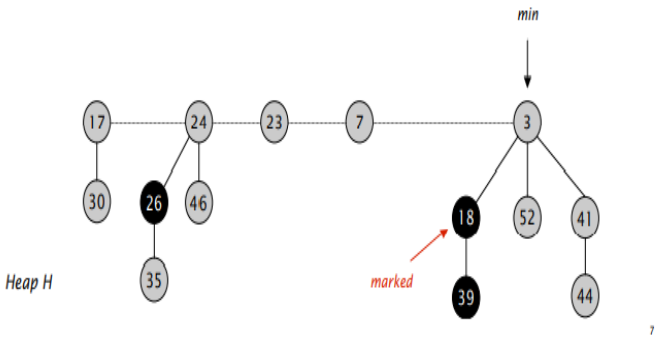
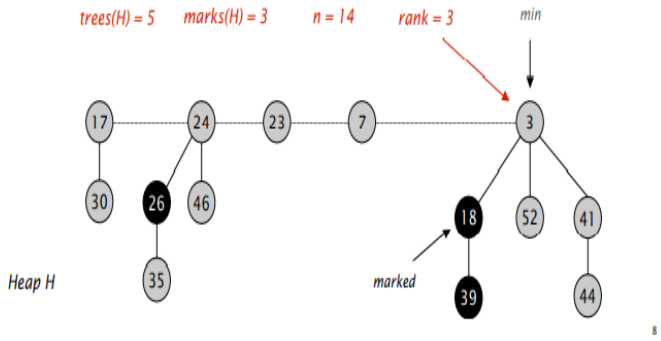
FIBONACCI HEAPS

- Fibonacci heaps are collection of trees.
- Similar to binomial heaps, but less rigid structure.
- ***Binomial heap*: eagerly consolidate trees after each insert.**
- ***Fibonacci heap*: lazily defer consolidation until next delete-min.**
- Trees within Fibonacci heaps are rooted but unordered. It can have any number of trees with the same order.Some nodes will be marked and some will be unmarked.
- So the basic properties of Fibonacci trees can be;
    - Set of heap-ordered trees.
    - Roots and children are circularly doubly linked.
    - Maintain pointer to minimum element.
    - Set of marked nodes. (use to keep heaps flat)
    - Newly created nodes will be always unmarked
- Basic terminologies:
    - Min[H] – points to the root containing the minimum key
    - n[H] – number of nodes in the heap H
    - t[H] – number of trees in the heap H
    - m[H] – number of marked nodes in the heap H
    - ϕ[H] – potential function

        This is the function to analyse the performance of the heap H.

        ϕ[H] = t[H] + 2m[H]



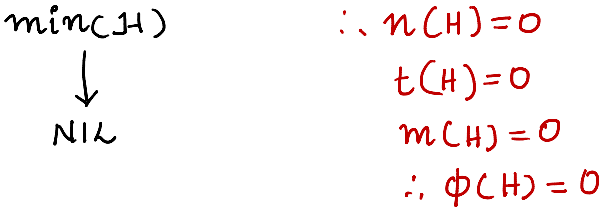


- Basic operations in Fibonacci heap
    1. Create node
    2. Insert_node
    3. Find_min
    4. Union
    5. Extract_min
    6. Decrease key
    7. Delete_node
- Time complexity comparison
    - In most of the operations, Fibonacci heaps consume less time while compared to binomial.
    - The time complexity comparison is given below.

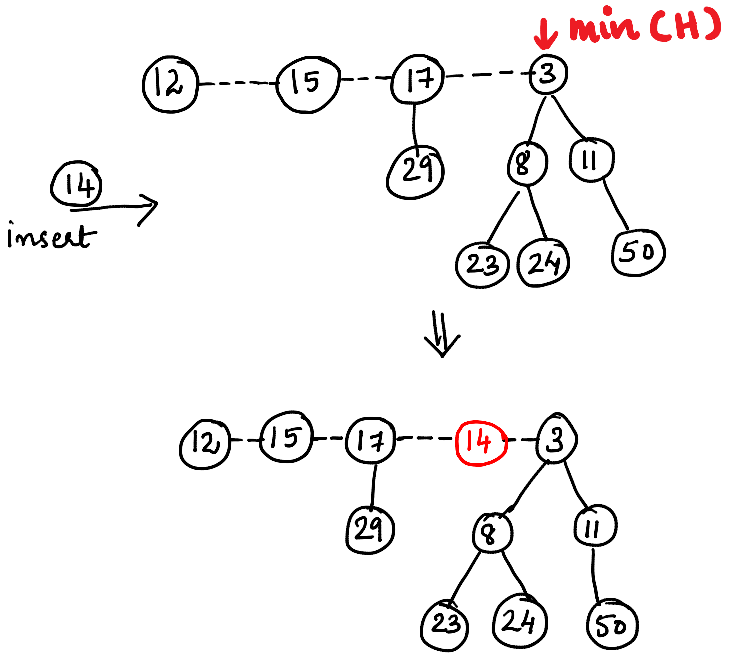
Procedure	Binary heap (worst-case)	Fibonacci heap (amortized)
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$	$O(\lg n)$
UNION	$\Theta(n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\lg n)$	$\Theta(1)$
DELETE	$\Theta(\lg n)$	$O(\lg n)$

- Lets see the Fibonacci heap operations in detail;

1. **CREATE_NODE**
   - This means creating an empty node
   - In that case min pointer will be pointed to NIL

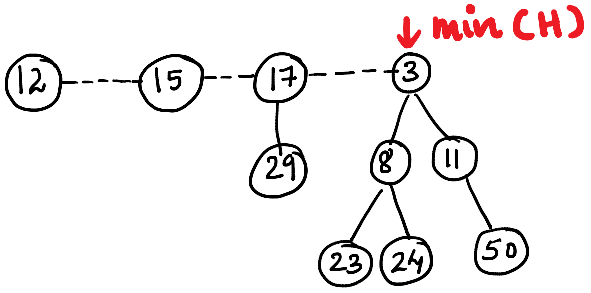


2. **INSERT NODE**
   - Operation **insert** works by creating a new heap with one element and doing merge.
   - It adds the newly created node to the immediate **left** of the ***min(H).***
   - This takes constant time, and the potential increases by one, because the number of trees increases.
   - The amortized cost is thus still constant.
   - Example :



3. FIND MIN

- The minimum element is always given by the min pointer.
- Find minimum is now trivial because we keep the pointer to the node containing it.
- It does not change the potential of the heap, therefore both actual and amortized cost are constant.
- That is, *The minimum node of a Fibonacci heap H is given by the pointer min[H], so we can find the minimum node in O(1) actual time. Because the potential of H does not change, the amortized cost of this operation is equal to its O(1) actual cost.*
-



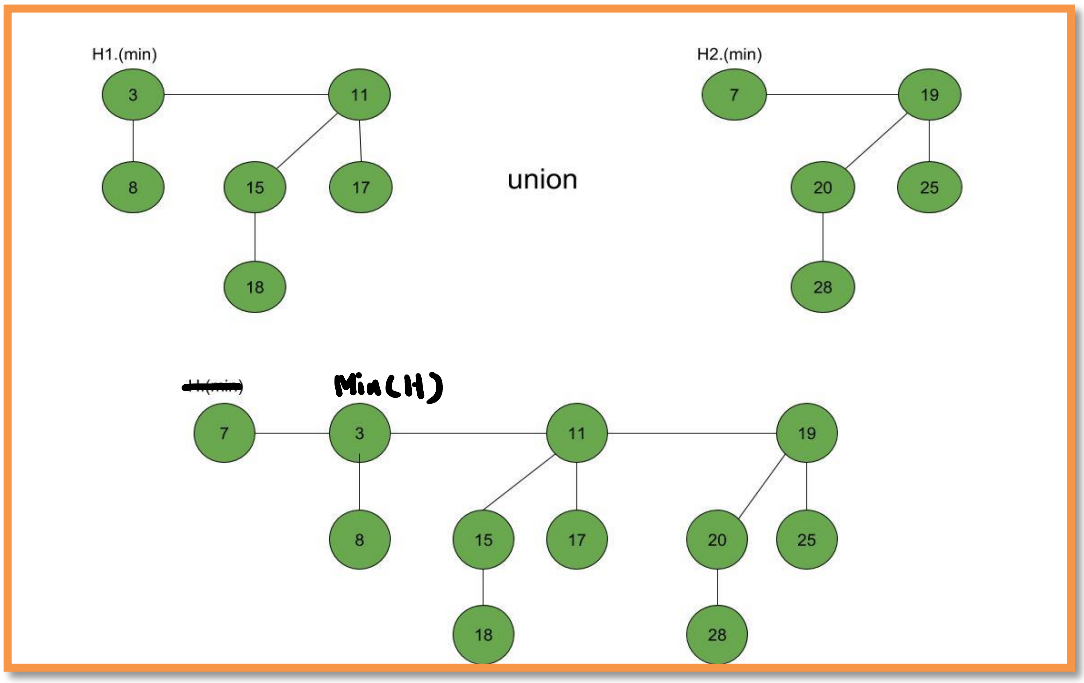
FindMin will return value pointed by the min(H). Here it is 3. Time taken is 1 as no search is needed. ∴ After the operation, the ϕ(H) remain same.

4. UNION/MERGE

- Union is implemented simply by concatenating the lists of tree roots of the two heaps.
- This can be done in constant time and the potential does not change, leading again to constant amortized time.
- Union: Union of two Fibonacci heaps H1 and H2 can be accomplished as follows:

1. Join root lists of Fibonacci heaps H1 and H2 and make a single Fibonacci heap H.
2. If H1(min) < H2(min) then:
    a. H(min) = H1(min).
3. Else:
    a. H(min) = H2(min).

- Example:

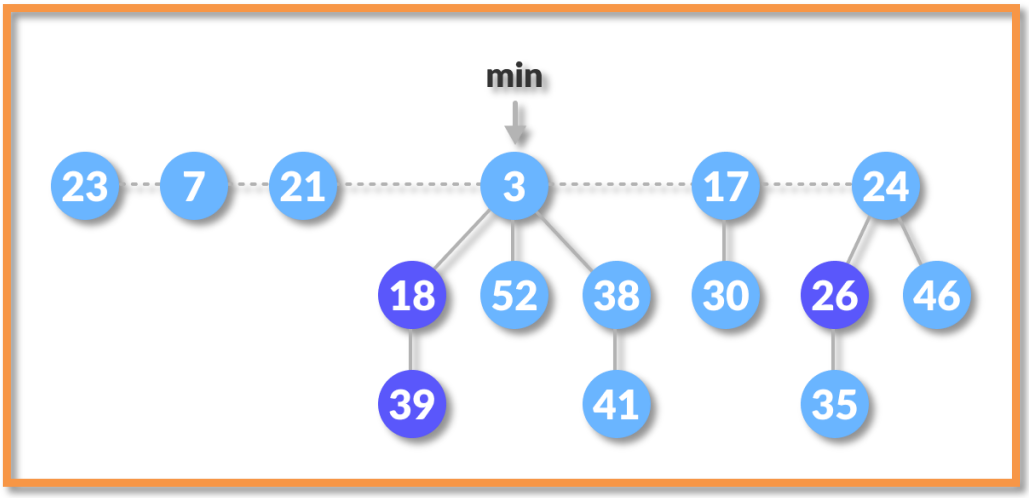


5. EXTRACT MIN

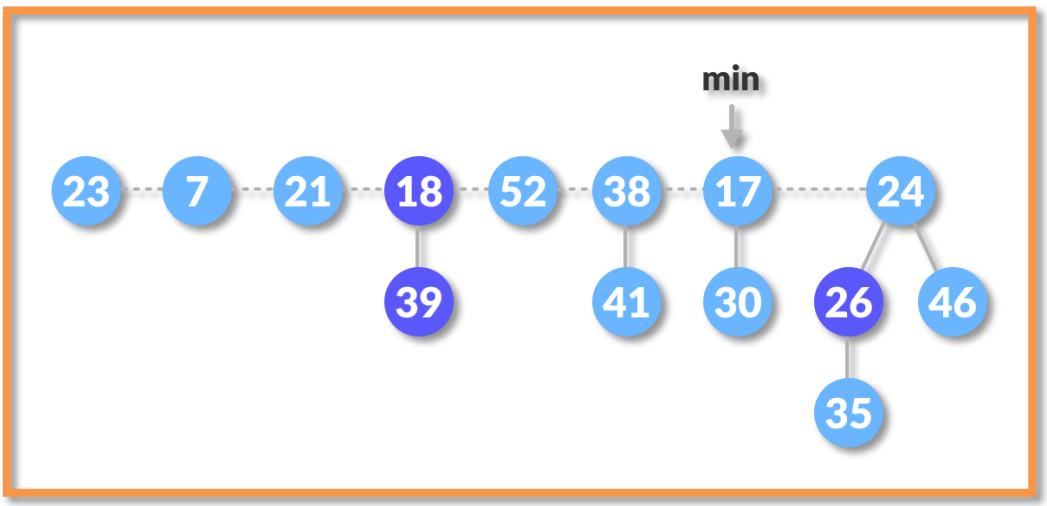
- It is the **most important operation on a fibonacci heap**. In this operation, ***the node with minimum value is removed from the heap and the tree is re-adjusted.***
- Extract minimum (same as delete minimum) operates in three phases. First we take the root containing the minimum element and remove it. Its children will become roots of new trees.
- If the number of children was d, it takes time O(d) to process all new roots and the potential increases by d−1. Therefore, the amortized running time of this phase is O(d) = O(log n).
- Algorithm:

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 a-c) until there are no multiple roots with the same degree.
    a) Map the degree of current root (min-pointer) to the degree in the array.
    b) Map the degree of next root to the degree in array.
    c) 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).

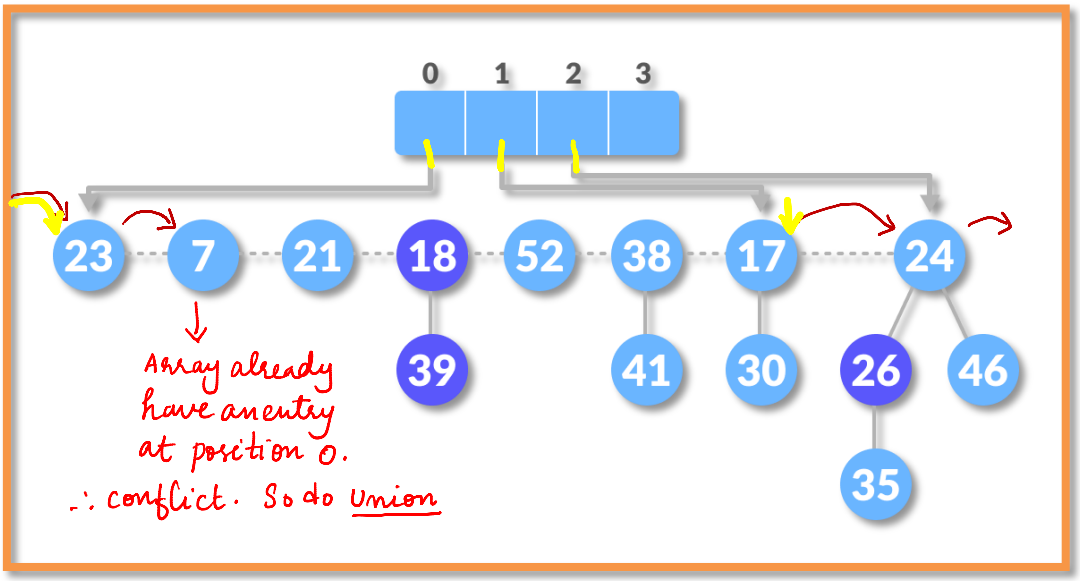
- Example implementation:



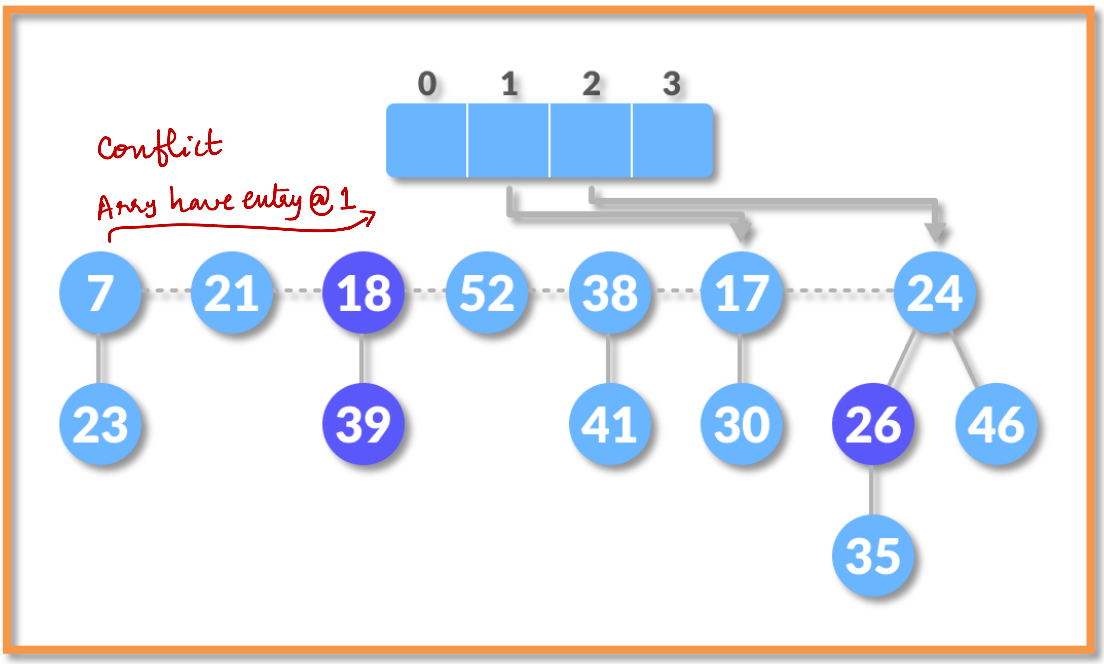
Delete the min node, add all its child nodes to the root list and set the min-pointer to the next root in the root list.



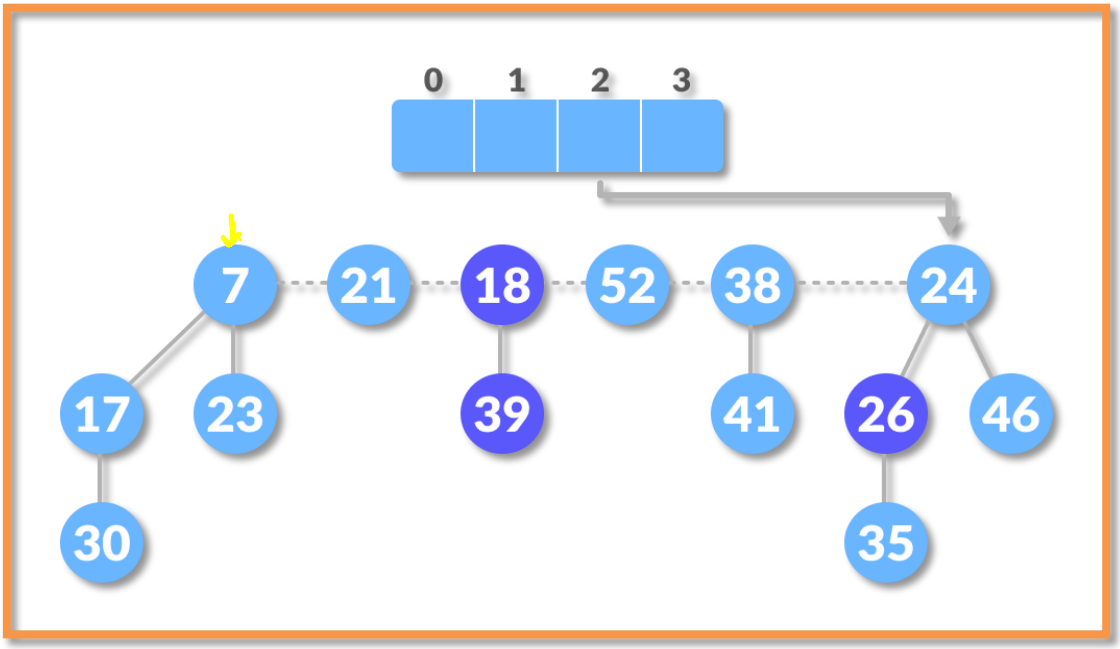
The maximum degree in the tree is 3. Create an array of size 4 and map degree of the next roots with the array.



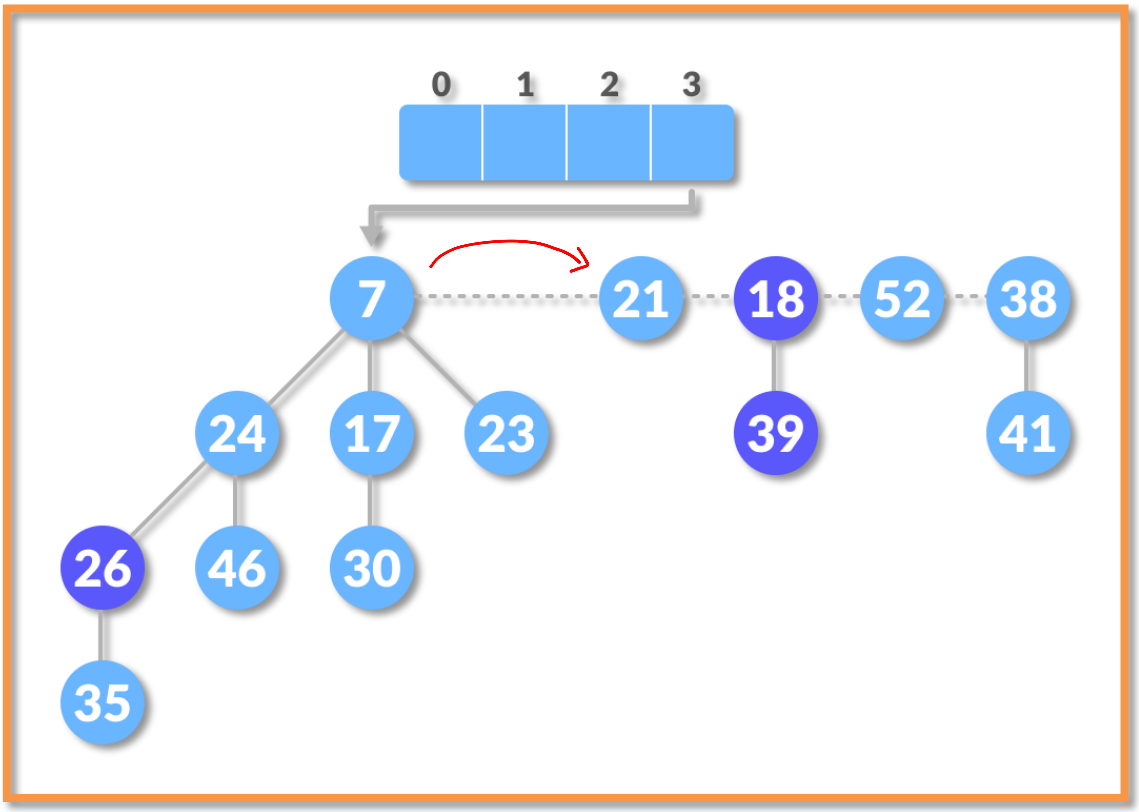
Here, 23 and 7 have the same degrees, so unite them.



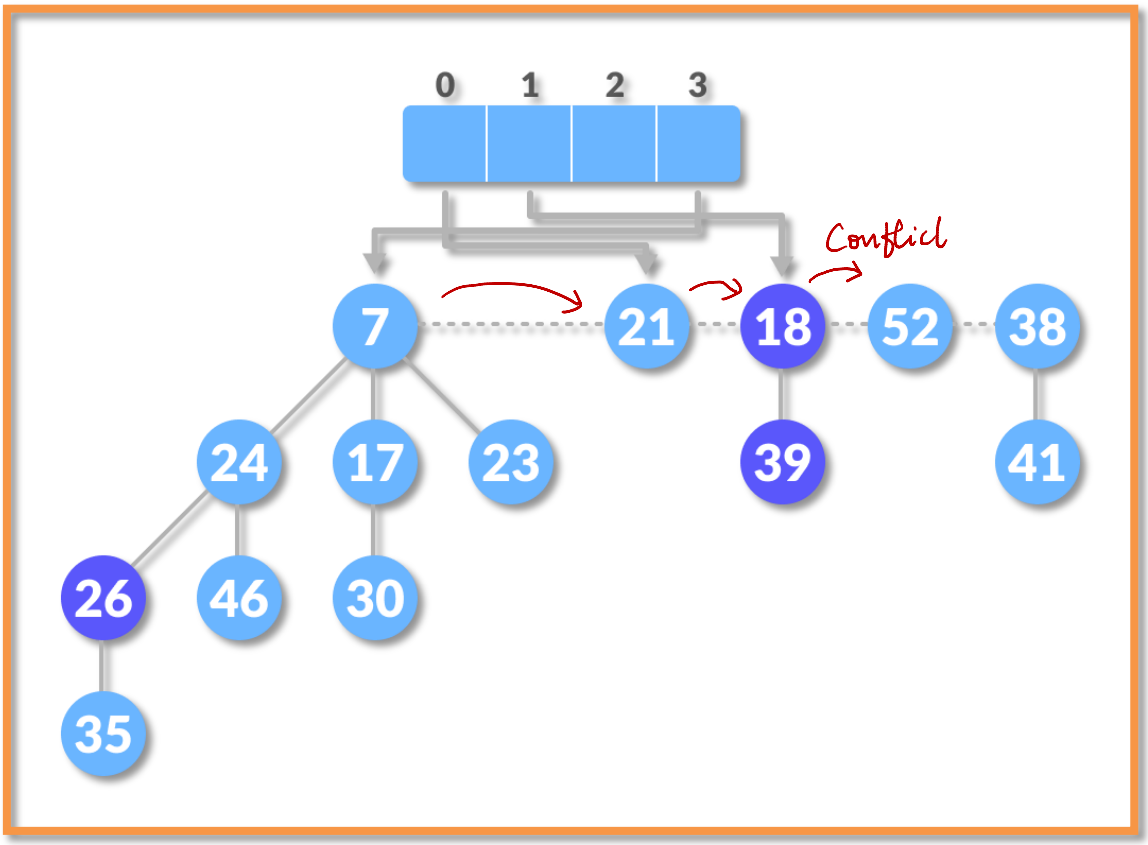
Again, 7 and 17 have the same degrees, so unite them as well



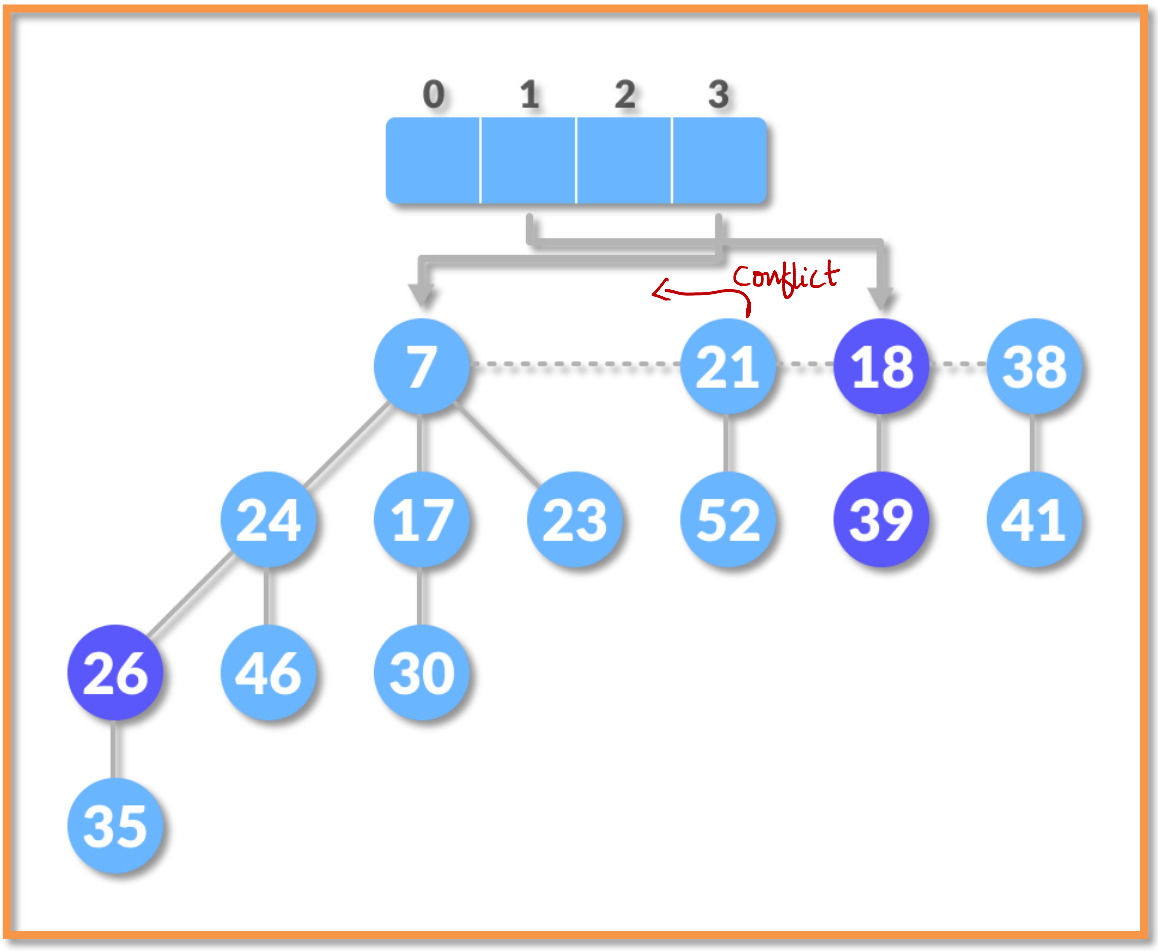
Again 7 and 24 have the same degree, so unite them



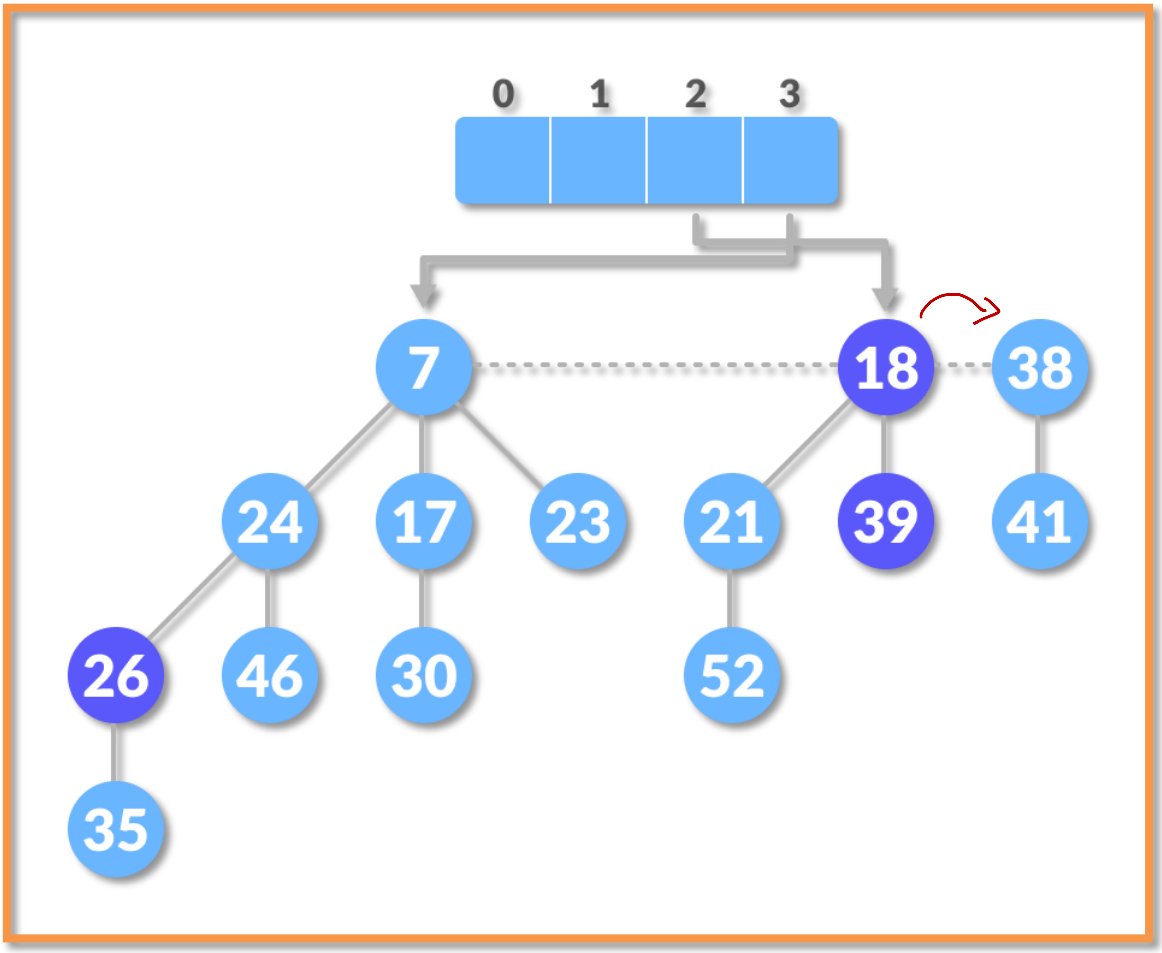
Map the next nodes.



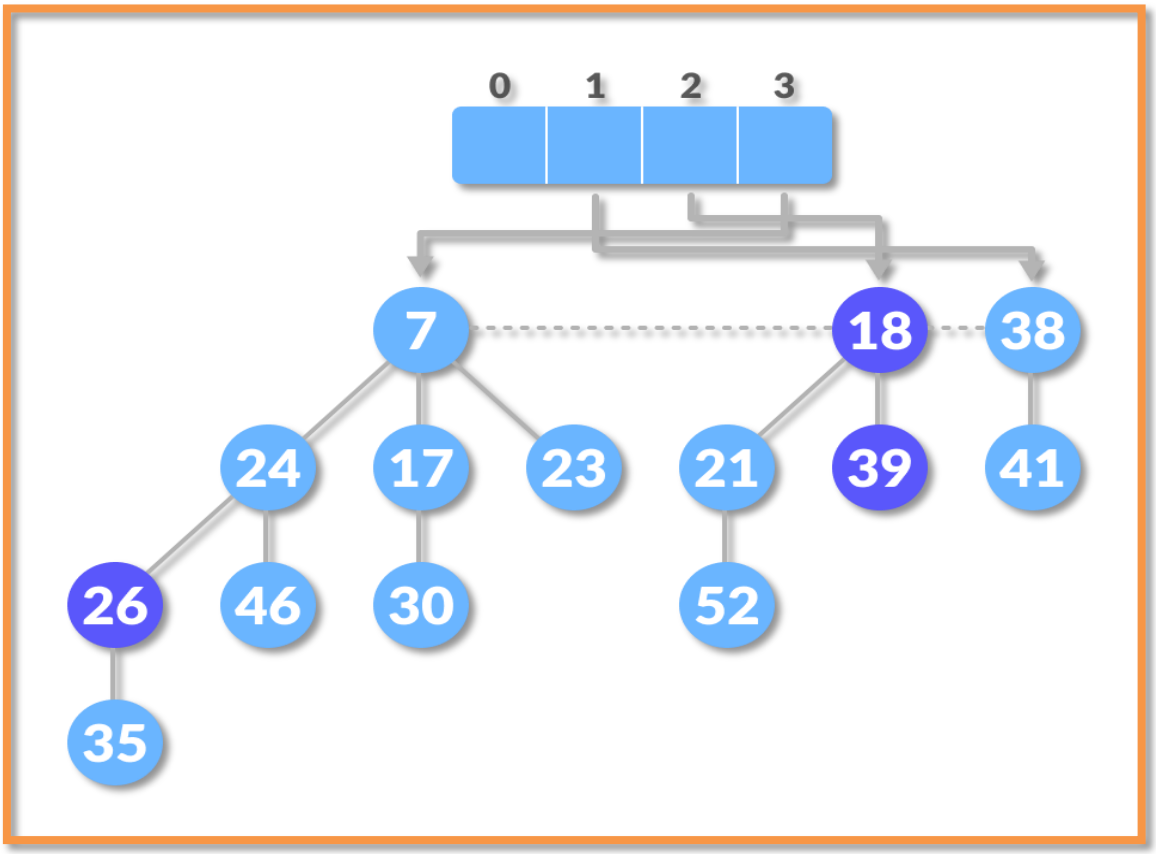
Again, 52 and 21 have the same degree, so unite them



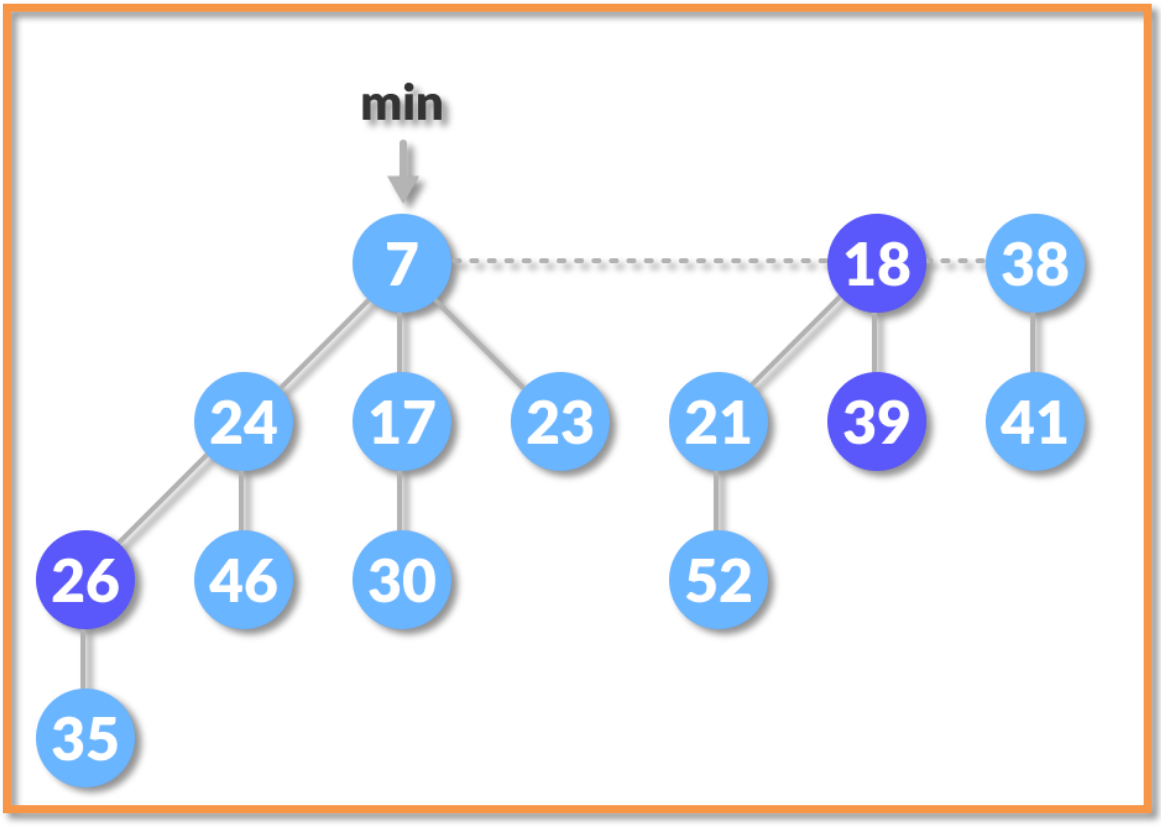
Similarly, unite 21 and 18



Map the remaining root



All roots traversed and no conflicts remain. Hence the final heap is;



6. DECREASE KEY

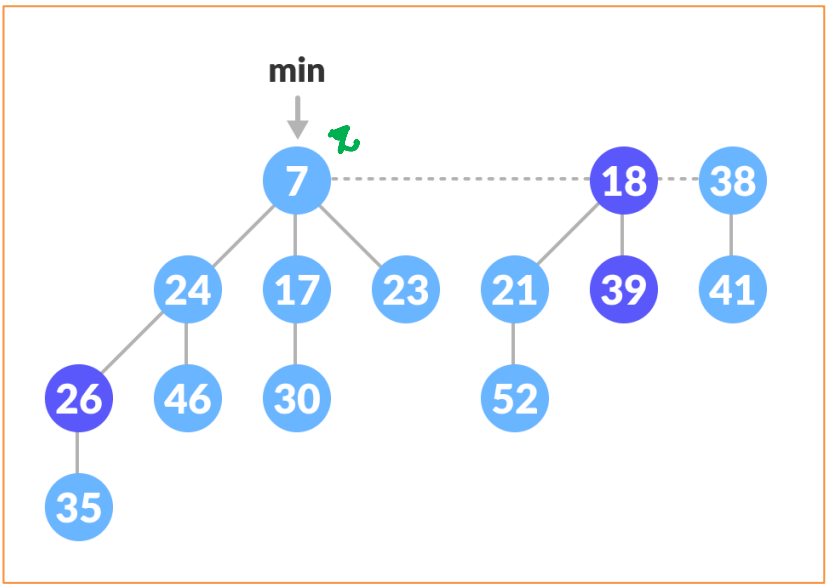
- In decreasing a key operation, the value of a key is decreased to a lower value.
- Decrease key will take the node, decrease the key and if the heap property becomes violated (the new key is smaller than the key of the parent), the node is cut from its parent.
- Following functions are used for decreasing the key.
- **Decrease-Key(H,x,k)**
  - Select the node to be decreased, x, and change its value to the new value k.
  - If the parent of x, ie 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**.
  - If the key of x is smaller than the key of min, then mark x as min.
- **Cut**
  - Remove x from the current position and add it to the root list.
  - If x is marked, then mark it as false.
- **Cascading-Cut**
  - If the parent of y is not null then follow the following steps.
    - If y is unmarked, then mark y.
    - Else, call Cut(y) and Cascading-Cut(parent of y)
- For implementation , lets see a formal algorithm:

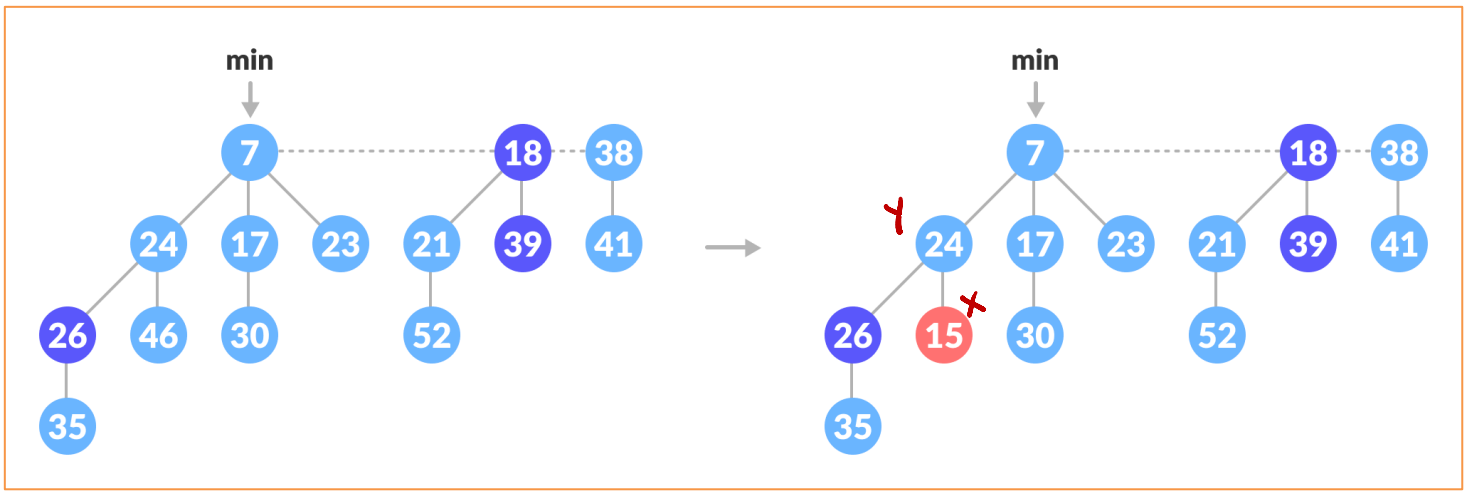
```
FIB-HEAP-DECREASE-KEY(H,x,k)
1  if k > key[x]
2     then error
      "new key is greater than current key"

3  key[x] ← k
4  y ← p[x]

5  if y ≠ NIL and key[x] < key[y]
6     then CUT(H,x,y)
7          CASCADING-CUT(H,y)
8  if key[x] < key[min[H]]
9     then min[H] ← x
```

```
CUT(H,x,y)
1  remove x from the child list of y,
decrementing degree[y]
2  add x to the root list of H
3  p[x] ← NIL
4  mark[x] ← FALSE
```

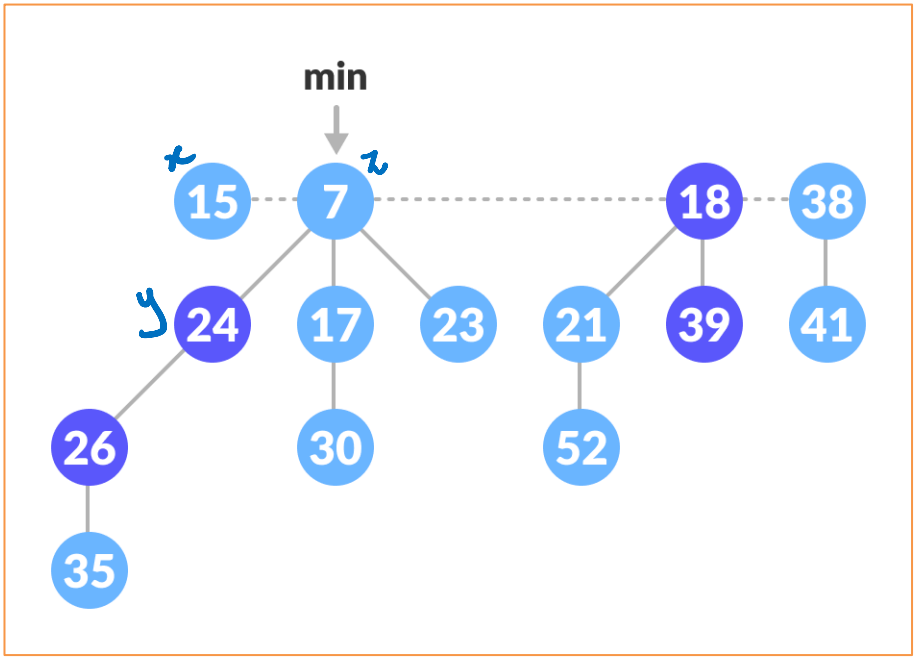
```
CASCADING-CUT(H,y)
1  z ← p[y]
2  if z ≠ NIL
3     then if mark[y] = FALSE
4             then mark[y] ← TRUE
5             else CUT(H,y,z)
6                  CASCADING-CUT(H,z)
```

- ***Example***: Decreasing 46 to 15.

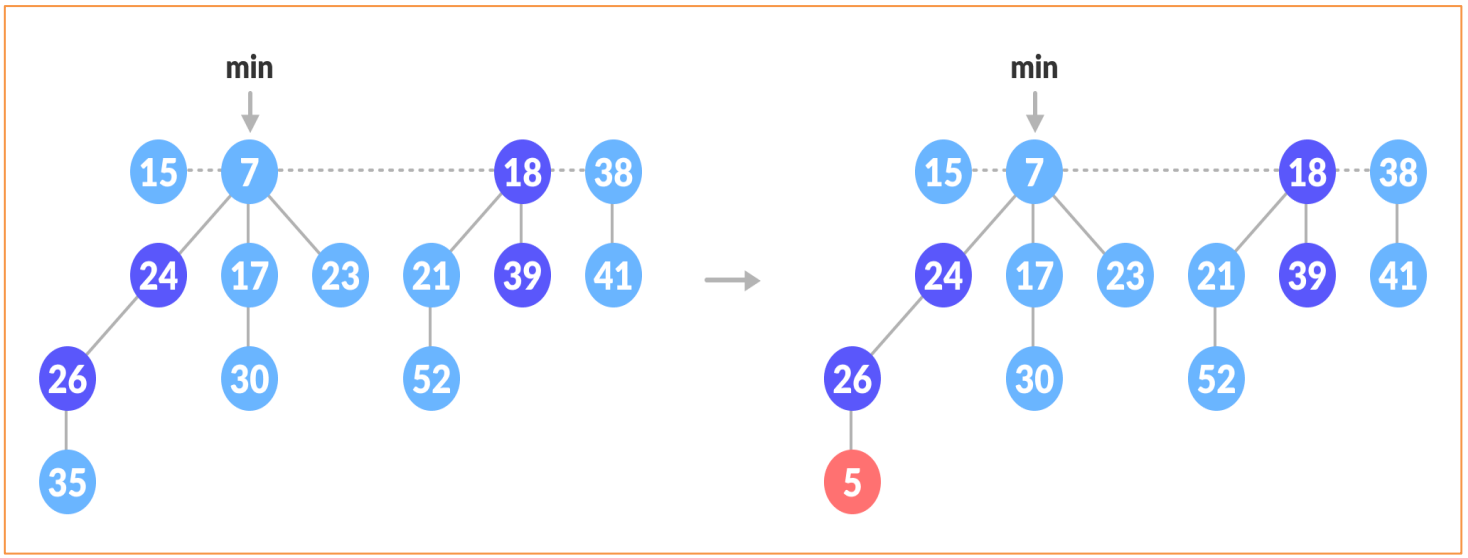




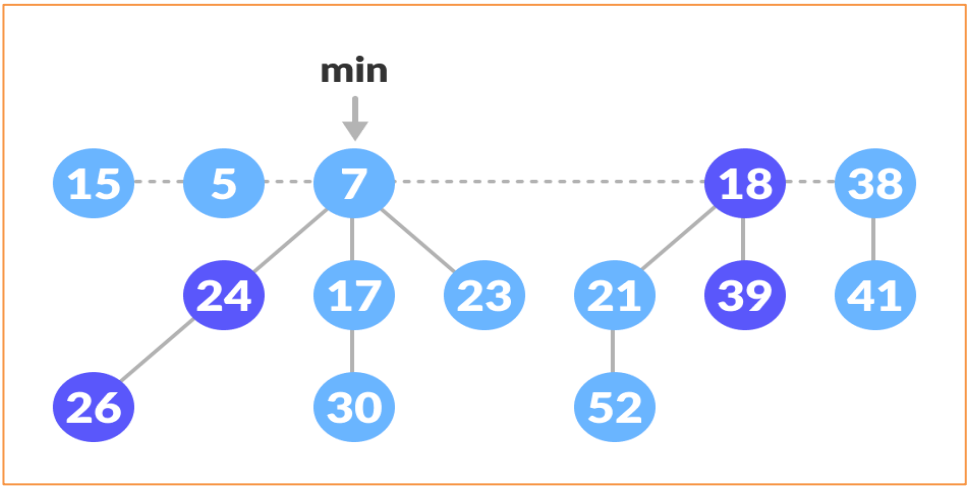
Cut part: Since **24**(ie **Y** parent of decreased key) ≠ **nill and 15 < its parent**, **cut** it and add it to the root list.
Cascading-Cut part: mark 24(as its unmarked).



Example2: Decreasing 35 to 5

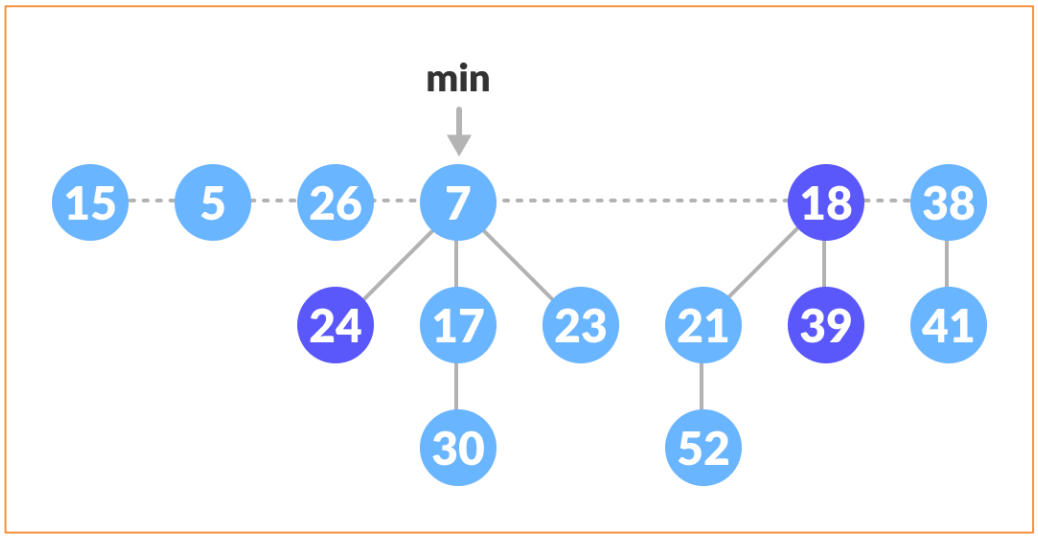


Cut part: Since 26 ≠ nill and 5<its parent, cut it and add it to the root list



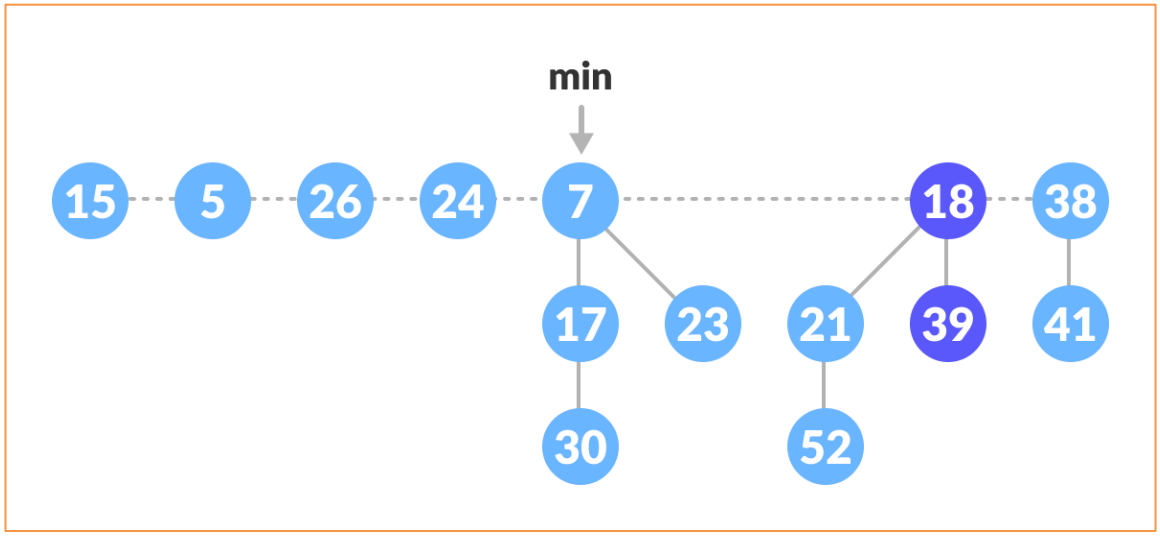
Cascading-Cut part: Since **26 is marked**, the flow ***goes to Cut and Cascading-Cut again***.

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

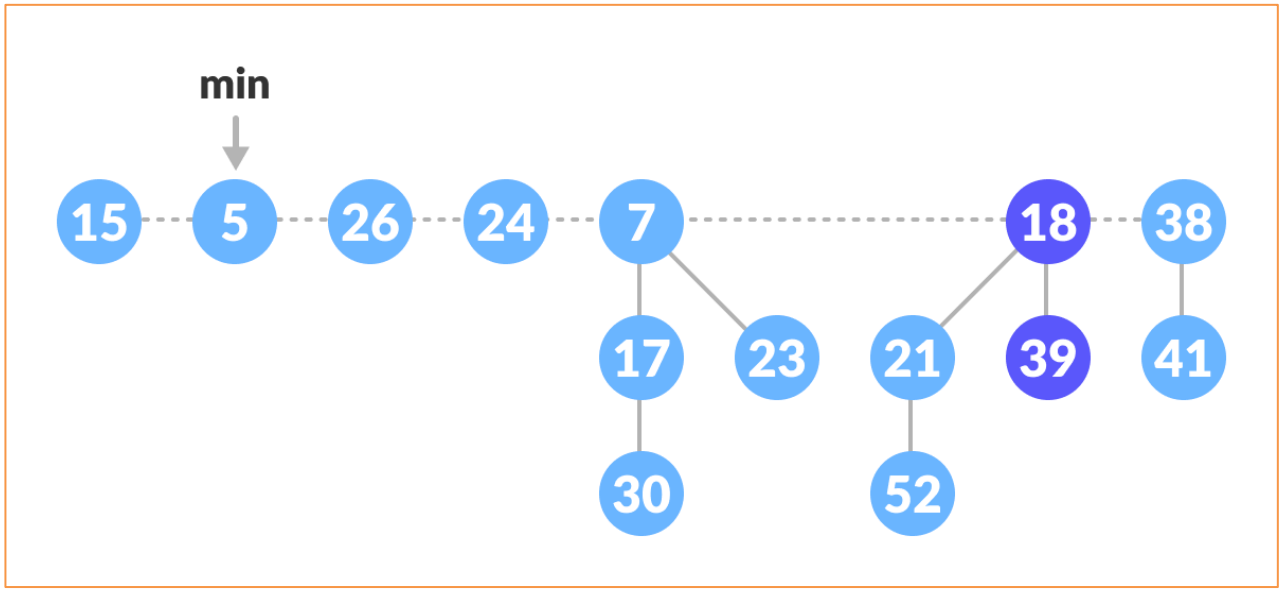


Cascading-Cut(24):

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



Since 5 < 7, mark 5 as min



7. **DELETE NODE**
   - Delete can be implemented simply by decreasing the key of the element to be deleted to minus infinity, thus turning it into the minimum of the whole heap.
   - Then we call extract minimum to remove it.
   - The amortized running time of this operation is O(log n).
   - Algorithm:
     - Let k be the node to be deleted.
     - Apply decrease-key operation to decrease the value of k to the lowest possible value (i.e. -∞).
     - Apply extract-min operation to remove this node.