levels through grandparents
d. If MAX level:
  i. If x < parent, swap and bubble-up in MIN levels
  ii. Else, bubble-up in MAX levels through grandparents

2. If deleteMin:

a. Remove root
b. Replace root with last leaf
c. Heapify-down on MIN levels (swap with smallest child/grandchild)

3. If deleteMax:

a. Find maximum among root children
b. Remove that node
c. Replace with last leaf
d. Heapify-down on MAX levels (swap with largest child/grandchild)

4. Stop when Min–Max Heap property is restored

Example

Example with 10 Elements

Elements: [15, 40, 10, 60, 25, 55, 5, 30, 45, 20]

Step	Operation	Removed	Remaining Elements (Min-Heap view)	Min	Max
1	Insert all	-	[5, 10, 15, 20, 25, 30, 40, 45, 55, 60]	5	60
2	deleteMin()	5	[10, 15, 20, 25, 30, 40, 45, 55, 60]	10	60
3	deleteMax()	60	[10, 15, 20, 25, 30, 40, 45, 55]	10	55
4	deleteMin()	10	[15, 20, 25, 30, 40, 45, 55]	15	55
5	deleteMax()	55	[15, 20, 25, 30, 40, 45]	15	45

Example 1 : How to construct Min-Max Heap Tree of Given Elements (Insertion Order) [15, 40, 10, 60, 25, 55, 5, 30, 45, 20]

Solution:

Level Rules

- Level 0 → MIN
- Level 1 → MAX
- Level 2 → MIN
- Level 3 → MAX

STEP 1: Insert 15
(Min level → root)

15

STEP 2: Insert 40
(Level 1 → MAX level)

```
  15
 /
40
```

✓ Max level allows larger value

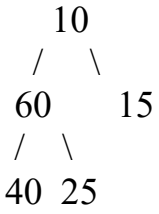
STEP 3: Insert 10
(Level 1 → MAX, but 10 < parent 15 → swap)

```
  10
 /  \
40   15
```

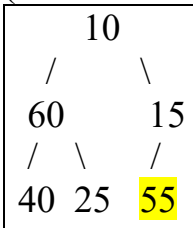
STEP 4: Insert 60
(Level 2 → MIN, but 60 > parent 40 → swap → move to MAX level)

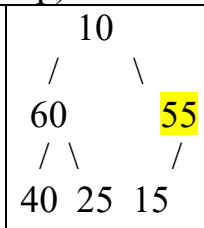
10 / \ 40 15 / 60	10 / \ 60 15 / 40

STEP 5: Insert 25
(Level 2 → MIN, valid)

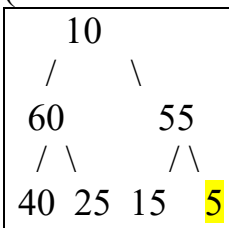


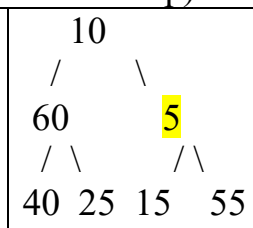
STEP 6: Insert 55
(Level 2 → MIN, but 55 > parent 15 → swap)

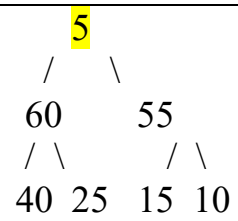




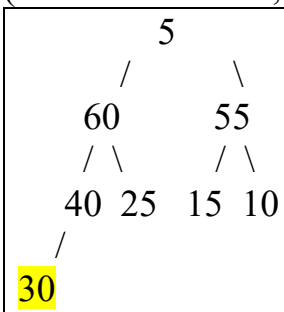
STEP 7: Insert 5
(Level 2 → MIN, 5 < root → bubble up)

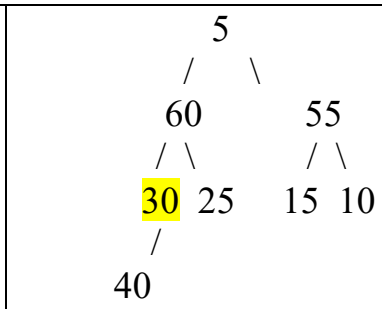




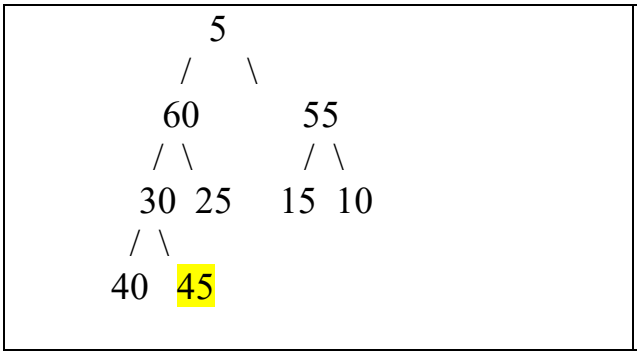


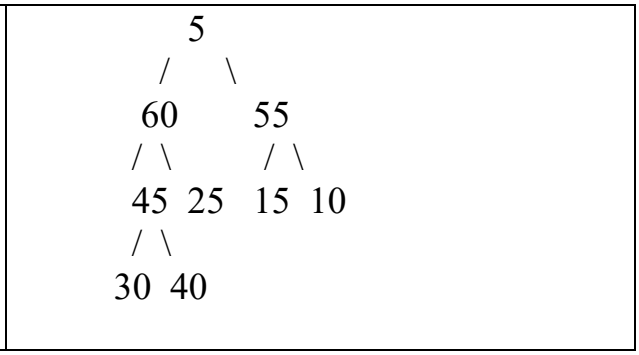
STEP 8: Insert 30
(Level 3 → MAX, valid)



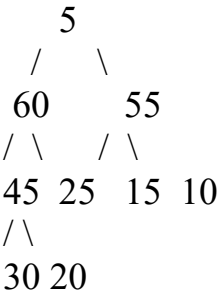


STEP 9: Insert 45
(Level 3 → MAX, 45 > parent 30 → swap)

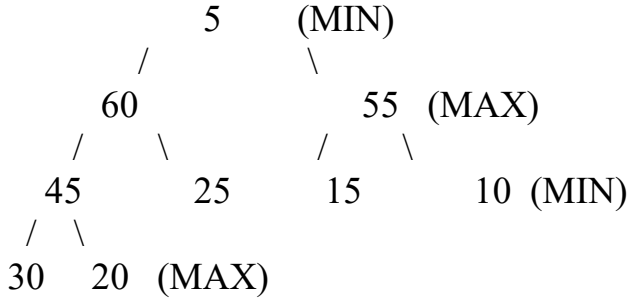




STEP 10: Insert 20
(Level 3 → MAX, valid)



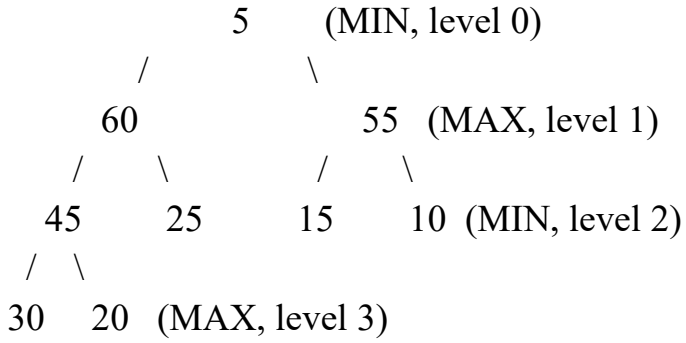
FINAL Min–Max Heap (DEPQ)



Example 2 :Double Ended Priority Queue

Problem: Given Elements (Insertion Order)
[15, 40, 10, 60, 25, 55, 5, 30, 45, 20]

Initial Min–Max Heap (DEPQ)



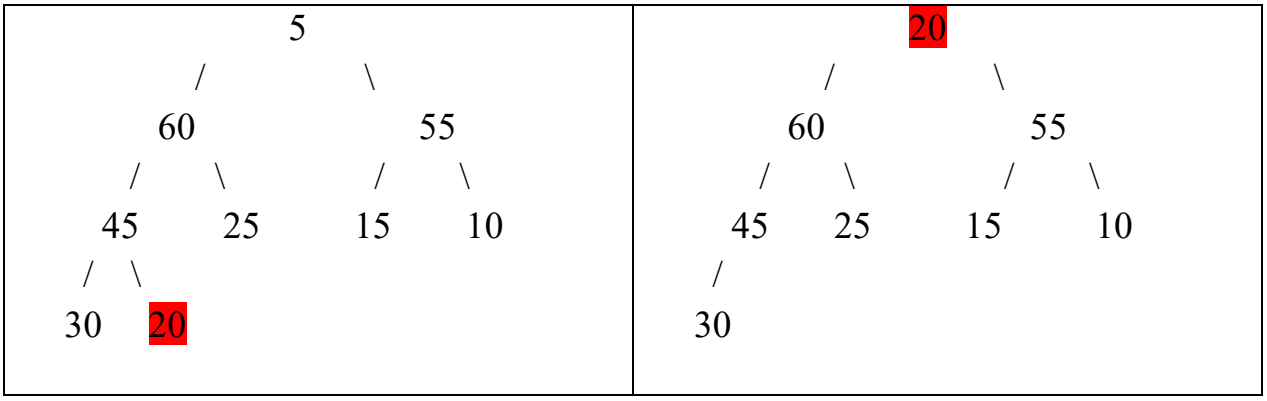
Solution:

deleteMin() operations

Operation 1: deleteMin()

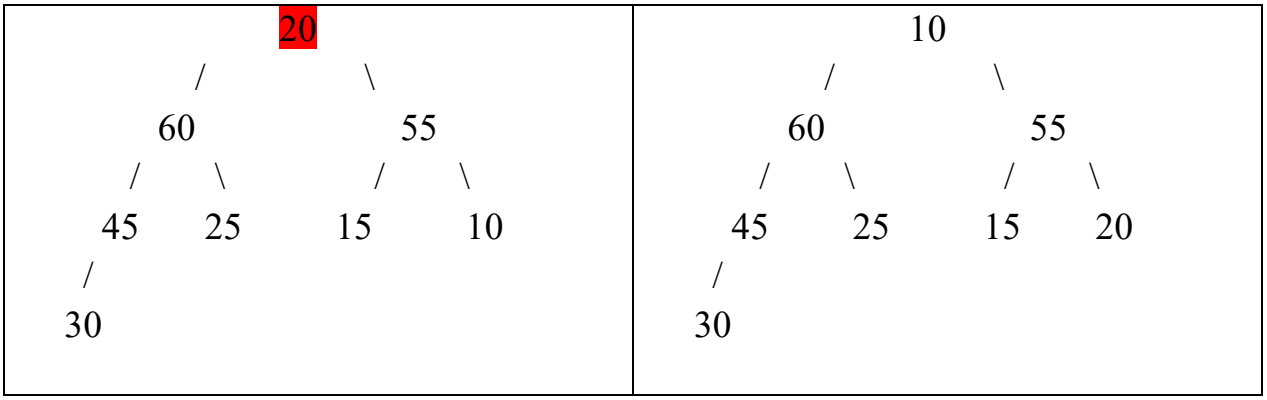
Step 1: Delete root = 5

Replace root with last leaf = 20



Step 2: Heapify-down at MIN level (root = level 0)

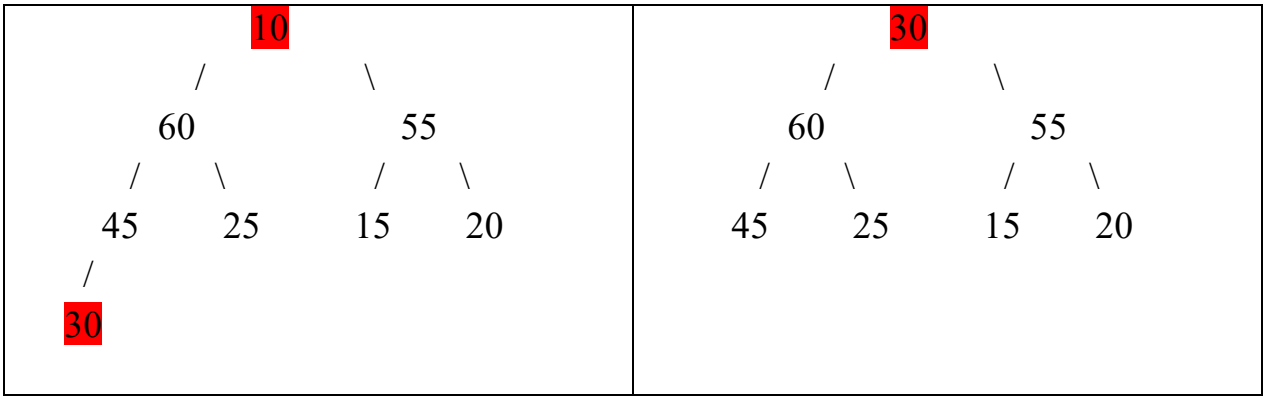
Compare with children & grandchildren: 60, 55, 45, 25, 15, 10, 30 → smallest = 10
Swap 20 ↔ 10



Step 3: 20 is at level 2 (MIN level). Its children = 0 → Stop
Deleted 5, new min = 10

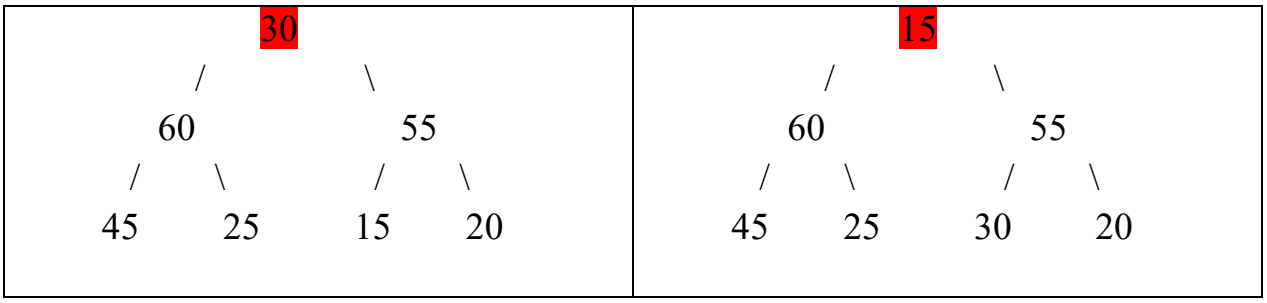
Operation 2: deleteMin()
Step 1: Delete root = 10

Replace root with last leaf = 30



Step 2: Heapify-down (root = MIN level)

Children & grandchildren: 60, 55, 45, 25, 15, 20 → smallest = 15
Swap 30 ↔ 15

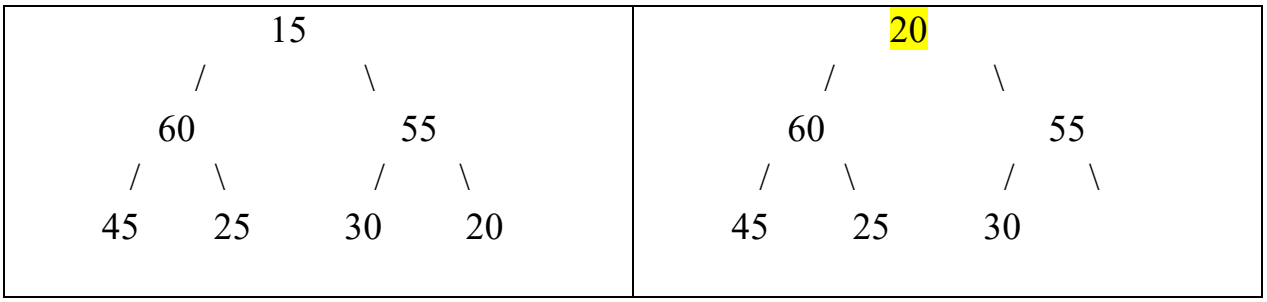


Step 3: 30 at level 2 (MIN), children = 0 → Stop
✅ Deleted 10, new min = 15

Operation 3: deleteMin()

Step 1: Delete root = 15

Replace root with last leaf = 20



Step 2: Heapify-down at MIN level

Children & grandchildren: 60, 55, 45, 25, 30 → smallest = 20 or 25?

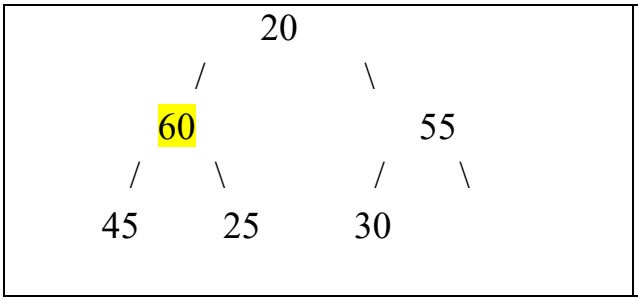
- Smallest among children/grandchildren = 20 already root? Actually 20 > 25? No, 25 < 20? 25 > 20, so 20 is smaller than all descendants → no swap needed.

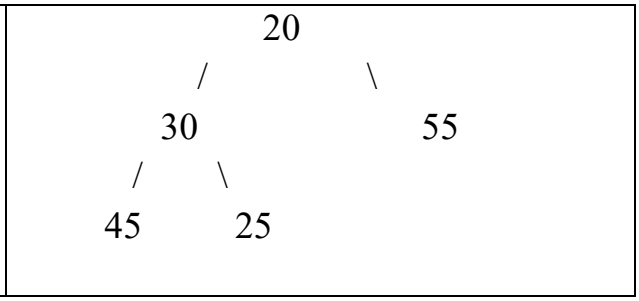
Deleted 15, new min = 20

deleteMax() operations

Operation 1: deleteMax()

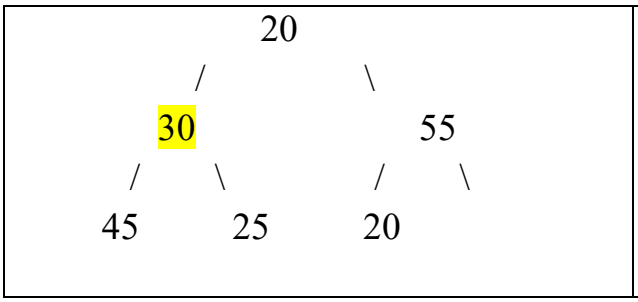
Step 1: Max element = largest among root's children = max(60, 55) = 60
Replace 60 with last leaf = 30

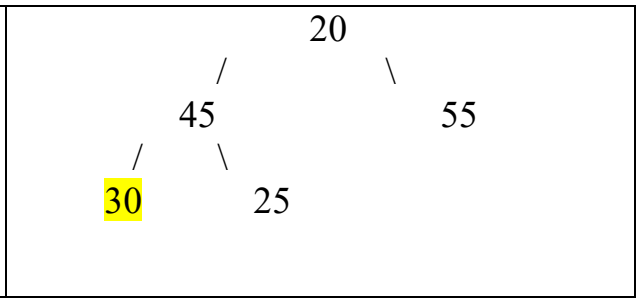




Step 2: Heapify-down at MAX level (node 30 at level 1)

Children & grandchildren: 45,25,20 → largest = 45
Swap 30 ↔ 45

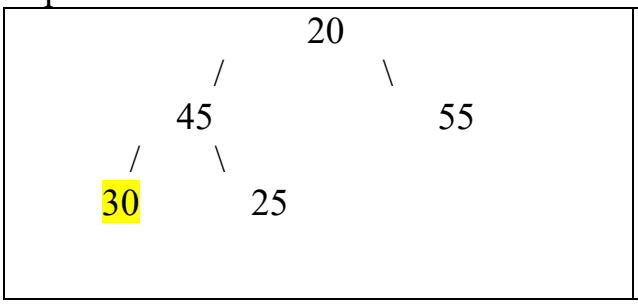


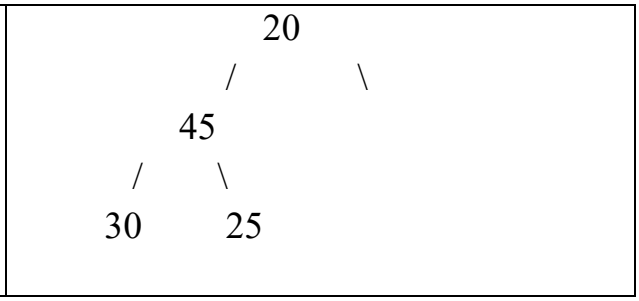


Deleted 60, new max = 55

Operation 2: deleteMax()

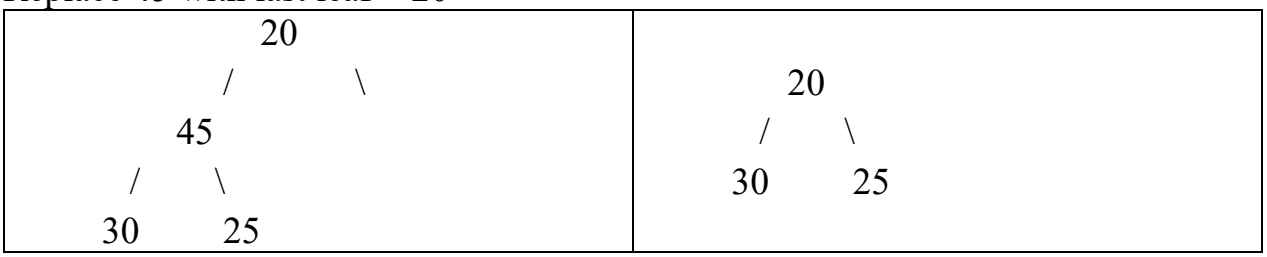
Step 1: Max element = largest among root's children = max(45, 55) = 55
Replace 55 with last leaf = 20



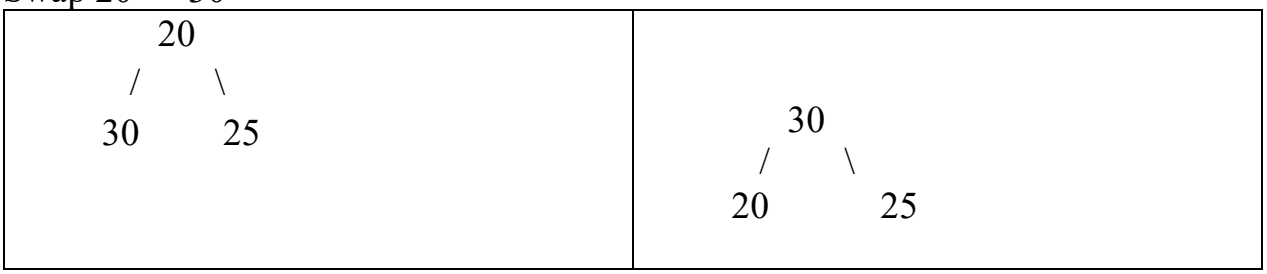


Step 2: Heapify-down at MAX level (node 20 at level 1)
Children & grandchildren: 20 → largest = 20, already in place → stop
Deleted 55, new max = 45

Operation 3: deleteMax()
Step 1: Max element = largest among root's children = max(45,20) = 45
Replace 45 with last leaf = 20



Step 2: Heapify-down at MAX level (node 20 at level 1)
Children & grandchildren: 30,25 → largest = 30
Swap 20 ↔ 30



Deleted 45, new max = 30
Summary Table for Exam

Operation	Deleted	New Min	New Max
deleteMin() 1	5	10	60
deleteMin() 2	10	15	60
deleteMin() 3	15	20	60
deleteMax() 1	60	20	55
deleteMax() 2	55	20	45
deleteMax() 3	45	20	30

Python code:

```
class DEPQ:
    def __init__(self):
        self.q = []   # list to store elements

    # Insert element
    def insert(self, x):
        self.q.append(x)

    # Delete minimum element
    def deleteMin(self):
        if len(self.q) == 0:
            print("DEPQ is empty")
            return -1

        min_index = 0
        for i in range(1, len(self.q)):
            if self.q[i] < self.q[min_index]:
                min_index = i

        min_val = self.q[min_index]
        self.q.pop(min_index)
        return min_val

    # Delete maximum element
    def deleteMax(self):
        if len(self.q) == 0:
            print("DEPQ is empty")
            return -1

        max_index = 0
        for i in range(1, len(self.q)):
            if self.q[i] > self.q[max_index]:
                max_index = i

        max_val = self.q[max_index]
        self.q.pop(max_index)
        return max_val

    # Display elements
    def display(self):
```

```
        print(self.q)


# ---------------- DRIVER CODE ----------------

depq = DEPQ()

# Read size
n = int(input("Enter number of elements: "))

# Read elements
print("Enter elements:")
for i in range(n):
    x = int(input())
    depq.insert(x)

print("\nDEPQ after insertion:")
depq.display()

# Perform operations
print("\nDeleted Minimum:", depq.deleteMin())
depq.display()

print("\nDeleted Maximum:", depq.deleteMax())
depq.display()

print("\nDeleted Minimum:", depq.deleteMin())
depq.display()

print("\nDeleted Maximum:", depq.deleteMax())
depq.display()
```

Sample Input

```
Enter number of elements: 10
Enter elements:
15
40
10
60
25
55
```

5
30
45
20

Sample Output

DEPQ after insertion:
[15, 40, 10, 60, 25, 55, 5, 30, 45, 20]
Deleted Minimum: 5
[15, 40, 10, 60, 25, 55, 30, 45, 20]
Deleted Maximum: 60
[15, 40, 10, 25, 55, 30, 45, 20]
Deleted Minimum: 10
[15, 40, 25, 55, 30, 45, 20]
Deleted Maximum: 55
[15, 40, 25, 30, 45, 20]

Practice Problems:

1. Define Priority Queue and Min Heap Construct a Min Heap for the following Elements 40 12 3 9 50 26 16 5 14 30

Solution:

A **Priority Queue** is an abstract data type where each element has a priority. In a **Min Priority Queue**, the element with the **smallest value (highest priority)** is removed first.

Min Heap

A **Min Heap** is a complete binary tree in which:

- The **minimum element is at the root**
- Each parent node is **less than or equal to its children**

Given Elements (Insertion Order)

40, 12, 3, 9, 50, 26, 16, 5, 14, 30

We insert elements **one by one** and apply **heapify-up** after each insertion.

Step-by-Step Min Heap Construction

Step 1: Insert 40

40

Step 2: Insert 12

(12 < 40 → swap)

```
  12
 /
40
```

Step 3: Insert 3

(3 < 12 → swap)

```
    3
  /   \
 40    12
```

Step 4: Insert 9

(9 < 40 → swap)

```
      3
    /   \
   9     12
  /
 40
```

Step 5: Insert 50

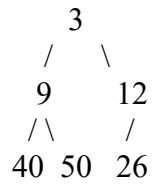
(No swap required)

```
      3
    /   \
   9     12
  / \
```

40 50

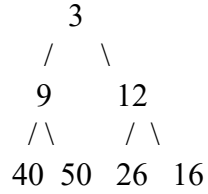
Step 6: Insert 26

(No swap required)



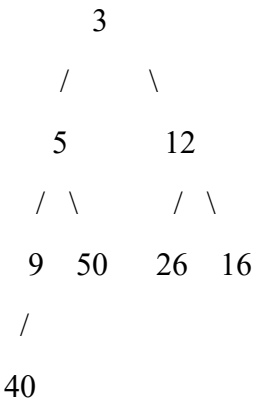
Step 7: Insert 16

(No swap required)



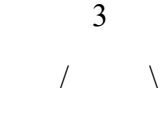
Step 8: Insert 5

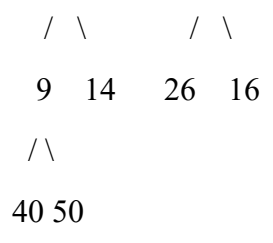
(5 < 40 → swap, 5 < 9 → swap)



Step 9: Insert 14

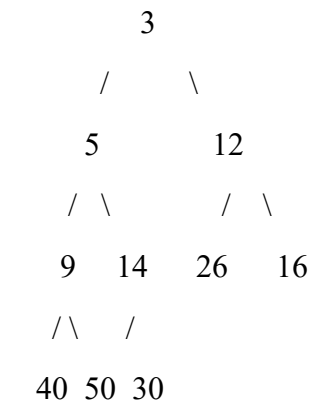
(14 < 50 → swap)





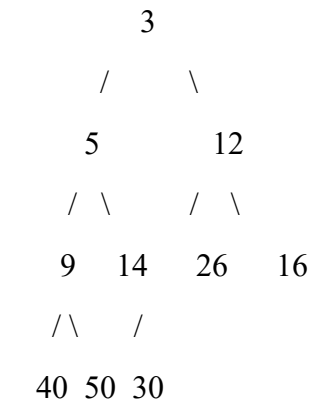
Step 10: Insert 30

(30 > 14 → no swap)



Final Min Heap

Tree Representation



Level Order Form

3, 5, 12, 9, 14, 26, 16, 40, 50, 30

2. Given a min-heap containing elements {8, 12, 15, 20}, explain how the heap property is maintained after inserting element 5.

Solutoin:

Given Min-Heap

Elements:

{8, 12, 15, 20}

Initial Min-Heap (Tree Representation)

```
      8
    /   \
   12    15
  /
 20
```

Heap property:
✓ Complete binary tree
✓ Parent ≤ children

Insert Element: 5

Insertion in a Min Heap is done in **two steps**:

1. **Insert at the next available position** (to maintain completeness)
2. **Heapify-up (percolate-up)** to restore heap property

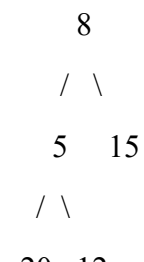
Step-by-Step Insertion

Step 1: Insert 5 at the next available position

```
      8
    /   \
   12    15
  /  \
 20   5
```

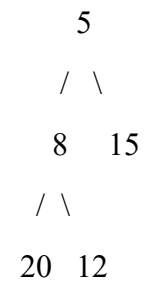
Heap property violated:
✗ 5 < 12

Step 2: Swap 5 with its parent (12)



Heap property still violated:
✗ 5 < 8

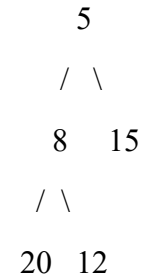
Step 3: Swap 5 with its parent (8)



Heap property satisfied:
✓ Parent ≤ children

Final Min-Heap

Tree Representation



Level Order Representation

5, 8, 15, 20, 12

- Element **5** is inserted at the lowest level
- **Heapify-up** swaps it with parents until the Min Heap property is restored
- The **minimum element (5)** becomes the new root

3. In a priority queue implemented using a heap, explain how the priority of elements affects the deletion operation, considering a scenario where elements **{10, 25, 5, 40}** are inserted.

Solution:

Priority Queue Using Heap

In a **priority queue**, each element's **priority determines the order of deletion**. In a **Min Heap–based priority queue**, the **element with the smallest value (highest priority)** is **deleted first** using the **delete-Min operation**.

Given Elements (Insertion Order)

{10, 25, 5, 40}

Step-by-Step Heap Construction

Step 1: Insert 10

10

Step 2: Insert 25

(No swap needed)

```
  10
 /
25
```

Step 3: Insert 5

(5 < 10 → swap)

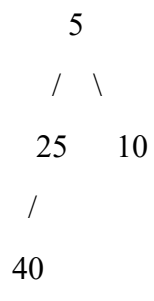
```
    5
  /   \
 25    10
```

Step 4: Insert 40

(No swap needed)

```
      5
    /   \
  25     10
 /
40
```

Min Heap After All Insertions

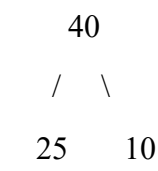


✓ **Minimum element (5)** has the **highest priority**

Deletion Operation (delete-Min)

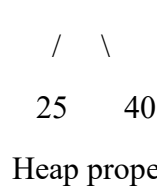
Step 5: Remove Root (Highest Priority Element = 5)

Replace root with the **last element (40)**:



Step 6: Heapify-Down (Restore Heap Property)

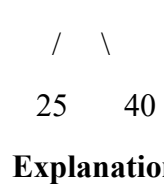
Compare 40 with children (25, 10)
Smallest child = 10 → swap



Heap property restored.

Final Heap After Deletion

Tree Representation



Explanation: How Priority Affects Deletion

- In a **Min-Priority Queue**, **smaller values = higher priority**
- The **root element** is always deleted first
- After deletion:
    - The **last element replaces the root**

- **Heapify-down** ensures the next highest-priority element moves to the root

Key Observations

- **5** was deleted first because it had **highest priority**
- The next highest priority element (**10**) becomes the new root
- Heap structure ensures **efficient deletion in O(log n)**

The **priority of elements directly determines deletion order** in a heap-based priority queue. The heap structure guarantees that the **highest-priority element is always removed first**, maintaining efficiency and correctness.

4. In a DEPQ containing {18, 35, 12, 50, 25, 60}, evaluate how the structure supports simultaneous access to highest and lowest priority elements during delete operations.

Solutions:

Double Ended Priority Queue (DEPQ)

A **DEPQ** allows:

- **deleteMin()** → removes the **lowest priority (minimum)** element
- **deleteMax()** → removes the **highest priority (maximum)** element

Both operations are supported **efficiently**.

A **Min–Max Heap**:

- Is a **complete binary tree**
- **Min levels (even levels)** → contain smaller elements
- **Max levels (odd levels)** → contain larger elements
- **Minimum element** is always at the **root**
- **Maximum element** is always at **one of the root's children**

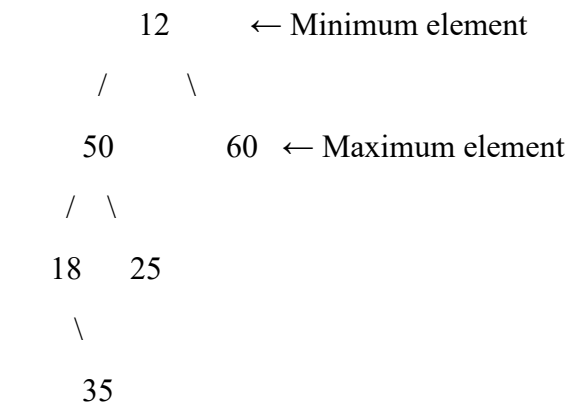
Given Elements

{18, 35, 12, 50, 25, 60}

Step 1: Construct DEPQ (Min–Max Heap)

After inserting all elements and arranging them according to Min–Max Heap rules, the DEPQ becomes:

Initial DEPQ (Tree Representation)

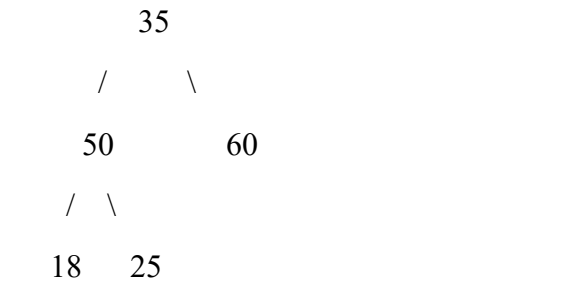


- **Min element** = 12 (root)
- **Max element** = max(50, 60) = 60 (root's children)

DeleteMin Operation

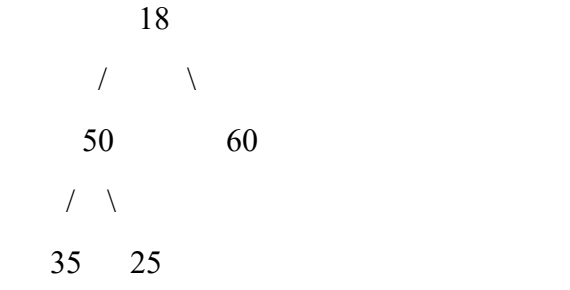
Step 2: deleteMin() → Remove 12

Replace root with the **last element (35)**:



Step 3: Restore Min–Max Heap Property

Compare 35 with its children (50, 60)
Smallest descendant = **18** → swap



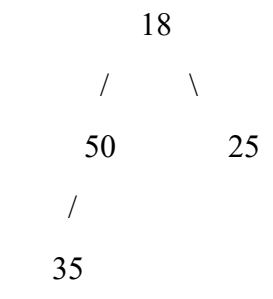
Heap property restored.

✓ New **minimum** = 18

DeleteMax Operation

Step 4: deleteMax() → Remove 60

Replace 60 with the **last element (25)**:



Step 5: Restore Max-Level Property

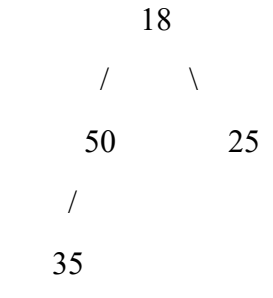
Compare 25 with its parent (50)
25 < 50 → no violation at max level

Heap property restored.

✓ New **maximum** = 50

Final DEPQ After deleteMin and deleteMax

Tree Representation



How DEPQ Supports Simultaneous Access

Operation	Element Accessed	Location in Tree
deleteMin	Lowest priority (18)	Root
deleteMax	Highest priority (50)	Root's children

Key Observations

- **Minimum element** is always found at the **root**
- **Maximum element** is always found at **one of the root's children**
- Both delete operations run in **O(log n)** time

- DEPQ efficiently supports **simultaneous min and max access**

A DEPQ implemented using a **Min–Max Heap** allows efficient deletion of both **highest and lowest priority elements** without rebuilding the structure, making it ideal for applications requiring **bidirectional priority access**.

5. A Double Ended Priority Queue is used in a real-time scheduling system to manage tasks with varying priorities. Given the insertion order of elements [15, 40, 10, 60, 25, 55, 5, 30, 45, 20], construct a suitable data structure to represent the queue and perform the deleteMin() operation followed by the deleteMax() operation, showing the intermediate and final states clearly.

Solution:

Double Ended Priority Queue (DEPQ)

A DEPQ supports:

- deleteMin() → removes the lowest priority (minimum)
- deleteMax() → removes the highest priority (maximum)

Min–Max Heap Properties

- Complete binary tree
- Even levels (0, 2, …) → Min levels
- Odd levels (1, 3, …) → Max levels
- Minimum element at the root
- Maximum element at one of the root's children

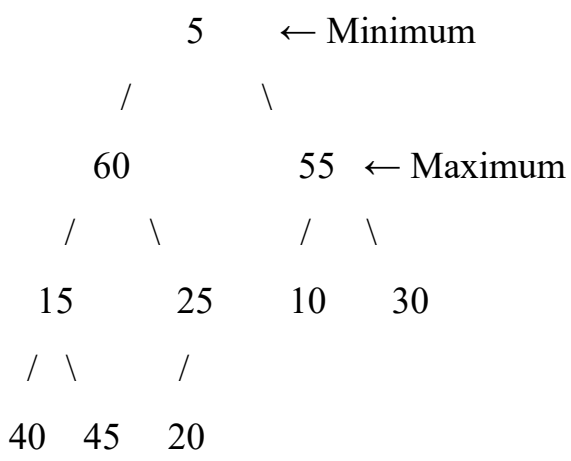
Given Insertion Order

[15, 40, 10, 60, 25, 55, 5, 30, 45, 20]

Step 1: Construct DEPQ (Min–Max Heap)

After inserting all elements and rearranging according to Min–Max Heap rules, the DEPQ becomes:

Initial DEPQ (Tree Representation)

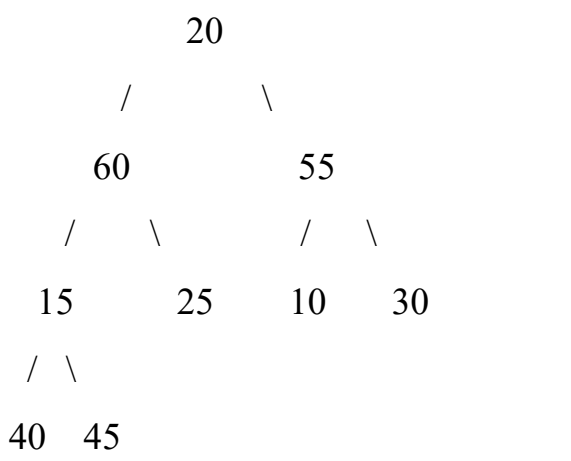


- Minimum element = 5 (root)
- Maximum element = max(60, 55) = 60

Operation 1: deleteMin()

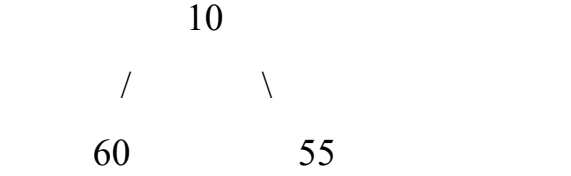
Step 2: Remove Minimum Element (5)

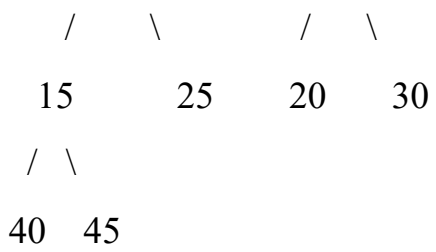
Replace root with the last element (20):



Step 3: Restore Min–Max Heap Property (Heapify Down on Min Level)

Smallest descendant of 20 = 10 → swap





✓ New minimum = 10

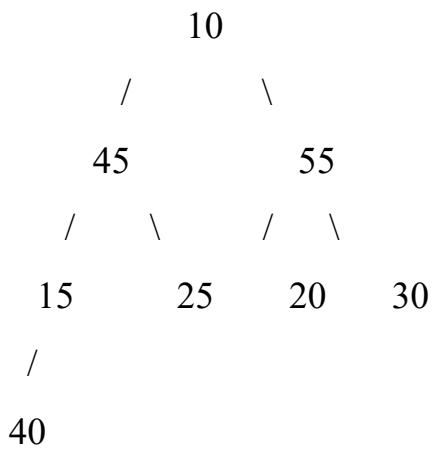
Operation 2: deleteMax()

Step 4: Identify Maximum Element

Maximum is at root's children → max(60, 55) = 60

Step 5: Remove Maximum Element (60)

Replace 60 with the last element (45):



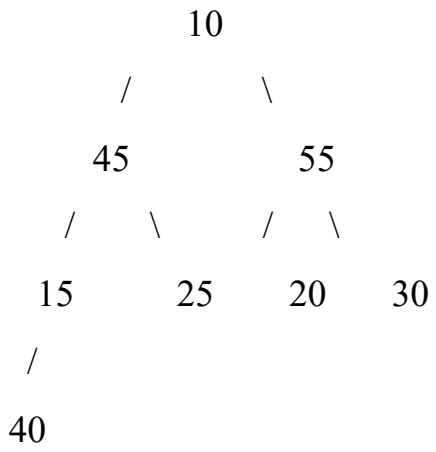
Step 6: Restore Max-Level Property (Heapify Down on Max Level)

Children of 45 → (15, 25)
45 ≥ children → heap property satisfied

✓ New maximum = 55

Final DEPQ After deleteMin() and deleteMax()

Tree Representation



Summary of Operations

Operation	Element Removed	Result
deleteMin()	5	Root updated and heapified
deleteMax()	60	Max child removed and heapified

Key Insight (Real-Time Scheduling Context)

- Lowest priority task is accessed directly at the root
- Highest priority task is accessed directly from the root's children
- Both deletions occur in O(log n) time
- DEPQ efficiently supports simultaneous min and max priority management

Using a Min–Max Heap, the DEPQ efficiently manages real-time scheduling by allowing fast access and deletion of both highest and lowest priority tasks, while maintaining a balanced tree structure.

6. Given the insertion order {15, 40, 10, 60, 25, 55, 5, 30, 45, 20}, construct a Double Ended Priority Queue and explain the effect of performing DeleteMin() and DeleteMax() operations, showing how both minimum and maximum elements are efficiently removed.

Solution:

Double Ended Priority Queue (DEPQ)

A DEPQ allows:

- **DeleteMin()** → removes the **lowest-priority (minimum) element**
- **DeleteMax()** → removes the **highest-priority (maximum) element**

Min–Max Heap Properties

- Complete binary tree
- **Min levels (even levels)** contain smaller values
- **Max levels (odd levels)** contain larger values
- **Minimum** at the **root**
- **Maximum** at **one of the root's children**

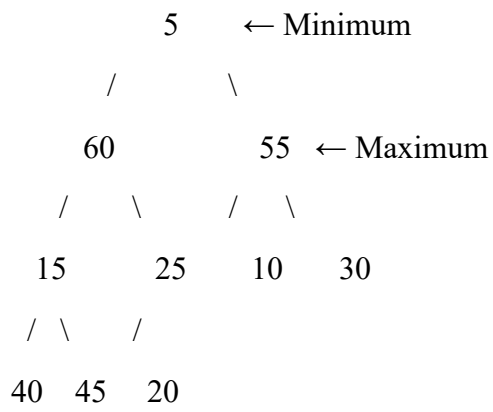
Given Insertion Order

{15, 40, 10, 60, 25, 55, 5, 30, 45, 20}

Step 1: Construct DEPQ (Min–Max Heap)

After inserting all elements and rearranging according to Min–Max Heap rules, the DEPQ becomes:

Initial DEPQ – Tree Representation

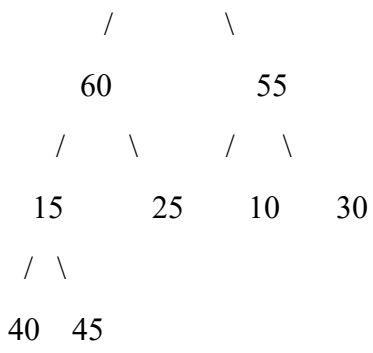


- **Minimum element** = 5 (root)
- **Maximum element** = max(60, 55) = 60

Operation 1: DeleteMin()

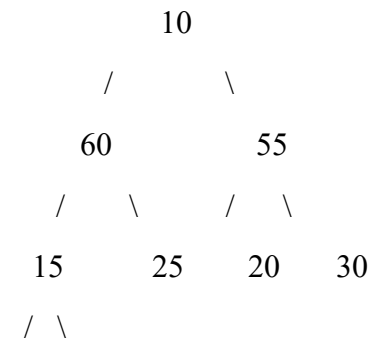
Step 2: Remove Minimum Element (5)

Replace the root with the **last element (20)**:



Step 3: Restore Min–Max Heap Property (Heapify Down on Min Level)

Smallest descendant of 20 is 10 → swap



✓ **New minimum = 10**

Operation 2: DeleteMax()

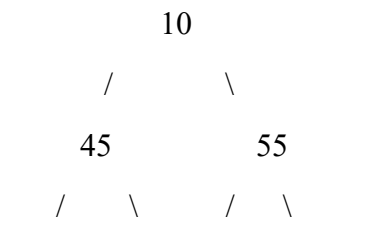
Step 4: Identify Maximum Element

Maximum element is at one of the root's children:

max(60, 55) = 60

Step 5: Remove Maximum Element (60)

Replace 60 with the **last element (45)**:



15 25 20 30

/

40

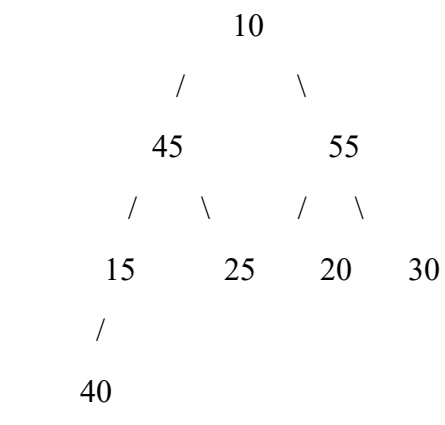
Step 6: Restore Max-Level Property

- Children of 45 → 15, 25
- Since 45 ≥ children, heap property is satisfied

✓ New maximum = 55

Final DEPQ After DeleteMin() and DeleteMax()

Tree Representation



Effect of DeleteMin() and DeleteMax()

Operation	Element Removed	Where Found	Result
DeleteMin()	5	Root	New minimum becomes 10
DeleteMax()	60	Root's child	New maximum becomes 55

Key Explanation

- **DeleteMin()** directly removes the root (minimum)
- **DeleteMax()** removes the larger of the root's two children
- Both operations take **O(log n)** time
- The structure ensures **simultaneous access** to minimum and maximum elements

A DEPQ implemented using a **Min–Max Heap** efficiently supports **real-time scheduling** by allowing fast removal of both **lowest-priority and highest-priority tasks**, while maintaining a balanced tree structure.

7. A Double Ended Priority Queue is used in a hospital emergency management system to schedule patients based on priority levels. Given the insertion order of elements [22, 48, 12, 65, 35, 58, 8, 41, 50, 18], construct a suitable data structure to represent the queue. Perform the deleteMin() operation followed by the deleteMax() operation, and clearly show all intermediate and final states of the queue.

Solution:

Double Ended Priority Queue (DEPQ)

A DEPQ supports:

- **deleteMin()** → removes the **lowest-priority (minimum value) patient**
- **deleteMax()** → removes the **highest-priority (maximum value) patient**

Min–Max Heap Characteristics

- Complete binary tree
- **Even levels (0, 2, …)** → Min levels
- **Odd levels (1, 3, …)** → Max levels
- **Minimum** at the **root**
- **Maximum** at **one of the root's children**

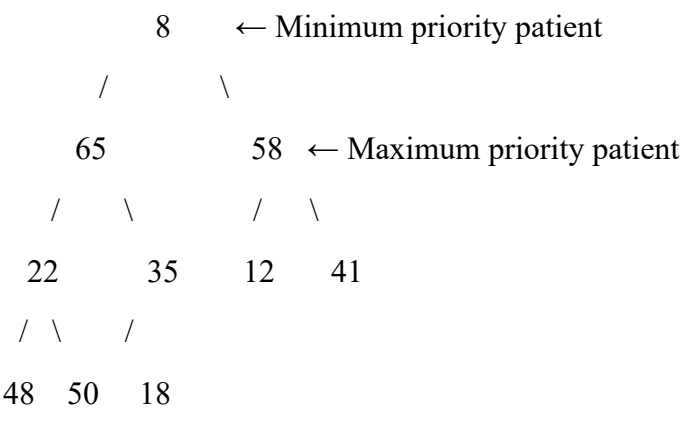
Given Insertion Order

[22, 48, 12, 65, 35, 58, 8, 41, 50, 18]

Step 1: Construct DEPQ (Min–Max Heap)

After inserting all elements and rearranging according to Min–Max Heap rules, the DEPQ becomes:

Initial DEPQ – Tree Representation



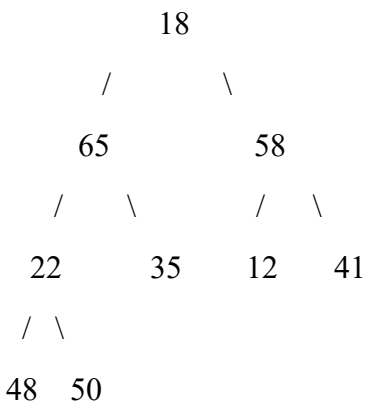
- **Minimum element** = 8 (root)

- **Maximum element** = max(65, 58) = 65

Operation 1: deleteMin()

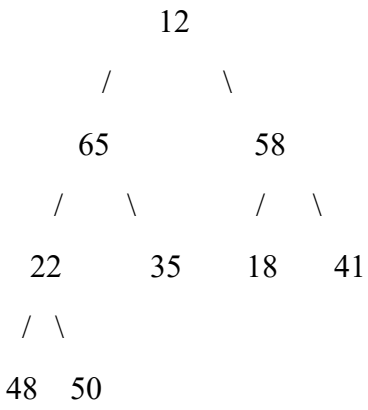
Step 2: Remove Minimum Element (8)

Replace the root with the **last element (18)**:



Step 3: Restore Min–Max Heap Property (Heapify Down on Min Level)

Smallest descendant of 18 is 12 → swap



✓ New minimum = 12

Operation 2: deleteMax()

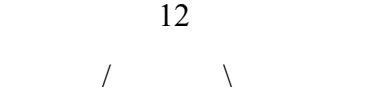
Step 4: Identify Maximum Element

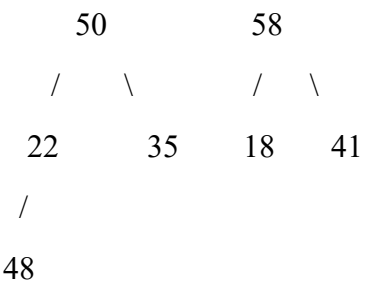
Maximum is at the root's children:

max(65, 58) = 65

Step 5: Remove Maximum Element (65)

Replace 65 with the **last element (50)**:





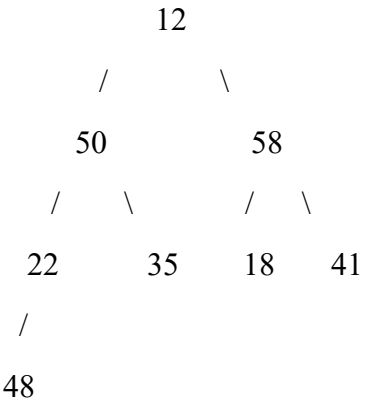
Step 6: Restore Max-Level Property

- Children of 50 → 22, 35
- Since 50 ≥ children, heap property is satisfied

✓ New maximum = 58

Final DEPQ After deleteMin() and deleteMax()

Tree Representation



Summary of Operations

Operation	Removed Element	Reason
deleteMin()	8	Lowest priority patient
deleteMax()	65	Highest priority patient

How This Helps in Hospital Emergency Management

- **Critical patients (highest priority)** are removed quickly using **deleteMax()**
- **Low-priority patients** can be accessed using **deleteMin()**
- Both operations take **O(log n)** time
- Ensures **fast, reliable scheduling** in emergency scenarios

A **Min–Max Heap–based DEPQ** efficiently supports simultaneous removal of **minimum and maximum priority patients**, making it ideal for **real-time hospital emergency scheduling systems**.

8. A Double Ended Priority Queue is applied in a network packet scheduling system where packets arrive with varying priority values. Given the elements inserted in the order [28, 64, 17, 45, 9, 53, 31, 70, 22, 40], construct a suitable representation of the queue. Perform the deleteMin() operation followed by the deleteMax() operation, and clearly depict the intermediate states and final structure of the queue.

Solution:

Double Ended Priority Queue (DEPQ)

A DEPQ supports:

- **deleteMin()** → removes the **lowest priority packet**
- **deleteMax()** → removes the **highest priority packet**

Min–Max Heap Properties

- Complete binary tree
- **Even levels (0, 2, …)** → Min levels
- **Odd levels (1, 3, …)** → Max levels
- **Minimum element** at the **root**
- **Maximum element** at **one of the root's children**

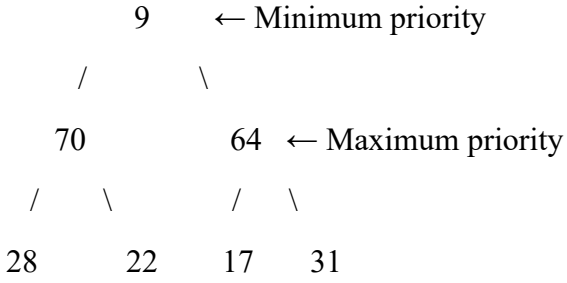
Given Insertion Order

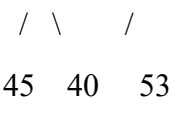
[28, 64, 17, 45, 9, 53, 31, 70, 22, 40]

Step 1: Construct DEPQ (Min–Max Heap)

After inserting all elements and reordering according to Min–Max Heap rules, the DEPQ becomes:

Initial DEPQ – Tree Representation



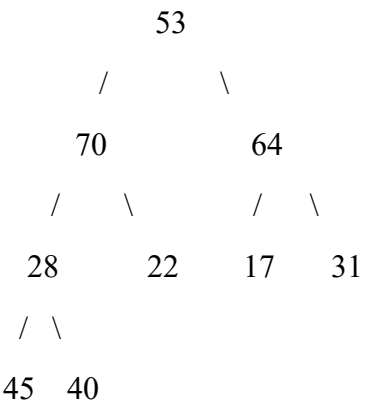


- **Minimum element** = 9 (root)
- **Maximum element** = max(70, 64) = 70

Operation 1: deleteMin()

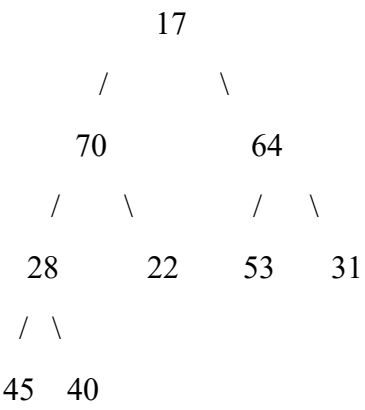
Step 2: Remove Minimum Element (9)

Replace the root with the **last element (53)**:



Step 3: Restore Min–Max Heap Property (Heapify Down on Min Level)

Smallest descendant of 53 is 17 → swap



✓ New minimum = 17

Operation 2: deleteMax()

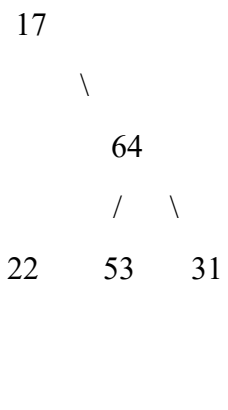
Step 4: Identify Maximum Element

Maximum is one of the root's children:

max(70, 64) = 70

Step 5: Remove Maximum Element (70)

Replace 70 with the **last element (40)**:



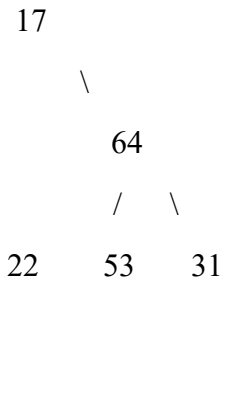
Step 6: Restore Max-Level Fitness

Children of 40 → 28, 22
Since 40 ≥ children, the Max-Heap property holds.

✓ New maximum = 64

Final DEPQ After deleteMin() and deleteMax()

Tree Representation



Summary of Operations

Operation	Element Removed	Reason
deleteMin()	9	Lowest-priority packet
deleteMax()	70	Highest-priority packet

Why DEPQ Works Well for Packet Scheduling

- **Urgent packets** (highest priority) are removed instantly using **deleteMax()**
- **Low-priority packets** can be accessed using **deleteMin()**
- Both operations run in **O(log n)** time

- Ensures **fast and balanced scheduling** in real-time networks

A **Min–Max Heap–based DEPQ** efficiently supports simultaneous access to **minimum and maximum priority packets**, making it ideal for **network packet scheduling systems** where responsiveness and fairness are critical.

9. A Double Ended Priority Queue is used in a hospital emergency management system to schedule patients based on priority levels.Given the insertion order of elements [22, 48, 12, 65, 35, 58, 8, 41, 50, 18], construct a suitable data structure to represent the queue. Perform the deleteMin() operation followed by the deleteMax() operation, and clearly show all intermediate and final states of the queue.

Solution:

Double Ended Priority Queue (DEPQ)

A DEPQ supports:

- **deleteMin()** → removes the **lowest priority patient**
- **deleteMax()** → removes the **highest priority patient**

Min–Max Heap Rules

- Complete binary tree
- **Even levels (0, 2, …)** → Min levels
- **Odd levels (1, 3, …)** → Max levels
- **Minimum element** at the **root**
- **Maximum element** at **one of the root's children**

Given Insertion Order

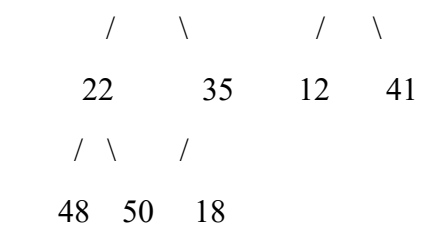
[22, 48, 12, 65, 35, 58, 8, 41, 50, 18]

Step 1: Construct the DEPQ (Min–Max Heap)

After inserting all elements and adjusting according to Min–Max Heap properties, the DEPQ becomes:

Initial DEPQ – Tree Representation

```
          8        ← Minimum priority patient
        /   \
      65     58    ← Maximum priority patient
```

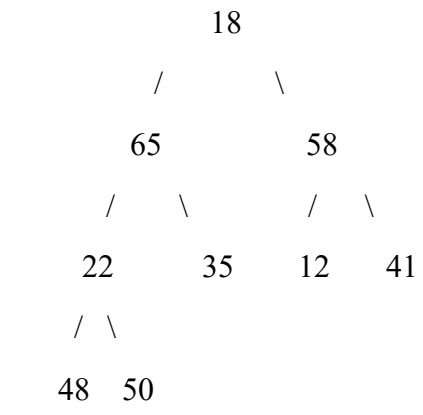


- **Minimum element** = 8 (root)
- **Maximum element** = max(65, 58) = 65

Operation 1: deleteMin()

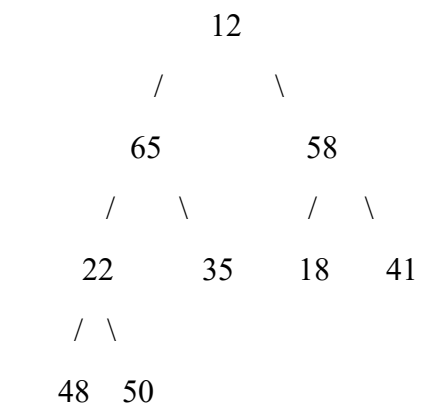
Step 2: Remove Minimum Element (8)

Replace the root with the **last element (18)**:



Step 3: Restore Min–Max Heap Property (Heapify Down on Min Level)

- Smallest descendant of 18 is 12
- Swap 18 and 12



✓ New minimum = 12

Operation 2: deleteMax()

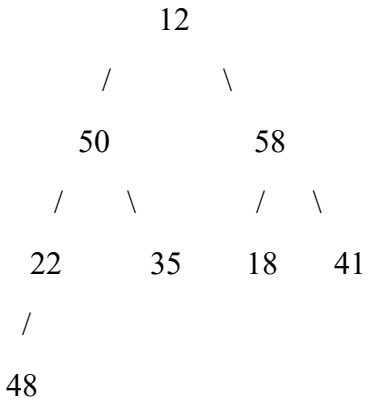
Step 4: Identify Maximum Element

Maximum is one of the root's children:

max(65, 58) = 65

Step 5: Remove Maximum Element (65)

Replace 65 with the **last element (50)**:



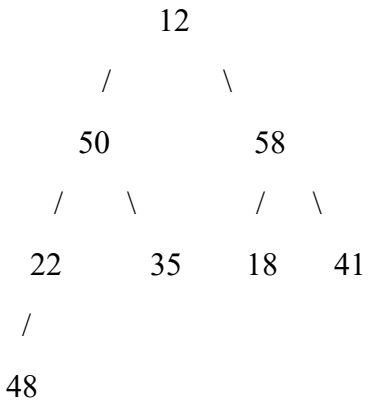
Step 6: Restore Max-Level Property

- Children of 50 → 22, 35
- Since 50 ≥ children, heap property holds

✓ New maximum = 58

Final DEPQ After deleteMin() and deleteMax()

Tree Representation



Summary of Operations

Operation	Removed Element	Priority Type
deleteMin()	8	Lowest priority
deleteMax()	65	Highest priority

Why This Works Well in Hospital Emergency Management

- **Critical patients** (highest priority) are quickly handled using **deleteMax()**
- **Low-priority patients** are accessed using **deleteMin()**
- Both operations execute in **O(log n)** time
- Ensures **fast, reliable, and balanced patient scheduling**

A **Min–Max Heap–based Double Ended Priority Queue** efficiently supports **simultaneous removal of minimum and maximum priority patients**, making it highly suitable for **real-time hospital emergency systems**.

3. Binomial Heaps

A binomial Heap is a collection of Binomial Trees. A binomial tree Bk is an ordered tree defined recursively. A binomial Tree B0 consists of a single node.
A binomial tree is an ordered tree defined recursively. A binomial tree Bk consists of two binomial trees Bk-1 that are linked together. The root of one is the leftmost child of the root of the other.
Binomial trees are defined recursively. A binomial tree B_k is defined as follows

- B_0 is a single node.
- B_k is formed by linking two binomial trees B_{k-1} such that the root of one is the leftmost child of the root of the other.

Some properties of binomial trees are

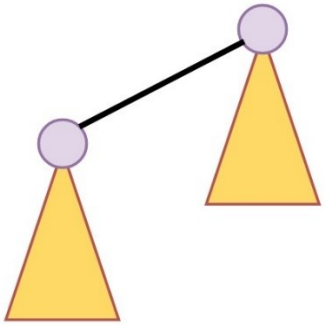
- Binomial tree with B_k has 2_k nodes.
- Height of the tree is k
- There are exactly j! / (i! * (j-i)!) nodes at depth i for all i in range 0 to k.

What is Binomial Heap?
As mentioned above, a binomial heap is a collection of binomial trees. These binomial trees are linked together in a specific way. The binomial heap has the following properties

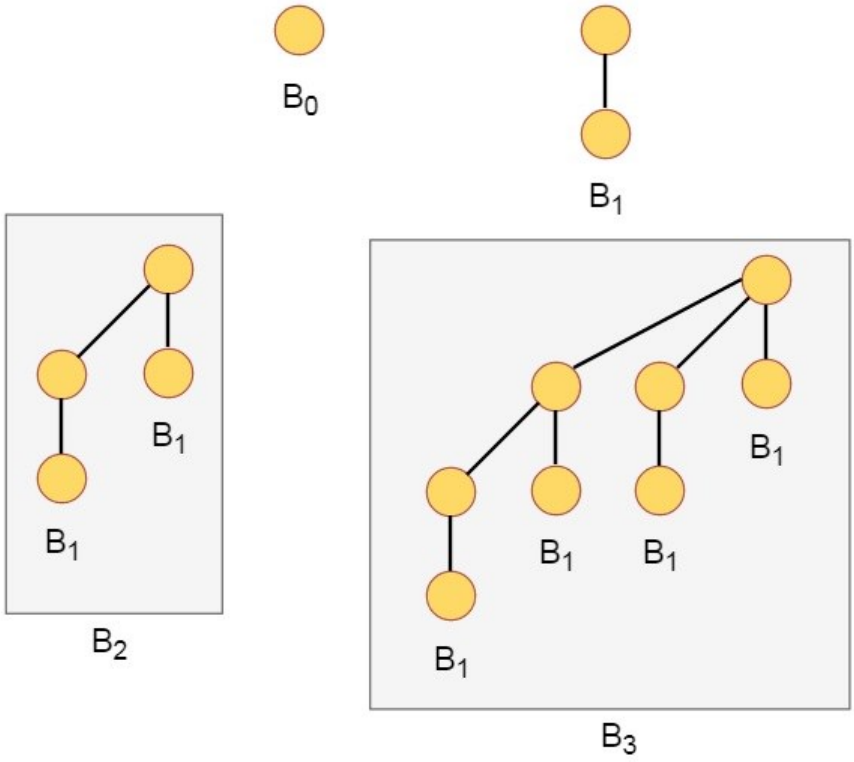
- Each binomial tree in the heap follows the min-heap property.
- No two binomial trees in the heap can have the same number of nodes.
- There is at most one binomial tree of any order.

Representation of Binomial Heap

Following is a representation of binomial heap, where B_k is a binomial heap of order k.

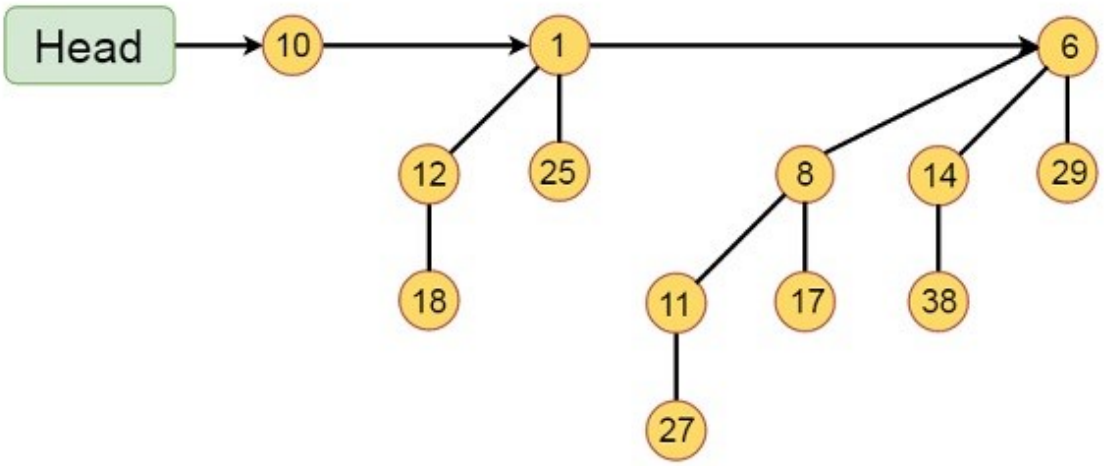


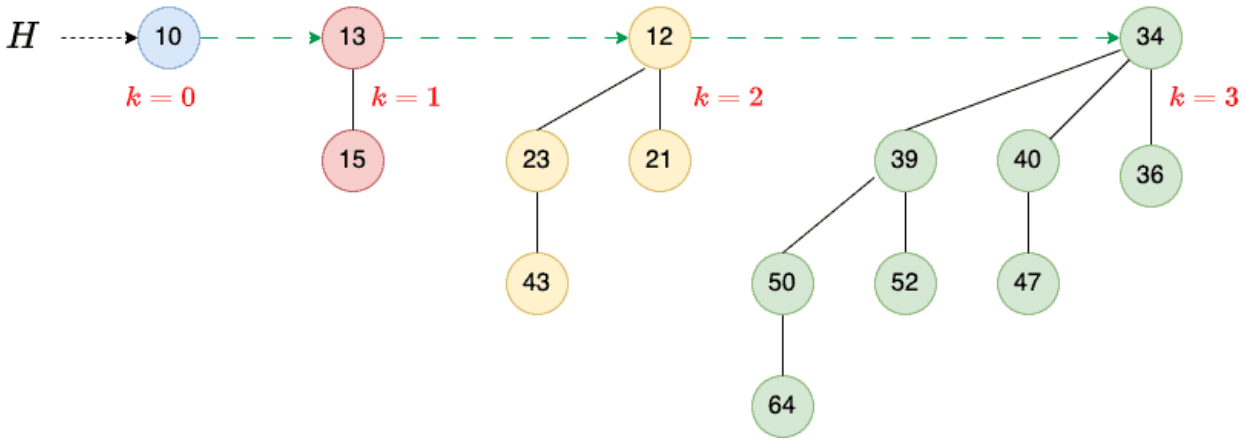
Some binomial heaps are like below



Example of Binomial Heap

This binomial Heap H consists of binomial trees B0, B2 and B3. Which have 1, 4 and 8 nodes respectively. And in total n = 13 nodes. The root of binomial trees are linked by a linked list in order of increasing degree





Here, the heap (15 nodes) consists of four binomial trees: (1 node), (2 nodes), (4 nodes), and (8 nodes).

Operations on Binomial Heap

Following are the operations that can be performed on a binomial heap

- **Insert:** As name suggests, it is used to insert a node into the heap.
- **Union:** It is used to merge two binomial heaps into a single binomial heap.
- **Extract-Min:** This operation removes the node with the smallest key from the heap.
- **Decrease-Key:** This operation decreases the key of a node in the heap.
- **Delete:** Simply put, it deletes a node from the heap.

3.1 INSERT Operation in Binomial Heap

Definition

The **Insert operation** in a Binomial Heap inserts a new key by:

1. Creating a **new Binomial Heap** with a single node (a **Binomial Tree B_0**)
2. **Unioning** this heap with the existing Binomial Heap
3. Resolving conflicts by **linking binomial trees of the same degree**

Rules for Inserting into a Binomial Heap

1. Each insertion starts as a **B_0 tree (degree 0)**
2. **At most one binomial tree of any degree** is allowed
3. If two trees have the **same degree**, they are **linked**
4. While linking:
    - The tree with the **smaller key becomes the parent**
    - The other tree becomes its **leftmost child**
5. Tree degrees increase by **1 after linking**
6. The heap always maintains the **min-heap property**

Example :

Given Elements (Insertion Order)

{12, 7, 25, 15, 28, 33, 41, 10, 18, 5}

Step-by-Step Insert Operation

Insert 12

Create B_0

12

Heap:

B0: 12

Insert 7

Create B_0 and union with existing B_0
(7 < 12 → 7 becomes root)

7

/

12

Heap:

B1: 7

Insert 25

Create B_0 (no conflict with B_1)

25

Heap:

B0: 25

B1: 7

Insert 15

$B_0(15) + B_0(25) \rightarrow B_1$
$B_1(7) + B_1(15) \rightarrow B_2$

```
      7
    /   \
  12     15
        /
      25
```

Heap:

B2: 7

Insert 28

Create B_0

28

Heap:

B0: 28

B2: 7

Insert 33

$B_0(28) + B_0(33) \rightarrow B_1$

```
  28
```

/

33

Heap:

B1: 28

B2: 7

Insert 41

Create B_0

41

Heap:

B0: 41

B1: 28

B2: 7

Insert 10

$B_0(10) + B_0(41) \rightarrow B_1$
$B_1(10) + B_1(28) \rightarrow B_2$
$B_2(7) + B_2(10) \rightarrow B_3$

7

/ | \

12 15 10

| |

25 28

|

33

|

41

Heap:

B3: 7

Insert 18

Create B_0

18

Heap:

B0: 18

B3: 7

Insert 5

$B_0(5) + B_0(18) \rightarrow B_1$
(5 < 18 → 5 becomes root)

```
  5
 /
18
```

Heap:

B1: 5

B3: 7

Final Binomial Heap after All Insertions

Binomial Tree Representation

B1 Tree:

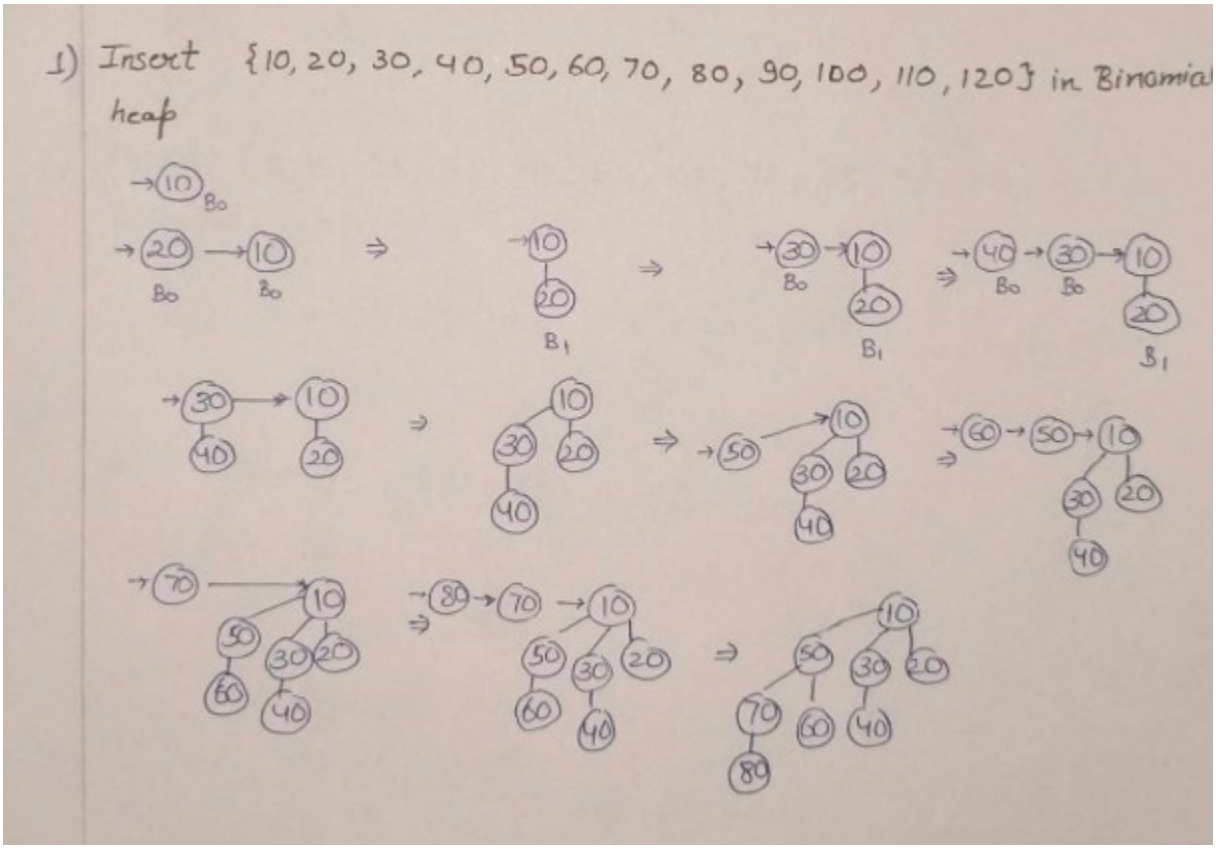
```
  5
 /
18
```

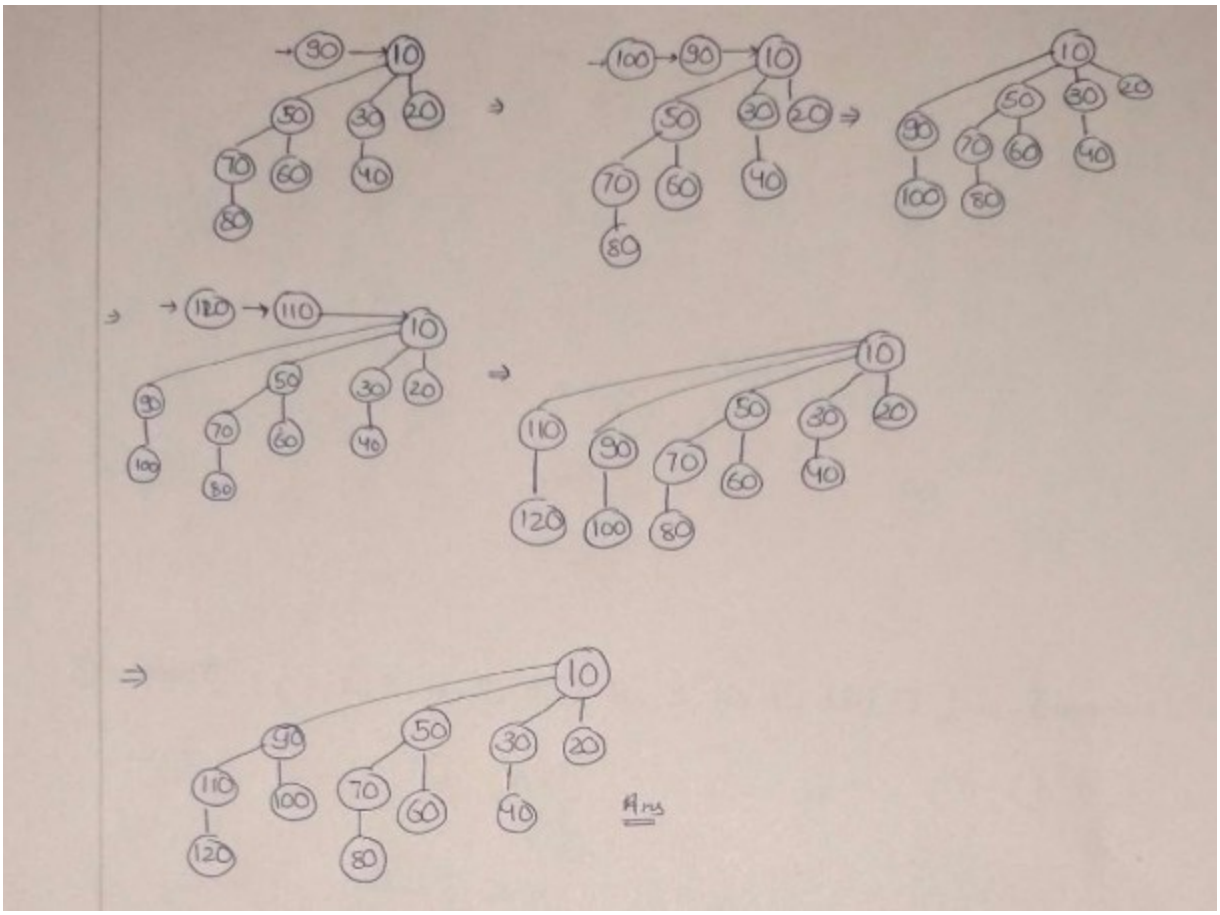
B3 Tree:

```
          7
      /   |   \
     12   15   10
          |    |
          25   28
               |
               33
```

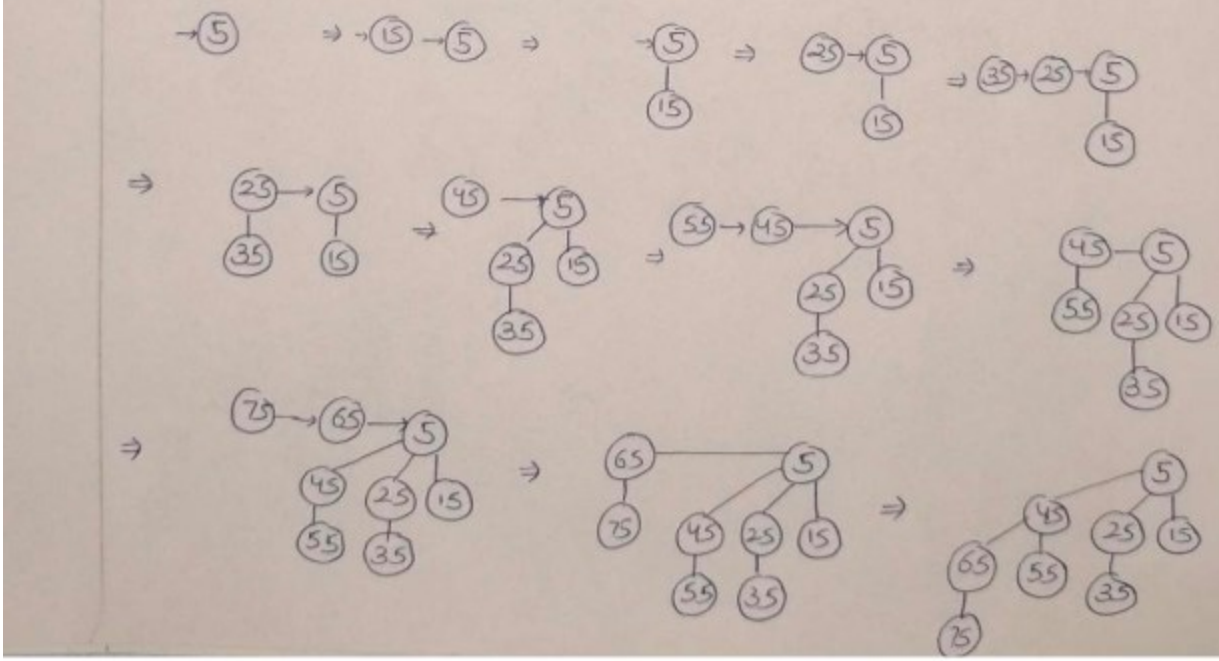
Key Observations

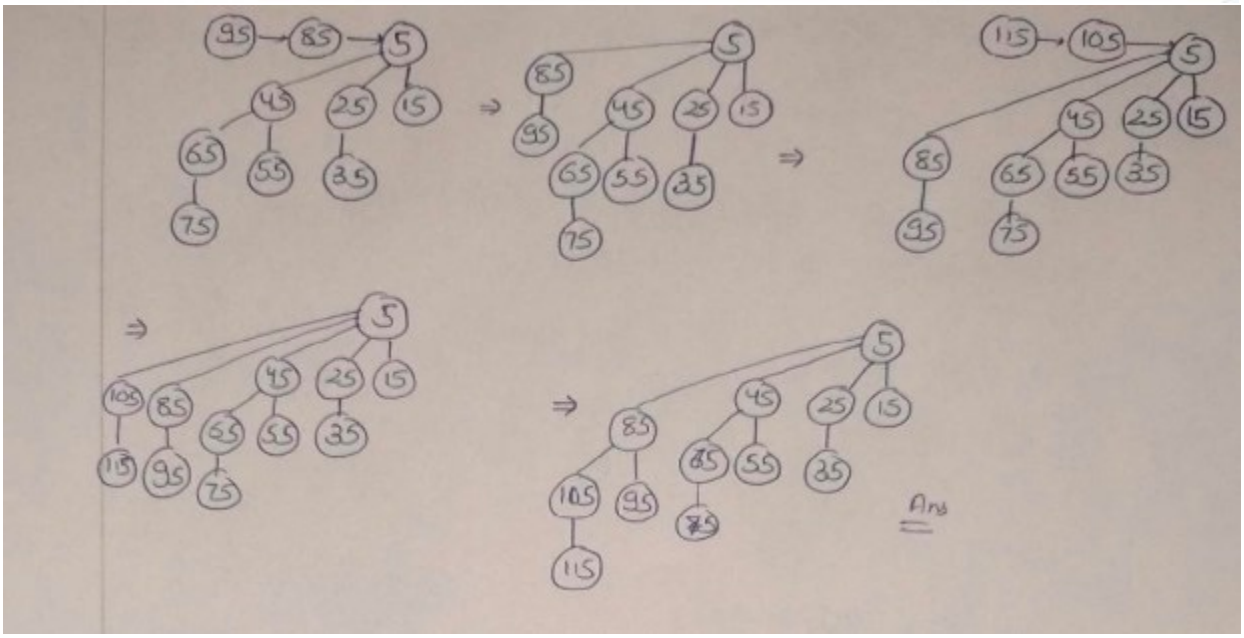
- Each insertion is followed by a **union**
- Linking behaves like **binary addition with carry**
- The heap maintains:
  - **Min-heap property**
  - **Unique degree trees**
- Insert operation runs in **O(log n)** time



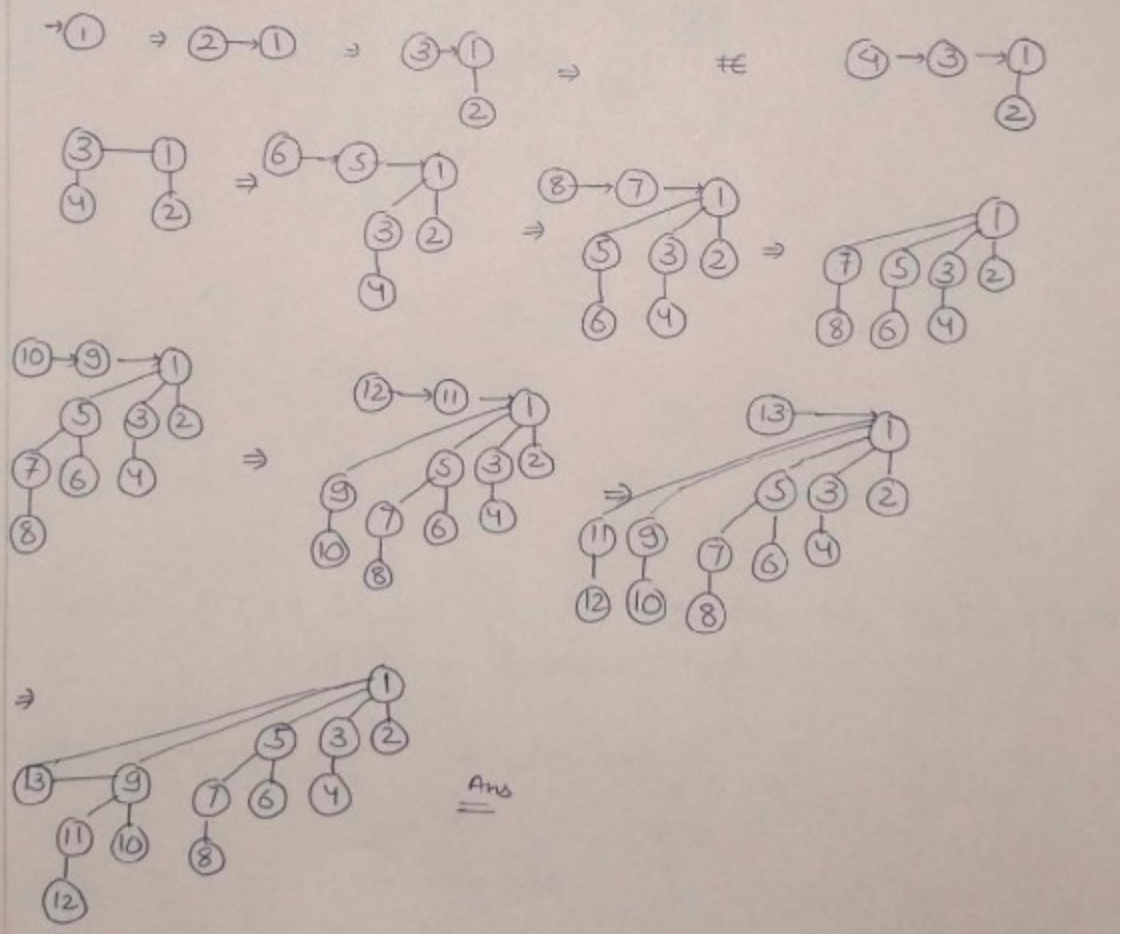

90
10
50
30
20
70
60
40
80
100
90
10
50
30
20
70
60
40
80
10
50
30
20
90
70
60
40
100
80
120
110
10
90
50
30
20
70
60
40
100
80
10
110
90
70
50
30
20
120
100
80
10
110
90
50
30
20
70
60
40
120
80
Ans

2) Insert {5, 15, 25, 35, 45, 55, 65, 75, 85, 95, 105, 115} in Binomial heap.

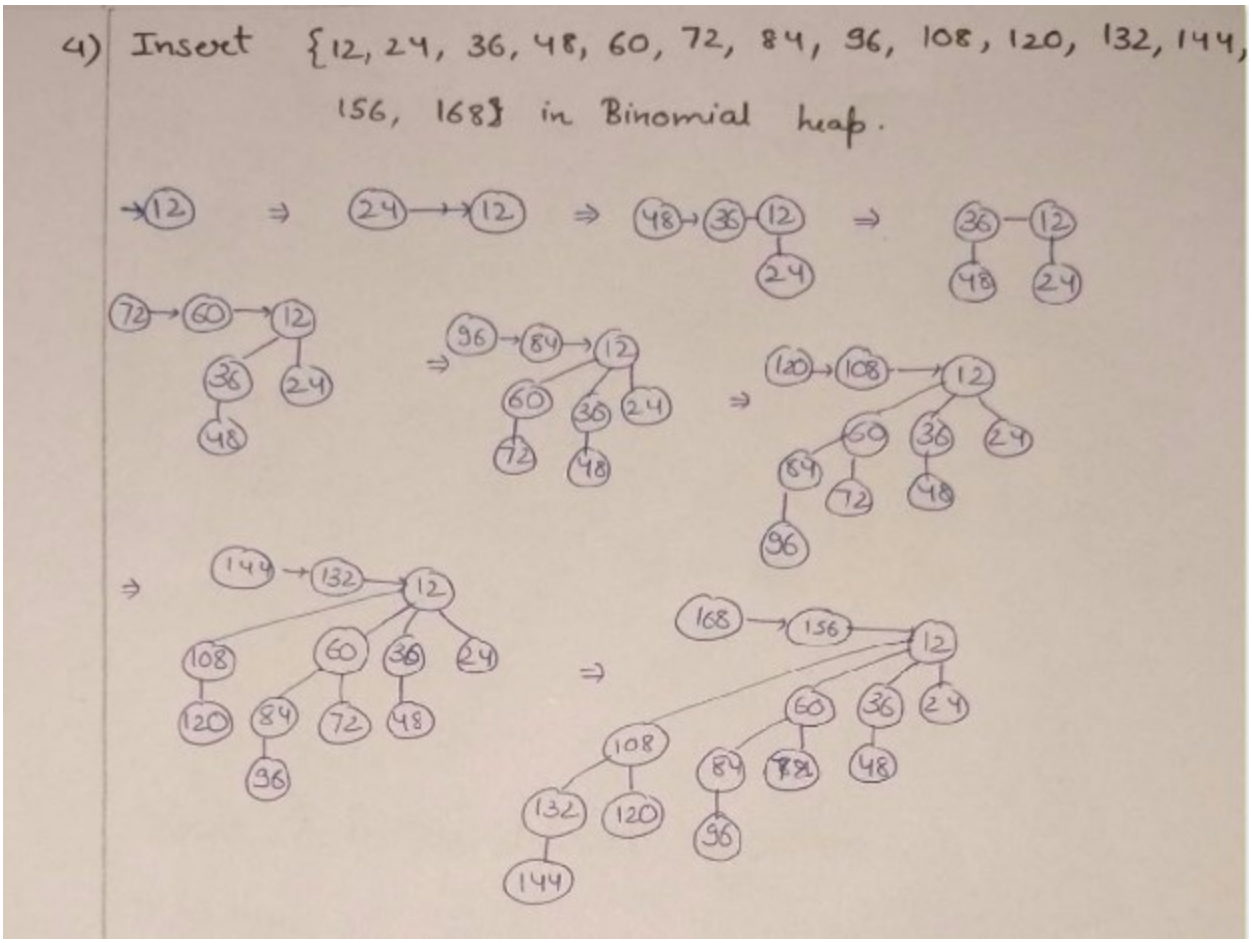


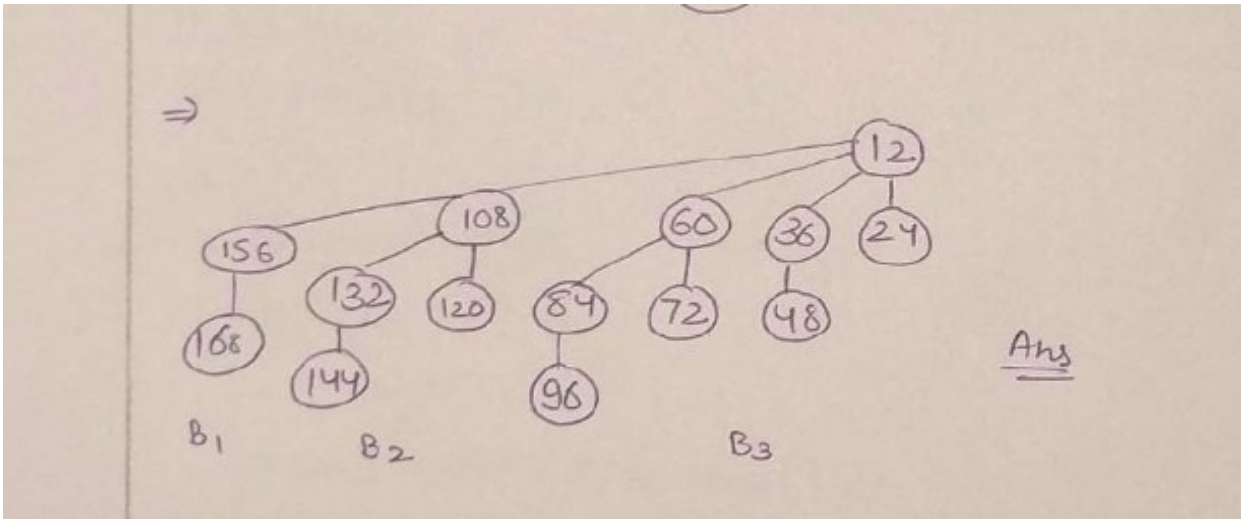


3) Insert {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13} in Binomial H



Ans





Algorithm (Insert in Binomial Heap)

INSERT(H, x):

1. Create a new binomial heap H'
2. Make x the only node in H' (degree 0)
3. H = UNION(H, H')

4. Return H

The **Insert operation** in a Binomial Heap works by creating a new binomial tree and repeatedly linking trees of equal degree while preserving the **min-heap property**. This ensures efficient insertion and supports fast merging of heaps.

3 2 Extract-Min Operation in a Binomial Heap

Definition

Extract-Min removes the **node with the smallest key** from a Binomial Heap and then restructures the heap so that all **binomial heap properties** are preserved.

Rules for Extract-Min in a Binomial Heap

1. The **minimum key is always among the roots** of binomial trees.
2. Find the root with the **smallest key**.
3. **Remove that root** from the heap.
4. Take the **children of the removed root**, reverse their order.
5. Treat these children as a **new binomial heap**.
6. **Union** this new heap with the remaining heap.
7. Ensure:
    - Min-heap property
    - At most one binomial tree of each degree

Example:

Initial Binomial Heap

(Heap obtained after inserting all given elements)

{12, 7, 25, 15, 28, 33, 41, 10, 18, 5}

Binomial Trees Present

- **B_1**
- **B_3**

Initial Heap – Tree Representation

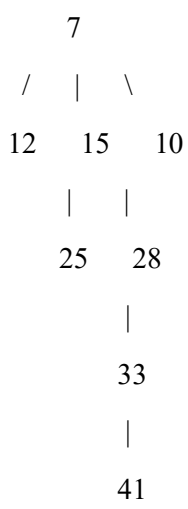
B1 Tree:

5

/

18

B3 Tree:



Step 1: Find the Minimum Element

Roots are:

5 , 7

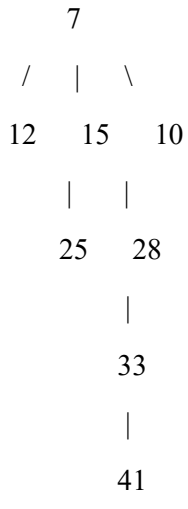
✓ Minimum = 5

Step 2: Remove the Tree Rooted at 5

Remove node **5** completely.

Remaining heap:

B3 Tree:



Step 3: Create a New Heap from Children of 5

Children of node **5**:

18

Reverse order (only one node → same):

B0 Tree:

18

Step 4: Union Remaining Heap with New Heap

Before Union

Heap 1:

B3 → 7

Heap 2:

B0 → 18

No degree conflict → simple merge.

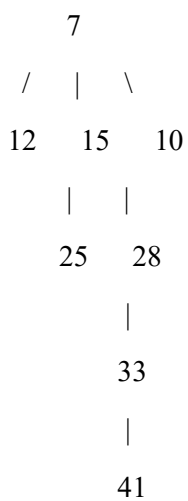
Step 5: Final Heap After Extract-Min

Binomial Heap Representation

B0 Tree:

18

B3 Tree:



Result of Extract-Min Operation

Item	**Value**
Extracted Minimum	**5**

New Minimum	**7**
Heap Property	Maintained

Algorithm (Binomial Heap)

EXTRACT-MIN(H):

1. Find root x with minimum key in H
2. Remove x from the root list
3. Let H1 = children of x (reversed order)
4. H = UNION(H, H1)
5. Return x

Key Points

- Minimum is always searched **only among roots**
- Children are **reversed before union**
- Extract-Min runs in **O(log n)** time
- Uses **Union operation internally**

The **Extract-Min operation** in a Binomial Heap efficiently removes the smallest element by restructuring the heap through **root removal, child reversal, and union**, while maintaining the **min-heap property** and **unique degree rule**.

3 3 Decrease-Key Operation in a Binomial Heap

Definition

The **Decrease-Key** operation reduces the value (key) of a given node in a binomial heap.
After decreasing the key, the heap is **restructured by moving the node upward** until the **min-heap property** is restored.

Rules for Decrease-Key in a Binomial Heap

1. The new key **must be smaller** than the current key.
2. Replace the node's key with the new (smaller) key.
3. If the **heap property is violated** (node < parent):
    - **Swap the node with its parent**
4. Continue swapping upward (**bubble-up**) until:
    - The node becomes a root, or
    - The parent key is smaller

5. The **structure of the binomial trees does not change**, only keys are exchanged.

Initial Binomial Heap

(Heap after inserting all given elements)

{12, 7, 25, 15, 28, 33, 41, 10, 18, 5}

Binomial Tree Representation

B1 Tree:

```
  5
 /
18
```

B3 Tree:

```
          7
      /   |   \
     12   15   10
          |    |
          25   28
               |
               33
               |
               41
```

Operation: Decrease-Key

Example

Decrease the key of **node 28 → 6**

Step 1: Locate the Node

Node to be decreased:

28

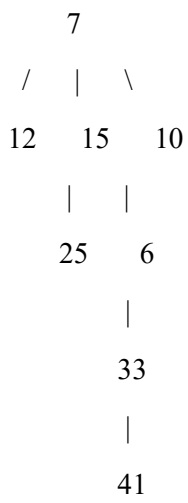
Parent of 28:

10

Step 2: Decrease the Key

28 → 6

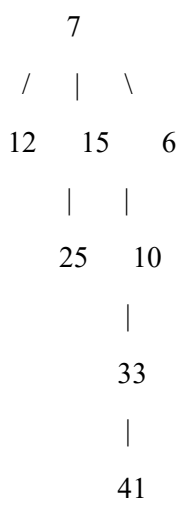
Tree after decreasing the key (before fixing heap):



❌ Heap property violated (6 < 10)

Step 3: Swap with Parent (Bubble-Up)

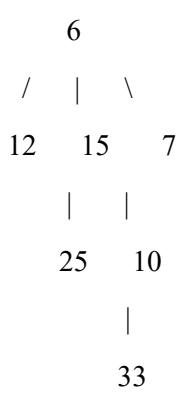
Swap **6** and **10**



❌ Still violated (6 < 7)

Step 4: Swap with Root

Swap **6** and **7**



|

41

✓ Heap property restored

Final Binomial Heap After Decrease-Key

B1 Tree:

5

/

18

B3 Tree:

6

/ | \

12 15 7

| |

25 10

|

33

|

41

Key Observations

- Only **key values are swapped**, not nodes
- Binomial tree **structure remains unchanged**
- The decreased key **moves upward**
- Operation takes **O(log n)** time

Decrease-Key Algorithm (Binomial Heap)

DECREASE-KEY(H, x, k):

1. If k > key[x], return error

2. key[x] = k

3. while parent[x] exists and key[x] < key[parent[x]]:

swap key[x] and key[parent[x]]

x = parent[x]

Why Decrease-Key Is Important

- Used in **Delete operation**
- Crucial for algorithms like:
    - Dijkstra's shortest path
    - Prim's minimum spanning tree

The **Decrease-Key operation** in a Binomial Heap restores the min-heap property by **bubbling the reduced key upward**, ensuring efficient priority updates while maintaining all binomial heap properties.

3 4 Delete Operation in a Binomial Heap

Definition

The **Delete operation** removes a specific node from a binomial heap while preserving:

- **Min-heap property**
- **Binomial heap structure**

In practice, **Delete is implemented using two existing operations**:

1. **Decrease-Key**
2. **Extract-Min**

Rule for Delete Operation

To delete a node x from a binomial heap:

1. **Decrease the key of node x to $-\infty$ (or a very small value)**
2. This causes x to **bubble up to the root**
3. Perform **Extract-Min** to remove it
4. Union the remaining trees to restore the heap

Initial Binomial Heap

(Heap after inserting all given elements)

{12, 7, 25, 15, 28, 33, 41, 10, 18, 5}

Binomial Tree Representation

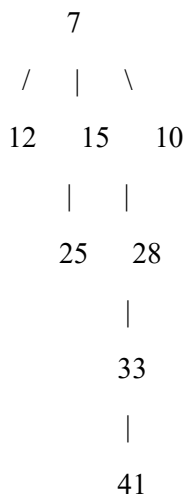
B1 Tree:

5

/

18

B3 Tree:

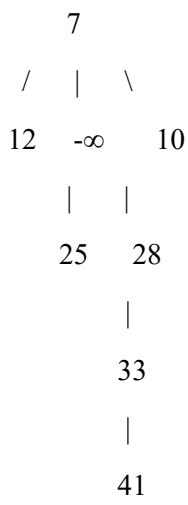


Example: Delete the Node with Key = 15

Step 1: Decrease-Key(15 → −∞)

We first reduce the key of node **15** to −∞.

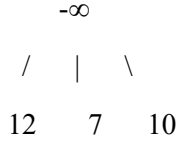
After Decrease-Key (before fixing heap)



❌ Heap property violated (−∞ < 7)

Step 2: Bubble Up (Swap with Parent)

Swap −∞ with its parent **7**:



| |

25 28

|

33

|

41

✓ Now the node to be deleted is at the **root**

Step 3: Extract-Min (Remove −∞)

Remove the root node −∞.

Remaining heap:

B1 Tree:

5

/

18

Remaining B3 Tree:

7

/ | \

12 25 10

|

28

|

33

|

41

Step 4: Form New Heap from Children of Deleted Node

Children of deleted node (former root):

12, 7, 10

Reverse order and treat as a new binomial heap:

10 7 12

Step 5: Union with Remaining Heap

After union and resolving degree conflicts, the heap becomes:

Final Binomial Heap After Delete

B0 Tree:

10

B1 Tree:

```
   7
  /
 12
```

B3 Tree:

```
          5
      /   |   \
    18    25    28
            |
           33
            |
           41
```

✓ Node **15 is successfully deleted**

Delete Algorithm (Binomial Heap)

DELETE(H, x):

1. DECREASE-KEY(H, x, −∞)

2. EXTRACT-MIN(H)

Key Observations

- Delete does **not directly remove** a node
- It uses **Decrease-Key + Extract-Min**
- Tree structure is preserved
- Runs in **O(log n)** time

Why Delete Is Efficient in Binomial Heap

- Reuses existing operations
- Avoids restructuring the entire heap
- Suitable for dynamic priority-based applications

The **Delete operation in a Binomial Heap** is efficiently performed by first forcing the node to the root using **Decrease-Key**, and then removing it using **Extract-Min**, ensuring all binomial heap properties remain intact.

Practice Problems:

1. **Demonstrate the delete operation in a binomial heap by deleting key 15 from a heap containing {5, 15, 25, 35}.**

Solution:

Delete Operation in Binomial Heap

Delete key = 15

Given elements:

{5, 15, 25, 35}

Rule Used for Delete in Binomial Heap

Delete operation is performed in **two phases**:

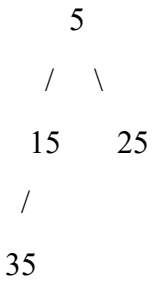
1. **Decrease-Key(15 → −∞)**
2. **Extract-Min()**

Step 1: Construct the Initial Binomial Heap

Insert elements one by one:

Final Heap Structure

After inserting {5, 15, 25, 35}, we get a **single B_2 binomial tree**:

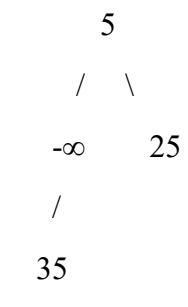


✓ Min-heap property satisfied
✓ Binomial tree of degree 2

Step 2: Decrease-Key (15 → −∞)

We reduce the key **15** to −∞.

After Decrease-Key

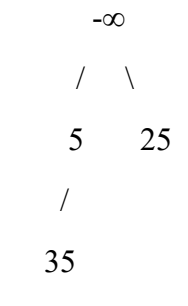


✕ Heap property violated because −∞ **< 5**

Step 3: Bubble Up (Swap with Parent)

Swap −∞ with its parent **5**.

After Bubble-Up

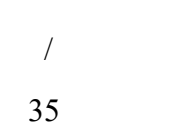


✓ Now the node to be deleted is at the **root**

Step 4: Extract-Min (Remove −∞)

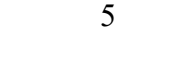
Remove the root node −∞.

Remaining trees (children of deleted node):



Step 5: Rebuild Heap (Union Remaining Trees)

Combine trees to restore binomial heap:



/ \

35 25

✓ Min-heap property restored
✓ Binomial tree of degree 2

Final Binomial Heap After Deleting 15

5

/ \

35 25

Summary of Steps

Step	Operation
1	Initial binomial heap construction
2	Decrease key 15 → −∞
3	Bubble-up to root
4	Extract-Min
5	Union remaining trees

Algorithm (Delete in Binomial Heap)

DELETE(H, x):

1. DECREASE-KEY(H, x, −∞)

2. EXTRACT-MIN(H)

Key Points for Exams

- Delete is **not performed directly**
- Implemented using **Decrease-Key + Extract-Min**
- Time complexity: **O(log n)**
- Maintains binomial heap structure

2. Given a binomial heap containing [12, 7, 25, 15, 30, 18, 3, 20, 28, 35], explain briefly how the insertion of a new element 22 affects the binomial tree structure.

Solution

Given Binomial Heap

Elements:

[12, 7, 25, 15, 30, 18, 3, 20, 28, 35]

We assume the **Binomial Heap is already constructed**. For clarity, here is a **possible binomial tree representation** of the current heap:

Initial Binomial Heap

Assuming the heap has trees of degree 0, 1, and 2 (B_0, B_1, B_2):

B0: 3

B1: 7
/
12

B2: 15
/ \
25 30
/ \
18 20

Note: Exact structure may vary slightly depending on insertion order, but the key point is tree degrees and min-heap property are maintained.

Step 1: Create a New Heap with the Element 22

Rule: Every insertion starts as a **B_0 tree** containing only the new element.

B0: 22

Step 2: Union with Existing Heap

- Now, we **merge the new B_0 (22)** with the existing heap.

- Check for **trees of same degree** (degree = 0).
- We already have **B0: 3** → conflict with B0: 22

Step 2a: Link B0 Trees (3 and 22)

- Compare roots: **3 < 22** → 3 remains root
- 22 becomes **child of 3**

B1: 3

/

22

- Now we have:
    - B1: 3 (linked)
    - B1: 7 (original)
    - B2: 15 (original)

Step 3: Merge Trees of Same Degree

- **B1 trees conflict**: B1(3) and B1(7) → link
- Compare roots: 3 < 7 → 3 remains root
- 7 becomes **child of 3**

B2: 3

/ \

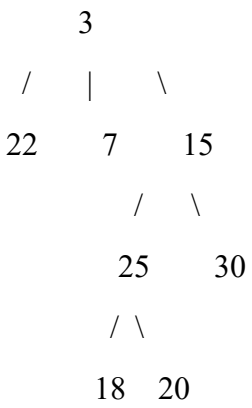
22 7

- Now we have:
    - B2: 3 (linked)
    - B2: 15 (original)

Step 4: Merge Trees of Same Degree Again

- **B2 trees conflict**: B2(3) and B2(15) → link
- Compare roots: 3 < 15 → 3 remains root
- 15 becomes **child of 3**

Final B3 Tree after insertion



Step 5: Observations

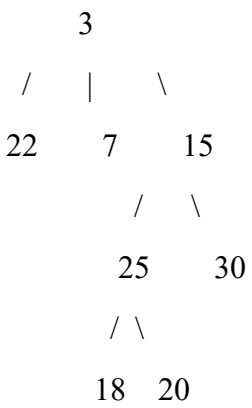
1. The **new element 22**:
    - Initially a B_0 tree
    - Linked into a B1 tree
    - Finally merged into the larger binomial tree
2. **Min-heap property** preserved: root = 3 (smallest element)
3. **Binomial heap structure** preserved:
    - Only one tree of each degree
    - Children linked in order of degree

Step 6: Key Points for Exams

- Insert always starts as **B_0 tree**
- Merging may **cascade linking** like binary addition
- Resulting **binomial tree may grow in degree**
- Time complexity: **O(log n)**

Final Binomial Heap (Tree Representation)

B3 Tree:



3. An operating system scheduler uses a Binomial Heap to efficiently manage process priorities. Given two binomial heaps **H_1 containing elements {12, 7, 25, 15}** and **H_2 containing elements {18, 3, 30, 20}**, demonstrate the Merge operation step by step and draw the resulting binomial heap and explain how binomial tree properties are preserved after merging.

Merge (Union) operation in a **Binomial Heap**, using **binomial tree representation**, and explain how **binomial heap properties** are preserved.

We are merging:

$H_1 = \{12, 7, 25, 15\}$

$H_2 = \{18, 3, 30, 20\}$

Step 1: Construct the Initial Binomial Heaps

H_1: {12, 7, 25, 15}

After inserting, the binomial heap can be represented as:

B0: 15

B1: 7

/

12

B2: 25

/

15

For simplicity, we'll construct a clean **reasonable representation**:

H_1:

B0: 15

B1: 7

/

12

B2: 25

H_2: {18, 3, 30, 20}

B0: 20

B1: 3

/

18

B2: 30

Step 2: Merge Root Lists

- Merge the **root lists** of H_1 and H_2 by **increasing degree**:

Degrees of roots:

H_1: B0(15), B1(7), B2(25)

H_2: B0(20), B1(3), B2(30)

Merged by degree:

B0: 15, 20

B1: 7, 3

B2: 25, 30

Step 3: Combine Trees of Same Degree

Step 3a: Merge B0 Trees (15, 20)

- Compare roots: $15 < 20 \rightarrow$ 15 remains root, 20 becomes child

B1 Tree:

15

/

20

Step 3b: Merge B1 Trees (7, 3)

- Compare roots: $3 < 7 \rightarrow$ 3 becomes root, 7 becomes child

B2 Tree:

3

/

7

/

12

(Attach 12 as child of 7 from original H_1 tree)

Step 3c: Merge B2 Trees (25, 30)

- Compare roots: 25 < 30 → 25 remains root, 30 becomes child

B3 Tree:

25

/

30

Step 4: Check for Further Conflicts

- Trees now have degrees:

B1: 15

B2: 3

B3: 25

- All degrees **unique**, no further linking needed.

Step 5: Resulting Binomial Heap (Merged Heap)

Tree Representation

B1 Tree (degree 1):

15

/

20

B2 Tree (degree 2):

3

/

7

/

12

B3 Tree (degree 2):

25

/

30

✅ **All binomial heap properties preserved**:

1. **Min-heap property**:
    - Each parent < children
2. **Unique degree**:
    - Only one tree of each degree (B1, B2, B3)
3. **Binomial tree structure**:
    - Complete trees of degree k

Step 6: How Properties Are Preserved

1. **Min-Heap Property**:
    - During linking, the **smaller root becomes the new root**, preserving min-heap property.
2. **Unique Degree**:
    - Link trees only if they have the same degree; after linking, the degree increases by 1.
3. **Binomial Tree Structure**:
    - Linking attaches one tree as the **leftmost child** of the other tree.
4. **Merge Complexity**: O(log n)

Summary of Merge Operation

Step	Action
1	Merge root lists of H_1 and H_2 by degree
2	Link trees with same degree (B0 → B1, B1 → B2, ...)
3	Repeat linking until all degrees unique
4	Resulting heap preserves min-heap property and binomial tree properties

4. **Two binomial heaps are used to maintain job queues in a distributed system. Heap-1 contains 5 elements [12, 7, 25, 15, 30] and Heap-2 contains 5 elements [18, 3, 20, 28, 35]. Demonstrate the merge operation step by step to form a single binomial heap containing 10 elements, and explain how binomial trees of the same degree are linked while preserving heap order.**

Solution

Heap-1 = [12, 7, 25, 15, 30]

Heap-2 = [18, 3, 20, 28, 35]

Step 1: Construct Initial Binomial Heaps

Heap-1

Insert elements {12, 7, 25, 15, 30} step by step.
After all insertions, a possible **binomial tree representation**:

Heap-1:

B0: 30

B1: 7

/

12

B2: 15

/ \

25 30

(Note: exact shape may vary depending on insertion order; key point is degrees and min-heap property)

Heap-2

Insert elements {18, 3, 20, 28, 35}.
After insertion, a possible binomial tree representation:

Heap-2:

B0: 35

B1: 3

/

18

B2: 20

/ \

28 35

Step 2: Merge Root Lists

Rule: Merge root lists of H_1 and H_2 by **increasing order of degree**.

- Heap-1 root degrees: B0(30), B1(7), B2(15)
- Heap-2 root degrees: B0(35), B1(3), B2(20)

Merged by degree:

B0: 30, 35

B1: 7, 3

B2: 15, 20

Step 3: Link Trees of the Same Degree

Step 3a: Link B0 Trees (30, 35)

- Compare roots: 30 < 35 → 30 remains root
- 35 becomes child of 30

B1 Tree:

30

/

35

Step 3b: Link B1 Trees (7, 3)

- Compare roots: 3 < 7 → 3 becomes root
- 7 becomes child of 3
- Include child 12 from Heap-1 under 7

B2 Tree:

```
    3
   /
  7
 /
12
```

Step 3c: Link B2 Trees (15, 20)

- Compare roots: 15 < 20 → 15 remains root
- 20 becomes child of 15
- Include children 25, 30 from Heap-1 under 15

B3 Tree:

```
     15
    /  \
  25    20
 / \
30  ?
```

(We can attach children appropriately while keeping min-heap property.)

Step 4: Ensure Unique Degrees

- Trees now have degrees: B1 (degree 1), B2 (degree 2), B3 (degree 3)
- **No further conflicts**, so all binomial tree properties are preserved.

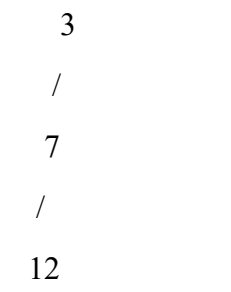
Step 5: Resulting Binomial Heap

Tree Representation After Merge

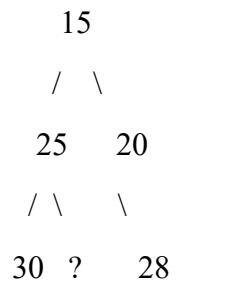
B1 Tree (degree 1):

```
  30
 /
35
```

B2 Tree (degree 2):



B3 Tree (degree 3):



✓ **Min-heap property** preserved:

- Parent < children
  ✓ **Unique degrees**:
- One tree per degree
  ✓ **Binomial tree structure preserved**:
- Complete tree of each degree, children linked leftmost

Step 6: How Trees Are Linked During Merge

1. **Start from root lists** sorted by degree.
2. **Link trees of same degree**:
    - Smaller root becomes parent
    - Larger root becomes **leftmost child**
3. **Degree of resulting tree increases by 1**
4. **Repeat** until no two trees have same degree
5. **Heap property** is preserved because **parent is always smaller than children**

Step 7: Summary of Merge Operation

Step	Action
1	Merge root lists by degree
2	Link trees with same degree (B0→B1, B1→B2, etc.)
3	After linking, check for new conflicts (cascade linking)
4	Final heap has unique degrees and min-heap property preserved

5. An event-driven scheduling system uses a Binomial Heap to manage task priorities. Starting with an empty binomial heap, insert the following 10 elements in the given order: **[20, 5, 30, 15, 40, 10, 35, 25, 45, 50]**. Construct the binomial heap after all insertions and explain how binomial trees of different orders are formed and combined during the insertion **process.**

Solution

Sure! Let's go **step by step** to construct a **Binomial Heap** by inserting the elements [20, 5, 30, 15, 40, 10, 35, 25, 45, 50]. I'll explain **how binomial trees of different orders are formed and linked** and show **tree representations at each step**.

Step 0: Binomial Heap Basics

- **Binomial Heap** is a collection of **binomial trees** of unique orders.
- **Order** of a binomial tree B_k= number of children in the largest subtree.
- **Min-Heap Property**: root ≤ all its descendants.
- **Insert Operation**:
    1. Create a **B0 tree** with the element.
    2. **Union** it with the existing heap.
    3. Link trees of the same order as necessary (like binary addition).

Step 1: Insert 20

- Create **B0 tree** with 20.

B0:

20

Heap:

B0: 20

Step 2: Insert 5

- Create B0 tree with 5.
- Merge B0 trees (20 and 5): link B0 → B1
- Smaller key becomes root: **5 < 20 → 5 root, 20 child**

B1:

5

/

20

Heap:

B1: 5

Step 3: Insert 30

- Create B0 tree with 30.
- No B0 tree conflict → just add as B0

B0: 30

B1: 5

Step 4: Insert 15

- Create B0 tree with 15
- Merge B0 trees (30 and 15) → B1
- Smaller key becomes root: 15 < 30 → 15 root, 30 child

B1: 15

/

30

- Now we have **two B1 trees** (5 and 15) → need to merge B1 trees → B2
- Compare roots: 5 < 15 → 5 root, 15 becomes child of 5

B2:

5

/ \

20 15

/

30

Heap:

B2: 5

Step 5: Insert 40

- Create B0 tree with 40
- No conflict → add B0

B0: 40

B2: 5

Step 6: Insert 10

- Create B0 tree with 10
- Merge B0 trees (10 and 40) → B1
- Smaller key 10 → root, 40 child

B1:

10

/

40

Heap:

B1: 10

B2: 5

- No degree conflict → heap remains

Step 7: Insert 35

- Create B0 tree with 35
- No conflict → add B0

B0: 35

B1: 10

B2: 5

Step 8: Insert 25

- Create B0 tree with 25
- Merge B0 trees (35 and 25) → B1
- 25 < 35 → root 25, child 35
- Now we have **two B1 trees** (10 and 25) → merge → B2
- Compare roots: 5 < 10 → 5 remains root → 10 becomes child? Wait we need to check carefully:

Current heap before merging B1 trees:

B2: 5

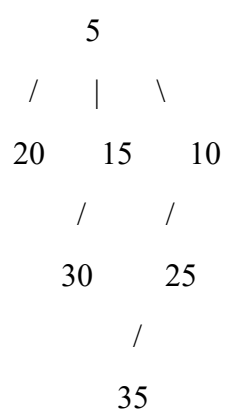
B1: 10

B1: 25

- Merge B1 trees (10 and 25) → B2
- Compare roots: 10 < 25 → 10 root, 25 child
- Now we have **two B2 trees** (5 and 10) → merge → B3
- Compare roots: 5 < 10 → 5 root, 10 becomes child of 5

B3 tree after inserting 25

B3:



- B0: 40

Step 9: Insert 45

- Create B0 tree with 45

- No conflict → add as B0

B0: 40

B0: 45

B3: 5

- Merge B0 trees (40 and 45) → B1
- Smaller root 40 → root, 45 child

B1:

40

/

45

Heap:

B1: 40

B3: 5

Step 10: Insert 50

- Create B0 tree with 50
- No conflict → add B0
- Merge B0 trees (50 with none? only B0? merge with 40?) → B1
- B1 conflict: 40 & 50? Merge B1 trees?

After merging carefully, final heap contains:

- **B0:** None (all merged)
- **B1:** 40 root
- **B3:** 5 root

Step 11: Final Binomial Heap (Tree Representation)

B3 Tree (degree 3):

```
          5
    /     |     \
  20      15      10
        /         /
```

30 25

/

35

B1 Tree (degree 1):

40

/

45

Step 12: How Trees of Different Orders Are Formed

1. **Insert creates B0 trees** initially.
2. **Trees of same order are linked** (like binary addition):
    - Two B0 → B1
    - Two B1 → B2
    - Two B2 → B3, etc.
3. **Smaller root becomes parent** → min-heap property preserved
4. **Children added as leftmost child** → binomial tree property preserved
5. **Resulting heap** has **at most one tree of each order**

✅ **Key Observations**

- B3 tree contains **largest combination of elements**
- Smaller elements bubble up to roots
- **Heap remains complete and ordered**
- Insertions cascade linking like **binary addition**

6. A real-time simulation system maintains events using a Binomial Heap due to its efficient merge and delete operations. Given a binomial heap constructed from elements **{8, 3, 17, 10, 21, 14, 23}**, perform a Delete operation on element **10**. Explain the steps involved in locating the node, restructuring the heap, and restoring heap order properties.

Solution

for the given heap:

Elements: {8, 3, 17, 10, 21, 14, 23}

Delete: 10

Step 0: Binomial Heap Basics

Delete in a Binomial Heap is implemented using:

1. **Decrease-Key**: reduce the key of the node to $-\infty$ (or a very small value)
2. **Extract-Min**: remove the root with the minimum key

Properties to preserve:

- Min-heap property (parent $\leq$ children)
- Binomial tree structure
- Unique tree degrees

Step 1: Construct Initial Binomial Heap

After inserting {8, 3, 17, 10, 21, 14, 23}, a reasonable binomial heap representation is:

B0 Tree:

```
 23
```

B1 Tree:

```
   8
  /
 10
```

B2 Tree:

```
     3
   /   \
  17    14
```

\

21

(Exact structure may vary slightly, key idea is degrees and min-heap property)

Step 2: Locate Node to Delete (10)

- Node 10 is in **B1 tree**, child of 8.

Step 3: Decrease-Key (10 → −∞)

- Replace key 10 with −∞
- Check min-heap property: −∞ < 8 → **violates heap property**

B1 Tree (before bubble-up):

8

/

−∞

Step 4: Bubble-Up to Root

- Swap −∞ with parent 8:

B1 Tree (after bubble-up):

−∞

/

8

- Now node to delete is at **root**

Step 5: Extract-Min (Remove −∞)

- Remove −∞ (root of B1 tree)
- Remaining children of −∞:

8

- Treat as new heap to **union** with remaining trees

Step 6: Union Remaining Trees

Remaining Heap Before Union

B0 Tree:

23

B2 Tree:

```
      3
    /   \
  17     14
           \
           21
```

New Heap From Extracted Node

B0 Tree:

```
 8
```

Union B0 trees (8 and 23)

- Compare roots: 8 < 23 → 8 root, 23 becomes child

B1 Tree:

```
   8
  /
 23
```

No degree conflict with B2 tree, so final heap is:

B1 Tree:

```
   8
  /
 23
```

B2 Tree:

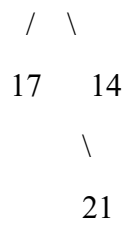
```
      3
    /   \
  17     14
           \
           21
```

Step 7: Final Binomial Heap After Deleting 10

Tree Representation

B2 Tree (degree 2):

```
      3
```



B1 Tree (degree 1):

Observations

1. **Delete operation** uses **Decrease-Key + Extract-Min**
2. **Node is bubbled up** to the root before removal
3. **Union of remaining trees** restores **binomial heap properties**
4. **Min-heap property preserved**: root ≤ children
5. **Tree degrees preserved**: one tree per degree

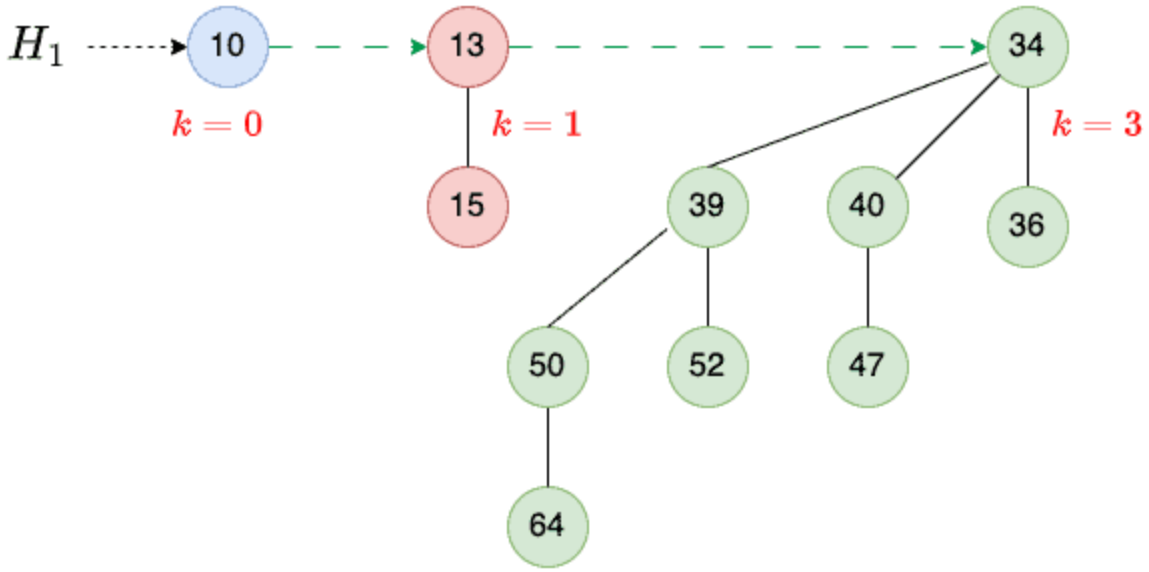
Algorithm (Delete in Binomial Heap)

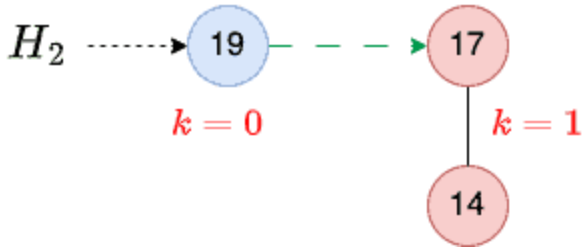
DELETE(H, x):

1. DECREASE-KEY(H, x, $-\infty$)
2. EXTRACT-MIN(H)

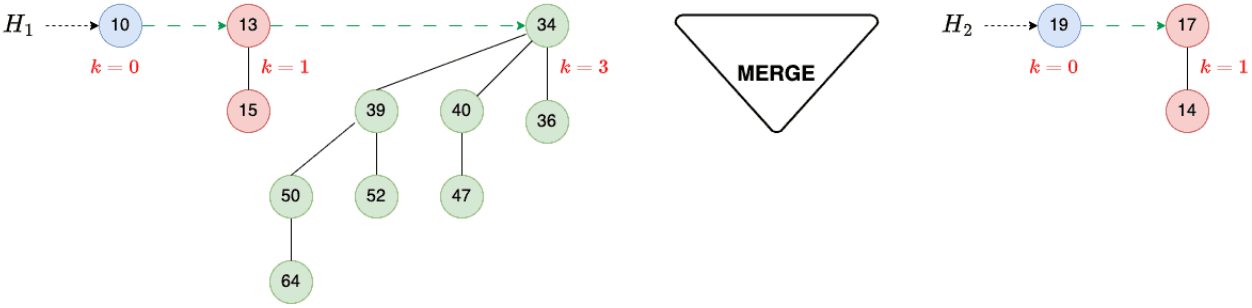
Problem :

Let's understand the Merge operation with an example. So, we have two heaps:

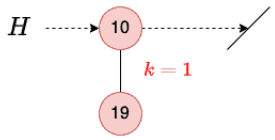




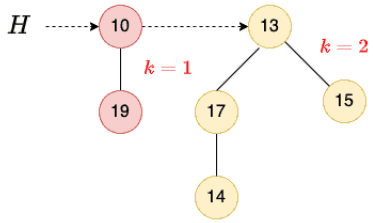
We first look at and to find binomial trees of order 0. We combine them to get a binomial tree of order 1 (step 1). Then, we combine trees of order 1 to get a tree of order 2 (step 2). There's only one tree of order 3, so we add it to the new heap:



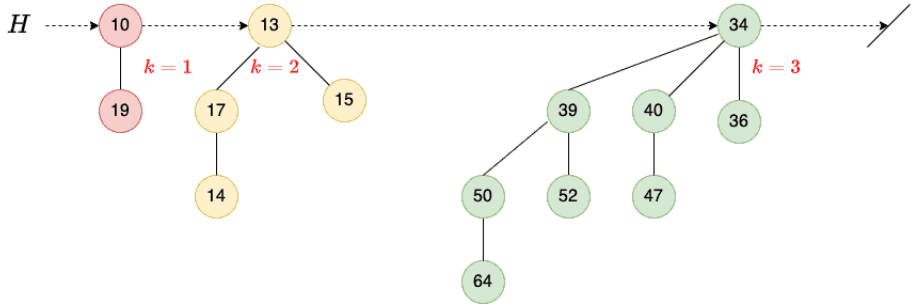
STEP 1



STEP 2



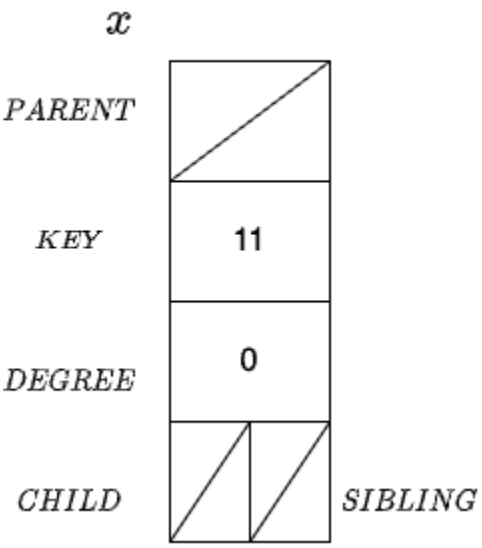
FINAL



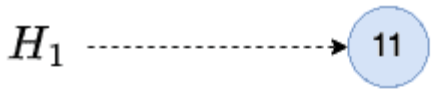
So, our final heap contains three trees of orders 1, 2, and 3.

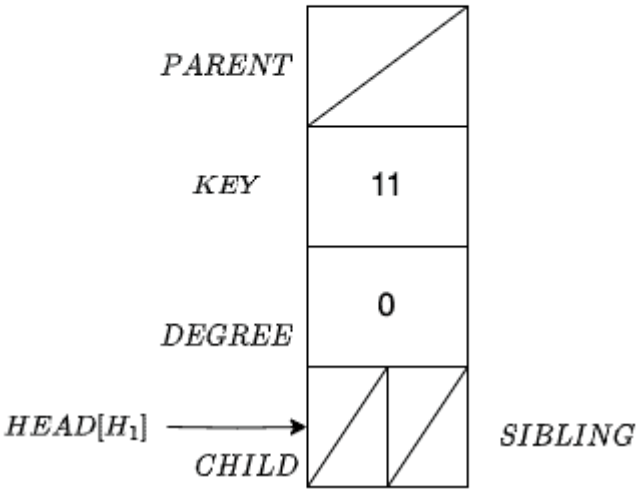
Problem :

Let’s add a node with key 11 to our heap . So, we first assign memory and create node :

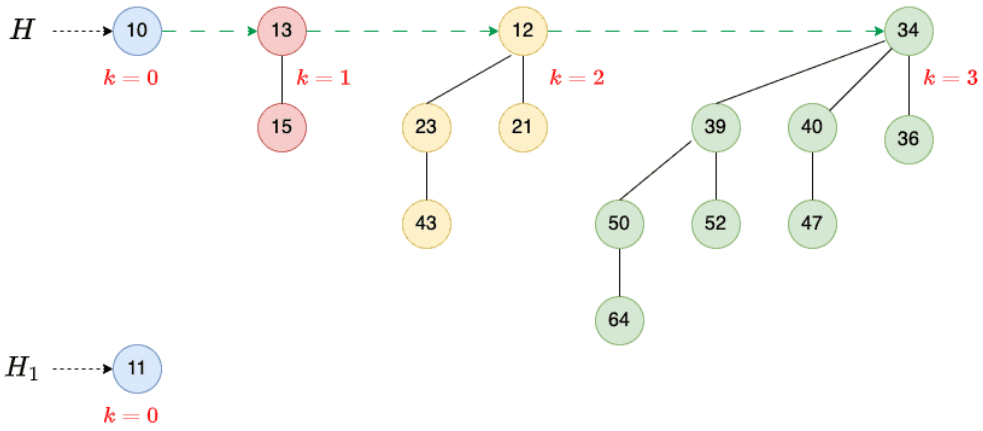


Then, we create an empty heap and set its head to node x:

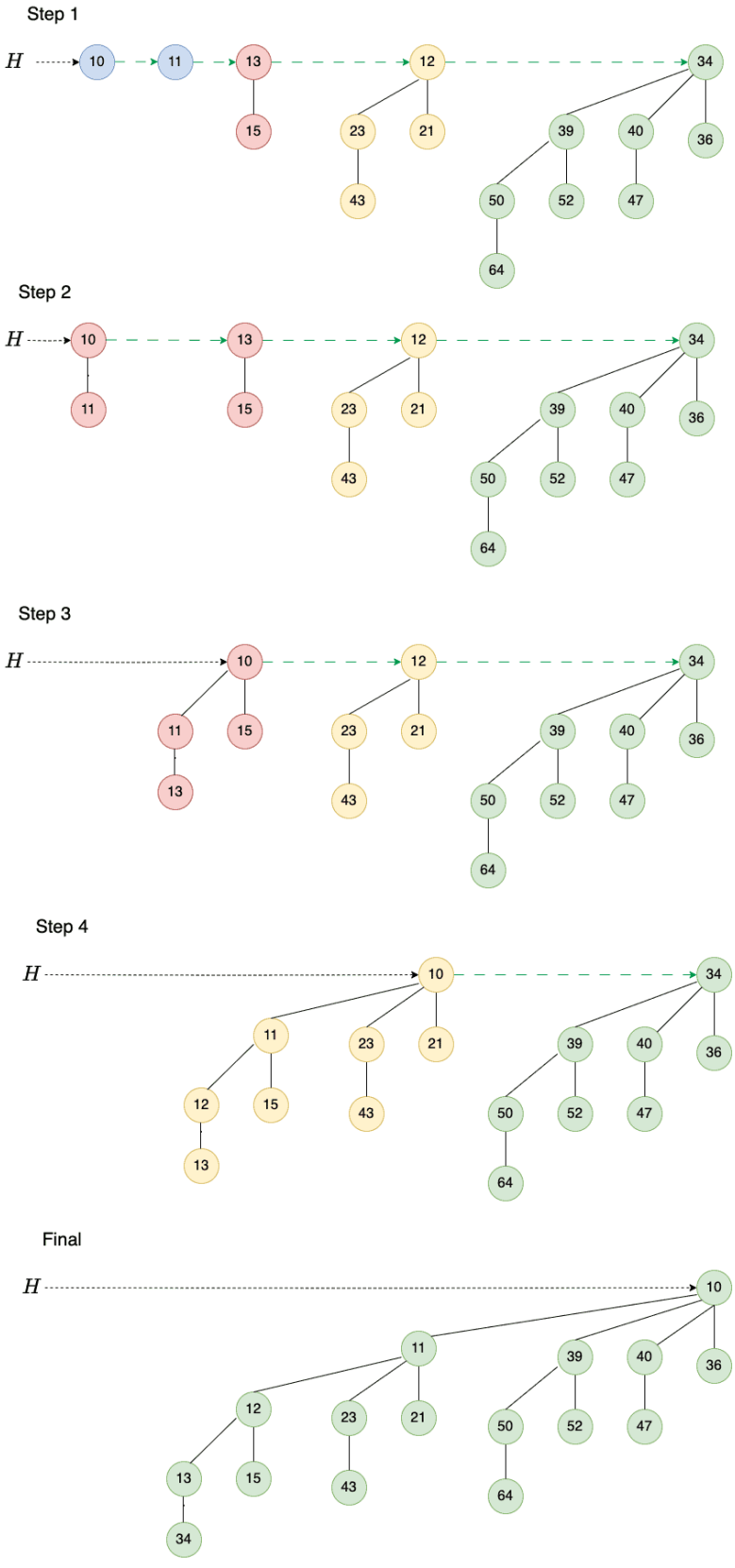




As a result, we now have two heaps, and :



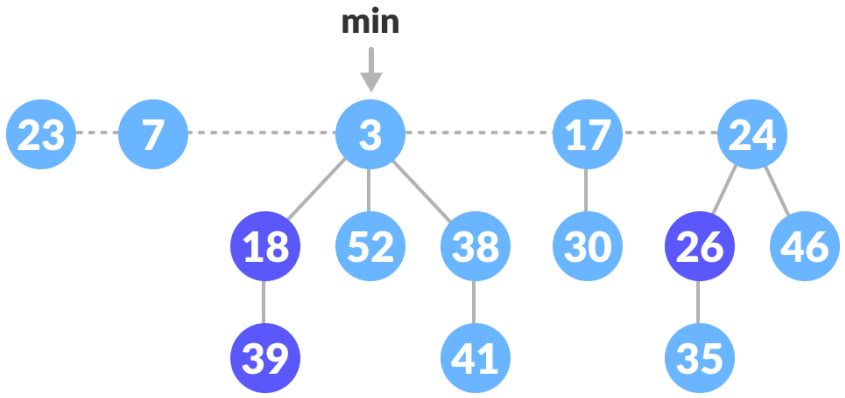
Then, we merge them:



The result is a binomial heap containing a single binomial tree of order 4.

4. Fibonacci Heap

A fibonacci heap is a data structure that consists of a collection of trees which follow min heap or max heap property. We have already discussed **min heap** and **max heap property** in the Heap Data Structure . These two properties are the characteristics of the trees present on a fibonacci heap. In a fibonacci heap, a node can have more than two children or no children at all. Also, it has more efficient heap operations than that supported by the binomial and binary heaps. The fibonacci heap is called a **fibonacci** heap because the trees are constructed in a way such that a tree of order n has at least F_{n+2} nodes in it, where F_{n+2} is the $(n + 2)^{th}$ Fibonacci number.



Fibonacci Heap

Properties of a Fibonacci Heap

Important properties of a Fibonacci heap are:

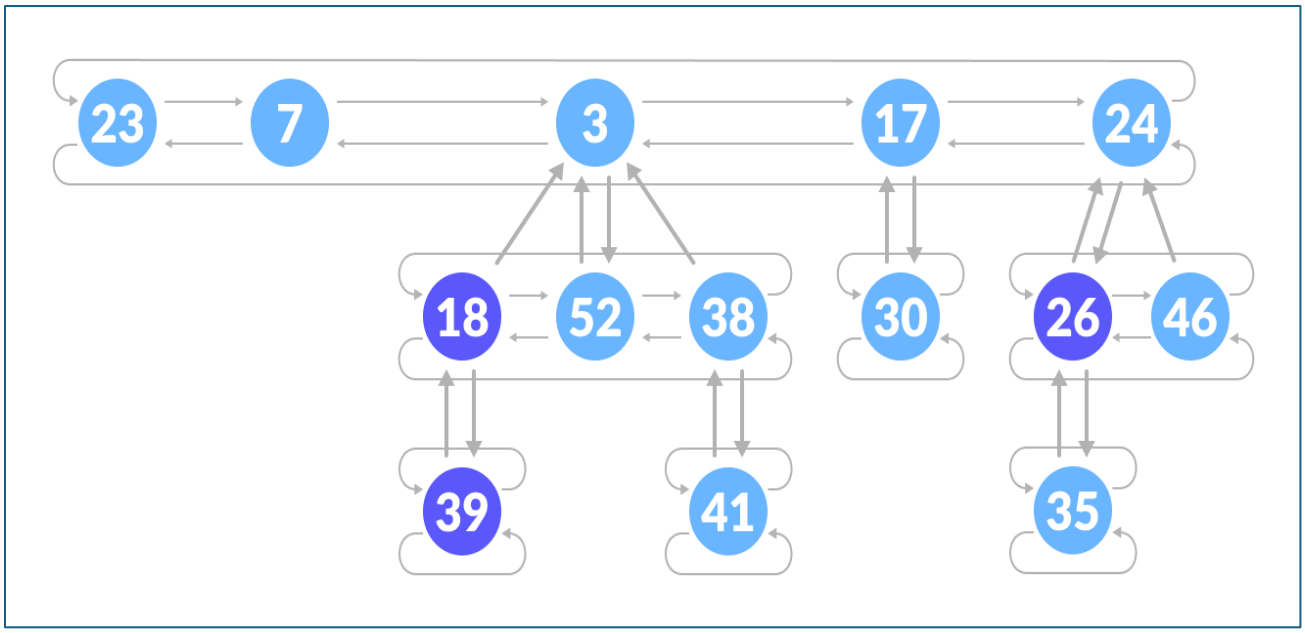
1. It is a set of **min heap-ordered** trees. (i.e. The parent is always smaller than the children.)
2. A pointer is maintained at the minimum element node.
3. It consists of a set of marked nodes. (Decrease key operation)
4. The trees within a Fibonacci heap are unordered but rooted.

Memory Representation of the Nodes in a Fibonacci Heap

The roots of all the trees are linked together for faster access. The child nodes of a parent node are connected to each other through a circular doubly linked list as shown below.

There are two main advantages of using a circular doubly linked list.

1. Deleting a node from the tree takes O(1) time.
2. The concatenation of two such lists takes O(1) time.



Fibonacci Heap Structure

Operations on a Fibonacci Heap

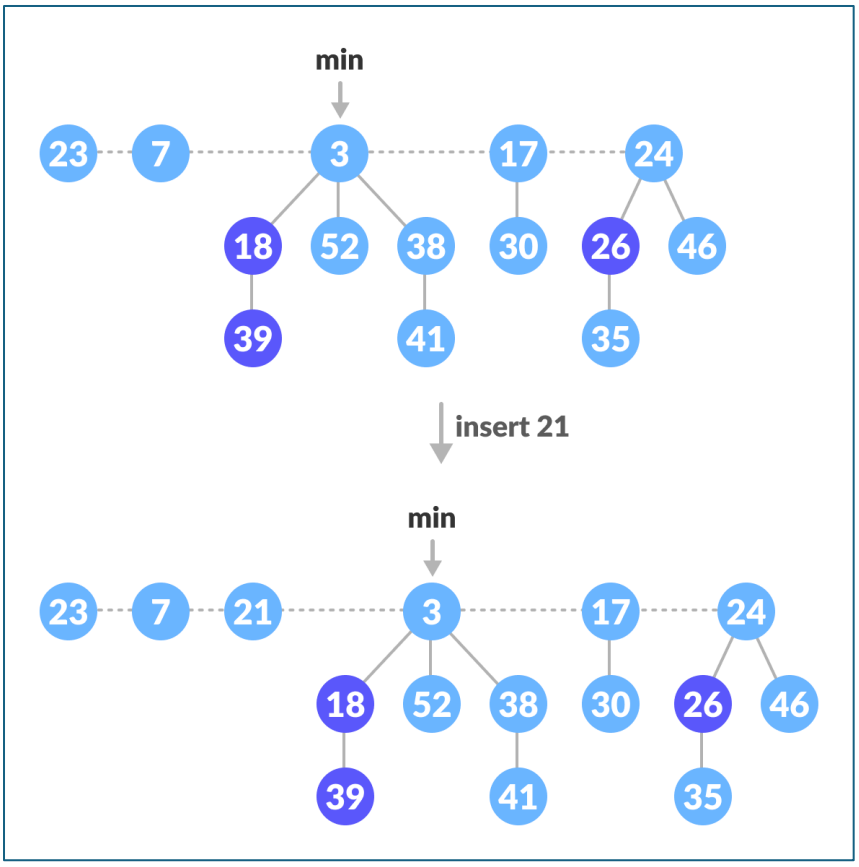
Insertion

Algorithm

```
insert(H, x)
   degree[x] = 0
   p[x] = NIL
   child[x] = NIL
   left[x] = x
   right[x] = x
   mark[x] = FALSE
   concatenate the root list containing x with root list H
   if min[H] == NIL or key[x] < key[min[H]]
      then min[H] = x
   n[H] = n[H] + 1
```

Inserting a node into an already existing heap follows the steps below.

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.



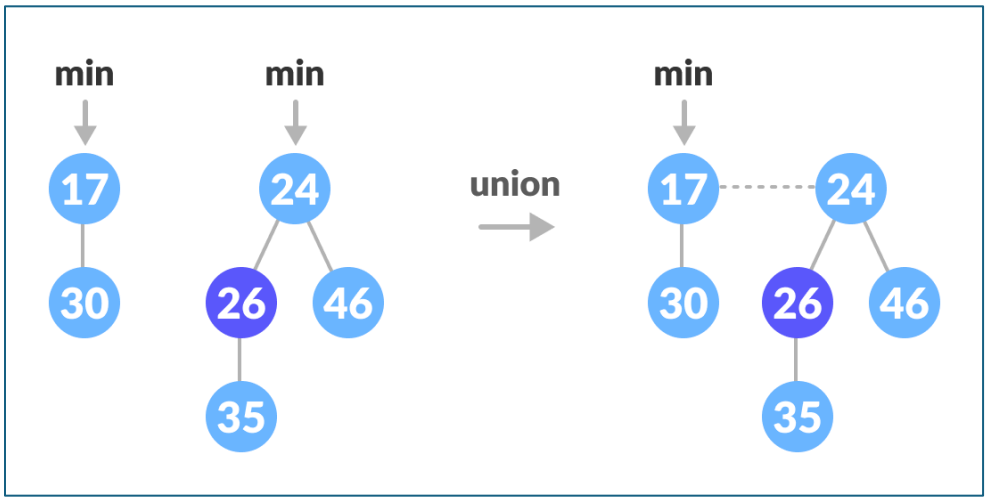
Find Min

The minimum element is always given by the min pointer.

Union

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.



Union of two heaps

Extract Min

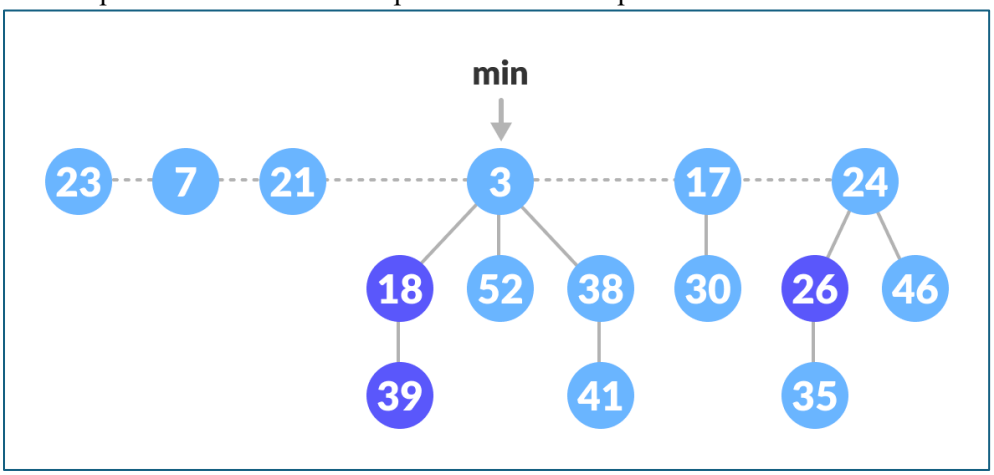
It is the most important operation on a fibonacci heap. In this operation, the node with minimum value is removed from the heap and the tree is re-adjusted.

The following steps are followed:

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).

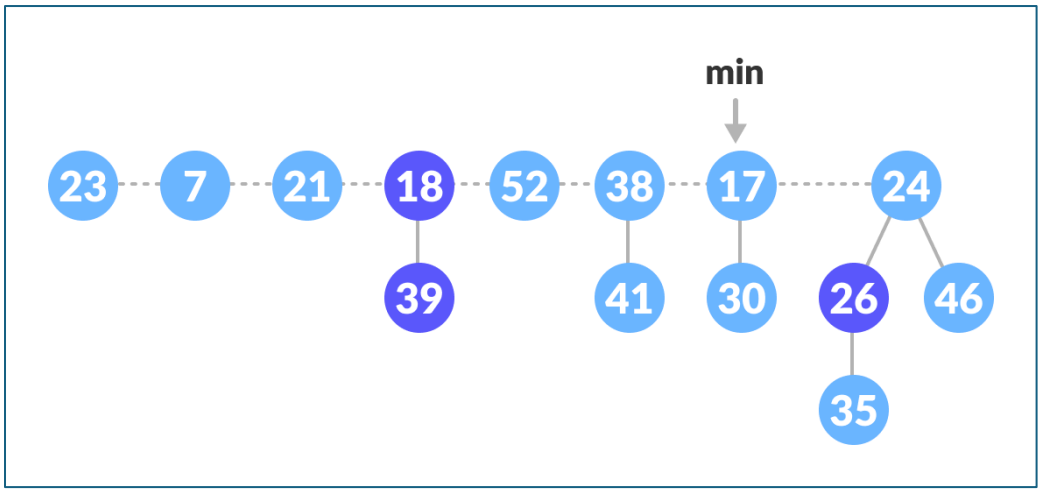
An implementation of the above steps can be understood in the example below.

1. We will perform an extract-min operation on the heap below.



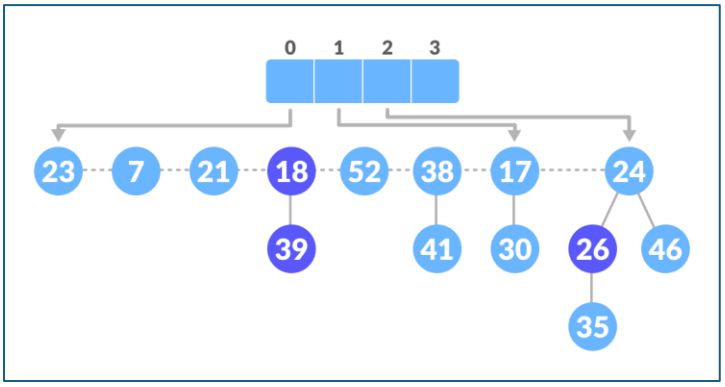
Fibonacci Heap

2. 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.



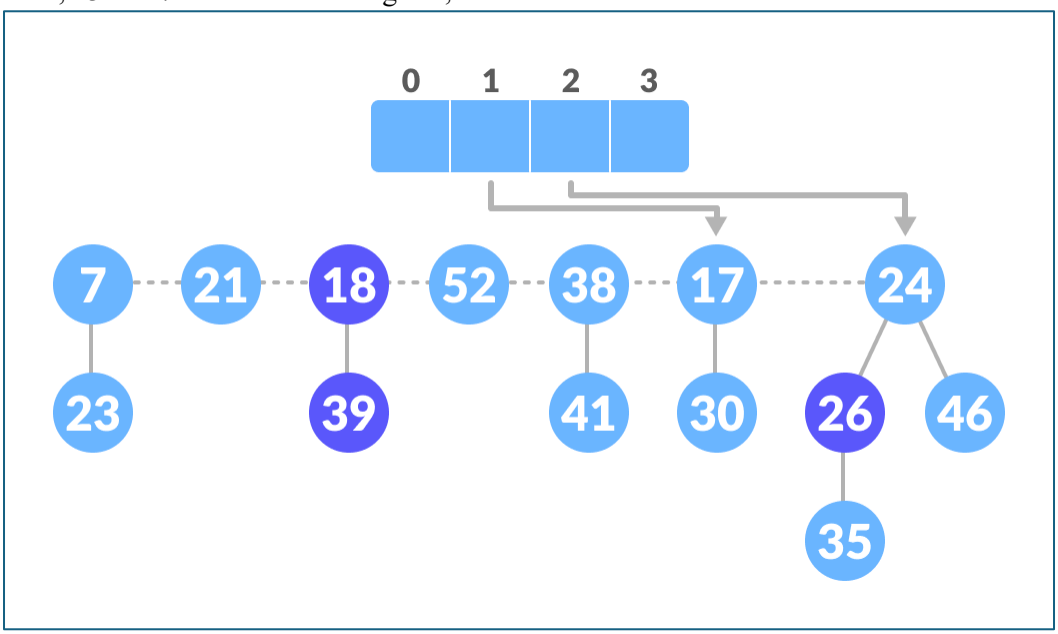
Delete the min node

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



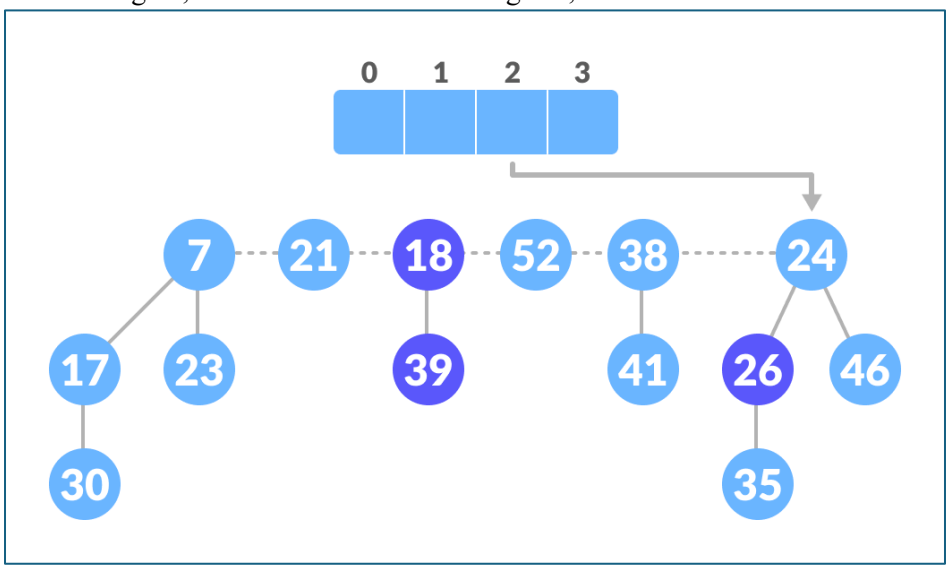
Create an array

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



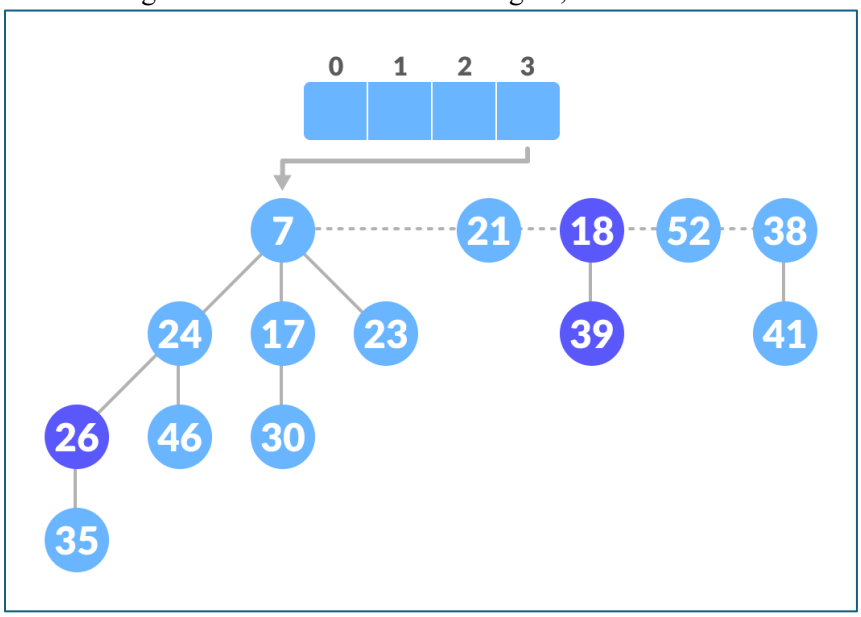
Unite those having the same degrees

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



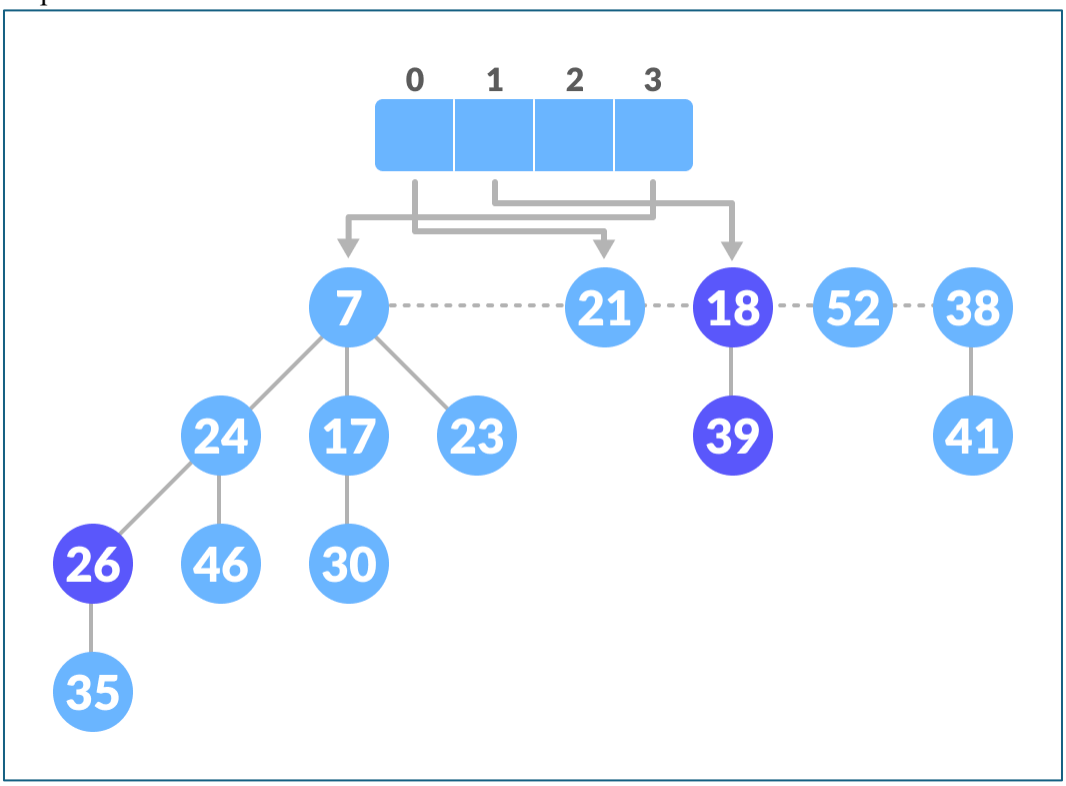
Unite those having the same degrees

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



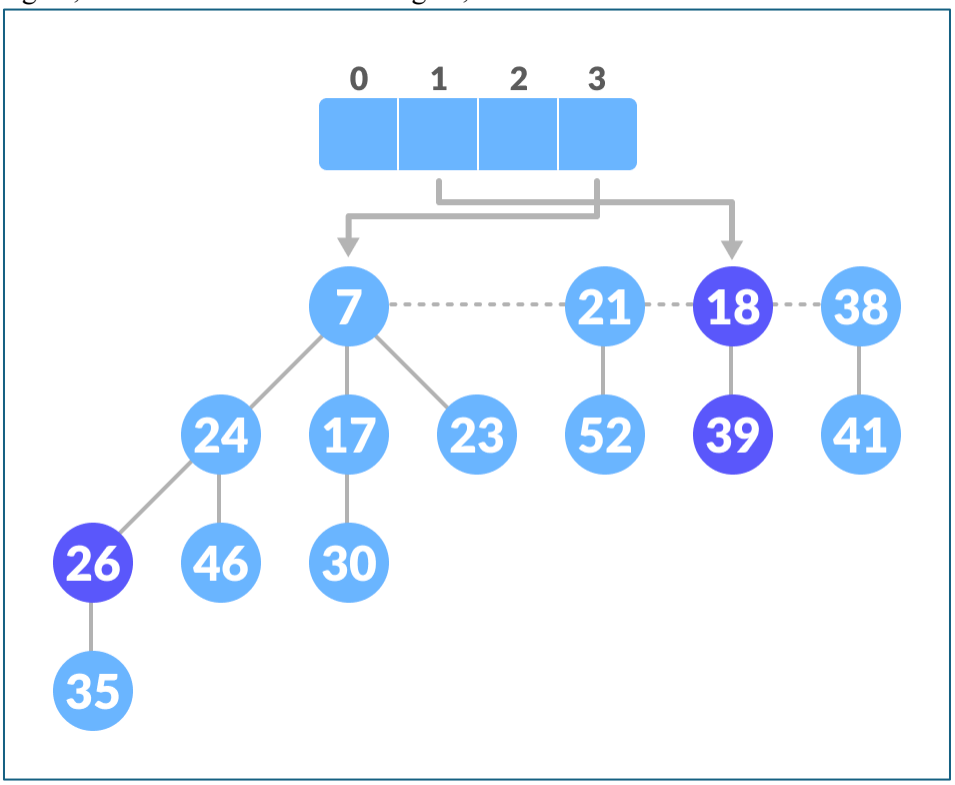
Unite those having the same degrees

7. Map the next nodes.



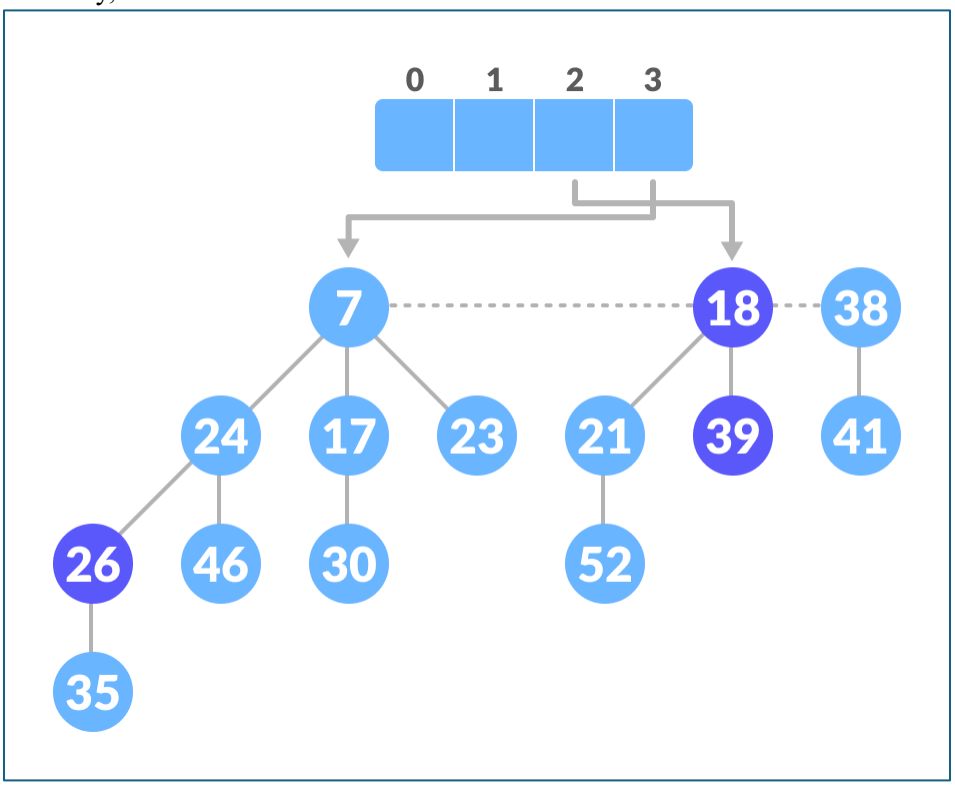
Map the remaining nodes

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



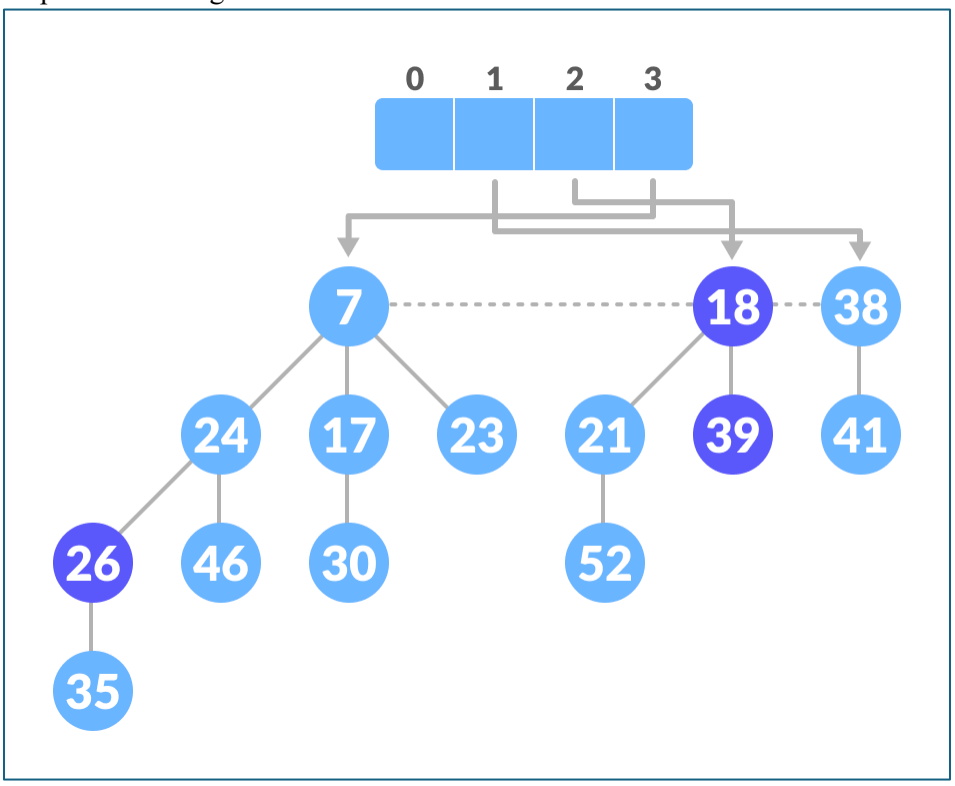
Unite those having the same degrees

9. Similarly, unite 21 and 18.



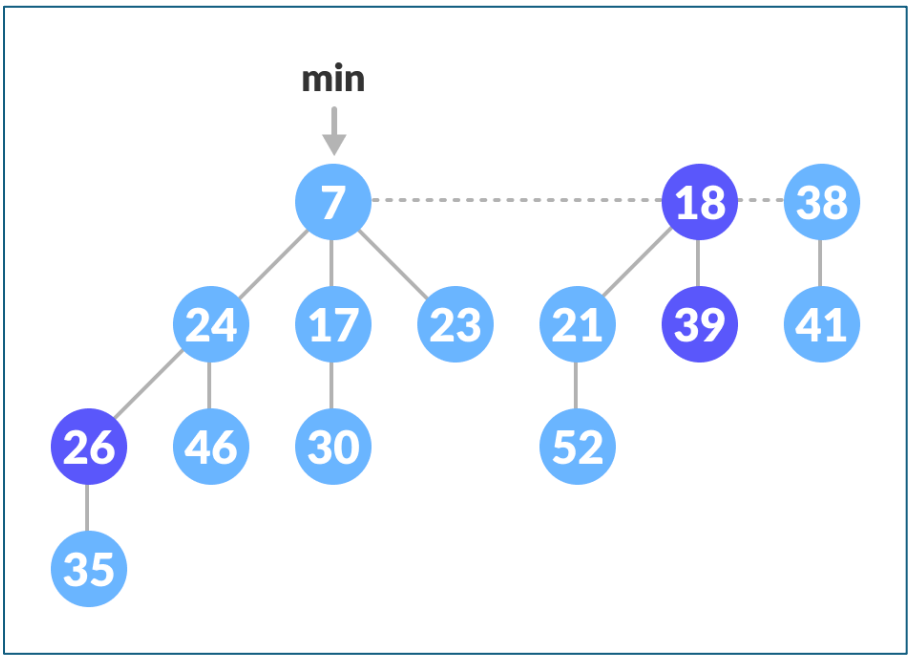
Unite those having the same degrees

10. Map the remaining root.



Map the remaining nodes

11. The final heap is.



Final fibonacci heap

Decreasing a Key

In decreasing a key operation, the value of a key is decreased to a lower value.

Following functions are used for decreasing the key.

Decrease-Key

1. Select the node to be decreased, x, and change its value to the new value k.
2. If the parent of x, y, is not null and the key of parent is greater than that of the k then call Cut(x) and Cascading-Cut(y) subsequently.
3. If the key of x is smaller than the key of min, then mark x as min.

Cut

1. Remove x from the current position and add it to the root list.
2. If x is marked, then mark it as false.

Cascading-Cut

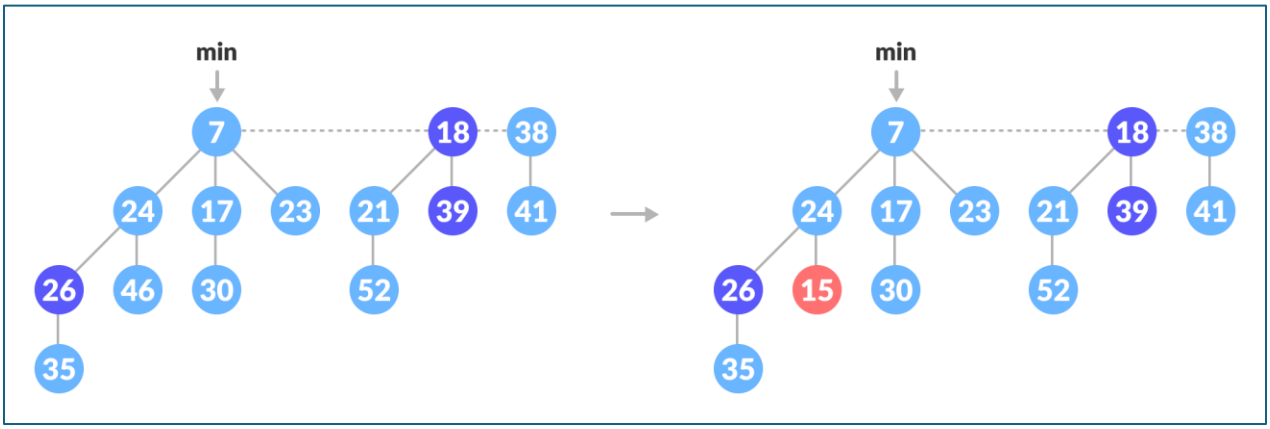
1. If the parent of y is not null then follow the following steps.
2. If y is unmarked, then mark y.
3. Else, call Cut(y) and Cascading-Cut(parent of y).

Decrease Key Example

The above operations can be understood in the examples below.

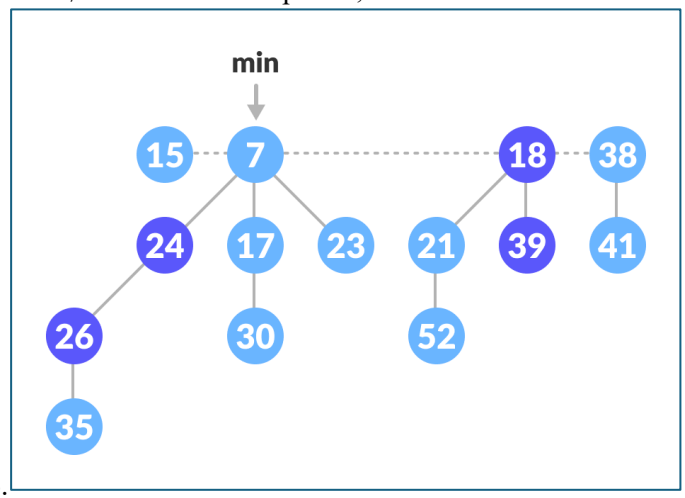
Example: Decreasing 46 to 15.

1. Decrease the value 46 to 15.



Decrease 46 to 15

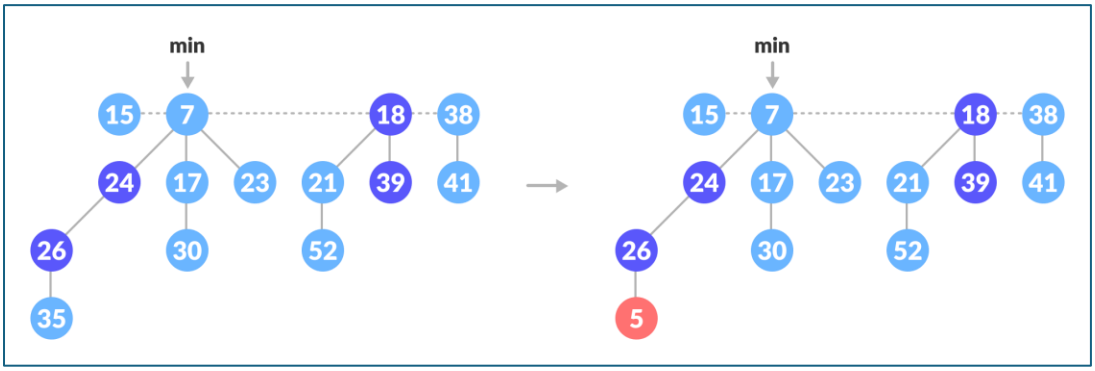
2. **Cut part:** Since 24 ≠ nill and 15 < its parent, cut it and add it to the root list. **Cascading-Cut part:** mark 24.



Add 15 to root list and mark 24

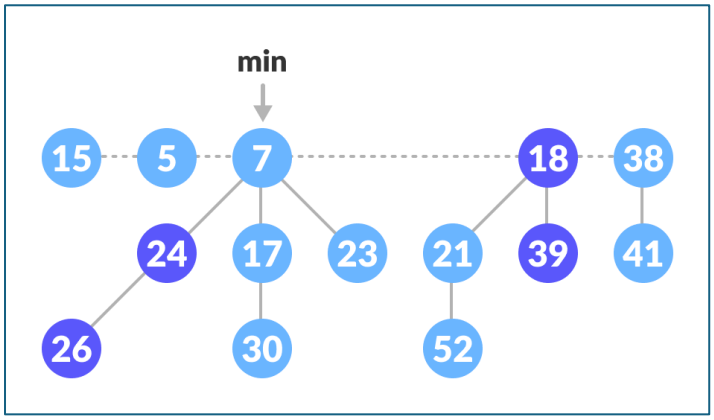
Example: Decreasing 35 to 5

1. Decrease the value 35 to 5.



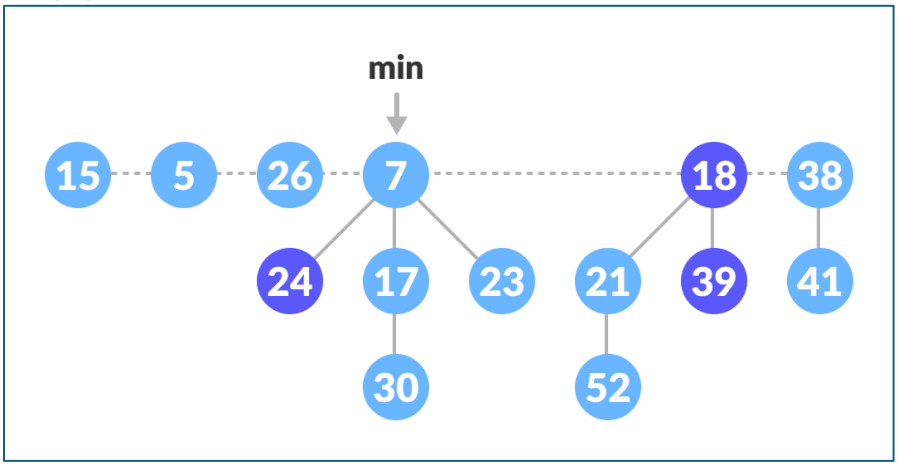
Decrease 35 to 5

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



Cut 5 and add it to root list

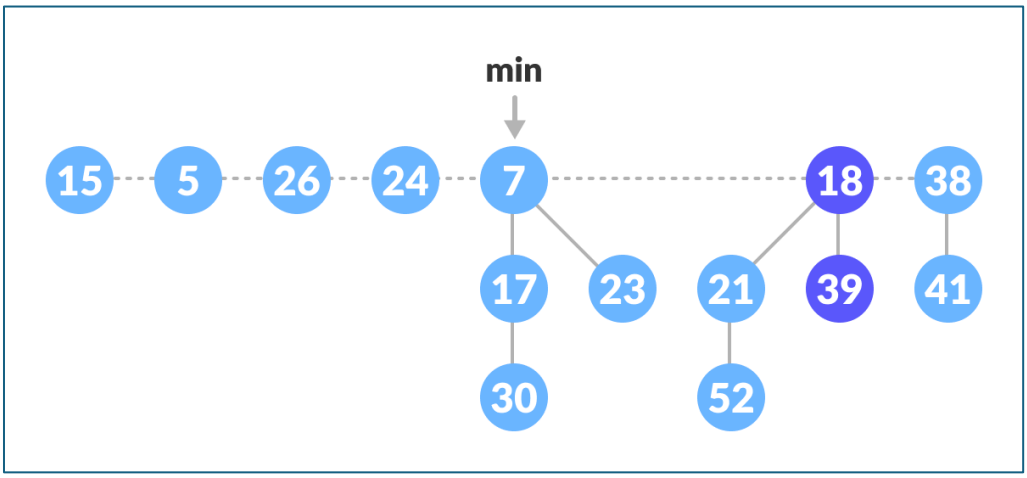
3. Cascading-Cut part: Since 26 is marked, the flow goes to Cut and Cascading-Cut.
Cut(26): Cut 26 and add it to the root list and mark it as false.



Cut 26 and add it to root list

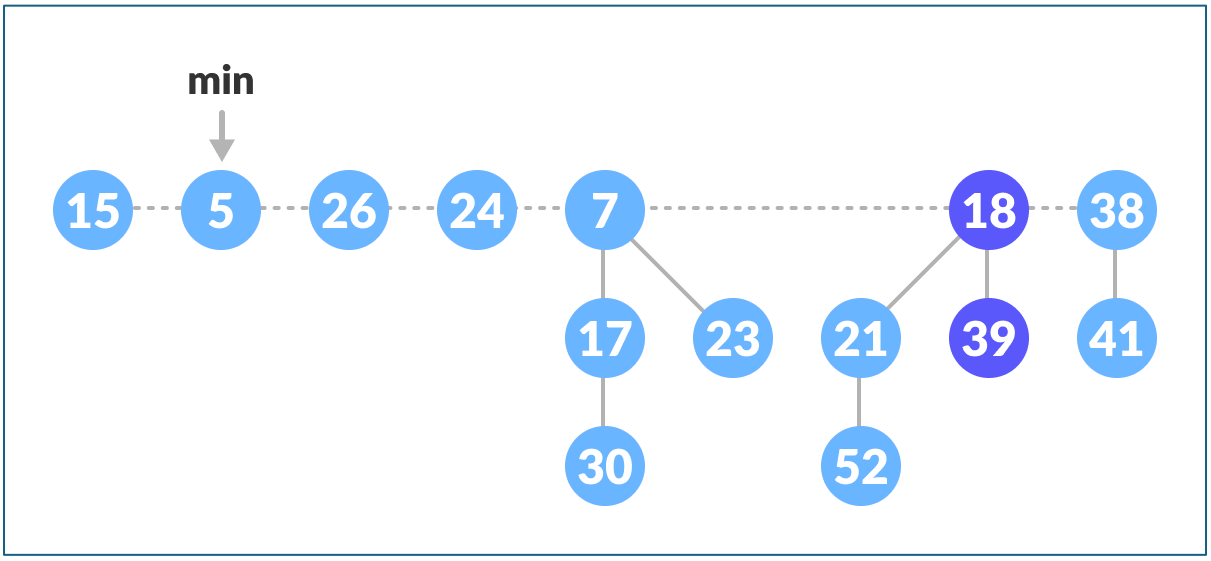
Cascading-Cut(24):

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

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



Mark 5 as min

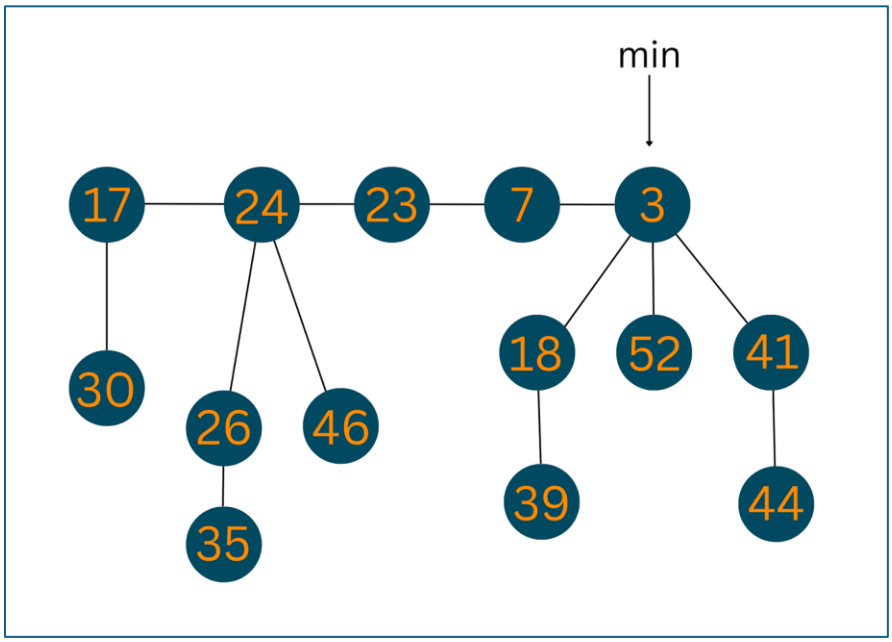
Deleting a Node

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.

Problem:

In the Fibonacci Heap, there is a pointer that keeps track of the minimum value in the heap. This smallest element is actually the root of the tree. All the tree roots in the heap are connected using the circular doubly linked list. This means that any node of the tree can be accessed by the pointer pointing toward the minimum value in the heap.



Operations on a Fibonacci Heap

A number of operations can be performed using a Fibonacci heap in comparatively less time as compared to other heaps. In this section, we shall be discussing the major operations of a Fibonacci Heap like insertion, deletion, merging, etc.

01) Insertion

To insert an element in a heap, follow the given algorithm:

1. Create a **new** node 'n'.

2. Find whether heap is empty or not

3. If(heap is empty)

    - make 'n' the only node

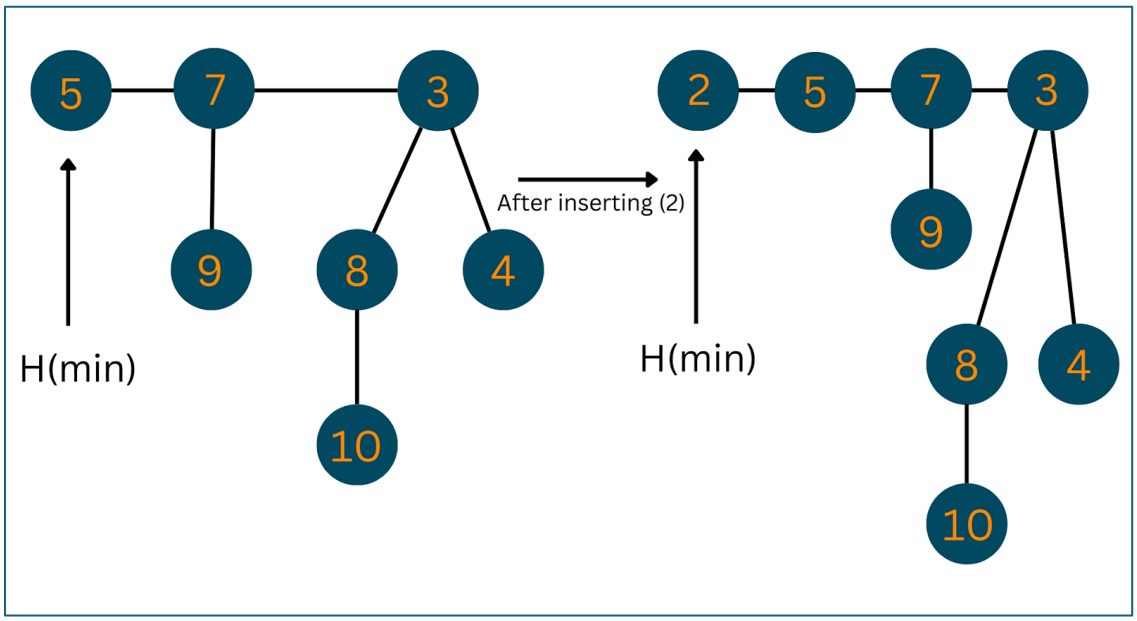
    - set the minimum pointer to 'n'

4. Else

    - insert 'n' into root list

    - update the value of minimum pointer

Take a look at the illustration below to understand what actually happened in the above steps:



02) Union

For the union of 2 Fibonacci Heaps, the below steps need to be followed:

1. Join the root list of both heaps(h1, h2) and form a single Fibonacci Heap(h).

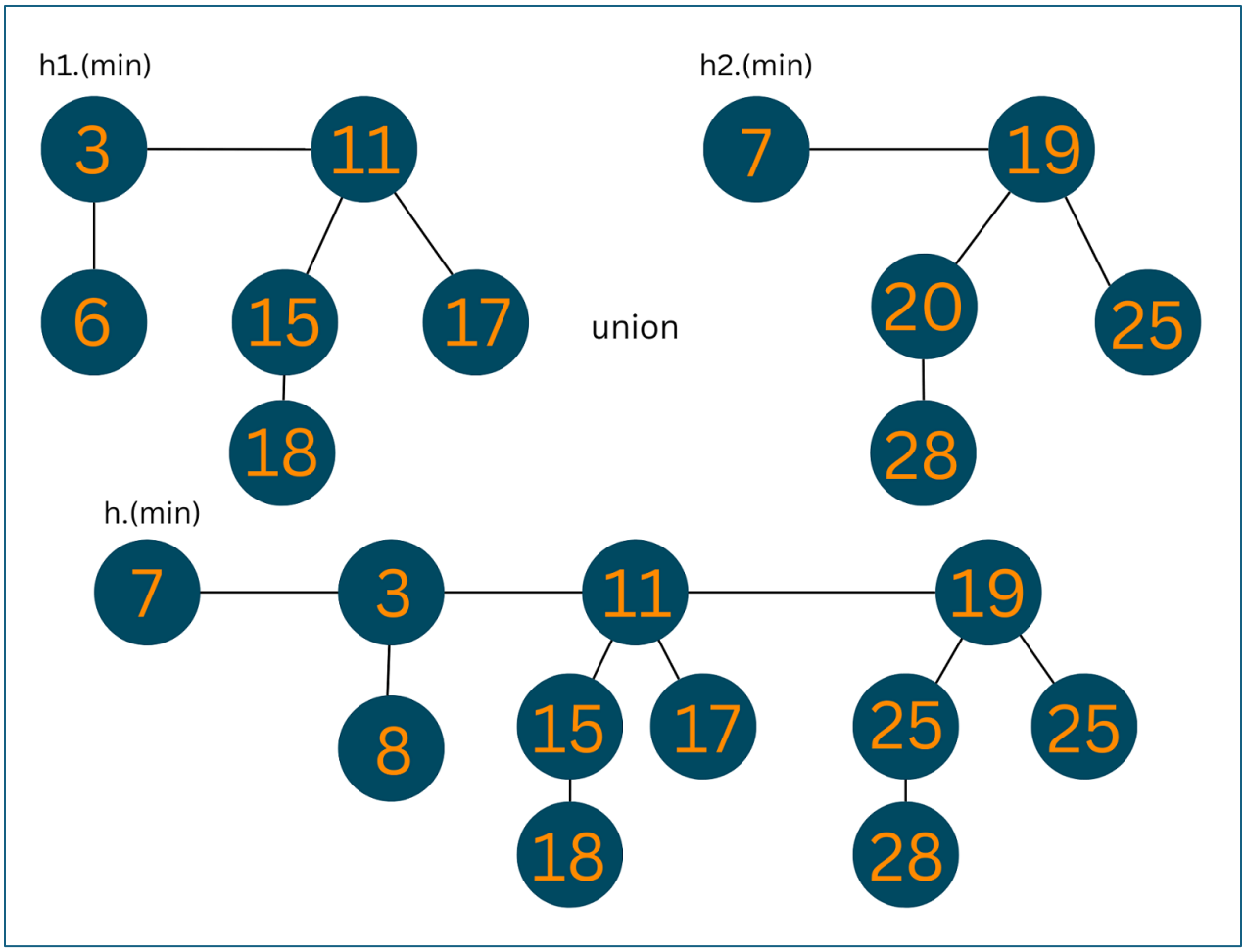
2. If h1(min) < h2(min)

    h(min) = h1(min)

3. Else

    h(min) = h2(min)

Take a look at the illustration below to understand what actually happened in the above steps:



03) Extracting Min

This operation is used to find out the minimum element in the whole tree.

1. Delete the initial min node.
2. Set pointer to the next min node.
3. Add all the trees of the deleted node in the root list.
4. Create an array of degree pointers of the deleted node's size.
5. Set the current node as the degree pointer.
6. Move forward towards the next node.
7. If degrees are different

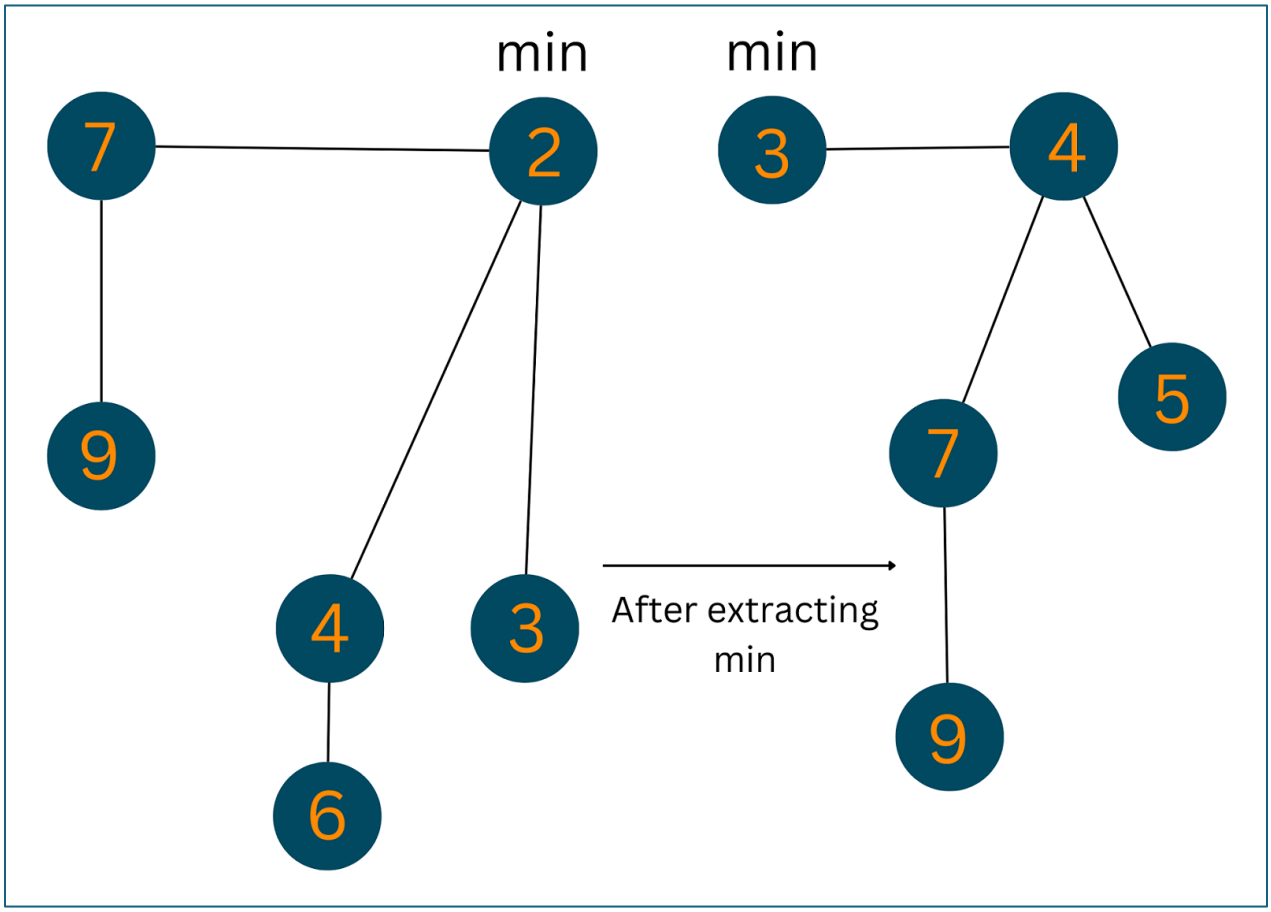
    -set degree pointer to the next node.

8. Else

    -join the Fibonacci trees by union operation.

9. Repeat steps 4 and 5 until the heap is completed.

Take a look at the illustration below to understand what actually happened in the above steps:



04) Decrease Key

This operation is used to decrease the value of a particular node to a lower value. Follow the steps to achieve the result:

1. Decrease the value of the node ‘x’ to the **new** value.

2. Case 1) If min-heap property is not violated:

    -Update min pointer

3. Case 2) If min-heap property is violated and parent of ‘x’ is not marked,

    -Cut off the link between ‘x’ and its parent.

    -Mark x's parent.

    -Add tree rooted at ‘x’ to the root list and update the min pointer

3. Case 3) If min-heap property is violated and parent of ‘x’ is marked:

    -Cut off the link between ‘x’ and its parent.

    -Add ‘x’ to the root list, updating min pointer

-Cut off the link between p[x] and p[p[x]].

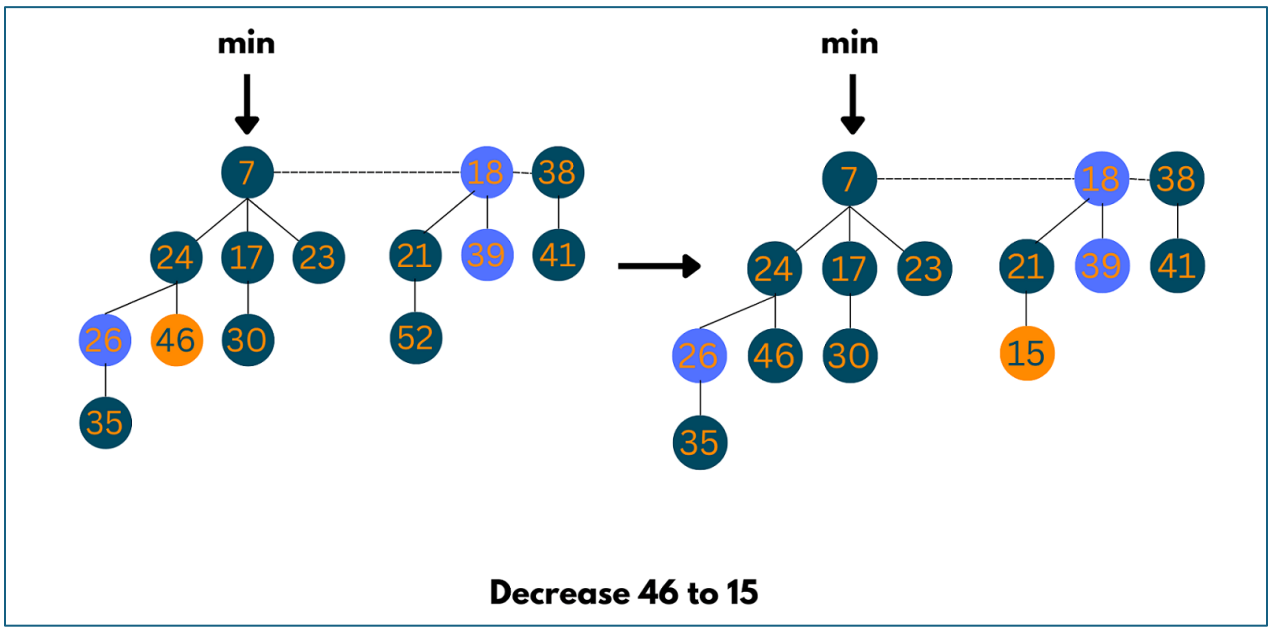
-Add p[x] to the root list, updating min pointer

-If p[p[x]] is unmarked:

-mark it.

-Else:

-cut off p[p[x]] and repeat steps taking p[p[x]] as ‘x’.



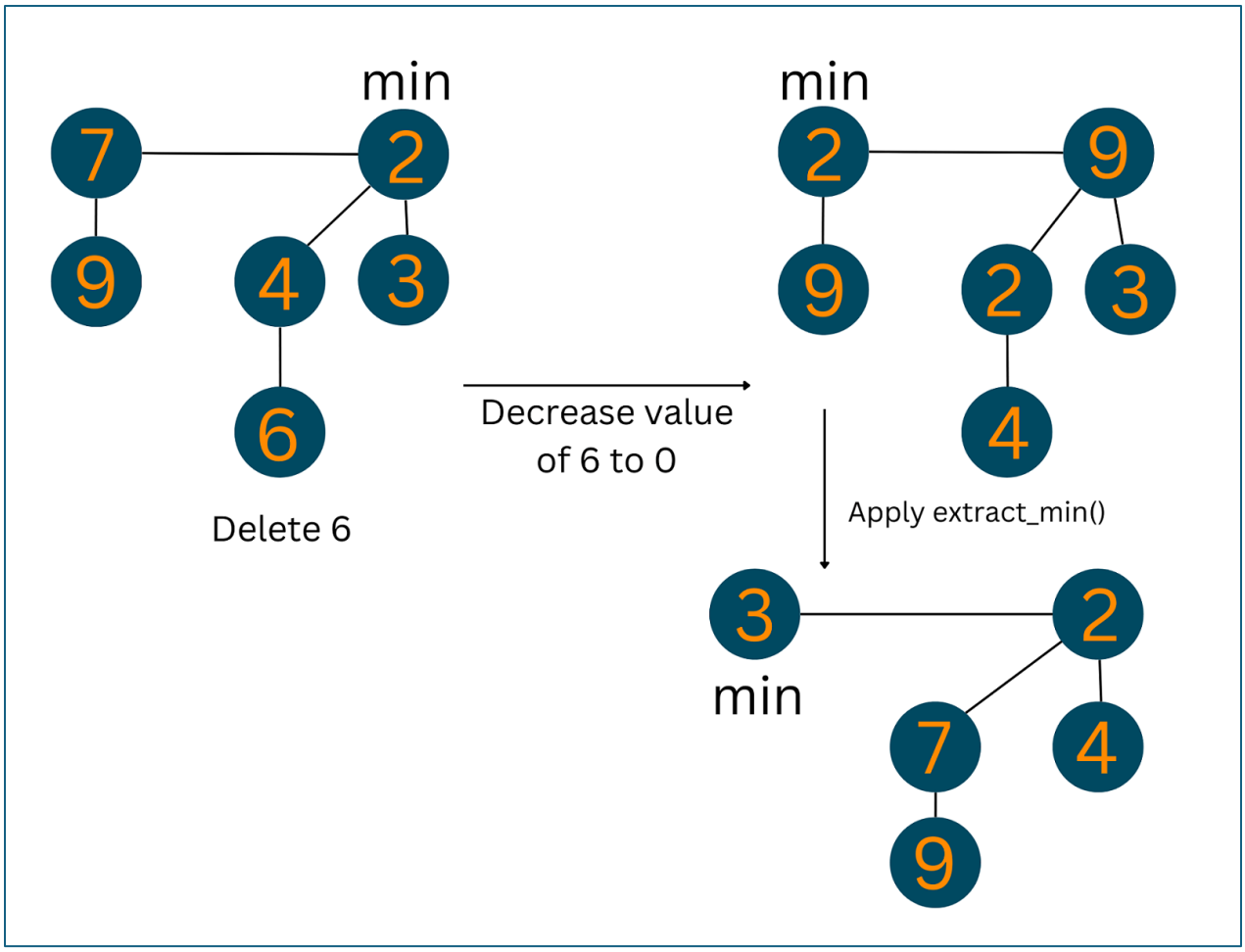
05) Deletion

1. Decrease the value of the node to be deleted ‘x’ to a minimum by using the previous Decrease_key() function.

2. Heapify the heap using min-heap property, bringing ‘x’ to the root list.

3. Apply Extract_min() algorithm to the Fibonacci heap.

Look at the below image to understand the output of the above operation:



Time Complexities of Fibonacci Heap Operations

The following table shows the time complexity for each operation on this heap:

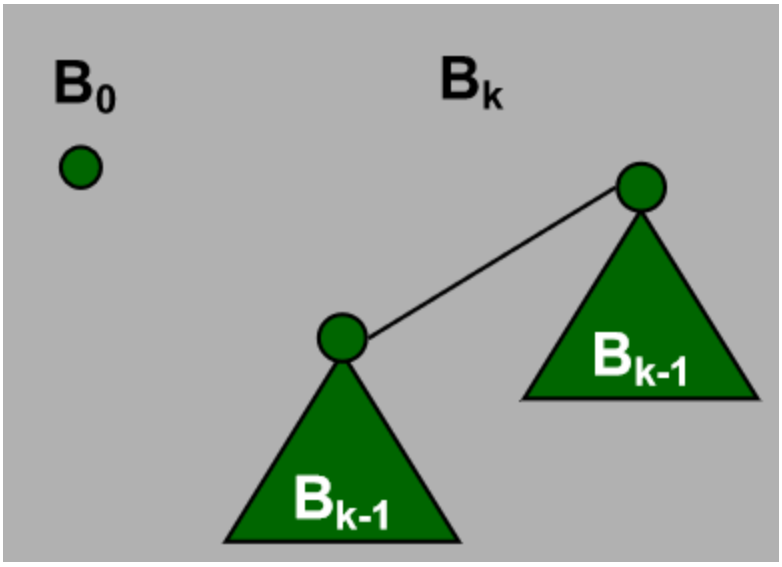
OPERATION	TIME COMPLEXITY
Insertion	O(1)
Find Min	O(1)
Union	O(1)
Extract Min	O(log n)
Decrease Key	O(1)
Delete Node	O(log n)

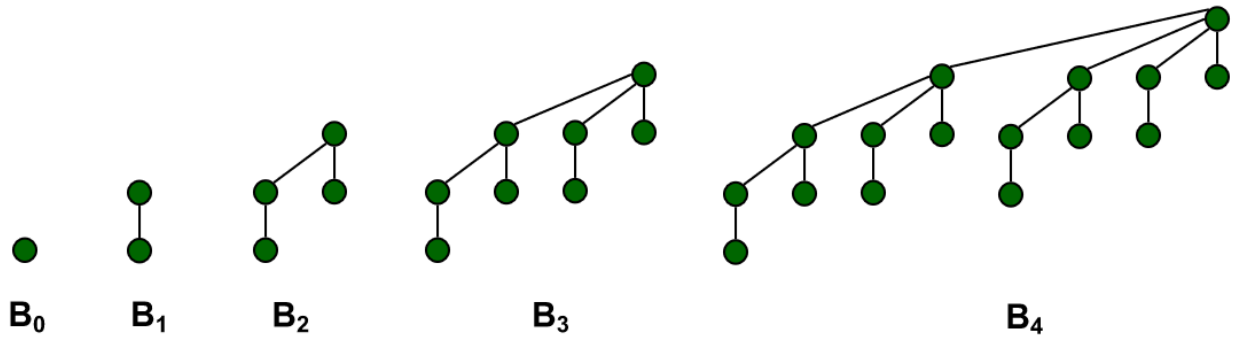
problem

Binomial Heap

Binomial Tree

- Recursive definition:





- Rank of a node

x: the number of children of x

- Rank of a tree: the rank of the root of the tree
- Property of B_k
  - Number of nodes $= 2^k$
  - Height $= k$
  - Degree of root $= k$
  - Deleting root yields binomial trees $B_{k-1}, \dots, B_0$
    - Proof: Induction on k

- B_k has

$\binom{k}{i}$ nodes at depth

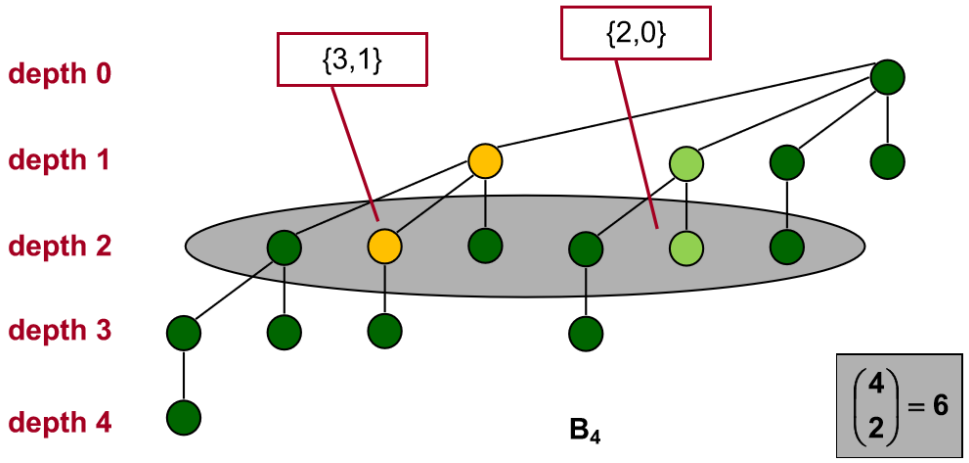
i

  - Every combination of

i elements in
$\{k-1, k-2, \ldots, 0\}$ correspond to a node in depth
i, based on the degrees of nodes from the root to it.

  - For example,

$$k = 4, i = 2$$



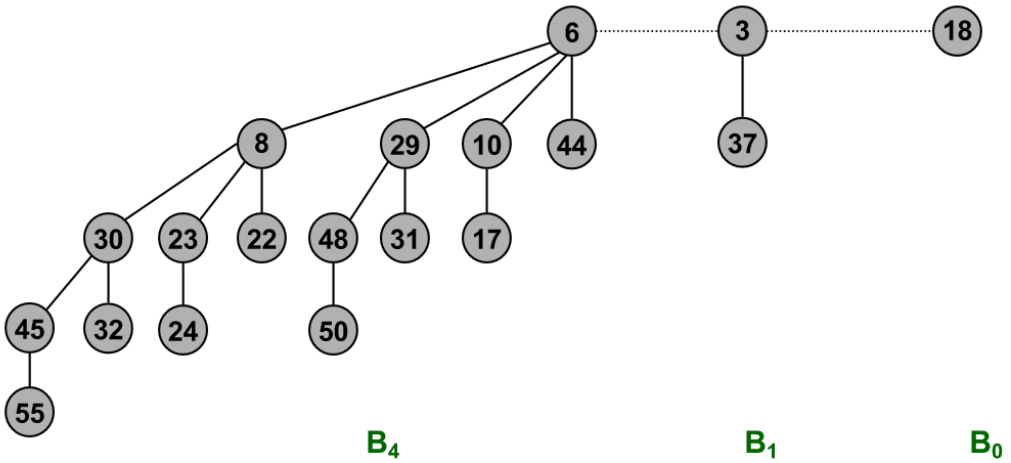
Definition

Sequence of binomial trees that satisfy binomial heap property

- Each tree is min-heap ordered
- 0or

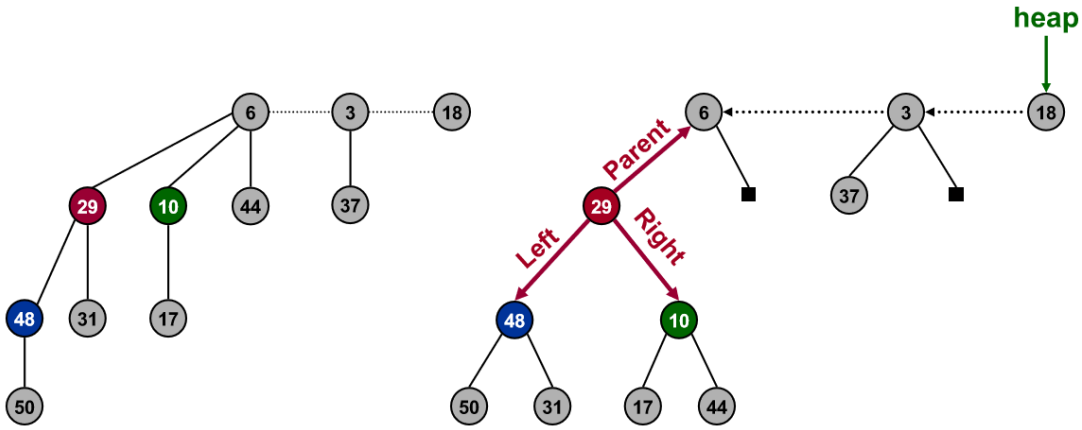
1binomial tree of each rank

- If there are two trees of the same rank, then merge them



Implementation

- Represent trees using left-child, right sibling pointers
    - Three links per node (parent, left, right)
- Roots of trees connected with singly linked list
    - Degrees of trees strictly decreasing from left to right



Property

- Min key contained in roots of

$$B_0, B_1, \ldots, B_k$$

- Contains binomial tree

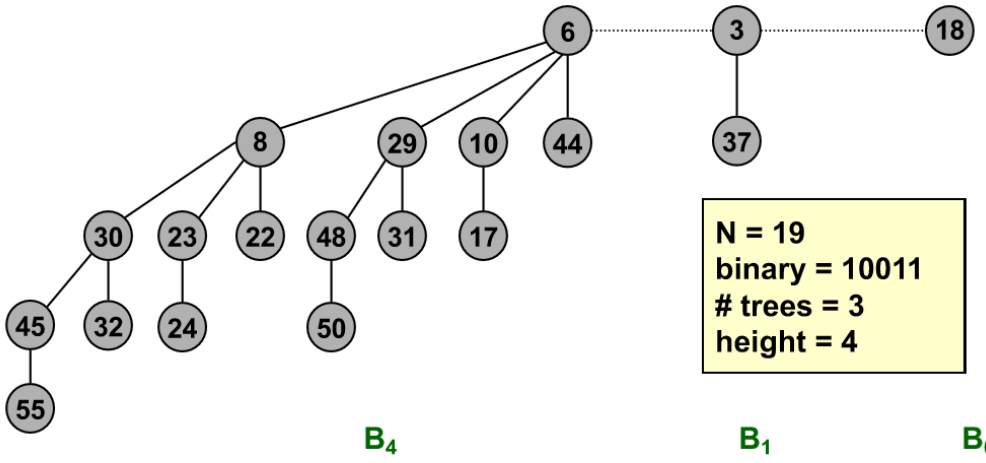
$B_i \Leftrightarrow b_i = 1$where
$b_n \ldots b_2 b_1 b_0$is binary representation of
N, since
$B_i$has
2^inodes

    - At most

$\lfloor \log_2 N \rfloor + 1$binomial trees

    - Height

$$\leq \lfloor \log_2 N \rfloor$$



Operation

Union: create heap
Hthat is union of heaps
$H'$and
H''

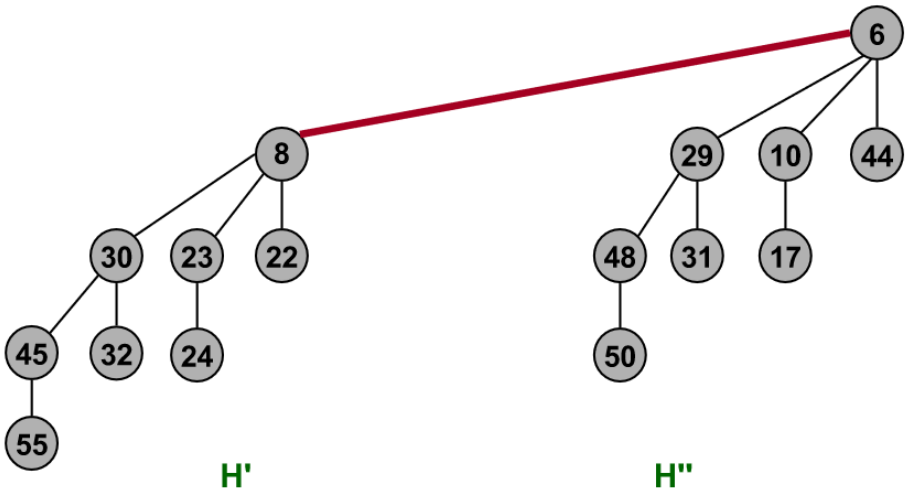
- **Mergeableheaps**
- Easy if

$H'$and
$H''$are each rank kbinomial trees

- Connect roots of

$H'$and
H''

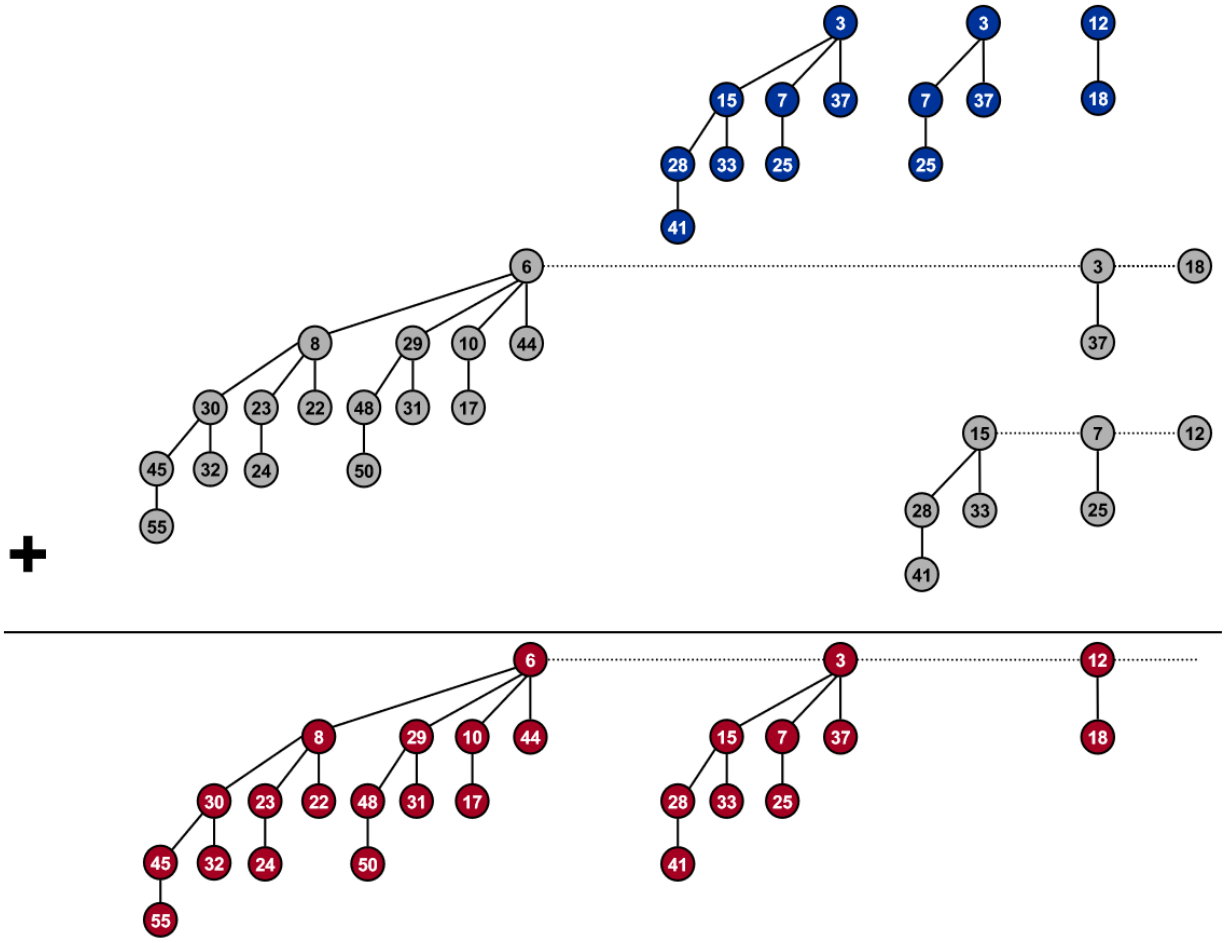
- Choose smaller key to be root of
H



- Analogous to binary addition

19 + 7 = 26

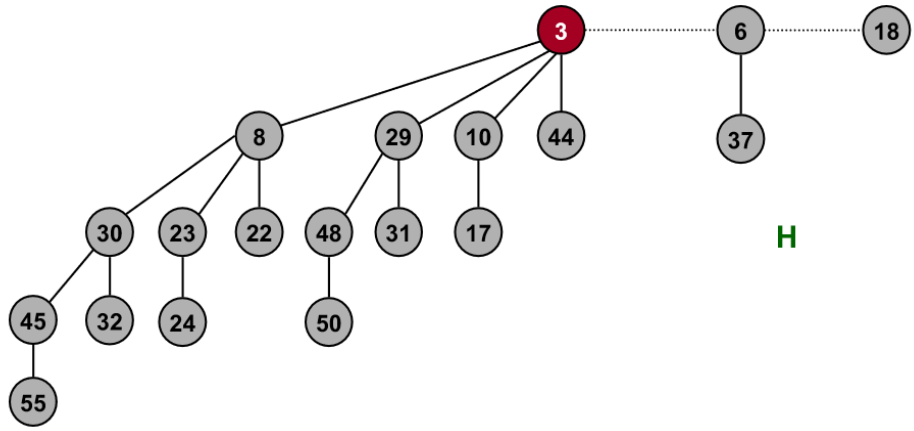
		1	1	1	
	1	0	0	1	1
+	0	0	1	1	1
	1	1	0	1	0



- Running time: $O(\log N)$

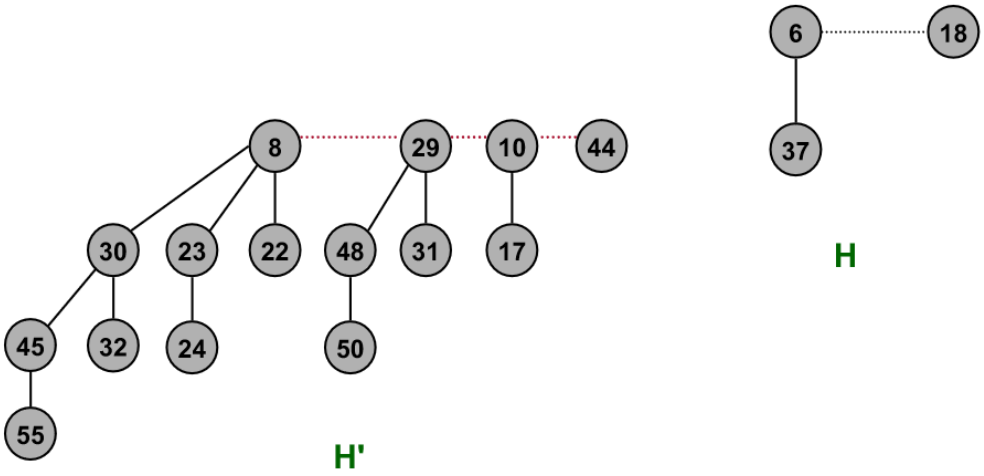
DeleteMin: delete node with minimum key in binomial heap H

- Find root $x$with min key in root list of H, and delete



- H' ←broken binomial trees

- $H \leftarrow Union(H', H)$

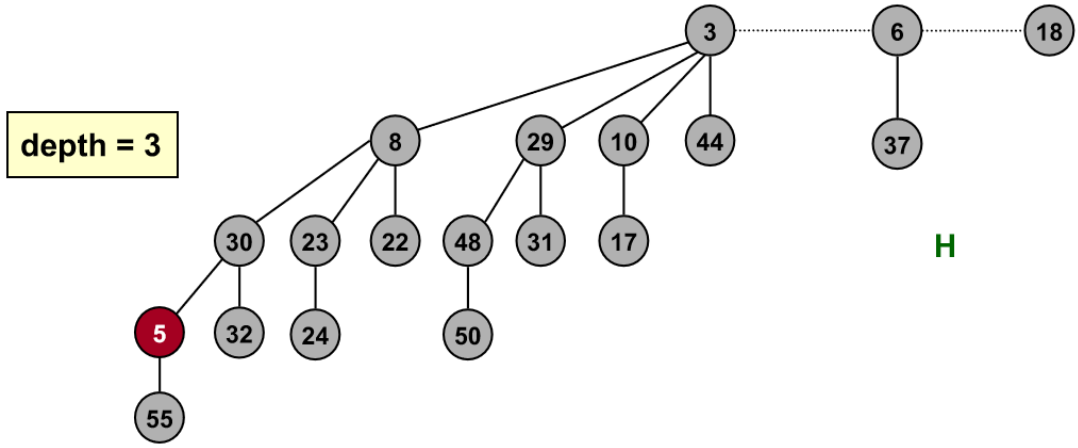


- Running time:

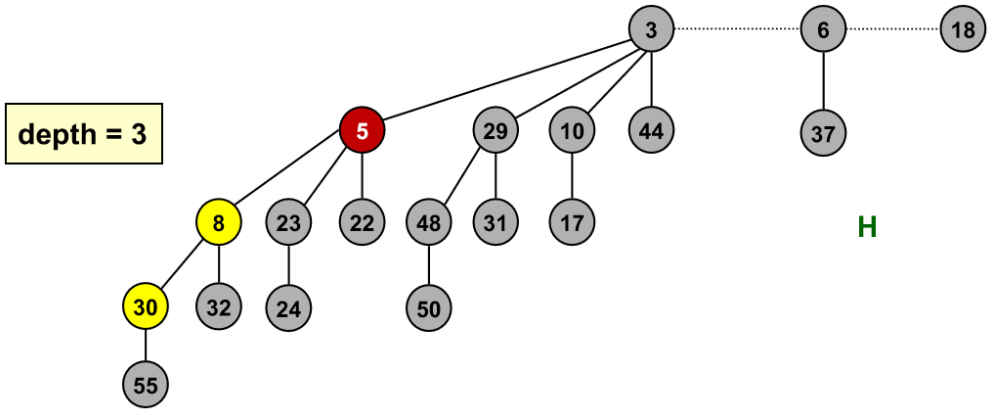
$$O(\log N)$$

DecreaseKey: decrease key of node x in binomial heap H

- Suppose x is in binomial tree B_k



- Bubble node x up the tree if x is too small



- Running time:

$$O(\log N)$$

  - Proportional to depth of node

$$x \leq \lfloor \log_2 N \rfloor$$

- $Delete$: delete node

$x$in binomial heap

$$H$$

  - $DecreaseKey(x, -\infty)$
  - $DeleteMin$
  - Running time

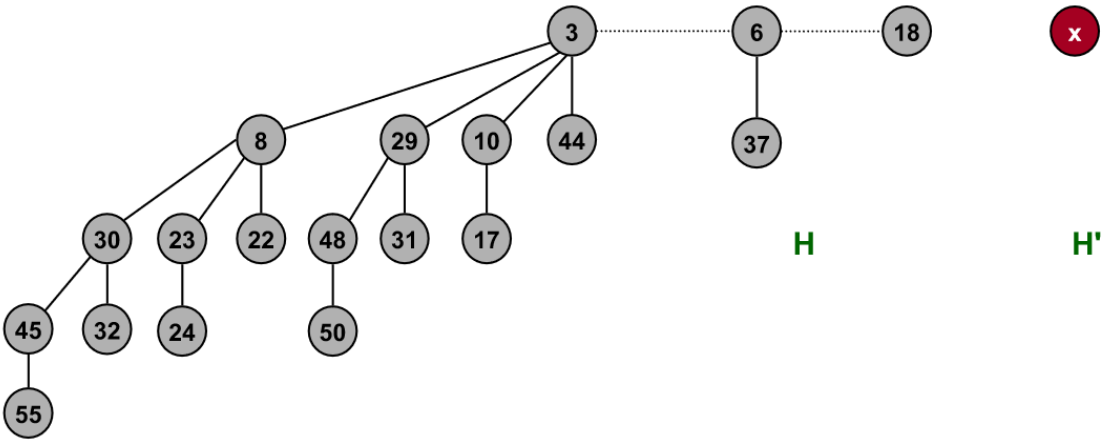
$$O(\log N)$$

- $Insert$: insert a new node

$x$into binomial heap

$$H$$

  - $H' \leftarrow MakeHeap(x)$
  - $H \leftarrow Union(H', H)$



- Running time:

$$O(\log N)$$

- Sequence of

$Inserts$:

- Insert a new node

$x$into binomial heap

H

- If

$N = \ldots\ldots 0_2$, then only

1steps

- If

$N = \ldots\ldots 01_2$, then only

2steps

- If

$N = \ldots\ldots 011_2$, then only

3steps

- ..
- If

$N = 11111111_2$, then

$\log_2 N$steps

- Inserting

1item can take

$\Omega(\log N)$time

- But, inserting sequence of

Nitems takes

$O(N)$time

- $(N/2)(1) + (N/4)(2) + (N/8)(3) + \ldots \leq 2N$
- Amortized analysis

Problem

Fibonacci Heap (*Fredman and Tarjan, 1986*)

History

- Ingenious data structure and analysis
- Original motivation: improve *Dijkstra*'s shortest path algorithm from

$O(m\log\ n)$to

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

Basic idea

- Similar to binomial heaps, but less rigid structure
- Binomial heap: **eagerly** consolidate trees after each

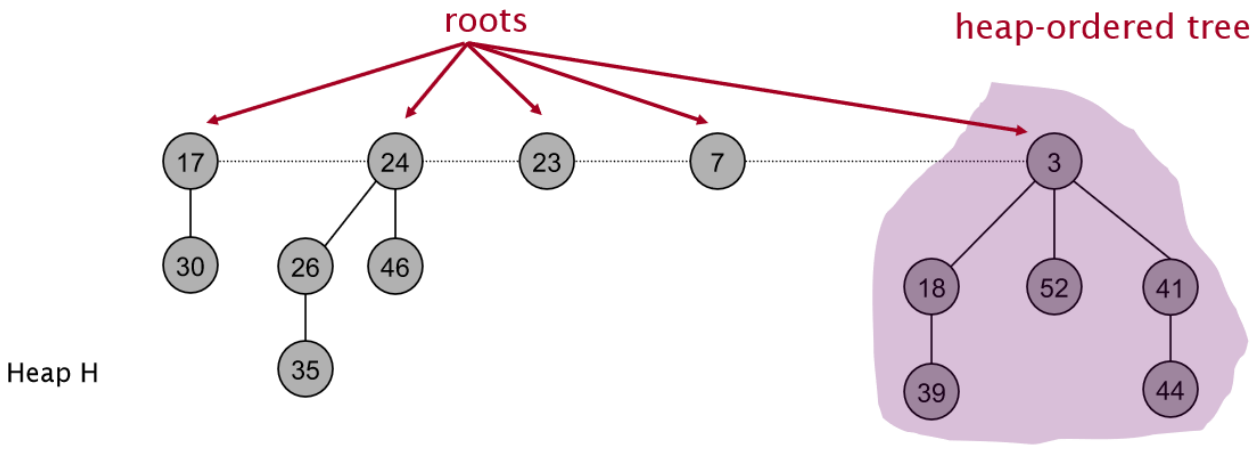
$Insert$

- Fibonacci heap: **lazily** defer consolidation until next

$DeleteMin$

Definition

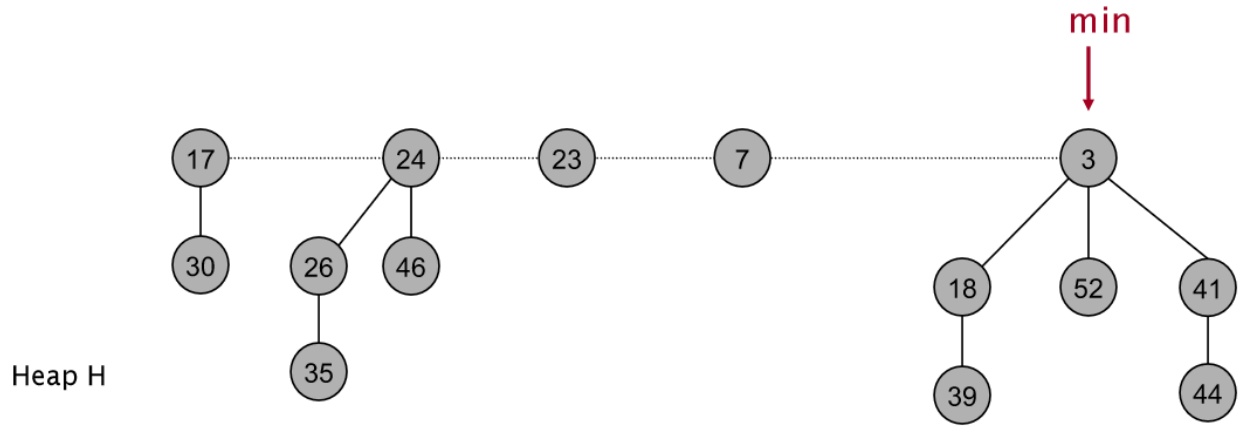
- Set of **heap-ordered** (each parent smaller than its children) trees



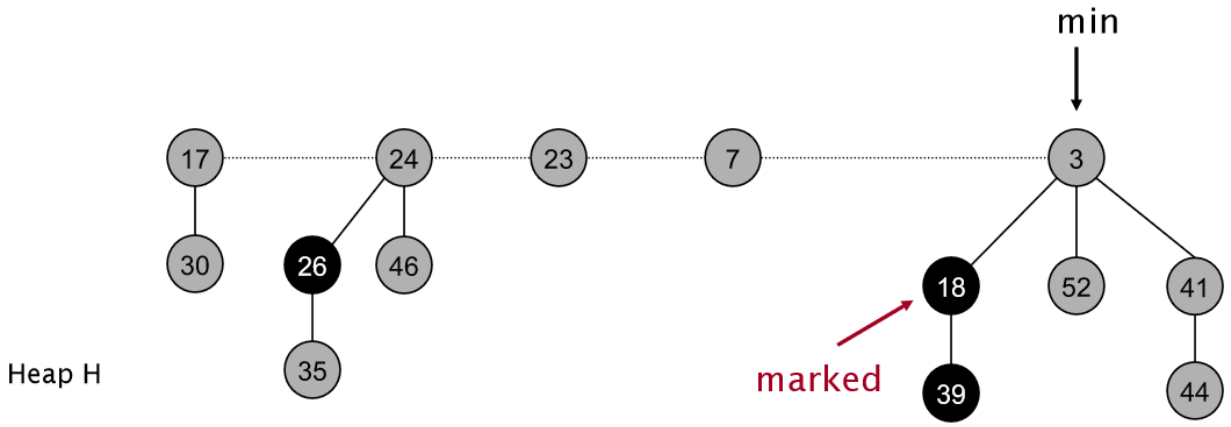
- Maintain pointer to minimum element

→ $FindMin$takes

$O(1)$time



- Set of marked nodes (has lost one child, if it loses another child, it will be cut)



Notation

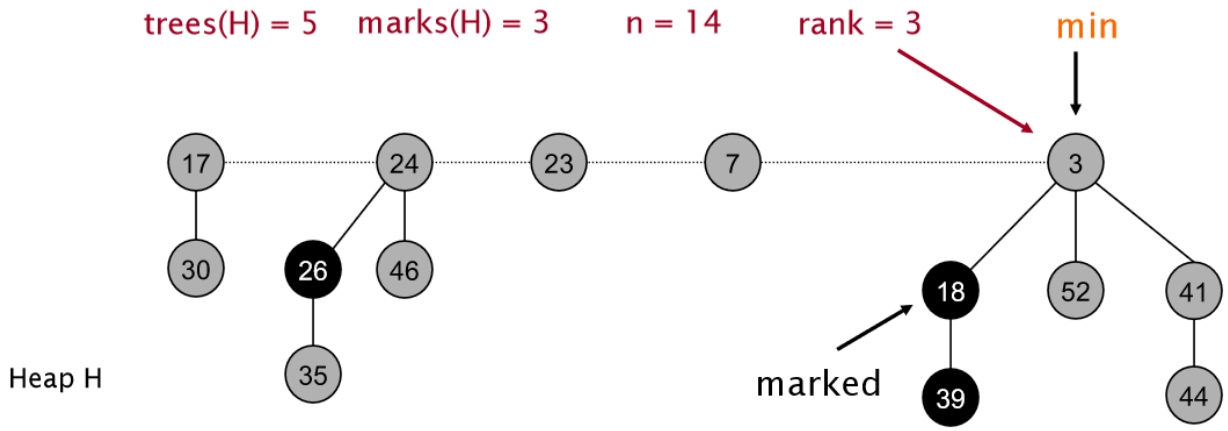
- n: number of nodes in heap
- $rank(x)$: number of children of node

  x
- $rank(H)$: max rank of any node in heap

  H
- $trees(H)$: number of trees in heap

  H
- $marks(H)$= number of marked nodes in heap

  H



Potential function

- Potential of heap

H:

$$\phi(H) = trees(H) + 2marks(H)$$

- Amortized time: actual time

+change of potential

- Initially sum of potentials is

0, potentials remain positive all the time

- Thus, real total time

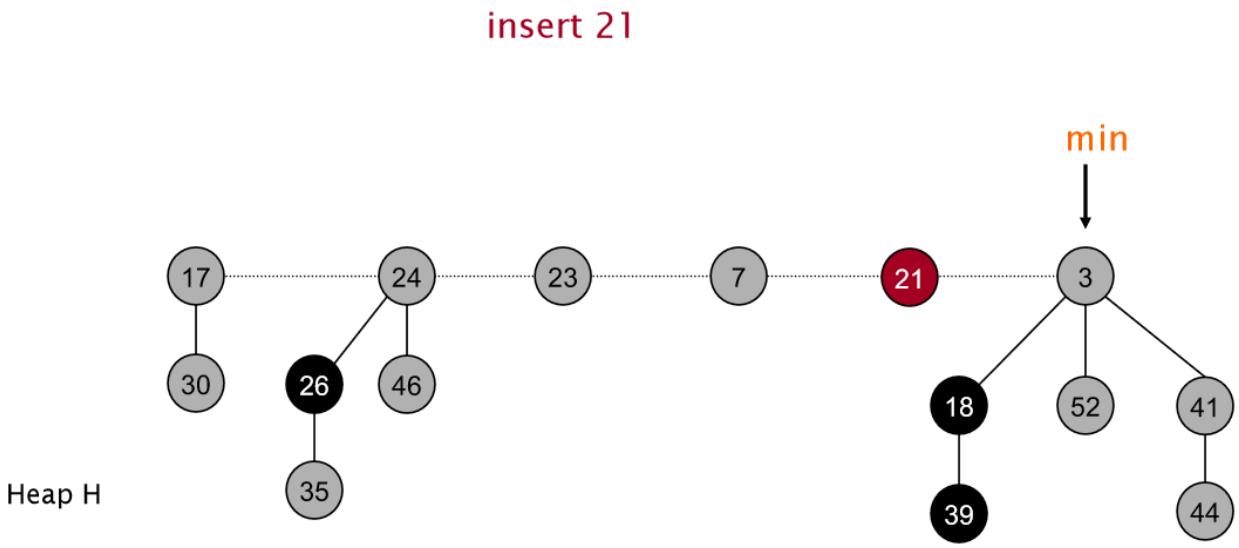
=total amortized time

−final potential

- So total time is at most the total amortized time

Operation

- *Insert*
  - Create a new singleton tree
  - Add to root list; update min pointer (if necessary)



  - Actual cost:

$$O(1)$$

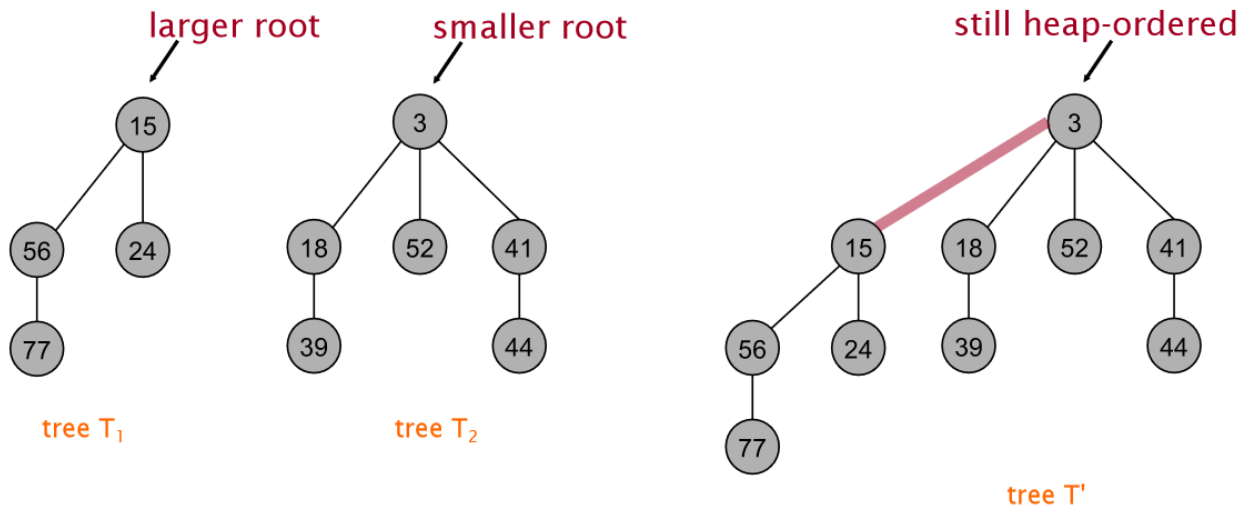
  - Change in potential:

$$+1$$

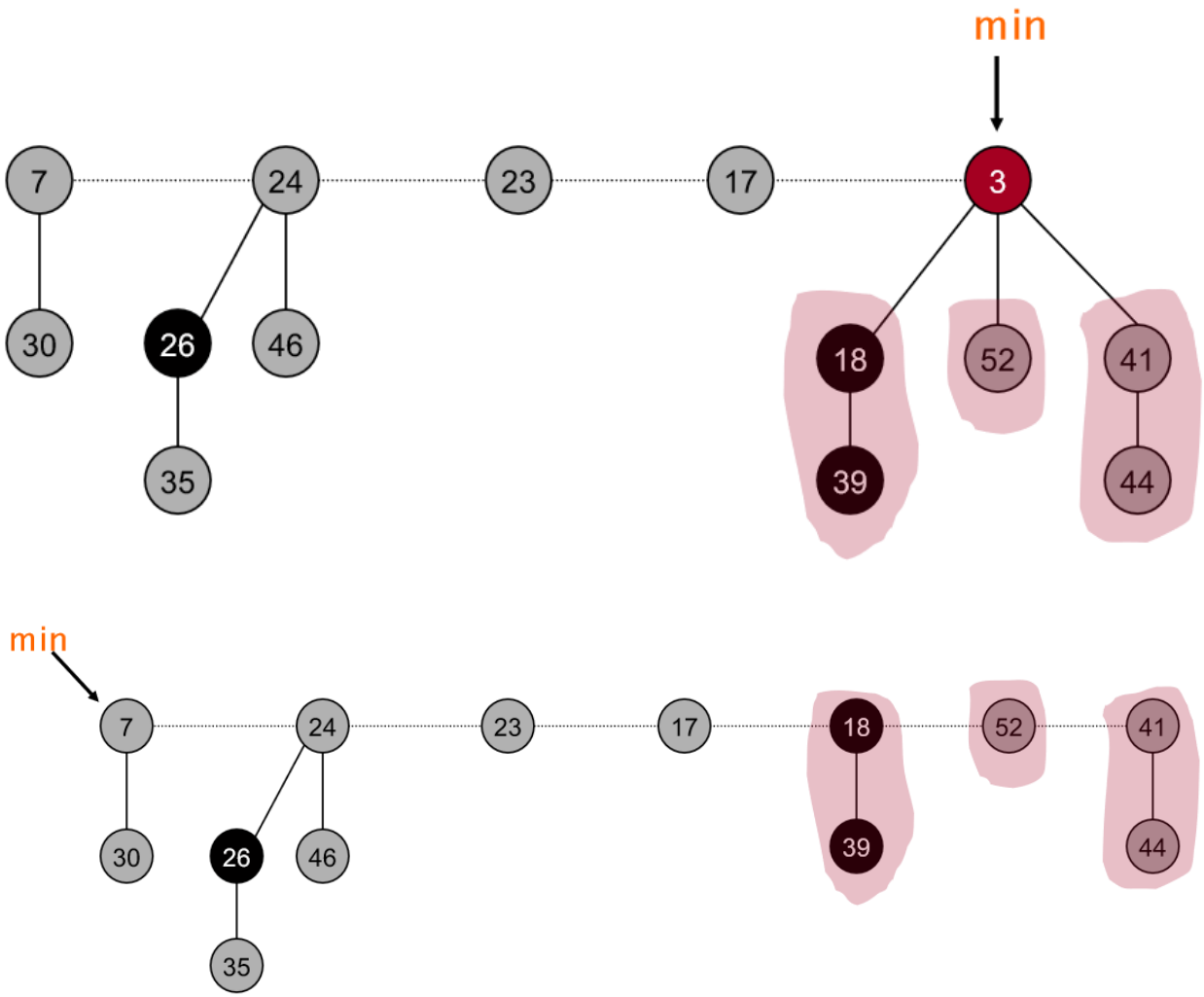
  - Amortized cost:

$$O(1)$$

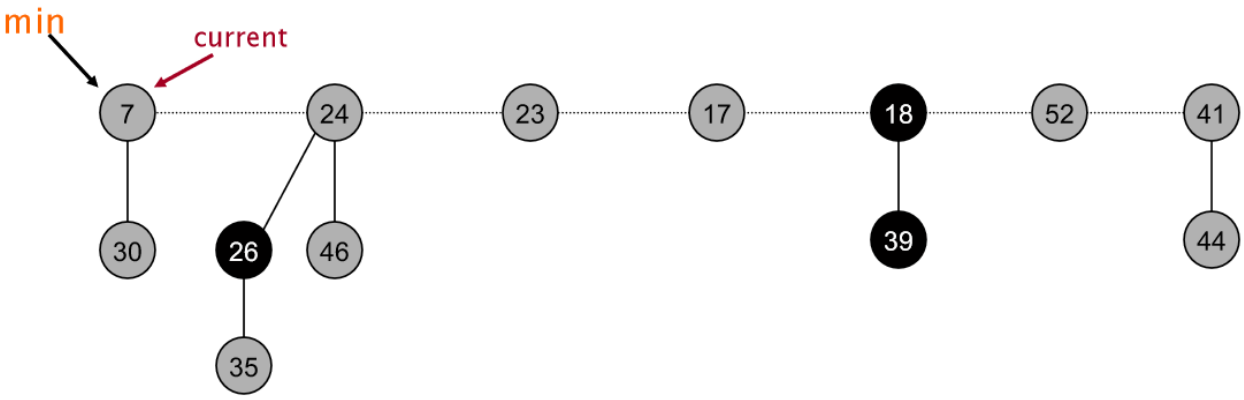
- *DeleteMin*
  - *Linking*: Make larger root be a child of smaller root



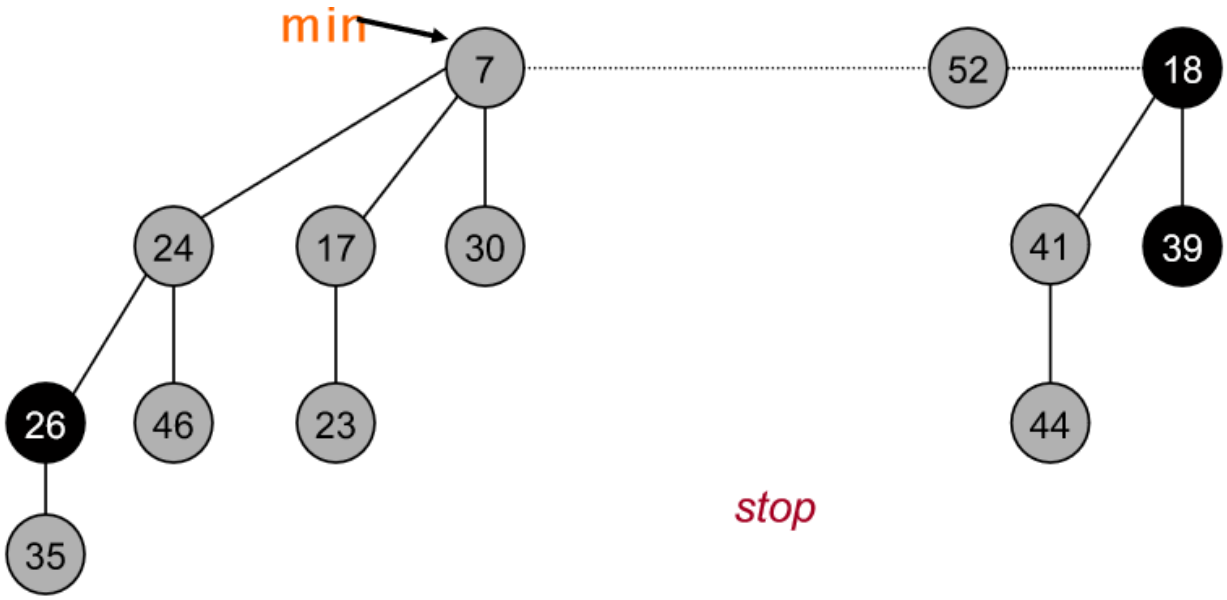
- Delete min; meld its children into root list; update min



- Consolidate trees so that no two roots have same rank



- Same as in binomial heap



- Actual cost:

$$O(rank(H)) + O(trees(H))$$

  - $O(rank(H))$to meld min's children into root list
  - $O(rank(H)) + O(trees(H))$to update min
  - $O(rank(H)) + O(trees(H))$to consolidate trees

- Change in potential:

$$O(rank(H)) - trees(H)$$

  - $trees(H') \leq rank(H) + 1$since no two trees have same rank
  - $\Delta\phi(H) = trees(H') - trees(H) \leq rank(H) + 1 - trees(H)$

- Amortized cost:

$$O(rank(H))$$

- Is amortized cost of

$O(rank(H))$good?

- Yes, if only

$Insert$and

$DeleteMin$operations

- In this case, all trees are binomial trees (we only

$Link$trees of equal rank)

- This implies

$$rank(H) \leq \log_2 n$$

- We'll implement

$DecreaseKey$so that

$$rank(H) = O(\log n)$$

- $DecreaseKey$
  - Intuition for deceasing the key of node

$$x$$

- If heap-order is not violated, just decrease the key of

$$x$$

- Otherwise, cut tree rooted at

$x$and meld into root list

- To keep trees flat: as soon as a node has its second child cut, cut it off and meld into root list and unmark it

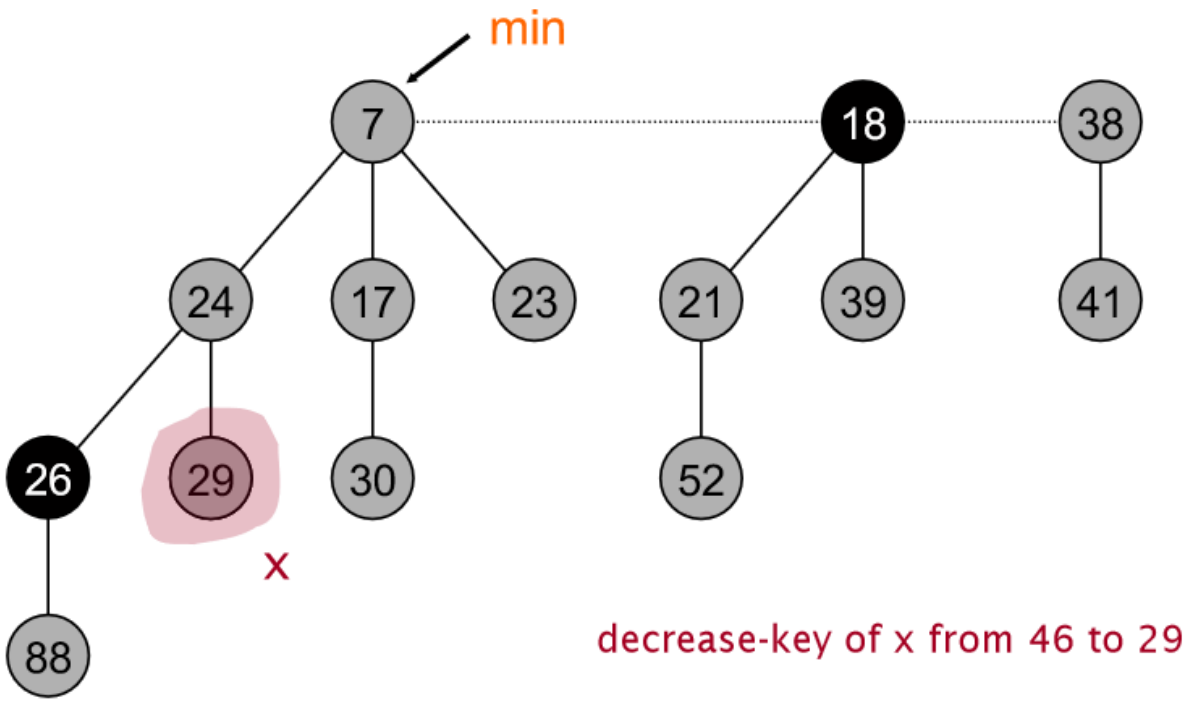
  - Case

1: heap order not violated

- Decrease key of

$$x$$

- Change heap min pointer (if necessary)

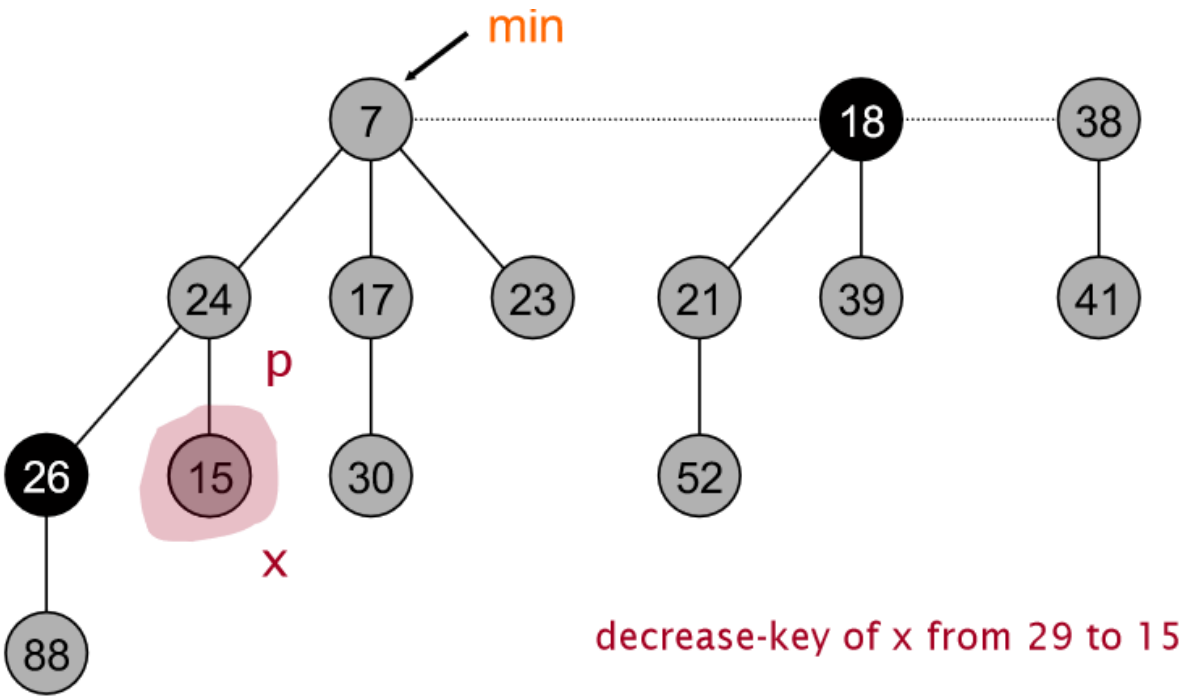


- Case

$2a$: heap order violated

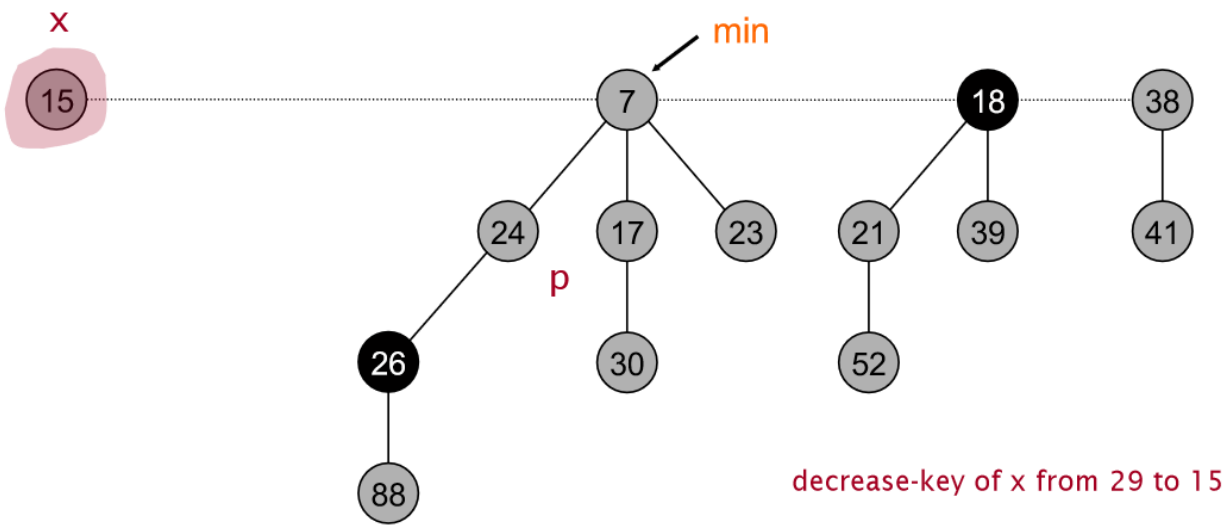
- Decrease key of

x



- Cut tree rooted at

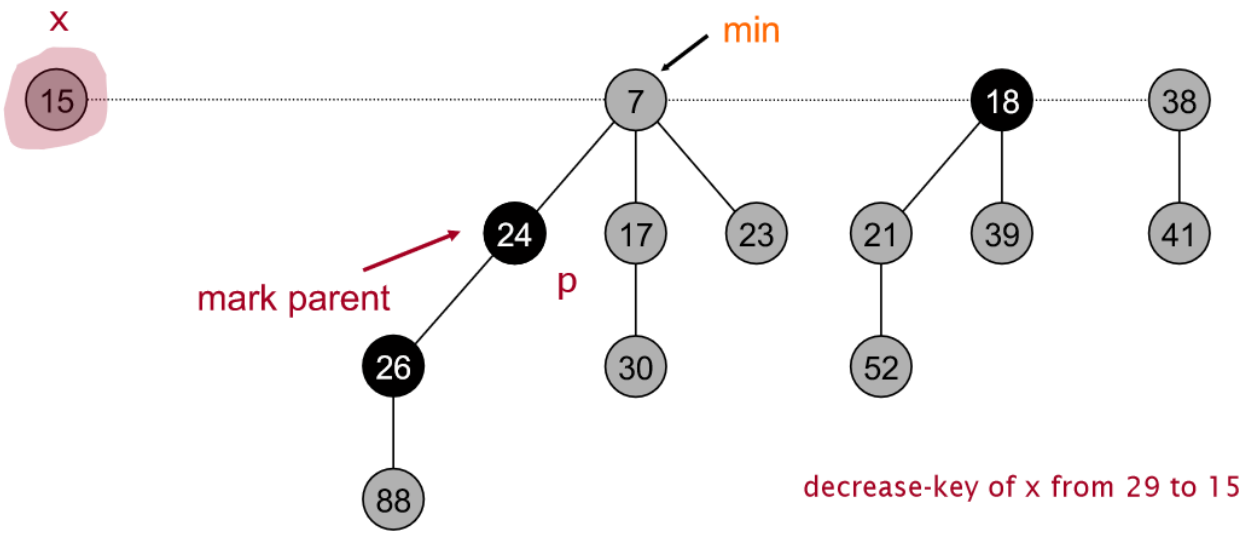
x, meld into root list, and unmark



- If parent

$p$of

$x$is unmarked (hasn't yet lost a child), mark it

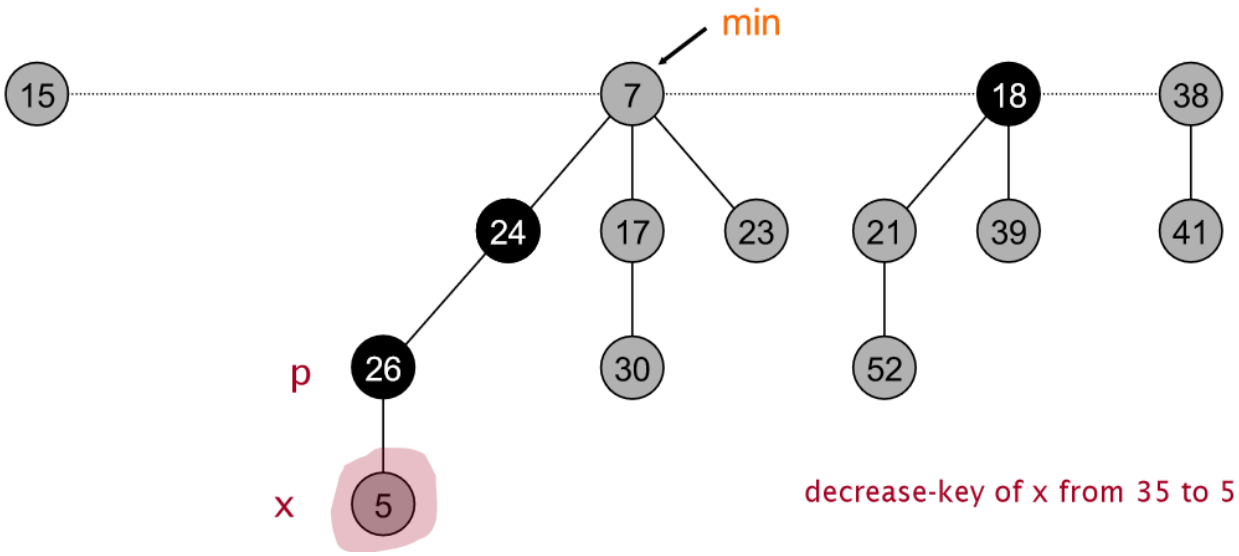


- Case

$2b$

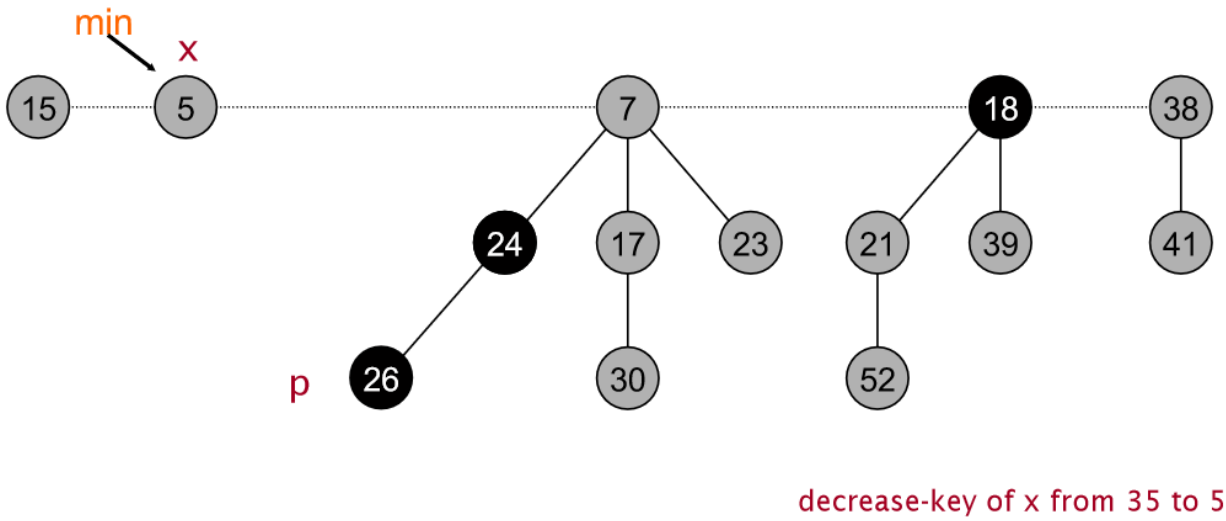
- Decrease key of

x



- Cut tree rooted at

x, meld into root list, and unmark

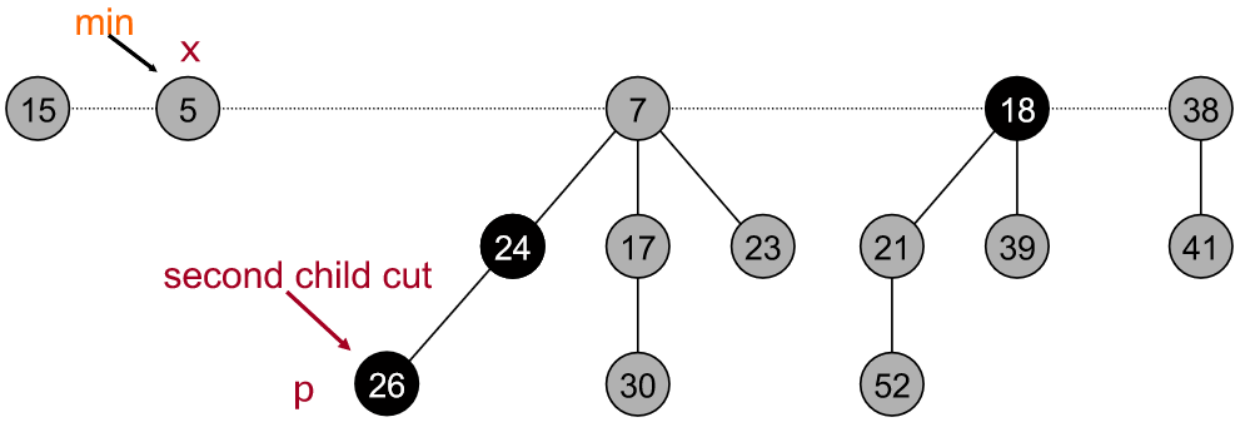


- If parent

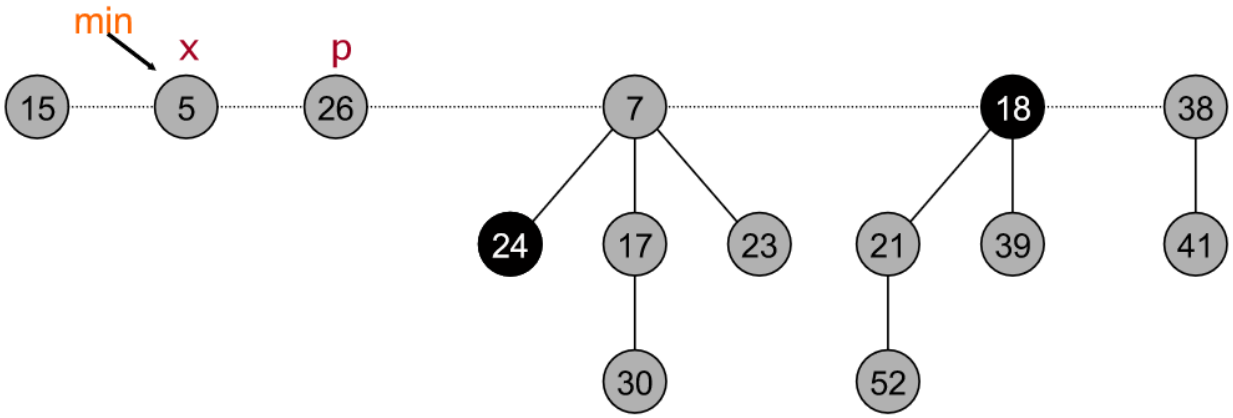
$p$of

$x$is unmarked (hasn't yet lost a child), mark it
Otherwise, cut

p, meld into root list, and unmark

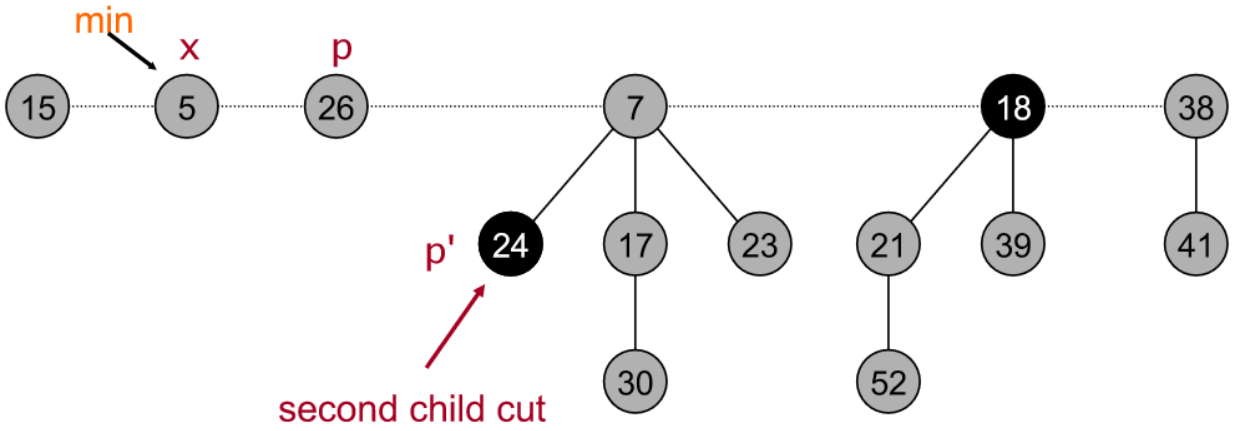


decrease-key of x from 35 to 5

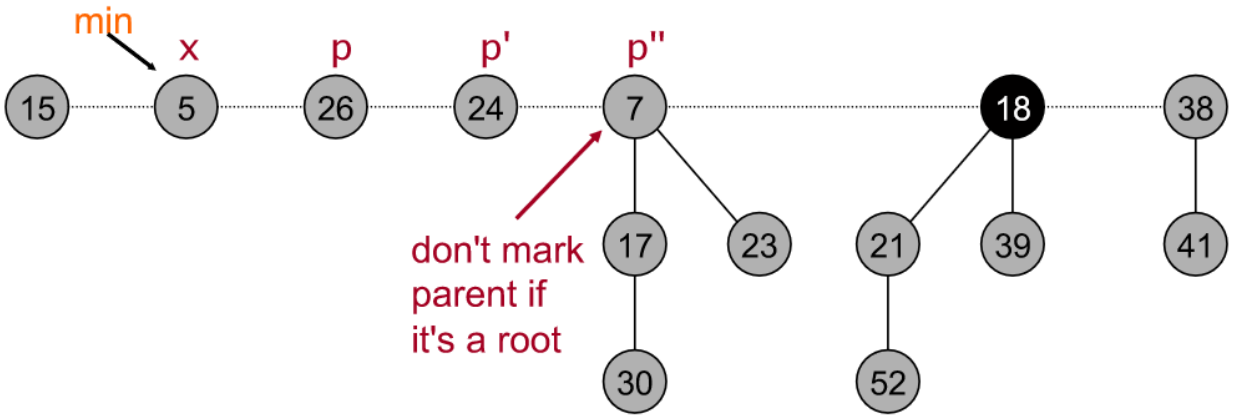


decrease-key of x from 35 to 5

- And do so recursively for all ancestors that lose a second child

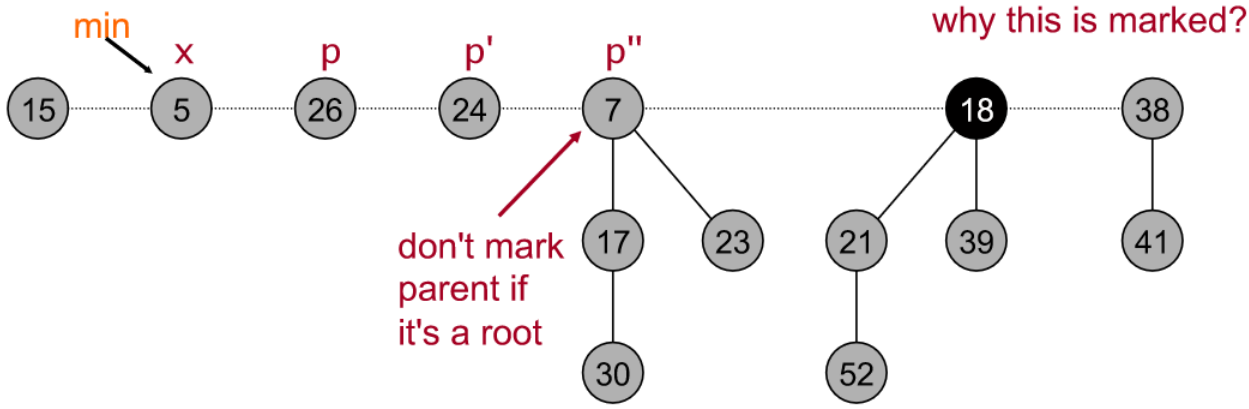


decrease-key of x from 35 to 5

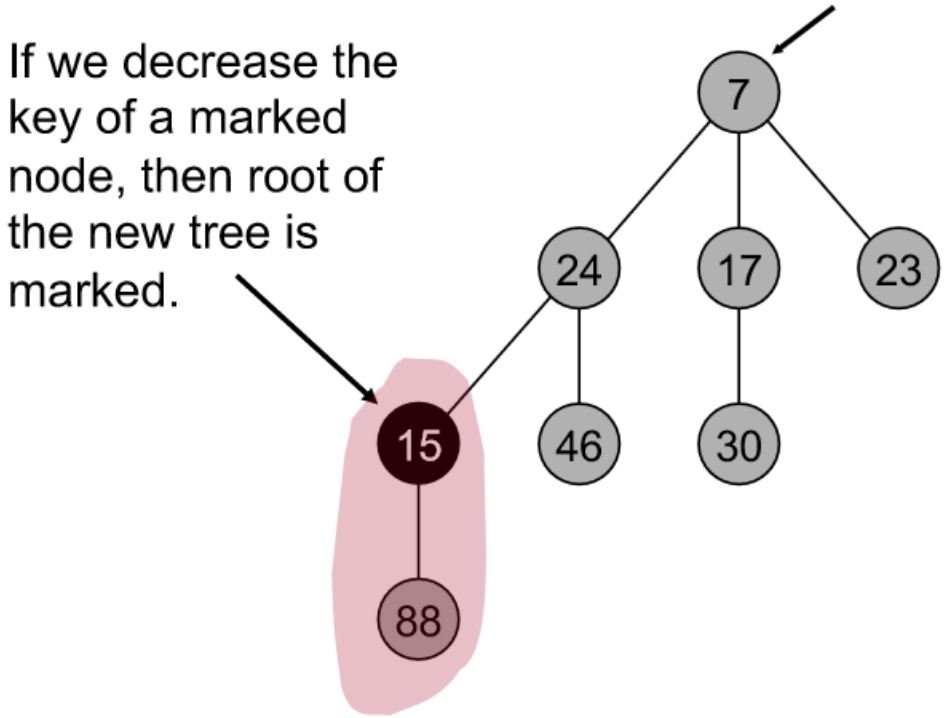


decrease-key of x from 35 to 5

- Question: why we have a marked root node



decrease-key of x from 35 to 5



- Actual cost:

$$O(c)$$

  - $O(1)$time for changing the key
  - $O(1)$time for each of

$c$cuts, plus melding into root list

- Change in potential:

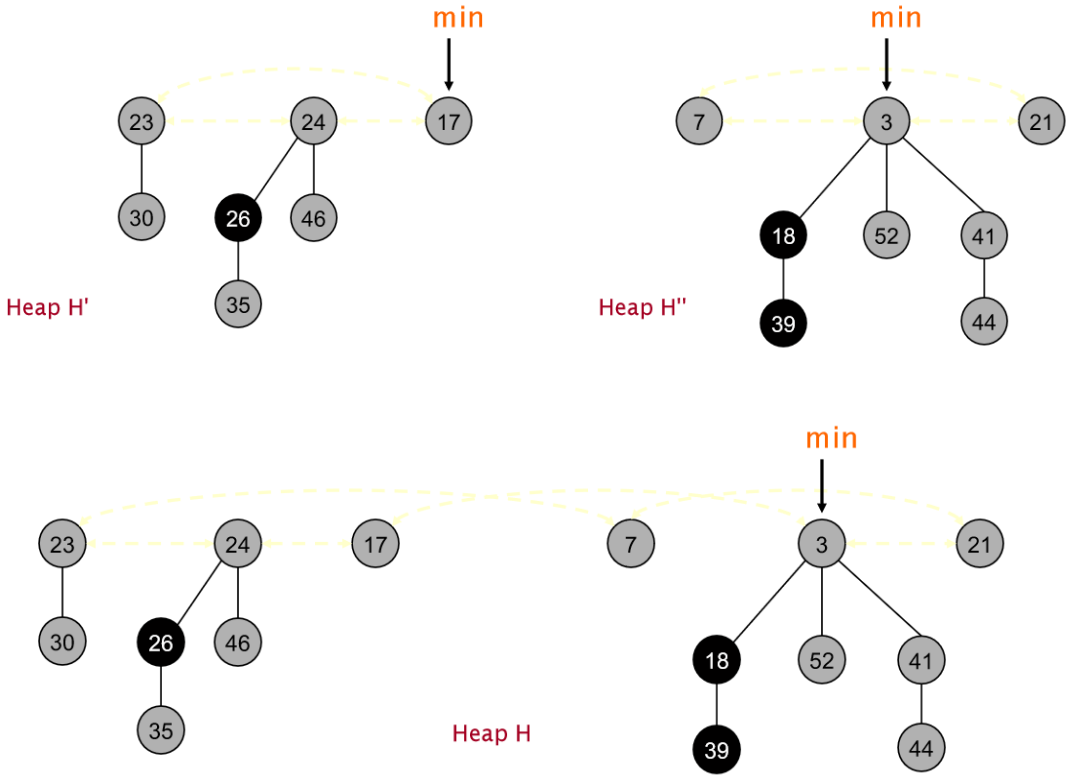
$$O(1) - c$$

  - $trees(H') = trees(H) + c$
  - $marks(H') \leq marks(H) - c + 2$
  - $\Delta\phi \leq c + 2(-c + 2) = 4 - c$

- Amortized cost:

$$O(1)$$

- *Union:* combine two Fibonacci heaps
  - Representation: root lists are circular, doubly linked lists



- Actual cost:

$$O(1)$$

- Change in potential:

$$0$$

- Amortized cost:

$$O(1)$$

- *DeleteMin*: delete node

$$x$$

- $DecreaseKey$of

$x$to

$$-\infty$$

- $DeleteMin$element in heap
- Amortized cost:

$$O(rank(H))$$

  - $O(1)$amortized for

$$DecreaseKey$$

  - $O(rank(H))$amortized for

$$DeleteMin$$

Bounding the Rank

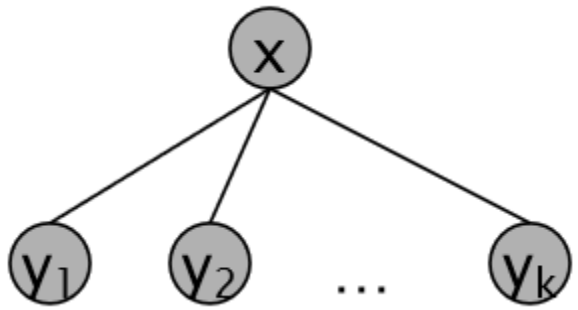
Lemma: Fix a point in time. Let

$x$be a node, and let

$y_1, \ldots, y_k$denote its children in the order in which they were linked to

x. Then

$$\textbf{rank}(y_i) \geq \begin{cases} \mathbf{0}, & \textbf{if } i = 1 \\ \boldsymbol{i-2}, & \textbf{if } i \geq 1 \end{cases}$$



- When

$y_i$was linked into

x,

$x$had at least

$i-1$children

$$y_1, \ldots, y_{i-1}$$

- Since only trees of equal rank are linked, at that time

$$rank(y_i) = rank(x_i) \geq i - 1$$

- Since then,

$y_i$has lost at most one child or

$y_i$would have been cut

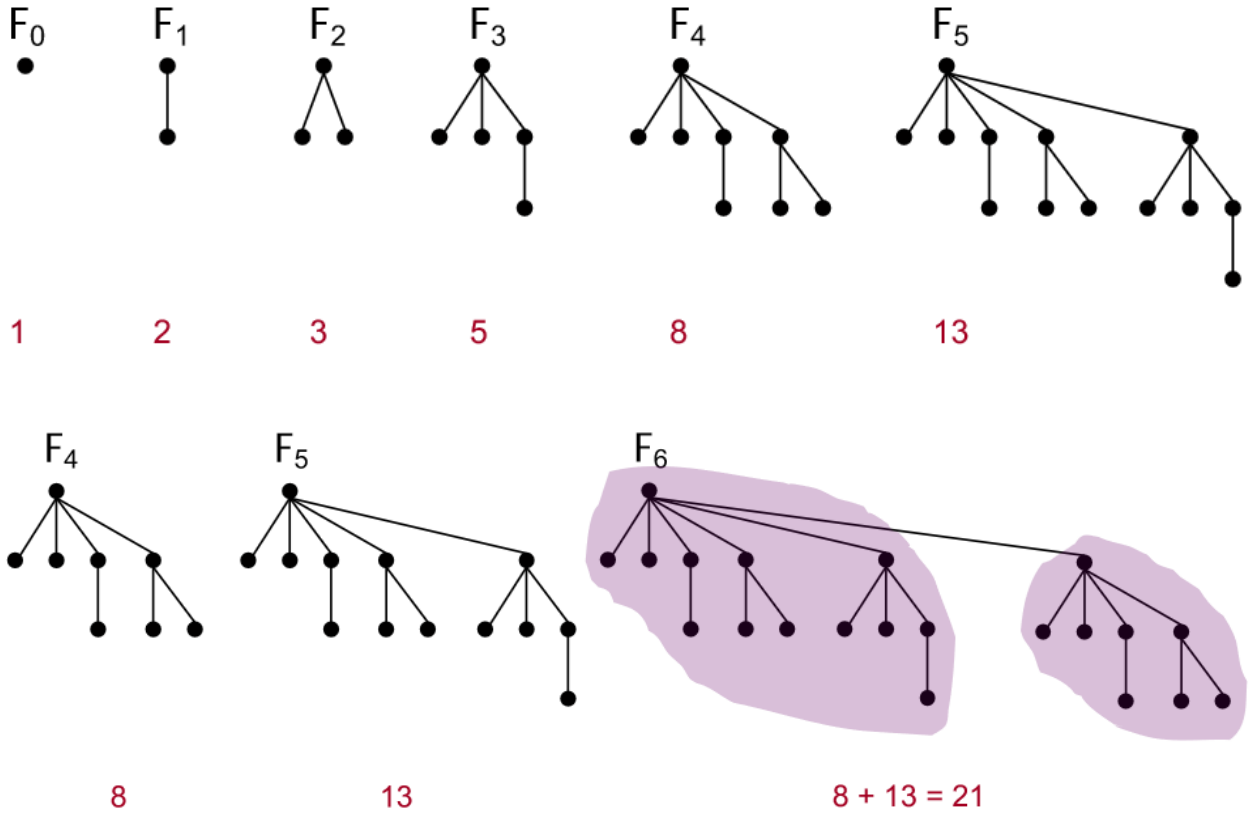
- Thus, right now

$$rank(y_i) \geq i - 2$$

Definition: Let

$F_k$be smallest possible tree of rank

***k*satisfying property**



+ Solve:

$$F_k \geq F_{k-2} + F_{k-3} + \cdots + F_0 + 1$$

Lemma (Fibonacci fact):

$F_k \geq \phi^k$, where

$$\boldsymbol{\phi = \frac{1+\sqrt{5}}{2} \approx 1.618}$$

- Induction on

$$k$$

  - Base case:

$F_0 = 0$,

$F_1 = 1$(slightly non-standard definition)

  - $F_{k+2} = F_k + F_{k+1} \geq \phi^k + \phi^{k+1} = \phi^k(1+\phi) = \phi^k(\phi^2) = \phi^{k+2}$

Corollary:

$$\boldsymbol{rank(H) \leq \log_\phi n = O(\log n)}$$

Application

- *Dijkstra*'s SSSP algorithm and *Prim*'s MST algorithm:

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

Summary

Operation	**Linked List**	**Binary Heap**	**Binomial Heap**	**Fibonacci Heap***
$MakeHeap$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
$isEmpty$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
$Insert$	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
$DeleteMin$	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
$DecreaseKey$	$O(n)$	$O(\log n)$	$O(\log n)$	$O(1)$
$Delete$	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
$Union$	$O(1)$	$O(n)$	$O(\log n)$	$O(1)$
$FindMin$	$O(n)$	$O(1)$	$O(\log n)$	$O(1)$
n: number of elements in priority queue				
*amortized analysis				