Fibonacci Heap

Heaps as Priority Queues

- You have seen binary min-heaps/max-heaps
- Can support creating a heap, insert, finding/extracting the min (max) efficiently
- Can also support decrease-key operations efficiently
- However, not good for merging two heaps
  - O(n) where n is the total no. of elements in the two heaps
- Variations of heaps exist that can merge heaps efficiently
  - May also improve the complexity of the other operations
  - Ex. Binomial heaps, Fibonacci heaps
- We will study Fibonacci heaps, an amortized data structure

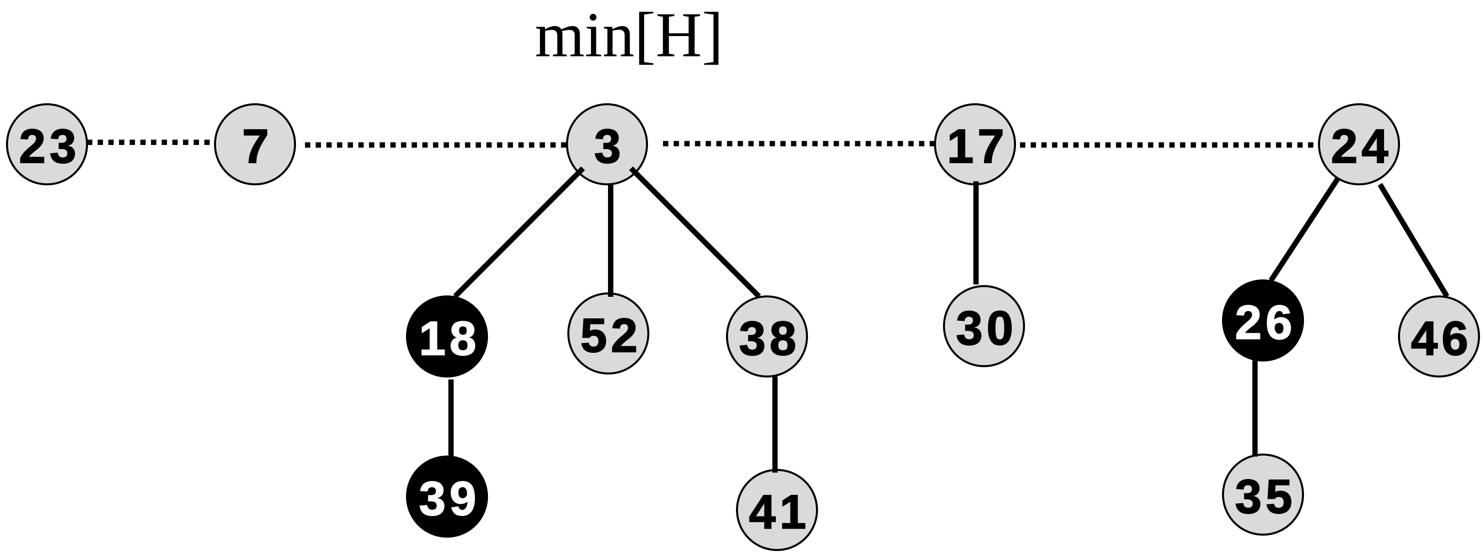
A Comparison

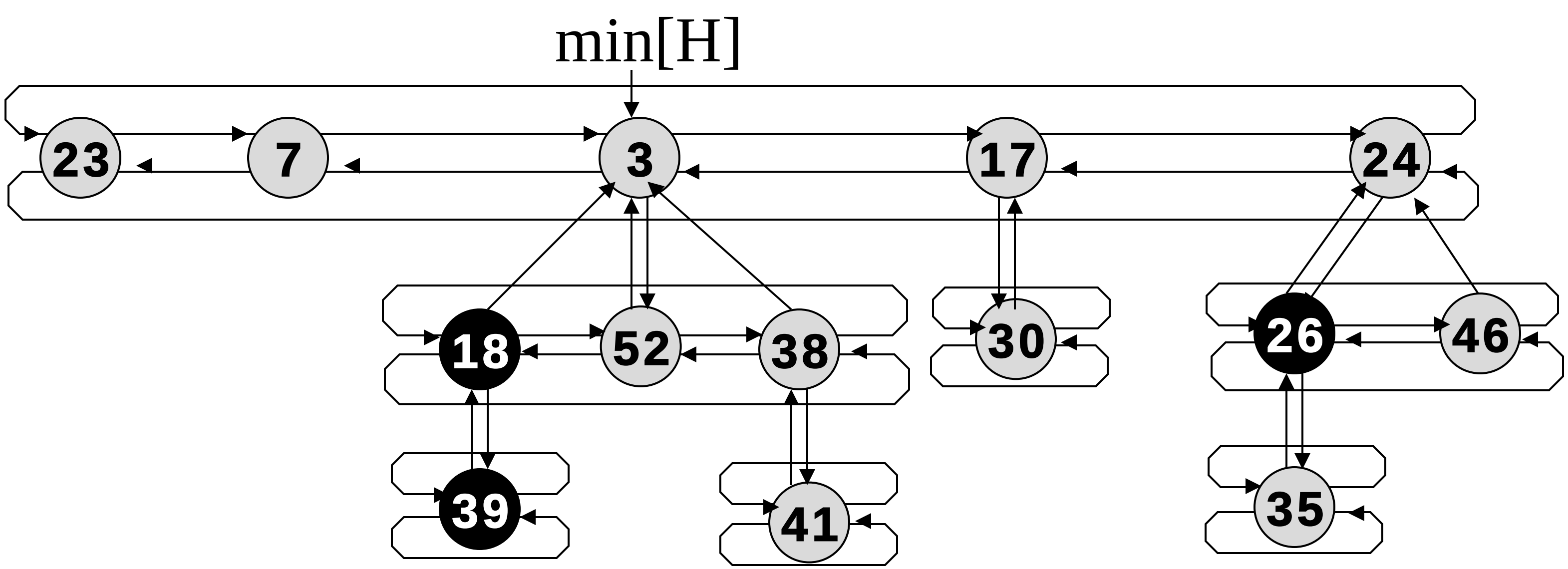
Operation	Binary heap (worst-case)	Binomial heap (worst-case)	Fibonacci heap (amortized)
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$O(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$O(\lg n)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$
MERGE/UNION	$\Theta(n)$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

Fibonacci Heap

- A collection of min-heap ordered trees
  - Each tree is rooted but "unordered" , meaning there is no order between the child nodes of a node (unlike, for ex., left child and right child in a rooted, ordered binary tree)
  - Each node x has
    - One parent pointer p[x]
    - One child pointer child[x] which points to an arbitrary child of x
    - The children of x are linked together in a circular, doubly linked list
      - Each node y has pointers left[y] and right[y] to its left and right node in the list
      - So x basically stores a pointer to start in this list of its children

- The root of the trees are again connected with a circular, doubly linked list using their left and right pointers
- A Fibonacci heap H is defined by
  - A pointer min[H] which points to the root of a tree containing the minimum element (minimum node of the heap)
  - A variable n[H] storing the number of elements in the heap


min[H]
23
7
3
17
24
18
52
38
30
26
46
39
41
35


min[H]
23
7
3
17
24
18
52
38
30
26
46
39
41
35

Additional Variables

- Each node x also has two other fields
  - degree[x] – stores the number of children of x
  - mark[x] – indicates whether x has lost a child since the last time x was made the child of another node
    - We will denote marked nodes by color black, and unmarked ones by color grey
    - A newly created node is unmarked
    - A marked node also becomes unmarked whenever it is made the child of another node

Amortized Analysis

- We mentioned Fibonacci heap is an amortized data structure
- We will use the potential method to analyse
- Let $t(H)$ = no. of trees in a Fibonacci heap H
- Let $m(H)$ = number of marked nodes in H
- Potential function used

$$\Phi(H) = t(H) + 2m(H)$$

Operations

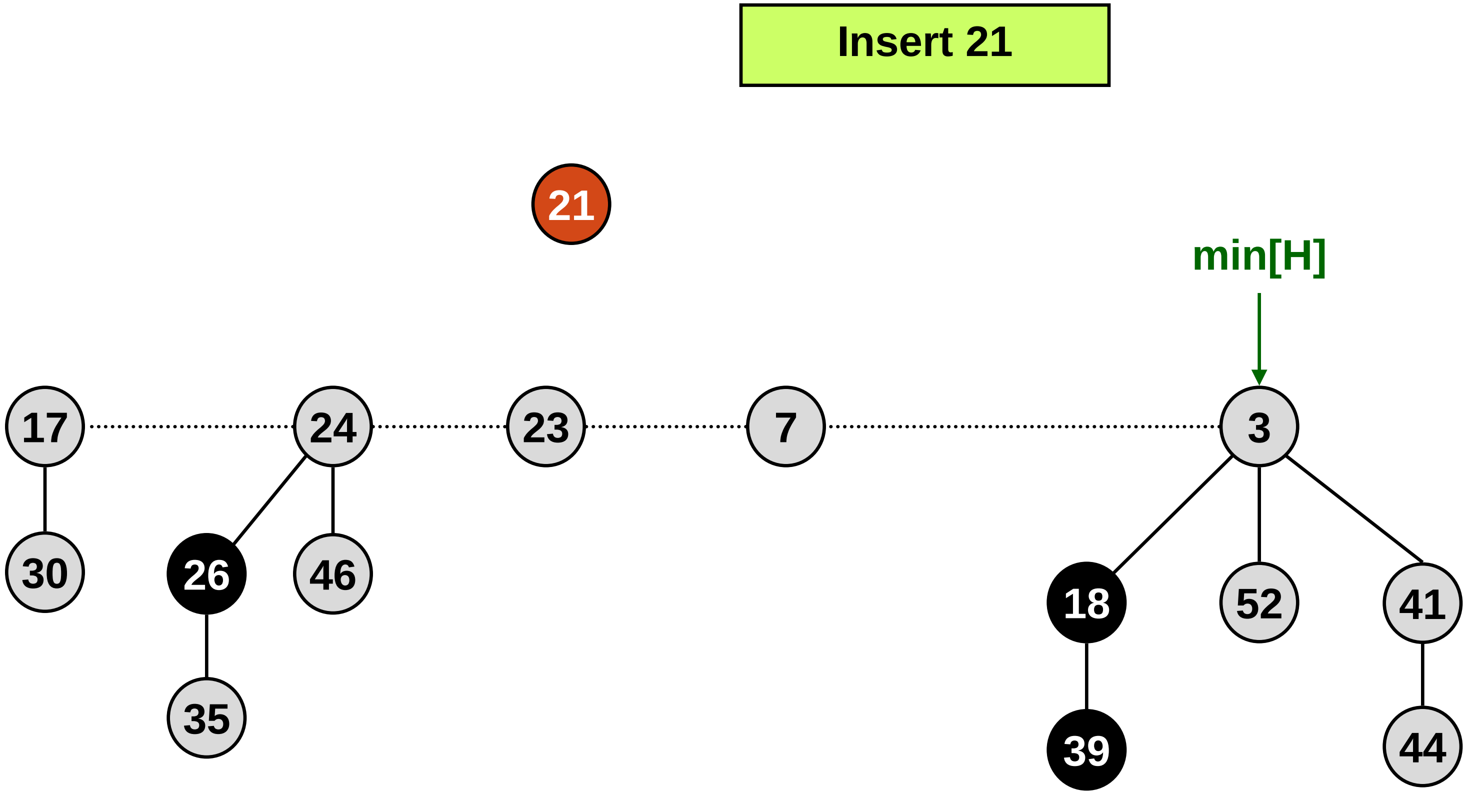
- Create an empty Fibonacci heap
- Insert an element in a Fibonacci heap
- Merge two Fibonacci heaps (Union)
- Extract the minimum element from a Fibonacci heap
- Decrease the value of an element in a Fibonacci heap
- Delete an element from a Fibonacci heap

Creating a Fibonacci Heap

- This creates an empty Fibonacci heap
- Create an object to store min[H] and n[H]
- Initialize min[H] = NIL and n[H] = 0
- Potential of the newly created heap Φ (H) = 0
- Amortized cost = actual cost = O(1)

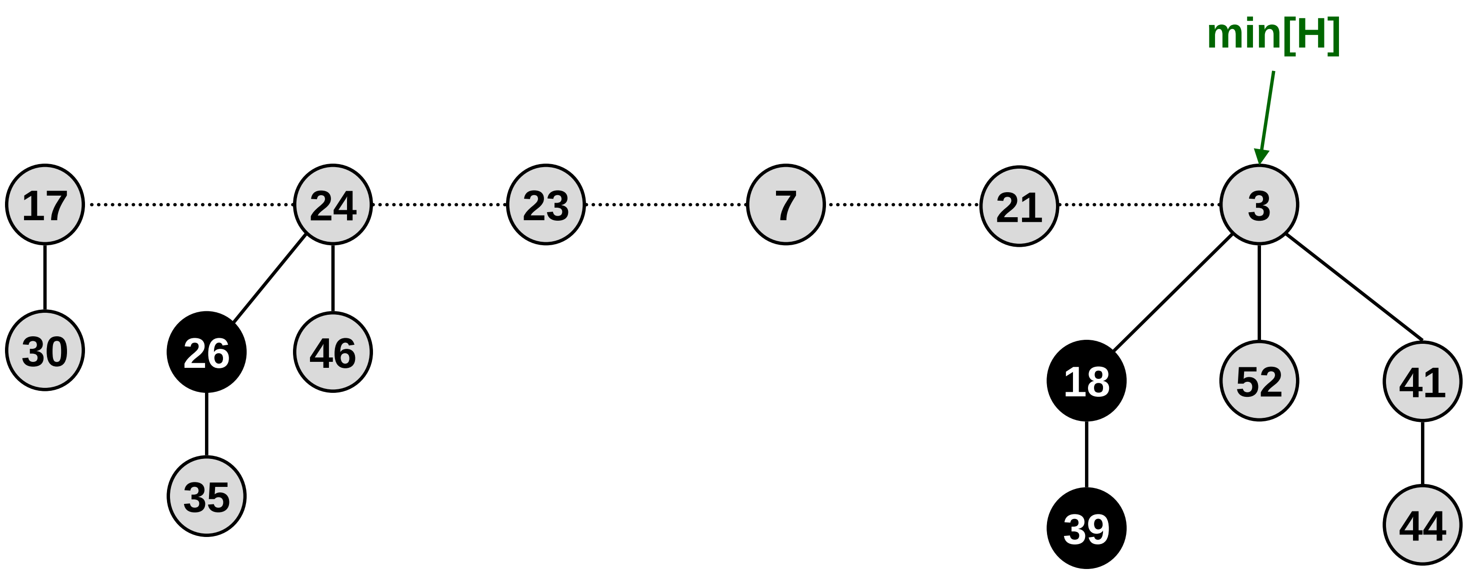
Inserting an Element

- Add the element to the left of min[H]
- Update min[H] if needed



Inserting an Element (contd.)

- Add the element to the left of node pointed to by min[H]
- Update min[H] if needed

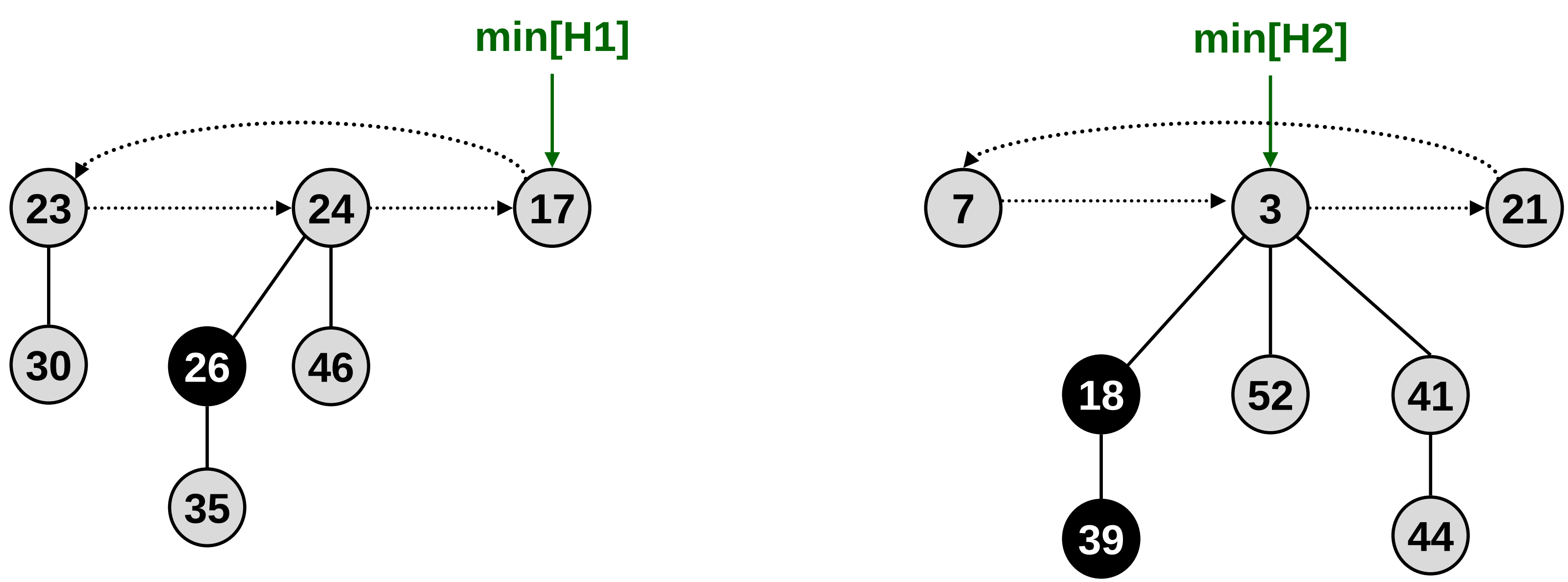


Amortized Cost of Insert

- Actual Cost O(1)
- Change in potential +1
  - One new tree, no new marked node
- Amortized cost O(1)

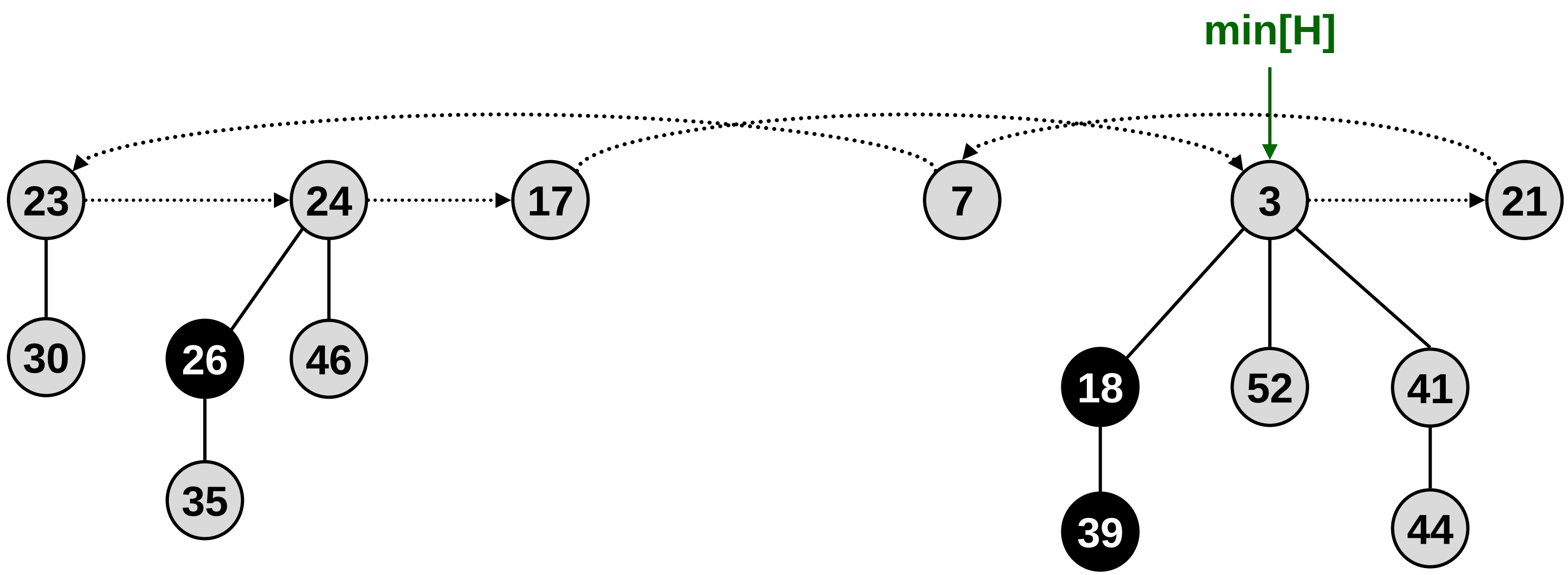
Merging Two Heaps (Union)

- Concatenate the root lists of the two Fibonacci heaps
- Root lists are circular, doubly linked lists, so can be easily concatenated



Merging Two Heaps (contd.)

- Concatenate the root lists of the two Fibonacci heaps
- Root lists are circular, doubly linked lists, so can be easily concatenated

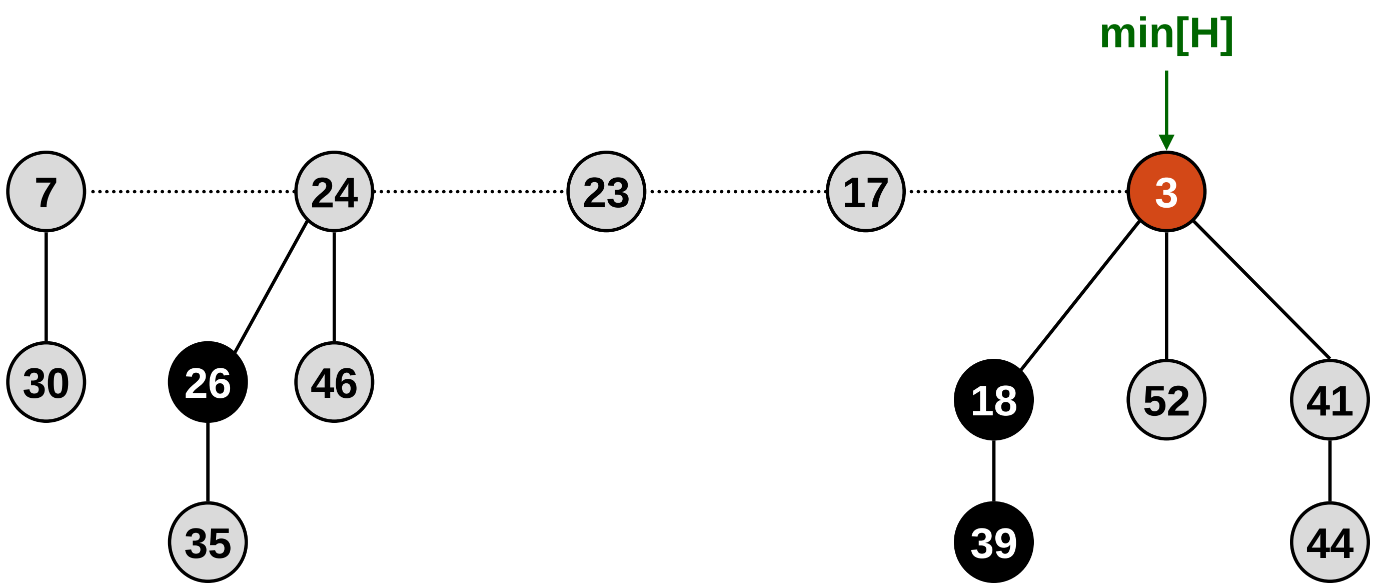


Amortized Cost of Merge/Union

- Actual cost = O(1)
- Change in potential = 0
- Amortized cost = O(1)

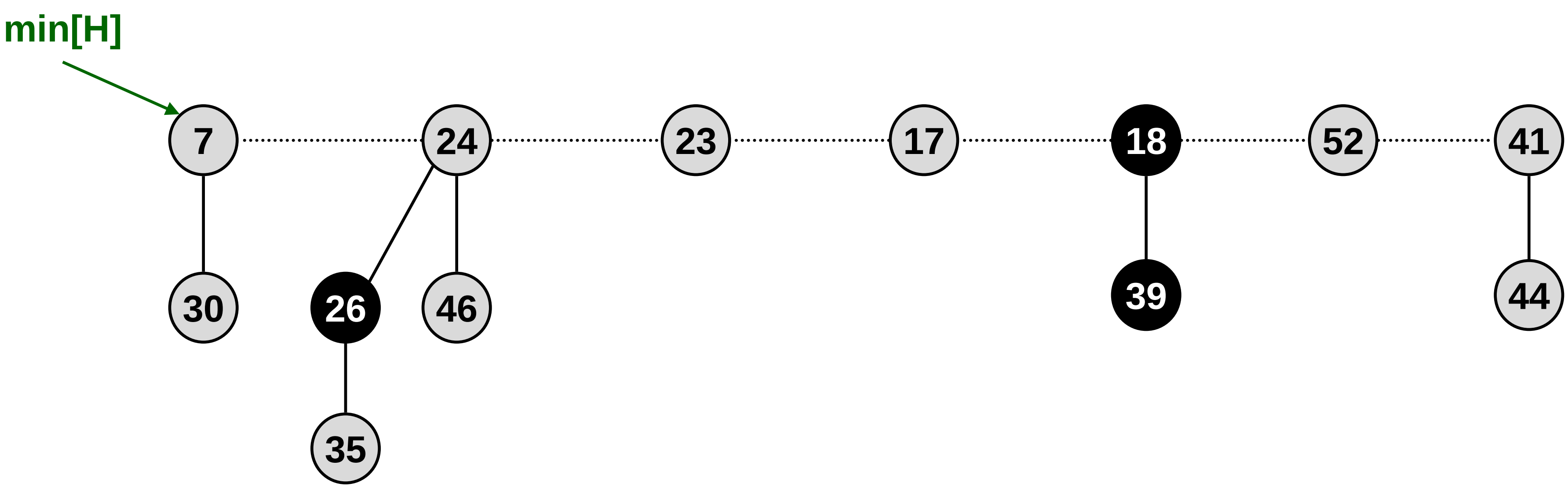
Extracting the Minimum Element

- **Step 1:**
  - Delete the node pointed to by min[H]
  - Concatenate the deleted node's children into root list



Extracting the Minimum (contd.)

- **Step 1:**
  - Delete the node pointed to by min[H]
  - Concatenate the deleted node's children into root list

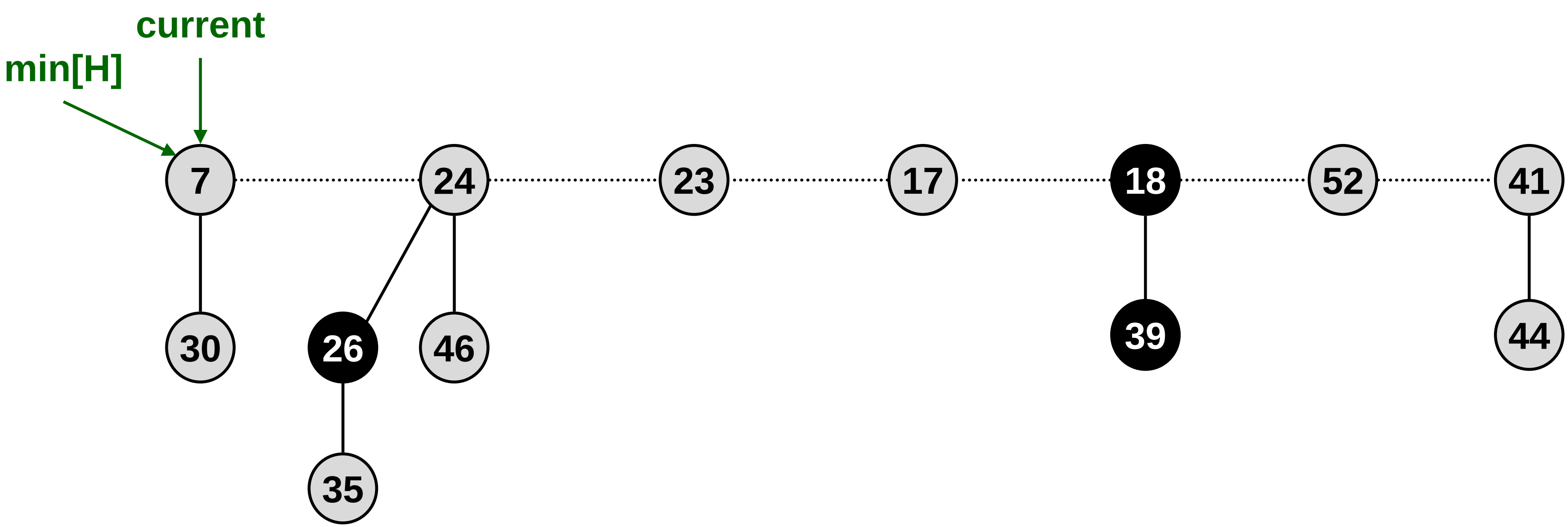


Extracting the Minimum (contd.)

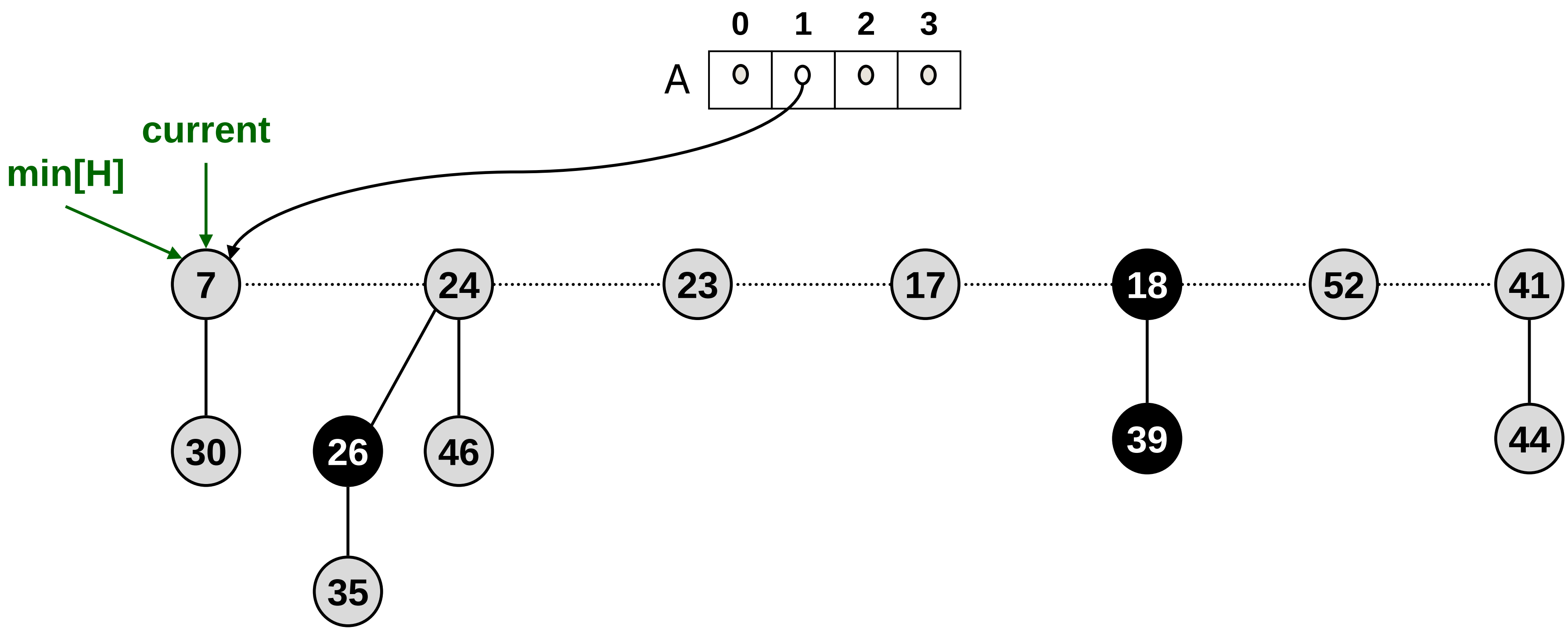
- **Step 2:** Consolidate trees so that no two roots have same degree
  - Traverse the roots from min towards right
  - Find two roots x and y with the same degree, with key[x] ≤ key[y]
  - Remove y from root list and make y a child of x
  - Increment degree[x]
  - Unmark y if marked
- We use an array A[0..D(n)] where D(n) is the maximum degree of any node in the heap with n nodes, initially all NIL
  - If A[k] = y at any time, then degree[y] = k

Extracting the Minimum (contd.)

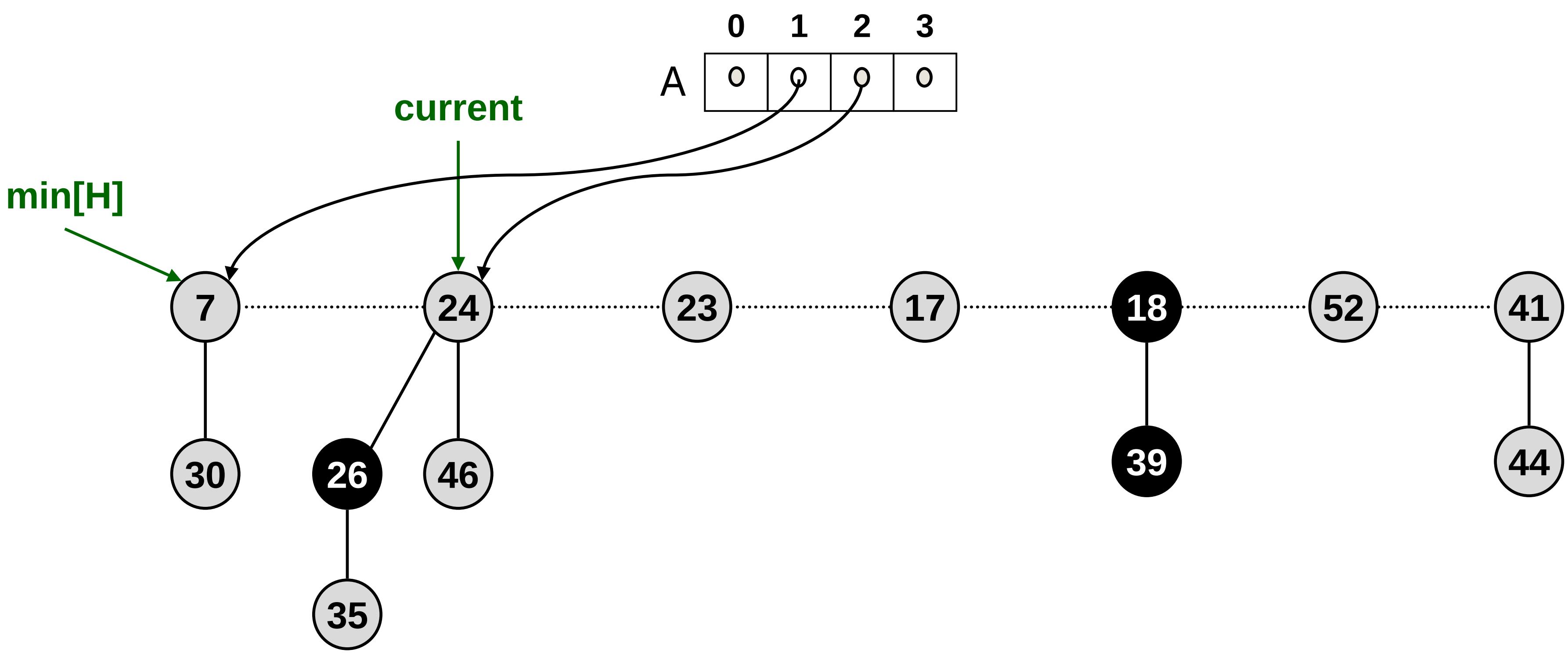
- **Step 2:** Consolidate trees so that no two roots have same degree. Update min[H] with the new min after consolidation.



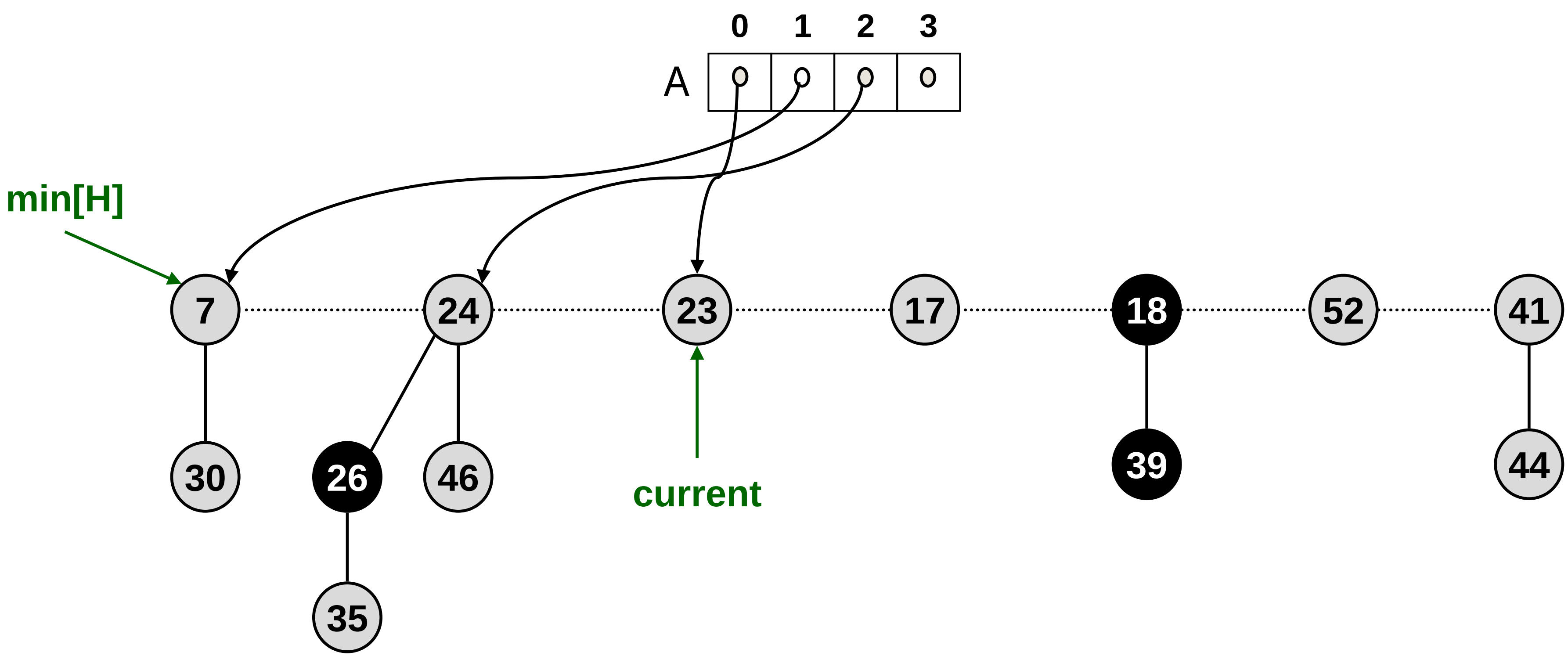
Extracting the Minimum (contd.)



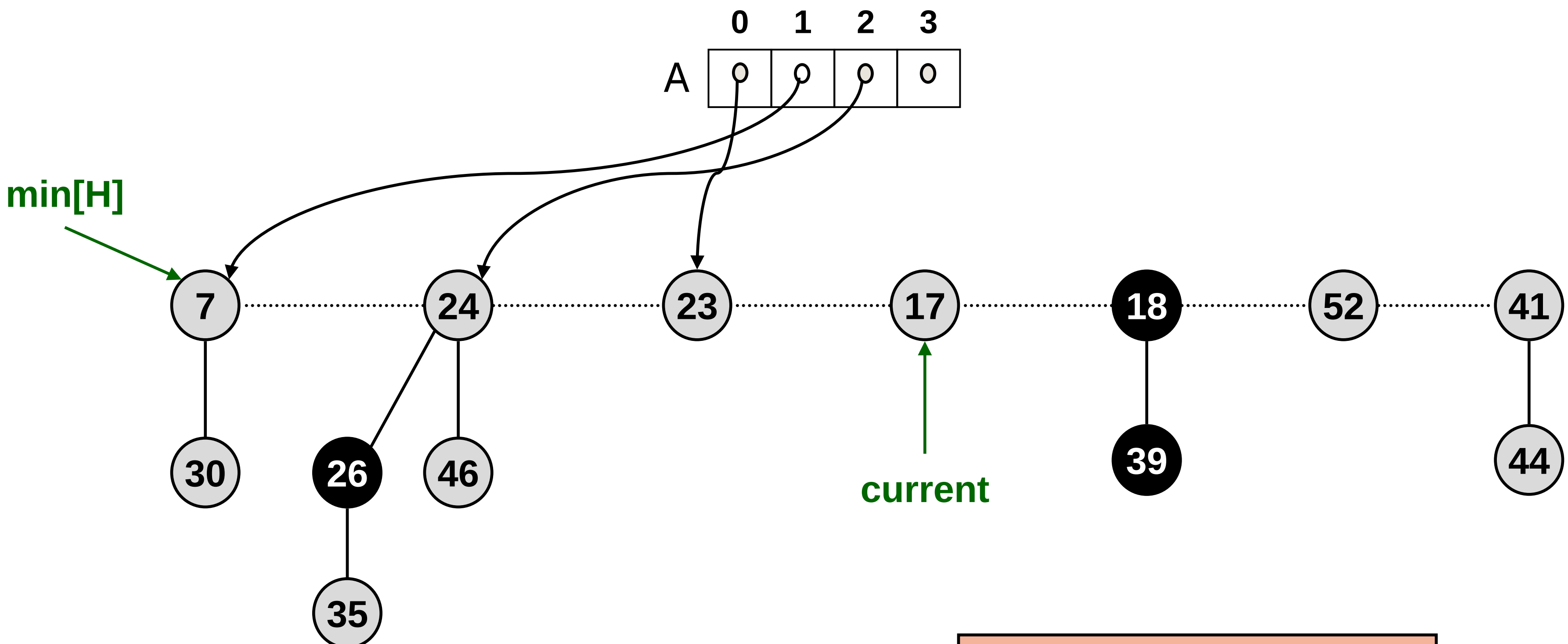
Extracting the Minimum (contd.)



Extracting the Minimum (contd.)

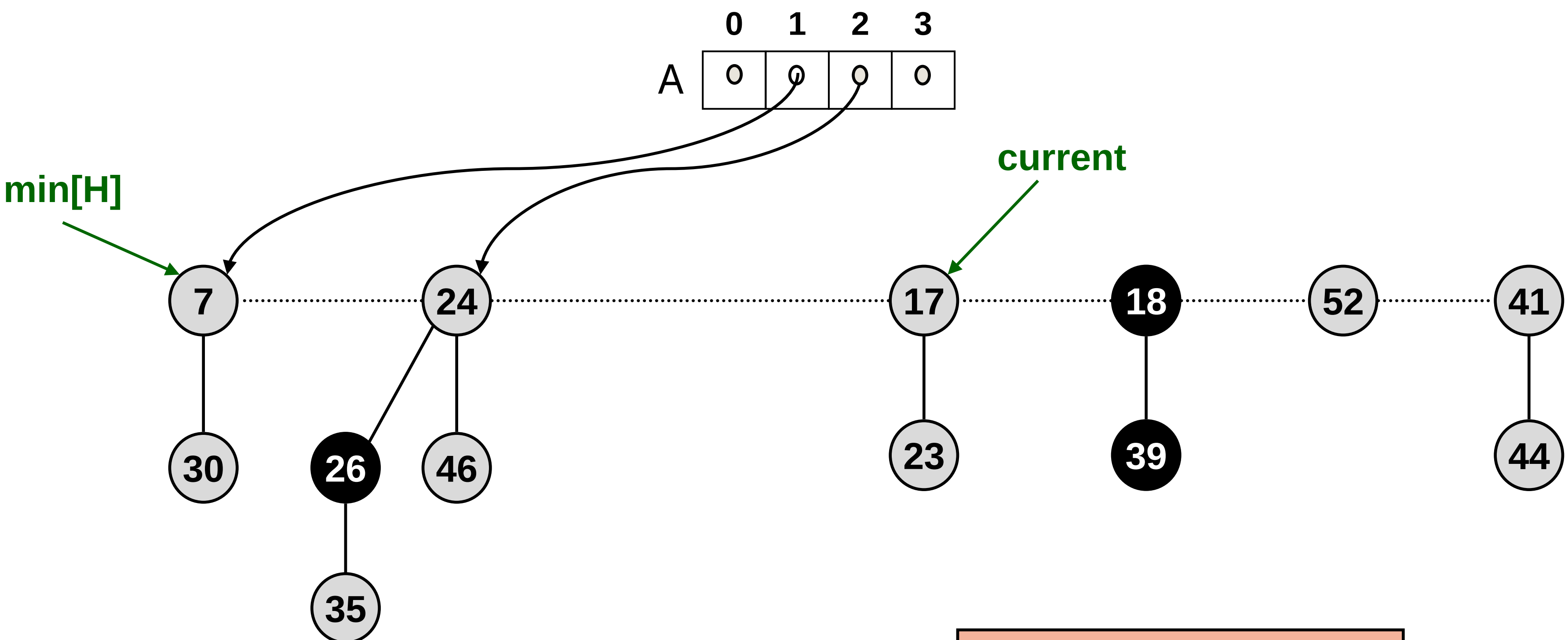


Extracting the Minimum (contd.)



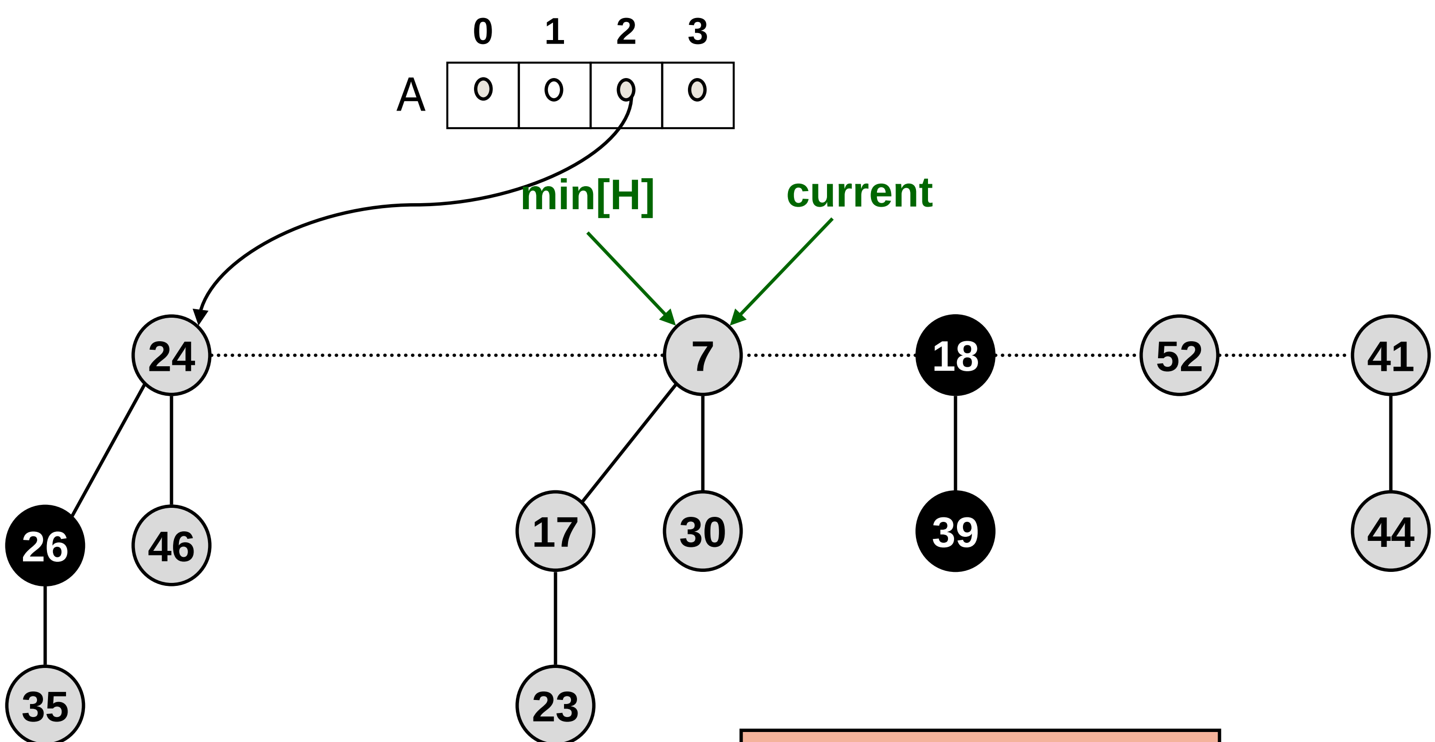
Merge 17 and 23 trees

Extracting the Minimum (contd.)



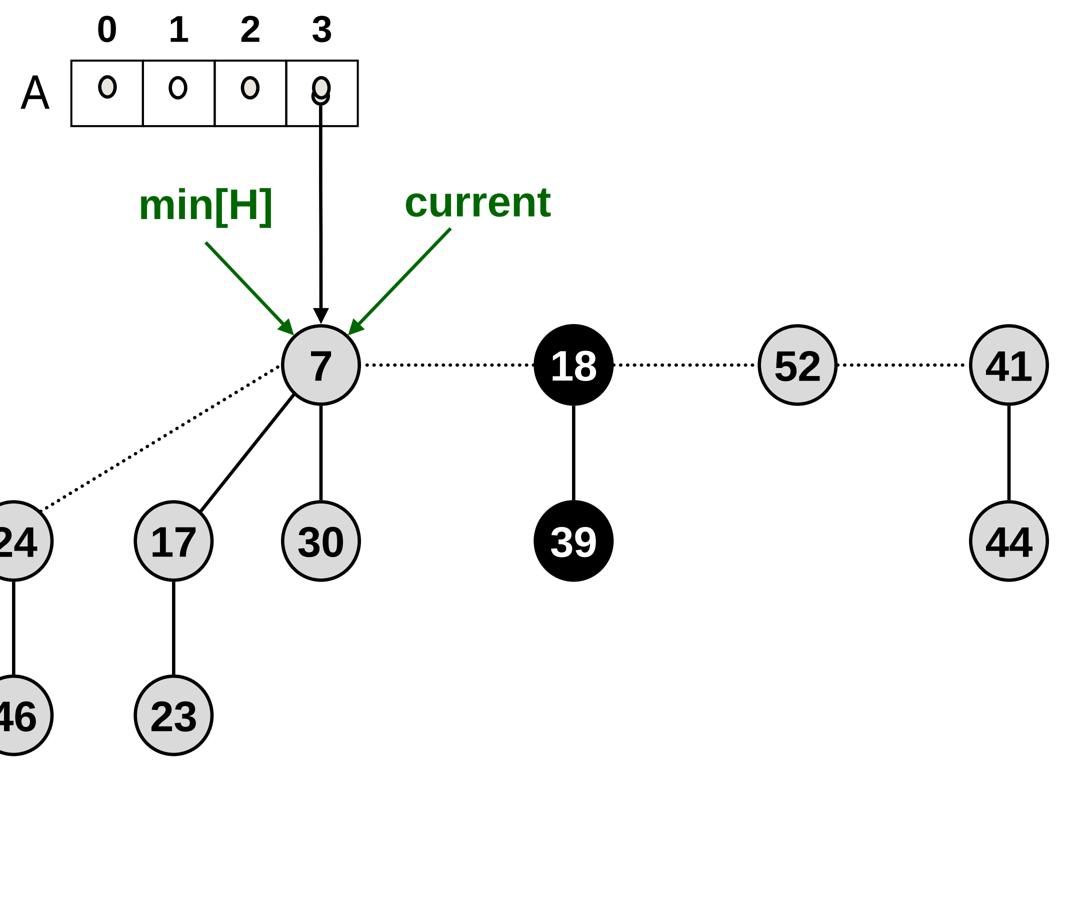
Merge 7 and 17 trees

Extracting the Minimum (contd.)

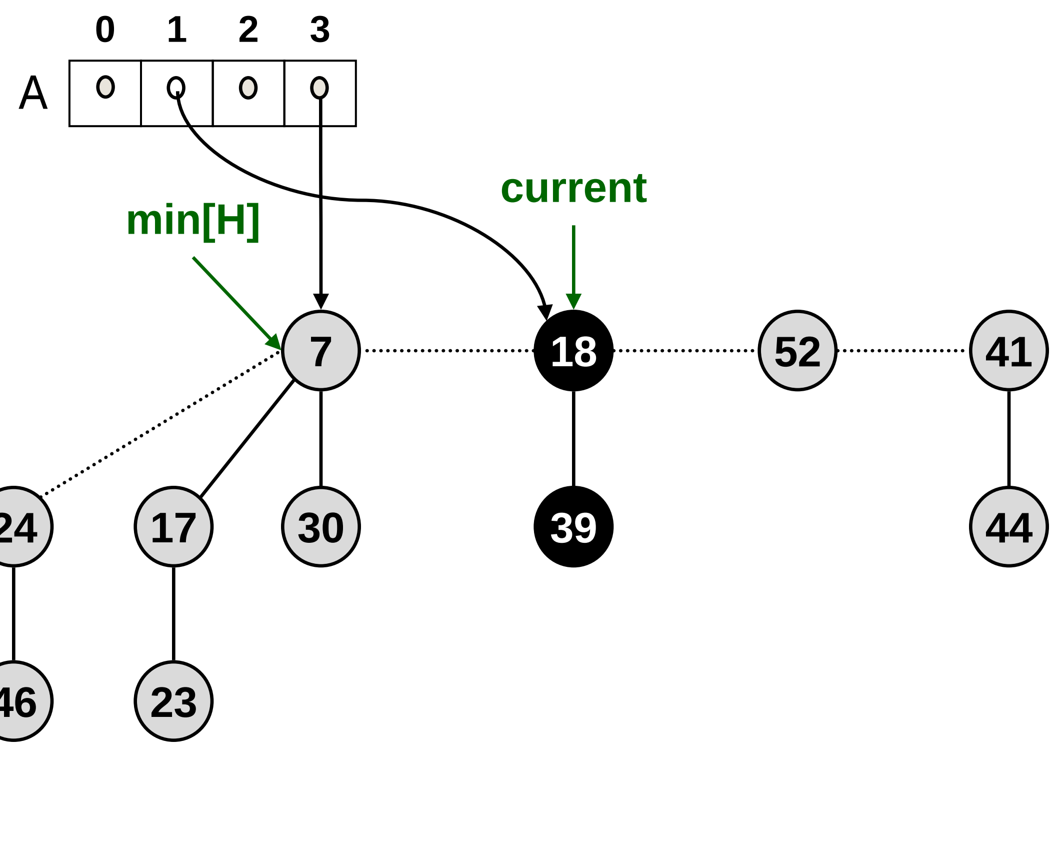


Merge 7 and 24 trees

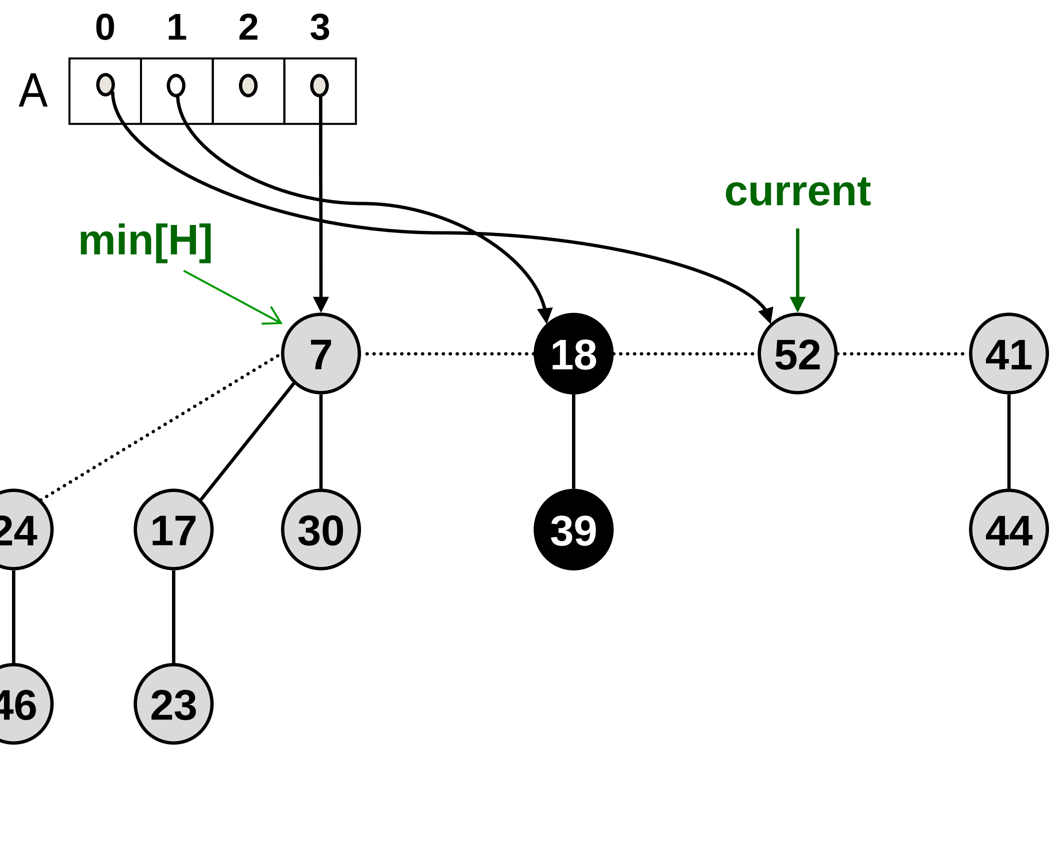
Extracting the Minimum (contd.)



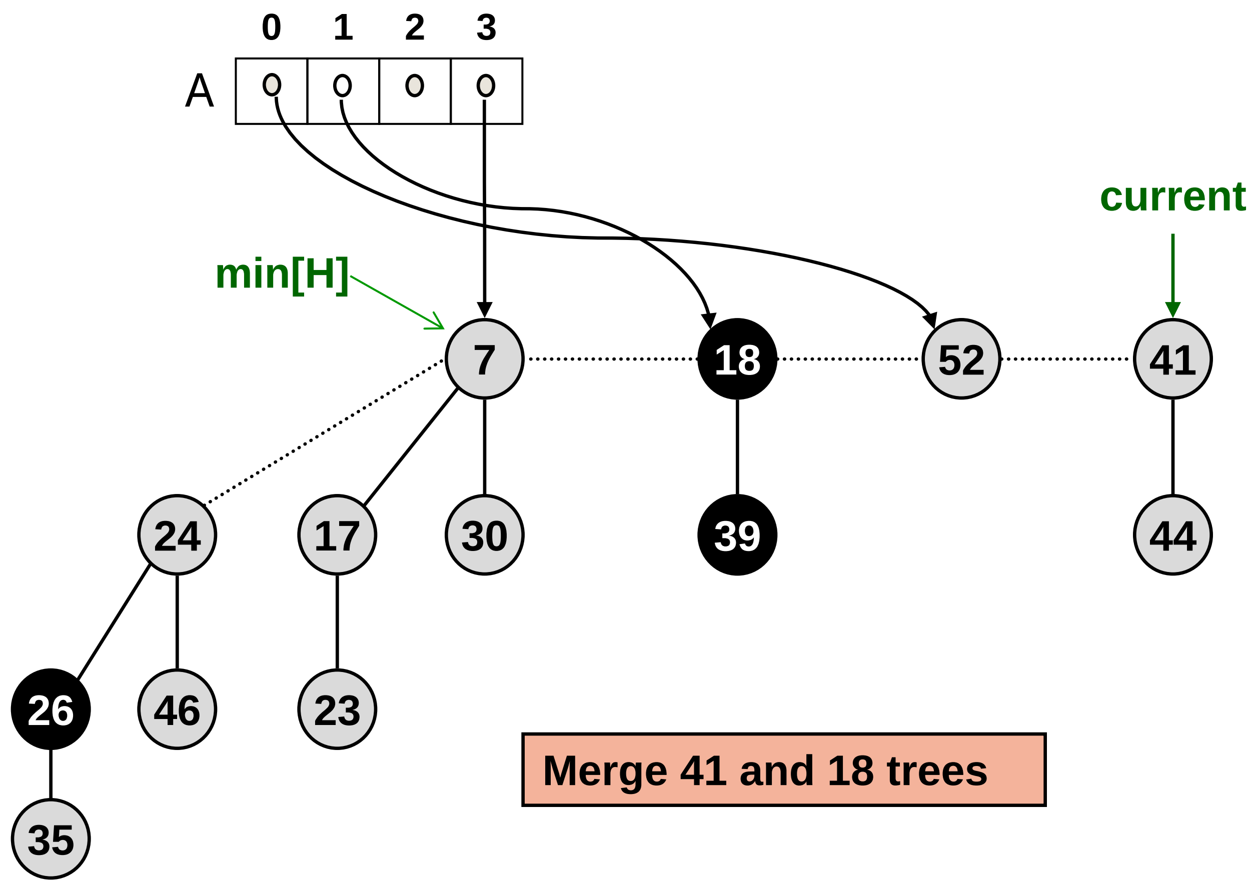
Extracting the Minimum (contd.)



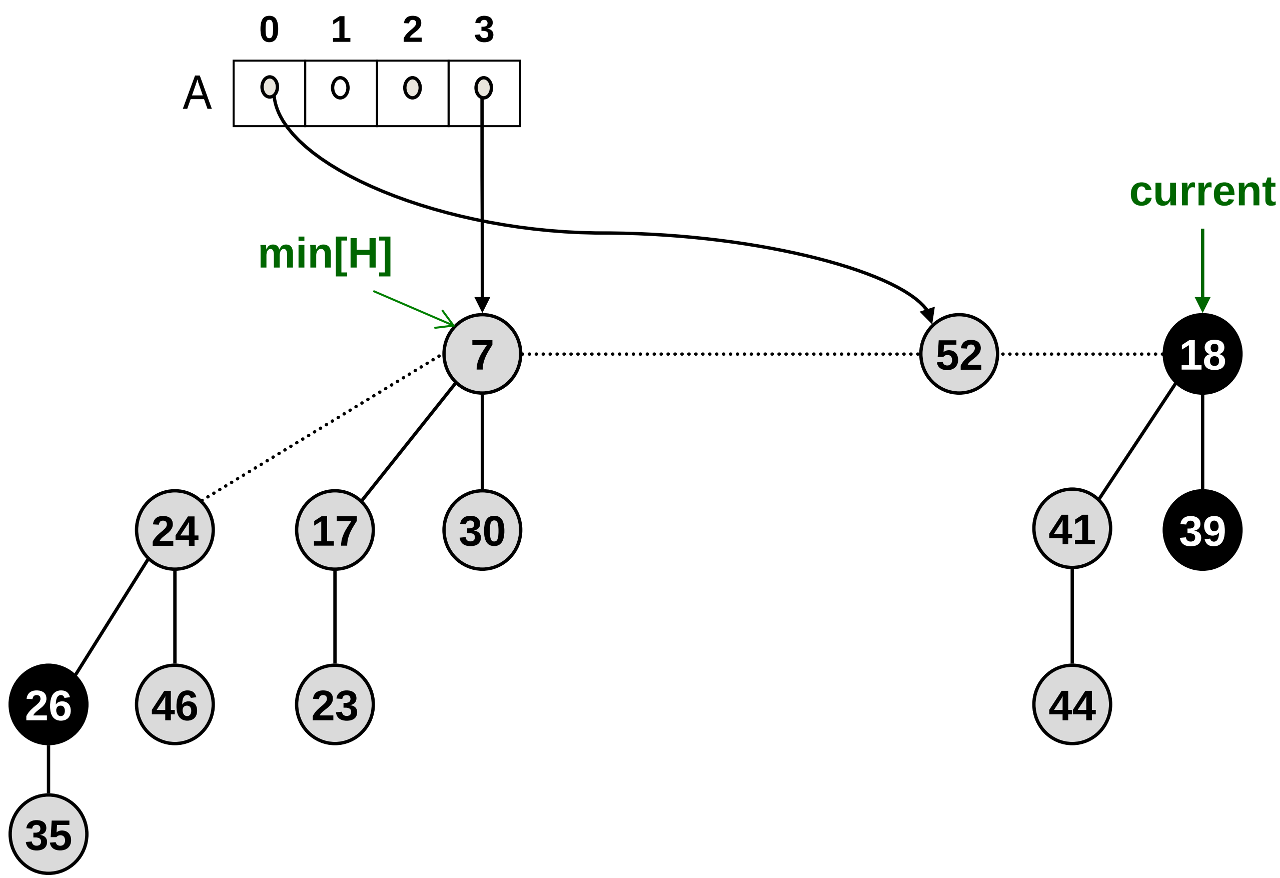
Extracting the Minimum (contd.)



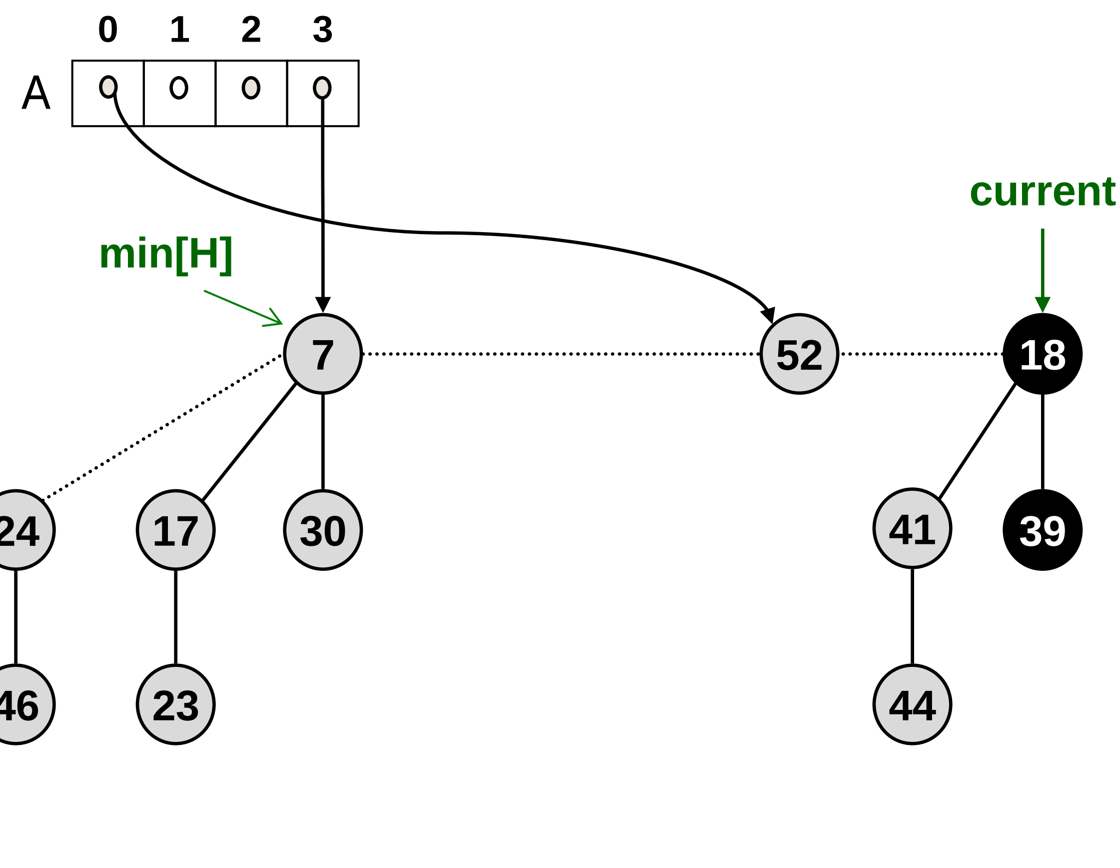
Extracting the Minimum (contd.)



Extracting the Minimum (contd.)

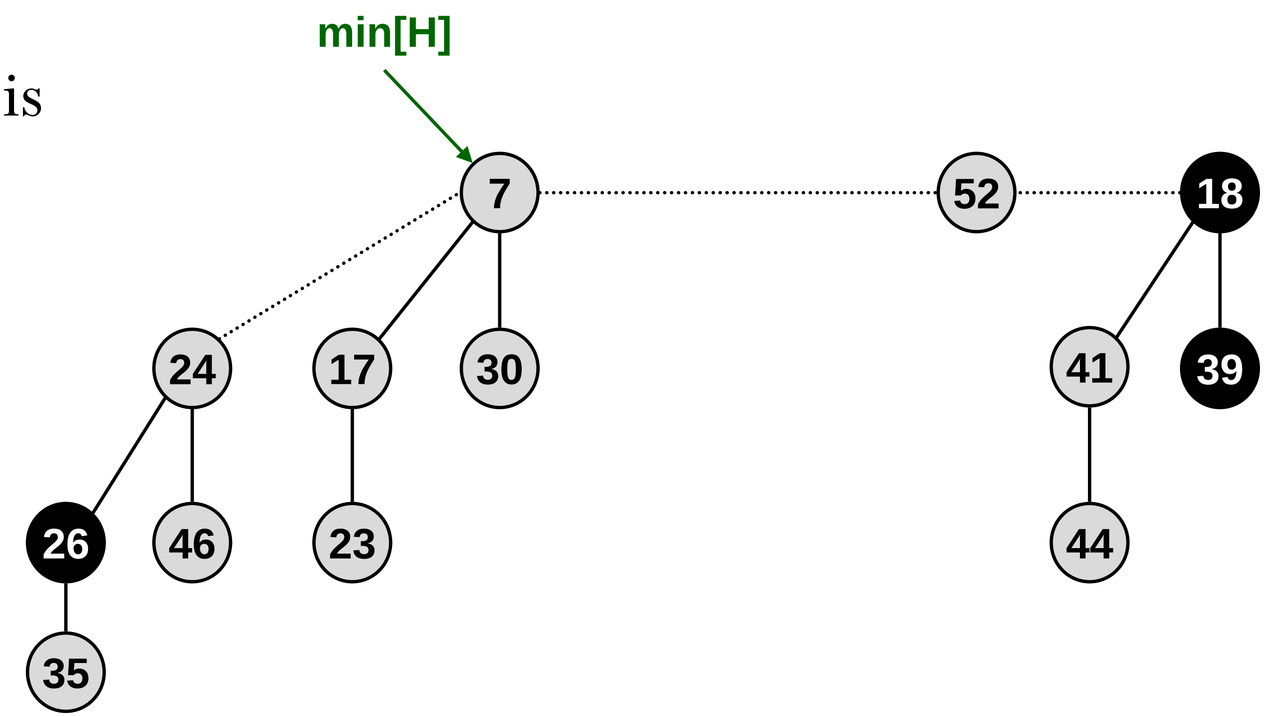


Extracting the Minimum (contd.)



Extracting the Minimum (contd.)

- All roots covered by current pointer, so done
- Now find the minimum among the roots and make min[H] point to it (already pointing to minimum in this example)
- Final heap is



Amortized Cost of Extracting Min

- Recall that
  - $D(n) =$ max degree of any node in the heap with n nodes
  - $t(H) =$ number of trees in heap H
  - $m(H) =$ number of marked nodes in heap H
  - Potential function $\Phi(H) = t(H) + 2m(H)$
- Actual Cost
  - Time for Step 1:
    - $O(D(n))$ work adding min's children into root list

- Time for Step 2 (consolidating trees)
  - Size of root list just before Step 2 is $\leq D(n) + t(H) - 1$
    - $t(H)$ original roots before deletion minus the one deleted plus the number of children of the deleted node
  - The maximum number of merges possible is the no. of nodes in the root list
  - Each merge takes $O(1)$ time
  - So total $O(D(n) + t(H))$ time for consoildation
  - $O(D(n))$ time to find the new min and updating min[H] after consolidation, since at most $D(n) + 1$ nodes in root list
- Total actual cost = time for Step 1 + time for Step 2

$$= O(D(n) + t(H))$$

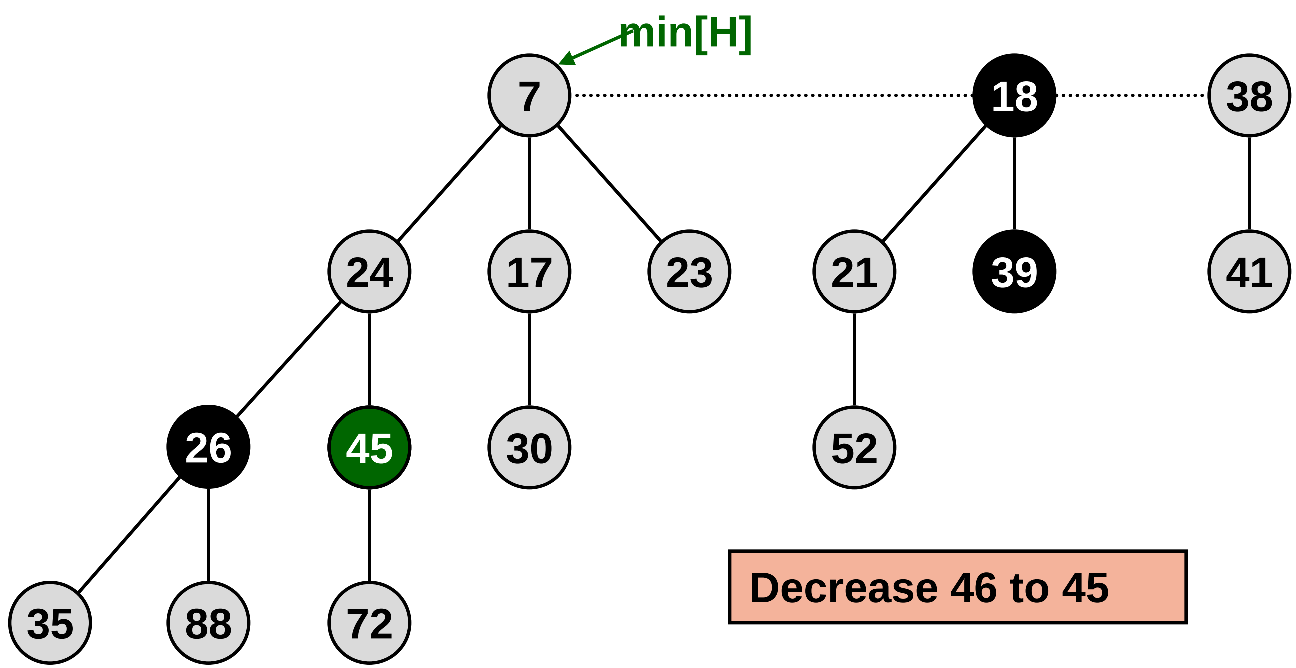
- Potential before extracting minimum $= t(H) + 2m(H)$
- Potential after extracting minimum $\leq (D(n) + 1) + 2m(H)$
  - At most $D(n) + 1$ roots are there after deletion
  - No new node is marked during deletion
    - Can be unmarked, but not marked
- Amortized cost = actual cost + potential change

  $= O(D(n)+ t(H)) + ((D(n)+1) +2m(H)) - (t(H) + 2m(H))$

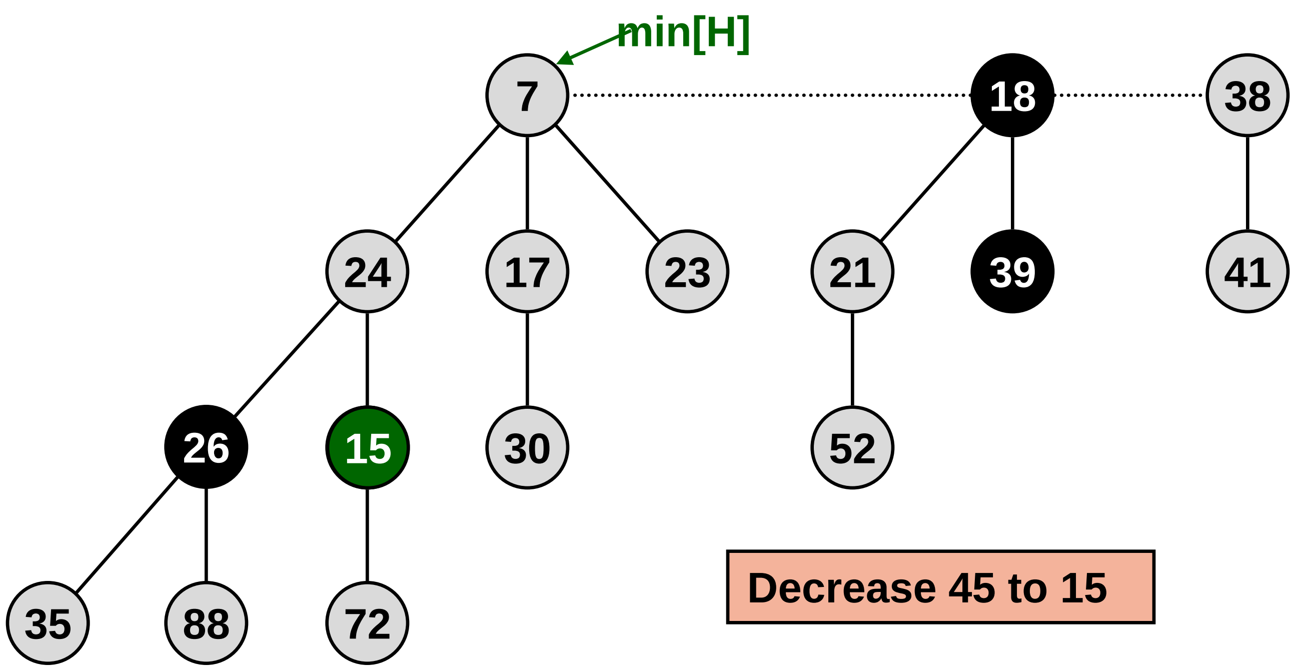
  $= O(D(n))$
- But $D(n)$ can be $O(n)$, right? That seems too costly! So is $O(D(n))$ any good?
  - Can show that $D(n) = O(\lg n)$ (proof omitted)
  - So amortized cost $= O(\lg n)$

Decrease Key

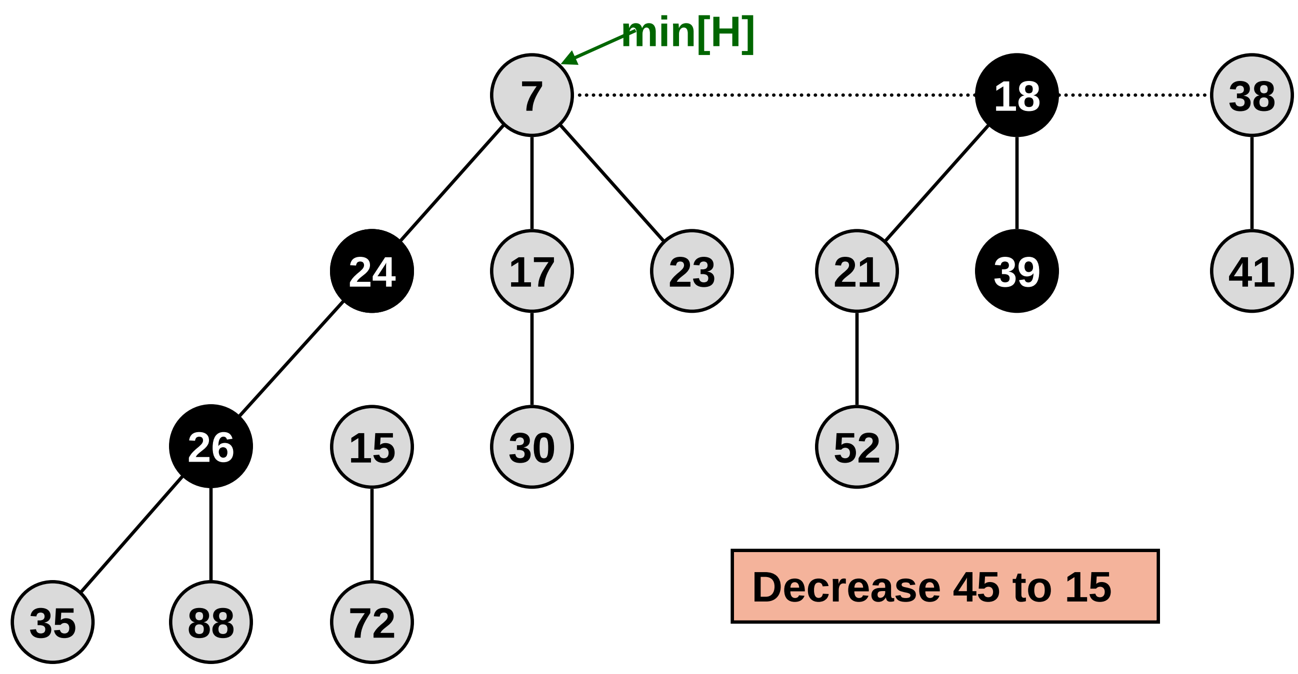
- Decrease key of element x to k
- Case 0: min-heap property not violated
  - decrease key of x to k
  - change heap min pointer if necessary



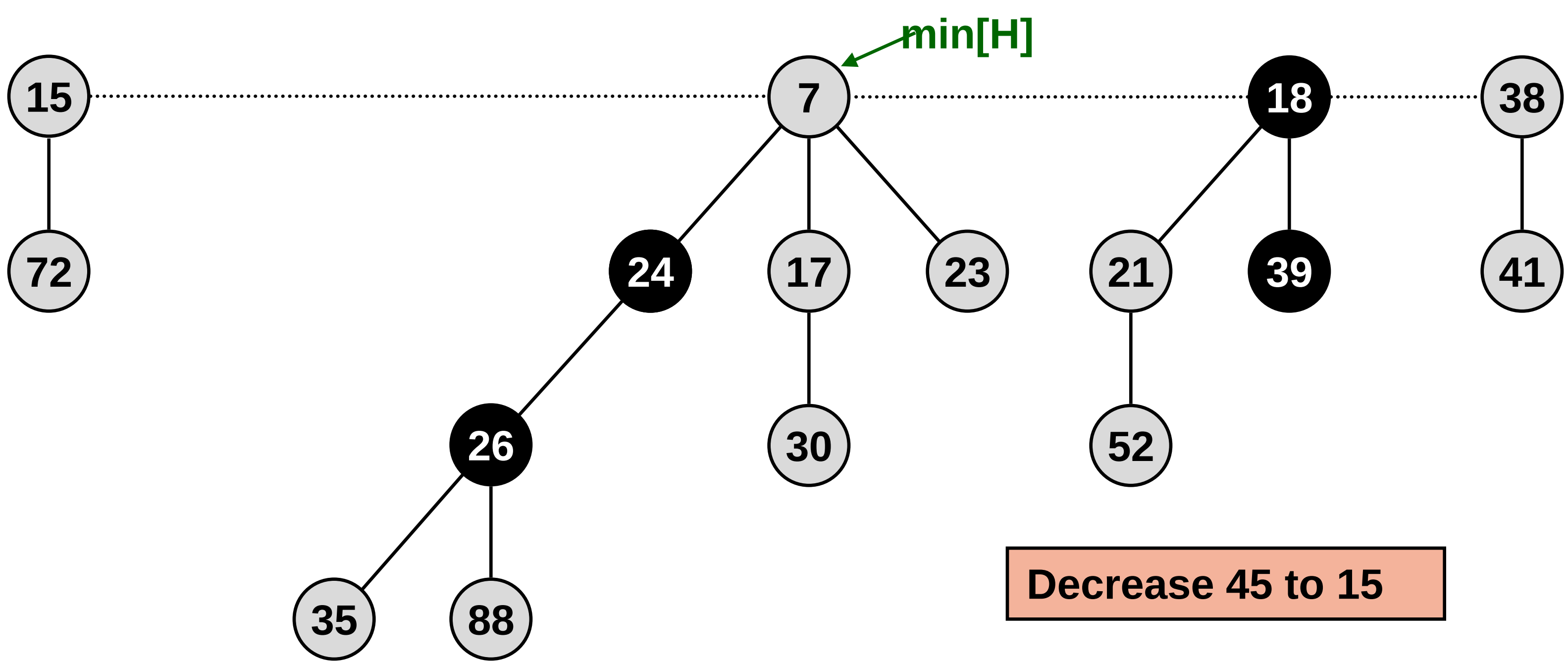
- Case 1: parent of x is unmarked
  - decrease key of x to k
  - cut off link between x and its parent, unmark x if marked
  - mark parent
  - add tree rooted at x to root list, updating heap min pointer



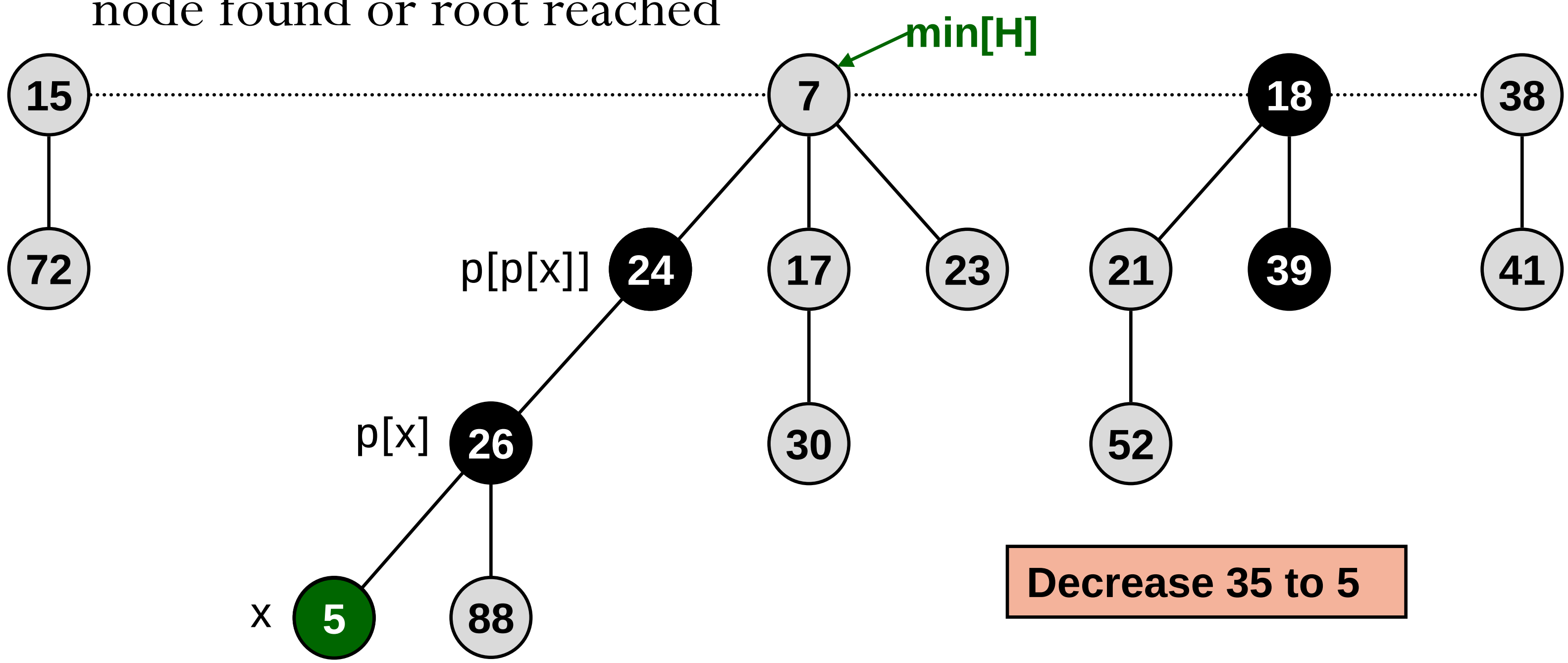
- Case 1: parent of x is unmarked
  - decrease key of x to k
  - cut off link between x and its parent, unmark x if marked
  - mark parent
  - add tree rooted at x to root list, updating heap min pointer



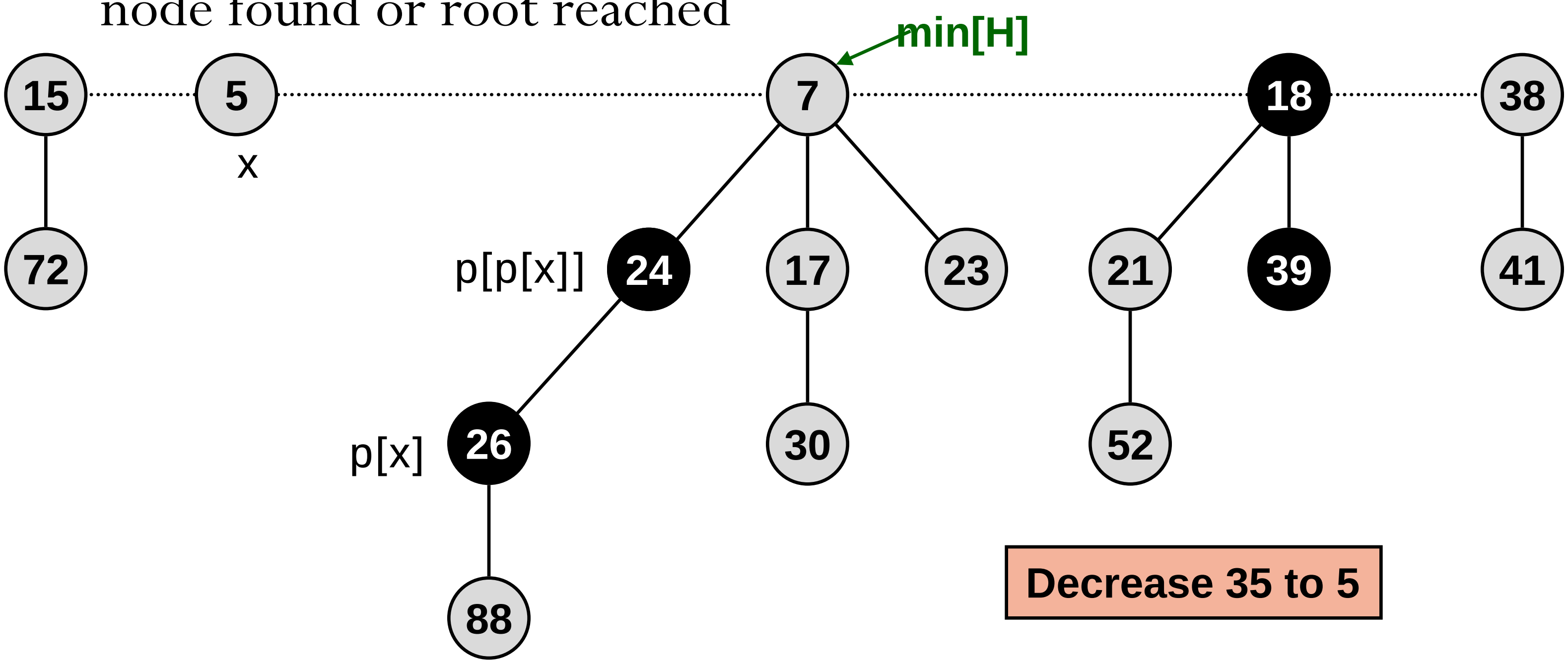
- Case 1: parent of x is unmarked
  - decrease key of x to k
  - cut off link between x and its parent, unmark x if marked
  - mark parent
  - add tree rooted at x to root list, updating heap min pointer



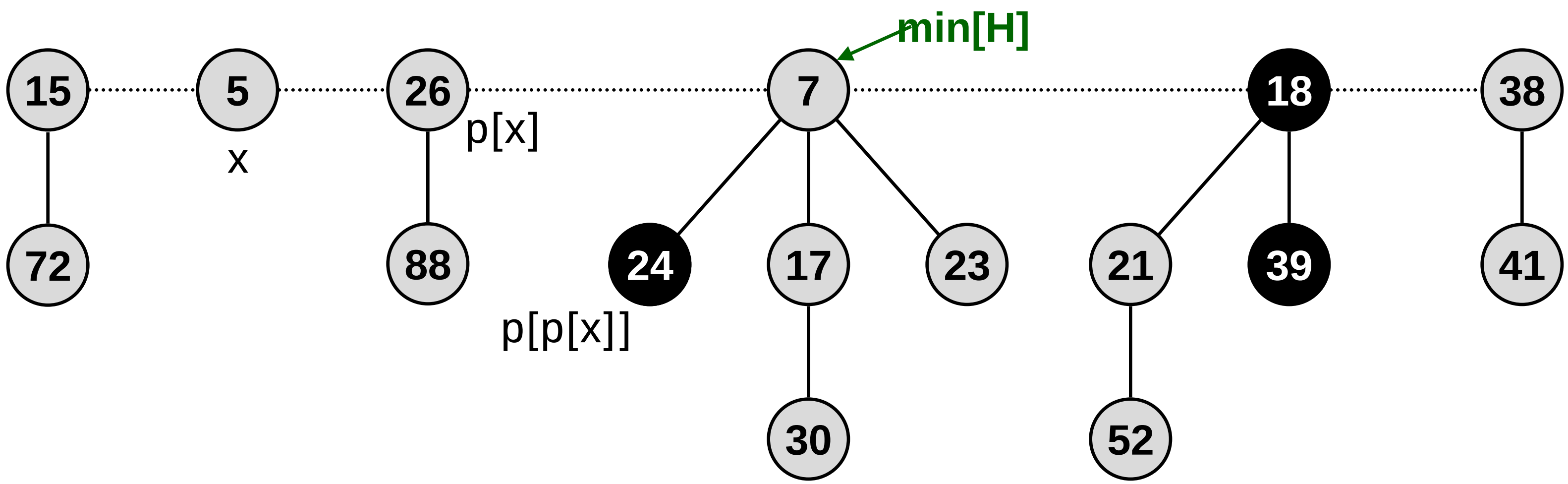
- Case 2: parent of x is marked
  - decrease key of x to k
  - cut off link between x and its parent p[x], add x to root list, unmark x if marked
  - cut off link between p[x] and p[p[x]], add p[x] to root list, unmark p[x] if marked
    - If p[p[x]] unmarked, then mark it and stop
    - If p[p[x]] marked, cut off p[p[x]], unmark, and repeat until unmarked node found or root reached



- Case 2: parent of x is marked
  - decrease key of x to k
  - cut off link between x and its parent p[x], add x to root list, unmark x if marked
  - cut off link between p[x] and p[p[x]], add p[x] to root list, unmark p[x] if marked
    - If p[p[x]] unmarked, then mark it and stop
    - If p[p[x]] marked, cut off p[p[x]], unmark, and repeat until unmarked node found or root reached

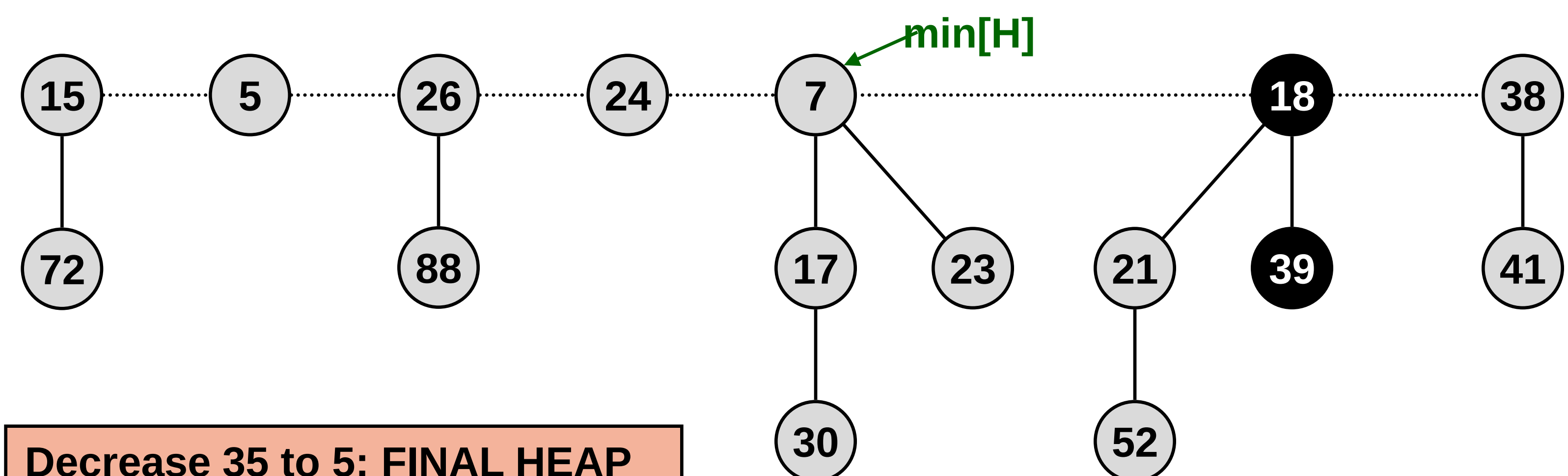


- Case 2: parent of x is marked
  - decrease key of x to k
  - cut off link between x and its parent p[x], add x to root list, unmark x if marked
  - cut off link between p[x] and p[p[x]], add p[x] to root list, unmark p[x] if marked
    - If p[p[x]] unmarked, then mark it and stop
    - If p[p[x]] marked, cut off p[p[x]], unmark, and repeat until unmarked node found or root reached



Decrease 35 to 5

- Case 2: parent of x is marked
  - decrease key of x to k
  - cut off link between x and its parent p[x], add x to root list, unmark x if marked
  - cut off link between p[x] and p[p[x]], add p[x] to root list, unmark p[x] if marked
    - If p[p[x]] unmarked, then mark it and stop
    - If p[p[x]] marked, cut off p[p[x]], unmark, and repeat until unmarked node found or root reached (cascading cut)



Decrease 35 to 5: FINAL HEAP

Fib-Heap-Decrease-key(H, x, k)

```
if k > key[x]
        error "new key is greater than current key"
key[x] = k
y ← p[x]
if y ≠ NIL and key[x] < key[y]
        { CUT(H, x, y)
          CASCADING-CUT(H, y) }
if key[x] < key[min[H]]
        min[H] = x
```

CUT(H, x, y)

1. remove x from the child list of y, decrement 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.        if mark[y] = FALSE
4.              mark[y] = TRUE
5.        else  CUT(H, y, z)
6.              CASCADING-CUT(H, z)
```

Amortized Cost of Decrease Key

- Actual cost
  - O(1) time for decreasing key value, and the first cut of x
  - O(1) time for each of c cascading cuts, plus reinserting in root list
  - Total O(c)
- Change in Potential
  - H = tree just before decreasing key, H' just after
  - $t(H') = t(H) + c$
    - $t(H) + (c-1)$ trees from the cascading cut + the tree rotted at x
  - $m(H') \leq m(H) - c + 2$
    - Each cascading cut unmarks a node except the last one $(-(c-1))$
    - Last cascading cut could potentially mark a node (+1)

- Change in potential

$$= (t(H') + 2m(H')) - (t(H) + 2m(H))$$

$$\leq c + 2(-c + 2) = 4 - c$$

- Amortized cost = actual cost + potential change

$$= O(c) + 4 - c = O(1)$$

Deleting an Element

- Delete node x
  - Decrease key of x to $-\infty$
  - Delete min element in heap

- Amortized cost
  - O(1) for decrease-key.
  - O(D(n)) for delete-min.
  - Total O(D(n))
    - Again, can show that D(n) = O(lg n)
    - So amortized cost of delete = O(lg n)